AN INVESTIGATION
OF
SOME NEW TREE STRUCTURES

By
BRENDA WOODFORD, B.Sc.

A Project Report
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University
August, 1977

© BRENDA WOODFORD 1978
MASTER OF SCIENCE (1977)      McMaster University
Hamilton, Ontario

TITLE: An Investigation of Some New Tree Structures

AUTHOR: Brenda Woodford, B.Sc. (Memorial University)

SUPERVISOR: Professor D. Wood

NUMBER OF PAGES: viii, 36
ABSTRACT

A study of the tree structures developed by Finkel and Bentley (3 & 4) was done and the results are documented in this report. These tree structures, i.e. the quad tree and the k-d tree, were especially developed for associative retrieval. A comparison of the above tree structures and the well known binary search tree is presented for exact match queries.

An implementation of the insertion algorithms for each tree structure and a generalization of Aldon Walker's (9) display algorithm are given.
ACKNOWLEDGEMENTS

I would like to thank Professor D. Wood for his supervision and suggestion of this project.

A special thanks goes to all the friends I made at McMaster who made my stay there enjoyable.

Next, I would like to thank my husband, Paul Kennedy, for his patience and help doing the dishes throughout the prolonged writing of this report.

Finally, my thanks go to Helen Kennelly for her superb typing and Melissa O’Malley for the finishing touches.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER I:</th>
<th>INTRODUCTION</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER II:</td>
<td>DEFINITIONS AND TERMINOLOGY</td>
<td>5</td>
</tr>
<tr>
<td>CHAPTER III:</td>
<td>ALGORITHMS AND IMPLEMENTATION</td>
<td></td>
</tr>
<tr>
<td>3.1 Overview</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2 Retrieval Algorithms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.1 General</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>3.2.2 Basic Binary Search Tree Retrieval Algorithm</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>3.2.3 K-Dimensional Search Tree Retrieval Algorithm</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>3.2.4 K-D Tree Retrieval Algorithm</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>3.3 Insertion Algorithms</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>3.4 Implementation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4.1 General</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>3.4.2 Basic Binary Tree Insertion Algorithm</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3.4.3 K-Dimensional Search Tree Insertion Algorithm</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>3.4.4 K-D Insertion Algorithm</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>CHAPTER IV:</td>
<td>RESULTS AND DISCUSSION</td>
<td></td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>4.2 Building Time</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>4.3 Internal Path Length</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>4.4 Height</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>4.5 Concluding Remarks</td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>APPENDIX A:</td>
<td>EXAMPLE OUTPUTS AND LISTINGS OF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>THE DISPLAY ROUTINES</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX B: LISTINGS OF THE TREE ALGORITHMS
APPENDIX C: THE COMPARE ROUTINE
LIST OF FIGURES

Figure 2.1  A binary tree of 11 nodes  6
Figure 2.2  A complete 2-dimensional tree  9
Figure 2.3  A perfect 2-dimensional search tree  13
Figure 2.4  A 2-d tree  15
Figure 4.2  Building time vs n  26
Figure 4.3.1  Internal path length vs nlog(n) for the k-dimensional search tree where k = 2, 3, 4  29
Figure 4.3.2  Internal path length vs nlog(n) for the 3-d tree and 3-dimensional search tree  30
Figure 4.4.1  Height vs log n for the k-dimensional search tree where k = 2, 3, 4  32
Figure 4.4.2  Height vs log n for the 3-d tree and 3-dimensional search tree  33
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.2.1</td>
<td>Building time data</td>
<td>25</td>
</tr>
<tr>
<td>Table 4.3.1</td>
<td>Internal path length data</td>
<td>28</td>
</tr>
<tr>
<td>Table 4.4.1</td>
<td>Height data</td>
<td>31</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

Data structures for retrieval of a record in a file using primary keys i.e. keys which uniquely define a record, have been well studied. We know from Knuth [6] that binary trees have proven to be a good structure for representing linearly ordered data and that balanced binary trees are efficient for fast retrieval.

As yet, such an ideal data structure for associative retrieval i.e. data retrieval dependant on the values of more than one attribute or key, hasn't been developed. It is a more complicated problem mainly because the structure has to be capable of answering many different kinds of queries efficiently. A query is a retrieval request of a file and it specifies a number of conditions which are to be satisfied by the attributes of the records in the file. According to Knuth [7] and Bentley [3] queries are usually of the following types:

I. Intersection queries consisting of:
   A. Simple query or exact match query -

requests the retrieval of a specific record in a file.
B. Partial match query -
specifies values for some of the attributes, and the most
general type of intersection query which includes A and B.

C. Boolean query -
assuming some 'less than' ordering on the attributes and
using the Boolean operators AND, OR, and NOT, we can
specify a Boolean function on ranges of values for some
or all of the attributes.

For example, consider the case of a file of employee records
having several attributes in the following order (name, age,
job classification, employee #). If we ask for all records
with the following values for the attributes:

1.) name = Smith.
    age = 28
    job classification = 10
    employee # = 212.

This is an example of an exact match query since we are re-
questing a specific record.

2.) age = 25

This is an example of a partial match query since we are in-
terested in only those records with the age attribute = 25.

3.) 21 < age < 25 and job classification = 12.

This is an example of a Boolean query.

We can think of the attributes as components of a vector,
that is, the records are points in a vector space.
II. Near neighbor query can be broken down into:

A. Nearest neighbor query - a request to retrieve the nearest neighbor in the set which is "closest" to a given point.

B. Fixed radius near neighbor query - a request to retrieve all points within a fixed "distance" of a given point.

Many data structures have been developed for building information retrieval systems to deal with these different associative queries. A few of these structures are discussed in great detail by Knuth [7]. For example, the inverted file which is one of the most important of the current techniques, compounded and binary attributes, superimposed coding, and combinatorial hashing. McCreight [8] proposes that a "super-key" of the attributes be formed and then linear retrieval algorithms be used. These and other techniques are discussed by Bentley [3].

The first general approach to use a tree structure in associative retrieval was introduced by Finkel and Bentley [4]. They considered records arranged in k-dimensional space with one dimension for each attribute and arrived at a generalization of the binary tree called a quad tree. Instead of each node having at most 2 sons, the nodes in a quad tree of k dimensions have at most \(2^k\) sons. Thus the binary tree is a special case of the quad tree of k dimensions where \(k = 1\).
In choosing a particular data structure certain criteria must be kept in mind, such as low storage requirements, efficient deletion techniques and the ability to efficiently satisfy retrieval requests from any of the possible queries. Recently a new tree structure has been proposed by Bentley [3] called the multi-dimensional search tree or k-d tree.

The k-d tree has been tested for all the different queries and has been shown to perform better than or just as well as the other techniques. This tree structure is a generalization of the binary search tree where each record containing k attributes is stored as a node. There is a discriminator between 0 and k-1 associated with each node that specifies the attribute in the record which is to be used to determine which of the subtrees to follow.

The objectives of the project are:

(1) to implement the basic insertion algorithms for the binary search tree, the quad tree of k dimensions and the k-d tree,

(2) to compare the building time, internal path length and height of each of the above tree structures by inserting the same set of records into each. Therefore, only exact match queries will be investigated.
Definition

Given m a positive integer, the empty tree $T_0$ of zero nodes is an $m$-ary tree. An $m$-ary tree, $T_n$ of $n > 1$ nodes is an ordered $m+1$ tuple $(T_1, \ldots, T_m, v)$ where $T_i, \ldots, T_m$ are $m$-ary trees of $i_j$ nodes respectively, $i_j \geq 0$, $1 \leq j \leq m$, $\sum_{j=1}^{m} i_j = n-1$, and $v$ is a single node called the root of $T_n$. The trees $T_1, \ldots, T_m$ are called the subtrees of the root $v$. In particular when $m=2$ we have a binary tree and we write $(T_l, v, T_r)$ in place of $(T_1, T_2, v)$, $T_l$ and $T_r$ denoting the left and right subtrees of $v$, respectively.

Definition

Given an $m$-ary tree, $T_n$, of $n > 1$ nodes, we define the level of a node $u$ in $T_n$ to be

$$\text{level}(u, T_n) = \begin{cases} 0 & \text{if } u = v \\ l + \text{level}(u, T_{ij}) & \text{where } u \text{ is in } T_{ij} \end{cases}$$

The height of an $m$-ary tree $T_n$ of $n > 1$ nodes is the maximum level of any node in $T_n$.

Refer to Figure 2.1 for an example of a binary tree, its height and level.
A binary tree of 11 nodes

Figure 2.1
The null sons are expressed by square boxes and are known as external nodes. The internal nodes are represented by circles.

**Definition**

The internal path length, $|T_n|_I$, of an m-ary tree $T_n$ is zero if $n \leq 1$, otherwise it is given by

$$|T_n|_I = \sum_{j=1}^{m} |T_{i_j}|_I + n - 1.$$

Similarly, the external path length, $|T_n|_E$, of an m-ary tree $T_n$ is zero if $n < 1$, otherwise it is given by

$$|T_n|_E = \begin{cases} m, & n = 1 \\ n + \sum_{j=1}^{n} |T_{i_j}|_E + n + 1, & n > 1. \end{cases}$$

For example, referring to the binary tree in Figure 2.1

$|T_{11}|_I = 23$ and $|T_{11}|_E = 45$.

The following theorem is given in Knuth [6].

**Theorem**

Given an m-ary tree $T_n$ with $n > 1$ nodes the internal path length $|T_n|_I$ and the external path length $|T_n|_E$ are related by the formula

$$|T_n|_E = (m - 1) |T_n|_I + mn.$$

Knuth gives a proof by induction on page 400 for $m = 2$. The same proof holds for the general formula.
The maximum path length among all m-ary trees with n nodes is attained by the degenerate tree with a linear structure. The maximum external and internal path lengths possible for an m-ary tree with n nodes are

\[
\left| T_n \right|_{E_{\text{max}}} = mn + \sum_{i=1}^{n-1} (m-1)(n-i) = \frac{(m-1)n^2 + (m+1)n}{2}
\]

and

\[
\left| T_n \right|_{I_{\text{max}}} = \frac{n(n-1)}{2}
\]

Correspondingly, the minimum path lengths of an m-ary tree occur when the nodes are nearest the root. Therefore, the minimum external and internal path lengths among all m-ary trees with n nodes are respectively

\[
\left| T_n \right|_{E_{\text{min}}} = ((m-1)n+1)q - \frac{(m^{q+1}-m)}{m-1} + mn,
\]

and

\[
\left| T_n \right|_{I_{\text{min}}} = (n + \frac{1}{m-1})q - \frac{(m^{q+1}-m)}{(m-1)^2}
\]

where \( q = \lfloor \log_m ((m-1)n+1) \rfloor \) and \( \lfloor \rfloor \) means integer part.

**Definition**

A \textit{k-dimensional tree}, \( T^k_n \), (Finkel and Bentley's quad tree [4]) of \( n \geq 1 \) nodes is an m-ary tree where \( m = 2^k \). Note that when \( k = 1 \) we have a binary tree.
A complete 2-dimensional tree

Figure 2.2

Definition

A k-dimensional tree, $T_n^k$ of $n > 1$ nodes with $t$ levels is said to be complete if and only if it has $(2^k)^i$ nodes on every level $i$, $0 \leq i \leq t-1$ (where the root is defined to be a level zero).

This reduces to the notion of completeness for binary trees when $k = 1$. These concepts are illustrated in Figure 2.2.
So far we have been discussing tree structures in abstract terms. Now we go on to investigate how these tree structures are utilized for storing and retrieving information. Preliminary definitions of the information which is to be stored and retrieved follow.

**Definition**

For \( k > 0 \), a \textit{k-tuple key} is a vector of \( k \) attributes for an item of information. When \( k = 1 \), it is just referred to as a key.

The set of all possible attributes of the \( k \)-tuple keys have some transitive relation \( \prec \) defined on it. If the set of attributes is a subset of the integers then \( \prec \) is the usual "less than" relation. Let us define a relation \( \prec \) read "\( i \)-less than", between two \( k \)-tuple keys \( K_1 \) and \( K_2 \).

**Definition**

Given two \( k \)-tuple keys \( K_1 \) and \( K_2 \) where

\[
K_1 = (h_1^1, h_2^1, \ldots, h_k^1),
\]

and

\[
K_2 = (h_1^2, h_2^2, \ldots, h_k^2)
\]

we say

\[
K_1 \prec_i K_2, \quad 1 \leq i \leq k
\]

if and only if \( h_i^1 < h_i^2 \). Note that when \( k = 1 \), \( i \) can only be 1 so that \( \prec \) is just referred to as \( < \).
Similarly, $K^1 \geq K^2$, $1 \leq i \leq k$ if and only if $h^1_i > h^2_i$ and $\geq$ is referred to as $\geq$.

**Definition**

Given two $k$-tuple keys $K^1$ and $K^2$ as defined above, let us define a transformation $F$ on $K^1$ and $K^2$ such that

$$F(K^1, K^2) = \sum_{i=1}^{k} f(h^1_i, h^2_i) \times 2^{i-1}$$

where

$$f(h^1_i, h^2_i) = \begin{cases} 
0, & \text{if } h^1_i > h^2_i \\
1, & \text{otherwise.} 
\end{cases}$$

Example: Assume $k = 3$ and the attributes are a subset of the integers. Let

$K^1 = (33, 5, 16)$ and $K^2 = (26, 45, 2)$.

Then

$$F(K^1, K^2) = 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 = 2.$$ 

Note that the resultant value of $F$ always lies between 0 and $2^{k-1}$.

Now we can continue to discuss how this information is linked with the tree structure.

**Definition**

A $k$-dimensional search tree, $T^k_n$, is a $k$-dimensional tree having a $k$-tuple key associated with each of its nodes. Moving from a node to one of its sons requires a comparison
of all $k$ pairs of keys. The transformation $F$ defined above is used to compare nodes in a $k$-dimensional search tree.

**Definition**

A $k$-dimensional search tree, $T_{n}^{k}$, $n \geq k$ with $t$ levels is said to be perfect if and only if

1. it is complete
2. every node is in the center of its bounds rectangle — that is, the region in which all descendants of the node must lie.

A perfect 2-dimensional search tree is illustrated in Figure 2.3.

Lastly a definition of Bentley's $k$-d tree [3].

**Definition**

A multi-dimensional binary search tree or $k$-d tree, $T_{n,k}$ of $n > 1$ nodes where $k$ represents the dimensionality of the search space, is a binary tree with a $k$-tuple key associated with each of its nodes. To determine which set of nodes to follow, the $<$ relation is used where $i$ is the discriminator pointing to the attribute to be used and is obtained as a function of the level. This discriminator can be obtained by the same method as used by Bentley [3] i.e.

$$i = l \mod k$$

where $i$ is the discriminator of level $l$ and the level of the root node is defined to be zero.
Graphical representation of records in 2-space
Note that the lines outline the bounds rectangle for each record

A perfect 2-dimensional search tree of the records illustrated above

Figure 2.3
A 2-d tree with respect to records in 2-space is shown in Figure 2.4. It follows that a binary search tree is a k-d tree with k = 1.
Graphical representation of records in 2-space

Records illustrated above in 2-space stored as nodes in a 2-d tree

Note:
1. The square boxes represent null sons
2. The lines drawn in the graph above outline the range of each subtree

Figure 2.4
CHAPTER III
ALGORITHMS AND IMPLEMENTATION

3.1 Overview

The algorithms described in Section 3.2 are the basic search algorithms for:

(a) the basic binary search tree
(b) the k-dimensional search tree
(c) the k-d tree.

The implementation details are given in Section 3.3.

A brief discussion and definition of these search trees and related functions were given in Chapter II. Two categories of retrieval are represented by these search trees i.e. a) represents primary key retrieval, b) and c) represent associative key retrieval.
3.2 Retrieval Algorithms

3.2.1 General

Let \( T_n \) be the address of the root of a tree with \( n \) nodes where each node is a \( k \)-tuple key. Assume we are searching for the node \( R \). If \( R \) exists in the tree, the address of that position is returned. Otherwise \( \phi \) representing an empty tree is returned. The retrieval algorithms are defined recursively as follows:

3.2.2 Basic Binary Search Tree Retrieval Algorithm

\[
\text{binary search} (T_n, R) =
\begin{align*}
1. & \quad \text{If } T_n = \phi \text{ then return } \phi. \\
2. & \quad \text{If } R < T_n \text{ then binary search } (\text{left}(T_n), R). \\
3. & \quad \text{If } R > T_n \text{ then binary search } (\text{right}(T_n), R). \\
4. & \quad \text{Else return } T_n.
\end{align*}
\]

where \( \text{left}(T_n) \) represents the address of the left successor of \( T_n \) and \( \text{right}(T_n) \) represents the right successor of \( T_n \).

Note that the operators \(<\) and \(>\) are the \( <\) and \(>\) operators defined in Chapter II and \( T_n \) and \( R \) are \( k \)-tuple keys with \( k = 1 \).

3.2.3 \(K\)-Dimensional Search Tree Retrieval Algorithm

\[
\text{k-dimsearch} (T_n, R) =
\begin{align*}
1. & \quad \text{If } T_n = \phi \text{ then return } \phi. \\
2. & \quad \text{If } T_n = R \text{ then return } T_n. \\
3. & \quad \text{Else k-dimsearch } (\text{son}_i(T_n), R)
\end{align*}
\]

where \( F(T_n, R) = i \) and \( \text{son}_i(T_n) \) represents the address of the \( i \)-th
successor of $T_n$. The function $F$ is defined in Chapter II.

Note that $T_n$ and $R$ are $k$-tuple keys where $k > 1$.

3.2.4 K-D Tree Retrieval Algorithm

Let $\ell$ represent the level in the tree which is zero at the root.

$$k$-d search $(T_n, R, \ell)$

1. If $T_n = \emptyset$ then return $\emptyset$.

2. If $R > T_n$ then $k$-d search($\text{left}(T_n), R, \ell+1$)

3. If $R > T_n$ then $k$-d search($\text{right}(T_n), R, \ell+1$)

4. Else return $T_n$

where $d(\ell) = \ell + 1 \mod k$

$\text{left}(T_n)$ represents the address of the left-successor of $T_n$.

$\text{right}(T_n)$ represents the address of the right-successor of $T_n$.

Note that $k > 1$ and the operators $<$ and $>$ are those defined in Chapter II.

3.3 Insertion Algorithms

The retrieval algorithms given in Section 3.2 with the following modifications could be used to insert a node $R$:

1. If $T_n = \emptyset$ then $R$ is inserted and $T_n$ is set to the address of $R$.

2. The address of each successor of $R$ is set to $\emptyset$. 
3.4 Implementation

3.4.1 General

The basic insertion algorithms for the given trees were implemented in the programming language Pascal as devised by N. Wirth [10]. Pascal's record and pointer facilities enable trees to be built directly. The algorithms were tested using the Pascal 6000 3.4 version available on the CDC 6400.

For the purposes of this project the following implementation details hold for each tree construction algorithm.

1. A tree is constructed by inserting n random records, one at a time. A record consists of a k-tuple key and each attribute of the k-tuple key is an integer between 1 and n. Random permutations are generated on each of the k lists of attributes from 1 to n and then on each of the k-tuple keys, n in number.

The method of Durstenfeld as modified by Pike [11] using the pseudo-random number generator of Pike and Hill [12] is used to generate the records i.e.: k-tuple keys. An example of a record could be (10, 151, 200, 99) where k = 4 and n = 200. A listing of these routines is given in Appendix B.

2. Each node in a tree is represented as a Pascal record.

3. The k-tuple key is not stored at each node but is referenced by an integer between 1 and n. This integer points to the relevant k-tuple key in the list of n keys to be inserted. This method reduces storage requirements considerably since only one word is required instead of
k words at each node. For discussion purposes and ease of explanation, we will assume that the k-tuple key is stored at each node.

(4) The special symbol NIL is used to indicate a null pointer.

(5) The root node of the tree is denoted by ROOT and is preset to NIL. Note that NIL is analogous to $\phi$.

(6) The k-tuple key stored at each node is referenced by the variable name KEY. KEY is a one-dimensional array of length k. For example, if the first attribute of the k-tuple key is to be referenced, it would be denoted by KEY[1].

Implementation details particular to each algorithm follow.

3.4.2 Basic Binary Search Tree Insertion Algorithm

Each node in a binary search tree contains three fields of information.

(1) The k-tuple key referenced by the array KEY.
(2) A pointer to the left successor represented by the variable name LPTR, and
(3) A pointer to the right successor represented by the variable name RPTR.

If a node is a leaf i.e. it has no successors LPTR and RPTR will be NIL.
The function COMPARE was implemented to compare two k-tuple keys, P and Q for instance. These values are returned:

0 if P = Q
1 if P > Q
2 if P < Q

There are many techniques for implementing this comparison routine. The most common technique is to concatenate the k attributes of each k-tuple key and do a straight comparison test. A listing of this technique is given in Appendix C. This method required a long execution time, therefore a simpler technique was implemented. A cyclic comparison is made on each pair of attributes of P and Q starting at the Jth attribute. The integer J is sent as a parameter. If the attributes are equal, J is incremented by 1 and the comparison repeated on the next pair of attributes. Otherwise the appropriate value, 1 or 2 is returned, as defined above. If all attributes are equal, 0 is returned. For the binary search tree insertion, J was set to 1 and the binary tree built on the 1st attribute of each k-tuple key, since each attribute is unique.

3.4.3 K-Dimensional Search Tree Insertion Algorithm

Each node in a k-dimensional search tree contains the following fields of information:

1) the k-tuple key denoted by the array KEY
2) an array PTR containing the address of each successor of the node. Note that there are $2^k$ successors per node.
The transformation $F$ defined in Chapter II was implemented as a function called EXAMINE. This function is used to determine which of the $2^k$ successors to follow while moving down a tree.

### 3.4.4 K-D Tree Insertion Algorithm

Each node in a k-d tree contains the following information:

1) A k-tuple key denoted by the array KEY.

2) Two pointers denoted as follows:
   a) LOSON pointing to the left subtree of a node
   b) HISQN pointing to the right subtree of a node.

3) A discriminator DISC which is an integer between 1 and $k$.
   DISC denotes which of the $k$ attributes to use for comparison while moving down the tree. The discriminator is determined by the function NEXTDISC which is identical to the function $d$ used in Section 3.2.3.

A function called SUCCESSOR is used to determine which successor to follow. It returns either LOSON or HISON. If the discriminating attributes are equal, the nodes are sent to the function COMPARE, as described in Section 3.3.2 and DISC + 1 is sent as the value of $J$.

A listing of the above functions and insertion algorithms is given in Appendix B. Examples of the three search trees are given in Appendix A.
CHAPTER IV
RESULTS AND DISCUSSION

4.1 Introduction

The results are presented and discussed in the following three sections corresponding to the properties investigated:

1) building time
2) internal path length
3) height.

Data was collected for each statistic on the following trees for \( k = 2, 3 \) and 4:

a) binary search tree
b) \( k \)-dimensional search tree.
c) \( k \)-d tree.

For this purpose, 200 trees of each type were built of size \( n \) and \( n \) varied from 50 to 1000 in increments of 50. The results were obtained by averaging over the 200 trees built for each tree type and size.

The \( k \)-tuple keys used to build the trees were generated as specified in Section 3.1.1 i.e. permutations of the ordered sequence 1 to \( n \) for each of the \( k \) attributes. A permutation was also done on the \( n \) \( k \)-tuple keys.

Graphical representations of the results are given and discussed.
4.2 Building Time

The building time for each tree was taken as the sum of the elapsed time intervals in seconds for each k-tuple key insertion.

Table 4.2.1 illustrates that there is no appreciable difference between the tree types investigated with respect to building time. This would indicate that on the average an exact match query would take approximately the same length of time using either tree algorithm.

For the reason stated above only one graph (see Figure 4.2) is given for the building time vs n (where n is the number of nodes in a tree). The building time values used to draw the graph were those collected for the 3-dimensional search tree. As drawn in the graph shows that building time is directly proportional to the size n.

4.3 Internal Path Length

The internal path length was calculated using the definition given in Chapter II. As discussed in Chapter II, the k-d tree is basically a binary search tree with a discriminator for each k-tuple key. One would expect the internal path length to be the same for these trees and independent of k. This is confirmed by the data given in Table 4.3.1. Therefore any graphs given for k-d trees apply to the corresponding binary trees as well.

By definition of the k-dimensional tree, each node
### BUILDING TIME DATA

#### A) BINARY SEARCH TREES

<table>
<thead>
<tr>
<th>n</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>290.64</td>
<td>298.81</td>
<td>275.52</td>
</tr>
<tr>
<td>250</td>
<td>1586.74</td>
<td>1595.02</td>
<td>1579.67</td>
</tr>
<tr>
<td>500</td>
<td>3283.29</td>
<td>3294.01</td>
<td>3300.89</td>
</tr>
<tr>
<td>1000</td>
<td>6818.88</td>
<td>6820.35</td>
<td>6840.01</td>
</tr>
</tbody>
</table>

#### B) K-DIMENSIONAL SEARCH TREES

<table>
<thead>
<tr>
<th>n</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>298.86</td>
<td>306.03</td>
<td>304.16</td>
</tr>
<tr>
<td>250</td>
<td>1624.94</td>
<td>1615.84</td>
<td>1637.62</td>
</tr>
<tr>
<td>500</td>
<td>3363.77</td>
<td>3334.55</td>
<td>3345.67</td>
</tr>
<tr>
<td>1000</td>
<td>6932.13</td>
<td>6842.86</td>
<td>6887.26</td>
</tr>
</tbody>
</table>

#### C) K-D TREES

<table>
<thead>
<tr>
<th>n</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>303.86</td>
<td>307.98</td>
<td>308.73</td>
</tr>
<tr>
<td>250</td>
<td>1649.78</td>
<td>1642.99</td>
<td>1649.96</td>
</tr>
<tr>
<td>500</td>
<td>3425.91</td>
<td>3410.33</td>
<td>3427.82</td>
</tr>
<tr>
<td>1000</td>
<td>7102.24</td>
<td>7096.17</td>
<td>7116.65</td>
</tr>
</tbody>
</table>

**TABLE 4.2.1**
FIGURE 4.2: Building time in seconds vs n, the size of the trees.
Figure 4.2
has up to $2^k$ sons. Therefore the internal path length would be expected to decrease as $k$ increases. Figure 4.3.1 shows that this is the case. For the same reason Figure 4.3.2 shows that the $k$-dimensional tree has shorter internal path length than the $k$-d tree.

These figures also show that internal path length is proportional to $n \log n$ irrespective of the tree insertion algorithm used. This implies that an exact match query in either tree type should be $O(\log n)$.

4.4 Height

For each tree built, height was taken to be the maximum level of any node in that tree. Since height and internal path length are closely related, it was expected there would be little difference between the binary search trees and $k$-d trees. The data in Table 4.4.1 confirmed this.

Height behaves in the same manner as the internal path length as illustrated by Figures 4.4.1 and 4.4.2. Figure 4.4.1 shows that for $k$-dimensional search trees the height decreases as $k$ increases. Figure 4.4.2 shows that the $k$-dimensional search tree has lower height than the $k$-d tree.

Both figures illustrate that height is proportional to $\log n$. 
INTERNAL PATH LENGTH DATA

A) BINARY SEARCH TREES

<table>
<thead>
<tr>
<th>n</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>258.67</td>
<td>255.93</td>
<td>261.13</td>
</tr>
<tr>
<td>250</td>
<td>2057.26</td>
<td>2042.97</td>
<td>2072.63</td>
</tr>
<tr>
<td>500</td>
<td>4795.70</td>
<td>4771.19</td>
<td>4829.12</td>
</tr>
<tr>
<td>1000</td>
<td>10968.26</td>
<td>10905.79</td>
<td>11031.95</td>
</tr>
</tbody>
</table>

B) K-D TREES

<table>
<thead>
<tr>
<th>n</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>258.15</td>
<td>256.39</td>
<td>259.93</td>
</tr>
<tr>
<td>250</td>
<td>2052.56</td>
<td>2050.09</td>
<td>2065.88</td>
</tr>
<tr>
<td>500</td>
<td>4777.80</td>
<td>4778.46</td>
<td>4819.11</td>
</tr>
<tr>
<td>1000</td>
<td>10925.43</td>
<td>10924.26</td>
<td>11017.90</td>
</tr>
</tbody>
</table>

TABLE 4.3.1
FIGURE 4.3.1: Internal path length vs $n \log(n)$ for the $k$-dimensional search tree where

- - - - - - represents $k = 2$

............. represents $k = 3$

- - - - - - - - represents $k = 4$
FIGURE 4.3.2: Internal path lengths vs $n \log (n)$, where:

--- represents 3-d tree

\[\text{...............}\] represents 3-dimensional search tree
### HEIGHT DATA

#### A) BINARY SEARCH TREES

<table>
<thead>
<tr>
<th>n</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>9.8</td>
<td>9.8</td>
<td>9.95</td>
</tr>
<tr>
<td>250</td>
<td>15.66</td>
<td>15.47</td>
<td>15.83</td>
</tr>
<tr>
<td>500</td>
<td>18.41</td>
<td>18.22</td>
<td>18.49</td>
</tr>
<tr>
<td>1000</td>
<td>21.25</td>
<td>20.94</td>
<td>21.30</td>
</tr>
</tbody>
</table>

#### B) K-D TREES

<table>
<thead>
<tr>
<th>n</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>9.75</td>
<td>9.78</td>
<td>9.81</td>
</tr>
<tr>
<td>250</td>
<td>15.71</td>
<td>15.63</td>
<td>15.78</td>
</tr>
<tr>
<td>500</td>
<td>18.34</td>
<td>18.18</td>
<td>18.33</td>
</tr>
<tr>
<td>1000</td>
<td>20.96</td>
<td>20.95</td>
<td>21.13</td>
</tr>
</tbody>
</table>

**TABLE 4.4.1**
FIGURE 4.4.1: Height vs log n for the k-dimensional search tree where

- represents $k = 2$
- - - - - - represents $k = 3$
- - - - - - - - - - represents $k = 4$
FIGURE 4.4.2: Height vs log n, where

--- represents the 3-d tree

.......... represents the 3-dimensional search tree
Figure 4.4.2
4.5 Concluding Remarks

In summary, the building time for the k-dimensional search tree and k-d trees was found to be no longer than a binary tree. The finding that the internal path length for the k-dimensional search tree and k-d tree is proportional to $n \log n$ corresponds with Finkel and Bentley's [3 & 4] results. As expected, the binary tree results coincide with those given in Knuth [7]. Height proved to be proportional to $\log n$ for all tree types and sizes.

The binary search tree and k-d tree have less storage requirements than the k-dimensional search tree. This is easily seen because the k-dimensional search tree requires $2^k$ pointers at each node versus 2 pointers at each node for the binary search tree and k-d tree.

From the above we observe that the k-dimensional search tree and the k-d tree are as efficient as the binary search tree for exact match queries. It would appear that partial match, boolean, and near neighbor queries are more complex for the binary search tree. A complete transversal of the binary search tree would be required for any of these queries since keys are compared on one value (i.e. the concatenation of the k-attributes of the k-tuple keys usually). This implies that average running time of these queries would be of $O(n)$. 
Bentley [3] found that the k-d tree had an average running time of $O(\log n)$ for partial match and near neighbor queries. Finkel and Bentley [4] showed that the 2-dimensional search tree is quite efficient for boolean and near neighbor queries.

A number of areas remain to be explored for the k-dimensional and k-d trees. For example:

1) Extensions to the basic k-d insertion algorithm could be investigated as to the possibility of a weighted k-d tree construction algorithm.

2) The feasibility of applying existing optimization techniques for binary trees to k-d trees could be looked into.

3) According to Bentley [3], as yet an optimal deletion algorithm does not exist for the k-dimensional search tree. Further studies of deletion algorithms could be done for this tree structure.
References


APPENDIX A

EXAMPLE OUTPUTS AND LISTINGS OF THE DISPLAY ROUTINES

The Binary Tree Display Algorithms written by Aldon N. Walker [9] were modified to display k-dimensional search trees and k-d trees. A listing of the modified routines follows. First, example outputs are given for each tree type.
EXAMPLE 1: A Binary Search Tree

N = 10  K = 4

The Root Node is (2, 5, 2, 10)
EXAMPLE 2: A K-D Tree

N = 10  K = 4

The Root Node is (2,5,2,10)
EXAMPLE 3: A K-Dimensional Search Tree

N = 10  K = 3

The Root Node is (5,6,1)
THE DISPLAY ROUTINE

This routine displays binary trees and k-d trees. The only modification was to output k-tuple keys rather than unary keys. A call to the utility routine RITE was inserted in the VISIT procedure. Two asterisks on the left denote this change.
PROCEDURE DISPLAY(ROOT:KDTREE;INDENT,WIDTH,NODELINE:INTEGER);

GLOBAL TYPE(S)

DIRECTION--A SCALAR TYPE USED TO INDICATE THE DIRECTIONS WHICH 
MAY BE FOLLOWED FROM A NODE IE DIRECTION = (RIGHT,LEFT)

POINT------VARIABLES OF THIS TYPE ARE POINTERS TO TREE NODES.

LOCAL CONSTANT(S)

PRINTLIM--THE DIMENSION OF THE BOOLEAN ARRAY FRAGRAY, IT INDICATES 
THE NUMBER OF LEVELS OF THE BINARY TREE WHICH CAN BE 
PRINTED ON A PAGE AND MUST BE DETERMINED BY THE USER.

MAX------THE MAXIMUM NUMBER OF CHARACTERS WHICH ARE ALLOWED 
IN A KEY

LOCAL VARIABLE(S)

BPRINT----A BOOLEAN ARRAY USED TO INDICATE IF A BRANCH-CHARACTER 
SHOULD BE PRINTED FROM A NODE ON A PARTICULAR LEVEL, IE 
IF BPRINT[I] = TRUE A BRANCH-CHARACTER 
MUST BE PRINTED.

F---------BOOLEAN VARIABLE INDICATING IF:
TRUE = SEGMENTS OF BRANCHES SHOULD BE PRINTED IN THE 
NEXT PRINT LINE
FALSE = THE KEY OF THE NEXT NODE SHOULD BE PRINTED IN THE 
NEXT PRINT LINE

CONST  PRINTLIM = 33;
        MAX = 10;

TYPE     BOOLEANARRAY = ARRAY[0..PRINTLIM] OF BOOLEAN;

VAR     BPRINT: BOOLEANARRAY;
        F: BOOLEAN;
PROCEDURE SPACE(I: INTEGER);
PURPOSE: TO PRINT THE NUMBER OF SPACES INDICATED BY ITS PARAMETER
VAR J: INTEGER;
BEGIN
  FOR J = 1 TO I DO WRITE(" ")
END;

PROCEDURE PENTOBANCH(LEVEL: INTEGER; SPRINT: BOOLEAN);
PURPOSE: TO PRINT THE CHARACTERS (COLONS) OF THE SEGMENTS OF THE
BRANCHES BETWEEN A NODE JUST VISITED AND THE NEXT NODE TO BE
VISITED (THIS IS A DISTANCE OF WIDTH PRINT LINES)
VAR M, N: INTEGER;
BEGIN
  FOR N = 1 TO WIDTH DO
  BEGIN
    CALLING CONTROL AND INITIAL SPACING TO FIRST BRANCH
    CHARACTER POSITION:
    SPACE(NORELINE);
    PRINT A BRANCH CHARACTER (COLON) AT THIS POSITION IF A
    BRANCH EXISTS (PRINT = TRUE) WITH REQUIRED SPACING TO
    NEXT POTENTIAL BRANCH POSITION:
    FOR M = 1 TO LEVEL DO,
    IF PRINT(M) THEN
      BEGIN
        WRITE(" ");
        SPACE(INDENT - 1)
      END;
      WRITE();
      IF PRINT(M) THEN
        BEGIN
          F := FALSE
        END;
    END;
  END;
END;

PROCEDURE XBARPRINT(LINE: STRING; PRINT: BOOLEAN);
PURPOSE: TO PRINT THE CHARACTERS OF THE SEGMENTS OF THE BRANCHES
BETWEEN A NODE JUST VISITED AND THE NEXT NODE TO BE VISITED (THIS IS A
DISTANCE OF WIDTH PRINT LINES)
VAR M: INTEGER;
BEGIN
  FOR M = 1 TO LENGTH(LINE) DO
  BEGIN
    IF PRINT(M) THEN
      BEGIN
        WRITE(" ");
        SPACE(NORELINE)
      END;
    WRITE(LINE(M));
  END;
END;

PROCEDURE XBARPRINT(LINE: STRING; PRINT: BOOLEAN);
PROCEDURE VISIT (P:DOTREE; LEVEL:INTEGER; PRINT:BOOLEAN);

PURPOSE: TO PRINT THE KEY OF THE NODE POINTED TO BY P. HOWEVER, IT MAY
ALSO NECESSARY TO (A) PRINT CHARACTERS OF PRECEDING BRANCHES
(COLONS) SO AS THEY ARE DISPLAYED AS CONTINUOUS BRANCHES
(B) PRINT FILLER-CHARACTERS (MINUS SIGNS) PRECEDING THE KEY TO
LINK IT TO ITS PARENT'S BRANCH (C) PRINT FILLER-CHARACTERS
FOLLOWING THE KEY TO LINK IT TO ITS SONS BRANCH.

LABEL 10:
VAR I,J:INTEGER;
BEGIN +VISIT+
    "CARRIAGE CONTROL"
    SPACE(11):
    "IF THE NODE IS NOT THE ROOT NODE, PRINT AND FILLER-CHARACTERS"
    "IN THE SAME PRINT LINE AS THE PRESENT KEY"
    IF P NE' ROOT
    THEN
        "TWO CASES ARISE: EITHER NODELINE<INDENT OR NODELINE>INDENT"
        "(AND BY DEFINITION NODELINE2<INDENT)"
        IF NODELINE GT INDENT
        THEN
            "IF NODE TO BE VISITED IS A SON OF THE ROOT NODE"
            "(LEVEL = 1), NO BRANCH-CHARACTERS WILL BE PRESENT"
            "HENCE, SPACE TO THE FIRST PRINT POSITION OF THE KEY"
            IF LEVEL = 1
                THEN SPACE(INDENT)
            ELSE BEGIN
                "SPACE TO THE FIRST POTENTIAL BRANCH-CHARACTER POSITION"
                SPACE(NODELINE - 1);
                "PRINT A BRANCH-CHARACTER (REPRINT[I] ="
                "TRUE) WITH REQUIRED SPACING TO THE"
                "NEXT POTENTIAL BRANCH-CHARACTER"
                "POSITION"
FOR I := 1 TO LEVEL - 2 DO
  IF BPRINT[I] THEN
    BEGIN
      WRITE(E:EN); SPACE(INDENT - 1)
      END
    ELSE SPACE(INDENT); IF BPRINT[LEVEL - I] THEN
      BEGIN
        WRITE(E:EN); SPACE(2*INDENT - NOEDLINE)
        END
    ELSE SPACE(INDENT - NOEDLINE + 1);
  END
ELSE BEGIN
  SPACE TO FIRST POTENTIAL BRANCH-CHARACTER
  POSITION;
  SPACE(NOEDLINE - 1);
  PRINT A BRANCH-CHARACTER (BPRINT[I] = TRUE) WITH
  REQUIRED SPACING TO THE NEXT POTENTIAL
  BRANCH-CHARACTER POSITION;
FOR I := 1 TO LEVEL - 1 DO
  IF BPRINT[I] THEN
    BEGIN
      WRITE(E:EN); SPACE(INDENT - 1)
      END
  ELSE SPACE(INDENT);
  PRINT SPACE-CHARACTERS (MINUS, SIGNS) BEFORE
  KEY IF NECESSARY;
  SPACE(1);
  IF INDENT - NOEDLINE GT 1 THEN
    BEGIN
      FOR I := 1 TO INDENT - NOEDLINE DO
        WRITE(E:EN); SPACE(1)
      END
    END
PRINT THE KEY

CALLRITE TO OUTPUT THE K KEYS OF THE NODE P.

RITE(P):

PRINT FILLER-CHARACTERS (MINUS SIGNS) AFTER KEY UNLESS IT IS A LEAF.

10: IF(P.LFT NE NIL) OR (P.RTT NE NIL)
   THEN
      BEGIN
         SPACE(1):
         FOR J = K TO NOCLINE - 1. DO WRITE(J-E-J);
      END:
      Writeln;

SET FLAG INDICATING NEXT N这种方式 PRINT LINES WILL CONTAIN BRANCH-
CHARACTERS:

F J = TRUE.

END: VISIT:

------------------------------------------------------------------------------------------------------

PROCEDURE TRAVERSE(P:KOTREE;LEVEL:INTEGER;WAY:DIP):

PURPOSE: TO PERFORM A REVERSE POSTORDER TRAVERSAL OF THE BINARY TREE AND INITIATE THE PRINTING OF KEYS AND BRANCHES.
BEGIN **TRAVVERSE**

- WHEN P IS NIL THE TRAVERSAL CAN PROCEED NO FURTHER

IF P NE NIL THEN

BEGIN

CASE WAY OF
  RIGHT := BPRINT[LEVEL] = FALSE;
  LEFT := BPRINT[LEVEL] = TRUE;
END;

TRAVVERSE($P$; PTR, LEVEL + 1, RIGHT);
IF F THEN
  Pointer[LEVEL; BPRINT]:
  CASE WAY OF
    LEFT := BPRINT[LEVEL] = FALSE;
    RIGHT := BPRINT[LEVEL] = TRUE;
  END;
  VISIT($P$; LEVEL, BPRINT);
  TRAVVERSE($P$; LPT;, LEVEL + 1, LEFT);
  IF F THEN
    Pointer[LEVEL; BPRINT];
END:

END: **TRAVVERSE**

-----------------------------------------------

BEGIN **DISPLAY**

**INITIALIZE**

F := FALSE;

**RESTRICT MODELINE TO BE LESS THAN 2*INDENT**

IF MODELINE < 2*INDENT

THEN MODELINE := 2*INDENT - 1;

TRAVVERSE(ROOT; 0, RIGHT)

END: **DISPLAY**

-----------------------------------------------
THE KDISPLAY ROUTINE

This routine displays k-dimensional search trees. Since a node in a k-dimensional search tree can have up to $2^k$ sons, the transversal algorithm had to be completely re-written. The new algorithm is called KTRAVERSE. As with the DISPLAY routine, a change was inserted in the VISIT procedure to call the routine RITE in order to output k-tuple keys. All changes are denoted by two asterisks on the left. Complete listings are given for both display routines for continuity purposes.
PROCEDURE KDISPLAY
(ROOT: TREE; INDENT: WIDTH; NODELINE: INTEGER);

GLOBAL TYPE(S)

DIRECTION - A SCALAR TYPE USED TO INDICATE THE DIRECTIONS WHICH MAY BE FOLLOWED FROM A NODE, IE DIRECTION = (RIGHT, LEFT).

POINT-----VARIABLES OF THIS TYPE ARE POINTERS TO TREE NODES

LOCAL CONSTANT(S)

PRINTLINE---THE DIMENSION OF THE BOOLEAN ARRAY PRARRAY, IT INDICATES THE NUMBER OF LEVELS OF THE BINARY TREE WHICH CAN BE PRINTED ON A PAGE AND MUST BE DETERMINED BY THE USER.

MAX------THE MAXIMUM NUMBER OF CHARACTERS WHICH ARE ALLOWED IN A KEY

LOCAL VARIABLE(S)

BRPRINT---A BOOLEAN ARRAY USED TO INDICATE IF A BRANCH-CHARACTER SHOULD BE PRINTED FROM A NODE ON A PARTICULAR LEVEL, I, IN THE 'TREE'. IF BRPRINT[I] = TRUE A BRANCH-CHARACTER MUST BE PRINTED.

F---------BOOLEAN VARIABLE INDICATING IF:

TRUE: SECTIONS OF BRANCHES SHOULD BE PRINTED IN THE NEXT PRINT LINE.
FALSE: THE KEY OF THE NEXT NODE SHOULD BE PRINTED IN THE NEXT PRINT LINE.

CONST

PRINTLINE = 50;
MAX = 10;

TYPE

ARRAY = ARRAY([..PRINTLINE] OF BOOLEAN);

VAR

BRPRINT: ARRAY;
F: BOOLEAN;
PROCEDURE SPACE (I: INTEGER);
PURPOSE: TO PRINT THE NUMBER OF SPACES INDICATED BY ITS PARAMETER
VAR J: INTEGER;
BEGIN SPACE
FOR J := 1 TO I DO WRITE (' ')
END SPACE;

PROCEDURE PRINTBRANCH (LEVEL: INTEGER; BPRINT: BOOLEAN);
PURPOSE: TO PRINT THE CHARACTERS (COLONS) OF THE SEGMENTS OF THE
BRANCHES BETWEEN A NODE JUST VISITED AND THE NEXT NODE TO BE
VISITED (THIS IS A DISTANCE OF WIDTH PRINT LINES)
VAR X, NI: INTEGER;
BEGIN PRINTBRANCH
FOR X := 1 TO WIDTH DO
BEGIN
CASEage CONTROL AND INITIAL SPACING TO FIRST BRANCH-CHARACTER POSITION
SPACE (NOEDLINE);
PRINT A BRANCH CHARACTER (COLON) AT THIS POSITION IF A
BRANCH EXISTS (BPRINT = TRUE) WITH REQUIRED SPACING TO
NEXT POTENTIAL BRANCH POSITION
FOR Y := 1 TO LEVEL DO
IF BPRINT THEN
BEGIN
WRITE (' : ');
SPACE (INCR - 1);
ELSE SPACE (INCR);
END;
END:
RESET FLAG INDICATING NEXT PRINT LINE WILL CONTAIN A KEY

F := FALSE;
END; • PRINTBRANCH

PROCEDURE KVVISIT(P:KTREE;LEVEL:INTEGER:NS=PRINT;B=PRINT;NUM:INTEGER);
PURPOSE: TO PRINT THE KEY OF THE NODE POINTED TO BY P. HOWEVER IT MAY
ALSO NECESSARY TO (A) PRINT CHARACTERS OF PRECEDING ANCHORS
(DOLLS) SO AS THEY ARE DISPLAYED AS CONTINUOUS BRANCHES,
(B) PRINT FILLER-CHARACTERS (MINUS SIGNS) PRECEDING THE KEY TO
LINK IT TO ITS PARENT'S BRANCH (C) PRINT FILLER-CHARACTERS
FOLLOWING THE KEY TO LINK IT TO ITS SONS BRANCH

LABEL 10:
VAR i,j:INTEGER;
FLAG : BOOLEAN;
BEGIN 
  KVVISIT
  • CARRIAGE CONTROL
  SPACE(1);
  • IF THE NODE IS NOT THE ROOT NODE PRINT AND FILLER-CHARACTERS
     IN THE SAME PRINT LINE AS THE PRESENT KEY
  IF PRINT FOOT THEN
    • TWO CASES ARISE EITHER NODELINE<INDENT OF NODELINE=INDENT
      (AND BY DEFINITION NODELINE=INDENT)
      IF NODELINE = INDENT THEN
        • IF NODE TO BE VISITED IS A SON OF THE ROOT NODE
          (LEVEL = 1), NO BRANCH-CHARACTERS WILL BE PRINTED,
          HENCE SPACE TO THE FIRST PRINT POSITION OF THE KEY
          IF LEVEL EQ 1
             THEN SPACE(INDENT)
          ELSE BEGIN:
SPACE TO THE FIRST POTENTIAL BRANCH-CHARACTER POSITION
SPACE(NODELINE - 1):
PRINT A BRANCH-CHARACTER (BPREPRINT[I] = TRUE) WITH REQUIRED SPACING TO THE NEXT POTENTIAL BRANCH-CHARACTER POSITION:
FOR I := 1 TO LEVEL - 2 DO IF BPREPRINT[I] THEN
BEGIN WRITE(S1): SPACE(INDENT - 1)
END ELSE SPACE(INDENT):
IF BPREPRINT[LEVEL - 1] THEN
BEGIN WRITE(S2): SPACE(2 + INDENT - NODELINE)
END ELSE SPACE(2 + INDENT - NODELINE + 1);
ELSE BEGIN
SPACE TO FIRST POTENTIAL BRANCH-CHARACTER POSITION
SPACE(NODELINE - 1):
PRINT A BRANCH-CHARACTER (BPREPRINT[I] = TRUE) WITH REQUIRED SPACING TO THE NEXT POTENTIAL BRANCH-CHARACTER POSITION:
FOR I := 1 TO LEVEL - 1 DO IF BPREPRINT[I] THEN
BEGIN WRITE(S1): SPACE(INDENT - 1)
END ELSE SPACE(INDENT):
*PRINT FILLED-CHARACTERS (MINUS SIGNS) BEFORE KEY IF NECESSARY*

SPACE(1):
IF INJENT - NODELINE GT 1
  THEN
    BEGIN
      FOR I := 1 TO INJENT-NODELINE-1 DO 
          WRITE(NUM:LENGTH(NUM), E='E'); 
    END:
  END:

*PRINT THE KEYS*

WRITE(P):

*PRINT FILLED-CHARACTERS (MINUS SIGNS) AFTER KEY UNLESS IT IS A LEAF*

FLAG := FALSE;
I := 1;
REPEAT
  IF P, PIP[I] NE NIL THEN FLAG := TRUE;
  I := I + 1;
UNTIL (FLAG OR (I GT KSQ));

IF FLAG THEN
  BEGIN
    SPACE(1):
    FOR J := K TO NODELINE - 1 DO WRITE(E='E);
  END:

WRITELY:

*SET FLAG-INDICATING NEXT WIDTH PRINT LINES WILL CONTAIN BRANCH-CHARACTERS*

F := "TRUE"
END: *KVIST*
**PROCEDURE KTRAVVERSE(P:KTREE;LEVEL:INTEGER;WAY:DIR;PTT,CODE:INTEGER);**

PURPOSE: TO PERFORM A REVERSE POSTORDER TRAVERSAL OF THE QUAD TREE AND INITIATE THE PRINTING OF KEYS AND BRANCHES

VAR: II, CT, FL: INTEGER;

BEGIN {KTRAVVERSE}

*WHEN P IS NIL THE TRAVERSAL CAN PROCEED NO FURTHER*:

IF P NE NIL THEN

BEGIN

IF (CODE LT 3). THEN

CASE WAY OF

RIGHT@PRINT[LEVEL] := FALSE;
LEFT@PRINT[LEVEL] := TRUE
END;

IF P@PRINT[II] NE NIL THEN CT := CT + 1;
II := II + 1;
UNTIL (II GT CT MID)

IF CT NE 0 THEN BEGIN
FL := 2;
II := 1;
REPEAT
IF P@PRINT[II] NE NIL THEN BEGIN
KTRAVVERSE(P@PRINT[II],LEVEL+1,RIGHT,II,FL);
CT := CT + 1;
END;
II := II + 1;
UNTIL (CT = 0)
END;

IF THEN
@PRINTS(L,LEVEL,PRINT);
IF (CODE GT 1) THEN
CASE WAY OF
LEFT := @PRINT[LEVEL] := FALSE;
END;
RIGHT: SPRINT(LEVEL) := TRUE

KVSIIT(P, LEVEL, SPRINT, PTLI):
CT := 0
II := SHID:
REPEAT
IF P* STR(III) NE NIL THEN CT := CT + 1;
II := II + 1
UNTIL (II ≤ CT KSQ):
IF CT NE 0 THEN BEGIN
IF CT SQ : 1 THEN FL := 2
II := SHID:
REPEAT
IF P* STR(III) NE NIL THEN BEGIN
KPAVERSE(P*, PTP(III), LEVEL+1, LEFT, II, FL):
CT := CT - 1;
IF CT EN 1 THEN FL := 3
ELSE FL := 0;
END;
II := II + 1
UNTIL (CT = 0):
END:
IF F THEN
PRINTBRANCH(LEVEL, SPRINT):
END; *KTRAVERSE*

BEGIN *KDISPLAY*

*INITIALIZE*
F := FALSE:

*RESTRICT NOEDLINE TO BE LESS THAN 2*INDENT*
IF NOEDLINE GE 2*INDENT
THEN NOEDLINE := 2*INDENT - 1;
KPAVERSE(ROOT, 0, LEFT, 0, 1):

END; *KDISPLAY*
APPENDIX B

LISTINGS OF THE TREE ALGORITHMS

GLOBAL CONSTANTS

The following global constants were used throughout each tree algorithm:

a) KK - represents the number of attributes in each key (KK = k with respect to k-tuple key).

b) NUMTREES - represents the number of trees to be built.

c) MAXSIZE - represents the number of keys to be inserted into each tree (MAXSIZE = N).

d) The global constant PROG used in the K-D/Binary Tree Algorithm denotes which of the two tree types were to be built. If PROG = 1, the Binary Search Tree Algorithm would be used to build the trees. If PROG ≠ 1, the K-D Tree Algorithm would be used to build the trees.

COMMON ROUTINES

A listing of the common routines follows the tree algorithms.

A listing of each tree algorithm follows:

A. The K-D/BINARY TREE Algorithm

B. The K-DIMENSIONAL TREE Algorithm
PROGRAM INFO(INPUT, OUTPUT);

K-D / BINARY TREE

PURPOSE: TO BUILD A K-D TREE FROM NODES INPUT WITH K KEYS USING
BENTLEY'S INSERTION ALGORITHM.

GLOBAL CONSTANTS, TYPES, VARIABLES

LABEL 999:

CONST

POG = 1;
KK = 4;
NUMTREES = 3;
MAXSIZE = 1000;
INTERVAL = 10;
NUMINT = 2;

TYPE

ALFA = 1..100000;
AALFA = ARRAY[1..KK] OF ALFA;
AINT = ARRAY[1..KK] OF INTEGER;
AINT = ARRAY[1..MAXSIZE] OF INTEGER;
ACHAR = ARRAY[1..1..4] OF CHAR;
KEYAZZAY = ARRAY[1..KK, 1..MAXSIZE] OF ALFA;
KOTREE = NODE;
NODE = RECORD

KEY = INTEGER;
RBT + LBT + KOTREE;
DISC = INTEGER;
END;
WAY = (CLOSE, HISONE);
DIR = (RIGHT, LEFT);
VAR
  ROOT, PP, SAV, R, ADDR, FREE : KDTREE;
  SEED : AIINT;
  RANDP : INT;
  RANDNUM : KEVARRAY;
  DIRECTION : DTR;
  STATS : ARRAY[1..NUMINTS, 1..3] OF REAL;
  PLENTH, HGT, HIGHT, TIME, NUM : REAL;
  NODECOUNT, K, J : INTEGER;
  T1, T2, SEC, ISUELO : INTEGER;
  NUMNODES, NODESIZE : INTEGER;
  STIME : REAL;
FUNCTION ACQUIRE(VAR P: NODE); BOOLEAN:

PURPOSE: BOOLEAN FUNCTION TO RETURN A POINTER TO A NODE. ACQUIRE
FIRST TRIES TO FIND A NODE IN THE FREE LIST AND IF THIS
FAILS IT TRIES TO ALLOCATE A NEW NODE USING THE STANDARD
PASCAL PROCEDURE NEW. IF ALL NODES HAVE BEEN ALLOCATED AND
ARE IN USE THE OUTPUT PARAMETER IS RETURNED AS NIL AND THE
FUNCTION IS TRUE.

OUTPUT PARAMETERS:
  P--POINTER TO THE NEW NODE

BEGIN
  ACQUIRE := FALSE;
  IF FREE EQ NIL THEN
    TRY TO ALLOCATE A NEW NODE; IF THIS FAILS PRINT A
    MESSAGE AND RETURN THE FUNCTION VALUE TRUE.
    BEGIN
      NEW(P):
      IF P EQ NIL THEN
        BEGIN
          GLASSOVERFLOW:
          ACQUIRE := TRUE;
        END;
      END;
    ELSE
    * TAKE THE NODE FROM THE FREE LIST *
    BEGIN
      P := FREE;
      FREE := FREE+.RPTR;
    END;
  END; * ACQUIRE *
PROCEDURE RELEASE (P: KDTPCE);

PURPOSE: TO PLACE A NODE (POINTED TO BY THE GIVEN INPUT PARAMETER) FIELDS OF THE NODES ARE USED TO FORM THE CHAIN.

INPUT PARAMETERS
P--POINTER TO THE DELETED NODE

BEGIN "RELEASE"
IF FREE = NIL THEN
  "THERE IS ONLY ONE NODE IN THE FREE LIST; DEFINE FREE TO POINT TO IT."
  BEGIN
    FREE := P;
    FREE+PTR := NIL;
  END
ELSE
  "PLACE THE NODE ON THE FREE LIST."
  BEGIN
    P+PRT := FREE;
    FREE := P;
  END:
END "RELEASE"

PROCEDURE DESTROY (RT: KDTPCE);

PURPOSE: TO BREAKDOWN THE TREE WHOSE ROOT IS THE INPUT PARAMETER IN ORDER TO RELEASE THE TREES POINTERS WHICH ARE TO BE USED TO CONSTRUCT THE NEXT TREE - GARBAGE COLLECTION.

INPUT PARAMETERS
RT--THE ROOT OF THE TREE TO BE BROKENDOWN
VAR CT, II : INTEGER;
BEGIN *DESTROY*
  CT := 0;
  + COUNT THE NUMBER OF NON-NIL POINTERS OFF THE NODE PT +
  IF PT*PPTF NE NIL THEN CT := CT + 1;
  IF PT*LPTF NE NIL THEN CT := CT + 1;
  + IF COUNT IS 0 THEN RELEASE THAT NODE SINCE THAT IS AN END-NODE +
  IF CT EQ 0 THEN RELEASE(PT)
    ELSE
      + IF NOT AN END-NODE THEN CALL RELEASE RECURSIVELY TO RELEASE +
      ALL THE NODES IN EACH OF THE SUB-TREES. +
      BEGIN
        + RECURSIVELY CALL DESTROY FOR EACH TREE WHEN THE POINTER +
        IS NOT NIL +
        IF RT*PPTF NE NIL THEN BEGIN
          DESTROY(RT*PPTF);
          RT*PPTF := NIL;
          END;
        IF RT*LPTF NE NIL THEN BEGIN
          DESTROY(RT*LPTF);
          RT*LPTF := NIL;
          END;
        RELEASE(RT);
        END;
      END; *DESTROY*
PROCEDURE PRINTFREE;
* PURPOSE: TO OUTPUT THE CONTENTS OF THE FREE LIST IN ORDER TO
   TEST THE DESTROY PROCEDURE.
BEGIN PRINTFREE;
   WRITELN(FREE);
   WHILE (FREE NE NIL) DO
     BEGIN
       WRITE(E);
       WRITE(RITE(FREE));
       WRITELN;
       FREE := FREE.RPTR;
     END;
END; PRINTFREE
FUNCTION COMPARE(FIRST,SECOND;OPTREE;JJ:INTEGER):INTEGER;

† PURPOSE: TO COMPARE THE KEYS OF FIRST AND SECOND STARTING AT THE JJ-TH KEY IF JJ GT 0 AND TO RETURN AS THE VALUE OF THE BOOLEAN FUNCTION COMPARE:

0: IF FIRST < SECOND
1: IF FIRST = SECOND
2: IF FIRST > SECOND

† PARAMETER(S):
FIRST--POINTER TO THE NODE TO BE INSERTED
SECOND--POINTER TO A NODE IN THE TREE TO BE COMPARED TO FIRST

THE METHOD USED TO COMPARE THE TWO NODES WAS TO COMPARE EACH CORRESPONDING KEY IN TURN AND SET THE RESULT IN COMPARE APPROPRIATELY.

VAR
OUT: BOOLFAN;
I,J: INTEGER;
BEGIN COMPAŒRE
† INITIALIZE
COMPARE := 0;
OUT := FALSE;
I := JJ
IF JJ GT 0 THEN J := JJ
ELSE J := 1;
† REPEAT LOOP COMPARING EACH CORRESPONDING KEY IN THE NODES UNTIL FIRST <> SECOND OR UNTIL THE LIST OF KEYS IS EXHAUSTED
WHILE NOT OUT) AND (I LE KK) DO
BEGIN
IF PANDNUM(J,FIRST,key) GT PANDNUM(J,SECOND,key) THEN
BEGIN
OUT := TRUE;
COMPARE := 1;
END
END
.
ELSE
IF PANDNUM[I;FIRST*KEY] LT PANDNUM[j;SECOND*KEY] THEN
BEGIN
OUT := TRUE;
COMPARE := 2;
END;
IF (IJ .GT 0) AND (J EQ KK) THEN J := 1
ELSE J := J + 1;
I := I + 1;
END;
END; "COMPARE".

FUNCTION NEXTDISC(PREV:INTEGER) : INTEGER;
* PURPOSE: TO DETERMINE THE NEXT DISCRIMINATOR FROM THE DISCRIMINATOR OF THE PREVIOUS IODE, I.E., NEXTDISC = (I+1) MOD KK
PARAMETER(S): PPREV--CURRENT DISCRIMINATOR USED TO DETERMINE THE NEXT DISCRIMINATOR.
VAR NEXT:INTEGER;
BEGIN "NEXTDISC"
NEXT := PREV + 1;
IF NEXT GT KK THEN NEXT := NEXT - KK;
NEXTDISC := NEXT;
END; "NEXTDISC".
FUNCTION SUCCESSOR(P, Q: KOTREE):WAY:
    PURPOSE: TO DETERMINE WHICH SON OF P THE NODE Q BELONGS.
    PARAMETER(S):
        P -- POINTER TO A NODE IN THE TREE TO WHICH Q IS TO BE COMPARED.
        Q -- POINTER TO THE NODE TO BE INSERTED

VAR J: INTEGER;
BEGIN SUCCESSOR:
    LET J = THE DISCRIMINATOR
    J := P^DISC;
    IF THE J-TH KEY OF Q < J-TH KEY OF P THEN RETURN LOSON

    IF RANDNUM(J, Q^K.KEY) LT RANDNUM(J, P^K.KEY) THEN SUCCESSOR := LOSON
    ELSE
        IF THE J-TH KEY OF Q > J-TH KEY OF P RETURN HISON

        IF RANDNUM(J, Q^K.KEY) GT RANDNUM(J, P^K.KEY) THEN SUCCESSOR := HISON
        ELSE BEGIN
            J := COMPARE(Q, P, J);
            CASE J OF
                0, 1: SUCCESSOR := HISON;
                2: SUCCESSOR := LOSON;
            END;
        END;

END: SUCCESSOR
FUNCTION BINSEP(KNODE:INTEGER; HK:INTEGER) : KOTREE;

PURPOSE: TO INSERT A NODE INTO A BINARY TREE IF IT DOES NOT ALREADY EXIST AND IF IT DOES, TO RETURN THE ADDRESS OF THE EQUIVALENT NODE IN THE TREE.

PARAMETER(S):

KNODE--THE ARRAY OF INTEGERS WHICH ARE USED AS INDIRECT ADDRESSES TO THE NODES TO BE INSERTED.

HK----THE POSITION IN THE KNODE ARRAY WHICH POINTS TO THE NODE BEING INSERTED.

VAR

SAME : BOOLEAN;
VAL : INTEGER;

BEGIN

ALLOCATE NEW POINTER ADDRESS AND SET KEY ADDRESS

IF ACQUIRE(PP) THEN GOTO 999;
PP.*KEY := KNODE(HK);

INITIALIZE VARIABLES

BINSEP := NIL;
PP := POCT;
SAME := FALSE;
HGT := 0;

REPEAT LOOP UNTIL A NULL NODE IS FOUND

WHILE PP NE NIL DO
BEGIN

SAVE PREVIOUS NODE ADDRESS IN ORDER TO INSERT POINTE TO INSERTED NODE.

SAVE := PP;

INCREMENT HGT COUNT I.E. HEIGHT OF TREE

HGT := HGT + 1;

999:

RETURN IN错误 (KRONE) ADDRESS OF NEW NODE.

END.
CALL ROUTINE COMPARE TO COMPARE THE KEYS OF THE NODES.

VAL := COMPARE(pp, P, D);

IF VAL = 0 THEN NODES KEYS ARE EQUAL THEN SET FUNCTION EQUAL TO NODE ADDRESS AND DELETE CREATED NODE POSITION.

IF VAL EQ 0 THEN BEGIN
    BENSET := P;
    P := NIL;
    SAME := TRUE;
    RELEASE(pp);
END ELSE

BEGIN

SET DIRECTION IN WHICH TO FOLLOW IN TREE

IF VAL EQ 1 THEN BEGIN
    DIRECTION := RIGHT;
    P := P$RT;
END ELSE BEGIN
    DIRECTION := LEFT;
    P := P$LT;
END;

END;

ELSE IF NODE NOT FOUND THEN INSERT IT INTO TREE

IF NOT SAME THEN BEGIN

PP$LPT := NIL;
PP$RPT := NIL;

INCREMENT NODECOUNT

NODECOUNT := NODECOUNT + 1;

ADD INTERNAL PATH LENGTH FOR CURRENT NODE TO TOTAL INTERNAL PATH LENGTH OF TREE

PLENGTH := PLENGTH + MGT;

END.
TEST IF FOOT IS NIL AND IF SO LET FOOT = ADDRESS +
IF FOOT /= NIL THEN FOOT := PP
ELSE CASE DIRECTION OF
    LEFT 1 SAVER++.LPtr := PP;
    RIGHT 1 SAVER++.RPtr := PP;
END;
END; *BINSEPT*
FUNCTION KDINSERT(KNODE:INT; IK:INTEGER):KOTREE;

PURPOSE: TO INSERT A NODE P INTO A K-D TREE IF IT DOES NOT ALREADY EXIST AND IF IT DOES, TO RETURN THE ADDRESS OF THE EQUAL NODE IN THE TREE.

PARAMETERS:

KNODE--THE ARRAY OF INTEGERS WHICH ARE USED AS INDIRECT ADDRESSES TO THE NODES TO BE INSERTED.

IK--THE POSITION IN THE KNODE ARRAY WHICH POINTS TO THE NODE BEING INSERTED.

VAR

SAME: BOOLEAN;
IJ: INTEGER;
SON: WAY;

BEGIN KDINSERT

ALLOCATE NEW POINTER AND SET KEYS

IF ACQUIRE(PP) THEN GOTO 999;
PP^.KEY := KNODE[IK];

initialize variables

KDINSERT := NIL;
SAME := FALSE;
P := FOOT;
HGT := 0;

REPEAT LOOP UNTIL A NULL NODE IS ENCOUNTERED
WHILE P NE NIL DO
BEGIN

INCREMENT HGT COUNT I.E. HEIGHT OF TREE

HGT := HGT + 1;
SAME := TRUE;

END.
SAVE PREVIOUS NODE IN ORDER TO INSERT POINTER TO INSERTED NODE.

SAVE: \( \text{SAVE} \equiv R \)

• COMPARE KEYS TO DETERMINE IF EQUAL NODES
  - SAME \( \equiv \) TRUE IF EQUAL
    - FALSE OTHERWISE.

\( IJ \neq 0 \)

PEEK: \( IJ = IJ + 1 \)

IF RAND\textsc{num}(IJ, PP\textsc{p}, \textsc{key}) \neq \textsc{RAND\textsc{num}(IJ, PP\textsc{p}, \textsc{key})} \) THEN
  SAME \( \equiv \) FALSE

UNTIL \( (IJ \equiv KK) \) OR (NOT SAME);

• IF THE KEYS ARE NOT EQUAL, MOVE DOWN THE TREE

IF NOT SAME THEN

BEGIN

• DETERMINE WHICH TREE TO GO DOWN - HISON OR LOSON

SON \( \equiv \) SUCCESSOR\textsc{p}(P, PP)

CASE SON OF

  - LOSON: \( P \equiv R\textsc{p}, \textsc{LPT} \)
  - HISON: \( P \equiv R\textsc{p}, \textsc{RPT} \)

END

ELSE

• WHEN KEYS ARE EQUAL RETURN THE ADDRESS OF THE EQUAL NODE
  AND DELETE THE ALLOCATED POINTER.

BEGIN

  KD\textsc{INSERT} \( \equiv P \);

  \( i \equiv \) NIL;

  \( *=\) DELETE(PP);

END

END: WHILE
"IF NODE NOT FOUND IN THE TREE, THEN INSERT IT INTO THE TREE."

IF NOT SAME THEN

BEGIN

  PP+ LPTP := NIL;
  PP+ RPTP := NIL;
  NODECOUNT := NODECOUNT + 1;
  ADD INTERNAL PATH LENGTH FOR CURRENT NODE TO TOTAL INTERNAL PATH LENGTH OF TREE:

  PLENTH := PLENTH + HGT;

  TEST IF ROOT IS NIL:

  IF ROOT EQ NIL THEN

  BEGIN

    LET ROOT = ADDRESS OF INSERTED NODE AND SET DISC TO 1:

    BEGIN

      ROOT := PP;
      ROOT*DISC := 1;
    END

    ELSE

    INSERT ADDRESS OF NEW NODE IN THE APPROPRIATE BRANCH OF THE PREVIOUS NODE AND CALL NEXTDISC TO DETERMINE THE DISCRIMINATOR OF THE NEW NODE:

    BEGIN

      CASE SON OF

        LEFT SON: SAV+*LPTP := PP;
        RIGHT SON: SAV+*RPTP := PP;
      END;

      PP+DISC := NEXTDISC(SAV+DISC);
    END;

END. *KINSERT.
PROCEDURE PRINT(P:KD_TREE);
BEGIN
  IF P = NIL THEN
    BEGIN
      WRITELN("NO E");
      IF P = 0 THEN
        WRITELN("BASIC BINARY TREE ***E");
      ELSE
        WRITELN("BASIC K-C TREE ***E");
      WRITELN("FATHER: " & P.FATHER); NODECOUNT = NODES IN THIS TREE;
      WRITE("THE ROOT OF THIS TREE IS " & P.ROOT);
      WRITE("PLE: " & P.HELP);
      WRITE("THE PARENT OF " & P.HELP & " IS " & P.HELP1);
      DISPLAY(P, P, 4, 5);
      WRITELN("E");
    END;
  WRITELN;
END; *PRINT*}
PROCEDURE INITIALIZE:

PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS OF VARIABLES AND ARRAYS.

VAR
   I, J : INT;
BEGIN INITIALIZE
   FREE I = NIL;
   XNUM := NUMTREES;
   INITIALIZE RANDOM NUMBER ARRAY
   FOR I := 1 TO KK DO
      FOR J := 1 TO MAXSIZE DO
         RNUM[I, J] := J;
   INITIALIZE INDIRECT ADDRESS ARRAY
   FOR I := 1 TO MAXSIZE DO
      PACR[I] := I;
   INITIALIZE SEEDS
   SED := 34333;
   FOR I := 1 TO KK DO
      BEGIN
         SEED[I] := SED;
         SED := SED + TSNUT(I*RAND(0, 0, 1, 0, SED)) + 1;
         IF NOT ODD(SED) THEN SED := SED + 1;
      END;
   INITIALIZE STATISTICS ARRAY
   FOR I := 1 TO NUMINTS DO
      FOR J := 1 TO 9 DO
         STATS[I, J] := 0;
END; INITIALIZE
PROCEDURE BUILDTREE;
  PURPOSE: TO BUILD UP A TREE AND COLLECT THE REQUIRED STATISTICS.

VAR
  INC: INTEGER;

BEGIN BUILDTREE

  INITIALIZE

  ROOT := NIL;
  NODECOUNT := n;
  HEIGHT := 0;
  PLENGTH := 0;
  NODEROUTE := INTERVAL;
  NUMNODES := 1;
  TIME := 0;

  PERMUTE THE ELEMENTS IN THE RANDOM NUMBER ARRAYS
  PERMUTE(PANDNUM, SEED, MAXSIZE);

  PERMUTE THE VECTORS OF KEYS USING THE INDIRECT ADDRESS ARRAY
  SHUFFLE(PADDR, SEED, MAXSIZE);

  CALL FOR CURRENT CLOCK TIME AT START OF TREE
  T1 := CLOCK;

  INSERT THE NODES TO BUILD TREE
  FOR INC := 1 TO MAXSIZE DO BEGIN
    IF PROG EQ 1 THEN ADDR := BININSERT(PADDR, INC)
    ELSE ADDR := KINSERT(PADDR, INC);

  END

END BUILDTREE;
IF THE NODE ALREADY EXISTS, OUTPUT A MESSAGE.

IF ADDR = NIL THEN
BEGIN
WRITE("Node already exists.");
END;
ELSE
IF HGT GT HEIGHT THEN HEIGHT := HGT;
IF INC = NODESIZE THEN BEGIN
  CALL FOR CURRENT CLOCK TIME +
  T2 := CLOCK;
  SUBTRACT TO GET BUILDING TIME +
  TIME1 := T2 - T1;
  TIME := TIME1 + STIME;
  GATHER STATISTICS +
  ACCEUMULATE;
  INCREMENT NODESIZE +
  NODESIZE := NODESIZE + INTERVAL;
  CALL FOR CURRENT TIME +
  T1 := CLOCK;
END;
END;
END; → BUILDTREE →
PROCEDURE PRINTHEADER;
  PURPOSE: TO PRINT A HEADING
BEGIN PRINTHEADER:

IF PROG = 1 THEN:
  WRITE(E1"STATISTICS FOR A BINARY TREE USING E, KK:2, E KEYSE")
ELSE:
  WRITE(E1"STATISTICS FOR A E, KK:2, E-0 TREE");
  WRITE(E1"NUMBER OF TREES BUILT = E, NUM:3:14");
  WRITE(E1"MAXSIZE BUILDING TIME E");
  WRITE(E1"INTERNAL PATH LENGTH HEIGHTE");
  WRITE(E1"AV MIN MAX AV MIN MAX");
END PRINTHEADER;
BEGIN ·INFO·

LINELIMIT(OUTPUT, 2500);

CALL PROCEDURE INITIALIZE TO PERFORM INITIALIZATIONS.

INITIALIZE:

PRINT HEADER

PRINT-HEADER:

COLLECT STATISTICS

FOR BUILD := 1 TO NUMTREES DO

BEGIN

BUILD TREE AND COLLECT STATISTICS ON THAT TREE

BUILD TREE;

CALL DESTROY TO BREAKDOWN THE TREE;

DESTROY(FOOT);

END:

OBTAIN AVERAGES AND OUTPUT RESULTS

NODESIZE := INTERVAL;

FOR I := 1 TO NUMINTS DO

BEGIN

STATS[I, 1] := STATS[I, 1] / XNUM;


PRINTCUT(I);

INCREMENT NODESIZE

NODESIZE := NODESIZE + INTERVAL;

END:

END ·INFO·
PROGRAM KINFO(INPUT, OUTPUT);

PURPOSE: TO CREATE A QUAD TREE USING THE BASIC INSERTION ALGORITHM
K-DIMENSIONAL TREE

GLOBAL CONSTANTS, TYPES, VARIABLES

LABEL 999:

CONST
KK = 4;
KSQ = 16;
NUMTEES = 3;
MAXSIZE = 100;
INTERVAL = 100;
NUMINTS = 1;

TYPE
ALFA = 1..100000;
AINT = ARRAY[1..KK] OF INTEGER;
MINT = ARRAY[1..MAXSIZE] OF INTEGER;
KEYARRAY = ARRAY[1..KK, 1..MAXSIZE] OF ALFA;
KTREE = NODE;
NODE = RECORD
KEY: INTEGER;
DIR: ARRAY[1..KSQ] OF KTREE;
END;
DIR = (LEFT, RIGHT);

VAR
FOOT, PP, SAVF, DADD, FPEE, KTPES;
PCHNUM, KEYARRAY;
SEED, AINT;
PADD, MINT;
STATS = ARRAY[1..NUMINTS, 1..9] OF REAL;
PLENGTH, HEIGHT, TIME, NUM : REAL;
I1, II, KSEQ : INTEGER;
MAX, NODECOUNT, CMID, CMID : INTEGER;
T1, T2, IBUILD : INTEGER;
NUMNODES, NODESIZE : INTEGER;
STIME : REAL;
FUNCTION ACQUIRE(VAR P: TREE) : BOOLEAN;
PURPOSE: BOOLEAN FUNCTION TO RETURN A POINTER TO A NODE. ACQUIRE
FIRST TRIES TO FIND A NODE IN THE FREE LIST AND IF THIS
FAILS IT TRIES TO ALLOCATE A NEW NODE USING THE STANDARD
PASCAL PROCEDURE NEW. IF ALL NODES HAVE BEEN ALLOCATED AND
ARE IN USE THE OUTPUT PARAMETER IS RETURNED AS NIL AND THE
AND THE FUNCTION IS TRUE.

OUTPUT PARAMETERS:
P -- POINTER TO THE NEW NODE

BEGIN ACQUIRE +
ACQUIRE := FALSE;
IF P = NIL THEN
TE TRY TO ALLOCATE A NEW NODE; IF THIS FAILS PRINT A
MESSAGE AND RETURN THE FUNCTION VALUE TRUE.
BEGIN
NEW(P):
IF P = NIL THEN
BEGIN
CLAS Overflow;
ACQUIRE := TRUE;
END;
ELSE:
TAKE THE NODE FROM THE FREE LIST +
BEGIN
P := FREE;
FREE := FREE + PTR(1);
END; ACQUIRE +
PROCEDURE RELEASE(P:KTREE);

PURPOSE: TO PLACE A NODE (POINTED TO BY THE GIVEN INPUT PARAMETER) IN THE FREE LIST. 
FIELDS OF THE NODES ARE USED TO FORM THE CHAIN.

INPUT PARAMETERS
P -- POINTED TO THE DELETED NODE.

BEGIN
  IF P = NULL THEN
    /* THERE IS ONLY ONE NODE IN THE FREE LIST; DEFINE FREE TO POINT TO IT. */
    BEGIN
      FREE := P;
      FREE_PTR[1] := NULL;
    END
  ELSE
    /* PLACE THE NODE ON THE FREE LIST. */
    BEGIN
      FREE_PTR[1] := FREE;
      FREE := P;
    END
  END
END; /* RELEASE */
PROCEDURE DESTROY(PT: TREE);

PURPOSE: TO BREAKDOWN THE TREE WHOSE ROOT IS THE INPUT PARAMETER IN ORDER TO RELEASE THE TREE'S POINTERS WHICH ARE TO BE USED TO CONSTRUCT THE NEXT TREE. - GARBAGE COLLECTION.

INPUT PARAMETER:

PT-- THE FOOT OF THE TREE TO BE BROKEN DOWN

VAR

CT, II : INTEGER;

BEGIN

DESTROY

CT := 1;

COUNT THE NUMBER OF NON-NIL POINTERS OFF THE NODE PT

FOR II := 1 TO KSQ DO

IF PT + .PTR[II] = NIL THEN CT := CT + 1;

IF COUNT IS 0 THEN RELEASE THAT NODE SINCE THAT IS AN END-NODE

IF CT EQ 0 THEN RELEASE(PT);

ELSE

IF NOT AN END-NODE THEN CALL RELEASE RECursively TO RELEASE ALL THE NODES IN EACH OF THE SUBTREES.

BEGIN

RECURSIVELY CALL DESTROY FOR EACH TREE WHEN THE POINTER IS NOT NIL

FOR II := 1 TO KSQ DO

IF PT + .PTR[II] NE NIL THEN BEGIN

DESTROY(PT + .PTR[II]);

END;

RELEASE(PT);

END;

DESTROY;

END.
FUNCTION EXAMINE(P,K : K T R E E) : INTEGER;

* PURPOSE: TO COMPARE THE KEYS OF NODE P AGAINST THE KEYS OF NODE K
IN ORDER TO DETERMINE WHICH DIRECTION IN THE TREE TO
PROCEED. THE FUNCTION COMPARE RETURNS AS ITS VALUE:
  0 : ALL KEYS OF P EQUAL K
  KK : AN INTEGER BETWEEN 1 AND KK WHICH REPRESENTS THE DIRECTION TO PROCEED
       IN THE TREE.

PARAMETER(S):

  P : POINTER TO THE NODE TO BE INSERTED
  K : POINTER IN TREE TO BE COMPARED WITH P

VAR

  A : ARRAY WHICH HOLDS THE RESULTS OF THE COMPARISON WHICH INDICATES THE DIRECTION IN WHICH TO FOLLOW IN THE TREE.
  SAME : BOOLEAN VARIABLE WHICH IS SET TO TRUE IF BOTH NODES ARE EQUAL, FALSE OTHERWISE.

A : ARRAY
I : POINTER, NUM : INTEGER
SAME : BOOLEAN

BEGIN EXAMINE
  SAME := TRUE;
  I := I;
  TEST IF THE TWO NODES HAVE EQUAL KEYS
REPEAT
IF RANDNUM[I,R*KEY] NE RANDNUM[I,K*KEY] THEN SAME := FALSE;
  I := I + 1;
UNTIL (NOT SAME) OR (I GT KK);

* IF NODES ARE EQUAL RETURN 0 *  
IF SAME THEN EXAMINE := 0
ELSE
  WHEN NODES ARE NOT EQUAL, COMPARE EACH CORRESPONDING KEY IN THE
  NODES AND SET THE CORRESPONDING POSITION IN THE ARRAY A TO A
  1 IF KEY IN R GE KEY IN K AND TO 0 IF KEY IN R LT KEY IN K.

BEGIN
  FOR I := 1 TO KK DO
    ELSE A[I] := 0;

  CONVERT THE BINARY DIGIT STORED IN A TO A DECIMAL INTEGER +
  NUM := 0;
  POWER := 1;
  FOR I := 1 TO KK DO
    BEGIN
      NUM := NUM + A[I] * POWER;
      POWER := POWER * 2;
    END;

  RETURN THIS INTEGER AS THE DIRECTION IN THE TREE IN WHICH
  TO PROCEED.  
  EXAMINE := NUM + .1;

END; *EXAMINE*
PROCEDURE PRINTFREE;
* PURPOSE: TO OUTPUT THE CONTENTS OF THE FREE LIST IN ORDER TO TEST THE DESTROY PROCEDURE.
BEGIN 
  PRINTFREE:
  WRITELN(SS15);
  WHILE (FREE NE NIL) DO 
  BEGIN 
    WRITE(I);
    WRITE(E); WRITE(FREE);
    WRITELN;
    FREE := FREE + PTR[1];
  END;
END; *PRINTFREE*
FUNCTION KINSEPT(KNODE: INTEGER; IJ: INTEGER); KPOOL;

PURPOSE: TO INSERT A NODE INTO A K-DIM TREE IF IT DOES NOT ALREADY EXIST, AND IF IT DOES, TO RETURN THE ADDRESS OF THE EQUIVALENT NODE IN THE TREE.

KNODE -- THE ARRAY OF INTEGERS WHICH ARE USED AS INDIRECT ADDRESSES TO THE NODES TO BE INSERTED

IJ -- THE POSITION IN THE KNODE ARRAY WHICH POINTS TO THE NODE BEING INSERTED.

VAR CHK : BOOLEAN;
BEGIN KINSEPT
  ALLOCATE NEW POINTER ADDRESS AND SET KEY ADDRESS
  IF ACHIEVE (PP) THEN GOTO 999;
  PP.KEY = KNODE[IJ];
  INITIALIZE VARIABLES
  KINSEPT := NIL;
  CHK := TRUE;
  CHK := FALSE;
  HGT := 0;

  REPEAT LOOP UNTIL A NULL NODE IS FOUND
  WHILE NOT NIL DO
  BEGIN 
    SAVE PREVIOUS NODE ADDRESS IN ORDER TO INSERT POINTERS TO INSERTED NODE.
    INCREMENT HGT COUNT, I.E., HEIGHT OF TREE
    HGT := HGT + 1;
    CALL ROUTINE EXAMINE TO COMPARE THE KEYS OF THE NODES
    WAY := EXAMINE (PP, P);
SET DIRECTION IN WHICH TO FOLLOW IN TREE, IF WAY NE 0
IF WAY NE 0 THEN P := P^WP[WAY]
* IF WAY = 0 THEN NODES KEYS ARE EQUAL THEN SET FUNCTION EQUAL TO NODE ADDRESS AND DELETE CREATED NODE POSITION.
ELSE BEGIN
  KINSEARCH := P;
  CHK := FALSE;
  $ := NIL;
  RELEASE(PP);
END;

IF NODE NOT FOUND THEN INSERT IT INTO TREE
IF CHK THEN BEGIN
  FOR I := 1 TO KSN DO PP := P^PP[WAY] := NIL;
  TEST IF PATH IS NIL AND IF SO LET ROOT := ADDRESS
  IF ROOT EQ NIL THEN ROOT := PP
  ELSE BEGIN
    SAVEN := P
    END;

INCREMENT NODECOUNT
NODECOUNT := NODECOUNT + 1;

ADD INTERNAL PATH LENGTH FOR CURRENT NODE
TO TOTAL INTERNAL PATH LENGTH OF TREE
PLENGTH := PLENGTH + HGT;
END; KINSEARCH
PROCEDURE INITIALIZE;

PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS OF VARIABLES AND ARRAYS.

VAR

I, J : INTEGER;
BEGIN INITIALIZE.

FREE := NIL;
MID := KSO DIV 2;
SHID := MID + 1;
XNUM := NUMFREE;

* INITIALIZE RANDOM NUMBER ARRAY *

FOR I := 1 TO KK DO
  FOR J := 1 TO MAXSIZE DO
    RANDUM[I, J] := J;

* INITIALIZE INDIRECT ADDRESS ARRAY *

FOR I := 1 TO MAXSIZE DO
  RAND[I] := I;

* INITIALIZE SEEDS *

SEED := 33433;
FOR I := 1 TO KK DO
BEGIN
  SEED := SEED + 1;
  IF NOT ODD(SEED) THEN SEED := SEED + 1;
END;

* INITIALIZE STATISTICS ARRAY *

FOR I := 1 TO NUMINTS DO
  FOR J := 1 TO 3 DO
    STAT[I, J] := 0;
END INITIALIZE.
PROCEDURE BUILD_TREE;

* PURPOSE: TO BUILD UP A TREE AND COLLECT THE REQUIRED STATISTICS.

VAR
INC: INTEGER;
BEGIN BUILD_TREE

- INITIALIZE

ROOT := NIL;
NODECOUNT := 0;
HEIGHT := 0;
LENGTH := 0;
NODESIZE := INTERVAL;
NUMNODES := 1;
TIME := 0;

- PERMUTE THE ELEMENTS IN THE RANDOM NUMBER ARRAYS

PERMUTE(RANDRUM, SEED, MAXSIZE);

- PERMUTE THE VECTORS OF KEYS USING THE INDIRECT ADDRESS ARRAY

SHUFFLE(RANDKEYS, SEED, MAXSIZE);

- CALL FOR CURRENT CLOCK TIME AT START OF TREE

T1 := CLOCK;

- INSERT THE NODES TO BUILD TREE

FOR INC := 1 TO MAXSIZE DO BEGIN

ADD := KINSERT(RANDP, INC);

- IF THE NODE ALREADY EXISTS OUTPUT A MESSAGE

IF ADD NE NIL THEN
BEGIN

WRITE("** THE NODE HAVING KEYS **");
WRITE(ADD);
WRITE(" ALREADY EXISTS.");
END
ELSE IF HGT GT HEIGHT THEN HEIGHT := HGT;

END

REMEMBER TO SAVE THE TEXT OF THE MESSAGE TO THE FILE.
IF INC = NODERIZE THEN BEGIN
  CALL FOR CURRENT CLOCK TIME ↓
  T2 := CLOCK;
  SUBTRACT TO GET BUILDING TIME ↓
  STIME := T2 - T1;
  TIME := TIME + STIME;
  GATHER STATISTICS ↓
  ACCUMULATE ↓
  INCREMENT NODERIZE ↓
  NODERIZE := NODERIZE + INTERVAL;
  CALL FOR CURRENT TIME ↓
  T1 := CLOCK;
END; END;

END; BUILDTREE ↓
PROCEDURE PRINT(P: TREE):
BEGIN IF P NE NIL
  THEN BEGIN
    WRITEln(P,E);
    WRITEln(P,E, BASIC: E, KD: E, DIM TREE);
    WRITEln(P,E, THERE ARE E NODECOUNT: NODES IN THIS TREE);
    WRITEln(P,E, THE ROOT OF THIS TREE IS E);
    WRITEln(P,E);
    WRITEln(P,E);
    WRITEln(P,E);
    WRITEln(P,E);
    KD: DISPLAY(P, 6, 4, 5);
    WRITEln(P,E);
  END;
END;

PROCEDURE PRINTHEAD:
  PURPOSE: TO PRINT A HEADING
  BEGIN PRINTHEAD:
    WRITE(E, STATISTICS FOR A QUAD TREE OF E, KD: E, DIMENSIONS: E);
    WRITEln(E)
    WRITEln(E, NUMBER OF TREES BUILT: E, NUM: E);
    WRITEln(E)
    WRITEln(E, MAX SIZE: E, BUILDING TIME: E);
    WRITEln(E, INTERVAL PATH LENGTH MIN MAX: E);
    WRITE(E)
    WRITEln(E, MIN MAX MIN MAX E);
  END; PRINTHEAD
BEGIN *KINF OC*

LINELIMIT(OUTPUT,2500);

*CALL PROCEDURE INITIALIZE TO PERFORM INITIALIZATIONS.*
 INITIALIZE;

*PRINT HEADER*
 PRINT HEADER;

*COLLECT STATISTICS*

FOR I B U I L D 1 = 1 TO NUMTREES DO
 BEGIN
  *BUILD TREE AND COLLECT STATISTICS ON THAT TREE*
  BUILDTREE;
  KD I S P L A Y ( R O O T , 8 , 4 , 8 ) ;
  *CALL DESTROY TO BREAKDOWN THE TREE*
  DESTROY ( R O O T ) ;
 END;

*OBTAIN AVERAGES AND OUTPUT RESULTS*

NODESIZE := INTERVAL;

FOR I := 1 TO NUMINTS DO
 BEGIN
  STAT ( I , 1 ) := STAT ( I , 1 ) / X A U M ;
  STAT ( I , 4 ) := STAT ( I , 4 ) / X A U M ;
  STAT ( I , 7 ) := STAT ( I , 7 ) / X A U M ;
  PRINTOUT ( I ) :
  *INCREMENT NODESIZE*
  NODESIZE := NODESIZE + INTERVAL;
 END;

END. *KINF OC*
COMMON ROUTINES
FUNCTION RANDOM(A,B:REAL; VAR Y:INTEGER): REAL;

**PURPOSE:** RANDOM generates a pseudo-random number in the open interval (A,B) where A < B.

**DESCRIPTION:** The procedure assumes that integer arithmetic up to 32768 * 67108864 = 236719136768 is available. The actual parameter corresponding to Y must be an integer, must be an odd integer within the limits 1 to 67108864 inclusive, and at the first call of the procedure its value must be a | random value. If a correct sequence is to be generated, the value of the integer must not be changed between successive calls to the function. (W. C. Pike, I. D. Hill, Algorithm 266 Comm. ACM 8 (Oct. 1965), p649).

BEGIN RANDOM
Y := 3125 * Y;
Y := Y - (Y DIV 67108864) * 67108864;
RANDOM := Y / 67108864.0 - (B - A) + A;
END RANDOM
PROCEDURE PERMUTE(VAR A:KEYARRAY;VAR SEED:INT;VAR N:INTEGER);


VAR I,J,K:INTEGER;
SAVE:INTEGER;

BEGIN PERMUTE;
FOR I:= 1 TO KK DO
FOR J:= N DOWNTO 1 DO
BEGIN
K := TRUNC(J-RAND(0.0,1.0,SEED(I))) + 1;
SAVE := A[I,J];
A[I,K] := SAVE;
END;
END; PERMUTE;
PROCEDURE SHUFFLE(VAR X:INT; VAR SEED:INTEGER; N:INTEGER):

PURPOSE: SHUFFLE applies a random permutation to the sequence X[1],
   I=1,...,N in such a way that after N calls of the procedure,
   RANDOM the elements X[1] for I=1,...,N are a random permutation
   of the original N elements X[1] where I=1,2,...,N taken N at
   a time.

VAR J,K : INTEGER;
SAVE : INTEGER;
BEGIN
   FOR J := N DOWNTO 1 DO
      BEGIN
         K := FLOOR(J - RANDOM(0.0,1.0,SEED)) + 1;
         SAVE := X[J];
         X[J] := X[K];
         X[K] := SAVE;
      END;  // SHUFFLE
END;  // CLASSOVERFLOW

PROCEDURE CLASSOVERFLOW;
BEGIN
   WRITELN(0, --- CLASSOVERFLOW ---> E);
END;  // CLASSOVERFLOW
FUNCTION LENGTH(V:INTEGER):INTEGER;

PURPOSE: TO DETERMINE THE SIZE OF THE PARAMETER V.

PARAMETER(S):
V--THE INTEGER whose LENGTH is TO BE DETERMINED FOR OUTPUT.

VAR
  SIZE: INTEGER;

BEGIN
  SIZE := 1;
  WHILE V GE 10 DO
    BEGIN
      V := V DIV 10;
      SIZE := SIZE + 1;
    END;
  LENGTH := SIZE;
END : LENGTH
PROCEDURE RITE(KKEYS: KTREE);

PURPOSE: TO OUTPUT THE K KEYS OF A NODE, SAY KKEYS.
PARAMETER(S): KKEYS--POINTER TO THE NODE WHOSE KEYS ARE TO BE OUTPUT
VAR: KPOS, I: INTEGER;
BEGIN
  WRITE;
  K COUNTS THE NUMBER OF CHARACTERS OUTPUT
  K := 0;
  WRITE(\034E);
  REPEAT UCCP TO OUTPUT THE K KEYS.
  FOR KPOS := 1 TO K DO
    BEGIN
      IF KPOS NE 1 THEN BEGIN
        OUTPUT A , BETWEEN THE KEYS
        WRITE(\034E);
        K := K + 1;
      END;
      OUTPUT A KEY
      I := LENGTH(FANDNUM(KPOS, KKEYS . KEY));
      WRITE(FANDNUM(KPOS, KKEYS . KEY) i);
      K := K + 1;
    END;
  WRITE(\034E);
END;
PROCEDURE ACCUMULATE;

* PURPOSE: TO GATHER STATISTICS FOR DIFFERENT NUMBERS OF NODES *

BEGIN ACCUMULATE

   * ACCUMULATE STATISTICS *

   STATS[NUMNODES,1] := STATS[NUMNODES,1] + TIME;
   IF ((STATS[NUMNODES,2] EQ 0) OR (TIME LT STATS[NUMNODES,2])) THEN
   STATS[NUMNODES,2] := TIME;
   IF ((STATS[NUMNODES,3] EQ 0) OR (TIME GT STATS[NUMNODES,3])) THEN
   STATS[NUMNODES,3] := TIME;
   IF ((STATS[NUMNODES,5] EQ 0) OR (PLENGTH LT STATS[NUMNODES,5])) THEN
   STATS[NUMNODES,5] := PLENGTH;
   IF ((STATS[NUMNODES,6] EQ 0) OR (PLENGTH GT STATS[NUMNODES,6])) THEN
   STATS[NUMNODES,6] := PLENGTH;
   STATS[NUMNODES,7] := STATS[NUMNODES,7] + HEIGHT;
   IF ((STATS[NUMNODES,8] EQ 0) OR (HEIGHT LT STATS[NUMNODES,8])) THEN
   STATS[NUMNODES,8] := HEIGHT;
   IF ((STATS[NUMNODES,9] EQ 0) OR (HEIGHT GT STATS[NUMNODES,9])) THEN
   STATS[NUMNODES,9] := HEIGHT;

   * INCREMENT NUMNODES *

   NUMNODES := NUMNODES + 1;

END; ACCUMULATE

-----------------------------------------------------------------

PROCEDURE PINTOUT(I:INTEGER);

* PURPOSE: TO OUTPUT THE RESULTS *

BEGIN PINTOUT

   WRITE(E0, 'E,NODESIZE',I, 'E, E);
   WRITE(STATS[I,1], 'S,12', STATS[I,2], 'S,17', STATS[I,3], 'S,610);
   WRITE(STATS[I,4], 'S,110', STATS[I,5], 'S,17', STATS[I,6], 'S,17', STATS[I,7], 'S,610);
   WRITELN(STATS[I,7], 'S,110', STATS[I,8], 'S,17', STATS[I,9], 'S,610');

END; PINTOUT


APPENDIX C

The COMPARE Routine.

This routine compares two k-tuple keys by first concatenating the k-attributes of each and then comparing. As discussed in Chapter III, this technique of concatenating the k-attributes is commonly used for comparing k-tuple keys in a Binary Search Tree. Because of the long execution time required for this routine, a simpler comparison routine was used and is given in Appendix B.
FUNCTION COMPARE(FIRST, SECOND: NODE; JJ: INTEGER): INTEGER;

PURPOSE: TO COMPARE THE KEYS OF FIRST AND SECOND STARTING AT
THE JJ-TH KEY IF JJ GT 0 AND TO RETURN THE VALUE OF
THE BOOLEAN FUNCTION COMPARE:
1: IF FIRST = SECOND
2: IF FIRST > SECOND
3: IF FIRST < SECOND

PARAMETER(S):
FIRST—POINTER TO THE NODE TO BE INSERTED
SECOND—POINTER TO A NODE IN THE TREE TO BE COMPARED TO FIRST
JJ—INTEGER INDICATING WHERE TO START THE CYCLIC REARRANGEMENT
COMPARED WILL BE AT THE START OF THE CONCATENATED STRING.

THE METHOD USED TO COMPARE THE TWO NODES IS TO CONCATENATE
EACH KEY IN TURN AND COMPARE THE APPROPRIATE ELEMENTS.

VAR
CHKA, CHK2: BOOLEAN;
OTX, IXY, FLAG1: INTEGER;
IJ, IJA, KK: INTEGER;
AB, AALFA: ARRAY[1..LENKEYS] OF CHAR;

FUNCTION NEXT(AB: AALFA; VAR IJ: INTEGER; VAR XY: CHAR): BOOLEAN;

PURPOSE: TO UNPACK THE NEXT KEY OF NODE *AB* INTO THE CHARACTER ARRAY
*XY* AND TO RETURN THE VALUE OF NEXT
FALSE WHEN ALL KEYS ARE EXHAUSTED
TRUE OTHERWISE.

PARAMETER(S):
AB—ARRAY CONTAINING THE KEYS OF THE NODE
IJ—INTEGER INDICATING THE CURRENT KEY POSITION IN AB
XY—CHARACTER ARRAY IN WHICH A KEY AT A TIME IS UNPACKED.
TYPE
  PCHAR = PACKED ARRAY[1..10] OF CHAR;
  KTYPE = RECORD
    CASE BOOLEAN OF
      TRUE(IN: ALFA):;
      FALSE(IN: PCHAR):;
    END;
END;

VAR
  TPINT : KTYPE;
  Z: 1 CHAR;
  I, J, JJ: INTEGER;
BEGIN
  IF IJ LE KK THEN NEXT := FALSE
  ELSE BEGIN
    NFEXT := TRUE;
    IJ := IJ + 1;
    TPINT.IN := AB[IJ];
    UNPACK(TPINT, ALF, Z, I);
    J := 0;
    PERFOR J := J + 1;
    UNTIL((Z(J1 NE Z[J]) OR (J EQ 10)));
    I := 1;
    FOR JJ := J TO 10 DO
      BEGIN
        Xy[I] := Z(JJ);
        I := I + 1;
      END;
    FOR JJ := J TO 10 DO
      Xy(JJ) := Z[J1];
      JJ := JJ + 1;
    END
END; /* NEXT */
BEGIN. COMPARE

IF JJ > 0 THEN REARRANGE THE KEYS STARTING WITH THE JJ ELEMENT
OTHERWISE LEAVE KEYS IN ORIGINAL ORDER TO BE TESTED.

IF JJ < 0 THEN
BEGIN
J := JJ;
FOR I := 1 TO KK DO
BEGIN
A[I] := RANDNUM(I;FIRST+KEY); 
B[I] := RANDNUM(I;SECOND+KEY);
IF J LT KK THEN J := J + 1
ELSE J := 1;
END;
 END ELSE
FOR I := 1 TO KK DO
BEGIN
A[I] := RANDNUM(I;FIRST+KEY); 
B[I] := RANDNUM(I;SECOND+KEY);
END;

INITIALIZE COUNTERS AND FLAGS

FLAG := 0;
I := 0;
J := 0;
X := X;
Y := Y;
K := 0;
K1 := 0;

REPEAT THE FOLLOWING LOOP UNTIL BOTH SETS OF KEYS ARE EXHAUSTED
OF A CHARACTER IN ONE CODE IS FOUND TO BE > THE CORRESPONDING
CHARACTER IN THE OTHER CODE.

CHKA := NEXT(A;I;X);
CHKB := NEXT(B;I;Y);
REPEAT
IF NEXT CHARACTER IS BLANK OR THE 11 CHARACTERS OF THAT KEY
HAVE BEEN TESTED GET NEXT KEY AND RESET COUNTER.


IF ( (X[K]) EQ E) OR (KA GT 10) ) THEN BEGIN
    CHKA := NEXT(A,I,X);
    KA := 1;
END:

IF ( (Y[K)] EQ E) OR (KB GT 10) ) THEN BEGIN
    CHKB := NEXT(B,J,Y);
    KB := 1;
END:

IF NEITHER NODER KEYS ARE EXHAUSTED THEN COMPARE THE CORRESPONDING CHARACTERS IN THE CURRENT KEYS.

IF (CHKA AND CHKB) THEN BEGIN
    IF X(KA) GT Y(KB) THEN FLAG := 1
    ELSE IF Y(KB) GT X(KA) THEN FLAG := 2;
    KA := KA + 1;
    CTX := CTX + 1;
    CY := CTY + 1;
END:

UNTIL ( (FLAG GT 0) OR (NOT (CHKA AND CHKB)) ) ;

COUNT THE REMAINING CHARACTERS IN THE NODES KEYS.

IF CHKA THEN
    REPEAT
    IF ( (X[K]) EQ E) OR (KA GT 10) ) THEN BEGIN
        CHKA := NEXT(A,I,X);
        KA := 0;
        END
    ELSE CTX := CTX + 1;
    KA := KA + 1;
UNTIL (NOT CHKA);

IF CHKB THEN
    REPEAT
    IF ( (Y[K]) EQ E) OR (KB GT 10) ) THEN BEGIN
        CHKB := NEXT(B,J,Y);
        KB := 0;
        END
    ELSE CTY := CTY + 1;
    KB := KB + 1;
UNTIL (NOT CHKB);
TEST IF EITHER NODE HAS A GREATER NUMBER OF CHARACTERS AND SET FLAG ACCORDINGLY.

IF CTX GT CTY THEN FLAG := 1
ELSE
IF CTY GT CTX THEN FLAG := 2;
COMPARE := FLAG;
END; /*COMPARE*/