01 Aug 1984

A Simple Method for Organizing Nearly Optimal Binary Search Trees

Joy L. Henderson

John R. Metzner

Missouri University of Science and Technology

A SIMPLE METHOD FOR ORGANIZING NEARLY OPTIMAL BINARY SEARCH TREES

Joy L. Henderson* and John R. Metzner

CSc-84-13

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401 (314)-341-4491

*This report is substantially the M.S. thesis of the first author, completed August, 1984.

ABSTRACT

Improving the efficiency of retrieving information concerns users of computer systems involved in many applications. One way of addressing this concern is to organize a sorted sequence into a binary search tree. Knuth's Algorithm K is a bottom-up organization algorithm that always constructs a binary tree which minimizes average search time. However, the cost of executing Algorithm K is prohibitive for a large tree. The aim of this work is to find a less costly method of organizing sorted sequences into nearly-optimal binary search trees.

We present a top-down organization method which yields better average search times than top-down methods already available, specifically height-balancing and weight-balancing. The variation in access frequency among the members of a sequence is used to recommend specific values for some of the parameters in this new method of organization.

The new method improves considerably on the cost of organization as opposed to the cost of using Algorithm K while producing trees whose average search times are close to minimal. The new algorithm yields an average search time that is usually within 1% of the minimal average search time and for every case attempted has been no worse than 1.5% larger than minimal.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figures Page

LIST OF TABLES

I. INTRODUCTION

Improving the efficiency of retrieving information from a database concerns users of computer systems involved in many applications. In today's computerized society, faster response time is an expected luxury in some systems while a necessary characteristic of other systems. There are many ways of addressing this concern.

Ordering a table of records or indexes of records inherently simplifies the process of searching for a particular record. Without ordering or sorting a table, the only practical choice for searching the table is sequential scanning. However, when searching an ordered table a binary search method is more efficient than sequential scanning. The basic idea behind binary search is to first compare the key being searched for to the middle key in the table. "The result of this probe tells which half of the table should be searched next, and the same procedure can be used again, comparing K [the key being searched for] to the middle key of the selected half, etc."[1, pp. 406-407] It is easier to understand binary search when it is thought of as a "binary decision tree".[1, p. 409] To take this idea further, "...any algorithm for searching an ordered table of length N by means of comparison can be represented as a binary tree ..."[1, p. 409] Knuth presents several variations of

binary search which are obviously intended for use when searching sequentially placed records. However, if the table is being continually updated, as most tables are, "...we might spend more time maintaining it than we save binary searching it."[1, p. 423] This brings us to the concept of using explicit binary tree structures.

> "The use of an explicit binary tree structure makes it possible to insert and delete records quickly, as well as to search the table efficiently. As a result, we essentially have a method which is useful both for searching and for sorting."[1, p. 423]

Figure 1 is an example of a binary search tree where the records contain information concerning an airline's daily scheduled flights. The nodes are labeled using the airline's flight numbers.

Deciding to use explicit binary tree structures gives the user another decision to make: How does the user want to organize the sorted database into a binary tree? Once sequential placement is no longer a factor, there are other characteristics of the database which may be considered. Specifically, these are the relative frequency of access of a given record and the relative frequency of an unsuccessful search being terminated when reaching a given record during the search process. D.E. Knuth has developed an algorithm which uses these two characteristics and always constructs a binary tree structure which results in minimal average search time. Why, then, is any further research necessary? The cost of

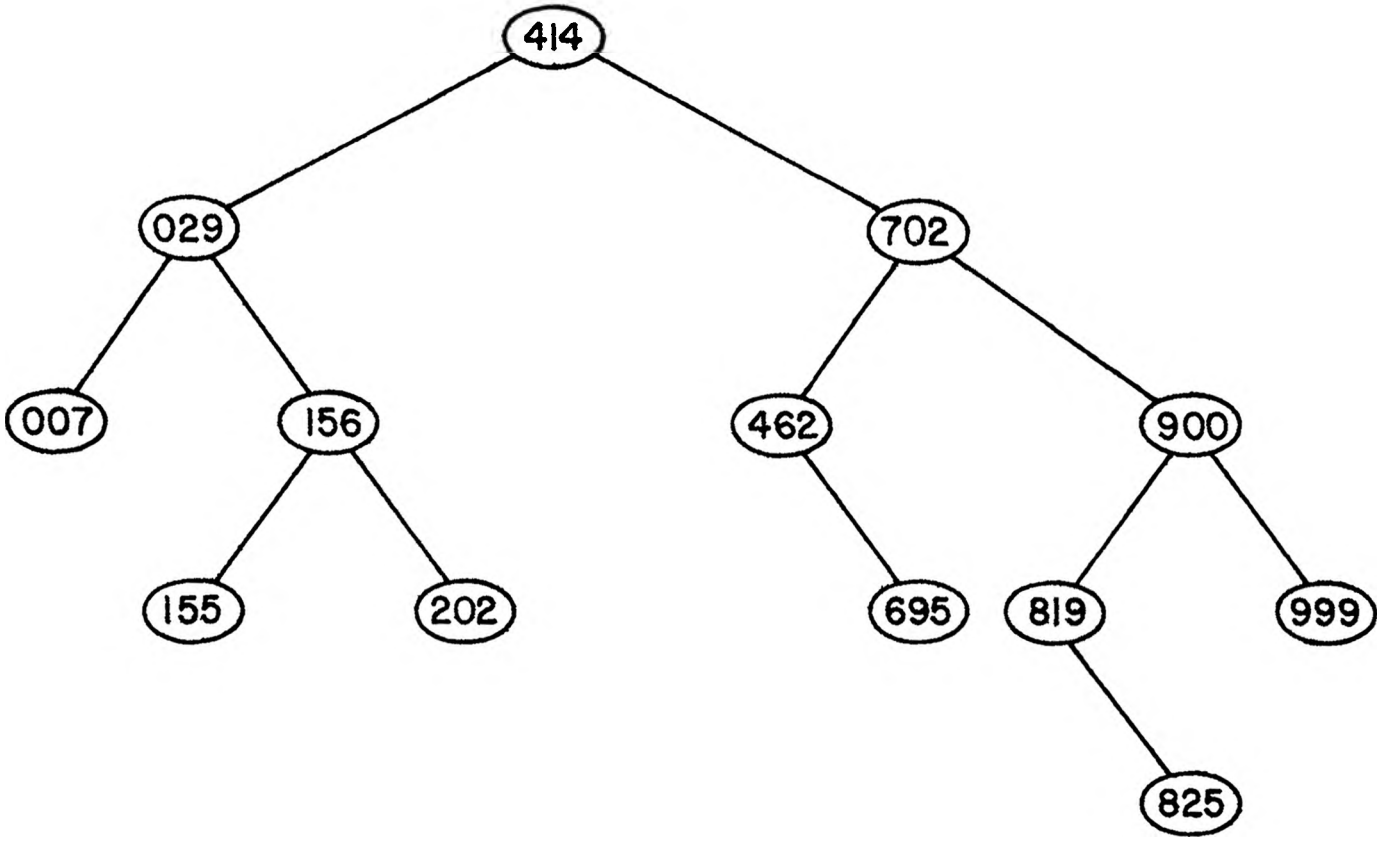

Figure 1. A Binary Search Tree Whose Keys are an Airline's Flight Numbers

Knuth's method of organization is prohibitive when a table is very large. This cost is even more restrictive when dealing with a volatile file which requires frequent reorganization. The focus of this paper is to find a method of organizing sorted tables into binary trees which results in nearly minimal average search time, yet does not have a restrictive cost of reorganization.

In order to clarify the remainder of this discussion, a few definitions will be given.[1]

binary search tree - This is an ordered group of elements organized so that one element is the root and the remaining elements are divided into two trees called subtrees of the root. Each subtree is either empty or consists of a root and two resulting subtrees.

level - The number of arcs between a node and the root of the tree.

n - The number of search keys (or elements) in a given database (or binary tree).

k(i) - The ith search key in a database whose keys are ordered alphabetically or numerically. $k(1)<k(2)<\ldots<k(n)$

t(i,j) - The subtree consisting of elements k(i+1) to to k(j) given the condition $0 \leq i \leq j \leq n$.

r(i,j) - The root of the subtree t(i,j).

p(i) - The relative frequency that k(i) is the search argument for any given search of t(i,j). This value is also referred to as relative frequency or relative frequency of access.

q(i) - The relative frequency "that the search argument lies between k(i) and k(i+1). (By convention, q(0) is the relative frequency that the search argument is less than k(1) and q(n) is the relative frequency that the search argument is greater than k(n).)"[1, p. 434]

c(i,j) - The cost of subtree t(i,j) as a function of relative access frequency. (Sum of all p(i)*(level+1) and all q(i)*level.)

w(i,j) - The weight of the subtree t(i,j), found by summing all p(i), (i=i+1 to j), and q(i), (i=i to j).

AST - Average search time (AST) is the average cost in number of accesses of a successful search for a given tree: c(0,n)/w(0,n).

Here is an illustration using some of the preceding terms in order to clarify their meanings. A binary search tree of thirteen elements, t(0,13), with node six as its root, r(0,13)=6, would have two subtrees, t(0,5) and t(6,13). Figure 2 shows a picture of what this search tree might

look like. Table I lists example values for all p(i) for this tree. Table II lists sample corresponding roots, weights, and costs.

There will be two main variables used in differentiating various organization methods. The first and most obvious indicator of whether or not a particular method is appropriate for a given system is the average search time of the resulting binary tree. The second variable is the cost of determining the exact organization of the tree. Some systems can afford to allow ample time for reorganizing the trees used in database searching. Other systems cannot afford to be out of service for the time necessary for a full reorganization yielding optimal efficiency of retrieval. Reorganization is done when additions and deletions of records to the database have degraded the average search time to an unsatisfactory level. Those systems which this research concerns are those who need to decide how much they are willing to give up in retrieval time in order to keep the cost of reorganization down.

In the experimentation for this research it is assumed that estimates of the relative access frequencies, p(i)'s, are available. This is a realistic assumption when working with a database which has been in use for some time. These may be actual values gathered over a period of time, or educated guesses by those persons who

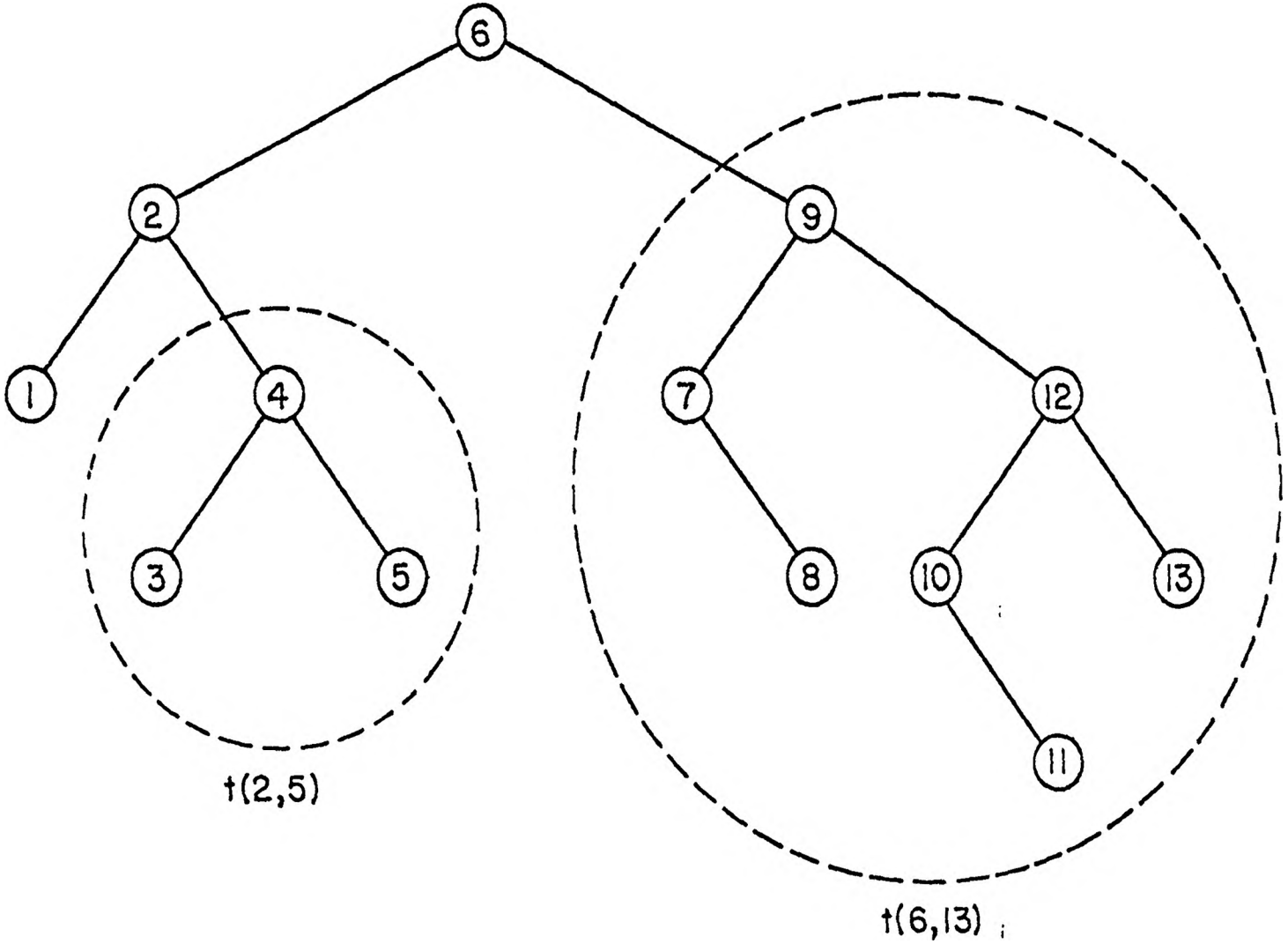

Figure 2. Binary Search Tree with Labeled Subtrees

TABLE I

SAMPLE RELATIVE ACCESS FREQUENCIES

FOR A BINARY SEARCH TREE OF SIZE THIRTEEN

NODE	p(i)
1	4.0
2	7.0
3	3.0
4	9.0
5	6.0
6	2.0
7	14.0
8	1.0
9	9.0
10	7.0
11	2.0
12	3.0
13	3.0

TABLE II

SAMPLE ROOTS, WEIGHTS, AND COSTS
FOR A BINARY SEARCH TREE OF SIZE THIRTEEN

ROOTS	WEIGHTS	COSTS
r(0,13) = 6	w(0,13) = 70.0	c(0,13) = 214.0
r(6,13) = 9	w(6,13) = 39.0	c(6,13) = 84.0
r(0,5) = 2	w(0,5) = 29.0	c(0,5) = 60.0
r(2,5) = 4	w(2,5) = 18.0	c(2,5) = 27.0
r(9,13) = 12	w(9,13) = 15.0	c(9,13) = 29.0

have been working with the database. In some database applications one key in a table and its corresponding p(i) represents a group of records. For instance, many insurance files contain information on several members of a single family instead of having a separate file for each family member. In this case, the table being organized is a table of indices, k(i)'s, and the corresponding p(i)'s.

Knuth has presented an algorithm for organization of a sorted table into a binary tree which yields minimal average search time.[1] However, the cost of this organization is prohibitive when working with large tables ($n \geq 100$). We have experimented with a heuristic-based tree organizing method to see how it compares to Knuth's optimal binary tree organization. Our aim was to find a method that organizes a table into a binary tree which yields near-minimal average search time while costing less than Knuth's algorithm. A description of this experimentation and the results are presented in this paper.

II. REVIEW OF LITERATURE

A. BOTTOM-UP ORGANIZATION METHODS

D.E. Knuth has presented an effective method of organizing a sorted database into an optimum binary tree. [1] This is a bottom-up procedure which Knuth calls Algorithm K. This algorithm first examines all pairs of adjacent elements in the sorted database to determine which element in each pair should be the root of that subtree (consisting of two elements) in order to achieve the lowest cost. Next, using the results of the two-element subtree root-search and the fact that "...all subtrees of an optimum binary tree are optimum."[1, p. 435], Algorithm K finds the roots of all adjacent triples. This process continues for all groups of four elements, five elements, etc. until the root for the entire tree (a group of n elements) has been found. Algorithm K has been described as a "...computation procedure which systematically finds larger and larger optimum subtrees."[1, p. 435] This method of organization will be described in more detail later in this paper (see Section III).

Although Algorithm K always produces an optimum binary tree, there is a drawback to using it for organization. Total running time of $O(n^2)$ is required to determine an optimum binary tree. This says that the run time increases in proportion to n squared. For instance,

given that the number of records in table A is n and the number of records in table B is 3n, table B requires approximately nine times as long to organize as table A does. If a small database is involved ($n \leq 100$), this run time is not necessarily a restriction. However, for larger trees most users need to look at alternate approaches to the problem of determining tree organization.

T.C. Hu and A.C. Tucker presented an algorithm for the special case where all p(i)=0.[2] (This says that all inquiries are unsuccessful.) In an extensive proof, Hu and Tucker explain how to first build an optimal binary tree disregarding alphabetical order using a "T-C level-by-level construction"[2, p. 520] and then convert this tree into an optimal binary tree in alphabetical order. The T-C stands for tentative connecting. Two nodes can be combined into a subtree (i.e. have a common father) only if they are T-C nodes. Two nodes are considered T-C nodes if the nodes are adjacent or their separation is only by internal nodes (roots of subtrees). The T-C level-by-level algorithm builds the binary tree in a bottom-up fashion, building a subtree of the pair of T-C nodes with minimum weight first. This method "...combines all nodes on the lowest level of the T-C tree first, then all nodes on the next-to-lowest level, and so on."[2, p. 520] (hence the name T-C level-by-level construction). A key theorem

in Hu and Tuckers' proof basically says that for every tree in the "...class of all T-C level-by-level forests (including trees)...there is an alphabetic forest (or tree) of the same cost."[2, p. 521] This theorem is the basis for the second phase of Hu and Tucker's algorithm, the conversion of an unordered optimal binary tree into an ordered optimal binary tree.

As in Knuth's Algorithm K, total running time of the Hu and Tucker algorithm is a restriction when a large database is used. The implementation presented required $O(n^2)$ operations. (In an ending note, they mention that Knuth suggests an implementation which "...needs only O(n log n) operations when suitable data structures are employed."[11] No details are presented on this implementation.)

B. TOP-DOWN ORGANIZATION METHODS

There are many ways of approaching a top-down tree structuring. The method which seems to be the natural choice is to simply choose the record with the largest frequency of the tree to be the root. Then choose the record with the largest frequency in subsequent subtrees as the subtrees' respective roots until the organization is complete. Reingold and Hansen call this the "monotonic rule."[3] However, practical experiments have shown that the monotic rule does "...not produce acceptable nearly optimal trees."[4, pp. 307-308] Some researchers have

gone as far as to say that this method results in an average search time which "...on the average...is no better than a tree constructed at random."[5, pp. 291-292]

Obviously, there are better top-down methods of organization. The balancing rule chooses each r(i,j) in order to balance as nearly as possible the weight of the subtrees on either side of the root.[5] It has been suggested that trees constructed using the balancing rule (weight-balanced trees [5]) are optimum when all q(i)=0 [6, pp. 142-144], but this is not the case.[7] Two closely related classes of organization are bisection trees and min-max trees.

Allen describes the construction of Mehlhorn's bisection trees as follows:

> "The root of the entire tree is chosen closest to the 50th percentile of the cumulative weight distribution. Its left and right sons are chosen closest to the 25th and 75th percentiles, respectively, and so on." [8, p. 259]

Min-max trees, introduced by Bayer, also use the weight of the tree during organization. The root of the initial tree is chosen in order to minimize the maximum weight of the resulting left and right subtrees. This procedure is repeated until all roots have been found.[8]

There are situations where all three of the previously discussed classes of trees - weight-balanced trees, bisection trees, and min-max trees - may result in the same binary tree. Yet, using the informal definitions

in this paper the resulting tree in any given class "...is not uniquely specified for certain weight distributions." [8, p. 259] These definitions are acceptable for weight-balanced trees and bisection trees. However, since Bayer "...makes a particular choice in his definition of min-max trees..."[8, p. 259], Allen's term "essentially min-max" will be used as a label for min-max trees which satisfy the more informal definition.[8]

When dealing with trees which have uniform relative frequencies (p(i)), any of these three methods perform an acceptable job of organizing a tree resulting in nearly optimal AST. However, as the relative values become more skewed, average search time becomes less predictable. Allen proves that for none of the three classes of organization methods (weight-balanced trees, bisection trees, and essentially min-max trees) is the cost of the tree bound.[8] The maximum value of the cost of a tree organized using these methods cannot be restricted. Therefore, a maximum value for average search time cannot be assumed.

Reingold and Hansen discuss another simple but relatively effective method of organization based solely on the number of elements in a tree or subtree (as opposed to relative frequency).[9] Their height-balanced trees are constructed by choosing the root of the tree such that the height of the resulting left and right subtrees

differs by no more than one. The process is repeated for subsequent subtrees. The average search time for a height-balanced tree is equivalent to that of a binary search which is given by Lewis and Smith [10] (L is average search time):

$$L=\log_2 (n+1)-1,\ n>50$$

when the p(i)'s are all equal. As a matter of fact, for uniform access frequencies, the height-balanced and weight-balanced trees are nearly equivalent (often they are the same trees). In this case, height balancing is the better choice due to the absence of comparisons needed to organize the tree.

Another top-down method of organization is proposed by Walker and Gotlieb.[11] Their approach requiring accurate estimates of all p(i) and q(i) combines a top-down method with Knuth's Algorithm K to yield close-to-minimal average search time. They use an example application of the author index of a library catalog. In this example the relative access frequency is not expected to change much over a short period.

This method of constructing a binary search tree chooses the largest p(i) in the neighborhood of the centroid, the key whose left and right subtrees are most equal in weight. Notice that the centroid is the key which would be chosen as the root when organizing using the balancing rule. If a subtree is less than or equal to

size N_0 (a parameter in this algorithm), an optimal tree is structured using Algorithm K. F is the parameter which determines what the search width for the root around the centroid is. F is greater than or equal to 1, and the search width is (1/F * w(i,j)) where the subtree currently being searched is t(i,j). The value for F varies according to the ratio of the relative frequency of successful accesses to the database and the relative frequency of unsuccessful accesses.

This top-down algorithm requires time proportional to $n \log_2 n$ to construct a binary search tree of size n. The authors say that an average search time within 1% of minimal can be expected. Knuth states that the results are "reportedly within 2 or 3 percent of the optimum."[12, p. 439]

III. THE MODEL

A. DATA GENERATION

We are going to test a heuristic-based tree organizing method (HEUR) and see how it compares to optimum. In order to clarify the development and experimentation of the new organization algorithm being presented in this paper, it is assumed that all q(i)=0. This is equivalent to assuming that all inquiries into a database are successful. Although this is not a totally realistic assumption, the initial results of experimentation are not biased by this assumption. In practice, the q(i)'s are quite small and very difficult to estimate from experience. All comparison results (binary trees built using Algorithm K) were constructed under the same assumption. While we are ignoring the q(i)'s, reality forbids assuming that the p(i)'s are equal, so we will attempt to model the nonuniformity of the p(i)'s (which we here consider known precisely). In effect we are modeling the expected traffic to the database.

Fifteen sets of data were generated for testing with each different table size. This data consisted of the values of p(i). These data sets, used in experimentation with the new organization algorithm being presented, HEUR, and comparison runs with Algorithm K, were generated in two steps. The first step was the generation of a uniform

distribution of n numbers between zero and one using the multiplicative congruential method of random number generation. Given r(i), b,and m, the (i+1)th random number is produced using the formula:

$$r(i+1) = r(i)*b \pmod{m}$$

The three values, r(i), b, and m, are all positive with r(i)<m. Five different values for r(0) were used. Following the guidelines outlined in Bobillier, Kahan, and Probst's simulation text [13], the following initial values were chosen:

$m = 10^7$

b = 200*16-37 = 3163

Seed1 = r(0) = 1483

Seed2 = r(0) = 1487

Seed3 = r(0) = 2153

Seed4 = r(0) = 3973

Seed5 = r(0) = 4793

Due to the nature of the multiplicative congruential method of random number generation, each r(i) fell between zero and m-1. Division by m was then done to normalize the random numbers. The resulting values were then used to generate variates with desired probability density functions.

Three different probability density functions were used. These will be referred to as "delta-gamma rules". Each of these rules say that delta percent of the accesses

to a table are made to gamma percent of the records in that table. The best known of the three is the 80-20 rule of thumb. This rule "holds approximately for many commercial files,"[14, p. 112] which is why it was chosen as a representative data set for testing. The 80-20 rule says that 80 percent of the accesses to a table are made to the most often used 20 percent of the records in that table.[14] The other two probability density functions are closely related to the 80-20 rule. They are the 70-30 rule and the 90-10 rule. These rules state that 70 percent of the accesses to a table are made to the most often used 30 percent of the records and 90 percent of the accesses are made to 10 percent of the records, respectively.

To produce the record access probabilities as random variables that conform to the "delta-gamma rules," uniform variates were transformed by the function

$$f(x) = BETA*[x^{(BETA-1)}] \quad , \; BETA = \log_{GAMMA} DELTA$$

This produced access probabilities for the records in each set to be organized. Since the set of such values for each tree was not constrained to sum to 1, the values were treated as estimates of record popularity and normalized later by division by the set sum when probabilities were needed.

To allow experimentation with relatively small record sets, the possible biasing effect of having a very popular

record was removed by linearizing the functions in the popular record group. That is, the functions were modified to the definition below.

$$f(x) = \begin{cases} mx+b & 0 \leq x \leq gamma \\ beta*x^{(beta-1)} & gamma \leq x \leq 1 \end{cases}$$

The parameters of these functions for the three experimental cases are given in Table II. The functions are plotted in Figure 3.[15]

B. DESCRIPTION OF ALGORITHM K

Since Algorithm K is being used throughout this paper for comparison values, we present a detailed description of this algorithm (taking into consideration the assumption that all q(i)=0). Some details of the following description would be altered should q(i) not equal zero.

Algorithm K first initializes the cost of all null trees to zero. ("If i=j, t(i,j) is null; else its left subtree is t(i,r[i,j] - 1) and its right subtree is t(r[i,j],j)."[16, p. 436] The weights of all subtrees are also initialized by summing the p(i)'s of all elements in a given subtree. Before going on to the next step, the cost of all 1-node trees are assigned (which is actually the weight of each 1-node tree), and the roots of all 1-node trees are assigned.

The next step of Algorithm K finds the roots of

TABLE III

VALUES USED IN RANDOM NUMBER GENERATION IN TRANSFORMING UNIFORM RANDOM NUMBERS TO DELTA-GAMMA RULES

DELTA	GAMMA	BETA	m	b
.70	.30	.29625	-10.94725	3.97542
.80	.20	.13865	-34.45413	7.44541
.90	.10	.04576	-171.76361	17.58817

$f(x) = mx + b, \quad 0 \leq x \leq \text{gamma}$

$f(x) = \text{BETA} * (x^{(\text{BETA}-1)}), \quad \text{gamma} \leq x \leq 1$

$\text{BETA} = \log_{\text{GAMMA}} \text{DELTA}$

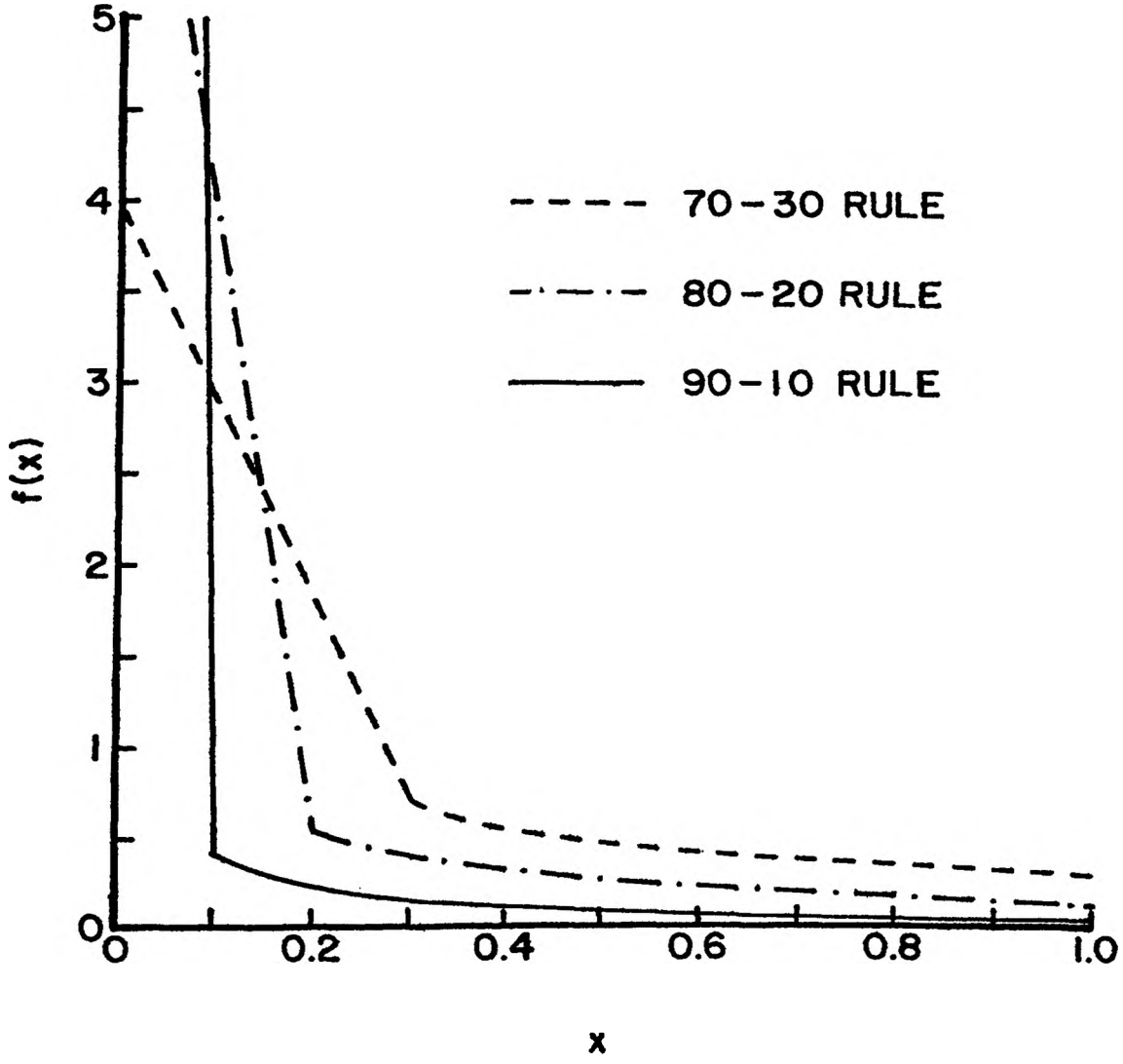

Figure 3. Transformation of Uniform Variates to Conform to the Delta-Gamma Rules

subtrees of 1) size two, 2) size three, 3) etc. until all roots are found. The algorithm starts at t(0,d) (d is the size of the subtree) and proceeds to find the optimal root of this subtree of size d. This is done by finding the node which, when chosen as the root of the subtree, produces the minimum cost possible for that subtree given $r[i,j-1] \leq k \leq r[i+1,j]$ [16]:

$$c[i,j] \leftarrow w[i,j]+\min_{k} (c[i,k-1]+c[k,j])$$

The "monotonicity property" is applied here, eliminating redundant examination of sub-optimal roots. This property says that when an ordered table is organized into an optimum binary search tree, the sets of roots, R(i,j), satisfy

$$R(i,j-1) \leq R(i,j) \leq R(i+1,j) \quad \text{for } j-i \geq 2$$

whenever all p(i)'s and q(i)'s are nonnegative.[16] Knuth gives an illustration: "But if we discover by some means that $R(0,n-1) \geq 5$, it is unnecessary to determine R(i,n) for $1 \leq i \leq 4$ when we compute R(0,n)."[17, p. 19] The roots of all subtrees of size d are found this way (moving next to t(1,d+1), the t(2,d+2), etc.). When all roots for subtrees of size d are found, d is incremented and the process is repeated. This procedure continues until r(0,n) is found.

The unit of "work" done in organizing a tree is measured in number of comparisons required to perform the organization. In Algorithm K, these comparisons arise in

finding the minimum cost of a subtree by testing a the set of roots, R(i,j), to find the root of the subtree. The Computer Science Department at UMR has a copy of the author's implementation of Algorithm K.

IV. DESIGN

The organizing method being presented in this paper, henceforth referred to as HEUR, is based on combining three previously discussed methods of organization. These are the balancing rule (weight-balanced trees), height balancing, and Knuth's Algorithm K.

The first step of HEUR uses the principle of the balancing rule while striving for better results. The median of the weight of the tree is calculated, w(0,n)/2. HEUR then locates the first element (sequentially) whose cumulative frequency of access is greater than or equal to the median (called the midpoint). This element is not automatically chosen to be the root of the tree. HEUR differs at this point from the balancing rule by comparing the p(i)'s within a given percent of the cumulative frequency of the tree to find the root. This search around the median is conducted to find a relatively large p(i) being located near the median. This is not an unlikely occurrence and it is often more efficient to choose the node with a larger p(i) as the root.

The percent of the cumulative frequency which is used to find the search width around the median is a parameter in HEUR. This parameter is called PERCEN. Note that this percent encompasses the entire search width. In other words, the search is conducted where the cumulative relative frequency falls between (median-w(i,j)*PERCEN/2)

and (median+w(i,j)* PERCEN/2), inclusively. The root is being found for the tree or subtree t(i,j). Therefore, w(i,j) is the weight of the subtree currently being searched. (There is a restriction in that the search width must include at least three nodes.)

Before HEUR actually chooses a root from the search width around the median, a temporary frequency reduction process occurs. This reduction somewhat penalizes nodes at a distance from the median and is done in order to prevent nodes whose relative frequency of access is insignificantly larger than one near the median from unnecessarily moving the root away from the median. Since the median normally falls near the center of a table (regardless of the distribution of frequencies), reduction of frequencies is a way of incorporating the idea behind the height-balanced class of trees. Also, empirical evidence from numerous actual organizations using Algorithm K indicates that Algorithm K tends to produce binary trees where the depths of the left and right subtrees (in respect to the root of the tree) do not differ drastically. Walker and Gotlieb's method of organization picked the largest p(i) in a search width as the root, and had no penalty for elements which were at a distance from the median.[18]

The reduction process is based on the idea that the importance of larger p(m)'s, with k(m) being the specific

element being compared, is reduced linearly according to the distance (in relative frequency) from the median in relation to the frequency span of the tree. The slope of the reduction, NU, is a parameter of HEUR. The actual reduction formula is:

$$\text{lambda} = \frac{\text{NU} * (\text{absolute value}[\text{median}-w(0,m)]) * 2}{w(i,j)}$$

$$p(m) = p(m) * (1-\text{lambda})$$

The frequency reduction factor is graphed as a function of the distance from the median in Figure 4.

Once the p(m) values are reduced, the maximum p(m) in the search width is chosen as the root of the tree. In the case of two p(m) having the same value and both of them being the maximum value, the element k(m) which falls closer to the midpoint is chosen as the root. The original values of p(m) are not destroyed. A temporary table consisting of the reduced values of all p(m) within the search width is built for the initial tree and is rebuilt for each subsequent subtree.

After a root is found, HEUR is repeated for all resulting subtrees until a "small" subtree is found. A subtree which contains less than or equal to a given number of elements (KVAL) cues the heuristic to organize the subtree using Algorithm K. KVAL is a third parameter used in HEUR. Since Algorithm K always builds an optimum binary tree and the cost is not restrictive when dealing

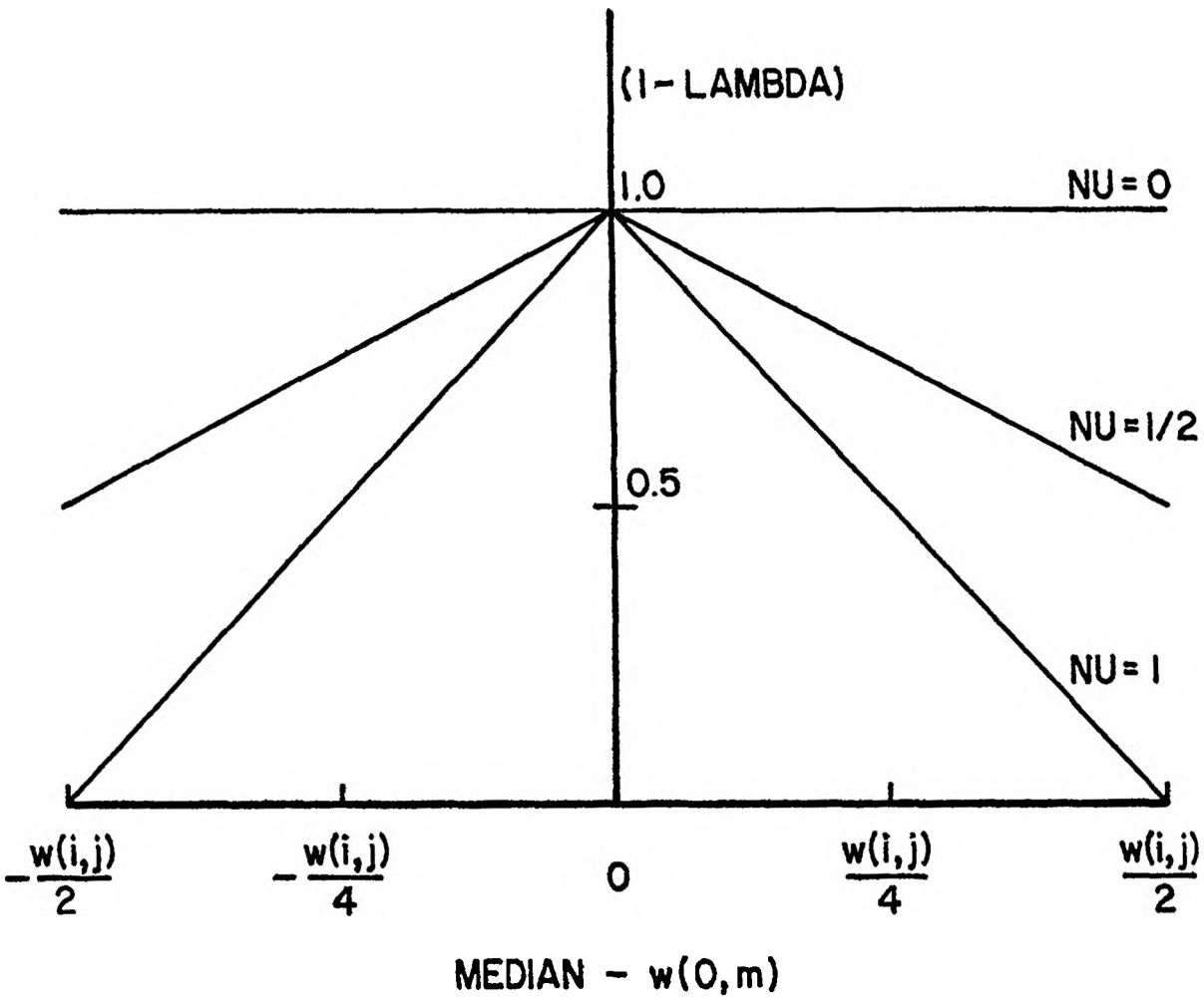

Figure 4. Frequency Reduction Factor as a Function of Distance from the Median

with small trees ($n \leq 100$), it seems logical to use this method to achieve possible improvement in AST. (The actual results of such a decision will be described in Section V.)

Therefore, there are three parameters which must be assigned values when using HEUR to reorganize a table into binary trees:

PERCEN - The percent of the cumulative frequency used to find the search width around the median of any given subtree.

NU - The slope of a linear reduction of all p(i) within the search width of any given subtree.

KVAL - The maximum size of a subtree to be reorganized using Algorithm K. A "cut-over" point telling HEUR to default to Algorithm K.

The Computer Science Department at UMR has a copy of the author's implementation of HEUR.

We have to experiment with varying these in order to be able to recommend values of the parameters to use according to the characterization of a particular database. This characterization is determined by the level of uniformity of the relative access frequencies. The tradeoff between organization cost and average search

time may depend upon how non-uniform the relative access times are. The implications of varying these values will be discussed in the following section.

V. EXPERIMENTAL RESULTS

A. TEST VALUES FOR UNSPECIFIED PARAMETERS

During experimentation, one of the three changing parameters in HEUR needed to be held constant. This choice was made by attempting to pick the parameter which matters the least or is the most predictable. Due to the well-understood nature of Knuth's Algorithm K, KVAL was chosen as the initial parameter to be held constant. The test cases used in Walker and Gotlieb's research showed that increasing KVAL past a certain point decreased average search time very slowly.[18] When the sum of q(i) is less than the sum of p(i), the average search time levels out at KVAL = 15. However there is not much difference in the test results for KVAL = 5, 10, and 15. We conducted several tests previous to picking a value for KVAL which indicated that a value of ten would be a logical initial "best choice" for KVAL. This choice had resulted in significant improvement in AST without sacrificing much run-time during organization.

Therefore, PERCEN and NU were the two remaining parameters to vary. A wide range of values were tested for PERCEN. These values spanned from 10 percent to 35 percent in steps of 5 percent. Previous analyses indicated that PERCEN = .35 was an unnecessarily extreme case, but it was included in order to gain a more accurate

perspective of any patterns established during experimentation.

NU is varied from zero to one, NU = 0, 1/4, 1/3, 1/2, 2/3, 3/4, and 1. Assigning a value of zero to NU is equivalent to eliminating the frequency reduction process. When NU equals one, the significance of a p(i) being greater than p(midpoint) is lessened considerably. If a large value of NU performs well, PERCEN could be reduced (narrowing the search width), as HEUR seems to be looking farther away from the median than required, unnecessarily increasing the number of comparisons.

Each of the previously discussed values of PERCEN and NU were tested on all three frequency distributions (the 70-30 rule, the 80-20 rule, and the 90-10 rule) with five different sets of sample data generated for each frequency distribution rule. (The variation in sample data sets was produced by changing the initial seed when generating random numbers.) In other words, each different combination of PERCEN and NU was tested for fifteen different data sets.

Tests were run on sample record set sizes of 100, 150, and 200 for both HEUR and Algorithm K. Due to the space required to test Algorithm K, it was prohibitive to run comparison tests for sample sizes greater than 200. The results from the largest sample size (n=200) are presented in this paper. The results from organizing

smaller trees supported the results of the larger record sets. Yet, as the size increased, the consistency of the data produced increased. Tests were run on HEUR for a sample record set of size 1000 in order to obtain information to assist in predicting the rate of increase of the number of comparisons as a function of the number of nodes in the tree.

B. ANALYSIS OF RESULTS

The analysis of experimental data will be presented in two parts. First, considering the quality of the search trees produced, the resulting AST values under each distribution rule will be discussed. Following that will be a look at the number of comparisons required to organize using HEUR.

When discussing the average search time achieved during experimentation the text will refer to the search penalty. This penalty is found by transforming the raw data (AST) into a ratio in respect to the values generated by Algorithm K. The ratio is then reduced by one and multiplied by one thousand to express the search penalty in parts per thousand.

search penalty = (HEUR AST/Optimal AST - 1) * 1000

A search penalty of 12 would thus indicate that the average tree search takes 1.2% more comparisons than for the optimal organization.

The data produced when organizing a database with a frequency distribution according to the 70-30 rule appears to put some limits on the range of PERCEN and NU. In every case, the average search penalty for each PERCEN is lowest when NU is equal to .667. However, NU=.5 and NU =.75 differ only slightly in average search penalty. The lowest average search penalty occurs when PERCEN=.15, with a difference of less than 2 (parts per thousand) in the search penalty with PERCEN=.20. In both cases (PERCEN=.15 and PERCEN=.20), the average AST when using HEUR is no more than .5% higher than optimal AST. There is an overall improvement in AST when PERCEN is greater than .10, and a degradation in AST when PERCEN is increased to .25 or greater.

The range of best values for PERCEN and NU when a database whose distribution of relative frequency of access fits the 70-30 rule are as follows.

$$.15 \leq \text{PERCEN} \leq .20$$
$$.50 \leq \text{NU} \leq .75$$

It appears that searching around the median is helpful, but a p(i) must be quite large in order to justify choosing the corresponding k(i) as the root. This is expected since the relative frequencies of the 70-30 rule are still not extremely deviant from a uniform distribution. (See Table IV.)

The results from testing the frequencies produced

TABLE IV

COMPARISON OF HEUR TO ALGORITHM K

AVERAGE PENALTY IN PARTS PER THOUSAND

70-30 Rule

Percent Search Width	Values of NU: 0	.25	.33	.50	.67	.75	1.0
10%	8.8	8.8	8.8	8.8	8.4	9.4	11.4
15%	6.6	6.0	6.0	4.4	3.6	4.0	6.0
20%	9.4	8.6	7.6	5.4	5.0	5.8	6.8
25%	10.4	10.0	9.0	7.0	6.8	7.4	8.8
30%	15.0	11.8	11.0	8.4	8.0	8.4	9.6
35%	23.4	17.6	14.0	10.0	9.4	10.2	11.2

Difference = (HEUR AST/Optimal AST - 1) * 1000

according to the 80-20 rule yield more specific limitations on both PERCEN and NU. Once again, there is definite improvement achieved by searching around the median with consistently better results when PERCEN=.20. There is a difference in the average search penalty of less than 2 (parts per thousand) when PERCEN is changed to 15% or 25%. There is a marked degradation of the search penalty when PERCEN is increased to 30%. The smallest search penalties result from NU=.667 or .75 in most cases. If the smallest search penalty does result from a different value of NU, the improvement over NU=.667 or .75 is no more than .6 (parts per thousand).

The range of best values for PERCEN and NU when a database whose distribution of relative frequency of access fit the 80-20 rule are as follows.

$$.15 \leq \text{PERCEN} \leq .25$$
$$.67 \leq \text{NU} \leq .75$$

Due to the wider span of values in the 80-20 rule (as opposed to the 70-30 rule), HEUR seems to be justified in looking farther away from the median in order to find a relatively large value of p(i). Still, off-median values of p(i) must "prove their worthiness" by being able to withstand large values for NU before the corresponding k(i) will be chosen as a root. (See Table V.)

The last data set (90-10 rule) proved to be the most interesting. The outstanding characteristic of these results is the extreme degradation of the average search

TABLE V

COMPARISON OF HEUR TO ALGORITHM K

AVERAGE PENALTY IN PARTS PER THOUSAND

80-20 Rule

		Values of NU						
		0	.25	.33	.50	.67	.75	1.0
Percent Search Width	10%	9.8	9.8	9.8	9.6	10.0	10.8	14.0
	15%	6.6	6.2	6.4	7.2	6.8	6.8	10.0
	20%	7.6	6.8	5.4	5.4	4.4	4.6	6.6
	25%	8.6	8.6	7.2	7.0	5.8	6.2	10.6
	30%	15.6	16.0	16.0	13.0	9.6	9.0	11.6
	35%	21.0	21.6	19.4	16.6	12.2	13.2	15.6

Difference = (HEUR AST/Optimal AST - 1) * 1000

penalty for NU=1 when compared to the average search penalty when NU assumed any other value. This was apparent for all values of PERCEN. Another feature of the 90-10 rule test results is that both the 10% and 35% search widths produced much poorer results than any other values of PERCEN. The best results are found when PERCEN=.15 or .20. NU may vary from .25 to .75 and still produce comparable results.

It seems that when the probability density function of p(i) fits the 90-10 rule, HEUR does not have to look as far for a k(i) with a suitably high p(i) in searching for an appropriate root. It is helpful to have NU>0 in order to weed out those p(i)'s which are insignificantly larger than the median. However, due to the wide variance of p(i)'s, a value of NU=.25 is not any worse (or better) on the average than a value of NU=.75. The appropriate p(i) can survive larger values of NU. (See Table VI.)

The average number of comparisons required to run HEUR on a sorted database was relatively consistent among all fifteen test cases for each test value of n. In each case, the number of comparisons required was approximated by ($alpha*nlog_2n$). The value of alpha decreased as the value of n increased. When n=100, alpha was equal to 1.44. When n=200, alpha was equal to 1.29, and a value of 1.11 for alpha resulted when n=1000. (See Table VII.) Obviously the decrease in alpha will level out at some

TABLE VI

COMPARISON OF HEUR TO ALGORITHM K

AVERAGE PENALTY IN PARTS PER THOUSAND

90-10 Rule

Percent Search Width	Values of NU: 0	.25	.33	.50	.67	.75	1.0
10%	12.6	12.2	12.4	12.2	12.2	12.4	51.0
15%	7.8	2.6	2.4	2.6	2.4	3.0	39.8
20%	4.8	5.4	5.2	5.2	6.6	6.8	34.2
25%	6.6	8.4	8.4	6.0	7.8	8.2	34.0
30%	9.4	8.8	8.6	6.4	8.2	9.2	35.6
35%	13.2	19.2	19.0	20.2	17.4	18.6	46.2

Difference = (HEUR AST/Optimal AST - 1) * 1000

TABLE VII

NUMBER OF COMPARISONS REQUIRED

FOR ORGANIZATION WITH HEUR

N	AVERAGE NUMBER OF COMPARISONS	ALPHA	RESULT
100	958.3	1.44	956.72
200	1979.1	1.29	1972.11
1000	11075.1	1.11	11062.02

$RESULT = ALPHA * N \log_2 N$

point. This does not detract from the low number of comparisons, especially when compared to Algorithm K which requires $O(n^2)$ comparisons. (See Table VIII.)

Variation of KVAL, the cut-over point to Algorithm K, has not been mentioned up to this point. Preliminary testing of HEUR showed that reducing KVAL from ten to two degraded the AST by no less than 300% and up to 600%. This reduction of KVAL was equivalent to eliminating Algorithm K from HEUR. Using Algorithm K apparently improves AST considerably without increasing run-time significantly. Some testing was done with KVAL=7. The results of this testing were promising. Although it would be difficult to improve on the results presented previously in this paper, it appears that KVAL=7 may produce comparable results (but not quite as "nearly optimal") with a slight decrease in number of comparisons. Increasing KVAL to a value much greater than ten would obviously begin to degrade run-time, due to the characteristics of Algorithm K. Further experimentation in this area might prove interesting.

TABLE VIII

AVERAGE COMPARISONS REQUIRED FOR HEUR AND ALGORITHM K

RULE	HEUR N=100	N=200	ALGORITHM K N=100	N=200
70-30	867.1	1790.4	9914.0	39400.6
80-20	964.8	1989.0	10033.6	38632.6
90-10	1043.0	2157.9	10365.6	38802.2
AVG	958.3	1979.1	10104.4	38945.1

VI. CONCLUSIONS

A. RECOMMENDATIONS

Testing for a root around the median of the relative frequencies of a table of ordered elements whose frequencies are not uniform can always result in a better average search time than just picking the median as the root. However, one must be careful in choosing how far away from the median to "look." Merging the results from experimentation on tables whose p(i)'s probability density function fits the 70-30 rule, the 80-20 rule, or the 90-10 rule, it is best to choose a percent of the cumulative frequency of a subtree (or a tree) which is between 15% and 20%. If a user knows a given database well enough to estimate exactly which category it falls under (the 70-30 rule, the 80-20 rule, or the 90-10 rule), more specific information is provided in Section V.

For each of the distributions, an average degradation of average search time of no more than 1% over optimal can be expected. In each of the five test cases for each rule, actual data using PERCEN=.15 or PERCEN=.20 never resulted in an AST which was more than 1.5% worse than optimal. The run time is considerably better than that of Algorithm K. The cost of HEUR increases in proportion to $n \log_2 n$ while the cost of Algorithm K increases proportional to n^2. As was mentioned earlier, as the

frequency distribution approaches uniformity, a height-balanced tree will produce acceptable results.

B. RELATED CONCERNS

There are several related areas which have not been covered or have not been covered in depth in this research. One is how to update a dynamic database as insertions and deletions occur. There are many different possibilities. Applying the ideas of height balancing, weight balancing, or a combination of these two are just a few possible approaches.

Another related concern is the problem of when to reorganize a dynamic database. Schneiderman states that, "Reorganization can be performed at fixed time intervals or when the average search cost has deteriorated to a certain level." [19, p. 362] He then proceeds to discuss different strategies for selecting optimum reorganization points according to the individual database's characteristics. Further investigation of that paper and its implications in the light of results reported here could prove profitable.

A third pertinent area of research which goes hand-in-hand with this discussion on how to organize is the problem of gathering and storing current information on the actual relative frequencies of access of each element in a table. How to accumulate this information and where to store it are the two most obvious problems. The

difficulty of record popularities changing over time is also a problem. Should there not be some sort of time penalty for accesses which were made a "long time ago?" Perhaps the access counts should just be restarted after a given period of time or after each reorganization. If so, should all elements be restarted at the same initial value, or is there some method wherein certain elements could be assigned higher initial values? There are obviously many approaches to the problem of gathering and storing access information. Getting good estimates of p(i)'s is vital to the productive use of methods like HEUR. The cost of obtaining this information must be realistically assessed.

Another necessary inquiry concerns the effect on AST from errors in the p(i)'s used to organize the search tree. The results from this inquiry are important in determining what cost of getting good estimates of p(i)'s is justifiable. If a large margin of error is acceptable, an expert's approximation may be accurate enough for a particular database.

These are just a few areas in which further investigation is suggested when studying the problem of organization. As in any sector of Computer Science, the possibilities for more extensive research seem endless.

Yet this effort has made a start in this exploration. It has shown that there are efficient methods by which nearly-optimum search trees may be organized.

BIBLIOGRAPHY

1. Knuth, D.E. The Art of Computer Programming, Volume 3: Sorting and Searching. Reading, MA: Addison-Wesley, 1973.

2. Hu, T.C. and A.C. Tucker. "Optimal Computer Search Trees and Variable-Length Alphabetical Codes," SIAM Journal of Applied Mathematics, Vol. 21, No. 4, December 1971, pp.514-532.

3. Reingold, Edward M. and Wilfred J. Hansen. Data Structures, ed. Gerald M. Weinberg. Boston: Little, Brown, and Company, 1983.

4. Walker, W.A. and C.C. Gotlieb. "A Top-Down Algorithm for Constructing Nearly Optimal Lexicographic Trees," Graph Theory and Computing, ed. Ronald C. Read. New York: Academic Press, 1972.

5. Reingold, Edward M. and Wilfred J. Hansen. Data Structures, ed. Gerald M. Weinberg. Boston: Little, Brown, and Company, 1983.

6. Iverson, K.E. A Programming Language. New York: John Wiley and Sons, Inc., 1962.

7. Knuth, D.E. "Optimum Binary Search Trees," Acta Informatica, 1:14-25, 1971.

8. Allen, Brian. "On the Costs of Optimal and Near-Optimal Binary Search Trees," Acta Informatica, 18:258-263, 1982.

9. Reingold, Edward M. and Wilfred J. Hansen. Data Structures, ed. Gerald M. Weinberg. Boston: Little, Brown, and Company, 1983.

10. Lewis, T.G. and M.Z. Smith. Applying Data Structures. Boston: Houghton Mifflin Company, 1976.

11. Walker, W.A. and C.C. Gotlieb. "A Top-Down Algorithm for Constructing Nearly Optimal Lexicographic Trees," Graph Theory and Computing, ed. Ronald C. Read. New York: Academic Press, 1972, pp.303-323.

12. Knuth, D.E. The Art of Computer Programming, Volume 3: Sorting and Searching. Reading, MA: Addison-Wesley, 1973.

13. Bobillier, P.A. and others. Simulation with GPSS and GPSS V. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1976.

14. Heising, W.P. "Note on Random Addressing Techniques," IBM Systems Journal, Vol. 2, June 1963, pp. 112-116.

15. Metzner, John R. Personal interview. June 20, 1984.

16. Knuth, D.E. The Art of Computer Programming, Volume 3: Sorting and Searching. Reading, MA: Addison-Wesley, 1973.

17. Knuth, D.E. "Optimum Binary Search Trees," Acta Informatica, 1:14-25, 1971.

18. Walker, W.A. and C.C. Gotlieb. "A Top-Down Algorithm for Constructing Nearly Optimal Lexicographic Trees," Graph Theory and Computing, ed. Ronald C. Read. New York: Academic Press, 1972.

19. Shneiderman, Ben. "Optimum Data Base Reorganization Points," Communications of the ACM, Vol.16, No.6, June, 1973, pp. 362-365.

VITA

Joy Lanelle Henderson was born on July 14, 1959 in Union City, Tennessee. She received her primary and secondary education in Martin, Tennessee. She received a Bachelor of Science degree in Computer Science from the University of Tennessee-Martin, in Martin, Tennessee in December 1981.

She has been enrolled in the Graduate School of the University of Missouri-Rolla since January 1982. She has held a graduate assistantship for the duration of her enrollment at UMR, and held a Chancellor's Fellowship from August 1982 through December 1984.