1974

A study of the efficiency of the bounded balanced binary tree technique for physical storage management of computerized data files /

R. Ian Bardsley
Lehigh University

A STUDY OF THE EFFICIENCY OF THE BOUNDED BALANCED BINARY TREE TECHNIQUE FOR PHYSICAL STORAGE MANAGEMENT OF COMPUTERIZED DATA FILES

by

R. Ian Bardsley

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial Engineering

Lehigh University

1974

CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

April 24, 1974
Date

M Wayne Shumaker
Professor in Charge

Chairman of the Department of
Industrial Engineering

ACKNOWLEDGEMENTS

The author expresses his appreciation to Dr. M. W. Shiveley for his advice and guidance during the preparation of this thesis.

Mr. G. E. Whitney of the Western Electric Research Center staff deserves special thanks for his counsel and support as well as for the suggestion of the topic for this thesis.

The author also wishes to thank Mr. C. R. Dudgeon for his assistance and guidance in the statistical analyses that were required during the preparation of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Many different techniques have been proposed or developed for the physical storage management of computerized data files. Of these numerous techniques no single one has emerged or even been suggested as an overall best as the performance of each technique is dependent upon the environment within which they are used. A great number of these techniques are based upon a tree type of structure so that they can process data both sequentially and randomly. It is generally accepted that an effective tree is one that is close to being balanced, so that search times are short, able to be restructured easily when it becomes necessary, and these restructurings should be required infrequently.

A technique for constructing a Binary Balanced Binary Tree was recently proposed by J. Nievergelt of the University of Illinois. Mr. Nievergelt suggests that his technique provides a very good, if not the best, compromise between the conflicting requirements of an effective tree type file management technique. Mr. Nievergelt gives equations which will predict the expected performance of his technique. These equations are based upon an analytical analysis of the technique.

This thesis examines the performance of this technique through simulation. To do this a computerized model of the technique was developed and used to build bounded balanced binary trees under varying conditions. By recording the characteristics of the trees

as they were built it was possible, through subsequent analyses, to develop predictors which can be used to generate the expected characteristics of a tree for any given environment.

A comparison of these two sets of performance evaluations shows that in the area of expected search times the predictors are very close, while those in the area of the expected number of rebalancings required by the technique differ greatly. These predictors can be used in the development of cost models which will enable the effective comparison of the performance of this technique against that of other techniques for any given environment.

CHAPTER I

INTRODUCTION AND BACKGROUND

Methods for the storing of data so that it can be stored and retrieved quickly and efficiently have been developed and utilized since the time man first recorded information in a recoverable format. The development of written languages gave man the ability to record large amounts of data and resulted in the creation of libraries in which to store that data. The physical storing and retrieval of data in such libraries was always a completely manual effort with the amount of time and effort required to locate a particular piece of data being dependent upon the method of organization of the library.

The invention of the electronic digital computer gave man the ability to process large amounts of data in a relatively short period of time. In order to be able to take full advantage of this processing capability, techniques and devices had to be developed that not only had the capability to store data in a machine readable format, but were able to organize the data so that it could be accessed quickly and efficiently when needed. The first two generations of computers were restricted mainly to sequential files such as cards, or paper or magnetic tape for high volume data storage. Processing of the data was done in batch type systems where the data was processed in the same sequence as it was stored. The introduction of the "third generation" computers not only brought

large increases in processing speed, but also the advances in hardware which made large random access files not only possible, but also feasible to more users. With each advance in computer hardware, new software, especially file organization techniques, had to be continually developed and revised so as to be able to take advantage of the updates in hardware.

Many different techniques have been developed and implemented for the physical storage management of computerized data files. The main reason for the variety and number of such techniques that are available is that the efficiency of any sigle technique is dependant upon not only the characteristics of the data to be stored, but also upon the requirements and priorities of the user. This creates the situation where a technique that is highly efficient in one environment may be highly inefficient in others.

Probably the quickest, easiest and most efficient way to store data is in a simple serial list. The first data item is placed at the top of the list and each succesive item is placed at the end of the current list. By keeping an indicator as to the location of the last record in the list, new records can be added very quickly and the end of list pointer updated with each addition. Using this technique records in the file can also be located using a very simple method. That method is to start at the top of the list and search serially through it until either the desired record is found or the end of the list is reached, which indicates that the desired record is not in the file. While this method of locating

method of locating a record is easily performed, it may be a very lengthy and inefficient procedure. The expected number of probes (a probe is defined as the process of locating and examining any single record in the file or list) required to locate a record that is known to be in the file is (n+1)/2, where n is the number of records in the file [2]. In the worst possible situation, when the desired record is as the end of the list, n probes are required. When the desired records are not in the file, n probes are required to determine that it is not in the file. This limits the usefulness of this technique to small, short lived files where the effort needed to implement a more sophisticated technique is greater than that wasted by the inefficiency of this technique.

For larger files when short retrieval times are desired a method which utilizes scatter storage techniques such as hashing can be implemented. In these techniques an area of storage large enough to contain the entire data file is allocated at the start of processing. This area is then divided into segments so that each segment can contain one record and there are as least as many segments as there are records to be stored. To place a record into storage in this type of file the key with which the record is identified must be processed through a randomizing algorithm and converted into a random number, or secondary key, between 1 and y (the number of segments in the storage area). The randomizing algorithm must be such that every time a given key is fed

into it an identical secondary key will be generated. It is allowable and expected that several primary keys (synonyms) may generate identical secondary keys (collisions). The record is then stored in the segment associated with the secondary key. To circumvent the problem of collisions, when they occur, additional routines must be executed to resolve the duplication. These routines may be as simple as searching the storage area for the next unused segment, or as sophisticated as chaining or the use of secondary psuedo-randomizing algorithms. To locate a record in the file using this type of technique a similar process is repeated. The key of the desired record is processed through the randomizing algorithm and converted into a segment address. The record at that segment address is then examined. If that segment is empty the desired record was not in the file. If a record is contained in that segment it may or may not be the desired record. It may not be the desired record if a collision occurred when entering the record and the technique needed to subsequently locate the desired record is dependent upon the collision handling routine that was used.

The efficiency of scatter storage techniques is dependent upon three factors. The first is the load factor of the storage area (The actual number of records in storage/total capacity of storage). When the load factor is low, few collisions are likely to occur and the expected number of probes required to locate a given record is close to one. As the load factor increases, additional

probes will be required to resolve the collisions which now occur more frequently. A second factor is the collision handling technique. Techniques such as linear search, which are simple to execute can often result in a large number of probes being necessary to resolve a collision, while techniques such as chaining or secondary pseudo-randomizers reduce the number of probes but are often complex and lengthy to execute. Table 1-1 [13] shows the expected number of probes required for various types of collision handling techniques. In the case of chaining

Load Factor	Collision Technique		
	Linear	Random	Chaining
.1	1.06	1.05	1.05
.5	1.50	1.39	1.25
.75	2.50	1.83	1.38
.9	5.50	2.56	1.45
1.5	-	-	1.75
2.0	-	-	2.00

TABLE 1.1

Expected Number of Probes to Locate a Given Record

load factors in excess of one were obtained by allocating additional storage in which the collision records were placed. The third factor is the randomizing algorithm itself. Very few

algorithms produce truly random secondary keys. Most algorithms create a condition called bunching where the secondary keys are not uniformly distributed over the entire storage area but concentrated in one or several sections. It should be noted that the data for Table 1-1 was obtained analytically assuming truly random randomizing algorithms. If bunching did occur, the expected number of probes would increase according to the amount of bunching that occurred.

One of the main drawbacks to scatter storage techniques is that to be effective they require large amounts of unused storage space in order to keep the load factor small. For dynamically growing files this means that either very large amounts of storage must be allocated initially, or that when the file outgrows its present storage space a larger area be allocated, a new randomizing algorithm developed, and the entire file reprocessed into the new storage area. Another factor to be considered is that sequential processing of the file is extremely difficult using scatter storage techniques unless an auxillary sequential file of keys is also maintained.

When storage space must be kept minimal and/or sequential processing is desired, a sequential table can be used. When straight sequential tables are used, storage utilization can be kept low by allocating storage space only when it is needed. Because the sequence of the file is known the utilization of binary type techniques can be used to keep access times relatively low.

The expected number of probes needed to locate a record is (log n) -1 with a maximum of log n in the worst cases [15]. If the file is kept physically in sequence the addition of records to the file can become quite lengthy, especially as n becomes large. This can be avoided by storing the file as a binary tree in which the physical sequence is destroyed but through the use of pointers the logical sequence can be retained. Now additions can be made in any available location and the sequence maintained by updating a few pointers.

Many techniques have been developed utilizing the features of a binary tree. Probably the best known offshoot is the Indexed Sequential Access Method which is available on most large computer systems. Some of these offshoots guarantee low access times to locate records but do so at the expense of requiring complex and lengthy procedures to add or delete records from the file. Many others do the opposite and guarantee ease of file update capabilities but at the expense of access times.

The efficiency of binary trees is related to the shape of the tree. Uniform, well balanced trees produce low record access times but are difficult to maintain uniform, and techniques which have quick update capabilities do not produce uniform trees [15]. Dozens of techniques are available that will for any given set of data build a uniform or nearly uniform tree, however, they require extremely lengthy update procedures for dynamic files.

To avoid these lengthy restructurings a method was developed by Adel'son-Vel'skii and Landis which was able to control the shape of the tree by regulating the height of all subtrees [15]. The height of a tree or subtree is defined as the maximum number of probes needed to locate any record in the tree. By maintaining indicators at each node, Adel'son-Vel'skii and Landis were able to devise a method of local restructurings. This method would maintain the shape of the tree by restructuring it whenever the heights of the left and right subtrees of any node differed by more than one, thereby restoring the difference in heights to at most one.

This technique was later modified by Caxton Foster who allowed the heights to differ by up to five [5]. By doing this his technique enabled the user, through his choice of maximum allowable height differences, to vary the shape of the tree so as to best fit his own requirements.

CHAPTER II

BOUNDED BALANCED BINARY TREES

Another method of regulating the shape of a binary tree was recently introduced by J. Nievergelt and E. M. Reingold of the University of Illinois [14]. The tree produced by this technique is called a Bounded Balanced Binary tree (BB tree) and it is suggested by Mr. Nievergelt that this technique will not only provide for low storage requirements but also allow for easily coded and efficiently executing algorithms which will permit minor modifications of the tree to insure that access and update times will be kept minimal. The frequency of these modifications to the tree can be controlled by the user and does not have to remain constant for the life of the tree. This feature enables the user to encourage these modifications during low activity periods and to discourage or limit them during high utilization periods or periods when fast responses to file updates are required. Mr. Nievergelt recognizes that his algorithm will not guarantee optimately with respect to any of the users requirements but suggests that it will provide the best compromise between the conflicting requirements such as high storage utilization, ease in modification and low access and update times.

In defining a Bounded Balanced Binary tree it is assumed that the reader understands the structure of a standard binary tree in addition to the following definitions:

To: A binary tree which contains no nodes. It is sometimes referred to as a null or empty tree.

Tn: A binary tree containing n nodes. It is considered an ordered triple (Tl, v, Tr), where Tl and Tr are the left and right subtrees respectively and v is a single node called the root of Tn. $(l \geq 0, r \geq 0, l+r+1=n)$.

ρ(Tn): The balance or root-balance of a binary tree (or subtree) Tn where $n \neq 0$ is given by

$$\rho(Tn) = \frac{l+1}{n+1}$$

Bounded Balance: A binary tree Tn is said to be in Bounded Balance alpha (α) if and only if the following conditions hold:

$$0 \leq \alpha \leq 1/2$$
$$\alpha \leq \rho(Tn) \leq 1-\alpha$$
Both Tl and Tr are of bounded balance

It is the choice of the balance limit α which allows the user to control the shape of the tree. If α is chosen as zero, since by definition ρ can never be equal to 0 or 1, no restrictions are placed upon the growth of the tree, whereas an α of .5 will place the maximum number of restriction on the tree and will result in the creation of a perfectly uniform tree. It has been analytically proved [14] that choices of alpha in the range between 1/3 and 1/2 will produce identical trees which gives alpha an effective range of 0 to 1/3. In order to ensure that the balance of a tree is maintained within the limits of α and $1-\alpha$ it is necessary to add a field to each record in the tree. This field will

contain the number of nodes or records contained in the subtree of which the desired record is the root node.

Since the shape of the tree cannot be changed during examination of the tree, it is only necessary to check the balance during updates. To add or delete a record, first the root node must be examined to determine the subtree (T_l, T_r) to which the change must be made. As each record contains a field which contains the number of records in the tree below that point, it is a simple operation to calculate what the new node balance will be after the update has been made. If the balance limits are not violated the size field in the root node can be updated to reflect the change and the process continued with the root node of the appropriate subtree. If the balance limits are violated the tree can be brought back into balance by performing one of two restructuring algorithms provided that a slight restriction is placed upon the choice of alpha. That restriction is that alpha must be less than $1-(\sqrt{2}/2)$. It is called a slight restriction because $1-(\sqrt{2}/2)$ is approximately .2928 and since an alpha of between .333 and .5 will produce identical trees, this leaves a range of approximately .04 uncovered, and if alpha was chosen within this range the resulting tree would be so close to being completely uniform that it would be impractical to try to maintain.

Assuming that during the update of a tree an unbalance was calculated to occur at node A and we have the following structure:

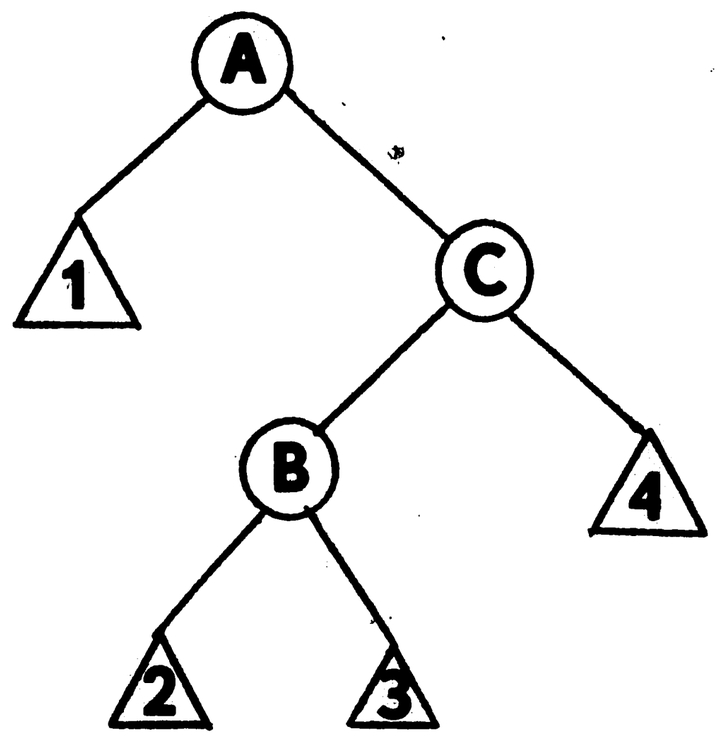

Figure 2.1 Sample tree before rebalancing

Where A, B and C are nodes and 1, 2, 3 and 4 are subtrees (may be null). Assuming that the update will be performed in the right subtree of node A it is only necessary to calculate what the balance of node C will be after the update has been performed. If this balance β is such that $\alpha \leq \beta \leq (1-2\alpha)/(1-\alpha)$ then the tree can be returned to balance and by performing an algorithm called rotation which transforms the tree into the following structure:

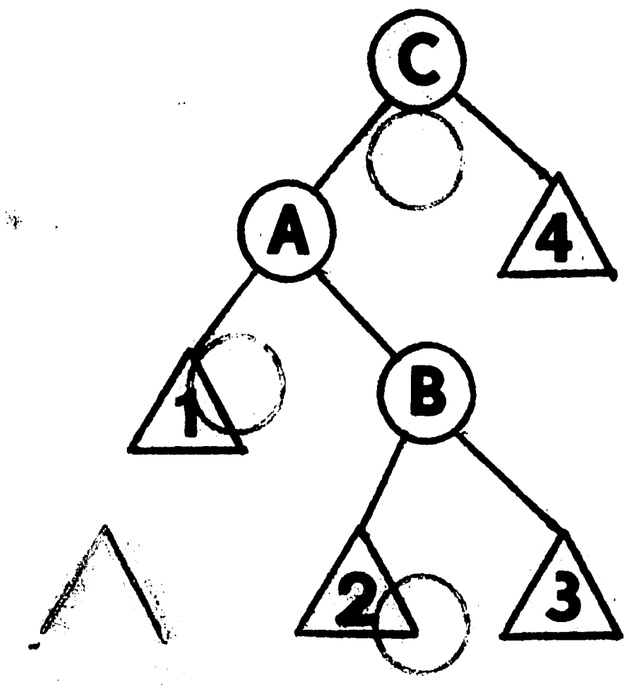

Figure 2.2 Sample tree after rebalancing

In situations where the balance limits have been recently changed the new balance at node C is not guaranteed to be between α and $1-\alpha$ but it will be closer than the old balance at node A. In this situation repeated applications of the restructuring algorithms will bring the balance within the desired limits.

If the balance β does not fall within the previously mentioned limits a second restructuring algorithm called split rotation must be used. This algorithm will result in the following structure:

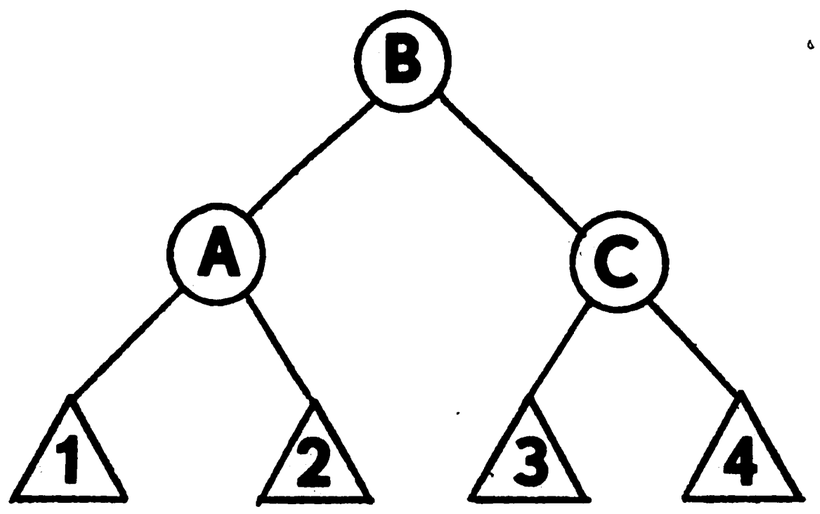

Figure 2.3 Sample tree after split-rotation

To perform these two algorithms it is not necessary to physically move any records. The restructurings can be achieved by changing the appropriate pointers and updating the appropriate size fields. In the examples of these two algorthims it was assumed that the update would be performed in the right subtree of the node where the imbalance occurred. If it were to occur in the left subtree the mirror image of these two algorithms can be used.

An analytic proof for these two algorithms can be found in an

article entitled "Binary Search Trees of Bounded Balance" by J. Nievergelt and E. M. Reingold published in the Procedures of the 4th annual A. C. M. Symposium on Theory of Computing (1972)[14].

If during an update of the tree one or more rebalancings were performed and it was later determined that either the record to be added already existed, or the record to be deleted did not exist, it is only necessary to correct the size fields that were changed during the update and no re-rebalancings will be necessary as the original rebalancings will only have brought the balance closer to 1/2 even though no actual update was performed.

Using a straight binary tree technique with no rebalancing the expected average search time is approximately 1.39 $\log_2 n$ but can go as high as n searches for the worst case. As no rebalancings are performed, addition and deletion are very easy to perform In the tightest balance 1/2, the expected average search time is $\log_2 n - 1$ and the worst possible search time is $\log_2 n$, but restructuring algorithms are extremely difficult and in the worst case may require restructuring every node. A tree of bounded balance $(1-\sqrt{2}/2)$ has an expected search time of less than 1.115 $\log_2 n$ (experimental data suggests it is actually closer to 1.05 $\log_2 n$) and the worst possible search time is only 2 $\log_2 n$ with the maximum number of restructurings to be performed in the worst possible case being 2 $\log_2 n$. The average number of rebalancings needed to add a single record has been analytically proven to be independent

of the tree size and less than $2/1=2\alpha$. Mr. Nievergelt suggests that experimentation indicates the actual number is far less than this.

The discrepancy between the analytic and experimental results is probably due to the unknown distribution of the balances of the nodes within a tree. The analytic results were achieved assuming a uniform distribution while experimentation shows it to be more close to a truncated normal distribution.

The main objective of this thesis is to, through simulation, attempt to build a model through the use of which the efficiency of BB trees can be accurately predicted.

CHAPTER III

DESCRIPTION OF THE MODEL

The model used to test the Bounded Balanced Binary tree technique was written in PL/1 for the I.B.M. 360, utilizing core for storage of the data. PL/1 was chosen because of its pointer variable feature. Through the use of pointer variables it was possible to maintain the entire storage file in core using a technique which is almost identical to that which would have been necessary if auxillary random access storage devices had been used. This was done by storing the actual core addresses of the records in the pointers. If auxillary storage had been used the physical addresses (such as CCTTRRR; C = cylinder #, T = track #, R = record #) would have been used. The choice of core for storage of data was made in order to reduce the run times of the many runs which were required to evaluate this technique.

The model was written as a self-contained package with all interface performed by a single parameter, or set of parameters. The model performs its own storage allocation and deallocation algorithms based upon the parameters passed to it by the user. All calls to the model can be performed by use of a single statement,

```
CALL BALTREE (PARAMS);
```

where PARAMS has the following structure:

```
1   PARAMS,
      3   COMMAND_CODE      CHARACTER (1),
      3   ERROR_CODE        FIXED DECIMAL (5,0),
      3   KEY_POINTER       POINTER,
      3   DATA_POINTER      POINTER;
```

The command code indicates to the model which of the 14 operations that is to be performed. The error code field is used as a feedback device so that the model can indicate to the user the status of his last request. This field should be checked after each call with zero indicating successful completion. The key and data pointers are used to point to the key and data portions of the current record. Both the key and data portions of the record must be in the PL/1 version of variable length fields. That is the actual data portion of the field must be preceded by a two byte binary field which contains the length of the data portion of the field. It does not include the two bytes needed for the length indicator (e.g. 'Ø3KEY' where Ø3 would be in binary).

The first command to the model must be an initialize command. To execute this command the command code must be set to 'I' and the key pointer field set to point to a set of four half word binary fields which are set up as follows:

```
1.    FIELDS,
        2   FLD1      FIXED BINARY (15,0),
        2   FLD2      FIXED BINARY (15,0),
        2   FLD3      FIXED BINARY (15,0);
```

FLD1 must contain the largest number of records that the file will be used to store. FLD2 must contain the maximum length of the key field and FLD3 must contain the maximum length of the data field. FLD4 is not used for this command. Based upon the information supplied to it the model will calculate the amount of storage required and obtain that amount of storage. In this version of the model each

record is allocated storage space based upon its maximum possible length. In addition to the record key and data fields, three additional fields are added before any record is placed in storage. One of these fields is a size indicator which will contain the number of records in the subtree of which this record is the root node, and the remaining two fields will contain pointer variables which contain the addresses of the left and right subtrees, or null indicators if the respective subtrees are empty.

Once storage allocation has been made to accommodate the number of records specified in FLD1, these storage slots, which are now empty, are chained together to allow efficient allocation when they are needed. Starting with the first slot in the file, the left pointer in each slot is initialized with the address of the next sequential slot, with the left pointer in the last slot being set as null. The right pointer of the first slot is then set to null. The first slot is never used for storage of any input records but is reserved for record keeping purposes. The left pointer of slot 1 always points to the next available slot into which an inputted record can be placed. The right pointer always points to the root node of the storage tree. It should also be noted that the size field of each storage slot is preset to zero to prepare for the use of a debugging aid which will be explained later. The initialization routine also presets the balance limits (alpha) of the tree to a default limit of .25. Using this alpha it calculates BETA equal to $1 - \alpha$ and BLIM equal to $\frac{1 - 2\alpha}{\alpha}$.

Upon successful completion the initialization routine will set the error code field to zero and return control to the user. If in calculating the amount of storage space required the model finds an error, that either less than 1 byte or more than 1,000,000 bytes are requested, it will return an error code of 7.

If it is desired to set the balance limits of the tree at other than the default values the command SET-ALPHA (command code 'A') is used. When this command is issued, similar to the initialize command, the Key pointer must be set to point to a two byte binary field. This field must contain the desired alpha value multiplied by 10,000. When the model accesses this field it will divide the contents by 10,000 and calculate the corresponding values for BETA and BLIM. The model will also data check the setting of alpha and if found to be in error it will reset it to the default value and return an error code of 6.

When it is desired to examine the storage file to retrieve data from it the command SEARCH (command code 'S') is used. The only requirements to use this command are that the key pointer field must point to a field which contains the key of the desired record and the data pointer field must point to an area into which the data portion of the record can be placed. In order to locate the desired record the model first examines the left pointer of the first slot in the storage area to determine where the root node of the tree has been stored. The model then compares the key of this node against the key of the requested record.

If these keys are equal the data portion of that record is copied into the field pointed to by the data pointer field and control is

returned to the user. If the keys are unequal to the left or right pointer can be examined, depending upon whether the desired record is lower or higher lexographically than the obtained root node. This procedure can then be repeated using the subtree whose root node is pointed to by the appropriate left or right pointer. This process continues until either the desired record is located or a null pointer is reached which indicates that the desired record is not in the file and control is returned to the user with an error code of 1.

Should it be necessary to change the data field associated with an existing record the command UPDATE (Command code 'U') is used. As in the search command the key pointer must point to a field which contains the key of the desired record and also the data pointer must point to a field which contains the new data which is to be placed into storage. To locate the desired record in storage the model initiates a search command. Once the record is located it is a simple operation to replace the data field of the record with the new data. If the desired record is not in the file the data portion cannot be updated and the model will return an error code of 1. It will <u>not</u> enter the record into the file as a new record.

In order to place a new record into the file the command ADD (Command code 'A') must be used. When this command is executed the key and data pointer parameters must point to fields which contain the key and data portions of the new record. To add the record into the binary tree the model performs a routine similar to the search routine. Starting with the root node of the tree it will determine into which subtree the new record will be added. It will then start checking to

see if the addition of the new record will violate the balance limits of the tree. If first checks the size of the tree to see if it is less than $1/\alpha + 2$. If it is, the addition of the new record cannot force the tree to violate the balance restrictions and the record can be added with no further balance checking. For alpha values between .2 and .5 this eliminates between 3/4 and 7/8 of the nodes in the tree as possible candidates for restructuring and as alpha decreases this fraction becomes larger. If however, the tree is a possible candidate for restructuring, the actual new balance at the candidate node must be calculated. As the model already knows into which subtree the new record will be added the new balance can be calculated easily. If the balance limits are violated, then one of the two restructuring algorithms described in the previous chapter can be executed. If no restructuring is necessary, then the size field of the considered record is updated by one and the process repeated with the root-node of the appropriate subtree. The position within the tree that the new record is to be placed is found when a null pointer is reached. To enter the record into the tree it first must be added into the storage file. The right pointer of slot one in the storage area contains the address of the next available slot in the storage area into which the new record can be placed. The right pointer of this open slot contains the address of a second available area and this address is used to update the field in slot 1 for future additions. The key and data of the new record can now be placed in the storage file. The null pointer which was previously realized can be updated to the address of the new

record and the left and right pointers of the new record set to null. Now control can be returned to the user. If in this process it is found that the record being added already exists in the file the error code is set to 2. Before control can be returned to the user all of the size fields which were updated by one must be returned to their original value. If any restructurings were performed during the addition they need not be undone as even with the removal of the record the tree will be closer to a balance of .5

Deletions are performed by calling the model using a DELETE command (Command Code 'D'), with the key pointer field pointing to a field which contains the key of the record to be deleted. The procedure to locate the record to be deleted is identical with that used to find the position in the tree into which a new record would be added except that when null pointer is reached an error condition exists because this indicates that the record to be deleted is not in the file. If this error condition is realized the error code will be set to 1 and the size fields corrected before control is returned to the user. Once the record to be deleted is located one of three conditions will be found to exist. Condition 1 is when the left and right subtrees of the record to be deleted are null. Condition 2 is when only one subtree is null and the third condition is when neither subtree is null. If the record to be deleted is in condition 1 or 2 it can be removed from the tree by simply replacing the pointer which points to it with whichever of its pointers that is non-null, or if both are null with a null pointer. If condition 3 exists a small

algorithm must be executed. This algorithm first determines which subtree of the record to be deleted is larger and locates within that subtree the record whose key is lexographically closest to the key of the record to be deleted. This secondary record will be in condition 1 or 2 and can be deleted from the file as previously described. The actual deletion of the desired record is then accomplished by replacing the key and data portions of it with those of the secondary record that was just deleted.

Sequential processing of the tree is accomplished through the use of two additional commands. The first command is START SEQUENTIAL (Command Code 'X'). When this command is executed the key pointer parameter must point to a field which contains the key with which sequential processing is to begin and the data pointer to an area in which the data portion of the record can be returned. To execute this command the model performs a routine identical to the SEARCH routine except that it records the path from the root node to the desired record in an address queue. If in this process it does not find a record with the desired key it will automatically locate the next record in sequence by generating a NEXT SEQUENTIAL command and return control to the user.

After sequential processing has been started by a 'X' command it can be continued with the NEXT-SEQUENTIAL Command (Command Code 'N'). For successful execution of this command the previous command must have been either a 'X' or 'N' command which executed successfully. To locate the next sequential record the model will examine the record whose address is at the end of the current address queue. If that record has

a right pointer which is non-null, that pointer will be added to the end of the queue. The record at that address will then be examined for a non-null left pointer. If the pointer is non-null it will be added to the end of the address queue and the process of examining the left pointer will continue until a null pointer is found. At that point the key and data portions of that record will be returned to the user after its address has been added to the queue. If in the initial examination no right pointer was found the last address on the address queue will be deleted from the queue. The record which is now at the end of the queue is examined and if its right pointer is equal to the address which was just deleted from the queue, its address is deleted from the queue and the process continued until an unequal address is found. The key and data of the last record in the queue are then returned to the user. When in this process of retracing up the tree the root node of the entire tree is deleted from the queue, an error code of 4 is returned to the user to indicate that the entire file has been processed.

For debugging purposes the model has a Command of DUMP (Command Code 'G'). Upon receipt of this command the model will provide a printout of all records presently active in the storage area along with the contents of their left and right address pointers. This command was added to enable the user to confirm the contents of the storage file in order to help debug his user subroutines or check the operation of the model.

When processing of the file has been completed the command DELETE

ALL (Command Code 'Z') should be used. This command will not only delete all of the records in the file but will also return the storage area to the operating system. If this instruction is used the only instruction which can follow it is another INITIALIZE routine. It should be noted that this deletion routine will also delete the present value of alpha requiring that alpha must be reinitialized when processing is continued or it will be preset to the default value.

There are also four additional commands which deal with SCOPE. SCOPE is a feature which allows the model to store Keys that have one or more prefixes. A description of these commands and a definition of SCOPE has been deferred to Apendix I as this feature does not affect the model as it will be used to evaluate the efficiency of the Bounded Balanced Binary tree technique.

CHAPTER IV

EVALUATION OF THE TECHNIQUE

Before any attempt could be made to evaluate the Bounded Balanced technique through simulation, criteria had to be established by which the tree produced by the model could be measured. The two criteria chosen were the average path length of the final tree, and the number of times the rebalancing algorithms had to be executed in order to build tree. The average path length was chosen because it is a measure of the expected search time required to either locate a record in the tree or to locate the position within the tree where a new record will be added. The number of rebalancings is an indicator as to how difficult it was to produce the tree within its bound limits and how difficult it will be to maintain it during future updates.

In order to be able to evaluate these two measurement criteria, or dependent variables, the model as previously described had to be modified slightly. In order to inform the control program of the number of rebalancings that were performed during each addition, the rebalance algorithms were modified to subtract one from the error code each time the algorithm was executed. In this way the control program could sum these negative error codes to obtain the total number of rebalancings performed to build the tree. To obtain the average path length the sequential processing routine was modified. In order to obtain the average path length the supervisor or control program would request the model to sequentially process every record in the tree. By doing this the model was easily modified to calculate

the average path length of the entire tree and return it to the control program.

Three independent variables or variables which would be inputted to the model to force it to vary the type of tree it produced were chosen. The first was the choice of the balance limit (α) and the second was the number of records in the tree (n). As the shape of binary trees is not affected by the type of distribution of the inputted data (i.e., normal, skewed, uniform) but only by the amount of sequentialness in the data, a measure of the sequentialness (p) was chosen as the third independent variable.

The data inputted to the model was generated by a modified random number generator. The modification was made to eliminate the possibility of generation of duplicate record keys and to force some sequentialness into the data when desired. Although the model would have rejected the duplicate keys automatically, it was considerably easier to program to ensure that no duplicate keys were generated than to initiate corrective action when they were rejected. In order to provide sequentialness into the data the random number generator was programmed to vary the mean of the distribution from which the keys were generated. The generator was programmed to produce numerical key uniformly distributed over a range of $\pm$5000 from the mean, with the mean being initialized at 5000. If no sequentialness was desired in the data, the mean was maintained at 5000 for the entire run. When sequentialness was desired the variable p was introduced with permissable values between 0 and 1. The mean of the

distribution was then incremented by p times the range of the distribution between the generation of each record key. If, for example, p was chosen as .001, the mean would be incremented by 10 each time it the generator was executed.

The values at which the independent variables were set for the initial simulation runs were chosen so as to span the range of reasonable possible values for that variable. The initial values chosen were:

α	n	p
.0	250	.0000
.1	500	.0025
.2	750	.0050
.2929	1000	.001

Choosing one of these values for each variable the model was run 10 times before any of the variables were varied. In order to ensure that no two runs were made using identical input data the seed of the random key generator was varied between each run. After each run was made, the average path length (AVLEN) of the resulting tree and the number of rebalancings required to build it (NREB) were recorded. After each set of 10 runs was complete the mean and variance of AVLEN and NREB for that set of independent variables, or cell, were computed. The results of these 800 runs are shown in Tables 4-1 and 4-2.

Sequentialness (P)	Tree Size (n)	BALANCE LIMITS (α) .0 Mean	.0 Var.	.1 Mean	.1 Var.	.2 Mean	.2 Var.	.2929 Mean	.2929 Var.
.0000	250	9.470	.275	8.312	.067	7.631	.015	7.212	.002
	500	10.864	.241	9.474	.089	8.713	.029	8.218	.001
	750	11.620	.269	10.162	.105	9.322	.021	8.816	.001
	1000	12.201	.254	10.574	.062	9.755	.019	9.226	.001
.0025	250	9.686	.180	8.602	.115	7.758	.044	7.272	.004
	500	14.866	.554	10.585	.240	8.870	.008	8.238	.004
	750	22.007	1.217	11.205	.063	9.378	.022	8.843	.003
	1000	29,616	2.370	11.996	.067	9.821	.005	9.248	.003
.0050	250	12.300	.578	9.026	.103	7.755	.025	7.224	.002
	500	22.011	1.483	10.682	.085	8.782	.022	8.276	.004
	750	32.762	4.158	11.404	.068	9.455	.017	8.816	.002
	1000	43.721	6.587	11.746	.168	9.757	.033	9.241	.003
.0075	250	14.020	1.884	9.302	.144	7.763	.036	7.239	.003
	500	28.727	2.920	10.738	.151	8.924	.024	8.235	.005
	750	41.508	5.973	11.201	.140	9.336	.018	8.812	.002
	1000	55.474	9.311	11.873	.215	9.863	.016	9.226	.002
.01	250	17.694	2.300	9.275	.095	7.783	.029	7.237	.002
	500	32.335	4.505	10.664	.056	8.754	.014	8.237	.003
	750	48.334	7.246	11.538	.128	9.360	.015	8.802	.003
	1000	64.172	12.004	11.835	.184	9.767	.025	9.250	.003

TABLE 4-1

AVERAGE PATH LENGTH

Sequentialness (P)	Tree Size (n)	BALANCE LIMITS (α)							
		.0		.1		.2		.2929	
		Mean	Var.	Mean	Var.	Mean	Var.	Mean	Var.
.0000	250	0.0	0.0	13.4	23.6	40.3	25.6	90.3	54.5
	500	0.0	0.0	26.0	58.4	80.2	51.3	180.6	137.4
	750	0.0	0.0	38.0	56.7	122.7	77.6	265.6	274.0
	1000	0.0	0.0	49.2	74.4	165.1	75.9	360.8	379.9
.0025	250	0.0	0.0	14.6	13.8	43.9	16.3	92.1	63.9
	500	0.0	0.0	32.9	17.9	89.9	35.7	189.1	181.2
	750	0.0	0.0	54.8	19.3	138.8	44.8	284.1	234.8
	1000	0.0	0.0	76.6	64.9	186.8	83.5	381.9	255.2
.0050	250	0.0	0.0	16.6	14.0	46.7	19.6	96.4	30.5
	500	0.0	0.0	40.5	54.5	100.2	31.5	193.4	85.1
	750	0.0	0.0	65.6	78.0	150.7	50.5	299.3	87.7
	1000	0.0	0.0	94.2	119.7	205.4	46.7	399.8	131.1
.0075	250	0.0	0.0	19.0	28.9	47.3	33.6	95.0	85.6
	500	0.0	0.0	48.8	51.7	100.0	26.0	199.5	133.8
	750	0.0	0.0	79.5	75.2	159.2	78.6	305.6	132.4
	1000	0.0	0.0	110.0	86.0	215.9	192.5	405.2	135.1
.0100	250	0.0	0.0	22.6	6.3	49.0	15.3	102.0	27.3
	500	0.0	0.0	55.1	35.4	107.2	37.5	205.0	54.0
	750	0.0	0.0	84.6	25.6	163.7	77.6	312.7	135.6
	1000	0.0	0.0	118.1	99.9	220.7	122.2	419.4	103.4

TABLE 4-2

NUMBER OF REBALANCINGS

From a brief glance the data from these runs appears reasonable. As could be expected, as the tree size increased so did the average path length and number of rebalancings. These variables also increased as the amount of sequentialness was raised. When the balance limits were raised, the resulting tree had a smaller average path length but this caused an increase in the number of rebalancings.

To determine which of the independent variables had an effect upon the dependent variables, a three way analysis of variance was performed on the data. A portion of these results are shown in Table 4-3. This analysis indicated that all variables had an effect. Subsequent attempts at regression analysis on this data were only able to produce regression equations that explained at most 80% of the

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Value
P	4	6102.2	1525.5	1817.6
α	3	45880.6	15293.5	18220.9
n	3	8532.3	2844.1	3388.5
pα	12	16609.9	1384.1	1649.1
pn	12	1717.6	143.1	170.5
αn	9	14329.7	159.2	1896.9
pαn	36	5174.8	143.7	171.2
within replicates	720	604.3	.8	
total	799	98951.6		

Analysis of Variance of AVLEN

TABLE 4-3

variation. A closer look at the analysis of variance, especially plotting the interaction of the variables at the various levels, suggested that the variation of the dependent variables could best be

explained if the data were divided into two groups. One where alpha is zero and the tree is allowed to grow uncontrolled, and a second where the balance limits actually control the growth of the tree. To verify that this condition actually existed the data from the simulation runs for which alpha was set at 0 were deleted and the three way analysis of variance was again performed. Table 4-4 shows a portion of the results of this analysis.

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Value
P	4	15.8	4.0	83.8
α	2	488.0	244.0	5161.3
n	3	429.0	143.0	3024.6
pα	8	24.4	3.0	64.4
pn	12	1.7	.1	3.0
αn	6	9.2	1.5	32.4
pαn	24	3.3	.1	3.0
Within	540	25.5	.05	
Total	599	996.9		

Analysis of Variance of AVLEN

TABLE 4-4

A comparison of the two sets of results shows that with the group of data where alpha was zero removed, the interaction of the independent variables was greatly reduced and subsequent attempts at regression analysis were able to explain in excess of 95% of the variation in the dependent variables with relatively simple models.

As the purpose of this investigation is to evaluate the performance of the BBT technique, and since where $\alpha = 0$ the technique is never allowed to function, it was decided to delete the

data obtained where $\alpha = 0$ and replace it with corresponding data by rerunning the model setting α at .05. After the appropriate 160 simulation runs were rerun the analysis of variance and regression analysis were again performed upon the new data.

With the conditions that complex transformations would be avoided and that regression analysis would be limited to second degree terms, two regression equations were developed that would predict y, the expected average path length, and w, the expected number of rebalancings to be experienced when an update (addition or deletion) is made to a tree of size n. To obtain these equations only one transformation was made. That transformation is $z = (\log_2 n)-1$. The regression equations are

$$y = 1.167z - 1430p\alpha + 43.79pz + 103.1\alpha^2 - 5.7\alpha z - .06z^2$$

$$w = -6.36p - 277.3p^2 + 1.842pz + 4.125\alpha^2$$

These regression equations were able to explain 99.6% of the variation in the observed values of y and 99.9% of the variation in the observed values of w. For situations where no sequentialness was introduced these equations should be able to be reduced by simply removing the terms which contain p. They would reduce to:

$$y^1 = 1.167z + 103.1\alpha^2 - 5.7\alpha z - .06z^2$$

$$w^1 = 4.125\alpha^2$$

In order to check these equations similar regression analysis

and analysis of variance were performed on the data obtained when p = 0. As this subset of data contained only 160 data points an additional 40 simulation runs were made setting α at .15 and .25 which increased the number of data observations to 240.

The regression equations obtained from this data were:

$$y' = -5.66\alpha + 1.40z + 18.56\alpha^2 - .99\alpha z$$

$$w' = 4.01\alpha^2$$

with 99.9% and 99.1% of the respective variation being explained.

A visual examination shows that for w the expected and actual equations are very close whereas the two equaions for y appear quite different. In an attempt to explain the seemingly large difference between the two equations for y the regression analysis was repeated several times allowing only the variables with relatively large F values to enter the equation. The resulting equations were:

$$y = 1.17z - 10920p^2 - 2818p\alpha + 84.4pz$$

$$y' = 1.28z - .9\alpha z$$

and explained 98.7% and 99.9% of the respective variation.

When the terms containing p are removed from the equation for y the two equations appear to be more equal than previous but there is still some discrepancy between them. This indicates that even though the regression equation for y explains 99.6% of the

variation of y, that a large part of the remaining .4% of the variation occurs when p = 0, and that for cases where extreme accuracy is desired, the equation for $y^{\perp}$ should be used.

The actual data upon which the previously described analyses were performed were the cell means which were obtained by performing ten runs within each cell. In order to attempt to predict how subsequent observed values of AVLEN and NREB could be expected to deviate from the values predicted through the use of the previously mentioned regression equations an analysis of how much the observed values varied within each cell was made. The resulting equation for the variance are given in table 4-5. The equation for $y^{\perp}$ and $w^{\perp}$ were obtained using only the data obtained when p = 0.

Variable	% of Variation Explained	Regression Equation
V(y)	71.16	$2.55\alpha^2 - .156\alpha y + .0024y^2$
V(w)	84.82	$.0086\alpha - .001w - .082p\alpha + .01\alpha z$
$V(y^{\perp})$	95.04	$.982\alpha^2 - .016zy^{\perp} + .0146y^{\perp 2}$
$V(w^{\perp})$	93.58	$.009w^{\perp} - .009\alpha^2 - .00065zw^{\perp}$

Expected Variance of Observed Values

TABLE 4-5

The low percentage of variance explained by the above regression equations is probably due to making only 10 observations within each cell but they are high enough to give a reasonable estimate as to how much variation from the predicted value could be expected if a subsequent observation were taken.

CHAPTER V

SUMMARY AND CONCLUSIONS

When Mr. Nievergelt proposed his technique [14] he made several assumptions in order to be able to analytically calculate the expected performance of his technique. The main assumption that he made was that the distribution of node balances throughout the entire tree was uniform over the interval α to $1-\alpha$. He also admitted that this was a very weak assumption and that the distribution was probably more like a truncated normal distribution. Using a uniform distribution, he was able to show analytically that the number of rebalancings that could be expected when a record is added or deleted from an existing tree to be $2/(1-2\alpha)$

His proof also stated that the number of rebalancings is independent of the size of the tree to which the update is made. The results in Chapter 4 support that the expected number of rebalancings is independent of the tree size by the fact that no term in the regression equation contained the variable n or z , but shows a large difference in the number of rebalancings expected by the analytic proof. Table 5-1 shows the difference.

α	Nievergelt	Simulation
.29	4.83	.36
.25	4.00	.25
.20	3.33	.16
.15	2.86	.09
.10	2.50	.04
.05	2.22	.01

Expected Number of Rebalancings

TABLE 5-1

The large difference between the two predictions either supports Mr. Nievergelt's theory that his assumption was a poor one or that there is an error in this proof.

Mr. Nievergelt does not give a method of predicting the expected average path length of the resulting trees. He only states that for an alpha value of $1-\sqrt{2}/2$ or approximately .293, that the expected average path length is less than 1.115 $\log_2 n$ and probably close to 1.05 $\log_2 n$. Using the regression equation from Chapter 4 one would get

$$y' = -5.66(.293)+1.4((\log_2 n)-1)+18.56(.293)^2-.99(.293)((\log_2 n)-1)$$

which simplifies to

$$y' = 1.11 \log_2 n - 1.32$$

which is less than 1.115 $\log_2 n$ and for certain values of α it is close to 1.05 $\log_2 n$.

One of the purposes of this study was to be able to predict the performance of this technique so that for a specific environment an efficient choice of alpha could be made so as to be able to tailor the technique performance to the requirements of the user. Through the use of the suggested regression equations it is now possible to do this.

The results of this study do not provide for a direct comparison of this technique to others such as the A.V.L. method. While the

resulting trees of each method can be compared on the basis of expected average path length, they cannot be effectively compared on number of rebalancings. If method A requires twice as many rebalancings as method B, it does not mean method B is better as the rebalancing algorithm for method B may take three times as long execute as that of method A. For an effective comparison to be made, some type of cost model must be developed for each technique. This model should include the cost of items such as storage space for both program and data and the cost of computer time to execute a file inquiry, update and restructuring. Only after this type of costing model has been developed for two or more techniques can the performance of these techniques in a given environment be compared. This study has provided sufficient understanding about the Bounded Balanced technique that such a model could now be built.

This study only considered the behavior of the tree in a static environment, that is the input variables α and p were held constant while n was increased. The study could now be extended to include the behavior of the technique in a transient environment. If one were to build a tree using specific values for α ,p and n, the characteristics of the tree can be easily predicted from the equation in Chapter 4. After this initial tree is completed the values of α or p could be changed and the building of the tree continued. From this study it would be possible to predict what the tree would be like if the variables had been at their new level for the entire life of the tree. Using these characteristics, it is possible to

observe the actual growth of the tree and determine the tree's behavior during its transition from its old set of characteristics to the new set that it must approach as growth continues.

It is the conclusion of this study that the technique of Bounded Balanced Binary Tree is competitive enough to be considered for use as a technique for physical storage management of data storage files but the decision as to which method is best is dependent upon the environment in which it is used.

BIBLIOGRAPHY

1. Bachman, C.W., "The Evolution of Storage Structures", Comm. of A.C.M., July, 1972, pp. 628 - 636.

2. Buchholtz, W., "File Organization and Addressing", I.B.M. Syst. J., June, 1963, pp. 86 - 111.

3. Clampelt, H.A., "Randomized Binary Searching with Tree Structures", Comm. of A.C.M., March, 1964, pp. 163 - 165.

4. Flores, J., and Madpis, G., "Average Binary Search Length for Dense Ordered Lists", Comm. of A.C.M., Sept., 1971, pp. 602 - 603.

5. Foster, C.C., "A Generalization of A.V.L. Trees", Comm. of A.C.M., August, 1973, pp. 513 - 517.

6. Johnson, L.R., "An Indirect Chaining Method for Addressing on Secondary Keys", Comm. of A.C.M., April, 1961, pp. 218 - 223.

7. Kennedy, S., "A Note on Optional Doubly-Chained Trees", Comm. of A.C.M., November, 1972, pp. 997 - 998.

8. Landauer, W.I., "The Balanced Tree and Its Utilization in Information Retrieval", IEEE Trans. on Elec. Computer, 1963, pp. 863 - 871.

9. Lum, V. Y., "General Performance Analysis of Key-to-Address Transformation Method Using an Abstract File Conept", Comm. of A.C.M., October, 1973, pp. 603 - 612.

10. Martin, W.A., and Ness, D.N., "Optimizing Binary Trees Grown with A Sorting Algorithm", Comm. of A.C.M., February, 1972, pp. 88 - 93.

11. Maurer, W.D., "An Improved Hash Code for Scatter Storage", Comm. of A.C.M., January, 1968, pp. 35 - 38.

12. McIlroy, M.D., "A Variant Method of File Searching", Comm. of A.C.M., March, 1963, p. 101.

13. Morris, R., "Scatter Storage Techniques", Comm. of A.C.M., January, 1968, pp. 38 - 44.

14. Nievergelt, J., and Reingold, E.M., "Binary Search Trees of Bounded Balance", Proc. of 4th Annual A.C.M. Symp. on Theory of Computing, 1973, pp. 137 - 142.

15. Nievergelt, J., "Binary Search Trees and File Organization", Sigfidet Workshop Data Descriptions and Access Control, 1972, pp. 165 - 187.

16. Overholt, K.J., "Optimal Binary Search Methods", BIT, 13 - 1, 1973, pp. 84 - 91.

17. Price, C.E., "Table Lookup Techniques", A.C.M. Computing Surveys, June, 1973, pp. 49 - 65.

18. Scidmore, A.K. and Weinberg, B.L., "Storage and Search Properties of a Tree-Organized Memory System", Comm. A.C.M., January, 1963, pp. 28 - 31.

19. Shniederman, B., "Optimum Data Base Reorganization Points", Comm. of A.C.M., June, 1973, pp. 362 - 365.

20. Stanfel, L.E., "A Comment on Optimal Tree Structures", Comm. of A.C.M., October, 1969, p. 582.

21. Stanfel, L.E., "Practical Aspects of Double Chained Trees for Retrieval", Journal of A.C.M., July, 1972, pp. 425 - 436.

22. Stanfel, L.E., "Tree Structures for Optimal Searching", Journal of A.C.M., July, 1970, pp. 508 - 517.

23. Sussenguth, E.H., Jr., "Use of Tree Structures for Processing Files", Comm. of A.C.M., May, 1963, pp. 272 - 279.

24. Tan, K.C., "On Foster's Information Storage and Retrieval Using A.V.L. Trees", Comm. of A.C.M., September, 1972, p. 843.

25. Ullman, J.D., "A Note on the Efficiency of Hashing Function", Journal of A.C.M., July, 1972, pp. 569 - 575.

26. Walker, W.A., and Gotlieb, C.C., "Hybrid Trees - a Data Structure for Lists of Keys", Sigfidet Workshop - Data Description and Access Control, 1972, pp. 189 - 211.

27. Zimmerman, B., Lefkovitz, D., and Prywer, N.S., "The Naval Aviation Supply Office Inventory Retrieval System -- A Case Study in File Merchanization", Moore School, University of Pennsylvania, Philadelphia, Pa., Tech. Rept. for Office of Naval Research, Bureau of Supplier and Accounts, NOnr551(40); 1963.

APPENDIX I

DEFINITION OF SCOPE

As was mentioned in Chapter III the model developed for this study includes four commands which deal with what is called scope. Scope is a technique by which records whose keys have prefixes can be stored. An example of such a key is A. KEY 1 where KEY 1 is the record key and A is a prefix.

Before such a record is stored the prefix is removed from the record. Within the storage area a separate bounded balanced binary tree is maintained which contains all the unique prefixes that have been used so far. The data portion of the records in this tree contain not data, but the address of the root node of the tree which contains all the records that have that unique prefix. By doing this it is not necessary to store the prefix with each record yet the model retains the ability to distinguish between records having identical keys but different prefixes. If a record does not have a prefix, it is stored exactly as described in Chapter III.

There are four commands that make possible the use of scope. The first is DECLARE SCOPE (command code 'P'). When this command is issued the key pointer field must point to a field which contains only the prefix with which all subsquent records are to be processed. When the model receives this command, it will add the prefix record to the scope tree and all subsequent records processed will be assumed to have this prefix although the prefix will not be present in the keys.

If the user wishes to change the scope or prefix presently in use, if any, to one which has previously been declared, this can be accomplished with the SET SCOPE command (command code 'E'). When this command is issued the model will delete the previous scope declaration, if any, and assume that all subsequent records to be processed are preceeded by the new prefix which is pointed to by the key pointer field.

If after setting or declaring scope the user wishes to continue processing without using prefixes the user can use either the FREE SCOPE or RELEASE SCOPE commands. (Command codes 'F' and 'R'). If the prefix presently active is to be removed from the SCOPE tree the command RELEASE SCOPE is used. This command can only be successfully executed when no records exist in the file containing the prefix presently active. The free scope command simply assumes that processing is to continue without using prefixes, but that those records in storage which have prefixes are maintained as is the SCOPE record of the prefix which was active when the command was issued.

There is a version of the model under development which will process records having multiple levels of prefixes such as A.B.C. KEY 1. This version uses the same four scope commands and a similar technique of separating the prefixes from the actual keys but has not yet been completely debugged.

VITA

PERSONAL HISTORY

Name: R. Ian Bardsley

Birth Place: Ashton-under-Lyne, Lancashire, England

Birth Date: September 8, 1944

Parents: Victor and Eva Bardsley

Wife: Joanne M. Bardsley

Children: Dawn M. Bardsley

EDUCATIONAL BACKGROUND

State University College of New York at New Paltz - Bachelor's Degree in mathematics - 1966

Lehigh University

Candidate for Master of Science Degree in Industrial Engineering

PROFESSIONAL EXPERIENCE

Western Electric Company, Inc.

Headquarters Finance Division

Member of Information Systems Staff (1966-1972)

Personnel Division - Lehigh Program

Member of Information System Staff (1972-1974)