THE DESIGN AND APPLICATION OF A RESEARCH TOOL FOR HEIGHT BALANCED TREES

By

MARY BETH HERNON

Bachelor of Science in Home Economics

Oklahoma State University

Stillwater, Oklahoma

1976

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1979

THE DESIGN AND APPLICATION OF A RESEARCH

TOOL FOR HEIGHT BALANCED TREES

Thesis Approved:

James P. Van Doren

Thesis Adviser

J. P. Chandler

Norman N. Durham

Dean of Graduate College

PREFACE

This study is concerned with the development and extension of a class of height balanced binary search trees known as HB(k) trees. HB(k) trees are an important alternative data structure in file systems where rapid access and rapid update are desired. However, a precise analysis of expected system performance using HB(k) trees is impossible since a precise analysis of the expected behavior of HB(k) trees remains unformulated. A generalized class of HB(k) trees, known as PHB(k1,k2) trees, may provide the tool necessary to analyze the expected behavior of HB(k) trees. The design of algorithms for maintaining these trees and the subsequent implementation of the algorithms as part of a research tool for height balanced trees are also discussed. Results from an initial use of the research tool are presented.

I would like to acknowledge the intellectual stimulation and encouragement provided by Dr. James R. Van Doren, my major advisor, not only in support of this thesis but also throughout my graduate education. Under his tutelage, every course has been both challenging and rewarding.

I also acknowledge the encouragement received from my committee members, Dr. Donald W. Grace, Dr. John P. Chandler, and Dr. Donald D. Fisher. This thesis is better because of their support.

A special note of recognition is given for the love, understanding, and support given me by my husband, Bill. His presence has enriched my graduate education.

I would also like to thank my parents, Robert and Jacquetta Jacks, and my parents-in-law, Peter and Bertie Hernon, for their support, both moral and financial, which has helped us over some hard times.

Finally, I would like to express appreciation for the friendship and comraderie I found among other graduate students.

TABLE OF CONTENTS

CHAPTER Page

I. INTRODUCTION 1

II. BINARY SEARCH TO BINARY SEARCH TREES 4

Binary Search Technique 5
The Binary Search Tree 8

III. HEIGHT BALANCED BINARY SEARCH TREES 20

HB(k) Binary Search Trees 21
Performance of HB(k) Trees 35
Alternatives to HB(k) Balanced Binary Search Trees 37

IV. PARTIALLY HEIGHT BALANCED TREES 41

Structure and Maintenance of PHB(k1,k2) Trees 43
PHB(1,1) Trees 47
Performance of PHB(k1,k2) Trees 52

V. A RESEARCH TOOL FOR HEIGHT BALANCED TREES 53

Logic Design 54
Using the Research Tool 55
Application of the Research Tool 55

VI. SUMMARY AND CONCLUSIONS 60

Results of the Study and Suggestions for Future Study 61
Expanding the Capabilities of the Research Tool 62

A SELECTED BIBLIOGRAPHY 63

APPENDIX A - DERIVATION OF BALANCE TAG MAINTENANCE EQUATIONS 66

APPENDIX B - HEIGHT OF THE SUBTREE DURING INSERTION ROTATION 81

APPENDIX C - BNF DESCRIPTION OF INPUT TO THE RESEARCH PROGRAM 86

APPENDIX D - AN ILLUSTRATION OF THE USE OF THE COMMAND LANGUAGE 92

LIST OF TABLES

TABLE Page

I. Test Case Information 56

II. Results from Test Case 1 57

III. Results from Test Case 2 58

IV. Results from Test Case 3 59

LIST OF FIGURES

Figure		Page
1.	Binary Search Algorithm	6
2.	Actions of Binary Search Algorithm	7
3.	A Binary Tree	9
4.	Binary Tree vs. Binary Search Tree	10
5.	BST Representation of Table of Letters	10
6.	Search Algorithm for Binary Search Trees	11
7.	Insertion Algorithm for Binary Search Trees	12
8.	Result of Inserting D into Binary Search Tree	13
9.	Result of Deleting G from Binary Search Tree	14
10.	Deletion Algorithm for Binary Search Trees	14
11.	A Degenerate Tree	16
12.	A Binary Search Tree Extended	17
13.	A Worst Case AVL Tree with 12 Nodes	22
14.	Balance Tag Maintenance in an HB(k) Tree	27
15.	Cases 1 and 3 as Mirror Images	28
16.	Simple Rotation in an HB(k) Tree	29
17.	Balance Tag Maintenance After a Simple Rotation	29
18.	Split Rotation in an HB(k) Tree	30
19.	Cases 2 and 4 as Mirror Images	31
20.	Balance Tag Maintenance After a Split Rotation	31

21. An HB(1) Tree 42

22. Result of Insertion Depends on Classification . . 42

23. A PHB(2,2) Tree After Insertion of a Node 45

24. The PHB(2,2) Tree After Restructuring 46

25. A Worst Case for PHB(1,1) Trees 47

26. Insertion Algorithm for PHB(1,1) Trees 49

27. Deletion in a PHB(1,1) Tree 50

28. Restructuring of a PHB(1,1) Tree After Deletion . 51

CHAPTER I

INTRODUCTION

An immense information explosion in the 60's and 70's has intensified the issue of how to warehouse information: How may information be stored so that any particular piece or any related group of pieces may be quickly retrieved for examination?

Computer hardware technology has provided part of a solution: the appropriate warehouse, machines with an ever increasing capability of storing volumes of information in small amounts of space. Computer software technology has provided another part: methods or structures for organizing the information (data) held within the machine. The motivation behind this study concerns the evolution of one such class of data structures, height balanced binary search trees, and the development of a research tool to aid in the theoretical analysis of the behavior of the trees. Height balanced binary search trees have been well-documented empirically but lack a definitive theoretical explanation for their behavior.

Chapter II traces the development of height balanced binary search trees from the early days of computing during which the impetus for binary search trees developed out of the binary search technique. The binary search tree as a logical entity was not presented until the early 1960's, but like an idea whose time has come, much attention was given to binary search trees in subsequent years.

Chapter III discusses the logical variants of binary search trees. These structures, each of which balances the tree in some way, have been developed in the attempt to obtain the best-possible worst case. The height balanced binary search tree, one of the variants, is selected for a detailed discussion of its structure, maintenance, and performance.

Chapter IV presents a generalization of height balanced binary search trees, partially height balanced binary search trees. This logical structure has been proposed as an aid in the effort to rigorously define the performance of height balanced trees. Yet, a subclass of this structure may prove to have interesting properties in and of itself.

Chapter V discusses the research tool developed to provide empirical data on the performance of binary search trees, height balanced binary search trees and partially height balanced binary search trees. An overview of the logic design and program structure is presented along with preliminary instructions for using the programs that have been written.

Application of the research tool for providing empirical data which may lead to a rigorous definition of the performance of height balanced binary search trees is also discussed. Some initial results concerning the subclass of partially height balanced trees which may become of some importance in its own right are presented.

Chapter VI summarizes the major ideas and findings of the study. Suggestions for further study and for expanding the capability of the research tool are also made.

CHAPTER II

BINARY SEARCH TO BINARY SEARCH TREES

The foundation for the development of height-balanced binary search trees was laid in the early days of computing by the binary search technique. The binary search was well known in the early 1940's although the first formally published algorithm which works for any number of items in the table was presented in 1962 (13). Use of a binary search can reduce tremendously the amount of effort devoted to one of the most frequent activities for any collection of information - looking for a particular item based upon a particular, unique identifier, called the key, such as a name or an account number. If there is no particular ordering of the items of information, then one must use a 'brute-force' approach, conventionally called a linear search, to find the desired item: beginning with the first of all items and examine each one in turn until the desired item is found or the item list is exhausted. This is somewhat akin to trying to find someone's phone number in a telephone directory in which people are listed in the order in which they requested phone service. The only recourse is to start with the first person listed and look through all people listed until the one desired is found.

The tediousness of such an approach should be apparent. On the average, approximately half the items are examined to find the desired item. In the worst case or if the item is not present, then the entire list of information is examined.

Order, lexicographic (dictionary-like) or numeric, greatly eases the burden of locating an item. If one is trying to find Peterson's phone number in the standard telephone directory, then one initially aims directly for the P's and thus eliminates the entire first half of the directory at once. If one happens to open to the N's then several pages are flipped in an attempt to get to the P's, thus eliminating many more entries from consideration at once.

Binary Search Technique

This is the essence of the binary search technique - given a list of items which are logically and physically in order, search for the desired item by successively eliminating from consideration unneeded portions of the list. However, for computer applications, this approach is rigorously formalized as in Figure 1.

The actions of the algorithm searching for the key P in a table of letters are illustrated in Figure 2. It can be shown (13) that the binary search technique makes at most lg(n) + 1 comparisons for an unsuccessful search and makes

```
BEGIN SEARCH(desired key,midpoint);
left_boundary  <- location of first item;
right_boundary  <- location of last item;
midpoint  <- FLOOR((left_boundary+right_boundary)/2)
DO UNTIL (left_boundary > right_boundary);
  IF key at midpoint is desired key
    THEN END SEARCH;
  END IF;
  IF desired key < key at midpoint
    THEN right_boundary  <- midpoint;
    ELSE left_boundary   <- midpoint;
  END IF;
END DO;
key not found in table;
END SEARCH;
```

Figure 1. Binary Search Algorithm

lg(n) - 1 comparisons for the average successful search ('lg' indicates the base 2 logarithm and will be used as such throughout the discussion without further explanation).

```
left_boundary=1     midpoint=4      right_boundary=7
                                 +---+
location     1      2      3     | 4 |    5      6      7
key          B      C      F     | G |    H      P      V
                                 +---+

left_boundary=4     midpoint=5      right_boundary=7
                                        +---+
location     1      2      3      4     | 5 |    6      7
key          B      C      F      G     | H |    P      V
                                        +---+

left_boundary=5     midpoint=6      right_boundary=7
                                               +---+
location     1      2      3      4      5     | 6 |    7
key          B      C      F      G      H     | P |    V
                                               +---+
```

Figure 2. Actions of Binary Search Algorithm

The time complexity has been reduced from O(n) for the brute force sequential search to O(log(n)) for the binary search.

Restrictions

The binary search technique is the best possible search algorithm that proceeds solely by comparing (the desired key) to keys in the table (11). However, the restriction that the keys be stored consecutively in a specified order has different implications when one considers activities other than searching such as inserting a new item or deleting an old one.

In order to insert (delete) an item one might do the following steps:

1. Determine the correct location for (of) the item. In terms of the binary search technique presented earlier, location = right boundary upon termination of an unsuccessful search for insertion and for deletion location = midpoint upon termination of a successful search.
2. Move all items between location and the end of the table down (up) one position.
3. For insertion, insert the new item at location.

This is potentially a very time-consuming task. With dynamic tables, tables which are constantly changing, reorganization time may far outweigh access time, time spent searching the table. For some applications this may be of no concern, but for others, such as an airlines reservation system, reorganization time may interfere with the rapid access time desired. Hence, it would be desirable to develop an approach to information storage that would give not only small search times but also small insertion and deletion times for any item of information. The binary search tree is such an approach.

The Binary Search Tree

Windley (26), and Booth and Colin (4) independently introduced binary search trees as logical and physical structures in 1960. Many of the later publications reporting work concerning binary search trees reference these two articles. The concept of binary search trees has been generalized to binary trees. A binary tree is

> a finite set of nodes which either is empty, or consists of a (node called the) root and two disjoint binary trees called the left and right subtrees of the root (12, p. 309).

Each node (or element) of the tree contains several items of information: a key by which one may uniquely identify the node, and two "pointer" fields which identify (or point to) the locations of the root nodes for the left and right subtrees. Other information relevant to the key may be stored in a node, but it is not a concern of this discussion. Figure 3 illustrates a binary tree.

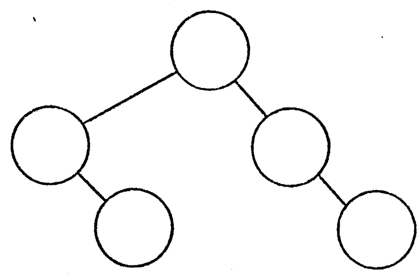


Figure 3. A Binary Tree

If the binary tree is to be used to maintain an ordered set of records, then a further requirement is that all nodes in the left subtree have keys which are less than the key in the root node in some sense whether numerically or lexicographically. In order to picture this, it is helpful to consider flattening the tree so that all nodes are aligned such that if node X were in the left subtree of node Y, then node X is to the left of node Y in the line. Such a binary tree is usually called a binary search tree (BST) or binary decision tree. Thus, if we let A and B represent the keys of the nodes in Figure 4, then Figure 4 (a), while a valid binary tree, is not a valid binary search tree. Figure 4 (b) is a valid binary search tree.

It may be helpful in understanding how a binary search tree is organized to consider that the binary search technique discussed earlier imposes an implicit tree structure upon a linearly ordered set of items. The initial midpoint is the root of the entire tree; the midpoint of the half-

Figure 4. Binary Tree vs. Binary Search Tree

list to the left of the initial midpoint is the root of the left subtree of the root of the entire tree; the midpoint of the half-list to the right of the initial midpoint is the root of the right subtree of the root of the entire tree; and so on. This is illustrated graphically in Figure 5.

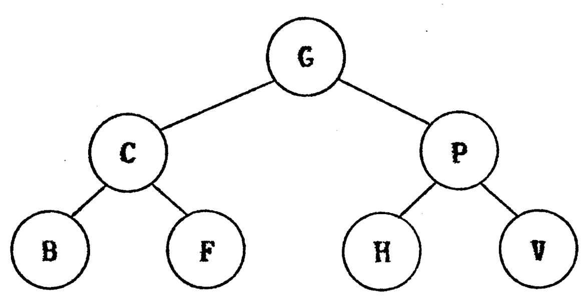


Figure 5. BST Representation of Table of Letters

The node with key G is called the 'parent' or immediate ancestor of the nodes with keys C and P. Conversely, the nodes with keys C and P are called 'siblings' and are the immediate 'descendants' or 'offspring' of the node with key G. In particular, C is the left offspring of G and P is the right offspring of G. Additionally, B, F, H, and V are leaf nodes (no offspring), and C, G, and P are interior nodes (two offspring).

It should be apparent that since each node now contains, or points to, the location of the next node to be examined, there is no need to require that the items be stored in order in consecutive locations. However, there must be a way to tell when there are no more nodes to examine; hence, a NULL value must be established for pointers which do not point to any offspring. The search algorithm for the binary search tree is illustrated in Figure 6.

```
BEGIN SEARCH(desired key,NODE,PARENT);
PARENT  <- NULL;
NODE  <- location of root of entire tree;
DO WHILE (NODE is not NULL);
  IF desired key = key at NODE
    THEN END SEARCH;
  END IF;
  PARENT  <- NODE;
  IF desired key < key at NODE
    THEN NODE  <- LEFT(NODE);
    ELSE NODE  <- RIGHT(NODE);
  END IF;
END DO;
key not found;
END SEARCH;
```

Figure 6. Search Algorithm for Binary Search Trees

Insertion is a relatively straightforward procedure although one must be careful to maintain the order associated with the structure. Figure 7 presents the algorithm for inserting a new item into the tree. Let us insert the key D into the tree of Figure 5. The search algorithm would detect a NULL value to the LEFT of the node for F and return the PARENT = location of F. The INSERT algorithm would then put D into the next available node and this node would become the LEFT descendant of F. Additionally, F is no longer a leaf node but is now a semi-leaf node (one descendant). The resulting tree would then appear as in Figure 8.

```
BEGIN INSERT(new key);
CALL SEARCH(new key,node,parent);
IF new key < key at parent
  THEN LEFT (parent) = next available node;
  ELSE RIGHT(parent) = next available node;
END IF;
Place new key in next available node;
END INSERT;
```

Figure 7. Insertion Algorithm for Binary Search Trees

Deletion of a node is more complicated, however. For instance, if one were to delete the node with key G from the tree in Figure 8, then its descendent subtrees would no longer be subtrees of a common root. They would be 'dangling subtrees' or distinct binary search trees with no

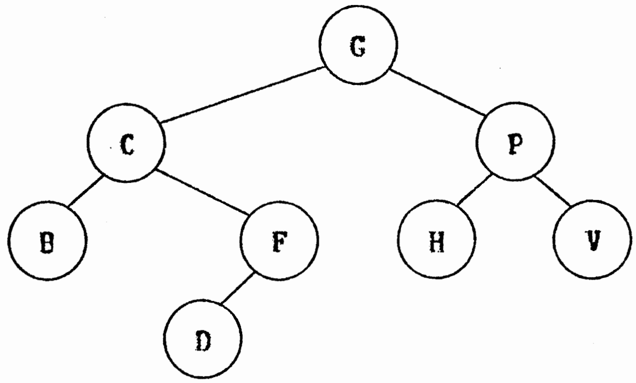


Figure 8. Result of Inserting D into Binary Search Tree

logical interconnection. Some way must be found to maintain the relationship between all nodes remaining in the tree. The problem is usually approached as follows:

1. Find the largest (smallest) key in the LEFT (RIGHT) subtree of the node to be deleted.

2. Substitute this node for the one being deleted being careful to reconnect all subtrees of the two nodes involved. (This substitution involves changing at most four pointers only.)

3. Return deleted node to an available pool.

The result of this algorithm after deleting the node G from Figure 8 is shown in Figure 9. Stated more formally, the algorithm for deletion is illustrated in Figure 10.

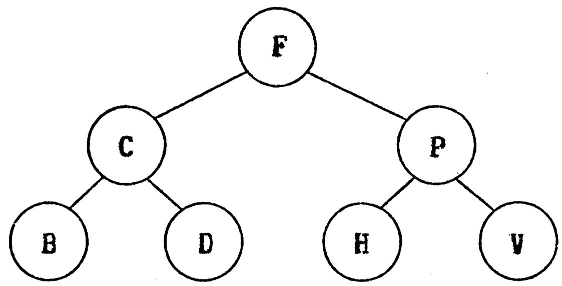


Figure 9. Result of Deleting G from Binary Search Tree

```
BEGIN DELETE(old key);
CALL SEARCH(old key,NODE,PARENT);
Find largest key in LEFT subtree of NODE;
RIGHT(parent of largest key)  <- LEFT(largest key);
LEFT(NODE of largest key)  <- LEFT(NODE);
RIGHT(NODE of largest key)  <- RIGHT(NODE);
Return NODE to available pool;
END DELETE;
```

Figure 10. Deletion Algorithm for Binary Search Trees

Time Complexity of Binary Search Tree Algorithms

A reasonable question that must be asked involves the time complexity of the algorithms associated with binary search trees. How long does it take to search the tree, to insert a new item into the tree, to delete an item?

For both deletion and insertion the average time complexity approximates that for an unsuccesful search. The changes made to the pointers are done in a constant amount of time which is negligible for trees containing large numbers of nodes.

The time complexities associated with the best, average, and worst case, in terms of average search time, binary search trees have been extensively documented (4, 5, 8, 13, 18, 21, 26). If one considers all nodes on one 'row' to constitute a 'level', then the best case binary search tree has all leaf and semi-leaf nodes on at most two adjacent levels. This is sometimes termed a complete binary tree. This corresponds precisely to the binary search tree interpretation of the binary search technique. The time complexity for searching the tree is O(log(n)) where n is the number of nodes in the tree.

A worst case, called a 'degenerate' tree, arises when all keys are inserted in order. If the keys in Figure 5 were inserted in lexicographic order then the tree would appear as in Figure 11. Searching a degenerate tree structure is equivalent to the sequential search discussed earlier; the time complexity is O(n).

However, if one assumes that the keys are inserted randomly then it can be proved that the time complexity approximates that for the best case since well balanced trees are common and degenerate trees are rare (13).

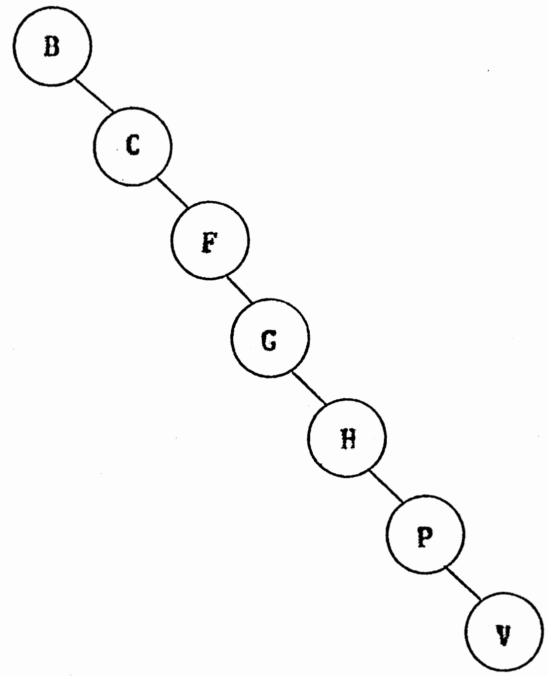


Figure 11. A Degenerate Tree

Terminology in Empirical Measurements

Much work in data structures has been done to try to guarantee that a degenerate tree never occurs. But before discussing some of this work, if would be helpful to define the terms commonly used in discussions of empirical performance of the data structures.

Since the time complexity for the algorithms for binary search trees are directly proportional to the number of comparisons made during searching the tree, performance concepts which may be measured empirically have been well-defined (although minor variations still exist). These include the height, the internal path length, and the external path length.

The level of a node corresponds to which 'row' it is on, the root node being level 1. Thus, in Figure 11, B is on level 1, C is on level 2, and so on. The height of a binary search tree or a subtree is the number of levels in the tree or subtree.

In order to formalize an empirical measurement for successful and unsuccessful searches, it is helpful to introduce the concept of 'external nodes'. An external node is a special node used to indicate a NULL subtree in the graphical representation of a tree. Of the nodes in Figure 12 (a), nodes A and D have two NULL subtrees, and node C has a NULL LEFT subtree. Figure 12 (b) shows the representation for and placement of external nodes. Nodes A, B, C, and D are now termed 'internal nodes'.

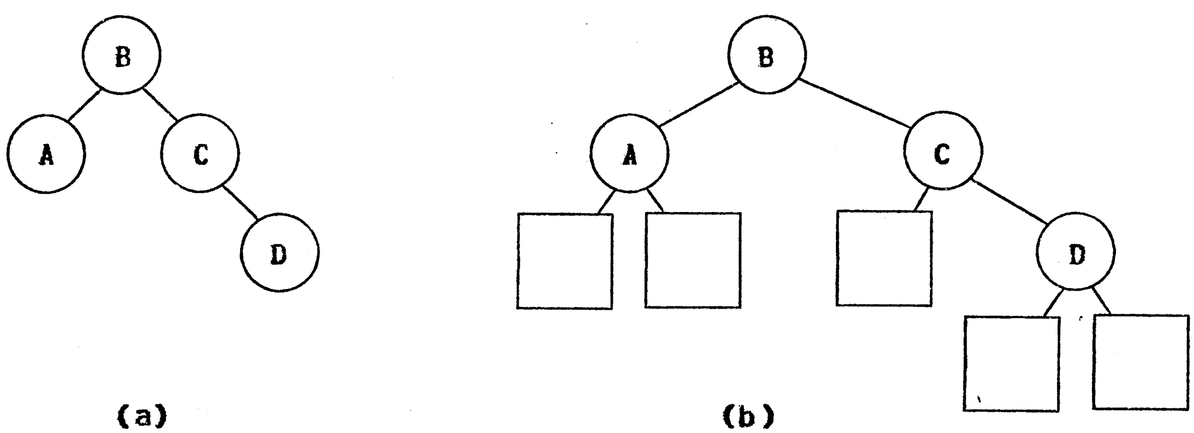


Figure 12. A Binary Search Tree Extended

For all trees, the following relationship holds:

number of external nodes = number of internal nodes + 1.

Figure 12 (b) has been termed an extended binary tree.

The path length between two nodes is the difference between their level numbers. Thus, in Figure 12 (b), the path length between B and D is 2, between D and one of its external nodes, it is 1. The path length may also be thought of as the number of additional comparisons needed to locate a particular node in a subtree from the root node of the subtree. The internal path length of a tree with n nodes, I(n), is the sum of all the path lengths between the root node (level 1) and each internal node. Thus, for Figure 12 (b),

$$I(n) = 1 + 1 + 2 = 4.$$

The external path length, E(n), is the sum of the path lengths between the root node (level 1) and each external node. Thus, for Figure 12 (b),

$$E(n) = 2 + 2 + 2 + 3 + 3 = 12.$$

The relationship between the internal and external path lengths is always

$$E(n) = I(n) + (2 * n).$$

It should be apparent that the average number of comparisons required for a successful search, C(n), is

$$C(n) = 1 + (I(n) / n).$$

One comparison is required to get to the root of the tree and decide which subtree to examine next. The expression $I(n)/n$ gives the average number of comparisons required to get from the root of the entire tree to any other particular internal node in the tree. Similarly, the average number of comparisons required for an unsuccessful search, $C'(n)$, is

$$C'(n) = E(n) / (n + 1).$$

$C(n)$ is a measure of the relative time required to retrieve a particular node from a tree. $C'(n)$ is a measure of the relative time required to insert or to delete a node or to search for a node that is not present.

These measures aid in comparing the relative efficacy of different algorithms designed to manipulate trees and will be used throughout the remainder of the discussion.

CHAPTER III

HEIGHT BALANCED BINARY SEARCH TREES

Even though, as was stated above, randomly constructed binary search trees behave quite well and degenerate trees rarely occur, there still remains the issue of degenerate trees. If, as is quite possible in 'real' applications, items are entered in order, then this wonderful construct, the binary search tree, has saved nothing except for the occasional random insertion. One would like to be able to guarantee a complete binary tree (one with all external nodes on at most two adjacent levels such as the binary tree interpretation of the binary search technique) all the time since this would save considerable searching effort. However, the time involved in maintaining this guarantee should not outweigh the time saved during a search.

One class of data structures that has been proposed to solve this problem is the class of weight balanced trees of which the optimal binary search tree is an example. Weight balanced trees use as a guideline the adage that '80% of the activity occurs in 20% of the file'. Information about frequency of access for each key is used to construct and reconstruct the tree so that the most frequently accessed keys are near the root level. This considerably reduces the average search time for a set of keys with known frequencies. Weight balanced binary search trees are a nice solution if one has a static file and can safely project the frequency of access to each key. However, for dynamic files, ones for which insertion and deletion are major activities, and frequency of access to any particular key cannot be predicted, weight balanced trees create more work than they save since access frequencies must be dynamically maintained and the entire tree must be constantly checked for optimality.

HB(k) Binary Search Trees

A nice solution to the problem of maintaining dynamic trees so that degenerate trees never occur but maintenance requires only local adjustment around a node and one or two of its descendants, was first proposed in 1962 by two Russian mathematicians, Adel'son-Vel'skii and Landis (1). The binary tree structure they proposed, subsequently termed an AVL tree, constrains the relative heights of the LEFT and RIGHT subtrees of the nodes. The height of the left subtree of a node may differ by no more than one from the height of the right subtree. This constraint does not always result in a complete binary tree. Figure 13 illustrates a worst case, in terms of average search path length, C(n), for an AVL tree with 12 nodes. In a complete binary tree, 12 nodes would require only four levels. However, the performance of an AVL tree approximates the best possible performance of a complete binary tree and requires only two bits per node to indicate whether the left subtree is longer than, balanced with, or shorter than the right subtree.

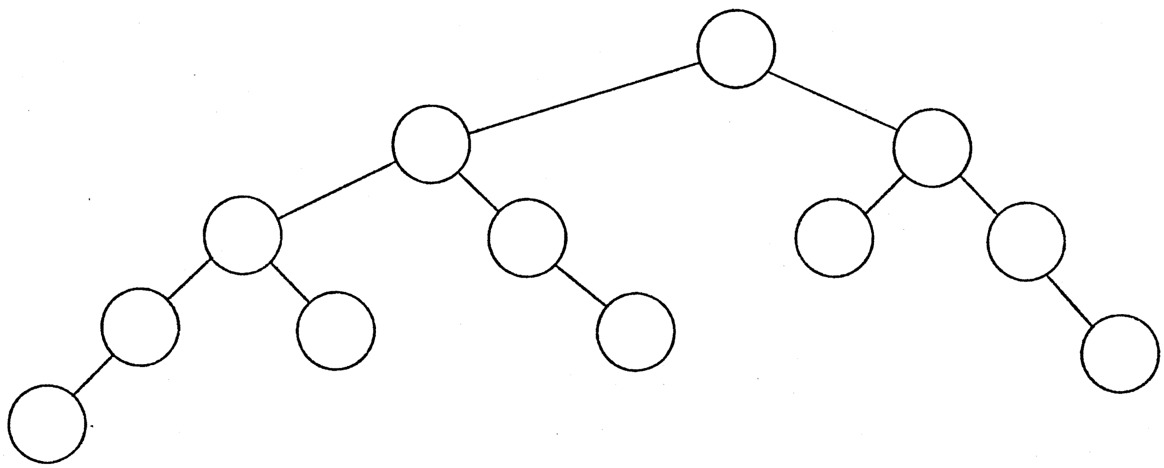


Figure 13. A Worst Case AVL Tree with 12 Nodes

This notion of 'height balanced' was generalized in 1973 by Foster (7) to permit relative height imbalances greater than one. These trees are called HB(k) trees where k, the allowed imbalance, is an arbitrary compromise between short search time and frequency of restructuring. AVL trees may be considered a special case of HB(k) trees - the HB(1) subclass. However, HB(k) trees require more storage per node since the relative imbalance may be between 0 and k for either subtree. The following discussion of structure and maintenance requirements applies equally to AVL and HB(k) trees.

Structure and Maintenance

If one is going to guarantee that the difference between the heights of the left and right subtrees of a node is no more than k, then one must maintain information about the heights with the nodes. One approach to this problem is to maintain the actual height of the (sub)tree rooted at a given node. If one defines the height of a null descendant to be zero, then this may be calculated for all internal nodes simply according to the rule:

Height(node) = MAX (Height(left descendant), Height(right descendant)) + 1.

A node which is critically unbalanced, whose subtrees have relative heights which violate the balance constraint, may be detected by the following test:

ABS (Height(left descendant) - Height(right descendant)) > balance constraint k.

Insertion and deletion may quite possibly change the heights associated with the nodes along the search path and create a critically unbalanced condition for some node. Thus, after insertion or deletion of a node, one must 'backup' along the search path modifying the heights according to the above rule until one of two things occur:

1. The height remains the same for some node.

2. A node is detected to be critically unbalanced.

In the first case, one may terminate the backup for height maintenance. In the second case, one must restructure the tree in order to bring it back into compliance with the balance constraint.

It should be evident that this involves a great deal of work. There are potentially four accesses per node along the search path: one during the search, and three during the backup procedure. It seems reasonable to expect that this method would detract from the usefulness of this data structure.

Fortunately, there is a second approach to the maintenance of height information which does not involve such a great amount of effort. This approach maintains a 'balance tag' for each node which is a measure of the relative difference in heights between the left and right subtrees of the node. The balance tag may be defined as follows:

balance tag(node) = Height(right descendant) - Height(left descendant).

Thus, three cases are established:

1. balance tag(node) = 0: the heights of the two subtrees are equal,

2. balance tag(node) < 0: Height(left descendant) > Height(right descendant), called left heavy,

3. balance tag(node) > 0: Height(right descendant) > Height(left descendant), called right heavy.

Backtracking from the inserted node along the path of insertion /deletion is still required in order to maintain the balance tags.

One should question why the second approach is better than the first, since the second approach defines the balance tag in terms of the heights of the subtrees and backtracking is still required. The answer is that the height need not be maintained; the balance tags may be maintained based upon their previous values. Until backtracking is terminated for insertion, if the new node were inserted in the right subtree, then the height of the right subtree is one greater than before; hence, add one to the balance tag. If the new node were inserted in the left subtree, then the height of the left subtree is one greater than before; hence, subtract one from the balance tag. Deletion from the left (right) subtree is equivalent to insertion in the right (left) subtree. Thus, backtracking involves only one access per node instead of three as with the first approach.

Insertion in an HB(k) Binary Search Tree

Basic insertion is identical to that for unconstrained binary search trees. After insertion, the backup is terminated if either of two cases occur:

1. At any unbalanced node along the search path, the new node were inserted in the shorter subtree. That is, if a node were left heavy and the new node were inserted in the right subtree, or if a node were right heavy and the new node were inserted in the left subtree, the backup maintenance may be terminated.

2. If a node is unbalanced to the point of violating the balance constraint. Two simultaneous conditions determine this case:

   a. ABS (balance tag(node)) = balance constraint,

   b. The node was inserted in the longer or heavy subtree.

   In this case, the tree must be restructured to conform to the constraints.

Figure 14 illustrates, in PDL form, the algorithm required to maintain balance tags in an HB(k) tree.

Restructuring

When a critically unbalanced node is encountered, that portion of the tree rooted at the critical node must be restructured or rotated so that the tree conforms to the given balance constraint. However, this restructuring must be done in a certain way in order to maintain the order associated with the nodes. Restructuring entails three steps:

1. Rearrange the nodes so that the subtree initially rooted at the critical node conforms to the balance constraint.

2. Reconnect any uninvolved descendants of the nodes directly involved that have been disconnected during the restructuring.

3. Modify the balance tags of the nodes involved to reflect their new positions. As during

backtracking, this may be done based on their previous values. (It is not intuitively obvious how this may be done during rotation. A demonstration of this fact may be found in Appendix A.)

```
BEGIN BTAG_MAINTENANCE;
DO WHILE (Btag(NODE) < balance constraint  OR
  insertion occurred in the shorter subtree);
  IF insertion occurred to the right of this NODE
    THEN Increment Btag(NODE) by 1;
      IF NODE is now balanced or still left heavy
        THEN END BTAG-MAINTENANCE;
      END IF;
    ELSE Decrement Btag(NODE) by 1;
      IF NODE is now balanced or still right heavy
        THEN END BTAG-MAINTENANCE;
      END IF;
  END IF;
  Back up to next previous NODE;
END DO;
Tree violates balance constraint at NODE;
END BTAG-MAINTENENCE;
```

Figure 14. Balance Tag Maintenance in an HB(k) Tree

At most three nodes along the search path are involved in this restructuring - the critical node, the immediate descendant of the critical node and the offspring of the immediate descendant (the grand-descendant) of the critical node.

Four cases may be identified in terms of the nodes involved as having differing restructuring requirements:

1. Critical node is left heavy, descendant of the critical node is left heavy.
2. Critical node is left heavy, descendant of the critical node is right heavy.
3. Critical node is right heavy, descendant of the critical node is right heavy.
4. Critical node is right heavy, descendant of the critical node is left heavy.

Case 3 is the mirror image of Case 1 (see Figure 15). Figure 16 illustrates Case 1, a simple rotation. Figure 17 illustrates, in PDL form, the balance tag maintenance requirements for the nodes involved in Case 1 or Case 3 restructuring. Case 4 is the mirror image of Case 2 (see Figure 18). Figure 19 illustrates Case 2 restructuring, a split rotation. Split rotation involves a subcase when dealing with balance tags. Figure 20 illustrates, in PDL form, the balance tag maintenance requirements for the nodes involved in Case 2 or Case 4 restructuring.

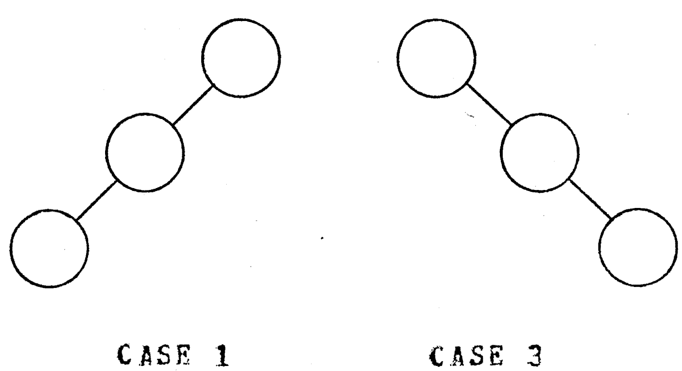


Figure 15. Cases 1 and 3 as Mirror Images

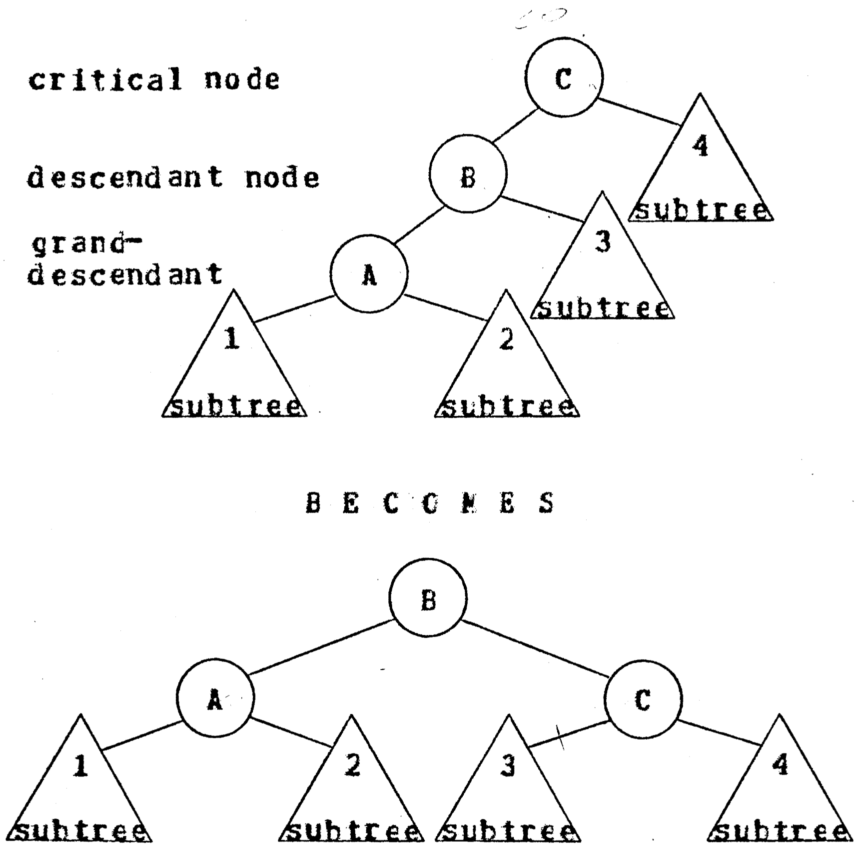


Figure 16. Simple Rotation in an HB(k) Tree

```
BEGIN SIMPLE-BTAG;
     /*comment:  let
                      CN represent the critical node
                      DCN represent the descendant */
IF insertion occurred right of CN
  THEN Btag(CN) <- balance constraint - Btag(DCN);
    Decrement Btag(DCN) by 1;
  ELSE Btag(CN) <-  -balance constraint - Btag(DCN);
    Increment Btag(DCN) by 1;
 END IF;
END SIMPLE-BTAG;
```

Figure 17. Balance Tag Maintenance After a Simple Rotation

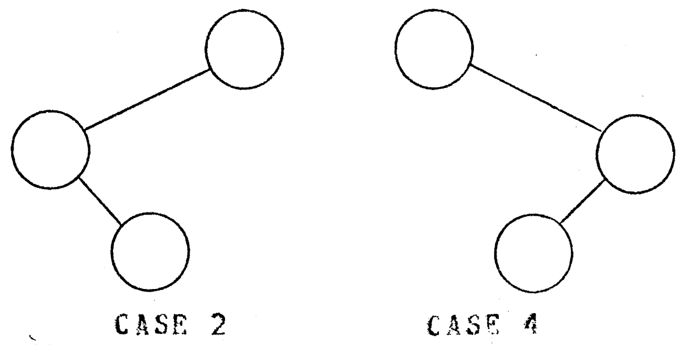


Figure 18. Cases 2 and 4 as Mirror Images

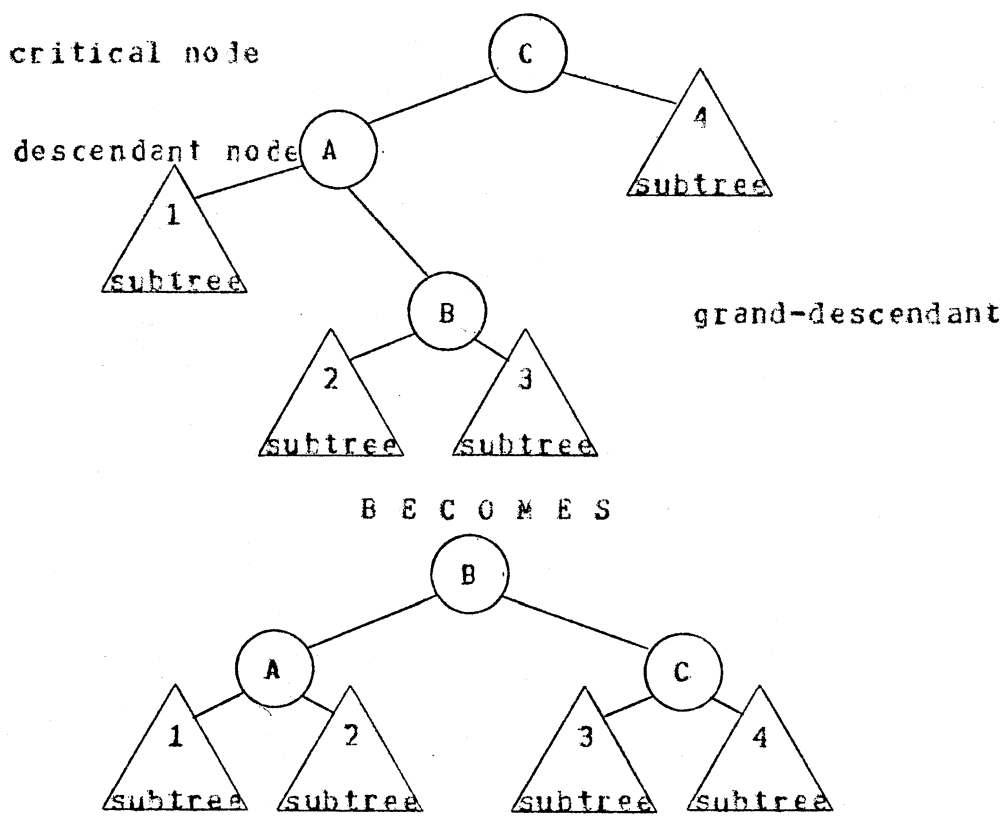


Figure 19. Split Rotation in an HB(k) Tree

It can be shown that this restructuring results in a (sub)tree of the same height as the (sub)tree before restructuring. (See Appendix B for a detailed presentation of this fact.) Thus, after restructuring, the insertion may be terminated.

```
BEGIN SPLIT_BTAG;
  /*comment - let
       CN represent the critical node
       DCN represent the descendant
       GDCN represent the grand-descendant
  */
SELECT;
  WHEN(insertion occurred right of both CN and GDCN):
    Btag(CN) <- balance constraint - 1 - Btag(gdcn);
    Increment Btag(DCN) by 1;
    Btag(GDCN) <- MIN (balance constraint - 1,
    Btag(GDCN) );
  WHEN(insertion occurred right of CN and left of GDCN):
    Save Btag(DCN);
    Btag(CN) <- balance constraint - 1;
    Btag(DCN) <- Btag(DCN) _ Btag(GDCN) + 1;
Btag(GDCN) <- MAX (Btag(GDCN),Saved Btag(DCN) + 1);
  WHEN(insertion occurred left of both CN and GDCN):
    BTag(CN) <- 1 - balance constraint - Btag(GDCN);
    Btag(GDCN) <- MAX (Btag(GDCN),1 - balance constraint);
    Decrement Btag(DCN) by 1;
  Otherwise: /*comment - insertion occurred left of CN
    and right of GDCN*/
    Save Btag(DCN);
    Btag(CN) <- 1 - balance constraint;
    Btag(DCN) <- Btag(DCN) - Btag(GDCN) _ 1;
    Btag(GDCN) <- MIN (Saved Btag(DCN) - 1,Btag(GDCN) );
END SELECT;
END SPLIT_BTAG;
```

Figure 20. Balance Tag Maintenance After a Split Rotation

Deletion in an HB(k) Tree

Deletion in an HB(k) tree is more complicated than insertion. Insertion always inserts a new node in an external node position and at most one rotation is required to bring the tree back into compliance with the balance constraint. Deletion removes an internal node which may have one or two descendant subtrees. These dangling subtrees must be reconnected to the tree in the proper manner to prevent violation of the balance constraint. This may involve multiple rotations as shall be shown.

Leaf vs. Non-leaf

Although once a node has been deleted, one must backtrack along the search path in order to maintain the balance tags, deletion presents differing initial problems depending on whether a leaf node (no descendants), a semi-leaf node (one descendant), or an interior node (two descendants) is being deleted. These differing requirements are outlined below:

1. If a leaf node is deleted, set its parent's pointer to NULL. Prepare to backtrack starting at the parent node.

2. If a semi-leaf node is deleted, set its parent's pointer to its non-null pointer. Prepare to backtrack starting at the parent node.

3. If an interior node is deleted, then do the following:

a. Find a node with which the node to be deleted may be replaced keeping track of the search path. This will be the node with the largest (smallest) key in the left (right) subtree. The usual approach is to select the longer subtree (the heavy side of the node to be deleted).

b. In effect, delete the replacement node from its present position. That is, delete the node but save the value of the key (and any information associated with the key).

c. Delete the intended node by substituting the replacement node. The balance tag of the deleted node becomes the balance tag of the replacement node.

d. Prepare to backtrack starting at the original parent of the replacement node.

Several different cases may arise during backtracking They are as follows:

1. The node was balanced before deletion. Adjust the balance tag to reflect in which subtree the deletion occurred (the opposite subtree is now longer by 1). Terminate the algorithm.

2. The node was left or right heavy before deletion; deletion occurred in the longer or heavy subtree. The heavy subtree is now less heavy (shorter) by one. The node becomes less unbalanced by 1. Continue backtracking.

3. The node was left or right heavy before deletion; deletion occurred in the shorter subtree. The node is now more unbalanced in the same direction as before (the shorter subtree has become one more level shorter than the longer subtree). Two subcases may be recognized:

a. The balance tag(node) was < balance constraint. The new balance tag remains <= balance constraint; hence, terminate the algorithm.

b. The balance tag(node) was = balance constraint. The node becomes critically unbalanced. The tree violates the balance constraint. Restructure the tree. After restructuring, continued backtracking may or may not be required.

When restructuring is required, the nodes involved are not along the search path except for the critical node itself. This is different from insertion but is as expected since the subtree containing the search path has been shortened in height to the point of causing the critical node to violate the balance constraint. Thus, the other subtree is the critically heavy one. With this difference in which node is meant by the immediate descendant of the critical node in mind, there are four cases for restructuring which correspond to those for insertion:

1. The critical node is left heavy; the descendant of the critical node is left heavy or balanced.

2. The critical node is left heavy; the descendant of the critical node is right heavy.

3. The critical node is right heavy; the descendant of the critical node is right heavy or balanced.

4. The critical node is right heavy; the descendant of the critical node is left heavy.

Note that the only difference between these cases and those for insertion is that the subtrees rooted at the descendant of the critical node may be balanced. This case may be rotated either way, simple (Cases 1 and 3) or split (Cases 2 and 4). It is placed with the simple rotation cases merely because these involve less work.

The rearrangement of the nodes is handled in identically the same way as for insertion with the exception of choosing the grand-descendant of the critical node during split rotations. In insertion, the grand-descendant is along the search path; in deletion, the grand-descendant is chosen from the heavy side of the descendant.

Balance tag maintenance is also similar to that done for insertion if one considers that inserting a new node in the right subtree of some existing node is akin to deleting a node from the left subtree. A difference arises because of the possibility that the descendant of the critical node may root balanced subtrees (balance tag = 0) before restructuring. In this case, only, backtracking may be terminated immediately since the rearrangement will result in a (sub)tree of exactly the same height as the subtree rooted at the critical node before deletion.

Performance of HB(k) Trees

The theoretical analysis that has been done for HB(k) trees has not been supported by empirical observation (7, 13). Some of the empirical results that have been reported are outlined below.

Foster (7) found that, for insertion, letting k be as large as four increased the average search path length by only one while the number of restructurings decreased by approximately 43% . Work reported by Van Doren (24) complemented Foster's findings for insertion and extended the results to deletion. The effect of a change in k under deletion follows a pattern similar to that for insertion: increasing k decreases the number of restructurings required. Van Doren also found that increasing k increases the number of nodes examined during the backtracking operation. This may offset the gain realized by fewer restructurings. Karlton, Fuller, Scroggs, and Kaehler (10) have provided the most complete set of empirical observations concerning the performance of height balanced trees. Part of their work substantiates the results reported by Foster and Van Doren. Other of their findings follow:

1. The average number of rotation seems to be independent of the number of nodes in the tree for trees containing more than 30 nodes.

2. The number of nodes visited during backtracking is independent of the the number of nodes for insertion but for deletion it increases slowly as the number of nodes increases.

3. The average number of nodes visited during backtracking is less for deletion than for insertion, for large k (balance constraint).

4. Deletion is more time consuming than insertion but search time is the dominant factor in both operations.

Experiments performed by Baer and Schwab (3) corroborate previously reported findings.

Alternatives to HB(k) Balanced Binary Search Trees

The work done on AVL and HB(k) trees has stimulated the development of alternative solutions to the problem of balancing a binary tree structure based on information about path lengths and heights of subtrees. Nievergelt and Reingold (19) introduced bounded balance or BB(a) trees where 'a' is a restriction on the relative number of nodes in the left and right subtrees of a node:

a <= (number of nodes in the left subtree + 1) /
(total number of nodes + 1) <= 1 - a.

BB(0) corresponds to an unconstrained binary search tree; BB(1/2) corresponds to a complete binary search tree. The authors admit that, based on empirical evidence, search time is somewhat worse for BB(a) trees than for HB(k) trees but they claim several advantages of BB(a) trees over HB(k) which may compensate for this:

1. Such important operations as finding the kth data element, or the qth quantile, or how many elements there are lexicographically between x and y, can all be done in time O(log(n)) (in a BB(a) tree), while they seem to require time O(n) (in an HB(k) tree), and

2. The smallest possible change in k (for HB(k) trees) changes the class of trees very drastically, and thus the compromise between search time and rebalancing time cannot be finely tuned (as it can be for BB(a) trees).

Work done by Van Doren and Gray (25) supports the stated disadvantage but no work has been reported to support the claimed advantages. More extensive research and analysis is required before the advantages and disadvantages can be fairly examined.

Pursuing an idea suggested by Knuth (13), Hirschberg (9) investigated one-sided height-balanced or OSHB trees which are a restricted subclass of AVL trees. OSHB trees require that the right subtree never has a smaller height than the left subtree. In other words, the nodes may be balanced or right heavy only. Although fast search time is maintained, insertion requires time O(log(n)**2) in an OSHB tree. Later work by Zweben and McDonald (27) shows that deletion of an arbitrary node may be done in time O(log(n)). OSHB trees saves one bit of storage per node when compared to the AVL trees introduced in 1962, but the trade off required for insertion may not be worth the storage saved. Hirschberg and Zweben and McDonald leave open the question of the actual (empirical) behavior of OSHB trees.

Drawing on the work with OSHB trees, Ottmann, Six, and Wood (22) developed right brother or RB trees. The authors indicate that RB trees are a subclass of brother trees which they had presented earlier. A brother tree requires that all leaf nodes be on the same level and that each node with only one descendant has a sibling (brother) with two descendants. Right brother trees qualify the latter condition, requiring that each node with only one descendant must have a right sibling (brother) with two descendants. Ottmann, Six, and Wood detail insertion and deletion requirements and theoretically prove that both insertion and deletion may be accomplished in O(log(n)) time although the algorithm for insertion is more complex. They also derive bounds for the height of the tree:

CEIL(lg(n)) <= height < 1.44 - lg(n + 1) - 0.32.

Empirical verification of these claims is lacking.

Another development in balanced trees is Power k or Pk trees introduced by Luccio and Pagli (16). Power trees maintain balance information as for AVL trees but only for the set of nodes on selected paths from the root to the leaves identified through the parameter k. The paths are identified as follows:

1. For k = 0, there exists at least one path = the height of the tree such that all nodes on the path satisfy

   | balance tag(node) | <= 1,

   and

2. For k > 0, all paths of length j where

   height of the tree - k + 1 <= j <= height of the tree

   are such that all nodes on each path satisfy

   | balance tag (node) | <= 1.

In other words, balance is maintained only for those nodes which lie along a path originating from the root of the entire tree which has reached a specified level relative to the height of the tree. Since the height of a tree is a dynamic quantity, the set of nodes for which the balance is maintained is also dynamic. Theoretical determination of the following quantities are obtained for P0 trees under insertion only:

1. Worst case path length = SQRT (2*n), and
2. Average search length for a worst case tree = 2/3 (SQRT(2*n)).

As for AVL and HB(k) trees, average search length for a Pk tree, assuming all key sequences equally likely, has yet to be successfully analyzed. Empirical results show that P0 trees approximate the behavior of AVL trees but drastically reduce the amount of restructuring required. The difficult question of deletion in a Pk tree is left open.

It is somewhat difficult to compare these alternatives since all of them lack a definitive analysis of their average behavior just as HB(k) trees do. As a result of this lack, it is difficult to compare the advantages and disadvantages between the classes of height balanced trees since there is no evident relationship between the constraining parameters. However, a generalization of HB(k) trees may provide the impetus for a rigorous analysis of HB(k) trees.

CHAPTER IV

PARTIALLY HEIGHT BALANCED TREES

A generalization of HB(k) trees, partially height balanced (PHB) trees, may provide empirical guidance to the development of a rigorous theoretical analysis of the behavior of HB(k) trees (23). PHB trees maintain the height balance criteria of HB trees but restrict the effect of the criteria to internal nodes within a specified path length to an external node. The notation used is PHB(k1,k2) where k1 is the height balance constraint and k2 is the path length constraint, the path length to an external node within which a given internal node must lie if the height balance constraint is to apply.

To illustrate the effect of k2 on HB trees, consider the HB(1) tree in Figure 21. This may also be classified as a PHB(1,1) tree. Assume that key A is inserted into this tree. If classified as an HB(1) tree, then Figure 22 (a) would be the result; but if classified as a PHB(1,1) tree, then Figure 22 (b) would be the result.

One can express an HB tree via a PHB tree in the following manner:

HB(k) = PHB(k,i)

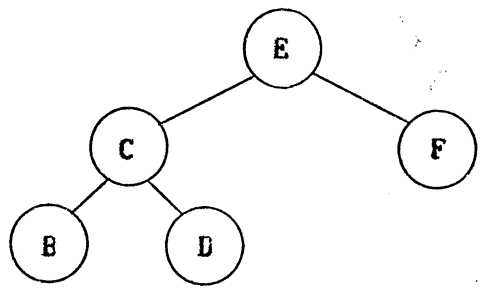
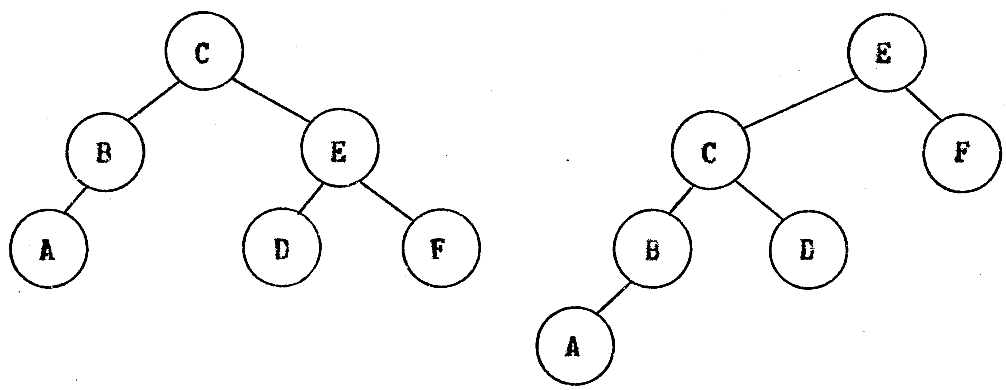


Figure 21. An HB(1) Tree



(a) (b)

Figure 22. Result of Insertion Depends on Classification

where 'i' stands for infinity. The PHB balance constraint is applied to all internal nodes within an infinite path length of an external node which is all internal nodes. Similarly, an unconstrained binary search tree is equivalent to a PHB(i,k).

Structure and Maintenance of PHB(k1,k2) Trees

Nodes of a PHB tree must contain the information required for nodes in an HB tree. In addition, in order to be able to maintain a PHB tree, one must know the minimum length to an external node of every internal node in the tree. Hence, the node structure must contain this information.

The question to be answered is how to maintain the path length to an external node. It should be apparent that the minimum path length to an external node is dependent on the minimum path lengths to external nodes of its two immediate descendants. If we define the path length to an external node from an external node to be 0, then this dependency can be expressed as:

mpl(node) = MIN (mpl(left descendant),mpl(right descendant)) + 1

for any internal node (mpl stands for minimum path length to an external node).

Algorithms for PHB(k1,k2) Trees

The search algorithm is identical to that for HB trees. The differences in the insertion and deletion algorithms arise in answering the question 'is this tree critically unbalanced' but not in the placement or removal of a node.

In order to determine if the tree is critically unbalanced, one must first maintain the balance tags associated with each node in the PHB tree as for those in an HB tree. At the same time, one must maintain the mpl's for each node. This must be done through the dependency expressed above between one node's mpl and its immediate descendants' mpl's, since it does not appear that there is a relationship between a node's mpl before insertion/deletion and after as there is for a node's balance tag.

As to whether or not the insertion/deletion resulted in a critically unbalanced condition, in PHB trees, the balance tag associated with any node may violate the balance constraint but the distance to an external node may exceed that specified by the path length constraint thus obviating restructuring. Hence, before a PHB tree is declared to be out of balance, the critical node must meet the following criteria:

1. Balance tag(node) > balance constraint.
2. Mpl(node) <= path length constraint.

For balance tag maintenance in HB trees, it is not necessary to backtrack past the critical node. However, for PHB trees it appears that backtracking must continue until the balance tag indicates that the height of the subtree rooted at a node has not changed. Minimum path length to an external node would also require backtracking past the critical node since the mpl determines which nodes are eligible for restructuring. Consider the PHB(2,2) tree of Figure 23 which depicts the state of the tree just after insertion of node B and balance tag maintenance to node D (the critical node).

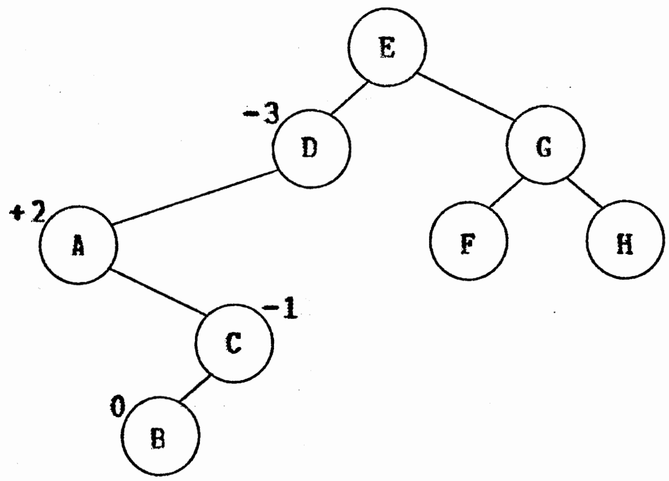


Figure 23. A PHB(2,2) Tree After Insertion of a Node

The balance tag of node D violates the balance constraint and its mpl is less than the path length constraint. Hence, the tree must be restructured. Figure 24 depicts the tree after restructuring. Note that the balance tags for node E, one level back from node D, the critical node, remains unchanged; however, node E's mpl has changed from 2 to 3. Whereas before insertion of node B, node E's mpl would have permitted its participation in restructuring if required, after insertion of node B, node E's mpl obviates its involvement in restructuring.

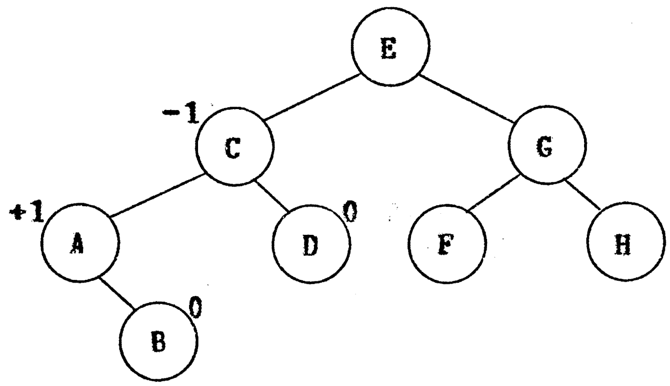


Figure 24. The PHB(2,2) Tree After Restructuring

By extension of this example, it should be evident that it is necessary to backtrack along the search path for insertion/ deletion past the critical node in order to maintain the structural information associated with each node. Thus, for PHB trees, backtracking involves maintenance of two quantities which have different requirements for terminating their maintenance. Balance tag maintenance may be terminated under the same conditions as for HB trees. Minimum path length maintenance continues until a node is encountered whose mpl does not change during maintenance. If one node's mpl does not change then its parent's mpl also will not change.

PHB(1,1) Trees

Of particular and additional interest is the subclass of PHB trees known as PHB(1,1) trees. The reasons for this interest are (23) :

1. Maintenance of PHB(1,1) trees does not require the generalized massively detailed algorithms of PHB(k1,k2) trees. The insertion algorithm in particular is much simpler since:

   a. Restructuring does not require dangling subtree considerations.

   b. Balance tags need not be maintained since balance may be easily computed as a function of insertion searching.

2. Its worst case (see Figure 25) is not as bad as an unconstrained binary search tree.

3. For moderately sized, randomly constructed trees, the expected search performance for PHB(1,1) trees is only slightly worse than HB(1) trees.

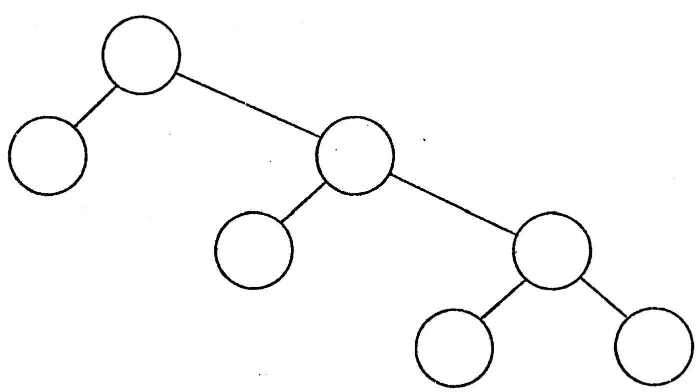


Figure 25. A Worst Case for PHB(1,1) Trees

Algorithms for PHB(1,1) Trees

The PHB(1,1) insertion algorithm is straightforward and is given in Figure 26. The information required from searching the tree for the key is given in the argument list to SEARCH; SEARCH itself is not shown. Deletion presents a more complex problem. Without balance information, it is difficult to determine how to restructure an unbalanced tree or how many restructurings are required. Consider deleting the key I from the PHB(1,1) tree of Figure 27 (a). Proceeding as for deletion in other binary search trees, one replaces I with H; Figure 27 (b) is the result. The subtrees of node G now violate the balance constraint. This could be easily determined by 'looking ahead' one level: if the non-null descendant has a descendant, then the (sub)tree is out of balance. However, how does one decide how to restructure the tree? Should a simple or split rotation be performed? A simple solution is to do a simple rotation then 'look ahead' one level to determine if the new subtree rooted at the critical node is unbalanced; if so, then do a simple rotation; then 'look ahead' . . . and so on, until the subtree is not critically unbalanced.

A much cleaner solution to the problem of deletion follows:

1. Delete the desired key by replacing it with the largest key in the left subtree.

2. If the original PARENT of the replacement node now has two NULL links, then terminate the algorithm; otherwise,

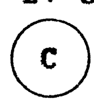
3. Remove the PARENT of the replacement node by replacing its parent's pointer with its non-null descendant.

4. Reinsert the PARENT in the subtree rooted at the descendant. This permits the insertion algorithm to restructure the tree where appropriate.

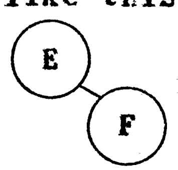
```
BEGIN INSERT (desired key);
CALL SEARCH (desired key,NODE,PARENT,GRANDPARENT,
  GREAT_GRANDPARENT);
IF desired key < key(PARENT)
  THEN attach desired key to LEFT(PARENT);
  ELSE attach desired key to RIGHT(PARENT);
END IF;
IF GRANDPARENT is NULL  OR  (GRANDPARENT
  is not NULL AND does not have a NULL link)
  THEN END INSERT;
END IF;
IF PARENT and GRANDPARENT have a NULL link
  on the same side
  THEN Perform a simple rotation;
  ELSE Perform a split rotation;
END IF;
END INSERT;
```

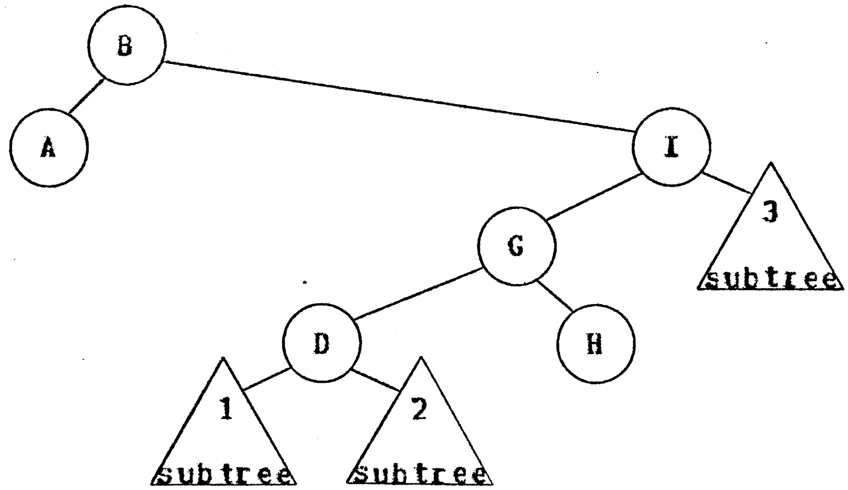
Figure 26. Insertion Algorithm for FHB(1,1) Trees

Let us assume that in Figure 27 subtree 1 looks like this:

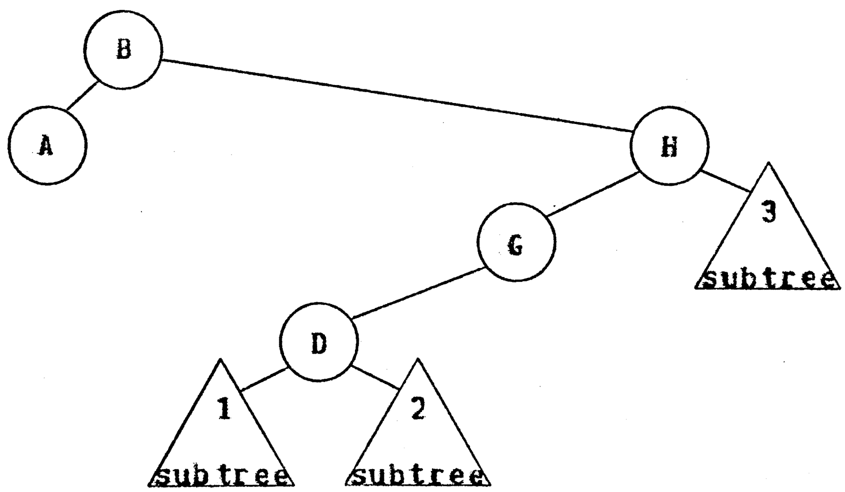


and that subtree 2 looks like this:





(a)



(b)

Figure 27. Deletion in a PHB(1,1) Tree

The result of this solution applied to Figure 27 is pictured in Figure 28. Occasionally, this approach restructures the tree (reinserts a node) unnecessarily; however, in order to prevent this, a one-level look ahead must be done. This would be extra work for those cases in which reinsertion must occur. Intuitively, it seems that an occasional unnecessary reinsertion of a node creates less extra work.

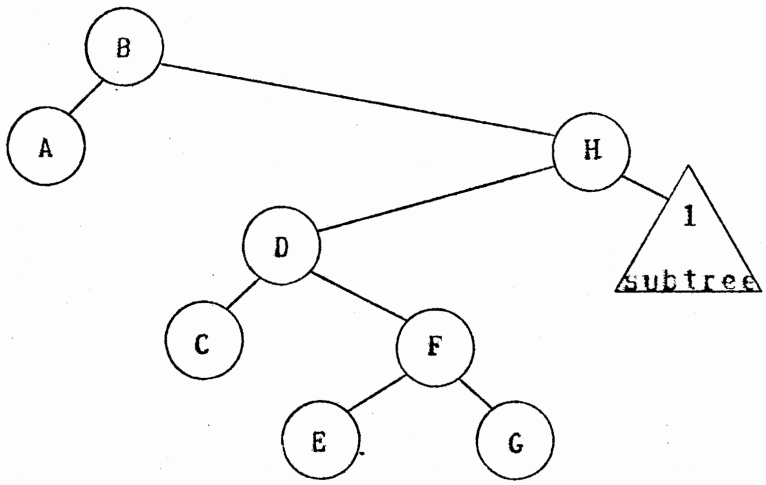


Figure 28. Restructuring of a PHB(1,1) Tree After Deletion

Performance of PHB(k1,k2) Trees

A formal theoretical analysis of the performance of PHB trees has been presented only for PHB(1,1) trees (23). Unfortunately, preliminary empirical results did not support the analysis. Use of a research tool to provide empirical data regarding height balanced trees may guide further development in this area.

CHAPTER V

A RESEARCH TOOL FOR HEIGHT BALANCED TREES

Basically, the research tool is a set of algorithms designed to build various height balanced trees with exactly the same keys and then give performance measures, such as internal and external path lengths, and number of restructurings required, so that the relative merit of each class of binary search trees may be compared. Such empirical data, gathered in an orderly fashion, may also guide the theoretical analysis of the behavior of the trees. At the present time, insertion and deletion algorithms have been implemented for the general classes HB(k) and PHB(k1,k2), and for the specific trees AVL and PHB(1,1), and the unconstrained binary search tree. It is intended that the programs be capable of being expanded and developed into an ongoing project with algorithms for other classes of height balanced trees being implemented. The programs are written in the PL/I programming language. A copy may be obtained through the Computer Science Department of Oklahoma State University.

Logic Design

A research tool must encourage its effective use by persons other than those who originally designed it. In accordance with this , the following points were considered in designing the driving program and its input requirements:

1. The explanation of how the input should be prepared should not require sections of the driving as documentation.

2. Preparing the input should not require an intimate knowledge of input list formats used in the driving program.

3. Input should be free-form (no column alignment requirements) to avoid errors that fixed-form may create.

4. Defaults on certain parameters should be allowed.

5. Expansion of input capabilities or a change in how something is specified should be easy to implement within the driving program.

For these reasons , the author chose to design and implement a small command language for use with the research tool. A signal character is used to signal that a keyword is to follow; therefore, no column requirements must be enforced. A complete Backus-Naur Form (BNF) description of the language may be found in Appendix C.

The language is interpreted via a top-down parser in sections. Each major section is terminated by the keyword GO which indicates that all information necessary to do some work with the trees (insertion and/or deletion) has been interpreted and may be used at this point. The top-down parser also allows greater ease of future modifications of input capabilities since each syntactic category may be implemented as a separate module.

Using the Research Tool

One may use the research tool to insert, or delete, or alternately insert and delete keys from any number of the available trees. The keys used in these operations may be ordered, random, or alternating. Alternating key sequences exercise both simple and split rotation capabilities and create degenerate unconstrained binary search trees and worst case PHB(1,1) trees. After each insertion and/or deletion sequence various performance measures may be taken.

The most appropriate way to introduce how to use the research tool is to illustrate the capabilities of the command language with a detailed example. Appendix D provides such an illustration.

Application of the Research Tool

PHB(1,1) trees may become an interesting structure in and of themselves. The reasons for this expectation are:

1. No extra storage for balance tags is required since, for insertion, balance may be computed as a function of searching and, for deletion, balance may be regained by reinserting the critical node.

2. The worst case for PHB(1,1) is not as bad as for unconstrained binary search trees.

3. For randomly constructed trees, the expected average search path length is no worse than for AVL or HB(1) trees.

Evidence for the first claim is presented above in the discussion of PHB trees. The initial application of the research programs was to provide empirical data concerning the last two claims. Three test cases were involved. Table I shows the information used in each test case. BST stands for unconstrained binary search tree.

TABLE I

TEST CASE INFORMATION

TEST CASE	DESCRIPTION	TREES USED Type	TREES USED Number	NUMBER of KEYS
1	FUNCTION: insert KEY SEQUENCE: alternating	BST HB(1) PHB(1,1)	1 each	100 each tree
2	FUNCTION: insert KEY SEQUENCE: permutations of a given sequence	BST HB(1) PHB(1,1)	10 each	100 each tree
		BST HB(1) PHB(1,1)	10 each	200 each tree
3	FUNCTION: alternate insert/delete KEY SEQUENCE: random	trees from TEST CASE 2		20 in trees of size 100 40 in trees of size 200

Test Case 1 demonstrates what will happen if the keys are inserted in such a manner as to create a degenerate unconstrained binary search tree. Although deducible without empirical testing, use of the research tool makes the results readily available. As can be seen in Table II, a PHB(1,1) tree is not as bad as an unconstrained tree; the average search path length and the average insertion/deletion search path length are about half those of an unconstrained binary search tree. However, they are more than three times those of an HB(1) tree. Of course, about half as many restructurings were required for the PHB(1,1) tree compared to the HB(1) tree, but this is not an intuitively reasonable tradeoff.

TABLE II

RESULTS FROM TEST CASE 1

TREE	n	C(n)	C'(n)	ROTATIONS
BST	100	50.5	51.0	-
HB(1)	100	6.0	7.0	92
PHB(1,1)	100	26.0	26.7	49

Test Case 2 demonstrates average behavior under the assumption that each permutation of a given key sequence is equally likely to occur. Table III shows the results garnered from Test Case 2. The values shown are the averages across all trees of the same type. Certainly the behavior of PHB(1,1) trees tends toward that of the HB(1) trees but it is slightly worse. However, note again that about half as many restructurings were required in order to maintain the trees. Whether this drastically reduced amount of restructuring is worth the small trade off in search time remains to be determined.

TABLE III

RESULTS FROM TEST CASE 2

TREE	n	C(n)	C'(n)	ROTATIONS
BST	100	7.5	8.4	-
	200	9.1	10.0	-
HB(1)	100	5.9	6.8	45.1
	200	6.9	7.8	87.7
PHB(1,1)	100	6.4	7.3	27.6
	200	7.7	8.6	55.4

Test Case 3 demonstrates the trend for average behavior after a period of activity within the tree. The data are presented in Table IV. As for Test Case 2, the values shown are the averages across all trees of the same type. About three-fourths as many rotations were required to maintain PHB(1,1) trees as opposed to HB(1) trees. This is a higher percentage than for insertion alone and probably reflects the occasional unnecessary reinsertion of a node. However, the number of rotations is still less and the average search and insertion/deletion path lengths are less than 1 greater for PHB(1,1) trees than for HB(1) trees. This indicates an advantage for PHB(1,1) trees. No extra storage is required for balance information, yet fewer rotations are required to maintain the tree and the average path lengths are not much longer. The exact extent of this trade off remains to be determined.

TABLE IV

RESULTS FROM TEST CASE 3

TREE	n	C(n)	C'(n)	ROTATIONS
BST	100	7.2	8.1	-
	200	8.9	9.8	-
HB(1)	100	5.9	6.8	52.7
	200	6.9	7.9	103.6
PHB(1,1)	100	6.5	7.4	37.2
	200	7.8	8.7	74.5

CHAPTER VI

SUMMARY AND CONCLUSIONS

This study has dealt with the evolution of height balanced binary search trees and with the design and implementation of a research tool to provide the impetus for rigorously analyzing their performance characteristics. Height balanced binary search trees are one solution to the problem one often encounters in information storage: how can one store information so that insertion, deletion, and searching can be accomplished quickly and efficiently? Generalized height balanced trees can guarantee logarithmic search time; however, since balance information must be maintained, and insertion and deletion involve backtracking along the search path, it is unclear how to decide what an optimal trade off between search time and maintenance time is. A specific subclass of height balanced trees, PHB(1,1) trees, has been introduced which do not require maintenance of balance tags nor backtracking but may still be able to provide close to logarithmic search time for the average case.

Results of the Study and Suggestions for Future Study

This study presents previously undocumented outlines of algorithms for a generalized class of height balanced trees, partially height balanced or PHB trees. One element of these algorithms remains unclear - is it necessary to maintain balance tag and path length information past the critical point of 1 greater than the constraint values? If one, instead, maintained them only until they reached these points during insertion, then could the appropriate values be regained during deletion such that a node would be recognized as once again eligible for restructuring? This question needs further study.

Also presented were algorithms for the subclass, PHB(1,1) trees. The deletion algorithm was previously undocumented. The algorithms presented have been implemented as part of a research tool for height balanced trees.

An initial application of the research tool was made for PHB(1,1) trees. Although PHB(1,1) trees exhibit slightly worse performance characteristics than do HB(1) trees, they also reduce by half the number of restructurings required. This seems to indicate that PHB(1,1) trees may be a viable alternative to HB(k) trees. However, more extensive analysis, empirical and theoretical, needs to be done. The research tool is also available to provide the empirical impetus to analyzing HB(k) trees. As its capabilities expand, comparisons with other height balanced trees shoul be made to weigh the relative advantages and disadvantages of each under particular circumstances.

Expanding the Capabilities of the Research Tool

In order to provide a more flexible command language, it is desirable to permit default values for more of the parameters such as FROM x TO y. The parser is designed in a modular fashion to facilitate this expansion. Most of the syntactic categories correspond to separate modules in the implementation. Hence, modifying at most one module per expansion is necessary.

It would be desirable at some time to implement algorithms for other classes of height balanced trees, such as the BB(a) or Pk classes described above, in order to facilitate comparisons between data structures. It is also suggested that knowing the number of nodes accessed during backtracking and maintenance of balance information may help evaluate the trade off between search time and maintenance time for height balanced trees. The research tool should prove a powerful aid in the study of height balanced trees.

A SELECTED BIBLIOGRAPHY

(1) Adel'son-Vel'skii, G. M. and Y. M. Landis. "An Algorithm for the Organisation of Information." Soviet Math, Vol. 6 (1963), 1259-1263.

(2) Baer, J-L. "Weight Balanced Trees." Proceedings AFIPS 1975 NCC, Vol. 44. Montvale, New Jersey: AFIPS Press, 1975.

(3) Baer, J-L., and B. Schwab. "A Comparison of Tree-Balancing Algorithms." Communications of the ACM, Vol. 20, No. 5 (May 1977), 322-330.

(4) Booth, A. D. and A. J. T. Colin. "On the Efficiency of a New Method of Dictionary Construction." Information and Control, Vol. 3, No. 4 (December 1960), 327-334.

(5) Burge, W. H. "Sorting, Trees and Measures of Order." Information and Control, Vol. 1, No. 3 (1958), 181-197.

(6) Clampett, H. A. "Randomized Binary Searching with Tree Structures." Communications of the ACM, Vol. 7, No. 3 (March 1964), 163-165.

(7) Foster, C. C. "A Generalization of AVL Trees." Communications of the ACM, Vol. 16, No. 8 (August 1973), 513-517.

(8) Hibbard, T. N. "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting." Journal of the ACM, Vol. 9, No. 1 (January 1962), 13-28.

(9) Hirschberg, D. S. "An Insertion Technique for One-sided Height Balanced Trees." Communications of the ACM, Vol. 19, No. 8 (August 1976), 471-473.

(10) Karlton, P. L., S. H. Fuller, R. E. Scroggs, and E. B. Kaehler. "Performance of Height Balanced Trees." Communications of the ACM, Vol. 19, No. 1 (January 1976), 23-28.

(11) Knuth, D. E. "Algorithms." Scientific American, Vol. 236, No. 4 (April 1977), 63-81.

(12) Knuth, D. E. The Art of Computer Programming, Vol. 1. Reading, Massachusetts: Addison-Wesley Pub. Co., 1973.

(13) Knuth, D. E. The Art of Computer Programming, Vol. 3. Reading, Massachusetts: Addison-Wesley Pub. Co., 1973.

(14) Kosaraju, S. R. "Insertions and Deletions in One-sided Height Balanced Trees." Communications of the ACM, Vol. 21, No. 3 (March 1978), 226-227.

(15) Luccio, F. and L. Pagli. "On the Height of Height-Balanced Trees." IEEE Transactions on Computers, Vol. C-25, No. 1 (January 1976), 87-90.

(16) Luccio, F. and L. Pagli. "Power Trees." Communications of the ACM, Vol. 21, No. 11 (November 1978), 941-947.

(17) Luccio, F. and L. Pagli. "Rebalancing Height Balanced Trees." IEEE Transactions on Computers, Vol. C-27, No. 5 (May 1978), 386-396.

(18) Nievergelt, J. "Binary Search Trees and File Organization." Computing Surveys, Vol. 6, No. 3 (September 1974), 195-207.

(19) Nievergelt, J. and E. M. Reingold. "Binary Search Trees of Bounded Balance." SIAM Journal of Computing, Vol. 2, No. 1 (March 1973), 33-43.

(20) Nievergelt, J. and C. K. Wong. "On Binary Search Trees." Proceedings of IFIP Congress 71, 91-98.

(21) Nievergelt, J. and C. K. Wong. "Upper Bounds for the Total Path Length of Binary Trees." Journal of the ACM, Vol. 20, No. 1 (January 1973), 1-6.

(22) Ottmann, T., H. W. Six, and D. Wood. "Right Brother Trees." Communications of the ACM, Vol. 21, No. 9 (September 1978), 769-776.

(23) Van Doren, J. R. Data and Storage Structures, (unpublished class notes). Stillwater, OK: Oklahoma State University, 1978.

(24) Van Doren, J. R. "Some Empirical Results on Generalized AVL Trees." Proceedings of the NSF-CBMS Regional Research Conference on Automatic Information Organizations and Retrieval, 1973, 46-62.

(25) Van Doren, J. R. and J. L. Gray. "An Algorithm for Maintaining Dynamic AVL Trees." in *Information Systems*, J. T. Tou, ed. New York, New York: Plenum Press, 1974, 161-180.

(26) Windley, P. F. "Trees, Forests and Rearranging." *Computer Journal*, Vol. 3, No. 2 (1960), 84-88.

(27) Zweben, S. H. and M. A. McDonald. "An Optimal Method for Deletion in One-sided Height Balanced Trees." *Communications of the ACM*, Vol. 21, No. 6 (June 1978), 441-445.

APPENDIX A

DERIVATION OF BALANCE TAG MAINTENANCE EQUATIONS

Maintenance of balance tags in an HB(k) or PHB(k1,k2) tree during rotation after insertion may be accomplished with the information provided by the previous values of the balance tags of the nodes involved. This is a demonstration of why it is possible. Similar results may be derived for deletion cases.

SYMBOL LEGEND:

CN : the critical node

DCN : the descendant of the critical node

GDCN : the grand-descendant of the critical node

h(#) : the height of the subtree indicated by #

h(null subtree) = 0

h(x) = MAX (h(LEFT(x)),h(RIGHT(x))) + 1

h(n) : the height of the (sub)tree rooted at node n

Bh(n) : h(n) BEFORE restructuring

Ah(n) : h(n) AFTER restructuring

b(n) : the balance associated with node n

b(n) = h(RIGHT(n)) - h(LEFT(n))

Bb(n) : b(n) BEFORE restructuring

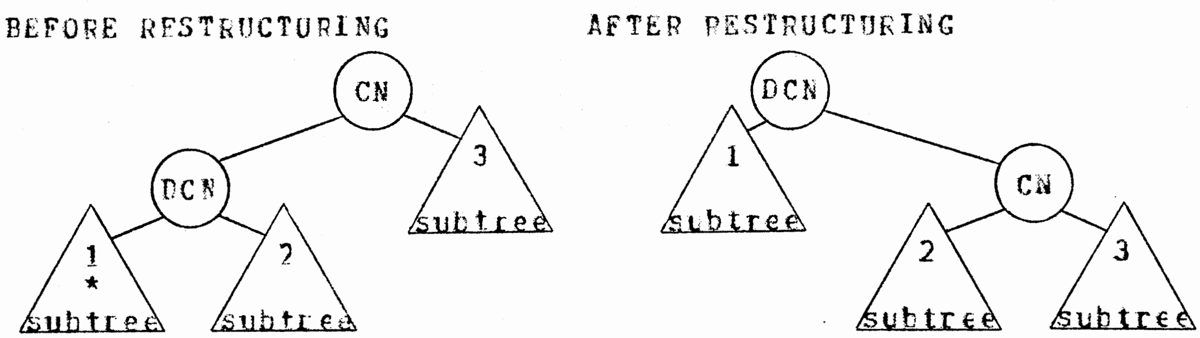
Ab(n) : b(n) AFTER restructuring

k : the balance constraint

* : insertion occurred in this subtree

CASE 1: SIMPLE ROTATION

CN left heavy; DCN left heavy.



BEFORE restructuring, we know that:

$$h(DCN) = h(1) + 1$$

$$h(CN) = h(DCN) + 1$$

$$= h(1) + 2$$

and

$$b(CN) = h(3) - h(DCN)$$

$$= h(3) - (h(1) + 1)$$

$$= -(k + 1)$$

$$b(DCN) = h(2) - h(1)$$

AFTER restructuring, we know that:

$$h(CN) = MAX\ (h(2), h(3)) + 1$$

$$h(DCN) = MAX\ (h(1), h(CN)) + 1$$

and

$$b(CN) = h(3) - h(2)$$

$$b(DCN) = h(CN) - h(1)$$
$$= MAX\ (h(2),h(3)) + 1 - h(1).$$

Since h(1), h(2), h(3), and k are constants through the rotation, the following expressions remain true through the rotation:

$$h(3) - (h(1) + 1) = -(k + 1)$$
$$h(3) = h(1) + 1 - k - 1$$
$$= h(1) - k$$

and

$$h(2) - h(1) = Bb(DCN)$$
$$h(2) = h(1) + Bb(DCN)$$

Substituting these expressions for h(2) and h(3) in the equations for AFTER b(n)'s gives:

$$b(CN) = h(3) - h(2)$$
$$= h(1) - k - (h(1) + Bb(DCN))$$
$$= -k - Bb(DCN)$$
$$b(DCN) = MAX\ (h(2),h(3)) + 1 - h(1)$$
$$= MAX\ (h(1)+Bb(DCN),h(1)-k) + 1 - h(1)$$
$$= MAX\ (Bb(DCN),-k) + 1$$

Q.E.D.

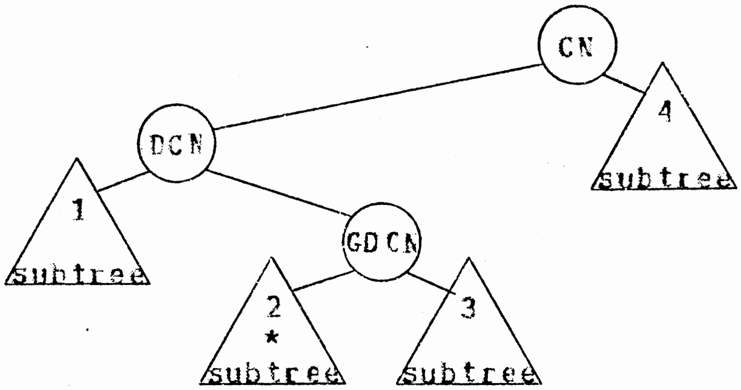
The expression for Ab(DCN) may be simplified further by noting that Bb(DCN) must be >= -k. Therefore, MAX (Bb(DCN),-k) will always yield Bb(DCN) and Ab(DCN) = Bb(DCN) + 1.

<u>CASE 2</u>: SPLIT ROTATION

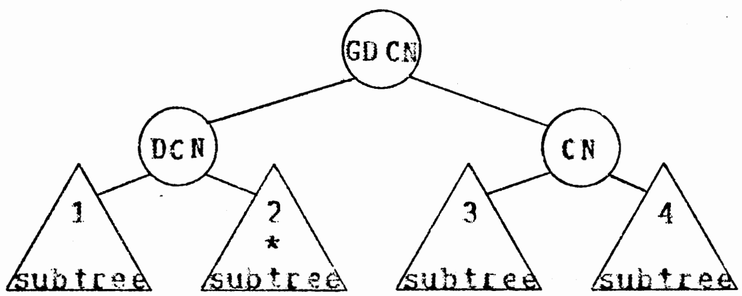
CN left heavy; DCN right heavy.

<u>Subcase a</u>: GDCN left heavy.

BEFORE RESTRUCTURING



AFTER RESTRUCTURING



BEFORE restructuring, we know that:

$$h(GDCN) = h(2) + 1$$

$$h(DCN) = h(GDCN) + 1$$

$$= h(2) + 2$$

$$h(CN) = h(DCN) + 1$$

$$= h(2) + 3$$

and

$$b(GDCN) = h(3) - h(2)$$

$$>= -(k - 1)$$

$$b(DCN) = h(GDCN) - h(1)$$

$$= h(2) - h(1) + 1$$

$$\leq k$$

$$b(CN) = h(4) - h(DCN)$$

$$= h(4) - h(2) - 2$$

$$= -(k + 1)$$

AFTER restructuring, we know that:

$$h(DCN) = MAX\ (h(1),h(2)) + 1$$

$$h(CN) = MAX\ (h(3),h(4)) + 1$$

$$h(GDCN) = MAX\ (h(DCN),h(CN)) + 1$$

and

$$b(DCN) = h(2) - h(1)$$

$$b(CN) = h(4) - h(3)$$

$$b(GDCN) = h(CN) - h(DCN)$$

$$= MAX\ (h(3),h(4)) - MAX\ (h(1),h(2))$$

Since h(1), h(2), h(3), h(4), and k are constants through the rotation, the following expressions remain true through the rotation:

$$Bb(GDCN) = h(3) - h(2)$$

$$h(3) = Bb(GDCN) + h(2)$$

$$Bb(DCN) = h(2) - h(1) + 1$$

$$h(1) = h(2) - Bb(DCN) + 1$$

$$-(k + 1) = h(4) - h(2) - 2$$

$$h(4) = h(2) - k + 1$$

Substituting these expressions for h(1), h(3), and h(4) in the equations for AFTER b(n)'s gives:

$$b(DCN) = h(2) - h(1)$$

$$= h(2) - (h(2) - Bb(DCN) + 1)$$

$$= Bb(DCN) - 1$$

$$b(CN) = h(4) - h(3)$$

$$= h(2) - k + 1 - (Bb(GDCN) + h(2))$$

$$= -k + 1 - Bb(GDCN)$$

$$b(GDCN) = MAX\ (h(3),h(4)) - MAX\ (h(1),h(2))$$

$$= MAX\ (h(3),h(4)) - MAX\ (h(2)-Bb(DCN)+1,h(2))$$

but since DCN was right heavy, Bb(DCN) > 0; hence,

$$= MAX\ (h(3),h(4)) - h(2)$$

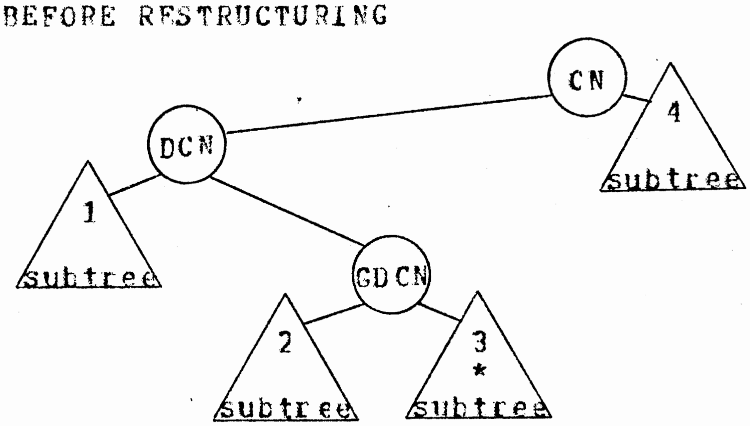
$$= MAX\ (Bb(GDCN)+h(2),h(2)-k+1) - h(2)$$

$$= MAX\ (Bb(GDCN),-k+1)$$

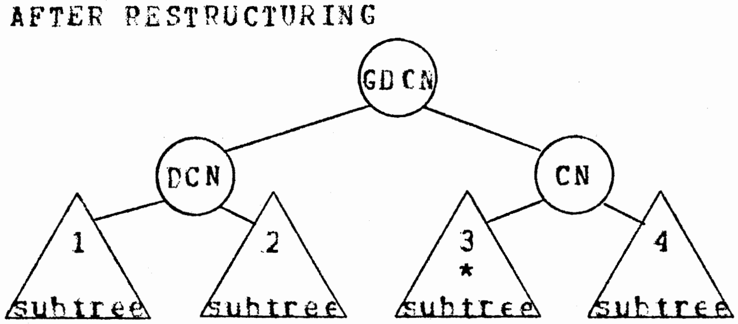
Q.E.D.

Subcase b: GDCN right heavy.

BEFORE RESTRUCTURING



AFTER RESTRUCTURING



BEFORE restructuring, we know that:

$$h(GDCN) = h(3) + 1$$

$$h(DCN) = h(GDCN) + 1$$
$$= h(3) + 2$$

$$h(CN) = h(DCN) + 1$$
$$= h(3) + 3$$

and

$$b(GDCN) = h(3) - h(2)$$
$$<= k - 1$$

$$b(DCN) = h(GDCN) - h(1)$$
$$= h(3) + 1 - h(1)$$
$$<= k$$

$$b(CN) = h(4) - h(DCN)$$
$$= h(4) - h(3) - 2$$
$$= -(k + 1)$$

AFTER restructuring, we know that:

$$h(DCN) = MAX\ (h(1),h(2)) + 1$$

$$h(CN) = MAX\ (h(3),h(4)) + 1$$

$$h(GDCN) = MAX\ (h(CN),h(DCN)) + 1$$

and

$$b(DCN) = h(2) - h(1)$$

$$b(CN) = h(4) - h(3)$$

$$b(GDCN) = h(CN) - h(DCN)$$
$$= MAX\ (h(3),h(4)) - MAX\ (h(1),h(2))$$

Since h(1), h(2), h(3), h(4), and k remain constant through the rotation, the following expressions remain true through the rotation:

$Bb(GDCN) = h(3) - h(2)$

$h(2) = h(3) - Bb(GDCN)$

$Bb(DCN) = h(3) + 1 - h(1)$

$h(1) = h(3) + 1 - Bb(DCN)$

$h(4) - h(3) - 1 = -(k + 1)$

$h(4) = h(3) + 1 - k - 2$

$= h(3) - k - 1$

Substituting these expressions for $h(1)$, $h(2)$, and $h(4)$ in the equations for AFTER b(n)'s gives:

$b(DCN) = h(2) - h(1)$

$= h(3) - Bb(GDCN) - (h(3) + 1 - Bb(DCN))$

$= Bb(DCN) - Bb(GDCN) - 1$

$b(CN) = h(4) - h(3)$

$= h(3) - k - 1 - h(3)$

$= -k - 1$

$b(GDCN) = MAX\ (h(3),h(4)) - MAX\ (h(1),h(2))$

$= MAX\ (h(3),h(3)-k-1) - MAX\ (h(1),h(2))$

$= h(3) - MAX\ (h(1),h(2))$

$= h(3) - MAX\ (h(3)+1-Bb(DCN),h(3)-Bb(GDCN))$

$= MIN\ (h(3)-(h(3)+1-Bb(DCN)),h(3)-(h(3)-Bb(GDCN)))$

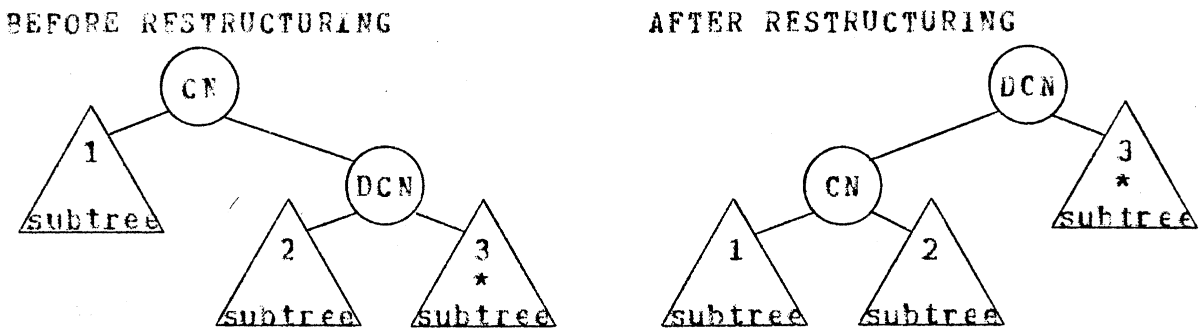
$= MIN\ (Bb(DCN)-1,Bb(GDCN))$

Q.E.D.

Note that since both Ab(DCN) and Ab(GDCN) depend upon Bb(DCN) and Bb(GDCN), one of the Bb values must be saved before changing it.

CASE 3: SIMPLE ROTATION

CN right heavy; DCN right heavy.



BEFORE restructuring, we know that:

h(DCN) = h(3) + 1

h(CN) = h(DCN) + 1

= h(3) + 2

and

b(DCN) = h(3) - h(2)

<= k

b(CN) = h(DCN) - h(1)

= h(3) + 1 - h(1)

= k + 1

AFTER restructuring, we know that:

h(CN) = MAX (h(1),h(2)) + 1

h(DCN) = MAX (h(CN),h(3)) + 1

and

b(CN) = h(2) - h(1)

b(DCN) = h(3) - h(CN)

= h(3) - MAX (h(1),h(2)) - 1

Since h(1), h(2), h(3), and k remain constant through the rotation, the following expressions remain true through the rotation:

$$Bb(DCN) = h(3) - h(2)$$

$$h(2) = h(3) - Bb(DCN)$$

$$h(3) + 1 - h(1) = k + 1$$

$$h(1) = h(3) - k$$

Substituting these expressions for h(1) and h(2) in the equations for AFTER b(n)'s gives:

$$\begin{aligned} b(CN) &= h(2) - h(1) \\ &= h(3) - Bb(DCN) - (h(3) - k) \\ &= k - Bb(DCN) \\ b(DCN) &= h(3) - MAX\ (h(1),h(2)) - 1 \\ &= h(3) - MAX\ (h(3)-k,h(3)-Bb(DCN)) - 1 \\ &= MIN\ (h(3)-(h(3)-k),h(3)-(h(3)-Bb(DCN))) - 1 \\ &= MIN\ (k,Bb(DCN)) - 1 \end{aligned}$$

Q.E.D.

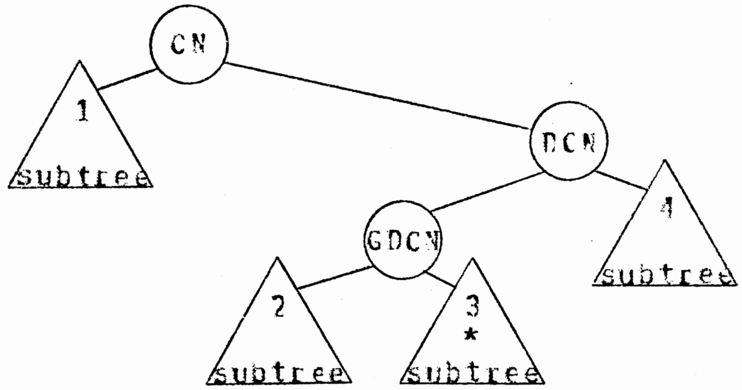
The expression for Ab(DCN) may be simplified further by noting that Bb(DCN) <= k. Therefore, MIN(k,Bb(DCN)) will always yield Bb(DCN) and Ab(DCN) = Bb(DCN) - 1.

<u>CASE 4</u>: SPLIT ROTATION

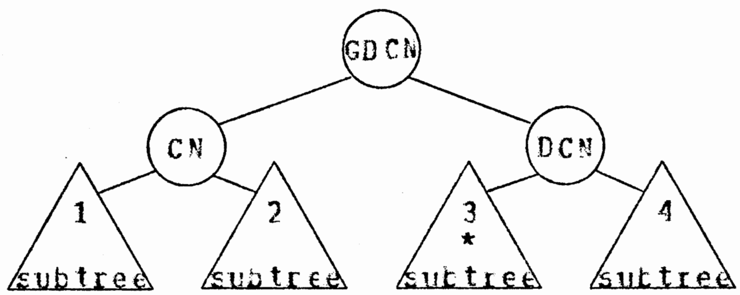
CN right heavy; DCN left heavy.

<u>Subcase a</u>: CDCN right heavy.

BEFORE RESTRUCTURING



AFTER RESTRUCTURING



BEFORE restructuring, we know that:

$$h(GDCN) = h(3) + 1$$

$$\begin{aligned} h(DCN) &= h(GDCN) + 1 \\ &= h(3) + 2 \end{aligned}$$

$$\begin{aligned} h(CN) &= h(DCN) + 1 \\ &= h(3) + 3 \end{aligned}$$

and

$$\begin{aligned} b(GDCN) &= h(3) - h(2) \\ &<= k - 1 \end{aligned}$$

$$\begin{aligned} b(DCN) &= h(4) - h(GDCN) \\ &>= -k \end{aligned}$$

$$\begin{aligned} b(CN) &= h(DCN) - h(1) \\ &= h(3) + 2 - h(1) \end{aligned}$$

$= k = 1$

AFTER restructuring, we know that:

$$h(DCN) = MAX\ (h(3), h(4)) + 1$$
$$h(CN) = MAX\ (h(1), h(2)) + 1$$
$$h(GDCN) = MAX\ (h(CN), h(DCN)) + 1$$

and

$$b(DCN) = h(4) - h(3)$$
$$b(CN) = h(2) - h(1)$$
$$b(GDCN) = h(DCN) - h(CN)$$
$$= MAX\ (h(3), h(4)) - MAX\ (h(1), h(2))$$

Since $h(1)$, $h(2)$, $h(3)$, $h(4)$, and k remain constant through the rotation, the following expressions remain true through the rotation:

$$Bb(GDCN) = h(3) - h(2)$$
$$h(2) = h(3) - Bb(GDCN)$$
$$Bb(DCN) = h(4) - h(3) - 1$$
$$h(4) = h(3) + Bb(DCN) + 1$$
$$h(3) + 2 - h(1) = k + 1$$
$$h(1) = h(3) + 2 - k - 1$$
$$= h(3) - k + 1$$

Substituting these expressions for $h(1)$, $h(2)$, and $h(4)$ in the equations for AFTER b(n)'n gives:

$$b(DCN) = h(4) - h(3)$$
$$= h(3) + Bb(DCN) + 1 - h(3)$$
$$= Bb(DCN) + 1$$
$$b(CN) = h(2) - h(1)$$
$$= h(3) - Bb(GDCN) - (h(3) - k + 1)$$

$= k - Bb(GDCN) - 1$

$b(GDCN) = MAX\ (h(3),h(4)) - MAX\ (h(1),h(2))$

$= MAX\ (h(3),h(3)+Bb(DCN)+1) - MAX\ (h(1),h(2))$

but since DCN was left heavy, $Bb(DCN) < 0$; hence,

$= h(3) - MAX\ (h(1),h(2))$

$= h(3) - MAX\ (h(3)-k+1,h(3)-Bb(GDCN))$

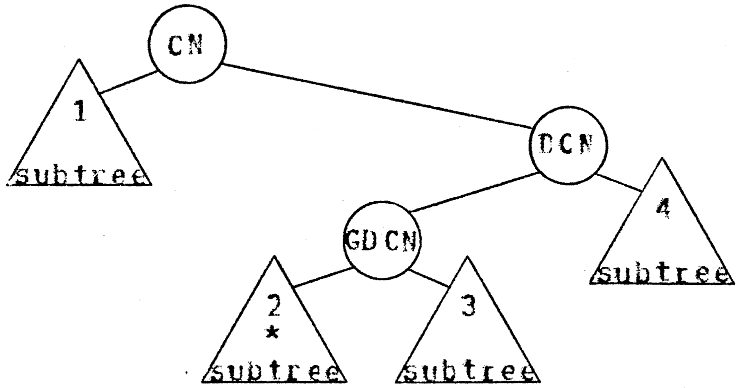
$= MIN\ (h(3)-(h(3)-k+1),h(3)-(h(3)-Bb(GDCN)))$

$= MIN\ (k-1,Bb(GDCN))$

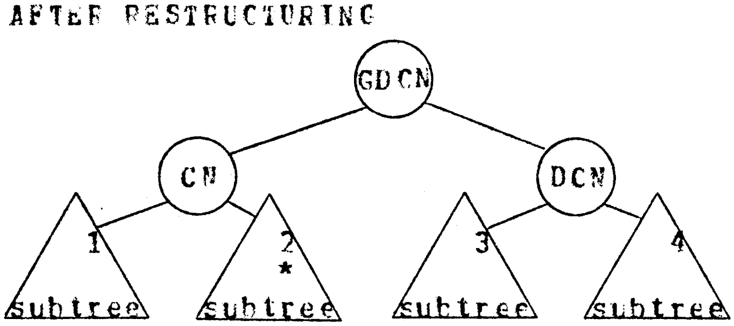
Q.E.D.

<u>Subcase b</u>: GDCN left heavy.

BEFORE RESTRUCTURING



AFTER RESTRUCTURING



BEFORE restructuring, we know that:

$$
\begin{aligned}
h(GDCN) &= h(2) + 1 \\
h(DCN) &= h(GDCN) + 1 \\
&= h(2) + 2 \\
h(CN) &= h(DCN) + 1 \\
&= h(2) + 3
\end{aligned}
$$

and

$$
\begin{aligned}
b(GDCN) &= h(3) - h(2) \\
&>= -(k - 1) \\
b(DCN) &= h(4) - h(GDCN) \\
&= h(4) - h(2) - 1 \\
&>= -k \\
b(CN) &= h(DCN) - h(1) \\
&= h(2) = 2 - h(1) \\
&= k + 1
\end{aligned}
$$

AFTER restructuring, we know that:

$$
\begin{aligned}
h(DCN) &= MAX\ (h(3),h(4)) + 1 \\
h(CN) &= MAX\ (h(1),h(2)) + 1 \\
h(GDCN) &= MAX\ (h(DCN),h(CN)) + 1
\end{aligned}
$$

and

$$
\begin{aligned}
b(DCN) &= h(4) - h(3) \\
b(CN) &= h(2) - h(1) \\
b(GDCN) &= h(DCN) - h(CN) \\
&= MAX\ (h(3),h(4)) - MAX\ (h(1),h(2))
\end{aligned}
$$

Since $h(1)$, $h(2)$, $h(3)$, $h(4)$, and k remain constant through the rotation, the following expressions remain true through the rotation:

$$Bb(GDCN) = h(3) - h(2)$$

$$h(3) = Bb(GDCN) + h(2)$$

$$Bb(DCN) = h(4) - h(2) - 1$$

$$h(4) = Bb(DCN) + h(2) + 1$$

$$h(2) + 2 - h(1) = k + 1$$

$$h(1) = h(2) - k + 1$$

Substituting these expressions for h(1), h(3), and h(4) in the equations for AFTER b(n)'s gives:

$$\begin{aligned}
b(DCN) &= h(4) - h(3) \\
&= Bb(DCN) + h(2) + 1 - (Bb(GDCN) = h(2)) \\
&= Bb(DCN) - Bb(GDCN) + 1 \\
b(CN) &= h(2) - h(1) \\
&= h(2) - (h(2) - k + 1) \\
&= k - 1 \\
b(GDCN) &= MAX\ (h(3),h(4)) - MAX\ (h(1),h(2)) \\
&= MAX\ (h(3),h(4)) - MAX\ (h(2)-k+1,h(2)) \\
&= MAX\ (h(3),h(4)) - h(2) \\
&= MAX\ (Bb(GDCN)+h(2),Bb(DCN)+h(2)+1) - h(2) \\
&= MAX\ (Bb(GDCN),Bb(DCN)+1)
\end{aligned}$$

Q.E.D.

Note that since both Ab(DCN) and Ab(GDCN) depend upon Bb(DCN) and Bb(GDCN), one of the Bb values must be saved before changing it.

APPENDIX B

HEIGHT OF THE SUBTREE DURING INSERTION ROTATION

During insertion restructuring, the height of the subtree involved remains the same . This is a demonstration of why it is true. Refer to Appendix A for a symbol description and preliminary derivation of formulas.

CASE 1: SIMPLE ROTATION

CN left heavy; DCN left heavy.

Before insertion, the height of the subtree rooted at CN = Bh(CN) - 1.

After restructuring, the height of the subtree = Ah(DCN).

$$Ah(DCN) = MAX\ (h(1), h(CN)) + 1$$
$$= MAX\ (h(1), h(2)+1, h(3)+1) + 1$$

but $h(2)+1 = h(1)+Bb(DCN)+1 <= h(1)$ since $Bb(DCN) < 0$

and $h(3)+1 = h(1)-k+1 \quad <= h(1)$ since $k > 0$

Hence,

$$Ah(DCN) = h(1) + 1$$
$$= Bh(DCN)$$

= Bh(CN) - 1

Q.E.D.

CASE 2: SPLIT ROTATION

CN left heavy; DCN right heavy.

Subcase a: GDCN left heavy.

Before insertion, the height of the subtree rooted at CN

= Bh(CN) - 1.

After restructuring, the height of the subtree = Ah(GDCN).

Ah(GDCN) = MAX (h(DCN),h(CN)) + 1

= MAX (h(1),h(2),h(3),h(4)) + 2

but h(1) = h(2) - Bb(DCN) + 1 <= h(2) since Bb(DCN) > 0

and h(3) = Bb(GDCN) + h(2) <= h(2) since Bb(GDCN) < 0

and h(4) = h(2) - k + 1 <= h(2) since k > 0

Hence,

Ah(GDCN) = h(2) + 2

= Bh(DCN)

= Bh(CN) - 1

Q.E.D.

Subcase b: GDCN right heavy.

Before insertion, the height of the subtree rooted at CN

= Bh(CN) - 1.

After restructuring, the height of the subtree = Ah(GDCN).

$$Ah(GDCN) = MAX\ (h(CN),h(DCN)) + 1$$
$$= MAX\ (h(1),h(2),h(3),h(4)) + 2$$

but $h(1) = h(3) + 1 - Bb(DCN) <= h(3)$ since $Bb(DCN) > 0$

and $h(2) = h(3) - Bb(GDCN) <= h(3)$ since $Bb(GDCN) > 0$

and $h(4) = h(3) - k - 1 <= h(3)$ since $k > 0$

Hence,

$$Ah(GDCN) = h(3) + 2$$
$$= Bh(DCN)$$
$$= Bh(CN) - 1$$

Q.E.D.

<u>CASE 3</u>: SIMPLE ROTATION

CN right heavy; DCN right heavy.

Before insertion, the height of the subtree rooted at CN = Bh(CN) - 1.

After restructuring, the height of the subtree = Ah(DCN).

$$Ah(DCN) = MAX\ (h(CN),h(3)) + 1$$
$$= MAX\ (h(1)+1,h(2)+1,h(3)) + 1$$

but $h(1)+1 = h(3) - k + 1 <= h(3)$ since $k > 0$

and $h(2)+1 = h(3) - Bb(DCN) + 1 <= h(3)$ since $Bb(DCN) > 0$

Hence,

$Ah(DCN) = h(3) + 1$

$= Bh(DCN)$

$= Bh(CN) - 1$

Q.E.D.

CASE 4: SPLIT ROTATION

CN right heavy; DCN left heavy.

Subcase a: GDCN right heavy.

Before insertion, the height of the subtree rooted at CN $= Bh(CN) - 1$.

After restructuring, the height of the subtree = Ah(GDCN).

$Ah(GDCN) = MAX\ (h(CN), h(DCN)) + 1$

$= MAX\ (h(1), h(2), h(3), h(4)) + 2$

but $h(1) = h(3) - k + 1 \quad <= h(3)$ since $k > 0$

and $h(2) = h(3) - Bb(GDCN) \quad <= h(3)$ since $Bb(GDCN) > 0$

and $h(4) = h(3) + Bb(DCN) + 1 <= h(3)$ since $Bb(DCN) < 0$

Hence,

$Ah(GDCN) = h(3) + 2$

$= Bh(DCN)$

$= Bh(CN) - 1$

Q.E.D.

<u>Subcase b</u>: GDCN left heavy.

Before insertion, the height of the subtree rooted at CN = Bh(CN) - 1.

After restructuring, the height of the subtree = Ah(GDCN).

Ah(GDCN) = MAX (h(DCN),h(CN)) + 1

= MAX (h(1),h(2),h(3),h(4)) + 2

but h(1) = h(2) - k + 1 <= h(2) since k > 0

and h(3) = Bb(GDCN) + h(2) <= h(2) since Bb(GDCN) < 0

and h(4) = Bb(DCN) + h(2) + 1 <= h(2) since Bb(DCN) < 0

Hence,

Ah(GDCN) = h(2) + 2

= Bh(DCN)

= Bh(CN) - 1

Q.E.D.

APPENDIX C

BNF DESCRIPTION OF INPUT TO THE RESEARCH PROGRAM

Appendix C gives the BNF (Backus-Naur Form) description of the input requirements for using the research program.

NOTATION LEGEND:

$ - The signal character; indicates that a keyword follows.

nnnnn - lower case letters; A syntactic category which must be rewritten.

NNNNN - uppercase letters; A keyword which must appear in that position.

e - epsilon; a null value or entry.

| - OR; indicates a choice.

{...} - indicates a set of information from which a choice may be made.

,()/ - single characters which must appear where indicated.

```
input -->
     test_case_series
test_case_series -->
     test_case_series  test_case
```

```
    |  test_case

test_case  -->

    $  { keyword_comment  $  |  e }  case_specification

    $  { keyword_comment  $  |  e }  initial_specification

    $  { keyword_comment  $  |  e }  manipulation_specification

    $  { keyword_comment  $  |  e }

       { measurement_specification  $

         {keyword_comment  $  |  e }  |  e }

       keyword_endcase

keyword_comment  -->

    COMMENT  { not reserved_words  |  e }

reserved_words  -->

    $

    |  ENDCASE

case_specification  -->

    CASE  case_number

case_number  -->

    integer  |  e


initial_specification  -->

    tree_specification

    $  { keyword_comment  $  |  e }  initial_function

    $  { keysord_comment  $  |  e }  keyword_go

tree_specification -->

    TREES  tree_spec

tree_spec  -->

    tree_spec  { ,  |  e }  cs_tree_spec
```

```
     |  gs_tree_spec

gs_tree_spec  -->

     number_of_trees  generalized_tree

     |  number_of_trees  specialized_tree

number_of_trees  -->

     integer  |  e

generalized_tree  -->

     HB ( balance_constraint )

     | PHB ( balance_constraint , path_length_constraint )

balance_constraint  -->

     integer  |  1

path_length-constraint  -->

     integer  |  1

specialized_tree  -->

     AVL

     |  BST

     |  PHB11


initial_function  -->

     INITIAL number_of_nodes

number_of_nodes  -->

     integer


keyword_go -->

     GO


manipulation_specification  -->
```

```
    FUNCTION  function_specification
    $  { keyword_comment  $  |  e }
       KEYSET  keyset_specification
    $  { keyword_comment  $  |  e}
       keyword_go
function_specification  -->
    INSERT
    |  DELETE
    |  INS/DEL   id_order-choice
id_order_choice  -->
    RANDOM
    |  ALTERNATING  { BY  set_size  |  e }
keyset_specification  -->
    ALTERNATING  alt_key_choice
    |  ORDERED  ord_key_choice
    |  RANDOM  ran_key_choice
    |  SHUFFLED  shuf_key_choice
alt_key_choice  -->
    ord_key_choice  { SET  setsize  |  e }
ord_key_choice  -->
    FROM  low_key  TO  high_key  { BY increment  |  e }
ran_key_choice  -->
    number_of_keys  BETWEEN  low_key  AND  high_key
    random_start
shuf_key_choice  -->
    ord_key_choice  random_start
low_key  -->
```

```
     integer
hich_key  -->
     integer
set_size  -->
     integer
increment  -->
     integer
number_of_keys  -->
     integer
random_start  -->
     SEED  series_start  |  e
series_start -->
     integer


measurement_specification  -->
     MEASURE  performance_measures
     $  { keyword_comment  $  |  e }
        keyword_go
performance_measures  -->
     performance_measures  measure
     |  measure
measure  -->
     ROTATION
     |  HEIGHT
     |  INTERNAL
     |  EXTERNAL
```

integer -->

integer digit

| digit

digit -->

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

There are several limitations and restrictions to the placement of some symbols and the values of others. These restrictions follow.

1. Since $ is used to signal that a keyword follows, it cannot be used in any place other than those indicated in the description (i.e. it cannot be within a COMMENT statement).

2. ENDCASE acts as a signal character for certain error correction procedures. Hence, its use in a COMMENT statement could create problems.

3. The largest integer which the program is currently designed to handle = 2**15 - 1 = 32767. Integers larger than 32767 will have unpredictable results. Similarly , the smallest integer which should be used is - (2**15 - 1) = - 32767. (- 32768 has special meaning within the program and should not be used.)

APPENDIX D

AN ILLUSTRATION OF THE USE OF THE COMMAND LANGUAGE

The meanings associated with each statement of the command language is best illustrated with examples. The following sample input sequences provide examples of the use of the command language statements. For ease in coordinating a statement with its explanation each input statement is placed on a separate line and is immediately followed by a COMMENT statement (indented in block form) explaining it. However, there are no column requirements for the input statements.

```
$CASE

    $COMMENT     - Singals the beginnign of a new test
    case. The driver program prepares to initialize a
    new set of trees. There is no test case number on
    the statement; since this is the first run, I will
    let the driver program number the test cases. On
    output, I expect this test case to be 'CASE NUMBER
    1.'

$TREES

    AVL      HB(1)      PHB(1,I)
```

$COMMENT - The trees to be used in this test case are being specified. Since there are no repeat counts in front of the trees' names, one tree of each type will be available. Since these specifications are equivalent trees (the I stands for infinity), what I will see is the results of any differences in the algorithms for maintaining the trees.

$INITIAL 1000

$COMMENT - I wish 1000 nodes to be available in the trees.

$GO

$COMMENT - Signals the end of the tree initialization input section. At this point, the trees are established with the requested number of nodes.

$FUNCTION INSERT

$COMMENT - I wish to build the tree by inserting a series of keys into the tree.

$KEYSET ALTERNATING FROM 1 TO 100

$COMMENT - The insertion is to use 100 keys with the values 1 - 100 in the alternating order: 1, 100, 2, 99, 3, 98, . . . , 50, 51. Since are is no BY specified, the default of 1 was assumed; hence, the sequence takes 1 value from the low end, then 1 value from the high end, then 1 from the low end, . . ., and so on.

$GO

$COMMENT - Indicates that a complete manipulation request has been found. The driver program should perform the request at this point.

$MEASURE

ROTATION, HEIGHT, INTERNAL

EXTERNAL

$COMMENT - I wish to see how many rotations were performed in order to maintain the balance criteria, what the heights of the trees are, and what the internal and external path lengths are. Note the lack of a comma after INTERNAL. Commas are optional; the facility has been provided only for user readability of the input data.

$GO

$COMMENT - All measurements desired have been listed. This is the time to take the measurements and print them out.

$FUNCTION INSERT

$COMMENT - Now, I wish to insert some more keys.

$KEYSET

100 RANDOM BETWEEN 1000 AND 32000

$COMMENT - This time, I want 100 keys randomly chosen between 1000 and 32000. Since I have not specified a SEED value, the driver program will generate one for me.
NOTE - The program attempts to generate 100 unique random keys; hence, the user should provide a large range to facilitate this process.

$GO

$COMMENT - Do the insertion of 100 random keys.

$MEASURE

INTERNAL EXTERNAL

$COMMENT - This time all I care about are the internal and external path lengths. Since I inserted 100 keys into a tree which already had 100 keys, I expect the statistics to print out that there are 200 keys currently in the trees.

$ENDCASE

$COMMENT - This is the end of the first test case.

VITA

Mary Beth Hernon

Candidate for the Degree of

Master of Science

Thesis: THE DESIGN AND APPLICATION OF A RESEARCH TOOL FOR HEIGHT BALANCED TREES

Major Field: Computing and Information Sciences

Biographical:

Personal data: Born 2 February 1954 in Longview, Washington. Moved to Massena, New York, in 1958. Married William Patrick Hernon on 24 May 1978.

Education: Graduated from Massena Central High School in June 1972. Received Bachelor of Science in Home Economics degree from Oklahoma State University, Stillwater, OK, in May 1976. Completed requirements for Master of Science degree in Computing and Information Sciences from Oklahoma State University in July 1979.

Professional Experience: Graduate teaching instructor for Intermediate Programming n the Computing and Information Sciences Department, Oklahoma State University, Fall 1978 - Summer 1979. Programming intern with the Research and Planning Information Division of the Oklahoma State Board of Regents for Higher Education, Summer 1978. Graduate teaching assistant for Introductory Programming in the Computing and Information Sciences Department, Fall 1977 - Spring 1978.