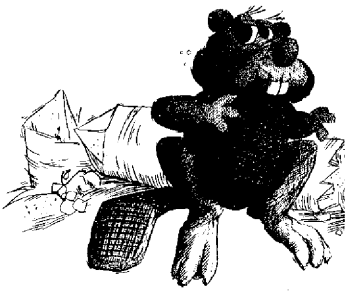
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT



Binary Search Trees with Binary Comparison Cost

Thomas Ottmann
Arnold L. Rosenberg
Hans-Werner Six
Derick Wood

Data Structuring Group
CS-83-37

December, 1983

BINARY SEARCH TREES WITH BINARY COMPARISON COST [1]

Thomas Ottmann [2]

Arnold L. Rosenberg [3]

Hans-Werner Six [2]

Derick Wood[4]

ABSTRACT

We introduce a new variant of the cost measure usually associated with binary search trees. This cost measure BCOST, results from the observation that during a search, a decision to branch left need require only one binary comparison, whereas branching right or not branching at all requires two binary comparisons. This is in contrast with the standard cost measure TCOST, which assumes an equal number of comparisons is required for each of the three possible actions. With BCOST in mind we re-examine its effect with respect to minimal and maximal BCOST trees, minimal and maximal BCOST-height trees, and introduce a class of BCOST-height-balanced trees, which have a logarithmically maintainable stratified subclass. Finally, a number of other issues are briefly touched upon.

1. INTRODUCTION

Although binary search trees have been used and investigated since the early days of computing there has always been a discrepancy between the implementation of searching in such a tree and the analysis of the cost of searching. On the one hand in Aho, Hopcroft, and Ullman (1983, p. 157), Gotlieb and Gotlieb (1978, p. 193), Horowitz and Sahni (1976, p. 439),

---

[1] This work was partially supported by NATO Grant RG 155.81, Natural Sciences and Engineering Research Council of Canada Grant No. A-5692, and National Science Foundation Grant No. MCS-8116522.

[2] Institute für Angewante Informatik und Formalebeschreibungsverfahren, Universität Karlsruhe, Postfach 6380, D-7500 Karlsruhe, W. Germany.

[3] Department of Computer Science, Duke University, Durham, NC 27706, U.S.A.

[4] Data Structuring Group, Computer Science Department, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

Knuth (1973, p. 424), Maurer (1977, p. 131), Standish (1980, p. 99) and Wirth (1976, p. 204) searching and updating binary search trees is carried out using binary comparators: with outcomes $[=, \neq]$, $[<, \geq]$ and/or $[>, \leq]$. On the other hand in Gotlieb and Gotlieb (1978, p. 195), Horowitz and Sahni (1976, p. 438), Knuth (1973, p. 427), Standish (1980, p. 101), and Wirth (1976, p. 212) the analysis of searching and updating costs is carried out under the assumption that a ternary comparator is used: with outcomes $[<, =, >]$ indicating whether $x<, =,$ or $> y$.

It is this discrepancy between implementation and analysis that has led us to consider a cost measure for binary search trees based on binary, rather than ternary, comparators. Note that not all authors fall into this abyss, for example. Aho, Hopcroft, and Ullman (1983) avoid it by considering the number of nodes visited in a search. However we claim that this new cost measure provides us, as we shall see in Section 2, with a model for search trees which is more realistic than the classical ternary-comparator based model.

The paper consists of a further four sections. Section 2 is motivational in nature, while in Section 3 we study the basic properties of BCOST and BCOST-height. In Section 4 we introduce the class of BCOST-height-balanced trees, the natural analogue of the well known class of height-balanced trees under TCOST. Finally in Section 5 we conclude with some comments on other possible avenues of investigation for BCOST.

2. SEARCHING IN BINARY SEARCH TREES

To see how a binary-comparator cost measure occurs in practice, it is worthwhile examining the typical search procedure provided in Aho, Hopcroft, and Ullman (1974, p. 117, 1983, p. 157), Gotlieb and Gotlieb (1978, p. 193), and Standish (1980, p. 99), for example.

```
function Search1(x;key;p:node):node;
{ initially p is the root of the given search tree. Search1 returns
either the node containing x if x is in the tree, or the value nil }

begin {Search1}
    if p= nil{ i.e. p is a leaf } then Search1:= nil else
    if x=selectkey(p) then Search1:=p else
    if x<selectkey(p)then  Search1:=Search1(x,leftchild(p)) else
                           Search1:=Search1(x,rightchild(p))
end {Search1};
```

In PASCAL *node* is a pointer type, and *selectkey*, *leftchild*, and *rightchild* are the corresponding selector functions.

First observe that the test for whether or not p denotes a leaf is an extra comparison at every node on the search path. We can avoid this by using the sentinel search technique (see for example Wirth (1976)). We would then introduce a new node named *Stop*, before creating the search tree. When constructing the tree, we have each leaf node point to *Stop*. Before each search, we insert the sought key in *Stop*. In this way we ensure that unsuccessful searches terminate at *Stop* and need not be tested for at each node. Specifically we have:

```
functionSearch2(xkey;p:node):node;
{ On entry p is the root of the given search tree. On exit
Search2 returns either the node containing x if x is in the tree, or
the value nil. A subsidiary function Srch2 is used }
var q: node;
function Srch2(x:key;p:node):node;
begin {Srch2}
    if x=selectkey(p) then Srch2:=p else
    if x<selectkey(p) then
                    Srch2:=Srch2(x,leftchild(p)) else
                    Srch2:=Srch2(x,rightchild(p))
end {Srch2};
begin {Search2}
    assignkey(Stop,x);q:=Srch2(x,p);
    if q=Stop then  Search2:= nil else
                    Search2:=q
end {Search2};
```

Observe that we place x in the node Stop before carrying out the search,

using an assignment procedure, which ensures that *Srch*2 is always successful, but if *Srch*2 terminates at Stop, then x is not in the search tree. The use of the sentinel node Stop has removed the extra comparison at each node on the search path that was present in *Search*1. However a further improvement can still be made, by noting that equality holds *only* at the final node on the search path. In other words since we have inequality at all nodes but the last one we should first test for inequality. We choose to test for $<$ first, giving *Srch*3:

```
function Srch3(x:key;p:node): node;
begin {Srch3}
    if x<selectkey(p) then Srch3:=Srch3(x,leftchild(p)) else
    if x=selectkey(p) then Srch3:=p else
                           Srch3:=Srch3(x,rightchild(p))
end {Srch3};
```

which replaces *Srch*2 in *Search*2 to give *Search*3. Essentially this version of the search procedure is found in, for example, Knuth (1973) and Wirth (1976). Interestingly if we are only allowed to use the while and repeat loop constructs in PASCAL, then *no* iterative version of *Search*3 with the same number of comparisons per search is possible, since any such would have to be similar to:

```
assignkey(Stop,x);
while x<>selectkey(p) do
    if x<selectkey(p) then p:=leftchild(p) else
                           p:=rightchild(p);
if p=Stop then Search4:= nil else Search4:=p
```

and immediately the test for equality is always carried out before the branching test. Only if we allow the use of a *goto* or an unconditional repeat with exit, can we obtain the same number of comparisons.

Returning to *Search*3, we see that a branch to the left results from one comparison, while a branch to the right results from two comparisons. Moreover equality results from two comparisons, plus a further comparison in *Search*3 to decide membership. Thus the number of comparisons required to decide membership of x in the search tree given by p does not depend only on the *length* of the search path as it does in *Search*2, for example, but also on the *number of left branches* (and hence right branches) on the search path. Hence if the search path contains m nodes, of which k are left-branching nodes, then the total number of comparisons is:

$$k+2(m-k)+1 \ ,$$

although there are only $k+2(m-k)$ key comparisons. It is this asymmetric measure of search cost that we study in the following sections.

3. EXTREMAL BCOST TREES

We first need:

Definition A *binary tree* T_n of n nodes is either the empty tree if $n = 0$ or, alternatively, consists of a triple (T_l, u, T_r) where $l+r+1 = n$, u is the root node, T_l is the left subtree of u, and T_r is the right subtree of u. A *binary search tree* for n distinct keys taken from a totally ordered key universe, is a binary tree T_n with each key associated to a unique node of T_n such that:

All the keys in the left subtree of each node u
$<$ the key associated with u
$<$ the keys associated with the right subtree of u.

The *height* of a binary (search) tree T_n is 0 if $n=0$ and (1+ the maximum of the heights of the left and right subtrees), otherwise.

We are now in a position to define the *binary-comparator* based search cost as well as the usual *ternary-comparator* based search cost. In each case the search cost is the total cost of searching for every key in the tree.

Definition Let T_n be a binary search tree with n nodes and keys. Then

$$BCOST(T_n) = \begin{cases} 0 \ , \ \text{if} \ \ n = 0 \\ \\ 2 + l + 2r + BCOST(T_l) + BCOST(T_r), \\ \qquad \text{otherwise} \ , \ \text{where} \ \ T_n = (T_l, u, T_r) \ , \end{cases}$$

and

$$TCOST(T_n) = \begin{cases} 0 \ , \ \ n = 0 \\ \\ n + TCOST(T_l) + TCOST(T_r) \ , \ \ \text{otherwise} \ , \\ \qquad \text{where} \ \ T_n = (T_l, u, T_r) \end{cases}$$

are the binary-comparator and ternary-comparator search costs, respectively.

Note that (with both *BCOST* and *TCOST*) we ignore the extra comparison required to examine a leaf, since it isn't a key comparison. In Figure 3.1 we display the cost of searching nodes in a prefix of the infinite binary tree.

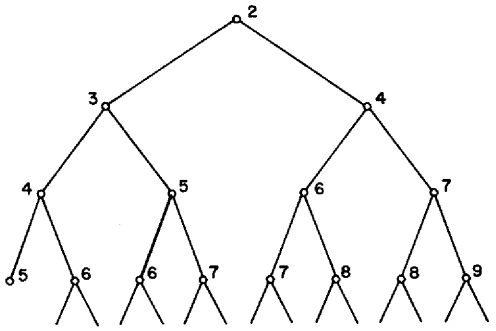


Figure 3.1

In [CW] binary search trees are investigated under a cost measure based on a cost of α to branch left, and a cost of β to branch right. These are called $\alpha-\beta$ binary search trees. Clearly we are investigating 1-2 binary search trees in this notation, and hence some of our results are subsumed by the more general investigation of [CW]. However the case $\alpha=1,\beta=2$ is easier to deal with directly and, moreover, the majority of our results are completely new.

It is well known (see Knuth (1973), for example) that:

$$n\left\lfloor \log_2 n \right\rfloor \leq TCOST(T_n) \leq \frac{n(n+1)}{2}$$

and that these extremal values are obtained with complete binary trees and *degenerate* binary trees (that is of height n), respectively.

Regarding *BCOST* we have:

$$? \leq BCOST(T_n) \leq n(n+1)$$

where the maximal or pessimal value is given by a degenerate right branching tree, see Figure 3.2. The minimal or optimal value is easily obtained from the following observation in Figure 3.1:

> A node u is reached from the root of a tree via k binary comparisons if and only if u is the root of the tree and $k = 2$, u is a left child whose parent is reached via $k-1$ comparisons, or u is a right child whose parent is reached via $k-2$

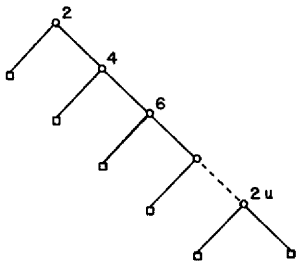


Figure 3.2

comparisons. We say u is *at binary distance* k from the root, meaning that k binary comparisons are needed to verify that the sought key resides there.

Hence the number of nodes at binary distance $k > 3$ from the root, is equal to the sum of the numbers of nodes at binary distances $k-1$ or $k-2$ from the root. Recalling that the i-th Fibonacci number

$$Fib(i) = \begin{cases} 0\,, \text{ if } i = 0 \\ 1\,, \text{ if } i = 1 \\ Fib(i-1) + Fib(i-2)\,, \quad \text{otherwise} \end{cases}$$

then we have:

Lemma 3.1 *In the infinite binary tree, for all* $k \geq 2$, *there are* $Fib(k-1)$ *nodes at binary distance* k *from the root.*

Proof: By the above remarks together with the fact that the root is at binary distance 2 and its leftchild is at binary distance 3 and these are the only nodes with these distances. □

Since branching to the left is less B-costly than branching to the right we are led to the following:

Definition The *left Fibonacci tree* F_i, $i \geq 0$, is defined recursively by:

$$F_i = \begin{cases} T_0 , & \text{if } i = 0 \\ T_1 , & \text{if } i = 1 \\ (F_{i-1}, u, F_{i-2}), & \text{for some new node } u \text{ , if } i \geq 2 . \end{cases}$$

As an example F_5 is shown in Figure 3.3. Note that F_i has height i and $Fib(i+2)$ leaves.

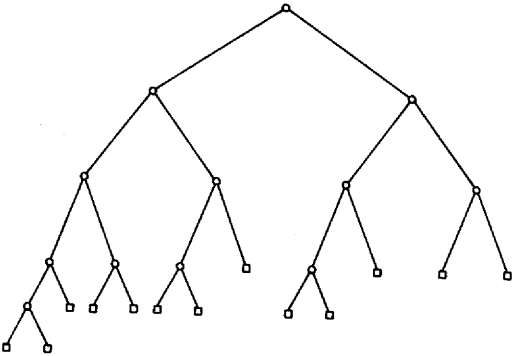


Figure 3.3

Lemma 3.2 *For all $n \geq 2$ with $n = Fib(i+2)-1$, for some $i \geq 1$, and for all trees T_n :*

$$BCOST(T_n) \geq BCOST(F_i) .$$

Moreover $BCOST(F_i) = iFib(i+2) - Fib(i+1) + 1.$

Proof: (by induction on i) We claim that F_i, $i > 0$, contains exactly $Fib(k-1)$ nodes at binary distance k from the root, for all k, $2 \leq k \leq i+1$. The claim is obvious for $i \leq 2$. Assume the induction hypothesis holds for all i, $1 \leq i < j$, for some $j > 1$. In F_{j-1} there are, for $3 \leq k \leq j+1$, $Fib(k-2)$ nodes at distance $k-1$ from the root of F_{j-1} and, for $4 \leq k \leq j+1$, $Fib(k-3)$ nodes at distance $k-2$ from the

root of F_{j-2}. Hence there are, for $4 \le k \le j+1$, $Fib(k-2) + Fib(k-3) = Fib(k-1)$ nodes at distance k from the root of F_j. For $k \le 3$ there are trivially $Fib(k-1)$ nodes at distance k from the root of F_j.

Second, $BCOST(F_i) = iFib(i+2) - Fib(i+1) + 1$ holds for $i = 0$ and 1. Assume it holds for all $i < m$, for some $m \ge 1$. Then

$$BCOST(F_m) = Fib(m+1) + 2 \cdot Fib(m) + BCOST(F_{m-1}) + BCOST(F_{m-2}) - 1$$

from the definition of *BCOST*. Substituting for $BCOST(F_{m-1})$ and $BCOST(F_{m-2})$ and rearranging terms we obtain the desired result. □

This leads to the characterization of minimal *BCOST* trees for all $n \ge 0$.

Theorem 3.3 *Let $n \ge 2$ be a given integer satisfying $Fib(i+2) \le n+1 < Fib(i+3)$, for some $i \ge 0$. Then a binary tree T_n has minimal BCOST iff either T_n equals F_n if $n = 0$ or 1, or T_n has F_i as a prefix and the remaining $n+1-Fib(i+2)$ nodes in T_n are at binary distance $i+2$ from the root.*

Proof: This follows directly from Lemma 3.2 and the observation that the $n+1-Fib(i+1)$ remaining nodes should be placed at the cheapest points, that is the positions at binary distance $i+2$ from the root. There are $Fib(i+1)$ of these positions, and because $Fib(i+1) > n+1-Fib(i+2)$, there are sufficiently many positions. □

It is worth noting at this point that the minimal *BCOST* trees are exactly the trees of a Fibonaccian search, see Knuth (1973).

After characterizing the optimal and pessimal *BCOST* trees, we turn to their average behavior. To begin with we first define the extended search costs, that is the total cost of searching a tree of n nodes for each of the $n+1$ gaps between the keys.

Definition Let T_n be a binary search tree of n nodes, $n \ge 0$. Then

$$EBCOST(T_n) = \begin{cases} 0, & \text{if } n = 0 \\ l+1+2(r+1)+EBCOST(T_l)+EBCOST(T_r) & \\ \quad \text{otherwise,} \quad \text{where} T_n = (T_l, u, T_r), & \end{cases}$$

and

$$ETCOST(T_n) = \begin{cases} 0\ , & \text{if } n = 0 \\ n+1+ETCOST(T_l)+ETCOST(T_r) & \text{otherwise} \\ \quad \text{where } T_n = (T_l,u,T_r)\ , & \end{cases}$$

are the extended *BCOST* and *TCOST* respectively.

The relationships between the cost measures are captured in:

Lemma 3.4 *For $n \geq 0$, and T_n a binary search tree:*

$$EBCOST(T_n) - BCOST(T_n) = ETCOST(T_n) - TCOST(T_n) = n\ .$$

Proof: Straightforward. □

When considering the average behavior of *BCOST* and *EBCOST*, we assume all $n!$ permutations of the integers 1 to n are equally likely insertion sequences for the standard insertion procedure when given an initially empty tree. Thus in the usual way (for example, see Knuth (1973)) we find, using *ABCOST* and *AEBCOST* to denote the average values of *BCOST* and *EBCOST*, respectively:

$$\begin{aligned} AEBCOST(n) &= AEBCOST(n-1) + \frac{AEBCOST(n-1)}{n} + 3 \\ &= \frac{(n+1)}{n} AEBCOST(n-1) + 3 \end{aligned}$$

assuming an insertion is equally likely to take place to the left of a frontier node as it is to the right. This recurrence can be solved using $AEBCOST(0)=0$ to give:

$$AEBCOST(n) = 3(n+1)[H_{n+1}-1]$$

which should be compared with:

$$AETCOST(n) = 2(n+1)[H_{n+1}-1]\ .$$

Applying the relationship given by Lemma 3.4 we obtain:

Theorem 3.5

$$ABCOST(n) = 3(n+1)H_{n+1} - 2n - 3 \ .$$

Recalling that each ternary comparison is usually implemented as two binary comparisons, cf. *Srch*2, this result implies that the expected search time should be 25% faster when using *Srch*3 as far as key comparisons are concerned. Furthermore since the expected search trees constructed by random insertions, as in the above model, are nearly optimal with respect to *TCOST*, we would expect a greater reduction in search time if the constructed trees are more nearly left Fibonacci. In the next section we introduce a class of balanced trees, balanced with respect to *BCOST*, which have this property.

4. BALANCED BCOST TREES

One may consider balanced varieties of *BCOST* trees, each member of a particular class having a near-optimal *BCOST*. In this section after providing an appropriate modification of the definition of height we define a class of height balanced trees which are nearly optimal. It is also possible to consider a modification of the definition of weight in order to obtain *BCOST*-weight balanced trees, but we leave this for the interested reader.

Definition Given a tree T_n , its *BCOST height* denoted by $Bht(T_n)$, is defined recursively as:

$$Bht(T_n) = \begin{cases} 0 \ , & \text{if } n = 0 \\ \max\{1+Bht(T_l) \ , \ 2 + Bht(T_r)\} & \text{otherwise} \\ \quad \text{where } T_n = (T_l,u,T_r) \ . & \end{cases}$$

Intuitively $Bht(T_n)$ denotes the maximal number of comparisons needed to locate a key in T_n, and is therefore a natural generalization of the usual height measure.

We may now define the class of *BCOST-height balanced trees*, or *Bhb trees*.

Definition A tree T_n is a *Bhb tree* if and only if:

either $n = 0$

or $T_n = (T_l,u,T_r)$, T_l and T_r are *Bhb* trees and $-1 \le Bht(T_l) - Bht(T_r) - 1 \le 1$.

In Figure 4.1 we display a *Bhb* tree with 8 nodes. Observe that the left Fibonacci trees are *Bhb* trees, since a left subtree of a left Fibonacci tree has a *Bht* exactly one greater than the *Bht* of its right brother. Hence the optimal *BCOST* trees are in the class of *Bhb* trees as we require.

Lemma 4.1 *Let $N_{\min}(d)$ be the minimum number of internal nodes necessary for a Bhb tree T which satisfies $Bht(T) = d$, for all $d \ge 2$.*

Then

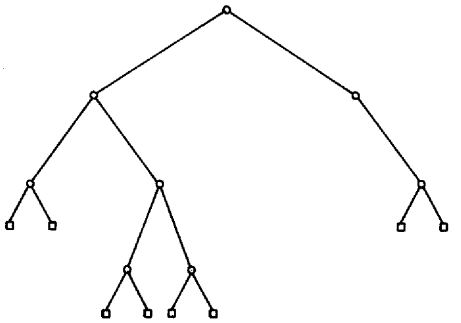
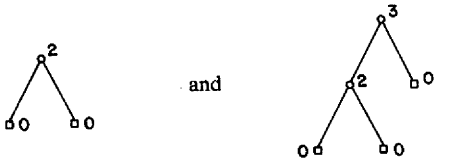


Figure 4.1

$$N_{\min}(d) = \begin{cases} 2^k-1, & \text{if } d = 2k \\ 3\cdot 2^{k-1}-1, & \text{if } d = 2k+1 \end{cases} .$$

Proof: By induction on k. Consider the basis $k=1$, then either $d=2$ or $d=3$. The only possible trees are:



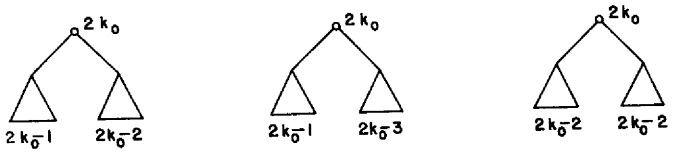
In the first case $N_{\min}(2) = 2^1-1 = 1$, and in the second $N_{\min}(3) = 3\cdot 2^0-1 = 2$, hence the lemma holds for $k=1$.

Now assume the lemma holds for all k, $1 \le k < k_0$, for some $k_0 \ge 2$ and consider the case $k=k_0$.

Case 1: $d = 2k_0$.

Clearly a *Bhb* tree T (with $Bhb(T) = d$) having a minimum

number of nodes must have left and right subtrees of the root with a minimum number of nodes. Thus we only need to consider the three possible cases:



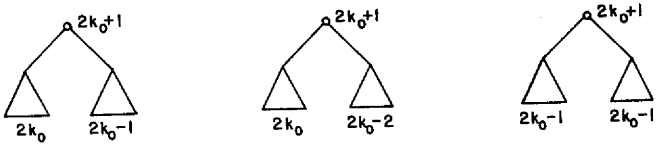
determined by the *Bhb* condition. Of these three the first clearly has more nodes than either of the other two. Moreover, because $(2k_0-1) = 2(k_0-1)+1$, $2k_0-3 = 2(k_0-2)+1$, and

$$3\cdot 2^{k_0-2} + 3\cdot 2^{k_0-3} = 9\cdot 2^{k_0-3} > 2^{k_0}$$

the second contains more nodes than the third. But this implies $N_{\min}(d) = 2^{k_0}-1$ as desired.

Case 2: $d = 2k_0+1$

A similar analysis yields three possible trees:



of which the first can be immediately discarded. Now

$$1 + 2^{k_0} - 1 + 2^{k_0-1}-1 = 3\cdot 2^{k_0-1} - 1$$

and

$$1 + 3\cdot 2^{k_0-2} - 1 + 3\cdot 2^{k_0-2} - 1 = 3\cdot 2^{k_0-1} - 1$$

hence both the second and third possibilities minimize the number of nodes, demonstrating that

$$N_{\min}(d) = 3 \cdot 2^{k_0-1} - 1 \ .$$

□

Observe that the proof of Lemma 4.1 also implies that complete binary trees maximize the *BCOST* height over all *Bhb* trees for a given number of nodes $n = 2^k - 1$ or $3 \cdot 2^{k-1} - 1$, for all $k \geq 1$. For all other values of n, the right complete binary trees fulfill this condition, where a right complete binary tree is a complete binary tree in which all nodes on the bottommost level are as rightmost as possible.

Hence we have:

Theorem 4.2 *Let T be a Bhb tree with n nodes. Then $Bht(T) \leq \lceil 2 \log_2(n+1) \rceil$.*

Proof: Since

$$N_{\min}(d+1) > n \geq N_{\min}(d)$$

for some $d \geq 2$, and

$$\log_2 x \leq \log_2(n+1)$$

where x is either 2^k or $3 \cdot 2^{k-1}$ depending on the evenness of d. The result then follows. □

Similarly just as among the height balanced trees the Fibonacci trees are the trees of maximal height (for the given number of nodes) so it is for the *Bhb* trees. For a given n the *Bhb* trees of maximal *BCOST* height are the right complete binary trees.

Having analyzed the static behavior of *Bhb* trees it remains to demonstrate that *Bhb* trees can be updated in logarithmic time, that is insertions and deletions of keys can be carried out in $O(\log n)$ time in an n node *Bhb* tree. For this purpose a subclass of the *Bhb* trees is defined. The subclass is stratified in the sense of van Leeuwen and Overmars (1982) and Ottmann, Schrapp and Wood (1983).

To define a class of stratified trees we need a number of *tops* and some trees which may appear in a *stratum*. For our purposes let $TOP = \{T_n : T_n \text{ is a } Bhb \text{ tree and } 0 \leq n \leq 33\}$ and $STRATUM = \{\bar{T}_5, \bar{T}_6\}$ where $\bar{T}_5$ and $\bar{T}_6$ are the *Bhb* trees of Figure 4.2. A *stratified BCOST tree* (or

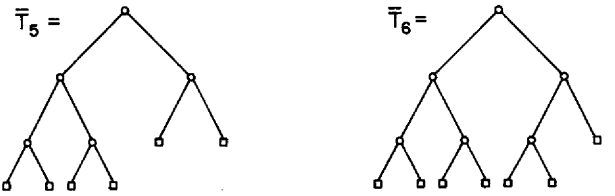


Figure 4.2

SB tree) S_m with respect to *TOP* and *STRATUM* belongs to *TOP* if $0 \leq m \leq 33$, and otherwise consists of some tree T from *TOP* with $\bar{T}_5$s and $\bar{T}_6$s attached to its leaves to form a stratum, and to the leaves of this first stratum $\bar{T}_5$s and $\bar{T}_6$s are attached to form a second stratum, and so on. See Figure 4.3 for an example of S_{135}.

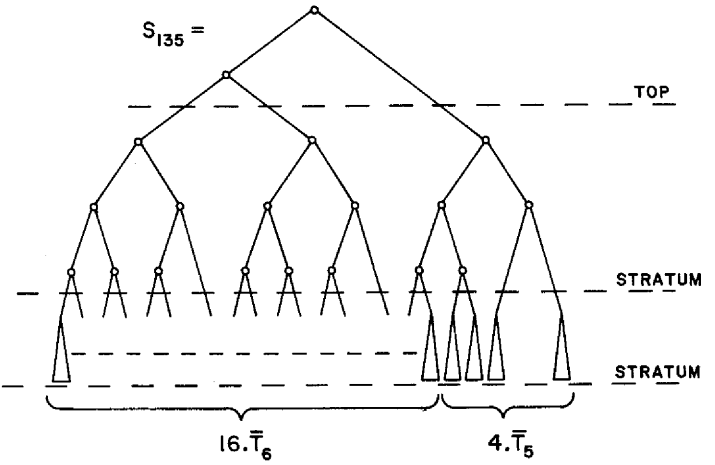


Figure 4.3

Lemma 4.3 *For all $m \geq 0$ there is an SB tree S_m. Furthermore every SB tree is a Bhb tree.*

Proof: The proof of the first part is by induction on m. The inductive step is constructive, that is to show that there is an S_k for given k, we begin with

an S_{k-1} and replace a $\bar{T}_5$ in the lowest stratum with a $\bar{T}_6$. If no such $\bar{T}_5$ exists, then 5 $\bar{T}_6$s, having 35 leaves, are replaced by 6 $\bar{T}_5$s, having 36 leaves, in the lowest stratum, passing the problem up to the penultimate stratum, see Figure 4.4.

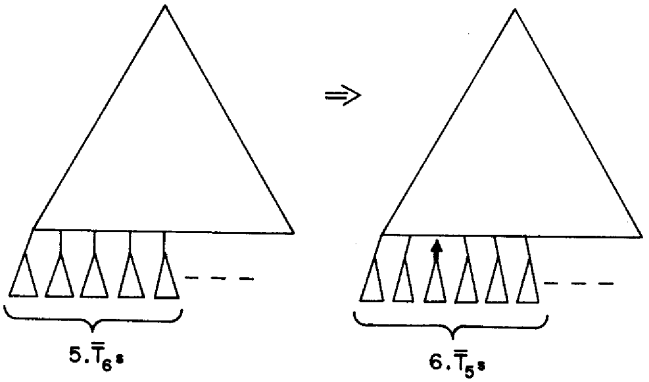


Figure 4.4

At worst this is repeated until the top T of the tree is reached. At this point if T has fewer than 33 nodes it is replaced by a new top with one more node than T, otherwise it is replaced by five $\bar{T}_6$ s hanging on a four-node tree from TOP, see Figure 4.5.

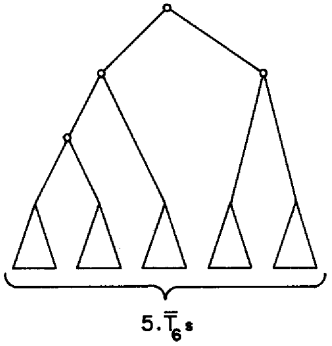


Figure 4.5

This yields a new top and a new stratum containing 34 nodes as required.

The only remaining problem is that when there is no $\bar{T}_5$ in a stratum there should be at least 5 $\bar{T}_6$ s to enable the problem to be passed up the tree at each stratum. Now if there are no $\bar{T}_5$ s and fewer than 5 $\bar{T}_6$ s then we must be in the stratum below the top. Clearly the top has at most 4 $\bar{T}_6$ s hanging from it. Hence $k \leq 27$ and a new top can be constructed directly.

The second part follows by observing that $Bht(\bar{T}_5) = Bht(\bar{T}_6) = 5$, hence adding trees from *STRATUM* to all the leaves of a *Bhb* tree T only affects the heights of the nodes in T, not the differences of heights of brother nodes. Moreover $\bar{T}_5$ and $\bar{T}_6$ are both *Bhb* trees, hence each *SB* tree is *Bhb*. □

Theorem 4.4 *For all $n \geq 0$, every SB tree S_n can be updated in* $O(\log n)$ *time.*

Proof: We consider insertion only, deletion follows in a similar manner. We are given an *SB* tree T and a value x, which is to be inserted. As usual first search for x in T to determine that it is not present (if it is present the insertion is redundant).

Case 1: This occurs when T belongs to *TOP*. If T contains less than 33 nodes, then simply replace T by a new tree T' from *TOP* having one more node and fill in the corresponding values. If T has 33 nodes then we must replace it by a T' having 34 nodes consisting of a top and one stratum. This is the tree of Figure 4.5. This requires constant time. Otherwise T has at least one stratum. In this setting we say two subtrees of T are *siblings* if they belong to *STRATUM*, are in the same stratum, and are attached to the leaves of a tree, its *parent*, either in *TOP* or again in *STRATUM*. Now the value x is to be added at the leaf level of the bottom stratum within a $\bar{T}_5$ or a $\bar{T}_6$, that is a new node is to be added. Consider these cases in turn;

Case 2: A node is to be added to a $\bar{T}_5$.
Simply replace $\bar{T}_5$ by a $\bar{T}_6$ and fill in the values appropriately.

Case 3: A node is to be added to a $\bar{T}_6$.

(3.1)One sibling of the $\bar{T}_6$ is a $\bar{T}_5$. Modify the parent and its siblings such that one $\bar{T}_5$ sibling is replaced by a $\bar{T}_6$. Fill in the values appropriately.

(3.2)All siblings are $\bar{T}_6$s and there are five siblings. Replace the five siblings by five $\bar{T}_5$s and one $\bar{T}_6$ (= 31 nodes). This causes a recursive insertion into the parent.

(3.3)All siblings are $\bar{T}_6$s and there are six siblings. Replace the six $\bar{T}_6$s by five $\bar{T}_5$s and two $\bar{T}_6$s (= 37 nodes). This causes a recursive

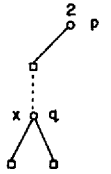
insertion into the parent.

It is not difficult to see that this node insertion algorithm performs correctly taking $O(\log n)$ time. Moreover the ordering of the keys at each step is maintained by considering only the current "window", that is a parent and all its children. □

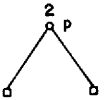
Whether or not $O(\log n)$ time update algorithms exist for the whole class of *Bhb* trees is an open question. It is, however, possible to design $O(\log^2 n)$ time update algorithms in this case.

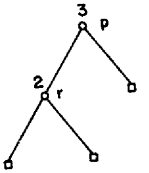
We provide a sketch of the $O(\log^2 n)$ time insertion algorithm for *Bhb* trees; the $O(\log^2 n)$ time deletion algorithm is similar. Both algorithms may call a subsidiary procedure Down at every node on the search path. It is this procedure, which requires $O(\log n)$ time, that causes $O(\log^2 n)$ time performance in the worst case.

To insert a new value x into a *Bhb* tree T, we first search for its associated leaf with parent p. A new node q is then created having value x. We always arrange for q to be added as the left child of p, viz:



If q is to be added as the right child of p, then the *Bhb* condition implies that p is:

either 

or 

and the values in p, q (and r) can easily be rearranged so that p is increased in *BCOSTheight* by one in all cases.

Now call *Restructure*(p). For convenience we use $\lambda p, \rho p$, and πp to denote the left child, right child, and parent of a node p.

Algorithm *Restructure(p);*
On entry p's children are *Bht* balanced and before the insertion let $Bht(p) = h+2$.
On exit all nodes on the path from the root to p are *Bht* balanced.

begin There are four cases to consider.

Case 1: The insertion occured in the right subtree of p, $Bht(\rho p)$ has increased from h to $h+1$, and, hence, $Bht(\lambda\rho) - Bht(\rho p) - 1 = -2$.

Case 2: The insertion occurred in the left subtree of p $Bht(\lambda p)$ has increased from $h+1$ to $h+2$, and, hence, $Bht(\lambda p) - Bht(\rho p) - 1 = 2$.

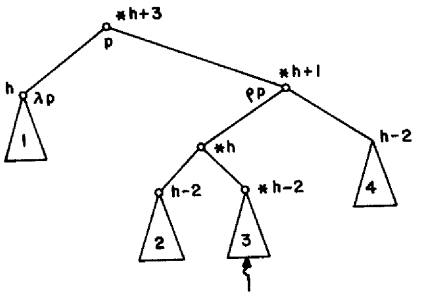
Case 3: $Bht(p) = h+3$ and the subtree at p is balanced. If p is the root then return otherwise $Restructure(\pi p)$.

Case 4: $Bht(p) = h+2$. Return.

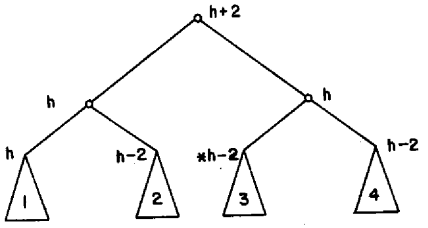
We examine the first two cases in more detail.

Case 1.1: $Bht(\lambda\rho p)$ has increased by 1.
Thus $Bht(\lambda\rho p) = h$ and $Bht(\rho\rho p) = h-2$ is the only possibility.

Case 1.1.1: $Bht(\rho\lambda\rho p)$ has increased by 1. We must have:

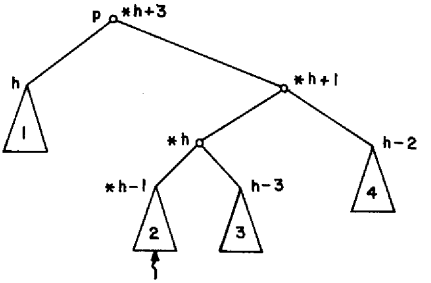


We perform a double rotation on p to the left, yielding:

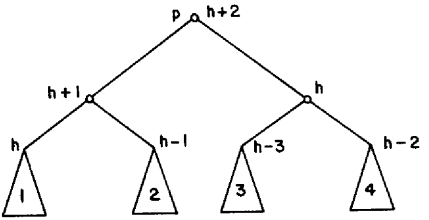


and p is not only balanced but also retains the same $BCOSTheight$, hence the tree is now rebalanced.

Case 1.1.2: ***Bht***(λλρ*p*) has increased by 1. We must have:

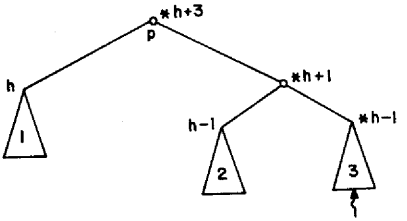


Again performing a double rotation on *p* to the left, we obtain:

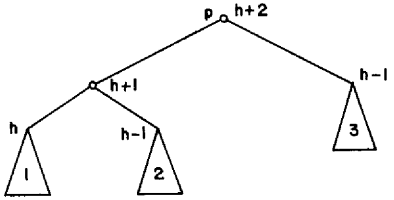


and *p* has the same ***BCOSTheight*** as before, but ρ*p* is now unbalanced. However a call of procedure ***Down***(ρ*p*) resolves this, resulting in ***Bht***(ρ*p*) = *h*−1 . Thus *p* remains balanced.

Case 1.2: ***Bht***(ρρ*p*) has increased by 1. We must have:



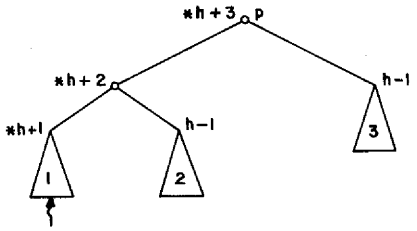
Perform a left rotation on *p*:



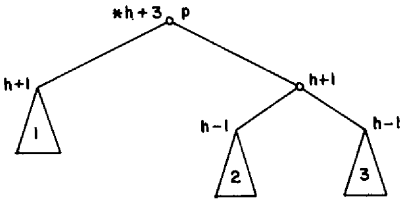
in which case the tree is now rebalanced.

Now turning to insertion in λp:

Case 2.1: ***Bht***$(\lambda\lambda p)$ **has increased by 1. We must have:**

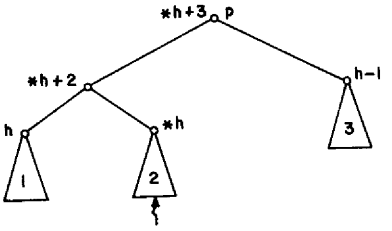


A right rotation at p yields:

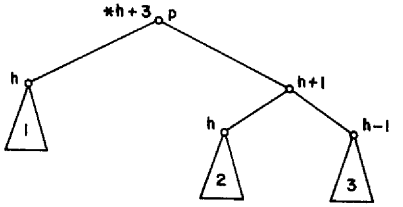


and we call ***Restructure***(πp) .

Case 2.2: ***Bht***$(\rho\lambda p)$ **has increased by 1. We must have:**



and a right rotation at p yields:



A call of *Down*(p) gives $Bht(p) = h+2$, hence the resulting subtree at p is balanced, and so is the whole tree.

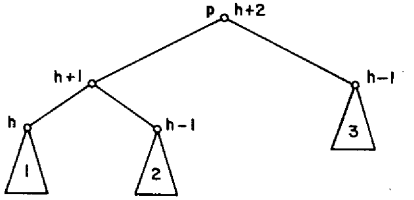
end of *Restructure*.

We now specify *Down*(p).

Algorithm *Down (p)*;
On entry let $Bht(p) = h+3$. Then either $Bht(\lambda p) = h$ and $Bht(\rho p) = h+1$, or $Bht(\lambda p) = h-1$ and $Bht(\rho p) = h+1$. The latter possibility only occurs within recursive calls of *Down*.
On exit $Bht(p) = h+2$ and the subtree at p is *Bht*-balanced.

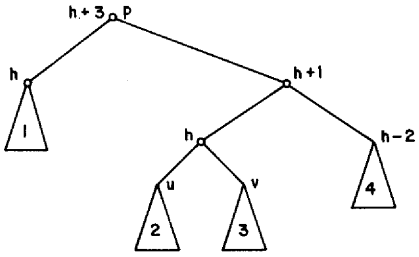
begin
Case 1: $Bht(\lambda p) = h$ and $Bht(\rho p) = h+1$.

Case 1.1: $Bht(\lambda\rho p) = Bht(\rho\rho p) = h-1$. Apply a left rotation:

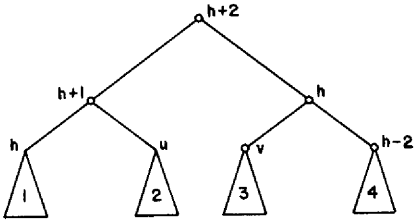


and return.

Case 1.2: $Bht(\lambda\rho p) = h$ and $Bht(\rho\rho p) = h-2$, that is:



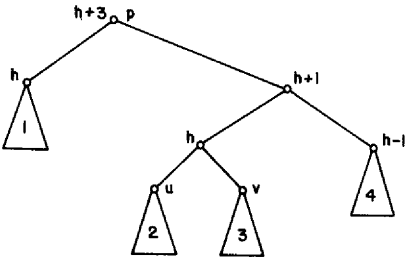
Apply a double left rotation yielding:



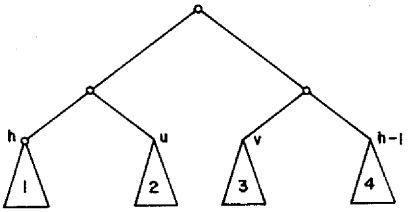
Case 1.2.1: $Bht(u) = h-1$ or $h-2$ and $Bht(v) = h-2$.
Return.

Case 1.2.2: $Bht(u) = h-1$ and $Bht(v) = h-3$.
Down(ρp).

Case 1.3: $Bht(\lambda\rho p) = h$ and $Bht(\rho\rho p) = h-1$, that is:



Applying a double rotation:



Case 1.3.1: $Bht(u) = h-1$ or $h-2$ and $Bht(v) = h-2$.
$Down(\rho p)$.

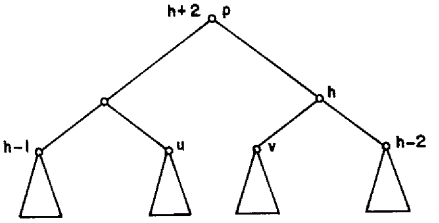
Case 1.3.2: $Bht(u) = h-1$ and $Bht(v) = h-3$.
$Down(\rho p)$.

Case 2: $Bht(\lambda p) = h-1$ and $Bht(\rho p) = h+1$.

Case 2.1: $Bht(\lambda\rho p) = Bht(\rho\rho p) = h-1$.
As Case 1.1.

Case 2.2: $Bht(\lambda\rho p) = h$ and $Bht(\rho\rho p) = h-1$.
A double left rotation at p is followed by a call of $Down(\rho p)$, similar to Case 1.2.

Case 2.3: $Bht(\lambda\rho p) = h$ and $Bht(\rho\rho p) = h-2$.
After a double left rotation at p we have:



Case 2.3.1: $Bht(u) = h-1$ or $h-2$ and $Bht(v) = h-2$.
Return.

Case 2.3.2: $Bht(u) = h-1$ and $Bht(v) = h-3$.
Down(ρp) .

end of *Down*.

We close this section by observing that there is an interesting duality between height balanced trees and *BCOST*-height balanced trees as far as their optimal and pessimal heights are concerned. We conjecture that this table also holds for comparison costs as well, that is *TCOST* for height balanced trees and *BCOST* for *BCOST*-height balanced trees.

DUALITY	Optimal	Pessimal
height balanced trees	complete binary tree	Fibonacci tree
BCOST-height balanced trees	left Fibonacci tree	right complete binary tree

5. CONCLUDING REMARKS

We have introduced a new cost measure for binary search trees, namely *BCOST*. We have investigated some of the theoretical aspects of this new cost measure, while leaving many questions open. Before briefly discussing some of these, we mention one experiment which needs to be carried out. If, as we claim, *BCOST* is a more realistic cost measure, then it is to be expected that this would show up in practice. For example computing the total time taken to perform many random searches of a complete binary tree with *Search*2 and *Search*3. These times should then be compared with those taken by searches of a left Fibonacci tree with *Search*2 and *Search*3. In both cases our theoretical results lead us to expect *Search*3 to perform better than *Search*2, and *Search*3 on a left Fibonacci tree should outperform *Search*3 on a complete binary tree of the same size. Such an experiment is currently being mounted.

Our new cost measure distinguishes between the number of binary comparisons required in a search and the number of nodes visited. Such as approach has been taken for 2-3 trees, see [RS] and the papers cited therein. The traditional cost measure is, in reality, a node visit cost. The time taken to search a tree should probably be modelled by a combination of these two cost measures rather than either alone.

If *BCOST* is indeed a more appropriate cost measure than *TCOST*, then the class of *Bhb* trees attains a greater significance than the traditional class of *hb* trees. Hence it becomes crucial to find efficient, that is $O(\log n)$, update algorithms for *Bhb* trees or, alternatively, to find a new class of trees which is *BCOST*-balanced and has $O(\log n)$ update algorithms. (For example is it possible to define a class of *BCOST* weight-balanced trees?) In [ORSW] the class of left-sided *hb* trees is considered as a possible candidate since it has $O(\log n)$ update algorithms. However, it shown in [ORSW] that the *Bht* of a left-sided *hb* tree T of n nodes, can be up to 44% greater than the maximal *Bht* of a *Bhb* tree of n nodes. However, this may, in practice, be a small price to pay for obtaining a *BCOST*-balanced class of trees with reasonably simple and logarithmic updating algorithms. The average or expected *BCOST* of left-sided *hb* trees remains an open problem, as indeed it is even under *TCOST*.

If the keys to be represented have weights associated with them, then the cost of constructing an optimal weighted binary search tree under *BCOST* is an obvious problem. It appears that the standard dynamic programming approach, see [K], will suffice. However it is conceivable that the monotonicity principle does not hold in this case, thereby preventing the speed up obtainable under *TCOST*.

Apart from the $\alpha-\beta$ binary trees of [CW], (recall that *BCOST* binary trees are $\alpha-\beta$ binary trees with $\alpha=1$ and $\beta=2$), the only other investigation of a biased search cost measure is that of [RS]. [RS] investigate minimal cost 2-3 trees, in which the cost of matching the second key of a ternary node is twice that required to match the first. However their cost measure is still based on ternary comparisons. Clearly minimal *BCOST* 2-3 trees may be

investigated along the lines of [RS] and the present paper leading, one would expect, to similar results.

REFERENCES

[AHU1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, Mass., 1974.

[AHU2] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *Data Structures and Algorithms*, Addison-Wesley Publishing Co., Reading, Mass., 1983.

[CW] Choy, D.M., and Wong, C.K., Bounds for Optimal $\alpha - \beta$ Binary Trees, *BIT 17* (1977), 1-15.

[GG] Gotlieb, C.C., and Gotlieb, L.R., *Data Types and Structures*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[HS] Horowitz, E., and Sahni, S., *Fundamentals of Data Structures*, Computer Science Press, Inc., Rockville, Maryland, 1976.

[K] Knuth, D.E., *The Art of Computer Programming, Volume III: Sorting and Searching*, Addison-Wesley Publishing Co., Reading, Mass. 1973.

[M] Maurer, H.A., *Data Structures and Programming Techniques*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[ORSW] Ottmann, Th., Rosenberg, A.L., Six, H.W., and Wood, D., A Realistic Cost Measure for Binary Search Trees, *Proceedings of the 7th Conference on Graph Theoretic Concepts in Computer Science (WG81)*, (ed. J.R. Mühlbacher), Carl Hanser Verlag, Vienna (1982), 163-172.

[OSW] Ottmann, Th., Schrapp, M., and Wood, D., On 1-Pass Top-Down Update Algorithms for Classes of Stratified Balanced Search Trees, Research Report CS-83-11, Computer Science Department, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

[RS] Rosenberg, A.L., and Snyder, L., Minimal Comparison 2, 3 Trees, *SIAM Journal on Computing 7* (1978), 465-480.

[S] Standish, T.A., *Data Structure Techniques*, Addison-Wesley Publishing Co., Reading, Mass., 1980.

[vLO] van Leeuwen, J., and Overmars, M., Stratified Balanced Search Trees, *Acta Informatica 18* (1982), 345-359.

[W] Wirth, N., *Algorithms and Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.