A BALANCED TREE STORAGE AND RETRIEVAL ALGORITHM

Gary D. Knott
Division of Computer Research and Technology
National Institutes of Health
Bethesda, Maryland

ABSTRACT

A storage and retrieval scheme which places items to be stored at the nodes of a binary tree is discussed. The tree is always balanced in a certain sense thus insuring that no excessively long search paths can exist. In addition to presenting the storage and retrieval algorithms, the deletion problem is also solved. The programming approaches involved yield a non-trivial case study of list-processing techniques. Finally, a cost analysis is given.

KEY WORDS AND PHRASES

storage and retrieval, searching, data structures, balanced trees

1. Introduction

Below we describe a storage and retrieval method using a data structure which we shall call a balanced tree. This scheme was given by C. C. Foster in [4], where he attributes it to G. M. Adel'son-Vel'skij and Y. M. Landis [1]. It has been used by David E. Ferguson [3] as a means for the organization of symbol tables in assemblers. The deletion procedure given later, as well as the determination of an upper bound for the number of compares required to search n items organized in a balanced tree, are due to comments by Donald E. Knuth. George A. Miller contributed significantly to the algorithms given here by uncovering several bugs.

The method of storing randomly-received items as nodes in a binary tree such that x is a left son of y only if $x<y$, and x is a right son of y only if $y \leq x$ is well-known [2, 6, 8, 9]. Given such a tree and an item key, the search time (number of 3-way compares) required to find a matching item is generally a logarithmic function of n, where n is the number of items or nodes in the given tree.

However, a search time which is necessarily a logarithmic function of n occurs only when the given tree is balanced, that is, when no excessively short and long paths exist. Such paths tend to increase the number of compares required on the average.

The scheme described below controls the structure of the tree which is generated by dynamically adjusting it, if required, to insure it will, in fact, always be balanced. These adjustments increase the update costs, but they insure that search costs are always bounded by a logarithmic function of n. Furthermore, the increase in update cost is modest and may often be less than actual update costs in an unbalanced tree, due to the increased search time which may be required there.

2. The Underlying Data Structure

We shall take as our basic data structure an area D composed of four vectors (V, L, R, B). Thus D may be considered as a matrix of four columns, or alternatively, as a collection of four-component row vectors. A node is represented by a particular quadruple $(V_i, L_i, R_i, B_i) = D_i$. V_i is the value of the node; in general we think of V_i holding an item, or a pointer to an item contained in the set of items to be stored and retrieved. L_i is an index to the left son node, D_{L_i}, of the current node, or zero if no left son exists. R_i is similarly an index to the right son node, D_{R_i}, of the current node, or zero if no right son exists. B_i is an integer value such that $|B_i| \leq 1$. B_i is called the balance of the tree or sub-tree whose root is the current node. B_i is defined more precisely below.

f is defined as an index to D such that D_f is the first empty quadruple in D. We shall assume that initially f is properly set. Moreover, V_f is an index to the next free quadruple in D, and so on. Thus we have a standard free-space list organized in D with f taken as the head of the list. If the quadruple D_a is the last node of the free-space list, then $V_a = 0$.

The root node of the entire tree stored in D is at D_{R_0}. That is, R_0 is an index to the quadruple which holds the root-node of our tree, wherever it may be in D. Thus R, considered as a vector, is defined as R[0:s], for some value s, and is thus indexed using 0-origin indexing. V, L, and B, on the other hand, are not accessed at zero, and hence, may be considered to be declared as V[1:s], L[1:s], and B[1:s]. We shall assume that R_0 is initially zero, indicating that no nodes exist.

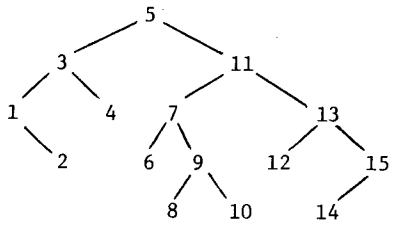
The tree stored in D is always organized such that for any node indexed by i, $V_{L_i} < V_i$ and $V_{R_i} > V_i$ whenever L_i and R_i are not zero, respectively.

Now, for our binary tree, consider each node $D_i = (V_i, L_i, R_i, B_i)$, and define h(i) as the height of the sub-tree with root node D_i. That is, h(i) is the number of nodes in a longest path in the sub-tree with root node i. Also h(0) = 0, since we understand the longest path of the null tree to be of length zero. Then we may define B_i for an arbitrary node D_i as follows:

$$B_i = h(R_i) - h(L_i) .$$

We shall say a tree is balanced if $|B_i| < 2$ for every node i in the tree. Clearly we should not demand more, since a number of nodes which is not a power of two less one cannot be organized so that every B_i is zero. A balanced tree thus approximates our notion of a tree where no excessively long or short paths exist. Moreover a stronger notion of balance would require excessive labor to maintain a balanced tree upon the addition or deletion of items.

Consider the following example:

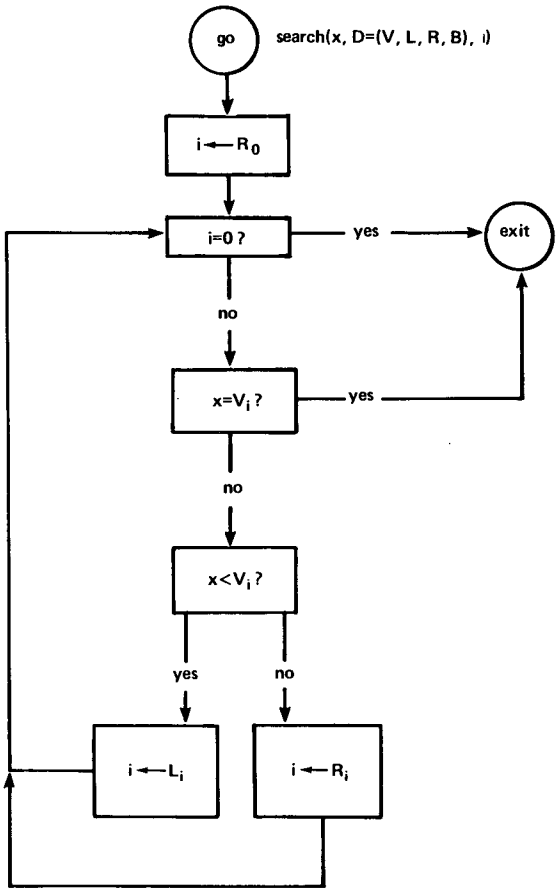


Here the integers represent the key values of the various nodes. This is seen to be a balanced tree by inspection. The above tree may actually be stored in D as follows:

	D			
	V	L	R	B
0			4	
1	10	0	0	0
2	1	0	6	1
3	3	2	12	-1
4	5	3	14	1
5	8	0	0	0
6	2	0	0	0
7	6	0	0	0
8	14	0	0	0
9	13	13	15	1
10	7	7	11	1
11	9	5	1	0
12	4	0	0	0
13	12	0	0	0
14	11	10	9	0
15	15	8	0	-1
f = 16	17	-	-	-
.				
.				
.				

3. The Search Procedure

The search routine for our defined tre is simple. It takes the argument x, which i key-value to be retrieved in D (we will cons tently confuse the key of an item with the i itself), and the argument D = (V, L, R, B). returns the final argument i as the index ir the first matching node, or as zero if there matching node. Thus we have:



4. The Storage Procedure

The add procedure for inserting a new node is more complex. It works in the following manner:

Given a new item x to be added, a search is done to discover what new leaf node position the new item is to occupy. This search defines a particular path from the root to that leaf node. Unlike the simple search given above, the path so defined must be remembered. A local push-down stack is a suitable device to save the successive nodes of such a path together with an indication of the direction from which they were exited.

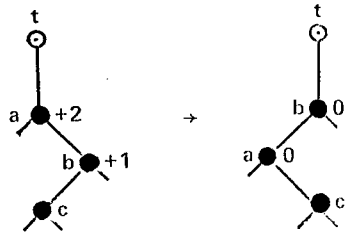
When x has been placed in D, the search path is retraced from bottom to top. At each previously-passed node, the balance at that node is recalculated, depending upon the direction of exit during the original traverse. Whenever a balance value indicates we have created an unbalanced sub-tree (i.e. whenever $|B_i|=2$), we shall immediately adjust the linkages in that sub-tree, including changing the root by changing a link which is "exterior" to the given sub-tree, so as to obtain a balanced tree with height equal to the previous height — before the addition occurred.

Note this can always be done because there is always some space or slack in an unbalanced tree. If every space were filled, we would have a perfect binary tree, but such a tree cannot be unbalanced.

Having balanced the first sub-tree which threatens the balance of the main tree, no further including tree will become unbalanced, nor will the various B-values be changed. Hence we are done. If during retracing we update all the B-values without needing to balance a sub-tree, then again we are finished. In particular, if we change a B-value to zero, then we may be assured we shall not need to balance any sub-tree, for no further B-values will change, once a zero is obtained. This is because if +1 (or -1) becomes a zero, then we have in fact only more perfectly balanced our tree.

4.1 The Balancing Algorithm

Now it remains only to describe the balancing algorithm. Whenever a B-value is +2 or -2, a right or left son respectively of that node exists and has a B-value of +1 or -1. Thus we have four cases. They will require two essentially different transformations. The basic idea is to "rotate" the tree to be balanced in the "direction" of its "small side." The simplest of the basic transformations is shown by example as:



The integers shown are the various balances. Here we have rotated the tree to the left. In terms of the actual representation we have:

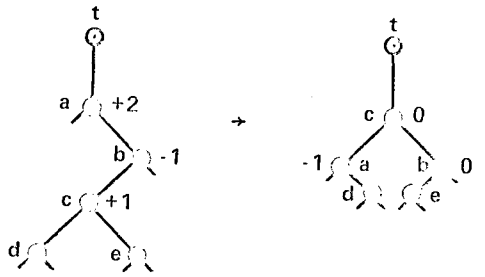
	V	L	R	B
t	V_t	a	R_t	B_t
a	V_a	L_a	b	2
b	V_b	c	R_b	1
c	V_c	L_c	R_c	B_c

→

	V	L	R	B
t	V_t	b	R_t	B_t
a	V_a	L_a	c	0
b	V_b	a	R_b	0
c	V_c	L_c	R_c	B_c

where we have assumed the sub-tree shown was reached from the left link of node D_t.

The second transformation arises when a simple rotation again produces an unbalanced tree. In such a case a deeper rotation will set things right. This case is shown in the following example.



Or, in terms of the actual representation, we have:

	V	L	R	B
t	V_t	a	R_t	B_t
a	V_a	L_a	b	2
b	V_b	c	R_b	-1
c	V_c	d	e	1
d	V_d	L_d	R_d	B_d
e	V_e	L_e	R_e	B_e

→

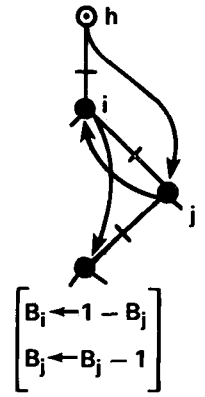
	V	L	R	B
t	V_t	c	R_t	B_t
a	V_a	L_a	d	-1
b	V_b	e	R_b	0
c	V_c	a	b	0
d	V_d	L_d	R_d	B_d
e	V_e	L_e	R_e	B_e

Here again, we have assumed that the left link of the "master node," D_t, pointed to node D_a.

Note that the transformations just given remain valid when the node D_c in the first case, and D_d and/or D_e in the second, do not exist.

Now some contemplation will show that only these two basic situations together with their reflected images can ever occur. Actually the case where we have a node with a balance value of ±2 and a son or sons with a balance value of 0 will arise later and is covered below. The only further complication is the recomputation of B-values in all cases. The entire situation is shown below. Here the bracketed expressions show the required B-value recomputations in each case. It can easily be seen that they are, in fact, correct. The links shown with arrow heads are "new." "Old" links which are changed are "cut" with a slash. Unlabeled nodes need not exist in which case the referent links are zero. The link coming from the master node (shown as ⊙), which is the parent node of the unbalanced sub-tree, may be in R_0 or may be a left or right link of some actual node. Its logical position remains unchanged.

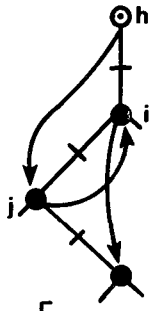
Case 1: $B_i = 2$, $B_j \geq 0$.



$B_i \leftarrow 1 - B_j$

$B_j \leftarrow B_j - 1$

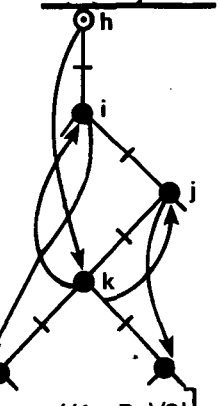
Case 2: $B_i = -2$, $B_j \leq 0$.



$B_i \leftarrow -B_j - 1$

$B_j \leftarrow 1 + B_j$

Case 3: $B_i = 2$, $B_j = -1$.

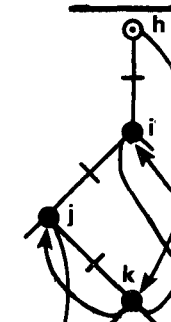


$B_i \leftarrow -\lfloor (1 + B_k)/2 \rfloor$

$B_j \leftarrow \lfloor (1 - B_k)/2 \rfloor$

$B_k \leftarrow 0$

Case 4: $B_i = -2$, $B_j = 1$.



$B_i \leftarrow \lfloor (1 - B_k)/2 \rfloor$

$B_j \leftarrow -\lfloor (1 + B_k)/2 \rfloor$

$B_k \leftarrow 0$

The transformations given above must be used to maintain the balance of our tree when updating, i.e. adding or deleting an item. Thus we give below a routine called balance to be used as a subroutine for invoking the various balance transformations. Its arguments are the area D = (V, L, R, B); a set of indices to nodes in D, namely h, i, j, and k; and a switch, d, which is to be +1 or -1. h is the index of the master node which has the node indexed by i as a son. The i-node is the left son of the h-node if d is -1, and is a right son of the h-node if d = +1. The balance, B_i, at the i-node is assumed to be +2 or -2. j is the index of the right son of the i-node if B_i = +2 and is the index of the left son of the i-node if B_i = -2. Finally if $B_iB_j < 0$ then k is the index of the left-son of the j-node if B_i = +2, and is the index of the right son of the j-node if B_i = -2. If $B_iB_j \geq 0$, then k is not used and may be an arbitrary value.

It will be seen that the routine given below merely encodes the transformations given in the diagrams above.

4.2 Various Item Addition Algorithms

Now we shall consider several versions of the add algorithm used to add an element to our tree. The required arguments are x, the element to be added; the area D; and the free-space pointer f.

In the first program a local stack, S, is used to save the search trail. In the second version a compactification of code is achieved by a suitable "renaming" imposed on D. In the third version, the search trail is kept in D itself and the appropriate pointers are restored as the backscan occurs.

Recall that R_0 is assumed to be zero initially, and that f is suitably initialized.

The first add program we consider is called basic-add. The balance program, balance, is invoked as a subroutine.

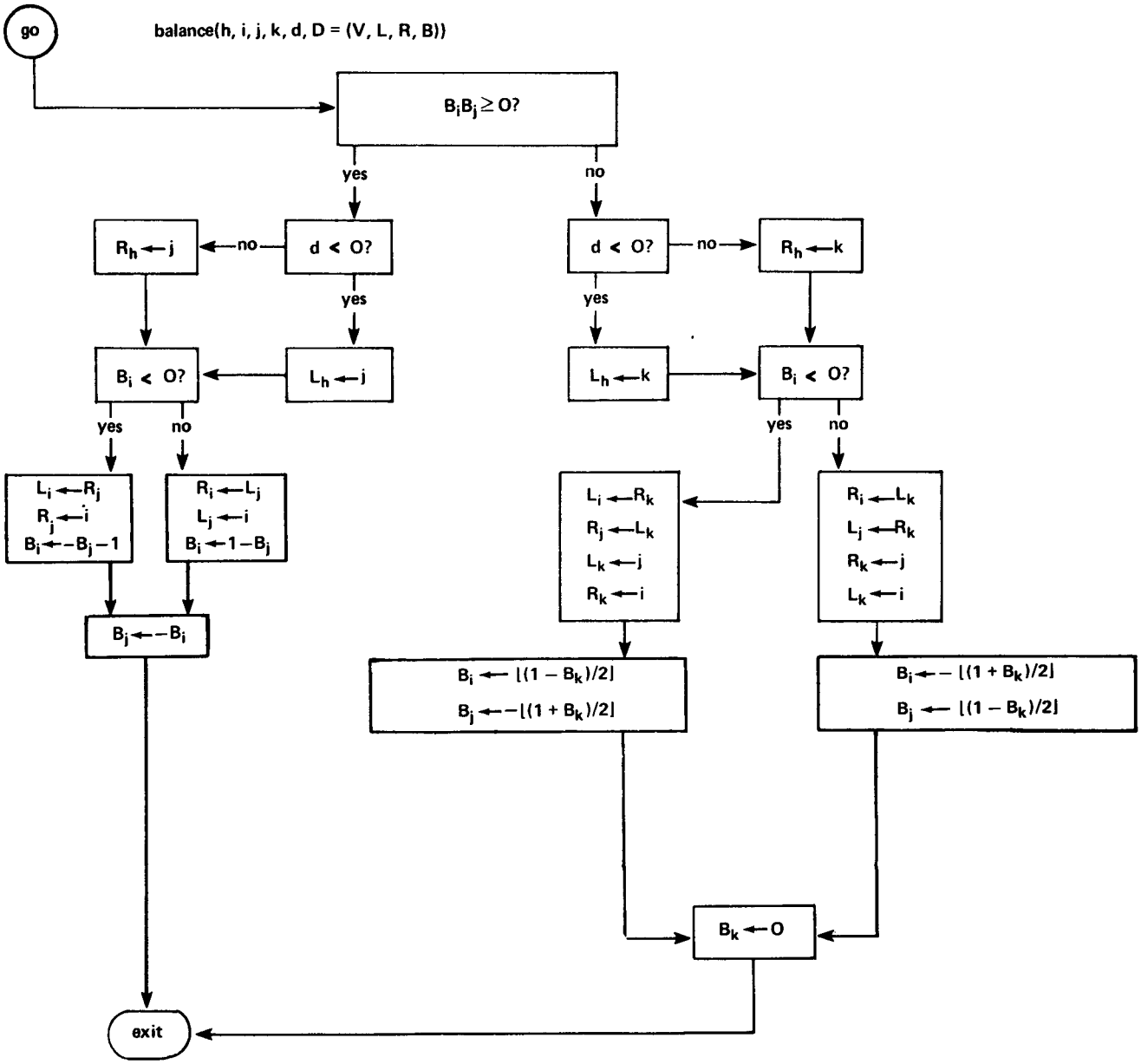
The basic-add program can be expressed more tersely.

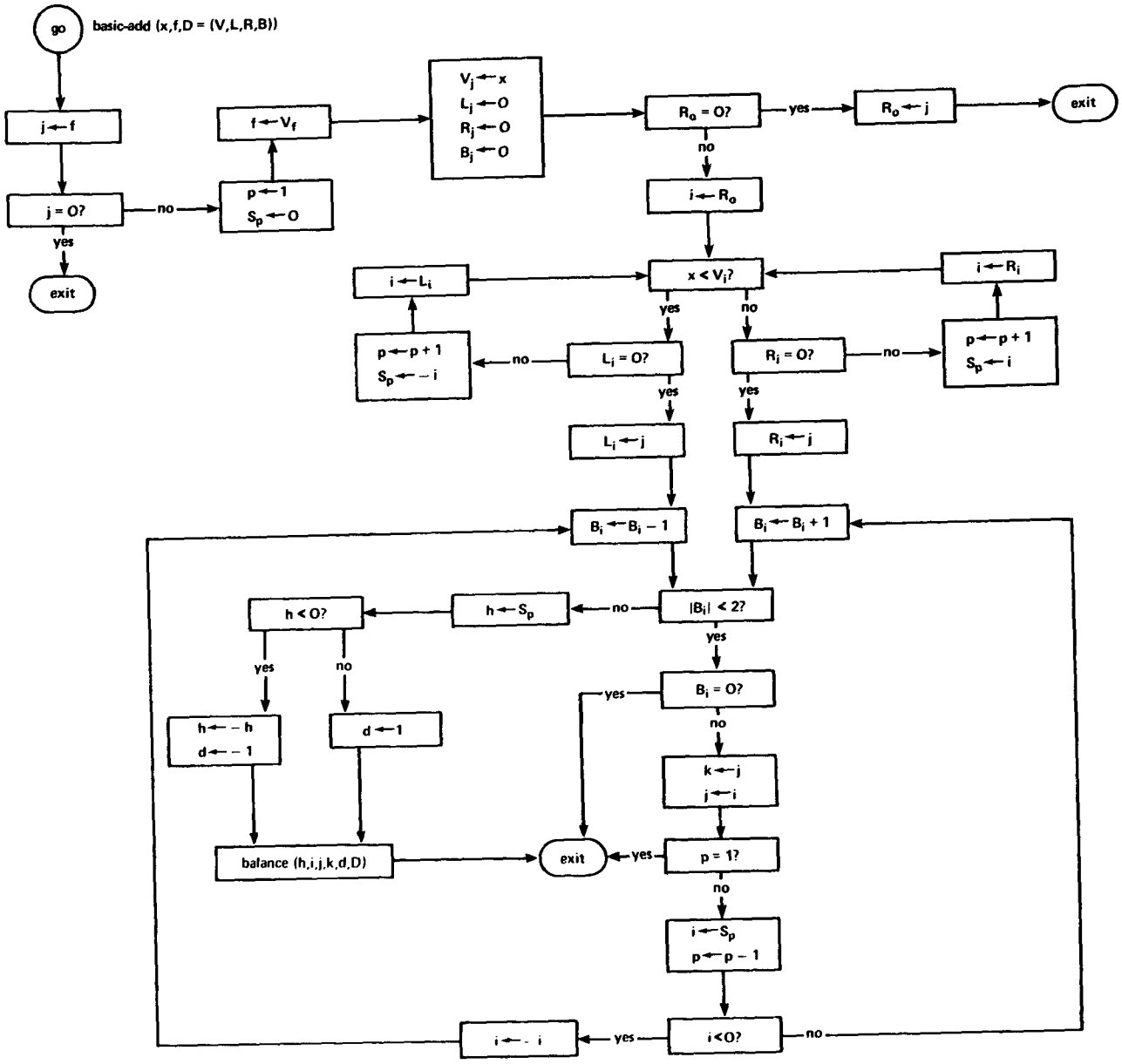
The formulation of the following algorithm, called new-basic-add, indicates some of the economies due to symmetry which one would consider in actually coding a set of programs for use in balanced tree storage and retrieval.

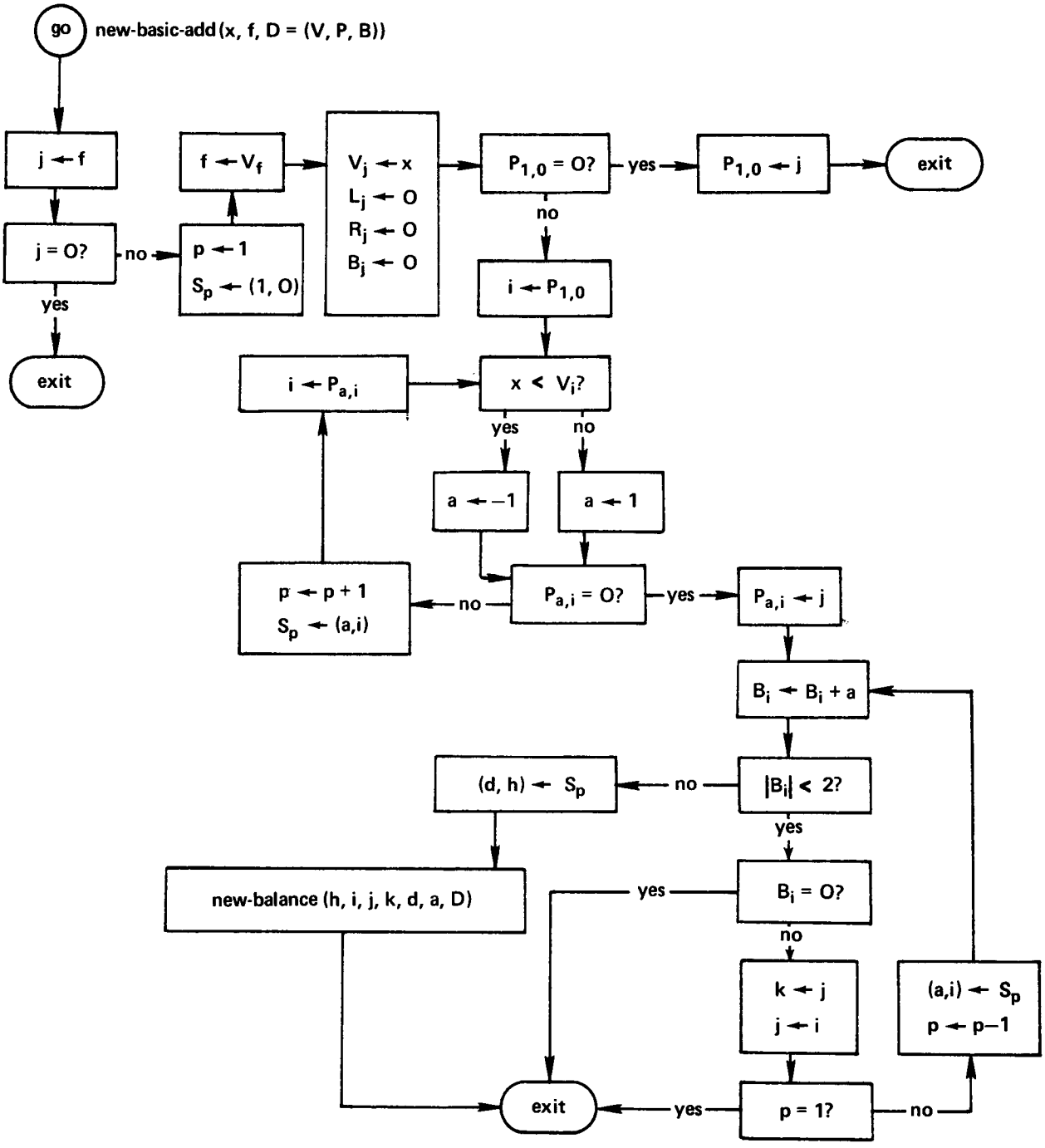
The balance routine must be modified to accept another parameter which determines which reflection of the appropriate balancing transformation is to be used. We call this modified balance routine new-balance, and it is given below following its calling program, new-basic-add. Also the stack S is used here to hold a "packed" entry or vector in each position. The underlying detailed logic should be clear.

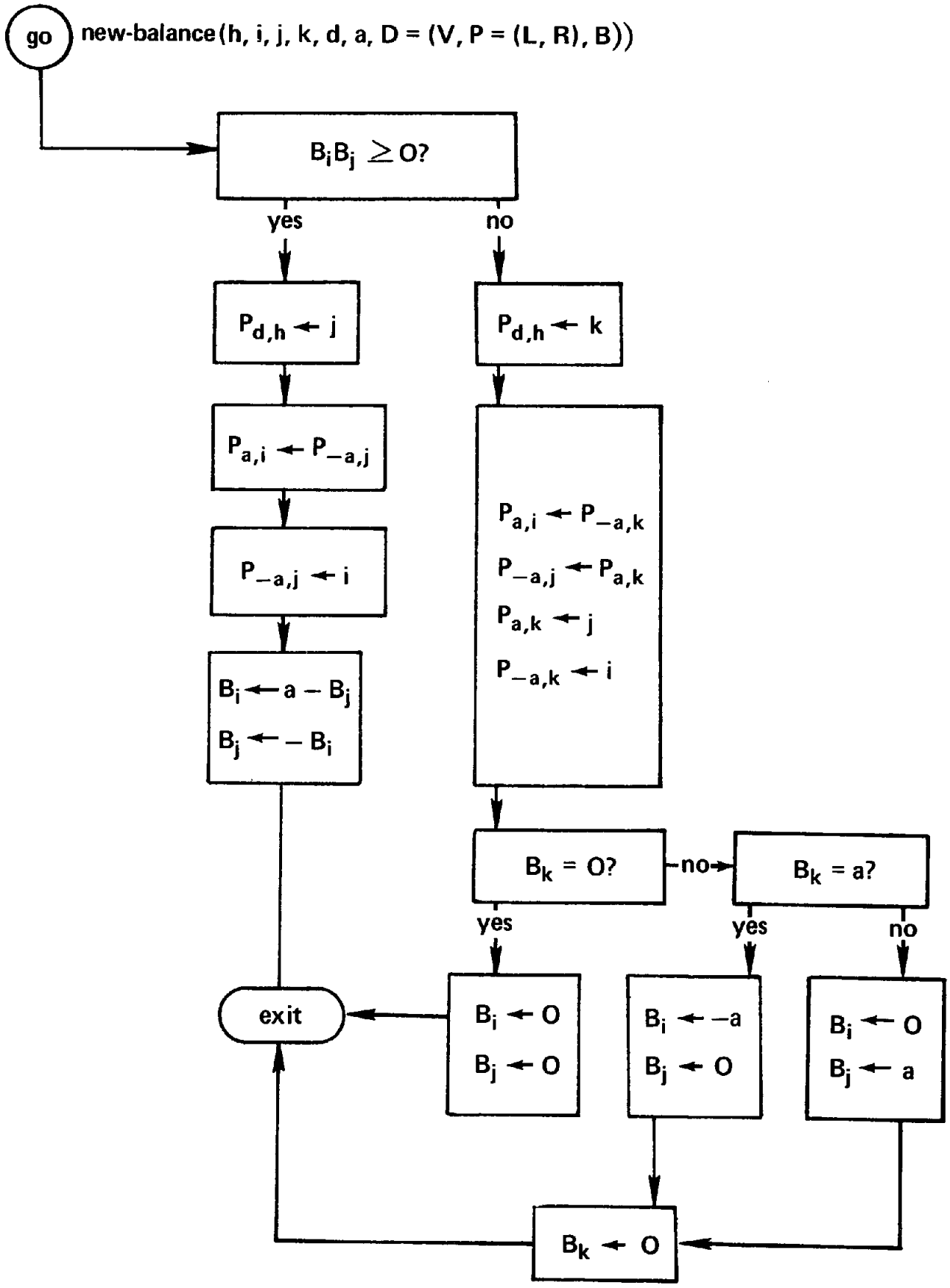
Finally, the add routine can also be given without the use of an auxiliary stack. Rather we may reverse pointers on the way "down" and re-reverse them on the way back "up" when the balance values, B_i, are being updated. The sign of a link will indicate its status and the position (L_i or R_i) of a negative link indicates whether the son of interest was a left son or right son. The "back-chain" thus defined terminates when a self-referent link is found. This program is given below as the routine called add. The balance program given originally is used as a subroutine.

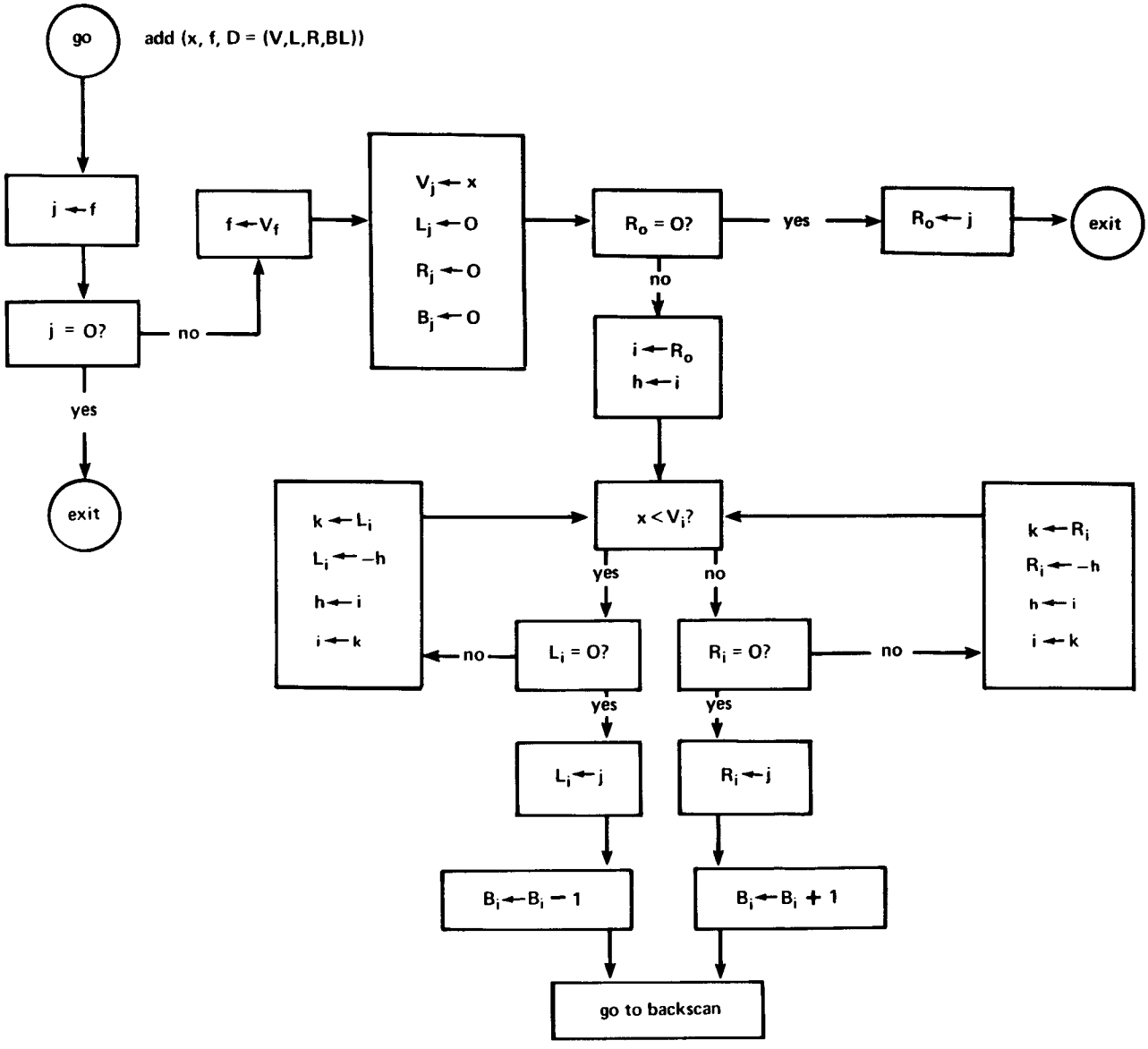
We now give the balance routine and the various addition routines.

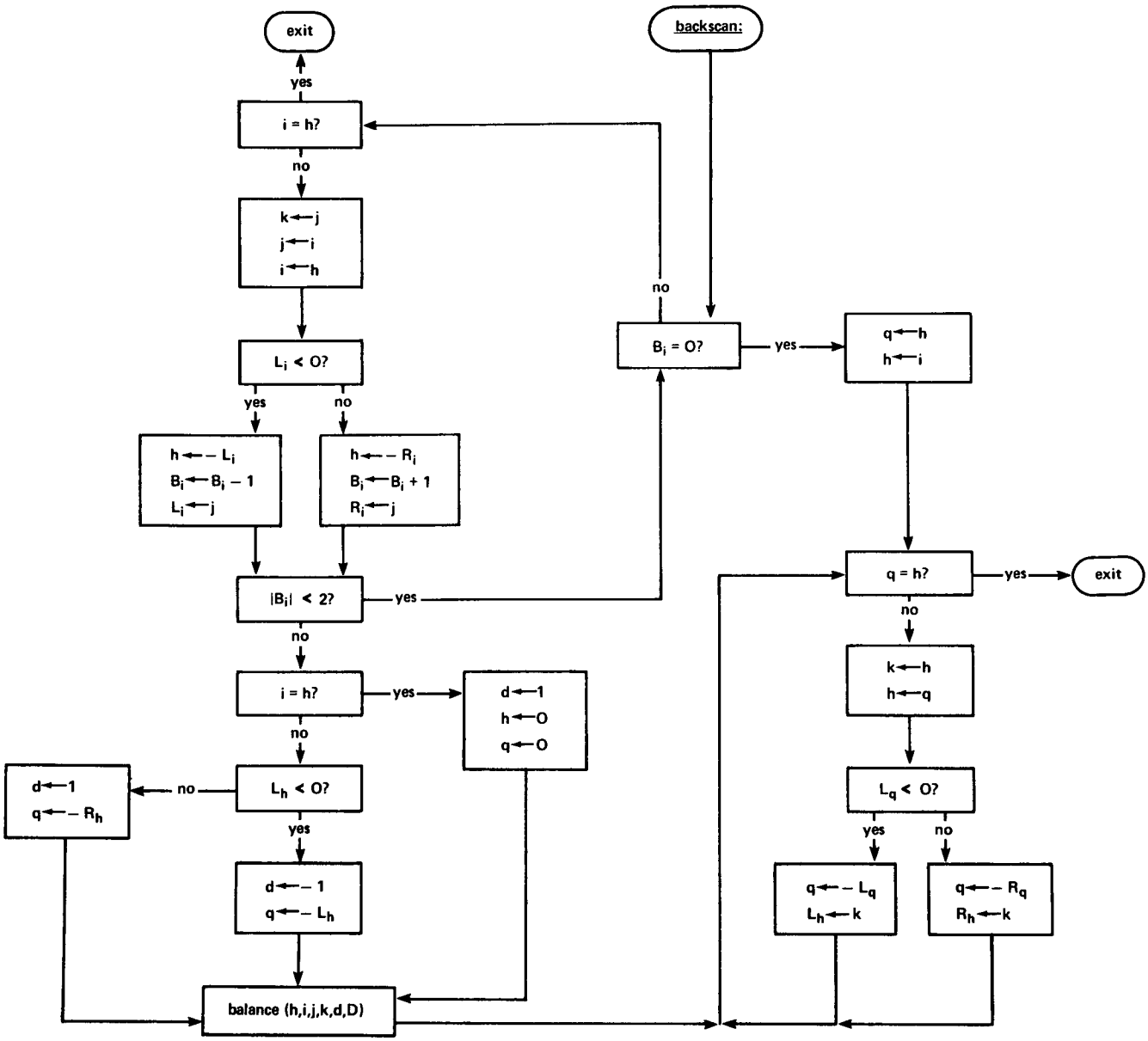

go
balance(h, i, j, k, d, D = (V, L, R, B))
$B_iB_j \geq O$?
yes
no
$R_h \leftarrow j$
no
d < O?
yes
d < O?
no
$R_h \leftarrow k$
yes
B_i < O?
$L_h \leftarrow j$
$L_h \leftarrow k$
B_i < O?
yes
no
yes
no
$L_i \leftarrow R_j$
$R_j \leftarrow i$
$B_i \leftarrow -B_j - 1$
$R_i \leftarrow L_j$
$L_j \leftarrow i$
$B_i \leftarrow 1 - B_j$
$L_i \leftarrow R_k$
$R_j \leftarrow L_k$
$L_k \leftarrow j$
$R_k \leftarrow i$
$R_i \leftarrow L_k$
$L_j \leftarrow R_k$
$R_k \leftarrow j$
$L_k \leftarrow i$
$B_j \leftarrow -B_i$
$B_i \leftarrow \lfloor(1 - B_k)/2\rfloor$
$B_j \leftarrow -\lfloor(1 + B_k)/2\rfloor$
$B_i \leftarrow -\lfloor(1 + B_k)/2\rfloor$
$B_j \leftarrow \lfloor(1 - B_k)/2\rfloor$
$B_k \leftarrow O$
exit


go
basic-add (x,f,D = (V,L,R,B))
j ← f
j = 0?
yes
exit
no
p ← 1
S_p ← 0
f ← V_f
V_j ← x
L_j ← 0
R_j ← 0
B_j ← 0
R_o = 0?
yes
R_o ← j
exit
no
i ← R_o
x < V_i?
yes
no
i ← L_i
p ← p + 1
S_p ← − i
no
L_j = 0?
yes
L_i ← j
B_i ← B_i − 1
i ← R_i
p ← p + 1
S_p ← i
no
R_j = 0?
yes
R_i ← j
B_i ← B_i + 1
|B_i| < 2?
yes
no
h ← S_p
h < 0?
yes
no
h ← − h
d ← − 1
d ← 1
balance (h,i,j,k,d,D)
B_i = 0?
yes
no
k ← j
j ← i
exit
p = 1?
yes
no
i ← S_p
p ← p − 1
i ← − i
i<0?
yes
no


go
new-basic-add(x, f, D = (V, P, B))
j ← f
j = O?
no
yes
exit
p ← 1
S_p ← (1, O)
f ← V_f
V_j ← x
L_j ← O
R_j ← O
B_j ← O
$P_{1,0}$ = O?
yes
$P_{1,0}$ ← j
exit
no
i ← $P_{1,0}$
i ← $P_{a,i}$
x < V_i?
yes
no
a ← −1
a ← 1
p ← p + 1
S_p ← (a,i)
no
$P_{a,i}$ = O?
yes
$P_{a,i}$ ← j
B_i ← B_i + a
(d, h) ← S_p
no
$|B_i|$ < 2?
yes
new-balance (h, i, j, k, d, a, D)
yes
B_i = O?
no
k ← j
j ← i
(a,i) ← S_p
p ← p−1
exit
yes
p = 1?
no


go
new-balance(h, i, j, k, d, a, D = (V, P = (L, R), B))
$B_iB_j \geq 0$?
yes
no
$P_{d,h} \leftarrow j$
$P_{d,h} \leftarrow k$
$P_{a,i} \leftarrow P_{-a,j}$
$P_{-a,j} \leftarrow i$
$B_i \leftarrow a - B_j$
$B_j \leftarrow -B_i$
$P_{a,i} \leftarrow P_{-a,k}$
$P_{-a,j} \leftarrow P_{a,k}$
$P_{a,k} \leftarrow j$
$P_{-a,k} \leftarrow i$
$B_k = 0$?
no
$B_k = a$?
yes
yes
no
exit
$B_i \leftarrow 0$
$B_j \leftarrow 0$
$B_i \leftarrow -a$
$B_j \leftarrow 0$
$B_i \leftarrow 0$
$B_j \leftarrow a$
$B_k \leftarrow 0$


go
add (x, f, D = (V,L,R,BL))
j ← f
j = O?
no
yes
exit
f ← V_f
V_j ← x
L_j ← O
R_j ← O
B_j ← O
R_o = O?
yes
R_o ← j
exit
no
i ← R_o
h ← i
x < V_i?
yes
no
k ← L_i
L_i ← −h
h ← i
i ← k
no
L_i = O?
yes
L_i ← j
B_i ← B_i − 1
R_i = O?
yes
R_i ← j
B_i ← B_i + 1
no
k ← R_i
R_i ← −h
h ← i
i ← k
go to backscan


exit
yes
i = h?
no
k←j
j←i
i←h
Li < O?
yes
no
h←– Li
Bi←Bi – 1
Li←j
h←– Ri
Bi←Bi + 1
Ri←j
|Bi| < 2?
yes
no
i = h?
yes
d←1
h←O
q←O
no
d←1
q←– Rh
no
Lh < O?
yes
d←– 1
q←– Lh
balance (h,i,j,k,d,D)
backscan:
no
Bi = O?
yes
q←h
h←i
q = h?
yes
exit
no
k←h
h←q
Lq < O?
yes
no
q←– Lq
Lh←k
q←– Rq
Rh←k

5. The Deletion Procedure

The deletion algorithm for our balanced tree storage scheme is somewhat difficult when considered in detail although it is very similar in spirit to the add procedures discussed above.

The deletion process proceeds by searching for the node which is to be deleted, given its key value. The path taken during this search must be saved for possible later retracing. If no matching node is found, we are done. Otherwise, the node to be deleted is removed from our tree by suitably relinking various surrounding nodes, and the space in D used for the deleted node is returned to the free-space list.

The required relinking may result in unbalancing our tree at a node of the disturbed region. Thus this situation must be tested for and, if present, may be corrected by an application of the balance routine given earlier. Unlike the add procedure, the deletion procedure may perform several balance operations; since, after relinking, we must retrace our steps back to the root, adjusting the balance values of each node of the back-trail as we go. Whenever the sub-tree defined by the current node of the back-trail is unbalanced, we must pause and apply the balancing algorithm before continuing. It is possible that a balancing operation will be required for every node in the back-trail, although this is not normally the case, for when a balance value becomes +1 or -1, no further changes will occur.

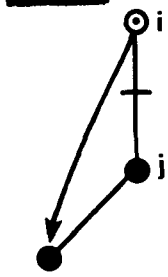
5.1 Deletion Relinking

We may now discuss in more detail the relinking process required to remove a node from our tree, whereupon we may further consider the "rewinding and rebalancing" logic.

It will suffice to give several general diagrams showing the transformations required. It can be seen that the situations shown below are exhaustive. As before, in each of the diagrams below, ⊙ represents the master node (in this case the node which is the father of the node to be deleted). If the root node is being deleted, the master node consists only of the R_0 "entry" link. Thus, in each case, the node labeled j is to be removed. Also, we understand that in each diagram an unlabeled node need not exist, in which case the referent link shown is zero. The transformations given remain valid under this interpretation. Moreover, as before, old links are shown by simple lines while new links have arrow heads shown. Finally, we note that it is easy to justify the balance value recomputations shown. The balance values of nodes above node i may not be correct, but this is taken care of in the course of the deletion process.

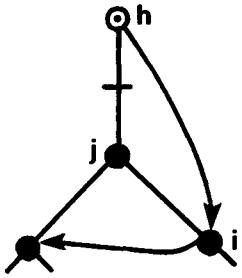
In general, the transformations given merely "replace" the deleted node with a more-or-less close neighbor such that the "little-to-the-left, big-to-the-right" structure of our tree is preserved. Thus, we have the following situations. Note that case 3 is just a special case of case 4, wherein t=i.

Case 1: $R_j = 0$.



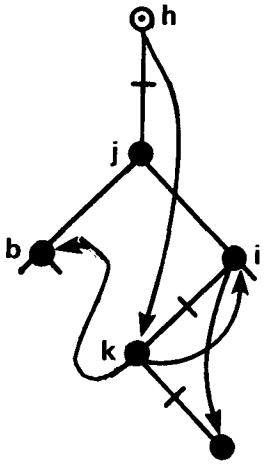
$[B_i \leftarrow B_i + 1$ if j is the left son of i, otherwise, $B_i \leftarrow B_i - 1.]$

Case 2: $L_i = 0$.



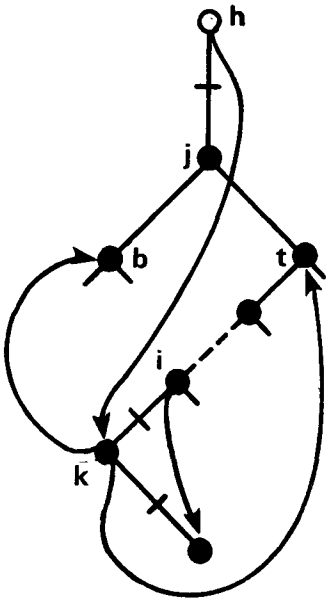
$[B_i \leftarrow B_j - 1]$

Case 3: $L_i = k$.



$$\begin{bmatrix} B_i \leftarrow B_i + 1 \\ B_k \leftarrow B_j \end{bmatrix}$$

Case 4: $L_t \neq L_i = k$.



$$\begin{bmatrix} B_i \leftarrow B_i + 1 \\ B_k \leftarrow B_j \end{bmatrix}$$

5.2 Various Deletion Algorithms

After the node to be deleted has been removed by means of the appropriate relinking transformation, we then consider the back-trail from the node labled i back to the root of our tree.

It will be seen that after relinking, the balance at node i is correct (and possibly ± 2) and the balance at each node which is a parent of node i may be incorrect by +1 or -1. Thus we first check the balance at node i and balance the subtree with root node i (which we now call the current sub-tree) if required. This may leave the current sub-tree with a height which is one less than the corresponding sub-tree in the same position (this is, with root node i) before we began the deletion process. This is the case if the current sub-tree before any required rebalancing is done, has $B_i=0$, or $B_i=2$ and $B_{R_i}=\pm 1$, or $B_i=-2$ and $B_{L_i}=\pm 1$. In these circumstances, after any necessary rebalancing, we see that we have "shortened the long side" of the current sub-tree and hence decreased its total height. Otherwise, we have $B_i=\pm 1$, or $B_i=2$ and $B_{R_i}=0$, or $B_i=-2$ and $B_{L_i}=0$, and after any necessary rebalancing the resulting current sub-tree will not have decreased in height. These two situations can be distinguished by the fact that after any necessary rebalancing is done, we have the balance value of the current sub-tree equal to zero if the height has decreased and ± 1 otherwise. Thus in case we obtain a balance value of zero, the resulting sub-tree is shorter by one than originally and hence, if the balance at its parent node is dependent upon the current sub-tree's previous height, we may find it necessary to rebalance the sub-tree whose root is the just-higher node in our back-trail. In particular, this additional rebalancing will be required only when our sub-tree (with root node i, unless balanced) has grown shorter in height by one and also, either its parent node has a balance value of +1 and our current sub-tree is to the left of the parent node, or its parent node has a balance value of -1 and our current sub-tree is to the right of the parent node.

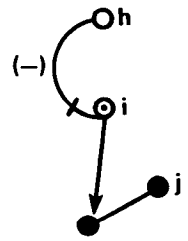
We may now see how we must proceed up the back-trail. If we enter a node from its left son, we increase the balance at the node by one, while if we enter a node from its right son, we decrease the balance at the node by one. If we thereby obtain a new balance value of +1 or -1, we are done and no further balance adjustments are required. This is because the height of our current sub-tree has not decreased. If, on the other hand, rebalancing is required, it is invoked. Then if the resulting sub-tree has a balance value of +1 or -1, we are done, since again the height of the current sub-tree has not changed. Otherwise, we must continue to the next node in the back-trail.

The process described above is given precisely in the routine below called basic-delete. The arguments are x, the key of the item to be deleted; f, the free-space pointer; and D, our basic tree area. The local stack, S, is used to save the back-trail as in the basic-add routine given earlier. Finally, the balance routine given earlier is used below.

We also give below the deletion algorithm which corresponds to the add procedure given earlier. This routine, called delete, takes the same arguments as basic-delete, but no local stack, S, is needed. Rather, the back-trail is kept by temporarily reversing pointers during the initial search.

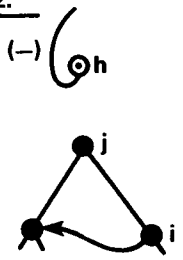
In this routine the logic of the relinking transformations given earlier is changed to correctly remove a node from our tree with the back-trail being maintained. The essential changes are shown in the following set of diagrams, where a link labeled (-) is understood to be part of our back-trail; otherwise, all our conventions are as before. Again case 3 is just a special case of case 4.

Case 1.



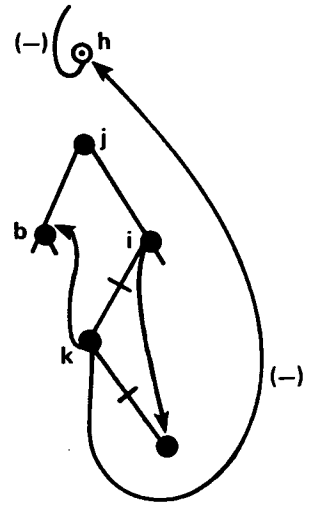
$[B_i \leftarrow B_i + 1$, if j is the left son of i.
$B_i \leftarrow B_i - 1$, otherwise.$]$

Case 2.



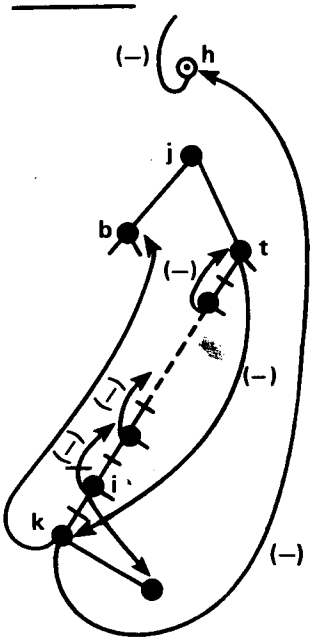
$[B_i \leftarrow B_j - 1]$

Case 3.



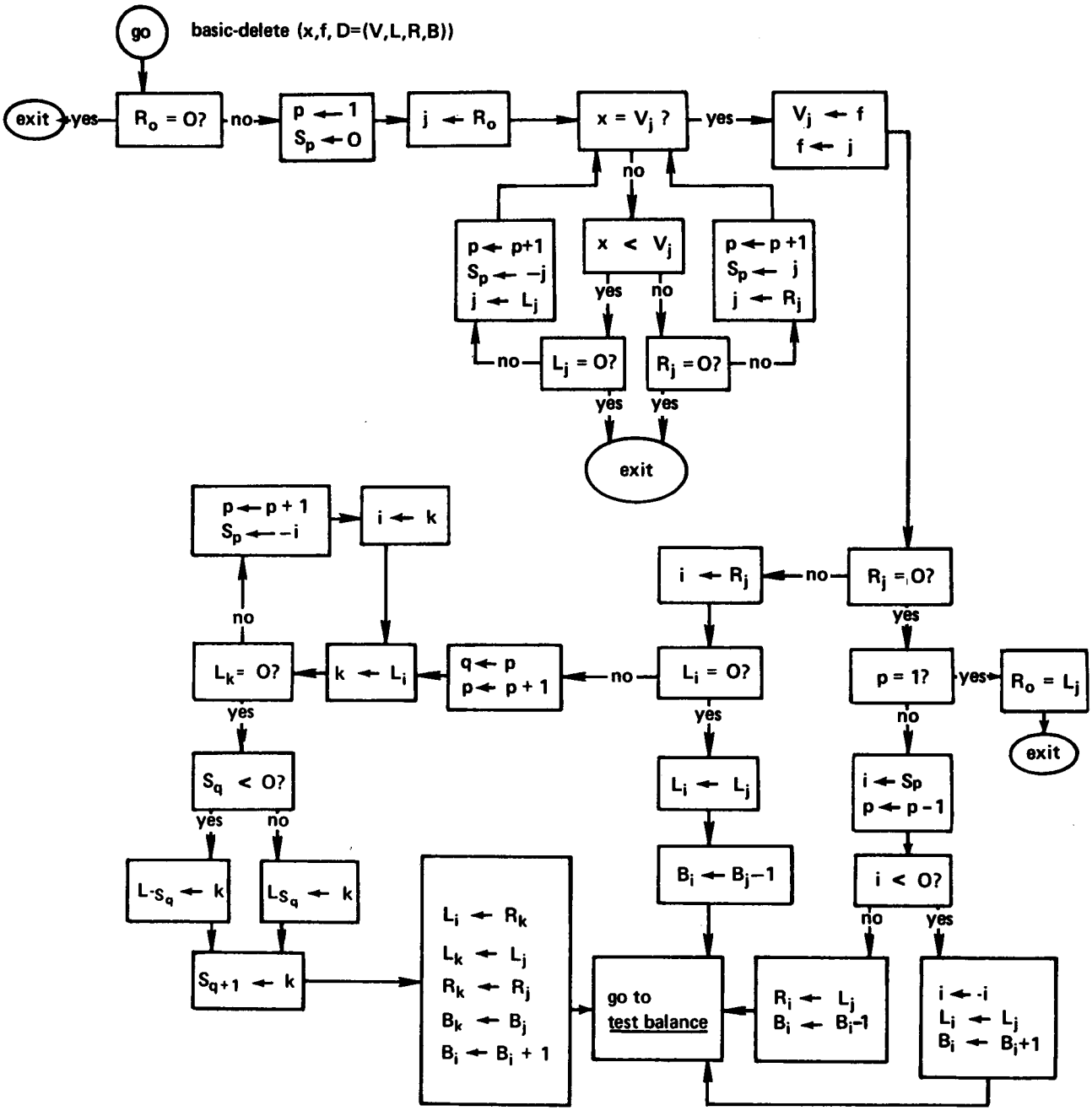
$[B_i \leftarrow B_i + 1$
$B_k \leftarrow B_j]$

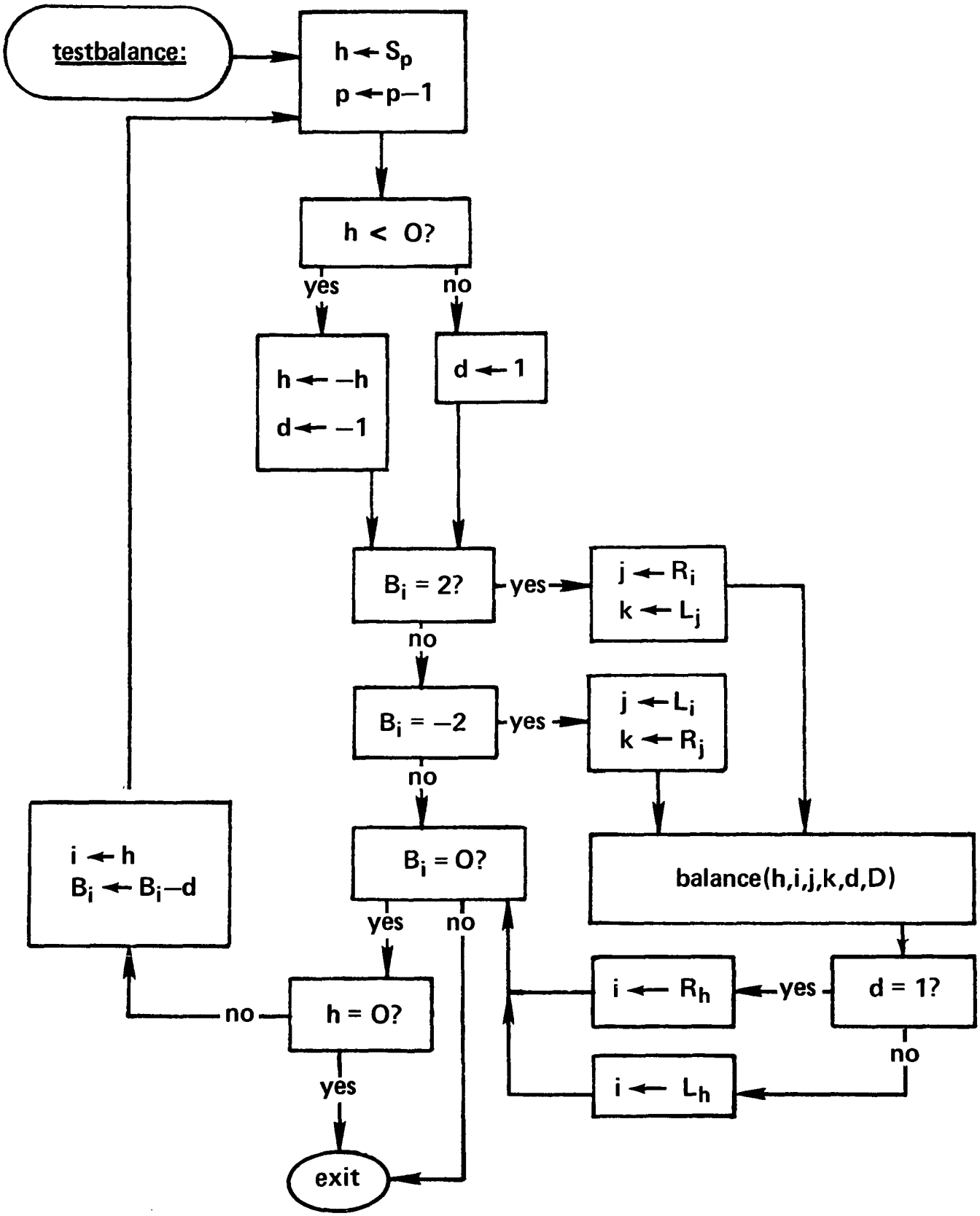
Case 4.

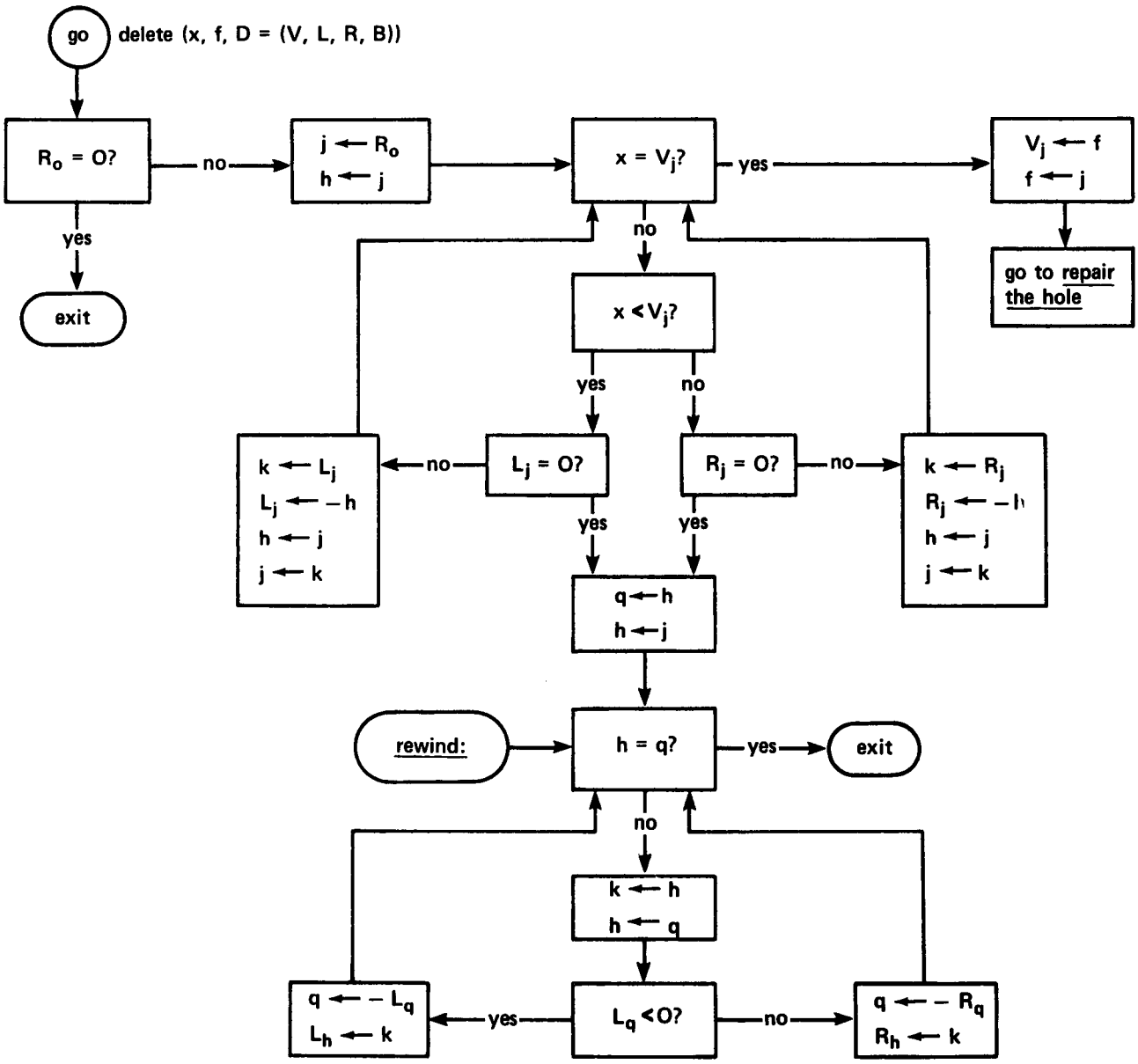


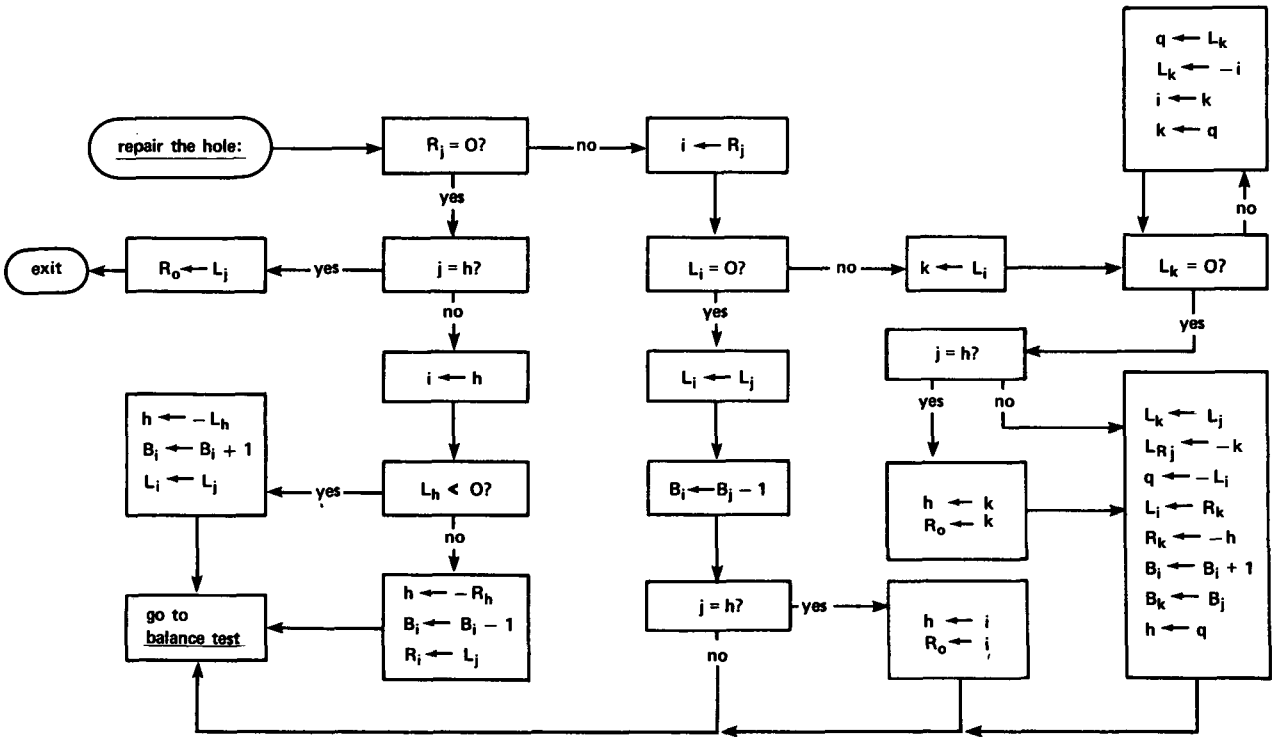
$[B_i \leftarrow B_i + 1$
$B_k \leftarrow B_j]$

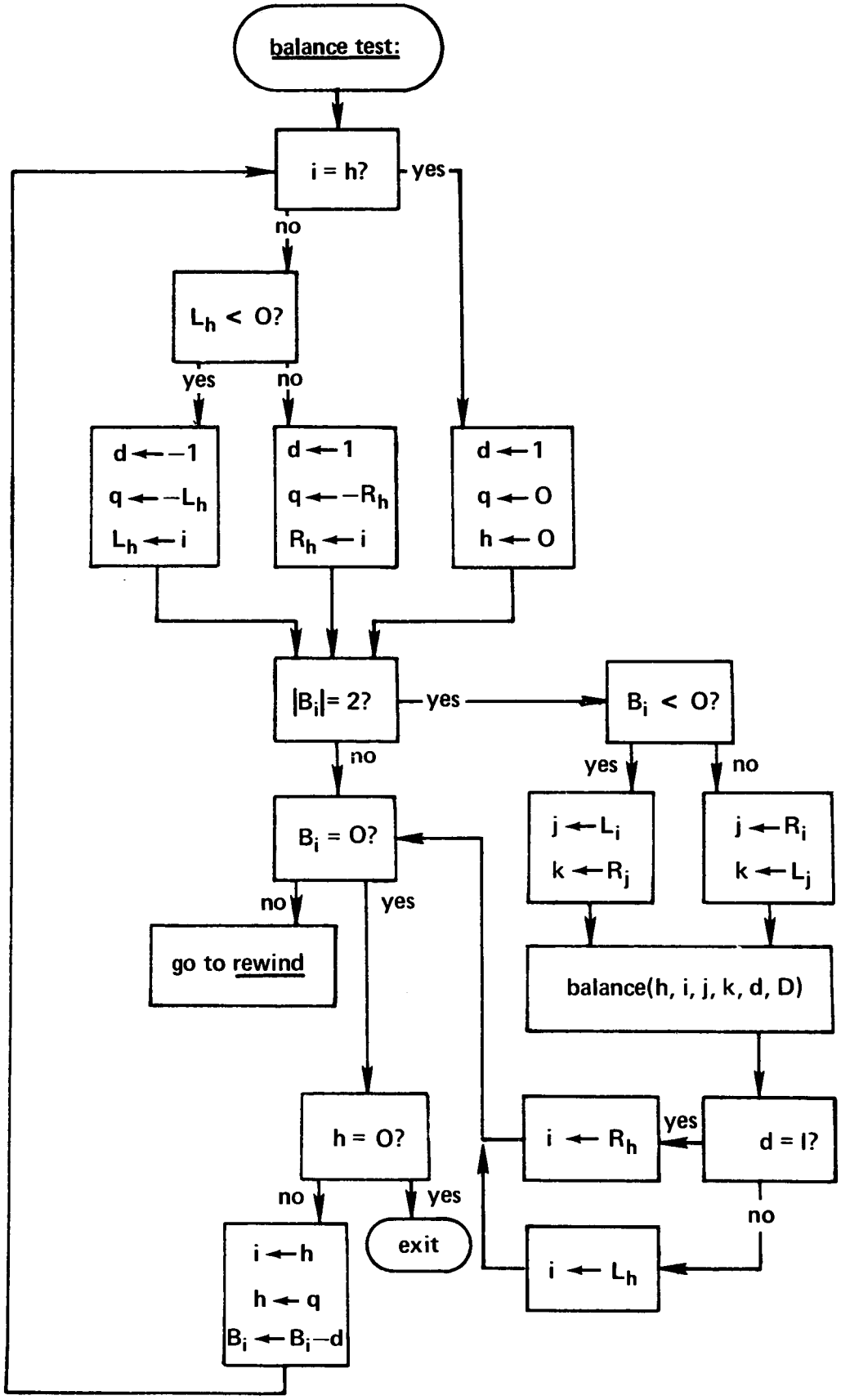
We now give the various deletion routines. basic-delete is followed by delete which uses the transformations given just above.


go
basic-delete (x,f, D=(V,L,R,B))
exit
yes
R_o = O?
no
p ← 1
S_p ← O
j ← R_o
x = V_j ?
yes
V_j ← f
f ← j
no
p ← p+1
S_p ← −j
j ← L_j
x < V_j
yes
no
p ← p+1
S_p ← j
j ← R_j
no
L_j = O?
R_j = O?
no
yes
yes
exit
p ← p + 1
S_p ← −i
i ← k
i ← R_j
no
R_j = O?
yes
no
L_k = O?
k ← L_i
q ← p
p ← p + 1
no
L_i = O?
p = 1?
yes
R_o = L_j
exit
yes
yes
no
S_q < O?
L_i ← L_j
i ← S_p
p ← p − 1
yes
no
L_{-S_q} ← k
L_{S_q} ← k
B_i ← B_j−1
i < O?
no
yes
S_{q+1} ← k
L_i ← R_k
L_k ← L_j
R_k ← R_j
B_k ← B_j
B_i ← B_i + 1
go to
test balance
R_i ← L_j
B_i ← B_i−1
i ← -i
L_i ← L_j
B_i ← B_i+1


testbalance:
h ← S_p
p ← p−1
h < O?
yes
no
h ← −h
d ← −1
d ← 1
B_i = 2?
yes
$j \leftarrow R_i$
$k \leftarrow L_j$
no
B_i = −2
yes
$j \leftarrow L_i$
$k \leftarrow R_j$
no
B_i = O?
yes
no
i ← h
$B_i \leftarrow B_i - d$
h = O?
no
yes
exit
balance(h,i,j,k,d,D)
d = 1?
yes
$i \leftarrow R_h$
no
$i \leftarrow L_h$


go
delete (x, f, D = (V, L, R, B))
R_o = O?
no
$j \leftarrow R_o$
$h \leftarrow j$
x = V_j?
yes
$V_j \leftarrow f$
$f \leftarrow j$
go to repair the hole
yes
exit
no
x < V_j?
yes
no
$k \leftarrow L_j$
$L_j \leftarrow -h$
$h \leftarrow j$
$j \leftarrow k$
no
L_j = O?
R_j = O?
no
$k \leftarrow R_j$
$R_j \leftarrow -h$
$h \leftarrow j$
$j \leftarrow k$
yes
yes
$q \leftarrow h$
$h \leftarrow j$
rewind:
h = q?
yes
exit
no
$k \leftarrow h$
$h \leftarrow q$
$q \leftarrow -L_q$
$L_h \leftarrow k$
yes
L_q < O?
no
$q \leftarrow -R_q$
$R_h \leftarrow k$


repair the hole:
$R_j = O$?
no
$i \leftarrow R_j$
yes
$L_i = O$?
no
$k \leftarrow L_i$
$L_k = O$?
no
yes
$q \leftarrow L_k$
$L_k \leftarrow -i$
$i \leftarrow k$
$k \leftarrow q$
exit
$R_o \leftarrow L_j$
yes
$j = h$?
yes
$L_i \leftarrow L_j$
$j = h$?
yes
no
$L_k \leftarrow L_j$
$L_{Rj} \leftarrow -k$
$q \leftarrow -L_i$
$L_i \leftarrow R_k$
$R_k \leftarrow -h$
$B_i \leftarrow B_i + 1$
$B_k \leftarrow B_j$
$h \leftarrow q$
no
$i \leftarrow h$
$h \leftarrow -L_h$
$B_i \leftarrow B_i + 1$
$L_i \leftarrow L_j$
yes
$L_h < O$?
$B_i \leftarrow B_j - 1$
$h \leftarrow k$
$R_o \leftarrow k$
no
go to
balance test
$h \leftarrow -R_h$
$B_i \leftarrow B_i - 1$
$R_i \leftarrow L_j$
$j = h$?
yes
$h \leftarrow i$
$R_o \leftarrow i$
no


balance test:
i = h?
yes
no
L_h < O?
yes
no
d ← −1
q ← $-L_h$
L_h ← i
d ← 1
q ← $-R_h$
R_h ← i
d ← 1
q ← O
h ← O
$|B_i|$ = 2?
yes
no
B_i < O?
yes
no
B_i = O?
no
yes
go to rewind
j ← L_i
k ← R_j
j ← R_i
k ← L_j
balance(h, i, j, k, d, D)
h = O?
no
yes
exit
i ← h
h ← q
B_i ← B_i−d
i ← R_h
yes
d = l?
no
i ← L_h

6. A Cost Analysis

Here we obtain an upper bound for C, the number of 3-way comparisons required to search for an item and retrieve it (if present) from a balanced tree of n nodes. Given a balanced tree of items and a key, each comparison either matches the key being searched for or allows us to make the next comparison at the next higher level in our tree. Thus, the desired upper bound is given as the maximum number of levels which can be found in some balanced tree of n nodes.

We begin by determining the minimum number of nodes, T_k, which may occur in a balanced tree with k levels.

Clearly, we have $T_0 = 0$ and $T_1 = 1$. Now we may see that $T_{k+1} = T_k + T_{k-1} + 1$ for $k \geq 1$, since we schematically have

$$\overline{T}_{k+1} = \quad \overset{\bullet}{\underset{\overline{T}_k \qquad \overline{T}_{k-1}}{}} \qquad (1)$$

where here we understand that $\overline{T}_k$ represents a balanced tree of k levels with T_k nodes, that is, a minimal balanced tree of k levels. Then disregarding the reflections which cause $\overline{T}_k$ to fail to be unique, we see that (1) graphically constructs a minimal balanced tree of k+1 levels from given minimal balanced trees of k and k-1 levels plus one additional node and thus the relation in integers, $T_{k+1} = T_k + T_{k-1} + 1$, results by counting.

Thus we have:

Number of Levels, k	Number of Nodes in Minimal Balanced Tree, T_k
0	0
1	1
2	2
3	4
4	7
5	12
6	20
7	33
8	54
9	88
10	143
.	.
.	.
.	.

Now it is easy to show by induction that $T_k = F_{k+2} - 1$ where F_i is the i^{th} Fibonacci number, that is, $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.

Another proof of this fact may be given as follows.

Let G(z) be the generating function for the T_k values. That is,

$$G(z) = T_0 + T_1 z + T_2 z^2 + T_3 z^3 + \ldots \quad .$$

Then, since $T_{k+1} = T_k + T_{k-1} + 1$ for $k \geq 1$ and also,

$\frac{1}{1-z} = 1 + z + z^2 + \ldots$, we have

$$G(z) - zG(z) - z^2 G(z) - \frac{z^2}{1-z} = z \; . \qquad (2)$$

Then, by partial fraction decomposition we have

$$G(z) = \frac{z^{-1}}{1-z-z^2} + \frac{-z^{-1}}{1-z} \quad .$$

But now we know (see [7], p. 82) that $\frac{z}{1-z-z^2}$ is the generating function for the Fibonacci numbers; that is,

$$\frac{z}{1-z-z^2} = F_0 + F_1 z + F_2 z^2 + F_3 z^3 + \ldots \quad .$$

Thus, since $F_0 = 0$, $F_1 = 1$, we use the expansion of $\frac{-z^{-1}}{1-z}$ to obtain

$$G(z) = (F_2 - 1) + (F_3 - 1)z + \ldots \quad .$$

But then, equating coefficients, we obtain $T_k = F_{k+2} - 1$ for $k \geq 0$.

Now we may proceed.

Given n, a number of nodes organized in a balanced tree, Q, then if k is such that

$$F_{k+2} - 1 \leq n < F_{k+3} - 1 \qquad (3)$$

then Q has at most k levels and hence, at most k compares are needed to search Q.

Now let us solve (3) for k in terms of n.

Let m = k + 2 for brevity.

Then we have

$$F_m \leq n + 1 < F_{m+1} \; . \qquad (4)$$

But now we know (see [7], p. 82) that

$$F_m = \frac{1}{\sqrt{5}} (\phi^m - \hat{\phi}^m) \text{ for } m \geq 0, \qquad (5)$$

where

$$\phi = \frac{1}{2} + \frac{\sqrt{5}}{2} = 1.61803 \ldots$$

and

$$\hat{\phi} = \frac{1}{2} - \frac{\sqrt{5}}{2} = -.61803 \ldots \quad .$$

Then from (4) and (5) with some manipulation we have

$$\phi^m \leq (n + 1)\sqrt{5} - \hat{\phi} < \phi^{m+1} \quad . \qquad (6)$$

Hence,

$$m \leq \ln_\phi ((n + 1)\sqrt{5} - \hat{\phi}) < m + 1$$

or,

$$m = \left\lfloor \ln_\phi ((n + 1)\sqrt{5} - \hat{\phi}) \right\rfloor \quad .$$

Thus, since $m = k + 2$,

$$k = \left\lfloor \ln_\phi ((n + 1)\sqrt{5} - \hat{\phi}) \right\rfloor - 2 \quad .$$

Now as stated above, the number of compares, C, required to search any balanced tree of n or fewer nodes is such that

$$C \leq \left\lfloor \ln_\phi ((n + 1)\sqrt{5} - \hat{\phi}) \right\rfloor - 2 \quad .$$

We may approximate our derived upper bound for C as

$$\left\lfloor 2.08 \ln(2.2n + 3) \right\rfloor - 2 \quad .$$

We may also note that the shortest search path in a balanced tree of k levels has at least $\left\lfloor \frac{k + 1}{2} \right\rfloor$ nodes. This can be taken as a lower bound on the number of compares required to terminate an unsuccessful search.

7. Concluding Remarks

Several interesting problems concerning balanced tree storage and retrieval remain to be solved. One problem is the computation of the exact mean and variance of C, the number of 3-way comparisons required to search for an item. Another problem is to compute the expected number of balance operations required per insertion and per deletion taken over the construction and equilibrium existence of a balanced tree which stores items chosen from a random set of items.

The balanced tree storage and retrieval scheme has several useful features. One feature is that it is easy to retrieve the least or greatest item currently stored without needing to know the appropriate key-value. It is also easy to pass over the stored items in order by their key-values if desired.

Moreover, although hash table storage and retrieval algorithms are superior to the balanced tree scheme for symbol tables and the like, it is difficult to develop a general storage and retrieval system based on hashing methods. It is easier to program the balanced tree algorithms to deal with user-specified item formats and key-values, thus a general filing system may be developed based on the balanced tree storage and retrieval algorithms.

The nearest competitor to the balanced tree scheme as a general filing system is a sequentially organized filing scheme with auxillary tables of indices, of which the ISAM (indexed sequential access method) facilities in the S/360 operating system [5] is an example. Given a reasonable amount of insertion-deletion activity, the number of items accessed during a search may be reasonably similar in the two methods, and the balanced tree scheme uses space more efficiently. However, the fact that ISAM accesses items mostly in the same cylinder while the balanced tree scheme may not is a serious deficiency. It is possible of course that a way of building the balanced tree can be found which takes into account the difficulties involved with using movable head discs. In any event for truly random access storage mediums the problems associated with varying access time disappear and the balanced tree scheme is useful in such circumstances.

REFERENCES

(1) ADEL'SON-VEL'SKIJ, G.M.; LANDIS, Y.M. An algorithm for the organization of information. Doklady Akademii Nauk USSR, vol. 16, no. 2, p. 263:266, Moscow, USSR; also available in translation as U.S. Dept. of Commerce OTS, JPRS 17,137; and as NASA Document N63-11777.

(2) BOOTH, A.D.; COLIN, H.J.T. On the efficiency of a new method of dictionary construction. Information and Control, vol. 3, p. 334:341, December 1960.

(3) FERGUSON, DAVID E. Balanced tree searching. internal document, Programmatics Inc., January 1968.

(4) FOSTER, C.C. Information storage and retrieval using AVL trees. Proceedings of the 20th National ACM Conference at Cleveland, 1965, p. 192:205.

(5) GHOSH, S.P.; SENKO, M.E. File organization: on the selection of random access index points for sequential files. JACM, vol. 16, no. 4, p. 569:579, October 1969.

(6) HIBBARD, THOMAS. Some combinatorial properties of certain trees. JACM, vol. 9, no. 1, p. 13:28, January 1962.

(7) KNUTH, DONALD E. The Art of Computer Programming, vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass., 1968.

(8) KNUTH, DONALD E. Optimum binary search trees. Stanford Computer Science Dept. Technical Report CS 149, Stanford University, January 1970.

(9) WINDLEY, P.R. Trees, forests, and rearranging. Computer Journal, vol. 3, no. 2, p. 84:88, July 1960.