

SEARCH TREE DATA STRUCTURES AND  
THEIR APPLICATIONS

By

YICK-KWAN CHEN

"

Bachelor of Science

National Tsing Hua University

Taiwan, Republic of China

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  
May, 1987

Thesis  
1987  
CS185  
Cop. 2



SEARCH TREE DATA STRUCTURES AND  
THEIR APPLICATIONS

Thesis Approved:

*Ronald W. Fisher*

Thesis Adviser

*Donald W. Grace*

*J. P. Chandler*

*Norman N. Durham*

Dean of the Graduate College

## PREFACE

This study concerns the discussion of search tree data structures and their applications. The thesis presents three new top-down updating algorithms for the concurrent data processing environment.

I owe much appreciation and gratitude to my major advisor Dr. Donald D. Fisher for his continuous guidance, motivation, and valuable instructions, especially during the final phase of the thesis writing.

I would extend my appreciation to Dr. Donald W. Grace, and Dr. John P. Chandler for their suggestions and advisements while serving on my committee.

This thesis is dedicated to my mother, Mrs. Lee-In Chen, and to my wife, Sue-Cheng for their sacrifice, support, encouragement, and love.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION . . . . .	1
1.1 Basic Terminology . . . . .	2
1.2 Rationale . . . . .	4
1.3 Objectives . . . . .	5
II. LITERATURE SURVEY . . . . .	7
2.1 AVL and HB(k) Trees . . . . .	9
2.2 Red-Black Trees . . . . .	19
2.3 Weight-Balanced Trees . . . . .	26
2.4 Brother Trees . . . . .	31
2.5 Self-Adjusting Binary Trees . . . . .	39
2.6 B-Trees . . . . .	43
2.7 Multidimensional Binary Search Trees . . . . .	45
III. TOP-DOWN UPDATING AND CONCURRENT OPERATIONS . . . . .	49
3.1 Top-Down Updating for HB(k) Trees . . . . .	50
3.2 Top-Down Updating for Red-Black Trees . . . . .	62
3.3 Top-Down Updating for WB-Trees . . . . .	66
3.4 Top-Down Updating for Splay Trees . . . . .	71
IV. AMORTIZED ANALYSIS AND PERFORMANCE EVALUATIONS . . . . .	76
V. COMPARISONS AND DISCUSSIONS . . . . .	80
VI. APPLICATIONS AND CONCLUSIONS . . . . .	83
6.1 Applications . . . . .	83
6.2 Conclusions . . . . .	84
6.3 Suggestions . . . . .	85
VII. A SELECTED BIBLIOGRAPHY . . . . .	87
APPENDIX A: DEFINITIONS OF TREE VARIANTS . . . . .	96
APPENDIX B: EXAMPLES OF TREE OPERATIONS . . . . .	98

LIST OF TABLES

Table	Page
I. Potentials for Updating Red-Black Trees . . . . .	79

## LIST OF FIGURES

Figure	Page
1. The Notations of A Binary Tree . . . . .	8
2. Bottom-Up Insertion Algorithm for HB(k) Trees . . .	11
3. Swapping Process in the Deletion of HB(k) Trees . .	14
4. Bottom-Up Deletion Algorithm for HB(k) Trees . . .	15
5. Modified Bottom-Up Deletion Algorithm for HB(k) Trees . . . . .	18
6. A Red-Black Tree . . . . .	20
7. The Equivalent of A 2-4 Tree and A Red-Black Tree .	20
8. Bottom-Up Insertion Algorithm for Red-Black Trees .	22
9. Swapping Process in the Deletion of Red-Black Trees . . . . .	24
10. Bottom-Up Deletion Algorithm for Red-Black Trees .	25
11. Examples of A WB(1/3) Tree and A WB(1/4) Tree . . .	28
12. Rotation Types in Weight-Balanced Trees . . . . .	30
13. A 1-2 Brother Tree . . . . .	32
14. The Equivalent of A 1-2 Brother Tree and An AVL Tree . . . . .	32
15. Bottom-Up Insertion Algorithm for 1-2 Brother Trees . . . . .	34
16. Bottom-Up Deletion Algorithm for 1-2 Brother Trees . . . . .	37
17. The Halving Effect of Splaying A Node in Binary Trees . . . . .	41
18. Bottom-Up Splaying Algorithm . . . . .	42

Figure	Page
19. Examples of A 2-3 Tree and A B-Tree of Order 5 . . .	44
20. An Example of A B+-Tree . . . . .	46
21. An Example of A K-d Tree with K=3 . . . . .	48
22. Updating A Nonhomogeneous Tree Structure of HB(k) Trees . . . . .	51
23. Current Node Selection in the Top-Down Insertion of HB(k) Trees . . . . .	52
24. Prebalancing in the Top-Down Insertion of HB(k) Trees . . . . .	54
25. Current Node Selection in the Top-Down Deletion of HB(k) Trees . . . . .	59
26. Prebalancing in the Top-Down Deletion of HB(k) Trees . . . . .	60
27. Updating A Nonhomogeneous Tree Structure of Red-Black Trees . . . . .	63
28. Current Node Selection in the Top-Down Insertion of Red-Black Trees . . . . .	65
29. Current Node Selection in the Top-Down Deletion of Red-Black Trees . . . . .	67
30. Modified Top-Down Update Algorithm for Weight-Balanced Trees . . . . .	69
31. Top-Down Splaying Algorithm . . . . .	74
32. An Example of Bottom-Up Insertion into An AVL Tree . . . . .	99
33. An Example of Bottom-Up Deletion from An AVL Tree . . . . .	100
34. An Example of Bottom-Up Insertion into A Red-Black Tree . . . . .	101
35. An Example of Bottom-Up Deletion from A Red-Black Tree . . . . .	102
36. An Example of Top-Down Insertion into A Weight-Balanced Tree . . . . .	103
37. An Example of Bottom-Up Insertion into A 1-2 Brother Tree . . . . .	104



Figure	Page
38. An Example of Bottom-Up Deletion from A 1-2 Brother Tree . . . . .	105
39. An Example of Bottom-Up Splaying A Node in Binary Tree . . . . .	106
40. An Example of Insertion into A K-d Tree . . . . .	107
41. An Example of Top-Down Insertion into An AVL Tree . . . . .	108
42. Examples of Top-Down Insertion into An HB(3) Tree .	109
43. An Example of Top-Down Deletion from An AVL Tree . . . . .	111
44. Examples of Top-Down Deletion from An HB(3) Tree .	112
45. An Example of Top-Down Insertion into A Red-Black Tree . . . . .	114
46. An Example of Top-Down Deletion from A Red-Black Tree . . . . .	115
47. An Example of Index-Position Search . . . . .	116
48. An Example of Top-Down Splaying Operations . . . .	117
49. A Classifications of Search Tree Data Structures .	118
50. An Example of An HB(3)-Tree . . . . .	119

## CHAPTER I

### INTRODUCTION

In data processing applications, large files of information must be searched to retrieve the requested data, and data may be added to or deleted from files. There are many ways to organize large files in order to perform these operations such as sequential, linked list, hashed, and search tree structures (2,45). However, the sequential file organization suffers the drawback of lengthy updating, the linked list structure has a disadvantage of long random access, and the hashed file has problems in that (1) it can not access the file in record-key sequence and (2) it can not grow substantially in size without redesigning of the hashing routine. Search tree structures, on the other hand, provide a compromise between sequential and random access, and yet have fast access and easy update operations.

Search tree structures have been studied extensively for the past two decades. Figure 49 in Appendix B shows the classifications of search tree data structures and the references of the previous work. The best known is the class of balanced binary search trees which include the AVL trees of Adelson-Velskii and Landis (1), the generalized height-balanced trees of Foster (30), the weight-balanced trees of

Nievergelt and Reingold (68), the red-black trees of Guibas and Sedgwick (36), the brother trees of Ottmann and Six (71), and the self-organizing binary trees of Allen and Munro (4). Another class is the class of multiway search trees such as the B-trees of Bayer and McCreight (10), the K-dimensional (K-d) trees of Bentley (13), and the quad trees of Finkel and Bentley (29).

### 1.1 Basic Terminology

Generally, a tree imposes a hierarchical structure on a collection of elements called nodes, one of which is the root and the rest of which are partitioned into trees, called the subtrees of the root. A tree  $T$  of  $N$  nodes can be defined recursively in the following manner.  $T$  is

1. a null tree (denoted by  $\Lambda$ ) if  $N = 0$ ;
2. a unary tree if  $N = 1$ , and the node also is the root of the tree;
3. an  $(m+1)$ -tuple  $(R, T_1, \dots, T_m)$  tree, where  $R$  is the root node with  $m$ -ary subtrees  $T_1, \dots, T_m$  of  $N_1, \dots, N_m$  nodes respectively.

Thus, a binary tree  $T$  is either a null tree or a three tuple  $(R, T_l, T_r)$  tree with the root node  $R$ , left binary tree  $T_l$ , and right binary tree  $T_r$ . If  $N_1, N_2, \dots, N_k$  is a sequence of nodes in a tree such that  $N_i$  is the parent of  $N_{i+1}$  for  $1 \leq i < k$ , then this sequence is called a "path" from node  $N_1$  to  $N_k$ . The height of a node in a tree is the length of the longest path from the node to a leaf.

Before going on, we need to define some file organiza-

tion terminology (43). "Data" is a collection of facts, concerning people, place, events, or other objects or concepts. A "data item" (also known as element, field, or attribute) is the smallest named unit of data in a data base, such as a student name. A "record" is a collection of data items that is named and referenced as a whole, such as a student record which may include student ID, student name, and other fields. A "file" such as a student file is a collection of all occurrences of a given record type. A "primary key" is a data item that uniquely identifies a record. A "secondary key" is a data item that normally does not uniquely identify a record, but identifies a number of records in a set that share the same property. Terms such as key and primary key are used interchangeably in this paper.

The concept of a search tree is well described by Knuth (45). The basic idea is that an  $m$ -ary node is associated with  $m-1$  keys from a totally ordered universe of keys. There are two distinct ways of associating sets of keys with a tree to give a search tree. The most popular one is to associate keys with internal nodes giving the internal search tree (also known as a homogeneous tree structure). Another one is to associate keys with external nodes giving the external-search tree (also known as a nonhomogeneous tree structure). In the internal-search scheme, a node consists of a "key" field (which holds the single key defining the record), left pointer, right pointer, and additional fields which hold the rest of the data associated with that record. With this approach, an internal node itself not

only contains a record but also serves as a road map for a search operation. On the other hand, the external-search scheme has two kind of nodes; internal nodes and external nodes. An internal node contains a key field and left and right pointers, but no data; all data are stored in external nodes. In this external-search structure, a key is necessary to provide routing or separating in the internal nodes so that searching can be carried out correctly. Kwong and Wood (50) studied a number of routing scheme, among them the left-maximum is the most popular one. In this scheme, the separating key of a binary tree node  $N$  is the maximum key in  $N$ 's left subtree. By the same token, the  $i$ -th separating key in a multiway tree is the maximum key in that  $i$ -th subtree.

It is important to make the convention that if a search key is equal to the value of an internal node, then the search continues in the left subtree until an external node is reached. The merit of a nonhomogeneous tree structure is that we may store all internal nodes in primary memory, and store lengthy data in secondary memory. With this arrangement only one secondary memory access is required to retrieve any record so that the search is much faster.

## 1.2 Rationale

It is not surprising that search tree data structures receive widespread attention because they are fast to search, easy to update, convenient to process both randomly and sequentially, and most of all they have been applied

successfully in different areas of applications such as:

1. the use of balanced binary search trees to maintain tables in the primary memory (45,67,96), and to organize files in magnetic bubble memory, a new secondary storage device (15,21,91,106,107);

2. the use of B-trees to manage files in secondary memory (10,24), to solve geographic range queries (54), and to support dedicated database applications (22,43);

3. the use of quad trees as a structure to perform image processing (70,88), object representation (5,89), and computer graphics (17,105);

4. the use of red-black trees as a persistent data structure to solve the geometric retrieval problems (90);

5. the use of K-d trees to handle multikey records in database applications (14,19).

With so many authors dedicating themselves to the study of search tree data structures, however, we have few survey papers such as the B-tree and its variants by Comer (22), and the quad tree by Samet (87). Thus, this paper concentrates on the study of search tree structures such as the AVL and generalized height-balanced trees, red-black trees, weight-balanced trees, brother trees, self-adjusting binary trees, B-trees, and K-d trees.

### 1.3 Objectives

The objectives of this paper are the survey of search tree data structures, from which the following studies can be conducted:

1. to state the definitions and to show examples for a variety of search tree structures;

2. to present updating algorithms, and to show examples of operations for a variety of search tree structures;

3. to develop new top-down updating algorithms for the generalized height-balanced trees, weight-balanced trees, and self-adjusting binary trees that make concurrent operations become possible;

5. to present the concept of amortized analysis and its applications;

6. to compare the differences among these search tree structures, and to draw conclusions for their applications.

Chapter II reviews the definitions and updating algorithms of these search tree data structures. Chapter III presents three new top-down updating algorithms for the generalized height-balanced trees, the weight-balanced trees, and the self-adjusting binary trees. Chapter IV illustrates the concept of amortized analysis. Chapter V discusses the differences among these search tree data structures. The applications and conclusions are included in Chapter VI.

## CHAPTER II

### LITERATURE SURVEY

Tree data structures are very efficient for performing a sequence of access operations on a set of items selected from a totally ordered universe where each item may contain some associated information such as key and data. The input to each operation is a key; the output of the operation is an indication of whether the key is in the set, along with the associated information if the key is in the set. A binary search tree is a tree which contains the items of the set, one item per node, with the items arranged in symmetric order: if  $P$  is a node containing an item (which has a key  $i$ ), the left subtree of  $P$  contains only items with keys less than  $i$  and the right subtree of  $P$  contains only items with keys greater than  $i$ . The "search" operation in a binary search tree refers to the ability of accessing any item in the tree by going down from the root, branching left if the accessed key is less than the key in the current node, branching right if the accessed key is greater than the key in the current node, or terminating the operation when a null node or a node containing the key is reached. This search operation takes  $O(d)$  time, where  $d$  is the depth of the node finally reached (internal-search scheme is



assumed). Figure 1 shows some notation for a binary tree where  $X$  denotes the current node,  $L(X)$  denotes the left child of  $X$ ,  $R(X)$  denotes the right child of  $X$ ,  $P(X)$  denotes the parent of  $X$ ,  $S(X)$  denotes the sibling of  $X$ ,  $S(P(X))$  denotes the sibling of  $P(X)$ , and  $P(P(X))$  denotes the grandparent of  $X$ .

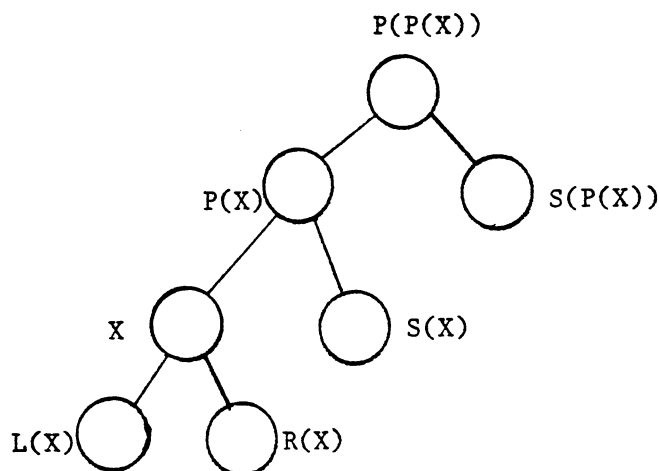


Figure 1

The Notations of A Binary Tree

## 2.1 AVL and HB(k) Trees

AVL-trees were first introduced in 1962 by Adel'son Vel'skii and Landis (1). The tree structure and updating algorithms are well described in Knuth (45). The height of node  $T$  in a tree is defined to be the length of the longest path from node  $T$  to a null (external) node. The height of a null node is zero. If  $T_l$  is the left subtree of  $T$  and  $T_r$  is the right subtree of  $T$ , then  $h(T_l)$  denotes the height of  $T_l$  and  $h(T_r)$  denotes the height of  $T_r$ . An AVL-tree satisfies

1.  $|h(T_l) - h(T_r)| \leq 1$ .
2.  $T_l$  and  $T_r$  are AVL-trees.

Foster (30) introduced the generalized height-balanced trees (HB(k) trees), which have the properties that, for every node, the heights of the right and left subtrees differ at most an integer  $k$  (Figure 50 in Appendix B shows an HB(3) tree). In other words, HB(1)-trees are AVL-trees. There are variants of AVL-trees such as height-ratio balanced trees of Gonnet et al. (33) and one-sided height balanced trees of Kosaraju (46) and Raiha (82). The definitions of these variants are listed in Appendix A. Tarjan (98) proposed update algorithms with  $O(1)$  rotations in the worst case for red-black trees. We can apply his method to construct an insertion algorithm with  $O(1)$  rotations in the worst case for a generalized height balanced tree.

### 2.1.1 Bottom-up Insertion Algorithm

The algorithm of rebalancing an HB(k)-tree after an

insertion proceeds as follows. To insert a new item, say  $X$ , first we perform a binary search for its proper place (a null node) and attach a new node in which the new item is stored (see Figure 2a). We then set the balance-tag of node  $X$  to zero ( $\text{tag}(X) = 0$ ) and mark this node "long". A "long" node is one for which the balance-tag has been modified due to insertion. To eliminate the "long", we let the "long" be the current node  $X$  and proceed with the following steps (algorithm HBI).

1. Test whether node  $X$  is a critical node; a critical node is a node which has a balance-tag of  $|\text{tag}(X)| = k+1$ . If the test is true, go to step (3). If the test is false, go to step (2).

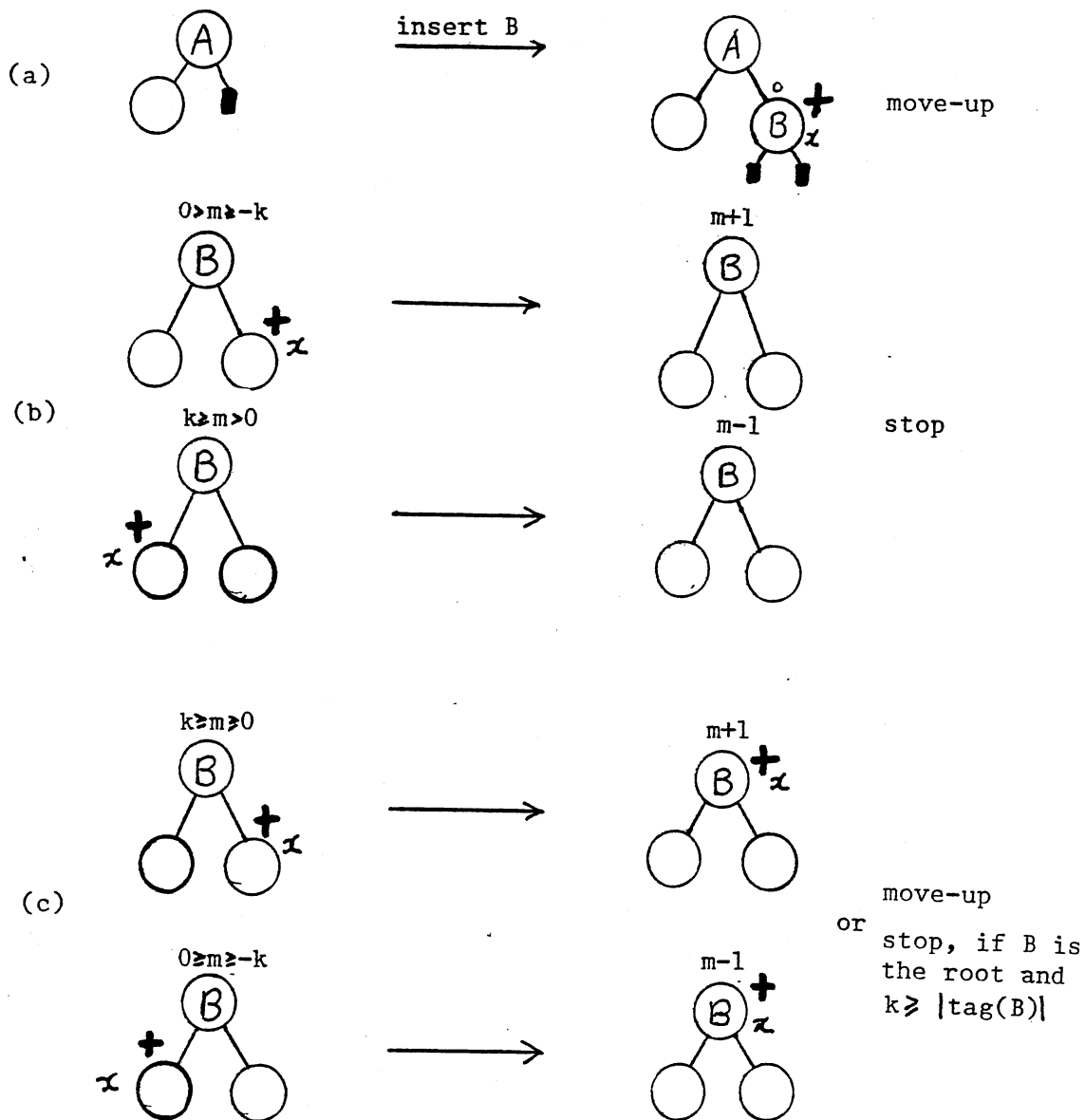
2. Test whether  $P(X)$  has a balance-tag satisfying  $k \geq \text{tag}(P(X)) > 0$  and  $X$  is the left child of  $P(X)$  (or symmetric variant). If the test is true, update the balance-tag of  $P(X)$  and stop (Figure 2b). If the test is false; update the balance-tag of  $P(X)$ , let  $P(X)$  be the new current node  $X$  and go to step (1) (Figure 2c).

3. Test whether  $X$  has a balance-tag satisfying  $k+1$  and  $R(X)$  (see Figure 1) has a balance-tag of  $k \geq \text{tag}(R(X)) > 0$  (or symmetric variant). If the test is true; perform a single rotation, update balance-tags, and stop (Figure 2d). If the test is false; perform a double rotation, update balance tags, and stop (Figure 2e).

Cases (d) and (e) in Figure 2 take  $O(1)$  rotations and terminate the insertion. We then conclude that the bottom up insertion takes  $O(1)$  rotations in the worst case. An

Figure 2

## Bottom-Up Insertion Algorithm for HB(k) Trees



$m, k$  denote balance-tags.

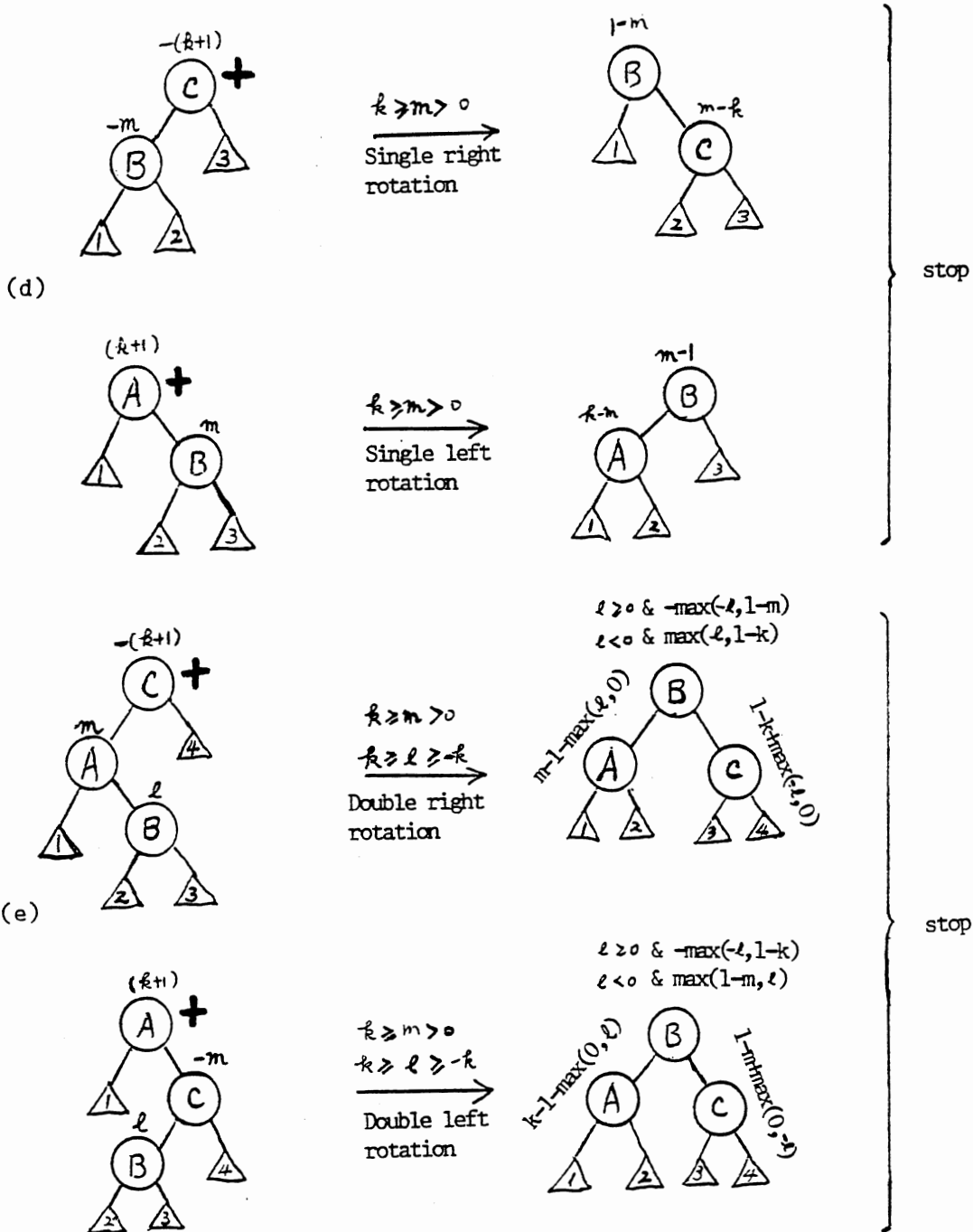
$\bigcirc$  denotes a subtree or a null.

$+$  denotes a "long" (the current node).

$\blacksquare$  denotes a null node.

case (a) has symmetric variant.

Figure 2 (Continued)



example of bottom-up insertion for an AVL-tree is shown in Figure 32 (in Appendix B).

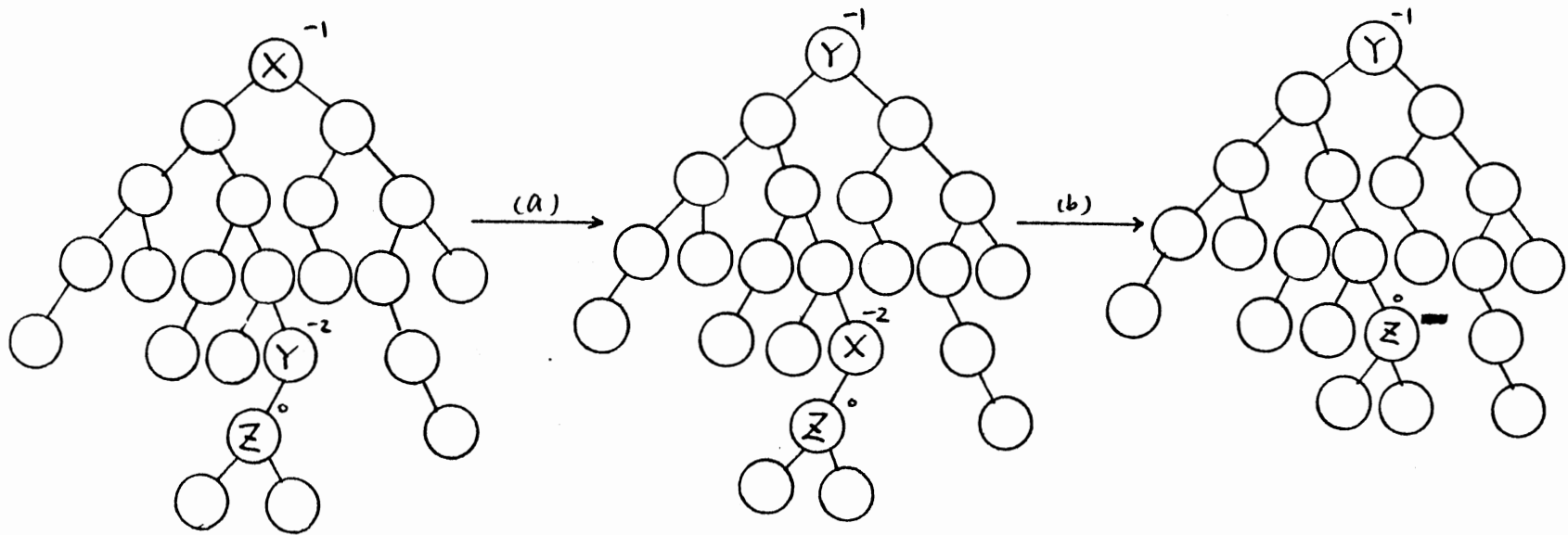
### 2.1.2 Bottom-up Deletion Algorithm

The deletion process is more complex than insertion because it may not be sufficient to apply a restructuring only at the lowest level of imbalance; restructuring may need to be applied at many levels between the site of deletion and the root. To delete an item, say X, from a height-balanced tree first we perform a binary search for X and test whether X has two internal children. If the test is true, we swap X with its inorder predecessor (see Figure 3) found by taking left branch of X and then right branches until reaching a node, say Y, with a null right child. We also exchange balance-tags between X and Y. Now X has at most one child. If X has one child, replace it by its child and produce a "short" (see Figure 4a). A "short" node is one for which the balance-tag has been modified due to deletion. If X has no child, simply delete X and produce a "short" at the null node. To eliminate a "short", we let the "short" be the current node X and proceed with the following steps (algorithm HBD).

1. Test whether X is the tree root. If it is true, we stop. If it is not true, go to step (2).

2. Test whether X is a critical node ( $|\text{tag}(X)| = k+1$ ). If the test is true, go to step (4). If the test is false, go to step (3).

3. Test whether  $P(X)$  has a balance-tag satisfying  $k >$

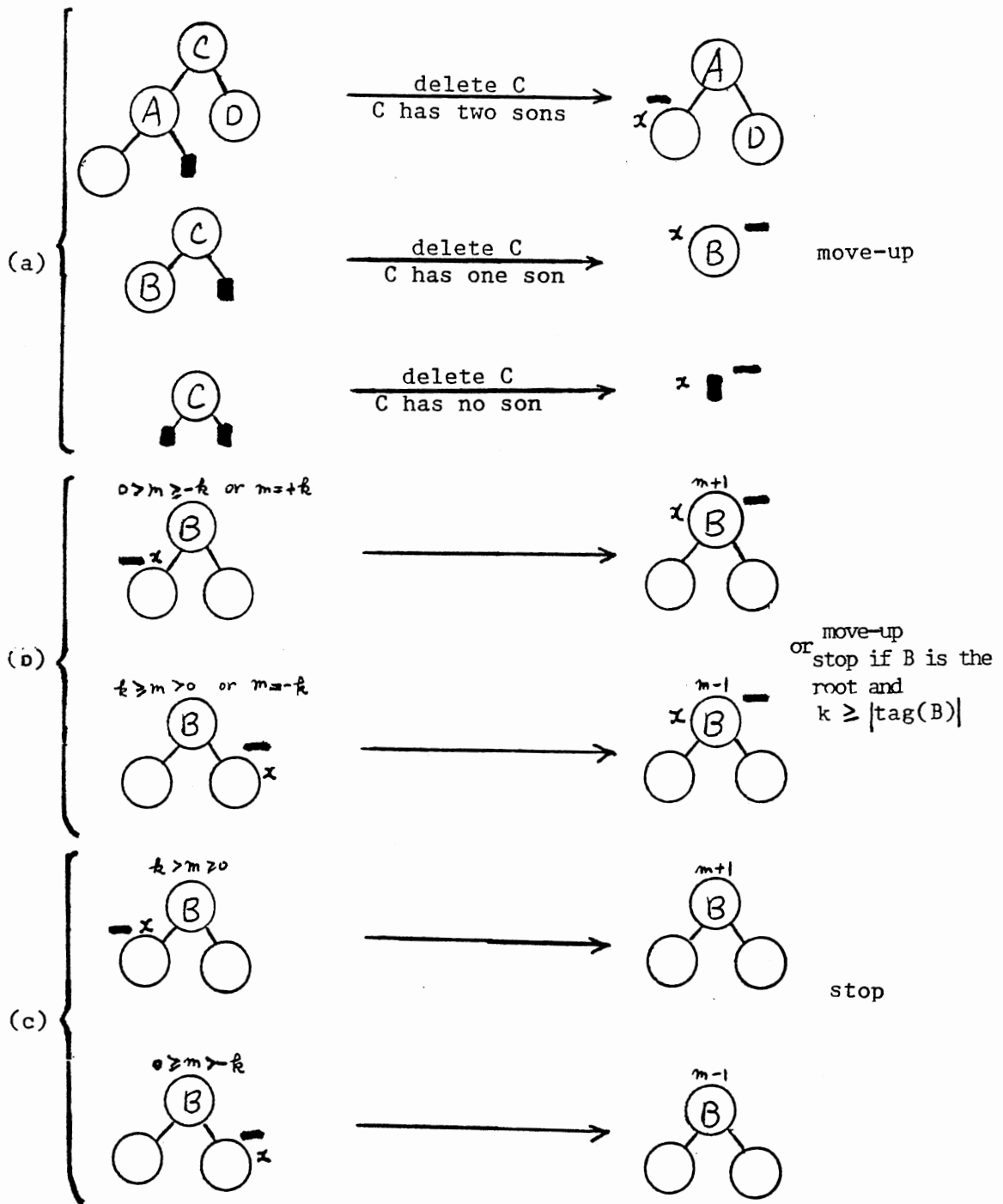


- (a) Swap X with Y and exchange balance tags between X and Y.  
 (b) Replace X with its child Z and produce a "short" at node Z.  
 — denotes the "short"

Figure 3

Swapping Process in the Deletion of HB(k) Trees

Figure 4  
Bottom-Up Deletion Algorithm for HB(k) Trees



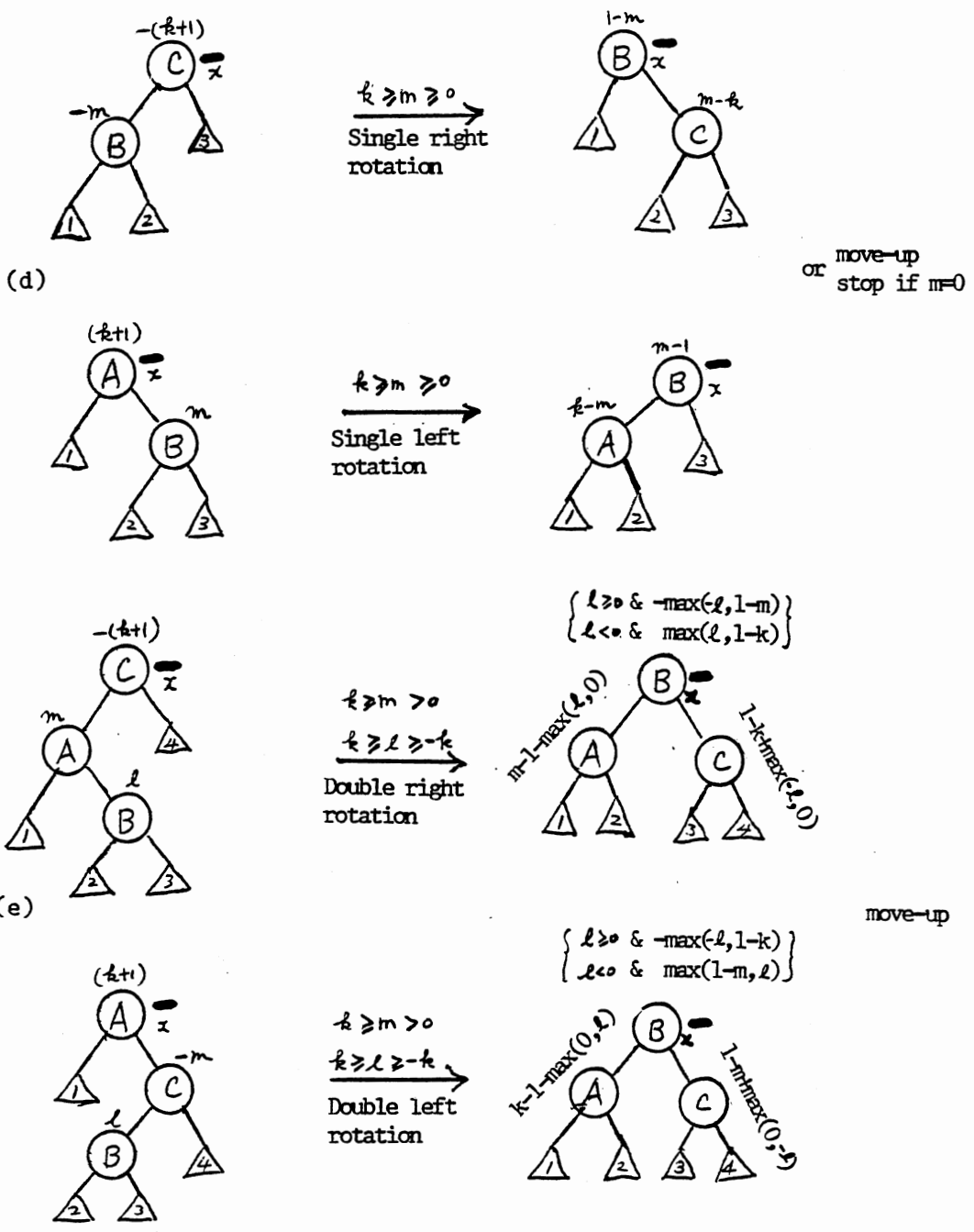
Case (a) has symmetric variant.

— denotes a "short".  $m, k$  denote balance-tags.

■ denote a null node. ○ denotes a subtree or null node.



Figure 4 (Continued)



Cases (d) and (e) are terminals if B is the root.

$\text{tag}(P(X)) \geq 0$  and  $X$  is the left child of  $P(X)$  (or symmetric variant). If the test is true, update the balance-tag of  $P(X)$  and stop (Figure 4c). If the test is false; update the balance-tag of  $P(X)$ , let  $P(X)$  be the new current node  $X$  and go to step (1) (Figure 4b).

4. Test whether  $X$  has a balance-tag satisfying  $k+1$  and  $R(X)$  has a balance-tag of  $k \geq \text{tag}(R(X)) \geq 0$  (or symmetric variant). If the test is true; perform a single rotation, update balance-tags, select a new current node, and go to step (1) (Figure 4d). If the test is false; perform a double rotation, update balance-tags, select a new current node, and go to step (1) (Figure 4e).

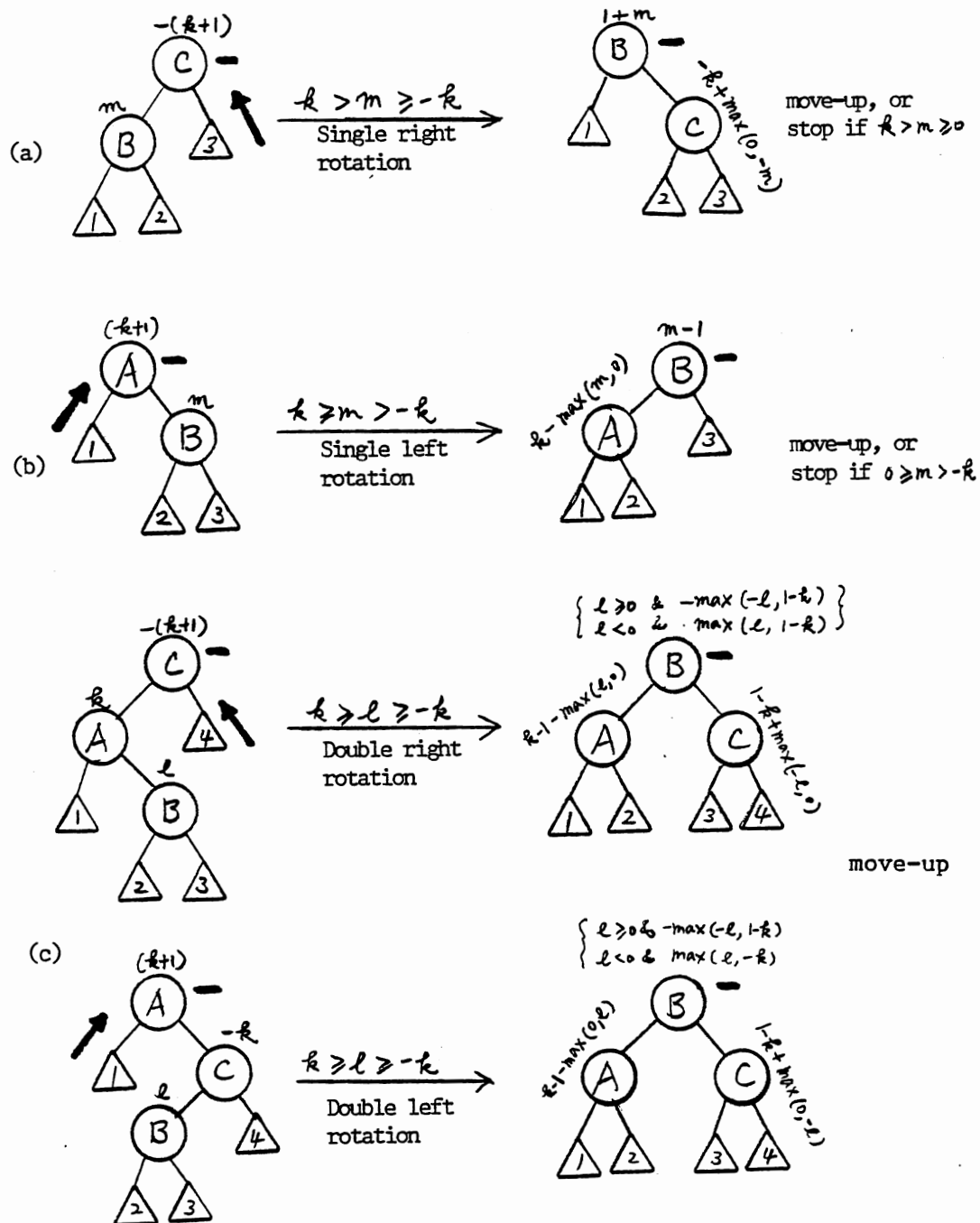
Cases (b), (d), and (e) are terminals only when the new current node is the root node. That is, the restructuring operation may be needed for many levels before the termination of a deletion. We then conclude that deletion from a generalized height-balanced tree takes  $O(\log n)$  rotations in the worst case. An example of bottom up deletion from an AVL-tree is shown in Figure 33 (in Appendix B).

To reduce the chance of requiring  $O(\log n)$  rotations in the worst case, we can modify the cases in Figure 4e to perform a single rotation instead of a double rotation that will eliminate the "short" and stop the deletion (see Figure 5). The steps (1) and (2) in the above algorithm remain unchanged and step (3) is modified as follows (algorithm HBMD).

3. Test whether  $X$  has a balance-tag satisfying  $k+1$  and  $R(X)$  has a balance-tag of  $-k$  (or symmetric variant). If the

Figure 5

## Modified Bottom-Up Deletion Algorithm for HB(k) Trees



test is true; perform a double rotation, update balance tags, select a new current node, and go to step (1) (see Figure 5c). If the test is false, go to step (4).

4. Now  $X$  has a balance-tag satisfying  $k+1$ , and  $R(X)$  has a balance-tag satisfying  $k \geq \text{tag}(R(X)) > -k$  (or symmetric variant). We perform a single rotation and update balance tags (Figure 5b), and then test whether  $0 \geq \text{tag}(R(X)) > -k$  holds. If the test is true, we stop. If the test is false; we select a new current node, and go to step (1).

## 2.2 Red-Black Trees

A red-black tree is a balanced binary search tree (see Figure 6) in which each node has a color, either red or black, subject to the following constraints.

1. An external node is marked as a black node.
2. All paths from the root to an external (null) node contain the same number of black nodes (black constraint).
3. Any red node, if it is not a root node, has a black parent (red constraint).

Bayer (9) first introduced these trees, calling them "symmetric binary B-trees." Guibas and Sedgewick (36) studied the properties of these and related trees, calling them "red-black trees." Olivie (69) used an equivalent definition, calling them "half-balanced trees". We can convert a red-black tree into a 2-4 tree, by condensing every red node into its parent, in which every internal node has either two, three, or four children and all external nodes have the same depth (see Figure 7).

Figure 6 A Red-Black Tree

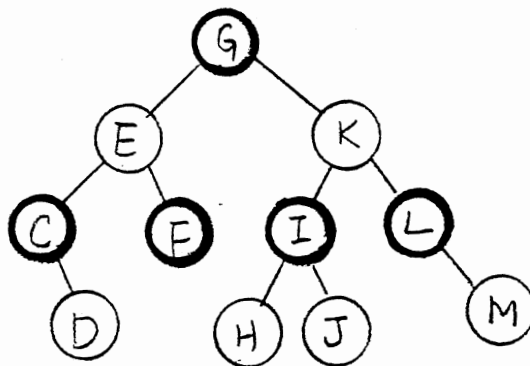
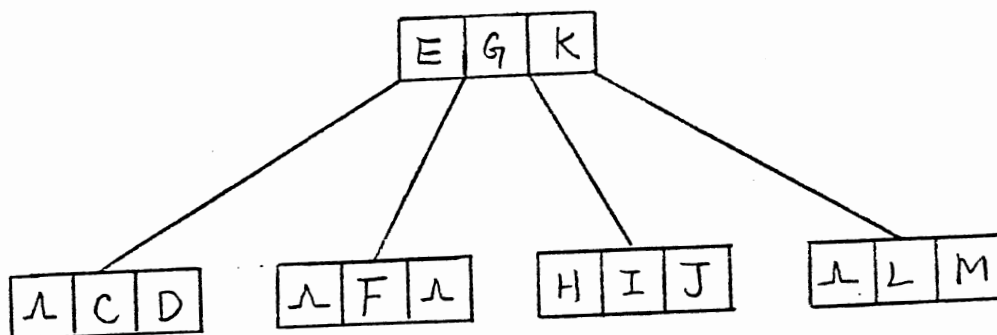


Figure 7

The Equivalent of A 2-4 Tree and A Red-Black Tree



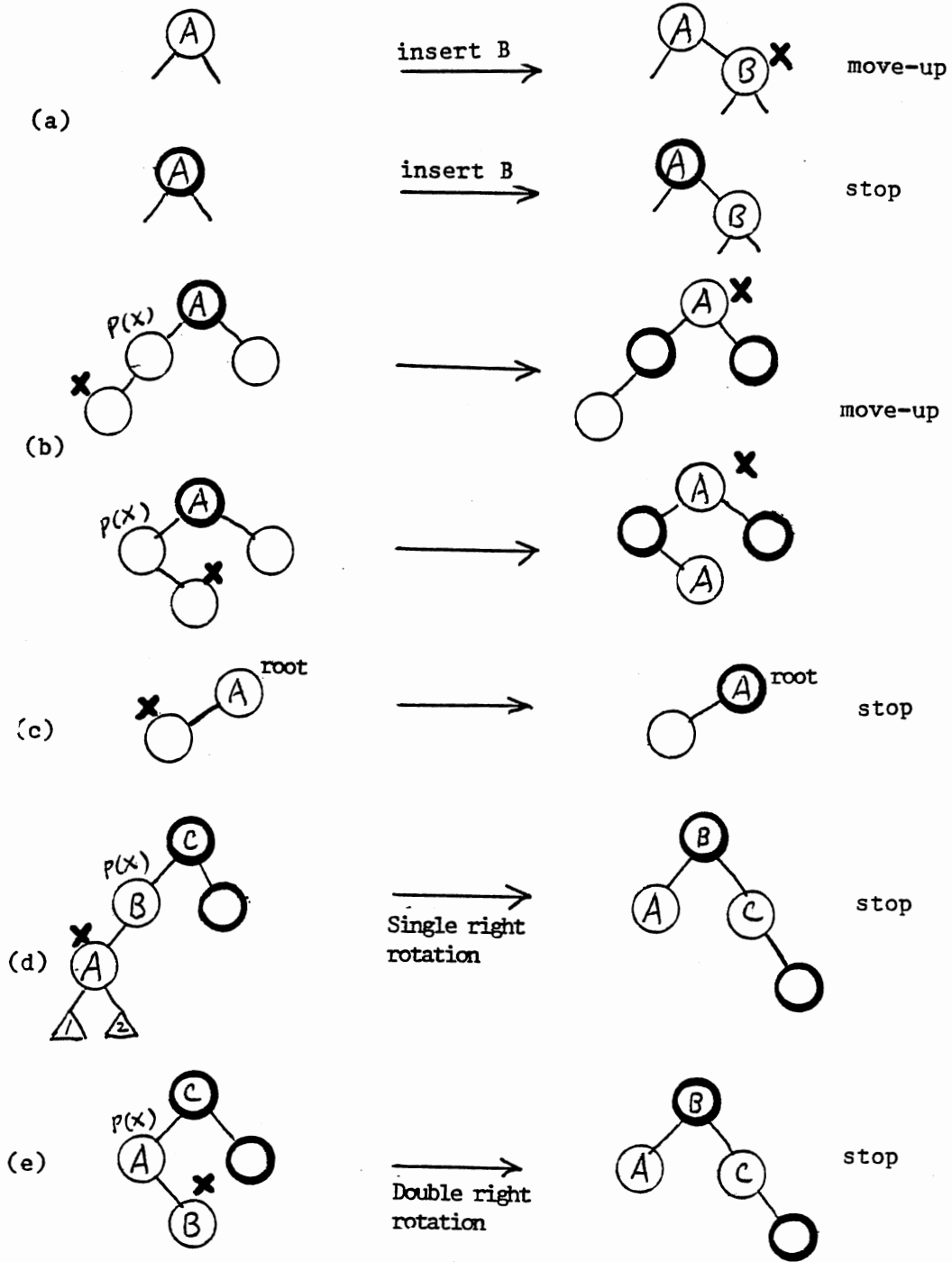
### 2.2.1 Bottom-Up Insertion Algorithm

Tarjan (98) proposed a bottom-up update method which requires  $O(1)$  rotations in the worst case and  $O(1)$  color updates in the amortized case (defined in Chapter IV). Tarjan's bottom-up insertion algorithm proceeds as follows: (A homogeneous tree structure is assumed.) To insert an item, first we perform binary search for the appropriate null node (the bottom of the tree) and attach a new node containing the new item (see Figure 8a). We color this node red. This preserves the black constraint but may violate the red constraint if the parent of the new node is a red node. To eliminate the violation, we let the new node be the current node  $X$  and proceed with the following steps (algorithm RBI).

1. If  $P(X)$  is a black node, we stop. If  $P(X)$  is a red node, go to step (2).
2. If  $P(X)$  is a root node, color  $P(X)$  black and stop. (Figure 8c). If  $P(X)$  is not a root node, go to step (3).
3. If  $P(X)$  has a red sibling, perform color changes (Figure 8b) and let  $P(P(X))$  be the new current node  $X$  and go to step (1). If  $P(X)$  has a black sibling, go to step (4).
4. If node  $X$  is the left child of  $P(X)$  and  $P(X)$  has a right black sibling (or symmetric variant), perform a single right rotation and stop. (Figure 8d). If node  $X$  is the right child of  $P(X)$  and  $P(X)$  has a right black sibling (or symmetric variant), perform a double right rotation and stop (Figure 8e).

Figure 8

Bottom-Up Insertion Algorithm for Red-Black Trees



Symmetric variants are not shown. Case (b) is a terminal if A is the root.  
 x denotes the current node. ○ denotes the red nodes.  
 △ denotes a subtree or null node. ● denotes the black node.

Figures 8d and 8e take  $O(1)$  rotations and terminate the insertion. We then conclude that the bottom-up insertion for red-black trees takes  $O(1)$  rotations and  $O(1)$  color changes in the amortized case (98). An example of bottom-up insertion for an red-black is shown in Figure 34 (in Appendix B).

### 2.2.2 Bottom-Up Deletion Algorithm

To delete an item, say  $X$ , first we perform binary search for  $X$  and test whether  $X$  has two nonnil children. If the test is true, swap  $X$  with its symmetric-order predecessor, say  $Y$ , and also exchange colors between  $X$  and  $Y$ . Now  $X$  has at most one child. If  $X$  has one child (see Figure 9a), say  $Z$ , then  $X$  must be a black node and  $Z$  must be a red node. We swap  $X$  with  $Z$  and exchange color between  $X$  and  $Z$ . We then delete  $X$  and stop. On the other hand, if  $X$  has no child (see Figure 9b), we test whether  $X$  is a red node. If the test is true, we simply delete  $X$  and stop. If the test is false, we delete  $X$  and mark this null node "short". A "short" node is one from which paths down from it contain one fewer black node than paths down from its sibling. We eliminate the "short" (assume node  $X$  has a "short") by proceeding with the following steps (algorithm RBD).

1. Test whether  $X$  has a black parent and a black sibling with two black children. If the test is true; we bubble the "short" up by one level, update colors, and repeat step (1) until it no longer applies. (see Figure 10b). If the test is false, go to step (2).



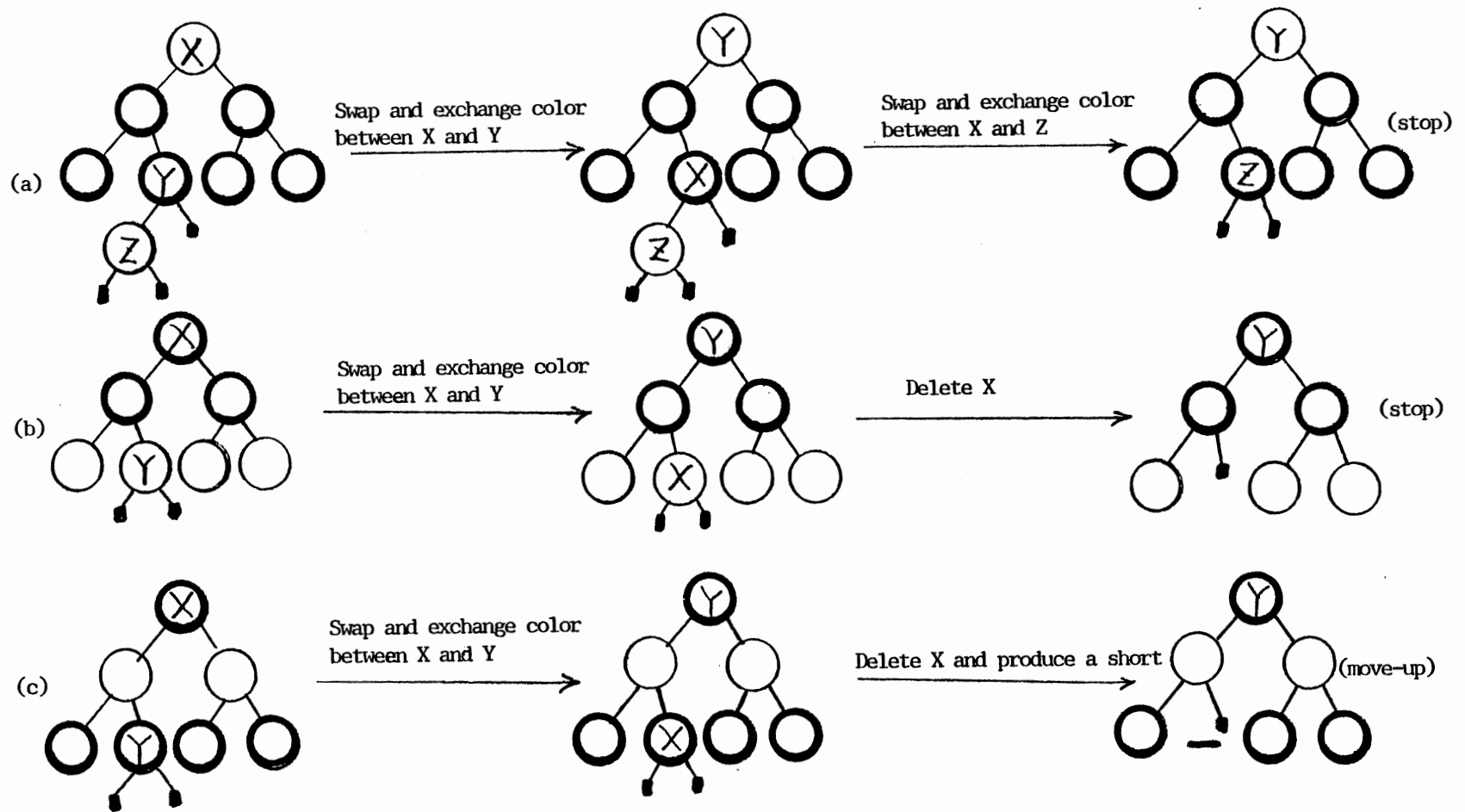
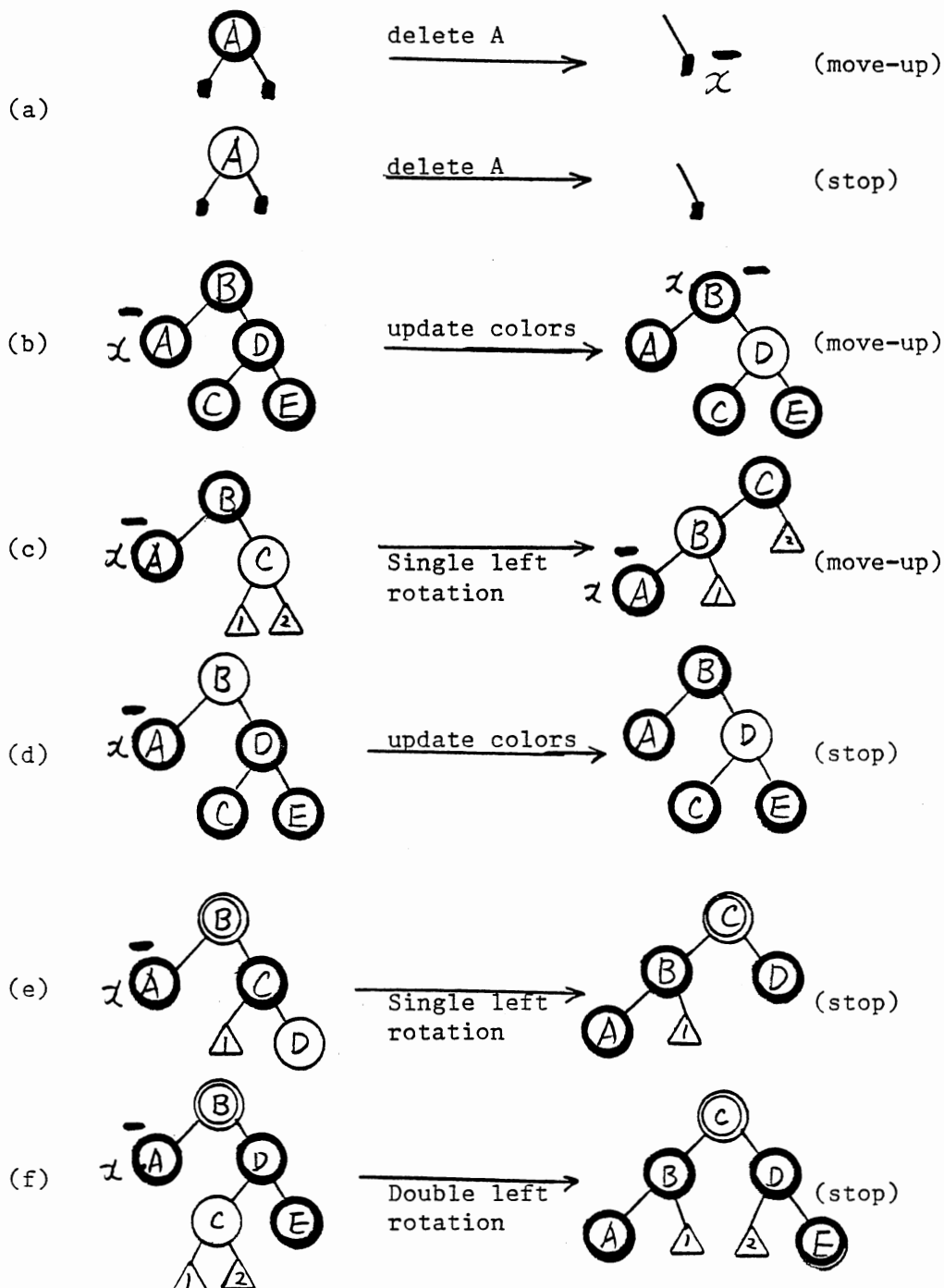


Figure 9 Swapping Process in the Deletion of Red-Black Trees

Figure 10

## Bottom-Up Deletion Algorithm for Red-Black Trees



Symmetric cases are not shown, case (b) is a terminal if B is the root.

⊙ denotes the color can be either red or black.

χ denotes the current node, — denotes the "short".

2. Test whether the sibling of  $X$  is a red node. If the test is true, perform a single rotation and update colors (see Figure 10c). Go to step (3) regardless of the test result.

3. Test whether the right child of  $X$ 's right sibling (or symmetric variant) is a red node. If the test is true; perform a single rotation, update colors (see Figure 10e), and stop. If the test is false, go to step (4).

4. Test whether the left child of  $X$ 's right sibling (or symmetric variant) is a red node. If the test is true; perform a double rotation, update colors (see Figure 10f), and stop. If the test is false, perform color changes (see Figure 10d) and stop.

In the deletion process the maximum number of rotations is two, that is, the worst case is a single rotation (Figure 10c) followed by another single rotation (Figure 10e) or double rotation (Figure 10f). We then conclude that the deletion can be done in  $O(1)$  rotations and  $O(1)$  color changes in the amortized case (98). An example of bottom-up deletion for a red-black tree is shown in Figure 35 (in Appendix B).

### 2.3 Weight-Balanced Trees

The concept of weight-balanced trees (also known as bounded-balance trees) was first introduced by Nievergelt and Reingold (68), instead of using the height of a tree node as a balance constraint in the AVL-tree. A weight balanced tree is restricted by the relative number of nodes in

left and right subtrees. Reingold and Hansen (83, pp. 311) gave the definition of weight-balanced trees (WB-trees) as follows.

Let  $T$  be an extended binary tree such that  $T$  consists of either a single external node or a root node with two subtrees  $T_l$  and  $T_r$ . The "balance factor",  $B(T)$  of a tree is then defined as

$$B(T) = \begin{cases} 1/2, & \text{if } T \text{ is an external node,} \\ |T_l|/|T|, & \text{otherwise.} \end{cases}$$

Where  $|T_l|$  denotes the total number of external nodes in  $T$ 's left subtree and  $|T|$  denotes the total number of external nodes of  $T$ . A tree  $T$  is said to be of "weight-balance" of  $\alpha$ , or in the set of  $WB(\alpha)$  for  $0 \leq \alpha \leq 1/2$ , if

1.  $\alpha \leq B(T) \leq 1-\alpha$ .
2. Both  $T_l$  and  $T_r$  are also in the set of  $WB(\alpha)$ .

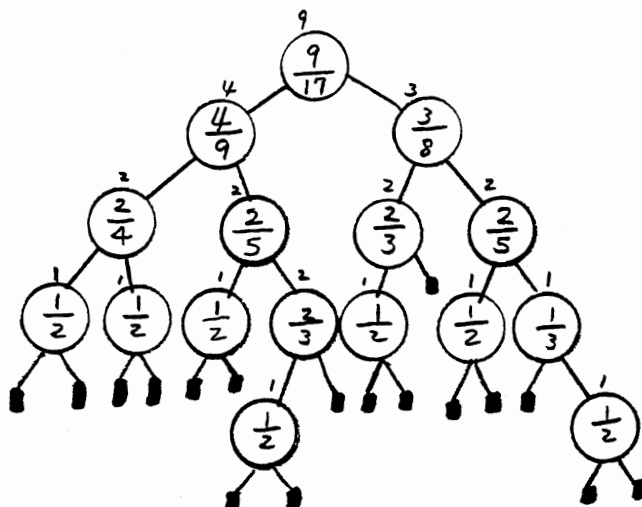
Figure 11 shows the examples of a  $WB(1/3)$  tree and a  $WB(1/4)$  tree. In order to compute the weight-balance factor  $\alpha$ , we maintain in every node a  $SIZE$  and  $SUBT$  fields. The  $SIZE(T)$  is the total number of internal nodes in  $T$  (includes  $T$  itself), and the  $SUBT(T)$  is the total number of internal nodes in  $T$ 's left subtree. The weight-balance factor of  $T$  is then computed by  $(SUBT(T)+1)/(SIZE(T)+1)$ .

### 2.3.1 Update Algorithms for Weight-Balanced Trees

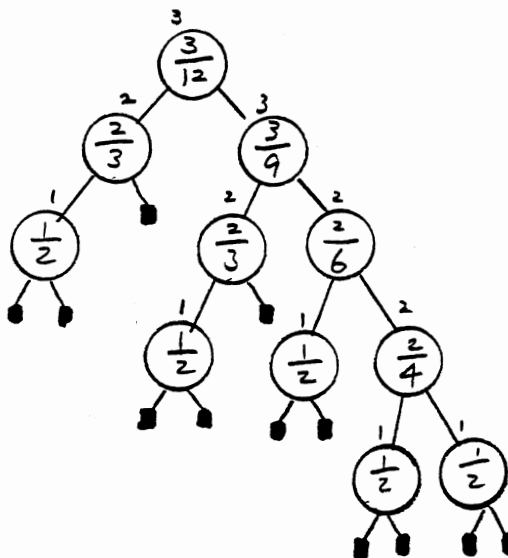
In most cases, the insertion and deletion of nodes in WB-trees are accomplished by a single top-down pass over the search path. However, in the case of redundant insertion or

Figure 11

Examples of A WB(1/3) Tree and A WB(1/4) Tree



(a) WB(1/3)-tree



(b) WB(1/4)-tree

deletion, a second top-down pass is required for correcting the SIZE and SUBT fields in every node along the access path. The update algorithm for weight-balanced trees proceeds as follows. To update an item, either insertion or deletion, first we perform binary search for that item along the access path. As each node is visited, we update the SIZE and SUBT fields of that node and compute its weight-balance factor. If an imbalance occurs, we perform rotations to rebalance the tree. Figure 12 shows the type of rotations used to rebalance WB-trees, and the derivation of those formulas is shown in Reingold and Hansen (83, pp. 317) and Fisher (OSU notes).

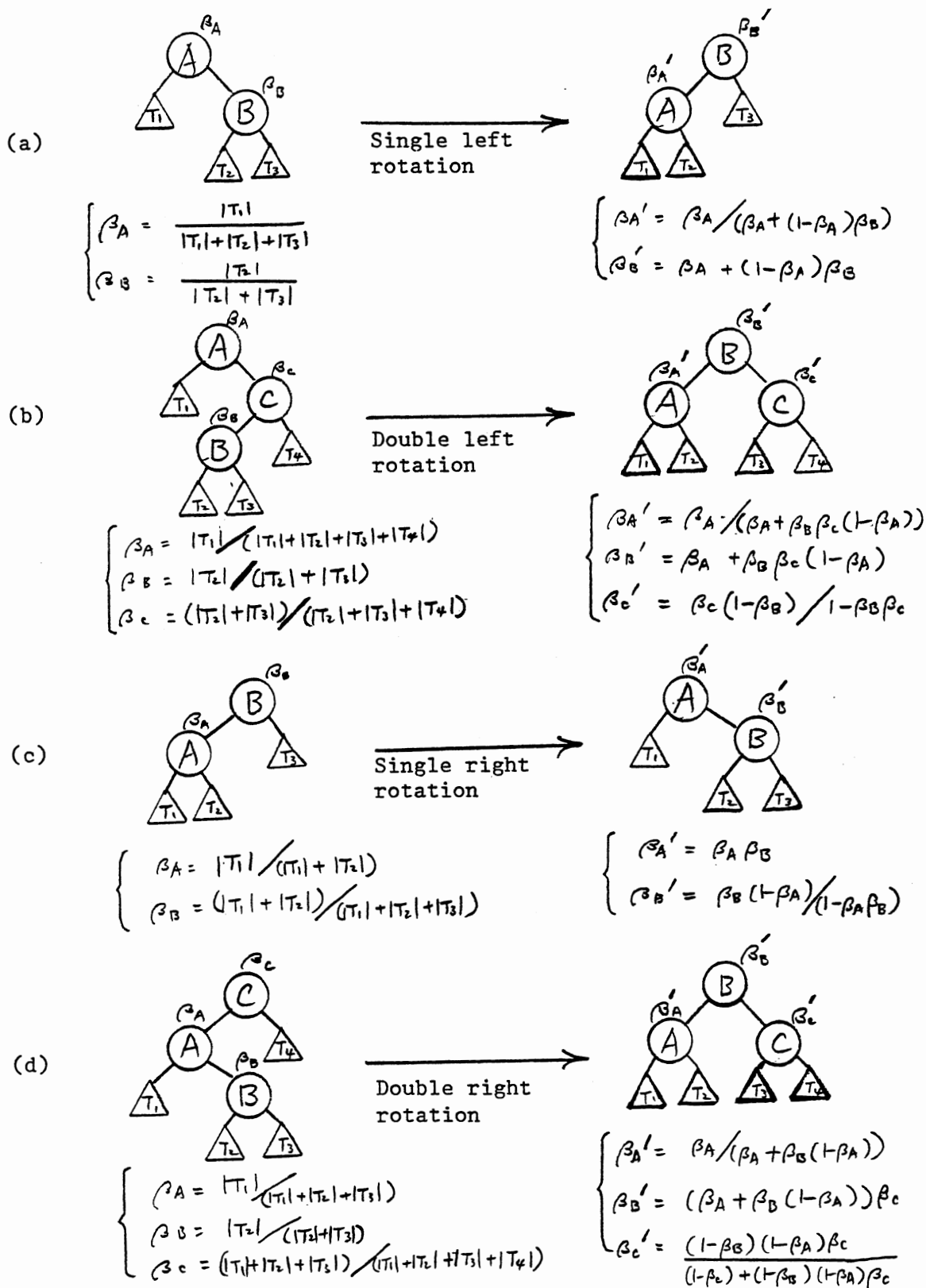
Let  $T$  be the current node,  $T_L$  and  $T_r$  be the left and right subtrees of  $T$ . The type of rotation is determined according to the following cases (algorithm WBT).

1. If  $|T_L|/|T| < \alpha$  and  $B(T_r) \leq 1/(2-\alpha)$ , we perform a single left rotation as shown in Figure 12a.
2. If  $|T_L|/|T| < \alpha$  and  $B(T_r) > 1/(2-\alpha)$ , we perform a double left rotation as shown in Figure 12b.
3. If  $|T_L|/|T| > (1-\alpha)$  and  $B(T_L) \geq (1-\alpha)/(2-\alpha)$ , we perform a single right rotation as shown in Figure 12c.
4. If  $|T_L|/|T| > (1-\alpha)$  and  $B(T_L) < (1-\alpha)/(2-\alpha)$ , we perform a double right rotation as shown in Figure 12d.

The update algorithm takes  $O(\log n)$  rebalances in the worst case. A second pass is needed for correcting the SIZE and SUBT fields of each node along the access path for a redundant update.

Figure 12

## Rotation Types in Weight-Balanced Trees



## 2.4 Brother Trees

Brother trees and their variants have been studied for the past decade. For example; the brother trees (also known as leaf-search trees) of Ottmann and Six (71), right brother trees of Ottmann, Six, and Wood (72), 1-2 brother trees of Ottmann and Wood (76), and 2-3 brother trees of Kriegel, Vaishnavi, and Wood (47). In this section, we choose the 1-2 brother tree as a representative of brother trees not only because 1-2 brother trees are closely related to AVL-trees but also because the cost analysis of random 1-2 brother trees is available in the literature.

The properties of 1-2 brother trees are clearly defined by Ottmann and Stucky (75). A 1-2 brother tree is a tree in which every node has either one or two children and satisfies

1. All external nodes have the same depth.
2. Every node with only one child has a brother with two children.
3. The root always has two children.
4. An internal node with two children has one key, and an internal node with only one child has no key.
5. 1-2 brother trees are binary search trees.

Figure 13 shows an example of a 1-2 brother tree. We can convert a 1-2 brother tree into an AVL tree by replacing each unary node (one which has only one child) with its only child (see Figure 14). Conversely, we can also transform an AVL tree into a 1-2 brother tree by adding a new



Figure 13 A 1-2 Brother Tree

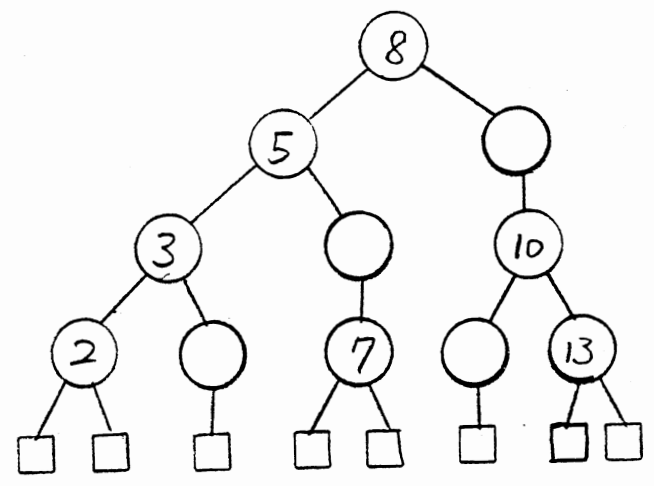
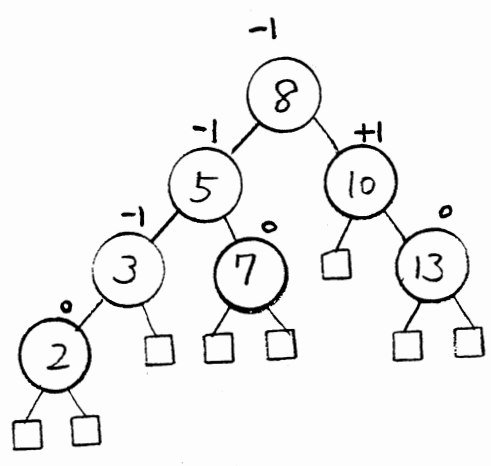


Figure 14

The Equivalent of A 1-2 Brother Tree and An AVL Tree



node into nodes which have non-zero balance-tag. For example; if node X has a balance-tag of +1 , we then insert a unary node as X's left child.

#### 2.4.1 Bottom-Up Insertion Algorithm

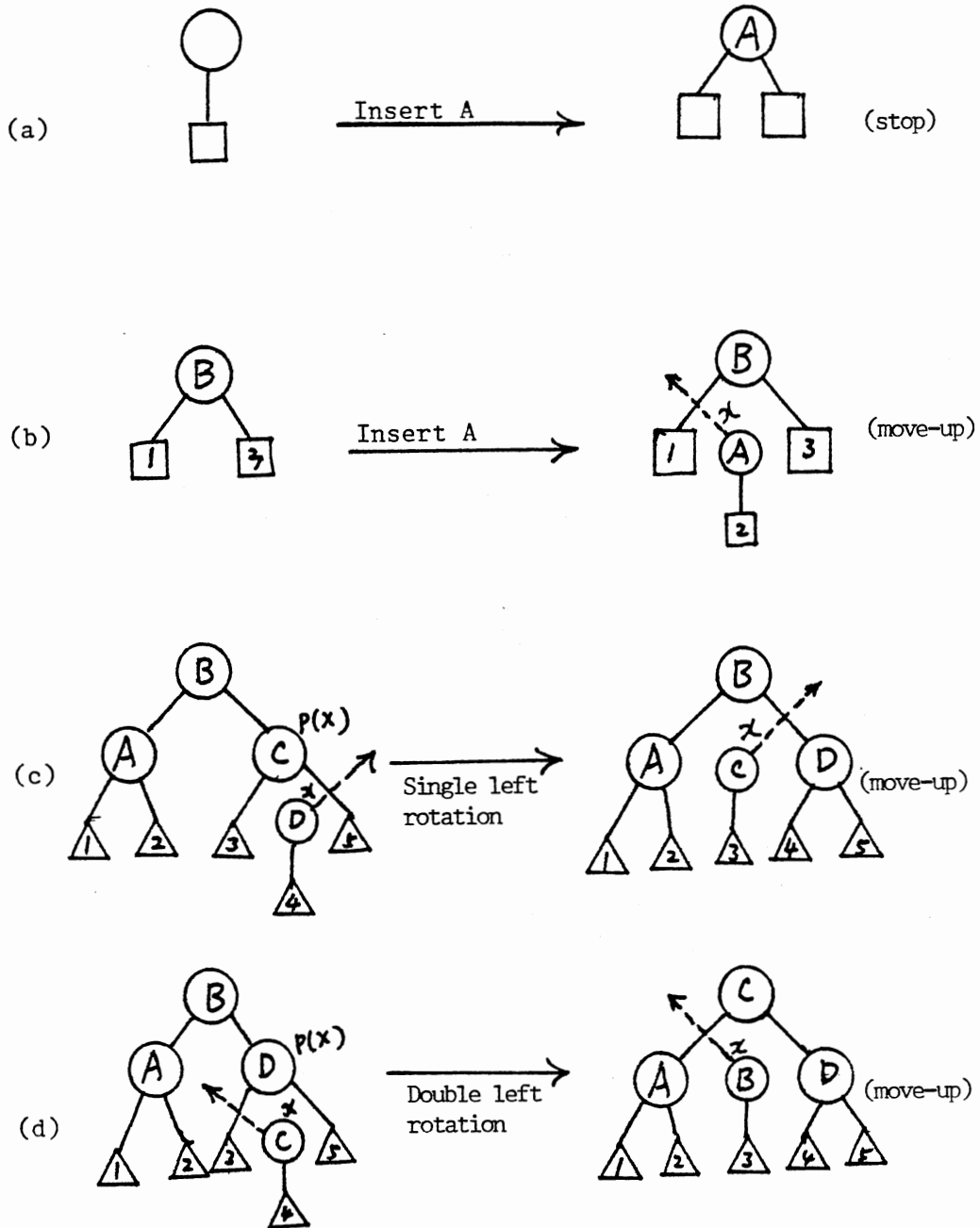
Ottmann and Wood (76) proposed bottom-up updating algorithms for 1-2 brother trees that work as follows. To insert a new item, first we perform a binary search for its proper site (an external node). If the parent of this external node is a unary node (which contains no key); we simply store the new item into the unary node, attach it with another external node to become a binary node (see Figure 15a), and terminate the insertion. If the parent of this external node is a binary node (which contains one key), we create a node (say X) containing the new item (see Figure 15b) and then perform rotations or increase the tree height according to the following steps (algorithm B12I).

1. Test whether the sibling of P(X) is a unary node. If the test is true; we perform a single rotation if X and P(X) are both right or left children (Figure 15e) , or perform a double rotation (Figure 15f) if X is a left child and P(X) is a right child (or vice versa), we then terminate the insertion. If the test is false, go to step (2).

2. Test whether P(X) is a binary node and P(P(X)) is a unary node or P(X) is a binary tree root. If the test is true; we perform a single rotation, increase the tree height by one if P(X) is the tree root, and terminate the insertion (Figure 15g). If the test is false, go to step (3).

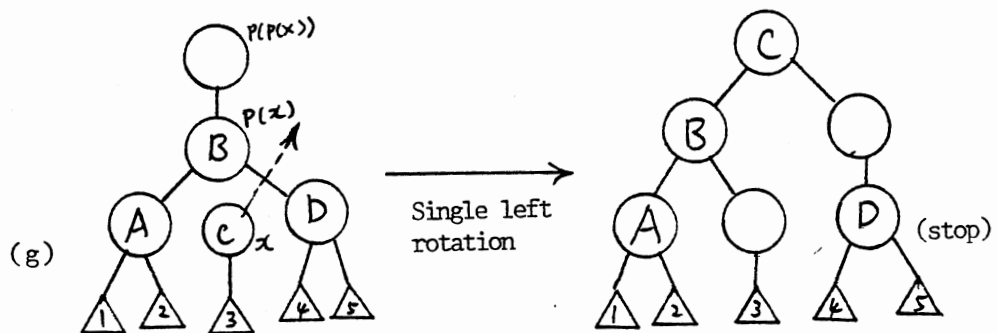
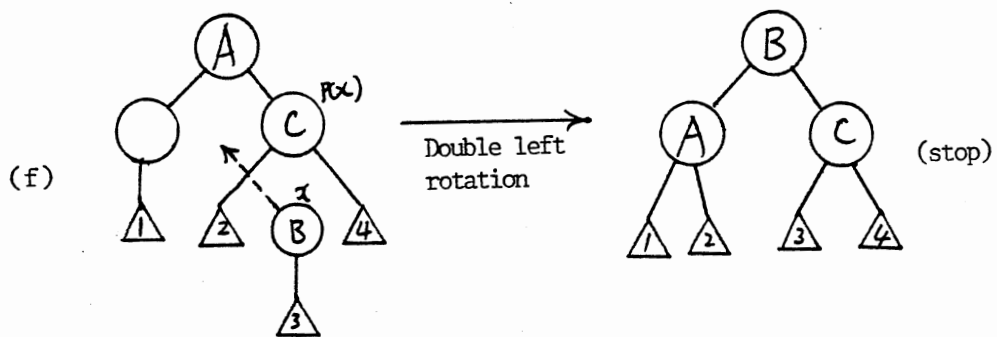
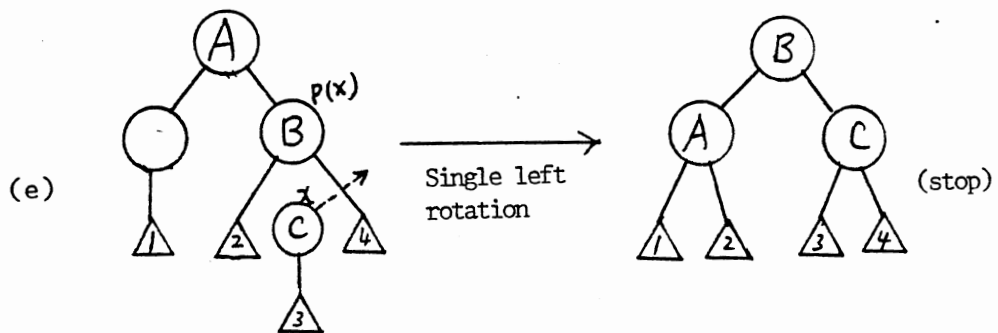
Figure 15

Bottom-Up Insertion Algorithm for 1-2 Brother Trees



Symmetric variants are not shown.  $\chi$  denotes the current node.

Figure 15 (Continued)



3. Now both  $P(X)$  and  $P(P(X))$  are binary nodes. We then test whether  $X$  and  $P(X)$  are both right or left children. If the test is true, we perform a single rotation (see Figure 15c) and go to step (1). If the test is false, we perform a double rotation and go to step (1).

The insertion algorithm B12I take  $O(\log n)$  rebalances in the worst case. An example of bottom-up insertion into a 1-2 brother tree is shown in Figure 37 (in Appendix B).

#### 2.4.2 Bottom-Up Deletion Algorithm

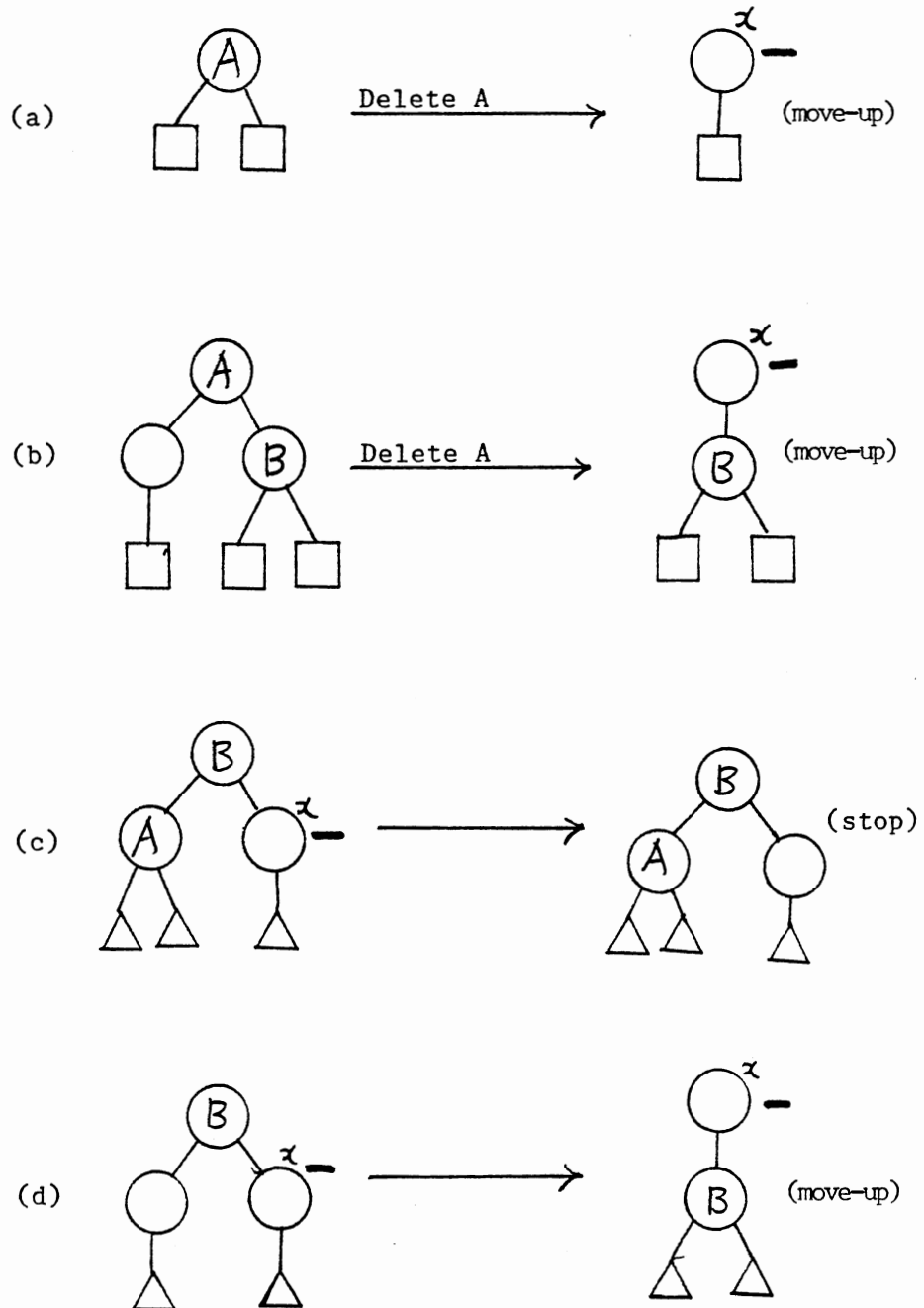
To delete an item (say  $X$ ), first we perform a binary search for  $X$  and test whether  $X$  has two binary children. If the test is true, we swap  $X$  with its symmetric-order predecessor. Now  $X$  has at most one binary child. If  $X$  has one binary child (see Figure 16b), we delete  $X$  and produce a "short". A "short" node is a unary node which is produced due to deletion. If  $X$  has no binary child, then  $X$  must have two external node (see Figure 16a). We delete  $X$  and produce a "short". To eliminate the "short", we let the "short" be the current node  $X$  and proceed with the following steps (algorithm B12D).

1. Test whether  $X$  has a binary brother. If the test is true, we terminate the deletion (see Figure 16c). If the test is false, go to step (2).

2. Test whether  $X$  has a unary brother. If the test is true, we move the shortness up by one level (see Figure 16d) and go to step (1). If the test is false, go to step (3).

Figure 16

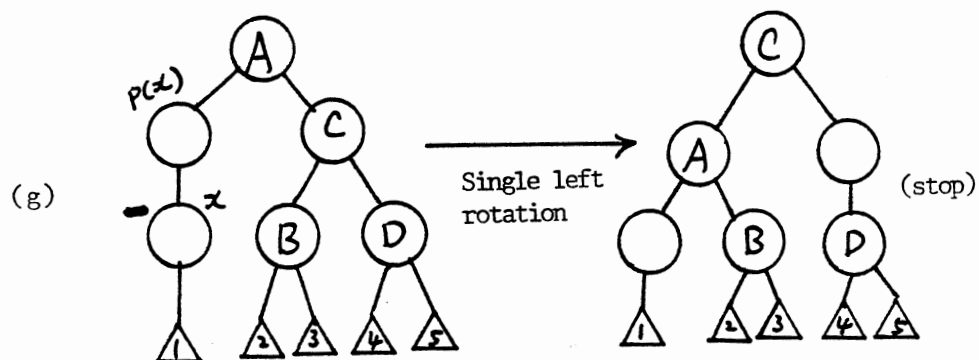
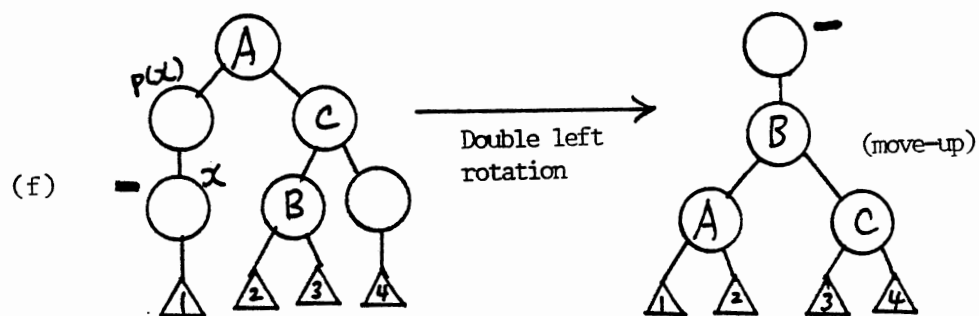
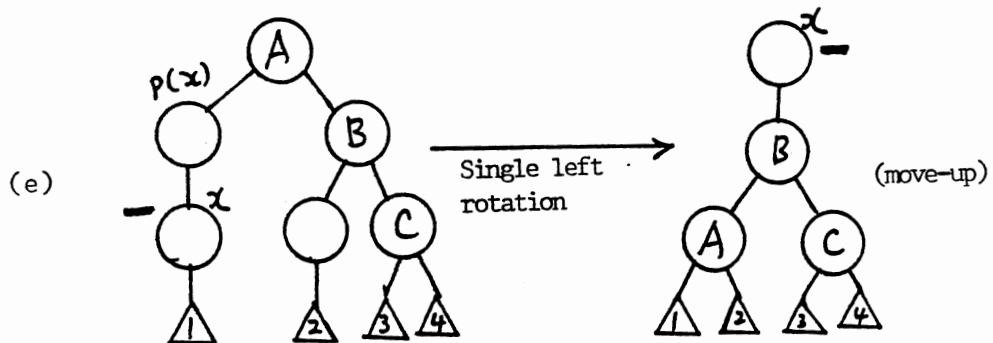
## Bottom-Up Deletion Algorithm for 1-2 Brother Trees



Symmetric variants are not shown.  $\text{---}$  denote the "short".

Cases (b), (d), (f), and (g) are terminals if B is the new tree root.

Figure 16 (Continued)



3. Now  $P(X)$  is a unary node, we then test whether  $P(X)$  has a binary brother with two binary children. If the test is true, we perform a single rotation (see Figure 16g) and terminate the deletion. If the test is false, go to step (4).

4. Test whether  $P(X)$  is a left child of  $P(P(X))$  and  $S(X)$  has a unary left child. If the test is true; we perform a single rotation (see Figure 16e), move the shortness up by one level, and go to step (1). If the test is false; we perform a double rotation (see Figure 16f), move the shortness up by one level, and go to step (1).

The deletion algorithm B12D takes  $O(\log n)$  rebalances in the worst case. An example of bottom-up deletion from a 1-2 brother tree is shown in Figure 38 (in Appendix B). The insertion and deletion algorithms for 1-2 brother trees were implemented with PASCAL (73).

## 2.5 Self-Adjusting Binary Trees

The idea behind "self-adjusting" tree structure is: whenever an item has been successfully located, it is moved to the tree root. In this way we assume that frequently requested items remain fairly near the root, so that their access is relatively less expensive.

Knuth (45) analyzed the related problems of self organizing sequential searching, and Allen and Munro (4) studied the similarities and differences between the sequential search and binary search tree models. They proposed a "simple exchange" method to move the requested item to the



tree root. This method suffers long access sequence such that the time per access is  $O(n)$  for a tree with  $n$  nodes.

Recently, Sleator and Tarjan (92) proposed a new restructuring algorithm, called "splaying", which does rotations along the search path and moves the requested item bottom-up to the tree root. The beauty of this "splaying" is that it does the rotation in pairs and roughly halves the depth of every node along the access path (whenever a double rotation occurs, the height is reduced by one). Figure 17 shows the examples of this halving effect. To splay a tree at node  $X$ , first we search for  $X$  and then trace back from  $X$  to the tree root along the access path and proceed with the following steps (algorithm SPLAY).

1. Test whether  $X$  is the tree root. If the test is true, terminate the splay operation. If the test is false, go to step (2).

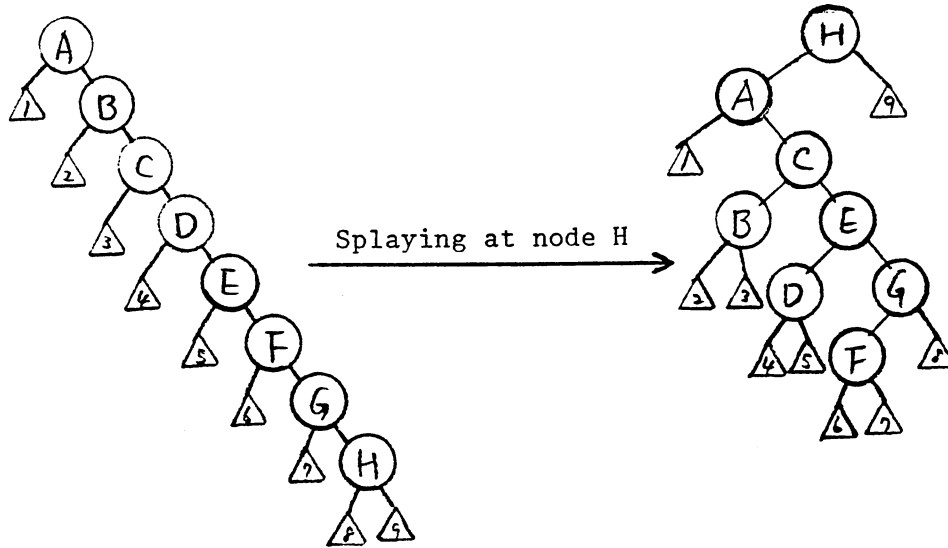
2. Test whether  $P(X)$  is the tree root. If the test is true, perform a single rotation (see Figure 18a) and terminate the operation. If the test is false, go to step (3).

3. Test whether  $P(X)$  and  $X$  are both left or both right children. If the test is true, perform two single rotations at the same time (see Figure 18b) and go to step (1). If the test is false (in this case  $X$  is a left child of  $P(X)$  and  $P(X)$  is a right child of  $P(P(X))$ , or vice versa), perform a double rotation (see Figure 18c) and go to step (1).

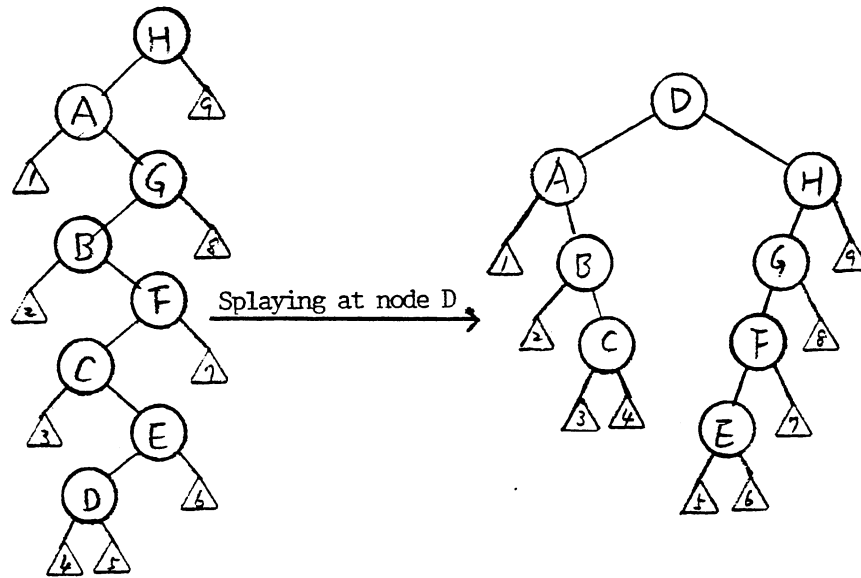
Splaying at node  $X$  of depth ( $d$ ) takes  $O(d)$  time, that is, time is proportional to the time to access at  $X$ . In the case of insertion and deletion (successful updating is

Figure 17

The Halving Effect of Splaying A Node in Binary Trees



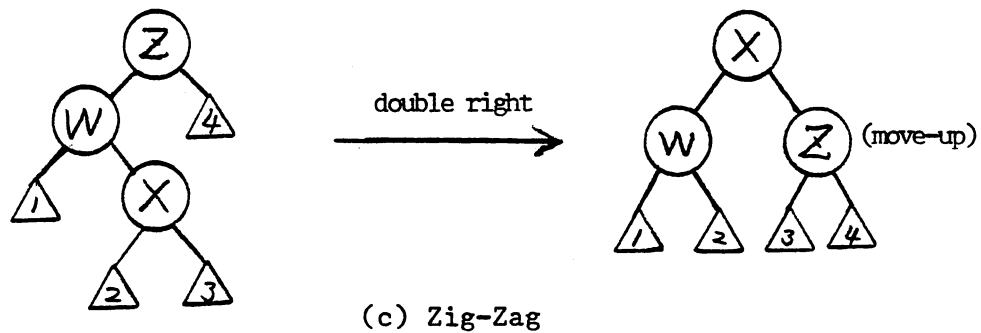
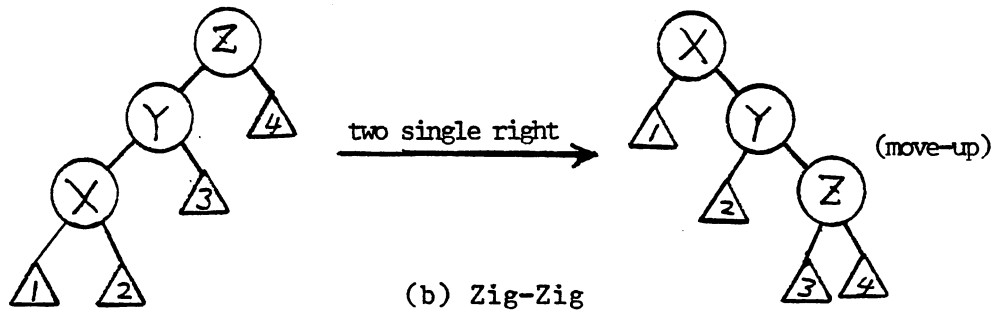
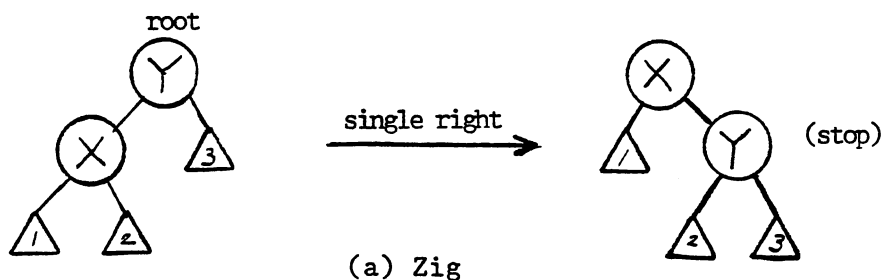
(a) All Zig-Zig splaying steps



(b) All Zag-Zag Splaying steps

△ denotes a subtree or a null node

Figure 18  
Bottom-Up Splaying Algorithm



Splaying at node X

Cases (b) and (c) are terminals when X is the root

△ denotes a subtree or a null node

assumed), Sleator and Tarjan (92) proposed: for insertion, simply splay at the inserted item; for deletion, we splay the parent of the deleted node. On the other hand, if the operation is unsuccessful (redundant), we splay at the last nonnull node reached during the search and return a null pointer. An example of splaying operation is shown in Figure 39 (in Appendix I).

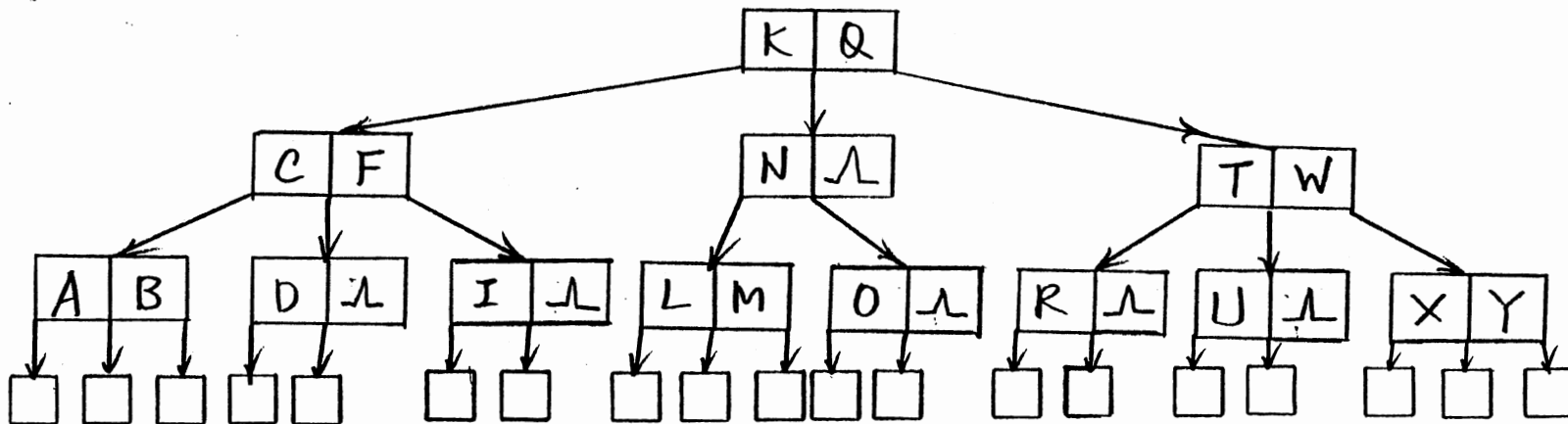
## 2.6 B-Trees

Bayer and McCreight (10) first introduced the B-tree data structure to organize and maintain large ordered indexes. Comer (22) gave extensive studies on the updating algorithm, cost of operations, multiuser environment, and applications for B-trees and their variants.

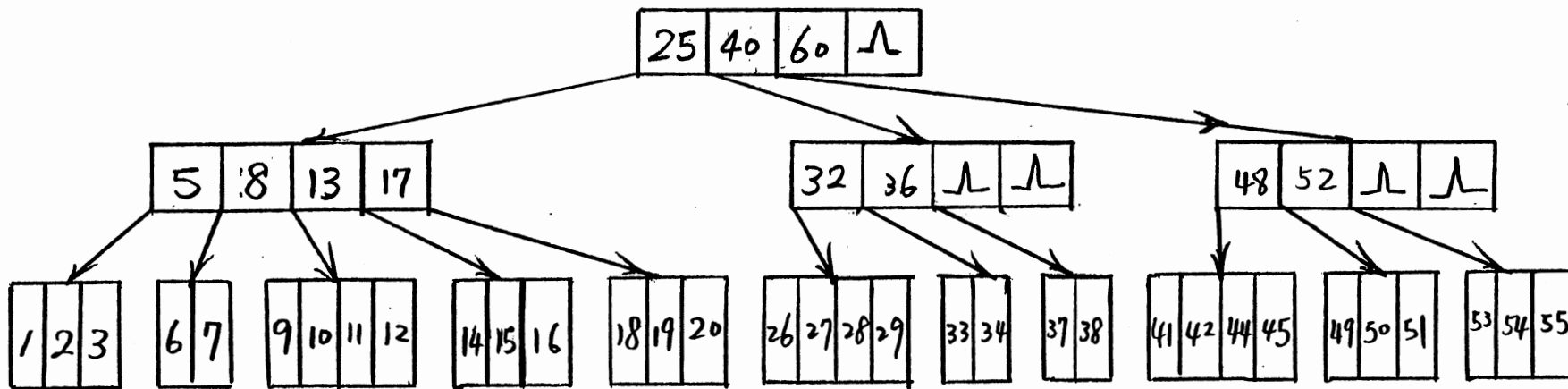
A B-tree of order  $m$  has the following properties:

1. All external nodes have the same depth.
2. Every internal node has at most  $m$  children.
3. Every internal node, except for the root, has at least  $\lceil m/2 \rceil$  children.
4. An internal node with  $k$  children contains  $k-1$  keys.
5. The root, except for a unary tree, has at least two children.

Figure 19 shows the examples of a 2-3 tree and a B-tree of order 5. The updating algorithms were well described in the survey paper of Comer and are not discussed here. This section concentrates on the B+-tree data structure, one of the B-tree's variant, because B+-trees are suitable to concurrent data processing. Comer summarized the properties



(a) A 2-3 Trees



(b) A B-Trees of Order 5

Figure 19 Examples of A 2-3 Tree and A B-Tree of Order 5

of B+-trees as follows.

1. All items (which contain key and data) are stored in the external nodes.

2. The internal nodes consist only of the index (no data), a road map to enable correct and fast access.

3. All external nodes are linked together, left to right, to provide easy sequential access.

4. During the deletion of a B<sup>+</sup>-tree, the internal index need not be changed as long as the external node remains at least half full.

Figure 20 shows an example of a B+-tree. The B+-tree data structure has advantages of supporting the same low operation cost,  $O(\log_d n)$ , as in the ordinary B-tree, and yet providing fast and easy sequential access,  $O(n)$  for a B+ tree and  $O(n \log_d n)$  for a B-tree.

## 2.7 Multidimensional Binary Search Trees

The search tree structures we have been discussing were limited to items which have only one key to identify themselves during the search operation. However in the large database environment, items are most likely associated with multiple keys. The multidimensional binary search tree (so called the K-d tree, where K is the dimension of associated keys) was first introduced by Bentley (13) as a data structure for storing multikey items.

The definitions and notations of a K-d tree are described as follows (homogeneous data structure is assumed).

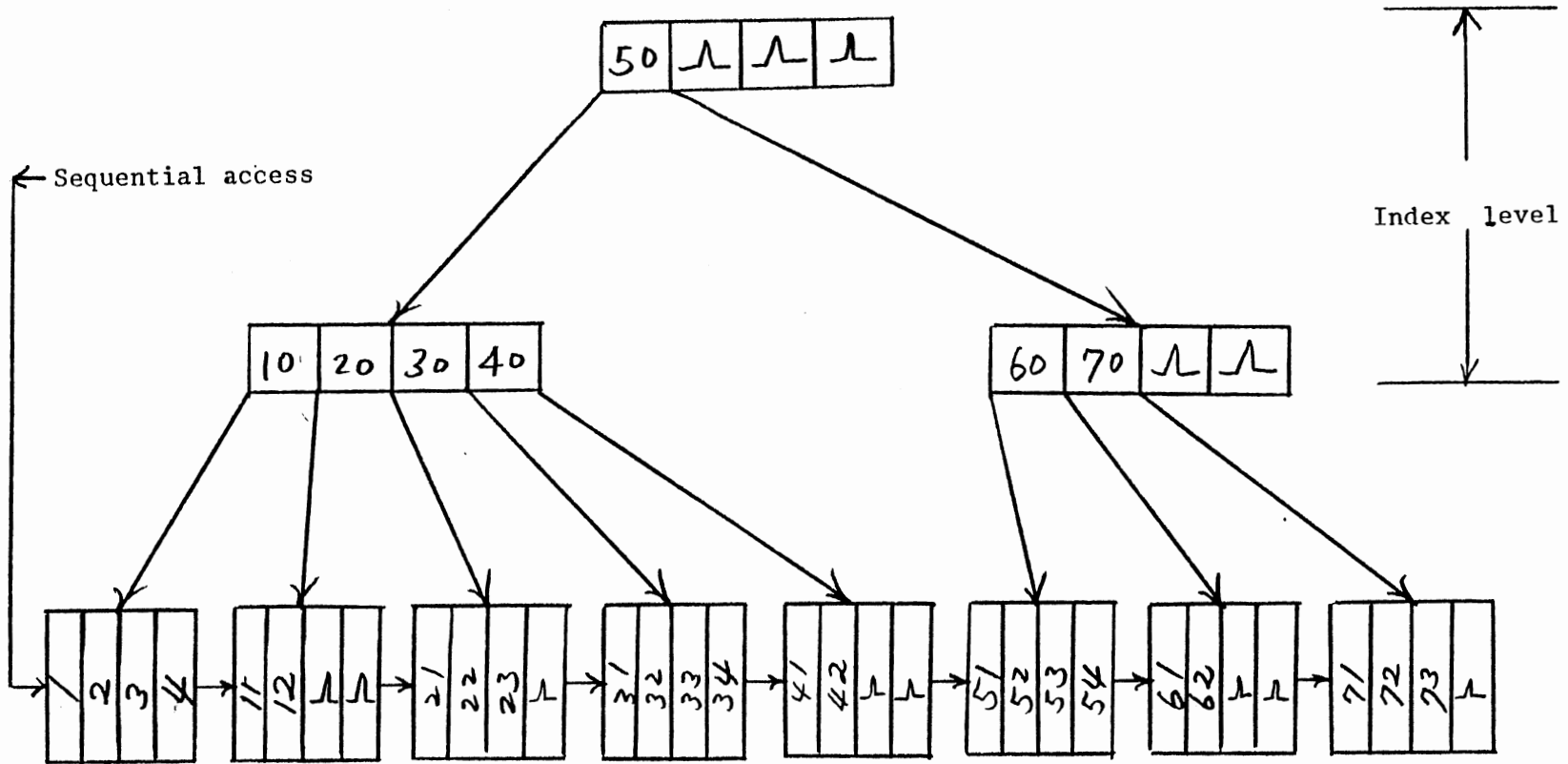


Figure 20 An Example of A B<sup>+</sup>-Tree

1. Every internal node contains  $k$  keys, and a discriminator, which is an integer between 0 and  $k-1$ .

2. All internal nodes on any given level have the same discriminator.

3. The root has discriminator 0, its two children have discriminator 1, and so on to the  $k$ -th level in which the discriminator is  $k-1$ ; the  $(k+1)$ -th level has discriminator 0, and the cycle repeats. Generally, an  $i$ th level has a discriminator of  $(i-1) \bmod k$ .

Before discussing the binary search in a  $K$ - $d$  tree, we need to define some notations. The  $K(P), \dots, K(P)$  denote the  $k$  keys in node  $P$ ,  $L(P)$  denotes the left child of  $P$ ,  $R(P)$  denotes the right child of  $P$ , and  $DISC(P)$  denotes the discriminator of  $P$ . The symmetric order imposed by a  $K$ - $d$  tree is: if  $P$  is an internal node and  $DISC(P)$  is  $j$ ; then for any node  $Q$  in the left subtree of  $P$ , it is true that  $K(Q) \leq K(P)$ ; likewise, for any node  $R$  in the right subtree of  $P$ , it is also true that  $K(R) > K(P)$ . (The equality of keys is possible in a  $K$ - $d$  tree.) The "search" operation in a  $K$ - $d$  tree refers to the ability to access any item in the tree by traversing down from the root, branching left if the  $j$ -th key of the accessed item is less than or equal to the  $j$ -th key of the current node, branching right if the  $j$ -th key of the accessed item is greater than the  $j$ -th key of the current node, or terminating the operation when every key in the accessed item is equal to the corresponding key in the current node.

Figure 21 shows an example of a  $K$ - $d$  tree with  $K=3$ .



Insertion sequence: (F,50,g), (D,20,e), (C,30,d), (G,90,d), (J,80,k), (A,10,a),  
 (K,20,b), (M,10,h), (B,20,b), (E,30,d), (I,99,i).

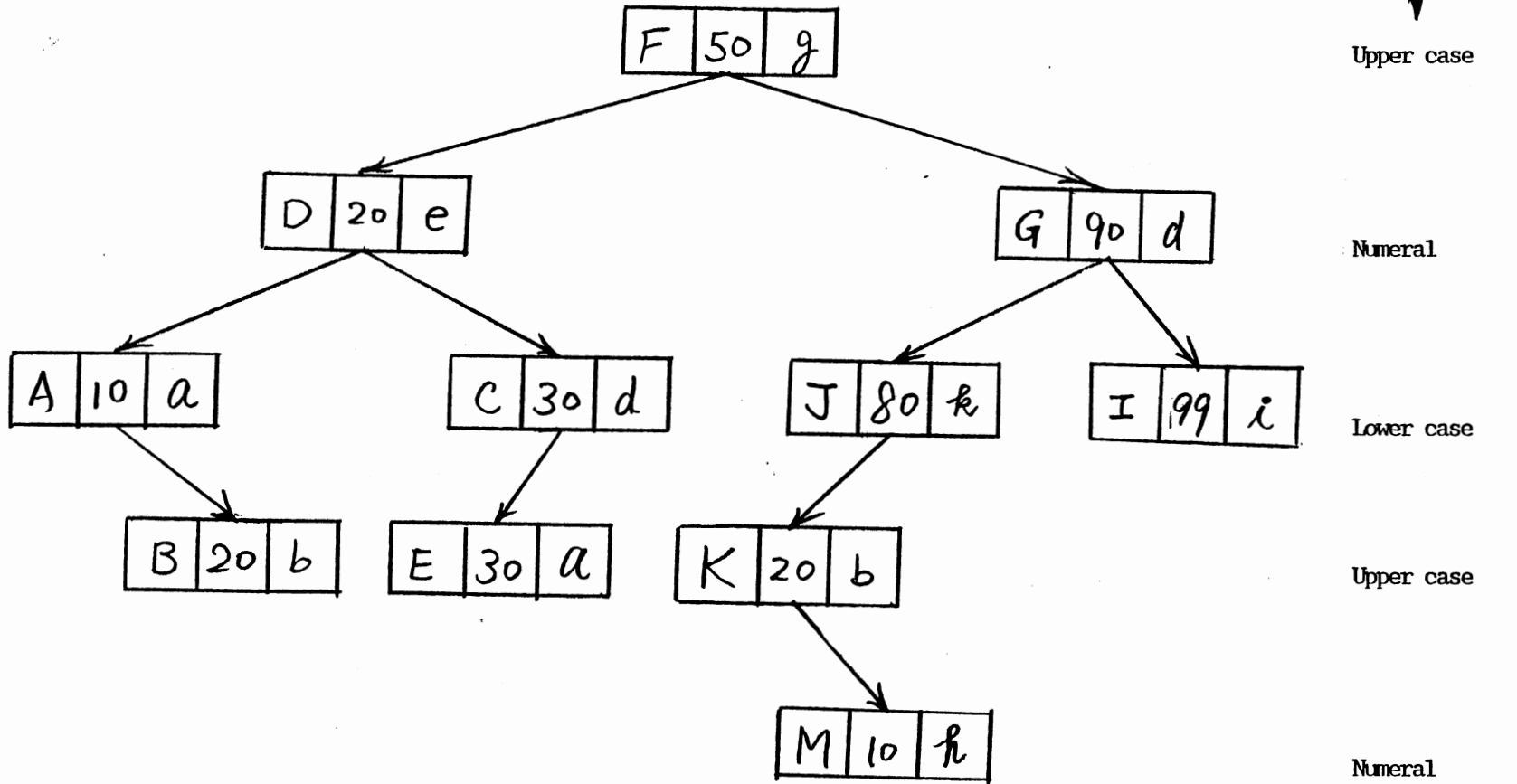


Figure 21 An Example of A K-d Tree with K=3

## CHAPTER III

### TOP-DOWN UPDATING AND CONCURRENT OPERATIONS

Recent advances in computer technology make the concurrent data processing in large database systems become practical (11,23,25,26,51,58,59,65). The idea of concurrent operation is that of allowing a maximum number of processes to operate on the tree without interfering with each other since it is unnatural and inefficient to restrict large database systems to sequential operation.

Guibas and Sedgewick (36) pointed out that the existence of purely top-down updating algorithms for balanced search trees is very important because a simple locking protocol can be used to enhance the concurrency. However, most updating algorithms for balanced search trees are bottom-up. Recently, Ottmann and et al. (74) proposed a purely top-down updating algorithm for stratified search trees and Tarjan (100) proposed updating algorithms for red black trees.

The advantages of a top-down update method are (1) it eliminates parent pointers (or a stack to hold the entire search path), and (2) it makes concurrent tree operation more efficient since one update operation need only lock a fixed number of tree nodes rather than lock the entire access path as in the bottom-up method.

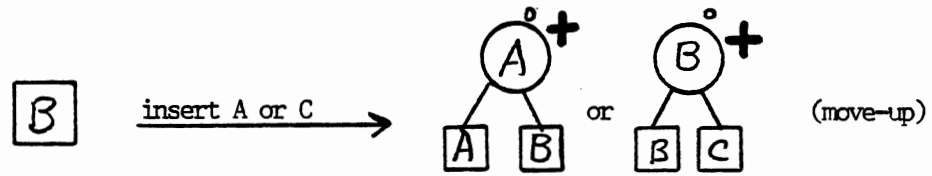
In the bottom-up updating, the arrangement of data is to store items (including key and data) in internal nodes, and external nodes are, essentially, null nodes (so called the homogeneous tree structure). With this arrangement, top-down insertion is not affected. However, top-down deletion becomes more difficult because we have to swap the internal node which is to be deleted and has two internal children, with its predecessor as we did previously in the bottom-up deletion. Thus, we need either extra pointers or a stack for a second pass along the access path. This produces problems when concurrent operations are allowed. Fortunately we can make another arrangement; that is to store items in external nodes and keys in internal nodes (the so called nonhomogeneous tree structure). An internal node acts as an index. Whenever a deletion is required, only the external node which contains that data item is deleted and the key in the internal node remains unchanged. This eliminates the swapping process and makes concurrent tree operation easier.

### 3.1 Top-Down Updating for HB(k)-Trees

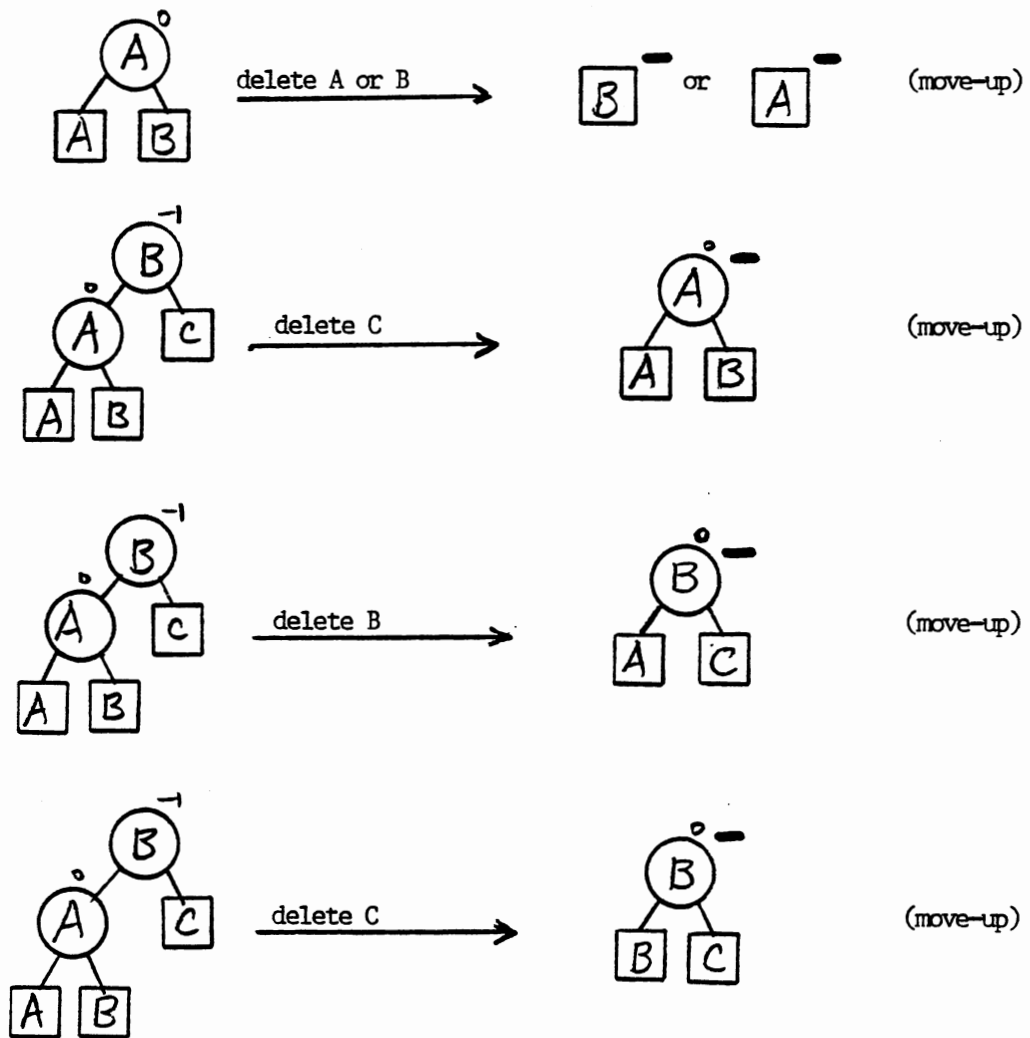
In this section, we propose new top-down updating algorithms for HB(k) trees that make the concurrent operations become possible. Figure 22 shows the updating of a nonhomogeneous tree structure for HB(k) trees. The rule for top-down insertion into HB(k) trees is to maintain a current node which can absorb the "long" without increasing its height. For example, if node Y has a balance-tag satisfying

Figure 22

## Updating A Nonhomogeneous Tree Structure of HB(k) Trees



(a) Insertion



(b) Deletion

$\overset{+}{\circ}$  denotes the "long"  
 $\overset{-}{\circ}$  denotes the "short"

$k \geq \text{tag}(Y) > 0$  and  $L(Y)$  is the next node to be visited (see Figure 23b), then  $Y$  can be selected as a current node because  $Y$  is right taller; an increase of the height of  $Y$ 's left subtree does not increase the height of  $Y$  (only the balance-tag is changed). Initially, we let the root node be the current node  $X$ . We then traverse down along the search path and take actions with respect to the following cases (algorithm HBTI).

1. When an external node is reached, proceed with the same steps as in the bottom-up insertion (algorithm HBI). (The rebalancing terminates when the current node  $X$  is reached bottom-up.)

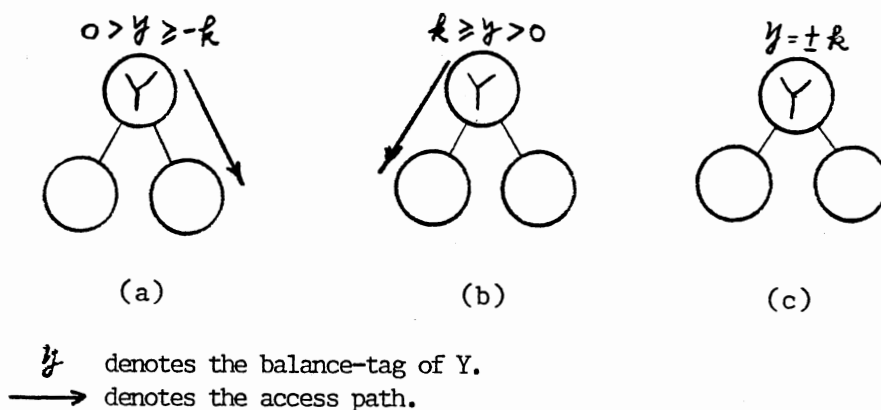


Figure 23

Current Node Selection in the Top-Down  
Insertion of  $HB(k)$  Trees

2. When a node, say  $Y$ , which has a balance-tag satisfying  $0 > \text{tag}(Y) \geq -k$  and  $R(Y)$  is the next node to be visited (see Figure 23a) is encountered, we let  $Y$  be the new current node  $X$  and continue traversing down along the access path. (Symmetric variant is shown in Figure 23b.)

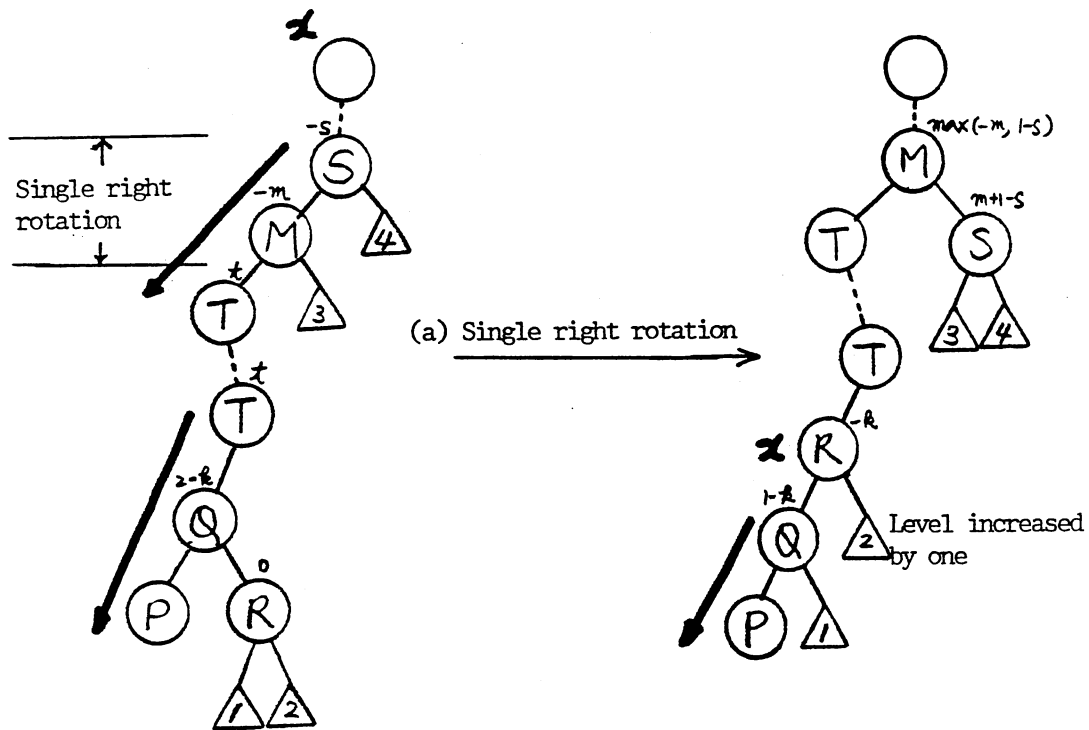
3. When a node, say  $Y$ , which has a balance-tag of  $|\text{tag}(Y)| = k$  (see Figure 23c) is encountered, we let  $Y$  be the new current node  $X$  and continue traversing down along the access path.

4. When several successive nodes which cannot be selected as a current node and the balance-tags of those nodes satisfy one of the cases in Figure 24 are encountered, we prebalance the tree, update balance-tags, select a new current node, and continue traversing down along the access path.

Case 4 takes two rotations for prebalancing a tree, however, it unlocks many nodes which can be updated by other users in the concurrent environment. We then conclude that the top-down insertion takes  $O(1)$  rotations in the amortized case. An example of top-down insertion for an AVL-tree is shown in Figure 41 and other examples of top-down insertion for an  $HB(k)$  tree are shown in Figure 42 (in Appendix B).

Figure 24

## Prebalancing in the Top-Down Insertion of HB(k) Trees



$\longrightarrow$  denotes the access path,  $\times$  denotes the current node,  
 $k, m, s, t$  denote balance-tags.

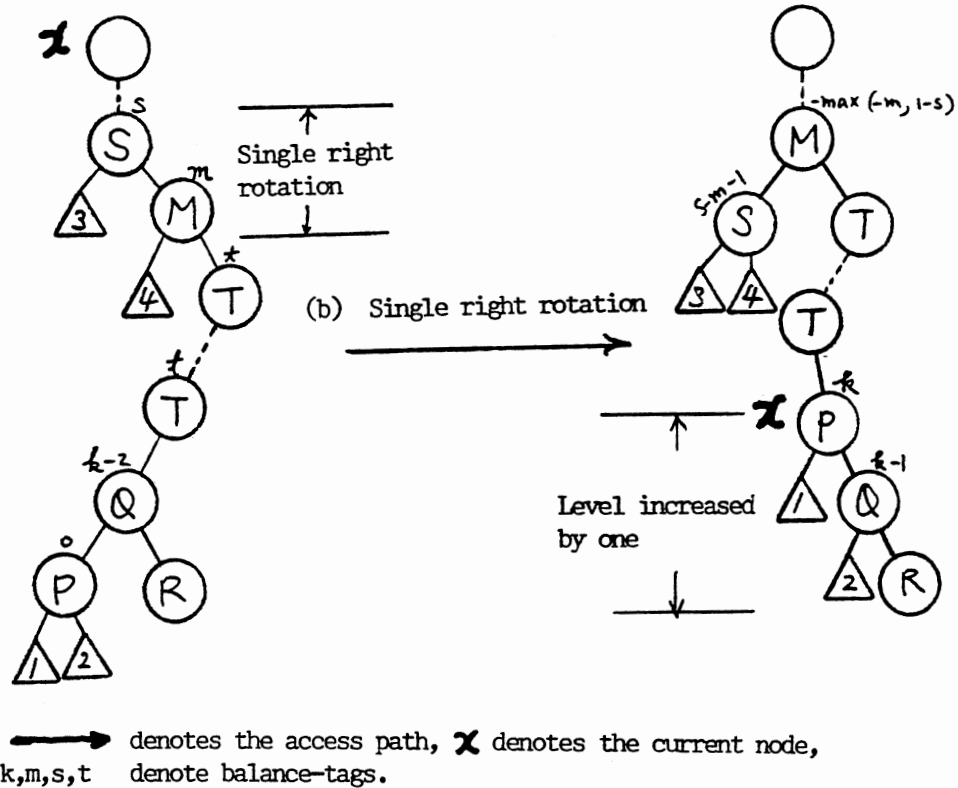
Balance-tags satisfy:

1.  $0 > \text{tag}(S) > -k$ .
2.  $0 > \text{tag}(M) > -k$ .
3.  $0 \geq \text{tag}(T) > -k$  & L(T) is the next node to be visited.  
 or  
 $k > \text{tag}(T) \geq 0$  & R(T) is the next node to be visited.
4.  $\text{tag}(Q) = 2-k$  &  $\text{tag}(R) = 0$ ,  
 or  
 $\text{tag}(Q) = k-2$  &  $\text{tag}(P) = 0$ .

Balance-tags update:

1.  $\text{tag}(S), -s \rightarrow m+1-s$ .
2.  $\text{tag}(M), -m \rightarrow \max(-m, 1-s)$ .
3.  $\text{tag}(T), t \rightarrow t+1$ , if R(T) is the next node to be visited.  
 or  
 $\text{tag}(T), t \rightarrow t-1$ , if L(T) is the next node to be visited.
4.  $\text{tag}(Q), 2-k \rightarrow 1-k$ ,  
 $\text{tag}(R), 0 \rightarrow -k$  & R is the new current node.  
 or  
 $\text{tag}(Q), k-2 \rightarrow k-1$ ,  
 $\text{tag}(P), 0 \rightarrow k$  & P is the new current node.

Figure 24 (Continued)



Balance-tags satisfy:

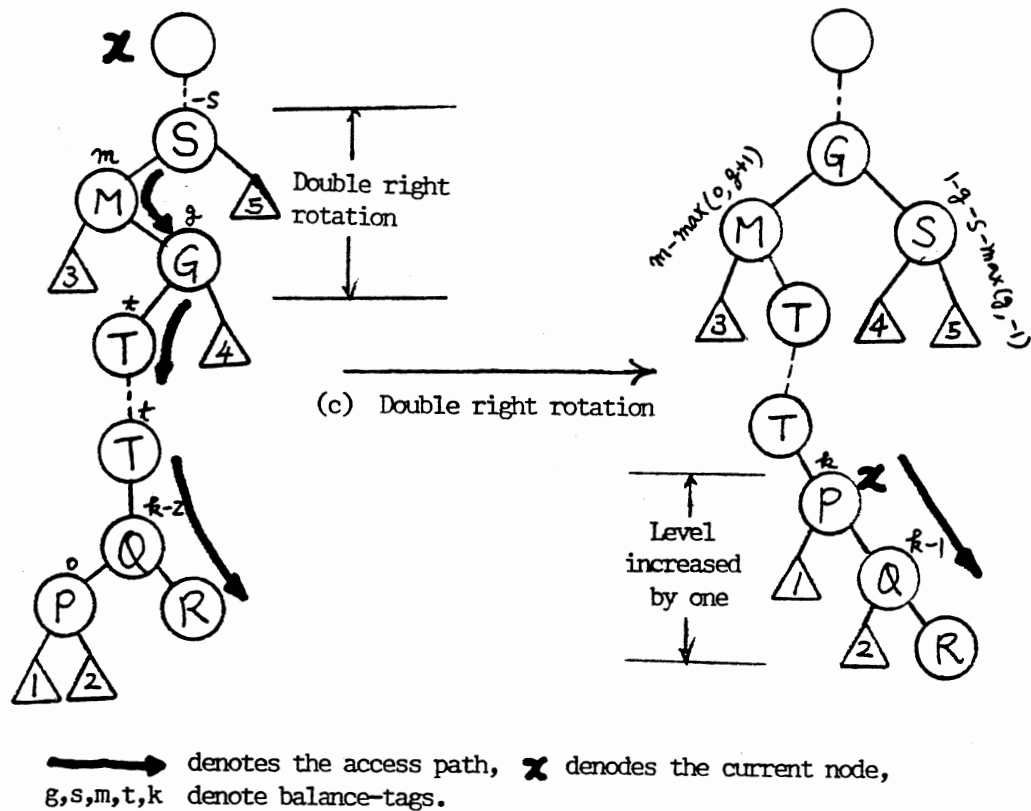
1.  $k > \text{tag}(S) > 0$ .
2.  $k > \text{tag}(M) > 0$ .
3.  $0 \geq \text{tag}(T) > -k$  & L(T) is the next node to be visited.  
 or  
 $k > \text{tag}(T) \geq 0$  & R(T) is the next node to be visited.
4.  $\text{tag}(Q) = 2-k$  &  $\text{tag}(R) = 0$ ,  
 or  
 $\text{tag}(Q) = k-2$  &  $\text{tag}(P) = 0$ .

Balance-tags update:

1.  $\text{tag}(S)$ ,  $s \rightarrow s-m-1$ .
2.  $\text{tag}(M)$ ,  $m \rightarrow -\max(-m, 1-s)$ .
3.  $\text{tag}(T)$ ,  $t \rightarrow t+1$ , if R(T) is the next node to be visited.  
 or  
 $\text{tag}(T)$ ,  $t \rightarrow t-1$ , if L(T) is the next node to be visited.
4.  $\text{tag}(Q)$ ,  $2-k \rightarrow 1-k$ ,  
 $\text{tag}(R)$ ,  $0 \rightarrow -k$ , & R is the new current node.  
 or  
 $\text{tag}(Q)$ ,  $k-2 \rightarrow k-1$ ,  
 $\text{tag}(P)$ ,  $0 \rightarrow k$ , & P is the new current node.



Figure 24 (Continued)



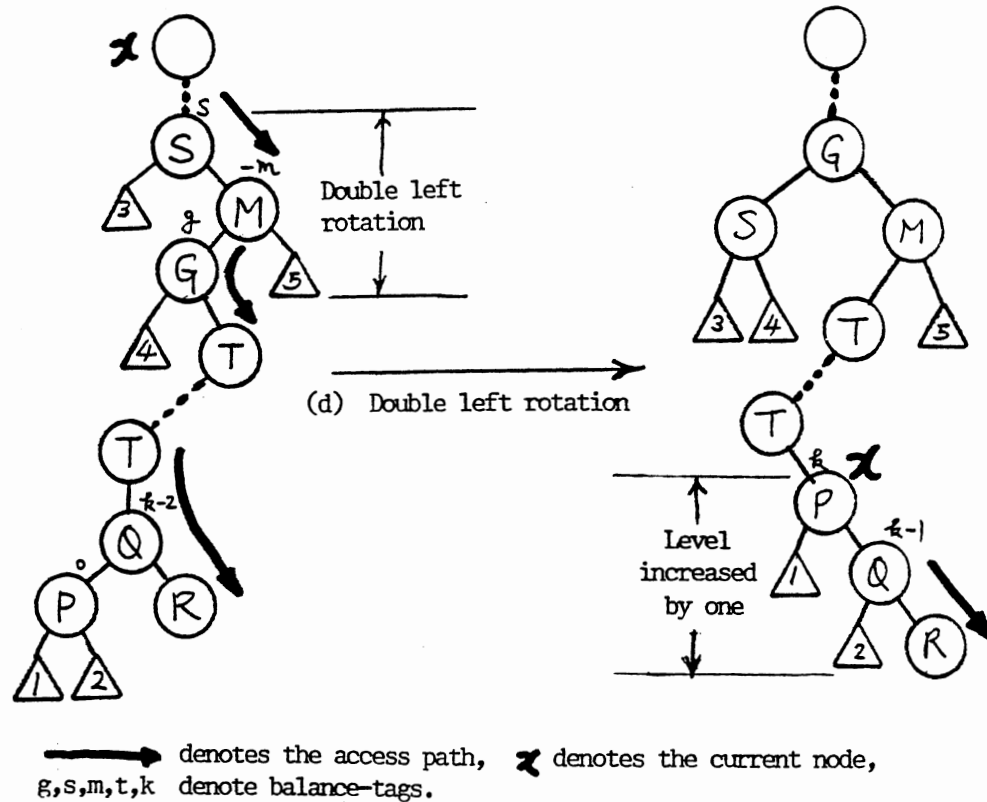
Balance-tags satisfy:

1.  $0 > \text{tag}(S) > -k.$
2.  $k > \text{tag}(M) > 0.$
3.  $k > \text{tag}(G) > -k.$
4.  $0 \geq \text{tag}(T) > -k$  & L(T) is the next node to be visited.  
 or  
 $k > \text{tag}(T) \geq 0$  & R(T) is the next node to be visited.
5.  $\text{tag}(Q) = 2-k$  &  $\text{tag}(R) = 0,$   
 or  
 $\text{tag}(Q) = k-2$  &  $\text{tag}(P) = 0.$

Balance-tags update:

1.  $\text{tag}(S), s \rightarrow 1-g-s-\max(g,-1).$
2.  $\text{tag}(M), m \rightarrow m-\max(0,g+1).$
3.  $\text{tag}(G), \begin{cases} g \rightarrow 1-\max(1-m,-g) & \text{if } g \geq 0, \\ g \rightarrow -1-\max(g,2-s) & \text{if } g < 0. \end{cases}$
4.  $\text{tag}(T), t \rightarrow t+1,$  if R(T) is the next node to be visited.  
 or  
 $\text{tag}(T), t \rightarrow t-1,$  if L(T) is the next node to be visited.
5.  $\text{tag}(Q), 2-k \rightarrow 1-k,$   
 $\text{tag}(R), 0 \rightarrow -k,$  & R is the new current node.  
 or  
 $\text{tag}(Q), k-2 \rightarrow k-1,$   
 $\text{tag}(P), 0 \rightarrow k,$  & P is the new current node.

Figure 24 (Continued)



Balance-tags satisfy:

1.  $k > \text{tag}(S) > 0$ .
2.  $0 > \text{tag}(M) > -k$ .
3.  $k > \text{tag}(G) > -k$ .
4.  $0 \geq \text{tag}(T) > -k$  &  $L(T)$  is the next node to be visited.  
 or  
 $k > \text{tag}(T) \geq 0$  &  $R(T)$  is the next node to be visited.
5.  $\text{tag}(Q) = 2-k$  &  $\text{tag}(R) = 0$ ,  
 or  
 $\text{tag}(Q) = k-2$  &  $\text{tag}(P) = 0$ .

Balance-tags update:

1.  $\text{tag}(S), \quad s \rightarrow s-1-\max(0, g+1)$ .
2.  $\text{tag}(M), \quad -m \rightarrow g-m-\max(g, -1)$ .
3.  $\text{tag}(G), \quad g \rightarrow 1-\max(-g, 2-s)$  if  $g \geq 0$ ,  
 $g \rightarrow -1-\max(g, 1-m)$  if  $g < 0$ .
4.  $\text{tag}(T), \quad t \rightarrow t+1$ , if  $R(T)$  is the next node to be visited.  
 or  
 $\text{tag}(T), \quad t \rightarrow t-1$ , if  $L(T)$  is the next node to be visited.
5.  $\text{tag}(Q), \quad 2-k \rightarrow 1-k$ ,  
 $\text{tag}(R), \quad 0 \rightarrow -k$ , &  $R$  is the new current node.  
 or  
 $\text{tag}(Q), \quad k-2 \rightarrow k-1$ ,  
 $\text{tag}(P), \quad 0 \rightarrow k$ , &  $P$  is the new current node.

The rule for top-down deletion from  $HB(k)$ -trees is to maintain a current node which can absorb the "short" without decreasing its height. For example, if node  $Y$  has a balance-tag satisfying  $k > \text{tag}(Y) \geq 0$  and  $L(Y)$  is the next node to be visited (see Figure 25b), then  $Y$  can be selected as a current node because  $Y$  is right heavier; a decrease of the height of  $Y$ 's left subtree does not decrease the height of  $Y$  (only the balance-tag is changed). Initially, we let the root be the current node  $X$ . We then traverse down along the search path and take actions with respect to the following cases (algorithm HBTD).

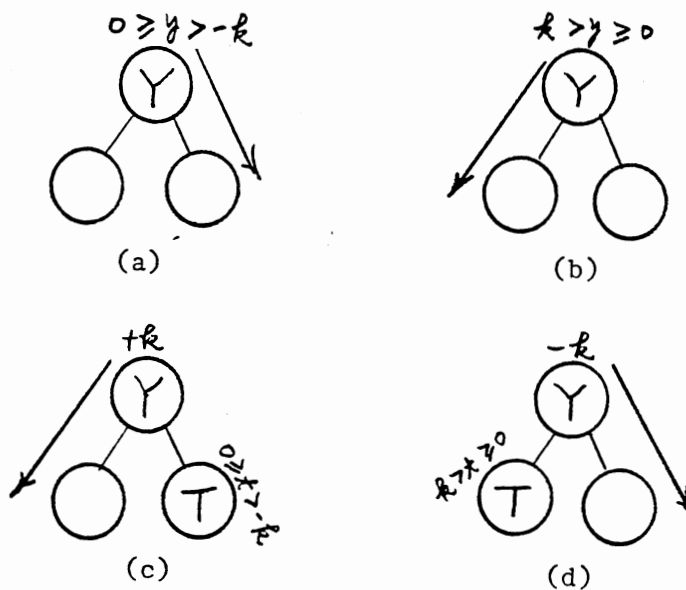
1. When an external node is reached, proceed with the same steps as in the modified bottom-up insertion (algorithm HBMD). (The rebalancing terminates when the current node  $X$  is reached bottom-up.)

2. When a node, say  $Y$ , which has a balance-tag satisfying  $0 \geq \text{tag}(Y) > -k$  and  $R(Y)$  is the next node to be visited (see Figure 25a) is encountered, we let  $Y$  be the new current node  $X$  and continue traversing down along the access path. (Symmetric variant is shown in Figure 25b.)

3. When a node, say  $Y$ , which has a balance-tag satisfying  $\text{tag}(Y) = k$  and  $R(Y)$  has a balance-tag of  $0 \geq \text{tag}(R(Y)) > -k$  and  $L(Y)$  is the next node to be visited (see Figure 25c) is encountered, we let  $Y$  be the new current node  $X$  and continue traversing down along the access path. (Symmetric variant is shown in Figure 25d.)

4. When several successive nodes are encountered which cannot be selected as a current node and the balance-tags of

those nodes satisfy one of the cases in Figures 26. we pre-balance the tree, update balance-tags, select a new current node, and continue traversing down along the access path.



$y, t$  denote the balance-tag of Y, T.

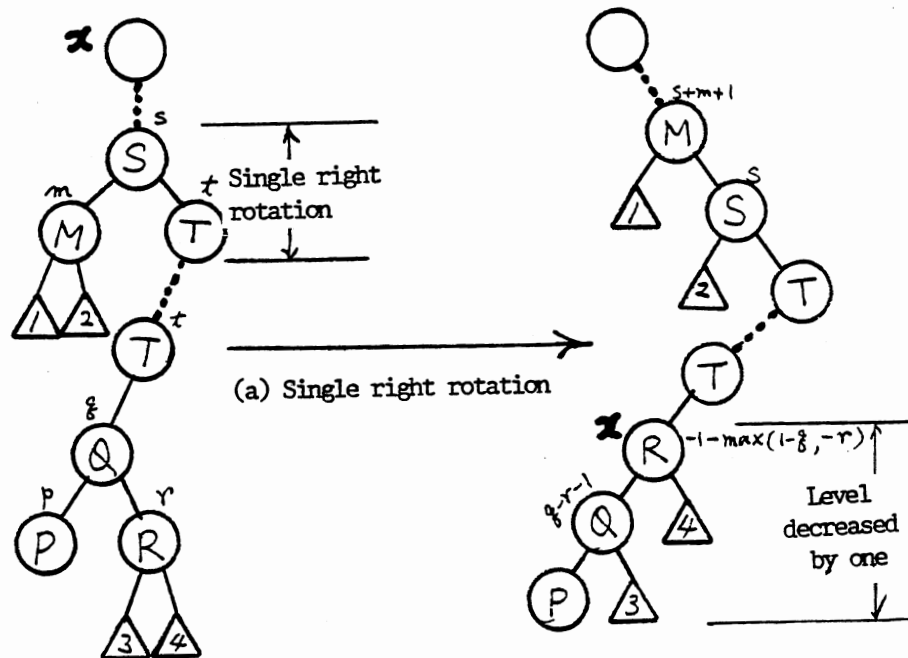
$\longrightarrow$  denotes the access path

Figure 25

Current Node Selection in the Top-Down  
Deletion of HB(k) Trees

Figure 26

## Prebalancing in the Top-Down Deletion of HB(k) Trees



denotes the access path,  
 $P, Q, R, S, M, T$  denote balance-tags.

denotes the current node,

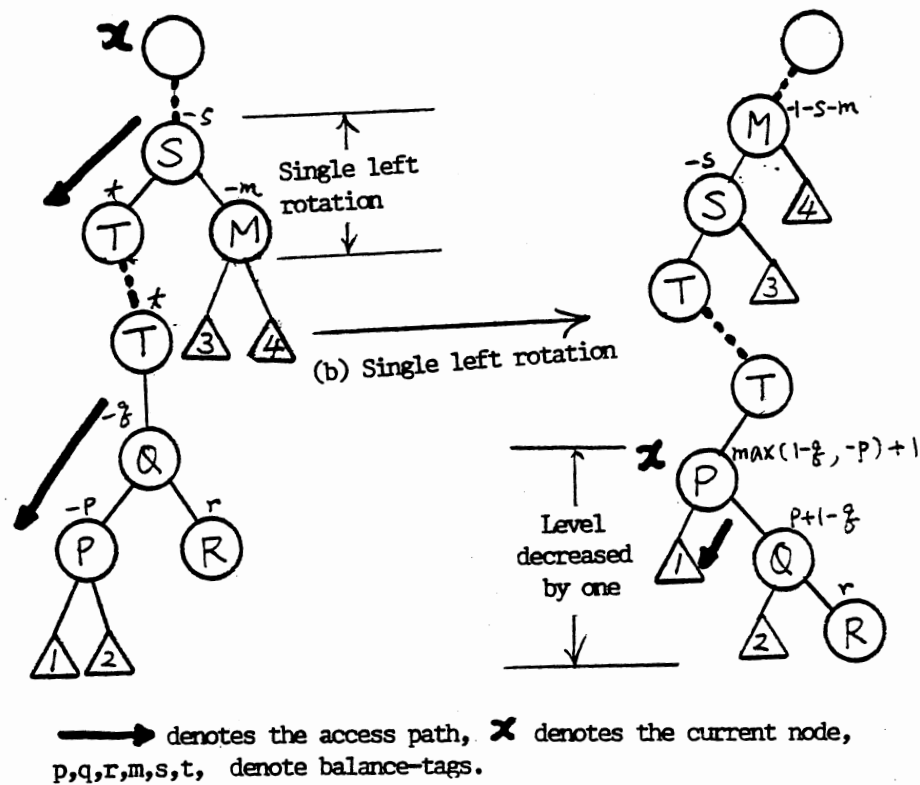
Balance-tags satisfy:

1.  $k > \text{tag}(S) > 0$ .
2.  $k > \text{tag}(M) > 0$  &  
 $k > (\text{tag}(S) + \text{tag}(M)) > 0$ .
3.  $0 \geq \text{tag}(T) > -k$  &  $L(T)$  is the  
 next node to be visited.  
 or  
 $k > \text{tag}(T) \geq 0$  &  $R(T)$  is the  
 next node to be visited.
4.  $-2 \geq \text{tag}(Q) > -k$  &  
 $-1 \geq \text{tag}(P) > -k$  & either  $\text{tag}(Q) = -2$   
 or  $\text{tag}(P) = -1$ .  
 or  
 $k > \text{tag}(Q) \geq 2$  &  $k > \text{tag}(R) \geq 1$  &  
 either  $\text{tag}(Q) = 2$  or  $\text{tag}(R) = 1$ .

Balance-tags update:

1.  $\text{tag}(S), +s \rightarrow +s$ .
2.  $\text{tag}(M), +m \rightarrow +1+s+m$ .
3.  $\text{tag}(T), t \rightarrow t-1$ , if  $R(T)$  is the next  
 node to be visited.  
 or  
 $\text{tag}(T), t \rightarrow t+1$ , if  $L(T)$  is the next  
 node to be visited.
4.  $\begin{cases} \text{tag}(Q), -q \rightarrow \max(1-q, -p)+1. \\ \text{tag}(P), -p \rightarrow p+1-q. \end{cases}$   
 $P$  is the new current node.  
 or  
 $\begin{cases} \text{tag}(Q), q \rightarrow q-r-1. \\ \text{tag}(R), r \rightarrow -1-\max(1-q, -r). \end{cases}$   
 $R$  is the new current node.

Figure 26 (Continued)



## Balance-tags satisfy:

1.  $0 > \text{tag}(S) > -k$ .
2.  $0 > \text{tag}(M) > -k$  &  
 $0 > (\text{tag}(S) + \text{tag}(M)) > -k$ .
3.  $0 \geq \text{tag}(T) > -k$  &  $L(T)$  is the  
 next node to be visited.  
 or  
 $k > \text{tag}(T) \geq 0$  &  $R(T)$  is the  
 next node to be visited.
4.  $-2 \geq \text{tag}(Q) > -k$  &  
 $-1 \geq \text{tag}(P) > -k$  & either  $\text{tag}(Q) = -2$   
 or  $\text{tag}(P) = -1$ .
- or  
 $k > \text{tag}(Q) \geq 2$  &  $k > \text{tag}(R) \geq 1$  &  
 either  $\text{tag}(Q) = 2$  or  $\text{tag}(R) = 1$ .

## Balance-tags update:

1.  $\text{tag}(S), -s \rightarrow -s$ .
2.  $\text{tag}(M), -m \rightarrow -1 - m - s$ .
3.  $\text{tag}(T), t \rightarrow t - 1$ , if  $R(T)$  is the next  
 node to be visited.  
 or  
 $\text{tag}(T), t \rightarrow t + 1$ , if  $L(T)$  is the next  
 node to be visited.
4.  $\text{tag}(Q), -q \rightarrow \max(1 - q, -p) + 1$ ,  
 $\text{tag}(P), -p \rightarrow p + 1 - q$ ,  
 $P$  is the new current node.  
 or  
 $\text{tag}(Q), q \rightarrow q - r - 1$ ,  
 $\text{tag}(R), r \rightarrow -1 - \max(1 - q, -r)$ ,  
 $R$  is the new current node.

Case 4 takes two rotations for prebalancing a tree; however, it unlocks many nodes which can be updated by other users in the concurrent environment. We then conclude that the top-down deletion takes  $O(1)$  rotations in the amortized case. An example of top-down deletion for an AVL-tree is shown in Figure 43 and other examples of top-down deletion for an HB(k)-tree are shown in Figure 44 (in Appendix B).

### 3.2. Top-Down Updating for Red-Black Trees

Guibas and Sedgwick (36) have proposed  $O(\log n)$  rotations of top-down update algorithm for red-black trees. Tarjan (100) modified his own bottom-up update algorithms into virtual top-down update algorithms that require only  $O(1)$  rotations and  $O(1)$  color changes in the amortized case.

Figure 27 shows the updating of nonhomogeneous tree structure for red-black trees. The rule of top-down insertion for red-black trees is to maintain a black current node, say  $X$ , which can remove the red constraint violation. For example, a node has at least one black child can be selected as a current node (review Figure 8). Initially, we let the root node be the current node  $X$  and color it black if it is red and color both its children black if both are red. This step does not violate any constraint. We then traverse from the current node  $X$  down along the search path and take actions with respect to the following cases (algorithm RBTI).

1. When an external node is reached, proceed with the same steps as in the bottom-up ~~deletion~~ (algorithm RBI).

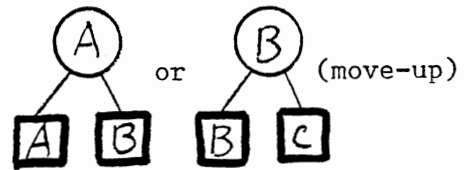
*insertion*

Figure 27

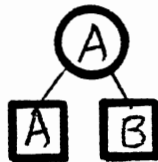
Updating a Nonhomogeneous Tree Structure of Red-Black Trees



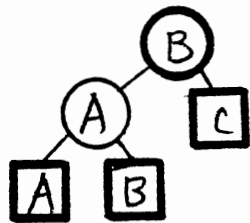
insert A or C



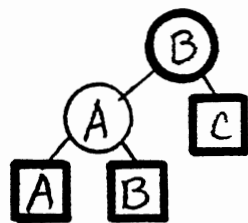
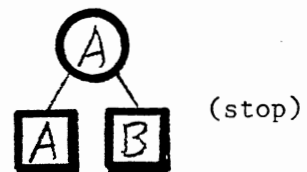
(a) Insertion



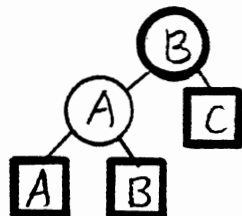
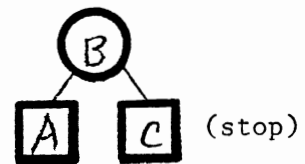
delete A or B



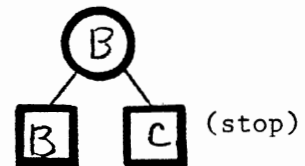
delete C



delete B



delete A



(b) Deletion

- denotes a black node
- denotes a red node
- denotes a "short"



(The rebalancing terminates when current node X is reached bottom-up.)

2. When a black node, say Y, with at least one black child is encountered, we let node Y be the new current node X and continue traversing down along the access path.

~~3. When a black node, say Y, with two red children and a black parent is encountered, we let node Y be the new current node X and continue traversing down along the access path.~~

3. When four successive black nodes, each of which has two red children, are encountered along the access path, we perform color changes (see Figure 28). If this violates the red constraint (see Figure 28a), proceed with the same steps as in the bottom-up insertion (algorithm RBI) until node X is reached. We then let the child (along access path) of Z be the new current node X and continue traversing down along the access path.

A disadvantage of this top-down insertion (algorithm RBTI) is that it may require several rotations rather than one rotation as in the bottom-up insertion (algorithm RBI); however, it still takes  $O(1)$  rotations and color changes in the amortized case (100). An example of top-down insertion for a red-black tree is shown in Figure 45 (in Appendix B).

The rule of top-down deletion for red-black trees is to maintain a current node which can absorb the "short" without violating the black constraint. Initially, we let the root node be the current node X and color X red if X has two black children. This step does not violate any constraint.

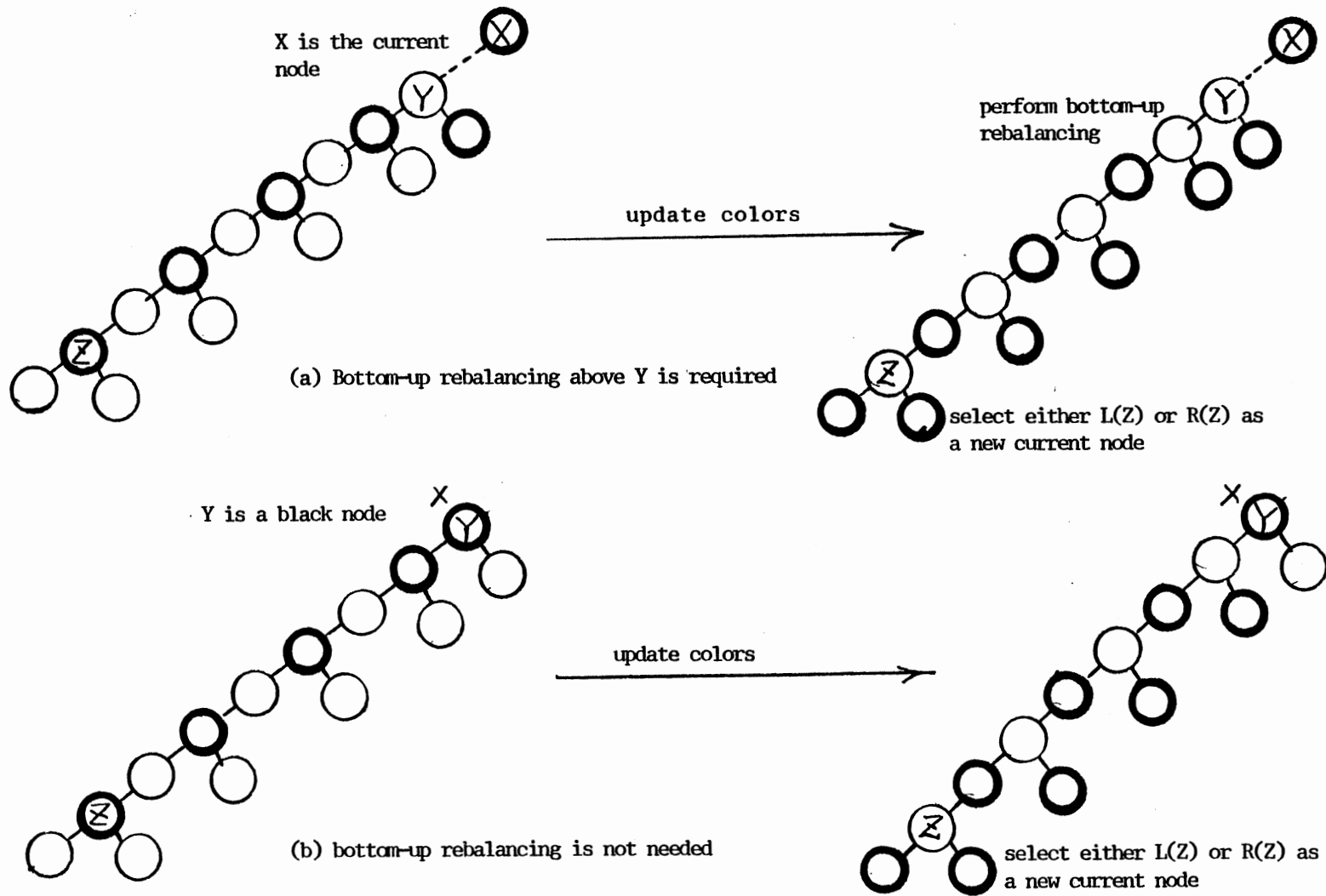


Figure 28 Current Node Selection in the Top-Down Insertion of Red-Black Trees

We then traverse from the current node X down along the access path and take actions with respect to the following cases (algorithm RBTD).

1. When an external node is encountered, proceed with the same steps as in the bottom-up deletion (algorithm RBD). (The rebalancing terminates when current node X is reached bottom-up.)

2. When a node, say Y, that is red or has a red child or grandchild is encountered, we let node Y be the new current node X and continue traversing down along the access path.

3. When three successive black nodes, each having all black children and grandchildren are encountered along the access path, we perform color changes (see Figure 29) and produce a "short" at node Y. We then follow the same method as in bottom-up deletion (algorithm RBD) to eliminate the "short". (The rebalancing terminates when node X is reached bottom-up.) We then let node Z be the new current node X and continue traversing down along the access path.

An example of top-down deletion for a red-black tree is shown in Figure 46 (in Appendix B).

### 3.3 Top-Down Updating for WB-Trees

In this section we propose a new top-down updating algorithm which is conceptually easy and has little computation overhead. The method is described as follows. In order to compute the weight-balance factor, we assign a RANK field to each internal node. The  $RANK(T)$  of node T is the

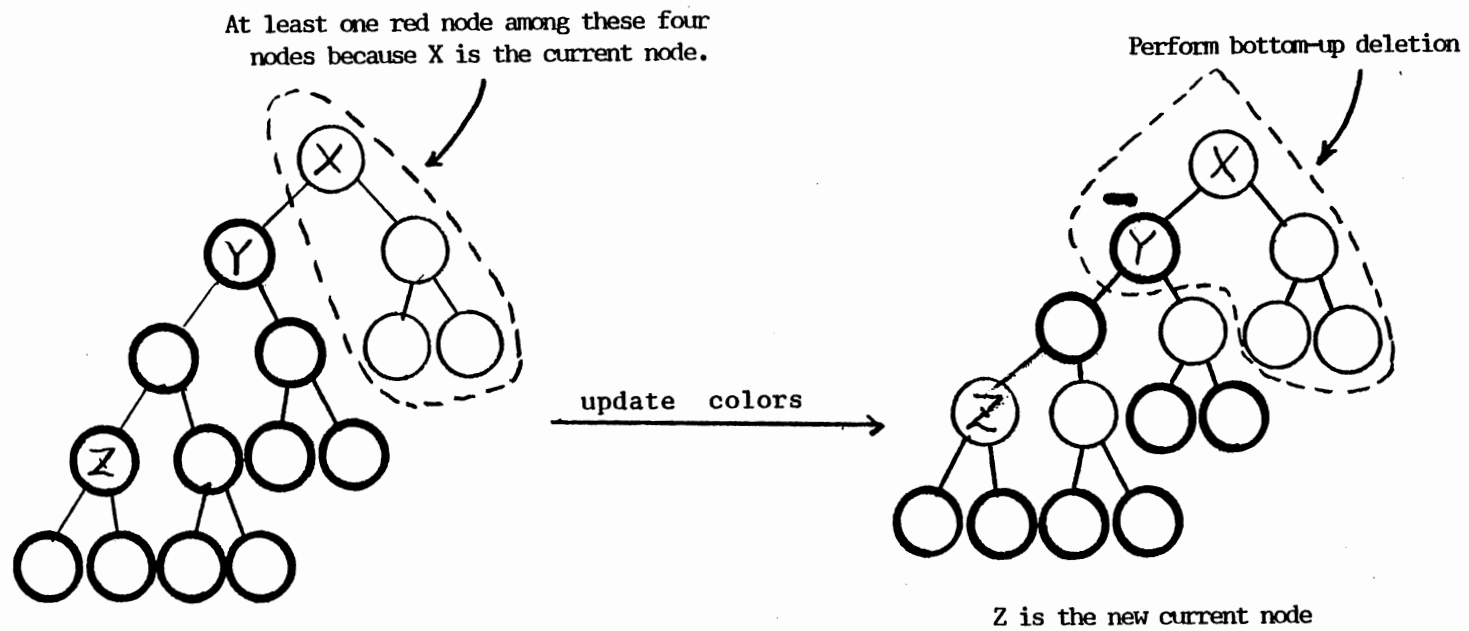


Figure 29  
 Current Node Selection in the Top-Down  
 Deletion of Red-Black Trees

total number of external nodes in T's left subtree. A tree header H which regards tree T as its left subtree also has a RANK field and points to the tree root. To update an item (either insertion or deletion), initially we update (add one to the RANK field if it is an insertion, subtract one from the RANK field if it is a deletion) the RANK(H) of tree header, and let the tree root be the current node T and  $N_t$  be the RANK(H). We then traverse down along the access path and proceed with the following steps (algorithm WBMT).

1. Test whether T is an external node. If the test is true; we replace T with a new node X containing the new item, set RANK(X) to 1, and stop. If the test is false; we compute the weight-balance factor at T, (where  $B(T) = \text{RANK}(T)/N_t$ ) and go to step (2).

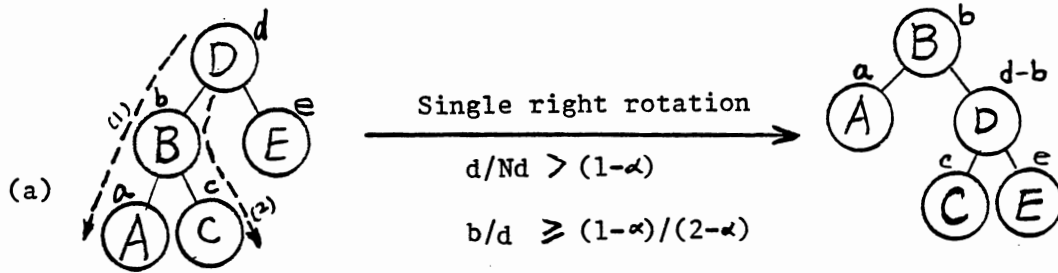
2. Test whether  $(1-\alpha) \geq B(T) \geq \alpha$  holds. If it is true we update the RANK field, select the child of T along the access path as a new current node, and go to step (1). If the test is false (violation occurs), go to step (3).

3. Test whether  $B(T) > (1-\alpha)$  holds. If the test is true; we perform a single right rotation (see Figure 30a) if  $B(T_L) \geq (1-\alpha)/(2-\alpha)$  or a double right rotation (see Figure 30b) if  $B(T_L) < (1-\alpha)/(2-\alpha)$ , update RANK fields, select a new current node, and go to step (1). If the test is false, go to step (4).

4. Now  $B(T) < \alpha$  holds. We perform a single left rotation (see Figure 30c) if  $B(T_r) \leq 1/(2-\alpha)$  or a double left rotation (see Figure 30d) if  $B(T_r) > 1/(2-\alpha)$ , update RANK fields, select a new current node, and go to step (1).

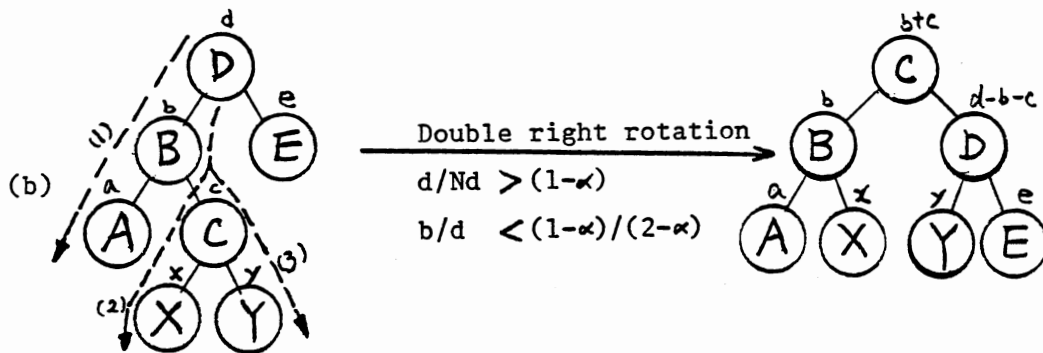
Figure 30

## Modified Top-Down Update Algorithm for Weight-Balanced Trees



Path (1): 1. update d and b, 2. compute b/d,  
3. A is the new current node,  
4.  $N_a = b$ .

Path (2): 1. update d, 2. compute b/d,  
3. C is the new current node,  
4.  $N_c = d - b$ .

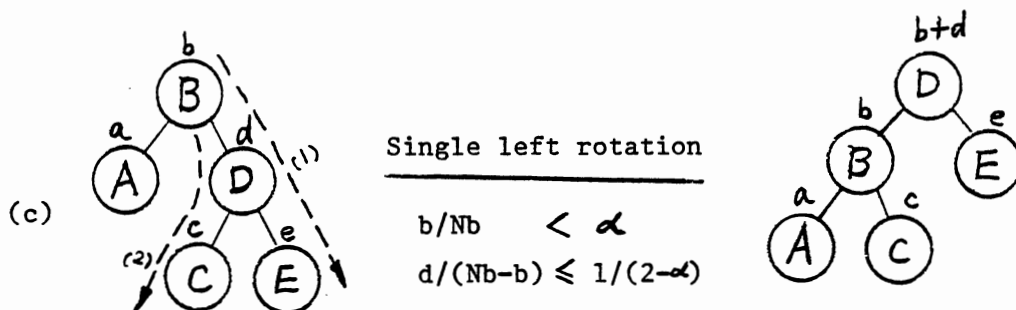


Path (1): 1. update d and b, 2. compute b/d,  
3. A is the current node,  
4.  $N_a = b$ .

Path (2): 1. update d and c, 2. compute b/d,  
3. X is the current node,  
4.  $N_x = c$ .

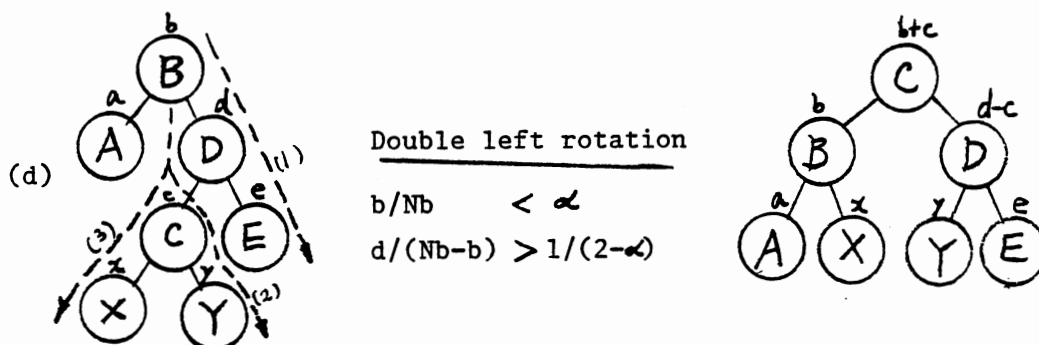
Path (3): 1. update d, 2. compute b/d,  
3. Y is the current node,  
4.  $N_y = d - b - c$ .

Figure 30 (Continued)



Path (1): 1. compute  $d/(N_b - b)$ ,  
 2. E is the new current node,  
 3.  $N_e = N_b - (b + d)$ .

Path (2): 1. update d, 2. compute  $d/(N_b - b)$ ,  
 3. C is the new current node,  
 4.  $N_c = d$ .



Path (1): 1. compute  $d/(N_b - b)$ ,  
 2. E is the new current node,  
 3.  $N_b = N_b - (b + d)$ .

Path (2): 1. update d, 2. compute  $d/(N_b - b)$ ,  
 3. Y is the new current node,  
 4.  $N_y = d - c$ .

Path (3): 1. update d and c, 2. compute  $d/(N_b - b)$ ,  
 3. X is the new current node,  
 4.  $N_x = c$ .

a, b, c, d, e, x, y denote the RANK of node A, B, C, D, E, X, Y.  
 $N_a, N_b, N_c, N_d, N_e, N_x, N_y$ , denote the total external node  
 of node A, B, C, D, E, X, Y.

A, C, E, X, Y can be null nodes.

This updating algorithm of WB-trees takes  $O(\log n)$  rotations and RANK field updates in the worst case. In the event of redundant insertion, a second top-down pass is required to correct the RANK field in every node along the access path; again, an  $O(\log n)$  time is needed. However, this method has the advantages of (1) reducing the burden of computation overhead, (2) saving space by using only one field instead of two fields (one for  $|T_L|$  and another for  $|T|$ ), and (3) providing efficient "index position search." The index position search is described as follows: If the  $m$ -th element of a tree (which has  $n$  nodes and  $n \geq m$ ) is to be retrieved, first we let the COUNT be 0 and let the root be the current node  $T$ . We then traverse down along the access path, branching left if  $(\text{RANK}(T) + \text{COUNT}) > m$ , branching to right and updating (by adding  $\text{RANK}(T)$  to COUNT) if  $m > (\text{RANK}(T) + \text{COUNT})$ , or terminating if  $m = (\text{RANK}(T) + \text{COUNT})$ . An Example of index position search is shown in Figure 47 (in Appendix B).

### 3.4 Top-Down Updating for Splay Trees

Sleator and Tarjan (92) proposed a top-down version of splaying which works as follows. During the splaying, the tree is broken into three parts: a left tree, a middle tree, and a right tree. The middle tree contains the subtrees of the current node which is on the access path. The left and right trees consist of all the items in the original tree so far known to be less than the access item (say  $X$ ) and greater than  $X$ , respectively. Initially, the current node



is the tree root and the left and right trees are empty. We then search for X from the root down along the access path, two nodes at a time, breaking links and adding subtrees either to the left tree or to the right tree until X is reached. Finally we assemble the left tree, middle tree, and right tree into one tree and terminate the splaying operation. This algorithm has a disadvantage of requiring lots of split and join operations which may produce problems when concurrent operation is allowed.

On the other hand, Stephenson (95) proposed a similar top-down insertion algorithm which does not require any split or join operation. This algorithm also suffers poor updating efficiency because it visits one node at a time and does not perform rotations while traversing down along the access path.

To retain the merits of the above two algorithms, we proposed a new top-down splaying algorithm which has advantages such as (1) it visits two nodes at a time, (2) it performs rotations to reduce the height of each node along the access path, and (3) it eliminates the split and join operations to allow easy concurrent operations. The algorithm proceeds as follows. To splay an item (say X), first we let X be the "left current node, LC", "right current node, RC", and the new tree root. We then traverse down along the access path, visit two nodes at a time, and perform restructuring according to the following cases (algorithm SPLAYMT).

1. If the first node (visit two nodes at a time) is the access item X; we store the associated data of X in the

tree root, delete X, update LC and RC, and stop (see Figure 31a).

2. If the second node is the access item X; we store the associated data of X in the tree root, delete X, update LC and RC, and stop (see Figure 31b).

3. If there is a "right-right" access path (see Figure 31c); we perform a single rotation, update LC and RC, and continue traversing down along the access path.

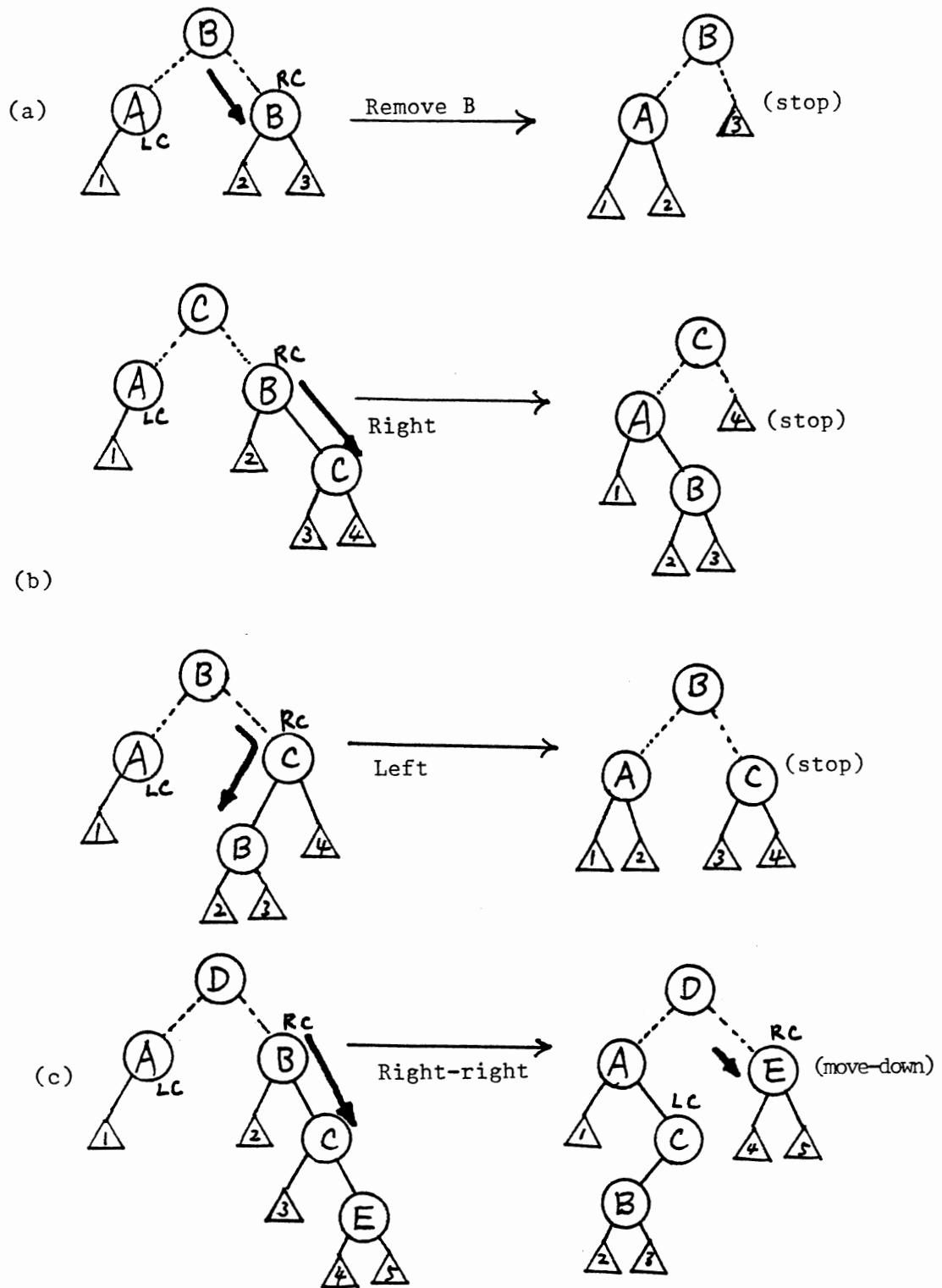
4. If there is a "right-left" access path (see Figure 31d), we update LC and RC and continue traversing down along the access path.

5. If there is a "left-right" access path (see Figure 31e), we update LC and RC and continue traversing down along the access path.

6. If there is a "left-left" access path (see Figure 31f); we perform a single rotation, update LC and RC, and continue traversing down along the access path.

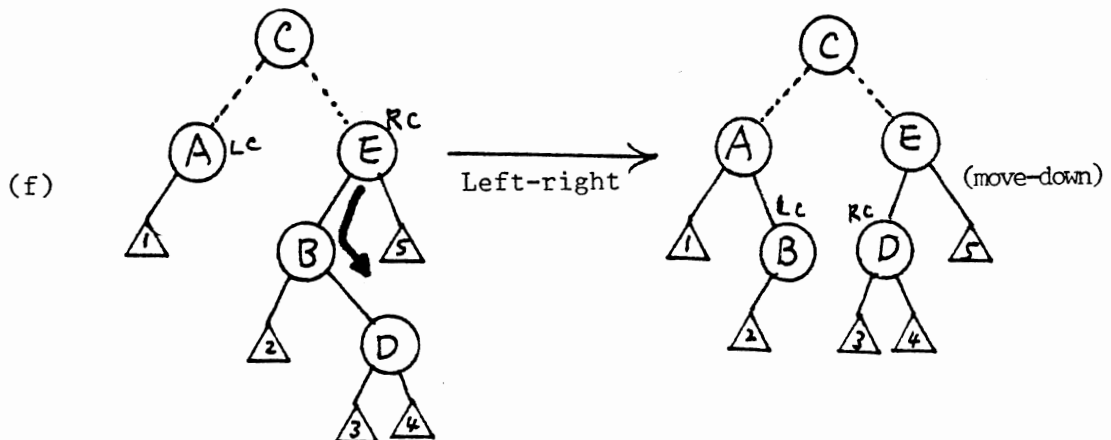
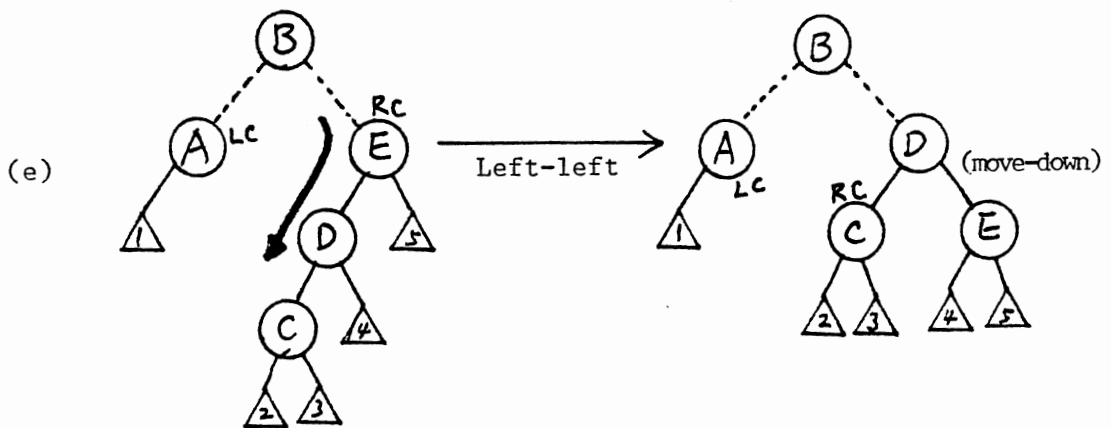
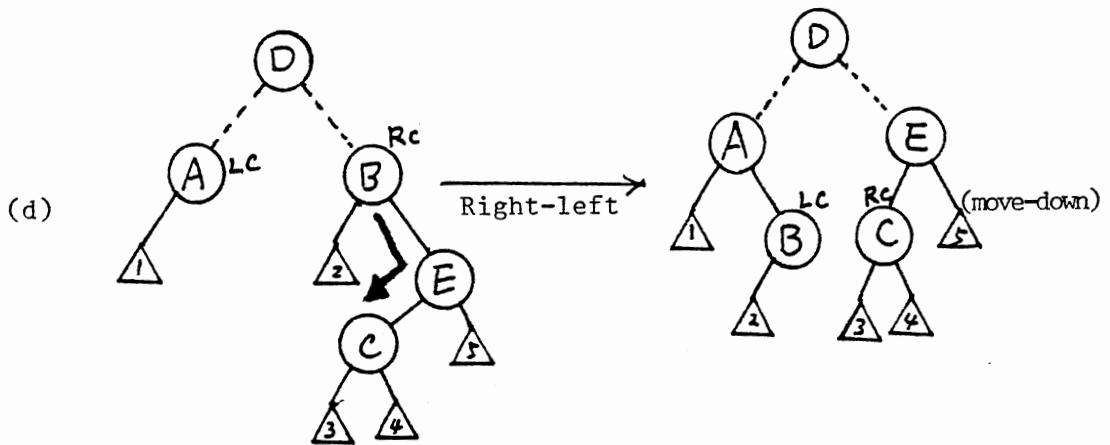
An example of this new top-down splaying operations is shown in Figure 48 (in Appendix B).

Figure 31 Top-Down Splaying Algorithm



→ denotes the access path. Symmetric variants are not shown.  
 LC denotes the left current node, RC denotes the right current node.

Figure 31 (Continued)



## CHAPTER IV

### AMORTIZED ANALYSIS AND PERFORMANCE EVALUATIONS

Common methods for the evaluation of the performance of search tree data structures include the worst-case analysis (45,52) and the average-case analysis (108,109). In the worst-case analysis, we sum the worst-case times of the individual operations which gives rise to a pessimistic evaluation for the structure. In typical search tree data structure applications, a sequence of operations is performed rather than a single operation; consequently, we are concerned with the total running time for that sequence of operations, not the individual running time of a single operation. The average-case analysis for a sequence of operations may be inaccurate because the probabilistic assumptions used to carry out the analysis may be incorrect.

Amortized analysis, a newly developed technique, has proven to be a realistic and robust method in complexity analysis of a variety of data structures. Tarjan (101) defined "amortization" as the average of total running times of operations in a sequence over the total number of operations. Tarjan also used two physical views to explain the concept of amortization. The first is the "bank's view" of

amortization: assume that we have an account in a bank. Each time we perform rotations after an update operation, we deposit some credits (the amount depends on the type of rotation performed) into the account; each time we complete an update operation without rebalancing, we withdraw some credits from the account. After a sequence of update operations is completed, an account balance is available from which the upper and lower bounds of performance of that data structure can be obtained. The second view is the "physicist view" of amortization: assume that we have a pump which can pump water from a lower level water tank into a higher level water tank (i.e., transform electrical energy into potential energy), and we also have a generator which can produce electricity by allowing water to flow from a higher level water tank to a lower level water tank (i.e., transform potential energy into electrical energy). Here the potential energy may be increased or decreased after an operation.

Tarjan defined a potential function  $\Phi$  that maps any configuration  $D$  of the data structure onto a real number  $\Phi(D)$  called the potential of  $D$ . " $T_i$ " is defined as the actual time of the  $i$ -th operation,  $\Phi_i$  and  $\Phi_{i-1}$  are the potentials of the data structure after and before the  $i$ -th operation, respectively, and the amortized time  $A_i$  of the  $i$ -th operation is defined to be  $A_i = T_i - \Phi_i + \Phi_{i-1}$ . For any sequence of  $m$  operations, the total running time is

$$\begin{aligned} \sum_{i=1}^m T_i &= \sum_{i=1}^m (A_i - \Phi_i + \Phi_{i-1}) \\ &= \Phi_0 - \Phi_m + \sum_{i=1}^m A_i \end{aligned}$$

where  $\Phi_0$  is the potential before the first operation, and  $\Phi_m$  is the potential after the m-th operation.

Tarjan applied this technique to evaluate three complexity problems such as the "move-to-front" linked list updating, the red-black tree updating, and the path compression for disjoint set problem. Mehlhorn and Tsakalidis (62) studied the amortized analysis of insertions into AVL-trees.

The study of amortized analysis of a red-black tree by using the banker's view is described as follows. Before computing the account balance, we need to assign credit to each type of structure. We assign one credit to a black node with two black children, zero credit to a black node with one red child, and two credits to a black node with two red children (only black nodes have credits). Table I shows the insertion and deletion potentials of a red-black tree. For insertion: if we attach a node to a black node and terminate the insertion (see Figure 8a), we withdraw one credit from the account; if we update colors and move-up (Figure 8b), we withdraw one credit from the account; if we perform a single rotation or a double rotation and terminate the insertion (Figures 8d and 8e), we deposit two credits. For

deletion: if we delete a black node and move-up (see Figure 10a), we withdraw one credit; if we update colors and move-up (Figure 10b), we withdraw two credits; if we update colors and terminate the deletion (Figure 10d), we withdraw one credit; if we perform a single rotation and terminate the deletion (Figure 10e), we may withdraw one credit or deposit two credits; if we perform a double rotation and terminate the deletion (Figure 10f), we deposit two credits. By using the above strategy, Tarjan (101) proved that the total time for  $m$  consecutive insertions in a tree of  $n$  nodes is  $O(n+m)$  which is  $O(1)$  in the amortized case. The  $O(n+m)$  bound does not include the search time which is  $O(\log n)$ .

Insertion Potentials (Figure 8)			Deletion Potentials (Figure 10)		
cases	before	after	cases	before	after
(a)	+1 $\cup$	0 $+1$	(a)	+1	0
(b)	+2	+1	(b)	+2	0
(c)	0	0 $+1$	(c)	0	0
(d)	0 $+1$	+2 <del><math>+1</math></del>	(d)	1	0
(e)	0	+2	(e)	0 or +2	+2 or +1
			(f)	0	+2

Table I. Potentials for Updating Red-Black Trees



## CHAPTER V

### COMPARISONS AND DISCUSSIONS

The comparisons among these search tree data structures are discussed below.

1. The bottom-up insertion (algorithm HBI) of  $HB(k)$  trees takes  $O(1)$  rotations in the worst case and  $O(1)$  balance-tag updates in the amortized case. The bottom-up deletion (algorithm HBD), on the other hand, takes  $O(\log n)$  rotations in the worst case. The top-down insertion (algorithm HBTI) of  $HB(k)$  trees takes  $O(1)$  rotations and balance-tag updates in the amortized case, the top-down deletion (algorithm HBTD) takes  $O(\log n)$  rotations in the worst case. The tree height in the worst case of an AVL tree is about  $1.44 * \log(N+1)$ .

2. The bottom-up updating (algorithms RBI and RBD) of red-black trees take  $O(1)$  rotations in the worst case and  $O(1)$  color changes in the amortized case, and the top-down updating (algorithms RBTI and RBTD) for red-black trees take  $O(1)$  rotations and color updates in the amortized case (100). The height of tallest red-black tree containing  $N$  internal nodes is about  $2 * \log(N+1)$ .

3. The top-down updating (algorithms WBT and WBMT) of weight-balanced trees take  $O(\log n)$  rebalances in the worst

case and a second top-down pass is required if the updating is redundant, and the worst case search time is about  $2 \cdot \log(N+1)$  where  $a = (1 - \sqrt{2}/2)$ .

4. The bottom-up updating (algorithms B12I and B12T) of 1-2 brother trees take  $O(\log n)$  rebalances in the worst case. Ottmann and Wood (76) studied the space utilization of 1-2 brother trees, and concluded that it requires  $1.618 \cdot N$  internal nodes to hold  $N$  records in the worst case and approximately  $1.5 \cdot N$  in the average case.

5. The splay tree structures have the amortized time bound of  $O(\log n)$  for all standard tree operations (algorithms SPLAY and SPLAYT) such as search, join, split, insert, and delete.

6. The K-d tree structures have the average performances of  $O(\log n)$  for insertion, search, and deletion. However, the deletion takes  $O(n^{(k-1)/k})$  and insertion takes  $O(n)$  in the worst case.

Red-black trees are slightly better than AVL trees on storage requirements because red-black trees require one bit per node for color-tag (black or red), and AVL trees require two bits per node for balanced-tag (+1, 0, or -1). The 1-2 brother trees do not carry balance information but they contain many unary nodes.

The truly top-down updating of a weight balanced tree has the advantages: (a) it eliminates the use of parent pointers or avoids the use of a stack to hold the access path, and (b) the weight factor  $\alpha$  can be chosen to trade off fast search time and rebalancing effort. The weight bal-

anced tree structures need more space to hold balance information, e.g., the RANK field (only 2 bits are needed for AVL trees).

The splay tree structure has the following advantages over balanced search tree structures: (a) the updating algorithms are conceptually simple and easy to implement, (b) the truly top-down updating version has simple locking protocol when concurrent operations are allowed, (c) it carries no balance information and needs less space, and (d) it has less operation cost, if the usage pattern is skewed (in the case when the locality model applies). The drawbacks of a splay tree structure are (a) it requires more local restructuring, and (b) it has very expensive individual operation cost and cannot be used in real time applications.

The merits of K-d trees are that a single data structure can store multikey records and handle a variety of queries very efficiently. The disadvantages of using K-d tree structures are that random deletion is very expensive and there is no restructuring technique which guarantees an  $O(\log n)$  access time.

## CHAPTER VI

### APPLICATIONS AND CONCLUSIONS

#### 6.1 Applications

Developments in memory technology have resulted in faster, larger, and less expensive memory implementations over time; these trends are likely to continue for some time. The existence of top-down update versions of tree data structures is important not only because such algorithms provide fast search, easy update, and convenient processing both randomly and sequentially but also because many users can access the structure concurrently without interfering with each other.

Balanced binary search tree structures have been implemented in primary memory to organize directories and tables for assemblers, compilers, and other system routines because these applications require fast access and have growing tables. For example the key of each record within an assembler or compiler may be a symbolic identifier denoting a variable in a FORTRAN program, and the rest of that record may contain information about the type of that variable and its storage allocation.

Wright (106,107) studied the use of balanced binary search trees to organize magnetic bubble memories (MBM), a

newly developed secondary memory device. He proved that the balanced binary search trees are reasonable alternatives to multiway trees (such as B-trees) for organizing large files in the MBM. Wright's result enhances the importance of developing top-down versions of updating balanced binary search trees.

Sarnak and Tarjan (90) gave the definition of persistent search tree structures: a persistent search tree has the property of that after an insertion or deletion, the old version of the tree can still be accessed. They used the red-black tree as a persistent data structure to solve planar point location problems because the amortized cost per update for red-black trees is  $O(1)$ . They concluded that this persistent data structure has  $O(\log m)$  search time,  $O(\log n)$  update time, and requires  $O(1)$  amortized space per update starting from the empty tree, where  $m$  is the total number of updates and  $n$  is the total number of tree nodes. The top-down algorithms presented in this thesis apply to persistent tree structures.

## 6.2 Conclusions

Search tree data structures are very important techniques for organizing large files, maintaining tables, solving geographic range queries, supporting dedicated database systems, and performing computer graphics. Several conclusions are described as follows.

1. The existence of top-down updating algorithms makes concurrent data processing easier, however, the use of non-

homogeneous tree structures causes the updating operations to be less efficient.

2. The performances of insertion algorithms for red black trees and AVL trees are similar; they require  $O(1)$  rotation in the worst case. However, the deletion algorithm for red-black trees requires only  $O(1)$  rotations in the amortized case, which is much better than the deletion algorithm of AVL trees which requires  $O(\log n)$  rotations in the worst case.

3. The top-down update algorithm for red-black trees is easy to implement and suitable for concurrent operations. On the other hand, the update algorithm for AVL trees has complex prebalancing strategy.

4. The WB-trees have longer search time and need more space for storing balance information but by selecting the weight factor, we can trade off fast search time and rebalancing effort.

5. The splay trees are very efficient for maintaining tables in the system if the locality model applies. The updating algorithms are conceptually simple and easy to implement, and the truly top-down updating version has a simple locking protocol when concurrent operations are allowed. The individual operation cost is very expensive and cannot be used in real time applications.

### 6.3 Suggestions

Several suggestions for further study are described as follows.

1. This thesis covers only part of the search tree data structures. Other search tree structures such as the biased binary search trees (12), the biased multiway search trees (12,28), heaps (94,99), and  $\alpha$ - $\beta$  trees (20) are necessary for the completeness of this study.

2. Three new versions of top-down updating algorithms for the HB(k)-trees, WB-trees, and splay trees are presented for concurrent data processing environment. We suggest that the technique of amortized analysis should be applied to evaluate the performance of the above top-down updating algorithms.

4. The self-adjusting version of B-tree structure requires further study.

5. There is a very challenging problem of K-d trees, namely, to develop an efficient updating algorithm which can perform rebalancing after an update operation and guarantee  $O(\log n)$  updates in the worst case.

## A SELECTED BIBLIOGRAPHY

- (1) Adelson-Velskii, G. M., and Landis, Y. M., "Algorithm for the organization of information," Dokl. Akad. Nauk. USSR, 146, (1962), 263-266. English Transl. in Soviet Math. Dokl., 3, (1962), 1259-1262.
- (2) Aho, A. V., Hopcroft, J. E., and Ullmann, J. D., Data structures and algorithms, Addison-Wesley Publishing Company, California, 1983.
- (3) Allen, B., "On the costs of optimal and near-optimal binary search trees," Acta Informatica, 18, (1982), 255-263.
- (4) Allen, B., and Munro, I., "Self-organizing binary search trees," J. of ACM, 25, 4, (Oct. 1978), 526-535.
- (5) Ayala, D., Brunet, P., Juan, R., and Navazo, I., "Object representation by means of nonminimal decision quadrees and octrees," ACM Trans. on Graphics, 4, 1, (Jan. 1985), 41-59.
- (6) Baer, J. L., "Weight balanced trees," Proc. AFIPS 1975 NCC, AFIPS Press, Montvale, NJ. 1975, 417-472.
- (7) Baer, J. L., Du, H. C., and Ladner, R. E., "Binary search in a multiprocessing environment," IEEE Trans. on Computer, C-32, 7, (July 1983), 667-677.
- (8) Baer, J. L., and Schwab, B., "A comparison of tree balancing algorithms," Communications of ACM, 20, 5, (May 1977), 322-330.
- (9) Bayer, R., "Symmetric binary B-trees: Data structure and maintenance algorithm," Acta Informatica, 1, (1972), 290-306.
- (10) Bayer, R., and McCreight, C. C., "Organization and maintenance large ordered indexes," Acta Informatica, 1, 3, (1972), 173-189.



- (11) Bayer, R., and Schkolnick, M., "Concurrency of operation on B-trees," Acta Informatica, 9, (1977), 1-21.
- (12) Bent, S., Sleator, D., and Tarjan, D. R., "Biased search trees," SIAM J. on Computer, 14, 3, (1985), 545-568.
- (13) Bentley, J. L., "Multidimensional binary search trees used for associative searching," Communications of ACM, 18, 9, (1975), 509-517.
- (14) Bentley, J. L., "Multidimensional binary search trees in database application," IEEE Trans. on Software Eng., SE-5, 4, (July 1979), 333-340.
- (15) Bongiovanni, G., and Wang, C. K., "Tree search in major/minor loop magnetic bubble memories," IEEE Trans. on Computer, C-30, 8, (Aug. 1981), 537-545.
- (16) Brown, F. W., and Kollias, J. G., "A partial analysis of random height-balanced trees," SIAM J. on Computer, 8, 1, (Feb. 1979), 33-41.
- (17) Burton, F. W., and Kollis, J. G., "Comments on the explicit quad tree as a structure for computer graphics," The Computer J., 26, 2, (1983), 188.
- (18) Chang, H. and Iyengar, S. S., "Efficient algorithm to globally balance a binary search trees," Communications of ACM, 27, 7, (July 1984), 695-702.
- (19) Chang, J. M., and Fu, K. S., "Extended K-d tree database organization a dynamic multiattribute clustering method," IEEE Trans. on Software Eng., SE-7, 3, (May 1981), 284-290.
- (20) Choy, D. M., and Wong, C. K., "Construction of optimal a-b leaf trees with applications to prefix code and information retrieval," SIAM J. on Computer, 12, 3, (1983), 426-446.
- (21) Chung, K. M., and Luccio, F., and Wong, C. K., "A tree storage scheme for magnetic bubble memories," IEEE Trans. on Computer, C-29, 10, (Oct. 1980), 864-874.
- (22) Comer, D., "The ubiquitous B-tree," Computing Surveys, 11, 2, (June 1979), 121-137.
- (23) Dekel, E., and Sahni, S., "Binary trees and parallel scheduling algorithms," IEEE Trans. on Computer, C-32, 3, (March 1983), 307-315.

- (24) Diehr, G., and Faaland, B., "Optimal pagination of B-trees with variable-length items," Communications of ACM, 27, 3, (March 1984), 241-247.
- (25) Ellis, C. S., "Concurrent search and insertion in AVL trees," IEEE Trans. on Computers, C-29, 9, (Sep. 1980), 811-817.
- (26) Ellis, C. S., "Concurrent search and insertion in 2-3 trees," Acta Informatica, 14, (1980), 63-86.
- (27) Eppinger, J. L., "An empirical study of insertion and deletion in binary search trees," Communications of ACM, 26, 9, (Sep. 1983), 663-669.
- (28) Feigenbaum, J., and Tarjan, R. E., "Two new kinds of biased search trees," Bell Sys. Tech. J. 62, 10, part 2, (1983), 3139-3158.
- (29) Finkel, R. A., and Bentley, J. L., "Quad trees: A data structure for retrieval on composite key," Acta Informatica, 4, (1974), 1-9.
- (30) Foster, C. C., "A generalization of AVL trees," Communications of ACM, 16, (1973), 513-517.
- (31) Frost, R. A., and Peterson, K. M., "A short note on binary search trees," The Computer J., 25, 1, (1982), 158.
- (32) Gonnet, G. H., "Balancing binary trees by internal path reduction," Communications of ACM, 26, 12, (Dec. 1983), 1074-1081.
- (33) Gonnet, G. H., and Oliver, H. J., and Wood, D., "Height-ratio-balanced trees," The Computer J., 26, 2, (1983), 106-108.
- (34) Gottlieb, "Optimal multi-way search trees," SIAM J. on Computer, 10, 3, (1981), 422-433.
- (35) Gottlieb, A., "Comments on concurrent search and insertion in AVL trees," IEEE Trans. on Computer, C-30, 10, (Oct. 1981), 812.
- (36) Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees," Proc. 19th Annual IEEE Symp. on Foundations of Comput. Sci., (1978), 8-21.
- (37) Gupta, U., Lee, D. T., and Wong, C. K., "Ranking and unranking of 2-3 trees," SIAM J. on Computer, 11, 3, (Aug. 1982), 582-590.

- (38) Hansen, W. J., "A cost model for the internal organization of B -tree models," ACM Trans. on Prog. Lang. and Sys., 3, 4, (Oct. 1981), 508-532.
- (39) Held., G., and Stonebraker, M., "B-trees re-examined," Communications of ACM, 21, 2, (Feb. 1978), 139-143.
- (40) Huang, S. H., and Wong, C. K., "Binary search trees with limited rotation," BIT, 23, (1983), 436-455.
- (41) Huang, S. H., and Wong, C. K., "Average number of rotation and access cost in IR-trees," BIT, 24, (1984), 387-390.
- (42) Karlton, P. L., and Fuller, S. H., "Performance of height balanced trees," Communications of ACM, 19, (Jan 1976) , 23-28.
- (43) Korth, H. F., and Silberschatz, A., "Database System Concepts," McGraw-Hill, Inc. New York, (1986).
- (44) Knuth, D. E., "Optimum binary search trees," Acta Informatica, 1, (1971), 14-25.
- (45) Knuth, D. E., The art of computer programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, (1973).
- (46) Kosaraju, S. R., "Insertion and deletion in one-sided height balanced trees," Communications of ACM, 21, 3, (March 1978), 226-229.
- (47) Kriegel, H. P., and Vaishnavi, V. K., and Wood, D., "2-3 brother trees," BIT, 18, (1978), 425-435.
- (48) Kung, H. T., and Lehman, P. L., "Concurrent manipulation of binary search trees," ACM Trans. on Database Sys., 5, 3, (Sep. 1980), 354-382.
- (49) Kuspert, K., "Storage utilization in B\*-trees with a generalized overflow technique," Acta Informatica, 19, (1983), 35-55.
- (50) Kwong, Y. S., and Wood, D., "On B-trees: Routing schemes and concurrency," Proc. 1980 ACM SIGMOD Internat. Conf. Management of Data, (1980), 207-213.
- (51) Kwong, Y. S., and Wood, D., "A new method for concurrency in B-trees," IEEE Trans on Software Eng., SE-8, 3, (May 1982) 211-222.

- (52) Lee, D. T., and Wong, C. K., "Worst-case analysis for region and partial region search in multidimensional binary search trees and balanced quad trees," Acta Informatica, 9, (1977), 23-29.
- (53) Leeuwen, J. V., and Overmars, M. H., "Stratified balanced search trees," Acta Informatica, 18, (1983), 345-359.
- (54) Libera, F. D., and Gosen, F., "Using B-trees to solve geographic range queries," The Computer J., 29, 2, (1986), 176-181.
- (55) Luccio, L., and Pagli, L., "On the height of height balanced trees," IEEE Trans. on Computer, C-25, 7, (Jan. 1976), 87-90.
- (56) Luccio, F., and Pagli, L., "Rebalancing height balanced trees," IEEE Trans. on Computer, C-27, 5, (May 1978), 386-396.
- (57) Luccio, F., and Pagli, L., "Comment on generalized AVL trees," Communications of ACM, 23, 7, (July 1980), 394-395.
- (58) Manber, U., "Concurrent maintenance of binary search trees," IEEE Trans. on Software Eng., SE-10, 6, (Nov. 1984), 777-784.
- (59) Manber, U., and Ladner, R. E., "Concurrency control in a dynamic search structure," ACM Trans. on Database Sys., 9, 3, (Sep. 1984), 439-455.
- (60) Martin, W. A., and Ness, D. N., "Optimizing binary trees growth with a sorting algorithm," Communications of ACM, 15, 2, (1972), 88-93.
- (61) Mehlhorn, K., "Nearly optimal binary search trees," Acta Informatica, 5, (1975), 287-295.
- (62) Mehlhorn, K., and Tsakalidis, A., "An amortized analysis of insertions into AVL-trees," SIAM J. on Computer, 15, 1, (Feb. 1986), 22-33.
- (63) Miller, R., and Snyder, L., "Multiple access to B-trees," Proc. Conf. Info. Sci. and Sys., (1973), 400-407.
- (64) Miller, R. E., and Pippenger, N., and Rosenberg, A. L., "Optimal 2,3-trees," SIAM J. on Computer, 8, 1, (Feb. 1979), 42-59.

- (65) Moitra, A., and Iyengar, S. S., "A maximally parallel balancing algorithm for obtaining complete balanced binary trees," IEEE Trans. on Computers, C-34, 6, (June 1985), 563-565.
- (66) Mullin, J. K., "Change area B-trees: A technique to aid error recovery," The Computer J., 24, 4, (1981), 367-373.
- (67) Nievergelt, J., "Binary search trees and file organization," Computing Surveys, 6, 3, (Sep. 1974), 195-207.
- (68) Nievergelt, J., and Reingold, E. M., "Binary search trees of bounded balance," SIAM J. on Computer, 2, 1, (March 1973), 33-43.
- (69) Oliver, H., "A new class of balanced search trees: Half-balanced binary search tree," RAIRO Informatique Theoretique, 16, (1982), 51-71.
- (70) Oliver, M. A., and Wiseman, N. E., "Operations on quadtree encoded images," The Computer J., 26, 1, (1983), 83-91.
- (71) Ottmann, T., and Six, H. W., "Eine neue klasse von ausgeglichenen binarbaumen," Angewandte Informatik, 18, (1976), 395-400.
- (72) Ottmann, T., and Six, H. W., and Wood, D., "Right brother trees," Communications of ACM, 21, 9, (Sep. 1978), 769-776.
- (73) Ottmann, T., and Karlsruhe, H. W. S., and Wood, D., "The implementation of insertion and deletion algorithm for 1-2 brother trees," Computing J., 26, (1981), 367-378.
- (74) Ottmann, T., Schrapp, M., and Wood, D., "Purely top down updating algorithms for stratified search trees," Acta Informatica, 22, (1985), 85-100.
- (75) Ottmann, T., and Stucky, W., "Higher order analysis of random 1-2 brother trees," BIT, 20, (1980), 302-314.
- (76) Ottmann, T., and Wood, D., "1-2 brother trees or AVL trees revisited," The Computer J., 23, 3, (1981), 248-255.
- (77) Ottmann, T., and Rosenberg, A. L., and Six, H. S., "Binary search trees with binary comparison cost," International J. Comput. and Info. Sci., 13, 2, (1984), 77-101.

- (78) Ottmann, T., and et al., "Minimal-cost brother trees," SIAM J. on Computer, 13, 1, (Feb. 1984), 197-217.
- (79) Ouksel, M., and Scheuermann, P., "Multidimensional B-trees: Analysis of dynamic behavior," BIT, 21, (1981), 401-418.
- (80) Overmars, M. H., and Leeuwen, J. V., "Dynamic multidimensional data structure based on quad- and k-d trees," Acta Informatica, 17, (1982), 267-285.
- (81) Pallo, J. M., "Enumerating, ranking and unranking binary trees," The Computer J., 29, 2, (1986), 171-175.
- (82) Raiha, K. J., and Zweben, S. H., "An optimal insertion algorithm for one-sided height-balanced binary search trees," Communications of ACM, 22, 9, (Sep. 1979), 508-512.
- (83) Reingold, E. M., and Hansen, W. J., Data Structures, Little, Brown and Company, Boston, (1983).
- (84) Rosenberg, A. L., and Snyder, L., "Minimal-comparison 2,3-trees," SIAM J. on Computer, 7, 4, (Nov. 1978), 465-480.
- (85) Rosenberg, A. L., and Snyder, L., "Time- and space optimality in B-trees," ACM Trans. on Database Sys., 6, 1, (March 1981), 174-183.
- (86) Samadi, B., "B-trees in a system of multiple users," Information Processing Letter, 5, 4, (1976), 107-112.
- (87) Samet, H., "The quadtree and related hierarchical data structures," Computing Surveys, 16, 2, (1984).
- (88) Samet, H., "A quadtree medial axis transform," Communications of ACM, 26, 9, (Sep. 1983), 660-693.
- (89) Samet, H., and Webber, R. E., "Storing a collection of polygons using quadtrees," ACM Trans. on Graphics, 4, 3, (July 1985), 182-222.
- (90) Sarnak, N., and Tarjan, R. E., "Planar point location using persistent search trees," Communication of ACM, 29, 7, (July 1986), 669-679.

- (91) Scidmore, A. K., and Weinberg, B. L., "Storage and search properties of a tree-organized memory system," Communications of ACM, 6, 1, (Jan. 1963), 28-31.
- (92) Sleator, D., and Tarjan, R. E., "Self-adjusting binary search trees," Communications of ACM, 32, 3, (1985), 652-686.
- (93) Sleator, D., and Tarjan, R. E., "Self-adjusting heaps," SIAM J. on Computer, 15, 1, (Feb. 1986), 52-69.
- (94) Stasko, J. T., and Vitter, J. S., "Pairing heaps: Experiments and analysis," Communications of ACM, 30, 3, (March, 1987), 234-249.
- (95) Stephenson, C. J., "A method for constructing binary search trees by making insertions at the root," International J. Comput. Info. Sci., 9, (1980), 15-29.
- (96) Sussenguth Jr., E. H., "Use of tree structure for processing files," Communications of ACM, 26, 1, (Jan. 1983), 17-20.
- (97) Szwarcfiter, J. L., "Optimal multiway search trees for variable size keys," Acta Informatica, 21, (1984), 47-60.
- (98) Tarjan, R. E., "Updating a balanced search tree in  $O(1)$  rotations," Information Processing Letter, 16, 5, (June 1983), 253-257.
- (99) Tarjan, R. E., Data Structures and Network Algorithms Society for Industrial and Applied Mathematics, Philadelphia, PA., (1983).
- (100) Tarjan, R. E., "Efficient Top-Down Update of Red-Black Trees," Computer Sci. Dept. Princeton Univ., CS-TR-006-85.
- (101) Tarjan, R. E., "Amortized computational complexity," SIAM J. Alg. Disc. Math., 6, 2, (1985), 306-318.
- (102) Tarjan, R. E., "Algorithm design," Communications of ACM, 30, 3, (March 1987), 204-212.
- (103) Unterauer, K., "Dynamic weighted binary search trees," Acta Informatica, 11, (1979), 341-362.

- (104) Vaishnavi, V. K., and Kriegd, H. P., and Wood, D., "Optimal multiway search trees," Acta Informatica, 14, (1980), 119-133.
- (105) Woodward, J. R., "The explicit quadtree as a structure for computer graphics," The Computer J., 25, 2, (1982), 235-238.
- (106) Wright, W. E., "Binary search trees in secondary memory," Acta Informatica, 15, (1981), 3-17.
- (107) Wright, W. E., "Some file structure considerations pertaining to magnetic bubble memory," The Computer J., 26, 1, (1983), 43-51.
- (108) Wright, W. E., "Some average performance measures for the B-tree," Acta Informatica, 21, (1985), 541-557.
- (109) Yao, C. C., "On random 2-3 trees," Acta Informatica, 9, (1978), 159-170.
- (110) Yau, M-M., and Srihari, S. N., "A hierarchical data structure for multidimensional digital images," Communications of ACM, 26, 7, (1983), 504-515.
- (111) Zaki, A. S., "A comparative study of 2-3 trees and AVL trees," International J. of Comput. and Info. Sci., 12, 1, (1983), 13-33.
- (112) Zaki, A. S., "A space saving inserting algorithm for 2-3 trees," The Computer J., 27, 4, (1984), 368-372.
- (113) Zaki, A. S., and Baer, J. L., "Query cost in HB(1) trees vs. 2-3 trees," International J. of Comput. and Info. Sci., 10, 6, (1981), 383-395.



APPENDIX A

DEFINITIONS OF TREE VARIANTS

### A.1 Height-Ratio-Balanced Trees

The height-ratio-balanced trees are defined (33) as follows.

Let  $T$  be an extended binary tree such that  $T$  consists of a root node with two subtrees  $T_l$  and  $T_r$ . The "balance factor",  $B(T)$  of a tree is then defined as

$$B(T) = \begin{cases} 1/2, & \text{if } T \text{ has no child.} \\ h(T_l)/(h(T_l)+h(T_r)), & \text{otherwise.} \end{cases}$$

Where  $h(T_l)$  denotes the height of  $T$ 's left subtree and  $h(T_r)$  denotes the height of  $T$ ' right subtree. A tree  $T$  is said to be "height-ratio-balanced" of order  $\alpha$ , or in the set of  $hrb(\alpha)$  for  $0 \leq \alpha \leq 1/2$ , if

1.  $\alpha \leq B(T) \leq 1-\alpha$ .
2. Both  $T_l$  and  $T_r$  are also in the set of  $hrb(\alpha)$ .

### A.2 One-Sided Height-Balanced Binary Trees

Knuth (45) put another restriction on the AVL trees that defines a class of one-sided height-balanced trees (also known as R-trees). The restriction is added by not allowing the height of any node's left subtree to exceed that of its right subtree. That is the balance-tag of any node is either 1 or 0, and can be represented by using one bit of storage per node. A one-sided height-balanced tree satisfies

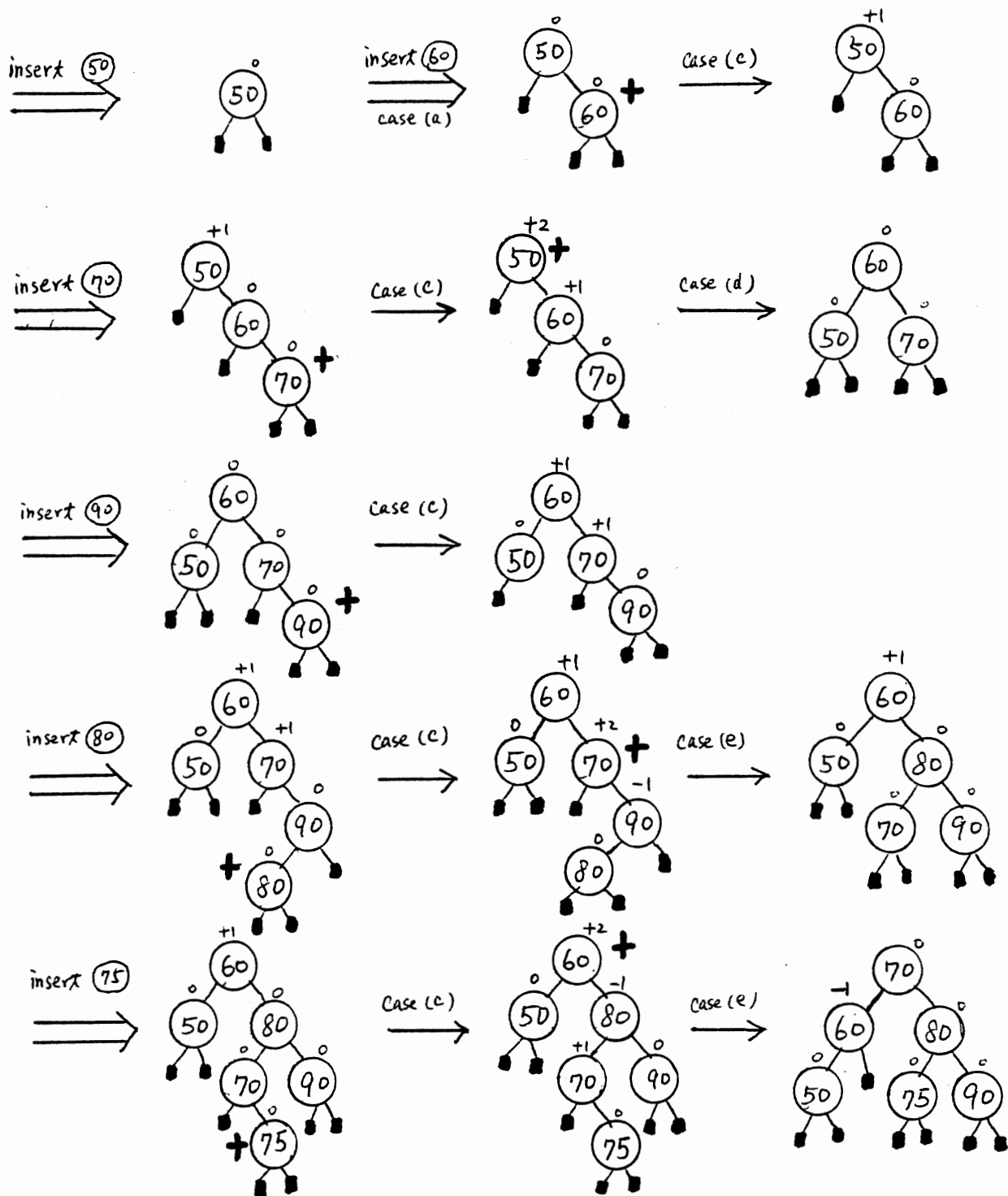
1.  $1 \geq (h(T_r)-h(T_l)) \geq 0$ .
2.  $T_r$  and  $T_l$  are one-sided height-balanced trees.

APPENDIX B

EXAMPLES OF TREE OPERATIONS

Figure 32

An Example of Bottom-Up Insertion into An AVL Tree



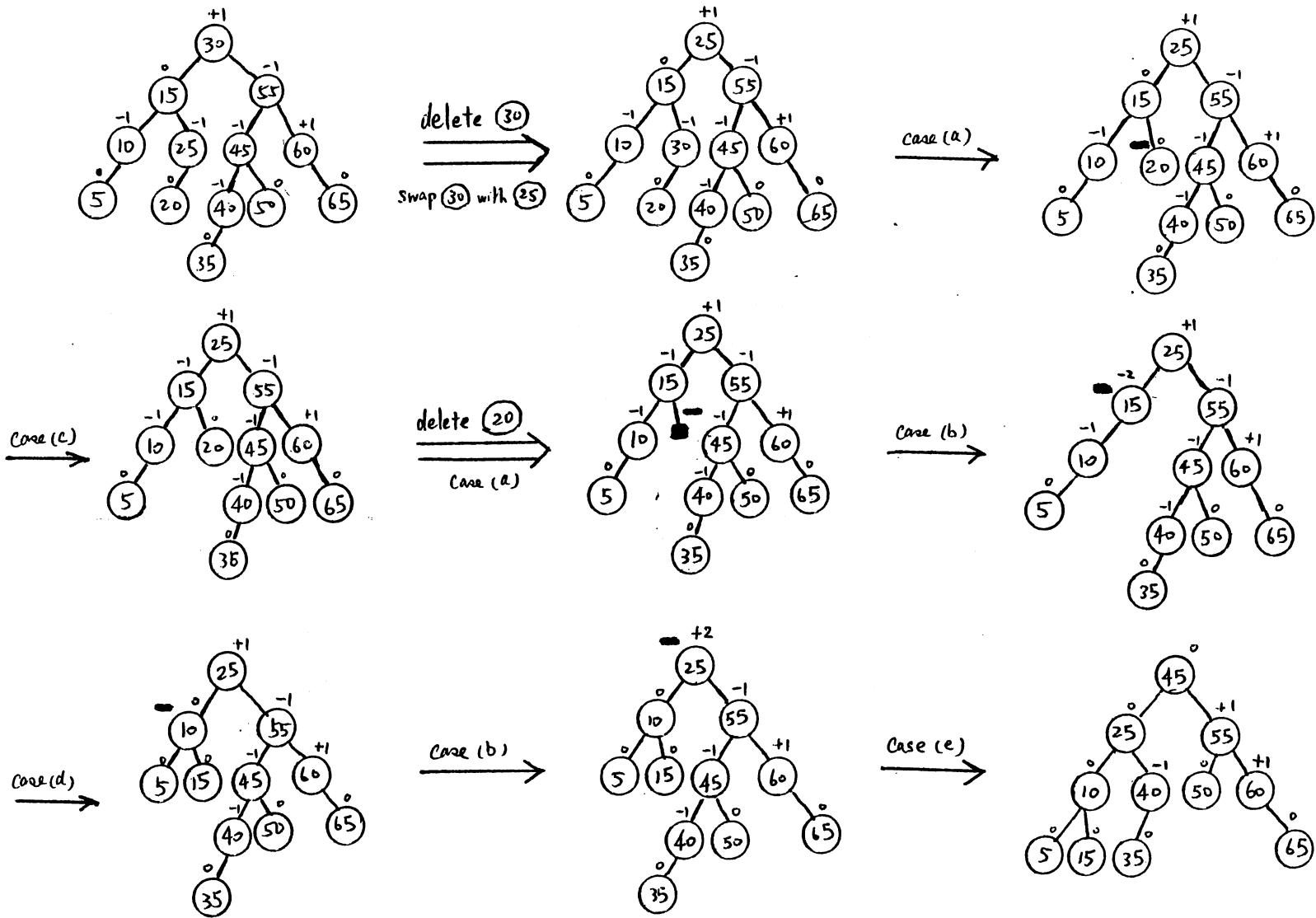


Figure 33 An Example of Bottom-Up Deletion from An AVL Tree

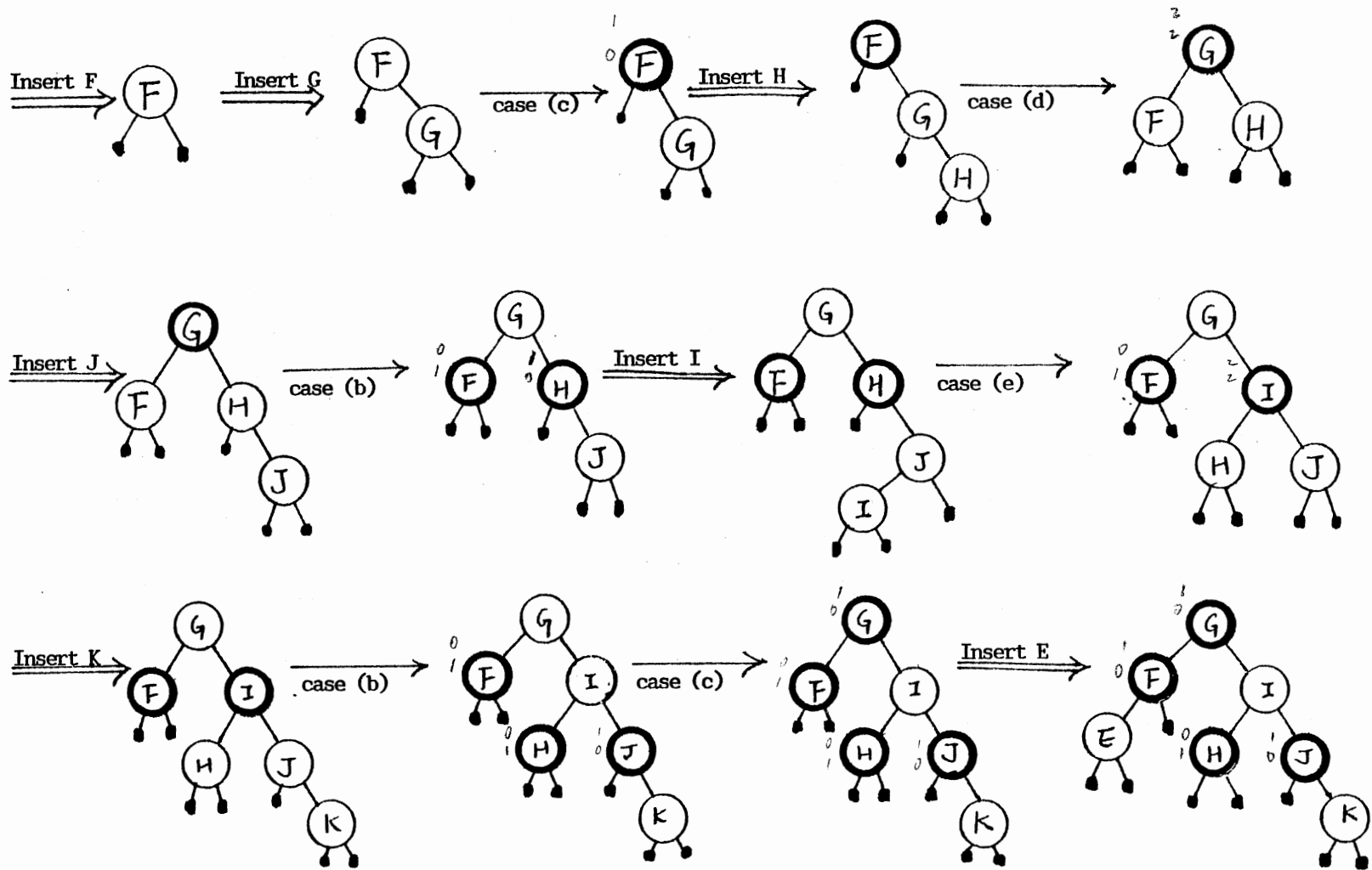


Figure 34 An Example of Bottom-Up Insertion into A Red-Black Tree

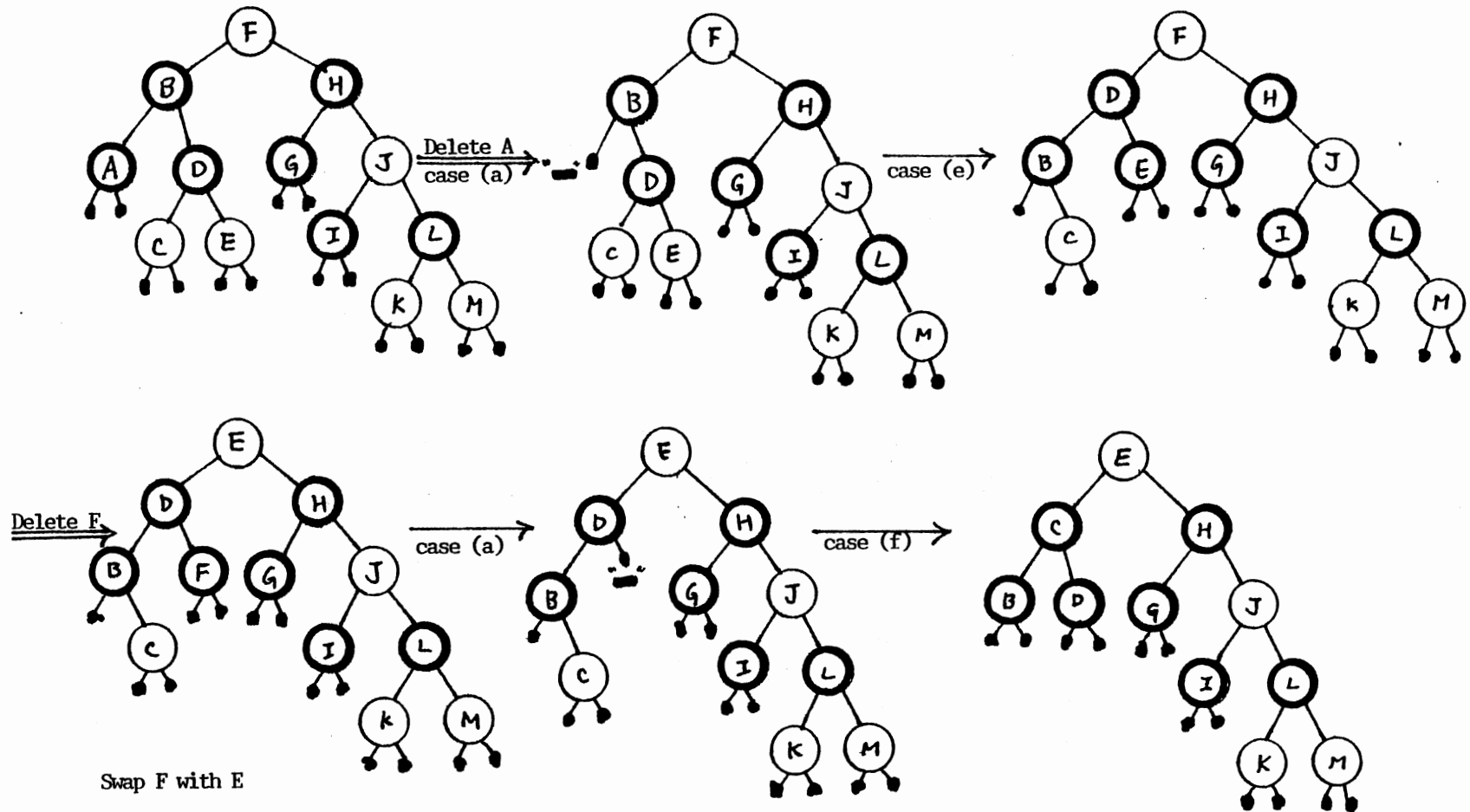


Figure 35 An Example of Bottom-Up Deletion from A Red-Black Tree

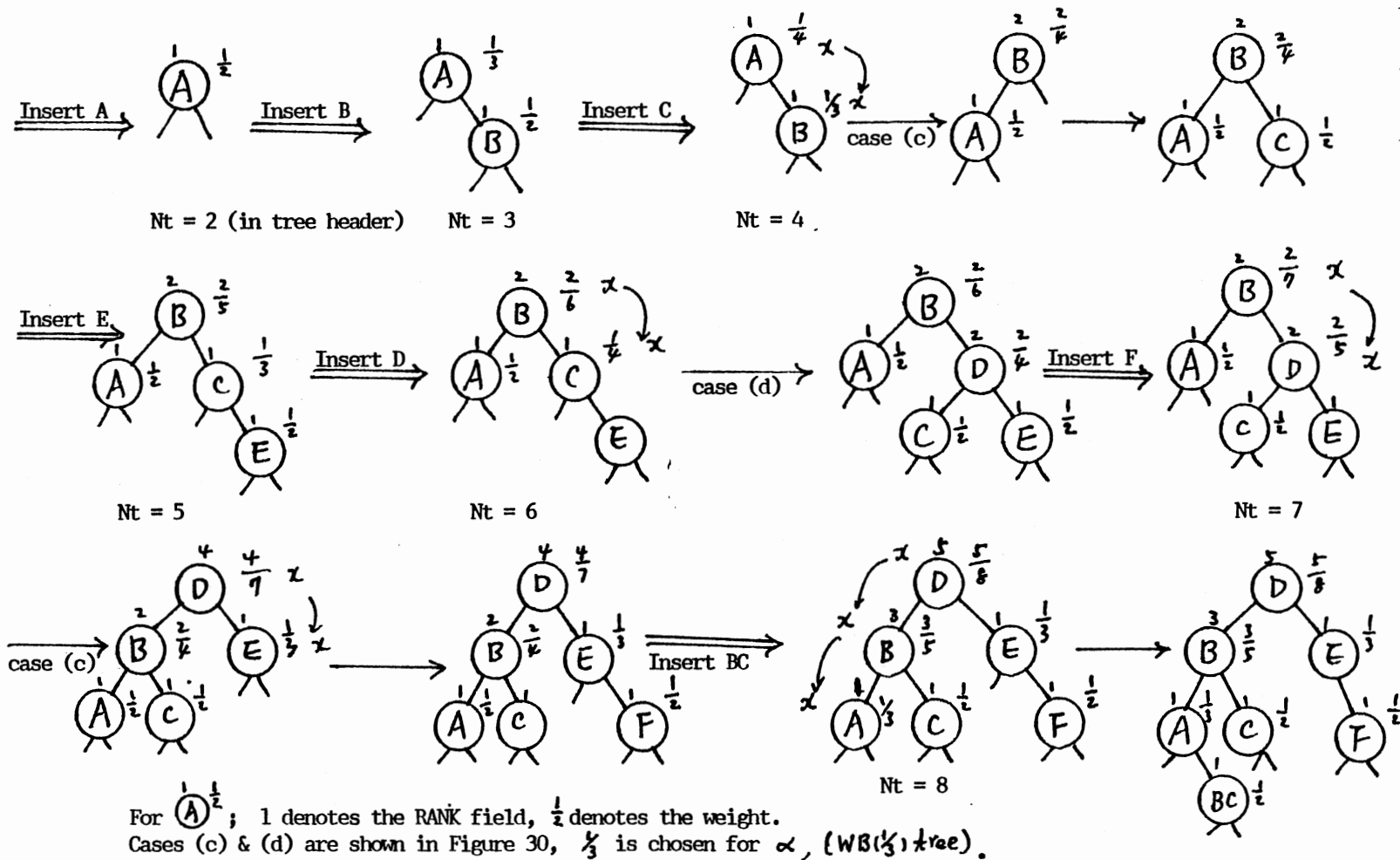


Figure 36 An Example of Top-Down Updating for A Weight-Balanced Tree



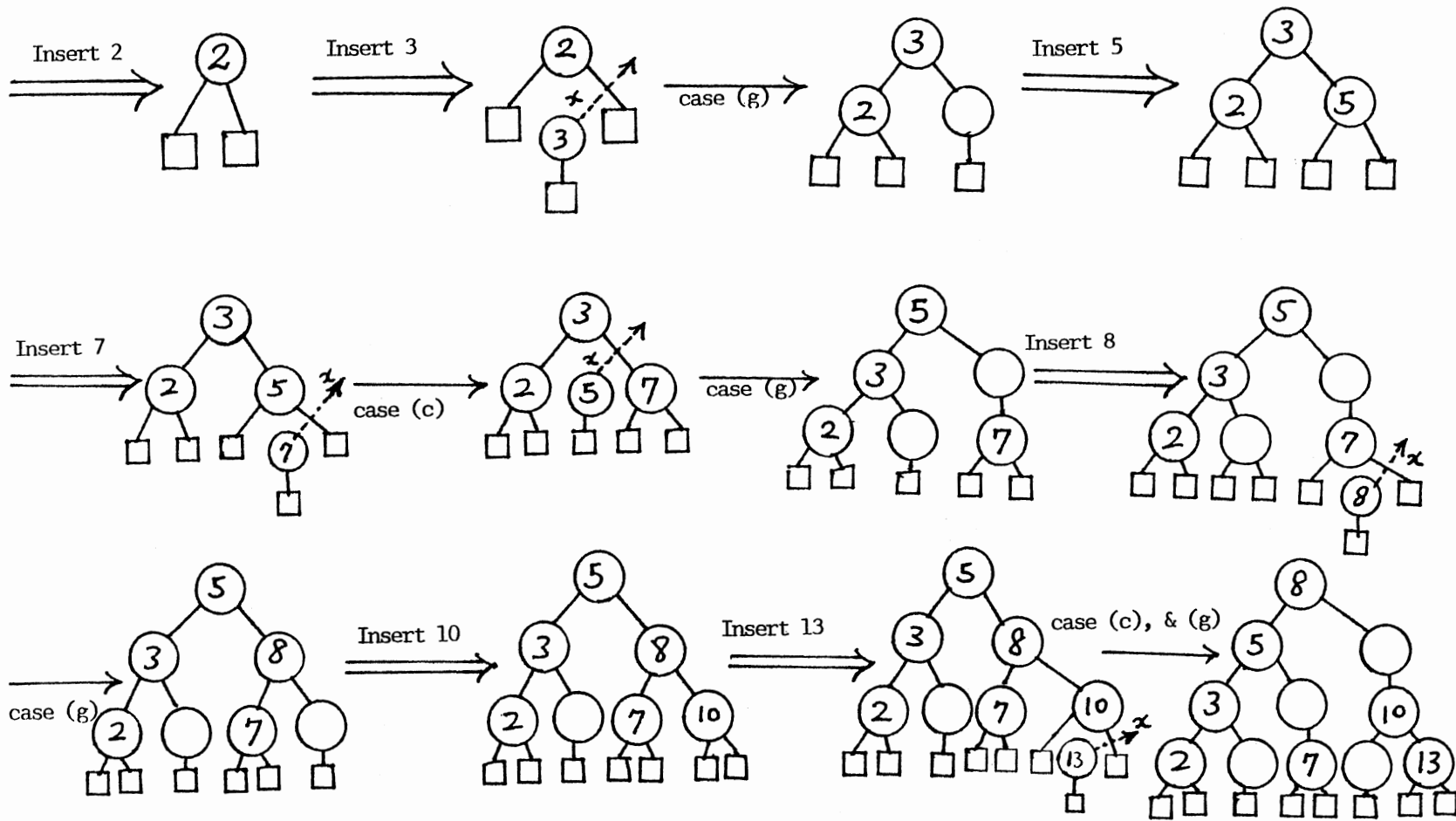


Figure 37 An Example of Bottom-Up Insertion into A 1-2 Brother Tree

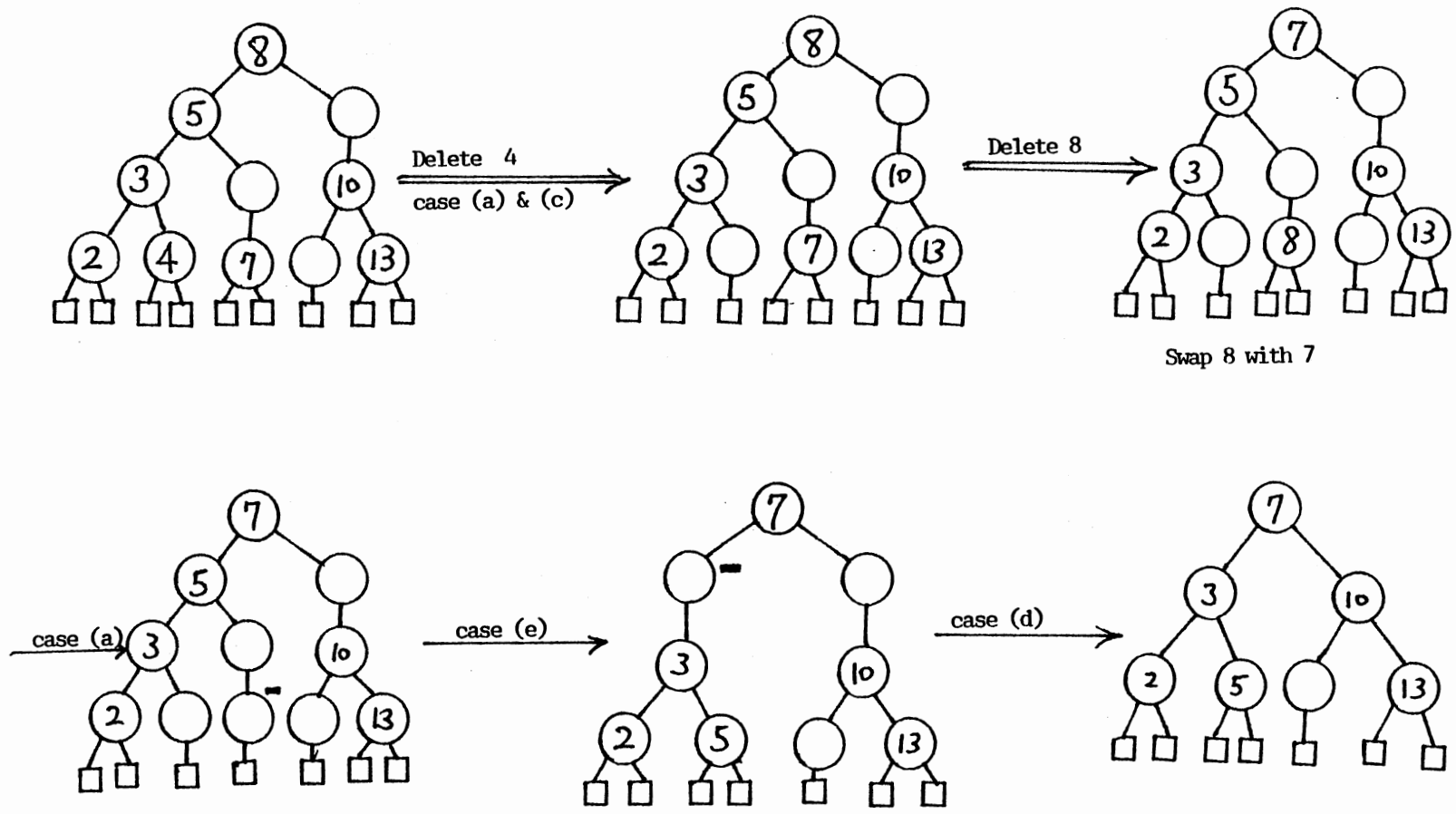
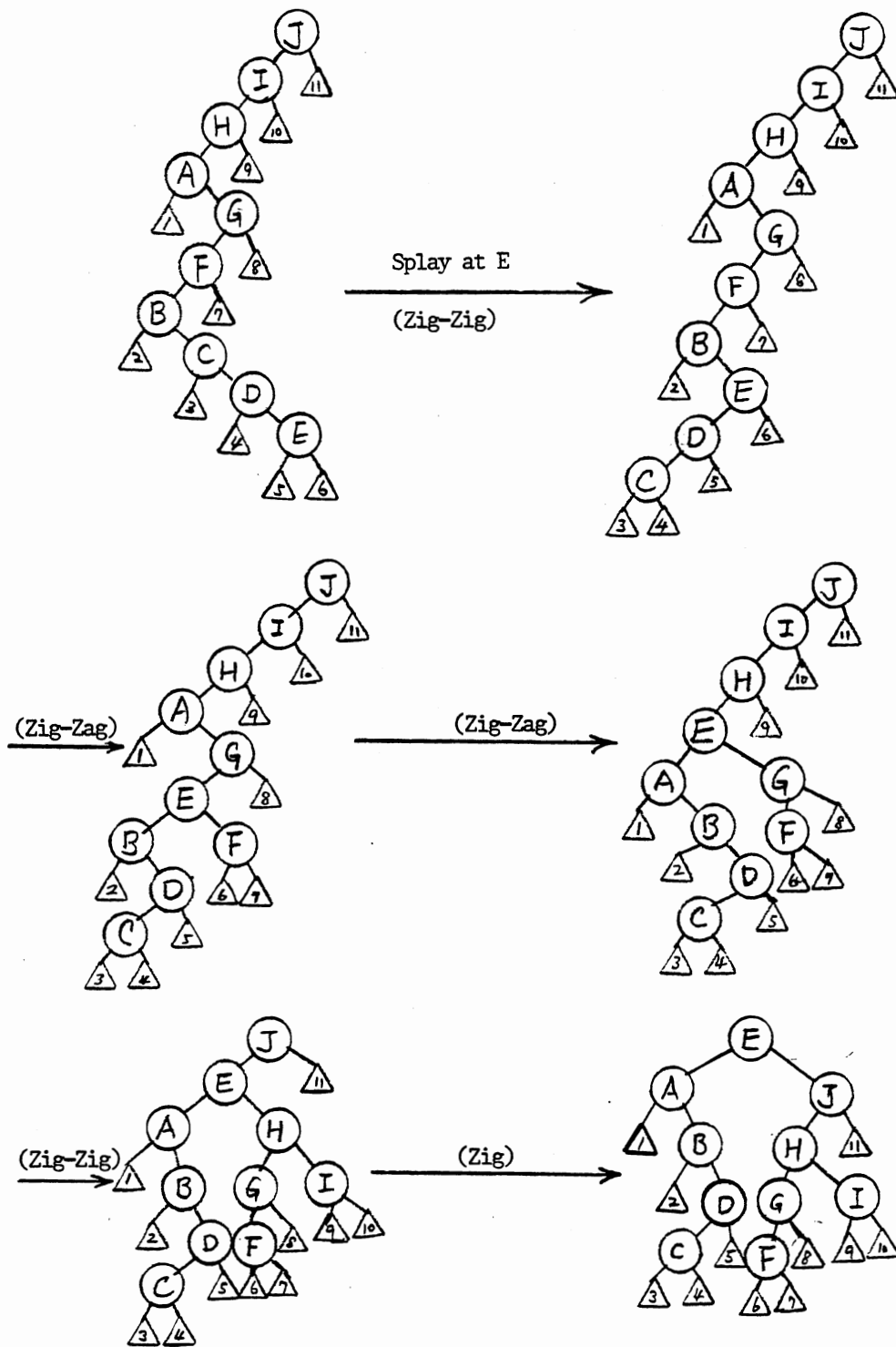


Figure 38 An Example of Bottom-Up Deletion from A 1-2 Brother Tree

Figure 39

An Example of Bottom-Up Splaying A Node in A Binary Tree



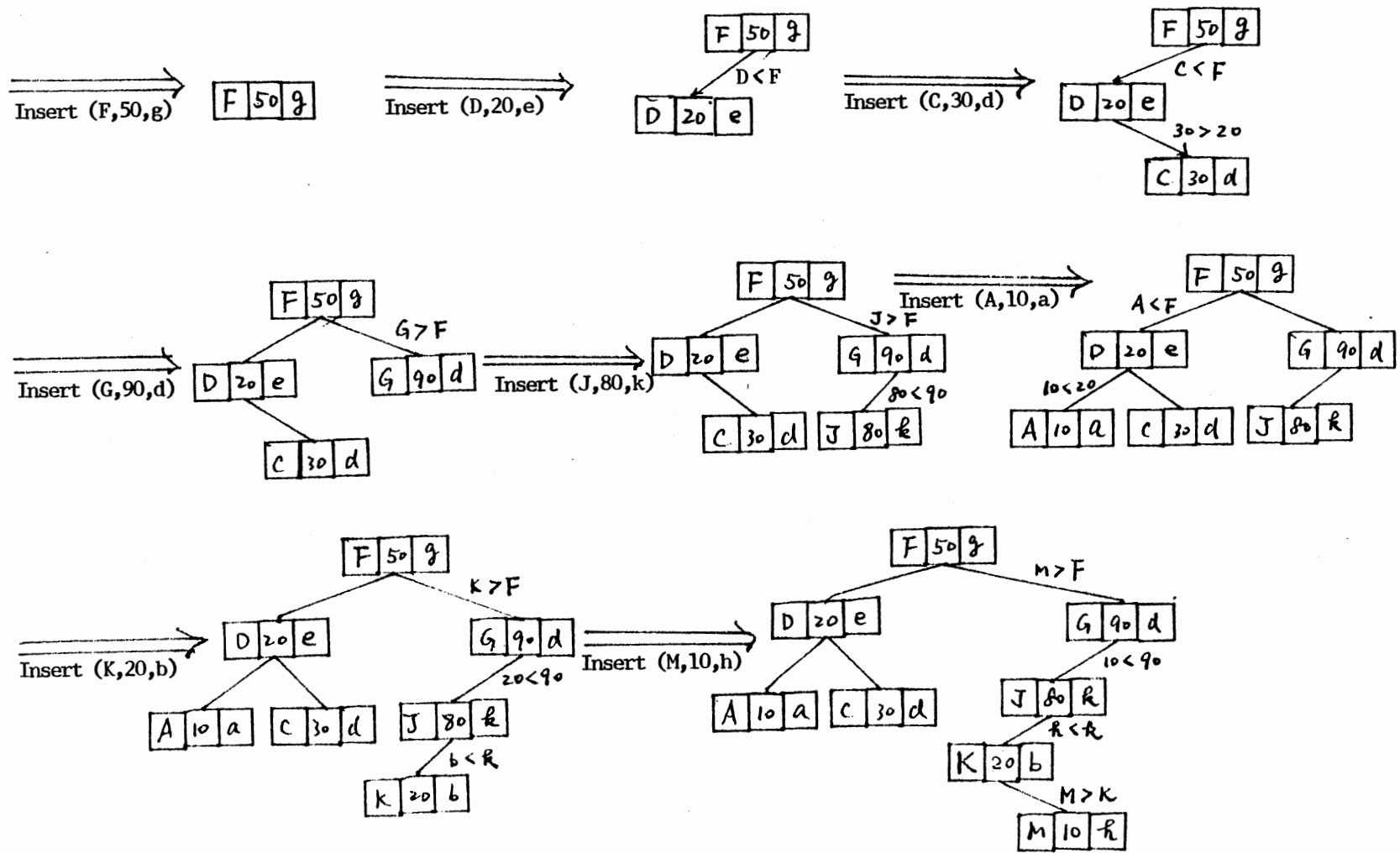
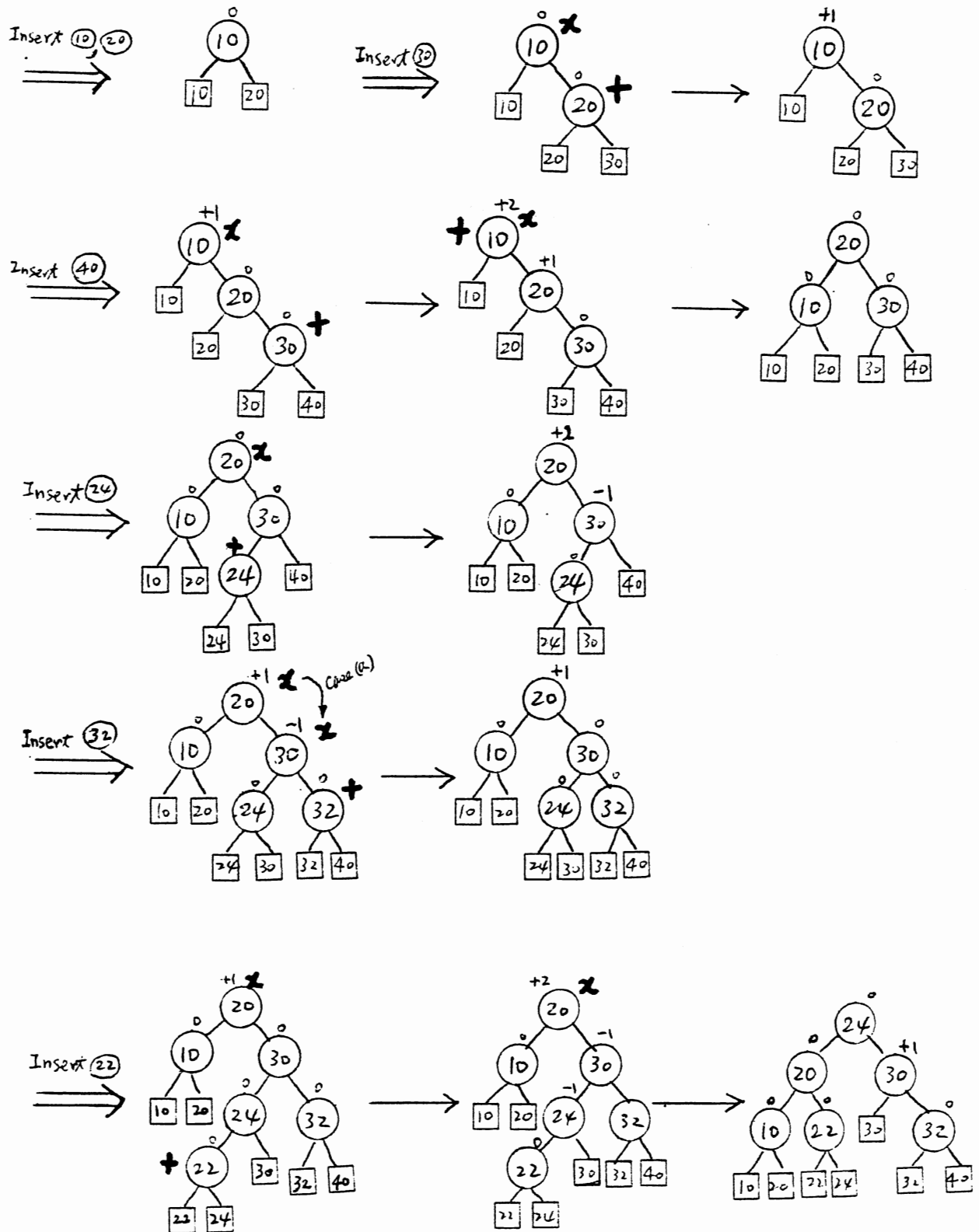


Figure 40 An Example of Insertion into A K-d Tree

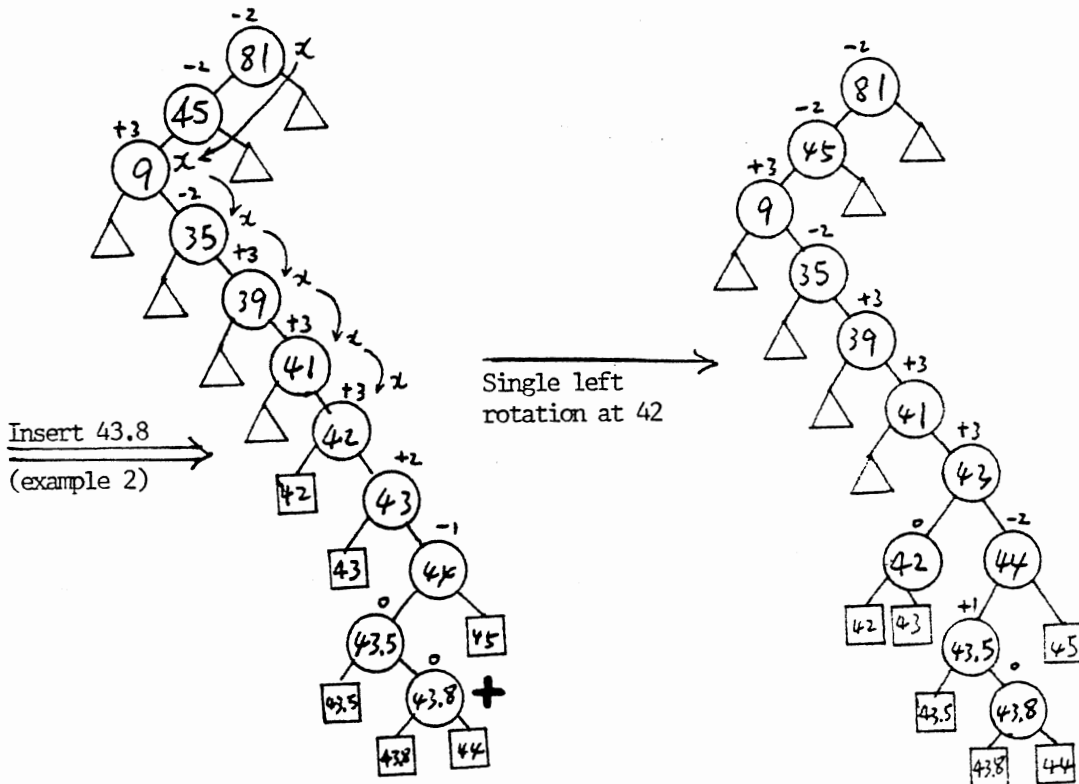
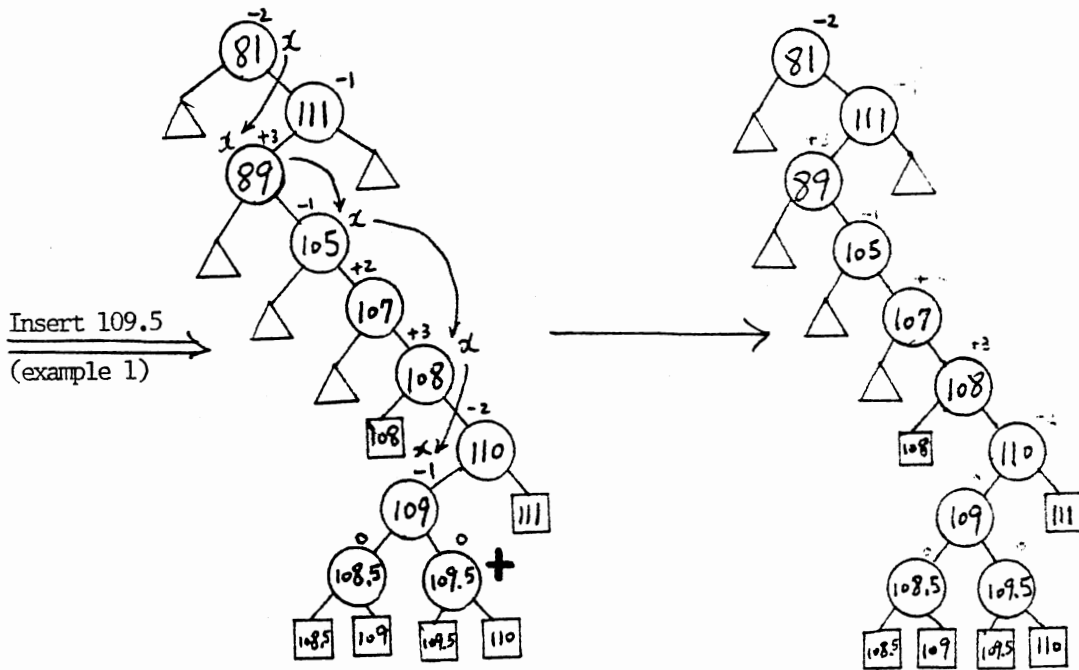
Figure 41

An Example of Top-Down Insertion into An AVL Tree



x denotes the current node, + denotes the "long"

Figure 42 Examples of Top-Down Insertion into An HB(3) Tree



$\lambda \rightsquigarrow \lambda$  denotes the current movement,  
 $\triangle$  denotes a subtree,  
 Insertions are made in Figure 50.

$+$  denotes the "long",  
 $\square$  denotes an external node,

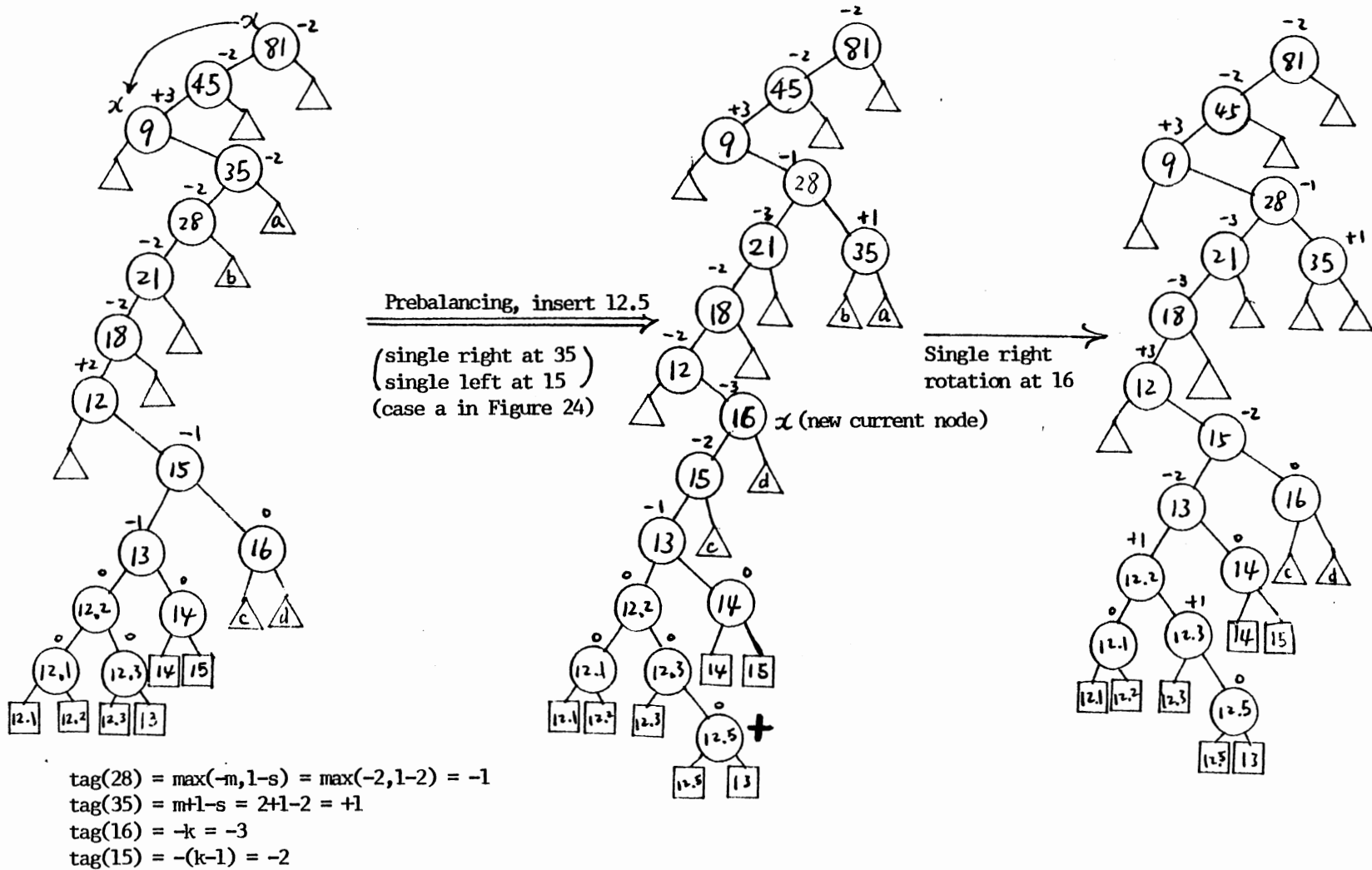


Figure 42 (Continued)

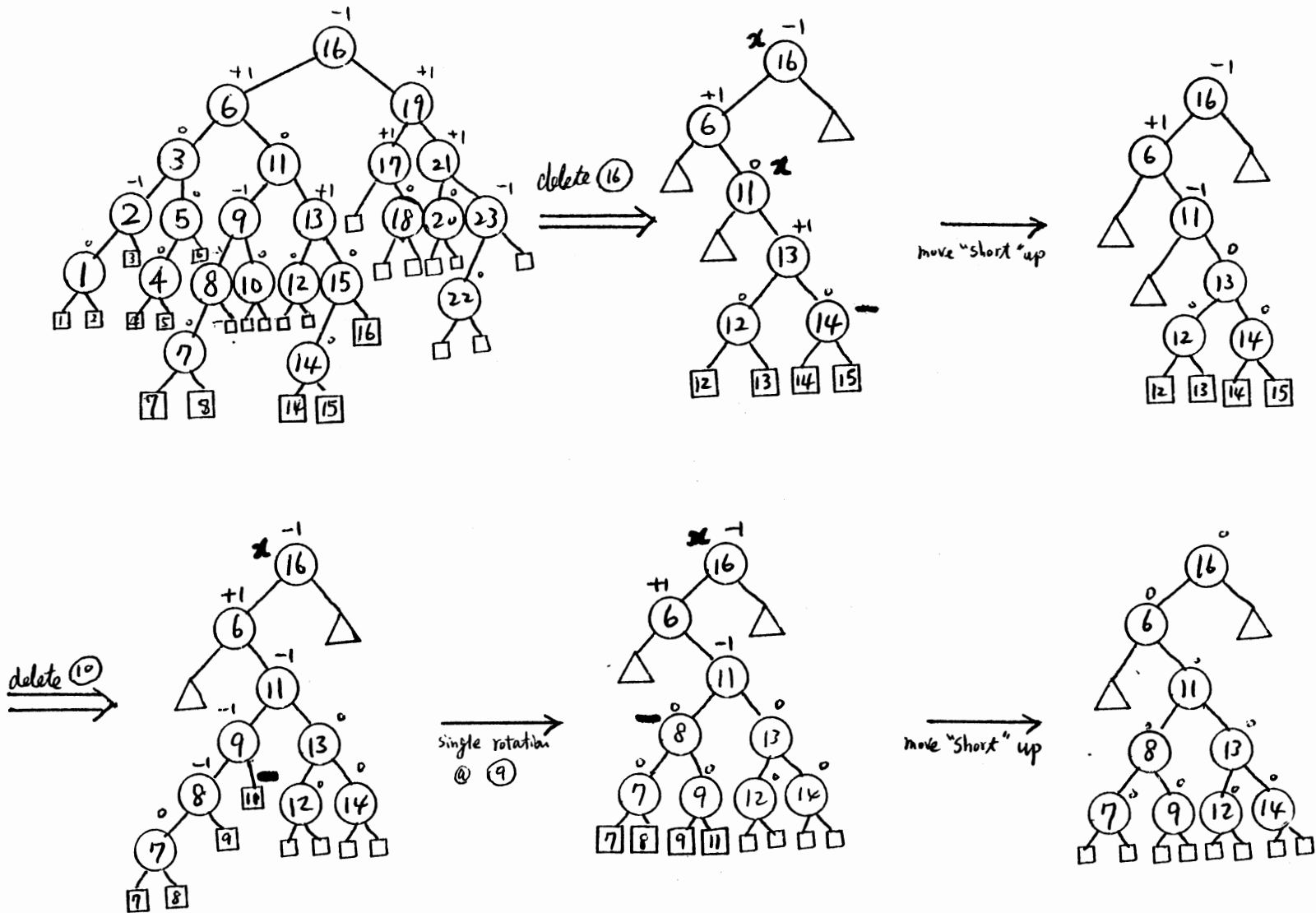
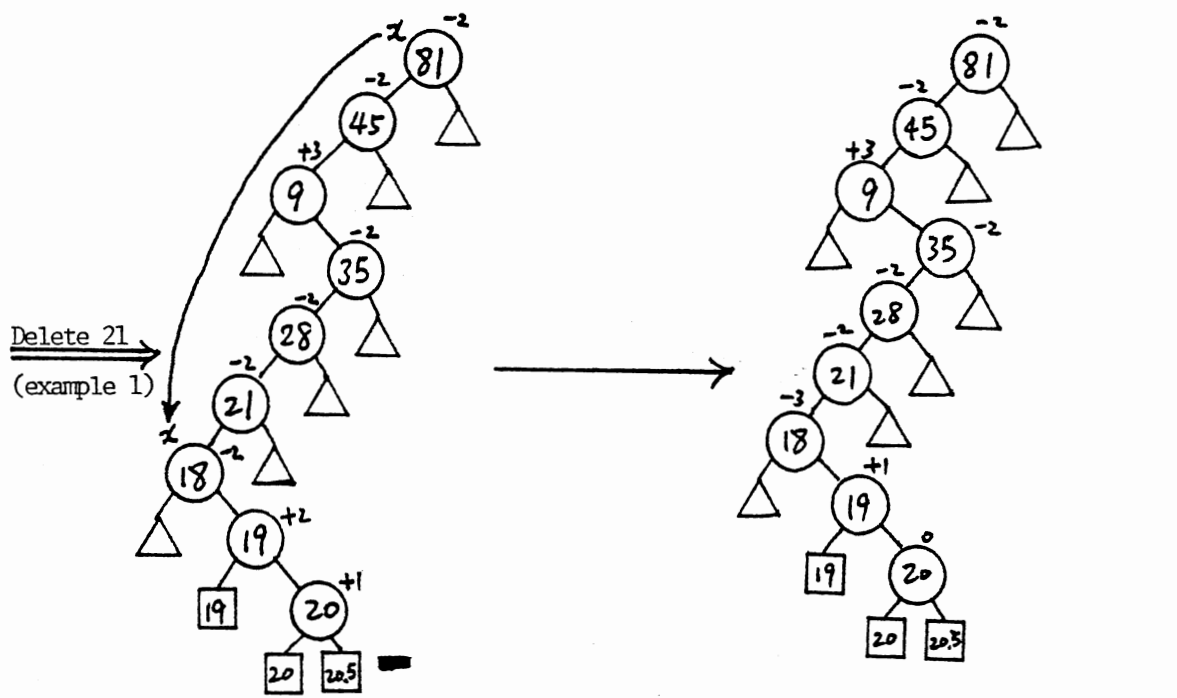


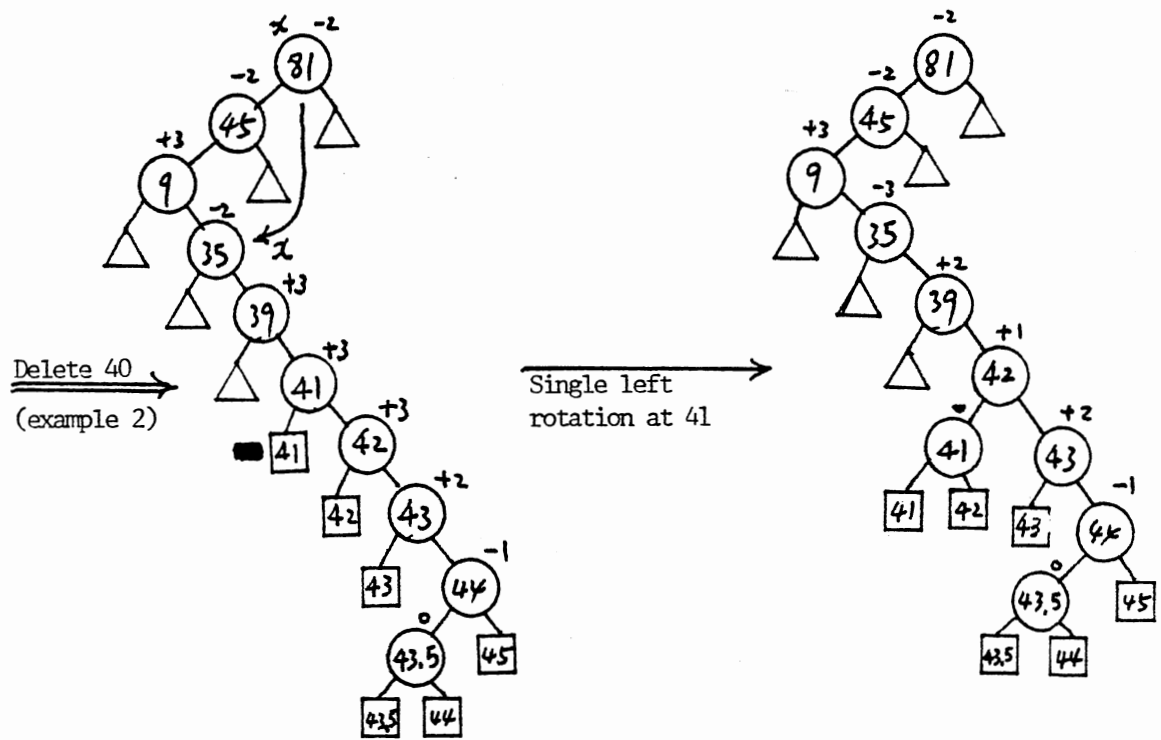
Figure 43 An Example of Top-Down Deletion from An AVL Tree



Figure 44 Examples of Top-Down Deletion from An HB(3) Tree



(the external node containing 21 is deleted)






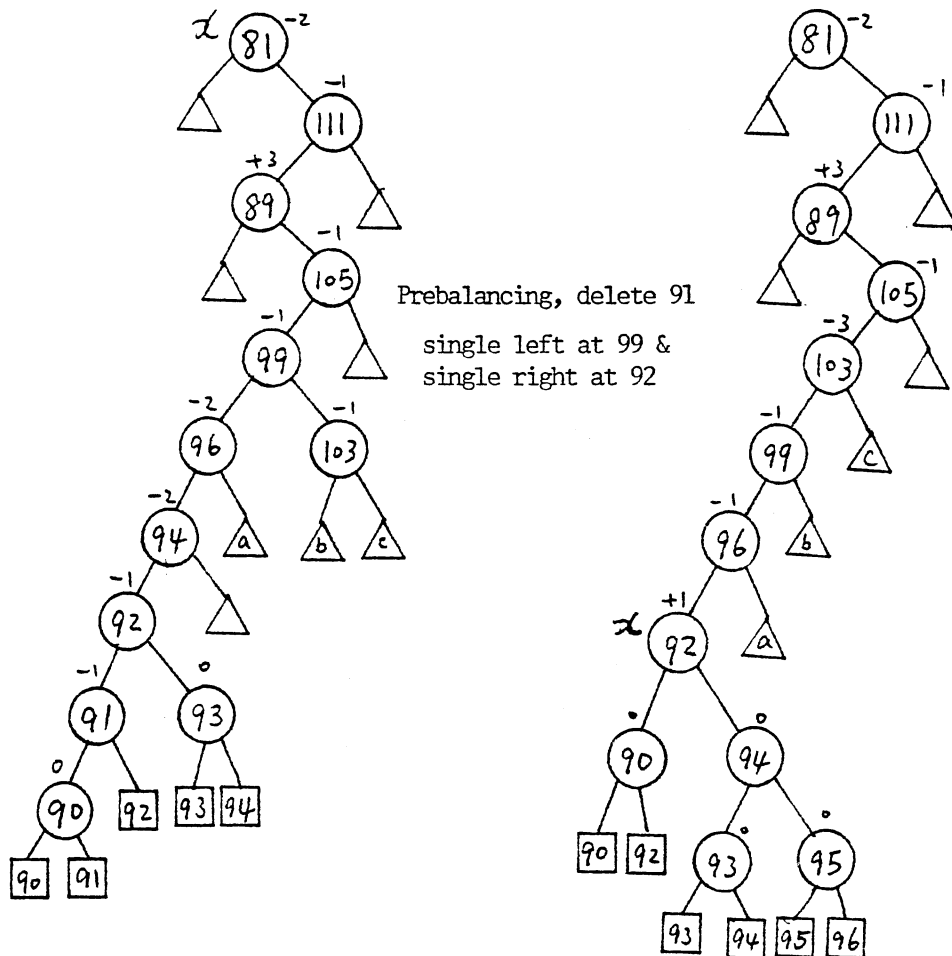
$\alpha \curvearrowright \alpha$  denotes the current node movement,  denotes the "short",  
 denotes a subtree,  denotes an external node,  
 Deletions are made in Figure 50

Figure 44 (Continued)



Node 92 becomes the new current after prebalancing,  
This is the case (b) in Figure 26.

$\text{tag}(99) = -1$  (not changed)

$\text{tag}(103) = -(s+m)-1 = -(1+1)-1 = -3$

$\text{tag}(94) = \max(1-q, -p) = \max(1-2, -1) = 0$

$\text{tag}(92) = (p+1-q) = 1+1-2 = 0$

after delete node 92,  $\text{tag}(92)$  becomes +1



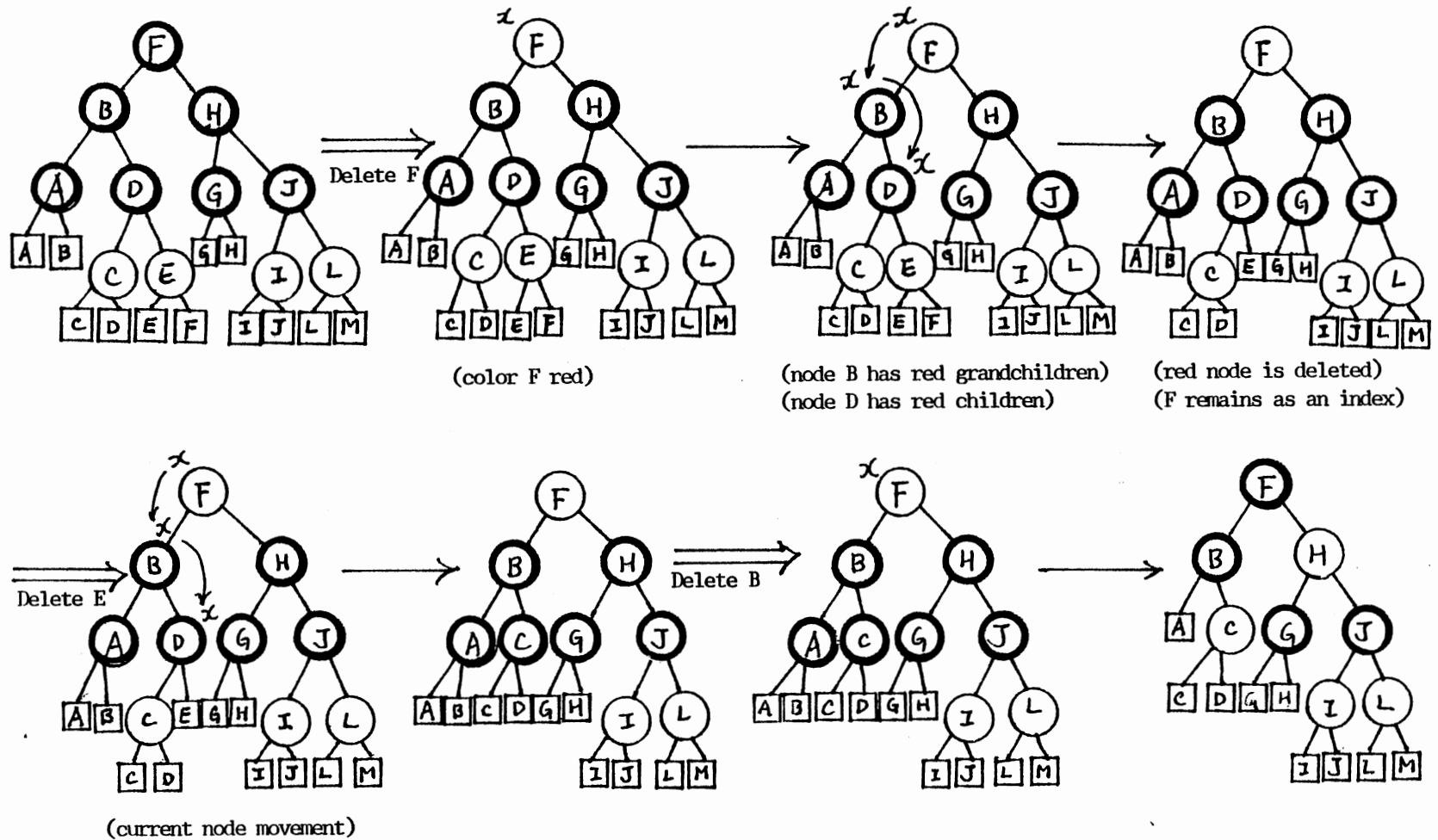
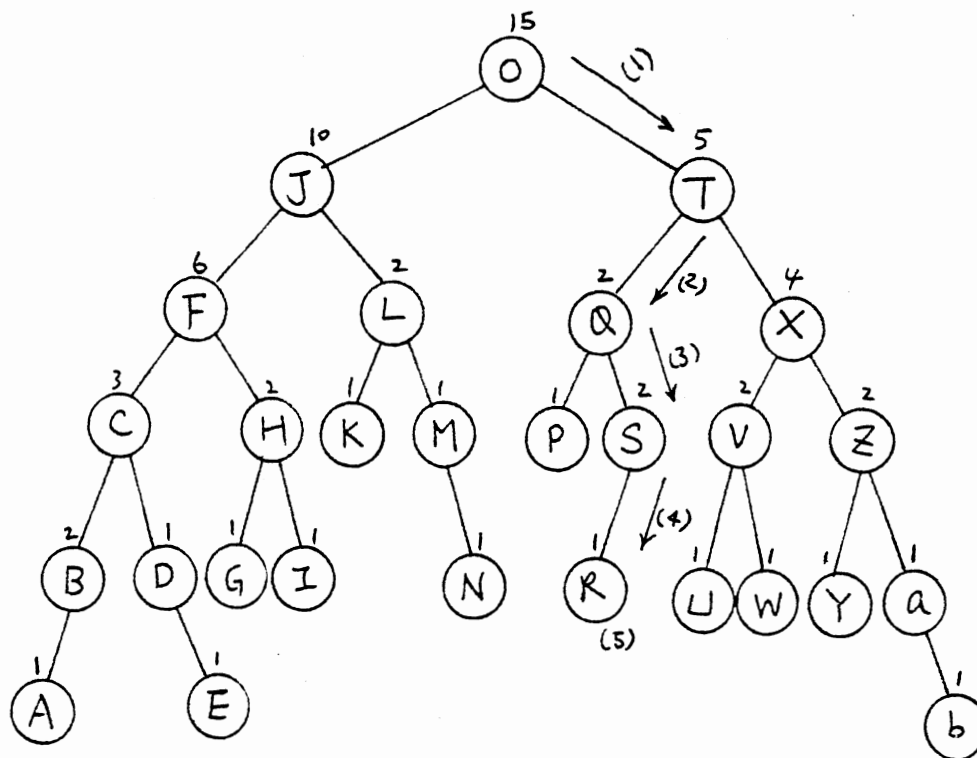


Figure 46 An Example of Top-Down Deletion from A Red-Black Tree

Figure 47  
An Example of Index-Position Search



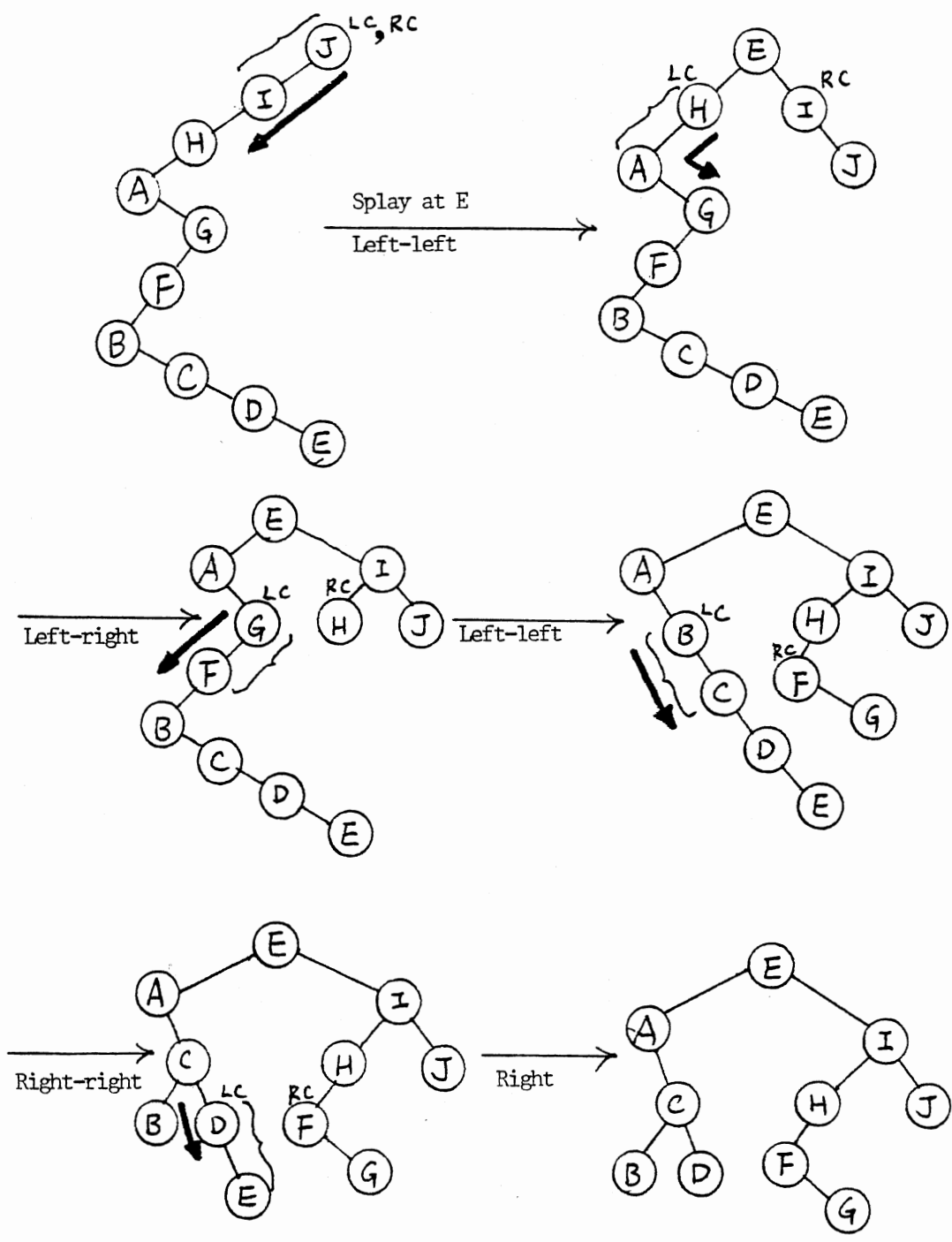
Search for the 18-th elements in the above tree.

- (1). (COUNT = 0), left access because  $18 > \text{RANK}(O)$ .
- (2). (COUNT = RANK(O)), right access because  $18 < (\text{COUNT} + \text{RANK}(T))$ .
- (3). (COUNT not changed), left access because  $18 > (\text{COUNT} + \text{RANK}(Q))$ .
- (4). (COUNT = COUNT + RANK(Q)), right access because  $18 < (\text{COUNT} + \text{RANK}(S))$ .
- (5). (COUNT not changed), stop because  $18 = (\text{COUNT} + \text{RANK}(R))$ .

<sup>15</sup>  
⊙ 15 denotes the RANK field of O.

Figure 48

An Example of Top-Down Splaying Operations



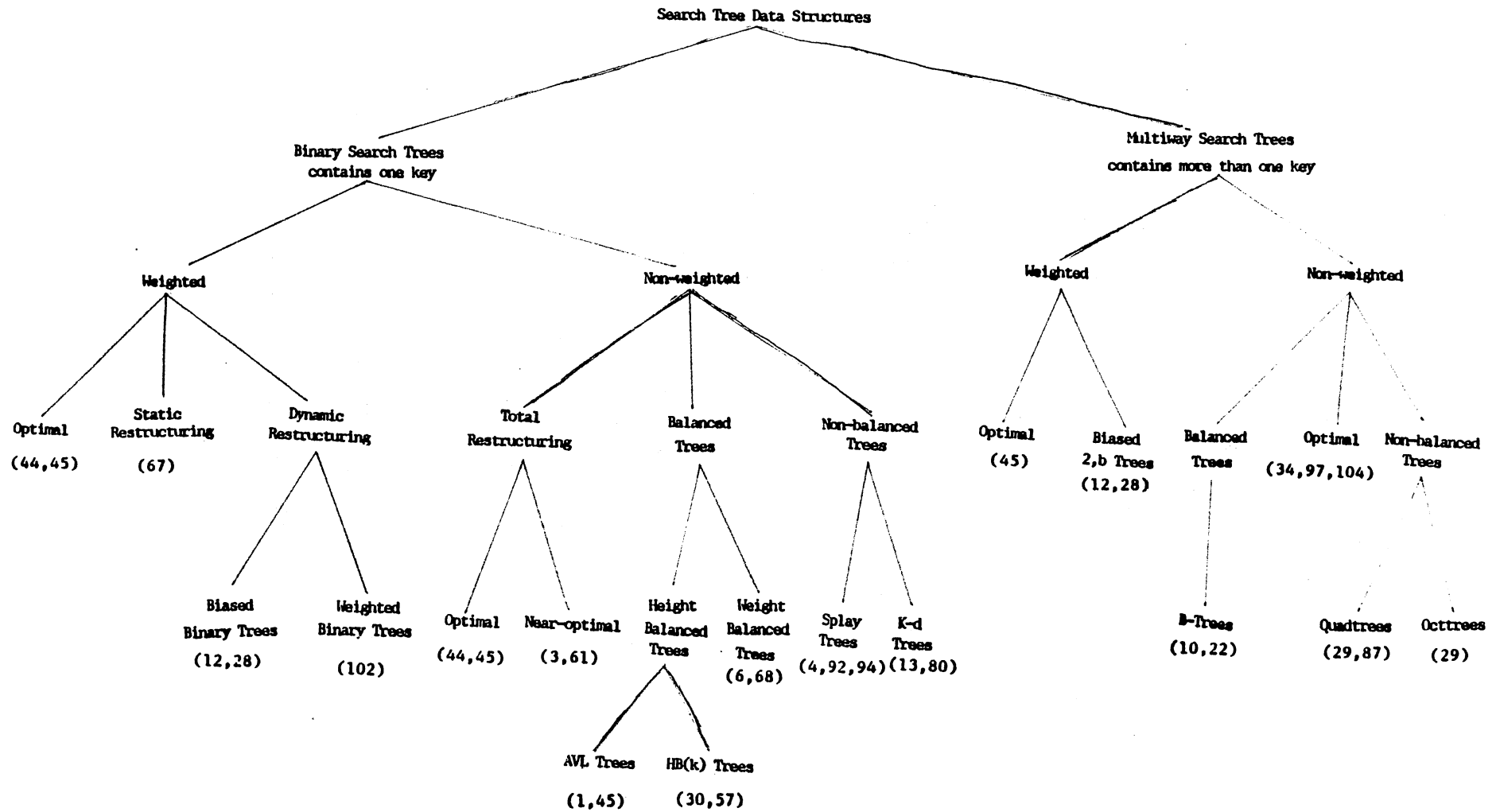


Figure 49 A Classifications of Search Tree Data Structures

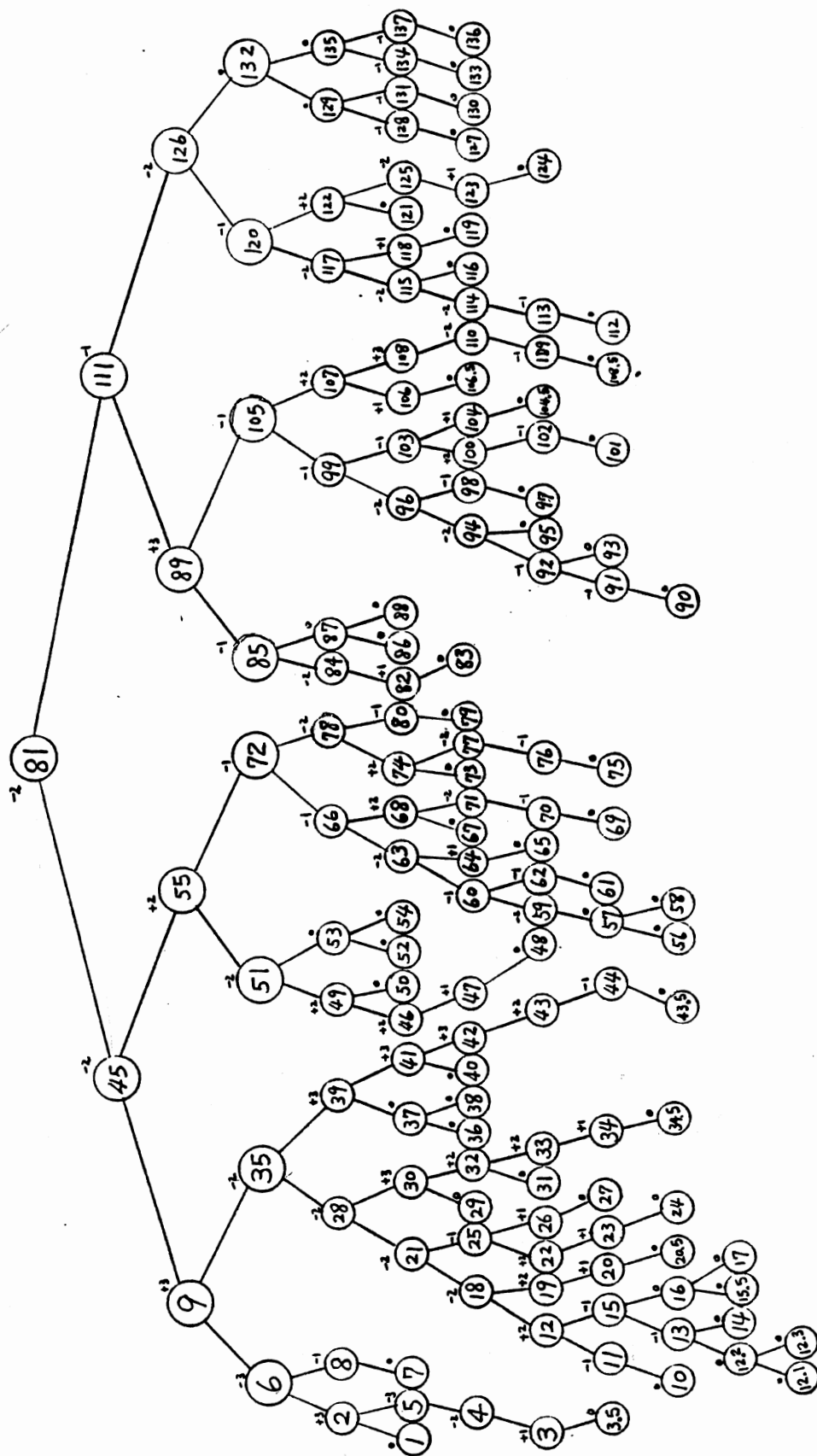


Figure 50 An Example of an HB(3)-Tree



VITA

Yick-Kwan Chen

Candidate for the Degree of  
Master of Science

Thesis: SEARCH TREE DATA STRUCTURES AND THEIR  
APPLICATIONS

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Taipei, Taiwan, Republic of  
China, January 8, 1953, the son of Mr. and Mrs.  
Yau-Sua Chen. Married to Sue-Cheng Chang on  
July 19, 1984.

Education: Graduated from National Normal High School,  
Taipei, Taiwan, in July, 1971; received the  
Bachelor of Science degree with a major in  
Chemical Engineering from National Tsing Hua  
University, in June, 1976; received the Master of  
Science degree with a major in Chemical  
Engineering from Oklahoma State University, in  
December, 1981; completed requirements for Master  
of Science degree in Computing and Information  
Science at Oklahoma State University, in May,  
1987.

Professional Experience: Research Associate, National  
Tsing Hua University, 1978-1979; Thermal Design  
Engineer, Applied Information Development Inc.,  
Tulsa, Oklahoma, 1981-1982.