EFFICIENT TOP-DOWN UPDATING OF RED-BLACK TREES

Robert Endre Tarjan

Computer Science Department
Princeton University
Princeton, NJ 08544

and

AT&T Bell Laboratories
Murray Hill, NJ 07974

June, 1985

CS-TR-006-85

Efficient Top-Down Updating of Red-Black Trees

Robert Endre Tarjan

Computer Science Department
Princeton University
Princeton, NJ 08544

and

AT&T Bell Laboratories
Murray Hill, NJ 07974

June, 1985

Abstract

The red-black tree is an especially flexible and efficient form of binary search tree. In this note we show that an insertion or deletion in a red-black tree can be performed in one top-down pass, requiring $O(1)$ rotations and color changes in the amortized case.

Efficient Top-Down Updating of Red-Black Trees

A *binary search tree* is a data structure that can be used to represent a set of items selected from a totally ordered universe. It consists of a binary tree containing the items of the set in its external nodes, one item per node, with the items in increasing order from left to right in the tree. In addition, each internal node contains an item in the universe (not necessarily in the set), called a *key*, such that all items in the left subtree of the node are less than or equal to the key and all items in the right subtree are greater than the key. One possible way to choose keys is to place in each internal node the largest item in its left subtree. (See Figure 1).

[Figure 1]

An item in the set can be accessed in time proportional to the depth of the tree by starting at the root and searching down through the tree, at each internal node branching left or right according to whether the desired item is no greater than or greater than the key in the node. Eventually the search reaches an external node; either this node contains the desired item or it is not in the set.

An n-node binary tree has $\Omega(\log n)$ depth; thus the worst-case access time in a binary search tree is $\Omega(\log n)$. There are many classes of *balanced binary trees*, such as height-balanced trees [1], weight-balanced trees [10], and red-black trees [5], that have $O(\log n)$ depth and can be rebalanced after an insertion or deletion in $O(\log n)$ time. The $O(\log n)$ depth bound follows from a *local balance constraint* that is enforced at each node. Maintaining this constraint requires storing some amount of balance information in each node. Rebalancing after an update (insertion or deletion) is done by performing local transformations, called *rotations*, on the tree structure and modifying the balance information appropriately. The structural changes take place on or near the *access path*, i.e. the path from the root to the newly inserted or deleted node. Each rotation takes $O(1)$ time and changes the depths of certain nodes while preserving the left-right order of keys and items. (See Figure 2.) Further discussion of the properties of binary search trees can be found in the books by Knuth [7] and Tarjan [13].

[Figure 2]

Some uses of binary search trees, such as in the priority search trees of McCreight [9] and in the persistent search trees of Sarnak and Tarjan [12], require that the number of structural changes per update be $O(1)$ rather than $O(\log n)$. Red-black trees satisfy this requirement. A *red-black tree* is a binary tree in which each node is colored red or black in a way satisfying the following constraints (see Figure 1):

(i) All external nodes are black.

(ii) (*black constraint*). All paths from the root to an external node contain the same number of black nodes.

(iii) (*red constraint*). The parent of any red node, if it exists, is black.

These trees were introduced by Bayer [2], who called them *symmetric binary B-trees*. The red-black definition was formulated by Guibas and Sedgewick [5]. Olivié [11] proposed an equivalent definition, that of *half-balanced trees*. Independent results of Maier and Salveter [8] and Huddleston and Mehlhorn [6] for 2,4 trees imply that updates on red-black trees can be performed in $O(1)$ rotations and color changes in the amortized case.* Tarjan [14] has given insertion and deletion algorithms taking $O(1)$ rotations in the worst case and $O(1)$ color changes in the amortized case.

These update algorithms perform the required rebalancing in a bottom-up pass that proceeds from the (previously located) inserted or deleted node up toward the root of the tree. However, there are applications in which it is convenient to do insertions and deletions in a single top-down pass from the root that simultaneously locates the node to be inserted or deleted and rebalances the tree. Top-down updating eliminates the need for parent pointers or a stack to hold the access path. It also makes concurrent tree operations [3,4] efficient; one operation need only lock a fixed number

* By *amortization* we mean averaging the cost of a worst-case sequence of operations over the sequence. Tarjan's survey paper [15] discusses this concept in detail.

of tree nodes rather than an entire access path to avoid interference.

Guibas and Sedgewick [5] have described top-down insertion and deletion algorithms for red-black trees, but their algorithms require $\Omega(\log n)$ rotations, even in the amortized case. We shall modify the bottom-up update algorithms of Tarjan [14] to produce top-down algorithms that require only $O(1)$ rotations and color changes in the amortized case. Thus the virtues of top-down updating can be obtained while preserving the $O(1)$ amortized restructuring bound of bottom-up updating.

We begin by discussing our framework for obtaining an amortized complexity bound. We use the *potential* paradigm [15]. To each red-black tree we assign a non-negative integer called the *potential* of the tree. The potential of the empty tree is zero. We define the *amortized cost* of an update operation to be the actual cost plus the net increase in tree potential caused by the operation. With this definition the total actual cost of a sequence of update operations is the total amortized cost minus the net increase in potential over the sequence. If the initial tree is empty, the net potential increase over any sequence is non-negative, and the sum of the amortized costs is an upper bound on the sum of the actual costs.

As the potential of a tree, we use a special case of the potential used by Maier and Salveter [8] and Huddleston and Mehlhorn [6] to analyze a, b trees, also known as "weak" or "hysterical" B-trees. We assign to each black internal node a potential of one if the node has no red children, zero if it has one red child, and two if it has two red children. The potential of a tree is the sum of the potentials of its nodes.

We shall describe top-down insertion and deletion algorithms having $O(1)$ amortized cost, where we charge unit cost for a constant number of rotations and color changes. The constant depends on the details of the algorithms; since we are ignoring constant factors, we shall not bother to compute its value.

The bottom-up insertion algorithm that we shall modify is as follows. First we replace the appropriate external node by an internal node having two external children, one containing the new item to be inserted and the other containing the item in the replaced node. The key of the new

internal node is the minimum of the items in its two children. We color the new internal node red. (See Figure 3(a).)

[Figure 3]

This preserves the black constraint (ii) but may violate the red constraint (iii). If a red node x has a red parent $p(x)$ whose sibling is also red, we color $p(x)$ and its sibling black and the grandparent $g(x)$ of x red. (See Figure 3(b).) This will cause a new violation of (iii) if the parent of $g(x)$ is red. We repeat this recoloring step until no new violation is created or a red node x has a red parent $p(x)$ that is the root or whose sibling is black. To eliminate the last violation we apply the appropriate one of the transformations in Figures 3(c), 3(d) and 3(e).

We make several observations about this insertion process. Each application of a case in Figure 3 takes $O(1)$ rotations and color changes and increases the potential by at most two. The only non-terminating case is 3(b), each application of which causes only color changes; each application but the last causes the potential to drop by one. It follows that a bottom-up insertion takes $O(1)$ rotations in the worst case and $O(1)$ color changes in the amortized case.

Our top-down insertion method proceeds from the root down along the access path, maintaining the invariant that the current node is black and has at least one black child. To make the invariant true initially, we let the current node be the root and change it to black if it is red or change both its children to black if they are both red. The general step of the insertion consists of walking from the current node, say x, down along the access path, until one of the following cases occurs:

(a) An external node is reached. Proceed as in bottom-up insertion. (The rebalancing terminates when x is reached bottom-up.)

(b) A black node, say y, with a black child is reached. Replace the current node x by y and repeat the general step.

(c) Four successive black nodes, each with two red children, are reached along the access path. Let z be the bottom-most such node. Color z red and its two children black. Proceed as in

bottom-up insertion to eliminate the resulting violation of the red constraint (see Figure 4.) This takes three applications of 3(b) followed possibly by an application of 3(c) or 3(d). The potential drops by at least one. Replace the current node x by the child of z along the access path and repeat the general step.

[Figure 4]

Since case (a) is terminal, case (b) does not change the tree, and case (c) takes $O(1)$ rotations and color changes and decreases the potential by at least one, the amortized cost of an insertion is $O(1)$ as desired.

Deletion is similar to insertion but more complicated. To discuss deletion we need the concept of a *short node*. A node is short if all paths from it down to an external node contain one less black node than all paths down from its sibling. The bottom-up deletion algorithm that we shall modify is as follows. We find the external node containing the item to be deleted and replace its parent by its sibling. (See Figure 5(a).) If the replaced node was black, the replacing node is short. We push the shortness up the tree by repeating the transformation in Figure 5(b) until it no longer applies. Then we perform the transformation in Figure 5(c) if appropriate. Finally, we apply 5(b), 5(d), 5(e), or 5(f) to eliminate the last shortness.

[Figure 5]

Case 5(b), the only repeated one, causes only color changes and produces a potential drop of two. Case 5(c) causes no change in potential; the other cases cause the potential to increase by at most two. It follows that a deletion takes $O(1)$ rotations in the worst case and $O(1)$ color changes in the amortized case.

Our top-down deletion algorithm maintains the invariant that the current node is red or has a red child or grandchild. The initial current node is the root; to make the invariant true initially we color the root red if it is black with two black children. The general step of the algorithm consists of proceeding from the current node, say x, down along the access path until one of the following

cases occurs:

(a) An external node is reached. Proceed as in bottom-up deletion. (The rebalancing terminates when x is reached bottom-up.)

(b) A node, say y, that is red or has a red child or grandchild is reached. Replace the current node x by y and repeat the general step.

(c) Three successive black nodes with all black children and grandchildren are reached along the access path. Let z be the bottom-most such node. Color z and its sibling red, making its parent short. Eliminate the shortness as in bottom-up deletion. (See Figure 6.) This takes one application of 5(b) followed possibly by 5(c) followed by 5(b), 5(d), 5(e), or 5(f). The potential drops by at least one. Replace the current node x by z and repeat the general step.

[Figure 6]

Since case (a) is terminal, case (b) does not change the tree, and case (c) takes $O(1)$ rotations and color changes and decreases the potential by at least one, the amortized cost of a deletion is $O(1)$.

We conclude with three remarks. First, our insertion and deletion algorithms can be easily modified to work in purely top-down rather than globally top-down but locally bottom-up fashion. Second, our techniques generalize to provide efficient top-down update algorithms for a, b trees. Third, there are other arrangements of keys in a binary search tree to which our algorithms extend. The most common alternative is to store the items in the internal nodes, one item per node, in symmetric order. Thus the items themselves are the keys, and external nodes need not be represented. With this arrangement top-down insertion is essentially unaffected, but top-down deletion becomes harder because if the item to be deleted is in a node with two internal children, it must be swapped with its predecessor or successor before the tree restructuring takes place. This requires either a second pass along the access path or the storage of extra pointers in the tree, and it makes the problem of avoiding deadlock during concurrent tree operations much harder.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis, "An algorithm for the organization of information," *Soviet Math. Dokl.* 3 (1962), 1259-1262.

[2] R. Bayer, "Symmetric binary *B*-trees: data structure and maintenance algorithms," *Acta Inform.* 1 (1972), 290-306.

[3] R. Bayer and M. Schkolnick, "Concurrency of operations on *B*-trees," *Acta Inform.* 9 (1977), 1-21.

[4] C. S. Ellis, "Concurrent search and insertion in 2-3 trees," *Acta Inform.* 14 (1980), 63-86.

[5] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," *Proc. 19th Annual IEEE Symp. on Foundations of Computer Science* (1978), 8-21.

[6] S. Huddleston and K. Mehlhorn, "A new data structure for representing sorted lists," *Acta Inform.* 17 (1982), 157-184.

[7] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[8] D. Maier and S. C. Salveter, "Hysterical *B*-trees," *Inform. Process. Lett.* 12 (1981), 199-202.

[9] E. M. McCreight, "Priority search trees," *SIAM J. Comput.* 14 (1985), 257-276.

[10] J. M. Nievergelt and E. M. Reingold, "Binary search trees of bounded balance," *SIAM J. Comput.* 2 (1973), 33-43.

[11] H. J. Olivïe, "A new class of balanced search trees: half-balanced trees," RAIRO Informatique Théoretique 16 (1982), 51-71.

[12] N. Sarnak and R. E. Tarjan, "Persistent search trees and geometric retrieval," to appear.

[13] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[14] R. E. Tarjan, "Updating a balanced search tree in $O(1)$ rotations," *Inform. Process. Lett.* 16 (1983), 253-257.

[15] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Alg. Disc. Meth.* 6 (1985), 306-318.

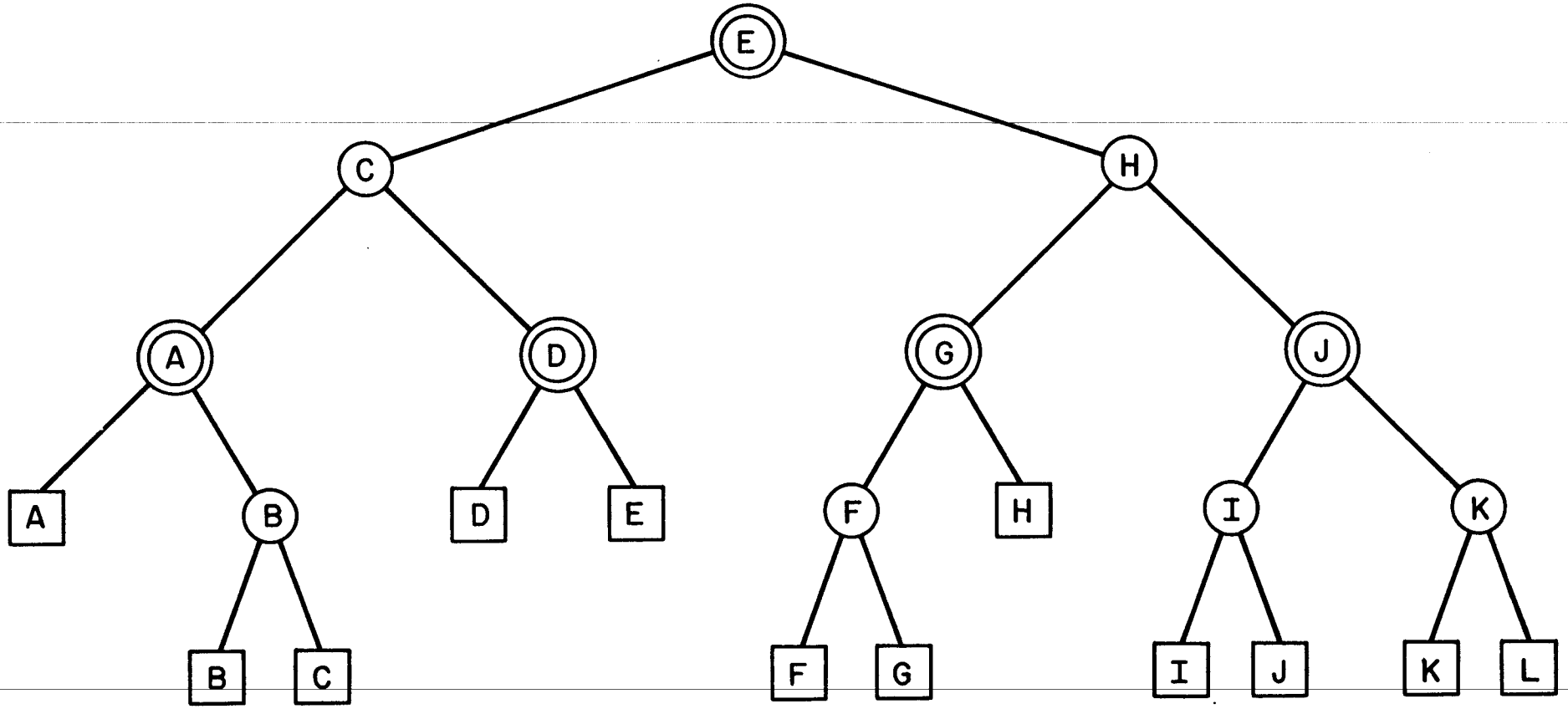

Figure 1. A red-black search tree. The double circles denote black internal nodes, the single circles red internal nodes, and the squares external nodes.

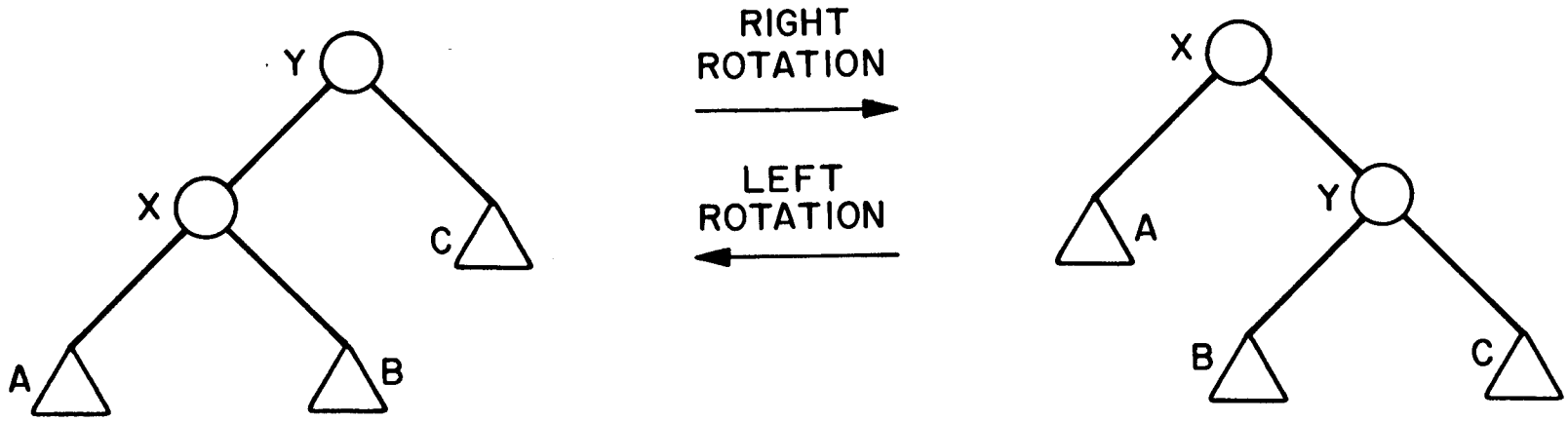

Figure 2. A rotation. The triangles denote arbitrary subtrees. The tree shown can be a subtree of a larger tree.

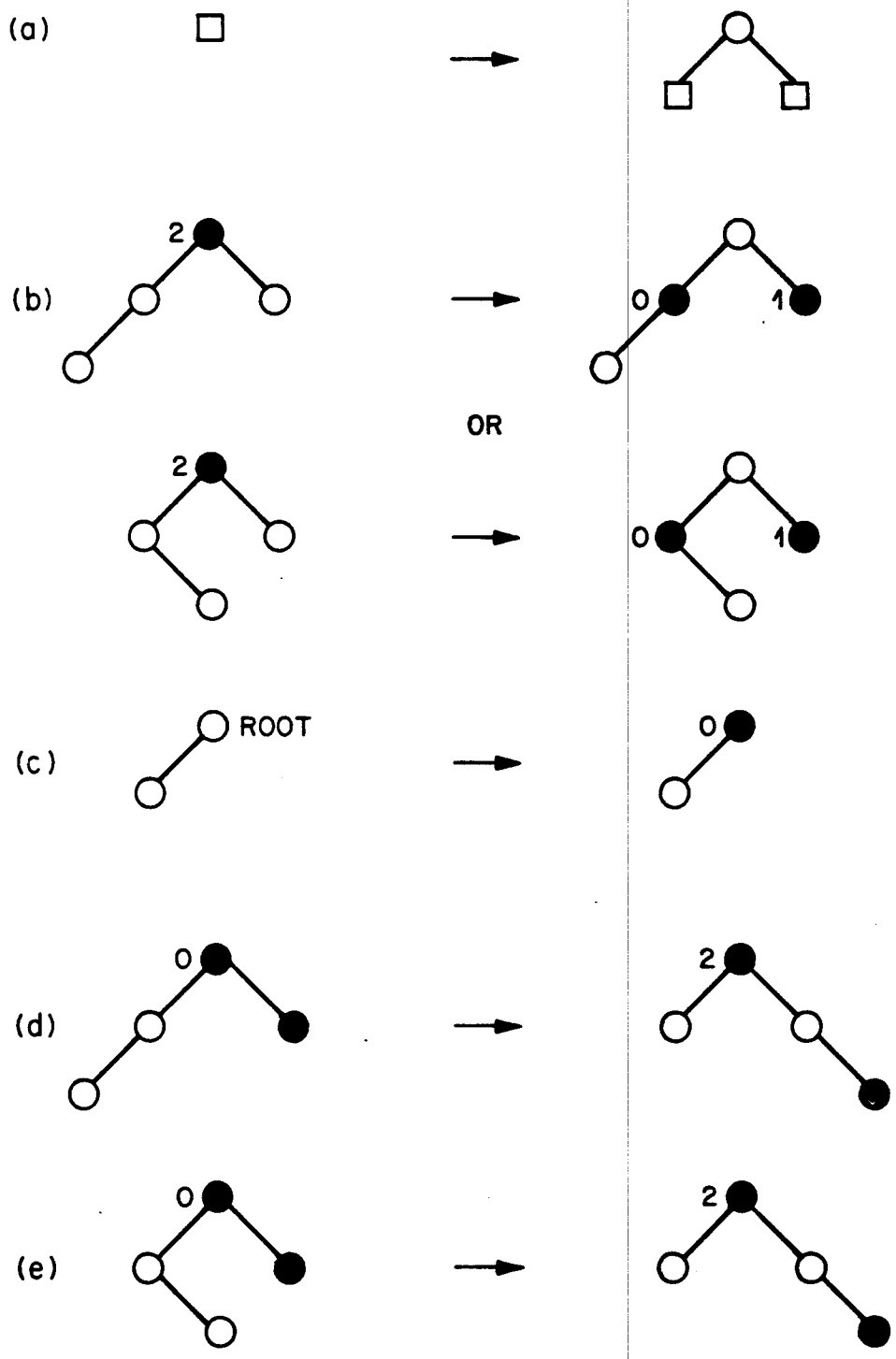

Figure 3. The cases in bottom-up insertion. Symmetric cases are not shown. Hollow nodes are red, solid nodes are black. The numbers are node potentials. All missing children of red nodes are black. In cases (d) and (e) the bottommost black node shown can be external.

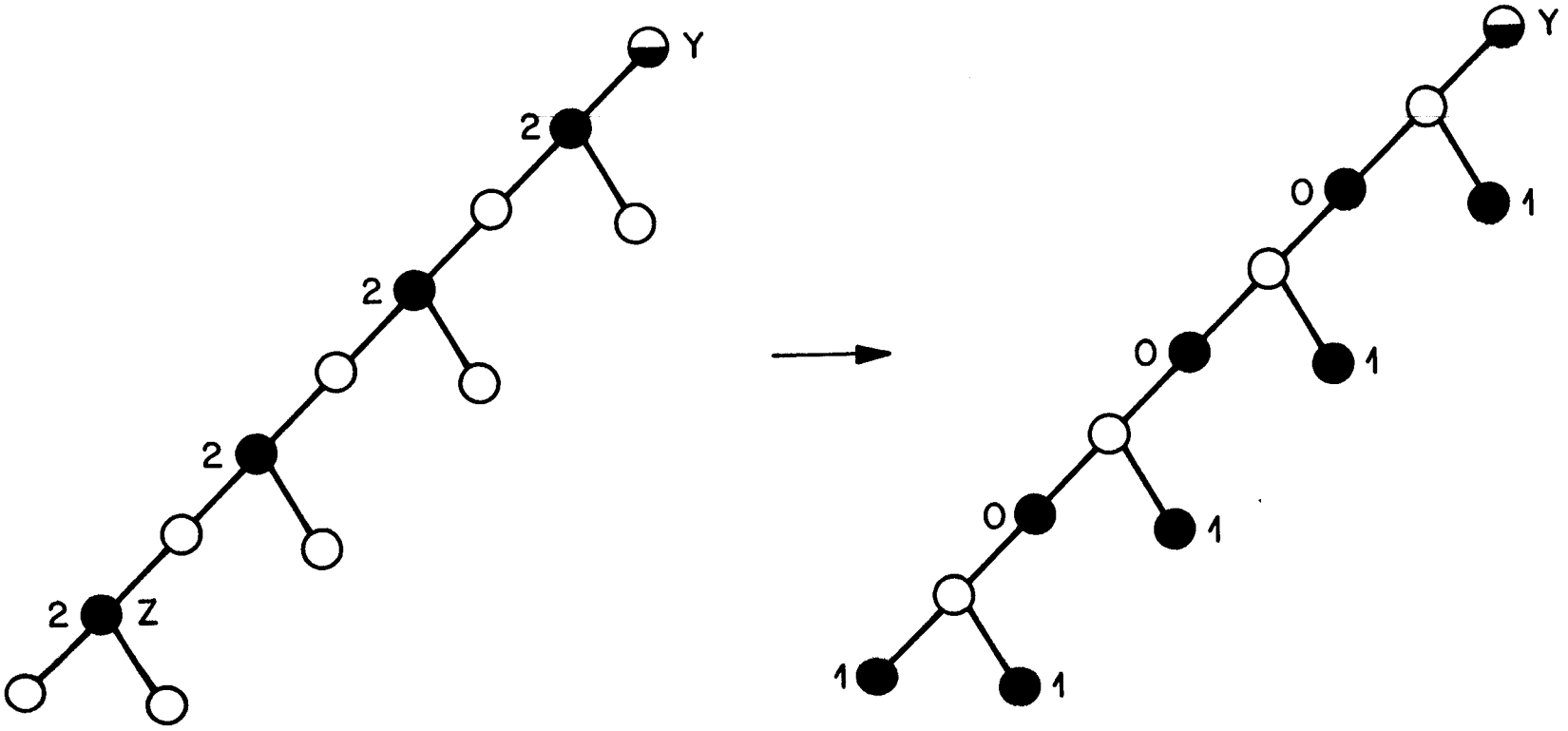

Figure 4. A typical application of case (c) in the top-down insertion algorithm. The half-filled node y is ambiguous in color, either black or red. If node y is black, it is x, and the application of case (c) takes only color changes. If node y is red, its parent is x and the indicated color changes are followed by an application of 3(c) or 3(d).

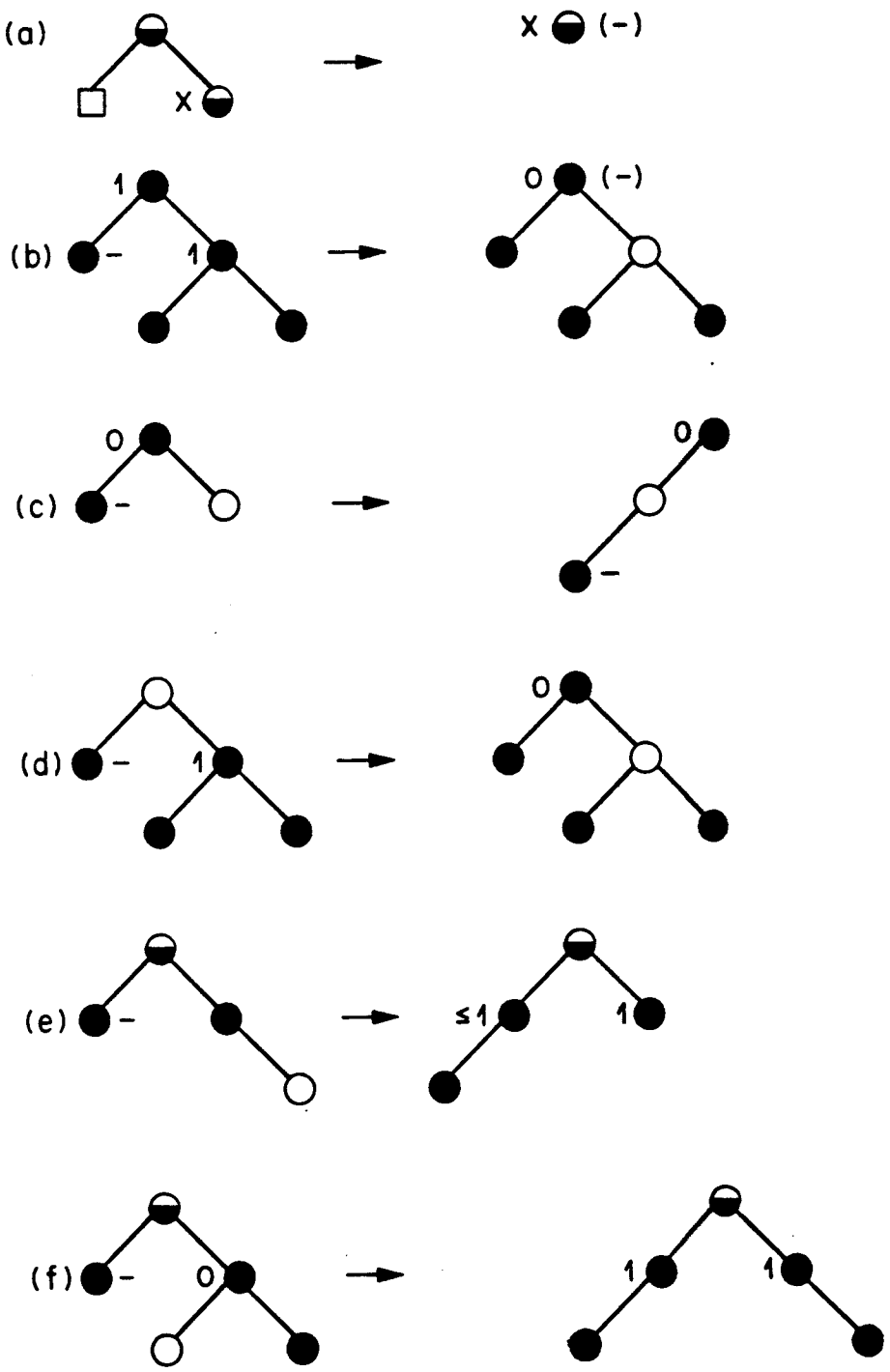

Figure 5. The cases in bottom-up deletion. The two ambiguous nodes in (e) have the same color, as do the two ambiguous nodes in (f). Minus signs denote short nodes. In (a) the replacing node x, which can be either internal or external, is short if the replaced internal node was black. In (b) the top node after the transformation is short unless it is the root.

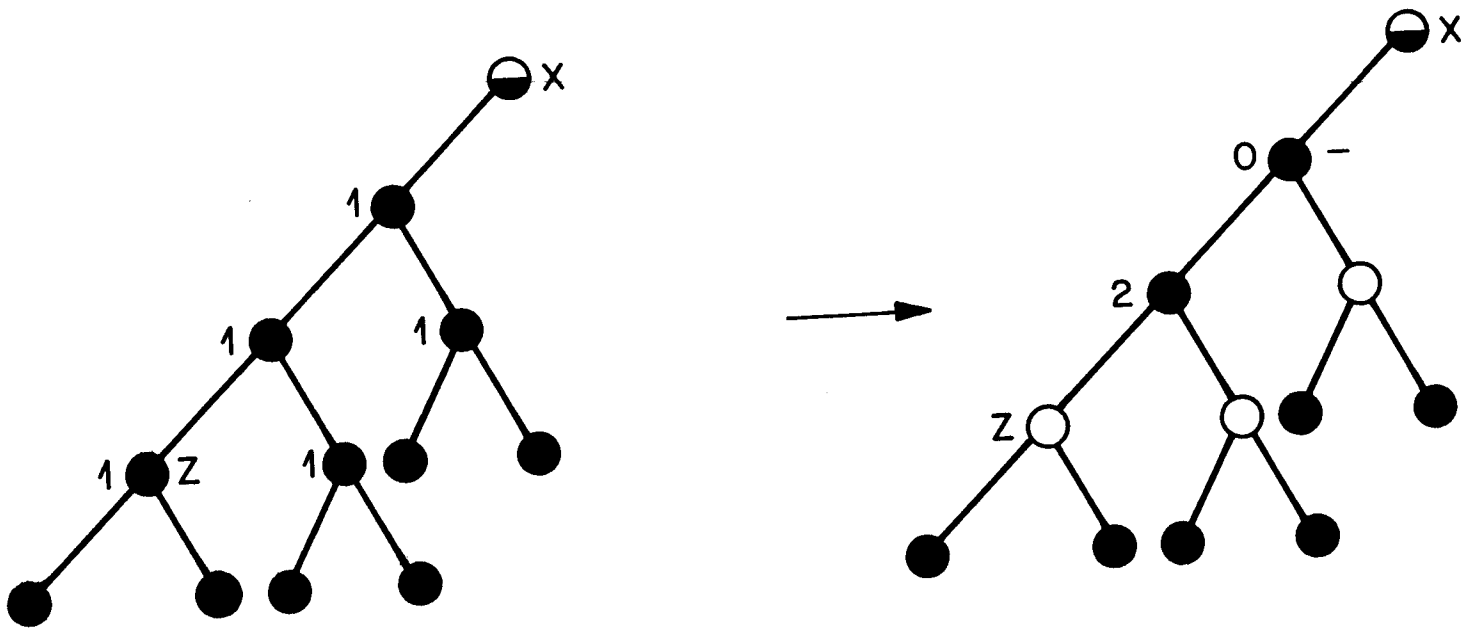

Figure 6. A typical application of case (c) in the top-down deletion algorithm. The shortness is eliminated by applying 5(c) if necessary followed by 5(b), 5(d), 5(e), or 5(f).