Red-Black Trees

Thomas A. Anastasio

17 March 2000

1 Introduction

(These notes on red-black trees expound on Chapter 14 of "Introduction to Algorithms," Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, MIT Press, 1992.)

• Red-black trees are binary search trees with a more relaxed notion of balance than AVL trees.

• In a red-black tree, no path from a node N to a leaf is more than twice as long as any other (proved in Theorem 3.

• For a red-black tree of n nodes and height h, $h \leq 2\lg(n+1)$ (proved in Theorem 4).

◇ **Definition**: A red-black tree is a binary search tree with the following additional properties:

1. Every node is either red or black.

2. Each `NULL` pointer is considered to be a black node.

3. If a node is red, then *both* of its children are black.

4. *Every* path from a node to a leaf contains the same number of black nodes.

Although it is not necessary for the definition, we adopt the further property that the root is black.

◇ **Definition**: The *black height* of a node N in a red-black tree is the number of black nodes on any path to a leaf, not counting N itself.

Figure 1 shows a full binary search tree of 15 nodes (`NULL` pointers not shown), then a number of equivalent red-black trees with the `NULL` pointers shown as black nodes.

2 Bottom-Up Insertion In a Red-Black Tree

• The first phase of insertion is done in the usual binary search tree fashion. Recursively descend the tree and insert the new node as a leaf.

• Every node is red upon insertion. This *may* cause violation of red-black properties.

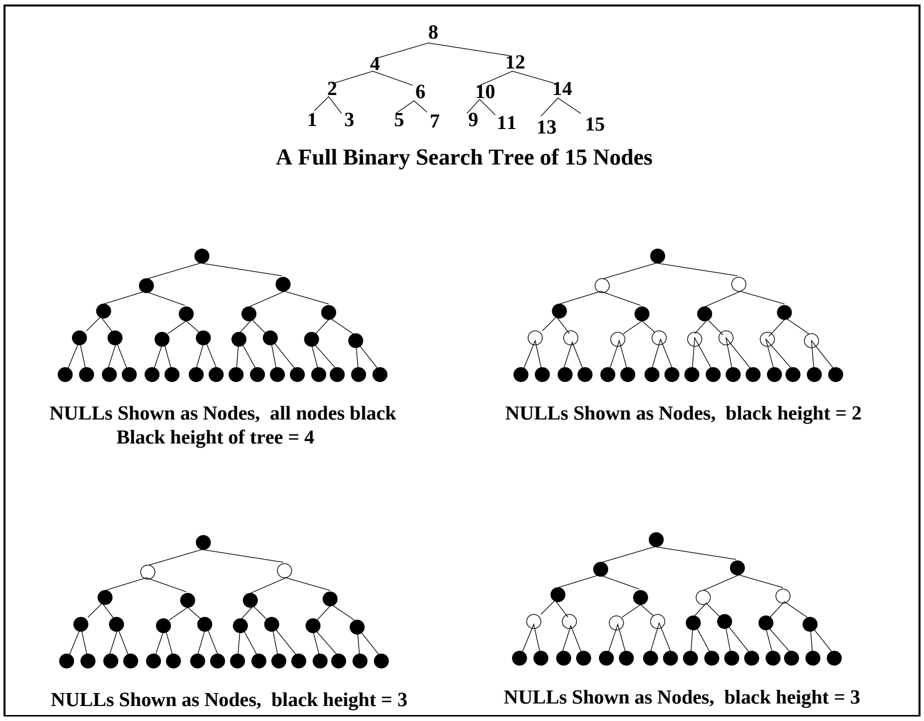

Figure 1: Examples of Red-Black Trees Based on Full BST of 15 Nodes

What red-black properties *could* be violated by insertion of a node? The binary search tree property would not be violated since insertion is done as for binary search trees. Looking at the additional properties for red-black trees:

1. Every node is red or black: **not** violated.

2. The leaf nodes are the `NULL` pointers from the nodes: **not** violated.

3. Every leaf (`NULL`) is black: **not** violated.

4. If a node is red, then *both* of its children are black: **not** violated for the inserted node (both `NULL` children are black); but **may be** violated for the parent of the inserted node.

5. *Every* path from a node to a descendent leaf contains the same number of black nodes: **not** violated.
   Proof: Suppose the path to the insertion point contained k black nodes before insertion. Insertion of the red node would reduce the path to $k-1$ black nodes, but the `NULL`s on the inserted node add 1 black. Therefore the number of black nodes on the paths is unchanged.

• The only violation possibility is if the inserted node has a red parent.

• Therefore, the only situation in which adjustment of the tree must be done after insertion is when the parent of `X` is red.

2.1 Details of the Insertion Process

Figure 2 summarizes the rules for insertion of a node into a red-black tree. At the start, `X` is a pointer to the newly-inserted node.

• Note: the inserted node does not move. The pointer `X` does.

• The color of the "uncle" node (sibling of parent node) is important in determining the actions to be taken.

There are four cases identified in the figure:

Case 0 `X` is the root.

Case 1 Both parent and parent's sibling ("uncle") are red.

Case 2 Parent is red but uncle is black. `X` and its parent are both left or both right children.

Case 3 Parent is red but uncle is black. `X` and its parent are opposite type children.

Note: In the Weiss text[1], Case 2 corresponds to Figure 12.10(top), and Case 3 corresponds to Figure 12.10(bottom). Weiss describes his Case 1 in the last paragraph of page 504. His approach is to perform the appropriate Case 2 or Case 3 rotation, then recolor. The approach described in this paper is to recolor, then rotate only if a red-red violation occurs between grandparent and great-grandparent.

2.2 Example of Insertion into a Red-Black Tree

Figure 3 shows how a red-black tree is modified after insertion of the node with value `4`. For clarity, the `NULL` nodes are not explicitly shown.

1. `X` points to the inserted node `4`. Node `4` is red since every node is red upon insertion. Since the parent of node `4` is red, correction is necessary. Node `4`s uncle (node `8`) is red, so case 1 applies.

   (a) Color the parent (node `5`) black.

   (b) Color the uncle (node `8`) black.

   (c) Color the grandparent (node `7`) red.

   (d) Point `X` at the grandparent.

2. Now, `X` points to node `7` which has a red parent (node `2`). The uncle of node `7` (namely node `14`) is black and `X` is a right child. Therefore, case 2 applies.

   Step 1

   (a) Color the grandparent (namely node `11`) red.

   (b) Color `X` (node `7`) black.

[1] Mark Allen Weiss, *Data Structures & Algorithm Analysis in C++, 2nd ed.*, Addison-Wesley, 1998

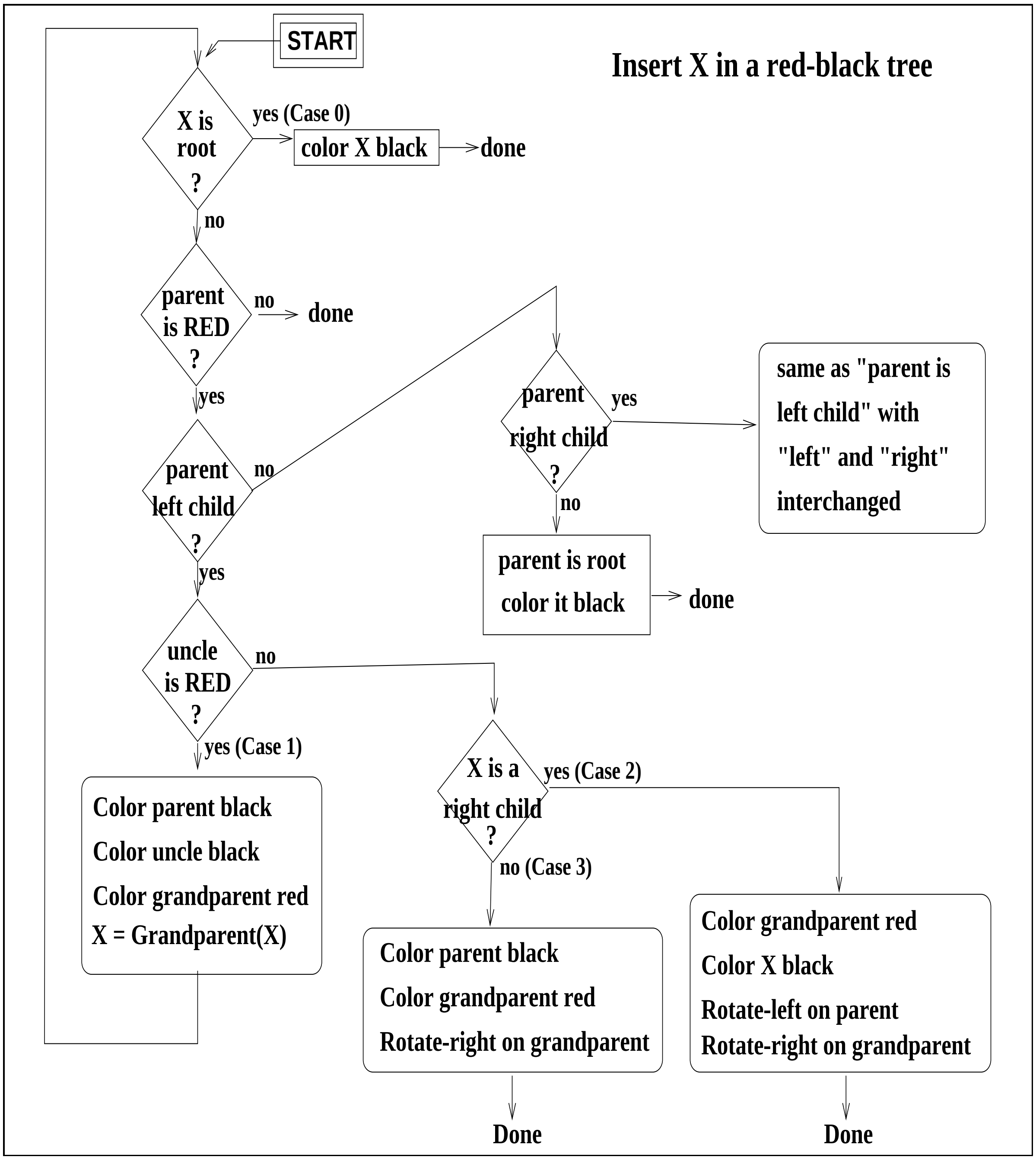

Figure 2: Flow Chart for Insertion in Red-Black Trees

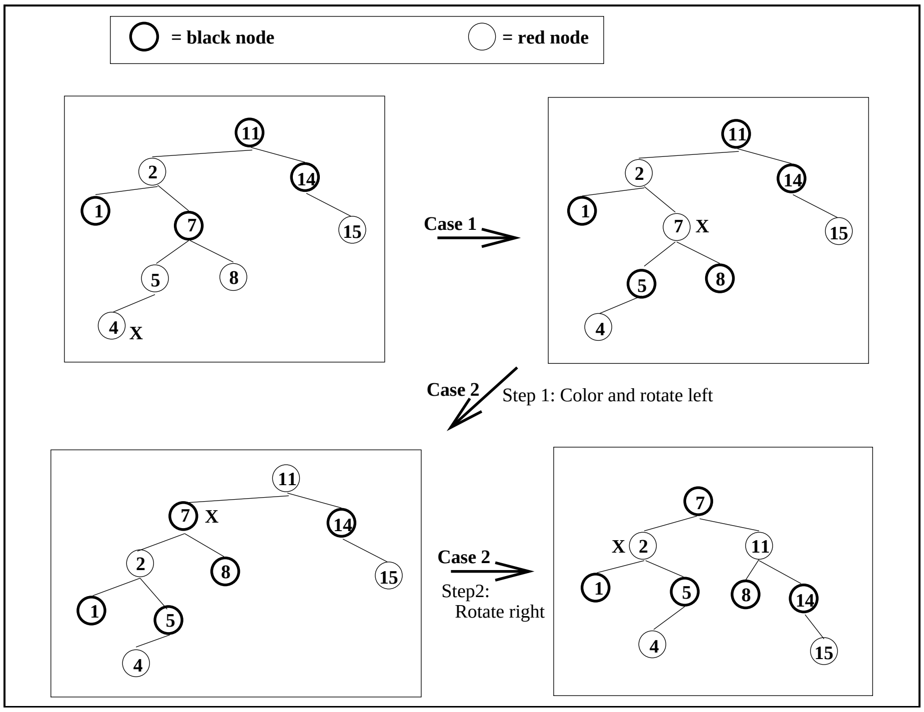

Figure 3: Example of Insertion in a Red-Black Tree

(c) Rotate left about `X`'s parent node `2`.

Step 2 Rotate right about the grandparent node `11`.

Theorem 1 Any red-black tree, with root x, has at least $n = 2^{\mathrm{bh}(x)} - 1$ internal nodes, where $\mathrm{bh}(x)$ is the black-height of node x.

$\triangleright$ **Proof**: by induction on height of x.
Base: x is a leaf (no internal nodes), height is zero, $\mathrm{bh}(x) = 0$. $2^0 - 1 = 0$. Therefore true for base case.
IH: Let x be an internal node with two children (which may be `NULL`, and assume the theorem holds for all nodes of height less than x.
If a child of x is red, its black height is $\mathrm{bh}(x)$. If the child is black, its black height is $\mathrm{bh}(x) - 1$. In any event, the IH holds for the children since their height is less than that of x. Therefore, each child subtree has at least $2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Therefore, the tree rooted at x has at least

$$2(2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$$

internal nodes. (The $+1$ counts node x itself.) $\triangleleft$

Theorem 2 In a red-black tree, at least half of the nodes on any path from root to a leaf must be black.

$\triangleright$ **Proof**: If there is a red node, there must also be a black node. $\triangleleft$

Theorem 3 In a red-black tree, no path from any node N to a leaf is more than twice as long as any other path from N to any other leaf.

$\triangleright$ **Proof**: By definition, every path from a node to a leaf contains the same number of black nodes. By Theorem 2, at least half of the nodes on any such path must be black. Therefore, there can be no more than twice as many red nodes on any path from a node to a leaf. Therefore the length of every path is no more than twice as long as that of any other path. $\triangleleft$

Theorem 4 A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$

$\triangleright$ **Proof**: Let h be the height of the red-black tree. By theorem 2, $\mathrm{bh}(x) \geq h/2$. Therefore,

$$n \geq 2^{\frac{h}{2}} - 1$$
$$n - 1 \geq 2^{\frac{h}{2}}$$
$$\lg(n-1) \geq \frac{h}{2}$$
$$2\lg(n-1) \geq h$$

$\triangleleft$

2.3 Asymptotic Cost of Insertion

- Theorem 4 states that the height of a red-black tree is $O(\lg n)$
  Therefore, cost of insertion is:
  - $O(\lg n)$ to descend to the insertion point.
  - $O(1)$ to do the insertion.
  - $O(\lg n)$ to ascend. Note that if case 2 or case 3 are done, the loop is terminated – these cases are done only once.

  Therefore, insertion is in $O(\lg n)$.

3 Bottom-up Deletion

Recall the rules for bottom-up deletion in ordinary binary search trees:

1. If the node to be deleted is a leaf - just delete it.
2. If the node to be deleted has just one child, replace it by that child.
3. If the node to be deleted has two children, replace the value at the node by the value at its inorder predecessor, then recursively delete the inorder predecessor.

To do bottom-up deletion in a red-black tree, first perform ordinary bottom-up deletion following the rules above. Eventually one of the two base cases will apply (delete a leaf or delete a node with just one child). Let u be the node to be deleted at that point.

1. If u is a leaf, we think of deletion as replacement by the null-pointer V. Recall that in red-black trees, the null-pointers are black.
2. If u has just one child, V, we think of deletion as replacement of u by V.

If u is red, then deleting it does not change any red-black property and no adjustments are necessary. However, if u is black (and is not the root) deleting it will change the black-length along some path. The "blackness" of u that is lost can be thought of as giving extra blackness to V. We must adjust the tree to move this extra blackness up the tree and eventually having it absorbed somewhere. This is just an accounting trick, but it makes the adjustments more understandable.

There are four cases to be considered (see Figure 4).

1. Node S, the sibling of V, is red. Do a rotation and recoloring to produce a situation that can be handled as one of the other three cases.
2. Node S is black and both of its children are black. Color S red. If P is red, absorb the extra blackness at P and terminate. Otherwise, one of cases 2, 3, or 4 applies, so continue.

3. Node S is black and its right child is red. Do a rotation followed by a color swap between S and P. This is a terminal case; no further work is needed.

4. Node S is black, its left child is red and its right child is black. Do a rotation around S and a recoloring. The situation can now be handled as a case 3, which is terminal.

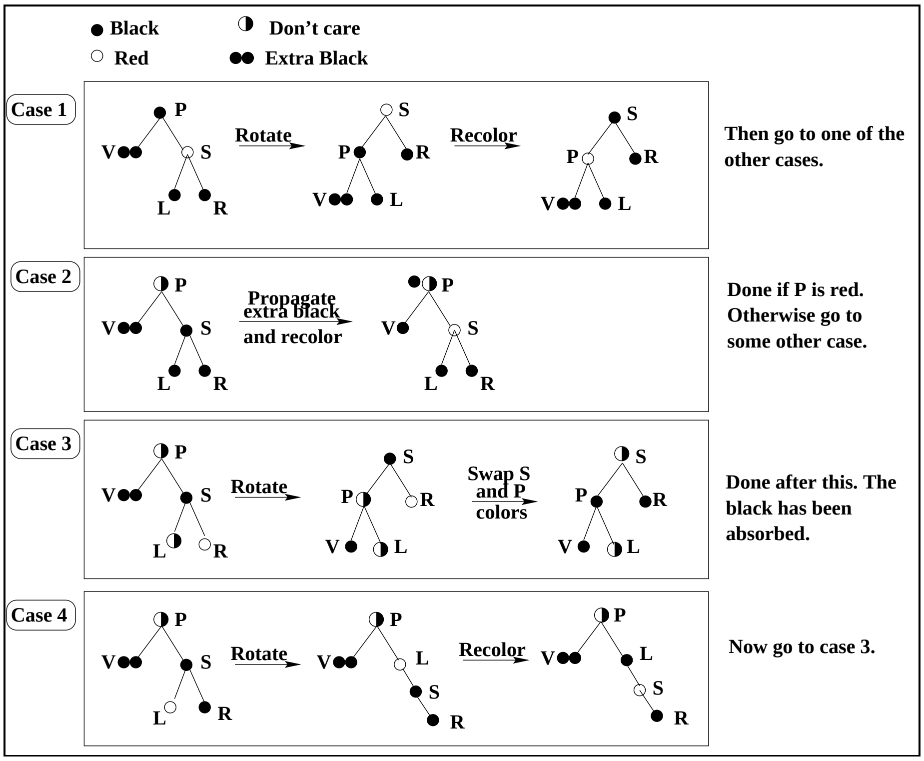

Figure 4: Cases for Bottom-Up Deletion in a Red-Black Tree

3.1 Examples of Bottom-Up Deletion in a Red-Black Tree

Example 1 is shown in Figure 5. The node holding the value 90 is to be deleted. It is black, therefore deleting it will cause a red-black property of the tree to become invalid. We replace 90 by its child null pointer V and indicate that V has extra blackness. This is case 2, so propagate the extra blackness up the tree to node 80. Since it is red, it "absorbs" the extra blackness. Finally, color the sibling 70 red.

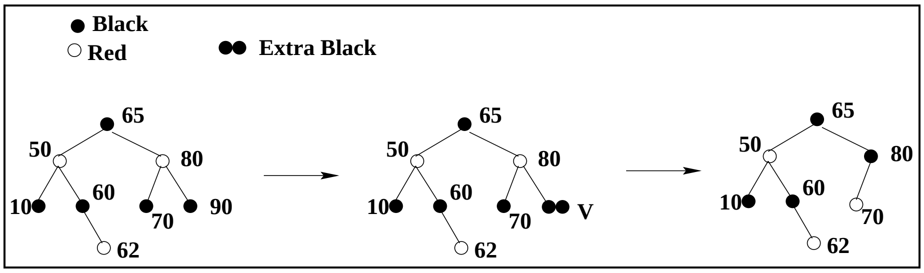

Figure 5: Example 1: Bottom-Up Deletion

Example 2 is shown in Figure 6. The node holding the value 80 is to be deleted. Since it has just one child, we replace it by that one child, namely 70. This causes the red node 70 to have extra blackness. Since it's red, it absorbs the extra blackness, making node 70 black.

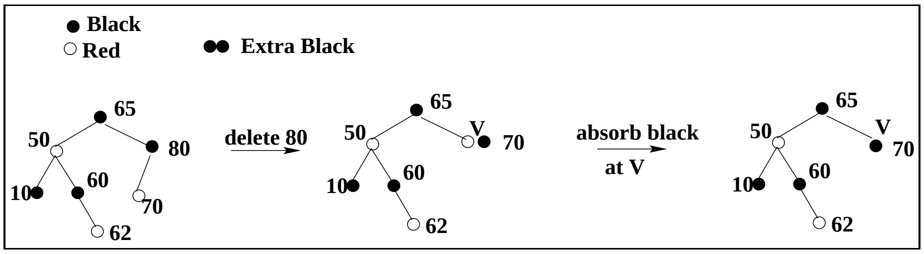

Figure 6: Example 2: Bottom-Up Deletion

Example 3 is shown in Figure 7. The node holding the value 70 is to be deleted. V is the former null pointer and has the extra blackness from node 70. Since the sibling of V (node 50) is red, this is case 1 (symmetric version). Handling the case results in a new tree in which the sibling of V (node 60) is black. This is case 4 (symmetric version). Handling that case results in a new tree in which the sibling of V (node 62) is black and the sibling's left child is red. This is case 3 (symmetric version). Handling this case results in the final tree shown.

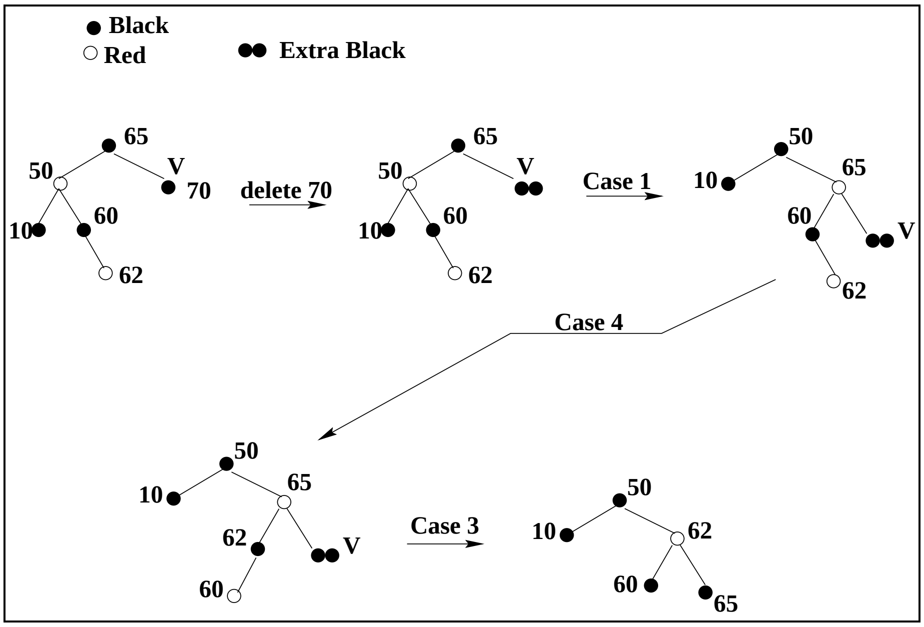

Figure 7: Example 3: Bottom-Up Deletion