RED-BLACK TREE INSERTION AND DELETION

1. Binary-Search Tree

The "binary-search" invariant is that a node is greater or equal to its left child and less than or equal to its right child.

1.1. **Insertion.** Simple: just walk down the tree going left or right depending on whether the value you're inserting is greater or smaller than the value at the node.

1.2. **Deletion.** Simple: if the node you're deleting only has one child just replace it with its child. The "binary-search" invariant isn't violated and you're all set. If the node you're deleting has both of its children then you need to replace it with a node that will preserve the "binary-search" invariant: either its predecessor or successor[1]. The successor (or the predecessor) is deleted outright and replaced with its appropriate child.

2. Red-black Tree

A Red-black tree is a Binary-search[2] tree that further satisfies the invariants

(1) Every node is "colored" either red or black.
(2) The root of the tree is black.
(3)
(4) No two red nodes in a row, i.e. if a node is red then both of its children are black.
(5) Every path down from a node to a leaf should have the same number of black nodes (called black-depth).

A conseqeuence of satisfying these invariants is that for any node x in the tree with n nodes it's the case that

$$\lg(n) < \text{height}(x) \leq 2\lg(n+1)$$

(conventional height). In this sense a Red-black tree is balanced.

2.1. **Insertion.** Insert a red node using standard BST insertion. Which Red-black properties could be violated? Property 1 holds since you inserted a red node. Property 5 holds since you didn't add any black nodes. Property 2 might be violated if the node you inserted replaced the root and Property 4 might violated if you inserted underneath a red node. So work needs to be done inorder re-assert the invariant. The rules appear in figure 2.1. Note that if y's parent is red then the fix needs to be repeating with y's grandparent (which will be black[3]). If you get all the way to the top and the new y, i.e. the root, is still not black then just color it black (why is this okay?).

---

[1]Finding the successor or predecessor is easy: go to the left child and then go all the down the chain of right children or the obverse go to the right child and all the way down the chain of left children. Note the successor won't have a left child and the predecessor won't have a right child (why?)

[2]Stolen from http://cs.wellesley.edu/~cs231/fall01/red-black.pdf

[3]why?

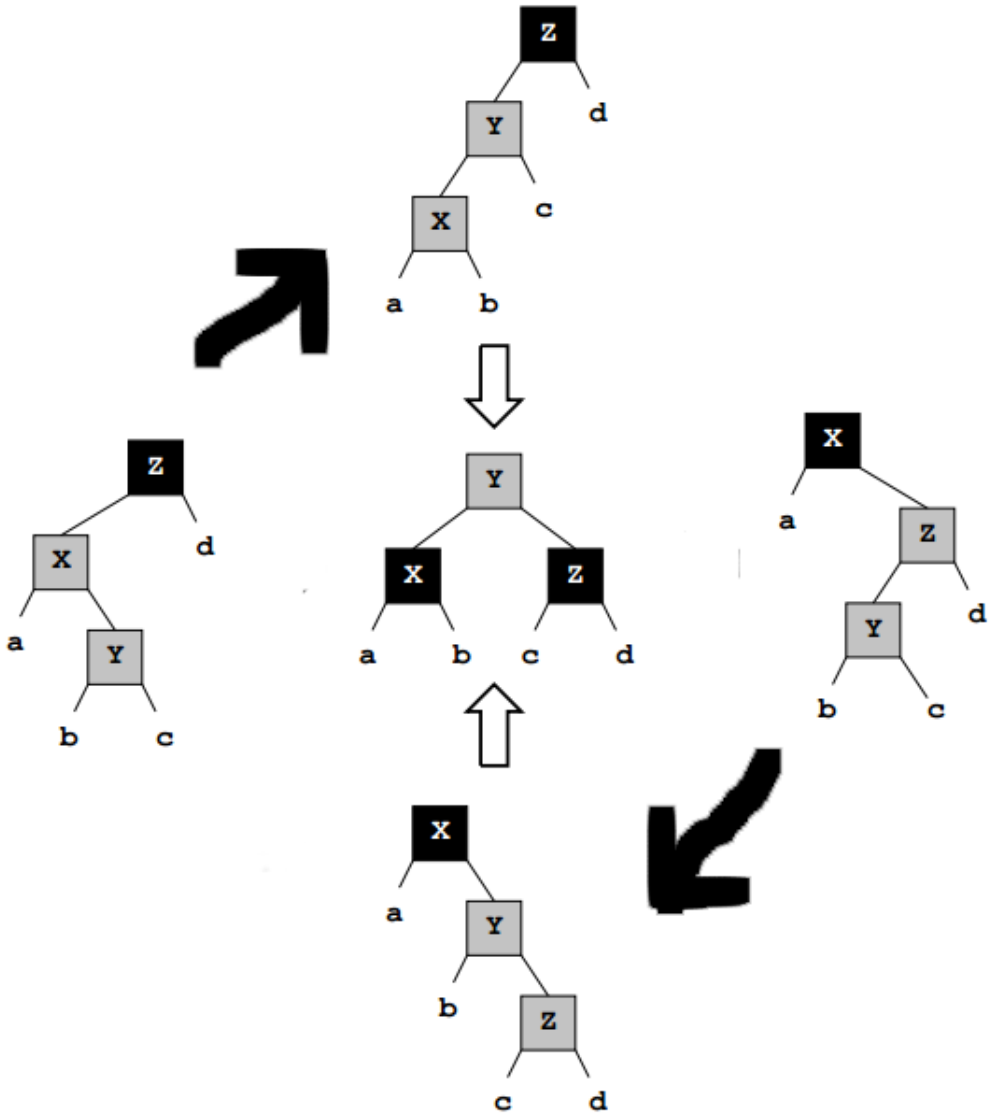

FIGURE 2.1. RB Insertion fix

2.2. **Deletion.** Use the usual BST deletion algorithm. Recall that in BST deletion a node with only one child is replaced at some point: either the original node has only one child (and is therefore replaced by that child) or the original node's successor (predecessor) is replaced by its right (left) child. Let this node (the one with only one child, by whom it is replaced) be called N and the child that moves into N's position be called a. It's important to keep N and a straight. In the case where the originally deleted node has only one child N is the original node and a is the child it's replaced with. In the case where the original node deleted is replaced by its successor N is its successor and a is the successor's right child that moves into N's original position.

Which properties might be violated? If N was the root and a red node becomes the root then property 2 is violated. Otherwise if N's parent was red and N's child was red then property 4 is violated (note N is necessarily black then). Finally if N is black then property 5 (black-depth) is violated by moving it up the tree (any path that contained N has one fewer black node on it now). We can correct this by making the child a that replaced N be "extra" black. This is a bookkeeping device and no really modification to a is done. This restores property 5 but violates property 1 (what does it mean to be "extra" black?).

The solution is to propagate the "extra" blackness up the tree according to the following rules until you get to the root (at which point it can just be discarded). Note that these are rules for a being the left child of its parent - the rules for a being the right child are completely symmetric. Big black squares represent black nodes, the small black square is "extra" blackness, white squares represent unknown/irrelevantly colored nodes, grey nodes represent red color nodes, triangle represent subtrees with color at the root:

Case A (3,4 CLRS) : The sibling of the "extra" black node is black and at least one nephew is red. This is taken care of by one or two rotations (figure 2.2; black square denotes "extra" blackness)

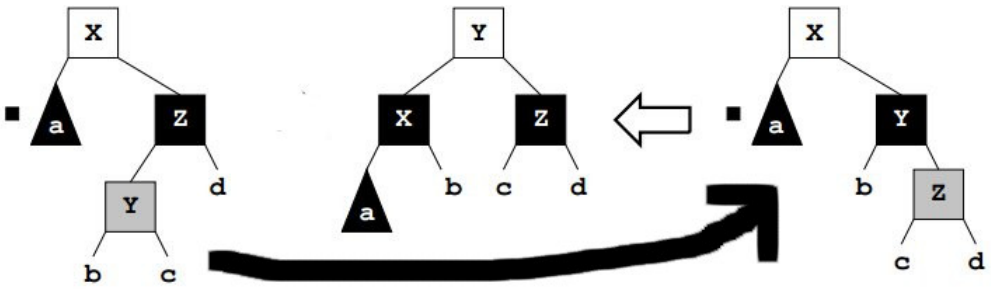

FIGURE 2.2. Sibling black and at least one newphew red; small black square denotes "extra" blackness fix

Case B (2 CLRS) : The sibling of the "extra" black node is black and both nephews are black. This rule propagates the blackness up (figure 2.3)

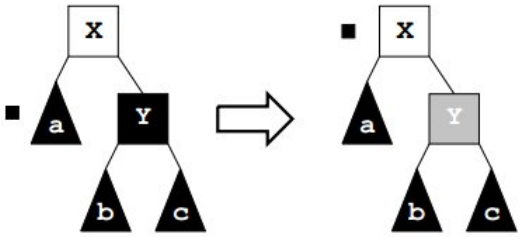

FIGURE 2.3. Sibling black and both newphews red; black square denotes "extra" blackness fix

Case C (1 CLRS) : The sibling of the "extra" black node is red. This rule transforms to either case A or B (figure 2.4)

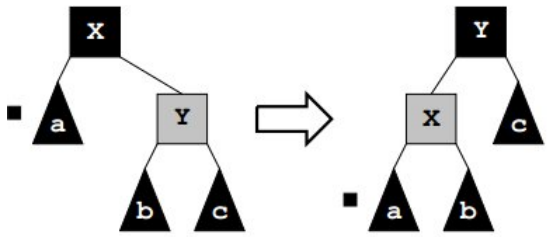

FIGURE 2.4. Sibling red; black square denotes "extra" blackness fix

2.2.1. *Summary.* 3 or 4 cases triggered depending on whether the node that gets replaced its direct child is black. Call this node "extra" black and

(1) If sibling is black with one red nephew: (make it be the right nephew with a right rotation and recolor if not already) rotate left around the parent keeping what's rotated into position the same color and distribute the blackness to a and old nephew.
(2) If sibling is black and both nephews black: push blackness up and color sibling red.
(3) If sibling is red: rotate into one of the other cases.