Red Black tree - continued

Insert in a Red Black tree

Requires re-organization, to maintain balanced tree properties of the Red Black tree. This can be done through operations such as <u>changing the color</u> of a node, and/or <u>rotation</u>.

We'll first discuss rotation, and then look at insert (will skip delete which is even more involved)

Rotation: a local operation that is used generally for balanced trees, including Red Black trees, and preserves the binary search tree properties (of ordering of keys). It runs in constant time. We typically use this to rebalance a tree after some modification has been made. It's not particular to Red Black, although we will use this in Red-Black examples of insert later.
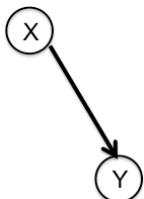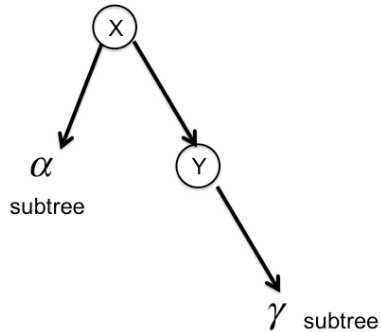
Schematic:
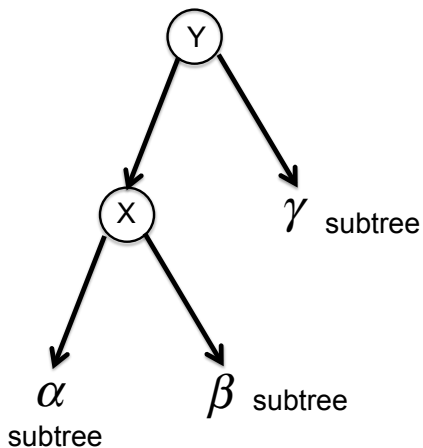
In this tree: keys in X.key < Y.key

We also have: alpha < X.key < beta < Y.key < gamma
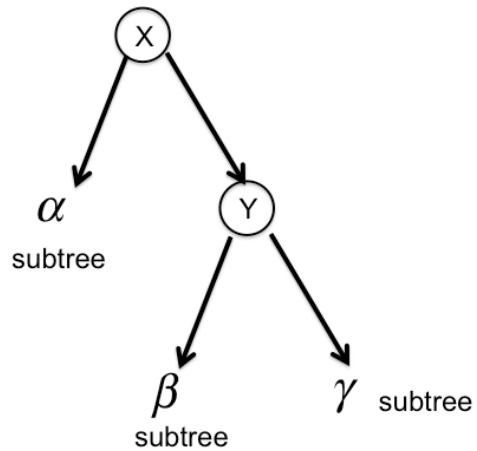
<u>We want to exchange the parent y and child x:</u>

We changed the direction (rotate right) to maintain: X.key < Y.key

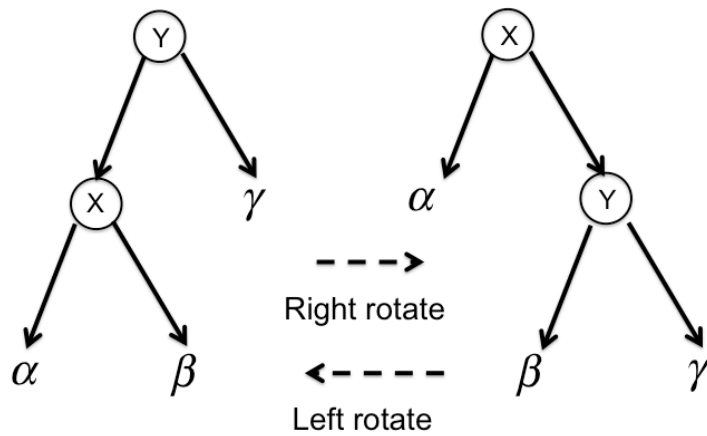After doing so, we need to also restructure the remaining tree to maintain the ordering of the keys of the subtrees.

X's left and Y's right children stay the same

Turn X's previously right subtree into Y's left subtree

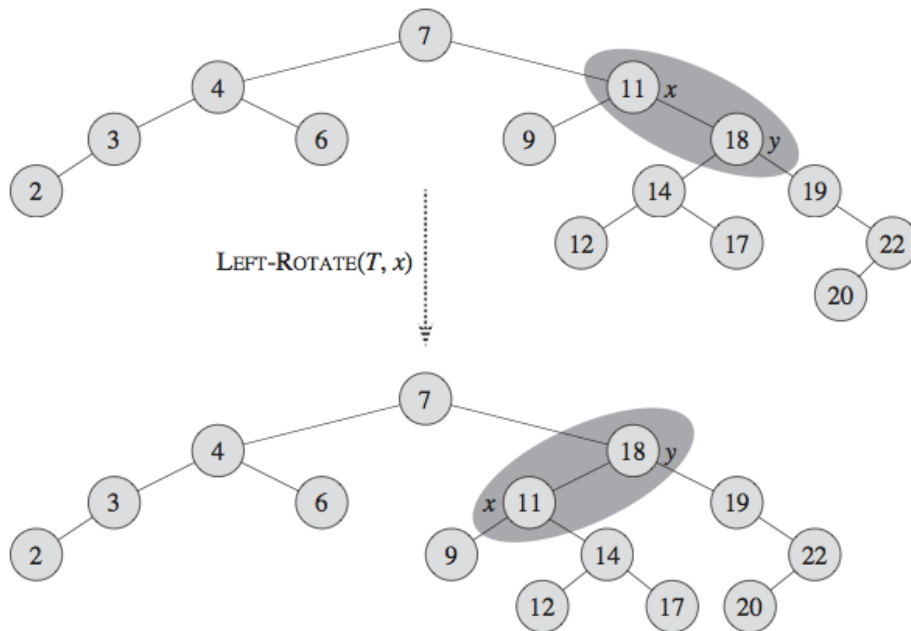As before: alpha < X.key < beta < Y.key < gamma

Summary schematic:

Summary for rotation:

- We exchanged parent Y and child X, and restructured tree
- Constant number of operations O(1)
- Preserves binary search tree (ordering) properties

Another example with numbers and left rotation:

Y's left subtree became X's right

X's left subtree remained the same and Y's right subtree remained the same

Insertion:

Main idea for regular insertion in a Binary Search Tree:

Start from the root of the tree; each time left or right depending on whether new node key smaller or larger than current node. Example: insert node 13
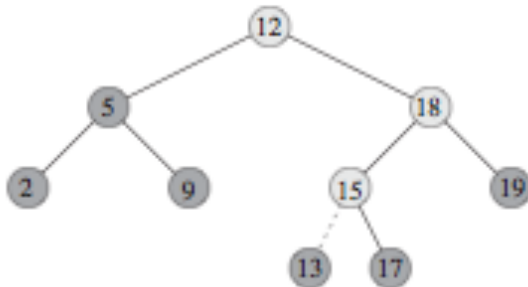
Main idea for insertion in Red Black Tree:

Insert node z:

1. Do regular insertion as in Binary Search Tree
2. Color inserted node z Red (why?)
3. Fix-up: Either recolor nodes or perform rotations such that Red Black properties preserved (this is done recursively starting from the inserted node and up the tree until a valid Red Black tree is obtained)

What properties of Red Black tree might be violated when inserting?

1. No two Reds in a row (property 4)
2. The root is black (property 2; easy fix at the end; just recolor red to black)
3. What about number of black nodes (Black height) the same on any path from node to leaves? (property 5). It won't be violated initially because we made the inserted node Red, but might be violated during the fix up.

Conditions for fix-up after inserting node z:

The fix-up occurs after we have inserted z and colored it Red.

We'll discuss 3 main cases (plus an extra case 0 in which no fix is needed). In practice there are rather than 6 cases rather than 3 main cases, but each set of 3 are entirely symmetric (exchanging left for right).

Case 0:

If z's parent (denoted z.p) is Black, then there is no violation (since z is Red and previously we had a Red Black tree). We are done.

<u>Otherwise (cases 1-3):</u>

<u>Otherwise, we have a violation, since z's parent is Red and we have two Reds in a row.</u> Moreover, since the Red Black tree before insertion was valid, then z's parent (which is Red) cannot be the root. Therefore, z has a grandparent (denoted z.p.p), and that grandparent is Black (otherwise there would have been two reds in a row prior to inserting z). All cases follow from this basic setup in which parent z.p is Red and grandparent z.p.p is Black.

<u>Case 1:</u> Other child of grandparent of z is also Red. This case applies whether z

Is a right (a) or left (b) child of its parent. Figure from the Cormen textbook:
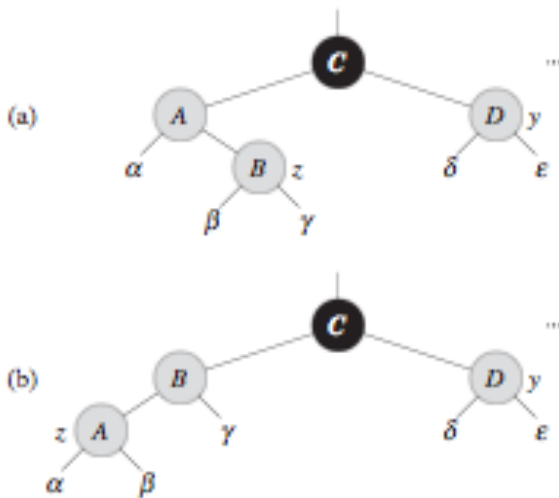
Here z (also labeled node A) is the inserted node; we denote the grandparent (node C) z.p.p, and the other child of the grandparent y (also labeled node D). We draw the subtrees of the nodes, and note that this could either happen at the initial stage of insert (in which case z's children are nil) or during a recursive step.

The solution here only requires recoloring:
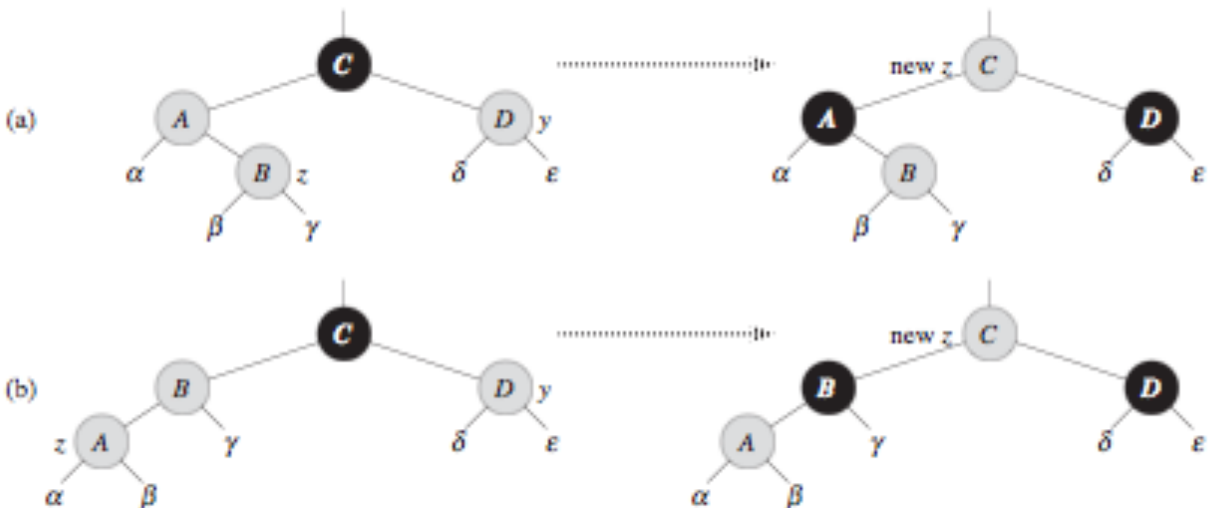
Main steps for case 1:

- z.p and y are colored Black

- Grandparent z.p.p is colored Red

- Are we done? Not necessarily. Grandparent now becomes the new z and we
  recurse up the tree and check again if there is a violation of two Reds in a row.

Cases 2 and 3: Other child of z.p.p is Black (or nil), and either z is a right child
(case 2) or z is a left child (case 3). The main aspect that visually distinguishes
case 2 from case 3 is that in case 3 we have a straight 3 chain between the node
z and the grandparent, and in case 2 it is a curved shape. Note also that in
detailing cases 2 and 3 below, we are not drawing the symmetric and equivalent
case in which the tree is flipped and every right node becomes a left node and
vice versa.

Case 2: Other child of z.p.p is Black (or nil), and z is a right child (for the
configuration below; more generally, we see the curved shape between z, the
parent, and the grandparent). Case 2 can always be converted to case 3 by a
single rotation:

Case 2   Case 3
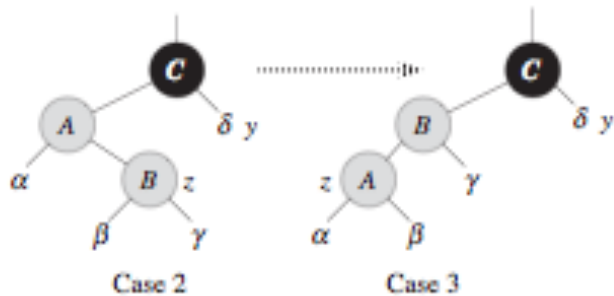
Note again that we are assuming that case 2 (or 3) can come up during the
recursion, and therefore node z may have children, and the figure above from the
Cormen textbook includes the relevant subtrees.

Note that case 2 (and 3) can also happen initially upon insertion, but only if the
other child of grandparent z.p.p (labeled y) is both black AND nil (otherwise we
would violate the Black height property). The tree before insertion is always
assumed to be a valid Red Black tree.

We will only discuss the solution to case 3 (taking account that if case 2 is
encountered, we always convert it to case 3 by rotation).

Case 3: Other child of z.p.p is black (or nil), and z is a left child (for the configuration below; more generally, we see the straight chain shape between z, the parent, and the grandparent).
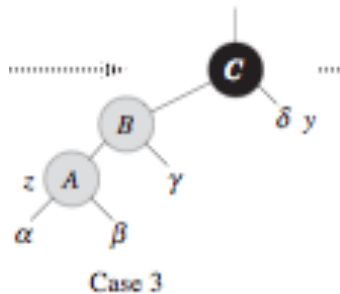
Case 3

Main steps for case 3: The fix in this case includes both recoloring (between z.p and z.p.p) and rotation (between z.p and z.p.p):
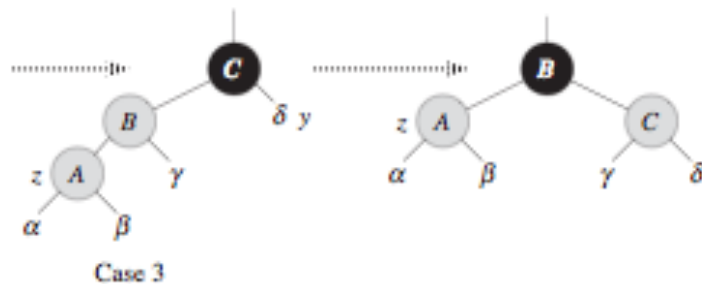
Case 3

Following the recoloring and rotation, we have a valid Red Black tree. Note that after applying the fix for case 3 one time, we are done, because there is no longer a two Reds in a row violation; the Black height property is maintained; and the root is guaranteed to be Black.

Short summary/note for inserting in a Red Black Tree: While the parent of z (z.p) is red, we recurse up the tree, each time fixing violations (each time we either observe a double Red violation, or at the very top the root might be Red). We fix the violations by recoloring (case 1), or by recoloring and/or rotations (cases 2 and 3). When we reach case 3, we are done (we will therefore only encounter case 3 one time at the most). When we reach case 2, we rotate to case 3 and then are done. The number of rotations is therefore at most two rotations at the end of the procedure (two rotations if we encounter case 2, and 1 rotation if we encounter case 3), and the remaining changes are color changes.

Run time: O(log n), since we are recursing up the height of the tree.

Example:

(a)

Case 1

(b)

Case 2

(c)

Case 3

(d)