CS211 - Data Structures

Spring 2013

Professor Christopher Andrews

Red-Black Trees

Key

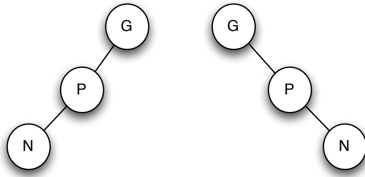
Gray nodes are of unknown color
Nodes with a double circle are either in violation or are potentially in violation
The nodes are named by their starting status (N - violating node, P - parent, S - sibling, G - grandparent) or numbered (numbered nodes are potentially standing in for whole subtrees)

Basic Rules

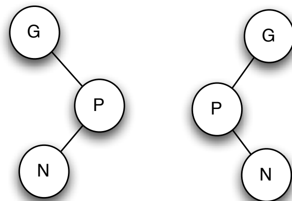
1. nodes are either red or black
2. newly inserted nodes are red
3. the root node is black (this is easy to enforce since we can alway make the root black if it isn't)
4. red nodes have two black children
5. for every simple node from a node to a descendant leaf contains the same number of black nodes (the number of black nodes is called the **black-height**)

Definitions

outer child the left child of a left child, or the right child of a right child

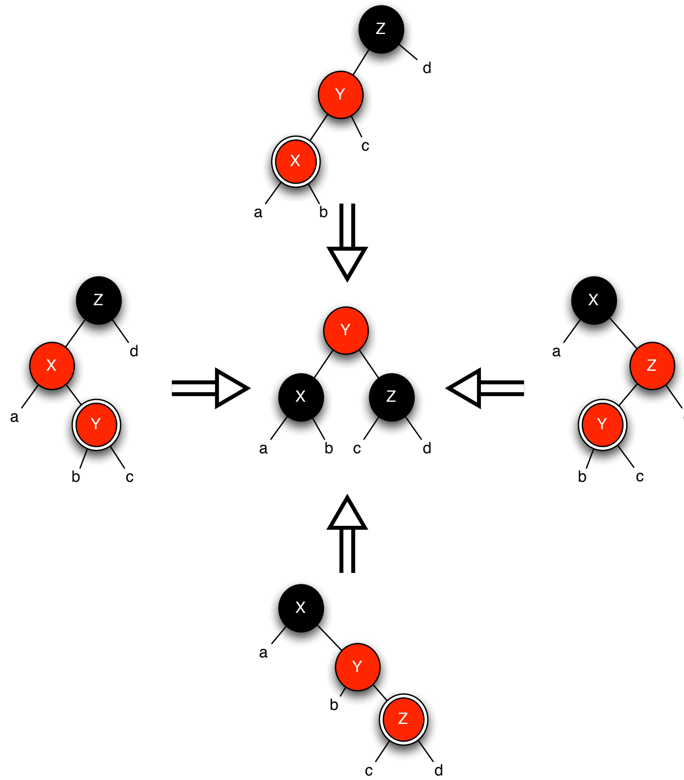


inner child The left child or a right child or the right child of a left child



Insertion

After insertion, we have a red violation if the parent of the new node is red. Our general strategy is to flatten the subtree and push the black in the grandparent down a level so that the number of black nodes increases without changing the black height of the subtree. [*This technique was taken from "Purely Functional Data Structures" by Chris Okaski.*]



We can implement this by realizing that this is really two cases:

violating node is an outer child
the top and bottom cases
rotate the parent
color the violating node black

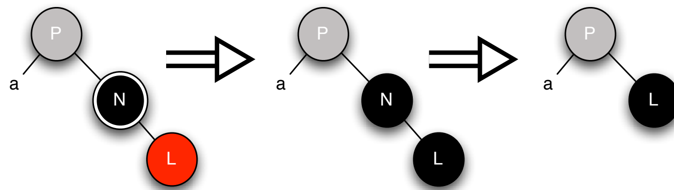
violating node is an inner child
the left and right cases
rotate the violating node twice
color the parent black

Deletion

Deletion causes a violation when we remove a black node since that will change the number of black nodes in one of the subtrees of the node's parent (provided the node isn't the root, in which case, we are fine). Our general approach is to try to "remove" the blackness from the node, since removing a red node can be done without causing a violation. We will generalize by considering the node we are trying to remove as being in violation (it is currently too black to remove).

If the node has two children, we will replace it with its successor. The successor should adopt the node's current color to avoid causing a violation. We don't have to worry about this node, it is the successor that will ultimately be the node that is actually removed from the tree. But the successor will have at most one child, so we can ignore this case.

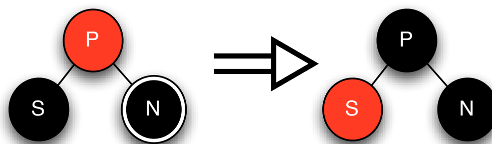
If the node has one child, it must be red (otherwise there would be a violation of the black height property). So give the black to the child before removing the node.



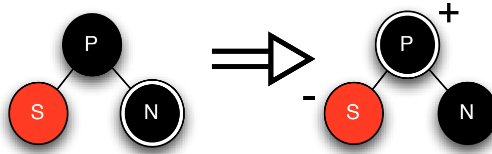
If the node has no children, then things get more complicated and we have to push the black up the tree into the parent. We have three basic cases (six if you count mirror images): parent is red and sibling is black, parent is black and sibling is red, and parent and sibling are both black.

In these cases, we will remove the violation from the node rather than marking it red. If it is the node we need to remove, it will be removed next, so we don't really need to change it. Otherwise it was a node that was "extra" black, so removing the violation just makes it plain black.

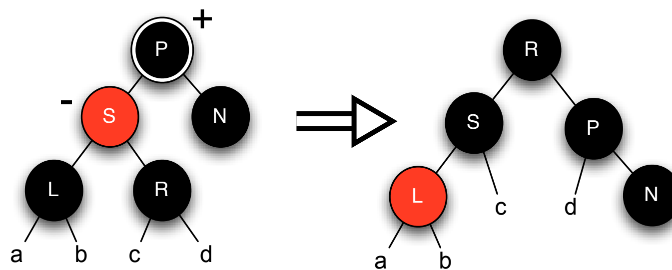
Case one: parent red / sibling black
This is a simple one. Remove a black from both children and put it in the parent. This creates a potential red violation if S has a red child.



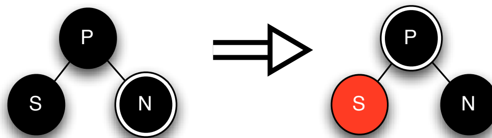
Case two: parent black / sibling red
When we perform the same operation, bubble the black up a level, we create a very broken situation. The parent is now carrying an extra black, and the sibling is now extra red.



To fix this, we observe that S must have two black children, and we can rearrange them to rebalance this subtree. This is two rotations of the sibling's inner child and some recoloring. Of course, this potentially causes a red violation with the outer child's children that should be fixed.



Case three: black parent / black sibling
In this instance, the sibling becomes red and the parent becomes extra black.



There are two ways to fix this based on the state of the sibling.

If the sibling causes a red violation, we can fix the red violation using the technique listed above. This will have the effect of pushing the parent with the "extra blackness" down one level, putting a red node above it (look at the effect of fixing red violations). We can now pass the extra black up one level, making the new root of this little subtree black.

If the sibling does not create a red violation, remove the black from the parent by going through this process again with it as the new violation node (you can do this via a recursive method or a loop). This can continue until we reach the root, from which we can remove the extra black without creating any violations in the tree.