Red Black Trees

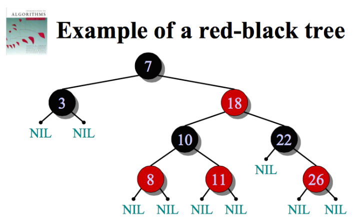
(CLRS 13)

A *Red-Black tree* is a binary search tree where each node is colored either RED or BLACK such that the following invariants are satisfied:

1. The root is BLACK.
2. A RED node can only have BLACK children.
3. Every path from a root down to a "leaf" contains the same number of BLACK nodes. Here a "leaf" means a node with less than two children.

Note that if invariant (3) holds for the root, then it must also hold for any node x in the tree.

To see invariant (3), it may be easier to conceptualize the tree such that the nodes with less than two children are linked to NIL leaves, see Figure below. Thus any path from the root to a NIL leaf has to have the same number of BLACK nodes.



The RB-tree invariants guarantee that the height of a RB-tree is $\Theta(\lg n)$.

Theorem: A red-black tree with n elements has height $\Theta(\lg n)$.

Proof: All paths from root to a leaf must have teh same number of black nodes, but we can have red nodes interleaved between black nodes. This means that the the longest and shortest path from the root to a leaf are such that $h_{max} \le 2h_{min}$. Then using the fact that a complete binary tree of height h has $2^{h+1}-1$ nodes, we get that

$$2^{h_{min}+1} - 1 \le n \le 2^{h_{max}+1} - 1$$

and from here.... we get that $h_{max} = \Theta(\lg n)$

Insertion in Red-Black Trees

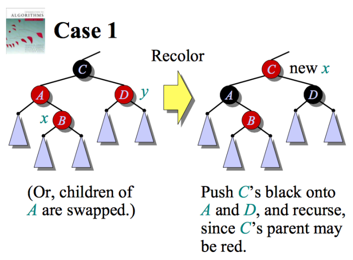
An insertion in a Red-Black tree is the same as insertion in a binary search tree, except that at the end the new node must be given a color. The question is what color.

INSERT(T, x): insert x in tree. We can't make it BLACK (violates inv.3); so we color x RED. Only invariant (2) might be violated locally, between x and its parent. Try to fix the violation by recoloring, or move the violation up the tree, until it can be fixed.

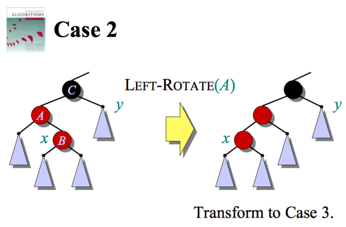
Let x be the current node (initially this is the leaf we just inserted); it is RED; it's children, if they exist, are BLACK.

- If x is the root. Make it BLACK. Done.

- If x's parent is BLACK, then invariant (2) is maintained. We are done.
- CASE 1: x's parent is RED, and its uncle exists and is RED: Recolor the parent and uncles to BLACK, recolor the grandparent to RED. Now the grandparent becomes the new x. Repeat.

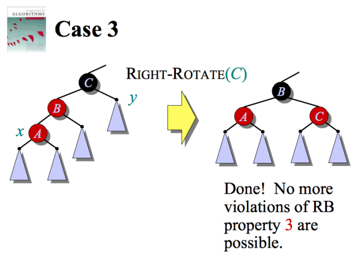

Case 1
Recolor
C
A
D
y
x
B
new x
(Or, children of A are swapped.)
Push C's black onto A and D, and recurse, since C's parent may be red.

- CASE 2: x's parent is RED, it's uncle is BLACK or does not exist, and x is a right child, and it parent is a left child: Rotate-left at x, and transform to case 3.



  Note: The symmetrical case where x is a left child, and it parent is a right child is handled symmetrically: Rotate-right at x, and transform to case 3.

- CASE 3: x's parent is RED, it's uncle is BLACK or does not exist, and x is a left child, and its parent is a left child: rotate-right at grandparent of x. Done.



Note: The case where x is a right child and its parent is a right child is handled symmetrically: rotate-left at grand-parent of x. Done.

Remarks: Each case either resolves the problem (via $O(1)$ rotations and recolorings) or propagates it up in the tree. Thus an insertion can be performed in $O(h) = O(\lg n)$ time.

Deletion in Red-Black Trees

Handling deletion is similar, and can be performed in $O(h) = O(\lg n)$ time.