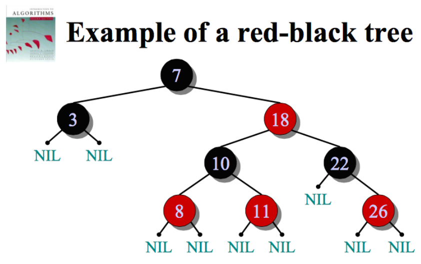
Red Black Trees

(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.

To understand 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 invariant (3) becomes: any path from the root to a NIL leaf has to have the same number of BLACK nodes.



Note that if invariant (3) holds for the root, then it must also hold for any node x in the tree (that is, the number of black nodes from an arbitrary node in a RB-tree to all "leaves" in its subtree is the same).

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 the 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 $h_{max} \leq 2h_{min}$. Then using the fact that a complete binary tree of height h has $2^{h+1} - 1$ nodes, we get that

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

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

Insertion in Red-Black Trees

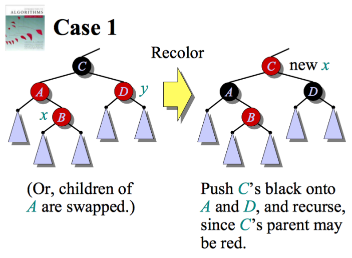
INSERT(T, k):

An insertion in a Red-Black tree is the same as insertion in a binary search tree, searching for k starting from the root, until reaching NULL and inserting a node with key k as a leaf node; at the end the new node must be given a color. The question is what color: We can't make it BLACK (bacause it violates inv. 3), so we color it RED. This may violate invariant (2) between the node and its parent. We'll 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 (initially x is a leaf so it has no children so this is true).

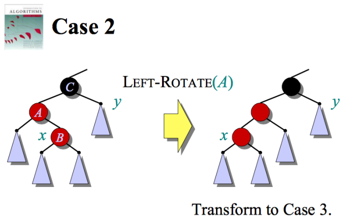
- If x is the root: actually, in this case, we can make it BLACK. The number of black nodes on all paths increases by one, but they all stay equal. We are done.
- If x's parent is BLACK, then invariant (2) is maintained. We are done.
- If x's parent is RED:

  1. 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.



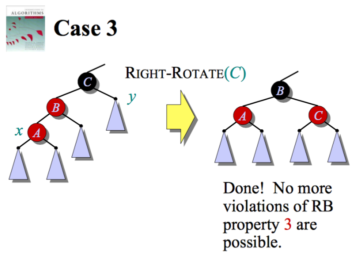
Case 1
Recolor
C
A
D
y
x
B
new x
(Or, children of A are swapped.)
Push C's black onto A and D, and recurse, since C's parent may be red.

  2. CASE 2: x's parent is RED, it's uncle is BLACK or does not exist, and x is a right child, and its 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.

  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 moves it up in the tree. In the worst case insetion will hit case 1 every time and will move the problem node x up the tree until x will be the root; then it will color it BLACK. 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.