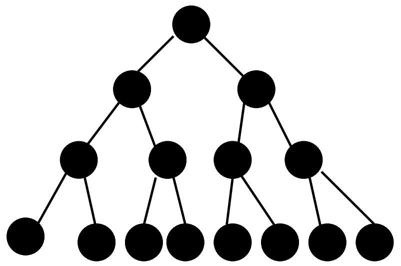
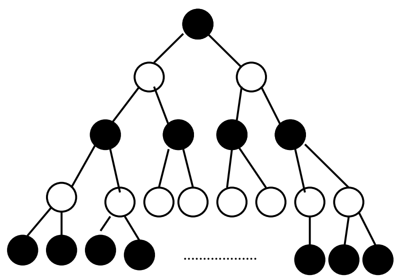
14.1-5 Describe a Red-Black tree with the largest and smallest ratio of red nodes.

To minimize the ratio of red-black nodes, make all black (possible for $n = 2^k - 1$)



To maximize the ratio of red nodes, interleave with red nodes as *real* leaves



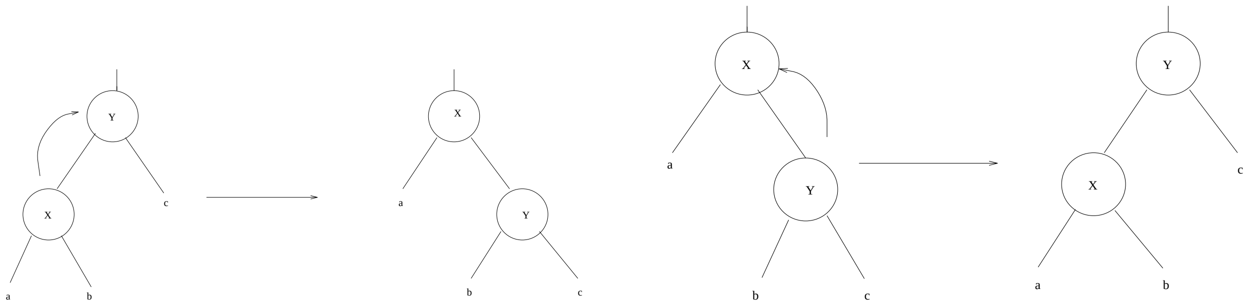
$$\#red/n \leq n/2 + n/8 + n/32 = .656n$$

$$\#black/n \leq n/4 + n/16 + n/64 = .328n$$

$$n - n\sum_{i=1}^{max}(1/4)^i = 2n/3$$

Rotations

The basic restructuring step for binary search trees are left and right rotation:



1. Rotation is a local operation changing $O(1)$ pointers.

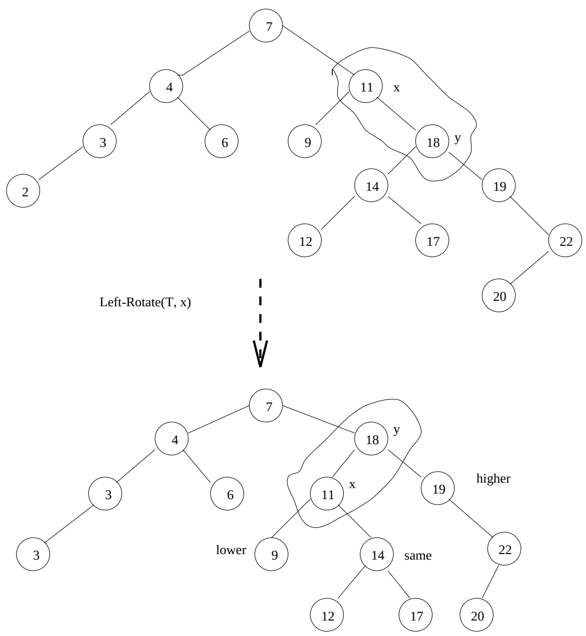
2. An in-order search tree before a rotation *stays* an in-order search tree.

3. In a rotation, one subtree gets one level closer to the root and one subtree one level further from the root.

LEFT-ROTATE(T,x)

$y \leftarrow right[x]$ (* Set y*)
$right[x] \leftarrow left[y]$ (* Turn y's left into x's right*)
if $left[y] = \text{NIL}$
  then $p[left[y]] \leftarrow x$
$p[y] \leftarrow p[x]$ (* Link x's parent to y *)
if $p[x] = \text{NIL}$
  then $root[T] \leftarrow y$
  else if $x = left[p[x]]$
    then $left[p[x]] \leftarrow y$
    else $right[p[x]] \leftarrow y$
$left[y] \leftarrow x$
$p[x] \leftarrow y$

Note the in-order property is preserved.



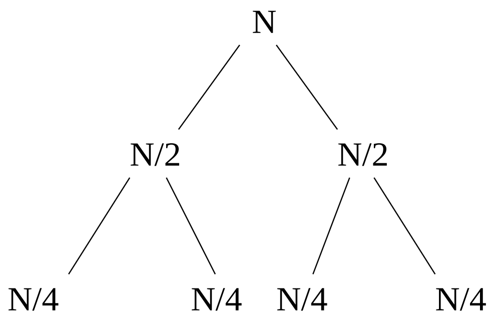
14.2-5 Show that any n-node tree can be transformed to any other using $O(n)$ rotations (hint: convert to a right going chain).

I will start by showing weaker bounds - that $O(n^2)$ and $O(n \log n)$ rotations suffice - because that is how I proceeded when I first saw the problem.

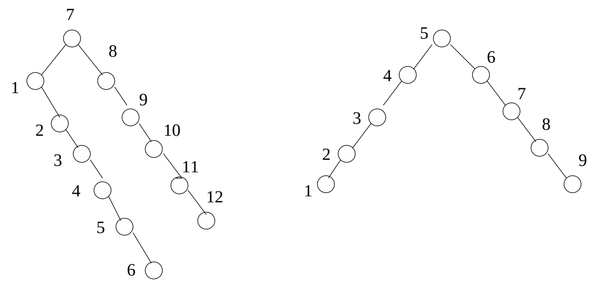
First, observe that creating a right-going, for t_2 path from t_1 ¡ and reversing the same construction gives a path from t_1 to t_2.

Note that it will take at most n rotations to make the lowest valued key the root. Once it is root, all keys are to the right of it, so no more rotations need go through it to create a right-going chain. Repeating with the second lowest key, third, etc. gives that $O(n^2)$ rotations suffice.

Now that if we try to create a completely balanced tree instead. To get the $n/2$ key to the root takes at most n rotations. Now each subtree has half the nodes and we can recur...

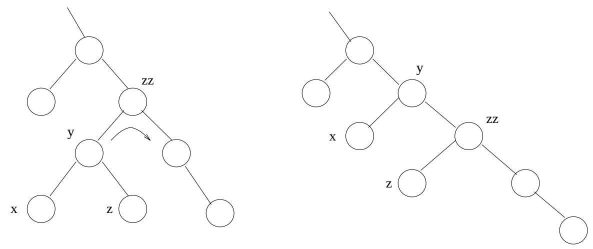


To get a linear algorithm, we must beware of trees like:



The correct answer is that $n - 1$ rotations suffice to get to a rightmost chain.

By picking the lowest node on the rightmost chain which has a left ancestor, we can add one node *per* rotation to the right most chain!



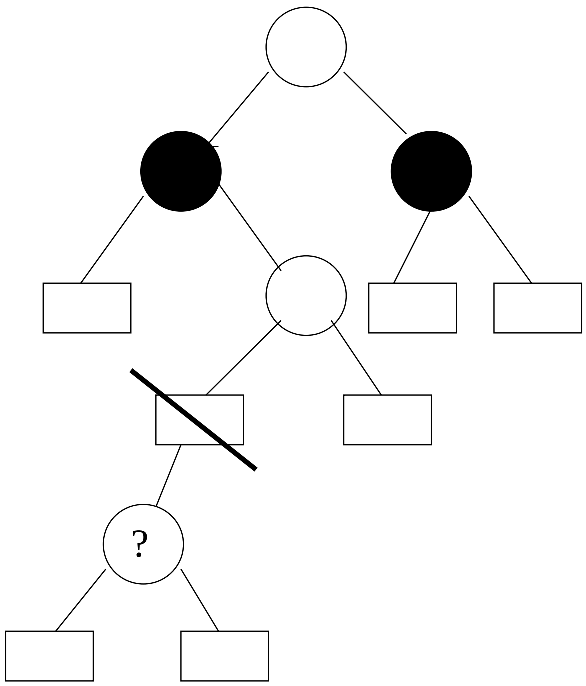
Initially, the rightmost chain contained at least 1 node, so after *1* rotations it contains all *n*. Slick!

Red-Black Insertion

Since red-black trees have $\Theta(\lg n)$ height, if we can preserve all properties of such trees under insertion/deletion, we have a balanced tree!

Suppose we just did a regular insertion. Under what conditions does it stay a red-black tree?

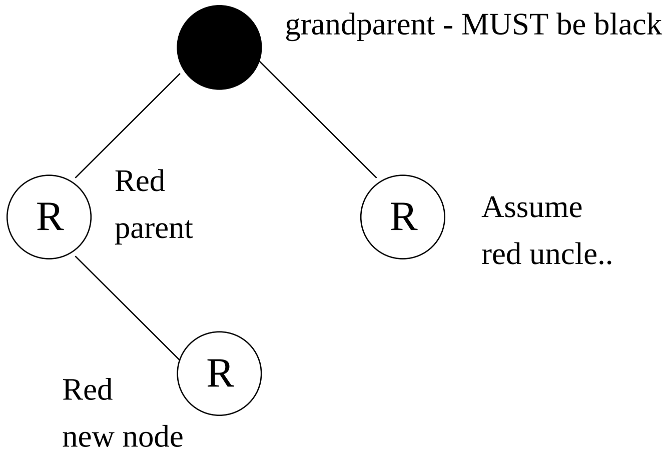
Since every insertion take places at a leaf, we will change a black NIL pointer to a node with two black NIL pointers.



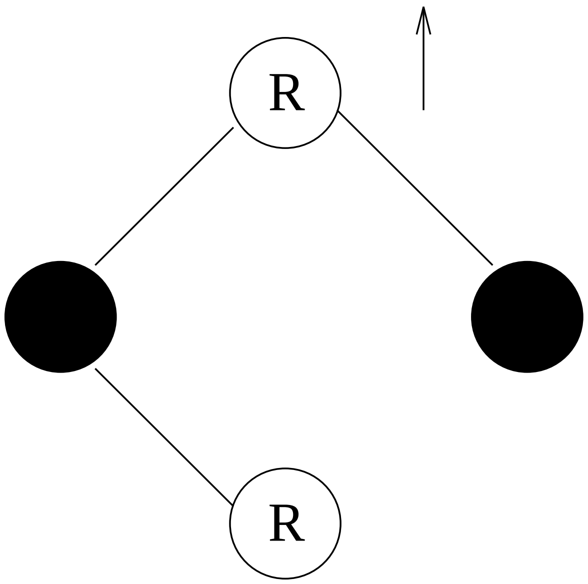
To preserve the black height of the tree, the new node must be *red*. If its *new* parent is black, we can stop, otherwise we must restructure!

How can we fix two reds in a row?

It depends upon our uncle's color:



If our uncle is red, reversing our relatives' color either solves the problem or pushes it higher!



Note that after the recoloring:

1. The black height is unchanged.

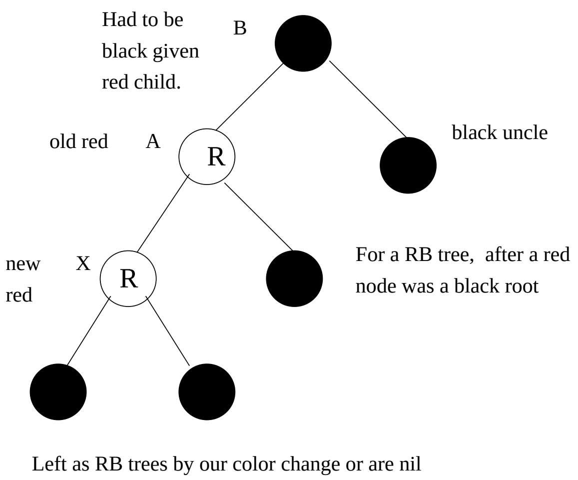
2. The shape of the tree is unchanged.

3. We are done if our great-grandparent is black.

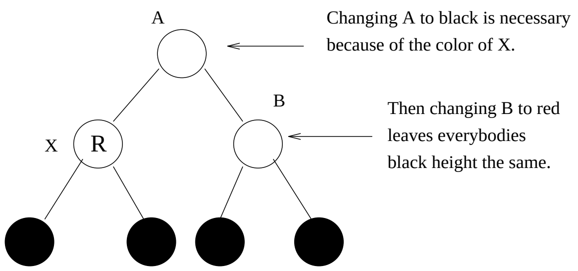
If we get all the way to the root, recall we can always color a red-black tree's root black. We always will, so initially it *was* black, and so this process terminates.

The Case of the Black Uncle

If our uncle was black, observe that all the nodes around us have to be black:



Solution - rotate right about B:



Since the root of the subtree is now black with the same black-height as before, we have restored the colors and can stop!

A double rotation can be required to set things up depending upon the left-right turn sequence, but the principle is the same.

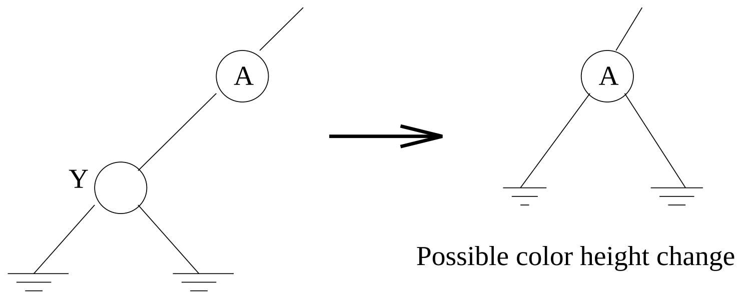
DOUBLE ROTATION ILLUSTRATION

Pseudocode and Figures

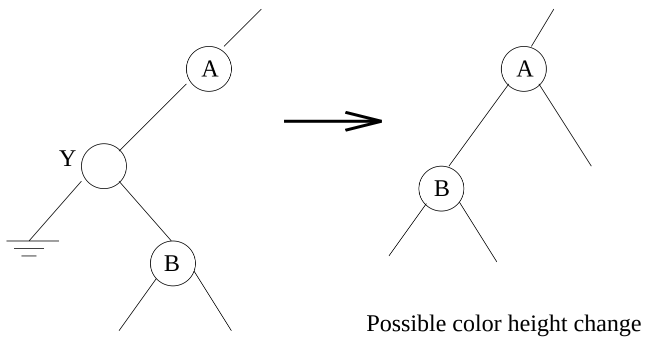
Deletion from Red-Black Trees

Recall the three cases for deletion from a binary tree:

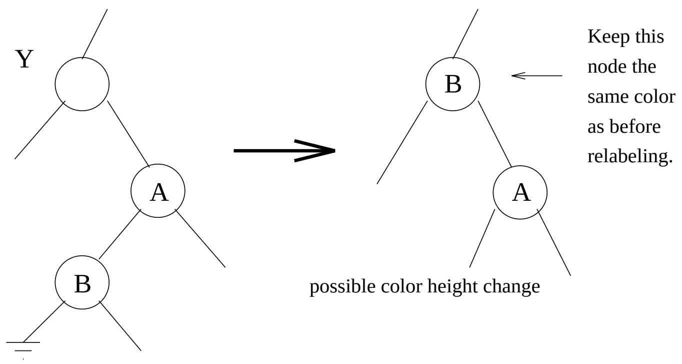
Case (a) The node to be deleted was a leaf;



Case (b) The node to be deleted had one child;



Case (c) relabel to node as its successor and delete the successor.



Deletion Color Cases

Suppose the node we remove was *red*, do we still have a red-black tree?

Yes! No two reds will be together, and the black height for each leaf stays the same.

However, if the dead node y was black, we must give each of its decendants another black ancestor. If an appropriate node is red, we can simply color it black otherwise we must restructure.

Case (a) black NIL becomes "double black";

Case (b) red β becomes black and black β becomes "double black";

Case (c) red β becomes black and black β becomes "double black".

Our goal will be to recolor and restructure the tree so as to get rid of the "double black" node.

In setting up any case analysis, we must be sure that:

1. All possible cases are covered.

2. No case is covered twice.

In the case analysis for red-black trees, the breakdown is:

Case 1: The double black node x has a red brother.

Case 2: x has a black brother and two black nephews.

Case 3: x has a black brother, and its left nephew is red and its right nephew is black.

Case 4: x has a black brother, and its right nephew is red (left nephew can be any color).

Conclusion

Red-Black trees let us implement all dictionary operations in $O(\log n)$. Further, in no case are more than 3 rotations done to rebalance. Certain very advanced data structures have data stored at nodes which requires a lot of work to adjust after a rotation — red-black trees ensure it won't happen often.

Example: Each node represents the endpoint of a line, and is augmented with a list of segments in its subtree which it intersects.

We will not study such complicated structures, however.