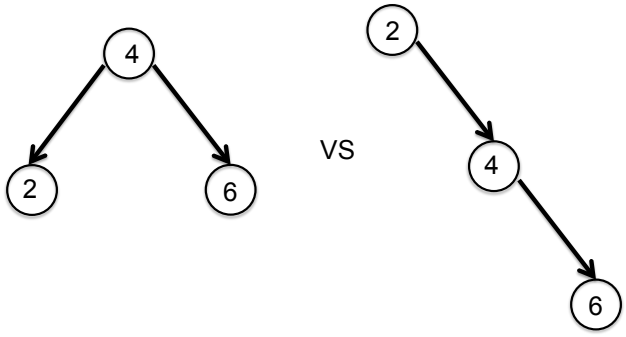
Red Black tree

Summary from last time: General Binary search tree. Many possible trees for one set of keys. Run time: O(h) with h the height of the given binary tree. If tree is evenly balanced this could be height log n, but if the tree each time just spreads in one direction (eg, always right) then operations are as slow as a linked list O(n).



We'd like: Tree that is more balanced; search; min; max; predecessor; successor, in time **O(log n)** with n number of nodes in the tree.

Red Black Tree: special case of a Binary Search Tree in which the tree is approximately balanced, and thus has the good O(log n) run time.

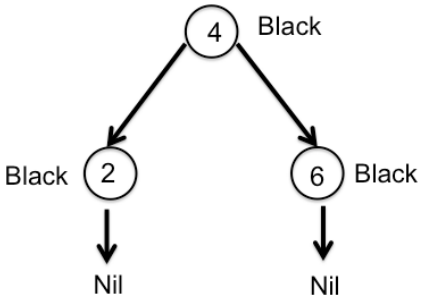
Data structure fields: As in a Binary Search Tree, but has extra attribute color: x.color, either red or black.

Properties of a red-black tree:

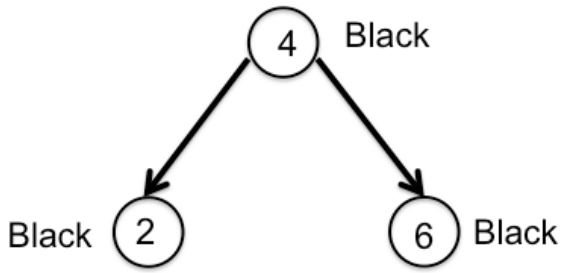
1. Every node is either red or black
2. The root is black
3. Every leaf is black and nil (often omitted in drawings)
4. **If a node is red, then both its children are black (no two reds in a row)**
5. **For each node: all paths from node to (nil) leaves contain same number of black nodes. Think of as all unsuccessful searches from the given node have same number of black nodes.**

(bolded are the main properties to check each time when dynamically changing a tree; other properties more simple)

Example Red-Black tree:



We usually draw without the nil:

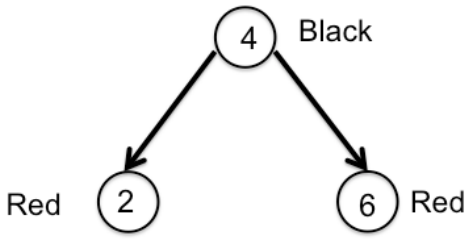


Satisfies all properties:

- root is black..
- no two reds in a row
- For every node, every path from the node to nil contains same number of black nodes (convention is not to count root node itself but to count the nil leaves, as in making an unsuccessful search). From root: every path has 2 black nodes; from 2 and 6 every path just goes to nil so 1 black node.

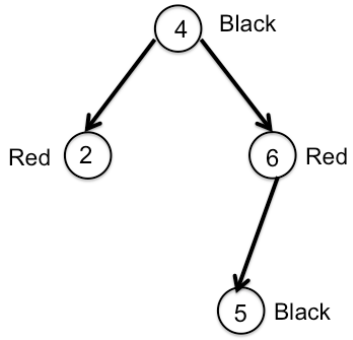
Unique Red-Black tree for these nodes?

No. Also:

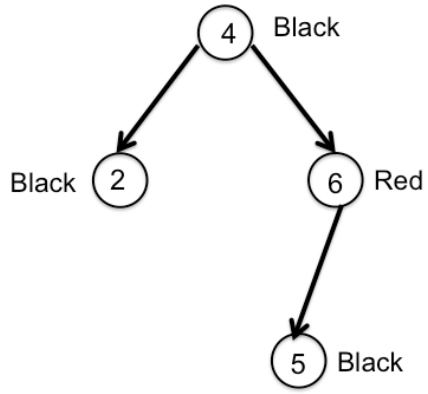


(convince yourself again that all properties hold)

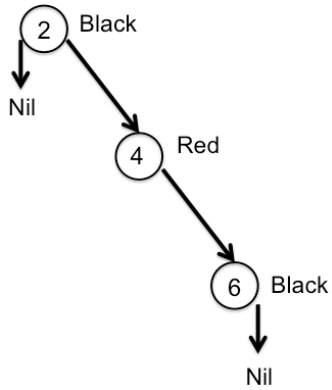
Adding another node: need to maintain Red-Black properties. This won't work (black nodes from root to leaf different; also can't place as Red since then two Reds in a row):



Adding another node: this will work with some reorganization (now number Black from root to node equals 2; more on re-organization later):



Can we make a chain / linked list that is a Red Black Tree for these nodes? **No.** Schematic below violates path from root to nil has same number black nodes (1 versus 2)



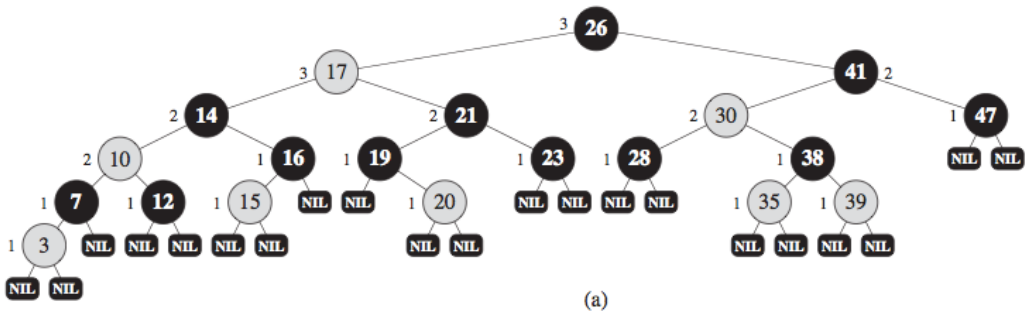
Node 6 also can't be red, because then have two reds in a row, not allowed.

If node 4 is Black and node 6 is Red, again every path from root to nil does not have same number of black nodes.

Conclusion from examples: So we see intuitively that a linked list, whose height is n, violates the Red Black tree properties. This is good, since we said we would like to maintain things relatively balanced and a height O(log n).

Definition: BH = Black Height of a node x: number nodes that are black from x down to leaves (not counting the node x itself). This is the same number of black nodes we have been counting in property number 5 of Red Black trees.

Example Red Black tree and BH marked:



Definition: Height of tree: number internal nodes from root down to leaves.

Lemma: A Red-Black tree with n nodes has height <= 2 log(n+1) (internal nodes; excluding the nil leaves) = nice height property that we would like for binary tree

Proof:

a. **Subtree with root node x contains at least $2^{BH(x)}-1$ (internal) nodes**

By induction:

Base case: If height of x is 0, then x is a leaf node (nil), BH(x)=0, and tree rooted at x indeed contains $2^0-1=1-1=0$ nodes.

Inductive step: Consider a node x that has positive height and has two children. Each child of x is either Red or Black, so has Black Height of either BH(x) or BH(x)-1. Since the height of each child is less than the height of x itself, we can use the inductive hypothesis that each of x's children contains at least $2^{BH(x)-1}-1$ internal nodes. Then the subtree from node x will contain at least $(2^{BH(x)-1}-1)+(2^{BH(x)-1}-1)+1=2^{BH(x)}-1$ nodes, which is what we wanted to prove inductively.

b. **We'll now use a property of Red-Black trees to complete the lemma proof.** Let h be the height of the tree from its root. According to property 4 (no two reds in a row), at least half the nodes from root to leaf are black, so the BH must be at least $h/2$.

So from part (a) of the proof we have: $n \geq 2^{BH(x)} - 1 \geq 2^{h/2} - 1$

Therefore: $n \geq 2^{h/2} - 1$

Moving the 1 to the right and taking log on both sides:
$\log(n+1) \geq h/2$

Therefore: $h \leq 2\log(n+1)$

Interpretation: Red-Black tree height is O(log n) and so we can do operations that depend on its height (search; min; max; predecessor; successor) in O(log n) time

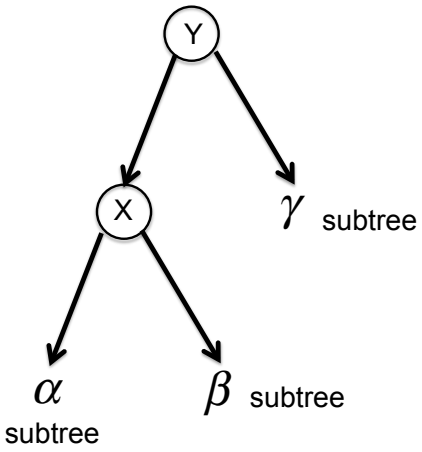
Insert (and delete) in a Red Black tree

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

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. Not particular for Red Black, although we will use this in Red-Black examples of insert later)

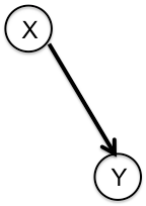
Schematic:



In this tree: keys in X.key < Y.key

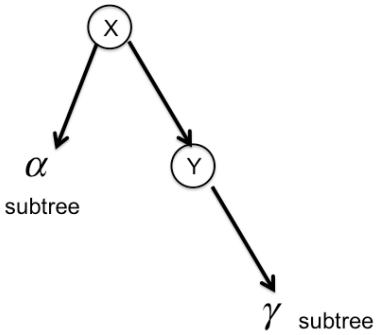
We also have: alpha < X.key < beta < Y.key < gamma

We want to exchange the parent y and child x:

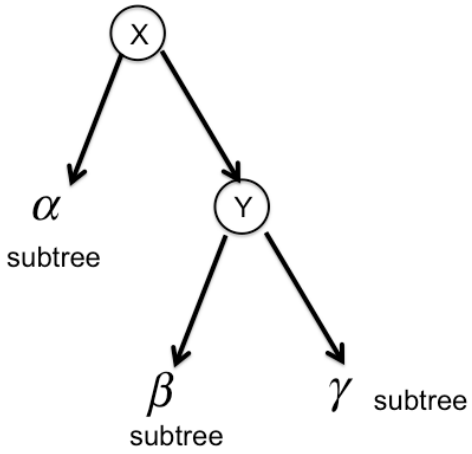


We changed the direction (rotate right) to maintain: X.key < Y.key

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



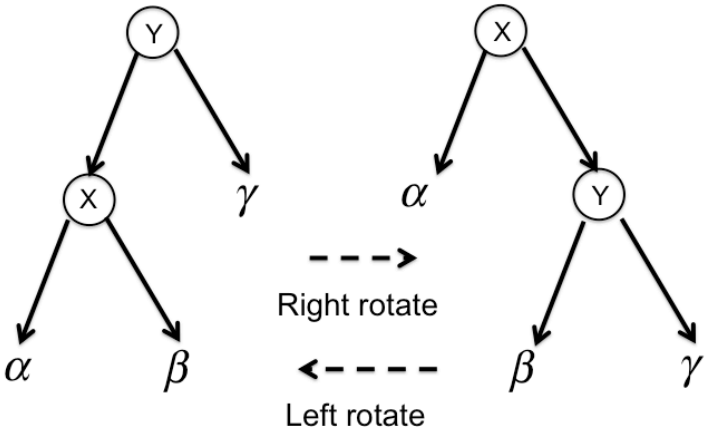
X's left and Y's right children stay the same



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

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

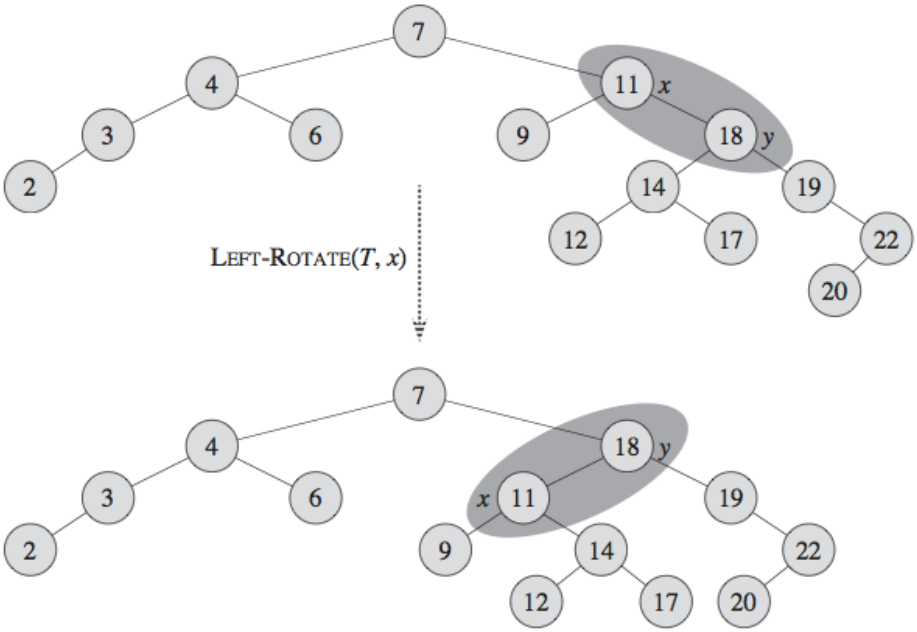
Overall like a rotation:



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: O(logn) time