**BST with AVL**

A unbalanced, or nearly linear, BST executes adding, searching, and removing in O(n) time. A self-balancing BST, depending on its implementation, can execute adding, searching, and removing in O(n log n) or even in O(logn) time. Some common self-balancing BST's include AVL trees, red-black trees, and AA trees.

**AVL** trees are always optimally balanced, and therefore slower. **Red-black** trees are less balanced, but faster. An **AA** tree is a red-black tree with restrictions that make the AA tree much easier to maintain.

We will implement the AVL algorithm in addBalanced. Notice that the code below adds the value, then balances the tree. This version runs in O(n logn) time.

public void addBalanced(String value) //new method  
{  
 root = add(value); // add O(log n)  
 root = balanceTree( root ); // then balance O(n log n)   
}

The method balanceTree is recursive, of course, recurring to the right and then to left. The actual balancing occurs from the bottom up, as the recursion unwinds. Watch this visualization: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

We also need:

private int calcBalance(TreeNode t) //height to right minus   
 //height to left

private int height(TreeNode t) //from TreeLab

and 4 “rotation” methods, like this:

private TreeNode rightRight(TreeNode root) //3 5 7

Your folder has three different .pdfs explaining AVL trees. The one I found most helpful was

**AVL-Tree-Rotations.pdf** because it has some pseudocode at the end. Beware! The .pdfs sometimes use different vocabulary to describe the same situation, e.g. the RightRight case needs a Left Rotation (LL), or the RightLeft case needs a Left-Right Rotation (LR).

Lectures are available on the internet.

MIT lecture: <https://www.youtube.com/watch?v=FNeL18KsWPc>

Jenny’s lecture: <https://www.youtube.com/watch?v=YWqla0UX-38>

Abdul Bari: <https://www.youtube.com/watch?v=jDM6_TnYIqE>

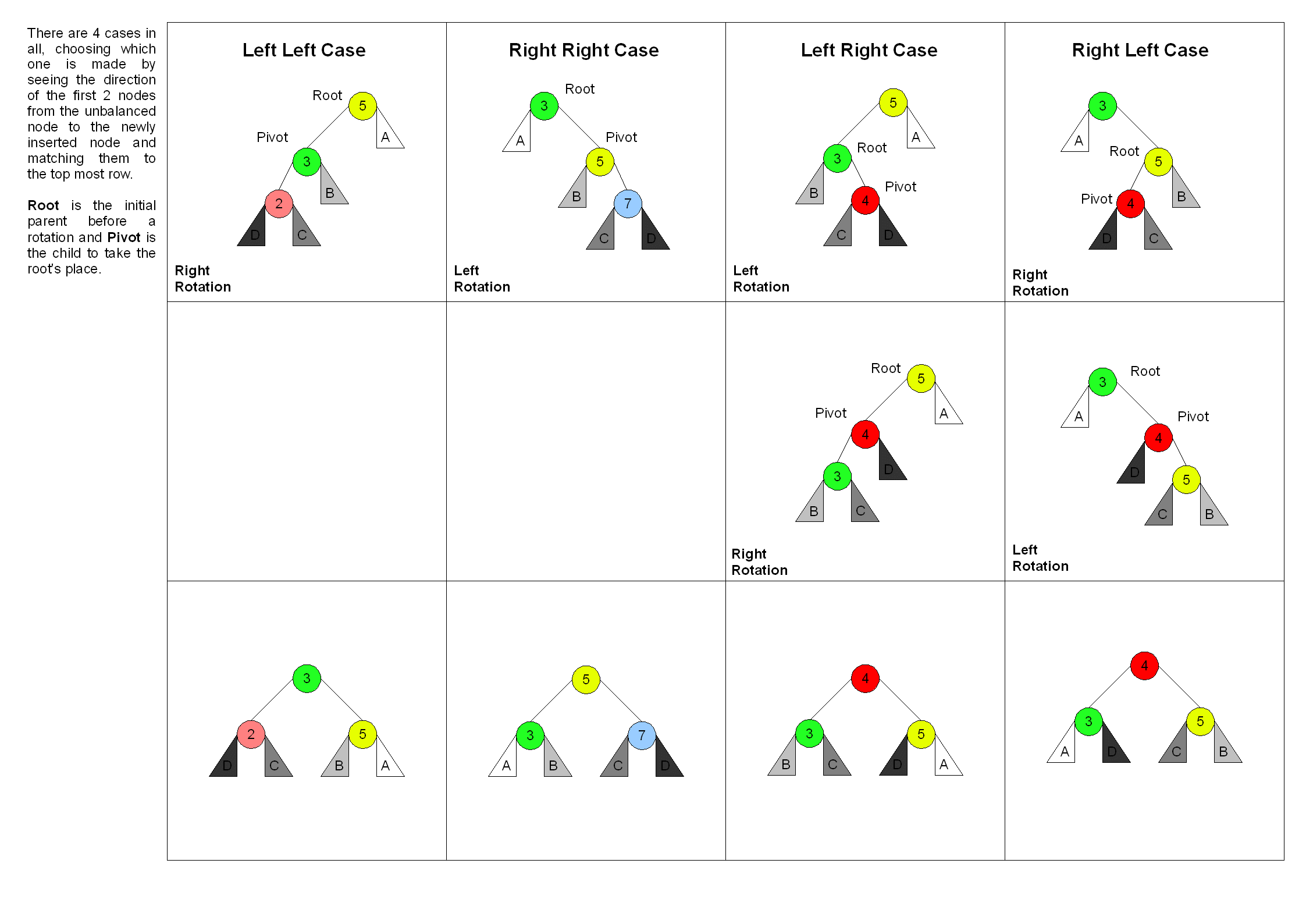
Rob Edwards:   
 <https://www.youtube.com/watch?v=7m94k2Qhg68&feature=emb_title&ab_channel=RobEdwards>

Helpful diagrams are on the next page.

2) This diagram comes from <https://en.wikipedia.org/wiki/Tree_rotation>

if(balance == -2) if(balance == 2) if(balance == -2) if(balance == 2)

“left heavy” “right heavy” “left heavy” “right heavy”



**Assignment**

Submit your BST (which can only store Strings) with the addBalanced method. The driver is called BST\_AVL\_Driver.java. The driver gives you several test cases. (Beware! The “5 3 2” is treated as three strings, not as three integers.) The first four test cases allow you to test your code using the nodes on the chart above.

//String line = "5 3 2"; //left-left case (right rotation)

//String line = "3 5 7"; //right-right case (left rotation)  
//String line = "5 3 4"; //left-right case (left rotation then   
 // right rotation)  
//String line = "3 5 4"; //right-left case (right rotation then

// left rotation

**Extensions**

1. Implement an O(log n) AVL rebalancing algorithm. One implementation adds an extra int height field to the TreeNode class to store the height at that node. After you add a new node, as the recursion unwinds, you rebalance at each node. O(2\* log n) 🡪 O(log n)
2. Another O(log n) implementation uses two parallel trees, one to store the data and the other to store the heights.
3. Comment in removeBalanced that removes a node from an AVL tree and then rebalances the tree. Since adding always occurs at a leaf, but removing could remove any node, you must find a brand-new rebalancing algorithm.
4. Write public class Red\_Black which implements a red-black tree. Look it up.