**AVL Trees**

An unbalanced, or nearly linear, BST executes adding and removing in O(n) time. The most efficient self-balancing BST’s can execute adding and removing 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.

The AVL algorithm is recursive. As the recursion unwinds, it balances the tree from the bottom up, according to the differences between height to the right and to the left of each node. To see the balancing in action, enter some values (e.g., 5 3 2) in this demo:

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

We will implement our balancing method in the new addBalanced:

public void addBalanced(String value) //new method

You may choose one of two approaches (neither is the absolutely most efficient way):

1) Call the old add method from BST, which does not balance. Then call a recursive balanceTree which recurs from the root all the way to the right and to the left. As the recursion unwinds, it visits every node, recalculating heights, from the bottom up. Because it recalculates each height by visiting each node again, the Big-O for this version is O(n2).

2) Write the addBalanced method so that it adds a node and recursively rebalances the tree starting from that node. This approach is more efficient because you are visiting only the nodes that are affected by the new node. The Big-O for this version is O(n log n).

You will need to copy a method from TreeLab: private int height(TreeNode t)

You will find it convenient to write 4 “rotation” methods, named for the 4 cases, like this:

private TreeNode leftLeft( . . . ) //5 3 2

//performs a right rotation

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

If you choose the first approach, you will also need:

private TreeNode balanceTree( TreeNode current )

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. balancing the Left-Right Case could be named, variously, a double right rotation, a right-left rotation, or RL.

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>

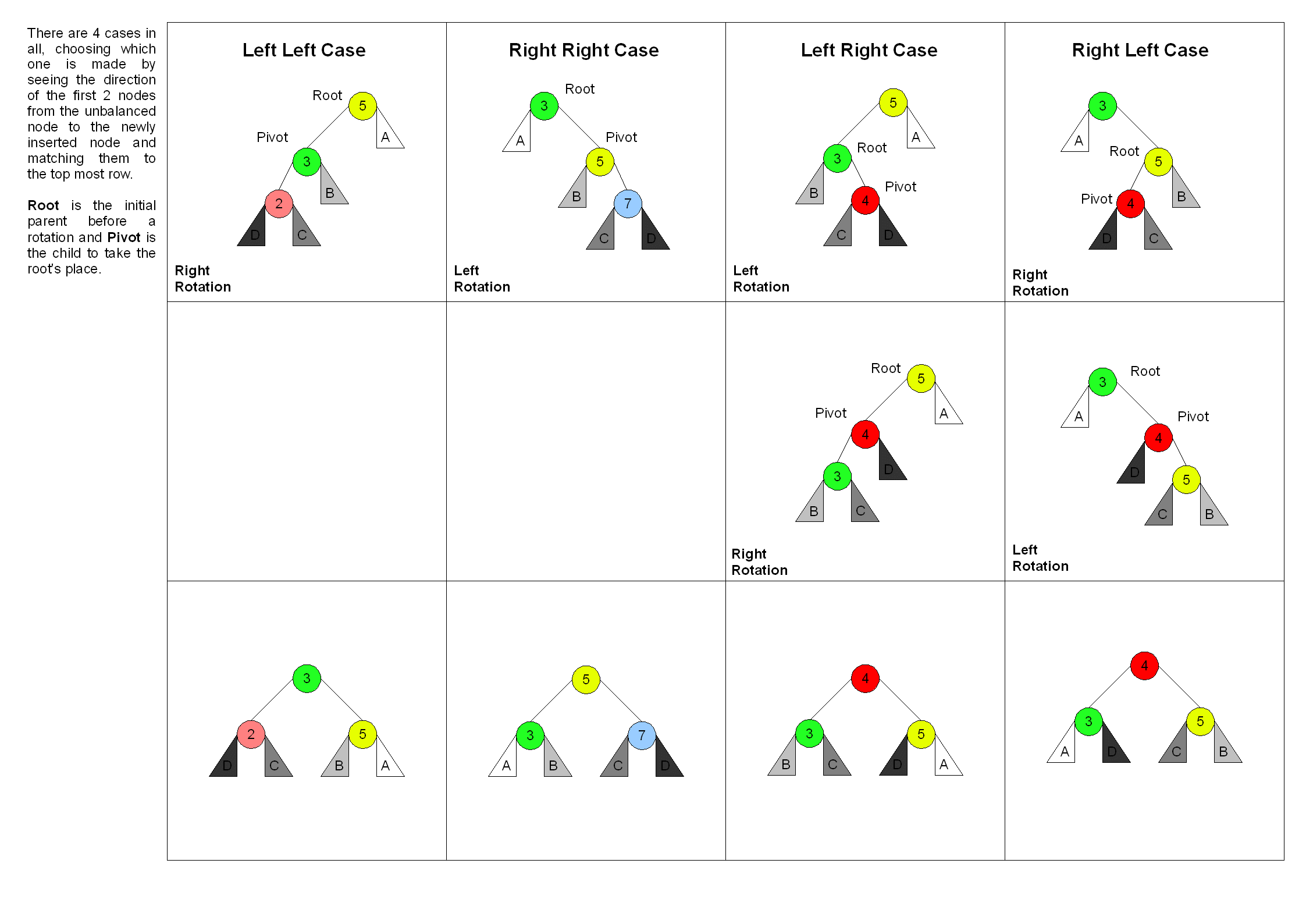
Rob Edwards:   
 <https://www.youtube.com/watch?v=7m94k2Qhg68&feature=emb_title&ab_channel=RobEdwards>

Two helpful diagrams are on the next page.

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”



Mr. Jurj prefers these diagrams:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | if(balance == -2)  “left heavy”  left-left case | if(balance == -2)  “left heavy”  left-right case | if(balance == 2)  “right heavy”  right-right case | if(balance == 2)  “right heavy”  right-left case |
| Before |  |  |  |  |
| After |  |  |  |  |

**Assignment**

Submit your BST with an implementation of the addBalanced method. The driver is called BST\_AVL\_Driver.java. The driver provides 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 diagram 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 a different, and even more efficient, O(log n) AVL rebalancing algorithm this way: add an int height field to the TreeNode class to store the height at that node. Then write a new addBalanced that both adds and rebalances (recursively) according to the heights to the right and left at each node. When you rotate, update the heights.
2. A different O(log n) implementation uses two parallel trees, one to store the data and the other to store the heights. Write the code.
3. In the BST shell, 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 develop a brand-new rebalancing algorithm.
4. Write public class Red\_Black which implements a red-black tree. Look it up. Use the driver to test your red-black tree.