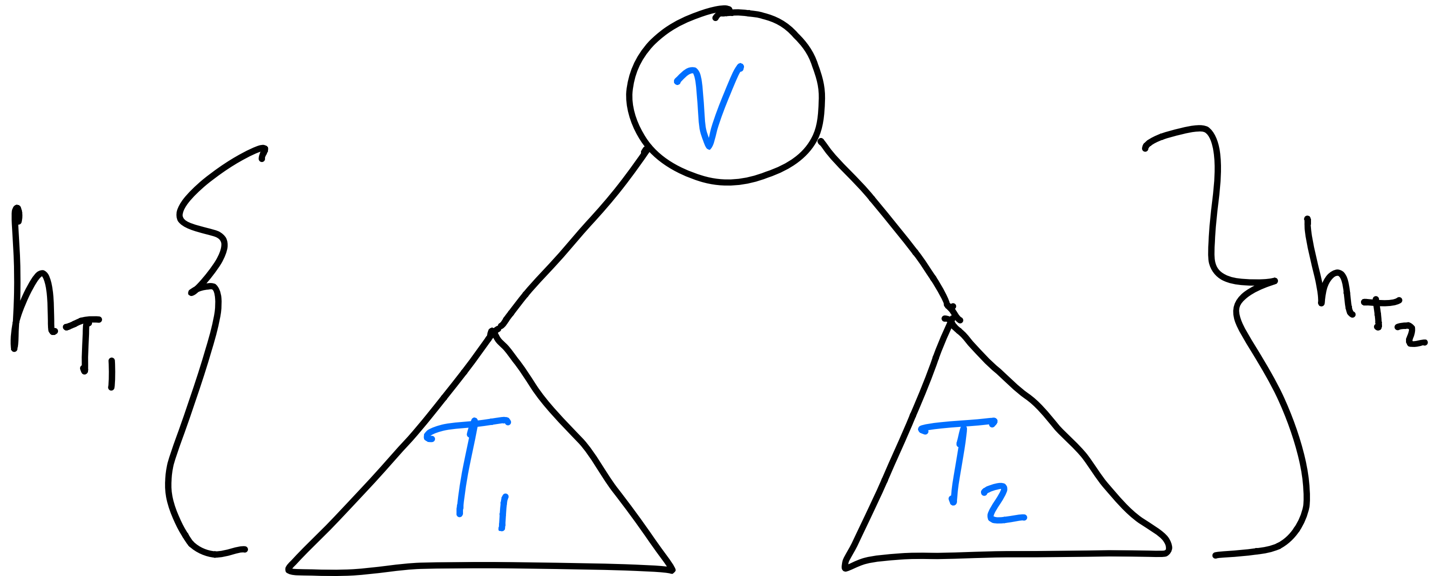
AVL Trees

An AVL (Adelson-Velskii Landis) tree is a BST with one additional rule: for every node v, the height of v's left subtree and the height of v's right subtree should differ by no more than one.



So, for all nodes in the entire tree the following must be true:

$$|h_{T_1} - h_{T_2}| \le 1$$

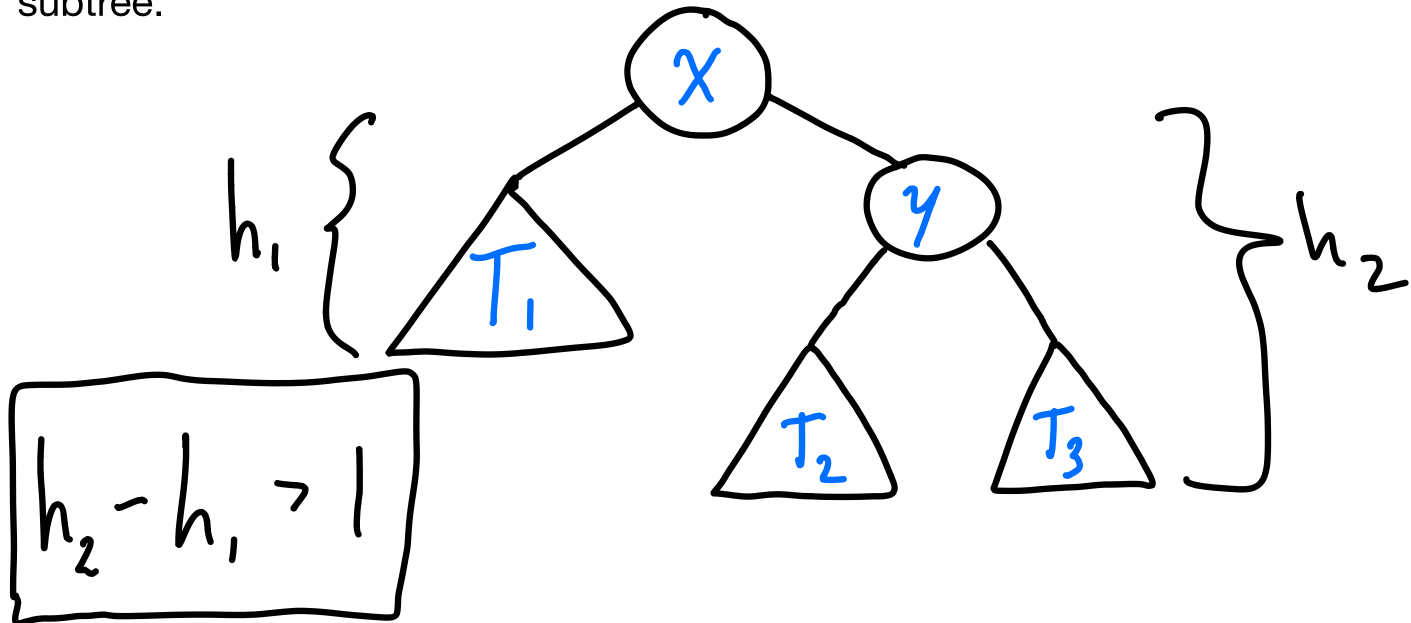
Insertion and removal of nodes in an AVL tree starts off the same as in a BST. You insert a node by traversing the tree, following the BST ordering property until you reach the appropriate leaf to add the node you are inserting. You remove a node by traversing the tree, searching for the node you intend to delete. If you find it, remove it by following one of the three cases we've previously discussed (no children, one child, or two children.)

However, it might be the case that by introducing a new node or removing an existing node you cause an imbalance at some node in the tree. Meaning, for some subtree rooted at node z, the height of z's left and right subtrees differ by more than one. When this happens, a "trinode restructuring" must take place to maintain the AVL height property.

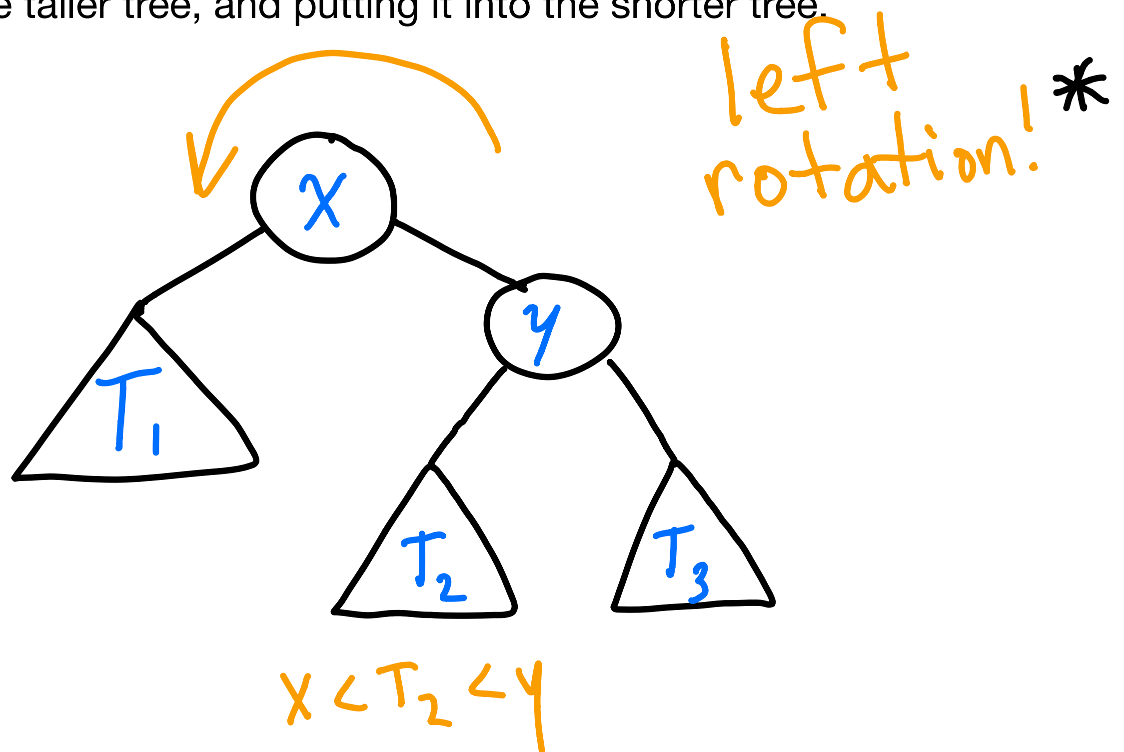
Trinode Restructuring (Rebalancing)

In order to fix the an imbalance subtree, you need an operation that can take the height of the taller subtree, and "donate" that height to the shorter subtree. The rebalance operation does exactly that. You can perform a rebalance on any node, but we will focus only on nodes that have broken the AVL height balance rule.

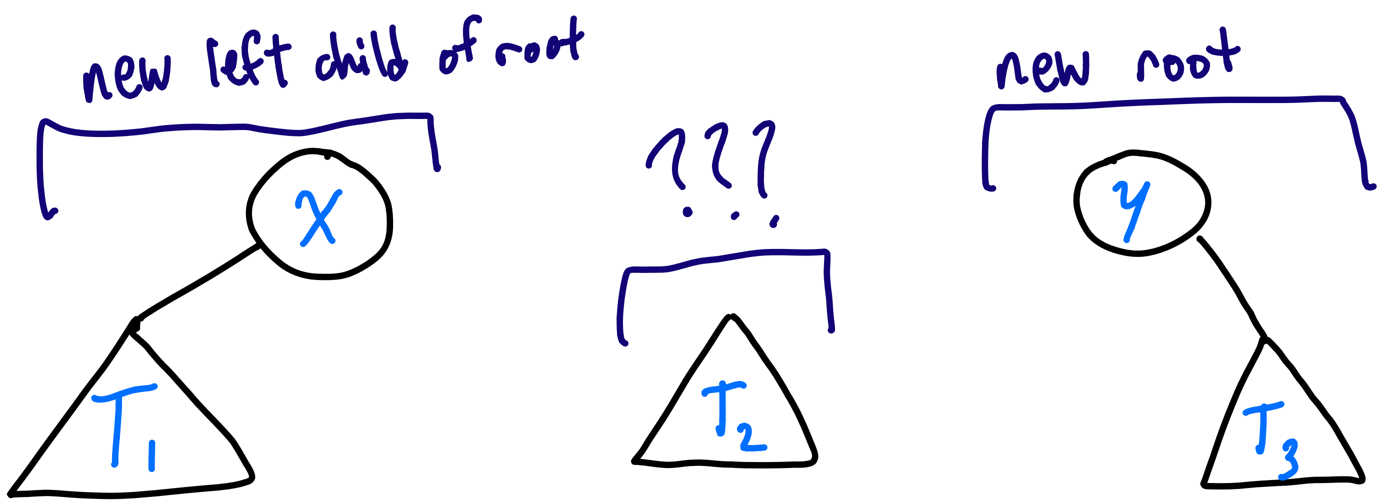
Suppose you have a subtree rooted at x, whose right subtree is taller than its left subtree.



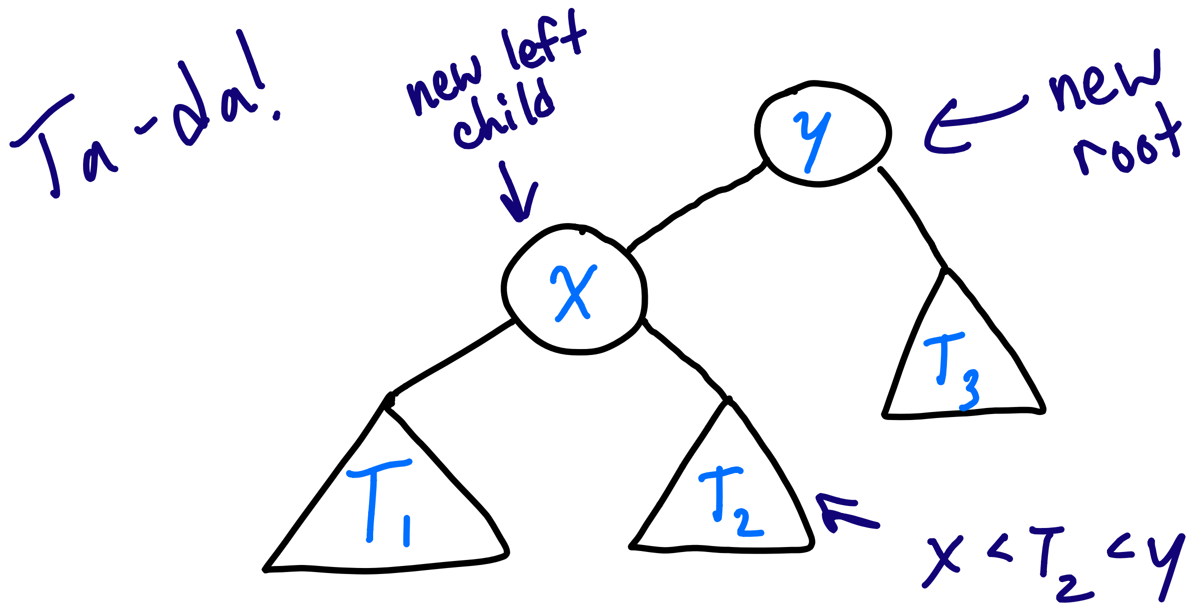
You can perform a left rotation at node x, bringing up x's right child to be the new root of this subtree, and pushing x down into the left subtree of this new root. Effectively taking height from the taller tree, and putting it into the shorter tree.



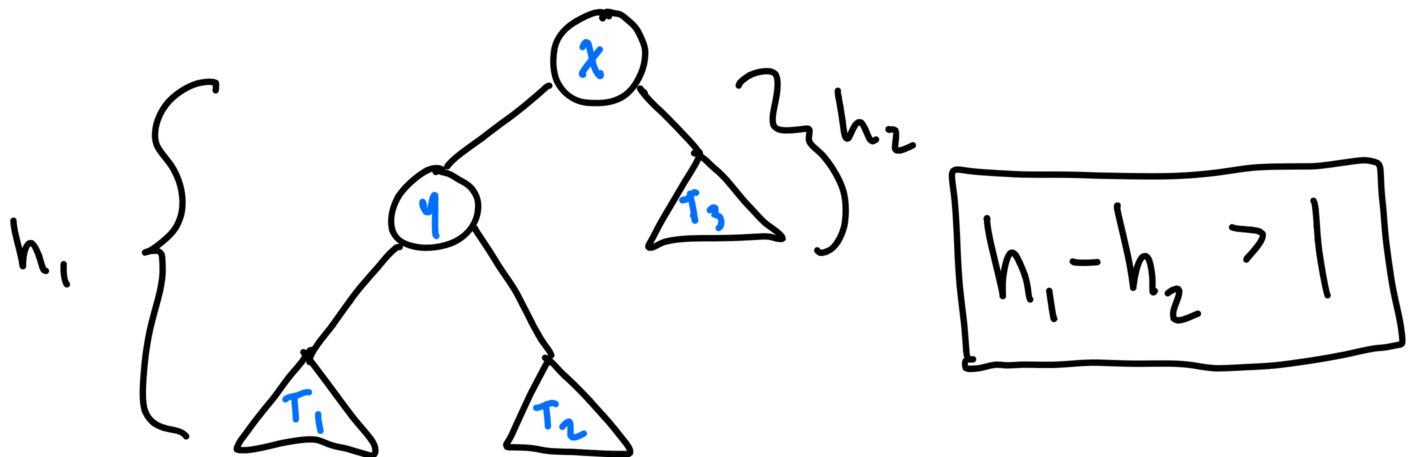
*to the tune of Beyoncé's Irreplaceable

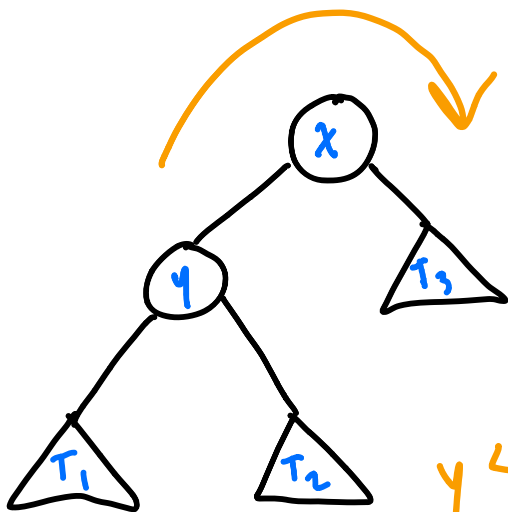


Everything smaller than x is not affected by this operation, and neither is anything larger than y. Those subtrees are always going to be in the correct place. However, the items in the middle of x and y need a new parent, since y's left child will become x. To maintain the BST ordering, x's right child must take the subtree containing all the middle elements.



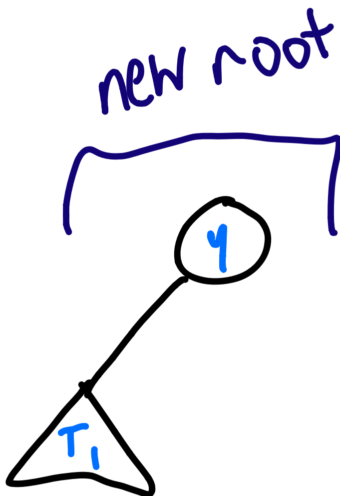
If the taller tree is on the left of a node, the rotation operation is the same idea except backwards!

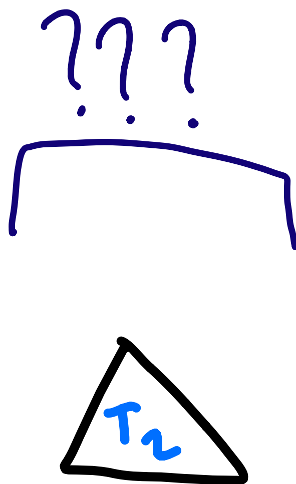




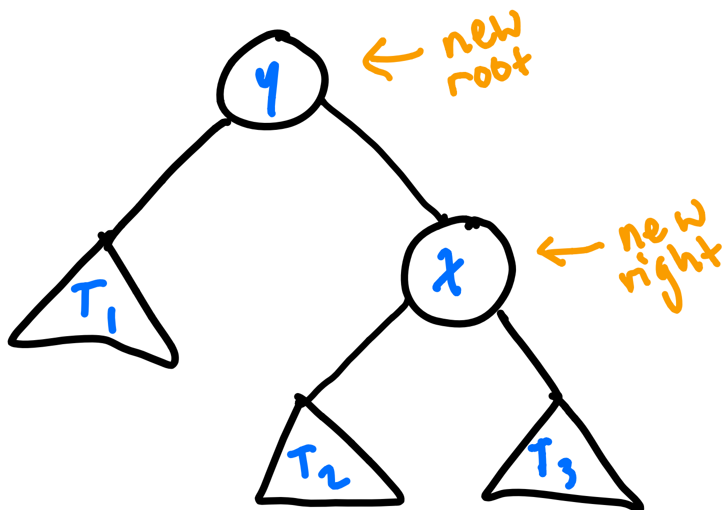
right rotation!

$y < T_2 < X$





Ta-da!



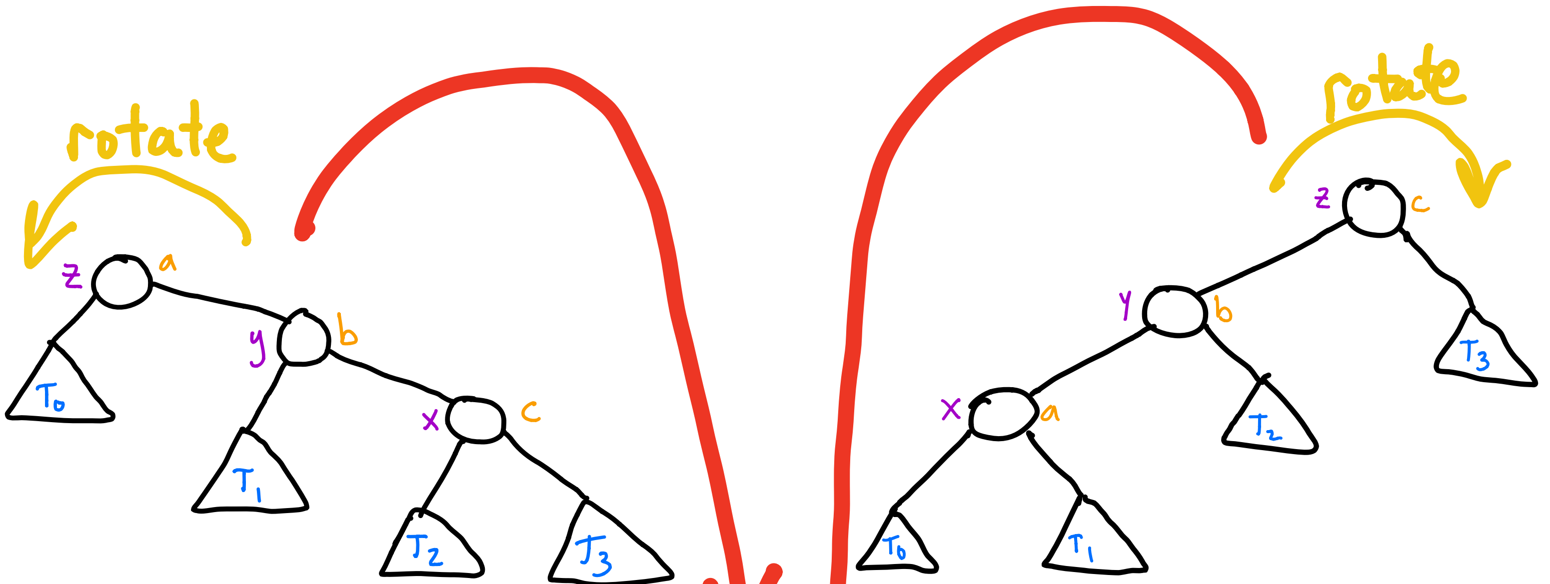
$y < T_2 < X$

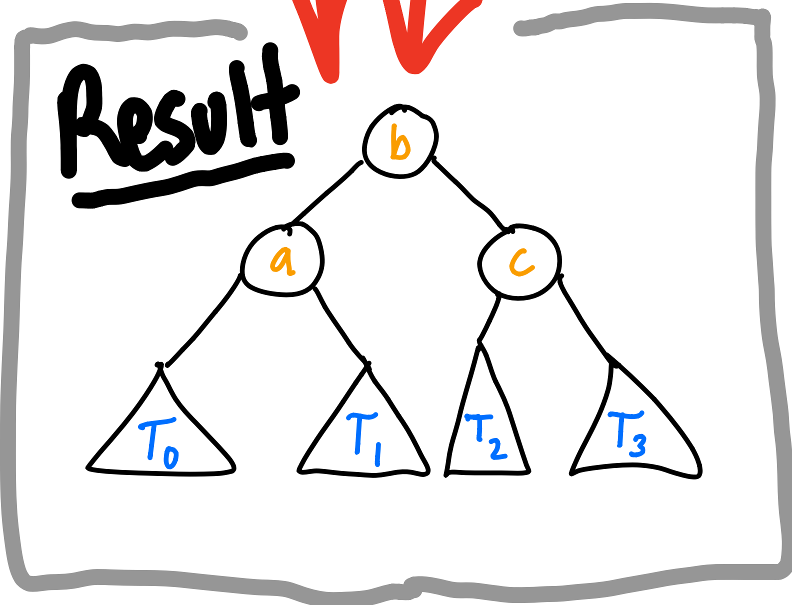
The rotation operation is O(1), because, regardless of the direction of the rotation, the operation itself is nothing more than changing a few pointers in a specific order.

To perform the restructure, you must first find the node traverse up the tree from the point you inserted/deleted. At each node, inspect the heights of the left and right subtrees.

1. Label the first violating node z.
2. Label the child of z with the larger height as node y.
3. Label the child of y with the larger height node x.
4. These nodes should then be labeled a, b, c corresponding to how each of these nodes would appear in an in order walk of the tree.
5. Rebalance at the appropriate node using the correct rotation. the could be a single or a double rotation. You will encounter four cases when a restructuring will take place. See the next page for how each of the four cases work
6. If you are restructuring as a result of a deletion, continue rebalancing up the tree until you hit the root.

Four Restructuring Cases





after rotation, this case will look similar the tree above