Deletion from an AVL Tree

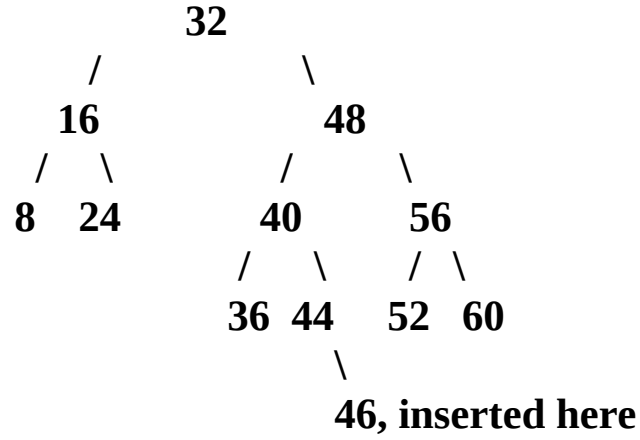
First we will do a normal binary search tree delete. Note that structurally speaking, all deletes from a binary search tree delete nodes with zero or one child. For deleted leaf nodes, clearly the heights of the children of the node do not change. Also, the heights of the children of a deleted node with one child do not change either. Thus, if a delete causes a violation of the AVL Tree height property, this would HAVE to occur on some node on the path from the parent of the deleted node to the root node.

Thus, once again, as above, to restructure the tree after a delete we will call the restructure method on the parent of the deleted node. One thing to node: whereas in an insert there is at most one node that needs to be unbalanced, there may be multiple nodes in the delete that need to be rebalanced. Technically speaking, at any point in the restructuring algorithm ONLY one node will ever be unbalanced. But, what may happen is when that node is fixed, it may propogate an error to an ancestor node. But, this is NOT a problem because our restructuring algorithm goes all the way to the root node. F

Let's trace through a couple examples each for inserts and deletes, using the code handout.

AVL Tree Examples

1) ***Consider inserting 46 into the following AVL Tree:***



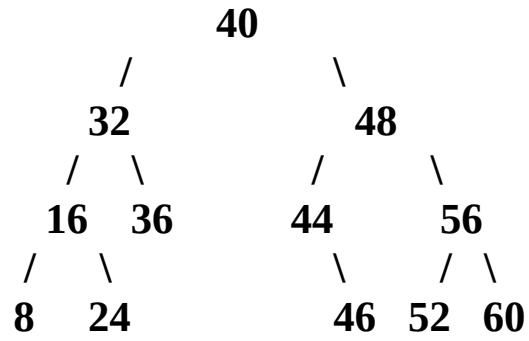
Initially, using the standard binary search tree insert, 46 would go to the right of 44. Now, let's trace through the rebalancing process from this place.

First, we call the method on this node. Once we set its height, we check to see if the node is balanced. (This simply looks up the heights of the left and right subtrees, and decides if the difference is more than 1.) In this case, the node is balanced, so we march up to the parent node, that stores 44.

We will trace through the same steps here, setting the new height of this node (this is important!) and determining that this node is balanced, since its left subtree has a height of -1 and the right subtree has a height of 0.

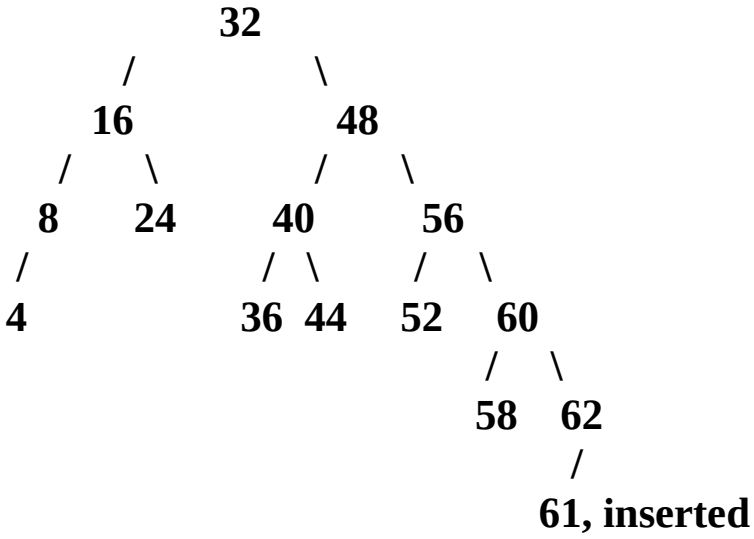
Similarly, we set the height and decide that the nodes storing 40 and 48 are balanced as well. Finally, when we reach the root node storing 32, we realize that our tree is imbalanced.

Now, we finally get to execute the code inside the if statement in the rebalance method. Here we set xPos to be the tallest grandchild of the root node. (This is the node storing 40, since its height is 2.) Thus, the restructuring occurs on the nodes containing the 32, 48 and 40. Using the method described from last lecture, we will restructure the tree as follows:

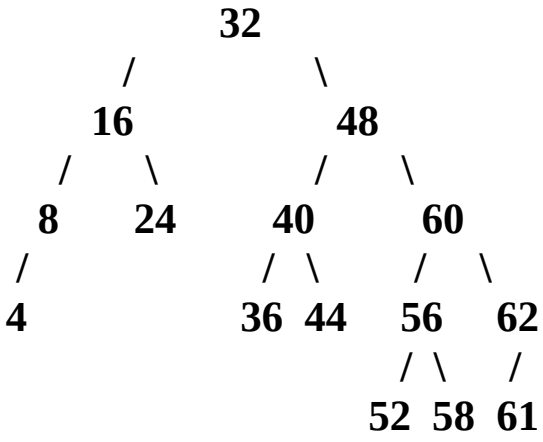


Using the variables from the last lecture, the node storing 40 is B, the node storing 32 is A, and the node storing 48 is C. T_0 is the subtree rooted at 16, T_1 is the subtree rooted at 36, T_2 is the subtree rooted at 44, and T_3 is the subtree rooted at 56.

2) ***Now, for the second example, consider inserting 61 into the following AVL Tree:***

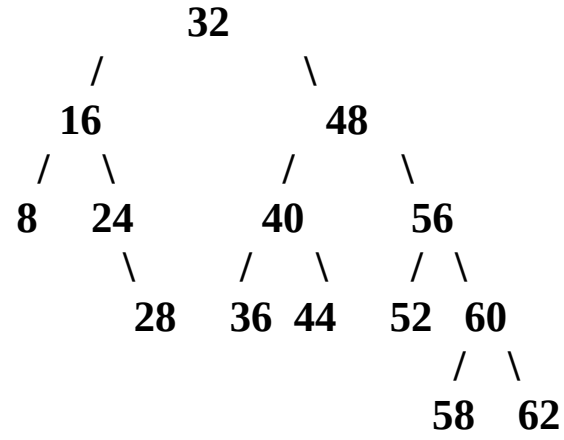


Tracing through the code, we find the first place an imbalance occurs tracing up the ancestry of the node storing 61 is at the noce storing 56. This time, we have that node A stores 56, node B stores 60, and node C stores 62. Using our restucturing algorithm, we find the tallest grandchild of 56 to be 62, and rearrange the tree as follows:



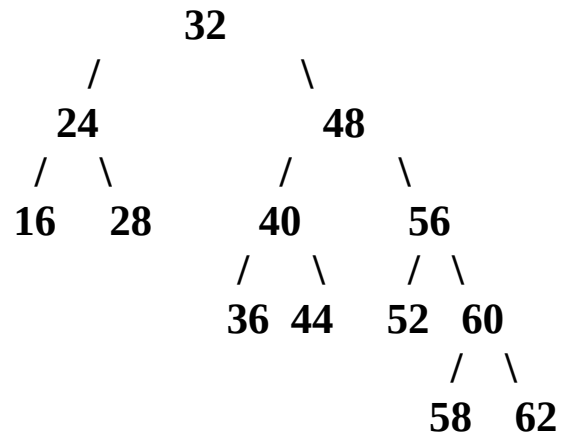
T_0 is the subtree rooted at 52, T_1 is the subtree rooted at 58, T_2 is the subtree rooted at 61, and T_3 is a null subtree.

3) ***For this example, we will delete the node storing 8 from the AVL tree below:***



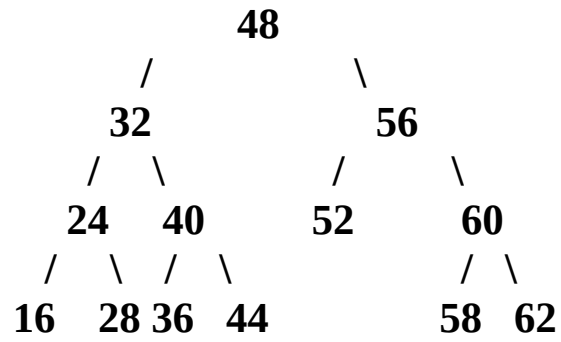
Tracing through the code, we find that we must first call the rebalance method on the parent of the deleted node, which

stores 16. This node needs rebalancing and gets restructured as follows:

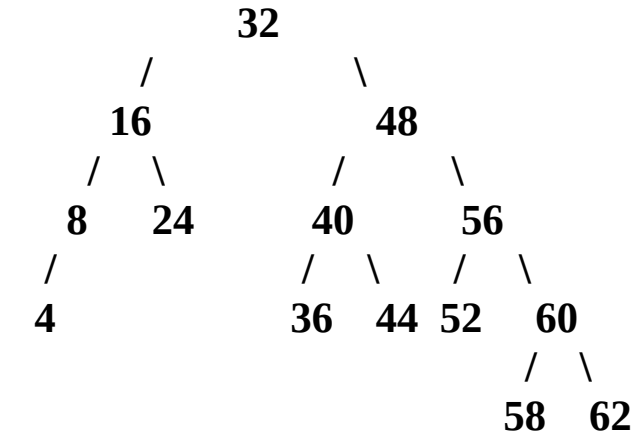


Notice that all four subtrees for this restructuring are null, and we only use the nodes A, B, and C. Next, we march up to the parent of the node storing 24, the node storing 32. Once again, this node is imbalanced. The reason for this is that the restructuring of the node with a 16 reduced the height of that subtree. By doing so, there was in INCREASE in the difference of height between the subtrees of the old parent of the node storing 16. This increase could propogate an imbalance in the AVL tree.

When we restructure at the node storing the 32, we identify the node storing the 56 as the tallest grandchild. Following the steps we've done previously, we get the final tree as follows:



4) ***The final example, we will delete the node storing 4 from the AVL tree below:***



When we call rebalance on the node storing an 8, (the parent of the deleted node), we do NOT find an imbalance at an ancestral node until we get to the root node of the tree. Here we once again identify the node storing 32 as node A, the node storing 48 as node B and the node storing 56 as node C. Accordingly, we restructure as follows: