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 note: 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.

At any point in the restructuring algorithm ONLY one node will ever be unbalanced.

What may happen is when that node is fixed, it may propagate an error to an ancestor node. But, this is NOT a problem because our restructuring algorithm goes all the way to the root node, removing any problems as they appear, one by one.

Choosing the Nodes A, B and C for a Delete Restructuring

One thing that is more complicated about choosing the nodes A, B and C for the AVL Tree delete restructuring is that these nodes are NOT from the ancestral path followed from the origin of the delete.

Clearly, if a delete will cause an imbalance, it will be because the subtree that contains the deleted node has become too short. (Since this subtree can only get "shorter" and the previous version of the tree was balanced, the only possible imbalance is caused when this tree goes from a height of k-1 to k-2, where k is the height of the tree on the other side.)

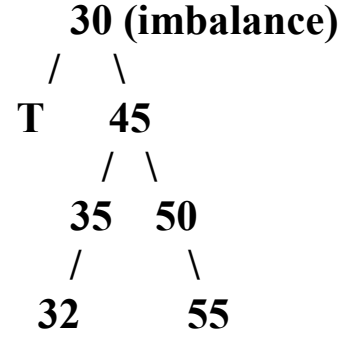
Remember that the nodes A, B and C are always on the "longest" path to the bottom of the tree. This means that when we find an imbalanced node after deleting, the node to the opposite side is guaranteed to be down the longer path.

From there, we have a choice for the third node of A, B and C. We could go to the right or the left.

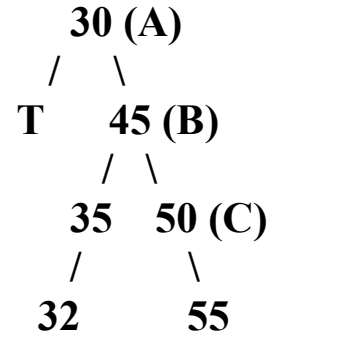
a) If one side is longer than the other, choose that side.
b) If the two sides are equal, go to the same side as the parent is to the grandparent.

This last rule is a bit confusing, so let's clarify with a couple examples.

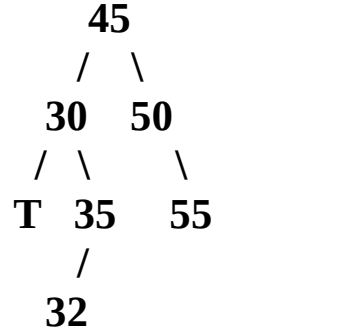
Let's say that the node deleted was in the subtree T shown below (which will store a single node in this instance) and the imbalance is caused at the node storing 30:



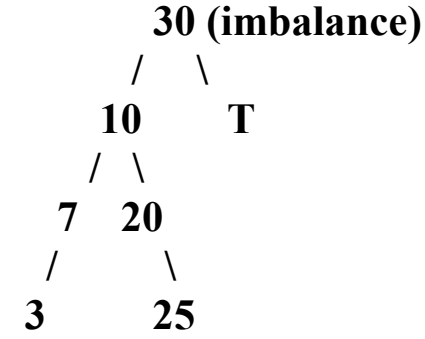
We know that 30 will be one of A, B and C. So will 45. Now we must choose between 35 and 50. Since both subtrees of 45 are of the same height, and since 45 is the RIGHT child of 30, we will choose 45's right child, 50 to be the third node of the group. Thus, our labels are as follows:



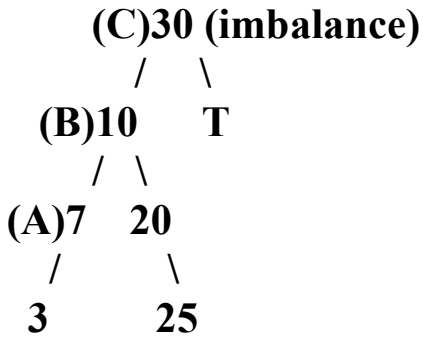
and our restructuring is as follows:



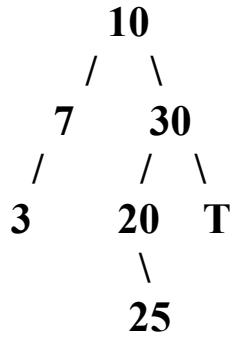
Similarly, if we had the following situation:



Because 10 is the left child of 30, we must choose the left child of 10, 7 to comprise our three nodes:



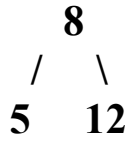
Our rebalance works as follows:



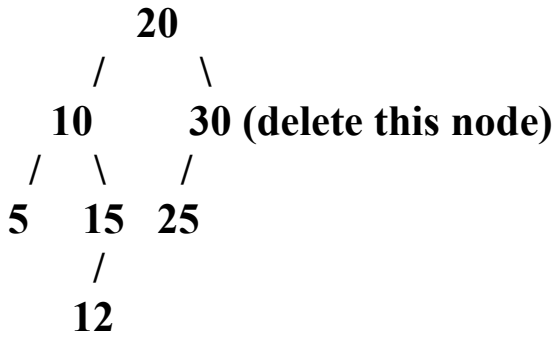
AVL Tree Delete Examples

1) The most simple example is formed when a node from a tree with four nodes gets deleted. In this example, consider the value 12 getting deleted:

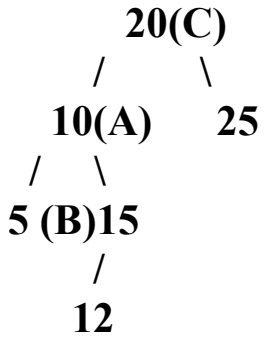
In this instance, after the nod storing 12 is deleted, we move up to the parent, 10. From here, it's fairly obvious that our nodes A, B and C will be the numbers 5, 8 and 10, respectively. Our resulting tree is:



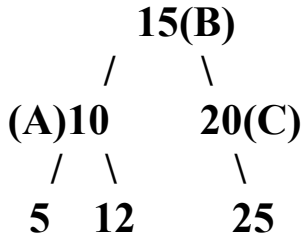
2) Consider deleting 30 from the tree below:



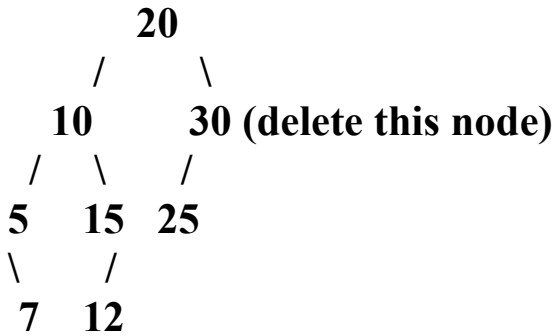
Using the rules for a regular binary search tree delete, we would make the 25 the right child of 20. The 25 is balanced, so then we trace up to the 20. This is unbalanced. To determine A, B and C, we know that our first choice must be to go left from 20. Thus, two of the three values are 20 and 10. Then to choose between 5 and 15, we choose 15 because that subtree is strictly TALLER than the subtree rooted at 5:



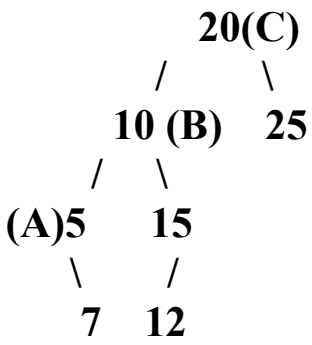
The restructure is as follows:



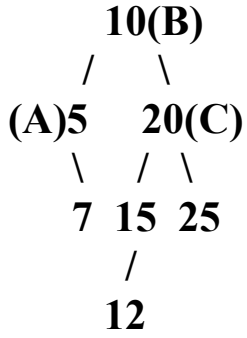
3) Now consider deleting the value 30 from the following slightly different tree:



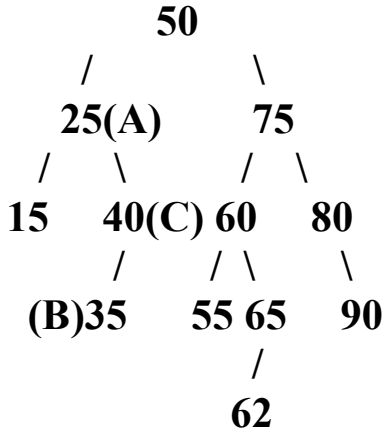
This situation, what has changed is that both 5 and 15 have subtrees of the exact same height. In this case, we must choose the direction of the first node to the second. Since 10 is the left child of 20, we must choose 5, the left child of 10 instead of 15. The resulting labels are as follows:



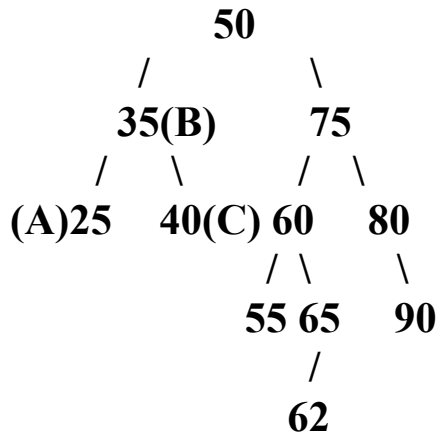
The resulting tree is:



4) Consider deleting the value 15 from the following tree:

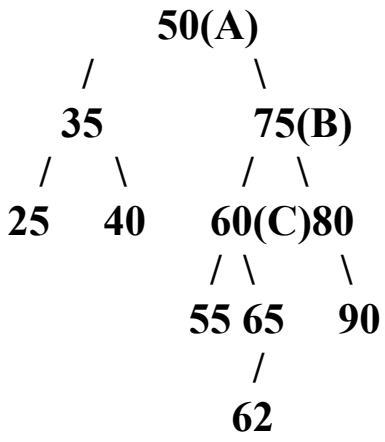


First, we can see that when we get to 25, we will have an imbalance. This situation is similar to the first case we saw, so we label A = 25, B = 35 and C = 40. Following this restructuring, we have:

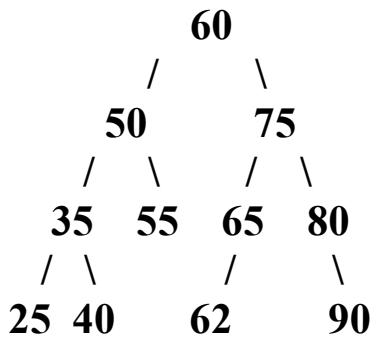


The problem, of course, is that now, 35 is balanced, but as we continue up the tree to 50, we've introduced a problem here, since we made the left of 50 shorter by one level.

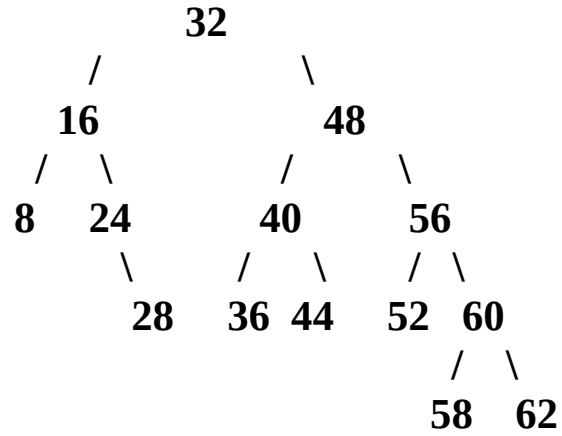
In this case, we will choose 50 and 75 to be two of our three nodes. To choose the third, we see that the tree rooted at 60 is strictly taller than the one rooted at 80. Thus, we have that A = 50, B = 50 and C = 75:



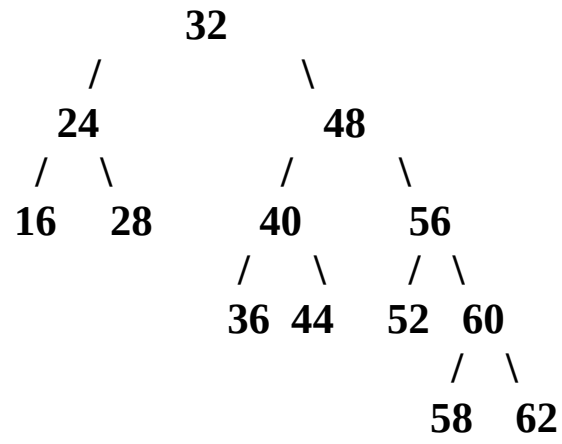
Finally, to do the restructuring, we'll have 60 be the new root of the tree, 50 to the left and 75 to the right:



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



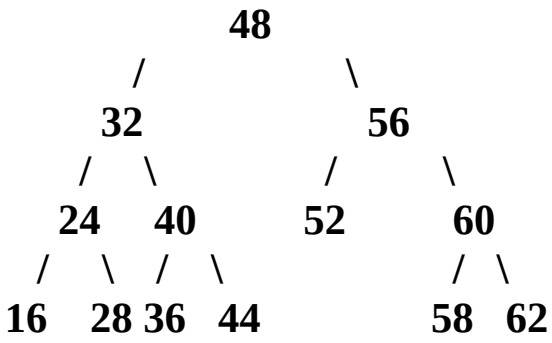
We must first rebalance the node storing 16, resulting in:



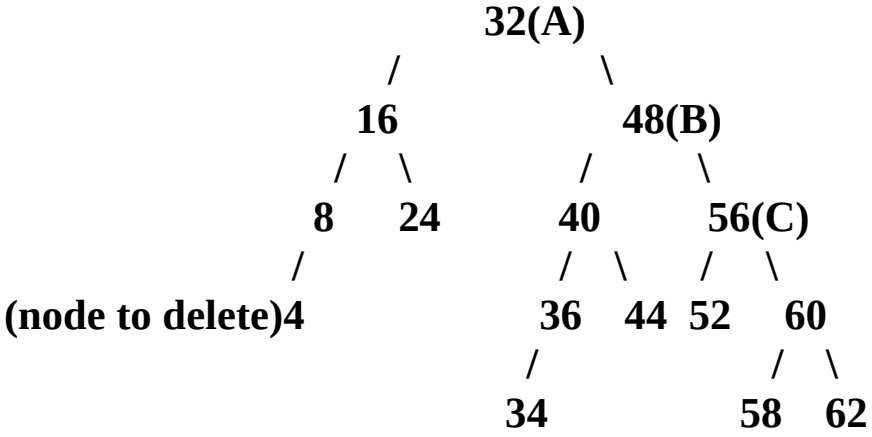
Notice that all four subtrees for this restructuring are null, and we only use the nodes A, B, and C. Next, we march up to the parent of the node storing 24, the node storing 32. Once again, just like the previous example, this node is imbalanced. The reason for this is that the restructuring of the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



6) In 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 identify both 32 and 48, like before as two of our three nodes. Now, when we decide between 40 and 56, we find the height of both subtrees to be the same. In this case, since 48 is the right child of 32, we must pick the right child of 48, which is 56, to be the third node. Thus A = 32, B = 48 and C = 56.

Accordingly, we restructure as follows: