Search Tree Assignment

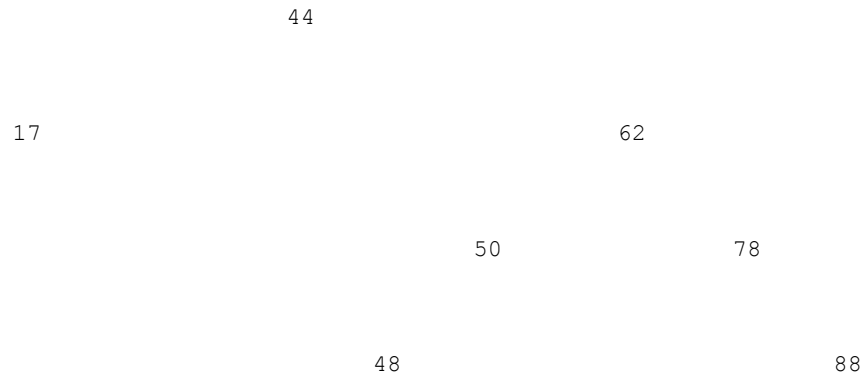
The final ModelBinarySearchTree class implements setitem and getitem operations, but it does not implement a delitem operation. Your assignment is to use ModelBinarySearchTree as a starting point for writing an AVLTree class that adds this capability by adding a `__delitem__()` method to your class.

There are two cases to be handled when deleting an item:

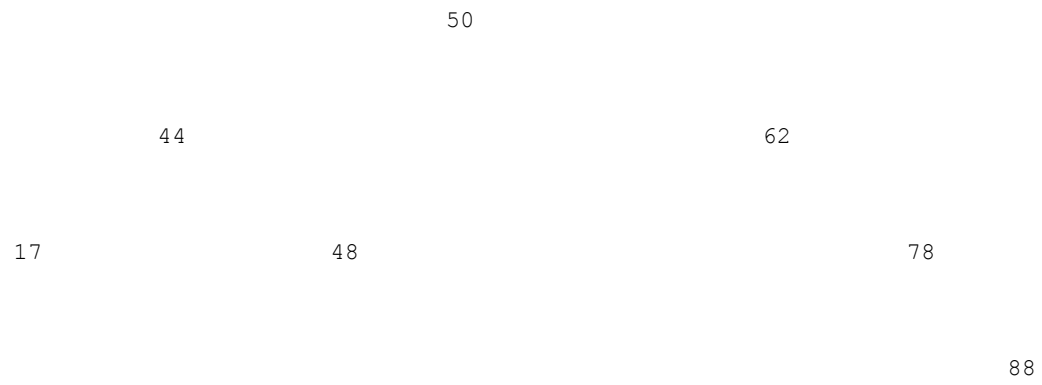
Case 1: The node to be deleted has at most one child. In this case, we can simply relink the child of the node to be deleted (if any) as the correct child (left or right) of the node's parent. Then we rebalance the tree, beginning with the parent of the deleted node.

Case 2: The node to be deleted has two children. In this case, we'll actually replace the data in this node with data from a different node and then delete that other node. Specifically, we will locate the node that immediately precedes, in an inorder traversal, the node to be deleted. This node is guaranteed to have at most one child. We will copy the key/value pair from this preceding node to the node that we were told to delete, which will remain in the tree. Finally, we'll treat the node from which we copied data as the node to be deleted and proceed as in Case 1, deleting this node and then rebalancing beginning with the node's parent.

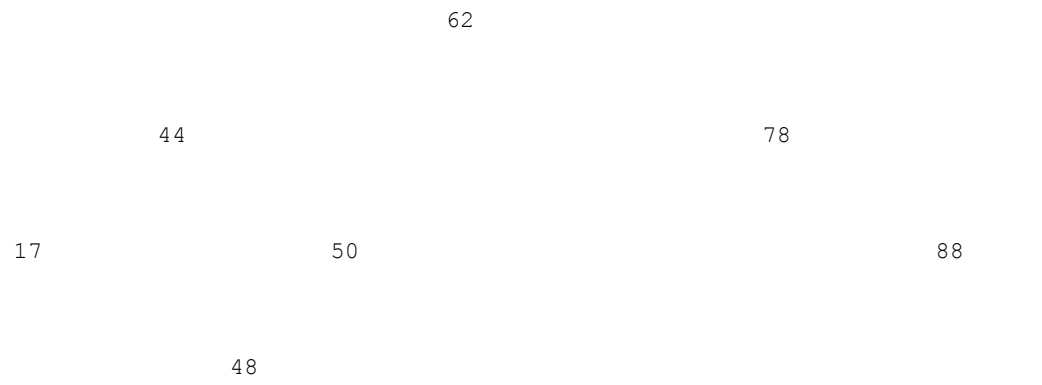
There is one other complication: The rebalancing must be extended slightly from the version used for inserting so that it breaks ties in height in a certain way (this was not an issue for insertion because there cannot be ties in that case). Specifically, let z represent a node that needs to be rebalanced and let y be the taller child of z (one of the children will definitely be taller). It might be that both children of y have the same height. In this case, the "tallest grandchild" x of z must be defined to be the child of y that makes the path from z to x straight rather than zig-zag. That is, if y is the left child of z and both children of y have the same height then x must be chosen to be the left child of y, and similarly if y is the right child of z the right child of y must be given preference in case of a tie. Otherwise, when rebalancing a tree such as this one:



which has an unbalanced node at 44, if `_restructure()` is passed grandchild 50 (the zig-zag grandchild) it will do two rotations of the 50 node, resulting in this tree



which has an unbalanced node at 62. But if instead `_restructure()` is passed grandchild 78, the straight-line grandchild, then it will do a single rotation of the 62 node and produce this AVL tree:



To summarize, you need to:

- Write code that can find the immediate predecessor of a node according to an inorder ordering of the nodes.
    - The immediate predecessor of a node is the rightmost descendant of the left child of the node, or the left child itself if that child has no right child.
- Modify the `_tall_grandchild()` method so that it breaks ties in heights of grandchildren as described above (this should have no effect on inserts, since heights are always unequal in that case).
- Write a `__delitem__()` method that uses the previous two methods along with others already provided to implement AVL deletion as described above.

- You'll probably want to use the existing `_subtree_search()`, `_relink()`, and `_rebalance()` methods.

You can test your code using the main method of the posted AVL_tree.py file as a starting point. You'll call `__delitem__()` indirectly using code such as

```
del t4[32]
```

where `t4` is an AVL tree belonging to your class.