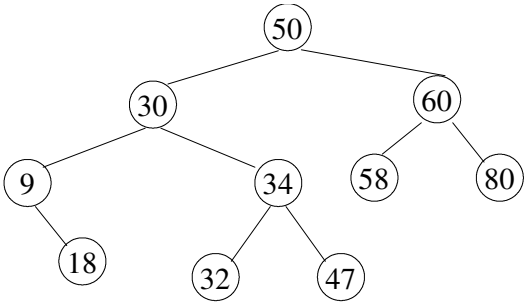
Data Structures (810:052) Name:_____

1. Complete the `findHelper` and `findReplacement` functions of the `remove` method for the BST class.

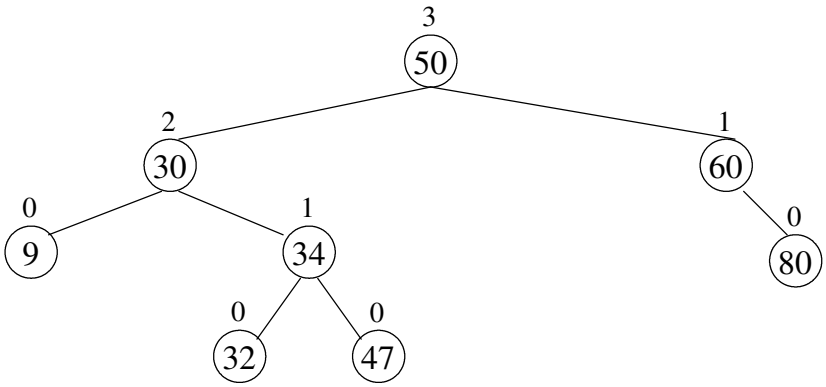
```
def remove(self, item):
    """Removes the item an returns data if item is found or None otherwise."""
    def findHelper(tree, parent):
```




```
    # end findHelper
    def findReplacement(tree, parent):

    # end findReplacement
    if self.isEmpty():
        return None
    elif self._tree.getRoot() == item and self._tree.getLeft().isEmpty():
        #
        #
        itemValue = self._tree.getRoot()
        self._tree = self._tree.getRight()
    elif self._tree.getRoot() == item and self._tree.getRight().isEmpty():
        #
        #
        itemValue = self._tree.getRoot()
        self._tree = self._tree.getLeft()
    else:
        #
        #
        itemValue, itemSubtree, itemParent = findHelper(self._tree, None)
        if itemValue == None:
            return None

        if itemSubtree.getLeft().isEmpty():
            if itemParent.getLeft() == itemSubtree:
                itemParent.setLeft(itemSubtree.getRight())
            else:
                itemParent.setRight(itemSubtree.getRight())

        elif itemSubtree.getRight().isEmpty():
            if itemParent.getLeft() == itemSubtree:
                itemParent.setLeft(itemSubtree.getLeft())
            else:
                itemParent.setRight(itemSubtree.getLeft())
        else:  #  item being removed has two children
            replacementValue,replacementSubtree,replacementParent=findReplacement(itemSubtree.getLeft(),itemSubtree
            itemSubtree.setRoot(replacementValue)
            if replacementParent == itemSubtree:
                itemSubtree.setLeft(replacementSubtree.getLeft())
            else:
                replacementParent.setRight(replacementSubtree.getLeft())

    self._size -= 1
    return itemValue
```

2. Complete the comments in the above code.

3. An *AVL Tree* is a special type of Binary Search Tree (BST) that it is *height balanced*. By height balanced I mean that the height of every nodes left and right subtrees differ by at most one. This is enough to guarantee that a AVL tree with n nodes has a height no worst than $\Theta(\log_2 n)$. Therefore, insertions, deletions, and search are in the worst case $\Theta(\log_2 n)$. An example of an AVL tree with integer keys is shown below. The height of each node is shown.



Each AVL-tree node usually stores a *balance factor* in addition to its key and data. The balance factor keeps track of the relative height difference between its left and right subtrees.

a) Label each node in the above AVL tree with one of the following *balance factors*:

- 'EQ' if its left and right subtrees are the same height
- 'TL' if its left subtree is one taller than its right subtree
- 'TR' if its right subtree is one taller than its left subtree

b) We start an `add` operation by adding the new item into the AVL as leaves just like we did for Binary Search Trees (BSTs). Add the key 90 to the above tree?

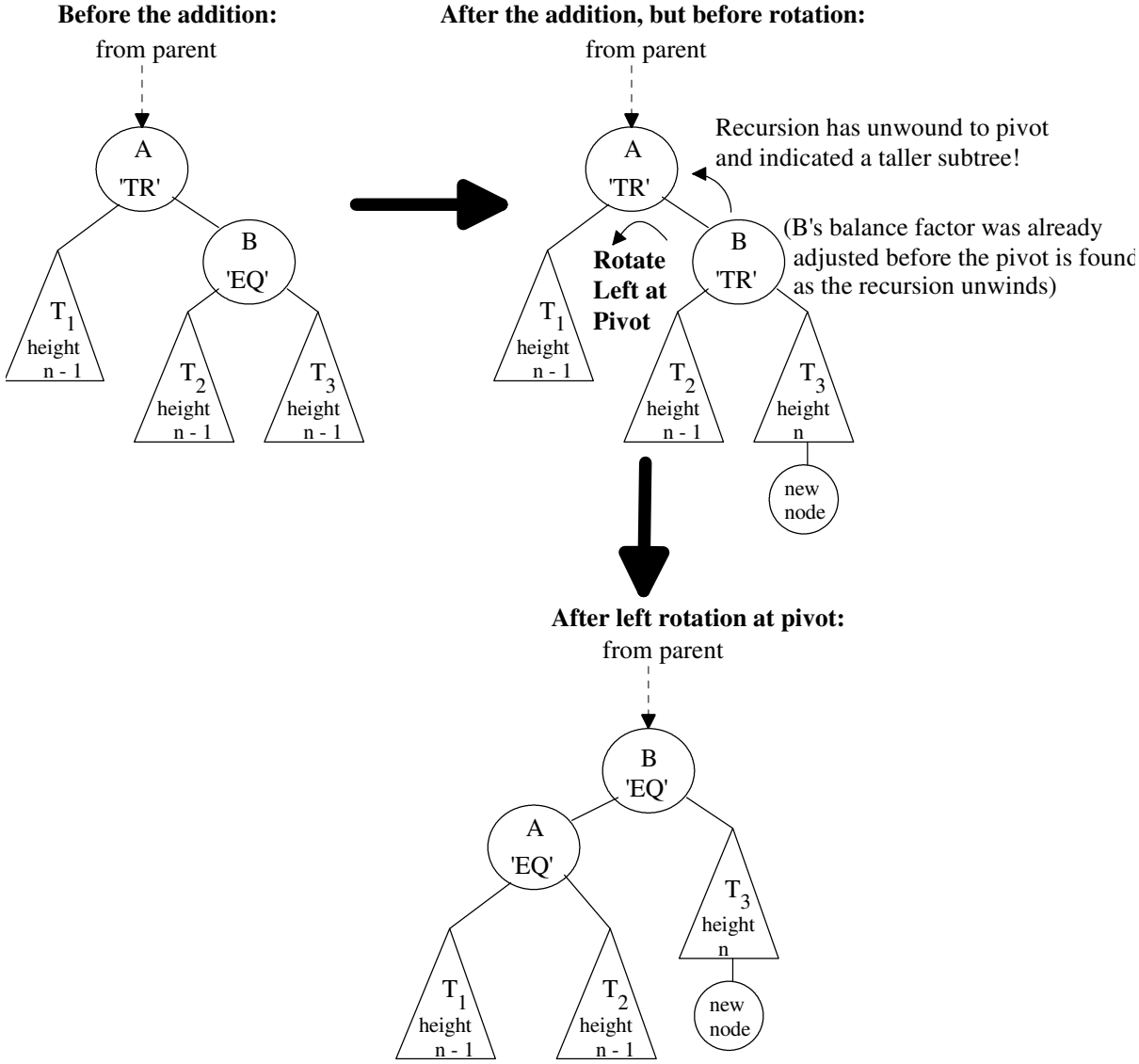
c) Identify the node "closest up the tree" from the inserted node (90) that no longer satisfies the height balanced property of an AVL tree. This node is called the *pivot node.* Label the pivot node above.

d) Consider the subtree whose root is the pivot node. How could we rearrange this subtree to restore the AVL height balanced property of AVL tree? (Draw the rearranged tree below)

4. Typically, the addition of a new key into an AVL requires the following steps:

- compare the new key with the current tree node's key (as we did in the `addHelper` function inside the `add` method in the BST) to determine whether to recursively add the new key into the left or right subtree
- add the new key as a leaf as the base case(s) to the recursion
- as the recursion "unwinds" (i.e., after you return from the recursive call) adjust the balance factors of the nodes on the search path from the new node back up to the root of the tree. To aid in adjusting the balance factors, we'll modify the `addHelper` function so that it returns `True` if the subtree got taller and `False` otherwise.
- as the recursion "unwinds" if we encounter a pivot node (as in question (c) above) we perform one or two "rotations" to restore the AVL tree's height-balanced property.

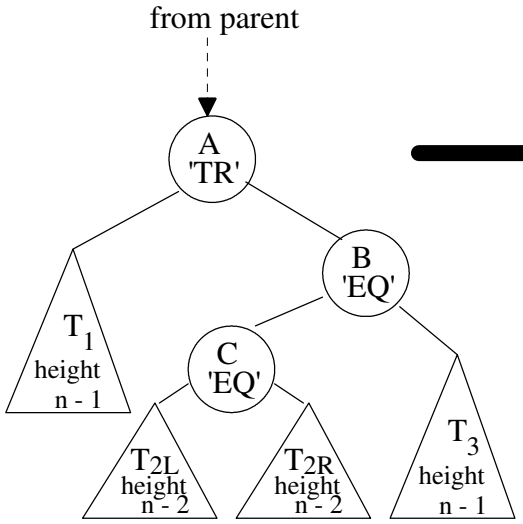
For example, consider the previous example of adding 90 to the AVL tree. Before the addition, the pivot node was already "TR" (tall right - right subtree had a height one greater than its left subtree). After inserting 90, the pivot's right subtree had a height 2 more than its left subtree which violates the AVL tree's height-balance property. This problem is handled with a *left rotation* about the pivot as shown in the following generalized diagram:



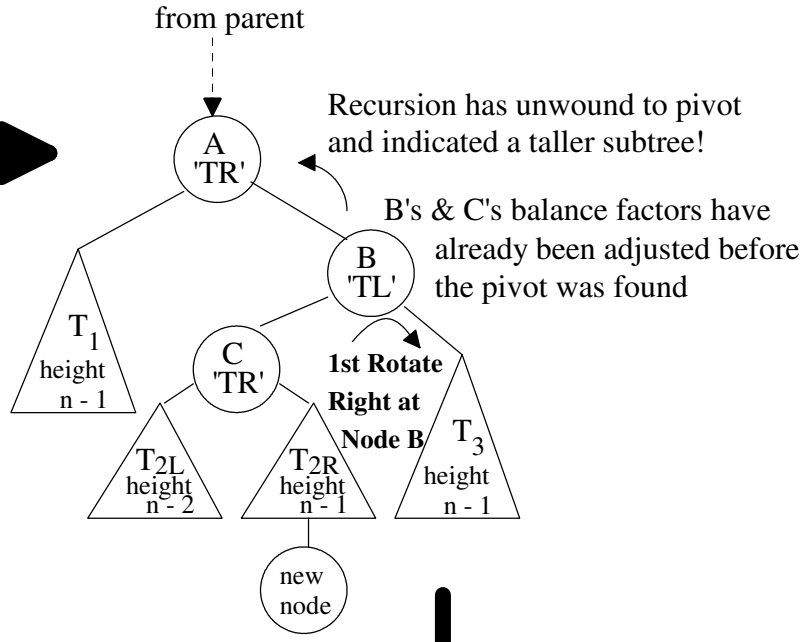
a) Assuming the same initial AVL tree (upper, left-hand of above diagram) if the new node would have increased the height of T_2 (instead of T_3), would a left rotation about the node A have rebalanced the AVL tree?

Before the addition, if the pivot node was already "TR" (tall right) and if the new node is inserted into the left subtree of the pivot node's right child, then we must do two rotations to restore the AVL-tree's height-balance property.

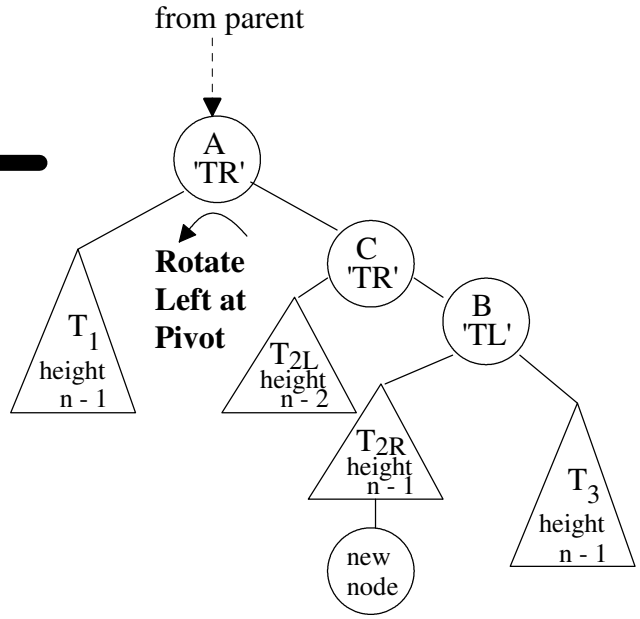
Before the addition:



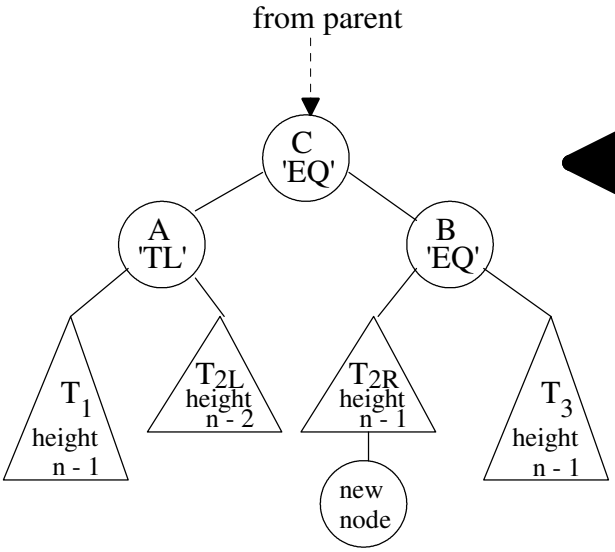
After the addition, but before first rotation:



After right rotation at B, but before left rotation at pivot:



After the left rotation at pivot and balance factors adjusted correctly:



b) Suppose that the new node was added into the right subtree of the pivot's right child, i.e., inserted in T_{2L} instead T_{2R}, then the same two rotations would restore the AVL-tree's height-balance property. However, what should the balance factors of nodes A, B, and C be after the rotations?