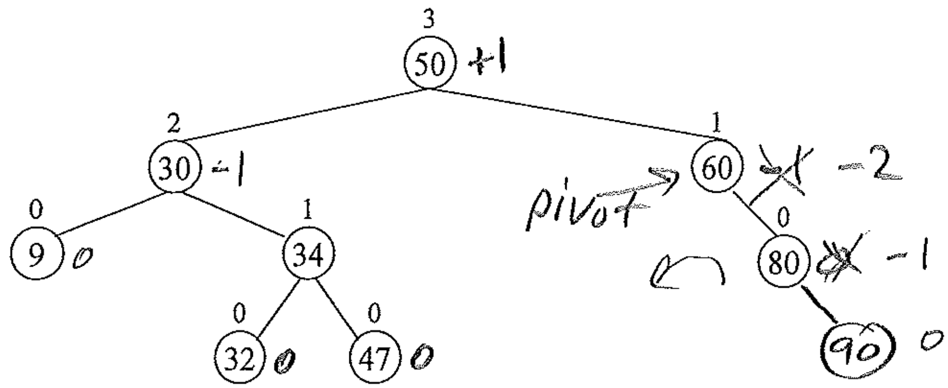
 Name: Mark F.

1. 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 node's 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 $O(1.44 \log_2 n)$. Therefore, insertions, deletions, and search are worst case $O(\log_2 n)$. 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 payload. The balance factor keeps track of the relative height difference between its left and right subtrees, i.e., height(left subtree) - height(right subtree).

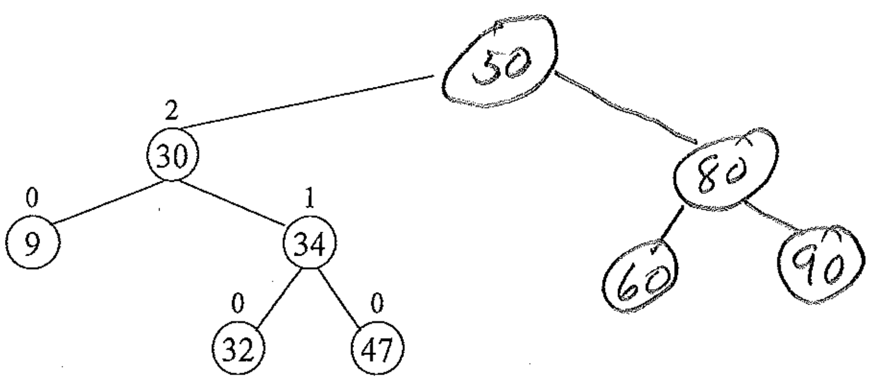
a) Label each node in the above AVL tree with one of the following *balance factors*:

- 0 if its left and right subtrees are the same height
- 1 if its left subtree is one taller than its right subtree
- -1 if its right subtree is one taller than its left subtree

b) We start a `put` operation by adding the new item into the AVL as a leaf 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.

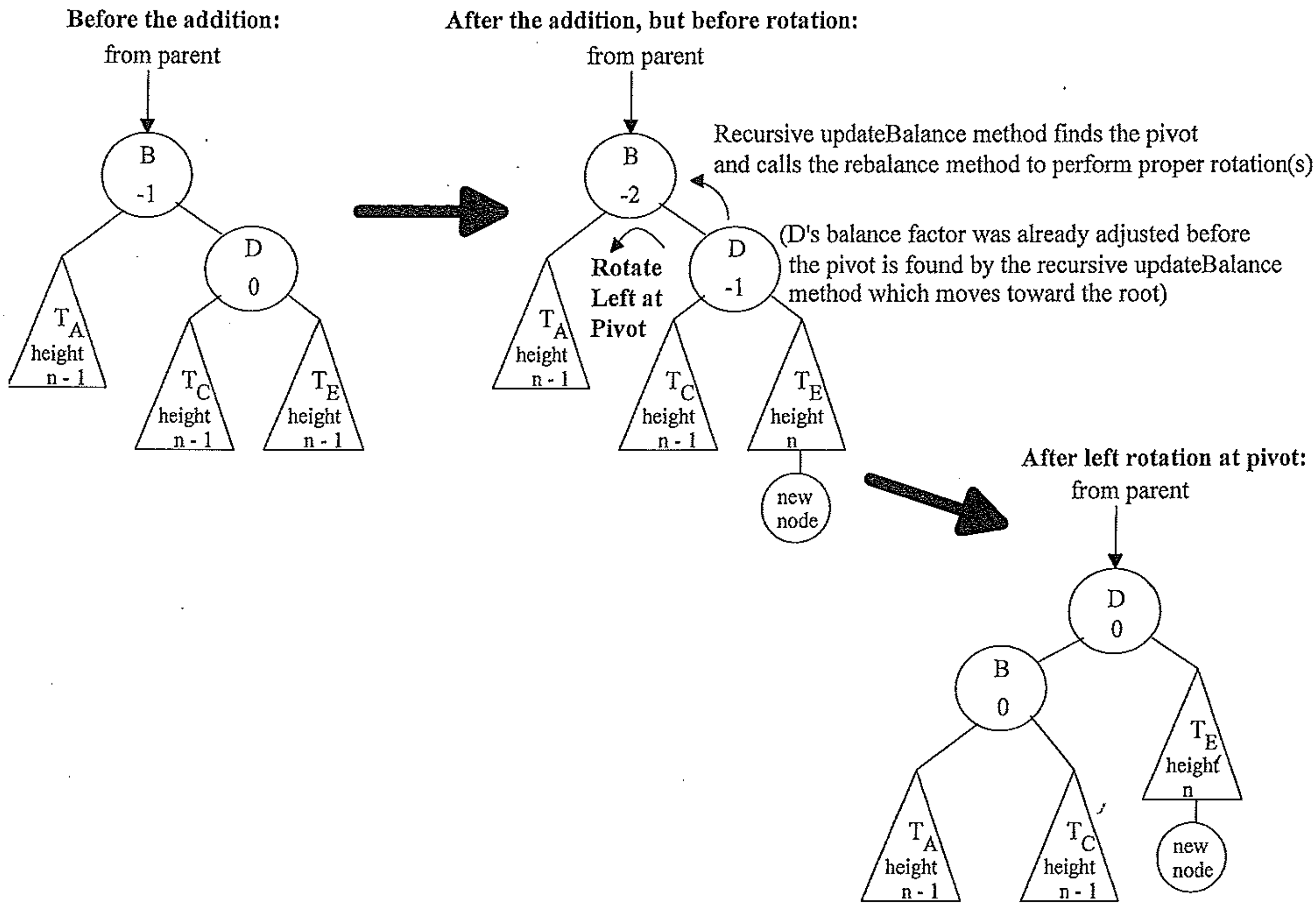
d) Consider the subtree whose root is the pivot node. How could we rearrange this subtree to restore the AVL height balanced property? (Draw the rearranged tree below)



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

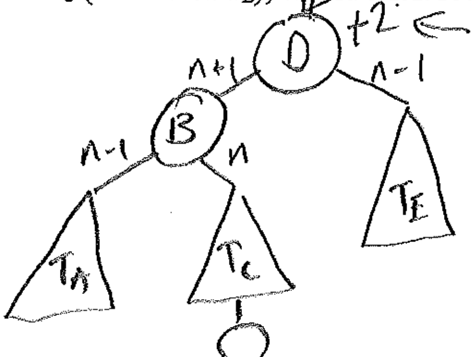
- compare the new key with the current tree node's key (as we did in the `_put` function called by the `put` method in the BST) to determine whether to recursively add the new key into the left or right subtree
- add the new key as a leaf as the base case(s) to the recursion
- recursively (`updateBalance` method) adjust the balance factors of the nodes on the search path from the new node back up toward the root of the tree. 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 (60) was already -1 ("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 (balance factor -2) 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:

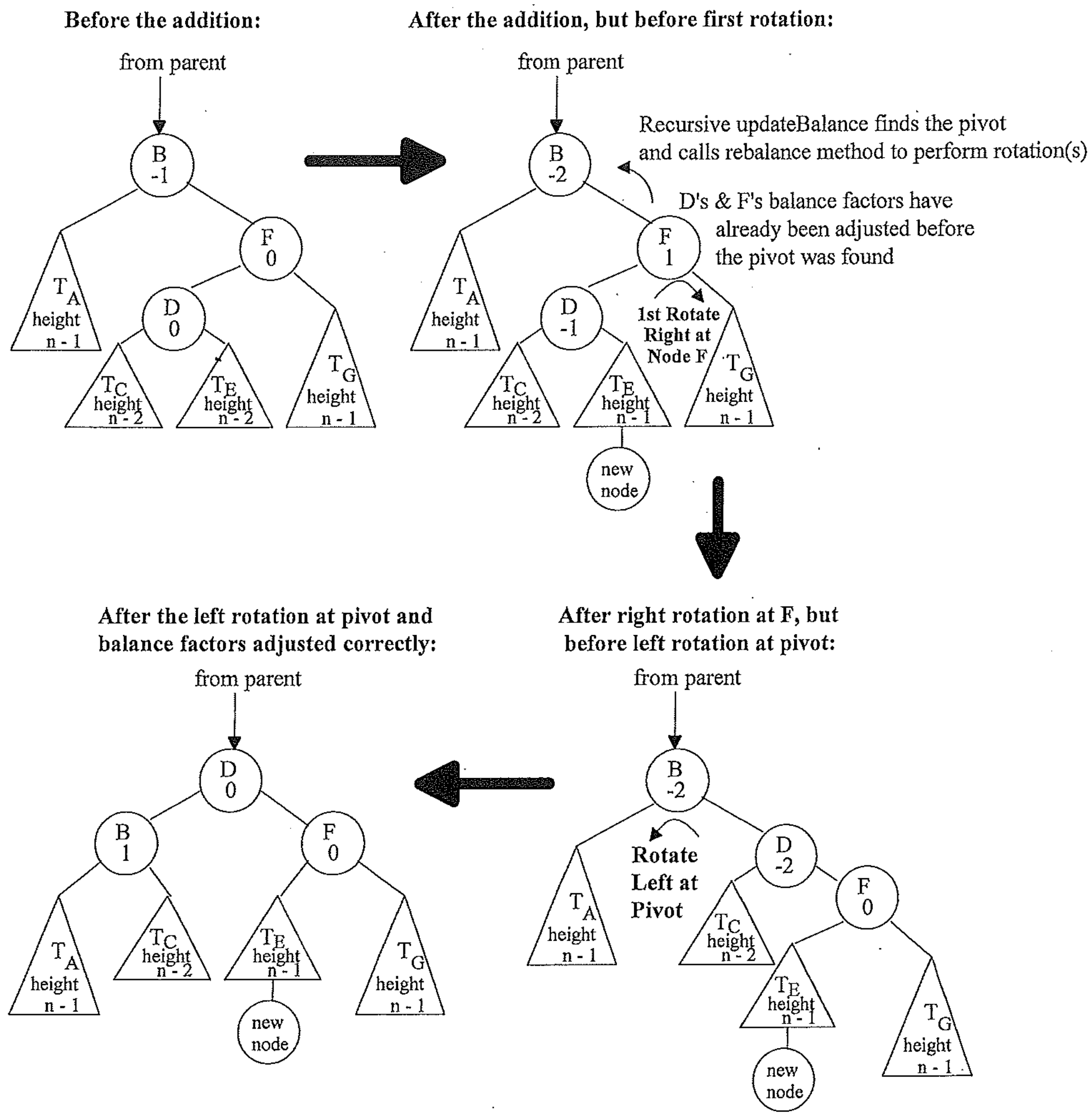


a) Assuming the same initial AVL tree (upper, left-hand of above diagram) if the new node would have increased the height of T_C (instead of T_E), would a left rotation about the node B have rebalanced the AVL tree? No

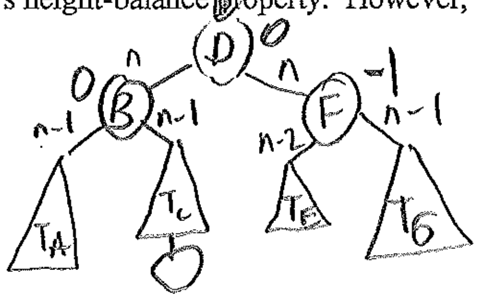
+2 ← balance factor node D not 0, 1, -1



b) Before the addition, if the pivot node was already -1 (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.



b) Suppose that the new node was added in T_C instead of T_E, then the same two rotations would restore the AVL-tree's height-balance property. However, what should the balance factors of nodes B, D, and F be after the rotations? 0 0 -1



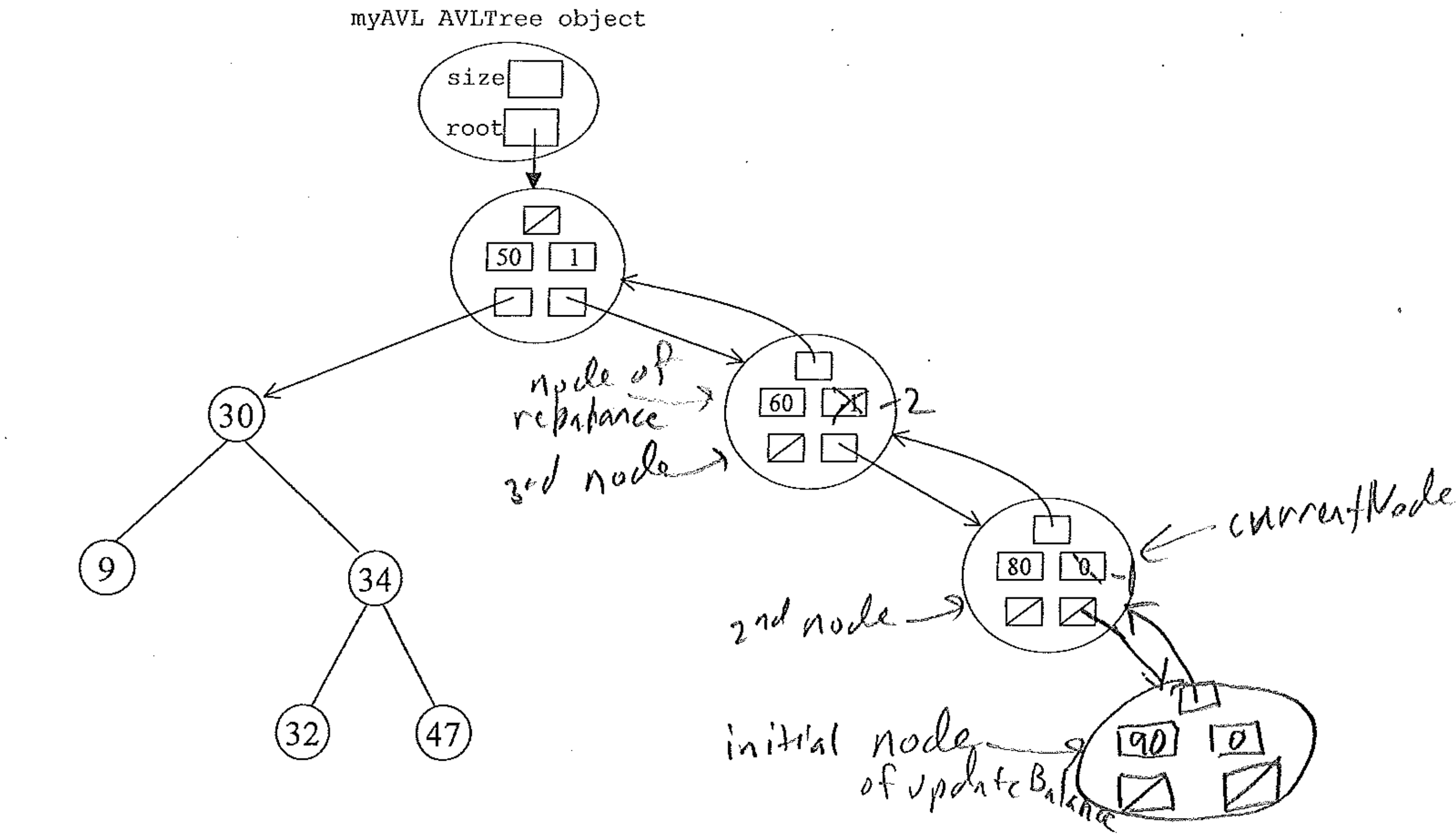
Consider the AVLTreeNode class that inherits and extends the TreeNode class to include balance factors.

```
from tree_node import TreeNode

class AVLTreeNode(TreeNode):
    def __init__(self,key,val,left=None,right=None,parent=None, balanceFactor=0):
        TreeNode.__init__(self,key,val,left,right,parent)
        self.balanceFactor = balanceFactor
```

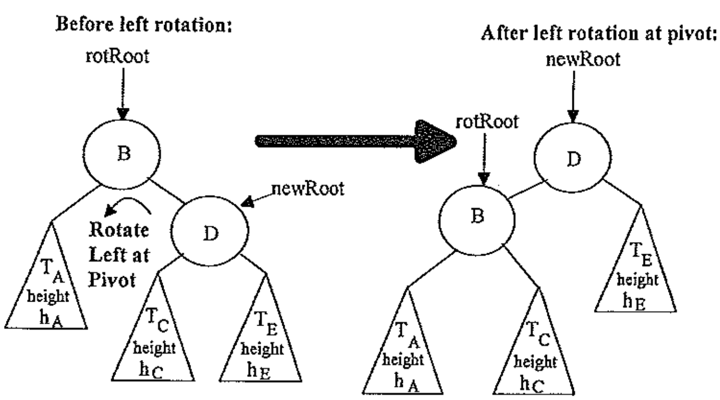
Now let's consider the partial AVLTree class code that inherits from the BinarySearchTree class:

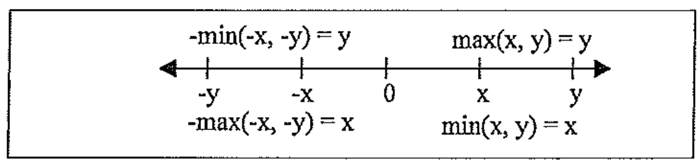
```
from avl_tree_node import AVLTreeNode
from binary_search_tree import BinarySearchTree

class AVLTree(BinarySearchTree):

    def put(self,key,val):
        if self.root:
            self._put(key,val,self.root)
        else:
            self.root = AVLTreeNode(key,val)
        self.size = self.size + 1

    def _put(self,key,val,currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
            else:
                currentNode.leftChild = AVLTreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.leftChild)
        elif key > currentNode.key:
            if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
            else:
                currentNode.rightChild = AVLTreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.rightChild)
        else:
            currentNode.payload = val   ← self.size -= 1
    def updateBalance(self,node):
        if node.balanceFactor > 1 or node.balanceFactor < -1:
            self.rebalance(node)
            return
        if node.parent != None:
            if node.isLeftChild():
                node.parent.balanceFactor += 1
            elif node.isRightChild():
                node.parent.balanceFactor -= 1

            if node.parent.balanceFactor != 0:
                self.updateBalance(node.parent)

    def rotateLeft(self,rotRoot):                    ## NOTE:  You will complete rotateRight in Lab
        newRoot = rotRoot.rightChild
        rotRoot.rightChild = newRoot.leftChild
        if newRoot.leftChild != None:
            newRoot.leftChild.parent = rotRoot
        newRoot.parent = rotRoot.parent
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
            else:
                rotRoot.parent.rightChild = newRoot
        newRoot.leftChild = rotRoot
        rotRoot.parent = newRoot
        rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(newRoot.balanceFactor, 0)
        newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFactor, 0)
    def rebalance(self,node):   ← -2 or 2 for balanceFactor
        if node.balanceFactor < 0:
            if node.rightChild.balanceFactor > 0:
                self.rotateRight(node.rightChild)
                self.rotateLeft(node)
            else:
                self.rotateLeft(node)
        elif node.balanceFactor > 0:
            if node.leftChild.balanceFactor < 0:
                self.rotateLeft(node.leftChild)
                self.rotateRight(node)
            else:
                self.rotateRight(node)
```

c) Trace the code for `myAVL.put(90,None)` by updating the below diagram:



Consider balance factor formulas for rotateLeft. We know: $newBal(B) = h_A - h_C$ and $oldBal(B) = h_A - (1+max(h_C, h_E))$
$newBal(D) = 1 + max(h_A, h_C) - h_E$ and $oldBal(D) = h_C - h_E$





Consider: $newBal(B) - oldBal(B)$

$newBal(B) - oldBal(B) = h_A - h_C - h_A + (1+max(h_C, h_E))$

$newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C$

$newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C$

$newBal(B) = oldBal(B) + 1 + max(h_C - h_C, h_E - h_C)$

$newBal(B) = oldBal(B) + 1 + max(0, -oldBal(D))$

$newBal(B) = oldBal(B) + 1 - min(0, oldBal(D))$, so

```
rotRoot.balanceFactor = rotRoot.balanceFactor + 1 -
                        min(newRoot.balanceFactor, 0)
```

d) Consider: newBal(D) - oldBal(D) $= 1 + max(h_A, h_C) - h_E - (h_C - h_E)$

$= 1 + max(h_A, h_C) - \cancel{h_E} - h_C + \cancel{h_E}$

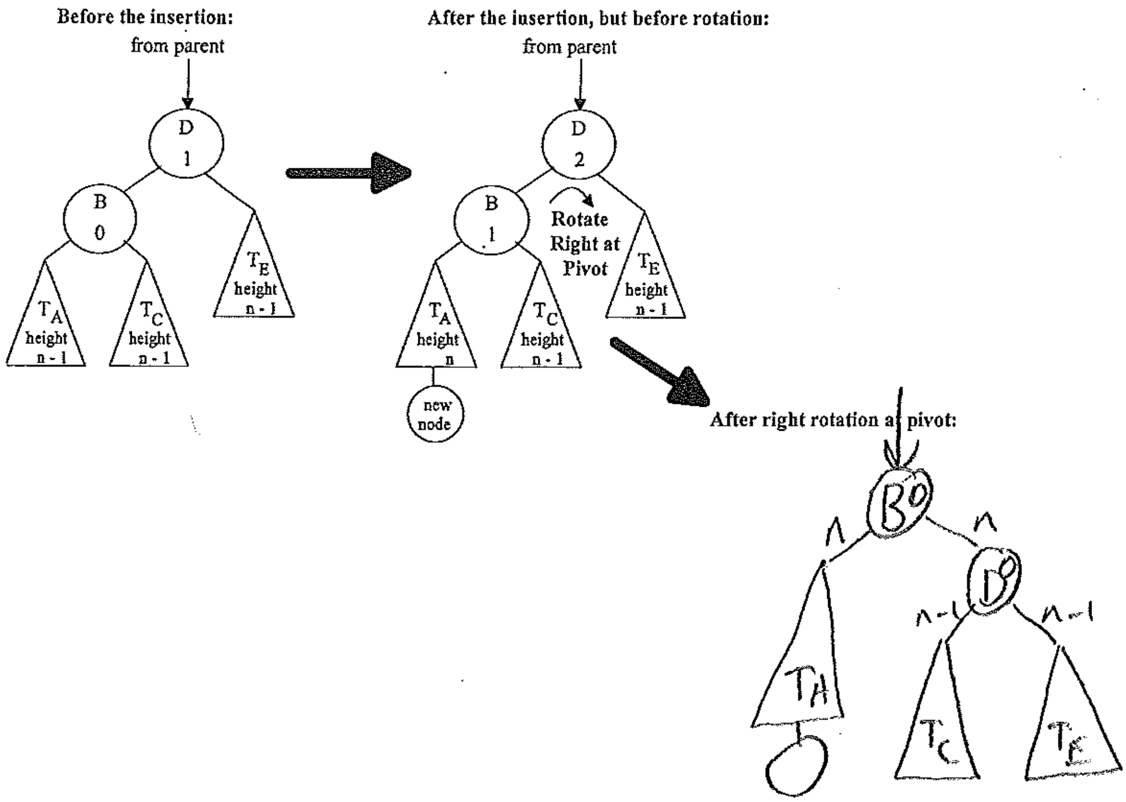
$= 1 + max(h_A - h_C, h_C - h_C)$

$newBal(D) - oldBal(D) = 1 + max(newBal(B), 0)$ previously updated to be newBal(B)

$newBal(D) = oldBal(D) + 1 + max(newBal(B), 0)$

newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFactor, 0)

3. Complete the below figure which is a "mirror image" to the figure on page 2, i.e., inserting into the pivot's left child's left subtree. Include correct balance factors after the rotation.



b) Complete the below figure which is a "mirror image" to the figure on page 3, i.e., inserting into the pivot's left child's right subtree. Include correct balance factors after the rotation.