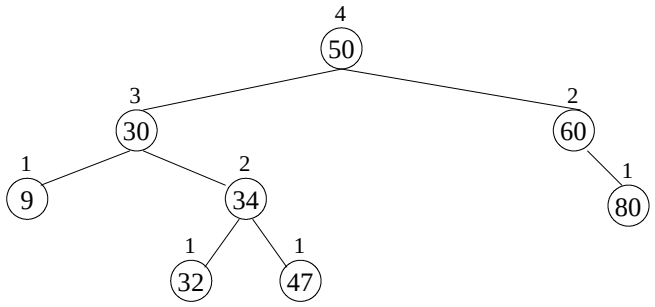
Team #:_____ Name:_____
Absent:

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 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.



a) Label each node 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

Each AVL tree node usually maintains a balance factor in addition to the key, payload, etc.

2. We'll add new nodes to the AVL as leaves just like we did for Binary Search Trees (BSTs).
a) Add the key 90 to the tree?

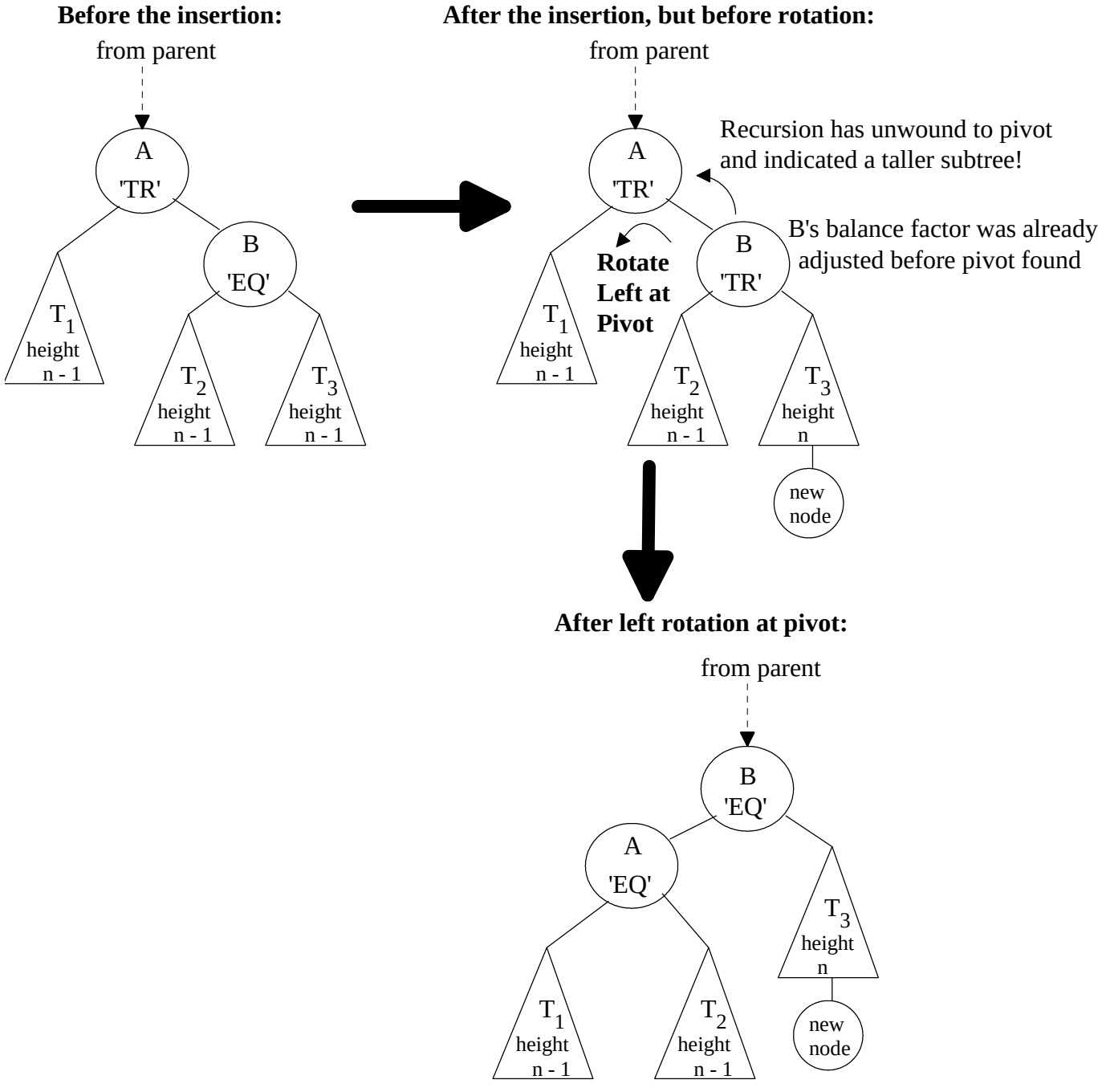
b) Identify the node "closest" to the inserted node (90) that no longer satisfies the height balanced property of an AVL tree. This node is called the *pivot node.*

c) 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 tree resulting tree below)

3. Typically, an insertion 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* method in the BST) to determine whether to recursively insert/put 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, will modify the insertiong/put method 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 2 above) we perform one or two "rotations" to restore the AVL tree's height-balanced property.

Team #:_____ Name:_____
Absent:

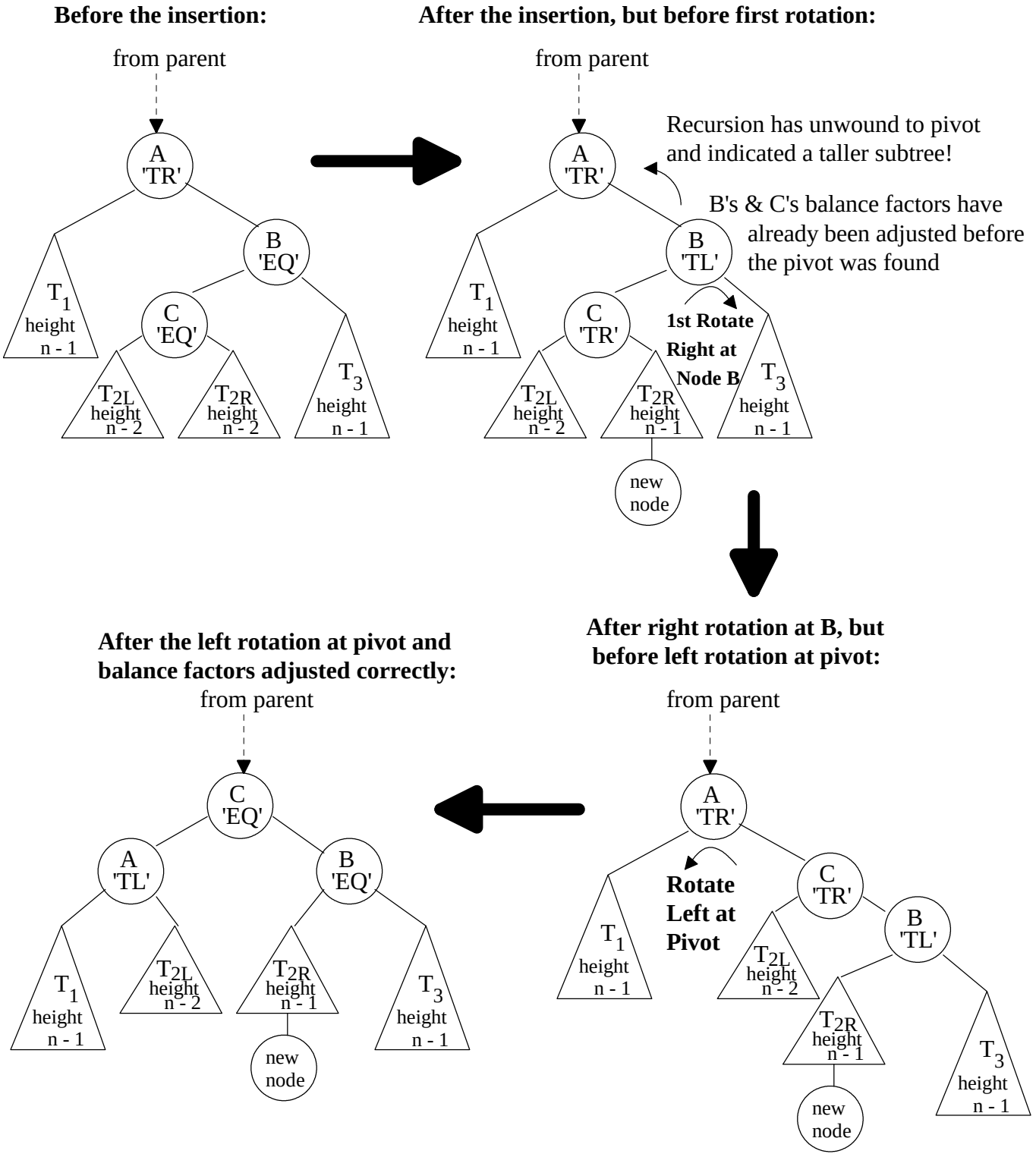
For example, consider the previous example of adding 90 to the AVL tree. Before the insertion the pivot node was already "TR" (tall right - right subtree had a height one greater than its left subtree). After inserting 90, the pivot's right subtree had a height 2 more than its left subtree which violates 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 (node A is TR) if the new node would have increased the height of T_2, would a left rotation about the node A have rebalanced the AVL tree?

Team #:_____ Name:_____
Absent:

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



a) Suppose that the new node was inserted 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?

Team #:_____ Name:_____
Absent:

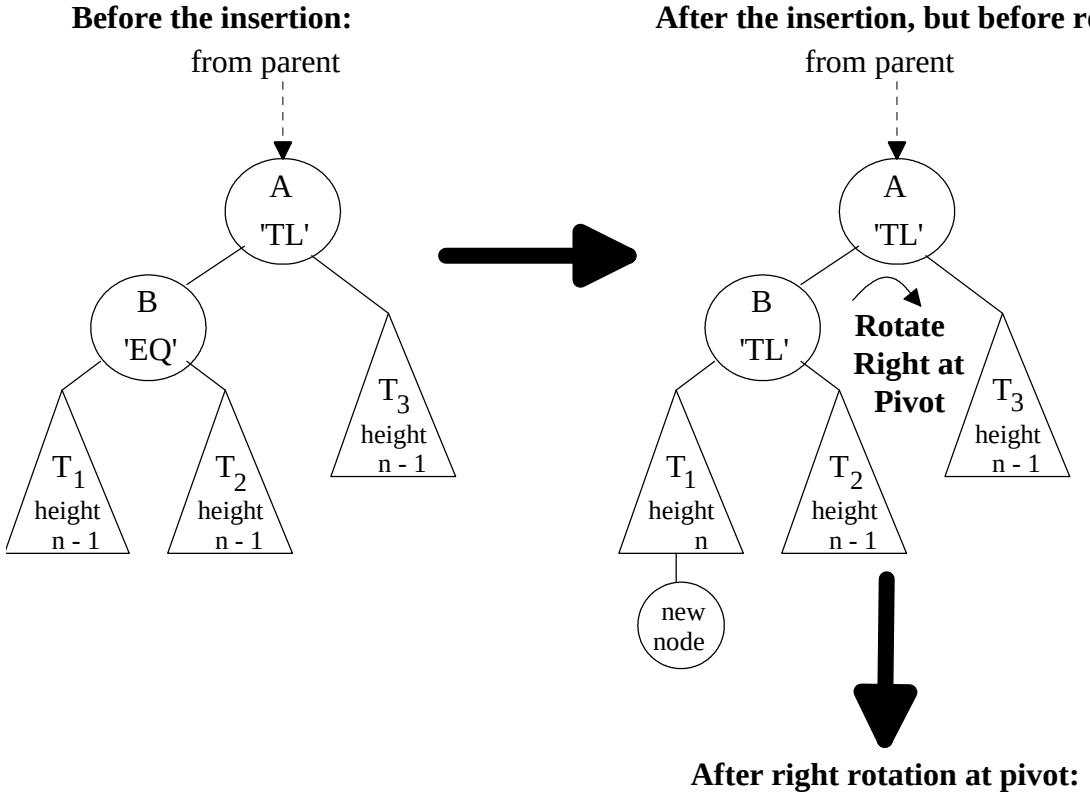
5. Now let's consider the partial AVL tree node code:

```
class TreeNode:
    def __init__(self,key,val,parent=None,left=None,right=None, bal='EQ'):
        self.key = key
        self.payload = val
        self.balance = bal
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def rotateRight(self):
        newSelf = TreeNode(self.key, self.payload, self, self.leftChild.rightChild,
                           self.rightChild,self.balance)
        self.key = self.leftChild.key
        self.payload = self.leftChild.payload
        self.balance = self.leftChild.balance
        self.rightChild = newSelf
        self.leftChild = self.leftChild.leftChild
        if newSelf.leftChild:
            newSelf.leftChild.parent = newSelf
        if newSelf.rightChild:
            newSelf.rightChild.parent = newSelf
        if self.leftChild:
            self.leftChild.parent = self

    def put(self,key,val):
        if key < self.key:
            <**** CODE OMITTED -- TO BE COMPLETED TOMORROW AS LAB 9 ****>
        else: # key > self.key so add new key to right subtree
            if self.rightChild:
                tallerRightSubtree = self.rightChild.put(key,val)
                if tallerRightSubtree:          # Here we check to see if the subtree got taller
                    if self.balance == 'TL':
                        self.balance = 'EQ'
                        return False
                    elif self.balance == 'EQ':
                        self.balance = 'TR'
                        return True
                    else: # Two too tall on right now
                        if self.rightChild.balance == 'TR':  # only rotate left at pivot
                            self.balance = 'EQ'                  # preset balance factors to
                            self.rightChild.balance = 'EQ'       # be correct after pivoting
                        else: # need to rotate right at B, then rotate left at pivot
                            if self.rightChild.leftChild.balance == 'TR':
                                self.balance = 'TL'
                                self.rightChild.balance = 'EQ'
                            elif self.rightChild.leftChild.balance == 'TL':
                                self.balance = 'EQ'
                                self.rightChild.balance = 'TR'
                            else:
                                self.balance = 'EQ'
                                self.rightChild.balance = 'EQ'

                            self.rightChild.leftChild.balance = 'EQ'
                            self.rightChild.rotateRight()
                        self.rotateLeft()
                        return False
            else:  # base case in search -- add new node as a leaf
                self.rightChild = TreeNode(key,val,self)
                if self.balance == 'EQ':
                    self.balance = 'TR'
                    return True
                else:
                    self.balance = 'EQ'
                    return False
```

a) Where in the above code do the balance factors get set for your answer to question 4 (a)?

Team #:_____ Name:_____
Absent:

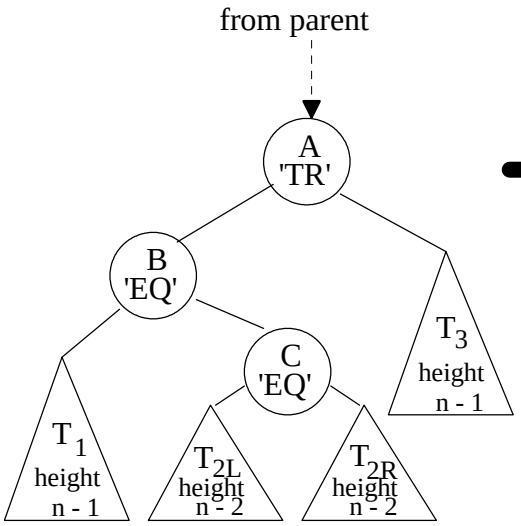
6. Complete the below figure which is a "mirror image" to the figure in question 3, i.e., inserting into the pivot's left child's left subtree. Include correct balance factors after the rotation.



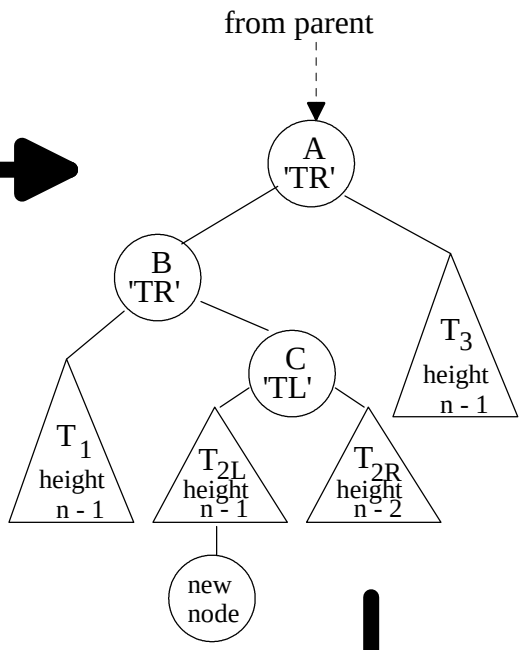
Team #:_____ Name:_____
Absent:

7. Complete the below figure which is a "mirror image" to the figure in question 4, i.e., inserting into the pivot's left child's right subtree. Include correct balance factors after the rotation.

Before the insertion:



After the insertion, but before first rotation:



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

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