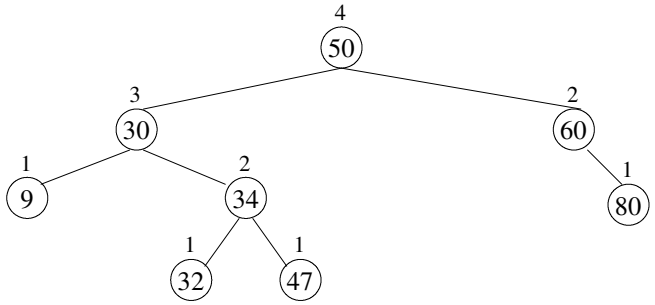
Data Structures Name:_____

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 $\Theta(\log_2 n)$. 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 value it contains.

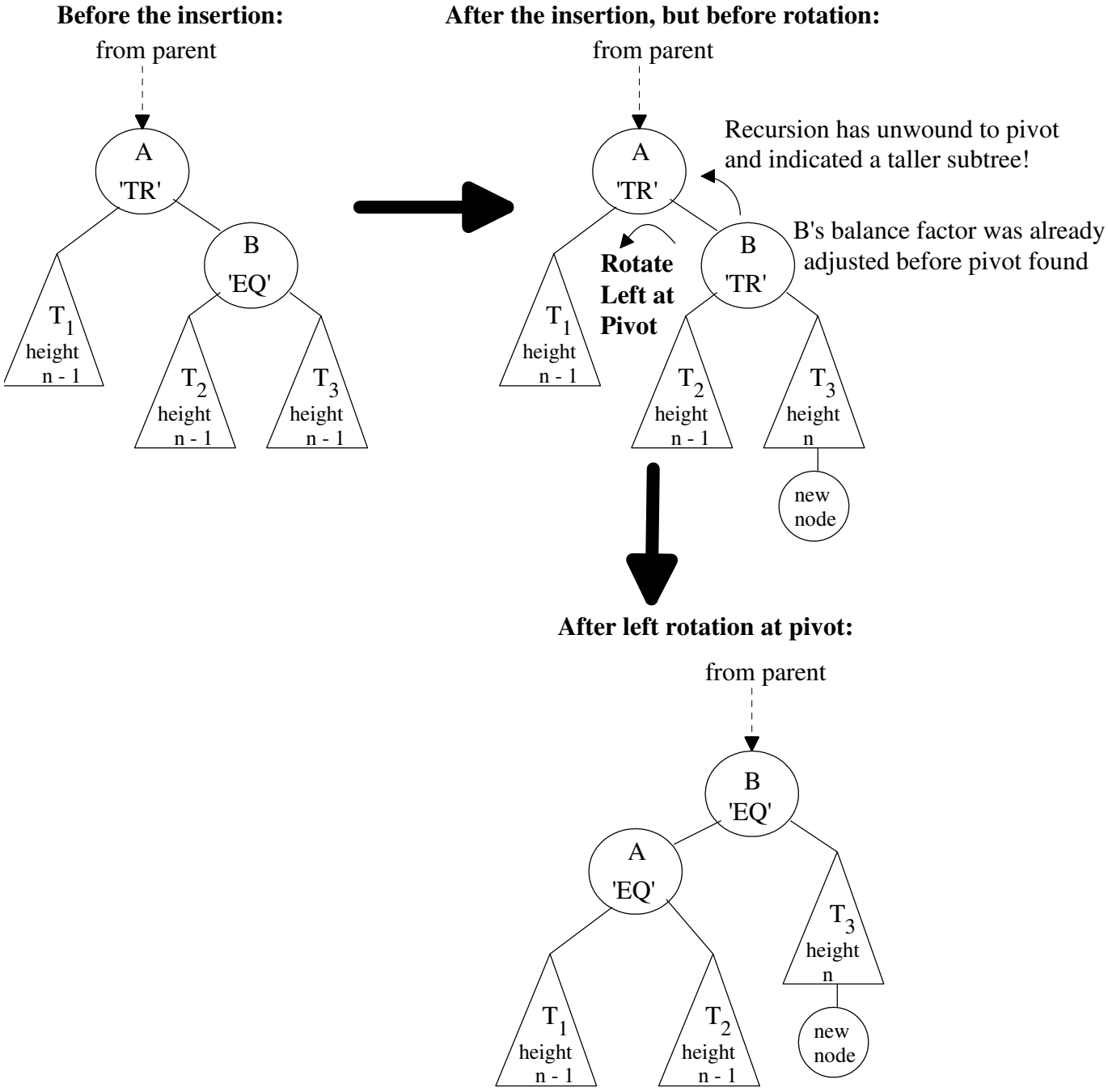
2. We'll add new nodes to the AVL as leaves just like we did for Binary Search Trees (BSTs).
a) Add the value 90 to the tree?

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? (Draw the tree resulting tree below)

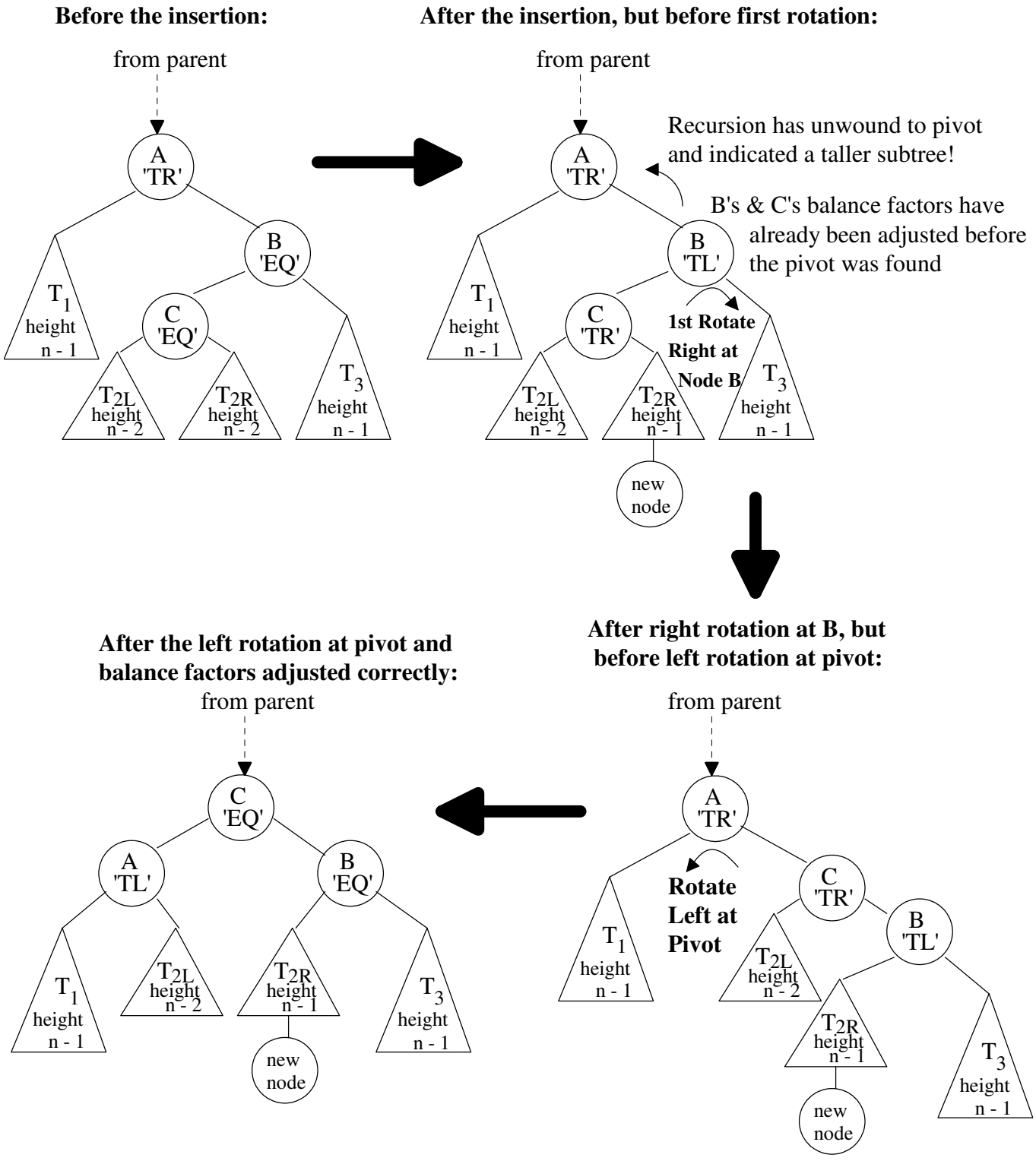
3. Typically, an insertion of a new value into an AVL requires the following steps:
- compare the new value with the current tree node's value (as we did in the `insert` method in the BST) to determine whether to recursively `insert` the new value into the left or right subtree
- add the new value as 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 will modify the `insert` method so that it returns a new parameter to indicate if the subtree got taller or not.
- 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.

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 node A have rebalanced the AVL tree?

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?

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

```
// IntAVLTree.h   Specification file for the IntBinaryTree class
#ifndef INTBINARYTREE_H
#define INTBINARYTREE_H

class IntBinaryTree
{
private:
      enum BalanceFactor { TL, EQ, TR };

   struct TreeNode
   {
      int value;          // The value in the node
        BalanceFactor balance;  // The balance factor of the node
      TreeNode *left;     // Pointer to left child node
      TreeNode *right;    // Pointer to right child node
   };

   TreeNode *root;        // Pointer to the root node
...
```

Partial code for the AVL tree insert in IntAVLTree.cpp:

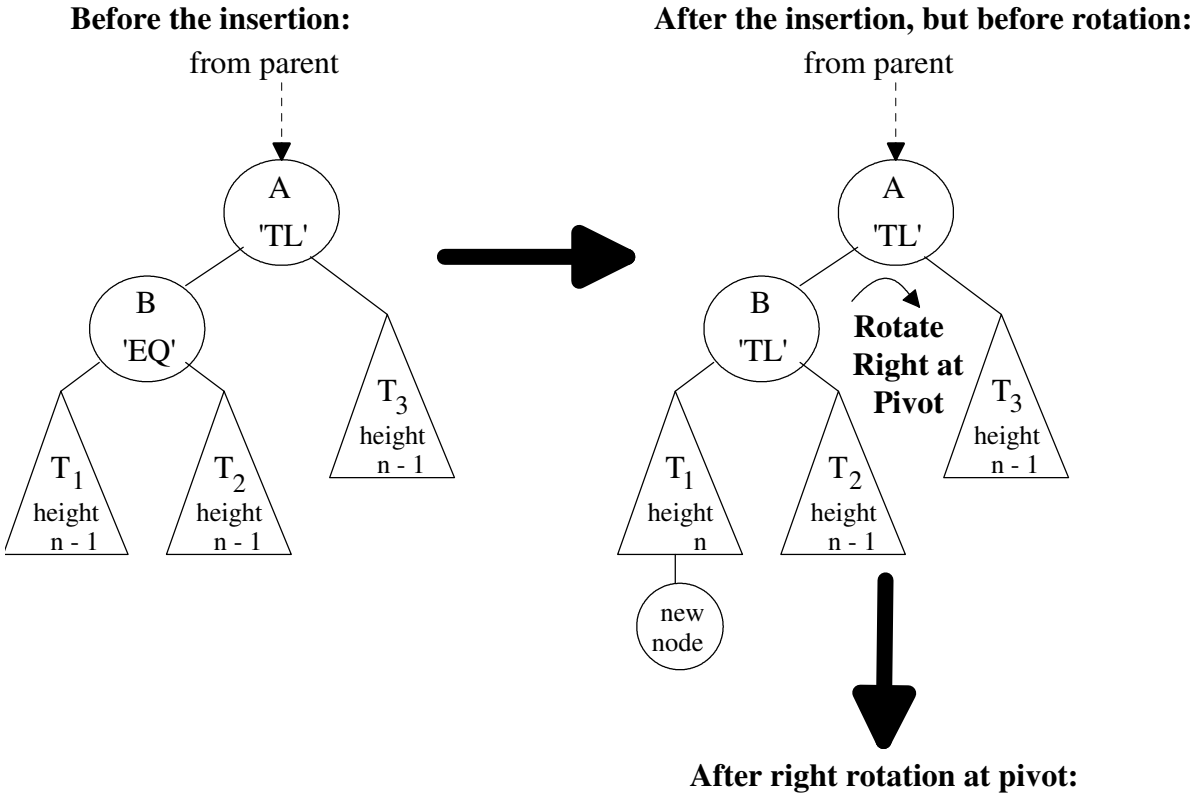
```
void IntBinaryTree::insert(TreeNode *&nodePtr, TreeNode *&newNode, bool & taller)
{
      bool tallerLeftSubtree, tallerRightSubtree;

      taller = false;   // Assume subtree does not grow unless we find out otherwise
      if (nodePtr == NULL) {
            nodePtr = newNode;                          // Insert the node.
            taller = true;
      } else if (newNode->value < nodePtr->value) {// insert into left subtree

            // CODE OMITTED HERE --- TO BE COMPLETED IN LAB TOMORROW

      } else {  // insert into right subtree
            insert(nodePtr->right, newNode, tallerRightSubtree);
            if (tallerRightSubtree) {
                  if (nodePtr->balance == TR) {
                        // Need rotation(s) to restore AVL height-balance property
                        rebalanceRightSubtree(nodePtr);
                  } else if (nodePtr->balance == EQ) {
                        nodePtr->balance = TR;
                        taller = true;
                  } else {
                        nodePtr->balance = EQ;
                  } // end if
            } // end if (tallerRightSubtree)
      } // end if
} // end insert

// Performs a right (clockwise) rotation about the "oldSubtreeRoot"
void IntBinaryTree::rotateRight(TreeNode *& oldSubtreeRoot)
{
      cout << "Rotating Right about: " << oldSubtreeRoot->value << endl;
      TreeNode * newSubtreeRoot = oldSubtreeRoot->left;
      oldSubtreeRoot->left = newSubtreeRoot->right;
      newSubtreeRoot->right = oldSubtreeRoot;
      oldSubtreeRoot = newSubtreeRoot;
} // end rotateRight
```

More partial code for the AVL tree insert in IntAVLTree.cpp:

```
// Rebalances the right subtree after the longer right subtree of the pivot
// got even longer with the newly inserted node
void IntBinaryTree::rebalanceRightSubtree(TreeNode *& pivot)
{
      TreeNode * pivotsRightChild = pivot->right;
      // Note:  The balance factor of all nodes below the pivot node have
      // already been adjusted to account for the new node!!!
      if (pivotsRightChild->balance == TR) { // RR case
            // Pre-set the balance factors to reflect the correct values after
            // the rotation in performed.
            pivot->balance = EQ;
            pivotsRightChild->balance = EQ;
            rotateLeft(pivot);
      } else if (pivotsRightChild->balance == EQ) { // an impossible case
            assert(pivotsRightChild->balance != EQ);
      } else { //pivotsRightChild->balance == TL  -- the RL cases
            TreeNode * leftChildOfPivotsRightChild = pivotsRightChild->left;
            // Pre-set the balance factors to reflect the correct values after
            // the rotations are performed.
            if (leftChildOfPivotsRightChild->balance == TR) { //RLR case
                  leftChildOfPivotsRightChild->balance = EQ;
                  pivotsRightChild->balance = EQ;
                  pivot->balance = TL;
            } else if (leftChildOfPivotsRightChild->balance == EQ) { // RLNew case
                  // the left child of the pivot's right child must be the new node
                  pivotsRightChild->balance = EQ;
                  pivot->balance = EQ;
            } else { // leftChildOfPivotsRightChild->balance == TL -- RLL case
                  leftChildOfPivotsRightChild->balance = EQ;
                  pivotsRightChild->balance = TR;
                  pivot->balance = EQ;
            } // end if
            rotateRight(pivot->right); // Uses "pivot->right" instead of
            // "pivotsRightChild" to change the pivot node's right pointer
            rotateLeft(pivot);
      } // end if
} // end rebalanceRightSubtree
```

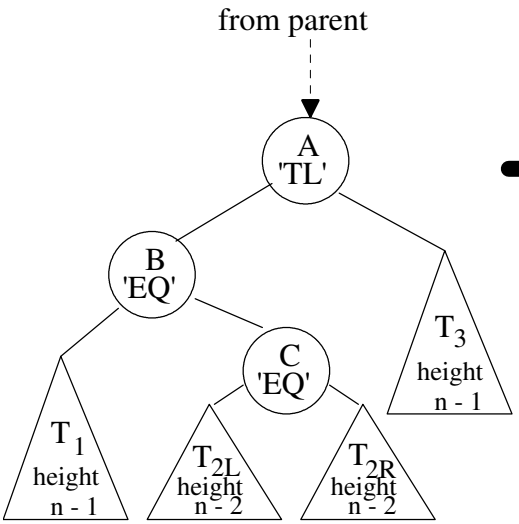
Where in the above code do the balance factors get set for your answer to question 4 (a)?

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.

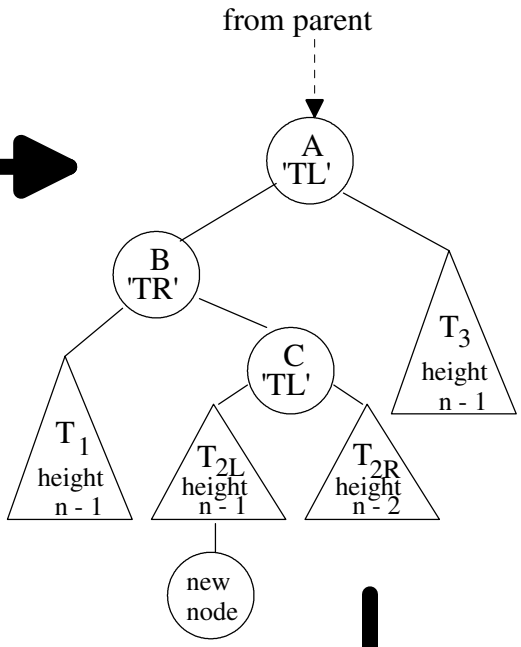


7. Complete the below figure which is a "mirror image" to the figure in question 4, i.e., inserting into the pivot's left child's right subtree. Include correct balance factors after the rotation.

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: