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 an AVL tree with $n$ nodes has a height no worse 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.

![AVL Tree Diagram]

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)
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 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:

```
Before the addition:

From parent

A
  /\  
 TR / \   
 B 'EQ'
  /\     
 T1  T2

After the addition, but before rotation:

From parent

A
  /\  
 TR / \   
 B 'EQ'
  /\     
 T1  T2

Recursion has unwound to pivot
and indicated a taller subtree!

Rotate Left at Pivot

B
  /\  
 'TR' / \   
 T1  T2

After left rotation at pivot:

From parent

B
  /\  
 'EQ'

A
  /\  
 'EQ'

T1

T2

T3

new node

new node
```

a) Assuming the same initial AVL tree (upper, left-hand of above diagram) if the new node would have increased the height of T2 (instead of T3), 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:**

Recursion has unwound to pivot and indicated a taller subtree!

B's & C's balance factors have already been adjusted before the pivot was found

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

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

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 of \( T_{2R} \), 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?
Consider the BinaryTreeAVL class that inherits and extends the BinaryTree class to include balance factors.

```python
from binarytree import *

class BinaryTreeAVL(BinaryTree):
    def __init__(self, item, balance = 'EQ'):
        BinaryTree.__init__(self, item)
        self._balance = balance

    def getBalance(self):
        return self._balance

    def setBalance(self, newBalance):
        self._balance = newBalance

    def __str__(self):
        """Returns a string representation of the tree rotated 90 degrees to the left."""
        def strHelper(tree, level):
            result ="
            if not tree.isEmpty():
                result += strHelper(tree.getRight(), level + 1)
                result += "| " * level
                result += str(tree.getRoot())+ " : " + tree.getBalance() + "\n"
                result += strHelper(tree.getLeft(), level + 1)
            return result
        return strHelper(self, 0)

Now let's consider the partial AVL class code that inherits from the BST class:

class AVL(BST):
    def __init__(self):
        BST.__init__(self)

    def add(self, newItem):
        """Adds newItem to the tree."""

        # Helper function to search for item's position
        def addHelper(tree):
            # Helper function to search for item's position
            def rotateLeft(tree):
                newTree = BinaryTreeAVL(tree.getRoot())
                newTree.setLeft(tree.getLeft())
                newTree.setRight(tree.getRight().getLeft())
                newTree.setBalance(tree.getBalance())
                tree.setRoot(tree.getRight().getRoot())
                tree.setBalance(tree.getRight().getBalance())
                tree.setLeft(newTree)
                tree.setRight(tree.getRight().getRight())

            def rotateRight(tree):
                ### ADD CODE HERE FOR rotateRight ###
```
c) Where in the above code do the balance factors get set for your answer to question (b)?
d) 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.

**Before the insertion:**

![Before the insertion diagram]

**After the insertion, but before rotation:**

![After the insertion, but before rotation diagram]

**After right rotation at pivot:**

![After rotation diagram]
3. a) 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.

**Before the insertion:**

![Before insertion diagram]

**After the insertion, but before first rotation:**

![After insertion diagram]

**After the right rotation at pivot and balance factors adjusted correctly:**

![After right rotation diagram]

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

![After left rotation diagram]

In lab you will complete and test the `add` method code including `rotateRight`. 