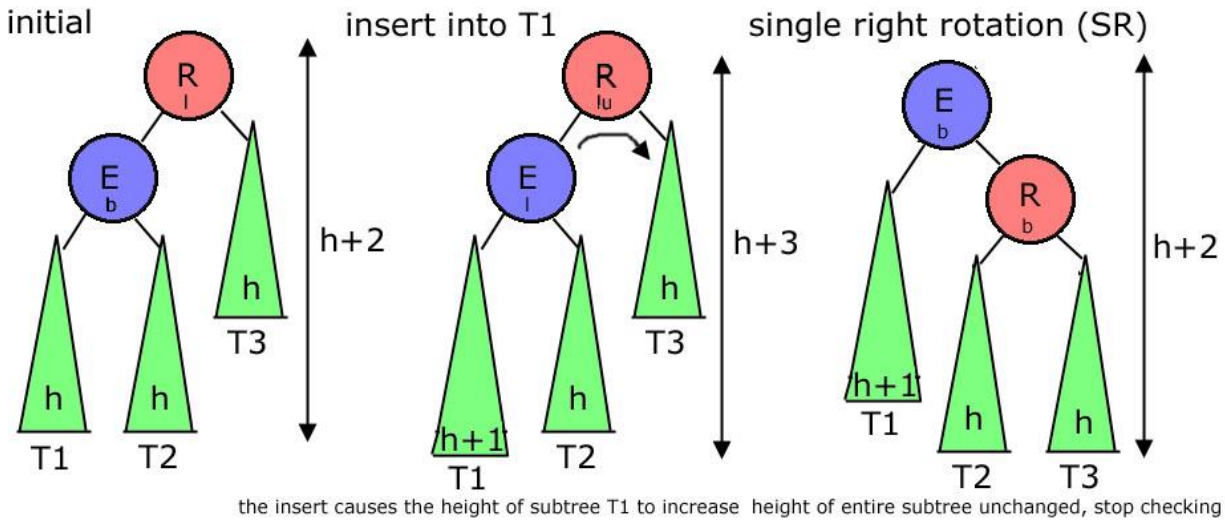
Laboratory 9: Method Cohesion and Implementation Inheritance Coupling

Java API
KeyedItem, CD, Song Documentation
FileIO Documentation

An AVLTree "is-a" BinarySearchTree that is also balanced. The given AVLTree implementation has several elegance problems. You will improve the AVLTree implementation.



Lab:

- More Class Invariants
- Method Cohesion
- More Method Decomposition

Part I: More Class Invariants & Assert Statements

Add public and protected methods (and possibly instance variables) to BinarySearchTree to check additional class invariants of the binary search tree. **Use the assert statement in appropriate places in BinarySearchTree (insert & delete) to call these methods.** Use good error messages for assertion exceptions. In particular, indicate the item or searchkey that was being processed when the assertion exception occurred.

- boolean isBalanced() //recursive and requires a method to compute the height of a subtree, see below

Note: May need convenience methods as well, where you will pass in the root node by default

Complete the following code to determine the height of the BST:

```
public int height() //convenience method passing in root node
{
   return getHeight(getRootNode());
}

protected int getHeight(TreeNode tNode) //recursive method
{
   if (tNode == null)
   {
      return 0; //no items in tree
   }

   int height;
   int leftHeight = ??   //get the height of tNode's left child
   int rightHeight = ??  //get the height of tNode's right child

   if (leftHeight >= rightHeight)
   {
      height = ??  //increment leftHeight
   }
   else
   {
      height = ??  //increment rightHeight
   }

   return height;
}
```

To determine whether the BST is balanced, the left height and the right height cannot be different by more than one. Also, every subtree in the BST must be balanced for the entire BST to be balanced. *Note that the recursive getHeight and isBalanced methods can be combined into a single recursive method with an int return type where -1 indicates not balanced.*

Part II: Method Cohesion

- BinarySearchTree.java
- AVLTree.java
- AVLTreeNode.java

Add assert statements in AVLTree methods to check for correct balancing of AVLTree. Your assert statements will call the isBalanced() method that you wrote in the last section. Using your driver, verify that the AVLTree is balanced after any operation that modifies the tree has completed. That is, the assert statements should never indicate a problem. Remember to use the -ea option to activate the assert statements.

Identify the two methods in AVLTree which are not cohesive (hint: look at the AVLfix methods). Correct this by writing new methods. This is "Replace Parameter with Explicit Methods" refactoring. **Split the methods according**

Part III: More Method Decomposition

Use the short methods that you created in the previous lab to simplify the code in AVLTree. By overriding the appropriate short methods in BinarySearchTree (but still calling super as appropriate), you should be able to **eliminate the insertItem method in AVLTree**. It is possible to also eliminate deleteItem, but this is not required.

You will find that there is a lot of simplification/cleanup possible in the long methods that maintain the balance of the tree. This is "Extract Method" refactoring. Using the provided rotation methods, make the overall code as readable as possible (specifically in your AVLFix methods, where rotations are performed).

Using your driver, apply unit testing as you work. That is, thoroughly test your modified AVLTree class after each modification. For example, after working on the single left rotation method, create a test case that will test this rotation. You are likely to see the advantages of using **assert** to check class invariants at this point.

Only one submission per team is necessary, but please make sure to include both names at the top of your source code, as well as in the comments section when submitting the lab, so both people can get credit.