Laboratory 8: Class Invariants (Assert Statements), Method Decomposition, & Javadoc comments

Java API
KeyedItem, CD, Song Documentation
FileIO Documentation

Using the BinarySearchTree implementation from the previous lab, you will investigate class invariants and method decomposition. **A class invariant** is an assertion that captures the properties and relationships, which remain stable throughout the life-time of instances of the class. You will **assert** Boolean methods, and if they return false, an exception is automatically thrown. **Method decomposition** is essentially the process of breaking down large methods into smaller methods that are easier to reuse, test, understand, and debug.

Lab:

- Class Invariants/Assert
  - Binary Search Tree Property
  - Number of Items Property
- Method Decomposition
  - Javadoc Documentation
  - Unit Testing

Part I: Class Invariants and Assert

Download the following files:

- SearchTreeInterface.java //a size() method has been added
- TreeException.java
- TreeIterator.java
- TreeNode.java
- BinaryTreeBasis.java
- BinarySearchTree.java
- BinaryTreeIterator.java
- BSTDriver.java
- FileIO.class
- FileIOException.class
- Song.class
- CD.class //the CD title is the search key
- KeyedItem.class
- queue.jar
- cds.txt

Add public and protected methods (and possibly instance variables) to BinarySearchTree to check the 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: ex. assert validateBSTProperty() : "not a binary search tree after insertion of key: " + item.getKey();

- boolean validateBSTProperty() // use an iterator and setInorder() to step through the tree, cast items pulled from the tree to KeyedItems and then to type Comparable (via the getKey() method), and finally compare the items using compareTo(). Each subsequent item should be greater in value than the previous item to satisfy the BST property.
- boolean validateSize() //add a size instance variable to BST and update it appropriately (look at insert and delete methods) then compare this size to a method that manually computes the size of the BST using an iterator.

Part II: Testing with BSTDriver

Complete BSTDriver to thoroughly test all of the public methods in SearchTreeInterface. That is, add, remove, and retrieve cds in an arbitrary fashion**. Don't simply add them all in and then remove them all, although doing this to get your testing started is fine**. Configure test cases to make sure that your method to determine whether the BST is balanced or not is working correctly. Intentionally insert errors in BST to observe how the assert statements operate. Use the -ea option in the command line to enable assertions when your program runs. Your driver class should be lengthy. Add simple comments to your driver to explain your tests.

Part III: Method Decomposition

Create new methods in BinarySearchTree. Use the insertItem method as a model (see below).

```
if (comparison == 0)
{
   tNode = method call  //one line of code (insertDuplicate method)
}
else if (comparison < 0)
{
   tNode = method call  //one line of code (insertLeft method)
}
else
{
   tNode = method call  //one line of code (insertRight method)
}
```

That is, the **insertItem, and deleteItem, and retrieveItem** methods should call other methods for each possible outcome in the conditionals, so these methods dispatch based on the search key comparison result. After method decomposition, each conditional outcome should be a single line of code. Look for other places in the code for possible method decomposition. As you will extend BinarySearchTree in a future lab, set the visibility modifier for these methods to protected.

Note: Although this process does make the methods in BST shorter and more readable, as you will see in a future lab, overdoing method decomposition can sometimes result in poor software design.

Every time you make a change, run your driver to make sure that your output is the same as at the end of the previous part.

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.