Laboratory 9: Class Invariants

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. 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
  - Balanced Tree Property
  - Number of Items Property
- Method Decomposition
  - Javadoc Documentation
  - Unit Testing
- Method Comments (Contract)

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** (retrieve, insert, delete) to call these methods. Although a binary search tree doesn't have to be balanced, write a method to check for this as you will use it in the next lab. Use good error messages for assertion exceptions. In particular, indicate the item or searchkey that was being processed when the assertion exception occurred.

- boolean validateBSTProperty() //there is an easy way to do this (look at the iterator setInorder() method)
- 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
- boolean isBalanced() //recursive and requires a method to compute the height of a subtree, see below

Note: some methods 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()
{
   return getHeight(getRootNode());
}

protected int getHeight(TreeNode tNode) //recursive method
{
   if (tNode == null)
   {
      return ??
   }

   int height;
   int leftHeight = ??
   int rightHeight = ??
   if (leftHeight >= rightHeight)
   {
      height = ??
   }
   else
   {
      height = ??
   }

   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: Unit Testing

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 basic 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 the next 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 the next 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.

Part IV: Javadoc

Fully comment **all** (including protected methods) of your BST methods using preconditions, postconditions, and throws. Note that interface comments are automatically inserted into the class that implements the interface.

Example:

```
/**
 * Searches for the leaf insertion location for item. <br>
 * Precondition: item is not null
 * Postcondition: returns the current node so that it can be linked (or relinked) to its parent
 * depending on whether a left or a right was taken in obtaining the leaf insertion location.
 * Throws: TreeException if an attempt to insert a duplicate is detected.
 * Calls: insertLeft, insertRight, insertDuplicate depending on the item's sk
 */
protected TreeNode insertItem(TreeNode tNode, KeyedItem item) throws TreeException
{
```

<u>You are required to choose one partner for this lab from your lab section.</u> 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.

The Lab is due the following Sunday before midnight.