Lab 3: Tracing and Testing

12:00 PM, Feb 7, 2018

Contents

1 Testing, testing, 1 2 3
2 Making an Oracle
3 Testing the Oracle
4 Testing our BST Makers

Objectives

By the end of this lab, you will know:

- how to trace the flow of a program two different ways
- how to test methods with more than one right answer

By the end of this lab, you will be able to:

- use substitution and stack frame methods to trace your programs
- write oracle testing methods

1 Testing, testing, 1 2 3

So far in CS 18 (and throughout your CS experience at Brown), you’ve primarily written unit tests. This is a test for functions where one particular input can only produce one correct output, i.e. there are not multiple solutions.

In this part of the lab, you’ll learn how to test functions where there are many correct solutions. Let’s suppose, for example, that we are sorting a list of pairs of numbers by the first number:

{(2, 3), (2, 2), (5, 1), (0, 5)}

This could produce either of the following:

{(0, 5), (2, 3), (2, 2), (5, 1)}
{(0, 5), (2, 2), (2, 3), (5, 1)}
Both are correctly sorted! So using the testing methods we’ve worked with so far, you could (wrongly!) fail the test. Let’s say our sorter gives us the first list, but our tester checks if the given list was the second list. If this is this case, then we would fail the test even though we had a correct answer. We need to change how we think about testing to account for multiple solutions.

In comes oracle testing. Oracle testing, rather than comparing your method’s output with one correct solution, checks certain characteristics of your method’s output and ensures they all hold true.

To create an oracle, we first need to know what characteristics make for a correct output. You should try to find the minimum number of characteristics, without allowing for any wrong output to fulfill those. Let’s take our sorting example. The two characteristics of a correct output are:

- The output list contains exactly the same elements as the input list (including the correct count if there are duplicate elements).
- The output list is sorted according to our sorting scheme.

If these characteristics hold, then we have a correct output, and if either one does not hold, our output is wrong by definition.

For this lab, you will be writing an oracle to test whether a program that claims to generates binary search trees (BSTs) from a list of numbers produces a valid answer. We have provided you with two implementations (in different classes) of a method called `makeBST`. The method takes a list of Integers and produces a binary search tree of Integers. You will write a function that takes the original list and the output of `makeBST` and produces a boolean indicating whether the output is a valid answer.

**Task:** Write down (on paper or in a text editor) the characteristics of a correct output, i.e. a correct BST created from an input list.

**Hint:** Feel free to look up the definition of a BST on Wikipedia to refresh yourself on its invariants!

| You’ve reached a checkpoint! Please call over a lab TA to review your work. |

## 2 Making an Oracle

Now that you know what constitutes a correct output, it’s time to write a method to check each of these conditions. Each method should take in any parameters it needs to test its particular characteristic, and return a boolean indicating if the characteristic holds true. For example, if we were creating methods to test our sorting method, we would have:

```java
public static boolean testSameElements(List<Integer> original, List<Integer> sorted) {
    //code goes here
    return isSame;
}
```

We would also have a test to see if a list is correctly sorted:
Task: At this time, go ahead and copy over our source code as follows:

cp /course/cs0180/src/lab03/src/* ~/course/cs0180/workspace/javaproject/src/lab03/src

Task: We have also provided you with all of the solution files you’ll be modifying today, so go ahead and copy those over as well, as follows:

cp /course/cs0180/sol/lab03/sol/* ~/course/cs0180/workspace/javaproject/sol/lab03/sol

Briefly, we’ll explain the class hierarchy we’ve created for you. In the source folder, you have IBSTMaker, an interface for a BST maker, which has the method makeBST. Implementing this interface are BSTMaker1 and BSTMaker2.

In the solution folder, you have IBinTree, an interface for a generic binary tree. This interface is implemented by BTNNode, a node class, and Leaf, a leaf class. Check out the IBinTree interface to see what methods those classes support, as you may find them useful.

Also in the solution folder, we have a class Oracle, containing the method sameElems, which takes in an IBinTree and a list of Integers and determines if they contain exactly the same elements (including correct duplicate counts). We also have provided a stub for the bstOracle method, which you’ll be writing in this next part!

That’s all you’ll need to know, for now!

Inside Oracle, we have already provided to you the sameElems method to determine if a BST containing the correct elements has been created. It’ll be your job to implement a method isBST, which takes in an IBinTree and returns a boolean indicating if it is a BST according to your written requirements from the last task.

This task seems really large! Never fear: we’ll help you break it down. First, you’ll need a method that can determine if every value to the left of the root is less than the root value. However, the root is of type IBinTree, so you’ll need to add this method to the IBinTree interface! This means you’ll need to implement this for both Leaf and BTNNode.

Task: Write a method allLess, which takes in an Integer and returns a boolean indicating if all the values in the IBinTree are less than that integer.

Task: Similarly, write a method allGreaterEq, which takes in an Integer and returns a boolean indicating if all the values in the IBinTree are greater than or equal to that integer.

Hint: You’ll need to add both of these methods to the IBinTree interface!

However, this is not enough! We need to check that for every node in the tree, (a) the value in the node is greater than every value in the left subtree, (b) the value in the node is less than or equal to every value in the right subtree, and (c) both the left and right subtrees are also BSTs.

Task: Write a new method, isBST, which checks that everything to the left of the IBinTree is less than the node’s data value, and everything to its right is greater than or equal to the node’s data value. Then, this method should ensure that both of the child nodes of the IBinTree are also BSTs.

Hint: Use the methods you just wrote, and some recursive calls!

Hint: Don’t forget to add this method to the interface as well!
**Task:** Now, inside Oracle, create fill in the method called bstOracle, which takes in a List<Integer> and the IBinTree created from the list, and calls both isBST and sameElems with the necessary arguments. oracle should return a boolean indicating if both tests passed.

| You’ve reached a checkpoint! Please call over a lab TA to review your work.

### 3 Testing the Oracle

Now, you should have your isBST and oracle methods written. However, we aren’t quite ready to test our BST implementations with this oracle. First, we have to make sure the oracle itself works!

**Task:** In the BinTreeTest class we’ve provided, write the checkExpect for each test method to make sure that various binary trees are (or are not) indeed BSTs. Our class provides several IBinTrees that you can use for testing, or you can create your own. Note that not all of our provided trees are actually BSTs; be sure to write the checkExpect correctly to reflect this!

**Hint:** If you want to create your own test trees, check out the BTN ode constructors to see your options!

| You’ve reached a checkpoint! Please call over a lab TA to review your work.

### 4 Testing our BST Makers

Now that you have your working oracle, it’s time to use it to check whether the BSTMaker implementations we provided are buggy.

**Task:** On a piece of paper, write down the various situations and edge cases that you think you should test. Remember that the input to the makeBST methods is a list of numbers, so your “edge cases” here should be various kinds of lists that you think make for a good collection of tests.

**Task:** In the BSTMakerTest class we provided, create checkExpect tests for several of your edge cases. Your goal should be to write enough tests to determine whether each of the two makeBST methods produces valid BSTs (if one does not, can you detect what is wrong with it?). Some tests should check each of the two makeBST implementations.

**Hint:** Look at our example test! You can add an optional String argument to your checkExpect to name your tests so that when they fail, you can tell which test it was.

| Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)