Lab 3: Loops and Testing
12:00 PM, Feb 5, 2020

Contents

1 Test Setup Methods 1
2 Testing, testing, 1 2 3 3
3 Making an Oracle 4
4 isBST() and toList() 5
5 Testing the Oracle 6
6 Testing our BST Makers 6

Objectives

By the end of this lab, you will know:

- how to use loops to do cool things
- how and when to use setup methods in testing

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

- write oracle testing methods

1 Test Setup Methods

Task: Before anything, copy over the src and sol files provided:

```bash
cp /course/cs0180/sol/lab03b/sol/* ~/course/cs0180/workspace/javaproject/sol/lab03/sol
cp /course/cs0180/src/lab03b/src/* ~/course/cs0180/workspace/javaproject/src/lab03/src
```

In CS18, we’ve seen how some of our methods are designed to update values, rather than to return computed answers. Methods that update values pose particular challenges for testing. Why? Consider the following two tests for the methods you have just implemented:
public class CartTester {
    Catalog testCat = new Catalog();
    Item radio = new Item("radio", 10.95);
    Item hat = new Item("green witch hat", 8.50);

    public void setUpCat(){
        testCat.addItem(radio);
        testCat.addItem(hat);
    }

    public boolean testGetPrice(Tester t) {
        t.checkExpect(testCat.lookupPrice(radio), 10.95);
    }

    public void testRemItem(Tester t) {
        testCat.addItem(radio);
        testCat.addItem(hat);
        testCat.remItem(radio);
        t.checkExpect(testCat.findMatches(radio), new LinkedList<Item>());
    }

    public static void main(String[] args) {
        tester.run(new CartTester());
    }
}

Will both of these tests pass, or not? If testGetPrice runs first, both tests should pass. But if testRemItem runs first, then testGetPrice will fail because there is no longer a radio in the catalog. This example points out something critical about mutation and how it impacts testing:

Once your program allow mutation, two identical calls to the same method (or function) may yield different results.

This cannot happen when you program functionally. It’s a new complexity that arises when methods or functions update values. This has significant impact on testing, because it implies that the order in which tests run changes the result of testing. This is a serious problem. Most testing frameworks don’t guarantee the order in which they will run your tests (precisely to force you to keep tests independent of each other). This makes sense, because in multi-programmer projects, people should be free to write tests without worrying about interference from tests written by others.

To prevent problems, we have to adapt how we write tests to be immune from such impacts. In particular, our tests need to perform Setup to make sure they start from a clean, predictable state.

Let’s change our CartTester class to have a method that creates a catalog with our two items:

```java
public class CartTester {

    public static Catalog initCatalog() {
        Catalog c = new Catalog();
        c.addItem(radio);
        c.addItem(hat);
        return c;
    }

    public CartTester() {
        testCat = initCatalog();
    }

    public void setUpCat() {
        testCat.addItem(radio);
        testCat.addItem(hat);
    }

    public boolean testGetPrice(Tester t) {
        t.checkExpect(testCat.lookupPrice(radio), 10.95);
    }

    public void testRemItem(Tester t) {
        testCat.addItem(radio);
        testCat.addItem(hat);
        testCat.remItem(radio);
        t.checkExpect(testCat.findMatches(radio), new LinkedList<Item>());
    }

    public static void main(String[] args) {
        tester.run(new CartTester());
    }
}
```
Now, inside each test method, we first use `initCatalog` to make sure we know the contents of our testing data:

```java
public static boolean testGetPrice(Tester t) {
    cat = this.initCatalog();
    return t.checkExpect(cat.lookupPrice(radio), 10.95);
}
```

```java
public static void testRemItem(Tester t) {
    cat = this.initCatalog();
    cat.remItem(radio);
    return t.checkExpect(cat.findMatches(radio), new LinkedList<Item>());
}
```

Note here that we don’t use the class-wide `testCat` catalog in testing – we use the fresh catalog that we set up for the tests in that method.

What if we want to test the effect of multiple methods though? Just call them all in the same method. For example:

```java
public static boolean testAddItem(Tester t) {
    newCat = new Catalog();
    newCat.addItem(radio);
    return t.checkExpect(cat.lookupPrice(radio), 10.95);
}
```

You can even put multiple `checkExpect`s in the same test method, as long as they build on the same initial data. The order of statements within a method is fixed (top to bottom), so there is no need to worry about these tests running out of order.

**Task:** Write tests for the Catalog class, using setup methods as needed. In particular, you should have some test methods that would otherwise interfere with each other if they all worked off of the same class-wide catalog.

**Task:** In general, pairs of methods that can affect each others’ output should be tested together (such as `addItem` and `lookupPrice`). What combinations of methods do you think need to be checked to make sure method interactions are as expected?

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

## 2 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:
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 the 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.

### 3 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:

```java
public static boolean testSorted(List<Integer> sorted) {
    //code goes here
    return isSorted;
}
```
Now, 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.

Additionally, there are several BST-based classes, similar to the structure you implemented in lab02: an interface IBST, a Node abstract class with subclasses Node1 and Node2, and a Leaf abstract class with subclasses Leaf1 and Leaf2. BSTMaker1 makes trees of type Node1 and Leaf1, and BSTMaker2 makes trees of type Node2 and Leaf2.

Also in the solution folder, we have a class Oracle, containing the method sameElems, which takes in an IBST 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 soon! First, though, we must write two procedures: isBST and toList.

4 isBST() and toList()

The isBST method should return true if the tree is a valid binary search tree and false otherwise. As a reminder, a tree is a valid binary search tree if, for each node in the tree, all nodes in its left subtree have values less than the node’s value and all nodes in its right subtree have values greater than or equal to the node’s value.

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 IBST, so you’ll need to add this method to the IBST interface! This means you’ll need to implement this for both Leaf and Node.

Task: Write a method allLess, which takes in an Integer and returns a boolean indicating if all the values in the IBST 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 IBST are greater than or equal to that integer.

Hint: You’ll need to add both of these methods to the IBST 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 IBST 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 IBST 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: Write a toList method that takes in nothing and outputs a List<Integer> that is a list of all of the integers in the BST.

Hint: Here’s how you can make an empty list:
List<Integer> myList = new LinkedList<Integer>();

Hint: Some helpful methods to look up are: add(), addAll(). Here’s how you can use them:

- myList.add(5); //adds one element to myList
- myList.addAll(anotherList); //adds all of the elements of a list to myList

Consult the documentation if you’d like more details.
**Hint:** It may be useful to use a helper function that can create a string representation of your list for testing purposes!

The `toList` method should return a list containing the elements contained in the BST. The order of these elements does not matter. However, duplicates do matter; each element should appear in the list the same number of times as it does in the tree.

**Task:** Now, inside `Oracle`, fill in the method called `bstOracle`, which takes in a `List<Integer>` and the `IBST` created from the list, and calls both `isBST` and `sameElems` with the necessary arguments. `bstOracle` should return a boolean indicating if both tests passed.

---

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

---

### 5 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 `OracleTest` 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 `IBST`s that you can use for testing. Note that not all of our provided trees are actually BSTs; be sure to write the `checkExpect` correctly to reflect this!

### 6 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?). Your tests should check both of the `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: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)