Homework 4 - Testing Multiple Correct Solutions

Due Apr 16, 2019 by 5pm **Points** 0

Before you begin, please make sure you are familiar with the guidelines discussed on the Expectations on Homework page and the "Homework Assignments" section of the CS 2102 FAQ Page.

If you're still feeling lost after reading through this assignment, try reading the Homework 4 Clarifications page and reviewing the Homework 4 Slides (https://canvas.wpi.edu/courses/12951/files/1906388/download?download_frd=1) .

Assignment Goals

- To become familiar with the heap data structure
- To be able to design test cases for methods that have more than one correct solution

Reminders

- **Please use the default package in your Java project.** There will be a penalty for using a named package.
- Please include a Javadoc statemnt above each method. There will be a penalty for forgetting your Javadoc statements.
- In your test cases, please use descriptive names for your test case methods and/or comments above each test case explaining what the method tests. This will help the TAs and SAs in grading. There will be a penalty for forgetting this step.

Files

Here are the starter files referenced in the assignment description:

- IBinTree.java (https://canvas.wpi.edu/courses/12951/files/1898682/download?download_frd=1)
- IHeap.java (https://canvas.wpi.edu/courses/12951/files/1898683/download?download_frd=1)
- TestHeap.java (https://canvas.wpi.edu/courses/12951/files/1898684/download?download_frd=1)

Problem Statement

Our discussion of data structures has introduced a couple of methods that could return one of several answers: `remElt()` on BSTs has this property, as do the `addElt()` and `remMinElt()` functions on heaps.

Writing test cases for such methods clearly requires more than writing a specific, concrete answer (as you are used to doing). Instead of writing a concrete answer, you have to write a method that checks whether your answer meets the requirements of a correct answer and run that method within your test case. In other words, rather than checking whether you got *the* correct answer, you have to check whether you got *a* correct answer. This assignment is teaching you how to write tests for situations when you need "*a* correct answer".

Your job here is to write two methods, the first of which is `addEltTester()`. `addEltTester()` takes a heap, an integer, and a binary tree (in that order) and returns true if the binary tree is a valid result of adding the given integer to the first Heap. In other words, you want to fill in

```
boolean addEltTester(IHeap hOrig, int elt, IBinTree hAdded) {

  ...code to compare hOrig and hAdded around the addition of elt as appropriate...

}
```

You can use this to test a given `addElt()` implementation by writing a test such as

```
@Test
public void test1(){

  assertTrue(addEltTester(myHeap,5,myHeap.addElt(5)));

}
```

Or, you can provide the binary tree "manually", as in

```
@Test
public void test2(){

  assertTrue(addEltTester(myHeap,5,myBinTree));

}
```

where `myHeap` and `myBinTree` are data that you defined. Note that every heap is also a binary tree, so you can use `addElt()` to generate binary trees from heaps.

Also provide `remMinEltTester`, which is similar (but tests `remMinElt`). For this, fill in

```
boolean remMinEltTester(IHeap hOrig, IBinTree hRemoved) {

  ...code to compare hOrig and hRemoved as appropriate...

}
```

What should addEltTester() and remMinEltTester() actually do?

Each of these methods checks whether the binary tree is a valid result of performing `addElt()` or `remMinElt()` (respectively) on the original heap. A valid result is defined by the following conditions:

- The binary tree returned must satisfy the definition of a heap (smallest element on top, left and right subtrees both heaps).
- `addElt()` must retain all elements that were in the original heap, while adding only the new element one time.
- `remMinElt()` must retain all elements that were in the original heap, other than removing one occurrence of the smallest element.

You may assume that running `addElt()` or `remMinElt()` on a heap does not modify the heap (this lets you reuse the same heap data in multiple test cases).

For this assignment, heaps may have duplicate elements.

Logistics

Put both of these methods in a new class called `HeapChecker`.

You do not need to implement `addElt()` or `remMinElt()`. We are giving them to you, along with the other heap operations. Here are both **a binary tree implementation** **(https://canvas.wpi.edu/courses/12951/files/1898682/download?download_frd=1)** and **an initial correct heap implementation** **(https://canvas.wpi.edu/courses/12951/files/1898683/download?download_frd=1)** to use in writing your tester. You are getting both files because we build heaps as subclasses of binary trees.

If you need additional methods in `Heap` or `BinTree`, modify the classes in these files as needed with your additional methods. **Do not change** the names of the classes, any of the existing methods, or make your own subclasses (that will break the auto-tester). Just edit these classes and/or interfaces by adding your own additional methods.

DO NOT call addElt() in the method body of addEltTester() or remMinElt() in the method body of remMinEltTester()!!! Doing so undermines the purpose of this assignment.

DO NOT use type-checking, i.e. do not use any of the following:

- `instanceof`
- casting
- `getClass()`
- `static`

BIG HINT: Please consider modifying the interfaces we give you.

Testing the Testers

Put your test cases in an `Examples` class, as usual. Your test cases need only to use `addEltTester()`

and `remMinEltTester()`. They may use `addElt()`/`remMinElt()` if you wish (as shown in `test1` above). You should have at least 6 tests for each method.

Note that since your method is in the `HeapChecker` class, your `Examples` class will need a `HeapChecker` object. This means your test cases will look like the following:

```
class Examples {

  Examples(){}

  HeapChecker HT = new HeapChecker();

  ...

  @Test
  public void test3(){
      assertTrue(HT.addEltTester(myHeap, 5, myBinTree));
  }
}
```

where `myHeap` and `myBinTree` are data defined in your `Examples` class.

What makes for good tests?

Good tests will capture various ways that `addElt()` and `remMinElt()` might fail. Here's another way to think about it. If you have a good `addEltTester`, then your tests will all pass if you use a correct implementation of `addElt()`. If you use a broken implementation of `addElt()`, then ideally one of your test cases would fail (because the `addElt()` used would produce the wrong answer).

Put differently, assume you wanted to add 8 to the following heap:

```
  4
 /
5
```

Imagine that the `addElt()` code you were using returned

```
  8
 / \
4   5
```

(which is wrong because it isn't a heap [with min elts on top]). Then your `addEltTester` would return false, because the produced heap is not a valid answer when adding 8 to the original heap. A good set of test cases, then, will cover different elements to add to different configurations of heaps, looking for ways in which `addElt`/`remMinElt` might go wrong.

Checking bad implementations

This part is optional but will give you extra assurance that your testers have been written correctly.

If you want to see whether your tester is detecting bad implementations of the `Heap` operations, you can use **this Java file** **(https://canvas.wpi.edu/courses/12951/files/1898684/download?download_frd=1)** (put it in your directory). It contains several incorrect heap implementations (each a separate class that extends `DataHeap`). You can build a broken heap doing something like

```
IHeap badHeap =

   new TestHeap2(3, new TestHeap2(4, new MTHeap(),

                                     new MTHeap()),

                    new MTHeap());
```

(which replaces `DataHeap` with `TestHeap2` when creating the example). If you now used `badHeap` as an input to your `addEltTester`, some tests should fail where they would have passed had you made the same example with `DataHeap`.

The tests in your final `Examples` class should just be written using `DataHeap`, not these broken implementations. The broken implementations are just to help you in checking your work. Either delete or comment out any tests that used the `TestHeap` classes before you submit.

What to submit

Submit (via **InstructAssist** **(https://ia.wpi.edu/cs2102/)**) a single zip file (not tar, rar, 7zip, etc) containing all of your .java files that contain your classes, interfaces, and examples for this assignment. The name of the project in InstructAssist is **Homework 4**. Do not submit the .class files.

Grading

Homework 4 Grading Rubric **(https://canvas.wpi.edu/courses/12951/files/1898670/download?download_frd=1)**

Programs must compile in order to receive credit. Code that is commented out will not be graded.

You must use the class and interface names given in the assignment. You may add new methods to the given `BinTree` and `Heap` files, but you may not make any changes to the existing code in those files.

We will be looking for code that correctly implements `addEltTester` and `remMinEltTester`.