CPSC 221

List Testing (Part 2)

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."
--Martin Golding

Objectives

- Implement a test suite for classes that implement the `IndexedUnsortedList` interface

Tasks

In **part 1** of this homework, you created a test plan for classes that implement the **IndexedUnsortedList** interface. For part 2, you will implement about one third of your plan. This means you will add seven of the change scenarios outlined in your test plan (in addition to two of those given as examples) to class **ListTester**. The change scenarios are: 2, 3, 7, 10, 12, 14, 26.

You are strongly advised to follow the scenario and test naming conventions demonstrated in the given examples.

- Scenarios are named as StartingState_Change_ResultingState. For example, Scenario 6, which begins with list [A] and contains [B,A] after *addToFront(B)*, is identified as A_addToFrontB_BA.

- Test names begin with the associated scenario followed by the method being tested. For example, the test for *size()* after scenario 6 is named A_addToFrontB_BA_testSize.

Tests have already been written for change scenarios resulting in empty and single-element lists. You will need to complete the set of tests for scenarios resulting in a two-element list. Notice that there will be more tests for a larger list than for a smaller list. Follow the pattern demonstrated in the given tests, but make sure you are testing all appropriate positions in larger lists.

Running Your Test Class

Because this is your first test class, it is helpful to have something to test right away (although this is rarely possible in real-world situations). Two implementations of `IndexedUnsortedList` are provided to illustrate how your test class will handle implementations of different quality. Do not modify either of these classes. They do exactly what they are supposed to do.

- **GoodList** wraps the Java API `LinkedList` class in the `IndexedUnsortedList` interface and all of its list methods work correctly. However, the `Iterator`'s in the Java API lists do not throw `ConcurrentModificationExeption`'s as your lists will be required to do. When an implementation of an ADT does not pass tests for ADT-defined behaviors, do not modify tests to make the list look good. Tests should always reflect the ADT's defined behavior. It is the burden of the list to satisfy the test. `GoodList`, therefore, fails some of the tests for `Iterator` concurrency. We aren't going to edit Java API library code, so we'll have to live with those failures, for now.

- **BadList** has stubs - methods containing only enough code to compile and run - for each ADT method. We *expect* it to fail all tests. When you test it with the given `ListTester`, however, you will see that it passes some of the tests. This example serves as a reminder that one test is unlikely to reveal if a method is really working.

Files

To complete the homework, you will need these files, but you should not modify them:

- GoodList.java
- BadList.java
- IndexedUnsortedList.java

You will also need this file, which you will modify to add scenarios and related tests

- ListTester.java

Grading

Points will be awarded according to the following breakdown:

Tasks	Points
ListTester implements tests for lists up to 2 elements and adds the 7 change scenarios from your test plan to the given scenarios	10
GoodList passes all IndexedUnsortedList tests (though not Iterator concurrency tests)	5
BadList fails most tests	5

Required Files

You should submit the following:

- ListTester.java
- TeamEvaluation.docx