CPSC 221

List Testing (Part 1)

"I never make the same mistake twice. I make it five or six times, just to be sure."
--Anonymous

Objectives

- Create a test plan for a class that implements the `IndexedUnsortedList` interface
  - Identify change scenarios for an implementation of the `IndexedUnsortedList` interface
  - Define tests for those change scenarios to confirm correct functionality of an implementation of the `IndexedUnsortedList` interface

Background

An Abstract Data Type (ADT) is defined by an interface with a set of well-documented methods. When an ADT is implemented, users want some assurance that the implementation will perform as advertised.

Exhaustive testing is impossible and no amount of testing can guarantee that your code will be free of bugs, but a well-planned set of tests can provide reasonable assurance that your code works in most situations.

"Black Box" Testing

Black box testing analyzes the functionality of a class or program without assumptions or knowledge about its internal workings.

In other words, the actual implementation doesn't matter to black box tests. Only the results of calls to the public interface methods will be tested.

For example, if a class implements the `IndexedUnsortedList` interface, black box testing verifies the class has implemented all of the methods required by that interface and that these methods work as documented in a variety of scenarios.

Because black box test classes can be reused for any implementation of the same ADT, every test defined in this assignment can be used to test the `IUArrayList`,`IUSingleLinkedList`, and `IUDoubleLinkedList` classes, which you will create later in the semester.

Change Scenarios

A test suite is divided into a number of *change scenarios*, or *test cases*, each of which represents a change in the state of an ADT after a specific change method has been called.

A change scenario, then, is defined as:

- the starting (or previous) state of a data structure
- a change made to the data structure
- the resulting state of the data structure

Given a list class that implements the `IndexedUnsortedList` interface, a change scenario could be:

The starting state:
the list contains one element `A`
The change:
invoke the `addToFront(B)` method
The resulting state:
the list contains two elements, `B` and `A`, in that order.

where `A` and `B` are objects of type `T`, the generic type of elements that can be stored in the list.

Types of Change Scenarios

Because we can't possibly test for every eventuality, we focus on a reasonable set of scenarios that include *boundary* and *equivalence* cases.

- **Boundary cases** are scenarios where problems are most likely to appear and should be thoroughly tested. Scenarios beginning or ending with an empty list or single-element list are examples of boundary cases.

- **Equivalence cases** are scenarios that represent a range of similar situations. For example, a scenario resulting in a list with three

elements, which is the smallest possible list with front, middle and rear elements, might be considered equivalent to all lists with a beginning, middle, and end. If all tests for that change scenario pass, you would expect any similar scenario with more elements in the middle to also pass.

Tests

The successful transition from one state to another in a change scenario is confirmed by calling each interface method and comparing its result to the expected result. Any call that produces the expected result is a passed test; any call that does not produce the expected result is a failed test. Each passed test helps confirm the correctness of the implementation.

For instance, after the change `[A] -> addToFront(B) -> [B, A]`, a call to the `removeFirst` method should return the element `B`. That is one test for the scenario. A complete set of tests for this change scenario would include at least one test for every method of the required interface, including calls with both valid and invalid inputs, where appropriate.

When all tests for a change scenario pass, it helps to confirm that the ADT was correctly transitioned from the starting state to the resulting state by the specific change method. Likewise, the repeated correct results from each method over multiple change scenarios helps to confirm the correct function of each individual method.

It is the combined results of multiple methods over multiple scenarios that provides assurance that the implementation of the ADT is correct. No one scenario or test in isolation can provide that assurance.

Identify Tests for a List Implementing IndexedUnsortedList

A set of basic list change scenarios (starting state, a change, and resulting state) for the IndexedUnsortedList is given, below. These scenarios cover most of the situations your list needs to handle correctly.

Your assignment is to list the expected results for calls to each of the ADT methods for selected scenarios in a plain-text document called `ListTests.txt`.

Even bearing in mind that we are using three-element lists as equivalence cases for all lists with a beginning, middle, and end, writing all appropriate tests for these scenarios (between 20 and 40 tests per scenario) would result in a total of over 1,500 tests. Time will not likely allow completing all tests for all of these scenarios, so we will identify a high-priority set of scenarios that will provide reasonable assurance of a correct implementation.

The goal of this assignment is to **show a plan** for testing, as follows:

- List all possible tests for each scenario that either starts with an empty list or ends with an empty list, specifically scenarios 2-5 and 12-15. Scenario 1 is given to you.
- List all possible tests for scenarios that result in a list of one element, specifically scenarios 16 and 25-30.
- List all possible tests for the two scenarios 7 & 10 that start with a one-element list and result in a two-element list (similar to given scenario 6).
- List all possible tests for the two scenarios 17 & 23 that start with a two-element list and result in a three-element list.
- List all possible tests for the two scenarios 33 & 37 that start with a three-element list and result in a two-element list (similar to given scenario 39).

The ADT interface javadocs describe the expected behavior of each method. Use the documentation and the final state of each change scenario to determine the expected result for each test. You might find it useful to print the interface as a reference and to highlight the required parameter(s) and any exceptions that can be thrown for each method.

For methods that take input parameters, be sure to include separate tests for inputs that should produce different results. If a method should return a value for a valid input parameter and throw an Exception for an invalid parameter, those are two separate tests. The total number of potential tests for each scenario depends on the scenario. Notice in the example tests, shown below, that the number of tests and the expected results from tests are different for each scenario.

Change Scenarios for IndexedUnsortedList

Each scenario listed below is described as a starting state, a change to the list, and the resulting state.
Example tests are shown for some scenarios.

```
1) no list -> constructor -> []
  Tests:
    addToFront(A) throws no Exception
    addToRear(A) throws no Exception
    addAfter(A, B) throws NoSuchElementException
    add(A) throws no Exception
    add(-1, A) throws IndexOutOfBoundsException
    add(0, A) throws no Exception
    add(1, A) throws IndexOutOfBoundsException
    removeFirst() throws NoSuchElementException
    removeLast() throws NoSuchElementException
    remove(A) throws NoSuchElementException
    remove(-1) throws IndexOutOfBoundsException
    remove(0) throws IndexOutOfBoundsException
    set(-1, A) throws IndexOutOfBoundsException
    set(0, A) throws IndexOutOfBoundsException
    get(-1) throws IndexOutOfBoundsException
    get(0) throws IndexOutOfBoundsException
    indexOf(A) returns -1
    first() throws NoSuchElementException
    last() throws NoSuchElementException
    contains(A) returns false
    isEmpty() returns true
    size() returns 0
    iterator() returns an Iterator reference
    listIterator() throws UnsupportedOperationException
    listIterator(0) throws UnsupportedOperationException
    toString returns "[]"
2) [] -> addToFront(A) -> [A]
3) [] -> addToRear(A) -> [A]
4) [] -> add(A) -> [A]
5) [] -> add(0,A) -> [A]
6) [A] -> addToFront(B) -> [B,A]
  Tests:
    addToFront(C) throws no Exceptions
    addToRear(C) throws no Exceptions
    addAfter(C, B) throws no Exceptions
    addAfter(C, A) throws no Exceptions
    addAfter(C, D) throws NoSuchElementException
    add(C) throws no Exception
    add(-1,C) throws IndexOutOfBoundsException
    add(0,C) throws no Exception
    add(1,C) throws no Exception
    add(2,C) throws no Exception
    add(3,C) throws IndexOutOfBoundsException
    removeFirst() returns B
    removeLast() returns A
    remove(A) returns A
    remove(B) returns B
    remove(C) throws NoSuchElementException
    remove(-1) throws IndexOutOfBoundsException
    remove(0) returns B
    remove(1) returns A
    remove(2) throws IndexOutOfBoundsException
    set(-1,C) throws IndexOutOfBoundsException
    set(0,C) throws no Exception
    set(1,C) throws no Exception
```

```
    set(2,C) throws IndexOutOfBoundsException
    get(-1) throws IndexOutOfBoundsException
    get(0) returns B
    get(1) returns A
    get(2) throws IndexOutOfBoundsException
    indexOf(A) returns 1
    indexOf(B) returns 0
    indexOf(C) returns -1
    first() returns B
    last() returns A
    contains(A) returns true
    contains(B) returns true
    contains(C) returns false
    isEmpty() returns false
    size() returns 2
    iterator() returns an Iterator reference
    listIterator() throws UnsupportedOperationException
    listIterator(0) throws UnsupportedOperationException
    toString() returns "[B, A]"
7) [A] -> addToRear(B) -> [A,B]
8) [A] -> addAfter(B,A) -> [A,B]
9) [A] -> add(B) -> [A,B]
10) [A] -> add(0,B) -> [B,A]
11) [A] -> add(1,B) -> [A,B]
12) [A] -> removeFirst() -> []
13) [A] -> removeLast() -> []
14) [A] -> remove(A) -> []
15) [A] -> remove(0) -> []
16) [A] -> set(0,B) -> [B]
17) [A,B] -> addToFront(C) -> [C,A,B]
18) [A,B] -> addToRear(C) -> [A,B,C]
19) [A,B] -> addAfter(C,A) -> [A,C,B]
20) [A,B] -> addAfter(C,B) -> [A,B,C]
21) [A,B] -> add(C) -> [A,B,C]
22) [A,B] -> add(0,C) -> [C,A,B]
23) [A,B] -> add(1,C) -> [A,C,B]
24) [A,B] -> add (2,C) -> [A,B,C]
25) [A,B] -> removeFirst() -> [B]
26) [A,B] -> removeLast() -> [A]
27) [A,B] -> remove(A) -> [B]
28) [A,B] -> remove(B) -> [A]
29) [A,B] -> remove(0) -> [B]
30) [A,B] -> remove(1) -> [A]
31) [A,B] -> set(0,C) -> [C,B]
32) [A,B] -> set(1,C) -> [A,C]
33) [A,B,C] -> removeFirst() -> [B,C]
34) [A,B,C] -> removeLast() -> [A,B]
35) [A,B,C] -> remove(A) -> [B,C]
36) [A,B,C] -> remove(B) -> [A,C]
37) [A,B,C] -> remove(C) -> [A,B]
38) [A,B,C] -> remove(0) -> [B,C]
39) [A,B,C] -> remove(1) -> [A,C]
  Tests:
    addToFront(D) throws no Exceptions
    addToRear(D) throws no Exceptions
    addAfter(D, A) throws no Exceptions
    addAfter(D, C) throws no Exceptions
```

```
    addAfter(D, B) throws NoSuchElementException
    add(D) throws no Exception
    add(-1, D) throws IndexOutOfBoundsException
    add(0, D) throws no Exception
    add(1, D) throws no Exception
    add(2, D) throws no Exception
    add(3, D) throws IndexOutOfBoundsException
    removeFirst() returns A
    removeLast() returns C
    remove(A) returns A
    remove(B) throws NoSuchElementException
    remove(C) returns C
    remove(-1) throws IndexOutOfBoundsException
    remove(0) returns A
    remove(1) returns C
    remove(2) throws IndexOutOfBoundsException
    set(-1, D) throws IndexOutOfBoundsException
    set(0, D) throws no Exception
    set(1, D) throws no Exception
    set(2, D) throws IndexOutOfBoundsException
    get(-1) throws IndexOutOfBoundsException
    get(0) returns A
    get(1) returns C
    get(2) throws IndexOutOfBoundsException
    indexOf(A) returns 0
    indexOf(B) returns -1
    indexOf(C) returns 1
    first() returns A
    last() returns C
    contains(A) returns true
    contains(B) returns false
    contains(C) returns true
    isEmpty() returns false
    size() returns 2
    iterator() returns an Iterator reference
    listIterator() throws UnsupportedOperationException
    listIterator(0) throws UnsupportedOperationException
    toString() returns "[A, C]"
40) [A,B,C] -> remove(2) -> [A,B]
41) [A,B,C] -> set(0,D) -> [D,B,C]
42) [A,B,C] -> set(1,D) -> [A,D,C]
43) [A,B,C] -> set(2,D) -> [A,B,D]
```

Files

Because all of the test cases will be testing the methods in the `IndexedUnsortedList` interface, you will need this file to complete the assignment:

- IndexedUnsortedList.java

For convenience, you may start with this ListTests.txt file, containing the scenarios and example tests identified here.

Grading

Points will be awarded according to the following breakdown:

Tasks	Points
Test plan for at least 24 change scenarios (3 given + 21 defined by you)	20

Required Files

Submit the following file:

- ListTests.txt