Testing IndexedUnsortedList

Mason Vail
Boise State University Computer Science

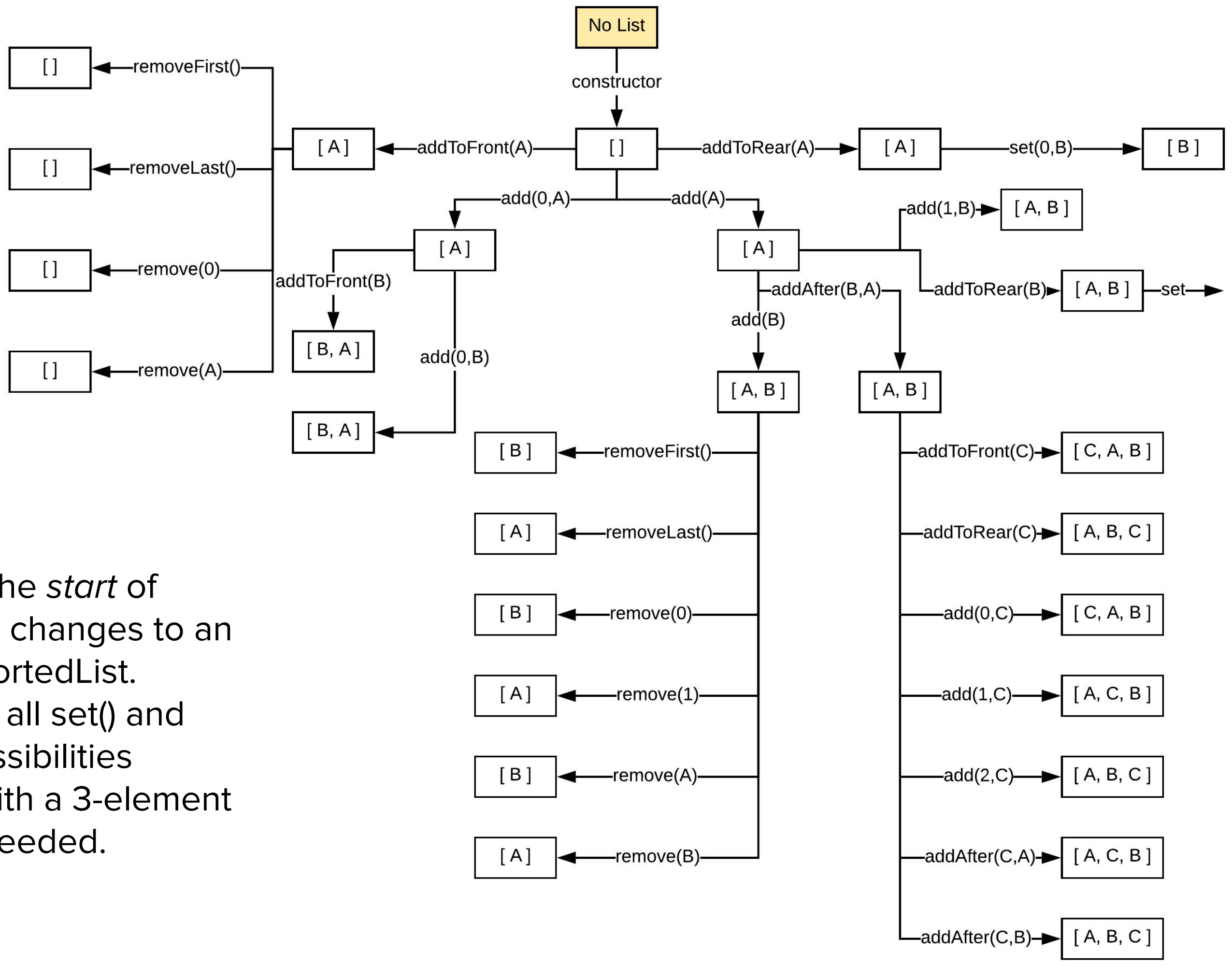
The IndexedUnsortedList Interface

Planning begins with studying the interface to understand the ADT.

IndexedUnsortedList defines an Iterable list that blends unsorted/unordered list and indexed list functionality with functionality expected of all Collections.

Designing a Test Plan: Mapping Changes

Test planning begins with mapping out change scenarios / test cases starting with initialization of a new empty list in a constructor call, all possible changes beginning with that empty list, all subsequent possible changes from a single-element list, and so on.



This is only the *start* of mapping out changes to an IndexedUnsortedList.
At minimum, all set() and remove() possibilities beginning with a 3-element list are still needed.

Basic set of IndexedUnsortedList test cases

1) no list -> constructor -> []
2) [] -> addToFront(A) -> [A]
3) [] -> addToRear(A) -> [A]
4) [] -> add(A) -> [A]
5) [] -> add(0,A) -> [A]
6) [A] -> addToFront(B) -> [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]
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]

Designing a Test Plan: Choosing Cases

All critical boundary cases must be included in the test plan. All changes involving an empty list, all changes to a single-element list, and all changes to the beginning or end of a multiple-element list are boundary cases.

Equivalence cases must also be included for all expected general scenarios, such as changes to a middle element of a multiple-element list. A 3-element list is the smallest list with a distinct beginning, middle, and end.

I would argue that the 43 change scenarios identified so far are the minimum set of boundary and equivalence cases involving only the IndexedUnsortedList ADT methods. (Iterator and ListIterator change methods have not, yet, been considered.)

Designing a Test Plan: Tests for Each Case

All IndexedUnsortedList methods must be tested for each change scenario based on the expected resulting state of the list after the change.

Both valid and invalid method inputs must be tested where appropriate. For example: Both in-bounds and out-of-bounds indexes should be tested for indexed methods.

Tests for Case 1: A New Empty List

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 "[]"

Tests for Case 6: [A] -> addToFront(B) -> [B, A]

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]"

Implementing a Test Suite

There are many ways to implement a test suite, each with advantages and disadvantages. In industry, powerful testing frameworks like JUnit and TestNG are frequently used, because they can be integrated into comprehensive build, test, and reporting systems.

We, however, are going to use a stand-alone test class rather than an industrial strength framework for several reasons.

- Frameworks have their own steep learning curves and would take our focus off of what we are trying to learn about the purpose and processes of testing. Once you understand testing in general, learning a framework later on will be less difficult.
- Frameworks often split test code across many classes - usually one or more per test case - adding to our project management and navigation challenge.
- Debugging through framework code adds significant layers of extra complexity and challenge that we can avoid by containing all test code in a single class.

ListTester.java: Choosing a List to Test

Choose a specific IndexedUnsortedList implementation to test via LIST_TO_USE.

Make sure the constructor associated with LIST_TO_USE is enabled in newList().

ListTester.java: “Scenario Builder” Methods

For each scenario to be tested, create a “scenario builder” method that gets its starting state from an existing scenario builder, calls the one change method for the scenario, and returns the resulting list.

Create an associated “lambda” reference to the method that allows passing the scenario builder method as an argument to tester methods.

ListTester.java: runTests()

Tests for each scenario are run from runTests(). Match scenarios with the appropriate set of tests for the resulting list length.

For scenario emptyList_addToFrontA_A (equivalent to [ ] -> addToFront(A) -> [A]) the resulting list has one element, so testSingleElementList() is called, passing in a reference to the “scenario builder” method and the expected contents of the list as an array, along with matching Strings for output.

ListTester.java: Tests for Lists of Each Size

There is a method containing a set of test calls for lists of each size, from empty lists to three-element lists.

A complete set of IndexedUnsortedList method test calls is given for empty lists (testEmptyList()) and single-element lists (testSingleElementList()), but it will be your responsibility to provide tests for two-element lists and three-element lists following your test plan and the given examples.

ListTester.java: Elements and Results

Named elements (ELEMENT_A, ELEMENT_B, etc.) used throughout testing are defined as constants to abstract away the actual values being used. We do not care that they are Integers or what their numeric values may be. We care only that they are independent objects. We could redefine all elements with a different data type and the tests would be equally valid.

The results of tests, both expected and actual, are represented by enumerated type Result. Expected Result values are passed to method tests and compared against the actual Result while running tests. When the expected and actual Results match, a test passes. Otherwise it fails.

ListTester.java: Individual Method Tests

Methods are provided for testing each IndexedUnsortedList method (e.g. testGet() tests the get() method).

You should look through these test methods to become familiar with how they work, but do not alter any of them. They work exactly as intended.

ListTester.java also provides concurrency tests for Iterators and helper utility methods for testing Iterators which we will discuss in more detail after introducing Iterators in more detail. You can ignore them, for now.

Testing IndexedUnsortedList

Mason Vail
Boise State University Computer Science