CS 36C Winter 2024 Homework 1:
Linked Lists

This project has you writing a significant implementation of a Linked List. This assignment is due on January 23rd at 1400 PST (6:00 pm Davis time). You must work on this assignment individually. We have provided you a rich skeleton project in **`Homework1.zip`** available from Canvas which includes a full Gradle and IntelliJ environment, a couple of non-trivial test functions, and the **`LinkedList.ks`** and **`LinkedListTest.ks`** files you need to extend to implement the project.

Expected Project Outcomes

1. Implement the major Linked List functionality including appending on either end, insertion, accessing of arbitrary elements, iterations, and higher order procedures
2. Implement a comprehensive test suite for your Linked List implementation

Implementation and Grading

You should implement the specified function in LinkedList.ks. These are:

```
1.  LinkedListIterator.computeNext() : Unit
2.  LinkedListCellIterator.computeNext() : Unit
3.  LinkedList.append(item: T) : Unit
4.  LinkedList.prepend(item: T) : Unit
5.  LinkedList.get(index: Int) : T
6.  LinkedList.set(index: Int, item: T) : Unit
7.  LinkedList.insertAt(index: Int, item: T) : Unit
8.  LinkedList.removeAt(index: Int, item: T) : T
9.  LinkedList.indexOf(item: T): Int
10. LinkedList.<R>fold(initial: R, operation: (R, T) -> R) : R
11. LinkedList.<R>map(operation: (T)->R): LinkedList<R>
12. LinkedList.mapInPlace(operation: (T)->T): LinkedList<T>
13. LinkedList.filter(operation: (T) -> Boolean): LinkedList<T>
14. LinkedList.filterInPlace(op: (T) -> Boolean): LinkedList<T>
```

You should also implement test functions in **`LinkedListTest.ks`** for all these functions. For grading on Gradescope you will submit both your **`LinkedList.ks`** and **`LinkListTest.ks`** files. Your **`LinkedList.ks`** file will be combined with our own tests to grade your LinkedList code, while your **`LinkedListTest.ks`** file will be combined with our own implementation of the LinkedList to evaluate how good a job you did on testing.

This does mean that your **`LinkedListTest.ks`** file can only test the public API since it can't call any other functions in the **`LinkedList`** class you may write. If you want to test your own internal functions as well create a separate test file.

The autograder will run both of your files, and you can submit as many times as you want, but only one part of your grade will be available for you to examine: the grade for testing. The

grade for your code itself will be hidden until grades are released in the evening on the 26th of January.

The testing grade is based on code coverage: For full credit your test code must call all the above functions in our implementation, all tests must succeed, and your tests must execute 95% of the lines in our version (this is ***code coverage***: a measure of how well a test suite is at actually checking different options).

Useful Notes

IntelliJ supports running tests in an easy manner from the IDE, by simply selecting the tests as the target you wish to run or debug. You can also run individual tests or all tests at once, and the IDE shows on the left which tests passed.

Additionally there is an option to "run tests with coverage" in the more (...) menu next to where you can select what to run. This will highlight all the code that is actually executed by your tests with green or red depending on whether each line was run or not. We will be using the command-line version of the same tool to see how comprehensive your tests are by running them against our implementation and measuring the coverage.

You are strongly encouraged to do a "red->green" programming style. Make a few test cases first, then code the corresponding code in your **`LinkedList`** class until your tests pass. One thing that helps is you can (and indeed are encouraged to) use the autograder's code coverage test as an oracle as this will help make sure both that your tests are correct[1] and comprehensive.

The **`AbstractIterator`** class in Kotlin is particularly useful. All you need to do to implement a complete iterator is inherit from **`AbstractIterator`** and implement the **`computeNext()`** function. This function should find the next element that should be returned in the iteration and call **`setNext()`**, or, if there are no more elements, call **`done()`**. The iterator operator in **`LinkedList`** uses the LinkedListIterator, while the **`cellIterator()`** function uses the **`LinkedListCellIterator`**. The latter is useful for functions like **`mapInPlace`** or any other case where you want to iterate through the cells in the link list, not just the data.

`mapInPlace()` and **`filterInPlace()`** are *in-place* variants: **`map()`** and **`filter()`** both return new LinkedList objects, while **`mapInPlace()`** and **`filterInPlace()`** modify the existing LinkedList object.

Most of the functions can be linear time (***O****(n)*), but there are exceptions: both **`prepend`** and **`append`** should be constant time while **`get`** and **`set`** should both be constant time for both the first element and the last element. The autograder will check to ensure these performance goals are met. The iterators should be ***O****(1)* for each iteration, which means the total iteration ends up being ***O****(n)*.

---

[1] If you believe that your test is reporting a failure due to a bug in our code submit a report by email with the test case that demonstrates the problem. If indeed it is a bug in our reference code rather than your test-case you will get 1 point towards the "Win A Pi" bug-finding contest.