Practical Documentation CS2001 W08

Linked List Java Implementation.

Student ID: 180023970
Submission Date: 04/11/20

Overview

In this practical I have developed an implementation to the Linked List data structure in accordance to the provided interface. The specification additionally forbids the use of `java.util.standard` classes. There are two parts in the assignment, which are an iterative implementation and a recursive.

I have completed both parts. My implementations are completely free from any `util` classes, including Arrays.

I believe my implementations achieve full functionality. However, due to my priority of not using any of the `java.util.standard` classes (including Arrays, which I believe were permitted), there is an excess complexity overhead in some of the methods. Further discussed in the design part.

Design

This part will be discussed firstly with overall design decisions, the choice of sorting algorithm, cycle detection algorithm and end with the placeholder approach for returning root nodes in methods.

The guiding requirement of this practical is to develop implementations from scratch, not using any of the `java.standard.util` data structures and operations. Therefore, it was the biggest priority to adhere to in developing my solutions. After that, I kept in mind incurred cost of some operations, and attempted to achieve efficiency in my code. I believe some methods could be further optimised, however, as it was not specified for the practical, I decided to focus on the main objectives.

Sorting Algorithm

At the moment of researching the suitable sorting algorithm for my case, I opted for the selection sort, thinking that the Bubble sort could not be implemented, as I thought I could not clone the passed List (to make changes to a list while not affecting the original one).

My implementation performs operations as in the Selection sort algorithm but creates a new List that would be returned. The result is a rather inefficient solution. This is mainly due to the need to filter the List from the current largest element (achived with a rewrite of `filter()` method into `filterFirst()).`

Had I had more time or noticed the inefficiency earlier I would have rewritten it to use bubble sort, and clone the list using the `append(ListNode head, null)` method.

In the recursive implementation, due to Java’s reference by value processing, I decided to sort the list in the descending order and reverse before returning it.

Cycle-detection Algorithm

The case to be solved with cycle detection algorithm (see Figure 1).

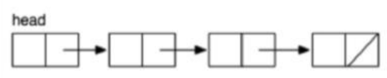
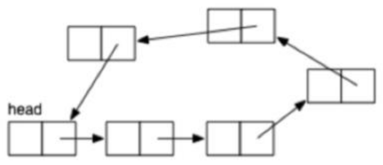
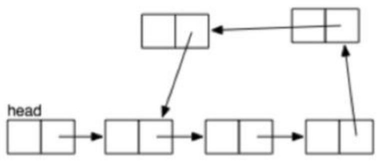
		
isCircular(): false containsCycles(): false	isCircular(): true containsCycles(): true	isCircular(): false containsCycles(): true

Figure 1

I have initially attempted to develop my own "algorithm" to detect cycles. The idea was to clone a `ListNode` and loop through it, removing the previous node from the chain. Supposedly, at the point of pointing backwards at some node that was removed it would generate a null pointer exception, catching which would indicate a list having a cycle. However, through testing I realized that I did not understand Java's reference by value model, thus I decided to research an existing algorithm for this task.

I have found a good fit for this task that would adhere to the spec requirements. It is Floyd's cycle detection algorithm, which loops through the same list with two nodes that move at different speeds (https://en.m.wikipedia.org/wiki/Cycle_detection).

My implementation for finding cycles is this algorithm, while for finding if the list is circular I check if the list has cycles, them break the root's next tie, and check if the `root.next` is NOT circular (later reverting the change to not influence the original list).

Placeholder Approach

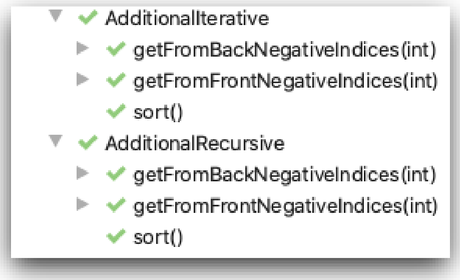
Many methods in the implementation need to return `ListNodes`, and they must be the root nodes of that list. To achieve this in a compact way, I instantiate a node with a `null` element, assign it to a root node value, and then add nodes to the node with a placeholder. When returning a value, I would return the next node of the root node. This proves especially handy in the recursive implementations where I have to pass a node to which append nodes (Reference by value). Additionally, it reduces repetition and unnecessary `if` statements on the node to check if its `null` or not when adding nodes to it.

Testing

The set of tests provided cover almost all cases, and I mainly relied on it while writing the implementations. I have written a few additional ones, to test against some things that were missing in the given tests.

Additional Tests

I have written additional tests to test sort and get by index methods. The sort one tests sorting an already sorted list, a descending sorted list and a list with duplicates. Tests passed without any corrections (see AdditionalTests 1).

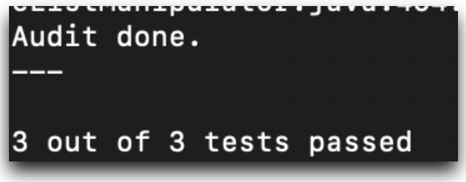


```
▼ ✓ AdditionalIterative
  ▶ ✓ getFromBackNegativeIndices(int)
  ▶ ✓ getFromFrontNegativeIndices(int)
    ✓ sort()
▼ ✓ AdditionalRecursive
  ▶ ✓ getFromBackNegativeIndices(int)
  ▶ ✓ getFromFrontNegativeIndices(int)
    ✓ sort()
```

AdditionalTests 1

Stacscheck

My implementation passes all 44 tests for iterative and recursive implementations (see Stacscheck 1). (one of the passed tests in the stacscheck tests against the 44 provided tests)

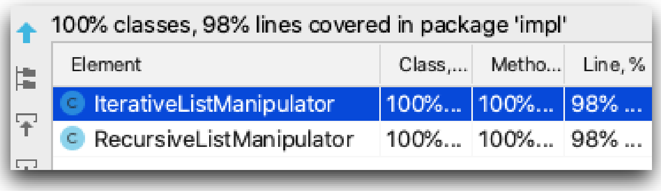


```
Audit done.
---

3 out of 3 tests passed
```

Stacscheck 1

The provided tests coverage (see Coverage 1).



100% classes, 98% lines covered in package 'impl'

Element	Class,...	Metho...	Line, %
IterativeListManipulator	100%...	100%...	98% ...
RecursiveListManipulator	100%...	100%...	98% ...

Coverage 1

Conclusion

I believe my program comes up to the expectations for this practical. It successfully implements the provided interface both iteratively and recursively. Moreover, my implementation is fully free from Array class usage, which I consider a good achievement.

Overall, it was a very new experience to program solutions without the `java.standard.util` classes. It was definitely something that required a different approach to problems and a rather different mindset overall. Furthermore, this was my first experience with writing recursive methods, which proved to be a challenge.

Given more time, I believe I would have mainly focused on reanalysing my solutions for efficiency improvements. Although I kept complexity of different actions in mind, I believe some operations could have being carried out with a lower cost, for example my implementation for the sorting method (given more time I would have rewritten it using the Bubble Sort Algorithm). Moreover, I would have written more tests.