Week 8: Linked Lists

CS2001

Due date: Wednesday 7th November, 21:00
25% of Practical Mark for the Module

Objective

To gain experience in implementing and debugging code that manipulates linked lists.

Learning Outcomes

By the end of this practical you should:

- have a good understanding of how to program using linked lists

Getting started

To start with, you should create a suitable assignment directory such as `CS2001/W08-Lists` on the Linux lab clients. You should decompress the `zip` file at

```
http://studres.cs.st-andrews.ac.uk/CS2001/Practicals/W08-Lists/code.zip
```

to your assignment directory. Please note that the `zip` file contains a number of files in the `src` directory, some of which are blank or only partially implemented. All your source code should be developed with `src` as the root directory for source code. **Once you have extracted the `zip` file, and have completed some of your own implementation, take care that you don't extract the `zip` file again thereby accidentally overwriting your `src` directory (and your own implementation) with files contained in the `zip`.**

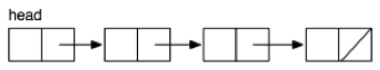
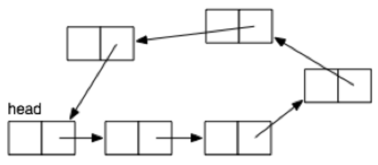
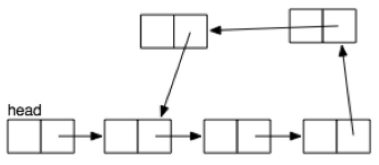
Requirements

You are to develop an implementation of the ADT interface `IListManipulator` (in the `interfaces` package) by writing correct implementations of the methods in the `IterativeListManipulator` and/or `RecursiveListManipulator` classes (in the `impl` package). You are also to include in your report, informal worst case time and space complexity estimates for each operation (i.e. whether the operations have constant, logarithmic, linear, or quadratic time/space complexity).

The ADT specifies a number of operations on linked lists, roughly in order of increasing implementation difficulty. To help you test your implementation(s), some JUnit tests are provided in the abstract `ListManipulatorTest` class which can be run via its concrete subclasses on the iterative and recursive implementations. For simplicity, the tests can also be run by using stacscheck (see section below on *Running the Automated Checker*). As it stands, the methods in the `IterativeListManipulator` and `RecursiveListManipulator` classes are auto-generated and return 0 or null, so will fail auto-checker tests until you provide your own correct implementation.

All of the operations are open to both iterative and recursive implementations. An iterative method contains explicit iteration in the form of a for or while loop, whereas a recursive method does the iteration implicitly by calling itself. Usually, the iterative version is more straight-forward to think about, and may be more efficient to execute, whereas the recursive version is often more elegant and concise. Unless you're very confident with recursion you may wish to start with iterative implementations.

To clarify the `isCircular()` and `containsCycles()` operations, the diagrams below show examples of the required results for certain lists. For all other operations, you can assume that any input lists have a conventional linear non-cyclic structure.

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

Please make sure that you do not modify the ADT interfaces, package structure, the names of the `IterativeListManipulator` and `RecursiveListManipulator` classes, the exception classes etc. for the basic requirements, or even for the suggested extensions below. Instead, you should focus on providing correct implementations of the methods in the supplied list manipulator classes. That said, if you attempt extension work beyond what is suggested below, which does require alterations, please submit this code in a second `W08-Lists-Extension` directory inside your assignment directory.

Running the Automated Checker

Similarly to previous assignments, you can run the automated checking system on your implementation(s) by opening a terminal window connected to the Linux lab clients/servers and executing the following commands:

```
cd ~/CS2001/W08-Lists
stacscheck /cs/studres/CS2001/Practicals/W08-Lists/Tests
```

assuming `CS2001/W08-Lists` is your assignment directory. This will run the JUnit tests in the test package on your ADT implementation. As usual, the auto-checker test `TestQ CheckStyle` runs a program called Checkstyle over your source code using the *Kirby Style*.

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/programming-style.html`

Deliverables

Hand in via MMS, by the deadline of 9pm on Wednesday of Week 8:

- The complete source code for your ADT implementation.
- A report with an advisory limit of around 1000 words, in PDF format, describing your design and implementation, any difficulties you encountered, how you tested your implementation above and beyond using stacscheck.
- The report should also include, with a short justification, informal worst case time and space complexity estimates for each operation as mentioned above.

Marking and Extensions

The submission will be marked on the University's common reporting scale according to the mark descriptors in the Student Handbook at `https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`

To achieve a mark of 14 - 16, the submission should contain clear, well-structured code achieving almost all required basic functionality using an iterative *or* recursive implementation, together with a clear report showing a good level of understanding. Please note that the basic requirements include producing a very good implementation of the ADT and correct informal complexity estimates.

To achieve a mark of 17 or above, the submission should contain excellent code satisfying all basic requirements, together with a clear and well-written report showing real insight into the subject matter and should attempt one or more of the extension activities indicated below. Please remember, it is key that you focus on getting a very good solution to the core requirements prior to touching extensions. A weak solution with an attempt at extensions is still a weak solution.

Extension activities include:

- Produce iterative *and* recursive versions of each operation and give complexity estimates for each.
- Consider whether there are any gaps in the provided tests, and implement new tests if so.

Make sure you describe any extensions you implement in your report.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof): `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Good Academic Practice

As usual, I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at

`https://www.st-andrews.ac.uk/students/rules/academicpractice/`