Carleton University
Department of Systems and Computer Engineering
SYSC 2100 — Algorithms and Data Structures — Winter 2023

Lab 6 - LinkedList: adding operations to the linked-list implementation of ADT List

Getting Started

1. Review *Important Considerations When Submitting Files to Brightspace*, which can be found in the course outline.

2. Launch Wing 101 and configure Wing's code reformatting feature. Instructions can be found in Appendix A of document, *SYSC 2100 - Style Guide for Python Code*.

   All code you submit for grading must be formatted. If you decide to disable automatic reformatting, make sure you manually reformat your code (Appendix A.2). At a minimum, we recommend that you reformat your file as you finish each exercise.

3. Download lab6_linkedlist.py and lab6_test_linkedlist.py from the *Lab Materials* module in Brightspace.

4. Open lab6_linkedlist.py in Wing 101. Locate these assignment statements at the top of the file:

   ```
   __author__ = ''
   __student_number__ = ''
   ```

   Replace the empty strings with strings containing your name and student number. (Don't modify the variable names.)

5. Your solutions to the exercises must conform to the coding conventions described in *SYSC 2100 - Style Guide for Python Code*.

6. The methods you write in class `LinkedList` are not permitted to use instances of Python's container types (`list`, `set`, etc.). The test classes in lab6_test_linkedlist.py are permitted to do this.

7. **Important**: if you decide to write a script in lab6_linkedlist.py, it **must** be placed in an `if __name__ == '__main__':` block, like this:

   ```
   class LinkedList:
       # methods not shown

   if __name__ == '__main__':
       empty_list = LinkedList()
       lst = LinkedList([1, 2, 3, 2, 1])
   ```

```
...
```

Doing this ensures that the script will only be executed when you run lab6_linkedlist.py as the "main" module, and won't be executed when the file is imported into another module; for example, the testing/grading program provided to the TAs.

Overview of Class `LinkedList`

Class `LinkedList` in lab6_linkedlist.py contains a partial implementation of ADT List that uses a singly-linked list as its data structure. The data structure is similar to one of the linked list designs that was presented in SYSC 2006: one variable (`_first`) points to the node at the head of the list and another variable (`_last`) points to the node at the tail of the list. There's one major difference: the first node (head node) is always a "dummy" node. The payload stored in the dummy node is always `None`. When *n* items are stored in a linked list, the list will contain *n*+1 nodes (*n* nodes, each containing one item, plus the dummy node).

Read methods `__init__`, `__str__`, `__repr__`, `__len__`, `__iter__`, `__getitem__`, `__setitem__`, `__add__`, `append` and `insert`. Use the Python shell to try the docstring examples. **Do not modify these methods.** In Part A, you'll use Python Tutor to trace the execution of some of these methods.

The class also has "stub" implementations of methods `__contains__`, `__eq__`, `__delitem__` and `remove`. If you call any of these methods on a `LinkedList` object, Python will raise a `NotImplementedError` exception. You'll complete the implementation of these methods in Part B, Exercises 1 - 4.

Part A

A link to a Python Tutor script that will help you visualize `LinkedList` objects is provided in the Lab 6 module in Brightspace. Open the script by clicking the link. Scroll down until you reach the code after the definition of `LinkedList`.

Exercise 1: Trace the execution of `__init__` when `lst = LinkedList()` is executed. After the `LinkedList` object is initialized, what do instance variables `_first` and `_last` refer to? An empty `LinkedList` has one node (the dummy node), so why is instance variable `_num_items` bound to 0 instead of 1? (Don't submit your answers to the questions in Part A, but make sure you can answer them before you start Part B.)

Exercise 2: Trace the three calls to `append`. After each call returns, which nodes do instance variables `_first` and `_last` refer to?

Exercise 3: Trace the execution of `__getitem__` when `lst[0] = y` is executed. How may times is the loop body (`cursor = cursor.next`) executed? To which node does `cursor` refer when `return cursor.item` is executed?

Repeat this exercise, this time tracing the execution of `__getitem__` when `y = lst[2]` is executed.

Exercise 4: Trace the execution of `__setitem__` when `x = lst[0]` is executed. How may times is the loop body (`cursor = cursor.next`) executed? To which node does `cursor` refer when `cursor.item = x` is executed?

Repeat this exercise, this time tracing the execution of `__setitem__` when `lst[2] = x` is executed.

Exercise 5: Trace the execution of the two calls to `insert`. After each call returns, which nodes do instance variables `_first` and `_last` refer to?

Methods `__getitem__`, `__setitem__` and `insert` have the same `for` loop, which uses `cursor` to traverse the linked list. Methods `__getitem__` and `__setitem__` initialize `cursor` this way: `cursor = self._first.next`, but `insert` initializes `cursor` this way: `cursor = self._first`. Explain the reason for this difference.

Part B

Exercise 1: Open lab6_linkedlist.py in Wing 101. Read the docstring for `__contains__`, delete the `raise` statement, then implement the method. The computational complexity of this method must be *O*(*n*).

Use the Python shell to run a few tests on `__contains__`.

Think about the test cases that would be required to thoroughly test `__contains__`. In lab6_test_linkedlist.py, implement these test cases as methods in class `ContainsTestCase`. Run the test script and review the output in the shell window. If necessary, edit `__contains__` and rerun the test script until all test cases pass.

Exercise 2: Read the docstring for `__eq__`, delete the `raise` statement, then implement the method. The computational complexity of this method must be *O*(*n*). Hint: you should be able to reuse much of the code from `__eq__` in class `ArrayList` in lab4_arraylist.py.

Use the Python shell to run a few tests on `__eq__`.

Think about the test cases that would be required to thoroughly test `__eq__`. Implement these test cases as methods in class `EqTestCase`. Run the test script and review the output in the shell window. If necessary, edit `__eq__` and rerun the test script until all test cases pass.

Exercise 3: Read the docstring for `__delitem__`, delete the `raise` statement, then implement the method. The computational complexity of this method must be *O*(*n*).

Are there any cases where instance variables `_first` or `_last` should be updated? If so, makes sure your code handles these cases.

Note: When the index is invalid, the method should raise an `IndexError` exception containing the message, `"LinkedList: assignment index out of range"`.

Think about the test cases that would be required to thoroughly test `__delitem__`. Implement these test cases as methods in class `DelTestCase`. (Don't delete `test_del1` and `test_del2`.) Run the test script and review the output in the shell window. If necessary, edit `__delitem__` and rerun the test script until all test cases pass.

Exercise 4: Read the docstring for `remove`, delete the `raise` statement, then implement the method. The computational complexity of this method must be $O(n)$.

Are there any cases where instance variables `_first` or `_last` should be updated? If so, makes sure your code handles these cases.

Note: When the list is empty or if the argument (parameter `x`) is not in the list, `remove` should raise a `ValueError` exception containing the message, "`LinkedList.remove(x): x not in list`".

Use the Python shell to run a few tests on `remove`.

Think about the test cases that would be required to thoroughly test `remove`. Implement these test cases as methods in class `RemoveTestCase`. (Don't delete `test_remove1` and `test_remove2`.) Run the test script and review the output in the shell window. If necessary, edit `remove` and rerun the test script until all test cases pass.

Wrap-up

- Before submitting lab6_linkedlist.py, review your code. Does it conform to the coding conventions, as specified in the *Getting Started* section? Has it been formatted? Did you edit the `__author__` and `__student_number__` variables?
- Submit lab6_linkedlist.py to Brightspace. Make sure you submit the file that contains your solutions, not the unmodified file you downloaded from Brightspace! You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the submission due date. Only the most recent submission will be saved by Brightspace.
- Don't submit lab6_test_linkedlist.py.
- Solutions that are emailed to your instructor or a TA will not be graded, even if they are emailed before the submission due date.