CSC148, Lab #8

This document contains the instructions for lab number 8 in CSC148H. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, **not** to make careful critical judgments on the results of your work.

General rules

We will use the same general rules as for the previous labs (including pair programming).

Overview

In this lab you will begin to work with linked lists. Here are some general hints on how to proceed:

- **Draw lots of pictures** to get a very clear idea of exactly what the linked structure should look like before, during, and after each operation you are trying to implement. This is important: if you skip this, you are likely to have only a vague idea of what it is your code should do, and you are much more likely to make mistakes!
- Think carefully about what variables and attributes each part of your picture represents. If you understand this, then you will be able to tell exactly what changes you need to make in order to carry out the required task.

re-implement Queue

(Student `s1` drives, student `s2` navigates)

Download `linked_list.py`. This is the linked list code from lecture.

In a new file `linked_queue.py`,
add a class Queue that implements the Queue ADT using a linked list (see `csc148queue.py` for the definition of the queue ADT and the methods you should implement). Do not use any list or dictionary or other standard container! Remember: the point of this exercise is to practice linked lists.

Use the file `testqueue.py` to test that your implementation works correctly for some inputs. Once you have done so, use the file `timequeue.py` to check the efficiency of your implementation. If you have done it correctly, `timequeue.py` should show that the time is the same no matter how many elements are in the queue. This is a core use case for a linked list and why it can be better than the Python built-in list for some purposes. If we use a Python list for a queue, the time taken to enqueue and dequeue a fixed number of items **increases linearly** with the size of the queue!

Hint: think about each queue operation and how it maps onto linked-list operations. For example, you'll certainly want a way to remove and return an item from one end of the linked list, and add an item at the other end of the linked list. You'll also want to be able to return an item from the linked list **without removing that item from the linked list**. Add any missing methods to `linked_list.py` as appropriate, and then call those methods to implement your queue operations.

Adding List Features

(Student s2 drives, student s1 navigates)

The built-in Python `list` objects support a lot of stuff that our `LinkedList` objects do not. Here, we will re-create some of those features and add them to our `LinkedList` class.

Implement the following methods in the `LinkedList` class:

```
__len__(self):
    Return the number of elements in this linked list.
__setitem__(self, index, val):
    Set the value at index to val, raise an exception if index
    is out-of-bounds
__contains__(self, val):
    Return True iff val is in linked list
__delitem__(self, index):
    Remove node at index.
    Raise an exception if index is out-of-bounds.
insert(self, index, val):
    Insert val at index in this linked list, raise an exception
    if index is out-of-bounds.
```

Some of these are special methods (denoted by the double-underscore `__`) that are never called directly, but that get called automatically in appropriate situations. Here is an example of how your class could be used and its expected behaviour:

```
list1 = LinkedList()  # creates a new empty linked list
print(len(list1))  # automatically calls list1.__len__(); prints '0'
print(5 in list1)  # automatically calls list1.__contains__(5); prints 'False'
print(list1[0])  # automatically calls list1.__getitem(0); raises IndexError
list1.prepend(15)
list1.prepend(17)
print(len(list1)) # prints '2'
print(list1[0]) # prints '17'
list1[0] = 19 # automatically calls list1.__setitem__(0, 19)
print(15 in list1) # automatically calls list1._contains(15); prints True
del list1[1] # automatically calls list1.__delitem__(1)
print(len(list1)) # prints 1
list1.prepend(19)
list1.insert(1, 23) # list is now 19->23->19
```

When you're done, show your TA what you've achieved.

extra

If you finish the preceding exercises, you may want to try the following (switch roles, of course!)

1. Implement the method `copy_ll()` for class `LinkedList`. This should produce a new `LinkedList` with equivalent items:

    ```
    def copy_ll(self: 'LinkedList') -> 'LinkedList':
        """Return a copy of self.

        >>> lnk = LinkedList(5)
        >>> lnk.prepend(7)
        >>> lnk2 = lnk.copy_ll()
        >>> lnk is lnk2
        False
        >>> repr(lnk) == repr(lnk2)
        True
        >>> lnk.prepend(11)
        >>> repr(lnk) == repr(lnk2)
        False
        """
    ```

2. Implement the method `filter_ll` for the class `LinkedList`. This should produce a new list where `item` is in the list iff `t(item)` is `True`.

    ```
    def filter_ll(self: 'LinkedList',
                  t: 'boolean single-argument function') -> 'LinkedList':
        """Return LinkedList of items for which t(item) == True, in the
        same order they appear in self.

        >>> lnk = LinkedList(5)
        >>> lnk.prepend(4)
        >>> lnk.prepend(7)
        >>> def f(n): return n % 2 == 0
        >>> lnk2 = lnk.filter_ll(f)
        >>> repr(lnk2) == repr(LinkedList(4))
        True
        """
    ```

3. Implement a stack using `linked_list`. (We wrote a stack using a Python list in week 2. Now we want you to use a linked list!)