CSC148, Lab #6
week of February 24th, 2014

This document contains the instructions for lab number 6 in CSC148H1. 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 first lab (including pair programming). See the instructions at the beginning of Lab 1 to refresh your memory.

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 task you want to carry out.

re-implement Queue

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

Download linked_list.py (you may have seen this in lecture).

In **linked_list.py**, add a class `Queue` that implements the Queue ADT using a linked list (see csc148queue.py for the definition of the ADT and of 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 (what did your queue implementation from lab 2 based on Python `list` do?)

Here are some hints:

Hint #1: Instances of your Queue class should store two attributes: a reference to where you add new LinkedList elements, and a reference to where you remove the oldest LinkedList element. Consider carefully which end of the linked list is best for adding new elements versus which end is best for removing the oldest element.

Hint #2: You need to be careful to update your attributes correctly when the queue starts out (or ends up) empty.

Once you are done and confident that your code works, show your work to your TA.

implement `list`, (sort of)

(Student s2 drives, student s1 navigates). If there is time, tackle the following task.

In the class **LinkedList**, elements are stored as a sequence of objects, where each object stores one list element (head) and a reference to the rest of the list. You will now add some methods to **LinkedList** that recreate some of the features of Python's built-in **list** class.

Your class should implement the following methods (try to complete at least a few):

```
__len__(self):
    Return the number of elements in this linked list.
__setitem__(self, ind, val):
    Set the head at position ind to val, raise an exception if ind
    is out-of-bounds
__delitem__(self, ind):
    Remove head at position ind, shifting every head after that down
    by 1.  Raise an exception if ind is out-of-bounds.
insert(self, ind, elem):
    Insert elem at index ind in this linked list, raise an exception
    if ind is out-of-bounds.
    Does not overwrite any element previously at index 0,
    though the index of that element (and all that follow) increases
    by one.
```

Some of these are special methods (denoted by the double-underscore `__`), that are never called directly, but that get called automatically in appropriate situations. For example, here is one way that your class could get 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)
len(list1) # calls list1.__len__()
```

```
list1.prepend(19)
list1.insert(1, 23)
```

extra

If you finish the preceding exercise, you may want to try the following:

1. In **linked_list.py** add the method **reverse** which reverses the LinkedList in place, without building up an extra list.

2. Implement the **Stack** using **linked_list**. You'll find an implementation based on Python **list** under Week 2 in the course web site.

3. Look over the API for Python list and see what other list methods you might implement in **linked_list.py**

4. Add the following import statement at the top of **linked_list.py**:

   ```
   from collections import MutableSequence
   ```

   ... and then make **LinkedList** a subclass of **MutableSequence**. Now comment out your implementation of **reverse** and notice that it has been inherited from MutableSequence, along with several other methods.