CSC148, Lab #7
week of March 3rd, 2014

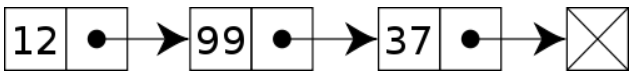
This document contains the instructions for lab #7 in CSC148. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, **not** to make careful critical judgments on the results of your work. We will use the same general rules as for the first lab (including pair programming). See the instructions at the beginning of Lab 1 to refresh your memory.

Overview

We are going to continue with, and then extend, an exercise from the second half of last week's lab on **LinkedLists**. Many students find mutation — changing the state of an object — a little challenging to keep track of, and linked lists give you lots of opportunity to change objects.

Re-read these hints!

- **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.
- Python methods that change an object usually don't return any value other than **None**. This may surprise you when you try to use an existing method in later code.

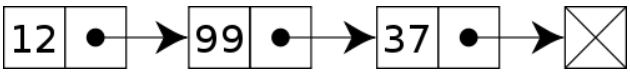
mimic Python list

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

Download linked_list.py .

In the class **LinkedList**, elements are stored as a sequence of objects, where each object stores one list element (item) and a reference to the rest of the list (rest). You will now add some methods to **LinkedList** that recreate some of the features of Python's built-in **list** class. **LinkedList** already has some methods, including **prepend**, which adds a new **item** to the list (making the list one element longer), and **decapitate**, which removes the existing **item** element from the list (making the list one element shorter).

Read over the code for **prepend** and **decapitate**, and draw pictures of what a typical **LinkedList** looks like before and after each operation.



Ask your TA to look at your drawings before you continue.

Now, add the following methods to your **LinkedList** class in the file **linked_list.py**:

```
__len__(self):
    Return the number of elements in this linked list.
__setitem__(self, ind, val):
    Set the item at position ind to val, raise an exception if ind
    is out-of-bounds
__delitem__(self, ind):
    Remove item at position ind, shifting every item 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)
```

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

extra

If you finish the preceding exercise, you may want to try the following (switch partners, 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 LinkedList self

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

2. Implement the method **filter_ll** for the class **LinkedList**. This should produce a new list of items that pass the test **t(item)**.

```
def filter_ll(self: 'LinkedList',
              t: 'boolean single-argument function') -> 'LinkedList':
    """Return LinkedList of items of self that pass t(item), 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)
    >>> lnk2 == LinkedList(4)
    True
    """
```

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

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

5. 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.

If you achieve some of these, talk to your TA. If you don't, also talk to your TA!