LINKED LISTS

COMPUTER SCIENCE MENTORS 61A

October 10 to October 14, 2016

For each of the following problems, assume linked lists are defined as follows:

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

To check if a `Link` is empty, compare it against the class attribute `Link.empty`:

```
if link is Link.empty:
    print('This linked list is empty!')
```

1 What Would Python Print?

1. What will Python output? Draw box-and-pointer diagrams to help determine this.

```
>>> a = Link(1, Link(2, Link(3)))
>>> a.first

>>> a.first = 5
>>> a.first

>>> a.rest.first

>>> a.rest.rest.rest.rest.first


>>> a.rest.rest.rest = a
>>> a.rest.rest.rest.rest.first
```

2 Code Writing Questions

2. Write a function `skip`, which takes in a `Link` and returns a new `Link`.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> a
    Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    Link(1, Link(3))
    >>> a
    Link(1, Link(2, Link(3, Link(4)))) # Original is unchanged
    """
```

3. Now write function `skip` by mutating the original list, instead of returning a new list. Do NOT call the `Link` constructor.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    Link(1, Link(3))
    >>> a
    Link(1, Link(3))
    """
```

4. Write a function `reverse`, which takes in a `Link` and returns a new `Link` that has the order of the contents reversed.
   Hint: You may want to use a helper function if you're solving this recursively.

```
def reverse(lst):
    """
    >>> a = Link(1, Link(2, Link(3)))
    >>> b = reverse(a)
    >>> b
    Link(3, Link(2, Link(1)))
    >>> a
    Link(1, Link(2, Link(3)))
    """
```

5. **(Optional)** Now write `reverse` by modifying the existing Links. Assume reverse returns the head of the new list (so the last `Link` object of the previous list).

   First, draw out the box and pointer for the following:

```
>>> a = Link(1, Link(2))
>>> a.rest.rest = a
>>> a.rest = Link.empty
```

   Observe how the pointers change, as well as the order in which they are modified.

   Now, generalize this to reverse an entire linked list.

```
def reverse(lst):
    """
    >>> a = Link(1, Link(2, Link(3)))
    >>> b = reverse(a)
    >>> b
    Link(3, Link(2, Link(1)))
    >>> a
    Link(1)
    """
```

6. **(Optional)** Write `has_cycle` which takes in a `Link` and returns `True` if and only if there is a cycle in the `Link`.

```
def has_cycle(s):
    """
    >>> has_cycle(Link.empty)
    False
    >>> a = Link(1, Link(2, Link(3)))
    >>> has_cycle(a)
    False
    >>> a.rest.rest.rest = a
    >>> has_cycle(a)
    True
    """
```

7. **Orders of Growth and Linked Lists:** Consider the following linked list function:

```
def insert_at_beginning(lst, x):
    return Link(x, lst)
```

(a) What does this function do?

(b) Assume lst is initially length n. How long does it take to do one insert? Two? n?

Now consider:

```
def insert_at_end(lst, x):
    if lst.rest is Link.empty:
        lst.rest = Link(x)
    else:
        insert_at_end(lst.rest, x)
```

(c) What does this function do?

(d) Say we want to repeatedly insert some numbers into the end of a linked list:

```
def insert_many_end(lst, n):
    for i in range(n):
        insert_at_end(lst, i)
```

i. Assume `lst` is initially length 1. How long will it take to do the first insertion? The second? The nth?

ii. In big-O notation, What is the total runtime to do all the inserts? (total runtime of `insert_many_end`)