[S17 HW10] linked lists

Due April 11, 2017, 9 a.m.

MATH 121 SPRING 2017 HOMEWORK 10

Linked Lists

In the problems below you are asked to develop a series of linked list methods and also classes of objects that use lists of linked nodes as their underlying representation. You should work from the starting code for linked lists that we showed you in lecture, available for download here. (http://jimfix.github.io/math121/lab/hw10/Linked.py) In some of the problems, we will ask you to extend the class definition with a new method. In others, we will ask you to devise a class from scratch, but you can our code to make your methods for that new class.

For these new classes, our testing code will be examining the structure of the linked list of nodes your code makes. You need to follow our instructions carefully, naming the instance variables the way we prescribe them so that our tests can examine the objects' underlying linked structure.

[S17 HW10 P1]

Exercise: sum

We will work on this exercise together in lab.

Add a method called `sum` to the the `Linked` class that computes and returns the sum of all the values held in the nodes of the linked list.

```
>>> l = Linked()
>>> l.append(7)
>>> l.append(10)
>>> l.append(1)
>>> l.display()
[7, 10, 1]
>>> l.sum()
18
```

[F16 HW7 P2]

Exercise: count

This is a partner exercise and is to be submitted by the LEFT partner.

Add a method called `count` to the `Linked` class that takes a value and returns how many times that value appears in the linked list.

in the nodes of the linked list.

```
>>> l = Linked()
>>> l.append(7)
>>> l.append(10)
>>> l.append(1)
>>> l.append(7)
>>> l.display()
[7, 10, 1, 7]
>>> l.count(10)
1
>>> l.count(7)
2
>>> l.count(6)
0
```

[F16 HW7 P3]

Exercise: apply

Add a method called `apply` to the `Linked` class that takes a function as its argument. It should modify each of the values in the nodes with the value of that function applied to it.

```
>>> l = Linked()
>>> l.append(7)
>>> l.append(10)
>>> l.append(1)
>>> l.display()
[7, 10, 1]
>>> l.apply(lambda x: x*x)
>>> l.display()
[49, 100, 1]
>>> l.apply(lambda x: x+1)
>>> l.display()
[50, 101, 2]
```

[F16 HW7 P4]

Exercise: sorted list

This is a partner exercise and is to be submitted by the RIGHT partner.

Within the same file, add a new class definition `Sorted` that inherits from `Linked` . It should instead maintain the items that are appended in sorted order. To do this, you'll want to override the `append` method of `Linked` objects by writing a new `append` method to the class `Sorted` that inserts nodes into the list in the order of their keys. The first node should hold the smallest item, and the subsequent nodes should be at least as large, or larger than this first item. Thus, moving down the nodes from the first to the last, the values should be in sorted order. You need not change or add any other methods to this class.

```
>>> l = Sorted()
>>> l.append(7)
>>> l.append(10)
>>> l.append(1)
>>> l.display()
[1, 7, 10]
```

[F16 HW7 P5]

Exercise: delete all

Add a new method `deleteAll` to class `Linked` that takes a value and modifies the linked list so that all nodes with that value are removed. If no nodes have that value, the list should not be changed.

```
>>> l = Linked()
>>> l.append(7)
>>> l.append(10)
>>> l.append(7)
>>> l.append(1)
>>> l.display()
[7, 10, 7, 1]
>>> l.deleteAll(7)
>>> l.display()
[10, 1]
>>> l.deleteAll(1)
>>> l.display()
[10]
```

[F16 HW7 P6]

Exercise: Queue

In class we will describe a `Stack` data structure. With a `Stack` , the item that you last "pushed" onto the stack is the item that gets removed when you "pop" an item off of the stack. Stacks are often used to keep track of a set of things when, for some reason, the "last thing in is the first thing out" behavior is the right organization for what you need to track. For example, a browser maintains a stack of web pages you visit as you follow links on a series of pages. That is, each time you click on a link to a next page, it gets put on top of a stack of pages that led you to that link. The "back" button takes you back to the page that led you there, "popping off" the page in front of you to take you to the prior page. It turns out that lots of algorithms rely on stacks, specifically ones that need this kind of "back trace" behavior.

There is another common collection, similar to stacks, for organizing a sequence of items called a `Queue` . The two main methods of queues are `enqueue` which places an item onto the queue, and `dequeue` which removes an item from the queue. Contrary to the stack where the most recently added item is removed, a `dequeue` instead removes the *least recently added* item from the queue.

A queue can be thought of as a line of items, where a new item gets added to the back of the line with `enqueue` , and where `dequeue` removes the item from the front. The front item is often called the "head" item of the queue.

Devise a linked list implementation of a `Queue` class. In addition to `__init__` , which makes an empty `Queue` , it should have the methods `enqueue` , `dequeue` , and `head` . We've provided a template of the code (http://jimfix.github.io/math121/lab/hw10/Queue.py) that you can complete (replacing each of the `pass` lines with a method's code).

Write the code so that it maintains two instance variables `first` and `last` . When the queue is empty, both its `self.first` and `self.last` should be `None` . When the queue has several items, `self.first` should contain the node with the value that's been sitting in the queue the longest. This would be the value that was least recently added, the head item. The next node after `self.first` should contain the value that was added after the "head", and so on, all the way to the node at the end, which should contain the value most recently added. The variable `self.last` should be that last node in the list.

When a value is added to the queue with `enqueue` , a new node should be placed at the end of the list, and `self.last` should be changed to refer to that new node. When a value is removed with `dequeue` the node referenced by `self.first` should be removed and `self.first` should change to the next node in the linked list.

The `dequeue` method should return the value that was held in that removed node. The `head` method should return the head value, but not remove it from the queue. If the queue is empty, the `head` method should return `None` .

When a queue has only one item, `self.first` and `self.last` will be referring to the same node in the linked list.

```
>>> q = Queue()
>>> q.enqueue(7)
>>> q.enqueue(10)
>>> q.enqueue(1)
>>> q.head()
7
>>> q.dequeue()
7
>>> q.head()
10
>>> q.enqueue(8)
>>> q.dequeue()
10
>>> q.head()
1
```

[F16 HW7 P7]

Exercise: doubly-linked

Let's now devise a new linked list class with a different `Node` type, one that provides a "doubly-linked" list which we describe below. We'll name the new class 'DLinked' and its node class 'DNode'.

In a doubly-linked list, nodes have a link to both the *previous* and the next node in their structure. That means that each 'DNode' should have a `value` , a `next` , and a `prev` . The use of `value` and `next` are as before for singly-linked lists. As before, a node's `next` refers to the node that follows it in the linked structure. In addition, a node's `prev` refers to the node that *precedes it* in the linked structure. When a node is the last in the linked list, its `next` should be `None` . If a node is the first in the linked list, its `prev` should be `None` . For any node that is neither the first or the last, its `next` node should have it as `prev` . Its `prev` node should have it as `next` . That is to say, for any node `n` sitting somewhere in the middle of a doubly-linked list

```
n.next.prev == n
```

and

```
n.prev.next == n
```

Rewrite the code for the *original* `Linked.py` source we provided. Call it `DLinked.py` and name its classes `DNode` and `DLinked` . Its `DNode` class should build doubly-linked nodes that contain both `next` and `prev` attributes. A `DLinked` object should have two attributes: `start` and `end` . When the list is empty, these should both be set to `None` . When the list has length one, that one node should be both the `start` and the `end` . When the list is longer, `start` should refer to that list's first node and `end` should refer to that list's last node.

Write the `append` , `insert` , and `delete` methods of `DLinked` so that they modify both `next` and `prev` when nodes are linked and unlinked with these methods. They should also modify the `start` and `end` references in cases where those parts of the list structure change.

Note that the other methods we wrote for `Linked` will not need to be changed for `DLinked` . They will still work since they only inspect the nodes in the list (they don't modify them) and rely only on the `next` attribute of nodes and the `start` attribute of lists.

Attempt [S17 HW10 P3] apply

Attempt [S17 HW10 P4] sorted list

Attempt [S17 HW10 P5] delete all

Attempt [S17 HW10 P6] Queue

Attempt [S17 HW10 P7] doubly-linked

Attempt [S17 HW10 P1] sum

Attempt [S17 HW10 P2] count