Linked Lists done right

In this assignment you will reimplement a `LinkedList` class, using nodes.

The API will remain the same as for Lab 03. In fact, the users of `LinkedList` should have **no** idea that we've switched the implementations on them! All the test code should work as before, and all details about nodes will only be known to your implementation.

In this lab, the goal is to achieve constant time complexity (written as `O(1)` time) for as many of the methods as possible. Meaning, their running time should not depend on the length of the list. We will specify which methods should be `O(1)`, and we will be testing for this!

Part 1: Initial Implementation

Your linked list will consist of Nodes that are "chained" together, each referencing the next. Each Node has a `value` and a `link` to the next node in the chain. Each Node only has one method, `__init__`. The Nodes are members of `class _Node`, which is private to the `LinkedList` class. Meaning, it will be defined **inside** the `LinkedList` class.

```
class LinkedList:

    # the _Node class name is local to the LinkedList class
    class _Node:
        # this is a method of the _Node class
        def __init__(self, item, link = None):
            self.value = val
            self.link = link

    # this is a method of the LinkedList class
    def SillyTest(self):
        # this creates a Node that holds the value “foo”
        # and stores it in the _head attribute of the LinkedList
        self._head = self._Node(“foo”)
        print(self._head.value)
```

For the initial implementation, start by defining the `Node` class and the `LinkedList` class, as well as `__init__` methods for each.

Add the `addFirst` method. `myList.addFirst(value)` will create a new `Node` holding this value, and will modify the `_head` attribute in the `LinkedList` class so it references this node. Also, for `LinkedList`, there will be `__str__` method that returns the list of values with brackets and semicolons, as before. There will also be the `__len__` magic method that returns the number of nodes in the list.

```
myList = LinkedList()

len(myList)                    # returns 0
print(myList)                  # prints []

myList.addFirst("apple")
myList.addFirst("grape")
print(myList)                  # prints ['grape'; 'apple']
len(myList)                    # returns 2
```

Remember, in this lab, the goal is to achieve `O(1)` time complexity for as many of the methods as possible.

While we cannot accomplish this for `__str__`, which will need to traverse the entire list to print all the values, we can accomplish it for `__len__`. Instead of traversing the list to count the nodes, it will just return the value of a `_nodeCount` attribute of the linked list. This means you need to remember to update this attribute whenever you add or remove nodes!

Part 2: More O(1) methods

Now add `LinkedList.addLast` method. It will create a new `Node` holding the new value, and will change the last node in the list so instead of referencing `None` as its next node, it will reference the new one (which will in turn reference `None`).

Remember, we want it to run in `O(1)` time, so we can't traverse the list just to find where the last node is! So we need a new `_tail` attribute in the `LinkedList` class, that stores the reference to the last node of `LinkedList`. This will allow us to find the last node right away! You will need to reset this attribute when you add a new last node, so it references this new one.

```
myList = LinkedList()
myList.addLast("apple")
myList.addFirst("grape")
myList.addLast("banana")
print(myList )                 # prints ['grape'; 'apple'; 'banana']
len(myList)                    # returns 3
```

Let's also add the `removeFirst` method. It too should be `O(1)`.

The last method we will implement here is `append`. This method appends a second list to the end of the first one. Afterwards, all the elements are in the first list, and the second one is empty:

```
otherList = LinkedList()
otherList.addLast("cake")
otherList.addFirst("cookie")
myList.append(otherList)

print(myList)           # prints ['grape'; 'apple'; 'banana'; ‘cookie’; ‘cake’]
len(myList)             # returns 5
len(otherList)          # returns 0
print(otherList)        # prints []
```

Believe it or not, this method is also `O(1)`! It is **very** similar to `addLast`, adding the first node of `list2` at the end of `list1`. Then the `_head` attribute of list2 is set to `None`.

Remember to make sure that all these methods update the `_nodeCount`, `_head`, and `_tail` attributes of `LinkedList` appropriately!

Part 3: Iteration

As before, we want to iterate through our `LinkedList`. So we need to create an iterator class as before as well as the `__iter__` and `__next__` magic methods. Just to make things comprehensive, in Python every iterator is also an iterable which means that you should implement `__iter__` in the iterator class as well. In this case the method should return self.

```
myList = LinkedList()
myList.addLast("apple")
myList.addFirst("grape")
myList.addLast("banana")

for fruit in myList:
    print(fruit)            # prints 'grape', then 'apple', then 'banana'
```

These magic methods should be `O(1)` as well! This means that the iterator cannot store the index of the current value, as in Lab 03! Instead, it must store a reference to the current `Node`.

Part 4: Filling Out

Let's add everything else that we've implemented for `LinkedList` in Lab 03:

```
removeLast
addAt
removeAt
```

and magic methods

```
__contains__
__getitem__
__setitem__
```

Note that these methods are not `O(1)`. When linked lists are implemented with chained nodes, if we want to find the node at some position i, then we need to actually traverse the list through i-1 nodes before we can get to the i^{th} node! (Unless i is at the very beginning or end of list, of course).

Make sure that `removeLast`, `addAt` and `removeAt` methods update the `_nodeCount`, `_head`, and `_tail` attributes of LinkedList appropriately!

Discussion: Doubly Linked Lists

Note: the reason we cannot implement `removeLast` in `O(1)` is because we do not have direct access to the node **before** the last one. You might try to add that as **another** attribute of `LinkedList` class, but if you do, you will see that it still won't give you `O(1)` time complexity! The only way to achieve that is by adding a new attribute to the `Node` class, which is a reference to the **previous** node. When all nodes reference both the node before and after them, the resulting data structure is known as a **doubly linked list**.