1. (20 pts) Implement add\_last and find\_min in the Node class below using recursion such that the LinkedList class (which you should not edit) works as expected.
2. class Node:  
    def \_\_init\_\_(self, data, link=None):  
    self.data = data  
    self.link = link  
     
    #######################################################  
    # All methods below should be implemented recursively #  
    #######################################################  
    def add\_last(self, item):  
    *'''Recursively adds last node to a List'''* if self.link is not None:  
    self.link.add\_last(item)  
    else:  
    self.link = Node(item)  
     
    def find\_min(self):  
    *'''Recursively find the minimum by comparing itself to the link'''* if self.link is None:  
    return self.data  
    else:  
    return min(self.data, self.link.find\_min())  
     
     
   #########################################################  
   # No changes below this point - all your work should be #  
   # in Node. #  
   #########################################################  
   class LinkedList:  
    def \_\_init\_\_(self, items=None):  
    self.\_head = None  
    if items is not None:  
    for item in items:  
    self.add\_last(item)  
     
    def add\_last(self, item):  
    # Edge case - first node added  
    if self.\_head is None:  
    self.\_head = Node(item)  
    else:  
    self.\_head.add\_last(item)  
     
    def find\_min(self):  
    if self.\_head is None:  
    raise ValueError("find\_min() on empty linked list")  
    return self.\_head.find\_min()
3. (20 pts) The following code implements a Queue using a linked list. HOWEVER, the code is not complete. You need to develop some test code (not unittest) to show that the code behaves as expected and fully implements all functionality of a Queue:

* enqueue(item)
* dequeue(item)
* peek() – Look at the next item in the Queue
* isempty()
* Be able to print the length of the Queue
* Be able to print a listing of the Queue

BUT, there may very well be an error when you attempt test all of this functionality – missing methods. You will need to build those methods.

class ListNode:  
 def \_\_init\_\_(self, data, link = None):  
 self.data = data  
 self.link = link  
  
class LinkedList:  
 def \_\_init\_\_(self):  
 self.\_head = None  
 self.\_tail = None  
 self.\_length = 0  
 def addfirst(self, item):  
 self.\_head = ListNode(item, self.\_head)  
 if self.\_tail is None:  
 self.\_tail = self.\_head  
 self.\_length += 1  
 def addlast(self, item):  
 if self.\_head is None:  
 self.addfirst(item)  
 else:  
 self.\_tail.link = ListNode(item)  
 self.\_tail = self.\_tail.link  
 self.\_length += 1  
 def removefirst(self):  
 item = self.\_head.data  
 self.\_head = self.\_head.link  
 if self.\_head is None:  
 self.\_tail = None  
 self.\_length -= 1  
 return item  
  
 def removelast(self):  
 if self.\_head.link is None:  
 return self.removefirst()  
 else:  
 currentnode = self.\_head  
 while currentnode.link.link is not None:  
 currentnode = currentnode.link  
 item = self.\_tail.data  
 self.\_tail = currentnode  
 self.\_tail.link = None  
 self.\_length -= 1  
 return item  
  
class LinkedQueue:  
 def \_\_init\_\_(self):  
 self.\_L = LinkedList()  
 def enqueue(self, item):  
 self.\_L.addlast(item)  
 def dequeue(self):  
 return self.\_L.removefirst()  
 def peek(self):  
 item = self.\_L.removefirst()  
 self.\_L.addfirst(item)  
 return item  
 def \_\_len\_\_(self):  
 return len(self.\_L)  
 def isempty(self):  
 return len(self) == 0

Describe what methods you needed create and provide a listing of just those methods below:

I needed a \_\_iter\_\_ and a \_\_len\_\_ magic method for LinkedList(), I needed a \_\_str\_\_ magic method for LinkedQueue. \_\_iter\_\_ allowed me to iterate over the linkedlist, while \_\_len\_\_ allowed me to define what a \_\_len\_\_ magic method should do.

def \_\_iter\_\_(self):  
 node = self.\_head  
 while node is not None:  
 yield node.data  
 node = node.link  
  
def \_\_len\_\_(self):  
 return self.\_length

def \_\_str\_\_(self):  
 return ' '.join(str(item) for item in self.\_L)

Providing the listing of your code that tests full functionality of your code below this line:

queue = LinkedQueue()  
#Test enqueue  
queue.enqueue('1')  
print(queue) # 1  
queue.enqueue('2')  
queue.enqueue('3')  
print(queue) # 1 2 3  
  
# Test dequeue  
print(queue.dequeue())  
print(queue) # 1  
  
# Test peek  
print(queue.peek())  
print(queue) # 2 3  
  
# Test length of the queue  
print(len(queue)) # 2  
  
# Test isempty  
print(queue.isempty()) # False  
  
# Empty the queue  
queue.dequeue()  
queue.dequeue()  
  
# Test isempty  
print(queue.isempty()) # True  
  
# Empty queue output  
print(queue) # Nothing

1. (20 pts) Write a function using the provided classes that sorts a List of elements by:

* Starting at the beginning of the list
* If an element is less than the previous one, then move the previous element to the end of the list.
* It does not move on, though. It checks whether the new previous item is greater and moves it to the end of the list if it is greater. It continues this until it gets to the beginning of the list.
* It continues in this manner until it checks the last item in the list against the next-to-last item.

So, if the list given were: [5, 7, 3, 6, 8, 1], the following would be this list as each swap takes place:

[5, 3, 6, 8, 1, 7] move 7 to the end (it is greater than 3)

[3, 6, 8, 1, 7, 5] move 5 to the end (it is greater than 3)

[3, 6, 1, 7, 5, 8] move 8 to the end (it is greater than 1)

[3, 1, 7, 5, 8, 6] move 6 to the end (it is greater than 1)

[1, 7, 5, 8, 6, 3] move 3 to the end (it is greater than 1)

[1, 5, 8, 6, 3, 7] move 7 to the end (it is greater than 5)

[1, 5, 6, 3, 7, 8] move 8 to the end (it is greater than 6)

[1, 5, 3, 7, 8, 6] move 6 to the end (it is greater than 3)

[1, 3, 7, 8, 6, 5] move 5 to the end (it is greater than 3)

[1, 3, 7, 6, 5, 8] move 8 to the end (it is greater than 6)

[1, 3, 6, 5, 8, 7] move 7 to the end (it is greater than 6)

[1, 3, 5, 8, 7, 6] move 6 to the end (it is greater than 5)

[1, 3, 5, 7, 6, 8] move 8 to the end (it is greater than 7)

[1, 3, 5, 6, 8, 7] move 7 to the end (it is greater than 6)

[1, 3, 5, 6, 7, 8] move 8 to the end (it is greater than 7)

Finish because you are at 7 now and that is at the next-to-last element.

You are permitted to add additional attributes to the classes. However, **you may not** store the Nodes in a list. **You may not use** the list type for anything for full credit. If you use a List only as a container, then you will get significantly less partial credits.

Here are the classes to use:

class ListNode:

def \_\_init\_\_(self, data, \_prev = None, \_next = None):

self.data = data

self.\_prev = \_prev

self.\_next = \_next

if \_prev is not None:

self.\_prev.\_next = self

if \_next is not None:

self.\_next.\_prev = self

class DoublyLinkedList:

def \_\_init\_\_(self):

self.\_head = None

self.\_tail = None

self.\_length = 0

Insert your code after this line:

def add\_last(self, item):  
 new\_node = ListNode(item)  
 if self.\_tail is None:  
 self.\_head = self.\_tail = new\_node  
 else:  
 new\_node.\_prev = self.\_tail  
 self.\_tail.\_next = new\_node  
 self.\_tail = new\_node  
 self.\_length += 1  
  
def \_\_str\_\_(self):  
 node = self.\_head  
 nodes = []  
 while node is not None:  
 nodes.append(str(node.data))  
 node = node.\_next  
 return ' -> '.join(nodes)  
  
def custom\_sort(self):  
 node = self.\_head  
 while node is not None and node.\_next is not None:  
 if node.data > node.\_next.data:  
 node.data, node.\_next.data = node.\_next.data, node.data  
 node = self.\_head  
 else:  
 node = node.\_next