Lab 4: Recursion and Linked Lists

"Traversal" refers to visiting every piece of data in a data structure. Here's linked list traversal with iteration:

```
def traverse(self) :
  tempPtr = self.head
  while tempPtr is not None:
    tempPtr = tempPtr.nextNode
```

and here it is with recursion:

```
def traverse(self) :
  self.__traverse(self.head)

#This is a helper method, so our user doesn't have to know or care about
#nodes.
def __traverse(self,node) :
  if node is None: #Empty lists make this easy, so its a good base case
    return
  #Do something here, for example print
  self.__traverse(node.nextNode)
```

Make sure you understand how this works. Soon, there will be data structures where the iterative version is pages long, and the recursive version is five lines. You want to know and understand how this works.

Traversal algorithms can be used not only to perform actions on the linked list (like printing) but also answer questions about it. For this to work, both the recursive function and the helper function should return a value. For example, consider the following function that calculates the length of a linked list:

```
def length(self):
  return self.__length(self.head)

def __length(self,node):
  if node is None:
    return 0
  return 1+self.__length(node.nextNode)
```

It works because each recursive call is asking, "what is the length of the linked list, starting at my next node?" If it can get that answer, then the length of the list including the node that asked the question is just that plus one. Each node asks that question of the node following it, until it gets to `None`, signifying the end of the list. At that point the answer to the question, asked by the very last node in the list, is zero (because there are no nodes after the last one). The last node now knows that it is the end of the list and can tell the node before it the correct answer, who can tell the node before it, etc., until the final answer propagates all the way back to the beginning.

We can also write functions that both do an action on the list *and* return a value. One of these functions is `addInOrder` from our previous linked list lab.

The way this function works is that each recursive call asks the next one who they should point to. This is because, thinking recursively, the workers (function calls) after us might make some change to the list. They need a way to signal back to us what `nextNode` should point to.

```
def addInOrder(self, element):
  #Start with the head, begin recursion
  self.head = self.__addInOrder(element, self.head);
  if self.tail is None:
    self.tail = self.head

def __addInOrder(self, element, node):
  #Check if we are at the end of the list, add here if we are
  if node is None:
    return Node(element)
  #Check if the element we want to insert belongs in front of this node
  if node.data > element:
    t = Node(element) #Create a new node
    t.nextNode=node   #Set its nextNode to the node we are looking at
    return t;         #Return the new node back to the previous level of recursion
  #If we haven't found the right spot, call recursively on the next node
  #Whatever the recursion returns, we set it to our nextNode, because they
  #might have changed the list and need to notify us who to point to next
  node.nextNode= self.__addInOrder( element, node.nextNode )
  return node
```

Make some examples on paper to understand why it works!

The Assignment

Copy-paste this file (labs/linkedlist.txt) into a file called `linkedlist.py` and complete it. Your constructor and add2Front obviously don't require recursion. All other methods, however, should use recursion and helper methods. `printInOrder` has the beginnings of a helper method built for you. **Do NOT use any for or while loops in this lab!** All repeating functionality should be done with recursion.

Test as you go.