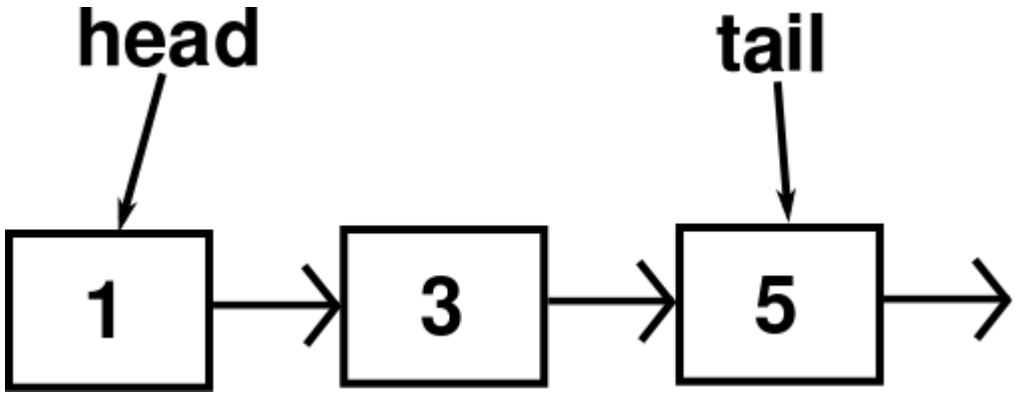
Lab 2: Linked Lists

One data structure that we're going to talk about a lot is a Linked List. A Linked List is a pretty simple thing, and can be good practice of Object-Oriented Programming.

Linked Lists are essentially alternatives to arrays. Like with an array, each piece of data has a spot, and each spot has a single piece of data. It's built in a pretty different way, though.

Linked Lists are built from a series of Nodes, where a Node holds two things: (1) a single piece of data, and (2) a pointer to the next Node in the list. Below, for example, is a drawing of a linked list containing the integers 1, 3, and 5. Each rectangle is a Node, and each arrow is a pointer to the next Node. The value of the pointer in the Node containing 5 is None (Python's null equivalent), because it's not pointing anywhere.



The first Node in the Linked List is commonly referred to as the "head," while the last is referred to as the "tail."

The Assignment

To build a Linked List, you're going to make two classes, Node and LinkedList, both of which are in a the file `linkedlist.py`.

Node

Your Node class will have two fields, one constructor, and no additional methods. Your fields will be:

- `data`: For holding the integer within the Node
- `nextNode`: For holding the pointer to the next Node

Your constructor will take as an argument the data, and should set that field accordingly. The nextNode field should be set to "None."

LinkedList

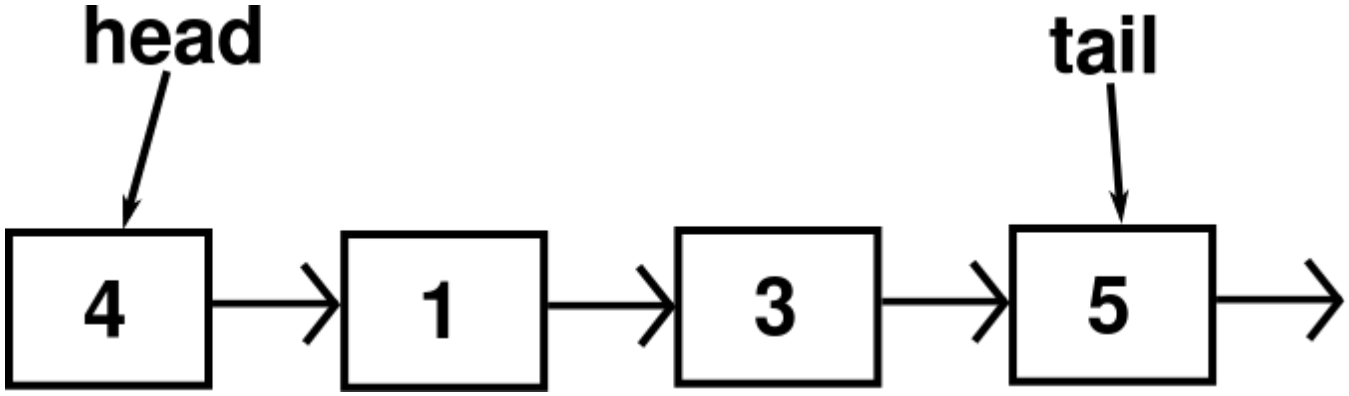
This class will have two fields, one constructor, and five methods. Your fields will be:

- `head`: For pointing to the first link in the list (the one containing 1 in our example)
- `tail`: For pointing to the last link in the list (the one containing 5 in our example)

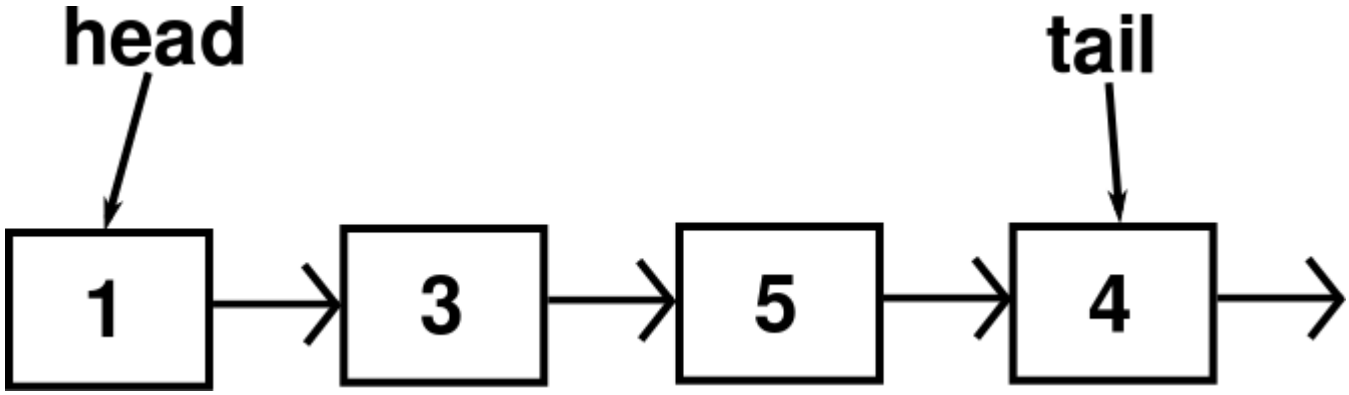
Your constructor should take no arguments (aside from `self`), and should set head and tail to None, indicating this new LinkedList is empty, and therefore has no head or tail Nodes.

Finally, you'll have five methods which alter your LinkedList, or display information about it:

- `add2Front(self,data)`: The data input as an argument should be put inside a new Node, which is then stuck on the front of the Linked List as the new head. In our example, calling myLL.add2Front(4) should result in this:



- `add2Back(self,data)`: The data input as an argument should be put inside a new Node, which is then stuck on the back of the Linked List as the new tail. In our example, calling myLL.add2Back(4) should result in this:



- `__str__(self)`: Return all the data within the Linked List as a string. On our example, calling this will result in the string `1 3 5`. Calling `str` on an empty list should return an empty string.
- `isIn(self,data)`: Calling myLL.isIn(someNumber) should return True if someNumber is contained within a Node somewhere in this LinkedList, and False if it is not.
- `addInOrder(self,data)`: For this method, you may assume the elements of your Linked List are already in increasing order. This method takes `data` and adds it in the appropriate place. For example, if your LL has 1,3,5, and you call myLL.addInOrder(4), it ends up as 1,3,4,5.

Your Approach

When using OOP, you'll find it's most effective to build these things up in small pieces, **testing as you go**. In this case, LinkedList can't do much unless you have Node written, so start there. Build a Node class, and then write some code to test it. For example, here's some testing code for the Node class:

```
from linkedlist import Node

node1=Node(3)
node2=Node(4)
print(node1.data) #should print 3
print(node2.data) #should print 4
node1.data=5
print(node1.data) #should print 5
node1.nextNode=node2
print(node1.nextNode.data) #should print 4, see why?
```

Once you're confident in your Node class, you can work on your LinkedList class, assuming that if anything breaks, it's in LinkedList, and not Node. It is tempting to shoot in the dark, program your whole program, and then debug. However, I've written very, very few programs that work on their first try. If you test your methods as you go, then you keep it so any errors that come up, can only be in the most recently written few lines. **This makes your life much, much easier.**

add2Front

For this lab, you will have to think about the structure of your linked list when designing each function. Consider what you want to accomplish and what steps you have to take to accomplish it, given the way the linked list is organized. For example, to add a node to the front of a linked list you have to adjust the data structure to accommodate the new node. If it is going on the front of the list, the `head` variable should be set to the new node. Breaking it down, there are four steps that need to be done:

1. Create a new `Node` object with the given data value.
2. Set the `nextNode` field of the new node to `self.head`. This reflects the fact that the new node you are adding is going onto the front of the list and so the node directly following it will be the one that used to be at the front of the list.
3. Set the `self.head` variable equal to the new node, making this the new start of the list.
4. **Important:** if the `self.tail` variable is `None`, that means that this is the first node being added to the list. Here there is a special case: if only one node is in the list it must be both the head *and* tail. If so, set `self.tail` to the new node.

Iterating the list

A common task for linked lists is to *iterate* through them; that is, starting with the first node go through the entire list and do something with the nodes one at a time. For instance, the `__str__` function iterates through the list and adds each data element to a string. With a normal list, we would do this with a for loop. However, linked lists do not work with for loops. Instead we will use a variable (also sometimes called an *iterator*) to move one by one through each node's `nextNode`, starting with the head. We know we have reached the end of the list when that iterator is equal to `None`, since the last node in a linked list has its `nextNode` field set to `None`. A common iterator function looks like this:

```
def iterator(self):
   cur = self.head
   while cur != None:
      #Do something with cur, i.e. print it or add its value to a string
      cur = cur.nextNode
```