Lab 9

Linked Lists part 2

Background

In this lab we'll explore more about linked lists. Linked list is such a fundamental structure that is actually a backbone for many other extremely important data structures such as trees and graphs etc.mm

Pre-requisites

From your file explorer navigate to the desktop and create a folder called "`lab09`". Extract the contents of "`lab09_starter.zip`" in this folder. The starter package contains the following files and folders:

1. ***`lab06.py:`***

   You'll implement your code here and submit this file.

2. ***`grade:`***

   This file performs automated grading on your solutions.

3. ***`requirements.txt:`***

   It is used to install prerequisite packages.

Type the following command in the command prompt to make sure python 3+ is installed and working as expected:

```
c:\> python --version
```

Following command installs all of the required packages if they're not already installed:

```
c:\> python -m pip install -r requirements.txt
```

Open up `lab09.py` and write your name, roll no, section and date on lines 3 to 6 respectively located at the top of the file:

```
# Name   :
# Roll no:
```

```
# Section:
# Date   :
```

Linked Lists part 2

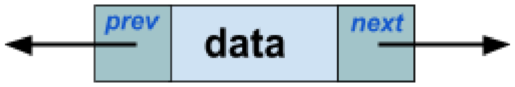
We continue from where we left in lab 08. In this lab we'll work with doubly linked lists.

Task L01: Doubly Node Class

Look at the starter code, you will find the node class. This is the most basic element in a linked list.

```
class Node:
   def __init__(self):
       pass
```

A node in the doubly linked list has a value, which can be an object even, a reference to the previous node, and a reference to the next as shown in the diagram below:



Let's create a node class first. Open up the "`lab09.py`" file and find the `Node` class. Add the following instance variables to the class as shown in the code below:

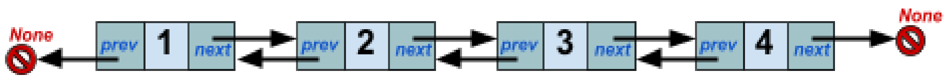
```
class Node:
   def __init__(self, data, prev = None, next = None):
       self.data = value
       self.prev = None
       self.next = None
```

Once you've completed this, run the autograder to check task L01:

```
> python grade
```

Task L02: Linking Multiple Nodes

In this task you will create a doubly linked list of ascending numbers `1,2,3,4` as shown below:



Go to the `TaskL02()` function. Add code to create the doubly linked list shown in the figure above. Note that the first node is already created for you. You have to add the remaining three.

Note: *There are NO sentinel (header or trailer) nodes in this doubly linked list. The first node is the head node which actually contains data.*

```
def TaskL02():
        head = Node(1)                  # DO NOT MODIFY THIS LINE
        #==================
        # Insert code here:
        #------------------
        pass
        #==================
        return head                     # DO NOT MODIFY THIS LINE
```

To check your logic, launch python and type the following:

```
$ python
Python 3.9.6 (default, Sep 26 2022, 11:37:49)
....
>>> from lab06 import *
>>> head = TaskL02()
>>> head.data
1
>>> n2 = head.next
>>> n2.data
2
>>> head.next.next.data
3
>>> n2.next.data
3
>>> head.next.next.prev.data
2
>>> print(head.next.next.next.next)
None
>>>
head.next.next.prev.next.prev.next.prev.next.prev.next.prev.data
2
```

After verifying your function, run the autograder to check task L02:

```
> python grade
```

Task L03: Find Node Having Some Data

Assume a linked list containing unique data items in its nodes. You are to find and return the reference of the node that contains given “`data`”. The `TaskL02(head,data)` function accepts two parameters namely “`head`” of the doubly linked list and “`data`” to be found. This function then finds the node that contains the data and returns its reference. If data is not found in any of the nodes then it returns `None`:

```
def TaskL03(head, data):
        node = None                 # DO NOT MODIFY THIS LINE
        #===================
        # Insert code here:
        #-------------------
        pass
        #===================
        return node                    # DO NOT MODIFY THIS LINE
```

Don’t forget the special case: The empty linked list !

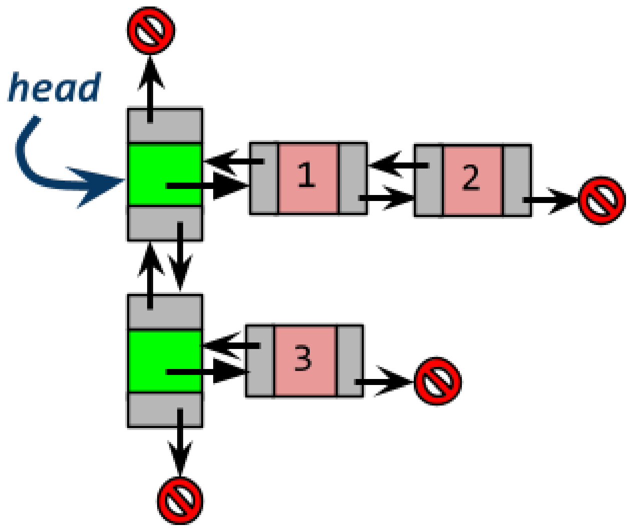
Task L04: Appending a List After Another

The function `TaskL04(L1, L2)` accepts two list heads as parameters. It appends `L2` after `L1` and returns the head of the merged list.. For example if `L1` is `1-2-3` and `L2` is `4-5-6`, then this function will return the head pointer of the doubly list `1-2-3-4-5-6`.

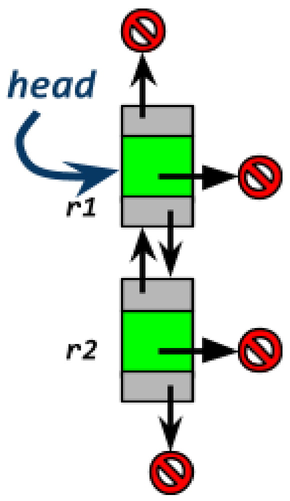
```
def TaskL04(L1, L2):
        #===================
        # Insert code here:
        #-------------------
        pass
        #===================
        return L1                    # DO NOT MODIFY THIS LINE
```

Task L05: Create a Nested Structure By Hand

In this lab you will create the following nested structure manually by hand:



Open up `TaskL05()`. Let’s first create the green list:



The following code creates this green list as shown in the diagram above:

```
def TaskL05():
   head = None                  # DO NOT MODIFY THIS LINE
    #----------------------
   head = Node(None)        # set None as the data
    r2 = Node(None)          # set None as the data
    head.next = r2
    r2.prev = head
    #----------------------
    return head                   # DO NOT MODIFY THIS LINE
```

Let’s fire up python, load the lab06 module and call the Task06 function. You can now watch the values of this structure using the head pointer:

```
>>> from lab06 import *
>>> h = TaskL05()
>>> print(h.data)
None
>>> h.next
<lab06.Node object at 0x7fdd233fc5e0>
>>> print(h.next.data)
None
>>> print(h.next.next)
None
```

The values are matching our figure.
After verifying your function, run the autograder to check task L02:

```
> python grade
```

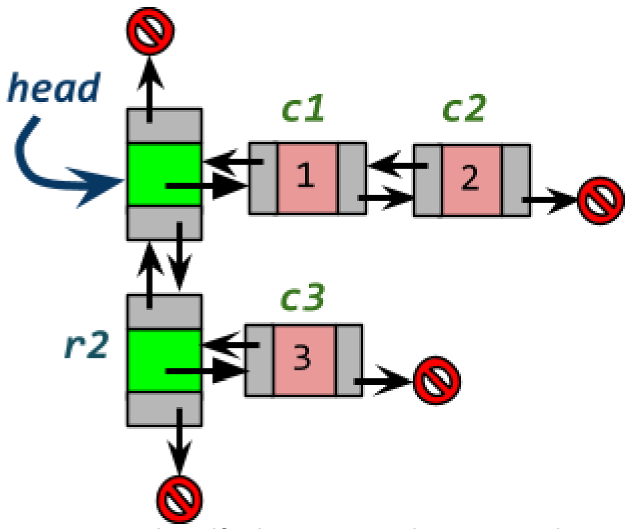
Looks like we’re on the right track. Now let’s extend this structure.

Task L06: Extending the Nested structure

Let’s extend the previous structure and add nested pink rows to each of the green nodes. First open up the `Task06()` function and type the following code to continue from the previous task:

```python
def TaskL06():
   head = None                     # DO NOT MODIFY THIS LINE
   #-------------------
   head = TaskL05()          # Incomplete structure from TaskL05

   # First pink row:
   c1 = Node(1)
   c2 = Node(2)
   c1.next = c2
   c2.prev = c1

   # c1 acts as the head of the first pink row
   head.data = c1  # NOTE: we're putting the row's head in data
   c1.prev = head

   # Second pink row:
   c3 = Node(3)

   r2 = head.next
   # c3 acts as the head of the second pink row
   r2.data = c3    # NOTE: we're putting the row's head in data
   c3.prev = r2
   #===================
   return head                     # DO NOT MODIFY THIS LINE
```

The following figure shows the structure after the code above has executed:



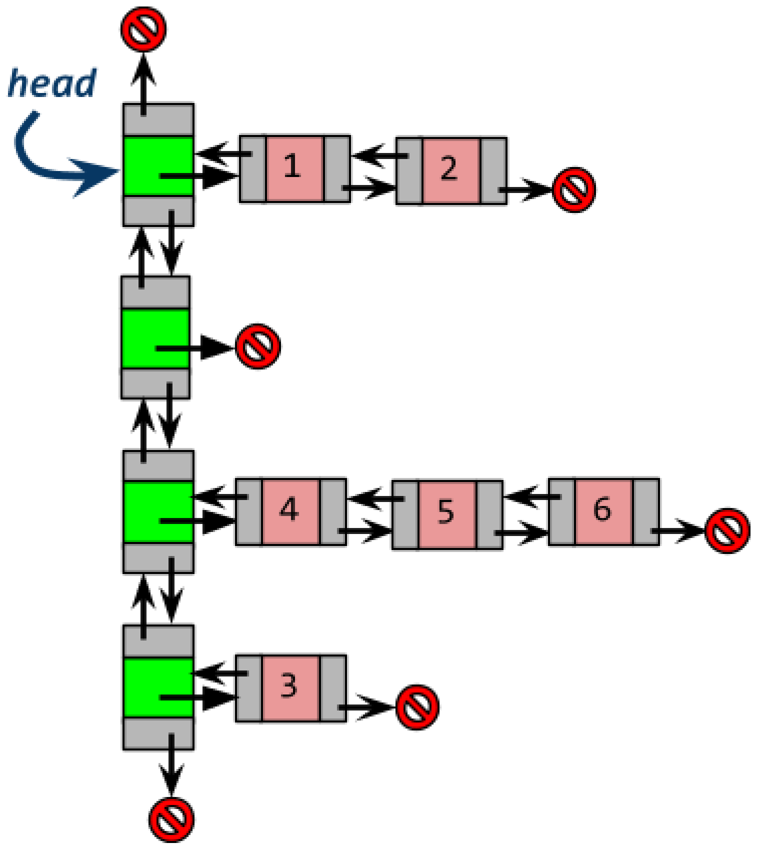
Launch python interpreter and verify that your code creates the structure above.
Let's try to navigate this structure:

```
>>> from lab06_dev import *
>>> h = TaskL06()
>>> h.data          # First Pink Row
<lab06_dev.Node object at 0x7f4840b52760>
>>> h.data.data        # c1.data
1
>>> h.data.next.data   # c2.data
2
>>> h.next.data.data   # c3.data
3
```

```
>>> print(h.next.data.next)   # c3.next
None
```

Task L07: Another Custom Structure

Now you know how to create custom structures. Consider the following diagram:



Open up `TaskL07()` function and write code to create the structure shown in the above diagram:

```
def TaskL07():
    head = None                    # DO NOT MODIFY THIS LINE
    #===================
    # Insert code here:
    #-------------------
    pass
    #===================
    return head                    # DO NOT MODIFY THIS LINE
```

Task L08: Counting Rows

Given a nested structure, write code to count the total number of non-empty rows in the structure. TaskL08(head) takes the nested head as a parameter and returns an integer number of non-empty pink rows:

```
def TaskL08(head):
    #===================
    # Insert code here:
    #-------------------
    pass
    #===================
```

After you've implemented you can check the above function by reusing the structure that we implemented in task L07. Go to the python interpreter and write the following piece of code:

```
>>> from lab06_dev import *
>>> head = TaskL07()
>>> Task08(head)
3
```

Task L09: Counting Pink Nodes

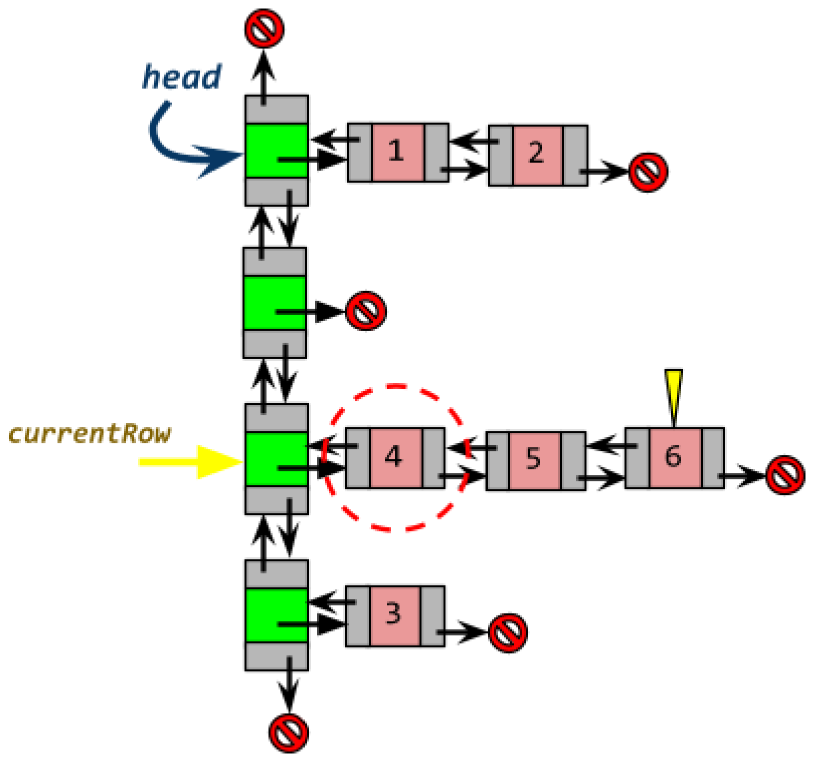
Count the total number of pink nodes in a given nested structure. This function hates the root of the nested structure and returns an integer number of pink nodes:

```
def TaskL09(head):
    #===================
    # Insert code here:
    #-------------------
    pass
    #===================
```

Task L10: Get The Head Of A Current Row

Given a reference to a random pink node *(the yellow triangle in the figure below)*, your task is to determine the first node or the head of that particular row. For example, given the reference of a random node with value `6` below *(yellow triangle)*, you are required to find and return the head node of that particular row. In this case that'll be the node containing value `4`, circled with red color.

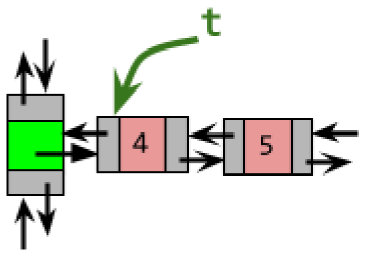
Hint:
You may need a dedicated "`currentRow`" pointer for easy navigation and a way to stop at the left end. This is also passed as a parameter in the function:



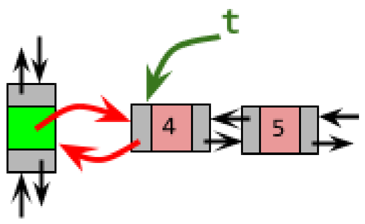
```
def TaskL10(head, node, currentRow):
    '''
    Parameters:
        heaa - the root reference of the entire nested structure
        node - the random node in a row to begin with
        currentRow - Reference of the GREEN node that contains
this row
    '''
    rowHead = None            # DO NOT MODIFY THIS LINE
    #==================
    # Insert code here:
    #------------------
    pass
    #==================
    return rowHead                    # DO NOT MODIFY THIS LINE
```

Task L11: Get The Head Of A Current Row (***A Better Way***)

We can find the head node without any currentRow pointer or even storing a color within the nodes. Imagine that a temporary variable "t" points to the head node of the row as shown below. The question is, how do you know this is the head node?



Take a moment to think about what condition should be true here? *(specially note the red arrows)*



Here, the head of a row is the only place where the following condition is true:

```
t.prev.data == t
```

Now we can easily know the head of the current row because we know exactly where to stop! Given a random node in a row, complete the following function to return the head node of that row. Note that this time there are only two parameters:

```
def TaskL11(head, node):
    rowHead = None           # DO NOT MODIFY THIS LINE
    #===================
    # Insert code here:
    #-------------------
    pass
    #===================
    return rowHead           # DO NOT MODIFY THIS LINE
```