Lab 4 (10 points) **Due date:** June 25, 2023, 11:59 PM ET

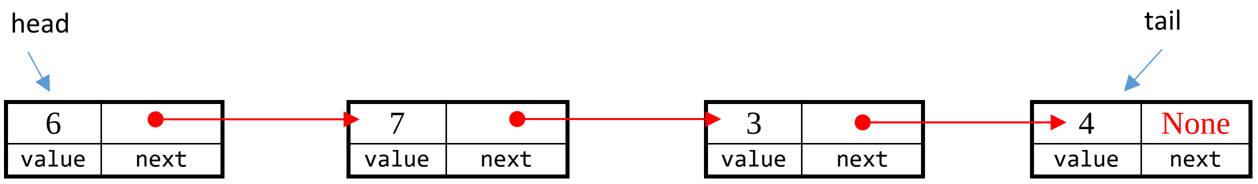
Before you start…

Keep in mind these tips from your instructor:

- Watch the lectures in Module 5 before you start this assignment, especially the hands-on videos
- Outline a strategy before you start typing code and share it with the course staff for additional support
- When it comes to linked list structures, paper and pen/pencil are just best friends. We encourage you to draw out the sequence of nodes to visualize the references you need to update/know/create to achieve the goal of each method
- Make sure you update the head and tail pointers according to the operation performed
- Starter code contains the special methods __str__ and __repr__. Do not modify them, use them to ensure your methods are updating the elements in the list correctly
- Instances of the Node class have two attributes: value and next. If you intend to use rich comparison operator such as == , <, >, <= or >= make sure you are using the value of the Node, not the object itself. The Node class does not have the special methods to overload those operators, but you are welcome to define them!
- Ask questions using our Lab 4 channel in Microsoft Teams

Practice: Using the Singly Linked Lis below, where head references the first node and tail references the last node, write the statements to do the following:

a) Insert 11 before 6
b) Remove 4
c) Insert 20 before 3



REMINDER: As you work on your coding assignments, it is important to remember that passing the examples provided does not guarantee full credit. While these examples can serve as a helpful starting point, it is ultimately your responsibility to thoroughly test your code and ensure that it is functioning correctly and meets all the requirements and specifications outlined in the assignment instructions. Failure to thoroughly test your code can result in incomplete or incorrect outputs, which will lead to deduction in points for each failed case.

Create additional test cases and share them with you classmates on Teams, we are in this together!

90% > Passing all test cases   10% > Clarity and design of your code

IMPORTANT: You are not allowed to use break or continue statements, swap data between nodes or to copy data from the linked lists into another structure such as a Python list (your code should not use any Python lists) or using any Python built-in copy methods.

The SortedLinkedList class

In our Hands On – Linked List I and II lecture from Module 5, we worked on the implementation of a Singly Linked List. That data structure keeps the elements of the linked list unsorted. Based on the LinkedList class code completed during the video-lecture, implement the data structure SortedLinkedList with the following methods:
Tip: Take the time to outline a strategy for each method (drawing the linked list is extremely helpful!), and reuse code by calling methods you already implemented.

add(self, item) (2 points)

adds a new Node with value=item to the list making sure that the **ascending** order is preserved. It needs the item and returns nothing but modifies the linked list. The items in the list might not be unique, but you can assume every value will be numerical (int and float).

- You are not allowed to traverse the linked list more than once (only one loop required).

```
Preconditions and Postconditions
item: int/float -> numerical value
Returns: None
```

Examples:

```
>>> x=SortedLinkedList()
>>> x.add(8.76)
>>> x.add(7)
>>> x.add(3)
>>> x.add(-6)
>>> x.add(58)
>>> x.add(33)
>>> x.add(1)
>>> x.add(-88)
>>> print(x)
Head:Node(-88)
Tail:Node(58)
List:-88 -> -6 -> 1 -> 3 -> 7 -> 8.76 -> 33 -> 58
```

split(self) **(2 points)**

Divides the original list in half, returning two new instances of SortedLinkedList, one with the first half, and another one with the second half. If the number of nodes is odd, the extra node should go in the first half. It does not mutate the original list. Method returns None if the original list is empty.

```
Postconditions
Returns: None -> if the original list is empty
         SortedLinkedList, SortedLinkedList -> two linked list objects
```

Examples:

```
>>> my_lst = SortedLinkedList()
>>> my_lst.add(5)
>>> my_lst.add(1)
>>> my_lst.add(6)
>>> my_lst.add(9)
>>> my_lst.add(8)
>>> my_lst
Head:Node(1)
Tail:Node(9)
List:1 -> 5 -> 6 -> 8 -> 9
>>> sub1, sub2 = my_lst.split()
>>> sub1
Head:Node(1)
Tail:Node(6)
List:1 -> 5 -> 6
>>> sub2
Head:Node(8)
Tail:Node(9)
List:8 -> 9
>>> sec_split1, sec_split2 = sub2.split()
>>> sec_split1
Head:Node(8)
Tail:Node(8)
List:8
>>> sec_split2
Head:Node(9)
Tail:Node(9)
List:9
>>> lst = SortedLinkedList()
>>> lst.add(5)
>>> lst.split()
(Head:Node(5)
Tail:Node(5)
List:5, Head:None
Tail:None
List:)
```

removeDuplicates(self) **(2 points)**

Removes any duplicate nodes from the list, so it modifies the original list. This method must traverse the list only once. Values such as 2 and 2.0 are considered duplicates, so you can remove either one.

Postconditions

Returns: None

Examples:

```
>>> x=SortedLinkedList()
>>> x.removeDuplicates()
>>> x
Head:None
Tail:None
List:
>>> x.add(1)
>>> x.add(1)
>>> x.add(1)
>>> x.add(1)
>>> x
Head:Node(1)
Tail:Node(1)
List:1 -> 1 -> 1 -> 1
>>> x.removeDuplicates()
>>> x
Head:Node(1)
Tail:Node(1)
List:1
>>> x.add(1)
>>> x.add(2)
>>> x.add(2)
>>> x.add(2)
>>> x.add(3)
>>> x.add(4)
>>> x.add(5)
>>> x.add(5)
>>> x.add(6.7)
>>> x.add(6.7)
>>> x
Head:Node(1)
Tail:Node(6.7)
List:1 -> 1 -> 2 -> 2 -> 2 -> 3 -> 4 -> 5 -> 5 -> 6.7 -> 6.7
>>> x.removeDuplicates()
>>> x
Head:Node(1)
Tail:Node(6.7)
List:1 -> 2 -> 3 -> 4 -> 5 -> 6.7
```

intersection(self, other) **(2 points)**

Given two instances of SortedLinkedList, returns a new instance of SortedLinkedList that contains the nodes that appear in both lists. The intersection should not keep duplicates. You are not allowed to alter the original lists.

`Preconditions and Postconditions`
```
other: SortedLinkedList
Returns: SortedLinkedList
```

Examples:

```
>>> p
Head:Node(1)
Tail:Node(7)
List:1 -> 2 -> 2 -> 2 -> 3 -> 4 -> 5 -> 6 -> 6 -> 7
>>> y
Head:Node(0)
Tail:Node(7)
List:0 -> 2 -> 2 -> 4 -> 6 -> 6 -> 7 -> 7
>>> p.intersection(y)
Head:Node(2)
Tail:Node(7)
List:2 -> 4 -> 6 -> 7
```

Overall Functionality **(2 pts)**

The grading script will perform a series of mixed linked list operations and compare the final status of your list. The last 2 points are based on the correctness of the Linked List after the mixed operations are completed (node links, head and tail references). There is no credit for design or clarity of code in this section. Verify that all methods work correctly when method calls are performed in a mixed manner.

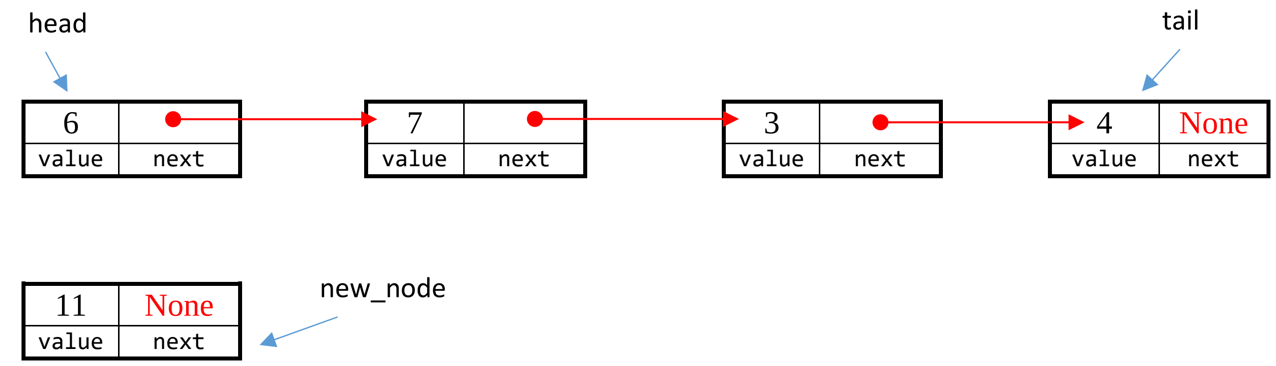
Example:
```
>>> x=SortedLinkedList()
>>> x.add(4.5)
>>> x.add(-3)
>>> x.add(0)
>>> x.add(5)
>>> x.add(-9)
>>> x.add(12.7)
>>> x.add(-3.5)
>>> x.add(2)
>>> x.add(4)
>>> x.add(1)
>>> x.add(3)
>>> x.add(2)
>>> x
Head:Node(-9)
Tail:Node(12.7)
List:-9 -> -3.5 -> -3 -> 0 -> 1 -> 2 -> 2 -> 3 -> 4 -> 4.5 -> 5 -> 12.7
```

```
>>> sublst1, sublst2 = x.split()
>>> sublst1
Head:Node(-9)
Tail:Node(2)
List:-9 -> -3.5 -> -3 -> 0 -> 1 -> 2
>>> sublst2
Head:Node(2)
Tail:Node(12.7)
List:2 -> 3 -> 4 -> 4.5 -> 5 -> 12.7
>>> x.add(2)
>>> x.add(2)
>>> x.add(2)
>>> x.add(3)
>>> x.add(3)
>>> x.add(-12)
>>> x
Head:Node(-12)
Tail:Node(12.7)
List:-12 -> -9 -> -3.5 -> -3 -> 0 -> 1 -> 2 -> 2 -> 2 -> 2 -> 2 -> 3 -> 3 -> 3 -> 4 -> 4.5 -> 5
-> 12.7
>>> x.add(-3)
>>> x
Head:Node(-12)
Tail:Node(12.7)
List:-12 -> -9 -> -3.5 -> -3 -> -3 -> 0 -> 1 -> 2 -> 2 -> 2 -> 2 -> 2 -> 3 -> 3 -> 3 -> 4 ->
4.5 -> 5 -> 12.7
>>> sublst1, sublst2 = x.split()
>>> sublst1
Head:Node(-12)
Tail:Node(2)
List:-12 -> -9 -> -3.5 -> -3 -> -3 -> 0 -> 1 -> 2 -> 2 -> 2
>>> sublst2
Head:Node(2)
Tail:Node(12.7)
List:2 -> 2 -> 3 -> 3 -> 3 -> 4 -> 4.5 -> 5 -> 12.7
>>> sublst1.intersection(sublst2)
Head:Node(2)
Tail:Node(2)
List:2
>>> sublst1.removeDuplicates()
>>> sublst2.removeDuplicates()
>>> sublst1
Head:Node(-12)
Tail:Node(2)
List:-12 -> -9 -> -3.5 -> -3 -> 0 -> 1 -> 2
>>> sublst2
Head:Node(2)
Tail:Node(12.7)
List:2 -> 3 -> 4 -> 4.5 -> 5 -> 12.7
>>> x
Head:Node(-12)
Tail:Node(12.7)
List:-12 -> -9 -> -3.5 -> -3 -> -3 -> 0 -> 1 -> 2 -> 2 -> 2 -> 2 -> 2 -> 3 -> 3 -> 3 -> 4 ->
4.5 -> 5 -> 12.7
>>> x.removeDuplicates()
>>> x
Head:Node(-12)
Tail:Node(12.7)
List:-12 -> -9 -> -3.5 -> -3 -> 0 -> 1 -> 2 -> 3 -> 4 -> 4.5 -> 5 -> 12.7
```

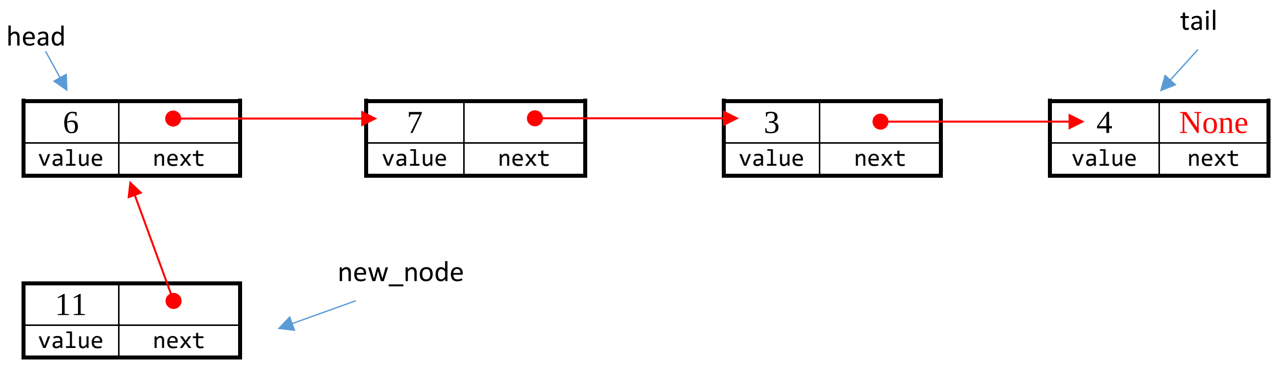
Solutions to Practice problem

a) Insert 11 before 6

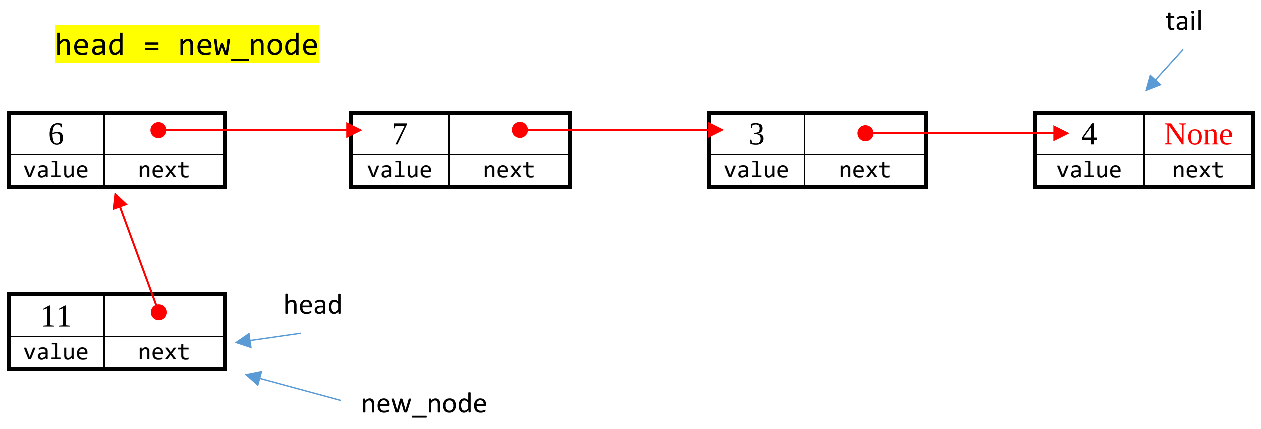
```
new_node = Node(11)
```



```
new_node.next = head
```

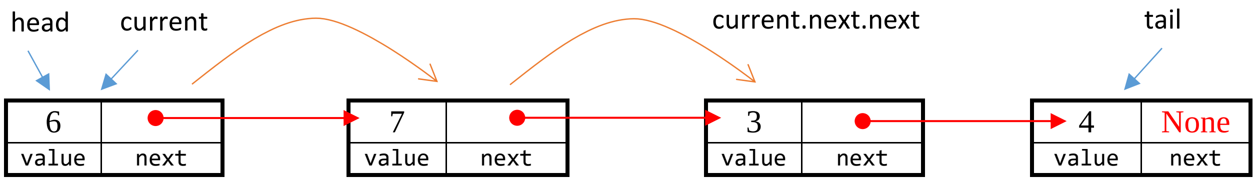


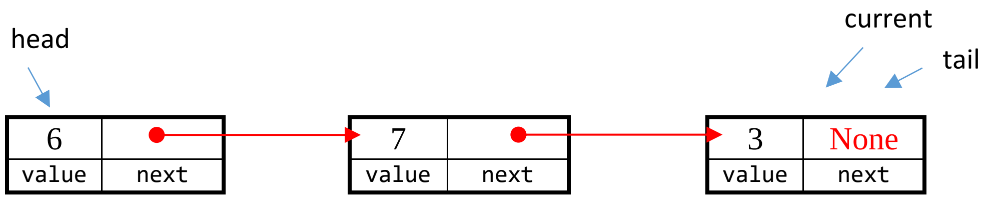
```
head = new_node
```



b) Remove 4

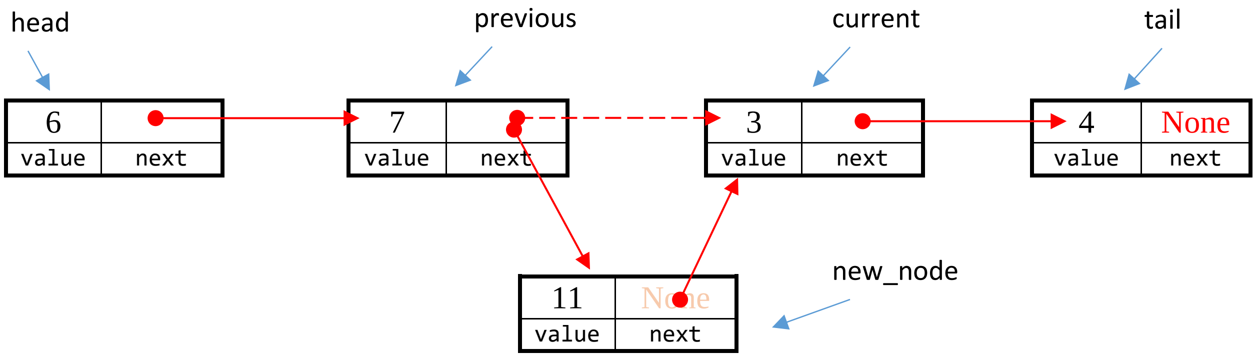
```
# Several approaches, this does not use a previous node
current = head
while current.next is not None and current.next.next is not None:
      current = current.next
current.next = None
tail=current
```





c) Insert 20 before 3

```
# Several approaches, this uses a previous node
current = head
previous = None
new_node = Node(20)
while current.value != 3:
      previous = current
      current = current.next
previous.next = new_node
new_node.next = current
```


head
previous
current
tail
6
value
next
7
value
next
3
value
next
4
None
value
next
11
None
value
next
new_node