Lab 4 (10 points) **Due date:** October 11, 2024, 11:59 PM

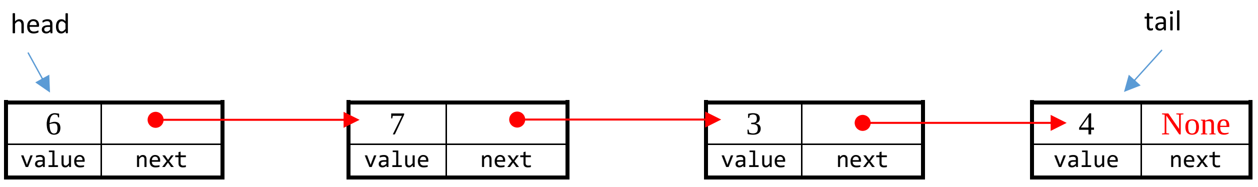
Before you start…

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 pointer 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 List 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

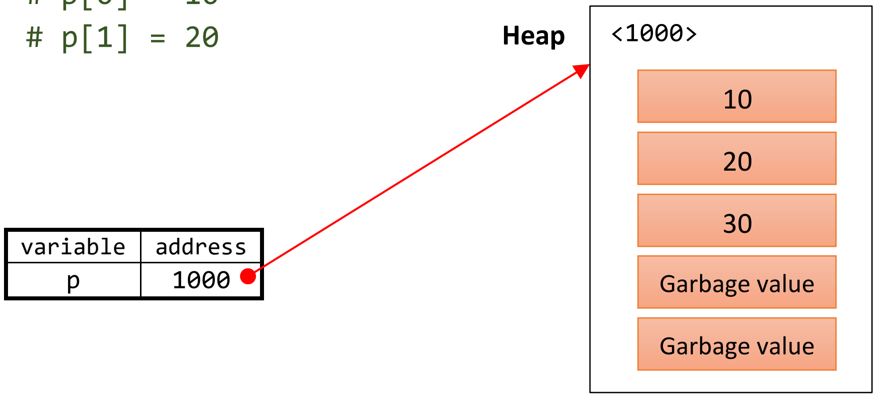
What is malloc and how are we emulating it in Python?

Disclaimer: The following information has been heavily abstracted, so don't think of the methods in this lab as to actually allocating or deallocating memory, just think of them as fancy names for linked list operations (add, remove, get, clear, etc.) that introduce you to some interesting topics of memory management covered in later courses.

Malloc, which stands for Memory Allocation, is how memory in a computer gets given to different portions of your program. In Python when you create a variable, the language itself will figure out how much memory to allocate and will dynamically change it for you. This is not the case in languages such as C or C++ (there are more, but these are the big ones). In C, if you want to create a list (an array) you need to specify how large, and you can only use as much data as what you originally allocate. For example, let's say you had a program that asks the user to input a number and your program outputs a list with a length equal to the input. You can't blindly guess the number, so you let the user input the number, then allocate a list of that length. This is a very basic example of when malloc would be used:

# Allocating memory for an array of 5 integers using malloc in C, size of int is 4 bytes

```
n = 5
int *p = (int*) malloc(n*sizeof(int))    #  5 * 4 = 20 bytes
*(p+0) = 10     # p[0] = 10
*(p+1) = 20     # p[1] = 20
*(p+2) = 30
```



In this lab we will imagine that calling malloc returns you back a pointer (reference) to the beginning of your data, and you can get the rest of the data using that pointer. To simulate this "pointer" we will use a linked list. The functions that you will be programming below are your standard linked list functions, just with some added functionality. The malloc function just creates an empty linked list of the length specified by the input. The free function unlinks all nodes in the linked list. The realloc function takes the list you have already created and either shortens or extends it.

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. You will not receive credit if you do not follow these directions.

The MallocLibrary class

In our Hands On – Linked List I and II lecture from Module 5, we worked on the implementation of a Singly Linked List. We will use the basics of that structure to implement a class that emulates the behavior the dynamic memory allocation in C.

Node class
Creates a Node object with a value attribute and a pointer to the next object. When value is not provided, it defaults to None. This class has been implemented for you in the starter code.

Examples:

```
>>> node_1 = Node()
>>> node_1
Node(None)
>>> node_2 = Node(7)
>>> node_2
Node(7)
>>> node_3 = Node('hi')
>>> node_3
Node(hi)
```

The Malloc_Library class has the following structure:
`__init__`: Initializes a head attribute to None. This method has been implemented for you in the starter code. You are not allowed to modify its implementation in any way.

`__repr__`: Returns a string representation of the linked list with each nodes value separated by an `->` character. This method has been implemented for you in the starter code.

`__len__`: Returns the number of nodes in the linked list as int. **(0.5 pts)**

`__setitem__`: Sets the node at the given *position* to the *value* specified. Index is defined in the range [0, size-1]. Just like with normal indexing, if indexing beyond the length of the linked list raise an index error. **(0.75 pt)**

Tip: The `raise` keyword is used to raise an exception. You can define what kind of error to raise, and the text to print to the user.

```
txt = "hello"
value = 12
if len(txt) <= value:
  raise IndexError("Index out of range")
```

```
Preconditions and Postconditions
pos: integer
value: any data type
Returns: None
```

Examples:

```
>>> my_lst = Malloc_Library()
>>> my_lst.malloc(5)
>>> my_lst
None -> None -> None -> None -> None
>>> my_lst[0] = 23    # invokes my_lst.__setitem__(0, 23)
>>> my_lst
23 -> None -> None -> None -> None
>>> my_lst[5]= 33
Traceback (most recent call last):
   ...
      raise IndexError
```

`__getitem__`: Returns the value of the node at the given position. Index is defined in the range [0, size-1]. Just as __setitem__, when indexing goes beyond the length of the linked list raise an index error **(0.75 pt)**

```
Preconditions and Postconditions
pos: integer
Returns: None -> if the list is empty
         Any value -> when index is in range
```

Examples:

```
>>> my_lst = Malloc_Library()
>>> my_lst.malloc(5)
>>> my_lst
None -> None -> None -> None -> None
>>> my_lst[0] = 23
>>> my_lst
23 -> None -> None -> None -> None
>>> my_lst[0]      # invokes my_lst.__getitem__(0)
23
```

`malloc`: Creates a linked list with length specified by *size* parameter. Initializes all values of each Node to None **(0.75 pts)**

```
Preconditions and Postconditions
size: positive integer
Returns: None
```

Examples:

```
>>> my_lst = Malloc_Library()
>>> my_lst.malloc(5)
>>> my_lst
None -> None -> None -> None -> None
>>> your_lst = Malloc_Library()
>>> your_lst.malloc(12)
>>> your_lst
```

```
None -> None -> None -> None -> None -> None -> None -> None -> None -> None -> None
-> None
>>> your_lst.malloc(3)
>>> your_lst
None -> None -> None
```

`calloc`: Creates a linked list with length specified by *size* parameter. Initializes all values of each Node to 0 **(0.75 pts)**

`Preconditions and Postconditions`
`size: positive integer`
`Returns: None`

Examples:

```
>>> my_lst = Malloc_Library()
>>> my_lst.calloc(4)
>>> my_lst
0 -> 0 -> 0 -> 0
>>> your_lst = Malloc_Library()
>>> your_lst.malloc(3)
>>> your_lst
None -> None -> None
>>> your_lst.calloc(4)
>>> your_lst
0 -> 0 -> 0 -> 0
>>> your_lst.calloc(9)
>>> your_lst
0 -> 0 -> 0 -> 0 -> 0 -> 0 -> 0 -> 0 -> 0
```

`free`: Clears the linked list by unlinking all nodes in the list. Note that simply setting head=None is not enough for this method, and you will not receive credit is that is the only line in your implementation. You must delete the links between all nodes in the list. **(1 pt)**

`Postconditions`
`Returns: None`

Examples:

```
>>> my_lst = Malloc_Library()
>>> my_lst.malloc(4)
>>> my_lst[3]=23
>>> my_lst[0]=23
>>> my_lst[1]=4
>>> my_lst[2]=89
>>> my_lst
23 -> 4 -> 89 -> 23
>>> temp1 = my_lst.head.next
>>> temp2 = temp1.next
>>> my_lst.free()
>>> my_lst.head is None
```

```
True
>>> temp1.next is None
True
>>> temp2.next is None
True
```

`realloc`: Reallocates the memory (resizes list) to be of the *size* specified in the parameter. When *size* is longer than length of list then keeps the list and extends to it with nodes of value None. If *size* is smaller than length of list, then remove nodes from the end of the list until list is of specified *size*. If size is 0, “memory is deallocated” (removes all nodes from the list). realloc() will act a as malloc() when the list is empty. **(2 pts)**

`Preconditions and Postconditions`
`size: integer >= 0`
`Returns: None`

Examples:

```
>>> my_lst = Malloc_Library()
>>> my_lst.realloc(4)
>>> my_lst[3] = 5
>>> my_lst[1] = 69
>>> my_lst
None -> 69 -> None -> 5
>>> my_lst.realloc(8)
>>> my_lst
None -> 69 -> None -> 5 -> None -> None -> None -> None
>>> my_lst.realloc(11)
>>> my_lst
None -> 69 -> None -> 5 -> None -> None -> None -> None -> None -> None -> None
>>> my_lst.realloc(5)
>>> my_lst
None -> 69 -> None -> 5 -> None
>>> my_lst.realloc(3)
>>> my_lst
None -> 69 -> None
>>> my_lst.realloc(1)
>>> my_lst
None
>>> my_lst.realloc(7)
>>> my_lst
None -> None -> None -> None -> None -> None -> None
>>> temp=my_lst.head.next
>>> my_lst.realloc(0)
>>> my_lst.head is None
True
>>> temp.next is None
True
>>> my_lst.realloc(6)
>>> my_lst
None -> None -> None -> None -> None -> None
```

`memcpy`: This method is intended to copy values from one list (original list) to another (pointer_2) starting at specified positions and up to a given number of nodes (size). The function takes in few arguments: *self* is original list or pointer, *ptr1_start_idx* is the starting position in the original list from which copying begins, *pointer_2* is the target list where the values will be copied, *ptr2_start_idx* is the starting position in the target list where copied values will be placed. Positions follow the syntax of Python sequences [0, size-1]. No changes are made to the lists when any of the starting points are not within the corresponding list size. *size* parameter is the number of values to be copied from source (*self*) to destination (*pointer_2*). If *size* is larger than original list's length, then size becomes the size of the original list.
NOTE :- Memory allocation is done only when malloc(), calloc() or realloc() methods are called on the pointer (list). So if there is no memory then, copying of values is not done. This is depicted in the first examples in the doctests given below. **(2 pts)**

```
Preconditions and Postconditions
ptr1_start_idx: integer >= 0, corresponds to starting position in self
pointer_2: Malloc_Library object
ptr2_start_idx: integer >= 0, corresponds to starting position in pointer_2
size: integer
Returns: None
```

Examples:

```
>>> lst1=Malloc_Library()
>>> lst2=Malloc_Library()
>>> lst1.memcpy(5, lst2, 3, 11)
>>> lst1.head is None
True
>>> lst2.head is None
True
>>> lst1.malloc(5)
>>> lst1[3]= 65
>>> lst1[0]= 72
>>> lst1.memcpy(0, lst2, 0, 2)
>>> lst1
72 -> None -> None -> 65 -> None
>>> lst2.head is None
True
>>> lst1.malloc(5)
>>> lst1[3]= 65
>>> lst1[1]= 5
>>> lst1[0]= 1
>>> lst1[2]= 12
>>> lst1[4]= 33
>>> lst1
1 -> 5 -> 12 -> 65 -> 33
>>> lst2.malloc(4)
>>> lst1.memcpy(2, lst2, 1, 15)
>>> lst2
None -> 12 -> 65 -> 33
>>> lst2.malloc(10)
>>> lst2
None -> None -> None -> None -> None -> None -> None -> None -> None -> None
>>> lst1.memcpy(0, lst2, 0, 2)
>>> lst2
1 -> 5 -> None -> None -> None -> None -> None -> None -> None -> None
>>> lst1.memcpy(0, lst2, 5, 4)
>>> lst2
1 -> 5 -> None -> None -> None -> 1 -> 5 -> 12 -> 65 -> None
>>> lst2.malloc(3)
>>> lst2
None -> None -> None
>>> lst1.memcpy(3, lst2, 5, 4)
>>> lst2
None -> None -> None
>>> lst1.memcpy(3, lst2, 2, 4)
>>> lst2
None -> None -> 65
```

Overall Functionality **(1.5 pts)**

The grading script will perform a series of mixed operations and compare the final status of your list(s). The last 1.5 points are based on the correctness of the Linked List(s) after the mixed operations are completed (node links, head 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:

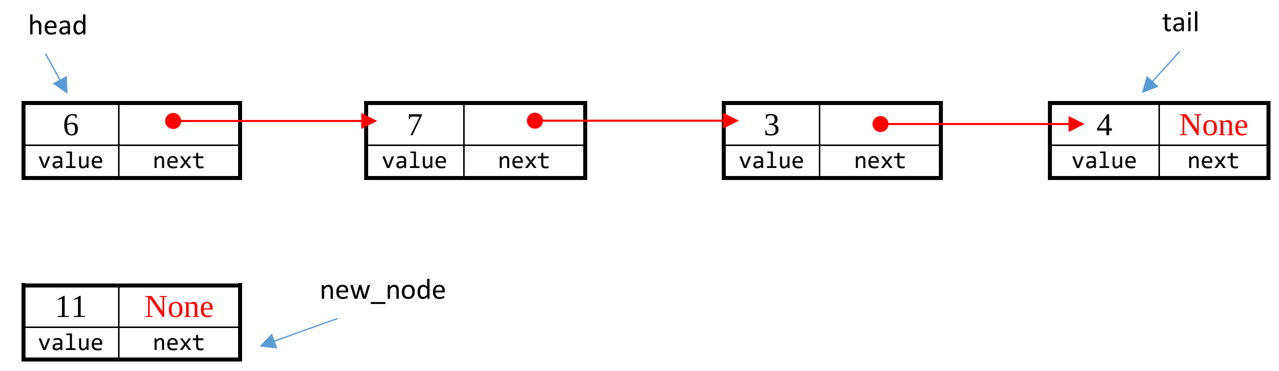
```
>>> lst1=Malloc_Library()
>>> lst2=Malloc_Library()
>>> lst3=Malloc_Library()
>>> lst1.calloc(3)
>>> lst2.malloc(6)
>>> lst3.realloc(5)
>>> lst1
0 -> 0 -> 0
>>> lst2
None -> None -> None -> None -> None -> None
>>> lst3
None -> None -> None -> None -> None
>>> lst1[1] = 65
>>> lst1[2] = 123
>>> lst2[6] = 41
Traceback (most recent call last):
    raise IndexError
>>> lst2[3] = lst1[2]
>>> lst2[0] = lst1[2]
>>> lst2[5] = lst1[2]*9
>>> lst2[1] = 23
>>> lst1
0 -> 65 -> 123
>>> lst2
123 -> 23 -> None -> 123 -> None -> 1107
>>> lst2.memcpy(3, lst3, 0, 4)
>>> lst3
123 -> None -> 1107 -> None -> None
>>> lst3.memcpy(2, lst1, 2, 1)
>>> lst1
0 -> 65 -> 1107
```

```
>>> lst2.realloc(2)
>>> lst1
0 -> 65 -> 1107
>>> lst2
123 -> 23
>>> lst3
123 -> None -> 1107 -> None -> None
>>> lst2.memcpy(2, lst3, 3, 3)
>>> lst3
123 -> None -> 1107 -> None -> None
>>> lst2.memcpy(0, lst3, 3, 3)
>>> lst3
123 -> None -> 1107 -> 123 -> 23
>>> lst2
123 -> 23
>>> lst1.realloc(2)
>>> lst1
0 -> 65
>>> lst1.realloc(1)
>>> lst1
0
>>> lst2.memcpy(1, lst1, 0, 2)
>>> lst1
23
>>> lst2
123 -> 23
>>> lst3
123 -> None -> 1107 -> 123 -> 23
```

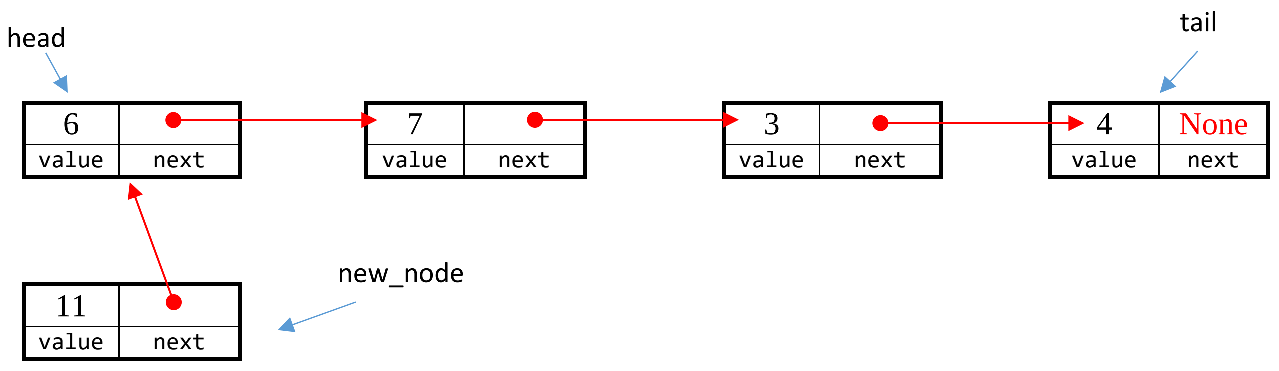
Solutions to Practice problem

a) Insert 11 before 6

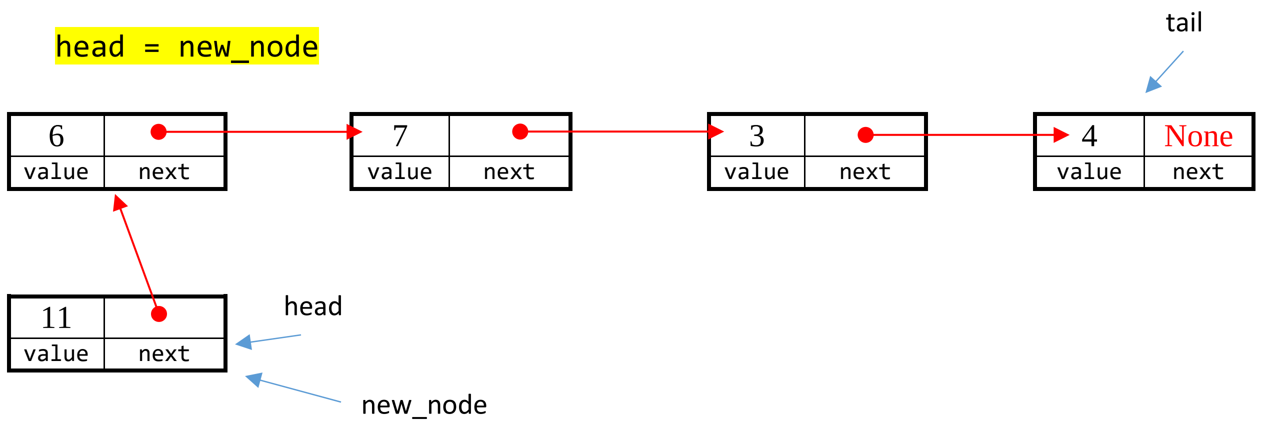
`new_node = Node(11)`



`new_node.next = head`

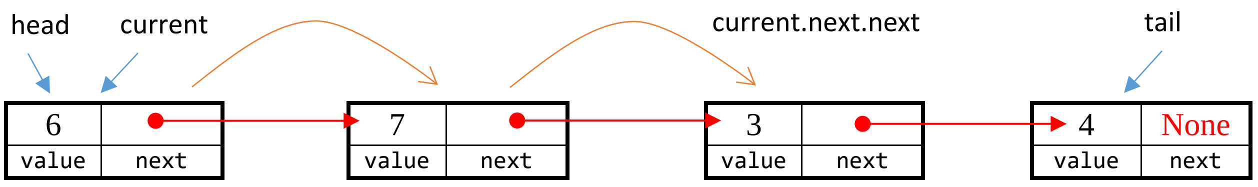


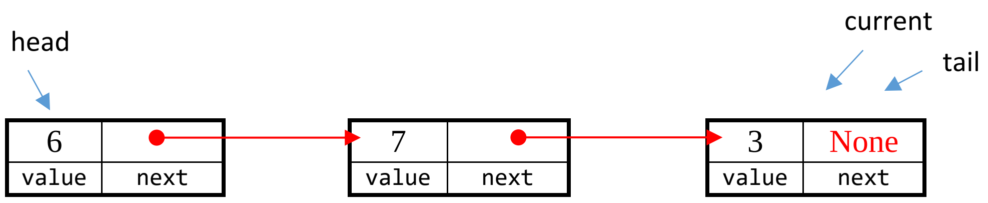
`head = new_node`



b) Remove 4

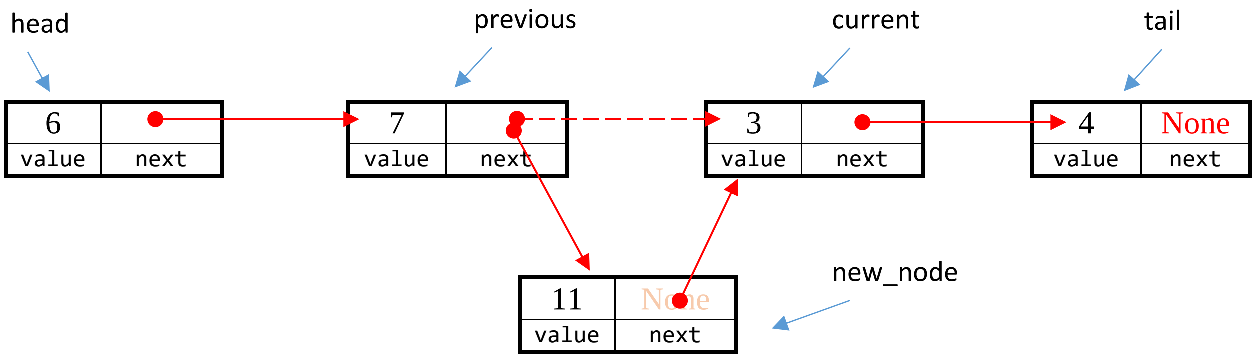
```
# Several approaches, this does not use a previous node
current = head
while current.next is not None and current.next.next is not None:
      current = current.next
current.next = None
tail=current
```





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