- This lab will cover Linked Lists.
- You may want to refer to the text and your lecture notes during lab as you solve the problems.
- When approaching the problems, think before you code. It is good practice and generally helpful to lay out possible solutions for yourself.
- You should write test code to try out your solutions.
- You must stay for the duration of the lab. If you finish early, you can help other students to complete the lab. If you don't finish by the end of the lab, it is recommended that you complete the lab on your own time.
- Your TAs are available to answer your questions in lab, during office hours, and on Piazza.

Vitamins (maximum 30 minutes)

---

1. During lecture you learned about the different methods of a doubly linked list. It is important to understand the runtime of each method. Provide the following worst-case runtime for those methods.

    a. `def __len__(self):`

    b. `def is_empty(self):`

    c. `def first_node(self):`

    d. `def last_node(self):`

    e. `def add_after(self, node, data):`

    f. `def add_first(self, data):`

    g. `def add_last(self, data):`

    h. `def add_before(self, node, data):`

    i. `def delete_node(self, node):`

    j. `def delete_first(self):`

    k. `def delete_last(self):`

2. Trace the following function. What is the output of the following code? Describe what the function does.

```
def func(string_input):
      L = DoublyLinkedList()
      for chr in string_input:
            L.add_last(chr)

      cursor = L.first_node()
      while cursor.data is not None:
            if cursor.data.lower() in ['a','e','i','o','u']:
                  temp = cursor.next
                  L.delete_node(cursor)
                  cursor = temp
            else:
                  cursor = cursor.next

      new_str = "".join(L)
      return new_str

string_input = "TheCatGoesMeowAndTheCowGoesMoo"
func(string_input)
```

3. What is the output of the following code?

```
L = DoublyLinkedList()
L.add_first(5)
L.add_last(8)
L.add_first(2)
L.add_last(12)
L.add_first(3)
L.add_last(6)

for elem in L:
     print(elem)

print("----------------")
cursor = L.first_node()
lst = [2,4,8,10,15]

i = 0;
while cursor.data != 12:
     L.add_last(lst[i])
     i = i + 1
     cursor = cursor.next

for elem in lst:
     L.add_after(cursor, elem)

for elem in L:
     print(elem)
```

Coding

In this section, it is strongly recommended that you solve the problem on paper before writing code.

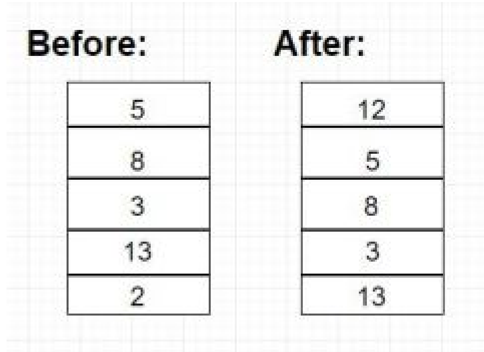
1. Implement a `LinkedStack` class. A linked stack is a stack implemented using a linked list as a data member (not a dynamic array as we used for the `ArrayStack` class). The standard operations for a `LinkedStack` will run in **O (1) worst-case** runtime. Your `LinkedStack` object should use a doubly linked list as a data member. Your implementation should include the following methods:

```
def push(self, e):
''' Add element e to the top of the stack '''

def pop(self):
''' Remove and return the top element from the stack. If the
stack is empty, raise an exception'''

def top(self):
''' Return a reference to the top element of the stack without
removing it. If the stack is empty, raise an exception '''

def is_empty(self):
''' Return True if stack is empty'''

def __len__(self):
'''Return the number of elements in the stack'''
```

2. Implement a `LeakyStack` class. A `LeakyStack` is similar to a stack, however, it differs from a stack in that the `LeakyStack` is bounded. During initialization, the `LeakyStack` is given a maximum size, n. If the `LeakyStack` has n elements, it is considered full. The next element that is added will be placed on top, but the bottom element will be removed.

```
For example, given n = 5 and the following code:
    S = LeakyStack(5)
    S.push(2)
    S.push(13)
    S.push(3)
    S.push(8)
    S.push(5)
```

The stack is now full as shown in the *before* stack in the diagram below. After calling `S.push(12)`, the bottom integer, 2, is removed to make space for the integer 12. This is shown in the *after* stack in the diagram below.



The standard operations for a `LeakyStack` occur in O(1) worst case runtime. Your `LeakyStack` object should use a doubly linked list as a data member. You should include the following methods:

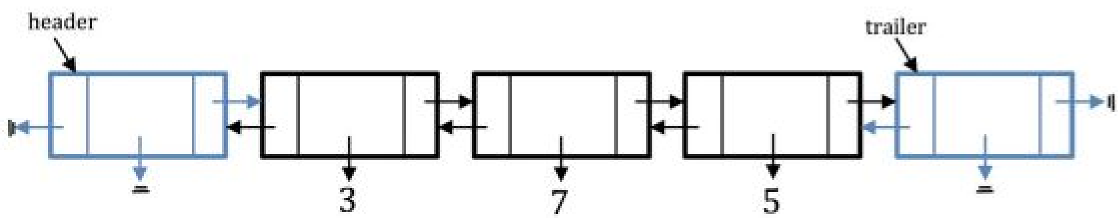
```
def __init__(self, max_num_of_elems):
''''''An empty leaky stack implemented using a doubly linked
list''''''

def push(self, e):
'''''' Add element e to the top of the stack ''''''

def pop(self):
'''''' Remove and return the top element from the stack. If the
stack is empty, raise an exception''''''

def top(self):
'''''' Return a reference to the top element of the stack without
removing it. If the stack is empty, raise an exception ''''''

def is_empty(self):
'''''' Return True if stack is empty''''''

def __len__(self):
''''''Return the number of elements in the stack''''''
```

3. Implement a method to recursively sum the values in a linked list. You can assume that the elements of the list are numbers. You are allowed to define a helper function with a runtime of Θ(n).

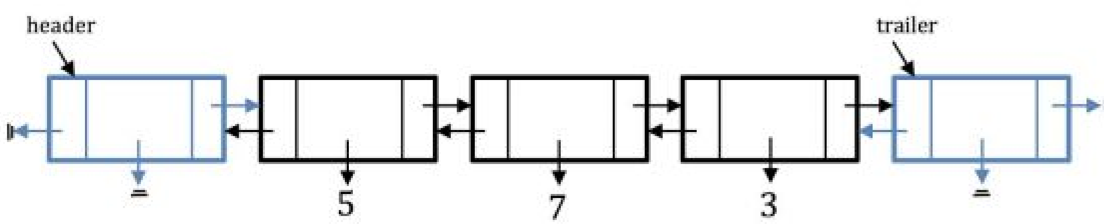
```
def sum_lnk_lst(lnk_lst):
''''''Return the sum of the values in the linked list''''''
```

4. Implement a method to reverse a doubly linked list. This method should be non-recursive and done **in place** (do not return a new list).

For example if your list looks like:



After calling the method on it, it will look like:



You will implement the reversal in two ways:

a. First implement a function which reverses the data in the list, but does not move any nodes.

```
def reverse_list_change_elements_order(lnk_lst):
''' Reverses the linked list '''
```

b. Next, implement a function which reverses the order of the nodes in the list.

```
def reverse_list_change_nodes_order(lnk_lst):
''' Reverses the linked list '''
```