- This lab will cover Linked Lists, Stacks, and Queues.
- It is assumed that you have reviewed chapter 6 & 7 of the textbook. 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. Doing so is good practice and can help you lay out possible solutions.
- Think of any possible test cases that can potentially cause your solution to fail!
- You must stay for the duration of the lab. If you finish early, you may help other students. If you don't finish by the end of the lab, we recommend you complete it on your own time.
- Your TAs are available to answer questions in lab, during office hours, and on Piazza.

Vitamins (maximum 30 minutes)

---

1. Fill in the chart for the following prefix, infix, postfix expressions.

Prefix	**Infix**	**Postfix**	**Value**
- * 3 4 10	3 * 4 - 10		2
	(5 * 5) + (10 / 2)	5 5 / * 10 2 / +	30
		10 2 - 4 / 8 +	
+ * 6 3 * 8 4			
	(8 * 2) + 4 - (3 + 6)		

2. During lecture you learned about the different methods of a doubly linked list.

   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):
```

3. What is the output of the following code?

```
from DoublyLinkedList import DoublyLinkedList

dll = DoublyLinkedList()

dll.add_first(10)
dll.add_last(20)
dll.add_first(30)
dll.add_last(40)
dll.add_last(50)
dll.add_first(60)

print(dll)
```

4. Trace the following function. What is the output of the following code using the doubly linked list from question 3? Give mystery an appropriate name.

```
#dll = Doubly Linked List
def mystery(dll):

    if len(dll) >= 2:
        node = dll.trailer.prev.prev
        node.prev.next = node.next
        node.next.prev = node.prev

        node.next = None
        node.prev = None
        return node

    else:
        raise Exception("dll must have length of 2 of
        greater")

print(mystery(dll.data))
```

Coding

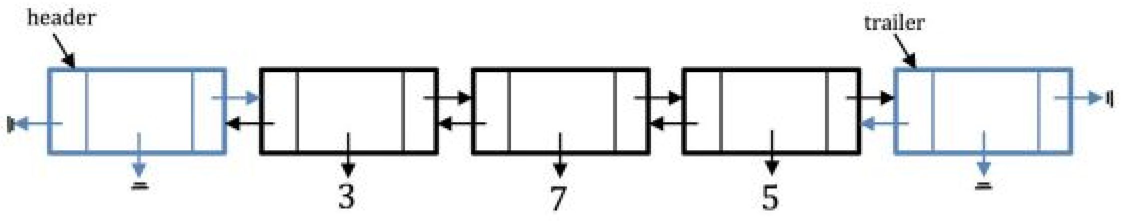
In this section, it is strongly recommended that you solve the problem on paper before writing code.

1. Implement the following method to the Doubly Linked List class. The get item operator takes in an index, i, and returns the value in the ith node of the Doubly Linked List.

   You only need to support positive indices. Your solution should try to optimize the get item method. This means that you should decide whether to iterating from either the header or trailer based on how close i is.

```
def __getitem__(self, i):
    '''Return the value at the ith node. If i is out of range,
    an IndexError is raised'''
```

   For example, if your Doubly Linked List looks like this:



```
print(dll[0]) # 3 (should iterate from the header)

print(dll[1]) # 7 (either way works)

print(dll[2]) # 5 (should iterate from the trailer)

print(dll[3]) # IndexError
```

   What is the worst-case run-time of the get item operator? Can we do better?

2. Implement a `LinkedStack` class.

   A linked stack is a stack, implemented using a doubly 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**.

```
class LinkedStack:

    def __init__(self):
        self.data = DoublyLinkedList()


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


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


    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 '''
```

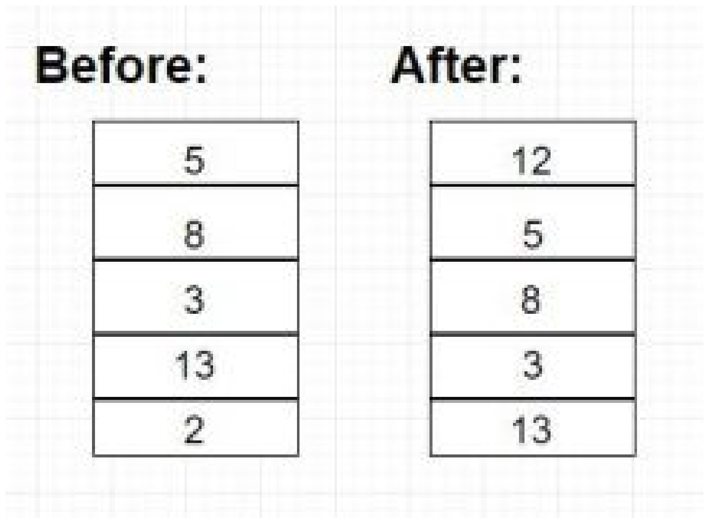
2. A `Leaky Stack` is similar to a stack except it can only contain a certain number of elements.

   During initialization, it is given a maximum size, n. If the leaky stack has n elements, it is considered full. The next element that is added will be placed on ‘top’, and the ‘bottom’ element will be removed. (The bottom element leaks out)

   For example, consider the following code:

```
#Before
leakyS = LeakyStack(5)  #May only contain 5 elements
leakyS.push(2)
leakyS.push(13)
leakyS.push(3)
leakyS.push(8)
leakyS.push(5)         #leakyS is now full (has 5 elements)

#After
leakyS.push(12)        #the 2 leaks out when 12 is pushed
```



The `LeakyStack` should have the following behavior:

```
def __init__(self, n):
'''Creates an empty stack. n is the maximum number of
elements'''

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

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

def push(self, e):
''' Add element e to the top of the stack. If the stack is
full, the bottom-most element is removed from 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 '''
```

In this question, you will suggest two different data structures (write two classes) that implement the *`Leaky Stack`* *ADT*.

In both implementations <u>all</u> operations have to run in O(1) worst case.

a. Define LinkedLeakyStack class, that uses a DoublyLinkedList to store the leaky stack's elements.

b. Define ArrayLeakyStack class, that uses an array to store the leaky stack's elements.

<u>Hint</u>: Think of a circular-array implementation for part b.

EXTRA (+4 pts)

---

3. In this question we will explore an alternative way to implements a *Stack* using just a *Queue* as the main underlying data collection.

   a. Write a QueueStack class, that implements a *Stack ADT* using an ArrayQueue as the only data member.

      You may only access the ArrayQueue's methods which include: len, is_empty, enqueue, dequeue, and first.

Implement two sets of the push & pop/top methods:

   b. Consider an implementation that optimizes **push so that it has a run-time of O(1)** amortized.

   c. Consider an implementation that optimizes **pop and top so that they have a run-time of O(1**) amortized.

   d. Analyze the worst case runtime of your two sets of implementations for push and pop/top methods.

Your implementation should be like so:

```
class QueueStack:

     def __init__(self):
          self.data = ArrayQueue()


     def __len__(self):
          return len(self.data)


     def is_empty(self):
          return len(self) == 0


     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 '''
```

4. Implement the `MeanQueue` ADT. The MeanQueue is mean because it only enqueues integers and floats and rejects any other data type (bool, str, etc)! However, a nice thing about this queue is that it can provide the sum and average (mean) of all the numbers stored in it in **O(1) run-time**. You may define additional member variables for this ADT.

   The MeanQueue will use a DoublyLinkedList as its underlying data member.

   Implement a MeanQueue with the following behavior, all with **O(1) run-time**:

```
class MeanQueue:

    def __init__(self):
        self.data = DoublyLinkedList()

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

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


    def enqueue(self, e):
    ''' Add element e to the front of the queue. If e is not
    an int or float, raise a TypeError '''


    def dequeue(self):
    ''' Remove and return the first element from the queue. If
    the queue is empty, raise an exception'''


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


    def sum(self):
    ''' Returns the sum of all values in the queue'''


    def average(self):
    ''' Return the average value in the queue'''
```