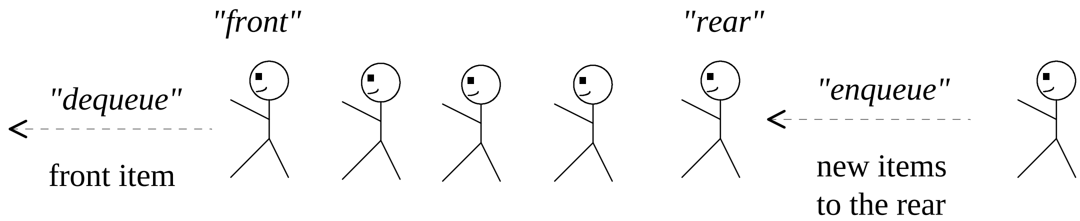
Data Structures (810:052) Lab 5 Name:__________________

Objective: To understand FIFO (First-In-First-Out) queue implementations in Python including being able to determine the big-oh of each operation.

Background: Read chapter 15 from the Lambert text. A FIFO queue is basically what we think of as a waiting line.



The operations/methods on a queue object, say myQueue are:

Method Call on myQueue object	**Description**
`myQueue.dequeue()`	Removes and returns the front item in the queue.
`myQueue.enqueue(myItem)`	Adds `myItem` at the rear of the queue
`myQueue.peek()`	Returns the front item in the queue without removing it.
`myQueue.isEmpty()`	Returns `True` if the queue is empty, or `False` otherwise.
`len(myQueue)`	Returns the number of items currently in the queue
`str(myQueue)`	Returns the string representation of the queue

To start the lab: Download and unzip the file at: www.cs.uni.edu/~fienup/cs052sum09/labs/lab5.zip

<u>Part A</u>: a) Complete the following table by indicating which of the queue operations should have preconditions and postconditions. Write “none” if a precondition or postcondition is not needed.

Method Call on myQueue object	**Precondition(s)**	**Postcondition(s)**
`myQueue.dequeue()`		
`myQueue.enqueue(myItem)`		
`myQueue.peek()`		
`myQueue.isEmpty()`		
`len(myQueue)`		
`str(myQueue)`		

One possible implementation of a queue would be to use an Array _items to store the queue items such that

- the front item is always stored at index 0,
- an integer _size data-attribute is used to maintain the number of items in the queue
- an integer _rear data-attribute is used to maintain the index of the rear item

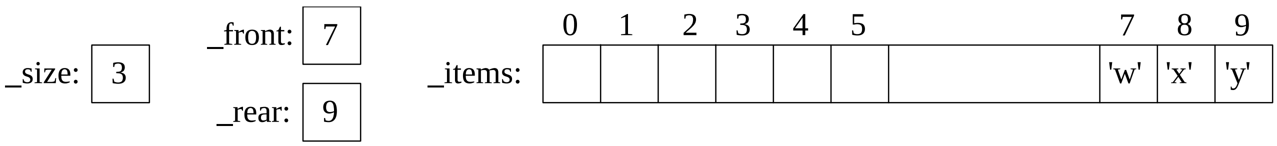
_size: 3 _rear: 2

	0	1	2	3	4	5		
_items:	'a'	'b'	'c'					

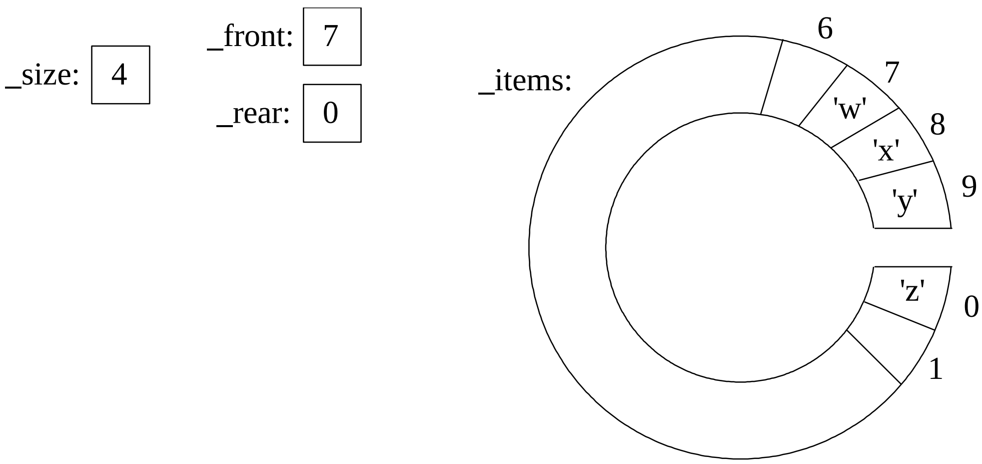
b) The file `queue.py` contains an `ArrayQueue` class that implements this representation. Complete the expected big-oh notation for each queue method for this implementation. Assuming "n" items in the queue.

`dequeue()`	`enqueue(item)`	`peek()`	`isEmpty()`	`__len__()`	`__str__()`

c) As pointed out in section 15.4.2 we can avoid “shifting the items left” on a dequeue operation by maintaining the index of the front item in addition to the rear. Overtime, the used portion of the array (where the actual queue items are) will drift to the right end of the array with the left end being unused, i.e.:



Now if we enqueue another item, we’d like the rear of the queue to “wrap” around to index 0, i.e., we’d like the array to behave “circularly.” After we enqueue('z'), we would have:



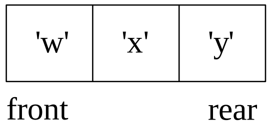
d) The file `queue2.py` contains the start of the `CircularArrayQueue` class. Complete the `CircularArrayQueue` class, and thoroughly test it using the menu-driven testQueue function included in the file.

e) Complete the expected big-oh notation for each queue method for this circular-array queue implementation. Assuming "n" items in the queue.

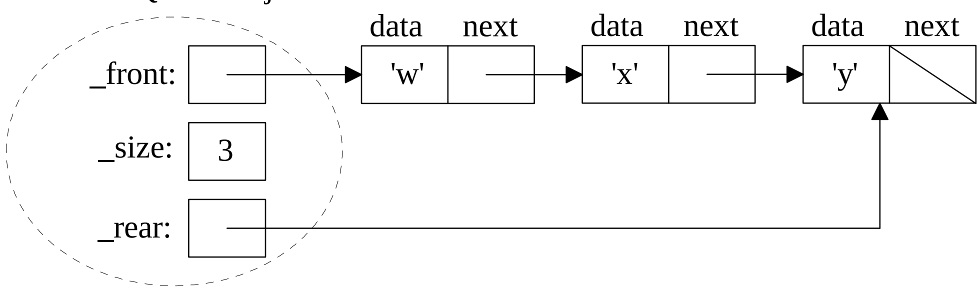
`dequeue()`	`enqueue(item)`	`peek()`	`isEmpty()`	`__len__()`	`__str__()`

Part B: The `Node` class defined in node.py is used to dynamically create storage for a new item added to a singly-linked list implementation of the `LinkedQueue` class in file `queue.py`. Conceptually, a `LinkedQueue` object would look like:

"Abstract Queue"



LinkedQueue Object



a) Unfortunately, the `Node` class does NOT practice good object-oriented design by allowing the LinkedQueue class access to its data and next attributes without going through accessor and mutator methods. I'd like you to reuse the improved `Node` class from Lab 4 and rewrite the `LinkedQueue` class to fixed this design problem.

b) Complete the expected big-oh notation for each queue method for this linked-list queue implementation. Assuming "n" items in the queue.

`dequeue()`	`enqueue(item)`	`peek()`	`isEmpty()`	`__len__()`	`__str__()`