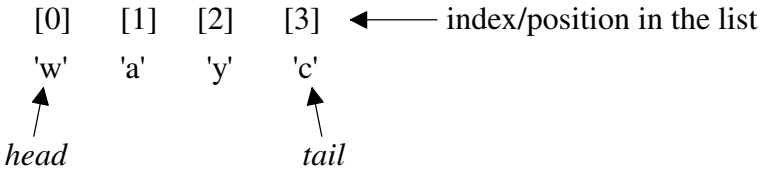
Data Structures (CS 1520) Lecture 13 Name:_____

Background: Read sections 13.5, 13.6 and chapter 16 from the Lambert text. A "list" is a generic term for a sequence of items in a linear arrangement. Unlike stacks and queues, access to list items is not limited to either end, but can be from an position in the list. The general terminology of a list is illustrated by:

"Abstract view of a list"



There are three broad categories of list operations defined in the textbook:

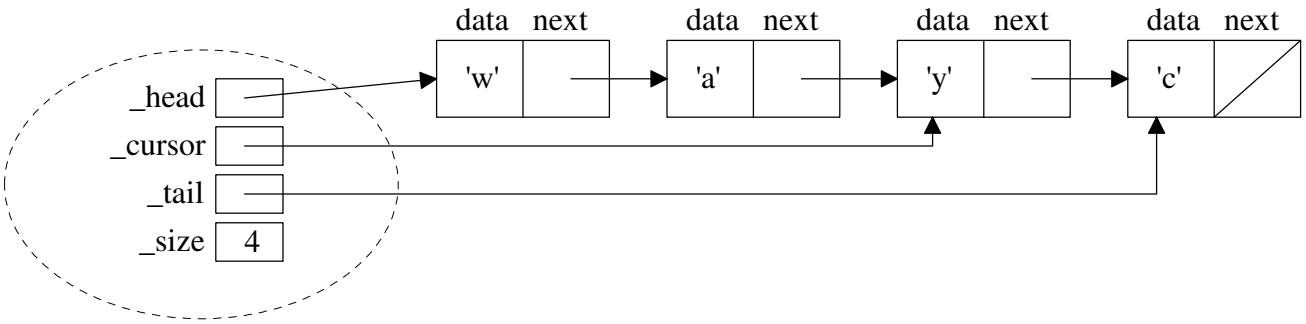
- **index-based operations** - the list is manipulated by specifying an index location, e.g.,
  myList.insert(3, item) # insert item at index 3 in myList
- **content-based operations** - the list is manipulated by specifying some content (i.e., item) in the list, e.g.,
  myList.index(item) # search for the item in the list and return its index if found; otherwise return -1
- **positional-base operations** - a *cursor* (current position in the list) can be moved around the list, and it is used to identify list items to be manipulated, e.g., **(WARNING: textbook's examples are wrong!!! pp. 649-651)**
  myList.first() # sets the cursor to the head of the list
  myList.remove() # deletes the first item in the list because that's where the cursor is located

The following table summarizes the operations from the three basic categories on a list, L:

Index-based operations	**Content-based operations**	**Positional-based operations**

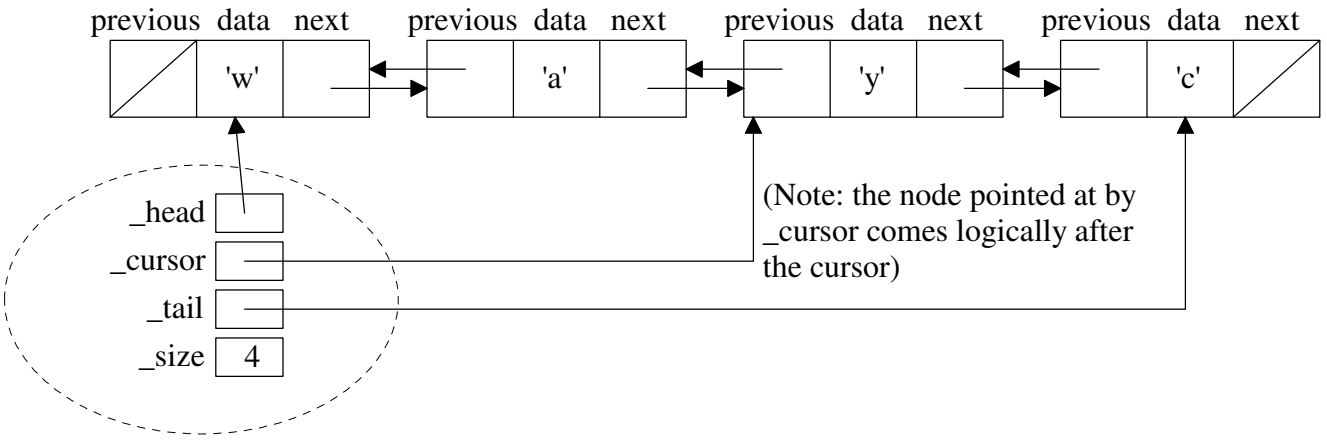
`L.insert(index, item)`
`item = L[index]`
`L[index] = newValue`
`L.remove(index)`
	`L.append(item)`
	`L.index(item)`
		`L.hasNext()`
		`L.next()`
		`L.hasPrevious()`
		`L.previous()`
		`L.first()`
		`L.last()`
		`L.insert(item)`
		`L.replace(item)`
		`L.remove()`

1. Why would a singly-linked list be a bad choice for implementing a positional-based list ADT?



2. Suppose we had a doubly-linked list implementation for a positional-based list ADT:

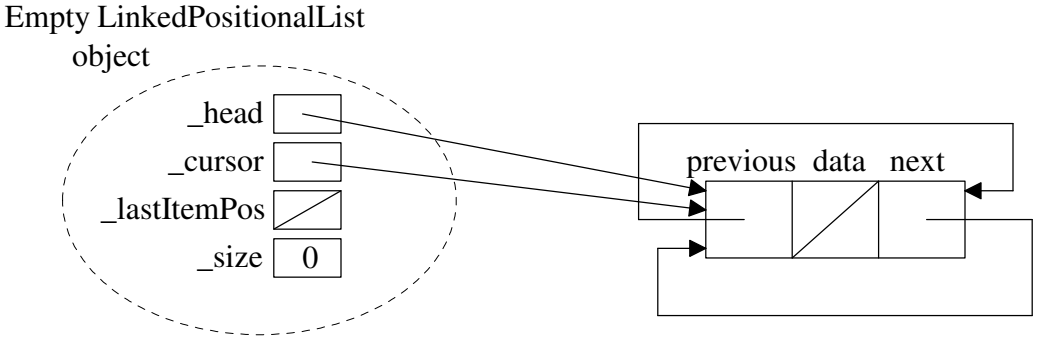
```
class TwoWayNode(Node):
    def __init__(self, data, previous = None, next = None):
        Node.__init__(self, data, next)
        self.previous = previous
```



a) `L.insert(item)` If the cursor is defined, insert `item` after it; otherwise, insert `item` at the end of the list. Write the "normal" case (cursor defined and inserting between two existing nodes) code.

b) In addition to the "normal" case (inserting between two existing nodes), what special cases would the `insert` operation need to handle for the doubly-linked list implementation above?

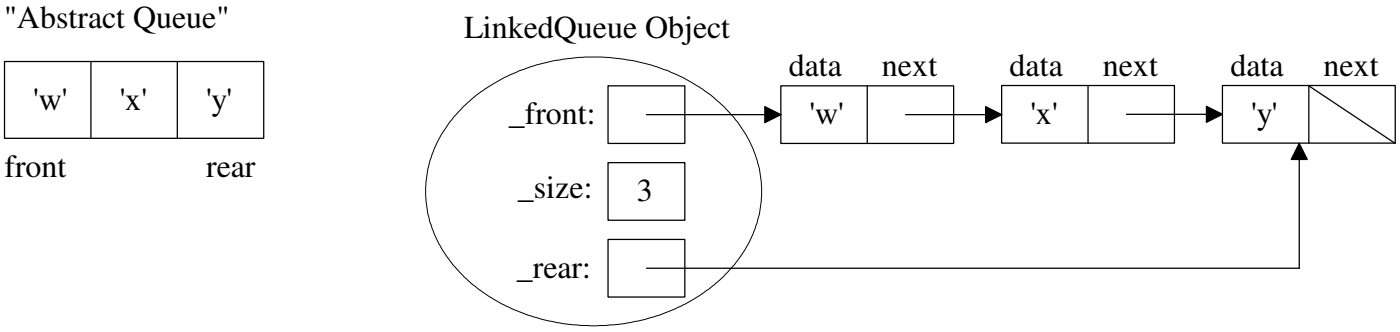
c) The textbook's `LinkedPositionalList` class, uses a circular, doubly-linked list with a *sentinel* (or *header*) node to reduce the number of "special cases". An empty list looks like:



d) Run your "normal" case code from (a) on an empty circular, doubly-linked list. Are there any special cases?

3. The Python `for` loop allows traversal of built-in data structures (strings, lists, tuple, etc) by an *iterator*. To accomplish this with ***our*** data structures we need to include an `__iter__(self)` method that gets used by the built-in `iter` function to create a special type of object called a *generator object*. The generator object executes as a separate process running concurrently with the process that created and uses the data structure being iterated. Iterators behave like stripped-down positional lists.

A singly-linked list implementation of the queue (`LinkedQueue` class in the text) conceptually would look like:

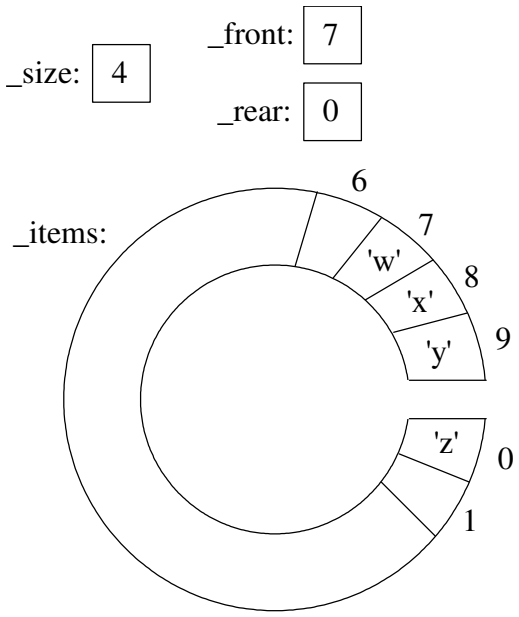


The iterator (`__iter__`) method for the `LinkedQueue` class would be:

```
class LinkedQueue(object):
    """ Link-based queue implementation."""


   def __iter__(self):
        """An iterator for a linked queue"""
        cursor = self._front
        while True:
            if cursor == None:
                raise StopIteration
            yield cursor.data
            cursor = cursor.next
```

The circular-array queue (`CircularArrayQueue`) implementation would look like:



a) Write an `__iter__` method for the `CircularArrayQueue`