Data Structures (810:052) Lab 7 Name:_____

Objective: Gain experience implementing a linked data structure by using a doubly-linked list to implement a positional list ADT.

Background: 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.,
  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

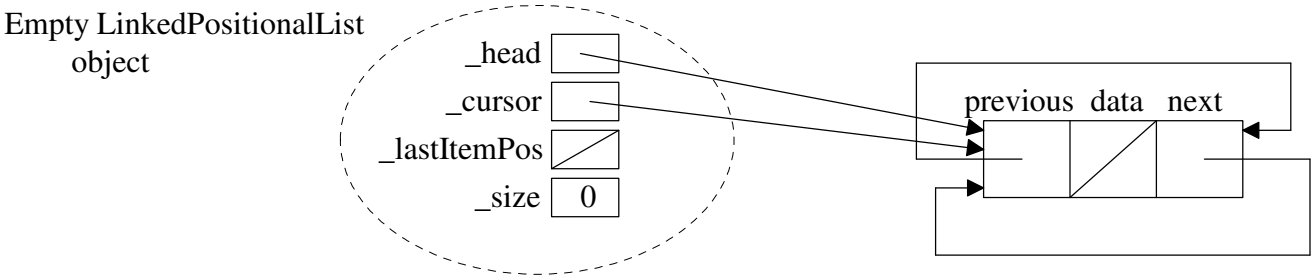
The following table summarizes the operations from the three basic categories on a list, L:

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()`

To start the lab: Download and unzip the file lab7.zip

Part A: The `positionalList.py` file contains a `LinkedPositionalList` class, which uses a circular, doubly-linked list with a *sentinel* (or *header*) node to reduce the number of "special cases" (e.g., inserting first item in an empty list). An empty list looks like:



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

b) Why would a circular, doubly-linked list with a *sentinel* (or *header*) node eliminate the "special cases" of "inserting first item in an empty list" or "removing the last item from a list."

After answering the above questions, raise you hand and explain your answers.

Part B: The position-base operations described in Tables 16.4 and 16.5 of the Lambert textbook treat the cursor poorly. For example, the `remove` operation is described as:

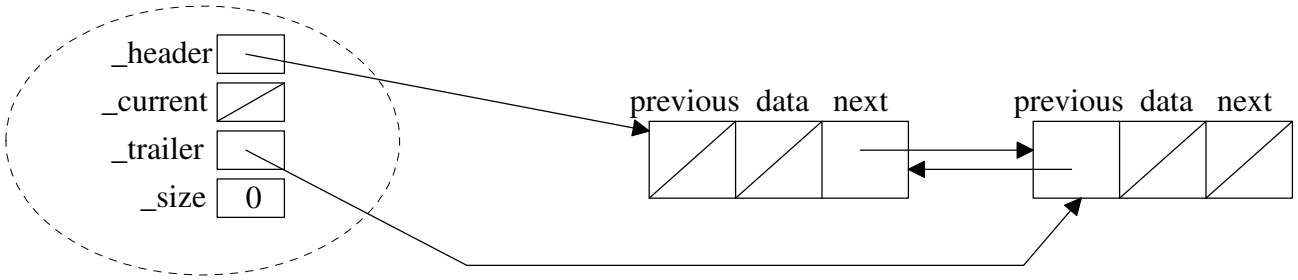
"*Precondition*: There have been no intervening `insert` or `remove` operations since the most recent `next` or `previous` operation. Removes the item returned by the most recent `next` or `previous`."

Typically, if a method has a precondition, then there is another method to check the precondition. For example, the `next` operation has a precondition: "`hasNext` returns `True`." However, the `remove` operation's precondition cannot be checked without the user application maintaining a list of preceeding operations -- a bad design! The problem occurs because the `insert` and `remove` operations leave the cursor in an undefined state.

Part B: Instead of thinking of a cursor between to list items, let's have a *current item* which is always defined as long as the list is not empty. We will insert and delete relative to the curent item.

Positional-based operations	**Description of operation**
`L.getItem()`	Returns the current item without removing it or changing the current position. Precondition: the list is not empty.
`L.hasNext()`	Returns True if the current item has a next item; otherwise return False. Precondition: the list is not empty.
`L.next()`	Precondition: hasNext returns True. Postcondition: The current item is has moved right one item
`L.hasPrevious()`	Returns True if the current item has a previous item; otherwise return False. Precondition: the list is not empty.
`L.previous()`	Precondition: hasPrevious returns True. Postcondition: The current item is has moved left one item
`L.first()`	Makes the first item the current item. Precondition: the list is not empty.
`L.last()`	Makes the last item the current item. Precondition: the list is not empty.
`L.insertAfter(item)`	Inserts item after the current item, or as the only item if the list is empty. The new item is the current item.
`L.insertBefore(item)`	Inserts item before the current item, or as the only item if the list is empty. The new item is the current item.
`L.replace(newValue)`	Replaces the current item by the newValue. Precondition: the list is not empty.
`L.remove()`	Removes and returns the current item. Making the next item the current item if one exists; otherwise the tail item in the list is the current item. Precondition: the list is not empty.

The `mypositionalList.py` file contains a `LinkedPositionalList` class, which uses a doubly-linked list with a *header* node and *trailer* node to reduce the number of "special cases" (e.g., inserting first item in an empty list). An empty list looks like:



After implementing and testing your LinkedPositionalList class, raise you hand and demonstrate your code.