Linked List Assignment

Marks: 25

The following questions assume that you have the example we have been working on in class. Although your ListNode and ListTest classes are important for the development of your solution only print out and hand in the modified version of the LinkedList class. Make sure you are using proper style in your final version.

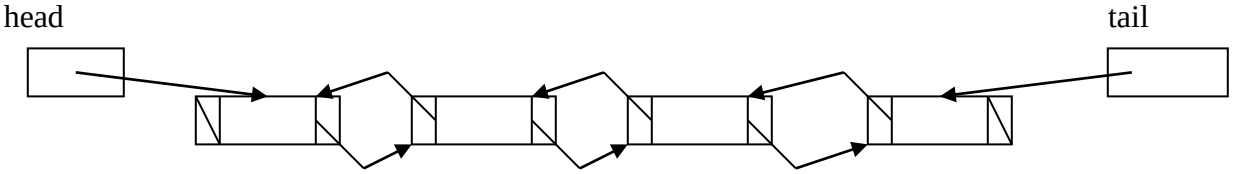
---

1. It is common to use Linked Lists to implement Stacks. A Stack is a data structure that orders its elements Last In First Out (LIFO.) Although it seems odd to reverse the order there are a number of situations where it is useful (e.g. an undo feature in paint.) Stacks have their own unusual terminology, the methods to operate on stacks are:

   push – add an element to the top of a stack.
   pop – remove an element from the top of the stack and return the element.

   Add push and pop to the LinkedList Class. (2 marks)

2. Another common use for Linked Lists is to implement a Queue. A Queue is a data structure that orders its elements First In First Out (FIFO.) A Queue is very common in the real world wherever you need to wait your turn in line (like waiting for food at the Cafeteria, except with computers there is no "cutting".) The problem with implementing a queue in our current data structure is that it would be inefficient to add new elements, O(n), because we would need to find the location of the last element so we could add our new element to it. We really want O(1) insertion time. To get this we need to modify the basic structure of our linked list to make it a Doubly Linked List. In a Doubly Linked List each node keeps track of both the next and previous node, and the Linked List itself keeps track of both a **head** and a **tail**. For Example:



   Modify Your Linked List to make it a doubly linked list. This will require changes to your node class, adding a tail reference and making changes to your add method so that when you add a new element this new structure is preserved. Also add two new methods to implement our queue functionality:

   enqueue – Add an element to the back of the list (at tail)
   dequeue – remove an element from the front of the queue and return the element. (4 marks)

3. Using your newly created doubly linked list, create three delete methods:

   delete – takes either an integer or ListNode reference (Yes, this means you need to overload delete. Yes, this means that this counts as two of the three methods) and removes that element from the list without destroying the integrity of the list.
   deleteAt – takes the position of the element to delete. (4 marks)

4. Linked lists are typically used in one of three situations:
    i. When we want the order of the nodes to be processed in LIFO order.
    ii. When we want the order of the nodes to be processed in FIFO order.
    iii. When we don't care about the order. Typically this is when any meaningful operation on our data involves accessing every element.

   Despite this, it is possible to maintain our list in sorted order. To do this, add a new method that inserts elements in sorted order. This must be done in O(n) time.

   sortedInsert – adds a new node into the correct position assuming that the list is already in sorted order (ascending from tail to head) (4 marks)

5. Create a method called **removeDuplicates** that removes the duplicates from the list. (4 marks)

6. Create a method called **reverse** that reverses the order of the nodes in the list. (4 marks)

7. Create a method called clone that returns a new copy of the list. Because your linked list simply refers to Node object in memory simply writing "**return this**" will not give you the answer you need because your new list will point to your old nodes and any changes to one list will affect both. You need to recreate every node in the list to prevent this. (3 marks)