Linked Lists

1. Describe an algorithm that reverses a singly linked list using only constant extra space. Note: you cannot use recursion to solve this problem (the function call stack involved in recursion has a space cost).

   Hint: use a new linked list.

   Initialize 2 nodes (call them 'previous' and 'after') as null (at beginning of list) and another (call it 'current') as head. Then, in a loop while current is not null, after = current's next, current's next = previous, previous = current, and current = after. After the loop, let head = previous.

2. Suppose that you have a pointer to a node in a singly linked list that is guaranteed not to be the last node in the list. You do not have pointers to any other nodes (except by following links). Describe an $O(1)$ algorithm that logically removes the *value* stored in such a node from the linked list, maintaining the integrity of the linked list.

   Hint: involve the next node.

   Let said pointer be called 'delete' and initialize current = head. In loop while current's next != delete, current = current's next. Then, current's next = delete's next.
   (Or, if we don't need to start current pointer at head then just initialize current's next at delete. Can we do that?)

3. Lazy deletion is a style of deletion whereby data isn't actually deleted from the data structure. Rather, it is simply marked as deleted. The rest of the data structure is then updated to ignore the marked elements as if they had been deleted. Would it be worthwhile to add lazy deletion to our linked list implementation? Why or why not?

   For "why," it is simpler to code, it costs less time to mark as deleted than to change pointers, and there is the potential to more easily "undelete". For "why not," it takes extra space to do the marking and takes up more nodes to be traversed with the rest.

4. Linked lists can be kept sorted by finding the correct position when inserting. For a sorted list, how does the time complexity of the `insert(...)`, `contains(...)`, `remove(...)` operations change?

   If these operations represent the search part, then they are (worst case) O(n) (w/o being sorted) and O(1) otherwise. For sorted, they are still O(n). I imagine change would be formulated as O(n)/O(n) = O(1), since O(n) - O(n) = O(n).

5. Design an $O(n)$ algorithm to determine whether a linked list of unknown size contains a cycle. You may only use $O(1)$ extra space.

   Note: A linked list contains a cycle if, starting from some node p, following a sufficient number of `next` links brings us back to node p. Node p does not have to be the first node in the list.

   Hint: use two pointers that are initially at the start of the list, but advance at different speeds.

Initialize pointer 'fast' and 'slow' at head. Then, fast = fast's next twice and slow = slow's next. In a loop while fast != slow, fast = fast's next twice, slow = slow's next and if fast's next = null, then print "no loop" and end the algorithm. When exiting the program from fast = slow, print "loop".

Stacks and Queues

6. Write the contents of each data structure after the following operations are performed:

```
add(1)
add(2)
remove()
add(3)
add(4)
remove()
remove()
add(5)
```

   (a) Stack

   (5, 1) where 5 is the top.

   (b) Queue

   (4, 5) where 4 is the front and 5 is the rear.

7. Describe why queues backed by linked lists perform better than queues backed by arrays.

   Enqueuing takes at worst O(1) time for linked list and at best O(1) for arrays and at worst O(n) time.

8. The time complexity for `size(...)` in an inefficient implementation of a stack or queue is $O(n)$ (iterate through every node and count how many there are). Describe an $O(1)$ algorithm to return the size of a container data structure such as a stack, queue, or linked list.

   Hint: keep track of the size independently, do not calculate it every time.

   Just use a counter variable that adds (subtracts) one to (from) itself for each added (removed) object. Then access that variable in O(1) time.

9. Describe a method to store two separate stacks within a single array. Your method should be able to push and pop from both stacks independently. Be as space efficient as possible.

   Have the stacks back to back pushing and popping in opposite directions.