Linked Lists

1. Describe an $O(n)$ 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: it might help to use a new linked list.

   Solution: Start a pointer at the front of the list. On each iteration, add the pointer's node to to the front of a new linked list and advance the pointer to the next node until the end of the list is reached. The new linked list will be in reverse order of the original.

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.

   Solution: Copy the data from the next node to the node to be deleted and delete the next node.

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?

   Solution: Lazy deletion wouldn't be helpful in a linked list. The effort to delete a node is constant (modification of a few pointers) with respect to the number of elements in the list. Lazily deleting nodes would simply complicate the other linked list operations by adding code to ignore them and keep memory allocated that could have been reclaimed by the operating system.

4. Linked lists can be kept sorted by finding the correct position when inserting. For a sorted list, how does the time complexity of insertion, removal, and checking to see if a value is in the list?

   Solution: The time complexity for `remove(...)` doesn't change. The time complexity for insertion will now become $O(n)$ because we have to find the correct position to insert the item. At worst case, we will be inserting the maximum element into the list each time and will have to scan the entire list to the end to find the correct location. The time complexity for `contains(...)` will remain $O(n)$ but we will be able to cut off the search earlier when we find an item greater than the one we are searching for.

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.

   Solution: Start two pointers at the start of the list. On each iteration, Pointer A advances ahead by one node and Pointer B advances ahead by two nodes. If Pointer A and Pointer B are ever equal while iterating through the list, there is a cycle.

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

Solution: 5 1

(b) Queue

Solution: 4 5

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

Solution: Enqueuing on an array is easy. Simply add the new element onto the end of the array. However, when it comes time to dequeue an element, popping off from the front of the array leaves an open array element that is no longer used. Frequent enqueue and dequeue operations would leave a large amount of space unused at the front of the array that would eventually need to be reclaimed. One might be able to alleviate this issue by wrapping the end of the queue around to the beginning of the array using modulo arithmetic. However, in this case, another problem in the array implementation arrises: the array will need to be resized if we enqueue more nodes than the array's capacity can hold. In such an event, we would have to resize the array to hold the extra nodes. These issues don't come up in a linked list implementation of a queue.

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.

Solution: Add a `size` member variable to our data structure. In the `insert(...)` function, increment the `size` member variable. Similarly, in the `remove(...)` function, decrement the `size` member variable. When `size(...)` is called, we can simply return the value of the `size` member variable.

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.

Solution: Each stack begins on the extreme ends of the array. The first stack starts from the leftmost element and grows towards the right. The second stack starts from the rightmost element and grows towards the left. Both stacks grow (or shrink) in opposite directions. To check for overflow, all we need to check is for space between top elements of both stacks.