# Problem 1: Find Intersection of Two Singly Linked List

*/\*\*  
 \* Find Intersection Of Two Singly Linked Lists  
 \* Given two singly linked lists, find their intersection if any.  
 \* The intersection is the node where the two lists merge, the first node that  
 \* belongs to both lists  
 \*  
 \* \* Output: Function must return an integer, the value from the intersection node  
 \* or -1 if no intersection exists.  
 \*  
 \* Constraints:  
 \*  
 \* 0 <= values in the list nodes <= 10^9  
 \* 0 <= number of nodes in a list <= 10^5  
 \*  
 \* Example:  
 \*  
 \* list1 = 1 --> 2 --> 3 --> 4 --> 7 --> 8 --> 9  
 \* list2 = 5 --> 6 --> 4 --> 7 --> 8 --> 9  
 \*  
 \* Output: 4  
 \*  
 \* Size of list1 = 7  
 \* Size of list2 = 6  
 \*  
 \* Approach:  
 \* 1) Calculate size of both the list.  
 \* 2) Start the longer list from the difference in size.  
 \* 3) Compare one to one  
 \* 4) Move forward  
 \*  
 \* Time Complexity: O(n)  
 \* Space Complexity: O(1)  
 \*  
 \* resources/IntersectionOfTwoLinkedList.jpg  
 \*  
 \* We could start comparing from the first node of both lists.  
 \* If the two nodes are the same (same address) then it is the intersection node  
 \* else we advance both pointers to their next nodes and again compare and so on.  
 \*  
 \* If we reach the end of both lists, it means they do not intersect.  
 \*  
 \* Now suppose one list is longer than the other by X nodes. It is easy to see  
 \* that the intersection cannot be found among the first X nodes of the longer list.  
 \* X is easy to calculate by measuring the length of each list.  
 \* Then we can reduce the problem to the simpler one by skipping the first X nodes  
 \* in the longer list.  
 \*  
 \* Using this approach we would walk through the lists two times total: once for  
 \* measuring their lengths and once again for comparing the nodes.  
 \*  
 \*/*

# Problem 2: Find median of sorted circularly singly linked list.

*/\*\*  
 \* Find Median Of Sorted Circular Singly Linked List  
 \* Given a pointer to an arbitrary node in a sorted circular linked list,  
 \* find the median value of the elements.  
 \*  
 \* Example  
 \* Input: Pointer to node 4 in this circular linked list:  
 \*  
 \* 2 --> 4 --> 6 --> 8 --> 10 --> 2  
 \*  
 \* Output: 6  
 \* There are 5 nodes. The middle node in the sorted sequence of the elements  
 \* would be the median. Regardless of the node given as an input,  
 \* the answer would be 6 as long as the list is as pictured.  
 \*  
 \* Constraints:  
 \*  
 \* 1 <= number of elements <= 10^5  
 \* - 2 \* 10^9 <= any value in the list <= 2 \* 10^9  
 \* All the values in the input list are even.  
 \*  
 \* If the list has an even number of elements,  
 \* the median is the average of the two middle elements in the sorted sequence.  
 \*  
 \* Approach  
 \*  
 \* 1) Find the length of the linked list.  
 \* 2) Given pointer to node, find the first small node.  
 \* 3) Start from that first small node and find if the linked list is increasing  
 \* or decreasing  
 \* 4) If it's increasing, then find the median from first small node.  
 \* 5) If it's decreasing, then find the first big node.  
 \* 6) Find the median from the first big node.  
 \* 7) In case of even nodes, median will be the average of two middle nodes.  
 \* 8) To avoid Integer overflow, when adding two numbers use long.  
 \*  
 \* Time Complexity: O(n)  
 \* Space Complexity: O(1)  
 \*  
 \* a) sorting order can be ascending or descending or  
 \* b) while any one value from the list is guaranteed to fit in int32,  
 \* a sum of two values might overflow int32.  
 \*  
 \* Now coming to the actual solution, a linked list will be one of these 3 types  
 \* 1) 2 -> 2 -> 2 -> 2 -> 2 -> 2  
 \* 2) 2 -> 2 -> 4 -> 6 -> 8 -> 8  
 \* 3) 8 -> 8 -> 6 -> 4 -> 2 -> 2  
 \*  
 \* First case is trivial.  
 \*  
 \* In the 2nd case if we can find the first smallest element (call it head)  
 \* then finding the median will be easy: just find the middle element/elements.  
 \* (2 -> 2 -> 4 -> 6 -> 8 -> 8) then head will be ([2]head -> 2 -> 4 -> 6 -> 8 -> 8)  
 \*  
 \* In the 3rd case if we can find the first largest element (call it head)  
 \* then finding the median will be easy: just find the middle element/elements.  
 \* (8 -> 8 -> 4 -> 6 -> 2 -> 2) then head will be ([8]head -> 8 -> 4 -> 6 -> 2 -> 2)  
 \*  
 \* resources/MedianOfCircularlySortedSinglyLinkedList1.jpg  
 \* resources/MedianOfCircularlySortedSinglyLinkedList2.jpg  
 \*/*

# Problem 3: Implement Queue using two stacks

*/\*\*  
 \* Implement Queue Using Two Stacks  
 \* Given a sequence of enqueue and dequeue operations, return a result of their  
 \* execution without using a queue data structure.  
 \*  
 \* Operations are given in the form of a linked list:  
 \*  
 \* A non-negative integer means “enqueue me”.  
 \* -1 means  
 \* If the queue is not empty, dequeue the current head and append it to the result.  
 \* If the queue is empty, append -1 to the result.  
 \*  
 \* Result is a linked list.  
 \* Use two stacks as an auxiliary data structure. Using a queue isn’t allowed.  
 \*  
 \* Example One  
 \* Input: 1, -1, 2, -1, -1, 3, -1  
 \* Output: 1, 2, -1, 3  
 \*  
 \* Example Two  
 \* Input: 0, 1, 2, -1, 3  
 \* Output: 0  
 \*  
 \* The only dequeue operation results in the first enqueued element,  
 \* 0, to be appended to the result list.  
 \*  
 \* Constraints:  
 \*  
 \* -1 <= value in operations list <= 2 \* 10^9  
 \* 1 <= N <= 10^5  
 \* There will be at least one dequeue (-1) operation.  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \*  
 \* Approach:  
 \*  
 \* 1) Use two stacks - stack1 and stack2  
 \* 2) All enqueue operations push an element to stack1 in O(1) time.  
 \* 3) For deque operation, move all elements from stack1 to stack2.  
 \* 4) Then pop from stack2 for each deque operation until stack2 is empty.  
 \* 5) After that repeat step3 if stack2 is empty.  
 \*  
 \* Time Complexity: O(n) - Atmost n elements are moved from stack1 to stack2  
 \* Space Complexity: O(n) - Atmost we are storing n elements in stack.  
 \*  
 \* resources/ImplementQueueUsingTwoStacks1.jpg  
 \* resources/ImplementQueueUsingTwoStacks2.jpg  
 \*/*

**optimal\_solution.cpp**

Let’s imagine that we started with the brute force solution described above and came to this situation:

stack1 = [1, 2, 3], stack2 = [], next operation: -1

To dequeue, we move all numbers from stack1 to stack2:

stack1 = [], stack2 = [3, 2], add append number 1 to the result.

Now we can notice that stack2 has all the remaining numbers in the order that’s perfect for us. For example, if the next operation is -1, we can simply pop and return number 2 from stack2 - a constant time operation.

We can dequeue elements this way if we leave them in stack2, but what about enqueueing new ones? It turns out that we can push them in stack1, and they can remain there until stack2 is empty. Once stack2 is empty and another dequeue operation comes, we can do what was described two paragraphs ago: pop all numbers from stack1 and push them into stack2.

**Time Complexity:**

Enqueue operation takes constant time, clearly. Let us examine dequeue operation.

Most dequeue operations will just need to pop one number from stack2, that’s constant time. Some dequeue operations however will need to move some numbers from stack1 to stack2.

Amortized time complexity of the dequeue operation is constant and intuitively we can observe that we never move any given number more than once between stack1 and stack2 (that’s constant time per number).

Overall time complexity of the algorithm would be O(N) since we process N operations, taking constant time per operation (amortized).

**Auxiliary Space Used:**

O(N) as we never store more than N numbers in our stacks.

**Space Complexity:**

O(N).