Approach 1: Two-Pointers Iteration

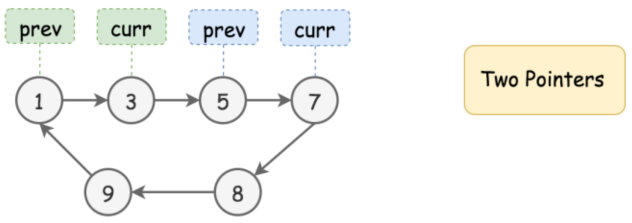
Intuition

As simple as the problem might seem to be, it is actually not trivial to write a solution that covers all cases.

Often the case for the problems with linked list, one could apply the approach of **Two-Pointers Iteration**, where one uses two pointers as surrogate to traverse the linked list.

One of reasons of having two pointers rather than one is that in singly-linked list one does not have a reference to the precedent node, therefore we keep an additional pointer which points to the precedent node.

For this problem, we iterate through the cyclic list using two pointers, namely `prev` and `curr`. When we find a suitable place to insert the new value, we insert it between the `prev` and `curr` nodes.



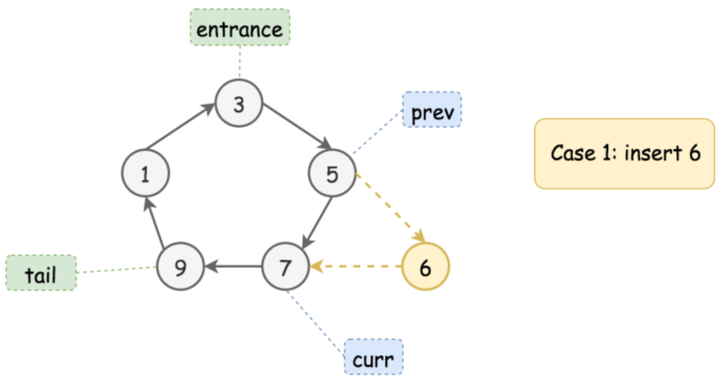
Algorithm

First of all, let us define the skeleton of two-pointers iteration algorithm as follows:

- As we mentioned in the intuition, we *loop over* the linked list with two pointers (*i.e.* `prev` and `curr`) step by step. The termination condition of the loop is that we get back to the starting point of the two pointers (*i.e.* `prev == head`)
- During the loop, at each step, we check if the current place bounded by the two pointers is the right place to insert the new value.
- If not, we move both pointers one step forwards.

Now, the tricky part of this problem is to sort out different cases that our algorithm should deal with within the loop, and then design a *concise* logic to handle them sound and properly. Here we break it down into *three* general cases.

Case 1). The value of new node sits between the minimal and maximal values of the current list. As a result, it should be inserted within the list.

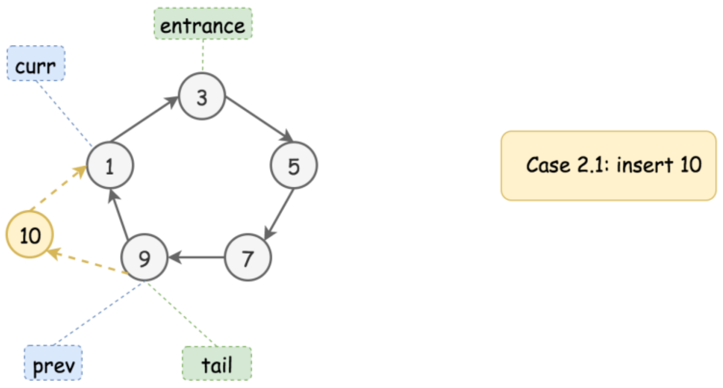


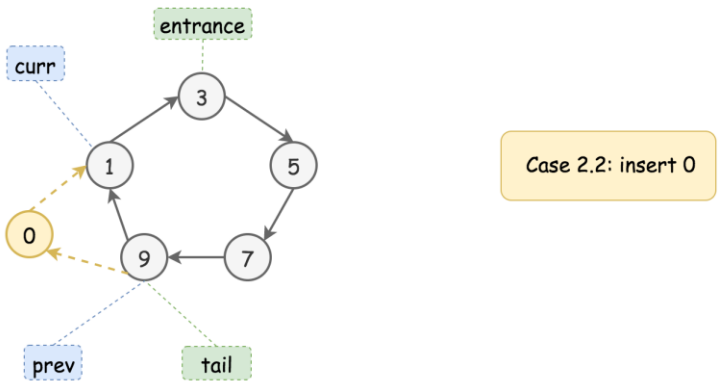
As we can see from the above example, the new value (6) sits between the minimal and maximal values of the list (*i.e.* 1 and 9). No matter where we start from (in this example we start from the node {3}), the new node would end up being inserted between the nodes {5} and {7}.

The condition is to find the place that meets the constraint of {`prev.val <= insertVal <= curr.val`}.

Case 2). The value of new node goes beyond the minimal and maximal values of the current list, either less than the minimal value or greater than the maximal value. In either case, the new node should be added right after the *tail* node (*i.e.* the node with the maximal value of the list).

Here are the examples with the same input list as in the previous example.





Firstly, we should locate the position of the **tail** node, by finding a descending order between the adjacent, *i.e.* the condition of {`prev.val > curr.val`}, since the nodes are sorted in ascending order, the tail node would have the greatest value of all nodes.

Furthermore, we check if the new value goes beyond the values of tail and head nodes, which are pointed by the `prev` and `curr` pointers respectively.

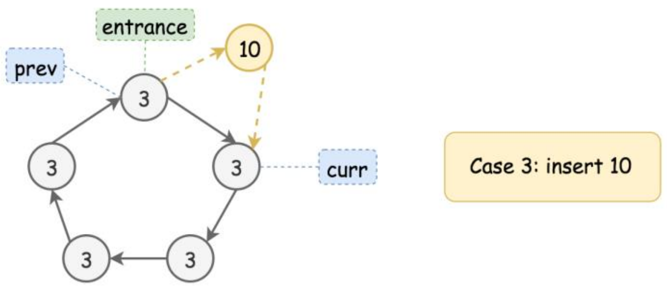
The Case 2.1 corresponds to the condition where the value to be inserted is *greater than or equal to* the one of tail node, *i.e.* `{insertVal >= prev.val}`.

The Case 2.2 corresponds to the condition where the value to be inserted is *less than or equal to* the head node, *i.e.* `{insertVal <= curr.val}`.

Once we locate the tail and head nodes, we basically ***extend*** the original list by inserting the value in between the tail and head nodes, *i.e.* in between the `prev` and `curr` pointers, the same operation as in the Case 1.

Case 3). Finally, there is one case that does not fall into any of the above two cases. This is the case where the list contains uniform values.

Though not explicitly stated in the problem description, our sorted list can contain some duplicate values. And in the extreme case, the entire list has only one single unique value.



In this case, we would end up looping through the list and getting back to the starting point.

The followup action is just to add the new node after any node in the list, regardless the value to be inserted. Since we are back to the starting point, we might as well add the new node right after the starting point (our entrance node).

Note that, we cannot skip the iteration though, since we have to iterate through the list to determine if our list contains a single unique value.

The above three cases cover the scenarios within and after our iteration loop. There is however one minor **corner** case we still need to deal with, where we have an **empty** list. This, we could easily handle before the loop.

Complexity Analysis

- Time Complexity: O(N)\mathcal{O}(N)O(*N*) where NN*N* is the size of list. In the worst case, we would iterate through the entire list.
- Space Complexity: O(1)\mathcal{O}(1)O(1). It is a constant space solution.
-