# Starting point of loop in a Linked List

In this article, we will learn how to solve the most asked coding interview question: "**Starting point of loop in a Linked List**"

**Problem Statement:** Given the head of a linked list, return *the node where the cycle begins. If there is no cycle, return* null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that the tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

**Examples:**

**Example 1:**

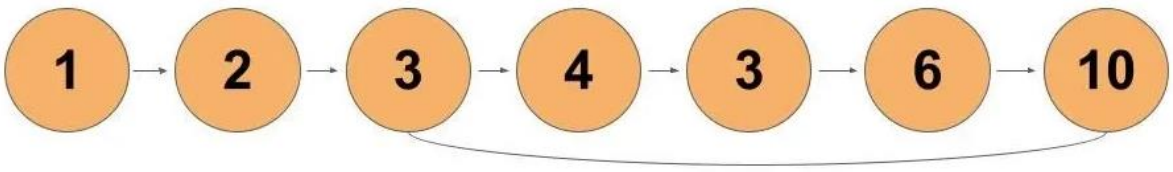
**Input:**

```
 head = [1,2,3,4,3,6,10]
```

**Output:**

```
 tail connects to node index 2
```

**Explanation:**




**Example 2:**

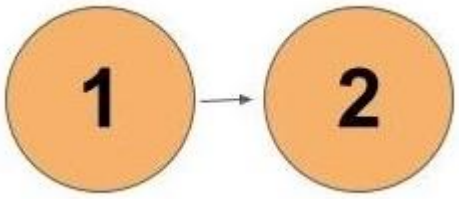
**Input:**

```
 head = [1,2]
```

**Output:**

```
no cycle
```

**Explanation:**




**Solution:**

**Solution 1: Brute Force**

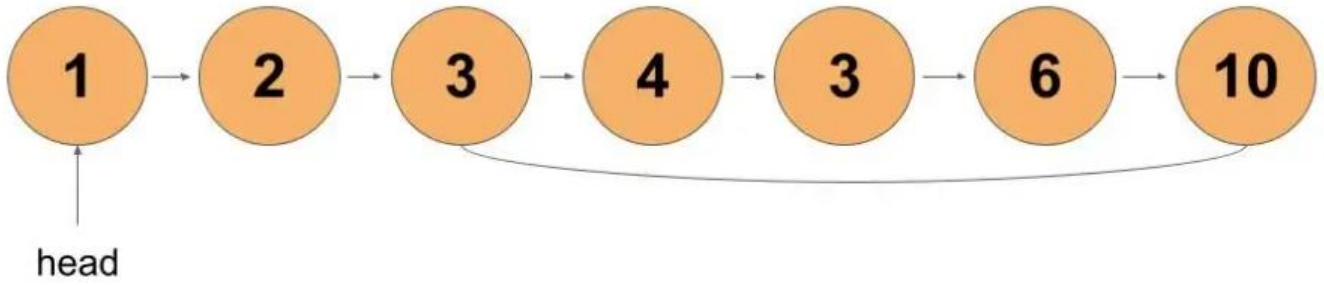
**Approach:**

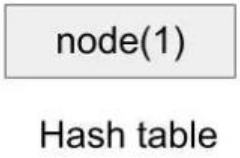
We can store nodes in a hash table so that, if a loop exists, the head will encounter the same node again. This node will be present in the table and hence, we can detect the loop. Steps are:-

- Iterate the given list.
- For each node visited by head pointer, check if node is present in the hash table.
- If yes, loop detected
- If not, insert node in the hash table and move the head pointer ahead.
- If head reaches null, then the given list does not have a cycle in it.

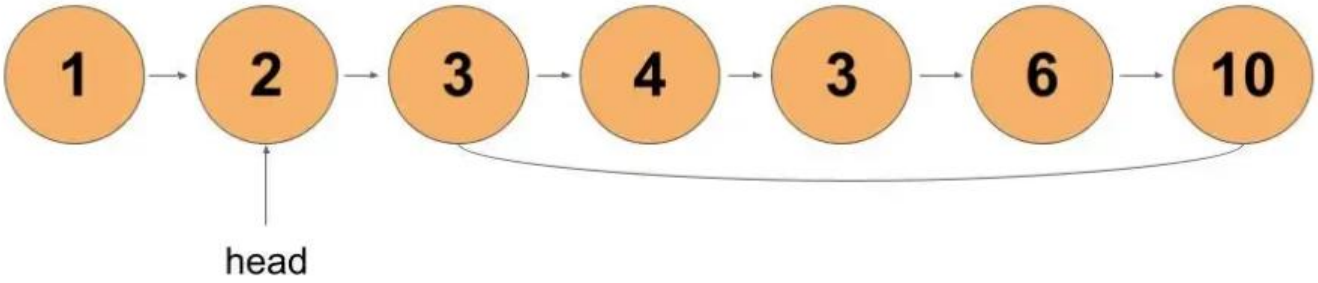
**Dry Run:**

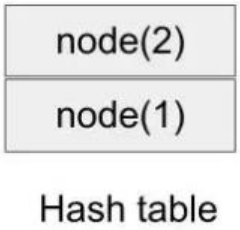
We start iterating each node and storing nodes in the hash table if an element is not present.



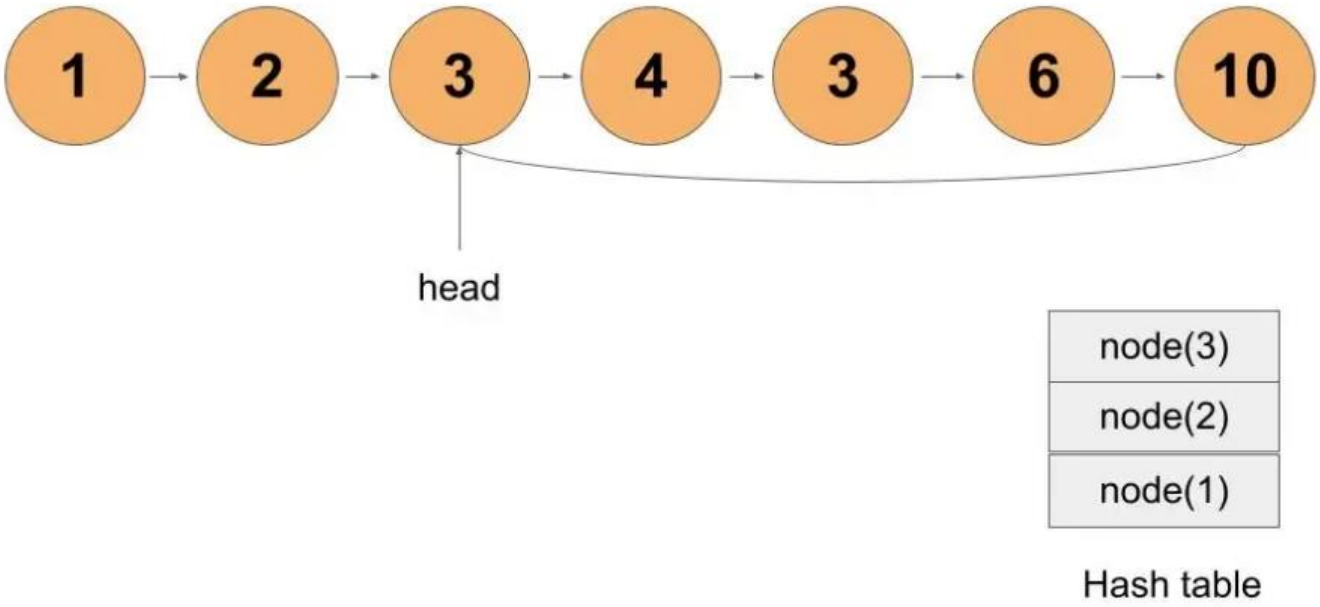




Node(1) is not present in the hash table. So, we insert a node in it and move head ahead.

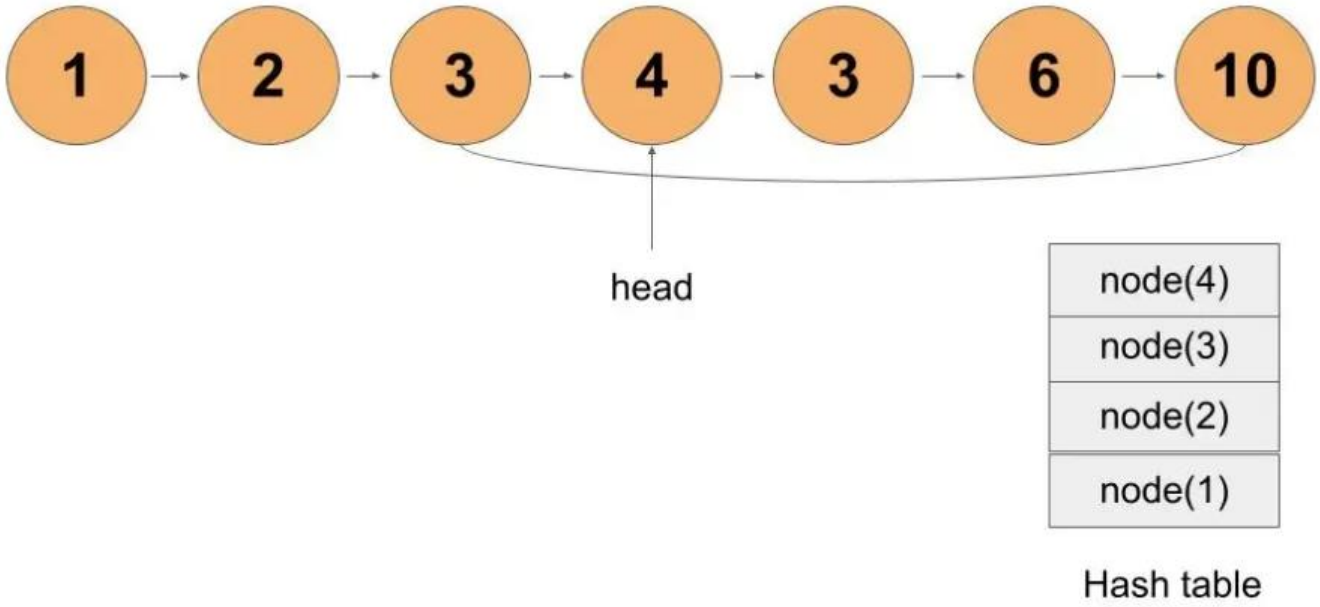


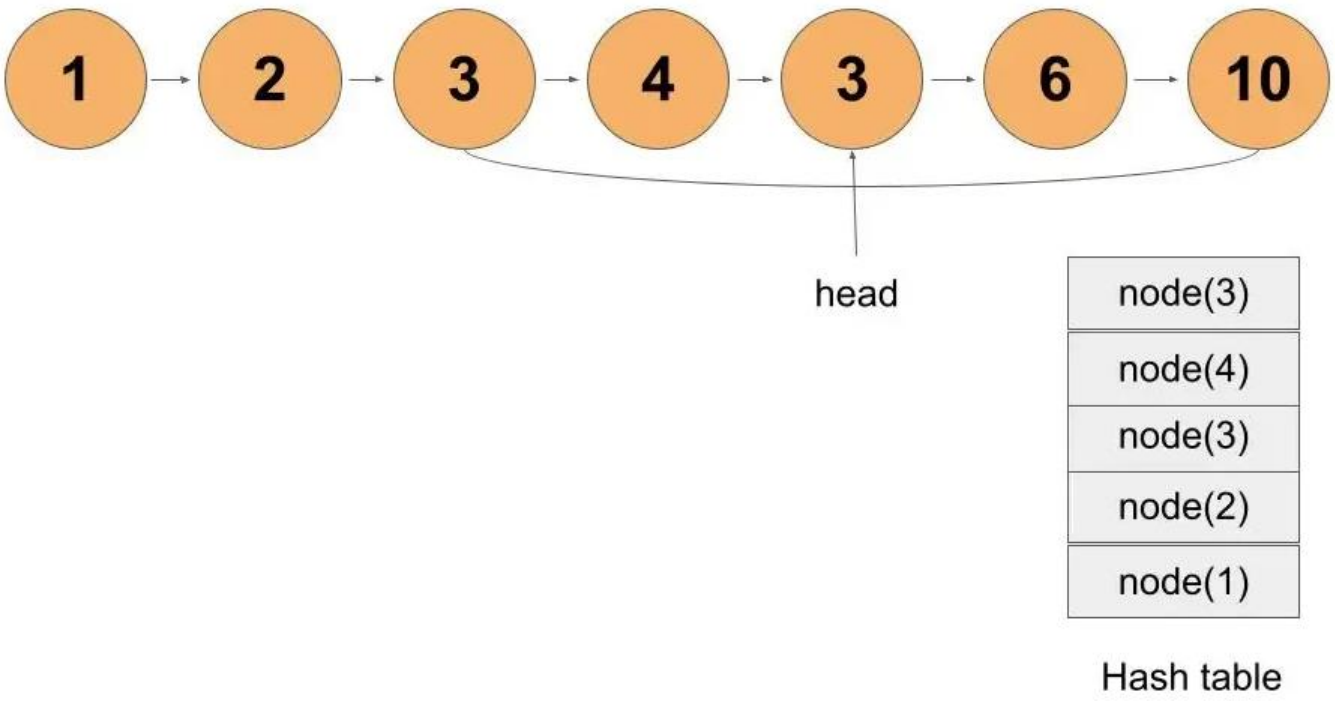

Node(2) is not present in the hash table. So, we insert a node in it and move head ahead.



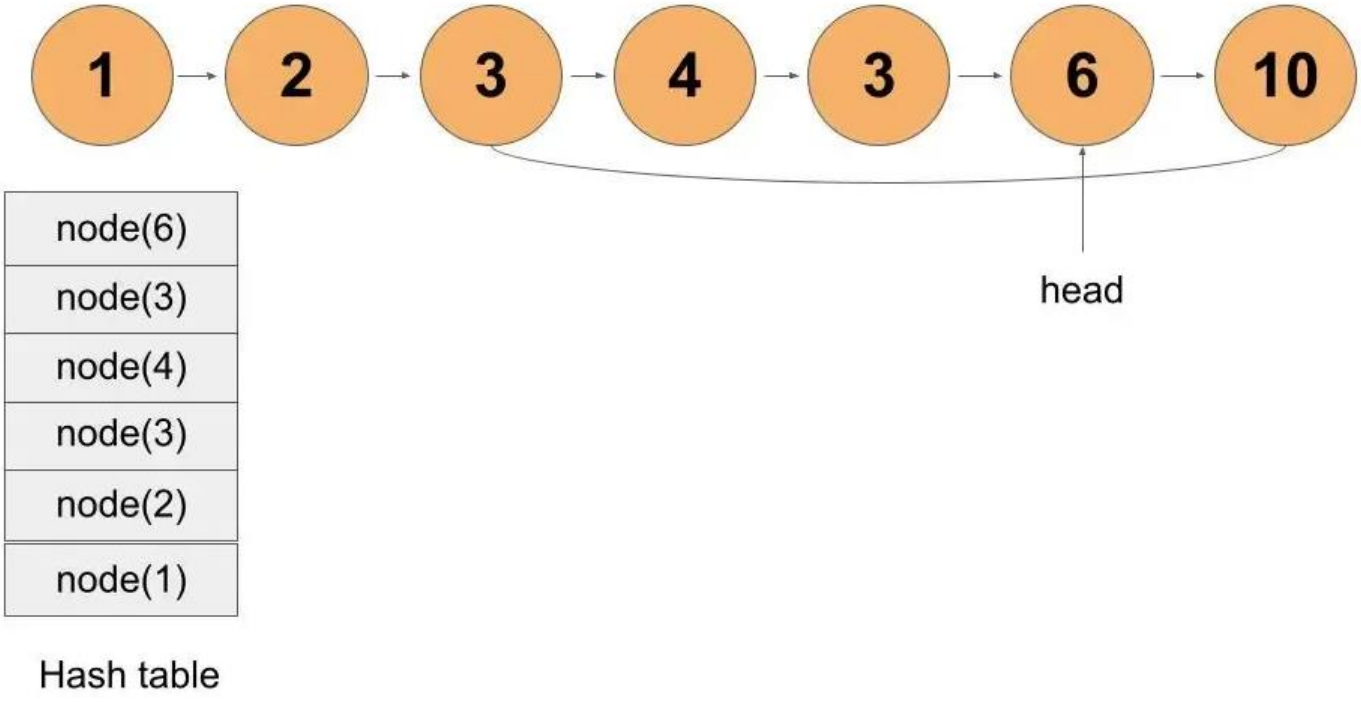

Node(3) is not present in the hash table. So, we insert a node in it and move head ahead.



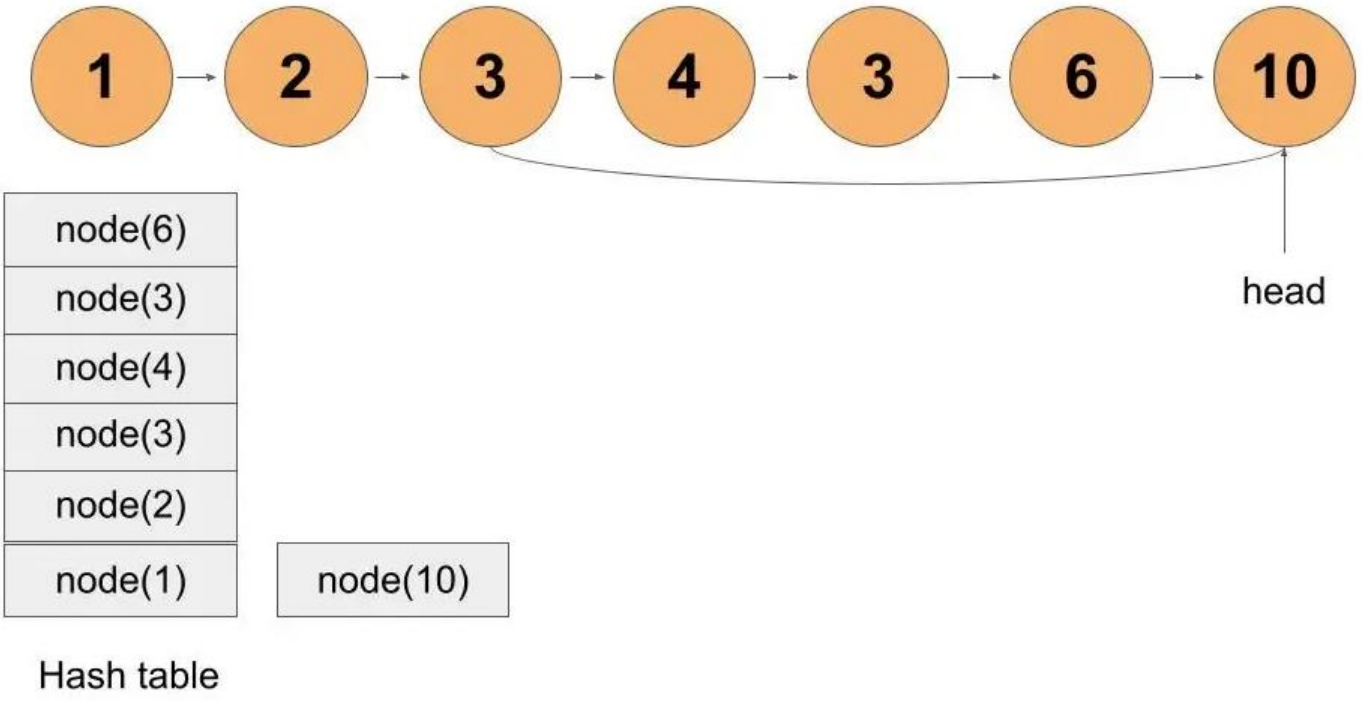

Node(4) is not present in the hash table. So, we insert a node in it and move head ahead.



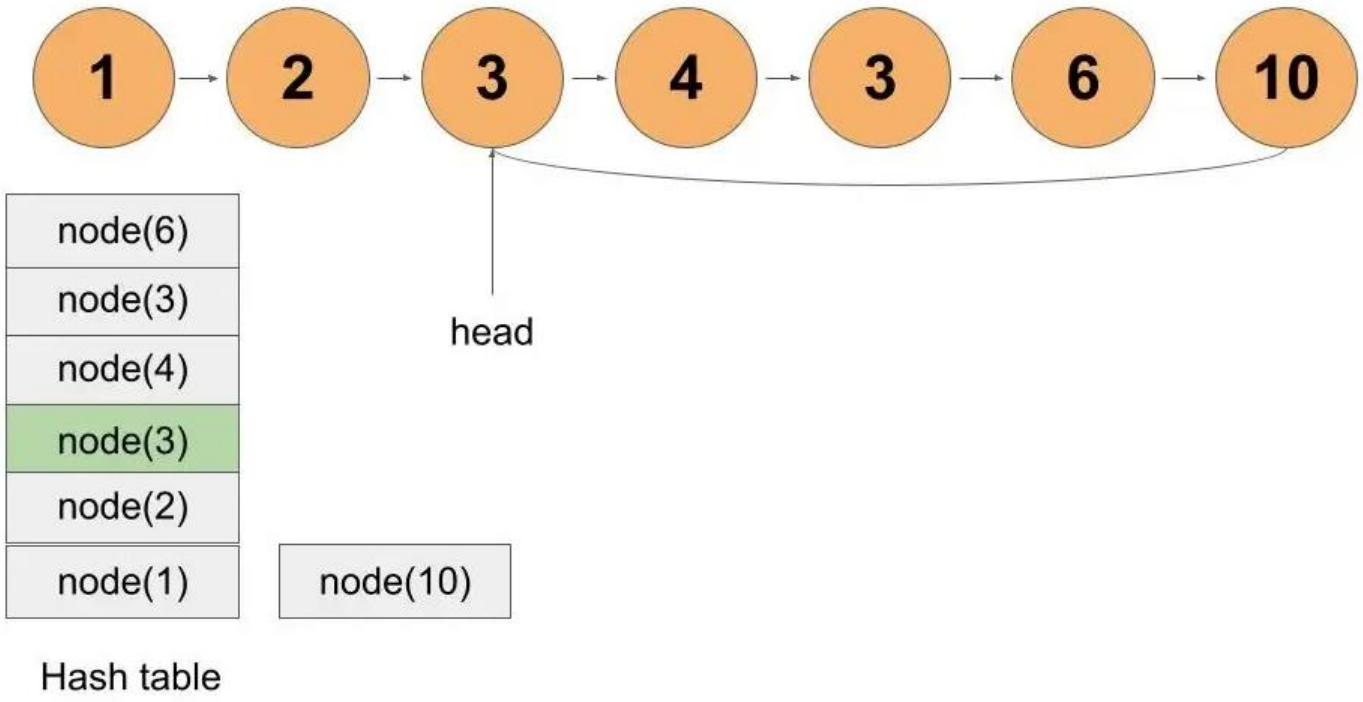

Node(3) is not present in the hash table. So, we insert a node in it and move head ahead. Though this node contains 3 as a value, it is a different node than the node at position 3.




Node(6) is not present in the hash table. So, we insert a node in it and move head ahead.




Node(10) is not present in the hash table. So, we insert a node in it and move head ahead.




We reached the same node which was present in the hash table. Thus, the starting node of the cycle is node(3).

**Code:**

```
static Node detectCycle(Node head) {
    HashSet<Node> st=new HashSet<>();
    while(head != null) {
        if(st.contains(head)) return head;
        st.add(head);
        head = head.next;
    }
    return null;
}
```

**Time Complexity:** O(N)

*Reason*: Iterating the entire list once.

**Space Complexity:** O(N)

**Solution 2: Slow and Fast Pointer Method**

**Approach:**

The following steps are required:

- Initially take two pointers, fast and slow. Fast pointer takes two steps ahead while slow pointer will take single step ahead for each iteration.
- We know that if a cycle exists, fast and slow pointers will collide.
- If cycle does not exists, fast pointer will move to NULL
- Else, when both slow and fast pointer collides, it detects a cycle exists.
- Take another pointer, say entry. Point to the very first of the linked list.
- Move the slow and the entry pointer ahead by single steps until they collide.
- Once they collide we get the starting node of the linked list.

But why use another pointer, entry?

Let's say a slow pointer covers the L2 distance from the starting node of the cycle until it collides with a fast pointer. L1 be the distance traveled by the entry pointer to the starting node of the cycle. So, in total, the slow pointer covers the L1+L2 distance. We know that a fast pointer covers some steps more than a slow pointer. Therefore, we can say that a fast pointer will surely cover the L1+L2 distance. Plus, a fast pointer will cover more steps which will accumulate to nC length where cC is the length of the cycle and n is the number of turns. Thus, the fast pointer covers the total length of L1+L2+nC.

We know that the slow pointer travels twice the fast pointer. So this makes equation to

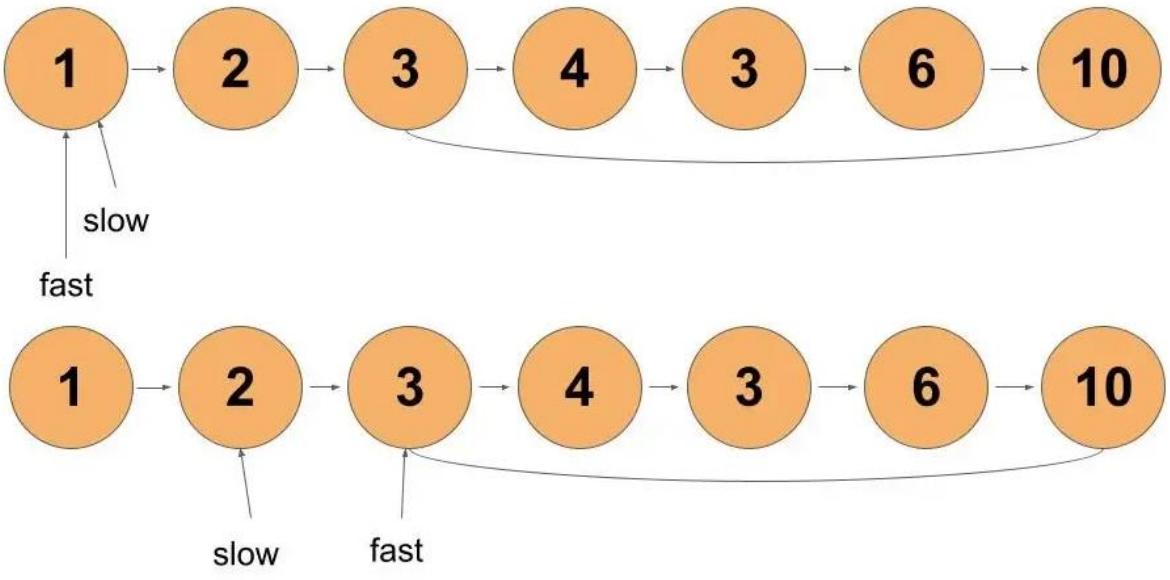
2(L1+L2) = L1+L2+nC. This makes equation to

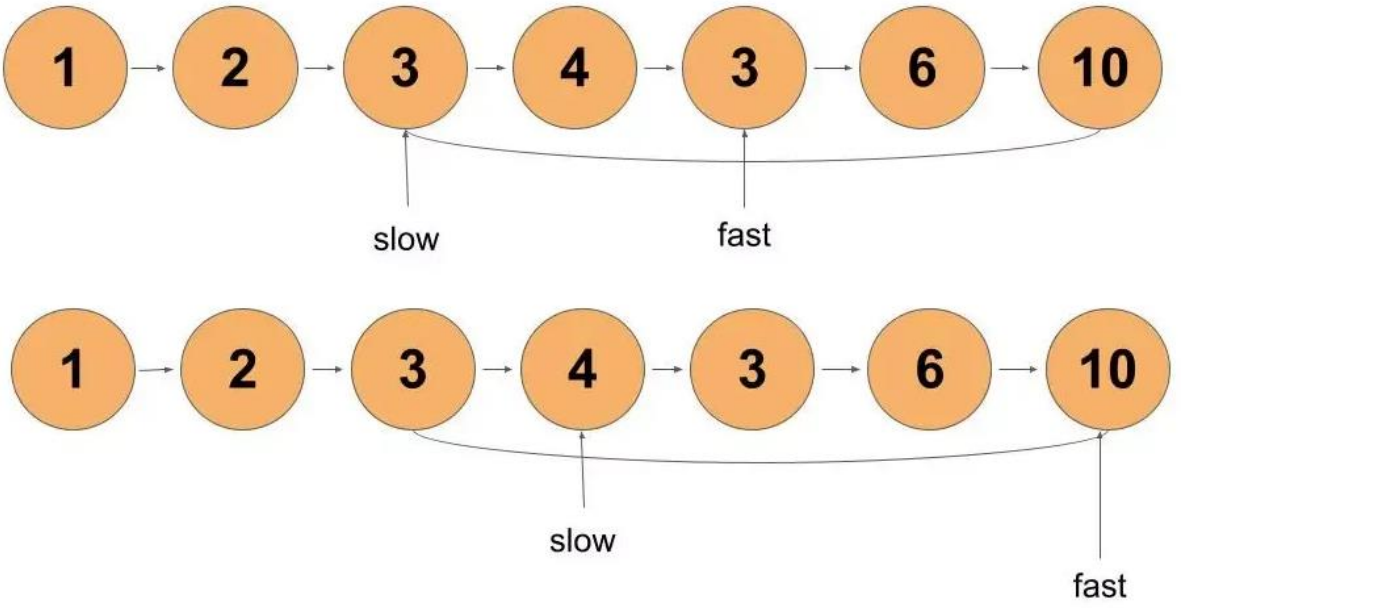
L1+L2 = nC. Moving L2 to the right side

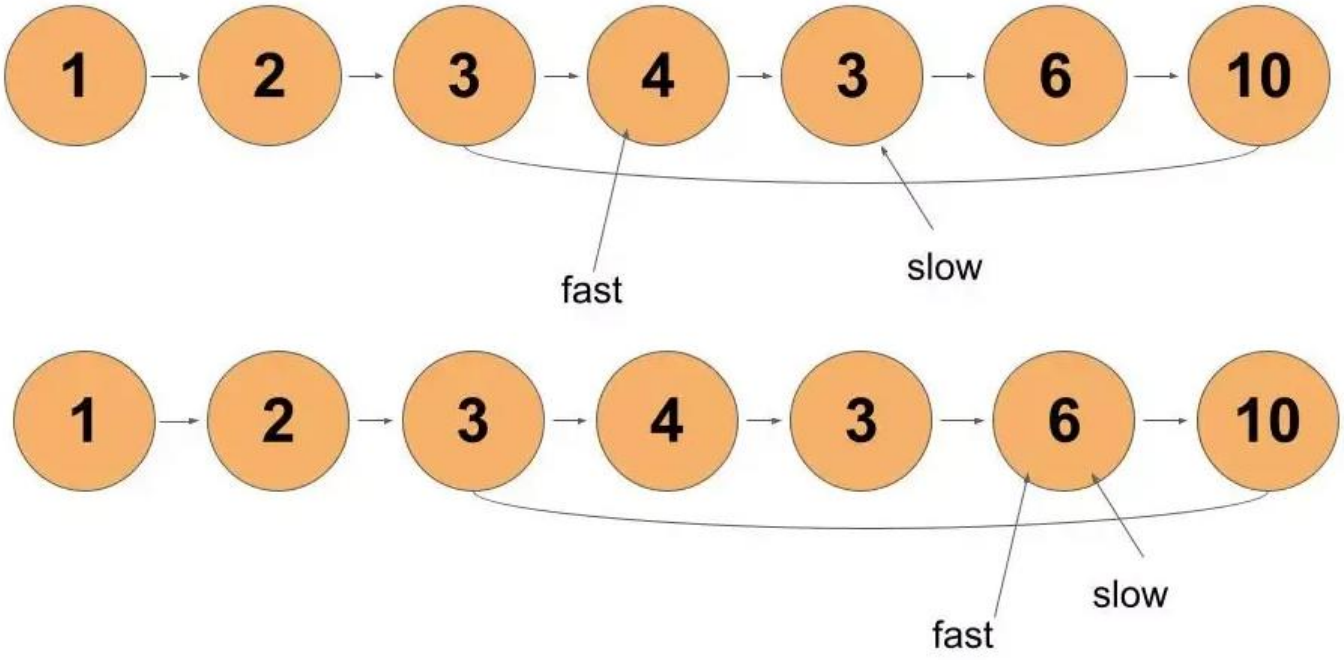
L1 = nC-L2 and this shows why the entry pointer and the slow pointer would collide.

**Dry Run:**

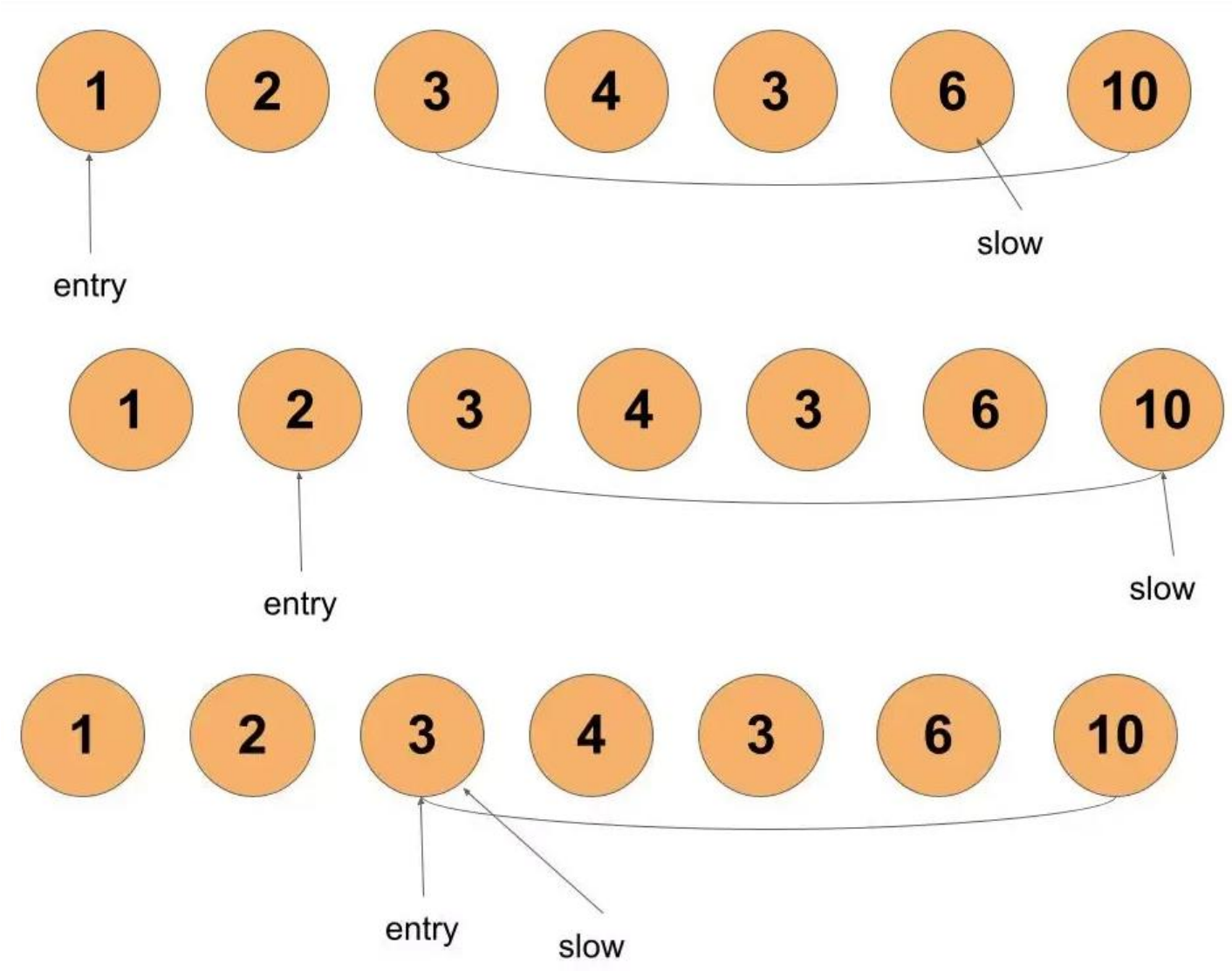
We initialize fast and slow pointers to the head of the list. Fast moves two steps ahead and slowly takes a single step ahead.






We can see that the fast and slow pointer collides which shows the cycle exists. The entry pointer is pointed to the head of the list. And move them forward until it collides with the slow pointer.




We see that both collide and hence, we get the starting node of the list.

**Code:**

```
public ListNode detectCycle(ListNode head) {
        ListNode slow=head;
        ListNode fast=head;
        boolean falg=true;
```

```
        while(fast!=null&&fast.next!=null)
        {
            fast=fast.next.next;
            slow=slow.next;
            if(fast==slow)
            {
                falg=false;
              break;
            }
        }
        slow=head;
        while(fast!=null&&slow!=fast)
        {
            slow=slow.next;
            fast=fast.next;
        }
        return falg?null:slow;
    }
```

**Time Complexity:** O(N)

*Reason*: We can take overall iterations club it to O(N)

**Space Complexity:** O(1)

*Reason*: No extra data structure is used.