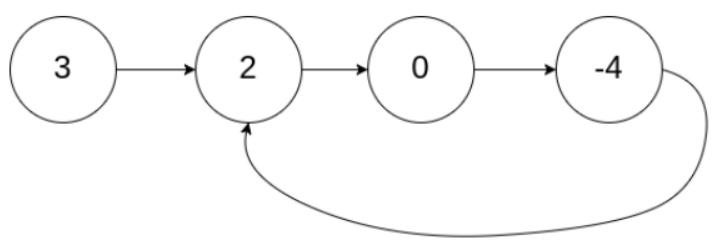
<https://leetcode.com/problems/linked-list-cycle-ii/>

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 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.

**Example 1:**

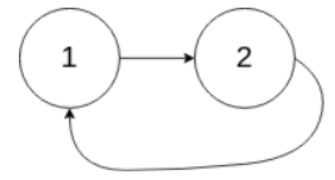


Input: head = [3,2,0,-4], pos = 1

Output: tail connects to node index 1

Explanation: There is a cycle in the linked list, where tail connects to the second node.

**Example 2:**



Input: head = [1,2], pos = 0

Output: tail connects to node index 0

Explanation: There is a cycle in the linked list, where tail connects to the first node.

**Example 3:**



Input: head = [1], pos = -1

Output: no cycle

Explanation: There is no cycle in the linked list.

**Constraints:**

* The number of the nodes in the list is in the range [0, 104].
* -105 <= Node.val <= 105
* pos is -1 or a **valid index** in the linked-list.

**Follow up:** Can you solve it using O(1) (i.e. constant) memory?

**Attempt 1: 2023-02-20**

**Solution 1: Fast and Slow pointer (30 min)**

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode detectCycle(ListNode head) {

boolean cycleExist = false;

ListNode slow = head;

ListNode fast = head;

while(fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if(slow == fast) {

cycleExist = true;

break;

}

}

if(cycleExist) {

while(head != slow) {

head = head.next;

slow = slow.next;

}

return head;

}

return null;

}

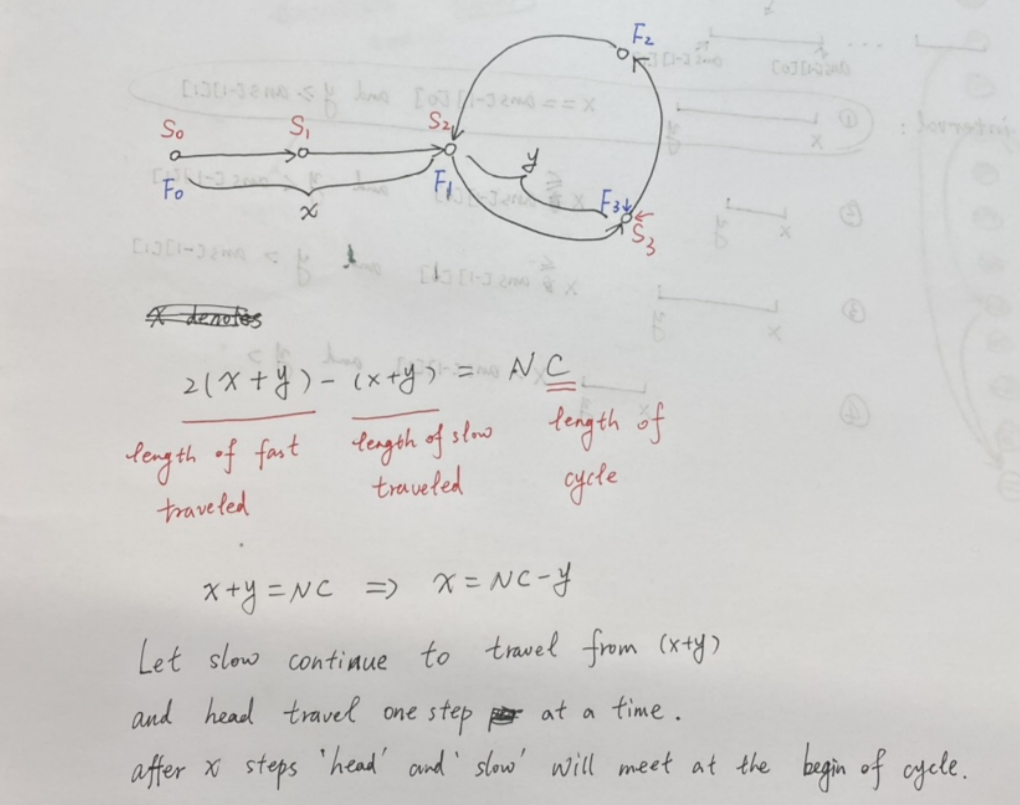
}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/linked-list-cycle-ii/solutions/1701128/c-java-python-slow-and-fast-image-explanation-beginner-friendly/>



slow moves 1 step at a time, fast moves 2 steps at a time.

when slow and fast meet each other, they must be on the cycle

x denotes the length of the linked list before starting the circle

y denotes the distance from the start of the cycle to where slow and fast met

C denotes the length of the cycle

when they meet, slow traveled (x + y) steps while fast traveled 2 \* (x + y) steps, and the extra distance (x + y) must be a multiple of the circle length C

note that x, y, C are all lengths or the number of steps need to move.

head, slow, fast are pointers.

head moves x steps and arrives at the start of the cycle.

so we have x + y = N \* C, let slow continue to travel from y and after x more steps, slow will return to the start of the cycle.

At the same time, according to the definition of x, head will also reach the start of the cycle after moving x steps.

so if head and slow start to move at the same time, they will meet at the start of the cycle, that is the answer.

public class Solution {

public ListNode detectCycle(ListNode head) {

ListNode slow = head, fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) break;

}

if (fast == null || fast.next == null) return null;

while (head != slow) {

head = head.next;

slow = slow.next;

}

return head;

}

}

Time Complexity: O(N)

Space Complexity: O(1)