Linked List Cycle(Day 20)

[Prepared By Azan Imtiaz](#)

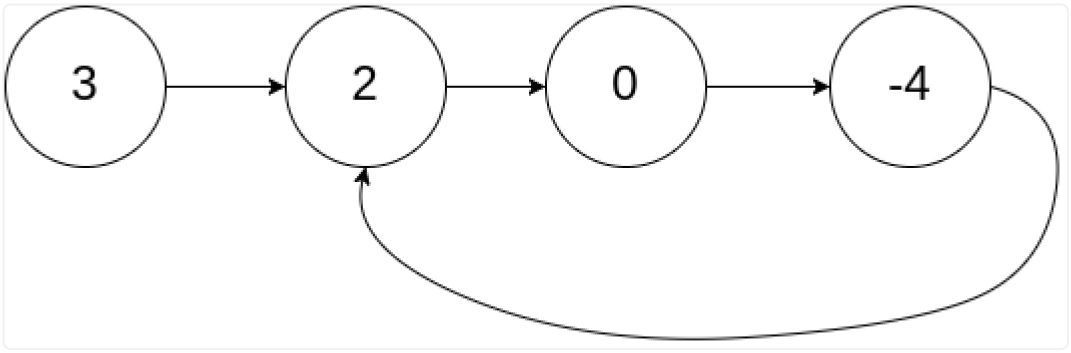
Linked List Cycle(Leatcode)

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

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. **Note that `pos` is not passed as a parameter**.

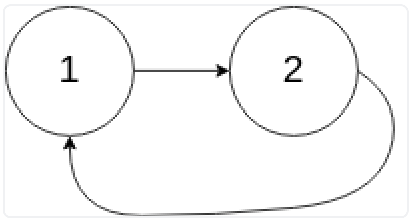
Return `true` *if there is a cycle in the linked list*. Otherwise, return `false`.

Example 1:



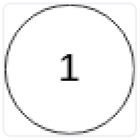
```
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st
node (0-indexed).
```

Example 2:



```
Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 0th
node.
```

Example 3:



```
Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.
```

Constraints:

- The number of the nodes in the list is in the range `[0, 10`^{4}`]`.
- `-10`^{5} `<= Node.val <= 10`^{5}
- `pos` is `-1` or a **valid index** in the linked-list.

Follow up: Can you solve it using `O(1)` (i.e. constant) memory? (**Check 2nd Solution**)

Problem

Detecting a cycle in a linked list is a common problem. A cycle occurs when a node's next pointer points to an earlier node, creating a loop.

Brute Force Intuition

The straightforward way to detect a cycle is to track each node we visit. If we encounter a node we've seen before, there's a cycle.

Approach

We use a HashSet to record visited nodes. As we traverse the list, we check if the current node is already in the set:

1. Initialize a HashSet.
2. Iterate through the linked list.
3. Check if the current node is in the HashSet.
4. If it is, a cycle exists.
5. If not, add the node to the HashSet and continue.
6. If we reach the end of the list without repeats, there's no cycle.

Solution Description

The provided code defines a method `hasCycle` that takes the head of a linked list and returns `true` if there's a cycle or `false` otherwise. It uses a HashSet to keep track of visited nodes, allowing us to detect cycles efficiently.

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        Set<ListNode> visitedNodes = new HashSet<>();
        ListNode current = head;

        while (current != null) {
            if (visitedNodes.contains(current)) {
                return true; // Cycle detected
            }
            visitedNodes.add(current);
            current = current.next;
        }

        return false; // No cycle detected
    }
}
```

Time and Space Complexity

The time complexity is O(n), where n is the number of nodes in the linked list, because we potentially visit each node once. The space complexity is also O(n) due to the HashSet storing every node we visit.

Problem with the Solution

While the solution works, it uses extra space proportional to the size of the linked list. This can be problematic for very large lists.

Follow-up Question

Can we solve this problem in O(1) space complexity?

Yes, we can solve this problem with O(1) space complexity using Floyd's Tortoise and Hare algorithm, which involves two pointers moving at different speeds.

Intuition

The idea behind this solution is to use two pointers moving at different speeds to traverse the linked list. The slower pointer moves one step at a time, while the faster pointer moves two steps at a time. If there's a cycle, the fast pointer will eventually meet the slow pointer. If there isn't a cycle, the fast pointer will reach the end of the list.

Approach

1. Check if the list is empty or has only one node. If so, return `false`.
2. Initialize two pointers, `slow` and `fast`, both pointing to the head of the list.
3. Loop through the list:
    - Move `slow` by one step.
    - Move `fast` by two steps.
    - If `slow` and `fast` meet, a cycle exists, so return `true`.
4. If `fast` reaches the end of the list, return `false`.

Solution

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false; // Early return for empty or single-node lists
        }

        ListNode slow = head;
        ListNode fast = head;

        while (fast != null && fast.next != null) {
            slow = slow.next;          // Move slow pointer by one step
            fast = fast.next.next;     // Move fast pointer by two steps

            if (slow == fast) {
                return true;           // Cycle detected
            }
        }

        return false; // No cycle detected
    }
}
```

Description

The provided code defines a method `hasCycle` that determines whether a linked list has a cycle. It uses two pointers, `slow` and `fast`. As long as `fast` and its next node are not null, it keeps advancing `slow` by one and `fast` by two nodes. If `slow` and `fast` ever point to the same node, we have found a cycle. If `fast` reaches the end of the list (`null`), then there is no cycle.

Time and Space Complexity

- **Time Complexity:** O(n), where n is the number of nodes in the linked list. In the worst case, we traverse the entire list once.
- **Space Complexity:** O(1), because we only use two nodes, `slow` and `fast`, regardless of the size of the input list.

Dry Run on Test Cases

Test Case 1:

- Input: `head = [3,2,0,-4], pos = 1` (cycle formed between last and second node)
- Output: `true`
- Explanation: Fast and slow pointers will eventually meet at the node with value 2, indicating a cycle.

Test Case 2:

- Input: `head = [1,2], pos = 0` (cycle formed between first and second node)
- Output: `true`
- Explanation: Fast and slow pointers will meet at the node with value 1, indicating a cycle.

Test Case 3:

- Input: `head = [1], pos = -1` (no cycle)
- Output: `false`
- Explanation: There's only one node, and `fast` will become `null`, indicating no cycle.