Linked List-2

Lesson Plan

PW SKILLS

Today’s Checklist:

1. Delete Node in a Linked List (Leetcode-237)
2. Middle of Linked List (Leetcode-876)
3. Remove Nth Node from End of List (Leetcode-19)
4. Intersection of two Linked Lists (Leetcode-160)
5. Linked List Cycle (Leetcode-141)
6. Linked List Cycle-II (Leetcode-142)

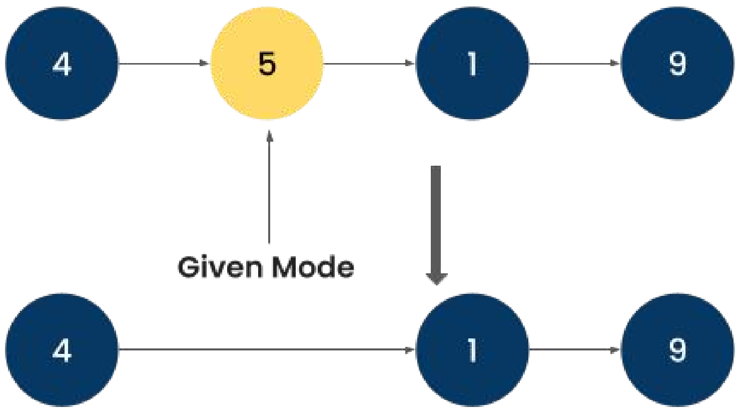
Delete Node in a Linked List (Leetcode-237)

There is a singly-linked list head and we want to delete a node node in it.
You are given the node to be deleted node. You will not be given access to the first node of head.
All the values of the linked list are unique, and it is guaranteed that the given node node is not the last node in the linked list.

Delete the given node. Note that by deleting the node, we do not mean removing it from memory. We mean:

- The value of the given node should not exist in the linked list.
- The number of nodes in the linked list should decrease by one.
- All the values before node should be in the same order.
- All the values after node should be in the same order.

Input: head = [4,5,1,9], node = 5
Output: [4,1,9]
Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.



Code:

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def deleteNode(self, node: ListNode) → None:
        if node is not None and node.next is not None:
            node.val = node.next.val
            node.next = node.next.next
```

Time complexity: O(m + n), where m and n are the lengths of the two linked lists. The pointers traverse the linked lists once, and the loop continues until either the intersection is found or both pointers reach the end.

Space complexity: O(1). The algorithm uses only a constant amount of extra space for the two pointers (a and b), regardless of the size of the linked lists.

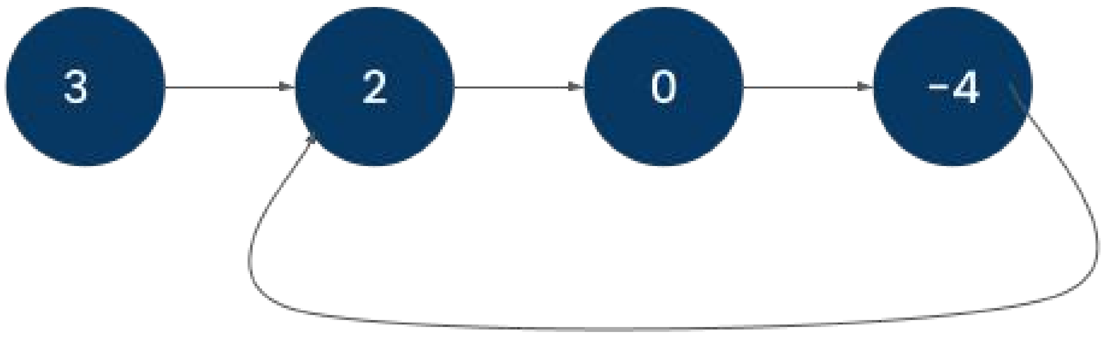
Linked List Cycle (Leetcode-141)

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.

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).



Code:

```
class Solution:
    def middleNode(self, head: ListNode) → ListNode:
        slow = head
        fast = head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        return slow
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm has at most two pointers traversing the linked list, and the loop will continue until the fast pointer reaches the end or the two pointers meet in a cycle.

Space complexity: O(1),The algorithm uses only a constant amount of extra space for the two pointers (slow_pointer and fast_pointer), regardless of the size of the linked list.

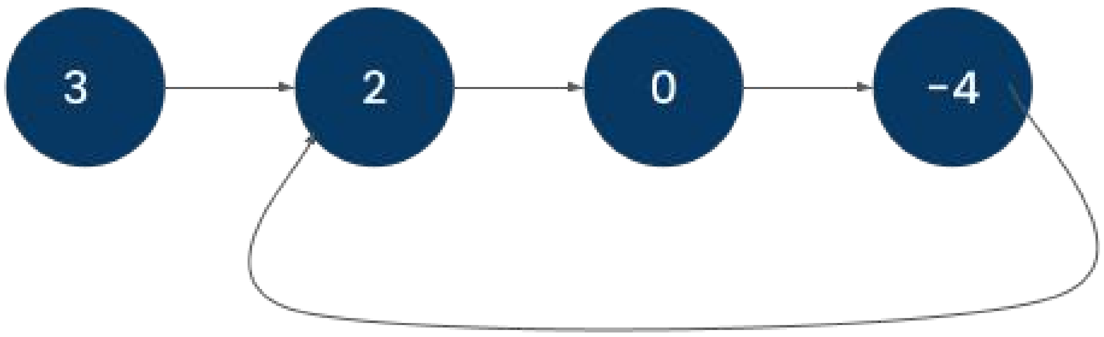
Linked List Cycle-II (Leetcode-142)

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.

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.



Code:

```
lass Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) → ListNode:
        fast = head
        slow = head

        for _ in range(n):
            fast = fast.next

        if fast is None:
            return head.next

        while fast.next is not None:
            fast = fast.next
            slow = slow.next

        slow.next = slow.next.next

        return head
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm uses two pointers, slow and fast, to traverse the linked list. In the worst case, it needs to iterate through the entire linked list once.

Space complexity: O(1),The algorithm uses only a constant amount of extra space for the two pointers (slow and fast), regardless of the size of the linked list. It doesn't use any additional data structures that scale with the input size.