PW SKILLS

Lesson Plan

Linked List-2

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)

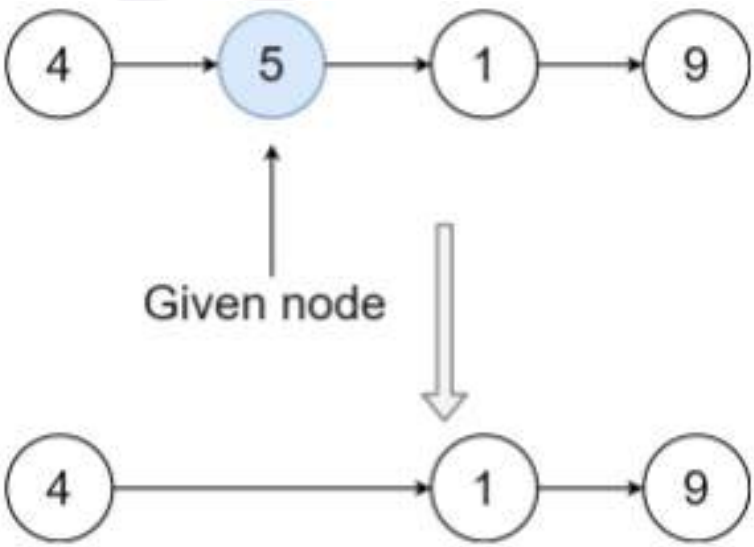
There is a singly-linked list head and we want to delete a node node in it.
You are given the node to be deleted node. You will not be given access to the first node of head.
All the values of the linked list are unique, and it is guaranteed that the given node node is not the last node in the linked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

```
public class Solution {
    public void deleteNode(ListNode node) {
        if (node ≠ null && node.next ≠ null) {
            node.val = node.next.val;
            node.next = node.next.next;
        }
    }
}
```

Time complexity: O(1) - Constant time complexity. The deletion operation involves updating the value of the given node with the value of its next node and then updating the next pointer to skip the next node. These operations are constant time.

Space complexity: O(1) - Constant space complexity. The algorithm uses only a constant amount of extra space, regardless of the size of the input linked list.

Middle of Linked List (Leetcode-876)

Given the head of a singly linked list, return the middle node of the linked list.
If there are two middle nodes, return the second middle node.

Input: head = [1,2,3,4,5]
Output: [3,4,5]

Explanation: The middle node of the list is node 3.

Code:

```
public class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode slow = head, fast = head;
        while (fast ≠ null && fast.next ≠ null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
}
```

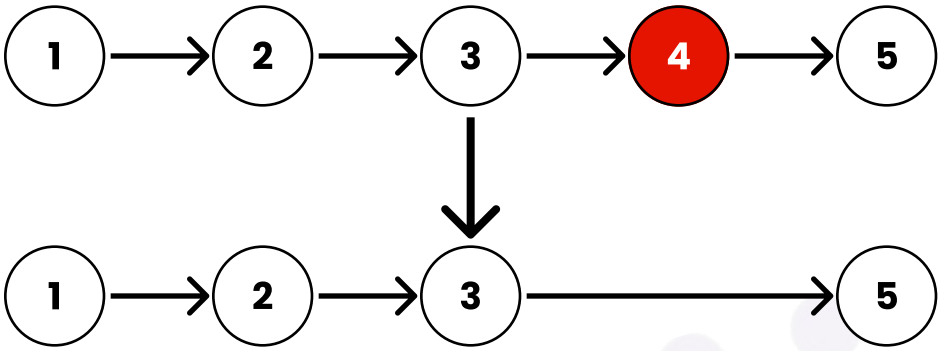
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list with two pointers (slow and fast), and in each iteration, the fast pointer moves two steps while the slow pointer moves one step.

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

Remove Nth Node from End of List (Leetcode-19)

Given the head of a linked list, remove the nth node from the end of the list and return its head.

Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]



Code:

```
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode fast = head, slow = head;

        // Move fast pointer ahead by n nodes
        for (int i = 0; i < n; i++) {
            fast = fast.next;
        }

        // If fast pointer becomes null, remove the head
        if (fast == null) {
            return head.next;
        }

        // Move both pointers until fast reaches the end
        while (fast.next != null) {
            fast = fast.next;
            slow = slow.next;
        }

        // Remove the nth node from the end
        slow.next = slow.next.next;

        return head;
    }
}
```

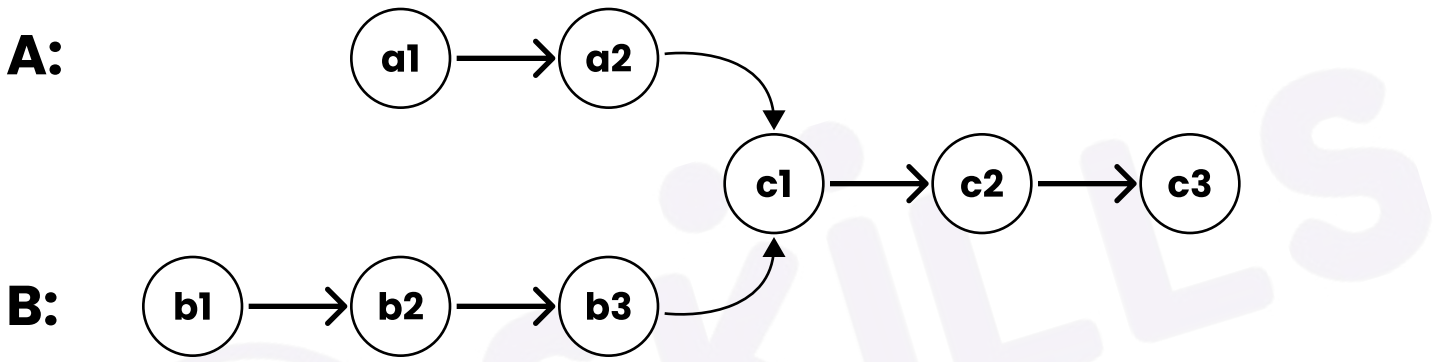
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm uses two pointers to traverse the linked list. The first loop advances the fast pointer by n nodes, and then the second loop advances both pointers until the fast pointer reaches the end.

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

Intersection of two Linked Lists (Leetcode-160)

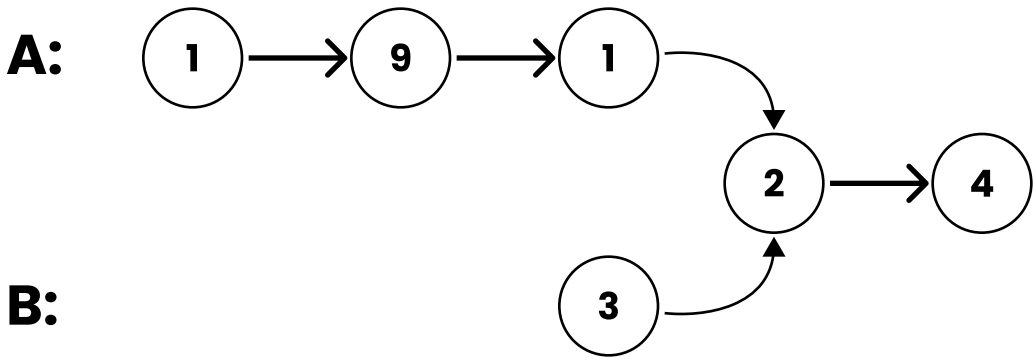
Given the heads of two singly linked-lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

For example, the following two linked lists begin to intersect at node c1:



Input: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
Output: Intersected at '2'

Explanation: The intersected node's value is 2 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are 3 nodes before the intersected node in A; There are 1 node before the intersected node in B.



Code:

```
public class Solution {
    public ListNode getIntersectionNode(ListNode head1, ListNode
head2) {
        if (head1 == null || head2 == null) {
            return null;
        }

        ListNode a = head1, b = head2;

        while (a != b) {
            a = (a == null) ? head2 : a.next;
            b = (b == null) ? head1 : b.next;
        }

        return a;
    }
}
```

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)

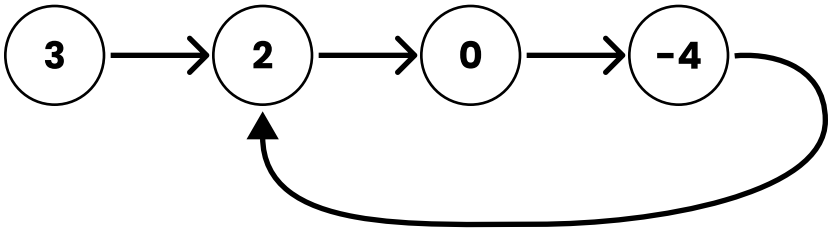
Given head, the head of a linked list, determine if the linked list has a cycle in it.

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:

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        ListNode slowPointer = head, fastPointer = head;
        while (fastPointer ≠ null && fastPointer.next ≠ null) {
            slowPointer = slowPointer.next;
            fastPointer = fastPointer.next.next;
            if (slowPointer == fastPointer) {
                return true;
            }
        }
        return false;
    }
}
```

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)

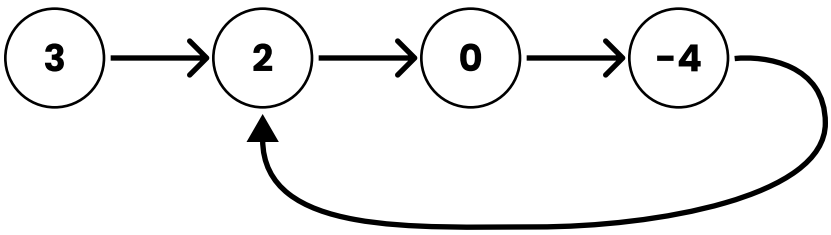
Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.

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:

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;

        while (fast ≠ null && fast.next ≠ null) {
            slow = slow.next;
            fast = fast.next.next;

            if (slow == fast) {
                slow = head;
                while (slow ≠ fast) {
                    slow = slow.next;
                    fast = fast.next;
                }
                return slow;
            }
        }

        return null;
    }
}
```

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.