ChatGPT - Remove Nth Node DSA

> Give notes for this DSA question. Give notes for both naive and optimal approach along with pseudo code. Given the head of a linked list, remove the nth node from the end of the list and return its head.

Here are detailed notes for solving the problem "Remove the Nth Node from the End of the List," including both the naive and optimal approaches, with explanations and pseudo code.

Problem Description

- You are given a singly linked list and an integer `n`.
- The task is to remove the `nth` node from the end of the linked list and return the updated head of the list.

Naive Approach

Idea

1. **Count the total number of nodes in the list**:
   - Traverse the list to determine its length (`len`).
2. **Identify the node to remove**:
   - The node to remove is at position `len - n + 1` from the start.
3. **Traverse again to remove the node**:
   - Traverse the list until the node before the target node and adjust the `next` pointers.

Time Complexity

- **O(2L)**: One traversal to count the nodes and another to locate and remove the target node.

Space Complexity

- **O(1)**: No additional space is required.

Pseudo Code

```plaintext
function removeNthFromEnd(head, n):
    # Step 1: Count the total number of nodes
    length = 0
    current = head
    while current is not null:
        length += 1
        current = current.next

    # Step 2: Find the target position from the start
    target = length - n + 1

    # Special case: If the head is to be removed
    if target == 1:
        return head.next

    # Step 3: Traverse to the node before the target
    current = head
    for i from 1 to target - 1:
        current = current.next

    # Step 4: Remove the target node
    current.next = current.next.next

    return head
```

Optimal Approach

Idea

1. **Use Two Pointers**:
    - Maintain two pointers, `first` and `second`.
    - Place `first` pointer `n` nodes ahead of `second`.

2. **Traverse the list with both pointers**:
    - Move both pointers one step at a time until `first` reaches the end.
    - At this point, `second` will be just before the node to remove.
3. **Remove the target node**:
    - Adjust the `next` pointer of `second` to skip the target node.

Advantages

- **Single traversal**: Traverses the list only once.

Time Complexity

- **O(L)**: Only one pass through the list is needed.

Space Complexity

- **O(1)**: No additional space is used.

Pseudo Code

plaintext

```
function removeNthFromEnd(head, n):
    # Step 1: Create a dummy node to handle edge cases
    dummy = new ListNode(0)
    dummy.next = head
    first = dummy
    second = dummy

    # Step 2: Move the `first` pointer `n+1` steps ahead
    for i from 0 to n:
        first = first.next

    # Step 3: Move both `first` and `second` until `first` is at the end
    while first is not null:
        first = first.next
        second = second.next

    # Step 4: Remove the target node
    second.next = second.next.next

    return dummy.next
```

Comparison

Approach	Time Complexity	Space Complexity	Traversals
Naive Approach	O(2L)	O(1)	2
Optimal Approach	O(L)	O(1)	1

Edge Cases

1. **Single-node list**: Removing the only node should return `null`.
2. **Remove the head**: If `n` equals the list length, the head node should be removed.
3. **Invalid `n`**: `n` greater than the length of the list should be handled gracefully.

Let me know if you'd like me to expand on any part!

Add java code for both Naive and optimal approach

Here's the Java implementation for both the naive and optimal approaches:

Naive Approach

Code

```java
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
```

```
}

public class RemoveNthNodeFromEnd {

    public ListNode removeNthFromEndNaive(ListNode head, int n) {
        // Step 1: Count the total number of nodes
        int length = 0;
        ListNode current = head;
        while (current != null) {
            length++;
            current = current.next;
        }

        // Step 2: Determine the target position from the start
        int target = length - n + 1;

        // Special case: If the head is to be removed
        if (target == 1) {
            return head.next;
        }

        // Step 3: Traverse to the node before the target
        current = head;
        for (int i = 1; i < target - 1; i++) {
            current = current.next;
        }

        // Step 4: Remove the target node
        current.next = current.next.next;

        return head;
    }
}
```

Optimal Approach

Code

java

```java
public class RemoveNthNodeFromEnd {

    public ListNode removeNthFromEndOptimal(ListNode head, int n) {
        // Step 1: Create a dummy node
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode first = dummy;
        ListNode second = dummy;

        // Step 2: Move the `first` pointer `n+1` steps ahead
        for (int i = 0; i <= n; i++) {
            first = first.next;
        }

        // Step 3: Move both pointers until `first` reaches the end
        while (first != null) {
            first = first.next;
            second = second.next;
        }

        // Step 4: Remove the target node
        second.next = second.next.next;

        return dummy.next;
    }
}
```

Usage

Example

java

```java
public class Main {
    public static void main(String[] args) {
        // Create a sample linked list: 1 -> 2 -> 3 -> 4 -> 5
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
```

```
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);

        RemoveNthNodeFromEnd solution = new RemoveNthNodeFromEnd();

        // Naive approach
        ListNode updatedHeadNaive = solution.removeNthFromEndNaive(head, 2);
        printList(updatedHeadNaive);

        // Reset the list for optimal approach testing
        head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);

        // Optimal approach
        ListNode updatedHeadOptimal = solution.removeNthFromEndOptimal(head, 2);
        printList(updatedHeadOptimal);
    }

    public static void printList(ListNode head) {
        ListNode current = head;
        while (current != null) {
            System.out.print(current.val + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}
```

Output for Input `1 -> 2 -> 3 -> 4 -> 5` and `n = 2`

```
rust

Naive Approach: 1 -> 2 -> 3 -> 5 -> null
Optimal Approach: 1 -> 2 ->
```

-