2095. Delete the Middle Node of a Linked List

Medium Linked List Two Pointers

Leetcode Link

Problem Description

You have a singly linked list. Your task is to remove the node that is in the middle of this list and return the head of the updated list. The middle node is defined as the $\lfloor n / 2 \rfloor^{th}$ node from the beginning of the list, where n is the total number of nodes in the list and $\lfloor x \rfloor$ signifies the greatest integer less than or equal to x. This means that you're not counting from 1, but from 0. So, in a linked list with:

- 1 node, the middle is the 0th node.
- 2 nodes, the middle is the 1st node.
- 3 nodes, the middle is the 1st node.
- 4 nodes, the middle is the 2nd node.
- 5 nodes, the middle is the 2nd node.

Your goal is to efficiently find and remove this middle node and ensure the integrity of the linked list is maintained after the removal.

Intuition

To solve this problem, the two-pointer technique is a perfect fit. The idea is to have two pointers, `slow` and `fast`. The `slow` pointer moves one step at a time, while the `fast` pointer moves two steps at a time. By the time the `fast` pointer reaches the end of the list, the `slow` pointer will be at the middle node.

Here's the step-by-step intuition:

1. Initialize a `dummy` node that points to the head. This dummy node will help us easily handle edge cases like when there's only one node in the list.
2. Start both `slow` and `fast` pointers. The `slow` pointer will start from the dummy node, while `fast` will start from the head node of the list.
3. Move `slow` one step and `fast` two steps until `fast` reaches the end of the list or has no next node (this is for cases when the number of nodes is even).
4. When the `fast` pointer reaches the end of the list, the `slow` pointer will be on the node just before the middle node (since it started from `dummy`, which is before the `head`).
5. Adjust the `next` pointer of the `slow` node so that it skips over the middle node, effectively removing it from the list.
6. Return the new head of the list, which is pointed to by `dummy.next`, since the `dummy` node was added before the original head.

By utilizing this approach, we can identify and remove the middle node in a singly linked list in a single pass and O(n) time complexity, where n is the number of nodes in the list.

Solution Approach

The solution utilizes a two-pointer approach, which is a common technique for problems involving linked lists or arrays where elements need to be compared or modified based on their positions. Here's a detailed walk-through:

1. **Initialization**: A dummy node is created and set to point to the head of the list. The dummy node, not present in the original list, serves as a starting point for the `slow` pointer, and helps in case the list has only one node, or if we need to delete the head of the list.

2. **Two-Pointers**: Two pointers are defined: `slow` starting at the dummy node and `fast` at the head node. This offset will allow `slow` to reach the node just before the middle node by the time `fast` reaches the end.

3. **Traversal**: The traversal begins with a `while` loop that continues until `fast` is neither null nor pointing to a node with a null next pointer. Inside the loop:

   - The `slow` pointer is moved one node forward with `slow = slow.next`.
   - The `fast` pointer is moved two nodes forward with `fast = fast.next.next`.

4. **Deletion**: After the loop, `slow` is at the node just before the middle node. To delete the middle node, `slow.next` is updated to point to `slow.next.next`. This effectively removes the middle node from the list by "skipping" it, as it is no longer referenced by the previous node.

5. **Return Updated List**: Finally, the head of the updated list is returned, which is `dummy.next`. This is because `dummy` is a pseudo-head pointing to the actual head of the list and its next pointer reflects the head of the updated list post-deletion.

The key data structure used is the singly-linked list, which is manipulated using pointer operations. This solution ensures that the middle node is deleted in a single pass, with a time complexity of O(n), where n is the number of nodes in the list. There is a constant space complexity of O(1), as the number of new variables used does not scale with the input size.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we have the following linked list:

```
1 [ A ] -> [ B ] -> [ C ] -> [ D ] -> [ E ]
```

We want to remove the $\lfloor 5 / 2 \rfloor^{th}$ = 2nd node from the beginning, which in this case is node `[ C ]`.

Following the solution approach:

Step 1: Initialization

- We create a dummy node `[ X ]` and point it to the head of the list `[ A ]`.
- The list now looks like this: `[ X ] -> [ A ] -> [ B ] -> [ C ] -> [ D ] -> [ E ]`.

Step 2: Two-Pointers

- We set the `slow` pointer to `[ X ]` (dummy node) and the `fast` pointer to `[ A ]` (head node).

Step 3: Traversal

- We start moving both pointers through the list with the loop condition in mind.

```
1 First iteration:
2   [ X ] -> [ A ] -> [ B ] -> [ C ] -> [ D ] -> [ E ]
3             ^        ^
4    slow               fast
5
6 Second iteration:
7   [ X ] -> [ A ] -> [ B ] -> [ C ] -> [ D ] -> [ E ]
8                      ^                 ^
9            slow                      fast
10
11 Third iteration (fast.next is null):
12   [ X ] -> [ A ] -> [ B ] -> [ C ] -> [ D ] -> [ E ]
13                     ^                          ^
14           slow                               fast (end of list)
```

Step 4: Deletion

- Now that `fast` has reached the end of the list, `slow` is just before the node we want to delete (`[ C ]`).
- We perform the deletion by updating the `next` pointer of the `slow` node to skip `[ C ]` and point to `[ slow.next.next ]`.

```
1 Before deletion:
2   [ X ] -> [ A ] -> [ B ] -> [ C ] -> [ D ] -> [ E ]
3                      ^
4            slow               slow.next (to be deleted)
5
6 After deletion:
7   [ X ] -> [ A ] -> [ B ] ------------> [ D ] -> [ E ]
8                      ^
9            slow                        slow.next
```

Step 5: Return Updated List

- We return the head of the updated list, which is `dummy.next`.
- The final updated list looks like this:

```
1 [ A ] -> [ B ] -> [ D ] -> [ E ]
```

Node `[ C ]` has been removed, and the integrity of the list is maintained.

Python Solution

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def deleteMiddle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Create a dummy node that points to the head of the list, to handle edge cases smoothly
        dummy_node = ListNode(next=head)

        # Initialize two pointers, slow will move one step at a time, fast will move two steps at a time
        slow_pointer, fast_pointer = dummy_node, head

        # Iterate through the list to find the middle
        while fast_pointer and fast_pointer.next:
            slow_pointer = slow_pointer.next  # Move slow pointer one step
            fast_pointer = fast_pointer.next.next  # Move fast pointer two steps

        # Now, slow_pointer is at the node just before the middle one. Delete the middle node
        slow_pointer.next = slow_pointer.next.next

        # Return the head of the updated list, by skipping over the dummy node
        return dummy_node.next
```

Java Solution

```java
// Definition of a singly-linked list node.
class ListNode {
    int value;
    ListNode next;

    ListNode() {}
    ListNode(int value) { this.value = value; }
    ListNode(int value, ListNode next) {
        this.value = value;
        this.next = next;
    }
}

class Solution {
    public ListNode deleteMiddle(ListNode head) {
        // Create a dummy node that acts as a predecessor of the head node.
        ListNode dummy = new ListNode(0, head);
        // Initialize two pointers, slow and fast. Slow moves 1 node at a time, while fast moves 2 nodes.
        ListNode slow = dummy, fast = head;

        // Iterate through the list with the fast pointer advancing twice as fast as the slow pointer
        // so that when the fast pointer reaches the end, the slow pointer will be at the middle.
        while (fast != null && fast.next != null) {
            slow = slow.next; // Move slow pointer one step.
            fast = fast.next.next; // Move fast pointer two steps.
        }

        // Skip the middle node. Slow pointer now points to the node before the middle node.
        slow.next = slow.next.next;

        // Return the modified list. The dummy's next points to the new list's head.
        return dummy.next;
    }
}
```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int value;
 *     ListNode *next;
 *     ListNode(int value = 0) : value(value), next(nullptr) {}
 *     ListNode(int value, ListNode *next) : value(value), next(next) {}
 * };
 */
class Solution {
public:
    // Function to delete the middle node of the linked list.
    ListNode* deleteMiddle(ListNode* head) {
        // Create a dummy node that points to the head of the list.
        ListNode* dummyNode = new ListNode(0, head);

        // Initialize slow and fast pointers for the runner technique.
        ListNode* slowPointer = dummyNode;
        ListNode* fastPointer = head;

        // Advance the fast pointer by two steps and the slow pointer by one step
        // until fast reaches the end of the list.
        while (fastPointer && fastPointer->next) {
            slowPointer = slowPointer->next;     // Move slow pointer by one
            fastPointer = fastPointer->next->next;  // Move fast pointer by two
        }

        // The slow pointer now points at the node before the middle node.
        ListNode* toDelete = slowPointer->next;  // Store the middle node to delete
        slowPointer->next = slowPointer->next->next;  // Remove the middle node

        delete toDelete; // Free memory of the node to be deleted

        // The head of the new modified list is the next node of dummyNode.
        ListNode* newHead = dummyNode->next;
        delete dummyNode; // Delete the dummyNode to prevent memory leak

        return newHead; // Return the new head of the list
    }
};
```

Typescript Solution

```typescript
/**
 * Deletes the middle node of a singly linked list.
 * If the list is empty or has only one node, returns null.
 * For even-sized lists, it deletes the second of the two middle nodes.
 * @param head - The head node of the singly linked list.
 * @returns The head node of the modified list with the middle node removed.
 */

function deleteMiddle(head: ListNode | null): ListNode | null {
    // If the list is empty or has only one node, it means there is no middle to delete.
    if (!head || !head.next) {
        return null;
    }

    let fast: ListNode | null = head.next; // This pointer will move at twice the speed of 'slow'.
    let slow: ListNode = head; // This pointer will move one step at a time.

    // Move through the list until 'fast' reaches the last or the second to last node.
    while (fast.next && fast.next.next) {
        slow = slow.next; // Move 'slow' one step.
        fast = fast.next.next; // Move 'fast' two steps.
    }

    // 'slow' is now behind the middle node, so set its 'next' to skip the middle node.
    slow.next = slow.next.next;

    // Return the modified list with the middle node removed.
    return head;
}
```

Time and Space Complexity

Time Complexity

The provided code implements an algorithm to delete the middle node of a singly-linked list. The `while` loop iterates through the list with two pointers: `slow` moves one step at a time, and `fast` moves two steps at a time. This loop will continue until `fast` (or its successor `fast.next`) reaches the end of the list. This means that we traverse the list only once, which leads to a time complexity of `O(N)`, where `N` is the number of nodes in the singly-linked list.

Space Complexity

The algorithm uses a few constant extra variables: `dummy`, `slow`, and `fast`. Regardless of the size of the list, the space used by these variables does not increase. Therefore, the space complexity of the algorithm is `O(1)`, indicating that it uses a constant amount of additional space beyond the input list.