876. Middle of the Linked List

Easy Linked List Two Pointers Leetcode Link

Problem Description

The task is to write a function that, given the head of a singly linked list, finds and returns the middle node of the list. A singly linked list is a collection of nodes where each node contains a value and a reference to the next node in the sequence. The key characteristic of this list is that you can only traverse it in one direction: from the head towards the end.

The challenge here is determining the middle node in a single pass, since the size of the list is not given in advance. It's also specified that if the linked list has an even number of nodes, we should return the second of the two middle nodes.

Intuition

To solve this problem efficiently, we make use of two pointers: `slow` and `fast`. Both pointers start at the head of the linked list. The `fast` pointer moves through the list at twice the speed of the `slow` pointer. This means that for every node the `slow` pointer travels, the `fast` pointer moves two nodes.

As a result of these different speeds, when the `fast` pointer reaches the end of the list, the `slow` pointer will be positioned at the middle. This happens because the `fast` pointer traverses two nodes for every single step taken by the `slow` pointer. If the total number of nodes is odd, the `fast` pointer will eventually point to `None`, which is the end of the list. If the number of nodes is even, the `fast` pointer will point to the last node. In both cases, the `slow` pointer will be at the middle, where if there are two middle nodes, because `fast` moves two steps at a time, `slow` will end up at the second middle node.

This approach allows us to find the middle node in a single pass through the list, which is more efficient than first counting the nodes and then traversing again to the middle—especially in the case of very large lists.

Solution Approach

The given solution makes use of two pointers, `slow` and `fast`, both initialized to the `head` of the list. This approach is commonly known as the "tortoise and hare" algorithm. Let's go through the steps of this approach:

1. Both `slow` and `fast` are initialized at the `head` of the linked list, thus they start at the same position.
2. We then enter a loop that continues until `fast` is no longer pointing to a valid node or `fast.next` is `None`. This condition ensures that we stop when `fast` (which moves faster) has reached the end of the list.
3. Inside the loop, `slow` is incremented to the next node (`slow.next`) and `fast` is incremented two nodes ahead (`fast.next.next`).
   - `slow = slow.next` moves the `slow` pointer one step forward.
   - `fast = fast.next.next` moves the `fast` pointer two steps forward.
4. When `fast` reaches the end of the list or there are no more nodes to traverse (`fast` becomes `None` or `fast.next` is `None`), the loop ends.
5. At this point, `slow` will be at the middle node of the list. If there are an odd number of nodes, it will be the exact middle. If there are an even number of nodes, it will be the second of the two middle nodes because of the way `fast` is moving two steps at a time.
6. The function then returns the `slow` pointer, which now points to the middle node of the linked list.

No additional data structure is used, making the space complexity O(1), as we only have two pointers regardless of the size of the linked list. Since each node is visited once by either `slow` or `fast`, the time complexity is O(n), where n is the number of nodes in the list.

The beauty of this solution lies in its simplicity and efficiency. It effectively halves the traversal time to find the middle node, which would otherwise take longer if we first counted the entire list to find the length, and then iterated again to the middle.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we have a singly linked list with 7 nodes, and their values when traversed from head to tail are `[1, 2, 3, 4, 5, 6, 7]`. We want to determine the middle node in a single pass.

We initialize two pointers at the start, `slow` points to the node with the value `1`, and `fast` also points to the node with value `1`.

As we loop through the list:

- In the first step, `slow` will move one node and point to `2`; `fast` will move two nodes and point to `3`.
- In the second step, `slow` will move to `3`; `fast` to `5`.
- In the third step, `slow` moves to `4`, `fast` moves to `7`.
- Now, `fast` is pointing at the last node, and `fast.next` is `None`. Thus, we reach the condition to end the loop.

At this point, `slow` is pointing to `4`, which is the middle node in this list of 7 nodes. Our function would then return the node with the value `4` as the middle node.

Now, let's consider an even-numbered list with 6 nodes, where the values are `[1, 2, 3, 4, 5, 6]`.

- On starting, both `slow` and `fast` point to `1`.
- In the first step, `slow` moves to `2`; `fast` jumps to `3`.
- In the second step, `slow` goes to `3`; `fast` skips to `5`.
- In the third step, `slow` proceeds to `4` and `fast` jumps to the last node, `6`.
- `fast.next` is now `None`. The loop ends.

`Slow` is at the value `4`, which is the second of the two middle nodes in this list of 6 nodes, meeting our requirement to return the second middle node in the case of an even-numbered list.

With this explanation, it should be clear how the "tortoise and hare" algorithm functions, allowing us to find the middle of the list in a single pass, hence delivering an efficient solution.

Python Solution

```python
# Class definition for a singly-linked list node.
class ListNode:
    def __init__(self, value=0, next_node=None):
        self.value = value
        self.next_node = next_node

class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        # Initialize two pointers, both starting at the head of the list.
        # 'slow' will move one step at a time, and 'fast' will move two steps at a time.
        slow_pointer = fast_pointer = head

        # Traverse the list. The loop runs until 'fast' reaches the end of the list.
        while fast_pointer and fast_pointer.next_node:
            # Move 'slow' one step forward.
            slow_pointer = slow_pointer.next_node
            # Move 'fast' two steps forward.
            fast_pointer = fast_pointer.next_node.next_node

        # When 'fast' reaches the end, 'slow' will be at the middle node.
        return slow_pointer

```

Java Solution

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

    // Constructor with no parameters
    ListNode() {}

    // Constructor with value parameter
    ListNode(int value) {
        this.value = value;
    }

    // Constructor with value and next node parameters
    ListNode(int value, ListNode next) {
        this.value = value;
        this.next = next;
    }
}

public class Solution {

    /**
     * Finds the middle node of a singly linked list.
     *
     * @param head The head of the linked list.
     * @return The middle node of the linked list.
     */
    public ListNode middleNode(ListNode head) {
        // Initialize two pointers, slow and fast.
        ListNode slowPointer = head;
        ListNode fastPointer = head;

        // Iterate through the list.
        // Fast pointer moves two steps at a time, slow pointer one step at a time.
        while (fastPointer != null && fastPointer.next != null) {
            slowPointer = slowPointer.next;       // Move slow pointer one step
            fastPointer = fastPointer.next.next;  // Move fast pointer two steps
        }

        // When fast pointer reaches the end of the list,
        // slow pointer will be at the middle element.
        return slowPointer;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a singly-linked list node.
 */
struct ListNode {
    int val;           // The value the node stores
    ListNode *next;    // Pointer to the next list node in the linked list

    // Constructor to initialize a node with no next node
    ListNode() : val(0), next(nullptr) {}

    // Constructor to initialize a node with a specific value and no next node
    ListNode(int x) : val(x), next(nullptr) {}

    // Constructor to initialize a node with a specific value and a next node
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

class Solution {
public:
    /**
     * Finds the middle node of a singly linked list.
     * If there are two middle nodes, the second middle node is returned.
     *
     * @param head The head of the linked list.
     * @return The middle node of the linked list
     */
    ListNode* middleNode(ListNode* head) {
        ListNode *slow = head; // 'slow' pointer moves one node at a time
        ListNode *fast = head; // 'fast' pointer moves two nodes at a time

        // Continue until 'fast' reaches the end of the list
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;         // Move 'slow' by one node
            fast = fast->next->next;   // Move 'fast' by two nodes
        }

        // When 'fast' reaches the end, 'slow' will be at the middle node
        return slow;
    }
};

```

Typescript Solution

```typescript
// Definition for singly-linked list node.
interface ListNode {
    val: number;
    next: ListNode | null;
}

/**
 * Finds the middle node of a singly linked list.
 * @param head The head of the singly linked list.
 * @return The middle node of the list.
 */
function middleNode(head: ListNode | null): ListNode | null {
    let fastPointer: ListNode | null = head; // Pointer that will move two steps at a time
    let slowPointer: ListNode | null = head; // Pointer that will move one step at a time

    // Loop until the fast pointer reaches the end of the list
    while (fastPointer !== null && fastPointer.next !== null) {
        fastPointer = fastPointer.next.next; // Move fast pointer two steps
        slowPointer = slowPointer.next;      // Move slow pointer one step
    }

    // When the fast pointer reaches the end of the list,
    // the slow pointer will be at the middle
    return slowPointer;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of nodes in the linked list. This is because the fast pointer advances two steps at a time and the slow pointer advances one step at a time. They start at the same point, so when the fast pointer reaches the end of the list, the slow pointer must be at the middle. Since the fast pointer traverses at most `n` nodes (where `n` is even) or `n-1` nodes (where `n` is odd), and it takes two iterations of the loop to move the fast pointer two nodes ahead, the loop executes approximately `n/2` iterations, which is linear with respect to the number of nodes.

The space complexity is `O(1)` irrespective of the number of nodes in the linked list because only two additional pointers (variables `slow` and `fast`) are used, which occupy constant space.