234. Palindrome Linked List

Easy · Stack · Recursion · Linked List · Two Pointers — Leetcode Link

Problem Description

The problem presents a scenario where you're given a singly linked list and asks you to determine if it is a palindrome. A palindrome is a sequence that reads the same forward and backward. For example, in the case of a linked list, `[1 -> 2 -> 2 -> 1]` is a palindrome, but `[1 -> 2 -> 3]` is not.

Intuition

To solve the problem of determining whether a linked list is a palindrome, we need an approach that allows us to compare elements from the start and end of the list efficiently. Since we can't access the elements of a linked list in a reverse order as easily as we can with an array, we have to be creative.

The solution involves several steps:

1. **Find the middle of the linked list**: We can find the middle of the linked list using the fast and slow pointer technique. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. When the fast pointer reaches the end of the list, the slow pointer will be in the middle.
2. **Reverse the second half of the linked list**: Starting from the middle of the list, reverse the order of the nodes. This will allow us to directly compare the nodes from the start and the end of the list without needing to store additional data or indices.
3. **Compare the first and the second half**: After reversal, we then compare the node values from the start of the list and the start of the reversed second half. If all corresponding nodes are equal, then the list is a palindrome.
4. **Restore the list (optional)**: If the problem required you to maintain the original structure of the list after checking for a palindrome, you would follow up by reversing the second half of the list again and reattaching it to the first half. However, this step is not implemented in the provided code, as it is not part of the problem's requirements.

The crux of this approach lies in the efficient O(n) traversal and no additional space complexity apart from a few pointers, which makes this method quite optimal.

Solution Approach

The solution approach follows the intuition which is broken down into the following algorithms and patterns:

1. **Two-pointer technique**: To find the middle of the list, we use two pointers (`slow` and `fast`). The `slow` pointer is incremented by one node, while the `fast` pointer is incremented by two nodes on each iteration. When the `fast` pointer reaches the end, `slow` will be pointing at the middle node.

```
slow, fast = head, head.next
while fast and fast.next:
    slow, fast = slow.next, fast.next.next
```

2. **Reversing the second half of the list**: Once we have the middle node, we reverse the second half of the list starting from `slow.next`. To do this, we initialize two pointers `pre` (to keep track of the previous node) and `cur` (the current node). We then iterate until `cur` is not `None`, each time setting `cur.next` to `pre`, effectively reversing the links between the nodes.

```
pre, cur = None, slow.next
while cur:
    t = cur.next
    cur.next = pre
    pre, cur = cur, t
```

3. **Comparison of two halves**: After reversing the second half, `pre` will point to the head of the reversed second half. We compare the values of the nodes starting from `head` and `pre`. If at any point the values differ, we return `False` indicating that the list is not a palindrome. Otherwise, we keep advancing both pointers until `pre` is `None`. If we successfully reach the end of both halves without mismatches, the list is a palindrome, so we return `True`.

```
while pre:
    if pre.val != head.val:
        return False
    pre, head = pre.next, head.next
return True
```

The code uses the two-pointer technique and the reversal of a linked list to solve the problem very effectively. The total time complexity of the algorithm is O(n), and the space complexity is O(1), because no additional space is used proportional to the input size; we're just manipulating the existing nodes in the linked list.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the linked list `[1 -> 2 -> 3 -> 2 -> 1]`. The goal is to determine if this list represents a palindrome.

Step 1: Finding the Middle

We use the two-pointer technique. Initially, both `slow` and `fast` point to the first element, with `fast` moving to the next immediately for comparison purposes.

```
Initial state:
slow -> 1
fast -> 2
List: 1 -> 2 -> 3 -> 2 -> 1
```

Now we begin traversal:

```
Iteration 1:
slow -> 2
fast -> 3
List: 1 -> 2 -> 3 -> 2 -> 1

Iteration 2:
slow -> 3
fast -> 1 (fast reaches the end of the list so we stop here)
List: 1 -> 2 -> 3 -> 2 -> 1
```

At this stage, `slow` is pointing to the middle of the list.

Step 2: Reverse the Second Half

Starting from the middle node (where `slow` is currently pointing), we proceed to reverse the second half of the list. We'll use two pointers `pre` and `cur` to achieve this:

```
pre points to None
cur points to 3 (slow.next)
List: 1 -> 2 -> 3 -> 2 -> 1
```

We now iterate and reverse the link between the nodes until `cur` is `None`:

```
Iteration 1:
pre -> 3
cur -> 2
Reversed part: None <- 3 List: 1 -> 2 -> 3 -> 2 -> 1

Iteration 2:
pre -> 2
cur -> 1
Reversed part: None <- 3 <- 2 List: 1 -> 2 -> 3 -> 2 -> 1

Iteration 3:
pre -> 1
cur -> None
Reversed part: None <- 3 <- 2 <- 1
```

After reversing, we have `pre` pointing to the new head of the reversed second half, which is the node with the value `1`.

Step 3: Compare Two Halves

We now have two pointers, `head` pointing to the first node of the list and `pre` pointing to the head of the reversed second half. We need to compare the values of both halves:

```
pre points to 1, head points to 1
```

We move both pointers and compare their values:

```
pre -> 2, head -> 2 (values match, move forward)
pre -> 3, head -> 3 (values match, move forward)
```

When `pre` becomes `None`, we've successfully compared all nodes of the reversed half with the corresponding nodes of the first half and found that all the values match, which implies that the list represents a palindrome. Hence, we return `True`.

When implementing these steps in a programming language like Python, the overall result of this example would be that the function confirms the linked list `[1 -> 2 -> 3 -> 2 -> 1]` is indeed a palindrome.

Python Solution

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

class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        # Initialize two pointers, slow moves one step at a time, fast moves two steps
        slow = head
        fast = head.next

        # Move fast pointer to the end of the list, and slow to the middle
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # Reverse the second half of the list
        prev = None
        current = slow.next
        while current:
            temp = current.next
            current.next = prev
            prev, current = current, temp

        # Compare the first half and the reversed second half
        while prev:
            if prev.val != head.val:
                return False
            prev, head = prev.next, head.next

        # If all nodes matched, it's a palindrome
        return True

```

Java Solution

```java
/**
 * Definition for singly-linked list.
 */
class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

class Solution {
    public boolean isPalindrome(ListNode head) {
        // Use two pointers: slow moves one step at a time and fast moves two steps at a time.
        ListNode slow = head;
        ListNode fast = head;

        // Move fast pointer to the end, and slow to the middle of the list
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        // Reverse the second half of the list
        ListNode prev = null;
        ListNode current = slow;
        while (current != null) {
            ListNode temp = current.next; // Stores the next node
            current.next = prev; // Reverses the link
            prev = current; // Moves prev to current node
            current = temp; // Move to the next node in the original list
        }

        // Compare the reversed second half with the first half
        ListNode firstHalfIterator = head;
        ListNode secondHalfIterator = prev;
        while (secondHalfIterator != null) {
            // If values are different, then it's not a palindrome
            if (secondHalfIterator.val != firstHalfIterator.val) {
                return false;
            }

            // Move to the next nodes in both halves
            secondHalfIterator = secondHalfIterator.next;
            firstHalfIterator = firstHalfIterator.next;
        }

        // All values matched, so it's a palindrome
        return true;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        // Use two pointers to find the middle of the list
        ListNode* slowPtr = head;
        ListNode* fastPtr = head->next;

        // Move fastPtr by two and slowPtr by one step
        // When fastPtr reaches the end, slowPtr will be at the middle
        while (fastPtr && fastPtr->next) {
            slowPtr = slowPtr->next;
            fastPtr = fastPtr->next->next;
        }

        // Reverse the second half of the list
        ListNode* prevNode = nullptr;
        ListNode* currentNode = slowPtr->next;
        while (currentNode) {
            ListNode* nextTemp = currentNode->next;
            currentNode->next = prevNode;
            prevNode = currentNode;
            currentNode = nextTemp;
        }

        // Compare the reversed second half with the first half
        while (prevNode) {
            // If the values are different, then it's not a palindrome
            if (prevNode->val != head->val) {
                return false;
            }
            // Move to the next nodes in both halves
            prevNode = prevNode->next;
            head = head->next;
        }

        // All values matched, so it is a palindrome
        return true;
    }
};

```

Typescript Solution

```typescript
/**
 * Function to determine if a given singly linked list is a palindrome.
 * @param {ListNode | null} head - The head of the singly linked list.
 * @returns {boolean} - True if the list is a palindrome, false otherwise.
 */
function isPalindrome(head: ListNode | null): boolean {
    // Two pointers: slow moves one step at a time, fast moves two steps.
    let slowPointer: ListNode | null = head;
    let fastPointer: ListNode | null = head?.next;

    // Traverse the list to find the middle
    while (fastPointer !== null && fastPointer.next !== null) {
        slowPointer = slowPointer.next;
        fastPointer = fastPointer.next.next;
    }

    // Reverse the second half of the list
    let current: ListNode = slowPointer.next;
    slowPointer.next = null;
    let previous: ListNode = null;
    while (current !== null) {
        let temp: ListNode = current.next;
        current.next = previous;
        previous = current;
        current = temp;
    }

    // Compare the two halves of the list
    while (previous !== null && head !== null) {
        if (previous.val !== head.val) {
            return false; // Values do not match, not a palindrome
        }
        previous = previous.next;
        head = head.next;
    }

    // If all values matched, then the list is a palindrome
    return true;
}

```

Time and Space Complexity

The code above checks if a given singly-linked list is a palindrome. Here is the analysis of its time and space complexity:

Time Complexity

The algorithm uses two pointers (`slow` and `fast`) to find the middle of the linked list. The `fast` pointer moves two steps for every step the `slow` pointer takes. This loop will run in `O(n/2)` time, which is `O(n)` where `n` is the number of nodes in the list.

After finding the middle of the list, the code reverses the second half of the linked list. This is another loop that runs from the middle to the end of the list, which is also `O(n/2)` or simplifies to `O(n)`.

Finally, the code compares the values of nodes from the start of the list and the start of the reversed second half. This comparison stops when the end of the reversed half is reached, which is at most `n/2` steps, so `O(n/2)` or `O(n)`.

The total time complexity is the sum of these steps, which are all linear with respect to the length of the linked list: `O(n) + O(n) + O(n)` which is `O(3n)` or simply `O(n)`.

Space Complexity

There are no additional data structures used that grow with the input size. The pointers and temporary variables use a constant amount of space regardless of the size of the linked list. Therefore, the space complexity is `O(1)`, which means it is constant.

So, the overall time complexity of the algorithm is `O(n)`, and the space complexity is `O(1)`.