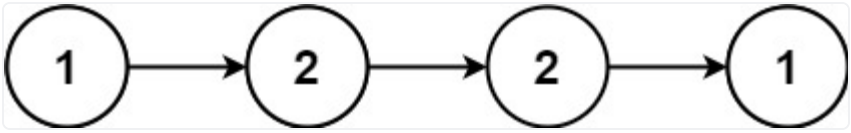
Palindrome Linked List(Day 25)

Prepared By Azan Imtiaz

Palindrome Linked List(Leatcode)

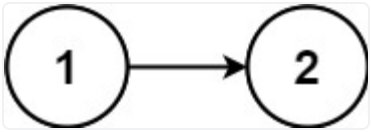
Given the `head` of a singly linked list, return `true` *if it is a palindrome* or `false` *otherwise*.

Example 1:




```
Input: head = [1,2,2,1]
Output: true
```

Example 2:




```
Input: head = [1,2]
Output: false
```

Constraints:

- The number of nodes in the list is in the range `[1, 10^5]`.
- `0 <= Node.val <= 9`

Follow up: Could you do it in `O(n)` time and `O(1)` space?

Problem Description:

The problem is to determine if a singly linked list is a **palindrome**. A palindrome is a sequence of elements that reads the same forward and backward. In the case of a singly linked list, we need to

verify if the sequence of values in the nodes is the same when traversed from the beginning and from the end.

Intuition:

To solve the problem, the main challenge is that we cannot easily traverse the list backwards due to the nature of a singly linked list. However, we can make use of a **stack** to simulate reverse traversal. A stack works in a Last In, First Out (LIFO) manner, meaning that when we pop elements off, we get them in reverse order compared to how we pushed them on.

The basic idea is:

1. Traverse the entire linked list and push all node values onto a stack.
2. Traverse the list again and, for each node, compare its value with the value popped from the stack (which gives us the reverse order).
3. If all values match, the list is a palindrome; otherwise, it is not.

Approach:

1. **Stack to Reverse Order**:
   - Traverse through the list once and push each element's value onto the stack.
   - This stores the values in reverse order, as the last element pushed will be the first one popped.
2. **Compare with Original List**:
   - Traverse the list again, and for each node, pop a value from the stack and compare it with the node's value.
   - If all values match, then the linked list is a palindrome; otherwise, it is not.

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        Stack<Integer> stack = new Stack<>();
        ListNode temp = head;

        // Step 1: Push all elements of the linked list to the stack
        while (temp != null) {
            stack.push(temp.val);
            temp = temp.next;
        }

        // Step 2: Traverse the list again and compare with stack
        temp = head;
        while (temp != null) {
            if (temp.val != stack.pop()) {
                return false; // Mismatch found, not a palindrome
            }
```

```
            temp = temp.next;
        }

        // If all elements matched, it's a palindrome
        return true;
    }
}
```

Code Description:

1. **ListNode Class**: Defines a node in a singly linked list, with an integer value and a reference to the next node.
2. **isPalindrome Function**:
   - First, it creates a stack to store the list values.
   - In the first `while` loop, it pushes all the values of the linked list onto the stack.
   - In the second `while` loop, it compares the values of the linked list with the values popped from the stack (i.e., in reverse order). If any value differs, it returns `false`.
   - If no mismatches are found, it returns `true`, indicating the list is a palindrome.

Time and Space Complexity:

- **Time Complexity**:
  - Traversing the linked list takes **O(n)** where `n` is the number of nodes. We traverse the list twice, so the total time complexity is **O(n)**.
- **Space Complexity**:
  - The stack stores all the elements of the list, so it requires **O(n)** space.

Dry Run:

1. **Test Case 1**: `[1, 2, 3, 2, 1]`
   - Step 1: Push all elements onto the stack: `stack = [1, 2, 3, 2, 1]`.
   - Step 2: Traverse the list again and compare with popped values:
     - Compare 1 with stack.pop() -> 1 (match)
     - Compare 2 with stack.pop() -> 2 (match)
     - Compare 3 with stack.pop() -> 3 (match)
     - Compare 2 with stack.pop() -> 2 (match)
     - Compare 1 with stack.pop() -> 1 (match)

- Since all values match, the result is **true**.

2. **Test Case 2**: `[1, 2, 2, 1]`
   - Step 1: Push all elements onto the stack: `stack = [1, 2, 2, 1]`.
   - Step 2: Traverse the list again and compare with popped values:
     - Compare 1 with stack.pop() -> 1 (match)
     - Compare 2 with stack.pop() -> 2 (match)
     - Compare 2 with stack.pop() -> 2 (match)
     - Compare 1 with stack.pop() -> 1 (match)
   - Since all values match, the result is **true**.
3. **Test Case 3**: `[1, 2, 3]`
   - Step 1: Push all elements onto the stack: `stack = [1, 2, 3]`.
   - Step 2: Traverse the list again and compare with popped values:
     - Compare 1 with stack.pop() -> 3 (mismatch)
   - As there is a mismatch, the result is **false**.

Could you do it in `O(n)` time and `O(1)` space?

we can optimize the solution both in terms of **space** and **time complexity**. The current solution uses **O(n)** space due to the stack. We can improve this by eliminating the need for extra space. The optimized approach involves finding the middle of the linked list, reversing the second half of the list, and then comparing it with the first half.

Optimized Approach:

1. **Find the middle of the linked list**:
   - Use the **slow and fast pointer technique** to find the middle of the list. The slow pointer moves one step at a time, while the fast pointer moves two steps. When the fast pointer reaches the end, the slow pointer will be at the middle.
2. **Reverse the second half of the list**:
   - Once we find the middle, reverse the second half of the list in-place.
3. **Compare the two halves**:

- Now compare the first half with the reversed second half. If all elements match, the list is a palindrome.

4. **Restore the list** (optional):
   - If required, you can restore the original list by reversing the second half again.

Steps:

1. Find the middle of the list using the slow and fast pointer technique.
2. Reverse the second half of the list.
3. Compare the first and second halves.
4. Restore the list if needed.

Optimized Code:

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        if (head == null || head.next == null) {
            return true; // An empty list or a single node list is a palindrome.
        }

        // Step 1: Find the middle of the linked list using slow and fast
pointers.ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        // Step 2: Reverse the second half of the list.ListNode secondHalfStart =
reverseList(slow);

        // Step 3: Compare the first half and the reversed second half.ListNode
firstHalfStart = head;
        ListNode secondHalf = secondHalfStart;
        while (secondHalf != null) {
            if (firstHalfStart.val != secondHalf.val) {
                return false; // Mismatch found, not a palindrome.
            }
            firstHalfStart = firstHalfStart.next;
            secondHalf = secondHalf.next;
        }

        // Step 4: (Optional) Restore the list by reversing the second half again.
        reverseList(secondHalfStart); // Restore the list if needed.return true; // If
all elements matched, it's a palindrome.
    }
```

```
    // Function to reverse a linked list.private ListNode reverseList(ListNode head) {
        ListNode prev = null;
        while (head != null) {
            ListNode next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }
}
```

Code Explanation:

1. **Finding the middle of the list**:
   - We use two pointers, `slow` and `fast`. The `fast` pointer moves twice as fast as the `slow` pointer. When `fast` reaches the end, `slow` will be at the middle.
2. **Reversing the second half of the list**:
   - Once the middle is found, the second half of the list starting from the middle is reversed using the `reverseList` function.
3. **Comparing the two halves**:
   - The first half is compared with the reversed second half. If all elements match, the list is a palindrome.
4. **Restoring the list** (optional):
   - After comparing, we restore the original order of the list by reversing the second half again.

Time and Space Complexity:

- **Time Complexity**:
  - Finding the middle takes **O(n)**.
  - Reversing the second half takes **O(n)**.
  - Comparing the two halves takes **O(n)**.
  - Thus, the total time complexity is **O(n)**, where `n` is the number of nodes.
- **Space Complexity**:
  - The optimized solution only uses **O(1)** extra space (for pointers) because we reverse the list in place, avoiding the use of a stack.

    The overall space complexity is **O(1)**.

Dry Run on Test Cases:

1. **Test Case 1**: `[1, 2, 3, 2, 1]`
   - **Finding the middle**: Slow pointer stops at 3.
   - **Reversing second half**: List becomes `[1, 2, 3, 1, 2]` (second half is reversed).
   - **Comparison**: Compare `1` with `1`, `2` with `2`, `3` with `3`. All match.
   - **Result**: The list is a palindrome, so the result is **true**.
2. **Test Case 2**: `[1, 2, 2, 1]`
   - **Finding the middle**: Slow pointer stops at the second `2`.
   - **Reversing second half**: List becomes `[1, 2, 1, 2]`.
   - **Comparison**: Compare `1` with `1`, `2` with `2`. All match.
   - **Result**: The list is a palindrome, so the result is **true**.
3. **Test Case 3**: `[1, 2, 3]`
   - **Finding the middle**: Slow pointer stops at `3`.
   - **Reversing second half**: List becomes `[1, 2, 3]` (second half is just one element).
   - **Comparison**: Compare `1` with `3`. Mismatch found.
   - **Result**: The list is not a palindrome, so the result is **false**.