Solution Review: Problem Challenge 1

We'll cover the following

Palindrome LinkedList (medium)

Given the head of a **Singly LinkedList**, write a method to check if the **LinkedList is a palindrome** or not.

Your algorithm should use **constant space** and the input LinkedList should be in the original form once the algorithm is finished. The algorithm should have $O(N)$ time complexity where 'N' is the number of nodes in the LinkedList.

Example 1:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> null
Output: true
```

Example 2:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> 2 -> null
Output: false
```

Solution

As we know, a palindrome LinkedList will have nodes values that read the same backward or forward. This means that if we divide the LinkedList into two halves, the node values of the first half in the forward direction should be similar to the node values of the second half in the backward direction. As we have been given a Singly LinkedList, we can't move in the backward direction. To handle this, we will perform the following steps:

1. We can use the **Fast & Slow pointers** method similar to Middle of the LinkedList to find the middle node of the LinkedList.
2. Once we have the middle of the LinkedList, we will reverse the second half.
3. Then, we will compare the first half with the reversed second half to see if the LinkedList represents a palindrome.
4. Finally, we will reverse the second half of the LinkedList again to revert and bring the LinkedList back to its original form.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java

class ListNode {
  int value = 0;
  ListNode next;

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

class PalindromicLinkedList {

  public static boolean isPalindrome(ListNode head) {
    if (head == null || head.next == null)
      return true;

    // find middle of the LinkedList
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next.next;
    }

    ListNode headSecondHalf = reverse(slow); // reverse the second half
    ListNode copyHeadSecondHalf = headSecondHalf; // store the head of reversed part to revert back later

    // compare the first and the second half
```

Run Save Reset

Time complexity

The above algorithm will have a time complexity of $O(N)$ where 'N' is the number of nodes in the LinkedList.

Space complexity

The algorithm runs in constant space $O(1)$.

← Back
Problem Challenge 1

Next →
Problem Challenge 2

✓ Completed

Report an Issue Ask a Question