Solution: Palindrome Linked List

Let's solve the Palindrome Linked List problem using the Fast and Slow Pointers pattern.

We'll cover the following

Statement

Given the head of a linked list, your task is to check whether the linked list is a palindrome or not. Return TRUE if the linked list is a palindrome; otherwise, return FALSE.

Constraints:

Let `n` be the number of nodes in a linked list.

- $1 \leq$ `n` ≤ 500
- $0 \leq$ `Node.value` ≤ 9.

Solution

The fast and slow pointers technique helps determine whether a linked list is a palindrome or not, because it allows us to efficiently traverse the list and find the middle node in a single pass. We can do this in linear time and with constant extra space.

To determine whether a linked list is a palindrome, we first find the middle node of the linked list using the fast and slow pointers approach. Then, we will reverse the second half of the linked list, starting from the node after the middle node until the end of the list. Next, we will compare the first half with the second half. If both halves of the list match, the linked list is a palindrome. Otherwise, it is not.

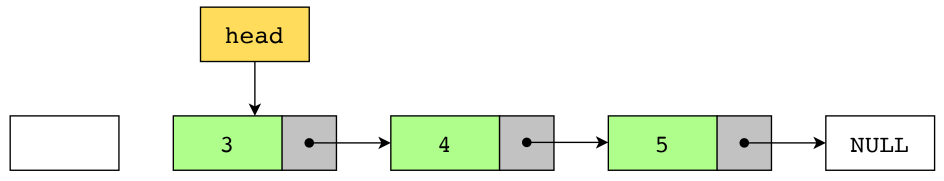
The algorithm to solve this problem is as follows:

1. First, we will find the middle node of the linked list. To do this, we'll traverse the linked list using two pointers, where the slow pointer will move one step forward, and the fast pointer will move two steps forward. We'll do this until the fast pointer reaches the end of the list or a null node. At this point, the slow pointer will be pointing at the middle node of the list.
2. Next, we'll reverse the second half of the linked list, starting from the node after the middle node. To reverse the list, we will follow these steps:

- Initialize three pointers: `prev`, `next`, and `curr`. The `prev` and `next` pointers are initialized as NULL, while `curr` is initialized to the head of the linked list.
- Iterate over the linked list. While iterating, perform the following steps:
  - Before changing the next of `curr`, store the next node using the following line of code: `next = curr.next`.

- Next, we'll assign the `next` pointer of `curr` to `prev` using the following line of code `curr.next = prev`. The effect of this line of code is that it will reverse the pointer from forward to backward to reverse the linked list.
- After reversing the pointer, we will update `prev` as `curr` and `curr` as `next`, using the following lines of code respectively: `prev = curr` and `curr = next`.

Let's look at the following illustration to get a better understanding of reversing the linked list:

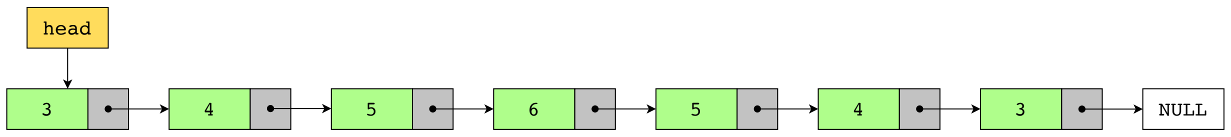


Given the above linked list, we will reverse it in place.

1 of 12

3. After finding the mid of the linked list and reversing its second half, the last step is to compare every element of the first half of the linked list with the corresponding element in the second half of the reversed linked list. If both halves are the same, the list is a palindrome, and we'll return TRUE. Otherwise, we'll return FALSE.

Let's look at the following illustration to get a better understanding of the solution:



Given the above linked list, we will first find its middle element.

1 of 11

Let's implement the algorithm as discussed above:

Java

main.java

LinkedListReversal.java

LinkedListNode.java

LinkedList.java

PrintList.java

```
import java.util.*;

        LinkedListNode slow = head;
        LinkedListNode fast = head;

        // Find the middle of the linked list using the slow and fast pointer
        while (fast != null && fast.next != null) {
            // move slow one step forward
            slow = slow.next;
            // move fast two steps forward
            fast = fast.next.next;
        }
        // Reverse the second half of the linked list starting from the middl
        LinkedListNode revertData = LinkedListReversal.reverseLinkedList(slow
        // Compare the first half of the linked list with the reversed seconc
        boolean check = compareTwoHalves(head, revertData);
        // Re-reverse the second half of the linked list to restore the origi
        revertData = LinkedListReversal.reverseLinkedList(revertData);
        // Return True if the linked list is a palindrome, else False
        if (check) {
            return true;
        }

        return false;
```

Palindrome Linked List

Time complexity

The algorithm’s time complexity is $O(n)$, where n is the total number of nodes in the linked list.

Space complexity

The space complexity of the algorithm above is $O(1)$, because it does not use any extra space in memory.

← Back

Palindrome Linked List

Next →

Sliding Window: Intro...

☑ Mark as Completed