Check if given Linked List is Plaindrome

Check if given Linked List is Plaindrome

Problem Statement: Given the head of a singly linked list, return true if it is a palindrome.

Examples:

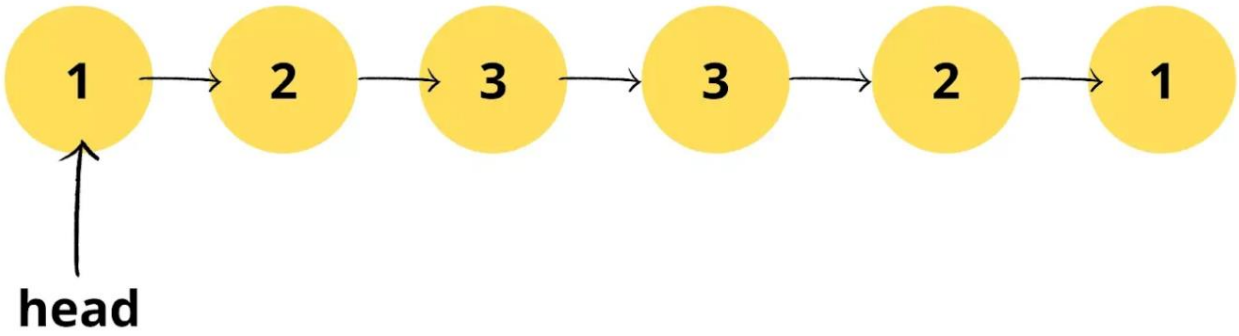
Example 1:

Input: head = [1,2,3,3,2,1]

Output:

true

Explanation: If we read elements from left to right, we get [1,2,3,3,2,1]. When we read elements from right to left, we get [1,2,3,3,2,1]. Both ways list remains same and hence, the given linked list is palindrome.



Example 2:

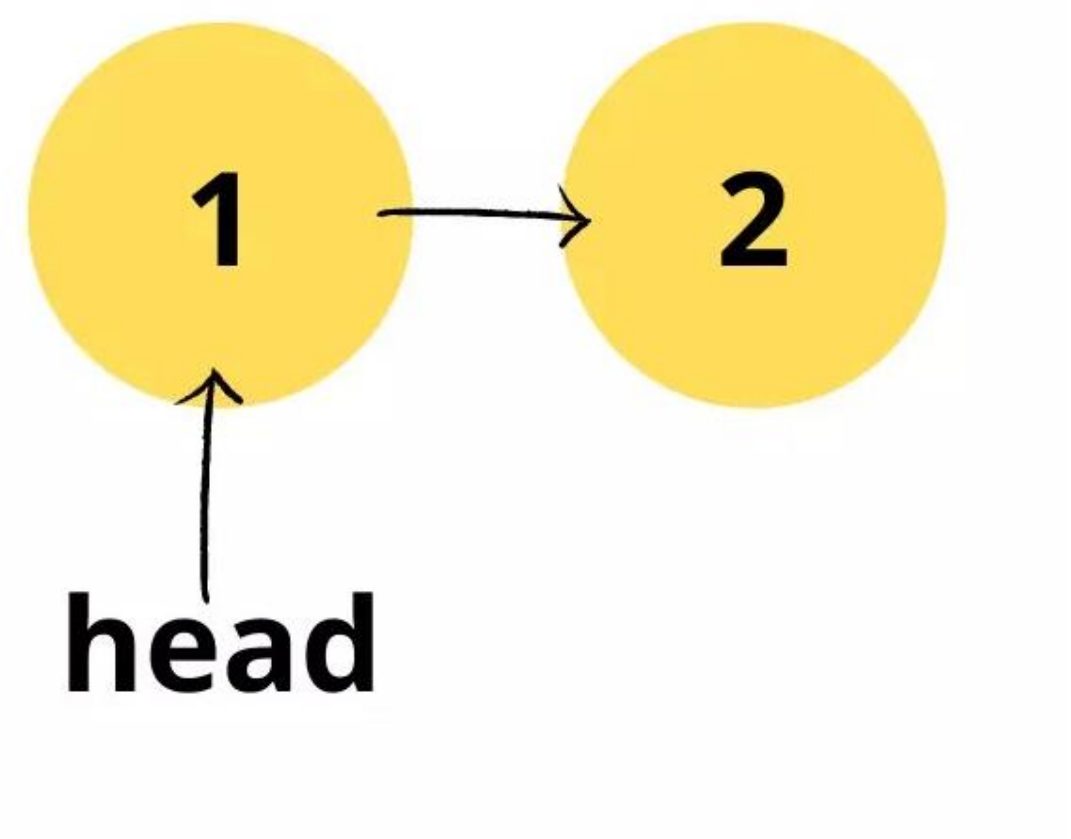
Input:

head = [1,2]

Output:

false

Explanation: When we read elements from left to right, we get [1,2]. Reading from right to left, we get a list as [2,1]. Both are different and hence, the given linked list is not palindrome.



Solution: Using the extra data structure

Approach:

We can store elements in an array. Then check if the given array is a palindrome. How to check if an array is a palindrome?

Let's take a string, say "level" which is a palindrome. Let's observe a thing.

Letter	Position	Letter	Position
l	0	l	4
e	1	e	3
v	2	v	2

So we can see that each index letter is the same as (length-each index -1) letter.

The same logic required to check an array is a palindrome.

Following are the steps to this approach.

- Iterate through the given list to store it in an array.
- Iterate through the array.
- For each index in range of n/2 where n is the size of the array
- Check if the number in it is the same as the number in the n-index-1 of the array.

Code:

```
static boolean isPalindrome(Node head) {
    ArrayList<Integer> arr=new ArrayList<>();
    while(head != null) {
        arr.add(head.num);
        head = head.next;
    }
    for(int i=0;i<arr.size()/2;i++)
        if(arr.get(i) != arr.get(arr.size()-i-1)) return false;
    return true;
}
```

Time Complexity: O(N)

Reason: Iterating through the list to store elements in the array.

Space Complexity: O(N)

Reason: Using an array to store list elements for further computations.

Solution 2: Optimized Solution

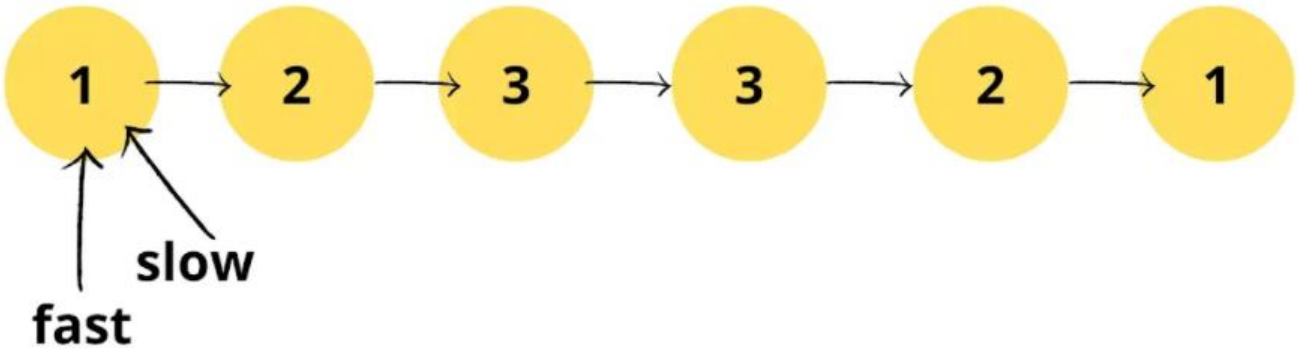
Approach:

Following are the steps to this approach:-

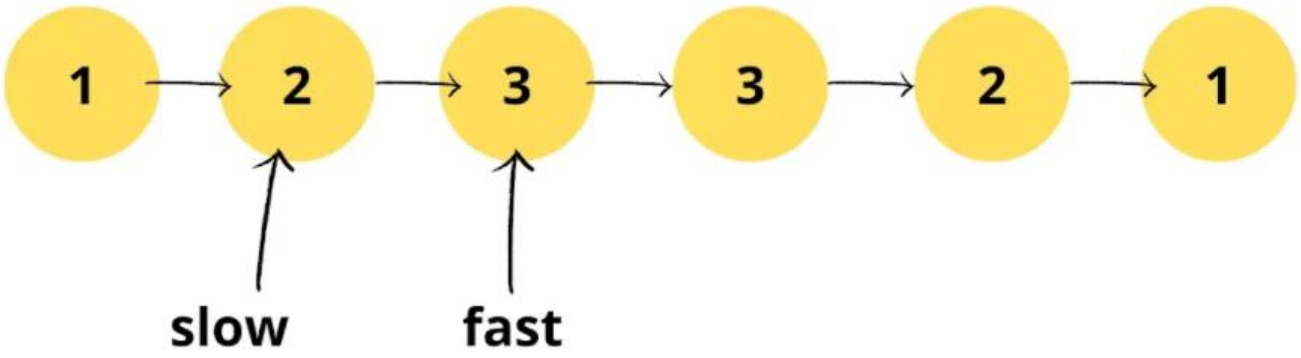
- Find the middle element of the linked list

- Reverse linked list from next element of middle element
- Iterate through the new list until the middle element reaches the end of the list.
- Use a dummy node to check if the same element exists in the linked list from the middle element.

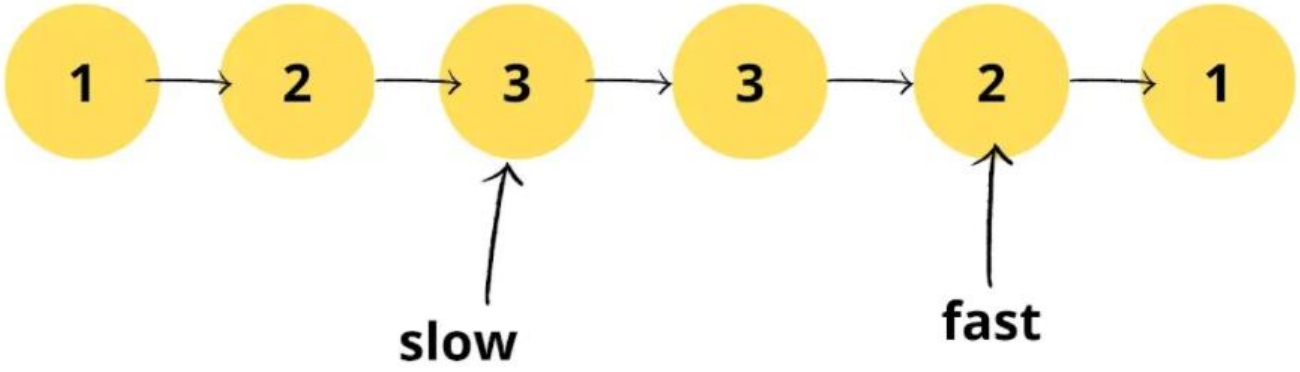
Dry Run:



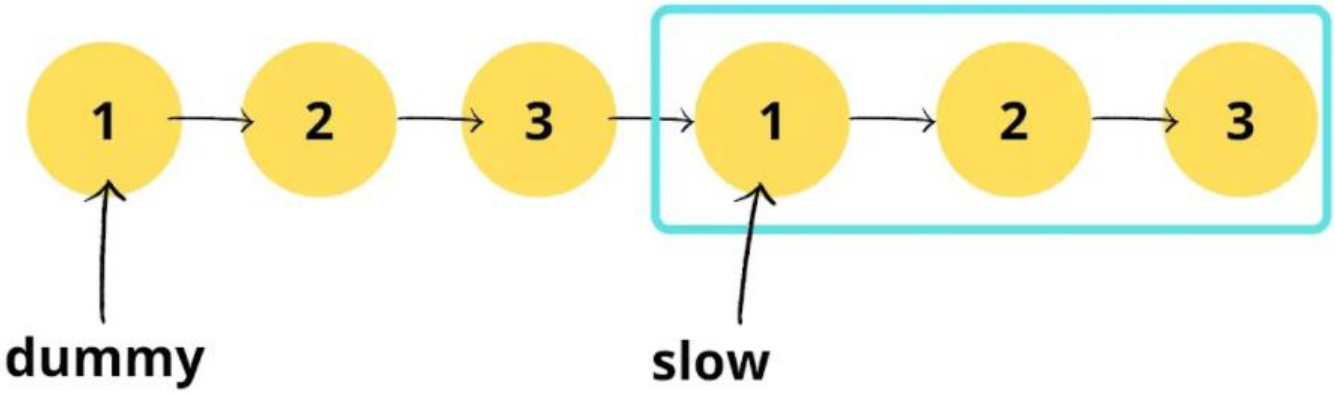
we will find the middle element of the list.
Move fast until it do not reach last or second last element



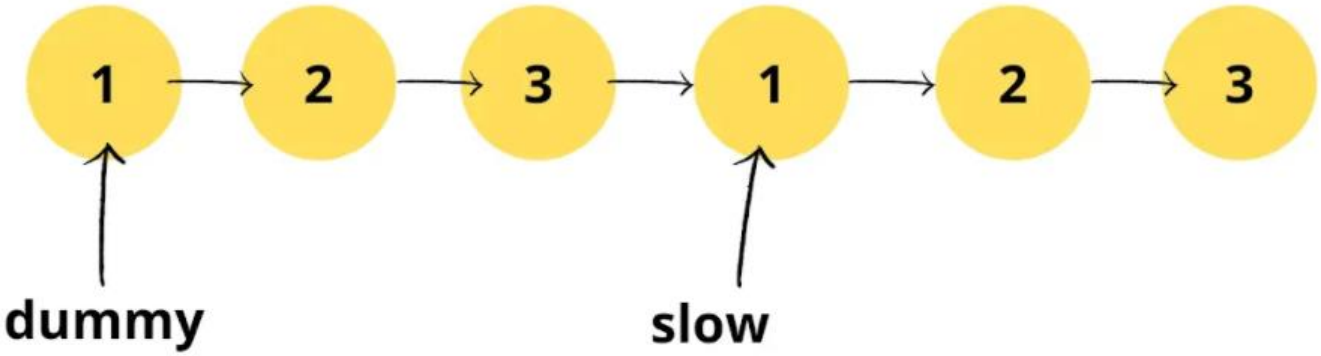
fast->next != NULL and fast->next->next != NULL
move slow to slow->next, fast to fast->next->next



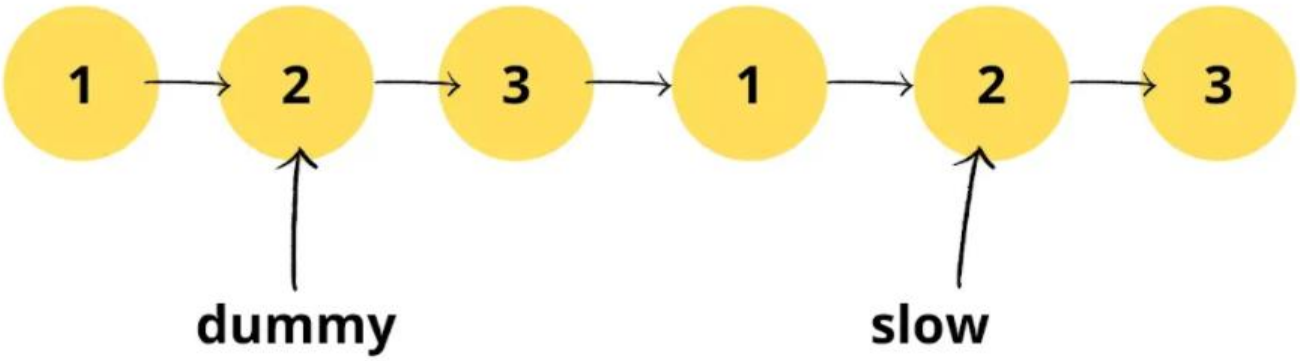
fast reached second last element. So we got oour middle element as 3. Now reverse the list from slow->next till end of the list.



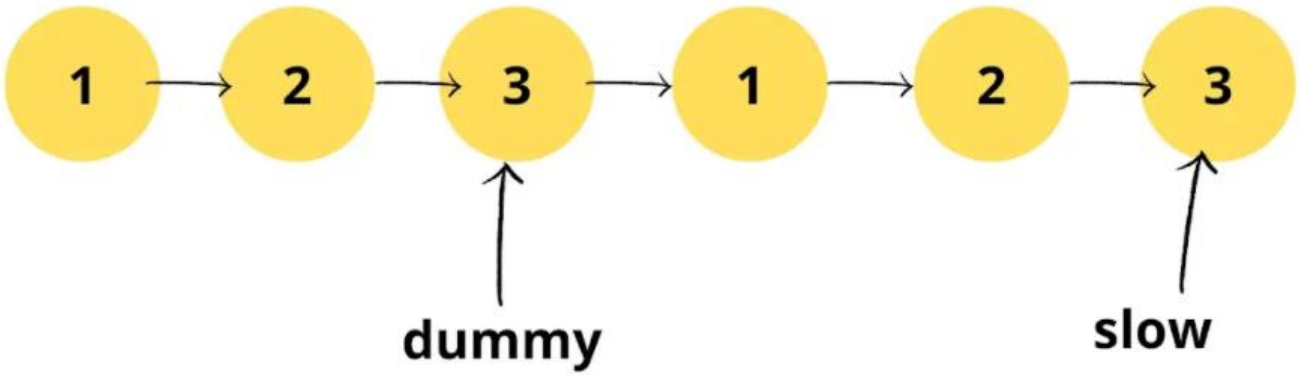
Reversed portion highlighted. Assign dummy node at the head of the list. Move slow pointer ahead. Start iterating until slow pointer reaches end of the list



dummy->num == slow->num, move dummy and slow ahead simultaneously



dummy->num == slow->num, move dummy and slow ahead simultaneously



dummy->num == slow->num, move dummy and slow ahead simultaneously. Slow reached end of the list and no element is different. Thus, list is palindrome.

Code:

```
static Node reverse(Node ptr) {
    Node pre=null;
    Node nex=null;
    while(ptr!=null) {
        nex = ptr.next;
        ptr.next = pre;
        pre=ptr;
        ptr=nex;
    }
    return pre;
}

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

```
    Node slow = head;
    Node fast = head;

    while(fast.next!=null&&fast.next.next!=null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    slow.next = reverse(slow.next);
    slow = slow.next;
    Node dummy = head;

    while(slow!=null) {
        if(dummy.num != slow.num) return false;
        dummy = dummy.next;
        slow = slow.next;
    }
    return true;
}
```

Time Complexity: O(N/2)+O(N/2)+O(N/2)

Reason: O(N/2) for finding the middle element, reversing the list from the middle element, and traversing again to find palindrome respectively.

Space Complexity: O(1)