Problem: Given a singly-linked list of integers, determine whether or not it is a palindrome.

A palindrome is something that reads the same backward as forward.

approach 1: we know that we're going to be doing some sort of reversal here.

A straightforward solution would be to iterate through the linked list & push these values into a stack.

ex 1) r → a → c → e → c → a → r → ∅

stack: r, a, c, e, c, a, r

ex 2) a → b → C → ∅

stack: a, b, C

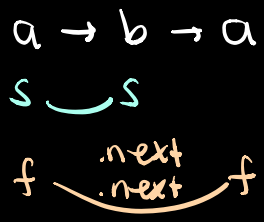
Then, iterate through the beginning of the stack again while popping the stack until it is empty.

The fast runner technique can also be used to cut the iteration in half, since one half of a palindrome should be equal to the other half reversed.

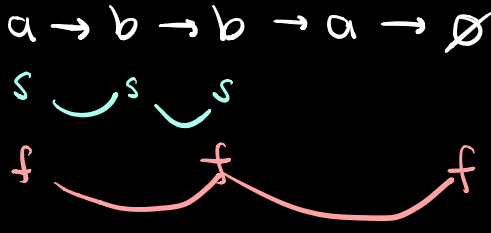
approach 2: To reduce space complexity, we can reverse the first half of the linked list, then iterate through the reversed half & the remaining un-reversed half to check if it is a valid palindrome.

1) The fast pointer travels 2 nodes for every 1 node of the slower pointer, its purpose is to reveal the midpoint of a linked list.

odd-numbered case:



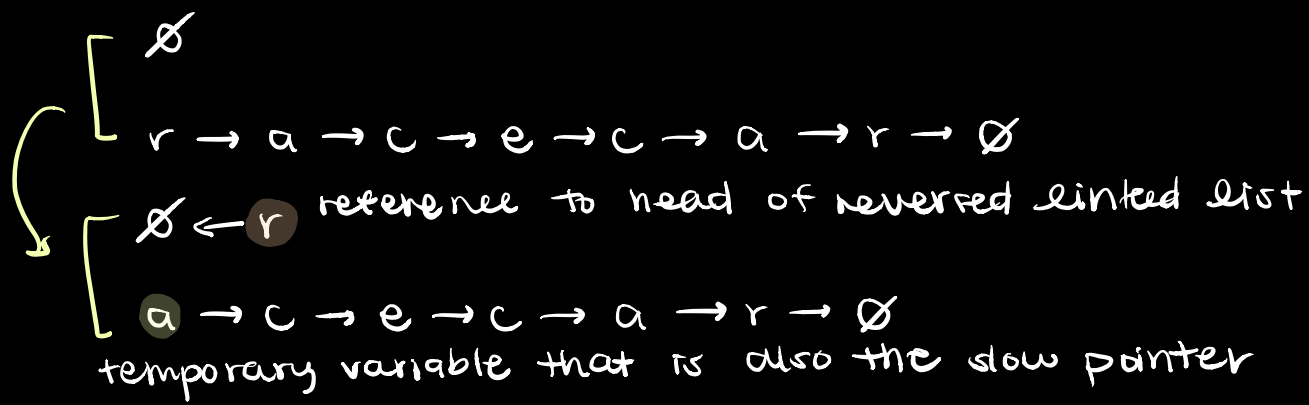
even-numbered case:



from these examples, we can conclude that we can continue iterating until f is null, or f.next is null.

2) The slow pointer moves 1 node at a time, and works as a temporary variable to overwrite the linked list & reverse it.

The first node of our linked list should be the last node of our reversed linked list. Therefore, this node's next node would be null.



Implementation

```
function isPalindrome(head) {
    let reversedHead = null;
    let fastPointer = head;
    while (fastPointer && fastPointer.next) {
        fastPointer = fastPointer.next.next;  // move the fast ptr first
        let temp = head.next;  // store the next node
        head.next = reversedHead;  // set the rev. list's next node
        reversedHead = head;  // move rev. list's pointer
        head = temp;  // move the slow pointer to next node to iterate
    }
    // the head should now be at the start of
    // the non-reversed half of the linked list
    // everything preceeding it has been prepended
    // to the reversed linked list
    // if fast is not null, the linked list has an
    // odd-numbered length, so we should disregard
```

also ensures length >= 2

```
    // the midpoint

    if (fastPointer) {
        head = head.next;
    }

    // iterate through the original linked list until we hit null
    while (head) {
        if (head.value !== reversedHead.value) {
            return false;
        }
        head = head.next;
        reversedHead = reversedHead.next;
    }

    return true;
}
```