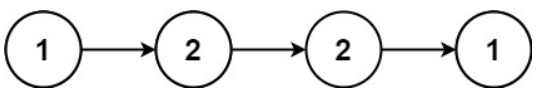
<https://leetcode.com/problems/palindrome-linked-list/>

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

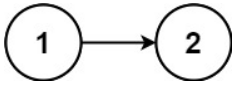
**Example 1:**



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?

**Attempt 1: 2023-02-16**

**Solution 1: Clone list and reverse list then compare (30 min)**

**Why we have to clone original list ?**

**If only reverse the original list, the mutation will happen on original list, the 'head' of original list will point to NULL after reverse as a 'tail', so before reversing original list, we have to clone it to reserve the original list for later 'reverse'**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\*    int val;

\*    ListNode next;

\*    ListNode() {}

\*    ListNode(int val) { this.val = val; }

\*    ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

    public boolean isPalindrome(ListNode head) {

        if(head == null || head.next == null) {

            return true;

        }

        ListNode clone = cloneList(head);

        ListNode rev = reverseList(head);

        ListNode iter1 = clone;

        ListNode iter2 = rev;

        while(iter1 != null) {

            if(iter1.val != iter2.val) {

                return false;

            }

            iter1 = iter1.next;

            iter2 = iter2.next;

        }

        return true;

    }

    // Why we have to clone original list?

    // If only reverse the original list, the mutation will happen on

    // original list, the 'head' of original list will point to NULL

    // after reverse as a 'tail', so before reversing original list,

    // we have to clone it to reserve the original list for later 'reverse'

    // Function takes a linked list and returns its complete copy

    private ListNode cloneList(ListNode head) {

        // 'iter' used to iterate over the original list

        ListNode iter = head;

        // 'newHead' as head of the new list

        ListNode newHead = null;

        // 'tail' point to the last node in a new list

        ListNode tail = null;

        while(iter != null) {

            // Special case for the first node of new list

            if(newHead == null) {

                newHead = new ListNode(iter.val);

                tail = newHead;

            } else {

                tail.next = new ListNode(iter.val);

                tail = tail.next;

                // Don't forget to remove old connection after 'tail' updated

// initially thought if not disconnect old connection will

// create dead loop, but after analysis, in next while loop,

// the tail.next will auto assign to next 'new ListNode(iter.val)',

// we don't need extra disconnect here

                //tail.next = null;

            }

            iter = iter.next;

        }

        return newHead;

    }

    private ListNode reverseList(ListNode head) {

        ListNode prev = null;

        ListNode cur = head;

        while(cur != null) {

            ListNode next = cur.next;

            cur.next = prev;

            prev = cur;

            cur = next;

        }

        return prev;

    }

}

Time Complexity: O(n), where n is the number of nodes in the Linked List

Space Complexity: O(n), where n is the number of nodes in the Linked List

**How to clone a linked list ?**

**Refer to**

<https://www.techiedelight.com/clone-given-linked-list/>

The idea is to iterate over the original list in the usual way and maintain two pointers to keep track of the new list: one head pointer and one tail pointer, which always points to the last node of the new list. The first node is done as a special case, and then the tail pointer is used in the standard way for the others.

// A Linked List Node

class Node

{

    int data;

    Node next;

    Node(int data, Node next)

    {

        this.data = data;

        this.next = next;

    }

    Node() {}

}

class Main

{

    // Helper function to print a given linked list

    public static void printList(Node head)

    {

        Node ptr = head;

        while (ptr != null)

        {

            System.out.print(ptr.data + " —> ");

            ptr = ptr.next;

        }

        System.out.println("null");

    }

    // Function takes a linked list and returns its complete copy

    public static Node copyList(Node head)

    {

        Node current = head;    // used to iterate over the original list

        Node newList = null;    // head of the new list

        Node tail = null;      // point to the last node in a new list

        while (current != null)

        {

            // special case for the first new node

            if (newList == null)

            {

                newList = new Node(current.data, null);

                tail = newList;

            }

            else {

                tail.next = new Node();

                tail = tail.next;

                tail.data = current.data;

                tail.next = null;

            }

            current = current.next;

        }

        return newList;

    }

    public static void main(String[] args)

    {

        // input keys

        int[] keys = {1, 2, 3, 4};

        // points to the head node of the linked list

        Node head = null;

        // construct a linked list

        for (int i = keys.length - 1; i >= 0; i--) {

            head = new Node(keys[i], head);

        }

        // copy linked list

        Node copy = copyList(head);

        // print duplicate linked list

        printList(copy);

    }

}

**Solution 2: Recursion and naturally get tail node recursively for comparison (30 min)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\*    int val;

\*    ListNode next;

\*    ListNode() {}

\*    ListNode(int val) { this.val = val; }

\*    ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

    ListNode left;

    public boolean isPalindrome(ListNode head) {

        left = head;

        return helper(head);

    }

    private boolean helper(ListNode right) {

        // Return true if you've reached the end of the list

        if(right == null) {

            return true;

        }

        // Keep pointing towards right and call function recursively

        // so that, when each recursive call is popped from the stack,

        // the nodes on the right will be compared in reverse order

        boolean result = helper(right.next);

        // If sub-list is not a palindrome then return false

        if(!result) {

            return false;

        }

        // Compare current left and right node value

        if(left.val != right.val) {

            return false;

        }

        // Move left node to next node, prepare for next recursion comparison

        left = left.next;

        return true;

    }

}

Time Complexity: O(n), where n is the number of nodes in the Linked List

The recursive function is run once for each of the nnn nodes, and the body of the recursive function is O(1). Therefore, this gives a total of O(n).

Space Complexity: O(n), where n is the number of nodes in the Linked List

I hinted at the start that this is not using O(1) space. This might seem strange, after all we aren't creating any new data structures. So, where is the O(n) extra memory we're using? Understanding what is happening here requires understanding how the computer runs a recursive function.

Each time a function is called within a function, the computer needs to keep track of where it is up to (and the values of any local variables) in the current function before it goes into the called function. It does this by putting an entry on something called the runtime stack, called a stack frame. Once it has created a stack frame for the current function, it can then go into the called function. Then once it is finished with the called function, it pops the top stack frame to resume the function it had been in before the function call was made.

Before doing any palindrome checking, the above recursive function creates n of these stack frames because the first step of processing a node is to process the nodes after it, which is done with a recursive call. Then once it has the nnn stack frames, it pops them off one-by-one to process them.

So, the space usage is on the runtime stack because we are creating nnn stack frames. Always make sure to consider what's going on the runtime stack when analyzing a recursive function. It's a common mistake to forget to.

Not only is this approach still using O(n) space, it is worse than the first approach because in many languages (such as Python), stack frames are large, and there's a maximum runtime stack depth of 1000 (you can increase it, but you risk causing memory errors with the underlying interpreter). With every node creating a stack frame, this will greatly limit the maximum Linked List size the algorithm can handle.

**Refer to**

<https://leetcode.com/problems/palindrome-linked-list/solutions/346730/java-1-ms-recursive-solution-easy-to-understand/>

Keep left & right pointer at head.

Call the function recursively with 'right.next' until you reach the end of the list.

When recursive calls end and are popped of the stack, we compare left and right node values and return the result.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\*    int val;

\*    ListNode next;

\*    ListNode(int x) { val = x; }

\* }

\*/

class Solution {

    ListNode left;

    public boolean isPalindrome(ListNode head) {

        left = head;

        return isListPalindrome(head);

    }

    boolean isListPalindrome(ListNode right) {

        // return true if you've reached the end of the list

        if (right == null) {

            return true;

        }

        // keep pointing towards right and call function recursively

        // so that, when each recursive call is popped from the stack,

        // the nodes on the right will be compared in reverse order

        boolean isPalin = isListPalindrome(right.next);

        // if sub-list is not a palindrome then return false

        if (!isPalin)

            return false;

        // compare current left and right node value

        boolean isEqual = left.val == right.val;

        left = left.next;

        return isEqual;

    }

}

**Refer to**

<https://leetcode.com/problems/palindrome-linked-list/solutions/433547/palindrome-linked-list/>

**Approach 2: Recursive (Advanced)**

**Intuition**

In an attempt to come up with a way of using O(1) space, you might have thought of using recursion. However, this is still O(n)space. Let's have a look at it and understand why it is **not** O(1) space.

Recursion gives us an elegant way to iterate through the nodes in reverse. For example, this algorithm will print out the values of the nodes *in reverse*. Given a node, the algorithm checks if it is null. If it is null, nothing happens. Otherwise, all nodes *after* it are processed, and *then* the value for the current node is printed.

function print\_values\_in\_reverse(ListNode head)

    if head is NOT null

        print\_values\_in\_reverse(head.next)

        print head.val

If we iterate the nodes in reverse using recursion, and iterate forward at the same time using a variable *outside* the recursive function, then we can check whether or not we have a palindrome.

**Algorithm**

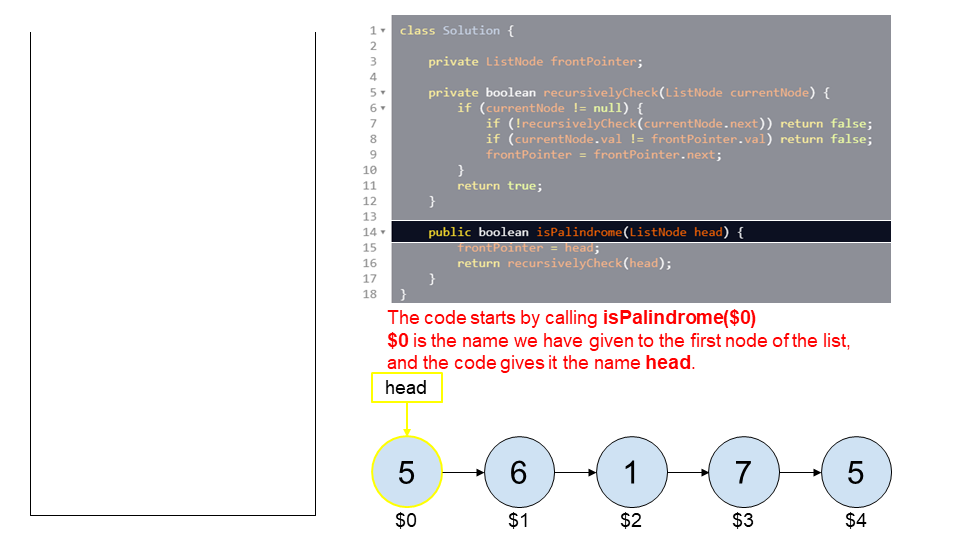
When given the head node (or any other node), referred to as currentNode, the algorithm firstly checks the *rest* of the Linked List. If it discovers that further down that the Linked List is *not* a palindrome, then it returns false. Otherwise, it checks that currentNode.val == frontPointer.val. If not, then it returns false. Otherwise, it moves frontPointer forward by 1 node and returns true to say that from this point forward, the Linked List is a valid palindrome.

It might initially seem surprisingly that frontPointer is always pointing where we want it. The reason it works is because the order in which nodes are processed by the recursion is in reverse (remember our "printing" algorithm above). Each node compares itself against frontPointer and then moves frontPointer down by 1, ready for the next node in the recursion. In essence, we are iterating both backwards and forwards at the same time.

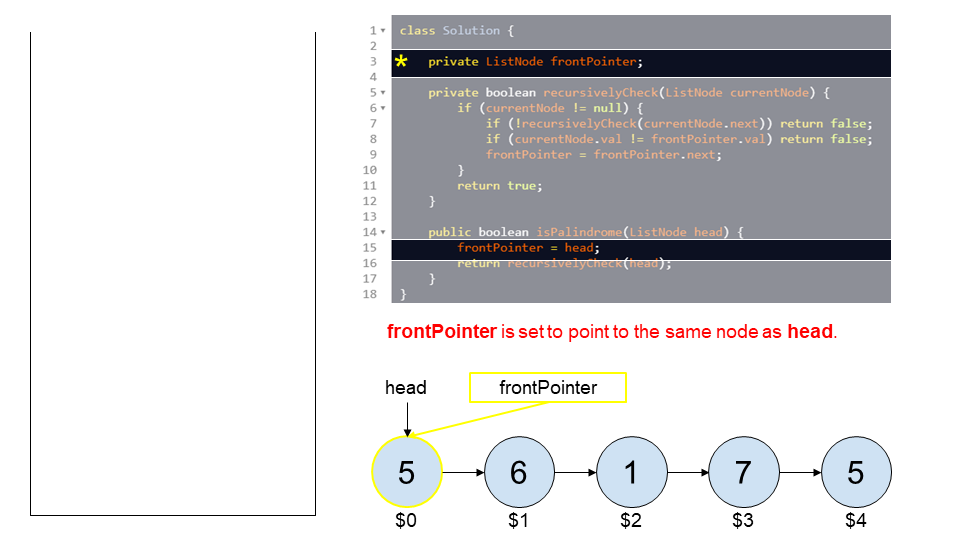
Here is an animation that shows how the algorithm works. The nodes have each been given a unique identifier (e.g. $1 and $4) so that they can more easily be referred to in the explanations. The computer needs to use its runtime stack for recursive functions.

**Step by step simulation recursion stack**

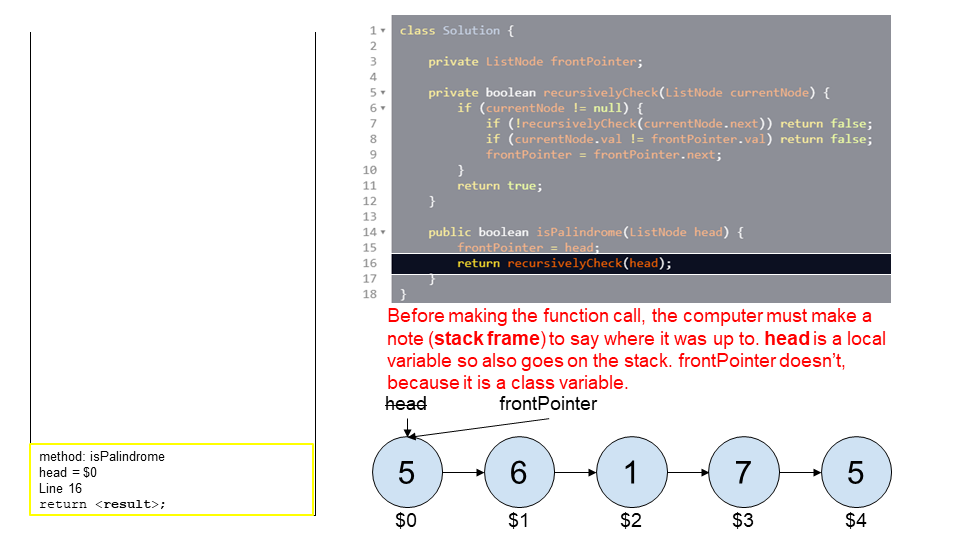
Step 1



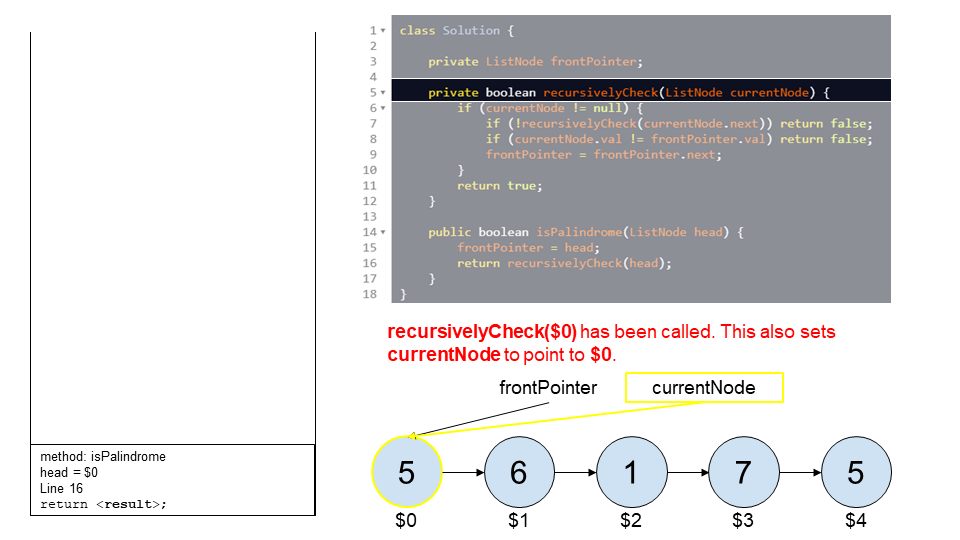
Step 2



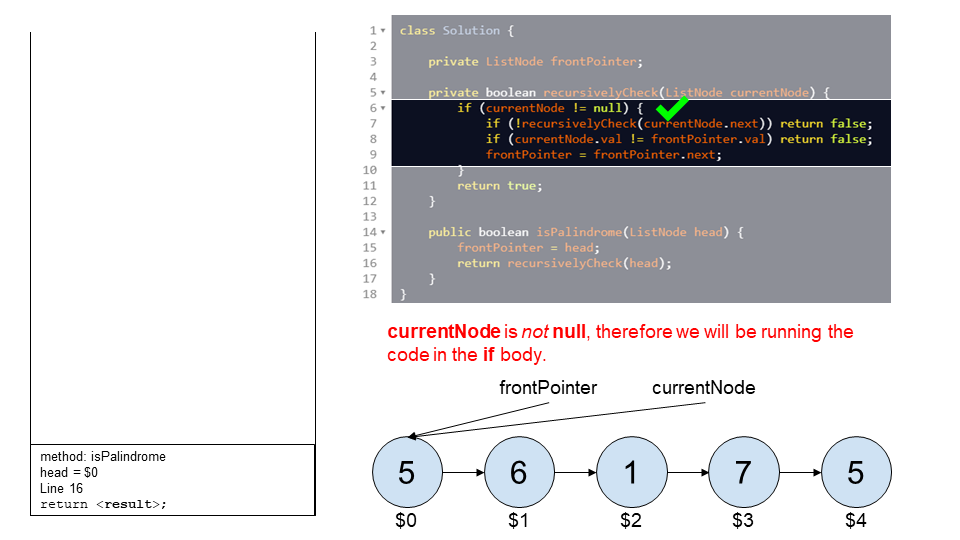
Step 3



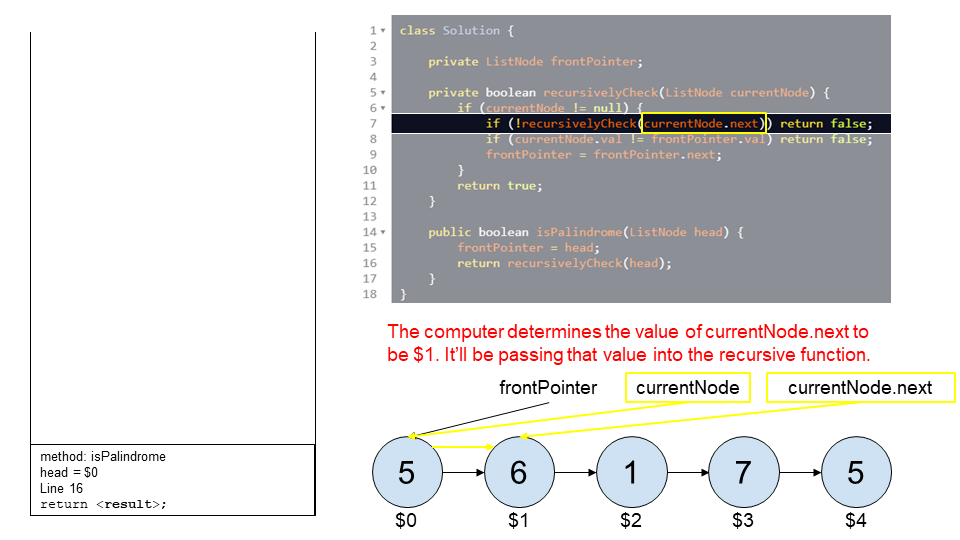
Step 4



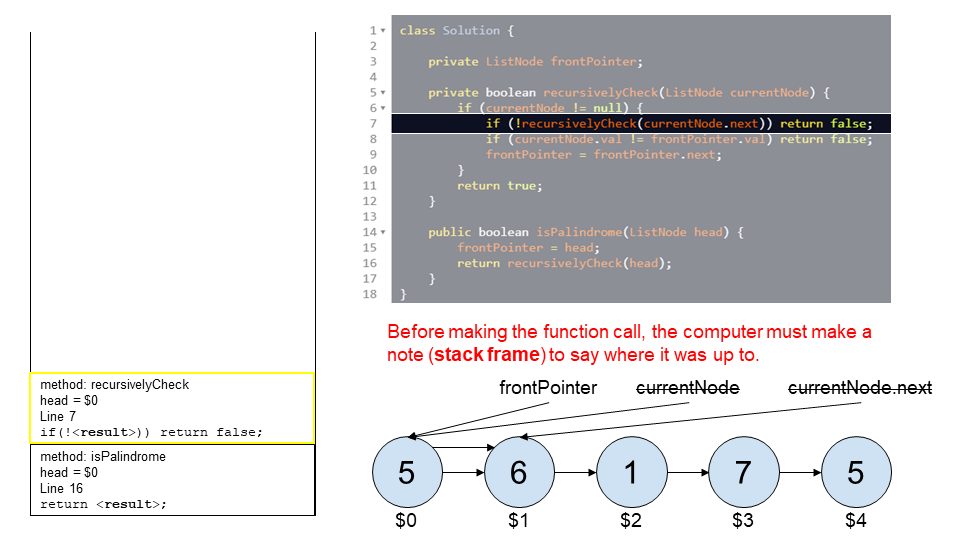
Step 5



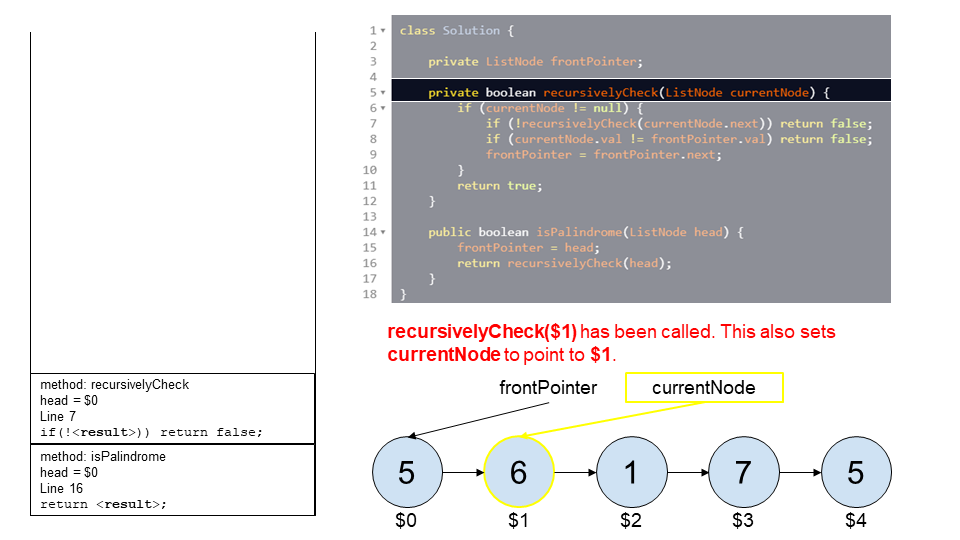
Step 6



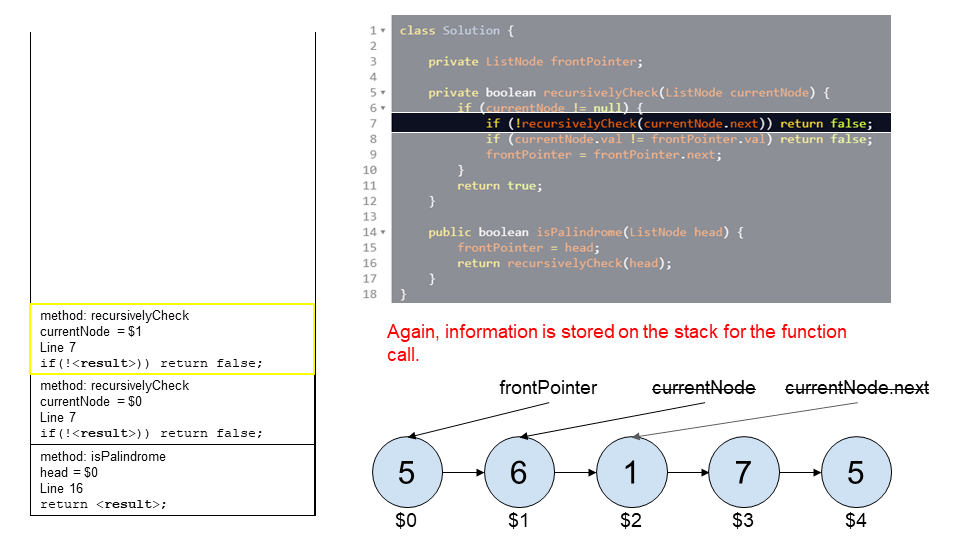
Step 7



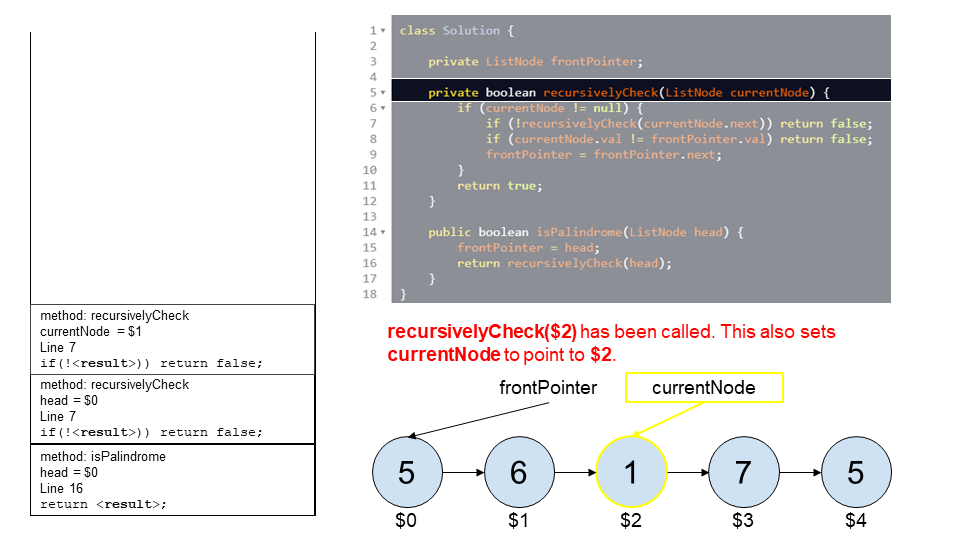
Step 8



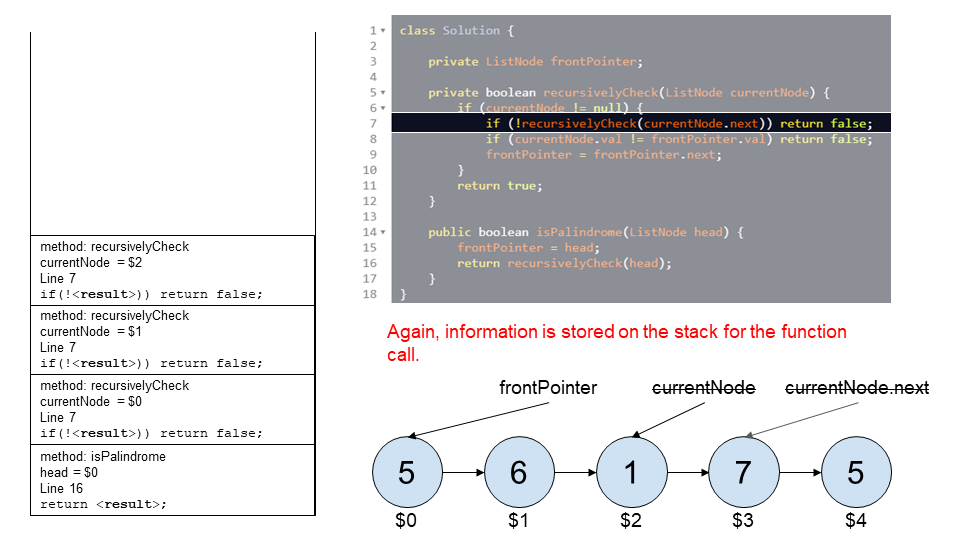
Step 9



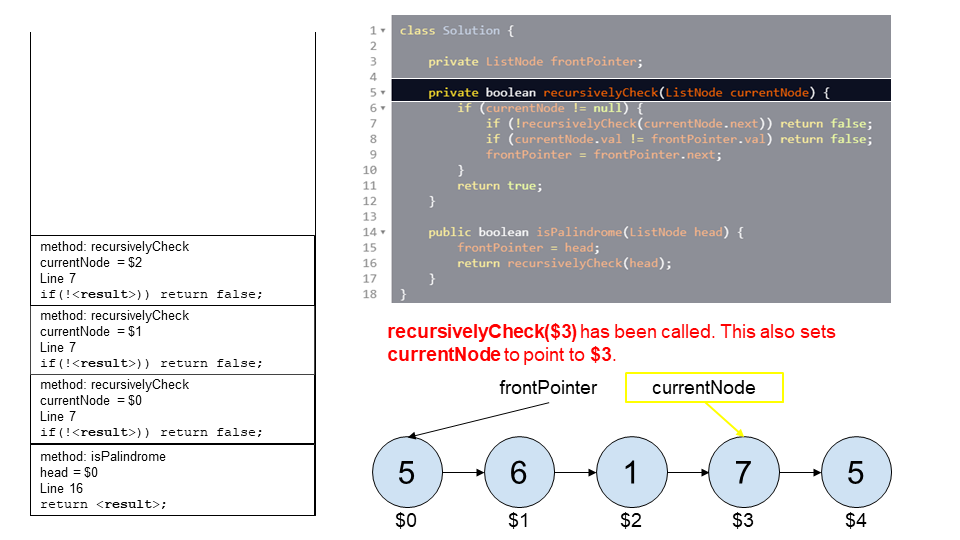
Step 10



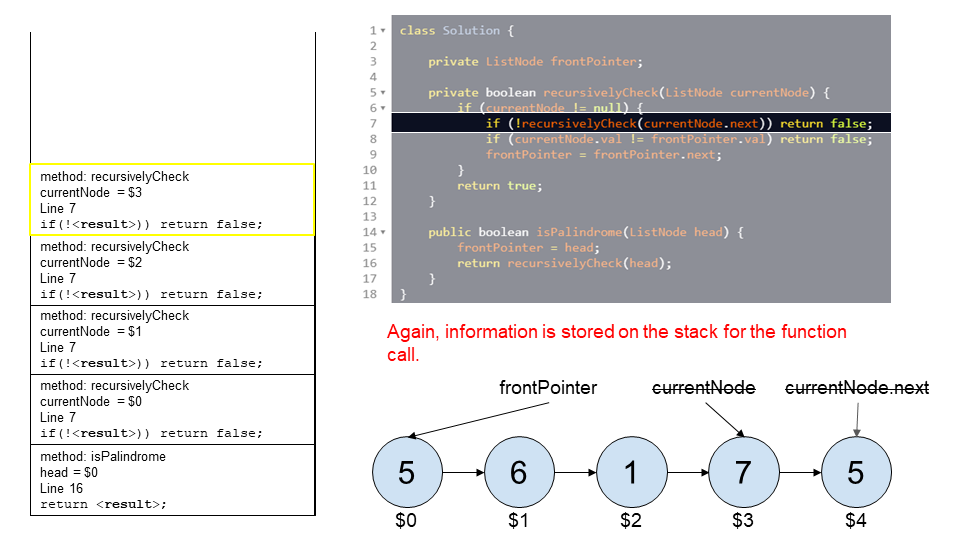
Step 11



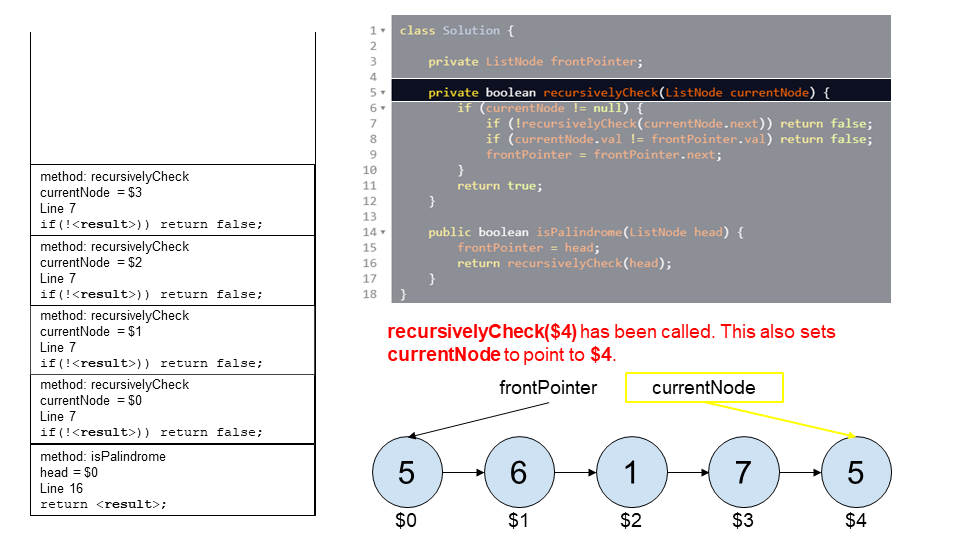
Step 12



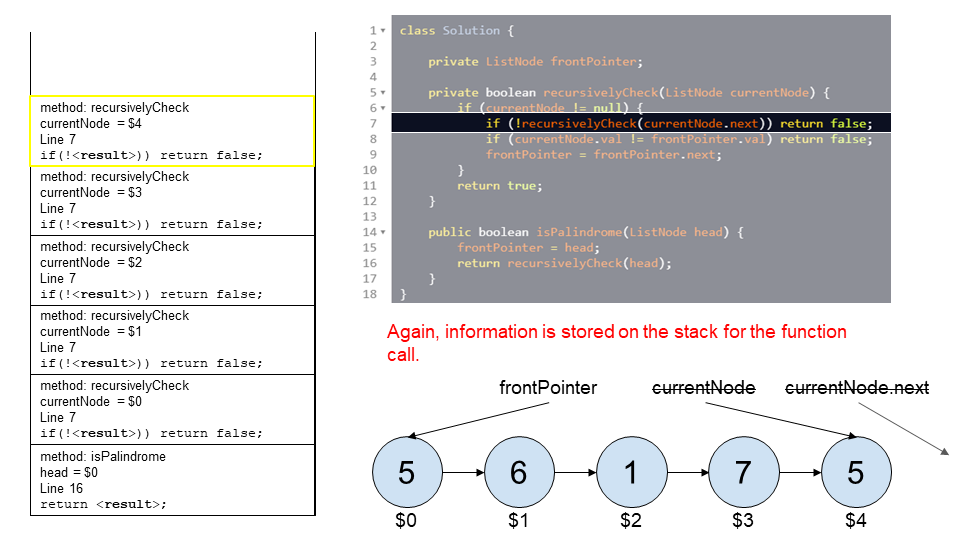
Step 13



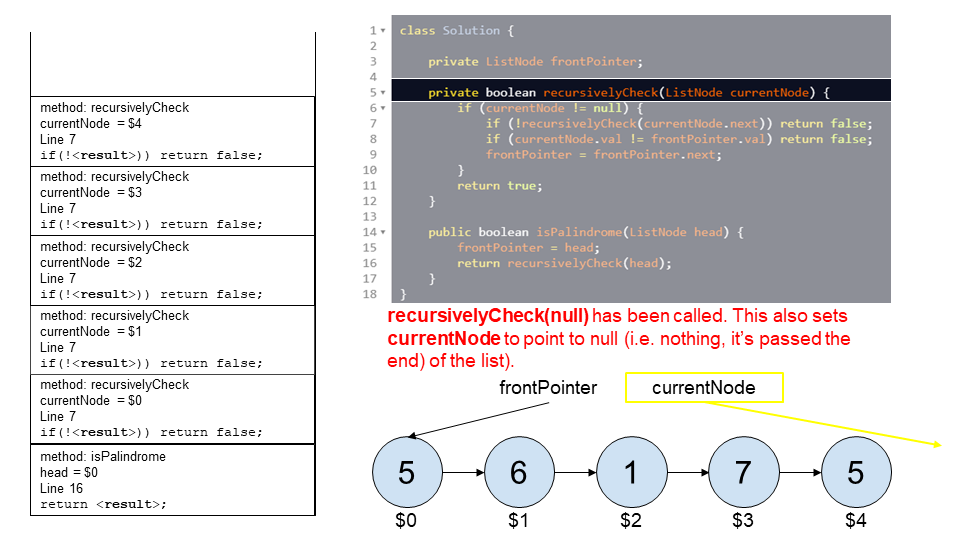
Step 14



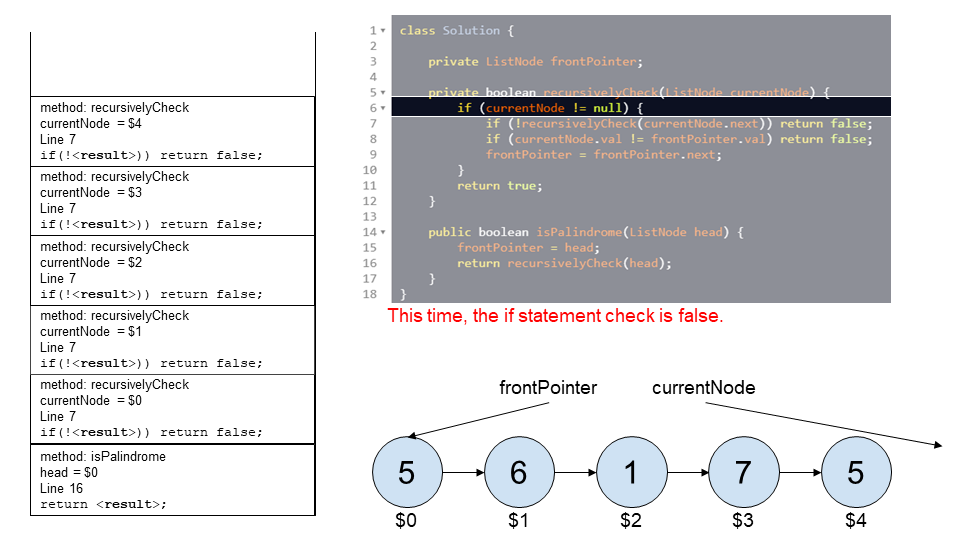
Step 15



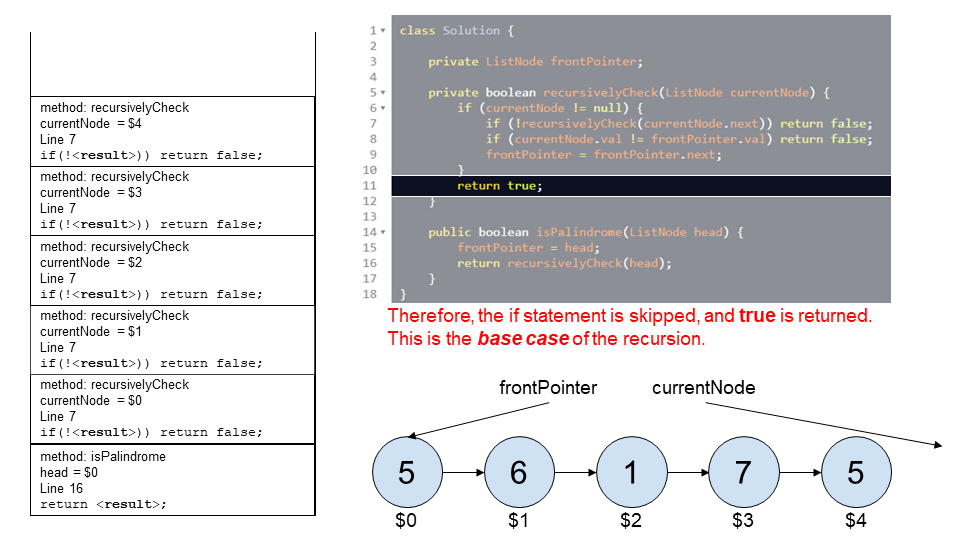
Step 16



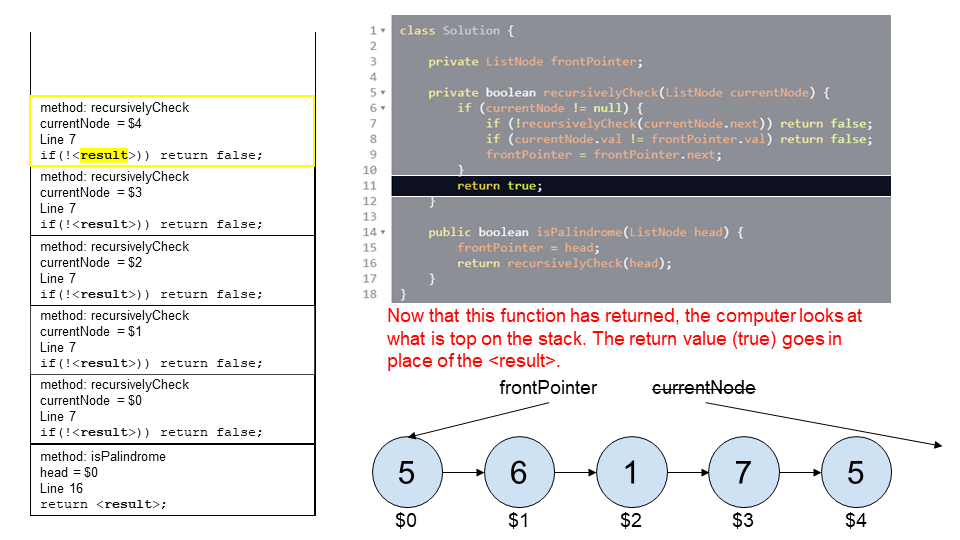
Step 17



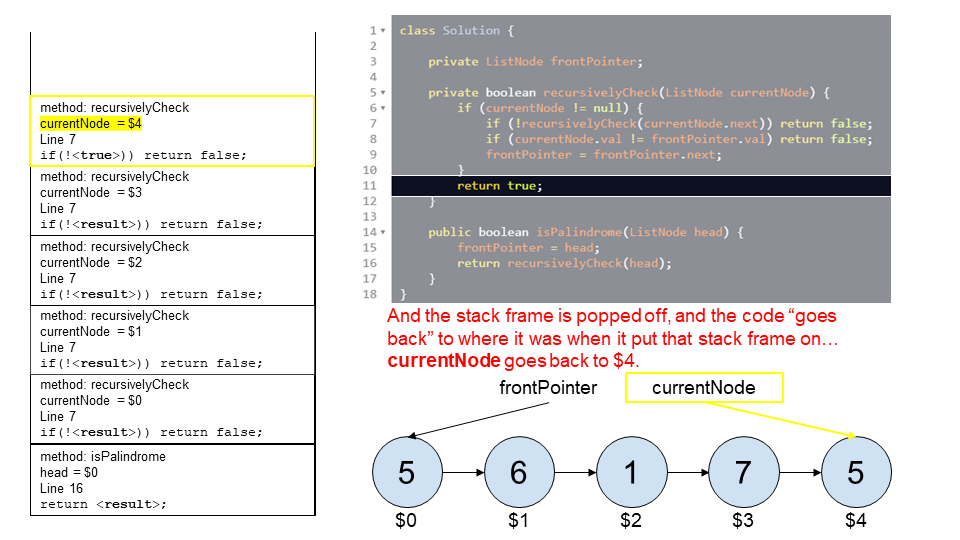
Step 18



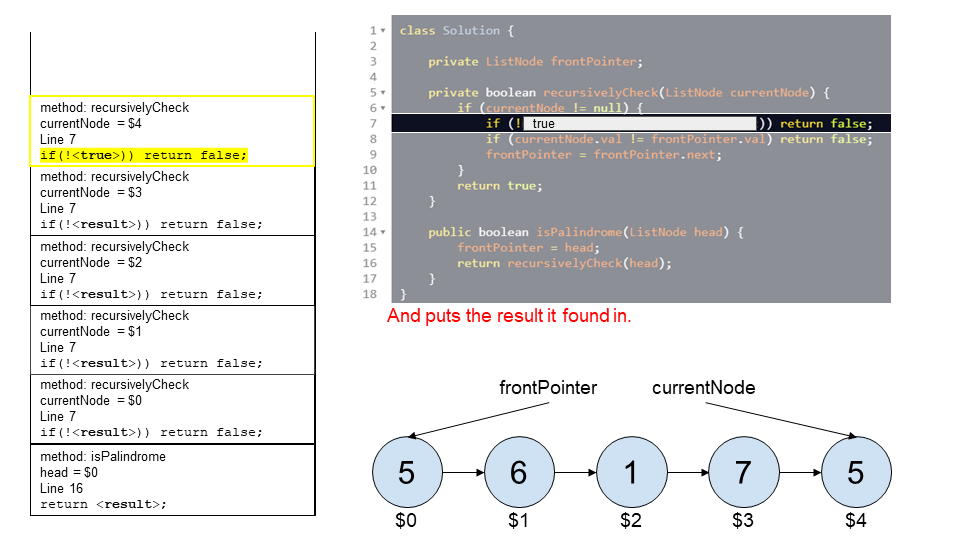
Step 19



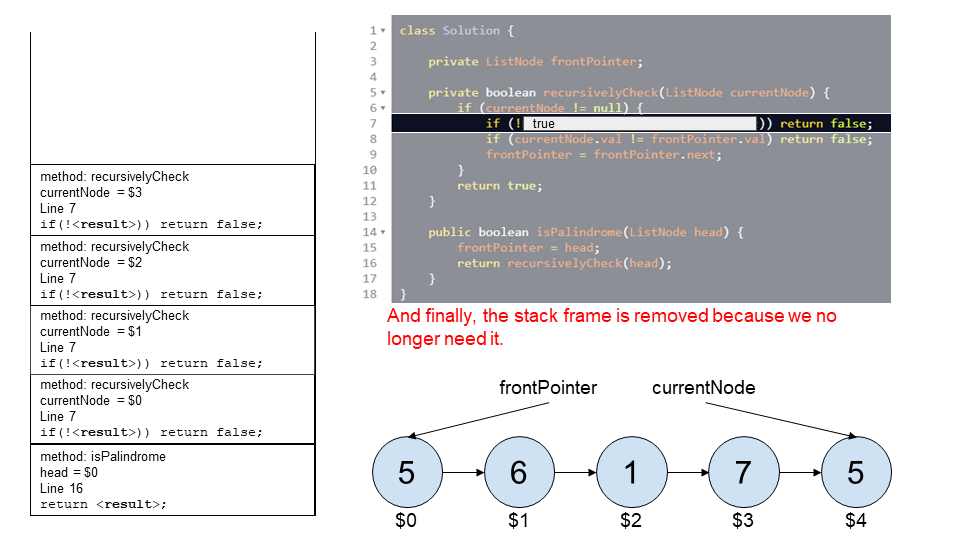
Step 20



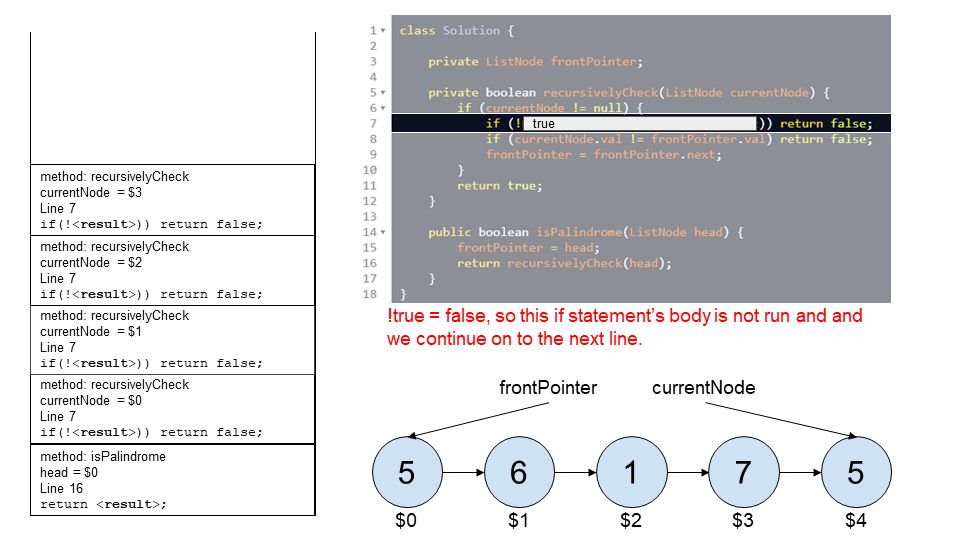
Step 21



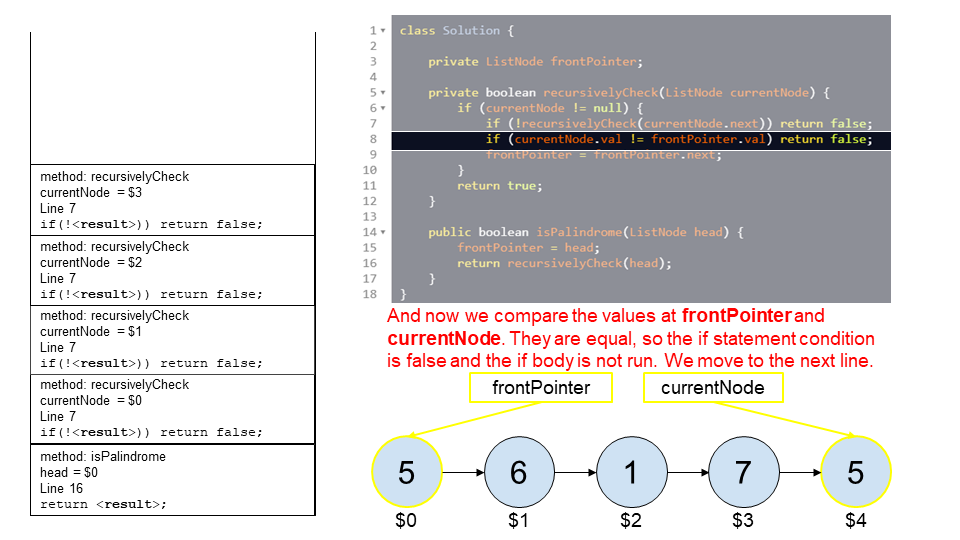
Step 22



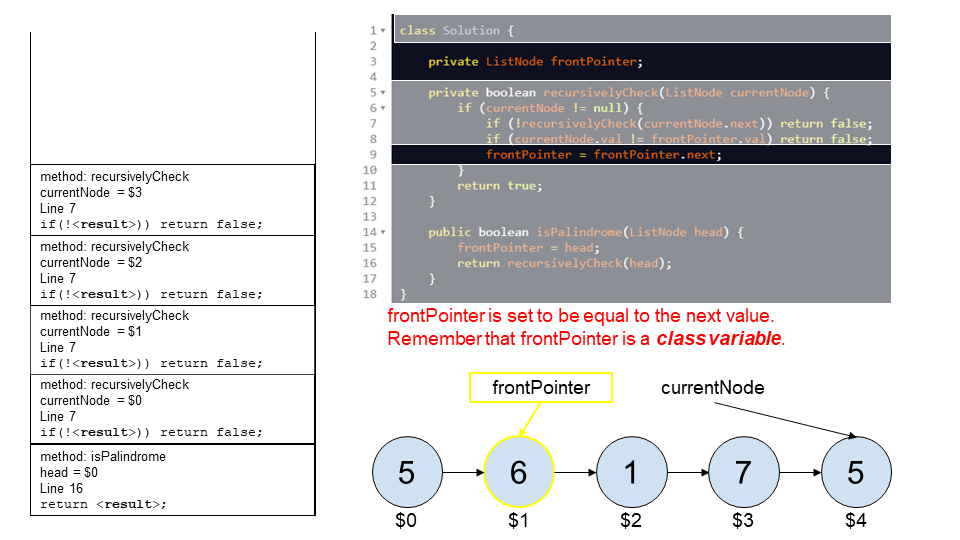
Step 23



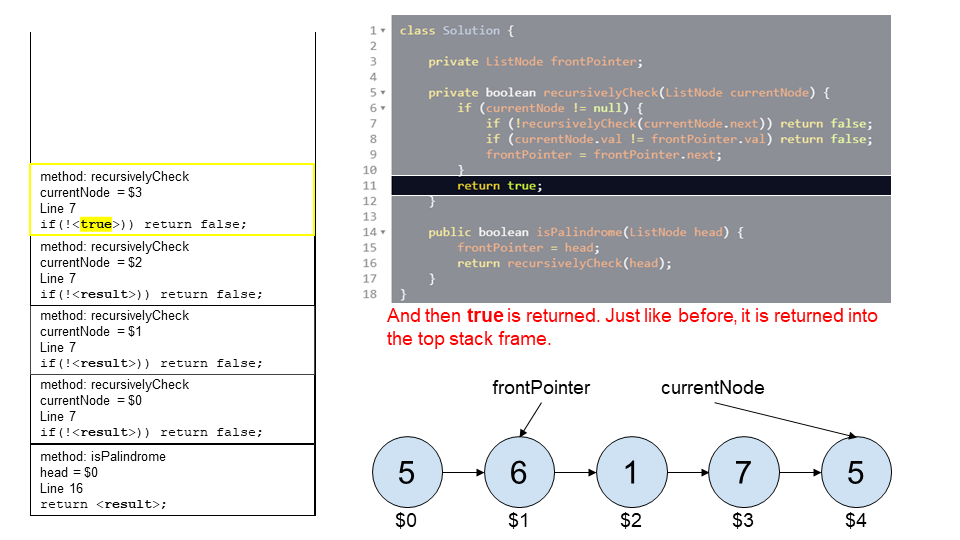
Step 24



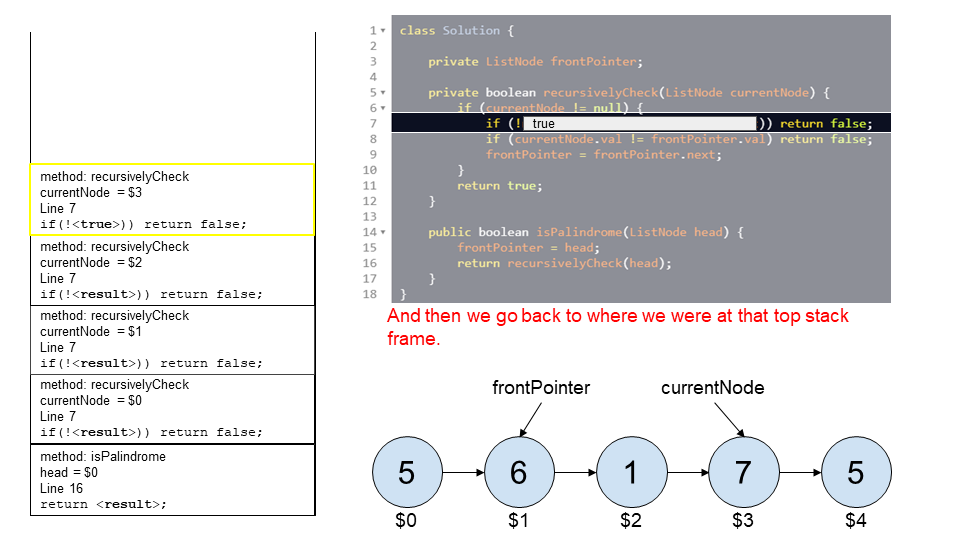
Step 25



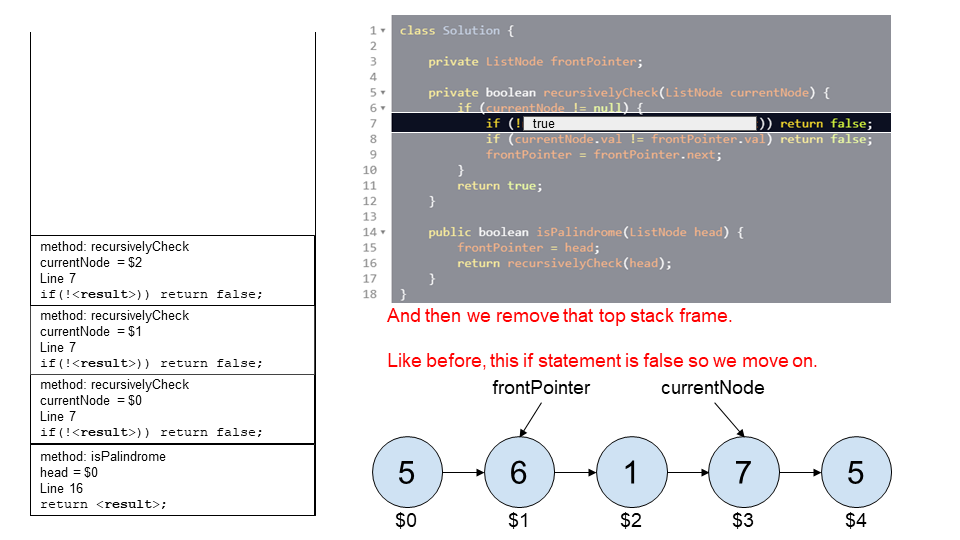
Step 26



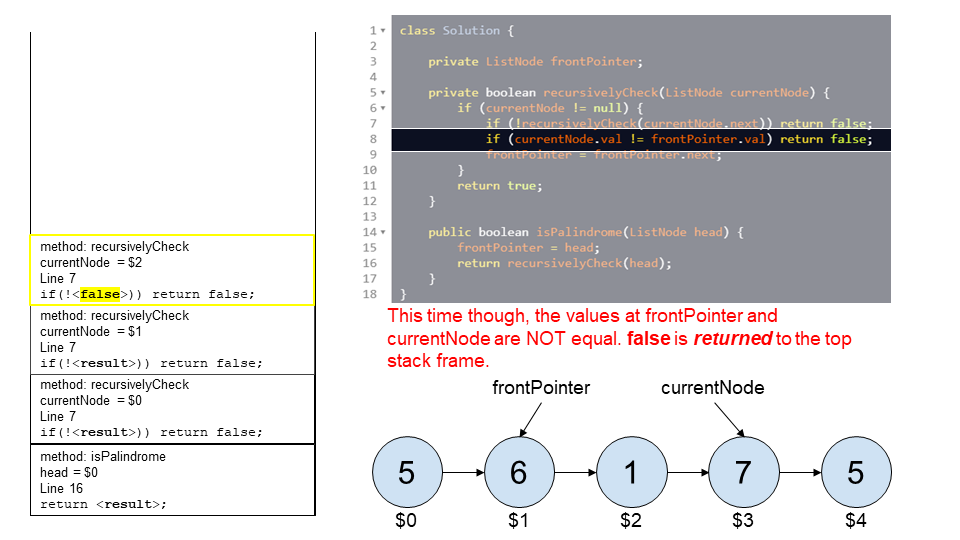
Step 27



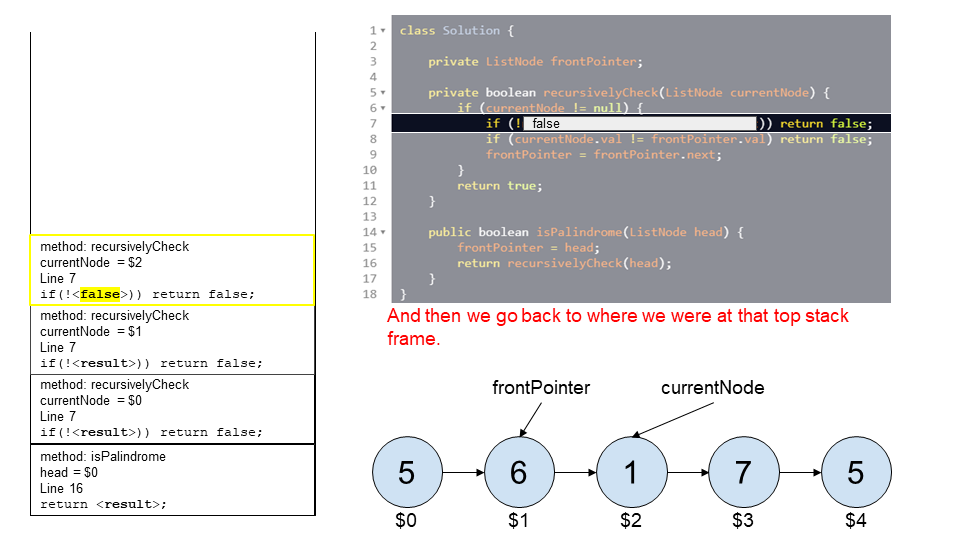
Step 28



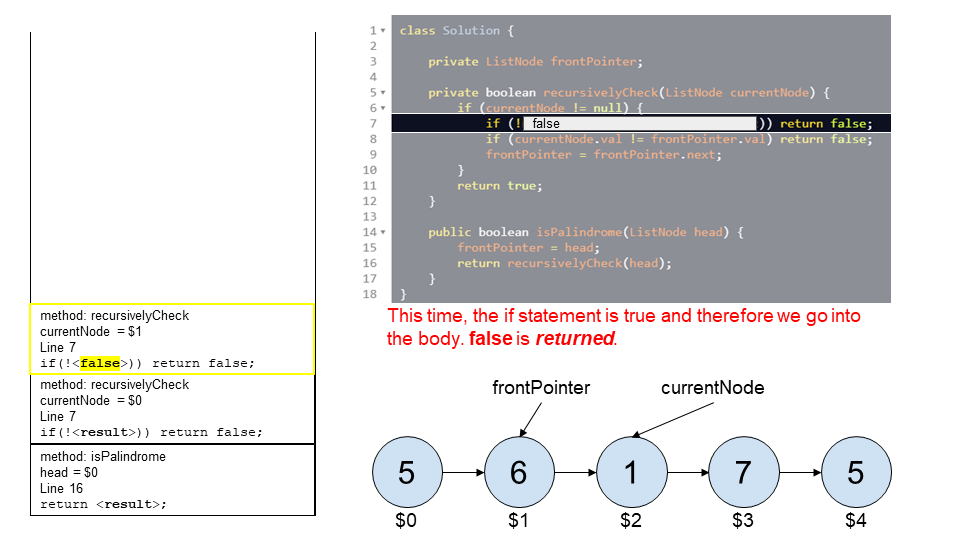
Step 29



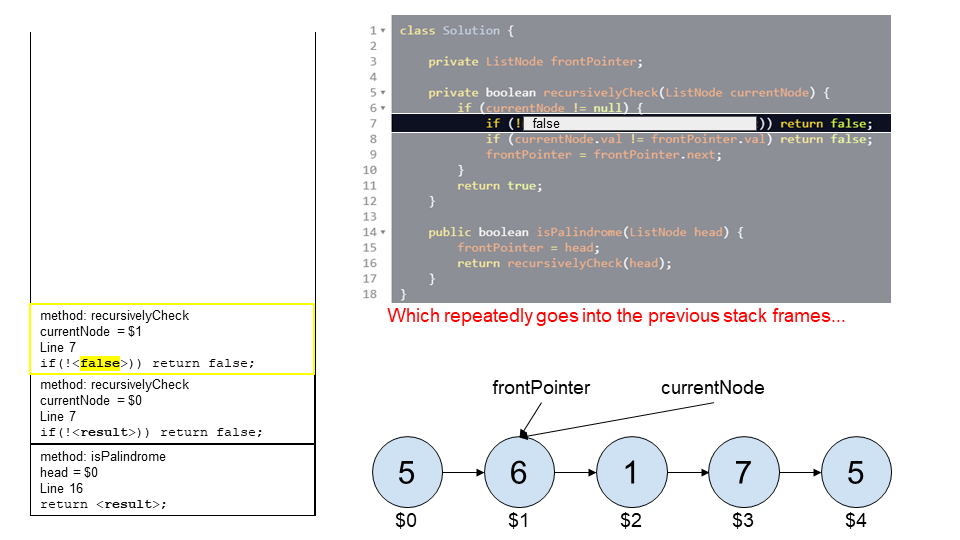
Step 30



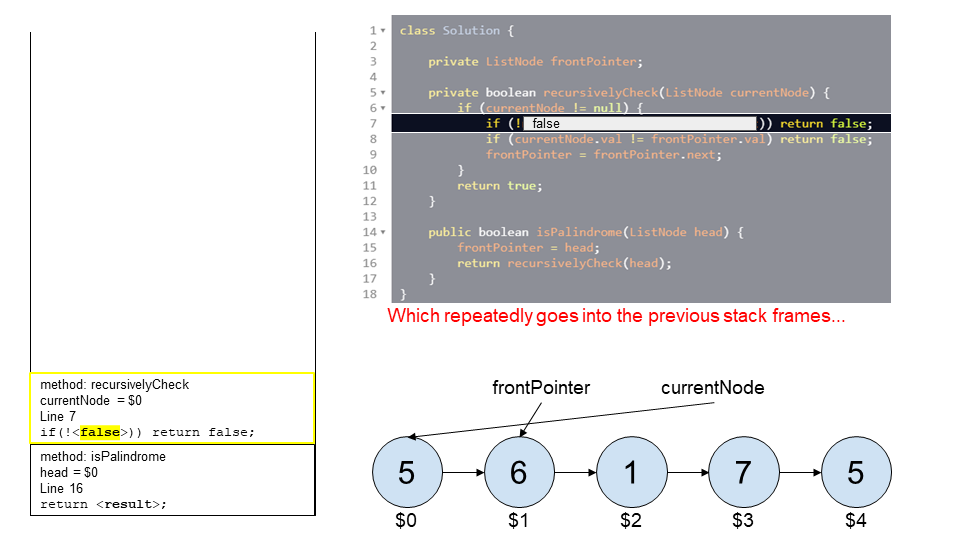
Step 31



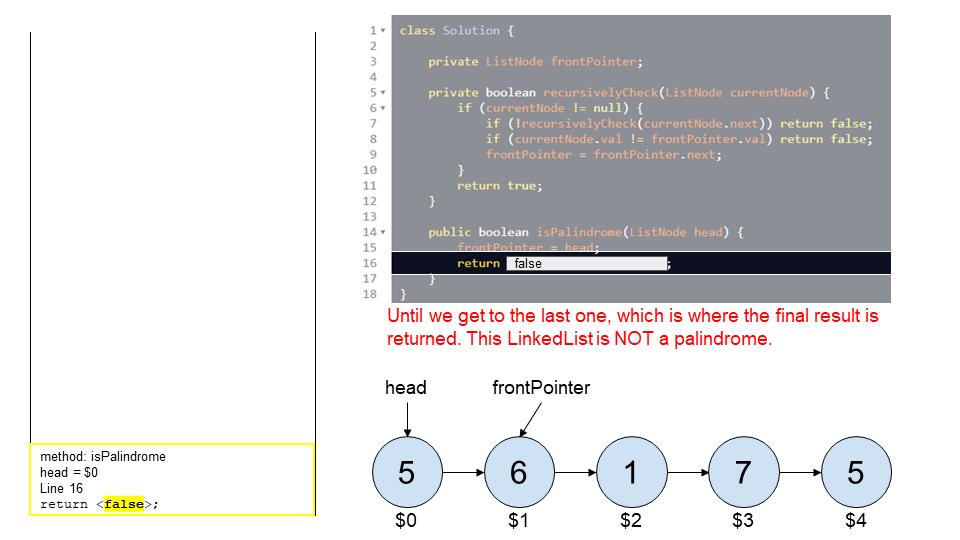
Step 32



Step 33



Step 34



class Solution {

    private ListNode frontPointer;

    private boolean recursivelyCheck(ListNode currentNode) {

        if (currentNode != null) {

            if (!recursivelyCheck(currentNode.next)) return false;

            if (currentNode.val != frontPointer.val) return false;

            frontPointer = frontPointer.next;

        }

        return true;

    }

    public boolean isPalindrome(ListNode head) {

        frontPointer = head;

        return recursivelyCheck(head);

    }

}

**Complexity Analysis**

Time complexity : O(n) , where n is the number of nodes in the Linked List.

The recursive function is run once for each of the n nodes, and the body of the recursive function is O(1). Therefore, this gives a total of O(n).

Space complexity : O(n), where n is the number of nodes in the Linked List.

I hinted at the start that this is not using O(1) space. This might seem strange, after all we aren't creating any new data structures. So, where is the O(n) extra memory we're using? Understanding what is happening here requires understanding how the computer runs a recursive function.

Each time a function is called within a function, the computer needs to keep track of where it is up to (and the values of any local variables) in the current function before it goes into the called function. It does this by putting an entry on something called the **runtime stack**, called a **stack frame**. Once it has created a stack frame for the current function, it can then go into the called function. Then once it is finished with the called function, it pops the top stack frame to resume the function it had been in before the function call was made.

Before doing any palindrome checking, the above recursive function creates n of these stack frames because the first step of processing a node is to process the nodes after it, which is done with a recursive call. Then once it has the n stack frames, it pops them off one-by-one to process them.

So, the space usage is on the *runtime stack* because we are creating n stack frames. Always make sure to consider what's going on the *runtime stack* when analyzing a recursive function. It's a common mistake to forget to.

Not only is this approach still using O(n) space, it is worse than the first approach because in many languages (such as Python), stack frames are large, and there's a maximum runtime stack depth of 1000 (you can increase it, but you risk causing memory errors with the underlying interpreter). With *every* node creating a stack frame, this will greatly limit the maximum Linked List size the algorithm can handle.

**Solution 3: Reverse second half in-place (30 min)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\*    int val;

\*    ListNode next;

\*    ListNode() {}

\*    ListNode(int val) { this.val = val; }

\*    ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

    public boolean isPalindrome(ListNode head) {

        ListNode secondHalfHead = findSecondHalfHead(head);

        ListNode revSecondHalfHead = reverse(secondHalfHead);

        ListNode iter1 = head;

        ListNode iter2 = revSecondHalfHead;

        while(iter1 != null && iter2 != null) {

            if(iter1.val != iter2.val) {

                return false;

            }

            iter1 = iter1.next;

            iter2 = iter2.next;

        }

        return true;

    }

    private ListNode findSecondHalfHead(ListNode head) {

        ListNode slow = head;

        ListNode fast = head;

        while(fast.next != null && fast.next.next != null) {

            slow = slow.next;

            fast = fast.next.next;

        }

        return slow.next;

    }

    private ListNode reverse(ListNode secondHalfHead) {

        ListNode prev = null;

        ListNode cur = secondHalfHead;

        while(cur != null) {

            ListNode next = cur.next;

            cur.next = prev;

            prev = cur;

            cur = next;

        }

        return prev;

    }

}

Time complexity : O(n), where n is the number of nodes in the Linked List.

Similar to the above approaches. Finding the middle is O(n), reversing a list in place is O(n),

and then comparing the 2 resulting Linked Lists is also O(n).

Space complexity : O(1)

We are changing the next pointers for half of the nodes. This was all memory that had already

been allocated, so we are not using any extra memory and therefore it is O(1).

**Refer to**

<https://leetcode.com/problems/palindrome-linked-list/solutions/433547/palindrome-linked-list/>

**Approach 3: Reverse Second Half In-place**

**Intuition**

The ***only*** way we can avoid using O(n) extra space is by modifying the input in-place.

The strategy we can use is to reverse the second half of the Linked List in-place (modifying the Linked List structure), and then comparing it with the first half. Afterwards, we should re-reverse the second half and put the list back together. While you don't need to restore the list to pass the test cases, it is still good programming practice because the function could be a part of a bigger program that doesn't want the Linked List broken.

**Algorithm**

Specifically, the steps we need to do are:

Find the end of the first half.

Reverse the second half.

Determine whether or not there is a palindrome.

Restore the list.

Return the result.

To do *step 1*, we could count the number of nodes, calculate how many nodes are in the first half, and then iterate back down the list to find the end of the first half. Or, we could do it in a single parse using the **two runners pointer technique**. Either is acceptable, however we'll have a look at the two runners pointer technique here.

Imagine we have 2 runners one fast and one slow, running down the nodes of the Linked List. In each second, the fast runner moves down 2 nodes, and the slow runner just 1 node. By the time the fast runner gets to the end of the list, the slow runner will be half way. By representing the runners as pointers, and moving them down the list at the corresponding speeds, we can use this trick to find the middle of the list, and then split the list into two halves.

If there is an odd-number of nodes, then the "middle" node should remain attached to the first half.

*Step 2* uses the algorithm that can be found in the solution article for the [Reverse Linked List](https://leetcode.com/problems/reverse-linked-list/) problem to reverse the second half of the list.

*Step 3* is fairly straightforward. Remember that we have the first half, which might also contain a "middle" node at the end, and the second half, which is reversed. We can step down the lists simultaneously ensuring the node values are equal. When the node we're up to in the second list is null, we know we're done. If there was a middle value attached to the end of the first list, it is correctly ignored by the algorithm. The result should be saved, but not returned, as we still need to restore the list.

*Step 4* requires using the same function you used for step 2, and then for *step 5*the saved result should be returned.

class Solution {

    public boolean isPalindrome(ListNode head) {

        if (head == null) return true;

        // Find the end of first half and reverse second half.

        ListNode firstHalfEnd = endOfFirstHalf(head);

        ListNode secondHalfStart = reverseList(firstHalfEnd.next);

        // Check whether or not there is a palindrome.

        ListNode p1 = head;

        ListNode p2 = secondHalfStart;

        boolean result = true;

        while (result && p2 != null) {

            if (p1.val != p2.val) result = false;

            p1 = p1.next;

            p2 = p2.next;

        }

        // Restore the list and return the result.

        firstHalfEnd.next = reverseList(secondHalfStart);

        return result;

    }

    // Taken from https://leetcode.com/problems/reverse-linked-list/solution/

    private ListNode reverseList(ListNode head) {

        ListNode prev = null;

        ListNode curr = head;

        while (curr != null) {

            ListNode nextTemp = curr.next;

            curr.next = prev;

            prev = curr;

            curr = nextTemp;

        }

        return prev;

    }

    private ListNode endOfFirstHalf(ListNode head) {

        ListNode fast = head;

        ListNode slow = head;

        while (fast.next != null && fast.next.next != null) {

            fast = fast.next.next;

            slow = slow.next;

        }

        return slow;

    }

}

**Complexity Analysis**

Time complexity : O(n), where n is the number of nodes in the Linked List.

Similar to the above approaches. Finding the middle is O(n), reversing a list in place is O(n), and then comparing the 2 resulting Linked Lists is also O(n).

Space complexity : O(1).

We are changing the next pointers for half of the nodes. This was all memory that had already been allocated, so we are not using any extra memory and therefore it is O(1).

I have seen some people on the discussion forum saying it has to be O(n) because we're creating a new list. This is incorrect, because we are changing each of the pointers one-by-one, in-place. We are not needing to allocate more than O(1) extra memory to do this work, and there is O(1) stack frames going on the stack. It is the same as reversing the values in an Array in place (using the two-pointer technique).

The downside of this approach is that in a concurrent environment (multiple threads and processes accessing the same data), access to the Linked List by other threads or processes would have to be locked while this function is running, because the Linked List is temporarily broken. This is a limitation of many in-place algorithms though.

**Refer to**

<https://leetcode.com/problems/palindrome-linked-list/solutions/1137027/js-python-java-c-easy-floyd-s-reversal-solution-w-explanation/>

Idea:

The naive approach here would be to run through the linked list and create an array of its values, then compare the array to its reverse to find out if it's a palindrome. Though this is easy enough to accomplish, we're challenged to find an approach with a space complexity of only O(1) while maintaining a time complexity of O(N).

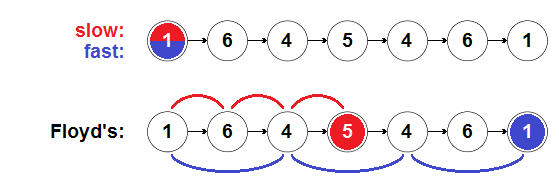
The only way to check for a palindrome in O(1) space would require us to be able to access both nodes for comparison at the same time, rather than storing values for later comparison. This would seem to be a challenge, as the linked list only promotes travel in one direction.

But what if it didn't?

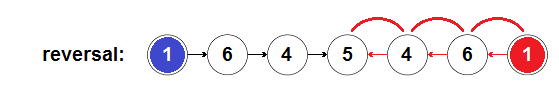
The answer is to reverse the back half of the linked list to have the next attribute point to the previous node instead of the next node. (Note: we could instead add a prev attribute as we iterate through the linked list, rather than overwriting next on the back half, but that would technically use O(N) extra space, just as if we'd created an external array of node values.)

The first challenge then becomes finding the middle of the linked list in order to start our reversing process there. For that, we can look to Floyd's Cycle Detection Algorithm.

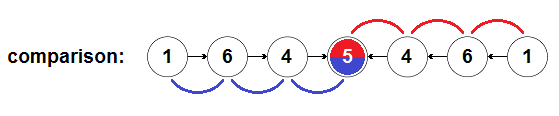
With Floyd's, we'll travel through the linked list with two pointers, one of which is moving twice as fast as the other. When the fast pointer reaches the end of the list, the slow pointer must then be in the middle.



With slow now at the middle, we can reverse the back half of the list with the help of another variable to contain a reference to the previous node (prev) and a three-way swap. Before we do this, however, we'll want to set prev.next = null, so that we break the reverse cycle and avoid an endless loop.

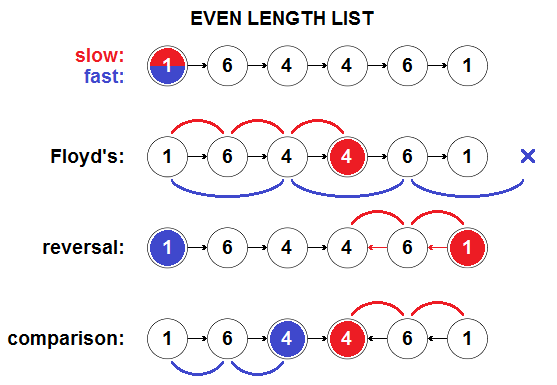


Once the back half is properly reversed andslow is once again at the end of the list, we can now start fast back over again at the head and compare the two halves simultaneously, with no extra space required.



If the two pointers ever disagree in value, we canreturn false, otherwise we can return true if both pointers reach the middle successfully.

(Note: This process works regardless of whether the length of the linked list is odd or even, as the comparison will stop when slow reaches the "dead-end" node.)



class Solution {

public boolean isPalindrome(ListNode head) {

ListNode slow = head, fast = head, prev, temp;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

}

prev = slow;

slow = slow.next;

prev.next = null;

while (slow != null) {

temp = slow.next;

slow.next = prev;

prev = slow;

slow = temp;

}

fast = head;

slow = prev;

while (slow != null) {

if (fast.val != slow.val) return false;

fast = fast.next;

slow = slow.next;

}

return true;

}

}