

Function to check if a singly linked list is palindrome

Given a singly linked list of characters, write a function that returns true if the given list is palindrome, else false.



METHOD 1 (Use a Stack)

A simple solution is to use a stack of list nodes. This mainly involves three steps.

- 1) Traverse the given list from head to tail and push every visited node to stack.
- 2) Traverse the list again. For every visited node, pop a node from stack and compare data of popped node with currently visited node.
- 3) If all nodes matched, then return true, else false.

Time complexity of above method is $O(n)$, but it requires $O(n)$ extra space. Following methods solve this with constant extra space.

METHOD 2 (By reversing the list)

This method takes $O(n)$ time and $O(1)$ extra space.

- 1) Get the middle of the linked list.
- 2) Reverse the second half of the linked list.
- 3) Check if the first half and second half are identical.
- 4) Construct the original linked list by reversing the second half again and attaching it back to the first half

To divide the list in two halves, method 2 of [this](#) post is used.

When number of nodes are even, the first and second half contain exactly half nodes. The challenging thing in this method is to handle the case when number of nodes are odd. We don't want the middle node as part of any of the lists as we are going to compare them for equality. For odd case, we use a separate variable

```

'midnode'.

/* Program to check if a linked list is palindrome */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Link list node */
struct node
{
    char data;
    struct node* next;
};

void reverse(struct node**);
bool compareLists(struct node*, struct node *);

/* Function to check if given linked list is
palindrome or not */
bool isPalindrome(struct node *head)
{
    struct node *slow_ptr = head, *fast_ptr = head;
    struct node *second_half, *prev_of_slow_ptr = head;
    struct node *midnode = NULL; // To handle odd size list
    bool res = true; // initialize result

    if (head!=NULL && head->next!=NULL)
    {
        /* Get the middle of the list. Move slow_ptr by 1
        and fast_ptr by 2, slow_ptr will have the middle
        node */
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;

            /*We need previous of the slow_ptr for
            linked lists with odd elements */
            prev_of_slow_ptr = slow_ptr;
            slow_ptr = slow_ptr->next;
        }

        /* fast_ptr would become NULL when there are even elements in list.
        And not NULL for odd elements. We need to skip the middle node
        for odd case and store it somewhere so that we can restore the
        original list*/
        if (fast_ptr != NULL)
        {
            midnode = slow_ptr;
            slow_ptr = slow_ptr->next;
        }

        // Now reverse the second half and compare it with first half
        second_half = slow_ptr;
        prev_of_slow_ptr->next = NULL; // NULL terminate first half
        reverse(&second_half); // Reverse the second half
        res = compareLists(head, second_half); // compare

        /* Construct the original list back */
        reverse(&second_half); // Reverse the second half again
        if (midnode != NULL) // If there was a mid node (odd size case) which
            // was not part of either first half or second half.
        {
            prev_of_slow_ptr->next = midnode;
            midnode->next = second_half;
        }
    }
}

```

```

    }
    else prev_of_slow_ptr->next = second_half;
}
return res;
}

/* Function to reverse the linked list Note that this
function may change the head */
void reverse(struct node** head_ref)
{
    struct node* prev = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to check if two input lists have same data*/
bool compareLists(struct node* head1, struct node *head2)
{
    struct node* temp1 = head1;
    struct node* temp2 = head2;

    while (temp1 && temp2)
    {
        if (temp1->data == temp2->data)
        {
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        else return 0;
    }

    /* Both are empty return 1*/
    if (temp1 == NULL && temp2 == NULL)
        return 1;

    /* Will reach here when one is NULL
    and other is not */
    return 0;
}

/* Push a node to linked list. Note that this function
changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

```

```
// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%c->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    char str[] = "abacaba";
    int i;

    for (i = 0; str[i] != '\0'; i++)
    {
        push(&head, str[i]);
        printList(head);
        isPalindrome(head)? printf("Is Palindrome\n\n"):
                           printf("Not Palindrome\n\n");
    }

    return 0;
}
```

[Run on IDE](#)

Output:

```
a->NULL
Palindrome

b->a->NULL
Not Palindrome

a->b->a->NULL
Is Palindrome

c->a->b->a->NULL
Not Palindrome

a->c->a->b->a->NULL
Not Palindrome

b->a->c->a->b->a->NULL
Not Palindrome

a->b->a->c->a->b->a->NULL
Is Palindrome
```

Time Complexity $O(n)$

Auxiliary Space: $O(1)$

METHOD 3 (Using Recursion)

Use two pointers left and right. Move right and left using recursion and check for following in each recursive call.

- 1) Sub-list is palindrome.
- 2) Value at current left and right are matching.

If both above conditions are true then return true.

The idea is to use function call stack as container. Recursively traverse till the end of list. When we return from last NULL, we will be at last node. The last node to be compared with first node of list.

In order to access first node of list, we need list head to be available in the last call of recursion. Hence we pass head also to the recursive function. If they both match we need to compare (2, n-2) nodes. Again when recursion falls back to (n-2)nd node, we need reference to 2nd node from head. We advance the head pointer in previous call, to refer to next node in the list.

However, the trick in identifying double pointer. Passing single pointer is as good as pass-by-value, and we will pass the same pointer again and again. We need to pass the address of head pointer for reflecting the changes in parent recursive calls.

Thanks to [Sharad Chandra](#) for suggesting this approach.

```
// Recursive program to check if a given linked list is palindrome
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* Link list node */
struct node
{
    char data;
    struct node* next;
};

// Initial parameters to this function are &head and head
bool isPalindromeUtil(struct node **left, struct node *right)
{
    /* stop recursion when right becomes NULL */
    if (right == NULL)
        return true;

    /* If sub-list is not palindrome then no need to
       check for current left and right, return false */
    bool isp = isPalindromeUtil(left, right->next);
    if (isp == false)
        return false;

    /* Check values at current left and right */
    bool isp1 = (right->data == (*left)->data);
```

```
/* Move left to next node */
*left = (*left)->next;

return isp1;
}

// A wrapper over isPalindromeUtil()
bool isPalindrome(struct node *head)
{
    isPalindromeUtil(&head, head);
}

/* Push a node to linked list. Note that this function
changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%c->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    char str[] = "abacaba";
    int i;

    for (i = 0; str[i] != '\0'; i++)
    {
        push(&head, str[i]);
        printList(head);
        isPalindrome(head)? printf("Is Palindrome\n\n"):
            printf("Not Palindrome\n\n");
    }

    return 0;
}
```

[Run on IDE](#)

Output:

```
a->NULL
Not Palindrome

b->a->NULL
Not Palindrome

a->b->a->NULL
Is Palindrome

c->a->b->a->NULL
Not Palindrome

a->c->a->b->a->NULL
Not Palindrome

b->a->c->a->b->a->NULL
Not Palindrome

a->b->a->c->a->b->a->NULL
Is Palindrome
```

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$ if Function Call Stack size is considered, otherwise $O(1)$.

Please comment if you find any bug in the programs/algorithms or a better way to do the same.



The advertisement features a blue header with a magnifying glass icon over the text 'T-SQL'. Below this, a green banner reads 'A Free Course From Microsoft'. The main title 'Querying with Transact-SQL' is displayed in a large, bold font. To the left of the title is the 'edX' logo with 'edx.org' underneath. To the right is a green 'Enroll now' button with 'Self-Paced' written below it. The background of the top section shows two men in a server room.

238 Comments Category: [Linked Lists](#)

Related Posts:

- [Merge two sorted linked lists such that merged list is in reverse order](#)
- [Compare two strings represented as linked lists](#)

- [Rearrange a given linked list in-place.](#)
- [Sort a linked list that is sorted alternating ascending and descending orders?](#)
- [Select a Random Node from a Singly Linked List](#)
- [Merge Sort for Doubly Linked List](#)
- [Point to next higher value node in a linked list with an arbitrary pointer](#)
- [Swap nodes in a linked list without swapping data](#)

([Login](#) to Rate and Mark)

3.3

Average Difficulty : **3.3/5.0**
Based on **12** vote(s)

☐

Add to TODO List

☐

Mark as DONE

[Like](#) [Share](#) 12 people like this. Be the first of your friends.

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

[@geeksforgeeks](#), [Some rights reserved](#)

[Contact Us!](#)

[About Us!](#)

[Advertise with us!](#)