



Solution Review: Find Middle Node of a Linked List

This review provides a detailed analysis of the different ways to solve the Find the Middle Value in a Linked List challenge.

We'll cover the following



- Solution #1: Brute Force Method
 - Time Complexity
- Solution #2: Two Pointers
 - Time Complexity

Solution #1: Brute Force Method

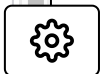
main.py

LinkedList.py

Node.py

```
1 from LinkedList import LinkedList
2 from Node import Node
3 # Access HeadNode => list.getHead()
4 # Check length => list.length()
5 # Check if list is empty => list.isEmpty()
6 # Node class { int data ; Node nextElement;}
7
8
9 def find_mid(lst):
10     if lst.is_empty():
```

```
11         return None
12
13     node = lst.get_head()
14     mid = 0
15     if lst.length() % 2 == 0:
16         mid = lst.length()//2
17     else:
18         mid = lst.length()//2 + 1
19
20     for i in range(mid - 1):
21         node = node.next_element
22
23     return node.data
24
25
26 lst = LinkedList()
27 lst.insert_at_head(22)
28 lst.insert_at_head(21)
```



This is the simplest way to go about this problem. We traverse the whole list to find its length. The middle position can be calculated by halving the length.

Note: For odd lengths, the middle value would be,

```
mid = length/2 + 1
```

Then, we iterate till the middle index and return the value of that node.

Time Complexity

The algorithm makes a linear traversal over the list. Hence, the time complexity is $O(n)$.

Solution #2: Two Pointers



main.py

LinkedList.py

Node.py

```
from LinkedList import LinkedList
from Node import Node
def find_mid(lst):
    if lst.is_empty():
        return -1
    current_node = lst.get_head()
    if current_node.next_element == None:
        #Only 1 element exist in array so return its value.
        return current_node.data

    mid_node = current_node
    current_node = current_node.next_element.next_element
    #Move mid_node (Slower) one step at a time
    #Move current_node (Faster) two steps at a time
    #When current_node reaches at end, mid_node will be at the middle of List
    while current_node:
        mid_node = mid_node.next_element
        current_node = current_node.next_element
        if current_node:
            current_node = current_node.next_element
    if mid_node:
        return mid_node.data
    return -1

lst = LinkedList()
lst.insert_at_head(22)
lst.insert_at_head(21)
lst.insert_at_head(10)
lst.insert_at_head(14)
lst.insert_at_head(7)

lst.print_list()
print(find_mid(lst))
```



This solution is more efficient as compared to the brute force method. We will use two pointers which will work simultaneously.



Think of it this way:



- The **fast** pointer moves two steps at a time till the end of the list
- The **slow** pointer moves one step at a time
- when the **fast** pointer reaches the end, the **slow** pointer will be at the middle

Using this algorithm, we can make the process faster because the calculation of the length and the traversal till the middle are happening side-by-side.

Time Complexity

We are traversing the linked list at twice the speed, so it is certainly faster. However, the bottleneck complexity is still $O(n)$.

The linked lists we have seen so far had unique values, but what if a list contains duplicates? We'll learn more about this in the next lesson.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. [See how](#) ⓘ



← Back

Next →

Challenge 7: Find Middle Node of Link...

Challenge 8: Remove Duplicates from ...

✓ Completed



Report an Issue



