Solution Review: Return the Nth Node from End

This review provides a detailed analysis of the different ways to return the nth node from the end of a linked list

We'll cover the following

Solution #1: Double Iteration

main.py

LinkedList.py

Node.py

```python
from LinkedList import LinkedList
from Node import Node


def find_nth(lst, n):
    if (lst.is_empty()):
        return -1

    # Find Length of list
    length = lst.length() - 1

    # Find the Node which is at (len - n + 1) position
    current_node = lst.get_head()

    position = length - n + 1

    if position < 0 or position > length:
        return -1

    count = 0

    while count is not position:
        current_node = current_node.next_element
        count += 1

    if current_node:
        return current_node.data
    return -1
```

In this approach, our main goal is to figure out the index of the node we need to reach. The algorithm follows these simple steps:

1. Calculate the length of the linked list
2. Check if `N` is within the length
3. Find the position of the node using `length - n + 1` (We start from the last node since we can't start from `None`)
4. Iterate over to the node and return its value

Time Complexity#

It performs two iterations on the list so the complexity is $O(n)$.

Solution #2: Two Pointers#

main.py

LinkedList.py

Node.py

```
from LinkedList import LinkedList
from Node import Node


def find_nth(lst, n):

    if lst.is_empty():
        return -1

    nth_node = lst.get_head()  # This iterator will reach the Nth node
    end_node = lst.get_head()  # This iterator will reach the end of the list

    count = 0
    while count < n:
        if end_node is None:
            return -1
        end_node = end_node.next_element
        count += 1

    while end_node is not None:
        end_node = end_node.next_element
        nth_node = nth_node.next_element

    return nth_node.data


lst = LinkedList()
lst.insert_at_head(21)
lst.insert_at_head(14)
lst.insert_at_head(7)
lst.insert_at_head(8)
lst.insert_at_head(22)
lst.insert_at_head(15)

lst.print_list()

print(find_nth(lst, 19))
print(find_nth(lst, 5))
```

This is the more efficient approach, although it is not an unfa r
Here's the flow of the algorithm:

1. Move `end_node` forward `n` times, while `nth_node` stays at the `head`
2. If `end_node` becomes `None`, `n` was out of bounds of the array. Return `-1` to indicate that the node is not found.
3. Once `end_node` is at **nth** position from the start, move both `end_node` and `nth_node` pointers simultaneously.
4. When `end_node` reaches the end, `nth_node` is at the Nth position from the end

5. Return the node's value

This algorithm also works in *O(n)* time complexity, but it still adopts the policy of one iteration over the whole list. We do not need to keep track of the length of the list.

Time Complexity#

A single iteration is performed, which means that time complexity is *O(n)*.

And there you have it, you've passed all the coding challenges for linked lists. Congratulations! The next section will deal with stacks and queues, two very useful data structures. Before that, try your hand at the quiz in the next lesson. It'll be a good way to reinforce your concepts on linked lists.

Back

Challenge 10: Return the Nth node fro...

Linked Lists Quiz: Test your understan...

✓ Completed

Report an Issue