Linked List Cycles

Learn how to find cycles in a Linked List. We'll cover two different methods.

Linked List Cycles

Instructions

Describe a function that can detect the presence of loops in a Linked List. Rather than writing an actual function, describe the strategy you would employ.

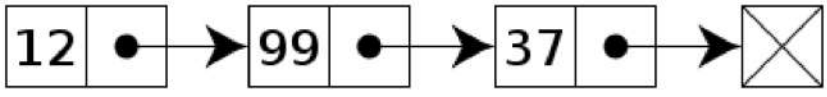
Input: Linked List

Output: Boolean

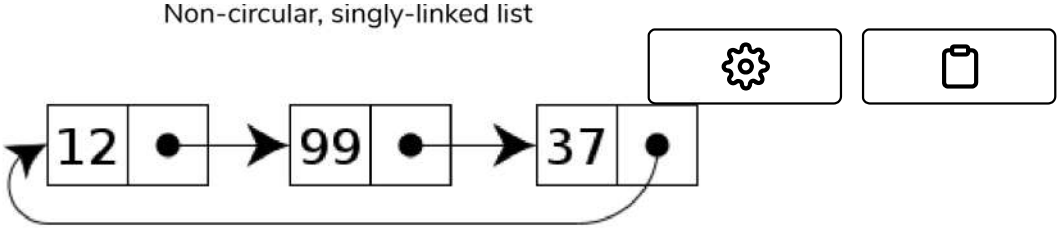
Hints

- From JS Data Structures: Linked List (https://codeburst.io/js-data-structures-linked-list-3ed4d63e6571):

> A Linked List is an ordered, linear structure, similar to an array. Instead of items being placed at indices, however, they are connected through a chain of references, with each item containing a reference to the next item.



Non-circular, singly-linked list



Circular linked list

Solution 1

```
function(list) {
    define storage object to contain nodes

    while next node exists in list
        if node present in storage
            return true
        else
            insert node into storage

    return false
}
```

How it Works

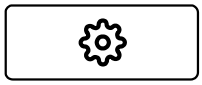
We can keep track of every single node we come across by throwing it into an object. As we go through each node in the linked list, we check our object for its presence. If present, we know we've come across it before, indicating a circular list.

Time

The time complexity of this would be:

O(n),

as we have to process potentially every node in the list.



Space

The space complexity is also:

O(n),

as we have to store potentially every value in the list.

Challenge

Make this function have `O(1)` space complexity.

Solution 2

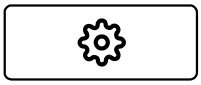
```
function(list) {
    define counter1 = 1st node,
        counter2 = counter1.next

    while true
        if (
            counter1.next == null or
            counter2.next == null or
            counter2.next.next == null
        )
            return false

        counter1 = counter1.next
        counter2 = counter2.next.next

        if counter1 == counter2
            return true
}
```

How it Works



We create two counters which both start at the beginning of the list. They advance forward. The first advances one node at a time and the second advances two nodes at a time.

If there is a null node anywhere, signifying the end of the list, we can return `false`. Otherwise, the two counters will eventually fall on the same node. If the fast and slow pointer ever converge, this verifies a circle in the list, allowing us to return `true`.

Time

The time complexity is:

O(n).

If the list isn't circular, we simply wait for the fast pointer to get to the end. It has to traverse half the nodes, so time complexity is `O(n)`.

If the list is circular, the slow and fast pointer will always find each other by the time the slow pointer has reached the end of the list.

Space

O(1),

since we don't store any of the nodes.

Conclusion

This solution is rather tricky and requires imaginative thinking. It has less to do with code and more with general problem-solving and reasoning.

The idea of using two pointers is extendable to other problems as well. Occasionally, going towards the center of an array from both ends will yield a better time complexity than going from start to end. The idea of two pointers is another tool to keep in mind during traversal problems.

← Back

Creating a Queue with Limited Data S...

Next →

Deep Equals

Mark as Completed

Report an Issue