Solution Review: Detect Loop in a Linked List

This review provides an analysis of the solution for the Detect a Loop in a Linked List challenge.

We'll cover the following

Solution: Using a Set

main.py

LinkedList.py

Node.py

```
from LinkedList import LinkedList
from Node import Node


def detect_loop(lst):
    # Used to store nodes which we already visited
    visited_nodes = set()
    current_node = lst.get_head()

    # Traverse the set and put each node in the visitedNodes set
    # and if a node appears twice in the map
    # then it means there is a loop in the set
    while current_node:
        if current_node in visited_nodes:
            return True
        visited_nodes.add(current_node)  # Insert node in visitedNode
        current_node = current_node.next_element
    return False

# ------------------------------


lst = LinkedList()

lst.insert_at_head(21)
lst.insert_at_head(14)
lst.insert_at_head(7)
print(detect_loop(lst))
```

This is the primitive approach, but it works nonetheless.

We iterate over the whole linked list and add each visited node to a `visited_nodes` **set**. At every node, we check whether it has been visited or not.

By principle, if a node is revisited, a cycle exists!

Time Complexity

We iterate the list once. On average, lookup in a set takes *O(1)* time. Hence, the average runtime of this algorithm is *O(n)*. However, in the worst case, lookup can increase up to *O(n)*, which would cause the algorithm to work in $O(n^2)$.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. See how

← Back

Challenge 10: Detect Loop in a Linked ...

Challenge 11: Remove Duplicates fro...

☑ Mark as Completed

Report an Issue