Assignment 8 (Linked List)

> **Name: Mohammed Varaliya**

> **Roll No: 54**

Questions

1. **Code of Recursive function for displaying/traversing all nodes of a link list.**
2. **Time & Space Complexity of above program.**

---

1. **Code of Recursive function for displaying/traversing all nodes of a link list.**
    a. **Code:**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return
```

```
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def prepend(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        cur_node = self.head
        self.head = new_node
        new_node.next = cur_node

    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data, end=" -> ")
            cur_node = cur_node.next
        print("None")

    def print_list_recursive(self, node):
        if node is None:
            return "None"
        else:
            return str(node.data) + " -> " + self.print_list_rec
ursive(node.next)


if __name__ == "__main__":

    llist = SinglyLinkedList()
    llist.append(1)
    llist.append(2)
    llist.append(3)
    llist.append(4)

    llist.print_list()

    print(llist.print_list_recursive(llist.head))
```

```
# Main logic of Recursive function for displaying/traversing all
nodes of a linked list in above code is.

def print_list_recursive(self, node):
    if node is None:
        return "None"
    else:
        return str(node.data) + " -> " + self.print_list_recursi
ve(node.next)
```

1. **Explanation:**
    a. **Purpose**: The program demonstrates how to traverse and display all nodes of a singly linked list using both iterative and recursive methods.
    b. **Logic**:
        i. The `print_list` method iteratively traverses the list, printing each node's `data` followed by `" -> "`, and ends with `"None"` to indicate the list's end.
        ii. The `print_list_recursive` method recursively traverses the list. If the current node is `None` (end of the list), it returns the string `"None"`. Otherwise, it concatenates the current node's data with the result of the recursive call on the next node.
    c. **Output**:
        i. For a list `1 -> 2 -> 3 -> 4`, both `print_list` and `print_list_recursive` output `1 -> 2 -> 3 -> 4 -> None`, demonstrating both traversal methods.
    d. **Example**:
        i. When running the program, the linked list `[1, 2, 3, 4]` is created. The `print_list` method prints `1 -> 2 -> 3 -> 4 -> None` iteratively, and the `print_list_recursive` method prints the same result recursively.

b. **Output:**

```
1 -> 2 -> 3 -> 4 -> None
1 -> 2 -> 3 -> 4 -> None
```

---

2. **Time & Space Complexity of above program.**
    a. **Time Complexity:**
        i. The time complexity of the recursive `print_list_recursive` function is determined by the number of nodes in the linked list.
        ii. **Recursive Call Stack**:

1. The function makes one recursive call for each node in the list. So, if there are `n` nodes in the list, the function will be called `n` times.
2. In each recursive call, the function performs a constant amount of work: checking if `node` is `None`, concatenating the string (`node.data + " -> "`), and calling the next recursive function.

iii. **Overall Time Complexity**:

1. For each node in the list, the function performs constant work (`O(1)`). The function is called once for each node, so the overall time complexity is $O(n)$, where `n` is the number of nodes in the list.

iv. **Steps to determine time complexity**:

1. We have `n` nodes in the list.
    a. `1 -> 2 -> 3 -> 4 -> None`
2. The recursive function is called once for each node in the list.
3. The work done inside the function (concatenating strings, checking the condition) is constant for each call.
4. Therefore, the total time complexity is:

$$\text{Time Complexity} = O(n)$$

b. **Space Complexity:**

i. The space complexity of the recursive `print_list_recursive` function is influenced by two factors:

ii. **The Recursion Stack**:

1. Each recursive call adds a new frame to the call stack. The recursion will proceed for `n` nodes, so the call stack will contain `n` frames at its deepest point.
2. This means the space required for the recursion stack is $O(n)$.

iii. **Auxiliary Space for Storing the Result**:

1. The function is building a string to represent the list. In the worst case, the string will have a size proportional to the number of nodes (since each node's data is added to the string, along with the `" -> "` separators).
2. Therefore, the auxiliary space required for building the string is also $O(n)$.

iv. **Steps to determine space complexity**:

1. For each recursive call, we use constant space (`O(1)`).
2. However, the total space required will be determined by the maximum depth of the recursion stack, which is `n`.
3. Additionally, we are constructing a string of size `O(n)`.

v. **Thus, the total space complexity is:**

$$\text{Space Complexity} = O(n) \text{ (for recursion stack)} + O(n) \text{ (for the string result)}$$

$$= O(n)$$
