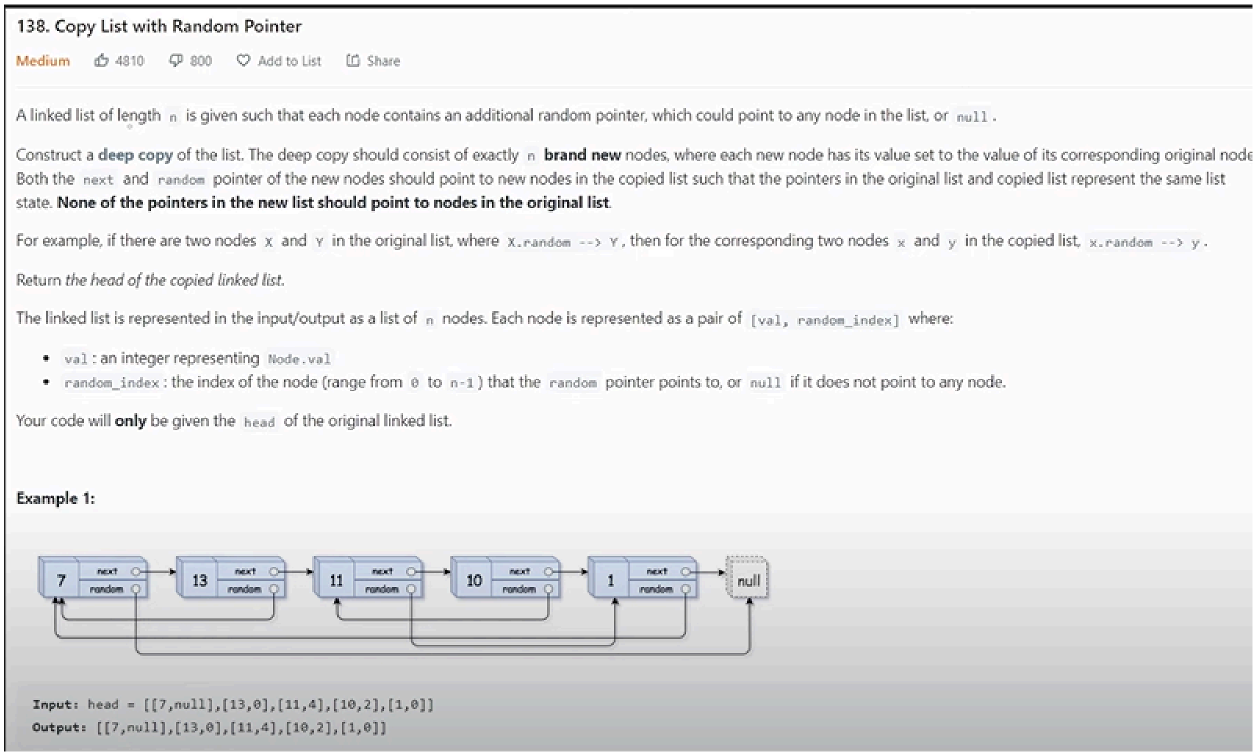
138. Copy List with Random Pointer

Problem Statement :



138. Copy List with Random Pointer

Medium 4810 800 Add to List Share

A linked list of length `n` is given such that each node contains an additional random pointer, which could point to any node in the list, or `null`.

Construct a **deep copy** of the list. The deep copy should consist of exactly `n` **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the `next` and `random` pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list.**

For example, if there are two nodes `X` and `Y` in the original list, where `X.random --> Y`, then for the corresponding two nodes `x` and `y` in the copied list, `x.random --> y`.

Return *the head of the copied linked list.*

The linked list is represented in the input/output as a list of `n` nodes. Each node is represented as a pair of `[val, random_index]` where:

- `val` : an integer representing `Node.val`
- `random_index` : the index of the node (range from `0` to `n-1`) that the `random` pointer points to, or `null` if it does not point to any node.

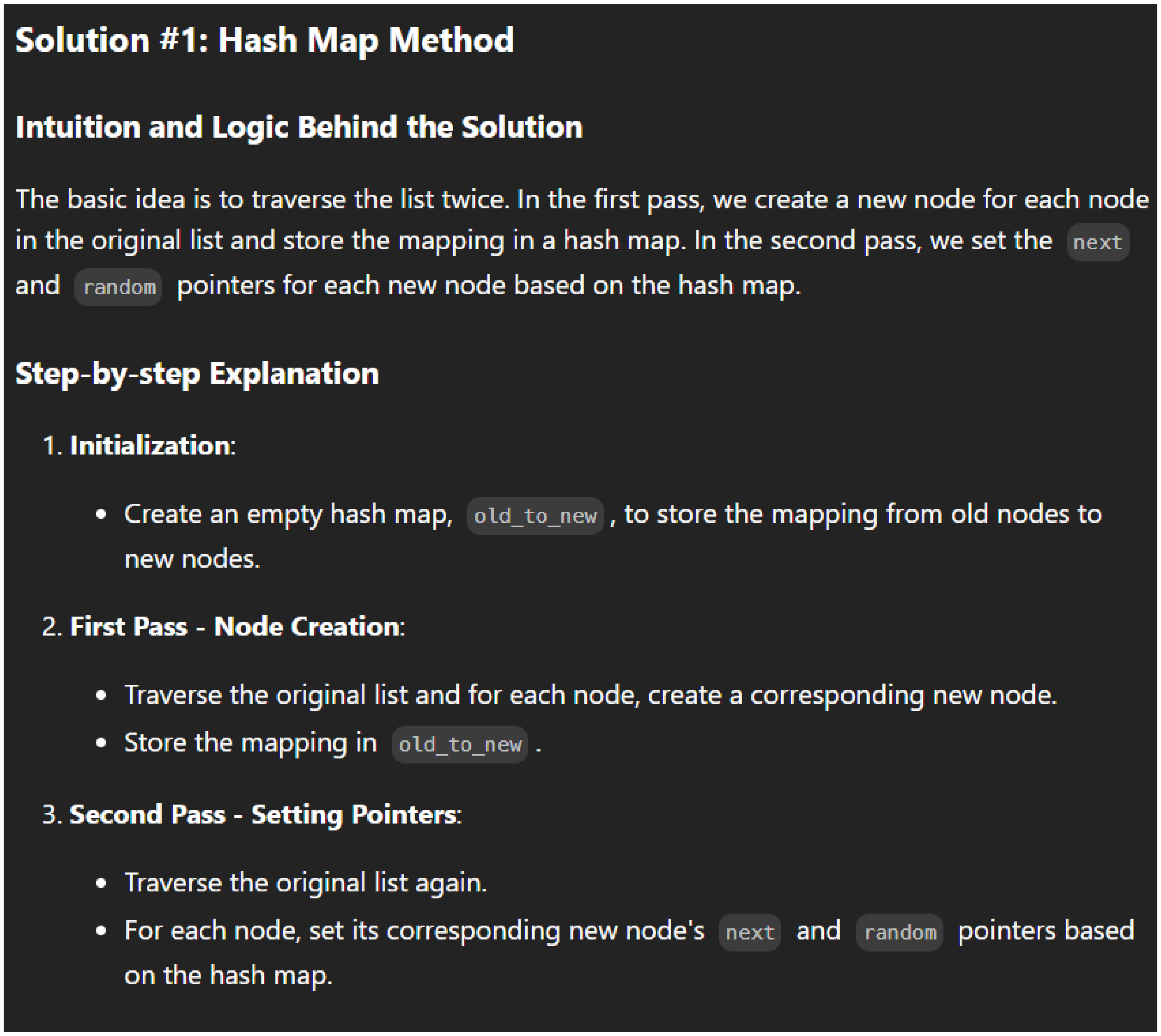
Your code will **only** be given the `head` of the original linked list.

Example 1:

```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

Solution:

1. Here we'll be making two passess in our list…where we create a hashmap..and can easily solve the problem

1.


Solution #1: Hash Map Method

Intuition and Logic Behind the Solution

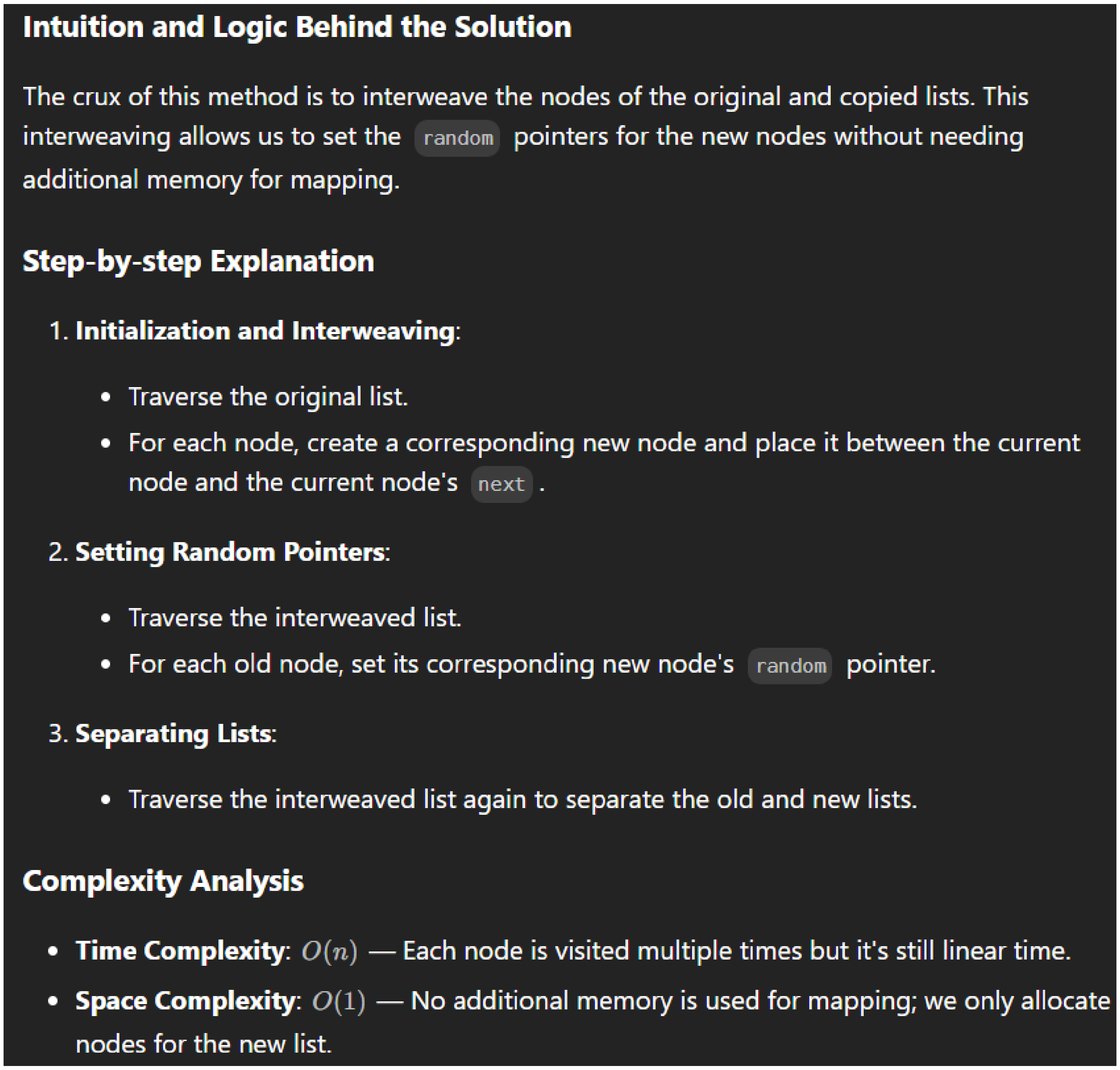
The basic idea is to traverse the list twice. In the first pass, we create a new node for each node in the original list and store the mapping in a hash map. In the second pass, we set the `next` and `random` pointers for each new node based on the hash map.

Step-by-step Explanation

1. **Initialization**:
   - Create an empty hash map, `old_to_new`, to store the mapping from old nodes to new nodes.
2. **First Pass - Node Creation**:
   - Traverse the original list and for each node, create a corresponding new node.
   - Store the mapping in `old_to_new`.
3. **Second Pass - Setting Pointers**:
   - Traverse the original list again.
   - For each node, set its corresponding new node's `next` and `random` pointers based on the hash map.

2.
3. Time complexity O(n) and Space O(n)

Solution 2 using Interweaving Nodes Method

1. 

Intuition and Logic Behind the Solution

The crux of this method is to interweave the nodes of the original and copied lists. This interweaving allows us to set the `random` pointers for the new nodes without needing additional memory for mapping.

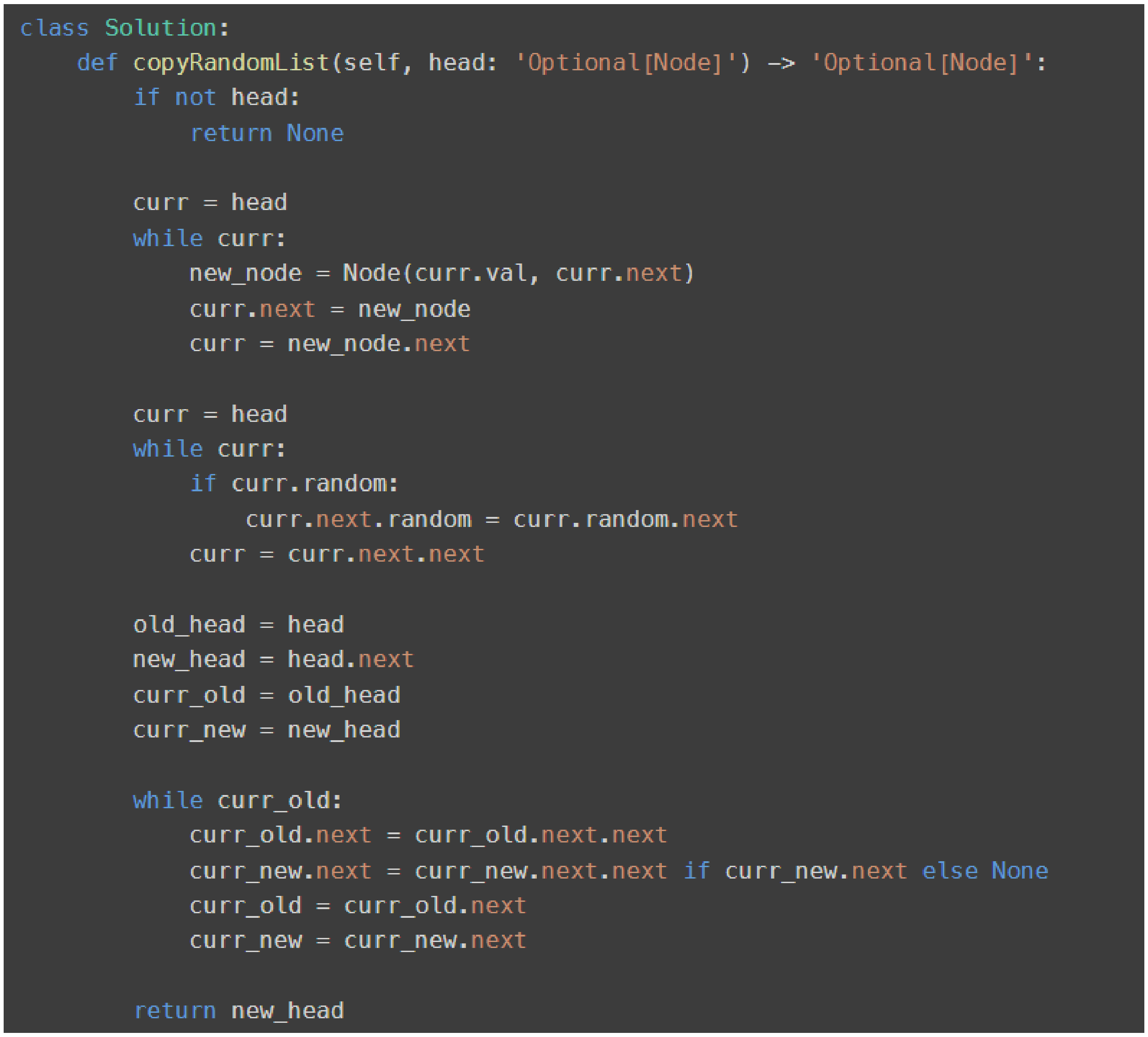
Step-by-step Explanation

1. **Initialization and Interweaving**:
    - Traverse the original list.
    - For each node, create a corresponding new node and place it between the current node and the current node's `next` .
2. **Setting Random Pointers**:
    - Traverse the interweaved list.
    - For each old node, set its corresponding new node's `random` pointer.
3. **Separating Lists**:
    - Traverse the interweaved list again to separate the old and new lists.

Complexity Analysis

- **Time Complexity**: $O(n)$ — Each node is visited multiple times but it's still linear time.
- **Space Complexity**: $O(1)$ — No additional memory is used for mapping; we only allocate nodes for the new list.

2. Code :



```python
class Solution:
    def copyRandomList(self, head: 'Optional[Node]') -> 'Optional[Node]':
        if not head:
            return None

        curr = head
        while curr:
            new_node = Node(curr.val, curr.next)
            curr.next = new_node
            curr = new_node.next

        curr = head
        while curr:
            if curr.random:
                curr.next.random = curr.random.next
            curr = curr.next.next

        old_head = head
        new_head = head.next
        curr_old = old_head
        curr_new = new_head

        while curr_old:
            curr_old.next = curr_old.next.next
            curr_new.next = curr_new.next.next if curr_new.next else None
            curr_old = curr_old.next
            curr_new = curr_new.next

        return new_head
```