138. Copy List with Random Pointer

Medium Hash Table Linked List

Leetcode Link

Problem Description

The problem provides a special type of linked list where each node has two pointers: a `next` pointer to the next node in the sequence and a `random` pointer that can point to any node in the list or be `null`. The task is to create a deep copy of this linked list. A deep copy means that you need to create a new list where each node in the new list corresponds to a node in the original list, but no pointers in the new list should reference nodes from the original list. The newly created list should maintain the original structure, including the `next` and `random` linkages. The input is the head of the original linked list, and the output should be the head of the newly copied linked list.

Intuition

To solve this problem, we need to create a copy of each node and also map the random pointers from the original list to the corresponding nodes in the copied list. This can become complex because the random pointer can point to any node, creating a non-linear structure.

The solution follows these steps:

1. Iterate through the original list and create a copy of each node, insert the copied node directly after its original node. This links the original and the copied nodes in a single mixed list.
2. Go through the mixed list again and set the `random` pointers for the copied nodes. Since we've interleaved the copied nodes with their corresponding originals, each original node's `next` is now its copy. This makes it easy to find and set the copied node's `random` pointer: if `originalNode.random` exists, then `copiedNode.random` will be `originalNode.random.next`.
3. Finally, we need to restore the original list and separate the copied list from it. We iterate through the mixed list, re-establish the original `next` links for the original nodes, and create the `next` links for the copied nodes.

Following this approach ensures that we create a deep copy of the list without needing extra space for a map to track the `random` pointer relationships which is a common approach for such problems.

Solution Approach

The solution for creating a deep copy of the linked list with a `random` pointer is implemented in three major steps:

Step 1: Interleaving the Original and Copied Nodes

We start by iterating through the original list. For each node in the list, we perform the following actions:

- Create a new node with the same value as the current node (`cur.val`).
- Insert this new node right after the current one by setting the `next` pointer of the new node to point to the node that `cur.next` was originally pointing to.
- Update `cur.next` to point to the newly created node.
- Move forward in the list by setting `cur` to the next original node (`cur = cur.next.next`).

This step essentially weaves the original and new nodes together.

Step 2: Setting the `random` Pointers for Copied Nodes

In the second pass over this interleaved list, we set the `random` pointer for each copied node. If the `random` pointer of the original node is not null, we can find the corresponding `random` pointer for the copied node as follows:

- Given an original node `cur`, its copied node is `cur.next`.
- The random node of `cur` is `cur.random`.
- Therefore, the corresponding copied random node would be `cur.random.next`.

We set this with `cur.next.random = cur.random.next`.

Step 3: Separating the Copied List from the Original List

Finally, we restore the original list to its initial state and extract the deep copy. We iterate over the interleaved list and do the following actions:

- Set `ret` to `cur.next`, which is the copied node following the original `cur` node.
- Re-assign `cur.next` to `ret.next`, which is the next original node in the list (or `null` if we are at the end).
- Then, we update `cur` to point to the copied node's next node, which should also be a copied node (or `null`), using `cur = ret`.
- Continue this process until all nodes have been visited.

By the end of step 3, we will have the original list restored and a separately linked new list, which is a deep copy of the original list.

This approach is efficient as it does not require extra space for maintaining a hash table to map the original to copied nodes, and it completes the copying in linear time.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following linked list where the `next` pointers form a single sequence and the `random` pointers point to various nodes, including `null`.

```
1. Original List:
1 -> 2 -> 3 -> null
|    |    |
v    v    v
2    null 1
```

Node 1 has a `random` pointer to Node 2, Node 2's `random` pointer is `null`, and Node 3's `random` pointer points back to Node 1.

Step 1: Interleaving the Original and Copied Nodes

We create a copy of each node and interleave it with the original list.

```
1. After interleaving:
1 -> 1' -> 2 -> 2' -> 3 -> 3' -> null
|     |     |     |     |
v     v     v     v     v
2    null   1'    null  2'
```

Here, 1', 2', and 3' are the copied nodes of 1, 2, and 3 respectively. Each original node's next pointer now points to its newly created copy.

Step 2: Setting the `random` Pointers for Copied Nodes

Now, we set up the `random` pointers for the copied nodes.

```
1. Interleaved list with 'random' pointers set:
1 -> 1' -> 2 -> 2' -> 3 -> 3' -> null
|     |     |     |     |
v     v     v     v     v
2    null   1'    null  2'
```

Copied node 1's `random` pointer goes to 2', 2''s `random` pointer remains `null`, and copied node 3''s `random` pointer goes to 1'.

Step 3: Separating the Copied List from the Original List

Finally, we untangle the two lists into the original list and its deep copy.

```
1. Restored original list:
1 -> 2 -> 3 -> null
|    |    |
v    v    v
2    1    1'
2. Separated copied list (deep copy):
1' -> 2' -> 3' -> null
|     |     |
v     v     v
2'    null  1'
```

We're left with two separate lists: the original list is unchanged, and we have a new deep copy list which maintains the original structure, including the `random` links.

This process creates a deep copy of a complex linked list with `random` pointers in O(n) time and O(1) additional space.

Python Solution

```python
class Node:
    def __init__(self, val: int, next: 'Node' = None, random: 'Node' = None):
        self.val = val
        self.next = next
        self.random = random

class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        # If the original list is empty, return None.
        if not head:
            return None

        # Step 1: Create new nodes weaved within the original list
        current = head
        while current:
            # clone_node is a copy of the current node without the random reference
            clone_node = Node(current.val, current.next)
            current.next = clone_node  # Insert the cloned node just after the current node
            current = clone_node.next  # Move to the next original node

        # Step 2: Assign random pointers to the cloned nodes
        current = head
        while current:
            if current.random:
                # Set the cloned node's random to the cloned node of the original node's random
                current.next.random = current.random.next
            current = current.next.next  # Move two steps forward

        # Step 3: Detach the original and cloned list from each other
        original_current = head
        cloned_head = head.next
        while original_current:
            original_current.next = original_current.next.next  # Fix the original list's next pointer
            # Fix the cloned list's next pointer, only if cloned_current has a next node
            if cloned_current.next:
                cloned_current.next = cloned_current.next.next
            # Move to the next original node
            original_current = original_current.next

        # Return the head of the cloned linked list
        return cloned_head
```

Java Solution

```java
class Solution {
    public Node copyRandomList(Node head) {
        // Return null if the original list is empty
        if (head == null) {
            return null;
        }

        // 1: Create a cloned node for each node in the original list and insert it
        // right next to the original node.
        for (Node current = head; current != null; ) {
            Node clone = new Node(current.val); // clone node
            clone.next = current.next; // Set clone's next to current node's next
            current.next = clone; // Insert cloned node after the current node
            current = clone.next; // Move to the next original node
        }

        // 2: Assign random pointers for the cloned nodes.
        for (Node current = head; current != null;) {
            if (current.random != null) {
                // Set the cloned node's random to the cloned node of the original node's random
                current.next.random = current.random.next;
            }
            current = current.next.next;
        }

        // 3: Restore the original list and extract the cloned list.
        Node copyHead = head.next; // head of the cloned list
        for (Node current = head; current != null;) {
            Node clone = current.next; // The cloned node
            // Restore the original list's `next` pointers
            current.next = clone.next;
            // Set the cloned node's `next` pointers. Check if the next original node exists.
            clone.next = clone.next != null ? clone.next.next : null;
            // Move to the next original node
            current = current.next;
        }

        // Return the head of the cloned list
        return copyHead;
    }
}

// Definition for a Node.
class Node {
    public int val; // Value of the node
    public Node next; // Reference to the next node in the list
    public Node random; // Reference to a random node in the list

    public Node(int val) {
        this.val = val;
        this.next = null;
        this.random = null;
    }
}
```

C++ Solution

```cpp
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};
*/
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) {
            return nullptr; // If the input list is empty, return nullptr.
        }

        // First pass: Create a new node for each existing node and interleave them.
        for (Node* current = head; current != nullptr;) {
            Node* newNode = new Node(current->val);
            newNode->next = current->next;
            current->next = newNode;
            current = newNode->next;
        }

        // Second pass: Assign random pointers for the new nodes.
        for (Node* current = head; current != nullptr; current = current->next->next) {
            if (current->random != nullptr) {
                current->next->random = current->random->next;
            }
        }

        // Third pass: Separate the original list from the copied list and prepare the return value.
        Node* copiedListHead = head->next;
        for (Node* current = head; current != nullptr;) {
            Node* copiedNode = current->next;
            // Advance the current pointer in the original list to the next original node.
            current->next = copiedNode->next;
            // Update the copiedNode pointer in the copied list to the next copied node,
            // if it exists.
            if (copiedNode->next != nullptr) {
                copiedNode->next = copiedNode->next->next;
            }
            // Move to the next original node
            current = current->next;
        }

        // Return the head of the copied list.
        return copiedListHead;
    }
};
```

Typescript Solution

```typescript
// Definition for Node that represents each node in a linked list.
class Node {
  val: number;
  next: Node | null;
  random: Node | null;
}

/**
 * Creates a deep copy of a linked list where each node has a random pointer.
 *
 * @param {Node | null} head - The head of the linked list to be copied.
 * @return {Node | null} The head of the copied linked list.
 */
function copyRandomList(head: Node | null): Node | null {
  // Initialize a map to keep track of already copied nodes.
  const clonedNodesMap = new Map<Node, Node>();

  // First pass: copy all the nodes and map the original nodes to their copies.
  let currentNode = head;
  while (currentNode !== null) {
    clonedNodesMap.set(currentNode, { val: currentNode.val, next: null, random: null });
    currentNode = currentNode.next;
  }

  // Second pass: Assign the next and random pointers.
  currentNode = head;
  while (currentNode !== null) {
    // Get the cloned node for the current node.
    const clonedNode = clonedNodesMap.get(currentNode)!;
    // Set the next pointer of the cloned node.
    clonedNode.next = currentNode.next ? clonedNodesMap.get(currentNode.next)! : null;
    // Set the random pointer of the cloned node.
    clonedNode.random = currentNode.random ? clonedNodesMap.get(currentNode.random)! : null;
    // Move to the next original node.
    currentNode = currentNode.next;
  }

  // Return the cloned head which is the copy of the original head.
  return head ? clonedNodesMap.get(head)! : null;
}
```

Time and Space Complexity

The given code implements a method to copy a linked list with a `next` and a `random` pointer.

Time Complexity:

The algorithm involves three main steps:

1. Iterating through the original list and creating a copy of each node which is inserted between the original node and its next node: This step takes `O(N)` time, where `N` is the number of nodes in the list.
2. Iterating through the list again to update the random pointers of the copied nodes: For each of the original nodes, the corresponding copied node's random pointer is updated to the copied random node. This step also takes `O(N)` time.
3. Finally, the copied nodes are separated from the original list to form the copied list: As we iterate through the list to reconstruct the original and copied lists, this step also takes `O(N)` time.

Since all steps are performed sequentially, the overall time complexity is `O(N) + O(N) + O(N)`, which simplifies to `O(N)`.

Space Complexity:

The space complexity consists of the additional space required by the algorithm excluding the input and output.

1. The copied nodes are created in between the original nodes, so there is no need for extra space for maintaining relationships or indexes which would otherwise be the case if a hash table were used. Therefore, no extra space proportional to the number of elements in the list is used.
2. The space used by the nodes themselves is not considered in the space complexity analysis as it is part of the output.

Thus, the space complexity is `O(1)` as we're only using a constant amount of extra space, regardless of the input size (ignoring the space required for the output).