138. Copy List with Random Pointer **Linked List** Medium Hash Table

Problem Description

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

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

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

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 solution follows these steps:

non-linear structure.

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

Solution Approach

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.
- The solution for creating a deep copy of the <u>linked list</u> 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

following actions:

1 Original List:

1 -> 2 -> 3 -> null

null 1

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).

In the second pass over this interleaved list, we set the random pointer for each copied node. If the random pointer of the original

Given an original node cur, its copied node is cur.next.

Step 3: Separating the Copied List from the Original List

• The random node of cur is cur, random.

This step essentially weaves the original and new nodes together.

Step 2: Setting the random Pointers for Copied Nodes

node is not null, we can find the corresponding random pointer for the copied node as follows:

• Therefore, the corresponding copied random node would be cur.random.next. We set this with cur.next.random = cur.random.next.

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

• Then, we update cur to point to the copied node's next node, which should also be a copied node (or null), using cur = nxt.

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

 Set nxt to cur.next, which is the copied node following the original cur node. Re-assign cur.next to nxt.next, which is the next original node in the list (or null if we are at the end).

Continue this process until all nodes have been visited.

simple sequence and the random pointers point to various nodes, including null.

completes the copying in linear time.

Step 1: Interleaving the Original and Copied Nodes

Step 2: Setting the random Pointers for Copied Nodes

1 Interleaved list with `random` pointers set:

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

Step 3: Separating the Copied List from the Original List

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

2 1 -> 1' -> 2 -> 2⁻ -> 3 -> 3' -> null

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

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

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.

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.

1 Restored original list:

8 1' -> 2' -> 3' -> null

null

1 class Node:

11

13

14

15

17

18

19

20

27

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

45

46

47

48

49

51

52

54

53 }

1 /*

8

9

10

13

18

20

21

22

23

24

25

26

27

28

29

30

32

33

34

35

36

37

38

40

41

48

49

50

51

52

53

54

55

57

56 };

14 };

15 */

17 public:

C++ Solution

class Node {

16 class Solution {

int val;

Node∗ next;

Node random;*

Node(int _val) {

 $val = _val;$

next = NULL;

if (!head) {

random = NULL;

Node* copyRandomList(Node* head) {

public:

2 // Definition for a Node.

public Node(int val) {

this.val = val;

this.next = null;

this.random = null;

Java Solution

class Solution {

class Solution:

11 2'

7 Separated copied list (deep copy):

self.val = val

if not head:

current = head

while current:

self.next = next

self.random = random

return None

2 1 -> 2 -> 3 -> null

null 1

After interleaving:

null

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'.

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.

-> 1' -> 2 -> 2' -> 3 -> 3' -> null 2' null null 1

Python Solution

def copyRandomList(self, head: 'Node') -> 'Node':

If the original list is empty, return None.

def __init__(self, val: int, next: 'Node' = None, random: 'Node' = None):

Step 1: Create new nodes weaved within the original list

current.next.random = current.random.next

cloned_current = original_current.next

original current = original_current.next

public Node next; // Reference to the next node in the list

public Node random; // Reference to a random node in the list

return nullptr; // If the input list is empty, return nullptr.

for (Node* current = head; current != nullptr;) {

for (Node* current = head; current != nullptr;) {

// Definition for Node that represents each node in a linked list.

// Second pass: Assign random pointers for the new nodes.

copiedNode->next = copiedNode->next->next;

current->next->random = current->random->next;

Node* newNode = new Node(current->val);

newNode->next = current->next;

if (current->random != nullptr) {

Node* copiedNode = current->next;

// Move to the next original node

// Return the head of the copied list.

current = current->next;

return copiedListHead;

Typescript Solution

interface Node {

val: number;

next: Node | null;

current->next = newNode;

current = newNode->next;

Node* copiedListHead = head->next;

// First pass: Create a new node for each existing node and interleave them.

for (Node* current = head; current != nullptr; current = current->next->next) {

// Third pass: Separate the original list from the copied list and prepare the return value.

// Advance the current pointer in the original list to the next original node

Move to the next original node

Return the head of the cloned linked list

Step 3: Detach the original and cloned list from each other

cloned_current.next = cloned_current.next.next

current = current.next.next # Move two steps forward

cloned_node = Node(current.val, current.next)

cloned_node is a copy of the current node without the random reference

current = cloned_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 = cloned_node # Insert the cloned node just after the current node

original_current.next = cloned_current.next # Fix the original list's next pointer

Fix the cloned list's next pointer, only if cloned_current has a next node

23 24 25

original_current = head

cloned_head = head.next

while original_current:

return cloned_head

if cloned_current.next:

```
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; ) {
10
               Node clone = new Node(current.val); // Clone node
11
               clone.next = current.next; // Set clone's next to current node's next
12
               current.next = clone; // Insert cloned node after the current node
14
               current = clone.next; // Move to the next original node
15
16
           // 2. Assign random pointers for the cloned nodes.
           for (Node current = head; current != null; current = current.next.next) {
               if (current.random != null) {
20
                   // Set the cloned node's random to the cloned node of the original node's random
21
                    current.next.random = current.random.next;
22
23
24
25
           // 3. Restore the original list and extract the cloned list.
26
           Node copyHead = head.next; // Head of the cloned list
27
            for (Node current = head; current != null; ) {
28
               Node clone = current.next; // The cloned node
29
               // Restore the original list's 'next' pointers
               current.next = clone.next;
30
               // Set the cloned node's 'next' pointers. Check if the next original node exists.
31
32
               clone.next = (clone.next != null) ? clone.next.next : null;
33
               // Move to the next original node
34
               current = current.next;
35
36
37
           // Return the head of the cloned list
           return copyHead;
38
39
40 }
41
  // Definition for a Node.
   class Node {
       public int val; // Value of the node
44
```

43 current->next = copiedNode->next; // Advance the copiedNode pointer in the copied list to the next copied node, 44 // if it exists 45 if (copiedNode->next != nullptr) { 46

```
random: Node | null;
6
    /**
    * Creates a deep copy of a linked list where each node has a random pointer.
10
    * @param {Node | null} head - The head of the linked list to be copied.
    * @return {Node | null} The head of the copied linked list.
13
    */
   function copyRandomList(head: Node | null): Node | null {
       // Initialize a map to keep track of already copied nodes.
       const clonedNodesMap = new Map<Node, Node>();
17
       // First pass: copy all the nodes and map the original nodes to their copies.
18
       let currentNode = head;
19
       while (currentNode !== null) {
           clonedNodesMap.set(currentNode, { val: currentNode.val, next: null, random: null });
22
           currentNode = currentNode.next;
23
24
25
       // Second pass: Assign the next and random pointers.
26
       currentNode = head;
       while (currentNode !== null) {
           // Set the next pointer of the cloned node.
28
           clonedNodesMap.get(currentNode)!.next = currentNode.next ? clonedNodesMap.get(currentNode.next) : null;
29
30
           // Set the random pointer of the cloned node.
           clonedNodesMap.get(currentNode)!.random = currentNode.random ? clonedNodesMap.get(currentNode.random) : null;
31
           // Move to the next original node.
33
           currentNode = currentNode.next;
34
35
36
       // Return the cloned head which is the copy of the original head.
37
       return head ? clonedNodesMap.get(head) : null;
38 }
39
Time and Space Complexity
The given code implements a method to copy a linked list with a next and a random pointer.
```

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.

Time Complexity:

The algorithm involves three main steps:

This step takes O(N) time, where N is the number of nodes in the list.

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.

1. Iterating through the original list and creating a copy of each node which is inserted between the original node and its next node:

2. Iterating through the list again to update the random pointers of the copied nodes: For each of the original nodes, the

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

- Thus, the space complexity is 0(1) as we're only using a constant amount of extra space, regardless of the input size (ignoring the space required for the output).