<https://leetcode.com/problems/copy-list-with-random-pointer/>

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](https://en.wikipedia.org/wiki/Object_copying" \l "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.

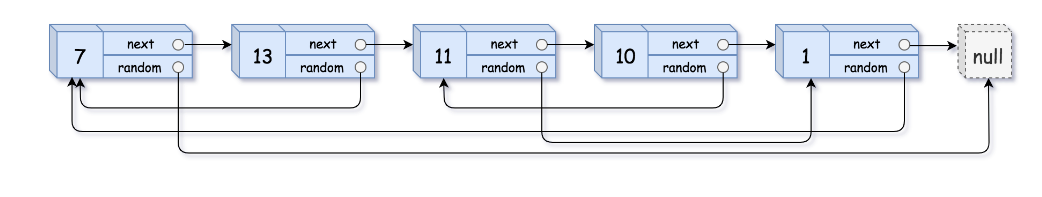
Return *the head of the copied linked list*.

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.

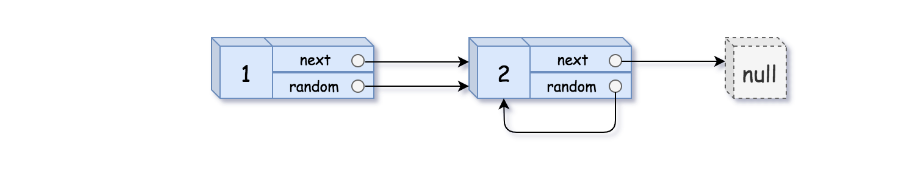
**Example 1:**



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

Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]

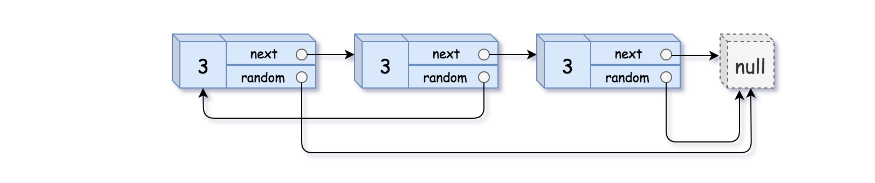
**Example 2:**



Input: head = [[1,1],[2,1]]

Output: [[1,1],[2,1]]

**Example 3:**



Input: head = [[3,null],[3,0],[3,null]]

Output: [[3,null],[3,0],[3,null]]

**Constraints:**

* 0 <= n <= 1000
* -104 <= Node.val <= 104
* Node.random is null or is pointing to some node in the linked list.

**Attempt 1: 2023-02-18**

**Solution 1: Copy list with three rounds (30 min)**

/\*

// Definition for a Node.

class Node {

int val;

Node next;

Node random;

public Node(int val) {

this.val = val;

this.next = null;

this.random = null;

}

}

\*/

class Solution {

public Node copyRandomList(Node head) {

if(head == null) {

return null;

}

// First round: make copy of each node, and link them together side-by-side in a single list.

Node iter = head;

while(iter != null) {

Node next = iter.next;

iter.next = new Node(iter.val);

iter.next.next = next;

iter = next;

}

// Second round: assign random pointers for the copy nodes.

iter = head;

while(iter != null) {

if(iter.random != null) {

iter.next.random = iter.random.next;

}

iter = iter.next.next;

}

// Third round: restore the original list, and extract the copy list.

iter = head;

Node copyHead = head.next;

Node copyIter = copyHead;

while(copyIter.next != null) {

iter.next = iter.next.next;

iter = iter.next;

copyIter.next = copyIter.next.next;

copyIter = copyIter.next;

}

// And actually I have a question for last "node.next = node.next.next;",

// why we need this? Sine copy node is the last one in list.

// (e.g 3 in 1 - 1' - 2 - 2' - 3 - 3').

// Because after the last loop, for the original list: 3->3'->null,

// so you need to remove 3'. Just make the last node.next = null

//iter.next = iter.next.next;

iter.next = null;

return copyHead;

}

}

Time Complexity: O(n), where n is the number of nodes in the Linked List

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/copy-list-with-random-pointer/solutions/43491/a-solution-with-constant-space-complexity-o-1-and-linear-time-complexity-o-n/>

An intuitive solution is to keep a hash table for each node in the list, via which we just need to iterate the list in 2 rounds respectively to create nodes and assign the values for their random pointers. As a result, the space complexity of this solution is O(N), although with a linear time complexity.

*Note: if we do not consider the space reversed for the output, then we could say that the algorithm does not consume any additional memory during the processing, i.e. O(1) space complexity*

As an optimizer solution, we could reduce the space complexity into constant. ***The idea is to associate the original node with its copy node in a single linked list. In this way, we don't need extra space to keep track of the new nodes.***

The algorithm is composed of the follow three steps which are also 3 iteration rounds.

* Iterate the original list and duplicate each node. The duplicate of each node follows its original immediately.
* Iterate the new list and assign the random pointer for each duplicated node.
* Restore the original list and extract the duplicated nodes.

The algorithm is implemented as follows:

public RandomListNode copyRandomList(RandomListNode head) {

RandomListNode iter = head, next;

// First round: make copy of each node,

// and link them together side-by-side in a single list.

while (iter != null) {

next = iter.next;

RandomListNode copy = new RandomListNode(iter.label);

iter.next = copy;

copy.next = next;

iter = next;

}

// Second round: assign random pointers for the copy nodes.

iter = head;

while (iter != null) {

if (iter.random != null) {

iter.next.random = iter.random.next;

}

iter = iter.next.next;

}

// Third round: restore the original list, and extract the copy list.

iter = head;

RandomListNode pseudoHead = new RandomListNode(0);

RandomListNode copy, copyIter = pseudoHead;

while (iter != null) {

next = iter.next.next;

// extract the copy

copy = iter.next;

copyIter.next = copy;

copyIter = copy;

// restore the original list

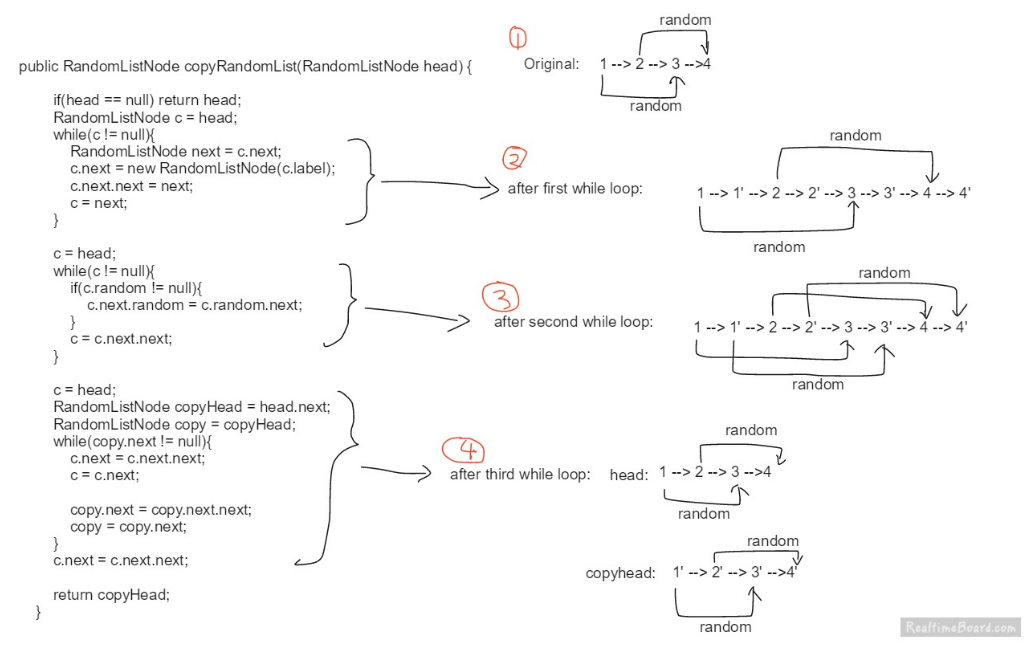
iter.next = next;

iter = next;

}

return pseudoHead.next;

}



A solution with constant space complexity O(1) and linear time complexity O(N)