<https://leetcode.com/problems/clone-graph/>

Given a reference of a node in a **connected** undirected graph.

Return a **deep copy** (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

class Node {

    public int val;

    public List<Node> neighbors;

}

**Test case format:**

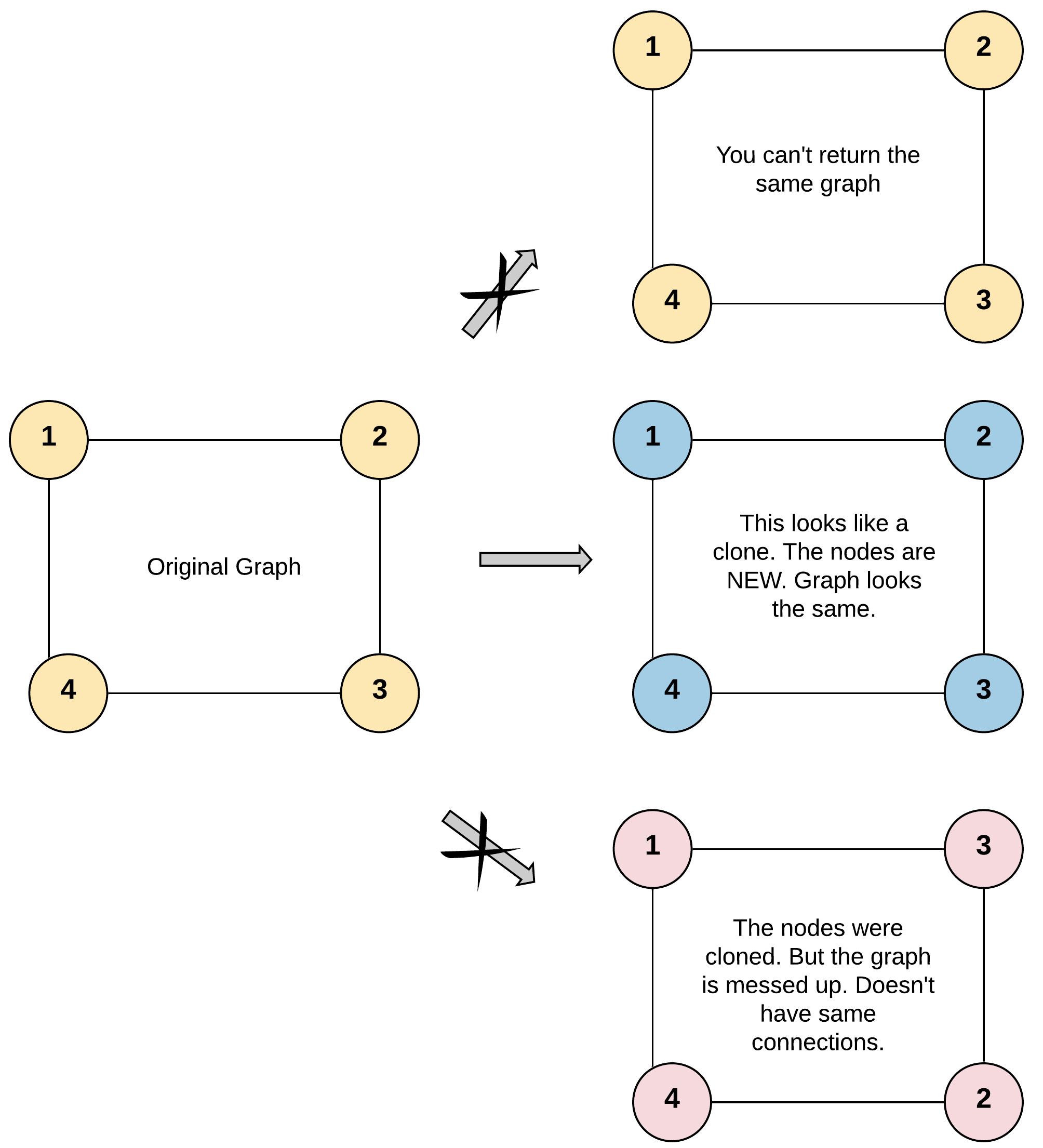
For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with val == 1, the second node with

val == 2, and so on. The graph is represented in the test case using an adjacency list.

**An adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the **copy of the given node** as a reference to the cloned graph.

**Example 1:**



Input: adjList = [[2,4],[1,3],[2,4],[1,3]]

Output: [[2,4],[1,3],[2,4],[1,3]]

Explanation: There are 4 nodes in the graph.

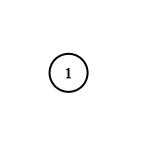
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

**Example 2:**



Input: adjList = [[]]

Output: [[]]

Explanation: Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

**Example 3:**

Input: adjList = []

Output: []

Explanation: This an empty graph, it does not have any nodes.

**Constraints:**

The number of nodes in the graph is in the range [0, 100].

1 <= Node.val <= 100

Node.val is unique for each node.

There are no repeated edges and no self-loops in the graph.

The Graph is connected and all nodes can be visited starting from the given node.

**Attempt 1: 2023-06-03**

**Solution 1: BFS (10 min)**

/\*

// Definition for a Node.

class Node {

    public int val;

    public List<Node> neighbors;

    public Node() {

        val = 0;

        neighbors = new ArrayList<Node>();

    }

    public Node(int \_val) {

        val = \_val;

        neighbors = new ArrayList<Node>();

    }

    public Node(int \_val, ArrayList<Node> \_neighbors) {

        val = \_val;

        neighbors = \_neighbors;

    }

}

\*/

class Solution {

    public Node cloneGraph(Node node) {

        if(node == null) {

            return null;

        }

        Map<Node, Node> map = new HashMap<Node, Node>();

        Queue<Node> q = new LinkedList<Node>();

        Node clone\_node = new Node(node.val);

        map.put(node, clone\_node);

        q.offer(node);

        while(!q.isEmpty()) {

            Node cur = q.remove();

            for(Node neighbor : cur.neighbors) {

                if(!map.containsKey(neighbor)) {

                    q.offer(neighbor);

                    Node clone\_neighbor = new Node(neighbor.val);

                    map.put(neighbor, clone\_neighbor);

                }

                // Build connection between clone node and its clone neighbor

                Node clone\_cur = map.get(cur);

                clone\_cur.neighbors.add(map.get(neighbor));

            }

        }

        return clone\_node;

    }

}

**Refer to**

<https://aaronice.gitbook.io/lintcode/graph_search/clone_graph>

思路1：使用BFS，先将头节点入queue，每一次queue出列一个node，然后检查这个node的所有的neighbors，如果没visited过，就入队，并更新neighbor

这是一种对图的遍历方法，对于一个节点来说先把所有neighbors都检查一遍，再从 第一个neighbor开始，循环往复。

由于BFS的这个特质，BFS可以帮助寻找最短路径。

通常BFS用queue+循环实现。

BFS - breadth first search, non-recursive

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\*    int label;

\*    ArrayList<UndirectedGraphNode> neighbors;

\*    UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

    /\*\*

    \* @param node: A undirected graph node

    \* @return: A undirected graph node

    \*/

    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

        if (node == null) {

            return null;

        }

        HashMap<UndirectedGraphNode, UndirectedGraphNode> hm = new HashMap<UndirectedGraphNode, UndirectedGraphNode>();

        LinkedList<UndirectedGraphNode> queue = new LinkedList<UndirectedGraphNode>();

        UndirectedGraphNode head = new UndirectedGraphNode(node.label);

        hm.put(node, head);

        queue.add(node);

        while (!queue.isEmpty()) {

            UndirectedGraphNode currentNode = queue.remove();

            for (UndirectedGraphNode neighbor : currentNode.neighbors) {

                if (!hm.containsKey(neighbor)) {

                    queue.add(neighbor);

                    UndirectedGraphNode newNeighbor = new UndirectedGraphNode(neighbor.label);

                    hm.put(neighbor, newNeighbor);

                }

                hm.get(currentNode).neighbors.add(hm.get(neighbor));

            }

        }

        return head;

    }

}

**Solution 2: DFS (30 min)**

**How to traverse a graph based on DFS ?**

**Refer to**

**Graph Traversals and Directed Graph Cycle Detection (in Graph Document)**

<https://www.ics.uci.edu/~thornton/ics46/Notes/GraphTraversals/>

**Style 1: Using "visited" set as classic DFS graph traversal way**

/\*

// Definition for a Node.

class Node {

    public int val;

    public List<Node> neighbors;

    public Node() {

        val = 0;

        neighbors = new ArrayList<Node>();

    }

    public Node(int \_val) {

        val = \_val;

        neighbors = new ArrayList<Node>();

    }

    public Node(int \_val, ArrayList<Node> \_neighbors) {

        val = \_val;

        neighbors = \_neighbors;

    }

}

\*/

class Solution {

    public Node cloneGraph(Node node) {

        if(node == null) {

            return null;

        }

        // In this question we will create an array of Node(not boolean)

        // why ? , because i have to add all the adjacent nodes of particular

        // vertex, whether it's visited or not, so in the Node[] initially

        // null is stored, if that node is visited, we will store the

        // respective node at the index, and can retrieve that easily.

        Node[] visited = new Node[101];

        Node clone\_node = new Node(node.val);

        // Make a dfs call for traversing all the vertices of the root node

        helper(node, clone\_node, visited);

        // Return the copy root node

        return clone\_node;

    }

    private void helper(Node node, Node clone\_node, Node[] visited) {

        // Store the current node at it's val index which will tell us that

        // this node is now visited

        visited[node.val] = clone\_node;

        // Traverse for the adjacent nodes of root node

        for(Node neighbor : node.neighbors) {

            // Check whether that node is visited or not

            // if it is not visited, there must be null

            if(visited[neighbor.val] == null) {

                // If it not visited, create a new node, add this node as

                // the neighbor of the prev copied node

                Node clone\_neighbor = new Node(neighbor.val);

                clone\_node.neighbors.add(clone\_neighbor);

                // Make dfs call for this unvisited node (start with 'neighbor')

                // to discover whether it's adjacent nodes are explored or not

                helper(neighbor, clone\_neighbor, visited);

            } else {

                // If that node is already visited, retrieve that node from visited

                // array and add it as the adjacent node of prev copied node

                // THIS IS THE POINT WHY WE USED NODE[] INSTEAD OF BOOLEAN[] ARRAY

                clone\_node.neighbors.add(visited[neighbor.val]);

                // No need dfs call for already visited node

            }

        }

    }

}

**Refer to**

<https://leetcode.com/problems/clone-graph/solutions/1793436/java-simple-code-with-heavy-comments/>

/\*

// Definition for a Node.

class Node {

    public int val;

    public List<Node> neighbors;

    public Node() {

        val = 0;

        neighbors = new ArrayList<Node>();

    }

    public Node(int \_val) {

        val = \_val;

        neighbors = new ArrayList<Node>();

    }

    public Node(int \_val, ArrayList<Node> \_neighbors) {

        val = \_val;

        neighbors = \_neighbors;

    }

}

\*/

class Solution {

    public void dfs(Node node , Node copy , Node[] visited){

        visited[copy.val] = copy;// store the current node at it's val index which will tell us that this node is now visited

//        now traverse for the adjacent nodes of root node

        for(Node n : node.neighbors){

//            check whether that node is visited or not

//              if it is not visited, there must be null

            if(visited[n.val] == null){

//                so now if it not visited, create a new node

                Node newNode = new Node(n.val);

//                add this node as the neighbor of the prev copied node

                copy.neighbors.add(newNode);

//                make dfs call for this unvisited node to discover whether it's adjacent nodes are explored or not

                dfs(n , newNode , visited);

            }else{

//                if that node is already visited, retrieve that node from visited array and add it as the adjacent node of prev copied node

//                THIS IS THE POINT WHY WE USED NODE[] INSTEAD OF BOOLEAN[] ARRAY

                copy.neighbors.add(visited[n.val]);

            }

        }

    }

    public Node cloneGraph(Node node) {

        if(node == null) return null; // if the actual node is empty there is nothing to copy, so return null

        Node copy = new Node(node.val); // create a new node , with same value as the root node(given node)

        Node[] visited = new Node[101]; // in this question we will create an array of Node(not boolean) why ? , because i have to add all the adjacent nodes of particular vertex, whether it's visited or not, so in the Node[] initially null is stored, if that node is visited, we will store the respective node at the index, and can retrieve that easily.

        Arrays.fill(visited , null); // initially store null at all places

        dfs(node , copy , visited); // make a dfs call for traversing all the vertices of the root node

        return copy; // in the end return the copy node

    }

}

**Style 2: Not using "visited" set as classic DFS graph traversal way,  Hashmap is doing the same thing that "visited" set do.**

/\*

// Definition for a Node.

class Node {

    public int val;

    public List<Node> neighbors;

    public Node() {

        val = 0;

        neighbors = new ArrayList<Node>();

    }

    public Node(int \_val) {

        val = \_val;

        neighbors = new ArrayList<Node>();

    }

    public Node(int \_val, ArrayList<Node> \_neighbors) {

        val = \_val;

        neighbors = \_neighbors;

    }

}

\*/

class Solution {

    public Node cloneGraph(Node node) {

        if(node == null) {

            return null;

        }

        return helper(node, new HashMap<Node, Node>());

    }

    private Node helper(Node node, Map<Node, Node> map) {

        Node clone\_node = new Node(node.val);

        map.put(node, clone\_node);

        for(Node neighbor : node.neighbors) {

            Node clone\_neighbor = map.get(neighbor);

            if(clone\_neighbor != null) {

                clone\_node.neighbors.add(clone\_neighbor);

            } else {

                clone\_node.neighbors.add(helper(neighbor, map));

            }

        }

        return clone\_node;

    }

}

**Refer to**

<https://aaronice.gitbook.io/lintcode/graph_search/clone_graph>

思路2：使用DFS，可以分为迭代和循环两种方式，后者需要利用stack。

DFS - depth first search, recursive

public class Solution {

    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

        if (node == null) {

            return null;

        }

        return cloneGraph(node, new HashMap<>());

    }

    private UndirectedGraphNode cloneGraph(UndirectedGraphNode node,

            Map<UndirectedGraphNode, UndirectedGraphNode> cloneMap) {

        UndirectedGraphNode clone = new UndirectedGraphNode(node.label);

        cloneMap.put(node, clone);

        for (UndirectedGraphNode neighbor : node.neighbors) {

            UndirectedGraphNode neighborClone = cloneMap.get(neighbor);

            if (neighborClone != null) {

                clone.neighbors.add(neighborClone);

            }

            else {

                clone.neighbors.add(cloneGraph(neighbor, cloneMap));

            }

        }

        return clone;

    }

}

DFS - depth first search, non-recursive

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\*    int label;

\*    ArrayList<UndirectedGraphNode> neighbors;

\*    UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

    /\*\*

    \* @param node: A undirected graph node

    \* @return: A undirected graph node

    \*/

    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

        if(node == null)

            return null;

        HashMap<UndirectedGraphNode, UndirectedGraphNode> hm = new HashMap<UndirectedGraphNode, UndirectedGraphNode>();

        LinkedList<UndirectedGraphNode> stack = new LinkedList<UndirectedGraphNode>();

        UndirectedGraphNode head = new UndirectedGraphNode(node.label);

        hm.put(node, head);

        stack.push(node);

        while(!stack.isEmpty()){

            UndirectedGraphNode curnode = stack.pop();

            for(UndirectedGraphNode aneighbor: curnode.neighbors){//check each neighbor

                if(!hm.containsKey(aneighbor)){//if not visited,then push to stack

                    stack.push(aneighbor);

                    UndirectedGraphNode newneighbor = new UndirectedGraphNode(aneighbor.label);

                    hm.put(aneighbor, newneighbor);

                }

                hm.get(curnode).neighbors.add(hm.get(aneighbor));

            }

        }

        return head;

    }

}