目 录

1.反转一个单链表

示例

```
输入：1->2->3->4->5->NULL
输出：5->4->3->2->1->NULL
```

Tips

1. 链表为空（`head == null`）或者链表只有一个元素（`head.next == null`）。反转后还是自身：`return head;`
2. 链表反转：首先用一个引用储存当前元素的下一个元素。`nextNode = curNode;` 然后把当前元素 `curNode` 的 `next` 指向它的前一个元素 `prevNode`：`curNode.next = prevNode`
3. 第 2 步执行完后，处理下一个元素：`prevNode = curNode;curNode = curNode.next`。
4. `curNode == null` 时，说明到了链表的末端（循环退出条件）。此时 `prevNode` 指向的是原链表的最后一个元素，即新链表的首节点：`return prevNode;`
5. 初始化：第一个元素没有前驱结点，反转后，原链表第一个节点为新链表的最后一个节点，新链表的最后一个节点的 `next` 应指向 `null`。所以 `prevNode` 初始化为 `null`：`Node prevNode = null;`，`curNode` 最开始应指向链表的首节点：`Node curNode = head;`。

迭代版本

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode prevNode = null;
        ListNode curNode = head;

        while(curNode != null){
            ListNode nextNode = curNode.next;
            curNode.next = prevNode;
            prevNode = curNode;
            curNode = nextNode;
        }
        return prevNode;
    }
}
```

Tips

1. 可以将问题分为一个头结点和链表的剩余部分。
2. 头结点为问题的平凡解：链表只有一个节点，直接返回。
3. 子问题的解：头节点（`head`）的下一个节点 `head.next` 的 `next` 指向头结点。
4. 头结点指向 `null`

递归版本

```
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode p = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return p;
    }
}
```

2.移除链表元素

删除链表中等于给定值 val 的所有节点。

示例

```
输入: 1->2->6->3->4->5->6, val = 6
输出: 1->2->3->4->5
```

Tips

要考虑链表为空的情况。链表为空，直接返回 `null`

使用傀儡节点 `dummy` 来进行辅助删除。 `dummy.next = head`

两个指针，一个指向待删除元素（ `cur` ）。一个指向待删除元素的前一个位置（ `prev` ）。

如果找到了待删除元素:将该元素前一个位置(`pre`)的 `next` 指向删除元素的 `next` : `pre.next = cur.next` 。然后将 `cur` 复位,即指向 `pre.next` : `cur = pre.next`

如果没有找到，两个指针同时向后移动: `pre = pre.next;cur = cur.next;`

循环结束的条件: `cur` 指向空，表示已经找到了链表的最后。

最后返回 `dummy.next`

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        if (head == null){
            return null;
        }
        ListNode dummy = new ListNode(0,head);
        ListNode prev = dummy;
        ListNode cur = dummy.next;
        while(cur != null){
            if(cur.val == val){
```

```
                prev.next = cur.next;
                cur = prev.next;
            }else{
                prev = prev.next;
                cur = cur.next;
            }
        }
        return dummy.next;
    }
}
```

3.杨辉三角

给定一个非负整数 `numRows`，生成杨辉三角的前 `numRows` 行。

```
class Solution {
    public List<List<Integer>> generate(int numRows) {

        List<List<Integer>> pasTri = new ArrayList<>();
        if(numRows == 0){
            return pasTri;
        }
        List<Integer> first = new ArrayList<>();
        first.add(1);
        pasTri.add(first);
        if(numRows == 1){
            return pasTri;
        }
        List<Integer> second = new ArrayList<>();
        second.add(1);
        second.add(1);
        pasTri.add(second);
        if(numRows == 2){
            return pasTri;
        }
        for(int i = 3; i <= numRows; i++){
            List<Integer> list1 = new ArrayList<>();
            list1.add(1);
            for(int j = 2; j <= i - 1; j++){
                int a = pasTri.get(i - 1 - 1).get(j - 1);
                int b = pasTri.get(i - 1 - 1).get(j - 1 - 1);
                list1.add(a + b);
            }
            list1.add(1);
            pasTri.add(list1);
        }
        return pasTri;
    }
}
```

4.存在连续三个奇数的数组

给你一个整数数组 `arr`，请你判断数组中是否存在连续三个元素都是奇数的情况：如果存在，请返

回 `true`；否则，返回 `false`。

示例 1

```
输入：arr = [2,6,4,1]
输出：false
解释：不存在连续三个元素都是奇数的情况。
```

示例 2

```
输入：arr = [1,2,34,3,4,5,7,23,12]
输出：true
解释：存在连续三个元素都是奇数的情况，即 [5,7,23] 。
```

版本 1：LeetCode 提供的

```
class Solution {
    public boolean threeConsecutiveOdds(int[] arr) {
        for(int i = 0; i < arr.length - 2; i++){
            if((arr[i] % 2 == 1) && (arr[i+1] % 2 == 1) && (arr[i + 2] % 2 == 1)){
                return true;
            }
        }
        return false;
    }
}
```

版本 2：自己使用顺序表实现

```
public class MyArrayList1 {
    public boolean threeConsecutiveOdds(List<Integer> arr) {
        for (int i = 0; i < arr.size() - 2; i++) {
            if ((arr.get(i) % 2 == 1) && (arr.get(i + 1) % 2 == 1) && (arr.get(i +
                2) % 2 == 1)){
                return true;
            }
        }
        return false;
    }
}
```

5.员工的重要性

给定一个保存员工信息的数据结构，它包含了员工唯一的 id，重要度和直系下属的 id。比如，员工 1 是员工 2 的领导，员工 2 是员工 3 的领导。他们相应的重要度为 15, 10, 5。那么员工 1 的数据结构是 [1, 15, [2]]，员工 2 的数据结构是 [2, 10, [3]]，员工 3 的数据结构是 [3, 5, []]。注意虽然员工 3 也是员工 1 的一个下属，但是由于并不是直系下属，因此没有体现在员工 1 的数据结构中。

现在输入一个公司的所有员工信息，以及单个员工 id，返回这个员工和他所有下属的重要度之和。

```
示例 1:

输入: [[1, 5, [2, 3]], [2, 3, []], [3, 3, []]], 1
```

```
输出: 11
解释:
员工1自身的重要度是5，他有两个直系下属2和3，而且2和3的重要度均为3。因此员工1的总重要度是 5 + 3 + 3 = 11。
注意:
```

一个员工最多有一个直系领导，但是可以有多个直系下属, 员工数量不超过 2000。

Tips

1. 遍历整个员工列表 employees，找到符合 id 的员工 employee
2. 这个员工 employee 如果没有下属 employee.subordinates.size()==0，重要度就是自己的重要度 employee.importance。
3. 如果这个员工有下属，算出每个下属及下属的重要度。

```
/*
// Definition for Employee.
class Employee {
    public int id;
    public int importance;
    public List<Integer> subordinates;
};
*/

class Solution {
    public int getImportance(List<Employee> employees, int id) {

        for(int i = 0;i < employees.size();i++){
            Employee employee = employees.get(i);
            if(employee.id == id){
                if(employee.subordinates.size() == 0){//没有下属
                    return employee.importance;
                }
                for(int j = 0; j < employee.subordinates.size(); j++){
                    employee.importance += getImportance(employees,employee.
                        subordinates.get(j));
                }
                return employee.importance;
            }
        }
        return 0;
    }
}
```