

链表常见题目

xbZhong

[本页PDF](#)

反转链表

206. 反转链表

已解答

简单

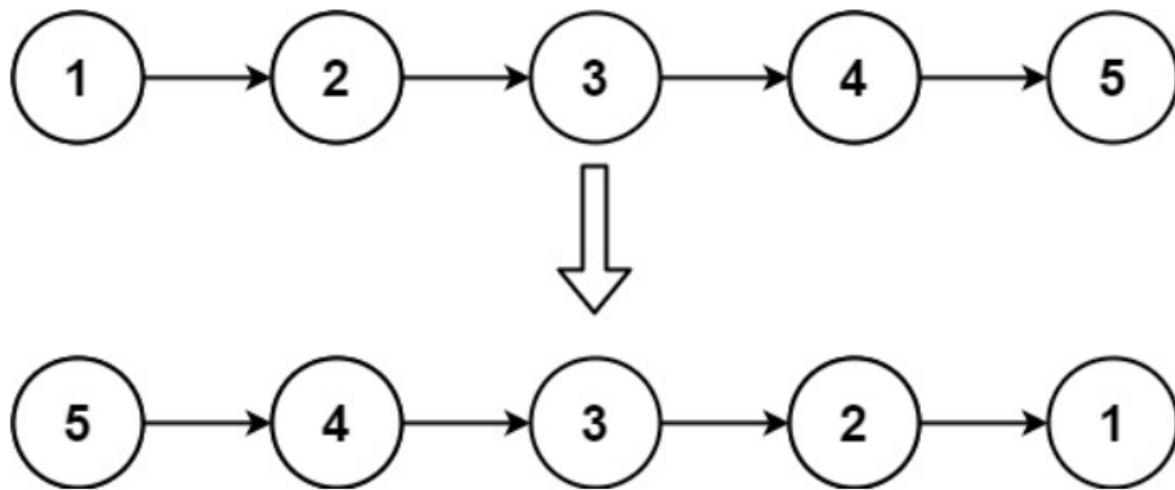
相关标签

相关企业

A_文

给你单链表的头节点 `head`，请你反转链表，并返回反转后的链表。

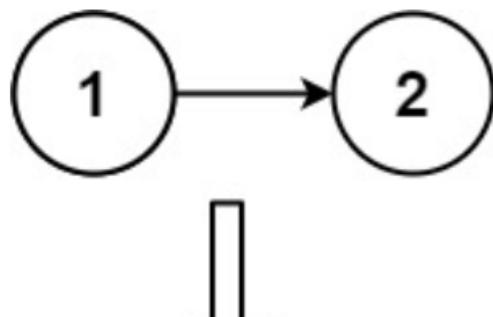
示例 1：



输入： `head = [1,2,3,4,5]`

输出： `[5,4,3,2,1]`

示例 2：



alt text

反转链表迭代做法

通过创建一个新的节点prev，将旧链表从前往后遍历

```
class Solution { public: ListNode* reverseList(ListNode* head) { ListNode  
*prev=NULL,*temp = head; while(temp) { ListNode *next = temp->next; temp->next =  
prev; prev = temp ; temp = next; } return prev; } };
```

反转链表头插法做法

```
class Solution { public: ListNode* reverseList(ListNode* head) { ListNode new_head,  
*p =head ,*q; new_head.next = NULL; while(p) { q = p->next; p->next =  
new_head.next; new_head.next = p; p = q; } return new_head.next; } };
```

反转链表递归做法

```
class Solution { public: ListNode* reverseList(ListNode* head) { if(head == NULL ||  
head->next == NULL) return head; ListNode* tail = head->next; ListNode* newhead =  
reverseList(head->next); //把reverseList head->next = tail->next; (head->next) 看成  
一个新的链表，其头节 tail->next = head; 点为newhead return newhead; } };
```

判断环形链表

快慢指针法

- 用两个指针来判断
- 一个指针跑得快，一个指针跑得慢
- 若不是环形结构，跑的慢的永远追不上跑得快的，因此一定会返回false
- 若是环形结构，跑得慢的一定会遇到跑得快的，因此一定会返回true

根据上述思考来设计代码

```
class Solution { public: bool hasCycle(ListNode *head) { ListNode *p = head , *q =  
head; while(q && q->next) //要加上q->next不指向null的条件 { 否则当链表只有一个节点时便会出现  
错 p = p->next; q = q->next->next; if(p == q) return true; } return false; } };
```

快乐数

(使用快慢指针思想来做这道题)

202. 快乐数

简单

相关标签

相关企业

A_z

编写一个算法来判断一个数 n 是不是快乐数。



「快乐数」 定义为：

- 对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和。
- 然后重复这个过程直到这个数变为 1，也可能是 **无限循环** 但始终变不到 1。
- 如果这个过程 **结果为 1**，那么这个数就是快乐数。

如果 n 是 快乐数 就返回 `true`；不是，则返回 `false`。

alt text

- 情况1：一直在1中死循环
- 情况2：一直在某些数据中死循环，但到达不了1 因此，我们可以使用快慢指针的思想来解题

代码：

```
class Solution { public: int get(int n) { int y=0,d; while(n) { d = n % 10; y += d * d; n = n / 10; } return y; } bool isHappy(int n) { int p = n,q = n; while(q!=1) // q是快指针，因此使用q来判定是否循环 (q若可以等于 { 1, 则p也一定可以等于1}) p=get(p); q=get(get(q)); if(p==q && q!=1) // p和q相等但不是在1中死循环 return false; } return true; };
```

双指针等距移动法

旋转链表

61. 旋转链表

已解答

中等

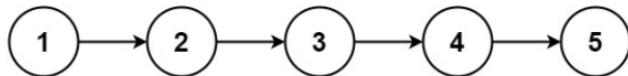
相关标签

相关企业

A+

给你一个链表的头节点 `head`，旋转链表，将链表每个节点向右移动 `k` 个位置。

示例 1：



输入: `head = [1,2,3,4,5], k = 2`

输出: `[4,5,1,2,3]`

alt text

个人解法：

```
class Solution { public: ListNode* rotateRight(ListNode* head, int k) { if(head == NULL) return head; int len = 0; //链表长度 ListNode *p = head ; ListNode *q = head,*temp,*newhead; while (q) { q = q->next; len+=1; } int x = k % len; if(x == 0) return head; for(int i = 1;i < len-x ; i++)//分割成俩链表 p = p->next; //遍历至前一链表末端 temp = p->next; //记录下一链表的头节点 newhead = p->next; p->next = NULL; //将前一链表末端指null while(temp->next != NULL) //遍历后一链表至末端 temp = temp->next; temp->next = head; return newhead; } };
```

双指针移动法：要判断特殊情况

```
class Solution { public: ListNode* rotateRight(ListNode* head, int k) { if(head == NULL) return head; ListNode *p = head , *q = head,*temp = head; int len = 0; while(head) { head = head->next; len+=1; } k %= len; if(k==0) return temp; for(int i = 0;i <= k ;i++) q = q->next; while(q) { p = p->next; q = q->next; } q = p->next; p->next = NULL; p = q; while(p->next) p = p->next; p->next = temp; return q; } };
```

删除链表的倒数第N个节点

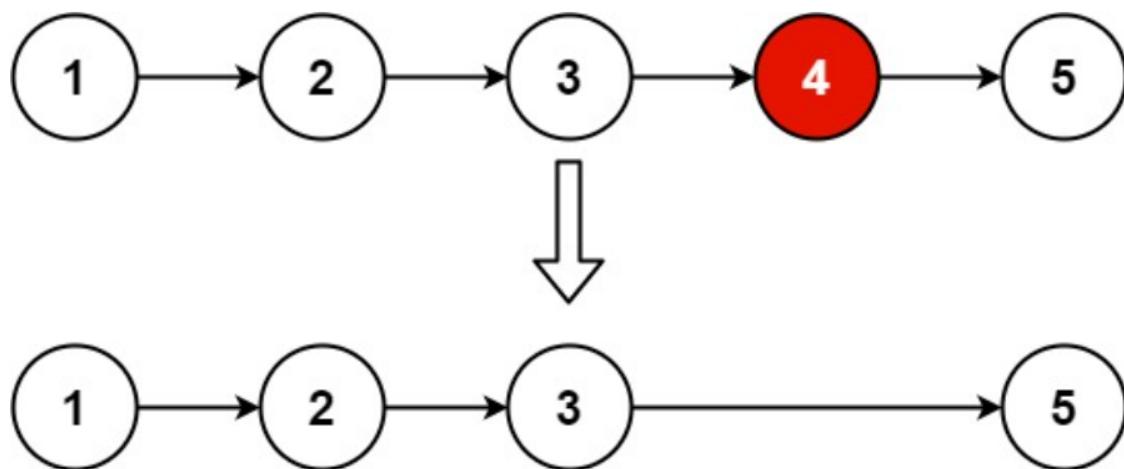
19. 删除链表的倒数第 N 个结点

已解答 ✓

中等 相关标签 相关企业 提示

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

示例 1：



输入: head = [1,2,3,4,5], n = 2

输出: [1,2,3,5]

个人解法：

```
class Solution { public: ListNode* removeNthFromEnd(ListNode* head, int n) {
    ListNode *p = head,*q = head; ListNode new_head; new_head.next = head; int len = 0;
    while(q) { q = q->next; len +=1; } if(n == len) { new_head.next = head->next;
    return new_head.next; } for(int i = 1;i < len-n ;i++) p = p->next; p->next = p-
    >next->next; return head; } };
```

需要考虑边界条件，当要删除第一个节点时，便不能return head，而需要找到虚拟头节点进行return，同时运行速度也不快。

双指针移动法：（使用虚拟头节点） * 使用两个指针，一快一慢，并且都指向虚拟头节点 * 让快的指针先往后走 $n+1$ 位 * 然后两个指针一起往后走，当快指针走到null时，慢指针走到了待删除节点的前一位

```

class Solution { public: ListNode* removeNthFromEnd(ListNode* head, int n) {
    ListNode new_head , *p = &new_head , *q = p; new_head.next = head; for(int i = 1; i
    <= n+1;i++) q = q->next; while (q) { p = p->next; q = q->next; } p->next =
    p->next->next; return new_head.next; } };

```

好处：1. 不需要考虑边界条件，因为有了虚拟头节点的存在 2. 运行速度更快

环形链表 II

142. 环形链表 II

中等

相关标签

相关企业

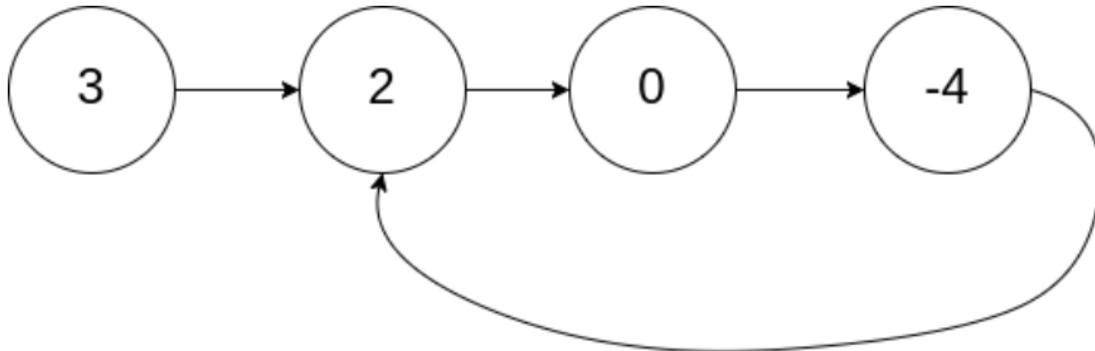
A+

给定一个链表的头节点 `head`，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 -1，则在该链表中没有环。注意：`pos` 不作为参数进行传递，仅仅是为了标识链表的实际情况。

不允许修改 链表。

示例 1：



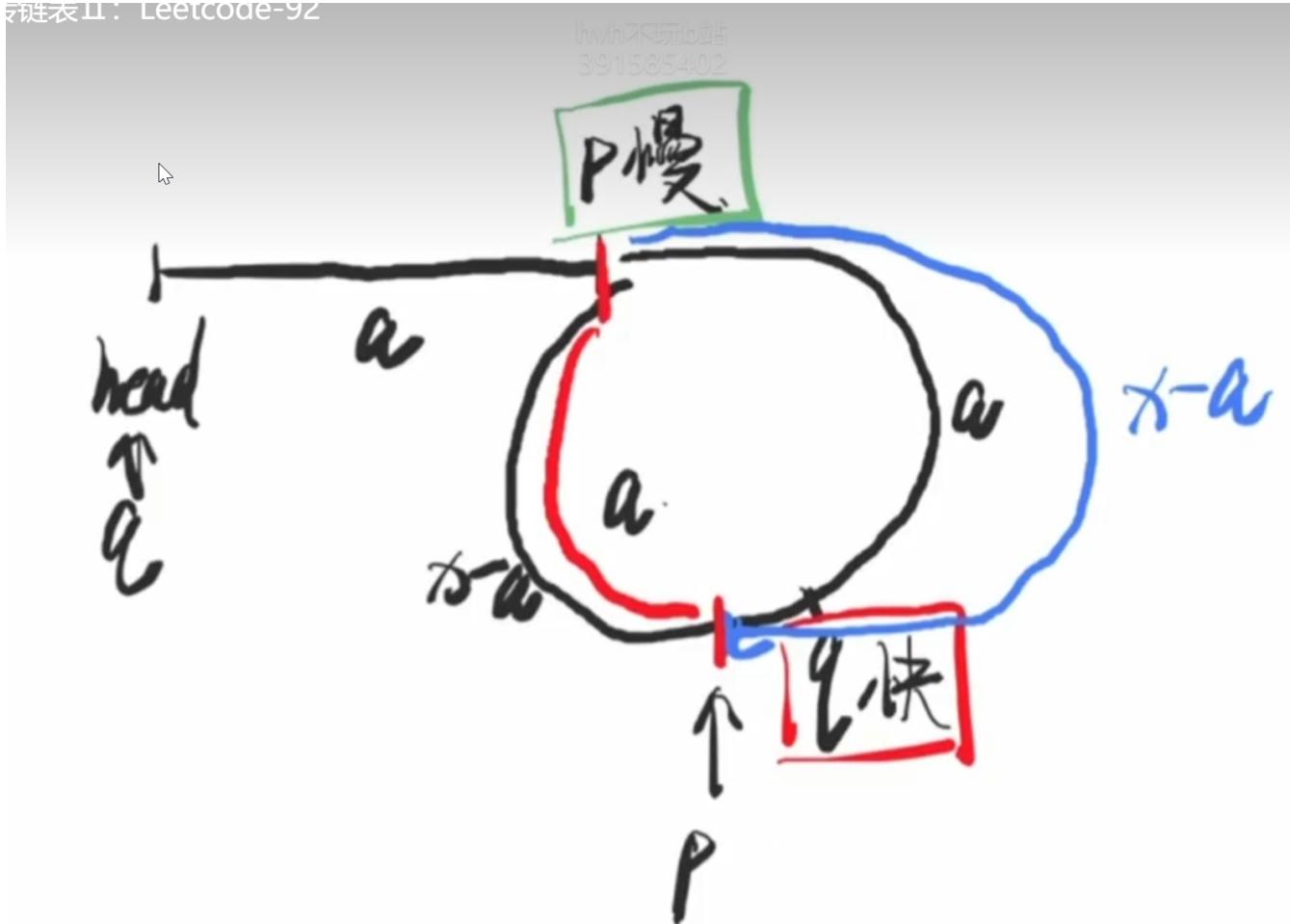
输入： `head = [3, 2, 0, -4]`, `pos = 1`

输出： 返回索引为 1 的链表节点

解释： 链表中有一个环，其尾部连接到第二个节点。

这道题的解法非常巧妙 同样使用快慢指针，得出其路程之间的关系

- 设定两个指针，一个一次走两步，一个一次走一步（设头节点距环入口为的距离为a）
- 如果有环，他俩必定相遇
- 通过数学关系得出相遇的位置到环入口的距离等于头节点到环入口的距离
 - 设环长度为x，在慢指针刚到达环入口时，快指针领先他a的距离
 - 快指针与慢指针距离为x-a
 - 快指针追上慢指针需要再走x-a次
 - 慢指针在此过程中走了x-a步，相遇时的位置距离环入口为x- (x-a) =a的距离
 - 相遇位置与头节点距环入口距离相同



- 将其中一个指针指向头节点，让它们以相同速度走，相遇位置即环入口

附上代码：

```
class Solution { public: ListNode *detectCycle(ListNode *head) { ListNode *q = head
, *p = head; while(p && p->next) { q = q->next; p = p->next->next; if(p==q) break;
} if (p == NULL || p->next == NULL) return NULL; q = head; while(p != q) { p = p-
>next; q = q->next; } return q; }};
```