0.链表总结.md

- 快慢指针法/双指针法
  - 19 删除链表的倒数第 N 个结点
  - 141 环形链表
  - 142 环形链表 II
  - 876 链表的中间节点
  - 1721 交换链表中的节点
- 链表常规操作：遍历，reverse，merge
  - 2 两数相加：不要忘记最后的remainder
  - 21 合并两个有序链表
  - 86 分割链表
  - 147 对链表进行插入排序: 一个指针顺序遍历，一个指针从头寻找插入位置，创建新的有序节点
  - 206 反转链表：newnode，交替向前
  - 725 分隔链表：先插入result，再遍历到位置断开链接
  - [剑指 Offer 06]. 从尾到头打印链表
- 修改link
  - 24 两两交换链表中的节点
  - 61 旋转链表
  - 82 删除排序链表中的重复元素 II：要向前判断两步
  - 83 删除排序链表中的重复元素
  - 138 复制带随机指针的链表: 注意Dict第一遍只创建val， 注意加入None节点
  - 203 移除链表元素
  - 237 删除链表中的节点
  - 328 奇偶链表: 不需要新节点，只需要遍历偶数节点，然后交叉赋值
- 组合题：用到多个链表操作，
  - 23 合并K个升序链表：归并排序，递归合并两个链表+sort merge
  - 92 反转链表 II：找到倒数N节点+翻转链表+合并链表
  - 143 重排链表v：中间节点+翻转链表+合并链表【原地合并：第二个合并到第一个链表上】
  - 148 排序链表: 归并排序，递归进行find_mid + sort merge【注意中间节点fast判断条件】
  - 234 回文链表：中间节点+翻转链表+遍历
- 设计类
  - 146 LRU：双向链表，插入head(先建立node左右link，再插入），超过长度移除tail.prev, 移除是把元素左右node链接
  - 460 LFU：
  - 707 设计链表
- 技巧题
  - 160 相交链表：可以用常规接发， 找到长度，拼接成相同长度，然后遍历判断是否有相交节点
  - 382 链表随机节点：蓄水池算法，第K个节点有1/K的概率overwrite之前sample的元素

138. 复制带随机指针的链表.md

给你一个长度为 n 的链表，每个节点包含一个额外增加的随机指针 random ，该指针可以指向链表中的任何节点或空节点。

构造这个链表的 深拷贝。 深拷贝应该正好由 n 个 全新 节点组成，其中每个新节点的值都设为其对应的原节点的值。新节点的 next 指针和 random 指针也都应指向复制链表中的新节点，并使原链表和复制链表中的这些指针能够表示相同的链表状态。复制链表中的指针都不应指向原链表中的节点 。

例如，如果原链表中有 X 和 Y 两个节点，其中 X.random --> Y 。那么在复制链表中对应的两个节点 x 和 y ，同样有 x.random --> y 。

返回复制链表的头节点。

用一个由 n 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 [val, random_index] 表示：

```
val：一个表示 Node.val 的整数。
random_index：随机指针指向的节点索引（范围从 0 到 n-1）；如果不指向任何节点，则为  null 。
```

你的代码 只 接受原链表的头节点 head 作为传入参数。

示例 1：

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

示例 2：

输入：head = [[1,1],[2,1]]
输出：[[1,1],[2,1]]

示例 3：

输入：head = [[3,null],[3,0],[3,null]]
输出：[[3,null],[3,0],[3,null]]

示例 4：

输入：head = []
输出：[]
解释：给定的链表为空（空指针），因此返回 null。

提示：

```
0 <= n <= 1000
-10000 <= Node.val <= 10000
Node.random 为空（null）或指向链表中的节点。
```

```
"""
```

```
# Definition for a Node.
class Node:
    def __init__(self, x: int, next: 'Node' = None, random: 'Node' = None):
        self.val = int(x)
        self.next = next
        self.random = random
"""

class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        dic = {}
        node = head
        #复制valu
        while node:
            dic[node] = Node(node.val)
            node = node.next
        #复制link
        node = head
        dic[None]=None
        while node:
            dic[node].next = dic[node.next]
            dic[node].random = dic[node.random]
            node = node.next
        return dic[head]
```

Tips

1. 核心不在random指针，而是如何深拷贝linklist. 先对每一个node创建新节点（复制value），然后把再拷贝link
2. 不能直接创建的原因是random index可能会多词访问相同的node，所以需要先创建node，并放到dic里面方便后序进一步访问

141. 环形链表.md

给定一个链表，判断链表中是否有环。

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，我们使用整数 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。

如果链表中存在环，则返回 true 。 否则，返回 false 。

进阶：

你能用 O(1)（即，常量）内存解决此问题吗?

示例 1：

输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。

示例 2：

输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。

示例 3：

输入：head = [1], pos = -1
输出：false
解释：链表中没有环。

提示：

```
链表中节点的数目范围是 [0, 104]
-105 <= Node.val <= 105
pos 为 -1 或者链表中的一个 有效索引 。
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        slow = head
        fast = head.next
        while fast.next:
            if slow == fast:
                return True
            fast = fast.next.next
            slow = slow.next

        return False
```

Tips

1. 快慢指针解法类似于追及问题，快的走两步，慢的走一步，如果有环，那二者总会相遇(遍历时间小于链表长

度N）。如果没有环终止条件就是快指针到tail

2.

142. 环形链表 II.md

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

为了表示给定链表中的环，我们使用整数 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 仅仅是用于标识环的情况，并不会作为参数传递到函数中。

说明：不允许修改给定的链表。

进阶：

```
你是否可以使用 O(1) 空间解决此题?
```

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。

示例 2：

输入：head = [1,2], pos = 0
输出：返回索引为 0 的链表节点
解释：链表中有一个环，其尾部连接到第一个节点。

示例 3：

输入：head = [1], pos = -1
输出：返回 null
解释：链表中没有环。

提示：

```
链表中节点的数目范围在范围 [0, 104] 内
-105 <= Node.val <= 105
pos 的值为 -1 或者链表中的一个有效索引
```

1. 常规解法

```python
class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        dic = set()
        while head:
            if head in dic:
                return head
            dic.add(head)
            head = head.next
        return None
```

Tips:

把遍历过的node都存放在字典里，当第一个再次遍历到的node就是环的入口

2. 技巧解法

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                node = head
                while slow != node:
                    slow = slow.next
                    node = node.next
                return node
        return None
```

Tips

在环形链表上再进一步。如果存在环, fast和slow相遇在C，B是环的入口。A->B->C->B->C->B, 各个路径的距离分别是x,y,z

slow能追上fast，会有

$$x + n(y + z) + y = 2 * (x + y)$$
$$x = (n - 1)(y + z) + z$$

也就是当slow和fast在C点相遇后，只要有一个ptr从head开始，和slow一起向前跑，当slow再次走到C，并绕着环跑了n-1圈后，两者会在B相遇

143. 重排链表v.md

给定一个单链表 L 的头节点 head ，单链表 L 表示为：

L0 → L1 → … → Ln - 1 → Ln

请将其重新排列后变为：

L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …

不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1：

输入：head = [1,2,3,4]
输出：[1,4,2,3]

示例 2：

输入：head = [1,2,3,4,5]
输出：[1,5,2,4,3]

提示：

```
链表的长度范围为 [1, 5 * 104]
1 <= node.val <= 1000
```

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

    def __str__(self):
        res = []
        cur = self
        while cur:
            res.append(str(cur.val))
            cur = cur.next
        return '->'.join(res)

class Solution:
```

```python
    def reorderList(self, head: ListNode) -> None:
        """
        Do not return anything, modify head in-place instead.
        """
        if not head:
            return
        middle= self.find_middle(head)
        l2 = middle.next
        l1 = head
        middle.next = None
        l2 = self.reverse(l2)
        result = self.merge_linklist(l1, l2)

    @staticmethod
    def find_middle(head):
        slow =head
        fast = head
        while fast.next and fast.next.next:
            fast = fast.next.next
            slow = slow.next
        return slow

    @staticmethod
    def reverse(head):
        cur = head
        newnode = None
        while cur:
            tmp = cur.next
            cur.next = newnode
            newnode = cur
            cur = tmp
        return newnode

    @staticmethod
    def merge_linklist(head1, head2):
        while head1 and head2:
            tmp1 = head1.next
            tmp2 = head2.next

            head1.next = head2
            head2.next = tmp1

            head1 = tmp1
            head2 = tmp2
```

Tips

1. 这道题融合了三道题，快慢指针找中点，反转链表，合并两个链表

146. LRU 缓存机制.md

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制 。

实现 LRUCache 类：

```
LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存
int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。
```

进阶：你是否可以在 O(1) 时间复杂度内完成这两种操作?

示例：

输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]

解释
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);   // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);   // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);   // 返回 -1 (未找到)
lRUCache.get(3);   // 返回 3
lRUCache.get(4);   // 返回 4

提示：

```
1 <= capacity <= 3000
0 <= key <= 10000
0 <= value <= 105
最多调用 2 * 105 次 get 和 put
```

```
class ListNode:
```

```
    def __init__(self, key=0, val=0, next=None, prev=None):
        self.key = key
        self.val = val
        self.next = next
        self.prev = prev

class LRUCache:

    def __init__(self, capacity: int):
        self.capacity = capacity
        self.counter = 0
        self.dic = {}
        self.head = ListNode(-1,-1)
        self.tail = ListNode(-1,-1)
        self.head.next,self.tail.prev = self.tail, self.head

    def _insert(self, node):
        # insert的位置很明确永远在head之前
        node.next,node.prev = self.head.next, self.head
        self.head.next = node
        node.next.prev = node

    def _remove(self, node):
        # remove是任意位置的只要断开链接即可
        node.prev.next = node.next
        node.next.prev = node.prev

    def get(self, key: int) -> int:
        if key not in self.dic:
            return -1
        node = self.dic[key]
        self._remove(node)
        self._insert(node)
        return node.val

    def put(self, key: int, value: int) -> None:
        if key in self.dic:
            node = self.dic[key]
            self._remove(node)
            node.val = value
            self._insert(node)
        else:
            if self.counter == self.capacity:
                discard = self.tail.prev
                self._remove(discard)
                self.counter-=1
                del self.dic[discard.key]

            node = ListNode(key, value)
```

```
            self._insert(node)
            self.counter +=1
            self.dic[key] = node
```

Tips

1. LRU的设计采用双向链表+Hash的方案。链表的头部是最近访问的，尾部是不访问的。每次插入都判断capacity如果超出移除尾部加入头部。如果get则把node先删除再在头部插入
2. 链表删除节点：如果能拿到节点本身，直接断开双侧或者单侧的链接就可以
3. 链表插入节点：先把新节点对两侧的链接构建好，再构建两侧对中间的关联。需要注意不要使用修改后的link

147. 对链表进行插入排序.md

对链表进行插入排序。

插入排序的动画演示如上。从第一个元素开始，该链表可以被认为已经部分排序（用黑色表示）。
每次迭代时，从输入数据中移除一个元素（用红色表示），并原地将其插入到已排好序的链表中。

插入排序算法：

> 插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。
> 每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。
> 重复直到所有输入数据插入完为止。

示例 1：

输入: 4->2->1->3
输出: 1->2->3->4

示例 2：

输入: -1->5->3->4->0
输出: -1->0->3->4->5

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def insertionSortList(self, head: ListNode) -> ListNode:
        dummy = ListNode(-1)
```

```
        cur = dummy
        while head:
            #回到最开始重新搜索
            if head.val < cur.val:
                cur = dummy
            #找到head插入的位置
            while cur.next and head.val > cur.next.val:
                cur= cur.next
            cur.next, cur.next.next = ListNode(head.val), cur.next
            head = head.next
        return dummy.next
```

Tips

- 插入搜索：如果比当前位置大直接插入，如果小就会到链表的起点进行重新搜索
- 注意比较时只能比较next节点，如果比较当前节点无法插入

148. 排序链表.md

给你链表的头结点 head ， 请将其按 升序 排列并返回 排序后的链表 。

进阶：

```
你可以在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序吗?
```

示例 1：

输入：head = [4,2,1,3]
输出：[1,2,3,4]

示例 2：

输入：head = [-1,5,3,4,0]
输出：[-1,0,3,4,5]

示例 3：

输入：head = []
输出：[]

提示：

```
链表中节点的数目在范围 [0, 5 * 104] 内
-105 <= Node.val <= 105
```

1. 归并排序1 从top到bottom：空间复杂度O(logn),时间复杂度O(blogs)

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head or not head.next:
            return head
        left_end = self.find_mid(head)
        mid  = left_end.next
        left_end.next=None
        left = self.sortList(head)
        right = self.sortList(mid)
        return self.sort_merge(left, right)

    def find_mid(self, head):
        if not head or not head.next:
            return head
        slow = head
        fast = head
        while fast and fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next
        return slow

    def sort_merge(self, head1, head2):
        dummy = ListNode()
        tmp = dummy
        while head1 and head2:
            if head1.val < head2.val:
                tmp.next, head1 = head1, head1.next
            else:
                tmp.next,head2 = head2, head2.next
            tmp = tmp.next
        tmp.next = head1 if head1 else head2
        return dummy.next
```

从bottom到top的归并排序，从中间分割链表直到链表长度<=1, 然后进行排序merge

160. 相交链表.md

给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表没有交点，返回 null 。

图示两个链表在节点 c1 开始相交：

题目数据 保证 整个链式结构中不存在环。

注意，函数返回结果后，链表必须 保持其原始结构 。

示例 1：

输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Intersected at '8'
解释：相交节点的值为 8 （注意，如果两个链表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。
在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：

输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Intersected at '2'
解释：相交节点的值为 2 （注意，如果两个链表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。
在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：

输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。
由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
这两个链表不相交，因此返回 null 。

提示：

```
listA 中节点数目为 m
listB 中节点数目为 n
0 <= m, n <= 3 * 104
1 <= Node.val <= 105
0 <= skipA <= m
0 <= skipB <= n
如果 listA 和 listB 没有交点，intersectVal 为 0
如果 listA 和 listB 有交点，intersectVal == listA[skipA + 1] == listB[skipB + 1]
```

```python
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        def get_len(node):
            counter =0
            while node:
                node=node.next
                counter+=1
            return counter
        len_a = get_len(headA)
        len_b = get_len(headB)
        if len_a>len_b:
            for _ in range(len_a-len_b):
                headA = headA.next
        else:
            for _ in range(len_b-len_a):
                headB = headB.next

        while headA and headB:
            if headA==headB:
                return headA
            headA = headA.next
            headB = headB.next
        return None
```

Tips

1. 先补齐两个链表的长度然后直接遍历找相同节点即可
2.

1721. 交换链表中的节点.md

给你链表的头节点 head 和一个整数 k 。

交换 链表正数第 k 个节点和倒数第 k 个节点的值后，返回链表的头节点（链表 从 1 开始索引）。

示例 1：

输入：head = [1,2,3,4,5], k = 2
输出：[1,4,3,2,5]

示例 2：

输入：head = [7,9,6,6,7,8,3,0,9,5], k = 5
输出：[7,9,6,6,8,7,3,0,9,5]

示例 3：

输入：head = [1], k = 1
输出：[1]

示例 4：

输入：head = [1,2], k = 1
输出：[2,1]

示例 5：

输入：head = [1,2,3], k = 2
输出：[1,2,3]

提示：

```
链表中节点的数目是 n
1 <= k <= n <= 105
0 <= Node.val <= 100
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def swapNodes(self, head: ListNode, k: int) -> ListNode:
        p1 = head
        p0 = head
        for i in range(k-1):
            p1 = p1.next

        p2 = p1
        while p1.next:
            p0 = p0.next
            p1 = p1.next

        p2.val, p0.val = p0.val, p2.val
        return head
```

Tips

1. 这道题用到寻找链表倒数第N个节点，正数第N个节点，以及交换节点值

19. 删除链表的倒数第 N 个结点.md

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。

进阶：你能尝试使用一趟扫描实现吗?

示例 1：

输入：head = [1,2,3,4,5], n = 2
输出：[1,2,3,5]

示例 2：

输入：head = [1], n = 1
输出：[]

示例 3：

输入：head = [1,2], n = 1
输出：[1]

提示：

```
链表中结点的数目为 sz
1 <= sz <= 30
0 <= Node.val <= 100
1 <= n <= sz
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        dummy = ListNode(next=head)
        tmp1 = dummy
        tmp2 = dummy
        for i in range(n):
            tmp1 = tmp1.next

        while tmp1.next:
            tmp1 = tmp1.next
            tmp2 = tmp2.next
        tmp2.next = tmp2.next.next
        return dummy.next
```

Tips

1. 可类比用快慢指针找链表中间点进行链表翻转
2. 需要用到dumynode，因为可能会删掉第一个节点
3. 初始化两个指针指向dummy，然后让第一个和第二个指针拉开n个距离，再同时开始遍历，这样当快指针到头，慢指针到倒数第n+1个，然后直接跳过倒数第n个节点就好

2. 两数相加.md

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1：

输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[7,0,8]
解释：342 + 465 = 807.

示例 2：

输入：l1 = [0], l2 = [0]
输出：[0]

示例 3：

输入：l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
输出：[8,9,9,9,0,0,0,1]

提示：

```
每个链表中的节点数在范围 [1, 100] 内
0 <= Node.val <= 9
题目数据保证列表表示的数字不含前导零
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        remainder = 0
        node = ListNode()
```

```
        ptr = node
        def helper(l1, l2):
            if l1 and l2:
                return l1.val + l2.val
            elif l1:
                return l1.val
            else:
                return l2.val

        while l1 or l2:
            val = helper(l1,l2) + remainder
            node.next = ListNode( val %10 )
            remainder = val//10
            node = node.next
            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next
        if remainder>0:
            node.next = ListNode(remainder)
        return ptr.next
```

Tips:

1. 注意链表的赋值一般都是对next，这样可以有效避免创建空节点。如果每次val都是赋值给当前节点，则需要额外判断l1和l2是否为空决定是否创建链表的next节点

203. 移除链表元素.md

给你一个链表的头节点 head 和一个整数 val ，请你删除链表中所有满足 Node.val == val 的节点，并返回 新的头节点 。

示例 1：

输入：head = [1,2,6,3,4,5,6], val = 6
输出：[1,2,3,4,5]

示例 2：

输入：head = [], val = 1
输出：[]

示例 3：

输入：head = [7,7,7,7], val = 7
输出：[]

提示：

```
列表中的节点数目在范围 [0, 104] 内
1 <= Node.val <= 50
0 <= val <= 50
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def removeElements(self, head: ListNode, val: int) -> ListNode:
        dummy = ListNode(next=head)
        cur = dummy

        while dummy.next:
            if dummy.next.val==val:
                dummy.next = dummy.next.next
            else:
                dummy = dummy.next
        return cur.next
```

Tips

1. 考虑到链表的特性，被移除的节点只能是next不能是current，所以每一步判断都在判断next
2. 因为head也可能为val，所以需要创建Dummny head，指向head，来保证head也可以作为next被移除
3. 且只有当next！=val的时候才向前移动，避免新创建的next依旧等于val。这个部分和移除链表中的重复元素相同

206. 反转链表.md

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：

输入：head = [1,2]
输出：[2,1]

示例 3：

输入：head = []
输出：[]

提示：

```
链表中节点的数目范围是 [0, 5000]
-5000 <= Node.val <= 5000
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        newnode = None
        cur=head
        while cur:
            tmp =cur.next
            cur.next = newnode
            newnode = cur
            cur=tmp
        return newnode
```

Tips

1. A->B->C的反转拿到A，保留A->B, 让A->None, 移到B，让B指向 A->None

21. 合并两个有序链表.md

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1：

输入：l1 = [1,2,4], l2 = [1,3,4]
输出：[1,1,2,3,4,4]

示例 2：

输入：l1 = [], l2 = []
输出：[]

示例 3：

输入：l1 = [], l2 = [0]
输出：[0]

提示：

```
两个链表的节点数目范围是 [0, 50]
-100 <= Node.val <= 100
l1 和 l2 均按 非递减顺序 排列
```

1.

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        l = ListNode(-1)
        tmp = l
        while l1 and l2:
            if l1.val <= l2.val:
                tmp.next = l1
                l1 = l1.next
            else:
                tmp.next = l2
                l2 = l2.next
            tmp = tmp.next

        if l1:
            tmp.next = l1
        if l2:
            tmp.next = l2

        return l.next
```

2.

Tips

1. 对于需要对当前数值进行判断都链表任务都需要加dummy head，因为链表只能对next进行处理，方便最后返回的时候返回表头
2. 链表运算的每一步都是定义当前val，以及next的pointer，然后向后移一步。next的定义可以是直接初始化一只新的ListNode(0),或者直接传其他链表。
3. 如果是传值需要考虑，l1.next or l2.next判断是否创建next节点，避免生成多余节点
4.

23. 合并K个升序链表.md

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

输入：lists = [[1,4,5],[1,3,4],[2,6]]
输出：[1,1,2,3,4,4,5,6]
解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6

示例 2：

输入：lists = []
输出：[]

示例 3：

输入：lists = [[]]
输出：[]

提示：

```
k == lists.length
0 <= k <= 10^4
0 <= lists[i].length <= 500
-10^4 <= lists[i][j] <= 10^4
lists[i] 按 升序 排列
lists[i].length 的总和不超过 10^4
```

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        if not lists:
```

```
            return
        def divide_conquer(start,end):
            if start==end:
                return lists[start]
            mid = (start+end)//2
            left = divide_conquer(start, mid)
            right = divide_conquer(mid+1, end)
            return sort_merge(left, right)

        def sort_merge(l1, l2):
            newnode = ListNode()
            ptr = newnode
            while l1 and l2:
                if l1.val<l2.val:
                    ptr.next = ListNode(l1.val)
                    l1 = l1.next
                else:
                    ptr.next = ListNode(l2.val)
                    l2 = l2.next
                ptr = ptr.next
            if l1:
                ptr.next = l1
            if l2:
                ptr.next = l2
            return newnode.next

        return divide_conquer(0, (len(lists)-1))
```

Tips

1. sort_merge合并两个链表
2. divide and conquer 递归两两合并链表，输入是index，返回是合并后的链表

234. 回文链表.md

给你一个单链表的头节点 head ，请你判断该链表是否为回文链表。如果是，返回 true ；否则，返回 false 。

示例 1：

输入：head = [1,2,2,1]
输出：true

示例 2：

输入：head = [1,2]
输出：false

提示：

```
链表中节点数目在范围[1, 105] 内
0 <= Node.val <= 9
```

1. O(n)的时间复杂度，和空间复杂度

```python
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        res = []
        while head:
            res.append(head.val)
            head = head.next
        return res == res[::-1]
```

2. O(1)的空间复杂度: 包括两个部分，快慢链表用于找到链表的中间点，然后反转链表反转后半部分

```python
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        def find_second_half(head):
            slow = head
            fast = head
            while fast and fast.next:
                slow = slow.next
                fast = fast.next.next
            return slow

        def reverse_linklist(head):
            newnode = None
            while head:
                tmp = head.next
                head.next = newnode
                newnode = head
                head = tmp
            return newnode
        slow = find_second_half(head)
        right = reverse_linklist(slow)
        while head and right:
            if head.val != right.val:
                return False
            else:
                head = head.next
                right = right.next
        return True
```

237. 删除链表中的节点.md

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点。传入函数的唯一参数为 要被删除的节点 。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:

示例 1：

输入：head = [4,5,1,9], node = 5
输出：[4,1,9]
解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.

示例 2：

输入：head = [4,5,1,9], node = 1
输出：[4,5,9]
解释：给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.

提示：

链表至少包含两个节点。
链表中所有节点的值都是唯一的。
给定的节点为非末尾节点并且一定是链表中的一个有效节点。
不要从你的函数中返回任何结果。

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def deleteNode(self, node):
        """
        :type node: ListNode
        :rtype: void Do not return anything, modify node in-place instead.
        """
        node.val = node.next.val
        node.next = node.next.next
```

Tips

这题有坑～～～～看输入就会发现这题是让你实现一个类似helper的function

因为链表的特殊性，是永远无法删除自身的，所以在之前删除重复元素/删除=val的元素中，都是对next进行判断。所以这里并没有删除自身，而是直接把当前node，替换成next node，然后把next后面的部分接在了当前node上，其实还是删掉了next

24. 两两交换链表中的节点.md

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1：

输入：head = [1,2,3,4]
输出：[2,1,4,3]

示例 2：

输入：head = []
输出：[]

示例 3：

输入：head = [1]
输出：[1]

提示：

```
链表中节点的数目在范围 [0, 100] 内
0 <= Node.val <= 100
```

1. 直接修改node value，这个很简单直接swap，然后每次向后移两位就好

```python
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        tmp = head
        while tmp and tmp.next:
            tmp.val, tmp.next.val = tmp.next.val, tmp.val
            tmp = tmp.next.next
        return head
```

2. 如果修改link的话，因为linklist只能修改+1的link所以需要创建dummy

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        dummy = ListNode(next=head)
        tmp = dummy
        while tmp.next and tmp.next.next:
            node1 = tmp.next
            node2 = tmp.next.next
            node1.next=node2.next
            node2.next = node1
            tmp.next = node2
            tmp = tmp.next.next
        return dummy.next
```

Tips

1. 0->1->2->3，站在位置0，先把1和2拎出来，link1—>3, link 2—1，link0->2, 跳到3
2. 考虑到linklist只能修改next的特点，以及每次修改两个位置，所以需要创建dummynode，站在dummy修改0和1，然后站在1修改2和3。如果允许修改节点值本身当然是可以不用创建dummy
3. 对于需要修改后面两个节点的操作，在每一步判断时需要同时判断tmp.next & tmp.next.next

328. 奇偶链表.md

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

示例 1:

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

示例 2:

输入: 2->1->3->5->6->4->7->NULL
输出: 2->3->6->7->1->5->4->NULL

说明:

> 应当保持奇数节点和偶数节点的相对顺序。
> 链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def oddEvenList(self, head: ListNode) -> ListNode:
        if not head:
            return
        even = head.next
        odd = head
        evenhead = even
        oddhead = odd
        while even and even.next:
            odd.next = even.next
            odd = odd.next
            even.next = odd.next
            even = even.next

        odd.next = evenhead
        return head
```

Tips

1. 先分离奇偶，再把偶数频道奇数后边
2. 这里时间复杂度是O(n)，空间复杂度是O(1)因为只改变；了link并没有创建新的节点

382. 链表随机节点.md

给定一个单链表，随机选择链表的一个节点，并返回相应的节点值。保证每个节点被选的概率一样。

进阶:
如果链表十分大且长度未知，如何解决这个问题？你能否使用常数级空间复杂度实现？

示例:

// 初始化一个单链表 [1,2,3].
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);
Solution solution = new Solution(head);

// getRandom()方法应随机返回1,2,3中的一个，保证每个元素被返回的概率相等。
solution.getRandom();

```python
class Solution:
```

```
    def __init__(self, head: Optional[ListNode]):
        self.head = head

    def getRandom(self) -> int:
        count = 0
        reserve = None
        cur = self.head
        while cur:
            count +=1
            rand = random.randint(1, count )
            if rand == count:
                reserve = cur.val
            cur = cur.next
        return reserve
```

Tips

蓄水池抽样

N个候选，第K个候选被采用的概率是1/K,

- K=1， p=1
- K=2，p=1/2，覆盖K=1的概率是1/2
- K=3, p=1/3, 覆盖之前的概率是2/3，所以K=2的元素被选择的概率变成1/2 * 2/3 = 1/3
-

460. LFU 缓存.md

请你为 最不经常使用（LFU）缓存算法设计并实现数据结构。

实现 LFUCache 类：

```
LFUCache(int capacity) - 用数据结构的容量 capacity 初始化对象
int get(int key) - 如果键存在于缓存中，则获取键的值，否则返回 -1。
void put(int key, int value) - 如果键已存在，则变更其值；如果键不存在，请插入键值对。当缓存达到其容量时，则应该在插入新项之前，使最不经常使用的项无效。在此问题中，当存在平局（即两个或更多个键具有相同使用频率）时，应该去除 最近最久未使用 的键。
```

注意「项的使用次数」就是自插入该项以来对其调用 get 和 put 函数的次数之和。使用次数会在对应项被移除后置为 0 。

为了确定最不常使用的键，可以为缓存中的每个键维护一个 使用计数器 。使用计数最小的键是最久未使用的键。

当一个键首次插入到缓存中时，它的使用计数器被设置为 1 (由于 put 操作)。对缓存中的键执行 get 或 put 操作，使用计数器的值将会递增。

示例：

输入：
["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
输出：
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

解释：
// cnt(x) = 键 x 的使用计数
// cache=[] 将显示最后一次使用的顺序（最左边的元素是最近的）
LFUCache lFUCache = new LFUCache(2);
lFUCache.put(1, 1);   // cache=[1,_], cnt(1)=1
lFUCache.put(2, 2);   // cache=[2,1], cnt(2)=1, cnt(1)=1
lFUCache.get(1);   // 返回 1
                      // cache=[1,2], cnt(2)=1, cnt(1)=2
lFUCache.put(3, 3);   // 去除键 2 ，因为 cnt(2)=1 ，使用计数最小
                      // cache=[3,1], cnt(3)=1, cnt(1)=2
lFUCache.get(2);   // 返回 -1（未找到）
lFUCache.get(3);   // 返回 3
                      // cache=[3,1], cnt(3)=2, cnt(1)=2
lFUCache.put(4, 4);   // 去除键 1 ， 1 和 3 的 cnt 相同，但 1 最久未使用
                      // cache=[4,3], cnt(4)=1, cnt(3)=2
lFUCache.get(1);   // 返回 -1（未找到）
lFUCache.get(3);   // 返回 3
                      // cache=[3,4], cnt(4)=1, cnt(3)=3
lFUCache.get(4);   // 返回 4
                      // cache=[3,4], cnt(4)=2, cnt(3)=3

提示：

```
0 <= capacity, key, value <= 104
最多调用 105 次 get 和 put 方法
```

进阶：你可以为这两种操作设计时间复杂度为 O(1) 的实现吗?

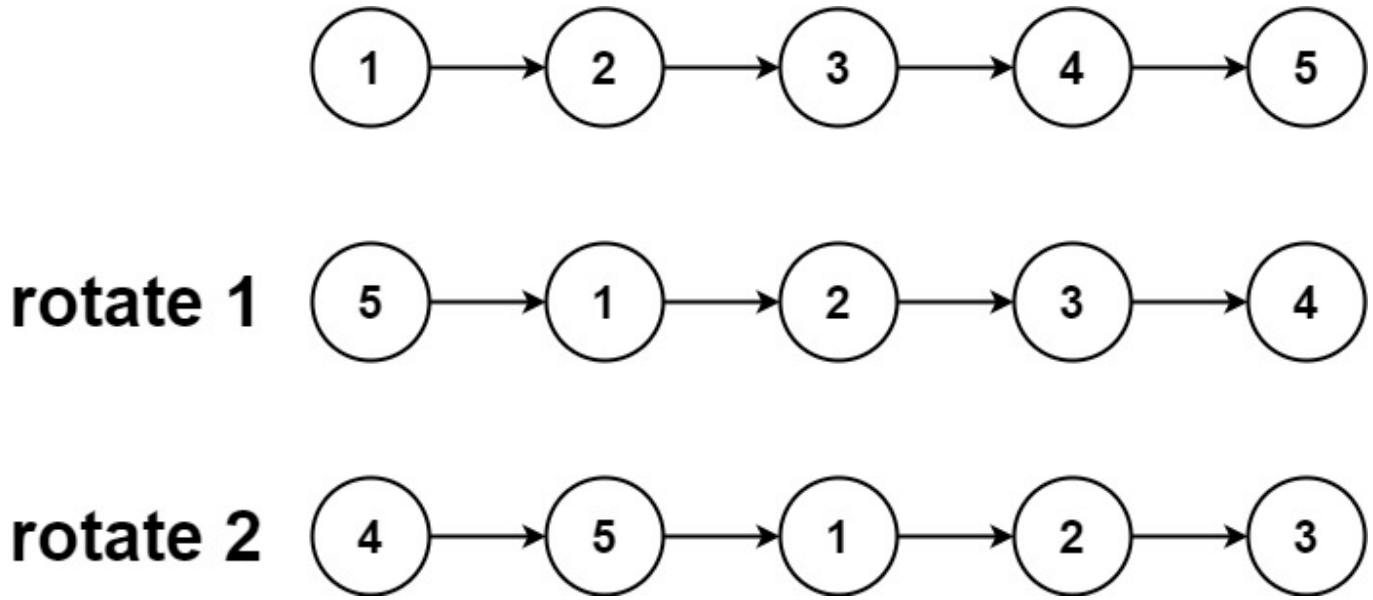
```

```

61. 旋转链表.md

####

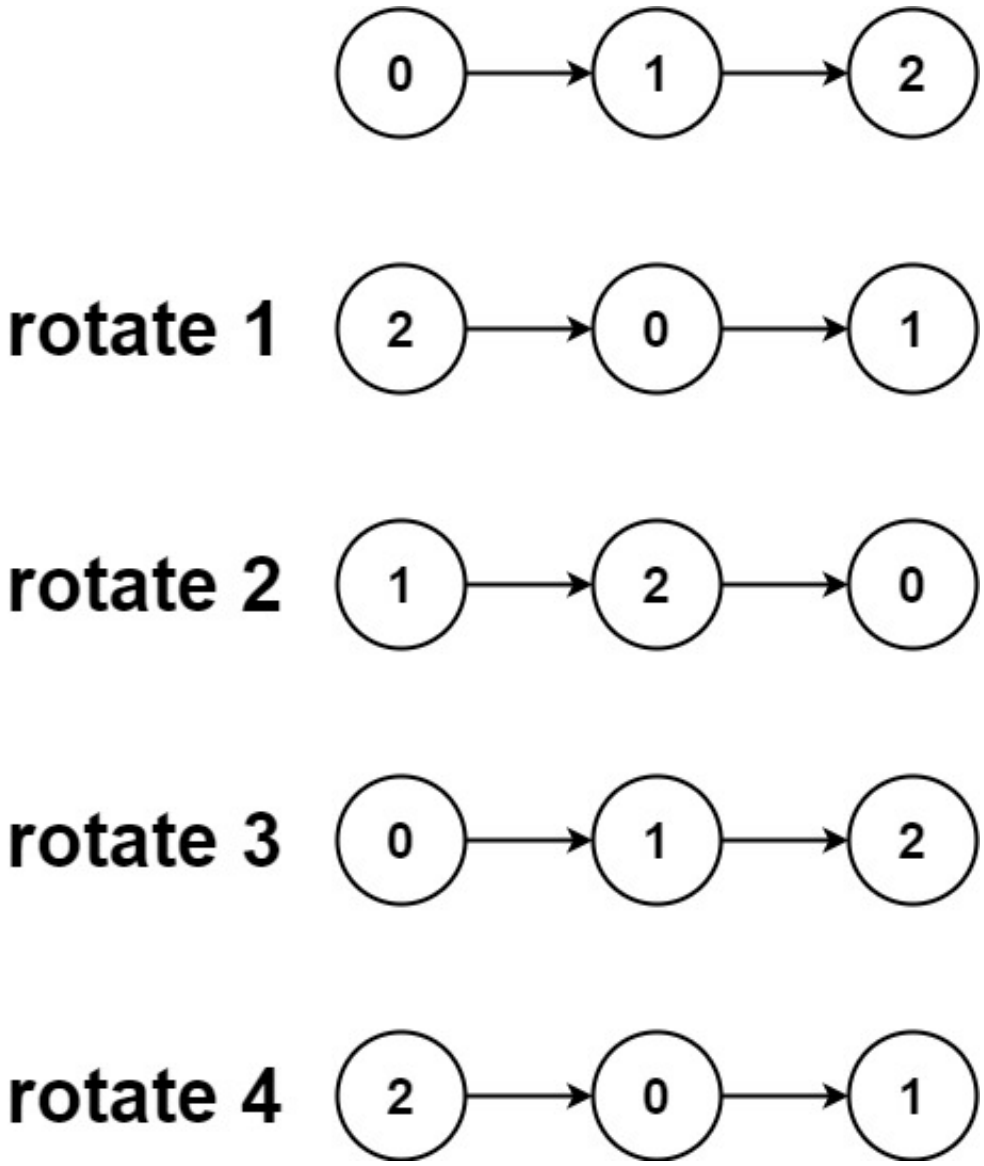
给你一个链表的头节点 `head` ，旋转链表，将链表每个节点向右移动 `k` 个位置。

示例 1：




```
输入：head = [1,2,3,4,5], k = 2
输出：[4,5,1,2,3]
```

示例 2：




```
输入：head = [0,1,2], k = 4
输出：[2,0,1]
```

提示：

- 链表中节点的数目在范围 `[0, 500]` 内
- `-100 <= Node.val <= 100`
- `0 <= k <= 2 * 109`

1. 技巧解法

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
```

```
        if k==0 or not head or not head.next:
            return head

        n = 1
        cur = head
        #得到最末尾的cur
        while cur.next:
            cur = cur.next
            n+=1

        if k%n ==0:
            return head
        k = n- k%n

        cur.next = head
        for i in range(k):
            cur = cur.next

        newhead = cur.next
        cur.next = None
        return newhead
```

Tips

链表题永远是要一个一个掰着手指头数。。。

1. 第一步统计链表长度，并让链表停止在最后一个节点。所以是while cur.next，以及计数从1开始
2. 把链表的尾巴和头部连起来，得到环形链表
3. k%n找到余数，也就是在倒数第几个点要断开环
4. 然后向前遍历到应该断开的位置，得到新的head，并断开链表

2. 常规解法：需要多遍历一次链表

```
class Solution:
    def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        if k==0 or not head or not head.next:
            return head

        n = 1
        cur = head
        #得到最末尾的cur
        while cur.next:
            cur = cur.next
            n+=1

        slow = head
        fast = head
        k = k%n
```

```
        if k ==0:
            return head

        for i in range(k):
            fast = fast.next

        while fast.next:
            slow = slow.next
            fast = fast.next
        newhead = slow.next
        slow.next = None


        tmp = newhead
        while tmp.next:
            tmp = tmp.next
        tmp.next = head
        return newhead
```

通过双指针找到倒数第K的元素，断开链接，把后面的部分都拼接到head之前

707. 设计链表.md

设计链表的实现。您可以选择使用单链表或双链表。单链表中的节点应该具有两个属性：val 和 next。val 是当前节点的值，next 是指向下一个节点的指针/引用。如果要使用双向链表，则还需要一个属性 prev 以指示链表中的上一个节点。假设链表中的所有节点都是 0-index 的。

在链表类中实现这些功能：

```
get(index)：获取链表中第 index 个节点的值。如果索引无效，则返回-1。
addAtHead(val)：在链表的第一个元素之前添加一个值为 val 的节点。插入后，新节点将成为链表的第一个节点。
addAtTail(val)：将值为 val 的节点追加到链表的最后一个元素。
addAtIndex(index,val)：在链表中的第 index 个节点之前添加值为 val  的节点。如果 index 等于链表的长度，则该节点将附加到链表的末尾。如果 index 大于链表长度，则不会插入节点。如果index小于0，则在头部插入节点。
deleteAtIndex(index)：如果索引 index 有效，则删除链表中的第 index 个节点。
```

示例：

MyLinkedList linkedList = new MyLinkedList();
linkedList.addAtHead(1);
linkedList.addAtTail(3);
linkedList.addAtIndex(1,2); //链表变为1-> 2-> 3
linkedList.get(1);   //返回2

linkedList.deleteAtIndex(1); //现在链表是1-> 3
linkedList.get(1);   //返回3

提示：

```
所有val值都在 [1, 1000] 之内。
操作次数将在  [1, 1000] 之内。
请不要使用内置的 LinkedList 库。
```

```python
class Node():
    def __init__(self, val):
        self.val = val
        self.next = None

    def __str__(self):
        cur = self
        res =[]
        while cur is not None:
            res.append(str(cur.val))
            cur = cur.next
        return '->'.join(res)

class MyLinkedList:

    def __init__(self):
        self.head = Node(0) # dummy head
        self.count =0

    def get(self, index: int) -> int:
        if 0<=index < self.count:
            cur = self.head
            for i in range(index+1):
                cur = cur.next
            return cur.val
        else:
            return -1

    def addAtHead(self, val: int) -> None:
        self.addAtIndex(0,val)

    def addAtTail(self, val: int) -> None:
        self.addAtIndex(self.count, val)

    def addAtIndex(self, index: int, val: int) -> None:
        #print(self.head)
```

```
        if index> self.count:
            return
        elif index<0:
            index =0
        cur = self.head
        for _ in range(index):
            cur = cur.next
        add_node = Node(val)
        tmp = cur.next
        add_node.next = tmp
        cur.next = add_node
        self.count+=1

    def deleteAtIndex(self, index: int) -> None:
        if 0<=index < self.count:
            cur = self.head
            for i in range(index):
                cur = cur.next
            cur.next = cur.next.next
            self.count-=1
        else:
            return



# Your MyLinkedList object will be instantiated and called as such:
# obj = MyLinkedList()
# param_1 = obj.get(index)
# obj.addAtHead(val)
# obj.addAtTail(val)
# obj.addAtIndex(index,val)
# obj.deleteAtIndex(index)
```

Tips

1. 注意dummy head的设置，在链表操作中一般有插入需要都会预留dummy head， 保证在插入头部的时候还是用next来进行插入

725. 分隔链表.md

给你一个头结点为 head 的单链表和一个整数 k ，请你设计一个算法将链表分隔为 k 个连续的部分。

每部分的长度应该尽可能的相等：任意两部分的长度差距不能超过 1 。这可能会导致有些部分为 null 。

这 k 个部分应该按照在链表中出现的顺序排列，并且排在前面的部分的长度应该大于或等于排在后面的长度。

返回一个由上述 k 部分组成的数组。

示例 1：

输入：head = [1,2,3], k = 5
输出：[[1],[2],[3],[],[]]
解释：
第一个元素 output[0] 为 output[0].val = 1 ，output[0].next = null 。
最后一个元素 output[4] 为 null ，但它作为 ListNode 的字符串表示是 [] 。

示例 2：

输入：head = [1,2,3,4,5,6,7,8,9,10], k = 3
输出：[[1,2,3,4],[5,6,7],[8,9,10]]
解释：
输入被分成了几个连续的部分，并且每部分的长度相差不超过 1 。前面部分的长度大于等于后面部分的长度。

提示：

```
链表中节点的数目在范围 [0, 1000]
0 <= Node.val <= 1000
1 <= k <= 50
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def splitListToParts(self, head: ListNode, k: int) -> List[ListNode]:
        res = []
        l = 0
        cur = head
        while head:
            head = head.next
            l +=1
        ll, remainder = l//k, l%k
        res = [None for i in range(k)]
        i=0
        while cur and i<k:
            if i < remainder:
                l_tmp = ll+1
            else:
                l_tmp = ll

            res[i] = cur
            for j in range(l_tmp-1):
                cur = cur.next
```

```
            nextcur = cur.next
            cur.next = None
            cur = nextcur
            i+=1
        return res
```

82. 删除排序链表中的重复元素 II.md

存在一个按升序排列的链表，给你这个链表的头节点 head ，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中 没有重复出现 的数字。

返回同样按升序排列的结果链表。

示例 1：

输入：head = [1,2,3,3,4,4,5]
输出：[1,2,5]

示例 2：

输入：head = [1,1,1,2,3]
输出：[2,3]

提示：

```
链表中节点数目在范围 [0, 300] 内
-100 <= Node.val <= 100
题目数据保证链表已经按升序排列
```

```
class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        dummy = ListNode(-1)
        dummy.next = head
        cur = dummy
        while cur.next and cur.next.next:
            if cur.next.val == cur.next.next.val:
                #当存在重复元素的时候只改变link不宜动当前指针
                x = cur.next.val
                while cur.next and cur.next.val==x:
                    cur.next = cur.next.next
            else:
                #只有当后续两个元素不重复的时候才移动当前指针
                cur= cur.next

        return dummy.next
```

Tips

1. 永远机制链表不能对当前节点进行修改，只能修改next
2. 这里因为要判断连续两个元素是否重复，所以要保留两个指针位置node.next和node.next.next
3. 且当发现存在重复元素师不要移动指针位置，只是继续向前判断知道发现不重复的元素，改变link，然后继续判断。只有当确认玄素不重复的时候才移动指针位置

83. 删除排序链表中的重复元素.md

存在一个按升序排列的链表，给你这个链表的头节点 head ，请你删除所有重复的元素，使每个元素 只出现一次 。

返回同样按升序排列的结果链表。

示例 1：

输入：head = [1,1,2]
输出：[1,2]

示例 2：

输入：head = [1,1,2,3,3]
输出：[1,2,3]

提示：

```
链表中节点数目在范围 [0, 300] 内
-100 <= Node.val <= 100
题目数据保证链表已经按升序排列
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        if not head:
            return head
        cur = head
        while head.next:
            if head.val == head.next.val:
                head.next=head.next.next
            else:
```

```
                head=head.next

        return cur
```

Tips

1. 链表注意事项永远记得保留head
2. 每一步都是调整当前节点的link，所以head=head.next一定要放在后面做不然pointer就已经离开当前节点了

86. 分隔链表.md

给你一个链表的头节点 head 和一个特定值 x ，请你对链表进行分隔，使得所有 小于 x 的节点都出现在 大于或等于 x 的节点之前。

你应当 保留 两个分区中每个节点的初始相对位置。

示例 1：

输入：head = [1,4,3,2,5,2], x = 3
输出：[1,2,2,4,3,5]

示例 2：

输入：head = [2,1], x = 2
输出：[1,2]

提示：

```
链表中节点的数目在范围 [0, 200] 内
-100 <= Node.val <= 100
-200 <= x <= 200
```

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def partition(self, head: ListNode, x: int) -> ListNode:

        small = ListNode(-1)
        ptr_small = small
        large = ListNode(-1)
        ptr_large = large
```

```
        while head:
            if head.val < x:
                small.next = head
                small = small.next
            else:
                large.next = head
                large = large.next
            head = head.next

        small.next = ptr_large.next
        large.next = None
        return ptr_small.next
```

876. 链表的中间结点.md

给定一个头结点为 head 的非空单链表，返回链表的中间结点。

如果有两个中间结点，则返回第二个中间结点。

示例 1：

输入：[1,2,3,4,5]
输出：此列表中的结点 3 (序列化形式：[3,4,5])
返回的结点值为 3 。 (测评系统对该结点序列化表述是 [3,4,5])。
注意，我们返回了一个 ListNode 类型的对象 ans，这样：
ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, 以及 ans.next.next.next = NULL.

示例 2：

输入：[1,2,3,4,5,6]
输出：此列表中的结点 4 (序列化形式：[4,5,6])
由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。

提示：

```
给定链表的结点数介于 1 和 100 之间。
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        slow = head
        fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        return slow
```

快慢指针找中点，注意fast的判断条件

92. 反转链表 II.md

给你单链表的头指针 head 和两个整数 left 和 right ，其中 left <= right 。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表 。

示例 1：

输入：head = [1,2,3,4,5], left = 2, right = 4
输出：[1,4,3,2,5]

示例 2：

输入：head = [5], left = 1, right = 1
输出：[5]

提示：

```
链表中节点数目为 n
1 <= n <= 500
-500 <= Node.val <= 500
1 <= left <= right <= n
```

进阶： 你可以使用一趟扫描完成反转吗?

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def reverseBetween(self, head: ListNode, left: int, right: int) -> ListNode:
        def reverse(head):
            newnode = None
            cur = head
            while cur:
                tmp =cur.next
                cur.next = newnode
                newnode = cur
                cur = tmp

        dummy = ListNode(next=head)
        cur = dummy
        for i in range(left-1):
            cur = cur.next

        leftnode = cur.next
        rightnode = leftnode
        for i in range(right-left):
            rightnode = rightnode.next

        rest = rightnode.next
        #把中间部分截出来
        rightnode.next=None
        cur.next = None

        reverse(leftnode)

        cur.next = rightnode
        leftnode.next = rest
        return dummy.next
```

1. 基本上medium的链表题都是多个easy的组合，这道题用到寻找链表中第K个节点，以及反转链表
2. 找到left和right以及left左边，和right右边的节点。断开连接把[left, right]隔离出来
3. 反转[left, right]，注意这里的反转是原地反转zhigaibianlink所以不需要返回，如果返回会占用额外内存创建新的节点

剑指 Offer 06. 从尾到头打印链表.md

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

输入：head = [1,3,2]
输出：[2,3,1]

限制：

0 <= 链表长度 <= 10000

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def reversePrint(self, head: ListNode) -> List[int]:
        res = []
        while head:
            res.append(head.val)
            head = head.next
        return res[::-1]
```

python的解法永远到没朋友，当然如果你想用栈，用递归也没人拦着你。。。。