Reverse Linked List

链表与递归也是大有渊源
虚拟节点 dummy 通常在处理特殊情
（例如头节点的处理）或在删除节点
用于维护链表的结构。但在反转链表
情况下，我们只需遍历原始链表，将
节点逐个移到新链表的头部，
不需要特殊的头节点处理。
因此，没有必要引入虚拟节点 dumm

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


class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        prev, curr = None, head

        while curr:
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp
        return prev
#记住 停止条件是在curr ==null，也就是说
prev是最后一个
```

指向下一个的指针
指向前面

这个过程中
就反转

recursive是回溯

"备下一"
"反指头"
进：

```
2      def reverseList(head):
3          if not head:
4              return None
5          newHead = head
6          if head.next:
7              newHead = self.reverseList(head.next)
8              head.next.next = head
9          head.next = None
10         return newHead
```

那后面的newHead = self.reverseList(head.next)开始回溯的时候都是从下面开始的
recursive call的执行都是7 包扩上面的执行

"备下一" (Prepare the next): temp = curr.next - Store the next node.
"反指头" (Reverse pointer): curr.next = prev - Point the current node back to the previous node.
进： prev = curr
        curr = temp
curr.next是指针

Base Return: 到底none
Dive Deep: 深入潜
Reverse Point: 指向反
Detach Node: 节点断

```
def reverseList(self, head: ListNode) -> ListNode:
    if not head:
        return none
    newHead = head
    if head.next:
        newHead = self.reverseList(head. 所以后面的head其实反转后的尾部？)
        如果当前节点有下一个节点，我们递归地对其进行反转，并且获取反转后链表的头节点，这是最关键的部分。因为当链表的最后两个节点反转时，最后一个节点就成了新的头节点。
        head.next.next = head
        //假设当前节点是A，它的下一个节点是B。这行代码实际上就是让B指向A，从而反转链表的方向。
    head.next = None
    return newHead
```

Merge Two Sorted Lists
想象很重要在这

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one sorted list.

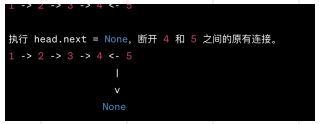
```
# Iterative
class Solution:
    def mergeTwoLists(self, list1: ListNode, list2: ListNode) -> ListNode:
        dummy = node = ListNode()

        while list1 and list2:
            if list1.val < list2.val:
                node.next = list1
                list1 = list1.next
            else:
                node.next = list2
                list2 = list2.next
            node = node.next

        node.next = list1 or list2

        return dummy.next
```

哨兵节点: 用一个哨兵(dummy)节点开始。
比较&连接: 比较两个链表的值，并将节点连接到新链表。
剩余连接: 一个链表为空后，将另一个链表的剩余部分连接上。

返回非空: 如果其中一个链表为空，返回另一个。
小值领先: 确定两个链表中哪一个的当前节点值较小。tuple课 做到
递归合并: 将小的节点与后续递归合并的结果连接起来。

```
# Recursive
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
        if not list1:
            return list2
        if not list2:
            return list1
        lil, big = (list1, list2) if list1.val < list2.val else (list2, list1)
        lil.next = self.mergeTwoLists(lil.next, big) #因为是个递归函数如果一个list没有，就到notlist了
        return lil
```

"空返": 当一个链表为空时，直接返回另一个。
"小选": 选择两个链表中的较小值为当前节点。
"递进": 递归调用自己，进入下一层，处理下一个节点。
"头返": 最终，返回合并后链表的头节点。



指针永远是最后一个 next，而节点永远是倒数第二个包括之前

isPalindrome
Given the head of a singly linked list, return true if it is a palindrome
 or false otherwise.

```
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        fast = head
        slow = head

        # find the middle (slow)
        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next

        # reverse second half
        prev = None
        while slow:
            tmp = slow.next
            slow.next = prev
            prev = slow
            slow = tmp

        # check palindrome
        left, right = head, prev
        while right:
            if left.val != right.val:
                return False
            left = left.next
            right = right.next
        return True
```

快慢找中: 快指针每次移动两步，慢指针每次移动一步，当快指针到达末尾时，慢指针正好在中间。
反转半边: 反转链表的后半部分。
双指对比: 从头和中间开始，同时遍历链表，比较值是否相同

1 -> 2 -> 3 <- 4 <- 5像这种，这样反转，第二个部分比第一个部分多一个3

Remove Linked List Elements

```
class Solution:
    def removeElements(self, head: ListNode, val: int) -> ListNode:
        # 创建一个虚拟节点，并将其 next 指向原始链表的头节点
        dummy = ListNode(next=head)
        # 初始化 prev 和 curr，用于遍历链表
        prev, curr = dummy, head

        while curr:
            nxt = curr.next  # 获取当前节点的下一个节点

            if curr.val == val:
                # 如果当前节点的值等于要删除的值，将 prev 的 next 指向 n
                prev.next = nxt
            else:
                # 否则，更新 prev 为当前节点，继续遍历
                prev = curr

            curr = nxt  # 移动到下一个节点

        # 返回虚拟节点的 next，即处理后的链表
        return dummy.next
```

需要特殊处理头节点的删除，而使用虚拟节点可以将头节点的删除操作与其他节点的删除操作一致地处理。

如果不使用 nxt 变量，删除当前节点后就无法访问下一个节点，会导致遍历链表中断。

Remove Duplicates From Sorted List

```
class Solution:
    def deleteDuplicates(self, head: Optional[ListNode]) -> Optional[ListNode]:
        cur = head
        while cur:
            while cur.next and cur.next.val == cur.val:
                cur.next = cur.next.next
            cur = cur.next
        return head
```

这种方法不需要使用虚拟节点，因为它只需要修改指针来跳过重复节点，而不涉及创建新的节点或修改节点的值

之所以没有显式使用额外的 nxt 变量，是因为循环结构允许直接使用 cur.next 来访问下一个节点，
并在需要时修改 cur.next 来实现删除重复节点的目的。代码中的 cur.next 就是下一个节点，可以直接使用它来访问下一个节点的值和指针。

如果链表的头节点重复，这段代码仍然有效。代码的内层循环会检查当前节点的下一个节点是否与当前节点相同，如果相同，它会将当前节点的 next 指针指向下下个节点，从而跳过重复的头节点。

Intersection of Two Linked Lists

这个是双指针
l1,l2各指向一个头但是！！
期中一个指到null, 的时候
就会跑到另一个头上面(else
headB)去
他们这个，相遇的时候，
while L1 ==L2开始往后都是
两个listnode的 intersection

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


class Solution:
    def getIntersectionNode(
        self, headA: ListNode, headB: ListNode
    ) -> Optional[ListNode]:
        l1, l2 = headA, headB
        while l1 != l2:
            l1 = l1.next if l1 else headB
            l2 = l2.next if l2 else headA
        return l1
```

Reorder List

You are given the head of a singly linked-list. The list can be represented as:

L0 → L1 → … → Ln - 1 → Ln
Reorder the list to be on the following form:

L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …
You may not modify the values in the list's nodes. Only nodes themselves may be changed.
看清楚规律，0到n/2都是正序，然后n到n/2是reverse，如何插入

```
class Solution:
    def reorderList(self, head: ListNode) -> None:
        # find middle
        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # reverse second half
        second = slow.next
        prev = slow.next = None
        while second:
            tmp = second.next
            second.next = prev
            prev = second
            second = tmp

        # merge two halfs
        first, second = head, prev
        while second:
            tmp1, tmp2 = first.next, second.next
            first.next = second
            second.next = tmp1
            first, second = tmp1, tmp2
```

我们这边reorder是以第一个为主要，所以第一个多一点好
所以second是 slow.next

1 -> 2 -> 3 <- 4 <- 5像这种，这样反转，第二个部分比第一个部分多一个3
存储后续：保留两部分的下一个节点。
接上第二：将第一部分的下一个节点指向第二部分的当前节点。
续接第一：将第二部分的下一个节点指向第一部分的下一个节点。
推进重排：移动两个指针到下一对待合并的节点。

prev = slow.next = None这一步什么意思和为什么

这行代码 prev = slow.next = None 在重排链表算法中有两个作用：

slow.next = None 将链表从中间位置断开，分为两部分。
slow 变成了第一部分的尾节点，它的 next 指向 None。
prev = None 初始化 prev 变量，为接下来的链表反转做准备，
prev 将用作反转过程中跟踪前一个节点的指针。

在判断链表是否为回文时，不需要断开链表，因为：

对称检查：回文检查的目的是验证链表从前到后是否与从后到前相同，所以只需反转后半部分并与前半部分进行比较。
无需重组：与重排链表不同，回文检查无需对链表进行重新组织或合并操作，因此不需要在中间断开链表。

Maximum Twin Sum Of A Linked List

如果链表的节点数是偶数，
slow 会停在前半部分的最后一个节点。
如果节点数是奇数，slow 会停在中间节点
，即实际上是第一部分的最后一个节点，因为中间节点

```
class Solution:
    def pairSum(self, head: Optional[ListNode]) -> int:
        slow, fast = head, head
        prev = None
        while fast and fast.next:
            fast = fast.next.next
            tmp = slow.next
            slow.next = prev
            prev = slow
            slow = tmp

        res = 0
        while slow:
            res = max(res, prev.val + slow.val)
            prev = prev.next
            slow = slow.next
        return res
```

tmp = slow.next 备下一
slow.next = prev（反指头）— 将 slow 当前的 next 指向 prev
prev = slow —— 将 prev 向前移动到 slow 的位置，
slow = tmp —— 移动 slow 到下一个节点

Remove Nth Node From End of List
设哑点，右保距，左右行，删指顺

重点是想象，two pointer，
一个在
dummy（故意在left前面一个
，这样）
，一个是 right差n个的距离，
两个指针向后滑动，然后删除

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        dummy = ListNode(0, head)
        left = dummy
        right = head

        while n > 0:
            right = right.next
            n -= 1

        while right:
            left = left.next
            right = right.next

        # delete
        left.next = left.next.next
        return dummy.next
```

Swapping Nodes in a Linked List
You are given the head of a linked

Return the head of the linked list af
r swapping the values of the kth no
(the list is 1-indexed).

left是 cur的暂存点
然后正好差k个，

```
class Solution:
    def swapNodes(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        cur = head
        for i in range(k - 1):
            cur = cur.next

        left = cur
        right = head
        while cur.next:
            cur = cur.next
            right = right.next

        left.val, right.val = right.val, left.val

        return head
```

Copy List With Random Pointer	You 为什么要先建立个节点，再next为什么不一开始就全部 GPT ChatGPT Classic 在复制具有随机指针的链表时，直接完全复制每个节点 next 和 random 指针是不可行的， 因为在复制任意节点 N 的 random 指针所指向的节点 R 时， 节点 R 可能还没有被创建和复制， 所以你不能简单地复制 random 指针		""" # 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": oldToCopy = {None: None} cur = head while cur: copy = Node(cur.val) oldToCopy[cur] = copy cur = cur.next cur = head while cur: copy = oldToCopy[cur] copy.next = oldToCopy[cur.next] copy.random = oldToCopy[cur.random] cur = cur.next return oldToCopy[head]														
design a linkedlist	这边有很多是相同的要记得树形结合，要创建两个dummy node给doubly linkedlist		class ListNode: def __init__(self, val): self.val = val self.prev = None self.next = None class MyLinkedList: def __init__(self): self.left = ListNode(0) self.right = ListNode(0) self.left.next = self.right self.right.prev = self.left def get(self, index: int) -> int: cur = self.left.next while cur and index > 0: cur = cur.next index -= 1 if cur and cur != self.right and index == 0: return cur.val return -1	def addAtHead(self, val: int) -> None: node, prev, next = ListNode(val), self.left, self.left.next node.next, node.prev = next, prev next.prev = node prev.next = node def addAtTail(self, val: int) -> None: node, prev, next = ListNode(val), self.right.prev, self.right node.next, node.prev = next, prev next.prev = node prev.next = node def addAtIndex(self, index: int, val: int) -> None: next = self.left.next while next and index > 0: next = next.next index -= 1 if next and index == 0: node, prev = ListNode(val), next.prev node.next, node.prev = next, prev next.prev = node prev.next = node def deleteAtIndex(self, index: int) -> None: node = self.left.next while node and index > 0: node = node.next index -= 1 if node and node != self.right and index == 0: node.prev.next = node.next node.next.prev = node.prev												

				针对上述的各种链表问题，我们可以归纳出以下的解题策略和方法： 反转链表（Reverse Linked List）： 迭代法：通过遍历链表，逐个更改节点的 next 指针，指向前一个节点。 递归法：将问题分解为子问题，递归地反转子链表，然后再将其组合起来。 合并两个排序链表（Merge Two Sorted Lists）： 迭代法：创建一个哑节点作为新链表的起始，然后依次比较两个链表的节点值，并将较小的节点添加到新链表中。 递归法：比较两个链表的头节点，选择较小的一个作为新链表的头，然后递归地合并剩余部分。 检测回文链表（isPalindrome）： 快慢指针法：使用快慢指针找到链表中间节点，然后反转后半部分链表，最后比较前后两部分是否相同。 删除链表元素（Remove Linked List Elements）： 哑节点法：创建一个哑节点指向头节点，然后遍历链表删除指定值的节点。 删除排序链表中的重复元素（Remove Duplicates From Sorted List）： 直接遍历：遍历链表，删除连续重复的节点，只保留一个。 查找两个链表的交点（Intersection of Two Linked Lists）： 双指针法：两个指针分别遍历两个链表，一旦到达尾部就切换到另一链表的头部继续遍历，最终会在交点相遇。 重排链表（Reorder List）： 找中点 + 反转 + 合并：找到链表中点，反转后半部分，然后将两部分节点交替合并。 最大孪生和（Maximum Twin Sum Of A Linked List）： 快慢指针 + 反转 + 比较：快慢指针找中点，反转后半部分，然后对应位置节点求和，记录最大值。 删除链表的倒数第 N 个节点（Remove Nth Node From End of List）： 双指针 + 哑节点：先让一个指针移动 N 步，然后两个指针同时移动直到先行指针到达链表末尾，此时另一指针的位置即是倒数第 N 个节点。 交换链表中的节点（Swapping Nodes in a Linked List）： 找到节点 + 交换值，保证距离：先找到正数第 k 个节点和倒数第 k 个节点，然后交换它们的值。 复制带随机指针的链表（Copy List With Random Pointer）： 哈希表：使用哈希表记录每个原节点及其复制节点的对应关系，再次遍历设置 next 和 random 指针。 原地复制：在每个原节点后面创建新节点，设置新节点的 random 指针，最后拆分原链表和复制链表。 设计链表（Design LinkedList）： 单链表或双向链表实现：根据需求实现链表结构，支持插入、删除、获取等操作。							
双指针技巧： 快慢指针：用于检测链表中的循环，寻找中间节点等。 前后指针：用于找到倒数第 N 个节点，删除特定节点等。 递归： 适用于反转链表、复制链表、比较链表等问题。 递归可以在返回过程中修改节点的链接，常用于反转操作。 哈希表： 用于存储节点之间的关系，尤其是在复制带有随机指针的链表时。 在删除重复节点时，也可以用哈希表记录已经出现过的节点。 哑节点： 在链表的头部引入一个额外的哑节点（dummy node），可以帮助简化边界条件的处理，特别是在头节点可能会变化的情况下。 多次遍历： 有时候，我们可能需要多次遍历链表来解决问题，比如先遍历计算长度，再根据长度信息处理。 合并分治技巧： 对于一些问题，比如排序链表，可以使用合并排序的方法，这通常涉及到分治策略和合并两个排序链表。 原地修改： 在可能的情况下，直接在原链表上进行修改，如原地删除节点，原地分割链表等，以节省空间。 模拟法： 对于一些操作，如链表加法，直接按照数字加法的方式模拟链表节点的处理。														
1472. Design Browser History		You have a browser of one tab where you start on the homepage and you can visit another url , get back in the history number of steps or move forward in the history number of steps. Implement the BrowserHistory class: BrowserHistory(string homepage) Initializes the object with the homepage of the browser. void visit(string url) Visits url from the current page. It clears up all the forward history. string back(int steps) Move steps back in history. If you can only return x steps in the history and steps > x, you will return only x steps. Return the current url after moving back in history at most steps. string forward(int steps) Move steps forward in history. If you can only forward x steps in the history and steps > x, you will forward only x steps. Return the current url after forwarding in history at most steps.		class BrowserHistory: def __init__(self, homepage: str): self.i = 0 // current self.len = 1 self.history = [homepage] # O(1) def visit(self, url: str) -> None: if len(self.history) < self.i + 2: // check if self.i+1 exists self.history.append(url) else: self.history[self.i + 1] = url self.i += 1 self.len = self.i + 1 # O(1) def back(self, steps: int) -> str: self.i = max(self.i - steps, 0) return self.history[self.i] # O(1) def forward(self, steps: int) -> str: self.i = min(self.i + steps, self.len - 1) return self.history[self.i]	在代码中的 if len(self.history) < self.i + 2: 这行是用来检查是否需要扩展历史记录的 列表容量。让我们来解释为什么要使用 self.i + 2： self.i 是当前历史记录的索引，它指向最后一个访 问过的网页。假设当前历史记录有3个网页， self.i 的值为2，因为列表索引是从0开始的。 如果要访问一个新的网页，它将成为历史记录 中的下一个页面，应该位于 self.i + 1 的位置。 这就是为什么我们使用 self.i + 1 来表示下一个位置。 如果在当前历史记录的末尾添加一个新网页， 我们需要确保列表的长度足够大，可以容纳 self.i + 1 的位置。但是， 如果我们只检查 len(self.history) < self.i + 1， 那么在当前历史记录长度为3时， 它看起来已经足够大了，因为 len(self.history) 等于3， 但实际上我们还需要一个额外的位置来存储下一个页面。	View on Github # Linked List Implementation class ListNode: def __init__(self, val, prev=None, next=None): self.val = val self.prev = prev self.next = next class BrowserHistory: def __init__(self, homepage: str): self.cur = ListNode(homepage) # O(1) def visit(self, url: str) -> None: self.cur.next = ListNode(url, self.cur) self.cur = self.cur.next # O(n) def back(self, steps: int) -> str: while self.cur.prev and steps > 0: self.cur = self.cur.prev steps -= 1 return self.cur.val # O(n) def forward(self, steps: int) -> str: while self.cur.next and steps > 0: self.cur = self.cur.next steps -= 1 return self.cur.val						
swap pairs		Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)	# 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(0, head) prev, curr = dummy, head while curr and curr.next: # save ptrs nxtPair = curr.next.next second = curr.next # reverse this pair second.next = curr curr.next = nxtPair prev.next = second # update ptrs prev = curr curr = nxtPair return dummy.next	题目：两个交换后，就是从第三个开始了！！！！ 存备2，reverse就很正常 口诀： 存备二， 反且前后连 prev变 反后curr curr变存前nxt 要表示出 第一个node反转后变成了第二个， 所以新的prev =curr就是走了两步， 然后利用存好的第三个更新为cur 这个update有点意思，因为，curr指的是第一个node，但 这个node已经被reverse 到第二个了，所以其实prev走了两步 cur是指的是nextPair，也就是我们存的 第三个node 也是走了两步									

sort list	Given the head of a linked list, return the list after sorting it in ascending order. 获取中点 递归左右, 合并两个部分, merge sort 两个helper function	# 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 # 获取中间节点 left = head right = self.get_mid(head) tmp = right.next right.next = None right = tmp # 递归排序左右两部分 left = self.sortList(left) right = self.sortList(right) # 合并排序后的两部分 return self.merge(left, right)	def get_mid(self, head: ListNode) -> ListNode: slow, fast = head, head.next while fast and fast.next: slow = slow.next fast = fast.next.next return slow def merge(self, list1: ListNode, list2: ListNode) -> ListNode: new = dummy = ListNode() while list1 and list2: if list1.val > list2.val: new.next = list2 list2 = list2.next else: new.next = list1 list1 = list1.next new = new.next if not list1: new.next = list2 if not list2: new.next = list1 return dummy.next										
partion	Create dummy nodes left and right. Use left_tail and right_tail as tail pointers. Iterate through the original list, moving nodes to either the left or right list. Connect left_tail to right.next to combine the lists. Set right_tail.next to None to end the right list. Return left.next as the modified linked list. 左右分离, 连接返回。	# 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: Optional[ListNode], x: int) -> Optional[ListNode]: #create dummy node left, right = ListNode(), ListNode() #tail iterator left_tail, right_tail = left, right while head: if x > head.val: left_tail.next = head left_tail = left_tail.next else: right_tail.next = head right_tail = right_tail.next head = head.next left_tail.next = right.next right_tail.next = None return left.next												
Rotate List	Given the head of a linked list, rotate the list to the right by k places.	class Solution: def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]: if not head and not head.next: return head #get length length, tail = 1, head while tail.next: tail = tail.next length += 1 #此刻, tail指针已经到了最后一个node k = k% length if k == 0: return head cur = head # 将 cur 移动到 roated位置, 变成新的tail for i in range(length - k -1): cur = cur.next # 新的头就是, 存储 newHead = cur.next # 断开尾巴 cur.next = None tail.next = head return newHead	class Solution: def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]: if not head or not head.next or k == 0: return head old_head = head curr, size = head, 0 //用于计算链表的长度。 while curr: curr, size = curr.next, size + 1 // 很好的方法得到重复, k%size if k % size == 0: return head // 移动到几个位置 k %= size slow = fast = head // slow 和fast保持固定窗口相差 K-1, 当fast走到最后一个, 因为k<=0 while fast and fast.next: if k <= 0: slow = slow.next fast = fast.next k -= 1 //slow 和 fast 指针之间的距离是 k-1 个节点 new_tail, new_head, old_tail = slow, slow.next, fast #三个, 新尾巴, 新头, 旧尾巴。新尾巴要断开, 新头要返回, 旧尾巴连接 旧头 //剪掉尾巴, 链接oldtail指针 new_tail.next, old_tail.next = None, old_head return new_head												

Reverse Linked List II	想象画图很重要 left 2 right 4 1-2-3-4-5 1-4-3-2-5 1是leftprev 2是leftprev.next 最后leftprev.next链接的是5 leftprev链接的是4	# 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: Optional[ListNode], left: int, right: int) -> Optional[ListNode]: dummy = ListNode(0,head) #reach the node of left # move left-1 step to get the left leftprev,cur = dummy,head for i in range(left-1): leftprev = cur cur = cur.next # cur = left, prevleft = cur-1 prev = None for i in range(right-left+1): #reverse tmpNext = cur.next cur.next = prev #move prev, cur = cur, tmpNext #left is point to cur(cur is the node after right) leftprev.next.next = cur leftprev.next = prev return dummy.next																	

Design Circular Queue	# linkelist方法
	class Node:
	  def __init__(self, val: int):
	  self.val = val
	  self.next = None
	
	class MyCircularQueue:
	
	  def __init__(self, k: int):
	  self.head = self.tail = None
	  self.capacity = k
	  self.size = 0
	
	  def enQueue(self, value: int) -> bool:
	  if self.isFull():
	  return False
	
	  node = Node(value)
	  if self.size == 0:
	  self.head = self.tail = node
	  else:
	  self.tail.next = self.tail = node
	
	  self.size += 1
	
	  return True
		def deQueue(self) -> bool:
		  if self.isEmpty():
		  return False
		
		  self.head = self.head.next
		  self.size -= 1
		
		  return True
		
		def Front(self) -> int:
		  return -1 if self.isEmpty() else self.head.val
		
		def Rear(self) -> int:
		  return -1 if self.isEmpty() else self.tail.val
		
		def isEmpty(self) -> bool:
		  return self.size == 0
		
		def isFull(self) -> bool:
		  return self.capacity == self.size
			class MyCircularQueue:
			
			  def __init__(self, k: int):
			  self.queue = []
			  self.k = k
			
			  def enQueue(self, value: int) -> bool:
			  if self.isFull():
			  return False
			  if len(self.queue) < self.k:
			  self.queue.append(value)
			  return True
			
			  def deQueue(self) -> bool:
			  if self.isEmpty():
			  return False
			  if len(self.queue) > 0:
			  self.queue.pop(0)
			  return True
			
			  def Front(self) -> int:
			  if len(self.queue) > 0:
			  return self.queue[0]
			  return -1
			
			  def Rear(self) -> int:
			  if len(self.queue) > 0:
			  return self.queue[-1]
			  return -1
			
			  def isEmpty(self) -> bool:
			  if len(self.queue) == 0:
			  return True
			
			  def isFull(self) -> bool:
			  if len(self.queue) == self.k:
			  return True
				使用链表（LinkedList）实现循环队列（Circular Queue）有其独特的优点和缺点：
				
				优点
				动态大小：链表不需要在创建时就确定大小，它可以根据需要动态地增加或减少节点。这在不确定队列长度的情况下非常有用。
				
				内存利用率高：由于链表元素只在需要时被创建，它不会像固定大小的数组那样浪费空间。
				
				插入和删除效率高：在链表中添加或删除元素（例如，在循环队列的头部或尾部）通常是时间复杂度为
				O(1) 的操作，因为不需要移动其他元素。
				
				无内存重分配：与数组不同，链表在增加元素时不需要重分配整个内存空间。
				
				缺点
				内存占用：每个链表节点除了存储数据外，还需要额外的空间来存储指向下一个节点的指针，这增加了内存的消耗。
				
				缓存不友好：由于链表的元素在内存中是分散存储的，它不利于CPU缓存，可能导致较慢的访问速度。
				
				访问时间：相对于数组，链表随机访问一个元素的时间复杂度是
				O(n)，因为需要从头开始遍历链表。
				
				复杂度：实现链表比实现数组复杂，特别是在处理指针和正确地管理内存时。
				
				在选择使用链表还是数组来实现循环队列时，应根据具体的应用场景和性能要求来决定。如果需要频繁地在队列中间插入或删除元素，链表可能是更好的选择。但如果关注内存使用和随机访问效率，数组可能更合适。

insertionSortList			先后再换顺序不乱	class Solution: def insertionSortList(self, head: Optional[ListNode]) -> Optional[ListNode]: if not head or not head.next: return head sentinel = ListNode() curr = head while curr: prev = sentinel while prev.next and curr.val >= prev.next.val: prev = prev.next curr.next, prev.next, curr = prev.next, curr, curr.next #先后再换，往前不乱 return sentinel.next														
一下子要 两个或者三个的 连写 变量，其实更有助于理解 与反应！！！！！这种写法更帮助你！就和declare filed什么的一样																			

LRU cauche		class Node: def __init__(self, key, val): self.key, self.val = key, val self.prev = self.next = None class LRUCache: def __init__(self, capacity: int): self.cap = capacity self.cache = {} # map key to node self.left, self.right = Node(0, 0), Node(0, 0) self.left.next, self.right.prev = self.right, self.left # remove node from list def remove(self, node): prev, nxt = node.prev, node.next prev.next, nxt.prev = nxt, prev # insert node at right记住 def insert(self, node): # 获取当前右端节点的前一个节点和当前右端节点 prev, nxt = self.right.prev, self.right # 更新前一个节点的下一个节点为新插入的节点，以 prev.next = nxt.prev = node # 更新新节点的下一个节点和前一个节点，将其插入 node.next, node.prev = nxt, prev																		
					def get(self, key: int) -> int: if key in self.cache: # 如果键在缓存中存在，将对应的节点从链表中移除并重新插入到链表的右端 self.remove(self.cache[key]) self.insert(self.cache[key]) # 返回键对应的值 return self.cache[key].val # 如果键不存在于缓存中，返回-1 return -1 def put(self, key: int, value: int) -> None: if key in self.cache: # 如果键已经存在于缓存中，先将对应的节点从链表中移除 self.remove(self.cache[key]) # 创建一个新的节点，并将其添加到缓存中 self.cache[key] = Node(key, value) # 将新节点插入到链表的右端表示它是最近使用的 self.insert(self.cache[key]) if len(self.cache) > self.cap: # 如果缓存中的键值对数量超过了容量限制，需要删除最近最少使用的节点 lru = self.left.next # 从链表中移除LRU节点 self.remove(lru) # 从缓存字典中删除LRU节点对应的键值对 del self.cache[lru.key]																
lowestCommonAncestor					class Solution: def lowestCommonAncestor(self, root: "TreeNode", p: "TreeNode", q: "TreeNode") -> "TreeNode": while True: if root.val < p.val and root.val < q.val: root = root.right elif root.val > p.val and root.val > q.val: root = root.left else: return root																