## 反转链表

<https://leetcode.cn/problems/reverse-linked-list/>

思路：

# 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: Optional[ListNode]) -> Optional[ListNode]:

if not head:

return

pre=None

cur=head

while cur:

nxt=cur.next

cur.next=pre

pre=cur

cur=nxt

return pre

## 哨兵节点

翻转k个（from deepseek）

给你链表的头节点 head ，每 k 个节点一组进行翻转，请你返回修改后的链表。

k 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

示例 1：

输入：head = [1,2,3,4,5], k = 2

输出：[2,1,4,3,5]

示例 2：

输入：head = [1,2,3,4,5], k = 3

输出：[3,2,1,4,5]

class Solution:

def reverseKGroup(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:

dummy = ListNode(next=head)

prev\_group\_end = dummy # 上一组的**尾节点**

while True:

# 1. 检查剩余节点是否足够k个

curr = prev\_group\_end.next

for \_ in range(k):

if not curr:

return dummy.next

curr = curr.next

# 2. 反转当前k个节点

curr\_head = prev\_group\_end.next # 当前组的原始头（反转后变为尾）

prev = None

curr = curr\_head

for \_ in range(k):

next\_node = curr.next

curr.next = prev

prev = curr

curr = next\_node

# 3. 连接上一组尾到当前组新头，并更新尾节点

prev\_group\_end.next = prev # prev是当前组新头,这一步同样包括dummy.next的调整（从后向前连的方法）

curr\_head.next = curr # 当前组尾（原头）连接下一组

prev\_group\_end = curr\_head # 更新尾节点

两两交换：

class Solution:

def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:

dummy = ListNode(next=head) # 创建虚拟头节点

cur = dummy

while cur.next and cur.next.next:

# 记录当前两个节点

node1 = cur.next

node2 = cur.next.next

# 交换节点

cur.next = node2 # 步骤1: 连接前驱到第二个节点

node1.next = node2.next # 步骤2: 第一个节点指向下一组的头（从前向后连的方法）

node2.next = node1 # 步骤3: 第二个节点指向第一个节点

# 移动指针到下一组的前驱位置

cur = node1

return dummy.next # 返回新头节点

用node1,node2暂存后修改node1，node2的地址要优于用cur.next（变化后的）.next修改地址

## 递归写链表：

class Solution:

def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:

if list1 is None: return list2 # 注：如果都为空则返回空

if list2 is None: return list1

if list1.val < list2.val:

list1.next = self.mergeTwoLists(list1.next, list2)

return list1

list2.next = self.mergeTwoLists(list1, list2.next)

return list2

## 链表的快速删除和存入：

class ListNode:

def \_\_init\_\_(self, key=None, value=None):

self.key = key

self.value = value

self.prev = None

self.next = None

class LRUCache:

def \_\_init\_\_(self, capacity: int):

self.capacity = capacity

self.hashmap = {}

# 新建两个节点 head 和 tail

self.head = ListNode()

self.tail = ListNode()

# 初始化链表为 head <-> tail

self.head.next = self.tail

self.tail.prev = self.head

# 因为get与put操作都可能需要将双向链表中的某个节点移到末尾，所以定义一个方法

def move\_node\_to\_tail(self, key):

# 先将哈希表key指向的节点拎出来，为了简洁起名node

# hashmap[key] hashmap[key]

# | |

# V --> V

# prev <-> node <-> next pre <-> next ... node

node = self.hashmap[key]

node.prev.next = node.next

node.next.prev = node.prev

# 之后将node插入到尾节点前

# hashmap[key] hashmap[key]

# | |

# V --> V

# prev <-> tail ... node prev <-> node <-> tail

node.prev = self.tail.prev

node.next = self.tail

self.tail.prev.next = node

self.tail.prev = node

def get(self, key: int) -> int:

if key in self.hashmap:

# 如果已经在链表中了久把它移到末尾（变成最新访问的）

self.move\_node\_to\_tail(key)

res = self.hashmap.get(key, -1)

if res == -1:

return res

else:

return res.value

def put(self, key: int, value: int) -> None:

if key in self.hashmap:

# 如果key本身已经在哈希表中了就不需要在链表中加入新的节点

# 但是需要更新字典该值对应节点的value

self.hashmap[key].value = value

# 之后将该节点移到末尾

self.move\_node\_to\_tail(key)

else:

if len(self.hashmap) == self.capacity:

# 去掉哈希表对应项

self.hashmap.pop(self.head.next.key)

# 去掉最久没有被访问过的节点，即头节点之后的节点

self.head.next = self.head.next.next

self.head.next.prev = self.head

# 如果不在的话就插入到尾节点前

new = ListNode(key, value)#构建节点

self.hashmap[key] = new

**new.prev = self.tail.prev**

**new.next = self.tail**

**self.tail.prev.next = new**

**self.tail.prev = new**#双向链表的交换位置

## 交换位置

# 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: Optional[ListNode]) -> Optional[ListNode]:

a,b=head,head

pre=None

if head.next:

b=b.next

while b:

if pre is not None :

pre.next=b

c=b.next

b.next=a

pre=a

a=c

if a:

b=c.next

else:

b=None

return head

双向链表的懒删除用法：

[3510. 移除最小数对使数组有序 II - 力扣（LeetCode）](https://leetcode.cn/problems/minimum-pair-removal-to-sort-array-ii/description/)

class Solution:

def minimumPairRemoval(self, nums: List[int]) -> int:

n = len(nums)

h = [] # (相邻元素和，左边那个数的下标)

dec = 0 # 递减的相邻对的个数

for i, (x, y) in enumerate(pairwise(nums)):

if x > y:

dec += 1

h.append((x + y, i))

heapify(h)

lazy = defaultdict(int)

# 每个下标的左右最近的未删除下标

left = list(range(-1, n)) # 加一个哨兵，防止下标越界

right = list(range(1, n + 1))

ans = 0

while dec:

ans += 1

while lazy[h[0]]:

lazy[heappop(h)] -= 1

s, i = heappop(h) # 删除相邻元素和最小的一对

# (当前元素，下一个数)

nxt = right[i]

if nums[i] > nums[nxt]: # 旧数据

dec -= 1

# (前一个数，当前元素)

pre = left[i]

if pre >= 0:

if nums[pre] > nums[i]: # 旧数据

dec -= 1

if nums[pre] > s: # 新数据

dec += 1

lazy[(nums[pre] + nums[i], pre)] += 1 # 懒删除

heappush(h, (nums[pre] + s, pre))

# (下一个数，下下一个数)

nxt2 = right[nxt]

if nxt2 < n:

if nums[nxt] > nums[nxt2]: # 旧数据

dec -= 1

if s > nums[nxt2]: # 新数据（当前元素，下下一个数）

dec += 1

lazy[(nums[nxt] + nums[nxt2], nxt)] += 1 # 懒删除

heappush(h, (s + nums[nxt2], i))

nums[i] = s

# 删除 nxt

l, r = left[nxt], right[nxt]

right[l] = r # 模拟双向链表的删除操作

left[r] = l

#也可以直接连接i和nxt2：

#right[i]=nxt2

#left[nxt2]=i#这一步也一定要记得

#注意这时候涉及到的超临界问题(上面扩展一个边界的原因)

return ans

class Solution:

def minimumPairRemoval(self, nums: List[int]) -> int:

sl = SortedList() # (相邻元素和，左边那个数的下标)

idx = SortedList(range(len(nums))) # 剩余下标

dec = 0 # 递减的相邻对的个数

for i, (x, y) in enumerate(pairwise(nums)):

if x > y:

dec += 1

sl.add((x + y, i))

ans = 0

while dec > 0:

ans += 1

s, i = sl.pop(0) # 删除相邻元素和最小的一对

k = idx.bisect\_left(i)

# (当前元素，下一个数)

nxt = idx[k + 1]

if nums[i] > nums[nxt]: # 旧数据

dec -= 1

# (前一个数，当前元素)

if k > 0:

pre = idx[k - 1]

if nums[pre] > nums[i]: # 旧数据

dec -= 1

if nums[pre] > s: # 新数据

dec += 1

sl.remove((nums[pre] + nums[i], pre))

sl.add((nums[pre] + s, pre))

# (下一个数，下下一个数)

if k + 2 < len(idx):

nxt2 = idx[k + 2]

if nums[nxt] > nums[nxt2]: # 旧数据

dec -= 1

if s > nums[nxt2]: # 新数据（当前元素，下下一个数）

dec += 1

sl.remove((nums[nxt] + nums[nxt2], nxt))

sl.add((s + nums[nxt2], i))

nums[i] = s # 把 nums[nxt] 加到 nums[i] 中

idx.remove(nxt) # 删除 nxt

return ans