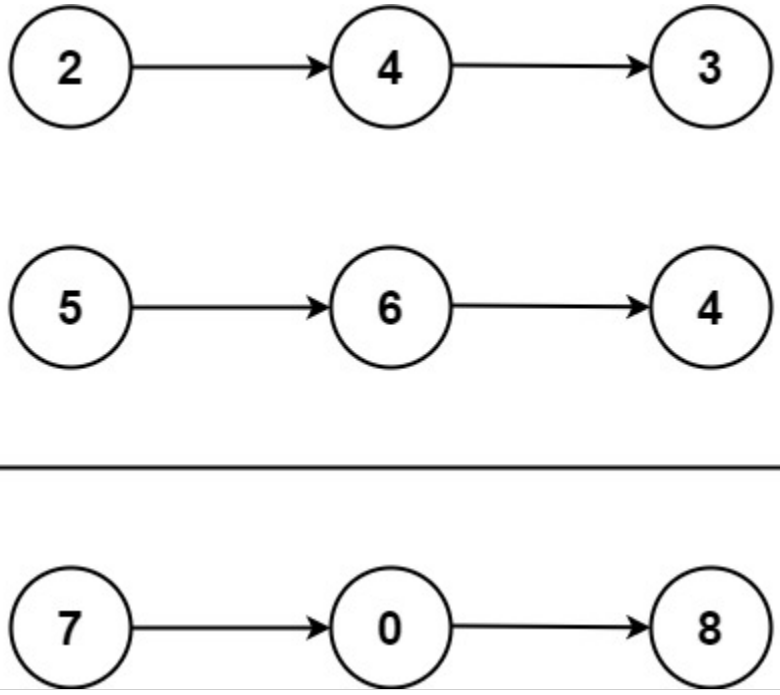
2. Add Two Numbers

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:




```
Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
Explanation: 342 + 465 = 807.
```

Example 2:

```
Input: l1 = [0], l2 = [0]
Output: [0]
```

Example 3:

```
Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
Output: [8,9,9,9,0,0,0,1]
```

Constraints:

- The number of nodes in each linked list is in the range `[1, 100]`.
- `0 <= Node.val <= 9`
- It is guaranteed that the list represents a number that does not have leading zeros.

---

class Solution: def addTwoNumbers(self, l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:

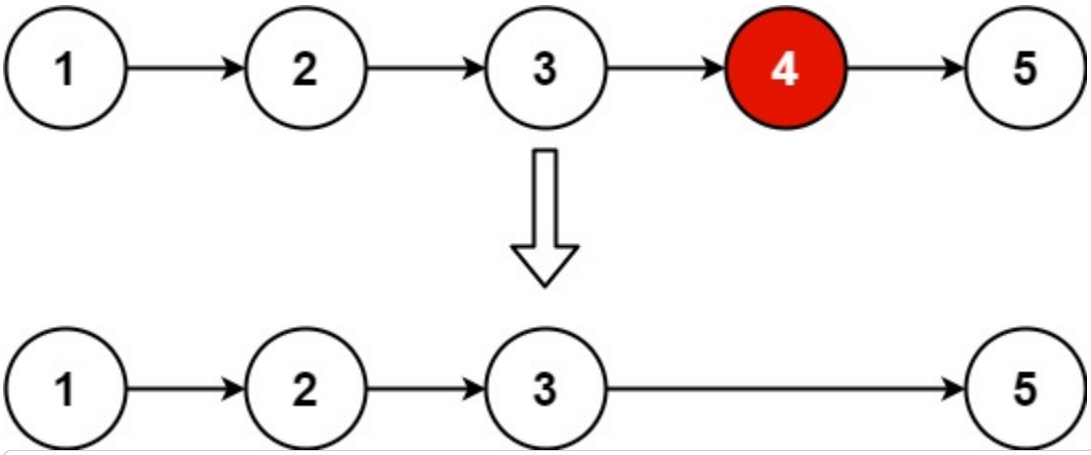
```
    h1 = l1
    h2 = l2
    carry = 0
    prev = None
    while h1 and h2:
        num =  h1.val+h2.val +carry
        if num < 10:
            h1.val = num
            carry = 0
        else:
            num = num%10
            h1.val = num
            carry = 1
        prev = h1
        h1 = h1.next
        h2 = h2.next
    if h2:
        prev.next = h2
        h1 = h2
    while h1:
        num =  h1.val+carry
        if num < 10:
            h1.val = num
            carry = 0
        else:
            num = num%10
            h1.val = num
            carry = 1
        prev = h1
        h1 = h1.next
    if carry:
        prev.next = Node(1)
    return l1
```

---

19. Remove Nth Node From End of List

Given the `head` of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:




```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

Example 2:

```
Input: head = [1], n = 1
Output: []
```

Example 3:

```
Input: head = [1,2], n = 1
Output: [1]
```

Constraints:

- The number of nodes in the list is `sz` .
- `1 <= sz <= 30`
- `0 <= Node.val <= 100`
- `1 <= n <= sz`

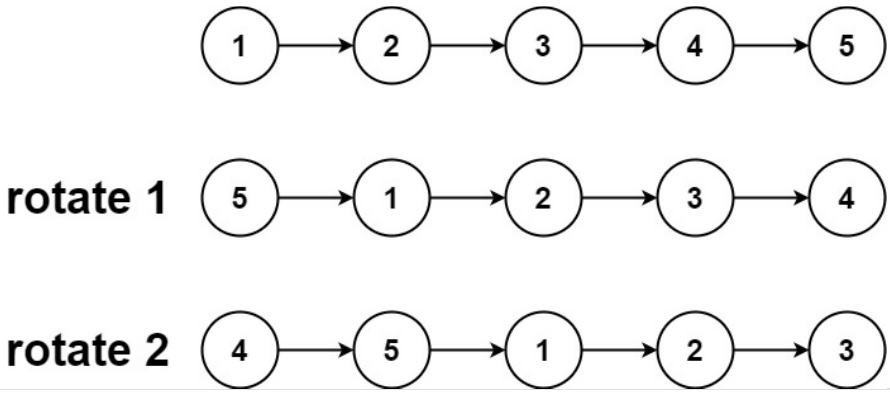
Follow up: Could you do this in one pass?

```
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNo
de]:
        dummy = ListNode(0, head)
        fast = slow = dummy

        for _ in range(n):
            fast = fast.next

        while fast.next:
            fast = fast.next
            slow = slow.next

        slow.next = slow.next.next
        return dummy.next
```

61. Rotate List

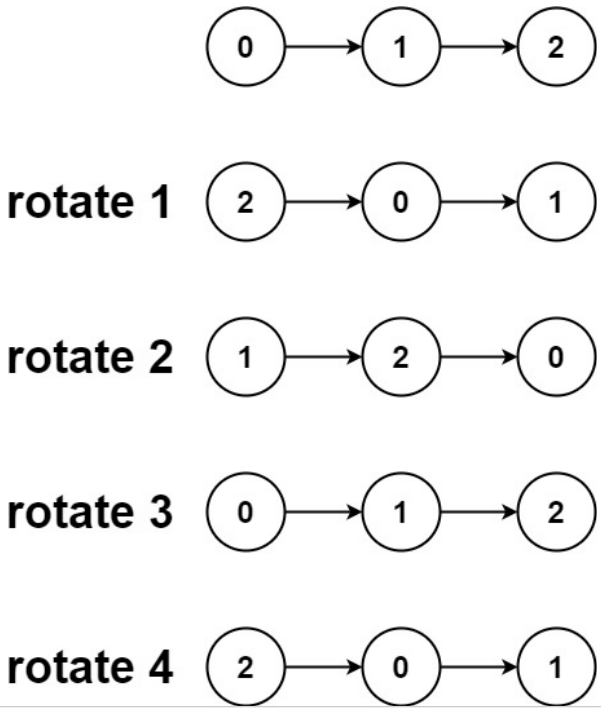
Given the `head` of a linked list, rotate the list to the right by `k` places.

Example 1:




```
Input: head = [1,2,3,4,5], k = 2
Output: [4,5,1,2,3]
```

Example 2:




```
Input: head = [0,1,2], k = 4
Output: [2,0,1]
```

Constraints:

- The number of nodes in the list is in the range `[0, 500]`.
- `-100 <= Node.val <= 100`
- `0 <= k <= 2 * 10^9`

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

        #Compute The length
        curr = head
        length = 1
        while curr.next:
            curr = curr.next
            length+=1

        curr.next = head #Make the list Circular

        #Find the Tail
        k = length - (k%length)
        while k:
            curr = curr.next
            k-=1

        #Break the circle and return the New Head
        newHead = curr.next
        curr.next = None
        return newHead
```

138. Copy List with Random Pointer

A linked list of length `n` is given such that each node contains an additional random pointer, which could point to any node in the list, or `null`.

Construct a **deep copy** (https://en.wikipedia.org/wiki/Object_copying#Deep_copy) of the list. The deep copy should consist of exactly `n` **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the `next` and `random` pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes `X` and `Y` in the original list, where `X.random --> Y`, then for the corresponding two nodes `x` and `y` in the copied list, `x.random --> y`.

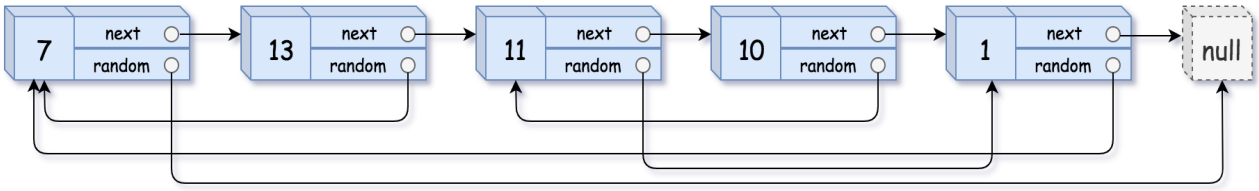
Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of `n` nodes. Each node is represented as a pair of `[val, random_index]` where:

- `val` : an integer representing `Node.val`
- `random_index` : the index of the node (range from `0` to `n-1`) that the `random` pointer points to, or `null` if it does not point to any node.

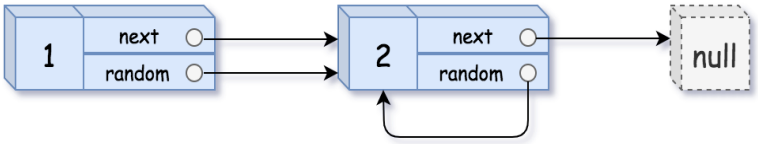
Your code will **only** be given the `head` of the original linked list.

Example 1:



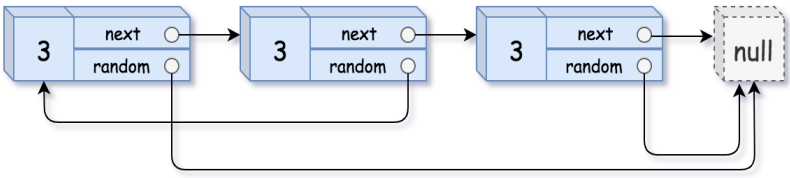

```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

Example 2:




```
Input: head = [[1,1],[2,1]]
Output: [[1,1],[2,1]]
```

Example 3:




```
Input: head = [[3,null],[3,0],[3,null]]
Output: [[3,null],[3,0],[3,null]]
```

Constraints:

- `0 <= n <= 1000`
- `-10^4 <= Node.val <= 10^4`
- `Node.random` is `null` or is pointing to some node in the linked list.

```
class Solution:
    def copyRandomList(self, head: 'Optional[Node]') -> 'Optional[Node]':
        cur=head
        i=0
        dummy=Node(100)
        head2=dummy
        mapp=[]
        original=[]
        while cur:
            original.append(cur.val)
            cur.val=i
            node=Node(i)
            mapp.append(node)
            dummy.next=node
            dummy=dummy.next
            cur=cur.next
            i+=1

        cur=head2.next
        i=0
        while head:
            if head.random is not None:
                cur.random=mapp[head.random.val]
            else:
                cur.random=None
            head=head.next
            cur.val=original[i]
            cur=cur.next
            i+=1

        return head2.next
```

141. Linked List Cycle

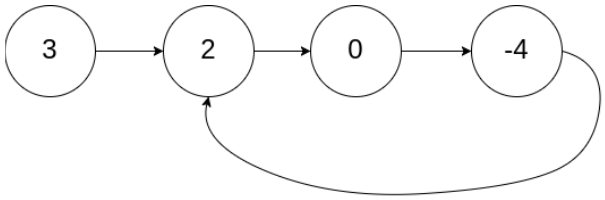
Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node

that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter**.

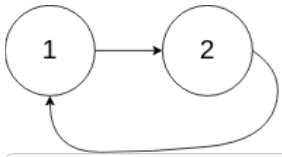
Return `true` *if there is a cycle in the linked list*. Otherwise, return `false`.

Example 1:




```
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st
```

Example 2:




```
Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 0th
```

Example 3:


```
Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.
```

Constraints:

- The number of the nodes in the list is in the range `[0, 10^4]`.
- `-10^5 <= Node.val <= 10^5`
- `pos` is `-1` or a **valid index** in the linked-list.

Follow up: Can you solve it using `O(1)` (i.e. constant) memory?

```
class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        slow = head
        fast = head
        while fast != None and fast.next!= None:
            fast = fast.next.next
            slow = slow.next
            if fast == slow:
                return True

        return False
```

---

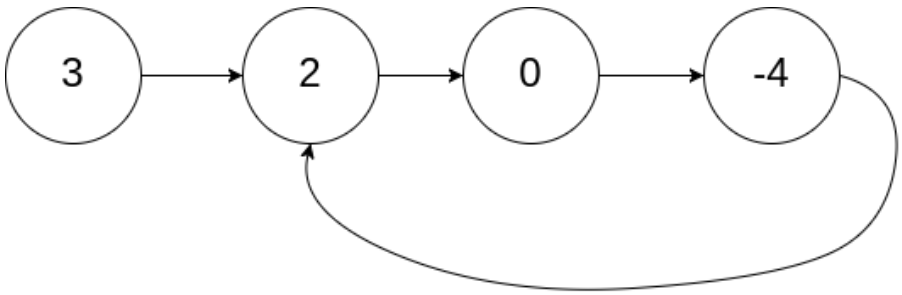
142. Linked List Cycle II

Given the `head` of a linked list, return *the node where the cycle begins. If there is no cycle, return* `null`.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to (**0-indexed**). It is `-1` if there is no cycle. **Note that** `pos` **is not passed as a parameter**.

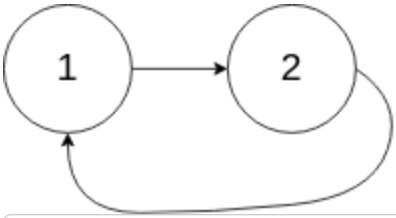
Do not modify the linked list.

Example 1:




```
Input: head = [3,2,0,-4], pos = 1
Output: tail connects to node index 1
Explanation: There is a cycle in the linked list, where tail connects to the second r
```

Example 2:




```
Input: head = [1,2], pos = 0
Output: tail connects to node index 0
Explanation: There is a cycle in the linked list, where tail connects to the first no
```

Example 3:


```
Input: head = [1], pos = -1
Output: no cycle
Explanation: There is no cycle in the linked list.
```

Constraints:

- The number of the nodes in the list is in the range `[0, 10^4]` .
- `-10^5 <= Node.val <= 10^5`
- `pos` is `-1` or a **valid index** in the linked-list.

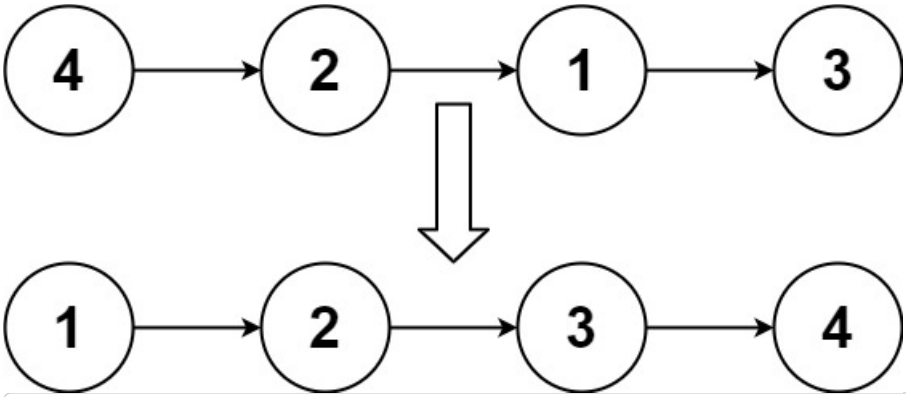
Follow up: Can you solve it using `O(1)` (i.e. constant) memory?

```
class Solution:
    def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        slow = head
        fast = head
        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next

            if slow == fast:
                slow = head
                while fast != slow:
                    fast = fast.next
                    slow = slow.next
                return slow
        return None
```

148. Sort List

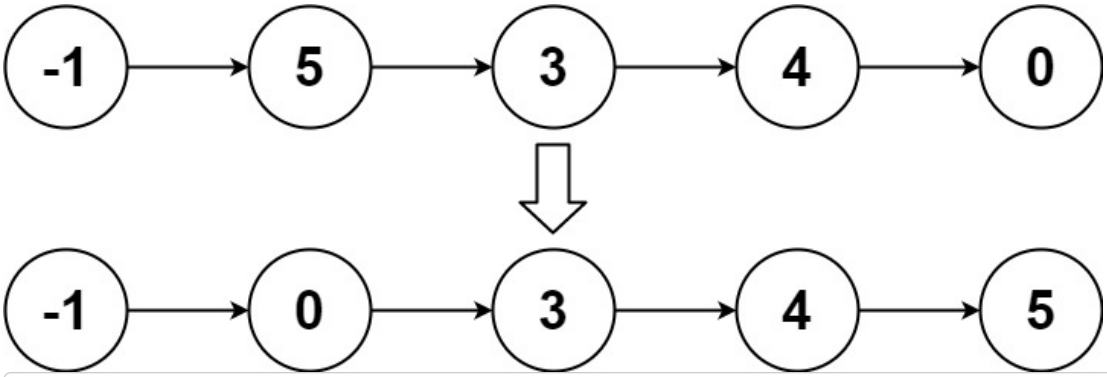
Given the `head` of a linked list, return *the list after sorting it in **ascending order***.

Example 1:




```
Input: head = [4,2,1,3]
Output: [1,2,3,4]
```

Example 2:




```
Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]
```

Example 3:

```
Input: head = []
Output: []
```

Constraints:

- The number of nodes in the list is in the range `[0, 5 * 10^4]`.
- `-10^5 <= Node.val <= 10^5`

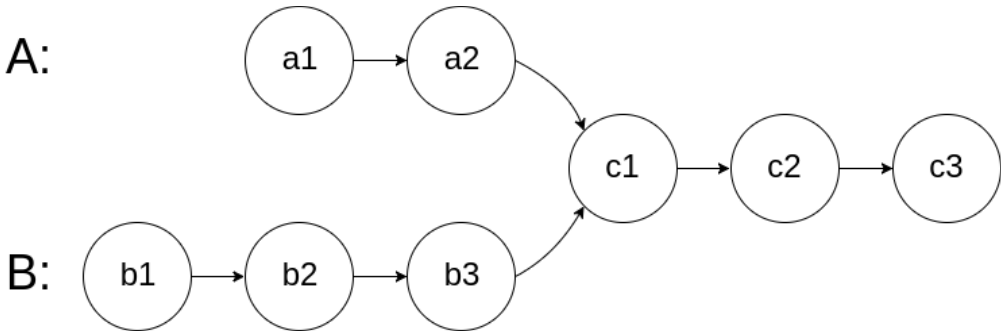
Follow up: Can you sort the linked list in `O(n logn)` time and `O(1)` memory (i.e. constant space)?

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

    def merge(self,l1,l2):
        dummy = ListNode(-1)
        cur = dummy
        while l1 and l2:
            if l1.val < l2.val:
                cur.next = l1
                l1= l1.next
            else:
                cur.next = l2
                l2 = l2.next
            cur = cur.next

        cur.next = l1 if l1 else l2
        return dummy.next

    def merge_sort(self,node):
        if not node or not node.next:
            return node

        mid_head = self.divide_LL(node)
        left = self.merge_sort(node)
        right = self.merge_sort(mid_head)
        return self.merge(left,right)
    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.merge_sort(head)
```

160. Intersection of Two Linked Lists

Given the heads of two singly linked-lists `headA` and `headB`, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return `null`.

For example, the following two linked lists begin to intersect at node `c1`:



The test cases are generated such that there are no cycles anywhere in the entire linked structure.

Note that the linked lists must **retain their original structure** after the function returns.

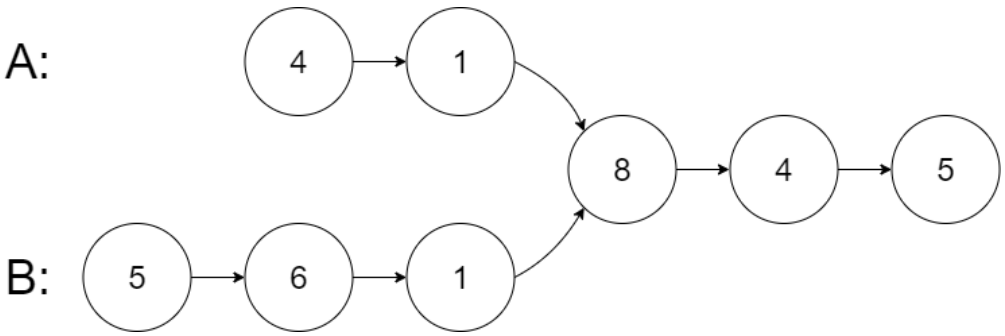
Custom Judge:

The inputs to the **judge** are given as follows (your program is **not** given these inputs):

- `intersectVal` - The value of the node where the intersection occurs. This is `0` if there is no intersected node.
- `listA` - The first linked list.
- `listB` - The second linked list.
- `skipA` - The number of nodes to skip ahead in `listA` (starting from the head) to get to the intersected node.
- `skipB` - The number of nodes to skip ahead in `listB` (starting from the head) to get to the intersected node.

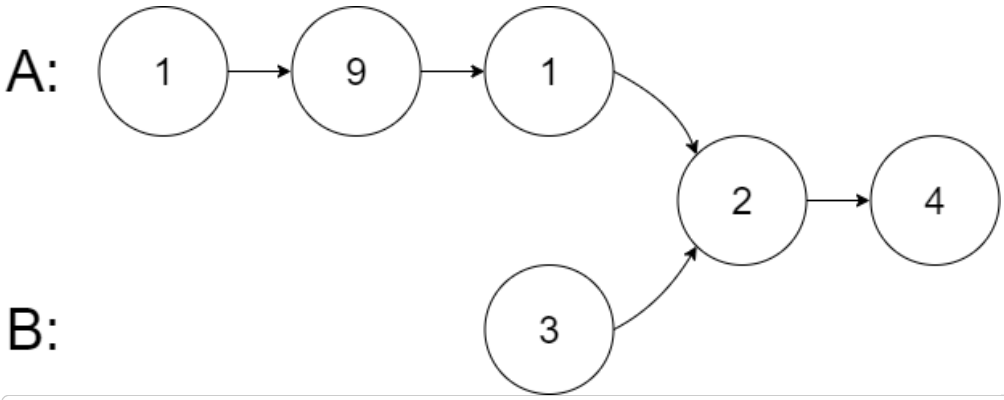
The judge will then create the linked structure based on these inputs and pass the two heads, `headA` and `headB` to your program. If you correctly return the intersected node, then your solution will be **accepted**.

Example 1:



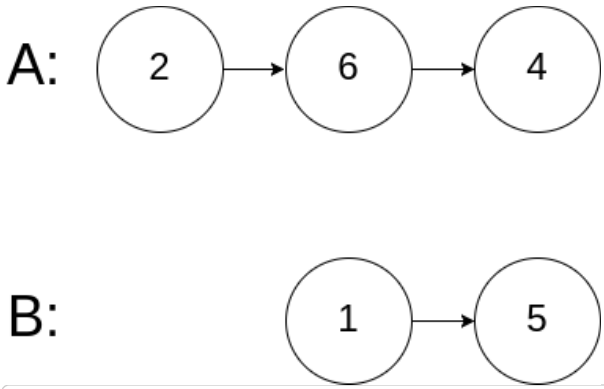
```
Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipE
Output: Intersected at '8'
Explanation: The intersected node's value is 8 (note that this must not be 0 if the t
From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8
- Note that the intersected node's value is not 1 because the nodes with value 1 in A
```

Example 2:




```
Input: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
Output: Intersected at '2'
Explanation: The intersected node's value is 2 (note that this must not be 0 if the t
From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4].
```

Example 3:




```
Input: intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
Output: No intersection
Explanation: From the head of A, it reads as [2,6,4]. From the head of B, it reads as
Explanation: The two lists do not intersect, so return null.
```

Constraints:

- The number of nodes of `listA` is in the `m`.
- The number of nodes of `listB` is in the `n`.
- `1 <= m, n <= 3 * 10^4`

- `1 <= Node.val <= 10^5`
- `0 <= skipA <= m`
- `0 <= skipB <= n`
- `intersectVal` is `0` if `listA` and `listB` do not intersect.
- `intersectVal == listA[skipA] == listB[skipB]` if `listA` and `listB` intersect.

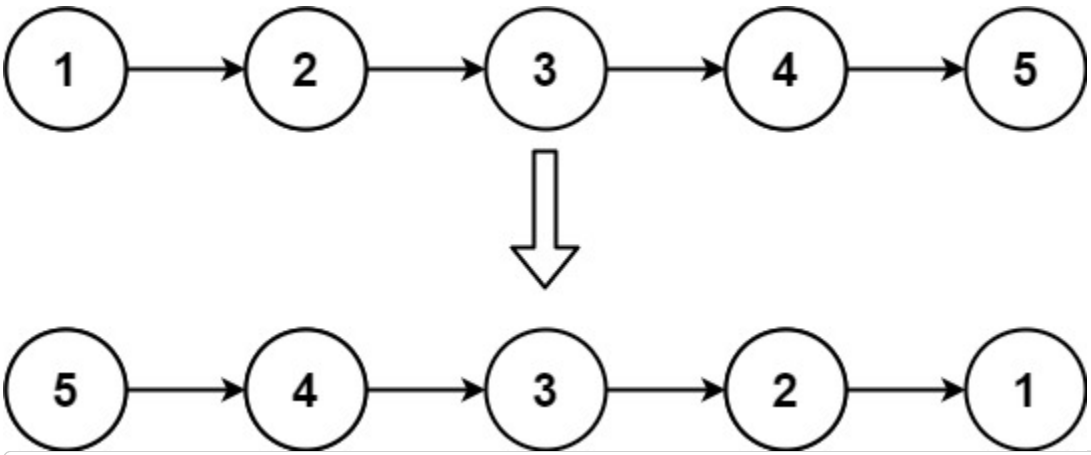
Follow up: Could you write a solution that runs in `O(m + n)` time and use only `O(1)` memory?

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

206. Reverse Linked List

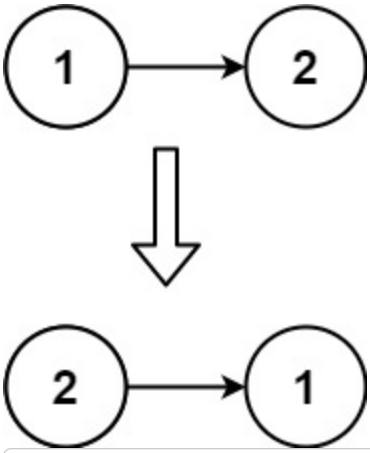
Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:




```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

Example 2:




```
Input: head = [1,2]
Output: [2,1]
```

Example 3:

```
Input: head = []
Output: []
```

Constraints:

- The number of nodes in the list is the range `[0, 5000]`.
- `-5000 <= Node.val <= 5000`

Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?

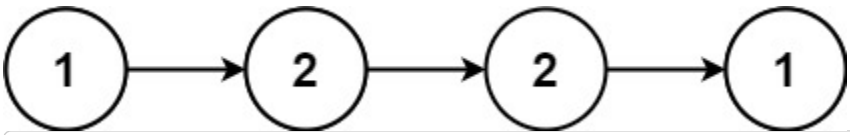
```
class Solution:
    def reverseList(self, root: Optional[ListNode]) -> Optional[ListNode]:

        def reverse(head):
            if not head or not head.next:
                return head

            new_head = reverse(head.next)
            front = head.next
            front.next = head
            head.next = None
            return new_head

        return reverse(root)
```

---

234. Palindrome Linked List

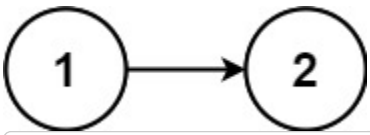
Given the `head` of a singly linked list, return `true` *if it is a palindrome or* `false` *otherwise.*

Example 1:




```
Input: head = [1,2,2,1]
Output: true
```

Example 2:




```
Input: head = [1,2]
Output: false
```

Constraints:

- The number of nodes in the list is in the range `[1, 10^5]` .
- `0 <= Node.val <= 9`

Follow up: Could you do it in `O(n)` time and `O(1)` space?

```
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        cur = head
        prev = None
        data = ""
        while cur:
            data += str(cur.val)
            temp = cur
            cur = cur.next
            temp.next = prev
            prev = temp
        return data[::-1] == data
```

237. Delete Node in a Linked List

There is a singly-linked list `head` and we want to delete a node `node` in it.

You are given the node to be deleted `node`. You will **not be given access** to the first node of `head`.

All the values of the linked list are **unique**, and it is guaranteed that the given node `node` is not the last node in the linked list.

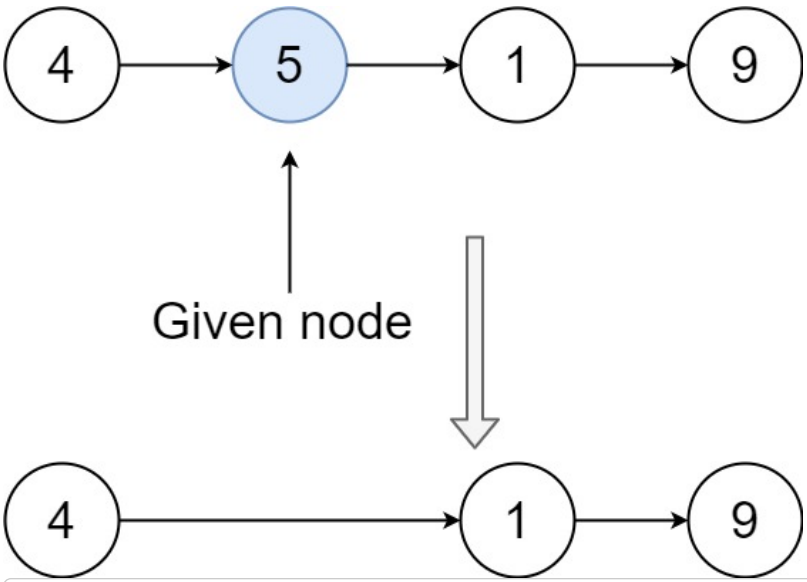
Delete the given node. Note that by deleting the node, we do not mean removing it from memory. We mean:

- The value of the given node should not exist in the linked list.
- The number of nodes in the linked list should decrease by one.
- All the values before `node` should be in the same order.
- All the values after `node` should be in the same order.

Custom testing:

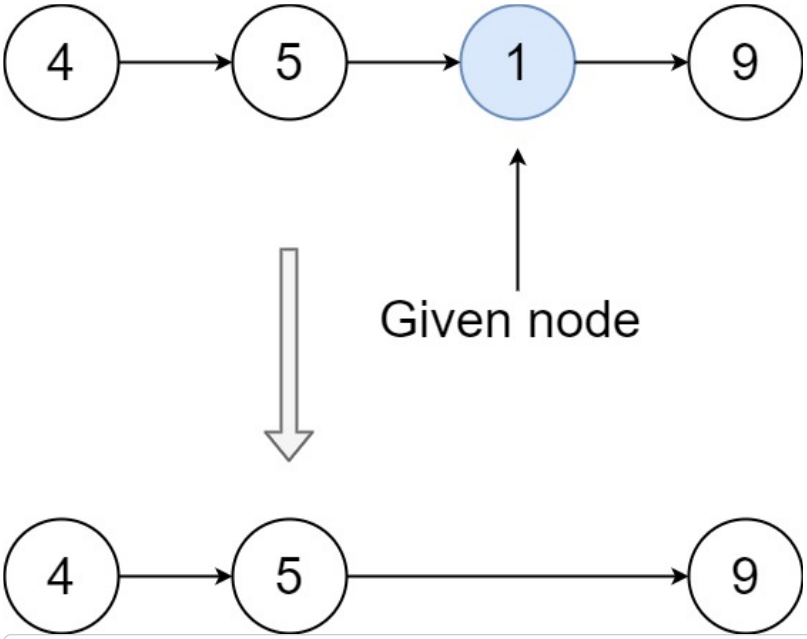
- For the input, you should provide the entire linked list `head` and the node to be given `node`. `node` should not be the last node of the list and should be an actual node in the list.
- We will build the linked list and pass the node to your function.
- The output will be the entire list after calling your function.

Example 1:




```
Input: head = [4,5,1,9], node = 5
Output: [4,1,9]
Explanation: You are given the second node with value 5, the linked list should becor
```

Example 2:




```
Input: head = [4,5,1,9], node = 1
Output: [4,5,9]
Explanation: You are given the third node with value 1, the linked list should become
```

Constraints:

- The number of the nodes in the given list is in the range `[2, 1000]`.
- `-1000 <= Node.val <= 1000`
- The value of each node in the list is **unique**.
- The `node` to be deleted is **in the list** and is **not a tail** node.

```
class Solution:
    def deleteNode(self, node):
        node.val = node.next.val
        if node.next.next != None:
            node.next = node.next.next
        else:
            node.next = None
```

---

328. Odd Even Linked List

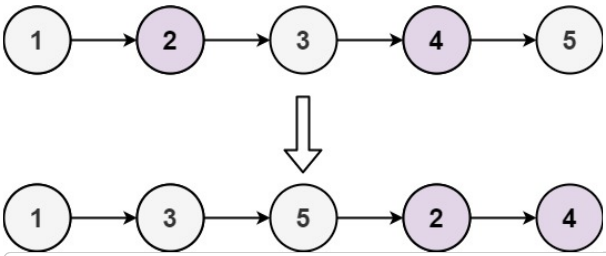
Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

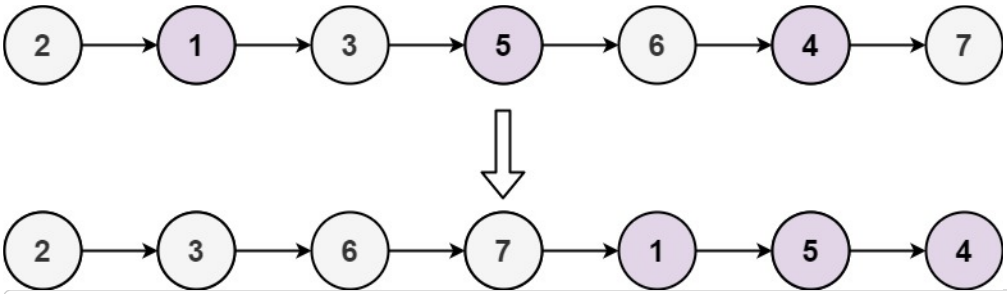
You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

Example 1:




```
Input: head = [1,2,3,4,5]
Output: [1,3,5,2,4]
```

Example 2:




```
Input: head = [2,1,3,5,6,4,7]
Output: [2,3,6,7,1,5,4]
```

Constraints:

- The number of nodes in the linked list is in the range `[0, 10^4]`.
- `-10^6 <= Node.val <= 10^6`

```python
class Solution:
    def oddEvenList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head or not head.next:
            return head

        odd = head
        even = head.next
        even_head = even

        while even and even.next:
            odd.next = even.next
            odd = odd.next
            even.next= odd.next
            even = even.next
        odd.next = even_head
        return head
```

876. Middle of the Linked List

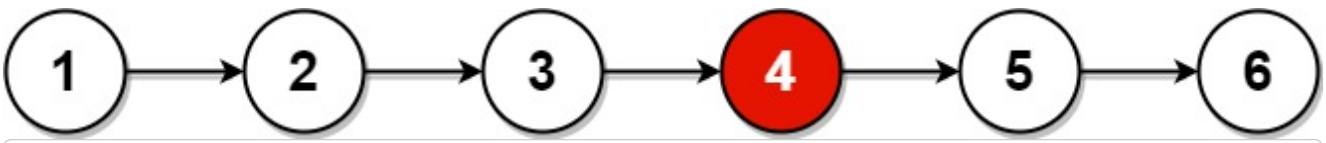
Given the `head` of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

Example 1:


```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.
```

Example 2:




```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4, we return the s
```

Constraints:

- The number of nodes in the list is in the range `[1, 100]`.
- `1 <= Node.val <= 100`

```
class Solution:
    def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head:return head

        slow = head
        fast = head
        while fast != None and fast.next != None:
            fast = fast.next.next
            slow = slow.next
        return slow
```

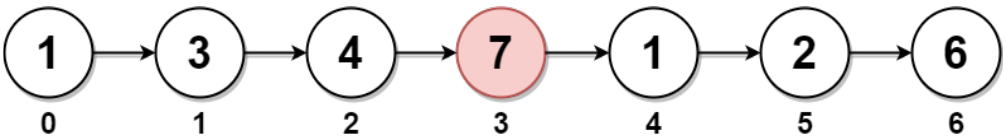
2095. Delete the Middle Node of a Linked List

You are given the `head` of a linked list. **Delete** the **middle node**, and return *the* `head` *of the modified linked list*.

The **middle node** of a linked list of size `n` is the `⌊n / 2⌋th` node from the **start** using **0-based indexing**, where `⌊x⌋` denotes the largest integer less than or equal to `x`.

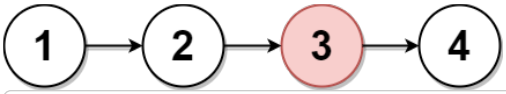
- For `n = 1, 2, 3, 4`, and `5`, the middle nodes are `0, 1, 1, 2`, and `2`, respectively.

Example 1:



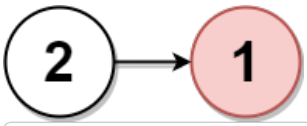

```
Input: head = [1,3,4,7,1,2,6]
Output: [1,3,4,1,2,6]
Explanation:
The above figure represents the given linked list. The indices of the nodes are writt
Since n = 7, node 3 with value 7 is the middle node, which is marked in red.
We return the new list after removing this node.
```

Example 2:




```
Input: head = [1,2,3,4]
Output: [1,2,4]
Explanation:
The above figure represents the given linked list.
For n = 4, node 2 with value 3 is the middle node, which is marked in red.
```

Example 3:




```
Input: head = [2,1]
Output: [2]
Explanation:
The above figure represents the given linked list.
For n = 2, node 1 with value 1 is the middle node, which is marked in red.
Node 0 with value 2 is the only node remaining after removing node 1.
```

Constraints:

- The number of nodes in the list is in the range `[1, 10^5]`.
- `1 <= Node.val <= 10^5`

```
class Solution:
    def deleteMiddle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head or head.next == None:
            return None
        dummy = ListNode(-1)
        dummy.next = head
        slow = fast =head
        prev  = None
        while fast and fast.next:
            prev = slow
            slow = slow.next
            fast = fast.next.next

        prev.next = slow.next
        return dummy.next
```