Linked List

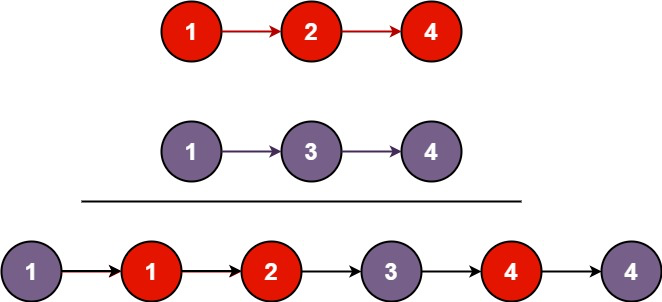
Monday, 30 November 2020

11:57 AM

* 1. **Merge Two Sorted Lists**

Merge two sorted linked lists and return it as a new **sorted** list. The new list should be made by splicing together the nodes of the first two lists.

**Example 1:**



**Input:** l1 = [1,2,4], l2 = [1,3,4]  
**Output:** [1,1,2,3,4,4]

**Example 2:**

**Input:** l1 = [], l2 = []  
**Output:** []

**Example 3:**

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

**Constraints:**

* 1. The number of nodes in both lists is in the range [0, 50].
  2. -100 <= Node.val <= 100
  3. Both l1 and l2 are sorted in **non-decreasing** order.

*Similar to merge sort*

class Solution:

def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:

dummy = ListNode()

prev = dummy

while l1 and l2:

if l1.val<l2.val:

prev.next = l1

l1 = l1.next

else:

prev.next = l2

l2 = l2.next

#prev.next.next = None

prev = prev.next

if l1:

prev.next = l1

if l2:

prev.next = l2

return dummy.next

* 1. **Merge k Sorted Lists**

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

*Merge all the linked-lists into one sorted linked-list and return it.*

**Example 1:**

**Input:** lists = [[1,4,5],[1,3,4],[2,6]]  
**Output:** [1,1,2,3,4,4,5,6]  
**Explanation:** The linked-lists are:  
[  
 1->4->5,  
 1->3->4,  
 2->6  
]  
merging them into one sorted list:  
1->1->2->3->4->4->5->6

**Example 2:**

**Input:** lists = []  
**Output:** []

**Example 3:**

**Input:** lists = [[]]  
**Output:** []

**Constraints:**

* + k == lists.length
  + 0 <= k <= 10^4
  + 0 <= lists[i].length <= 500
  + -10^4 <= lists[i][j] <= 10^4
  + lists[i] is sorted in **ascending order**.
  + The sum of lists[i].length won't exceed 10^4.

Approach 2: Compare one by one

**Algorithm**

* + Compare every \text{k}k nodes (head of every linked list) and get the node with the smallest value.
  + Extend the final sorted linked list with the selected nodes.

Refer the url : <https://leetcode.com/problems/merge-k-sorted-lists/solution/> (Approach 2)

**Complexity Analysis**

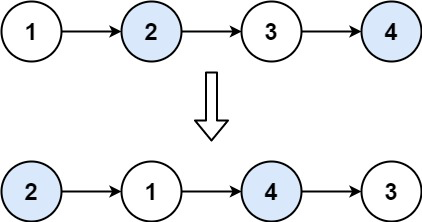
* + Time complexity : O(kN)*O*(*kN*) where \text{k}k is the number of linked lists.
    - Almost every selection of node in final linked costs O(k)*O*(*k*) (\text{k-1}k-1 times comparison).
    - There are N*N* nodes in the final linked list.
  + Space complexity :
    - O(n)*O*(*n*) Creating a new linked list costs O(n)*O*(*n*) space.
    - O(1)*O*(1) It's not hard to apply in-place method - connect selected nodes instead of creating new nodes to fill the new linked list.

* + **Swap Nodes in pairs**

Given a linked list, swap every two adjacent nodes and return its head.

You may **not** modify the values in the list's nodes. Only nodes itself may be changed.

**Example 1:**



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

**Example 2:**

**Input:** head = []  
**Output:** []

**Example 3:**

**Input:** head = [1]  
**Output:** [1]

**Constraints:**

* + The number of nodes in the list is in the range [0, 100].
  + 0 <= Node.val <= 100

<https://leetcode.com/problems/swap-nodes-in-pairs/discuss/171788/Python-or-Dummynode>

class Solution(object):  
 def swapPairs(self, head):  
 if not head or not head.next: return head  
 dummy = ListNode(0)  
 dummy.next = head  
 cur = dummy  
   
 while cur.next and cur.next.next:  
 first = cur.next  
 sec = cur.next.next  
 cur.next = sec  
 first.next = sec.next  
 sec.next = first  
 cur = cur.next.next  
 return dummy.next

* + **Rotate List**

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

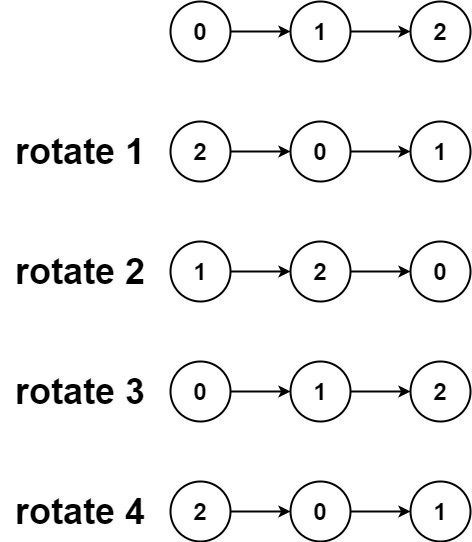
**Example 1:**

Diagram

Description automatically generated

**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 \* 109

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

class Solution:  
 def rotateRight(self, head: ListNode, k: int) -> ListNode:  
   
 if not head:  
 return None  
   
 lastElement = head  
 length = 1  
 # get the length of the list and the last node in the list  
 while ( lastElement.next ):  
 lastElement = lastElement.next  
 length += 1

# If k is equal to the length of the list then k == 0  
 # ElIf k is greater than the length of the list then k = k % length  
 k = k % length  
   
 # Set the last node to point to head node  
 # The list is now a circular linked list with last node pointing to first node  
 lastElement.next = head  
   
 # Traverse the list to get to the node just before the ( length - k )th node.  
 # Example: In 1->2->3->4->5, and k = 2  
 # we need to get to the Node(3)  
 tempNode = head  
 for \_ in range( length - k - 1 ):  
 tempNode = tempNode.next  
   
 # Get the next node from the tempNode and then set the tempNode.next as None  
 # Example: In 1->2->3->4->5, and k = 2  
 # tempNode = Node(3)  
 # answer = Node(3).next => Node(4)  
 # Node(3).next = None ( cut the linked list from here )  
 answer = tempNode.next  
 tempNode.next = None  
   
 return answer

* + **Remove Duplicates from Sorted List**

Given a sorted linked list, delete all duplicates such that each element appear only *once*.

**Example 1:**

**Input:** 1->1->2  
**Output:** 1->2

**Example 2:**

**Input:** 1->1->2->3->3  
**Output:** 1->2->3

This is a simple problem that merely tests your ability to manipulate list node pointers. Because the input list is sorted, we can determine if a node is a duplicate by comparing its value to the node *after* it in the list. If it is a duplicate, we change the next pointer of the current node so that it skips the next node and points directly to the one after the next node.

class Solution(object):  
 def deleteDuplicates(self, head):  
 """  
 :type head: ListNode  
 :rtype: ListNode  
 """  
 if head == None:  
 return head  
   
 current = head.next  
 prev = head  
   
 while current != None:  
 if current.val == prev.val:  
 prev.next = current.next  
 current = current.next  
 else:  
 current = current.next  
 prev = prev.next  
   
 return head

* 1. **Remove Duplicates from Sorted List II**

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.

Return the linked list sorted as well.

**Example 1:**

**Input:** 1->2->3->3->4->4->5  
**Output:** 1->2->5

**Example 2:**

**Input:** 1->1->1->2->3  
**Output:** 2->3

def deleteDuplicates(self, head):  
 dummy = pre = ListNode(0)  
 dummy.next = head  
 while head and head.next:  
 if head.val == head.next.val:  
 while head and head.next and head.val == head.next.val:  
 head = head.next  
 head = head.next  
 pre.next = head  
 else:  
 pre = pre.next  
 head = head.next  
 return dummy.next

* 1. **Partition List**

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.

You should preserve the original relative order of the nodes in each of the two partitions.

**Example:**

**Input:** head = 1->4->3->2->5->2, *x* = 3  
**Output:** 1->2->2->4->3->5

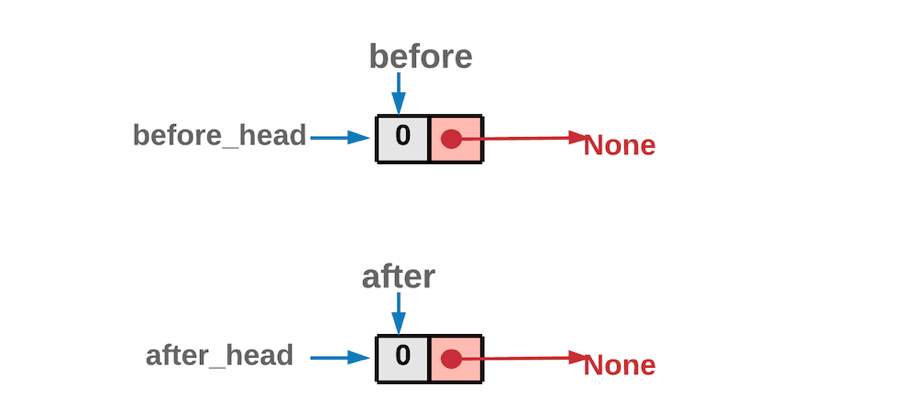
Approach 1: Two Pointer Approach

**Intuition**

We can take two pointers before and after to keep track of the two linked lists as described above. These two pointers could be used two create two separate lists and then these lists could be combined to form the desired reformed list.

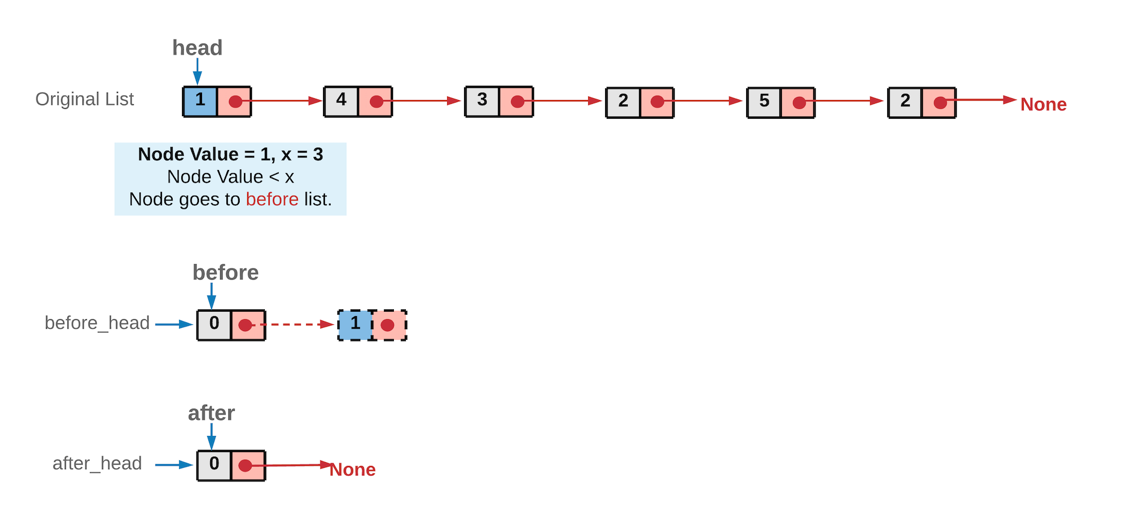
**Algorithm**

* 1. Initialize two pointers before and after. In the implementation we have initialized these two with a dummy ListNode. This helps to reduce the number of conditional checks we would need otherwise. You can try an implementation where you don't initialize with a dummy node and see it yourself!

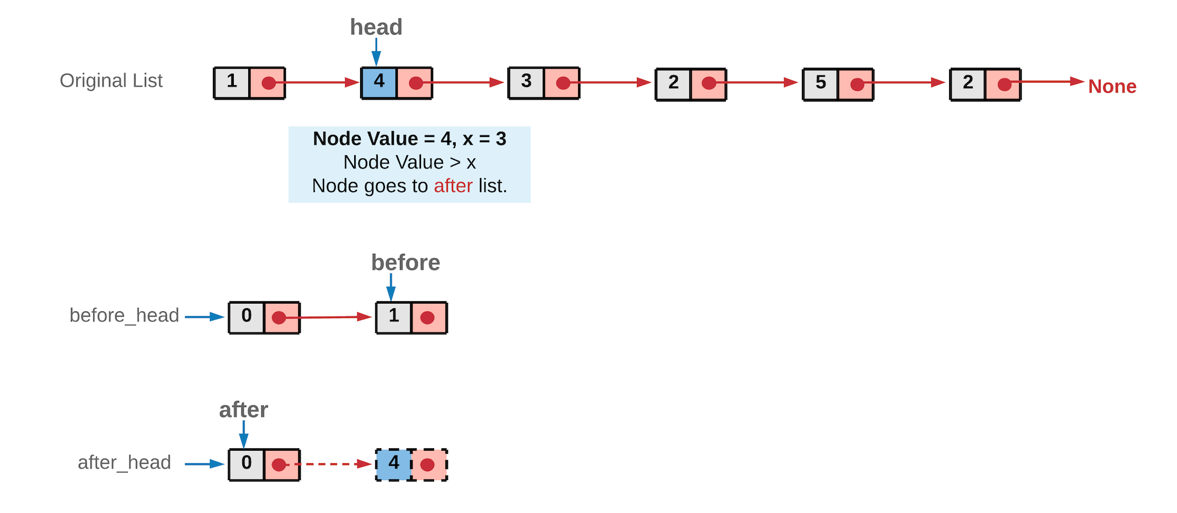


Dummy Node Initialization

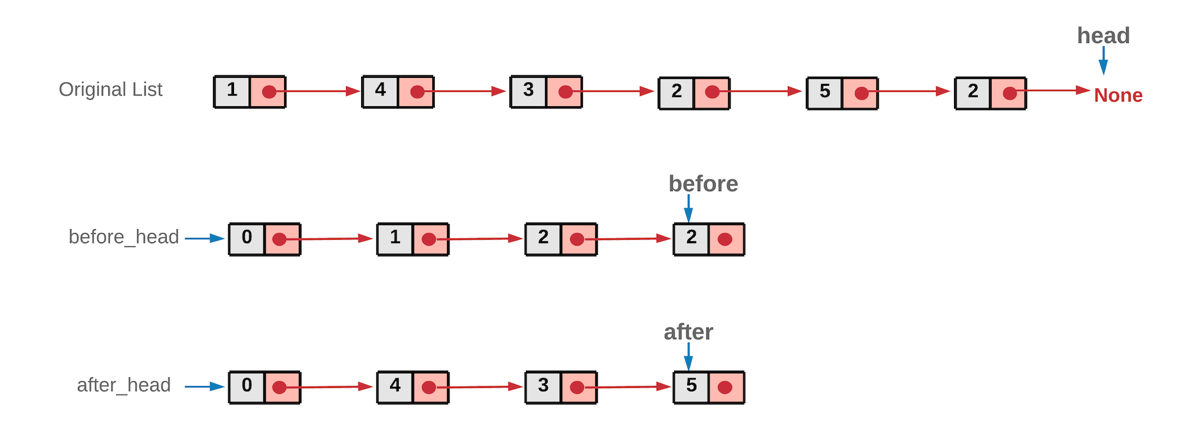
* 1. Iterate the original linked list, using the head pointer.
  2. If the node's value pointed by head is *lesser* than x, the node should be part of the before list. So we move it to before list.



* 1. Else, the node should be part of after list. So we move it to after list.

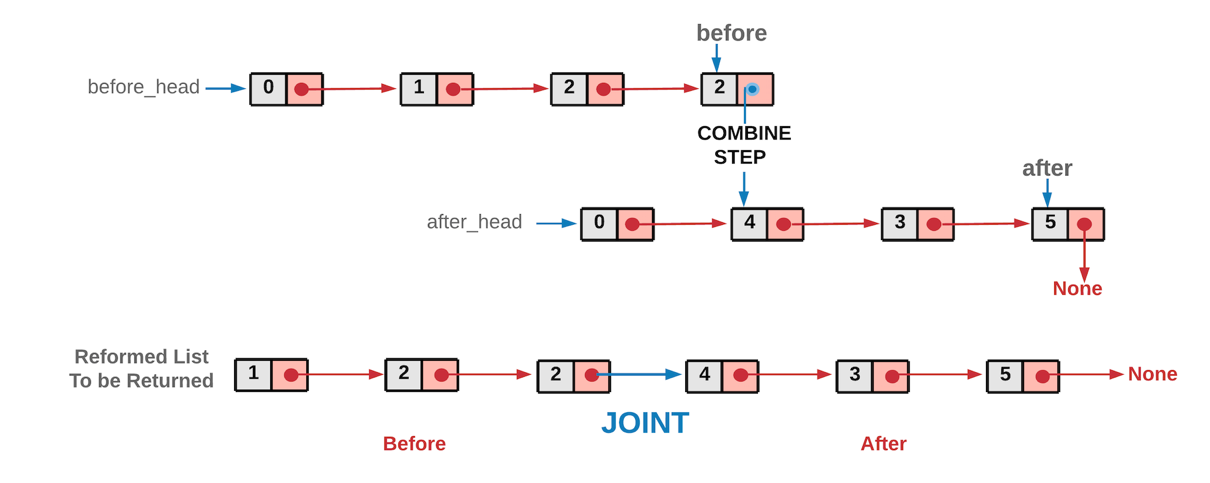


* 1. Once we are done with all the nodes in the original linked list, we would have two list before and after. The original list nodes are either part of before list or after list, depending on its value.



*Note: Since we traverse the original linked list from left to right, at no point would the order of nodes change relatively in the two lists. Another important thing to note here is that we show the original linked list intact in the above diagrams. However, in the implementation, we remove the nodes from the original linked list and attach them in the before or after list. We don't utilize any additional space. We simply move the nodes from the original list around.*

* 1. Now, these two lists before and after can be combined to form the reformed list.



We did a dummy node initialization at the start to make implementation easier, you don't want that to be part of the returned list, hence just move ahead one node in both the lists while combining the two list. Since both before and after have an extra node at the front.

class Solution(object):

def partition(self, head, x):

"""

:type head: ListNode

:type x: int

:rtype: ListNode

"""

# before and after are the two pointers used to create two list

# before\_head and after\_head are used to save the heads of the two lists.

# All of these are initialized with the dummy nodes created.

before = before\_head = ListNode(0)

after = after\_head = ListNode(0)

while head:

# If the original list node is lesser than the given x,

# assign it to the before list.

if head.val < x:

before.next = head

before = before.next

else:

# If the original list node is greater or equal to the given x,

# assign it to the after list.

after.next = head

after = after.next

# move ahead in the original list

head = head.next

# Last node of "after" list would also be ending node of the reformed list

after.next = None

# Once all the nodes are correctly assigned to the two lists,

# combine them to form a single list which would be returned.

before.next = after\_head.next

return before\_head.next

* 1. **Reverse Linked List II**

Reverse a linked list from position *m* to *n*. Do it in one-pass.

**Note:**1 ≤ *m* ≤ *n* ≤ length of list.

**Example:**

**Input:** 1->2->3->4->5->NULL, *m* = 2, *n* = 4  
**Output:** 1->4->3->2->5->NULL

Suppose we have at our disposal, two pointers. One of them points to the node A and the other one points to the node B. Let's call these pointers prev and cur respectively. We can simply use these two pointers to reverse the link between A and B.

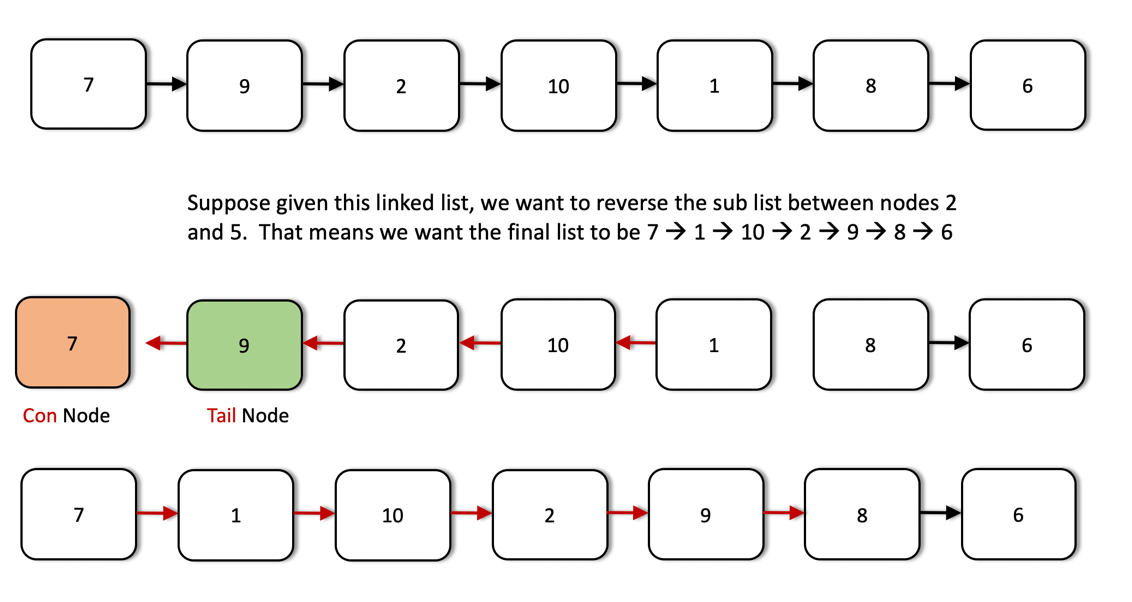
cur.next = prev

The only problem with this is, we don't have a way of progressing further i.e. once we do this, we can't reach the node C. That's why we need a third pointer that will help us continue the link reversal process. So, we do the following instead.

third = cur.next  
cur.next = prev  
prev = cur  
cur = third

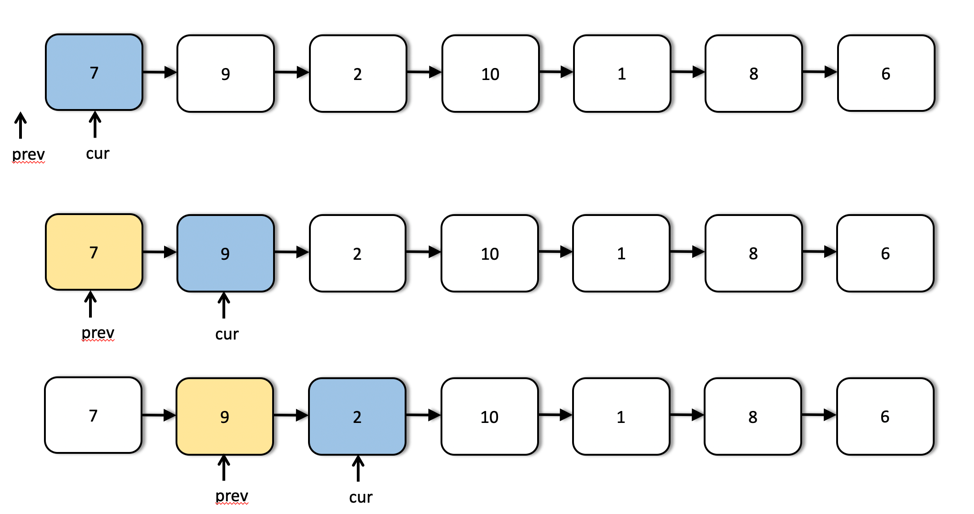
We do the above *iteratively* and we will achieve what the question asks us to do. Let's look at the steps for the algorithm now.

* 1. We need two pointers, prev and cur as explained above.
  2. The prev pointer should be initialized to None initially while cur is initialized to the head of the linked list.
  3. We progress the cur pointer one step at a time and the prev pointer follows it.
  4. We keep progressing the two pointers in this way until the cur pointer reaches the m^{th}*mth* node from the beginning of the list. This is the point from where we start reversing our linked list.
  5. An important thing to note here is the usage of two additional pointers which we will call as tail and con. The tail pointer points to the m^{th}*mth* node from the beginning of the linked list and we call it a *tail* pointer since this node becomes the tail of the reverse sublist. The con points to the node one before m^{th}*mth* node and this connects to the new head of the reversed sublist. Let's take a look at a figure to understand these two pointers better.

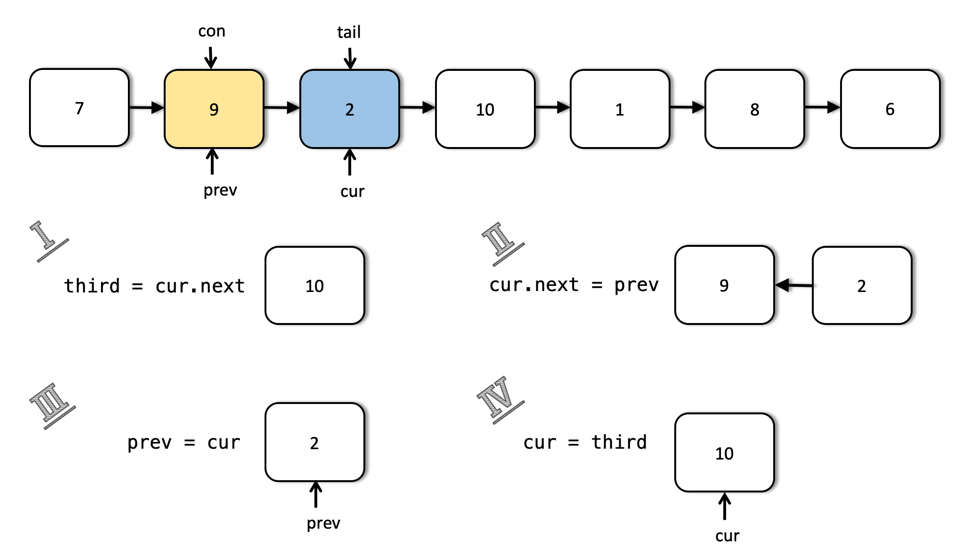


* 1. The tail and the con pointers are set once initially and then used in the end to finish the linked list reversal.
  2. Once we reach the m^{th}*mth* node, we iteratively reverse the links as explained before using the two pointers. We keep on doing this until we are done reversing the link (next pointer) for the n^{th}*nth* node. At that point, the prev pointer would point to the n^{th}*nth* node.
  3. We use the con pointer to attach to the prev pointer since the node now pointed to by the prev pointer (the n^{th}*nth* node from the beginning) will come in place of the m^{th}*mth* node due after the reversal. Similarly, we will make use of the tail pointer to connect to the node next to the prev node i.e. (n+1)^{th}(*n*+1)*th* node from the beginning.

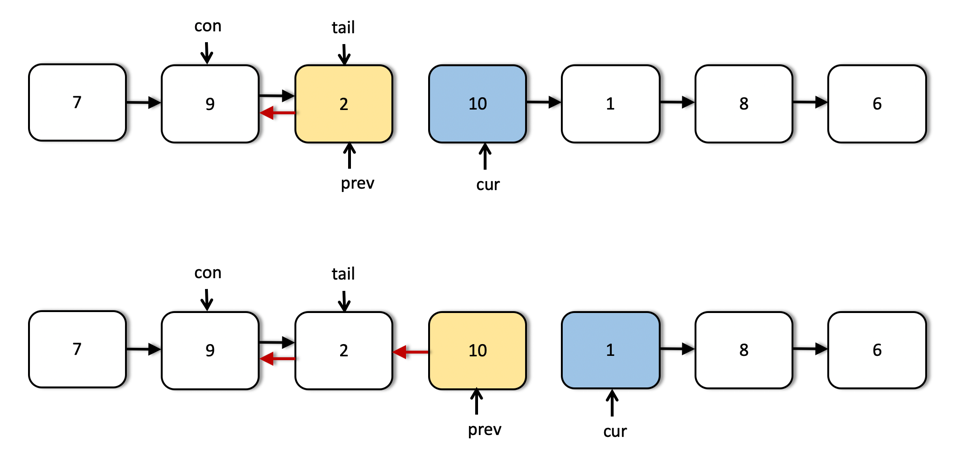
Let's have a look at the algorithm execute on a sample linked list to make the use case for all these pointers clearer. We are given a linked list initially with elements 7 → 9 → 2 → 10 → 1 → 8 → 6 and we need to reverse the list from node 3 through 6.

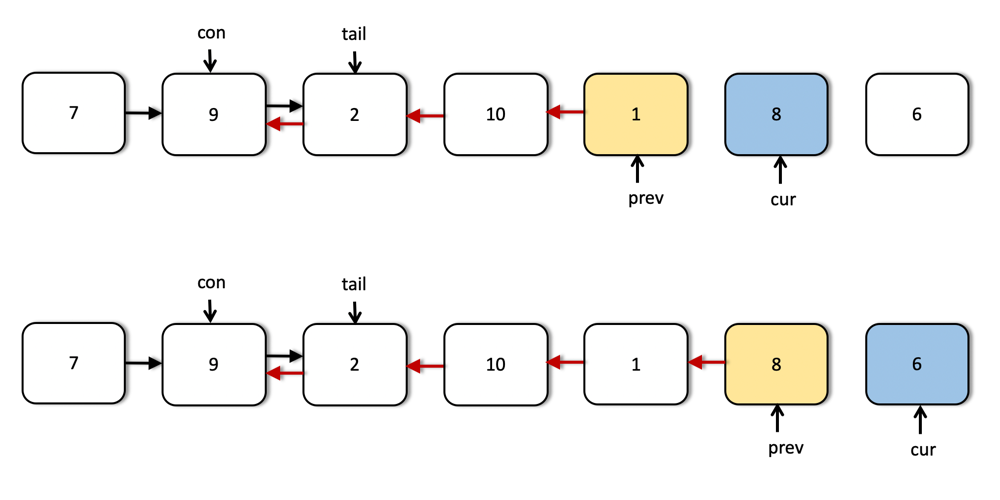


We can see the first few steps of our iterative solution above. The first step shows the initialization of the two pointers and the third step shows us the starting point for the list reversal process.

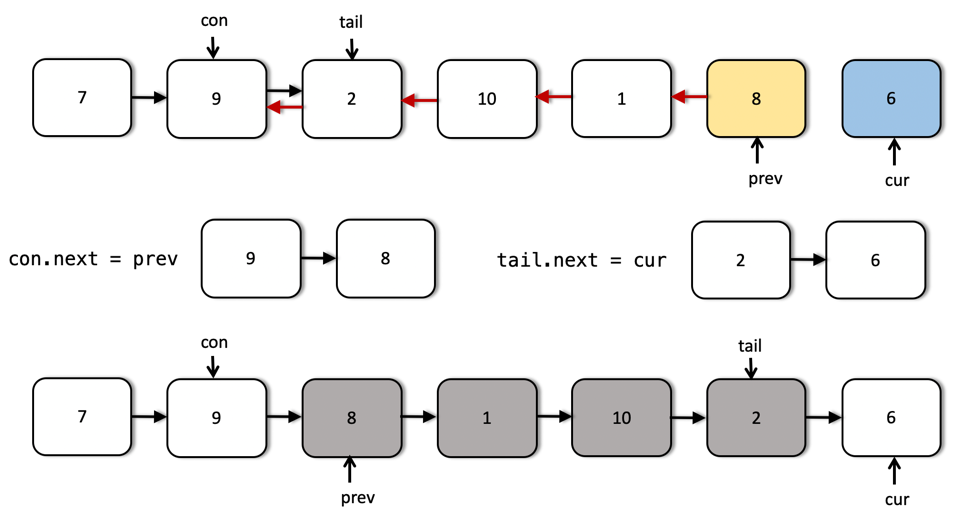


This shows us in detail how the links are reversed and how we move forward after reversing the links between two nodes. This step is done multiple times as shown in the following images.





As we can see from the above images, now the two pointers have reached their final positions. We are done reversing the sublist that we were required to do i.e. nodes 3 through 6. However, we still have to fix some connections. The next image explains how we use the tail and con pointers to make the final connections.



class Solution:

def reverseBetween(self, head, m, n):

"""

:type head: ListNode

:type m: int

:type n: int

:rtype: ListNode

"""

# Empty list

if not head:

return None

# Move the two pointers until they reach the proper starting point

# in the list.

cur, prev = head, None

while m > 1:

prev = cur

cur = cur.next

m, n = m - 1, n - 1

# The two pointers that will fix the final connections.

tail, con = cur, prev

# Iteratively reverse the nodes until n becomes 0.

while n:

third = cur.next

cur.next = prev

prev = cur

cur = third

n -= 1

# Adjust the final connections as explained in the algorithm

if con:

con.next = prev

else:

head = prev

tail.next = cur

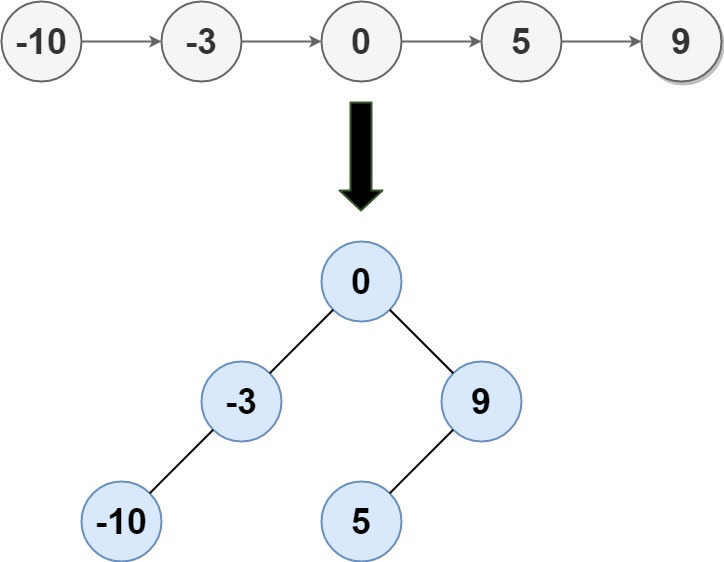
return head

* 1. **Convert Sorted List to Binary Search Tree**

Given the head of a singly linked list where elements are **sorted in ascending order**, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

**Example 1:**



**Input:** head = [-10,-3,0,5,9]  
**Output:** [0,-3,9,-10,null,5]  
**Explanation:** One possible answer is [0,-3,9,-10,null,5], which represents the shown height balanced BST.

**Example 2:**

**Input:** head = []  
**Output:** []

**Example 3:**

**Input:** head = [0]  
**Output:** [0]

**Example 4:**

**Input:** head = [1,3]  
**Output:** [3,1]

**Constraints:**

* + The number of nodes in head is in the range [0, 2 \* 104].
  + -10^5 <= Node.val <= 10^5

<https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/discuss/35474/Python-recursive-solution-with-detailed-comments-(operate-linked-list-directly)>.

<https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/discuss/35526/Python-solutions-(convert-to-array-first-top-down-approach-bottom-up-approach)>

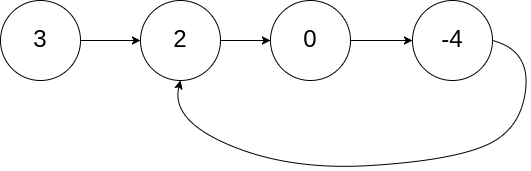
* + **Linked List Cycle II**

Given 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. **Note that pos is not passed as a parameter**.

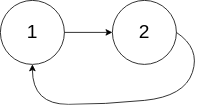
**Notice** that you **should not modify** the linked list.

**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 node.

**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 node.

**Example 3:**

A picture containing diagram

Description automatically generated

**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, 104].
  + -105 <= Node.val <= 105
  + pos is -1 or a **valid index** in the linked-list.

Consider the following linked list, where E is the cylce entry and X, the crossing point of fast and slow.  
 H: distance from head to cycle entry E  
 D: distance from E to X  
 L: cycle length  
 \_\_\_\_\_  
 / \  
 head\_\_\_\_\_H\_\_\_\_\_\_E \  
 \ /  
 X\_\_\_\_\_/   
   
   
 If fast and slow both start at head, when fast catches slow, slow has traveled H+D and fast 2(H+D).   
 Assume fast has traveled n loops in the cycle, we have:  
 2H + 2D = H + D + L --> H + D = nL --> H = nL - D  
 Thus if two pointers start from head and X, respectively, one first reaches E, the other also reaches E.   
 In my solution, since fast starts at head.next, we need to move slow one step forward in the beginning of part 2

class Solution:  
 # @param head, a ListNode  
 # @return a list node  
 def detectCycle(self, head):  
 try:  
 fast = head.next  
 slow = head  
 while fast is not slow:  
 fast = fast.next.next  
 slow = slow.next  
 except:  
 # if there is an exception, we reach the end and there is no cycle  
 return None

# since fast starts at head.next, we need to move slow one step forward  
 slow = slow.next  
 while head is not slow:  
 head = head.next  
 slow = slow.next

return head

* + **Reorder List**

Given a singly linked list *L*: *L*0→*L*1→…→*Ln*-1→*L*n,

reorder it to: *L*0→*Ln*→*L*1→*Ln*-1→*L*2→*Ln*-2→…

You may **not** modify the values in the list's nodes, only nodes itself may be changed.

**Example 1:**

Given 1->2->3->4, reorder it to 1->4->2->3.

**Example 2:**

Given 1->2->3->4->5, reorder it to 1->5->2->4->3.

to succeed we need to do the following steps:

* 1. Find the middle of or list - be careful, it needs to work properly both for even and for odd number of nodes. For this we can either just count number of elements and then divide it by to, and do two traversals of list. Or we can use slow/fast iterators trick, where slow moves with speed 1 and fast moves with speed 2. Then when fast reches the end, slow will be in the middle, as we need.
  2. Reverse the second part of linked list. Again, if you never done it before, it can be quite painful, please read oficial solution to problem **206. Reverse Linked List**. The idea is to keep **three** pointers: prev, curr, nextt stand for previous, current and next and change connections in place. Do not forget to use slow.next = None, in opposite case you will have list with loop.
  3. Finally, we need to merge two lists, given its heads. These heads are denoted by head and prev, so for simplisity I created head1 and head2 variables. What we need to do now is to interchange nodes: we put head2 as next element of head1 and then say that head1 is now head2 and head2 is previous head1.next. In this way we do one step for one of the lists and rename lists, so next time we will take element from head2, then rename again and so on.

**Complexity**: Time complexity is O(n), because we first do O(n) iterations to find middle, then we do O(n) iterations to reverse second half and finally we do O(n) iterations to merge lists. Space complexity is O(1).

class Solution:  
 def reorderList(self, head):  
 #step 1: find middle  
 if not head: return []  
 slow, fast = head, head  
 while fast.next and fast.next.next:  
 slow = slow.next  
 fast = fast.next.next  
   
 #step 2: reverse second half  
 prev, curr = None, slow.next  
 while curr:  
 nextt = curr.next  
 curr.next = prev  
 prev = curr  
 curr = nextt   
 slow.next = None  
   
 #step 3: merge lists  
 head1, head2 = head, prev  
 while head2:  
 nextt = head1.next  
 head1.next = head2  
 head1 = head2  
 head2 = nextt

* 1. **Insertion Sort List**

**Algorithm of Insertion Sort:**

* 1. Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list.
  2. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.
  3. It repeats until no input elements remain.

**Example 1:**

**Input:** 4->2->1->3  
**Output:** 1->2->3->4

**Example 2:**

**Input:** -1->5->3->4->0  
**Output:** -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()

cur = head

while cur:

prev = dummy

nextNode = prev.next

while nextNode:

if cur.val<nextNode.val:

break

prev = nextNode

nextNode = nextNode.next

temp = cur.next

cur.next = nextNode

prev.next = cur

cur = temp

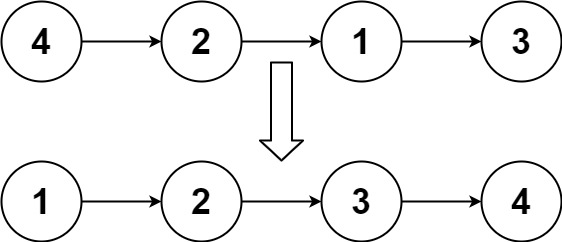
return dummy.next

* 1. **Sort List**

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

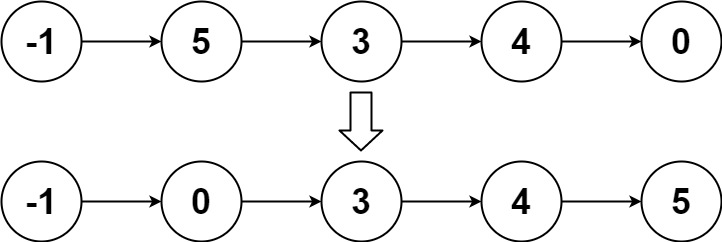
**Follow up:** Can you sort the linked list in O(n logn) time and O(1) memory (i.e. constant space)?

**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 \* 104].
  + -105 <= Node.val <= 105

<https://leetcode.com/problems/sort-list/solution/>

* + **Remove Linked List Elements**

Remove all elements from a linked list of integers that have value ***val***.

**Example:**

**Input:** 1->2->6->3->4->5->6, ***val*** = 6  
**Output:** 1->2->3->4->5

<https://leetcode.com/problems/remove-linked-list-elements/discuss/158651/Simple-Python-solution-with-explanation-(single-pointer-dummy-head)>.

* 1. **Add Two Numbers II**

You are given two **non-empty** linked lists representing two non-negative integers. The most significant digit comes first and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

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

**Follow up:**

What if you cannot modify the input lists? In other words, reversing the lists is not allowed.

**Example:**

**Input:** (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)  
**Output:** 7 -> 8 -> 0 -> 7

# 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:

s1 = []

s2 = []

while l1!=None:

s1.append(l1.val)

l1 = l1.next

while l2!=None:

s2.append(l2.val)

l2 = l2.next

res = ListNode(-1)

carry = 0

#print(s1, s2)

while s1 and s2:

n1 = s1.pop()

n2 = s2.pop()

carry, res.val = divmod(n1+n2+carry, 10)

temp = ListNode(-1)

temp.next = res

res = temp

while s1:

n1 = s1.pop()

carry, res.val = divmod(n1+carry, 10)

temp = ListNode(-1)

temp.next = res

res = temp

while s2:

n2 = s2.pop()

carry, res.val = divmod(n2+carry, 10)

temp = ListNode(-1)

temp.next = res

res = temp

if carry:

res.val = carry

return res if res.val!=-1 else res.next

* 1. **Split Linked List in Parts**

Given a (singly) linked list with head node root, write a function to split the linked list into k consecutive linked list "parts".

The length of each part should be as equal as possible: no two parts should have a size differing by more than 1. This may lead to some parts being null.

The parts should be in order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal parts occurring later.

Return a List of ListNode's representing the linked list parts that are formed.

Examples 1->2->3->4, k = 5 // 5 equal parts [ [1], [2], [3], [4], null ]

**Example 1:**

**Input:**  
root = [1, 2, 3], k = 5  
**Output:** [[1],[2],[3],[],[]]  
**Explanation:**  
The input and each element of the output are ListNodes, not arrays.  
For example, the input root has root.val = 1, root.next.val = 2, \root.next.next.val = 3, and root.next.next.next = null.  
The first element output[0] has output[0].val = 1, output[0].next = null.  
The last element output[4] is null, but it's string representation as a ListNode is [].

**Example 2:**

**Input:**   
root = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], k = 3  
**Output:** [[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]  
**Explanation:**  
The input has been split into consecutive parts with size difference at most 1, and earlier parts are a larger size than the later parts.

**Note:**

* + The length of root will be in the range [0, 1000].
  + Each value of a node in the input will be an integer in the range [0, 999].
  + k will be an integer in the range [1, 50].

If there are N*N* nodes in the linked list root, then there are N / k*N*/*k* items in each part, plus the first N \% k*N*%*k* parts have an extra item. We can count N*N* with a simple loop.

Now for each part, we have calculated how many nodes that part will have: width + (i < remainder ? 1 : 0). We create a new list and write the part to that list.

Our solution showcases constructs of the form a = b = c. Note that this syntax behaves differently for different languages.

class Solution(object):

def splitListToParts(self, root, k):

cur = root

for N in xrange(1001):

if not cur: break

cur = cur.next

width, remainder = divmod(N, k)

ans = []

cur = root

for i in xrange(k):

head = write = ListNode(None)

for j in xrange(width + (i < remainder)):

write.next = write = ListNode(cur.val)

if cur: cur = cur.next

ans.append(head.next)

return ans

* + **Linked List Components**

We are given head, the head node of a linked list containing **unique integer values**.

We are also given the list G, a subset of the values in the linked list.

Return the number of connected components in G, where two values are connected if they appear consecutively in the linked list.

**Example 1:**

**Input:**   
head: 0->1->2->3  
G = [0, 1, 3]  
**Output:** 2  
**Explanation:**   
0 and 1 are connected, so [0, 1] and [3] are the two connected components.

**Example 2:**

**Input:**   
head: 0->1->2->3->4  
G = [0, 3, 1, 4]  
**Output:** 2  
**Explanation:**   
0 and 1 are connected, 3 and 4 are connected, so [0, 1] and [3, 4] are the two connected components.

**Note:**

* + If N is the length of the linked list given by head, 1 <= N <= 10000.
  + The value of each node in the linked list will be in the range [0, N - 1].
  + 1 <= G.length <= 10000.
  + G is a subset of all values in the linked list.

**Intuition**

Instead of thinking about connected components in G, think about them in the linked list. Connected components in G must occur consecutively in the linked list.

**Algorithm**

Scanning through the list, if node.val is in G and node.next.val isn't (including if node.next is null), then this must be the end of a connected component.

For example, if the list is 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7, and G = [0, 2, 3, 5, 7], then when scanning through the list, we fulfill the above condition at 0, 3, 5, 7, for a total answer of 4.

class Solution(object):

def numComponents(self, head, G):

Gset = set(G)

cur = head

ans = 0

while cur:

if (cur.val in Gset and

getattr(cur.next, 'val', None) not in Gset):

ans += 1

cur = cur.next

return ans

* + **Next Greater Node In Linked List**

We are given a linked list with head as the first node.  Let's number the nodes in the list: node\_1, node\_2, node\_3, ... etc.

Each node may have a *next larger* **value**: for node\_i, next\_larger(node\_i) is the node\_j.val such that j > i, node\_j.val > node\_i.val, and j is the smallest possible choice.  If such a j does not exist, the next larger value is 0.

Return an array of integers answer, where answer[i] = next\_larger(node\_{i+1}).

Note that in the example **inputs** (not outputs) below, arrays such as [2,1,5] represent the serialization of a linked list with a head node value of 2, second node value of 1, and third node value of 5.

**Example 1:**

**Input:** [2,1,5]  
**Output:** [5,5,0]

**Example 2:**

**Input:** [2,7,4,3,5]  
**Output:** [7,0,5,5,0]

**Example 3:**

**Input:** [1,7,5,1,9,2,5,1]  
**Output:** [7,9,9,9,0,5,0,0]

**Note:**

* 1. 1 <= node.val <= 10^9 for each node in the linked list.
  2. The given list has length in the range [0, 10000].

We can use a stack that stores nodes in monotone decreasing order of value. When we see a node\_j with a larger value, every node\_i in the stack has next\_larger(node\_i) = node\_j .

Save <index, value> pair to the stack.

def nextLargerNodes(self, head):  
 res, stack = [], []  
 while head:  
 while stack and stack[-1][1] < head.val:  
 res[stack.pop()[0]] = head.val  
 stack.append([len(res), head.val])  
 res.append(0)  
 head = head.next  
 return res

* 1. **Merge In Between Linked Lists**

# Definition for singly-linked list.

# class ListNode:

# def \_\_init\_\_(self, val=0, next=None):

# self.val = val

# self.next = next

class Solution:

def mergeInBetween(self, list1: ListNode, a: int, b: int, list2: ListNode) -> ListNode:

count = 0

head = list1

prev = None

t = None

while count!=a:

temp = head

head = head.next

t = head

count+=1

while count!=b:

prev = head

head = head.next

count+=1

temp.next = list2

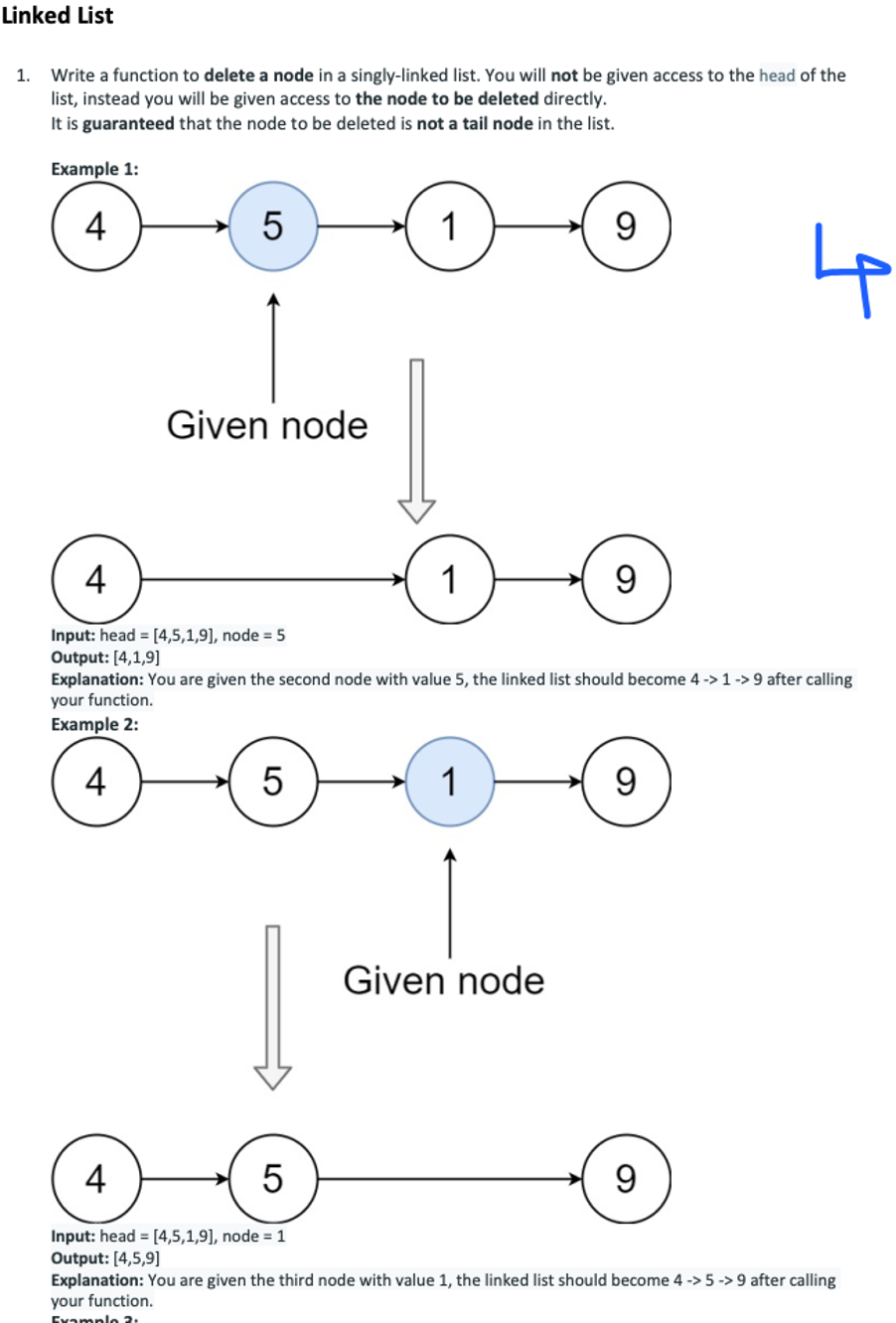
while list2.next!=None:

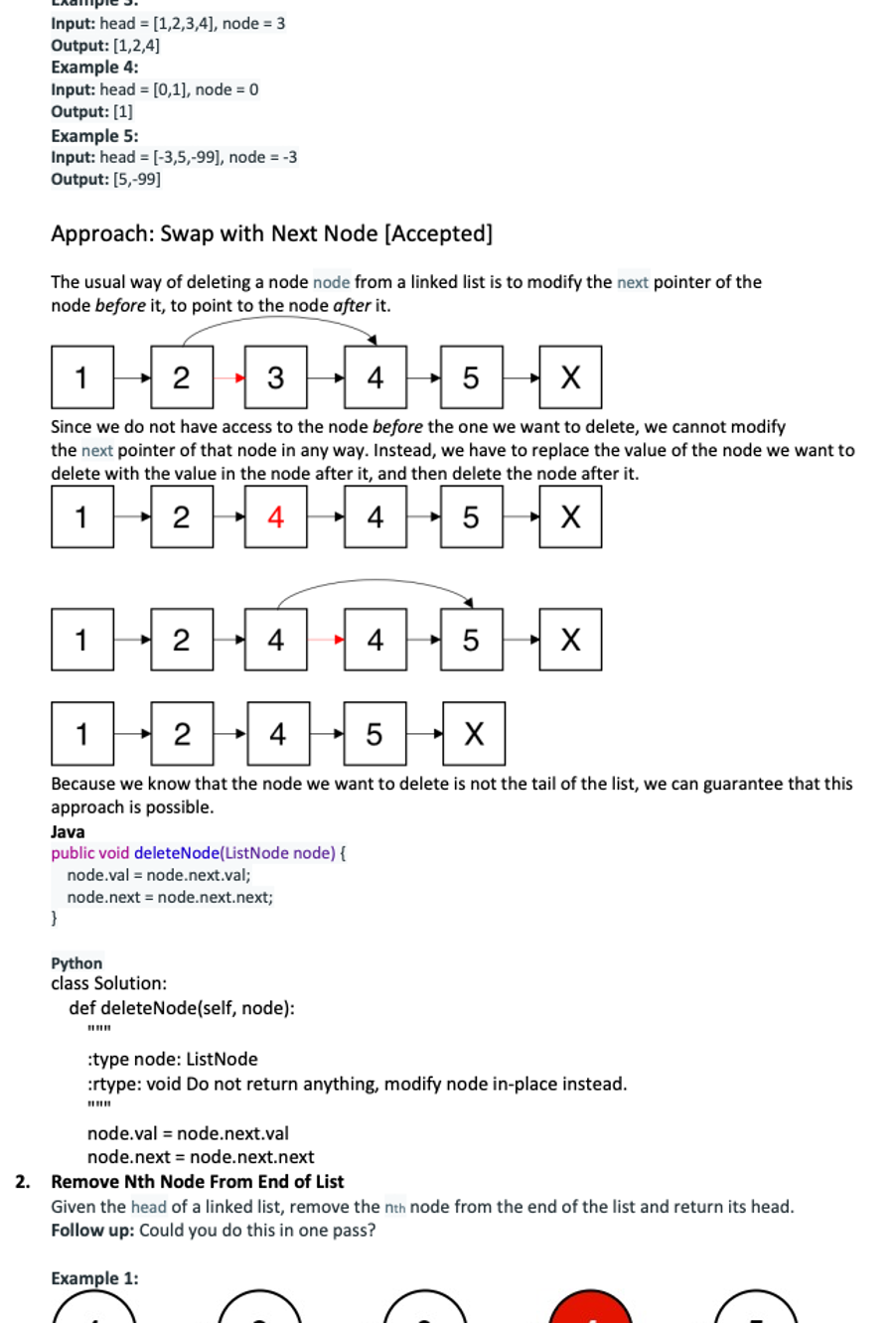
list2 = list2.next

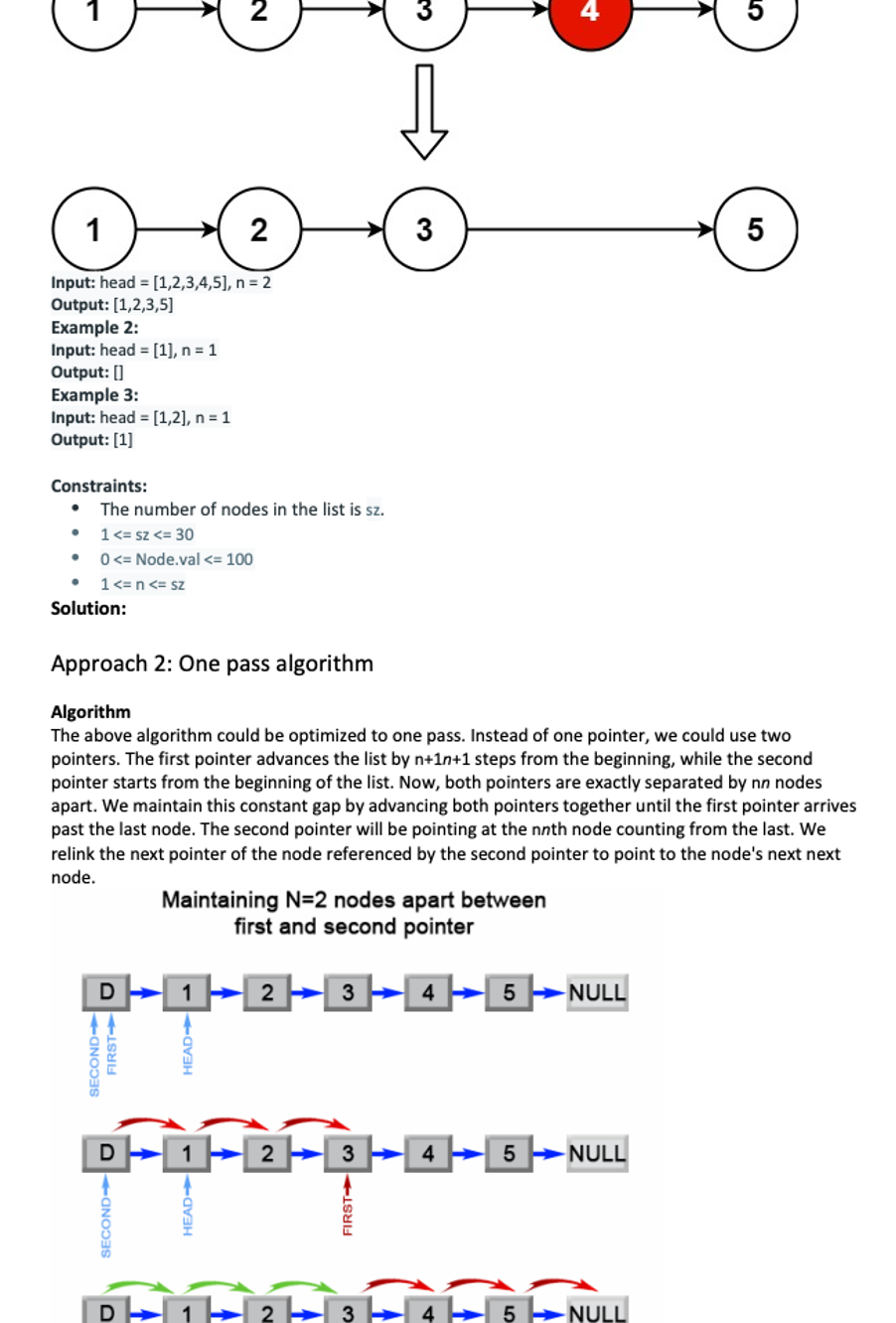
list2.next = head.next

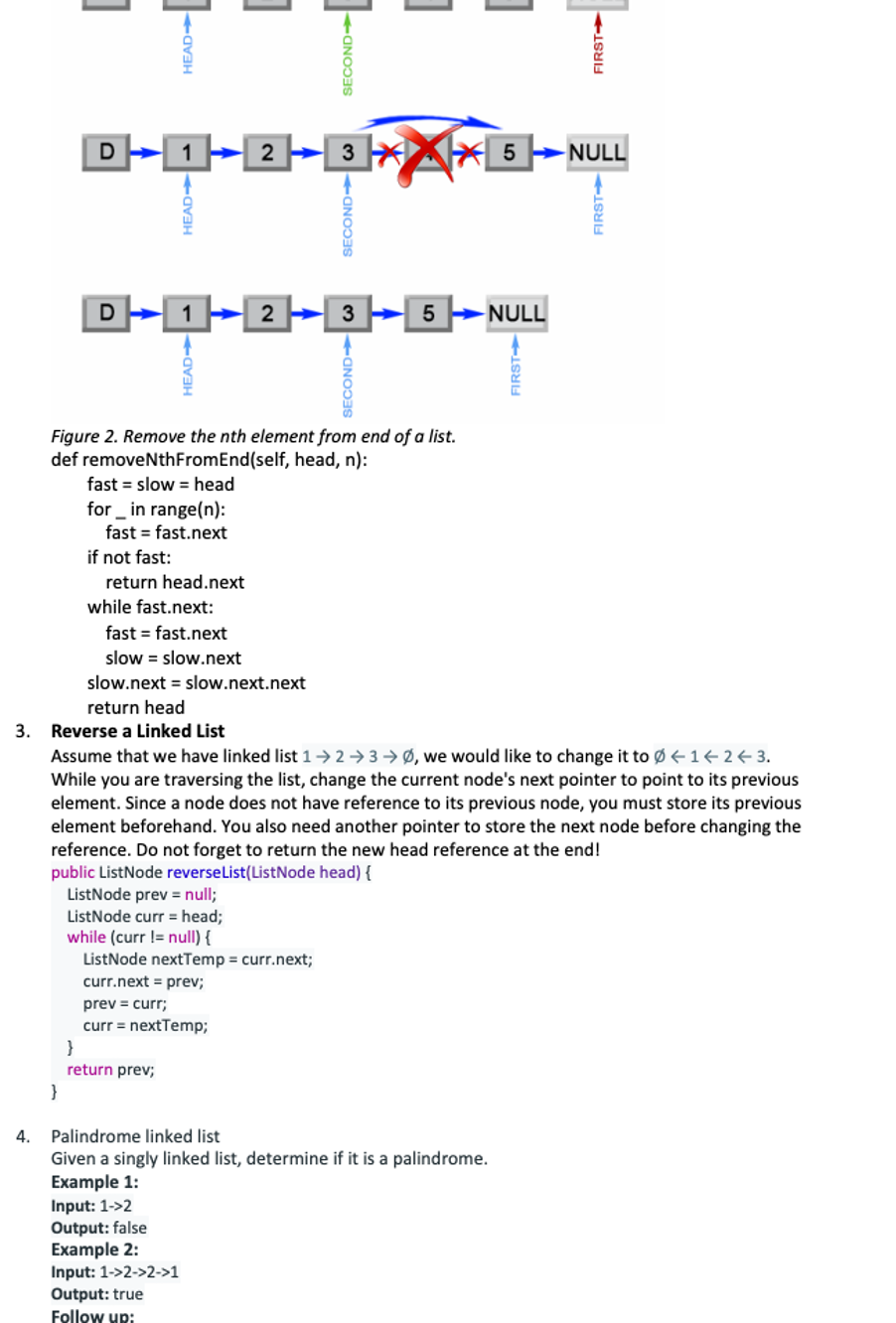
return list1

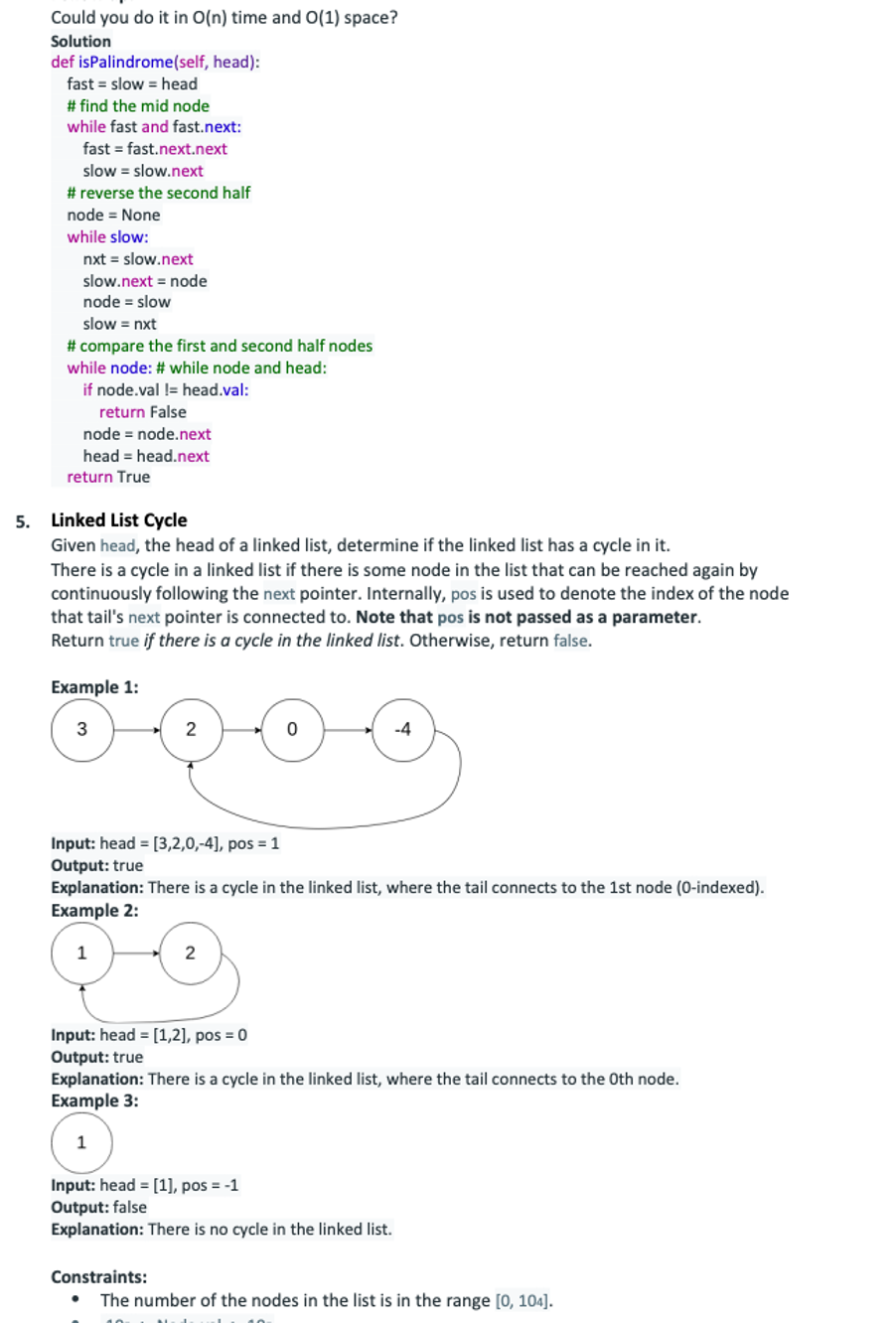
* 1. **Linked List in Binary Tree**

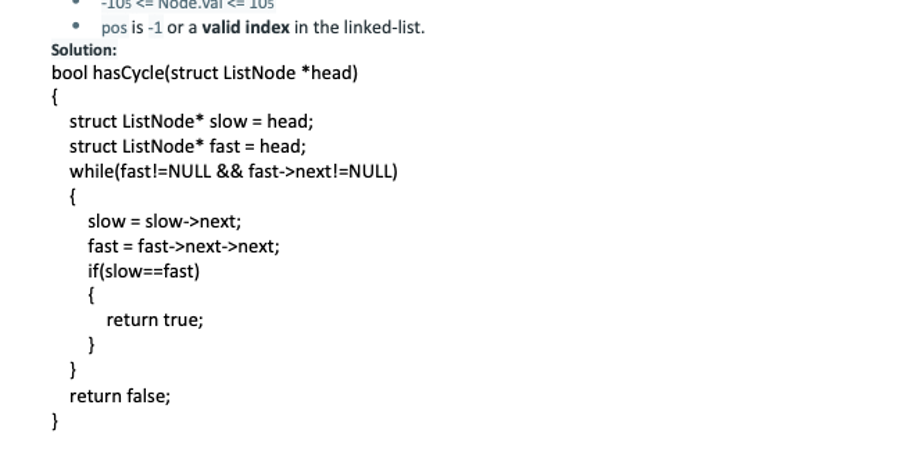










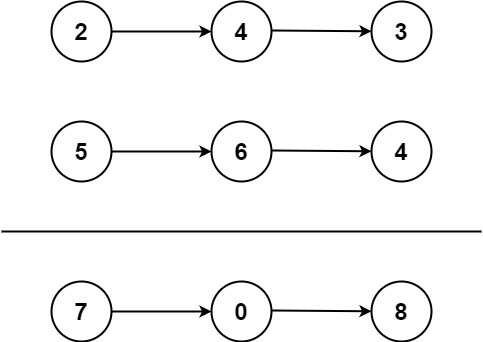


* 1. **Add Two Numbers**

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:**

* 1. The number of nodes in each linked list is in the range [1, 100].
  2. 0 <= Node.val <= 9
  3. It is guaranteed that the list represents a number that does not have leading zeros.

**Solution:**

class Solution:

def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:

s1 = []

s2 = []

while l1!=None:

s1.append(l1.val)

l1 = l1.next

while l2!=None:

s2.append(l2.val)

l2 = l2.next

res = ListNode(-1)

carry = 0

#print(s1, s2)

while s1 and s2:

n1 = s1.pop()

n2 = s2.pop()

carry, res.val = divmod(n1+n2+carry, 10)

temp = ListNode(-1)

temp.next = res

res = temp

while s1:

n1 = s1.pop()

carry, res.val = divmod(n1+carry, 10)

temp = ListNode(-1)

temp.next = res

res = temp

while s2:

n2 = s2.pop()

carry, res.val = divmod(n2+carry, 10)

temp = ListNode(-1)

temp.next = res

res = temp

if carry:

res.val = carry

return res if res.val!=-1 else res.next

* 1. **Odd Even Linked List**

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

**Example 1:**

**Input:** 1->2->3->4->5->NULL

**Output:** 1->3->5->2->4->NULL

**Example 2:**

**Input:** 2->1->3->5->6->4->7->NULL

**Output:** 2->3->6->7->1->5->4->NULL

**Constraints:**

* + The relative order inside both the even and odd groups should remain as it was in the input.
  + The first node is considered odd, the second node even and so on ...
  + The length of the linked list is between [0, 10^4].

**Solution:**

**Intuition**

Put the odd nodes in a linked list and the even nodes in another. Then link the evenList to the tail of the oddList.

**Algorithm**

The solution is very intuitive. But it is not trivial to write a concise and bug-free code.

A well-formed LinkedList need two pointers head and tail to support operations at both ends. The variables head and odd are the head pointer and tail pointer of one LinkedList we call oddList; the variables evenHead and even are the head pointer and tail pointer of another LinkedList we call evenList. The algorithm traverses the original LinkedList and put the odd nodes into the oddList and the even nodes into the evenList. To traverse a LinkedList we need at least one pointer as an iterator for the current node. But here the pointers odd and even not only serve as the tail pointers but also act as the iterators of the original list.

The best way of solving any linked list problem is to visualize it either in your mind or on a piece of paper. An illustration of our algorithm is following:

<https://leetcode.com/problems/odd-even-linked-list/Figures/328_Odd_Even.svg>

*Figure 1. Step by step example of the odd and even linked list.*

public class Solution {

public ListNode oddEvenList(ListNode head) {

if (head == null) return null;

ListNode odd = head, even = head.next, evenHead = even;

while (even != null && even.next != null) {

odd.next = even.next;

odd = odd.next;

even.next = odd.next;

even = even.next;

}

odd.next = evenHead;

return head;

}

}

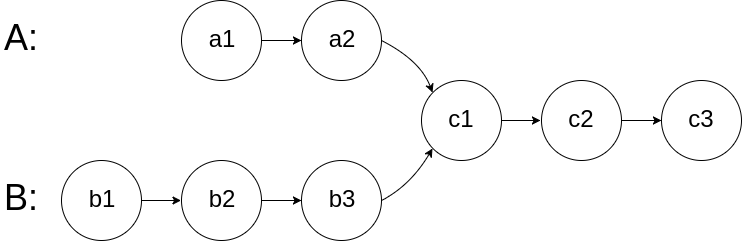
**Complexity Analysis**

* + Time complexity : O(n)*O*(*n*). There are total n*n* nodes and we visit each node once.
  + Space complexity : O(1)*O*(1). All we need is the four pointers.

* + **Intersection of Two Linked Lists**

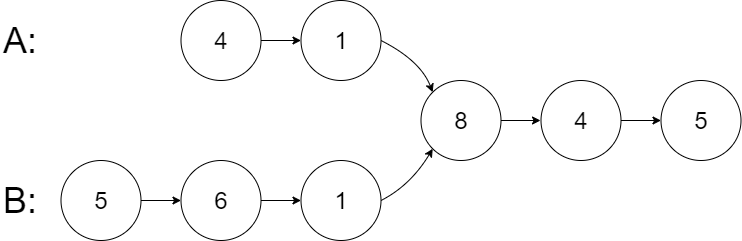
Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

[](https://assets.leetcode.com/uploads/2018/12/13/160_statement.png)

begin to intersect at node c1.

**Example 1:**

[](https://assets.leetcode.com/uploads/2020/06/29/160_example_1_1.png)

**Input:** intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3

**Output:** Reference of the node with value = 8

**Input Explanation:** The intersected node's value is 8 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

**Example 2:**

[Diagram

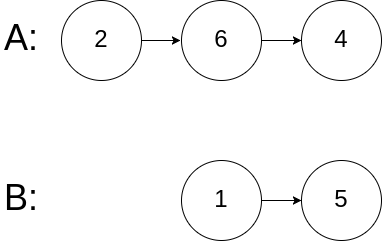
Description automatically generated](https://assets.leetcode.com/uploads/2020/06/29/160_example_2.png)

**Input:** intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

**Output:** Reference of the node with value = 2

**Input Explanation:** The intersected node's value is 2 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are 3 nodes before the intersected node in A; There are 1 node before the intersected node in B.

**Example 3:**

[](https://assets.leetcode.com/uploads/2018/12/13/160_example_3.png)

**Input:** intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

**Output:** null

**Input Explanation:** From the head of A, it reads as [2,6,4]. From the head of B, it reads as [1,5]. Since the two lists do not intersect, intersectVal must be 0, while skipA and skipB can be arbitrary values.

**Explanation:** The two lists do not intersect, so return null.

**Notes:**

* + If the two linked lists have no intersection at all, return null.
  + The linked lists must retain their original structure after the function returns.
  + You may assume there are no cycles anywhere in the entire linked structure.
  + Each value on each linked list is in the range [1, 10^9].
  + Your code should preferably run in O(n) time and use only O(1) memory.

**Solution:**

Approach 3: Two Pointers

* + Maintain two pointers pA*pA* and pB*pB* initialized at the head of A and B, respectively. Then let them both traverse through the lists, one node at a time.
  + When pA*pA* reaches the end of a list, then redirect it to the head of B (yes, B, that's right.); similarly when pB*pB* reaches the end of a list, redirect it the head of A.
  + If at any point pA*pA* meets pB*pB*, then pA*pA*/pB*pB* is the intersection node.
  + To see why the above trick would work, consider the following two lists: A = {1,3,5,7,9,11} and B = {2,4,9,11}, which are intersected at node '9'. Since B.length (=4) < A.length (=6), pB*pB* would reach the end of the merged list first, because pB*pB* traverses exactly 2 nodes less than pA*pA* does. By redirecting pB*pB* to head A, and pA*pA* to head B, we now ask pB*pB* to travel exactly 2 more nodes than pA*pA* would. So in the second iteration, they are guaranteed to reach the intersection node at the same time.
  + If two lists have intersection, then their last nodes must be the same one. So when pA*pA*/pB*pB* reaches the end of a list, record the last element of A/B respectively. If the two last elements are not the same one, then the two lists have no intersections.

**Complexity Analysis**

* + Time complexity : O(m+n)*O*(*m*+*n*).
  + Space complexity : O(1)*O*(1).

class Solution:

# @param two ListNodes

# @return the intersected ListNode

def getIntersectionNode(self, headA, headB):

if headA is None or headB is None:

return None

pa = headA # 2 pointers

pb = headB

while pa is not pb:

# if either pointer hits the end, switch head and continue the second traversal,

# if not hit the end, just move on to next

pa = headB if pa is None else pa.next

pb = headA if pb is None else pb.next

return pa # only 2 ways to get out of the loop, they meet or the both hit the end=None

# the idea is if you switch head, the possible difference between length would be countered.

# On the second traversal, they either hit or miss.

# if they meet, pa or pb would be the node we are looking for,

# if they didn't meet, they will hit the end at the same iteration, pa == pb == None, return either one of them is the same,None