

## **EXPERIMENT:-3**

NAME:Lokesh UID: 22BET10170

BRANCH: IT SECTION: 702(B)

**SEMESTER: 6<sup>th</sup> DOP: 12/02/2025** 

SUBJECT: AP II SUBJECT CODE: 22ITP-351

Aim: Detect a cycle in a linked list

**Objective:** 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.

#### Code:

```
class ListNode:
```

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

#### class Solution:

```
def hasCycle(self, head):
    slow = head
    fast = head
```



## while fast and fast.next:

slow = slow.next

fast = fast.next.next

if slow == fast:

return True

return False

```
• Case 1
• Case 2
• Case 3

Input

head =
[3,2,0,-4]

pos =
1

Output

true

Expected

true
```



• Case 1	• Case 2	• Case 3	
Input			
head = [1,2]			
pos =			
Output			
true			
Expected			
true			

• Case 1	• Case 2	• Case 3	
Input			
head = [1]			
pos = -1			
Output			
false			
Expected			
false			



Aim: Reverse linked list 2

**Objective:** Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return *the reversed list*.

#### Code:

```
class ListNode:
  def init (self, val=0, next=None):
    self.val = val
    self.next = next
class Solution:
  def reverseBetween(self, head, left, right):
    if not head or left == right:
       return head
    dummy = ListNode(0)
    dummy.next = head
    prev = dummy
    for in range(left - 1):
       prev = prev.next
    curr = prev.next
```

next node = None

```
for _ in range(right - left):
    next_node = curr.next
    curr.next = next_node.next
    next_node.next = prev.next
    prev.next = next_node
```

return dummy.next

```
• Case 1 • Case 2

Input

head =
[1,2,3,4,5]

left =
2

right =
4

Output

[1,4,3,2,5]

Expected

[1,4,3,2,5]
```



• Case 1	• Case 2	
Input		
head =		
left =		
right =		
Output		
[5]		
Expected		
[5]		

Aim: rotate a list

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

## **Code:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

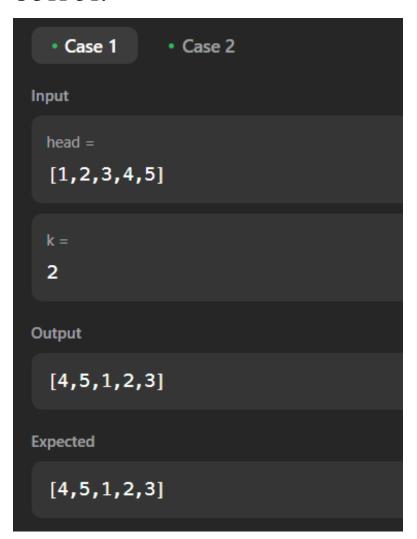
#### class Solution:

```
def rotateRight(self, head, k):
  if not head or not head.next or k == 0:
     return head
  # Compute the length of the list
  length = 1
  tail = head
  while tail.next:
     tail = tail.next
     length += 1
  # Make it a circular list
  tail.next = head
  # Find the new tail position
  k = k \% length
  steps_to_new_tail = length - k
  new tail = head
  for in range(steps to new tail - 1):
     new_tail = new_tail.next
  # Break the circle and set the new head
  new head = new tail.next
```



new\_tail.next = None

return new\_head





• Case 1	• Case 2	
Input		
head = [0,1,2]		
k = <b>4</b>		
Output		
[2,0,1]		
Expected		
[2,0,1]		

Aim: Merge k sorted lists

**Objective:** 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.

### Code:

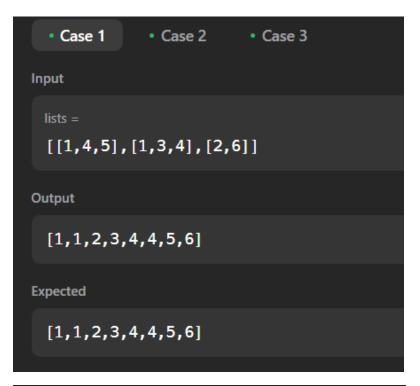
import heapq

class ListNode:

```
def init (self, val=0, next=None):
     self.val = val
    self.next = next
class Solution:
  def mergeKLists(self, lists):
    heap = []
     # Push the head nodes of all lists into the heap
     for i, node in enumerate(lists):
       if node:
          heapq.heappush(heap, (node.val, i, node))
     dummy = ListNode(0)
     curr = dummy
     while heap:
       val, i, node = heapq.heappop(heap)
       curr.next = node
       curr = curr.next
       if node.next:
          heapq.heappush(heap, (node.next.val, i, node.next))
```



# return dummy.next



• Case 2	• Case 3
	• Case 2



```
• Case 1 • Case 2 • Case 3

Input

Iists =
[[]]

Output

[]

Expected

[]
```

Aim: Sort List

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

### **CODE:**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def sortList(self, head):
        if not head or not head.next:
        return head
```

```
# Split the list into halves
  slow, fast = head, head.next
  while fast and fast.next:
     slow = slow.next
     fast = fast.next.next
  mid = slow.next
  slow.next = None
  # Recursively sort both halves
  left = self.sortList(head)
  right = self.sortList(mid)
  # Merge the sorted halves
  return self.merge(left, right)
def merge(self, 11, 12):
  dummy = ListNode(0)
  curr = dummy
  while 11 and 12:
     if 11.val < 12.val:
       curr.next = 11
       11 = 11.next
```



else:

curr.next = 12

12 = 12.next

curr = curr.next

curr.next = 11 if 11 else 12

return dummy.next

```
• Case 1 • Case 2 • Case 3

Input

head =
[4,2,1,3]

Output

[1,2,3,4]

Expected

[1,2,3,4]
```



• Case 1	• Case 2	• Case 3	
Input			
head = [-1,5,3,4	,0]		
Output			
[-1,0,3,4	,5]		
Expected			
[-1,0,3,4	,5]		

• Case 1	• Case 2	• Case 3
Input		
head =		
Output		
[]		
Expected		
П		

