



## EXPERIMENT:-3

**NAME: Tarun Sharma**

**UID: 22BET10016**

**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
```

## OUTPUT:

• 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 =  
0

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  $\text{left} \leq \text{right}$ , 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
```

## OUTPUT:

• 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 =  
[5]

left =  
1

right =  
1

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
```

## OUTPUT:

• Case 1

• Case 2

Input

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

k =  
2

Output

[4,5,1,2,3]

Expected

[4,5,1,2,3]



• Case 1

• Case 2

Input

head =  
[0,1,2]

k =  
4

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

## OUTPUT:

• Case 1

• Case 2

• Case 3

Input

lists =  
[[1,4,5],[1,3,4],[2,6]]

Output

[1,1,2,3,4,4,5,6]

Expected

[1,1,2,3,4,4,5,6]

• Case 1

• Case 2

• Case 3

Input

lists =  
[]

Output

[]

Expected

[]

• Case 1

• Case 2

• Case 3

Input

lists =  
[[]]

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, l1, l2):  
    dummy = ListNode(0)  
    curr = dummy  
  
    while l1 and l2:  
        if l1.val < l2.val:  
            curr.next = l1  
            l1 = l1.next
```

else:

curr.next = l2

l2 = l2.next

curr = curr.next

curr.next = l1 if l1 else l2

return dummy.next

## OUTPUT:

• 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

```
[]
```



**CHANDIGARH**  
**UNIVERSITY**

Discover. Learn. Empower.