

Tanish Mittal
IT-C
2000290130174

Day – 6 : Linked List -II

Problem 1: 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.

```
class ListNode:
```

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

```
    self.val = val
```

```
    self.next = next
```

```
def getIntersectionNode(headA, headB):
```

```
    def getLength(node):
```

```
        length = 0
```

```
        while node:
```

```
            length += 1
```

```
            node = node.next
```

```
        return length
```

```
lenA = getLength(headA)
```

```
lenB = getLength(headB)
```

```
ptrA = headA
```

```
ptrB = headB
```

```
diff = abs(lenA - lenB)
```

```
if lenA > lenB:
```

```
    for _ in range(diff):
```

```
        ptrA = ptrA.next
```

```
else:
```

```
    for _ in range(diff):
```

```
        ptrB = ptrB.next
```

```
while ptrA and ptrB:
    if ptrA == ptrB:
        return ptrA
    ptrA = ptrA.next
    ptrB = ptrB.next
```

```
return None
```

```
def createLinkedList(lst):
    if not lst:
        return None
    head = ListNode(lst[0])
    current = head
    for i in range(1, len(lst)):
        current.next = ListNode(lst[i])
        current = current.next
    return head
```

```
list1 = createLinkedList([1, 3, 1, 2, 4])
list2 = createLinkedList([3, 2, 4])
intersection_node = getIntersectionNode(list1, list2)
if intersection_node:
    print(intersection_node.val)
else:
    print("No intersection")
```

```
17 ptrA = headA
18
input
No intersection
...Program finished with exit code 0
Press ENTER to exit console.
```

Problem 2: 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.

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

class ListNode:

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

```
    self.val = val
```

```
    self.next = next
```

```
def hasCycle(head):
```

```
    if not head or not head.next:
```

```
        return False
```

```
    slow = head
```

```
    fast = head.next
```

```
    while slow != fast:
```

```
        if not fast or not fast.next:
```

```
            return False
```

```
        slow = slow.next
```

```
        fast = fast.next.next
```

```

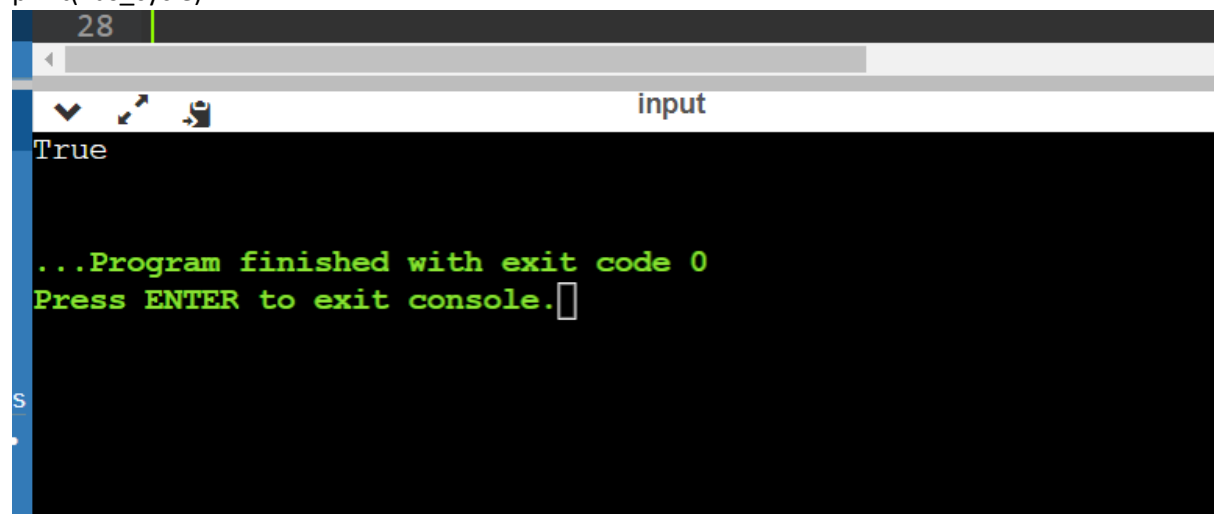
    return True

head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = head.next

```

```
has_cycle = hasCycle(head)
```

```
print(has_cycle)
```



```

28
True

...Program finished with exit code 0
Press ENTER to exit console.

```

Problem 3: Given the head of a linked list, reverse the nodes of the list k at a time, and return *the modified list*. k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

```

class ListNode:

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

        self.val = val

        self.next = next

def reverseKGroup(head, k):

    def reverse_group(start, end):

```

```
prev, curr = None, start
while curr != end:
    temp = curr.next
    curr.next = prev
    prev = curr
    curr = temp
return prev
```

```
def find_kth_node(node, k):
    for i in range(k):
        if not node:
            return None
        node = node.next
    return node
```

```
dummy = ListNode(0)
dummy.next = head
prev_group_tail = dummy
current = head
```

```
while current:
    group_end = find_kth_node(current, k)
    if not group_end:
        break
    next_group_head = group_end.next
    new_group_head = reverse_group(current, group_end)
    prev_group_tail.next = new_group_head
    current.next = next_group_head
    prev_group_tail = current
    current = next_group_head
```

```
return dummy.next
```

```
def create_linked_list(lst):  
    dummy = ListNode(0)  
    current = dummy  
    for val in lst:  
        current.next = ListNode(val)  
        current = current.next  
    return dummy.next
```

```
def linked_list_to_list(head):  
    lst = []  
    current = head  
    while current:  
        lst.append(current.val)  
        current = current.next  
    return lst
```

```
head = create_linked_list([1, 2, 3, 4, 5, 6, 7, 8])  
k = 3  
result = reverseKGroup(head, k)  
print(linked_list_to_list(result))
```



```
64
input
[3, 2, 1, 7, 6, 5]
...Program finished with exit code 0
Press ENTER to exit console.
```

Problem 4: Given the head of a singly linked list, return true if it is a palindrome.

```
class ListNode:
```

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

```
    self.val = val
```

```
    self.next = next
```

```
def isPalindrome(head):
```

```
    if head is None or head.next is None:
```

```
        return True
```

```
    slow = fast = head
```

```
    while fast.next and fast.next.next:
```

```
        slow = slow.next
```

```
        fast = fast.next.next
```

```
    second_half = reverse_linked_list(slow.next)
```

```
    slow.next = None
```

```
    first_half = head
```

```
while first_half and second_half:
    if first_half.val != second_half.val:
        return False
    first_half = first_half.next
    second_half = second_half.next

return True
```

```
def reverse_linked_list(head):
```

```
    prev = None
    curr = head
```

```
    while curr:
```

```
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node
```

```
    return prev
```

```
head = ListNode(1)
```

```
head.next = ListNode(2)
```

```
head.next.next = ListNode(3)
```

```
head.next.next.next = ListNode(3)
```

```
head.next.next.next.next = ListNode(2)
```

```
head.next.next.next.next.next = ListNode(1)
```

```
print(isPalindrome(head))
```



```
44 head.next.next = ListNode(3)
input
True
...Program finished with exit code 0
Press ENTER to exit console.
Js
•
```

Problem 5: Given the head of a [linked list](#), return *the node where the cycle begins*.
If there is no cycle, return null.

```
class ListNode:
```

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

```
    self.val = val
```

```
    self.next = next
```

```
def detectCycle(head):
```

```
    if not head or not head.next:
```

```
        return None
```

```
    slow = head
```

```
    fast = head
```

```
    while fast and fast.next:
```

```
        slow = slow.next
```

```
        fast = fast.next.next
```

```
        if slow == fast:
```

```
            break
```

```
    else:
```

```
        return None
```

```
fast = head
while slow != fast:
    slow = slow.next
    fast = fast.next
```

```
return slow
```

```
def createLinkedList(values):
```

```
    head = None
```

```
    current = None
```

```
    for val in values:
```

```
        node = ListNode(val)
```

```
        if not head:
```

```
            head = node
```

```
            current = head
```

```
        else:
```

```
            current.next = node
```

```
            current = node
```

```
    return head
```

```
values = [1, 2, 3, 4, 3, 6, 10]
```

```
head = createLinkedList(values)
```

```
result = detectCycle(head)
```

```
if result:
```

```
    print("tail connects to node index", values.index(result.val))
```

```
else:
```

```
print("No cycle found.")
```

A screenshot of a terminal window. The title bar at the top says 'input'. The terminal has a black background. The first line of text is 'No cycle found.' in white. The second line is '...Program finished with exit code 0' in green. The third line is 'Press ENTER to exit console.' in green, followed by a white cursor character.

Problem 6: Given a [Linked List](#) of size N, where every node represents a sub-linked-list and contains two pointers:

- (i) a next pointer to the next node,
- (ii) a bottom pointer to a linked list where this node is head.

class Node:

```
def __init__(self, data):
```

```
    self.data = data
```

```
    self.next = None
```

```
    self.bottom = None
```

```
def merge_lists(list1, list2):
```

```
    if not list1:
```

```
        return list2
```

```
    if not list2:
```

```
        return list1
```

```
    merged_list = None
```

```
    if list1.data < list2.data:
```

```
        merged_list = list1
```

```

        merged_list.bottom = merge_lists(list1.bottom, list2)
    else:
        merged_list = list2
        merged_list.bottom = merge_lists(list1, list2.bottom)

    merged_list.next = None
    return merged_list

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

    # Merge the first two lists
    head.next = merge_lists(head, head.next)

    # Flatten the remaining lists
    return flatten_linked_list(head.next)

def create_linked_list(arr, size_arr):
    head = None
    curr_node = None

    list_index = 0
    node_index = 0

    while list_index < len(size_arr):
        for _ in range(size_arr[list_index]):
            new_node = Node(arr[node_index])

            if not head:
                head = new_node

```

```
        curr_node = new_node
    else:
        curr_node.bottom = new_node
        curr_node = new_node
```

```
    node_index += 1
```

```
    list_index += 1
```

```
    return head
```

```
def print_linked_list(head):
```

```
    while head:
```

```
        print(head.data, end=" ")
```

```
        head = head.bottom
```

```
    print()
```

```
if __name__ == "__main__":
```

```
    arr = [5, 7, 8, 30, 10, 20, 19, 22, 50, 28, 35, 40, 45]
```

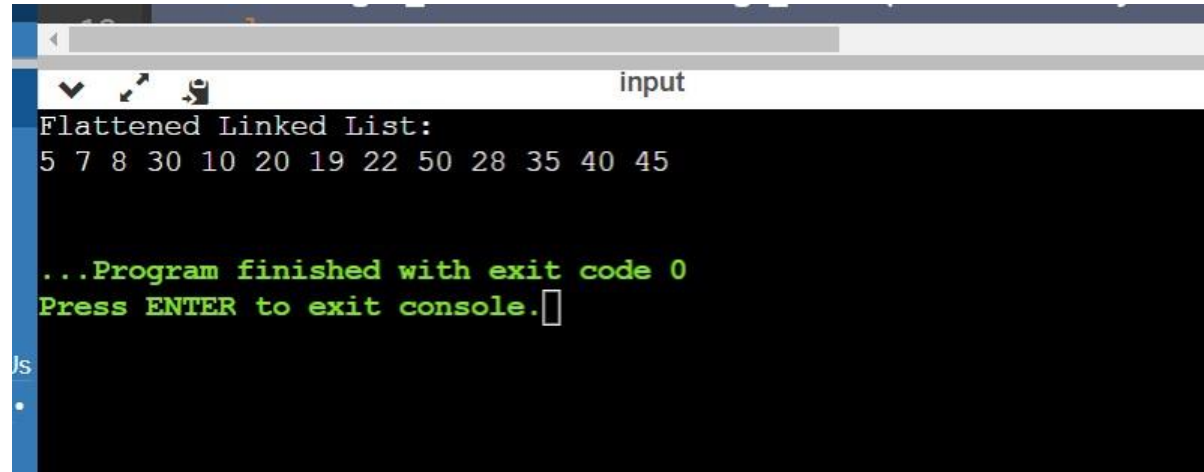
```
    size_arr = [4, 2, 3, 4]
```

```
    head = create_linked_list(arr, size_arr)
```

```
    print("Flattened Linked List:")
```

```
    flattened_head = flatten_linked_list(head)
```

```
print_linked_list(flattened_head)
```



```
input
Flattened Linked List:
5 7 8 30 10 20 19 22 50 28 35 40 45

...Program finished with exit code 0
Press ENTER to exit console.█
```