Qno.1)

# Definition for a singly-linked list node

class ListNode:

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

self.val = val

self.next = next

def srt\_LL(ls: str) -> str:

# Parse the input string to create the linked list

nodes = ls.split("->")

head = ListNode("Head")

curr = head

for node in nodes[1:]:

curr.next = ListNode(node)

curr = curr.next

# Perform selection sort on the linked list

curr = head.next

while curr:

min\_node = curr

runner = curr.next

while runner:

if runner.val.isalpha() and runner.val.lower() < min\_node.val.lower():

min\_node = runner

runner = runner.next

if min\_node != curr:

curr.val, min\_node.val = min\_node.val, curr.val

curr = curr.next

# Convert the sorted linked list back to a string and return it

res = "Head"

curr = head.next

while curr:

res += f"->{curr.val}"

curr = curr.next

res += "->NULL"

return res

srt\_LL("Head->D->A->C->A->G->NULL")

Qno2)

class ListNode:

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

self.val = val

self.next = next

def LL\_add(u, v):

# Traverse linked list u and extract its values to form number1

number1 = 0

while u:

number1 = number1 \* 10 + u.val

u = u.next

# Traverse linked list v and extract its values to form number2

number2 = 0

while v:

number2 = number2 \* 10 + v.val

v = v.next

# Add the two numbers

result = number1 + number2

# Create a new linked list to store the result

dummy = ListNode(0)

curr = dummy

# Convert the result to a linked list

for digit in str(result):

curr.next = ListNode(int(digit))

curr = curr.next

return dummy.next

# Create linked list u

u = ListNode(1)

u.next = ListNode(2)

u.next.next = ListNode(3)

u.next.next.next = ListNode(4)

# Create linked list v

v = ListNode(5)

v.next = ListNode(6)

v.next.next = ListNode(7)

v.next.next.next = ListNode(8)

# Call the LL\_add function

result = LL\_add(u, v)

# Print the result

while result:

print(result.val, end="->")

result = result.next

Qno.3)

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

def find\_monkey\_king(num\_monkeys, m\_value):

# Create a circular linked list of monkeys

head = Node(0)

curr = head

for i in range(1, num\_monkeys):

curr.next = Node(i)

curr = curr.next

curr.next = head

# Traverse the linked list, removing every m-th monkey

prev = curr

curr = head

count = 1

while curr.next != curr:

if count == m\_value:

prev.next = curr.next

count = 1

else:

count += 1

prev = curr

curr = curr.next

# Return the last remaining monkey's index (count starts at 1)

return curr.value

# Take user input

num\_monkeys = int(input("Enter total number of monkeys in a group: "))

m\_value = int(input("Enter m value: "))

# Call the find\_monkey\_king function and print the result

king\_index = find\_monkey\_king(num\_monkeys, m\_value)

print("The king will be", king\_index)

Qno.4)

class ListNode:

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

self.val = val

self.next = next

def commElem(l, m):

# Create a new linked list to store common characters

dummy = ListNode(0)

curr = dummy

# Traverse both linked lists and find common characters

l\_ptr = l

while l\_ptr:

m\_ptr = m

while m\_ptr:

if l\_ptr.val == m\_ptr.val:

curr.next = ListNode(l\_ptr.val)

curr = curr.next

break

m\_ptr = m\_ptr.next

l\_ptr = l\_ptr.next

return dummy.next

# Create linked list l

l = ListNode('G')

l.next = ListNode('O')

l.next.next = ListNode('O')

l.next.next.next = ListNode('D')

# Create linked list m

m = ListNode('G')

m.next = ListNode('o')

m.next.next = ListNode('O')

m.next.next.next = ListNode('g')

m.next.next.next.next = ListNode('L')

m.next.next.next.next.next = ListNode('E')

# Call the commElem function

result = commElem(l, m)

# Print the result

while result:

print(result.val, end="->")

result = result.next

Qno.5)

class ListNode:

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

self.val = val

self.next = next

def diffElem\_LL(l, m):

# Create a new linked list to store non-common characters

dummy = ListNode(0)

curr = dummy

# Traverse linked list l and find non-common characters

l\_ptr = l

while l\_ptr:

is\_common = False

m\_ptr = m

while m\_ptr:

if l\_ptr.val == m\_ptr.val:

is\_common = True

break

m\_ptr = m\_ptr.next

if not is\_common:

curr.next = ListNode(l\_ptr.val)

curr = curr.next

l\_ptr = l\_ptr.next

# Traverse linked list m and find non-common characters

m\_ptr = m

while m\_ptr:

is\_common = False

l\_ptr = l

while l\_ptr:

if m\_ptr.val == l\_ptr.val:

is\_common = True

break

l\_ptr = l\_ptr.next

if not is\_common:

curr.next = ListNode(m\_ptr.val)

curr = curr.next

m\_ptr = m\_ptr.next

return dummy.next

# Create linked list l

l = ListNode('G')

l.next = ListNode('O')

l.next.next = ListNode('O')

l.next.next.next = ListNode('D')

# Create linked list m

m = ListNode('G')

m.next = ListNode('o')

m.next.next = ListNode('O')

m.next.next.next = ListNode('g')

m.next.next.next.next = ListNode('L')

m.next.next.next.next.next = ListNode('E')

# Call the diffElem\_LL function

result = diffElem\_LL(l, m)

# Print the result

while result:

print(result.val, end="->")

result = result.next

Qno.6)

class ListNode:

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

self.val = val

self.next = next

def is\_loopLL(head):

# Traverse the linked list and mark each node as visited

curr = head

while curr:

if curr.val == "visited":

# The node has already been visited, so the linked list contains a loop

return True

curr.val = "visited"

curr = curr.next

# The linked list does not contain a loop

return False

# Create a linked list with a loop

a = ListNode(1)

a.next = ListNode(2)

a.next.next = ListNode(3)

a.next.next.next = ListNode(4)

a.next.next.next.next = ListNode(5)

a.next.next.next.next.next = ListNode(6)

a.next.next.next.next.next.next = a.next.next

# Call the is\_loopLL function

result = is\_loopLL(a)

# Print the result

print(result) # Output: True

# Create a linked list without a loop

b = ListNode(1)

b.next = ListNode(2)

b.next.next = ListNode(3)

b.next.next.next = ListNode(4)

b.next.next.next.next = ListNode(5)

b.next.next.next.next.next = ListNode(6)

# Call the is\_loopLL function

result = is\_loopLL(b)

# Print the result

print(result) # Output: False

Qno.7)

class ListNode:

def \_\_init\_\_(self, coeff=0, exp=0, next=None):

self.coeff = coeff

self.exp = exp

self.next = next

def conv\_LL(l, m):

# Initialize a new linked list to store the resulting polynomial

dummy = ListNode()

curr = dummy

# Traverse the linked list l and multiply each term with each term in m

l\_ptr = l

while l\_ptr:

m\_ptr = m

while m\_ptr:

# Multiply the terms and add them to the resulting polynomial

coeff = l\_ptr.coeff \* m\_ptr.coeff

exp = l\_ptr.exp + m\_ptr.exp

curr.next = ListNode(coeff, exp)

curr = curr.next

m\_ptr = m\_ptr.next

l\_ptr = l\_ptr.next

# Combine like terms in the resulting polynomial

prev = dummy

curr = dummy.next

while curr:

next\_node = curr.next

if next\_node and curr.exp == next\_node.exp:

curr.coeff += next\_node.coeff

prev.next = next\_node.next

else:

prev = curr

curr = next\_node

return dummy.next

# Create linked list l

l = ListNode(1, 0)

l.next = ListNode(2, 2)

l.next.next = ListNode(3, 3)

l.next.next.next = ListNode(4, 5)

# Create linked list m

m = ListNode(5, 0)

m.next = ListNode(6, 1)

m.next.next = ListNode(7, 3)

m.next.next.next = ListNode(8, 4)

# Call the conv\_LL function

result = conv\_LL(l, m)

# Print the result

while result:

print(f"{result.coeff}x^{result.exp}", end=" + ")

result = result.next