Qno.1)

class Node:

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

self.label = label

self.next = None

class CircularLinkedList:

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

self.head = None

# function to create the circular linked list with 10 nodes labeled from 1 to 10

def create(self):

for i in range(1, 11):

new\_node = Node(i)

if self.head is None:

self.head = new\_node

new\_node.next = self.head

else:

temp = self.head

while temp.next != self.head:

temp = temp.next

temp.next = new\_node

new\_node.next = self.head

# function to traverse the circular linked list and determine which nests are safe to sleep in

def traverse(self, n):

# create a list of labels to mark the nests that the wolf has checked

checked\_nests = [1]

skipped = 1

current\_node = self.head.next

# loop through the circular linked list n times

for i in range(n):

# skip the number of nests based on the wolf's checking pattern

for j in range(skipped):

current\_node = current\_node.next

# add the label of the nest that the wolf checks to the checked\_nests list

checked\_nests.append(current\_node.label)

# set the current node to the next nest in the circular linked list

current\_node = current\_node.next

# increment the skipped number based on the wolf's checking pattern

skipped += 1

if skipped > 3:

skipped = 1

# print the safe nests by subtracting the checked\_nests from the set of all nests labeled from 1 to 10

safe\_nests = set(range(1, 11)) - set(checked\_nests)

print("Safe nests to sleep in:", sorted(list(safe\_nests)))

# create the circular linked list

cll = CircularLinkedList()

cll.create()

# get input from the user for the number of times the wolf checks the nests

n = int(input("Enter the number of times the wolf checks the nests: "))

# traverse the circular linked list and determine which nests are safe to sleep in

cll.traverse(n)

Qno.2)

class Node:

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

self.value = value

self.next = None

class CircularLinkedList:

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

self.head = None

# function to create the circular linked list with nodes containing char values in non-descending sequence

def create(self, values):

for value in values:

new\_node = Node(value)

if self.head is None:

self.head = new\_node

new\_node.next = self.head

else:

temp = self.head

while temp.next != self.head and ord(temp.next.value) < ord(value):

temp = temp.next

new\_node.next = temp.next

temp.next = new\_node

# function to extract common node values from two circular linked lists and generate a new circular linked list without duplicates

@staticmethod

def extract\_common\_nodes(l, m):

# create the new circular linked list

new\_cll = CircularLinkedList()

# traverse the l and m linked lists and add common node values to the new circular linked list

l\_node = l.head

m\_node = m.head

while True:

if l\_node.value == m\_node.value:

new\_cll.create([l\_node.value])

l\_node = l\_node.next

m\_node = m\_node.next

elif ord(l\_node.value) < ord(m\_node.value):

l\_node = l\_node.next

else:

m\_node = m\_node.next

# check for circular linked list end condition

if l\_node == l.head or m\_node == m.head:

break

return new\_cll

# function to print the circular linked list

def print(self):

temp = self.head

while True:

print(temp.value, end=' ')

temp = temp.next

if temp == self.head:

break

print()

# create the circular linked lists l and m

l\_values = ['a', 'c', 'd', 'g']

m\_values = ['a', 'b', 'd', 'e', 'g']

l\_cll = CircularLinkedList()

l\_cll.create(l\_values)

m\_cll = CircularLinkedList()

m\_cll.create(m\_values)

# extract common node values from l and m and generate a new circular linked list without duplicates

new\_cll = CircularLinkedList.extract\_common\_nodes(l\_cll, m\_cll)

# print the new circular linked list

new\_cll.print()

Qno.3)

class Node:

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

self.data = data

self.next = None

def is\_prime(n):

if n <= 1:

return False

for i in range(2, int(n\*\*0.5) + 1):

if n % i == 0:

return False

return True

def delete\_prime\_CLL(head):

current = head

while current.next != head:

if is\_prime(current.next.data):

current.next = current.next.next

else:

current = current.next

if is\_prime(head.data):

current.next = head.next

head = head.next

return head

# create the linked list

head = Node(13)

head.next = Node(12)

head.next.next = Node(15)

head.next.next.next = Node(14)

head.next.next.next.next = head

# delete all prime number nodes

head = delete\_prime\_CLL(head)

# print the linked list

current = head

print(current.data, end=' ')

current = current.next

while current != head:

print(current.data, end=' ')

current = current.next

Qno.4)

class Node:

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

self.value = value

self.next = None

class CircularLinkedList:

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

self.head = None

def append(self, value):

node = Node(value)

if self.head is None:

self.head = node

self.head.next = self.head

else:

current = self.head

while current.next != self.head:

current = current.next

current.next = node

node.next = self.head

def split\_CLL(org, N):

if org is None:

return None

if org.next == org:

return org, None

count = 1

current = org

while count < N and current.next != org:

current = current.next

count += 1

if count == N:

new\_head = current.next

current.next = org

org1 = org

org2 = new\_head

while org2.next != org:

org2 = org2.next

org2.next = new\_head

return org1, org2

else:

return org, None

# Example usage

org = CircularLinkedList()

org.append(2)

org.append(3)

org.append(4)

org.append(5)

org.append(6)

org.append(7)

org.append(8)

org.head = org.head.next

org1, org2 = split\_CLL(org.head, 3)

print("org1: ", end="")

current = org1

while current.next != org1:

print(current.value, end="->")

current = current.next

print(current.value)

print("org2: ", end="")

current = org2

while current.next != org2:

print(current.value, end="->")

current = current.next

print(current.value)

Qno.5)

class Node:

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

self.value = value

self.next = None

self.prev = None

class DoublyLinkedList:

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

self.head = None

def append(self, value):

node = Node(value)

if self.head is None:

self.head = node

else:

current = self.head

while current.next is not None:

current = current.next

current.next = node

node.prev = current

def rotate\_DLL(dll, N):

if dll is None:

return None

current = dll.head

count = 1

while count < N and current is not None:

current = current.next

count += 1

if current is None:

return dll

new\_head = current

while current.next is not None:

current = current.next

current.next = dll.head

dll.head.prev = current

dll.head = new\_head.next

dll.head.prev = None

new\_head.next = None

return dll

# Example usage

dll = DoublyLinkedList()

dll.append('c')

dll.append('i')

dll.append('v')

dll.append('i')

dll.append('c')

dll.head.prev = Node('Head')

Node('Head').next = dll.head

dll.head = dll.head.prev

dll = rotate\_DLL(dll, 3)

current = dll.head

while current is not None:

print(current.value, end="<=>")

current = current.next

print("None")

Qno.6)

class Node:

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

self.value = value

self.next = None

self.prev = None

self.random = None

class DoublyLinkedList:

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

self.head = None

def append(self, value):

node = Node(value)

if self.head is None:

self.head = node

else:

current = self.head

while current.next is not None:

current = current.next

current.next = node

node.prev = current

def has\_random\_pointer(dll):

current = dll.head

while current is not None:

if current.random is not None:

return True

current = current.next

return False

def correct\_random\_pointer(dll):

if dll is None or dll.head is None:

return dll

node\_map = {}

current = dll.head

while current is not None:

node\_map[current] = Node(current.value)

current = current.next

current = dll.head

new\_dll = DoublyLinkedList()

while current is not None:

new\_node = node\_map[current]

new\_node.next = node\_map.get(current.next)

new\_node.prev = node\_map.get(current.prev)

new\_node.random = node\_map.get(current.random)

new\_dll.append(new\_node)

current = current.next

return new\_dll

# Example usage

dll = DoublyLinkedList()

dll.append('c')

dll.append('i')

dll.append('v')

dll.append('i')

dll.append('c')

dll.head.random = dll.head.next.next

dll.head.next.random = dll.head.next.next.next

dll.head.next.next.random = dll.head

dll.head.next.next.next.random = dll.head.next

dll.head.next.next.next.next.random = dll.head

print(has\_random\_pointer(dll)) # True

new\_dll = correct\_random\_pointer(dll)

print(has\_random\_pointer(new\_dll)) # False