## Topics

1. Implement Node Class

class Node:

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

self.data = data

self.next = None

1. Implement CircularlyLinkedList Class

class CircularlyLinkedList:

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

self.head = None

self.tail = None

def is\_empty(self):

return self.head is None

def add\_to\_head(self, data):

new\_node = Node(data)

if self.is\_empty():

self.head = new\_node

self.tail = new\_node

new\_node.next = new\_node # Make the new node point to itself

else:

new\_node.next = self.head

self.tail.next = new\_node

self.head = new\_node

def add\_to\_tail(self, data):

new\_node = Node(data)

if self.is\_empty():

self.head = new\_node

self.tail = new\_node

new\_node.next = new\_node # Make the new node point to itself

else:

new\_node.next = self.head

self.tail.next = new\_node

self.tail = new\_node

def remove\_from\_head(self):

if self.is\_empty():

return None

data = self.head.data

if self.head == self.tail:

self.head = None

self.tail = None

else:

self.head = self.head.next

self.tail.next = self.head

return data

def remove\_from\_tail(self):

if self.is\_empty():

return None

data = self.tail.data

if self.head == self.tail:

self.head = None

self.tail = None

else:

current = self.head

while current.next != self.tail:

current = current.next

current.next = self.head

self.tail = current

return data

def display(self):

if self.is\_empty():

print("Circularly linked list is empty.")

else:

current = self.head

print("Circularly linked list:")

while current.next != self.head:

print(current.data, end=" ")

current = current.next

print(current.data) # Print the last node's data

# Example usage

clist = CircularlyLinkedList()

clist.add\_to\_head(3)

clist.add\_to\_head(2)

clist.add\_to\_head(1)

clist.display() # Output: Circularly linked list: 1 2 3

clist.add\_to\_tail(4)

clist.add\_to\_tail(5)

clist.display() # Output: Circularly linked list: 1 2 3 4 5

clist.remove\_from\_head()

clist.remove\_from\_tail()

clist.display() # Output: Circularly linked list: 2 3 4

1. Implement Basic Methods of CircularlyLinkedList

* isEmpty()
* size()
* first()
* last()
* addFirst()
* addLast()
* removeFirst()
* rotate()

class CircularlyLinkedList:

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

self.head = None

self.tail = None

self.size = 0

def is\_empty(self):

return self.size == 0

def get\_size(self):

return self.size

def first(self):

if self.is\_empty():

return None

return self.head.data

def last(self):

if self.is\_empty():

return None

return self.tail.data

def add\_first(self, data):

new\_node = Node(data)

if self.is\_empty():

self.head = new\_node

self.tail = new\_node

new\_node.next = new\_node

else:

new\_node.next = self.head

self.tail.next = new\_node

self.head = new\_node

self.size += 1

def add\_last(self, data):

new\_node = Node(data)

if self.is\_empty():

self.head = new\_node

self.tail = new\_node

new\_node.next = new\_node

else:

new\_node.next = self.head

self.tail.next = new\_node

self.tail = new\_node

self.size += 1

def remove\_first(self):

if self.is\_empty():

return None

data = self.head.data

if self.head == self.tail:

self.head = None

self.tail = None

else:

self.head = self.head.next

self.tail.next = self.head

self.size -= 1

return data

def rotate(self):

if not self.is\_empty():

self.head = self.head.next

self.tail = self.tail.next

def display(self):

if self.is\_empty():

print("Circularly linked list is empty.")

else:

current = self.head

print("Circularly linked list:")

while current.next != self.head:

print(current.data, end=" ")

current = current.next

print(current.data)

## Homework

1. Consider the implementation of CircularlyLinkedList.addFirst, in Code Fragment 3.16. The else body at lines 39 and 40 of that method relies on a locally declared variable, newest. Redesign that clause to avoid use of any local variable.

class CircularlyLinkedList:

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

self.head = None

self.tail = None

self.size = 0

def is\_empty(self):

return self.size == 0

def add\_first(self, data):

new\_node = Node(data)

if self.is\_empty():

new\_node.next = new\_node

self.head = new\_node

self.tail = new\_node

else:

new\_node.next = self.head

self.tail.next = new\_node

self.head = new\_node

self.size += 1

1. Give an implementation of the size( ) method for the CircularlyLinkedList class, assuming that we did not maintain size as an instance variable.

class CircularlyLinkedList:

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

self.head = None

self.tail = None

def is\_empty(self):

return self.head is None

def size(self):

count = 0

if self.is\_empty():

return count

current = self.head

while True:

count += 1

current = current.next

if current == self.head:

break

return count

1. Implement the equals( ) method for the CircularlyLinkedList class, assuming that two lists are equal if they have the same sequence of elements, with corresponding elements currently at the front of the list.

class CircularlyLinkedList:

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

self.head = None

self.tail = None

def is\_empty(self):

return self.head is None

def equals(self, other\_list):

if self.size() != other\_list.size():

return False

if self.is\_empty() and other\_list.is\_empty():

return True

current\_self = self.head

current\_other = other\_list.head

while current\_self.data == current\_other.data:

current\_self = current\_self.next

current\_other = current\_other.next

if current\_self == self.head:

return True

return False

1. Suppose you are given two circularly linked lists, L and M. Describe an algorithm for telling if L and M store the same sequence of elements (but perhaps with different starting points).

class Node:

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

self.data = data

self.next = None

class CircularlyLinkedList:

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

self.head = None

self.tail = None

def is\_empty(self):

return self.head is None

def equals(self, other\_list):

if self.is\_empty() and other\_list.is\_empty():

return True

current\_L = self.head

current\_M = other\_list.head

while True:

if current\_L.data != current\_M.data:

return False

current\_L = current\_L.next

current\_M = current\_M.next

if current\_L == self.head or current\_M == other\_list.head:

break

return True

1. Given a circularly linked list L containing an even number of nodes, describe how to split L into two circularly linked lists of half the size.

class Node:

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

self.data = data

self.next = None

class CircularlyLinkedList:

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

self.head = None

self.tail = None

def is\_empty(self):

return self.head is None

def split\_in\_half(self):

if self.is\_empty() or self.head.next == self.head:

return self

slow = self.head

fast = self.head

while fast.next != self.head and fast.next.next != self.head:

slow = slow.next

fast = fast.next.next

L2 = CircularlyLinkedList()

L2.head = slow.next

slow.next = self.head

L2.tail = self.tail

self.tail = slow

L2.tail.next = L2.head

self.tail.next = self.head

return L2

1. Implement the clone( ) method for the CircularlyLinkedList class.

class Node:

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

self.data = data

self.next = None

class CircularlyLinkedList:

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

self.head = None

self.tail = None

def is\_empty(self):

return self.head is None

def clone(self):

if self.is\_empty():

return CircularlyLinkedList()

cloned\_list = CircularlyLinkedList()

current = self.head

while True:

cloned\_node = Node(current.data)

if cloned\_list.is\_empty():

cloned\_list.head = cloned\_node

cloned\_list.tail = cloned\_node

cloned\_node.next = cloned\_node

else:

cloned\_node.next = cloned\_list.head

cloned\_list.tail.next = cloned\_node

cloned\_list.tail = cloned\_node

current = current.next

if current == self.head:

break

return cloned\_list