**Assignment**

1\

class ArrayQueue:  
 def \_\_init\_\_(self):  
 self.\_data = []  
 self.\_front = 0  
  
 def is\_empty(self):  
 return len(self.\_data) == 0  
  
 def \_\_len\_\_(self):  
 return len(self.\_data)  
  
 def enqueue(self, item):  
 self.\_data.append(item)  
  
 def dequeue(self):  
 if self.is\_empty():  
 raise IndexError("Queue is empty")  
 item = self.\_data[self.\_front]  
 self.\_front += 1  
 return item  
  
 def rotate(self):  
 if len(self.\_data) > 1:  
 self.\_front = (self.\_front + 1) % len(self.\_data)

2\

class ArrayQueue:  
 def \_\_init\_\_(self):  
 self.\_data = []  
 self.\_front = 0  
  
 def is\_empty(self):  
 return len(self.\_data) == 0  
  
 def \_\_len\_\_(self):  
 return len(self.\_data)  
  
 def enqueue(self, item):  
 self.\_data.append(item)  
  
 def dequeue(self):  
 if self.is\_empty():  
 raise IndexError("Queue is empty")  
 item = self.\_data[self.\_front]  
 self.\_front += 1  
 return item  
  
 def clone(self):  
 cloned\_queue = ArrayQueue()  
 cloned\_queue.\_data = self.\_data[self.\_front:] # Create a copy of the underlying list  
 cloned\_queue.\_front = 0 # Reset the front index of the cloned queue  
 return cloned\_queue

3\

public class Node:  
 def \_\_init\_\_(self, data=None, next=None):  
 self.data = data  
 self.next = next  
  
  
 class LinkedQueue:  
 def \_\_init\_\_(self):  
 self.\_head = None  
 self.\_tail = None  
 self.\_size = 0  
  
 def is\_empty(self):  
 return self.\_size == 0  
  
 def \_\_len\_\_(self):  
 return self.\_size  
  
 def enqueue(self, item):  
 new\_node = Node(item)  
  
 if self.is\_empty():  
 self.\_head = new\_node  
 else:  
 self.\_tail.next = new\_node  
  
 self.\_tail = new\_node  
 self.\_size += 1  
  
 def dequeue(self):  
 if self.is\_empty():  
 raise IndexError("Queue is empty")  
  
 item = self.\_head.data  
 self.\_head = self.\_head.next  
 self.\_size -= 1  
  
 if self.is\_empty():  
 self.\_tail = None  
  
 return item  
  
 def concatenate(self, Q2):  
 if Q2.is\_empty():  
 return  
  
 if self.is\_empty():  
 self.\_head = Q2.\_head  
 else:  
 self.\_tail.next = Q2.\_head  
  
 self.\_tail = Q2.\_tail  
 self.\_size += len(Q2)

4\

from collections import deque  
  
def josephus(n, k):  
# Create a queue with numbers 1 to n  
queue = deque(range(1, n + 1))  
  
# Iterate until the queue has only one person left  
while len(queue) > 1:  
# Eliminate k-1 people by dequeuing and enqueuing  
for \_ in range(k - 1):  
queue.append(queue.popleft())  
  
# Eliminate the kth person by dequeuing  
queue.popleft()  
  
# Return the last remaining person  
return queue[0]  
  
# Example usage  
n = 7 # Number of people in the circle  
k = 3 # Every 3rd person is eliminated  
last\_person = josephus(n, k)  
print("The last person remaining is:", last\_person)

5\

from collections import deque  
  
class Process:  
 def \_\_init\_\_(self, name, burst\_time):  
 self.name = name  
 self.burst\_time = burst\_time  
  
 def round\_robin\_scheduling(processes, time\_quantum):  
 queue = deque(processes)  
 completed\_processes = []  
  
 while queue:  
 current\_process = queue.popleft()  
  
 if current\_process.burst\_time > time\_quantum:  
 current\_process.burst\_time -= time\_quantum  
 queue.append(current\_process)  
 else:  
 completed\_processes.append(current\_process)  
  
 return completed\_processes