Algorithms and Data Structures

CS106.3

1. **What is algorithm of circular queue.**

- Create an array of fixed size to hold the elements of the queue.
- Set two pointers, front and rear, to -1, indicating an empty queue.
- If (rear + 1) % size == front, the queue is full.
- If true, display an overflow error or return an appropriate value. otherwise
- If front is -1, set front to 0.
- Increment rear by 1 (modulo size to wrap around).
- Insert the new element at index rear in the array.
- If front is -1, the queue is empty.
- If true, display an underflow error or return an appropriate value. Otherwise:
- Store the value at index front as the element to be removed.
- If front and rear are equal, set front and rear back to -1, indicating an empty queue.
- Otherwise, increment front by 1 (modulo size).
- Check for an empty queue:
- If front is -1, the queue is empty.
- Return true to indicate an empty queue; otherwise, return false.
- Check for a full queue:
- If (rear + 1) % size == front, the queue is full.
- Return true to indicate a full queue; otherwise, return false.

2. **Write a simple program of circular.**

```
// Check if the queue is full
int isFull() {
  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;
  return 0;
}
// Check if the queue is empty
int isEmpty() {
  if (front == -1) return 1;
  return 0;
}
// Adding an element
void enQueue(int element) {
  if (isFull())
    printf("\n Queue is full!! \n");
  else {
    if (front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    items[rear] = element;
    printf("\n Inserted -> %d", element);
  }
}
```

```
// Removing an element
int deQueue() {
  int element;
  if (isEmpty()) {
    printf("\n Queue is empty !! \n");
    return (-1);
  } else {
    element = items[front];
    if (front == rear) {
      front = -1;
      rear = -1;
    }
    // Q has only one element, so we reset the
    // queue after dequeing it. ?
    else {
      front = (front + 1) % SIZE;
    }
    printf("\n Deleted element -> %d \n", element);
    return (element);
  }
}
// Display the queue
void display() {
  int i;
  if (isEmpty())
```

```
    printf(" \n Empty Queue\n");
  else {
    printf("\n Front -> %d ", front);
    printf("\n Items -> ");
    for (i = front; i != rear; i = (i + 1) % SIZE) {
      printf("%d ", items[i]);
    }
    printf("%d ", items[i]);
    printf("\n Rear -> %d \n", rear);
  }
}

int main() {
  // Fails because front = -1
  deQueue();

  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);

  // Fails to enqueue because front == 0 && rear == SIZE - 1
  enQueue(6);
  display();
  deQueue();
  display();
  enQueue(7);
  display();
  // Fails to enqueue because front == rear + 1
  enQueue(8);
  return 0;
}
```

3. Compare and contrast linear queue and circular queue.

Linear queue	**Circular queue**
• Arranges the data in a linear pattern.	• Arranges the data in a circular order where the rear end is connected with the front end.
• The insertion and deletion operations are fixed	• Insertion and deletion are not fixed, and it can be done in any position.
• Linear queue requires more memory space.	• It requires less memory space.
• In the case of a linear queue, the element added in the first position is going to be deleted in the first position. The order of operations performed on any element is fixed (FIFO).	• In the case of circular queue, the order of operations performed on an element may change.
• In a linear queue, we can easily fetch out the peek value.	• In a circular queue, we cannot fetch out the peek value easily.

- Not suitable for real-time systems where overflow can lead to data loss.	Suitable for real-time systems where continuous data insertion is required.