Data Structures and Algorithms

Lecture 4: Queues

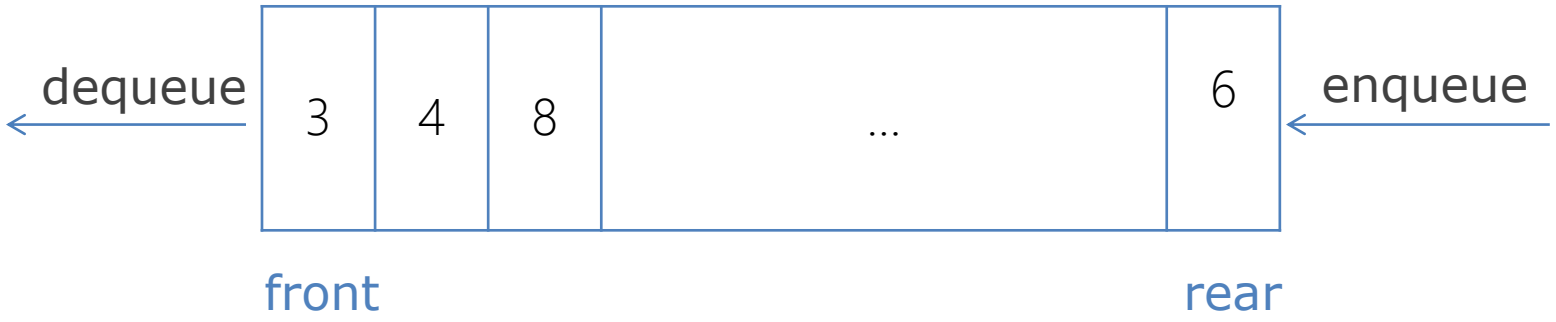
Department of Computer Science & Technology
United International College

Outline

- Queue ADT
- Basic operations of queue
  - enqueue, dequeue
- Applications of queue
- Implementation of queue
  - Array
  - Linked list

Queue ADT

- Like a stack, a *queue* is also a list.
- However, with a queue
  - insertion is done at one end
    - The rear
  - while deletion is performed at the other end.
    - The front



Queue Animation

- http://www.cs.armstrong.edu/liang/animation/web/Queue.html
- Queues are known as FIFO (First In, First Out) lists.

Enqueue and Dequeue

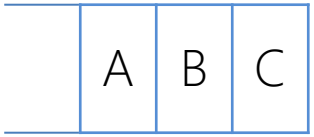
- Primary operations: Enqueue and Dequeue
- Enqueue
  - insert an element at the rear of the queue
- Dequeue
  - remove an element from the front of the queue

Enqueue and Dequeue

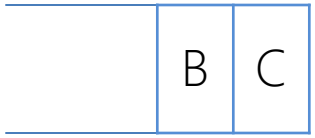
Empty queue

Enqueue(A)

Enqueue(B)



Enqueue(C)



Dequeue()

Dequeue()

Queue Applications

- Printer Queue
- Web Crawler
- System Buffer
- Any sequence maintained in a FIFO order

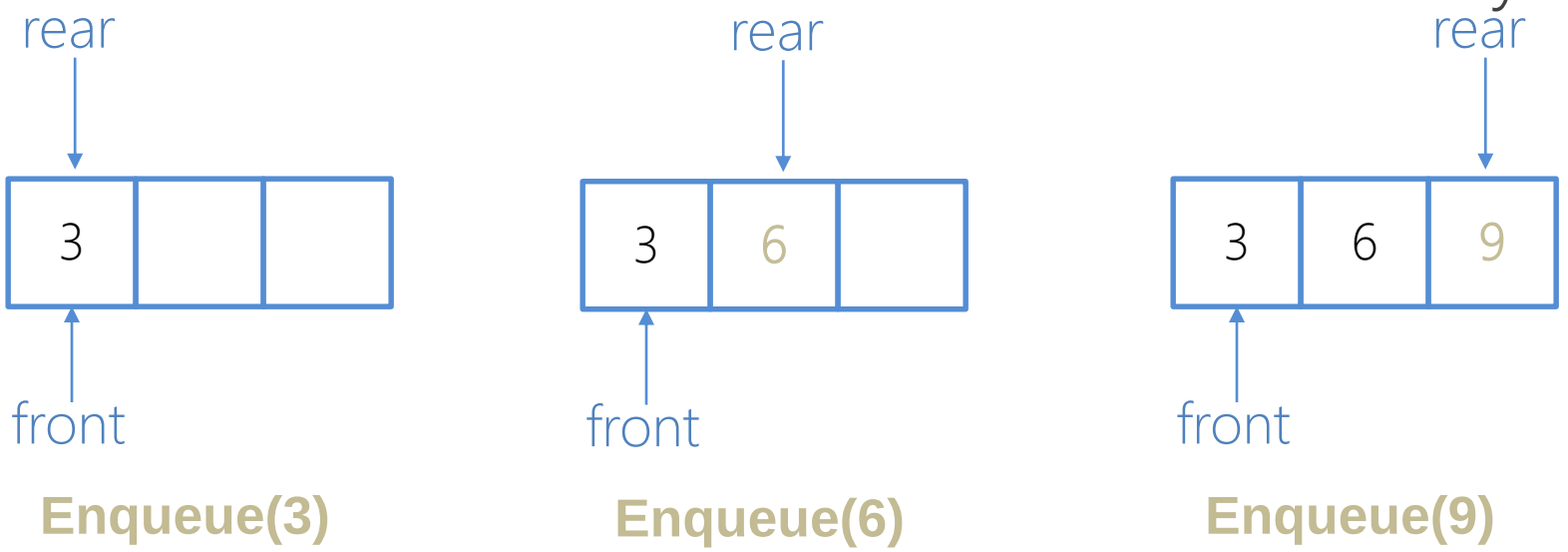
Implementation of Queue

- Recall the reason why we usually
  - implement a list using links?
  - implement a stack using array?

Topic		Array	Linked List
Efficiency	Enqueue		
	Dequeue		
space			

Queue Implementation Using Array

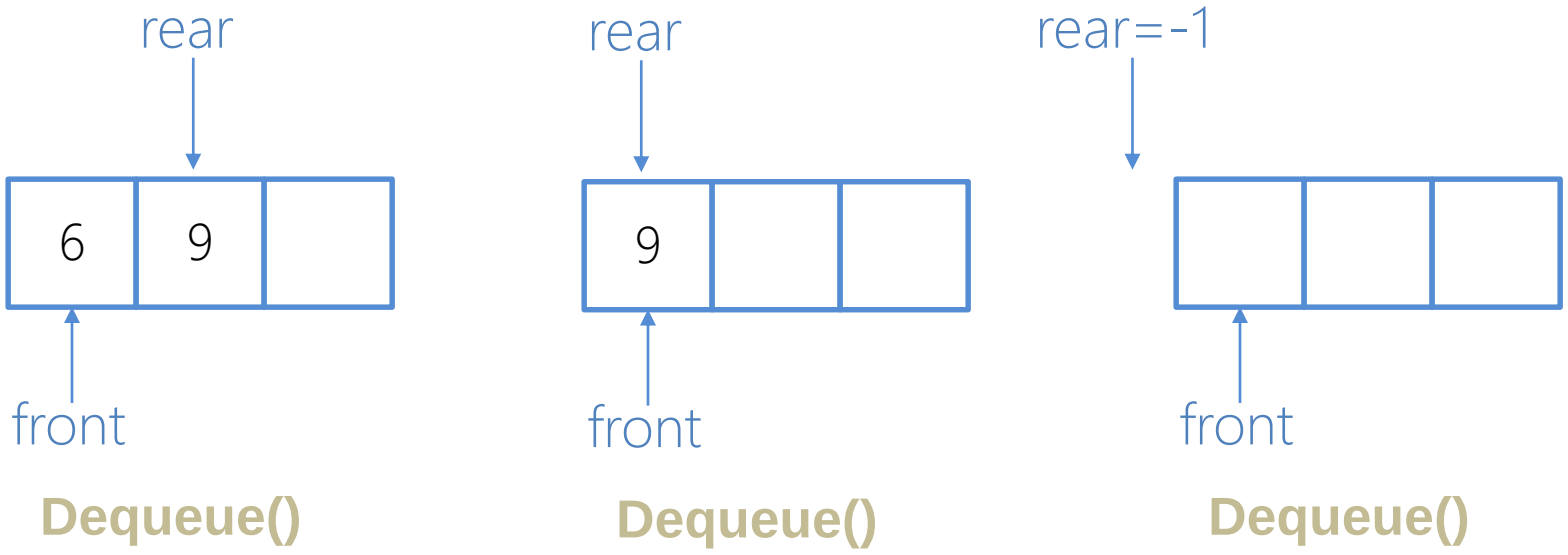
- There are several different algorithms to implement Enqueue and Dequeue
- Naive way: Enqueue
  - the front index is always fixed
  - the rear index moves forward in the array.




10
10
10

Queue Implementation Using Array

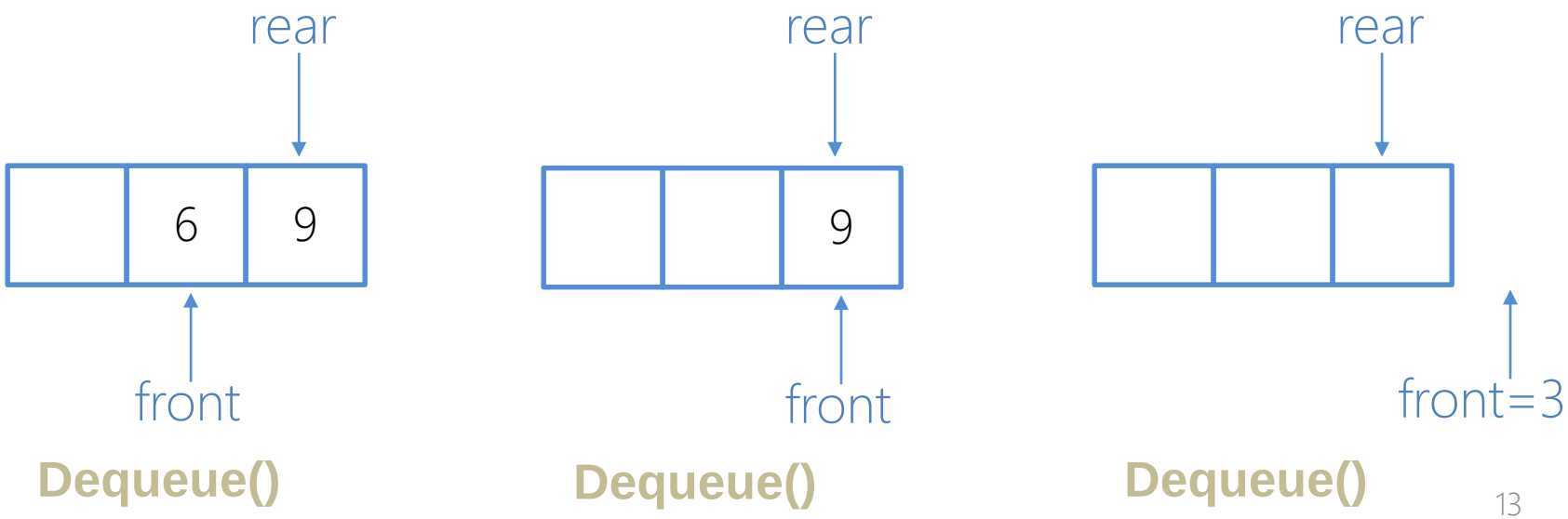
- Naive way: Dequeue
  - the front index is fixed
  - the element at the front the queue is removed
  - Move all the elements after it by one position.



NO GOOD

A Better Array Implementation

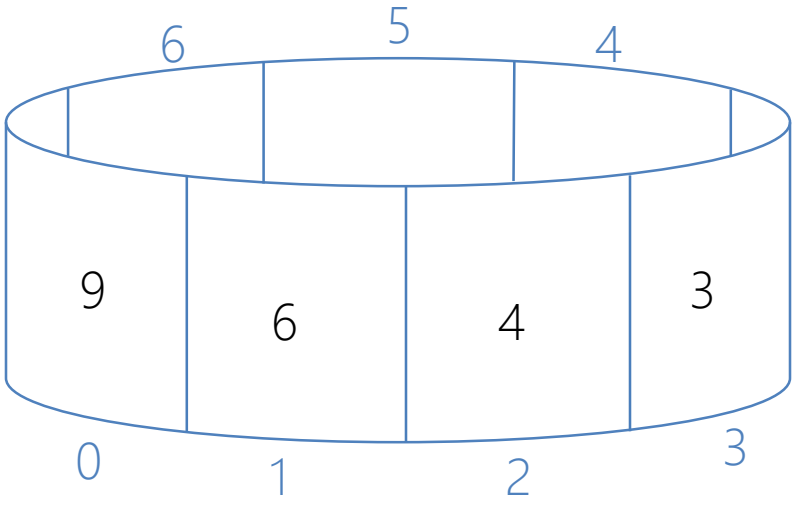
- When an item is enqueued, the rear index moves forward.
- When an item is dequeued, the front index also moves forward by one element



Efficient…but what is the **PROBLEM?**

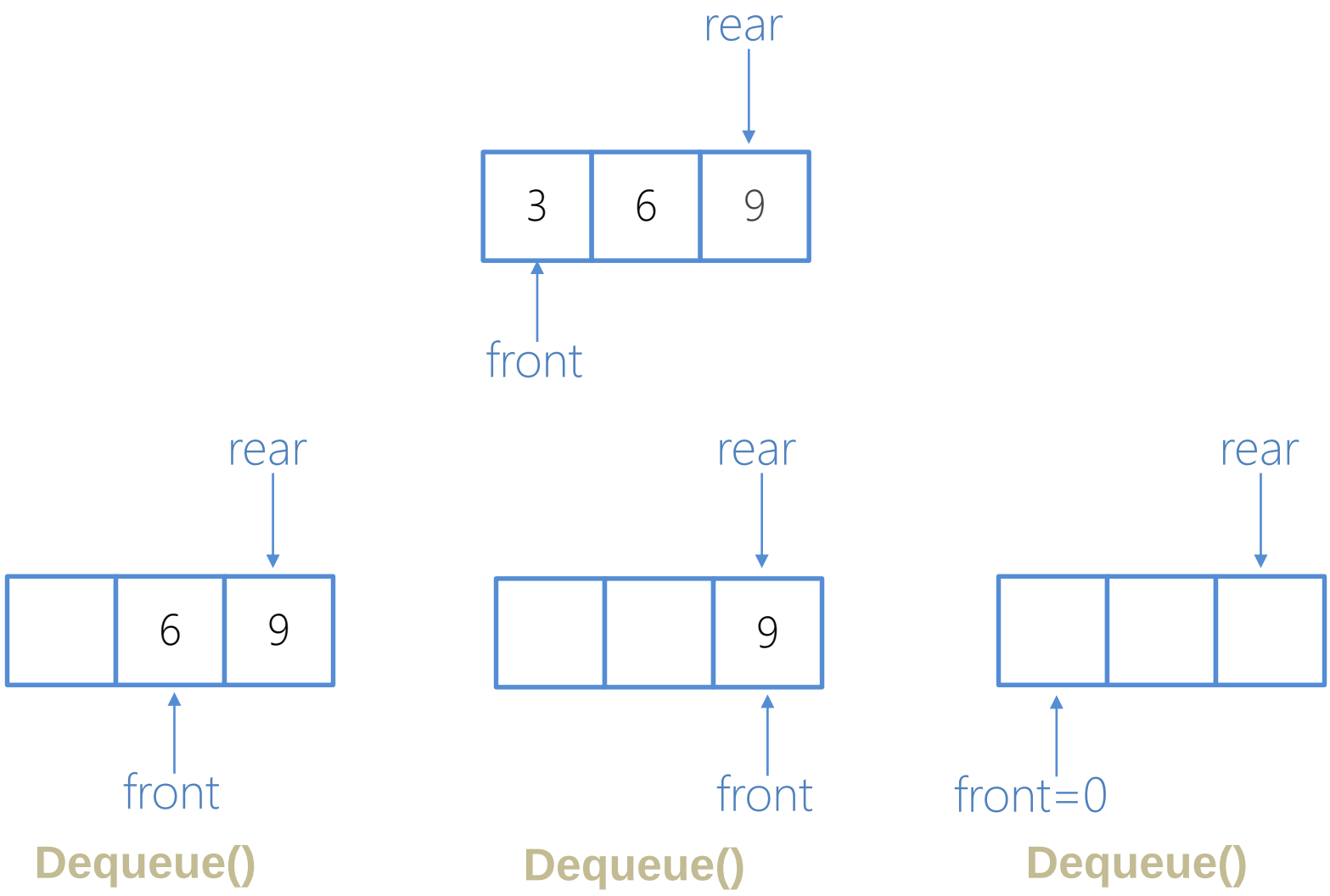
Final Solution: A Circular Array

- Circular array
  - When an element moves past the end of a circular array, it wraps around to the beginning

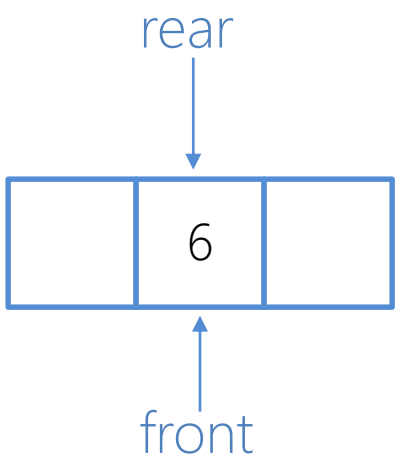


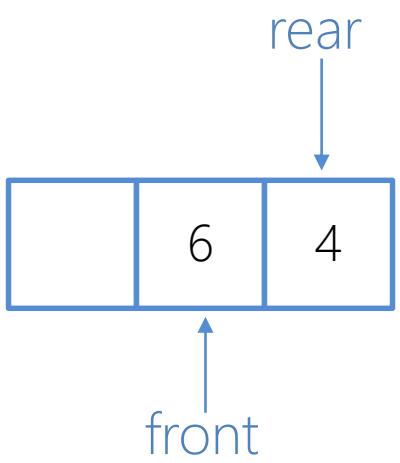
Index Growth: 0 → 1 → 2 → 3 → 4 → 5 → 6 → 0 → …

Dequeue in a Circular Array

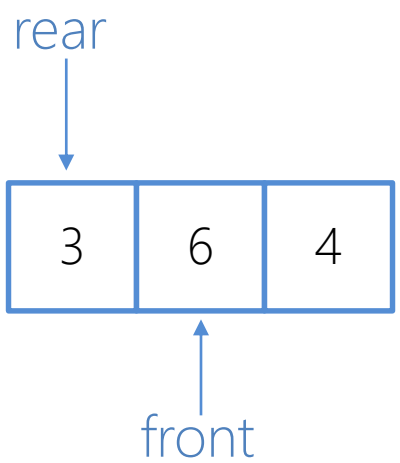


Enqueue in a Circular Array

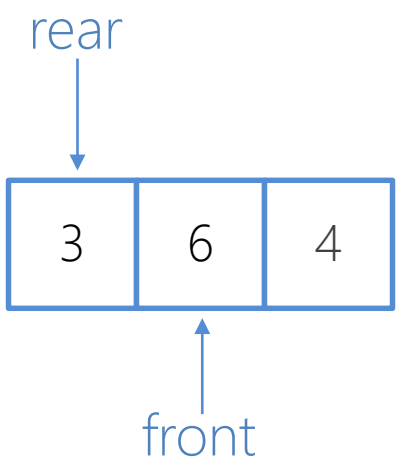




Eequeue(4)



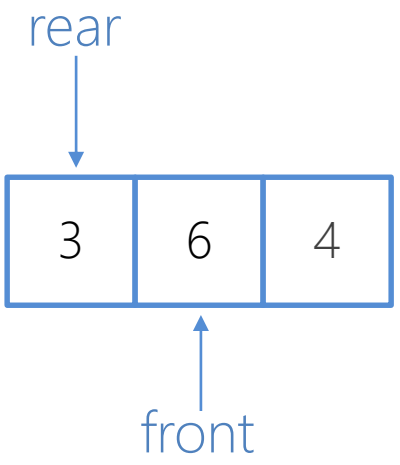
Eequeue(3)



Eequeue(1) Fails

Question to Ponder

- How to detect an empty or full queue using a circular array?
  - Empty
    - rear is one position before front
  - Full
    - rear is one position before front
  - Is this queue empty or full?



Question to Ponder

- How to detect an empty or full queue, using a circular array algorithm?

Use a counter which records number of elements in the queue.

Queue Implementation Using Array

- Data

```
typedef struct{
        double* values;
        int front;
        int rear;
        int counter;
        int maxSize;
} Queue;
```

- front: index of the front
- rear: index of the rear
- counter: number of elements in the queue
- maxSize: maximum size of the queue
- values: can be of any data type but we use double for demonstration

Queue Implementation Using Array

- Methods

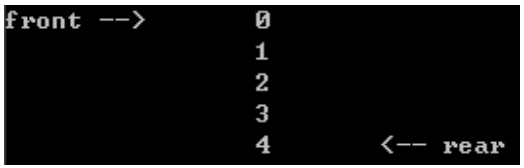
```
bool CreateQueue(Queue *queue, int size);
bool IsEmpty(Queue* queue);
bool IsFull(Queue* queue);
bool Enqueue(Queue* queue, double x);
bool Dequeue(Queue* queue, double* x);
void DisplayQueue(Queue* queue);
void DestroyQueue(Queue* queue);
```

Methods

- bool CreateQueue(Queue *queue, int size);
  - Creates an empty queue whose capacity is *size*
- bool IsEmpty(Queue *queue);
  - Returns true if the queue is empty and false otherwise
- bool IsFull(Queue *queue);
  - Returns true if the queue is full and false otherwise
- bool Enqueue(Queue* queue, double x);
  - Adds a new element with value *x* to the rear of the queue
  - Returns true if the operation is successful and false otherwise

Methods

- bool Dequeue(Queue* queue, double* x);
  - Removes an element from the front of the queue
  - Returns true if the operation is successful and false otherwise
  - Passes the value of the front element to *x*
- void DisplayQueue(Queue* queue);
  - Prints all the elements of the queue



- void DestroyQueue(Queue* queue);
  - Frees the memory occupied by the queue

CreateQueue

```
#include <stdlib.h>
bool CreateQueue(Queue *queue, int size){
    if (size <= 0)
        return false;
    queue->values = (double*)malloc(sizeof(double)*size);
    queue->front = 0;
    queue->rear = -1;
    queue->counter = 0;
    queue->maxSize = size;
    return true;
}
```

Why?
Any other valid initialization values?

Enqueue

```
bool Enqueue(Queue* queue, double x){
    if(IsFull(queue))
        return false;
    queue->rear = (queue->rear+1) % queue->maxSize;
    queue->values[queue->rear] = x;
    queue->counter++;
    return true;
}
```

Circular index increase

Using Queue

```
include "queue.h"
int main(void) {
   Queue queue;
   double value;
   CreateQueue(&queue, 5);
   puts("Enqueue 5 items.");
   for (int x = 0; x < 5; x++)
      Enqueue(&queue, x);
   puts( "Now attempting to enqueue
   again...");
   Enqueue(&queue, 5);
   DisplayQueue(&queue);
   Dequeue(&queue, &value);
   printf("Retrieved element = %g\n", value);
   DisplayQueue(&queue);
   Enqueue(&queue, 7);
   DisplayQueue(&queue);
   DestroyQueue(&queue);
}
```

```
include "queue.h"
int main(void) {
   Queue queue;
   double value;
   CreateQueue(&queue, 5);
   puts("Enqueue 5 items.");
   for (int x = 0; x < 5; x++)
      Enqueue(&queue, x);
   puts( "Now attempting to enqueue aga
   Enqueue(&queue, 5);
   DisplayQueue(&queue);
   Dequeue(&queue, &value);
   printf("Retrieved element = %g\n", v
   DisplayQueue(&queue);
   Enqueue(&queue, 7);
   DisplayQueue(&queue);
   DestroyQueue(&queue);
}
```

```
Enqueue 5 items.
Now attempting to enqueue again...
Error: the queue is full.
front -->       0
                1
                2
                3
                4       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4       <-- rear
front -->       1
                2
                3
                4
                7       <-- rear
```

Side Notes: File Structure in a C Project

queue.h

```
typedef struct{
            double* values;
            int front;
            int rear;
            int counter;
            int maxSize;
} Queue;
bool CreateQueue(Queue *queue,
int size);
/*
   function: …
   input:
      queue - …
      size - …
   output: …
*/
bool IsEmpty(Queue* queue);
bool IsFull(Queue* queue);
…
```

queue.cpp

```
include "queue.h"
#include <stdlib.h>
bool CreateQueue(Queue *queue,
int size){
      if (size <= 0)
            return false;
      queue->values =
      (double*)malloc(sizeof(do
      uble)*size);
      queue->front = 0;
      queue->rear = -1;
      …
}
…
```

main.cpp

```
include "queue.h"
int main(void) {
      Queue queue;
      double value;
      CreateQueue(&queue, 5);
      puts("Enqueue 5 items.");
      for (int x = 0; x < 5;
      x++)
            Enqueue(&queue, x);
      puts( "Now attempting to
      enqueue again...");
      …
}
```

Task

- Write *queue.h* and *queue.cpp* which implement the queue data structure.
- Refer to *list.h* and write proper comments in *queue.h* to describe every function.
- Compress *queue.h* and *queue.cpp* to a *.zip* file and submit it to iSpace.