Lecture 9.1

Topics:

1. Queue ADT – Introduction
2. Queue Implementation – (Circular) Array-Based Approach

---

1. Queue ADT -- Introduction

In general, a queue is a data structure that stores and retrieves items in a **First-In-First-Out** (FIFO) order. Note that a stack is a **Last-In-First-Out** (LIFO) data structure.

1.1 Definition

A queue is a data structure that holds a sequence of objects (or, set of data values) and provides access to its objects in **FIFO** order. A queue has two ends, which are called the **rear** and the **front** of the queue. These two ends will provide current locations of the first and last objects from which operations can take place in the queue.

Figure 1 depicts a queue with its rear and front indicated.

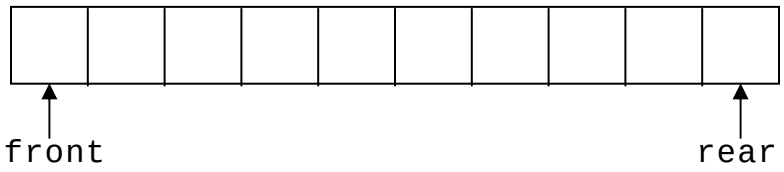

Figure 1 General queue

2.2 Applications

There are many applications of queue data structures.

a. Computer Operating System
b. Communication Software
c. Airline Checking Counter
d. Store Checking Counter

2.3 Queue Operations

When an object is added to a queue, it will be added to the rear. When an element is removed from the queue, it is removed from the front. The two primary queue operations are enqueuing and dequeuing.

- **To enqueue means to insert or add an object at the rear of a queue.**
- **To dequeue means to remove or delete an object at the front of a queue.**

There are several algorithms for implementing these operations based on the structures and behaviors such as linked lists, arrays, fixed-end, moving-end, etc.

The linked list approach offers dynamic queue with no size limit. The array-based approach does inherit the size limitation but in general faster than the linked list approach.

In the array-based approach, the basic considerations are with fixed ends and moving ends. The fixed end approach presents inefficiency while a moving end approach offers a better choice. In forming one of these moving-end queues, one must consider the case where the ends need to be

wrapped around as the queue is not full and being added with new object while the ends are moving around.

Thus, the consideration of circular queue comes into the picture. The following implementation will be for a moving end and circular array approach.

2. Queue Implementation – (Circular) Array-Based Approach

When using an array with moving ends, the queue would need to have the ends move past the last objects in the array. If one can think of the array as circular instead of a linear array, then the queue allows the ends to wrap around the ends to the beginning.

The wrapping will take place with the index of the array element being used in **`mod(increasing_order)`** value. That means the index is incremented and may be modulated (modulo-operated) to have its value in the range of **`[ 0, arraySize – 1 ]`**.

For examples, let's consider the example depicted in **Figure 2** with a partially filled queue.

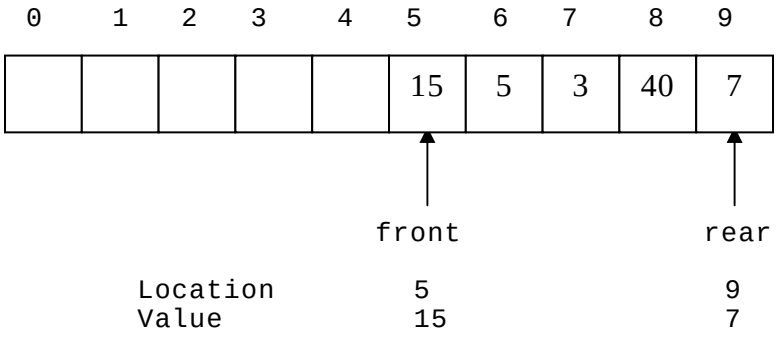

Figure 2 A partially filled queue

Suppose that an enqueue operation is performed to add the value 100 into the queue. The result is shown is **Figure 3** below.

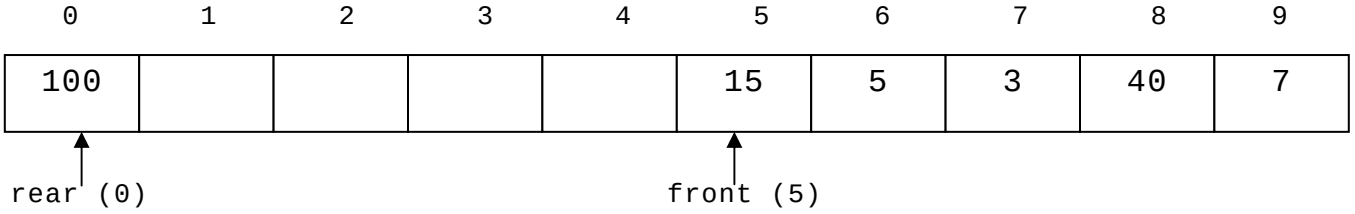

Figure 3 A wrapped around circular queue

What is the code to handle this? Let's consider the implementation below.

Example 1

```
/**
 * Program:    cis27L0911.c
 * Discussion: Queue -- Array Based Implementation
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct IntQueue {
  int queueSize;
  int front;
  int rear;
  int currentSize;
```

```
  int* intArray;
};

typedef struct IntQueue* QueueInt;

// Function prototypes
void enqueue(int obj, QueueInt old);
void dequeue(QueueInt old);
int isEmpty(QueueInt old);
int isFull(QueueInt old);
void empty(QueueInt old);

int main() {
  int ch;

  QueueInt intQueue = (QueueInt) malloc(sizeof(QueueInt));
                      /*What is wrong here?*/
  assert(intQueue);

  intQueue->queueSize = 4; /*Or, asking for queue size*/

  intQueue->intArray = (int *) malloc(sizeof(int) *
                                      intQueue->queueSize);
  assert(intQueue->intArray);

  empty(intQueue);

  enqueue(5, intQueue);
  enqueue(5, intQueue);
  enqueue(5, intQueue);
  enqueue(5, intQueue);
  enqueue(5, intQueue);

  printf("\nThe queue is full at this point!\n");

  dequeue(intQueue);
  dequeue(intQueue);
  dequeue(intQueue);
  dequeue(intQueue);
  dequeue(intQueue);

  enqueue(5, intQueue);
  enqueue(5, intQueue);
  enqueue(5, intQueue);

  printf("\nEnter a character + ENTER to quit!\n");
  scanf("%d", &ch);

  return 0;
}

// Function definitions
void enqueue(int obj, QueueInt old) {
  if (isFull(old)) {
    printf("\nThis queue is full!\n");
  } else {
    /*How do you simplify/replace this code fragment?*/
    if (old->rear == (old->queueSize - 1)) {
      old->rear = 0;
    } else {
```

```
      old->rear++;
    }
    old->currentSize++;
  }
  return;
}

void dequeue(QueueInt old) {
  if (isEmpty(old)) {
    printf("\nThis is an empty queue!\n");
  } else {
    old->front = (old->front + 1) % old->queueSize;
    old->currentSize--;
  }
  return;
}

int isEmpty(QueueInt old) {
  if (old->currentSize)
    return 0;
  else
    return 1;
}

int isFull(QueueInt old) {
  if (old->currentSize == old->queueSize)
    return 1;
  else
    return 0;
}

void empty(QueueInt old) {
  old->front = old->queueSize - 1;
  old->rear = old->queueSize - 1;
  old->currentSize = 0;

  return;
}

/*PROGRAM OUTPUT

This queue is full!

The queue is full at this point!

This is an empty queue!

Enter a character + ENTER to quit!
q
*/
```