Queues

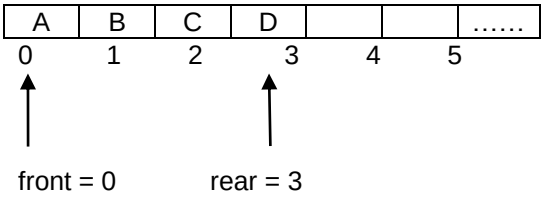
- The second data structure we will deal with is a *queue*. Physically, similar to a stack, a queue is also an array containing data. However, the operations on a queue can be done only in a specific order.

- As you know, in a stack, there is only one end, the top, and any new item to be added to the stack must be added at the top, and any item to be removed from the stack must be removed from the top. In a queue, there are two ends, the *front* and the *rear*. A new item must be added to the rear, and any item to be removed must be removed from the front.

- In order to understand the logic of queues, always think of the items in the queue as people standing in a queue in real life. If someone is going to leave the queue, the person at the front has the right to leave. If a new person wants to join the queue, he must go to the rear of the queue.

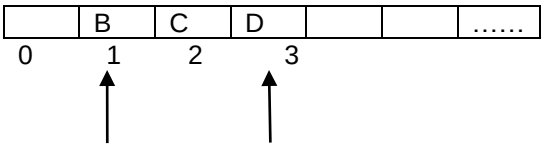
- Remember that, a stack works in a Last In First Out (LIFO) strategy. The last inserted item should be taken first out of the stack. However, a queue works in a First In First Out (FIFO) strategy. The first inserted item should be removed first from the queue.

Example: Let the figure below represent a queue:



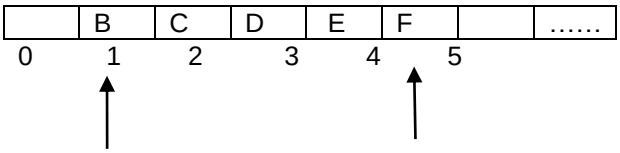
front = 0 rear = 3

If we remove one element from the queue ('A'):



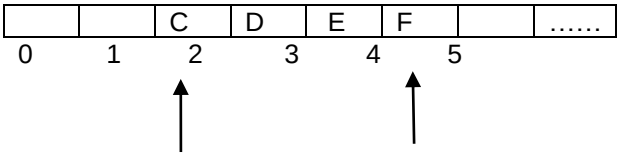
front = 1 rear = 3

If we insert 'E' and 'F' to the queue now:



front = 1 rear = 5

Let us remove one element from the queue ('B')



front = 2 rear = 5

Definition and Declaration of Queues

- An array based queue can be defined as a structure with three members:
  - an array to hold the data items.
  - two integer values, one shows the position of the element at the front, the other shows the position of the element at the rear.

Example: Define a queue type and declare a queue to store at most 50 integers.

```
typedef struct {
      int front, rear;
      int data[50];
} queue_t;

queue_t q;
```

Example: Define a queue type and declare a queue to contain the name, sex and birthdate of at most 2000 persons.

- Our data is complex. It is easier if we define its structure first:

```
typedef struct {
      int   day, month, year;
} date_t;

typedef struct {
      char   name[40], sex;
      date_t birthdate;
} person_t;

typedef struct {
      int      front, rear;
      person_t data[2000];
} perqueue_t;
...

perqueue_t per;
```

- If we want to keep information about the people in three villages, in three separate queues, we need to declare three queue variables as:

```
perqueue_t village1, village2, village3;
```

  or an array with three elements of the `per_queue` type as:

```
perqueue_t village[3];
```

- This type of queue is referred to as a *linear queue*, checking for a full queue must be done by checking the value of `rear` only.

- As you have probably noticed, a linear queue is very inefficient, because although the data array is not completely full, if the last element is full, it will not allow us to insert new items. Hence, you may have a large queue with a single item, but just because this item is sitting in the last element of the data array, you can not insert any new items.

- Some real life queues also work like this. For example, if you go to a government hospital, you must first of all register to a queue. However, in such hospitals there is a certain limit on number of patients a doctor can examine in one day, for instance 25. This is the size of the queue. If you are late and that many people arrived before you, the queue will become full, and you will not be accepted.

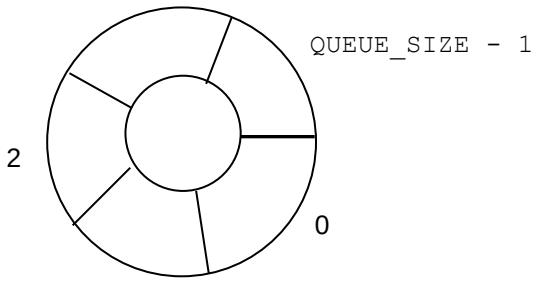
- In such a situation, can you require to wait for a while, so that some patients will be examined and some empty places will be opened in front of the queue? Impossible! Because while those patients are being examined, some time will pass, and the remaining time will be enough to examine only the remaining patients.

- However, in some other real life queues, such as those in front of a casier, there is not such a limit. After a person is served, the others step forward to fill in that position. Thus, to implement such a queue, you need to shift all items in the queue one position left, after each removal. But, this is too costly, because it will take a lot of CPU time.

Another solution is using a different data structure, such as a circular queue.

- In a *circular queue*, `q.data[0]` comes after `q.data[QUEUE_SIZE - 1]`.

When you try to insert a new item into the queue, even if `q.rear` is equal to `QUEUE_SIZE-1`, if `q.front != 0`, instead of saying that the queue is full, make `q.rear` 0, and put the new item into that position.



Define a circular queue structure and write the functions for operations on a circular queue.

Hint: Store the number of items in the queue as a part of the queue structure:

```
typedef struct {
      int   front, rear;
      QType data[QUEUE_SIZE];
      int   counter;
} queue_t;
```

Initialization of an empty queue means initializing its `front` to 0, `rear` to -1, so that when `insert` inserts the first data item into the queue, both will become 0.

```
      q->front = 0;
      q->rear = -1;
and number of elements are set to 0.
      q->counter = 0;
```

Queue Operations

- There are five main queue operations:

    1. Initialization of an empty queue
    2. Checking if the queue is empty
    3. Checking if the queue is full
    4. Inserting a new item into the queue (INSERT / ENQUEUE)
    5. Removing an item from the queue (REMOVE / DEQUEUE)

- The functions for a circular queue are defined and put into the header file `queue_int.h`. So, you just need to include that file to your programs if you need to deal with integer queues.

```
// Circular Queue Implementation, CTIS 152

#define QUEUE_SIZE  100

typedef  int  QType;

QType  QUEUE_EMPTY = -987654321;

typedef struct _Queue
{     int front, rear;
      QType data[QUEUE_SIZE];
      int counter;
} queue_t;

//Functions in this file...
void initializeQ (queue_t *q);
int isEmptyQ (queue_t *q);
int isFullQ (queue_t *q);
void insert (queue_t *q, QType item);
QType remove (queue_t *q);

//--------------------------------------------------------------------------

void initializeQ (queue_t *q)
{
      q->front = 0;
      q->rear  = -1;
      q->counter = 0;
}

//--------------------------------------------------------------------------

int isEmptyQ (queue_t *q)
{
      if (q->counter == 0)
            return 1;
      return 0;
}

//--------------------------------------------------------------------------

int isFullQ (queue_t *q)
{
      if (q->counter == QUEUE_SIZE)
            return 1;
      return 0;
}
```

```
//---------------------------------------------------------------------------

void insert (queue_t *q, QType item)
{
      if (isFullQ (q))
            printf("Error: Queue is full!\n");
      else
      {     q->rear = (q->rear + 1) % QUEUE_SIZE;   // make it circular
            q->data[q->rear] = item;
            (q->counter)++;
      }
}

//---------------------------------------------------------------------------

QType remove (queue_t *q)
{
      QType item;
      if (isEmptyQ (q))
      {     printf("Error: Queue is empty!\n");
            item = QUEUE_EMPTY;
      }
      else
      {     item = q->data[q->front];
            q->front = (q->front + 1) % QUEUE_SIZE;
            (q->counter)--;
      }
      return item;
}
```