Queues

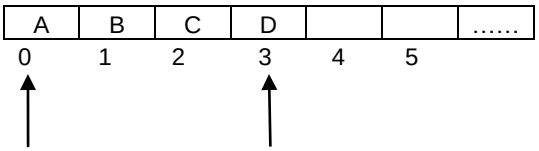
- The second data structure we will deal with is a *queue*. Physically, similar to a stack, a queue is also an array containing data. However, the operations on a queue can be done only in a specific order.

- 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.

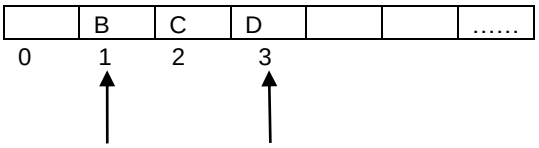
- Remember that, a stack works in a Last In First Out (LIFO) strategy. The last inserted item should be taken first out of the stack. However, a queue works in a First In First Out (FIFO) strategy. The first inserted item should be removed first from the queue.

Example: Let the figure below represent a queue:



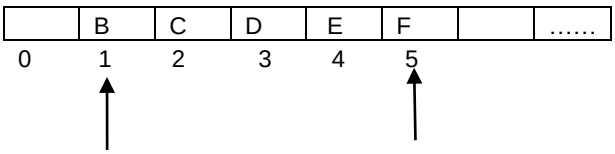
front = 0 rear = 3

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



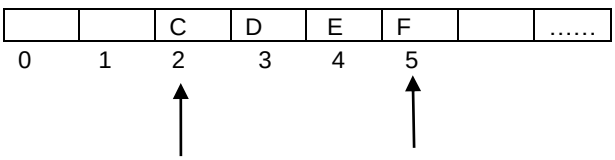
front = 1 rear = 3

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];
} per_queue;
...

per_queue 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:

```
per_queue village1, village2, village3;
```

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

```
per_queue 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.

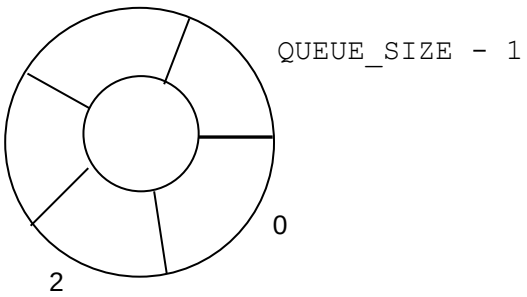
- In such a situation, can you require to wait for a while, so that some patients will be examined and some empty places will be opened in front of the queue? Impossible! Because while those patients are being examined, some time will pass, and the remaining time will be enough to examine only the remaining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

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;
#define QUEUE_EMPTY -987654321

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

//Functions in this file...
void initialize_q(queue_t *q);
int is_full_q(queue_t *q);
int is_empty_q (queue_t *q);
void insert (queue_t *q, QType item);
QType remove (queue_t *q);

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

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

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

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

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

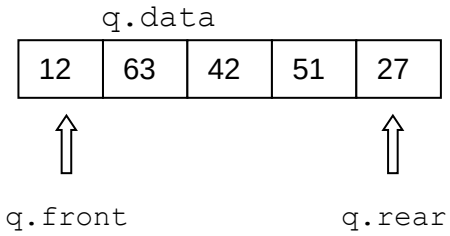
Queue Exercises

Example: Given 5 integers, insert them into a queue.

- First of all, we must initialize our queue as empty. Then, we need to read the input data using a loop. We must first read each integer into a variable, then we can insert this variable into the queue:

```
queue_t q;
int num, k;
initialize_q(&q);
for (k = 1; k <= 5; k++) {
      scanf("%d", &num);
      insert(&q, num);
}
```

- Assume that our input data is 12, 63, 42, 51, 27. After the loop terminates, all data will be put into the queue, and the queue will look like the following, if the QUEUE_SIZE was 5.



- If you are asked to fill a queue with data, no matter what the size of the queue is or no matter how many items the queue already contains, you can go on reading data until the queue becomes full. Therefore our program segment should be as follows:

```
while (!is_full_q(&q)) {
      scanf("%d", &num);
      insert(&q, num);
}
```

Example: Output the data in the queue.

- When outputting, there is an important decision to make: Do you want the data to remain in the queue, or do you want to empty the queue as you output? If you want to empty the queue each time you output the value, you need to remove it, and don't forget that this value will be lost after outputting.

```
while (!is_empty_q(&q))
      printf("%5d\n", remove(&q));
printf("\n");
```

- After the loop terminates, we will see the following on the screen:

```
12   63   42   51   27
```

- Notice that, the output is in the same order as the input data.

- Since we removed each value before outputting, the queue is empty after the loop terminates. Thus, we destroyed the queue.

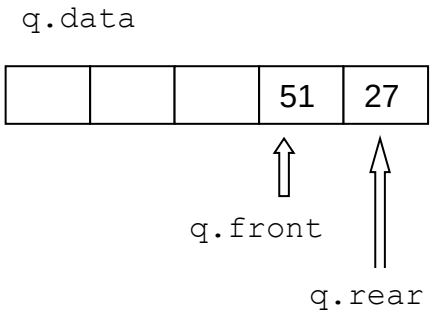
- If we don't want to destroy the queue, we can define the above program segment as a function, sending the queue as a value parameter, so that even if it becomes empty in the function, the actual queue will remain the same:

```
void display_q(queue_t q) {
     while (!is_empty_q(q))
         printf("%5d\n", remove(&q));
     printf("\n");
}
```

Example: Remove the first 3 items (means we want to remove 12, 63 and 42).

```
for (k = 1; k <= 3; k++)
     num = remove (&q);
```

- After the loop, the queue is as follows:



- Remember that, with two variables of the same structure, direct assignment is possible. Thus, we can copy all values in queue `q` back into queue `e` with a single assignment statement as:

```
e = q;
```

Home Exercises:

1) Define a function that removes all occurrences of a certain item from a queue, without changing the order of the other items.

2) Define a function that inserts an item after a certain item in a queue, without changing the order of the other items. Assume that there is enough space in the queue.

Example: Define a function that reverses a queue.

```
#include <stack_int.h>
#include <queue_int.h>

void reverse_queue(queue_t *q) {
      stack_t s; //local stack structure
      initialize_s(&s);
      /* Remove all items from the queue and push them onto the stack */
      while (!is_empty_q(q))
           push(&s, remove(q));
      initialize_q(q);
      /* Pop all items from the stack and insert them into the queue */
      while (!is_empty_s(&s))
           insert(q, pop(&s));
}
```

Example: Trace the following program segment, and show the output. Assume that `QUEUE_SIZE` is 5. (Remember that `q.rear` is initialized to -1 in `queue_int.h`)

```
#include <stack_int.h>
#include <queue_int.h>
//main
...
stack_t s;
queue_t q;
int n;
initialize_s(&s);
initialize_q(&q);
while (!is_full_q(&q)) {
      insert(&q, q.rear);
      push(&s, s.top);
}
while (!is_empty_s(&s)) {
      n = pop(&s);
      printf("%d   %d\n", n, s.top);
      n = remove(&q);
      printf("%d   %d\n", n, q.front);
}
```

Homework:

Write a C program that will get the queue values from the user until the user enters a sentinel value (-9). Then the program will display a menu with the options below:

- Print Queue
- Clear Queue
- Count Queue
- Remove Maximum Element
- Send Nth To End
- Exit

After getting the choice of the user, the program will do the operations according to the choice by using the functions following:

PrintQueue	: Display the elements of the queue (Queue content will not change)
ClearQueue	: Remove all elements
CountQueue	: Count the elements in the queue
RemMaxQueue	: Remove the maximum element from the queue
SendNthToEnd	: Send the Nth element from the start to the end of the queue

Example Run:

```
Enter the numbers for the queue (-9 to stop): 3 4 5 2 -9

1) Print Queue
2) Clear Queue
3) Count Queue
4) Remove Maximum Element
5) Send Nth To End
6) Exit
Enter your choice: 1
3  4  5  2

Enter your choice: 3
Number of elements in the queue: 4

Enter your choice: 5
Enter N: 5
N must be between 1 and 4

Enter your choice: 5
Enter N: 2

Enter your choice: 1
3  5  2  4

Enter your choice: 4

Enter your choice: 1
3  2  4

Enter your choice: 2
The queue is empty!!

Enter your choice: 3
Number of elements in the queue: 0

Enter your choice: 7

Enter your choice: 6
```