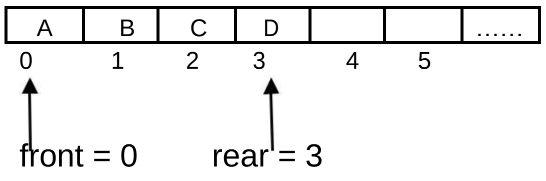
Queues

- 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.

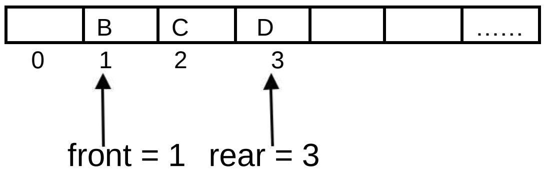
- As you know, in a stack, there is only one end, the top, and any new item to be added to the stack must be added at the top, and any item to be removed from the stack must be removed from the top. In a queue, there are two ends, the *front* and the *rear*. A new item must be added to the rear, and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

- 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.

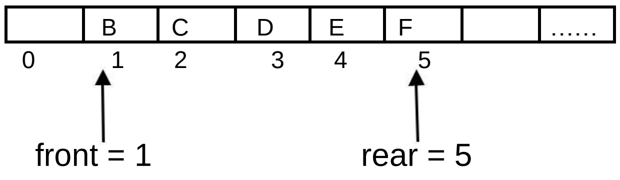
Example: Let the figure below represent a queue:



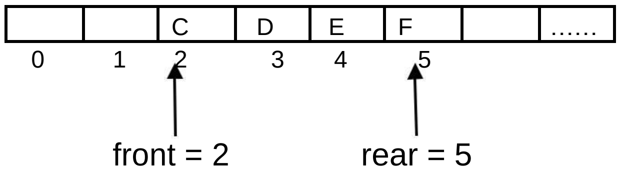
If we remove one element from the queue (‘A’):



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



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



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];
  ```

- How can we reach to the name of the first person in the first village?

  ```
  village[0].data[village[0].front].name
  ```

- How can we reach to the birth year of the last person in the third village?

  ```
  village[2].data[village[2].rear].birthdate.year
  ```

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)

- Assume that we have already defined a queue structure as follows:

```
#define QUEUE_SIZE   100
#define QUEUE_EMPTY -987654321
typedef int QType;
typedef struct {
     int   front, rear;
     QType data[QUEUE_SIZE];
} queue_t;
```

- As we did above, we may define an alias for the type of the items in the queue, and use it in the queue type definition. By this way, later if we want to use the same definition for other type of items, such as double or character, we only need to change the meaning of `QType`.

- 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 stack, both will become 0. Hence the function is so simple as:

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

- Checking if the queue is empty is not as trivial as in stacks. Checking if `front` is 0 and `rear` is -1 is not the solution, because these conditions are true only for an empty queue after initialization. What about if a queue becomes empty after some insert and remove operations? Notice that, both operations will increment the indices. Thus, each time an item is removed from the queue, the value of `front` increases. What happens when the queue becomes empty? `front` exceeds `rear`.

```
int is_empty_q(queue_t *q) {
     if (q->front > q->rear)
         return 1;
     return 0;
}
```

- Notice that, the function returns 1 just after the queue is initialized, because 0 > -1.

- If the rear of the queue shows the last position of the data array, it means that the queue is full. Therefore, to check if the queue is full, we just need to check if its rear is equal to `QUEUE_SIZE – 1`. You may pass the size as a parameter, or assume that it is a global constant, as in our example. Hence the function is

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

- Now, we are ready to write functions for insert and remove operations. As I said before, insert operation will increment the `rear`, and insert a new item into the rear element. Therefore, `insert` function needs the queue, and the item to be inserted as parameters. The data type of the item must be the same as the type of the data array in the queue. To be able to insert the new item into the queue, `insert` must check if the queue is full or not. The function is:

```
void insert(queue_t *q, QType item) {
     if (is_full_q(q))
         printf("Error : Queue is full!\n");
     else {
         q->rear = q->rear + 1;
         q->data[q->rear] = item;
     }
}
```

- Remove operation will remove the front element, and increment the `front`. Therefore, `remove` function needs the queue as parameter. The function will return the item in front of the queue. Thus, the data type of the function must be the same as the type of the data array in the queue. The function must check if the queue is empty or not. If the queue is empty, it can give an error message. The function is:

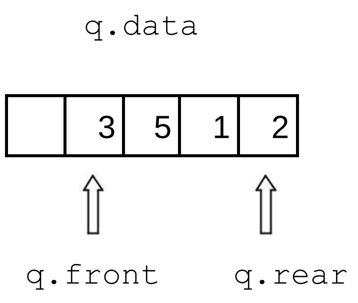
```
QType remove(queue_t *q) {
     QType item;
     if (is_empty_q(q)) {
         printf("Error : Queue is empty!\n");
         item = QUEUE_EMPTY;
     }
     else {
         item = q->data[q->front];
         q->front = q->front + 1;
     }
     return (item);
}
```

- Since the above function must return an item in any situation, but there are no items in the queue when it is empty, we assigned `item` to a constant `QUEUE_EMPTY` representing that no item could be removed. The value of that constant should be choosen as a value which does not appear within the queue.

Example: Trace the following program segment, and show how the queue looks like at the end.

```
#define QUEUE_SIZE 5
...
queue_t q;
int k, n;
initialize_q(&q);
for (k = 1; k <= 6; k += 2)
     insert(&q, k);
n = remove(&q);
for (k = 0; k < 3; k++)
     insert(&q, n + k);
```

- For convenience, when drawing queues, we will draw our data array horizontally, from the first element towards the last one. First the values 1, 3 and 5 are inserted into the queue. Then the value 1 is removed from the front. Finally the values 1 and 2 are inserted into the queue, and an error message is displayed, when the value 3 is being tried to be inserted, because the queue is full. So the queue looks like:



- How can you find the number of items in a queue?

```
q.rear - q.front + 1
```

- What about the number of items in a stack?

```
s.top + 1
```

- Can we use this property to check if the queue is empty? Yes, if the number of items in the queue is zero, the queue is empty. Therefore, the `is_empty_q` function could be written as:

```
int is_empty_q(queue_t *q) {
     /*  if the number of items in the queue is zero,
         the queue is empty */
     if (q->rear - q->front + 1 == 0)
         return 1;
     return 0;
}
```

- Can we say that if the number of items in the queue is equal to the size of the data array of the queue, then the queue is full? Although this is perfectly true, it should be remembered that the full condition for a queue is used to prevent insertion if there is no more space at the rear to insert a new item.

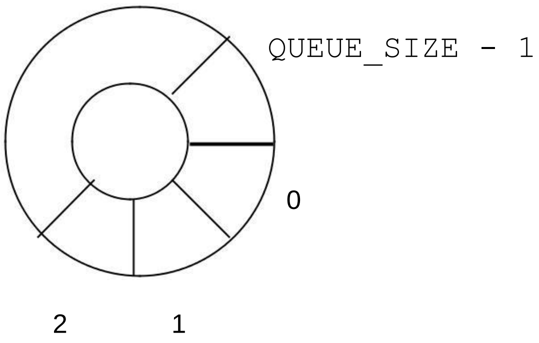
- With a queue such as the one in our example, which is referred to as a *linear queue*, checking for a full queue must be done by checking the value of `rear` only.

- 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.

- In some 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 or a linked list. You will learn linked lists later.

- 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.



- Therefore, if we used a circular queue in our example, after inserting 1 and 2 in `q.data[3]` and `q.data[4]`, it would be possible to insert 3 in `q.data[0]`, and the final values of `q.front` and `q.rear` would be 1 and 0, respectively.

Home Exercise: 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;
```

- 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.

```
#ifndef _CENG_QUEUE_H
#define _CENG_QUEUE_H

// Author: Evren
// Circular Queue Implementation, CENG 104
//

#define QUEUE_SIZE  100
#define NULL_ITEM -987654321

typedef  int  QType;

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


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


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


void insertQ(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->queue[q->rear] = item;
     q->counter++;
  }
}
```

```
QType removeQ(queue_t *q)
{
  QType tmp;
  if (isEmptyQ(q))
  {
     printf("Error : Queue is empty\n");
     tmp = NULL_ITEM;
  }
  else
  {
     tmp = q->queue[q->front];
     q->front = ( q->front + 1 ) % QUEUE_SIZE;
     q->counter--;

  }
  return tmp;
}

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

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

#endif
```