Universidad Autónoma de Madrid

Escuela Politécnica Superior

Estructuras de Datos

Colas

Índice

❶ El TAD Cola

❷ EdD y primitivas

❸ Implementación en C

❹ Implementación en C con *front* y *rear* punteros

Cola

Una **Cola** es una colección de elementos organizados de tal modo que el primero en entrar es el primero en salir

TAD Cola (I)

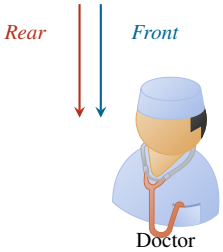
***First In, First Out* (FIFO):**

- Los elementos se insertan en la cola de uno en uno: insert/push
- Los elementos se extraen de la cola de uno en uno: extract/pop
- Se puede acceder tanto al primer elemento (front/head) como al último (back/rear/tail)
- Los elementos siempre se insertan en la última posición
- Los elementos siempre se extraen de la primera posición

TAD Cola (II)

Ejemplo: cola del médico

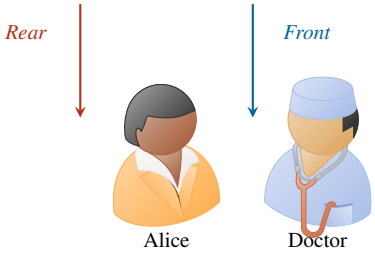
1. Cola vacía



TAD Cola (II)

Ejemplo: cola del médico

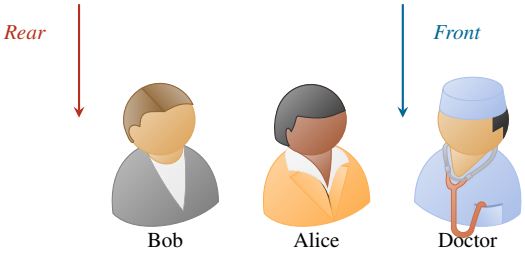
2. Entra Alice



TAD Cola (II)

Ejemplo: cola del médico

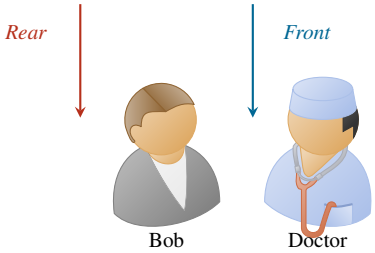
3. Entra Bob



TAD Cola (II)

Ejemplo: cola del médico

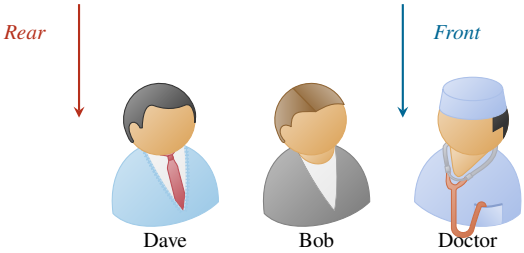
4. Sale Alice



TAD Cola (II)

Ejemplo: cola del médico

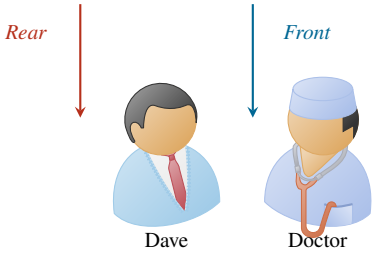
5. Entra Dave



TAD Cola (II)

Ejemplo: cola del médico

6. Sale Bob



Aplicaciones de las colas

En general, cualquier aplicación que implique el acceso concurrente a un recurso
Algunos **ejemplos**:

- En una impresora, el primer trabajo en llegar es normalmente el primero que se imprime (*first come first served*)
- Peticiones a un servidor
- Planificador del sistema operativo

En ocasiones puede ser necesario alterar el orden FIFO (elementos con diferentes prioridades)

Primitivas

- `Queue queue_new()`: crea e inicializa una cola
- `queue_free(Queue q)`: libera una cola
- `Boolean queue_isEmpty(Queue q)`: devuelve true si la cola está vacía
- `Status queue_push(Queue q, Element e)`: inserta un elemento en la cola
- `Element queue_pop(Queue q)`: extrae un elemento de la cola
- `Element queue_getFront(Queue q)`: devuelve el primer elemento **sin** modificar la cola
- `Element queue_getBack(Queue q)`: devuelve el último elemento **sin** modificar la cola

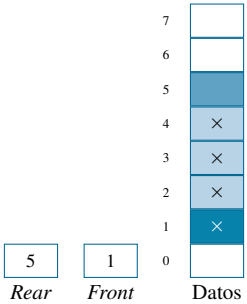
Primitivas en C++

Implementación en C++: `std::queue`

`(constructor)`	Construct queue (public member function)
`empty`	Test whether container is empty (public member function)
`size`	Return size (public member function)
`front`	Access next element (public member function)
`back`	Access last element (public member function)
`push`	Insert element (public member function)
`emplace`	Construct and insert element (public member function)
`pop`	Remove next element (public member function)
`swap`	Swap contents (public member function)

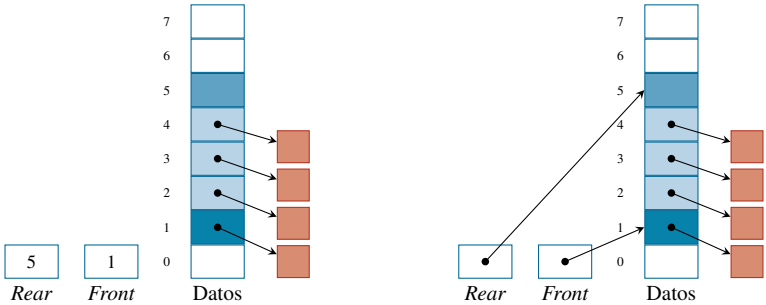
Estructura de datos

- **Datos**: Colección de elementos, en general del mismo tipo, almacenados de forma secuencial y accesibles desde dos puntos: front y rear
- ***Front***: Indicador de la posición del próximo elemento a extraer
- ***Rear***: Indicador de la posición donde se insertará el siguiente elemento



Estructura de datos en C

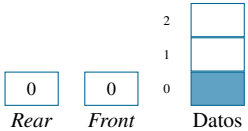
- **Datos**: se usará un array de punteros de tamaño fijo: `void *data[];`
- ***Front/Rear***: se consideran dos posibilidades:
  - Índices enteros: `int front, rear;`
  - Punteros: `void **front, **rear;`



Estructura de datos como array lineal (I)

Ejemplo de operaciones en la cola:

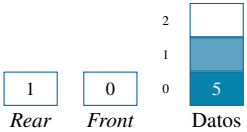
1. `q = queue_new()`



Estructura de datos como array lineal (I)

Ejemplo de operaciones en la cola:

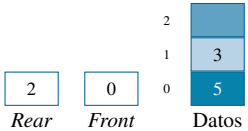
1. `q = queue_new()`
2. `queue_push(q, 5)`



Estructura de datos como array lineal (I)

Ejemplo de operaciones en la cola:

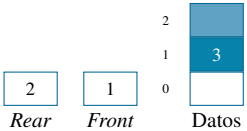
1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`



Estructura de datos como array lineal (I)

Ejemplo de operaciones en la cola:

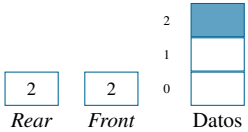
1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`



Estructura de datos como array lineal (I)

Ejemplo de operaciones en la cola:

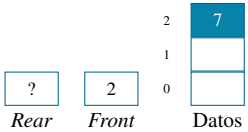
1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`
5. `queue_pop(q)`



Estructura de datos como array lineal (I)

Ejemplo de operaciones en la cola:

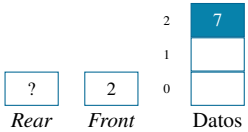
1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`
5. `queue_pop(q)`
6. `queue_push(q, 7)`



Estructura de datos como array lineal (I)

Ejemplo de operaciones en la cola:

1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`
5. `queue_pop(q)`
6. `queue_push(q, 7)`



Este enfoque tiene **dos problemas principales**:

- Limitación del número máximo de elementos
- Desperdicio de espacio

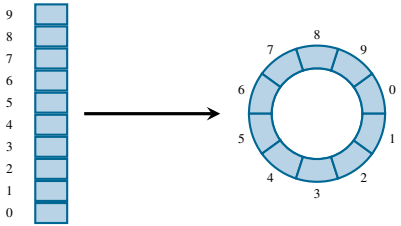
Estructura de datos como array lineal (II)

Posibles soluciones:

1. Con cada extracción se desplazan todos los elementos del array una posición (ineficiente)
2. Cuando el *rear* alcanza el final del array, se desplazan todos los elementos hasta que el *front* esté al principio (menos ineficiente)
3. Uso de un **array circular**

Estructura de datos como array circular (I)

Array circular:



Implementación:

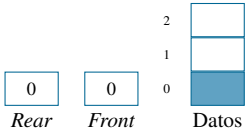
- Con cada extracción: `front = (front + 1) % MAX_QUEUE`
- Con cada inserción: `rear = (rear + 1) % MAX_QUEUE`

Sigue el problema de la limitación del número máximo de elementos

Estructura de datos como array circular (II)

Ejemplo de operaciones en la cola:

1. `q = queue_new()`

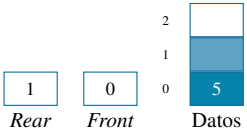


Cola vacía

Estructura de datos como array circular (II)

Ejemplo de operaciones en la cola:

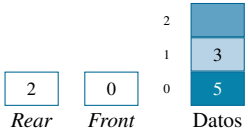
1. `q = queue_new()`
2. `queue_push(q, 5)`



Estructura de datos como array circular (II)

Ejemplo de operaciones en la cola:

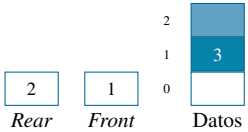
1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`



Estructura de datos como array circular (II)

Ejemplo de operaciones en la cola:

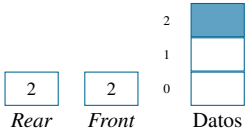
1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`



Estructura de datos como array circular (II)

Ejemplo de operaciones en la cola:

1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`
5. `queue_pop(q)`

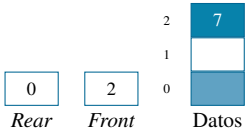


Cola vacía

Estructura de datos como array circular (II)

Ejemplo de operaciones en la cola:

1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`
5. `queue_pop(q)`
6. `queue_push(q, 7)`

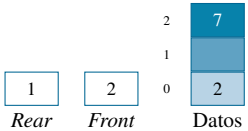


Se usa la circularidad y *rear* pasa al comienzo

Estructura de datos como array circular (II)

Ejemplo de operaciones en la cola:

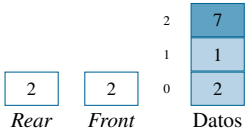
1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`
5. `queue_pop(q)`
6. `queue_push(q, 7)`
7. `queue_push(q, 2)`



Estructura de datos como array circular (II)

Ejemplo de operaciones en la cola:

1. `q = queue_new()`
2. `queue_push(q, 5)`
3. `queue_push(q, 3)`
4. `queue_pop(q)`
5. `queue_pop(q)`
6. `queue_push(q, 7)`
7. `queue_push(q, 2)`
8. `queue_push(q, 1)`



¿Cola vacía o llena?

Estructura de datos como array circular (III)

Nuevo problema:

- Cuando `front == rear`, ¿la cola está llena o vacía?

Solución:

- Sacrificar una posición del array
- No insertar cuando queda solo una posición libre
- Una cola con tamaño `MAX_QUEUE` tendrá espacio para `MAX_QUEUE - 1` elementos

Implementación en C (I)

EdD para Cola:

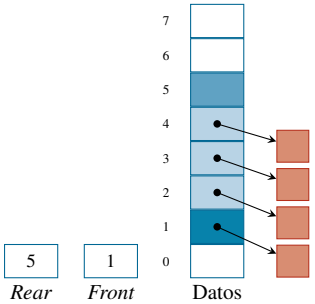
- Implementación con `int front, rear;`

Tipo de dato en queue.h

```
typedef struct _Queue Queue;
```

Estructura en queue.c

```
#define MAX_QUEUE 8

struct _Queue {
  void *data[MAX_QUEUE];
  int front;
  int rear;
};
```



Implementación en C (II)

Primitivas:

Cabeceras de las primitivas en `queue.h`

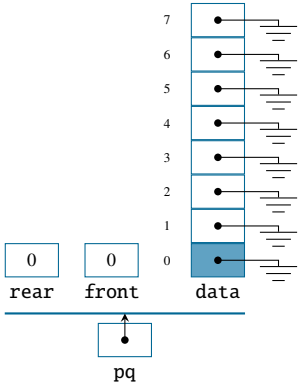
```
Queue *queue_new();
void queue_free(Queue *pq);
Boolean queue_isEmpty(const Queue *pq);
Status queue_push(Queue *pq, const void *e);
void *queue_pop(Queue *pq);
void *queue_getFront(const Queue *pq);
void *queue_getBack(const Queue *pq);
```

Implementación en C (III)

Crear e inicializar Cola:

Función `queue_new` **en** `queue.c`

```
Queue *queue_new() {
  Queue *pq = NULL;
  int i;

  pq = (Queue *)malloc(sizeof(Queue));
  if (pq == NULL) {
    return NULL;
  }

  for (i = 0; i < MAX_QUEUE; i++) {
    pq->data[i] = NULL;
  }
  pq->front = 0; pq->rear = 0;
  return pq;
}
```



Implementación en C (IV)

Destrucción de Cola:

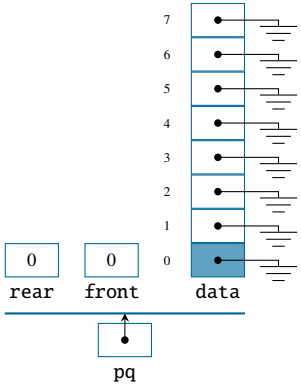
Función `queue_free` **en** `queue.c`

```
void queue_free(Queue *pq) {
  free(pq);
}
```

- En esta implementación:
  - La función que reservó la memoria para los elementos *debería* encargarse de liberarlos
  - La asignación `pq = NULL;` debe hacerse después de la llamada a `queue_free(pq)`

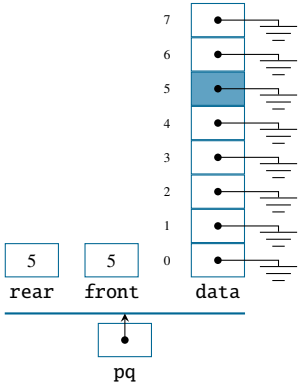
Implementación en C (V)

Comprobar si la cola está vacía:



Implementación en C (V)

Comprobar si la cola está vacía:



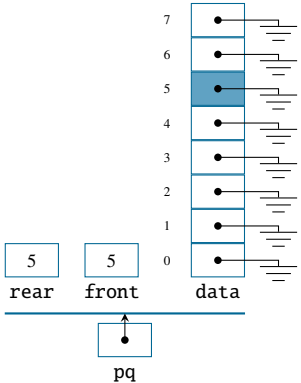
Implementación en C (V)

Comprobar si la cola está vacía:

Función `queue_isEmpty` en `queue.c`

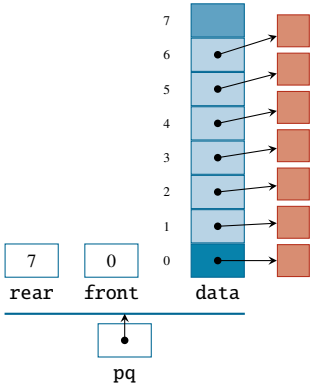
```
Boolean queue_isEmpty(const Queue *pq) {
  if (pq == NULL) {
    return TRUE;
  }

  if (pq->front == pq->rear) {
    return TRUE;
  }

  return FALSE;
}
```

- Esta función debe llamarse antes de extraer



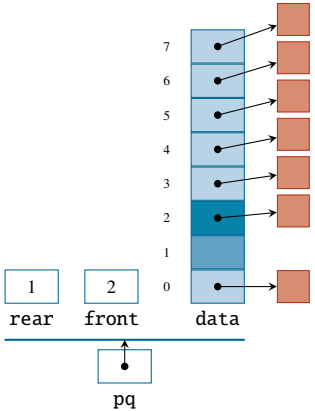
Implementación en C (VI)

Comprobar si la cola está llena:



Implementación en C (VI)

Comprobar si la cola está llena:



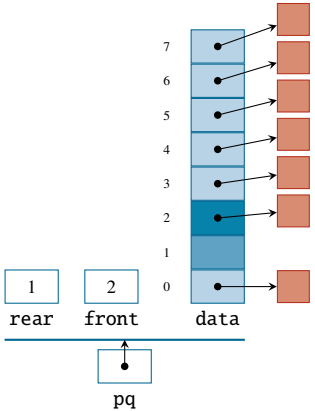
Implementación en C (VI)

Comprobar si la cola está llena:

Función _queue_isFull en queue.c

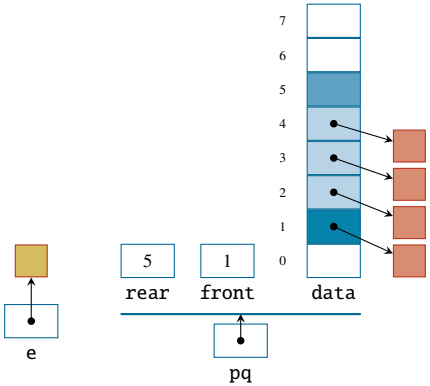
```
Boolean _queue_isFull(const Queue *pq) {
  if (pq == NULL) {
    return TRUE;
  }

  if (pq->front == (pq->rear + 1) %
      MAX_QUEUE) {
    return TRUE;
  }

  return FALSE;
}
```

Nota: Función privada; debe llamarse antes de insertar un elemento



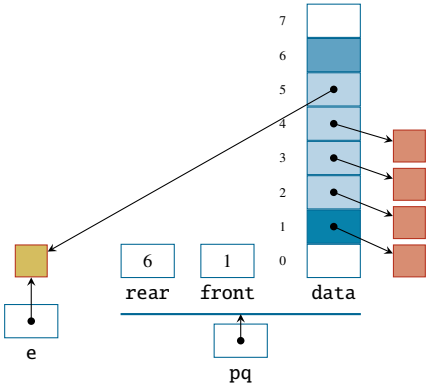
Implementación en C (VII)

Insertar elemento:



Implementación en C (VII)

Insertar elemento:



Implementación en C (VII)

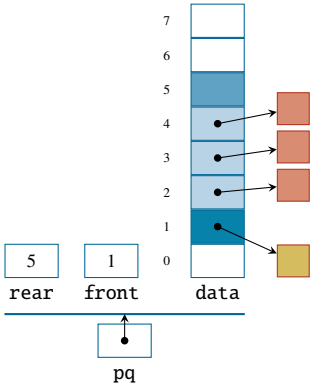
Insertar elemento:

Función queue_push **en** queue.c

```
Status queue_push(Queue *pq, const void *e) {
  if (pq == NULL || e == NULL || _queue_isFull(pq) == TRUE) {
    return ERROR;
  }

  pq->data[pq->rear] = (void *)e;
  pq->rear = (pq->rear + 1) % MAX_QUEUE;

  return OK;
}
```

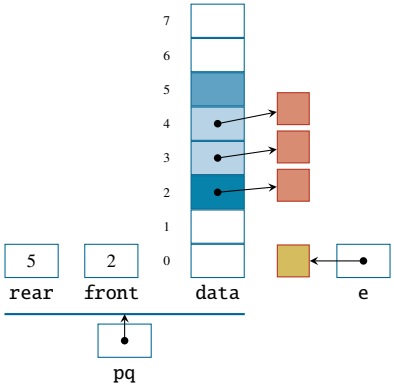
Implementación en C (VIII)

Extraer elemento:



Implementación en C (VIII)

Extraer elemento:



Implementación en C (VIII)

Extraer elemento:

Función queue_pop **en** queue.c

```
void *queue_pop(Queue *pq) {
  void *e = NULL;

  if (pq == NULL || queue_isEmpty(pq) == TRUE) {
    return NULL;
  }

  e = pq->data[pq->front];
  pq->data[pq->front] = NULL;
  pq->front = (pq->front + 1) % MAX_QUEUE;

  return e;
}
```

Implementación en C (IX)

Obtener primer elemento (sin modificar la cola):

Función `queue_getFront` **en** `queue.c`

```
void *queue_getFront(const Queue *pq) {
  if (pq == NULL || queue_isEmpty(pq) == TRUE) {
    return NULL;
  }

  return pq->data[pq->front];
}
```

Implementación en C (X)

Obtener último elemento (sin modificar la cola):

Función `queue_getBack` **en** `queue.c`

```
void *queue_getBack(const Queue *pq) {
  int last_elem;

  if (pq == NULL || queue_isEmpty(pq) == TRUE) {
    return NULL;
  }

  last_elem = (MAX_QUEUE + pq->rear - 1) % MAX_QUEUE;
  return pq->data[last_elem];
}
```

Implementación en C con *front* y *rear* punteros (I)

EdD para Cola:

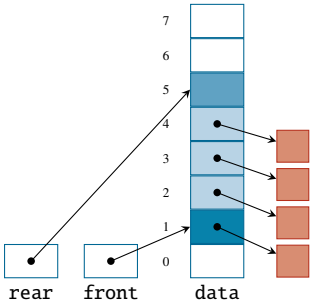
- Implementación con `void **front, **rear;`

Tipo de dato en `queue.h`

```
typedef struct _Queue Queue;
```

Estructura en `queue.c`

```
#define MAX_QUEUE 8

struct _Queue {
  void *data[MAX_QUEUE];
  void **front;
  void **rear;
};
```



Implementación en C con *front* y *rear* punteros (II)

Primitivas:

Cabeceras de las primitivas en `queue.h`

```
Queue *queue_new();
void queue_free(Queue *pq);
Boolean queue_isEmpty(const Queue *pq);
Status queue_push(Queue *pq, const void *e);
void *queue_pop(Queue *pq);
void *queue_getFront(const Queue *pq);
void *queue_getBack(const Queue *pq);
```

La interfaz no cambia

Implementación en C con *front* y *rear* punteros (III)

Crear e inicializar Cola:

Función queue_new **en** queue.c

```
Queue *queue_new() {
  Queue *pq = NULL;
  int i;

  pq = (Queue *)malloc(sizeof(Queue));
  if (pq == NULL) {
    return NULL;
  }

  for (i = 0; i < MAX_QUEUE; i++) {
    pq->data[i] = NULL;
  }
  pq->front = &(pq->data[0]); pq->rear = &(pq->data[0]);
  return pq;
}
```

Implementación en C con *front* y *rear* punteros (IV)

Destrucción de Cola:

Función `queue_free` **en** `queue.c`

```
void queue_free(Queue *pq) {
  free(pq);
}
```

Implementación en C con *front* y *rear* punteros (V)

Comprobar si la cola está vacía:

Función `queue_isEmpty` **en** `queue.c`

```
Boolean queue_isEmpty(const Queue *pq) {
  if (pq == NULL) {
    return TRUE;
  }

  if (pq->front == pq->rear) {
    return TRUE;
  }

  return FALSE;
}
```

Implementación en C con *front* y *rear* punteros (VI)

Comprobar si la cola está llena:

Función `_queue_isFull` **en** `queue.c`

```
Boolean _queue_isFull(const Queue *pq) {
  if (pq == NULL) {
    return TRUE;
  }

  if ((pq->rear + 1 - pq->front) % MAX_QUEUE == 0) {
    return TRUE;
  }

  return FALSE;
}
```

Implementación en C con *front* y *rear* punteros (VII)

Insertar elemento:

Función queue_push **en** queue.c

```c
Status queue_push(Queue *pq, const void *e) {
  if (pq == NULL || e == NULL || _queue_isFull(pq) == TRUE) {
    return ERROR;
  }

  *(pq->rear) = (void *)e;
  pq->rear = pq->data + (pq->rear + 1 - pq->data) % MAX_QUEUE;
  /* Alt:
  pq->rear++; if (pq->rear == pq->data + MAX_QUEUE) pq->rear = pq->data;
  */

  return OK;
}
```

Implementación en C con *front* y *rear* punteros (VIII)

Extraer elemento:

Función queue_pop **en** queue.c

```
void *queue_pop(Queue *pq) {
  void *e = NULL;

  if (pq == NULL || queue_isEmpty(pq) == TRUE) {
    return NULL;
  }

  e = *(pq->front);
  *(pq->front) = NULL;
  pq->front = pq->data + (pq->front + 1 - pq->data) % MAX_QUEUE;
  /* Alt:
  pq->front++; if (pq->front == pq->data + MAX_QUEUE) pq->front = pq->data;
  */

  return e;
}
```

Implementación en C con *front* y *rear* punteros (IX)

Obtener primer elemento (sin modificar la cola):

Función `queue_getFront` **en** `queue.c`

```
void *queue_getFront(const Queue *pq) {
  if (pq == NULL || queue_isEmpty(pq) == TRUE) {
    return NULL;
  }

  return *(pq->front);
}
```

Implementación en C con *front* y *rear* punteros (X)

Obtener último elemento (sin modificar la cola):

Función `queue_getBack` **en** `queue.c`

```
void *queue_getBack(const Queue *pq) {
  void **last_elem;

  if (pq == NULL || queue_isEmpty(pq) == TRUE) {
    return NULL;
  }

  if (pq->rear == pq->data) {
    last_elem = ((Queue *)pq)->data + MAX_QUEUE - 1;
  } else {
    last_elem = pq->rear - 1;
  }
  return *last_elem;
}
```