Queues

Manolis Koubarakis

The ADT Queue

- A **queue** *Q* of items of type *T* is a sequence of items of type *T* on which the following operations are defined:
  - Initialize the queue to the **empty queue**.
  - Determine whether or not the queue is **empty**.
  - Determine whether or not the queue is **full**.
  - Provided *Q* is not full, **insert** a new item onto the rear of the queue.
  - Provided *Q* is nonempty, **remove** an item from the front of *Q*.

The ADT Queue (cont'd)

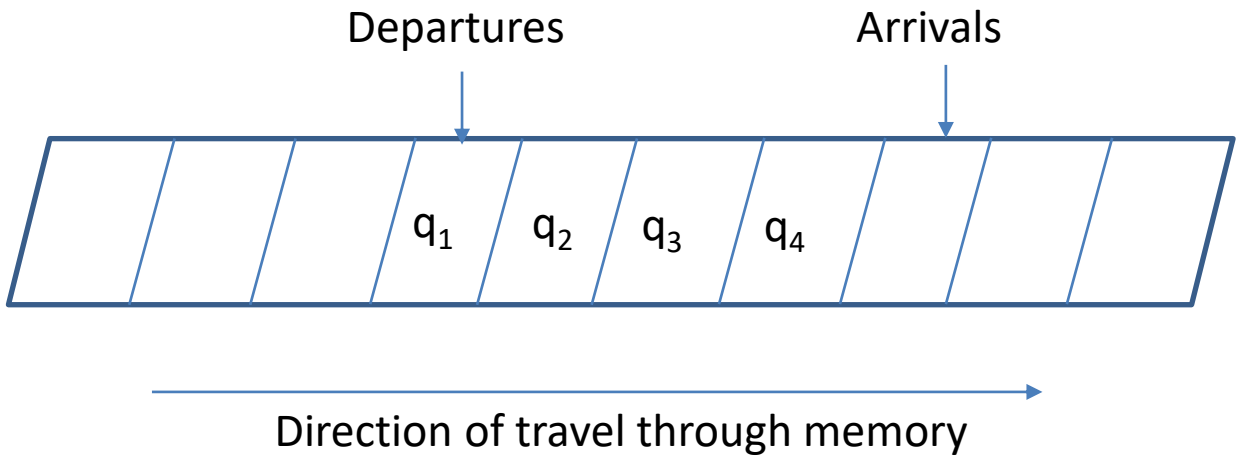
- Queues are also known as **FIFO lists** (first-in first-out).

Queue Representations

- The ADT queue can be implemented using either **sequential** or **linked** representations.

Sequential Queue Representations

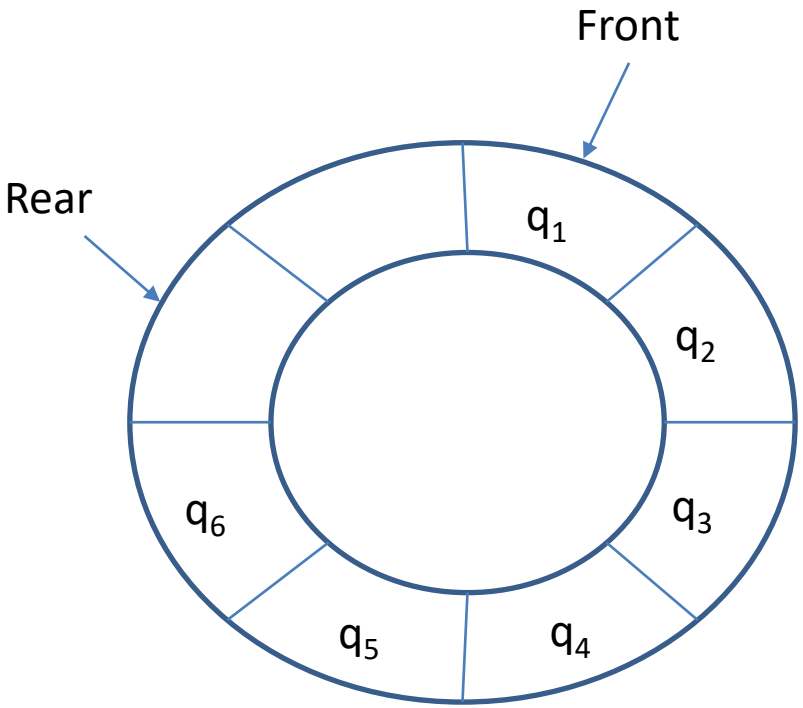
- We can use an **array** as follows:



Sequential Queue Representations (cont'd)

- This representation is **not very handy**.
- The positions of the array to the right will be filled until there is space to do so, while the positions to the left of the array will be freed but we will not be able to use that free space.
- The bounded space representation proposed next is a better one.

Circular Queue Representation



Circular Queue Representation (cont'd)

- If we have an array `Items[0:N-1]` and two pointers `Front` and `Rear` as in the previous figure, then we can use the following assignment statements to increment the pointers so that they always wrap around after falling off the high end of the array.

```
Front=(Front+1)%N
Rear=(Rear+1)%N
```

- The operator `%` computes the **remainder** of the division by `N` so the values of `Front` and `Rear` are always in the range `0` to `N-1`.

Defining the Queue Data Type

```
/* This is the file QueueTypes.h */

#define MAXQUEUESIZE 100

typedef int ItemType;
/* the item type can be arbitrary */

typedef struct {
           int Count;
           int Front;
           int Rear;
           ItemType Items[MAXQUEUESIZE];
         } Queue;
```

The Interface File

```
/* This is the file QueueInterface.h   */

#include “QueueTypes.h”

void InitializeQueue(Queue *Q);
int Empty(Queue *Q);
int Full(Queue *Q);
void Insert(ItemType R, Queue *Q);
void Remove(Queue *Q, ItemType *F);
```

The Implementation

```
/* This is the file QueueImplementation.c */

#include <stdio.h>
#include <stdlib.h>
#include "QueueInterface.h"

void InitializeQueue(Queue *Q)
{
   Q->Count=0;
   Q->Front=0;
   Q->Rear=0;
}
```

The Implementation (cont'd)

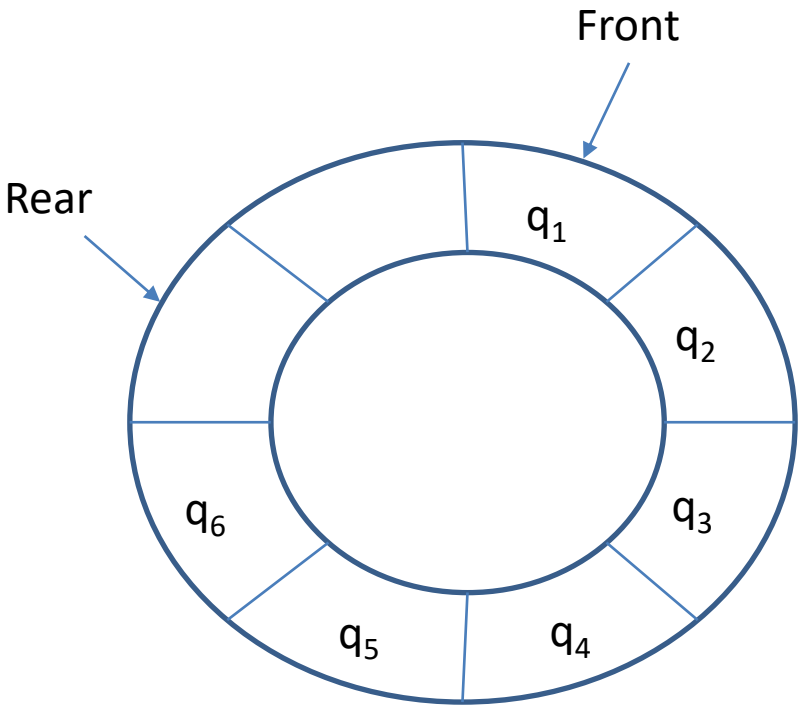
```
int Empty(Queue *Q)
{
    return(Q->Count==0);
}

int Full(Queue *Q)
{
   return(Q->Count==MAXQUEUESIZE);
}
```

The Implementation (cont'd)

```
void Insert(ItemType R, Queue *Q)
{
   if (Q->Count==MAXQUEUESIZE){
      printf("attempt to insert item into a
full queue");
   } else {
      Q->Items[Q->Rear]=R;
      Q->Rear=(Q->Rear+1)%MAXQUEUESIZE;
      ++(Q->Count);
   }
}
```

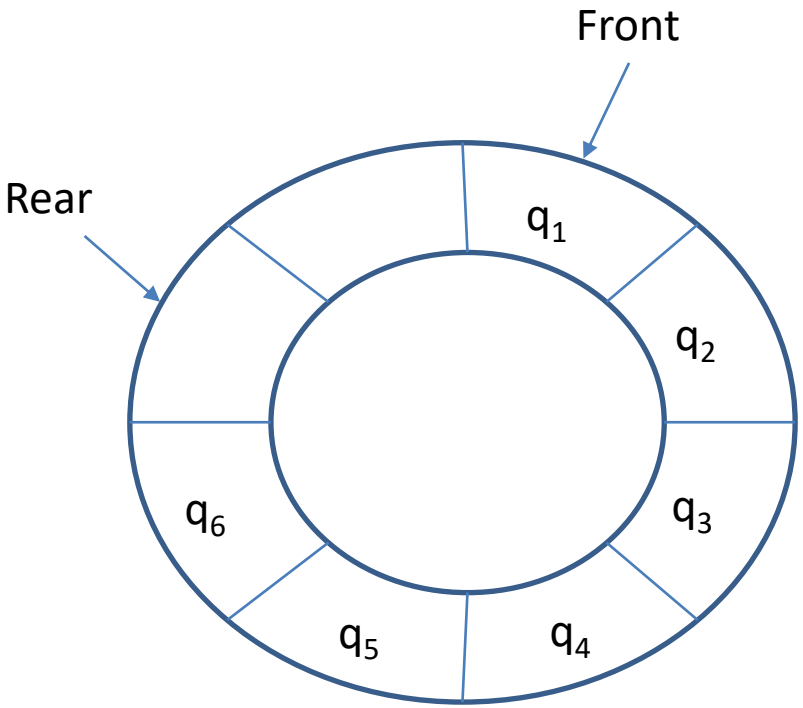
Example



The Implementation (cont'd)

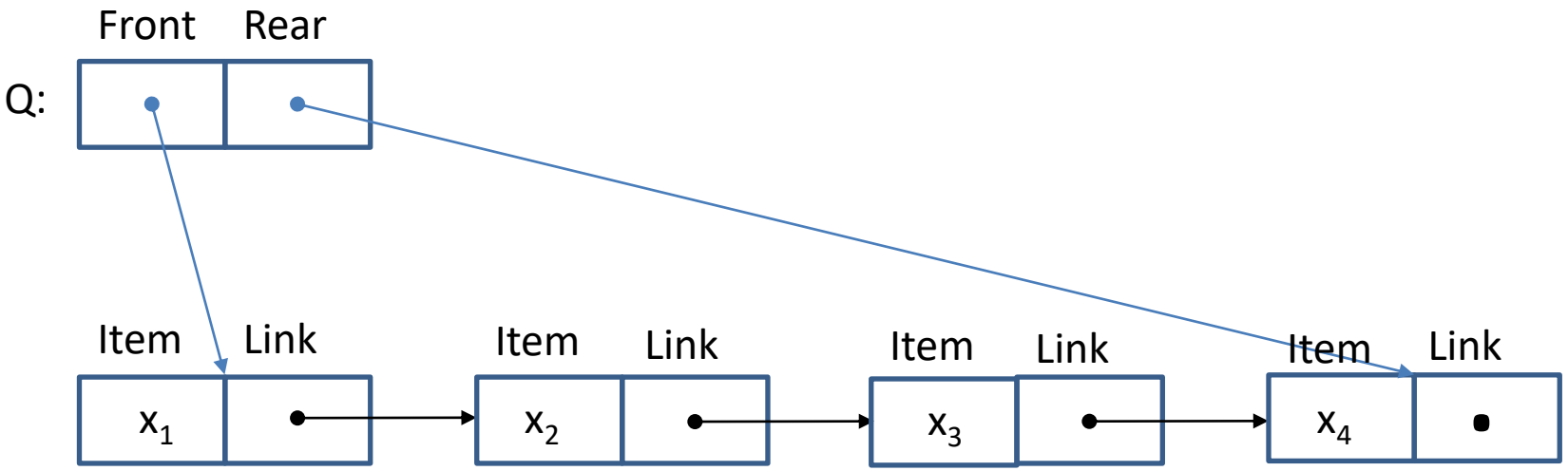
```
void Remove(Queue *Q, ItemType *F)
{
   if (Q->Count==0){
      printf("attempt to remove item from
empty queue");
   } else {
      *F=Q->Items[Q->Front];
      Q->Front=(Q->Front+1)%MAXQUEUESIZE;
      --(Q->Count);
   }
}
```

Example



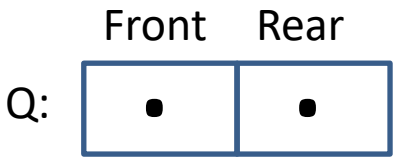
Linked Queue Representation

- In this implementation, we represent a queue by a struct containing pointers to the front and rear of a linked list of nodes.



Linked Queue Representation (cont'd)

- The **empty queue** is a special case and it is represented by a structure whose front and rear pointers are NULL.



Defining the Queue Data Type

```
/* This is the file QueueTypes.h  */

typedef int ItemType;
/* the item type can be arbitrary */

typedef struct QueueNodeTag {
                    ItemType Item;
                    struct QueueNodeTag *Link;
                } QueueNode;

typedef struct {
                    QueueNode *Front;
                    QueueNode *Rear;
                } Queue;
```

The Implementation

```
/* This is the file QueueImplementation.c */

#include <stdio.h>
#include <stdlib.h>
#include "QueueInterface.h"

void InitializeQueue(Queue *Q)
{
    Q->Front=NULL;
    Q->Rear=NULL;
}
```

The Implementation (cont'd)

```
int Empty(Queue *Q)
{
   return(Q->Front==NULL);
}

int Full(Queue *Q)
{
   return(0);
}
/* We assume an already constructed queue */
/* is not full since it can potentially grow */
/* as a linked structure.                    */
```

The Implementation (cont'd)

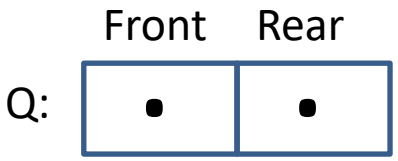
```
void Insert(ItemType R, Queue *Q)
{
   QueueNode *Temp;

   Temp=(QueueNode *)malloc(sizeOf(QueueNode));

   if (Temp==NULL){
      printf("System storage is exhausted");
   } else {
      Temp->Item=R;
      Temp->Link=NULL;
      if (Q->Rear==NULL){ /* this is the case when the queue is empty */
         Q->Front=Temp;
         Q->Rear=Temp;
      } else { /* this is the case when the queue is not empty */
         Q->Rear->Link=Temp;
         Q->Rear=Temp;
      }
   }
}
```

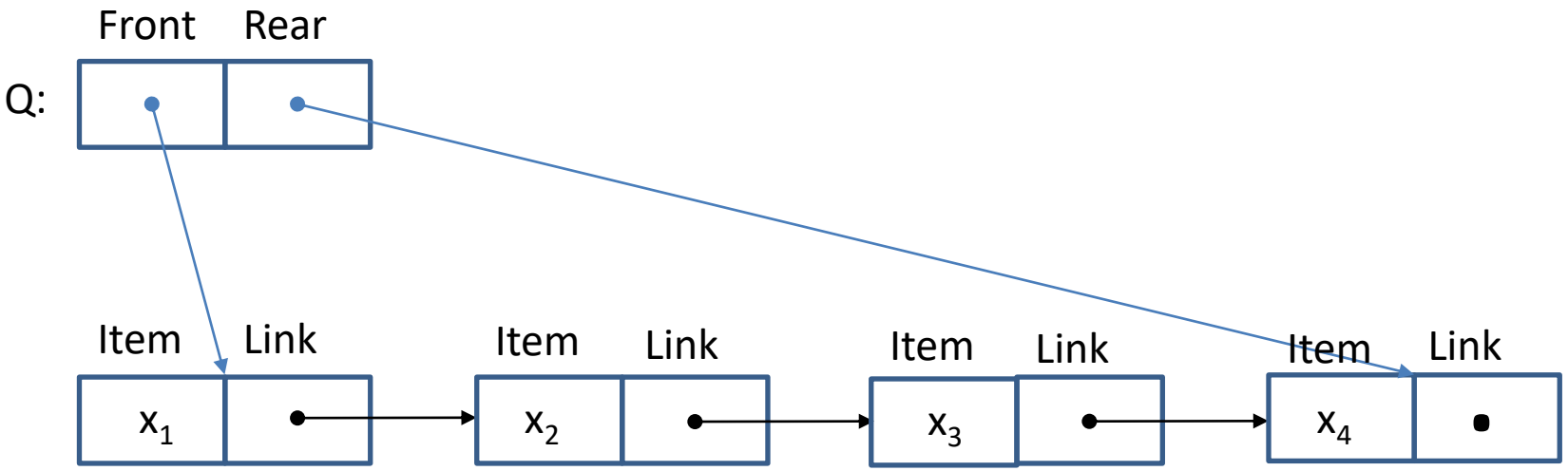
Exercise

- Show visually what happens when we insert a new element in the empty queue below by executing the function `Insert` step by step like we did in the lecture for stacks.



Exercise (cont'd)

- Show visually what happens when we insert a new element in the non-empty queue below by executing the function `Insert` step by step like we did in the lecture for stacks.



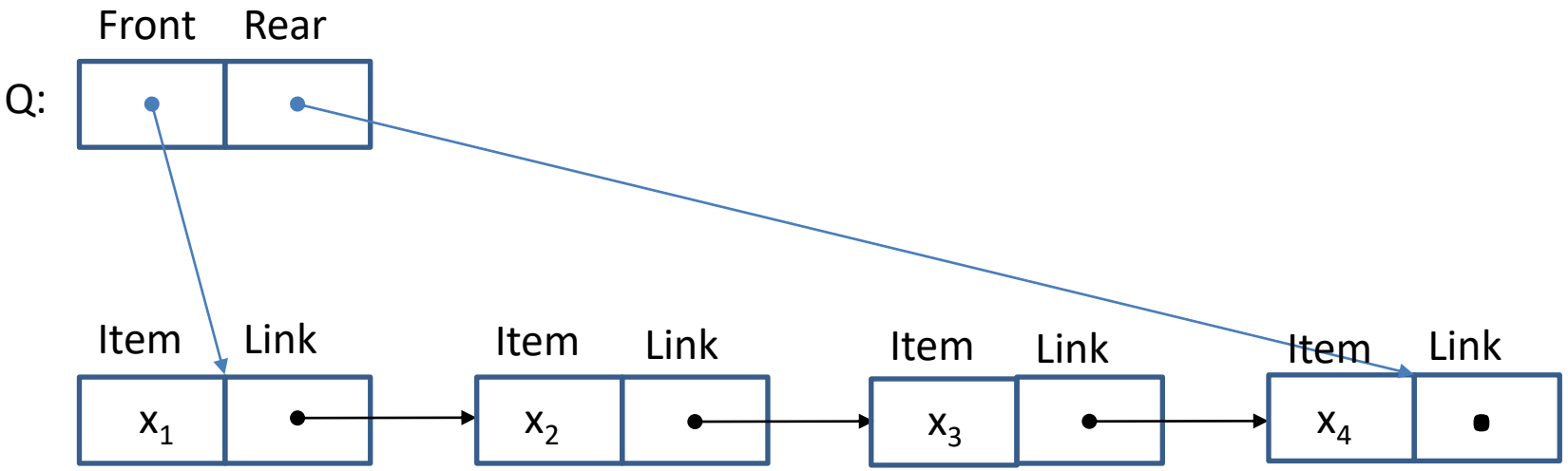
The Implementation (cont'd)

```
void Remove(Queue *Q, ItemType *F)
{
   QueueNode *Temp;


   if (Q->Front==NULL){
      printf("attempt to remove item from an empty queue");
   } else {
      *F=Q->Front->Item;
      Temp=Q->Front;
      Q->Front=Temp->Link;
      free(Temp);
      if (Q->Front==NULL) Q->Rear=NULL; /* this if statement
covers the case when the resulting queue will be empty */
   }
}
```

Exercise (cont'd)

- Show visually what happens when we remove an element from the queue below by executing the function `Remove` step by step like we did in the lecture for stacks.



Example main program

```
#include <stdio.h>
#include <stdlib.h>
#include "QueueInterface.h"


int main(void)
{
   int i,j;
   Queue Q;

   InitializeQueue(&Q);

   for(i=1; i<10; ++i){
      Insert(i, &Q);
   }

   while (!Empty(&Q)){
      Remove(&Q, &j);
      printf("Item %d has been removed.\n", j);
   }

   return 0;
}
```

Comparing Linked and Sequential Queue Representations

- The **sequential queue representation** is appropriate when there is a bound on the number of queue elements at any time.

- The **linked representation** is appropriate when we do not know how large the queue will grow.

Information Hiding Revisited

- The previous definitions and implementations of the ADT queue do not do good information hiding since client programs can get access to the queue representation because the file `QueueTypes.h` is included in the file `QueueInterface.h`.
- We will now give another way to define the ADT queue that does not have this weakness and also has all the nice features of the previous code such as the ability to define multiple queues in a client program.

The Queue ADT Interface

```
typedef struct queue *QPointer;

QPointer QUEUEinit(int maxN);
int QUEUEempty(QPointer);
void QUEUEput(QPointer, Item);
Item QUEUEget(QPointer);
```

In this interface the `typedef` statement defines the type `QPointer` which is a **handle** to a structure for which we only give the name `queue`. The details of this structure are given in the implementation file and, in this way, they are **hidden from client programs**.

The functions of the interface take arguments of type `QPointer`.

The Implementation of the Interface

- Let us now see how we can implement this interface using the linked list representation of a queue that we introduced earlier.
- The front and the rear of the queue are now accessed using pointer variables `head` and `tail`.

The Implementation

```
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"

typedef struct QUEUEnode* link;
struct QUEUEnode { Item item; link next; };
struct queue { link head; link tail; };

link NEW(Item item, link next)
{
    link x = malloc(sizeof *x);
    x->item = item;
    x->next = next;
    return x;
}

QPointer QUEUEinit(int maxN)
{
    QPointer q = malloc(sizeof *q);
    q->head = NULL;
    q->tail = NULL;
    return q;
}
```

The Implementation (cont'd)

```
int QUEUEempty(QPointer q)  { return q->head == NULL; }

void QUEUEput(QPointer q, Item item)
{
    if (q->head == NULL)
    { /* this if statement covers the case when the input queue is empty */
       q->tail = NEW(item, q->head);
       q->head = q->tail;
       return;
    }
    q->tail->next = NEW(item, q->tail->next);
    q->tail = q->tail->next;
}

Item QUEUEget(QPointer q)
{
    Item item = q->head->item;
    link t = q->head->next;
    free(q->head);
    q->head = t;
    return item;
}
```

Notes

- The implementation of queue shown in the previous slides uses an auxiliary function `NEW` to allocate memory for a queue node, set its fields from the function arguments, and return a link to the node.

Exercise

- Show the execution of the functions `QUEUEput` and `QUEUEget` visually using an example queue.

Queue Simulation

- Let us now use the previous queue interface and implementation in a client program.
- The following client program simulates an environment with `M` queues where clients (queue members) are assigned to one of these queues randomly.

The Client Program

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"
#define M 10

main(int argc, char *argv[])
{
  int i, j, N = atoi(argv[1]);
  QPointer queues[M];

  for (i = 0; i < M; i++) queues[i] = QUEUEinit(N);
  for (i = 0; i < N; i++) QUEUEput(queues[rand() % M], i);
  for (i = 0; i < M; i++, printf("\n"))
     for (j = 0; !QUEUEempty(queues[i]); j++)
        printf("%3d ", QUEUEget(queues[i]));
}
```

Information Hiding Revisited

- Notice that the previous client program cannot access the structure that represents the queue because this information is not revealed by the interface file `QUEUE.h`.
- The details are hidden in the implementation which is not accessible to the client.

Using Queues

- **Queues of jobs** are used a lot in operating systems and networks (e.g., a printer queue).
- Queues are also used in **simulation**.
- **Queuing theory** is a branch of mathematics that studies the behaviour of systems with queues.

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.

  Chapter 7.
- R. Sedgewick. Αλγόριθμοι σε C.

  Κεφ. 4.

- As we also said for other lectures, the code that does not do good information hiding is from the first book, while the code that does good information hiding is from the second one.