# Tema 7. Colas

`http://aulavirtual.uji.es`

**José M. Badía, Begoña Martínez, Antonio Morales y José M. Badía**

**{badia, bmartine, morales}@icc.uji.es**

**Estructuras de datos y de la información**

**Universitat Jaume I**

# Índice

# Bibliografía


- (Nyhoff'06), capítulo 8.
- (Orallo'02), capítulo 21.
- *Using the STL, the C++ Standard Template Library*. R. Robson. Springer, Segunda Edición, 1999. Capítulo 9.

# Objetivos

- Conocer el concepto, funcionamiento y aplicaciones de tipo *cola*.
- Conocer el TAD *cola* y sus operaciones asociadas.
- Saber usar el tipo *cola* para resolver problemas.
- Saber implementar el tipo *cola* mediante vectores y nodos enlazados.
- Conocer variantes de *cola* como la *cola con prioridad* y la *bicola*.
- Conocer y saber usar los contenedores `queue`, `priority_queue` y `deque` de la STL.

# 1 Definición y aplicaciones

## Definición

Cola $\Rightarrow$ estructura de datos lineal donde las inserciones y extraciones de datos se realizan siguiendo una política FIFO (*first in first out*). Primer dato en entrar es el primer dato en salir.

## Características

- Homogénea: todos los elementos de la cola son del mismo tipo.
- Existe un orden de elementos ya que es una estructura lineal, pero los elementos no están ordenados por su valor, sino por orden de introducción en la cola.
- Existen dos extremos en la estructura lineal *cola*, el **frente** y el **final** de la cola.
- Sólo se puede acceder y eliminar al dato que está en el frente de la cola.
- Sólo se puede añadir información al final de la cola.

# 1 Definición y aplicaciones (II)

- Operaciones de manipulación:
  - **Añadir**: Añade un elemento al final de la cola.
  - **Eliminar**: Elimina el elemento del frente de la cola.

| | | |
|---|---|---|
| $\Rightarrow$ | X F G A H | $\Rightarrow$ |
| $\Rightarrow$ | X F G A | $\Rightarrow$ |
| $\Rightarrow$ | L X F G A | $\Rightarrow$ |
| $\Rightarrow$ | L X F G | $\Rightarrow$ |
| $\Rightarrow$ | M L X F G | $\Rightarrow$ |
| $\Rightarrow$ | M L X F | $\Rightarrow$ |

**Ejemplos**: cola de clientes esperando pagar en una caja de supermercado, cola de clientes esperando ser atendidos por algún cajero en un banco, cola de procesos esperando ser ejecutados por una CPU.

Al igual que la pila, la cola es una estructura de datos dinámica.

# 2 Tipo Abstracto de Datos *cola*

**TAD** cola

**Usa** logico, tipobase

**Operaciones:**

CrearCola: $\longrightarrow$ cola

ColaVacia: cola $\longrightarrow$ logico

Frente: cola $\longrightarrow$ tipobase

Añadir: cola x tipobase $\longrightarrow$ cola

Eliminar: cola $\longrightarrow$ cola

**Axiomas:** $\forall$ Q $\in$ cola, $\forall$ e $\in$ tipobase,

1) ColaVacia(CrearCola) = verdadero
2) ColaVacia(Añadir(Q,e)) = falso
3) Eliminar(CrearCola) = error
4) Eliminar(Añadir(CrearCola,e)) = CrearCola
5) Eliminar(Añadir(Q,e)) =
   Añadir(Eliminar(Q),e)
6) Frente(CrearCola) = error
7) Frente(Añadir(CrearCola,e)) = e
8) Frente(Añadir(Q,e)) = Frente(Q)

# 3 La clase *queue*

- **`queue`** es una clase definida en la STL.
- Según las nomenclatura de la STL, **`queue`** es un *adaptador*, es decir, un contenedor que se basa en otro contenedor más versátil, y que define nuevas operaciones que son normalmente un subconjunto de las del contenedor en que se basa.
- Un *contenedor* es una clase para "contener" (guardar) una colección de datos según una determinada política de acceso. Por ejemplo, contenedores de la STL son **`vector`**, **`list`**, **`deque`**, **`string`**, entre otros.
- Las clases **`stack`** y **`queue`** son adaptadores. Ambos se basan en el contenedor **`deque`**, del que hablaremos más adelante.
- Otro adaptador de la STL que estudiaremos es **`priority_queue`**.

# 3 La clase *queue* (II)

La clase **queue** será:

```
typedef int tipobase;
class queue {
  public:
    queue();
    bool empty() const;
    const tipobase & front() const;
    void push(const tipobase & x);
    void pop();
  private:
    ...
};
```

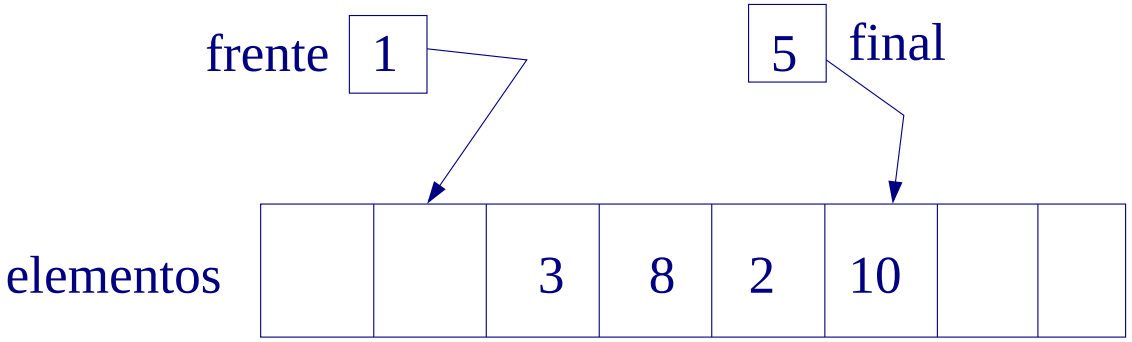
# 3 La clase *queue* (III)

Ejemplo de utilización:

```
#include <queue>
#include <iostream>
using namespace std;
void main() {
  queue q1;
  int i;
  for (i=0; i<10; i++) q1.push(i);
  while (! q1.empty() ) {
      cout << q1.front() << ' ';
      q1.pop();
  }
  cout << endl;
}
```

# 4 Cola con un vector

- Los elementos de la cola se guardan en un vector.
- El frente y el final de la cola son índices del vector.




- **frente** ”apunta” a la posición anterior a donde está el primer elemento de la cola.
- **final** ”apunta” a la posición donde está el último elemento.

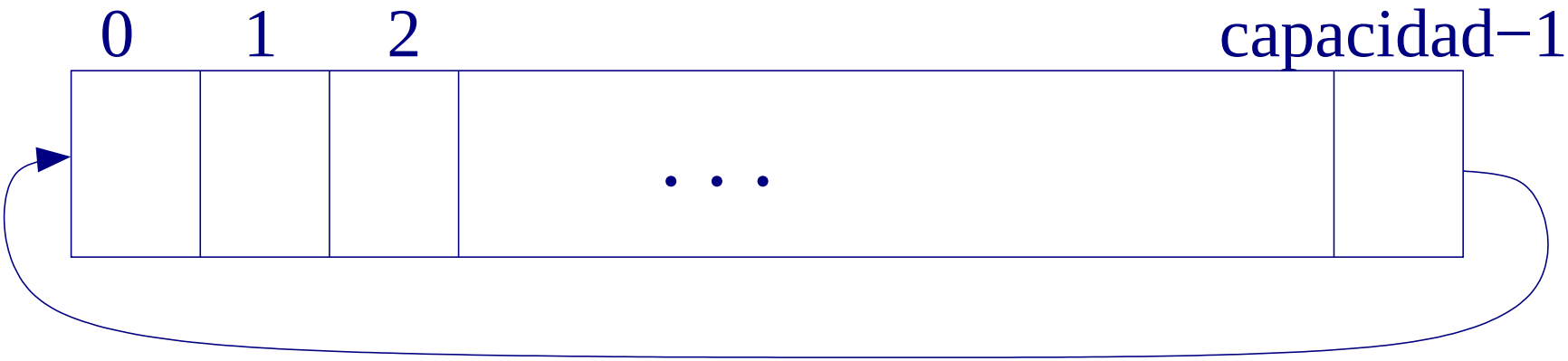
**Primera opción:**

- Vector lineal.
  - Cola llena: `final` ”apunta” a la última posición del vector.
  - Problema: puede desaprovecharse mucho espacio.

# 4 Cola con un vector (II)

**Segunda opción:**

- Vector circular

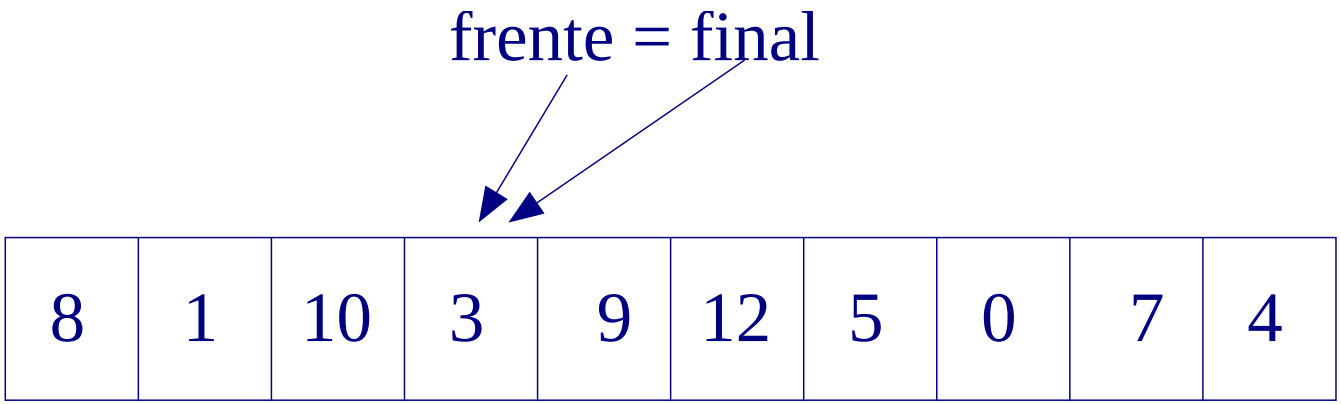



¿Cómo se desplazan los índices **`frente`** y **`final`**?

```
frente = (frente+1) % capacidad;
final = (final+1) % capacidad;
```

La función privada **`int Siguiente(int)`** facilitará los incrementos.

# 4 Cola con un vector (III)




**¿La cola está llena o vacía?**

Para evitar esta ambigüedad se puede:

- Llevar un contador del número de elementos en la cola.
- No dejar que la cola se llene del todo, dejando al menos una posición libre.

Tomaremos esta segunda opción para nuestra implementación, por ser más simple.

# 4 Cola con un vector (IV)

```
typedef int tipobase;
class queue {
public:
  queue(int n=256);
  ~queue();
  queue(const queue &origen); //constructor de copia
  queue & operator=(const queue &origen); // Op. asignación
  bool full() const;
  bool empty() const;
  const tipobase &front() const;
  void push(const tipobase &x);
  void pop();
```

# 4 Cola con un vector (V)

```
private:
  int siguiente(int i) const { return ((i+1) % capacidad); }
  int frente,final;
  T *elementos;
  int capacidad;
};
```

**Tratamiento de errores:**

Es responsabilidad del programador llamar a `empty()` antes de llamar a `front()` o `pop()`, y a `full()` antes de llamar a `push()`.

# 4 Cola con un vector (VI)

```
queue(int n) {
    capacidad = n;
    elementos = new tipobase[capacidad];
    frente = 0; // por ejemplo
    final = 0;  // ha ser igual a frente
}

~queue() {
    delete [] elementos;
}
```

Escribe el código del constructor de copia y del operador de asignación.

# 5 Cola enlazada

- **Cola**: secuencia de nodos enlazados.
- **`frente`** y **`final`** son punteros al primer y último nodo.

Definimos la clase nodo como:

```
class nodo {
public:
  tipobase info;
  nodo *sig;
  // constructor
  nodo(const tipobase &valor=tipobase(), nodo *signodo=NULL):
       info(valor), sig(siguiente) { }
};
```

# 5 Cola enlazada (II)

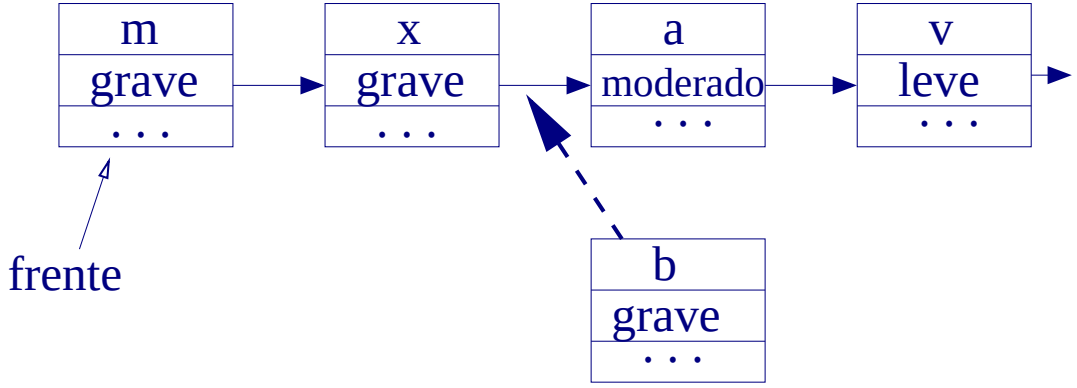
**Clase cola enlazada:**

```
typedef int tipobase;
class queue {
  public:
    // constructores y destructores
    // operaciones: empty, front, push, pop
  private:
    class nodo {
      // datos y constructor
    };
    nodo *frente;
    nodo *final;
};
```

# 6 Colas con prioridad

- **Colas con prioridad:** los elementos tienen asociada una prioridad.
- Comportamiento:
  - Los elementos más prioritarios salen primero.
  - A igual prioridad, política FIFO.

**Ejemplo**: consulta médica. Prioridad: estado del paciente. Implementación:




Implementación:

- Se usan *montículos* (*heaps*): un tipo de árbol binario (tema 10).
- STL: `priority_queue`.

# 6 Colas con prioridad (II)

La interfaz de la clase **`priority_queue`** es similar a la de la clase **`queue`**, pero:

- Se debe proporcionar el operador $<$ del tipo base, en caso de que no esté ya definido.
- Se puede especificar otra función de comparación distinta.

De hecho, la plantilla de **`priority_queue`** está definida como:

```
template <class T, class Sequence=vector<T>,
          class Compare=less<typename Sequence::value_type> >
```

Es decir, la secuencia básica es un vector, pues un vector es muy adecuado para implementar un montículo, como se verá en su momento. Y la función de comparación es la función **`less`**, que a su vez se basa en el operador $<$.

# 6 Colas con prioridad (III)

```
template <class T, Sequence=vector<T>,
          class Compare=less<typename Sequence::value_type> >
class priority_queue {
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c; Compare comp;
public:
    priority_queue();
    priority_queue(const Compare &x);
    bool empty() const;
    size_type size();
    const_reference top();
    void push(const value_type &x);
    void pop();     };//class
```

# 6 Colas con prioridad (IV)

Ejemplo de uso:

```
#include <string>
using namespace std;
class Persona {
  public:
    string nombre; int edad;
    Persona(string nmbr,int ed);
    friend bool operator<(const Persona &p1,const Persona &p2);
    friend bool operator==(const Persona &p1,const Persona &p2);
};
bool operator<(const Persona &p1,const Persona &p2)
    { return (p1.edad < p2.edad); }
bool operator==(const Persona &p1,const Persona &p2)
    { return (p1.edad == p2.edad); }
```

# 6 Colas con prioridad (V)

```
#include <iostream>
#include <queue>
using namespace std;
void main() {
   priority_queue<Persona> pq;
   pq.push(Persona(``Andres'',14));
   pq.push(Persona(``Marta'',7));
   pq.push(Persona(``Isabel'',12));
   pq.push(Persona(``Daniel'',13));
   while (! pq.empty() ) {
      cout << (pq.top()).nombre << "," << (pq.top()).edad << endl;
      pq.pop();
   }
}
```

# 7 Bicolas

- **Bicola:** cola en la que se pueden realizar inserciones y extracciones por ambos lados.
- Bicola en la STL: contenedor **`deque`**. Sobre él están basados los adaptadores **`stack`** y **`queue`**.
- Implementación: **`deque`** permite acceder en tiempo constante a cualquier elemento, como un vector, e implementa una complicada gestión dinámica de memoria que queda fuera del ámbito de la asignatura.

# 7 Bicolas (II)

```
class deque {
public:
    deque();
    ~deque();
    bool empty() const;
    size_type size() const;
    const tipobase &front() const;
    const tipobase &back() const;
    void push_front(const tipobase &x);
    void push_back(const tipobase &x);
    void pop_front();
    void pop_back();
    ...
};
```

# 7 Bicolas (III)

- Las pilas (**`stack`**) y colas (**`queue`**) pueden verse como bicolas (**`deque`**) con restricciones de acceso.
- En la STL, las clases **`stack`** y **`queue`** se implementan usando la clase **`deque`**. Son adaptadores.
- Podrían implementarse en base a otras secuencias: listas, vectores, etc.

# 7 Bicolas (IV)

```
template <class T, class Sequence = deque<T> >
class stack {
   friend bool operator==(const stack &x, const stack &y);
   friend bool operator<(const stack &x, const stack &y);
public:
   typedef typename Sequence::value_type value_type;
   typedef typename Sequence::size_type size_type;
   typedef typename Sequence::reference reference;
   typedef typename Sequence::const_reference const_reference;
protected:
   Sequence c;
public:
   bool empty() const { return c.empty(); }
   size_type size() const { return c.size(); }
   reference top() { return c.back(); }
   const_reference top() const { return c.back(); }
   void push(const value_type &x) { c.push_back(x); }
   void pop() { c.pop_back();}  };//de la clase
```

# 7 Bicolas (V)

```
template <class T, class Sequence = deque<T> >
class queue {
   friend bool operator==(const queue &x, const queue &y);
   friend bool operator<(const queue &x, const queue &y);
public:
   ...
protected:
   Sequence c;
public:
   bool empty() const { return c.empty(); }
   size_type size() const { return c.size(); }
   reference front() { return c.front(); }
   const_reference front() const { return c.front(); }
   reference back() { return c.back(); }
   const_reference back() const { return c.back(); }
   void push(const value_type &x) { c.push_back(x); }
   void pop() { c.pop_front(); }
};
```