

Estructuras de Datos y Algoritmos

TAD cola genérica

LECCIÓN 9

© All wrongs reversed – bajo licencia [CC-BY-NC-SA 4.0](#)



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA



Índice

1 Especificación

2 Implementación estática

- Representación de datos
- Implementación del módulo

3 Implementación dinámica

- Representación de datos
- Implementación del módulo
- Implementación en C++

Índice

1 Especificación

2 Implementación estática

3 Implementación dinámica

TAD cola

Especificación

espec colasGenéricas

usa booleanos, naturales

parámetro formal

género elemento

fpf

género cola

{Los valores del TAD cola representan secuencias de elementos con acceso FIFO (first in, first out), esto es, el primer elemento añadido será el primero en ser borrado}

operaciones

crear: -> cola

{Devuelve una cola vacía, sin elementos}

encolar: cola c, elemento e -> cola

{Devuelve la cola resultante de añadir e a c}

esVacía?: cola c -> booleano

{Devuelve verdad si y sólo si c no tiene elementos}

parcial primero: cola c -> elemento

{Devuelve el primer elemento encolado de los que hay en c.

Parcial: la operación no está definida si c es vacía}

desencolar: cola c -> cola

{Si c es no vacía, devuelve la cola resultante de eliminar de c el primer elemento que fue encolado. En caso contrario, devuelve una cola igual a c}

longitud: cola c -> natural

{Devuelve el número de elementos de c}

fespec

Índice

1 Especificación

2 Implementación estática

- Representación de datos
- Implementación del módulo

3 Implementación dinámica

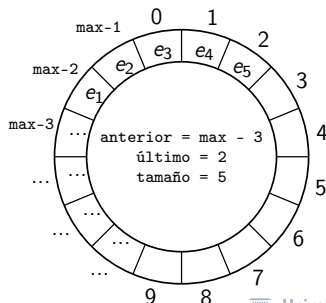
TAD cola

Implementación estática – representación de datos

Representación estática (o contigua)

- Basada en un “vector circular”
- Vector con espacio para un máximo max de elementos
- Núm. elementos válidos en el vector: contador tamaño , $0 \leq \text{tamaño} \leq \text{max}$

```
constante max = ...  
tipos vectorDatos = vector[0.. $\text{max}-1$ ] de elemento;  
    cola = registro  
        datos: vectorDatos;  
        anterior, último: 0.. $\text{max} - 1$ ;  
        actual: 0.. $\text{max} - 1$ ;  
        {actual se usa para el iterador}  
        tamaño: 0.. $\text{max}$   
freg
```



TAD cola

Implementación estática – módulo genérico colas

módulo genérico colasGenéricasEstáticas

parámetro

tipo elemento

exporta

constante max = 100

tipo cola

*{Los valores del TAD cola representan secuencias de 0 o más elementos,
con longitud máxima max; llamamos primer elemento de la cola al primero
que fue añadido y último al último que fue añadido}*

procedimiento crear(sal c: cola)

{Devuelve una cola vacía, sin elementos}

procedimiento encolar(ent e: elemento; e/s c: cola; sal error: booleano)

*{Si c no está llena, añade e a c como último elemento;
si está llena, devuelve error}*

función esVacía(c: cola) devuelve booleano

{Devuelve verdad si y sólo si c no tiene elementos}

procedimiento desencolar(e/s c: cola; sal error: booleano)

*{Si c es no vacía, devuelve la cola resultante de borrar su primer elemento;
si es vacía, devuelve error}*

procedimiento primero(ent c: cola; sal e: elemento; sal error: booleano)

*{Si c es no vacía, devuelve en e su primer elemento;
si es vacía, devuelve error}*

función longitud(c: cola) devuelve natural

{Devuelve el número de elementos de c}

...

...

```
procedimiento duplicar(sal cSal: cola; ent cEnt: cola)
{Duplica la representación de la cola cEnt en la cola cSal}
función iguales(c1, c2: cola) devuelve booleano
{Devuelve verdad si y sólo si las colas c1 y c2 tienen la misma longitud y
  los mismos elementos en idénticas posiciones}

{Añadimos las tres operaciones básicas de un iterador}
procedimiento iniciarIterador(e/s c: cola)
{Prepara el iterador para que el siguiente elemento a visitar sea el
  primero de la cola (situación de no haber visitado ningún elemento)}
función haySiguiente(c: cola) devuelve booleano
{Devuelve falso si ya se ha visitado el último elemento,
  cierto en caso contrario}
procedimiento siguiente(e/s c: cola; sal e: elemento: sal error: booleano)
{Si existe siguiente devuelve en e el siguiente elemento de c y avanza el curso
  si no, devuelve error}
```

...

```

...
implementación
    tipos vectorDatos = vector[0..max - 1] de elemento;
        cola = registro
            datos: vectorDatos;
            anterior, último, actual: 0..max - 1;
                {actual se usa para el iterador}
            tamaño: 0..max

        freg

procedimiento crear(sal c: cola)
principio
    c.anterior := 0; {por ejemplo}
    c.último := 0;
    c.tamaño := 0
fin

procedimiento encolar(e/s c: cola; ent e: elemento; sal error: booleano)
principio
    si c.tamaño < max entonces
        error := falso;
        c.último := (c.último + 1) mod max;
        c.datos[c.último] := e;
        c.tamaño := c.tamaño + 1
    sino
        error := verdad
    fsi
fin
...

```

...

```
función esVacía(c: cola) devuelve booleano
principio
    devuelve c.tamaño = 0
fin
```

```
procedimiento desencolar(e/s c: cola; sal error: booleano)
principio
    si esVacía(c) entonces
        error := verdad
    sino
        error := falso;
        c.anterior := (c.anterior + 1) mod max;
        c.tamaño := c.tamaño - 1
    fsi
fin
```

```
procedimiento primero(ent c: cola; sal e: elemento; sal error: booleano)
principio
    si esVacía(c) entonces
        error := verdad
    sino
        error := falso;
        e := c.datos[(c.anterior + 1) mod max]
    fsi
fin
```

```
función longitud(c: cola) devuelve natural
principio
    devuelve c.tamaño
fin
```

...

```

...
procedimiento duplicar(sal cSal: cola; ent cEnt: cola)
variable i: natural
principio
    si esVacía(cEnt) entonces
        crear(cSal)
    sino
        cSal.tamaño := cEnt.tamaño;
        cSal.anterior := cEnt.anterior;
        cSal.último := cEnt.último;
        para i:=1 hasta cEnt.tamaño hacer
            cSal.datos[(cSal.anterior + i) mod max] := cEnt.datos[(cEnt.anterior + i) mod m]
        fpara
    fsi
fin

función iguales(c1, c2: cola) devuelve booleano
variable i: natural; igual: booleano
principio
    si esVacía(c1) and esVacía(c2) entonces
        devuelve verdad
    sino_si longitud(c1) ≠ longitud(c2) entonces
        devuelve falso
    sino
        i := 1;
        igual := verdad;
        mientrasQue igual and i ≤ longitud(c1) hacer
            igual := c1.datos[(c1.anterior + i) mod max] = c2.datos[(c2.anterior + i) mod m]
            i := i + 1
        fmq;
        devuelve igual
    fsi
fin
...

```

```

...
procedimiento iniciarIterador(e/s c: cola)
principio
    c.actual := (c.anterior + 1) mod max
fin

función haySiguiente(c: cola) devuelve booleano
principio
    devuelve c.actual  $\neq$  (c.último+1) mod max
fin

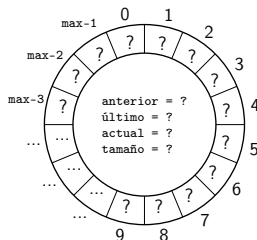
procedimiento siguiente(e/s c: cola; sal e: elemento: sal error: booleano)
principio
    si haySiguiente(c) entonces
        error := falso;
        e := c.datos[c.actual];
        c.actual := (c.actual + 1) mod max
    sino
        error := verdad
    fsi
fin
fin {del módulo}

```

TAD cola

Implementación estática – ejemplo

```
c: colaEstaticaDeEnteros;  
error: booleano;  
e: entero;
```

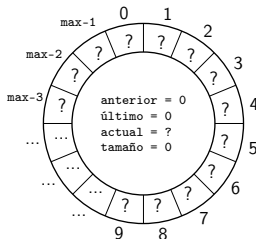


TAD cola

Implementación estática – ejemplo

```
c: colaEstaticaDeEnteros;  
error: booleano;  
e: entero;
```

```
crear(c);
```



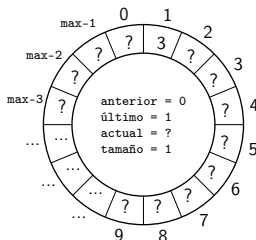
```
procedimiento crear(sal c: cola)  
principio  
    c.anterior := 0; {por ejemplo}  
    c.último := 0;  
    c.tamaño := 0  
fin
```

TAD cola

Implementación estática – ejemplo

```
c: colaEstaticaDeEnteros;  
error: booleano;  
e: entero;
```

```
crear(c);  
encolar(c, 3, error);
```



```
procedimiento encolar(e/s c:cola; ent e:elemento; sal error:booleano)  
principio
```

```
    si c.tamaño < max entonces  
        error := falso;  
        c.último := (c.último + 1) mod max;  
        c.datos[c.último] := e;  
        c.tamaño := c.tamaño + 1
```

```
    sino  
        error := verdad
```

```
    fsi
```

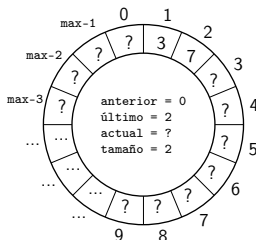
```
fin
```

TAD cola

Implementación estática – ejemplo

```
c: colaEstaticaDeEnteros;  
error: booleano;  
e: entero;
```

```
crear(c);  
encolar(c, 3, error);  
encolar(c, 7, error);
```



```
procedimiento encolar(e/s c:cola; ent e:elemento; sal error:booleano)  
principio
```

```
    si c.tamaño < max entonces  
        error := falso;  
        c.último := (c.último + 1) mod max;  
        c.datos[c.último] := e;  
        c.tamaño := c.tamaño + 1
```

```
    sino  
        error := verdad
```

```
    fsi
```

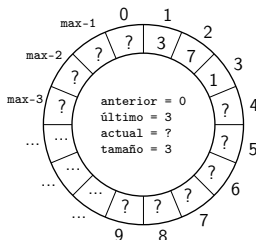
```
fin
```

TAD cola

Implementación estática – ejemplo

```
c: colaEstaticaDeEnteros;  
error: booleano;  
e: entero;
```

```
crear(c);  
encolar(c, 3, error);  
encolar(c, 7, error);  
encolar(c, 1, error);
```



```
procedimiento encolar(e/s c:cola; ent e:elemento; sal error:booleano)  
principio
```

```
    si c.tamaño < max entonces  
        error := falso;  
        c.último := (c.último + 1) mod max;  
        c.datos[c.último] := e;  
        c.tamaño := c.tamaño + 1
```

```
    sino  
        error := verdad
```

```
    fsi
```

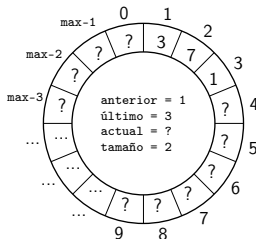
```
fin
```

TAD cola

Implementación estática – ejemplo

```
c: colaEstaticaDeEnteros;  
error: booleano;  
e: entero;
```

```
crear(c);  
encolar(c, 3, error);  
encolar(c, 7, error);  
encolar(c, 1, error);  
desencolar(c, error);
```



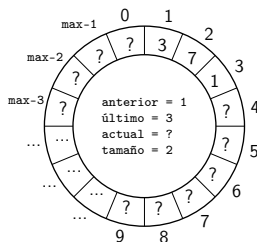
```
procedimiento desencolar(e/s c: cola; sal error: booleano)  
principio  
  si esVacía(c) entonces  
    error := verdad  
  sino  
    error := falso;  
    c.anterior := (c.anterior + 1) mod max;  
    c.tamaño := c.tamaño - 1  
  fsi  
fin
```

TAD cola

Implementación estática – ejemplo

```
c: colaEstaticaDeEnteros;  
error: booleano;  
e: entero;
```

```
crear(c);  
encolar(c, 3, error);  
encolar(c, 7, error);  
encolar(c, 1, error);  
desencolar(c, error);  
primero(c, e, error);
```



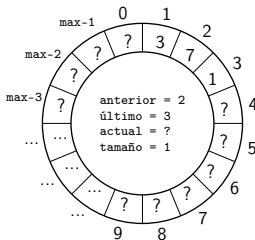
```
procedimiento primero(ent c: cola; sal e: elemento; sal error: booleano)  
principio  
    si esVacía(c) entonces  
        error := verdad  
    sino  
        error := falso;  
        e := c.datos[(c.anterior + 1) mod max]  
    fsi  
fin
```

TAD cola

Implementación estática – ejemplo

```
c: colaEstaticaDeEnteros;  
error: booleano;  
e: entero;
```

```
crear(c);  
encolar(c, 3, error);  
encolar(c, 7, error);  
encolar(c, 1, error);  
desencolar(c, error);  
primero(c, e, error);  
desencolar(c, error);
```



```
procedimiento desencolar(e/s c: cola; sal error: booleano)
```

```
principio
```

```
    si esVacía(c) entonces  
        error := verdad
```

```
    sino
```

```
        error := falso;  
        c.anterior := (c.anterior + 1) mod max;  
        c.tamaño := c.tamaño - 1
```

```
    fsi
```

```
fin
```

Índice

1 Especificación

2 Implementación estática

3 Implementación dinámica

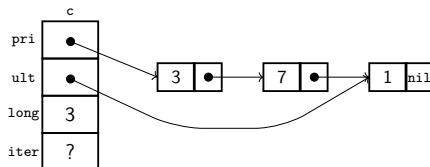
- Representación de datos
- Implementación del módulo
- Implementación en C++

TAD cola

Implementación dinámica – representación de datos

```
tipos ptDato = ↑unDato;  
      unDato = registro  
              dato: elemento;  
              sig: ptDato;  
      freg;  
cola = registro  
      pri, ult: ptDato;  
      long: natural;  
      iter: ptDato; {se utiliza para implementar el iterador}  
      freg
```

Sea c una variable de tipo cola de enteros:



TAD colas

Implementación estática – módulo genérico colas

módulo genérico colasGenéricasEstáticas

parámetro

tipo elemento

exporta

tipo cola

{Los valores del TAD cola representan secuencias de 0 o más elementos, con longitud máxima max; llamamos primer elemento de la cola al primer que fue añadido y último al último que fue añadido}

procedimiento crearVacía(sal c: cola)

{Devuelve en c la cola vacía, sin elementos}

procedimiento encolar(e/s c: cola; ent e: elemento)

{Devuelve en c la cola resultante de añadir e a c}

función esVacía(c: cola) devuelve booleano

{Devuelve verdad si y sólo si c no tiene elementos}

procedimiento primero(ent c: cola; sal e: elemento; sal error: booleano)

{Si c es no vacía, devuelve en e el primer elemento añadido a c y error=falso. Si c es vacía, devuelve error=verdad y e queda indefinido}

procedimiento desencolar(e/s c:cola)

{Si c es no vacía, devuelve en c la cola resultante de eliminar de c el primer elemento que fue añadido. Si c es vacía, la deja igual}

...

...

```
función longitud(c: cola) devuelve natural
{Devuelve el número de elementos de c}
procedimiento duplicar(sal colaSal: cola; ent colaEnt: cola)
{Devuelve en colaSal una cola igual a colaEnt, duplicando la
  representación en memoria}
función iguales(cola1, cola2: cola) devuelve booleano
{Devuelve verdad si y sólo si cola1 y cola2 tienen los mismos elementos
  y en las mismas posiciones}
procedimiento liberar(e/s c: cola)
{Devuelve en c la cola vacía y además libera la memoria utilizada
  previamente por c}

{Las tres operaciones siguientes conforman un iterador interno para la cola}
procedimiento iniciarIterador(e/s c: cola)
{Prepara el iterador para que el siguiente elemento a visitar sea un
  primer elemento de c, si existe (situación de no haber visitado
  ningún elemento)}
función existeSiguiete(c: cola) devuelve booleano
{Devuelve falso si ya se han visitado todos los elementos de c;
  devuelve cierto en caso contrario}
procedimiento siguiente(e/s c: cola; sal e: elemento; sal error: booleano)
{Si existe algún elemento de c pendiente de visitar, devuelve en e el
  siguiente elemento a visitar y error=falso, y además avanza el iterad
  para que a continuación se pueda visitar otro elemento de c.
  Si no quedan elementos pendientes de visitar devuelve error=verdad
  y e queda indefinido}
```

...

```

...
implementación
    tipos ptDato = ↑unDato;
        unDato = registro
            dato: elemento;
            sig: ptDato
        freg;
    cola = registro
        pri, ult: ptDato;
        long: natural;
        iter: ptDato {se utiliza para implementar el iterador}
        freg

procedimiento crearVacía(sal c: cola)
principio
    c.pri := nil;
    c.ult := nil;
    c.long := 0

fin

procedimiento encolar(e/s c: cola; ent e: elemento)
principio
    si c.long = 0 entonces
        nuevoDato(c.ult);
        c.pri := c.ult
    sino
        nuevoDato(c.ult↑.sig);
        c.ult := c.ult↑.sig
    fsi;
    c.ult↑.dato := e;
    c.ult↑.sig := nil;
    c.long := c.long + 1

fin
...

```

```

...
función esVacía(c: cola) devuelve booleano
principio
    devuelve c.pri = nil
fin

procedimiento primero(ent c: cola; sal e: elemento; sal error: booleano)
principio
    si esVacía(c) entonces
        error := verdad
    sino
        error := falso;
        e := c.pri↑.dato
    fsi
fin

procedimiento desencolar(e/s c: cola)
variable aux: ptDato
principio
    si not esVacía(c) entonces
        aux := c.pri;
        c.pri := c.pri↑.sig;
        disponer(aux);
        c.long := c.long - 1;
        si c.long = 0 entonces c.ult := nil fsi
    fsi
fin

función longitud(c: cola) devuelve natural
principio
    devuelve c.long
fin
...

```

```

...
procedimiento duplicar(sal colaSal: cola; ent colaEnt: cola)
variables ptSal, ptEnt: ptDato
principio
    si esVacía(colEnt) entonces
        crearVacía(colSal);
    sino
        ptEnt := colaEnt.pri;
        nuevoDato(colSal.pri);
        colaSal.pri↑.dato := ptEnt↑.dato;
        ptSal := colaSal.pri;
        ptEnt := ptEnt↑.sig;
        mientrasQue ptEnt ≠ nil hacer
            nuevoDato(ptSal↑.sig);
            ptSal := ptSal↑.sig;
            ptSal↑.dato := ptEnt↑.dato;
            ptEnt := ptEnt↑.sig
        fmq;
        ptSal↑.sig := nil;
        colaSal.ult := ptSal;
        colaSal.long := colEnt.long
    fsi
fin

función iguales(col1, col2: cola) devuelve booleano
variables pt1, pt2: ptDato; iguales: booleano := verdad
principio
    si col1.long ≠ col2.long entonces
        devuelve falso;
    sino
        pt1 := col1.pri;
        pt2 := col2.pri;
        mientrasQue pt1 ≠ nil and iguales hacer
            iguales := pt1↑.dato = pt2↑.dato;
            pt1 := pt1↑.sig;
            pt2 := pt2↑.sig
        fmq;
        devuelve iguales
    fsi
fin
...

```

```

...
procedimiento liberar(e/s c: cola)
variable aux: ptData
principio
    aux := c.pri;
    mientrasQue aux ≠ nil hacer
        c.pri := c.pri↑.sig;
        disponer(aux);
        aux := c.pri
    fmq;
    c.ult := nil;
    c.long := 0
fin

procedimiento iniciarIterador(e/s c: cola)
principio
    c.iter := c.pri
fin

función existeSiguiente(c: cola) devuelve booleano
principio
    devuelve c.iter ≠ nil
fin

procedimiento siguiente(e/s c: cola; sal e: elemento; sal error: booleano)
principio
    si existeSiguiente(c) entonces
        error := falso;
        e := c.iter↑.dato;
        c.iter := c.iter↑.sig
    sino
        error := verdad
    fsi
fin

fin {del módulo}

```

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

c	
pri	?
ult	?
long	?
iter	?

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

```
crearVacía(c);
```

c	
pri	nil
ult	nil
long	0
iter	?

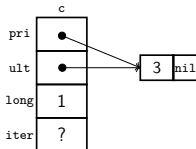
```
procedimiento crearVacía(sal c: cola)  
principio  
    c.pri := nil;  
    c.ult := nil;  
    c.long := 0  
fin
```

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

```
crearVacía(c);  
encolar(c, 3);
```



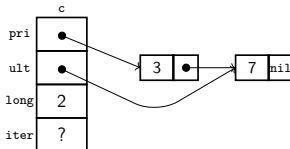
```
procedimiento encolar(e/s c: cola; ent e: elemento)  
principio  
  si c.long = 0 entonces  
    nuevoDato(c.ult);  
    c.pri := c.ult  
  sino  
    nuevoDato(c.ult↑.sig);  
    c.ult := c.ult↑.sig  
fisi;  
c.ult↑.dato := e;  
c.ult↑.sig := nil;  
c.long := c.long + 1  
fin
```

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

```
crearVacía(c);  
encolar(c, 3);  
encolar(c, 7);
```



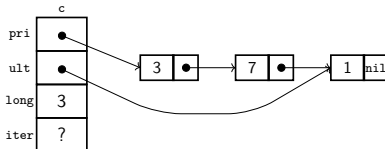
```
procedimiento encolar(e/s c: cola; ent e: elemento)
principio
    si c.long = 0 entonces
        nuevoDato(c.ult);
        c.pri := c.ult
    sino
        nuevoDato(c.ult↑.sig);
        c.ult := c.ult↑.sig
    fsi;
    c.ult↑.dato := e;
    c.ult↑.sig := nil;
    c.long := c.long + 1
fin
```

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

```
crearVacía(c);  
encolar(c, 3);  
encolar(c, 7);  
encolar(c, 1);
```



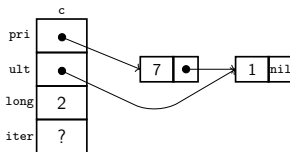
```
procedimiento encolar(e/s c: cola; ent e: elemento)
principio
    si c.long = 0 entonces
        nuevoDato(c.ult);
        c.pri := c.ult
    sino
        nuevoDato(c.ult↑.sig);
        c.ult := c.ult↑.sig
    fsi;
    c.ult↑.dato := e;
    c.ult↑.sig := nil;
    c.long := c.long + 1
fin
```

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

```
crearVacía(c);  
encolar(c, 3);  
encolar(c, 7);  
encolar(c, 1);  
desencolar(c);
```



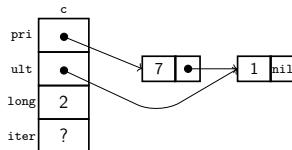
```
procedimiento desencolar(e/s c: cola)
variable aux: pDatao
principio
    si not esVacía(c) entonces
        aux := c.pri;
        c.pri := c.pri↑.sig;
        disponer(aux);
        c.long := c.long - 1;
        si c.long = 0 entonces c.ult := nil fsi
    fsi
fin
```

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

```
crearVacía(c);  
encolar(c, 3);  
encolar(c, 7);  
encolar(c, 1);  
desencolar(c);  
primero(c, e, error);
```



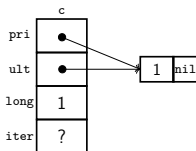
```
procedimiento primero(ent c: cola; sal e: elemento; sal error: booleano)
principio
    si esVacía(c) entonces
        error := verdad
    sino
        error := falso;
        e := c.pri↑.dato
    fsi
fin
```

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

```
crearVacía(c);  
encolar(c, 3);  
encolar(c, 7);  
encolar(c, 1);  
desencolar(c);  
primero(c, e, error);  
desencolar(c);
```



```
procedimiento desencolar(e/s c: cola)  
variable aux: ptDato  
principio  
    si not esVacía(c) entonces  
        aux := c.pri;  
        c.pri := c.pri↑.sig;  
        disponer(aux);  
        c.long := c.long - 1;  
        si c.long = 0 entonces c.ult := nil fsi  
    fsi  
fin
```

TAD cola

Implementación dinámica – ejemplo

```
c: colaDinamicaDeEnteros;  
error: booleano;  
e: entero;
```

```
crearVacía(c);  
encolar(c, 3);  
encolar(c, 7);  
encolar(c, 1);  
desencolar(c);  
primero(c, e, error);  
desencolar(c);  
desencolar(c);
```

c	
pri	nil
ult	nil
long	0
iter	?

```
procedimiento desencolar(e/s c: cola)  
variable aux: ptDato  
principio  
  si not esVacía(c) entonces  
    aux := c.pri;  
    c.pri := c.pri↑.sig;  
    disponer(aux);  
    c.long := c.long - 1;  
    si c.long = 0 entonces c.ult := nil fsi  
  fsi  
fin
```

TAD cola

Implementación dinámica en C++

// Interfaz del TAD. Pre-declaraciones:

```
template <typename Elemento> struct Cola;

template <typename Elemento> void vacia(Cola<Elemento>& c);
template <typename Elemento> void encolar(Cola<Elemento>& c,
                                           const Elemento& dato);

template <typename Elemento> void desencolar(Cola<Elemento>& c);
template <typename Elemento> void primero(const Cola<Elemento>& c,
                                           Elemento& dato, bool& error);

template <typename Elemento> bool esVacia(const Cola<Elemento>& c);
template <typename Elemento> int longitud(const Cola<Elemento>& c);
template <typename Elemento> void duplicar(const Cola<Elemento>& cOrigen,
                                           Cola<Elemento>& cDestino);

template <typename Elemento> bool operator== (const Cola<Elemento>& c1,
                                              const Cola<Elemento>& c2);

template <typename Elemento> void liberar(Cola<Elemento>& c);
template <typename Elemento> void iniciarIterador(Cola<Elemento>& c);
template <typename Elemento> bool existeSiguiente(const Cola<Elemento>& c);
template <typename Elemento> bool siguiente(Cola<Elemento>& c, Elemento& dato);
...
```

```

...
// Declaración

template <typename Elemento> struct Cola{
    friend void vacia<Elemento>(Cola<Elemento>& c);
    friend void encolar<Elemento>(Cola<Elemento>& c,
                                   const Elemento& dato);
    friend void desencolar<Elemento>(Cola<Elemento>& c);
    friend void primero<Elemento>(const Cola<Elemento>& c,
                                   Elemento& dato, bool& error);
    friend bool esVacia<Elemento>(const Cola<Elemento>& c);
    friend int longitud<Elemento>(const Cola<Elemento>& c);
    friend void duplicar<Elemento>(const Cola<Elemento>& cOrigen,
                                   Cola<Elemento>& cDestino);
    friend bool operator==<Elemento> (const Cola<Elemento>& c1,
                                       const Cola<Elemento>& c2);
    friend void liberar<Elemento>(Cola<Elemento>& c);
    friend void iniciarIterador<Elemento>(Cola<Elemento>& c);
    friend bool existeSiguiente<Elemento>(const Cola<Elemento>& c);
    friend bool siguiente<Elemento>(Cola<Elemento>& c, Elemento& dato);
...

```

```

...
// Representación de los valores del TAD
private:
    struct unDato {
        Elemento dato;
        unDato* sig;
    };

    unDato* pri;
    unDato* ult;
    int longi;
    unDato* iter;
};

// Implementación de las operaciones
template<typename Elemento> void vacia(Cola<Elemento>& c) {
    c.longi = 0;
    c.pri = nullptr;
    c.ult = nullptr;
}

template <typename Elemento> void encolar(Cola<Elemento>& c, const Elemento& e) {
    if (c.longi == 0) {
        c.ult = new typename Cola<Elemento>::unDato;
        c.pri = c.ult;
    } else
        c.ult -> sig = new typename Cola<Elemento>::unDato;
        c.ult = c.ult -> sig;
    }
    c.ult -> dato = e;
    c.ult -> sig = nullptr;
    c.longi = c.longi + 1;
}

// y más operaciones...
...

```

[ver implementación completa en el material de clase]

TAD cola

El método en acción: desencolar (implementación dinámica)

- 1 **Operación y tamaño:** `desencolar(c)`; sea n el número de elementos de la cola c
- 2 **Caso:** peor caso
- 3 **Descomposición:** comprobar si la cola está vacía; guardar el puntero al primer registro; avanzar el puntero primero; si la cola queda vacía, actualizar último; liberar el registro
- 4 **Coste local:** las cinco sub-operaciones son $\mathcal{O}(1)$
- 5 **Composición:** secuencia $\Rightarrow$ suma de costes constantes
- 6 **Resultado:** $\mathcal{O}(1)$ en caso peor, independiente de n

Método de análisis de coste en 6 pasos (lección 8): **rutina del curso** al cerrar cada implementación

TAD cola

El método en acción: encolar, ¿por qué un puntero al último?

- Con puntero al **último**: enlazar el nuevo registro tras el último y actualizar el puntero $\Rightarrow \mathcal{O}(1)$
- **Sin** puntero al último: habría que *recorrer* la cola desde primero hasta el final $\Rightarrow \mathcal{O}(n)$
- Moraleja **F4**: un **campo extra en la representación** (un puntero, un contador) puede cambiar el orden de una operación
 - ...pero hay que **mantenerlo** en *todas* las operaciones que lo afectan (**F2**: cardinales y punteros sin mantener)

Tabla de decisión: estructura $\leftrightarrow$ requisito

Estructura	Coste de las operaciones clave	Elegirla cuando...	Antipatrón
Pila	apilar, desapilar, cima: $O(1)$	solo importa el último en llegar (LIFO): deshacer, llamadas, delimitadores	buscar o recorrer «dentro» de una pila
Cola	encolar, desencolar, primero: $O(1)$	importa el orden de llegada (FIFO): turnos, esperas	usarla cuando se necesita acceso por clave
Lista	insertar/borrar en el punto de interés: $O(1)$; buscar: $O(n)$	colección recorrible con orden posicional	lista donde el requisito pide búsqueda eficiente
Diccionario	según implementación ($\rightarrow$ ABB, AVL, trie, hash)	acceso por clave	fixar la implementación sin leer el requisito de coste

La tabla se irá completando a lo largo del curso: cópiala y ve rellenándola tú también.

Estructuras de Datos y Algoritmos

TAD cola genérica

LECCIÓN 9

© All wrongs reversed – bajo licencia [CC-BY-NC-SA 4.0](#)



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA

