Los TAD's pila y cola. Especificación e implementaciones

Lección 9

Concepto de Pila

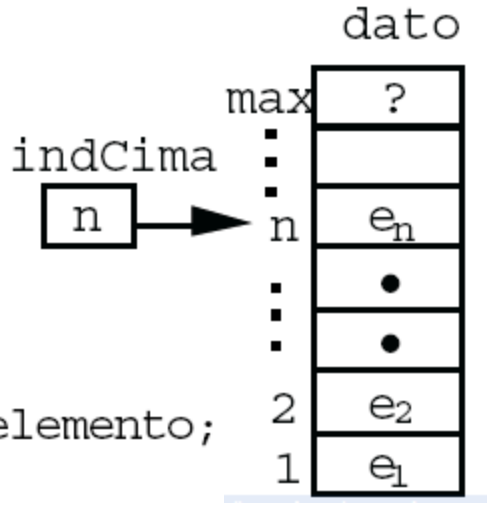
- Pila:
  - Secuencia de elementos de un cierto tipo dispuestos en una dimensión (***tipo lineal*** de datos)
  - Valor especial: ***pila vacía***
  - Operaciones: **apilar** *(añadir elemento)*, **desapilar** *(quitar elemento)*, **cima** *(observar último elemento)*
  - Comportamiento: operaciones limitadas a un extremo (*cima*)
    - **Cima** de pila no vacía es el último elemento que fue apilado
    - **Apilar** añade un elemento "encima" del que estaba en la cima
    - **Desapilar**, de pila no vacía, elimina el último elemento apilado
  - Denominadas también estructuras ***LIFO*** *(Last In, First Out)*

Especificación de pilas genéricas

espec pilasGenéricas
usa booleanos, naturales
parámetro formal
género elemento
fpf
género pila *{Los valores del TAD pila representan secuencias de elementos con acceso LIFO (last in, first out), esto es, el último elemento en ser añadido (entrar) será el primero en ser borrado (salir)}*
operaciones
crear: -> pila
{Devuelve una pila vacía, sin elementos}
apilar: pila p , elemento e → pila
{Devuelve la pila igual a la resultante de añadir e a p}
esVacía?: pila p → booleano
{Devuelve verdad si y sólo si p no tiene elementos}
parcial cima: pila p → elemento
{Devuelve el último elemento apilado en p.
Parcial: la operación no está definida si p es vacía}
desapilar: pila p → pila
{Si p es no vacía, devuelve la pila igual a la resultante de eliminar de p el último elemento que fue apilado. En caso contrario, devuelve una pila igual a p}
altura: pila p → natural
{Devuelve el nº de elementos de p}
fespec

Representación estática del TAD pila

- Implementación estática:




```
constante max = ...
tipo pila = registro
              dato: vector[1..max] de elemento;
              indCima: 0..max
            freg
```

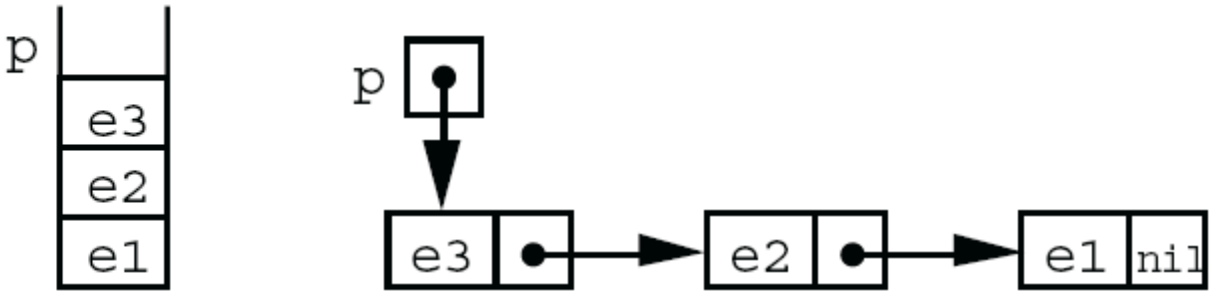
- Costes:

  Todas las operaciones Θ(1) en tiempo

- Inconvenientes:
  - *Coste espacio en memoria **Θ(MAX)** por el **max** de elementos previstos*
  - *Operación apilar debe implementarse como parcial*
  - *Se necesitan operaciones de **igualdad** y **asignar/duplicar** entre pilas*

Representación Dinámica del TAD Pila

- Representación por *encadenamiento mediante punteros* de los elementos de la pila
  - la pila será un puntero que apunta al elemento que está en la cima de la pila, y cada elemento apuntará al anterior
  - Ejemplos:
    - Pila con tres elementos



    - Pila vacía: p 

Implementación Dinámica del TAD Pila

- Características de la implementación:
  - En esta implementación, todas las operaciones de la especificación del TAD (VACIA, APILAR, DESAPILAR, CIMA, VACIA?, *ALTURA*(*)) tienen tiempo de ejecución de ***Θ(1)***

    (*) Siendo pila registro con puntero a cima y contador de elementos o altura
  - La pila puede crecer hasta ocupar el máximo de la memoria (dinámica) disponible
  - Coste de memoria ***Θ(N)***:
    - Coste "extra" para almacenar los punteros (encadenar datos apilados)
    - Coste ajustado al necesario para almacenar los datos actuales de la pila
  - *Se necesitará* redefinir las operaciones de ***duplicar/copiar*** e ***igualdad***
  - Se necesita una operación para ***liberar*** la memoria ocupada por una pila

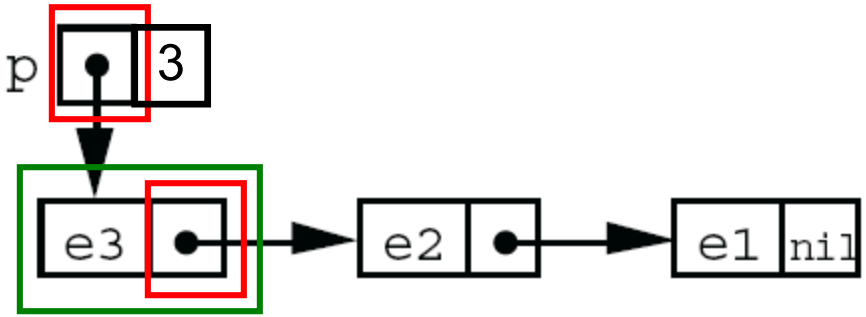
➢ **Para que sea aceptable, todas las operaciones del TAD pila deben tener *Θ(1)***

Implementación Dinámica

```
módulo genérico pilasGenéricas
parámetro tipo elemento
exporta
   tipo pila
   procedimiento crear(sal p:pila)
   función esVacía?(p:pila) devuelve booleano
   función altura(p:pila) devuelve natural
   procedimiento cima(ent p:pila; sal e:elemento;
                      sal error:booleano)
   procedimiento apilar(e/s p:pila; ent e:elemento)
   procedimiento desapilar(e/s p:pila)
   procedimiento duplicar(sal pSal:pila; ent pEnt:pila)
   {duplica la representación de pEnt en pSal (copia profunda)}
   función iguales?(p1,p2:pila) devuelve booleano
   {devuelve verdad si las pilas p1 y p2 tienen la misma altura y los
    mismos elementos en idénticas posiciones}
   procedimiento liberar(e/s p:pila)
   {libera toda la memoria dinámica de p, dejando p vacía}
   procedimiento iniciarIterador(e/s p:pila)
   función haySiguiente(p:pila) devuelve booleano
   procedimiento siguienteYAvanza(e/s p:pila; sal e:elemento;
                                  sal error:booleano)
...
```

Implementación Dinámica

Implementación




```
tipos ptDato = ↑unDato;
    unDato = registro
                dato:elemento;
                sig:ptDato      {puntero a la anterior cima}
             freg;
    pila = registro
             cim:ptDato;  {puntero a la cima de la pila}
             alt:natural; {altura de la pila}
             iter:ptDato {se utiliza para implementar el iterador}
           freg
```

{Implementación basada en lista enlazada simple, con puntero (campo cim del registro pila) al nodo o registro unDato de la lista enlazada que contiene: el elemento que se encuentra en la cima de la pila; y puntero al registro unDato que contiene el elemento que fue cima antes que su dato. La altura de la cima se mantiene calculada en el campo alt del registro pila. Si la pila es vacía cim será nil y alt será 0. El puntero iter, campo de la pila, apuntará al nodo o registro unDato de la lista enlazada que contenga el elemento que toque visitar con el iterador, o tendrá valor nil si no existe elemento a visitar. El iterador permitirá visitar los elementos empezando por el que está en la cima y acabando por el del fondo de la pila (el que lleva más tiempo en la pila)}

Implementación Dinámica

```
procedimiento crearVacía(sal p:pila)
principio
   p.cim:=nil; p.alt:=0; c.iter=nil
fin

función esVacía(p:pila) devuelve booleano
principio
   devuelve p.cim=nil
fin

función altura(p:pila) devuelve natural
principio
   devuelve p.alt
fin
```

Implementación Dinámica

```
procedimiento cima(ent p:pila; sal e:elemento; sal error:booleano)
principio
    si esVacía(p) entonces
       error:=verdad
    sino
       error:=falso;
       e:=p.cim↑.dato
    fsi
fin

procedimiento apilar(e/s p:pila; ent e:elemento)
  variable aux:ptDato
principio
     aux:=p.cim;
     nuevoDato(p.cim);
     p.cim↑.dato:=e;
     p.cim↑.sig:=aux;
     p.alt:=p.alt+1
fin
```

```
tipos ptDato = ↑unDato;
      unDato = registro
                 dato:elemento;
                 sig:ptDato
               freg;
      pila = registro
               cim:ptDato;
               alt:natural;
               iter:ptDato
             freg
```

Implementación Dinámica

```
procedimiento desapilar(e/s p:pila)
  variable aux:ptDato
principio
    si not esVacía(p) entonces
      aux:=p.cim;
      p.cim:=p.cim↑.sig;
      disponer(aux);
      p.alt:=p.alt-1
    fsi
fin
```

```
tipos ptDato = ↑unDato;
      unDato = registro
                 dato:elemento;
                 sig:ptDato
               freg;
      pila = registro
               cim:ptDato;
               alt:natural;
               iter:ptDato
             freg
```

… *{todas las operaciones con coste Θ(1), excepto iguales, liberar y duplicar}*

fin *{del módulo}* *{ver completa en el material de clase}*

Implementación Dinámica

```
tipos ptDato = ↑unDato;
      unDato = registro
                 dato:elemento;
                 sig:ptDato
               freg;
      pila = registro
               cim:ptDato;
               alt:natural;
               iter:ptDato
             freg
```

```
procedimiento duplicar(sal pilaSal:pila; ent pilaEnt:pila)
  variables ptSal,ptEnt:ptDato
principio
    si esVacía(pilaEnt) entonces
      crearVacía(pilaSal);
    sino
      ptEnt:=pilaEnt.cim;
      nuevoDato(pilaSal.cim);
      pilaSal.cim↑.dato:=ptEnt↑.dato;
      ptSal:=pilaSal.cim;
      ptEnt:=ptEnt↑.sig;
      mientrasQue ptEnt≠nil hacer
        nuevoDato(ptSal↑.sig);
        ptSal:=ptSal↑.sig;
        ptSal↑.dato:=ptEnt↑.dato;
        ptEnt:=ptEnt↑.sig
      fmq;
      ptSal↑.sig:=nil;   pilaSal.alt:=pilaEnt.alt
    fsi
fin
```

{copia profunda, coste Θ(N)}

Pila en C++

```
// Interfaz del TAD. Pre-declaraciones:
template<typename T> struct Pila;
template<typename T> void crearVacia(Pila<T>& p);
template<typename T> void apilar(Pila<T>& p, const T& e);
template<typename T> void desapilar(Pila<T>& p);
template<typename T> void cima(const Pila<T>& p, T& e, bool& error);
template<typename T> bool esVacia(const Pila<T>& p);
template<typename T> int altura(const Pila<T>& p);
template<typename T> void duplicar(const Pila<T>& pilaEnt, Pila<T>& pilaSal);
template<typename T> bool operator==(const Pila<T>& p1, const Pila<T>& p2);
template<typename T> void liberar(Pila<T>& p);
template<typename T> void iniciarIterador(Pila<T>& p);
template<typename T> bool existeSiguiente(const Pila<T>& p);
template<typename T> void siguienteYAvanza(Pila<T>& p, T& e, bool& error);
//template<typename T> bool siguienteYAvanza(Pila<T>& p, T& e);

…
```

// VER completa y con documentación en el material de clase . . .

Concepto de Cola

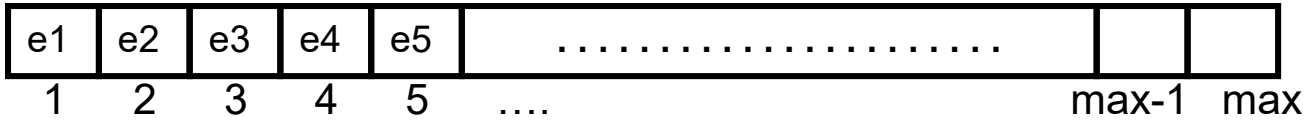
- Cola:
  - Secuencia de elementos de un cierto tipo, dispuestos en una dimensión (***tipo lineal*** de datos)
  - Valor especial: **cola vacía**
  - Operaciones limitadas a los extremos:
    - Nuevos elementos se añaden *al final* de la cola (añadir – **encolar**)
    - Los elementos salen o se eliminan, de la *cabeza* de la cola (eliminar – **desencolar**)
    - Operación de consulta: *observar* el *primero* de los elementos de la cola (**primero**)
  - Estructura **FIFO (First In, First Out):** el orden de salida debe corresponderse con el orden de llegada

Especificación de colas genéricas

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 en ser añadido (entrar) será el primero en ser borrado (salir)}*
operaciones
crear: → cola
{ Devuelve una cola vacía, sin elementos}
encolar: cola c, elemento e→ cola
{ Devuelve la cola igual a la resultante de añadir e a c como último elemento}
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 igual a la 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 nº de elementos de c}
fespec

Representación Estática

- Implementación estática de Cola:
  - Vector de tamaño **max**:
    - longitud limitada de la cola
    - Operación **Encolar** debe implementarse como *parcial* por el caso de **cola llena**
    - **Desencolar** tendrá *coste lineal* $\Theta(N)$
      - Mejora: contadores recordando la posición del *último* elemento de la cola y del *primer* elemento de la cola
- Por ejemplo, primera idea:



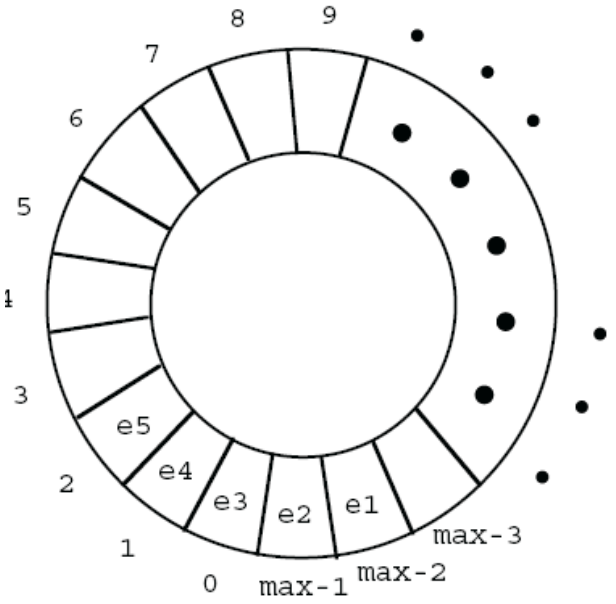
➢ **Para que sea aceptable, todas las operaciones del TAD cola deben tener $\Theta(1)$**

Representación Estática

- Implementación estática circular de Cola:
  - Vector de tamaño **max**:
    - longitud limitada de la cola
    - Encolar operación *parcial* por el caso de cola llena
  - Contadores recordando el ***último*** elemento de la cola y el ***anterior*** al primero

• Circular → Operación de los índices del vector basándose en aritmética **modular** (*mod max*)

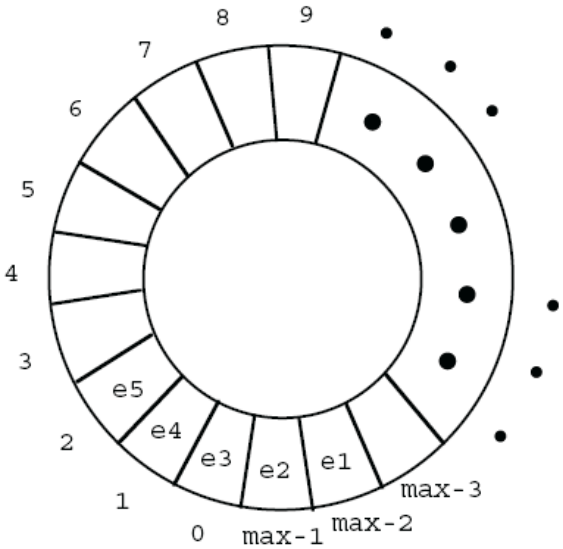
✓ Todas las operaciones tienen coste $\Theta(1)$



Representación Estática

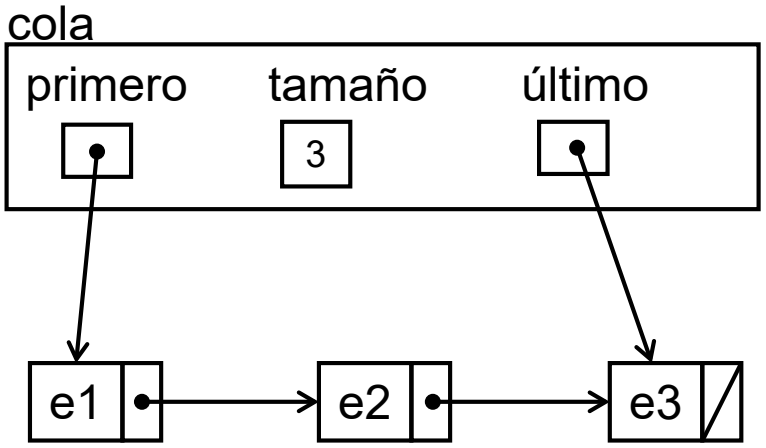
- Características de la implementación estática circular de Cola:
  - Vector de dimensión **max (**indexado 0..max-1)
  - Problema: *¡CONFUSIÓN!*
    - si *anterior* = *último*, ¿Cola vacía o cola llena?
    - Alternativas, para eliminar confusión y distinguir el caso:
      - O dejar máximo tamaño de colas representables en max-1
      - O tener información adicional que permita distinguir el caso:
        - » tamaño de la cola (*0* si cola vacía, *Max* si cola llena)
        - » booleano para indicar si cola vacía o no

✓ Todas las operaciones tienen coste $\Theta(1)$



Representación dinámica

- Representación dinámica de Cola:
  - encadenando mediante punteros los elementos de la cola
    - Coste extra en memoria para los punteros
    - Sin "límite" establecido para la longitud de la cola
  - Puntero a la cabeza o primero de la cola
  - Puntero al último de la cola
  - Todas las operaciones definidas para el TAD cola se implementan con coste $\Theta(1)$

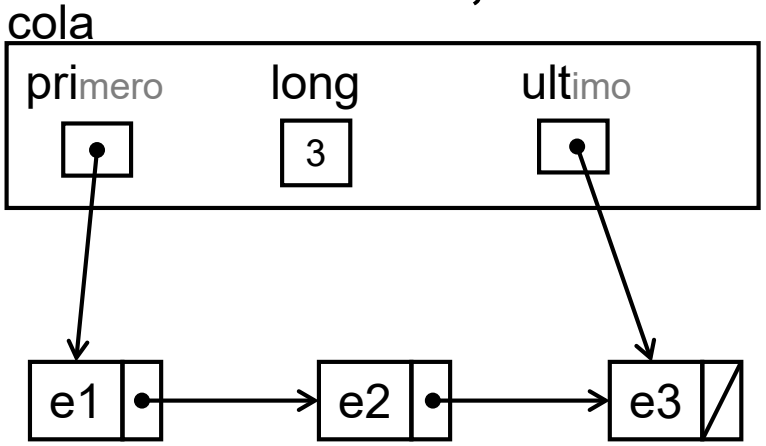



```
{para longitud con O(1) añadir
contador de tamaño al registro cola}
tipos punteroCelda = ↑Celda;
      Celda = registro
                dato:elemento;
                sig:punteroCelda
              freg;
      cola = registro
               primero:punteroCelda;
               último:punteroCelda;
               tamaño:natural
             freg
```

Implementación Dinámica

```
módulo genérico colasGenéricas
parámetro
  tipo elemento
exporta
  tipo cola
  procedimiento crearVacía(sal c:cola)
  procedimiento encolar(e/s c:cola; ent e:elemento)
  función esVacía(c:cola) devuelve booleano
  procedimiento primero(ent c:cola; sal e:elemento; sal error:booleano)
  procedimiento desencolar(e/s c:cola)
  función longitud(c:cola) devuelve natural
  procedimiento duplicar(sal colaSal:cola; ent colaEnt:cola)
  función iguales(cola1,cola2:cola) devuelve booleano
  procedimiento liberar(e/s c:cola)
  procedimiento iniciarIterador(e/s c:cola)
  función haySiguiente(c:cola) devuelve booleano
  procedimiento siguienteYAvanza(e/s c:cola; sal e:elemento;
                                 sal error:booleano)
implementación
...
```

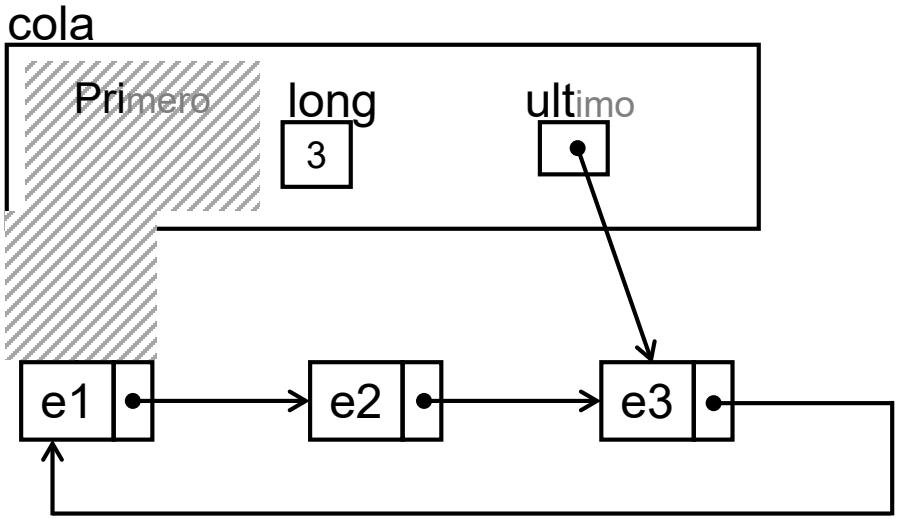
- DINÁMICA, no circular:




```
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
```

{Implementación dinámica, ver completa en el material de clase la versión dinámica no circular}

- DINÁMICA, circular:

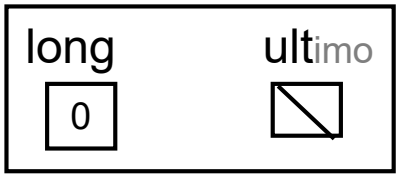


{ambas implementaciones del TAD tendrán idéntica interface…}

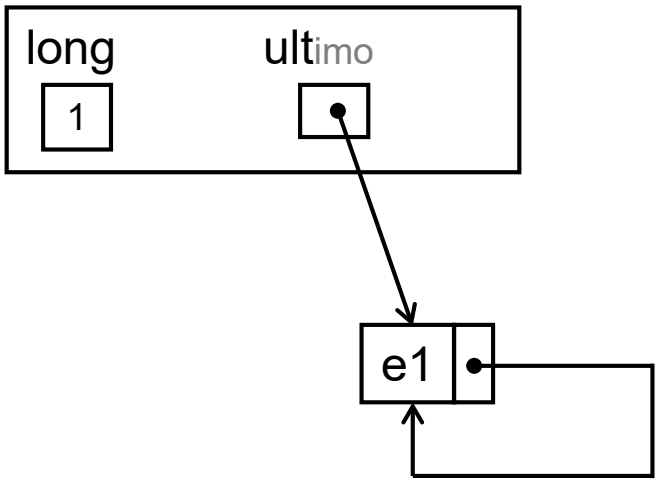
Cola: implementación dinámica circular

Casos a tener en cuenta:

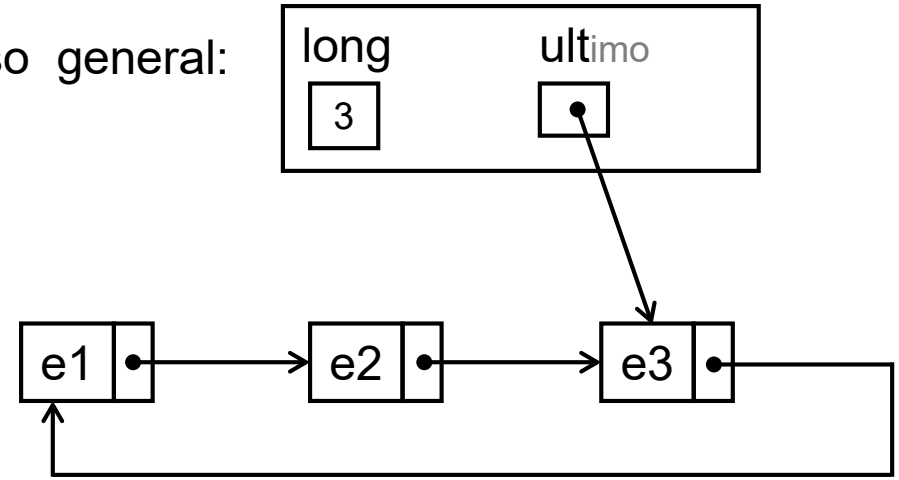
Cola vacía:



Cola con 1 elemento:



Caso general:



¡Cuidado con ciclar infinitamente!

Cola: implementación dinámica circular

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

{versión para lista enlazada circular}

```
tipos ptDato = ↑unDato;
      unDato = registro
                 dato:elemento;
                 sig:ptDato
               freg;
      cola = registro
               ult:ptDato;
               long:natural;
               iter:ptDato
             freg
```

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

Cola: implementación dinámica circular

{versión para lista enlazada circular}

```
procedimiento desencolar(e/s c:cola)
 variable auxPri:ptDato
 principio
   si not esVacía(c) entonces
     auxPri:=c.ult↑.sig;
     si c.ult=auxPri entonces {solo hay un elemento en la cola}
       c.ult:=nil
     sino
       c.ult↑.sig:= auxPri↑.sig;
     fsi;
     disponer(auxPri);
     c.long:=c.long-1;
   fsi
 fin
...
```

```
tipos ptDato = ↑unDato;
      unDato = registro
                 dato:elemento;
                 sig:ptDato
               freg;
      cola = registro
               ult:ptDato;
               long:natural;
               iter:ptDato
             freg
```

{todas las operaciones con coste Θ(1), excepto liberar, duplicar e iguales}

{Cuidado, deben evitarse bucles infinitos:
- *con las operaciones del iterador, y*
- *en liberar, duplicar e iguales}*

Cola 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> void liberar(Cola<Elemento>& c);
template <typename Elemento> void iniciarIterador(Cola<Elemento>& c);
template <typename Elemento> bool haySiguiente(const Cola<Elemento>& c);
template <typename Elemento> bool siguienteYAvanza(Cola<Elemento>& c, Elemento& dato);

…                                                          // sigue . . .
```

Cola en C++

```
// Declaración representación interna

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 void liberar<Elemento>(Cola<Elemento>& c);
    friend void iniciarIterador<Elemento>(Cola<Elemento>& c);
    friend bool haySiguiente<Elemento>(const Cola<Elemento>& c);
    friend bool siguienteYAvanza<Elemento>(Cola<Elemento>& c, Elemento& dato);

  …      // sigue . . .
```

Cola en C++

```
    // Representación de los valores del TAD
    private:
            struct Nodo {
                    Elemento valor;
                    Nodo* siguiente;
            };
            Nodo* elPrimero ;  { no es necesario para una implementación con lista circular}
            Nodo* elUltimo;
            int numDatos;
            Nodo* iter;  {… en implementación con lista circular, será necesario añadir algo más para
                            implementar correctamente el iterador}
};

// Implementación de las operaciones
template<typename Elemento> void vacia(Cola<Elemento>& c) {
            c.numDatos = 0;
            c.elPrimero = nullptr; c.elUltimo = nullptr;
            c.iter=nullptr; … // por seguridad
}
// etc etc implementación de las demás operaciones
```