

# Algoritmos y Estructuras de Datos II

TAD Cola

11 y 13 de abril de 2016

## Clase de hoy

- 1 Repaso
  - Tipos concretos versus abstractos
- 2 Buffer de datos entre productor y consumidor
  - Buffer de datos
  - TAD cola
  - Resolviendo el problema
  - Implementaciones de colas
- 3 Cola de prioridades
  - Especificación del TAD PCola
  - Ordenando con una cola de prioridades
  - Implementación

# Repaso

- cómo vs. qué
- 3 partes
  - 1 análisis de algoritmos
    - algoritmos de ordenación
    - orden de un algoritmo
    - propiedades y jerarquía
    - recurrencias divide y vencerás
  - 2 tipos de datos
    - tipos concretos (arreglos, listas, tuplas, punteros), listas enlazadas
    - tipos abstractos (TAD contador, TAD pila)
  - 3 técnicas de resolución de problemas

# Tipos de datos

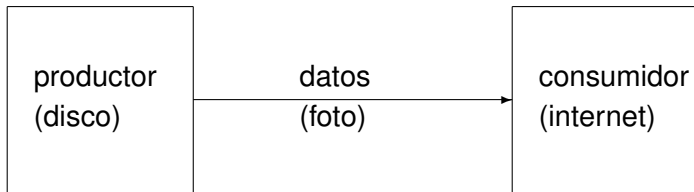
Conceptualmente distinguimos dos clases de tipos de datos:

- Tipos de datos **concretos**:
  - son provistos por el lenguaje de programación,
  - concepto **dependiente** del lenguaje de programación,
  - vimos: arreglos, listas, tuplas y punteros.
- Tipos de datos **abstractos**:
  - **surgen de analizar el problema** a resolver,
  - concepto **independiente** del lenguaje de programación,
  - eventualmente se implementará utilizando tipos concretos,
  - eso da lugar a una **implementación** o **representación** del tipo abstracto
  - vimos:
    - paréntesis balanceados → TAD contador
    - delimitadores balanceados → TAD pila

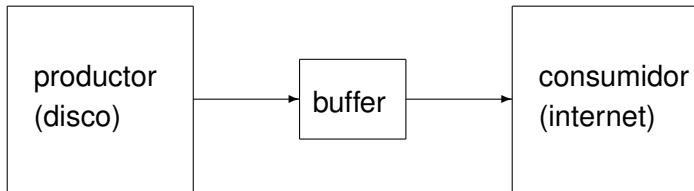
## Buffer de datos

- Imaginemos cualquier situación en que ciertos datos deben transferirse desde una unidad a otra,
- por ejemplo, datos (¿una foto?) que se quiere subir a algún sitio de internet desde un disco,
- un agente suministra o produce datos (el disco) y otro que los utiliza o consume (el sitio de internet),
- esta relación se llama **productor-consumidor**
- para amortiguar el impacto por la diferencia de velocidades, se puede introducir un buffer entre ellos,
- un buffer recibe y almacena los datos a medida que se producen y los emite en el mismo orden, a medida que son solicitados.

## Gráficamente



se interpone un buffer



## Interés

- El programa que realiza la subida de datos puede liberar más rápidamente la lectora del disco.
- El proceso que realizaba la lectura se desocupa antes.
- Se articulan las dos etapas sin necesidad de sincronización.
- El productor se ocupa de lo suyo.
- El consumidor se ocupa de lo suyo.
- El buffer se ocupa de la interacción entre ambos.

## Uso del buffer

- La interposición del buffer no debe afectar el orden en que los datos llegan al consumidor.
- El propósito es sólo permitir que el productor y el consumidor puedan funcionar cada uno a su velocidad sin necesidad de sincronización.
- El buffer inicialmente está **vacío**.
- A medida que se van **agregando** datos suministrados por el productor, los mismos van siendo alojados en el buffer.
- Siempre que sea necesario enviar un dato al consumidor, habrá que comprobar que el buffer no se encuentre **vacío** en cuyo caso se enviará el **primero** que llegó al buffer y se lo **eliminará** del mismo.



# Cola

- Es **algo**, con lo que se pueda
  - inicializar **vacía**,
  - agregar o **encolar** un dato,
  - comprobar si quedan datos en el buffer, es decir, si **es** o no **vacía**
  - examinar el **primer** dato (el más viejo de los que se encuentran en el buffer),
  - quitar o **decolar** un dato.
- El primer dato que se agregó, es el primero que debe enviarse y quitarse de la **cola**.
- Por eso se llama **cola** o también **cola FIFO** (First-In, First-Out).

## Tad cola

- La cola se define por lo que sabemos: sus cinco operaciones
  - inicializar en **vacía**
  - **encolar** un nuevo dato (o elemento)
  - comprobar si **está vacía**
  - examinar el **primer** elemento (si no está vacía)
  - **decolarlo** (si no está vacía).
- Las operaciones **vacía** y **encolar** son capaces de generar todas las colas posibles,
- **está vacía** y **primero**, en cambio, solamente examinan la cola,
- **decolarla** no genera más valores que los obtenibles por **vacía** y **apilar**.

# Especificación del TAD cola

**module** TADCola **where**

**data** Cola e = Vacía  
          | Encolar (Cola e) e

es\_vacía :: Cola e → Bool

primero :: Cola e → e

decolar :: Cola e -> Cola e

-- las dos últimas se aplican sólo a cola no vacía

es\_vacía Vacía = True

es\_vacía (Encolar q e) = False

primero (Encolar q e) = ?

decolar (Encolar q e) = ?

# Especificación del TAD cola

**module** TADCola **where**

**data** Cola e = Vacía  
          | Encolar (Cola e) e

es\_vacía :: Cola e → Bool

primero :: Cola e → e

decolar :: Cola e -> Cola e

- - las dos últimas se aplican sólo a cola no vacía

es\_vacía Vacía = True

es\_vacía (Encolar q e) = False

primero (Encolar q e) | es\_vacía q = ?  
                          | otherwise = ?

decolar (Encolar q e) = ?

# Especificación del TAD cola

**module** TADCola **where**

**data** Cola e = Vacía  
          | Encolar (Cola e) e

es\_vacía :: Cola e → Bool

primero :: Cola e → e

decolar :: Cola e → Cola e

- - las dos últimas se aplican sólo a cola no vacía

es\_vacía Vacía = True

es\_vacía (Encolar q e) = False

primero (Encolar q e) | es\_vacía q = e  
                          | otherwise = ?

decolar (Encolar q e) = ?

# Especificación del TAD cola

**module** TADCola **where**

**data** Cola e = Vacía  
          | Encolar (Cola e) e

es\_vacía :: Cola e → Bool

primero :: Cola e → e

decolar :: Cola e -> Cola e

- - las dos últimas se aplican sólo a cola no vacía

es\_vacía Vacía = True

es\_vacía (Encolar q e) = False

primero (Encolar q e) | es\_vacía q = e  
                          | otherwise = primero q

decolar (Encolar q e) = ?

# Especificación del TAD cola

**module** TADCola **where**

**data** Cola e = Vacía  
          | Encolar (Cola e) e

es\_vacía :: Cola e → Bool

primero :: Cola e → e

decolar :: Cola e -> Cola e

- - las dos últimas se aplican sólo a cola no vacía

es\_vacía Vacía = True

es\_vacía (Encolar q e) = False

primero (Encolar q e) | es\_vacía q = e  
                          | otherwise = primero q

decolar (Encolar q e) | es\_vacía q = ?  
                          | otherwise = ?

# Especificación del TAD cola

**module** TADCola **where**

**data** Cola e = Vacía  
          | Encolar (Cola e) e

es\_vacía :: Cola e → Bool

primero :: Cola e → e

decolar :: Cola e -> Cola e

- - las dos últimas se aplican sólo a cola no vacía

es\_vacía Vacía = True

es\_vacía (Encolar q e) = False

primero (Encolar q e) | es\_vacía q = e  
                      | otherwise = primero q

decolar (Encolar q e) | es\_vacía q = Vacía  
                      | otherwise = ?



# Especificación del TAD cola

**module** TADCola **where**

**data** Cola e = Vacía  
          | Encolar (Cola e) e

es\_vacía :: Cola e → Bool

primero :: Cola e → e

decolar :: Cola e -> Cola e

- - las dos últimas se aplican sólo a cola no vacía

es\_vacía Vacía = True

es\_vacía (Encolar q e) = False

primero (Encolar q e) | es\_vacía q = e  
                      | otherwise = primero q

decolar (Encolar q e) | es\_vacía q = Vacía  
                      | otherwise = Encolar (decolar q) e

## Explicación

Los valores posibles de una cola están expresados por

- ningún elemento: Vacía
- un elemento: Encolar Vacía A, Encolar Vacía B, Encolar Vacía C
- dos elementos: Encolar (Encolar Vacía A) B, Encolar(Encolar Vacía A), A ...
- tres elementos: Encolar (Encolar (Encolar Vacía B) A) A ...
- etcétera

Mostrar en Haskell.

## Ejemplo

- Gracias a las ecuaciones, podemos comprobar que
- primero (Encolar (Encolar (Encolar (Encolar Vacía B) A) A) C) = ?
- decolar (Encolar (Encolar (Encolar (Encolar Vacía B) A) A) C) = ?

## Ejemplo

- Gracias a las ecuaciones, podemos comprobar que
- primero  $(\text{Encolar} (\text{Encolar} (\text{Encolar} (\text{Encolar Vacía B}) A) A) C) = B$
- decolar  $(\text{Encolar} (\text{Encolar} (\text{Encolar} (\text{Encolar Vacía B}) A) A) C) = \text{Encolar} (\text{Encolar} (\text{Encolar Vacía A}) A) C$

## En efecto

primero (Encolar (Encolar (Encolar (Encolar Vacía B) A) A) C)  
= primero (Encolar (Encolar (Encolar Vacía B) A) A)  
= primero (Encolar (Encolar Vacía B) A)  
= primero (Encolar Vacía B)  
= B

## En efecto

decolar (Encolar (Encolar (Encolar (Encolar Vacía B) A) A) C)  
= Encolar (decolar (Encolar (Encolar (Encolar Vacía B) A) A)) C  
= Encolar (Encolar (decolar (Encolar (Encolar Vacía B) A)) A) C  
= Encolar (Encolar (Encolar (decolar (Encolar Vacía B)) A) A) C  
= Encolar (Encolar (Encolar Vacía A) A) C

## Especificación e implementación

**type** queue = ... {- no sabemos aún cómo se implementará -}

**proc** empty(**out** q:queue) {Post: q ~ Vacía}

{Pre: q ~ Q  $\wedge$  e ~ E}

**proc** enqueue(**in/out** q:queue, **in** e:elem)

{Post: q ~ Encolar Q E}

{Pre: q ~ Q  $\wedge$   $\neg$ is\_empty(q)}

**fun** first(q:queue) **ret** e:elem

{Post: e ~ primero Q}

## Especificación e implementación

{Pre:  $q \sim Q \wedge \neg \text{is\_empty}(q)$ }

**proc** dequeue(**in/out** q:queue)

{Post:  $q \sim \text{decolar } Q$ }

**fun** is\_empty(q:queue) **ret** b:**bool**

{Post:  $b = (q \sim \text{Vacía})$ }



## Algoritmo de transferencia de datos con buffer

```
proc buffer ()  
  var d: data  
  var q: queue of data  
  empty(q)  
  produce:= false  
  demand:= false  
  do forever  
    if produce → receive d from producer  
      enqueue(q,d)  
      produce:= false  
    demand  $\wedge$   $\neg$  is_empty(q) → d:= first(q)  
      send d to consumer  
      demand:= false  
      dequeue(q)  
    fi  
  od  
end proc
```

- Hemos asumido que hay dos variables booleanas compartidas:
- produce:
  - variable compartida entre el programa buffer y el productor,
  - el productor le asigna verdadero cuando produce un dato,
  - el programa buffer accede mediante el comando receive.
- demand:
  - variable compartida entre el programa buffer y el consumidor,
  - el consumidor le asigna verdadero cuando espera un dato,
  - el programa buffer se lo envía mediante el comando send.

## Utilización

- El TAD cola tiene numerosas aplicaciones.
- Siempre que se quieran atender pedidos, datos, etc. en el orden de llegada.
- Una aplicación interesante es el algoritmo de ordenación llamado Radix Sort.

# Implementación

Veremos implementaciones:

- Usando listas (si las listas son tipos concretos)
- Usando arreglos.
- Usando listas enlazadas.

## Implementación de colas usando tipo concreto lista

- **type** queue = [elem]
- **proc** empty(**out** q:queue)  
    q:= [ ]  
**end proc**  
    {Post: q ~ Vacía}
- {Pre: q ~ Q}  
**proc** enqueue(**in/out** q:queue; **in** e:elem)  
    q:= (q ◁ e)  
**end proc**  
    {Post: q ~ Encolar Q e}

## Implementación de colas usando tipo concreto lista

- {Pre:  $q \sim Q \wedge \neg \text{is\_empty}(q)$ }  
**fun** first( $q$ :queue) **ret**  $e$ :elem  
     $e := \text{head}(q)$   
**end fun**  
    {Post:  $e \sim \text{primero } Q$ }
- {Pre:  $q \sim Q \wedge \neg \text{is\_empty}(q)$ }  
**proc** dequeue(**in/out**  $q$ :queue)  
     $q := \text{tail}(q)$   
**end proc**  
    {Post:  $q \sim \text{decolar } Q$ }

## Implementación de colas usando tipo concreto lista

- **fun** is\_empty(q:queue) **ret** b:Bool  
    b:= (q = [ ])  
**end fun**  
    {Post: b = (q ~ Vacía)}
- Todas las operaciones son  $\mathcal{O}(1)$ , salvo enqueue que es  $\mathcal{O}(n)$  (lineal) en la longitud de la cola. Pero hay implementaciones del tipo concreto lista que la tornan constante.

## Implementación de colas usando arreglos

- **type** queue = **tuple**  
                  elems: **array**[1..N] **of** elem  
                  size: **nat**  
                  **end**
- **proc** empty(**out** q:queue)  
          q.size:= 0  
          **end proc**  
          {Post: q ~ Vacía}
- {Pre: q ~ Q  $\wedge$   $\neg$ is\_full(q)}  
          **proc** enqueue(**in/out** q:queue, **in** e:elem)  
                  q.size:= q.size + 1  
                  q.elems[q.size]:= e  
                  **end proc**  
                  {Post: q ~ Encolar Q e}



## Implementación de colas usando arreglos

- {Pre:  $q \sim Q \wedge \neg \text{is\_empty}(q)$ }  
**fun** first( $q$ :queue) **ret**  $e$ :elem  
     $e := q.\text{elems}[1]$   
**end fun**  
    {Post:  $e \sim \text{primero } Q$ }
- {Pre:  $q \sim Q \wedge \neg \text{is\_empty}(q)$ }  
**proc** dequeue(**in/out**  $q$ :queue)  
     $q.\text{size} := q.\text{size} - 1$   
    **for**  $i := 1$  **to**  $q.\text{size}$  **do**  
         $q.\text{elems}[i] := q.\text{elems}[i+1]$   
    **od**  
**end proc**  
    {Post:  $q \sim \text{decolar } Q$ }

## Implementación de colas usando arreglos

- **fun** is\_empty(q:queue) **ret** b:Bool  
    b:= (q.size = 0)  
**end fun**  
    {Post: b = (q ~ Vacía)}
- **fun** is\_full(q:queue) **ret** b:Bool  
    b:= (q.size = N)  
**end fun**
- Todas las operaciones son  $\mathcal{O}(1)$ , salvo dequeue que es lineal.

## Implementación eficiente de colas usando arreglos

- **type** queue = **tuple**  
                  elems: **array**[0..N-1] **of** elem  
                  fst: **nat**  
                  size: **nat**  
                  **end**
- **proc** empty(**out** q:queue)  
      q.fst:= 0  
      q.size:= 0  
      **end proc**
- **proc** enqueue(**in/out** q:queue, **in** e:elem)  
      q.elems[(q.fst + q.size) mod N]:= e  
      q.size:= q.size + 1  
      **end proc**

## Implementación eficiente de colas usando arreglos

- **fun** first(q:queue) **ret** e:elem  
    e:= q.elems[q.fst]  
**end fun**
- **proc** dequeue(**in/out** q:queue)  
    q.size:= q.size - 1  
    q.fst:= (q.fst + 1) mod N  
**end proc**
- **fun** is\_empty(q:queue) **ret** b:Bool  
    b:= (q.size = 0)  
**end fun**
- **fun** is\_full(q:queue) **ret** b:Bool  
    b:= (q.size = N)  
**end fun**
- Todas las operaciones son  $\mathcal{O}(1)$ .

# Implementación del TAD cola con listas enlazadas

## Implementación ingenua

- Reusar lo más posible la del TAD pila,
- **type** queue = **pointer to** node
- donde node se define como para el TAD pila,
- empty, is\_empty y destroy como para el TAD pila,
- first como top,
- y dequeue como pop.

# Implementación del TAD cola con listas enlazadas

## Implementación ingenua

Sólo cambia la implementación de enqueue:

{Pre:  $p \sim Q \wedge e \sim E$ }

**proc** enqueue(**in/out** p:queue, **in** e:elem)

**var** q,r: **pointer to** node

alloc(q) {se reserva espacio para el nuevo nodo}

q→value:= e {se aloja allí el elemento e}

q→next:= **null** {el nuevo nodo (\*q) está listo, va a ser el último de la cola}

r:= p {r realiza la recorrida, la búsqueda de su último nodo}

**while** r→next ≠ **null do** {mientras \*r no sea el último nodo}

    r:= r→next {que r pase a señalar el nodo siguiente}

**od** {ahora \*r es el último nodo}

r→next:= q {que el siguiente del que era último sea ahora \*q}

**end proc**

{Post:  $p \sim \text{Encolar } Q \ E$ }

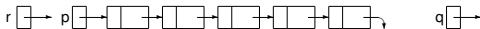
# Implementación del TAD cola con listas enlazadas

## Implementación ingenua

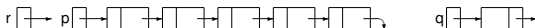
{Pre:  $p \sim Q \wedge e \sim E$ }

**proc** enqueue(**in/out** p:queue, **in** e:elem)

**var** q,r: **pointer to node**

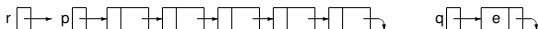


alloc(q)

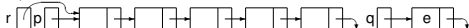


q->value:= e

q->next:= null



r:= p



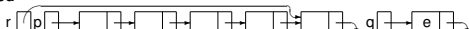
**while** r->next  $\neq$  null **do**

r:= r->next

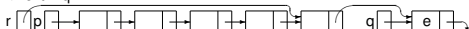
{mientras \*r no sea el último nodo}

{que r pase a señalar el nodo siguiente}

**od**



r->next:= q



**end proc**

{Post:  $p \sim \text{Encolar } Q \ E$ }

## Encolar (implementación ingenua)

En limpio

```
proc enqueue(in/out p:queue,in e:elem)
  var q,r: pointer to node
  alloc(q)
  q→value:= e
  q→next:= null
  r:= p
  while r→next  $\neq$  null do
    r:= r→next
  od
  r→next:= q
end proc
```

¿Anda bien si p es la cola vacía?



# Implementación del TAD cola con listas enlazadas

## Implementación ingenua (corregida)

Sólo cambia la implementación de enqueue:

```
{Pre: p ~ Q ∧ e ~ E}
proc enqueue(in/out p:queue,in e:elem)
  var q,r: pointer to node
  alloc(q)                                {se reserva espacio para el nuevo nodo}
  q→value:= e                              {se aloja allí el elemento e}
  q→next:= null                          {el nuevo nodo (*q) va a ser el último de la cola}
                                           {el nodo *q está listo, debe ir al final de la cola}
                                           {si la cola es vacía con esto alcanza}
  if p = null → p:= q
  p ≠ null →                               {si no es vacía, se inicia la búsqueda de su último nodo}
    r:= p                                  {r realiza la búsqueda a partir del primer nodo}
    while r→next ≠ null do              {mientras *r no sea el último nodo}
      r:= r→next                          {que r pase a señalar el nodo siguiente}
    od                                    {ahora *r es el último nodo}
    r→next:= q                            {que el siguiente del que era último sea ahora *q}
  fi
end proc
{Post: p ~ Encolar Q E}
```

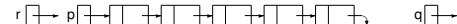
# Implementación del TAD cola con listas enlazadas

## Implementación ingenua (corregida)

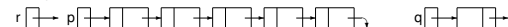
{Pre:  $p \sim Q \wedge e \sim E$ }

**proc** enqueue(**in/out** p:queue, **in** e:elem)

**var** q,r: **pointer to node**

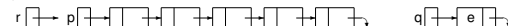


alloc(q)



q->value := e

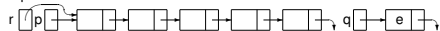
q->next := null



**if** p = null  $\rightarrow$  p := q

{no engañarse con el dibujo, la cola puede ser vacía}

p  $\neq$  null  $\rightarrow$  r := p



**while** r->next  $\neq$  null **do**

{mientras r no sea el último nodo}

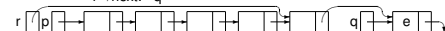
r := r->next

{que r pase a señalar el nodo siguiente}

**od**



r->next := q



**fi**

**end proc**

{Post: p  $\sim$  Encolar Q E}

# Encolar (implementación ingenua, corregida)

En limpio

```
proc enqueue(in/out p:queue,in e:elem)
  var q,r: pointer to node
  alloc(q)
  q→value:= e
  q→next:= null
  if p = null → p:= q
  p ≠ null → r:= p
  while r→next ≠ null do
    r:= r→next
  od
  r→next:= q
fi
end proc
```

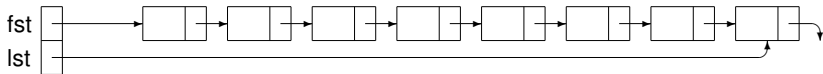
## Conclusiones

- Todas las operaciones son constantes,
- salvo enqueue que es lineal,
- ya que debe recorrer toda la lista hasta encontrar el último nodo.
- Hay al menos dos soluciones a este problema:
  - Mantener dos punteros: uno al primero y otro al último,
  - o utilizar listas enlazadas **circulares**.

# Implementación del TAD cola con listas enlazadas y dos punteros

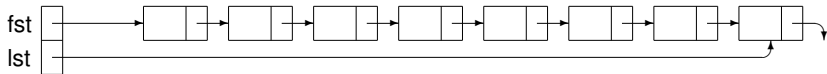
```
type node = tuple  
    value: elem  
    next: pointer to node  
end  
type queue = tuple  
    fst: pointer to node  
    lst: pointer to node  
end
```

Gráficamente, puede representarse de la siguiente manera



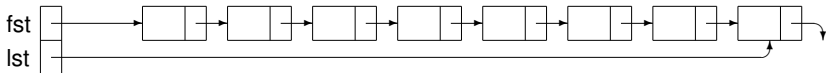
# Cola vacía

```
proc empty(out p:queue)  
  p.fst:= null  
  p.lst:= null  
end proc  
{Post: p ~ Vacía}
```



# Primer elemento

```
{Pre: p ~ Q ∧ ¬is_empty(p)}  
fun first(p:queue) ret e:elem  
    e := p.fst → value  
end fun  
{Post: e ~ primero Q}
```

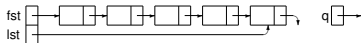


# Encolar

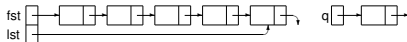
{Pre:  $p \sim Q \wedge e \sim E$ }

**proc** enqueue(**in/out** p:queue, **in** e:elem)

**var** q: **pointer to node**

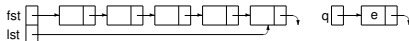


alloc(q)



q->value := e

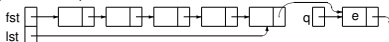
q->next := null



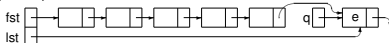
**if** p.lst = null → p.fst := q                      {caso enqueue en cola vacía}

    p.lst := q

p.lst ≠ null → p.lst->next := q



p.lst := q



**fi**

**end proc**

{Post:  $p \sim \text{Encolar } Q \ E$ }



# Encolar

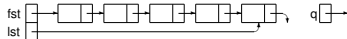
En limpio

```
proc enqueue(in/out p:queue,in e:elem)
  var q: pointer to node
  alloc(q)
  q→value:= e
  q→next:= null
  if p.lst = null → p.fst:= q
                    p.lst:= q
    p.lst ≠ null → p.lst→next:= q
                    p.lst:= q
  fi
end proc
```

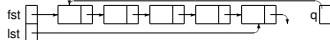
# Decolar

{Pre:  $p \sim Q \wedge \neg \text{is\_empty}(p)$   
**proc** dequeue(**in/out** p:queue)

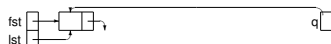
**var** q: pointer to node



q := p.fst



**if** p.fst = p.lst →

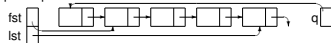


p.fst := null (caso cola con un solo elemento)

p.lst := null



p.fst ≠ p.lst → p.fst := p.fst->next



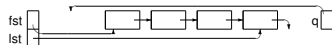
**fi**

free(q)



**end proc**

{Post:  $p \sim \text{decolar } Q$ }



# Decolar

En limpio

```
proc dequeue(in/out p:queue)
  var q: pointer to node
  q:= p.fst
  if p.fst = p.lst  $\rightarrow$  p.fst:= null
                    p.lst:= null
    p.fst  $\neq$  p.lst  $\rightarrow$  p.fst:= p.fst->next
  fi
  free(q)
end proc
```

## Examinar si es vacía

```
{Pre:  $p \sim Q$ }  
fun is_empty(p:queue) ret b:Bool  
    b:= (p.fst = null)  
end fun  
{Post:  $b \sim \text{es\_vacía } Q$ }
```

# Destroy

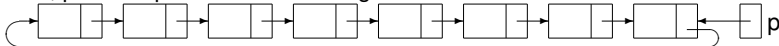
```
proc destroy(in/out p:queue)  
    while  $\neg$  is_empty(p) do dequeue(p) od  
end proc
```

Todas las operaciones son constantes, salvo el destroy que es lineal.

# Implementación del TAD cola con listas enlazadas cíclicas

```
type node = tuple  
    value: elem  
    next: pointer to node  
end  
type queue = pointer to node
```

Gráficamente, puede representarse de la siguiente manera



## Explicación

- La lista es circular,
- es decir que además de los punteros que ya teníamos en implementaciones anteriores,
- el último nodo tiene un puntero al primero,
- alcanza con saber dónde se encuentra el último nodo para saber también dónde está el primero.

## Cola vacía

```
proc empty(out p:queue)
    p := null
end proc
{Post: p ~ vacia}
```



## Primer elemento

```
{Pre:  $p \sim Q \wedge \neg \text{is\_empty}(p)$ }  
fun first(p:queue) ret e:elem  
    e:= p→next→value  
end fun  
{Post:  $e \sim \text{primero } Q$ }
```

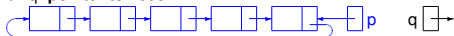


# Encolar: caso cola no vacía (en azul, la cola)

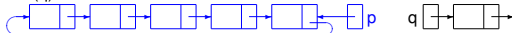
{Pre:  $p \sim Q \wedge e \sim E$ }

**proc** enqueue(**in/out** p:queue,**in** e:elem)

**var** q: **pointer to node**



alloc(q)

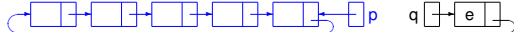


$q \rightarrow \text{value} := e$

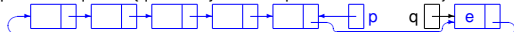


**if**  $p = \text{null}$   $\rightarrow$   $q \rightarrow \text{next} := q$  {caso enqueue en cola vacía}

$p \neq \text{null}$   $\rightarrow$   $q \rightarrow \text{next} := p \rightarrow \text{next}$  {que el nuevo último apunte al primero}



$p \rightarrow \text{next} := q$  {que el viejo último apunte al nuevo último}



**fi**

$p := q$

{que p también apunte al nuevo último}



**end proc**

{Post:  $p \sim \text{Encolar } Q \ E$ }

# Encolar: caso cola vacía (en azul, la cola)

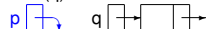
{Pre:  $p \sim Q \wedge e \sim E$ }

**proc** enqueue(**in/out** p:queue,**in** e:elem)

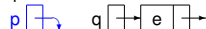
**var** q: **pointer to node**



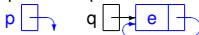
alloc(q)



q→value:= e



**if** p = **null** → q→next:= q



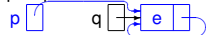
p ≠ **null** → q→next:= p→next

p→next:= q

{caso enqueue en cola no vacía}

**fi**

p:= q



**end proc**

{Post: p ~ Encolar Q E}

# Encolar

En limpio

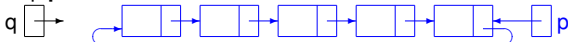
```
proc enqueue(in/out p:queue,in e:elem)
  var q: pointer to node
  alloc(q)
  q→value:= e
  if p = null → q→next:= q
    p ≠ null → q→next:= p→next
                p→next:= q
  fi
  p:= q
end proc
```

# Decolar: caso cola con más de un elemento

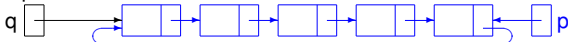
{Pre:  $p \sim Q \wedge \neg \text{is\_empty}(p)$ }

**proc** dequeue(**in/out** p:queue)

**var** q: **pointer to node**



$q := p \rightarrow \text{next}$



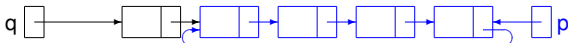
**if**  $p = q \rightarrow p := \text{null}$

{caso cola con un solo elemento}

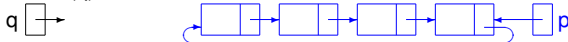
$p \neq q \rightarrow p \rightarrow \text{next} := q \rightarrow \text{next}$

{caso cola con más de un elemento}

**fi**



free(q)



**end proc**

{Post:  $p \sim \text{decolar } Q$ }

## Decolar: caso cola con un solo elemento

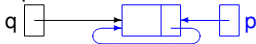
{Pre:  $p \sim Q \wedge \neg \text{is\_empty}(p)$ }

**proc** dequeue(**in/out** p:queue)

**var** q: **pointer to node**



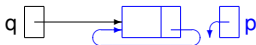
$q := p \rightarrow \text{next}$



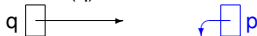
**if**  $p = q \rightarrow p := \text{null}$

$p \neq q \rightarrow p \rightarrow \text{next} := q \rightarrow \text{next}$

**fi**



$\text{free}(q)$



**end proc**

{Post:  $p \sim \text{decolar } Q$ }

{caso cola con un solo elemento}  
{caso cola con más de un elemento}

## Examinar si es vacía

```
{Pre:  $p \sim Q$ }  
fun is_empty(p:queue) ret b:Bool  
    b:= (p = null)  
end fun  
{Post:  $b \sim \text{es\_vacía } Q$ }
```

# Destroy

```
proc destroy(in/out p:queue)  
    while  $\neg$  is_empty(p) do dequeue(p) od  
end proc
```

Todas las operaciones son constantes, salvo el destroy que es lineal.



## Cola de prioridades

- Volvamos al problema de ordenación: tengo ciertos elementos que quiero ordenar.
- Se nos ocurre la siguiente solución:
  - si tuviera **algo** inicialmente **vacío**, donde pueda **ir almacenando** los elementos uno a uno.
  - si tuviera una operación que permitiera **ver** cuál es el mayor de todos los almacenados, en caso de que **haya alguno almacenado**.
  - si tuviera una operación para **eliminar** el mayor de todos los almacenados, en caso de que **haya alguno almacenado**.
- Entonces podría ordenar en 2 etapas:
  - comenzando con el **vacío**, **almaceno** uno a uno todos los elementos a ordenar
  - luego **extraigo** uno a uno desde el **mayor** hasta el menor de todos, colocandolo en su lugar

## Cola de prioridades

- Es similar a la cola,
- pero el orden de llegada no importa,
- cada elemento tiene asignada una prioridad,
- los elementos son atendidos según su prioridad exclusivamente,
- primero al de mayor prioridad y luego los de menor prioridad.

## Asumimos

- Asumimos que existe un orden total entre los elementos de la cola,
- que un elemento sea mayor que otro significa que tiene mayor prioridad,
- las operaciones son las mismas que la de cola:
  - **vacía**
  - **encolar**
  - **es\_vacía**
  - **primero**
  - **decolar**
- sólo que **primero** devuelve el de mayor prioridad,
- y **decolar** quita el de mayor prioridad.

# Especificación del TAD Pcola

**module** TADPCola **where**

**data** Ord e  $\Rightarrow$  PCola e = Vacía  
| Encolar (PCola e) e

es\_vacía :: Ord e  $\Rightarrow$  PCola e  $\rightarrow$  Bool

primero :: Ord e  $\Rightarrow$  PCola e  $\rightarrow$  e

decolar :: Ord e  $\Rightarrow$  PCola e  $\rightarrow$  PCola e

- - las dos últimas se aplican sólo a pcola no vacía

es\_vacía Vacía = True

es\_vacía (Encolar q e) = False

primero (Encolar q e) = ?

decolar (Encolar q e) = ?

# Especificación del TAD PCola

**module** TADPCola **where**

...

**primero** (Encolar q e) | **es\_vacía** q = e  
| otherwise = ?

**decolar** (Encolar q e) = ?

## Especificación del TAD PCola

**module** TADPCola **where**

...

**primero** (Encolar q e) | **es\_vacía** q = e  
| e >= **primero** q = ?  
| otherwise = ?

**decolar** (Encolar q e) = ?

## Especificación del TAD PCola

**module** TADPCola **where**

...

**primero** (Encolar q e) | **es\_vacía** q = e  
| e >= **primero** q = e  
| otherwise = **primero** q

**decolar** (Encolar q e) = ?

# Especificación del TAD Pcola

**module** TADPCola **where**

...

**primero** (Encolar q e) | **es\_vacia** q = e  
| e >= **primero** q = e  
| otherwise = **primero** q

**decolar** (Encolar q e) | **es\_vacia** q = ?  
| e >= **primero** q = ?  
| otherwise = ?



# Especificación del TAD Pcola

**module** TADPCola **where**

...

**primero** (**Encolar** q e) | **es\_vacia** q = e  
| e >= **primero** q = e  
| otherwise = **primero** q

**decolar** (**Encolar** q e) | **es\_vacia** q = **Vacia**  
| e >= **primero** q = q  
| otherwise = **Encolar** (**decolar** q) e

## Usando la PCola para ordenar

**type** pqueue = ... {- no sabemos aún cómo se implementará -}

**proc** empty(**out** q:pqueue) {Post: q ~ Vacía}

{Pre: q ~ Q  $\wedge$  e ~ E}

**proc** enqueue(**in/out** q:pqueue,**in** e:elem)

{Post: q ~ Encolar Q E}

{Pre: q ~ Q  $\wedge$   $\neg$ is\_empty(q)}

**fun** first(q:pqueue) **ret** e:elem

{Post: e ~ primero Q}

## Usando la PCola para ordenar

```
{Pre:  $q \sim Q \wedge \neg \text{is\_empty}(q)$ }  
proc dequeue(in/out q:pqueue)  
{Post:  $q \sim \text{decolar } Q$ }  
  
fun is_empty(q:pqueue) ret b:bool  
{Post:  $b = (q \sim \text{Vacía})$ }
```

## Algoritmo de ordenación

```
proc pqueue_sort (in/out a: array[1..n] of T)
  var q: pqueue of T
  empty(q);
  for i:= 1 to n do
    enqueue(q,a[i])
  od
  for i:= n downto 1 do
    a[i]:= first(q)
    dequeue(q)
  od
end proc
```

## ¿Cómo podríamos implementar PCola?

- ¿Usando listas (asumiendo que vienen con el lenguaje)?
- ¿Usando arreglos?
- ¿Usando listas enlazadas?
- ¿Qué orden tendrían las operaciones en cada caso?
- Más adelante vamos a ver una implementación muy ingeniosa de cola de prioridades.