

# Algoritmos y Estructuras de Datos II

TADS: Implementaciones de contadores

10 de abril de 2017

# Clase de hoy

- 1 TAD contador
  - Especificación
  - Interface
  - Implementación

# Especificación del TAD Contador

```
module TADContador where
```

```
data Contador = Inicial  
             | Incrementar Contador
```

```
es_inicial :: Contador → Bool
```

```
decrementar :: Contador → Contador
```

- - se aplica solo a un Contador que no sea Inicial

```
es_inicial Inicial = True
```

```
es_inicial (Incrementar c) = False
```

```
decrementar (Incrementar c) = c
```

# Interface

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

**proc** init (**out** c: counter) {Post: c ~ Inicial}

{Pre: c ~ C} **proc** inc (**in/out** c: counter) {Post: c ~ Incrementar C}

{Pre: c ~ C  $\wedge$   $\neg$ is\_init(c)}

**proc** dec (**in/out** c: counter)

{Post: c ~ decrementar C}

**fun** is\_init (c: counter) **ret** b: **bool** {Post: b = (c ~ Inicial)}

# Implementación natural

- Lo más natural es implementarlo con un natural o un entero.
- **type** counter = **nat**
- **proc** init (**out** c: counter)  
    c:= 0  
**end proc**  
    {Post: c ~ Inicial}
- {Pre: c ~ C}  
**proc** inc (**in/out** c: counter)  
    c:= c+1  
**end proc**  
    {Post: c ~ Incrementar C}

# Implementación natural

- $\{\text{Pre: } c \sim C \wedge \neg \text{is\_init}(c)\}$   
**proc** dec (**in/out** c: counter)  
    c := c-1  
**end proc**  
 $\{\text{Post: } c \sim \text{decrementar } C\}$
- **fun** is\_init (c: counter) **ret** b: **bool**  
    b := (c = 0)  
**end fun**  
 $\{\text{Post: } b = (c \sim \text{Inicial})\}$
- Todas las operaciones son  $\mathcal{O}(1)$ .

# Implementación rara

- Pero otras implementaciones son posibles:
- **type** counter = **int**
- **proc** init (**out** c: counter)  
    c:= 17  
**end proc**
- **proc** inc (**in/out** c: counter)  
    c:= c-4  
**end proc**

# Implementación rara

- **proc** dec (**in/out** c: counter)  
    c:= c+4  
**end proc**
- **fun** is\_init (c: counter) **ret** b: **bool**  
    b:= (c = 17)  
**end fun**
- Todas las operaciones son  $\mathcal{O}(1)$ .



## Implementación rara 2

- **type** counter = nat
- **proc** init (**out** c: counter)  
    c:= 1  
**end proc**
- **proc** inc (**in/out** c: counter)  
    c:= c\*2  
**end proc**
- **proc** dec (**in/out** c: counter)  
    c:= c / 2  
**end proc**

## Implementación rara 2

- **fun** is\_init (c: counter) **ret** b: **bool**  
    b:= (c = 1)  
**end fun**
- Todas las operaciones son  $\mathcal{O}(1)$ .
- ¿Cuál es el inconveniente de esta implementación?

# Observaciones

- Cuando especificamos definiremos qué deben hacer las operaciones.
- Al implementar, definimos el cómo y prestamos atención a la eficiencia.
- En las 3 implementaciones que dimos del TAD Contador, todas las operaciones resultaron constantes (¡buenísimo!).
- Pero la primera fue más eficiente en el uso del espacio.

# Algoritmos y Estructuras de Datos II

TAD Cola

12 de abril de 2017

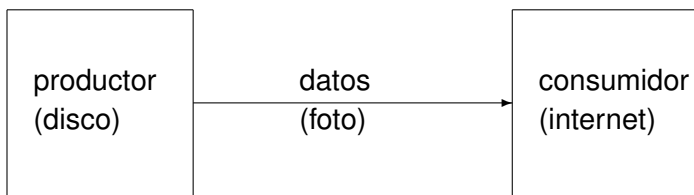
# Clase de hoy

- 1 Buffer de datos entre productor y consumidor
  - Buffer de datos
  - TAD cola
  - Resolviendo el problema
  
- 2 Cola de prioridades
  - Especificación del TAD PCola
  - Ordenando con una cola de prioridades

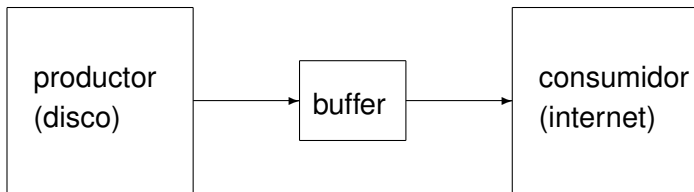
## Buffer de datos

- 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.
- 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** (Encolar (Encolar (Encolar (Encolar Vacía B) A) A) C) = B
- **decolar** (Encolar (Encolar (Encolar (Encolar Vacía B) A) A) C) = Encolar (Encolar (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

## Sobre las ecuaciones que definen una operación

- Además de cubrir todos los casos posibles (como señalamos en los TADs Contador y Pila), ahora es necesario observar que tanto **primero** como **decolar** **ocurren en el lado derecho** de las ecuaciones.
- Si en cada una de esas ocurrencias:
  - el argumento al que se aplica satisface la precondition de la operación, y
  - es sintácticamente más pequeño que el argumento correspondiente en el lado izquierdo de la ecuación.
- entonces, podemos estar seguros de que es una definición sintácticamente correcta.



## Por ejemplo

- Por ejemplo,
- **primero** se aplica al argumento  $q$ , en el lado derecho de la ecuación.
- comprobamos que  $q$  satisface la precondition de **primero**, pues “otherwise” en este caso significa que  $q$  no es vacía.
- comprobamos que  $q$  es sintácticamente más pequeño que **Encolar**  $q$  e, que es el argumento correspondiente en el lado izquierdo de la ecuación.
- como además podemos comprobar que todos los casos están cubiertos por las ecuaciones,
- entonces **primero** es una definición sintácticamente correcta.

## 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.

## 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,
- los elementos son valores,
- los elementos son atendidos según su valor exclusivamente,
- primero al de mayor valor y luego los de menor valor.
- Por esa razón a ese valor se le puede llamar prioridad: se atiende a cada elemento según su 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 mayor valor,
- y **decolar** quita el de mayor valor.

# 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\_vacía** q = e  
| e >= **primero** q = e  
| otherwise = **primero** q

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

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

## Observaciones

- Es posible convencerse de que **primero** devuelve siempre el máximo de la pcola:
  - en la primera ecuación devuelve el único (por lo tanto, el máximo) de la pcola,
  - en la segunda ecuación devuelve e, que por hipótesis inductiva es mayor o igual al máximo del resto de los elementos de la pcola,
  - en la tercera ecuación devuelve el que por hipótesis inductiva es el máximo de los elementos del resto de la pcola, que por estar en el "otherwise" es también mayor que e.
- De la misma manera, es posible convencerse de que **decolar** elimina el primero (el máximo) de la pcola (en caso de estar repetido, elimina una sola ocurrencia del mismo).



## Ecuaciones entre constructores

- Hemos dicho que en el TAD PCola el orden de llegada de los elementos a la pcola no tiene ninguna importancia.
- Sin embargo no hay ninguna propiedad en nuestra especificación que nos permita afirmar, por ejemplo, que
  - Encolar (Encolar (Encolar Vacía 2) 3) 4
  - Encolar (Encolar (Encolar Vacía 3) 4) 2son en realidad la misma pcola.
- ¿Cómo podemos expresar que el orden de llegada no es relevante?
- Puede haber varias, una forma económica es con el axioma
  - $\text{Encolar}(\text{Encolar } q \text{ e}) \text{ e}' = \text{Encolar}(\text{Encolar } q \text{ e}') \text{ e}$   
para toda cola  $q$  y elementos  $e$  y  $e'$ .

## Ecuaciones entre constructores

- En efecto, a pesar de que el axioma
  - $\text{Encolar}(\text{Encolar } q \text{ e}) \text{ e}' = \text{Encolar}(\text{Encolar } q \text{ e}') \text{ e}$
- dice que dos elementos consecutivos son intercambiables,
- es posible utilizar repetidamente el axioma para establecer que, por ejemplo,
  - $\text{Encolar}(\text{Encolar}(\text{Encolar Vacía } 2) 3) 4$
  - $\text{Encolar}(\text{Encolar}(\text{Encolar Vacía } 3) 4) 2$son iguales.
- Por ejemplo, intercambiando primero 2 con 3, y luego 2 con 4.
- Es posible convencerse de que el axioma expresa exactamente lo que buscábamos: que el orden entre los elementos es irrelevante.
- Agreguemos entonces este axioma a la especificación.

# 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

Encolar (Encolar q e) e' = Encolar (Encolar q e') e

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

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

## Nueva dificultad

- Si bien este axioma fue necesario, introduce una nueva dificultad sobre la definición de las operaciones:
- ¿están bien definidas **primero** y **decolar**?
- ¿o es posible que al aplicar **primero** (o **decolar**) a “dos pcolas iguales” obtengamos resultados diferentes?
- afortunadamente, ya hemos observado que **primero** devuelve siempre el máximo, por consiguiente no depende del orden de sus elementos.
- Y un razonamiento similar hemos hecho para **decolar**.
- Podemos concluir que tanto **primero** como **decolar** **respetan el axioma**.

## 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
```