

# Algoritmos y Estructuras de Datos II

TADS: Implementaciones elementales

15 de abril de 2013

## Clase de hoy

- 1 Repaso
  - Tipos concretos versus abstractos
- 2 TAD contador
  - Especificación
  - Interface
  - Implementación
- 3 TAD pila
  - Especificación del TAD pila
  - Interface
  - Implementación
- 4 TAD cola
  - Especificación
  - Interface
  - Implementación
  - Implementación eficiente de colas usando arreglos

# Repaso

- cómo vs. qué
- 3 partes
  - 1 análisis de algoritmos
    - algoritmos de ordenación
    - notación  $\mathcal{O}$ ,  $\Omega$  y  $\Theta$ .
    - propiedades y jerarquía
    - recurrencias (D. y V., homogéneas y no homogéneas)
  - 2 tipos de datos
    - tipos concretos (arreglos, listas, tuplas, punteros)
    - tipos abstractos (TAD contador, TAD pila, TAD cola, TAD pcola)
    - hoy: implementaciones elementales
  - 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 entre productor y consumidor → TAD cola
    - cola de prioridades, TAD pcola

# Especificación del TAD contador

## TAD contador

### constructores

`inicial` : contador

`incrementar` : contador  $\rightarrow$  contador

### operaciones

`es_inicial` : contador  $\rightarrow$  booleano

`decrementar` : contador  $\rightarrow$  contador

{se aplica sólo a un contador que no sea inicial}

### ecuaciones

`es_inicial(inicial)` = verdadero

`es_inicial(incrementar(c))` = falso

`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 ~ init}
- {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 = nat
- **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 problema de esta implementación?

## Especificación del TAD pila

**TAD** pila[elem]

### constructores

**vacía** : pila

**apilar** : elem  $\times$  pila  $\rightarrow$  pila

### operaciones

**es\_vacía** : pila  $\rightarrow$  booleano

**primero** : pila  $\rightarrow$  elem {se aplica sólo a una pila no vacía}

**desapilar** : pila  $\rightarrow$  pila {se aplica sólo a una pila no vacía}

### ecuaciones

**es\_vacía**(**vacía**) = verdadero

**es\_vacía**(**apilar**(e,p)) = falso

**primero**(**apilar**(e,p)) = e

**desapilar**(**apilar**(e,p)) = p

# Interface

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

**proc** empty(**out** p:stack) {Post: p ~ vacia}

{Pre: p ~ P  $\wedge$  e ~ E}

**proc** push(**in** e:elem,**in/out** p:stack)

{Post: p ~ apilar(E,P)}

{Pre: p ~ P  $\wedge$   $\neg$ is\_empty(p)}

**fun** top(p:stack) **ret** e:elem

{Post: e ~ primero(P)}

# Interface

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

**proc** pop(**in/out** p:stack)

{Post:  $p \sim \text{desapilar}(P)$ }

**fun** is\_empty(p:stack) **ret** b:bool

{Post:  $b = (p \sim \text{vacía})$ }

# Implementación

Veremos dos implementaciones:

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



## Implementación de pilas usando tipo concreto lista

- **type** stack = [elem]
- **proc** empty(**out** p:stack)  
    p:= [ ]  
**end proc**  
    {Post: p ~ vacía}
- {Pre: p ~ P }  
**proc** push(**in** e:elem,**in/out** p:stack)  
    p:= (e ▷ p)  
**end proc**  
    {Post: p ~ apilar(e,P)}

## Implementación de pilas usando tipo concreto lista

- {Pre:  $p \sim P \wedge \neg \text{is\_empty}(p)$ }  
**fun** top( $p:\text{stack}$ ) **ret**  $e:\text{elem}$   
      $e := \text{head}(p)$   
**end fun**  
 {Post:  $e \sim \text{primero}(P)$ }
- {Pre:  $p \sim P \wedge \neg \text{is\_empty}(p)$ }  
**proc** pop(**in/out**  $p:\text{stack}$ )  
      $p := \text{tail}(p)$   
**end proc**  
 {Post:  $p \sim \text{desapilar}(P)$ }

## Implementación de pilas usando tipo concreto lista

- **fun** is\_empty(p:stack) **ret** b:Bool  
    b:= (p = [ ])  
**end**  
    {Post: b = (p ~ vacía)}
- Todas las operaciones son  $\mathcal{O}(1)$ .

## Implementación de pilas usando arreglos

- **type** stack = **tuple**  
     elems: **array**[1..N] **of** elem  
     size: **nat**  
     **end**
- **proc** empty(**out** p:stack)  
     p.size:= 0  
   **end proc**  
   {Post: p ~ vacía}
- {Pre: p ~ P  $\wedge$   $\neg$ is\_full(p)}  
   **proc** push(**in** e:elem,**in/out** p:stack)  
     p.size:= p.size + 1  
     p.elems[p.size]:= e  
   **end proc**  
   {Post: p ~ apilar(e,P)}

## Implementación de pilas usando arreglos

- $\{\text{Pre: } p \sim P \wedge \neg \text{is\_empty}(p)\}$   
**fun** top(p:stack) **ret** e:elem  
     e:= p.elems[p.size]  
**end fun**  
 $\{\text{Post: } e \sim \text{primero}(P)\}$
- $\{\text{Pre: } p \sim P \wedge \neg \text{is\_empty}(p)\}$   
**proc** pop(in/out p:stack)  
     p.size:= p.size - 1  
**end proc**  
 $\{\text{Post: } p \sim \text{desapilar}(P)\}$

## Implementación de pilas usando arreglos

- **fun** is\_empty(p:stack) **ret** b:Bool  
    b:= (p.size = 0)  
**end**  
    {Post: b = (p ~ vacía)}
- **fun** is\_full(p:stack) **ret** b:Bool  
    b:= (p.size = N)  
**end**
- Todas las operaciones son  $\mathcal{O}(1)$ .

# Especificación del TAD cola

**TAD** cola[elem]

## constructores

**vacía** : cola

**encolar** : cola  $\times$  elem  $\rightarrow$  cola

## operaciones

**es\_vacía** : cola  $\rightarrow$  booleano

**primero** : cola  $\rightarrow$  elem

**decolar** : cola  $\rightarrow$  cola

{se aplica sólo a una cola no vacía}

{se aplica sólo a una cola no vacía}

## ecuaciones

es\_vacía(**vacía**) = verdadero

es\_vacía(**encolar**(q,e)) = falso

primero(**encolar**(**vacía**,e)) = e

primero(**encolar**(**encolar**(q,e'),e)) = primero(**encolar**(q,e'))

decolar(**encolar**(**vacía**,e)) = **vacía**

decolar(**encolar**(**encolar**(q,e'),e)) = **encolar**(decolar(**encolar**(q,e')),e)

# Interface

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

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

{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)}



# Interface

{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:  $b = (q \sim \text{vacía})$ }

# Implementación

Veremos implementaciones:

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

## Implementación de colas usando tipo concreto lista

- **type** queue = [elem]
- **proc** empty(**out** q:queue)  
     q:= [ ]  
**end**  
 {Post: q ~ vacía}
- {Pre: q ~ Q}  
**proc** enqueue(**in/out** q:queue; **in** e:elem)  
     q:= (q  $\triangleleft$  e)  
**end**  
 {Post: q ~ encolar(Q,e)}

## Implementación de colas usando tipo concreto lista

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

## Implementación de colas usando tipo concreto lista

- **fun** is\_empty(q:queue) **ret** b:Bool  
    b:= (q = [ ])  
**end**  
    {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

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

## Implementación de colas usando arreglos

- **fun** is\_empty(q:queue) **ret** b:Bool  
    b:= (q.size = 0)  
**end**  
    {Post: b = (q ~ vacía)}
- **fun** is\_full(q:queue) **ret** b:Bool  
    b:= (q.size = N)  
**end**
- Todas las operaciones son  $\mathcal{O}(1)$ , salvo enqueue 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.elements[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** is\_full(q:queue) **ret** b:Bool  
    b:= (q.size = N)  
**end**
- Todas las operaciones son  $\mathcal{O}(1)$ .