

Algoritmos y Estructuras de Datos II

TADS: Implementaciones de contadores y pilas

6 de abril de 2016

Clase de hoy

- 1 TAD contador
 - Especificación
 - Interface
 - Implementación
- 2 TAD pila
 - Especificación del TAD pila
 - Interface
 - Implementación
- 3 Listas enlazadas
 - Implementación del TAD pila con listas enlazadas

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?

Especificación del TAD Pila

module TADPila **where**

data Pila e = Vacía
 | Apilar e (Pila e)

es_vacía :: Pila e \rightarrow Bool

primero :: Pila e \rightarrow e

desapilar :: Pila e \rightarrow Pila e

- - las dos últimas se aplican sólo a pila no Vacía

es_vacía Vacía = True

es_vacía (Apilar e p) = False

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 ~ Vacía}

{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 fun
 {Post: b = (p ~ Vacía)}
- Todas las operaciones son $\mathcal{O}(1)$.

Implementación de pilas usando arreglos

Mostrar en

<https://www.cs.usfca.edu/~galles/visualization/StackLL.html>

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

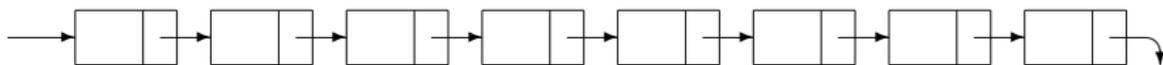
- {Pre: $p \sim P \wedge \neg \text{is_empty}(p)$ }
fun top(p:stack) **ret** e:elem
 e:= p.elems[p.size]
end fun
 {Post: $e \sim \text{primero } P$ }
- {Pre: $p \sim P \wedge \neg \text{is_empty}(p)$ }
proc pop(in/out p:stack)
 p.size:= p.size - 1
end proc
 {Post: $p \sim \text{desapilar } P$ }

Implementación de pilas usando arreglos

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

Listas enlazadas

- Por **listas enlazadas** se entiende una manera de implementar listas utilizando tuplas y punteros.
- Hay diferentes clases de listas, la más simple se representa gráficamente así



- cada **nodo** se dibuja como una tupla
- y la flecha que enlaza un nodo con el siguiente nace desde un campo de esa tupla.
- Los nodos son tuplas y las flechas punteros.

Declaración

- Los nodos son tuplas y las flechas punteros.
- **type** node = **tuple**
 - value: elem
 - next: **pointer to** node**end**
- type** list = **pointer to** node

Observaciones

- Una lista es un puntero a un primer nodo,
- que a su vez contiene un puntero al segundo,
- éste al tercero, y así siguiendo hasta el último,
- cuyo puntero es **null**
- significando que la lista termina allí.
- Para acceder al i -ésimo elemento de la lista, debo recorrerla desde el comienzo siguiendo el recorrido señalado por los punteros.
- Esto implica que el acceso a ese elemento no es constante, sino lineal.
- A pesar de ello ofrecen una manera de implementar convenientemente algunos TADs.

Implementación del TAD pila con listas enlazadas

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

Pila vacía

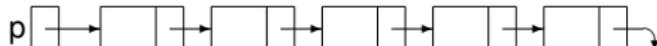
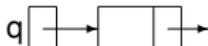
- El procedimiento `empty` inicializa `p` como la pila vacía.
- La pila vacía se implementa con la lista enlazada vacía
- que consiste de la lista que no tiene ningún nodo,
- el puntero al primer nodo de la lista no tiene a quién apuntar.
- Su valor se establece en **null**.

```
proc empty(out p:stack)
    p:= null
end proc
{Post: p ~ Vacía}
```

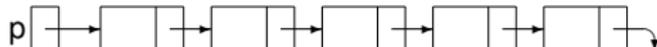
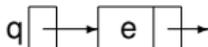
Apilar

 $\{\text{Pre: } p \sim P \wedge e \sim E\}$ **proc** push(**in** e:elem,**in/out** p:stack)**var** q: pointer to node

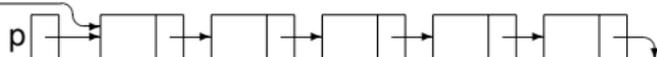
alloc(q)



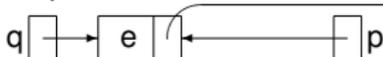
q->value:= e



q->next:= p



p:= q

**end proc** $\{\text{Post: } p \sim \text{Apilar } E \ P\}$

Apilar

Explicación

- El procedimiento push debe alojar un nuevo elemento en la pila.
- Para ello crea un nuevo nodo ($\text{alloc}(q)$),
- aloja en ese nodo el elemento a agregar a la pila ($q \rightarrow \text{value} := e$),
- enlaza ese nuevo nodo al resto de la pila ($q \rightarrow \text{next} := p$)
- y finalmente indica que la pila ahora empieza a partir de ese nuevo nodo que se agregó ($p := q$).

Apilar

En limpio

```
{Pre:  $p \sim P \wedge e \sim E$ }  
proc push(in e:elem,in/out p:stack)  
    var q: pointer to node  
    alloc(q)  
    q→value:= e  
    q→next:= p  
    p:= q  
end proc  
{Post:  $p \sim \text{Apilar } E \ P$ }
```

Importancia de la representación gráfica

- Las representaciones gráficas que acompañan al pseudocódigo son de ayuda.
- Su valor es relativo.
- Sólo sirven para entender lo que está ocurriendo de manera intuitiva.
- Hacer un tratamiento formal está fuera de los objetivos de este curso.
- Deben extremarse los cuidados para no incurrir en errores de programación que son muy habituales en el contexto de la programación con punteros.
- Por ejemplo, ¿es correcto el procedimiento push cuando p es la pila vacía?

Apilar a una pila vacía

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

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

var q: **pointer to node**



alloc(q)



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



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



$p := q$



end proc

{Post: $p \sim \text{Apilar } E \ P$ }

Primero de una pila

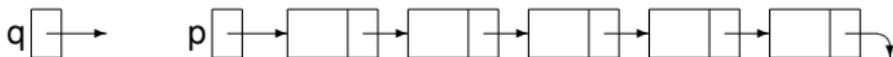
- La función top no tiene más que devolver el elemento que se encuentra en el nodo apuntado por p.
- $\{\text{Pre: } p \sim P \wedge \neg \text{is_empty}(p)\}$
fun top(p:stack) **ret** e:elem
 e:= p→value
end fun
 $\{\text{Post: } e \sim \text{primero } P\}$

Desapilar

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

proc pop(in/out p:stack)

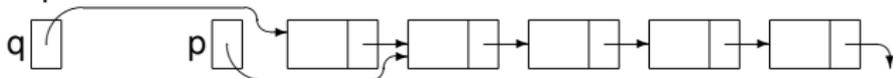
var q: pointer to node



q := p



p := p → next



free(q)



end proc

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

Desapilar

Explicación

- El procedimiento pop debe liberar el primer nodo de la lista
- y modificar p de modo que apunte al nodo siguiente.
- Observar que el valor que debe adoptar p se encuentra en el primer nodo (campo next).
- Por ello, antes de liberarlo es necesario utilizar esa información que se encuentra en él.
- Si modifico el valor de p, ¿cómo voy a hacer luego para liberar el primer nodo que sólo era accesible gracias al viejo valor de p?
- Hay que recordar en q el viejo valor de p ($q := p$),
- hacer que p apunte al segundo nodo ($p := p \rightarrow \text{next}$)
- y liberar el primer nodo ($\text{free}(q)$).
- Al finalizar, p apunta al primer nodo de la nueva pila.

Desapilar

En limpio

```
{Pre:  $p \sim P \wedge \neg \text{is\_empty}(p)$ }  
proc pop(in/out p:stack)  
    var q: pointer to node  
    q:= p  
    p:= p→next  
    free(q)  
end proc  
{Post:  $p \sim \text{desapilar } P$ }
```

P no puede ser vacía.

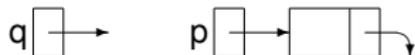
Pero ¿qué pasa si tiene un solo elemento?

Desapilar de una pila unitaria

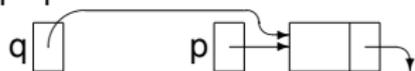
{Pre: $p \sim P \wedge \neg \text{is_empty}(p)$ }

proc pop(in/out p:stack)

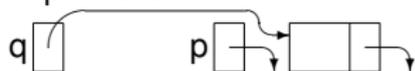
var q: pointer to node



q := p



p := p → next



free(q)



end proc

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

Examinar si es vacía

- La función `is_empty` debe comprobar que la pila recibida esté vacía, que se representa por el puntero **null**.

- {Pre: $p \sim P$ }

```
fun is_empty(p:stack) ret b:Bool
```

```
    b:= (p = null)
```

```
end fun
```

```
{Post: b ~ es_vacía P}
```

Destrucción de la pila

- Como el manejo de la memoria es explícito, es conveniente agregar una operación para destruir una pila.
- Esta operación recorre la lista enlazada liberando todos los nodos que conforman la pila.
- Puede definirse utilizando las operaciones proporcionadas por la implementación del TAD pila.
- **proc** destroy(**in/out** p:stack)
 while \neg is_empty(p) **do** pop(p) **od**
end proc

Conclusiones

- Todas las operaciones (salvo destroy) son constantes.
- Destroy es lineal.
- stack y **pointer to** node son sinónimos,
- pero las hemos usado diferente:
 - stack, cuando la variable representa una pila,
 - **pointer to** node cuando se trata de un puntero que circunstancialmente aloja la dirección de un nodo.