Algoritmos y Estructuras de Datos II

TADS: Implementaciones elementales

13 de abril de 2015

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. Listas enlazadas
   - Implementación del TAD pila con listas enlazadas

Repaso

- cómo vs. qué
- 3 partes
  1. análisis de algoritmos
     - algoritmos de ordenación
     - notación $\mathcal{O}$, Ω y Θ.
     - 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)
     - hoy: implementándolos
  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 $\longrightarrow$ TAD contador
    - delimitadores balanceados $\longrightarrow$ TAD pila

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 $\sim$ Inicial}

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

{Pre: c $\sim$ C $\wedge\neg$is_init(c)}
proc dec (**in**/**out** c: counter)
{Post: c $\sim$ decrementar C}

fun is_init (c: counter) **ret** b: **bool** {Post: b $=$ (c $\sim$ 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 $\sim$ Inicial}
- {Pre: c $\sim$ C}
  proc inc (**in**/**out** c: counter)
  c:= c+1
  end proc
  {Post: c $\sim$ Incrementar C}

Implementación natural

- {Pre: c $\sim$ C $\wedge \neg$is_init(c)}
  proc dec (**in**/**out** c: counter)
  c:= c-1
  end proc
  {Post: c $\sim$ decrementar C}

- **fun** is_init (c: counter) **ret** b: **bool**
  b:= (c = 0)
  end fun
  {Post: b $=$ (c $\sim$ 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?

Especificacion del TAD Pila

```
module TADPila where

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

es_vacía :: Pila e → Bool
primero :: Pila e → e
desapilar :: Pila e -> 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 $\sim$ Vacía}

{Pre: p $\sim$ P $\wedge$ e $\sim$ E}
proc push(**in** e:elem,**in**/**out** p:stack)
{Post: p $\sim$ Apilar E P}

{Pre: p $\sim$ P $\wedge\neg$is_empty(p)}
fun top(p:stack) **ret** e:elem
{Post: e $\sim$ primero P}

Interface

{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
{Post: p $\sim$ desapilar P}

fun is_empty(p:stack) **ret** b:**bool**
{Post: b $=$ (p $\sim$ 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 $\sim$ Vacía}
- {Pre: p $\sim$ P }
  proc push(**in** e:elem,**in/out** p:stack)
  p:= (e $\triangleright$ p)
  end proc
  {Post: p $\sim$ Apilar e P}

Implementación de pilas usando tipo concreto lista

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

Implementación de pilas usando tipo concreto lista

- **fun** is_empty(p:stack) **ret** b:**Bool**
  b:= (p = [ ])
  end fun
  {Post: b $=$ (p $\sim$ 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 $\sim$ Vacía}
- {Pre: p $\sim$ 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 $\sim$ Apilar e P}

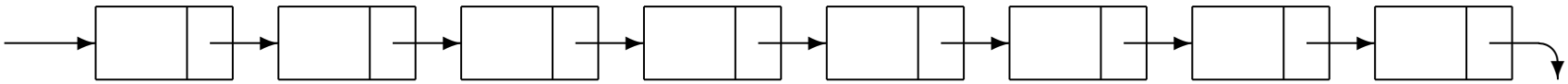
Implementación de pilas usando arreglos

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

Implementación de pilas usando arreglos

- **fun** is_empty(p:stack) **ret** b:**Bool**
  b:= (p.size = 0)
  end fun
  {Post: b $=$ (p $\sim$ 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
```

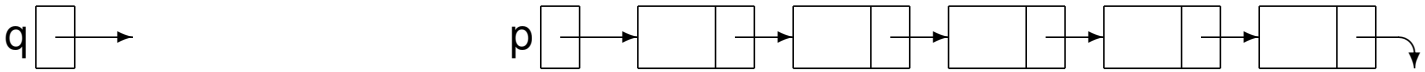
Pila vacía

- 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**.

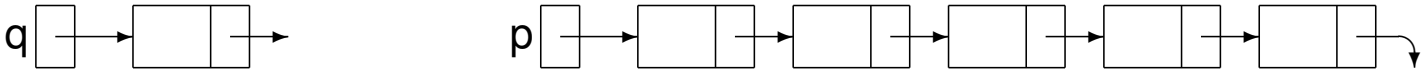
proc empty(**out** p:stack)
    p:= **null**
end proc
{Post: p $\sim$ Vacía}

Apilar

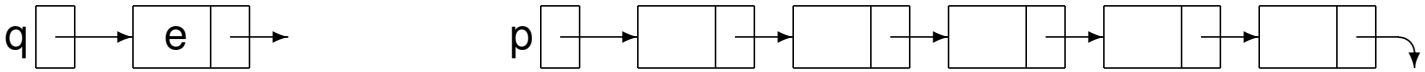
{Pre: p $\sim$ P $\wedge$ e $\sim$ E}
proc push(**in** e:elem,**in/out** p:stack)
var q: **pointer to** node



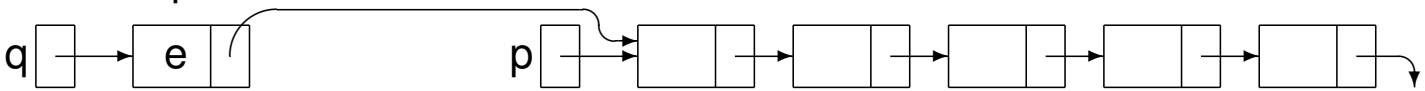
alloc(q)



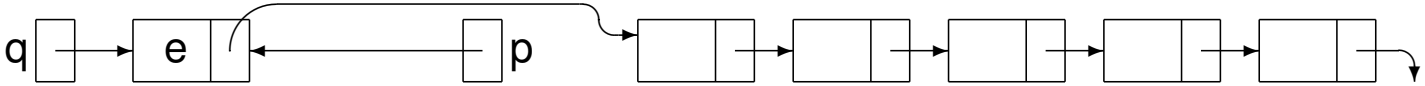
q$\rightarrow$value:= e



q$\rightarrow$next:= p



p:= q



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

Apilar
Explicación

- El procedimiento push debe alojar un nuevo elemento en la pila.
- Para ello crea un nuevo nodo (alloc(q)),
- aloja en ese nodo el elemento a agregar a la pila (q$\rightarrow$value:= e),
- enlaza ese nuevo nodo al resto de la pila (q$\rightarrow$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$\rightarrow$value:= e
    q$\rightarrow$next:= p
    p:= q
end proc
{Post: p $\sim$ 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?

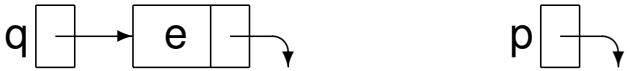
Apilar a una pila vacía

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

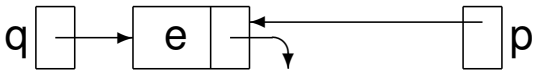
alloc(q)

q$\rightarrow$value:= e

q$\rightarrow$next:= p



p:= q



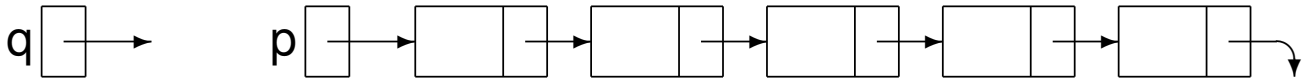
end proc
{Post: p $\sim$ Apilar E P}

Primero de una pila

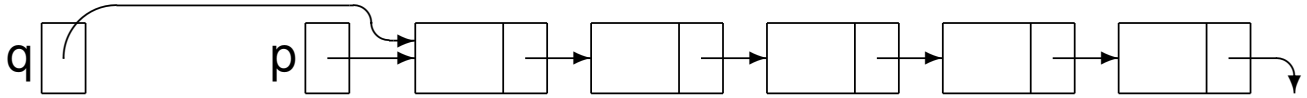
- La función top no tiene más que devolver el elemento que se encuentra en el nodo apuntado por p.
- {Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
  fun top(p:stack) **ret** e:elem
  e:= p$\rightarrow$value
  end fun
  {Post: e $\sim$ primero P}

Desapilar

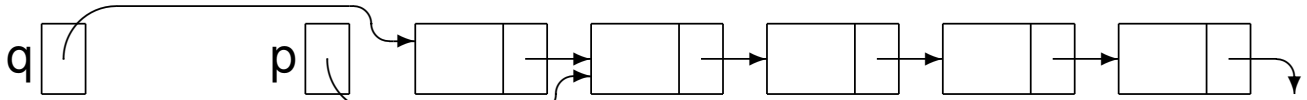
{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
var q: **pointer to** node



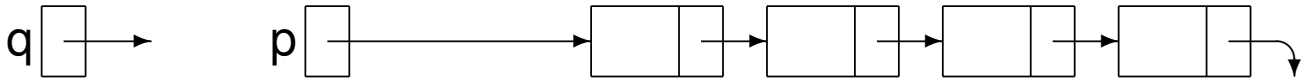
q:= p



p:= p$\rightarrow$next



free(q)



end proc
{Post: p $\sim$ 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$\rightarrow$next)
- y liberar el primer nodo (free(q)).
- Al finalizar, p apunta al primer nodo de la nueva pila.

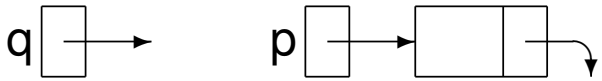
Desapilar
En limpio

{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
    var q: **pointer to** node
    q:= p
    p:= p$\rightarrow$next
    free(q)
end proc
{Post: p $\sim$ desapilar P}

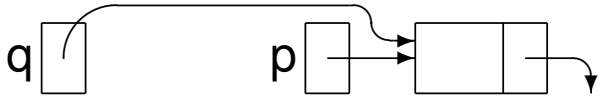
P no puede ser vacía.
Pero ¿qué pasa si tiene un solo elemento?

Desapilar de una pila unitaria

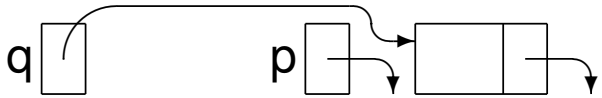
{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
var q: **pointer to** node



q:= p



p:= p$\rightarrow$next



free(q)

end proc
{Post: p $\sim$ 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 $\sim$ 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 ¬ 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.