Algoritmos y Estructuras de Datos II

TADS: Implementaciones de pilas y colas

17 de abril de 2017

Clase de hoy

1. Implementación del TAD pila
    - Usando listas
    - Usando arreglos
    - Usando listas enlazadas
2. Implementación del TAD cola
    - Usando listas
    - Usando arreglos ingenuamente
    - Implementación eficiente de colas usando arreglos
    - Usando listas enlazadas ingenuamente
    - Usando listas enlazadas y dos punteros
    - Usando listas enlazadas circulares

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 tres implementaciones:

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

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/StackArray.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}

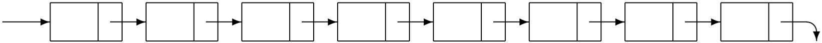
Implementación de pilas usando arreglos

- $\{$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**

```
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 de pilas usando listas enlazadas

Mostrar en
https://www.cs.usfca.edu/~galles/visualization/StackLL.html

Implementación del TAD pila con listas enlazadas

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

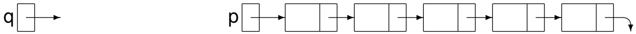
Pila vacía

- 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**.

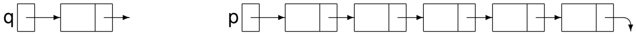
proc empty(**out** p:stack)
    p:= **null**
end proc
$\{$Post: p $\sim$ Vacía$\}$

Apilar

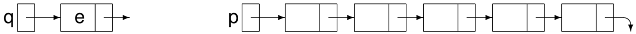
{Pre: $p \sim P \wedge e \sim E$}
proc push(**in** e:elem,**in**/**out** p:stack)
    var q: **pointer to** node



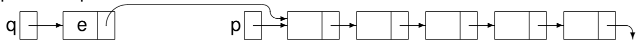
alloc(q)



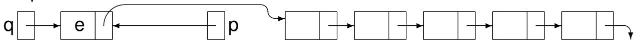
q→value:= e



q→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→value:= e),
- enlaza ese nuevo nodo al resto de la pila (q→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?

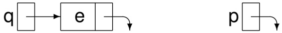
Apilar a una pila vacía

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

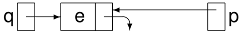
    alloc(q)

    q$\rightarrow$value:= e

    q$\rightarrow$next:= p



    p:= q



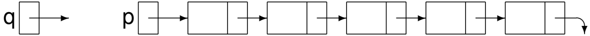
end proc
{Post: p $\sim$ Apilar E P}

Primero de una pila

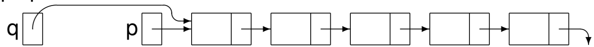
- La función top no tiene más que devolver el elemento que se encuentra en el nodo apuntado por p.
- {Pre: p $\sim$ P $\wedge\neg$is_empty(p)}
  fun top(p:stack) **ret** e:elem
      e:= p$\rightarrow$value
  end fun
  {Post: e $\sim$ primero P}

Desapilar

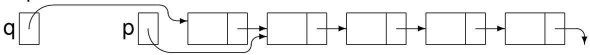
{Pre: p $\sim$ P $\wedge\neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
    var q: **pointer to** node



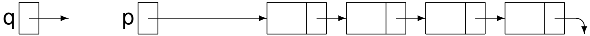
    q:= p



    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 guardar ese valor.
- Si lo asignamos a p, p pierde su viejo valor ¿cómo vamos a liberar luego el primer nodo?
- Solución: recordamos 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

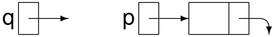
En limpio

{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}

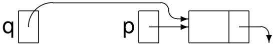
P no puede ser vacía.
Pero ¿qué pasa si tiene un solo elemento?

Desapilar de una pila unitaria

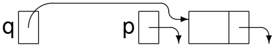
{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
    var q: **pointer to** node



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.

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
```

Interface

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

proc empty(**out** q:queue) {Post: q $\sim$ Vacía}

{Pre: q $\sim$ Q $\wedge$ e $\sim$ E}
proc enqueue(**in**/**out** q:queue,**in** e:elem)
{Post: q $\sim$ Encolar Q E}

{Pre: q $\sim$ Q $\wedge \neg$is_empty(q)}
fun first(q:queue) **ret** e:elem
{Post: e $\sim$ primero Q}

Interface

{Pre: q $\sim$ Q $\wedge\neg$is_empty(q)}
proc dequeue(**in**/**out** q:queue)
{Post: q $\sim$ decolar Q}

fun is_empty(q:queue) **ret** b:**bool**
{Post: b $=$ (q $\sim$ Vacía)}

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 $\sim$ Vacía$\}$
- $\{$Pre: q $\sim$ Q $\wedge$ e $\sim$ E$\}$
  proc enqueue(**in**/**out** q:queue; **in** e:elem)
  q:= (q $\triangleleft$ e)
  end proc
  $\{$Post: q $\sim$ Encolar Q E$\}$

Implementación de colas usando tipo concreto lista

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

Implementación de colas usando tipo concreto lista

- **fun** is_empty(q:queue) **ret** b:**Bool**
  b:= (q = [ ])
  end fun
  $\{$Post: b $=$ (q $\sim$ 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 $\sim$ Vacía}
- {Pre: q $\sim$ Q $\wedge$ e $\sim$ E $\wedge \neg$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 $\sim$ Encolar Q E}

Implementación de colas usando arreglos

- {Pre: q $\sim$ Q $\wedge\neg$is_empty(q)}
  fun first(q:queue) **ret** e:elem
      e:= q.elems[1]
  end fun
  {Post: e $\sim$ primero Q}
- {Pre: q $\sim$ Q $\wedge\neg$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
  {Post: q $\sim$ decolar Q}

Implementación de colas usando arreglos

- **fun** is_empty(q:queue) **ret** b:**Bool**
  b:= (q.size = 0)
  end fun
  $\{$Post: b $=$ (q $\sim$ 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 de colas usando arreglos

Mostrar en
https://www.cs.usfca.edu/~galles/visualization/QueueArray.html

Aunque la que hacemos acá no es exactamente la misma.

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

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

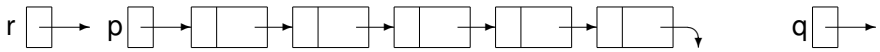
Sólo cambia la implementación de enqueue:

```
{Pre: p ∼ 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) 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 ∼ Encolar Q E}
```

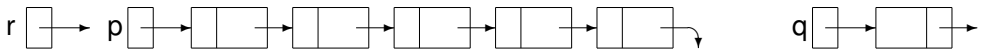
Implementación del TAD cola con listas enlazadas

Implementación ingenua

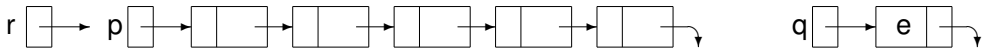
{Pre: p ∼ Q ∧ e ∼ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
  var q,r: **pointer to** node



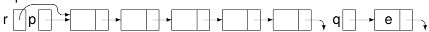
  alloc(q)



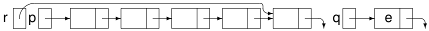
  q→value:= e
  q→next:= **null**



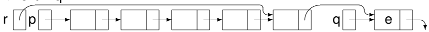
  r:= p



  while r→next ≠ **null do**   {mientras ⋆r no sea el último nodo}
    r:= r→next   {que r pase a señalar el nodo siguiente}
  od



  r→next:= q



end proc
{Post: p ∼ 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 ≠ 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

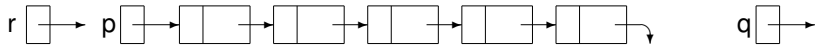
Implementación ingenua (corregida)

```
{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}
    if p = null → p:= q                          {si la cola es vacía con esto alcanza}
      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}
```

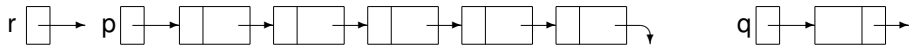
Implementación del TAD cola con listas enlazadas

Implementación ingenua (corregida)

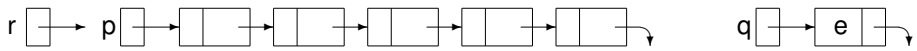
{Pre: p $\sim$ Q $\wedge$ e $\sim$ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
  var q,r: **pointer to** node



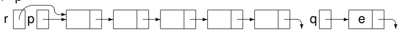
  alloc(q)



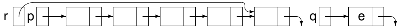
  q$\to$value:= e
  q$\to$next:= **null**



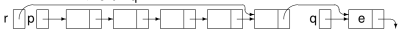
  if p = **null** $\to$ p:= q {no engañarse con el dibujo, la cola puede ser vacía}
   p $\neq$ **null** $\to$ r:= p



    while r$\to$next $\neq$ **null** **do** {mientras $\star$r no sea el último nodo}
      r:= r$\to$next {que r pase a señalar el nodo siguiente}
    od



    r$\to$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**.

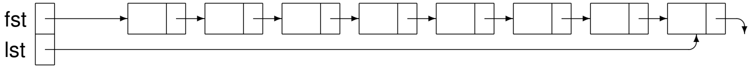
Implementación de colas usando listas enlazadas (con dos punteros)

Mostrar en
https://www.cs.usfca.edu/~galles/visualization/QueueLL.html

Implementación del TAD cola con listas enlazadas y dos punteros

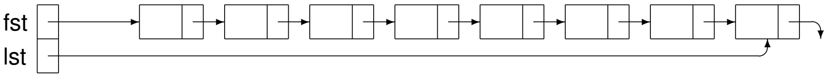
type node = **tuple**
  value: elem
  next: **pointer to** node
 end
type queue = **tuple**
  fst: **pointer to** node
  lst: **pointer to** node
 end

Gráficamente, puede representarse de la siguiente manera



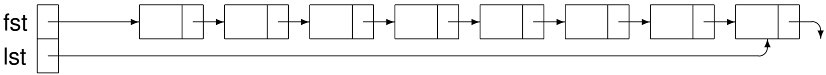
Cola vacía

proc empty(**out** p:queue)
    p.fst:= **null**
    p.lst:= **null**
end proc
{Post: p $\sim$ Vacía}



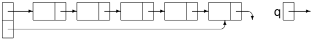
Primer elemento

{Pre: p $\sim$ Q $\wedge \neg$is_empty(p)}
fun first(p:queue) **ret** e:elem
    e:= p.fst$\rightarrow$value
end fun
{Post: e $\sim$ primero Q}

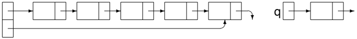


Encolar

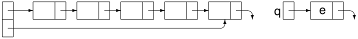
{Pre: p $\sim$ Q $\wedge$ e $\sim$ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
    var q: **pointer to** node



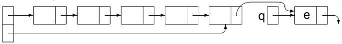
    alloc(q)



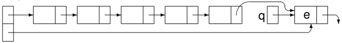
    q$\rightarrow$value:= e
    q$\rightarrow$next:= **null**



    if p.lst = **null** $\rightarrow$ p.fst:= q   {caso enqueue en cola vacía}
                                p.lst:= q
       p.lst $\neq$ **null** $\rightarrow$ p.lst$\rightarrow$next:= q



    p.lst:= q



    fi
end proc
{Post: p $\sim$ Encolar Q E}

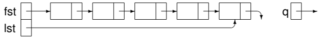
Encolar

En limpio

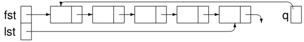
```
proc enqueue(in/out p:queue,in e:elem)
    var q: pointer to node
    alloc(q)
    q→value:= e
    q→next:= null
    if p.lst = null → p.fst:= q
                      p.lst:= q
       p.lst ≠ null → p.lst→next:= q
                      p.lst:= q
    fi
end proc
```

Decolar

{Pre: p ∼ Q ∧¬is_empty(p)}
proc dequeue(**in**/**out** p:queue)
  var q: **pointer to** node

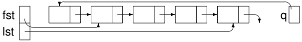


  q:= p.fst

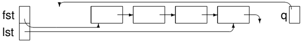


  if p.fst = p.lst →
    p.fst:= **null**   {caso cola con un solo elemento}
    p.lst:= **null**

   p.fst ≠ p.lst → p.fst:= p.fst->next



  fi
  free(q)



end proc
{Post: p ∼ 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 $\sim$ es_vacía Q}

Destroy

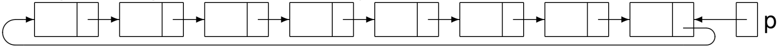
proc destroy(**in**/**out** p:queue)
    while ¬ is_empty(p) **do** dequeue(p) **od**
end proc

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

Implementación del TAD cola con listas enlazadas circulares

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

Gráficamente, puede representarse de la siguiente manera



Explicación

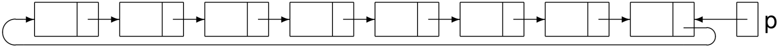
- La lista es circular,
- es decir que además de los punteros que ya teníamos en implementaciones anteriores,
- el último nodo tiene un puntero al primero,
- alcanza con saber dónde se encuentra el último nodo para saber también dónde está el primero.

Cola vacía

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

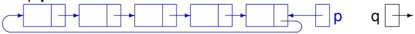
Primer elemento

{Pre: p $\sim$ Q $\wedge\neg$is_empty(p)}
fun first(p:queue) **ret** e:elem
    e:= p$\rightarrow$next$\rightarrow$value
end fun
{Post: e $\sim$ primero Q}

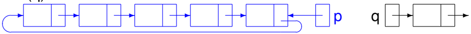


Encolar: caso cola no vacía (en azul, la cola)

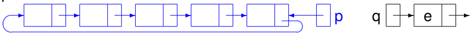
{Pre: p $\sim$ Q $\wedge$ e $\sim$ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
    var q: **pointer to** node



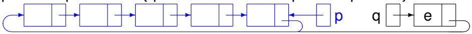
    alloc(q)



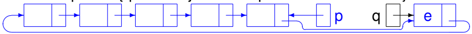
    q$\rightarrow$value:= e



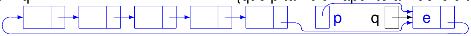
    if p = **null** $\rightarrow$ q$\rightarrow$next:= q   {caso enqueue en cola vacía}
      p $\neq$ **null** $\rightarrow$ q$\rightarrow$next:= p$\rightarrow$next   {que el nuevo último apunte al primero}



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



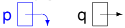
    fi
    p:= q   {que p también apunte al nuevo último}



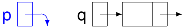
    end proc
    {Post: p $\sim$ Encolar Q E}

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

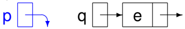
{Pre: p $\sim$ Q $\wedge$ e $\sim$ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
    var q: **pointer to** node



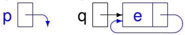
    alloc(q)



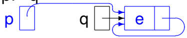
    q$\rightarrow$value:= e



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



       p $\neq$ **null** $\rightarrow$ q$\rightarrow$next:= p$\rightarrow$next   {caso enqueue en cola no vacía}
                p$\rightarrow$next:= q
    fi
    p:= q



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

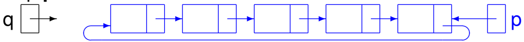
Encolar

En limpio

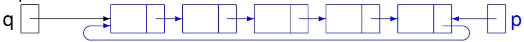
proc enqueue(**in**/**out** p:queue,**in** e:elem)
    var q: **pointer to** node
    alloc(q)
    q→value:= e
    if p = **null** → q→next:= q
      p ≠ **null** → q→next:= p→next
                  p→next:= q
    fi
    p:= q
end proc

Decolar: caso cola con más de un elemento

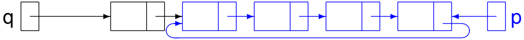
{Pre: p $\sim$ Q $\land \neg$is_empty(p)}
proc dequeue(**in/out** p:queue)
  var q: **pointer to** node



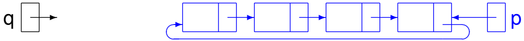
  q:= p$\rightarrow$next



  if p = q $\rightarrow$ p:= **null** {caso cola con un solo elemento}
  p $\neq$ q $\rightarrow$ p$\rightarrow$next:= q$\rightarrow$next {caso cola con más de un elemento}
  fi



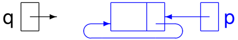
  free(q)



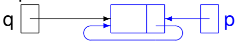
end proc
{Post: p $\sim$ decolar Q}

Decolar: caso cola con un solo elemento

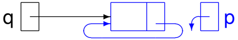
{Pre: p $\sim$ Q $\wedge\neg$is_empty(p)}
proc dequeue(**in**/**out** p:queue)
    var q: **pointer to** node



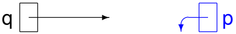
    q:= p$\rightarrow$next



    if p = q $\rightarrow$ p:= **null**   {caso cola con un solo elemento}
      p $\neq$ q $\rightarrow$ p$\rightarrow$next:= q$\rightarrow$next   {caso cola con más de un elemento}
    fi



    free(q)



end proc
{Post: p $\sim$ decolar Q}

Examinar si es vacía

{Pre: p $\sim$ Q}
fun is_empty(p:queue) **ret** b:**Bool**
    b:= (p = **null**)
end fun
{Post: b $\sim$ es_vacía Q}

Destroy

proc destroy(**in**/**out** p:queue)
  while ¬ is_empty(p) **do** dequeue(p) **od**
end proc

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

Algoritmos y Estructuras de Datos II

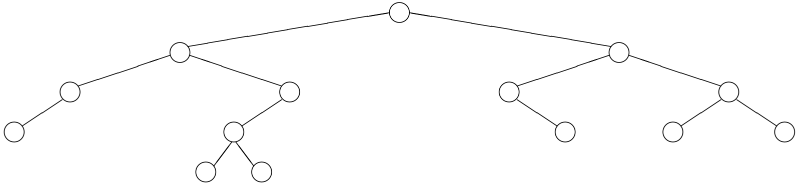
Árboles binarios

19 de abril de 2017

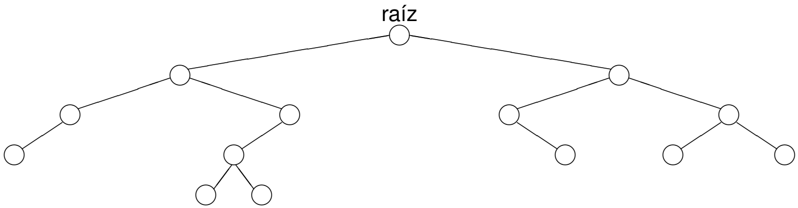
Clase de hoy

1. Árboles Binarios
   - Intuición
   - Especificación
   - Terminología habitual
   - Implementación con punteros
   - Posiciones

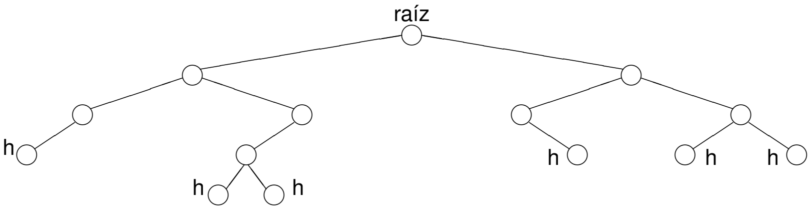
Intuición



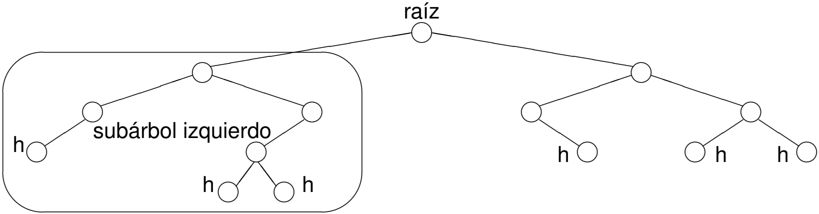
Intuición



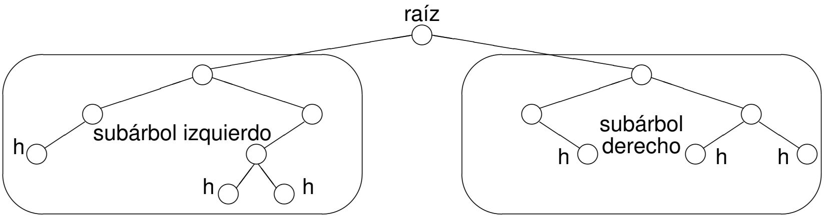
Intuición



Intuición



Intuición



- De cada circulo, cuelgan dos subárboles.
- En el caso de las hojas, los subárboles son vacíos.
- Todos los árboles pueden construirse con los constructores
  - $<>$, que construye un árbol vacío
  - $<\_,\_,\_>$, que construye un árbol no vacío a partir de un elemento y dos subárboles

Especificación del TAD árbol binario

module TADÁrbolBinario **where**

data ÁrbolBinario e = $<>$
$\quad\quad\quad\quad\quad\quad\quad\quad$ | $< \_, \_, \_ >$ (ÁrbolBinario e) e (ÁrbolBinario e)

es_vacío :: ÁrbolBinario e $\rightarrow$ Bool
raíz :: ÁrbolBinario e $\rightarrow$ e
izquierdo :: ÁrbolBinario e $\rightarrow$ ÁrbolBinario e
derecho :: ÁrbolBinario e $\rightarrow$ ÁrbolBinario e
- - las tres últimas se aplican sólo a árbol no vacío

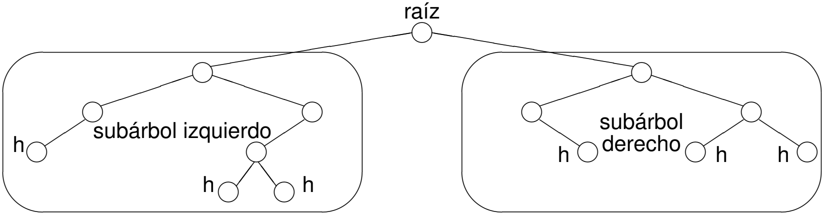
es_vacío $<>$ = True
es_vacío $<$i,r,d$>$ = False

raíz $<$i,r,d$>$ = r
izquierdo $<$i,r,d$>$ = i
derecho $<$i,r,d$>$ = d

Notación $<>$

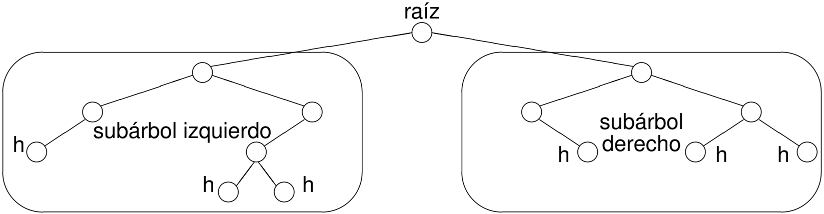
- Notar la sobrecarga de la notación $<>$:
- cuando no tiene ningún argumento, $<>$ es el árbol vacío,
- cuando tiene tres argumentos, $<$i,r,d$>$ es el árbol no vacío cuya raíz es *r*, subárbol izquierdo es *i* y subárbol derecho es *d*.
- Una hoja es un árbol de la forma $<<>$,r, $<>>$. Se la abrevia $<$r$>$.
- Conclusión: la notación $<>$ puede tener 0, 1 ó 3 argumentos.

Botánica y genealogía



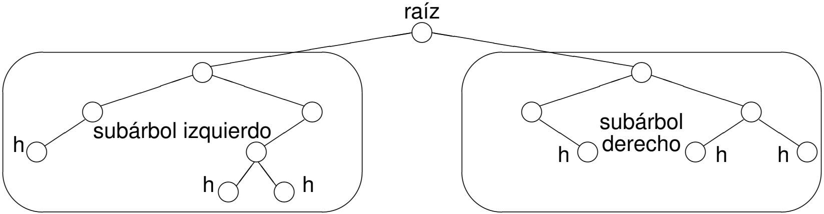
- Un **nodo** es un árbol no vacío.
- Tiene **raíz**, **subárbol izquierdo** y **subárbol derecho**.
- A los subárboles se los llama también **hijos** (izquierdo y derecho).
- Y al nodo se le dice **padre** de sus hijos.
- Una **hoja** es un nodo con los dos hijos vacíos.

Más genealogía



- Dos **hermanos** son hijo izquierdo y derecho del mismo padre.
- Un **camino** es una secuencia $A_1, \ldots, A_n$ donde cada árbol es hijo del anterior.
- A es **ancestro** de B sii hay un camino (de longitud ≥ 0) de A a B,
- en ese caso B es **descendiente** o **subárbol** de A
- un camino se puede identificar con un recorrido descendente del árbol.

Altura, profundidad, nivel



- Decimos que siempre hay un camino de longitud 0 de un árbol a sí mismo.
- La **altura** de un árbol es la longitud del camino que va desde él hasta el subárbol vacío más lejano. Según otras definiciones, altura es la longitud del camino que va desde él hasta la hoja más lejana.
- La **profundidad** de un subárbol es la longitud del camino que va desde el árbol hasta dicho subárbol.
- Se llama **nivel** al conjunto de los subárboles de igual profundidad.

Implementación con punteros

```
type node = tuple
                 lft: pointer to node
                 value: elem
                 rgt: pointer to node
             end
type bintree = pointer to node

fun empty() ret t:bintree
    t:= null
end
{Post: t ~<> }
```

Implementación con punteros

```
{Pre: l ∼ L ∧ e ∼ E ∧ r ∼ R}
fun node(l:bintree,e:elem,r:bintree) ret t:bintree
    alloc(t)
    t→lft:= l
    t→value:= e
    t→rgt:= r
{Post: t ∼ <L,E,R>} end

{Pre: t ∼ T ∧¬ is_empty(t)}
fun root(t:bintree) ret e:elem
    e:= t→value
end
{Post: e ∼ raíz T}
```

Implementación con punteros

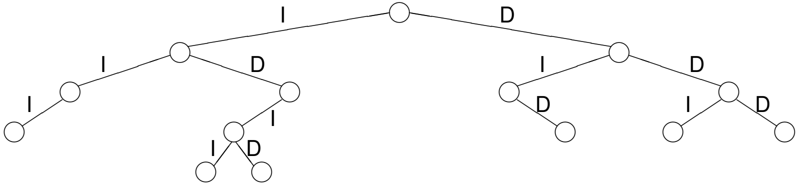
{Pre: t $\sim$ T $\wedge \neg$ is_empty(t)}
fun left(t:bintree) **ret** l:bintree
    l:= t$\rightarrow$lft
end
{Post: l $\sim$ izquierdo T}

{Pre: t $\sim$ T $\wedge \neg$ is_empty(t)}
fun right(t:bintree) **ret** r:bintree
    r:= t$\rightarrow$rgt
end
{Post: r $\sim$ derecho T}

Implementación con punteros

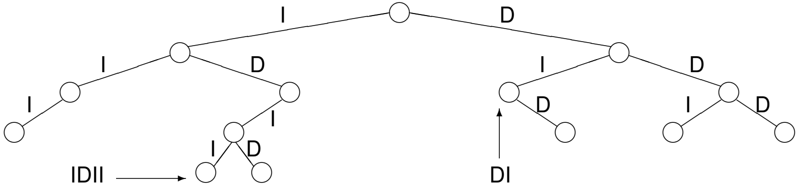
```
{Pre: t ~ T}
fun is_empty(t:bintree) ret b:bool
    b:= (t = null)
end
{Post: b ~ es_vacío(T)}

proc destroy(in/out t:bintree)
      if ¬ is_empty(t) then destroy(left(t))
                            destroy(right(t))
                            free(t)
                            t:= null
      fi
end
```

Indicaciones o posiciones



- A cada arista que conecta un padre con su hijo se la rotula I si es con el hijo izquierdo y D si es el derecho,
- Este I ó D puede entenderse como dando **indicaciones**
- I es ir a la izquierda
- D es ir a la derecha

Posiciones



- Una secuencia de I's y D's sirve para desplazarse desde la raíz hacia las hojas.
- Cada subárbol queda señalado por una secuencia de I's y D's.
- Estas secuencias de I's y D's marcan **posiciones** dentro del árbol.
- El TAD Posición es el tipo de tales secuencias.

TAD Posición

module TADPosición **where**

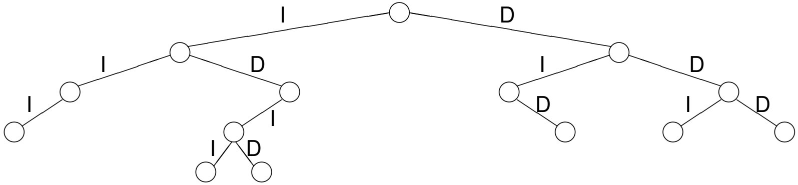
data Posición = R
| I Posición
| D Posición

_ ↓ _ :: ÁrbolBinario e → Posicion → ÁrbolBinario e
_._ :: ÁrbolBinario e → Posicion → e
- - se aplica solo si la posición cae en el árbol (o sea, si t↓p ≠ <>)

t↓p | es_vacío(t) = <>
t↓R = t
t↓(I p) = (izquierdo t)↓p
t↓(D p) = (derecho t)↓p

t.p = raíz (t↓p)

Selección de subárbol

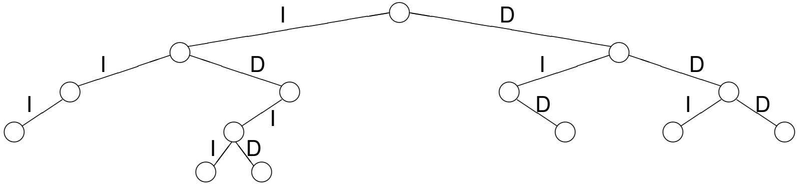


Dado un árbol t y una posición $p \in pos$, $t \downarrow p$ es el subárbol de t que se encuentra en la posición p:

$<> \downarrow p = <>$
$< i, e, d > \downarrow$ R $= < i, e, d >$
$< i, e, d > \downarrow$ (I p) $= i \downarrow p$
$< i, e, d > \downarrow$ (D p) $= d \downarrow p$

Se define $pos(t) = \{p \in pos \mid t \downarrow p \neq <>\}$. Es el conjunto de las posiciones del árbol binario t.

Selección de elemento



Dado un árbol t y una posición $p \in pos(t)$, $t.p$ es el elemento de t que se encuentra en la posición p:

$< i, e, d >$ .R = e
$< i, e, d >$.(I p) = i.p
$< i, e, d >$.(D p) = d.p

o equivalentemente $t.p = raiz\ (t \downarrow p)$.