Algoritmos y Estructuras de Datos II

Listas enlazadas

14 de abril de 2014

Clase de hoy

1. Repaso
   - TAD contador
   - TAD pila
   - TAD cola
2. Listas enlazadas
   - Implementación del TAD pila con listas enlazadas
   - Implementación del TAD cola con listas enlazadas
   - Implementación de colas con listas enlazadas y dos punteros
   - Implementación del TAD cola con listas enlazadas circulares
   - Implementación del TAD pcola 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, TAD cola, TAD pcola)
     - implementaciones elementales
     - hoy: implementaciones utilizando listas enlazadas
  3. técnicas de resolución de problemas

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 $\sim$ 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

- **type** counter = **nat**
- **proc** init (**out** c: counter)
  c:= 0
  end proc
- **proc** inc (**in**/**out** c: counter)
  c:= c+1
  end proc
- **proc** dec (**in**/**out** c: counter)
  c:= c-1
  end proc
- **fun** is_init (c: counter) **ret** b: **bool**
  b:= (c = 0)
  end fun
- Todas las operaciones son $\mathcal{O}(1)$

Especificacion 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 $\sim$ vacia}

{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

Vimos dos implementaciones:

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

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
- {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

Implementación de pilas usando arreglos

- **fun** top(p:stack) **ret** e:elem
  e:= p.elems[p.size]
  end fun
- **proc** pop(**in**/**out** p:stack)
  p.size:= p.size - 1
  end proc
- **fun** is_empty(p:stack) **ret** b:**Bool**
  b:= (p.size = 0)
  end fun
- **fun** is_full(p:stack) **ret** b:**Bool**
  b:= (p.size = N)
  end fun
- 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 {se aplica sólo a una cola no vacía}
decolar : cola $\rightarrow$ cola {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 $\sim$ vacia}

{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

Vimos implementaciones:

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

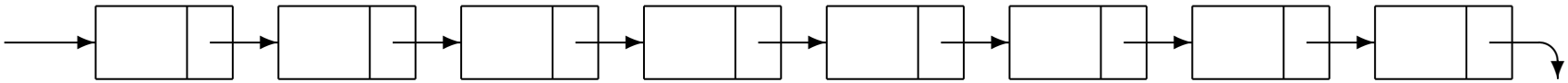
Implementación eficiente de colas usando arreglos

- **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
- {Pre: q $\sim$ Q $\wedge \neg$is_full(q)}
  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
- 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
- 
- La representación es parecida a la de cola,
- la diferencia es que 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**.

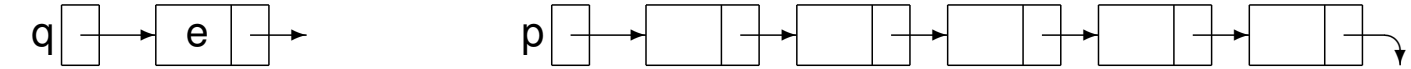
proc empty(**out** p:stack)
    p:= **null**
end proc
{Post: p $\sim$ vacía}

Apilar

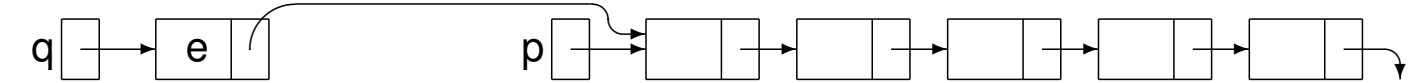
{Pre: p $\sim$ P $\wedge$ e $\sim$ E}
proc push(**in** e:elem,**in**/**out** p:stack)
var q: **pointer to** node

alloc(q)

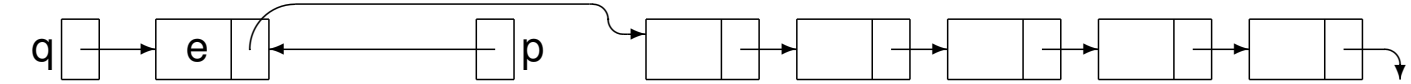
q$\rightarrow$value:= e



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)}

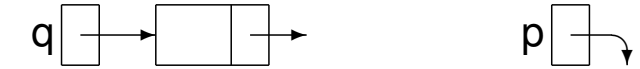
Importancia de la representación gráfica

- 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?

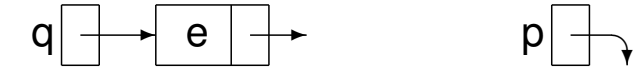
Apilar a una pila vacía

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

alloc(q)

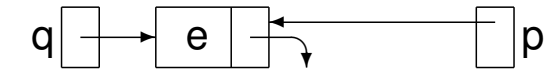


q$\rightarrow$value:= e



q$\rightarrow$next:= p

p:= q



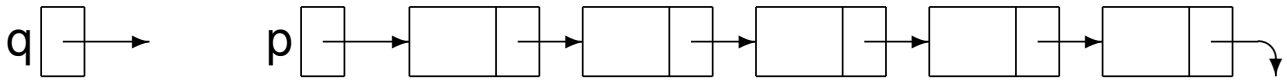
end proc
{Post: p $\sim$ apilar(E,P)}

Primero de una pila

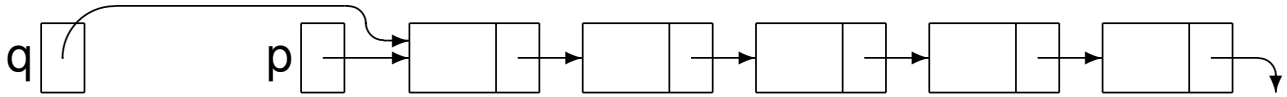
- La función top no tiene más que devolver el elemento que se encuentra en el nodo apuntado por p.
- {Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
  fun top(p:stack) **ret** e:elem
  e:= p$\rightarrow$value
  end fun
  {Post: e $\sim$ primero(P)}

Desapilar

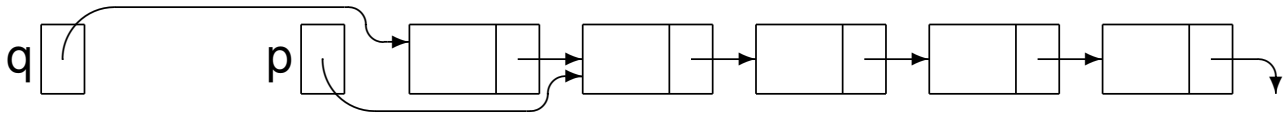
{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
var q: **pointer to** node



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.

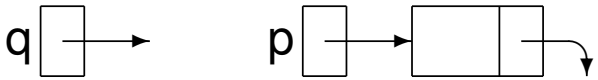
Desapilar
En limpio

{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)}

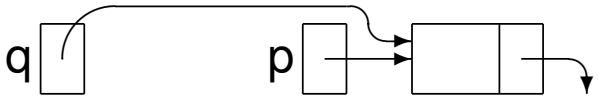
P no puede ser vacía.
Pero ¿qué pasa si tiene un solo elemento?

Desapilar de una pila unitaria

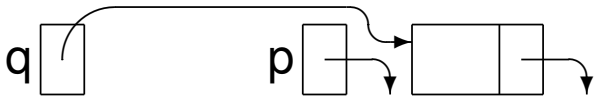
{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
var q: **pointer to** node



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.

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 destroye como para el TAD pila,
- first como top,
- y dequeue como pop.

Implementación del TAD cola con listas enlazadas

Implementación ingenua

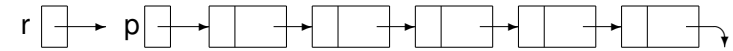
Sólo cambia la implementación de enqueue:

```
{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) 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)}
```

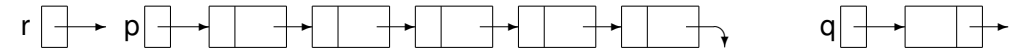
Implementación del TAD cola con listas enlazadas

Implementación ingenua

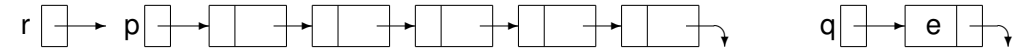
{Pre: p $\sim$ Q $\wedge$ e $\sim$ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
var q,r: **pointer to** node



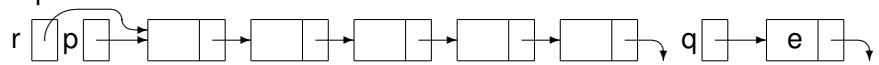
alloc(q)



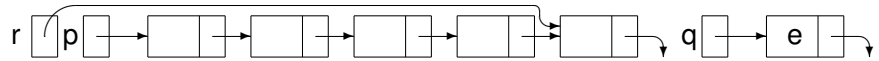
q$\rightarrow$value:= e
q$\rightarrow$next:= **null**



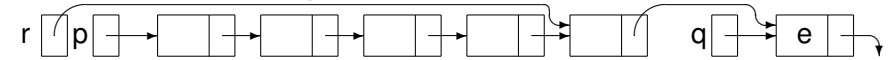
if p = **null** $\rightarrow$ p:= q {no engañarse con el dibujo, la cola puede ser vacía}
p $\neq$ **null** $\rightarrow$ r:= p



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



r$\rightarrow$next:= q



fi
end proc
{Post: p $\sim$ 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
    if p = null → p:= q
       p ≠ null → r:= p
                  while r→next ≠ null do
                      r:= r→next
                  od
                  r→next:= q
    fi
end proc
```

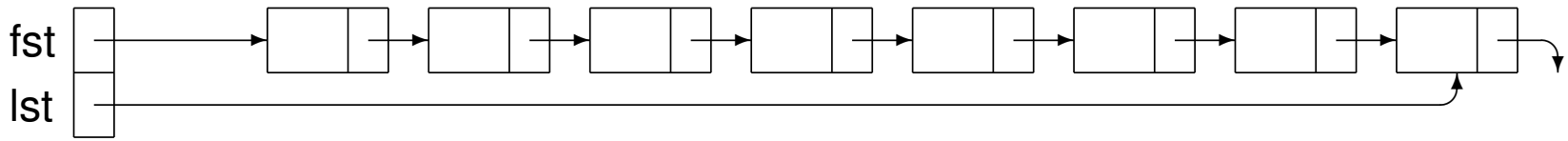
Conclusiones

- 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 del TAD cola con listas enlazadas y dos punteros

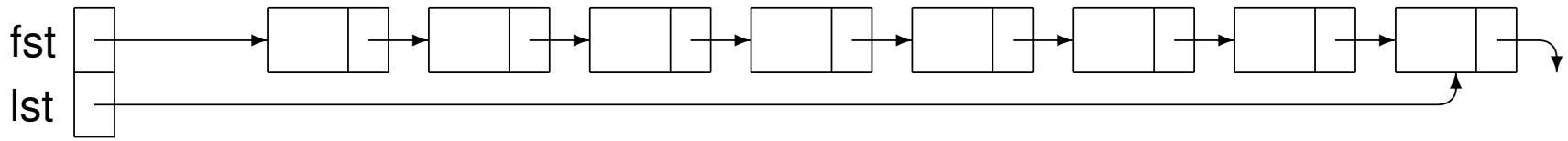
```
type node = tuple
              value: elem
              next: pointer to node
            end
type queue = tuple
               fst: pointer to node
               lst: pointer to node
             end
```

Gráficamente, puede representarse de la siguiente manera



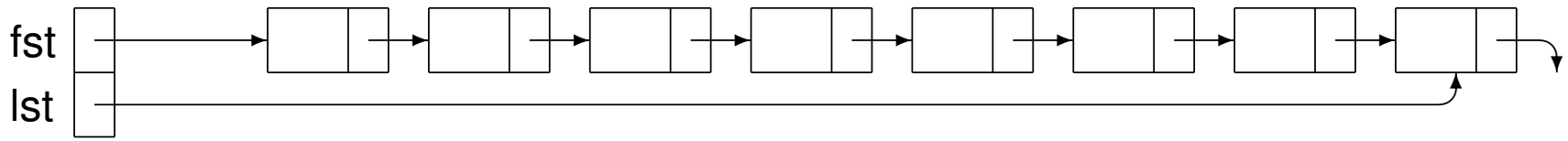
Cola vacía

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



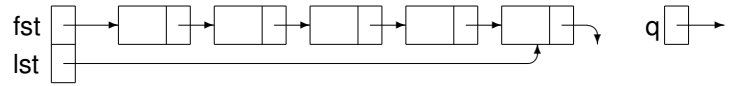
Primer elemento

{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)}

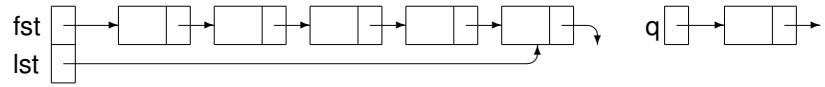


Encolar

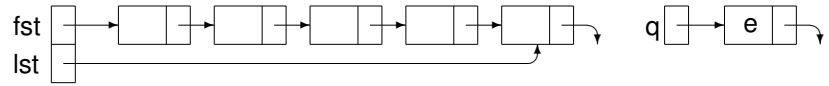
{Pre: p ∼ Q ∧ e ∼ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
var q: **pointer to** node



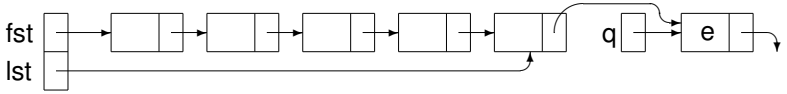
alloc(q)



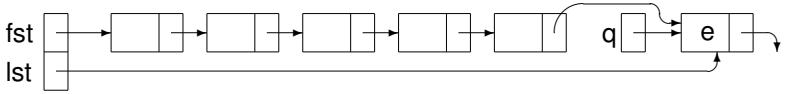
q→value:= e
q→next:= **null**



if p.lst = **null** → p.fst:= q {caso enqueue en cola vacía}
p.lst:= q
p.lst ≠ **null** → p.lst→next:= q



p.lst:= q



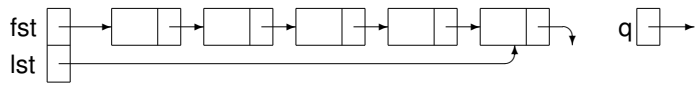
fi
end proc
{Post: p ∼ encolar(Q,E)}

Encolar
En limpio

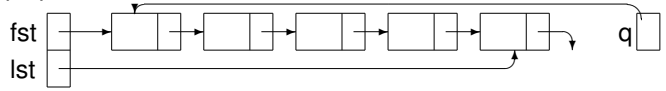
```
proc enqueue(in/out p:queue,in e:elem)
    var q: pointer to node
    alloc(q)
    q→value:= e
    q→next:= null
    if p.lst = null → p.fst:= q
                      p.lst:= q
      p.lst ≠ null → p.lst→next:= q
                      p.lst:= q
    fi
end proc
```

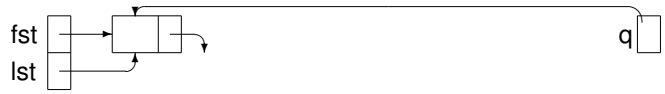
Decolar

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



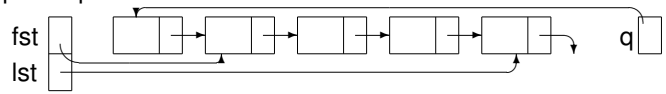
q:= p.fst





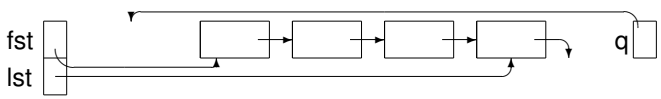
if p.fst = p.lst $\rightarrow$
p.fst:= **null** {caso cola con un solo elemento}
p.lst:= **null**

p.fst $\neq$ p.lst $\rightarrow$ p.fst:= p.fst->next



fi
free(q)

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



Decolar
En limpio

{Pre: p $\sim$ Q $\wedge\neg$is_empty(p)}
proc dequeue(**in**/**out** p:queue)
    var q: **pointer to** node
    q:= p.fst
    if p.fst = p.lst $\rightarrow$ p.fst:= **null** {caso cola con un solo elemento}
                p.lst:= **null**
      p.fst $\neq$ p.lst $\rightarrow$ p.fst:= p.fst->next
    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.fst = **null**)
end fun
{Post: b $\sim$ es_vacía(Q)}

Destroy

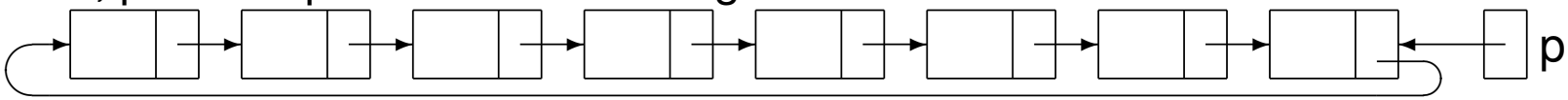
proc destroy(**in**/**out** p:queue)
    while ¬ is_empty(p) **do** dequeue(p) **od**
end proc

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

Implementación del TAD cola con listas enlazadas ciculares

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

Gráficamente, puede representarse de la siguiente manera



Explicación

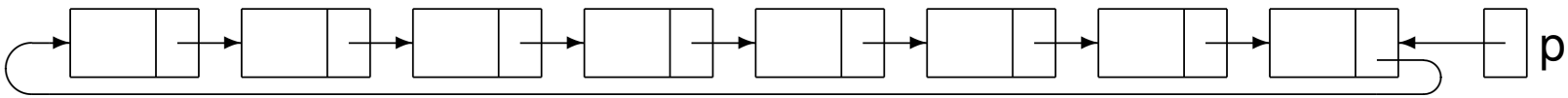
- La lista es circular,
- es decir que además de los punteros que ya teníamos en implementaciones anteriores,
- el último nodo tiene un puntero al primero,
- alcanza con saber dónde se encuentra el último nodo para saber también dónde está el primero.

Cola vacía

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

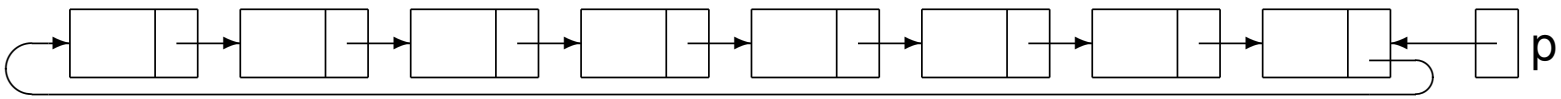
Primer elemento

{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)}



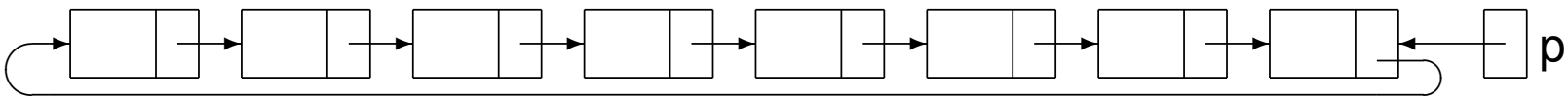
Encolar

{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** $\rightarrow$ p:= q {caso enqueue en cola vacía}
                q$\rightarrow$next:= q
      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}
                p:= q {que p también apunte al nuevo último}
    fi
end proc
{Post: p $\sim$ encolar(Q,E)}



Decolar

{Pre: p $\sim$ Q $\wedge \neg$is_empty(p)}
proc dequeue(**in**/**out** p:queue)
    var q: **pointer to** node
    q:= p.fst
    if p.fst = p.lst $\rightarrow$ p.fst:= **null** {caso cola con un solo elemento}
                p.lst:= **null**
      p.fst $\neq$ p.lst $\rightarrow$ p.fst:= p.fst->next
    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.

Implementación del TAD pcola con listas enlazadas circulares

- Se reusa la del TAD cola con listas enlazadas circulares,
- **type** queue = **pointer to** node
- Se asume un orden “mayor prioridad que” entre los elementos.
- Las operaciones vacía, encolar, es vacía y destroy se mantienen intactas.
- Las operaciones primero y decolar deben recorrer la lista para buscar o borrar el máximo.

```
{Pre: p ∼ Q ∧¬is_empty(p)}
fun first(p:pqueue) ret e:elem
    var r: pointer to node          {el puntero r para recorrer la lista enlazada}
    r:= p→next                                          {⋆r es el primer nodo}
    e:= r→value                                         {por ahora éste es el máximo}
    while r ≠ p do                                      {mientras ⋆r no sea el último nodo}
        r:= r→next                                      {que r pase a señalar el nodo siguiente}
        if e < r→value then e:= r→value fi              {que e sea el nuevo máximo}
    od
end fun
{Post: e ∼ primero(Q)}
```

```
{Pre: p ∼ Q ∧¬is_empty(p)}
proc dequeue(in/out p:pqueue)
    var r, pmax: pointer to node
    r:= p                                          {⋆(r→next) es el primer nodo}
    pmax:= p                          {por ahora pmax→next→value es el máximo}
    if p = p→next then p:= null                    {caso cola con un solo elemento}
    else while r→next ≠ p do          {mientras ⋆(r→next) no sea el último nodo}
            r:= r→next                   {que r pase a señalar el nodo siguiente}
            if pmax→next→value < r→next→value
            then pmax:= r              {pmax→next→value es el nuevo máximo}
            fi
        od                                  {ahora hay que saltear ⋆(pmax→next)}
        r:= pmax→next
        pmax→next := r→next
        p:= pmax      {sólo es útil cuando r = p, pero se puede hacer siempre}
    fi
    free(r)
end proc
{Post: p ∼ decolar(Q)}
```

Conclusiones

- Las dos operaciones son lineales,
- no es una buena implementación del TAD pcola
- ya que existen implementaciones mucho más eficientes,
- basadas en una estructura llamada heap.
- Lo veremos más adelante.