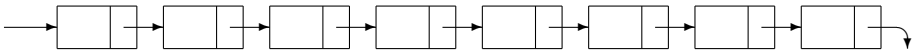
APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 2: ESTRUCTURAS DE DATOS

Listas Enlazadas

Por **listas enlazadas** se entiende una manera de implementar listas utilizando tuplas y punteros. Hay una variedad de listas enlazadas, la representación gráfica de la versión más simple de las mismas es la siguiente.



Es parecida a la de cola, con la diferencia de 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:

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

Es decir que una lista es en realidad 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. Es decir que 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. A continuación se implementa el tipo abstracto pila utilizando listas enlazadas.

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

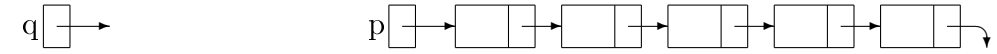
El procedimiento empty inicializa p como la pila vacía. Ésta se implementa con la lista enlazada vacía que consiste de la lista que no tiene ningún nodo, es decir, el puntero al primer nodo de la lista no tiene a quién apuntar. Su valor se establece en **null**.

```
proc empty(out p:stack)
    p:= null
end proc
```

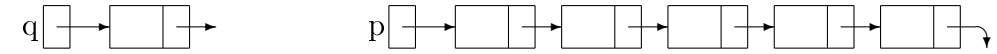
{Post: p $\sim$ vacía}

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).

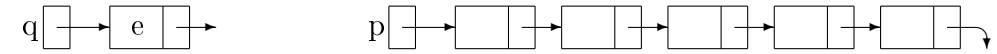
{Pre: p ∼ P ∧ e ∼ E}
proc push(**in** e:elem,**in**/**out** p:stack)
var q: **pointer to** node



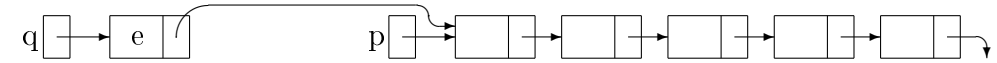
alloc(q)



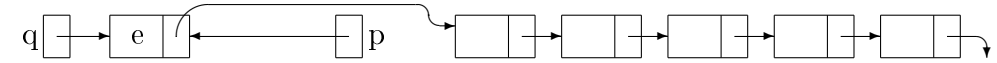
q→value:= e



q→next:= p



p:= q



end proc
{Post: p ∼ apilar(E,P)}

En limpio queda

{Pre: p ∼ P ∧ e ∼ 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 ∼ apilar(E,P)}

El valor de las representaciones gráficas que acompañan al código es muy relativo. Sólo sirven para entender lo que está ocurriendo de manera intuitiva. Hacer un tratamiento formal está fuera de los objetivos de este curso. De todas maneras, deben extremarse los cuidados para no incurrir en errores de programación que son muy habituales en el contexto de la programación con punteros.

En el ejemplo del procedimiento push además de la representación gráfica que parece indicar que el algoritmo es correcto, habría que asegurarse de que el algoritmo presentado funcione también cuando la pila p a la que se agrega e esté vacía, caso no contemplado en las representaciones gráficas que incluimos en el código. Dejamos al lector la tarea de comprobar que efectivamente se comporta de manera correcta cuando la pila está vacía, es decir, cuando el puntero p es **null**.

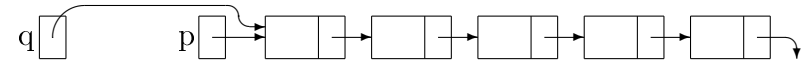
La función top no tiene más que devolver el elemento que se encuentra en el nodo apuntado por p.

{Pre: p ∼ P ∧¬is_empty(p)}
fun top(p:stack) **ret** e:elem
    e:= p→value
end fun
{Post: e ∼ primero(P)}

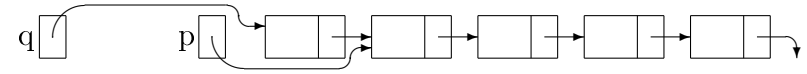
El procedimiento pop debe liberar el primer nodo de la lista y modificar p de modo que pase a apuntar al nodo siguiente. Observar que el valor que debe adoptar p se encuentra en el primer nodo. Por ello, antes de liberarlo es necesario utilizar la información que en él se encuentra. Por otro lado, si 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? La solución es recordar en una variable q el viejo valor de p (q:= p), hacer que p apunte al segundo nodo (p:= p→next) y liberar el primer nodo (free(q)). Observar que una vez liberado, p apunta al primer nodo de la nueva pila.

{Pre: p ∼ P ∧¬is_empty(p)}
proc pop(**in**/**out** p:stack)
    var q: **pointer to** node

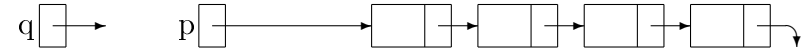
q:= p



p:= p→next



free(q)



end proc
{Post: p ∼ desapilar(P)}

En limpio queda

{Pre: p ∼ P ∧¬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 ∼ desapilar(P)}

Nuevamente, se deja al lector la tarea de asegurarse de que el procedimiento pop funciona bien cuando la pila p tiene un sólo elemento.

La función is_empty debe comprobar simplemente que la pila recibida corresponda a la lista enlazada vacía que se representa por el puntero **null**.

```
{Pre: p ∼ P}
fun is_empty(p:stack) ret b:Bool
    b:= (p = null)
end fun
{Post: b ∼ es_vacía(P)}
```

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
```

Por último, es fácil comprobar que todas las operaciones definidas resultan constantes con la sola excepción del procedimiento destroy que recorre toda la lista, por lo que su comportamiento es lineal en el número de elementos de la pila.

Implementación del TAD cola con listas enlazadas. Si uno quiere implementar el tipo cola usando listas enlazadas, puede usar la misma representación. La dificultad principal estará en la implementación de enqueue que deberá recorrer toda la lista para llegar al final de la misma, posición en donde debe agregarse el nuevo elemento. Asumiendo entonces que se ha definido

```
type queue = pointer to node
```

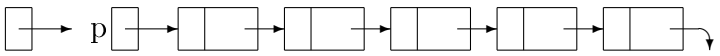
donde node se define igual que para las pilas, se puede definir empty, first, dequeue, is_empty, destroy en forma casi idéntica a como se definió empty, top, pop, is_empty, destroy para las pilas. Sólo cambia la implementación de enqueue, que sería como sigue.

```
{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)}
```

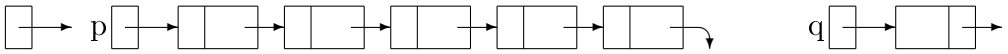
Como en casos anteriores, a continuación decoramos este algoritmo con gráficos que faciliten su comprensión. Observar que en la primera parte del algoritmo la cola puede

o no estar vacía, luego del condicional **if** se distinguen esos dos casos, por lo que en una rama la cola es vacía y en la otra no.

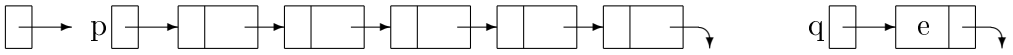
{Pre: p ∼ Q ∧ e ∼ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
var q,r: **pointer to** node



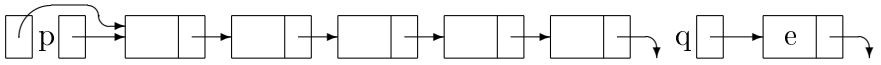
alloc(q)



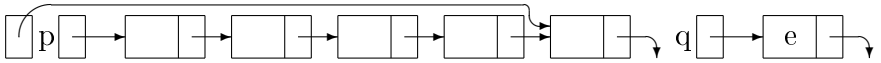
q→value:= e
q→next:= **null**



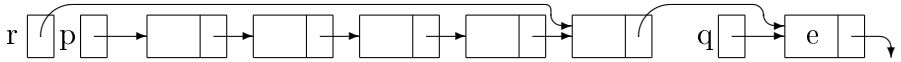
if p = **null** → p:= q {no engañarse con el dibujo, la cola puede ser vacía}
p ≠ **null** → r:= p



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

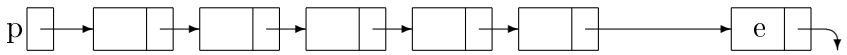


r→next:= q



fi
end proc
{Post: p ∼ encolar(Q,E)}

Al finalizar la ejecución de enqueue, las variables locales q y r desaparecen. Queda

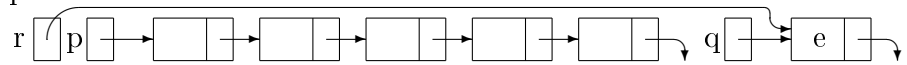


que es la cola con el nuevo elemento agregado al final. El invariante del **while** podría enunciarse diciendo "el último nodo de la cola se encuentra siguiendo las flechas a partir de r".

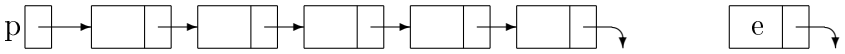
Este algoritmo revela que no es sencillo trabajar con punteros. Exige extremar las precauciones para mantener la integridad de la estructura. Por ejemplo, si bien en la quinta línea la asignación de **null** a q→next puede parecer poco importante, eso es lo que garantiza que una vez terminado el procedimiento, el puntero del último nodo de la cola sea **null**. Por otro lado, la condición del **while**, r→next ≠ **null** requiere que r no sea **null**. Afortunadamente esto es cierto porque inicialmente r es p que a su vez es distinto de **null**, y en el cuerpo del **while** siempre se asigna a r un valor distinto de **null**. Es decir, el invariante implica que r no es **null**. La condición del **while** también requiere que r señale una dirección de memoria que ha sido reservada. Afortunadamente, esto también es consecuencia del invariante porque toda la cola p se asume compuesta de espacios de memoria que han sido oportunamente reservados. Por supuesto que para

que esto a su vez sea cierto, uno debe asegurarse de que la implementación de todas las operaciones del TAD cola preserven la integridad de la estructura.

Por último, puede parecer correcto simplificar la condición del **while**, reemplazandolo por **while** r $\neq$ **null do** r:= r->next **od**. Si bien esto también recorre toda la cola, cuando el **while** termina se tiene r = **null**, es decir que r no señala al último nodo de la cola, que es donde debe insertarse q. La asignación r:= q modificaría r, pero no modificaría la última flecha de la cola p para que señale al nuevo nodo $\star$q. Gráficamente, obtendríamos simplemente



que evidentemente no inserta el nuevo elemento al final. Además, al final del procedimiento, al desaparecer las variables locales q y r, el nuevo nodo no sería accesible jamás. Quedaría



donde lo único accesible es lo que se puede alcanzar siguiendo las flechas partiendo de una variable. En este caso, la única variable es p por lo que el nodo de la derecha no se libera pero se pierde.

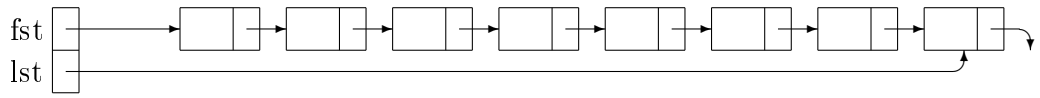
De la misma manera, si usáramos p mismo para recorrer el arreglo en vez de declarar una variable auxiliar r, perderíamos la información que originariamente tenía p, y por lo tanto, perderíamos acceso al comienzo de la lista y, con ello, a toda la lista.

Esta implementación del TAD cola no sólo involucra una recorrida de la cola, que es lo más complicado que hemos visto hasta ahora con punteros, si no que a causa de esa misma recorrida la operación enqueue obtenida es lineal en la longitud de la cola, es decir, es poco eficiente.

Una implementación más eficiente puede obtenerse manteniendo dos punteros en vez de sólo uno; uno al primer nodo de la cola y otro al último:

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

Gráficamente, puede representarse de la siguiente manera



Las operaciones del tipo cola se implementan a continuación:

```
proc empty(out p:queue)
    p.fst:= null
    p.lst:= null
end proc
{Post: p ~ vacia}
```

Es decir, la cola vacía estará representada por los dos punteros igualados a **null**. Sólo en un caso más ambos punteros pueden coincidir. En efecto, cuando la cola tenga

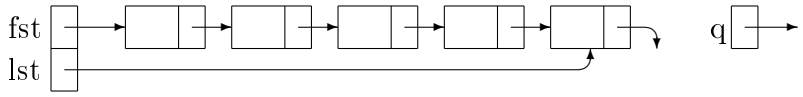
exactamente un elemento éste será a la vez el primero y último. En ese caso tanto el campo fst como el campo lst de p tendrán la dirección de ese único nodo de la cola.

El primer elemento se encuentra en el nodo señalado por el campo fst de p.

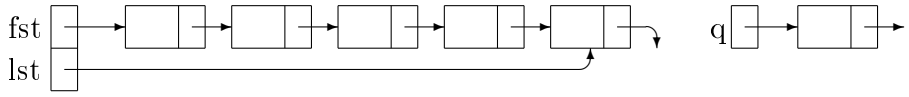
{Pre: p ∼ Q ∧¬is_empty(q)}
fun first(p:queue) **ret** e:elem
    e:= p.fst→value
end fun
{Post: e ∼ primero(Q)}

Para el procedimiento enqueue necesitamos, como en varios de los procedimientos vistos, un puntero auxiliar q para crear el nuevo nodo.

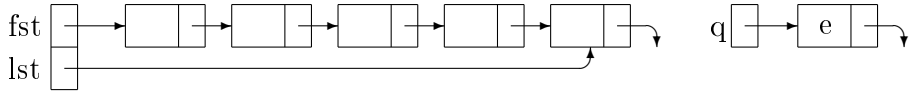
{Pre: p ∼ Q ∧ e ∼ E}
proc enqueue(**in**/**out** p:queue,**in** e:elem)
    var q: **pointer to** node



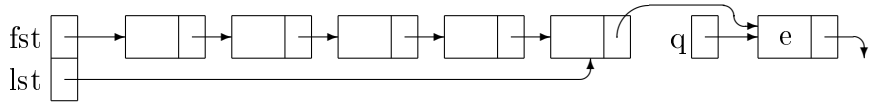
    alloc(q)



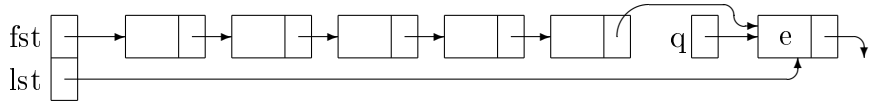
    q→value:= e
    q→next:= **null**



    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)}

Pasamos en limpio el procedimiento enqueue eliminando los gráficos que hemos usado para describir los cambios de estado. Nos queda:

```
{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
                        p.lst:= q
        p.lst ≠ null → p.lst→next:= q
                        p.lst:= q
      fi
end proc
{Post: p ∼ encolar(Q,E)}
```

En el procedimiento dequeue se utiliza q para conservar temporariamente la dirección del primer nodo y así poder liberar ese nodo al final del procedimiento, cuando la información que contiene deja de ser necesaria.

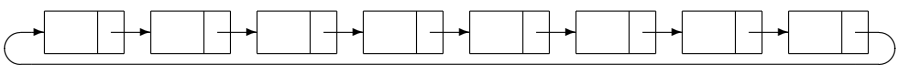
```
{Pre: p ∼ Q ∧¬is_empty(p)}
proc dequeue(in/out p:queue)
      var q: pointer to node
      q:= p.fst
      if p.fst = p.lst → p.fst:= null                {caso cola con un solo elemento}
                         p.lst:= null
        p.fst ≠ p.lst → p.fst:= p.fst->next
      fi
      free(q)
end proc
{Post: p ∼ decolar(Q)}

{Pre: p ∼ Q}
fun is_empty(p:queue) ret b:Bool
    b:= (p.fst = null)
end fun
{Post: b ∼ es_vacía(Q)}

{libera todo el espacio de memoria ocupado por p}
proc destroy(in/out p:queue)
      while ¬ is_empty(p) do dequeue(p) od
end proc
```

En esta implementación todas las operaciones resultaron constantes excepto destroy que como debe recorrer toda la cola es lineal en la cantidad de elementos de la cola.

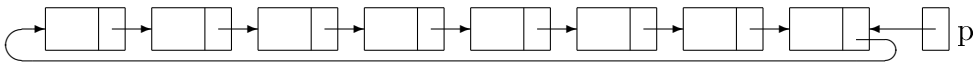
Una implementación alternativa igualmente eficiente es posible usando listas circulares. Éstas son listas enlazadas en las que el último nodo no tiene un puntero con valor nulo **null** sino un puntero al primero de la lista. Gráficamente,



Con esta representación, evidentemente no hace falta mantener un puntero al primero de la cola, alcanza con un puntero al último, aquél puede conseguirse a partir de éste en tiempo constante.

type queue = **pointer to** node

Gráficamente, una cola p puede representarse de la siguiente manera



Las operaciones del TAD cola se implementan a continuación. La cola vacía se representa como es habitual con **null**.

```
proc empty(out p:queue)
    p:= null
end proc
{Post: p ∼ vacia}
```

Para el procedimiento enqueue volvemos a utilizar un puntero auxiliar, sólo que esta vez luego de utilizarlo para reservar espacio para el nuevo (último) nodo no asignamos **null** al puntero de dicho nodo. Esto se debe a que el último nodo de la cola, en esta representación, debe apuntar al primero en vez de tener el valor **null** como en las versiones anteriores. La novedad radica en "insertar" el nodo en la lista sin romper la circularidad.

```
{Pre: p ∼ Q ∧ e ∼ E}
proc enqueue(in/out p:queue,in e:elem)
    var q: pointer to node
    alloc(q)
    q→value:= e
    if p = null → p:= q                              {caso enqueue en cola vacía}
                  q→next:= q
       p ≠ null → q→next:= p→next          {que el nuevo último apunte al primero}
                  p→next:= q          {que el viejo último apunte al nuevo último}
                  p:= q                    {que p también apunte al nuevo último}
    fi
end proc
{Post: p ∼ encolar(Q,E)}
```

En esta representación, para devolver el primero de la cola es necesario acceder al campo value, no de p que es el último de la cola, sino de p→next que es el primero.

```
{Pre: p ∼ Q ∧¬is_empty(q)}
fun first(p:queue) ret e:elem
   e:= p→next→value
end fun
{Post: e ∼ primero(Q)}
```

Para eliminar el primero de la cola usamos un puntero auxiliar q que señala el nodo que se ha de eliminar, se realiza la eliminación preservando la circularidad de la cola. Es decir, en caso de tener un sólo elemento la cola pasa a ser la cola vacía, y en caso de

tener más de un elemento, el puntero del último nodo debe ahora señalar el nodo que hasta entonces era el segundo de la cola.

```
{Pre: p ∼ Q ∧¬is_empty(p)}
proc dequeue(in/out p:queue)
    var q: pointer to node
    q:= p→next                                        {q apunta al primero}
    if p = q → p:= null                               {caso cola con un solo elemento}
      p ≠ q → p→next:= q→next   {que el último apunte al que antes era segundo}
    fi
    free(q)
end proc
{Post: p ∼ decolar(Q)}
```

Las dos operaciones restantes no presentan novedades.

```
{Pre: p ∼ Q}
fun is_empty(p:queue) ret b:Bool
   b:= (p = null)
end fun
{Post: b ∼ es_vacía(Q)}
```

El procedimiento destroy libera todo el espacio de memoria ocupado por p.

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

Implementación del TAD cola de prioridades con listas enlazadas. Hemos logrado implementaciones eficientes de los TADs contador, pila y cola. En efecto, si bien algunas implementaciones daban lugar a operaciones lineales, pronto encontramos variantes que permitieron implementar el TAD con operaciones constantes.

No es el caso de la cola de prioridades. Las implementaciones más eficientes que existen requieren operaciones logarítmicas. Las veremos pronto, cuando presentemos árboles binarios.

En esta sección veremos qué se obtiene si adaptamos las implementaciones de colas para implementar colas de prioridades.

Por ejemplo, si tomamos la implementación de cola que usa listas enlazadas circulares, el problema aparece al definir la función first, que debe devolver el mayor elemento de la cola de prioridades y para ello, debe recorrerla toda, y esto la vuelve lineal. La operación dequeue, que debe eliminar dicho elemento, también es lineal. Otra posibilidad sería mantener la cola ordenada. En ese caso la función first y el procedimiento dequeue serían constantes, pero el procedimiento enqueue sería lineal.

Consideraciones similares pueden hacerse con la implementación de cola en un arreglo, con el agravante de que parece necesario desplazar los elementos del arreglo al insertar o borrar un elemento.

Tomamos como ejemplo la primer posibilidad: usamos listas enlazadas circulares y los procedimientos empty y enqueue y la función is_empty se mantienen idénticas que para cola (cambiando en todos lados queue por pqueue). Es decir que la lista enlazada

circular no se mantiene ordenada. La función first debería entonces recorrer toda la lista enlazada:

```
{Pre: p ∼ Q ∧¬is_empty(q)}
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)}
```

El procedimiento dequeue es más complicado. Requiere eliminar el nodo donde se encuentra el máximo. Para ello, debemos recordar la dirección del nodo anterior a ese para poder modificar su campo next de modo de saltearlo. Utilizamos la misma variable r para recorrer, pero esta vez r va a alojar todo el tiempo la dirección del nodo anterior al que se está observando. Además es necesario recordar el nodo anterior a donde se encuentra el máximo (para ello usamos el puntero pmax). También es necesario tratar separadamente el caso de la cola con un solo elemento.

```
{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)}
```

Como anticipamos estas operaciones resultan lineales, pero no es fácil encontrar una implementación mejor. Mantener la lista enlazada ordenada volvería constantes estas operaciones, pero lineal el procedimiento enqueue. Recién obtendremos una implementación mejor utilizando heaps, cuando veamos árboles binarios.

De todas formas, las implementaciones anteriores son ejemplos interesantes que ilustran las dificultades del uso de punteros para recorrer estructuras enlazadas. En este caso, r y pmax se utilizan para señalar nodos anteriores a los que pareciera lógico. Esto es por la necesidad de eliminar un nodo, hace falta contar con la dirección del nodo anterior para modificar su campo next. Esto complica la lógica de todo el procedimiento.

La asignación p:= pmax sólo se necesita en caso de que el máximo sea el último, es decir, el valor del nodo $\star$p. En ese caso el dequeue va a borrar ese nodo y por lo tanto el puntero p, es decir la cola, debe apuntar a otro nodo. La asignación p:= pmax elige que apunte al anterior al que se está borrando. Tratándose de listas circulares en las que el orden no importa, la asignación puede hacerse siempre.

Se puede observar que la condición del **while** es verdadera la primera vez que se evalúa, ya que es consecuencia de que no se cumpla la condición del **if** externo. Por ello, puede reemplazarse **while** r$\rightarrow$next $\neq$ p **do** ... **od** por **repeat** ... **until** r$\rightarrow$next = p.