ESTRUCTURAS DE DATOS: LISTAS(2)

Profesora: Mª José Domínguez Alda

MÁS USOS DE LAS LISTAS

Con `LISTA[ELEMENTO]` y `LISTA2[ELEMENTO]` creamos listas poniendo datos en distintos puntos de la estructura pero lo que realmente diferencia a las listas de las colas y pilas es que no tienen un punto de acceso para las consultas.

Esto quiere decir que las listas pueden **recorrerse**, mientras que las pilas y colas para poder tratarlas había que ir eliminando los elementos que las componían.

En la siguiente especificación se proporcionan operaciones para insertar, modificar o borrar datos en el interior de la lista, o para buscar datos a lo largo de la lista.

ESPECIFICACIÓN:LISTAS AMPLIADAS

espec *LISTA+[ELEMENTO]*

usa *LISTA[ELEMENTO], NATURALES2*

operaciones

{Ver el dato dada una posición, insertar, modificar o borrar}

parcial _ [_]*: lista natural → elemento*

parcial *insertar: elemento lista natural → lista*

parcial *modificar: elemento lista natural → lista*

parcial *borrar: lista natural → lista*

{Ver si un dato está en la lista, y en qué posición se encuentra}

***está?**: elemento lista → bool*

***buscar**: elemento lista → natural*

ESPECIFICACIÓN:LISTAS AMPLIADAS (2)

var

n *: natural*

x,y *: elemento*

l *: lista*

ecuaciones de definitud *{escritas informales por claridad}*

$$(1 \leq i \leq long(l)) = T \Rightarrow Def(l[i])$$

$$(1 \leq i \leq long(l)) = T \Rightarrow Def(modificar(x,l,i))$$

$$(1 \leq i \leq long(l)) = T \Rightarrow Def(borrar(l,i))$$

$$(1 \leq i \leq long(l)+1) = T \Rightarrow Def(insertar(x,l,i))$$

ESPECIFICACIÓN:LISTAS AMPLIADAS (3)

ecuaciones

{Las listas empiezan en la posición 1}

$$(x:l)[suc(0)] = x$$

$$n > 0 \Rightarrow (x:l)[suc(n)] = l[n]$$

{Insertar **x** *en la posición 1 es poner* **x** *el primero de todos}*

$$insertar(x,l,suc(0)) = x:l$$

{Insertar **x** *después de la posición 1 es dejar el primero de la lista exactamente igual y seguir buscando el sitio de* **x***}*

$$n > 0 \Rightarrow$$

$$insertar(x,y:l,suc(n)) = y:insertar(x,l,n)$$

ESPECIFICACIÓN:LISTAS AMPLIADAS (4)

{Borrar la posición 1 es quitar la cabeza de la lista}

`borrar(x:l,suc(0)) = l`

{Borrar después de la posición 1 es dejar la cabeza en su sitio y seguir buscando el que hay que borrar }

`n > 0 ⇒ borrar(x:l,suc(n)) = x:borrar(l,n)`

{Modificar el dato de la posición **`n`** *es como si se quitase y luego se insertarse el dato nuevo}*

`n > 0 ⇒ modificar(x,l,n) =`
`insertar(x,borrar(l,n),n)`

ESPECIFICACIÓN:LISTAS AMPLIADAS (5)

{Para ver si un dato está en la lista se recorre y se compara con cada uno de los elementos de la lista, hasta que no queden más}

esta?(x,[]) = F

esta?(x,y:l) = (x eq y) ∨ esta?(x,l)

{Buscar la posición que ocupa un dato es mirar si está o no en la lista, y si está se va contando de uno en uno hasta llegar al dato}

esta?(x,l) = F ⇒ buscar(x,l) = 0

esta?(x,y:l) = T ∧ (x eq y) = T ⇒
buscar(x,y:l) = suc(0)

esta?(x,y:l) = T ∧ (x eq y) = F ⇒
buscar(x,y:l) = suc(buscar(x,l))

fespec

LISTAS AMPLIADAS. OBSERVADORAS. PSEUDOCÓDIGO

{Obtener el elemento en una posición _ [_] }

```
func coger (l:lista, n:natural):elemento
{devuelve el elto que está en la posición n)
   si (n=0)
   entonces error(El elemento 0 no existe)
   sino  si (long(l)<n)
      entonces error (No hay n elementos)
      sino  si n=1 entonces devolver prim(l)
         sino  resto(l)
               devolver coger (l, pred(n))
   finsi
finfunc
```

LISTAS AMPLIADAS. OBSERVADORAS. PSEUDOCÓDIGO (2)

*{ Ver si un elemento está en la lista **esta?**}*

```
func esta?( e:elemento, l: lista):booleano
var primero:elemento
   si vacia?(l)
      entonces devolver F
   sino primero←prim(l)
        resto(l)
        devolver (primero eq e) V esta?(e, l)
   finsi
finfunc
```

LISTAS AMPLIADAS. OBSERVADORAS. PSEUDOCÓDIGO (3)

*{Buscar la posición de un elemento en la lista **buscar**}*

```
func buscar (e:elemento, l: lista):natural
var primero:elemento
   si vacia?(l)
      entonces devolver 0
   sino    primero←prim(l)
           si (primero eq e)
              entonces devolver 1
           sino resto(l)
              devolver suc(buscar(e, l))
   finsi
finfunc
```

LISTAS AMPLIADAS. MODIFICADORAS. PSEUDOCÓDIGO (4)

*{Insertar un elemento en una posición **insertar**}*

```
proc insertar(e:elemento, E/S l:lista, n:natural)
   var primero:elemento
      si (n=0)
      entonces error(El elemento 0 no existe)
      sino si (n=1) entonces e:l
         sino primero←prim(l)
              resto(l)
              insertar(e,l,pred(n))
              primero:l
      finsi
   finproc
```

LISTAS AMPLIADAS. MODIFICADORAS. PSEUDOCÓDIGO (5)

*{Borrar el elemento de una posición **borrar**}*

```
proc borrar (E/S l:lista, n:natural)
   si (n=0)
   entonces error(El elemento 0 no tiene
   sentido)
   sino si (long(l)<n)
      entonces error (No hay n elementos)
      sino si n=1 entonces resto(l)
            sino borrar (resto(l), pred(l))
                  prim(l):l
   finsi
finproc
```

LISTAS AMPLIADAS. MODIFICADORAS. PSEUDOCÓDIGO (6)

*{Cambiar el elemento en una posición **modificar**}*

```
proc modificar (e:elemento, E/S l:lista, n:natural)
var primero:elemento
   si (n=0)
   entonces error(El elemento 0 no existe)
   sino  si (long(l)<n)
            entonces error (No hay n elementos)
         sino si n=1 entonces resto(l) e:l
                 sino  primero←prim(l) resto(l)
                       modificar(e, l, pred(n))
                       primero:l
   finsi
finproc
```

LISTAS ENLAZADAS. TIPOS

```
tipos
    nodo-lista = reg
    valor: elemento
      sig: puntero a nodo-lista        {esto es lo mínimo}
freg
    lista = reg
    longitud: nat                      {no siempre es necesaria}
    primero: puntero a nodo-lista      {cabecera de lista}
    freg
ftipos
```

LISTAS AMPLIADAS. OBSERVADORAS

{Obtener el elemento en una posición _ [_] }

```
fun coger(l: lista, E n: nat) dev e: elemento
```

LISTAS AMPLIADAS. OBSERVADORAS

{Obtener el elemento en una posición _ [_] }

```
fun coger(l: lista, E n: nat) dev e: elemento
var p: puntero a nodo-lista; pos: nat
   si (n = 0) o (n > l.longitud) entonces
         error(posición inexistente)
   si no    p ← l.primero
            pos ← 1
            mientras (pos < n)hacer
                  pos ← pos+1 p ← p^.sig
            fmientras
            e ← p^.valor
   fsi
   ffun
```

LISTAS AMPLIADAS. OBSERVADORAS (2)

*{Buscar la posición de un elemento en la lista **buscar**}*

```
fun buscar(E e: elemento, l: lista) dev n: nat
```

LISTAS AMPLIADAS. OBSERVADORAS (2)

*{Buscar la posición de un elemento en la lista **buscar**}*

```
fun buscar(E e: elemento, l: lista) dev n: nat
var    p: puntero a nodo-lista
       pos: nat
       si es_lista_vacia(l) entonces
          error(Lista vacía)
       si no  p ← l.primero
                 pos ← 1
       mientras (p ≠ nil) y (p^.valor ≠ e) hacer
                 pos ← pos+1; p ← p^.sig
       fmientras
       si (p ≠ nil) entonces
                 n ← pos
          si no n ← 0
       fsi
   fsi
ffun
```

LISTAS AMPLIADAS. OBSERVADORAS (3)

*{ Ver si un elemento está en la lista **esta?**}*

```
fun esta(E e: elemento, l: lista) dev b: bool
```

LISTAS AMPLIADAS. OBSERVADORAS (3)

```
                { Ver si un elemento está en la lista esta?}
fun esta(E e: elemento, l: lista) dev b: bool
var   p: puntero a nodo-lista
      pos: nat
      si es_lista_vacia(l) entonces b ← Falso
      si no p ← l.primero
      mientras (p ≠ nil) y (p^.valor ≠ e) hacer
         p ← p^.sig
      fmientras
      b ← p ≠ nil
   fsi
ffun
```

LISTAS AMPLIADAS. MODIFICADORAS

*{Cambiar el elemento en una posición **modificar**}*

```
proc modificar(l: lista, E n: nat, E e: elemento)
```

LISTAS AMPLIADAS. MODIFICADORAS

*{Cambiar el elemento en una posición **modificar**}*

```
proc modificar(l: lista, E n: nat, E e: elemento)
var   p: puntero a nodo-lista
      pos: nat
      si (n = 0) o (n > l.longitud)
         entonces error(posición inexistente)
      si no p ← l.primero
      pos ← 1
      mientras (pos < n)hacer
         pos ← pos+1  p ← p^.sig
      fmientras
      p^.valor ← e
   fsi
fproc
```

LISTAS AMPLIADAS. MODIFICADORAS (2)

*{Insertar un elemento en una posición **insertar**}*

```
proc insertar(l: lista, E n: nat, E e: elemento)
```

LISTAS AMPLIADAS. MODIFICADORAS (2)

*{Insertar un elemento en una posición **insertar**}*

```
proc insertar(l: lista, E n: nat, E e: elemento)
var  p, aux: puntero a nodo-lista
     pos: nat
si (n = 0) o (n > l.longitud+1)
   entonces error(posición inexistente)
   si no
      reservar(p)
      p^.valor ← e
      si (n=1) entonces {se pone el primero}
         p^.sig ← l.primero
         l.primero ← p
      si no
         (...)
```

LISTAS AMPLIADAS. MODIFICADORAS (3)

*{Insertar un elemento en una posición **insertar**}*

```
        (...) {busco su lugar}
        aux ← l.primero
        pos ← 1
        mientras (pos+1 < n) hacer
           pos ← pos+1 aux ← aux^.sig
        fmientras
        p^.sig ← aux^.sig
        aux^.sig ← p
      fsi {se ha insertado}
      l.longitud ← l.longitud+1
   fsi {hubo un error}
fproc
```

LISTAS AMPLIADAS. MODIFICADORAS (4)

*{Borrar el elemento de una posición **borrar**}*

```
proc borrar(l: lista, E n: nat)
```

LISTAS AMPLIADAS. MODIFICADORAS (4)

*{Borrar el elemento de una posición **borrar**}*

```
proc borrar(l: lista, E n: nat)
var  p, aux: puntero a nodo-lista
     pos: nat
     si (n = 0) o (n > l.longitud)
     entonces error(posición inexistente)
     si no p ← l.primero
        si (n=1) entonces
                          {se quita el primero}
        l.primero ← p^.sig
        p^.sig ← nil     {por seguridad}
        liberar(p)
     si no
        (...)
```

LISTAS AMPLIADAS. MODIFICADORAS (5)

*{Borrar el elemento de una posición **borrar**}*

```
        (...) pos
        ← 1
        mientras (pos+1 < n) hacer pos ←
           pos+1
           p ← p^.sig
        fmientras
        aux ← p^.sig
        p^.sig ← aux^.sig
        aux^.sig ← nil {por seguridad}
        liberar(aux)
      fsi {se ha borrado}
      l.longitud ← l.longitud-1
   fsi {hubo un error}
fproc
```

COMENTARIOS

- Algunas operaciones se consigue que sean muy rápidas: **`[]`**, **`[x]`**, **`vacia?(l)`**, **`prim(l)`**, **`resto(l)`**, **`long(l)`** y **`x:l`** se ejecutan en tiempo constante O(1).
- Sin embargo, hay otras operaciones que son lentas: **`l++l'`**, **`x#l`**, **`ult(l)`**, **`eult(l)`**, **`l[i]`**, **`buscar(x,l)`**, **`esta?(x,l)`**, **`insertar(l,i,x)`**, **`modificar(l,i,x)`**, y **`borrar(x,i)`** se ejecutan en tiempo O(n) ya que implican recorrer linealmente la estructura.

IMPLEMENTACIÓN: LISTAS (2)

Podemos enriquecer la implementación añadiendo un puntero al último elemento:

- Si se añade al tipo lista un puntero al último elemento y un contador las operaciones **`l++l'`**, **`x#l`**, **`long(l)`** y **`ult(l)`** pasan a tiempo constante O(1).
- En cada celda puede añadirse un puntero al anterior para tener listas doblemente enlazadas:
    - La complejidad en tiempo no varía (esencialmente).
    - Aumenta el empleo de memoria.

LISTAS DOBLEMENTE ENLAZADAS. TIPOS

```
tipos
   nodo-listad = reg
      valor: elemento
      ant, sig: puntero a nodo-lista
                freg
listad = reg
      primero, ultimo: puntero a nodo-listad
      longitud: nat
ftipos
```

LISTAS DOBLEMENTE ENLAZADAS. CONSTRUCTORAS

*{Crear una lista vacía **[ ]**}*

```
func lista_vacia():listad
```

LISTAS DOBLEMENTE ENLAZADAS. CONSTRUCTORAS

*{Crear una lista vacía **[ ]**}*

```
func lista_vacia():listad
var l:lista
   l.primero←nil
   l.ultimo←nil
   l.longitud←0
   devolver(l)
finfunc
```

LISTAS DOBLEMENTE ENLAZADAS. CONSTRUCTORAS (2)

*{Crear una lista de un elemento **[_]** }*

```
func unitaria (e:elemento):listad
```

LISTAS DOBLEMENTE ENLAZADAS. CONSTRUCTORAS (2)

*{Crear una lista de un elemento **[_]** }*

```
func unitaria (e:elemento):listad
var p:listad
   reservar(p.primero)
   p.primero^.valor←e
   p.primero^.sig←nil
   p.primero^.ant←nil
   p.ultimo←p.primero
   p.longitud←1
   devolver p
finfunc
```

LISTAS DOBLEMENTE ENLAZADAS. CONSTRUCTORAS (3)

{Añadir un elemento a una lista por la izquierda _ : _ }

```
proc añadir_izq (e:element, l:lista)
```

LISTAS DOBLEMENTE ENLAZADAS. CONSTRUCTORAS (3)

{Añadir un elemento a una lista por la izquierda _ : _ }

```
proc añadir_izq (e:elemento, l:listad)
var p p:puntero a nodo_listad
   reservar(p)
   p^.valor←e
   p^.sig←l.primero
   p^.ant←nil

   l.primero^.ant←p
   l.primero ←p
   l.longitud← l.longitud+1
finproc
```

LISTAS DOBLEMENTE ENLAZADAS. CONSTRUCTORAS (4)

{Añadir un elemento a una lista por la derecha _ # _ }

```
proc añadir_der (e:element, l:listad)
```

LISTAS DOBLEMENTE ENLAZADAS. CONSTRUCTORAS (4)

{Añadir un elemento a una lista por la derecha _ # _ }

```
proc añadir_der (e:element, l:listad)
var p, aux:puntero a nodo_listad
   reservar(p)
   p^.valor←e
   p^.sig←nil
   p^.ant←l.ultimo

   l.ultimo^.sigue←p
   l.ultimo←p
   l.longitud←l.longitud+1
finproc
```

LISTAS DOBLEMENTE ENLAZADAS. OBSERVADORAS

*{Obtener el elemento inicial **prim**}*

```
func inicial(l:listad):elemento
```

LISTAS DOBLEMENTE ENLAZADAS. OBSERVADORAS

*{Obtener el elemento inicial **prim**}*

```
func inicial(l:listad):elemento
   si vacia?(l) entonces error (lista vacía)
   sino devolver l.primero^.valor
   finsi
finfunc
```

LISTAS DOBLEMENTE ENLAZADAS. OBSERVADORAS(2)

*{Obtener el elemento extremo **ult**}*

```
func final(l:listad):elemento
```

LISTAS DOBLEMENTE ENLAZADAS. OBSERVADORAS(2)

*{Obtener el elemento extremo **ult**}*

```
func final(l:listad):elemento
   si vacia?(l) entonces error (lista vacía)
   sino devolver l.ultimo^.valor
   finsi
finfunc
```

LISTAS DOBLEMENTE ENLAZADAS. OBSERVADORAS(3)

{Ver si la lista es vacía}

```
func vacia?(l:listad):booleano
```

LISTAS DOBLEMENTE ENLAZADAS. OBSERVADORAS(3)

{Ver si la lista es vacía}

```
func vacia?(l:listad):booleano
    devolver l.longitud=0
finfunc
```

LISTAS DOBLEMENTE ENLAZADAS. MODIFICADORAS

*{Eliminar el primer elemento de una lista **resto**}*

```
proc elim_inicial(l:listad):listad
```

LISTAS DOBLEMENTE ENLAZADAS. MODIFICADORAS

*{Eliminar el primer elemento de una lista **resto**}*

```
proc elim_inicial(l:listad):listad
var ini:puntero a nodo_listad
   si vacia?(l) entonces error (lista vacía)
   sino  ini←l.primero
      l.primero←l.primero^.sig
      l.primero^.ant←nil
      ini^.sig←nil                    {por seguridad}
      liberar(ini)
      l.longitud← l.longitud-1
finproc
```

LISTAS DOBLEMENTE ENLAZADAS. MODIFICADORAS(2)

*{Eliminar el último elemento de una lista **eult**}*

```
proc elim_final(l:listad)
```

LISTAS DOBLEMENTE ENLAZADAS. MODIFICADORAS(2)

*{Eliminar el último elemento de una lista **eult**}*

```
proc elim_final(l:listad)
var ult:puntero a nodo_listad
    si vacia?(l) entonces error (lista vacía)
    sino ult←l.ultimo
         l.ultimo^.ant^.sig←nil
         l.ultimo←l.ultimo^.ant
         ult^.ant←nil                          {por seguridad}
         liberar(ult)
         l.longitud← l.longitud-1
    finsi
finproc
```