

# **ESTRUCTURAS DE DATOS**

## **LISTAS**



# LISTAS

Una lista  $L$  es una estructura lineal caracterizada porque no tiene puntos de acceso obligatorios (sin embargo, éstos suelen ser fijos y dependientes de la construcción).

Una lista:

- o bien es vacía, en cuyo caso se denomina *lista vacía*,
- o bien puede distinguirse un elemento  $x$ , llamado *cabeza*, y el resto de elementos forman una lista secundaria  $L'$ , que se denomina *resto* de la lista inicial.

Emplearemos la notación  $L = x:L'$ .



# **ESPECIFICACIÓN: LISTAS**

*{Esta especificación es para la creación básica de listas}*

**espec** *LISTA[ELEMENTO]*

**usa** *BOOLEANOS*

**parametro formal**

**generos** *elemento*

**operaciones**

*{igualdad entre elementos}*

*\_ eq \_: elemento elemento → bool*

**fparametro**

**generos** *lista*



# ESPECIFICACIÓN: LISTAS (2)

## operaciones

{Generadoras}

[ ]:  $\rightarrow$  lista {lista vacía}

\_:\_ : elemento lista  $\rightarrow$  lista {añadir por la izquierda}

{Modificadoras}

parcial resto : lista  $\rightarrow$  lista {eliminar primero }

parcial eult : lista  $\rightarrow$  lista {eliminar último}

{Observadoras}

parcial prim : lista  $\rightarrow$  elemento {primero de la lista}

parcial ult : lista  $\rightarrow$  elemento {último de la lista}

vacía? : lista  $\rightarrow$  bool {ver si tiene datos}



## ESPECIFICACIÓN: LISTAS (3)

**var**

***x** : elemento*

***l** : lista*

**ecuaciones de definitud**

***Def** (prim (x:l) )*

***Def** (ult (x:l) )*

***Def** (resto (x:l) )*

***Def** (eult (x:l) )*



## ESPECIFICACIÓN: LISTAS (4)

ecuaciones

$resto( x:l ) = l$

$vacía?(l)=T \Rightarrow eult( x:l ) = []$

$vacía?(l)=F \Rightarrow eult( x:l ) = x:eult( l )$

$prim( x:l ) = x$

$vacía?(l)=T \Rightarrow ult( x:l ) = x$

$vacía?(l)=F \Rightarrow ult( x:l ) = ult( l )$

$vacía?( [] ) = T$

$vacía?( x:l ) = F$

fespec



## EJEMPLO 1

Ejemplo: Suponiendo la operación que devuelve el elemento más pequeño de dos *min: elemento elemento → elemento*, obtener el elemento más pequeño de una lista.

- La operación debe ser parcial, se requiere tener datos:

*parcial mínimo: lista → elemento*

- Las ecuaciones (incluyendo la de definición) pueden ser:

*vacía?(l)=F ⇒ Def(mínimo(l))*

*vacía?(l)=T ⇒ mínimo( x:l ) = x*

*vacía?(l)=F ⇒ mínimo( x:l ) =*

*min(x, mínimo( l ))*

- Otra posibilidad para la ecuación con más de un elemento:

*mínimo( x:(y:l) ) = mínimo( min(x,y):l )*



## EJEMPLO 1-Pseudocódigo

Ejemplo: Suponiendo la operación que devuelve el elemento más pequeño de dos *min: elemento elemento → elemento*, obtener el elemento más pequeño de una lista.

```
func mínimo(l:lista):elemento
```

```
var    m:elemento
```

```
    aux:lista
```

```
    si vacia?(l) entonces error (Lista Vacía)
```

```
    sino m←prim(l)
```

```
        aux←resto(l)
```

```
        (...)
```

```
        {comparamos con el resto de la lista}
```



(....)

**mientras** ! (vacía?(aux)) **hacer**

$m \leftarrow \min( m, \text{prim}(aux) )$

$aux \leftarrow \text{resto}(aux)$

**finmientras**

**devolver** m

**finfunc**



## EJEMPLO 1-Pseudocódigo (2)

```
func mínimo(l:lista):elemento
    si vacia?(l) entonces error (Lista Vacía)
        sino si vacia?(resto(l))
            entonces devolver (prim(l))
        sino devolver min (prim(l), mínimo(resto(l)))
finfunc
```

¡Cuidado con la implementación concreta de “resto(l)”! Si la función devuelve una lista nueva todo va bien, pero si modifica al parámetro “l” (que suele ser lo normal) entonces hay un problema.

Esta observación puede extenderse al resto de ejemplos.



## EJEMPLO 2

Ejemplo: Comprobar si una lista está ordenada de menor a mayor.

- Esta operación puede considerarse parcial o total.

*min\_a\_max?: lista → bool*

- Las ecuaciones, utilizando propiedades, pueden ser:

*vacía?(l)=T ⇒ min\_a\_max?( l ) = T*

*vacía?(l)=F ⇒ min\_a\_max?( l ) =*

*(prim(l) eq mínimo(l)) ∧ min\_a\_max?( resto(l) )*

- Otra forma, usando generadoras de lista:

*min\_a\_max?( x:[] ) = T*

*min\_a\_max?( x:(y:l) ) =*

*(x ≤ y) ∧ min\_a\_max?( y:l )*



## EJEMPLO 2-Pseudocódigo

Comprobar si una lista está ordenada de menor a mayor.

```
func mín_a_max (l:lista):booleano
    si mínimo(l) eq prim(l)
        entonces devolver (mín_a_max(resto(l))
        sino devolver F
    finsi
finfunc
```



## EJEMPLO 2-Pseudocódigo (2)

```
func mín_a_max(l:lista):booleano
```

```
    devolver
```

```
mínimo(l) eq prim(l) Λ mín_a_max(resto(l))
```

```
finfunc
```



## **USO DE LAS LISTAS**

Con la especificación `LISTA[ELEMENTO]` creamos listas tal y como se representan: diferenciando la lista vacía de la lista que tiene un elemento en la cabeza.

Sin embargo, las listas no tienen requisitos en la forma en que se ponen los elementos, así que podrían ponerse al final de la lista o incluso en posiciones intermedias.

La siguiente especificación proporciona generadoras alternativas para la generación de las listas y una operación para determinar el tamaño de la lista.



## ESPECIFICACIÓN: LISTAS2

*{Esta especificación añade formas distintas de crear listas}*

**espec** *LISTA2[ELEMENTO]*

**usa** *LISTA[ELEMENTO], NATURALES*

**operaciones**

*[ \_ ] : elemento  $\rightarrow$  lista*

*{lista unitaria}*

*\_#\_ : elemento lista  $\rightarrow$  lista*

*{añadir por la derecha}*

*\_++\_ : lista lista  $\rightarrow$  lista*

*{concatenar dos listas}*

*long : lista  $\rightarrow$  natural*

*{longitud de la lista}*

**var** *x, y : elemento*

*l, l' : lista*



## ESPECIFICACIÓN: LISTAS2 (2)

ecuaciones

$$[x] = x: []$$

$$x\#[] = x: []$$

$$x\#(y:l) = y: (x\#l)$$

$$[]++l = l$$

$$(x:l)++l' = x: (l++l')$$

$$\text{long}([]) = 0$$

$$\text{long}(x:l) = \text{suc}(\text{long}(l))$$

fespec



# ESPECIFICACIÓN: LISTAS2 (Otra forma de verlo)

ecuaciones

$$[x] = x: []$$

$$vacía?(l)=T \Rightarrow x\#l = x: []$$

$$vacía?(l)=F \Rightarrow x\#l = prim(l) : (x\#resto(l))$$

$$vacía?(l)=T \Rightarrow l++l' = l'$$

$$vacía?(l)=F \Rightarrow l++l' = prim(l) : (resto(l)++l')$$

$$vacía?(l)=T \Rightarrow long(l) = 0$$

$$vacía?(l)=F \Rightarrow long(l) = suc(long(resto(l)))$$

fespec



## EJEMPLO 3

Ejemplo: Invertir una lista.

- La operación invertir siempre es total.

*invertir: lista  $\rightarrow$  lista*

- Usando las operaciones de inserción nuevas haríamos...

*invertir( [] ) = []*

*invertir( x:l ) = x#invertir(l)*

- ... o también podríamos hacer

*invertir( [] ) = []*

*invertir( x:l ) = invertir(l) ++ [x]*



## **EJEMPLO 3-Pseudocódigo**

```
func invertir (l:lista):lista
    si vacia?(l) entonces
        devolver l
    sino
        prim(l) #invertir(resto(l))
    finsi
finfunc
```



## EJEMPLO 4

Ejemplo: Comprobar si una lista es simétrica.

- la operación (es observadora) es la siguiente:

*simétrica?*: *lista* → *bool*

- ¡Cuidado! No podemos hacer  
*simétrica?(l) = (l == invertir(l))*  
porque no sabemos comprobar si dos listas son iguales.

- Las ecuaciones (basadas en propiedades) podrían ser...

*long(l) = 0* ⇒ *simétrica?(l) = T*

*long(l) = suc(0)* ⇒ *simétrica?(l) = T*

*long(l) > suc(0)* ⇒

*simétrica?(l) = (prim(l) eq ult(l))*

*∧ simétrica?(eult(resto(l)))*



## EJEMPLO 4-Pseudocódigo

```
func simétrica(l:lista):booleano
    si vacia?(l) entonces
        devolver T
    si no
        devolver
            prim(l) eq ult(l) ^
                simétrica(resto(eult(l)))
    finsi
finfunc
```



## **EJEMPLO 5-Pseudocódigo**

Ejemplo: Insertar en orden en una lista ordenada

```
proc insertar_orden (e:elemento, l:lista,)
    {inserta de menor a mayor en una lista ordenada}
var aux:lista
    si !min_a_max(l)
        entonces error (Lista no ordenada)
    si no si vacia?(l) entonces Devolver e:l
    si no
        si min(e, prim(l)) eq e
            entonces Devolver e:l {es el primero}
        (...)
    finproc
```



## EJEMPLO 5-Pseudocódigo

(...) **si no** *{buscamos la posición en orden}*

**aux**  $\leftarrow$  [] *{creamos lista vacía}*

**mientras**

**!** vacia?(l)  $\wedge$  min (prim(l), e) eq prim(l)

**hacer**

**aux**  $\leftarrow$  aux++prim(l)

**l**  $\leftarrow$  resto(l)

**finmientras** *{en aux están los menores que e}*

**l**  $\leftarrow$  aux++ (e:l)

**Devolver** l

**finsi**

**finproc**



# **IMPLEMENTACIÓN DE LISTAS**

La implementación más habitual es la de **celdas enlazadas**:

- El tipo lista es un puntero a una celda
  - Si la lista es la vacía, el puntero es “NIL”
  - Si no, apunta a una celda que contiene el primer elemento y un puntero al siguiente elemento
- El puntero de la celda correspondiente al último elemento contiene el valor “NIL”.

Hay muchas formas de enriquecer la implementación de las listas, por ejemplo...

- Listas doblemente enlazadas
- Listas circulares
- Listas con accesos por bloques de elementos



# 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 ENLAZADAS. CONSTRUCTORAS

*{Crear una lista vacía [ ]}*

```
fun lista_vacia() dev l:lista
```



# LISTAS ENLAZADAS. CONSTRUCTORAS

*{Crear una lista vacía [ ]}*

```
fun lista_vacia() dev l:lista
  l.longitud ← 0
  l.primerio ← nil
ffun
```



## LISTAS ENLAZADAS. CONSTRUCTORAS (2)

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

```
fun unitaria(e: elemento) dev l:lista
```



## LISTAS ENLAZADAS. CONSTRUCTORAS (2)

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

```
fun unitaria(e: elemento) dev l:lista
var p: puntero a nodo-lista
    reservar(p)
    p^.valor ← e
    p^.sig ← nil
    l.primerο ← p
    l.longitud ← 1
ffun
```



## LISTAS ENLAZADAS. CONSTRUCTORAS (3)

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

```
proc añadir_izq(E e: elemento, l: lista)
```



## LISTAS ENLAZADAS. CONSTRUCTORAS (3)

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

```
proc añadir_izq(E e: elemento, l: lista)
var p: puntero a nodo-lista
    reservar(p)
    p^.valor ← e
    p^.sig ← l.primerο
    l.primerο ← p
    l.longitud ← l.longitud+1
fproc
```



## LISTAS ENLAZADAS. CONSTRUCTORAS (4)

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

```
proc añadir_der(E e: elemento, l: lista)
```



## LISTAS ENLAZADAS. CONSTRUCTORAS (4)

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

```
proc añadir_der(E e: elemento, l:lista)
var p, aux: puntero a nodo-lista
  reservar (p)
  p^.valor ← e
  p^.sig ← nil
  si es_lista_vacia(l) entonces
    l.primerο ← p
  si no aux ← l.primerο
    mientras aux^.sig ≠ nil hacer
      aux ← aux^.sig
    fmientras
      aux^.sig ← p
  fsi
  l.longitud ← l.longitud+1
fproc
```



## LISTAS ENLAZADAS. OBSERVADORAS

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

```
fun inicial(l: lista) dev e:elemento
```



## LISTAS ENLAZADAS. OBSERVADORAS

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

```
fun inicial(l: lista) dev e:elemento
  si es_lista_vacia(l) entonces
    error(Lista vacía)
  si no e ← (l.primer) ^.valor
  fsi
ffun
```



## LISTAS ENLAZADAS. OBSERVADORAS (2)

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

```
fun final(l: lista) dev e:elemento
```



## LISTAS ENLAZADAS. OBSERVADORAS (2)

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

```
fun final(l: lista) dev e:elemento
var p: puntero a nodo-lista
  si es_lista_vacia(l) entonces
    error(Lista vacía)
  si no p ← l.primerio
  mientras p^.sig ≠ nil hacer
    p ← p^.sig
  fmientras
    e ← p^.valor
  fsi
ffun
```



## LISTAS ENLAZADAS. OBSERVADORAS (3)

*{Ver si la lista es vacía y obtener su longitud **long**}*

```
fun es_lista_vacia(l: lista) dev b:bool
```

```
fun longitud(l:lista) dev n:nat
```



## LISTAS ENLAZADAS. OBSERVADORAS (3)

*{Ver si la lista es vacía y obtener su longitud **long**}*

```
fun es_lista_vacia(l: lista) dev b:bool
```

```
  b ← (l.primerio = nil)
```

```
ffun
```

```
fun longitud(l:lista) dev
```

```
  n:nat n ← l.longitud
```

```
ffun
```



## **LISTAS ENLAZADAS. MODIFICADORAS**

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

```
proc elim_inicial(l: lista)
```



## LISTAS ENLAZADAS. MODIFICADORAS

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

```
proc elim_inicial(l: lista)
  var p: puntero a nodo-lista
  si es_lista_vacia(l) entonces
    error(Lista vacía)
  si no
    p ← l.primer0
    l.primer0 ← p^.sig
    l.longitud ← l.longitud-1
    p^.sig ← nil      {por seguridad}
    liberar(p)
  fsi
fproc
```



## **LISTAS ENLAZADAS. MODIFICADORAS (2)**

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

```
proc elim_final(l: lista)
```



## LISTAS ENLAZADAS. MODIFICADORAS (2)

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

```
proc elim_final(l: lista)
var p, aux: puntero a nodo_lista
    si es_lista_vacia(l)
        entonces error(Lista vacía)
    si no
        {hay elementos}
    p ← l.primerο
    si p^.sig = nil entonces {si solo hay uno}
        l.primerο ← nil
        liberar(p)
    si no {si hay más de uno}
        (...)
```



## LISTAS ENLAZADAS. MODIFICADORAS (3)

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

*(...)*

*{si hay más de un elemento}*

**mientras** (p^.sig)^.sig ≠ **nil** **hacer**

    p ← p^.sig

**fmientras**

aux ← p^.sig                   *{aux es el último}*

p^.sig ← **nil**                   *{rompemos el enlace}*

**liberar**(aux)

**fsi**

l.longitud ← l.longitud-1

**fsi**

**fproc**