



Ejercicio 1.- Extender la especificación de listas LISTA2[ELEMENTO], vista en clase, añadiendo las siguientes operaciones (pueden ser parciales):

- eliminar: elemento lista \rightarrow lista, que elimina todas las apariciones de un elemento en una lista.
- repeticiones: elemento lista \rightarrow natural, para calcular el número de veces que aparece un elemento en una lista.
- `_==_`: lista lista \rightarrow bool, que determina si dos listas son iguales.

Escribir en pseudocódigo estas operaciones partiendo únicamente de las operaciones de la especificación vistas en clase.

eliminar: elemento lista \rightarrow lista

func eliminar (e:elemento, E/S l:lista): lista {recursiva}

si (!vacía(l)) **entonces**

si (prim(l) eq e) **entonces dev** eliminar(e, resto(l))

sino dev prim(l): eliminar(e, resto(l))

finsi

sino dev l {si es vacía devuelve la lista vacía}

finsi

finfunc

func eliminar (e:elemento, l:lista):lista {Iterativa}

var laux:lista

laux \leftarrow []

mientras !vacía(l) **hacer**

si !(prim(l) eq e) **entonces**

laux \leftarrow prim(l)#laux

finsi

resto(l)

finmientras

dev laux

finfunc



repeticiones: elemento lista \rightarrow natural

func repeticiones (e:elemento, l:lista):natural **{recursiva}**

si vacia(l) **entonces devolver** 0

si no **si** (prim(l) eq e) **entonces**

devolver suc(repeticiones (e, resto(l))

si no

devolver repeticiones (e, resto(l))

finsi

finsi

finfunc

func repeticiones (e:elemento, l:lista):natural **{Iterativa}**

var rep:natural

 rep \leftarrow 0

mientras !vacía (l) **hacer**

si (prim(l) eq e)

entonces rep \leftarrow rep+1

finsi

 resto(l)

finmientras

dev rep

finfunc

Nota: Dependiendo del paso de parámetros, la lista puede destruirse en ambas funciones.



`_==_ : lista lista → bool` {recursiva}

func iguales (l1,l2:lista):booleano

si vacia?(l1) \wedge vacia?(l2) **entonces devolver** T

si no **si** vacia?(l1) \vee vacia?(l2) **entonces devolver** F

si no **si** !(prim(l1) eq prim(l2)) **entonces devolver** F

si no

 resto(l1)

 resto(l2)

devolver iguales (l1,l2)

finsi

finsi

finsi

finfunc

Nota: Dependiendo del paso de parámetros, las listas pueden destruirse en ambas funciones.

func iguales (l1,l2:lista):booleano {Iterativa}

var seguir:booleano

seguir \leftarrow T

mientras !vacia?(l1) \wedge !vacia?(l2) \wedge seguir **hacer**

si ! (prim(l1) eq prim(l2)) **entonces** seguir \leftarrow F

sino resto(l1)

 resto(l2)

finsi

finmientras

devolver vacia?(l1) \wedge vacia?(l2) \wedge seguir

finfunc



Escribir en pseudocódigo estas operaciones utilizando la representación con memoria dinámica vista en clase.

func repeticiones (e:elemento, l:lista):natural {Iterativa}

var rep:natural **paux**:**puntero a nodo**_lista

 rep ← 0

si !vacía (l) **entonces**

 paux ← l.primer

mientras !(paux=nil) **hacer**

si (paux^.valor eq e) **entonces** rep ← rep+1 **finsi**

 aux ← aux^.sig

finmientras

finsi

devolver rep

finfunc

proc eliminar (e:elemento, E/S l:lista) {Iterativa}

var primero:elemento laux, borrado:**puntero a nodo**_lista

si !vacía (l) **entonces** {eliminar e en la *primera* posición}

mientras (!vacía(l) ∧ l.primer^.valor eq e) **hacer** elim_inicial(l)

 laux ← l.primer {eliminar e en posición *distinta* de la primera}

mientras !laux^.sigue=nil **hacer**

si (laux^.sigue^.valor eq e) **entonces**

 borrado ← laux^.sigue

 laux^.sigue ← laux^.sigue^.sigue

 borrado^.sigue ← nil liberar(borrado)

finsi

 laux ← laux^.sigue

finmientras

finsi **finproc**



```
func iguales (l1,l2.lista):booleano                                {Iterativa}
var laux1, laux2:puntero a nodo_lista
si !vacía?(l1)  $\wedge$  !vacía?(l2) entonces
    seguir  $\leftarrow$  T
    laux1  $\leftarrow$  l1.primerio    laux2  $\leftarrow$  l2.primerio
    mientras !laux1=nil  $\wedge$  !laux2=nil  $\wedge$  seguir hacer
        si ! (laux1^.valor eq laux2^.valor) entonces seguir  $\leftarrow$  F
        sino    laux1  $\leftarrow$  laux1^.sig
                laux2  $\leftarrow$  laux2^.sig
    finsi
finmientras
finsi
devolver (laux1=nil)  $\wedge$  (laux2=nil)  $\wedge$  seguir
                                {si solo una es vacía no son iguales}
finfunc
```



Ejercicio 2.- Extender la especificación del TAD básico LISTA[BOOLEAN] con operaciones adicionales para:

- a) `maximo_seguidos`: lista \rightarrow natural, que obtiene cuál es la mayor cantidad de booleanos iguales seguidos que se encuentra en la lista; por ejemplo (simplificado), `maximo_seguidos(FFTFTTFFFTTF)` = 3 por la secuencia FFF.
- b) `reducir_datos`: lista \rightarrow lista, que reduce todas las secuencias de booleanos iguales que están seguidos a un único dato, es decir, se reducen los trozos seguidos; por ejemplo (simplificado), `reducir_datos(FFTFTTFFFTTF)` = FTFTFTF.

```
func máximo_seguidos(l:lista):natural
var    máximo, cont:natural
        primero:elemento
si vacia?(l) entonces devolver 0
sino    máximo  $\leftarrow$  1
            cont  $\leftarrow$  1
            primero  $\leftarrow$  prim(l)    resto(l)
            mientras !vacia?(l) hacer
                si (primero eq prim(l)) entonces cont  $\leftarrow$  cont+1
                sino    si máximo < cont                {la secuencia es mayor}
                        entonces    máximo  $\leftarrow$  cont
                fin si
                cont  $\leftarrow$  1                {contar nueva secuencia}
                fin si
                primero  $\leftarrow$  prim(l)    resto(l)
            finmientras
            si máximo < cont                {la última secuencia es mayor}
                entonces    máximo  $\leftarrow$  cont
            fin si
            devolver(máximo)
fin func
```



Para hacerlo *recursivo* necesitamos dos operaciones auxiliares:

contar_seguidos:lista → natural {cuenta los elementos seguidos al principio de la lista}

quitar_seguidos:lista → lista {quita los elementos iguales al principio de la lista}

func contar_seguidos(l:lista):natural

var primero:elemento

si vacia?(l) **entonces devolver** 0

sino primero ← prim(l) resto(l)

si vacia?(l) **V** !(primero eq prim((l)) **entonces devolver** 1

sino **devolver** 1+contar_seguidos(l)

finsi

finsi

finfun

func quitar_seguidos (l:lista) {recursiva}

var primero:elemento {quita los elementos iguales al principio de la lista}

si !vacia?(l)

entonces

primero ← prim(l) resto(l)

si !vacia?(l) **entonces**

si (primero eq prim(l)) {repetido, seguimos quitando}

entonces **dev** quitar_seguidos(l)

sino dev l {no repetido, no seguimos quitando}

finsi

sino dev l {la lista es vacía}

finsi

sino dev l {la lista es vacía}

finsi

finfunc



{suponemos conocida la especificación de la operación máximo de dos naturales}

func máximo_seguidos(l:lista):natural {recursiva}

var seguidos:natural

si vacia?(l) **entonces** devolver 0

sino seguidos \leftarrow contar_seguidos(l)

devolver máximo(seguidos, máximo_seguidos(quitar_seguidos(l)))

finsi

finfunc

reducir_datos: lista \rightarrow lista {quita los datos iguales consecutivos dejando uno}

func reducir_datos(l:lista):lista

{Iterativa}

var primero:elemento

laux:lista

laux \leftarrow []

si !vacia?(l) **entonces**

laux \leftarrow prim(l)#laux

resto(l)

mientras !vacia?(l) **hacer**

si !(prim(l) eq ult(laux)) **entonces**

laux \leftarrow prim(l)#laux

finsi

resto(l)

finmientras

finsi

dev laux

finfunc



```
func reducir_datos (l:list):lista                                {recursiva}
var primero:elemento
    si !vacía?(l) entonces
        primero ← prim(l)    resto(l)
        si !vacía(l)
            entonces
                si    (primero eq prim (l))    {hay datos consecutivos repetidos}
                    dev primero:reducir_datos(quitar_seguidos(l))
                sino    dev prim(l):reducir_datos(l)
            finsi
        sino dev l                                {la lista es vacía}
    finsi
    sino dev l                                {la lista es vacía}
finsi
finfunc
```



Ejercicio 3.- Escribir en pseudocódigo la operación `insertar_en_orden`, que inserta un elemento en una lista ordenada (Ejemplo 5 en Tema 4 Listas Básicas).

- a) Utilizando la implementación con memoria dinámica para lista simplemente enlazada vista en clase.
- b) Utilizando la implementación con memoria dinámica para lista doblemente enlazada vista en clase.

Utilizando la implementación con memoria dinámica para lista simplemente enlazada:

```
proc insertar_orden (E/S l:lista, e:elemento)      {inserta de menor a mayor}
var p, aux:puntero a nodo_lista
    reservar(p)
    p^.valor ← e
    p^.sigue ← nil
si vacia(l) entonces l.primerο ← p
sino   si menor(e, l.primerο^.valor) {es el primero}
    entonces
        p^.sigue ← l.primerο
        l.primerο ← p
    sino                                     {buscamos la posición en orden}
        aux ← l.primerο

        mientras ! (aux^.sigue=nil) ^ menor (aux^.sigue^.valor, e) hacer
            aux ← aux^.sigue
        finmientras
        {aux apunta a la posición anterior a la de elemento en la lista}
        p^.sigue ← aux^.sigue
        aux^.sigue ← p
        {Es importante el orden de estas asignaciones!}

    finsi
    l.longitud ← l.longitud+1
finproc
```



- Utilizando la implementación con memoria dinámica para lista doblemente enlazada:

proc insertar_orden (E/S l:listad, e:elemento) {inserta de menor a mayor}

var p, aux:**puntero a nodo_listad**

reservar(p)

p^.valor←e

p^.sigue←nil

p^.ant←nil

si vacia(l) **entonces** l.primerο←p l.ultimo←p

sino

si menor(e, l.primerο^.valor) {es el primero}

entonces

p^.sigue←l.primerο

l.primerο^.ant←p

l.primerο←p

sino {buscamos la posición en orden}

aux←l.primerο

mientras ! (aux =nil) ^ menor (aux^. valor, e) **hacer**

aux←aux^.sigue

finmientras

si aux=nil **entonces** {e va detrás del último}

l.ultimo^.sigue←p

p^.ant←l.ultimo

l.ultimo←p

sino

{aux apunta a la posición posterior a la de elemento en la lista}

p^.sigue←aux

aux^.ant^.sig←p

p^.ant←aux^.ant

aux^.ant←p {Es importante el orden de estas asignaciones!}



finsi

$l.\text{longitud} \leftarrow l.\text{longitud} + 1$

finproc



Ejercicio 4.- Suponiendo conocidas las operaciones \leq : elemento elemento \rightarrow bool, y mínimo: lista \rightarrow elemento, especificar operaciones para:

- ordenar una lista de menor a mayor usando el método de selección.
- ordenar una lista de menor a mayor usando el método de inserción (aunque no es necesario, puede ser útil tener un acumulador para las ordenaciones parciales).
- ordenar una lista de menor a mayor usando el método de ordenación rápida o Quicksort, separando los datos de la lista en “pequeños” (menores que un pivote) y “grandes” (mayores que un pivote).

proc ordenar_selección (E/S l:lista)

var min:elemento

```
    si !vacía?(l) entonces      min ← mínimo(l)
                                l ← quitar(min,l)
                                min:ordenar_selección(l)
```

finsi

finproc

proc ordenar_inserción_aux (E/S lord, l:lista)

var primero:elemento

```
    si !vacía?(l) entonces
        primero ← prim(l)
        ordenar_inserción_aux(insertar_orden(primerο, lord), resto(l))
```

finsi

finproc

proc ordenar_insercion(E/S lo, l:lista) {devuelve en lo la lista l ordenada}

```
    si !vacía?(l) entonces
        lo ← unitaria(prim(l)) resto(l)
        ordenar_insercion_aux(lo, l)
```

finsi

finproc



```
proc quicksort (E/S l)
var   pivote:elemento
      peq, gran:lista
si !(vacía?(l) V longitud(l) =1) entonces
      pivote ← prim(l)
      resto(l)
      peq ← lista_vacia
      gran ← lista_vacia
      divide (l, pivote, peq, gran)
      l ← quicksort(peq) ++ pivote ++ quicksort(gran)
finsi
finproc

proc divide(E/S l, pivote, peq, gran)
      mientras !(vacía?(l)) hacer
      si (prim(l) ≤ pivote) entonces peq ← prim(l):peq
      sino gran ← prim(l):gran
      finsi
      resto(l)
      finmientras
finproc
```