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

- eliminar: elemento lista → lista, que elimina todas las apariciones de un elemento en una lista.
- repeticiones: elemento lista → natural, para calcular el número de veces que aparece un elemento en una lista.
- _==_: lista lista → 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 → lista

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

var dato: elemento

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

dato ← prim(l)

resto(l)

si (dato eq e) **entonces eliminar(e,**l)

si no

eliminar(e,l)

l ← dato:l

finsi

finproc

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

{elimina todas las apariciones de e en la lista l}

var laux:lista

laux ← [ ]

mientras !vacia(l) **hacer**

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

laux←prim(l)#laux

finsi

resto(l)

```
        finmientras
        l←laux
finproc
```

repeticiones: elemento lista → natural

```
func repeticiones (e:elemento, l:lista):natural                    {recursiva}
        si vacia(l) entonces devolver 0
        si no   si (prim(l) eq e) entonces
                        resto(l)
                        devolver suc(repeticiones (e, l)
                si no
                        resto(l)
                        devolver repeticiones (e, l)
                finsi
        finsi
finfunc

func repeticiones (e:elemento, l:lista):natural                    {Iterativa}
var rep:natural
        rep←0
        mientras !vacia (l) hacer
                si (prim(l) eq e)
                        entonces rep←rep+1  resto(l)
                sino  resto(l)
                finsi
        finmientras
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) ∧ vacia?(l2) **entonces devolver** T

si no **si** vacia?(l1) **V** 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

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

var seguir:boolean

seguir←T

mientras !vacia?(l1) ∧ !vacia?(l2) ∧ seguir **hacer**

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

sino resto(l1)

resto(l2)

finsi

finmientras

devolver vacia?(l1) ∧ vacia?(l2) ∧ 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 paux←l.primero
    mientras !(paux=nil) hacer
        si (paux^.valor eq e) entonces
                                    rep←rep+1
                                    aux←aux^.sig
        sino
                                    aux←aux^.sig
        finsi
    finmientras
devolver rep
finfunc


proc eliminar(e:elemento, E/S l:lista)
    {Iterativa}
var primero:elemento  laux, borrado:puntero a nodo_lista
    si !es_lista_vacia (l) entonces          {eliminar e en la primera posición}
    mientras (!vacía(l) ∧ l.primero^.valor eq e)) hacer elim_inicial(l)
                                {eliminar e en posición distinta de la primera}
        laux←l.primero
        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
    seguir←T
    luax1←l1.primero
    laux2←l2.primero
    mientras !laux1=nil ∧ !laux2=nil ∧ seguir hacer
        si ! (laux1^.valor eq laux2^.valor) entonces seguir←F
        sino    laux1←laux1^.sig
                laux2←laux2^.sig
        finsi
    finmientras
    devolver (laux1=nil) ∧ (laux2=nil) ∧ 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 → 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 → 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            maximo←1
                cont←1
                primero←prim(l)
                resto(l)
                mientras !vacia?(l) hacer
                        Si (primero eq prim(l))entonces cont←cont+1
                        Sino si máximo<cont                 {la secuencia es mayor}
                                entonces        máximo←contador
                                finsi
                                cont←1                      {contar nueva secuencia}


                        finsi
                        primero←prim(l)
                        resto(l)
                finmientras
                si máximo<cont                      {la última secuencia es mayor}
                                entonces        máximo←contador
                finsi
                devolver(máximo)
finsi
finfunc
```

Para hacerlo *recursivo* necesitamos dos operaciones auxiliares. contar_seguidos y quitar_seguidos:

contar_seguidos.lista→natural {cuenta cuantos elementos hay 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

proc quitar_seguidos(E/S l:lista)
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))
                                entonces        {son repetidos}
                                    quitar_seguidos(l)
    finsi
finproc
{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← contar_seguidos(l),
                quitar_seguidos(l)
                devolver máximo(seguidos, máximo_seguidos(l))
        finsi
finfunc
```

reducir_datos: lista → lista

```
proc reducir_datos(E/S l:lista)
{Iterativa}
var     primero:elemento
        laux:lista

        laux←[ ]
        si !vacia?(l) entonces
                laux←prim(l)#laux
                resto(l)
                mientras !vacia?(l) hacer
                        si !(prim(l) eq ult(laux)) entonces
                                laux←prim(l)#laux
                        finsi
                        resto(l)
                finmientras
        finsi
        l←laux
finproc
```

```
proc reducir_datos (E/S l:lista)                                {recursiva}
var primero:elemento
        si !vacia?(l) entonces
                primero←prim(l)
                resto(l)
                si !vacia(l) Λ (primero eq prim (l))
                        entonces
                        quitar_seguidos(l)
                        primero:reducir_datos(l)
                sino prim(l):reducir_datos(l)
```

```
        finsi
finproc
```

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.

c) 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.primero←p
sino    si menor(e, l.primero^.valor) {es el primero}
        entonces
                p^.sigue←l.primero
                l.primero←p
        sino                                    {buscamos la posición en orden}
                aux←l.primero
                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.primero←p l.ultimo←p

sino

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

entonces

p^.sigue←l.primero

l.primero^.ant←p

l.primero←p

sino {buscamos la posición en orden}

aux←l.primero

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.longitud←l.longitud+1

finproc

Ejercicio 4.- Suponiendo conocidas las operaciones _≤_: elemento elemento→ bool, y mínimo: lista→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 !vacia?(l) **entonces** min←minimo(l)

l←quitar(min,l)

min:ordenar_selección(l)

finsi

finproc

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

var primero:elemento

si !vacia?(l) entonces

```
                primero←prim(l)
                ordenar_inserción_aux(insertar_orden(primero, lord), resto(l))
        finsi
finproc
proc ordenar_insercion(E/S lo, l:lista)          {devuelve en lo la lista l ordenada}
        si !vacia?(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 !(vacia?(l) V longitud(l) =1) entonces
                pivote←prim(l)
                resto(l)
                peq←lista_vacia
                gran←lista_vacia
                divide (l, pivote, peq, grn)
                l←quicksort(peq)++pivote++quicksort(gran)
        finsi
finproc

proc divide(E/S l, pivote, peq, gran)
        mientras !(vacia?(l)) hacer
```

si (prim(l) <=pivote) **entonces** peq←prim(l):peq

sino gran←prim(l):gran

finsi

resto(l)

finmientras

finproc