Unidad 4: Listas y sus aplicaciones

1. Implementar una función no recursiva en C que reciba un puntero a una lista y proporcione el número de nodos de una lista enlazada. Escribir primero su pseudocódigo y considerar posibles situaciones de error. Suponga la siguiente estructuras y tipos en C:

```
// En lista.h
typedef struct _LISTA LISTA;

// En lista.c
typedef struct nodo {
     Elementro *info;
     struct nodo *next;
} NODO;

struct _LISTA {
      NODO *pn;
}
```

```
int numNodosLista (LISTA *pl) {
     NODO *pn;
     int cont=0;

     for(pn=pl->pn; pn !=NULL;  cont++, pn=pn->next);
     return cont;
}
```

2. Implementar una función **recursiva** en C que reciba un puntero a una lista enlazada y proporcione el número de nodos de la lista. Escribir primero su pseudocódigo y considerar posibles situaciones de error.

```
int numNodosLista (LISTA pl) {
     return numNodosLista_rec ( FIRST(pl) )
}

int numNodosLista_rec (NODO n ) {
     if  n== NULL return 0;
     return  1 + numNodosLista_rec (NEXT(l));
}
```

```
int numNodosLista (LISTA *pl) {
     return numNodosLista_rec (pl->pn);
}

int numNodosLista_rec (NODO *pn) {
     if (pn==NULL) return 0;
     return  1 + numNodosLista_rec( pn->next);
}
```

3. Implemente una función con complejidad O(N) que imprima el contenido del campo info de los nodos de una lista enlazada en orden inverso, es decir empezando por el último nodo de la lista. Dar el pseudocódigo y el código C.

```
// Pseudocódigo  versión recursiva

void printInverso (Lista l) {
     return printInverso_rec (FIRST(l) )
}


int printInverso_rec (NODO n) {
     if n == NULL  return 0;

     printInverso_rec (NEXT (pl))
     print_Elemento (INFO (pl))
}

// Código C. Versión recursiva (1)
int printInverso (Lista *pl) {
  return printinverso (pl->pn);
}

void printInverso_rec (NODO *pn) {
     if (pn==NULL) return 0;
     printInverso_rec (pl->next);
     print_Elemento  (pl->info);    //  imprime  el
     info
}
```

```
// Código C.  Versión no-recursiva. Requiere una pila. No CdE
#include "pila.h"

int printInverso (LISTA *pl) {
  PILA *p;
  Elemento *e;

   p = pila_ini( ):

  while  (lista_vacia(pl) == FALSE) {
 e = lista_extractIni (pl); //asumo no se crea memoria
    push (p, e);
    elemento_libera (e);
  }

 // imprime el contenido de la pila
  while (pila_ vacia(p) == FALSE) {
    e = pop(p);
    print_Elemento (e);
    lista_insertIni (e, pl) ;
    elemento_libera (e);
  }
  pila_libera(p);
  return OK:
 }
```

4. Escribir un algoritmo que invierta una lista, de modo que el último nodo se convierta en el primero y así sucesivamente. Dar el pseudocódigo y el código C.

5. Suponga que se utiliza la siguiente estructura de datos para implementar una lista

```
typedef struct _LISTARARA LISTARARA;

struct _LISTARARA {
     NODO *front;  //guarda la dirección primer nodo de la lista
     NODO *rear;  //guarda la dirección segundo nodo de la lista
};
```

a) Dar el código C de la primitivas que inicializan una lista, determinan si una lista está vacía, borra el primer nodo de la lista, inserta un nodo al final de la lista e inserta un nodo al principio de la lista.

```
STATUS listarara_insFront (LISTARARA *pl, Elemento *pe) {
  NODO *pn;

  if (!pl || !pe) return ERROR;

  pn=getNode();
  if (pn==NULL) return ERROR;

  pn->info= elemento_copy(pe);
  if (pn->info==NULL) {
       free(pn);
      return ERROR;
 }
  //Si la lista esta vacia
  if (pl->front==NULL) {
     pl->front=pl->rear=pn;
     return OK;
   }

   pn->next = pl->front;
   pl->front=pn;
   return OK;
}
```

```
STATUS listarara_insEnd (LISTARARA *pl, Elemento *pe)
{
  NODO *pn;

  if (!pl || !pe) return ERROR;

  pn=getNode();
  if (pn==NULL) return ERR;

  pn->info= elemento_copy(pe);
 if (pn->info==NULL) {
       free(pn);
      return ERROR;
  }
  //Si la lista esta vacia
  if (pl->front==NULL) {
     pl->front=pl->rear=pn;
     return OK;
   }

   pl->rear->next = pn;
   pl->rear=pn;
   return OK;
}
```

b) Discutir las ventajas de la estructura de datos anterior sobre una lista enlazada para implementar una cola.

Tanto las operaciones de inserción como extracción de la cola tienen complejidad O(1).

c) Utilizando la estructura de datos anterior proporcione el fichero `cola.h`

```
// cola.h no cambia

#ifndef _COLAH_
#define _COLAH_

#include “elemento.h”

typedef struct _COLA COLA;
```

```
// y los prototipos de las  primitivas de cola....

COLA *cola_ini();
void cola_liberar(COLA *pq);
STATUS cola_insert (COLA *pc, Elemento *pe);
Elemento *cola_extarct (COLA *pc);
Bool cola_vacia(COLA *pq);
Bool cola_llena(COLA *pq);

#endif
```

d) Utilizando la estructura de datos anterior proporcione el fichero `cola.c`

```
#include “listarara.h”
```

```
COLA *cola_ini () {
    COLA *pc;
```

```
struct _Cola {
   Listarara *pl;
}

Elemento *cola_extract (COLA *pc) {
    Elemento *e;
     if (pc==NULL) return ERROR;

    e = listarara_extractFront( pc->pl);
    return e;
}

STATUS cola_insert (COLA *pc, Elemento *pe) {
     if (pc==NULL) return ERROR;
     return listarara_insEnd( pc->pl,  pe);
}
```

```
    pc = (COLA *) malloc(sizeof(COLA));
    if (pc==NULL) return ERROR;

    pc->pl = listarara_ini( );
    if (pc->pl==NULL)  {
       free(pc);
       return NULL;
    }

    return pc;
 }

void cola_libera (COLA *pc){
  if (pc==NULL) return;
  listarara_libera(pc->pl);
  free(pc);
}
```

1. Dar el código C una función *Elemento *delAfter (NODO *pn)*, que devuelva el elemento del nodo siguiente al nodo apuntado por pn y después elimine el nodo. Considerar las posibles situaciones de error.

```
// versión1: No crea memoria para el elemento

Elemento *delAfter (NODO *pn) {
 Eloemento *e;
 NODO *paux;

 if (pn==NULL || pn->next==NULL ) return ERROR;

 e = pn->next->info;
 paux = pn->next;
 pn->next = paux->next;
 free (paux);

 return e;
}

//Version 2. Crea memoria para el elemento

Elemento *delAfter (NODO *pn) {
 Eloemento *e;
 NODO *paux;

 if (pn==NULL || pn->next==NULL ) return ERROR;

 e = elemento_copiar(pn->next->info);
 paux = pn->next;
 pn->next = paux->next;
 nodo_liberar (paux);

 return e;
}
```

2. Escribir una función C que reciba un puntero a una lista enlazada e intercambie las posiciones de sus nodos primero y último. Escribir primero su pseudocódigo y considerar posibles situaciones de error.

```
STATUS intercambiaLista(LISTA *pl) {
  NODO *pn, *ultimo;
 //Vacia o un unico nodo
 if (( lista_vacia(pl) ||  pl->pn->next == NULL )
   return ERROR;

 // Buscar penúltimo nodo
 for(pn=pl->pn; pn->next->next!=NULL; pn=pn->next);

 //Intercambiar, pl->pn contiene primer nodo
```

```
//Solución con primitivas

STATUS intercambiaLista(LISTA *pl) {
   Elemento *x1, *x2;
   //lista vacia
   if (x1= lista_extractEnd (pl)== ERROR) return
ERROR;

   //caso de un solo nodo
   if (x2 = lista_extractFront (pl) == ERROR) {
```

```
 ultimo= pn->next;
 pn->next= pl->pn;
 ultimo->next= pl->next;
 pl->pn->next= NULL;
 pl->pn= ultimo;
 return OK;
}
```

```
    lista_insertEnd (pl, x1);
    return ERROR;
  }

  lista_insertFront (pl, x1);
  lista_insertFront (pl, x2);
  return OK;
}
```

1. Escribir una función C que inserte un nodo después del i-ésimo nodo de una lista enlazada.

```
STATUS listaInsPos(LISTA *pl, int pos, const Elemento *e) {
        NODO *pn, *qn;
        int i;

                    if(!pl || !e || p<0) return ERROR;

        pn= getNode();
        if (pn == NULL) return ERROR;
        pn->info= elemento_copy(e);

    //insertar al inicio
        if (pos == 0) {
            pn->next= pl->pn;
                pl->pn= pn;
        }

        //Iteramos tantas veces como (pos - 1) hasta como mucho el último nodo
        for (i= 1, qn= pl->pn; (i < pos) && (qn->next != NULL); i++, qn= qn->next);

        if (i != pos) return ERROR;

        //Insertamos el nodo pn como siguiente
        pn->next= qn->next;
        qn->next= pn;
        return OK;
}
```

1. Escribir una función en C de prototipo *NODO *lista_insertNode (NODE *pn, const Elemento *aux)* que inserte un nodo después del nodo cuya dirección está dada por pn. La función debe devolver la dirección del nodo generado.

```
NODE *lista_insertNode (NODE *pn, const Elemento *e) {
  NODE *paux;

  if (pn == NULL || e==NULL) return NULL;

  paux= getNode();
  if (paux == NULL) return paux;

  paux->info= elemento_copy(e);       //crea memoria
  paux->next = pn->next;
  pn->next =paux;

 return paux;
}
```

Suponiendo las estructuras de datos y tipos definidos en el prblema 1 ¿Sería correcto el siguiente programa main.c sin control de errores? Justifique la respuesta.

```
//En lista.h
 typedef struct _LISTA LISTA;

 // En lista.c

#define FIRST(l) (l)->pn
```

```
// En main.c
#include "lista.h"

main( ) {
 Lista *pl;
 Elemento  *e;
```

```c
typedef struct nodo {
     Elementro *info;
     struct nodo *next;
 } NODO;

struct _LISTA {
      NODO *pn;
 }
```

```c
 pl=lista_ini( );
//inicializo el elemento y le doy valor
 e = elemento_ini( );
 elemento_set (e, "hola");
 //inserto el elemento en la lista
  lista_insertFront (pl, e);

  //inserto otra vez el elemento
  lista_insertNode (FIRST(pl), e);

  lista_libera(pl);
  elemento_libera (e);
}
```

No. En la función main en la sentencia `lista_insertNode (FIRST(pl), e);` se está tratando de acceder a un "tipo ínvalido" (tipo oculto). Por el mismo motivo tampoco sería váido la la sentencia `lista_insertNode (pl->pn, e);`

2. Escribir un algoritmo, LISTA `combinaOrden (const LISTA a, const LISTA b)`, que combine dos listas enlazadas ordenadas de menor a mayor, a y b, en una sola lista ordenada también de menor a mayor. No se deben modificar las listas originales. Dar primero su pseudocódigo y después su código en C. Analice la eficiencia del código. Supóngase para ello la siguiente definición de LISTA:

```c
// En lista.h
 typedef struct _LISTA LISTA;

 // En lista.c
typedef struct nodo {
     Elementro *info;
     struct nodo *next;
 } NODO;

struct _LISTA {
      NODO *pn;
 }
```

No hace falta considerar el control de errores.

> **<u>Ayuda</u>**: *Considere dos punteros a nodo pa y pb. Los punteros se inicializan al principio de cada una de las listas ordenadas de entrada. Se compara el contenido de los campos info de los punteros. El menor de los campos info se inserta en la lista de salida y se avanza el puntero hasta el siguiente nodo (solo el de menor campo info). Se repite el paso anterior hasta que se alcance el final de una de las dos listas. Finalmente se copian en la lista de salida los campos infos de la lista de entrada que no se había acabado de recorrer.*

```
// PSEUDOCODIGO (1)

LISTA combinaOrden (LISTA a, LISTA b) {

 pa = a
 pb = b

 laux = lista_ini ( )

 while pa != NULL AND pb != NULL
       //hayo el minimo y avanzo el puntero
   if INFO (pa) <= INFO (pb)
          min = INFO (pa)
          pa = NEXT (pa)
```

```
// PSEUDOCODIGO (2)

LISTA combinaOrden (LISTA a, LISTA b) {

 pa = a
 pb = b
 iniLista (laux)

 while pa != NULL OR  pb!= NULL
    if pa != NULL AND  pb!= NULL
         if INFO (pa) <= INFO (pb)
            listInsertEnd (laux, INFO(pa))
            pa = NEXT (pa)
         else
```

```
   else
            min = INFO (pb)
            pb = NEXT (pb)
  //inserto en la lista de salida el minimo
     listInsertEnd (laux, min)

 //hayo cual de las dos listas ha finalizado
  if   lista_vacia (pa) == TRUE
           pb = pa;

  //inserto en la lista de salida los elemntos de la
  // lista que no haya acabado de recorrer
 while  pb != NULL
        listInsertEnd (laux, INFO(pb))
        pb = NEXT (pb)

 return laux
```

```
           listInsertEnd (laux, INFO(pb))
           pb = NEXT (pb)
   else if pa !=NULL
         listInsertEnd (laux, INFO(pa))
         pa = NEXT (pa)
   else if pb !=NULL
         listInsertEnd (laux, INFO(pb))
         pb = NEXT (pb)

return laux
```

```
// CODIGO C. No eficiente O(N²)

#define INFO(a) (a)->info
#define NEXT(a) (a)->next


LISTA * combinaOrden (const LISTA *a, const LISTA *b) {
  NODO *pa, *pb;
  Elemento *min;

 LISTA *l;

  //crea la lista de salida
  l =lista_ini( );
  if (l==ERROR) return NULL;

  pa = a->pn;
  pb = b->pn;

 while (pa != NULL && pb != NULL) {
      //Obtengo el minimo y avanzo el puntero
  if ( elemento_cmp (INFO (pa), INFO (pb)) < 0) {
         min = INFO (pa);
         pa = NEXT (pa);
      }
 else {
         min = INFO (pb);
         pb = NEXT (pb);
     }
 //inserto al final de la lista de salida el minimo
     lista_insertEnd (l, min);
  }

 //Determino cual de las dos listas no ha finalizado
   if (pa != NULL) pb = pa;

   //vuelco en la lista de salida los elemntos de la
   // lista de entrada que no haya acabado de recorrer
 while (pb != NULL)  {
       lista_insertEnd (l, INFO(pb));
       pb= NEXT (pb)
   }

  return l;
}
```

```
// CODIGO C. Eficiente O(N).

/* Para evitar recorrer la lista cada vez que tengo
que insertar en la lista, guardo la
dirección del último nodo insertado.  */

#define INFO(a) (a)->info
#define NEXT(a) (a)->next

Lista *combinaOrden (const LISTA *a, const LISTA *b)
{
  NODO *pa, *pb, *paux;
  generic min;
  LISTA *l;

  //crea la lista de salida
  l =lista_ini( );
  if (l==ERROR) return NULL;

  pa = a->pn;
  pb = b->pn;

  while (pa != NULL && pb != NULL) {
     //hayo el minimo y avanzo el puntero
    if ( elemento_cmp (INFO (pa), INFO (pb)) < 0) {
         min = INFO (pa);
         pa = NEXT (pa);
      }
     else {
         min = INFO (pb);
         pb = NEXT (pb);

  //inserto al final de la lista de salida el minimo
     paux = lista_insertNode (paux, min);  //(*)
     paux = NEXT (paux);
  }

 //Determino cual de las dos listas no he finalizado
  if (pa != NULL) pb = pa;

  //vuelco en la lista de salida los elemntos de la
  //  lista  de  entrada  que  no  haya  acabado  de
recorrer
  while (pb != NULL)  {
       paux = lista_insertNode (paux, INFO(pb));
       puax = NEXT (paux)
       pb= NEXT (pb)
  }

 return paux;
}
```

3. Reescribir las funciones básicas sobre listas para su funcionamiento en listas circulares enlazadas.

4. Implemente las primitivas del TAD cola usando como EdD una lista enlazada de nodos circular. Proporcione los ficheros `cola.h` y `cola.c`. Discuta las ventajas de utilizar una lista enlazada circular frente a otras estructuras de datos (e.g lista enlazada de nodos y una tabla)

5. Implementar una función que reciba un puntero a un nodo de una lista enlazada e intercambie dicho nodo con el que le sigue. Escribir primero su pseudocódigo y considerar posibles situaciones de error

```
STATUS intercambiaNodoLista(LISTA *pl, NODO *pn){
        NODO *qn, **ant;
        ant= pl;
        for (qn= *pl; (qn != NULL) && (qn != pn); qn= qn->next){
                ant= &qn->next;
        }

        //Error ha llegado al final y no lo ha encontrado
        if (qn == NULL) return ERROR;

        //Error es el último nodo
        if (qn->next == NULL) return ERROR;

        //Intercambiar
        *ant= qn->next;
        qn->next= (*ant)->next;
        (*ant)->next= qn;
        return OK;
}
```

6. Flavio Josefo fue un famoso historiador del primer siglo. Aparentemente, su talento matemático le permitió vivir lo suficiente para alcanzar la fama. En efecto, durante la guerra judeoromana, Josefo formó parte de un grupo de 41 rebeldes judíos que fueron cercados por los romanos. Prefiriendo morir a rendirse, decidieron formar un círculo y procediendo en el sentido de las agujas de un reloj, matar a uno de cada tres rebeldes hasta que sólo quedaran dos que, en teoría, se suicidarían entonces. Sin embargo, a Josefo y a un amigo suyo esta idea no les parecía demasiado buena y decidieron situarse adecuadamente para ser los dos últimos supervivientes. ¿Cómo podrían Josefo y su amigo haber decidido las posiciones adecuadas de haber dispuesto de una tableta y de una implementación en C de listas circulares enlazadas? (*Solución: páginas 232 y 233 del libro Langsam et al.)*

7. Escribir una función C que intercambie el nodo i-ésimo con el que le sigue en una lista doblemente enlazada.

```
STATUS listaIntPos (LISTA *pl, int pos) {

    int i;
    if (listaVacia(pl)) return ERR;                    //lista vacía
    if (pl->pn->next == NULL) return ERR;    //Un nodo

    //Iteramos tantas veces como (pos - 1) hasta el penúltimo nodo
    for (i=0, pn= pl->pn; (i < pos) && (pn->next->next != NULL); i++, pn= pn->next);

    if (i != pos) return ERROR;
    //Intercambiamos pn con el siguiente
```

```
        qn= pn->next;
        pn->next= qn->next;
        qn->ant= pn->ant;
        pn->ant->next= qn;
        pn->ant= qn;
        qn->next= pn;
        return OK;
          }
```

8. Escribir una función C que reciba un puntero a una lista doblemente enlazada e intercambie las posiciones de sus nodos primero y último. Escribir primero su pseudocódigo y considerar posibles situaciones de error.

9. Escriba una función que inserte un dato en una lista enlazada ordenada decreciente

```
/* Inserta un dato en una lista ordenada */
STATUS lista_insOrder (LISTA *pl, const Elemento *x){
     NODO *pn, *qn=NULL;

     for( pn=pl->pn; pn!=NULL && elemento_compara (pn->info, x); pn=pn->next) {
               qn=pn;
     }
     if (qn==NULL)   //lista vacia
               return (lista_insFront (pl, x));
     else
               return (insAfter(qn, x));
}

//donde

/* inserta un dato despues de un NODO dado */
STATUS insAfter (NODO *pq, const Elemento *x) {
     NODO *pn;

     pn=getNode();
     if (pn==NULL) return ERROR;

     pn->info= elemento_copy(x);
        pn->next=pq->next;
     pq->next = pn;
     return OK;
}
```

10. Se desea implementar una cola de prioridad. En una cola de prioridad los elementos tienen asignada una determinada prioridad, de forma que los elementos de mayor prioridad salen antes de la cola. Entre elementos de igual prioridad el primero que saldrá de la cola será el primero que se incorporó.
(a) Proporcione el fichero *colaprioridad.h* con la interfaz del TAD.

(b) Proporcione el fichero *colaprioridad.c* utilizando como estructura de datos un array de listas donde el índice del array indica la prioridad de los elementos almacenados en la lista (siendo 0 la máxima prioridad).

```
// cola.h

#ifndef _COLAP_H_
#define _COLAP_H_

typedef struct _Colap Colap

COLAP * colap_ini();
void colap_eliminar (Colap *pc);
```

```
// cola.c

#include "lista.h"

#define MAXPRIORITY 100

struct _COLAP {
     lista *lprior[MAXPRIORITY];
}
```

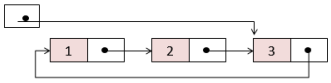
```
STATUS colap_insert (Colap *pc, Elemento *pe);
Elemento *colap_extract (Colap *pc);
Bool cola_vacia(Colap *pq);
Bool cola_llena(Colap *pq);

#endif
```

```
Colap * colap_ini() {
   int i;
   int flag=1;

   Colap *pc;

  pc = (Colap*) malloc(sizeof(Colap));
  if (pc==NULL) return NULL;

   for (i=0; i< MAXPRIORITY && flag==1; i++)  {
      if (lista_ini(lprior[i]==NULL) flag=0;
   }

   //libera si ha habido error
    if (flag==1) {
        for(j

STATUS colap_insert (Colap *pc, Elemento *pe) {
    int prior;

    if (!pc || !pe) return ERROR;

    prior = elemento_getPriority(pe);
    if (esPrioridadOk (pior) == FALSE) return ERROR;

    return ( lista_insert (lprior[prior], pe) );
}

Elemento *colap_extarct (COLAP *pc) {
    int i;
    Elemento *e;
    if (!pc) return ERROR;

    for (i=0; i<MAXPRIORITY; i++) {
     e = cola_extract (lprior[i]);
     if (e) return e;
   }
    return NULL;
}
```

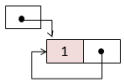
11. Se desea implementar en C el **TAD Lista Enlazada Circular (LEC).**

a) Dibuja un esquema de una LEC que tenga:

3 nodos:



1 nodo:



0 nodos:

Dada la siguiente estructura de datos para implementar un nodo en C y la siguiente definición del tipo Nodo:

```
struct _Nodo {
   Elemento *info;
   struct _Nodo *next;
};
typedef  struct _Nodo   Nodo;
```

b) Escribe en C la estructura de datos (EdD) necesaria para implementar una lista enlazada circular (LEC) y define un nuevo tipo de dato en C llamado Lista.

```
struct _Lista {
   Nodo *first;
};

typedef struct _Lista Lista;
```

a) Implementa en C la primitiva

```
Boolean listaCircular_ordenada (ListaCircular *pl);
```

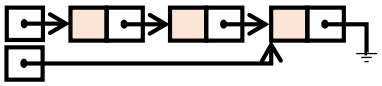
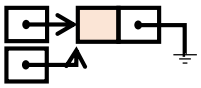
que chequea si los elementos de la lista están ordenados de menor a mayor, o no.
Puedes usar la primitiva del TAD elemento

```
Boolean elemento_menor_o_igual(Elemento *pe1, Elemento *pe2);
```

que devuelve True si el elemento 1 es menor o igual que el elemento 2, y False en caso contrario.

```
Boolean listaCircular_ordenada (ListaCircular *pl) {

        Nodo *pn;

        if (!pl ) return FALSE;
        if (! last(pl)  ||  (last(pl) == next(last(pl)) ) return TRUE;

        //pn empieza apuntando al primero
        for (pn=next(last(pl));  pn != last(pl);  pn=next(pn), i++)
                if(elemento_menor_o_igual(info(pn), info(next(pn)) == FALSE)
                        return FALSE;
        return TRUE;
}
```

c) Dibuja el esquema de este tipo de Lista Enlazada Simple con 2 puntos de acceso (Lista2), si tiene:

<table>
<tr><td colspan="2">3 nodos  </td></tr>
<tr><td>1 nodo  </td><td>0 nodos  </td></tr>
</table>

Imagina que, como alternativa al TAD LEC, se implementa una **Lista Enlazada Simple** (no circular) con 2 puntos de acceso **(Lista2)**, para poder tener acceso directo tanto al primer nodo como al último, mediante sendos punteros**:**

d) Escribe en C la estructura de datos (EdD) necesaria para implementar una lista de este tipo y define un nuevo tipo de dato en C llamado Lista2.

```
struct _Lista {
   Nodo *first, *last;
};
typedef struct _Lista Lista2;
```

e) Implementa en C (con control de errores) la función status lista_insertFin (Lista2 *pl, Elemento *pe); que inserte una copia del elemento pe en la lista a la que hace referencia pl.

```
status lista_insertFin (Lista2  *pl, const Elemento *pe){
  Nodo *pn = NULL;

  if (!pl || !pe) return ERROR;

  pn = nodo_crear();          // Se crea el nodo pn a insertar
  if (!pn) {
      return ERROR;
  }

  info(pn) = elemento_copiar(pe);
  if (!info(pn)) {
      nodo_liberar(pn);
      return ERROR;
  }
  // Caso 1: lista vacía
  if (lista_vacia(pl) == TRUE) {
      first(pl) = last(pl) = pn;
               // Ambos punteros apuntan al nuevo nodo
  }
  // Caso 2: lista no vacía
  else {
      next(last(pl)) = pn;
       // El siguiente al último en la lista es el nuevo nodo
      last(pl) = pn; // El último de la lista pasa a ser pn
  }

  return OK;
}
```