Unidad 4: Listas

Listas enlazadas

1. Implementar con control de errores una función no recursiva en C llamada list_tam que devuelva el número de nodos de la lista enlazada proporcionada. Suponga las siguientes estructuras y tipos en C:

```
// En list.h
typedef struct _List List;

// En list.c
typedef struct _Node {
   Element *info;
   struct _Node *next;
};
typedef struct _Node Node;

struct _List {
   Node *first;
};
```

2. Implementar recursivamente la función anterior.

3. (a) Implementar list_tam(List *pl) de una manera no recursiva y como función derivada, usando solamente las primitivas de List (no se puede acceder a la EdD). Asumir que list_insertX devuelve ERROR cuando no se puede insertar un Elemento en la lista por error en la reserva de memoria.
(b) Discutir ventajas e inconvenientes de una implementación como función derivada, basada en llamadas a las funciones disponibles en la interfaz, frente a una implementación como primitiva, que tenga permiso para acceder a la estructura de datos.
(c) Evaluar el coste (complejidad) de la función anterior.

```
/* Como solo podemos usar las primitivas, no queda más remedio que extraer todos
los elementos e ir contándolos. Si extraigo e inserto en la misma lista, ¿cuándo
paro de contar? Sería un bucle infinito... Por tanto, hay que extraer de la lista
los elementos uno a uno e ir insertándolos en una lista auxiliar para no perderlos.
A la vez, iremos contando cuántos elementos hay. Después, tocará restaurar la lista
inicial, extrayendo uno a uno los elementos de la lista auxiliar e insertándolos de
nuevo en la lista original. Como es más rápido insertar y extraer por el inicio,
usaremos esas funciones (reflexiona si quedan bien colocados los elementos tras
esas llamadas) */

int list_tam(List *pl) {
    List *pl2 = NULL;
    Element *pe = NULL;
    int tam = 0;

    if (!pl) return -1;

    if (list_isEmpty(pl) == TRUE) return 0;

    // Creamos una list auxiliar pl2
    pl2 = list_ini();
    if (!pl2) return -1;

    // Extraemos cada elemento de pl insertándolo en pl2 e incrementamos tam en 1

    while (list_isEmpty(pl)==FALSE) {
        pe = list_extractIni (pl);
        if ( list_insertIni(pl2, pe) == ERROR) {
             list_free(pl2); // CdE. Liberamos la list auxiliar
             return -1;
        }
        element_free(pe); //list_insertar copia el elemento-->ya no hace falta
        tam++;
    }

    /* Recuperamos los elementos de pl extrayéndolos de pl2, para dejar la lista
       original como estaba*/
    while (list_isEmpty(pl2)==FALSE) {
        pe = list_extractIni(pl2);
        if (list_insertIni(pl, pe)==ERROR) {
             list_free (pl2); // CdE. Liberamos la lista auxiliar
             return -1;
        }
       element_free(pe); //list_insertar copia el elemento-->ya no hace falta
    }

   // Liberamos los recursos auxiliares
    list_free(pl2);

    return tam;
}
```

4. Implementar 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. La función devolverá el número de caracteres impresos por pantalla. Dar el pseudocódigo y el código C.

```
// Pseudocódigo versión recursiva.
// Nota: en pseudocódigo usamos l para referirnos o bien a la lista o bien a un nodo
// cualquiera (simplificación).
/*
   Para imprimir en orden inverso, dado un nodo habrá que imprimir primero todo el resto
   (desde next de ese nodo) y después el campo info del propio nodo --> primero llamamos
   a la función con el next y después imprimimos la info.
*/
entero printInverse (List l)
    si list_isEmpty(l):
        devolver 0
    x = printInverse (NEXT(l))
    x = x + element_print (INFO(l)))  //info del 1º (first)
                                          //o del n-ésimo
    devolver x

// Código C. Versión recursiva. Pregunta: ¿Es O(N)?
// Ojo: En C sí hay que distinguir entre lista y nodo!
/*
Siempre que tengamos que implementar una función recursiva para una lista,
crearemos una función recursiva para los nodos, que será del tipo "hacer algo
con un nodo y también con los siguientes (next)", o bien "hacer algo con los
siguientes (next) y por último hacerlo con el nodo". Además, crearemos una
primera función no recursiva que compruebe los argumentos de entrada y que llame
a la función recursiva con pl->first (puntero al primer nodo, que no es el
"next" de otro nodo, sino el "first" de la lista - por eso no nos vale comenzar
directamente con la función recursiva, que hará referencia a "next").
*/

int list_printInverse (List *pl, FILE *pf) {
    if (pl == NULL)
        return -1;
    return printInverse_rec (pl->first, pf);
}
/* En la siguiente función recursiva, primero se llama a la misma función con pn->next
(para imprimir el resto de la lista primero) y después se imprime el nodo en el que
estamos */
int printInverse_rec (Node *pn, FILE *pf){
     if (pn==NULL)
        return 0;
     return printInverse_rec (pn->next, pf) + element_print (pn->info, pf);
}
Variación con la última sentencia separada en 2, por si se ve más claro:
int printInverse_rec (Node *pn, FILE *pf){
     int cont;
     if (pn==NULL)
        return 0;
     cont = printInverse_rec (pn->next, pf);
     return cont + element_print (pn->info, pf);
}
```

Esta forma de resolver el ejercicio, con la función recursiva, cada vez que se entra en la función se vuelve a llamar a la misma función y después se imprime el nodo. Si dibujamos las áreas de datos de las funciones, veremos que se trata de una secuencia de llamadas hasta llegar al next del último nodo, que es NULL. Cuando se llame a la función recursiva con NULL, se saldrá de la misma devolviendo 0 y ese retorno irá a la función desde la cual se llamó a esta (cuando pn apunta al último nodo), de modo que continuará, tras volver de printInverse_rec, con lo que le queda por hacer antes de salir, llamar a element_print, y, por tanto, se imprimirá el último nodo. Al salir de esa función, se volverá a la que llamó a esta con el último nodo; es decir, al área de datos de la llamada en la que pn apunta al penúltimo nodo; y le quedará imprimir ese penúltimo elemento. Y así sucesivamente, se va volviendo, imprimiendo todos los elementos hasta llegar al primero.
Si una lista tiene N elementos, el coste de esta función es O(N), pues se va a recorrer 1 vez la lista.

// Código C. Versión no recursiva 1. Sin control de errores.

// Importante : Analizar coste ¿Tiene coste O(N)? ¿Es $O(N^2)$?

/*

En este caso, como se trata de imprimir todos los elementos, se va a hacer uso de la función que calcula el tamaño de la lista y después, sabiendo ya cuántos elementos hay, se va ir extrayendo, imprimiendo e insertando de nuevo (para no perderlo) cada uno de los elementos de la lista:

*/

```
int list_printInverse (List *pl) {
    int cont=0, tam=0, i;
    Element *pe;

    if(!pl) return -1;
    tam = list_tam(pl);

    for (i=0; i<tam; i++) {
        pe = list_extractEnd (pl);
        cont += element_print(pe, pf);
        list_insertIni (pl, pe);
        element_free(pe);
    }
    return cont;
}
```

En este caso, si una lista tiene tamaño N, el coste de calcular su tamaño es O(N). Después, por cada elemento (N elementos), hay que extraerlo del final (coste O(N)), imprimir (O(1)) e insertarlo por el principio (O(1)). Por tanto, despreciando los costes O(1), el coste sería:

$O(N) + N*(O(N)) = O(N)+O(N^2) = O(N^2)$. Esta versión es más costosa que la versión recursiva.

```
// Código C. Versión no recursiva 2. Basada en el uso de una pila. Sin control de errores
// Importante analizar coste. ¿Es O(N)?
// ¿Tarda más que la versión recursiva?
// ¿Uso de pila tiene coste similar a llamadas recursivas?
// ¿Operaciones de copia y liberación de memoria vs. llamadas recursivas?
 #include “stack.h”
 int list_printInverse (List *pl) {
    Stack *ps = NULL;
    Element *pe = NULL;
    int cont = 0;

    p = stack_ini();

    while  (list_isEmpty(pl) == FALSE) {
        pe = list_extractIni (pl);
        stack_push (ps, pe); //Incluir CdE
        element_free (pe);
    }

    // imprime el contenido de la pila y deja pl como estaba
    while (stack_isEmpty(ps) == FALSE) {
        pe = stack_pop (ps);
        cont += element_print (pe);
        list_insertIni (pl, pe);
        element_free (pe);
    }

    stack_free(ps);

    return cont;
 }
/* En este caso, cada uno de los N elementos se extrae del inicio de la lista (O(1)) y se
inserta en pila (O(1)) para pasarlo todo a la pila. Esto es O(N). Después, los N
elementos se extraen uno a uno de la pila para imprimirlos (O(1)) y se insertan en la
lista por el principio (O(1)). Por tanto, esta parte es O(N).
O(N)+O(N) = O(N). Por tanto, en principio tarda un orden de magnitud similar a la versión
recursiva.
Sin embargo, en la versión recursiva no se realizan más operaciones que imprimir y llamar
a la función con next (hay una pila implícita, la de las áreas de datos de las funciones,
y en cada llamada se copia el argumento de entrada), mientras que aquí se extrae de
lista, se inserta en pila, hay nuevas reservas de memoria en cada insert, etc...
Por tanto, aunque el orden es el mismo en ambas versiones, la versión recursiva sería
mejor en este caso.
```

5. 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.

6. Supongamos que utilizamos la siguiente estructura de datos para implementar una lista:

```
typedef struct _ListStrange ListStrange;

struct _ListStrange {
    Node *first;  //guarda la dirección del primer nodo de la lista
    Node *last;   //guarda la dirección del último nodo de la lista
};
```

a) Dar el código C de las primitivas:

- ListStrange_ini
- ListStrange_isEmpty
- ListStrange_insertIni
- ListStrange_insertEnd
- ListStrange_extractIni
- ListStrange_extractEnd

```
ListStrange *ListStrange_ini () {
    ListStrange *pl;

    pl=(ListStrange *) malloc(sizeof(ListStrange));
    if (!pl) return NULL;

    pl->first = pl->last = NULL;
    return pl;
}
Bool *ListStrange_isEmpty () {
    if (!pl || !pl->first) return TRUE;
    return FALSE;
}
Status ListStrange_insertIni (ListStrange *pl,
                              Element *pe) {
  Node *pn;

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

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

  pn->info = element_copy(pe);
  if (pn->info==NULL) {
     node_free (pn);
     return ERROR;
  }
  //Si la lista está vacía
  if (pl->first==NULL) {
     pl->first = pl->last = pn;
     return OK;
  }
  pn->next = l-> first;
  pl->first = pn;
  return OK;
}
```

```
Status ListStrange_insertEnd (ListStrange *pl,
                              Element *pe) {
  Node *pn;

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

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

  pn->info = element_copy(pe);
  if (pn->info==NULL) {
     node_free (pn);
     return ERROR;
  }
  //Si la list está vacía
  if (pl->first==NULL) {
      pl->first = pl->last = pn;
      return OK;
  }

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

b) Discutir las ventajas de ListStrange respecto a la lista enlazada vista en clase si nuestro objetivo es implementar con ella una Queue.

ListStrange tiene un coste O(1) en la operación de inserción al final, porque se accede directamente utilizando last, mientras que en la lista enlazada simple vista en clase hay que buscar el último elemento recorriendo toda la lista, por lo que tiene complejidad O(N).

c) Escribe el fichero queue.h que utilizará la implementación anterior

```
// queue.h no cambia

#ifndef _QueueH_
#define _QueueH_
#include "element.h"

typedef struct _Queue Queue;

// prototipos de las primitivas
Queue *queue_ini();
void Queue_free(Queue *pq);

Status queue_insert (Queue *pc, Element *pe);
Element *queue_extract (Queue *pc);

Bool Queue_isEmpty(Queue *pq);
Bool Queue_isFull(Queue *pq);

#endif
```

d) Escribe el fichero queue.c que utilizará la implementación anterior

```
#include <stdlib.h>
#include <stdio.h>

#include "queue.h"
#include "ListStrange.h"

struct _Queue {
   ListStrange *pl;
};

Element *queue_extract (Queue *pc) {
   Element *pe;
   if (pc==NULL) return NULL;

   pe = ListStrange_extractIni (pc->pl);
   return pe;
}

Status queue_insert (Queue *pc, Element *pe) {
    if (pc==NULL) return ERROR;
    return ListStrange_insertEnd (pc->pl,  pe);
}
```

```
Queue *queue_ini () {
    Queue *pc;
    pc = (Queue *) malloc(sizeof(Queue));
    if (pc==NULL) return NULL;
    pc->pl = ListStrange_ini();
    if (pc->pl==NULL)  {
        free(pc);
        return NULL;
    }
    return pc;
}

void Queue_free (Queue *pc) {
    if (pc==NULL) return;
    ListStrange_free (pc->pl);
    free(pc);
}
```

7. Escribir una función C que reciba un puntero a una lista enlazada e intercambie sus elementos primero y último considerando las posibles situaciones de error. Hacer dos versiones: una que pueda acceder a la EdD de lista, y otra que no. Analizar costes.

```
// Versión sin acceso a EdD

Status swapListPU(List *pl) {
    Element *pe1=NULL, *pe2=NULL;
    if (pl == NULL)
        return ERROR;

    //Extrae 1 elemento del final. Si lista estaba vacía, no hay nada que intercambiar, queda tal cual.
    if ((pe1=list_extractEnd(pl))== NULL)
        return OK;

    // Si la primera extracción va bien, intenta extraer otro elemento, ahora del principio.
    // Si solo había 1 nodo, no podrá. En ese caso, no hay nada que intercambiar. Se vuelve a
    // insertar el extraído y ya está.
    if ((pe2=list_extractIni(pl))== NULL){
         list_insertEnd (pl, pe1);
        return OK;
    }
     // Si ambos se han extraído correctamente, insertar cada uno en la posición opuesta a la que
     // estaba.
    list_insertIni (pl, pe1);
    list_insertEnd (pl, pe2);

     // Como insertar hace una copia de los elementos, queda liberar pe1 y pe2.
    element_free(pe1);
    element_free(pe2);
    return OK;
}
```

```
// Versión con acceso a EdD

Status swapListPU(List *pl) {
    Node *pn, *ultimo;

    if (pl == NULL)
        return ERROR;

    // Lista vacía o con un único nodo
    if (pl->first ==NULL|| pl->first->next == NULL)
        return OK;

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

    // Intercambiar punteros: pn apunta al penúltimo
    ultimo = pn->next;
    pn->next = pl->first;
    ultimo->next = pn->next->next;
    pn->next->next = NULL;
    pl->first = ultimo;

    return OK;
}
```

```
// Versión con acceso a EdD, intercambiando
elementos en vez de nodos

Status swapListPU(List *pl) {
    Node *pn_ult;
    Element *pe;

    if (pl == NULL)
        return ERROR;

    // Lista vacía o con un único nodo
    if (pl->first ==NULL|| pl->first->next == NULL)
        return OK;

    // buscamos el último nodo
    for(pn=pl->first; pn->next!=NULL; pn=pn->next);

    // Intercambiar info de nodos primero y ultimo
    pe = pn->info;
    pn->info = pl->first->info;
    pl->first->info = pe;

    return OK;
}
```

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

```c
Status listInsPos(List *pl, int pos, const Element *pe) {
    Node *pn, *qn;
    int i;

    if(!pl || !pe || pos<0) return ERROR;

    // Creamos un nodo con la información
    pn = node_ini ();
    if (pn == NULL) return ERROR;


    pn->info = element_copy(pe);
    if (pn->info == NULL) {
        nodo_free(pn);
        return ERROR;
    }
    // Si hay que insertar al inicio:
    if (pos == 0) {
        pn->next = pl->first;
        pl->first = pn;
    }

    // Iteramos tantas veces como (pos - 1) hasta, como mucho, el último nodo.
    // La idea es que qn sea el nodo en posición pos-1
    for (i=0, qn=pl->first;  i<pos-1 && qn->next!=NULL;  i++, qn=qn->next);

    // Si sale del bucle anterior porque ha llegado al final de la lista, sin que la
    // posición de inserción sea la esperada, error
    if (i != pos-1) {
        nodo_free(pn); // esto también libera el elemento
        return ERROR;
    }

    // Si no, ha encontrado la posición -> Inserta el nodo pn como siguiente a qn
    pn->next = qn->next;
    qn->next = pn;

    return OK;
}
```

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

```
/* Inserta un dato en una lista ordenada de mayor a menor*/
Status list_insOrder (List *pl, const Element *pe) {
    Node *pn, *paux;

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

    // casos en los que hay que insertar al principio: lista vacía o el
    // elemento a insertar es mayor que el primero
    if ((list_isEmpty(pl)==TRUE)||element_comparar(pe, pl->first->info)>=0) {
        return list_insertIni (pl, pe);
    }
    // casos en los que no hay que insertar al principio; mientras no llegue
    // al final de la lista y pe sea menor que el info de cada nodo, avanza.
    paux = pl->first; //no puede ser NULL (sería el caso anterior)
    while(paux->next!=NULL && element_comparar(pe, paux->next->info)<0)
         paux = paux->next;

    // Si ha encontrado el lugar, o bien ha llegado al final de la lista, insertar.
    pn = node_ini();
    if (pn == NULL) return ERROR;
    pn->info = element_copy(pe);
    if (pn->info == NULL) {
        nodo_free(pn);
        return ERROR;
    }
    pn->next = paux->next;
    paux->next = pn;

    return OK;
}
```

10. Escribir el código C de una función Status list_concat(List *pl1, List *pl2) que enlace dos listas pl1 y pl2 de modo que la segunda quede enlazada directamente tras la primera. La lista pl2 quedará vacía.

```
Status list_concat(List *pl1, List *pl2) {
    Node *pn;

    if(!pl1 || !pl2) return ERROR;

    if(list_isEmpty(pl1) == TRUE) {
        pl1->first = pl2->first;
        pl2->first = NULL;
        return OK;
    }
    //Buscar el final de la primera lista. pn apuntará al último nodo

    pn = pl1->first; // no puede ser NULL (sería el caso anterior)
    while(pn->next != NULL)
         pn = pn->next;

    //Enlazar el último nodo de la 1ª lista con el primero de la 2ª
    pn->next = pl2->first;
    pl2->first = NULL;

    return OK;
}
```

11. Escribir el código C de una función List *list_concat2(const List *pl1, const List *pl2) que devuelva una nueva lista cuyos nodos son los de pl1 seguidos de los de pl2. Las listas pl1 y pl2 quedarán como estaban inicialmente. Esta función será parte de la implementación de la lista (puede acceder a la estructura de datos). Por simplicidad de la solución, no es necesario añadir control de errores en las funciones de creación de nodos y copia de elementos.

```
List *list_concat2(const List *pl1, const List *pl2) {
       List *pl_ret = NULL;
       Node *pn = NULL, *pn_ret = NULL;

       if(!pl1 || !pl2 || pl1->first==NULL || pl2->first==NULL) return ERROR;

       pl_ret = (List *) malloc(sizeof(List));  //Lista nueva, a devolver
       if(!pl_ret) return ERROR;

       pn = pl1->first;               //pn recorre listas originales, nodo a copiar
       pn_ret = node_ini();           //pn_ret es puntero a nuevos nodos. Añadir CdE
       pn_ret->info = element_copy(pn->info);   //Añadir CdE
       pl_ret->first = pn_ret;        //el first de la lista nueva es el nuevo nodo

       while(pn->next != NULL) {      //mientras no haya copiado todos los nodos
            pn = pn->next;            //pn avanza en la lista original
           pn_ret->next = node_ini(); //crea nuevo nodo seguido al último.CdE
           pn_ret->next->info = element_copy(pn->info); //Añadir CdE
            pn_ret = pn_ret->next;    //pn_ret avanza al último nodo de la nueva
       }

       pn = pl2->first;

       /* Lo mismo para la lista 2 */
       pn_ret = node_ini();           //pn_ret es puntero a nuevos nodos. Añadir CdE
       pn_ret->info = element_copy(pn->info);   //Añadir CdE
       pl_ret->first = pn_ret;        //el first de la lista nueva es el nuevo nodo

       while(pn->next != NULL) {       //mientras no haya copiado todos los nodos
            pn = pn->next;             //pn avanza en la lista original
           pn_ret->next = node_ini();  //crea nuevo nodo seguido al último.CdE
           pn_ret->next->info = element_copy(pn->info); //Añadir CdE
           pn_ret = pn_ret->next;      //pn_ret avanza al último nodo de la nueva
       }

       return pl_ret;
}
```

12. Escribir un algoritmo, `List list_combineOrder (const List *pla, const List *plb)`, 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. Analizar la eficiencia del código. Suponer para ello la siguiente definición de Lista:

```
// En list.h
typedef struct _List List;

// En list.c
struct _Node {
      Element *info;
      struct _Node *next;
};
typedef struct _Node Nodo;

struct _List {
      Node *first;
};
```

No hace falta considerar el control de errores.

Ayuda: *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.*

```
// PSEUDOCÓDIGO (1)

List list_combineOrder (List a, List b)
    pa = a
    pb = b

    laux = list_ini ()
    mientras pa ≠ NULL AND pb ≠ NULL:
        // hallo el mínimo y avanzo el puntero
        si INFO (pa) ≤ INFO (pb):
            min = INFO (pa)
            pa = NEXT (pa)
        caso contrario:
            min = INFO (pb)
            pb = NEXT (pb)

        // inserto en la lista de salida el mín
        list_insertEnd (laux, min)

    // hallo cual de las dos listas ha finalizado
    si list_isEmpty (pa) = TRUE:
        pb = pa

    // inserto en la lista de salida los elementos
    // de la lista que queden
    mientras pb ≠ NULL:
        list_insertEnd (laux, INFO(pb))
        pb = NEXT (pb)

    dev laux
```

```
// PSEUDOCÓDIGO (2)

List list_combineOrder (List a, List b)
    pa = a
    pb = b

    laux = list_ini ()
    mientras pa ≠ NULL OR pb ≠ NULL:
        si pa ≠ NULL AND pb ≠ NULL:
            si INFO (pa) ≤ INFO (pb):
                list_insertEnd (laux, INFO(pa))
                pa = NEXT (pa)
            else:
                list_insertEnd (laux, INFO(pb))
                pb = NEXT (pb)
        else:
            si pa ≠ NULL:
                list_insertEnd (laux, INFO(pa))
                pa = NEXT (pa)
            else:
                si pb ≠ NULL:
                    list_insertEnd(laux, INFO(pb))
                    pb = NEXT (pb)

     dev laux
```

```
// CÓDIGO C. No eficiente O(N²)

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

List *list_combineOrder (const List
*pl_a, const List *pl_b) {
    Node *pa, *pb;
    Element *pmin;
    List *pl;

    // crea lista de salida
    pl = list_ini();
    if (pl == NULL) return NULL;
    pa = pl_a -> first;
    pb = pl_b -> first;

    while  (pa!=NULL && pb!=NULL) {
        // Obtengo mínimo y avanzo
        if (element_cmp(INFO(pa),
                        INFO(pb)) < 0) {
            pmin = INFO (pa);
            pa = NEXT (pa);
        }
        else {
            min = INFO (pb);
            pb = NEXT (pb);
        }
        // inserto al final de la lista de
        // salida el mínimo
        list_insertEnd (pl, pmin);
    }

    // Chequeo qué lista no ha finalizado
    if (pa != NULL) pb = pa;

    // vuelco en la lista de salida los
    // elementos de la lista de entrada
    // que no haya acabado de recorrer
    while (pb != NULL)  {
        list_insertEnd (pl, INFO(pb));
        pb = NEXT (pb);
    }
    return pl;
}
```

```
// CÓDIGO C. Eficiente O(N).
/* Para evitar recorrer la lista cada vez que
tengo que insertar en ella, guardo la
dirección del puntero al último nodo
insertado, en un puntero a puntero (paux) */
#define INFO(a) (a)->info
#define NEXT(a) (a)->next

List *list_combineOrder (const List *pl_a,
                         const List *pl_b) {
    Node *pa, *pb, *pn, **paux;
    Element *pmin;
    List *pl;

    // crea la lista de salida
    pl = list_ini();
    if (pl == NULL) return NULL;
    pa = pl_a -> first;
    pb = pl_b -> first;
    paux = &(pl->first);

    while  (pa != NULL && pb != NULL) {
        // hallo el mínimo y avanzo
       if (element_cmp(INFO(pa),INFO(pb))<0){
            pmin = INFO (pa);
            pa = NEXT (pa);
       }
       else {
            pmin = INFO (pb);
            pb = NEXT (pb);
       }
       // inserto al final de la lista de
       // salida el mínimo
        pn = node_ini();
        pn->info = element_copy(pmin);
        *paux = pn;
        paux = &( (*paux) -> next);
    }
    // Chequeo qué lista no ha finalizado
    if (pa != NULL) pb = pa;

    // vuelco en la lista de salida los
    // elementos de la lista de entrada que
    // no haya acabado de recorrer
    while (pb != NULL)  {
        pn = node_ini();
        pn->info = element_copy(INFO(pb));
        *paux = pn;
        paux = &( (*paux) -> next);
        pb = NEXT (pb);
    }
    return pl;
}
```

13. 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 *queuep.h* con la interfaz del TAD.

(b) Proporcione el fichero *queuep.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).

```
// queuep.h
#ifndef _QueueP_H_
#define _QueueP_H_

typedef struct _Queuep Queuep;

Queuep *queuep_ini();
void queuep_free (Queuep *pc);

Status queuep_insert (Queuep *pc, Element *pe);
Element *queuep_extract (Queuep *pc);
Bool queuep_isEmpty (Queuep *pq);
Bool queuep_isFull (Queuep *pq);
#endif

// queuep.c

#include "list.h"
#define MAXPRIORITY 100

/*1 lista/nivel de prioridad. Los elementos de
prioridad 0 se insertarán en la lista 0, los de
1 en la lista 1, etc. La lista de cada nivel de
prioridad siempre existe, aunque esté vacía*/

struct _Queuep {
    List *lprior[MAXPRIORITY];
}

Queuep *queuep_ini() {
    int i, j;
    int flag=0;
    Queuep *pc;

    pc = (Queuep*) malloc(sizeof(Queuep));
    if (pc==NULL) return NULL;
    for (i=0; i< MAXPRIORITY && flag==0; i++)
       if ((pc->lprior[i]=list_ini()) == NULL)
            flag=1;
    // libera si ha habido error
    if (flag==1) {
        for(j=0; j<i-1; j++)
            list_free(pc->lprior[j]);
        free(pc);
        return NULL;
    }
    return pc;
}
```

```
//Insertar el elemento según su prioridad
Status queuep_insert (Queuep *pc, const Element
*pe) {
    int prior;
    if (!pc || !pe) return ERROR;
    prior = element_getPriority(pe);
    if (isPrioOK(prior) == FALSE)
          return ERROR;
    return (list_insertEnd (pc->lprior[prior],
pe));
}

/* Extraer 1 elemento de Queue de prioridad =
extraer 1 elemento de la lista de mayor
prioridad en la que haya elementos (no vacía) */

Element *queuep_extract (Queuep *pc) {
    int i;
    Element *pe =NULL;
    Boolean extr = FALSE;
    if (!pc) return ERROR;
    for (i=0; i<MAXPRIORITY && extr==FALSE;i++){
        if (list_isEmpty(pc->lprior[i])==FALSE){
           pe = list_extractIni(pc->lprior[i]);
           extr = TRUE;
        }
    }
    return pe; //NULL si no había elementos
}

void queuep_free (Queuep *pc) {
    int i;
    if (!pc) return;
    for (i=0; i<MAXPRIORITY; i++)
        list_free (pc->lprior[i]);
    free(pc);
    return;
}
```

Listas enlazadas circulares

14. Implemente las primitivas del TAD Queue usando como EdD una lista enlazada circular. Proporcione los ficheros queue.h y queue.c. Discuta las ventajas de utilizar una lista enlazada circular frente a otras estructuras de datos (por ejemplo, una lista enlazada de nodos y un array).

```
#include "listCircular.h"

struct _Queue {
   ListCircular *pl;
}

Element *queue_extract (Queue *pc) {
   Element *pe;
   if (pc==NULL) return NULL;

   pe = listCircular_extractIni (pc->pl);
   return pe;
}

Status queue_insert (Queue *pc, Element *pe) {
    if (pc==NULL) return ERROR;
    return listCircular_insertEnd (pc->pl, pe);
}

BOOLEAN Queue_isEmpty (const Queue *pc) {
    return FALSE;
}
```

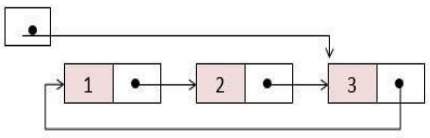
```
Queue *queue_ini () {
    Queue *pc;
    pc = (Queue *) malloc(sizeof(Queue));
    if (pc==NULL) return NULL;
    pc->pl = listCircular_ini();
    if (pc->pl==NULL)  {
        free(pc);
        return NULL;
    }
    return pc;
}

void Queue_free (Queue *pc) {
    if (pc==NULL) return;

    listCircular_free (pc->pl);
    free(pc);
}
```

Explicación ventajas de lista circular frente a lista enlazada simple o array: ver diapositivas de clase.

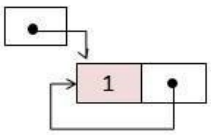
15. Consideremos el **TAD Lista Enlazada Circular (LEC).**

a) Dibuja un esquema de una LEC que tenga:
   1. 3 nodos
   2. 1 nodo
   3. 0 nodos

3 nodos:



1 nodo:



0 nodos:

b) Implementa en C la primitiva:

```
Boolean listCircular_ordered (ListCircular *pl);
```

que chequea si los elementos de la lista están ordenados de menor a mayor.

Puedes usar la siguiente primitiva del TAD elemento:

```
Boolean element_lessEqual (const Element *pe1, const Element *pe2);
```

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

```
Primitiva = puede acceder a la estructura de datos.
Boolean listCircular_ordered (const ListCircular *pl) {
   Node *pn;
   if (!pl ) return FALSE;
   if (!last(pl) || (last(pl)==next(last(pl)))
         return TRUE; // lista con 0 o 1 elementos

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

Listas doblemente enlazadas

16. Dar el código C de la función `Status listDE_insertIni(ListDE *pl, Element *pe)` que inserta un elemento al inicio de una lista doblemente enlazada.

```
/* En las listas doblemente enlazadas los nodos tienen puntero a next y puntero a prev.
Si se inserta al inicio, el next del nuevo nodo será el que antes era primer elemento de
la lista, y el prev de ese primer elemento será el nuevo nodo. */

Status listDE_insertIni(ListDE *pl, const Element *pe) {
       Node *pn = NULL;
       if(!pl || !pe) return ERROR;

       pn = node_ini();
       if (!pn) return ERROR;

       pn->info = element_copy(pe);
       if (pn->info==NULL) {
          node_free (pn);
          return ERROR;
       }

       if (pl->first != NULL){  //Si la lista no estaba vacía:
           pn->next = pl->first;   //el nuevo nodo apunta al que antes era el primero
           pl->first->prev = pn;   // el previo del que antes era el primero es el nuevo
       }
        pl->first = pn;         //En cualquier caso, el primero ahora es el nuevo nodo
       return OK;
}
```

17. Dar el código C de la función `Status listDE_insertEnd(ListDE *pl, const Element *pe)` que inserta un elemento al final de una lista doblemente enlazada.

```
Status listDE_insertEnd(ListDE *pl, const Element *pe) {
	Node *pn = NULL, *pq = NULL;
	if(!pl || !pe) return ERROR;

	pn = node_ini();
	if (!pn) return ERROR;

	pn->info = element_copy(pe);
	if (pn->info==NULL){
	   node_free (pn);
	   return ERROR;
	}
	// Si lista vacía
	if (pl->first == NULL){
	   pl->first = pn;
	   return OK
	}
	// Si la lista no está vacía, buscar el último nodo; pq queda apuntando al último.
	pq = pl->first;
	while(pq->next != NULL) {
		pq = pq->next;
	}
	pq->next = pn;    //El next del que antes era el último será el nuevo nodo
	pn->prev = pq;    //El prev del nuevo nodo será el que antes era último
	return OK;
}
```

18. Dar el código C de la función `Element *listDE_extractIni(ListDE *pl)` que extrae un elemento del inicio de una lista doblemente enlazada.

```
Element *listDE_extractIni(ListDE *pl) {
	Node *pn = NULL;
	Element *pe = NULL;

	if(!pl || pl->first == NULL)
	    return NULL;

	pn = pl->first;     //pn apunta al primer nodo, de donde se quiere extraer el elemento
	pl->first = pn->next;  //ya podemos hacer que el puntero a lista apunte al siguiente
	pl->first->prev = NULL; //el previo al que ahora queda primero será NULL
	pe = pn->info;          //se guarda en pe el puntero a la info de pn, para devolverlo
	free (pn);              //se libera el nodo (ojo: no es node_free, no libera la info)
	return pe;
}
```

19. Dar el código C de la función `Element *listDE_extractEnd(ListDE *pl)` que extrae un elemento del final de una lista doblemente enlazada.

```
Element *listDE_extractEnd(ListDE *pl) {
	Node *pn = NULL;
	if(!pl || pl->first == NULL)
		return NULL;

	//se busca el último nodo; pn apuntará al mismo
	pn = pl->first;
	while(pn->next != NULL) {
		pn = pn->next;
	}
	pn->prev->next = NULL;  //el next del penúltimo apuntará a NULL
	pe = pn->info;          //se guarda en pe el puntero a la info de pn, para devolverlo
	free (pn);              //se libera el nodo (ojo: no es node_free, no libera la info)
	return pe;
}
```

20. 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.

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

```
Status listIntPos (List *pl, int pos) {

    Node *pn, *qn;
    int i;

    // lista vacía o con un solo elemento
    if (pl->first==NULL  || pl->first->next == NULL) return ERROR;

    // Iteramos tantas veces como (pos-1) hasta el penúltimo nodo
    pn=pl->first;
    for (i=0; i<pos && pn->next->next != NULL; i++)
            pn = pn->next;
    if (i != pos) //posición no encontrada
            return ERROR;

    // Intercambiar pn con el siguiente
    qn = pn->next;          //Ponemos puntero apuntando al siguiente a pn
    pn->next = qn->next;    //Ya podemos variar el siguiente a pn
    qn->ant = pn->ant;      // Vamos actualizando los punteros con cuidado de
    pn->ant->next = qn;     // no perder ninguna referencia y de dejar todo
    pn->ant = qn;           // como corresponde (hacer el dibujo).
    qn->next = pn;

    return OK;
}
```