Programación II

Tema 4. Listas enlazadas

Iván Cantador
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contenidos

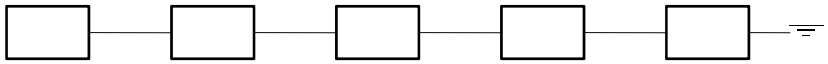
- El TAD Lista
- Estructura de datos de Lista
- Implementación en C de Lista
- Implementación de Pila y Cola con Lista
- Tipos de Listas

Contenidos

- **El TAD Lista**
- Estructura de datos de Lista
- Implementación en C de Lista
- Implementación de Pila y Cola con Lista
- Tipos de Listas

El TAD Lista. Definición

- **Lista**. Colección de objetos donde:
  - todos menos uno tienen un objeto “siguiente”
  - todos menos uno tienen un objeto “anterior”



- Permite la representación secuencial y ordenada de objetos de cualquier tipo
  - Insertando o extrayendo objetos al principio/final
  - Insertando o extrayendo objetos en cualquier punto
- Puede verse como una meta-EdD más que como un TAD
  - Puede usarse para implementar pilas, colas, colas de prioridad, etc.

El TAD Lista. Funciones primitivas

- **Funciones primitivas básicas**

```
Lista lista_crear()
void lista_liberar(Lista l)
boolean lista_vacia(Lista l)       // ¡Ojo! No existe lista_llena
status lista_insertarIni(Lista l, Elemento e) // Inserta al inicio
Elemento lista_extraerIni(Lista l)            // Extrae del inicio
status lista_insertarFin(Lista l, Elemento e) // Inserta al final
Elemento lista_extraerFin(Lista l)            // Extrae del final
```

- **… y otras**

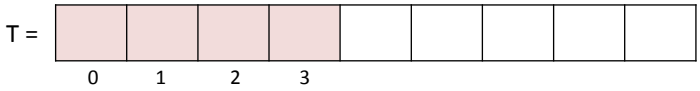
```
// Inserta el elemento e en la posicion pos de la lista L
status lista_insertarPos(Lista l, Elemento e, int pos)
// Inserta el elemento e en la lista L en orden
status lista_insertarOrden(Lista l, Elemento e)
...
```


Contenidos

- El TAD Lista
- **Estructura de datos de Lista**
- Implementación en C de Lista
- Implementación de Pila y Cola con Lista
- Tipos de Listas

EdD de Lista: array

- Opción 1: **tabla/array de elementos**

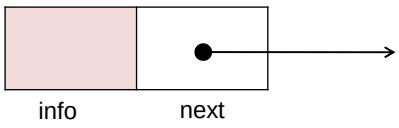


siguiente(T[i]) ≡ T[i + 1]; anterior(T[i])≡ T[i – 1]

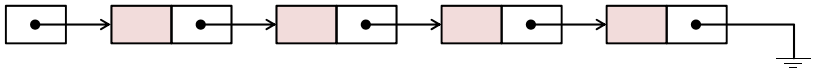
- *Ventajas*
  - Fácil implementación
  - Memoria estática
- *Inconvenientes*
  - Desperdicio de espacio
  - Ineficiencia al insertar al inicio y en posiciones intermedias: hay que mover todos los elementos a la derecha una posición (posible solución: lista circular? ocurre lo mismo?)

EdD de Lista: lista enlazada (LE)

- Opción 2: **Lista Enlazada (LE)**
  - Listas de **nodos**
  - Nodo
    - Campo **info**: contiene el objeto/dato a guardar
    - Campo **next**: apunta al siguiente nodo de la lista

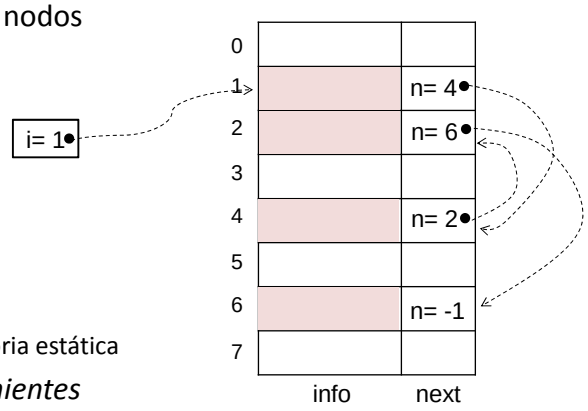


  - **Lista enlazada**: colección de nodos enlazados + puntero al nodo inicial. El next del último nodo apunta a NULL.



EdD de Lista: LE estática

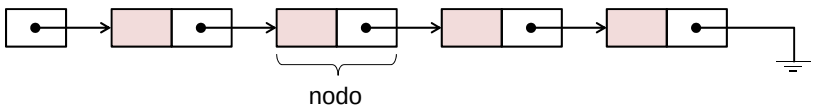
- **Estructura estática para LE**
  - Tabla de nodos



  - *Ventajas*
    - Memoria estática
  - *Inconvenientes*
    - Desperdicio de memoria
    - Complejidad (p.e. ¿siguiente nodo libre?)

EdD de Lista: LE dinámica

- **Los nodos se crean/destruyen dinámicamente**
  - Uso de memoria dinámica
  - Creación de nodos ≡ malloc
  - Liberación de nodos ≡ free

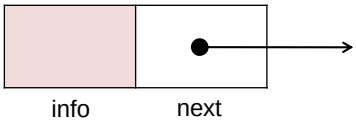


- *Ventajas*
  - Sólo se tiene reservada la memoria que se necesita en cada momento
  - Se pueden albergar tantos elementos como la memoria disponible permita
  - Insertar/extraer nodos no requiere desplazamientos de memoria
- *Inconvenientes*
  - Bueno para acceso secuencial; malo para acceso aleatorio

EdD de Lista en C

- **EdD de Nodo**
  - Se oculta al usuario definiéndola y poniendo la implementación de sus funciones asociadas en lista.c

```
// En lista.c (antes de la definición de Lista)
struct _Nodo {
   Elemento *info;
   struct _Nodo *next;
};

typedef struct _Nodo Nodo;
```



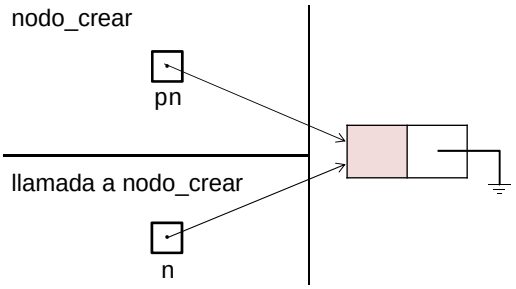
EdD de Lista en C

- **Creación de un nodo**

```
Nodo *nodo_crear() {
   Nodo *pn = NULL;
   pn = (Nodo *) malloc(sizeof(Nodo));
   if (!pn) return NULL;
   pn->info = NULL;   // Habrá que apuntar info a un elemento
   pn->next = NULL;
   return pn;
}
```

- Ejemplo de llamada

```
Nodo *n = NULL;
n = nodo_crear();
if (!n) {
   // CdE
}
```



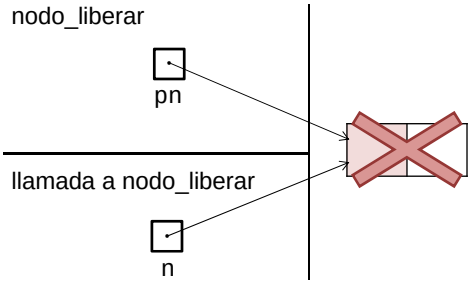
EdD de Lista en C

- **Liberación de un nodo**

```
void nodo_liberar(Nodo *pn) {
   if (pn) {
      elemento_liberar(pn->info); // Libera elemento de info
      free(pn);                   // Libera nodo
   }
}
```

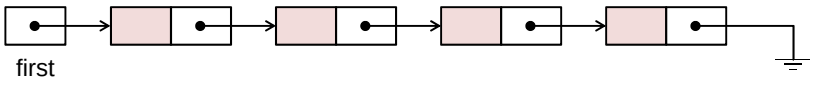
- Ejemplo de llamada

```
Nodo *n = NULL;
n = nodo_crear();
if (!n) {
   // CdE
}
...
nodo_liberar(n);
```



EdD de Lista en C

- **Lista enlazada**
  - Colección de nodos enlazados
  - La lista es un puntero (*first*) al nodo inicial
  - El último nodo apunta a NULL



- Tipo de dato Lista ≡ Puntero al nodo inicial

```
// En lista.h
typedef struct _Lista Lista;

// En lista.c
struct _Lista {
   Nodo *first;
};
```

Contenidos

- El TAD Lista
- Estructura de datos de Lista
- **Implementación en C de Lista**
- Implementación de Pila y Cola con Lista
- Tipos de Listas

Implementación en C: primitivas


```
// Primitivas de Nodo
Nodo *nodo_crear()      // luego habrá que apuntar info a un elemento
void nodo_liberar(Nodo *pn) // llama a elemento_liberar sobre info

// Primitivas de Lista (Nodo está oculto para el usuario)
Lista *lista_crear()
void lista_liberar(Lista *pl)
boolean lista_vacia(Lista *pl)
status lista_insertarIni(Lista *pl, Elemento *pe)
Elemento *lista_extraerIni(Lista *pl)
status lista_insertarFin(Lista *pl, Elemento *pe)
Elemento *lista_extraerFin(Lista *pl)
```

Importante: para mayor legibilidad, en algunas de las implementaciones que siguen NO se realizan ciertos controles de argumentos de entrada y de errores → ¡habría que hacerlos!

Implementación en C: lista_crear

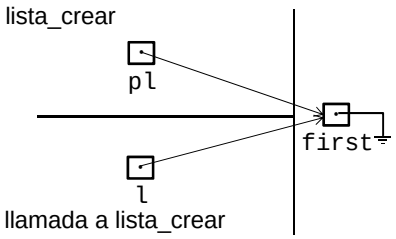

```
struct _Lista {
   Nodo *first;
};
```

- **Crear una lista**

```
Lista *lista_crear() {
   Lista *pl = NULL;
   pl = (Lista *) malloc(sizeof(Lista));
   if (!pl) return NULL;
   pl->first = NULL;
   return pl;
}
```

- Ejemplo de llamada

```
Lista *l = NULL;
l = lista_crear();
```



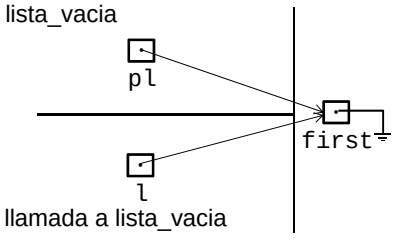
Implementación en C: lista_vacia

- **Comprobar si una lista está vacía**

```
boolean lista_vacia(Lista *pl) {
   if (!pl) return TRUE;            // Caso de error
   if (!pl->first) return TRUE;     // Caso de lista vacía
   return FALSE;                    // Caso de lista no vacía
}
```

- Ejemplo de llamada

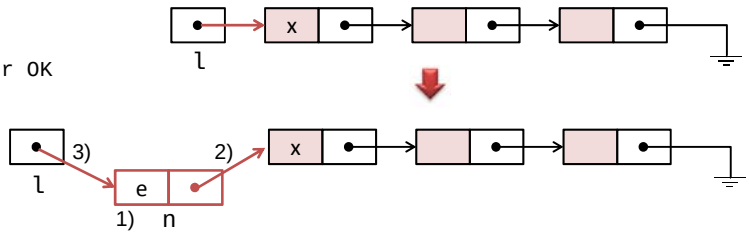
```
Lista *l = NULL;
l = lista_crear();
...
if (lista_vacia(l) == FALSE) {
   ...
}
```



Implementación en C: lista_insertarIni

- **Insertar un elemento al inicio de una lista**
  1) Crear un nuevo nodo
  2) Hacer que este nodo apunte al inicio de la lista
  3) El nuevo nodo es ahora el inicio de la lista
- Pseudocódigo

```
status lista_insertarIni(Lista l, Elemento e) {
   Nodo n = nodo_crear()
   info(n) = e
   next(n) = l
   l = n

   devolver OK
}
```



Implementación en C: lista_extraerIni

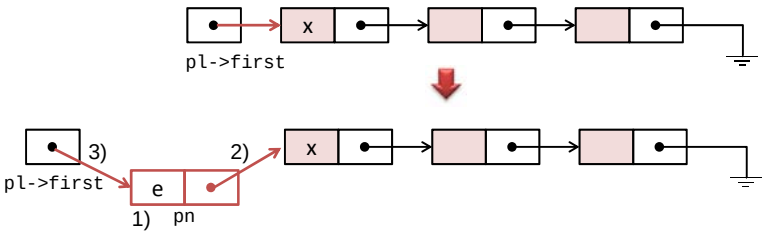
- Implementar la función **lista_insertarIni**

```
status lista_insertarIni(Lista *pl, Elemento *pe)
```

Implementación en C: lista_insertarIni

- Implementación

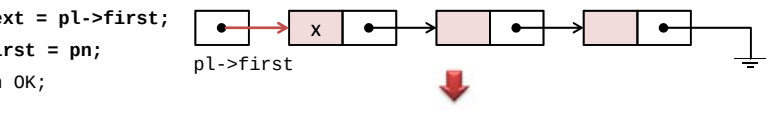
```
status lista_insertarIni(Lista *pl, Elemento *pe) {

}
```

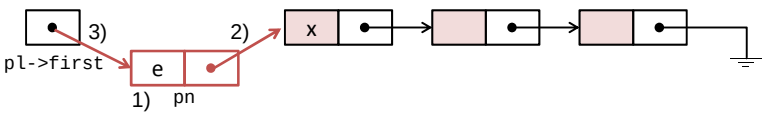


Implementación en C: lista_insertarIni

- Implementación

```
status lista_insertarIni(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }

    pn->info = elemento_copiar(pe);
    if (!pn->info) {
        nodo_liberar(pn);
        return ERROR;
    }

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



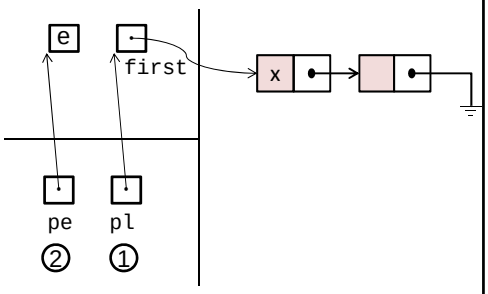


Implementación en C: lista_insertarIni

- Implementación

```
status lista_insertarIni(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }

    pn->info = elemento_copiar(pe);
    if (!pn->info) {
        nodo_liberar(pn);
        return ERROR;
    }

    pn->next = pl->first;

    pl->first = pn;

    return OK;
}
```

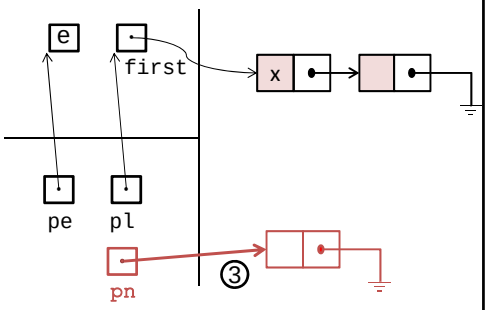
① ②



Implementación en C: lista_insertarIni

- Implementación

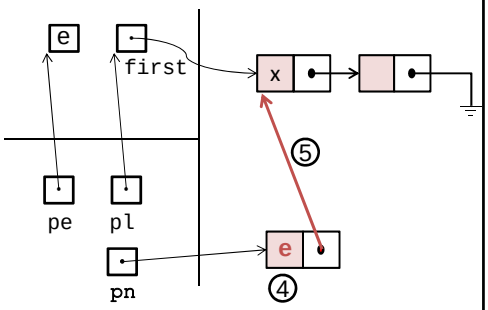
```
status lista_insertarIni(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL;

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

③   pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }

    pn->info = elemento_copiar(pe);
    if (!pn->info) {
        nodo_liberar(pn);
        return ERROR;
    }

    pn->next = pl->first;

    pl->first = pn;

    return OK;
}
```



Implementación en C: lista_insertarIni

- Implementación

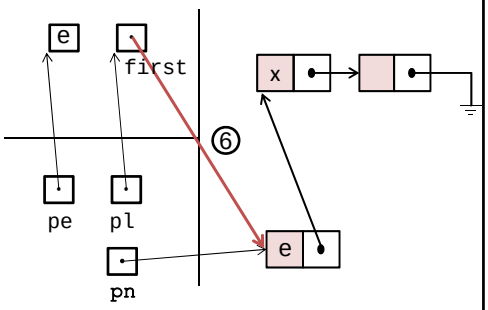
```
status lista_insertarIni(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
④   pn->info = elemento_copiar(pe);
    if (!pn->info) {
        nodo_liberar(pn);
        return ERROR;
    }

⑤   pn->next = pl->first;
    pl->first = pn;

    return OK;
}
```



Implementación en C: lista_insertarIni

- Implementación

```
status lista_insertarIni(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
    pn->info = elemento_copiar(pe);
    if (!pn->info) {
        nodo_liberar(pn);
        return ERROR;
    }

    pn->next = pl->first;
⑥   pl->first = pn;

    return OK;
}
```



Implementación en C: lista_insertarIni

- Alternativa implementación: uso de macros

```
#define next(pnodo) (pnodo)->next
#define info(pnodo) (pnodo)->info
#define first(plista) (plista)->first

status lista_insertarIni(Lista *pl, Elemento *pe) {
    ...
    pn = nodo_crear();
    ...
④ info(pn) = elemento_copiar(pe);
⑤ next(pn) = first(pl);
⑥ first(pl) = pn;
   ...
}
```

- *Ventajas*
  - Más parecido al pseudocódigo; más fácil de entender/manejar
  - Se puede hacer (más o menos) independiente de la EdD

Implementación en C: lista_insertarIni

- Ejemplo de uso de macros

```
#define info(A) (A)->info
```

- ¿Es lo mismo que `#define info(A) A->info`? ¡No!
  - Si en el código encontramos `info(abc)` ➔ `abc->info` `// OK`
  - Si encontramos `info(*ppn)` ➔ `*ppn->info` `// Problema: '->' se aplica antes`
  - **Solución**: definir la macro como sigue:

```
#define info(A) (A)->info
```

  - Ahora: `info(*ppn)` ➔ `(*ppn)->info` `// OK`

  - Importante: no olvidarse de los paréntesis:

```
#define cuadrado(x) x*x   ➔ ¿correcto?
cuadrado(z+1) ➔ z + 1 * z + 1
#define cuadrado (x) (x)*(x) ➔ (z+1)*(z+1)
```

  - ¡Ojo! `cuadrado(z++)` ➔ `(z++)*(z++)` `// 2 incrementos!`

Implementación en C: lista_extraerIni

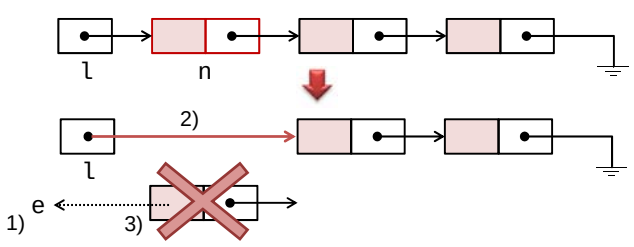
- Implementar la función **lista_extraerIni**

```
Elemento *lista_extraerIni(Lista *pl)
```

Implementación en C: lista_extraerIni

- **Extraer un elemento del inicio de una lista**
  1) Devolver el campo info del primer nodo
  2) Hacer que la lista apunte al siguiente nodo
  3) Eliminar el primer nodo
- Pseudocódigo

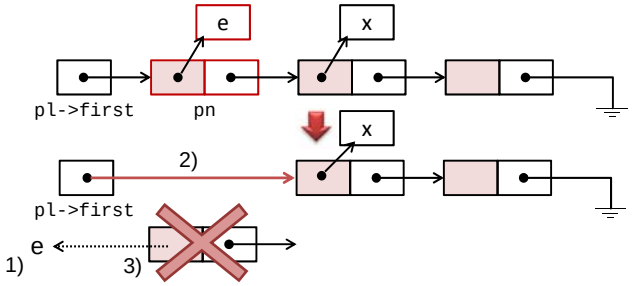
```
Elemento lista_extraerIni(Lista l) {
   if (lista_vacia(l)) devolver NULL

   n = l
   e = info(n)
   l = next(n)

   nodo_liberar(n)

   devolver e
}
```



Implementación en C: lista_extraerIni

- Implementación

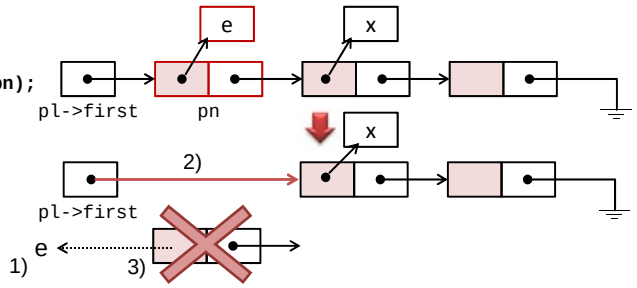
```
Elemento *lista_extraerIni(Lista *pl) {

}
```



Implementación en C: lista_extraerIni

- Implementación

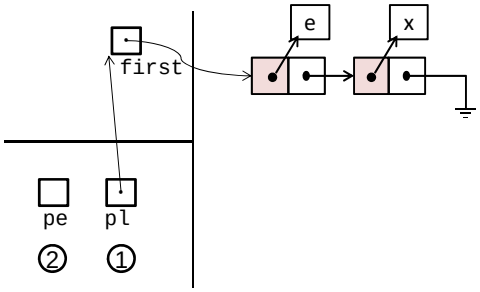
```
Elemento *lista_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl) == TRUE) {
        return ERROR;
    }

    pn = first(pl);
    pe = elemento_copiar(info(pn));
    if (!pe) {
        return NULL;
    }

    first(pl) = next(pn);

    nodo_liberar(pn);

    return pe;
}
```



Implementación en C: lista_extraerIni

- Implementación

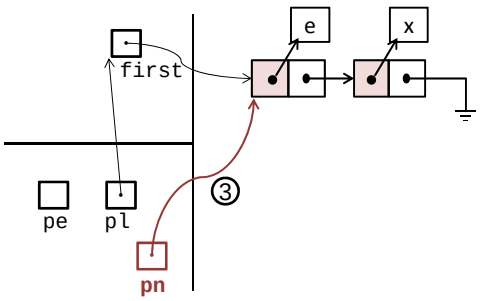
```
Elemento *lista_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl) == TRUE) {
        return ERROR;
    }

    pn = first(pl);
    pe = elemento_copiar(info(pn));
    if (!pe) {
        return NULL;
    }

    first(pl) = next(pn);

    nodo_liberar(pn);

    return pe;
}
```



Implementación en C: lista_extraerIni

- Implementación

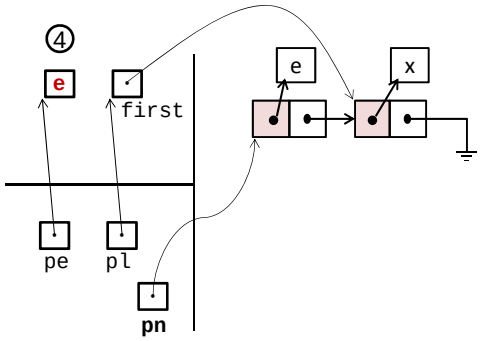
```
Elemento *lista_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl) == TRUE) {
        return ERROR;
    }

    pn = first(pl);
    pe = elemento_copiar(info(pn));
    if (!pe) {
        return NULL;
    }

    first(pl) = next(pn);

    nodo_liberar(pn);

    return pe;
}
```



Implementación en C: lista_extraerIni

- Implementación

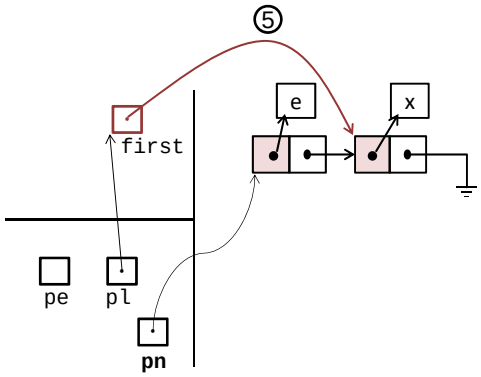
```
Elemento *lista_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl) == TRUE) {
        return ERROR;
    }

    pn = first(pl);
④   pe = elemento_copiar(info(pn));
    if (!pe) {
        return NULL;
    }

    first(pl) = next(pn);

    nodo_liberar(pn);

    return pe;
}
```



Implementación en C: lista_extraerIni

- Implementación

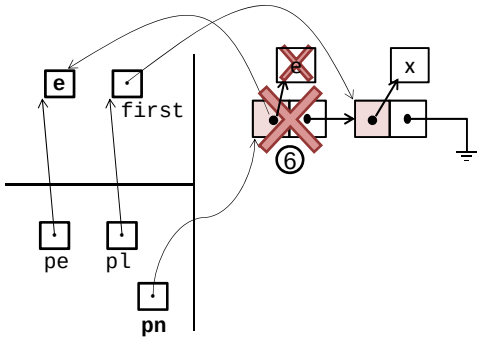
```
Elemento *lista_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl) == TRUE) {
        return ERROR;
    }

    pn = first(pl);
    pe = elemento_copiar(info(pn));
    if (!pe) {
        return NULL;
    }

⑤   first(pl) = next(pn);

    nodo_liberar(pn);

    return pe;
}
```



Implementación en C: lista_extraerIni

- Implementación

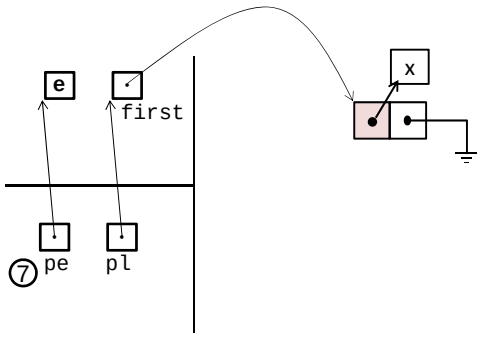
```
Elemento *lista_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl) == TRUE) {
        return ERROR;
    }

    pn = first(pl);
    pe = elemento_copiar(info(pn));
    if (!pe) {
        return NULL;
    }

    first(pl) = next(pn);

⑥  nodo_liberar(pn);

    return pe;
}
```



Implementación en C: lista_extraerIni

- Implementación

```
Elemento *lista_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl) == TRUE) {
        return ERROR;
    }

    pn = first(pl);
    pe = elemento_copiar(info(pn));
    if (!pe) {
        return NULL;
    }

    first(pl) = next(pn);

    nodo_liberar(pn);

⑦  return pe;
}
```



Implementación en C: lista_extraerIni

- Implementación (sin macros)

```
Elemento *lista_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl) == TRUE) {
        return ERROR;
    }

    pn = pl->first;
    pe = elemento_copiar(pn->info);
    if (!pe) {
        return NULL;
    }

    pl->first = pn->next;

    nodo_liberar(pn);

    return pe;
}
```

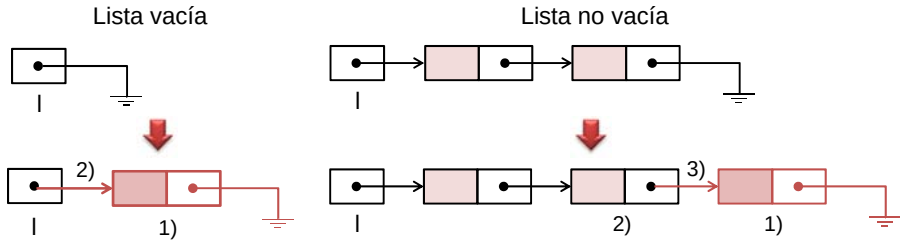

Implementación en C: lista_insertarFin

- Implementar la función **lista_insertarFin**

```
status lista_insertarFin(Lista *pl, Elemento *pe)
```

Implementación en C: lista_insertarFin

- **Insertar un elemento al final de una lista**
  1) Crear un nuevo nodo y asignar campo info
  - Si lista vacía:
    2) Asignar el nuevo nodo como primer nodo de la lista
  - Si lista no vacía:
    2) Recorrer la lista hasta situarse en el último nodo
    3) Hacer que el último nodo apunte al nuevo nodo



Implementación en C: lista_insertarFin

- Implementación

```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
    info(pn) = elemento_copiar(pe);
    if (!info(pn)) {
        nodo_liberar(pn);
        return ERROR;
    }

    if (lista_vacia(pl) == TRUE) {
        first(pl) = pn;
        return OK;
    }
    for (qn=first(pl); next(qn)!=NULL; qn=next(qn))
        ;
    next(qn) = pn;
    return OK;
}
```

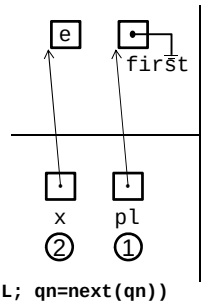
```
// Con bucle while
qn = first(pl);
while (next(qn)!=NULL) {
    qn = next(qn);
}
```

2 casos
1) Lista vacía
2) Lista no vacía

Implementación en C: lista_insertarFin

- Implementación (lista vacía) ①②

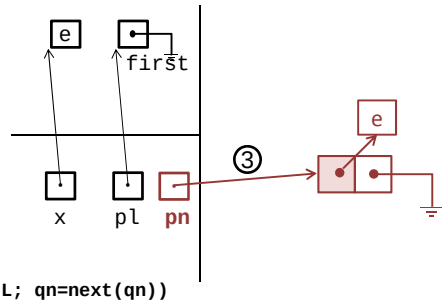
```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
    info(pn) = elemento_copiar(pe);
    if (!info(pn)) {
        nodo_liberar(pn);
        return ERROR;
    }

    if (lista_vacia(pl) == TRUE) {
        first(pl) = pn;
        return OK;
    }
    for (qn=first(pl); next(qn)!=NULL; qn=next(qn))
        ;
    next(qn) = pn;
    return OK;
}
```



Implementación en C: lista_insertarFin

- Implementación (lista vacía)

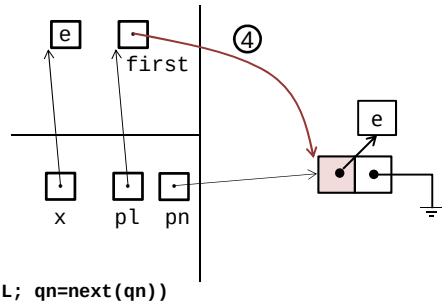
```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
③   info(pn) = elemento_copiar(pe);
    if (!info(pn)) {
        nodo_liberar(pn);
        return ERROR;
    }

    if (lista_vacia(pl) == TRUE) {
        first(pl) = pn;
        return OK;
    }
    for (qn=first(pl); next(qn)!=NULL; qn=next(qn))
        ;
    next(qn) = pn;
    return OK;
}
```



Implementación en C: lista_insertarFin

- Implementación (lista vacía)

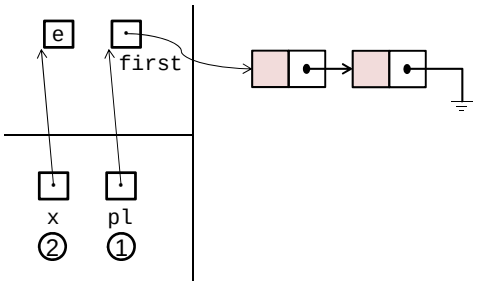
```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
    info(pn) = elemento_copiar(pe);
    if (!info(pn)) {
        nodo_liberar(pn);
        return ERROR;
    }
    if (lista_vacia(pl) == TRUE) {
        ④ first(pl) = pn;
        return OK;
    }
    for (qn=first(pl); next(qn)!=NULL; qn=next(qn))
        ;
    next(qn) = pn;
    return OK;
}
```



Implementación en C: lista_insertarFin

- Implementación (lista no vacía) ①②

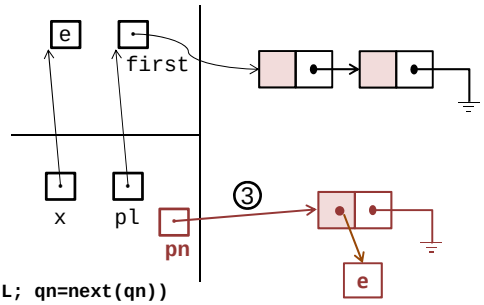
```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
    info(pn) = elemento_copiar(pe);
    if (!info(pn)) {
        nodo_liberar(pn);
        return ERROR;
    }
    if (lista_vacia(pl) == TRUE) {
        first(pl) = pn;
        return OK;
    }
    for (qn=first(pl); next(qn)!=NULL; qn=next(qn))
        ;
    next(qn) = pn;
    return OK;
}
```



Implementación en C: lista_insertarFin

- Implementación (lista no vacía)

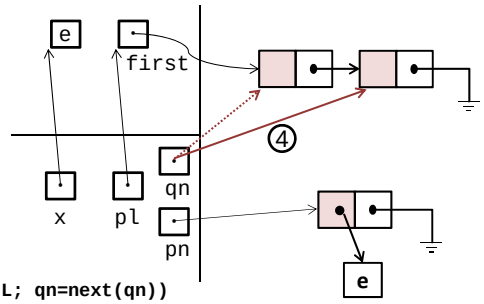
```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
③   info(pn) = elemento_copiar(pe);
    if (!info(pn)) {
        nodo_liberar(pn);
        return ERROR;
    }

    if (lista_vacia(pl) == TRUE) {
        first(pl) = pn;
        return OK;
    }
    for (qn=first(pl); next(qn)!=NULL; qn=next(qn))
        ;
    next(qn) = pn;
    return OK;
}
```



Implementación en C: lista_insertarFin

- Implementación (lista no vacía)

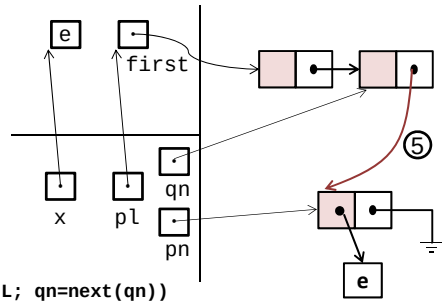
```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
    info(pn) = elemento_copiar(pe);
    if (!info(pn)) {
        nodo_liberar(pn);
        return ERROR;
    }

    if (lista_vacia(pl) == TRUE) {
        first(pl) = pn;
        return OK;
    }
④   for (qn=first(pl); next(qn)!=NULL; qn=next(qn))
        ;
    next(qn) = pn;
    return OK;
}
```



Implementación en C: lista_insertarFin

- Implementación (lista no vacía)

```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
    info(pn) = elemento_copiar(pe);
    if (!info(pn)) {
        nodo_liberar(pn);
        return ERROR;
    }

    if (lista_vacia(pl) == TRUE) {
        first(pl) = pn;
        return OK;
    }
    for (qn=first(pl); next(qn)!=NULL; qn=next(qn))
        ;
⑤   next(qn) = pn;
    return OK;
}
```



Implementación en C: lista_insertarFin

- Implementación (sin macros)

```
status lista_insertarFin(Lista *pl, Elemento *pe) {
    Nodo *pn = NULL, *qn = NULL;

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

    pn = nodo_crear();
    if (!pn) {
        return ERROR;
    }
    pn->info = elemento_copiar(pe);
    if (!pn->info) {
        nodo_liberar(pn);
        return ERROR;
    }

    if (lista_vacia(pl) == TRUE) {
        pl->first = pn;
        return OK;
    }
    for (qn=pl->first; qn->next!=NULL; qn=qn->next)
        ;
    qn->next = pn;
    return OK;
}
```

```
// Con bucle while
qn = pl->first;
while (qn->next!=NULL) {
    qn = qn->next;
}
```

Implementación en C: lista_extraerFin

- Implementar la función **lista_extraerFin**

```
Elemento *lista_extraerFin(Lista *pl)
```

Implementación en C: lista_extraerFin

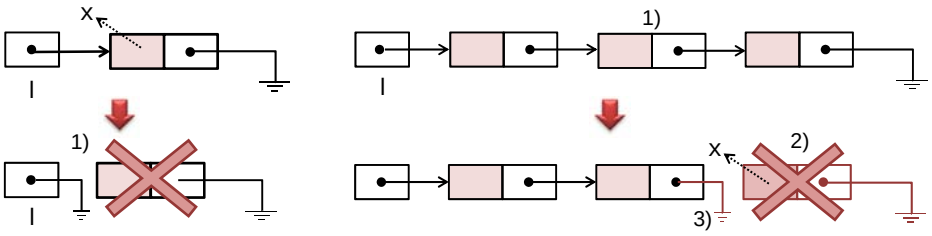
- **Extraer un elemento del inicio de una lista**

Si se tiene un nodo:

1) Eliminar ese nodo y dejar lista vacía

Si se tiene más de un nodo:

1) Recorrer la lista hasta situarse en el penúltimo nodo
2) Liberar el último nodo
3) Hacer que el "penúltimo" nodo apunte a NULL



Implementación en C: lista_extraerFin

- Implementación

```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl)==TRUE) return ERROR;

    // Caso: 1 nodo
    if (!next(first(pl)) {
        pe = elemento_copiar(info(first(pl)));
        if (!pe) return NULL;
        nodo_liberar(first(pl));
        first(pl) = NULL;
        return pe;
    }
    // Caso: 2 o más nodos
    // -> se situa pn en el penúltimo nodo de la lista
    for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
    pe = elemento_copiar(info(next(pn)));
    if (!pe) return NULL;
    nodo_liberar(next(pn));
    next(pn) = NULL;
    return pe;
}
```

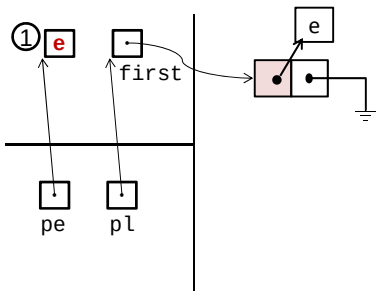
```
// Con bucle while
pn = first(pl);
while (next(next(pn))!=NULL) {
    pn = next(pn);
}
```

2 casos
1) Lista de un nodo
2) Lista de varios nodos

Implementación en C: lista_extraerFin

- Implementación (lista de un nodo)

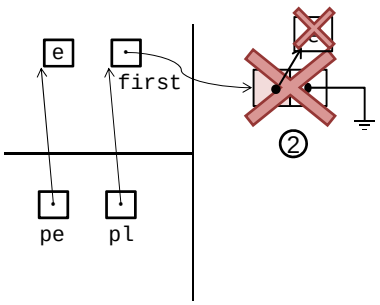
```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl)==TRUE) return ERROR;

    // Caso: 1 nodo
    if (!next(first(pl)) {
①   pe = elemento_copiar(info(first(pl)));
        if (!pe) return NULL;
        nodo_liberar(first(pl));
        first(pl) = NULL;
        return pe;
    }
```



Implementación en C: lista_extraerFin

- Implementación (lista de un nodo)

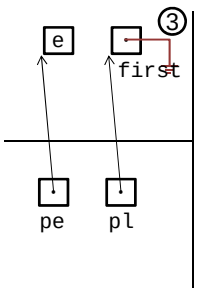
```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl)==TRUE) return ERROR;

    // Caso: 1 nodo
    if (!next(first(pl)) {
        pe = elemento_copiar(info(first(pl)));
        if (!pe) return NULL;
    ② nodo_liberar(first(pl));
        first(pl) = NULL;
        return pe;
    }
```



Implementación en C: lista_extraerFin

- Implementación (lista de un nodo)

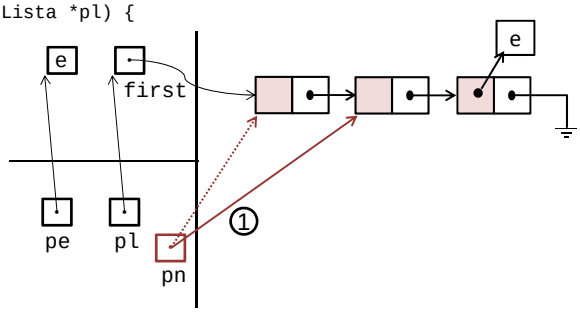
```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl)==TRUE) return ERROR;

    // Caso: 1 nodo
    if (!next(first(pl)) {
        pe = elemento_copiar(info(first(pl)));
        if (!pe) return NULL;
        nodo_liberar(first(pl));
    ③ first(pl) = NULL;
        return pe;
    }
```



Implementación en C: lista_extraerFin

- Implementación (lista de varios nodos)

```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;
```

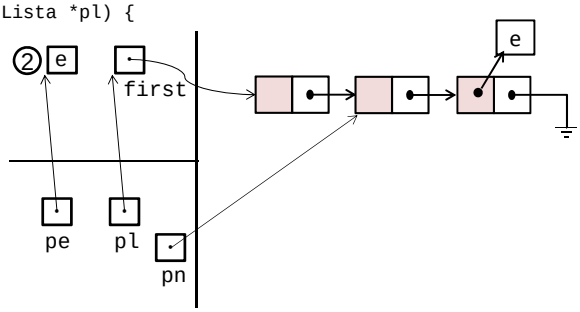



```
    // Caso: 2 o más nodos
    // -> se situa pn en el penúltimo nodo de la lista
①   for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
    pe = elemento_copiar(info(next(pn)));
    if (!pe) return NULL;
    nodo_liberar(next(pn));
    next(pn) = NULL;
    return pe;
}
```


Implementación en C: lista_extraerFin

- Implementación (lista de varios nodos)

```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;
```

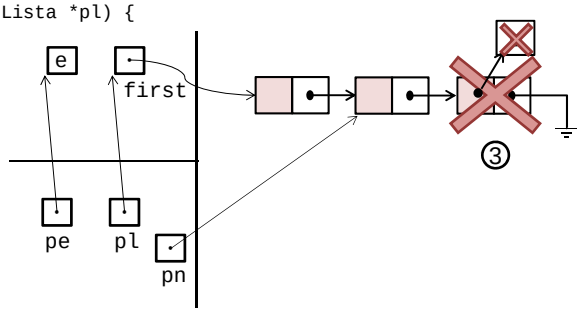



```
    // Caso: 2 o más nodos
    // -> se situa pn en el penúltimo nodo de la lista
    for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
②   pe = elemento_copiar(info(next(pn)));
    if (!pe) return NULL;
    nodo_liberar(next(pn));
    next(pn) = NULL;
    return pe;
}
```

Implementación en C: lista_extraerFin

- Implementación (lista de varios nodos)

```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;
```

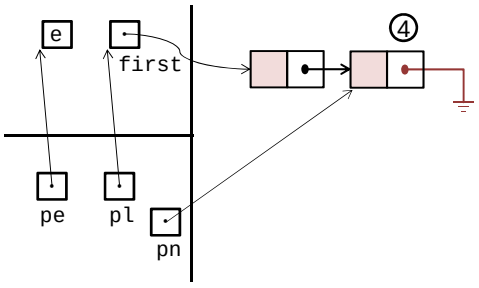



```
    // Caso: 2 o más nodos
    // -> se situa pn en el penúltimo nodo de la lista
    for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
    pe = elemento_copiar(info(next(pn)));
    if (!pe) return NULL;
③   nodo_liberar(next(pn));
    next(pn) = NULL;
    return pe;
}
```


Implementación en C: lista_extraerFin

- Implementación (lista de varios nodos)

```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;
```




```
    // Caso: 2 o más nodos
    // -> se situa pn en el penúltimo nodo de la lista
    for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
    pe = elemento_copiar(info(next(pn)));
    if (!pe) return NULL;
    nodo_liberar(next(pn));
④   next(pn) = NULL;
    return pe;
}
```

Implementación en C: lista_extraerFin

- Implementación (sin macros)

```
Elemento *lista_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl)==TRUE) return ERROR;

    // Caso: 1 nodo
    if (!pl->first->next) {
        pe = elemento_copiar(pl->first->info);
        if (!pe) return NULL;
        nodo_liberar(pl->first);
        pl->first = NULL;
        return pe;
    }

    // Caso: 2 o más nodos
    // -> se situa pn en el penúltimo nodo de la lista
    for (pn=pl->first; pn->next->next!=NULL; pn=pn->next)
        ;
    pe = elemento_copiar(pn->next->info);
    if (!pe) return NULL;
    nodo_liberar(pn->next);
    pn->next = NULL;
    return pe;
}
```

```
// Con bucle while
pn = pl->first;
while (pn->next->next!=NULL) {
    pn = pn->next;
}
```


Implementación en C: lista_liberar

- Implementar la función **lista_liberar**

```
void lista_liberar(Lista *pl)
```

Implementación en C: lista_liberar

1. Implementación usando primitivas

```
void lista_liberar(Lista *pl) {
    if (!pl) return;

    while (lista_vacia(pl) == FALSE) {
        elemento_liberar(lista_extraerIni(pl));
    }
    free(pl);
}
```

2. Implementación accediendo a la estructura

```
void lista_liberar(Lista *pl) {
    Nodo *pn = NULL;

    if (!pl) return;

    while (first(pl) != NULL) {
        pn = first(pl);
        first(pl) = next(first(pl));
        nodo_liberar(pn);
    }
    free(pl);
}
```


Implementación en C: lista_liberar

3. Implementación recursiva

```
void lista_liberar(Lista *pl) {
    if (!pl) {
        return;
    }

    // Origen de llamadas recursivas: primer nodo de la lista
    lista_liberar_rec(first(pl));

    // Liberación de la estructura Lista
    free(pl);
}

void lista_liberar_rec(Node *pn) {
    // Condición de parada: el nodo al que hemos llegado es NULL
    if (!pn) {
        return;
    }

    // Llamada recursiva: liberacion de nodos siguientes al actual
    lista_liberar_rec(next(pn));

    // Liberación del nodo actual
    nodo_liberar(pn);
}
```

Contenidos

- El TAD Lista
- Estructura de datos de Lista
- Implementación en C de Lista
- **Implementación de Pila y Cola con Lista**
- Tipos de Listas

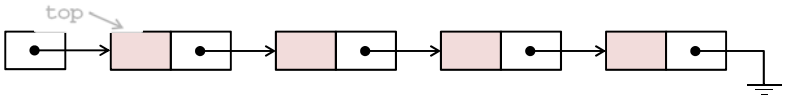
Pila. Implementación con una lista enlazada

- EdD de Pila con datos en un **array**

```
// En pila.c
struct _Pila {
   Elemento *datos[PILA_MAX];
   int top;
};
// En pila.h
typedef struct _Pila Pila;
```

- EdD de Pila con datos en una **lista enlazada**

```
// En pila.c
struct _Pila {
   Lista *pl;
};
// En pila.h
typedef struct _Pila Pila;
```



Pila. Implementación con una lista enlazada

- Implementación de las primitivas de Pila sobre la EdD basada en lista enlazada

```
Pila *pila_crear()
void pila_liberar(Pila *ps)
boolean pila_vacia(Pila *ps)
boolean pila_llena(Pila *ps)
status pila_push(Pila *ps, Elemento *pe) ) → lista_insertarIni
Elemento *pila_pop(Pila *ps) ) → lista_extraerIni
```

```
struct _Pila {
   Lista *pl;
};
```


Pila. Implementación con una lista enlazada


```
Pila *pila_crear() {
   Pila *ps = NULL;
   ps = (Pila *) malloc(sizeof(Pila));
   if (!ps) {
      return NULL;
   }
   ps->pl = lista_crear();
   if (!ps->pl) {
      free(ps);
      return NULL;
   }
   return ps;
}

void pila_liberar(Pila *ps) {
   if (ps) {
      lista_liberar(ps->pl);
      free(ps);
   }
}
```

Pila. Implementación con una lista enlazada


```
boolean pila_vacia(Pila *ps) {
   if (!ps) {
      return TRUE;             // Caso de error
   }
   return lista_vacia(ps->pl); // Caso normal
}

boolean pila_llena(Pila *ps) {
   if (!ps) {
      return TRUE;             // Caso de error
   }
   return FALSE;               // Caso normal: la pila nunca está llena
}
```


Pila. Implementación con una lista enlazada


```
status pila_push(Pila *ps, Elemento *pe) {
   if (!ps || !pe) {
      return ERROR;
   }
   return lista_insertarIni(ps->pl, pe);
}

Elemento *pila_pop(Pila *ps) {
   if (!ps) {
      return ERROR;
   }
   return lista_extraerIni(ps->pl);
}
```

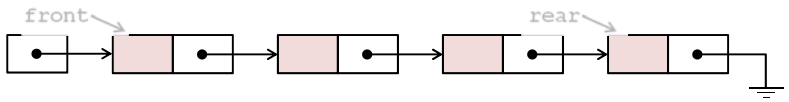
Cola. Implementación con una lista enlazada

- EdD de Cola con datos en un **array**

```
// En cola.c
struct _Cola {
   Elemento *datos[COLA_MAX];
   int front, rear;
};
// En cola.h
typedef struct _Cola Cola;
```

- EdD de Cola con datos en una **lista enlazada**

```
// En cola.c
struct _Cola {
   Lista *pl;
};
// En cola.h
typedef struct _Cola Cola;
```



Cola. Implementación con una lista enlazada

- Implementación de las primitivas de Cola sobre la EdD basada en lista enlazada

```
Cola *cola_crear()
void cola_liberar(Cola *pq)
boolean cola_vacia(Cola *pq)
boolean cola_llena(Cola *pq)
status cola_insertar(Cola *pq, Elemento *pe) → lista_insertarFin
Elemento *cola_extraer(Cola *pq) → lista_extraerIni
```

```
struct _Cola {
   Cola *pl;
};
```

Cola. Implementación con una lista enlazada 72

```
Cola *cola_crear() {
   Cola *pq = NULL;
   pq = (Cola *) malloc(sizeof(Cola));
   if (!pq) {
      return NULL;
   }
   pq->pl = lista_crear();
   if (!pq->pl) {
      free(pq);
      return NULL;
   }
   return pq;
}

void pila_liberar(Cola *pq) {
   if (pq) {
      lista_liberar(pq->pl);
      free(pq);
   }
}
```


Cola. Implementación con una lista enlazada 73

```
boolean cola_vacia(Cola *pq) {
   if (!pq) {
      return TRUE;             // Caso de error
   }
   return lista_vacia(pq->pl); // Caso normal
}

boolean cola_llena(Cola *pq) {
   if (!pq) {
      return TRUE;             // Caso de error
   }
   return FALSE;               // Caso normal: la cola nunca está llena
}
```

Cola. Implementación con una lista enlazada


```
status cola_insertar(Cola *pq, Elemento *pe) {
   if (!pq || !pe) {
      return ERROR;
   }
   return lista_insertarFin(pq->pl, pe);
}

Elemento *cola_extraer(Cola *pq) {
   if (!pq) {
      return ERROR;
   }
   return lista_extraerIni(pq->pl);
}
```

Colas. Implementación con listas enlazadas

- Inconveniente: la función de **inserción es ineficiente**, pues recorre toda la lista para insertar un elemento al final de la misma
  - Posible solución: usar una lista circular

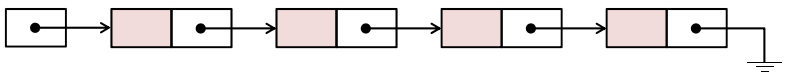
```
struct _Cola {
   ListaCircular *pl;
};
typedef struct _Cola Cola;


status cola_insertar(Cola *pq, Elemento *pe) {
  if (!pq || !pe) {
     return ERROR;
  }
  return listaCircular_insertarFin(pq->pl, pe);
}
```

Contenidos

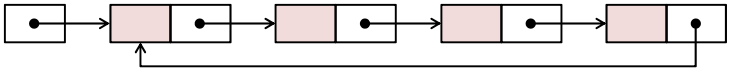
- El TAD Lista
- Estructura de datos de Lista
- Implementación en C de Lista
- Implementación de Pila y Cola con Lista
- **Tipos de Listas**

Tipos de listas enlazadas

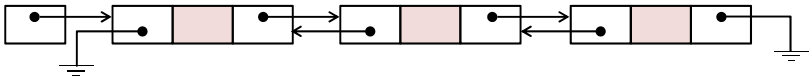
- Lista enlazada



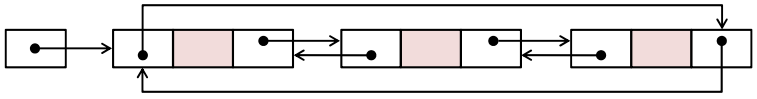
- Lista enlazada circular



- Lista doblemente enlazada



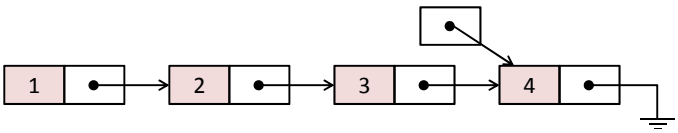
- Lista doblemente enlazada circular



Lista enlazada circular

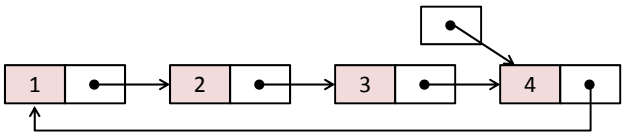
78

- Problema: **insertar/extraer del final de una lista enlazada es costoso**, pues hay que recorrer la lista para situarse en el (pen)último nodo
- Solución 1: hacer que lista->first apunte al último nodo



Pero, ¿cómo acceder al primer nodo?

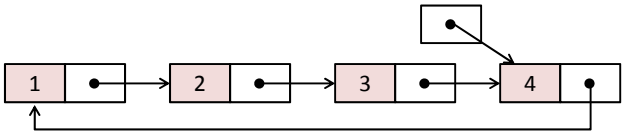
Haciendo que el último nodo apunte al primero



Lista enlazada circular

79

- **Lista enlazada circular**. Lista enlazada en la que:
  - el campo first de la lista apunta al último nodo
  - el campo next del último nodo apunta al primer nodo



- Aplicación: implementación de la Cola circular

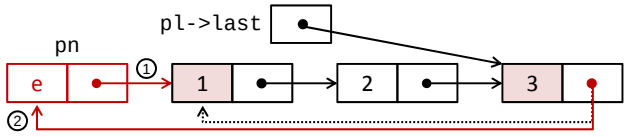
```
// EdD análoga a la de Lista
struct _ListaCircular {
    Nodo *last;
};

typedef struct _ListaCircular ListaCircular;
```

```
struct _Cola {
    ListaCircular *pl;
};
typedef struct _Cola Cola;
```

```
// Diferente implementación de primitivas de Lista
```

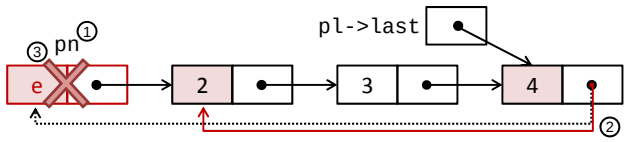
listaCircular_insertarIni


```
status listaCircular_insertarIni(Lista *pl, 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) {
        next(pn) = pn;                  // El next de pn apunta a sí mismo
        last(pl) = pn;
    }
    // Caso 2: lista no vacía
    else {
      (1) next(pn) = next(last(pl));    // El next de pn apunta al primer nodo
      (2) next(last(pl)) = pn;          // El next del último nodo apunta a pn
    }
    return OK;
}
```



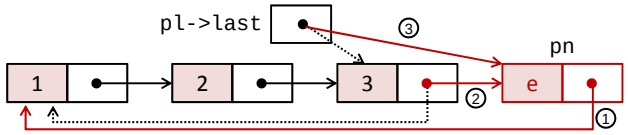
listaCircular_extraerIni


```
Elemento *listaCircular_extraerIni(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl)==TRUE) return NULL;

    pe = elemento_copiar(info(next(last(pl))));   // Se copia el elemento del primer nodo
    if (!pe) {
        return NULL;
    }

    // Caso 1: lista de un nodo
    if (next(last(pl)) == last(pl)) {
        nodo_liberar(last(pl));         // Se libera el nodo
        last(pl) = NULL;                // Se deja la lista vacía
    }
    // Caso 2: lista de varios nodos
    else {
      (1) pn = next(last(pl));          // Se sitúa pn en el primer nodo
      (2) next(last(pl)) = next(pn);    // El next del último nodo apunta al segundo nodo
      (3) nodo_liberar(pn);             // Se libera pn
    }
    return pe;
}
```



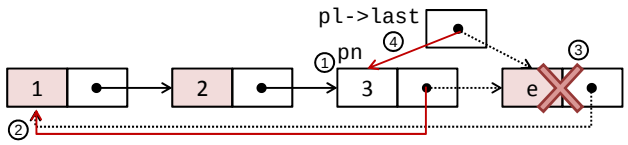
listaCircular_insertarFin


```
status listaCircular_insertarFin(Lista *pl, 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) {
        next(pn) = pn;                  // El next de pn apunta al propio nodo
        last(pl) = pn;
    }
    // Caso 2: lista no vacía
    else {
      (1) next(pn) = next(last(pl));    // El next de pn apunta al primer nodo
      (2) next(last(pl)) = pn;          // El next del que era el último nodo apunta a pn
      (3) last(pl) = pn;                // El nuevo último elemento es pn
    }
    return OK;
}
```



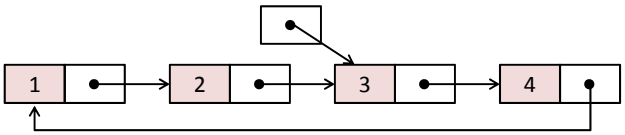
listaCircular_extraerFin


```
Elemento *listaCircular_extraerFin(Lista *pl) {
    Nodo *pn = NULL;
    Elemento *pe = NULL;

    if (!pl || lista_vacia(pl)==TRUE) return NULL;

    pe = elemento_copiar(info(last(pl)));   // Se copia el elemento del último nodo
    if (!pe) {
        return NULL;
    }

    // Caso 1: lista de un nodo
    if (next(last(pl)) == last(pl)) {
        nodo_liberar(last(pl));         // Se libera el nodo
        last(pl) = NULL;                // Se deja la lista vacía
        return pe;
    }

    // Caso 2: lista de varios nodos
  (1) for (pn=last(pl); next(pn)!=last(pl); pn=next(pn))  // Se sitúa pn en el penúltimo nodo
        ;
  (2) next(pn) = next(last(pl));        // El next de pn apunta al primer nodo
  (3) nodo_liberar(last(pl));           // Se libera el último nodo
  (4) last(pl) = pn;                    // El nuevo último noodo es pn
    return pe;
}
```



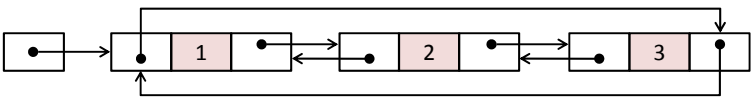
Lista enlazada circular

- **Ventajas**
  - Las primitivas `insertarFin`, `extraerFin`, `insertarIni` son eficientes, al no tener que recorrer la lista en general
  - No se usa más memoria que la que se usaba para Lista
- **Inconveniente**
  - La primitiva `extraerFin` tiene que recorrer la lista
  - Solución 1: **apuntar al penúltimo nodo** de la lista
    - Añade complejidad a todas las primitivas



Lista enlazada circular

- **Inconveniente**
  - La primitiva `extraerFin` tiene que recorrer la lista
  - Solución 2: **lista doblemente enlazada circular**
    - Permite acceso inmediato a elementos anteriores y posteriores
    - Pero añade complejidad a todas las primitivas



Lista doblemente enlazada

- EdD de Nodo y Lista doblemente enlazada

```
// Nodo
struct _NodoDE {
    struct _NodoDE *prev;
    Elemento *info;
    struct _NodoDE *next;
};
typedef struct _NodoDE NodoDE;

// Lista
struct _ListaDE {
   NodoDE *first;
};
typedef struct _ListaDE ListaDE;
```