Programación II

Tema 4. Listas enlazadas

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contenidos

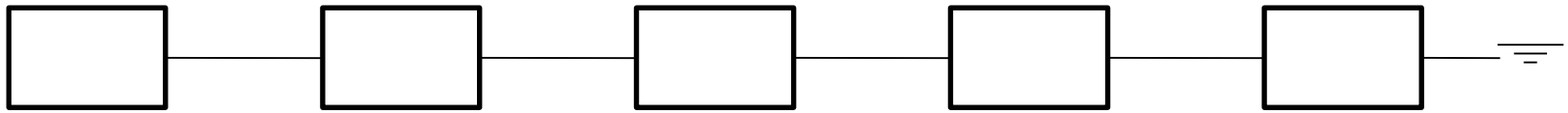
- El TAD Lista
- Estructura de datos de Lista
- Implementación en C de Lista
- Implementación de Pila y Cola con Lista
- Tipos de Listas

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 list_init()
void list_free(Lista l)
Boolean list_isEmpty(Lista l)          // No existe list_isFull!!!
Status list_insertIni(List l, Element e)  // Inserta al inicio
Element list_extractIni(Lista l)          // Extrae del inicio
Status list_insertEnd(List l, Element e)  // Inserta al final
Element list_extractEnd(Lista l)          // Extrae del final
```

- **… y otras**

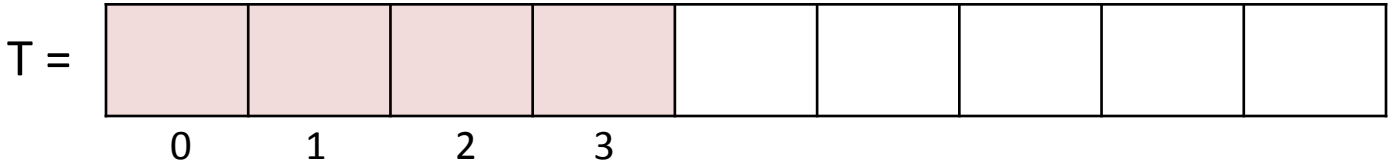
```
// Inserta el elemento e en la posición pos de la lista L
Status list_insertPos(List l, Element e, int pos)
// Inserta el elemento e en la lista L en orden
Status list_insertOrder(List l, Element e)
...
```

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**

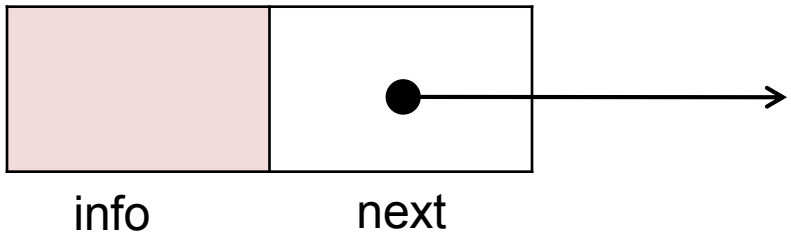


siguiente(T[i]) ≡ T[i + 1]; anterior(T[i]) ≡ T[i – 1]

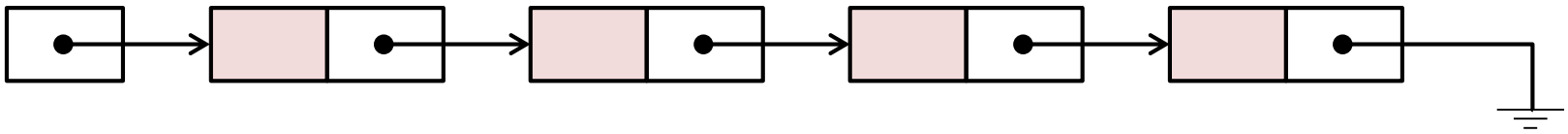
  - *Ventajas*
    - Fácil implementación
    - Memoria estática
  - *Inconvenientes*
    - Desperdicio de espacio
    - Ineficiencia al insertar al inicio y en posiciones intermedias: hay que mover todos los elementos a la derecha una posición
      ¿Posible solución? ¿lista circular? ¿ocurre lo mismo?

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

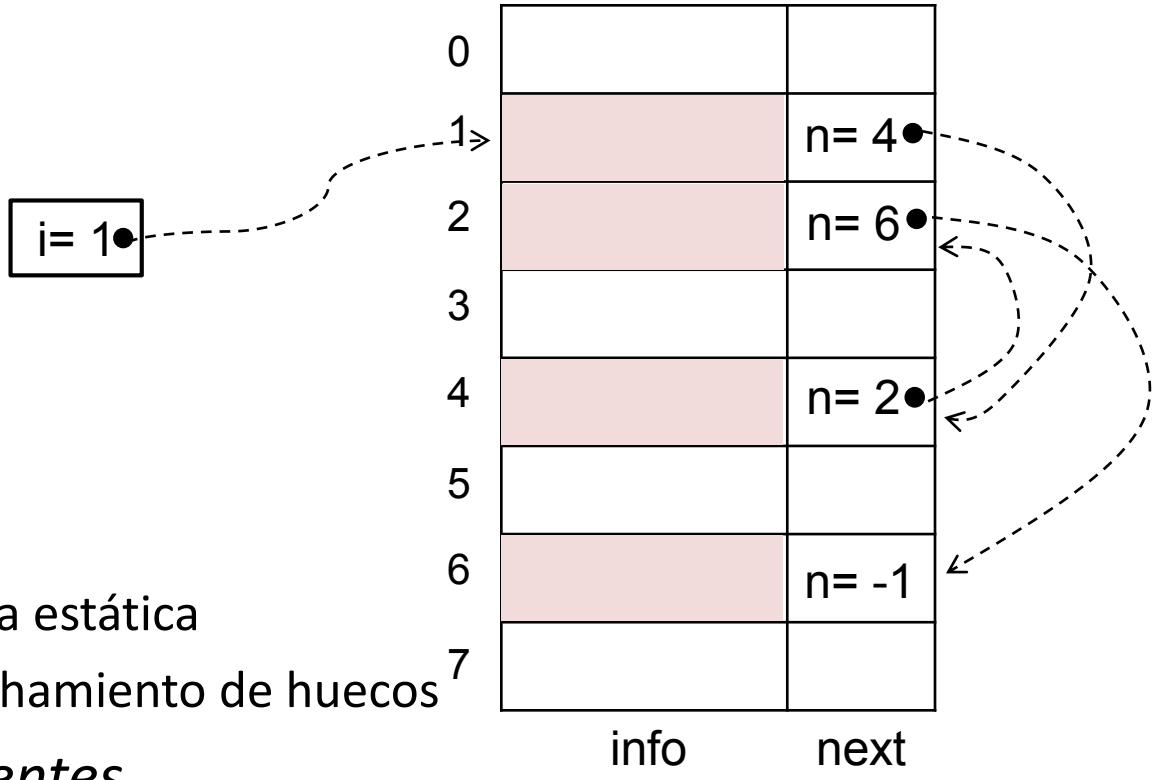


- **Lista enlazada**: colección de nodos enlazados + puntero al nodo inicial. El next del último nodo apunta a NULL.



EdD de Lista: LE estática

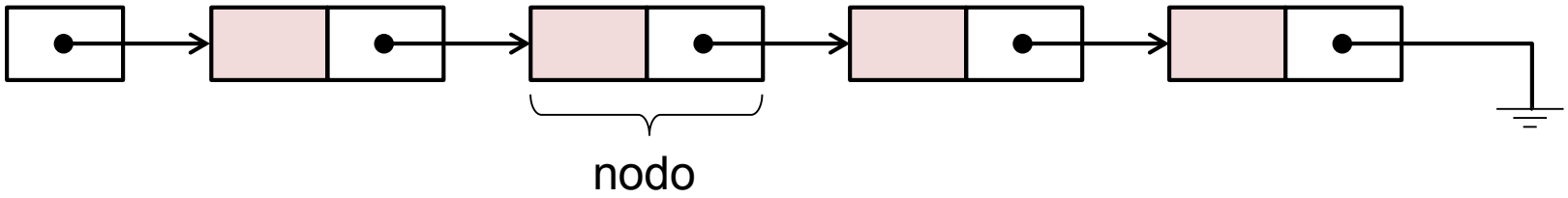
- **Estructura estática para LE**
  - Tabla de nodos



  - *Ventajas*
    - Memoria estática
    - Aprovechamiento de huecos
  - *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

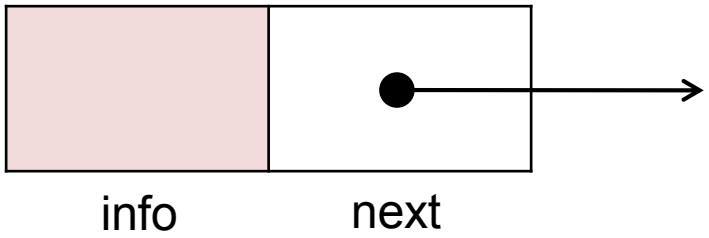


- *Ventajas*
  - Sólo se tiene reservada la memoria que se necesita en cada momento
  - Se pueden albergar tantos elementos como la memoria disponible permita
  - Insertar/extraer nodos no requiere desplazamientos de memoria
- *Inconvenientes*
  - Bueno para acceso secuencial; malo para acceso aleatorio

EdD de Lista en C

- **EdD de Nodo**
  - Se oculta al usuario. Es privado, es un recurso “interno” para gestionar la lista.
    La EdD se define en list.c y las funciones asociadas también.
    Ni el tipo ni los prototipos de las funciones se incorporan a list.h, de modo que no están disponibles en la interfaz para el manejo de listas.

```
// En list.c (antes de la definición del tipo List)
typedef struct _Node {
   Element *info;
   struct _Node *next;
} Node;
```



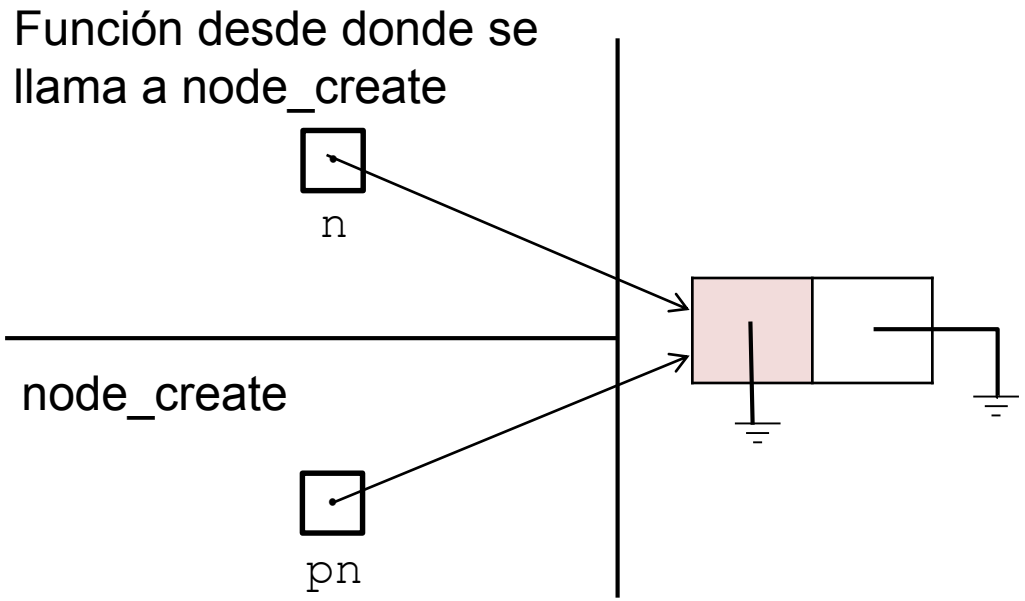
EdD de Lista en C

- **Creación de un nodo**

```
Node *node_create() {
   Node *pn = NULL;
   pn = (Node *) malloc(sizeof(Nodo));
   if (!pn) return NULL;
   pn->info = NULL;   // info apuntará a un elemento
   pn->next = NULL;
   return pn;
}
```

- Ejemplo de llamada

```
Node *n = NULL;
n = node_create();
if (!n) {
   // CdE
}
```



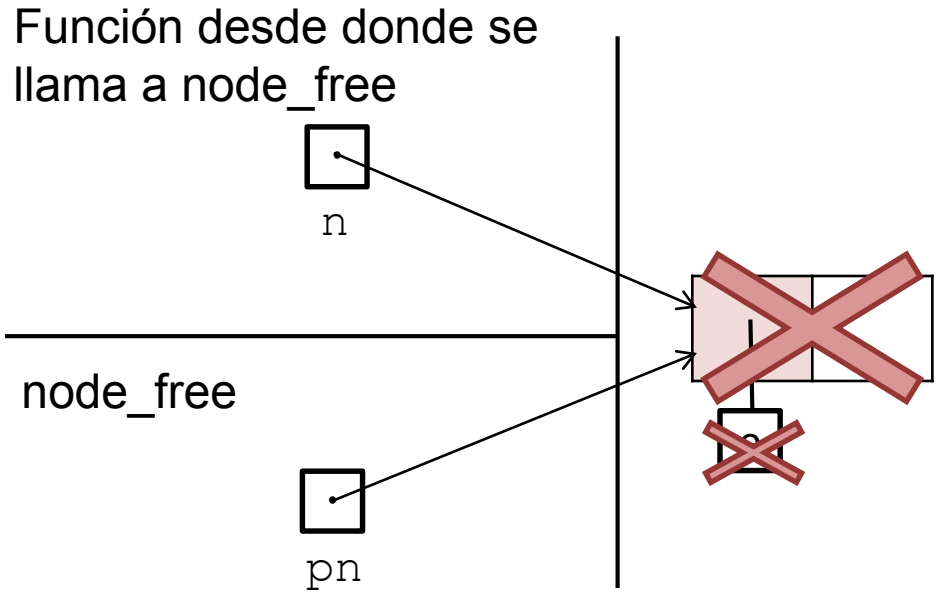
EdD de Lista en C

- **Liberación de un nodo**

```
void node_free(Node *pn) {
   if (pn) {
      element_free(pn->info); // Libera elemento de info
      free(pn);               // Libera nodo
   }
}
```

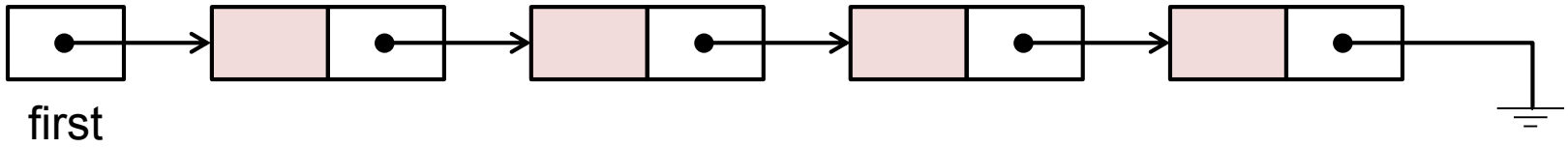
- Ejemplo de llamada

```
Node *n = NULL;
n = node_create();
if (!n) {
   // CdE
}
...
node_free(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 list.h
typedef struct _List List;

// En list.c
struct _List {
   Node *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 (privadas)
Node * node_create() //tras reservar el nodo,apuntar info a un elemento
void node_free(Node *pn) //llama a element_free sobre info

// Primitivas de Lista (públicas).
// Observación: Nodo está oculto para el usuario
List * list_init()
void list_free(List *pl)
Boolean list_isEmpty(List *pl)
Status list_insertIni(List *pl, Element *pe)
Element * list_extractIni(List *pl)
Status list_insertEnd(List *pl, Element *pe)
Element * list_extractEnd(List *pl)
```

Importante: para mayor legibilidad, en algunas de las implementaciones que siguen se han dejado indicado el control de errores de argumentos de entrada o retornos de funciones (habría que completarlos todos)

Implementación en C: list_init

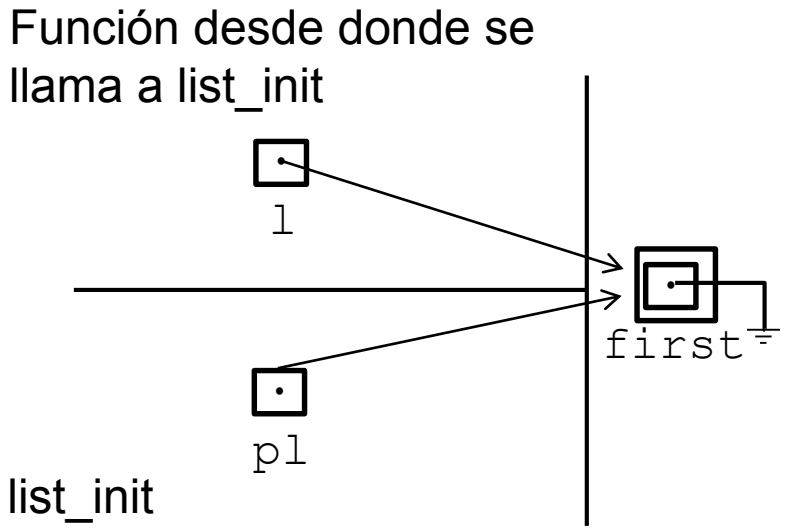

```
struct _List {
   Node *first;
};
```

- **Crear una lista**

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

- Ejemplo de llamada

```
List *l = NULL;
l = list_init();
if (!l)
    ...
```



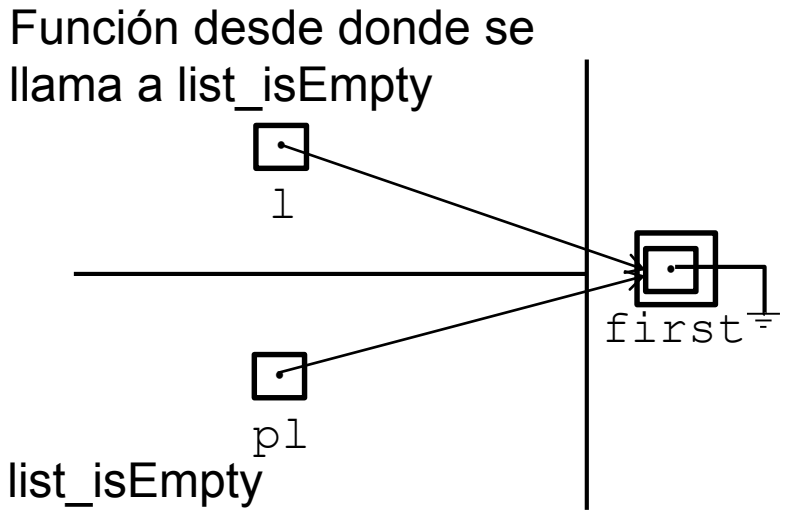
Implementación en C: list_isEmpty

- **Comprobar si una lista está vacía**

```
Boolean list_isEmpty(List *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

```
List *l = NULL;
l = list_init();
...
if (list_isEmpty(l) == FALSE) {
   ...
}
```

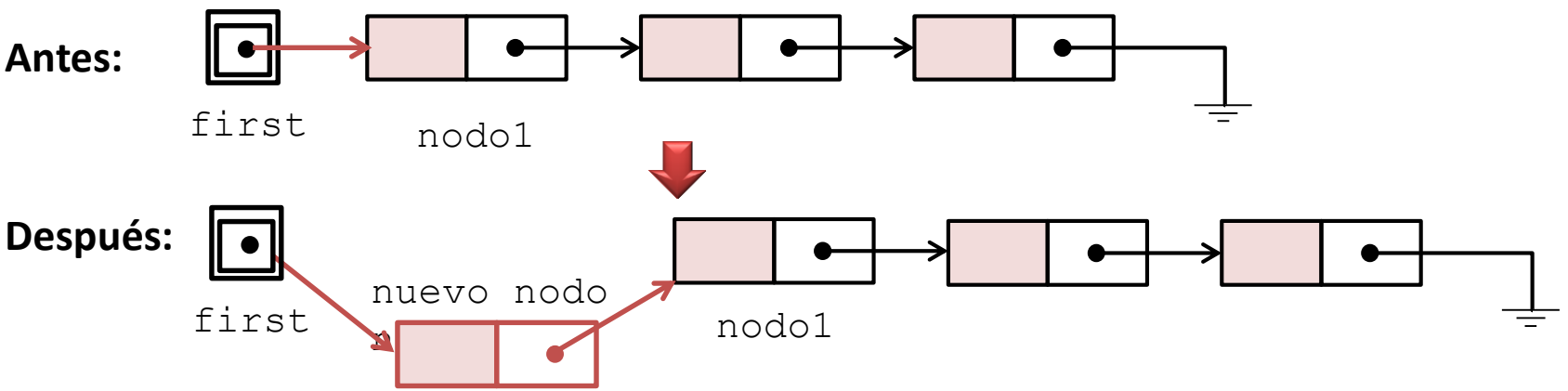


Implementación en C: list_insertIni

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

- Pseudocódigo más detallado (sin CdE)

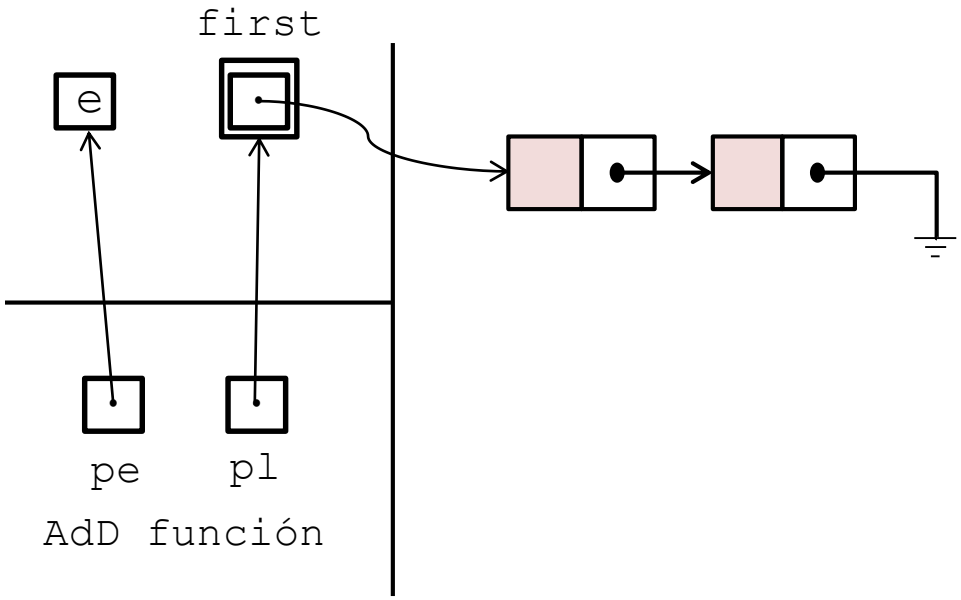
```
Status list_insertIni(List l, Element e)
   Nodo n = node_create()
   info(n) = copy_element(e)
   next(n) = first(l)
   first(l) = n
   return OK
```



Implementación en C: list_insertIni

- ¿Implementación?

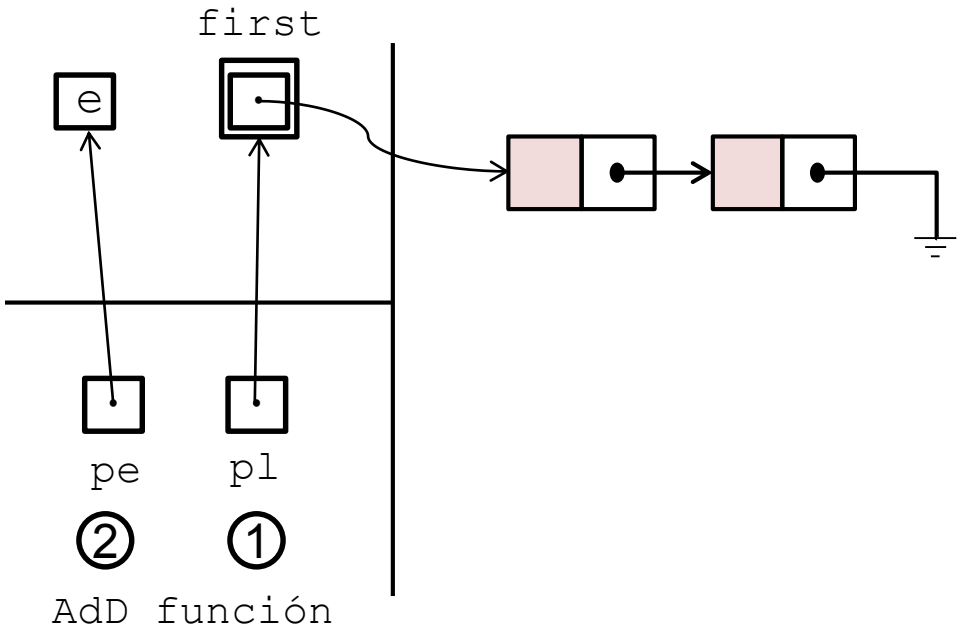
```
Status list_insertIni(List *pl, Element *pe) {
```



Implementación en C: list_insertIni

- ¿Implementación? ① ②

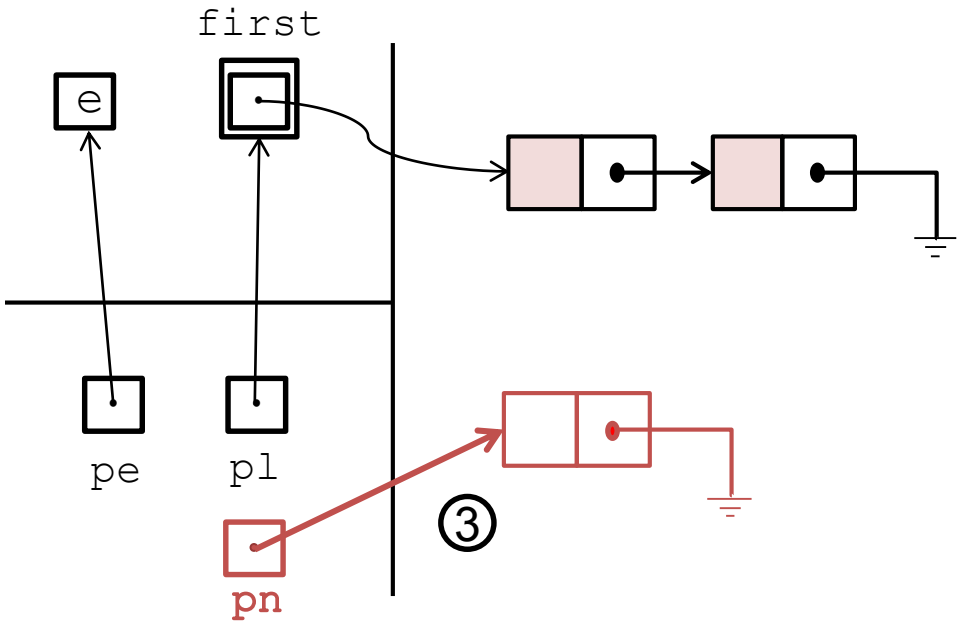
```
Status list_insertIni(List *pl, Element *pe) {
    Node *pn = NULL;

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



Implementación en C: list_insertIni

- ¿Implementación?

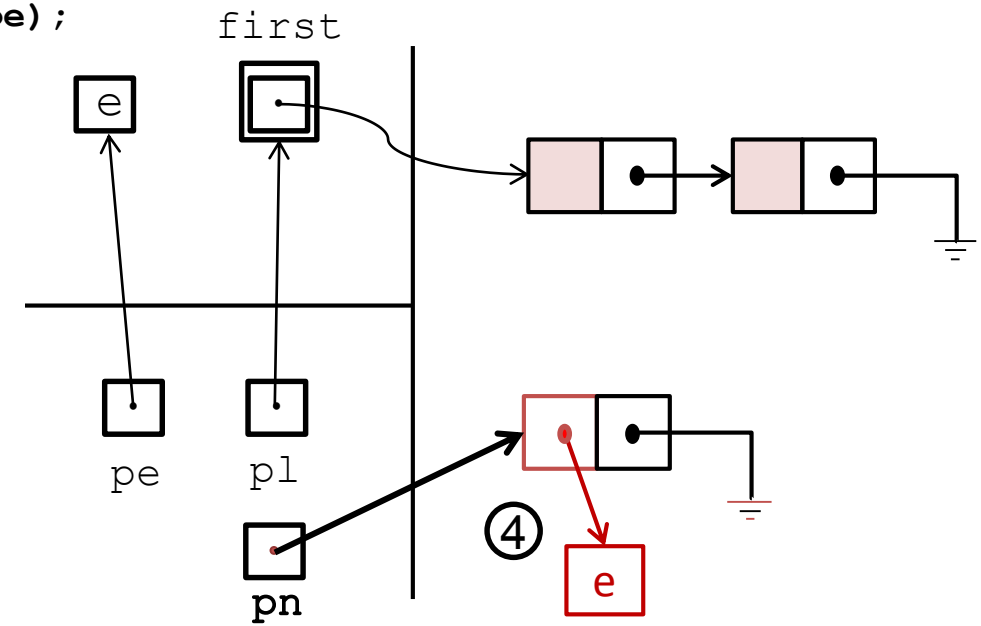
```
Status list_insertIni(List *pl, Element *pe) {
    Node *pn = NULL;

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

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



Implementación en C: list_insertIni

- ¿Implementación?

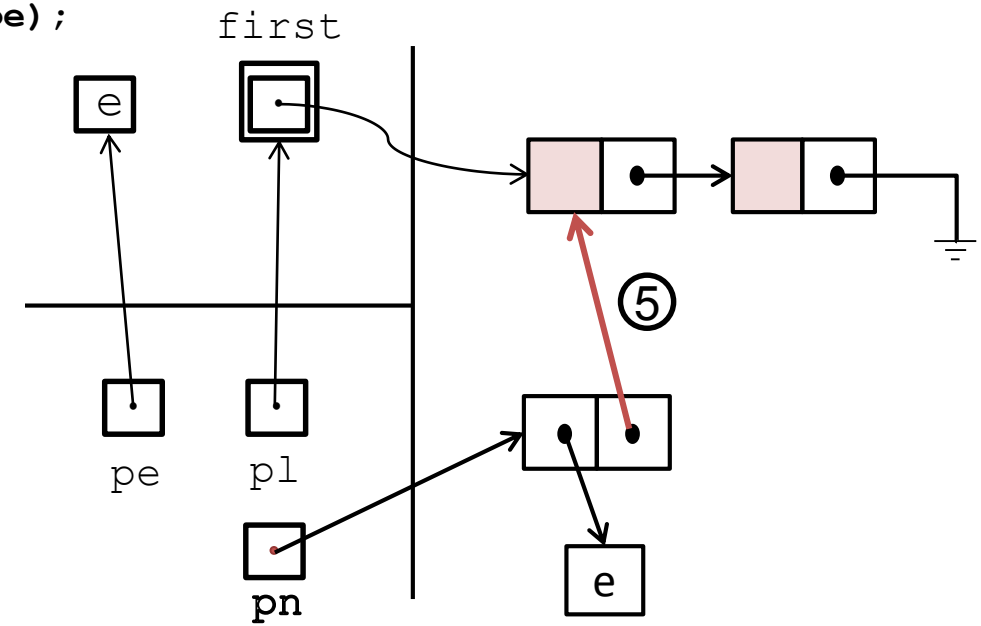
```
Status list_insertIni(List *pl, Element *pe) {
    Node *pn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
④   pn->info = element_copy(pe);
    if (!pn->info) {
        node_free(pn);
        return ERROR;
    }
```



Implementación en C: list_insertIni

- ¿Implementación?

```
Status list_insertIni(List *pl, Element *pe) {
    Node *pn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
    pn->info = element_copy(pe);
    if (!pn->info) {
        node_free(pn);
        return ERROR;
    }
⑤   pn->next = pl->first;
```

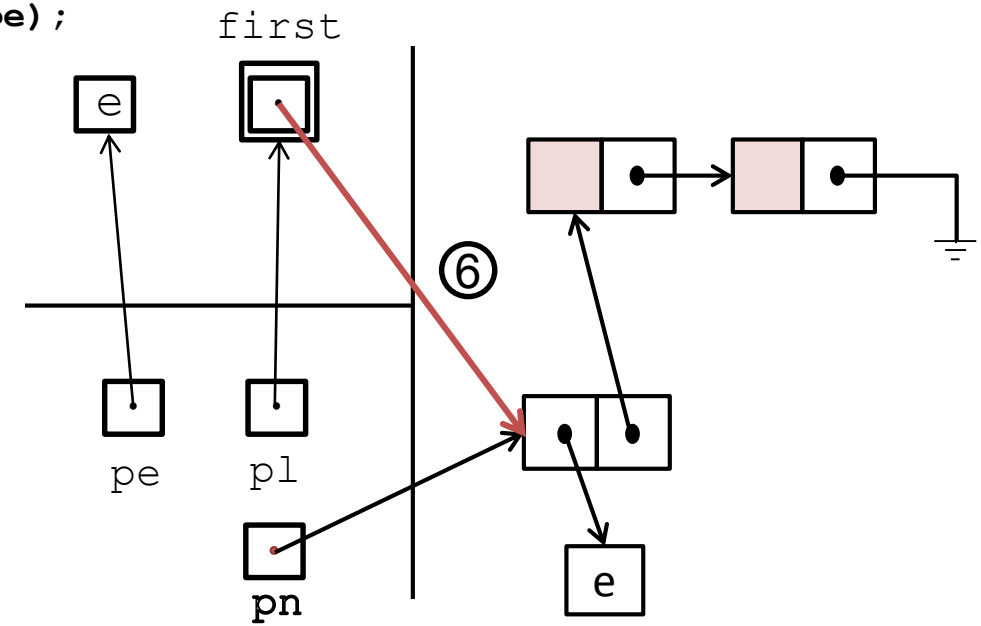


④

Implementación en C: list_insertIni

- ¿Implementación?

```
Status list_insertIni(List *pl, Element *pe) {
    Node *pn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
    pn->info = element_copy(pe);
    if (!pn->info) {
        node_free(pn);
        return ERROR;
    }
    pn->next = pl->first;

⑥  pl->first = pn;

    return OK;
```



Implementación en C: list_insertIni

- Implementación alternativa: uso de macros

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

Status list_insertIni(List *pl, Element *pe) {
    ...
    pn = node_create();
    ...
④   info(pn) = element_copy(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: list_insertIni

- 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:

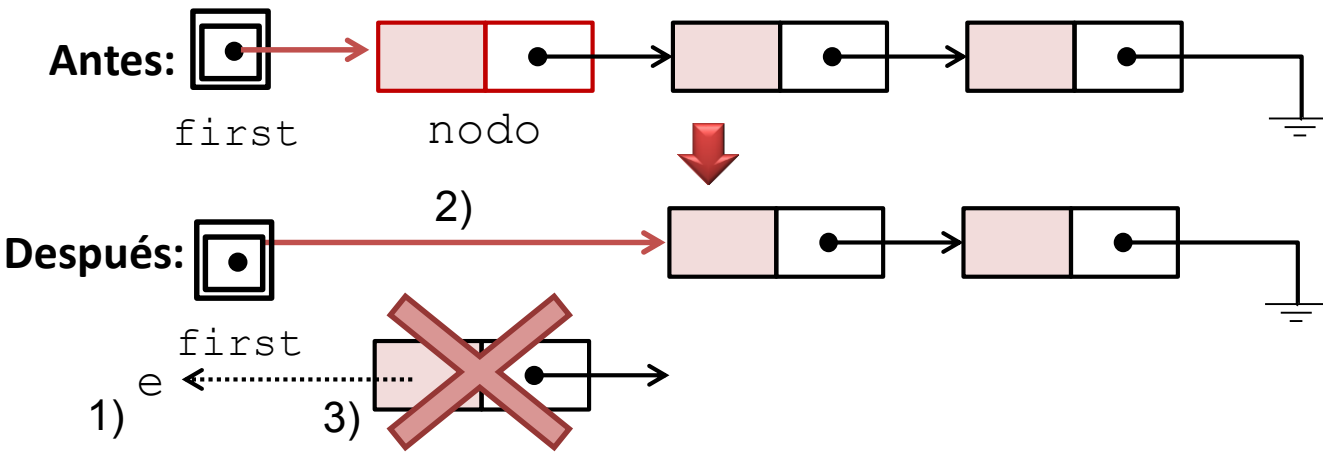
```
#define cuadrado(x) x*x   ➔ ¿correcto?
cuadrado(z+1) ➔ z + 1 * z + 1
#define cuadrado (x) (x)*(x) ➔ (z+1)*(z+1)
```

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

Implementación en C: list_extractIni

- **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 más detallado:

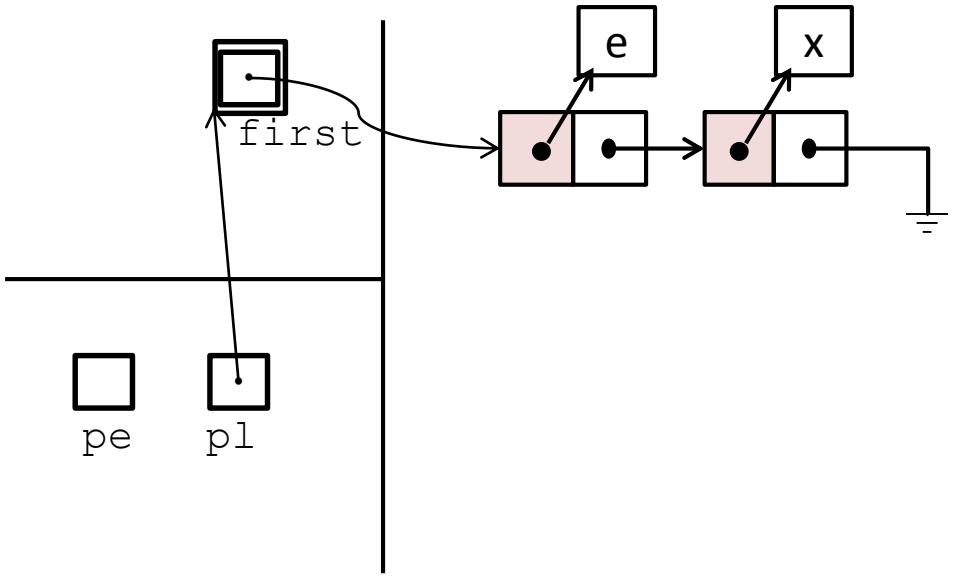
```
Element list_extractIni(List l) {
    if (list_isEmpty(l)) return NULL
    n = first(l)
    e = info(n)
    first(l) = next(n)
    node_free(n)
    return e
}
```



Implementación en C: list_extractIni

- ¿Implementación?

```
Element *list_extractIni(List *pl) {
```

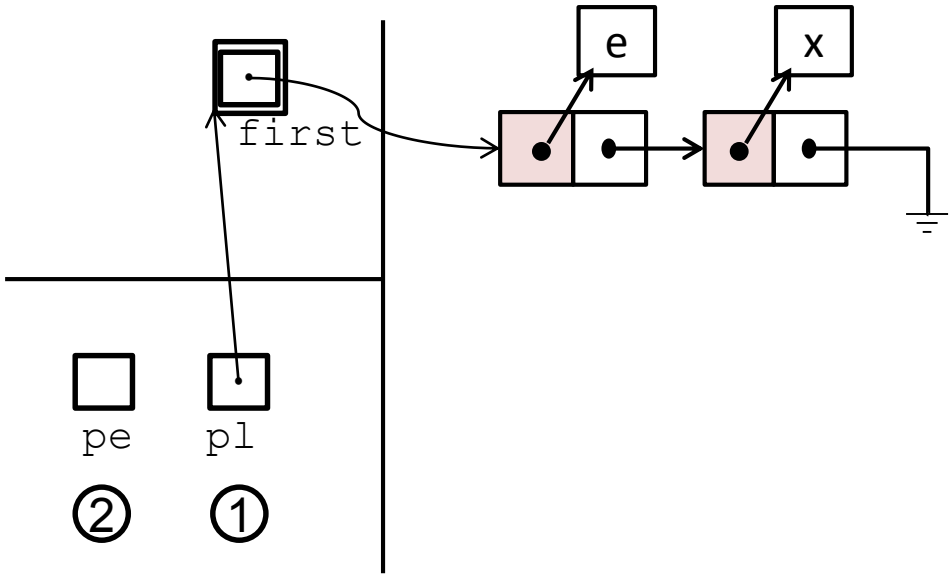


Implementación en C: list_extractIni

- Implementación

①

```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
②   Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }
```

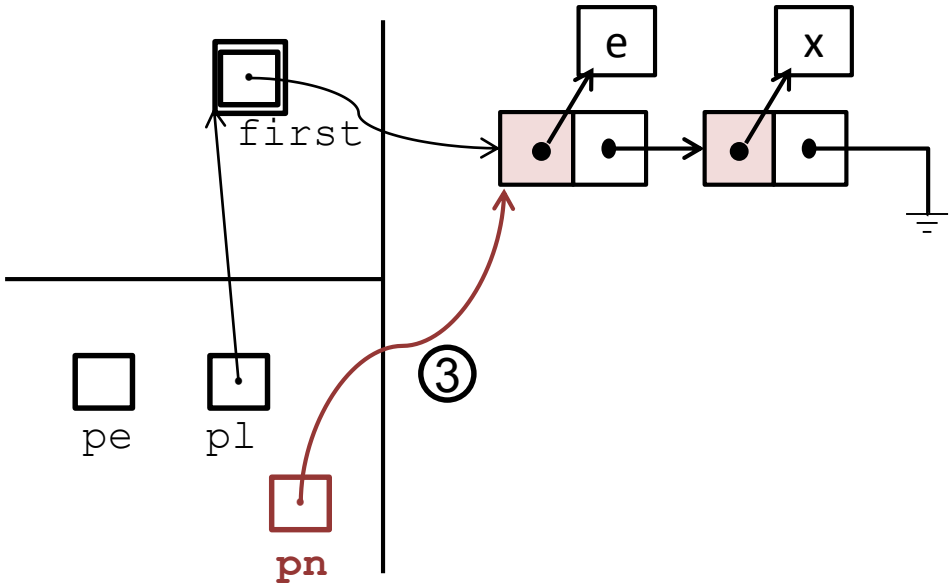


Implementación en C: list_extractIni

- Implementación

```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }
```

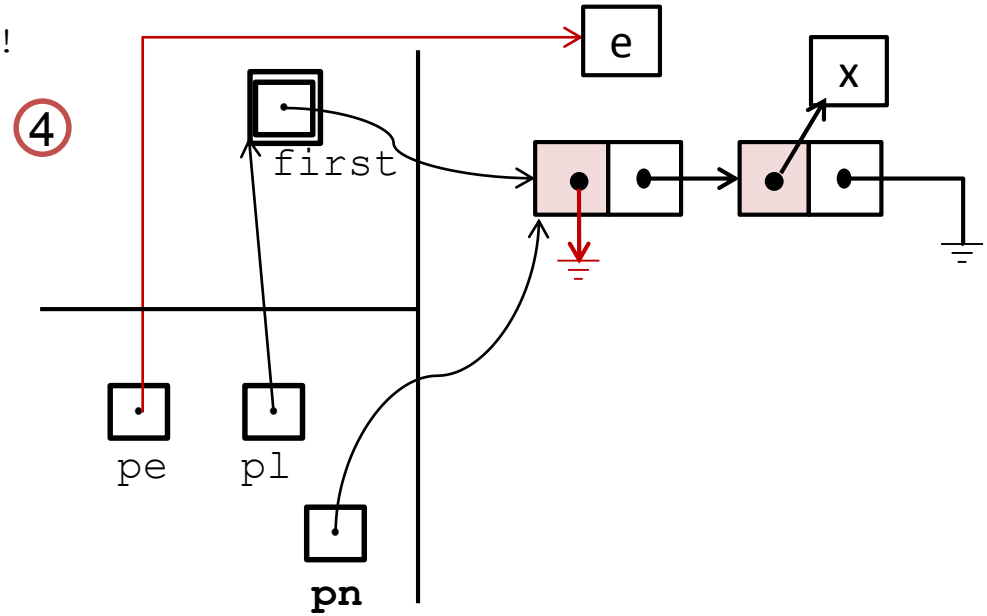
③ **pn = first(pl);**



Implementación en C: list_extractIni

- Implementación

```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }

    pn = first(pl);
④   pe = info(pn);
    info(pn) = NULL; //Importante!!!
```

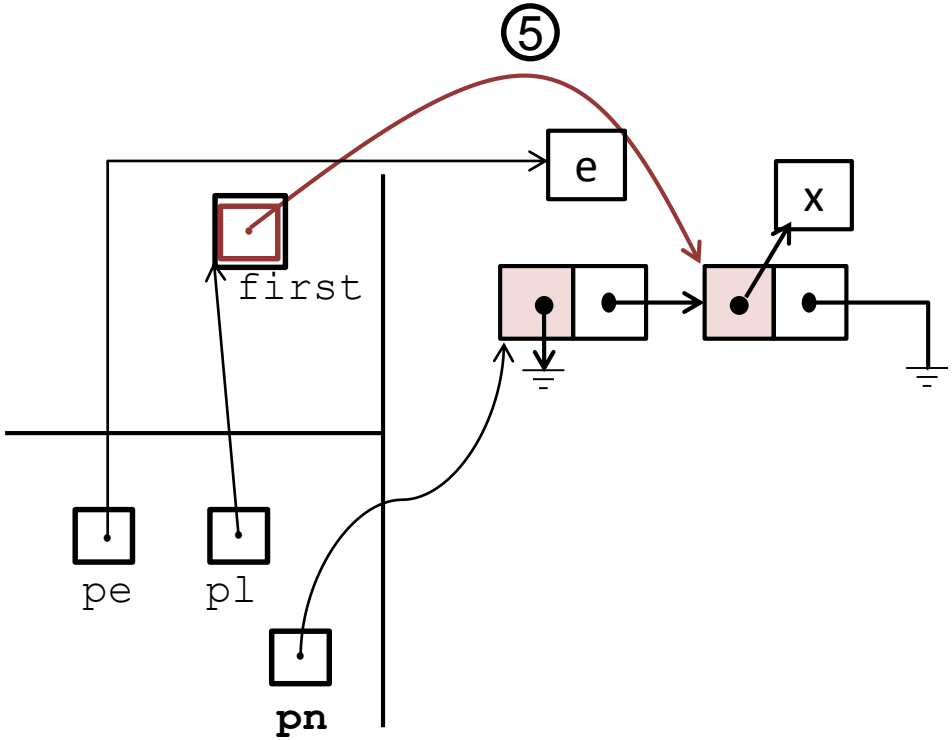


Implementación en C: list_extractIni

- Implementación

```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }

    pn = first(pl);
    pe = info(pn);
    info(pn) = NULL;
```

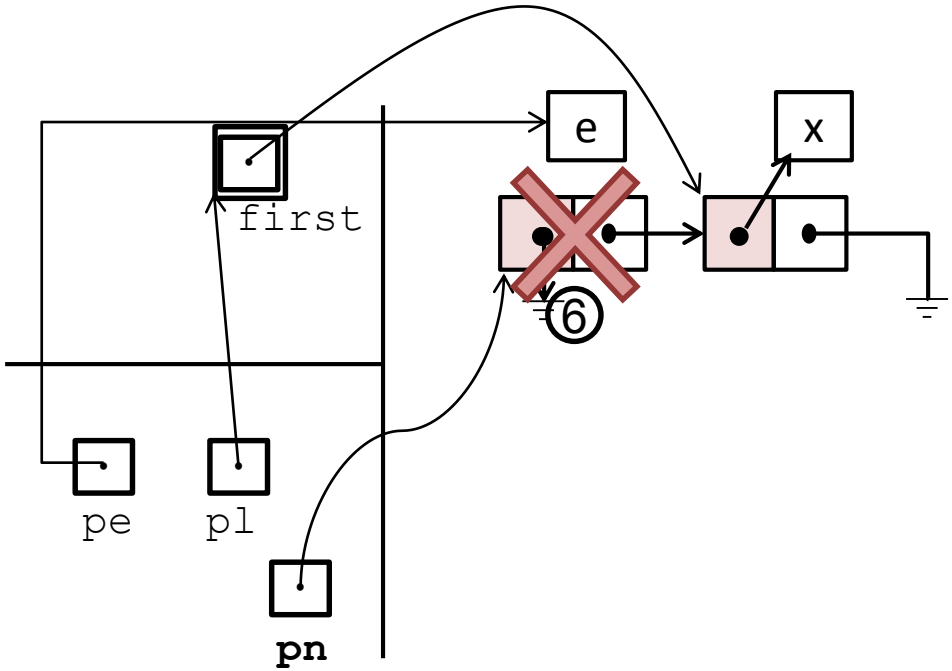
⑤ **first(pl) = next(pn);**



Implementación en C: list_extractIni

- Implementación

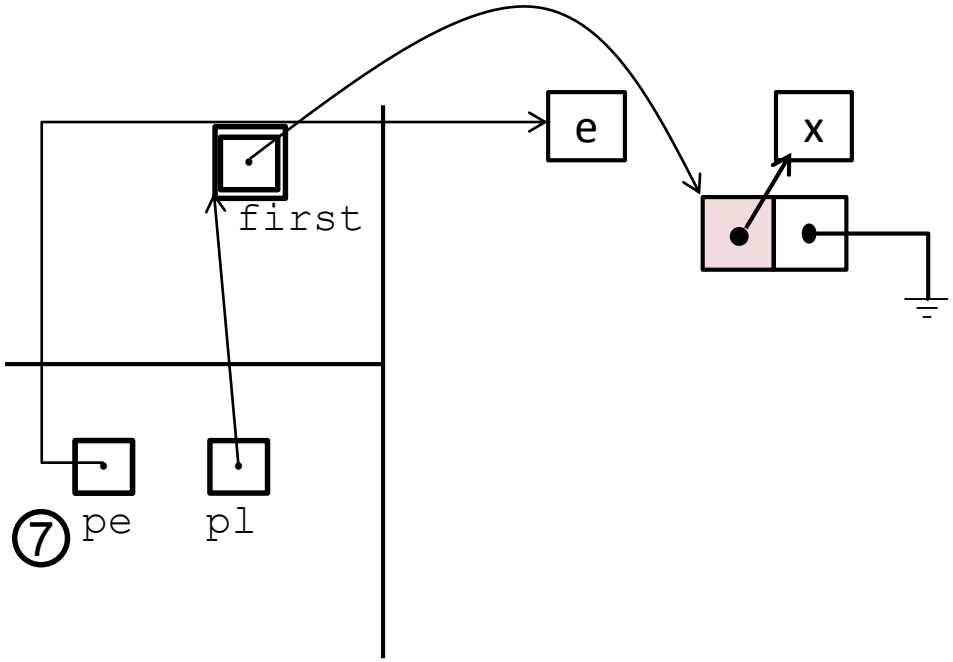
```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }

    pn = first(pl);
    pe = info(pn);
    info(pn) = NULL;   //Importante!!!



    first(pl) = next(pn);

⑥  node_free(pn);
```



Implementación en C: list_extractIni

- Implementación

```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }

    pn = first(pl);
    pe = info(pn);
    info(pn) = NULL;



    first(pl) = next(pn);

    node_free(pn);

⑦  return pe;
}
```



Implementación en C: list_extractIni

- Implementación (sin macros)

```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }

    pn = pl->first;
    pe = pn->info;
    pn->info = NULL;



    pl->first = pn->next;

    node_free(pn);

    return pe;
}
```

Implementación en C: list_extractIni

- Implementación (sin macros)

```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }

    pn = pl->first;
    pe = pn->info;
    pn->info = NULL;


    pl->first = pn->next;

    node_free(pn);

    return pe;
}
```

¿Hace falta llamar a node_free, que libera el nodo completo (incluyendo la memoria apuntada por el campo info, en caso de que no sea NULL)?

Como hemos puesto pn->info=NULL, ya no hace falta liberar el campo info → ¿no sería más eficiente liberar solo la estructura del nodo usando free, en vez de llamar a node_free, que va a llamar a element_free sin necesidad, pues ya no hay ninguna info que liberar?

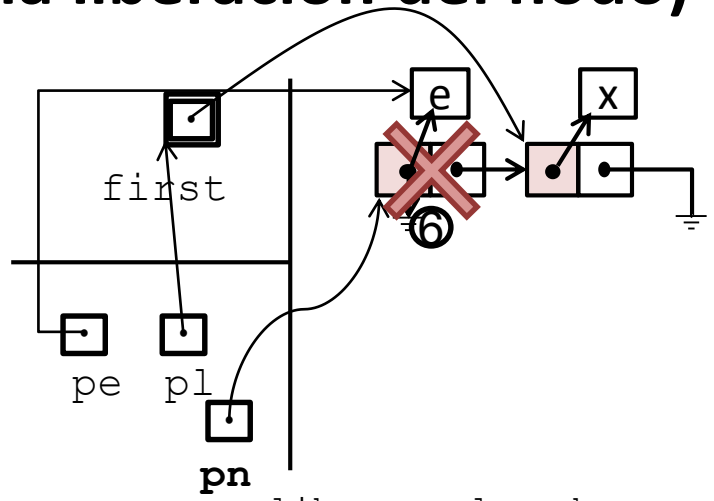
Si liberamos el nodo con free, ¿hace alta poner el campo info a NULL, o se puede dejar apuntando al elemento, sabiendo que con free solo se va a liberar el nodo y no va a afectar a la memoria apuntada por info (no se va a llamar a ninguna función que libere esa info)?

Solución: diapositiva siguiente

Implementación en C: list_extractIni

- Implementación
  (alternativa más eficiente para la liberación del nodo)

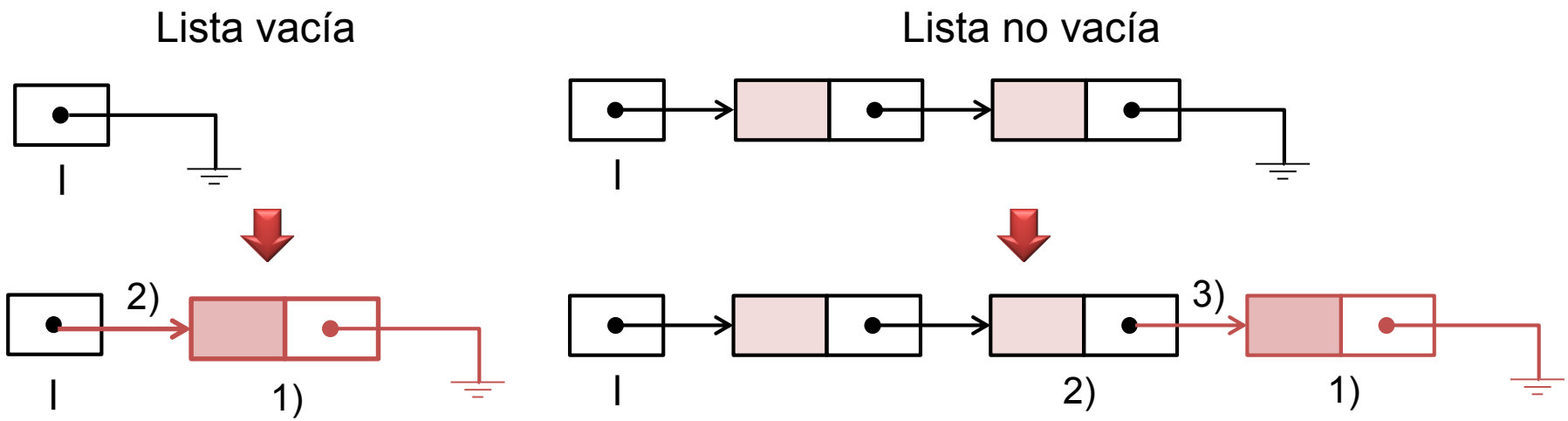
```
Element *list_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

    if (!pl || list_isEmpty(pl) == TRUE) {
        return NULL;
    }

    pn = pl->first;
    pe = pn->info;
    pn->info = NULL;   /* No hace falta, porque no vamos a liberar el nodo con
                          node_free, sino que liberaremos solo la estructura de
                          nodo con free, sin liberar el lugar adonde apunta el campo
                          info (que quedaría apuntado por pe, para devolverlo)*/

    pl->first = pn->next;

    node_free(pn); /* Ya no liberamos el nodo y la memoria a la que apuntan sus
    free(pn);          campos (elemento apuntado por info), sino únicamente la
    return pe;         memoria ocupada por la estructura Node */
}
```



Lo haremos así en todas las funciones de extracción (más eficiente)

Implementación en C: list_insertEnd

- **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: list_insertEnd

- ¿Implementación?

```
Status list_insertEnd(List *pl, Element *pe) {
```

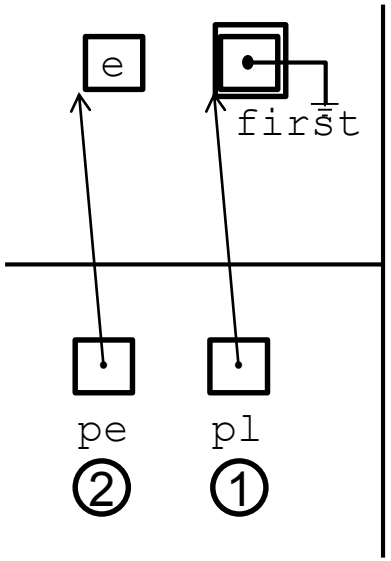
2 casos
1) Lista vacía
2) Lista no vacía

Implementación en C: list_insertEnd

- Implementación

① ②

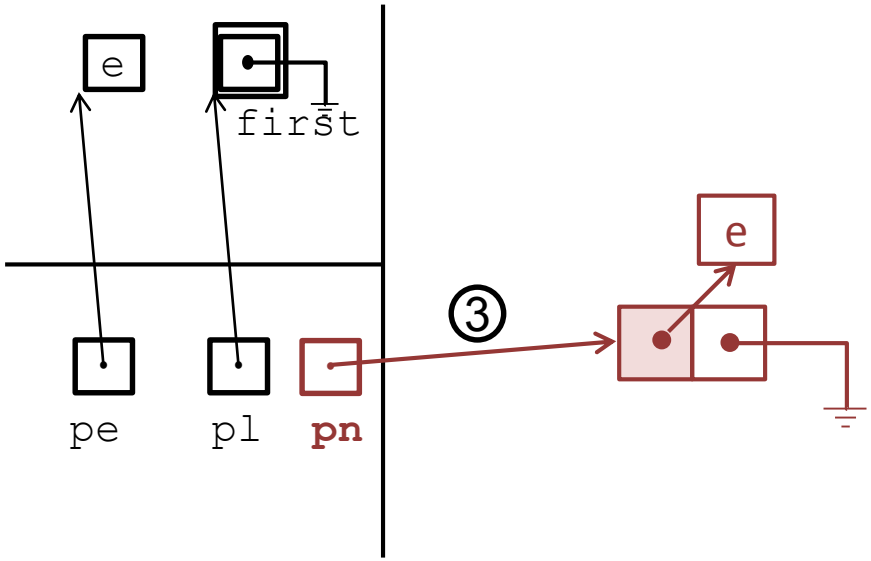
```
Status list_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL, *qn = NULL;

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



Implementación en C: list_insertEnd

- Implementación

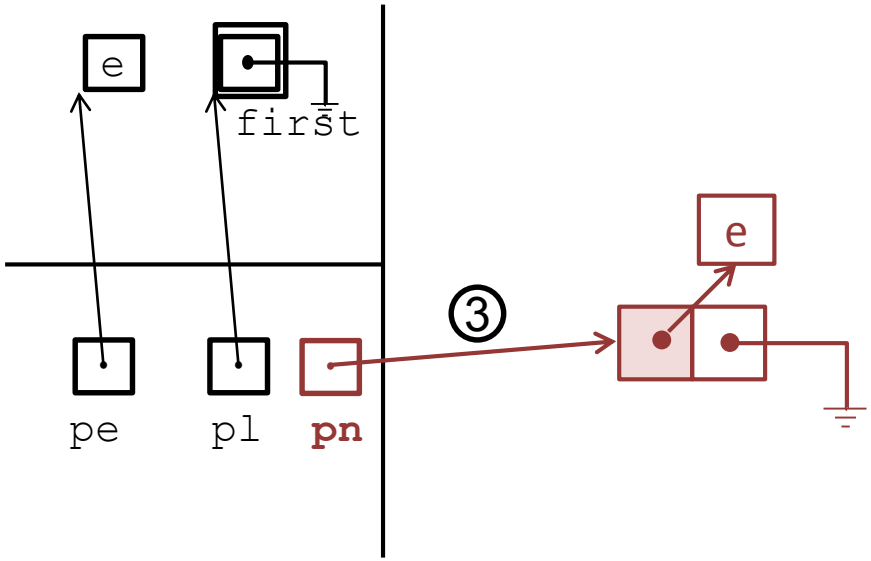
```
Status list_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL, *qn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
③   info(pn) = element_copy(pe);
    if (!info(pn)) {
        node_free(pn);
        return ERROR;
    }
```



Implementación en C: list_insertEnd

- Implementación

```
Status list_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL, *qn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
    info(pn) = element_copy(pe);
    if (!info(pn)) {
        node_free(pn);
        return ERROR;
    }
```

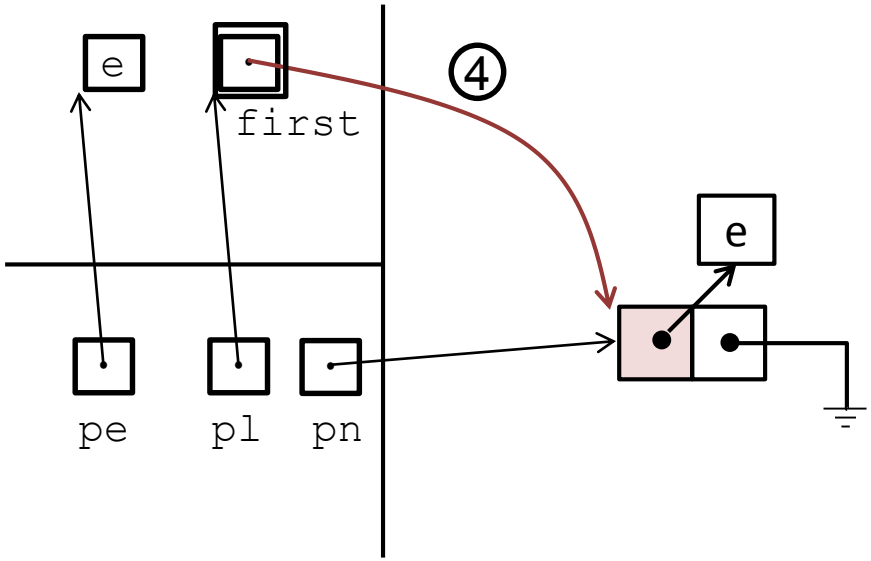


2 casos
1) Lista vacía
2) Lista no vacía

Implementación en C: list_insertEnd

- Implementación (dibujos caso lista vacía)

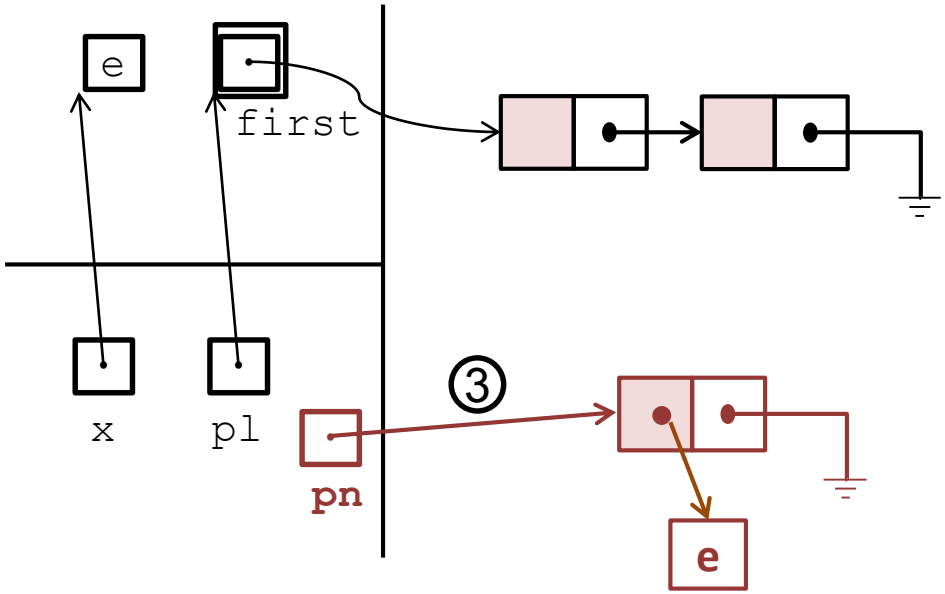
```
Status list_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL, *qn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
    info(pn) = element_copy(pe);
    if (!info(pn)) {
        node_free(pn);
        return ERROR;
    }

    if (list_isEmpty(pl) == TRUE) {
④       first(pl) = pn;
        return OK;
    }
```



Implementación en C: list_insertEnd

- Implementación (lista no vacía, tras crear el nodo…)

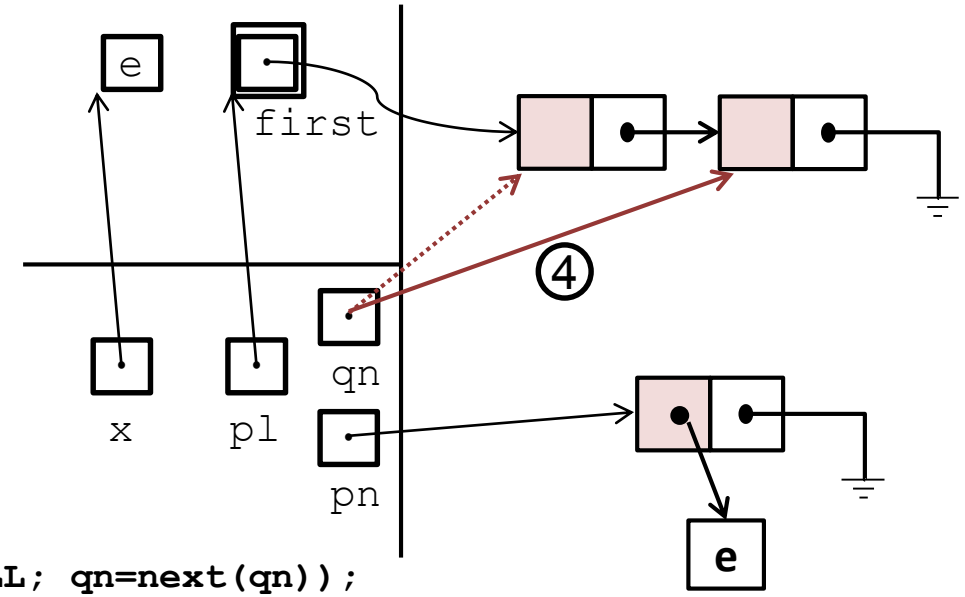
```
Status list_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL, *qn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
③   info(pn) = element_copy(pe);
    if (!info(pn)) {
        node_free(pn);
        return ERROR;
    }

    if (list_isEmpty(pl) == TRUE) {
        first(pl) = pn;
        return OK;
    }
```



Implementación en C: list_insertEnd

- Implementación (lista no vacía)

```
Status list_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL, *qn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
    info(pn) = element_copy(pe);
    if (!info(pn)) {
        node_free(pn);
        return ERROR;
    }

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

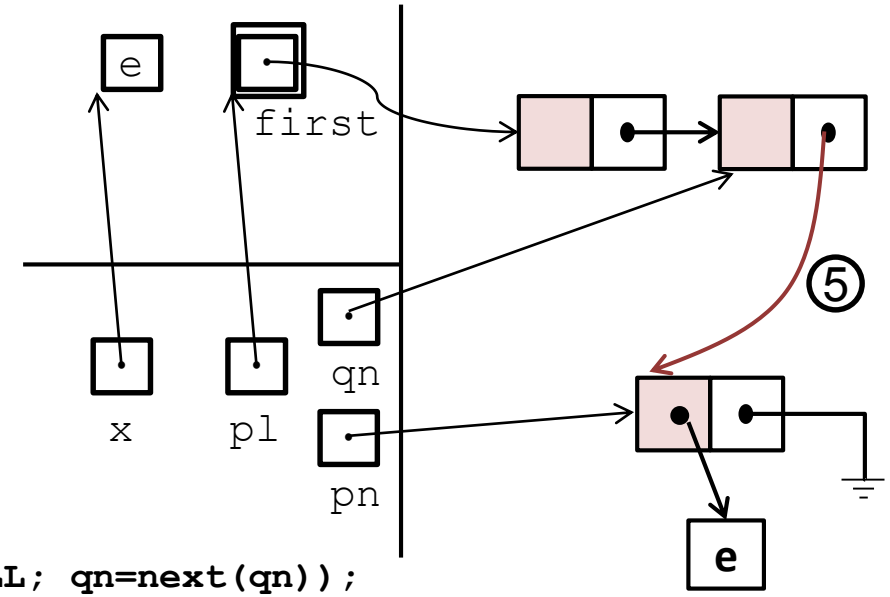



```
// Avanzar, con bucle while, sería:
qn = first(pl);
while (next(qn)!=NULL) {
    qn = next(qn);
}
```

Implementación en C: list_insertEnd

- Implementación (lista no vacía)

```
Status list_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL, *qn = NULL;

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

    pn = node_create();
    if (!pn) {
        return ERROR;
    }
    info(pn) = element_copy(pe);
    if (!info(pn)) {
        node_free(pn);
        return ERROR;
    }

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



Implementación en C: list_insertEnd

- Implementación (**sin macros**)

```
Status list_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL, *qn = NULL;

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

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

    if (list_isEmpty(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: list_extractEnd

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

  Si lista vacía, devolver NULL

  Si se tiene exactamente un nodo:

  1) Obtener info de ese nodo
  2) Eliminar ese nodo y dejar lista vacía (first apunta a NULL)
  3) Devolver info

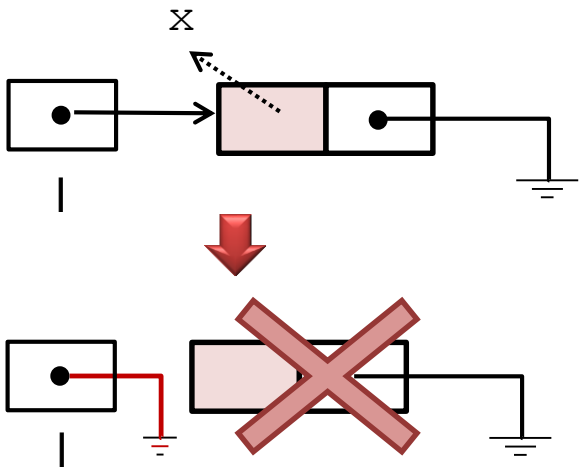
  Si se tiene más de un nodo:

  1) Recorrer la lista hasta situarse en el penúltimo nodo
  2) Obtener la info del último nodo (el siguiente al penúltimo)
  3) Eliminar el último nodo
  4) Hacer que el penúltimo nodo apunte a NULL
  5) Devolver info

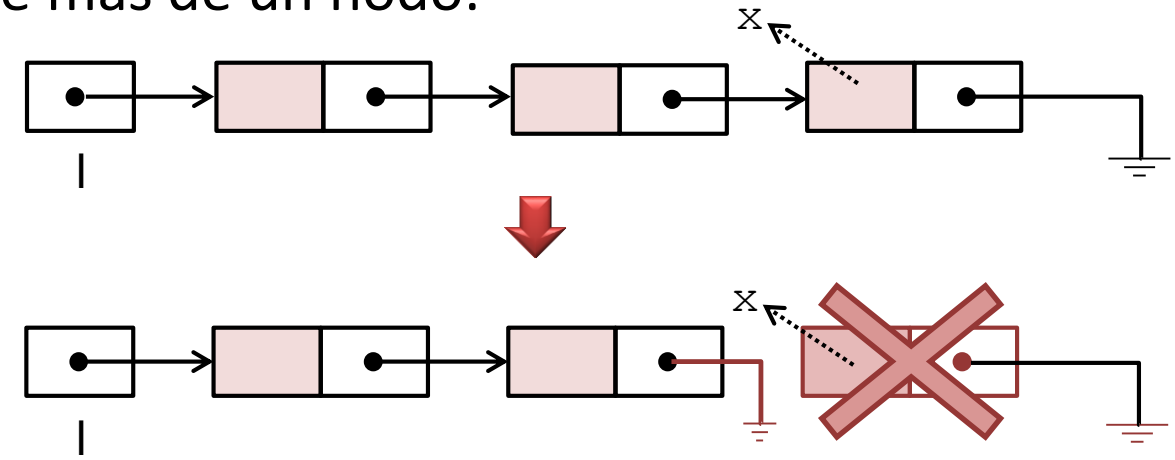
Implementación en C: list_extractEnd

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

Si se tiene un nodo:



Si se tiene más de un nodo:



Implementación en C: list_extractEnd

- ¿Implementación?

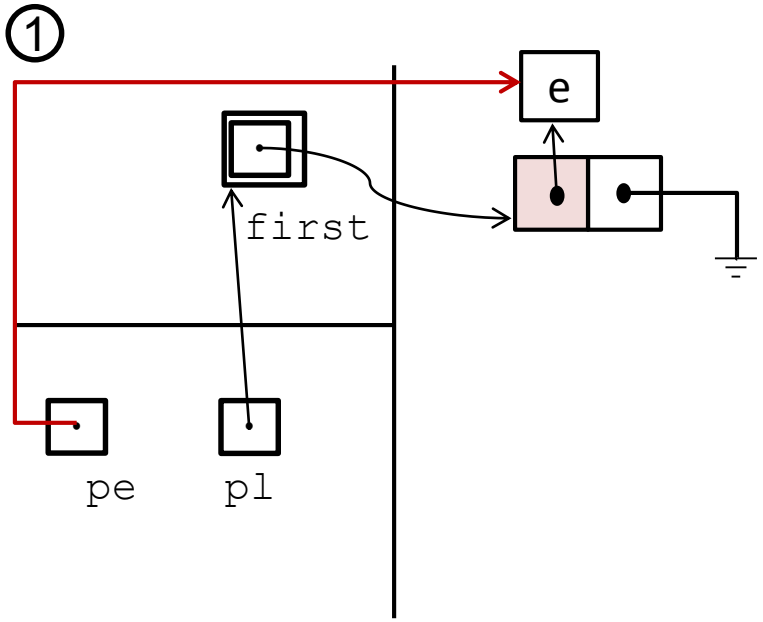
```
Element *list_extractEnd(List *pl) {
```

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

Implementación en C: list_extractEnd

- Implementación (lista de un nodo)

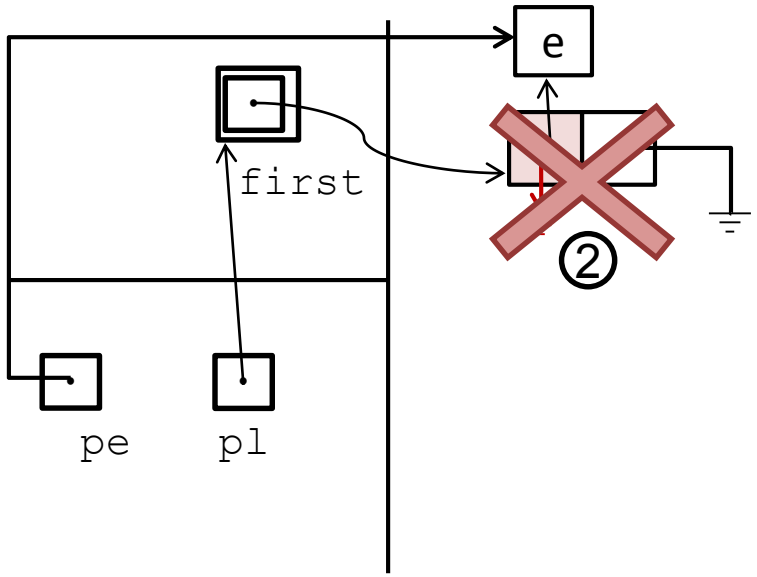
```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

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

    // Caso: 1 nodo
    if (!next(first(pl)) {
①       pe = info(first(pl));
        free(first(pl));  //Libera solo el nodo, no su info (no es nodefree)
        first(pl) = NULL;
        return pe;
    }
```



Implementación en C: list_extractEnd

- Implementación (lista de un nodo)

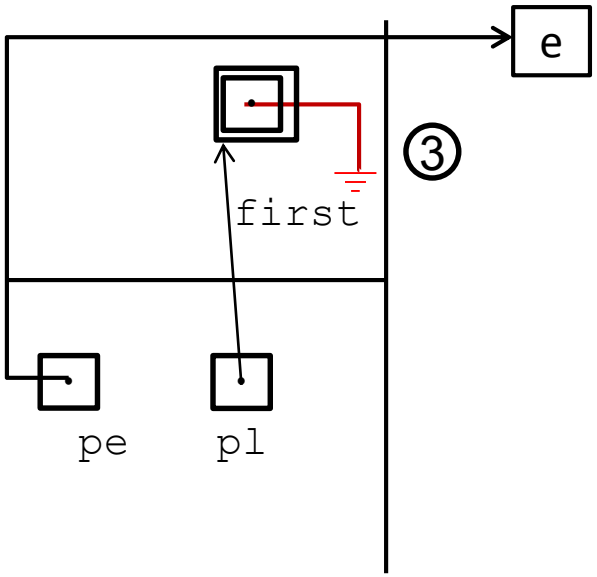
```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

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

    // Caso: 1 nodo
    if (!next(first(pl)) {
        pe = info(first(pl));
②       free(first(pl));  //Libera solo el nodo, no su info (no es nodefree)
        first(pl) = NULL;
        return pe;
    }
```



Implementación en C: list_extractEnd

- Implementación (lista de un nodo)

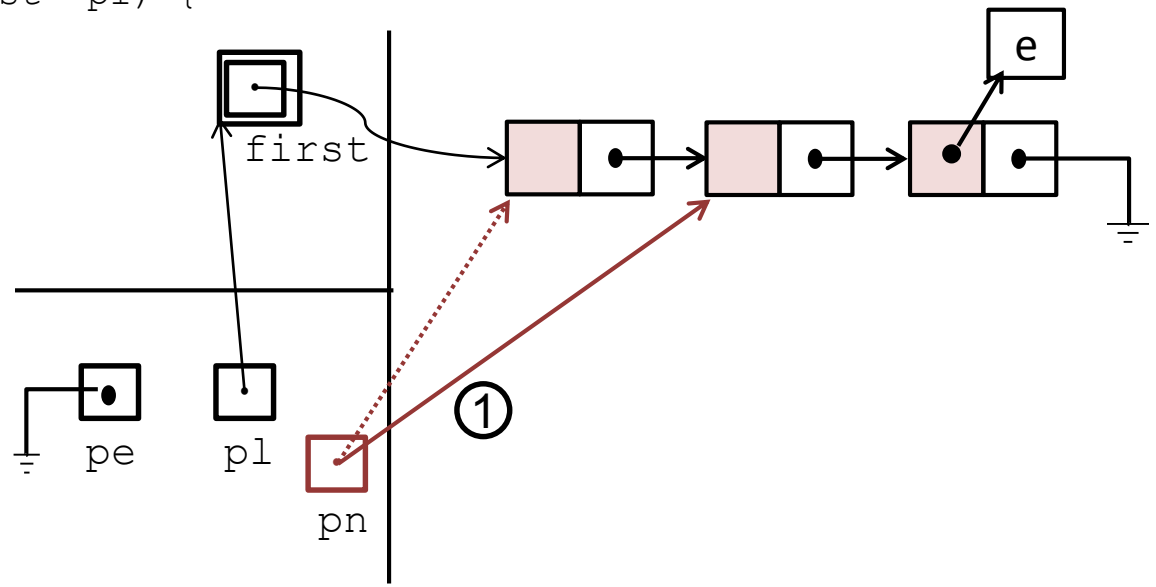
```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

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

    // Caso: 1 nodo
    if (!next(first(pl)) {
        pe = info(first(pl));
        free(first(pl));  //Libera solo el nodo, no su info (no es nodefree)
③       first(pl) = NULL;
        return pe;
    }
```



Implementación en C: list_extractEnd

- Implementación (lista de varios nodos)

```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;
```

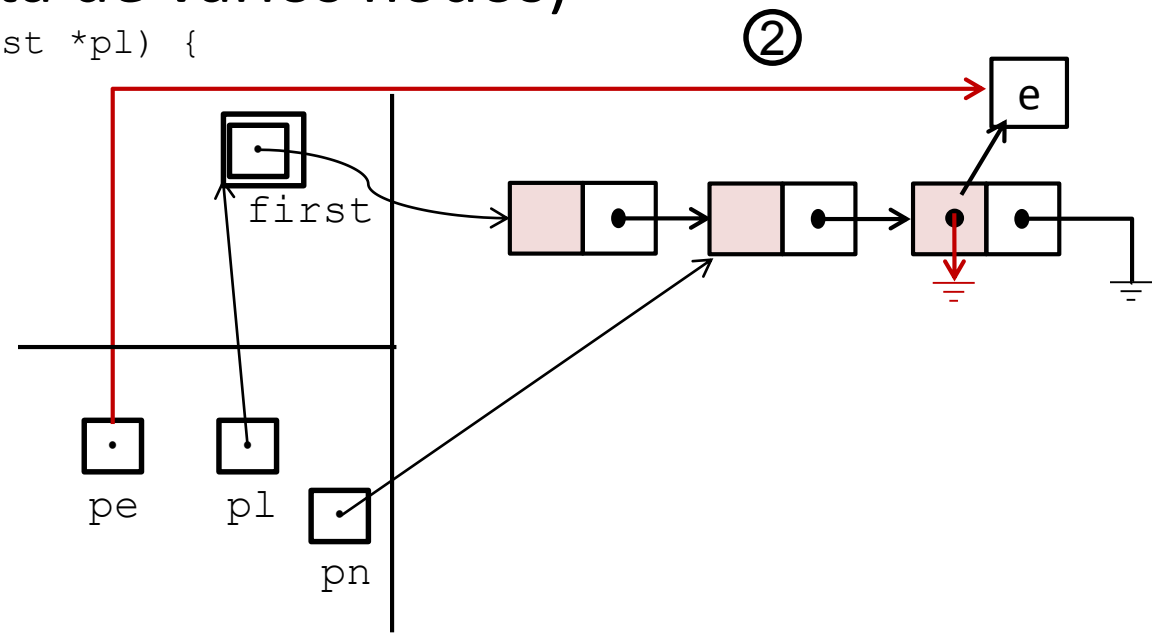



```
    // Caso: 2 o más nodos
    // -> se sitúa pn en el penúltimo nodo de la lista
1   for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
    pe = info(next(pn));
    free(next(pn)); //Libera solo el nodo, no su info (no es nodefree)
    next(pn) = NULL;
    return pe;
}
```

Implementación en C: list_extractEnd

- Implementación (lista de varios nodos)

```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;
```

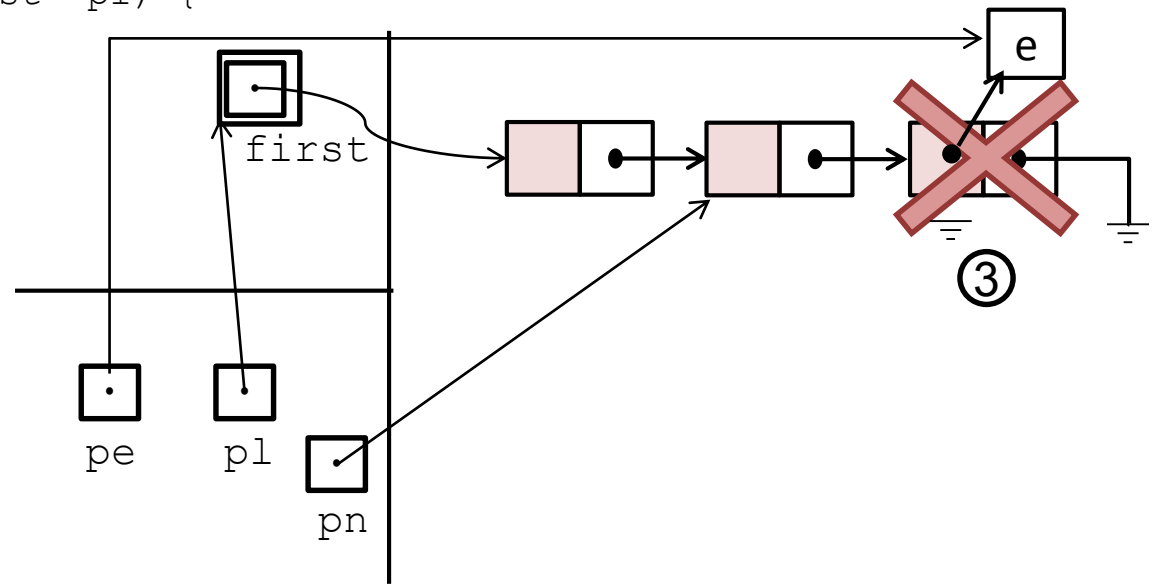



```
    // Caso: 2 o más nodos
    // -> se sitúa pn en el penúltimo nodo de la lista
    for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
②   pe = info(next(pn));
    free(next(pn)); //Libera solo el nodo, no su info (no es nodefree)
    next(pn) = NULL;
    return pe;
}
```

Implementación en C: list_extractEnd

- Implementación (lista de varios nodos)

```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;
```

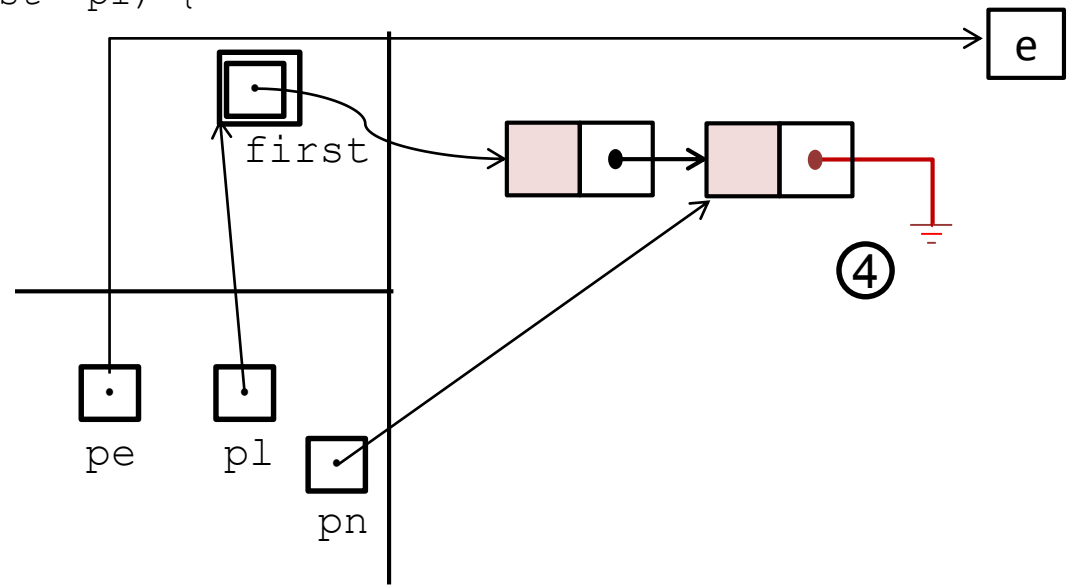



```
    // Caso: 2 o más nodos
    // -> se sitúa pn en el penúltimo nodo de la lista
    for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
    pe = info(next(pn));
③   free(next(pn)); //Libera solo el nodo, no su info (no es nodefree)
    next(pn) = NULL;
    return pe;
}
```

Implementación en C: list_extractEnd

- Implementación (lista de varios nodos)

```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;
```




```
    // Caso: 2 o más nodos
    // -> se sitúa pn en el penúltimo nodo de la lista
    for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
    pe = info(next(pn));
    free(next(pn)); //Libera solo el nodo, no su info (no es nodefree)
④   next(pn) = NULL;
    return pe;
}
```

Implementación en C: list_extractEnd

- Implementación

```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

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

    // Caso: 1 nodo
    if (!next(first(pl)) {
        pe = info(first(pl)));
        free(first(pl)); //Libera solo el nodo, no su info (no es nodefree)
        first(pl) = NULL;
        return pe;
    }

    // Caso: 2 o más nodos
    // -> se sitúa pn en el penúltimo nodo de la lista
    for (pn=first(pl); next(next(pn))!=NULL; pn=next(pn))
        ;
    pe = info(next(pn));
    free(next(pn)); //Libera solo el nodo, no su info (no es nodefree)
    next(pn) = NULL;
    return pe;
}
```

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

Implementación en C: list_extractEnd

- Implementación (sin macros)

```
Element *list_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

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

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

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

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

Implementación en C: list_free

- Implementar la función **list_free**

```
void list_free(List *pl)
```

Implementación en C: list_free

1. Implementación usando primitivas

```
void list_free(List *pl) {
    if (!pl) return;

    while (list_isEmpty(pl) == FALSE)
        element_free(list_extractIni(pl));

    free(pl);
}
```

2. Implementación accediendo a la estructura

```
void list_free(List *pl) {
    Node *pn = NULL;

    if (!pl) return;

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

Implementación en C: list_free

3. Implementación recursiva

```
void list_free(List *pl) {
    if (!pl)
        return;

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

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

void list_free_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
    list_free_rec(next(pn));

    // Liberación del nodo actual
    node_free(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

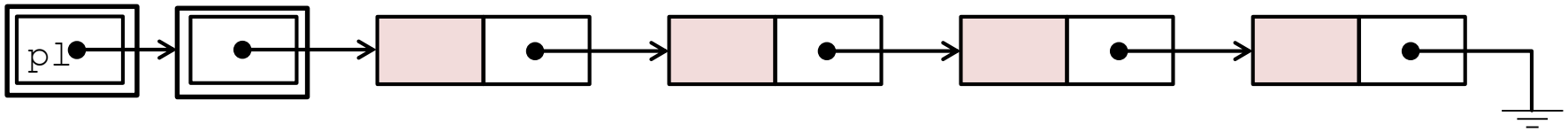
Pila. Implementación con una lista enlazada

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

```
// En stack.c
struct _Stack {
 Element * datos[STACK_MAX];
 int       top;
};
// En stack.h
typedef struct _Stack Stack;
```

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

```
// En stack.c
struct _Stack {
   List *pl;
};
// En stack.h
typedef struct _Stack Stack;
```



Pila. Implementación con una lista enlazada

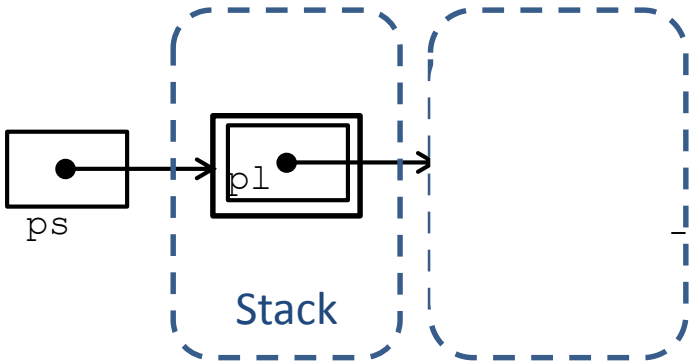
- Implementación de las primitivas de Pila sobre la EdD basada en lista enlazada

```
Stack *stack_init();
void stack_free(Stack *ps);
Boolean stack_isEmpty(const Stack *ps); → list_isEmpty
Boolean stack_isFull(const Stack *ps);
Status stack_push(Stack *ps, const Element *pe);→ list_insertIni
Element *stack_pop(Stack *ps); → list_extractIni
Element *stack_top(Stack *ps); → lista_getIni (nueva)
```

Pila. Implementación con una lista enlazada


```
Stack *stack_init() {
   Stack *ps = NULL;
   ps = (Stack *) malloc(sizeof(Stack));
   if (!ps)
      return NULL;

   ps->pl = list_init();
   if (!ps->pl) {
      free(ps);
      return NULL;
   }
   return ps;
}

void stack_free(Stack *ps) {
   if (ps) {
      list_free(ps->pl);
      free(ps);
   }
}
```

```
struct _Stack {
   List *pl;
};
```



Pila. Implementación con una lista enlazada


```
struct _Stack {
   List *pl;
};
```

```
Boolean stack_isEmpty(Stack *ps) {
   if (!ps)
      return TRUE;                  // Caso de error
   return list_isEmpty(ps->pl); // Caso normal
}

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

Pila. Implementación con una lista enlazada


```
Status stack_push(Stack *ps, Element *pe) {
   if (!ps || !pe)
      return ERROR;

   return list_insertIni(ps->pl, pe);
}
```

```
struct _Stack {
   List *pl;
};
```

```
Element *stack_pop(Stack *ps) {
   if (!ps)
      return ERROR;

   return list_extractIni(ps->pl);
}
```

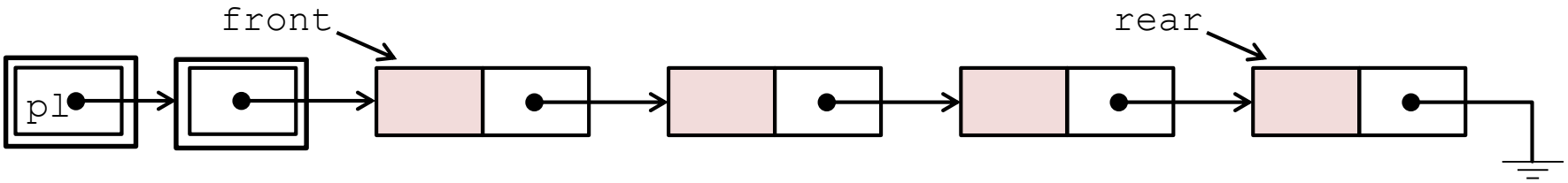
Cola. Implementación con una lista enlazada

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

```
// En cola.c
struct _Queue {
   Element *datos[COLA_MAX];
   int front, rear;
};
// En cola.h
typedef struct _Queue Queue;
```

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

```
// En cola.c
struct _Queue {
   List *pl;
};
// En cola.h
typedef struct _Queue Queue;
```



Cola. Implementación con una lista enlazada

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

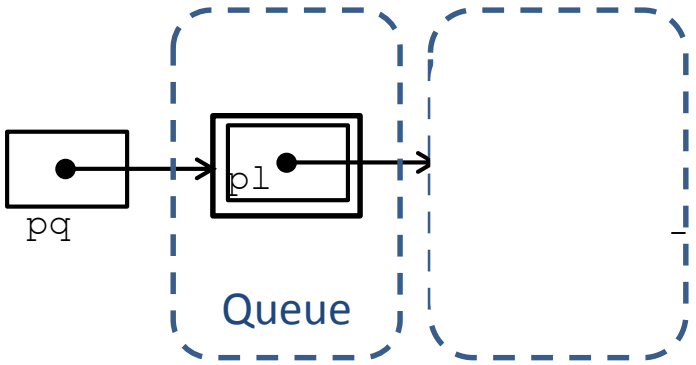
```
Queue *cola_crear()
void cola_liberar(Queue *pq)
Boolean cola_vacia(Queue *pq)
Boolean cola_llena(Queue *pq)
Status cola_insertar(Queue *pq, Element *pe) → list_insertEnd
Element *cola_extraer(Queue *pq) → list_extractIni
```

```
struct _Queue
{
   List *pl;
};
```

Cola. Implementación con una lista enlazada


```
Queue *queue_init() {
   Queue *pq = NULL;
   pq = (Queue *) malloc(sizeof(Queue));
   if (!pq)
      return NULL;

   pq->pl = list_init();
   if (!pq->pl) {
      free(pq);
      return NULL;
   }
   return pq;
}

void queue_free(Queue *pq) {
   if (pq) {
      list_free(pq->pl);
      free(pq);
   }
}
```

```
struct _Queue {
   List *pl;
};
```



Cola. Implementación con una lista enlazada


```
Boolean queue_isEmpty(Queue *pq) {
   if (!pq)
      return TRUE;      // Caso de error

   return list_isEmpty(pq->pl); // Caso normal
}


Boolean queue_isFull(Queue *pq) {
   if (!pq)
      return TRUE;      // Caso de error

   return FALSE;        // Caso normal: la cola nunca está llena
}
```

Cola. Implementación con una lista enlazada


```
Status queue_insert(Queue *pq, Element *pe) {
   if (!pq || !pe)
      return ERROR;

   return list_insertEnd(pq->pl, pe);
}
```

```
Element *queue_extract(Queue *pq) {
   if (!pq)
      return NULL;

   return list_extractIni(pq->pl);
}
```

Colas. Implementación con listas enlazadas

- Inconveniente: la función de **inserción en listas enlazadas simples es ineficiente**, pues recorre toda la lista para insertar un elemento al final de la misma

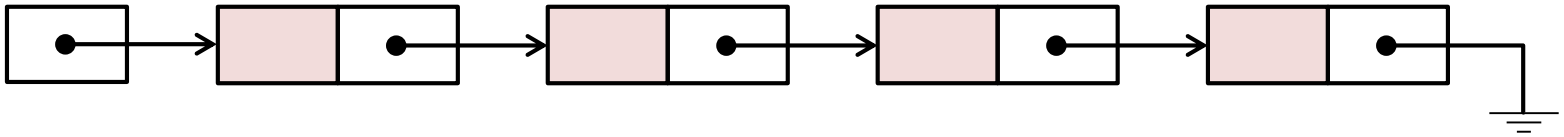
→ queueInsert implementada sobre lista enlazada simple es ineficiente

Contenidos

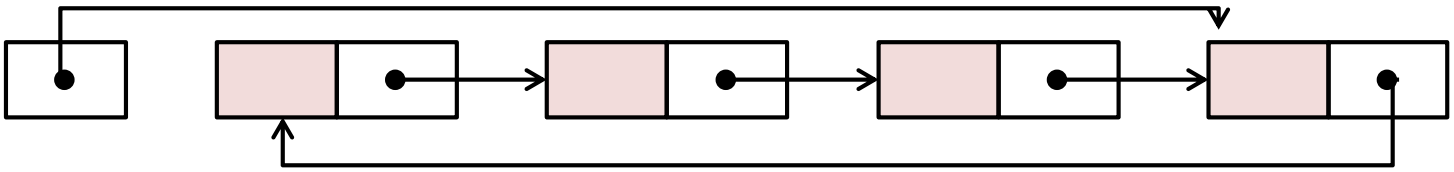
- El TAD Lista
- Estructura de datos de Lista
- Implementación en C de Lista
- Implementación de Pila y Cola con Lista
- **Tipos de Listas**

Tipos de listas enlazadas

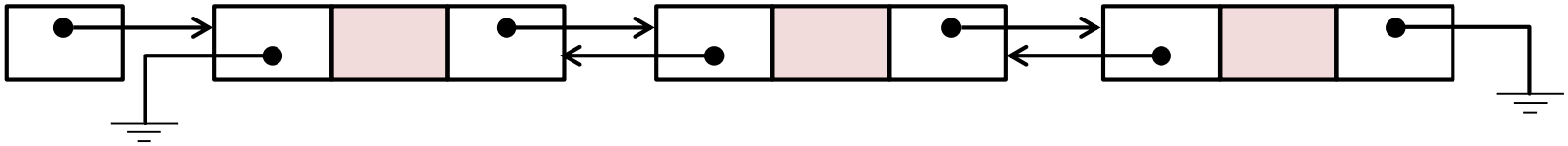
- Lista enlazada (simple)



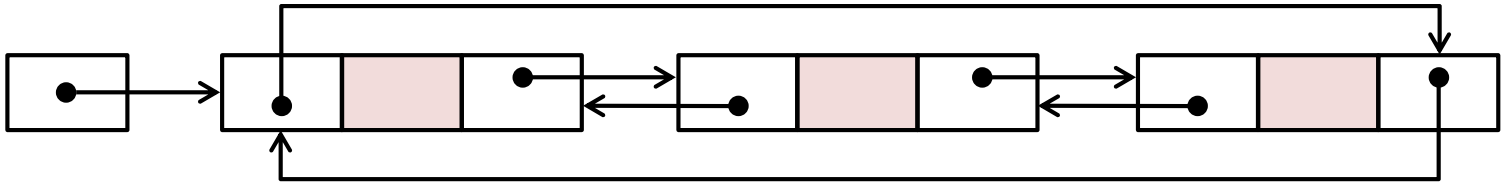
- Lista enlazada circular



- Lista doblemente enlazada

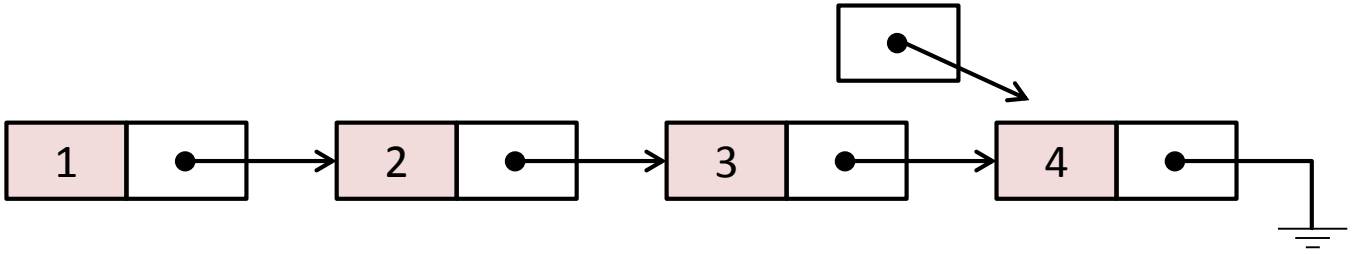


- Lista doblemente enlazada circular



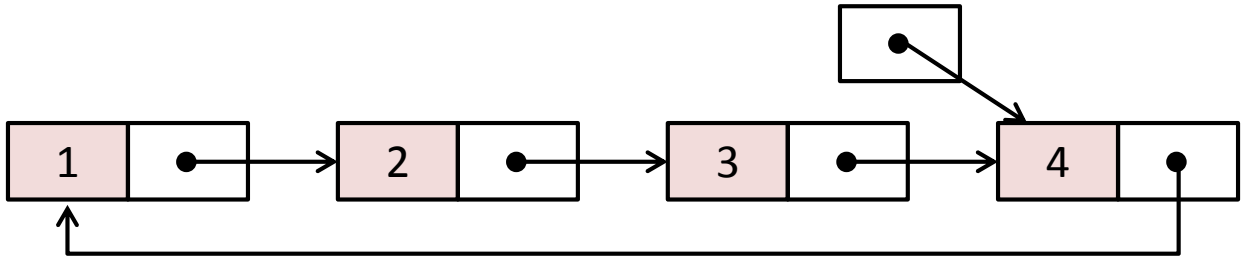
Listas enlazadas

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



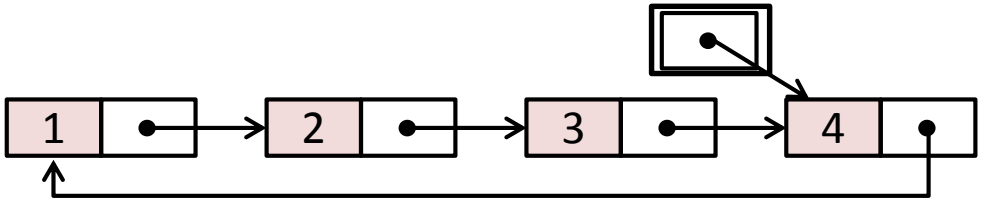
Pero, ¿cómo acceder al primer nodo?

Haciendo que el último nodo apunte al primero



Lista enlazada circular

- **Lista enlazada circular (LEC)**. 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



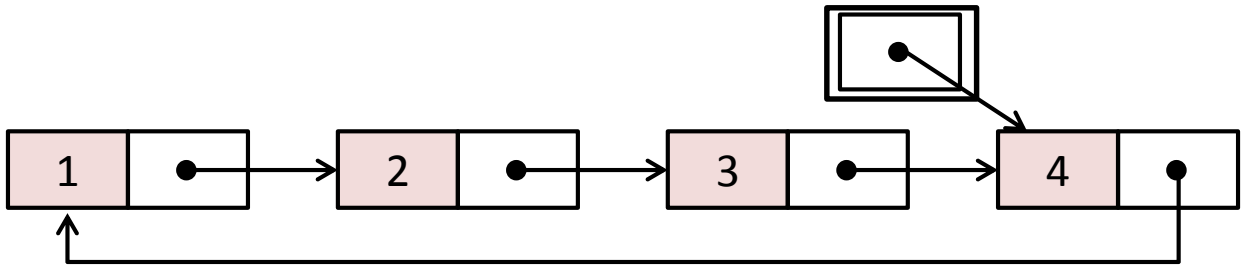
- Una utilidad: implementar **Queue sobre LEC**

```
struct _Queue {
   LEC *pl;
};
typedef struct _Queue Queue;

Status queue_insert(Queue *pq, Element *pe) {
  if (!pq || !pe) {
     return ERROR;
  }
  return LEC_insertEnd(pq->pl, pe);
}
```

Se extraería del inicio, que en LEC también es eficiente

Lista enlazada circular

- **Lista enlazada circular (LEC).**

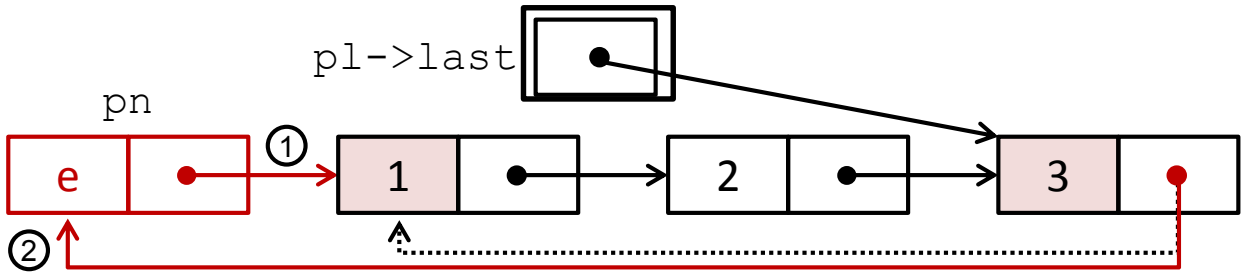



```
// EdD de LEC análoga a la de List
struct _LEC {
   Node *last;
};

typedef struct _LEC LEC;
```

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

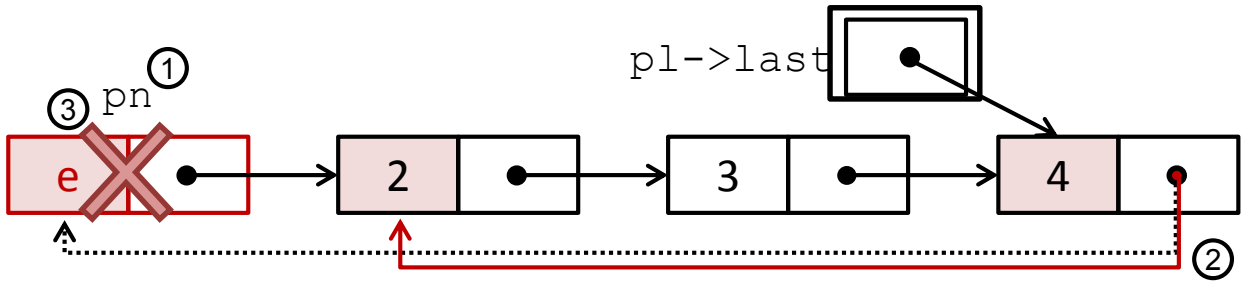
Lista circular: insertar en inicio


```
Status LEC_insertIni(List *pl, Element *pe) {
    Node *pn = NULL;

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

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

    pn->info = element_copy(pe);
    if (!info(pn)) {
        node_free(pn);
        return ERROR;
    }
    // Caso 1: lista vacía
    if (list_isEmpty(pl) == TRUE) {
        pn->next = pn;                     // El next de pn apunta a sí mismo
        pl->last = pn;
    }
    // Caso 2: lista no vacía
    else {
      (1) pn->next = pl->last->next;       // El next de pn apunta al primer nodo
      (2) pl->last->next = pn;             // El next del último nodo apunta a pn
    }
    return OK;
}
```



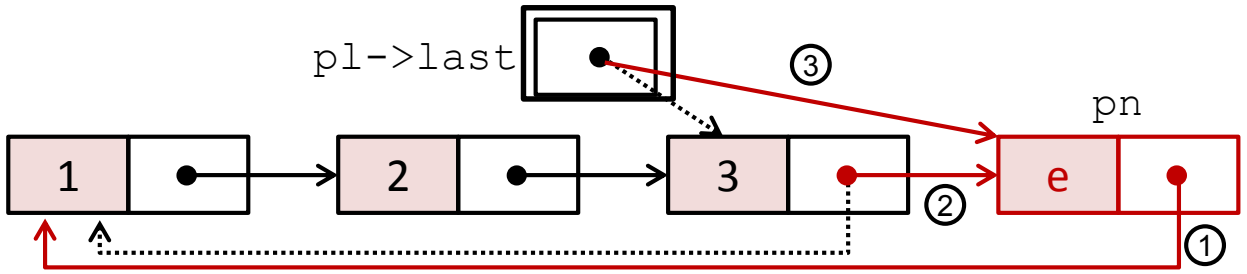
Lista circular: extraer de inicio


```
Element *LEC_extractIni(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

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

    pe = pl->last->next->info;    // Se extrae el elemento del primer nodo



    // Caso 1: lista de un nodo
    if (pl->last->next == pl->last) {
        free(pl->last);                  // Se libera pn (la estructura Node, no libera info)
        pl->last = NULL;                 // Se deja la lista vacía
    }
    // Caso 2: lista de varios nodos
    else {
      (1) pn = pl->last->next;           // Se sitúa pn en el primer nodo
      (2) pl->last->next = pn->next;     // El next del último nodo apunta al segundo nodo
      (3) free(pn);                      // Se libera pn (la estructura Node, no libera info)
    }
    return pe;
}
```



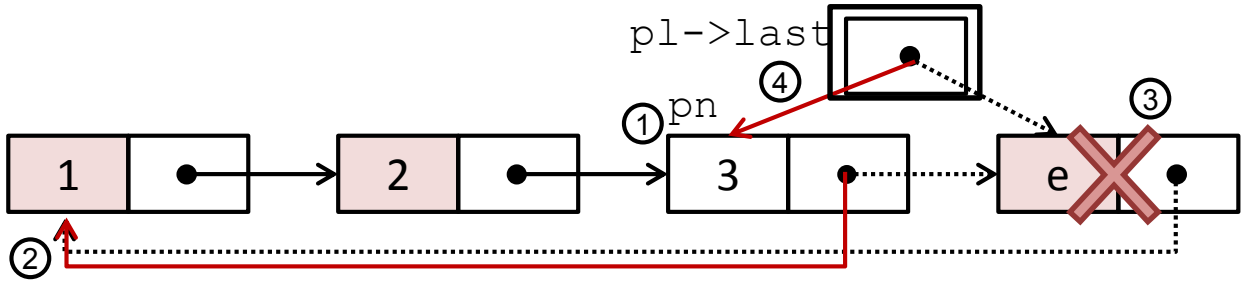
Lista circular: insertar al final


```
Status LEC_insertEnd(List *pl, Element *pe) {
    Node *pn = NULL;

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

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

    pn->info = element_copy(pe);
    if (pn->info == NULL) {
        node_free(pn);
        return ERROR;
    }
    // Caso 1: lista vacía
    if (list_isEmpty(pl) == TRUE) {
        pn->next = pn;                     // El next de pn apunta al propio nodo
        pl->last = pn;
    }
    // Caso 2: lista no vacía
    else {
      (1) pn->next = pl->last->next;       // El next de pn apunta al primer nodo
      (2) pl->last->next= pn;              // El next del que era el último nodo apunta a pn
      (3) pl->last = pn;                   // El nuevo último Element es pn
    }
    return OK;
}
```



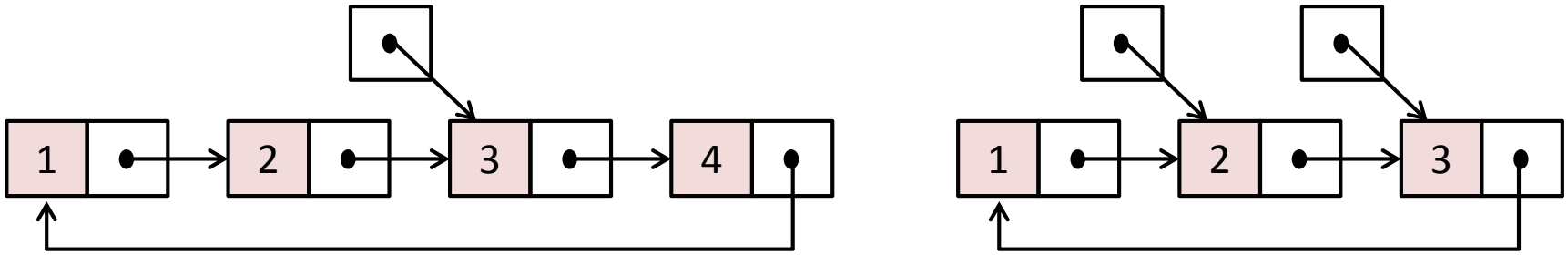
Lista circular: extraer del final


```
Element *LEC_extractEnd(List *pl) {
    Node *pn = NULL;
    Element *pe = NULL;

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

    pe = pl->last->info;    // Se coge el elemento del último nodo


    // Caso 1: lista de un nodo
    if (pl->last->next == pl->last) {
        free(pl->last);                 // Se libera pn (la estructura Node, no libera info)
        pl->last = NULL;                // Se deja la lista vacía
        return pe;

    }

    // Caso 2: lista de varios nodos
(1) for (pn=pl->last; pn->next!=pl->last; pn=pn->next)  // Se sitúa pn en el penúltimo nodo
        ;
(2) pn->next = pl->last->next; // El next de pn apunta al primer nodo
(3) free(pl->last);            // Se libera el último nodo (la estructura, no la info)
(4) pl->last = pn;             // El nuevo último nodo es pn
    return pe;
}
```



Lista enlazada circular

- **Ventajas**
  - Las primitivas `insertEnd`, `extractIni`, `insertIni` son eficientes, al no tener que recorrer la lista en general
  - No se usa más memoria que la que se usaba para List
- **Inconveniente**
  - La primitiva `extractEnd` tiene que recorrer la lista
  - **Apuntar al penúltimo nodo** de la lista en vez de al último no es una opción eficiente ya que tras la extracción habría que buscar cuál es el nuevo penúltimo recorriendo también la lista

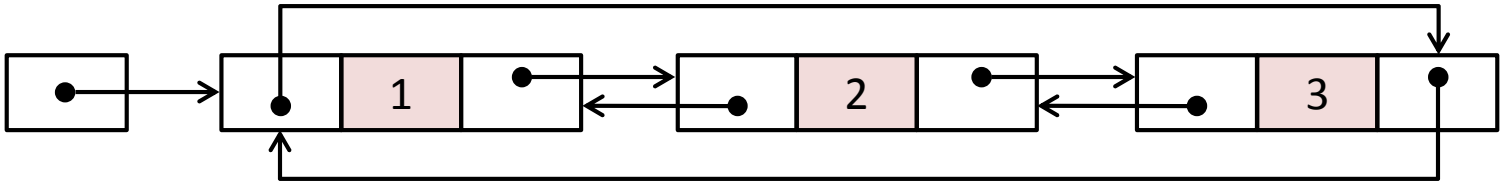


Colas: Implementación con lista circular

- **Ventajas**
  - Para implementar una cola, solo necesitamos las primitivas `LECinsertEnd` y `LECextractIni`, que sí son eficientes, al no tener que recorrer la lista
  - No se usa más memoria que la que se usaba para List
  - Tamaño arbitrariamente grande
- **Desventajas**
  - Una lista usa más memoria que un array (lleno), y en general no es memoria contigua.

Lista enlazada circular

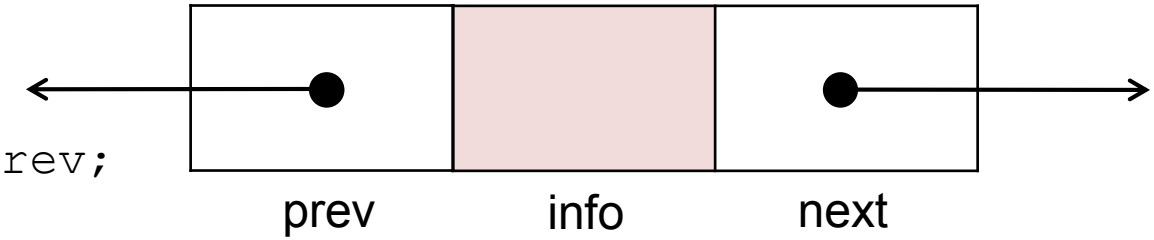
- Solución al problema de LECextractEnd: **lista doblemente enlazada circular**
  - Permite acceso inmediato a elementos anteriores y posteriores
  - Pero añade complejidad a todas las primitivas

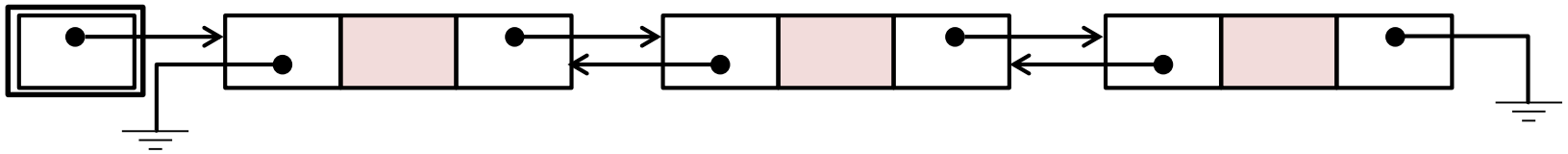


Lista doblemente enlazada

- EdD de Nodo y Lista doblemente enlazada

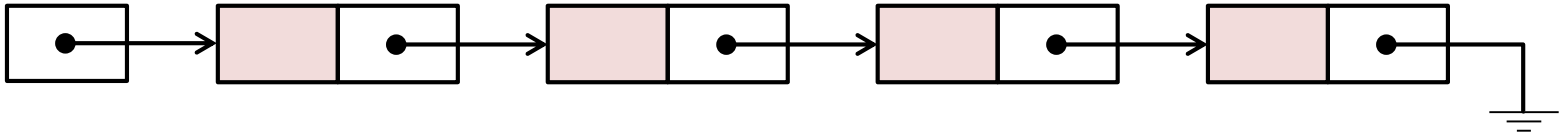
```
// Nodo
struct _NodeDE {
    struct _NodeDE *prev;
    Element *info;
    struct _NodeDE *next;
};
typedef struct _NodeDE NodeDE;

// Lista
struct _ListDE {
   NodeDE *first;
};
typedef struct _ListDE ListDE;
```



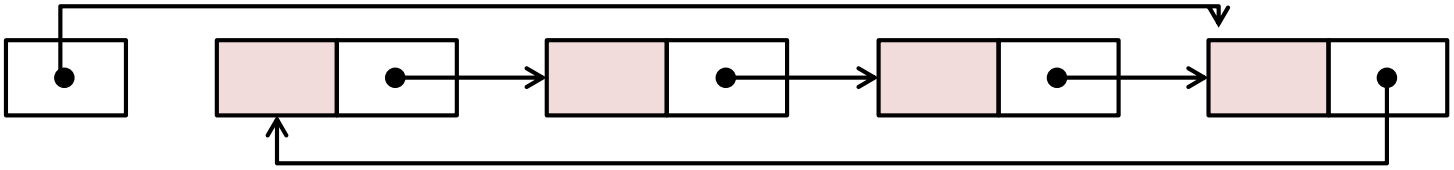


Tipos de listas enlazadas

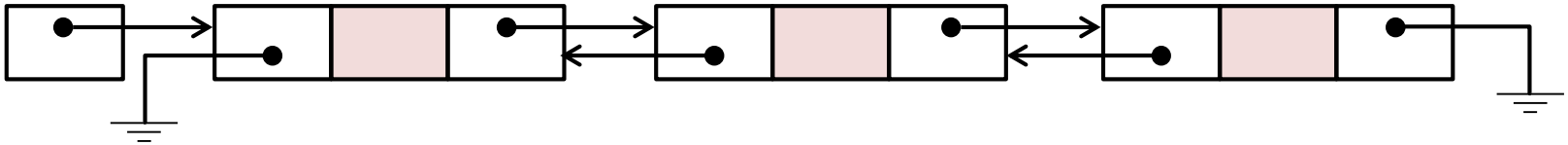
- Lista enlazada



- Lista enlazada circular



- Lista doblemente enlazada



- Lista doblemente enlazada circular