Listas, pilas y colas

Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

Operadores de estructuras

- Operador "*miembro de estructura*". Conecta el nombre de la estructura con el nombre del miembro

```
struct punto
{
  int x;
  int y;
};
struct punto p;
printf("%d, %d", p.x, p.y);
```

Operadores de estructuras

- En muchos casos, se utilizan punteros a estructuras

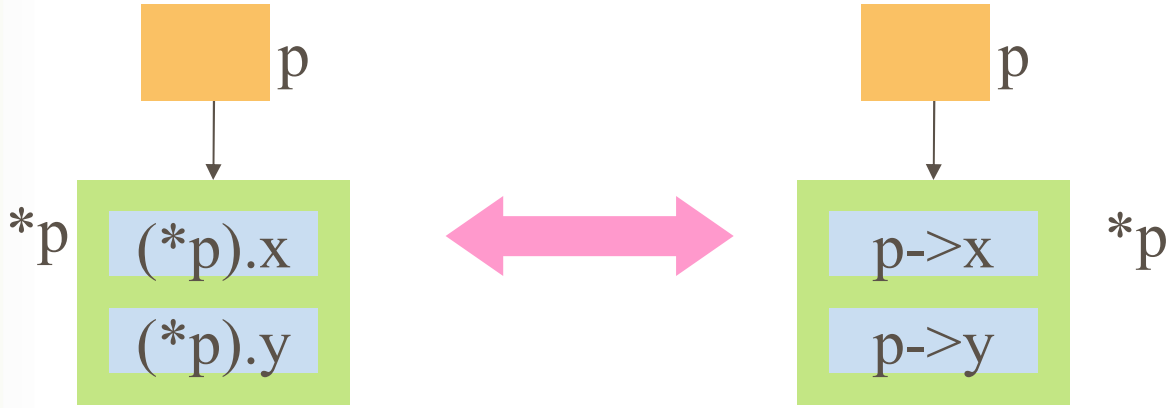
```
struct punto a, *p=&a;
printf("%d, %d", (*p).x, (*p).y);
```

- Los paréntesis son necesarios en "(*p).x" debido a que la precedencia del operador "." es mayor que la de "*"
- La expresión "*p.x" significa "*(p.x)", lo cual es ilegal ya que x no es un puntero

Operadores de estructuras

- Los punteros a estructuras son tan frecuentes que se ha proporcionado una notación alternativa como abreviación
- Si p es un puntero a estructura, entonces "*p->miembro de estructura*" se refiere al miembro en particular

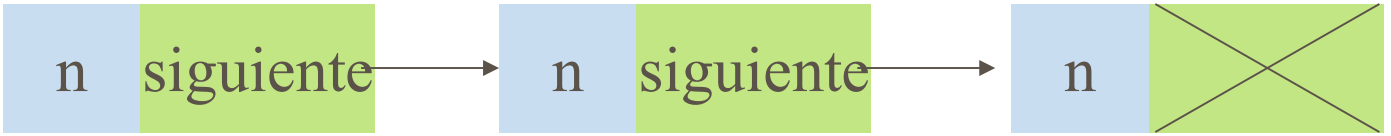
```
struct punto a  *p=&a;

printf("%d, %d", p->x, p->y);

// Equivale a printf("%d, %d", (*p).x, (*p).y);
```



Listas

- **Estructuras autoredefinidas**. Estructuras que se definen de forma recursiva
- Una lista de elementos es una sucesión finita de elementos del mismo tipo **unidas por punteros**

```
struct lista
{
 int n;
 struct lista *sig;
};
```



Listas

- **Vectores**: se almacenan consecutivos en memoria
  - Es posible acceder por índice V[i] ⇨ Acceso muy rápido
  - Inserciones y borrados costosos
- **Listas**: NO se almacenan consecutivas en memoria
  - No es posible acceder por índice
  - Inserciones y borrados rápidos
  - Espacio adicional para almacenar los punteros

Listas

- `struct lista *nuevoElemento();`
- `void insertarDelante(struct lista **cabeza, int n);`
- `void imprimirLista(struct lista *cabeza);`
- `void imprimirListaInverso(struct lista *elemento);`
- `int buscarElemento(struct lista *cabeza, int n);`
- `void insertarDetras(struct lista **cabeza, int n);`
- `void borrarElemento(struct lista **cabeza, int n);`
- `void borrarLista(struct lista **cabeza);`
- `void borrarElementoRecursivo(struct lista **cabeza, int n);`
- `void borrarListaRecursiva(struct lista **elemento);`
- `void insertarOrden(struct lista **cabeza, int n);`
- `void insertarOrdenRecursivo(struct lista **cabeza, int n);`
- `void ordenarLista(struct lista *cabeza)`

Listas. En la función main()

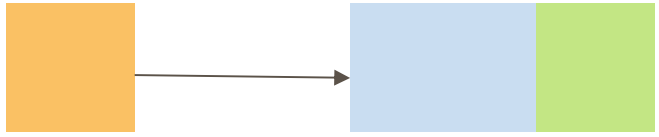
```
main()
{
  int n, encontrado;
  struct lista *cabeza = NULL; //LISTA VACIA

  printf("Elemento a insertar :");
  scanf("%d", &n);

  /* Comprueba si el elemento ya existe */
  encontrado = buscarElemento(cabeza, n); //PASO POR VALOR
  if (!encontrado)
  {
     insertarDelante(&cabeza, n); //PASO POR REFERENCIA
     printf("\n Elemento insertado");
  }
}
```

```
int buscarElemento(struct lista *cabeza, int n);
void insertarDelante(struct lista **cabeza, int n);
```

Listas. NuevoElemento

- Crea en el *heap* un nuevo elemento `struct lista`
- Función auxiliar que se utiliza en la inserción de un elemento en la lista
- No inicializa los campos de la estructura

```
struct lista *nuevoElemento()
{
   return ((struct lista *)malloc(sizeof(struct lista)));
}
```

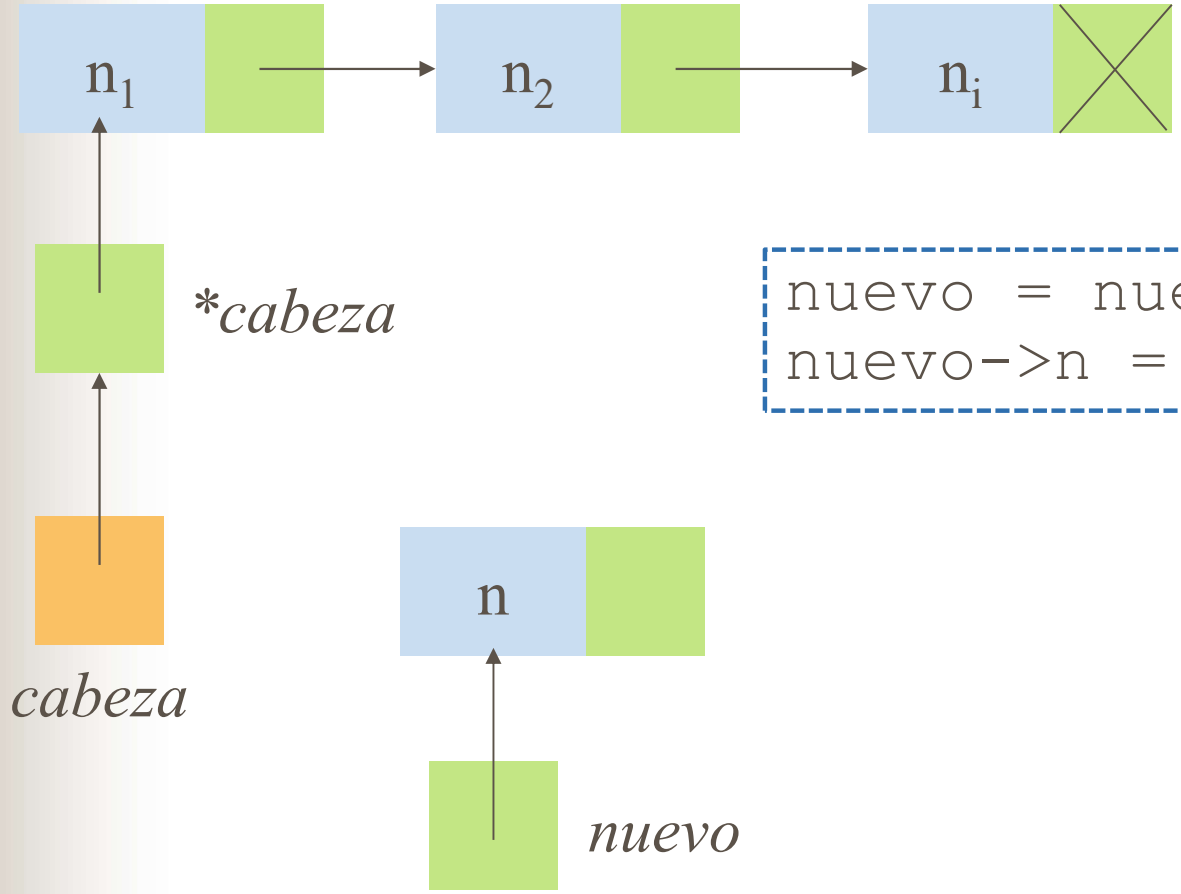


Listas.InsertarDelante

pasamos doble puntero (**) porque vamos a modificar la dirección de comienzo de la lista

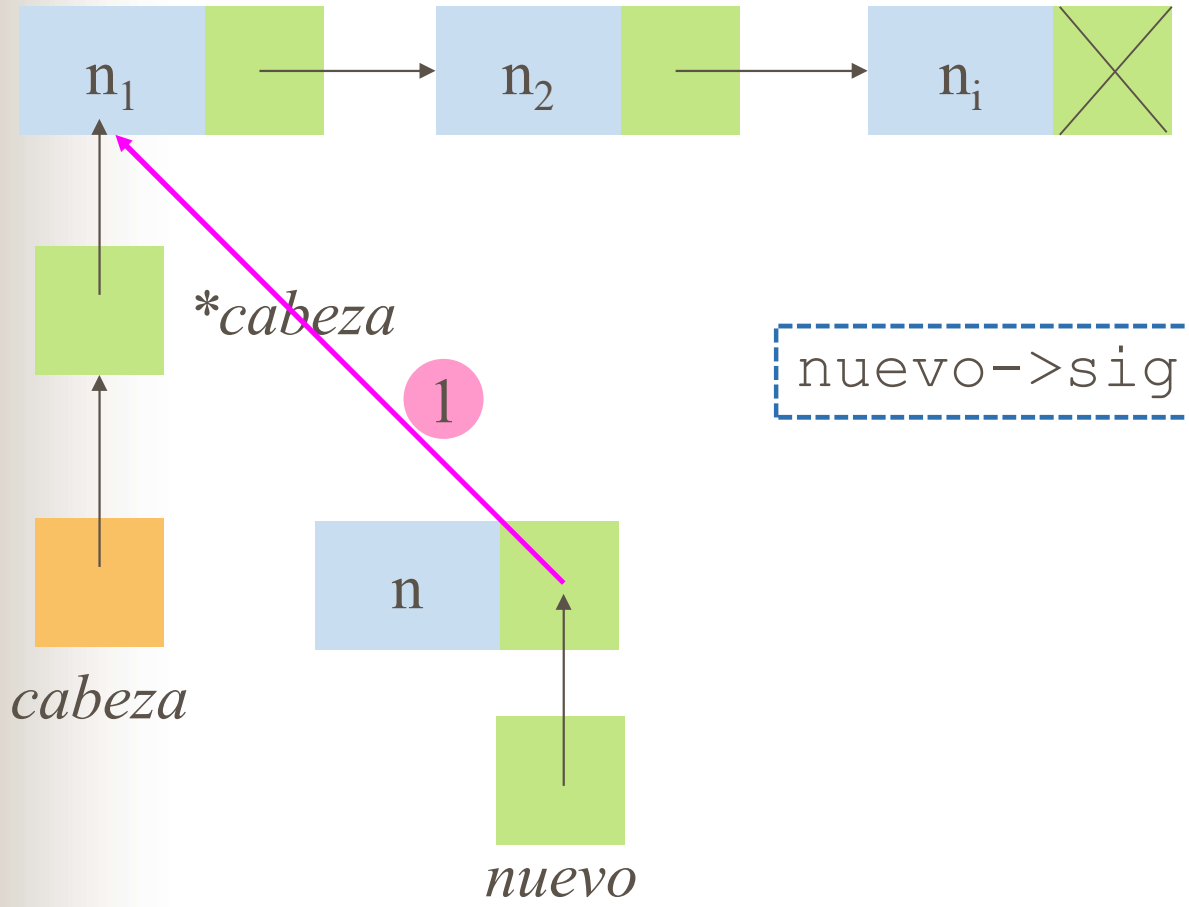
```
void insertarDelante(struct lista **cabeza, int n)
{
 struct lista *nuevo = NULL;

 /* Se reserva espacio para el nuevo elemento */
 nuevo = nuevoElemento();
 nuevo->n = n;

 /*El nuevo elemento se enlaza a la cabeza, y este
   será la nueva cabeza */
 nuevo->sig = *cabeza; 1
 *cabeza = nuevo; 2
}
```

Listas.InsertarDelante



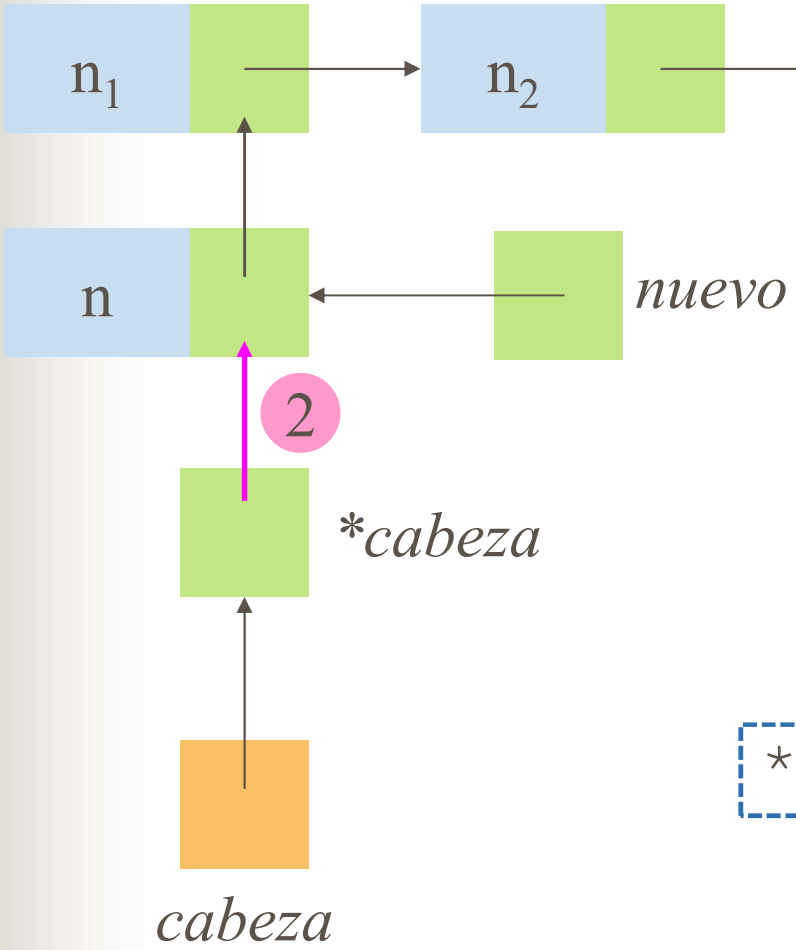

```
nuevo = nuevoElemento();
nuevo->n = n;
```

Listas.InsertarDelante




```
nuevo->sig = *cabeza;
```

Listas.InsertarDelante




```
*cabeza = nuevo;
```

Listas.ImprimirLista

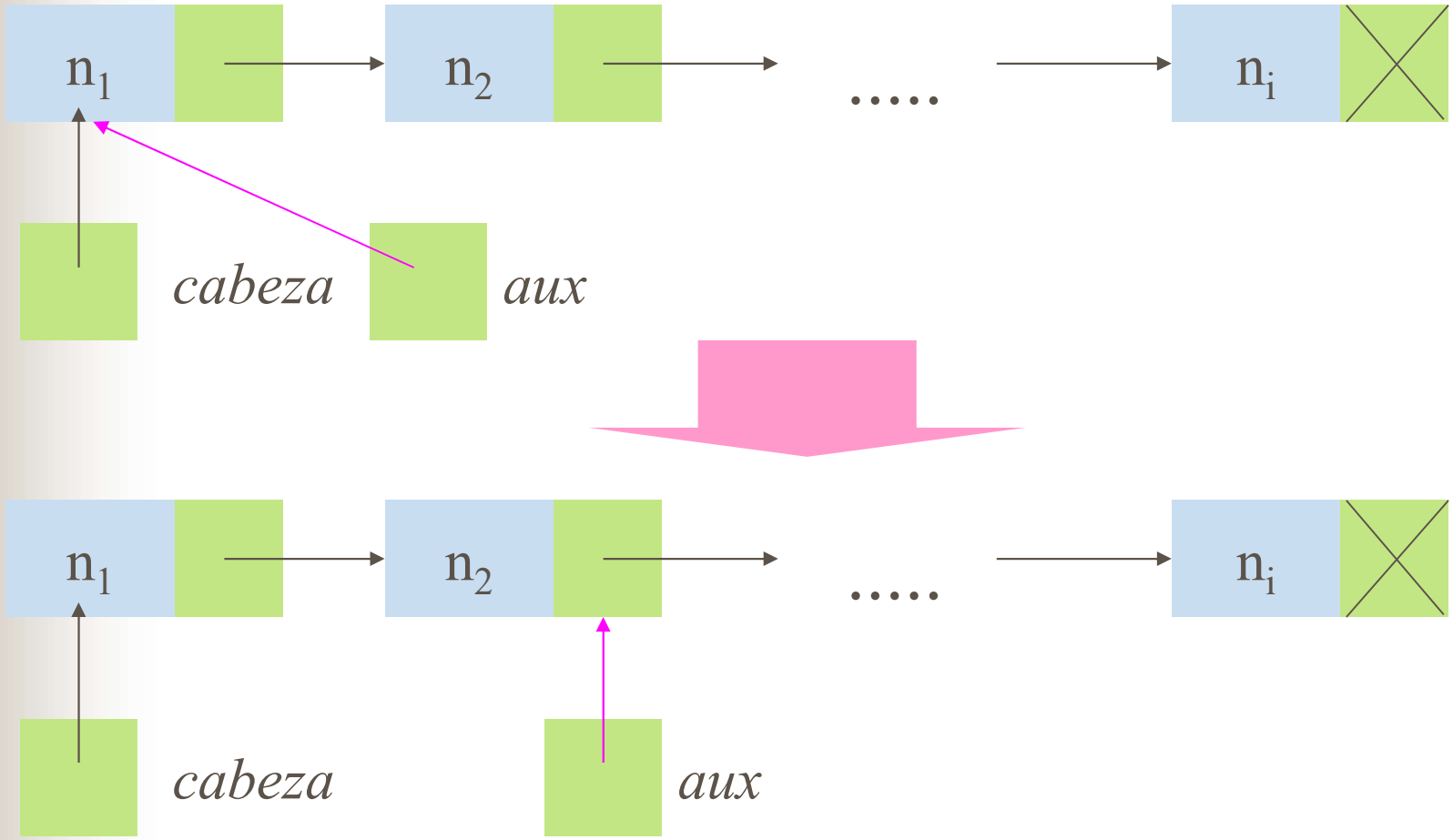
```
void imprimirLista(struct lista *cabeza)
{
 struct lista *aux = NULL;

 aux = cabeza;
 while (aux != NULL)
  {
    printf(" %d \n", aux->n);
    aux = aux->sig;
  }
}
```

Esto nos servirá siempre que queramos recorrer la lista

Si utilizáramos aux->sig!=NULL no escribiríamos el último elemento

También habría problemas si la lista estuviera vacía

Listas.ImprimirLista



Listas.ImprimirListaInverso

```
void imprimirListaInverso(struct lista
   *elemento)
{
 if (elemento != NULL)
   {
    imprimirListaInverso(elemento->sig);
    printf(" %d \n", elemento->n);
   }
}
```

Listas.BuscarElemento

```
int buscarElemento(struct lista *cabeza, int n)
{
 int encontrado = 0;
 struct lista *aux = NULL;
 aux = cabeza;
 /* Se recorre la lista hasta encontrar el elemento o hasta
    llegar al final */
 while (aux != NULL && encontrado ==0)
   {
    if (aux->n == n)
     encontrado = 1;
    else
     aux = aux->sig;
   }
 return encontrado;
}
```

Listas.InsertarDetras

```
void insertarDetras(struct lista **cabeza, int n)
{
 struct lista *nuevo = NULL;
 struct lista *aux = NULL;

 /* se reserva espacio para el nuevo elemento */
 nuevo = nuevoElemento();
 nuevo->n = n;
 nuevo->sig = NULL;

/* la lista está vacia y el nuevo será la cabeza */
 if (*cabeza == NULL)
  *cabeza = nuevo;
```

Condición de último elemento de la lista. No lo hace el constructor y lo tenemos que hacer a mano

Listas.InsertarDetras

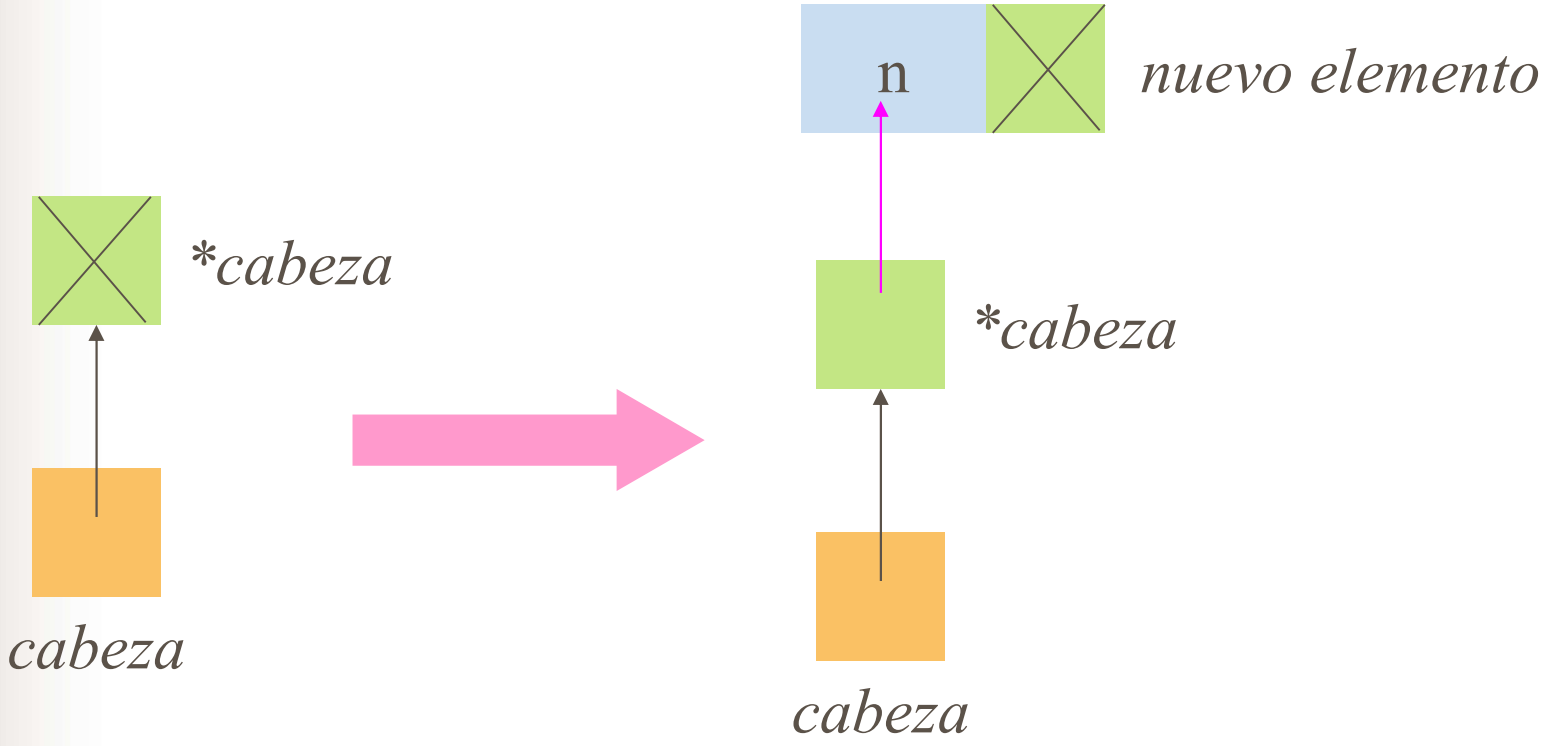
```
else /* se localiza el último elemento para enlazarlo
    al nuevo */
   {
    aux = *cabeza;
    while(aux->sig != NULL)
    {
     aux = aux->sig;
    }
    aux->sig = nuevo;
   }
}
```

Si utilizáramos aux!=NULL no nos quedaríamos con un puntero al último elemento, sino al último puntero a NULL

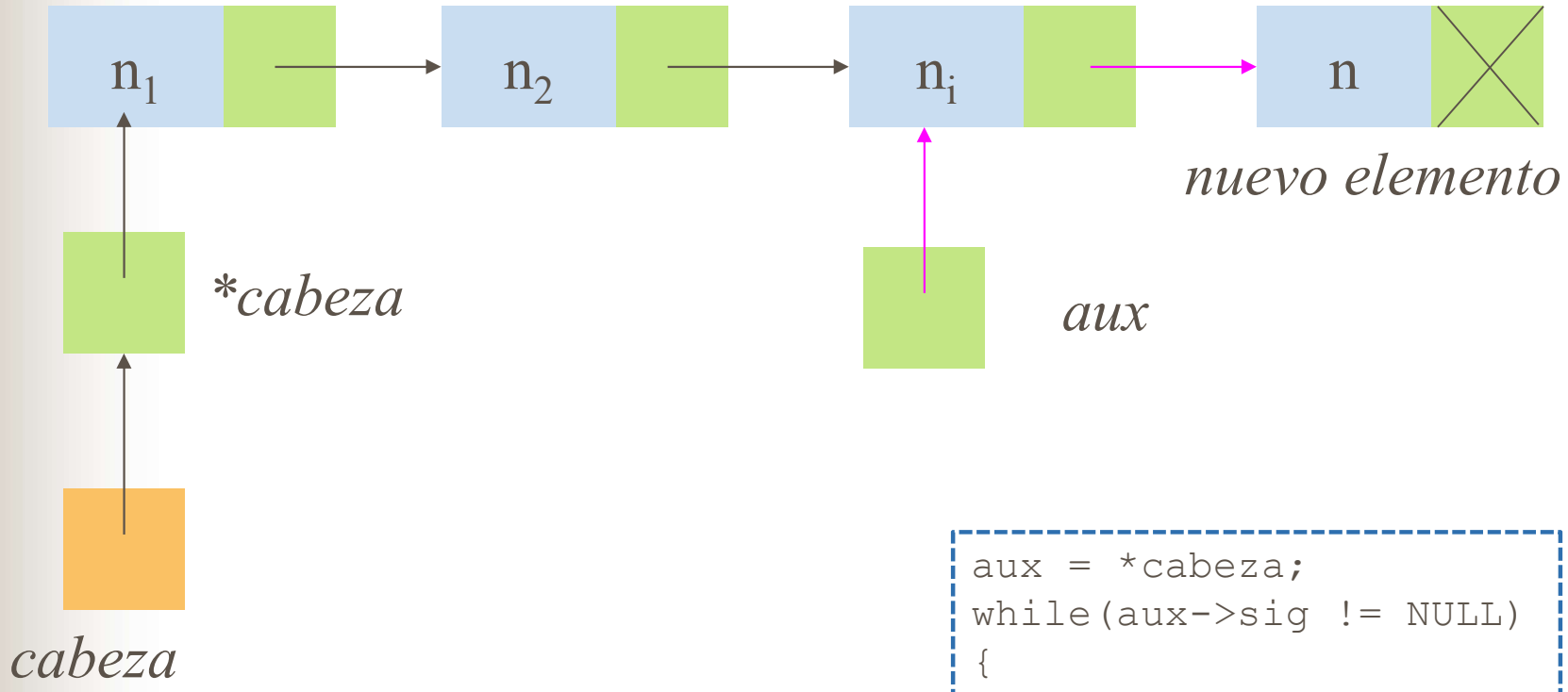
Listas.InsertarDetras

Caso 1. Lista vacía

```
if (*cabeza == NULL)
  *cabeza = nuevo;
```



Listas.InsertarDetras




```
aux = *cabeza;
while(aux->sig != NULL)
{
    aux = aux->sig;
 }
 aux->sig = nuevo;
```

Caso 2. Lista NO vacía

Listas.BorrarElemento

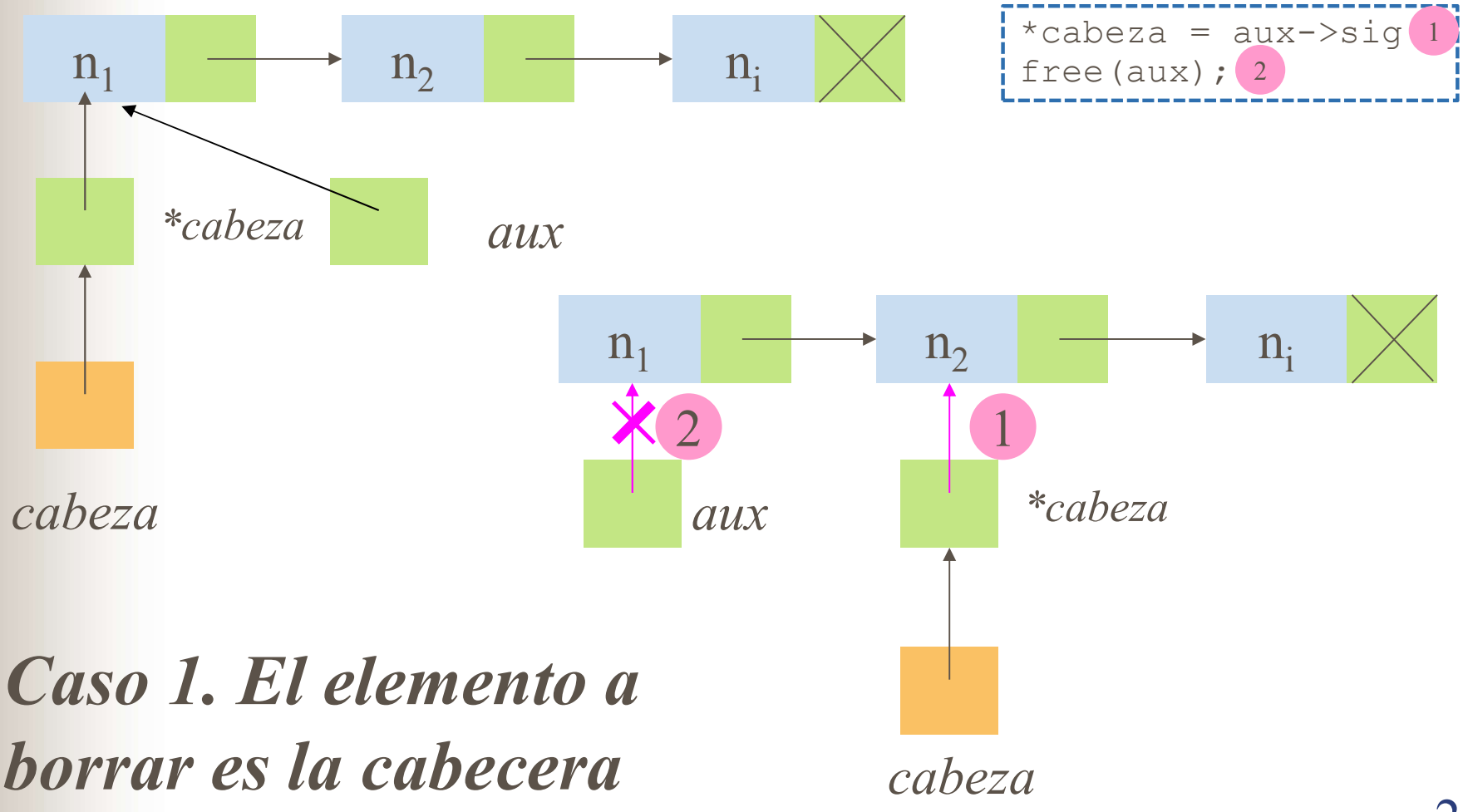
```
void borrarElemento(struct lista **cabeza, int n)
{
 struct lista *ant = NULL;/*anterior al que se borra*/
 struct lista *aux = NULL;/* elemento a borrar */

 /* Busqueda del elemento a borrar y su anterior */
 aux = *cabeza;
 while (aux->n != n)
 {
   ant = aux;
   aux = aux->sig;
 }
```

Se presupone que el elemento está en la lista

Listas.BorrarElemento

```
if (aux == *cabeza) {
/* caso 1: el elemento a borrar es la cabeza */
1   *cabeza = aux->sig;/*la nueva cabeza es el siguiente*/
2   free(aux); /* se libera la antigua cabeza */
  }
 else
/* Caso 2: El elemento a borrar no es la cabeza */
  {
3   ant->sig = aux->sig; /* Enlazar el anterior con el
    siguiente */
4   free(aux); /* se libera el nodo a borrar */
  }
}
```

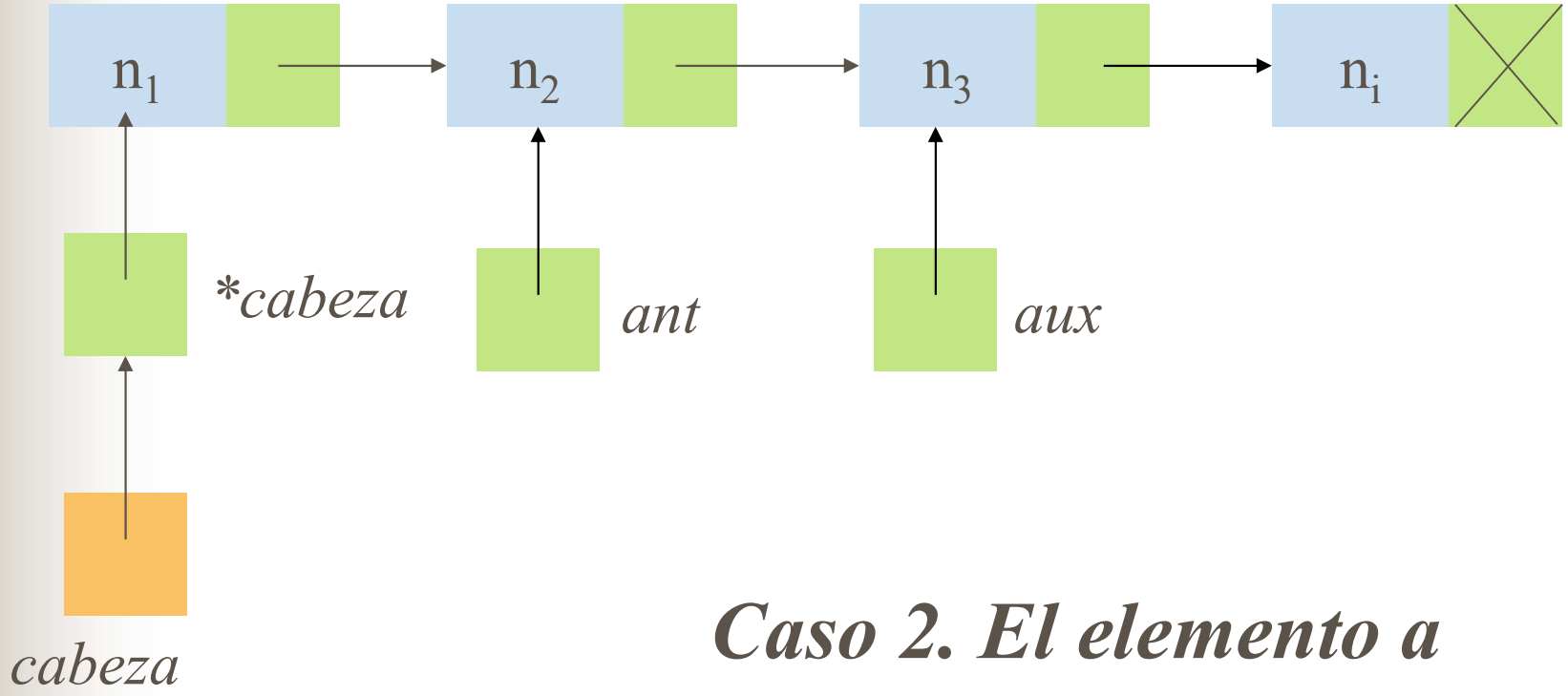
Listas.BorrarElemento



Caso 1. El elemento a borrar es la cabecera

```
3  ant->sig = aux->sig;
4  free(aux);
```

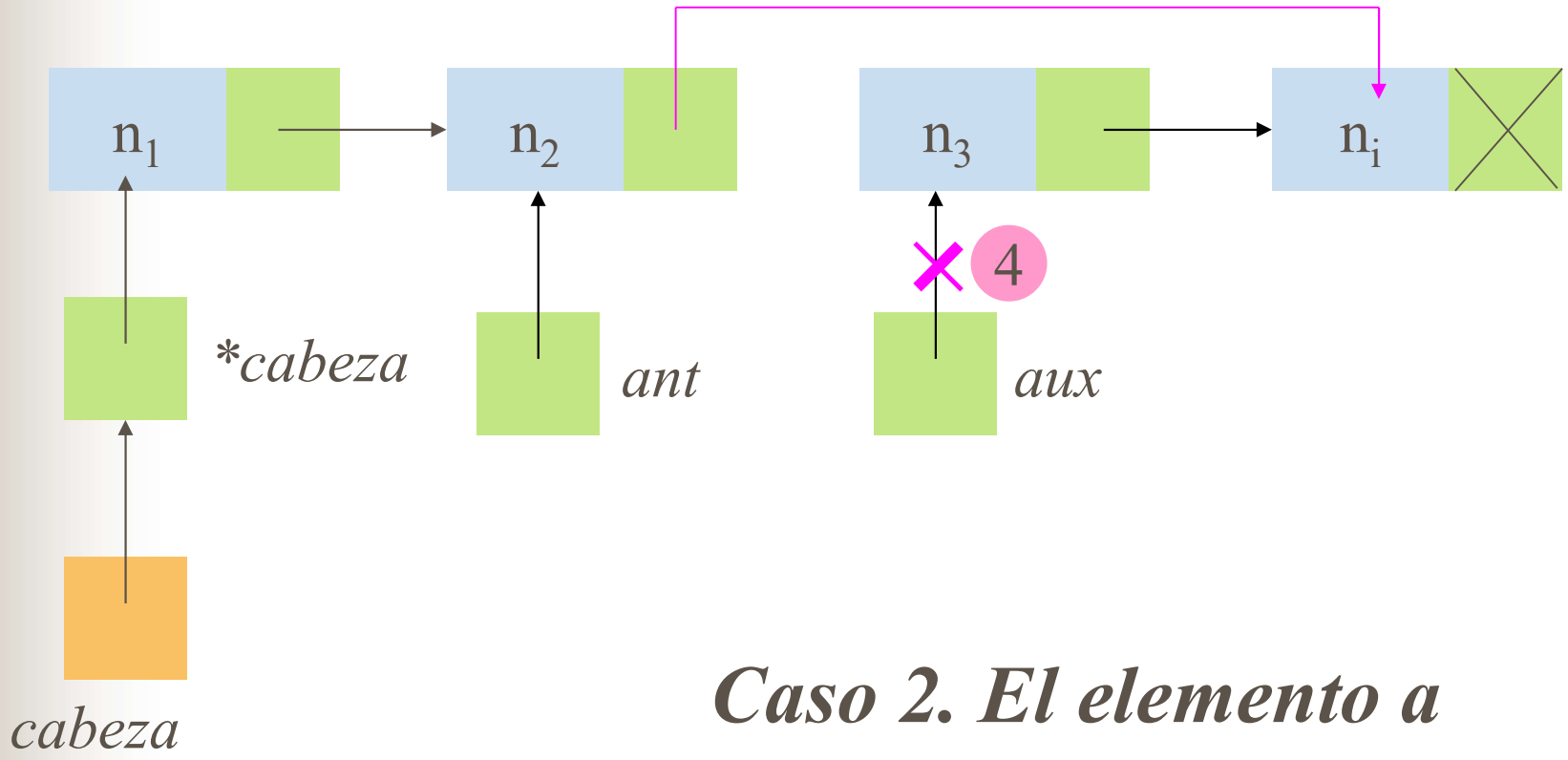
Listas.BorrarElemento



Caso 2. El elemento a borrar NO es la cabecera

```
3  ant->sig = aux->sig;
4  free(aux);
```

Listas.BorrarElemento



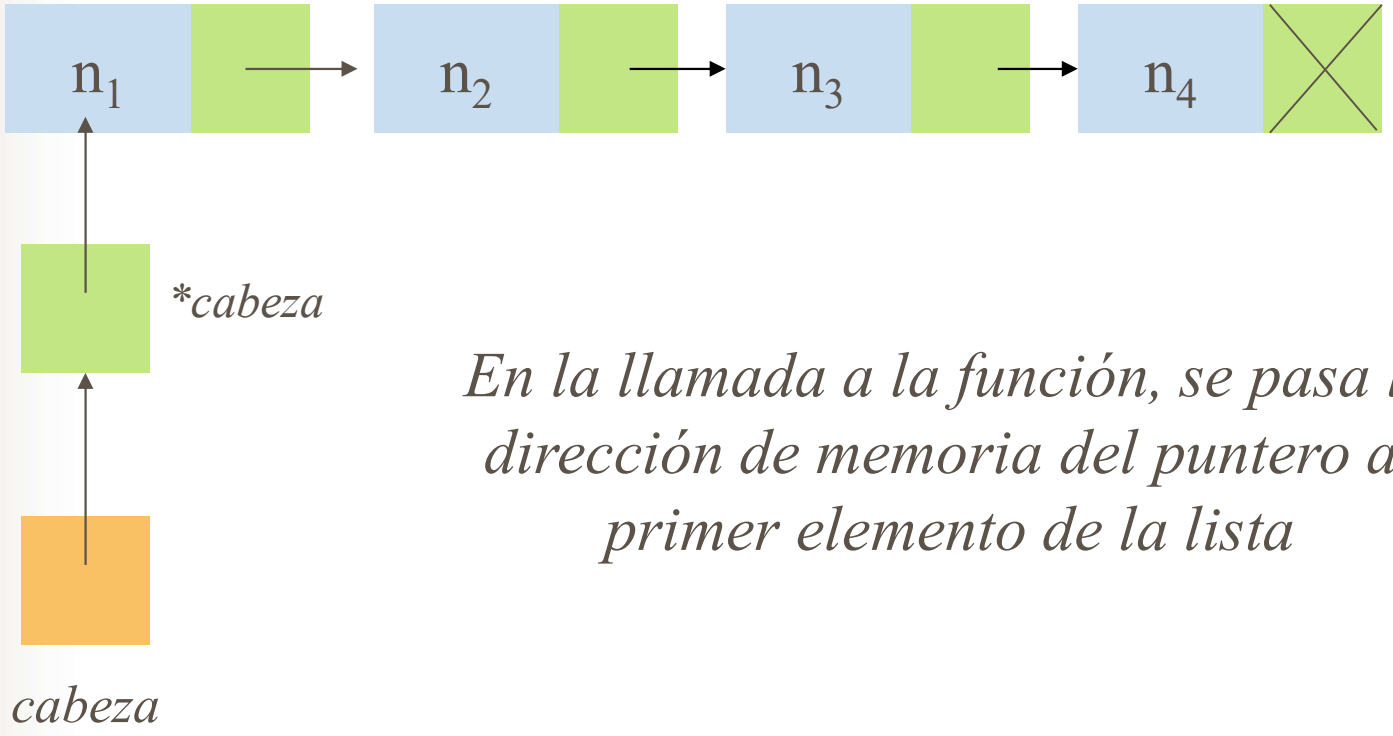
Caso 2. El elemento a borrar NO es la cabecera

Listas.BorrarElementoRecursivo

```
void borrarElementoRecursivo(struct lista **cabeza, int n)
{
 struct lista *aux = NULL; /*elemento a borrar */

 if ((*cabeza)->n != n)
  {
    borrarElementoRecursivo( &((*cabeza)->sig), n);
  }
 else
  {
   aux = *cabeza; 1
   *cabeza = aux->sig; 2
   free (aux); 3
  }
}
```

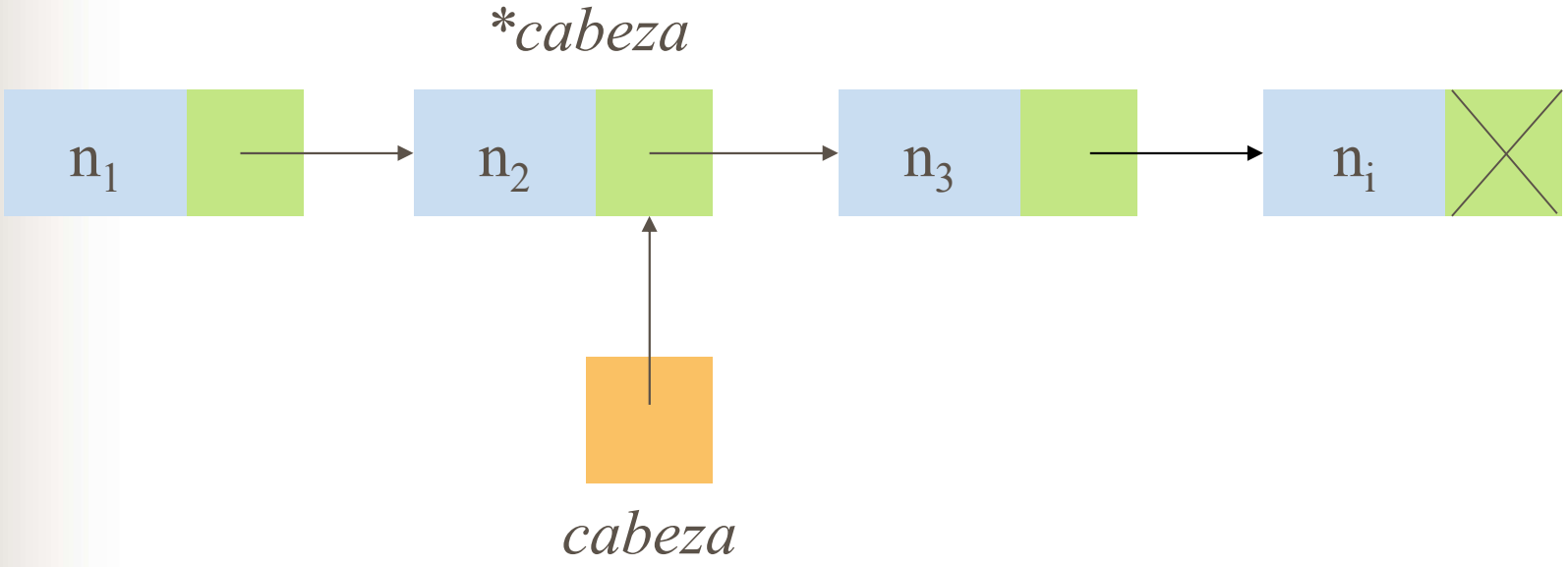
Se presupone que el elemento está en la lista

Listas.BorrarElementoRecursivo



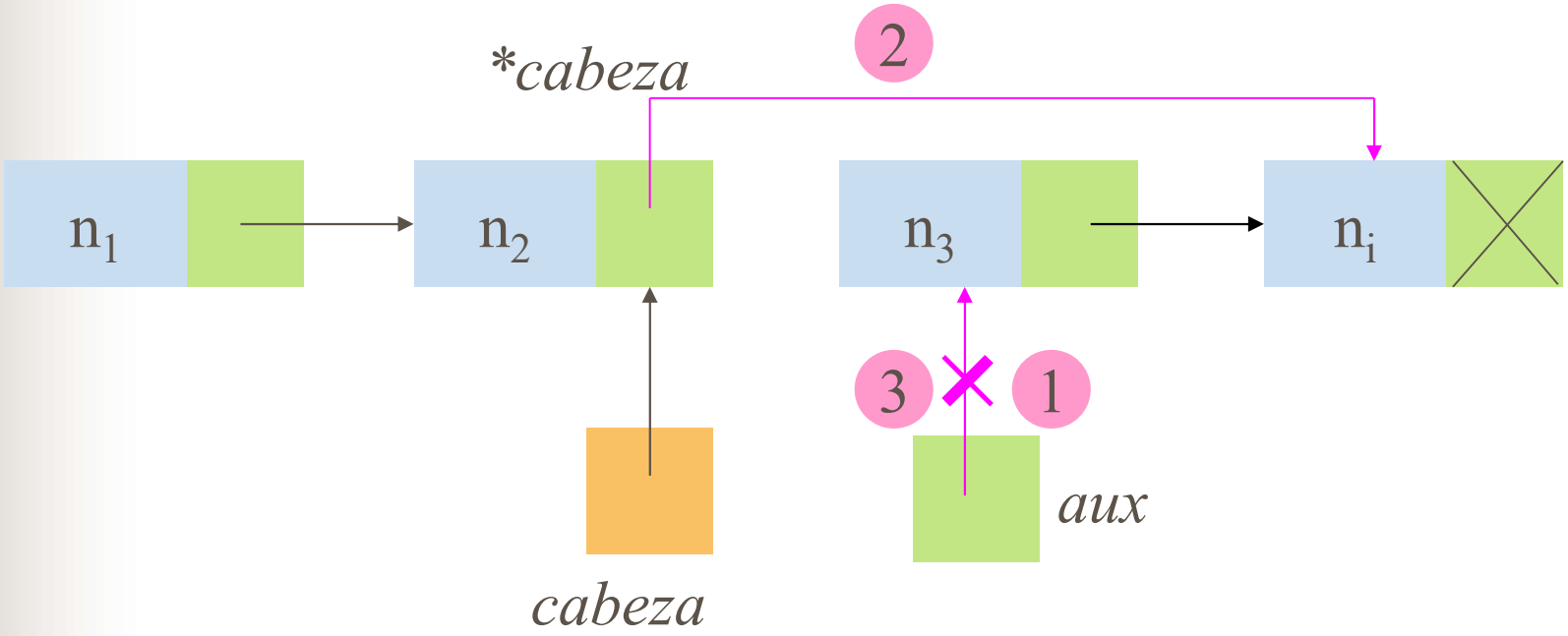
En la llamada a la función, se pasa la dirección de memoria del puntero al primer elemento de la lista

Listas.BorrarElementoRecursivo



*En las llamadas sucesivas, al hacer &((*cabeza)->siguiente) pasamos la dirección de memoria del puntero (*cabeza)->siguiente*

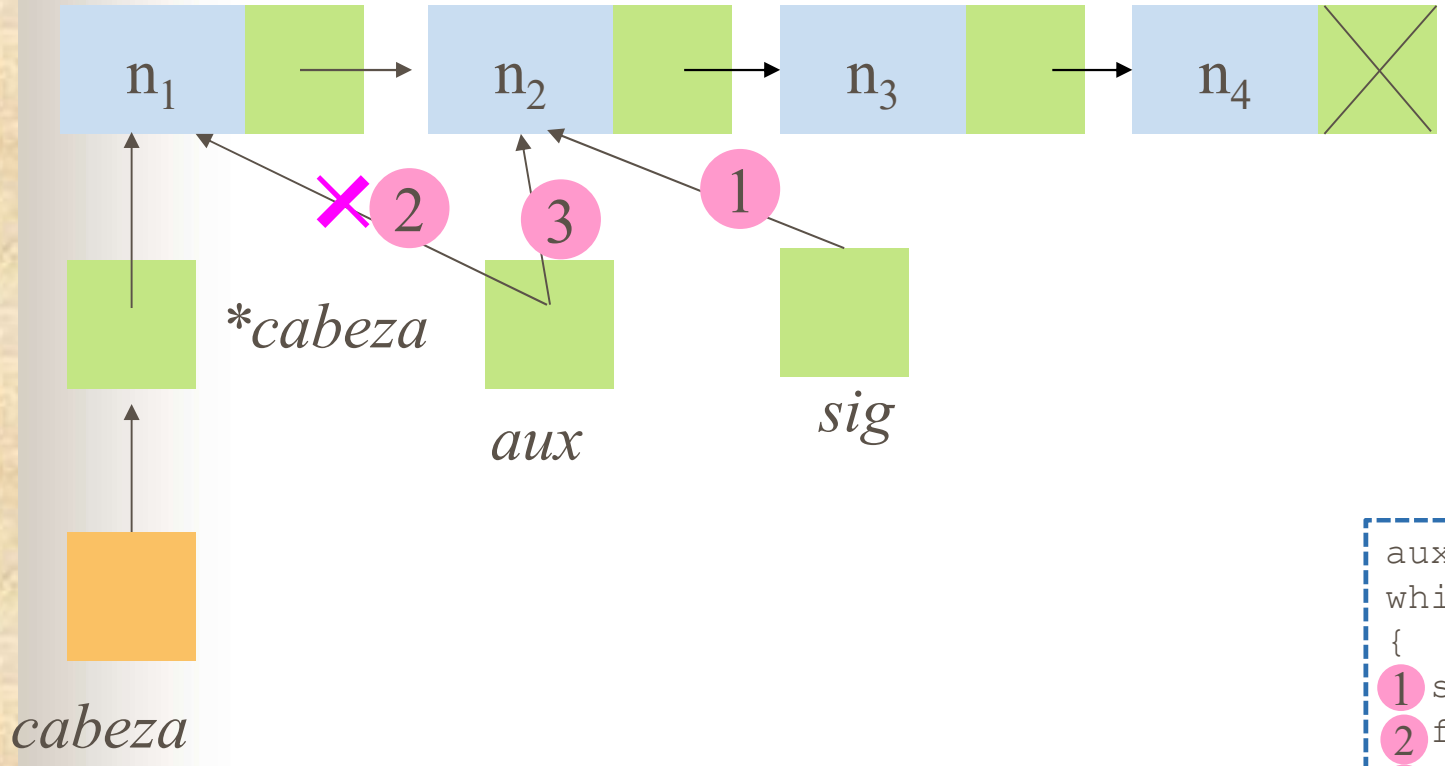
Listas.BorrarElementoRecursivo



Listas.BorrarLista

```
void borrarLista(struct lista **cabeza)
{  struct lista * aux, * sig;

   aux=*cabeza;
    while(aux!=NULL)
     {
       sig=aux->sig;   1
       free(aux);   2
       aux=sig;   3
     }
    *cabeza=NULL; //Lista vacia
}
```

Listas.BorrarLista

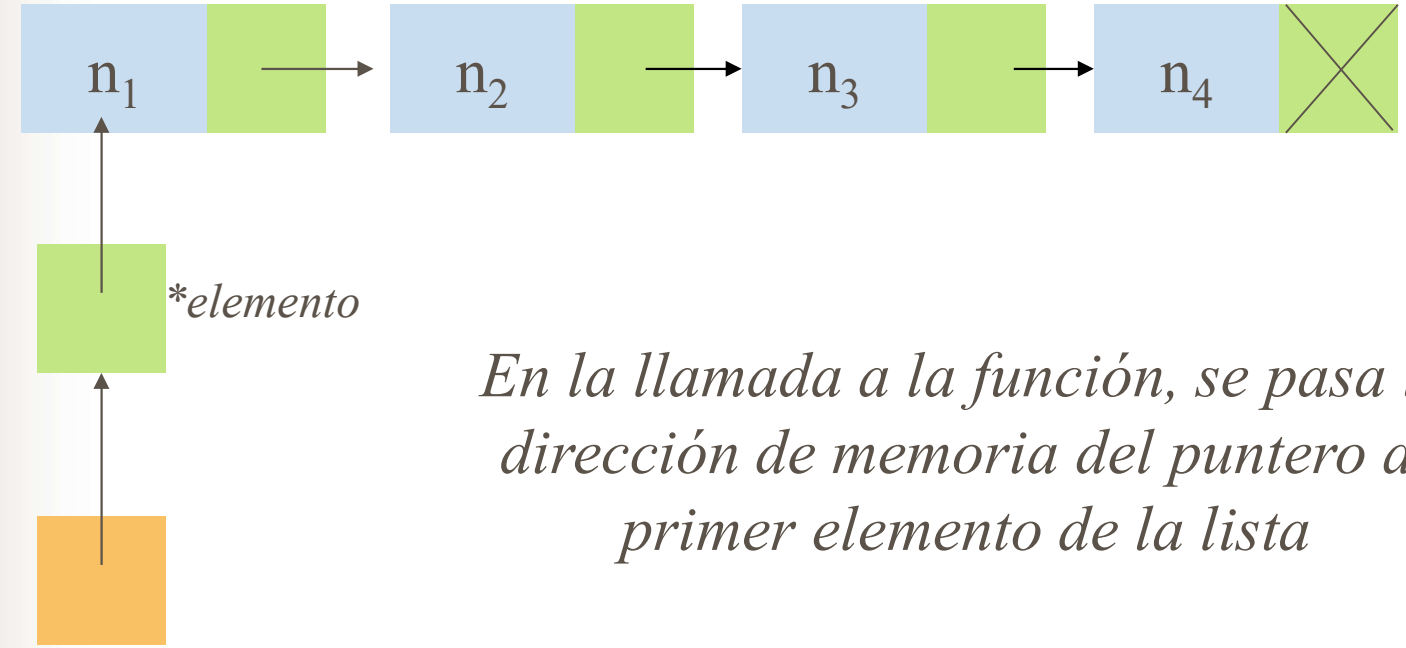



```
aux=*cabeza;
while(aux!=NULL)
{
1 sig=aux->sig;
2 free(aux);
3 aux=sig;
}
*cabeza=NULL;
```

Listas.BorrarListaRecursiva

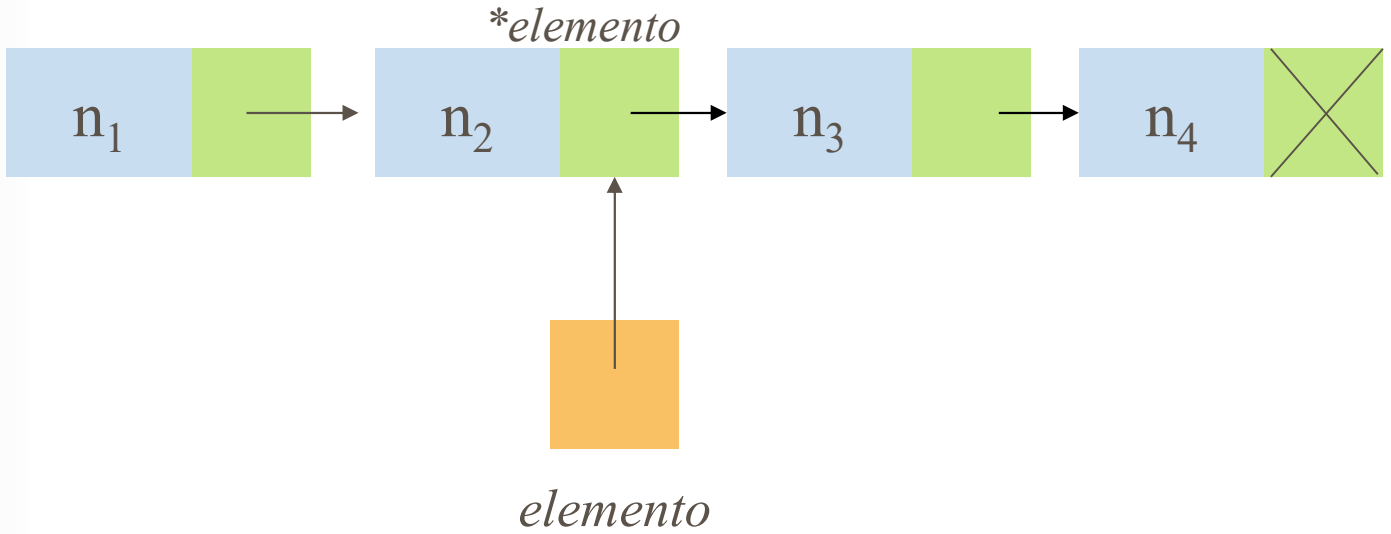
```
void borrarListaRecursiva(struct lista **elemento)
{
 if (*elemento != NULL)
   {
    borrarListaRecursiva(&((*elemento)->sig));
    free(*elemento);
    *elemento = NULL; //Fin de lista o lista vacia
   }
}
```

Listas.BorrarListaRecursiva



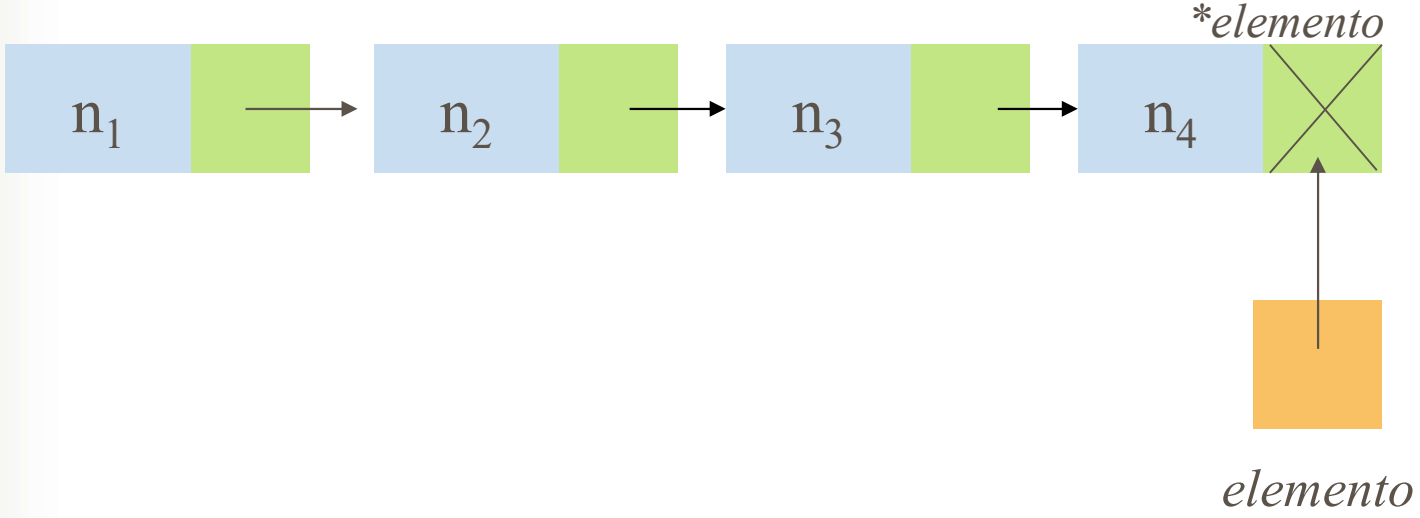
En la llamada a la función, se pasa la dirección de memoria del puntero al primer elemento de la lista

Listas.BorrarListaRecursiva



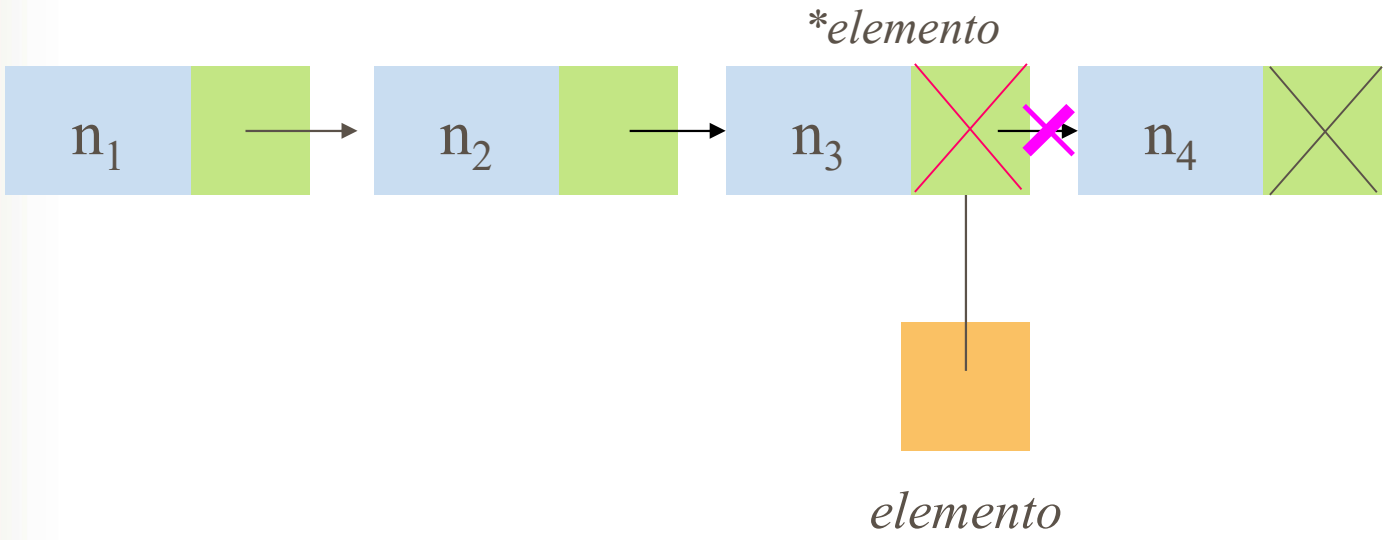
*En las llamadas sucesivas, al hacer &((*elemento)->siguiente) pasamos la dirección de memoria del puntero (*elemento)->siguiente*

Listas.BorrarListaRecursiva



*Caso base: *elemento==NULL*

Listas.BorrarListaRecursiva



Vuelta atrás de la recursividad

Listas.InsertarOrden

Se presupone que la lista ha sido previamente ordenada, o que todas las inserciones se han hecho en orden

```
void insertarOrden(struct lista **cabeza, int n)
{
 struct lista *ant = NULL;    /*anterior al que se inserta*/
 struct lista *aux = NULL;    /*posterior al que se inserta*/
 struct lista *nuevo = NULL; /* nuevo elemento */
 int encontrado = 0;          /*posición de inserción encontrada*/

 /* Se reserva espacio para el nuevo elemento */
 nuevo = nuevoElemento();
 nuevo->n = n;

 /* lista vacía o el elemento se inserta delante de la cabeza*/
 if((*cabeza == NULL) || ((*cabeza)->n > n))
 {
    nuevo->sig = *cabeza; 1
    /*la cabeza será el nuevo elemento*/
    *cabeza = nuevo; 2
 }
```

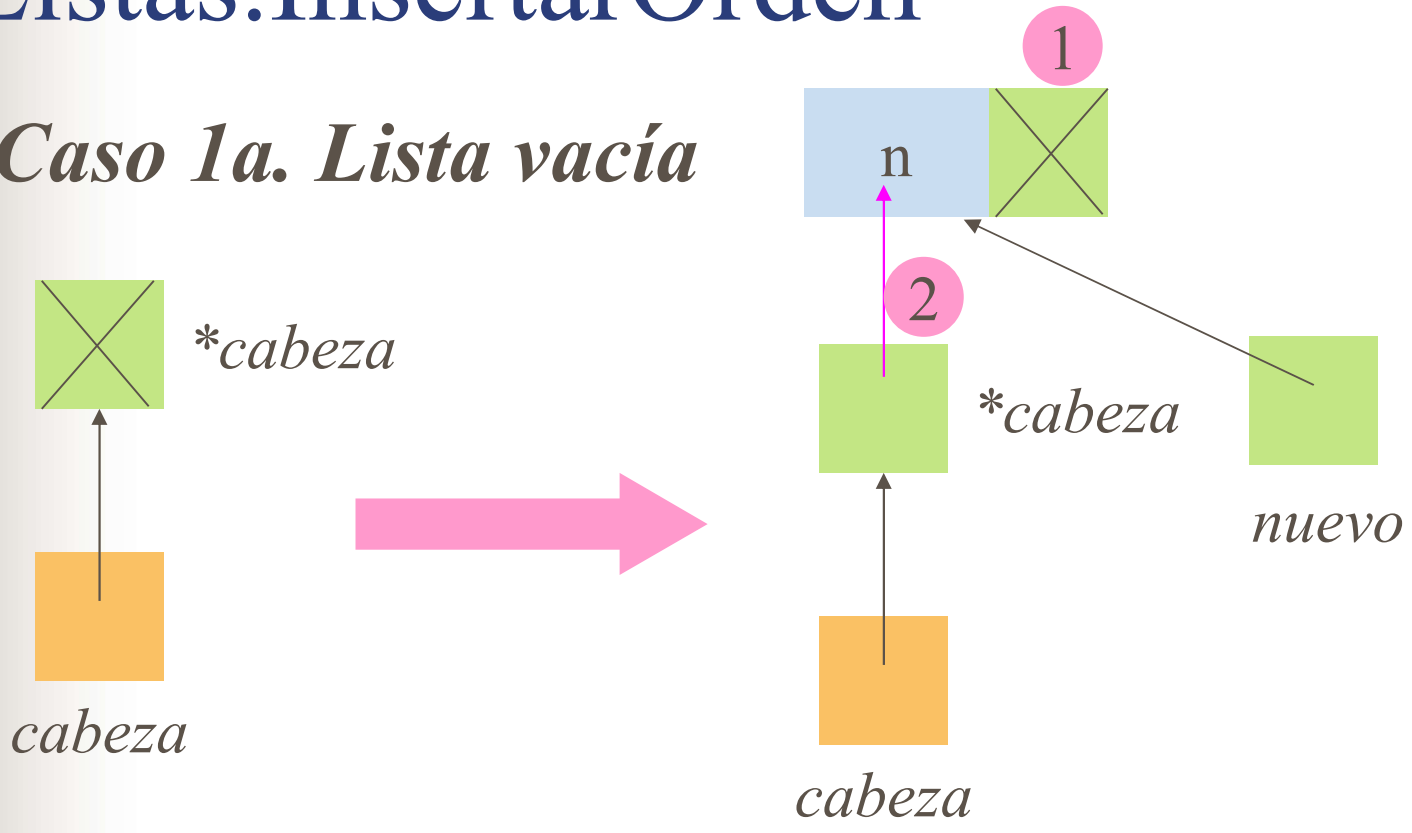
Comprobar en primer lugar si *cabeza==NULL para evitar accesos a memoria no reservada. Evaluación del OR en cortociruito

Listas.InsertarOrden

```
else{ /*lista no vacía o insercion en el medio de la lista*/
   /*busqueda de la posicion de insercion, se interrumpe cuando se
    encuentra el primer elemento mayor que n o cuando se llega al
    final de la lista*/
    aux = *cabeza;
    while (aux != NULL && encontrado == 0)
    {
      if (aux->n > n)/*se ha encontrado la posicion de insercion*/
          encontrado = 1;
      else /*se actualizan los valores de aux y ant*/
      {
          ant = aux;
          aux = aux->sig;
      }
    }//while
    /* ubicamos el elemento nuevo entre ant y aux. Estas acciones
        son válidas aunque aux sea igual a NULL */
    nuevo->sig = aux;   3
    ant->sig = nuevo;   4
 } //else
}
```

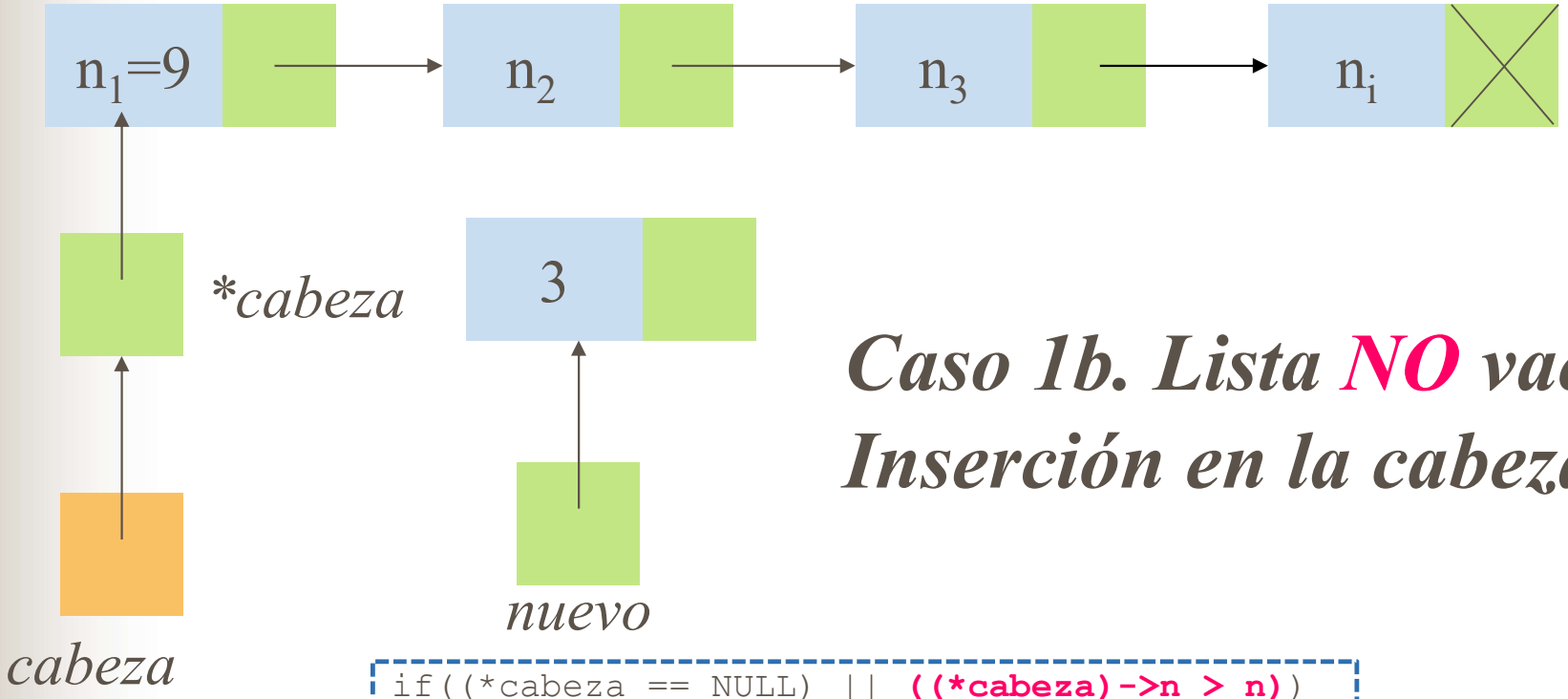
Listas.InsertarOrden

Caso 1a. Lista vacía




```
if((*cabeza == NULL) || ((*cabeza)->n > n))
 {
    nuevo->sig = *cabeza;   1
    *cabeza = nuevo;        2
 }
```

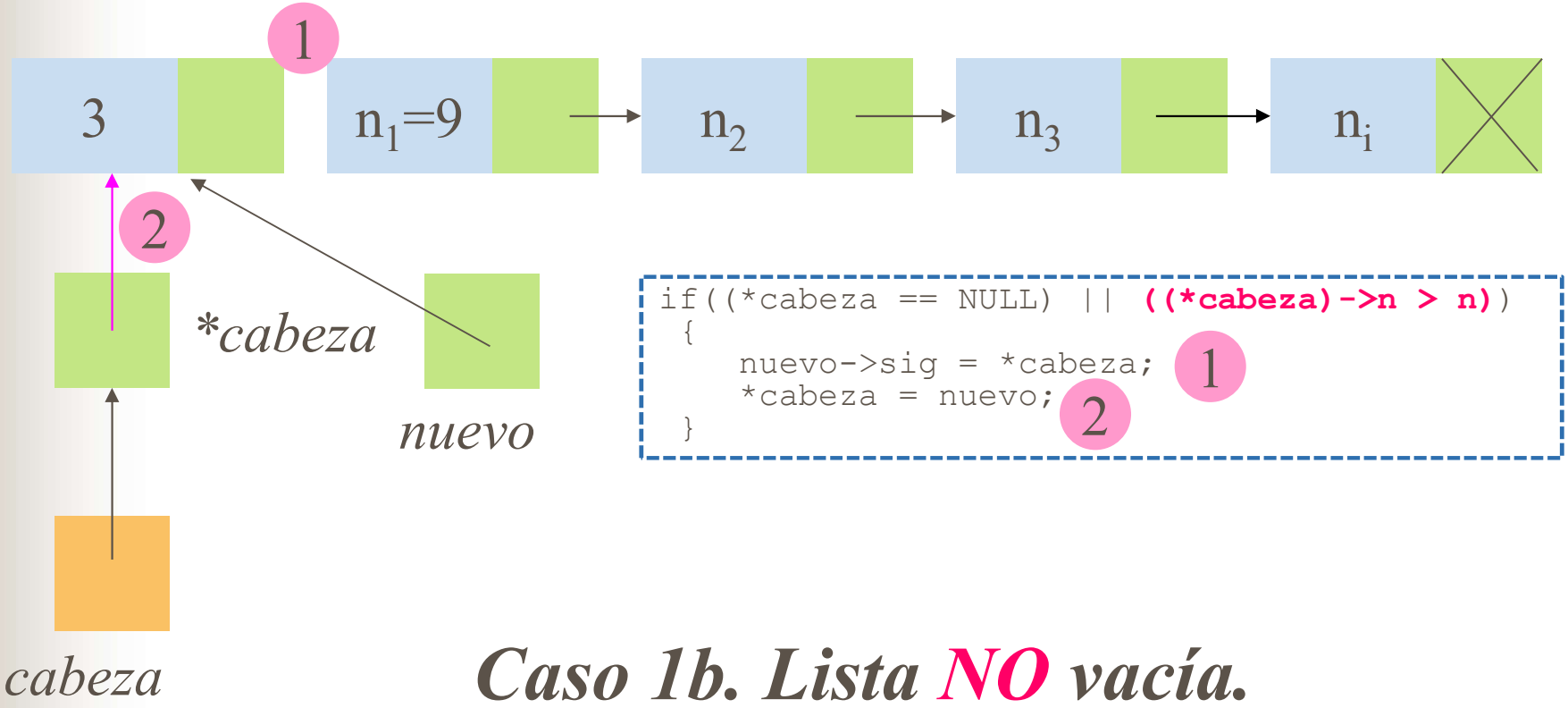
Listas.InsertarOrden



Caso 1b. Lista NO vacía. Inserción en la cabeza

```
if((*cabeza == NULL) || ((*cabeza)->n > n))
 {
    nuevo->sig = *cabeza;   1
    *cabeza = nuevo;
 }                          2
```

Listas.InsertarOrden




```
if((*cabeza == NULL) || ((*cabeza)->n > n))
 {
    nuevo->sig = *cabeza;
    *cabeza = nuevo;
 }
```

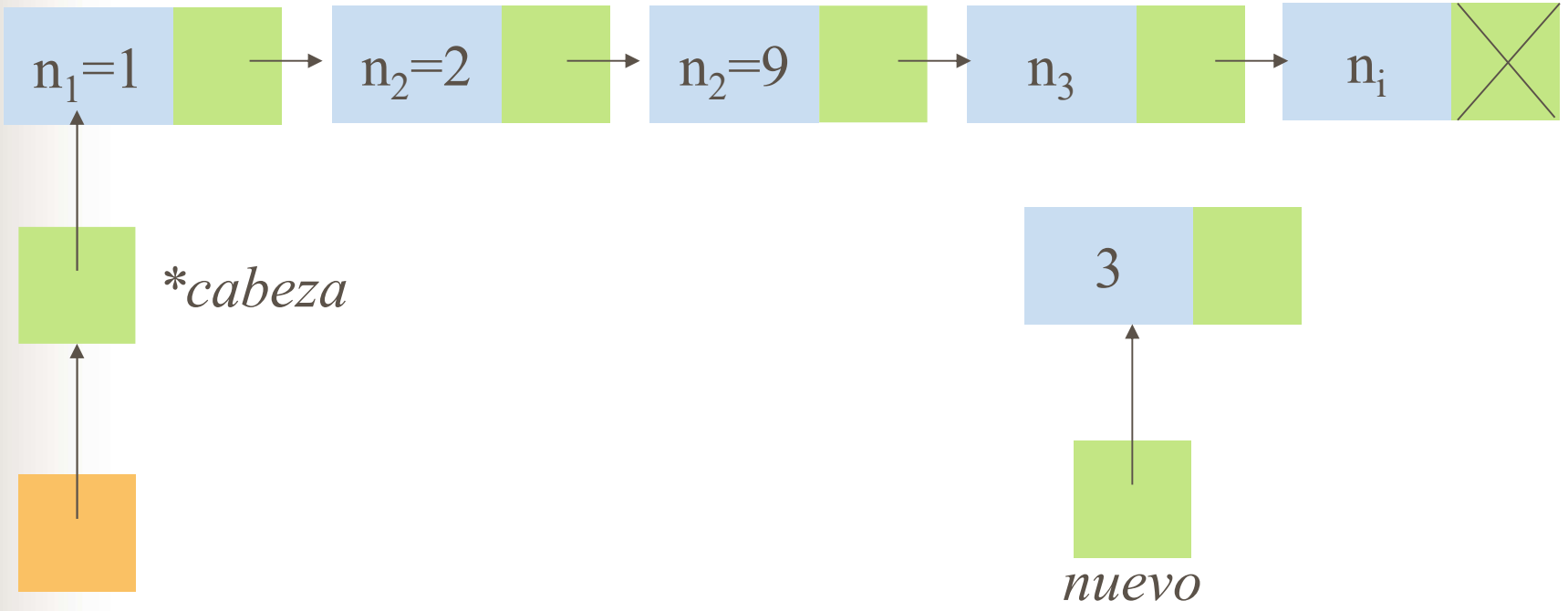
1

2

Caso 1b. Lista NO vacía.
Inserción en la cabeza

Listas.InsertarOrden

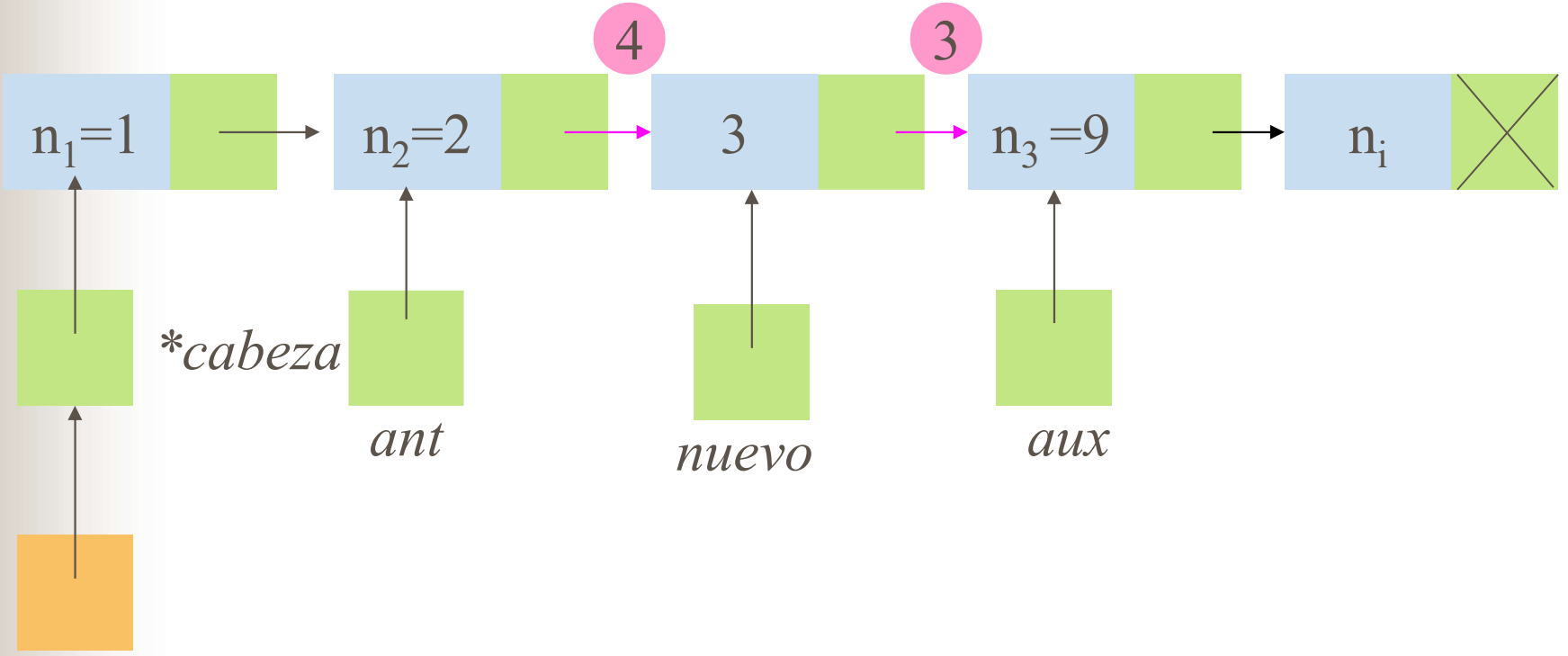
```
aux = *cabeza;
while (aux != NULL && encontrado == 0)
{  if (aux->n > n
     encontrado = 1;
   else{
     ant = aux;
     aux = aux->sig;
   }
}
```



Caso 2. Lista NO vacía. Inserción en el centro de la lista

```
3  nuevo->sig = aux;
4  ant->sig = nuevo;
```

Listas.InsertarOrden



Caso 2. Lista NO vacía. Inserción en el centro de la lista

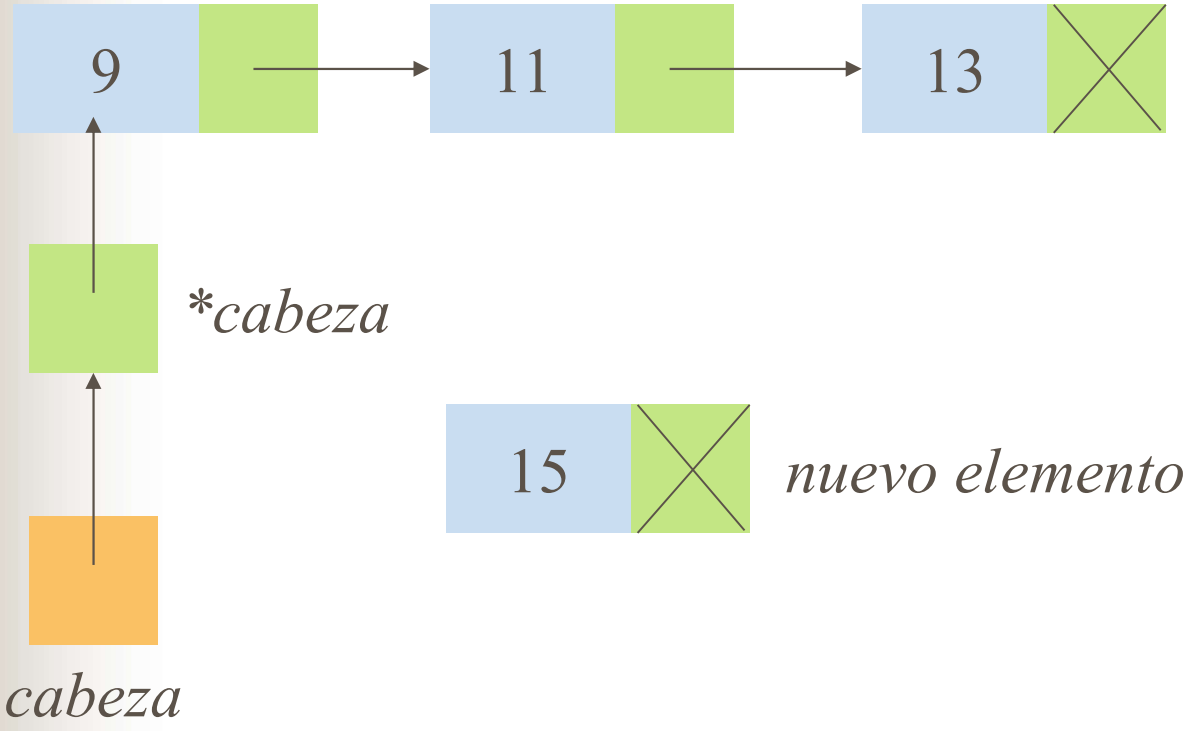
Se presupone que la lista ha sido previamente ordenada, o que todas las inserciones se han hecho en orden

Listas.InsertarOrdenRecursivo

```
void insertarOrdenRecursivo(struct lista **cabeza,
    int n)
{
   if( (*cabeza==NULL) || ((*cabeza)->n > n) )
     {
      nuevo = nuevoElemento();
      nuevo->n = n;
      nuevo->sig = (*cabeza);   1
      *cabeza = nuevo;          2
     }
   else
     {
      insertarOrdenRecursivo( &((*cabeza)->sig), n);
     }
}
```

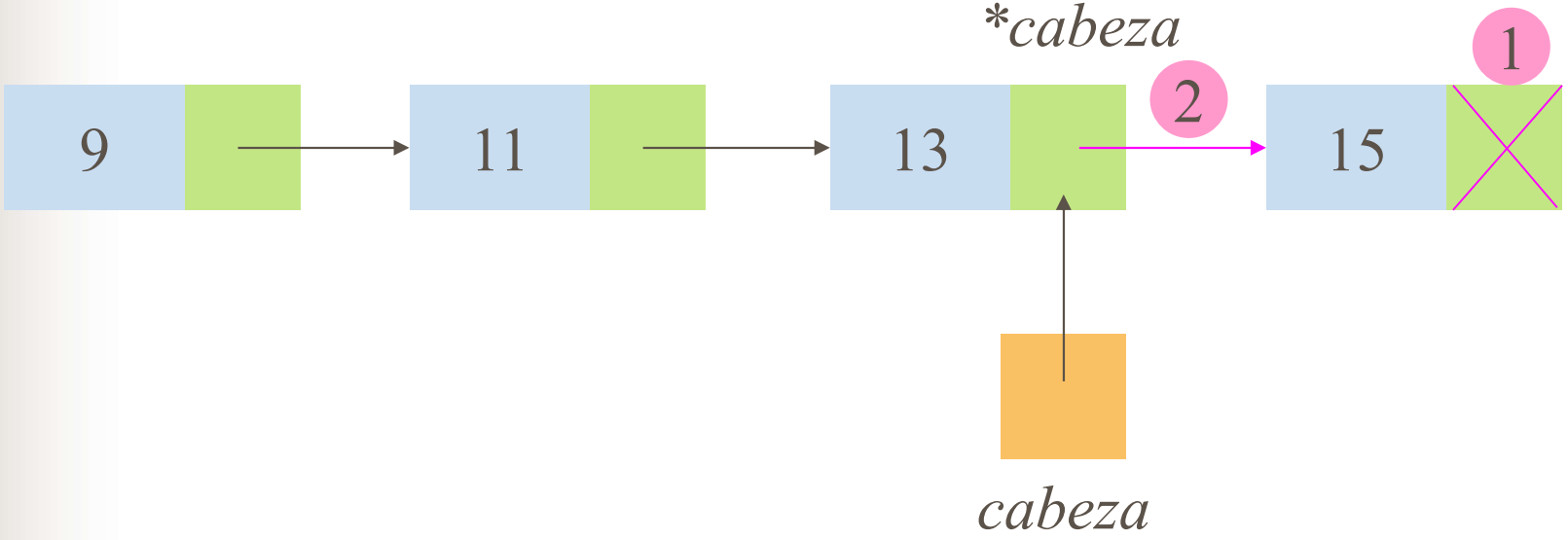
Comprobar en primer lugar si *cabeza==NULL para evitar accesos a memoria no reservada. Evaluación del OR en cortociruito

Listas.InsertarOrdenRecursivo



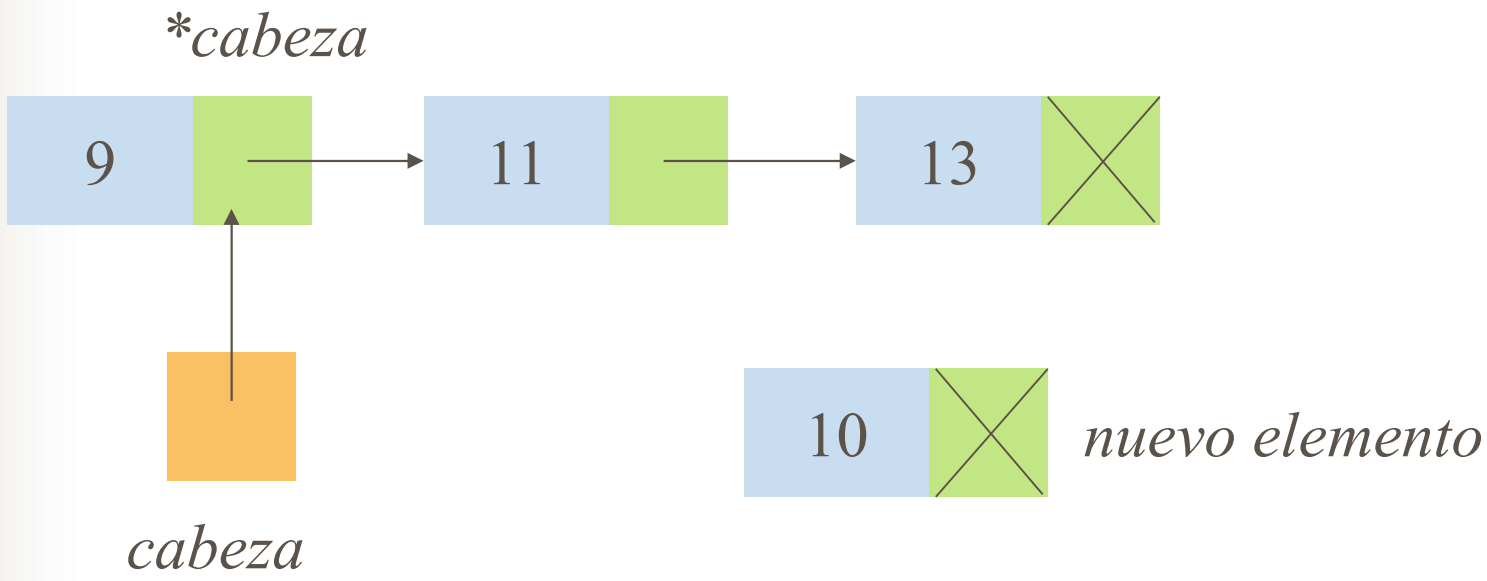
Caso 1. Lista vacía o mayor elemento

Listas.InsertarOrdenRecursivo



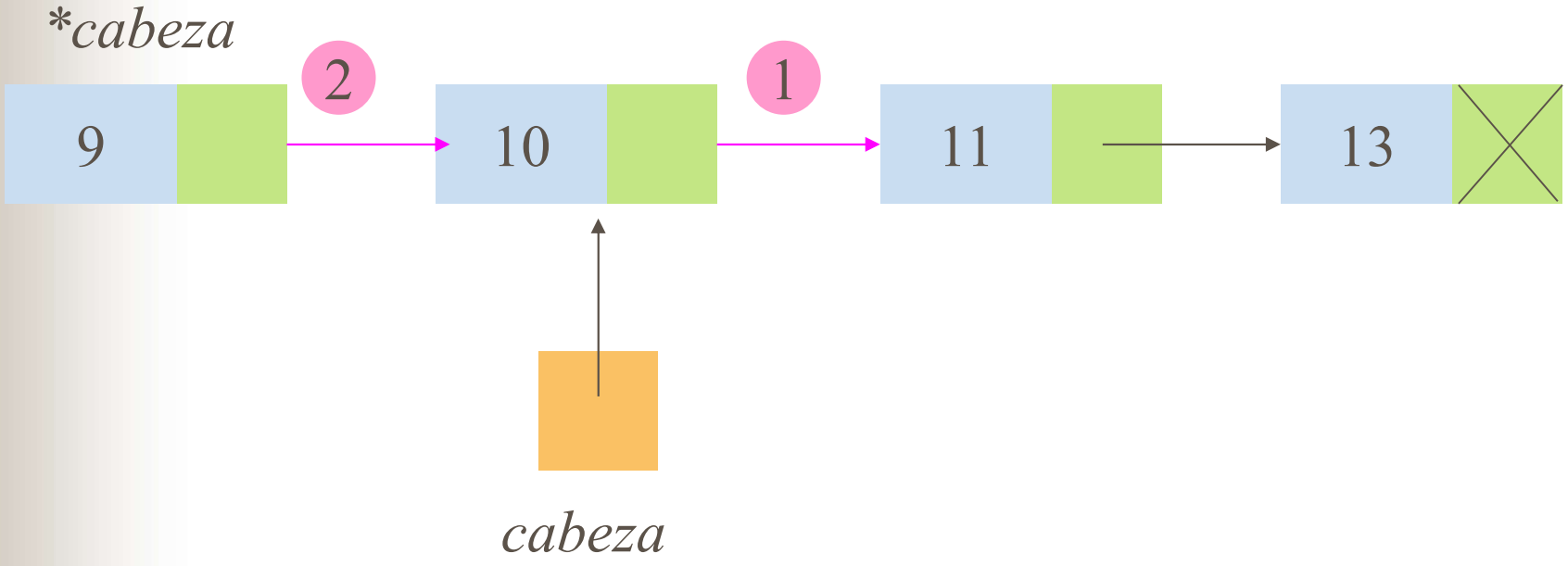
Caso 1. Lista vacía o mayor elemento

Listas.InsertarOrdenRecursivo



Caso 2. Inserción en el centro

Listas.InsertarOrdenRecursivo



Caso 2. Inserción en el centro

Listas.OrdenarLista

Algoritmo de Selección

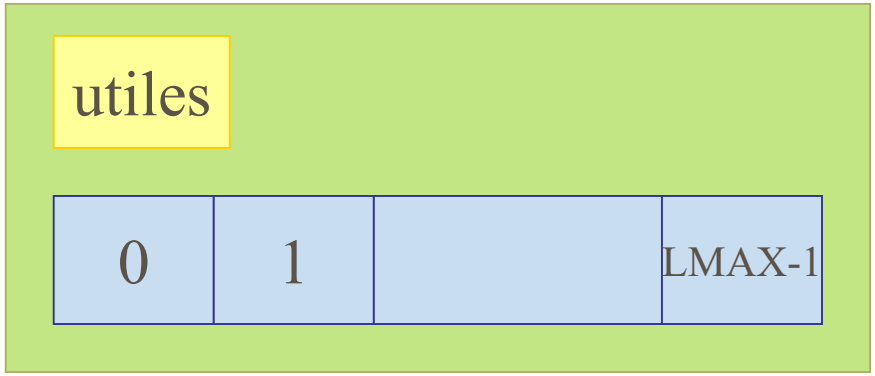
```
void ordenarLista(struct lista *cabeza)
{
 struct lista *aux;
 struct lista *aux1;
 struct lista *minimo;
 int minimo_n;
 aux = cabeza;
 //Recorrer toda la lista
 while(aux->sig != NULL)//while1
 { aux1 = aux->sig;
   minimo = aux;
   while(aux1 != NULL)//while2
   {
      if (aux1->n < minimo->n)
      {
         minimo = aux1; //seleccionar minimo
      }
      aux1 = aux1->sig;
 }//while 2
 //Intercambio aux y minimo
 minimo_n = minimo->n;
 minimo->n = aux->n;
 aux->n = minimo_n;
 //Preparar siguiente iteración
 aux = aux->sig;
 }//While1
}
```

Listas.Otras implementaciones

Mediante vectores

- Los elementos son posiciones consecutivas de un vector
- Las listas son de longitud variable y los vectores de longitud fija. Se considerarán vectores de tamaño igual a la longitud máxima de la lista
- Se añade un entero para indicar el último elemento válido de la lista

```
struct lista
{
 int utiles;
 int elementos[LMAX];
};
```



Listas.Otras implementaciones

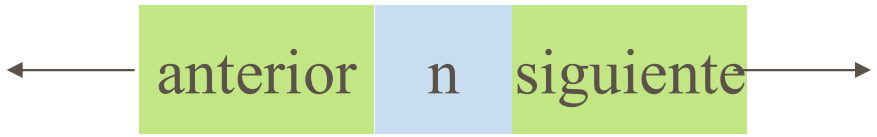
- Inconvenientes
  - En algunas operaciones hay que mover todos los elementos que ocupen una posición superior a la considerada para realizar dicha operación. Esto tiene como consecuencia que la eficiencia de las operaciones no es muy buena, del orden del tamaño de la lista
    - Para la operación de inserción hay que hacer previamente un hueco donde realizar dicha inserción
    - Para el borrado, hay que rellenar el hueco dejado por el elemento borrado
  - Otro inconveniente es que las listas tienen un tamaño máximo que no se puede pasar
  - Siempre hay una porción de espacio reservada para los elementos de la lista, y que no se utiliza al ser el tamaño de la lista en un momento dado, menor que el tamaño máximo

Lista.Otras implementaciones

Listas doblemente enlazadas

- En algunas aplicaciones podemos desear:
  - Recorrer una lista hacia delante y hacia atrás
  - Conocer rápidamente los elementos anterior y siguiente dado un un determinado elemento
- La solución es dotar a cada elemento de la lista de un puntero a los elementos anterior y siguiente
- El precio que se paga es la presencia de un puntero adicional en cada elemento

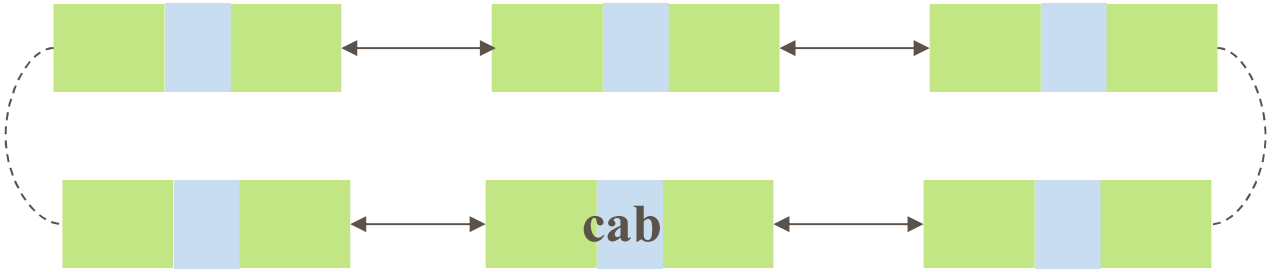
```
struct lista{
 int n;
 struct lista *sig, * ant;
};
```



Listas.Otras implementaciones

Listas doblemente enlazadas

- Se suele hacer que la cabecera de una lista doblemente enlazada sea una celda que complete el círculo es decir:
  - El anterior a la celda de la cabecera es la última celda de la lista
  - El siguiente a la celda de la cabecera es es la primera de la lista
- El resultado es una implementación de listas doblemente enlazadas con cabecera y estructura circular en el sentido de que, dado un nodo, y por medio de los punteros siguiente, podemos volver hasta el

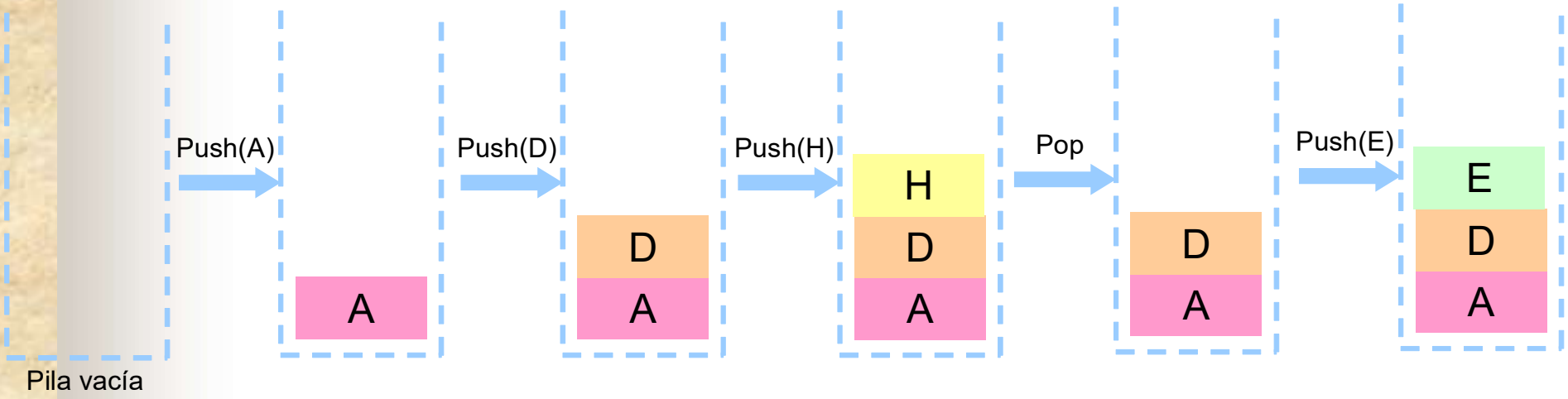


Lista circular

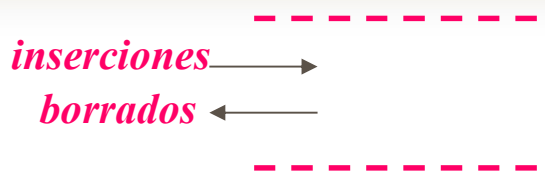
Pilas

- En una pila las inserciones y borrados tienen lugar en un extremo denominado *extremo*, *cabeza* o *tope*
- El modelo intuitivo de una pila es un conjunto de objetos apilados de forma que:
  - Al añadir un objeto se coloca encima del último añadido
  - Al quitar un objeto hay que quitar antes los que están encima de el
- Otros nombres de las pilas son:
  - Listas LIFO (*Last Input First Output*)
  - Listas *pushdown* (empujadas hacia abajo)

Pilas



Pilas




```
struct pila
{
 char nombre[20];
 struct pila *sig;
};
```

- `struct pila * nuevoElemento();`
- `int vacia(struct pila *cabeza);`
- `void verCima(struct pila *cabeza, char *nombre);`
- `void apilar(struct pila **cabeza, char *nombre);`
- `void desapilar(struct pila **cabeza, char *nombre);`

Pilas.NuevoElemento.Vacia.VerCima

```
struct pila * nuevoElemento()
{
 return((struct pila *)malloc(sizeof(struct pila)));
}

int vacia(struct pila *cabeza)
{
 if (cabeza ==  NULL)
  return 1;
 return 0;
}

void verCima(struct pila *cabeza, char *nombre)
{
 strcpy(nombre, cabeza->nombre);
}
```

Tiene que haber al menos un elemento en la pila

Pilas.Apilar (*push*)

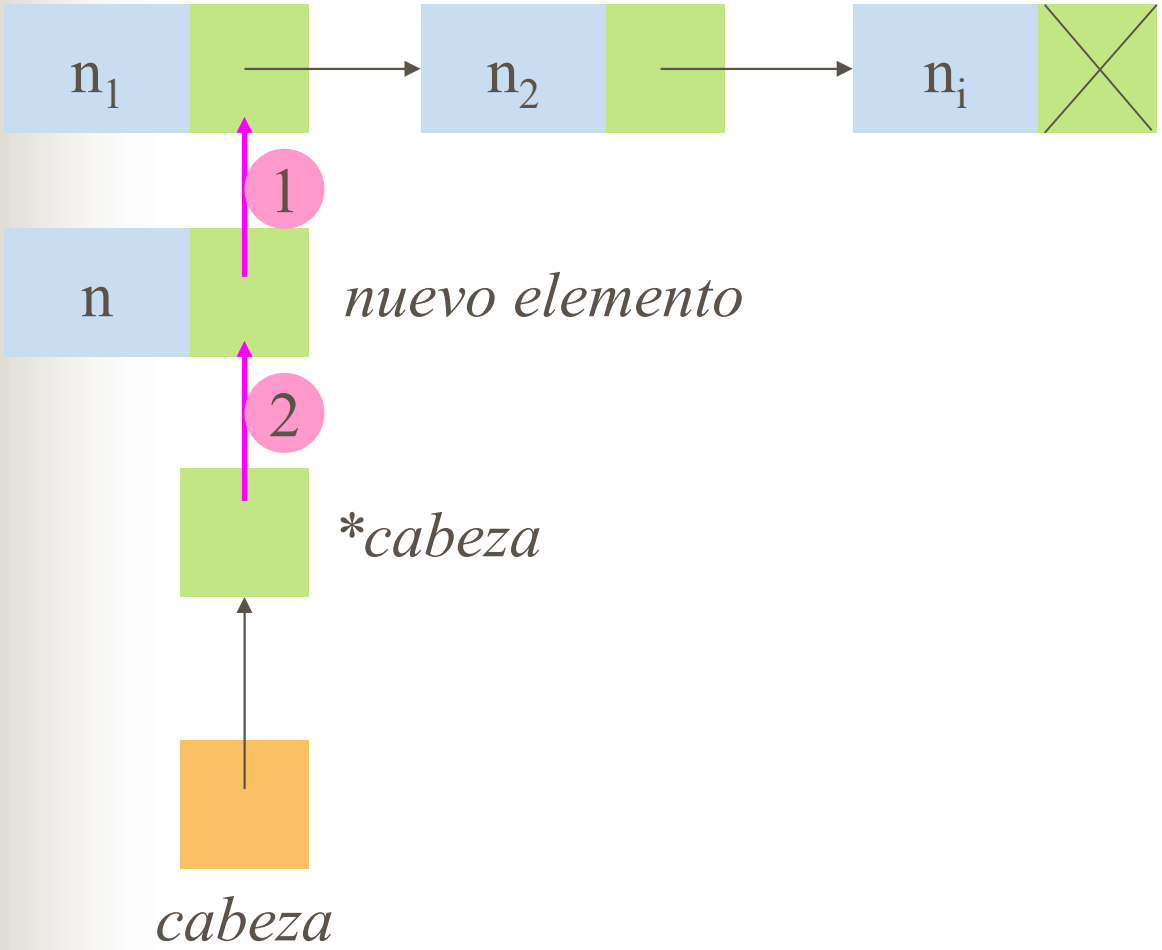
```
void apilar(struct pila **cabeza, char
   *nombre)
{
 struct pila *nuevo = NULL;

 nuevo = nuevoElemento();
 strcpy(nuevo->nombre, nombre);

 nuevo->sig = *cabeza;
 *cabeza = nuevo;
}
```

1

2

Es una inserción por delante en una lista!

Pilas.Apilar (*push*)



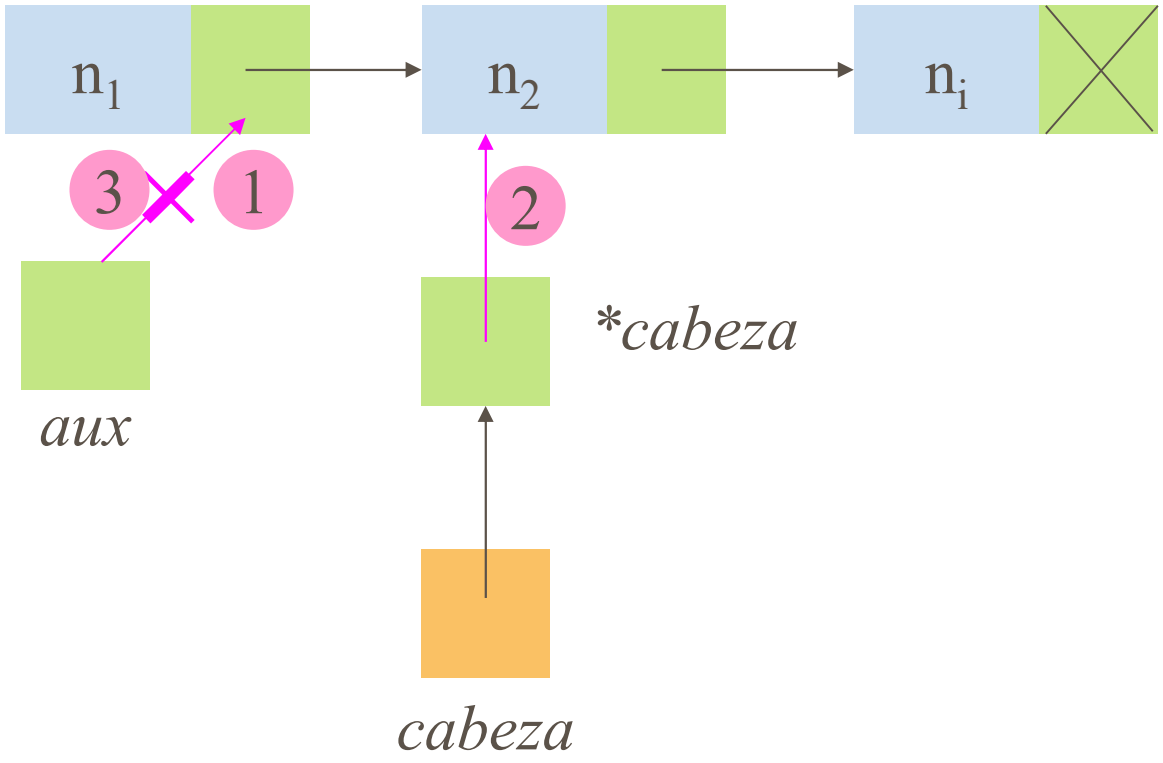
Pilas.Desapilar (*pop*)

```
void desapilar(struct pila **cabeza, char *nombre)
{
 struct pila *aux;

 aux = *cabeza;  1
 strcpy(nombre, (*cabeza)->nombre);
 *cabeza = aux->sig;  2
 free (aux);  3
}
```

Tiene que haber al menos un elemento en la pila

Es un borrado por delante en una lista!

Pilas.Desapilar (*pop*)



Pilas.Contar nodos

```
int contarNodos(struct pila** cabeza)
{   struct pila* pilaAux = NULL; //PILA AUXILIAR VACÍA
    int nodos = 0;
    char nombre[20];

    while(!vacia(*cabeza))
    {
        desapilar(cabeza, nombre);
        apilar(&pilaAux, nombre);
        nodos ++;
    }
    while(!vacia(pilaAux))
    {
        desapilar(&pilaAux, nombre);
        apilar(cabeza, nombre);
    }
    return nodos;
}
```

Colas

- Una cola es otro tipo especial de lista en la cual los elementos se insertan por el extremo anterior (por el principio) y se suprimen por el posterior (por el final)
- Se conocen también como listas FIFO (*First Input First Output*)
- Las operaciones para una cola son análogas a las de las pilas
- Las diferencias sustanciales consisten en que:
  - Los borrados se hacen por un extremo de la lista y las inserciones por otro
  - La terminología tradicional para colas y pilas no es la misma

Colas

```
struct punto{
  float x;
  float y;
};
struct cola{
  struct punto p;
 struct cola *sig;
};
```

- `struct cola *nuevoElemento();`
- `void insertarCola(struct cola **cabeza, struct punto p);`
- `struct punto extraerCola(struct cola **cabeza);`
- `int contiene(struct cola *cabeza);`

Colas.NuevoElemento.Contiene

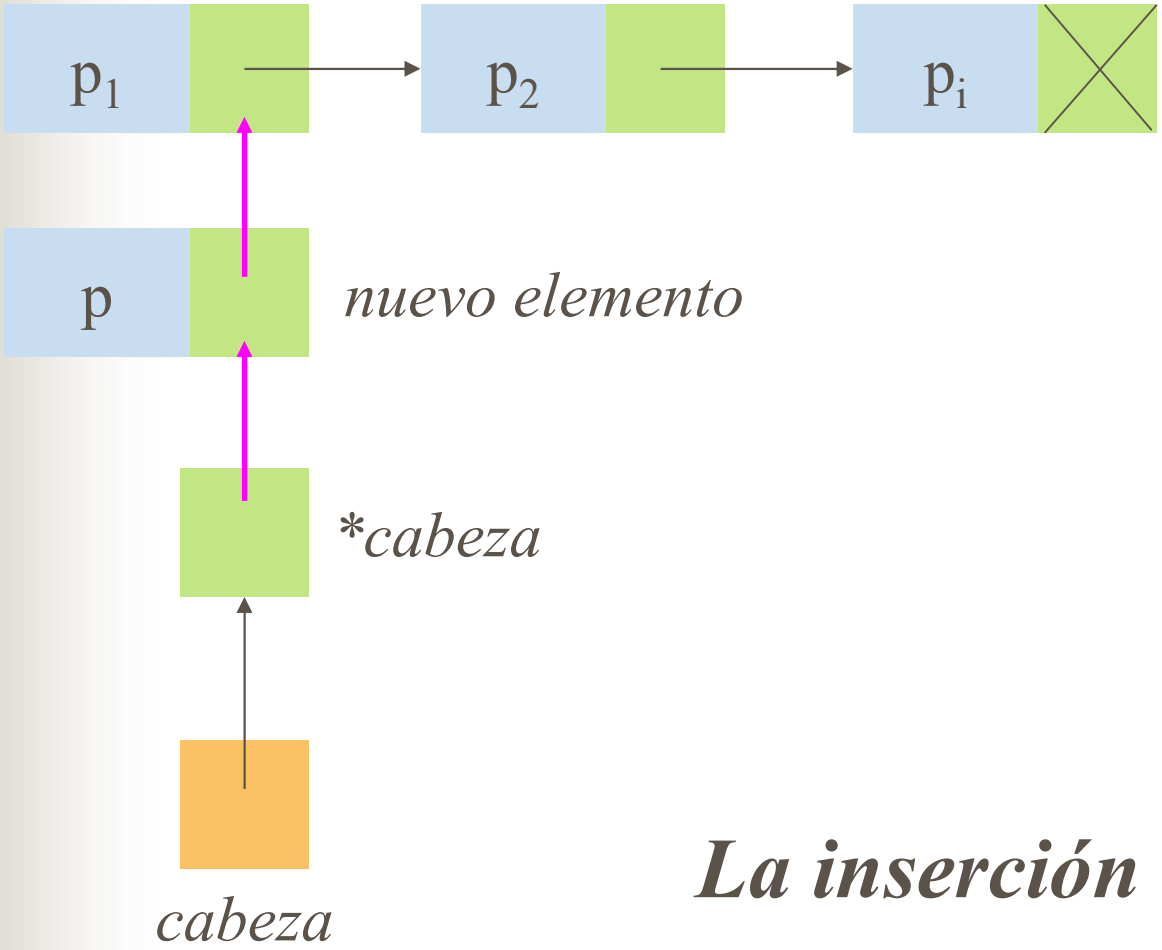
```
struct cola *nuevoElemento()
{
 return ((struct cola *)malloc(sizeof(struct cola)));
}


int contiene(struct cola *cabeza)
{
 if (cabeza == NULL)
  return 0;
 return 1;
}
```

Colas.InsertaCola

```
void insertarCola(struct cola **cabeza, struct
   punto p)
{
 struct cola *nuevo = NULL;
 nuevo = nuevoElemento();
 nuevo->p = p;

 nuevo->sig = *cabeza;
 *cabeza = nuevo;
}
```

Es una inserción por delante en una lista!

Colas.InsertaCola



La inserción es delante

Colas.ExtraerCola

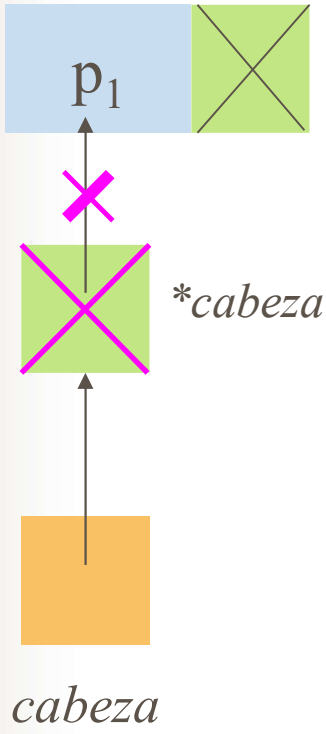

```
struct punto extraerCola(struct cola **cabeza)
{
 struct cola *aux = NULL;
 struct cola *ant = NULL;
 struct punto p;

 if (((*cabeza)->sig) == NULL)//Un solo nodo
   {
    p = (*cabeza)->p;
    free(*cabeza);
    *cabeza = NULL;
    return p;
   }
```

En la cola debe haber al menos un elemento

Colas.ExtraerCola

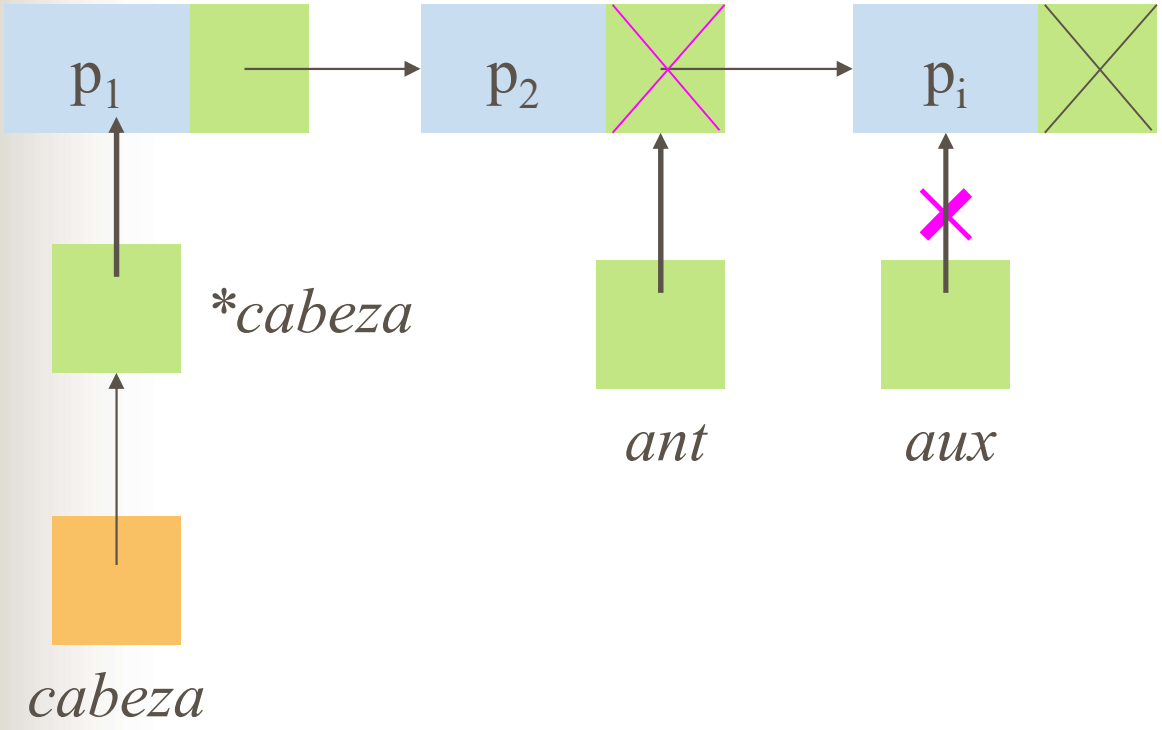
```
else
   {aux = *cabeza;
    while(aux->sig != NULL)
    {
        ant = aux;
        aux = aux->sig;
    }
    p = aux->p;
    free(aux); /* se borra el ultimo*/
    ant->sig = NULL;
    return p;
   }
}
```

Colas.ExtraerCola



Caso 1. Hay un solo elemento

Colas.ExtraerCola



Caso 2. Hay más de un elemento. Se borra el último

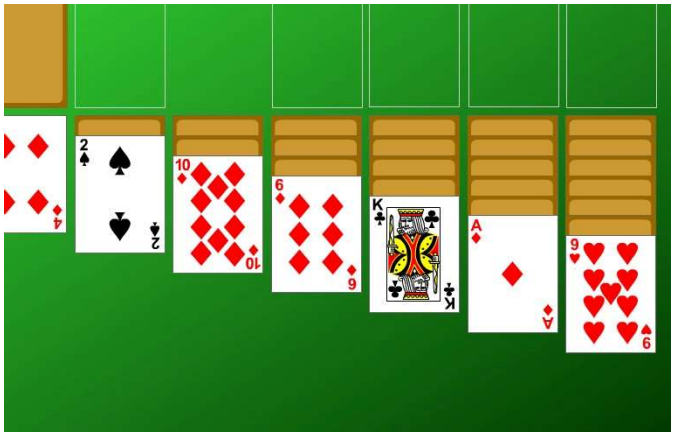
Colas.ContarNodos

```
int contarNodos(struct cola** cabeza)
{   int nodos = 0;
    struct punto p;
    struct cola* colaAux = NULL;

    while(contiene(*cabeza))
    {
        p=extraerCola(cabeza);
        insertarCola(&colaAux, p);
        nodos++;
    }
    *cabeza = colaAux;
    return nodos;
}
```

Aplicaciones de las pilas y las colas

- Las **pilas** son frecuentemente utilizadas en el desarrollo de sistemas informáticos y software en general
  - El sistema de soporte en tiempo de compilación y ejecución de lenguajes como C se utiliza una pila para llevar la cuenta de los parámetros de procedimientos y funciones, variables locales, globales y dinámicas
  - Traducir expresiones aritméticas
  - Cuando se quiere recordar una secuencia de acciones u objetos en el orden inverso del ocurrido
- Con respecto a las **colas**:
  - Representación simulada de eventos dependientes del tiempo, como por ejemplo el funcionamiento de un aeropuerto, controlando partidas y aterrizajes de aviones (cola con prioridad)
  - La CPU asigna prioridades a las distintas tareas que debe ejecutar y las inserta en su cola, para de esta manera realizarlas en el orden correcto (multitareas)
  - Planificación del uso de los distintos recursos del ordenador


HANIHE
PANAMA




Bebidas con
sabor a Cola,
Pila y Lista