Programación 2

Tipos Inductivos (Recursivos)

Estructuras Dinámicas: Punteros y Listas

Tipos inductivos

- tienen, en general, cardinalidad infinita
- contienen valores de "tamaño arbitrario"
  Ejemplos:
    - números naturales
    - listas
- Tenemos por ello un problema para representar estos tipos en computadoras. Concretamente: Cómo se implementan las variables de estos tipos?

Valores

Que significa la declaración

```
T  x;  ?
```

- un espacio fijo en memoria con una dirección (secuencia de bits ubicada en cierta posición de la memoria)
- El espacio debe ser suficiente para contener cualquier valor de tipo **`T`**

Si los valores de tipo **`T`** pueden ser de tamaño arbitrariamente grande, entonces no hay espacio finito suficiente para contener cualquier valor de tipo **`T`**.

Listas

- El caso de las listas (más detallado):

Cómo tendría que ser el espacio reservado para una variable capaz de contener cualquier lista ?

- o bien infinito
- o bien de tamaño ajustable

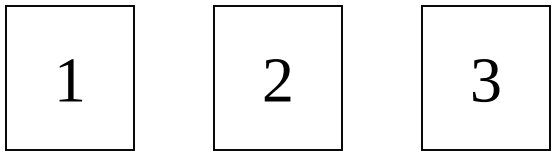
(Cada lista es finita; pero la variable debería poder crecer y contraerse al agregar y quitar elementos de la lista)

Solución:

la lista se extiende "hacia afuera" a lo largo de la memoria ocupando un número de variables. Cómo ?

Listas (cont.)

- Ejemplo: Consideramos [1, 2, 3]
  Cada elemento es representable en una variable común (de tipo **`int`**, por ejemplo):

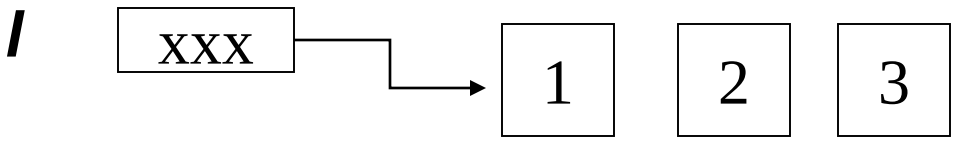


A esta información, hay que agregar otra, que representa la estructura (lineal) de la lista :

- Dónde está el primer elemento ?
- Dónde está el siguiente de cada elemento ?

Listas (cont.)

- Si llamamos *l* a la lista en cuestión, podemos representarla con una variable *l* que:
  - indique dónde está el primer elemento
  - apunte a la variable que contiene el primer elemento
  - contenga como valor la dirección (un nombre de, una referencia a) la variable que contiene el primer elemento



Un nuevo constructor de tipos

Necesitamos un nuevo tipo!!

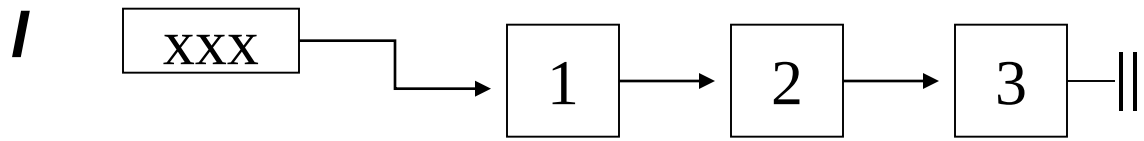
Valores

direcciones
referencias
nombres
indicadores (indicaciones) (en el sentido de lo que se hace con el dedo índice o la flecha)
punteros

- Debe poderse representar la lista vacía, cuyo primer elemento no existe. Necesitamos un valor "que no apunte": **`NULL`**

Secuencialidad

- Además necesitamos representar la estructura secuencial. Para ello, hacemos que cada componente de la lista apunte al siguiente:



***l* = 1 . 2 . 3 . []**

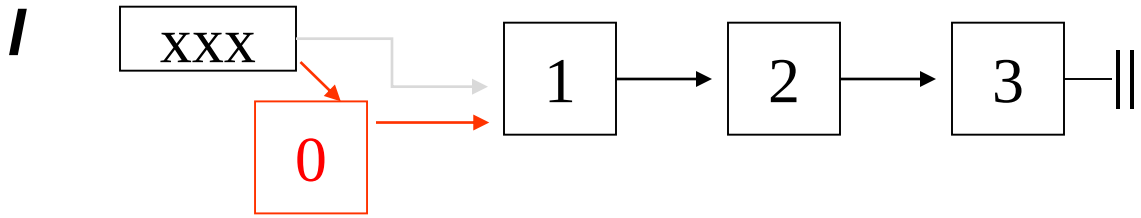
- Tenemos **NODOS** formados por dos componentes (campos):
  i. El elemento propio de la lista (información)
  ii. El puntero al siguiente nodo (posiblemente no existente)
- En **C/C++** podemos representar los nodos como registros

Construcción y destrucción

1. Debe poderse crear y destruir variables en forma dinámica (mediante instrucciones de programa, durante la ejecución de los programas). Por qué ?
    - Debemos poder implementar asignaciones (abstractas) como: **`S = x.S`**
    - En tal caso, la lista contenida en la variable S se agranda en un elemento (comparar con n= n+1) En la representación con nodos y punteros, corresponde crear un nuevo nodo y agregarlo a la lista.

Construcción y destrucción

- nuevo nodo
  - debe ser creado
  - debe ser cargado de información
  - por supuesto, *l* debe ser actualizada



- Igualmente, deben poderse destruir nodos, correspondientemente con las operaciones que borran elementos de listas
  Tenemos pues **ESTRUCTURAS DINÁMICAS**. Su tamaño es gobernado por *instrucciones de programa*.

Estructuras dinámicas

- Hasta ahora todas las estructuras de datos que hemos analizado (arrays, registros) poseen como característica común el tener tamaño fijo, el cual es declarado en tiempo de compilación. Debido a esta restricción es que son llamadas *estructuras estáticas*.

- Pero en ocasiones podría interesarnos manipular estructuras de datos a las cuales no sea necesario especificarle su tamaño. Estas son llamadas *estructuras dinámicas*.

Estructuras dinámicas (cont.)

- La facultad de variar su tamaño es la propiedad característica que claramente distingue las estructuras de datos *dinámicas* de las *estáticas*. Una estructura dinámica puede tanto contraerse como expandirse durante la ejecución de un programa.

- Esto implica que en tiempo de compilación **no** es posible asignarle una cantidad fija de memoria a una estructura dinámica.

Estructuras dinámicas (cont.)

- Una técnica que se usa para resolver este problema consiste en realizar una *asignación dinámica* de memoria.

- Es decir, en forma explícita se va asignando memoria para los componentes individuales de la estructura de datos a medida que éstos son creados durante la ejecución del programa.

Punteros o Referencias

Al crear un componente dinámico, el compilador va a almacenar en una cantidad fija de memoria la *dirección* donde se encuentra tal componente.

A nivel del lenguaje esto se trasluce en una distinción entre *datos* y *referencias a datos*. Se introducen así tipos de datos cuyos valores son direcciones de memoria en las cuales están almacenados otros datos. Variables de estos tipos son llamados *punteros*, o *referencias*.

Punteros o Referencias (cont.)

La notación que se utiliza para este fin es:

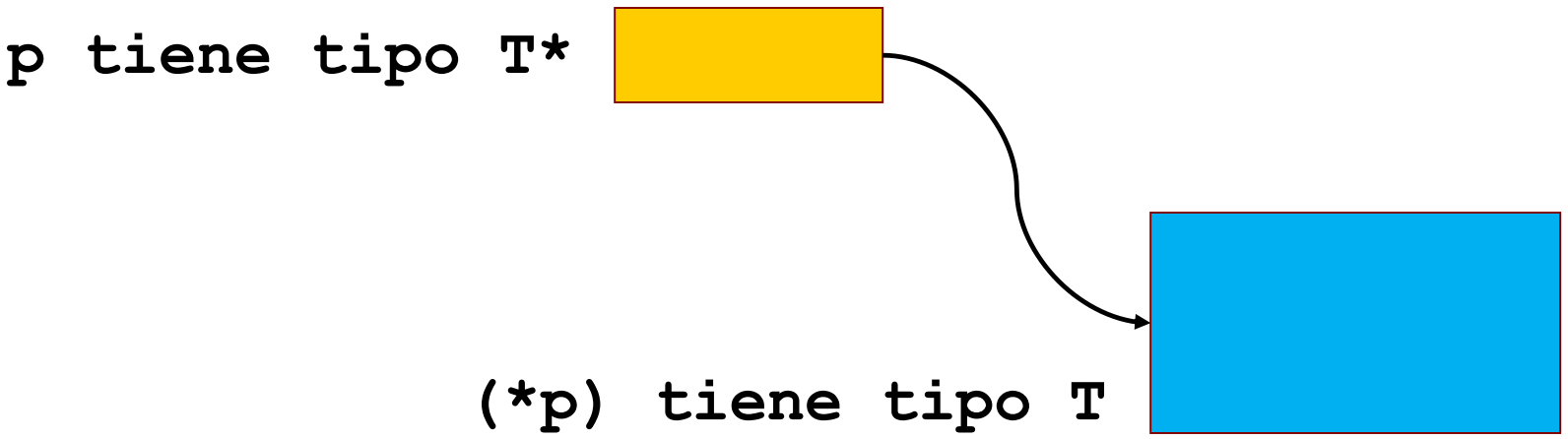
```
T *<variable>;
```

Una declaración de esta clase expresa que la variable declarada es un puntero a datos de tipo **`T`**. El asterisco **`*`** se debe leer como "*puntero a*".

Los valores de tipo puntero se generan cuando a un dato se le asigna memoria en forma dinámica.

Punteros o Referencias (cont.)

Gráficamente,



Ejemplo

La declaración:

```
int *ptr1, *ptr2;
```

define dos punteros a direcciones de memoria que contienen un valor entero.

- Es de hacer notar que los valores de estos punteros son inicialmente **indefinidos**.
- Y que **`*`** no distribuye.

Operador de dirección

El operador &, u *operador de dirección*, es un operador unario que retorna la dirección de su operando.

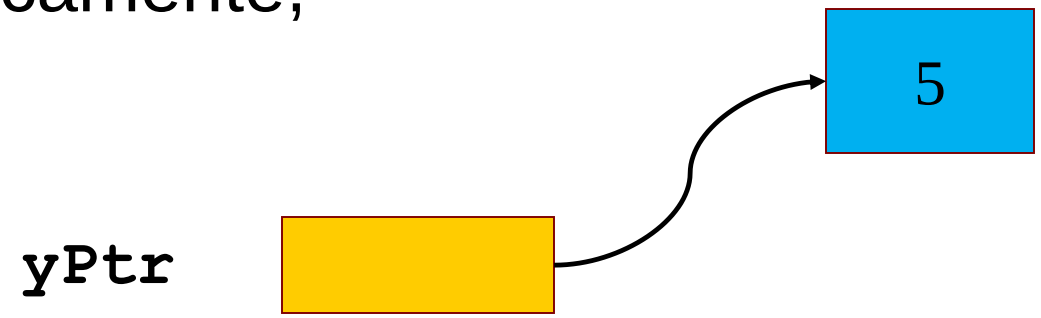
Ejemplo:

```
int y = 5;
int *yPtr;
```

La sentencia **`yPtr = &y`** asigna la dirección de **`y`** a la variable **`yPtr`** (**`yPtr`** "apunta" a **`y`**)

Operador de dirección (cont.)

Gráficamente,




```
int y = 5;
int* yPtr;
yPtr = &y;
```

o

Asignación dinámica de memoria

En C la asignación dinámica de memoria se realiza usando los procedimientos estándar de memoria **`malloc y free`**. Considere la declaración:

```
nomTipo *ptr;
```

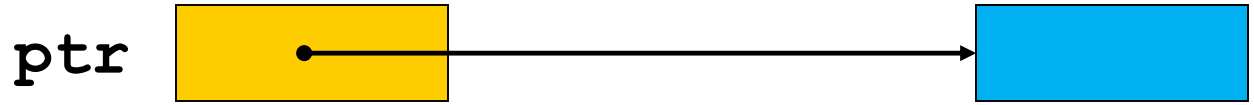
La siguiente sentencia asigna en forma dinámica un objeto **`nomTipo,`** regresa un puntero **`void`** al objeto y asigna dicho puntero a **`ptr:`**

```
ptr = malloc(sizeof(nomTipo));
```

Asignación dinámica de memoria (cont.)

En C++, la sentencia

```
ptr = new nomTipo;
```

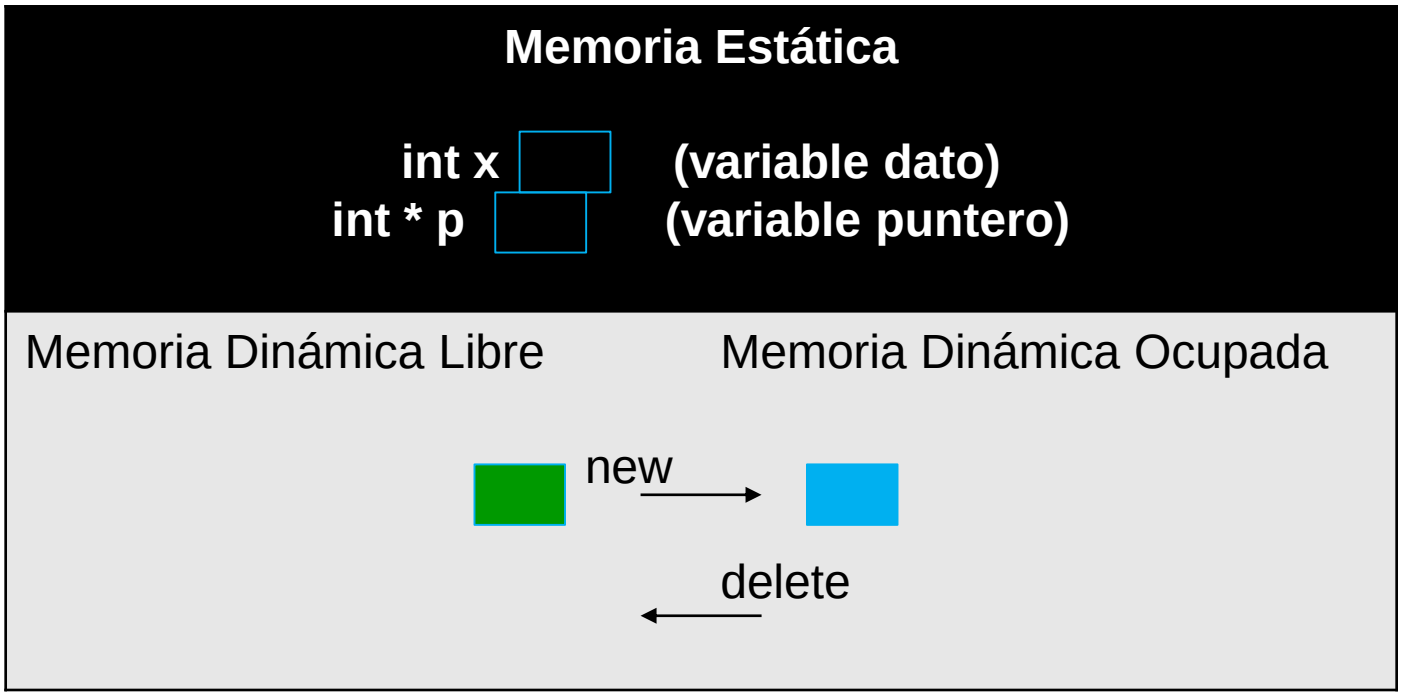


asigna memoria para un objeto del tipo **`nomTipo`**.

El operador **`new`** crea automáticamente un objeto del tamaño apropiado y regresa un puntero del tipo apropiado. Si mediante **`new`** no se puede asignar memoria, se regresa un apuntador nulo.

Asignación dinámica de memoria (cont.)

Administración de la memoria estática y dinámica de un programa



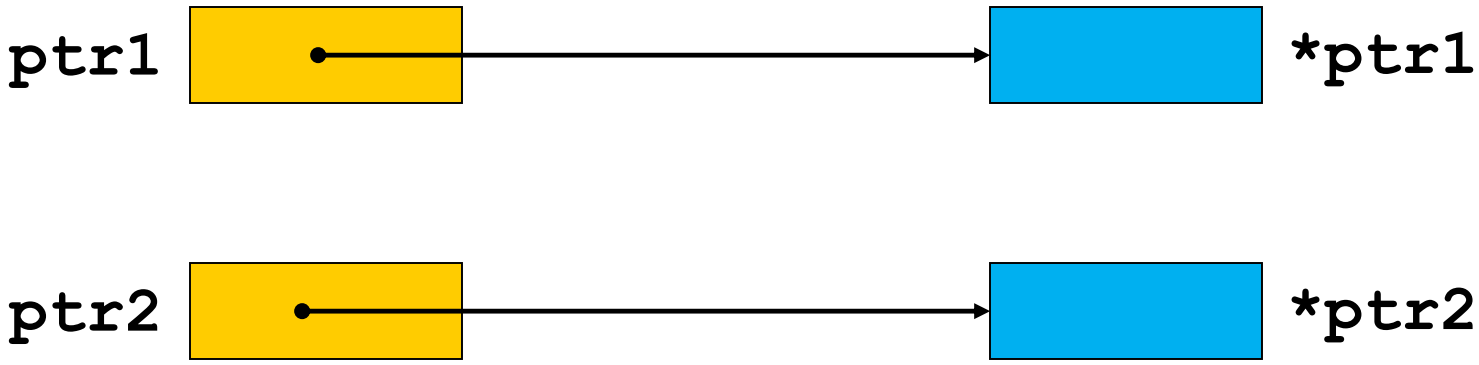
Ejemplo (cont.)

Al ejecutar las sentencias

```
ptr1 = new int;
ptr2 = new int;
```

se obtienen las direcciones de dos lugares de memoria (actualmente sin usar) apropiados para almacenar enteros. Una dirección será almacenada en **`ptr1`** y la otra en **`ptr2`**.

Ejemplo (cont.)



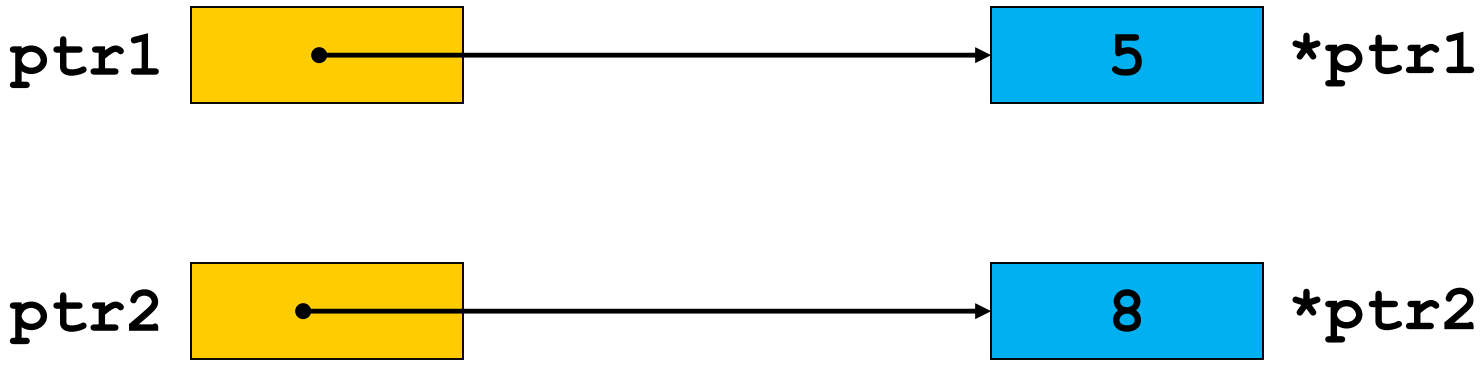
Las variables enteras pueden ser accedidas, pero solo a través de los punteros, usando la notación **`*ptr1`** y **`*ptr2`**.

Ejemplo (cont.)

Las siguientes son sentencias válidas:

```
*ptr1 = 5;
*ptr2 = *ptr1 + 3;
```

y tienen este efecto:



Operaciones sobre Punteros

El valor de los punteros (direcciones de memoria) depende del particular sistema de computación. Esto causa que la manipulación de punteros sea hecha en forma restringida.

Por ejemplo,

- punteros sólo pueden ser comparados por igualdad.
- Variables de punteros no pueden ser leídos ni escritos desde o hacia un archivo de texto. Esto es, punteros no tienen representación textual.
- No manejaremos aritmética de punteros.

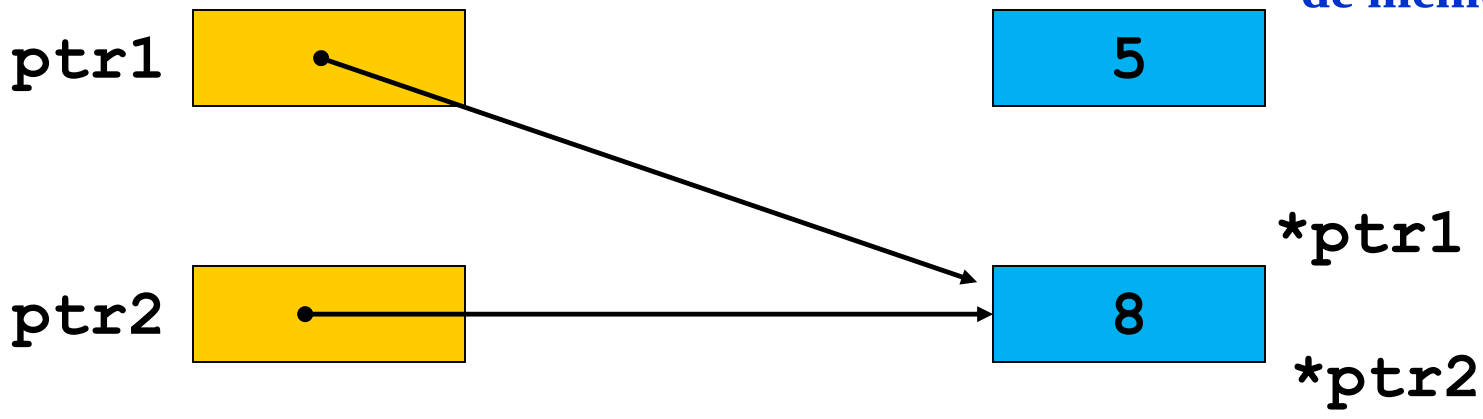
Operaciones sobre Punteros (cont.)

Es válido realizar asignaciones de la forma:

`ptr1 = ptr2;` **(comparar con *ptr1 = *ptr2)**

para punteros del mismo tipo. Luego de esta asignación **`ptr1`** va apuntar a la misma dirección que **`ptr2`**, o sea, ahora **`*ptr1 y *ptr2`** son la misma variable.

(notar desperdicio de memoria)



Liberación de memoria

C++ provee un procedimiento estándar llamado **`delete`** que permite liberar espacio de memoria dinámica que ya no se desee utilizar. La memoria liberada queda disponible para que pueda ser reutilizada por futuras sentencias **`new`**.

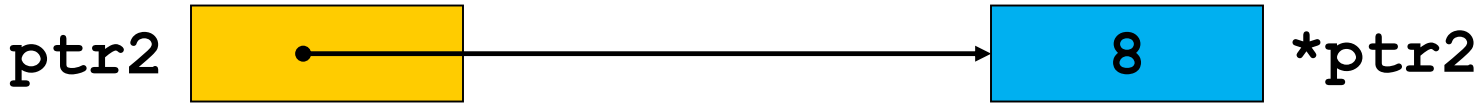
Por ejemplo,

```
delete ptr2;
```

va a liberar el espacio de memoria apuntada por el puntero **`ptr2`**.

Liberación de memoria (cont.)

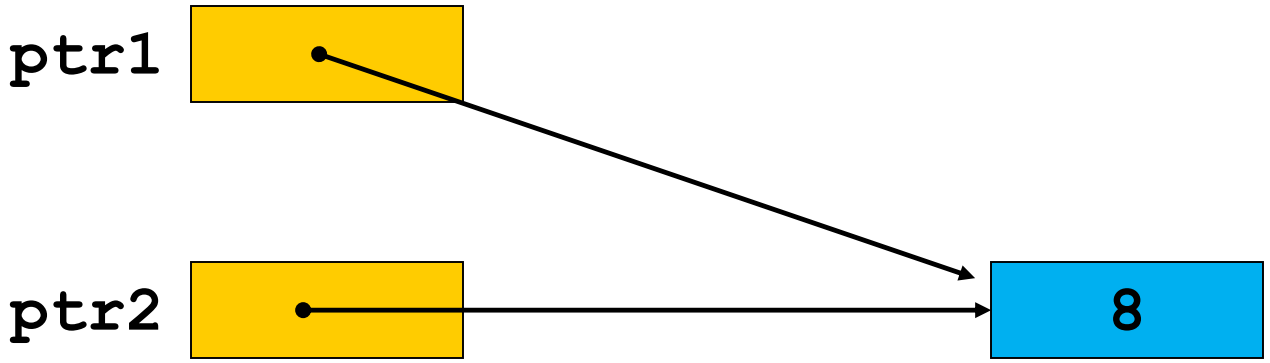
Notar que luego de tal acción el puntero **`ptr2`** sigue existiendo, pero su valor ahora es **indefinido**.



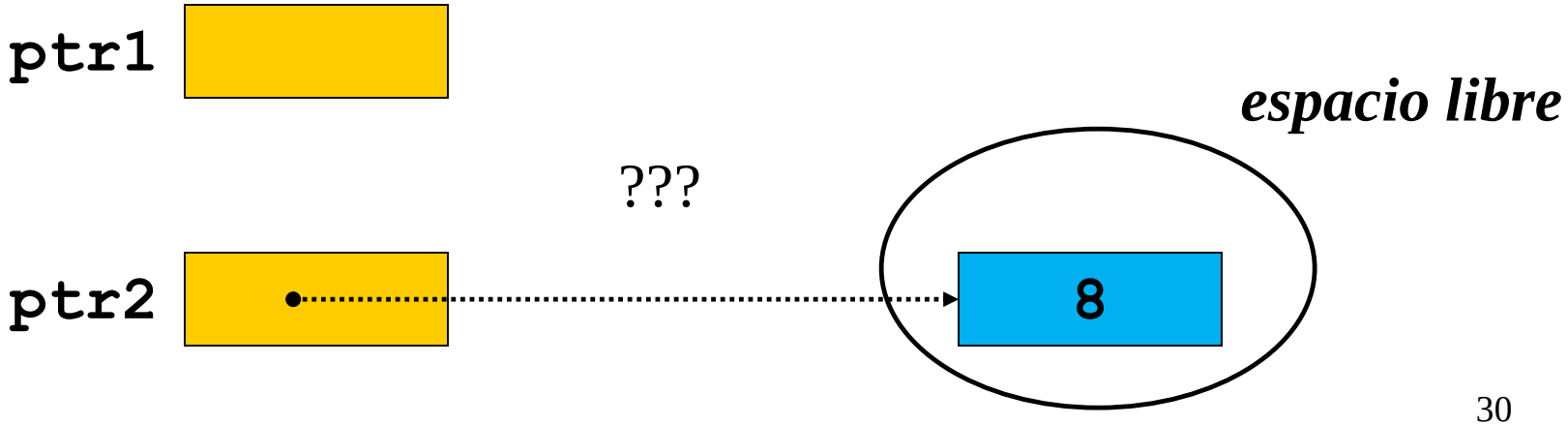

```
delete ptr2;
```

Liberación de memoria (cont.)

Problemas de Alias.



delete ptr1;



La constante NULL

Punteros con valor indefinido son fuente común de problemas. Por ello C/C++ incluyen una constante, llamada **`NULL`**, la cual puede ser asignada a cualquier puntero para indicar que éste ya no apunta a ninguna dirección en particular.

Por ejemplo, podemos hacer:

```
ptr1 = NULL;
```

La constante NULL (cont.)

Notar que un puntero con valor NULL es diferente a uno con valor indefinido.

Por ejemplo, un puntero con valor NULL puede ser comparado por igualdad o desigualdad con otro puntero.

En cambio, un puntero con valor indefinido **nunca** debe ser usado en una comparación ni intentar acceder a través de él a la memoria; hacerlo produciría un error de ejecución o daría como resultado un valor indefinido.

La constante NULL (cont.)

Por ejemplo, si **`ptr1`** no es indefinido podemos escribir lo siguiente:

```
if (ptr1 == NULL)
   ...;
else
   ... se puede acceder a *ptr1;
```

Recapitulando

Hemos visto cuatro diferentes formas de cambiar la dirección almacenada en un puntero:

- Usar el procedimiento estandar **`new`**.
- Asignar la dirección contenida en otro puntero del mismo tipo.
- Usar el operador & (tener cuidado!).
- Asignar el valor NULL.

Arreglos/Vectores

¿Cómo se definen arreglos de un tamaño determinado por una constante (conocido en tiempo de compilación)?

int arreglo[cte];

// de 0 a cte-1

¿Cómo se definen arreglos de un tamaño determinado por un valor variable (en tiempo de ejecución)?

int * arreglo = new int[var];

// de 0 a var-1

¿Cómo operar con arreglos?

arreglo[i]

Punteros a registros

Registos u otros tipos de datos estructurados también pueden ser apuntados por punteros. Veamos un ejemplo:

```
struct Regdatos {
        char * nombre;
        float salario;
};
typedef Regdatos * Regpunt;
```

Podemos entonces declarar:

```
Regpunt regpersona;
```

Punteros a registros (cont.)

Al ejecutar la sentencia

```
regpersona = new Regdatos;
```

regpersona pasará a tener la dirección de memoria donde se encuentra alojado el nuevo registro. Podemos entonces manipular los campos del registro a través del puntero y usando el operador **->**:

```
regpersona -> nombre = "Juan";
regpersona -> salario = 3500.50;
(*regpersona).salario = 3500.50;
```

Estructuras dinámicas

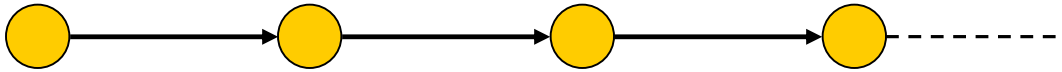
Luego de haber introducido punteros pasaremos ahora a estudiar estructuras de datos cuyos componentes están enlazados precisamente por punteros.

La primera estructura que vamos a analizar son las llamadas *listas* o *secuencias encadenadas*.

Listas encadenadas

Esta es la forma mas simple de enlazar o relacionar un conjunto de elementos.

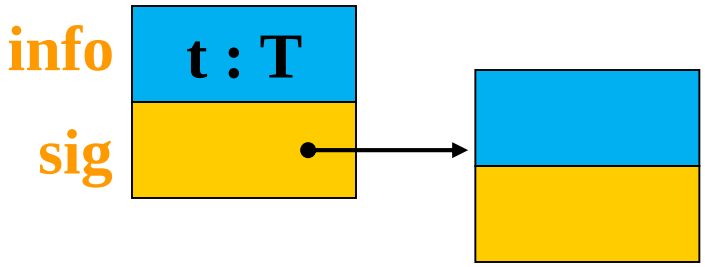
Una *lista encadenada* es una estructura lineal formada por una cadena de registros cada uno de los cuales posee una referencia a su sucesor en la lista.



Listas encadenadas (cont.)

En C++ podemos definir, también, los registros componentes de una lista encadenada como sigue:

```
struct nodoLista {
        T info;
        nodoLista *sig;
 };
```




```
typedef nodoLista * Lista;
```

(Notar la autoreferencia)

Listas encadenadas (cont.)

En esta definición hemos asumido que los datos contenidos en cada registro de la lista son de tipo **`T`**.

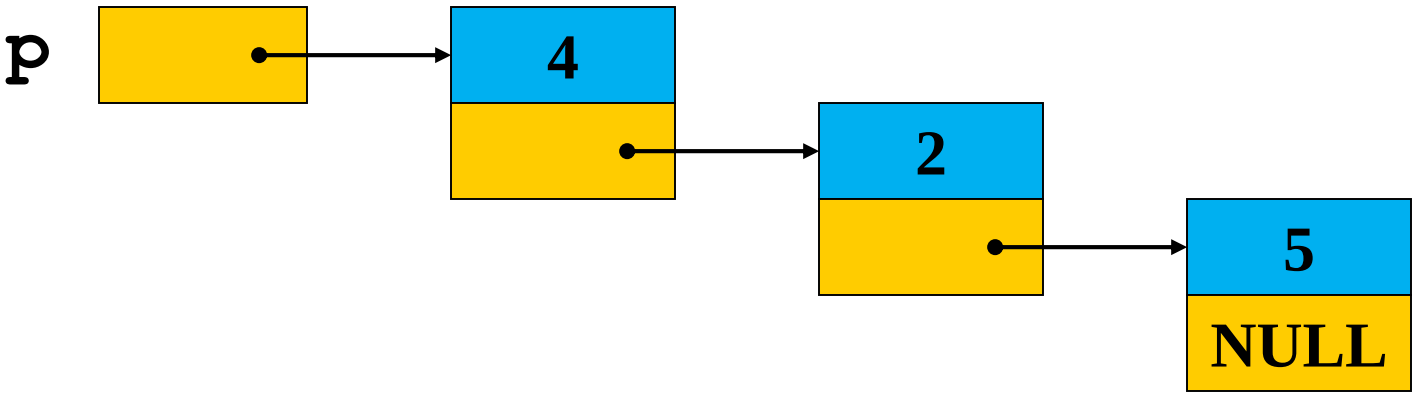
Para acceder a una lista de este tipo vamos a tener una variable puntero

```
Lista p;
```

la cual va a contener la dirección del comienzo (esto es, del primer registro) de la lista.

Listas encadenadas (cont.)

Supongamos, por ejemplo, que **`T`** es **`int.`** Esta entonces sería una situación posible:



Notar que el puntero **`sig`** del último elemento de la lista tiene valor **`NULL`**.

Operaciones sobre listas

Veamos ahora algunas operaciones para manipular listas. Consideremos dada la siguiente declaración de punteros:

```
Lista p, q, r;
```

donde asumimos que `p` apunta al comienzo de la lista.

La operación más simple es la ***creación*** de una lista vacía:

```
p = NULL;
```

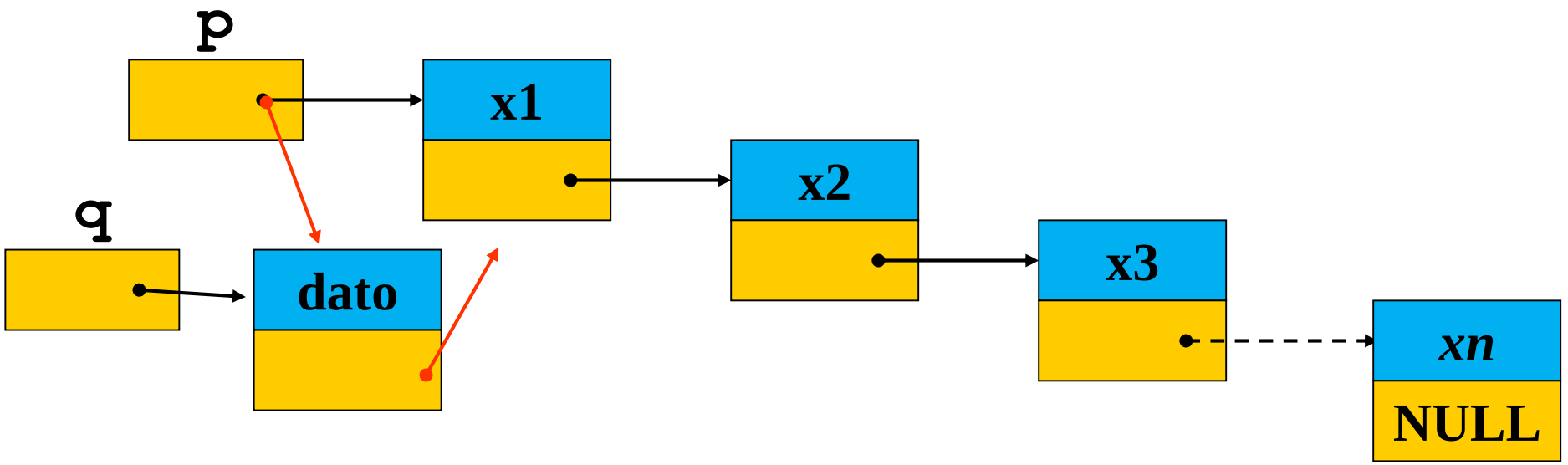
Es la lista vacía ?

- Dada una lista isEmpty retorna true si la lista está vacía y false sino.

```
bool isEmpty (Lista p){
     return (p == NULL);
}
```

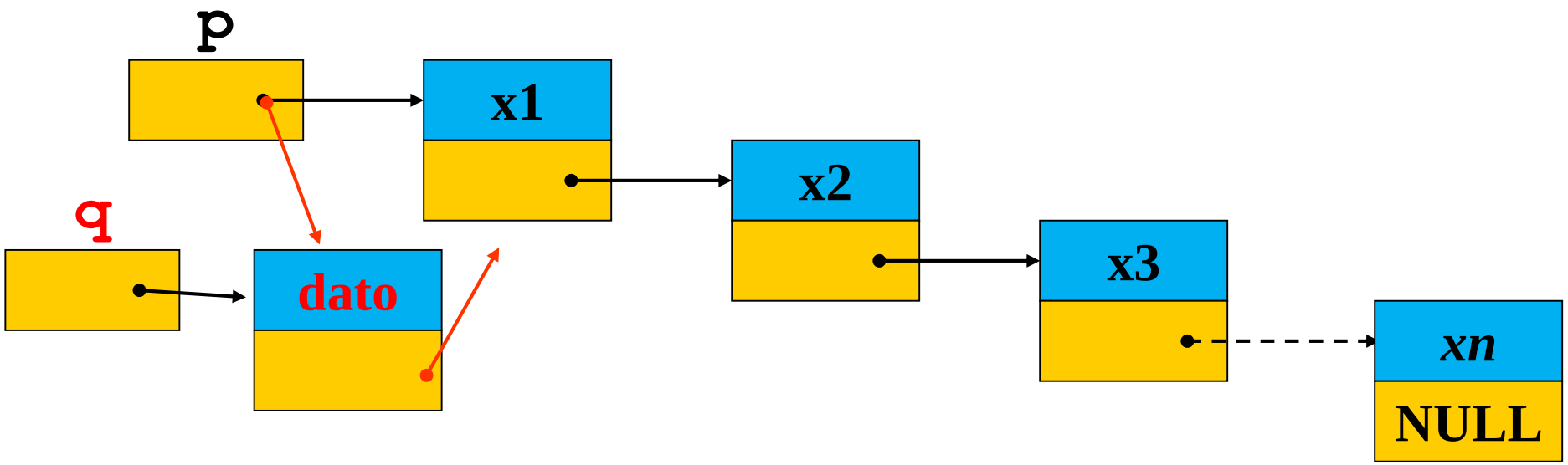
Insertar en una lista

- Dada una lista, deseamos agregar al comienzo de la misma un nuevo registro. Supongamos que los datos a ser colocados en el nuevo registro están en una variable **`dato`** de tipo **`T`**.



Insertar en una lista

```
q = new nodoLista;
q -> info = dato; //(*q).info = …
q -> sig = p;
p = q;
```



Insertar en una lista (cont.)

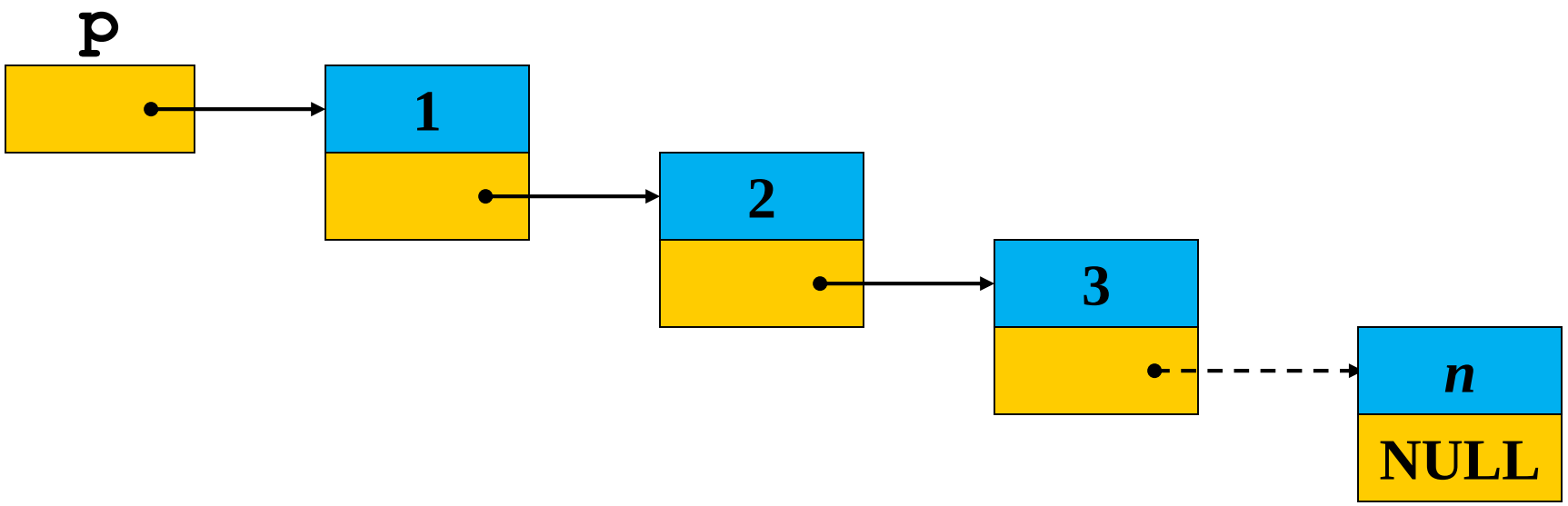
Usando repetidamente el procedimiento de insertar un elemento al comienzo de la lista podemos *generar* una lista de *n* elementos. Supongamos que **T** es **int**.

```
p = NULL;
while (n > 0){
  q = new nodoLista;
  q -> info = n;
  q -> sig = p;
  p = q;
  n = n - 1;
}
```

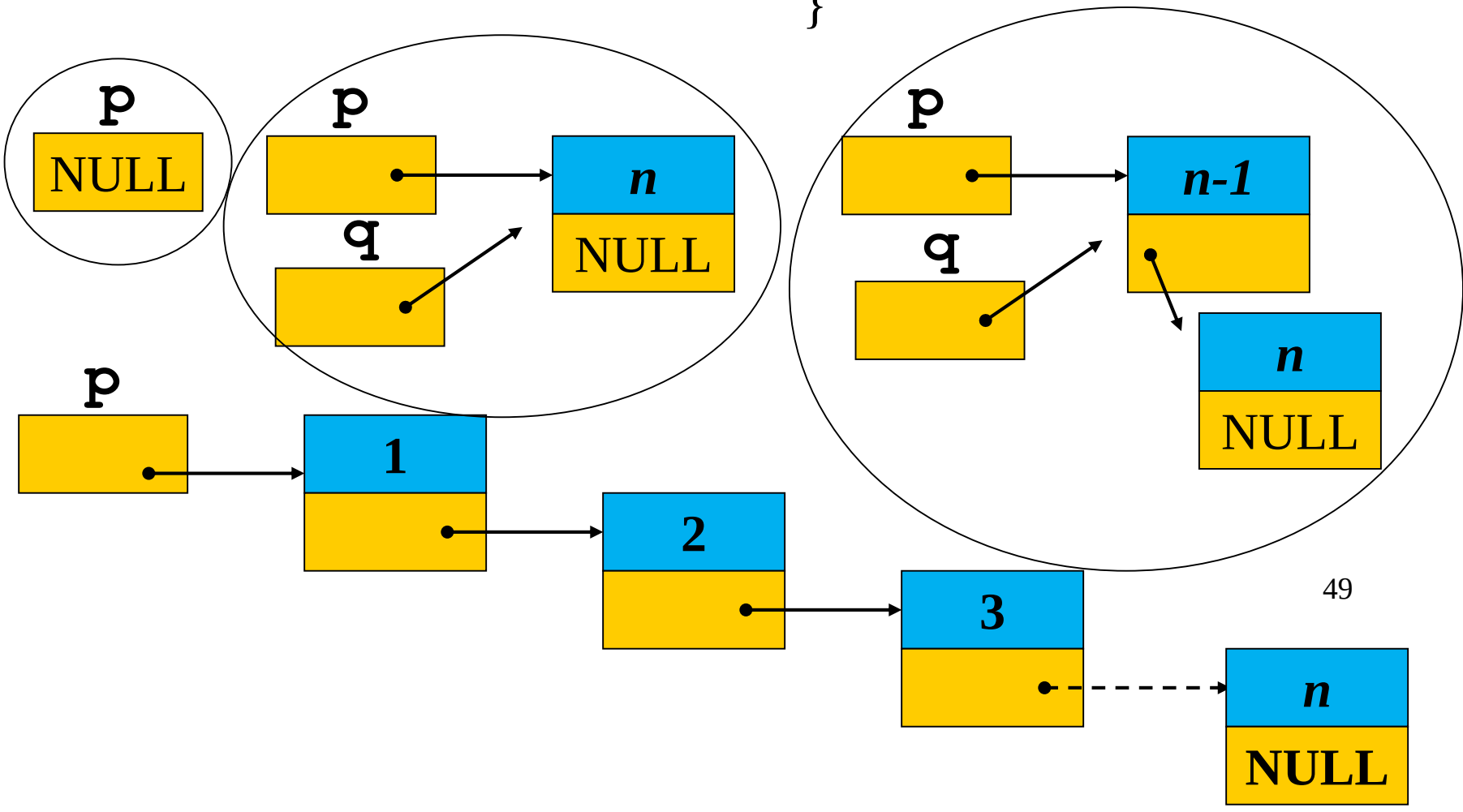
¿Cómo queda la lista p?

Insertar en una lista (cont.)

La lista generada es la siguiente:



```
p = NULL;
while (n>0){
   q = new nodoLista;
   q -> info = n;
   q -> sig = p;
   p = q;
   n = n - 1;
}
```



Insertar en el medio de una lista

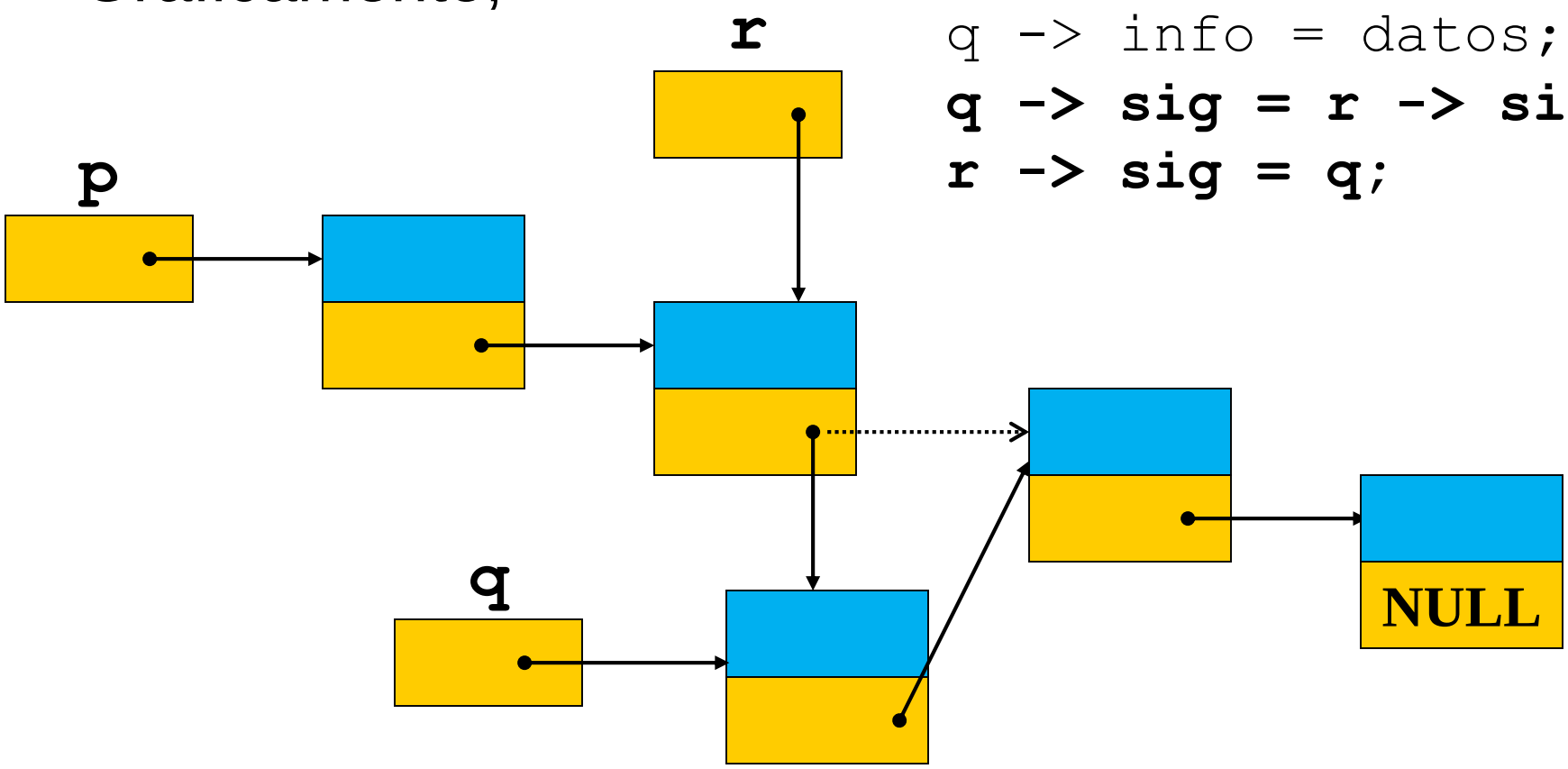
- En ciertas ocasiones nos puede interesar agregar un nuevo registro en el medio de una lista. Un caso es el de insertar *luego* de aquel registro apuntado por **`r de tipo Lista`**.

```
q = new NodoLista;
q -> info = datos;
q -> sig = r -> sig;
r -> sig = q;
```

Insertar en el medio de una lista (cont.)

Gráficamente,

```
q = new NodoLista;
q -> info = datos;
q -> sig = r -> sig;
r -> sig = q;
```



Insertar en el medio de una lista (cont.)

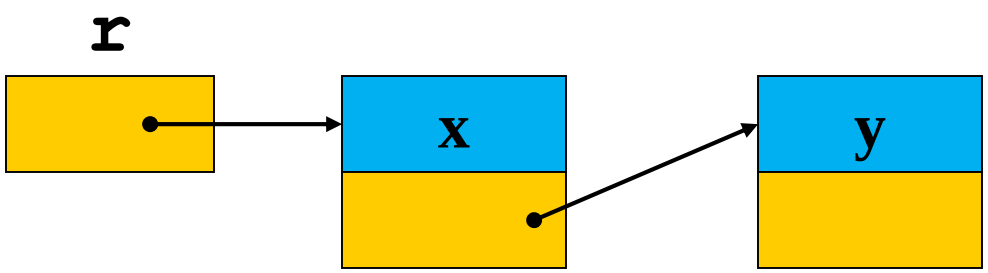
Otro caso es el de insertar *antes* de aquel registro apuntado por **`r de tipo Lista`**.

```
q = new nodoLista;
*q = *r;
r -> info = dato;
r -> sig = q;
```

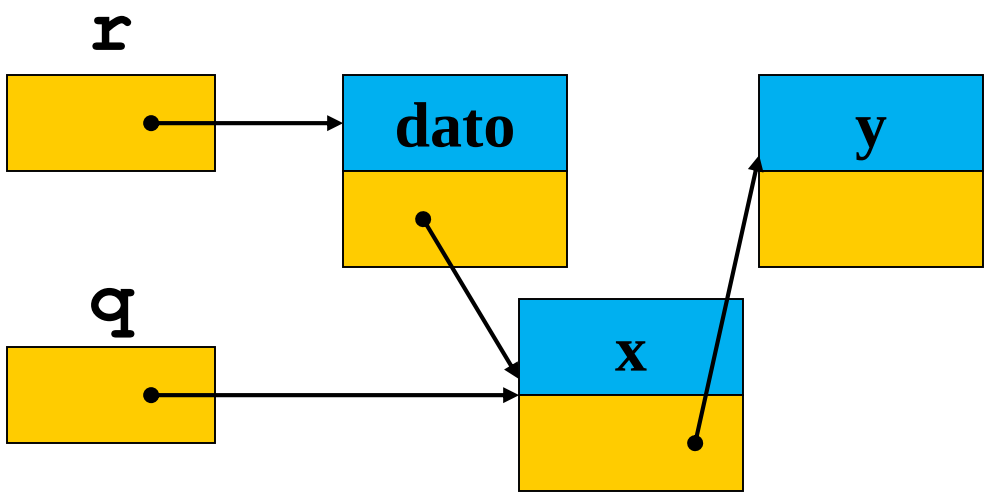
O sea, copiamos el contenido del registro **`*r`** al nuevo registro **`*q`** y luego cargamos **`*r`** con los nuevos datos.

Insertar en el medio de una lista (cont.)

Esto es, si originalmente tenemos:



luego de insertar queda:



Borrar en una lista

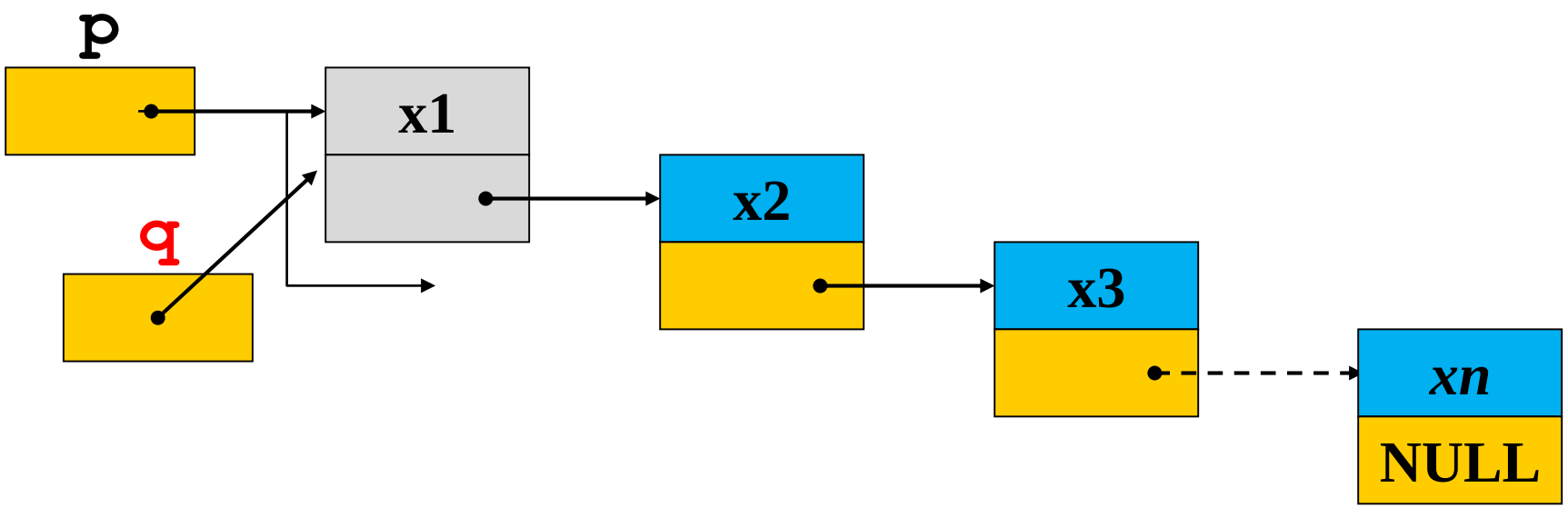
En lo que respecta a borrar elementos de una lista, analizaremos ahora los casos más simples:

- borrar el primer elemento (suponiendo que la lista no es vacía).
- borrar el sucesor de un elemento (suponiendo que no es el último).

Borrar en una lista (cont.)

Para borrar el primer elemento de la lista hacemos:

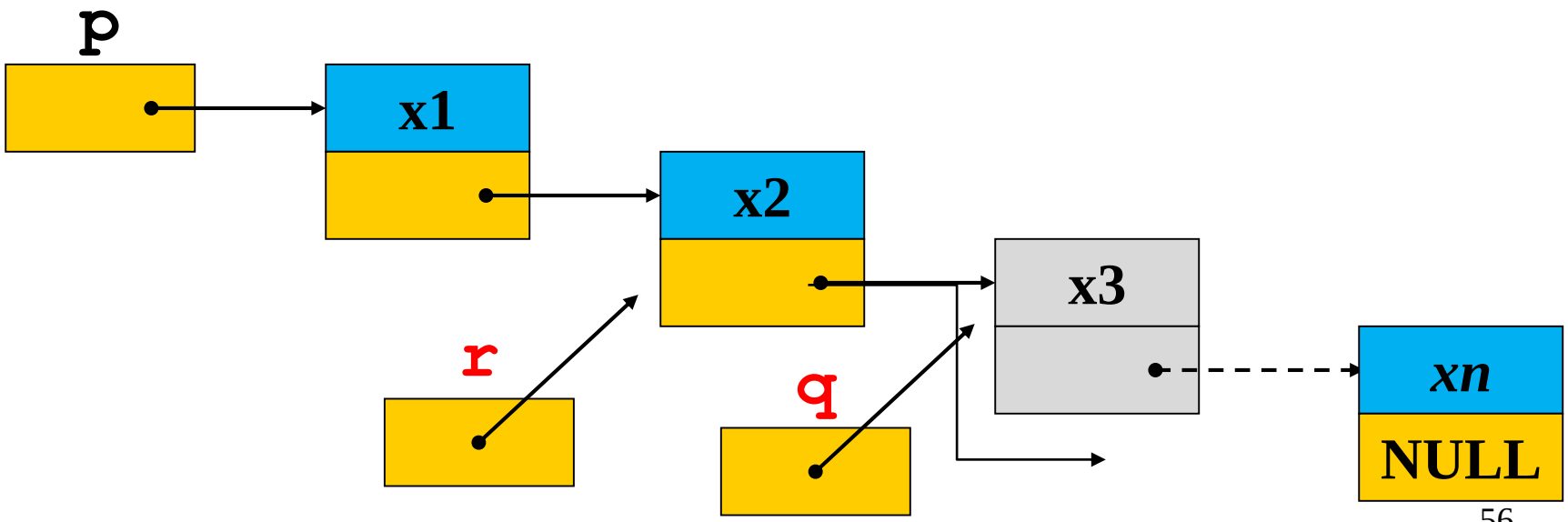
```
q = p;
p = p -> sig;// borrado lógico
delete q;     // borrado físico
```



Borrar en una lista (cont.)

Mientras que para borrar el sucesor del registro apuntado por **`r de tipo Lista`**:

```
q = r -> sig;
r -> sig = q -> sig;
delete q;
```

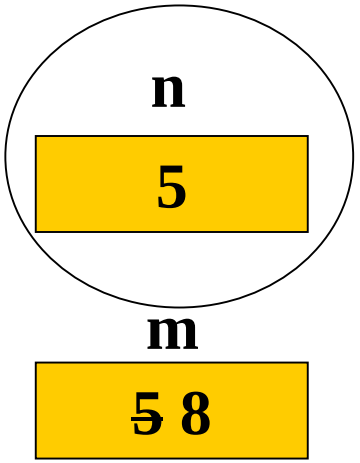


Salto de Tema

Pasaje de parámetros a
funciones y procedimientos
por valor
y
por referencia

Pasaje de parámetros por "*valor*"

```
void P (int);
main() { int n = 5;
         P(n); cout << n;}
void P (int m) { m = 8; }
```



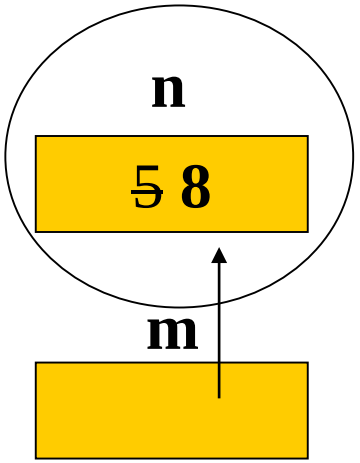
"el valor impreso es 5 (no 8)"

En C y en C++ el pasaje de parámetros por defecto (siempre) es por valor: "al pasar el argumento se realiza una copia del mismo y no se modifica el valor del parámetro con el que se invocó a la función (o procedimiento)"

Pasaje de parámetros por "***referencia***"

```
void P (int *);
main() { int n = 5;
         P(&n); cout << n;}
void P (int * m) { *m = 8; }
```

"el valor impreso es 8 (no 5)"

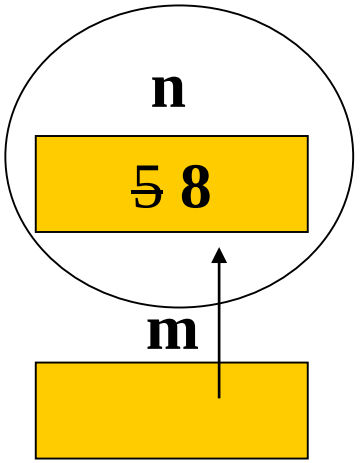


Cuando se desea que una variable v pasada como parámetro pueda modificarse se pasa la dirección (&) de la misma (pasaje por referencia). Luego, en la función (o procedimiento) puede modificarse el valor de v con el uso del operador (*).

Nota: &n se pasa por valor (en C el pasaje es siempre por valor).

Pasaje de parámetros por "***referencia***" (cont.)

```
void P (int &);
main() { int n = 5;
         P(n); cout << n;}
void P (int &m) { m = 8; }
```



"el valor impreso es 8 (no 5)"

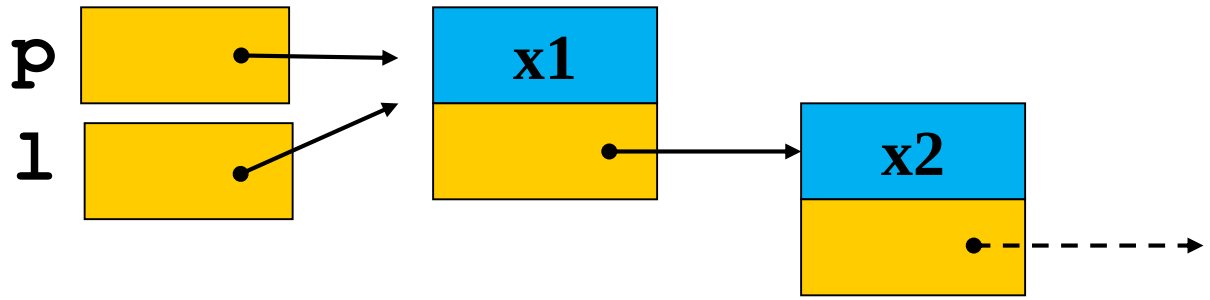
Esta es una forma alternativa de hacer un pasaje por referencia, permitida por C++ (no por C). Usaremos normalmente esta forma de aquí en más.

Ejemplos simples con Listas

Considerar qué pasa si se invoca a los siguientes procedimientos con una lista no vacía p:

```
void P1 (Lista l) { l = NULL; }
void P2 (Lista &l) { l = NULL; }
void P3 (Lista &l) { l -> sig = NULL; }
void P4 (Lista l) { l -> sig = NULL; }
```

Cómo queda luego p en cada caso? Analice P1 primero luego del pasaje de parámetro:

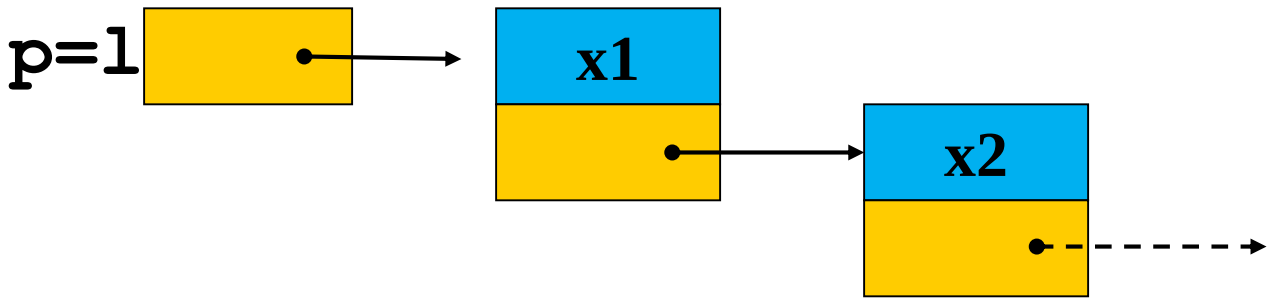


Ejemplos simples con Listas

Considerar qué pasa si se invoca a los siguientes procedimientos con una lista no vacía p:

```
void P1 (Lista l) { l = NULL; }
void P2 (Lista &l) { l = NULL; }
void P3 (Lista &l) { l -> sig = NULL; }
void P4 (Lista l) { l -> sig = NULL; }
```

Cómo queda luego p en cada caso? Analice P2 ahora luego del pasaje de parámetro:

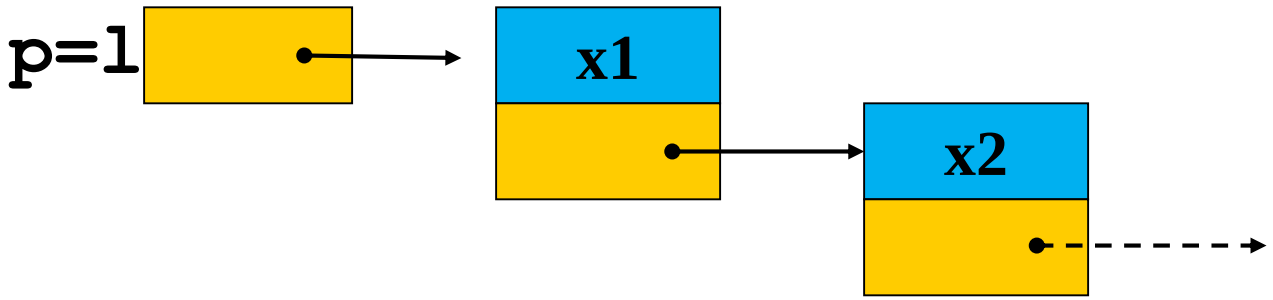


Ejemplos simples con Listas

Considerar qué pasa si se invoca a los siguientes procedimientos con una lista no vacía p:

```
void P1 (Lista l) { l = NULL; }
void P2 (Lista &l) { l = NULL; }
void P3 (Lista &l) { l -> sig = NULL; }
void P4 (Lista l) { l -> sig = NULL; }
```

Cómo queda luego p en cada caso? Analice P3 ahora luego del pasaje de parámetro:

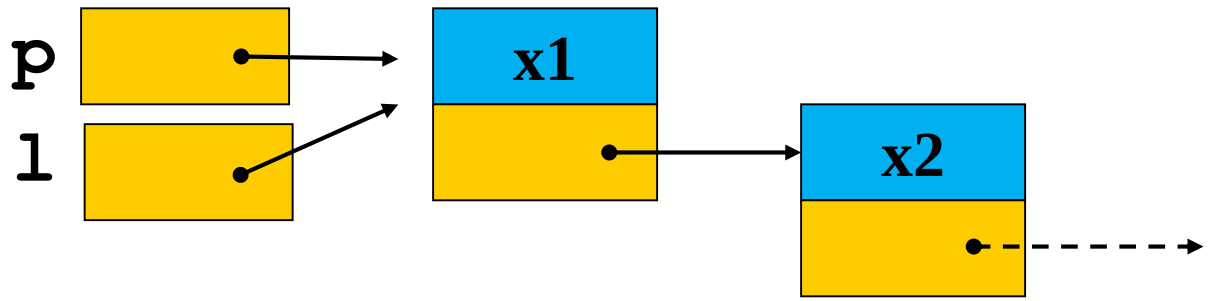


Ejemplos simples con Listas

Considerar qué pasa si se invoca a los siguientes procedimientos con una lista no vacía p:

```
void P1 (Lista l) { l = NULL; }
void P2 (Lista &l) { l = NULL; }
void P3 (Lista &l) { l -> sig = NULL; }
void P4 (Lista l) { l -> sig = NULL; }
```

Cómo queda luego p en cada caso? Analice P4 ahora luego del pasaje de parámetro:



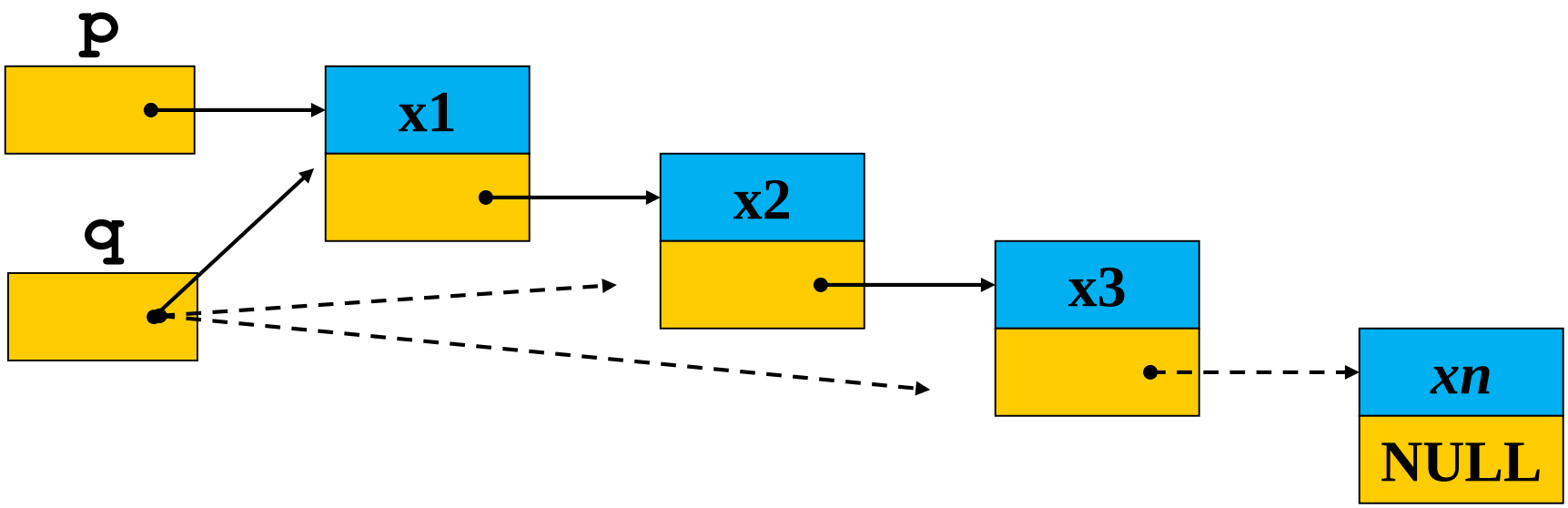
Volviendo a Listas:
Recorrido de una lista

- Ahora deseamos ejecutar una operación (dada por un cierto procedimiento **`PP`**) sobre cada elemento de una lista. Como en las operaciones anteriores, asumimos que **`p`** apunta al comienzo de la lista:

```
q = p;
while (q != NULL){
   PP(q->info);
   q = q -> sig;
}
```

Volviendo a Listas:
Recorrido de una lista

```
q = p;
while (q != NULL){
    PP(q->info);
    q = q -> sig;
}
```



Impresión de una lista

Un ejemplo clásico de recorrida es la impresión de los elementos de una lista, donde la operación **`P`** viene dada por un procedimiento **`ImpDatos`** que imprime los datos de tipo **`T`** contenidos en un registro.

Definiremos el proceso de impresión como un procedimiento que recibe el puntero al comienzo de la lista como un ***parámetro por valor***. Notar que al hacerlo así, no es necesario usar una variable auxiliar **`q`** (cuya utilidad era no perder el puntero al comienzo de la lista).

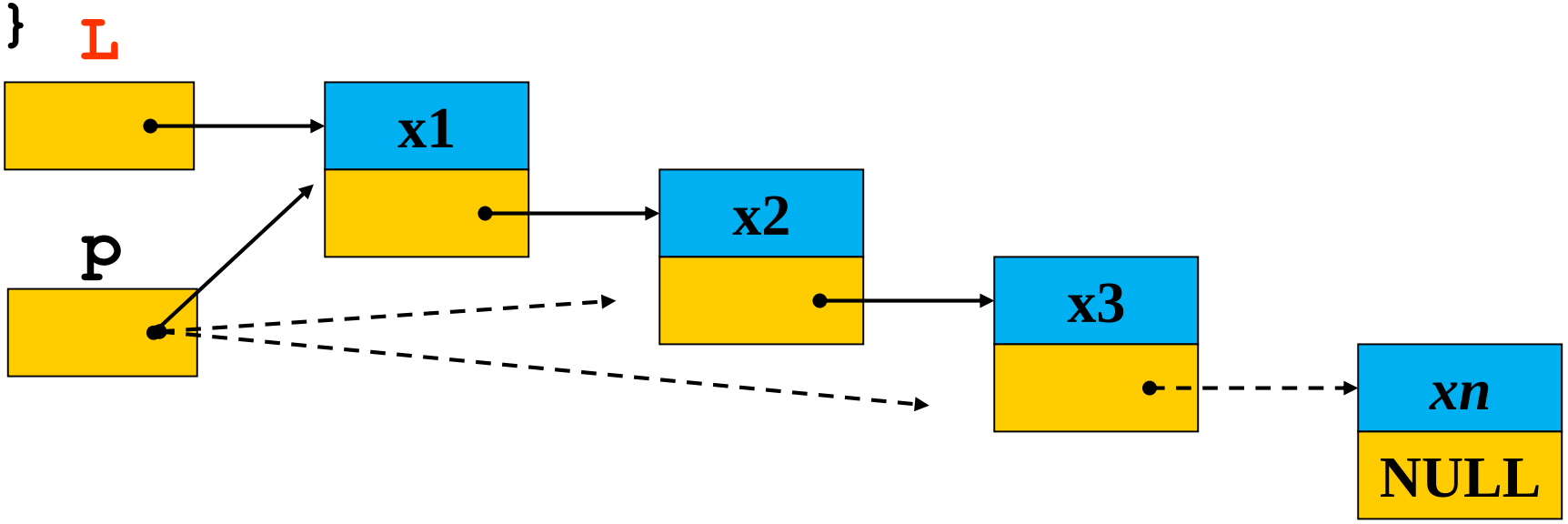
Impresión de una lista (cont.)

```
void ImpLista(Lista p) {
  while (p != NULL) {
    ImpDatos(p -> info);
    p = p -> sig;
  }
}
```

Al ser **p** un parámetro por valor, el puntero al comienzo de la lista se ve afectado sólo localmente!

Impresión de una lista (cont.)

```
void impLista(Lista p){
  while (p != NULL) {
    impDatos(p -> info);
    p = p -> sig;
  }
}
```



Impresión de una lista (cont.)

Una forma natural de describir este procedimiento es en forma ***recursiva***:

```
void ImpLista(Lista p){
if (p != NULL){
    ImpDatos(p -> info);
     ImpLista(p -> sig);
   }
}
```

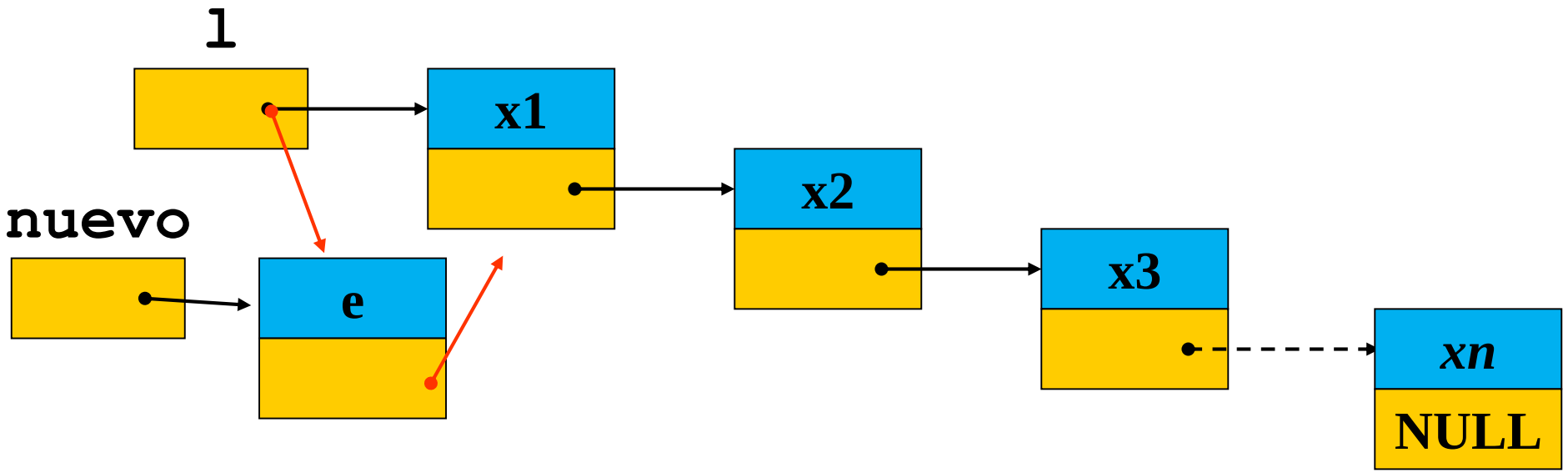
Impresión de una lista (cont.)

Podemos incluso imprimir la lista en orden inverso simplemente permutando las llamadas a los procedimientos:

```
void ImpLista(Lista p){
if (p != NULL){
    ImpLista(p -> sig);
    ImpDatos(p -> info);
  }
}
```

Inserción al comienzo de una lista

```
void insComienzo(A e, Lista & l){
   Lista nuevo = new nodoLista;
   nuevo -> info = e;
   nuevo -> sig = l;
   l = nuevo;
}
```



Concatenar dos listas

Otro procedimiento clásico es el de concatenar dos listas, el cual consiste en juntar el final de la primera lista con el comienzo de la segunda.

El resultado de la concatenación será devuelto en la primer lista.

Usando la notación de tipos inductivos escribir en pseudocódigo la función *concat*.

Concatenar dos listas

Usando la notación de tipos inductivos escribir en pseudocódigo la función *concat*.

[] **x.S**

concat: ALista x ALista → ALista

concat ([], l2) = …..

concat (x.S, l2) = ….. concat (S, l2)

Concatenar dos listas (cont.)

concat: ALista x ALista → Alista

concat ([], l2) = l2

concat (x.S, l2) = x.concat (S, l2)

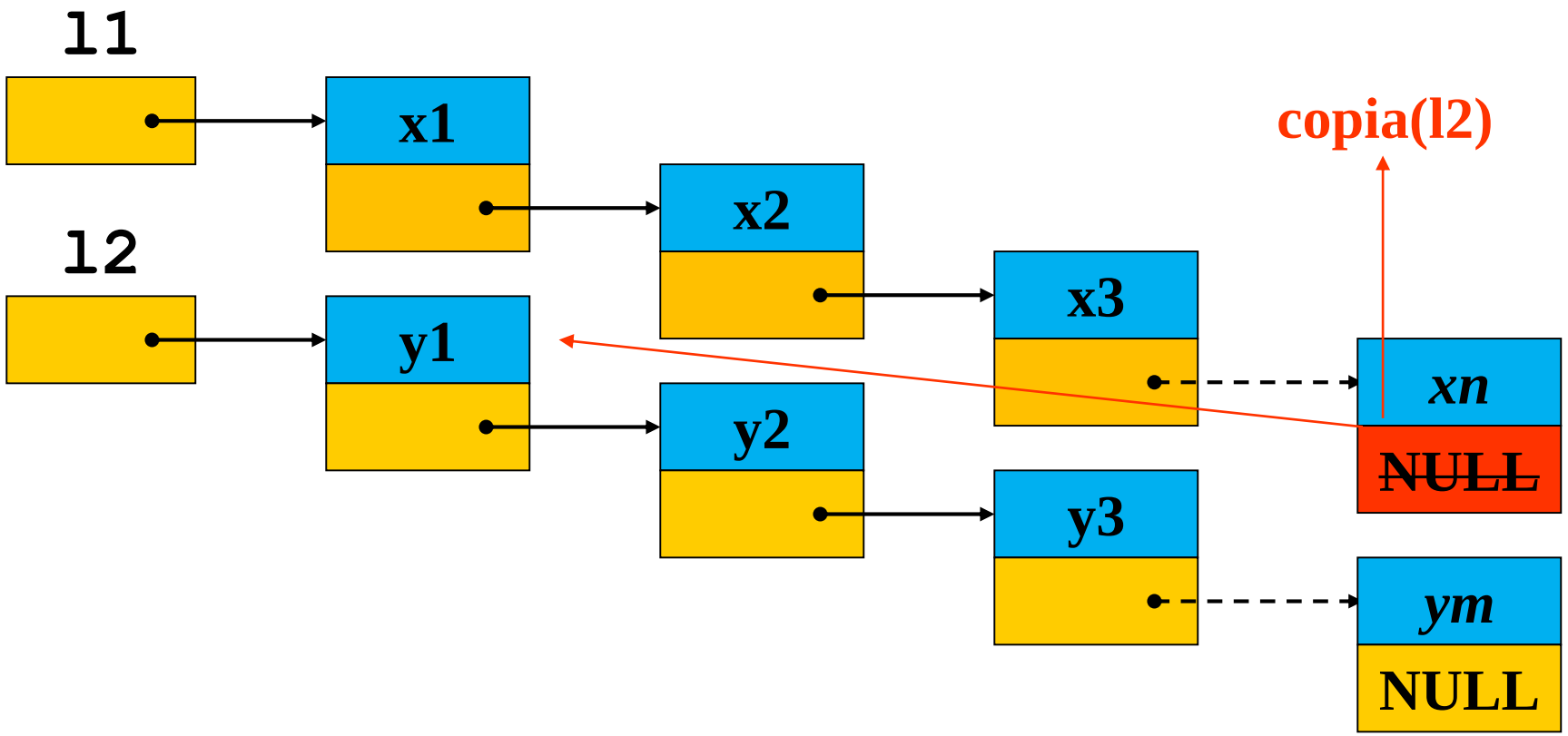
```
void concat(Lista & l1, Lista l2){
   if (l1 == NULL) l1 = l2; // l1 = copia(l2)
   else concat(l1 -> sig, l2);
}
```

Ojo con el caso base! (memoria compartida!): `copia(l) retorna una lista igual a l pero sin compartir memoria con l.`

```
\\ Al estilo de C
void concat(Lista * l1, Lista l2){
  if (*l1 == NULL) *l1 = copia(l2);
  else concat(&((*l1) -> sig), l2);
}
```

Concatenar dos listas (cont.)

```
void concat(Lista & l1, Lista l2){
  if (l1 == NULL) l1 = l2; //l1 = copia(l2)?
  else concat(l1 -> sig, l2);
}
```



Concatenar dos listas (cont.)

Implementar:

- **La función copia.**

- **void concat(Lista & l1, Lista l2)**

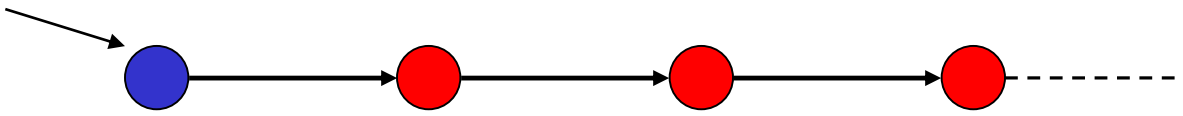
```
//iterativa
```

- **Lista concat(Lista l1, Lista l2)**

```
/* la lista resultado es independiente
de las listas parámetro */
```

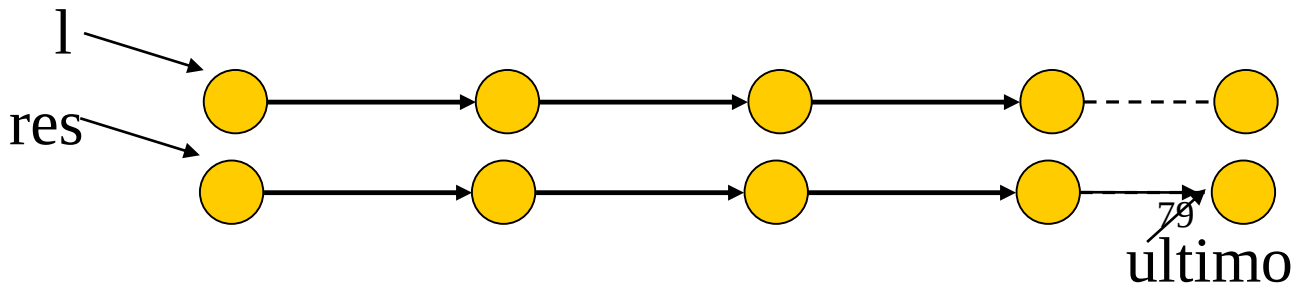
Copia

```
Lista copia (Lista l){
  if (l == NULL) return NULL;
  else{
     Lista res = new nodoLista;
     // no comparte memoria
     res->info = l->info;
     res->sig = copia(l->sig);
     return res;
  }
}
```



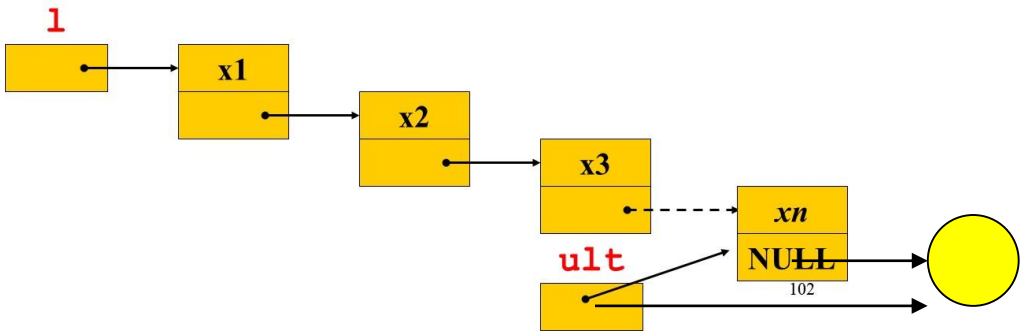
Copia (iterativa)

```
Lista copia (Lista l){
   Lista res, nuevo;
   if (l==NULL) res = NULL;
   else{
        Lista ultimo = NULL; // inserciones al final de res
        while (l!=NULL){
               nuevo = new nodoLista;
               nuevo->info = l->info;
               nuevo->sig = NULL;
               if (ultimo==NULL){ res = ultimo = nuevo; }
               else{  ultimo->sig = nuevo;
                      ultimo = nuevo;
               }
               l = l->sig;
        }
   }
   return res;
}
```



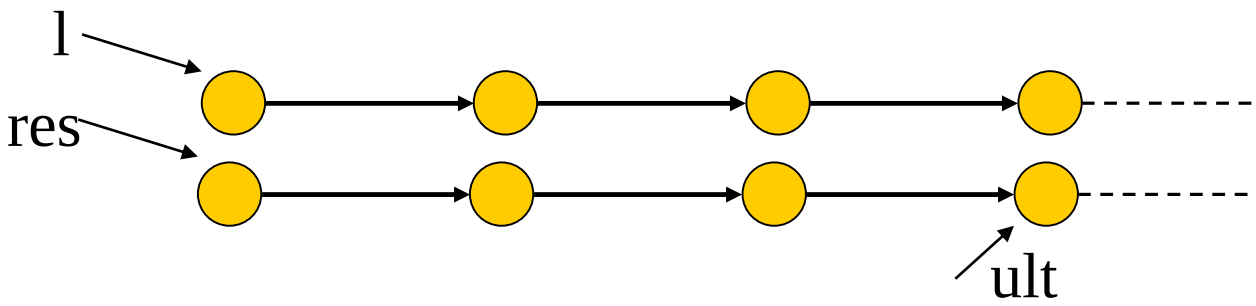
Inserción al final de una lista

```
void insFinal(T e, Lista & l, Lista & ult){
   Lista nuevo = new nodoLista;
   nuevo->info = e;
   nuevo->sig = NULL;
   if (l==NULL){ // ult==NULL
        l = ult = nuevo;
   }
   else{
        ult->sig = nuevo;
        ult = ult->sig;
   }
}
```



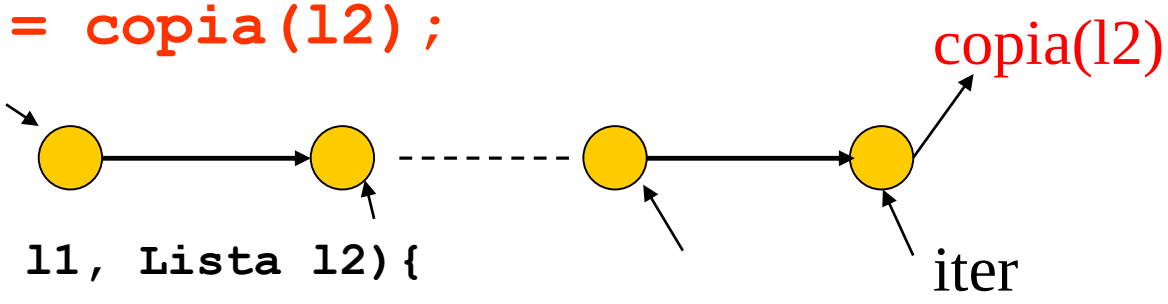
Copia (iterativa, versión 2)

```
Lista copia (Lista l){
   Lista res = NULL;
   Lista ult = NULL; // inserciones al final de res
   while (l!=NULL){
        insFinal(l->info, res, ult);
        l = l->sig;
   }
   return res;
}
```



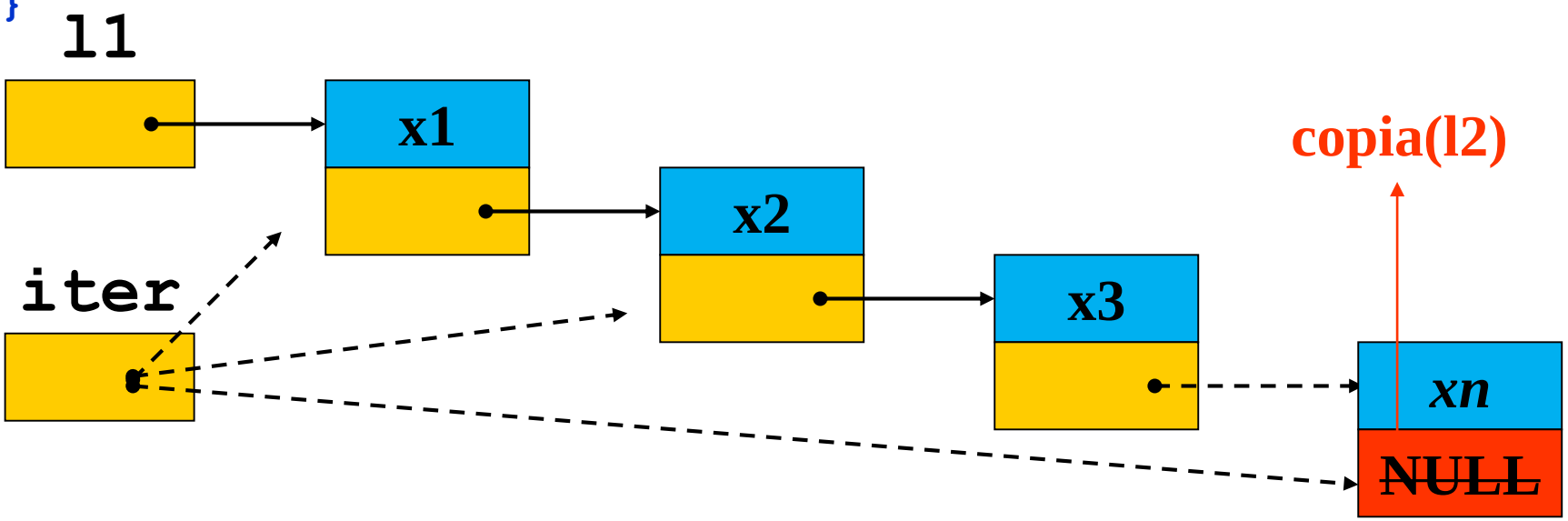
Concatenar dos listas (cont.)

```
//iterativa
void concat(Lista & l1, Lista l2){
  if (l1 == NULL) l1 = copia(l2);
  else{
      Lista iter = l1;
      while (iter->sig!=NULL){
            iter = iter->sig;
      }
      iter->sig = copia(l2);
}
//recursiva
void concat(Lista & l1, Lista l2){
  if (l1 == NULL) l1 = l2; // l1 = copia(l2)
  else concat(l1 -> sig, l2);
}
```



Concatenar dos listas (cont.)

```
//iterativa
void concat(Lista & l1, Lista l2){
  if (l1 == NULL) l1 = copia(l2);
  else{
       Lista iter = l1;
       while (iter->sig != NULL){
              iter = iter->sig;
       }
       iter->sig = copia(l2);
}
```



Concatenar dos listas (cont.)

```
/* la lista resultado es independiente
de las listas parámetro */

Lista concat(Lista l1, Lista l2){
  if (l1 == NULL) return copia(l2);
  else{
     Lista res = new nodoLista;
     res->info = l1->info;
     res->sig = concat(l1->sig, l2);
     return res;
}
```

Inserción ordenada

Insertar de manera ordenada un elemento en una lista ordenada (vista en el teórico de recursión).

insOrd: A x ALista → ALista

insOrd (e, []) = e.[]

insOrd (e, x.S) = e.x.S, Si e<=x

insOrd (e, x.S) = x.insOrd(e, S), Sino

```
void insOrd(A e, Lista & l){
  if (l == NULL) insComienzo(e, l);
  else{
     if (e <= l->info) insComienzo(e, l);
     else insOrd(e, l->sig);
  }
}
```

Insert Sort

Ordenar una lista usando la función previa insOrd.

Ord: ALista → ALista

Ord ([]) = []

Ord (x.S) = insOrd(x, Ord(S))

```
// Insert Sort iterativa pero usando insOrd
Lista Ord(Lista l){
  Lista lres = NULL;
  while (l != NULL){
      insOrd(l->info, lres);
      l = l->sig;
  }
  return lres;
}
```

Invertir una lista

Otro procedimiento clásico es el de invertir el orden de los elementos de una lista. Este procedimiento es muy similar al de recorrida.

En principio no vamos a reutilizar los registros de la lista original, sino que vamos a crear otra lista con nuevos registros.

Invertir una lista (cont.)

```
Lista invLista (Lista l) {
  Lista reg; /* var auxiliar */
  Lista l_inv = NULL;
  while (l != NULL){
    reg = new nodoLista;
    reg->info = l->info;
    reg->sig = l_inv;
    l_inv = reg;
    l = l->sig;
  }
  return l_inv;
}
```

insComienzo(l->info, l_inv);

Invertir una lista (cont.)

Pero la inversión puede ser realizada reutilizando los mismos registros de la lista original. En esta versión la lista resultado se pasa por referencia al procedimiento:

```
void invLista (Lista l, Lista & l_inv){
      Lista reg;
      l_inv = NULL;
      while (l != NULL){
            reg = l->sig;
            l->sig = l_inv;
            l_inv = l;
            l = reg;
      }
}
```

Merge

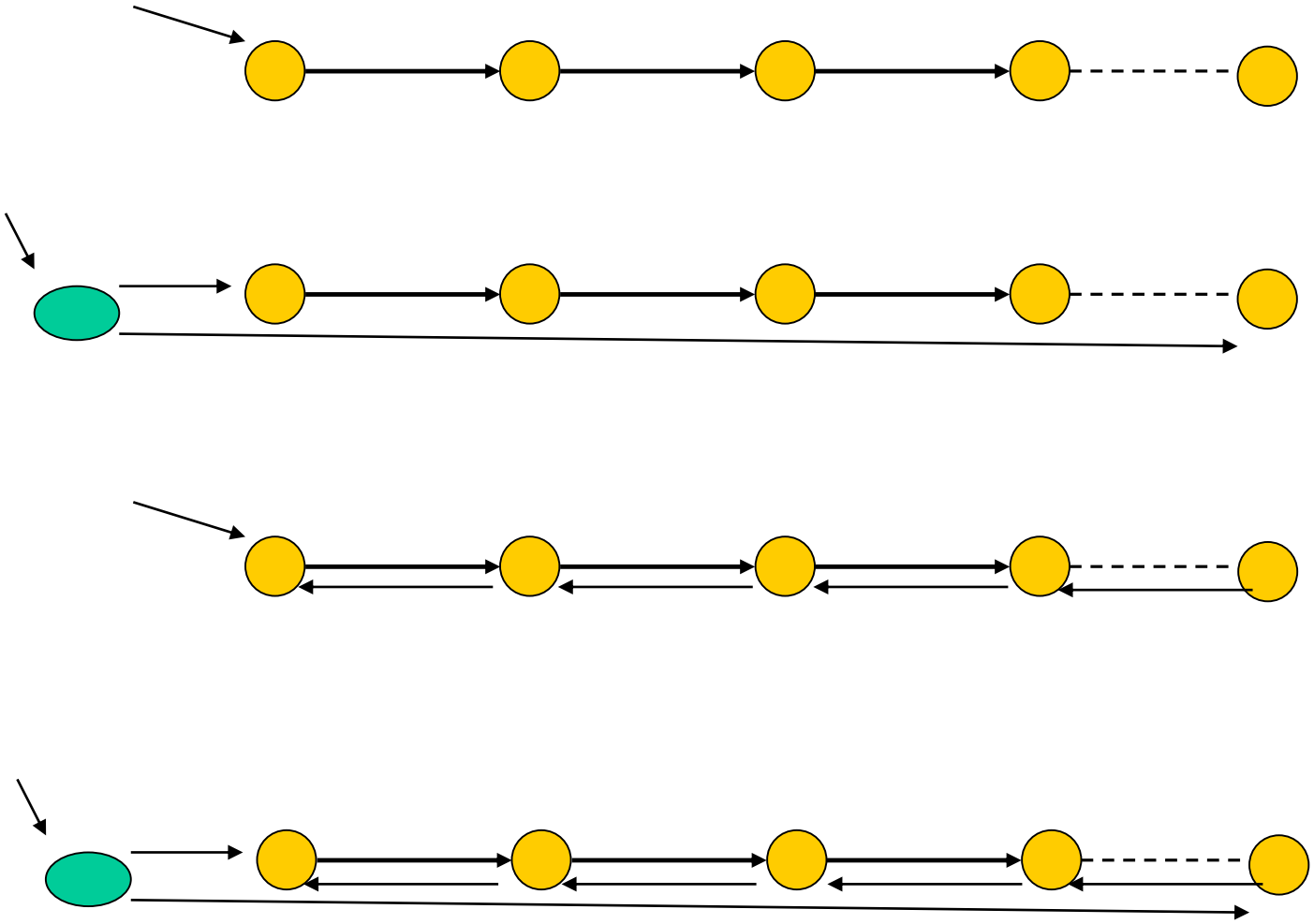
Defina una función *merge* que dadas dos listas de enteros ordenadas de menor a mayor, construya y retorne una nueva lista ordenada de menor a mayor que contenga todos los elementos de ambas listas.

Por ejemplo, si las listas son [1,3,3,9] y [1,2,5], el resultado debería ser: [1,1,2,3,3,5,9].

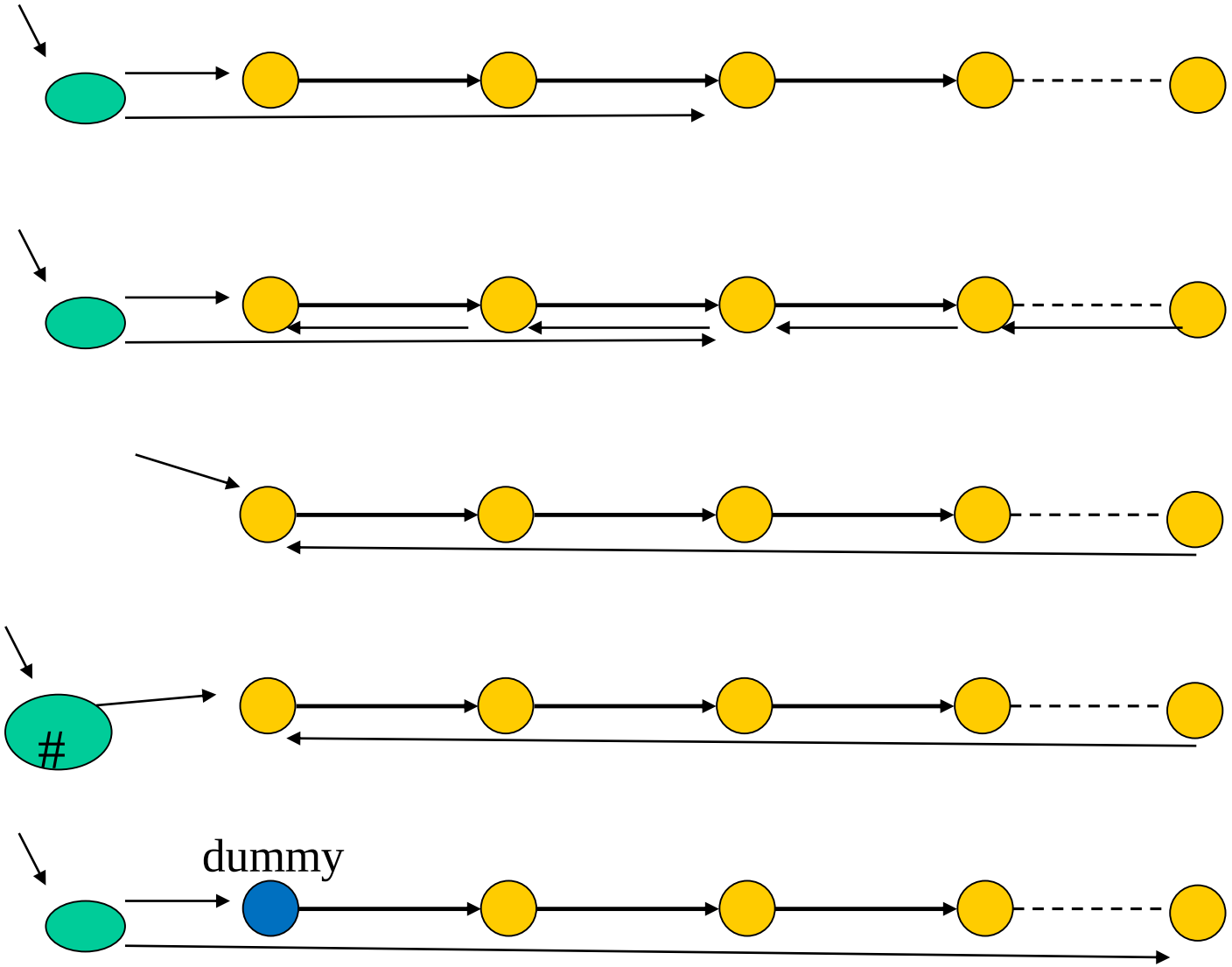
```
Lista merge (Lista l1, Lista l2) {

…
```

```
Lista merge (Lista l1, Lista l2) {
    Lista lres = NULL; Lista ult = NULL;
    while (l1!=NULL && l2!=NULL){
        if (l1->info < l2->info){
                insFinal(l1->info,lres,ult);
                l1 = l1->sig;
        } else {
                insFinal (l2->info,lres,ult);
                l2 = l2->sig;
        }
    } // O(n+m) siendo n y m los largos de las listas.
    while (l1!=NULL) { insFinal(l1->info,lres,ult); l1 = l1->sig; }
    while (l2!=NULL) { insFinall(l2->info,lres,ult); l2 = l2->sig; }
    return lres;
} // O(n+m) siendo n y m los largos de las listas.
```

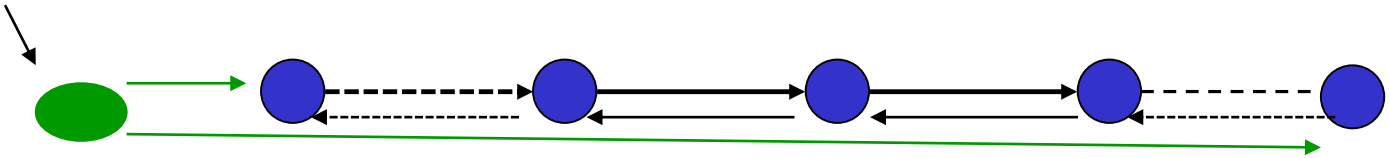
Algunas variantes de Listas



Algunas variantes de Listas



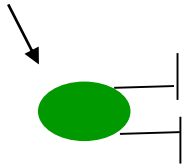
Lista doblemente encadenada con punteros al inicio y al final



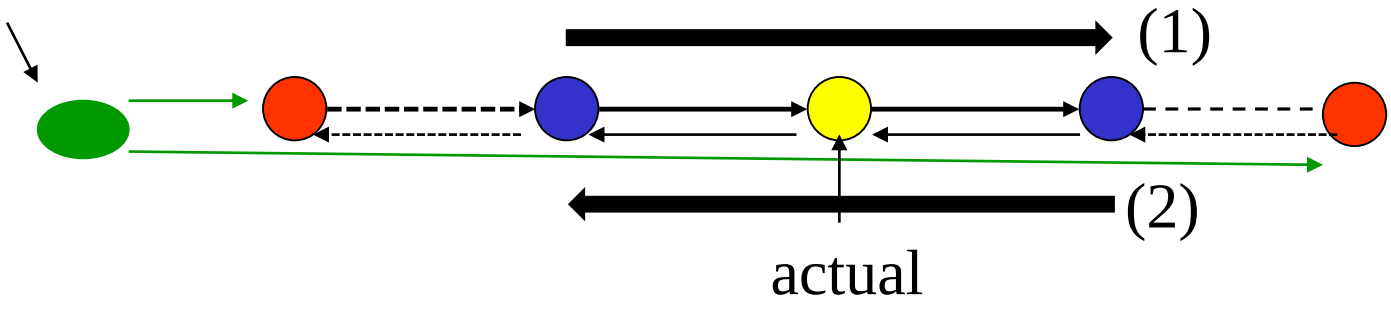
```
struct nodoDoble {
        T dato;
        nodoDoble* sig;
        nodoDoble* ant;
}
struct cabezal {
        nodoDoble* inicio;
        nodoDoble* final;
}
```

Lista doblemente encadenada con punteros al inicio y al final

```
struct nodoDoble {
        T dato;
        nodoDoble* sig;
        nodoDoble* ant;
}
struct cabezal {
        nodoDoble* inicio;
        nodoDoble* final;
}
```



Lista vacía: inicio y final en NULL



actual->ant->sig = actual->sig; // (1)

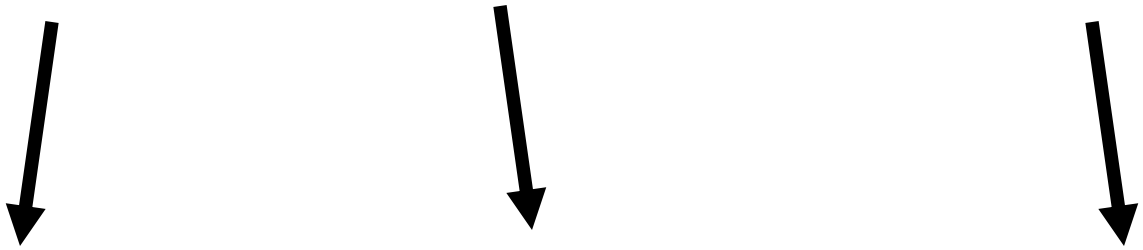
actual->sig->ant = actual->ant; // (2)

Bibliografía recomendada

– Como Programar en C/C++

H.M. Deitel & P.J. Deitel

(Cap. 7, algo del 12 y algo del 15)



Punteros en C

Algunas Estruc. dinámicas en C

Punteros en C++

-- El libro de Weiss.