4 Estructuras Dinámicas

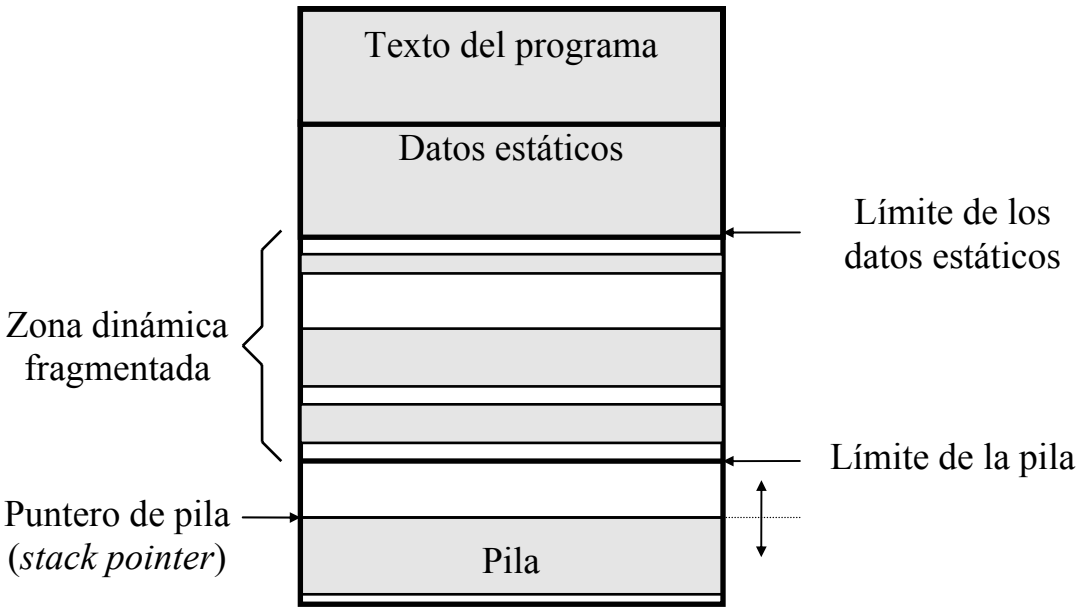
Contenido

1. **Gestión dinámica de memoria.**
2. **El tipo puntero.**
3. **Operaciones con punteros.**
4. **Operaciones básicas sobre listas enlazadas.**

1. Gestión dinámica de memoria.

Datos estáticos y dinámicos

- Datos *estáticos*: su tamaño y forma es constante durante la ejecución de un programa y por tanto se determinan en tiempo de compilación. El ejemplo típico son los arrays. Tienen el problema de que hay que dimensionar la estructura de antemano, lo que puede conllevar desperdicio o falta de memoria.
- Datos *dinámicos*: su tamaño y forma es variable (o puede serlo) a lo largo de un programa, por lo que se crean y destruyen en tiempo de ejecución. Esto permite dimensionar la estructura de datos de una forma precisa: se va *asignando* memoria en tiempo de ejecución según se va necesitando.
- Cuando el sistema operativo carga un programa para ejecutarlo y lo convierte en proceso, le asigna cuatro partes lógicas en memoria principal: texto, datos (estáticos), pila y una zona libre. Esta zona libre (o *heap*) es la que va a contener los datos dinámicos, la cual, a su vez, en cada instante de la ejecución tendrá partes asignadas a los mismos y partes libres que *fragmentarán* esta zona, siendo posible que se agote si no se liberan las partes utilizadas ya inservibles. (La pila también varía su tamaño dinámicamente, pero la gestiona el sistema operativo, no el programador):



- Para trabajar con datos dinámicos necesitamos dos cosas:

a) Subprogramas *predefinidos* en el lenguaje que nos permitan gestionar la memoria de forma dinámica (asignación y liberación).

b) Algún tipo de dato con el que podamos acceder a esos datos dinámicos (ya que con los tipos vistos hasta ahora sólo podemos acceder a datos con un tamaño y forma ya determinados).

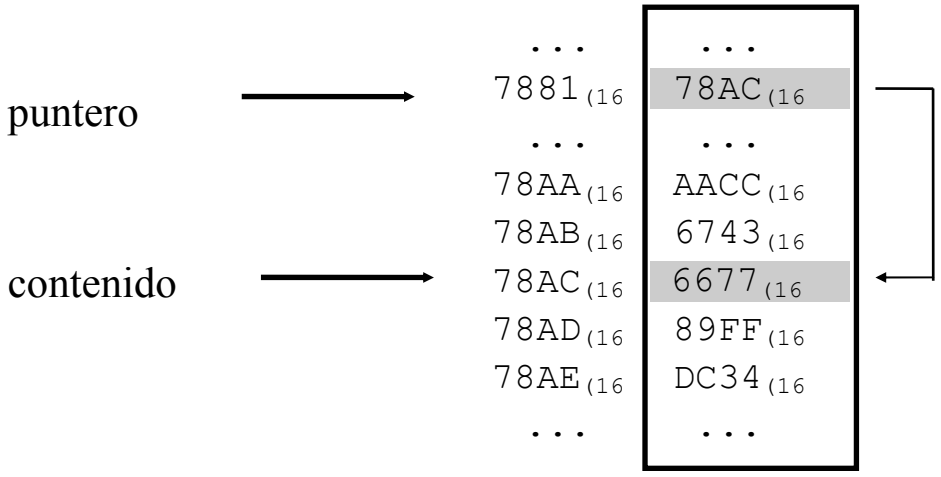
2. Tipo puntero

- Las variables de *tipo puntero* son las que nos permiten referenciar datos dinámicos. Tenemos que diferenciar claramente entre:

  a) la variable referencia o apuntadora, de *tipo puntero*;

  b) la variable *anónima* referenciada o apuntada, de cualquier tipo, tipo que estará asociado siempre al puntero.

- Físicamente, un puntero no es más que una dirección de memoria. En el siguiente ejemplo se muestra el contenido de la memoria con un puntero que apunta a la dirección $78AC_{(16}$, la cual contiene $6677_{(16}$:



- Antes de definir los punteros, vamos a explicar como puede darse nombre a tipos de datos propios utilizando la palabra reservada **typedef**.

  El uso de *typedef* (definición de tipo) en C permite definir un nombre par un tipo de datos en C. La declaración de *typedef* es similar a la declaración de una variable. La forma es:

  typedef tipo nuevo-tipo;

  Ejemplos:

  typedef char LETRA;

  LETRA carácter;

  typedef enum estado_luces (Rojo, Amarillo, Verde);

estado_luces semáforo;
typedef vector_de_20 nombre[20];
vector_de_20 mivector;

Introduciremos las declaraciones de tipo antes de las declaraciones de variables en el código.

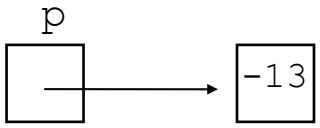
- Definiremos un tipo puntero con el carácter asterisco (*) y especificando siempre el tipo de la variable referenciada: Ejemplo:

```
typedef int *PtrInt; /* puntero a enteros */
PtrInt p;            /* puntero a enteros */
```

O bien directamente:

```
int *p;  /* puntero a enteros */
```

Cuando `p` esté apuntando a un entero de valor `-13`, gráficamente lo representaremos así:



- Para acceder a la variable apuntada hay que hacerlo a través de la variable puntero, ya que aquélla no tiene nombre (por eso es *anónima*). La forma de denotarla es *`p`. En el ejemplo *`p` = `-13` (y `p` = dirección de memoria de la celda con el valor `-13`, dirección que no necesitamos tratar directamente).
- El tipo registro o **estructura** en C: *struct*.

Una declaración de estructura define una variable estructura e indica una secuencia de nombres de variables –denominadas **miembros de la estructura** o **campos del registro**– que pueden tener tipos diferentes. La forma básica es:

struct
{
tipo identificador_1;
tipo identificador_2;
.
.
.

```
   tipo identificador_3;
}identificador_de_la_estructura;
```

Ejemplo:

```
struct
{
   char fabricante[20];   /*Fabricante de la resistencia */
   int cantidad;          /* Número de resistencias */
   float precio_unitario; /* Precio de cada resistencia */
}resistencias;
```

El acceso a los miembros de la estructura se realiza mediante el operador punto (.). Ejemplo: *resistencias.cantidad=20; gets(resistencias.fabricante); if (resistencias.precio_unitario==50.0)*

Podemos crear una plantilla de estructura para más tarde declarar variables de ese tipo. Ejemplo:

```
struct registro_resistencias
{
   char fabricante[20];   /*Fabricante de la resistencia */
   int cantidad;          /* Número de resistencias */
   float precio_unitario; /* Precio de cada resistencia */
};
struct registro_resistencias resistencias;
```

Por último, podemos usar *typedef*:

```
typedef struct
{
   char fabricante[20];   /*Fabricante de la resistencia */
   int cantidad;          /* Número de resistencias */
   float precio_unitario; /* Precio de cada resistencia */
}registro_resistencias;
registro_resistencias resistencias;
```

- Ejemplo de punteros a estructuras:

```
typedef struct
{
    int num;
    char car;
}TipoRegistro;
typedef TipoRegistro *TipoPuntero;
TipoPuntero p;
```

Así:

`p` es la dirección de un registro con dos campos (tipo puntero)

`*p` es un registro con dos campos (tipo registro)

`(*p).num` es una variable simple (tipo entero)

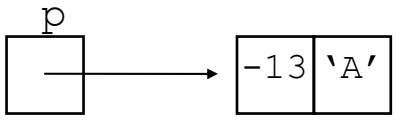
`p->num` es una variable simple (tipo entero)

`(*p).car` es una variable simple (tipo carácter)

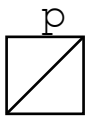
`p->car` es una variable simple (tipo carácter)

`&x` es la dirección de una variable x, siendo x, por ejemplo `int x;`

Gráficamente:



- Si deseamos que una variable de tipo puntero (a algo), en un momento determinado de la ejecución del programa no apunte a nada, le asignaremos la palabra reservada `NULL` (`p = NULL`) y gráficamente lo representaremos:

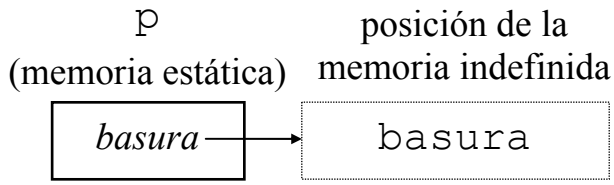


Gestión de la memoria dinámica

- Cuando declaramos una variable de tipo puntero, por ejemplo

```
int *p;
```

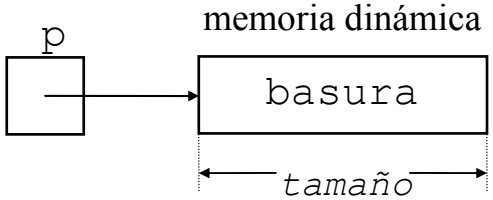
estamos creando la variable `p`, y se le reservará memoria –estática– en tiempo de compilación; pero la variable referenciada o anónima no se crea. En este momento tenemos:



- La variable anónima debemos crearla después mediante una llamada a un procedimiento de *asignación de memoria* –dinámica– predefinido. El archivo de cabecera <*stdlib.h*> y <*alloc.h*> contienen una función llamada ***malloc()***, que se utiliza para asignar memoria dinámica en tiempo de ejecución. *malloc()* recibe el número de bytes que necesitan ser asignados si se encuentra memoria disponible. Si no hay memoria disponible para satisfacer la operación, *malloc()* devuelve un puntero a *NULL*.
- Así:

```
p=malloc(sizeof(int));
```

donde `p` es una variable de tipo puntero y *`sizeof(int)`* es el tamaño en bytes del tipo de la variable anónima. En tiempo de ejecución, después de la llamada a este procedimiento, tendremos ya la memoria (dinámica) reservada pero sin inicializar:



- Para saber el tamaño necesario en bytes que ocupa una variable de un determinado tipo, dispondremos también de una función predefinida:

```
sizeof(Tipo)
```

que nos devuelve en nº de bytes que necesita una variable del tipo de datos *`Tipo`*. Ejemplo: `p=malloc(sizeof(int))`.

- Podemos usar este procedimiento hasta agotar la memoria dinámica. Debemos comprobar, en cada reserva, que sigue habiendo memoria para hacerla. Esto se hace de la siguiente manera:

```
typedef struct
{
  int num;
  char car;
}TipoRegistro;
typedef TipoRegistro *TipoPuntero;
TipoPuntero p;
if ((p=(TipoPuntero)malloc(sizeof(TipoRegistro)))==NULL){
  printf("No hay suficiente memoria.\n");
  exit(1);
}
```

- Tras usar la memoria asignada a un puntero, hay que liberarla para poder utilizarla con otros fines y no dejar datos inservibles en el mapa de la memoria dinámica. Se libera la memoria asignada a un puntero *p*, cuyo tipo ocupa *t* bytes, también dinámicamente, mediante el la función *free()* que se encuentra en el archivo de cabecera <*stdlib.h*>:

3. Operaciones con Punteros

- Al definir una variable de tipo puntero, implícitamente podemos realizar las siguientes operaciones con ella:
  1. *Acceso a la variable anónima*
  2. *Asignación*
  3. *Comparación*
  4. *Paso como parámetros*

  Utilizaremos el siguiente ejemplo para ver estas operaciones:

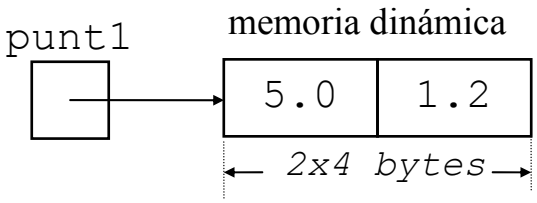
```
typedef struct
{
   float ParteReal, ParteCompleja;
}PartesComplejas;
typedef PartesComplejas *Complejo;
void main(){
   Complejo punt1, punt2;

   if ((punt1=(Complejo) malloc(size...
   ...
```

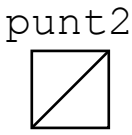
- El *acceso a la variable anónima* se consigue de la forma ya vista con el operador `*`, y la *asignación* se realiza de la misma forma que en los tipos simples. En el ejemplo, suponiendo que `punt1` ya tenga memoria reservada, podemos asignar valores a los campos de la variable anónima de la siguiente forma:

```
punt1->PartReal = 5.0;
punt1->PartImag = 1.2;
```

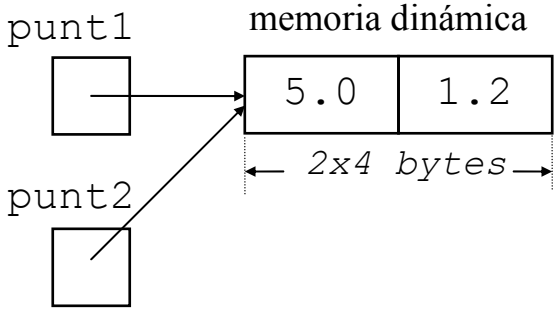
Tras las asignaciones tendremos (suponiendo que un real ocupa 4 bytes):



A una variable puntero también se le puede asignar la palabra reservada `NULL` (sea cual sea el tipo de la variable anónima): `punt2 = NULL;`, tras lo cual tendremos:



Se puede asignar el valor de una variable puntero a otra siempre que las variables a las que apuntan sean del mismo tipo: `punt2 = punt1;`



- La *comparación* de dos punteros, mediante los operadores <> y =, permite determinar si apuntan a la misma variable referenciada:

```
if (punt1 == punt2) /* expresión lógica */
```

o si no apuntan a nada:

```
  if (punt1 == NULL)    /* expresión lógica */
```

- El *paso de punteros como parámetros* permite el paso de variables por refencia a funciones. Ejemplo:

```
#include <stdio.h>
void llamame(int *p);
void main()
{
 int x;

 x=0;
 printf("El valor de x es %d\n", x);
 llamame(&x);
 printf("El nuevo valor de x es %d\n", x);
}
void llamame(int *p)
{
 *p=5;
}
```

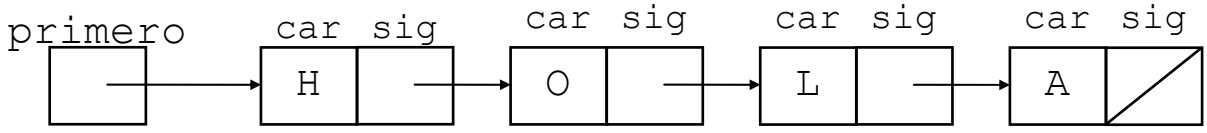
4. Operaciones básicas con Listas enlazadas

- Lo que realmente hace de los punteros una herramienta potente es la circunstancia de que pueden apuntar a variables que a su vez contienen punteros. Los punteros así creados sí son variables dinámicas, a diferencia de los declarados explícitamente, los cuales son estáticos porque pueden crearse en tiempo de compilación.
- Cuando en cada variable anónima o *nodo* tenemos un solo puntero que apunta al siguiente nodo tenemos una *lista enlazada.*
- Ejemplo: *cadenas de caracteres de longitud variable.*

```
typedef struct nodo{
   char car;          /* Carácter */
   struct nodo *sig; /* Al siguiente nodo */
}TipoRegistro;

TipoRegistro *primero;
```

Con estos tipos de datos podemos crear estructuras que no malgastan la memoria, y que sí malgastaría un array de longitud fija:



- Un algoritmo que crea una lista enlazada similar sería:

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

typedef struct nodo
{
    int dato;
    struct nodo *enlace;
} LISTA;

void mostrar_lista(LISTA *ptr);
void insertar(LISTA **ptr, int elemento);

void main()
{
    LISTA *n1 = NULL;
    int elemento;

    do
    {
        printf("\nIntroduzca elemento: ");
        scanf("%d", &elemento);
        if(elemento != 0)
            insertar(&n1, elemento);
    } while(elemento != 0);
    printf("\nLa nueva lista enlazada es: ");
    mostrar_lista(n1);
}

void mostrar_lista(LISTA *ptr)
{
    while(ptr != NULL)
    {
        printf("%c",ptr->dato);
        ptr = ptr->enlace;
    }
    printf("\n");
}

void insertar(LISTA **ptr, int elemento)
{
    LISTA *p1, *p2;

    p1 = *ptr;
    if(p1 == NULL)
```

```
    {
        p1 = malloc(sizeof(LISTA));
        if (p1 != NULL)
        {
            p1->dato = elemento;
            p1->enlace = NULL;
            *ptr = p1;
        }
    }
    else
    {
        while(p1->enlace != NULL)
            p1 = p1->enlace;
        p2 = malloc(sizeof(LISTA));
        if(p2 != NULL)
        {
            p2->dato = elemento;
            p2->enlace = NULL;
            p1->enlace = p2;
        }
    }
}
```

Operaciones básicas sobre listas enlazadas

- Son las siguientes:
  a) *Inserción de un nodo al principio*
  b) *Insertar un nodo en una lista enlazada ordenada*
  c) *Eliminar el primer nodo*
  d) *Eliminar un nodo determinado*

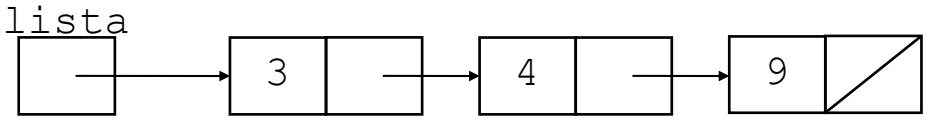
Vamos a basarnos en las declaraciones de una lista enlazada de enteros:

```
typedef struct nodo{
   int dato;          /* Elemento entero */
   struct nodo *sig; /* Al siguiente nodo */
}TipoNodo;

typedef TipoNodo *TipoLista;

TipoLista lista; /* Cabeza de la lista */
TipoLista nodo;  /* Nuevo nodo a insertar */
TipoLista ptr;   /* Puntero auxiliar */
```

Y partiremos del estado inicial de la lista:



- ***Inserción de un nodo al principio***

Los pasos a dar son los siguientes:

1) Reservar memoria para el nuevo nodo:

```
if ((nodo=(TipoLista) malloc(sizeof(TipoNodo)))==NULL)...
```

2) Asignar valor nuevo al nuevo nodo:

```
nodo->dato = 5 /* por ejemplo */
```

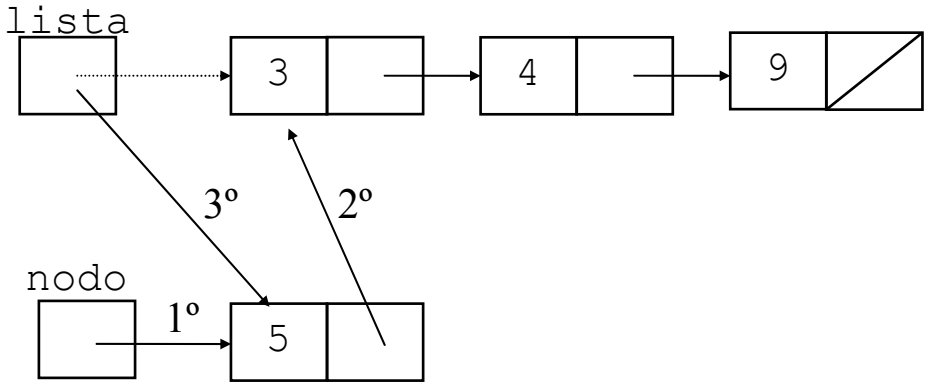
3) Enlazar la lista al siguiente del nuevo nodo:

```
nodo->sig = lista
```

4) Actualizar la cabeza de la lista:

```
lista = nodo
```

Gráficamente (la línea discontinua refleja el enlace anterior a la operación, y también se especifica el orden de asignación de valores a los punteros):



- ***Inserción de un nodo en una lista ordenada (ascendentemente)***

  Los pasos a seguir son los siguientes:

  1) Reservar memoria para el nuevo nodo:

```
if ((nodo=(TipoLista) malloc(sizeof(TipoNodo)))==NULL)...
```

  2) Asignar valor nuevo al nuevo nodo:

```
nodo->dato = 5 /* por ejemplo */
```

  3.a) Si la lista está vacía o el dato nuevo es menor que el de la cabeza, el nodo se inserta al principio y se actualiza la cabeza `lista` para que apunte al nuevo nodo.

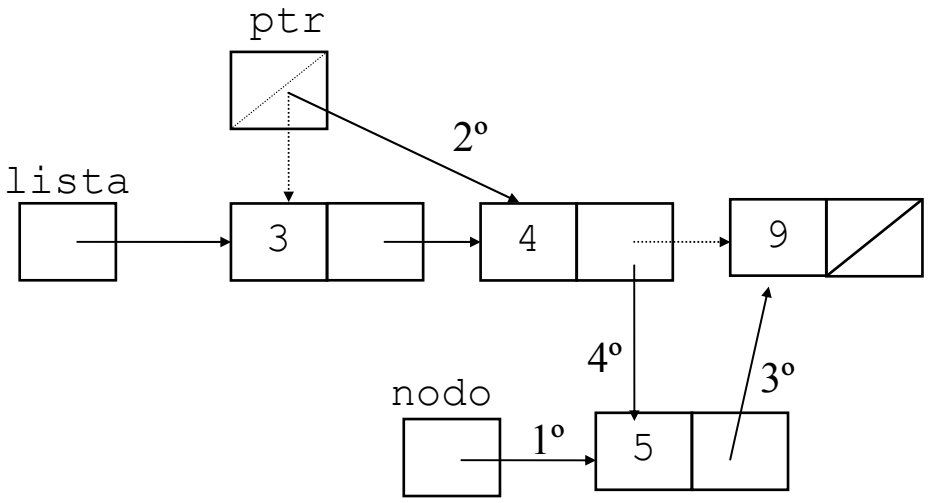
  3.b) Si la lista no está vacía y el dato nuevo es mayor que el que hay en la cabeza, buscamos la posición de inserción usando un bucle del tipo:

```
while ((ptr->sig != NULL) &&
          (nodo->dato > (ptr->sig)->dato))
   ptr = ptr->sig;
/* ptr queda apuntando al predecesor */
```

  y enlazamos el nodo:

```
nodo->sig = ptr->sig /* con el sucesor */
ptr->sig = nodo      /* con el predecesor */
```

  Gráficamente (observar el orden de asignación de punteros –el segundo paso son las sucesivas asignaciones al puntero `ptr`):



- ***Eliminar el primer nodo***

Los pasos son (suponemos lista no vacía):

1) Guardar el puntero a la cabeza actual:

```
ptr = lista
```

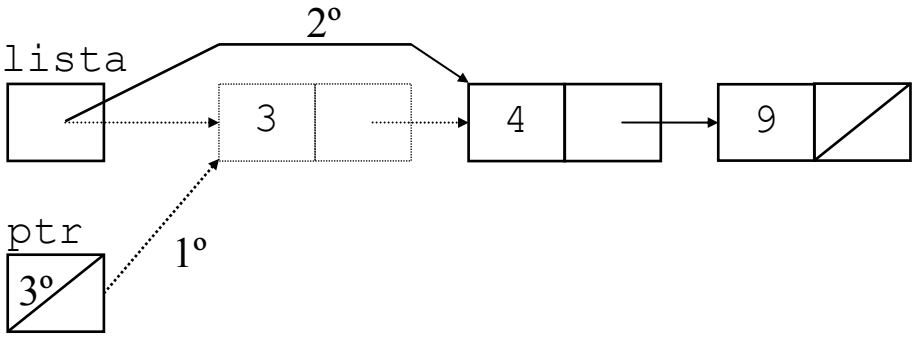
2) Avanzar una posición la cabeza actual:

```
lista = lista->sig
```

3) Liberar la memoria del primer nodo:

```
free(ptr)
```

Gráficamente (el tercer paso pone `ptr` a `NULL`):



- ***Eliminar un nodo determinado***

Vamos a eliminar el nodo que contenga un valor determinado. Los pasos son los siguientes (suponemos lista no vacía):

Caso a) El dato coincide con el de la cabeza:

Se elimina como en la operación anterior.

Caso b) El dato no coincide con el de la cabeza:

1) Se busca el predecesor y se almacena en una variable de tipo puntero, por ejemplo, en `ant`. En otra variable, `ptr`, se almacena el nodo actual:

```
ant = lista;
ptr = lista->sig;
while ((ptr ¡= NULL) &&
          (ptr->dato ¡= valor)){
   ant = ptr;
   ptr = ptr->sig;
}
```

2) Se comprueba si se ha encontrado, en cuyo caso se enlaza el predecesor con el sucesor del actual:

```
if (ptr ¡= NULL){ /* encontrado */
   ant->sig = ptr->sig;
   free(ptr);
}
```

Gráficamente quedaría (eliminando el nodo de valor `4`):