

Estructura de Datos

Sesión 2



Facultad de Ingeniería de Sistemas e
Informática

Gustavo Arredondo C.
garredondoc@unmsm.edu.pe

Contenido

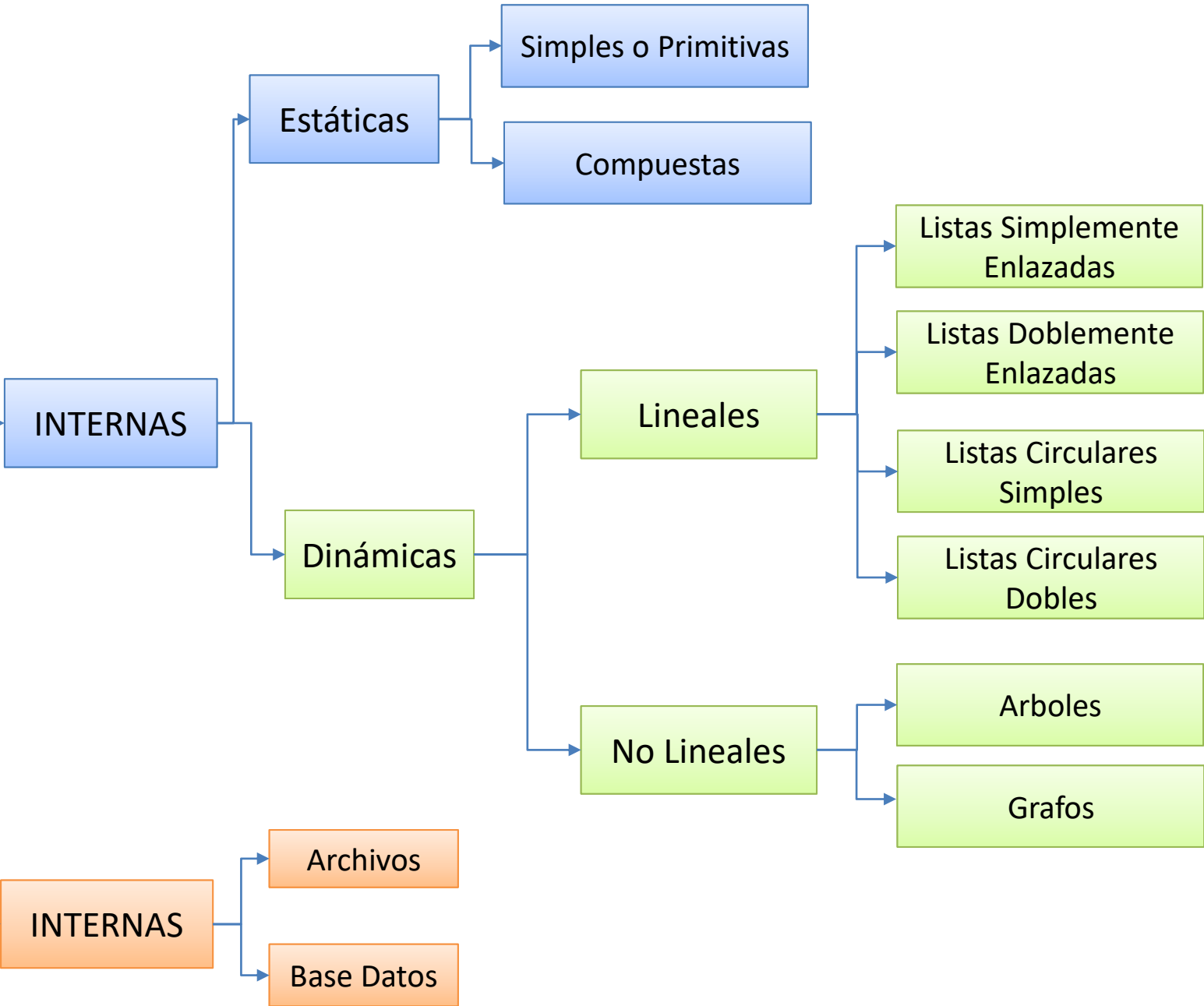
Temas de Importancia

- Punteros
- Estructura de Registros

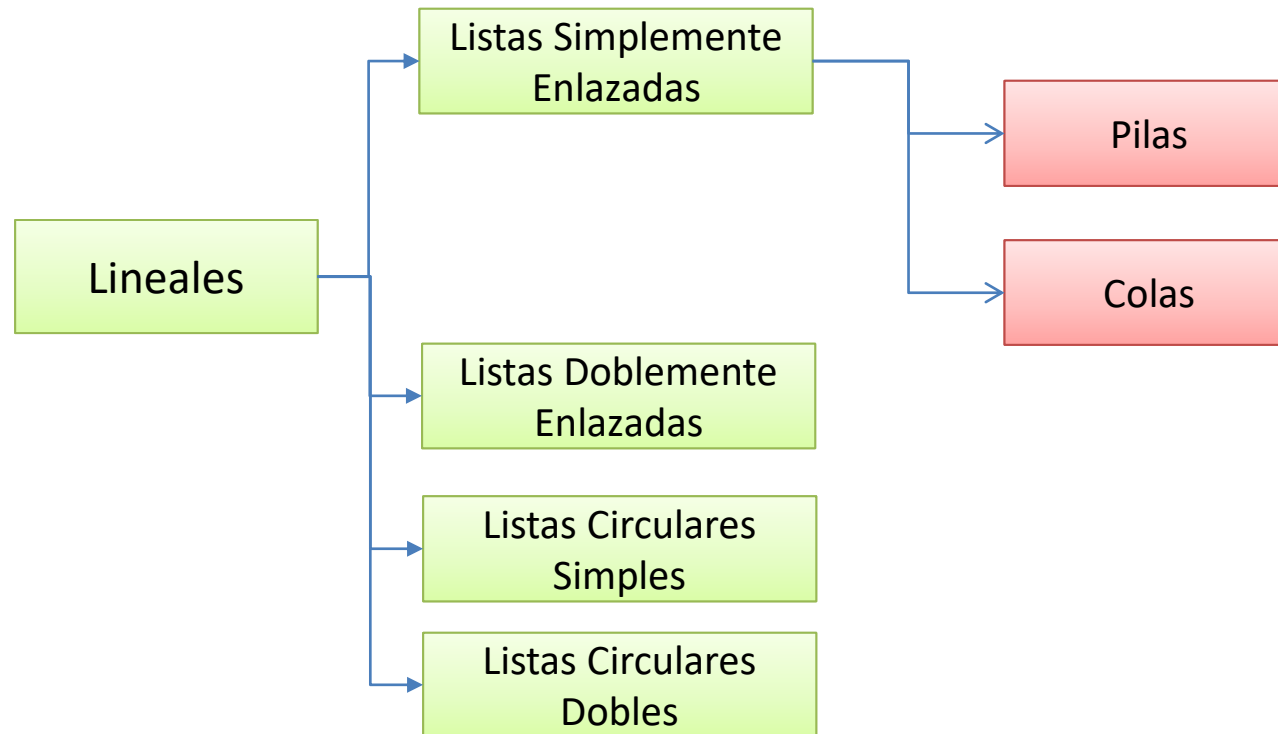
Estructuras Dinámicas

- Listas Enlazadas Simples
- Insertar Elementos en una LES
- Recorrer la lista de elementos en una LES

Estructura de Datos



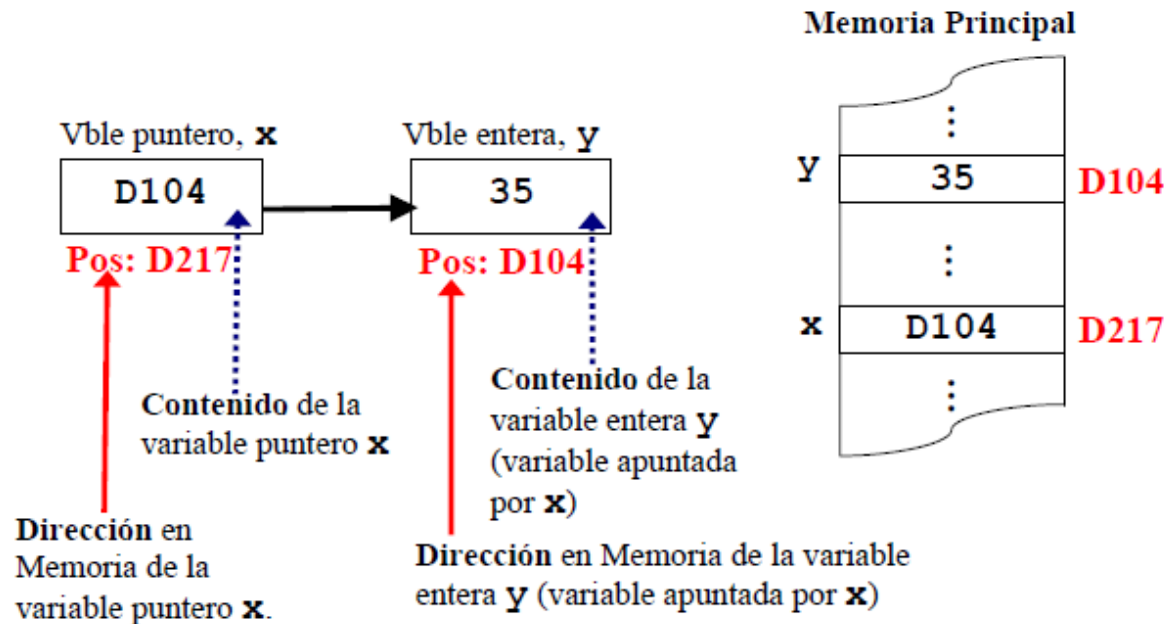
Estructura de Datos Lineales



Punteros

Punteros

Es una variable estática que, en vez de contener valores de datos, contiene valores que representan direcciones de memoria de variables.



PUNTERO → Variable que contiene la DIRECCIÓN de memoria en la que se encuentra almacenada otra variable.

La sintaxis general para declarar una variable tipo puntero es:

```
tipo *nombre_puntero;
```

donde tipo es el tipo de variables a la que apuntará el puntero.

```
int *p;
```

Los operadores punteros

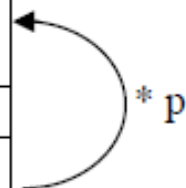
Existen dos operadores especiales de punteros: * y &.

- El operador & (operador dirección), aplicado sobre el nombre de una variable, devuelve su dirección de memoria.
- El operador * (operador indirección) aplicado sobre una variable de tipo puntero permite acceder al dato al que apunta, es decir, al valor de la variable situada en esa dirección de memoria.

Ejemplo1:

```
int *p, b, x = 1250;  
p = &x;  
b = *p; /* b = x; */
```

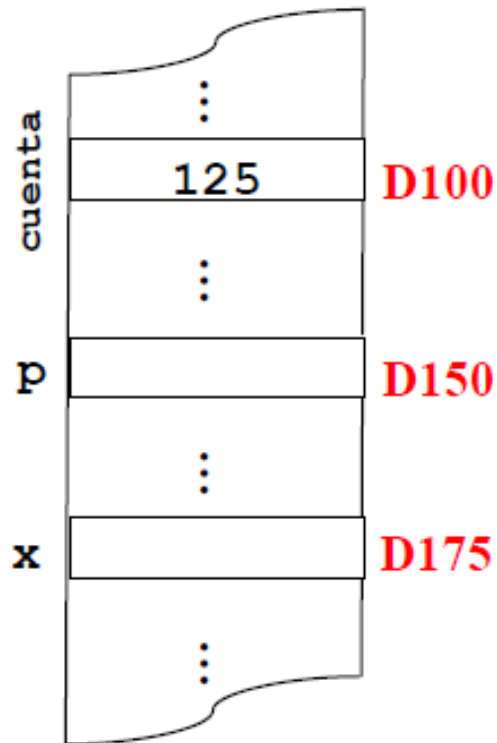
Nombre de la variable	Dirección de memoria	Variable en memoria
b	0x10004	1250
	0x10005	
x	0x10006	1250
	0x10007	
	0x10008	
p	0x10009	0x10006
	0x1000A	
	0x1000B	



Ejemplo2:

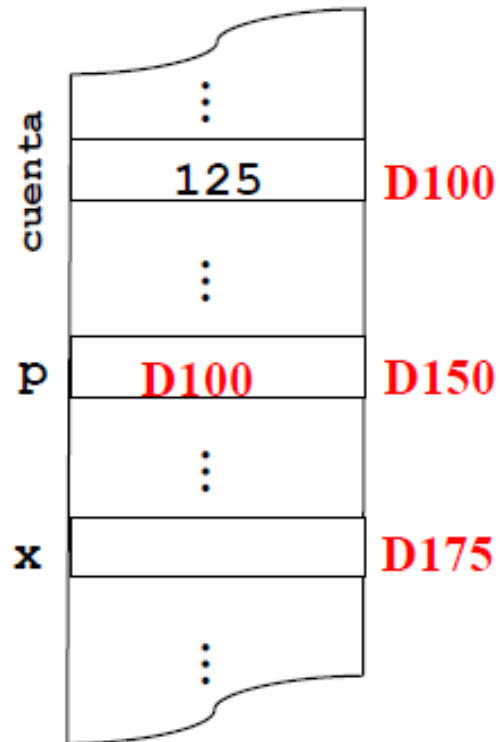
```
int cuenta = 125;  
int *p;  
int x;
```

Memoria Principal



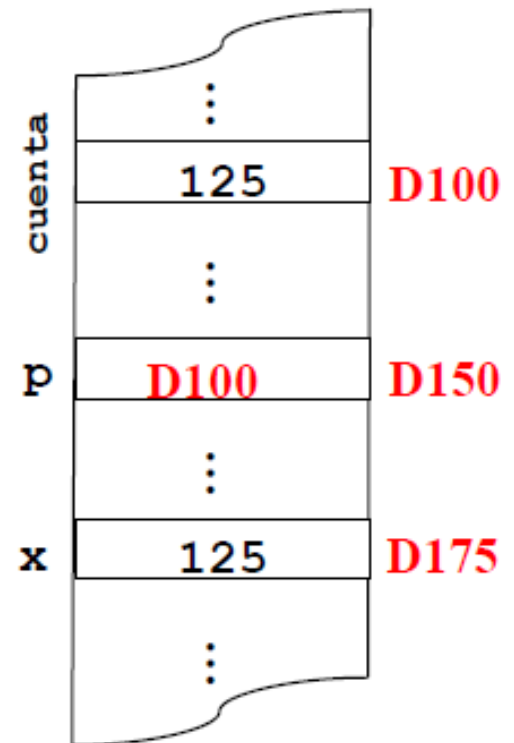
```
p = &cuenta;
```

Memoria Principal



```
x = *p;
```

Memoria Principal



Asignación de Punteros

Respecto a la comparación y a la asignación, los punteros se ajustan a las mismas reglas que cualquier otra variable en C:

- Un puntero puede utilizarse a la derecha de una declaración de asignación para asignar su valor a otro puntero.
- Podemos comparar dos punteros en una expresión relacional.

```
#include <stdio.h>
main() {
    int i=100, *p1, *p2;
    p1 = &i;
    p2 = p1;
    if (p1==p2) /* estamos comparando dos punteros */
        printf("p1 apunta a la misma dirección de memoria que p2");
    *p1 = *p1 + 2; /* El * tiene más prioridad que el + */
    printf ("El valor de *p2 es %d\n", *p2);
    (*p2)++; /* ante la duda de prioridades usamos parentesis */
    printf("El valor de *p1 es %d\n", *p1);
    i--;
    printf("El valor de i es %d\n", i);
}
```

A un puntero se le puede asignar:

- El **valor de otro puntero**, del mismo tipo.
- La **dirección de memoria** de una **variable** cuyo tipo coincida en el tipo_base del puntero.

Ejemplo:

```
int x=15, *p1=NULL, *p2=NULL;
```

```
p1 = &x; // Se le asigna la dirección de memoria de x
```

```
p2 = p1; // Se le asigna el valor de p1
```



Los dos apuntan a la misma variable

Inicialización de Punteros

Al igual que otras variables, C no inicializa los punteros cuando se declaran y es preciso inicializarlos antes de su uso.

TODO PUNTERO DEBE INICIALIZARSE, ya que en caso contrario tras ser declarado apuntaría a cualquier sitio (PELIGROSO) → al usarlo puede p.ej. modificar una parte de la memoria reservada a otra variable

Si aún no sabemos dónde debe apuntar, se le asignará el valor NULL (nulo) → No apunta a ningún sitio en especial.

Ejemplo: `int *p = NULL;`

Las variables tipo puntero deben apuntar al tipo de dato correcto. Si no es así, pueden producirse resultados inesperados. Por ejemplo:

```
main() {  
    unsigned char a=255; /* 255 en binario es 11111111 (8 bits) */  
    char *p; /* p es un puntero a un signed char (por defecto) */  
    p = &a; /* almaceno en p la dir de memoria de la variable a */  
    printf("%i", *p); /* ¡muestra el valor -1, en vez de 255! */  
}
```

```
main() {  
    unsigned int a=269, b; /* 269 en binario: 100001101 (9 bits) */  
    unsigned char *p; /* p: puntero a unsigned char (8 bits) */  
    p = &a; /* p supone que a es unsigned char→solo coge 8 bits */  
    b = *p; /* asigno a b el valor que hay en la dir apuntada por p */  
    printf("%i", b); /* ¡muestra el valor 13, en vez de 269! */  
} /* coge solo los 8 1º bits (00001101), cuyo valor es es 13 */
```

Problemas con los Punteros

Un puntero con un valor erróneo es **MUY PELIGROSO**, y el **ERROR** más **DIFÍCIL DE DEPURAR**, porque en compilación **NO SE DETECTA**, y los errores tras ejecutar pueden **DESPISTAR**.



Hay que hacer caso a los **AVISOS (WARNINGS)** ya que una sospecha sobre un **PUNTERO** puede provocar un gravísimo error en ejecución.

Ejemplo de puntero que no apunta al tipo correcto

El tipo base del puntero determina el tipo de datos al que apunta el puntero y por tanto es el que le indica al compilador cuantos bytes debe transferir para cualquier asignación que usa un puntero. Como podemos asignar cualquier dirección a un puntero, no se produce error cuando asignamos a un puntero, la dirección de una variable de un tipo distinto.

```
#include <stdio.h>
main() {
    float x = 55.4;
    int *p;          /* p es un puntero a un entero */
    p = &x;          Error: "cannot convert 'float*' to 'int*' in assignment"
    printf("El valor correcto es: %f\n", x);
    printf("Valor apuntado (como float): %f \n", *p);
    printf("Valor apuntado (como int) : %i \n", *p);
}
```

El valor correcto es: 55.400000

Valor apuntado (como float): 508524232706.400024

Valor apuntado (como int) : -26214

El error es debido a que **p** es un **puntero a int** y hacemos que apunte a un **float**.

Ejemplo de puntero no inicializado

```
main() {  
    int i, *p;  
    i = 50;  
    *p = i; /* ¡ERROR! ¿A dónde apunta el puntero p? */  
    printf("El valor de i es %i \n", i);  
    printf("El valor de *p es %i \n", *p);  
}
```

El valor de i es 50
El valor de *p es 50

Ponemos el valor de i (50) en la dir de memoria a la que apunta p, pero ¿a dónde apunta p?. Si p apunta a una zona libre no hay problema, pero si es a una zona donde hay datos entonces habrá problemas.

Lo peor en este caso es que el compilador no muestra ningún mensaje de error o de aviso.

Ejemplo de puntero usado de forma incorrecta

Este error suele ocurrir cuando olvidamos poner el operador de dirección (&). Por ejemplo, consideremos el siguiente programa:

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

```
main() {
```

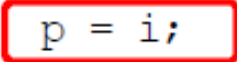
```
    int i, *p;
```

```
    i = 50;
```

```
    p = i;
```

El valor de i es 50

El valor de *p es -3578

 Error: "invalid conversion from 'int' to 'int*'"

```
    printf("El valor de i es %i \n", i);
```

```
    printf("El valor de *p es %i \n", *p);
```

```
}
```

Lo correcto sería haber escrito

```
p = &i;
```

El error es debido a que hacemos que p apunte a la dir 50 de memoria, en vez de a la dir de memoria donde se encuentra la variable i.

Ejercicio Punteros

Ingresa dos valores enteros, realiza las siguientes operaciones utilizando punteros (solo usar las variables para asignar los valores)

- Identificar las direcciones de Memoria de las dos variables
- Acumular al primer elemento, el valor del segundo. Visualizar su valor de los elementos
- El segundo elemento debe ser igual a su contenido multiplicado por el primer valor. Visualizar los valores.
- Obtener la suma de ambos valores y mostrarlos

ESTRUCTURAS

TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR: Estructuras

Introducción

Hemos visto anteriormente el tipo de dato compuesto **ARRAY** definido como una colección de elementos del mismo tipo.

Nombres =

"Ana"	"Pedro"	"Antonio"	...	"Luis"
0	1	2		N-1

Elementos
de tipo cadena

Teléfonos =

6129215	6154215	6144258	...	6165024
---------	---------	---------	-----	---------

Elementos
de tipo entero

En algunos casos nos puede interesar trabajar con colecciones de elementos de distinto tipo:

Alumno =

"Ana"	6129215	1	"Sevilla"
-------	---------	---	-----------

Nombre
(tipo cadena)

Teléfono
(tipo int)

Curso
(tipo int)

Lugar de nacimiento
(tipo cadena)

ESTRUCTURA

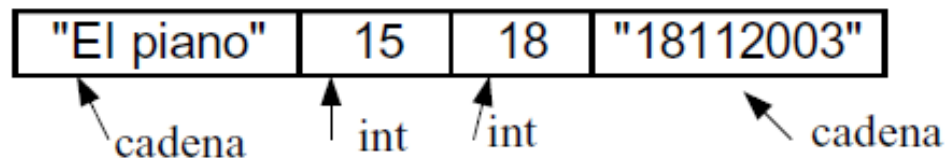
TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR: Estructuras

Concepto de estructura:

- Una **estructura** es una colección de uno o más elementos, cada uno de los cuales puede ser de un tipo de dato diferente.
- Cada elemento de la estructura se denomina **miembro**.
- Una estructura puede contener un número ilimitado de miembros.
- A las estructuras también se las llama **registros**.

Ejemplo:

Podemos crear una estructura llamada **disco** que contiene 4 miembros: **título del disco**, **número de canciones**, **precio** y **fecha de compra**.



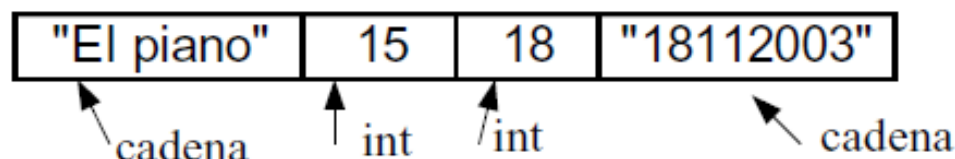
TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR: Estructuras

Concepto de estructura:

- Una **estructura** es una colección de uno o más elementos, cada uno de los cuales puede ser de un tipo de dato diferente.
- Cada elemento de la estructura se denomina **miembro**.
- Una estructura puede contener un número ilimitado de miembros.
- A las estructuras también se las llama **registros**.

Ejemplo:

Podemos crear una estructura llamada **disco** que contiene 4 miembros: **título del disco**, **número de canciones**, **precio** y **fecha de compra**.



TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR: Estructuras

Ejemplos:

Para definir el tipo de dato **disco** como una estructura con 4 miembros:

```
void main( )
{
    ....
    struct disco
    {
        char titulo[30];
        int num_canciones;
        float precio;
        char fecha_compra[8];
    };

    ...
}
```

Declaración del tipo de dato **complejo** como una estructura con 2 miembros.

```
void main( )
{
    ....
    struct complejo
    {
        int real;
        int imaginaria;
    };

    ...
}
```

Declaración de variables de tipo estructura:

Una vez definido el tipo de dato estructura, necesitamos declarar variables de ese tipo (como para cualquier tipo de dato !!!).

Existen dos formas diferentes:

➡ En la definición del tipo de datos estructura.

```
struct complejo
{
    int real;
    int imaginaria;
} comp1, comp2, comp3 ;
```

➡ Como el resto de las variables.

```
...
complejo comp4, comp5 ;
...
```

- ◆ Los miembros de cada variable se almacenan en posiciones consecutivas en memoria.
- ◆ El compilador reserva la memoria necesaria para almacenar las 5 variables.

Inicialización de variables de tipo estructura:

Las variables de tipo estructura las podemos inicializar de dos formas:

1. Inicialización en el cuerpo del programa: Lo veremos más adelante.

2. Inicialización en la declaración de la variable:

Se especifican los valores de cada uno de los miembros entre llaves y separados por comas.

```
struct complejo
{
    int real;
    int imaginaria;
} comp1= {25, 2} ;
```

```
...
complejo comp4 = {25, 2} ;
...
```

TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR: Estructuras

Inicialización de variables de tipo estructura:

```
void main( )
{
    ....
    struct disco
    {
        char titulo[30];
        int num_canciones;
        float precio;
        char fecha_compra[8];
    };
    ....
    disco cd = { "El piano", 15, 18, "18112003" };
    ...
}
```

Mas ejemplos de inicialización
de variables de tipo **disco**

Definición de la estructura
(formato)

¿Cuánta memoria reserva
el compilador ?

cd =

"El piano"

15

18

"18112003"

TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR: Estructuras

Acceso a los miembros de una variable de tipo estructura:

Una vez que hemos declarado una variable de tipo estructura, podemos acceder a los miembros de dicha variable:

cd =

"El piano"	15	18	"18112003"
------------	----	----	------------

Nos puede interesar *modificar* la información de alguno de los miembros, *recuperar* información para imprimirla por pantalla, etc.

El acceso a los miembros se puede hacer de dos formas:

Utilizando el operador punto (.)

Selector
directo

variable.miembro

cd.titulo , cd.precio , cd.num_canciones

Utilizando el operador puntero (->)

Selector
indirecto

variable -> miembro

Lo veremos
más
adelante


Acceso a los miembros de una variable de tipo estructura:

```
struct disco
{
    char titulo[30];
    int num_canciones;
    float precio;
    char fecha_compra[8];
};

....
disco cd;

...
cd.titulo = "El piano";
cd.num_canciones = 15;
cd.precio = 18;
cd.fecha_compra = "18112003";
...
```

Inicialización de la
variable cd1



cd =

"El piano"

15

18

"18112003"

TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR: Estructuras

Acceso a los miembros de una variable de tipo estructura:

```
struct disco
{
    char titulo[30];
    int num_canciones;
    float precio;
    char fecha_compra[8];
};
```

```
.....
disco cd;
```

```
...
cout << "\n Introduzca título" ;
cin.getline ( cd.titulo, 30 ) ;
cout << "\n Introduzca precio" ;
cin>> cd.precio ;
```

```
....
cout << cd.titulo;
precio_final = cd.precio - 10;
cd.precio = precio_final;
```

Almacenar información en la
variable cd
mediante el teclado

Recuperar y modificar información
de la variable cd

Uso de variables de tipo estructura en asignaciones:

Se puede asignar una estructura a otra de la siguiente manera:

```
struct disco
{
    char titulo[30];
    int num_canciones;
    float precio;
    char fecha_compra[8];
};
.....
disco cd1 = { "El piano", 15, 18, "18112003" };
....
disco cd2, cd3;

cd2 = cd1;
cd2 = cd3 = cd1;
```

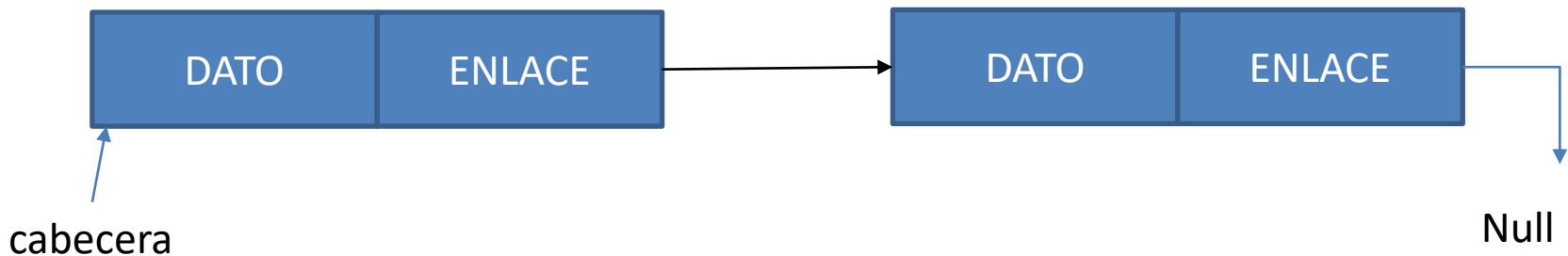
Estructuras Dinámicas

Estructuras Dinámicas

- Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales a las que suelen enfrentarse nuestros programas.
- Nos permiten crear estructuras de datos muy flexibles, ya sea en cuanto al orden, la estructura interna o las relaciones entre los elementos que las componen.

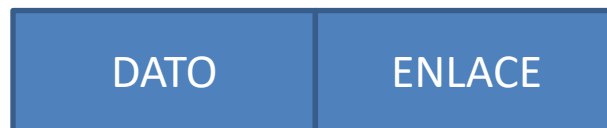
¿Qué es una lista?

- Una lista es una estructura de datos que nos permite agrupar elementos de una manera organizada. Las listas al igual que los algoritmos son importantísimas en la computación y críticas en muchos programas informáticos.
- Una lista enlazada es un conjunto de elementos llamados nodos en los que cada uno de ellos contiene un dato y también la dirección del siguiente nodo.
- El orden de los mismos se establece mediante punteros.
- Cada componente de la lista debe incluir un puntero que indique donde puede encontrarse el siguiente componente por lo que el orden relativo de estos puede ser fácilmente alterado modificando los punteros lo que permite, a su vez, añadir o suprimir elementos de la lista.
- El primer elemento de la lista es la **cabecera**, que sólo contiene un puntero que señala el **primer elemento de la lista**.
- El último nodo de la lista apunta a NULL (nulo) porque no hay más nodos en la lista.
- Se usará el término NULL para designar el final de la lista.



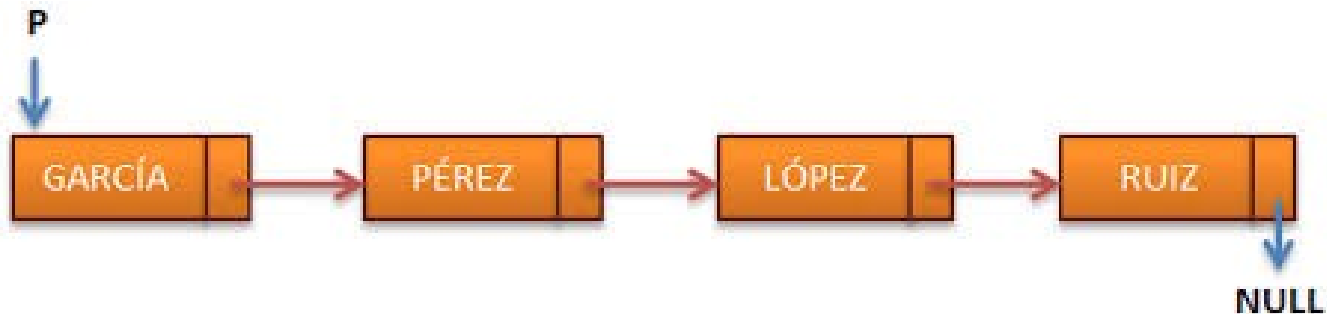
LISTAS ENLAZADAS

- Una lista enlazada o encadenada es una colección de elementos ó nodos, en donde cada uno contiene datos y un enlace.
- Un nodo es una secuencia de caracteres en memoria dividida en campos (de cualquier tipo).
- Un nodo siempre contiene la dirección de memoria del siguiente nodo de información si este existe.
- Un apuntador es la dirección de memoria de un nodo



LISTAS ENLAZADAS

- El campo ENLACE, que es de tipo puntero, es el que se usa para establecer el enlace con el siguiente nodo de la lista.
- Si el nodo fuera el último, este campo recibe como valor NULL (vacío).



Listas Enlazadas frente a Arreglos

Las listas enlazadas tienen las siguiente ventajas sobre los arrays:

- Las listas por ser una estructura de datos dinámica, el tamaño de la misma puede cambiar durante la ejecución del programa a diferencia con los arreglos.
- No tener que redimensionar la estructura y poder agregar elemento tras elemento indefinidamente.

Arreglos frente a Listas Enlazadas

En contraste, los arrays ofrecen las siguiente ventajas sobre las listas enlazadas:

- Los elementos de los arrays ocupan menos memoria que los nodos porque no requieren campos de enlace.
- Los arrays ofrecen un acceso más rápido a los datos, mediante índices basados en enteros.

Ejemplo

1EJEMPLO DE LISTA ENLAZADA

255-> 60-> 31-> 5-> 4-> 51-> 9-> 27-> 68-> 62-> NULL

3

4Internamente:

5Nodo-> Dato: 55 Direccion: 0x3d2c00 Siguiente: 0x3d2c80

6Nodo-> Dato: 60 Direccion: 0x3d2c80 Siguiente: 0x3d2c90

7Nodo-> Dato: 31 Direccion: 0x3d2c90 Siguiente: 0x3d2ca0

8Nodo-> Dato: 5 Direccion: 0x3d2ca0 Siguiente: 0x3d2cb0

9Nodo-> Dato: 4 Direccion: 0x3d2cb0 Siguiente: 0x3d2cc0

10Nodo-> Dato: 51 Direccion: 0x3d2cc0 Siguiente: 0x3d3ab8

11Nodo-> Dato: 9 Direccion: 0x3d3ab8 Siguiente: 0x3d3ac8

12Nodo-> Dato: 27 Direccion: 0x3d3ac8 Siguiente: 0x3d3ad8

13Nodo-> Dato: 68 Direccion: 0x3d3ad8 Siguiente: 0x3d3ae8

14Nodo-> Dato: 62 Direccion: 0x3d3ae8 Siguiente: 0

Estructura de un elemento Lista

Las estructuras de datos están compuestas de otras pequeñas estructuras llamadas NODOS o elementos, que agrupan los datos con los que trabajará nuestro programa y además uno o más punteros autoreferenciales, es decir, punteros a objetos del mismo tipo nodo.

Una estructura básica de un nodo para crear listas de datos seria:

```
struct nodo {  
    int dato;  
    struct nodo *otronodo;  
};
```

```
struct nodo { int dato; struct nodo *otronodo; };
```

```
struct nodo {  
    int dato;  
    struct nodo *otronodo;  
};
```

- El campo "otronodo" puede apuntar a un objeto del tipo nodo. De este modo, cada nodo puede usarse como un ladrillo para construir listas de datos, y cada uno mantendrá ciertas relaciones con otros nodos.
- Para acceder a un nodo de la estructura sólo necesitaremos un puntero a un nodo.

Las estructuras dinámicas son una implementación de TADs (Tipos Abstractos de Datos).

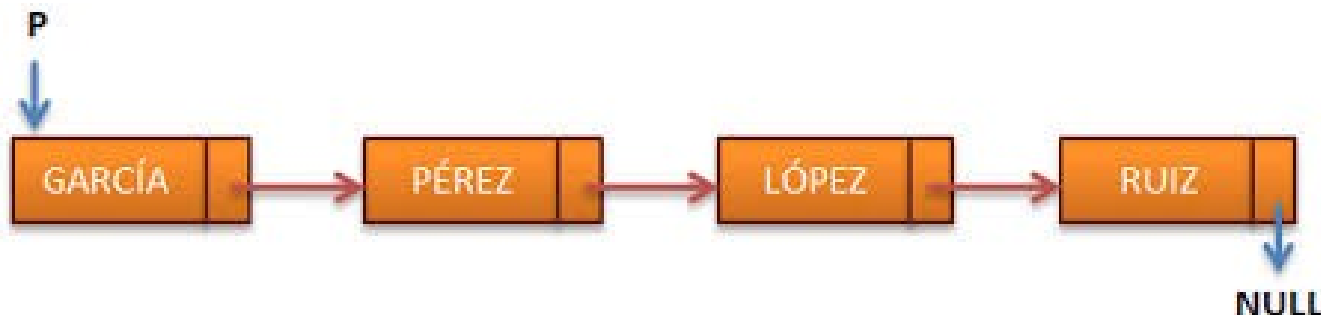
En estos tipos el interés se centra más en la estructura de los datos que en el tipo concreto de información que almacenan.

Tipos de Listas Enlazadas

Simplemente Enlazada

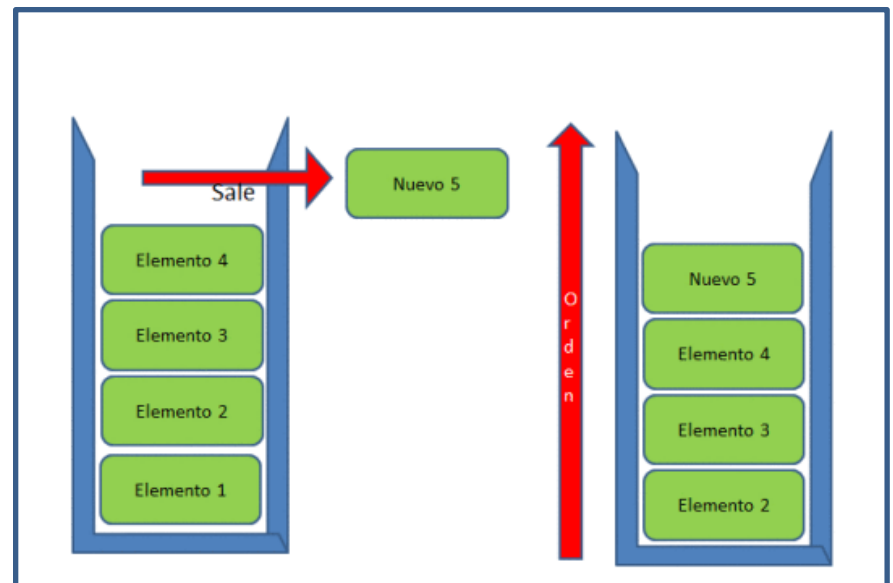
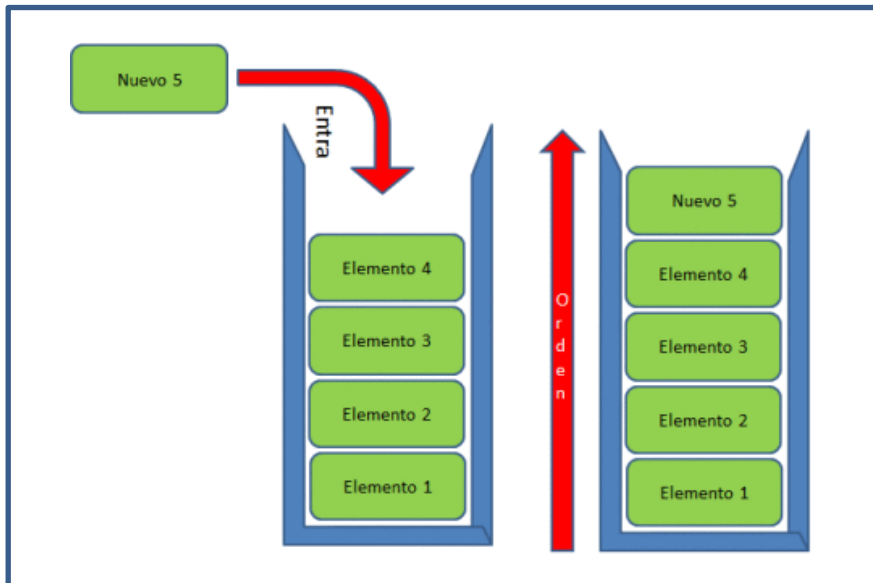
Cada elemento sólo dispone de un puntero, que apuntará al siguiente elemento de la lista o valdrá NULL si es el último elemento.

Se debe conocer la dirección del primer elemento de la lista (p)



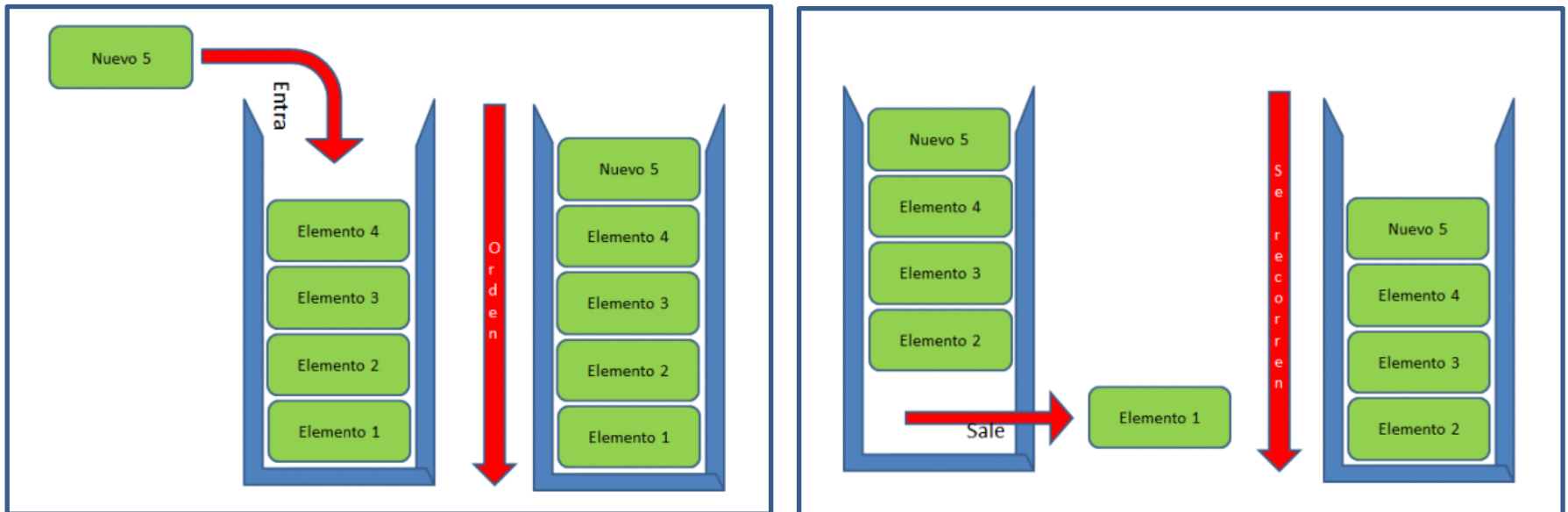
Estructuras Lineales de Acceso Restringido

- **Pilas:** Son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out: el último en entrar es el primero en salir). Los elementos se "amontonan" o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila.



Estructuras Lineales de Acceso Restringido

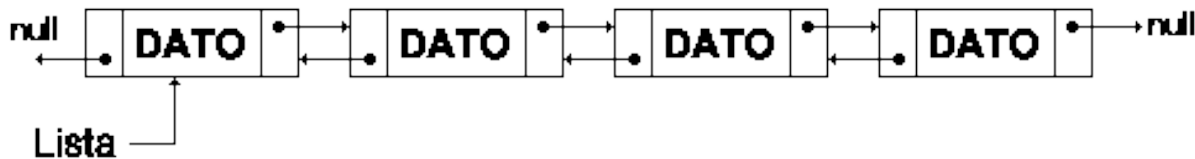
- **Colas:** Otro tipo de listas, conocidas como listas FIFO (First In, First Out: El primero en entrar es el primero en salir). Los elementos se almacenan en fila, pero sólo pueden añadirse por un extremo y leerse por el otro.



Tipos de Listas Enlazadas

Listas doblemente enlazadas:

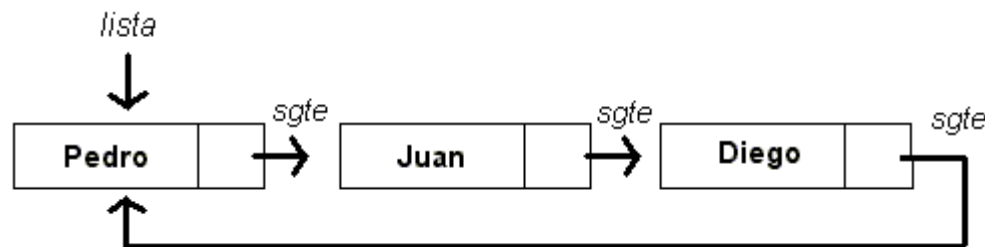
- Cada elemento dispone de dos punteros, uno a punta al siguiente elemento y el otro al elemento anterior. Al contrario que las listas abiertas anteriores, estas listas pueden recorrerse en los dos sentidos.



Tipos de Listas Enlazadas

Listas circulares simplemente enlazada o listas cerradas:

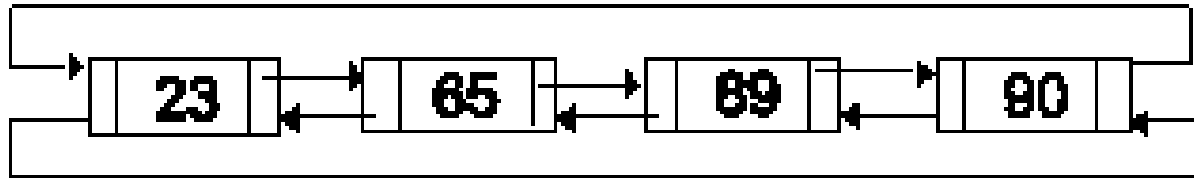
- Son parecidas a las listas abiertas, pero el último elemento apunta al primero. De hecho, en las listas circulares no puede hablarse de "primero" ni de "último". Cualquier nodo puede ser el nodo de entrada y salida.



Tipos de Listas Enlazadas

Listas circulares doblemente enlazada

- Son parecidas a las listas circulares, pero puede recorrerse en ambos sentidos.



Estructura de Datos

- Internas
 - Dinámicas
 - No Lineales
 - Arboles
 - Arboles Binarios
 - Grafos

Estructura de Datos No Lineales

- **Arboles:** cada elemento dispone de dos o más punteros, pero las referencias nunca son a elementos anteriores, de modo que la estructura se ramifica y crece igual que un árbol.
- **Arboles binarios:** son árboles donde cada nodo sólo puede apuntar hasta dos nodos.
 - **Arboles binarios de búsqueda (ABB):** son árboles binarios ordenados. Desde cada nodo todos los nodos de una rama serán mayores, según la norma que se haya seguido para ordenar el árbol, y los de la otra rama serán menores.
 - **Arboles AVL:** son también árboles de búsqueda, pero su estructura está más optimizada para reducir los tiempos de búsqueda.
 - **Arboles B:** son estructuras más complejas, aunque también se trata de árboles de búsqueda, están mucho más optimizados que los anteriores.
- **Grafos:** es el siguiente nivel de complejidad, podemos considerar estas estructuras como árboles no jerarquizados.

Listas enlazadas.

- La lista enlazada es un TDA que nos permite almacenar datos de una forma organizada, al igual que los vectores pero, a diferencia de estos, esta estructura es dinámica, por lo que no tenemos que saber "a priori" los elementos que puede contener.
- En una lista enlazada, cada elemento apunta al siguiente excepto el último que no tiene sucesor y el valor del enlace es NULL. Por ello los elementos son registros que contienen el dato a almacenar y un enlace al siguiente elemento. Los elementos de una lista, suelen recibir también el nombre de NODOS de la lista.

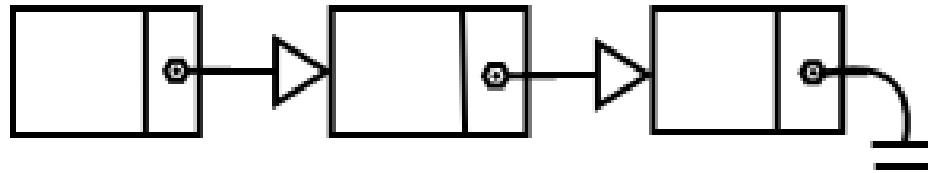
```
struct lista {  
    int dato;  
    lista *siguiente;  
};
```

- 1.- Representa el dato a almacenar. Puede ser de cualquier tipo; en este ejemplo se trata de una lista de enteros.
- 2.- Es un puntero al siguiente elemento de la lista; con este puntero enlazamos con el sucesor, de forma que podamos construir la lista.

Esquema de un nodo y una lista enlazada.



Estructura de un nodo



Lista enlazada

Listas abiertas

- La forma más simple de estructura dinámica es la lista abierta. En esta forma los nodos se organizan de modo que cada uno apunta al siguiente, y el último no apunta a nada, es decir, el puntero del nodo siguiente vale NULL.
- En las listas abiertas existe un nodo especial: el primero. Normalmente diremos que nuestra lista es un puntero a ese primer nodo y llamaremos a ese nodo la cabeza de la lista. Eso es porque mediante ese único puntero podemos acceder a toda la lista.
- Cuando el puntero que usamos para acceder a la lista vale NULL, diremos que la lista está vacía.
- El nodo típico para construir listas tiene esta forma:

```
struct nodo {  
  int dato;  
  struct nodo *siguiente;  
};
```

Declaraciones de tipos para manejar listas en C

Normalmente se definen varios tipos que facilitan el manejo de las listas, en C, la declaración de tipos puede tener una forma parecida a esta:

```
typedef struct _nodo {  
  int dato;  
  struct _nodo *siguiente;  
} tipoNodo;  
typedef tipoNodo *pNodo;  
typedef tipoNodo *Lista;
```

tipoNodo es el tipo de dato para declarar nodos.

pNodo es el tipo para declarar punteros a un nodo.

Lista es el tipo para declarar listas,

Como puede verse, un puntero a un nodo y una lista son la misma cosa. En realidad, cualquier puntero a un nodo es una lista, cuyo primer elemento es el nodo apuntado.



Lista enlazada

Operaciones básicas con listas

Con las listas tendremos un pequeño repertorio de operaciones básicas que se pueden realizar:

- Añadir o insertar elementos.
- Buscar o localizar elementos.
- Borrar elementos.
- Moverse a través de una lista, anterior, siguiente, primero.

Cada una de estas operaciones tendrá varios casos especiales, por ejemplo, no será lo mismo insertar un nodo en una lista vacía, o al principio de una lista no vacía, o la final, o en una posición intermedia.

Insertar elementos en una lista abierta

Insertar un elemento en una lista vacía

- Se parte que se tiene el nodo a insertar y un puntero que apunte a él, además el puntero a la lista valdrá NULL:

Lista vacía

El proceso es muy simple, bastará con que:

1. nodo->siguiente apunte a NULL.
2. Lista apunte a nodo.

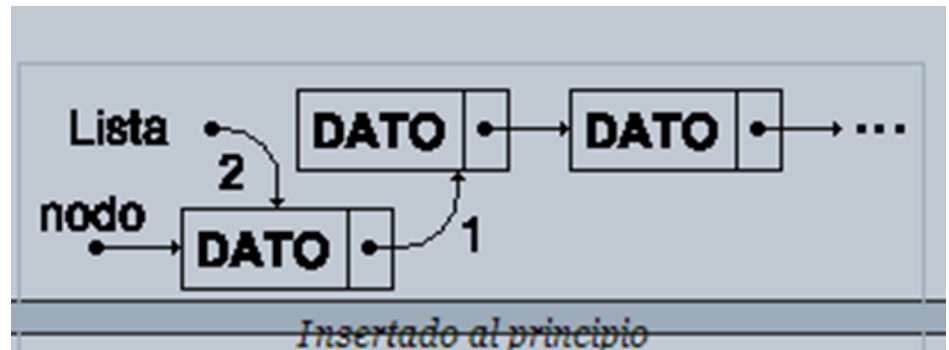
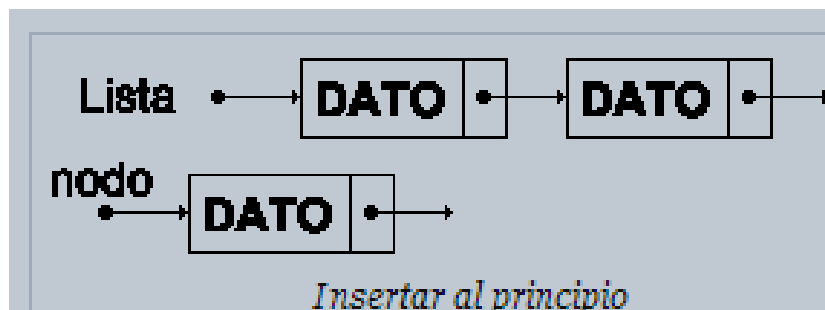
Insertar un elemento en la primera posición de una lista

- Considerando una lista no vacía.
- Se parte de un nodo a insertar, con un puntero que apunte a él, y de una lista no vacía:

Insertado al principio

El proceso sigue siendo muy sencillo:

1. Hacemos que nodo->siguiente apunte a Lista.
2. Hacemos que Lista apunte a nodo.

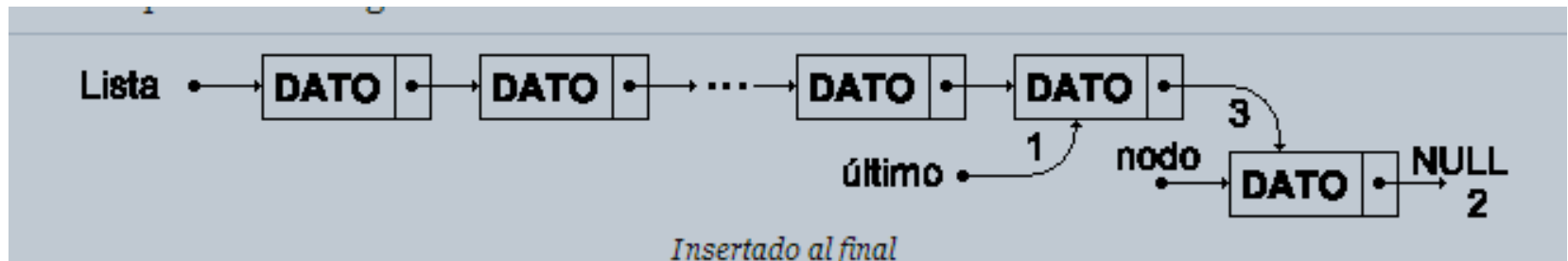


Insertar un elemento en la última posición

- Se parte de considerar una lista no vacía:

Insertar al final

- Necesitamos un puntero que señale al último elemento de la lista. La manera de conseguirlo es empezar por el primero y avanzar hasta que el nodo que tenga como siguiente el valor NULL.
1. Hacer que nodo->siguiente sea NULL.
 2. Hacer que ultimo->siguiente sea nodo.



Insertar un elemento a continuación de un nodo cualquiera de una lista

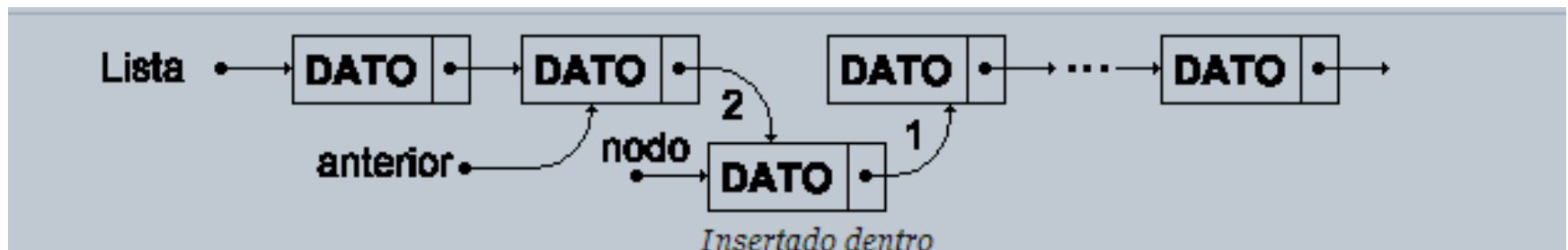
- Ahora el nodo "anterior" será aquel a continuación del cual insertaremos el nuevo nodo:

Insertar dentro

- Suponemos que ya disponemos del nuevo nodo a insertar, apuntado por nodo, y un puntero al nodo a continuación del que lo insertaremos.

El proceso a seguir será:

- Hacer que nodo->siguiente señale a anterior->siguiente.
- Hacer que anterior->siguiente señale a nodo.



Localizar elementos en una lista abierta

- Para recorrer una lista, ya sea buscando un valor particular o un nodo concreto. Las listas abiertas sólo pueden recorrerse en un sentido, ya que cada nodo apunta al siguiente, pero no se puede obtener, por ejemplo, un puntero al nodo anterior desde un nodo cualquiera si no se empieza desde el principio.
- Para recorrer una lista usaremos un puntero auxiliar como índice:
 1. Asignamos al puntero índice el valor de Lista.
 2. Abriremos un bucle que al menos debe tener una condición, que el índice no sea NULL.
 3. Dentro del bucle asignaremos al índice el valor del nodo siguiente al índice actual.

```

typedef struct _nodo {
    int dato;
    struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
...
pNodo indice;
...
indice = Lista;
while(indice) {
    printf("%d\n", indice->dato);
    indice = indice->siguiente;
}
...

```

Supongamos que sólo queremos mostrar los valores hasta que encontremos uno que sea mayor que 100, podemos sustituir el bucle por:

```

...
indice = Lista;
while(indice && indice->dato <= 100) {
    printf("%d\n", indice->dato);
    indice = indice->siguiente;
}
...

```

Localizar elementos en una lista abierta

- Muy a menudo necesitaremos recorrer una lista, ya sea buscando un valor particular o un nodo concreto. Las listas abiertas sólo pueden recorrerse en un sentido, ya que cada nodo apunta al siguiente, pero no se puede obtener, por ejemplo, un puntero al nodo anterior desde un nodo cualquiera si no se empieza desde el principio.

Para recorrer una lista procederemos siempre del mismo modo, usaremos un puntero auxiliar como índice:

1. Asignamos al puntero índice el valor de Lista.
2. Abriremos un bucle que al menos debe tener una condición, que el índice no sea NULL.
3. Dentro del bucle asignaremos al índice el valor del nodo siguiente al índice actual.