



## Tema 2. Estructuras de datos

### Algoritmia

Profesor: Andrés Muñoz

Escuela Politécnica



Andrés Muñoz - Tlf: (+34) 968278821  
Universidad Católica San Antonio de Murcia - Tlf: (+34) 968 27 88 00 info@ucam.edu - [www.ucam.edu](http://www.ucam.edu)

## Índice

---

- Introducción
- Estructuras lineales
- Estructuras no lineales
- Estructuras en memoria secundaria



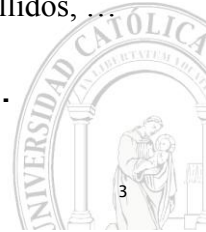
# Introducción

---

- Estructura de datos:
  - *Representación organizada de un conjunto de información.*
  - Manera natural de manejar la información.

Registro Persona    Vs.    Variable nombre, apellidos, ...

- Fuerte relación E.D. y algoritmos.

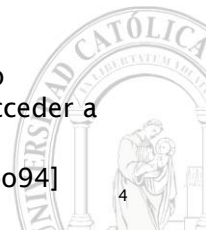


# Introducción

---

- Tipo Abstracto de Datos
  - Se puede definir por partes ...
    - *Tipo*: agrupación de elementos con características similares.
    - *Abstracto*: no concreto, conceptual.
    - *Dato*: información que una computadora puede entender.
  - O de manera más formal ...
    - Un tipo abstracto de datos es un tipo cuya representación como tipo concreto ha sido abstraída y a cuyos datos sólo se puede acceder a través de un conjunto de operaciones.

[Coo94]



# Introducción

---

- Operaciones básicas de una E.D.
  - Inserción
  - Eliminación
  - Búsqueda
  - Vaciado
  - Inicialización



# Introducción

---

- Operaciones básicas de una E.D.
  - Inserción
    - Incrementa los elementos de la estructura.
    - Devuelve la posición del primer elemento.
  - Eliminación
  - Búsqueda
  - Vaciado
  - Inicialización



# Introducción

---

- Operaciones básicas de una E.D.
  - Inserción (pasos generales)
    - Comprobar que hay espacio.
    - Crear espacio para el nuevo elemento.
    - Insertar el elemento.
    - Devolver la primera posición.
  - Eliminación
  - Búsqueda
  - Vaciado
  - Inicialización



# Introducción

---

- Operaciones básicas de una E.D.
  - Inserción
  - Eliminación
    - Reduce los elementos de la estructura.
    - Devuelve la posición del primer elemento.
  - Búsqueda
  - Vaciado
  - Inicialización



# Introducción

---

- Operaciones básicas de una E.D.
  - Inserción
  - Eliminación (pasos generales)
    - Comprobar que hay elementos.
    - Localizar el elemento a eliminar.
    - Liberar la memoria.
    - Devolver la primera posición.
  - Búsqueda
  - Vaciado
  - Inicialización



# Introducción

---

- Operaciones básicas de una E.D.
  - Inserción
  - Eliminación
  - Búsqueda
    - Devuelve el elemento que cumpla el criterio.
  - Vaciado
  - Inicialización



# Introducción

---

- Operaciones básicas de una E.D.
  - Inserción
  - Eliminación
  - Búsqueda
  - Vaciado
    - Elimina todos los elementos de la estructura.
  - Inicialización



# Introducción

---

- Operaciones básicas de una E.D.
  - Inserción
  - Eliminación
  - Búsqueda
  - Vaciado
  - Inicialización
    - Crea una estructura vacía



# Introducción

---

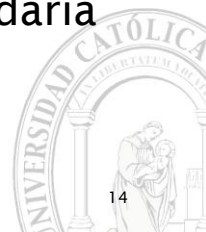
- Tipos de estructuras de datos
  - Dinámicas vs. Estáticas
  - Lineales vs. Arborescentes



# Índice

---

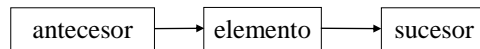
- Introducción
- Estructuras lineales
- Estructuras no lineales
- Estructuras en memoria secundaria



# Estructuras lineales

---

- Colección ordenada de datos.
- Cada elemento tiene como máximo un predecesor y un sucesor.



- La diferencia entre las distintas clases de estructuras radica en el modo de acceso.
- Todas ellas se pueden implementar de forma dinámica (punteros) y estática (arrays)



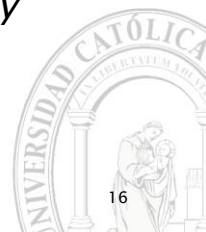
15

# Estructuras lineales

---

- Tipos:
  - Pilas
  - Colas
  - Listas

*Todas ellas en su versión estática y dinámica*



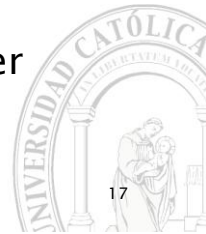
16



# Pila

---

- Política LIFO (Last Input, First Output)
- Similar a una pila de platos
- Ejemplos de uso:
  - Microprocesadores, instrucciones máquina,
  - Llamadas a subrutinas,
  - Editores de texto: botón deshacer



17

# Pila

---

- Operaciones básicas:
  - CREA: crea una pila vacía.
  - VACIA: devuelve un valor cierto si la pila está vacía, y falso en caso contrario.
  - TOPE: devuelve el elemento situado el tope de la pila, sin extraerlo.
  - PUSH: añade un elemento a la pila, quedando éste situado en el tope.
  - POP: suprime el elemento situado en el tope de la pila.



18

# Pila

---

- Especificación Formal

- Sintaxis:

- crea = Pila
    - vacia(Pila) = booleano
    - tope(Pila) = Elemento
    - push(Pila,Elemento) = Pila
    - pop(Pila) = Pila

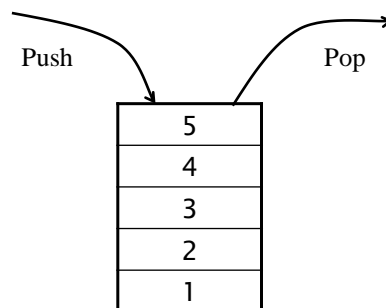
- Semántica:  $\square P = \text{Pila}$ ,  $\square E = \text{Elemento}$ :

- vacia(crea) = cierto
    - vacia(push(P,E)) = falso
    - tope(crea) = error
    - tope(push(P,E)) = E
    - pop(crea) = error
    - pop(push(P,E)) = P



# Pila

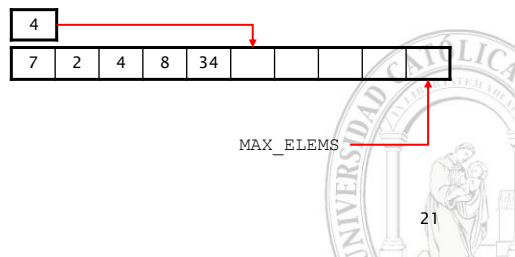
---



# Pila: versión estática

- Registro que contiene
  - Array de datos
  - Un entero que indica la última celda vacía.

```
#define MAX_ELEMS 10
typedef struct pila{
    int cima;
    int valores[MAX_ELEMS];
}tipo_pila;
```

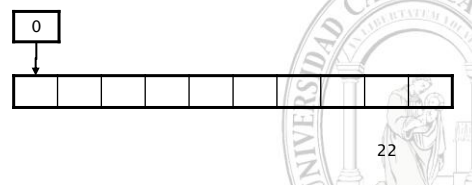


# Pila: versión estática

- Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}
```

```
En main()
int error = 0;
tipo_pila pila, *pto_pila
pto_pila = &pila;
pto_pila = push(pto_pila, 7, &error);
pto_pila = push(pto_pila, 2, &error);
pto_pila = push(pto_pila, 4, &error);
```



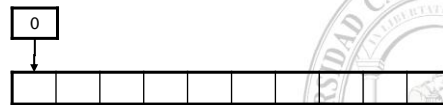
# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS) ← ¿0 < 10?, Sí
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
int error = 0;
tipo_pila pila, *pto_pila
pto_pila = &pila;
pto_pila = push(pto_pila, 7, &error);
pto_pila = push(pto_pila, 2, &error);
pto_pila = push(pto_pila, 4, &error);
```



23

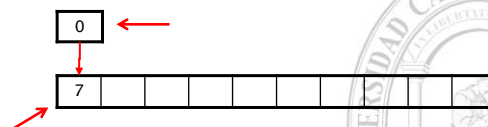
# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)
        p->valores[p->cima++] = i; ←
    else
        *error = -1;
    return p;
}
```

En **main()**

```
int error = 0;
tipo_pila pila, *pto_pila
pto_pila = &pila;
pto_pila = push(pto_pila, 7, &error);
pto_pila = push(pto_pila, 2, &error);
pto_pila = push(pto_pila, 4, &error);
```



24

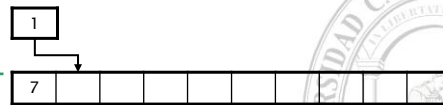
# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
int error = 0;
tipo_pila pila, *pto_pila
pto_pila = &pila;
pto_pila = push(pto_pila,7,&error);
pto_pila = push(pto_pila,2,&error);
pto_pila = push(pto_pila,4,&error);
```



25

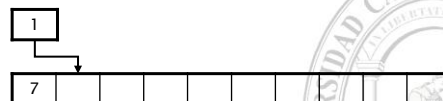
# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
int error = 0;
tipo_pila pila, *pto_pila
pto_pila = &pila;
pto_pila = push(pto_pila,7,&error);
pto_pila = push(pto_pila,2,&error);
pto_pila = push(pto_pila,4,&error);
```



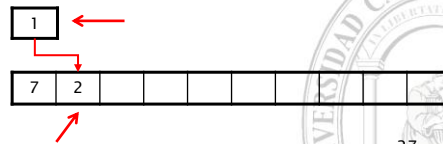
26

# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}

En main()
int error = 0;
tipo_pila pila, *pto_pila
pto_pila = &pila;
pto_pila = push(pto_pila,7,&error);
pto_pila = push(pto_pila,2,&error);
pto_pila = push(pto_pila,4,&error);
```

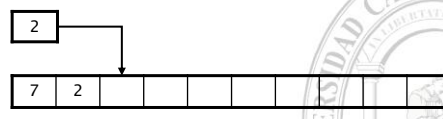


# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}

En main()
int error = 0;
tipo_pila pila, *pto_pila
pto_pila = &pila;
pto_pila = push(pto_pila,7,&error);
pto_pila = push(pto_pila,2,&error);
pto_pila = push(pto_pila,4,&error);
```

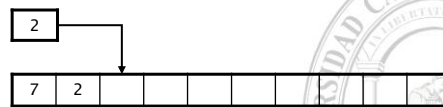


# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)      ← ¿2 < 10?, Si
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}
```

En **main()**  
int error = 0;  
tipo\_pila pila, \*pto\_pila  
pto\_pila = &pila;  
pto\_pila = push(pto\_pila, 7, &error);  
pto\_pila = push(pto\_pila, 2, &error);  
pto\_pila = push(pto\_pila, 4, &error);



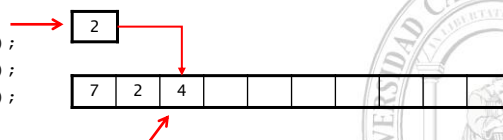
29

# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}
```

En **main()**  
int error = 0;  
tipo\_pila pila, \*pto\_pila  
pto\_pila = &pila;  
pto\_pila = push(pto\_pila, 7, &error);  
pto\_pila = push(pto\_pila, 2, &error);  
pto\_pila = push(pto\_pila, 4, &error);



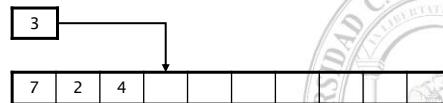
30

# Pila: versión estática

## • Función push (apilar)

```
tipo_pila* push(tipo_pila *p, int i, int *error){
    *error = 0;
    if (p->cima < MAX_ELEMS)
        p->valores[p->cima++] = i;
    else
        *error = -1;
    return p;
}

En main()
int error = 0;
tipo_pila pila, *pto_pila
pto_pila = &pila;
pto_pila = push(pto_pila,7,&error);
pto_pila = push(pto_pila,2,&error);
pto_pila = push(pto_pila,4,&error);
```



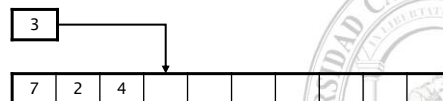
31

# Pila: versión estática

## • Función tope (cima de la pila)

```
int tope(tipo_pila *p, *int *error){
    *error = 0;
    if (p->cima > 0 && p->cima <= MAX_ELEMS)
        return p->valores[p->cima - 1];
    else{
        *error = -1;
        return -1;
    }
}

printf("Tope: %d\n", tope(pto_pila, &error));
```



>4

32



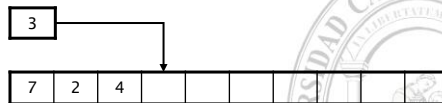
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



33

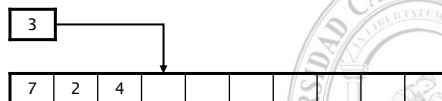
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error); ←
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



34

# Pila: versión estática

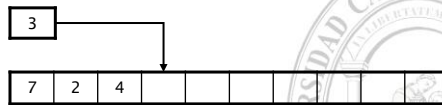
## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)      ← ¿3 > 0?, Sí
        --p->cima;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);

>
```



35

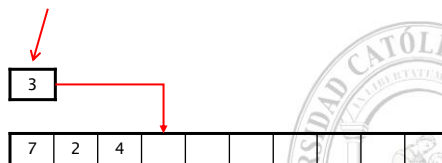
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima;      ←
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



36

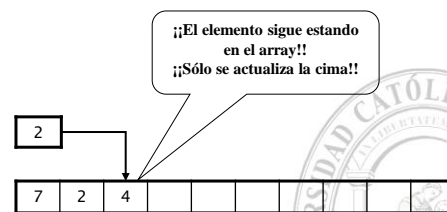
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima; ←
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



37

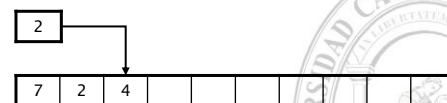
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error); ←
pto_pila = pop(pto_pila, &error);
```



38

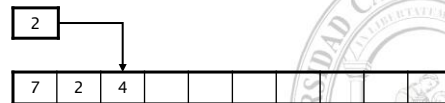
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0) ← ¿2>0?, Sí
        --p->cima;
    else
        *error = -1;
    return p;
}
```

En *main()*

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



39

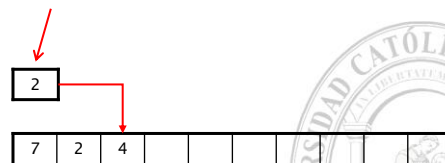
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima; ←
    else
        *error = -1;
    return p;
}
```

En *main()*

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



40

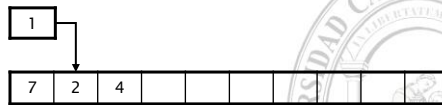
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



41

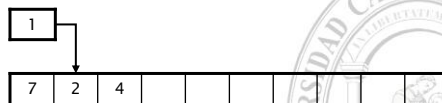
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



42

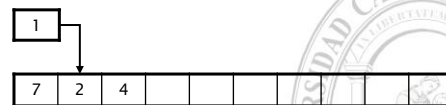
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)      ← ¿1 > 0?, Sí
        --p->cima;
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



43

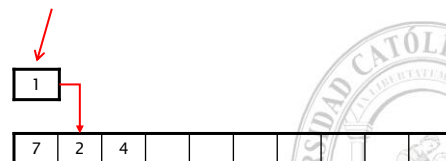
# Pila: versión estática

## • Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima; ←
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



44

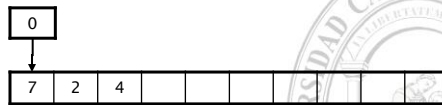
# Pila: versión estática

- Función pop (desapilar)

```
tipo_pila* pop(tipo_pila *p, int *error){
    *error = 0;
    if (p->cima > 0)
        --p->cima; ←
    else
        *error = -1;
    return p;
}
```

En **main()**

```
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
pto_pila = pop(pto_pila, &error);
```



45

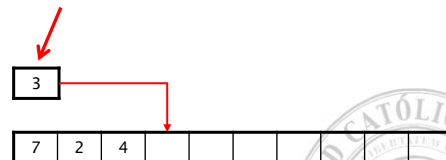
# Pila: versión estática

- Función clear (vaciar)

```
tipo_pila* vaciar(tipo_pila *p){
    p->cima = 0;
    return p;
}
```

En **main()**

```
pto_pila = vaciar(pto_pila);
```



46

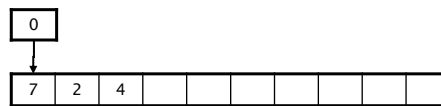
## Pila: versión estática

---

- Función clear (vaciar)

```
tipo_pila* vaciar(tipo_pila *p){  
    p->cima = 0;  
    return p;  
}
```

En **main()**  
`pto_pila = vaciar(pto_pila);`



## Pila: versión dinámica

---

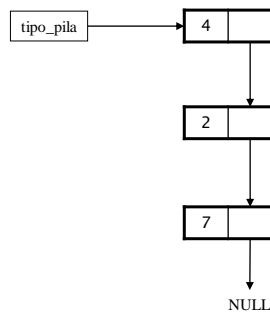
- Se implementa mediante una estructura autorreferenciada con encadenamiento simple.
- Permite disponer de toda la memoria.
- Los datos se almacenan en un campo.
- “tipo\_pila” es un puntero a la última celda insertada.





# Pila: versión dinámica

```
typedef struct pila{
    int valor;
    struct pila *sgte;
}nodo_pila, *tipo_pila;
```



Último elemento insertado

Primer elemento insertado



# Pila: versión dinámica

## • Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL; ←
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

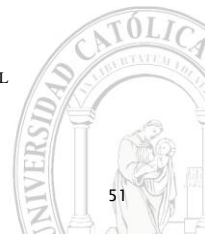


# Pila: versión dinámica

- Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```



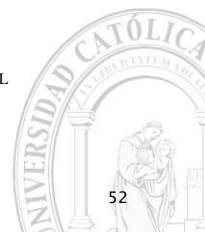
51

# Pila: versión dinámica

- Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```



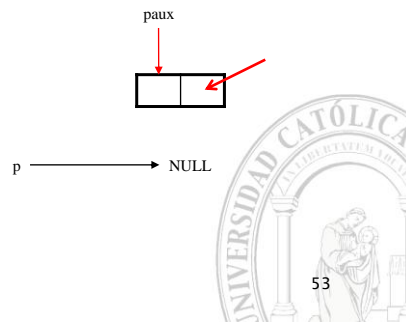
52

# Pila: versión dinámica

## • Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p; ←
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

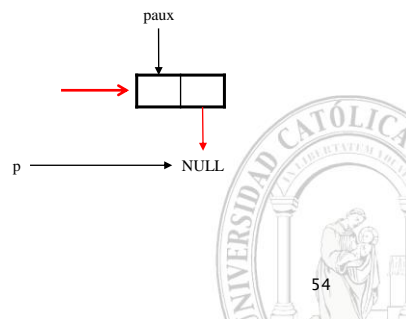


# Pila: versión dinámica

## • Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p; ←
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

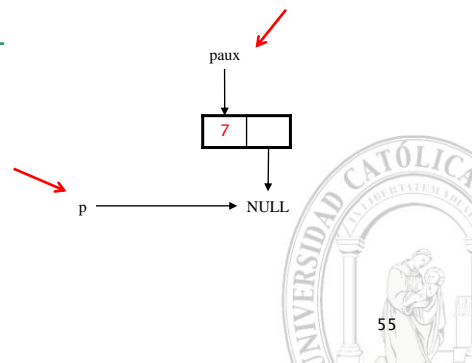


# Pila: versión dinámica

## • Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

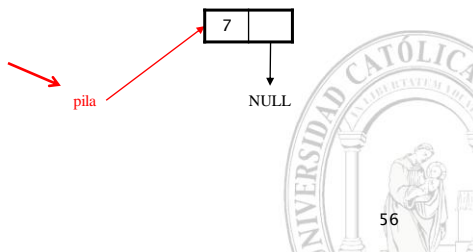


# Pila: versión dinámica

## • Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

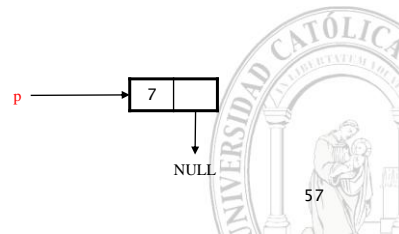


# Pila: versión dinámica

- Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

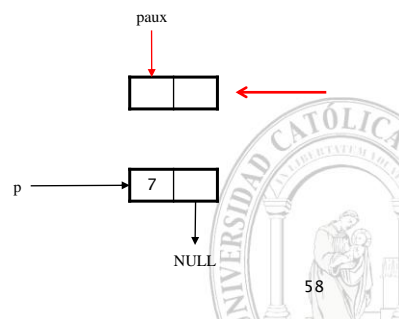


# Pila: versión dinámica

- Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

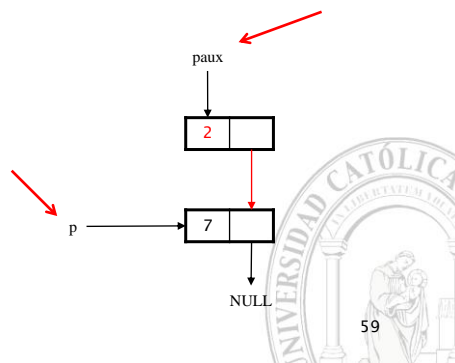


# Pila: versión dinámica

- Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

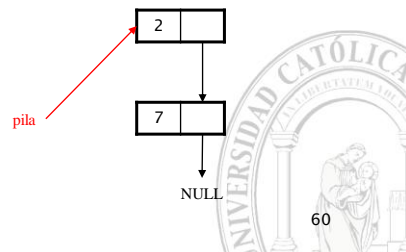


# Pila: versión dinámica

- Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```

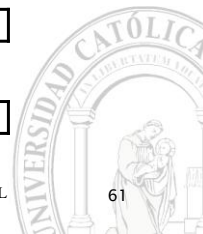
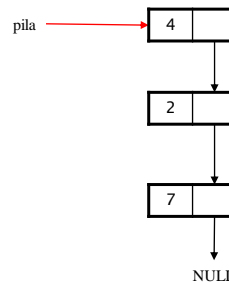


# Pila: versión dinámica

## • Función push (apilar)

```
tipo_pila push(tipo_pila p, int i){
    tipo_pila paux;
    paux = (tipo_pila)malloc(sizeof(nodo_pila));
    paux->sgte = p;
    paux->valor = i;
    return paux;
}
```

```
tipo_pila pila = NULL;
pila = push(pila,7);
pila = push(pila,2);
pila = push(pila,4);
```



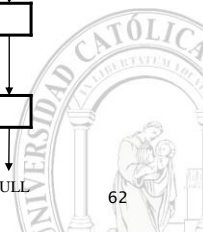
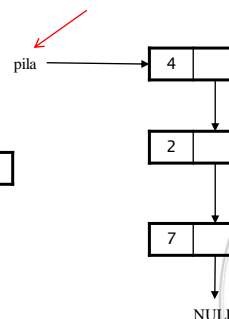
61

# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pop(pila, &i);
pop(pila, &i);
pop(pila, &i);
```



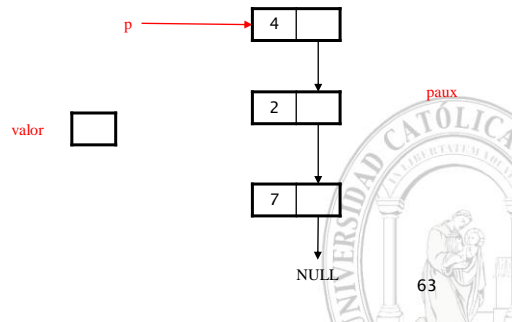
62

# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pop(pila, &i);
pop(pila, &i);
pop(pila, &i);
```

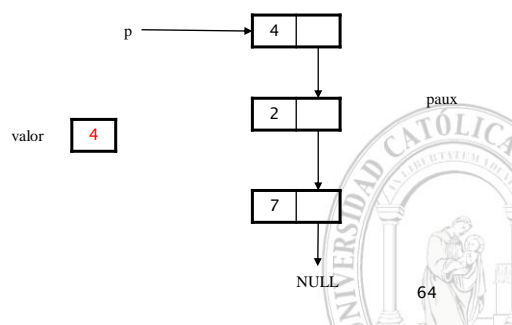


# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pop(pila, &i);
pop(pila, &i);
pop(pila, &i);
```



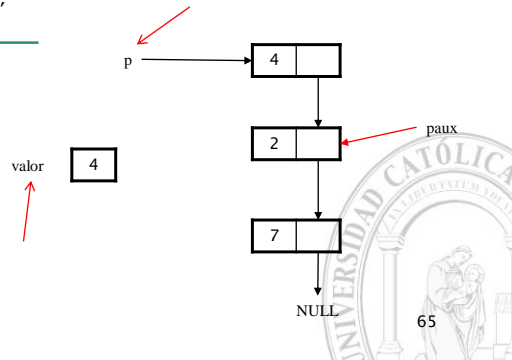


# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pop(pila, &i);
pop(pila, &i);
pop(pila, &i);
```

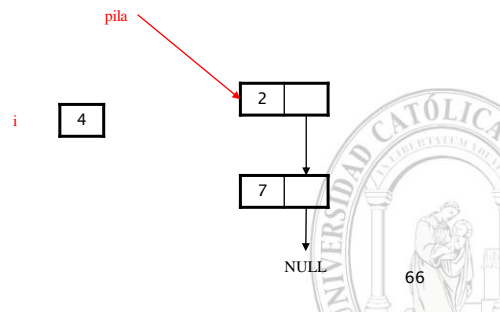


# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pila = pop(pila, &i);
pila = pop(pila, &i);
pila = pop(pila, &i);
```

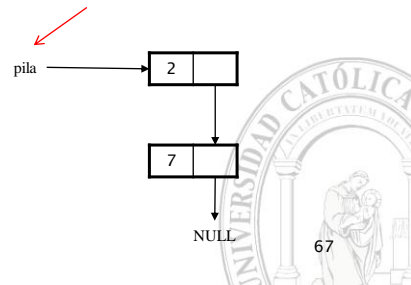


# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pop(pila, &i);
pop(pila, &i);
pop(pila, &i);
```

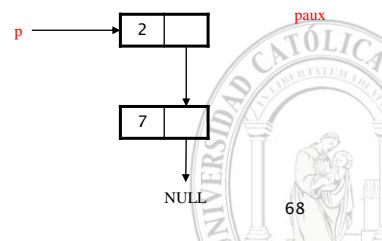


# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pop(pila, &i);
pop(pila, &i);
pop(pila, &i);
```



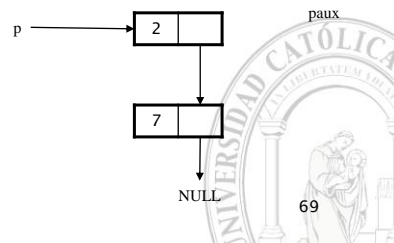
# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte; ←
    free(p);
    return paux;
}
```

```
int i;
pop(pila, &i);
pop(pila, &i);
pop(pila, &i);
```

valor 2



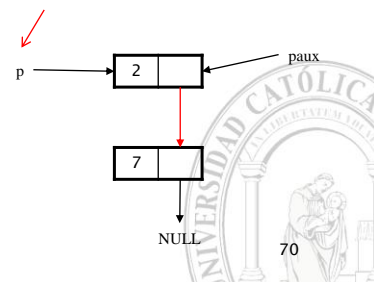
# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p); ←
    return paux; ←
}
```

```
int i;
pop(pila, &i);
pop(pila, &i);
pop(pila, &i);
```

valor 2

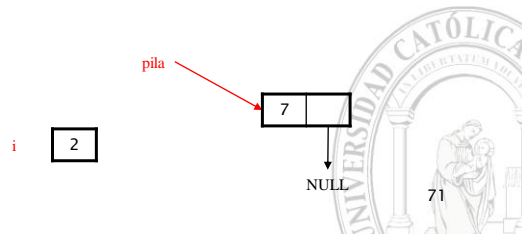


# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pila = pop(pila, &i);
pila = pop(pila, &i); ←
pila = pop(pila, &i);
```

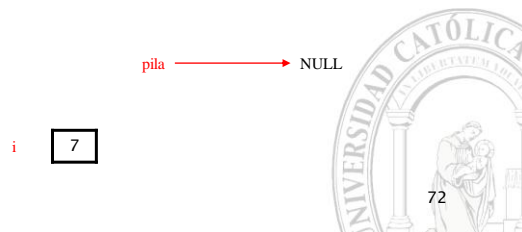


# Pila: versión dinámica

## • Función pop (desapilar)

```
tipo_pila pop(tipo_pila p, int* valor){
    tipo_pila paux;
    if (p == NULL) return NULL;
    *valor = p->valor;
    paux = p->sgte;
    free(p);
    return paux;
}
```

```
int i;
pila = pop(pila, &i);
pila = pop(pila, &i);
pila = pop(pila, &i); ←
```



# Pila: versión dinámica

- Función clear (vaciar)
  - Eliminar todo el contenido de la pila.
  - Nunca hacer *pila = NULL* ¡¡HAY QUE LIBERAR MEMORIA!!

```
tipo_pila vaciar(tipo_pila p){
    tipo_pila paux,taux;
    paux = p ;
    while(paux != NULL){
        taxa = paux;
        paux = paux->sgte;
        free(taux);
    }
    return NULL;
}
```

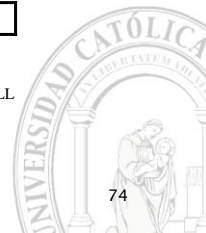
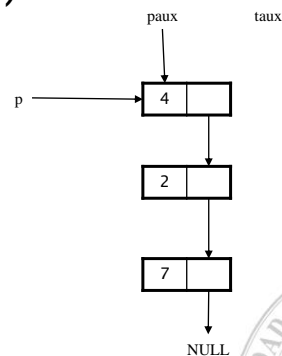


73

# Pila: versión dinámica

- Función clear (vaciar)

```
tipo_pila vaciar(tipo_pila p){
    tipo_pila paux,taux;
    paux = p ;
    while(paux != NULL){
        taxa = paux; ←
        paux = paux->sgte;
        free(taux);
    }
    return NULL;
}
```

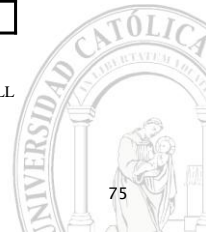
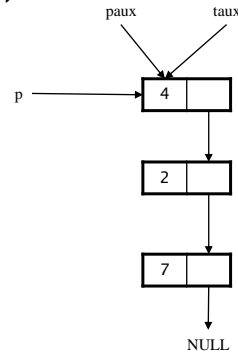


74

# Pila: versión dinámica

- Función clear (vaciar)

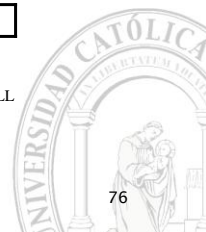
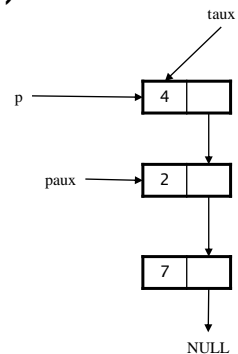
```
tipo_pila vaciar(tipo_pila p){
    tipo_pila paux,taux;
    paux = p ;
    while(paux != NULL){
       iaux = paux;
        paux = paux->sgte;
        free(taux);
    }
    return NULL;
}
```



# Pila: versión dinámica

- Función clear (vaciar)

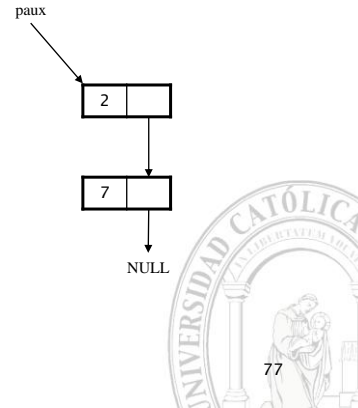
```
tipo_pila vaciar(tipo_pila p){
    tipo_pila paux,taux;
    paux = p ;
    while(paux != NULL){
       iaux = paux;
        paux = paux->sgte;
        free(taux);
    }
    return NULL;
}
```



# Pila: versión dinámica

- Función clear (vaciar)

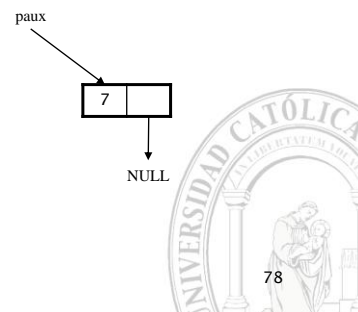
```
tipo_pila vaciar(tipo_pila p){
    tipo_pila paux,taux;
    paux = p ;
    while(paux != NULL){ ←
        taxa = paux;
        paux = paux->sgte;
        free(taux);
    }
    return NULL;
}
```



# Pila: versión dinámica

- Función clear (vaciar)

```
tipo_pila vaciar(tipo_pila p){
    tipo_pila paux,taux;
    paux = p ;
    while(paux != NULL){ ←
        taxa = paux;
        paux = paux->sgte;
        free(taux);
    }
    return NULL;
}
```



# Pila: versión dinámica

---

- Función clear (vaciar)

```
tipo_pila vaciar(tipo_pila p){
    tipo_pila paux,taux;
    paux = p ;
    while(paux != NULL){
        taux = paux;
        paux = paux->sgte;
        free(taux);
    }
    return NULL; ←
}
```

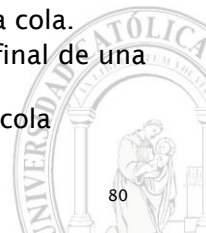
paux → NULL



# Cola

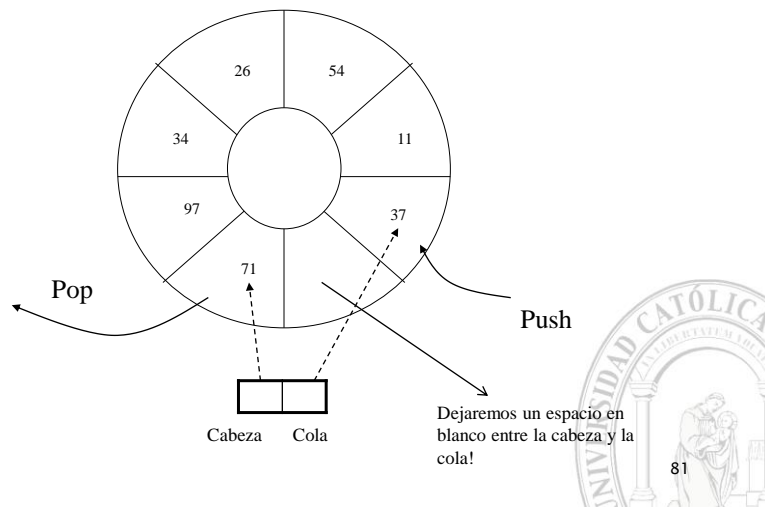
---

- Política FIFO (First Input, First Output)
- Similar a una cola del cine
- Ejemplo de uso:
  - Desplazamiento de la ALU
- Operaciones
  - CREA: Crea una cola vacía.
  - VACIA: Devuelve un valor cierto si la cola está vacía, y falso en caso contrario.
  - PRIMERO: Devuelve el primer elemento de una cola.
  - INSERTA: Añade un elemento por el extremo final de una cola.
  - SUPRIME: Suprime el primer elemento de una cola





# Cola



# Cola

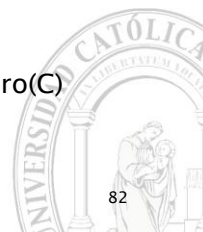
## • Especificación Formal

### – Sintaxis:

- crea = Cola
- vacia(Cola) = booleano
- primero(Cola) = Elemento
- inserta(Cola,Elemento) = Cola
- supprime(Cola) = Cola

### – Semántica:

- vacia(crea) = cierto
- vacia(inserta(C,E)) = falso
- primero(crea) = error
- primero(inserta(C,E)) = si vacia(C) ? E : primero(C)
- supprime(crea) = error
- supprime(inserta(C,E)) = si vacia(C) ? crea : inserta(supprime(C),E)



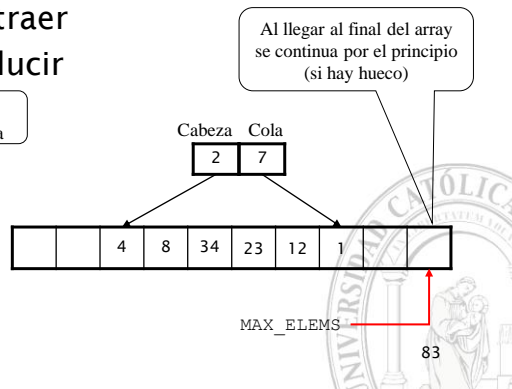
# Cola: versión estática

- Registro que contiene

- Array de datos
- Dos enteros:
  - *cabeza*: para extraer
  - *cola*: para introducir

```
#define MAX_ELEMS 10
typedef struct cola{
    int cabeza, cola;
    int valores[MAX_ELEMS];
}tipo_cola;
```

Ojo! 9 huecos útiles y un espacio entre cabeza y cola



# Cola: versión estática

- Se va a hacer uso de esta función para calcular la posición en el array

```
int suma_uno(int i){
    return ((i+1) % MAX_ELEMS);
}
```

Se utiliza el operador % para limitar la posición máxima dentro del array

Por ejemplo,

si se quiere introducir en la posición 7 =>  $\text{suma\_uno}(7) = (7+1) \% 10 = 8$

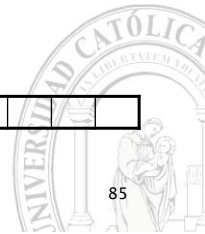
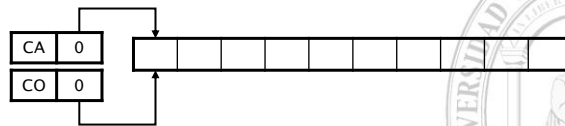
si se quiere introducir en la posición 15 =>  $\text{suma\_uno}(15) = (15+1) \% 10 = 6$

# Cola: versión estática

- Función iniciar

```
void iniciar(tipoCola *c){  
    c->cabeza = 0;  
    c->cola = 0;  
}
```

```
tipoCola cola;  
iniciar(&cola);
```



85

# Cola: versión estática

- Función push (apilar)

```
void push(tipoCola *c, int i){  
    if (suma_uno(c->cola) == c->cabeza){  
        printf("Error: cola llena \n");  
    }else{  
        c->valores[c->cola] = i;  
        c->cola = suma_uno(c->cola);  
    }  
}
```

```
push(&cola,7);  
push(&cola,4);  
push(&cola,2);
```



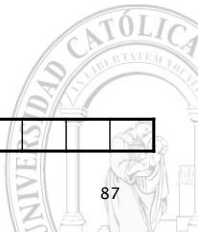
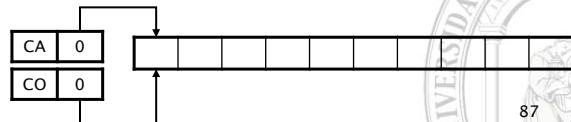
86

# Cola: versión estática

## • Función push (apilar)

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

```
push(&cola,7);
push(&cola,4);
push(&cola,2);
```



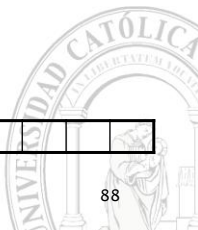
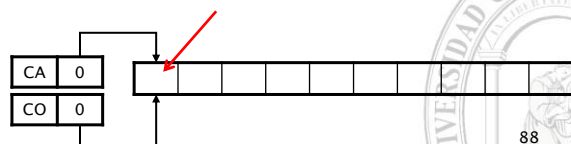
87

# Cola: versión estática

## • Función push (apilar)

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

```
push(&cola,7);
push(&cola,4);
push(&cola,2);
```



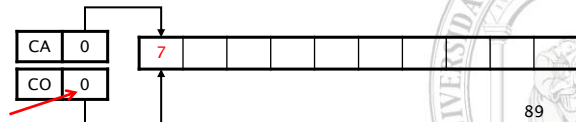
88

# Cola: versión estática

## • Función push (apilar)

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

```
push(&cola,7);
push(&cola,4);
push(&cola,2);
```

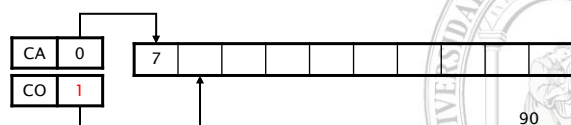


# Cola: versión estática

## • Función push (apilar)

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

```
push(&cola,7);
push(&cola,4);
push(&cola,2);
```



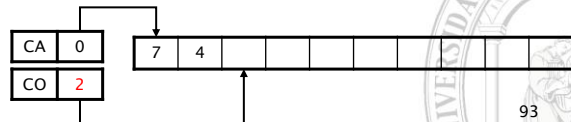


# Cola: versión estática

- Función push (apilar)

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

```
push(&cola,7);
push(&cola,4);
push(&cola,2);
```

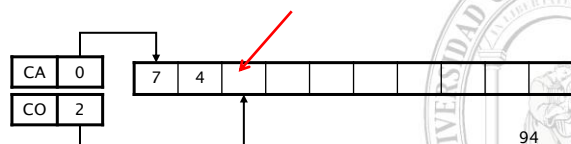


# Cola: versión estática

- Función push (apilar)

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

```
push(&cola,7);
push(&cola,4);
push(&cola,2);
```

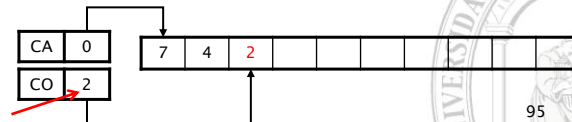


# Cola: versión estática

## • Función push (apilar)

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

```
push(&cola,7);
push(&cola,4);
push(&cola,2);
```



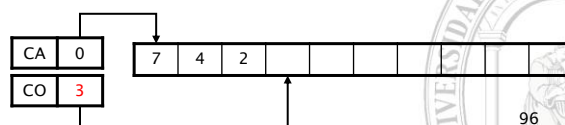
95

# Cola: versión estática

## • Función push (apilar)

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

```
push(&cola,7);
push(&cola,4);
push(&cola,2);
```



96

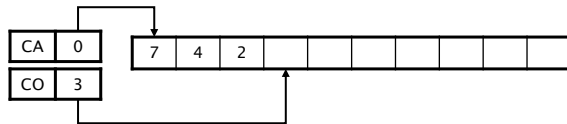


# Cola: versión estática

- Función pop (desapilar)

```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```

→ printf("%d", pop(cola));  
printf("%d", pop(cola));  
printf("%d", pop(cola));



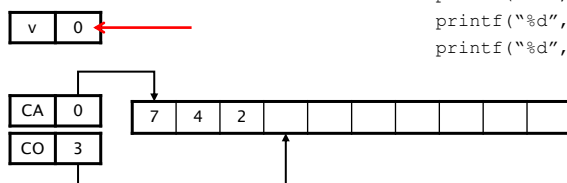
97

# Cola: versión estática

- Función pop (desapilar)

```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```

→ printf("%d", pop(cola));  
printf("%d", pop(cola));  
printf("%d", pop(cola));

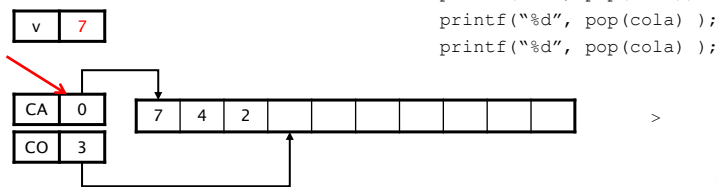


98

# Cola: versión estática

## • Función pop (desapilar)

```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza); ←
    }
    return v;
}
```

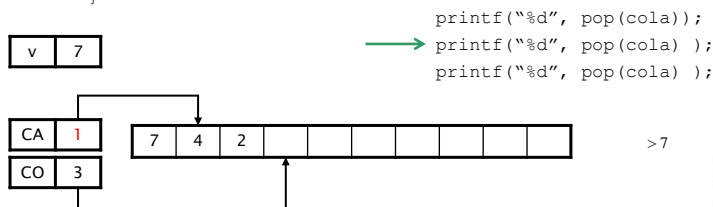


99

# Cola: versión estática

## • Función pop (desapilar)

```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```

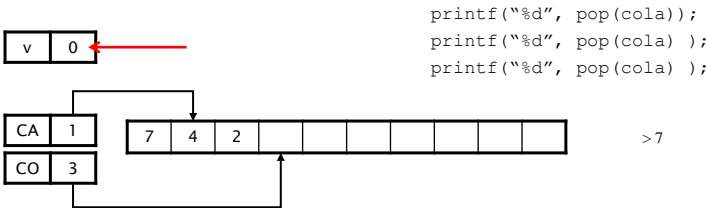


100

# Cola: versión estática

- Función pop (desapilar)

```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```



```
printf("%d", pop(cola));
printf("%d", pop(cola));
printf("%d", pop(cola));
```

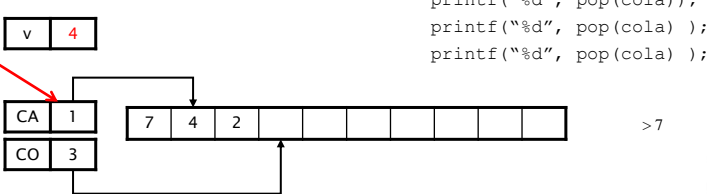


101

# Cola: versión estática

- Función pop (desapilar)

```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```



```
printf("%d", pop(cola));
printf("%d", pop(cola));
printf("%d", pop(cola));
```

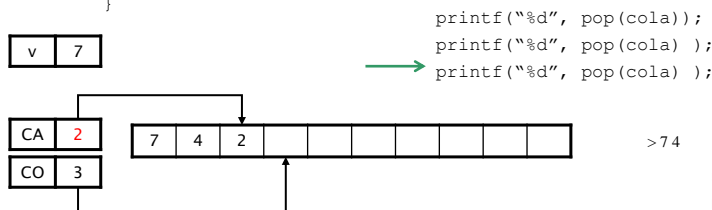


102

# Cola: versión estática

- Función pop (desapilar)

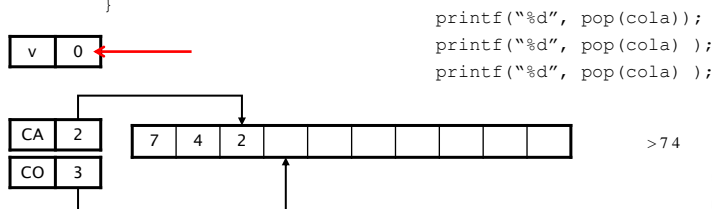
```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```



# Cola: versión estática

- Función pop (desapilar)

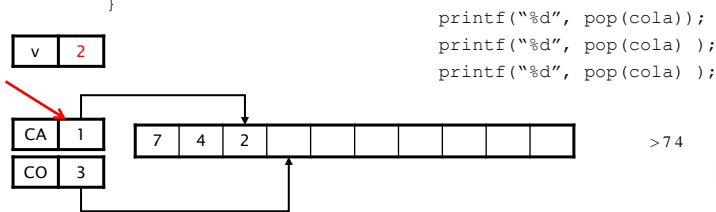
```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```



# Cola: versión estática

- Función pop (desapilar)

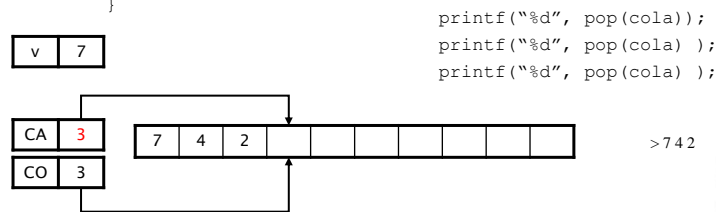
```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```



# Cola: versión estática

- Función pop (desapilar)

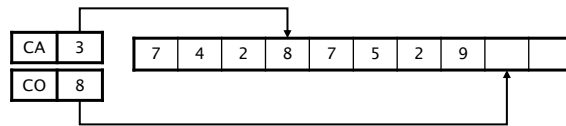
```
int pop(tipoCola *c){
    int v;
    if (es_vacia(c)){
        v=-1;
        printf("Error: cola vacia\n");
    }else{
        v = c->valores[c->cabeza];
        c->cabeza = suma_uno(c->cabeza);
    }
    return v;
}
```



# Cola: versión estática

- ¿Y si llegamos al final del array?

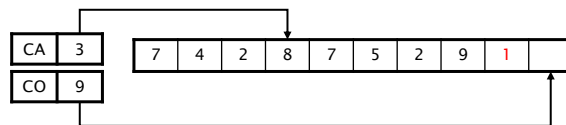
```
→ apilar(cola, 1);  
   apilar(cola, 8);  
   apilar(cola, 6);  
   apilar(cola, 5);  
   apilar(cola, 1);
```



# Cola: versión estática

- ¿Y si llegamos al final del array?

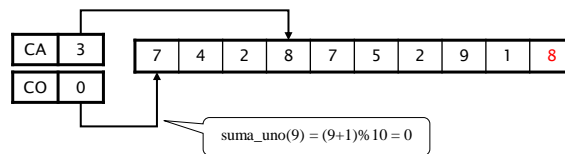
```
→ apilar(cola, 1);  
   apilar(cola, 8);  
   apilar(cola, 6);  
   apilar(cola, 5);  
   apilar(cola, 1);
```



# Cola: versión estática

- ¿Y si llegamos al final del array?

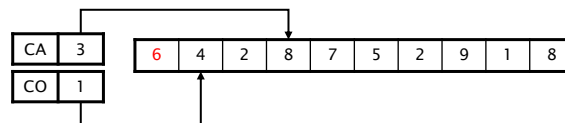
```
apilar(cola, 1);  
apilar(cola, 8);  
→ apilar(cola, 6);  
apilar(cola, 5);  
apilar(cola, 1);
```



# Cola: versión estática

- ¿Y si llegamos al final del array?

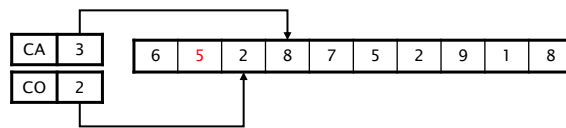
```
apilar(cola, 1);  
apilar(cola, 8);  
apilar(cola, 6);  
→ apilar(cola, 5);  
apilar(cola, 1);
```



# Cola: versión estática

- ¿Y si llegamos al final del array?

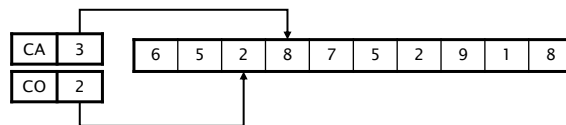
```
apilar(cola,1);  
apilar(cola,8);  
apilar(cola,6);  
apilar(cola,5);  
→ apilar(cola,1);
```



# Cola: versión estática

- ¿Y si llegamos al final del array?

```
void apilar(tipoCola *c, int i){  
    if (suma_uno(c->cola) == c->cabeza){ ←  
        printf("Error: cola llena \n");  
    }else{  
        c->valores[c->cola] = i;  
        c->cola = suma_uno(c->cola);  
    }  
}
```

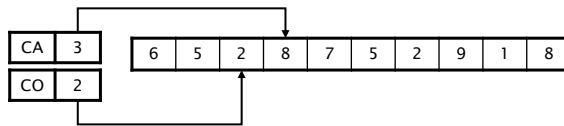




# Cola: versión estática

- ¿Y si llegamos al final del array?

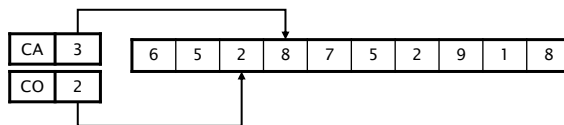
```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza) { ← suma_uno(2)
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```



# Cola: versión estática

- ¿Y si llegamos al final del array?

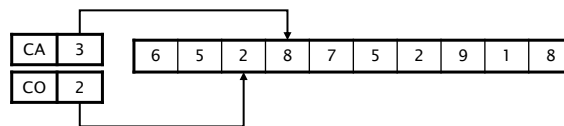
```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza) { ← (2+1)%10
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```



# Cola: versión estática

- ¿Y si llegamos al final del array?

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){ ← ¿3==3?
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

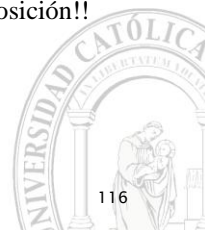
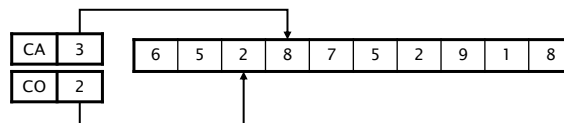


# Cola: versión estática

- ¿Y si llegamos al final del array?

```
void apilar(tipoCola *c, int i){
    if (suma_uno(c->cola) == c->cabeza){
        printf("Error: cola llena \n");
    }else{
        c->valores[c->cola] = i;
        c->cola = suma_uno(c->cola);
    }
}
```

¡¡Perdemos una posición!!



## Cola: versión dinámica

---

- Se implementa mediante una estructura autorreferenciada con encadenamiento simple.
- Permite disponer de toda la memoria.
- Los datos se almacenan en un campo.



## Cola: versión dinámica

---

- Tendremos dos registros
  - *struct nCola*: almacena los datos y un puntero al siguiente elemento.
  - *struct tCola*: almacena dos punteros al registro anterior.



## Cola: versión dinámica

---

- Tendremos dos registros
  - *struct nCola*: almacena los datos y un puntero al siguiente elemento.

```
typedef struct nCola{  
    int valor;  
    struct nCola *sig;  
}NODO_COLA, *P_NODO_COLA;
```

- *struct tCola*: almacena dos punteros al registro anterior.



## Cola: versión dinámica

---

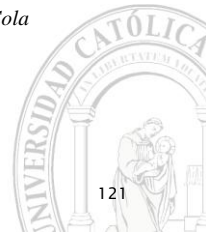
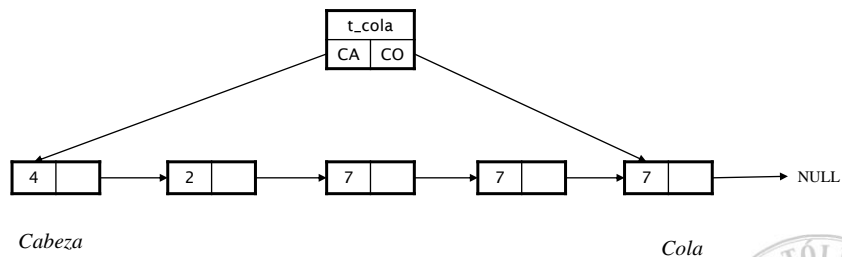
- Tendremos dos registros
  - *struct nCola*: almacena los datos y un puntero al siguiente elemento.
  - *struct tCola*: almacena dos punteros al registro anterior.

```
typedef struct tCola{  
    P_NODO_COLA cabeza;  
    P_NODO_COLA cola;  
}tipoCola;
```



# Cola: versión dinámica

---



# Cola: versión dinámica

---

- Función push (introducir)

```
tipoCola* iniciar(tipoCola *q){
    q->cabeza = NULL;
    q->cola = NULL;
    return q;
}
```

```
tipoCola cola, *pto_est_cola;    // Puntero a estructura de cola
pto_est_cola = &cola;
pto_est_cola = iniciar(pto_est_cola); ←
```

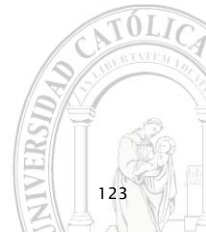


# Cola: versión dinámica

- Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}
```

```
push(pto_est_cola ,7); ←
push(pto_est_cola ,2);
push(pto_est_cola ,4);
```

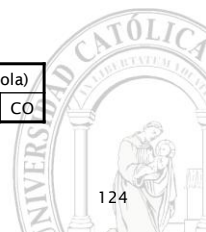
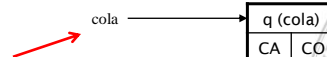


# Cola: versión dinámica

- Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}
```

```
push(pto_est_cola ,7); ←
push(pto_est_cola ,2);
push(pto_est_cola ,4);
```

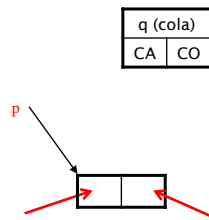


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}
```

```
push(pto_estCola ,7);
push(pto_estCola ,2);
push(pto_estCola ,4);
```

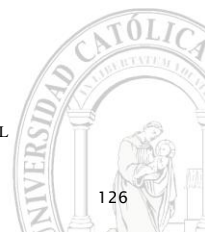
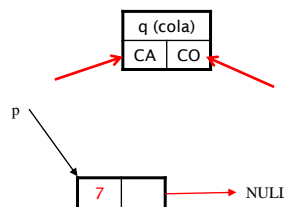


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}
```

```
push(pto_estCola ,7);
push(pto_estCola ,2);
push(pto_estCola ,4);
```

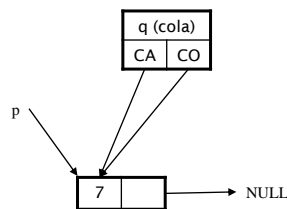


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}
```

```
push(pto_estCola ,7);
push(pto_estCola ,2); ←
push(pto_estCola ,4);
```

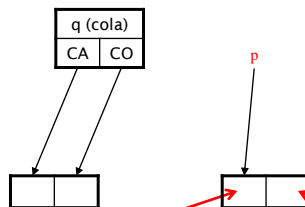


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL; ←
    p->valor = i; ←
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}
```

```
push(pto_estCola ,7);
push(pto_estCola ,2);
push(pto_estCola ,4);
```



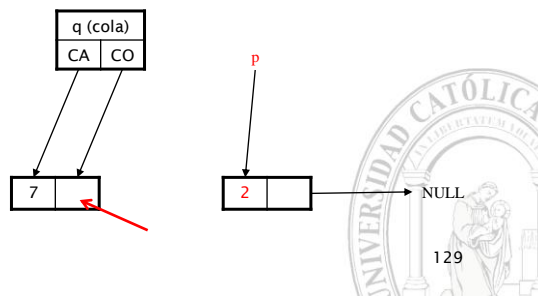


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}

push(pto_estCola ,7);
push(pto_estCola ,2);
push(pto_estCola ,4);
```

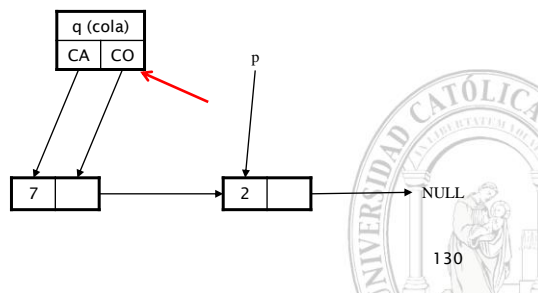


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}

push(pto_estCola ,7);
push(pto_estCola ,2);
push(pto_estCola ,4);
```

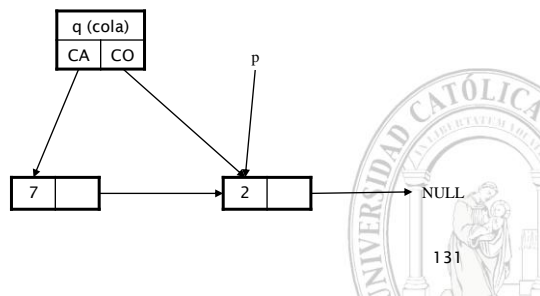


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO Cola p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}

push(pto_est_cola ,7);
push(pto_est_cola ,2);
push(pto_est_cola ,4);
```

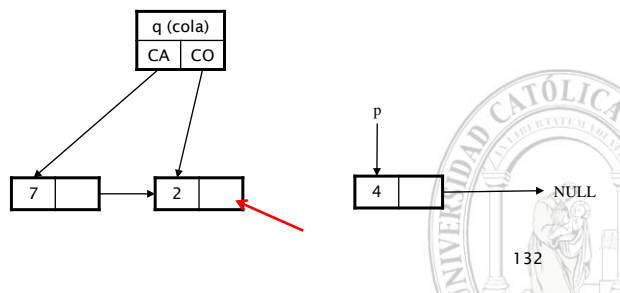


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO Cola p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}

push(pto_est_cola ,7);
push(pto_est_cola ,2);
push(pto_est_cola ,4);
```

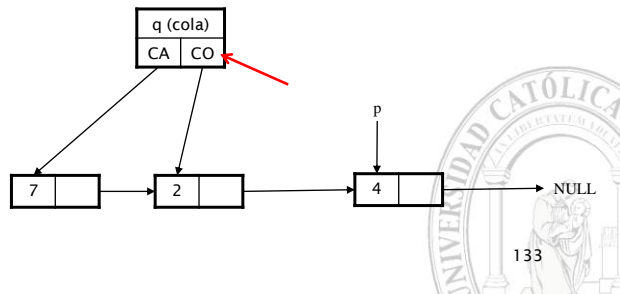


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}

push(pto_est_cola ,7);
push(pto_est_cola ,2);
push(pto_est_cola ,4);
```

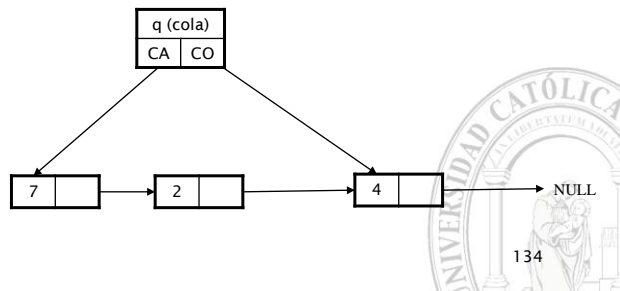


# Cola: versión dinámica

## • Función push (introducir)

```
void push(tipoCola *q, int i){
    P_NODO_COLA p; p = alojar_nodoCola();
    p->sig = NULL;
    p->valor = i;
    if(es_vacia(q)) q->cola = q->cabeza = p;
    else{
        q->cola->sig= p;
        q->cola = p;
    }
}

push(pto_est_cola ,7);
push(pto_est_cola ,2);
push(pto_est_cola ,4);
```

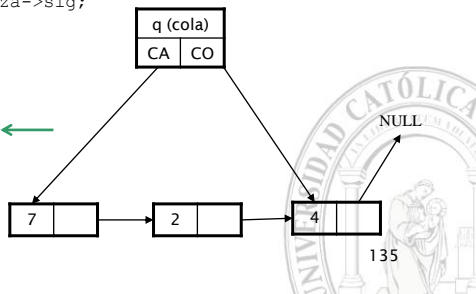


# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```

printf("Valor:%d\n", pop(pto\_est\_cola); ←  
printf("Valor:%d\n", pop(pto\_est\_cola);  
printf("Valor:%d\n", pop(pto\_est\_cola);

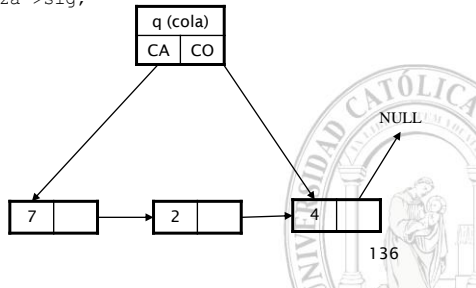
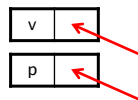


# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```

printf("Valor:%d\n", pop(pto\_est\_cola);  
printf("Valor:%d\n", pop(pto\_est\_cola);  
printf("Valor:%d\n", pop(pto\_est\_cola);



# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig; ←
        free(p);
    }
    return v;
}
```

printf("Valor:%d\n", pop(pto\_est\_cola);  
 printf("Valor:%d\n", pop(pto\_est\_cola);  
 printf("Valor:%d\n", pop(pto\_est\_cola);

# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig; ←
        free(p);
    }
    return v;
}
```

printf("Valor:%d\n", pop(pto\_est\_cola);  
 printf("Valor:%d\n", pop(pto\_est\_cola);  
 printf("Valor:%d\n", pop(pto\_est\_cola);

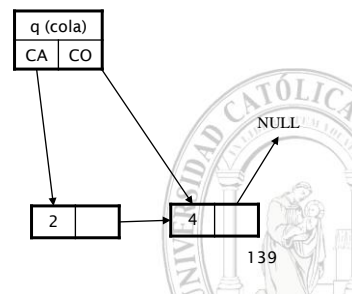
# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```



```
printf("Valor:%d\n", pop(pto_estCola));
printf("Valor:%d\n", pop(pto_estCola));
printf("Valor:%d\n", pop(pto_estCola));
```

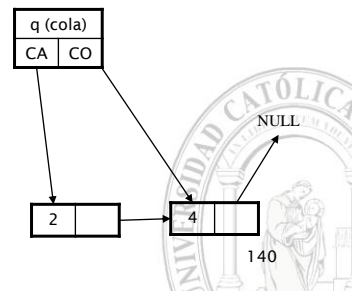


# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```

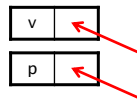
```
printf("Valor:%d\n", pop(pto_estCola));
printf("Valor:%d\n", pop(pto_estCola));
printf("Valor:%d\n", pop(pto_estCola));
```



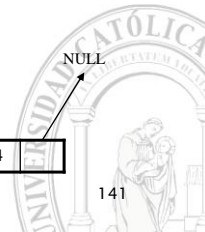
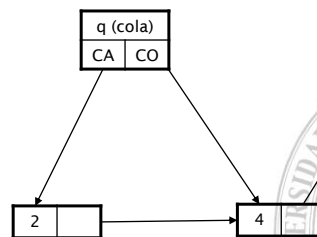
# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```



```
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
```



141

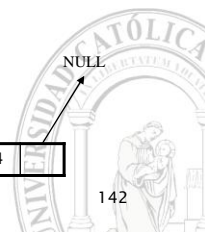
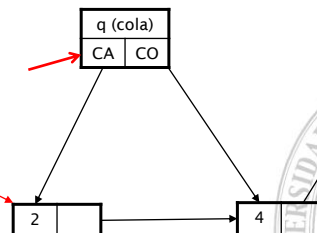
# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```



```
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
```



142

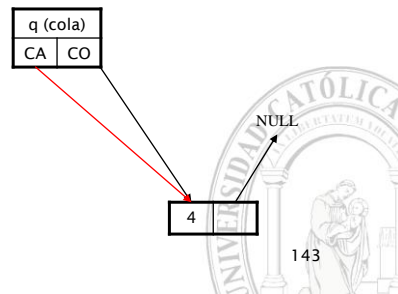
# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```

v	2
p	

```
printf("Valor:%d\n", pop(pto_estCola));
printf("Valor:%d\n", pop(pto_estCola));
printf("Valor:%d\n", pop(pto_estCola));
```

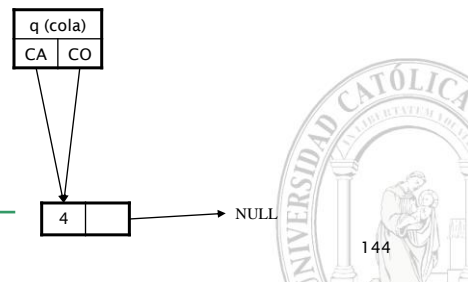


# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```

```
printf("Valor:%d\n", pop(pto_estCola));
printf("Valor:%d\n", pop(pto_estCola));
printf("Valor:%d\n", pop(pto_estCola));
```



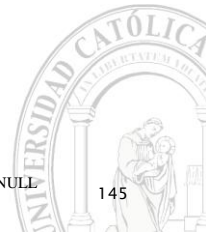
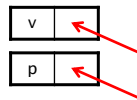


# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```

```
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
```

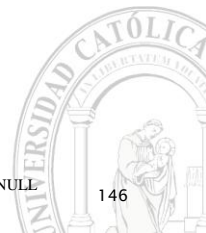


# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}
```

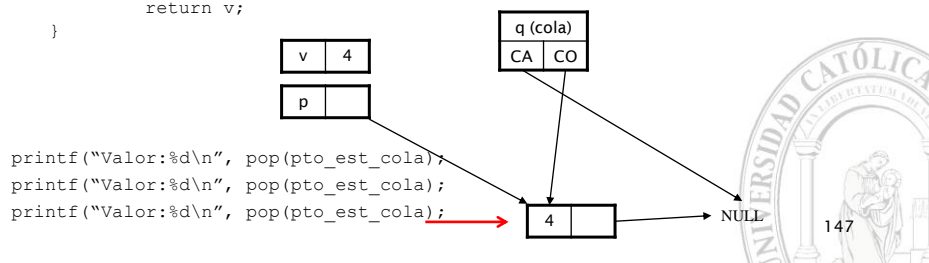
```
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
```



# Cola: versión dinámica

## • Función pop (extraer)

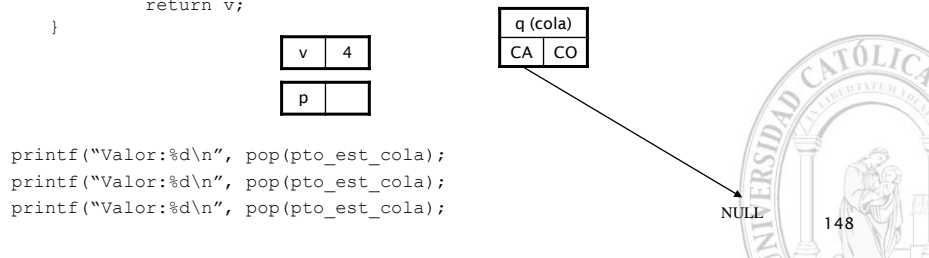
```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p); ←
    }
    return v;
}
```



# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p); ←
    }
    return v;
}
```

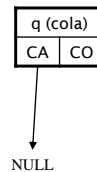


# Cola: versión dinámica

## • Función pop (extraer)

```
int pop(tipoCola *q){
    P_NODO_COLA p;
    int v;
    if(es_vacia(q)){
        v = -1; printf("Error: cola vacia\n");
    }else{
        v = q->cabeza->valor;
        p = q->cabeza;
        q->cabeza = q->cabeza->sig;
        free(p);
    }
    return v;
}

printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
printf("Valor:%d\n", pop(pto_est_cola);
```

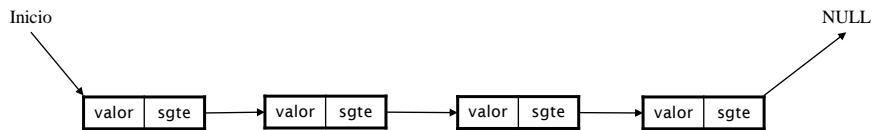


## Lista

- Estructura lineal más general que se puede definir.
- Cada elemento tiene un único antecesor y sucesor.
- Operaciones
  - CREA: Crea una cola vacía.
  - VACIA: Devuelve un valor cierto si la cola está vacía, y falso en caso contrario.
  - BUSCA: Devuelve el elemento de una posición.
  - INSERTA: Añade un elemento en la posición indicada.
  - SUPRIME: Suprime el elemento en la posición indicada.



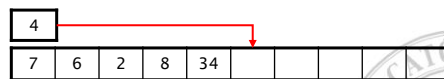
# Lista



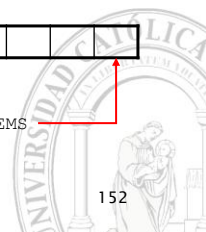
## Lista: versión estática

- Registro que contiene
  - Array de datos
  - Un entero que indica la siguiente posición libre.

```
#define MAX_ELEMS 10
typedef struct lista{
    int valores[MAX_ELEMS];
    int p_libre;
}tipo_lista;
```



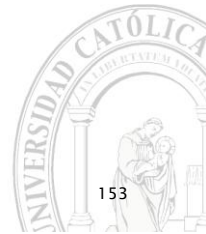
MAX\_ELEMS



# Lista: versión estática

- Función insertar

```
void insertar(tipo_lista *l, int v, int p){ // p es la posición a extraer
    int i; // entre 0 y p->libre -1
    if (p > l->p_libre || p < 0){
        printf("Error: posición fuera de lista\n");
        return;
    }
    if (l->p_libre == MAX_ELEMS){
        printf("Error: lista llena\n");
        return;
    }
    for(i=l->p_libre; i > p ; i--){
        l->valores[i] = l->valores[i-1];
    }
    l->valores[p]=v;
    ++l->p_libre;
}
```

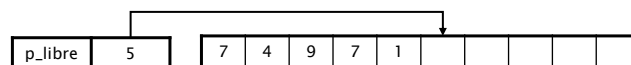


# Lista: versión estática

- De manera visual

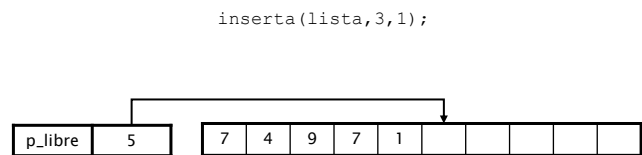
Inserta en *lista* un 3  
en la posición 1

```
inserta(lista,3,1);
```



# Lista: versión estática

- De manera visual

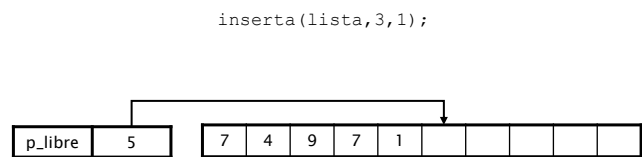


```
void insertar(tipo_lista *l, int v, int p){  
    (...)  
    if (p > l->p_libre || p < 0){  
        (...)  
    }  
}
```



# Lista: versión estática

- De manera visual



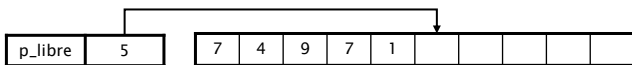
```
void insertar(tipo_lista *l, int v, int p){  
    (...)  
    if (l->p_libre == MAX_ELEMS){  
        (...)  
    }  
}
```



# Lista: versión estática

- De manera visual

```
inserta(lista,3,1);
```



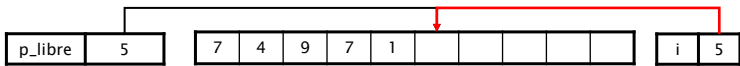
```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>p ;i--){
        l->valores[i] = l->valores[i-1];
    }
    (...)
}
```



# Lista: versión estática

- De manera visual

```
inserta(lista,3,1);
```



```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ;i--){
        l->valores[i] = l->valores[i-1];
    }
    (...)
}
```



# Lista: versión estática

- De manera visual

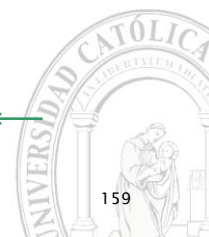
`inserta(lista,3,1);`

p_libre	5
---------	---

7	4	9	7	1	1				
---	---	---	---	---	---	--	--	--	--

i	5
---	---

```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ; i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



159

# Lista: versión estática

- De manera visual

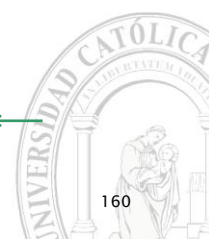
`inserta(lista,3,1);`

p_libre	5
---------	---

7	4	9	7	1	1				
---	---	---	---	---	---	--	--	--	--

i	4
---	---

```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ; i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



160



# Lista: versión estática

- De manera visual


`inserta(lista,3,1);`

p_libre	5
---------	---

7	4	9	7	1	1				
---	---	---	---	---	---	--	--	--	--

i	4
---	---

```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ;i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



161

# Lista: versión estática

- De manera visual


`inserta(lista,3,1);`

p_libre	5
---------	---

7	4	9	7	7	1				
---	---	---	---	---	---	--	--	--	--

i	4
---	---

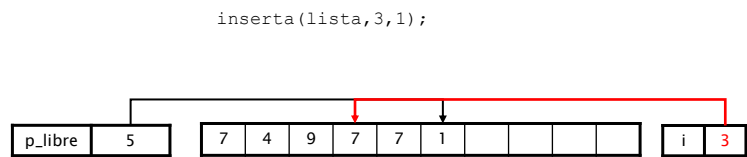
```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ;i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



162

# Lista: versión estática

- De manera visual

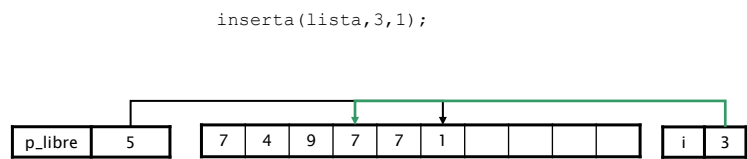


```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p; i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



# Lista: versión estática

- De manera visual



```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ;i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



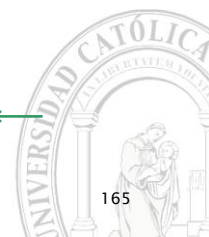
# Lista: versión estática

- De manera visual

`inserta(lista,3,1);`

p_libre	5	7	4	9	9	7	1					i	3
---------	---	---	---	---	---	---	---	--	--	--	--	---	---

```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ; i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



165

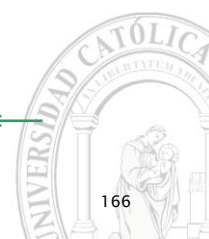
# Lista: versión estática

- De manera visual

`inserta(lista,3,1);`

p_libre	5	7	4	9	9	7	1					i	2
---------	---	---	---	---	---	---	---	--	--	--	--	---	---

```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ; i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



166

# Lista: versión estática

- De manera visual


`inserta(lista,3,1);`

p_libre	5
---------	---

7	4	9	9	7	1				
---	---	---	---	---	---	--	--	--	--

i	2
---	---

```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ;i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



167

# Lista: versión estática

- De manera visual


`inserta(lista,3,1);`

p_libre	5
---------	---

7	4	4	9	7	1				
---	---	---	---	---	---	--	--	--	--

i	2
---	---

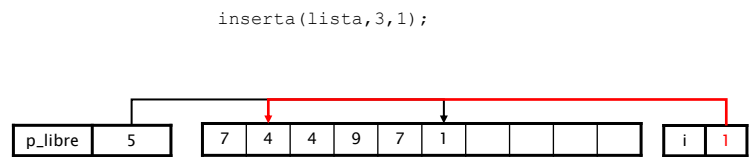
```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p ;i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



168

# Lista: versión estática

- De manera visual

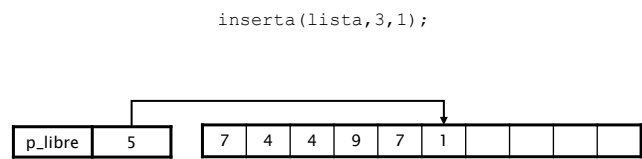


```
void insertar(tipo_lista *l, int v, int p){
    (...)
    for(i=l->p_libre; i>=p; i--)
        l->valores[i] = l->valores[i-1];
    (...)
}
```



# Lista: versión estática

- De manera visual



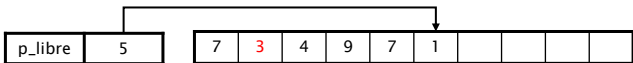
```
void insertar(tipo_lista *l, int v, int p){
    (...)
    l->valores[p-1]=v;
    ++l->p_libre;
    (...)
}
```



# Lista: versión estática

- De manera visual

```
inserta(lista,3,1);
```



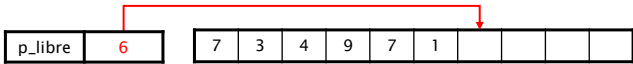
```
void insertar(tipo_lista *l, int v, int p){
    (...)
    l->valores[p-1]=v;
    ++l->p_libre;
    (...)
}
```



# Lista: versión estática

- De manera visual

```
inserta(lista,3,1);
```



```
void insertar(tipo_lista *l, int v, int p){
    (...)
    l->valores[p-1]=v;
    ++l->p_libre;
    (...)
}
```



# Lista: versión estática

- Función extraer

```
int extraer(tipo_lista *l, int p){
    int i,v;
    if (p >= l->p_libre || p < 0){
        printf("Error: posición fuera de lista\n");
        return -1;
    }
    v = l->valores[p];
    for(i=p; i<l->p_libre; i++){
        l->valores[i] = l->valores[i+1];
    }
    --l->p_libre;
    return v;
}
```

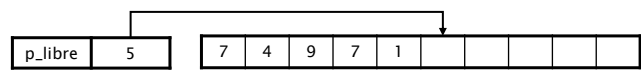


# Lista: versión estática

- De manera visual

Extrae de *lista* el elemento de la posición 2

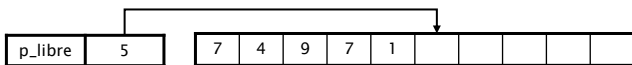
```
extraer(lista,2);
```



# Lista: versión estática

- De manera visual

extraer(lista,2);



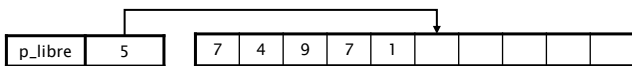
```
int extraer(tipo_lista *l, int p){
    (...)
    v = l->valores[p-1];
    (...)
}
```



# Lista: versión estática

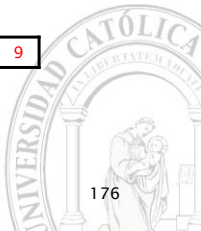
- De manera visual

extraer(lista,2);



```
int extraer(tipo_lista *l, int p){
    (...)
    v = l->valores[p-1];
    (...)
}
```

v 9

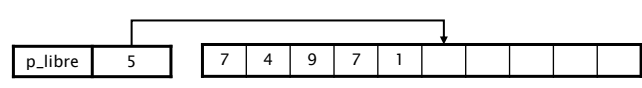




# Lista: versión estática

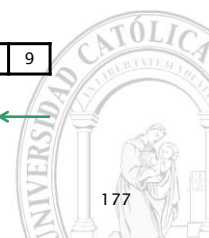
- De manera visual

extraer(lista,2);



```
int extraer(tipo_lista *l, int p){
    (...)
    for(i=p; i<l->p_libre-1; i++)
        l->valores[i] = l->valores[i+1];
    (...)
}
```

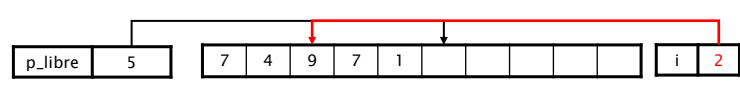
v 9



# Lista: versión estática

- De manera visual

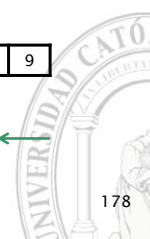
extraer(lista,2);



```
int extraer(tipo_lista *l, int p){
    (...)
    for(i=p-1; i<l->p_libre-1; i++)
        l->valores[i] = l->valores[i+1];
    (...)
}
```

v 9

i 2



# Lista: versión estática

- De manera visual

extraer(lista,2);

```
int extraer(tipo_lista *l, int p){
    (...)
    for(i=p-1; i<l->p_libre-1; i++)
        l->valores[i] = l->valores[i+1];
    (...)
}
```

v 9

179

# Lista: versión estática

- De manera visual

extraer(lista,2);

```
int extraer(tipo_lista *l, int p){
    (...)
    for(i=p-1; i<l->p_libre-1; i++)
        l->valores[i] = l->valores[i+1];
    (...)
}
```

v 9

180

# Lista: versión estática

- De manera visual

`extraer(lista,2);`

```
int extraer(tipo_lista *l, int p){
    (...)
    for(i=p-1; i<l->p_libre-1; i++)
        l->valores[i] = l->valores[i+1];
    (...)
}
```

UNIVERSIDAD CATOLICA

181

# Lista: versión estática

- De manera visual

`extraer(lista,2);`

```
int extraer(tipo_lista *l, int p){
    (...)
    for(i=p-1; i<l->p_libre-1; i++)
        l->valores[i] = l->valores[i+1];
    (...)
}
```

UNIVERSIDAD CATOLICA

182

# Lista: versión estática

- De manera visual

extraer(lista,2);

```
int extraer(tipo_lista *l, int p){
    (...)
    for(i=p-1; i<l->p_libre-1; i++)
        l->valores[i] = l->valores[i+1];
    (...)
}
```

v 9

183

# Lista: versión estática

- De manera visual

extraer(lista,2);

```
int extraer(tipo_lista *l, int p){
    (...)
    --l->p_libre;
    (...)
}
```

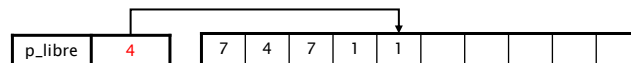
v 9

184

# Lista: versión estática

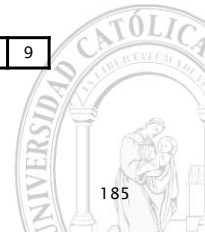
- De manera visual

```
extraer(lista,2);
```



```
int extraer(tipo_lista *l, int p){  
    (...)  
    --l->p_libre;  
    (...)  
}
```

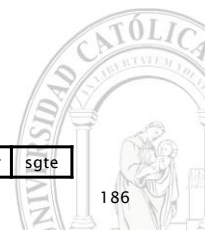
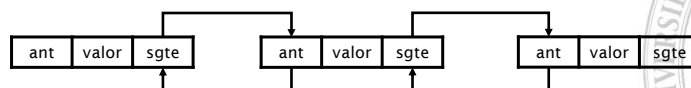
v 9



# Lista: versión dinámica

- Registro que contiene
  - Dos punteros que hacen referencia a la misma estructura
    - *anterior y siguiente*
  - El dato a guardar.

```
typedef struct t_lista{  
    int valor;  
    struct t_lista *anterior;  
    struct t_lista *siguiente;  
}NODO_LISTA,*P_NODO_LISTA;
```



# Lista: versión dinámica

## • Función esVacia()

```
int esVacia(P_NODO_LISTA l) {
    if (l == NULL) return true;
    else return false;
}
```



# Lista: versión dinámica

## • Función insertar

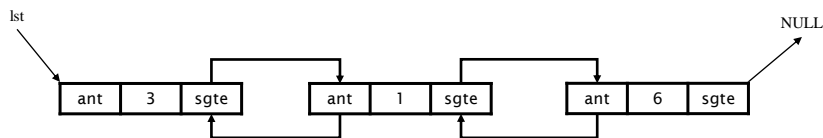
```
P_NODO_LISTA insertar (P_NODO_LISTA lst, int valor, int pos){ /Ahora pos entre 1 y el
último insertado
    P_NODO_LISTA laux, lsgte;
    int i;
    if (pos < 1) { printf("ERROR, fuera de posicion\n"); return lst;}
    if ((pos == 1) || esVacia(lst)){
        laux = (P_NODO_LISTA) malloc (sizeof(NODO_LISTA));
        laux->valor = valor;
        laux->anterior = NULL;
        laux->siguiente = lst;
        if(!esVacia(lst)) lst->anterior = laux;
        return laux;
    }
    for (i = 1, laux = lst; (i<pos-1)&&(laux != NULL);i++)
        laux = laux->siguiente;
    if (laux != NULL){
        lsgte = laux -> siguiente;
        laux->siguiente = (P_NODO_LISTA)malloc(sizeof(NODO_LISTA));
        laux->siguiente->valor = valor;
        laux->siguiente->siguiente = lsgte;
        laux->siguiente->anterior = laux;
        if (lsgte != NULL)
            lsgte->anterior = laux->siguiente;
    }
    else printf("ERROR, fuera de posicion\n");
    return lst;
}
```



# Lista: versión dinámica

- Función insertar

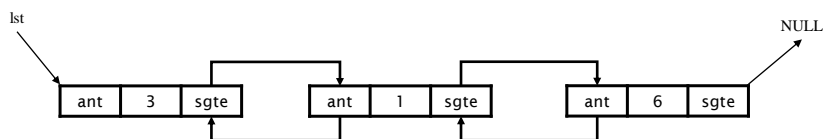
```
insertar(1st,8,3);
```



# Lista: versión dinámica

- Función insertar

```
insertar(1st,8,3);
```



```
for (i = 1, laux = 1st; (i < pos-1) && (laux != NULL); i++)  
    laux = laux->sgte;
```



# Lista: versión dinámica

- Función insertar

```
insertar(1st,8,3);
```

i	1
---	---

```
for (i = 1, laux = 1st; (i<pos-1) && (laux != NULL); i++)  
    laux = laux->siguiente;
```

191

# Lista: versión dinámica

- Función insertar

```
insertar(1st,8,3);
```

i	1
---	---

```
for (i = 1, laux = 1st; (i<pos-1) && (laux != NULL); i++)  
    laux = laux->siguiente;
```

192



# Lista: versión dinámica

- Función insertar

```
insertar(1st,8,3);
```

1st

ant

3

sgte

ant

1

sgte

ant

6

sgte

NULL

laux

i

1

```
for (i = 1, laux = 1st; (i<pos-1)&&(laux != NULL); i++)  
    laux = laux->siguiente;
```



193

# Lista: versión dinámica

- Función insertar

```
insertar(1st,8,3);
```

1st

ant

3

sgte

ant

1

sgte

ant

6

sgte

NULL

laux

i

2

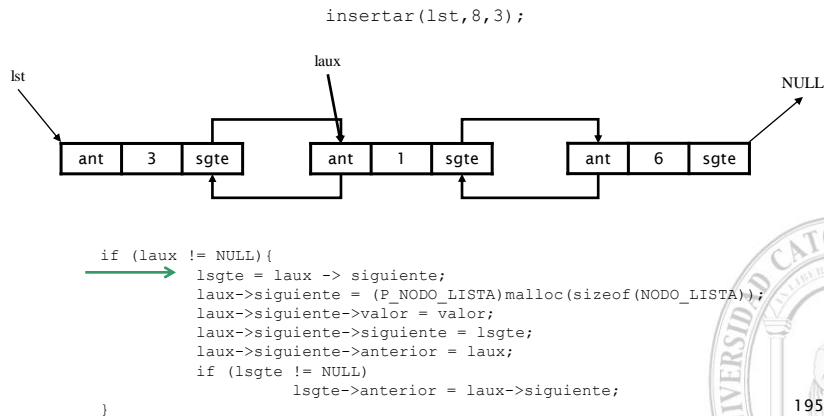
```
for (i = 1, laux = 1st; ((i<pos-1)&&(laux != NULL)); i++)  
    laux = laux->siguiente;
```



194

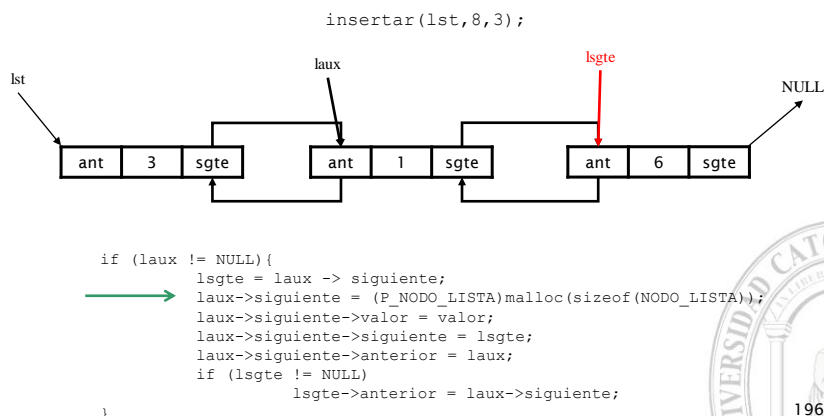
# Lista: versión dinámica

- Función insertar



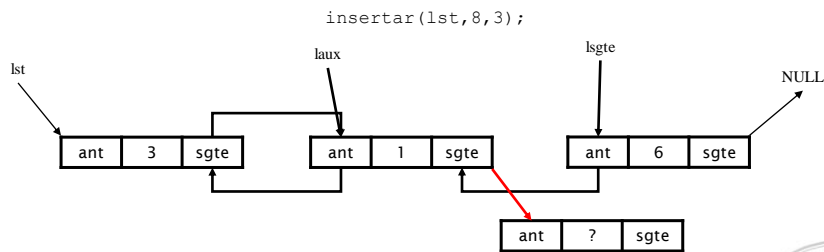
# Lista: versión dinámica

- Función insertar

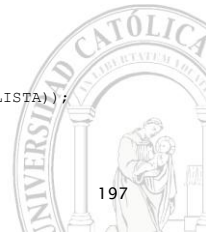


# Lista: versión dinámica

- Función insertar

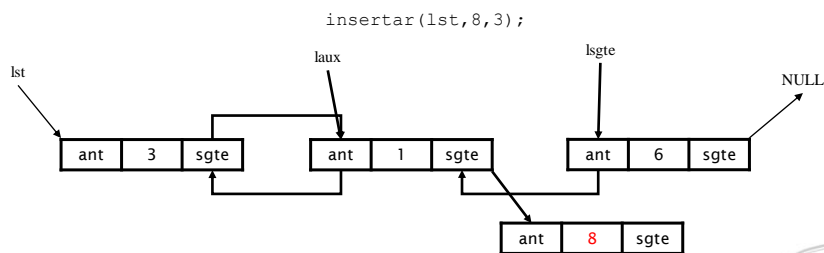


```
if (laux != NULL){  
    lsgte = laux -> siguiente;  
    laux->siguiente = (P_NODO_LISTA)malloc(sizeof(NODO_LISTA));  
    laux->siguiente->valor = valor;  
    laux->siguiente->siguiente = lsgte;  
    laux->siguiente->anterior = laux;  
    if (lsgte != NULL)  
        lsgte->anterior = laux->siguiente;  
}
```



# Lista: versión dinámica

- Función insertar

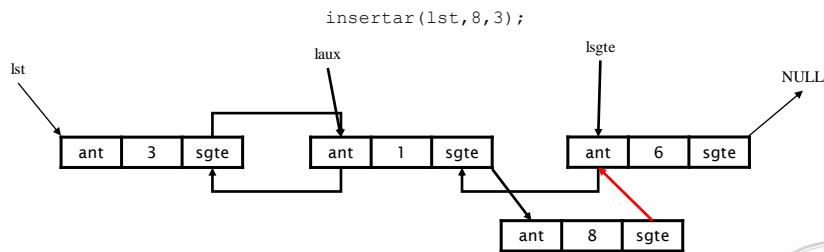


```
if (laux != NULL){  
    lsgte = laux -> siguiente;  
    laux->siguiente = (P_NODO_LISTA)malloc(sizeof(NODO_LISTA));  
    laux->siguiente->valor = valor;  
    laux->siguiente->siguiente = lsgte;  
    laux->siguiente->anterior = laux;  
    if (lsgte != NULL)  
        lsgte->anterior = laux->siguiente;  
}
```



# Lista: versión dinámica

- Función insertar

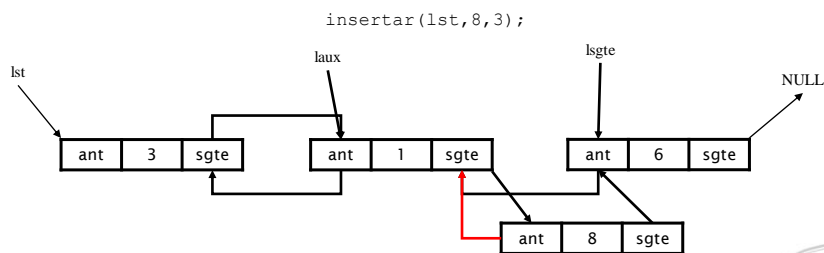


```
if (laux != NULL){  
    lsgte = laux -> siguiente;  
    laux->siguiente = (P_NODO_LISTA)malloc(sizeof(NODO_LISTA));  
    laux->siguiente->valor = valor;  
    laux->siguiente->siguiente = lsgte;  
    laux->siguiente->anterior = laux;  
    if (lsgte != NULL)  
        lsgte->anterior = laux->siguiente;  
}
```

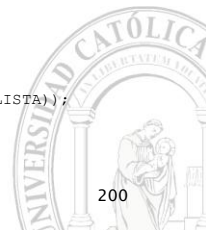


# Lista: versión dinámica

- Función insertar

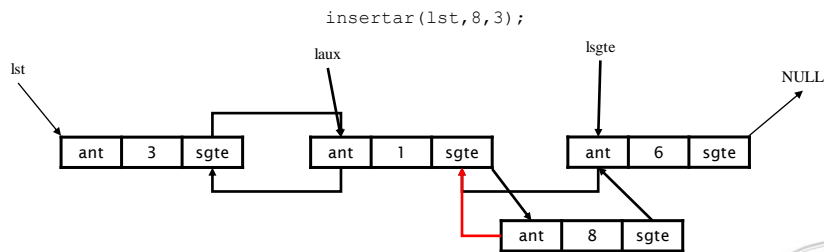


```
if (laux != NULL){  
    lsgte = laux -> siguiente;  
    laux->siguiente = (P_NODO_LISTA)malloc(sizeof(NODO_LISTA));  
    laux->siguiente->valor = valor;  
    laux->siguiente->siguiente = lsgte;  
    laux->siguiente->anterior = laux;  
    if (lsgte != NULL)  
        lsgte->anterior = laux->siguiente;  
}
```



# Lista: versión dinámica

- Función insertar

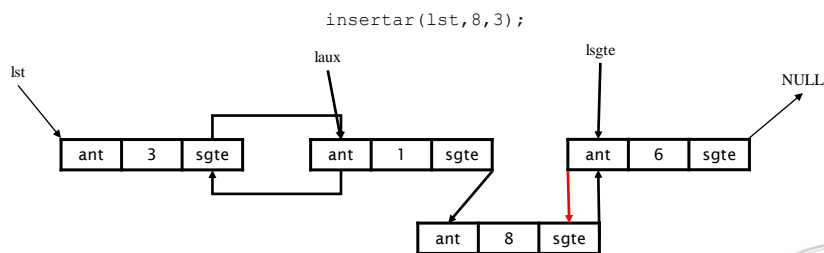


```
if (laux != NULL){
    lsgte = laux -> siguiente;
    laux->siguiente = (P_NODO_LISTA)malloc(sizeof(NODO_LISTA));
    laux->siguiente->valor = valor;
    laux->siguiente->siguiente = lsgte;
    laux->siguiente->anterior = laux;
    if (lsgte != NULL)
        lsgte->anterior = laux->siguiente;
}
```



# Lista: versión dinámica

- Función insertar



```
if (laux != NULL){
    lsgte = laux -> siguiente;
    laux->siguiente = (P_NODO_LISTA)malloc(sizeof(NODO_LISTA));
    laux->siguiente->valor = valor;
    laux->siguiente->siguiente = lsgte;
    laux->siguiente->anterior = laux;
    if (lsgte != NULL)
        lsgte->anterior = laux->siguiente;
}
```



# Lista: versión dinámica

## • Función borrar (parte I)

```
P_NODO_LISTA borrar (P_NODO_LISTA lst, int pos, int *valor){
    P_NODO_LISTA laux;
    int i;

    //Si la posición está fuera de rango o la lista es vacía no
    //se hace nada. Devolvemos la lista tal cual.
    if ((pos < 1) || (esVacia(lst))) {
        *valor = -1;
        return lst;
    }
    //Si es la primera posición, eliminamos el elemento y ponemos
    //como cabeza de la lista el segundo elemento.
    if (pos == 1){
        laux = lst->siguiente;
        *valor = lst->valor;
        free(lst);
        //En el caso de que no fuera el último elemento se pone
        //el puntero a anterior apuntando a NULL
        if (laux != NULL)
            laux->anterior = NULL;
        return laux;
    }
}
```



203

# Lista: versión dinámica

## • Función borrar (parte II)

```
P_NODO_LISTA borrar (P_NODO_LISTA lst, int pos, int *valor){

    // En otro caso, lo primero que se debe hacer es encontrar la posición a borrar.
    for (i = 1, laux = lst; (i < pos) && (laux->siguiente != NULL); i++)
        laux = laux->siguiente;

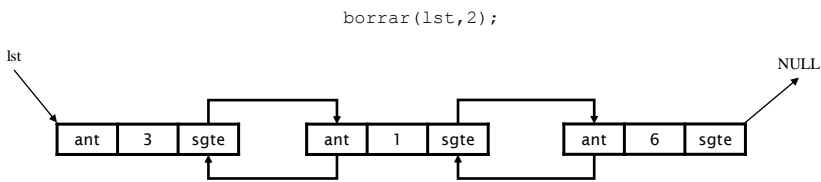
    //Si no hemos llegado al último nodo, eliminamos y
    //actualizamos las referencias del nodo anterior y posterior del eliminado
    if (laux->siguiente != NULL){
        laux->anterior->siguiente = laux->siguiente;
        laux->siguiente->anterior = laux->anterior;
        *valor = laux->valor;
        free(laux);
    } else if (pos == i){
        //Si hemos llegado al final de lista y ese es el
        //elemento a eliminar sólo tenemos que actualizar el puntero a siguiente.
        laux->anterior->siguiente = laux->siguiente;
        *valor = laux->valor;
        free(laux);
    }
    else {
        *valor = -1;
        printf("ERROR, fuera de posición\n");
    }
    return lst;
}
```



204

# Lista: versión dinámica

- Función borrar

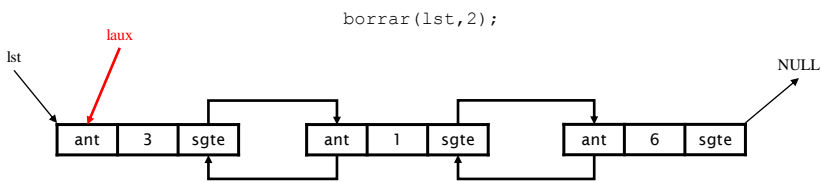


→ for (i = 1, laux=1st; (i<pos) && (laux->siguiente != NULL); i++)  
laux = laux->siguiente;

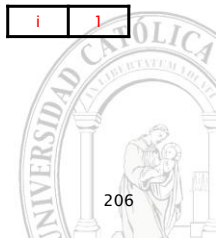


# Lista: versión dinámica

- Función borrar

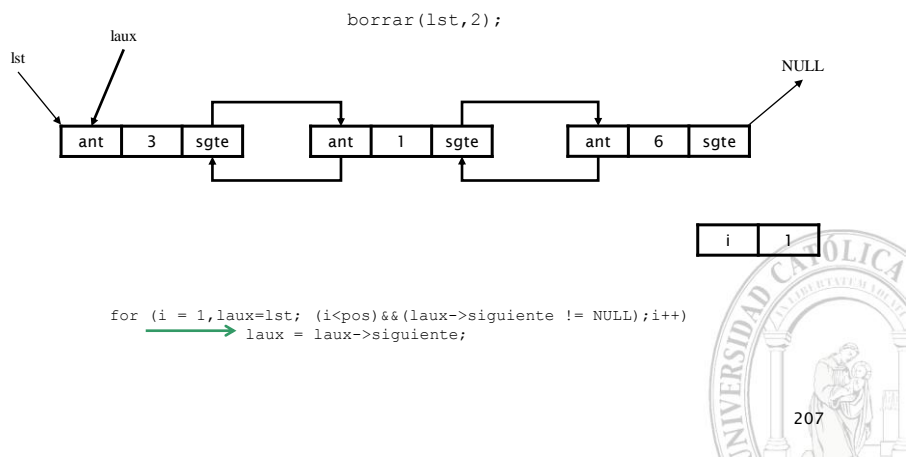


→ for (i = 1, laux=1st; (i<pos) && (laux->siguiente != NULL); i++)  
laux = laux->siguiente;



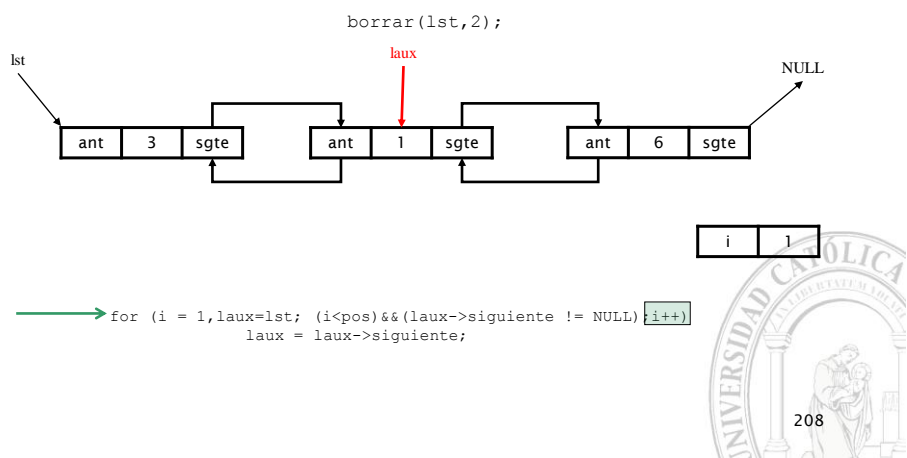
# Lista: versión dinámica

- Función borrar



# Lista: versión dinámica

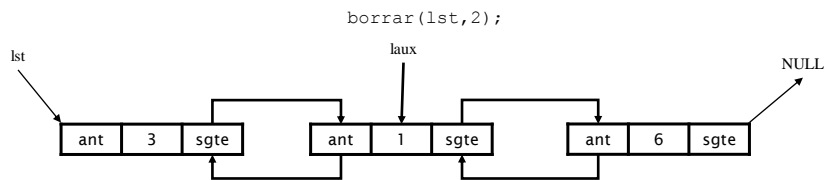
- Función borrar



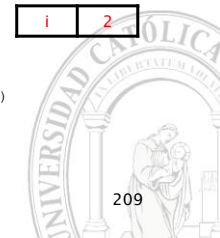


# Lista: versión dinámica

- Función borrar

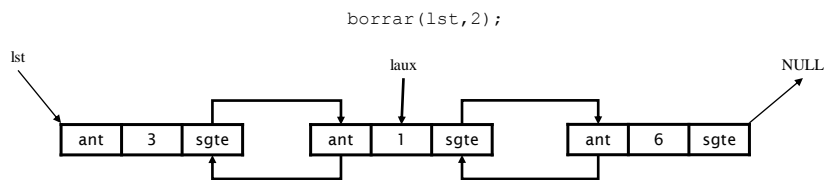


→ for (i = 1, laux=lst; (i<pos) && (laux->siguiente != NULL); i++)  
laux = laux->siguiente;

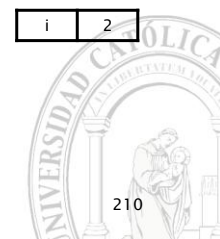


# Lista: versión dinámica

- Función borrar

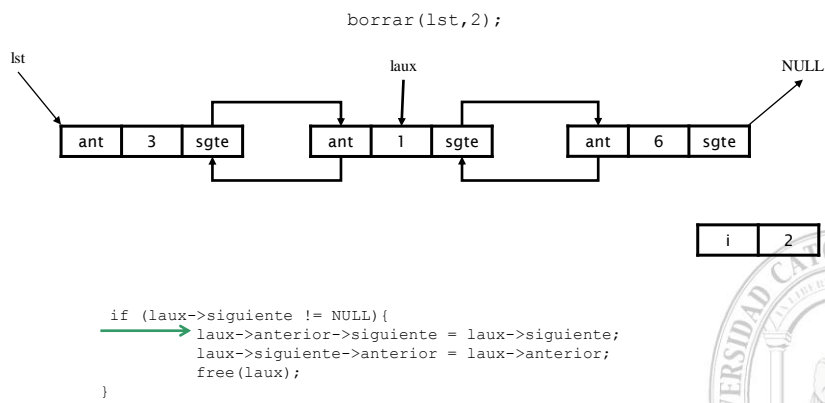


→ if (laux->siguiente != NULL){  
laux->anterior->siguiente = laux->siguiente;  
laux->siguiente->anterior = laux->anterior;  
free(laux);  
}



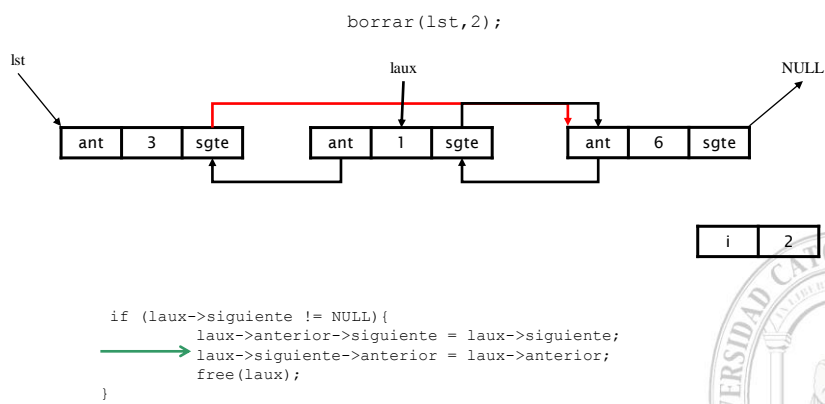
# Lista: versión dinámica

- Función insertar



# Lista: versión dinámica

- Función insertar



# Lista: versión dinámica

- Función borrar

borrar(lst,2);

```
if (laux->siguiente != NULL){  
    laux->anterior->siguiente = laux->siguiente;  
    laux->siguiente->anterior = laux->anterior;  
    free(laux);  
}
```

i 2

UNIVERSIDAD CATOLICA

213

# Lista: versión dinámica

- Función borrar

borrar(lst,2);

```
if (laux->siguiente != NULL){  
    laux->anterior->siguiente = laux->siguiente;  
    laux->siguiente->anterior = laux->anterior;  
    free(laux);  
}
```

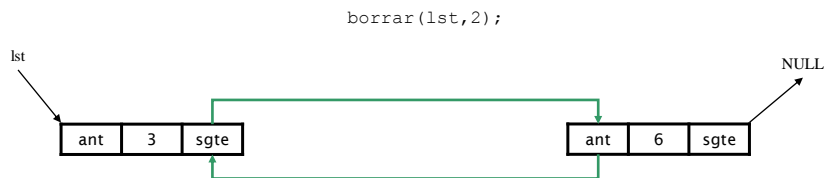
i 2

UNIVERSIDAD CATOLICA

214

# Lista: versión dinámica

- Función borrar



```
if (laux->siguiente != NULL){
    laux->anterior->siguiente = laux->siguiente;
    laux->siguiente->anterior = laux->anterior;
    free(laux);
}
```



# Lista: versión dinámica

- Función buscar

- Recorre la estructura hasta que encuentra el nodo con el valor o llega al final

```
P_NODO_LISTA buscar (P_NODO_LISTA lst, int valor){
    P_NODO_LISTA laux;
    for (laux = lst; laux != NULL; laux = laux->siguiente;)
        if (laux->valor == valor) break;
    return laux;
}
```



# Índice

---

- Introducción
- Estructuras lineales
- Estructuras no lineales
- Estructuras en memoria secundaria



## Estructuras no lineales

---

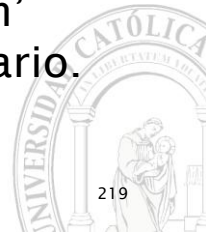
- Los nodos no tienen una relación uno a uno.
- Cada nodo puede tener de 0 a N sucesores y antecesores.
- Estudiaremos dos tipos:
  - Árboles
  - Grafos



# Árboles

---

- Cada nodo tiene
  - Un único predecesor
  - Dos o más sucesores
- Excepto el nodo raíz que no tiene predecesor.
- Un árbol cuyos nodos tienen 'n' elementos se le llama árbol n-ario.
  - Con dos sucesores: *binario*.



# Árboles

---

- Un árbol es siempre acíclico.
  - Un nodo no puede ser sucesor-t/predecesor-t de sí mismo.
  - No se produce cierre transitivo.
- Nosotros trabajaremos con árboles binarios:

```
arbol_binario ::= arbol_nulo | nodo
nodo          ::= dato + hijo_derecho + hijo_izquierdo
hijo_derecho  ::= arbol_binario
hijo_izquierdo ::= arbol_binario
```



# Árboles

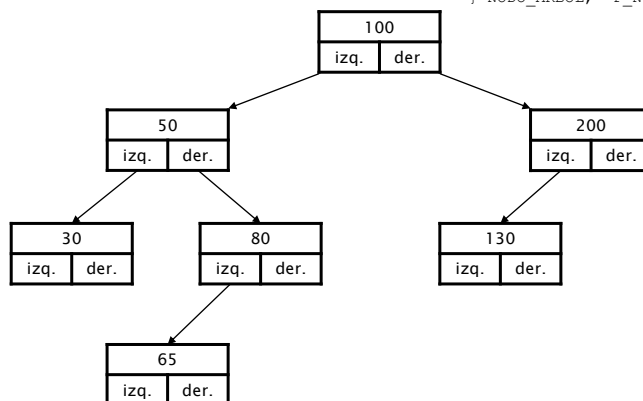
- Nada restringe los datos almacenados en un árbol.
- En la práctica, si existe una relación de orden ya que el dato de un nodo:
  - Suele ser mayor que el de todos los nodos que “cuelgan” del lado izquierdo.
  - Suele ser menor que el de todos los nodos que “cuelgan” del lado derecho.



# Árboles

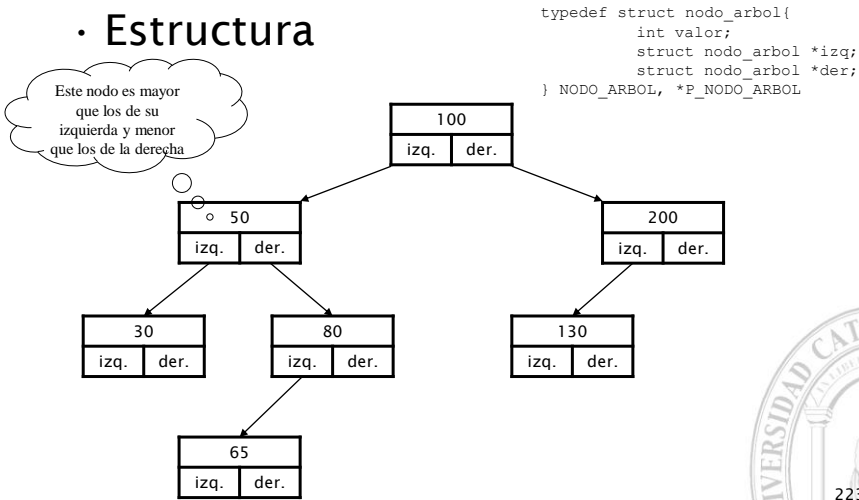
## • Estructura

```
typedef struct nodo_arbol{
    int valor;
    struct nodo_arbol *izq;
    struct nodo_arbol *der;
} NODO_ARBOL, *P_NODO_ARBOL
```



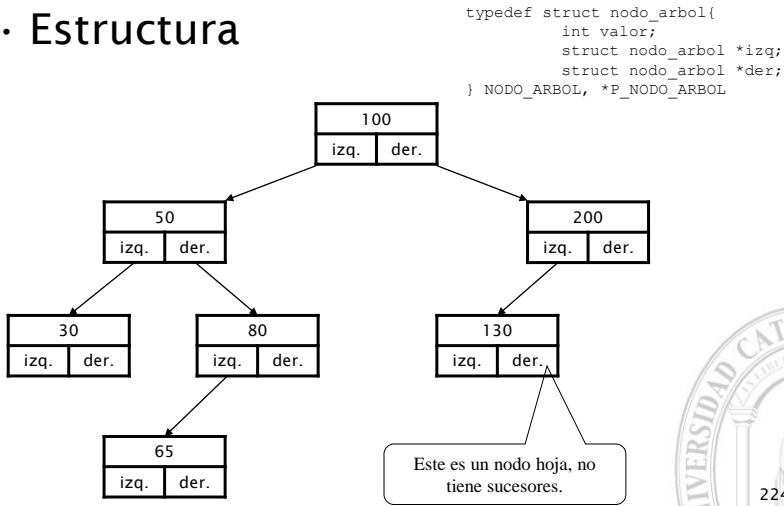
# Árboles

## • Estructura



# Árboles

## • Estructura





# Árboles

---

- Órdenes en un árbol binario
  - Recorrido en orden previo
  - Recorrido en orden simétrico
  - Recorrido en orden posterior



# Árboles

---

- Órdenes en un árbol binario
  - Recorrido en orden previo

Compuesto por: nodo raíz, seguido del recorrido en orden previo del subárbol izquierdo y seguido recorrido en orden previo del subárbol derecho
  - Recorrido en orden simétrico
  - Recorrido en orden posterior



# Árboles

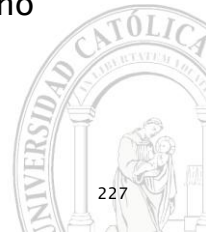
---

- Órdenes en un árbol binario

- Recorrido en orden previo
- Recorrido en orden simétrico

Compuesto por: recorrido en orden simétrico del subárbol izquierdo, seguido del nodo raíz y seguido del recorrido en orden simétrico del subárbol derecho

- Recorrido en orden posterior



# Árboles

---

- Órdenes en un árbol binario

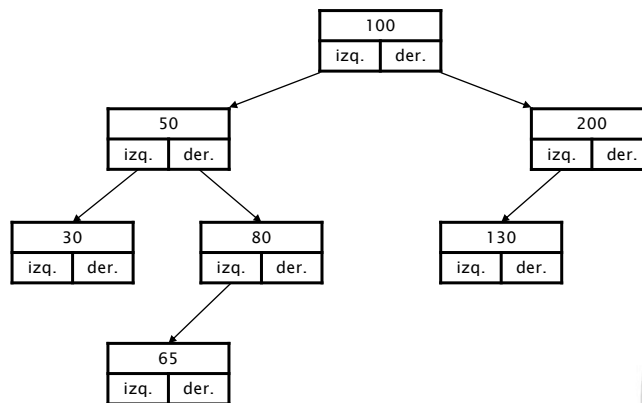
- Recorrido en orden previo
- Recorrido en orden simétrico
- Recorrido en orden posterior

Compuesto por: recorrido en orden posterior del subárbol izquierdo, seguido del recorrido en orden posterior del subárbol derecho y seguido del nodo raíz.



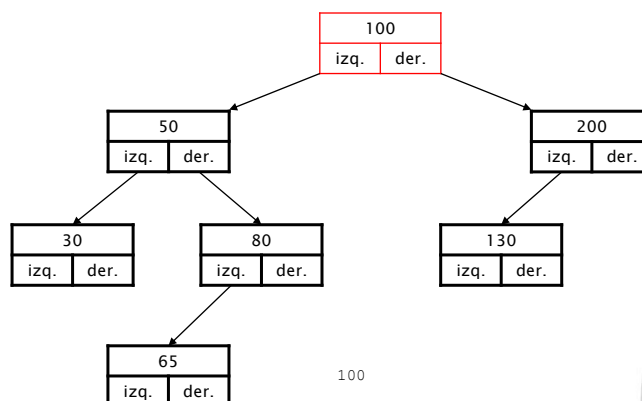
# Árboles

- Orden previo



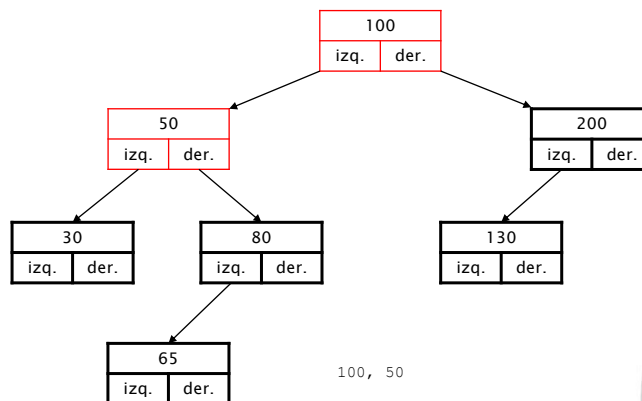
# Árboles

- Orden previo



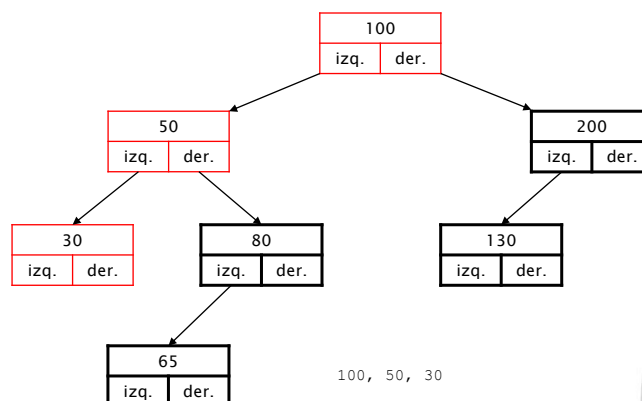
# Árboles

- Orden previo



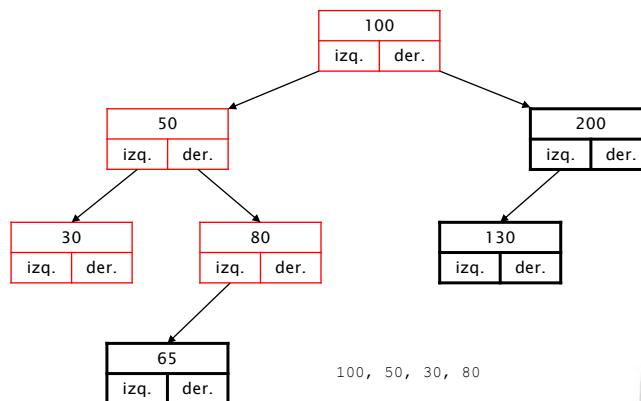
# Árboles

- Orden previo



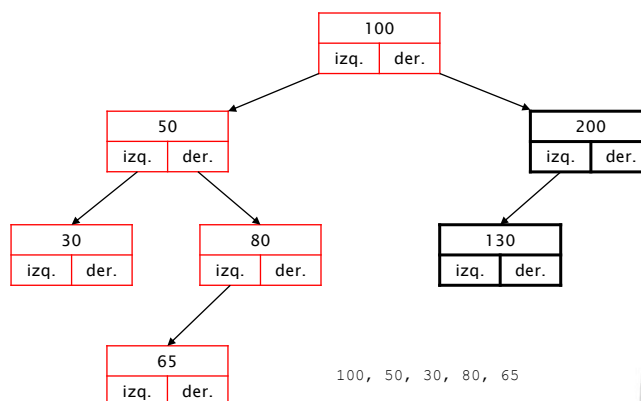
# Árboles

- Orden previo



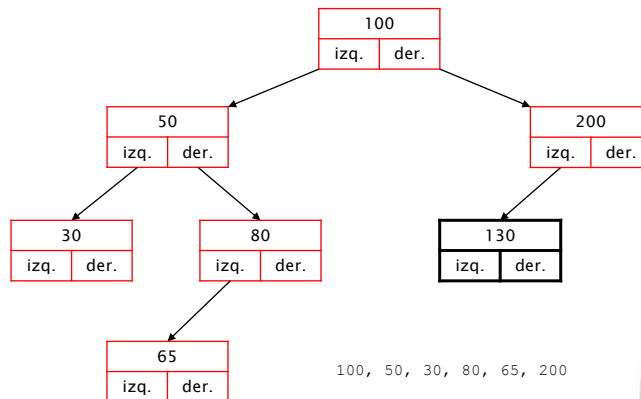
# Árboles

- Orden previo



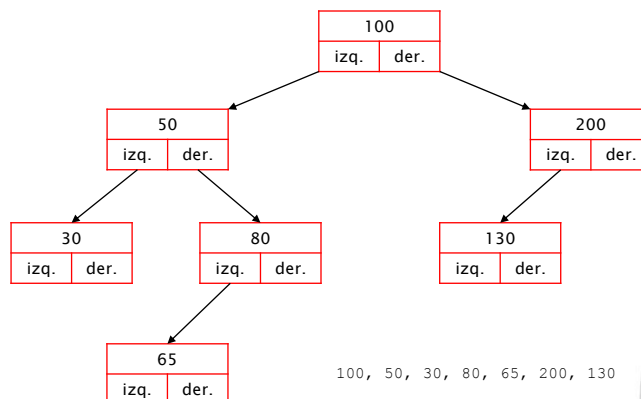
# Árboles

- Orden previo



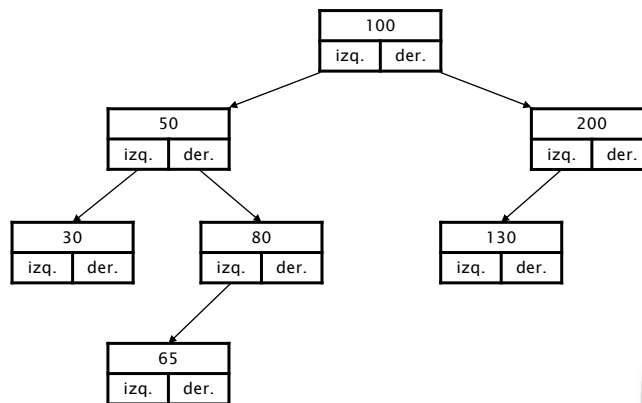
# Árboles

- Orden previo



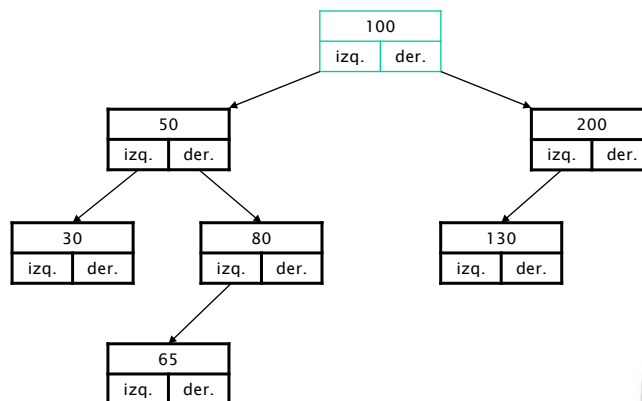
# Árboles

- Orden simétrico



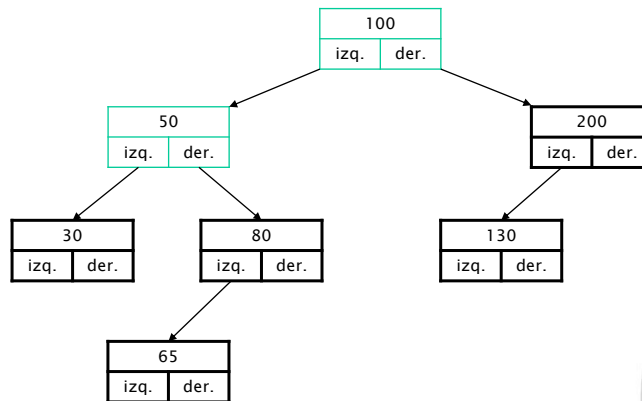
# Árboles

- Orden simétrico



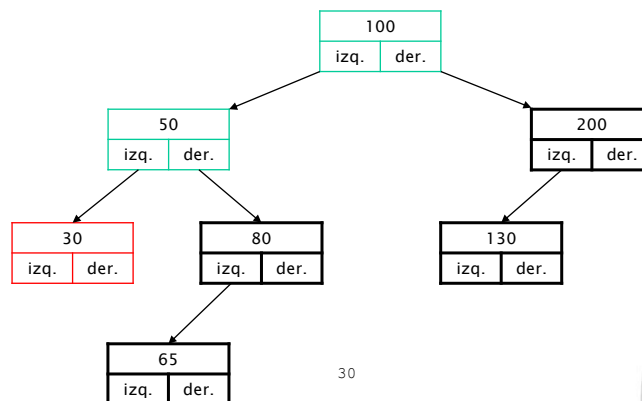
# Árboles

- Orden simétrico



# Árboles

- Orden simétrico



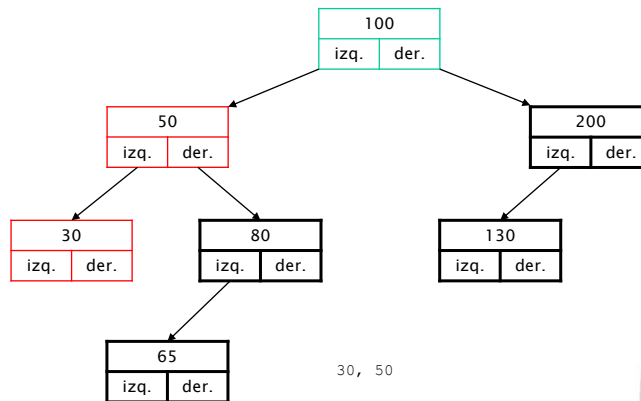
30





# Árboles

- Orden simétrico

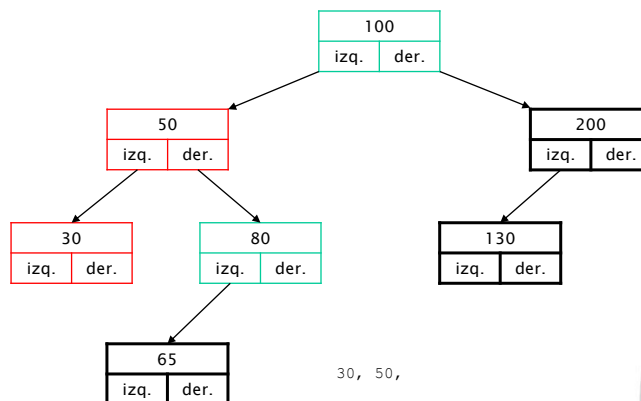


30, 50



# Árboles

- Orden simétrico

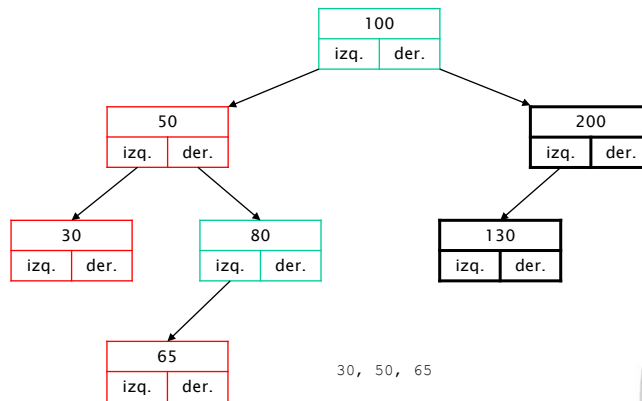


30, 50,



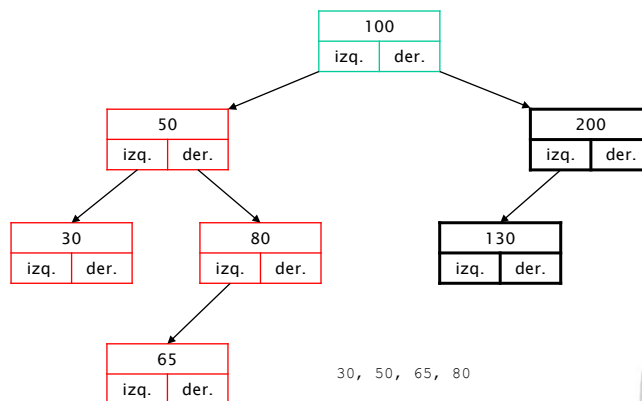
# Árboles

- Orden simétrico



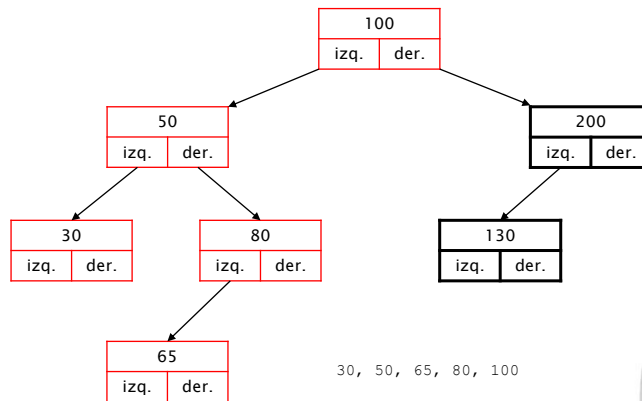
# Árboles

- Orden simétrico



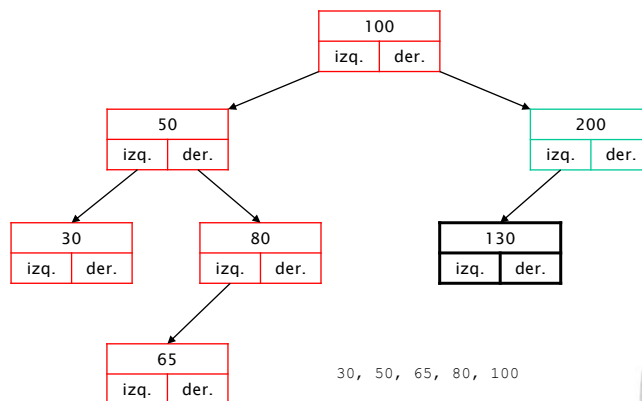
# Árboles

- Orden simétrico



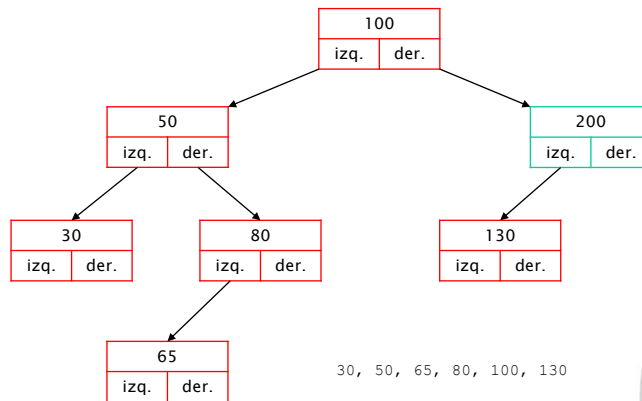
# Árboles

- Orden simétrico



# Árboles

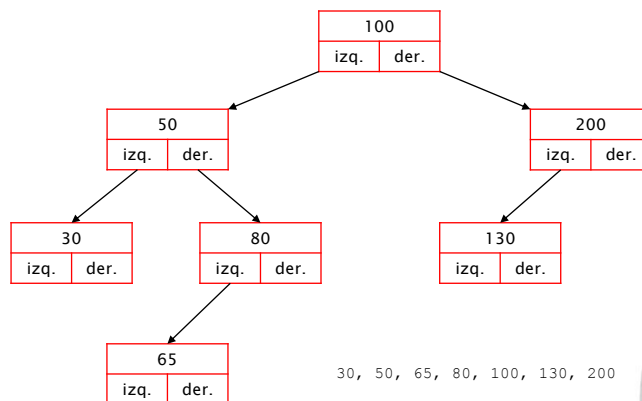
- Orden simétrico



247

# Árboles

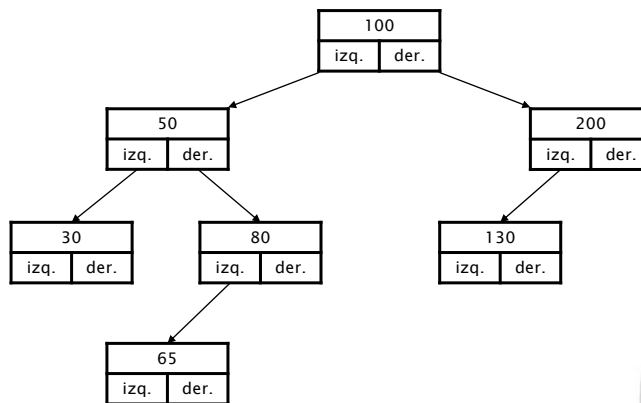
- Orden simétrico



248

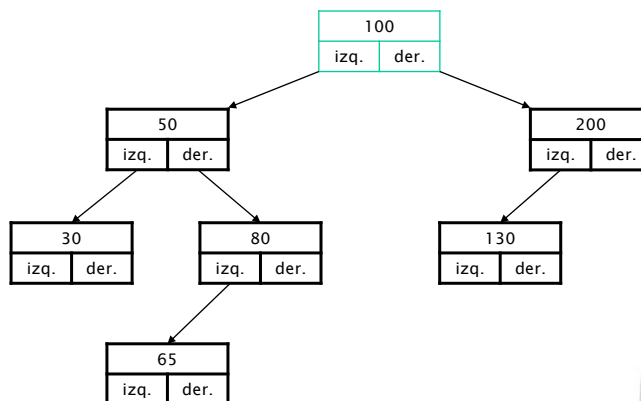
# Árboles

- Orden posterior



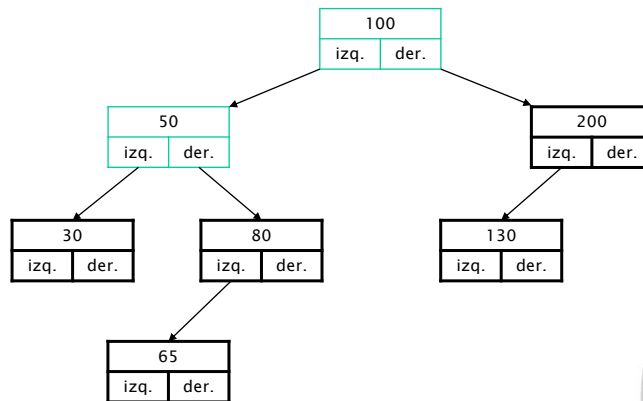
# Árboles

- Orden posterior



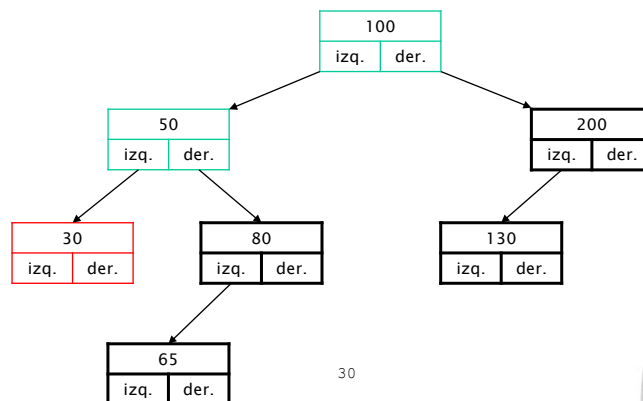
# Árboles

- Orden posterior



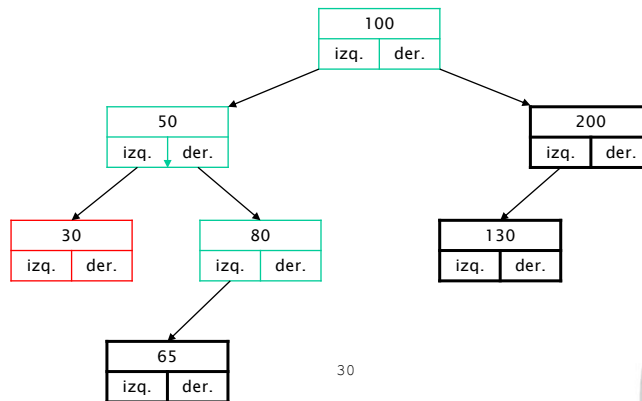
# Árboles

- Orden posterior



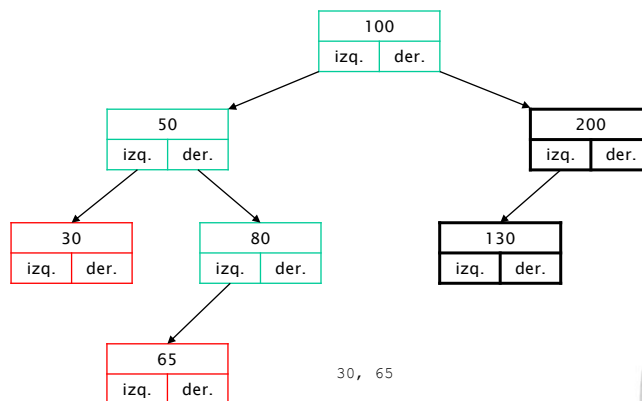
# Árboles

- Orden posterior



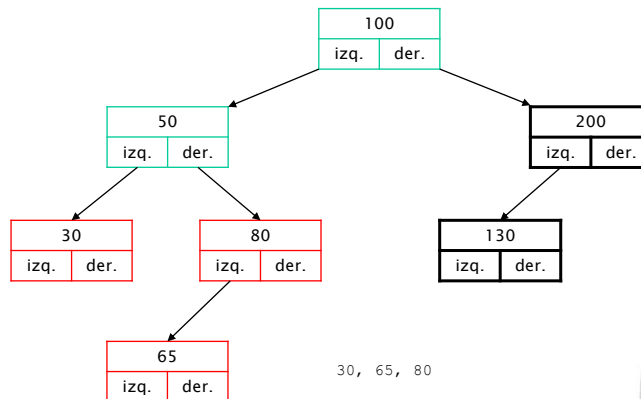
# Árboles

- Orden posterior



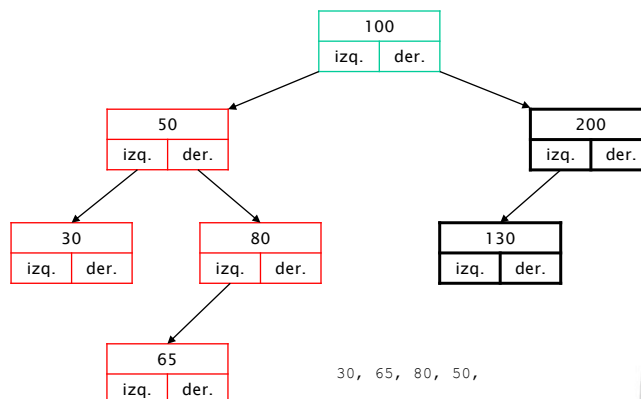
# Árboles

- Orden posterior



# Árboles

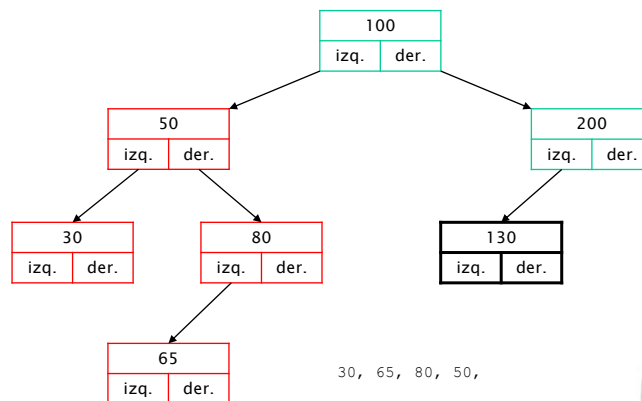
- Orden posterior





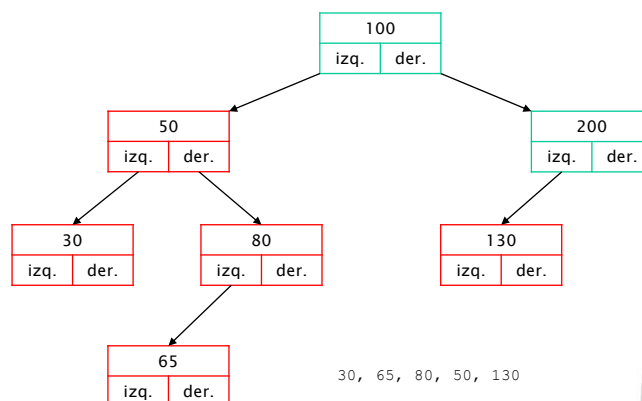
# Árboles

- Orden posterior



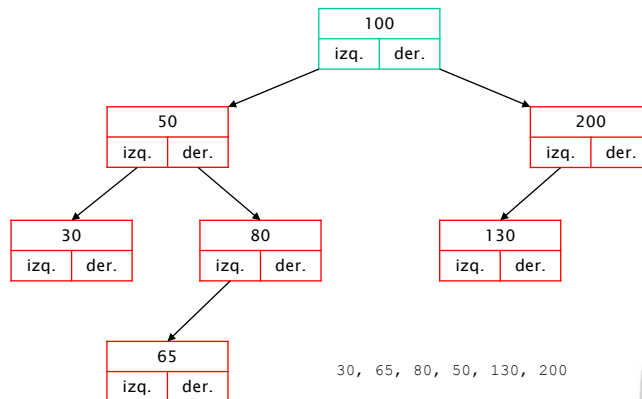
# Árboles

- Orden posterior



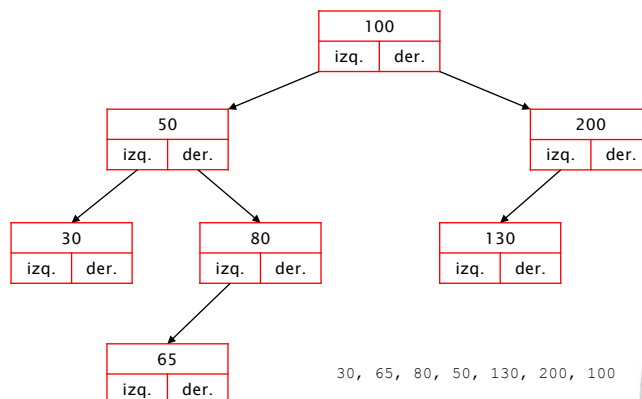
# Árboles

- Orden posterior



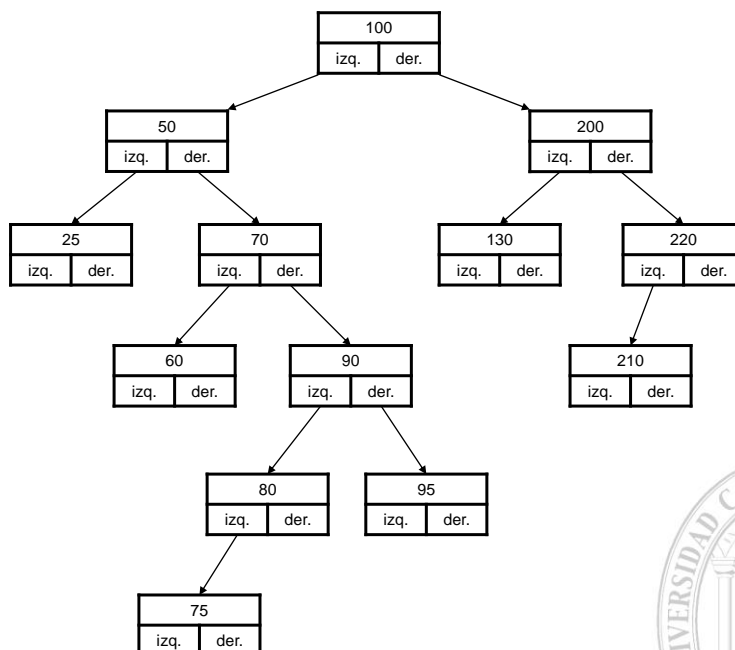
# Árboles

- Orden posterior



# Árboles

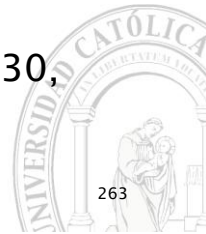
- Ejercicio:
  - Orden previo, simétrico y posterior del siguiente árbol



# Árboles

---

- Orden previo
  - 100, 50, 25, 70, 60, 90, 80, 75, 95, 200, 130, 220, 210.
- Orden simétrico
  - 25, 50, 60, 70, 75, 80, 90, 95, 100, 130, 200, 210, 220.
- Orden posterior
  - 25, 60, 75, 80, 95, 90, 70, 50, 130, 210, 220, 200, 100



# Árboles

---

- Operaciones básicas:
  - Insertar
  - Pertenece
  - Encontrar
  - Padre de
  - Borrar



# Lista: versión dinámica

---

- Operaciones básicas:
  - Insertar

```
P_NODO_ARBOL insertar (P_NODO_ARBOL arbol, int i){
    P_NODO_ARBOL p;
    if (arbol == NULL){
        p = alojar_nodo_arbol();
        p->izq = p->der = NULL;
        p->valor = i;
        return p;
    }
    if (arbol->valor == i) return arbol;

    if (arbol->valor > i)
        arbol->izq = insertar(arbol->izq, i);
    else
        arbol->der = insertar(arbol->der, i);

    return arbol;
}
```



# Lista: versión dinámica

---

- Operaciones básicas:
  - Pertenece

```
int pertenece (P_NODO_ARBOL arbol, int i){
    if (arbol == NULL){
        return 0;
    }
    if (arbol->valor == i) return 1;

    if (arbol->valor > i)
        return pertenece(arbol->izq, i);
    else
        return pertenece(arbol->der, i);

}
```



# Lista: versión dinámica

---

- Operaciones básicas:
  - Encontrar

```
P_NODO_ARBOL encontrar (P_NODO_ARBOL arbol, int i){
    if (arbol == NULL){
        return NULL;
    }
    if (arbol->valor == i) return arbol;

    if (arbol->valor > i)
        return encontrar(arbol->izq, i);
    else
        return encontrar(arbol->der, i);
}
```



# Lista: versión dinámica

---

- Operaciones básicas:
  - Encontrar

```
P_NODO_ARBOL padre_de (P_NODO_ARBOL arbol, P_NODO_ARBOL arbol){
    if (arbol == NULL){
        return NULL;
    }
    if (arbol->izq == arbol || arbol->der == arbol)
        return arbol;

    else if (arbol->valor > arbol->valor)
        return padre_de(arbol->izq, arbol);
    else
        return padre_de (arbol->der, arbol);
}
```



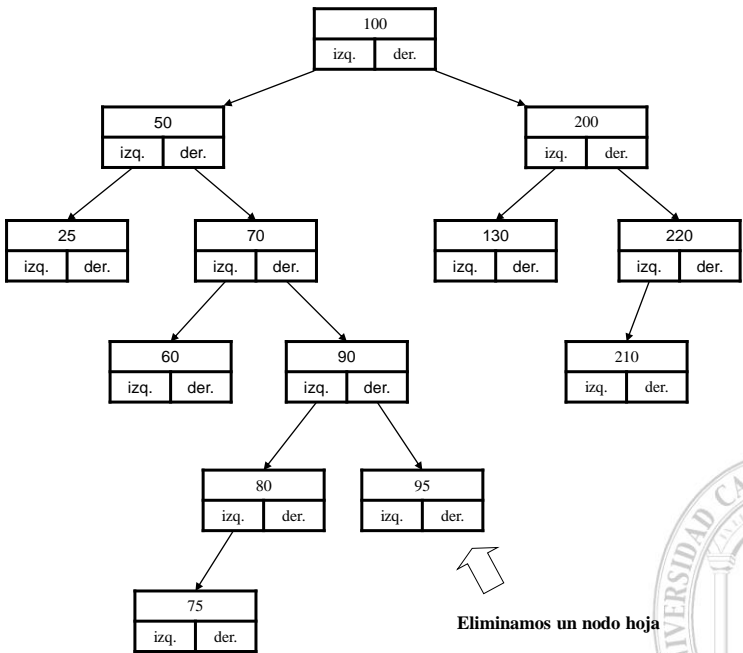
# Lista: versión dinámica

- Operaciones básicas:
  - Eliminar
    - La función de eliminar es más complicada ya que el árbol debe de quedar ordenado
    - Habrá tres casos:

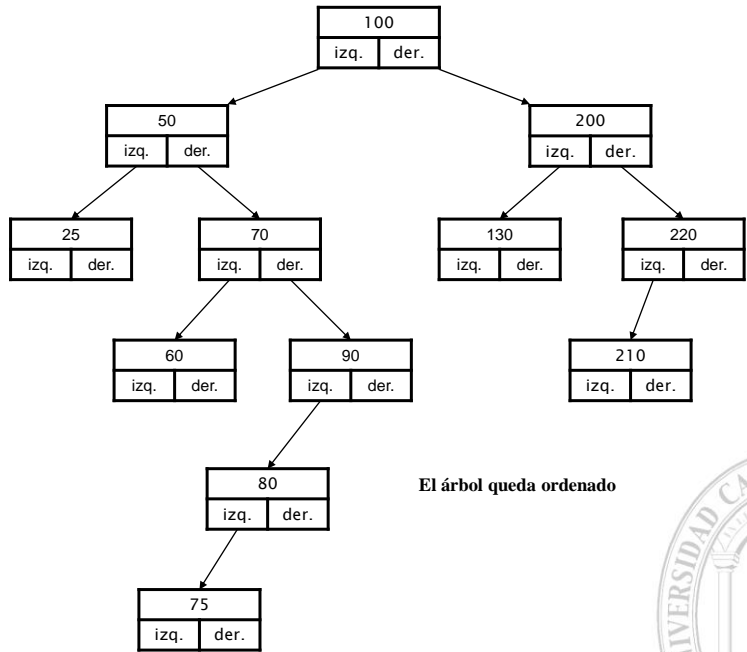
Nº	Caso	Solución
1	El nodo sea hoja	Se borra directamente
2	El nodo tenga una rama vacía	Se borra y se deja la otra rama "colgando" de donde lo hace ahora el nodo
3	El nodo tenga nodos en ambas ramas	Se busca el mayor de los menores para sustituirlo por el que se quiere eliminar.

269

Caso 1



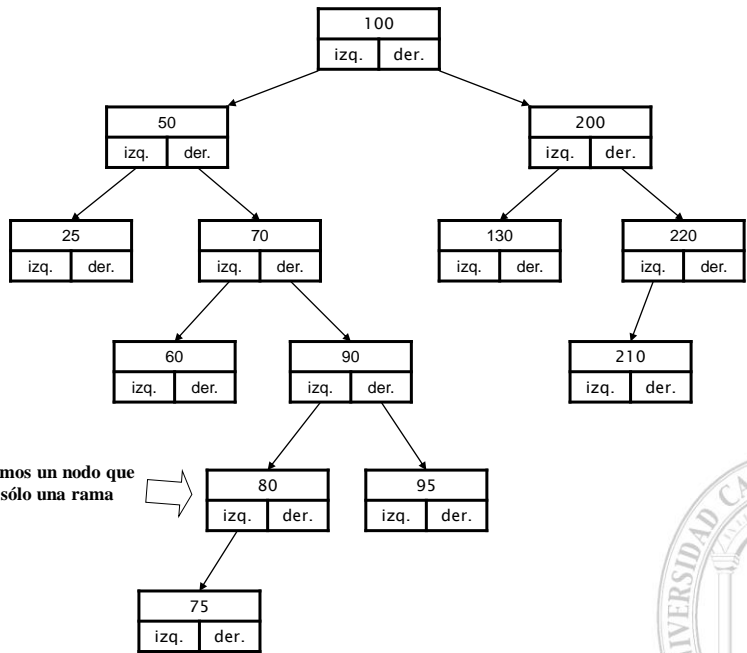
Caso 1



El árbol queda ordenado



Caso 2

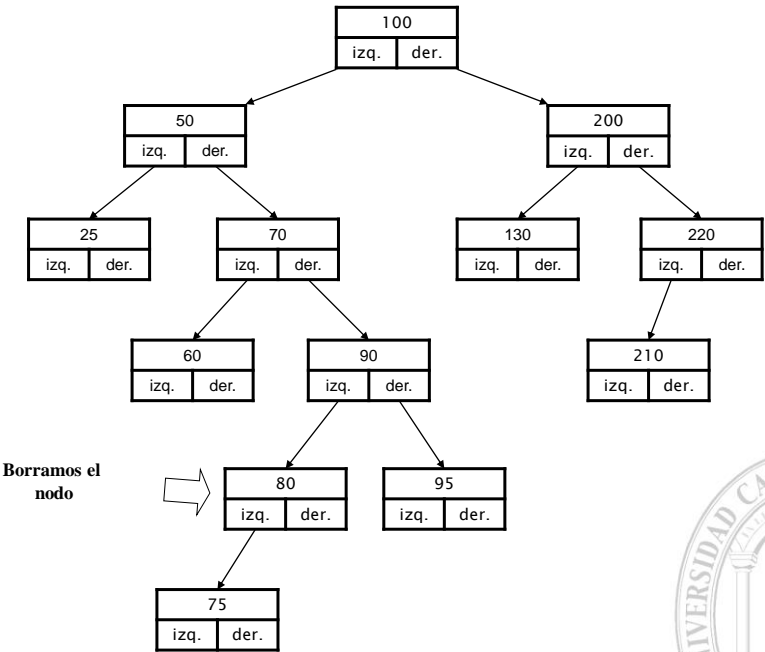


Eliminamos un nodo que tiene sólo una rama

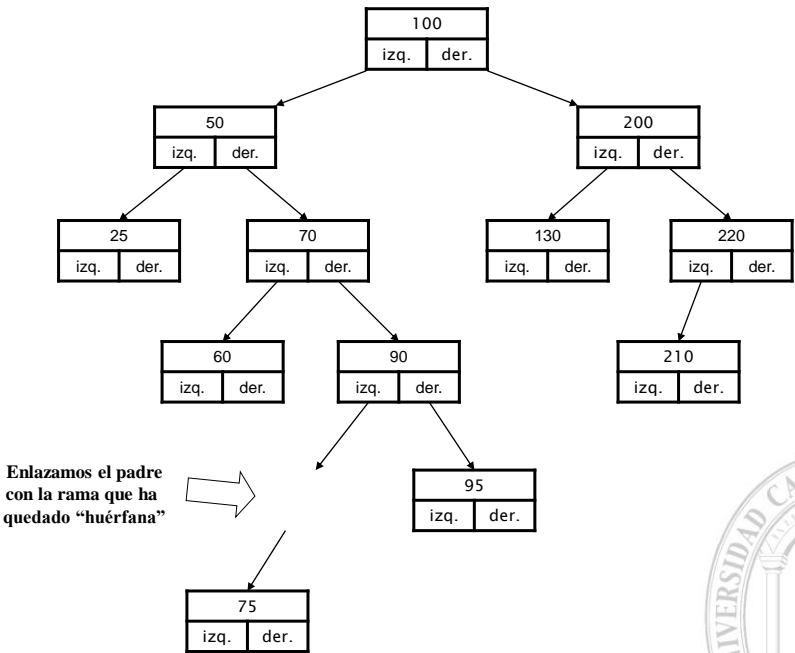




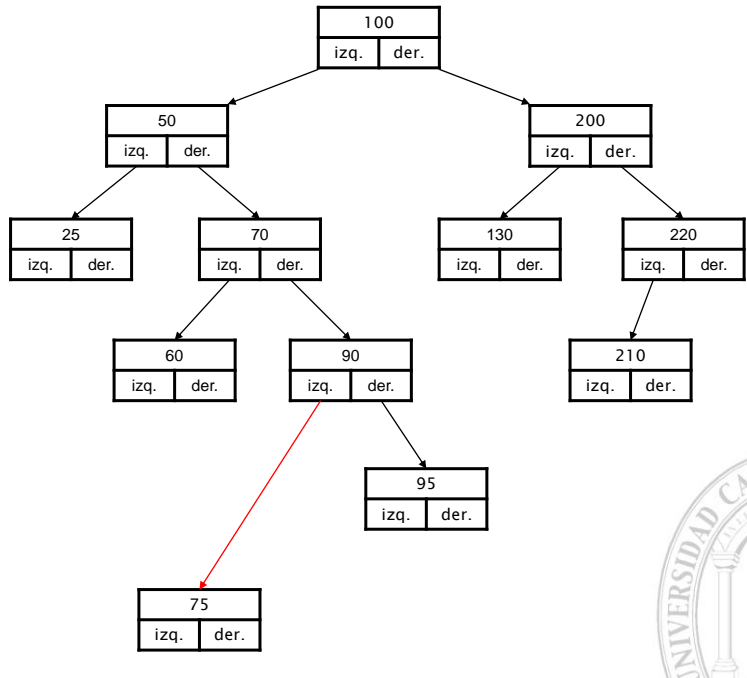
Caso 2



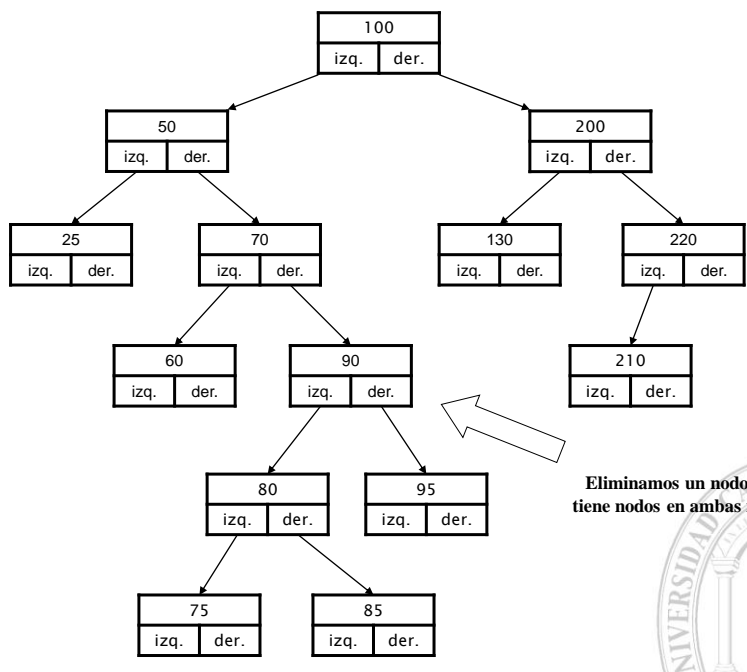
Caso 2



Caso 2



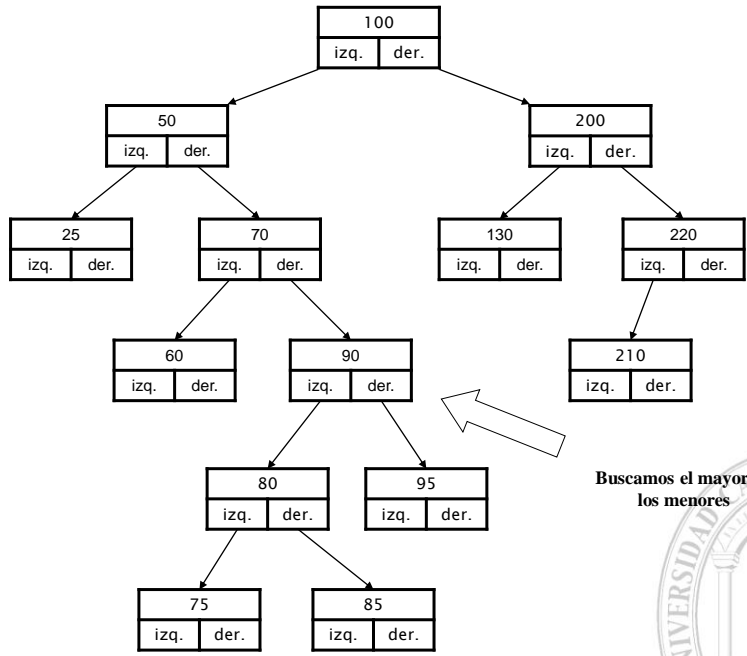
Caso 3



Eliminamos un nodo que tiene nodos en ambas ramas



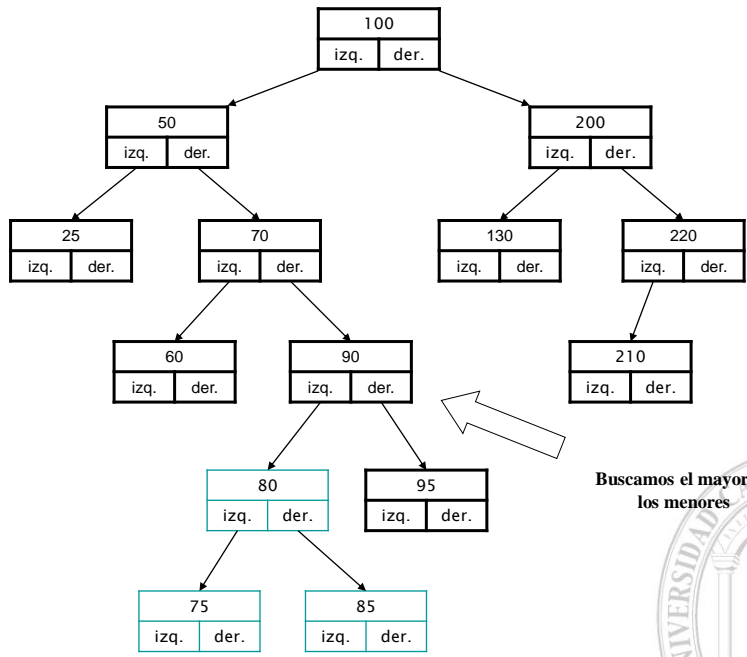
Caso 3



Buscamos el mayor de los menores



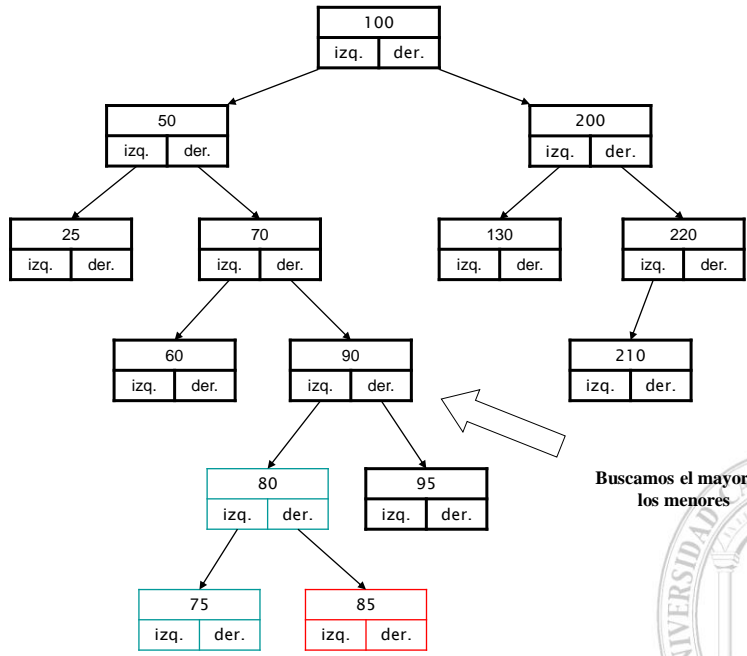
Caso 3



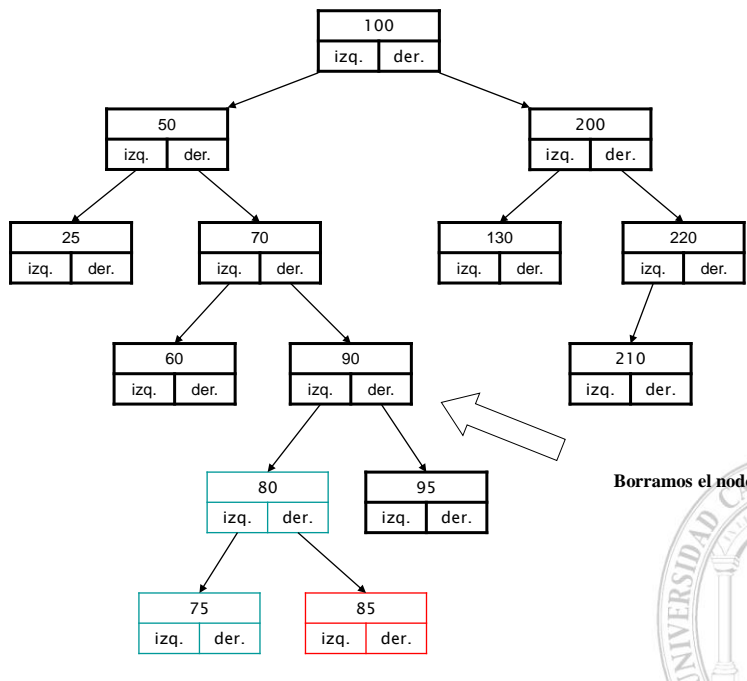
Buscamos el mayor de los menores



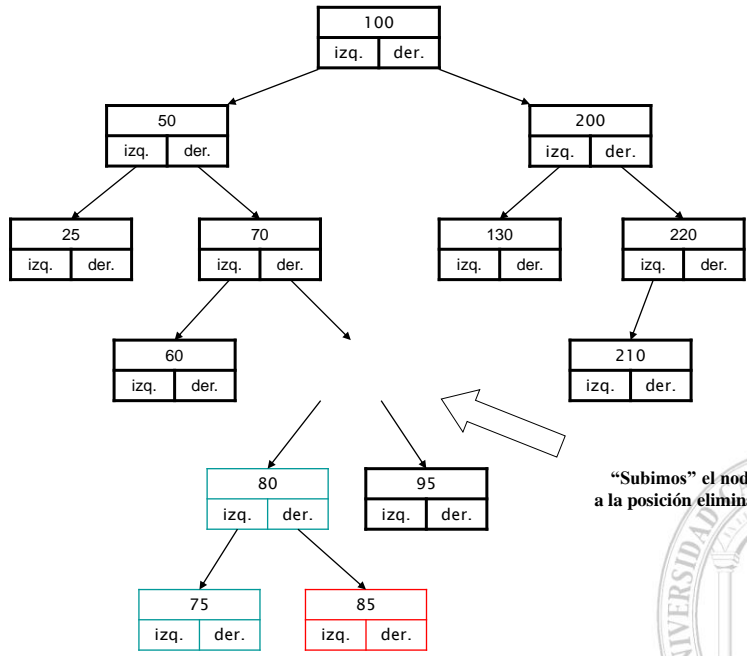
Caso 3



Caso 3



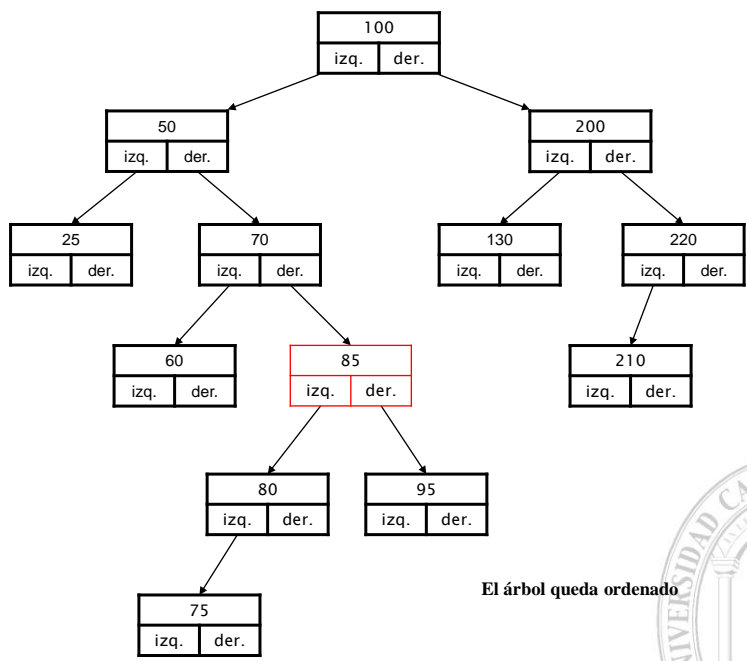
Caso 3



“Subimos” el nodo a la posición eliminada



Caso 3



El árbol queda ordenado



```

P_NODO_ARBOL eliminar (P_NODO_ARBOL a, int i){
    P_NODO_LISTA q,p;
    if (a == NULL) return a;
    if (a->valor > i) a-> izq = eliminar (a->izq, i);
    else if (a->valor < i) a-> der = eliminar (a->der, i);
    else{
        if ((a->der == NULL)&&(a->izq == NULL)) //caso 1
            return borraNodo(a);
        //Caso 2
        if (a->der == NULL){
            q = a->izq;
            borraNodo (a);
            return q;
        }
        if (a->izq == NULL){
            q = a->der;
            borraNodo (a);
            return q;
        }
        //Caso 3
        for(p=a,q=a->izq; q->der != NULL; q = q->der)
            p=q;
        if (p!=a)
            p-> der = q->izq;
        else
            p->izq =q->izq;
        a->valor =q->value;
        borrarNodo(q);
    }
    return a;
}

```



## Grafos

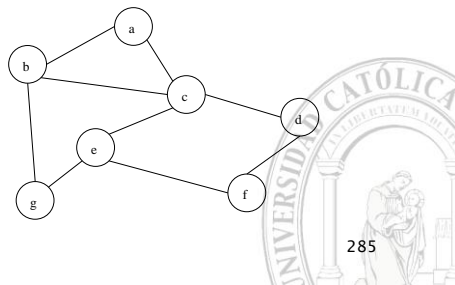
- Cada nodo tiene
  - Uno o más predecesores
  - Uno o más sucesores
- Dos tipos:
  - No dirigido
  - Dirigido



# Grafos

---

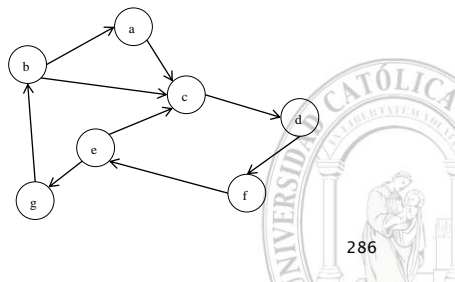
- Cada nodo tiene
  - Uno o más predecesores
  - Uno o más sucesores
- Dos tipos:
  - No dirigido
    - Sin dirección
  - Dirigido



# Grafos

---

- Cada nodo tiene
  - Uno o más predecesores
  - Uno o más sucesores
- Dos tipos:
  - No dirigido
  - Dirigido
    - Con dirección



# Grafos

---

- Ejemplos:
  - Red de carreteras de un país.
    - Un arco = carretera entre dos localidades.
    - Se asocia un peso al arco.
  - Un autómata
    - Un arco = transición entre dos estados
    - Se asocia el evento que genera la transición al arco



# Grafos

---

- Representación:
  - Matriz de adyacencia A
  - $A[i,j] = \text{true}$  si existe un arco entre el nodo 'i' y el 'j'
  - Se pueden definir las etiquetas de los arcos mediante estructuras en la posición de la matriz.

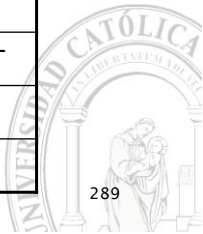




# Grafos

- Matriz de adyacencia (no dirigido)

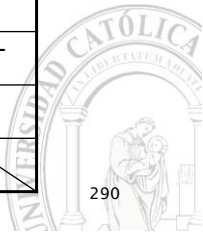
	a	b	c	d	e	f	g
a		T	T				
b	T		T				T
c	T	T		T	T		
d			T			T	
e			T			T	T
f				T	T		
g		T			T		



# Grafos

- Matriz de adyacencia (no dirigido)

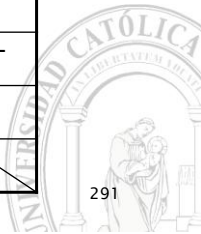
	a	b	c	d	e	f	g
a		T	T				
b	T		T				T
c	T	T		T	T		
d			T			T	
e			T			T	T
f				T	T		
g		T			T		



# Grafos

- Matriz de adyacencia (no dirigido)

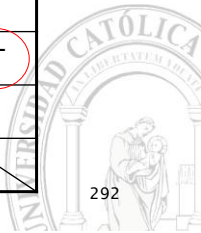
	a	b	c	d	e	f	g
a		T	T				
b	T		T				T
c	T	T		T	T		
d			T			T	
e			T			T	T
f				T	T		
g		T			T		



# Grafos

- Matriz de adyacencia (no dirigido)

	a	b	c	d	e	f	g
a		T	T				
b	T		T				T
c	T	T		T	T		
d			T			T	
e			T			T	T
f				T	T		
g		T			T		

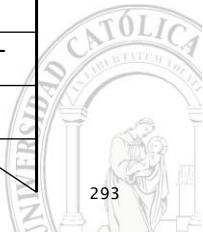


# Grafos

- Matriz de adyacencia (no dirigido)

	a	b	c	d	e	f	g
a		T	T				
b			T				T
c				T	T		
d						T	
e						T	T
f							
g							

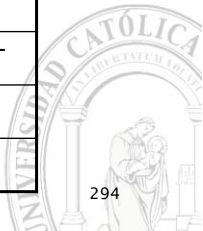
Hay una redundancia de información  
"Sobra" toda esta parte



# Grafos

- Matriz de adyacencia (dirigido)

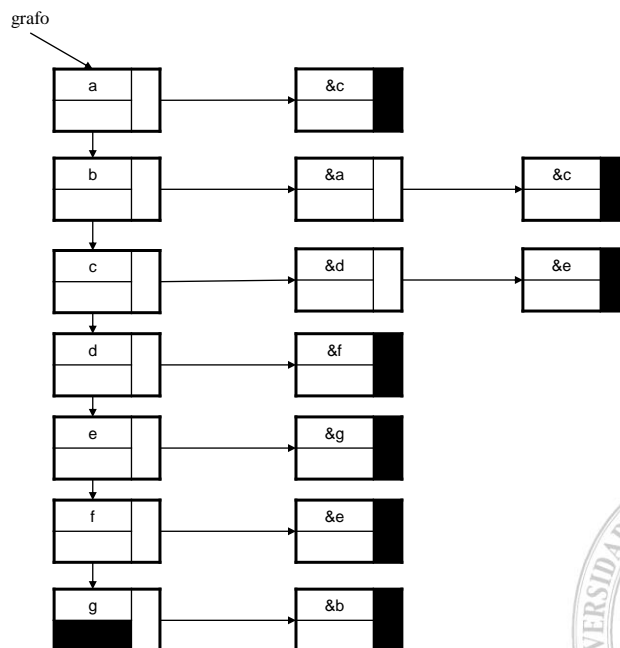
	a	b	c	d	e	f	g
a			T				
b	T		T				
c				T	T		
d						T	
e							T
f					T		
g		T					

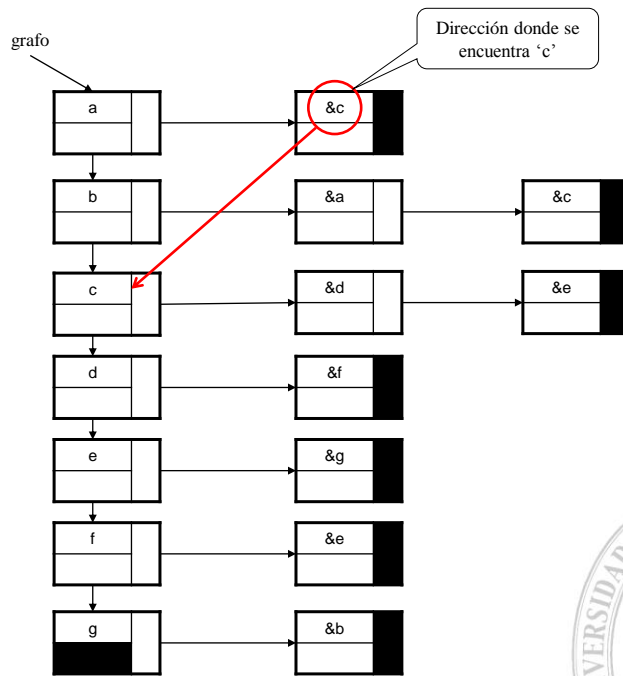


# Grafos

- Un grafo también se puede representar mediante listas
  - Es irrelevante el orden en que aparecen los nodos en la lista principal

(Ver siguiente transparencia)





## Grafos

- Problemas típicos
  - Dijkstra: caminos más cortos con un solo origen.
  - Floyd: recuperación de caminos.
  - Warshall: cierre transitivo de la conectividad.



## Grafos: Dijkstra

---

- Problema: encontrar la forma más económica de moverse desde un nodo a otro.
- Elementos:
  - N nodos numerados de 0 a  $n-1$ .  
Siendo 0 el nodo origen.
  - Matriz de costes  $C[i,j]$ : coste de ir del nodo 'i' al 'j'.
    - No negativos.



## Grafos: Dijkstra

---

- Problema: encontrar la forma más económica de moverse desde un nodo a otro.
- Elementos:
  - Vector de costes  $D[i]$ : coste total de ir del nodo origen al 'i'.
    - Inicialmente  $D[i] = C[0,i]$
  - Si no se puede ir del nodo 'i' al 'j' entonces  $C[i,j] = \infty$



# Grafos: Dijkstra

- Esquema del algoritmo (*pseudo-C*)

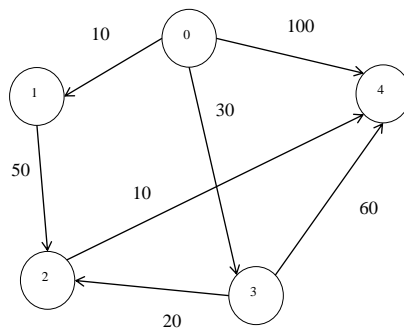
```
void Dijkstra(unsigned int c[][], unsigned int d[], unsigned int nNodos){
    ConjuntoDeEnteros U;      //conjunto universal de nodos
    ConjuntoDeEnteros S;      //Conjunto universal de nodos
    int v, w;                  //Números de nodos

    U={};
    for (i = 1; i< nNodos; i++){
        D[i] = C[0,i];
    }
    repetir (nNodos -1) veces{
        w = elegir un nodo en U-S sea minimo
        S = S ∪ { w };
        para cada nodo v en U-S
            D[v] = min(D[v], D[w] + C[w,v]);
    }
}
```



# Grafos: Dijkstra

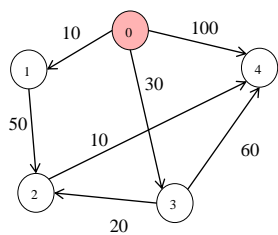
- Ejemplo



# Grafos: Dijkstra

• Ejemplo

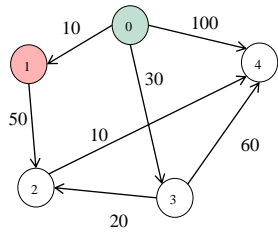
Iteración	S	w	D[1]	D[2]	D[3]	D[4]
inicial	{0}	-	10	$\infty$	30	100



# Grafos: Dijkstra

• Ejemplo

Iteración	S	w	D[1]	D[2]	D[3]	D[4]
inicial	{0}	-	10	$\infty$	30	100
1	{0,1}	1	10	60	30	100

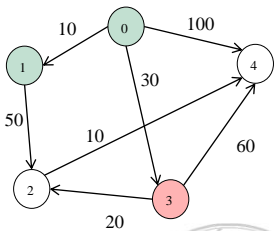




# Grafos: Dijkstra

• Ejemplo

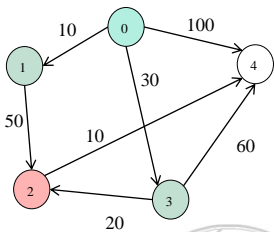
Iteración	S	w	D[1]	D[2]	D[3]	D[4]
inicial	{0}	-	10	$\infty$	30	100
1	{0,1}	1	10	60	30	100
2	{0,1,3}	3	10	50	30	90



# Grafos: Dijkstra

• Ejemplo

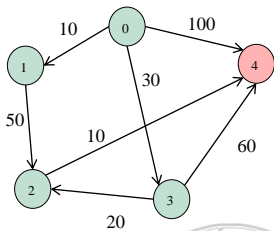
Iteración	S	w	D[1]	D[2]	D[3]	D[4]
inicial	{0}	-	10	$\infty$	30	100
1	{0,1}	1	10	60	30	100
2	{0,1,3}	3	10	50	30	90
3	{0,1,3,2}	2	10	50	30	60



# Grafos: Dijkstra

- Ejemplo

Iteración	S	w	D[1]	D[2]	D[3]	D[4]
inicial	{0}	-	10	$\infty$	30	100
1	{0,1}	1	10	60	30	100
2	{0,1,3}	3	10	50	30	90
3	{0,1,3,2}	2	10	50	30	60
4	{0,1,3,2,4}	4	10	50	30	60



307

# Grafos: Floyd

- Problema: conocer cuál es el camino que debemos recorrer para conseguir el coste mínimo.
- Elementos:
  - Matriz de costes no negativos  $C[i,j]$ .
  - Matriz de costes  $A[i,j]$ : coste mínimo entre cualquier par de nodos 'i' y 'j'.
  - Matriz de nodos  $P[i,j]$ , número de un nodo que se encuentra en algún punto intermedio de camino mínimo.



308

# Grafos: Floyd

---

- Esquema del algoritmo

```
void mas_corto(unsigned int c[][], unsigned int a[][], int P[][], unsigned
int nNodos){
    int i,j,k;

    for (i = 0; i < nNodos; i++){
        for(j=0; j < nNodos; j++){
            A[i][j] = C[i][j];
            P[i][j] = -1;
        }
    }
    for (k = 0; k < nNodos; k++)
        for (i = 0; i < nNodos; i++)
            for (j=0; j< nNodos; j++)
                if (A[i][k]+A[k][j] < A[i][j]){
                    A[i][j] = A[i][k] + A[k][j];
                    P[i][j] = k;
                }
}
```



# Grafos: Floyd

---

- Función para obtener el listado del camino más corto

```
void camino (int P[][], int i, int j){
    int k;

    if ((k=P[i][j])== -1)
        return;
    camino(i,k);
    printf("%d ",k);
    camino(k,j);
}
```



# Grafos: Warshall

---

- Problema: conocer si existe conectividad entre dos nodos
  - No interesa el coste.
  - No interesa el camino.
- Elementos:
  - Matriz de conexiones  $C[i,j]$ , indica la existencia de conexión directa.
  - Matriz de conexiones  $A[i,j]$ , indica la existencia de conexión entre 'i' y 'j'.



# Grafos: Floyd

---

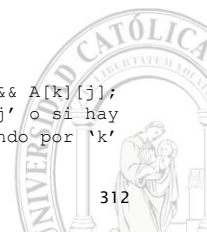
- Esquema del algoritmo

```
#define boolean int

void warshall (boolean c[], boolean a[], unsigned int nNodos){
    int i,j,k;

    for (i = 0; i < nNodos; i++)
        for (j=0; j< nNodos; j++)
            A[i][j] = C[i][j];

    for (k = 0; k < nNodos; k++)
        for (i = 0; i < nNodos; i++)
            for (j=0; j< nNodos; j++)
                A[i][j] = A[i][j] || A[i][k] && A[k][j];
                //Si hay un camino de 'i' a 'j' o si hay
                //un camino de 'i' a 'j' pasando por 'k'
}
```



# Índice

---

- Introducción
- Estructuras lineales
- Estructuras no lineales
- Estructuras en memoria secundaria



## Memoria secundaria

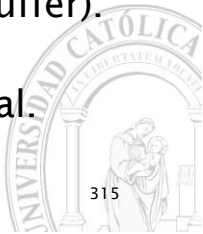
---

- Los *archivos* o *ficheros* en memoria secundaria son estructuras de datos con persistencia.
- Características
  - Realizar operaciones de apertura.
  - Realizar operaciones de cierre.
  - El acceso a los datos no se hace directamente
    - Operaciones de lectura y escritura



# Memoria secundaria

- Los *archivos* o *ficheros* en memoria secundaria son estructuras de datos con persistencia.
- Características
  - Se utiliza un buffer para leer y escribir.
  - FILE \* es un puntero al fichero (buffer).
    - Se incluye en stdio.h
  - El acceso a los datos es secuencial.

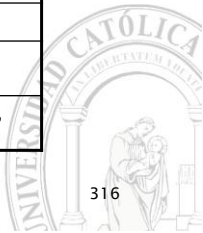


# Memoria secundaria

- Operación de apertura

```
FILE *fopen(char *nombre_fichero, char *modo)
```

"r"	Abre el fichero para lectura
"w"	Crea el fichero para escritura, o lo trunca a longitud cero si ya existe
"a"	Añadir. Abre el fichero (o lo crea si no existe) para escritura al final
"r+"	Abre el fichero para actualización
"w+"	Crea el fichero para actualización o lo trunca a longitud cero si existe
"a+"	Abre el fichero (o lo crea si no existe) para actualización, pero las escrituras serán realizadas al final



# Memoria secundaria

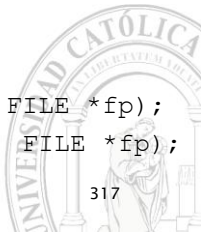
- Operación de cierre

```
int fclose( FILE *fp);
```

- Devuelve cero si se cierra correctamente
- Devuelve otro valor si no se realiza correctamente.

- Lectura y escritura

```
size_t fread(void *buf, size_t tam, size_t n, FILE *fp);
size_t fwrite(void *buf, size_t tam, size_t n, FILE *fp);
```



# Memoria secundaria

- Acceso secuencial

```
int fseek(FILE *fp, long int offset, int origen);
```

SEEK_SET	La posición indicada por offset se considera con respecto al comienzo del archivo
SEEK_CUR	La posición indicada por offset se considera con respecto a la posición actual dentro del archivo
SEEK_END	La posición indicada por offset se considera con respecto al final del archivo

fseek(fp,0,SEEK_SET)	Posiciona el fichero fp al principio
int i fseek(fp,i,SEEK_CUR)	Retrocede el fichero fp tantas posiciones como ocupa el entero
fseek(fp,0,SEEK_END)	Posiciona el fichero fp al final

