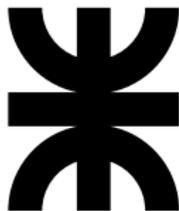


# Informática II

## Estructuras de datos

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional  
Facultad Regional Córdoba  
UTN-FRC

– 2019 –

# Estructuras de datos – Importancia de los datos

Conocimiento básico de C:

# Estructuras de datos – Importancia de los datos

Conocimiento básico de C:

- ▶ **Tipos de datos:** int, float/double, arreglos
- ▶ **Aritmética** (operadores)
- ▶ **Condiciones:** if, else, switch
- ▶ **Bucles:** for, while, do-while
- ▶ **Punteros, Funciones** (recursión), entrada/salida, etc.

# Estructuras de datos – Importancia de los datos

Conocimiento básico de C:

- ▶ **Tipos de datos:** int, float/double, arreglos
- ▶ **Aritmética** (operadores)
- ▶ **Condiciones:** if, else, switch
- ▶ **Bucles:** for, while, do-while
- ▶ **Punteros, Funciones** (recursión), entrada/salida, etc.

Esto es suficiente para resolver “cualquier” problema.

# Estructuras de datos – Importancia de los datos

Conocimiento básico de C:

- ▶ **Tipos de datos:** `int`, `float/double`, arreglos
- ▶ **Aritmética** (operadores)
- ▶ **Condiciones:** `if`, `else`, `switch`
- ▶ **Bucles:** `for`, `while`, `do-while`
- ▶ **Punteros, Funciones** (recursión), entrada/salida, etc.

Esto es suficiente para resolver “cualquier” problema.

Enfoques para la resolución de problemas:

- ▶ A partir de los datos entrada, el problema es determinar la secuencia de instrucciones (algoritmo) para obtener los datos de salida.

# Estructuras de datos – Importancia de los datos

Conocimiento básico de C:

- ▶ **Tipos de datos:** `int`, `float/double`, arreglos
- ▶ **Aritmética** (operadores)
- ▶ **Condiciones:** `if`, `else`, `switch`
- ▶ **Bucles:** `for`, `while`, `do-while`
- ▶ **Punteros, Funciones** (recursión), entrada/salida, etc.

Esto es suficiente para resolver “cualquier” problema.

Enfoques para la resolución de problemas:

- ▶ A partir de los datos entrada, el problema es determinar la secuencia de instrucciones (algoritmo) para obtener los datos de salida. La importancia está en el algoritmo.

# Estructuras de datos – Importancia de los datos

Conocimiento básico de C:

- ▶ **Tipos de datos:** int, float/double, arreglos
- ▶ **Aritmética** (operadores)
- ▶ **Condiciones:** if, else, switch
- ▶ **Bucles:** for, while, do-while
- ▶ **Punteros, Funciones** (recursión), entrada/salida, etc.

Esto es suficiente para resolver “cualquier” problema.

Enfoques para la resolución de problemas:

- ▶ A partir de los datos entrada, el problema es determinar la secuencia de instrucciones (algoritmo) para obtener los datos de salida. La importancia está en el algoritmo.
- ▶ **Nuevo enfoque:** poner en el centro a los datos o la forma de almacenar los datos.

# Estructuras de datos – Importancia de los datos

Conocimiento básico de C:

- ▶ **Tipos de datos:** int, float/double, arreglos
- ▶ **Aritmética** (operadores)
- ▶ **Condiciones:** if, else, switch
- ▶ **Bucles:** for, while, do-while
- ▶ **Punteros, Funciones** (recursión), entrada/salida, etc.

Esto es suficiente para resolver “cualquier” problema.

Enfoques para la resolución de problemas:

- ▶ A partir de los datos entrada, el problema es determinar la secuencia de instrucciones (algoritmo) para obtener los datos de salida. La importancia está en el algoritmo.
- ▶ **Nuevo enfoque:** poner en el centro a los datos o la forma de almacenar los datos. Estos datos se van modificando a lo largo del programa (algoritmo).

# Estructuras de datos

- ▶ Tamaño fijo → arreglos (uno o múltiples subíndices) y **struct**

# Estructuras de datos

- ▶ Tamaño fijo → arreglos (uno o múltiples subíndices) y **struct**
- ▶ Tamaño variable → *estructuras dinámicas de datos* (lineales y no-lineales)

# Estructuras de datos

- ▶ Tamaño fijo → arreglos (uno o múltiples subíndices) y **struct**
- ▶ Tamaño variable → *estructuras dinámicas de datos* (lineales y no-lineales)

Estructuras dinámicas de datos (4):

# Estructuras de datos

- ▶ Tamaño fijo → arreglos (uno o múltiples subíndices) y **struct**
- ▶ Tamaño variable → *estructuras dinámicas de datos* (lineales y no-lineales)

Estructuras dinámicas de datos (4):

1. *Listas enlazadas*: colección de elementos de datos alineados en una fila. La inserción y eliminación se efectúa en cualquier parte.

# Estructuras de datos

- ▶ Tamaño fijo → arreglos (uno o múltiples subíndices) y **struct**
- ▶ Tamaño variable → *estructuras dinámicas de datos* (lineales y no-lineales)

Estructuras dinámicas de datos (4):

1. *Listas enlazadas*: colección de elementos de datos alineados en una fila. La inserción y eliminación se efectúa en cualquier parte.
2. *Pilas*: apilamiento de datos (LIFO). La inserción y eliminación se efectúa en un extremo (cima).

# Estructuras de datos

- ▶ Tamaño fijo → arreglos (uno o múltiples subíndices) y **struct**
- ▶ Tamaño variable → *estructuras dinámicas de datos* (lineales y no-lineales)

Estructuras dinámicas de datos (4):

1. *Listas enlazadas*: colección de elementos de datos alineados en una fila. La inserción y eliminación se efectúa en cualquier parte.
2. *Pilas*: apilamiento de datos (LIFO). La inserción y eliminación se efectúa en un extremo (cima).
3. *Colas*: representan listas de espera (FIFO). La inserción se efectúa en la parte trasera (cola), y la eliminación de la parte delantera (cabeza).

# Estructuras de datos

- ▶ Tamaño fijo → arreglos (uno o múltiples subíndices) y **struct**
- ▶ Tamaño variable → *estructuras dinámicas de datos* (lineales y no-lineales)

Estructuras dinámicas de datos (4):

1. *Listas enlazadas*: colección de elementos de datos alineados en una fila. La inserción y eliminación se efectúa en cualquier parte.
2. *Pilas*: apilamiento de datos (LIFO). La inserción y eliminación se efectúa en un extremo (cima).
3. *Colas*: representan listas de espera (FIFO). La inserción se efectúa en la parte trasera (cola), y la eliminación de la parte delantera (cabeza).
4. *Árboles binarios*: facilita la búsqueda y clasificación de datos a alta velocidad, la eliminación eficiente de elementos duplicados de datos, etc.

# Estructuras auto-referenciadas

Estructura que contiene un miembro de puntero al mismo tipo de estructura

# Estructuras auto-referenciadas

Estructura que contiene un miembro de puntero al mismo tipo de estructura

Ejemplo:

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

# Estructuras auto-referenciadas

Estructura que contiene un miembro de puntero al mismo tipo de estructura

Ejemplo:

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

Se define un tipo de estructura `struct nodo` con:

1. un miembro de dato entero `dato`
2. un miembro de puntero al mismo tipo de estructura `ptrSig`

# Estructuras auto-referenciadas

Estructura que contiene un miembro de puntero al mismo tipo de estructura

Ejemplo:

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

Se define un tipo de estructura `struct nodo` con:

1. un miembro de dato entero `dato`
2. un miembro de puntero al mismo tipo de estructura `ptrSig`

El miembro puntero (`ptrSig`) se conoce como *enlace* o *vínculo*, y permite vincular la estructura `struct nodo` a otra estructura.

# Estructuras auto-referenciadas

Estructura que contiene un miembro de puntero al mismo tipo de estructura

Ejemplo:

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

Se define un tipo de estructura `struct nodo` con:

1. un miembro de dato entero `dato`
2. un miembro de puntero al mismo tipo de estructura `ptrSig`

El miembro puntero (`ptrSig`) se conoce como *enlace* o *vínculo*, y permite vincular la estructura `struct nodo` a otra estructura.

Se pueden enlazar varias estructuras auto-referenciadas para formar estructuras de datos útiles como: listas, pilas, colas, y árboles.

# Estructuras auto-referenciadas

Estructura que contiene un miembro de puntero al mismo tipo de estructura

Ejemplo:

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

Se define un tipo de estructura `struct nodo` con:

1. un miembro de dato entero `dato`
2. un miembro de puntero al mismo tipo de estructura `ptrSig`

El miembro puntero (`ptrSig`) se conoce como *enlace* o *vínculo*, y permite vincular la estructura `struct nodo` a otra estructura.

Se pueden enlazar varias estructuras auto-referenciadas para formar estructuras de datos útiles como: listas, pilas, colas, y árboles.





# Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

# Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- ▶ Se tiene acceso a la lista mediante un puntero al primer nodo

# Listas enlazadas

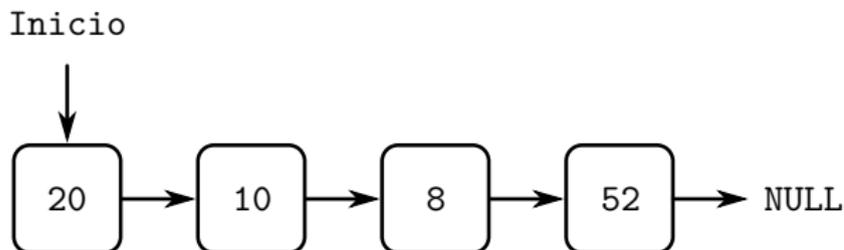
Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- ▶ Se tiene acceso a la lista mediante un puntero al primer nodo
- ▶ El acceso a otros nodos es mediante el puntero de enlace de cada nodo

# Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

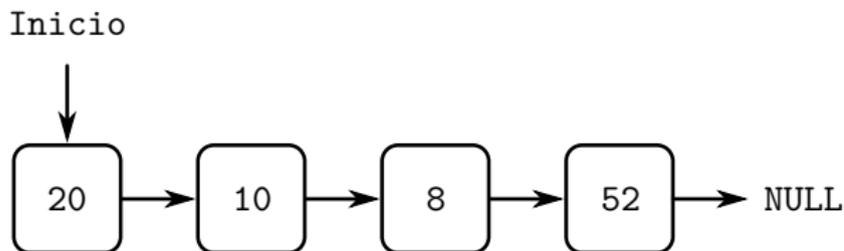
- ▶ Se tiene acceso a la lista mediante un puntero al primer nodo
- ▶ El acceso a otros nodos es mediante el puntero de enlace de cada nodo
- ▶ El final de la lista queda indicado por un puntero de enlace a NULL



# Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- ▶ Se tiene acceso a la lista mediante un puntero al primer nodo
- ▶ El acceso a otros nodos es mediante el puntero de enlace de cada nodo
- ▶ El final de la lista queda indicado por un puntero de enlace a NULL



Concepto recursivo

- ▶ Tiene un inicio/cabeza y una cola
- ▶ La cola es otra lista

# Listas enlazadas

## Ventajas y desventajas respecto a los arreglos

- ▶  Son dinámicas, por lo que puede aumentar y disminuir su tamaño en tiempo de ejecución

# Listas enlazadas

## Ventajas y desventajas respecto a los arreglos

- ▶  Son dinámicas, por lo que puede aumentar y disminuir su tamaño en tiempo de ejecución
- ▶  Adecuadas cuando no se sabe la cantidad de elementos a guardar en la estructura

# Listas enlazadas

## Ventajas y desventajas respecto a los arreglos

- ▶ 👍 Son dinámicas, por lo que puede aumentar y disminuir su tamaño en tiempo de ejecución
- ▶ 👍 Adecuadas cuando no se sabe la cantidad de elementos a guardar en la estructura
- ▶ 👎 No existe un índice por lo que no se puede acceder a un elemento de forma aleatoria

# Listas enlazadas

## Ventajas y desventajas respecto a los arreglos

- ▶ 👍 Son dinámicas, por lo que puede aumentar y disminuir su tamaño en tiempo de ejecución
- ▶ 👍 Adecuadas cuando no se sabe la cantidad de elementos a guardar en la estructura
- ▶ 👎 No existe un índice por lo que no se puede acceder a un elemento de forma aleatoria
- ▶ 👎 Necesitan más espacio en memoria para incluir el puntero

# Listas enlazadas

## Ventajas y desventajas respecto a los arreglos

- ▶ 👍 Son dinámicas, por lo que puede aumentar y disminuir su tamaño en tiempo de ejecución
- ▶ 👍 Adecuadas cuando no se sabe la cantidad de elementos a guardar en la estructura
- ▶ 👎 No existe un índice por lo que no se puede acceder a un elemento de forma aleatoria
- ▶ 👎 Necesitan más espacio en memoria para incluir el puntero

Normalmente, los nodos de las listas enlazadas no están almacenados en memoria en forma contigua. Sin embargo, lógicamente, los nodos de una lista enlazada aparecen como contiguos.

# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.

# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio

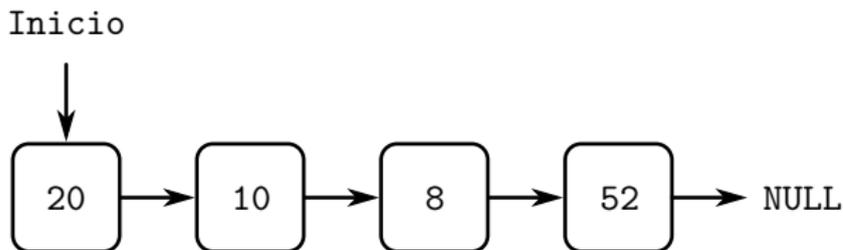
# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

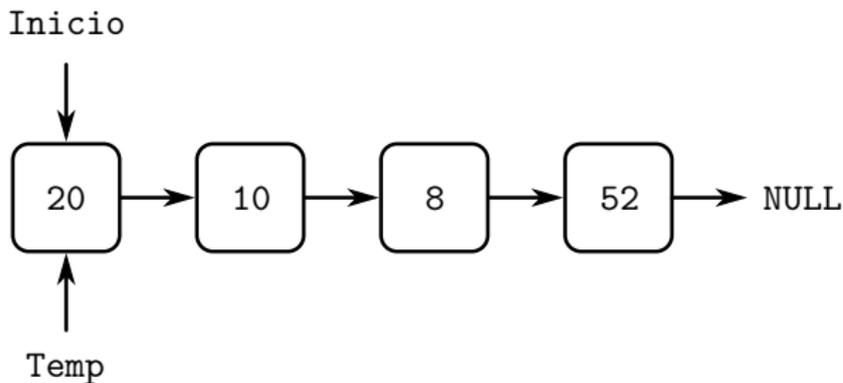
1. Puntero temporal apuntando a la cabeza de la lista (Temp=Inicio)



# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

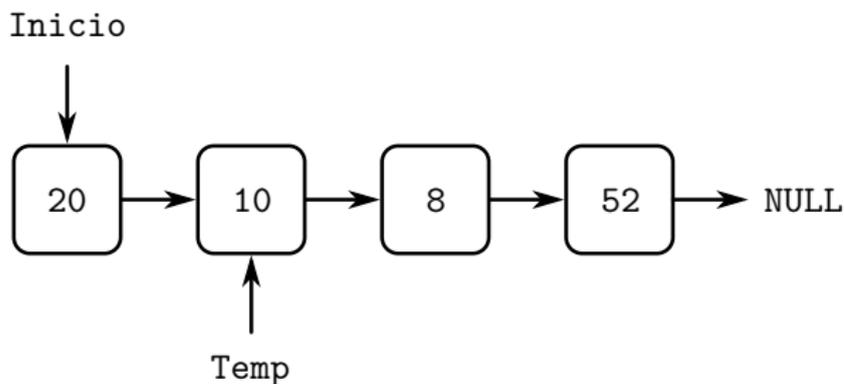
1. Puntero temporal apuntando a la cabeza de la lista (Temp=Inicio)



# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

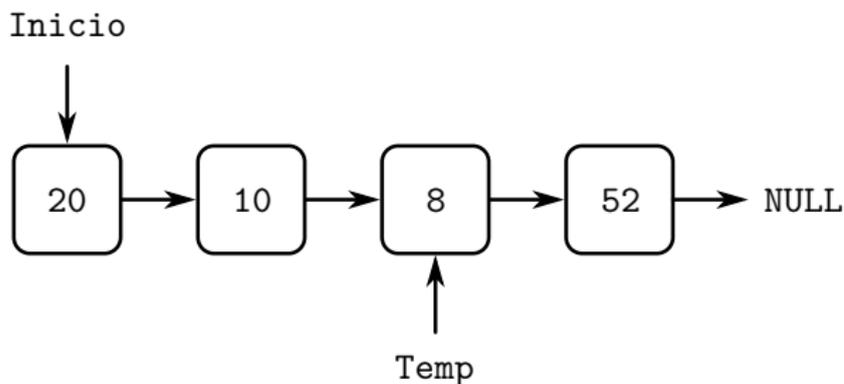
1. Puntero temporal apuntando a la cabeza de la lista ( $Temp=Inicio$ )
2. Ir reasignando el puntero temporal al enlace del nodo actual



# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

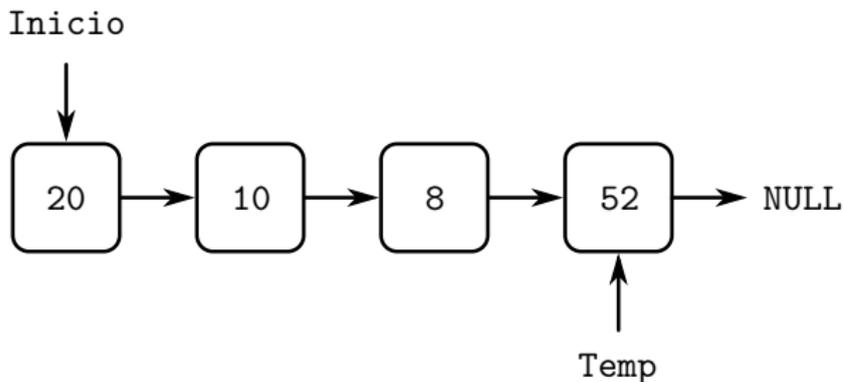
1. Puntero temporal apuntando a la cabeza de la lista ( $Temp=Inicio$ )
2. Ir reasignando el puntero temporal al enlace del nodo actual



# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

1. Puntero temporal apuntando a la cabeza de la lista ( $\text{Temp}=\text{Inicio}$ )
2. Ir reasignando el puntero temporal al enlace del nodo actual



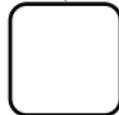
# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Lista vacía:

1. Crear un nuevo nodo apuntado por Inicio

Inicio

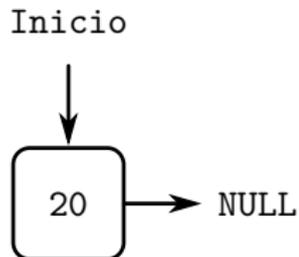


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Lista vacía:

1. Crear un nuevo nodo apuntado por Inicio
2. Almacenar datos y asignar enlace a NULL

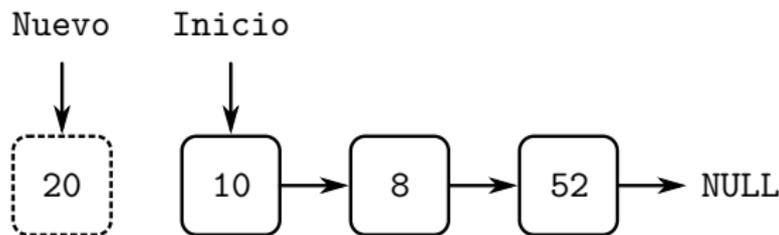


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al principio (prepend):

1. Crear un nuevo nodo y almacenar datos

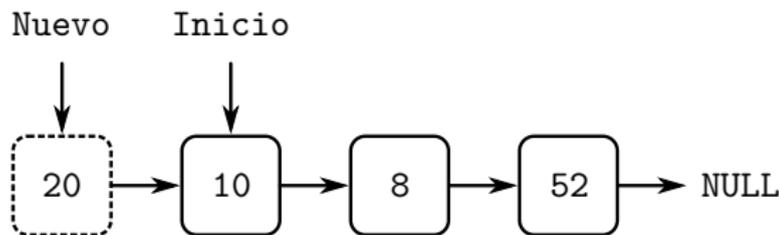


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al principio (prepend):

1. Crear un nuevo nodo y almacenar datos
2. Enlace del nuevo nodo apunta al inicio

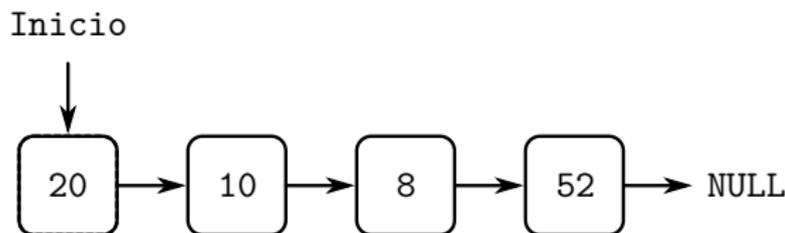


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al principio (prepend):

1. Crear un nuevo nodo y almacenar datos
2. Enlace del nuevo nodo apunta al inicio
3. Actualizar el puntero al inicio de la lista (**Inicio=Nuevo**)

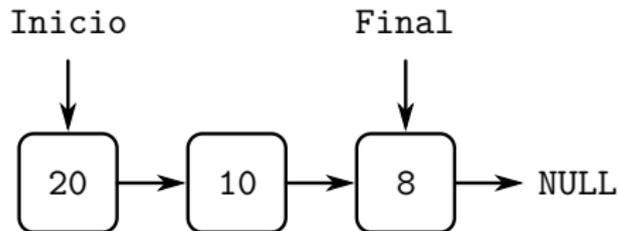


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al final (append):

1. Recorrer la lista hasta el último nodo (enlace: NULL)

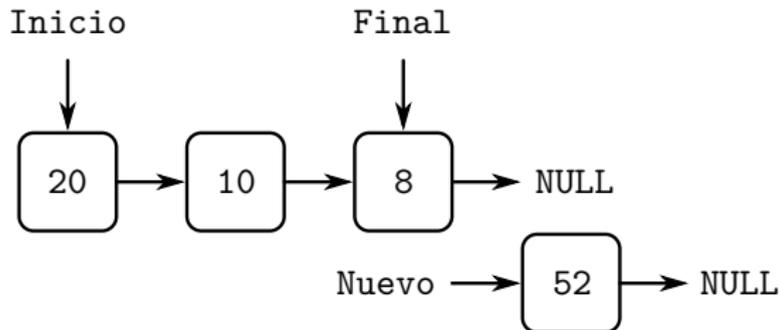


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al final (append):

1. Recorrer la lista hasta el último nodo (enlace: NULL)
2. Crear un nuevo nodo, almacenar datos y poner enlace a NULL

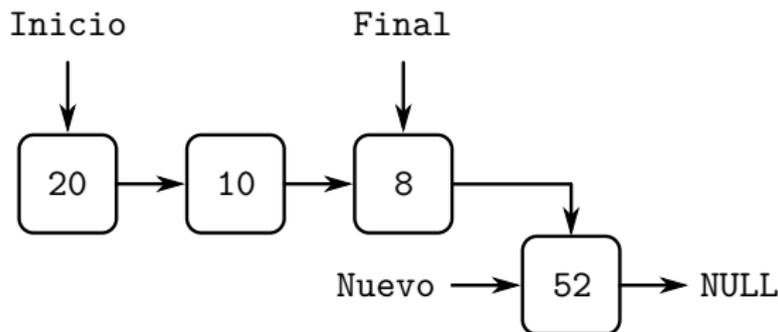


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al final (append):

1. Recorrer la lista hasta el último nodo (enlace: NULL)
2. Crear un nuevo nodo, almacenar datos y poner enlace a NULL
3. Último enlace apunta al nuevo nodo (enlace: Nuevo)

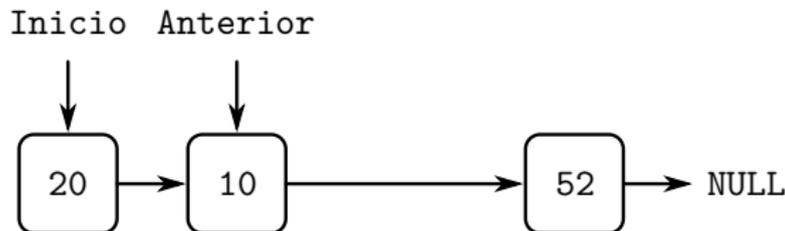


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al medio:

1. Recorrer la lista hasta el nodo anterior

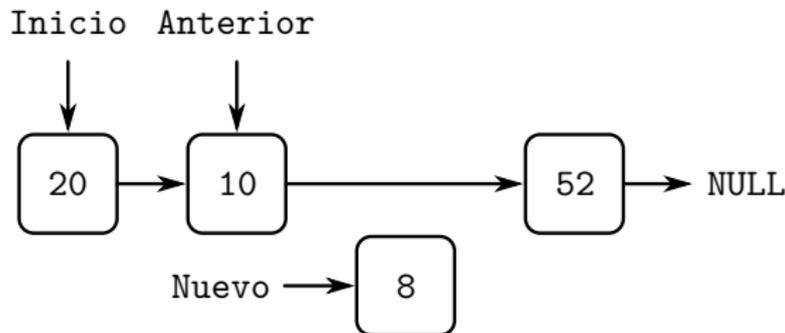


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al medio:

1. Recorrer la lista hasta el nodo anterior
2. Crear nuevo nodo y almacenar datos

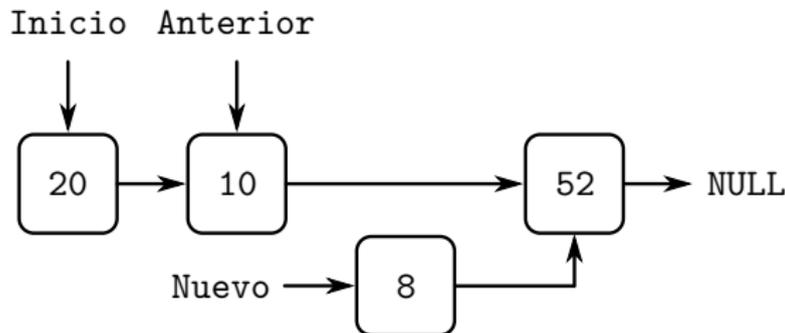


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al medio:

1. Recorrer la lista hasta el nodo anterior
2. Crear nuevo nodo y almacenar datos
3. Enlace del nuevo nodo apunta al nodo siguiente (enlace de anterior)

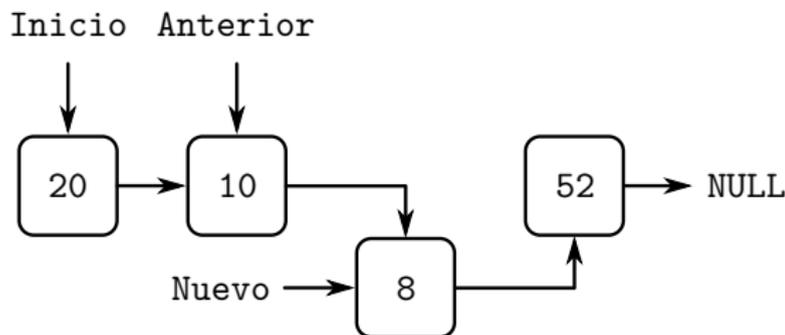


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Agregar al medio:

1. Recorrer la lista hasta el nodo anterior
2. Crear nuevo nodo y almacenar datos
3. Enlace del nuevo nodo apunta al nodo siguiente (enlace de anterior)
4. Actualizar enlace del nodo anterior al nuevo nodo creado

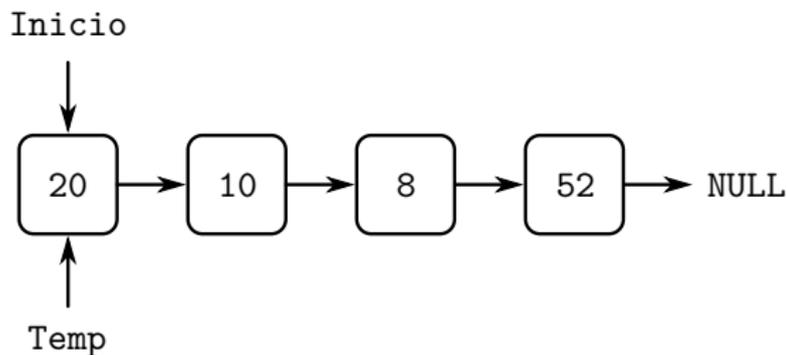


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar el primer nodo:

1. Puntero temporal al primer elemento (Temp=Inicio)

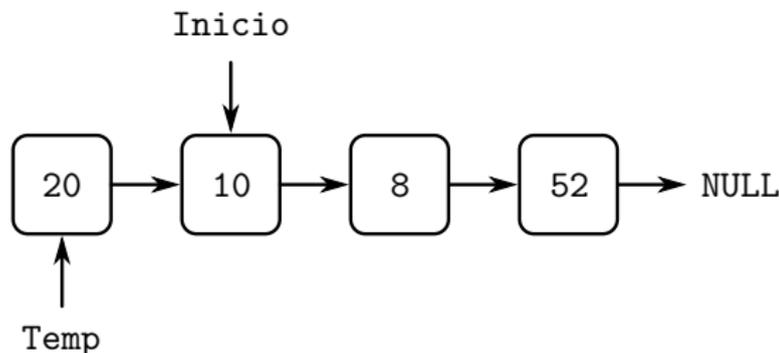


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar el primer nodo:

1. Puntero temporal al primer elemento ( $\text{Temp}=\text{Inicio}$ )
2. Hacer que  $\text{Inicio}$  apunte al siguiente (enlace)

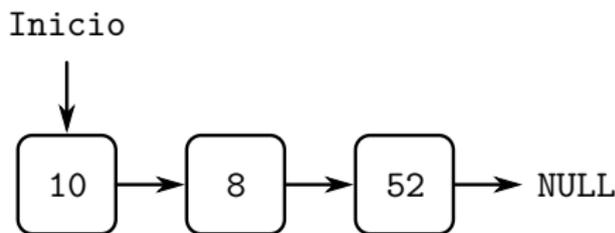


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar el primer nodo:

1. Puntero temporal al primer elemento ( $\text{Temp}=\text{Inicio}$ )
2. Hacer que **Inicio** apunte al siguiente (enlace)
3. Eliminar de memoria primer nodo (apuntado por temporal)

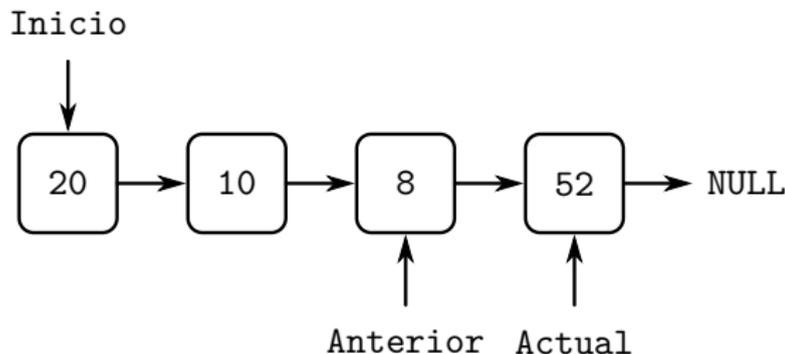


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar el último nodo:

1. Recorrer la lista hasta último nodo (Actual)

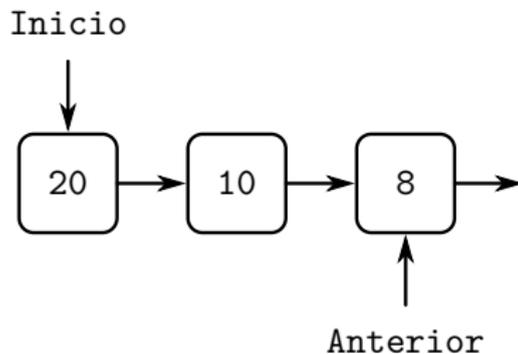


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar el último nodo:

1. Recorrer la lista hasta último nodo (**Actual**)
2. Eliminar el nodo con el puntero **Actual**

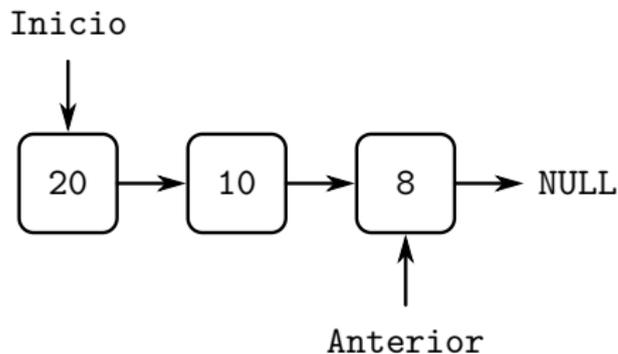


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar el último nodo:

1. Recorrer la lista hasta último nodo (**Actual**)
2. Eliminar el nodo con el puntero **Actual**
3. Enlace del último nodo a **NULL**

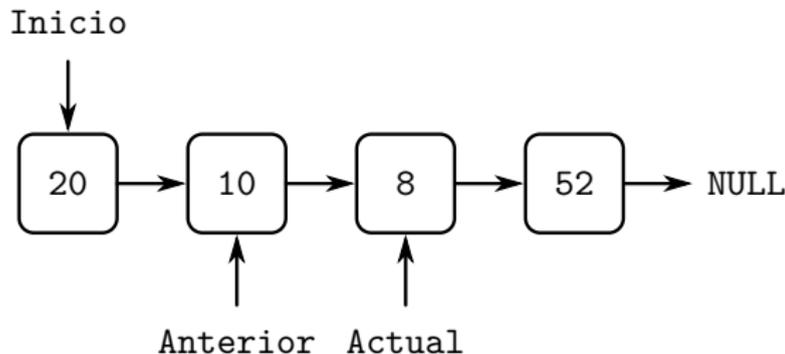


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar un nodo del medio:

1. Recorrer la lista hasta el nodo a eliminar

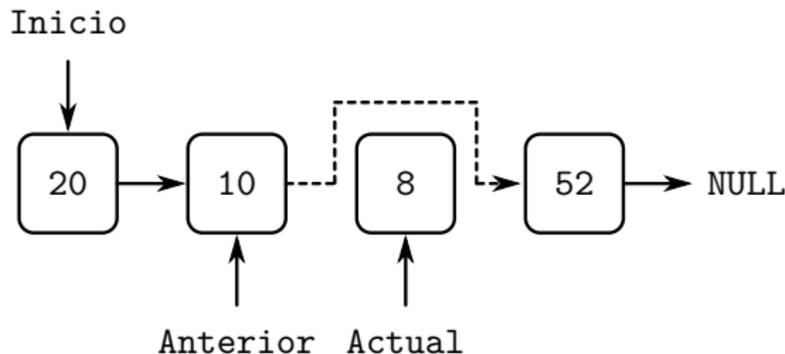


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar un nodo del medio:

1. Recorrer la lista hasta el nodo a eliminar
2. Apuntar enlace del nodo **Anterior** al nodo siguiente

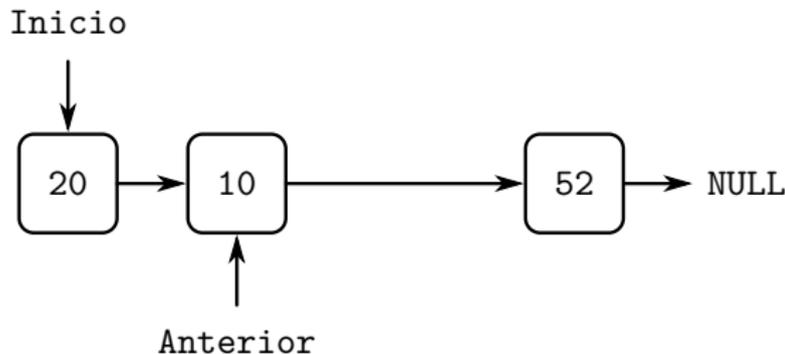


# Operaciones con listas enlazadas

- ▶ **Recorrer la lista:** p.e. para imprimir la lista, buscar un elemento, etc.
- ▶ **Insertar un nodo:** lista vacía, agregar al principio, al final o en medio
- ▶ **Eliminar un nodo:** primer nodo, último o del medio

Eliminar un nodo del medio:

1. Recorrer la lista hasta el nodo a eliminar
2. Apuntar enlace del nodo **Anterior** al nodo siguiente
3. Eliminar el nodo con el puntero **Actual**



# Listas enlazadas – ejemplo D&D Fig.12.3

Se insertan caracteres en la lista en orden alfabético.

```
La lista es:  
A --> B --> C --> E --> NULL
```

# Listas enlazadas – ejemplo D&D Fig.12.3

Se insertan caracteres en la lista en orden alfabético.

```
La lista es:  
A --> B --> C --> E --> NULL
```

Definición de tipo y prototipos de funciones:

---

```
/* Estructura auto-referenciada */  
struct nodoLista {  
    char dato; /* Cada nodoLista contiene un caracter */  
    struct nodoLista *ptrSig; /* Apuntador al siguiente nodo */  
};
```

---

# Listas enlazadas – ejemplo D&D Fig.12.3

Se insertan caracteres en la lista en orden alfabético.

```
La lista es:  
A --> B --> C --> E --> NULL
```

Definición de tipo y prototipos de funciones:

---

```
/* Estructura auto-referenciada */  
struct nodoLista {  
    char dato; /* Cada nodoLista contiene un caracter */  
    struct nodoLista *ptrSig; /* Apuntador al siguiente nodo */  
};  
  
typedef struct nodoLista NodoLista; /* Sinónimo para la estructura nodoLista */  
typedef NodoLista *ptrNodoLista; /* Sinónimo para el puntero de nodoLista */
```

---

## Listas enlazadas – ejemplo D&D Fig.12.3

Se insertan caracteres en la lista en orden alfabético.

```
La lista es:  
A --> B --> C --> E --> NULL
```

Definición de tipo y prototipos de funciones:

---

```
/* Estructura auto-referenciada */  
struct nodoLista {  
    char dato; /* Cada nodoLista contiene un caracter */  
    struct nodoLista *ptrSig; /* Apuntador al siguiente nodo */  
};  
  
typedef struct nodoLista NodoLista; /* Sinónimo para la estructura nodoLista */  
typedef NodoLista *ptrNodoLista; /* Sinónimo para el puntero de nodoLista */  
  
/* Prototipos */  
void insertar(ptrNodoLista * , char );  
char eliminar(ptrNodoLista * , char );  
int estaVacia(ptrNodoLista );  
void imprimeLista(ptrNodoLista );  
void instrucciones(void);
```

---

# Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

---

```
ptrNodoLista ptrInicial = NULL;  
/* Se cargan valores en la lista */  
  
/* Agregar el valor 'C' */  
insertar(&ptrInicial, elemento);
```

---



# Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

---

```
ptrNodoLista ptrInicial = NULL;  
/* Se cargan valores en la lista */  
  
/* Agregar el valor 'C' */  
insertar(&ptrInicial, elemento);
```

---

---

```
void insertar(ptrNodoLista *ptrS,  
             char elemento) {  
    /* Al modificar *ptrS se modifica  
       ptrInicial de la función main */  
}
```

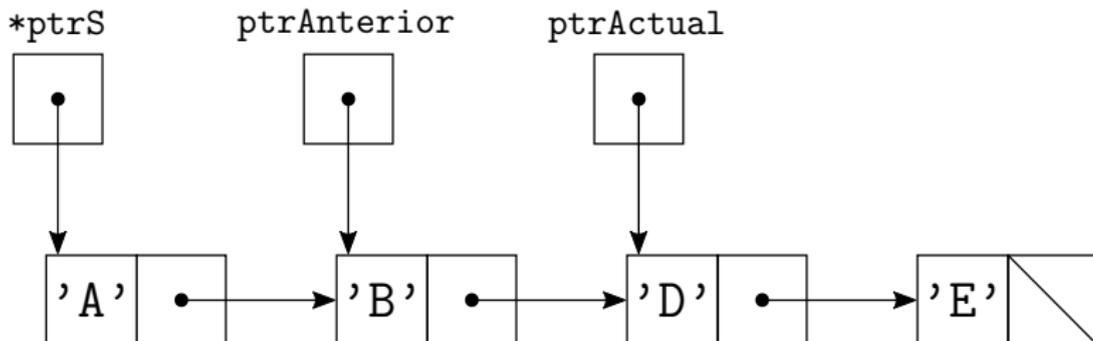
---



# Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

```
ptrNodoLista ptrInicial = NULL;  
/* Se cargan valores en la lista */  
  
/* Agregar el valor 'C' */  
insertar(&ptrInicial, elemento);
```

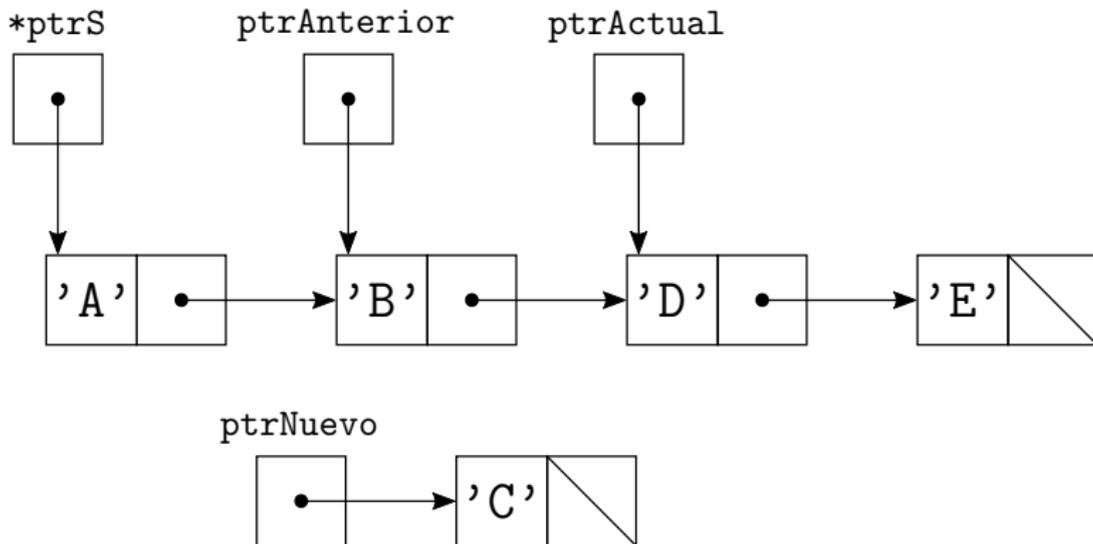
```
void insertar(ptrNodoLista *ptrS,  
             char elemento) {  
    /* Al modificar *ptrS se modifica  
       ptrInicial de la función main */  
}
```



# Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

```
ptrNodoLista ptrInicial = NULL;  
/* Se cargan valores en la lista */  
  
/* Agregar el valor 'C' */  
insertar(&ptrInicial, elemento);
```

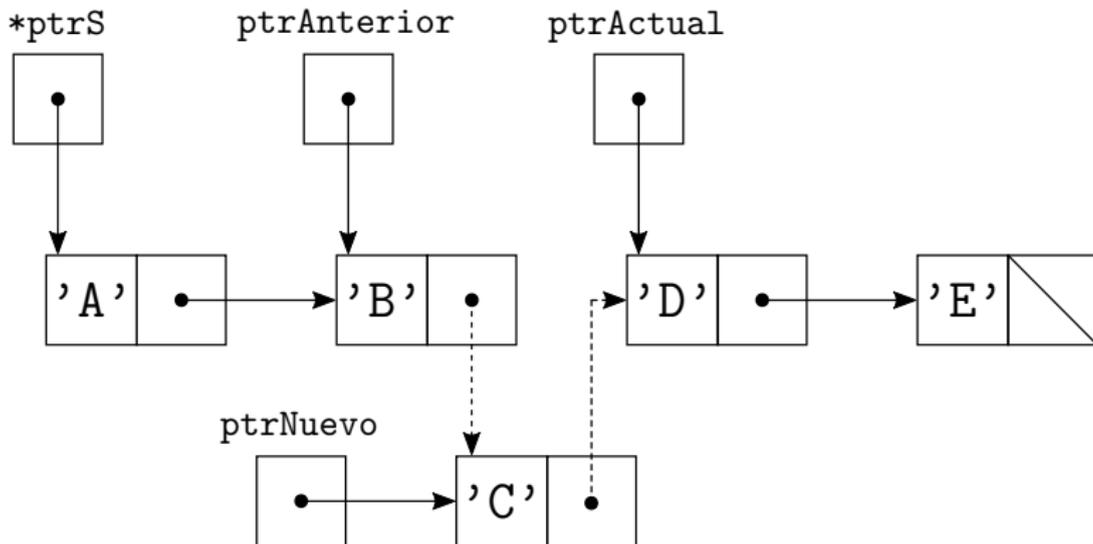
```
void insertar(ptrNodoLista *ptrS,  
             char elemento) {  
    /* Al modificar *ptrS se modifica  
       ptrInicial de la función main */  
}
```



# Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

```
ptrNodoLista ptrInicial = NULL;  
/* Se cargan valores en la lista */  
  
/* Agregar el valor 'C' */  
insertar(&ptrInicial, elemento);
```

```
void insertar(ptrNodoLista *ptrS,  
             char elemento) {  
    /* Al modificar *ptrS se modifica  
       ptrInicial de la función main */  
}
```



# Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

Dentro de la función `insertar()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL` (`ptrNuevo->ptrSig`)

# Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

Dentro de la función `insertar()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL` (`ptrNuevo->ptrSig`)
2. Inicializa `ptrAnterior` a `NULL` y `ptrActual` a `*ptrS`

## Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

Dentro de la función `insertar()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL` (`ptrNuevo->ptrSig`)
2. Inicializa `ptrAnterior` a `NULL` y `ptrActual` a `*ptrS`
3. En tanto `ptrActual` no sea `NULL` y el valor a insertar sea mayor que `ptrActual->dato` se van moviendo los punteros. Determina el punto de inserción en la lista.

## Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

Dentro de la función `insertar()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL` (`ptrNuevo->ptrSig`)
2. Inicializa `ptrAnterior` a `NULL` y `ptrActual` a `*ptrS`
3. En tanto `ptrActual` no sea `NULL` y el valor a insertar sea mayor que `ptrActual->dato` se van moviendo los punteros. Determina el punto de inserción en la lista.
4. Inserta en nuevo nodo
  - ▶ Si `ptrAnterior` es `NULL` el nuevo nodo es el primero de la lista

## Ejemplo D&D Fig.12.3 – Pasos para insertar nodo

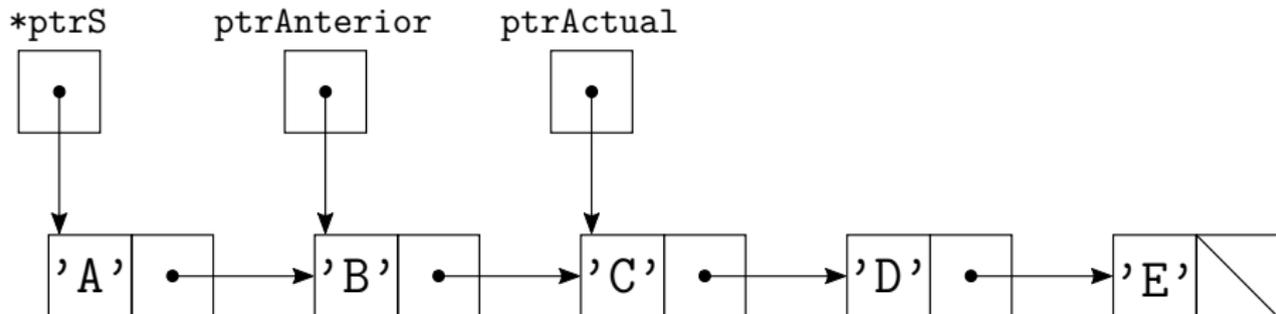
Dentro de la función `insertar()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL` (`ptrNuevo->ptrSig`)
2. Inicializa `ptrAnterior` a `NULL` y `ptrActual` a `*ptrS`
3. En tanto `ptrActual` no sea `NULL` y el valor a insertar sea mayor que `ptrActual->dato` se van moviendo los punteros. Determina el punto de inserción en la lista.
4. Inserta en nuevo nodo
  - ▶ Si `ptrAnterior` es `NULL` el nuevo nodo es el primero de la lista
  - ▶ Si no, el nuevo nodo se inserta en su lugar, y ordena los punteros

# Ejemplo D&D Fig.12.3 – Pasos para eliminar nodo

```
ptrNodoLista ptrInicial = NULL;  
/* Se cargan valores en la lista */  
  
/* Agregar el valor 'C' */  
eliminar(&ptrInicial, elemento);
```

```
char eliminar(ptrNodoLista *ptrS,  
             char valor) {  
    /* Al modificar *ptrS se modifica  
       ptrInicial de la función main */  
}
```

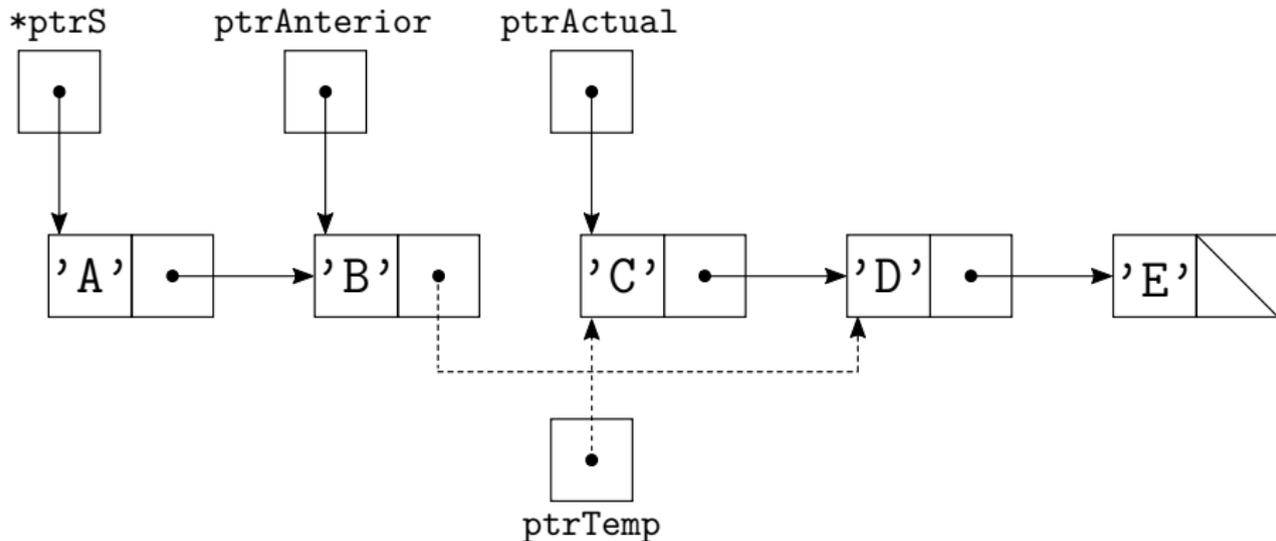


# Ejemplo D&D Fig.12.3 – Pasos para eliminar nodo

```
ptrNodoLista ptrInicial = NULL;
/* Se cargan valores en la lista */

/* Agregar el valor 'C' */
eliminar(&ptrInicial, elemento);
```

```
char eliminar(ptrNodoLista *ptrS,
              char valor) {
/* Al modificar *ptrS se modifica
   ptrInicial de la función main */
}
```



## Ejemplo D&D Fig.12.3 – Pasos para eliminar nodo

Dentro de la función `eliminar()`:

1. Si el caracter es el primero de la lista
  - ▶ Asigna `*ptrS` a `ptrTemp`
  - ▶ Asigna `(*ptrS)->ptrSig` a `*ptrS`
  - ▶ Libera la memoria usando `ptrTemp`
2. Si no, inicializa `ptrAnterior` con `*ptrS` y `ptrActual` con `(*ptrS)->ptrSig`
3. Mientras `ptrActual` no sea `NULL` y el valor a borrar no sea `ptrActual->dato` reasigna los punteros. Esto ubica el caracter a borrar.
4. Elimina el caracter
  - ▶ Si `ptrActual` no es `NULL`, asigna `ptrActual` a `ptrTemp` y `ptrActual->ptrSig` a `ptrAnterior->ptrSig`, libera el nodo apuntado por `ptrTemp`, y regresa el caracter borrado
  - ▶ Si `ptrActual` es `NULL` regresa el caracter nulo (`'\0'`) que indica que no se encontró el caracter a borrar



# Clases de listas enlazadas

- ▶ Las listas anteriores se denominan *listas enlazadas simples*

# Clases de listas enlazadas

- ▶ Las listas anteriores se denominan *listas enlazadas simples*
- ▶ También existen *listas doblemente enlazadas* en las que cada elemento tiene dos punteros, uno al elemento anterior y otro al posterior

# Clases de listas enlazadas

- ▶ Las listas anteriores se denominan *listas enlazadas simples*
- ▶ También existen *listas doblemente enlazadas* en las que cada elemento tiene dos punteros, uno al elemento anterior y otro al posterior
- ▶ Una lista doblemente enlazada se puede recorrer en ambos sentidos

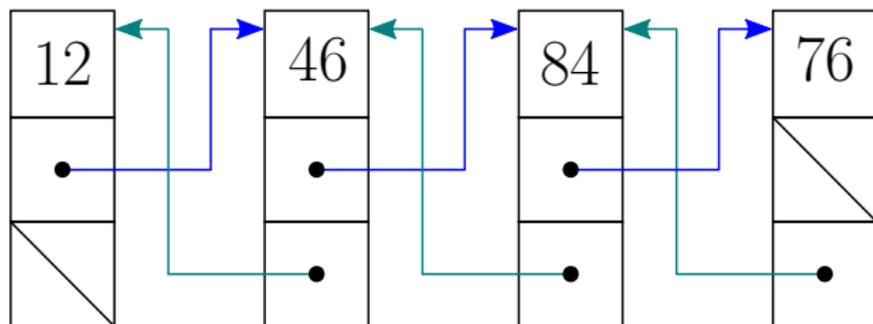
# Clases de listas enlazadas

- ▶ Las listas anteriores se denominan *listas enlazadas simples*
- ▶ También existen *listas doblemente enlazadas* en las que cada elemento tiene dos punteros, uno al elemento anterior y otro al posterior
- ▶ Una lista doblemente enlazada se puede recorrer en ambos sentidos
- ▶ Si los extremos se apuntan entre sí se tiene un *anillo lógico*

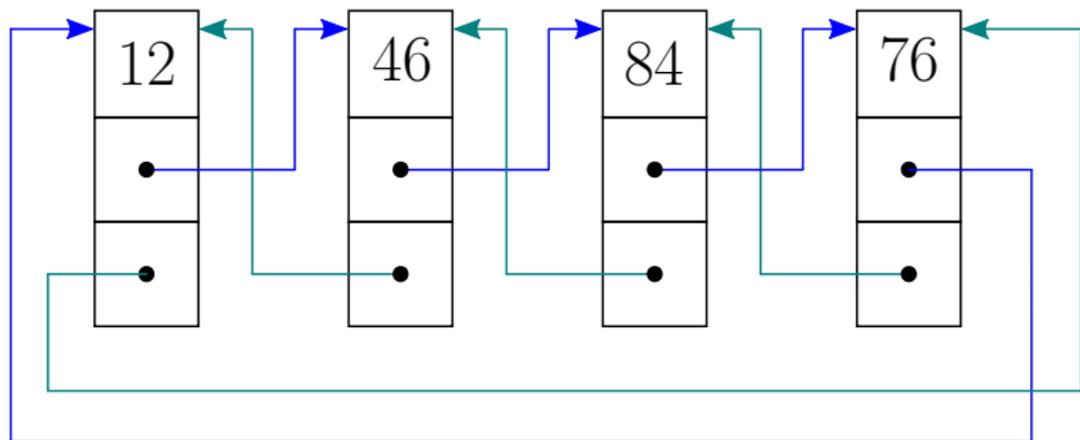
# Clases de listas enlazadas

- ▶ Las listas anteriores se denominan *listas enlazadas simples*
- ▶ También existen *listas doblemente enlazadas* en las que cada elemento tiene dos punteros, uno al elemento anterior y otro al posterior
- ▶ Una lista doblemente enlazada se puede recorrer en ambos sentidos
- ▶ Si los extremos se apuntan entre sí se tiene un *anillo lógico*
- ▶ Una lista de anillo lógico se puede recorrer de forma circular en ambos sentidos

# Ejemplo de lista doblemente enlazada



# Ejemplo de lista doblemente enlazada





# Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

# Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- ▶ Se referencia a una pila mediante un puntero al elemento superior de la misma

# Pilas

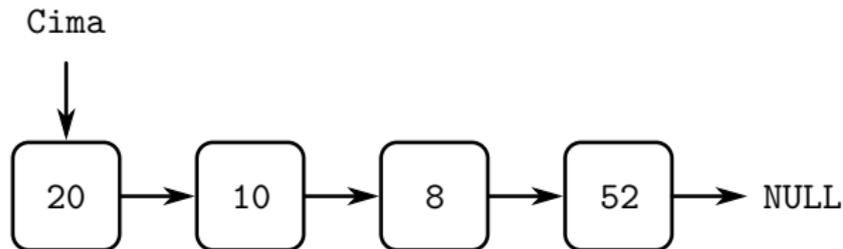
Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- ▶ Se referencia a una pila mediante un puntero al elemento superior de la misma
- ▶ Hay que definir a NULL el miembro de enlace del último nodo para indicar que se trata de la parte inferior de la pila

# Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

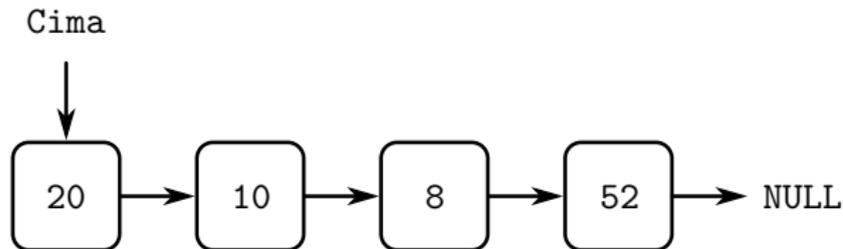
- ▶ Se referencia a una pila mediante un puntero al elemento superior de la misma
- ▶ Hay que definir a NULL el miembro de enlace del último nodo para indicar que se trata de la parte inferior de la pila



# Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- ▶ Se referencia a una pila mediante un puntero al elemento superior de la misma
- ▶ Hay que definir a NULL el miembro de enlace del último nodo para indicar que se trata de la parte inferior de la pila

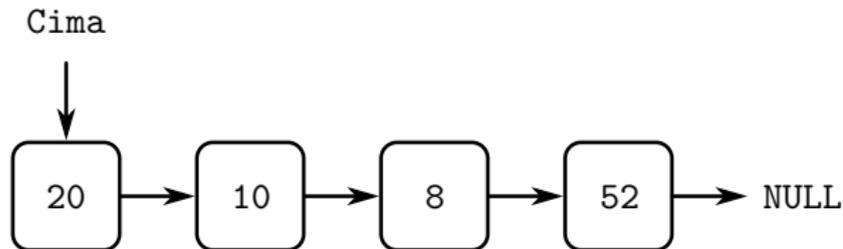


- ▶ Es similar a una lista pero con operaciones restringidas

# Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- ▶ Se referencia a una pila mediante un puntero al elemento superior de la misma
- ▶ Hay que definir a NULL el miembro de enlace del último nodo para indicar que se trata de la parte inferior de la pila



- ▶ Es similar a una lista pero con operaciones restringidas
- ▶ Se agregan y extraen elementos desde la Cima

# Operaciones con una Pila

- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`

# Operaciones con una Pila

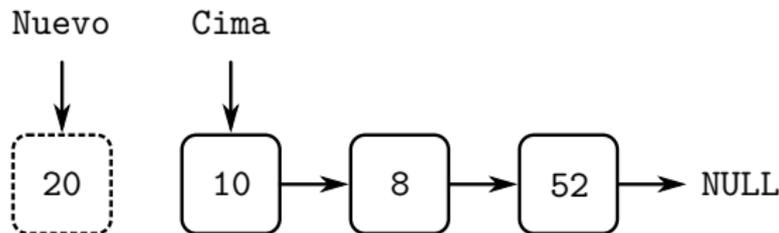
- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`
- ▶ **Extraer un elemento:** extrae el valor de la cima. Función `pop()`

# Operaciones con una Pila

- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`
- ▶ **Extraer un elemento:** extrae el valor de la cima. Función `pop()`

Similar a “agregar al inicio” en una lista:

1. Crear un nuevo nodo y almacenar datos

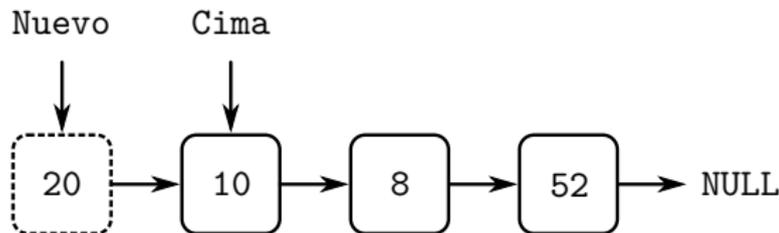


# Operaciones con una Pila

- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`
- ▶ **Extraer un elemento:** extrae el valor de la cima. Función `pop()`

Similar a “agregar al inicio” en una lista:

1. Crear un nuevo nodo y almacenar datos
2. Enlace del nuevo nodo apunta a la cima

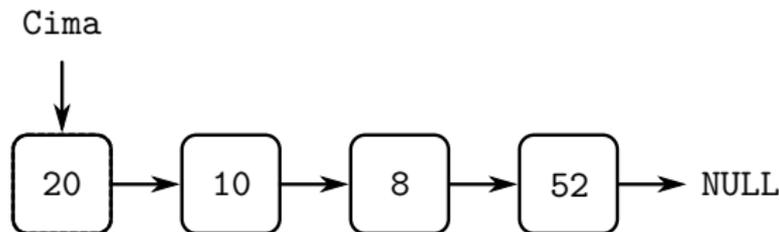


# Operaciones con una Pila

- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`
- ▶ **Extraer un elemento:** extrae el valor de la cima. Función `pop()`

Similar a “agregar al inicio” en una lista:

1. Crear un nuevo nodo y almacenar datos
2. Enlace del nuevo nodo apunta a la cima
3. Actualizar el puntero a la cima de la pila (`Cima=Nuevo`)

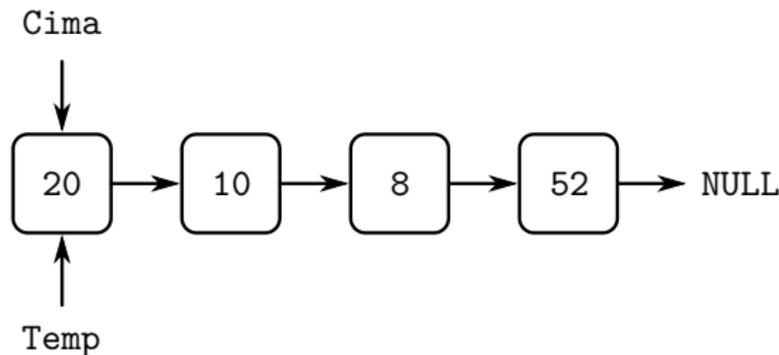


# Operaciones con una Pila

- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`
- ▶ **Extraer un elemento:** extrae el valor de la cima. Función `pop()`

Similar a “eliminar el primer nodo” en una lista:

1. Puntero temporal al nodo de la cima (`Temp=Cima`)

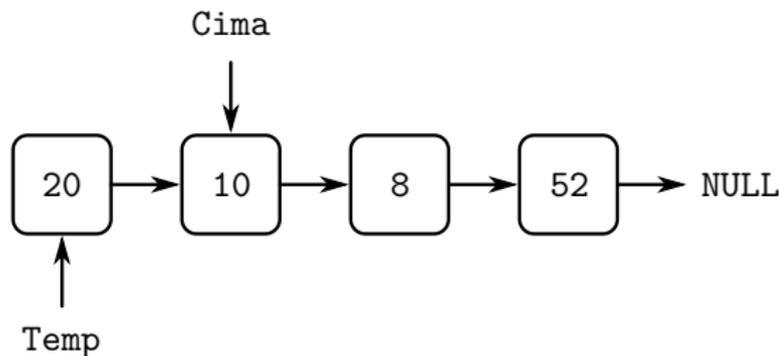


# Operaciones con una Pila

- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`
- ▶ **Extraer un elemento:** extrae el valor de la cima. Función `pop()`

Similar a “eliminar el primer nodo” en una lista:

1. Puntero temporal al nodo de la cima (`Temp=Cima`)
2. Hacer que `Cima` apunte al siguiente (enlace)

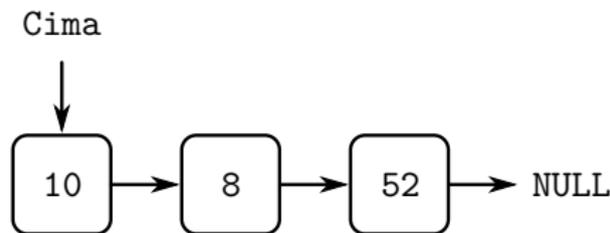


# Operaciones con una Pila

- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`
- ▶ **Extraer un elemento:** extrae el valor de la cima. Función `pop()`

Similar a “eliminar el primer nodo” en una lista:

1. Puntero temporal al nodo de la cima (`Temp=Cima`)
2. Hacer que `Cima` apunte al siguiente (enlace)
3. Eliminar de memoria nodo cima (apuntado por temporal)

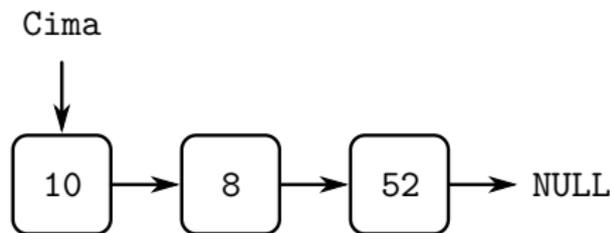


# Operaciones con una Pila

- ▶ **Agregar un elemento:** se agrega en la cima. Función `push()`
- ▶ **Extraer un elemento:** extrae el valor de la cima. Función `pop()`

Similar a “eliminar el primer nodo” en una lista:

1. Puntero temporal al nodo de la cima (`Temp=Cima`)
2. Hacer que `Cima` apunte al siguiente (enlace)
3. Eliminar de memoria nodo cima (apuntado por temporal)
4. Devolver el valor obtenido previamente guardado



# Pila – ejemplo D&D Fig.12.8

```
Introduzca su elección:
  1 para empujar un valor dentro de la pila
  2 para sacar un valor de la pila
  3 para terminar el programa
. . .
. . .
? 1
Introduzca un entero: 4
La pila es:
4 --> 6 --> 5 --> NULL

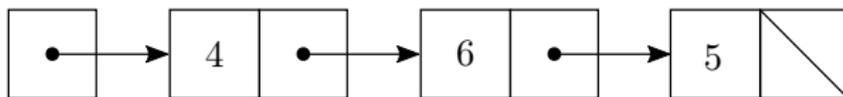
? 2
El valor sacado es 4.
La pila es:
6 --> 5 --> NULL
```

# Pila – ejemplo D&D Fig.12.8

```
Introduzca su elección:
  1 para empujar un valor dentro de la pila
  2 para sacar un valor de la pila
  3 para terminar el programa
. . .
. . .
? 1
Introduzca un entero: 4
La pila es:
4 --> 6 --> 5 --> NULL

? 2
El valor sacado es 4.
La pila es:
6 --> 5 --> NULL
```

ptrPila



# Pila – ejemplo D&D Fig.12.8

Definición de tipo y prototipos de funciones:

---

```
/* Estructura auto-referenciada */
struct nodoPila {
    int dato; /* Define un dato como int */
    struct nodoPila *ptrSig; /* Apuntador a nodoPila */
};

typedef struct nodoPila NodoPila; /* Sinónimo de la estructura nodoPila */
typedef NodoPila *ptrNodoPila; /* Sinónimo para NodoPila */

/* Prototipos */
void push(ptrNodoPila * , int );
int pop(ptrNodoPila * );
int estaVacia(ptrNodoPila );
void imprimePila(ptrNodoPila );
void instrucciones(void);
```

---

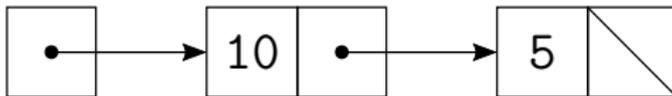
# Ejemplo D&D Fig.12.8 – Pasos para insertar nodo

```
ptrNodoPila ptrPila = NULL;  
/* Se cargan valores en la pila */  
  
/* Agregar el valor 9 */  
push(&ptrPila, valor);
```

```
void push(ptrNodoPila *ptrCima,  
         int info)  
/* Al modificar *ptrCima se modifica  
   ptrPila de la función main */  
}
```

Representación de la pila dentro de push()

\*ptrCima



ptrNuevo

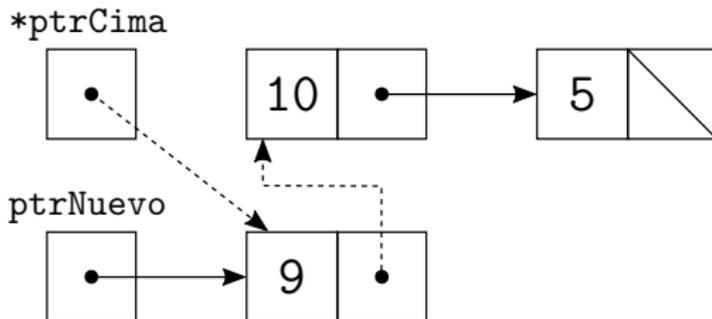


# Ejemplo D&D Fig.12.8 – Pasos para insertar nodo

```
ptrNodoPila ptrPila = NULL;  
/* Se cargan valores en la pila */  
  
/* Agregar el valor 9 */  
push(&ptrPila, valor);
```

```
void push(ptrNodoPila *ptrCima,  
         int info)  
/* Al modificar *ptrCima se modifica  
   ptrPila de la función main */  
}
```

Representación de la pila dentro de push()



# Ejemplo D&D Fig.12.8 – Pasos para insertar nodo

Dentro de la función `push()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL`

## Ejemplo D&D Fig.12.8 – Pasos para insertar nodo

Dentro de la función `push()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL`
2. Asigna el puntero de la pila `*ptrCima` a `ptrNuevo->ptrSig`. El nuevo enlace ahora apunta al nodo superior de la pila.

## Ejemplo D&D Fig.12.8 – Pasos para insertar nodo

Dentro de la función `push()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL`
2. Asigna el puntero de la pila `*ptrCima` a `ptrNuevo->ptrSig`. El nuevo enlace ahora apunta al nodo superior de la pila.
3. Asigna `ptrNuevo` a `*ptrCima`. `*ptrCima` apunta ahora a la nueva cima de la pila.

## Ejemplo D&D Fig.12.8 – Pasos para insertar nodo

Dentro de la función `push()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, almacena el nuevo dato en `ptrNuevo->dato`, y pone el enlace a `NULL`
2. Asigna el puntero de la pila `*ptrCima` a `ptrNuevo->ptrSig`. El nuevo enlace ahora apunta al nodo superior de la pila.
3. Asigna `ptrNuevo` a `*ptrCima`. `*ptrCima` apunta ahora a la nueva cima de la pila.

Las modificaciones realizadas a `*ptrCima` modifican el valor de `ptrPila` en `main()`.

# Ejemplo D&D Fig.12.8 – Pasos para extraer nodo

---

```
ptrNodoPila ptrPila = NULL;  
/* Se cargan valores en la pila */  
  
/* Extraer el valor de la cima */  
pop(&ptrPila);
```

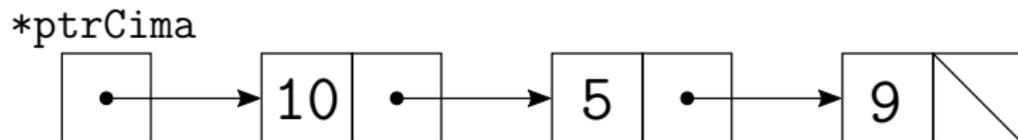
---

---

```
int pop(ptrNodoPila *ptrCima)  
/* Al modificar *ptrCima se modifica  
ptrPila de la función main */  
}
```

---

Representación de la pila dentro de pop()

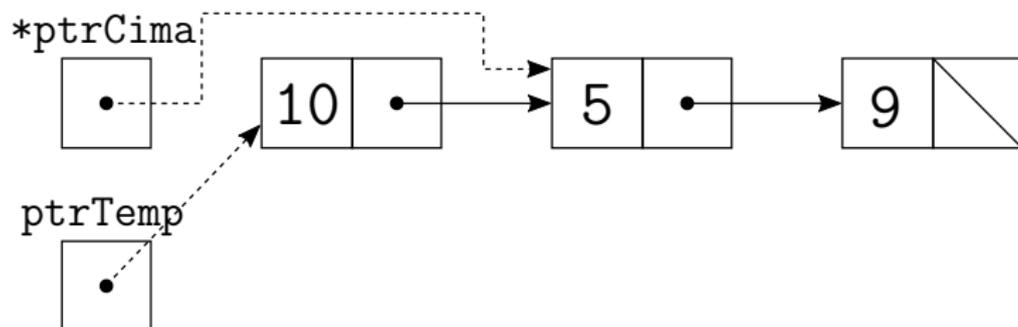


# Ejemplo D&D Fig.12.8 – Pasos para extraer nodo

```
ptrNodoPila ptrPila = NULL;  
/* Se cargan valores en la pila */  
  
/* Extraer el valor de la cima */  
pop(&ptrPila);
```

```
int pop(ptrNodoPila *ptrCima)  
/* Al modificar *ptrCima se modifica  
ptrPila de la función main */  
}
```

Representación de la pila dentro de pop()



# Ejemplo D&D Fig.12.8 – Pasos para extraer nodo

Dentro de la función `pop()`:

1. Asignar `*ptrCima` a `ptrTemp` (para luego liberar memoria)

## Ejemplo D&D Fig.12.8 – Pasos para extraer nodo

Dentro de la función `pop()`:

1. Asignar `*ptrCima` a `ptrTemp` (para luego liberar memoria)
2. Asignar `(*ptrCima)->dato` a `valorElim` (guarda el valor del nodo)

## Ejemplo D&D Fig.12.8 – Pasos para extraer nodo

Dentro de la función `pop()`:

1. Asignar `*ptrCima` a `ptrTemp` (para luego liberar memoria)
2. Asignar `(*ptrCima)->dato` a `valorElim` (guarda el valor del nodo)
3. Asignar `(*ptrCima)->ptrSig` a `*ptrCima` (nuevo nodo cima)

## Ejemplo D&D Fig.12.8 – Pasos para extraer nodo

Dentro de la función `pop()`:

1. Asignar `*ptrCima` a `ptrTemp` (para luego liberar memoria)
2. Asignar `(*ptrCima)->dato` a `valorElim` (guarda el valor del nodo)
3. Asignar `(*ptrCima)->ptrSig` a `*ptrCima` (nuevo nodo cima)
4. Liberar la memoria apuntada por `ptrTemp` con `free()`

## Ejemplo D&D Fig.12.8 – Pasos para extraer nodo

Dentro de la función `pop()`:

1. Asignar `*ptrCima` a `ptrTemp` (para luego liberar memoria)
2. Asignar `(*ptrCima)->dato` a `valorElim` (guarda el valor del nodo)
3. Asignar `(*ptrCima)->ptrSig` a `*ptrCima` (nuevo nodo cima)
4. Liberar la memoria apuntada por `ptrTemp` con `free()`
5. Regresar `valorElim` a la función llamadora (en este caso `main()`)



# Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

# Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

- ▶ Se eliminan nodos de la cola solo de la parte delantera o *cabeza* de la cola,

# Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

- ▶ Se eliminan nodos de la cola solo de la parte delantera o *cabeza* de la cola,
- ▶ y son incluidos o insertos únicamente en la *parte trasera* de la cola.

# Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

- ▶ Se eliminan nodos de la cola solo de la parte delantera o *cabeza* de la cola,
- ▶ y son incluidos o insertos únicamente en la *parte trasera* de la cola.

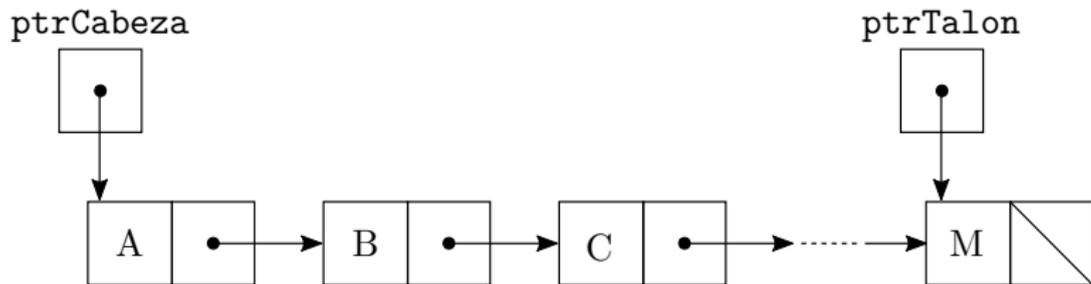
Las operaciones de insertar y retirar se conoce como **enqueue** y **dequeue**.

# Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

- ▶ Se eliminan nodos de la cola solo de la parte delantera o *cabeza* de la cola,
- ▶ y son incluidos o insertos únicamente en la *parte trasera* de la cola.

Las operaciones de insertar y retirar se conoce como *enqueue* y *dequeue*.



# Cola – ejemplo D&D Fig.12.13

```
Introduzca su elección:
  1 para ingresar un elemento a la cola
  2 para eliminar un elemento de la cola
  3 para terminar
. . .
. . .
? 1
Introduzca un caracter: C
La cola es:
A --> B --> C --> NULL

? 2
Se desenfilo A.
La cola es:
B --> C --> NULL
```

# Cola – ejemplo D&D Fig.12.13

Definición de tipo y prototipos de funciones:

---

```
/* Estructura auto-referenciada */
struct nodoCola {
    char dato; /* Define dato como un char */
    struct nodoCola *ptrSig; /* Apuntador nodoCola */
};

typedef struct nodoCola NodoCola;
typedef NodoCola *ptrNodoCola;

/* Prototipos de funciones */
void imprimeCola(ptrNodoCola );
int estaVacia(ptrNodoCola );
char dequeue(ptrNodoCola * , ptrNodoCola * );
void enqueue(ptrNodoCola * , ptrNodoCola * , char );
void instrucciones(void);
```

---

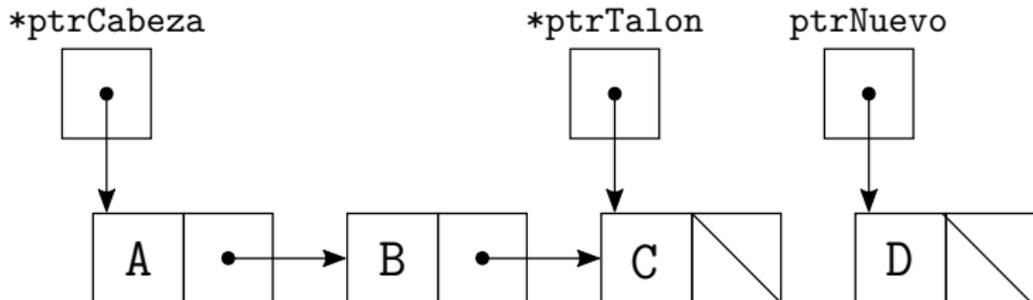
# Ejemplo D&D Fig.12.13 – Pasos para insertar nodo

```
ptrNodoCola ptrCabeza = NULL;  
ptrNodoCola ptrTalon = NULL;
```

```
/* Agregar un nodo */  
enqueue(&ptrCabeza, &ptrTalon  
        elemento);
```

```
void enqueue(ptrNodoCola *ptrCabeza,  
            ptrNodoCola *ptrTemp,  
            char valor)  
/* Modificar *ptrTalon local modifica  
ptrTalon de la función main */  
}
```

Representación de la cola dentro de enqueue()



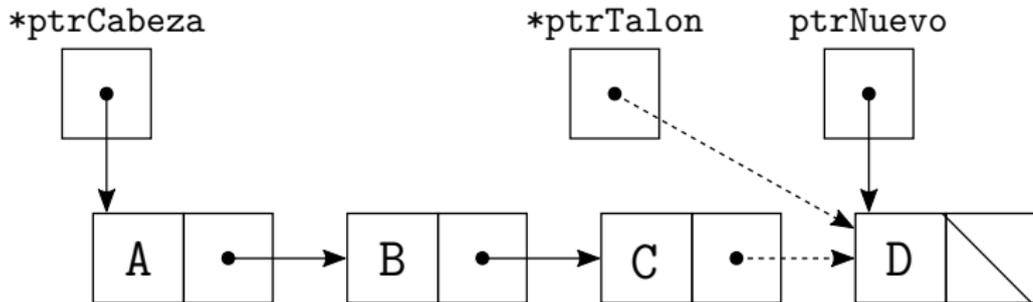
# Ejemplo D&D Fig.12.13 – Pasos para insertar nodo

```
ptrNodoCola ptrCabeza = NULL;  
ptrNodoCola ptrTalon = NULL;
```

```
/* Agregar un nodo */  
enqueue(&ptrCabeza, &ptrTalon  
        elemento);
```

```
void enqueue(ptrNodoCola *ptrCabeza,  
            ptrNodoCola *ptrTemp,  
            char valor)  
/* Modificar *ptrTalon local modifica  
ptrTalon de la función main */  
}
```

Representación de la cola dentro de enqueue()



# Ejemplo D&D Fig.12.13 – Pasos para insertar nodo

Dentro de la función `enqueue()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, guarda el valor a insertar en la cola `ptrNuevo->dato`, y el enlace `ptrNuevo->ptrSig` a `NULL`

## Ejemplo D&D Fig.12.13 – Pasos para insertar nodo

Dentro de la función `enqueue()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, guarda el valor a insertar en la cola `ptrNuevo->dato`, y el enlace `ptrNuevo->ptrSig` a `NULL`
2. Si la cola está vacía, asigna `ptrNuevo` a `*ptrCabeza`; si no, asigna `ptrNuevo` a `(*ptrTalon)->ptrSig`

## Ejemplo D&D Fig.12.13 – Pasos para insertar nodo

Dentro de la función `enqueue()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `ptrNuevo`, guarda el valor a insertar en la cola `ptrNuevo->dato`, y el enlace `ptrNuevo->ptrSig` a `NULL`
2. Si la cola está vacía, asigna `ptrNuevo` a `*ptrCabeza`; si no, asigna `ptrNuevo` a `(*ptrTalon)->ptrSig`
3. Asigna `ptrNuevo` a `*ptrTalon`

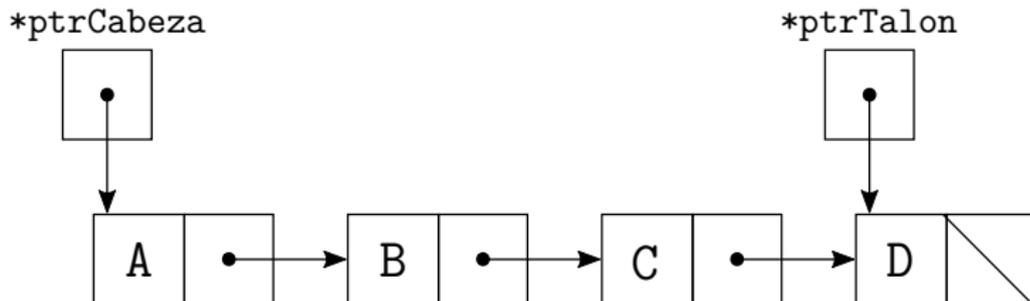
# Ejemplo D&D Fig.12.13 – Pasos para extraer nodo

```
ptrNodoCola ptrCabeza = NULL;
ptrNodoCola ptrTalon = NULL;

/* Extraer un nodo */
elemento = dequeue(&ptrCabeza,
                  &ptrTalon);
```

```
char dequeue(ptrNodoCola *ptrCabeza,
             ptrNodoCola *ptrTemp)
/* Modificar *ptrCabeza local modifica
   ptrCabeza de la función main */
}
```

Representación de la cola dentro de dequeue()

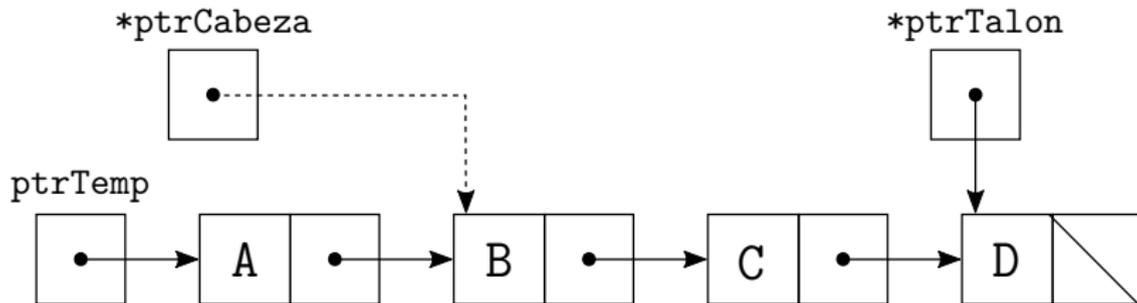


# Ejemplo D&D Fig.12.13 – Pasos para extraer nodo

```
ptrNodoCola ptrCabeza = NULL;  
ptrNodoCola ptrTalon = NULL;  
  
/* Extraer un nodo */  
elemento = dequeue(&ptrCabeza,  
                  &ptrTalon);
```

```
char dequeue(ptrNodoCola *ptrCabeza,  
            ptrNodoCola *ptrTemp)  
/* Modificar *ptrCabeza local modifica  
ptrCabeza de la función main */  
}
```

Representación de la cola dentro de dequeue()



# Ejemplo D&D Fig.12.13 – Pasos para extraer nodo

Dentro de la función `dequeue()`:

1. Asigna `(*ptrCabeza)->dato` a `valor` (valor del nodo)

# Ejemplo D&D Fig.12.13 – Pasos para extraer nodo

Dentro de la función `dequeue()`:

1. Asigna `(*ptrCabeza)->dato` a `valor` (valor del nodo)
2. Asigna `*ptrCabeza` a `ptrTemp` (para luego liberar memoria)

## Ejemplo D&D Fig.12.13 – Pasos para extraer nodo

Dentro de la función `dequeue()`:

1. Asigna `(*ptrCabeza)->dato` a `valor` (valor del nodo)
2. Asigna `*ptrCabeza` a `ptrTemp` (para luego liberar memoria)
3. Asigna `(*ptrCabeza)->ptrSig` a `*ptrCabeza` (apunta al primer nodo de la cola)

# Ejemplo D&D Fig.12.13 – Pasos para extraer nodo

Dentro de la función `dequeue()`:

1. Asigna `(*ptrCabeza)->dato` a `valor` (valor del nodo)
2. Asigna `*ptrCabeza` a `ptrTemp` (para luego liberar memoria)
3. Asigna `(*ptrCabeza)->ptrSig` a `*ptrCabeza` (apunta al primer nodo de la cola)
4. Si `*ptrCabeza` es `NULL` asigna `NULL` a `*ptrTalon`

# Ejemplo D&D Fig.12.13 – Pasos para extraer nodo

Dentro de la función `dequeue()`:

1. Asigna `(*ptrCabeza)->dato` a `valor` (valor del nodo)
2. Asigna `*ptrCabeza` a `ptrTemp` (para luego liberar memoria)
3. Asigna `(*ptrCabeza)->ptrSig` a `*ptrCabeza` (apunta al primer nodo de la cola)
4. Si `*ptrCabeza` es `NULL` asigna `NULL` a `*ptrTalon`
5. Libera la memoria apuntada por `ptrTemp` con `free()`

# Ejemplo D&D Fig.12.13 – Pasos para extraer nodo

Dentro de la función `dequeue()`:

1. Asigna `(*ptrCabeza)->dato` a `valor` (valor del nodo)
2. Asigna `*ptrCabeza` a `ptrTemp` (para luego liberar memoria)
3. Asigna `(*ptrCabeza)->ptrSig` a `*ptrCabeza` (apunta al primer nodo de la cola)
4. Si `*ptrCabeza` es `NULL` asigna `NULL` a `*ptrTalon`
5. Libera la memoria apuntada por `ptrTemp` con `free()`
6. Regresa `valor` a la función llamadora (en este caso `main()`)



# Árboles

- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)

# Árboles

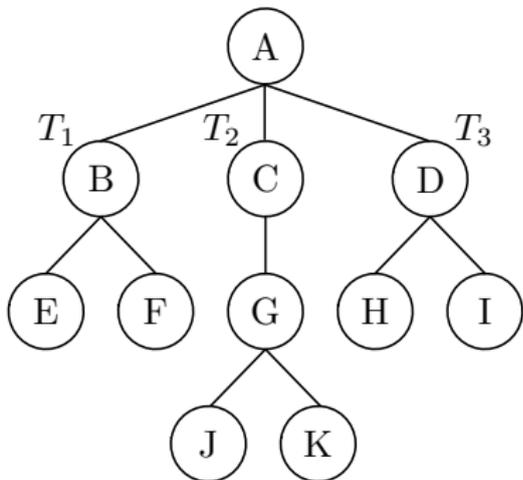
- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- ▶ Los nodos de los *árboles* contienen dos o más enlaces

# Árboles

- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- ▶ Los nodos de los *árboles* contienen dos o más enlaces
- ▶ Utilizados para guardar datos que tiene una relación de jerarquía

# Árboles

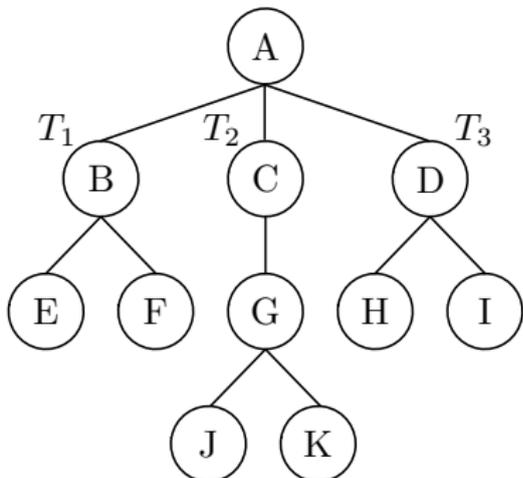
- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- ▶ Los nodos de los *árboles* contienen dos o más enlaces
- ▶ Utilizados para guardar datos que tiene una relación de jerarquía



- ▶ Nodo raíz (R): nodo superior (no tiene nodo padre). Si R es NULL el árbol está vacío

# Árboles

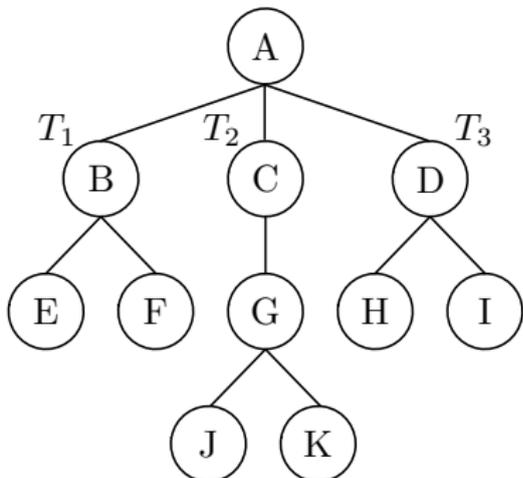
- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- ▶ Los nodos de los *árboles* contienen dos o más enlaces
- ▶ Utilizados para guardar datos que tiene una relación de jerarquía



- ▶ Nodo raíz (R): nodo superior (no tiene nodo padre). Si R es NULL el árbol está vacío
- ▶ Sub-árbol: si el nodo raíz R no es NULL,  $T_1$ ,  $T_2$  y  $T_3$  son sub-árboles de R

# Árboles

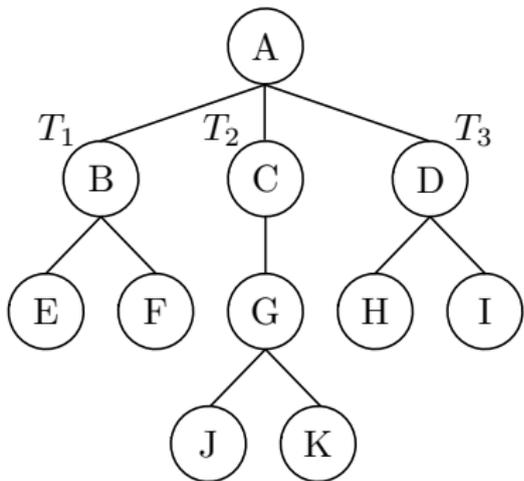
- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- ▶ Los nodos de los *árboles* contienen dos o más enlaces
- ▶ Utilizados para guardar datos que tiene una relación de jerarquía



- ▶ Nodo raíz (R): nodo superior (no tiene nodo padre). Si R es NULL el árbol está vacío
- ▶ Sub-árbol: si el nodo raíz R no es NULL,  $T_1$ ,  $T_2$  y  $T_3$  son sub-árboles de R
- ▶ Nodo hoja: nodo que no tiene hijos

# Árboles

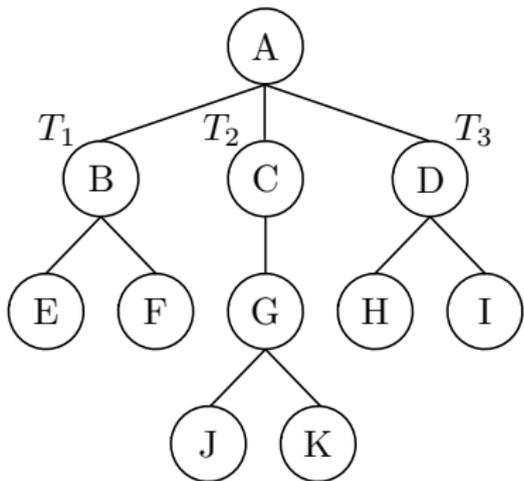
- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- ▶ Los nodos de los *árboles* contienen dos o más enlaces
- ▶ Utilizados para guardar datos que tiene una relación de jerarquía



- ▶ Nodo raíz (R): nodo superior (no tiene nodo padre). Si R es NULL el árbol está vacío
- ▶ Sub-árbol: si el nodo raíz R no es NULL,  $T_1$ ,  $T_2$  y  $T_3$  son sub-árboles de R
- ▶ Nodo hoja: nodo que no tiene hijos
- ▶ Camino: se forma por una secuencia de aristas, p.e.: el camino de A a I es: A, D, e I

# Árboles

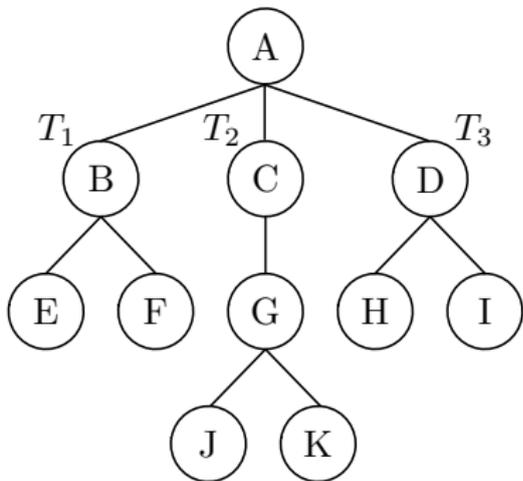
- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- ▶ Los nodos de los *árboles* contienen dos o más enlaces
- ▶ Utilizados para guardar datos que tiene una relación de jerarquía



- ▶ Nodo raíz (R): nodo superior (no tiene nodo padre). Si R es NULL el árbol está vacío
- ▶ Sub-árbol: si el nodo raíz R no es NULL,  $T_1$ ,  $T_2$  y  $T_3$  son sub-árboles de R
- ▶ Nodo hoja: nodo que no tiene hijos
- ▶ Camino: se forma por una secuencia de aristas, p.e.: el camino de A a I es: A, D, e I
- ▶ Nro de nivel: el nodo raíz tiene nivel 0 y sus hijos nivel 1, y así sucesivamente

# Árboles

- ▶ Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- ▶ Los nodos de los *árboles* contienen dos o más enlaces
- ▶ Utilizados para guardar datos que tiene una relación de jerarquía



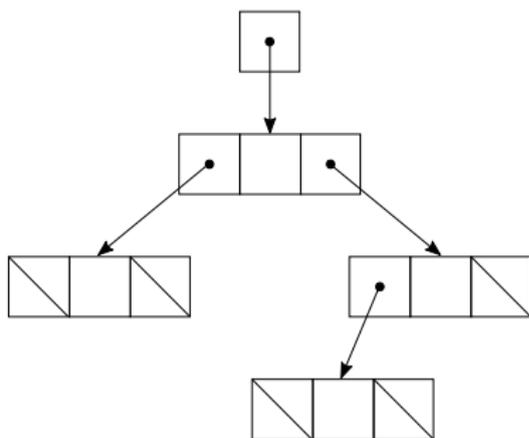
- ▶ Nodo raíz (R): nodo superior (no tiene nodo padre). Si R es NULL el árbol está vacío
- ▶ Sub-árbol: si el nodo raíz R no es NULL,  $T_1$ ,  $T_2$  y  $T_3$  son sub-árboles de R
- ▶ Nodo hoja: nodo que no tiene hijos
- ▶ Camino: se forma por una secuencia de aristas, p.e.: el camino de A a I es: A, D, e I
- ▶ Nro de nivel: el nodo raíz tiene nivel 0 y sus hijos nivel 1, y así sucesivamente
- ▶ Grado: el grado de un nodo es la cantidad de nodos hijos que tiene

# Árbol binario

Todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a NULL)

# Árbol binario

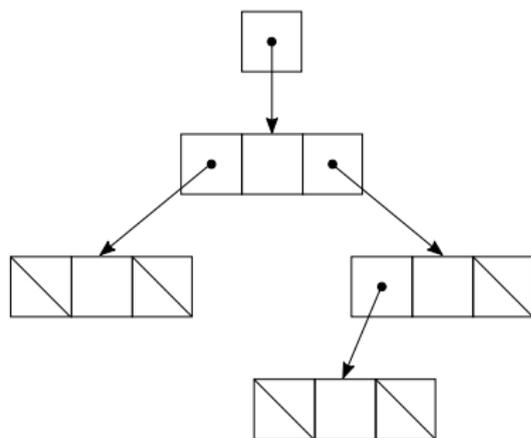
Todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a NULL)



- El *nodo raíz* es el primer nodo del árbol

# Árbol binario

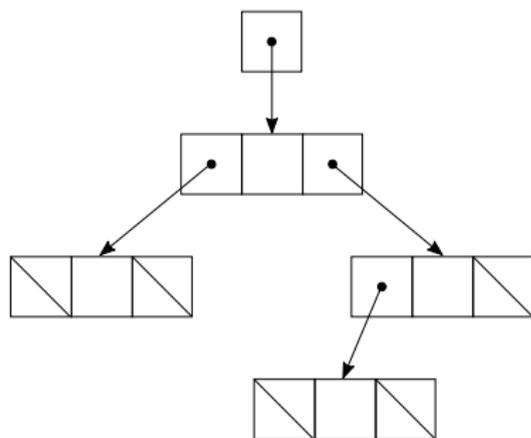
Todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a NULL)



- ▶ El *nodo raíz* es el primer nodo del árbol
- ▶ Cada enlace del nodo raíz apunta a un hijo

# Árbol binario

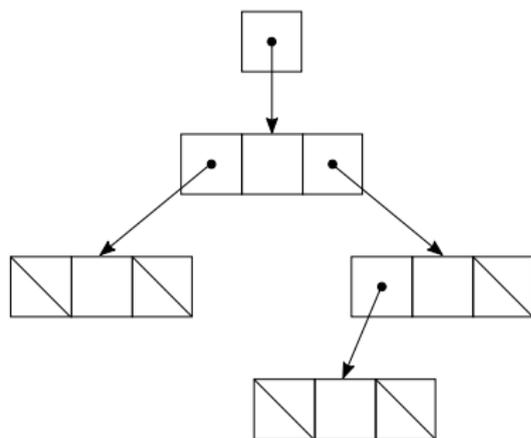
Todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a NULL)



- ▶ El *nodo raíz* es el primer nodo del árbol
- ▶ Cada enlace del nodo raíz apunta a un hijo
- ▶ El *hijo izquierdo* es el primer nodo del *sub-árbol izquierdo*

# Árbol binario

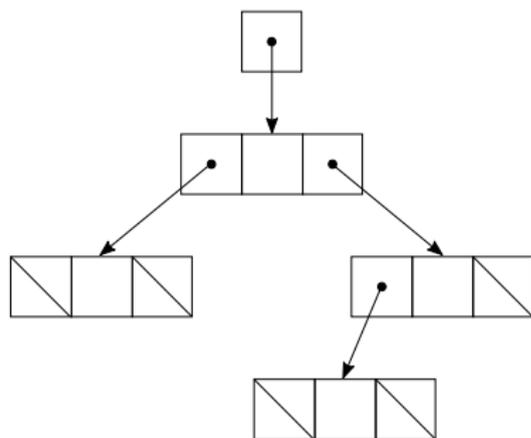
Todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a NULL)



- ▶ El *nodo raíz* es el primer nodo del árbol
- ▶ Cada enlace del nodo raíz apunta a un hijo
- ▶ El *hijo izquierdo* es el primer nodo del *sub-árbol izquierdo*
- ▶ El *hijo derecho* es el primer nodo del *sub-árbol derecho*

# Árbol binario

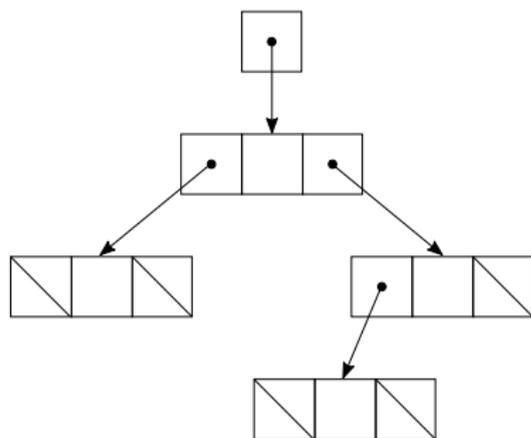
Todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a NULL)



- ▶ El *nodo raíz* es el primer nodo del árbol
- ▶ Cada enlace del nodo raíz apunta a un hijo
- ▶ El *hijo izquierdo* es el primer nodo del *sub-árbol izquierdo*
- ▶ El *hijo derecho* es el primer nodo del *sub-árbol derecho*
- ▶ Los hijos de los nodos se conocen como *descendientes*

# Árbol binario

Todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a NULL)



- ▶ El *nodo raíz* es el primer nodo del árbol
- ▶ Cada enlace del nodo raíz apunta a un hijo
- ▶ El *hijo izquierdo* es el primer nodo del *sub-árbol izquierdo*
- ▶ El *hijo derecho* es el primer nodo del *sub-árbol derecho*
- ▶ Los hijos de los nodos se conocen como *descendientes*
- ▶ Un nodo sin hijos se conoce como *nodo hoja*

# Árbol de búsqueda binario

- ▶ No contiene nodos duplicados

# Árbol de búsqueda binario

- ▶ No contiene nodos duplicados
- ▶ Los valores de cualquier sub-árbol izquierdo son menores que el valor de su nodo padre

# Árbol de búsqueda binario

- ▶ No contiene nodos duplicados
- ▶ Los valores de cualquier sub-árbol izquierdo son menores que el valor de su nodo padre
- ▶ Los valores de cualquier sub-árbol derecho son mayores que el valor de su nodo padre

# Árbol de búsqueda binario

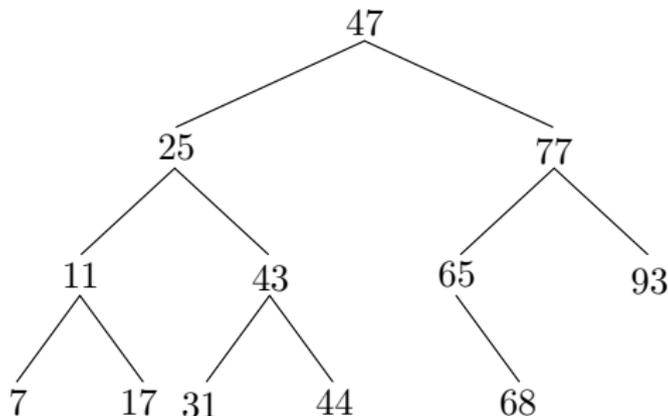
- ▶ No contiene nodos duplicados
- ▶ Los valores de cualquier sub-árbol izquierdo son menores que el valor de su nodo padre
- ▶ Los valores de cualquier sub-árbol derecho son mayores que el valor de su nodo padre

También se conoce como árbol binario ordenado.

# Árbol de búsqueda binario

- ▶ No contiene nodos duplicados
- ▶ Los valores de cualquier sub-árbol izquierdo son menores que el valor de su nodo padre
- ▶ Los valores de cualquier sub-árbol derecho son mayores que el valor de su nodo padre

También se conoce como árbol binario ordenado.



# Árbol binario – ejemplo D&D Fig.12.19

- ▶ En un árbol binario el nodo se inserta únicamente como nodo hoja
- ▶ Recorrer un árbol binario: proceso de visitar cada nodo de un árbol solo una vez de forma sistemática
- ▶ No tienen una única forma de ser recorrido como las estructuras lineales
- ▶ Se recorren de tres formas: *enorden*, *preorden*, y *postorden*

# Árbol binario – ejemplo D&D Fig.12.19

- ▶ En un árbol binario el nodo se inserta únicamente como nodo hoja
- ▶ Recorrer un árbol binario: proceso de visitar cada nodo de un árbol solo una vez de forma sistemática
- ▶ No tienen una única forma de ser recorrido como las estructuras lineales
- ▶ Se recorren de tres formas: *enorden*, *preorden*, y *postorden*

```
Los números colocados en el árbol son:  
5 6 12 0 10 11 12dup 7 7dup 11dup
```

```
El recorrido en preorden es:  
5 0 6 12 10 7 11
```

```
El recorrido en inorden es:  
0 5 6 7 10 11 12
```

```
El recorrido en postorden es:  
0 7 11 10 12 6 5
```

# Árbol binario – ejemplo D&D Fig.12.19

Definición de tipo y prototipos de funciones:

---

```
/* Estructura auto-referenciada */
struct nodoArbol {
    int dato; /* valor del nodo */
    struct nodoArbol *ptrIzq; /* Apuntador al subárbol izquierdo */
    struct nodoArbol *ptrDer; /* Apuntador al subárbol derecho */
};

typedef struct nodoArbol NodoArbol; /* Sinónimo de la estructura nodoArbol */
typedef NodoArbol *ptrNodoArbol; /* Sinónimo de NodoArbol* */

/* Prototipos */
void insertarNodo(ptrNodoArbol * , int );
void inOrden(ptrNodoArbol );
void preOrden(ptrNodoArbol );
void postOrden(ptrNodoArbol );
```

---

## Ejemplo D&D Fig.12.19 – Pasos para insertar nodo

---

```
void insertarNodo(ptrNodoArbol *ptrArbol, int valor)
{
    /* Si el árbol está vacío */
    if(*ptrArbol == NULL) {
        *ptrArbol = malloc(sizeof(NodoArbol));

        if(*ptrArbol != NULL) { /* Hay memoria */
            (*ptrArbol)->dato = valor;
            (*ptrArbol)->ptrIzq = NULL;
            (*ptrArbol)->ptrDer = NULL;
        } /* fin de if */
        else
            printf("No se insertó %d. No hay memoria disponible.\n", valor);
    } else { /* el árbol no está vacío */
        if(valor < (*ptrArbol)->dato) /* dato < nodo actual */
            insertarNodo( &( (*ptrArbol)->ptrIzq ), valor);
        else if(valor > (*ptrArbol)->dato) /* dato > nodo actual */
            insertarNodo( &( (*ptrArbol)->ptrDer ), valor);
        else /* valor duplicado */
            printf("dup");
    }
}
```

---

## Ejemplo D&D Fig.12.19 – Pasos para insertar nodo

Dentro de la función `insertarNodo()`:

1. Si `*ptrArbol` es `NULL`, crea un nuevo nodo llamando a `malloc()` y apuntado por `*ptrArbol`, asigna el valor a almacenar en `(*ptrArbol)->dato`, y los enlaces `(*ptrArbol)->ptrIzq` y `(*ptrArbol)->ptrDer` a `NULL`.

## Ejemplo D&D Fig.12.19 – Pasos para insertar nodo

Dentro de la función `insertarNodo()`:

1. Si `*ptrArbol` es `NULL`, crea un nuevo nodo llamando a `malloc()` y apuntado por `*ptrArbol`, asigna el valor a almacenar en `(*ptrArbol)->dato`, y los enlaces `(*ptrArbol)->ptrIzq` y `(*ptrArbol)->ptrDer` a `NULL`.
2. Si `*ptrArbol` no es `NULL`, y el valor a insertar es menor que `(*ptrArbol)->dato` se llama a `insertarNodo()` con la dirección `(*ptrArbol)->ptrIzq`; si no, se llama a `insertarNodo()` con la dirección `(*ptrArbol)->ptrDer`.

## Ejemplo D&D Fig.12.19 – Pasos para insertar nodo

Dentro de la función `insertarNodo()`:

1. Si `*ptrArbol` es `NULL`, crea un nuevo nodo llamando a `malloc()` y apuntado por `*ptrArbol`, asigna el valor a almacenar en `(*ptrArbol)->dato`, y los enlaces `(*ptrArbol)->ptrIzq` y `(*ptrArbol)->ptrDer` a `NULL`.
2. Si `*ptrArbol` no es `NULL`, y el valor a insertar es menor que `(*ptrArbol)->dato` se llama a `insertarNodo()` con la dirección `(*ptrArbol)->ptrIzq`; si no, se llama a `insertarNodo()` con la dirección `(*ptrArbol)->ptrDer`.
3. Se sigue la recursividad hasta que se encuentre un puntero `NULL` y se ejecuta el paso 1

## Ejemplo D&D Fig.12.19 – Recorrido pre-orden

---

```
/* Recorrido preorden del árbol */
void preOrden(ptrNodoArbol ptrArbol)
{
    /* Si el árbol no está vacío, entonces recórrelo */
    if(ptrArbol != NULL) {
        printf("%3d", ptrArbol->dato);
        preOrden(ptrArbol->ptrIzq);
        preOrden(ptrArbol->ptrDer);
    }
}
```

---

1. Visita el nodo raíz,
2. recorre el sub-árbol izquierdo, y finalmente
3. recorre el sub-árbol derecho.

## Ejemplo D&D Fig.12.19 – Recorrido en-orden

---

```
/* Recorrido inorden del árbol */
void inOrden(ptrNodoArbol ptrArbol)
{
    /* Si el árbol no está vacío, entonces recórrelo */
    if(ptrArbol != NULL) {
        inOrden(ptrArbol->ptrIzq);
        printf("%3d", ptrArbol->dato);
        inOrden(ptrArbol->ptrDer);
    }
}
```

---

1. Recorrer el sub-árbol izquierdo,
2. visita el nodo raíz, y finalmente
3. recorre el sub-árbol derecho.

## Ejemplo D&D Fig.12.19 – Recorrido post-orden

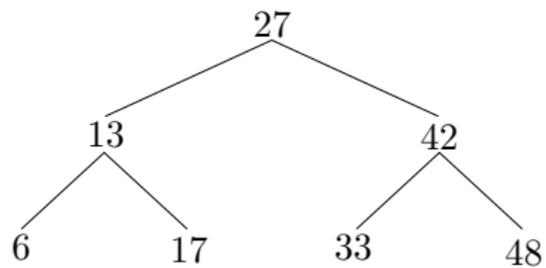
---

```
/* Recorrido postorden del árbol */
void postOrden(ptrNodoArbol ptrArbol)
{
    /* Si el árbol no está vacío, entonces recórrelo */
    if(ptrArbol != NULL) {
        postOrden(ptrArbol->ptrIzq);
        postOrden(ptrArbol->ptrDer);
        printf("%3d", ptrArbol->dato);
    }
}
```

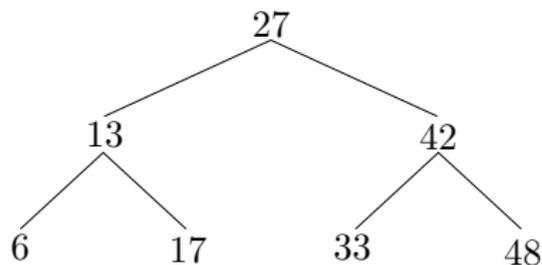
---

1. Recorrer el sub-árbol izquierdo,
2. recorrer el sub-árbol derecho, y finalmente
3. visita el nodo raíz.

# Resumen de recorridos



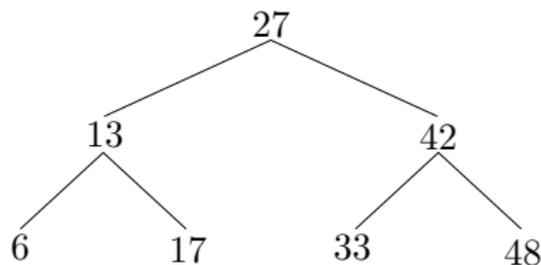
# Resumen de recorridos



► *preorden:* 27 13 6 17 42 33 48

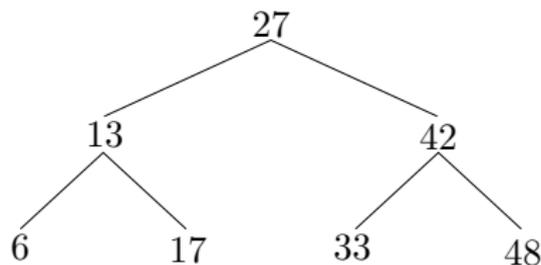
[<raíz> <izq> <der>]

# Resumen de recorridos



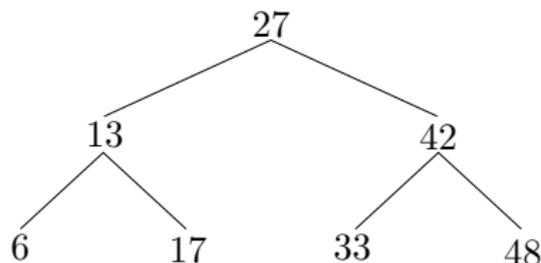
- ▶ *preorden:* 27 13 6 17 42 33 48 [**<raíz>** <izq> <der>]  
se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho

# Resumen de recorridos



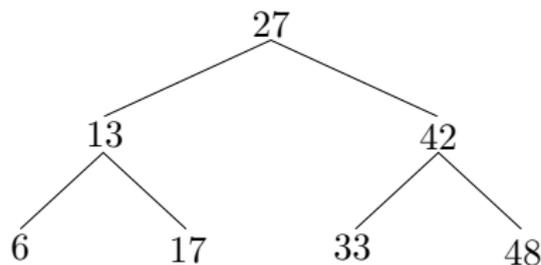
- ▶ *preorden:* 27 13 6 17 42 33 48 [**<raíz>** **<izq>** **<der>**]  
se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho
- ▶ *enorden:* 6 13 17 27 33 42 48 [**<izq>** **<raíz>** **<der>**]

# Resumen de recorridos



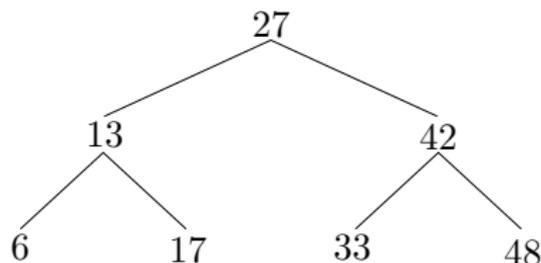
- ▶ *preorden:* 27 13 6 17 42 33 48 [**<raíz>** <izq> <der>]  
se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho
- ▶ *enorden:* 6 13 17 27 33 42 48 [**<izq>** **<raíz>** **<der>**]  
imprime los valores de los nodos en forma ascendente (el proceso de cargar un árbol de búsqueda binario de hecho ordena los datos)

# Resumen de recorridos



- ▶ *preorden:* 27 13 6 17 42 33 48 [**<raíz>** <izq> <der>]  
se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho
- ▶ *enorden:* 6 13 17 27 33 42 48 [**<izq>** <raíz> <der>]  
imprime los valores de los nodos en forma ascendente (el proceso de cargar un árbol de búsqueda binario de hecho ordena los datos)
- ▶ *postorden:* 6 17 13 33 48 42 27 [**<izq>** <der> <raíz>]

# Resumen de recorridos



- ▶ *preorden:* 27 13 6 17 42 33 48 [**<raíz>** <izq> <der>]  
se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho
- ▶ *enorden:* 6 13 17 27 33 42 48 [**<izq>** <raíz> <der>]  
imprime los valores de los nodos en forma ascendente (el proceso de cargar un árbol de búsqueda binario de hecho ordena los datos)
- ▶ *postorden:* 6 17 13 33 48 42 27 [**<izq>** <der> <raíz>]  
no se imprime el valor de cada nodo hasta que sean impresos los valores de sus hijos

