

Grado en Ingeniería Informática  
2018-2019

*Apuntes*  
**Estructura de datos y algoritmos**

---

Jorge Rodríguez Fraile<sup>1</sup>



Esta obra se encuentra sujeta a la licencia Creative Commons  
**Reconocimiento - No Comercial - Sin Obra Derivada**

---

<sup>1</sup>Universidad: 100405951@alumnos.uc3m.es | Personal: jrf1616@gmail.com



## **ÍNDICE GENERAL**

<b>I    Información</b>	<b>3</b>
<b>II    Tema 0. POO</b>	<b>29</b>
<b>III    Tema 1. Tipos Abstractos de Datos</b>	<b>93</b>
<b>IV    Tema 2. TADS</b>	<b>115</b>
<b>V    Tema 3. Algoritmos I - Análisis de algoritmos</b>	<b>225</b>
<b>VI    Tema 4. Recursion</b>	<b>281</b>
<b>VII    Tema 5. Arboles</b>	<b>329</b>
<b>VIII    Tema 6. Grafo</b>	<b>487</b>
<b>IX    Tema 7. Estrategias de algoritmos</b>	<b>557</b>



# **Parte I**

## **Información**



# Estructura de Datos y Algoritmos

Curso 2018-2019

Material elaborado por los profesores de la asignatura

# Estructura de Datos y Algoritmos

---

- ▶ Grado en Ingeniería Informática.
- ▶ Curso Iº, segundo semestre.
- ▶ 6 ECTS.
- ▶ Horas lectivas / semana: 1,66 + 1.66 = 3.32 h
- ▶ Coordinadora: Isabel Segura Bedmar  
[\(isegura@inf.uc3m.es\)](mailto:isegura@inf.uc3m.es), despacho 2.2.B05

# Los profesores del grupo 84M

---

- ▶ Profesor Grupo Magistral y Reducidos
- ▶ Lourdes Moreno López  
(lmoreno@inf.uc3m.es), 2.2.B05.
  
- ▶ Tutorías (mirar Aula Global)

# Objetivos docentes

---

- ▶ Comprender la necesidad de estructurar el software y usar la abstracción como herramienta conceptual para conseguirlo.
- ▶ Capacidad para especificar de manera informal una estructura de datos y sus operaciones asociadas.
- ▶ Conocer las estructuras de datos lineales (pilas, colas, listas) y jerárquicas (árboles), grafos, sus implementaciones más comunes, sus algoritmos de manipulación y su utilidad en aplicaciones reales
- ▶ Aprender a manejar las estructuras de datos básicas y extenderlas a otras más complejas para solucionar problemas

# Objetivos docentes

---

- ▶ Entender la importancia que tiene la selección de una determinada estructura de datos para resolución de un problema y razonar sobre la solución según la eficiencia.
- ▶ Capacidad para analizar algoritmos y decidir entre la aplicación de uno o varios algoritmos para un problema dado, en función de la eficiencia y contexto de uso
- ▶ Capacidad para usar el concepto de recursividad en la formulación de un algoritmo
- ▶ Capacidad para usar distintas estrategias algorítmicas para resolver problemas, en particular, divide y vencerás y vuelta atrás.
- ▶ Implementar estructuras de datos y sus operaciones en Java

# Contenidos

---

- ▶ Tema 0 – Conceptos Básicos Programación Orientada a Objetos.
- ▶ Tema I – Tipos Abstractos de Datos (TADs)
- ▶ Tema 2 – TAD Lineales:
  - ▶ Pilas y Colas
  - ▶ Listas simplemente enlazadas
  - ▶ Listas doblemente enlazadas
- ▶ Tema 3 – Algoritmos I. Análisis de Algoritmos
- ▶ Tema 4 – Algoritmos II. Recursión

# Contenidos

---

- ▶ **Tema 5 – TAD Jerárquicos:**
  - ▶ Árboles Binarios de Búsqueda.
  - ▶ Equilibrado de árboles en tamaño y altura (árboles AVL).
- ▶ **Tema 6 – Grafos**
  - ▶ Implementación basada en matriz de adyacencias.
  - ▶ Implementación basada en lista de adyacencias.
  - ▶ Algoritmos de búsqueda en amplitud y en profundidad.

# Contenidos

---

- ▶ **Tema 7 – Algoritmos III. Estrategias Algorítmicas**
  - ▶ Divide y Vencerás.
  - ▶ Panorámica de otras estrategias: vuelta atrás, programación dinámica, algoritmos voraces, fuerza bruta y algoritmos heurísticos.

# Pruebas de Evaluación Continua

---

- 1) Primer Examen Parcial (20% nota) → **21 de Marzo**
  - ▶ Temas 0,1,2,3 y 4.
- 2) Trabajos semanales (+5%, si al menos 4 son correctos)
- 3) Caso Práctico (20% nota)
  - ▶ Grupos de 2 alumnos.
  - ▶ Publicación Enunciado: 12 de Marzo.
  - ▶ Entrega por aula global: **6 Mayo 9.00 am**
  - ▶ Defensa: 6 Mayo.
- 4) Examen final (60% nota) → **31 de Mayo**
  - ▶ Carácter OBLIGATORIO.Todos los temas.
  - ▶ Nota mínima para evaluación continua: 4

# Trabajos semanales

---

- ▶ Fomentar en el alumno su aprendizaje activo y compromiso con la asignatura.
- ▶ En la primera parte del curso (28/01-18/03), cada semana se publicará un problema que los alumnos deberán tratar de resolver de forma individual.
- ▶ Cada semana, durante la clase de laboratorio, se publicará y explicará el enunciado del trabajo semanal.
- ▶ La fecha límite para entregar la solución del problema será el lunes siguiente antes de las 9.00 am.
- ▶ La entrega se hará siempre vía aula global.

## Trabajos Semanales

---

- ▶ En total, se publicarán 6 problemas
- SÓLO los alumnos que hayan entregado los ejercicios (un mínimo de 5 problemas resueltos correctamente) podrán beneficiarse de subir su nota final un 5%.

## Evaluación no continua

---

- ▶ **Examen de la convocatoria ordinaria.**
  - ▶ Fecha **31 de Mayo**.
  - ▶ Nota final será el 60% de la nota obtenida en este examen.
  - ▶ **Ejemplos:**
    - Si alumno saca un 9 (sobre 10) en este examen, su nota final en la asignatura sería un 5.4.
    - Si el alumno saca un 5 (sobre 10), su nota final sería un 3.
  - ▶ **Por tanto, para aprobar la asignatura sin seguir la evaluación continua, es necesario sacar un 8.3 en este examen.**

## Convocatoria Extraordinaria (18 Junio)

---

- ▶ Se aplicará evaluación continua siempre que sea más favorable para el alumno. En este caso, el valor del examen es 60%.
- ▶ Nota mínima para aplicar evaluación continua: 4 puntos (sobre 10).
- ▶ Si no se sigue la evaluación continua, el valor del examen final es el 100%.

# Bibliografía

---

- ▶ Goodrich and Tamassia  
Data Structures and Algorithms in Java, 4th edition  
John Wiley & Sons
- ▶ H.Al-Jumaily et al.  
Estructuras de Datos y Algoritmos. Manual de Ejercicios.  
UC3M
- ▶ I. Segura-Bedmar et al  
A friendly notebook on Data Structures and Algorithms. UC3M
- ▶ Mark Allen Weiss  
Data Structures and Algorithms Analysis in Java, 2nd edition  
Pearson International Edition

# Otros recursos online

---

## ► Aula Global

- ▶ Se publicará
  - ▶ Material de clase
  - ▶ Enunciados y soluciones
  - ▶ Código Java
  - ▶ ...

## ► OCW 2018

## ► Tutoriales y libros on-line

- ▶ <http://docs.oracle.com/javase/tutorial/java/TOC.html> (nivel medio)
- ▶ <http://introcs.cs.princeton.edu/java/home/> (nivel medio-alto)
- ▶ Etc....

## **Estructuras de Datos y Algoritmos**

Grado Ingeniería Informática

Presentación Clases en Aula Informática

Grupos 84-85, Curso 2018-2019



# Profesor de Prácticas

---

Lourdes Moreno López, 2.2.B05,  
lmoreno@inf.uc3m.es

## Clases:

- Grupo 84: Lunes, 17.00-18.40 am, 1.2.G01
- Grupo 85: Lunes, 15.00-16.40 am, 1.2.G03

## Tutorías (mirad horarios Aula Global)

## Qué vamos a hacer en las clases de laboratorio?

---

- Durante las primeras seis semanas, trabajaremos en problemas relacionados con los temas 0-4.
  - Repaso Programación.
  - Problemas sobre estructuras de datos lineales (pilas, colas y listas).
  - Problemas sobre análisis de algoritmos.
  - Problemas sobre recursión.
- A partir de la séptima semana hasta final de curso, trabajaremos en un caso práctico.

# ¿Qué problemas vamos a resolver durante las primeras 6 semanas?

Semana	Fecha	Problema
1	28 Ene	<b>Matriz</b> ( <i>revisión de programación</i> )
2	4 Feb	<b>TAD Polinomio</b> ( <i>definición e implementación de tipos abstractos de datos</i> )
3	11 Feb	<b>Paréntesis balanceados</b> ( <i>problema de pilas</i> ) <b>y Problema de Josephus</b> ( <i>problema de colas</i> )
4	18 Feb	<b>Implementación de una lista simplemente enlazada</b>
5	25 Feb	<b>Implementación de una lista doblemente enlazada</b>

# Qué problemas vamos a resolver?

---

Semana	Fecha	Problema
6	4 Marzo	<b>Problemas sobre análisis de algoritmos</b>
7	11 Marzo	<b>Problemas sobre recursión</b>

## Trabajos individuales (semanas 1 - 6)

---

- Además, **cada semana** se publicará el enunciado de **un problema**, que deberéis intentar **resolver de forma individual**.
- El enunciado se publicará el lunes, y su **solución** se deberá subir a aula global antes de las **9.00 am del próximo lunes**.
- **Incremento nota final = 0.5 puntos** (si habéis entregado **al menos 5 de los 6 problemas** propuestos. Sus soluciones deben ser correctas).

# Trabajos individuales

Fecha publicación	Problema	Fecha entrega (tarea aula global)
4 Feb	<i>Implementación Biblioteca basado en array</i>	11 Feb, 9 a.m
11 Feb	<i>Problema sobre TAD pilas y colas (pendiente de definición)</i>	18 Feb, 9 a.m
18 Feb	<i>Implementación Biblioteca basado en lista simple</i>	25 Feb, 9 a.m
25 Feb	<i>Implementación Biblioteca basado en lista doble</i>	4 Marzo, 9 a.m
4 Marzo	<i>Problema sobre complejidad de los métodos de los TAD Lineales</i>	11 Marzo, 9 a.m
11 Marzo	<i>Problema sobre recursión (pendiente de definición)</i>	18 Marzo, 9 a.m

# Caso Práctico

---

- Fecha de Publicación: **18 de Marzo**
- Implementación red social. Tres fases: listas, árboles binarios de búsqueda y grafos.
- Grupos de dos alumnxs.
- **Fecha de entrega: 6 May, 9.00 am (a través de aula global).**
- **Defensa: 6 May, 9.00 am.**
- **Evaluación basada en JUnit.**



# **Parte II**

## **Tema 0. POO**



# Tema 0. Conceptos básicos Programación Orientada a Objetos

Estructura de Datos y Algoritmos (EDA)

# Objetivos

---

- ▶ Al final de la clase, los estudiantes deben ser capaces de:
  - 1) Comprender los conceptos de clase y objeto
  - 2) Comprender las características del soporte de Java a la Programación Orientada a Objetos (POO)

# Contenidos

---

1. ¿Qué es la POO?
2. Clases y objetos
3. Atributos
4. Métodos
5. Paso de parámetros en Java
6. Destrucción de objetos

# Contenidos

---

1. **¿Qué es la POO?**
2. Clases y objetos
3. Atributos
4. Métodos
5. Paso de parámetros en Java
6. Destrucción de objetos

# ¿Qué es la POO?

---

- ▶ Un paradigma de programación que define un programa como un conjunto de objetos que realizan acciones
  - ▶ Un objeto tiene:
    - ▶ propiedades: datos almacenados en atributos
    - ▶ operaciones: métodos
  - ▶ Una clase define una plantilla para un tipo de objetos
-

# Ventajas de la POO

---

- ▶ Los datos y los comportamientos se encapsulan en una sola unidad (clase)
- ▶ Fácil de mantener
- ▶ Permite un desarrollo más rápido
- ▶ Promueve la reutilización

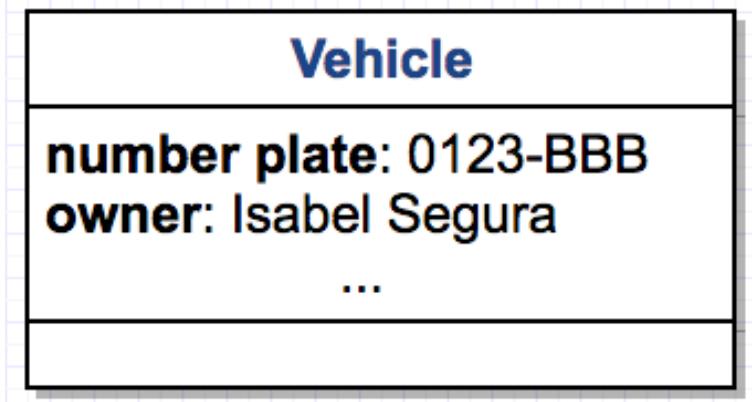
# Contenidos

---

1. **¿Qué es la POO?**
2. **Clases y objetos**
3. Atributos
4. Métodos
5. Paso de parámetros en Java
6. Destrucción de objetos

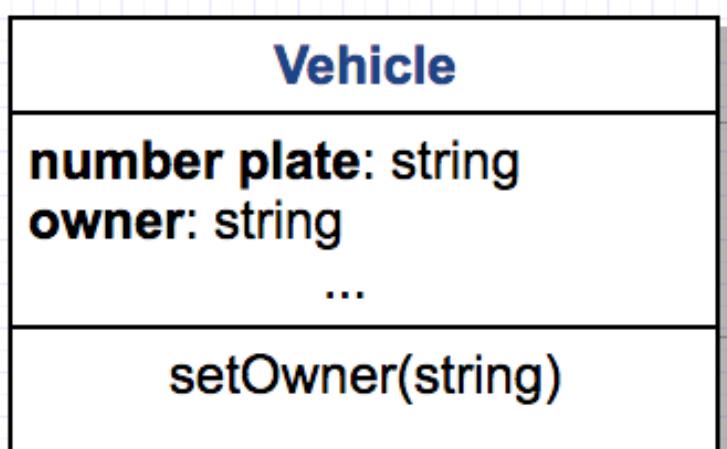
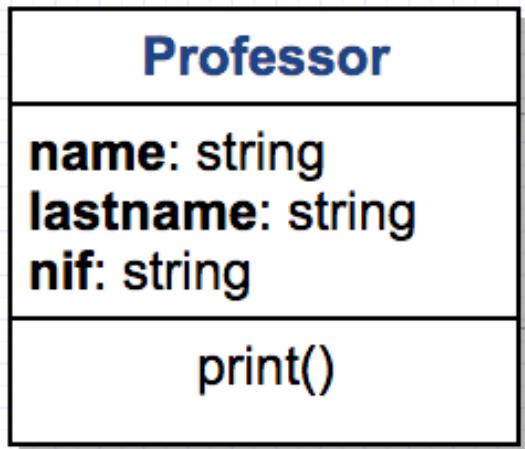
# Clases y objetos

- ▶ Objeto: representa una entidad del dominio de la aplicación



# Clases y objetos

- ▶ Clase: plantilla para objetos
- ▶ Clase = **estructura de datos + operaciones**



▶ 9

<https://www.gliffy.com/uses/uml-software/>

# Clases en Java

---

```
public class <name_class> {  
    <attributes>  
    <methods>  
}
```

```
public class Point {  
}
```

# Modificadores

---

- **public:** accesible desde cualquier otra clase
- **private:** no accesible fuera de esta clase
- **protected:** accesible por subclases
- **none:** solo accesible en el mismo paquete
- **final:** no se puede modificar
- **static:** compartido por todos los objetos de la clase

# Recomendaciones sobre el alcance de los modificadores

---

- ▶ **public** para clase
- ▶ **private** para todos los atributos y para definir sus métodos getters y setters (podemos controlar cómo se modifican los atributos)
- ▶ **public** para métodos (excepto para métodos internos)

# Contenidos

---

1. ¿Qué es la POO?
2. Clases y objetos
3. Atributos
4. Métodos
5. Paso de parámetros en Java
6. Destrucción de objetos

# Atributos

---

- ▶ Representar el estado de un objeto.  
**[modifiers] <type> <name\_attribute>;**
  
- ▶ Puede ser un tipo básico (char, int, float, boolean, etc), una colección (array) o incluso un objeto

# Static vs. Non-static atributos

---

- ▶ **Static (or class) atributos:** valores comunes para todos los objetos (constants, counters, etc)
  - ▶ Se pueden llamar directamente (sin crear ningún objeto): nombre de la clase, punto (.), seguido del nombre del método
- ▶ **Non-static atributos:** son locales a una instancia del objeto (cada objeto tiene sus propios valores)
  - ▶ Se debe crear una instancia de objeto para acceder a ellos

```
public class Point { Declaración de static attribute  
    public final static float MAX=100;  
    public float x;  
    public float y; Declaración de non-static attribute  
}
```

```
public class TestPoint {  
    public static void main(String args[]) {  
        System.out.println(Point.MAX);  
        Point p=new Point(); Acceso al static  
        p.x=3; attribute  
        p.y=4;  
    } Acceso al non-static attribute  
}
```

# Contenidos

---

1. ¿Qué es la POO?
2. Clases y objetos
3. Atributos
- 4. Métodos**
5. Paso de parámetros en Java
6. Destrucción de objetos

# Métodos

---

- ▶ Definir el comportamiento de un objeto
- ▶ Operar sobre los atributos
- ▶ Pueden aceptar parámetros como entrada
- ▶ Pueden devolver un valor u objeto como salida

# Constructores

---

- ▶ Crean un objeto y proporciona valores iniciales para sus atributos
- ▶ Mismo nombre que la clase
- ▶ Pueden tomar argumentos
- ▶ No devuelven nada (ni siquiera vacío)
- ▶ Se ejecutan con el operador **new** (se asigna memoria para el objeto)
- ▶ Una clase puede tener varios constructores

# Constructores

```
public class Point {  
    public float x;  
    public float y;  
}
```

Si no se define ningún constructor, Java proporciona un constructor implícito que crea un objeto e inicializa sus atributos con valores por defecto

```
public static void main(String args[]) {  
    Point p=new Point();  
    System.out.println("Point="+ p.x +"," + p.y);  
}
```

Produce el siguiente resultado: Point= 0.0 , 0.0

# Constructores

```
public Point(float a, float b) {  
    x=a;  
    y=b;  
}
```

Si defines un constructor, el constructor implícito no existe

```
4  
5     public static void main(String args[]) {  
6         The constructor Point() is undefined.  
7         Point p=new Point();  
8         System.out.println("Point="+ p.x +"," + p.y);  
9     }
```

Error: El constructor Point () no está definido

# Creación de un objeto

Point **p**;

p es una variable de referencia para un objeto de clase Point. Esta instrucción declara la variable p, pero el objeto aún no se ha creado

**p=new Point(3,5);**

El operador **new** asigna memoria para el objeto

El constructor crea el objeto

La variable p contiene la dirección de memoria donde se almacena el objeto

p también nos permite llamar a los atributos y métodos del objeto



# Creación de un objeto

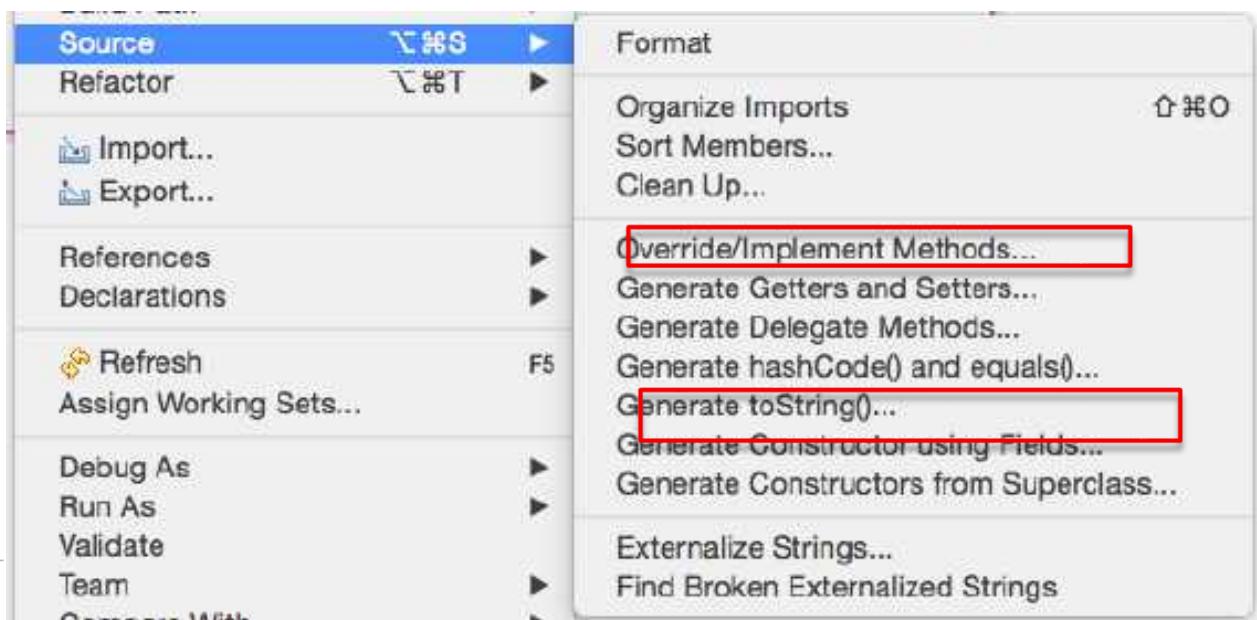
---

```
Point p =new Point(3,5);
```

También puedes crearlo en la misma línea !!!

# Escribe menos con Eclipse ...

- ▶ La opción **Source** permite generar automáticamente un constructor así como los métodos getters y setters (para manipular los atributos)



```
public class Point {  
    public float x;  
    public float y;  
    public Point(float a, float b) {  
        x = a;  
        y = b;  
    }  
    public float getX() {return x;}  
    public float getY() {return y;}  
  
    public void setX(float a) {  
        x = a;  
    }  
    public void setY(float b) {  
        y = b;  
    }  
}
```

## Cómo acceder a los atributos y métodos del objeto

---

```
Point p=new Point(3,5);
p.setX(p.getX()+2);
p.setY(p.getY()-5);
System.out.println(p.getX()+" "+p.getY());
```

Usando la variable de referencia, p, seguido de punto (.) Y el nombre del método o atributo

## ¿Qué pueden devolver los métodos?

```
public float getX() {           a tipo básico (char, int,  
    return x;                  float, double, boolean, etc)  
}  
  
public Point clone() {          Un objeto  
    return new Point(x,y);  
}  
  
public void show() {            void  
    System.out.println("(" + x + ", " + y + ")");  
}
```

**Nota:** también pueden devolver arrays de objetos o de tipos básicos

## Static vs. Non-static métodos

---

- ▶ **Static (o class) métodos:** son métodos generales que devuelve valores calculados a partir de los valores de los parámetros
  - ▶ **Se puede sólo acceder** a static atributos/métodos
- ▶ **Non-static métodos:** manipula los atributos de un objeto
  - ▶ **Se puede acceder** a non-static y static atributos/métodos

## this

---

- ▶ Un objeto puede referirse a sí mismo con la palabra clave **this**

```
public Point(float a, float b) {  
    this.x = a;  
    this.y = b;  
}
```

# this

---

- ▶ Si un atributo y un parámetro comparten el mismo nombre => use **this** para distinguirlos.

```
public class Point {  
    public float x;  
    public float y;  
  
    public Point(float x, float y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

# Contenidos

---

1. ¿Qué es la POO?
2. Clases y objetos
3. Atributos
4. Métodos
5. **Paso de parámetros en Java**
6. Destrucción de objetos

# Paso de parámetros en Java

---

- ▶ **Una variable de tipo básico** siempre se pasa por valor (si el método cambia este valor, el efecto no es visible fuera del método)
- ▶ Las **Referencias a variables** de arrays y objetos siempre pasan por valor. Estas referencias son su dirección de memoria.
- ▶ Sin embargo, los valores de los arrays o de los atributos de un objeto puede ser modificado por el método, y el cambio será visible fuera del método

# Ámbito

---

- ▶ Parámetros y variables locales: un único método donde se definen
- ▶ Non-static atributos: variables globales para todos los non-static métodos
- ▶ Static atributos: variables globales para static y non-static métodos

# Contenidos

---

1. ¿Qué es la POO?
2. Clases y objetos
3. Atributos
4. Métodos
5. Paso de parámetros en Java
6. Destrucción de objetos

## Destrucción de objetos

---

- ▶ Java proporciona un **garbage collector** para encontrar automáticamente objetos y arrays que ya no son necesarios

## Chuleta de Java

Jose Jesus García Rueda  
(Adaptado de Jialong He)

### Tipos de datos básicos

<b>Byte</b>	8	-128..127
<b>short</b>	16	-32,768..32,767
<b>int</b>	32	-2,147,483,648.. 2,147,483,647
<b>long</b>	64	-9,223,372,036,854,775,808.. 9,223,372,036,854,775,807
<b>float</b>	32	3.4e-038.. 3.4e+038
<b>double</b>	64	1.7e-308.. 1.7e+308
<b>char</b>	16	Carácter Unicode
<b>Boolean</b>		true, false

Declaración de variable:  
*tipo* identificador [= *valor*];

### Arrays

<b>int nombre_array[ ];</b>	declara un array de enteros de tamaño 100
<b>int nombre_array[ ] = new int [100];</b>	declara un array de enteros en una sola línea
<b>int nombre_array [ ] = { 1, 2, 3, 4};</b>	
<b>int nombre_array[] = new int [10][20];</b>	array multidimensional

**null**

### Clase

```
{public|final}abstract class nombre
{ declaracion_variables_clase }
public static void main{String[] args} {código}
{métodos}
}
```

this, super

### Método

```
{public|private } [static] {tipo | void} nombre(arg, ..., arg)
{código}
```

### Variable

```
{public|private } [static] tipo name [= expresión];
```

### Interface

interface nombre

```
{
    Lista de métodos declarados.
}
```

implements

```
sentencias;
}
definición de método 1
...
definición de método N
}
```

### Operadores aritméticos

```
+, -, *, /
%, ++, --
sumas, resta, multiplicación, división
modulo, incremento, decremento
```

### Operadores Relacionales

```
==, !=, >, <,
>=, <=
igual, distinto, mayor, menor
mayor o igual, menor o igual
```

### Operadores Lógicos

```
&, |, !, ^, ||, &&
AND, OR, NOT, XOR, if OR, AND
```

### Comentarios

```
// resto de la línea
/* comentario multilinea */
/** comentario para documentación */
```

### Compilación y Ejecución

```
javac NombreDeFichero.java
java NombreDeFichero
El CLASSPATH debe estar correctamente configurado
El nombre del fichero ha de ser igual al nombre de la clase
```

### Control de Flujo

```
if (expresión booleana) sentencia1; [ else sentencia2; ]
```

```
Bucle while
[inicialización]
while (condición de terminación) {
    cuerpo;
    [iteración];
}
```

```
Bucle do while
do {
    cuerpo;
    [iteración];
} while (condición de terminación);
```

```
Bucle for
for (inicialización; condición de terminación; iteración) {
    cuerpo;
}
```

```
Estructura de Programa
class NombreClase {
    public static void main (String args[ ]) {
```

### Tratamiento de excepciones

```
try
{
    Código que puede dar lugar a excepciones
}
catch(TipoDeExcepción1 obj1)
{
    Manejador de excepciones de tipo 1
}
catch(TipoDeExcepción2 obj2)
{
    Manejador de excepciones de tipo 2
}
finally{
    Se ejecuta, haya habido o no excepción, tras terminarse con el try
}
```

### Palabras Clave

<b>abstract</b>	<b>boolean</b>	<b>break</b>	<b>byte</b>	<b>case</b>
<b>char</b>	<b>class</b>	<b>default</b>	<b>do</b>	<b>double</b>
<b>else</b>	<b>extends</b>	<b>false</b>	<b>final</b>	<b>float</b>
<b>for</b>	<b>if</b>	<b>implements</b>	<b>import</b>	
<b>instanceof</b>	<b>int</b>	<b>interface</b>	<b>long</b>	<b>new</b>
<b>null</b>	<b>package</b>	<b>private</b>	<b>protected</b>	<b>public</b>
<b>return</b>	<b>short</b>	<b>super</b>	<b>switch</b>	
<b>synchronized</b>	<b>static</b>	<b>throw</b>	<b>throws</b>	<b>true</b>
<b>try</b>	<b>this</b>	<b>void</b>	<b>while</b>	

### Paquetes de Java

**java.awt** Incluye las clases necesarias para crear un applet y las clases y las clases que un applet usa para comunicarse con su contexto.

**java.awt** Contiene todas las clases para crear interfaces de usuario para dibujar gráficos e imágenes.

**java.awt.color** Incluye clases para tratar el color.

**java.awt.event** Incluye interfaces y clases para tratar los diferentes tipos de eventos.

**java.awt.font** Incluye clases para tratar las fuentes de letra.

**java.awt.image** Incluye clases para crear y modificar imágenes.

**java.io** Permite la entrada y salida de datos del sistema a través de "data streams", serialización y el sistema de ficheros.

**java.lang** Incluye clases fundamentales para el diseño de programas en lenguaje Java.

**java.util** Miscelánea.

Universidad Carlos III de Madrid

# Eclipse Tutorial

Algorithms & Data Structures

Algorithms and Data Structures (ADS) lecturers

## Content

1.	Defining a Workspace .....	3
1.1	Workspace.....	3
1.2	Java Projects .....	3
1.2.1	Creating projects .....	3
1.3	Packages.....	4
1.4	Classes.....	4
2.	Running a project in Eclipse.....	5
2.1.1	Using running parameters .....	5
2.1.2	Debugging programs .....	6
3.	Reusing projects in Eclipse .....	6
3.1.1	Exporting projects.....	6
3.1.2	Importing projects .....	7
3.1.3	Libraries.....	8
4.	Related additional links .....	10
5.	Exercise. Make your work environment ready .....	10

# 1. Defining a Workspace

Eclipse is an open source Java development environment supported by a Graphical User Interface (GUI).

## 1.1 Workspace

The first time we run Eclipse (and the following ones if we want so), we are asked about the path to the desired workspace. It is the place in our hard drive where Eclipse will store user generated code (.java files) and compiled code (.class files) along with all the information regarding the Java project we are developing.

It is recommended for students to create the workspace directly in a USB memory stick if there is one available. In this way, all the work done by students during a session will go with them once finished. For this purpose, it is only necessary to write or select a path in the pen drive when launching Eclipse.



Figure 1: Selecting a workspace

## 1.2 Java Projects

Eclipse manages Java developments through *projects*.

### 1.2.1 Creating projects

Before being able to write code in Eclipse it is needed to create a project. To do so, follow these steps:

1. Select option menu *File >> New >> Java Project*
2. Fill in data for your project. For simple projects, like the ones to be developed along this course, filling in the name of the project is enough. Create a project named 'EDALab'.
3. Once project data have been introduced, press *Finish* button.

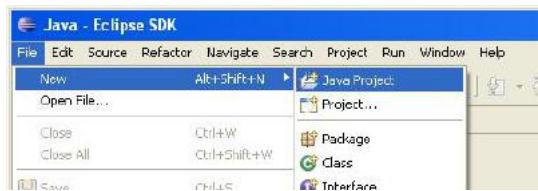


Figure 2: Creating a project.

Eclipse will create a new folder in the workspace for the project with *.classpath* and *.project* files. These files do not contain source code but data about configuration, compilation options and so on, for the given project. Java source code will be stored in a subfolder named 'src'

under the project folder while the compiled code (.class files) will be stored in the subfolder 'bin'. It is possible to change the current workspace by selecting *File >> Switch WorkSpace*.

### 1.3 Packages

A package is a special folder inside a project including a group of classes that have been created into the project. If no package is given when creating a class, the *default package* is used. Nonetheless, it is recommended to create at least one package per project.

The steps to follow when creating a package are:

1. Select *File>> New >> Package*.
2. Name the package in the dialog window appearing. By convention, all packages in Java are named starting with a non-capital letter. As an example, for the purpose of this lab work, you could create a package called 'maths'.

### 1.4 Classes

To create a Java class being part of the project you are developing, you must follow the following steps:

1. Select *File>> New >> Class*.
2. Name the class in the dialog window appearing. You must also say in which package the class must be included. By convention, all Java classes start with a capital letter. As an example, you could create a class called 'Calculator'.
3. It is needed to explicitly mark if a `main` method is desired by selecting the right check box.
4. Select *Finish* button.



Figure 6: Dialog window to create a class

Eclipse will immediately create a file with the name <class\_name>.java with the structure for the class (i.e.: the definition of the header for the class according to the elements selected in the

Class creation dialog box). Then, the rest of the code for the class must be provided. For our purposes, you can use the following code:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

## 2. Running a project in Eclipse

Once a project has been coded, it can be run by selecting the option *Run >> Run* or by pressing *Control + F11*. When execution is started Eclipse will show the console (at the bottom of the Eclipse window) with the execution result.

### 2.1.1 Using running parameters

It is possible to include arguments when calling the main method included in the class by selecting *Run >> Open Run Dialog*. In the *Arguments* tab it is possible to include the desired arguments list. Each of these arguments (separated by a blank space or a carriage return) will be stored in adjacent positions of the *args* array. So, the first argument will be stored in *args[0]*, the second one in *args[1]* and so on.

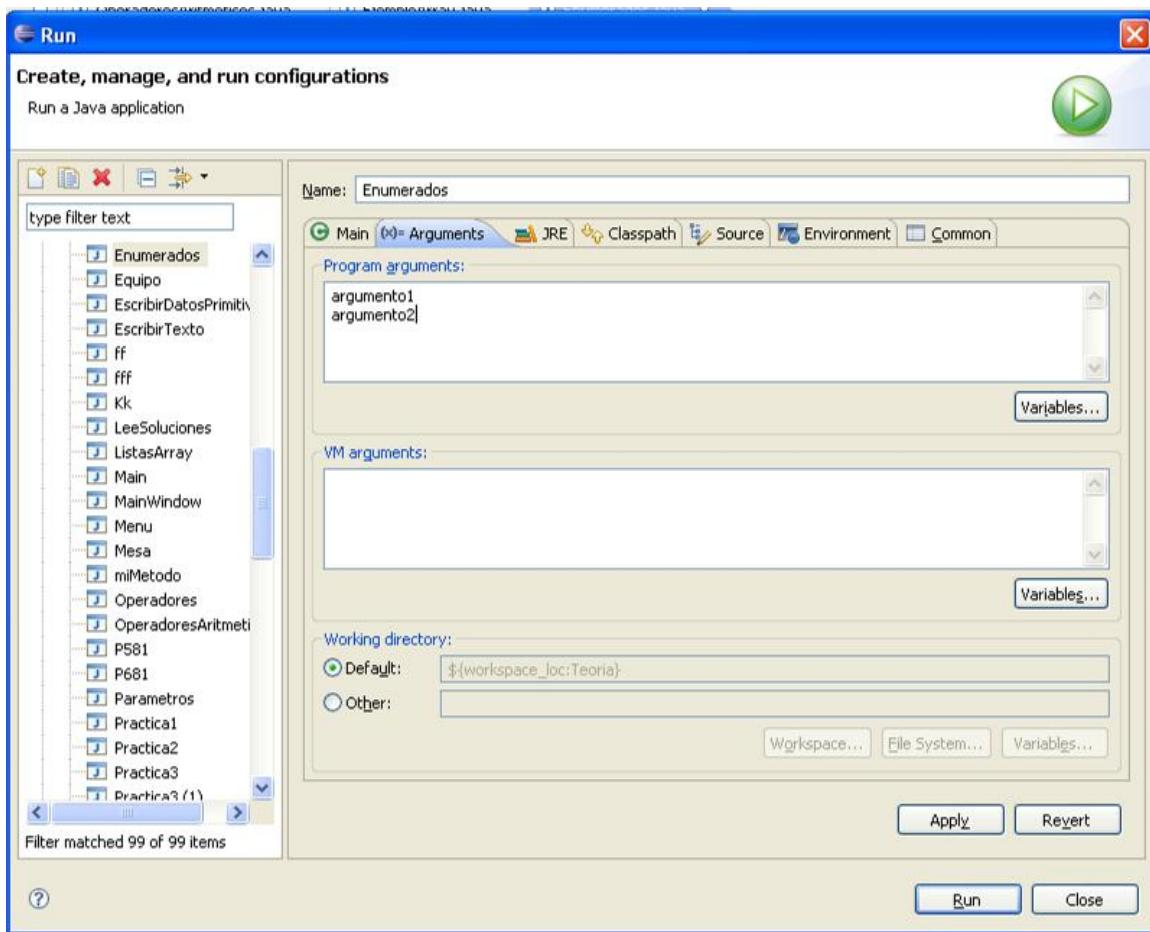


Figure 7: Using arguments when running Java applications

Please, notice that if no `main` method is given in a class implementation there is nothing to run, so an error will appear.

### 2.1.2 Debugging programs

Eclipse workspace is distributed along the concept of *perspective*. Each of these perspectives shows different functionalities according to a given task. For example, the Java perspective it is useful to include source code and running programs while the Debug perspective it is useful when debugging source code. The active perspective in the Java environment can be changed in *Window >> Open perspective* or by clicking in the button appearing in the right upper corner. As already mentioned, to introduce Java source code the Java perspective must be selected.

Once in the Debug perspective, program execution can be debugged by using breakpoints' i.e., source code lines where the execution process will stop. At these breakpoints it is possible to see the data contained in any of the variables or data structures used in the source code. Menu option *Run >> Toggle BreakPoint* is used to mark a given line and stopping running execution at that line. Once breakpoints have been selected, the debugging process can be launched with the option *Run >> Debug*. Program execution will stop at defined breakpoints.

## 3. Reusing projects in Eclipse

### 3.1.1 Exporting projects

Exporting and importing projects it is a core functionality that every Eclipse user should know. Thanks to these functions it is possible for students to work in the same projects at the classroom and at home.

Storing an active project in a .zip file can be done following these steps:

1. Select the desired project to be exported in the *Package Explorer* window – (*Java Perspective*)
2. Select the option menu *File >> New >> Export* (or click on the option *>>Export* appearing in the contextual menu shown by clicking in the right mouse button). See Figure 3.
3. To use a .zip file as a container for the exported project select *General >> Archive File* and press *Next* button.
4. Mark the contents you want to export. For example, it is recommended NOT to export the 'bin' folder because it contains .class files that can be obtained again by compiling the project. In this way you can save disk space. Press *Browse* button to select the path and .zip file where the project will be exported. Be sure that you have selected to store the project as a .zip file and that '*Create only selected directories*' option is marked. Press *Finish* to perform the export process. (see Figure 4).

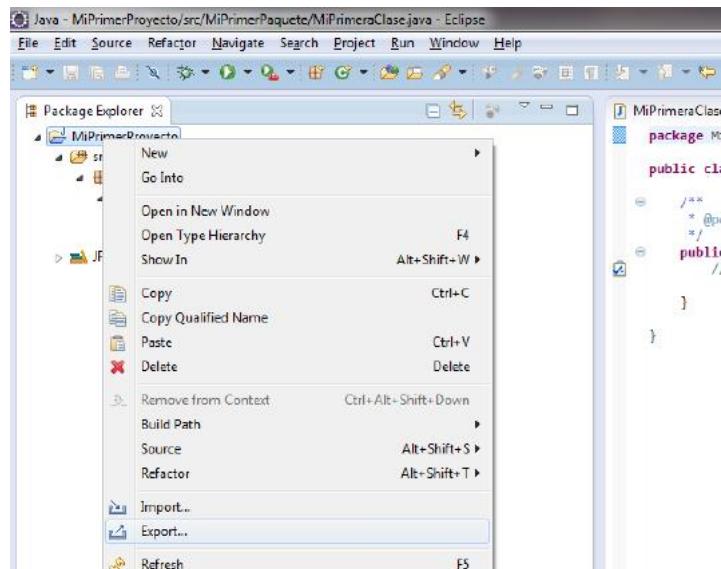


Figure 3: Exporting a project.

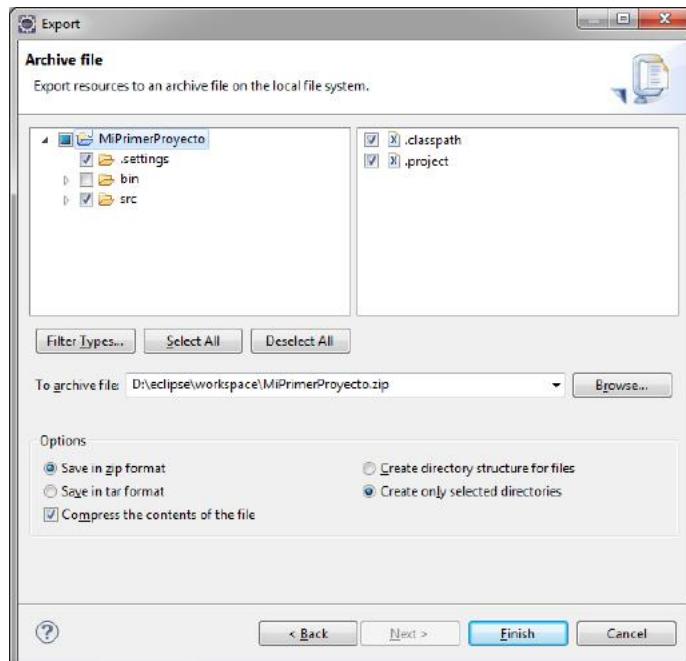


Figure 4: Export to file window

**WARNING:** It is also possible to export packages, classes, and so on instead of exporting the whole project. To do so you only need to place your pointer (in step 1) in the element you want to export instead of the project folder.

### 3.1.2 Importing projects

To import a previously exported .zip project (see section 3.1.1) you only have to take the following steps:

1. Create a project
2. Select *File >> New >> Import* (or press right button and select *>>Import* when the mouse is just over the just created project). See Figure 5.
3. Select *General >> Existing Projects into Workspace* and press *Next* button to continue.
4. Select the input .zip file through the option *Select archive File* (located where the project has been previously exported) and press *Finish* button (see Figure 6)

- Check that all packages, classes and so on have been correctly imported.

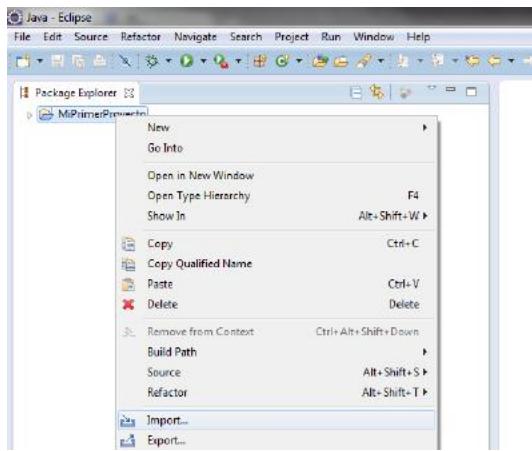


Figure 5: Importing a project.

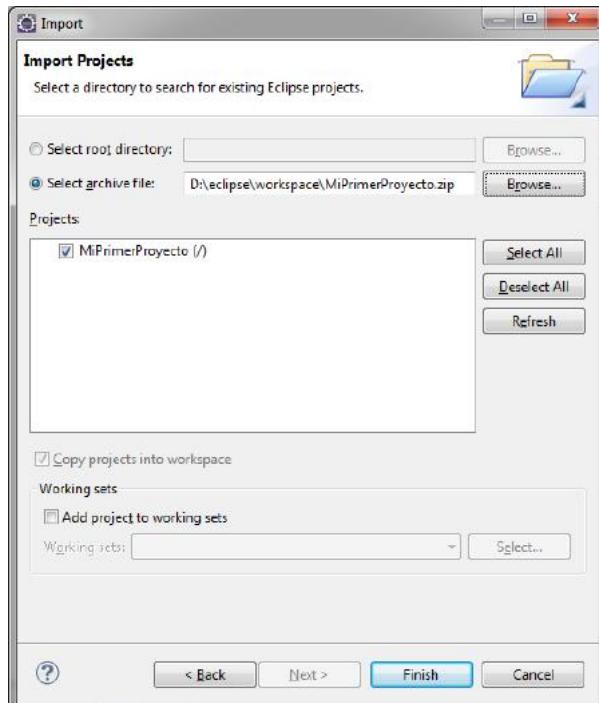


Figure 6: Importing a project from an external .zip file.

**WARNING:** To successfully import a previously exported project you must be sure that, when the project is deleted after exporting, the files in the hard disk that are part of the project have also been deleted.

### 3.1.3 Libraries

The most common way of using third party code is by using libraries. A library is a set of compiled code (structured in classes and, probably, including dependencies with other software components) providing a given functionality, developed and deployed by a third party. A library in Java is a .jar file, i.e.: a kind of compressed file containing compiled code (and, sometimes, also source code) ready to be run on any Java environment.

### 3.1.3.1 Creating a library

Any Java project can be included in a .jar file to be deployed to third-parties. The steps to create a library from a Java project are:

1. Right-click on the name of the project and select the Export option (see Figure 8)
2. Select the option Java >> JAR file. The dialog window shown in Figure 7 will appear.
3. Select the resources you want to export and the .jar file where you want to store the result.

### 3.1.3.2 Using a third-party library

The use of libraries in Eclipse is managed in the option menu Project >> Properties. A dialog box, shown in Figure 8, appears with several configuration sections related to the project. The library management section can be reached by clicking on ‘Java Build Path’ and select the ‘Libraries’ tab. To include an external library in your project follow these steps:

1. Include the JAR file with the library you want to use in a folder in your project (it is recommended to create a ‘lib’ folder in your project for this purpose).
2. Click on ‘Add JARs ...’

Now, you can make reference to classes included in that library by inserting the corresponding ‘import’ directive in your source code.

So, there are two ways of using external code in our programs: by importing a project (if we have access to the source code) and by using libraries. The preferred way to use third-party code in our programs is by using libraries, even if you have access to the source code. Importing a project is needed when you need to change the source code of the third-party library but it is not the usual situation.

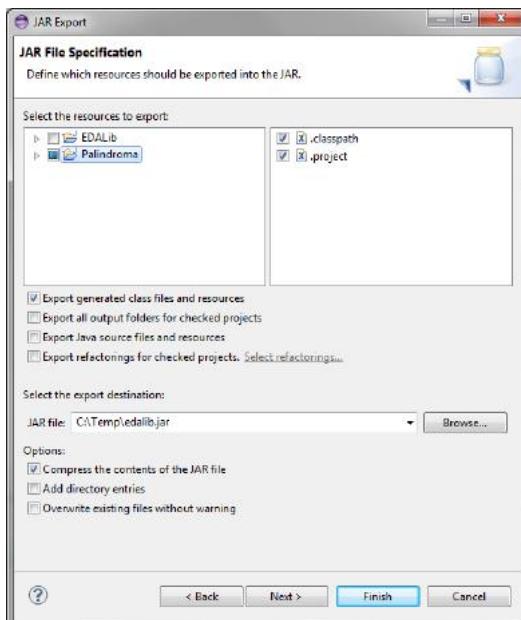


Figure 7. Creating a JAR file for a given project.

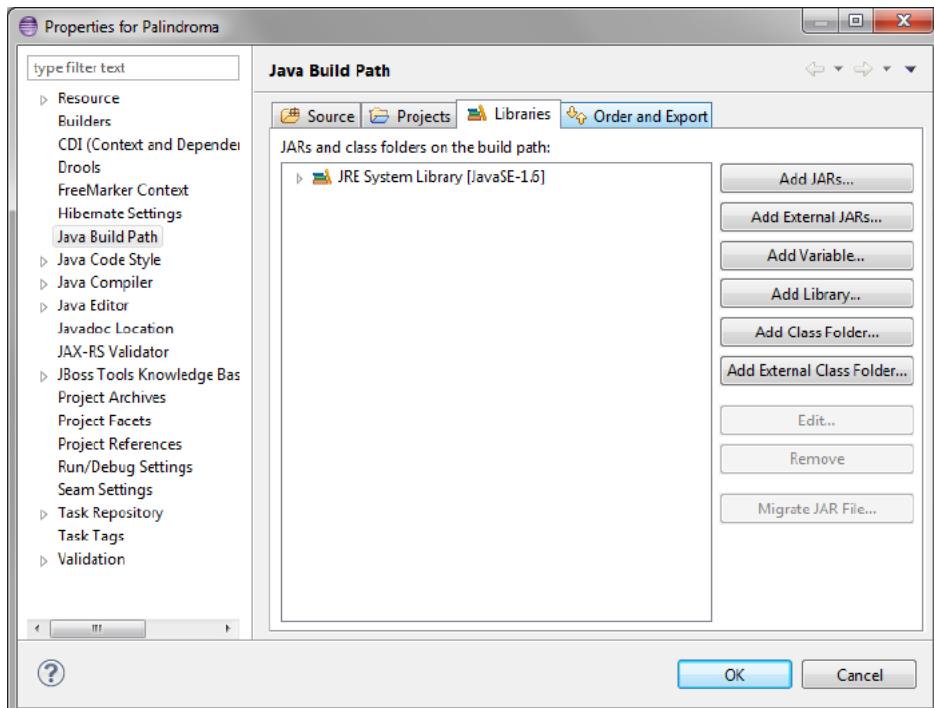


Figure 8. Importing a JAR file to be used in a project.

## 4. Related additional links

<http://eclipsutorial.sourceforge.net/totalbeginner.html>  
[http://eclipsutorial.sourceforge.net/Total\\_Beginner\\_Companion\\_Document.pdf](http://eclipsutorial.sourceforge.net/Total_Beginner_Companion_Document.pdf)  
<http://eclipsutorial.sourceforge.net/totalbegginer01/lesson01.html>

## 5. Exercise. Make your work environment ready

This exercise will be devoted to get familiar with the Eclipse IDE:

1. Create a project called *EDALab*. This project could be used to store the different exercises that will be developed during lab sessions.
2. Create a package called 'maths'.
3. Create a class called Calculator and provide source code for the methods 'sum'.
4. Create a main method in Calculator class to sum two numbers given as arguments. What is the difference between using a main method and not using it?
5. Export your project to a .zip file
6. Import your project from the .zip file (remember that, before importing, you must delete the project from Eclipse).
7. Export your project as a library (for example you can name it 'maths').
8. Create a new project called *OperationsProject*.
9. Implement a class called 'Addition' with a method called 'add' to obtain the addition of two numbers by using the 'maths' library.
10. Remember: you are working in a classroom. Many people use this classroom and the computers available. You will likely not find your source code the next day you come back to class so, SAVE YOUR work frequently and, again, before leaving the class!! DO NOT FORGET TO carry your work with you (export the project in a .zip file and save it in a secure place).

11. It is recommended for you to test the exporting process by importing your project again (remember that, before that, you must delete the project from Eclipse).

## JAVA Naming Conventions

Algorithms & Data Structures (ADS)

# Contents

---

- ▶ **Files**
- ▶ **Comments**
- ▶ **Indentation**
- ▶ **Declarations**
- ▶ **Naming conventions**

# Files

---

► Java source files have the following ordering:

1. Beginning comments
2. Package and Import statements
3. Class and interface declarations

## Comments

---

All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 *
 */
```

The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```



# Indentation

---

- ▶ Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.
- ▶ **Wrapping Lines**
- ▶ When an expression will not fit on a single line, break it according to these general principles:
  - ▶ Break after a comma.
  - ▶ Break before an operator.
  - ▶ Prefer higher-level breaks to lower-level breaks.
  - ▶ Align the new line with the beginning of the expression at the same level on the previous line.
  - ▶ If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

## Indentation. Examples

---

```
someMethod(longExpression1, longExpression2, longExpression3,  
          longExpression4, longExpression5);  
  
var = someMethod1(longExpression1,  
                  someMethod2(longExpression2,  
                             longExpression3));
```

## Indentation. Examples (II)

---

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
    + 4 * longname6; // PREFER
```

```
longName1 = longName2 * (longName3 + longName4
    - longName5) + 4 * longname6; // AVOID
```

## Indentation. Examples (III)

---

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();           //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

# Declarations

---

- ▶ One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
```

```
int size; // size of table
```

- ▶ is preferred over

```
int level, size;
```

- ▶ Do not put different types on the same line.  
Example:

```
int foo, foo array[]; //WRONG!
```

# Declarations

---

- ▶ Try to initialize local variables where they're declared.  
The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first. Example:

```
int count;  
...  
myMethod() {  
    if (condition) {  
        int count = 0; // AVOID!  
    ...}  
...}
```

# Declarations

---

- ▶ **Classes and interfaces**
  - ▶ No space between a method name and the parenthesis "(" starting its parameter list
  - ▶ Open brace "{" appears at the end of the same line as the declaration statement
  - ▶ Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

# Naming conventions

---

<b>Packages</b>	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
	Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	

# Naming conventions

---

## Classes

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

```
class Raster;  
class ImageSprite;
```

## Interfaces

Interface names should be capitalized like class names.

```
interface RasterDelegate;  
interface Storing;
```

## Methods

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

```
run();  
runFast();  
getBackground();
```

# Naming conventions

---

## Variables

Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters.

```
int i;  
char c;  
float myWidth;
```

Variable names should not start with underscore \_ or dollar sign \$ characters, even though both are allowed.

Variable names should be short yet meaningful. The choice of a variable name should be mnemonic—that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

# Naming conventions

---

## Constants

The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("\_"). (ANSI constants should be avoided, for ease of debugging.)

```
static final int MIN_WIDTH = 4;  
static final int MAX_WIDTH = 999;  
static final int GET_THE_CPU = 1;
```

## **Parte III**

### **Tema 1. Tipos Abstractos de Datos**



## Tema 1. Tipo Abstracto de Datos (TAD)

Estructura de Datos y Algoritmos (EDA)

## Objetivos

---

- ▶ Entender el concepto de **abstracción**
- ▶ Entender los conceptos de **Tipo Abstracto de Datos (TAD)** and **estructura de datos** y explicar la diferencia entre ambos
- ▶ Aprender **cómo especificar formalmente** un TAD
- ▶ Escribir **implementaciones correctas** para TAD simples

# Contenidos

---

- ▶ Abstracción
- ▶ Tipo Abstracto de Datos (TAD)

## Abstracción

- ▶ Centrarse en los detalles relevantes para un problema y descartar detalles irrelevantes



Student  
...  
NotaAcceso  
ListOfSubjects

Patient  
...  
High  
ListOfAllergies  
ListOfSurgeries

Permitir definir software (**qué debe hacer** sin decir **cómo debe hacerse**)

## Tipo Abstracto de Datos (TAD)

---

- ▶ Un modelo matemático con una colección de operaciones definidas en ese modelo
  - ▶ Ejemplo: conjunto de enteros con las operaciones de unión, intersección y diferencia
- ▶ Una estructura de datos es una representación (**implementación**) de un TAD en un lenguaje de programación. Las estructuras de datos a menudo son colecciones de variables de diferentes tipos de datos
- ▶ Un TAD puede tener varias implementaciones (estructuras de datos)

## Especificación

---

- ▶ Un TAD puede describirse usando dos maneras diferentes:
  - ▶ **Especificación No-Formal** (usando lenguaje natural)
  - ▶ **Especificación Formal** (usando pseudo-código o incluso algún lenguaje de programación)
  - ▶ En JAVA, las interfaces se utilizan para definir los TAD

## TAD complejo (Especificación informal)

---

Un número complejo se puede representar como un par de números: números reales e imaginarios. Este TAD implica las siguientes operaciones:

- ▶ **sum(Complex):**

$$(a+ib) + (c+id) = (a+c) + (b+d)i$$

- ▶ **times(Complex):**

$$(a+ib) * (c+id) = (ac-bd) + (ad+bc)i$$

- ▶ **module():** devuelve el módulo del número complejo.

$$|a+ib| = \sqrt{a^2+b^2}$$

# Especificación informal

---

## Operaciones

- ▶ **getReal ()**: devuelve su número real  
*(a+ib->a)*
- ▶ **getImag()**: devuelve su número imaginario  
*(a+ib->b)*
- ▶ **setReal(x)**: toma un número x como entrada y lo asigna a su número real
- ▶ **setImag(x)**: toma un número x como entrada y lo asigna a su número imaginario

## Especificación formal de un TAD Complejo

---

```
public interface iComplex {  
  
    public float getReal();  
    public float getImag();  
  
    public void setReal(float x);  
    public void setImag(float x);  
  
    public float module();  
  
    public iComplex sum(iComplex obj);  
    public iComplex times(iComplex obj);  
  
    public boolean equals(iComplex obj);  
}
```

## Estructuras de datos: implementaciones de TAD

---

- ▶ En Java, las estructuras de datos se implementan como clases
- ▶ Una estructura de datos (class) debe implementar todas las operaciones definidas en su TAD (interfaz)
- ▶ Una class describe cómo se realizan las operaciones de TAD

```
public class Complex implements iComplex {  
  
    private float real;  
    private float imag;  
    public Complex(float r, float i) {  
        real=r;  
        imag=i;  
    }  
    public float getReal() {  
        return real;  
    }  
    public float getImag() {  
        return imag;  
    }  
    public void setReal(float x) {  
        real=x;  
    }  
    public void setImag(float x) {  
        imag=x;  
    }  
}
```

La clase debe implementar todos los métodos definidos en la interfaz

```
public iComplex sum(iComplex obj) {  
    iComplex result=new Complex(real+obj.getReal(),imag+obj.getImag());  
    return result;  
}  
  
public iComplex times(iComplex obj) {  
    iComplex result=new Complex(real*obj.getReal()-imag*obj.getImag(),  
        real*obj.getImag()+imag*obj.getReal());  
    return result;  
}  
  
public boolean equals(iComplex obj) {  
    return (real==obj.getReal() && imag==obj.getImag());  
}
```

## TAD Fecha (especificación informal)

---

- ▶ Datos para representar una Fecha (*Date*)
- ▶ **Operaciones:**
  - ▶ **before(*Date d*):** devuelve verdadero si la fecha de invocación es anterior a la especificada por *d*, de lo contrario, devuelve falso.
  - ▶ **after(*Date d*):** devuelve verdadero si la fecha de invocación es posterior a la especificada por *d*, de lo contrario, devuelve falso
  - ▶ **compareTo(*Date d*):** devuelve 0 si las fechas son iguales.  
Devuelve -1 si la fecha de invocación es anterior a *d*. Devuelve 1 si la fecha de invocación es posterior a *d*.

# TAD Fecha (especificación informal)

---

## ► Operaciones:

- ▶ **getDay()**: devuelve el día de la fecha de invocación.
- ▶ **getMonth()**: devuelve el mes de la fecha de invocación
- ▶ **getYear()**: devuelve el año de la fecha de invocación
- ▶ **show()**: muestra la fecha en el formato dd-mm-aaaa

## Especificación formal TAD iDate

```
public interface iDate {  
  
    public int getDay();  
    public int getMonth();  
    public int getYear();  
  
    public boolean before(iDate d);  
    public boolean after(iDate d);  
    public int compareTo(iDate d);  
    public void show();  
}
```

```
public class Date1 implements iDate {  
  
    int d;//day  
    int m;//month  
    int y;//year  
  
    //To simplify, we suppose that the date is right  
    public Date1(int d, int m, int y) {  
        this.d=d;  
        this.m=m;  
        this.y=y;  
    }  
  
    public int getDay() {  
        return d;  
    }  
}
```

```
public boolean before(iDate obj) {  
    if (y<obj.getYear()) return true;  
    else if (y==obj.getYear()) {  
        if (m<obj.getMonth()) return true;  
        else if (m==obj.getMonth()) {  
            if (d<obj.getDay()) return true;  
            else return false;  
        } else return false;  
    }  
    //y>obj.getYear()  
    return false;  
}
```

Encuentra el código incompleto de la clase en aulaglobal. Implementa los otros métodos

## Preguntas ???



Otra  
implementación  
para iDate???

Con solo 2  
atributos???

```
public class Date2 implements iDate {  
  
    int day; //number of days from 1st January.  
            //Ex: 32 is Feb 1.  
    int year;  
  
    public Date2(int d, int y) {  
        day=d;  
        year=y;  
    }  
  
    public int getYear() {  
        return year;  
    }  
  
    //To simplify we suppose there is no leap years.  
    public int getDay() {  
        if (day<=31) return day;  
        if (31<day && day<=59) return day-31;  
        //complete  
    }  
}
```

Encuentra el código incompleto de la clase en aulaglobal. Implementa los otros métodos.

## Resumen

---

- ▶ Un TAD define **qué operaciones**, pero no **cómo hacerlas** (implementar)
- ▶ Un TAD puede tener diferentes implementaciones
- ▶ En Java, las **interfaces** permiten definir formalmente TAD
- ▶ En Java, las **clases** (**classes**) permiten implementar TAD
- ▶ Una clase (**class**) que implementa una interfaz debe implementar todos los métodos de la interfaz

# **Parte IV**

## **Tema 2. TADS**



## **Tema 2.1 TAD lineales**

Estructura de Datos y Algoritmos (EDA)

# Contenidos

---

- ▶ **2.1. ¿Qué es un TAD Lineal?**
- ▶ 2.2. TAD Pila
- ▶ 2.3. TAD Cola
- ▶ 2.4. TAD Lista
  - ▶ 2.4.1 Implementación con una Lista Simplemente Enlazada
  - ▶ 2.4.2 Implementación con una Lista Dblemente Enlazada

# Objetivos

---

- ▶ Al final de la clase, los estudiantes deben ser capaces de:
  - ▶ Definir un TAD lineal
  - ▶ Explicar las principales ventajas y desventajas de usar arrays para implementar un TAD lineal
  - ▶ Explicar cómo las estructuras de datos dinámicas son una alternativa a los arrays
  - ▶ Implementar una Cola usando una estructura dinámica
  - ▶ Explicar el concepto de nodo e implementarlo como una clase

# TAD LINEAL

---

- ▶ Representa una secuencia de elementos de algún tipo
- ▶ Ejemplos:
  - ▶ Nombre se personas (strings): María, Pepe, Juan, ...
  - ▶ Números enteros: 5,6,1,-3,0,2,1,...
  - ▶ Objetos (instancias) de la clase Punto
  - ▶ Objetos (instancias) de la clase Empleado
- ▶ Todos los elementos deben pertenecer al mismo tipo de datos

# Arrays

---

- ▶ ¡¡¡Felicitaciones!!!. ;Ya has implementado una Lista usando un array! (primer trabajo semanal)
- ▶ Una array se presenta en posiciones consecutivas en la memoria



# Arrays

---

- ▶ Ejemplo: un array (tamaño 6) of números enteros.
- ▶ Un entero toma 4 bytes en espacio de memoria.
- ▶ Si se conoce la dirección inicial de un array y el índice de un elemento, puede calcular fácilmente su dirección

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	15	7	11	44	93	20
Address with respect to the Array (Subscript)	0	1	2	3	4	5

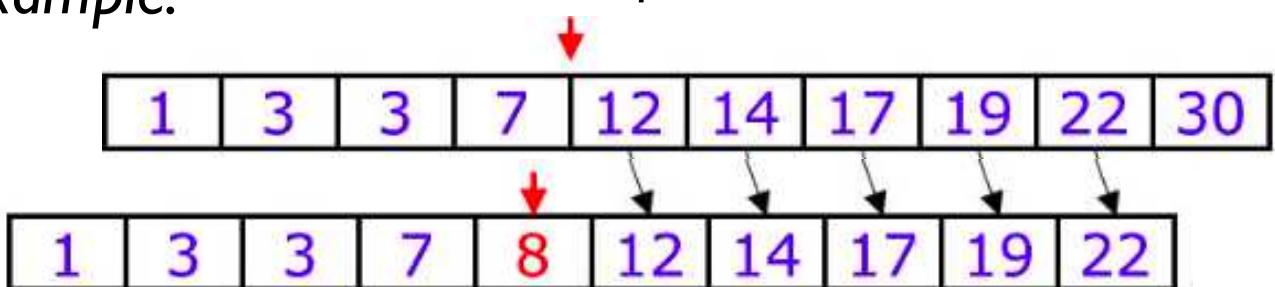


# Arrays

- ▶ **Ventajas:** acceso directo a cada elemento del array a través de su índice.
- ▶ **Desventajas:**
  - ▶ Las operaciones de borrado e inserción pueden requerir que algunos elementos se muevan. Estas operaciones son **lentas**.

*Example:*

*insert 8 at position 4*



- ▶ *Es mas, tenemos que descartar 30!!!*

# Arrays

---

## ► Desventajas:

- ▶ Los arrays tienen tamaños fijos (no se pueden modificar en tiempo de ejecución)
- ▶ ¡Algunas veces el tamaño puede ser insuficiente!
- ▶ Un tamaño muy grande puede conllevar a un uso ineficiente de la memoria
- ▶ A veces no se puede saber el tamaño necesario para tu problema

# Arrays

---

- ▶ Ejemplos:
  - ▶ Un array para almacenar la temperatura media diaria del último año. Tu puedes usar un array de tamaño 365 (366 es un año bisiesto).
  - ▶ Sin embargo, ¿qué tamaño necesitas para almacenar las puntuaciones del juego Candy Crush?

# Estructuras de datos dinámicas

---

- ▶ Los elementos no se almacenan en posiciones de memoria consecutivas, sino también con espacios entre ellos.
- ▶ Puede crecer o reducirse en tiempo de ejecución
- ▶ Uso eficiente de la memoria (solo pueden ocupar la memoria necesaria)
- ▶ Alternativa a los arrays para implementar TAD lineales. Vamos a verlo !!!

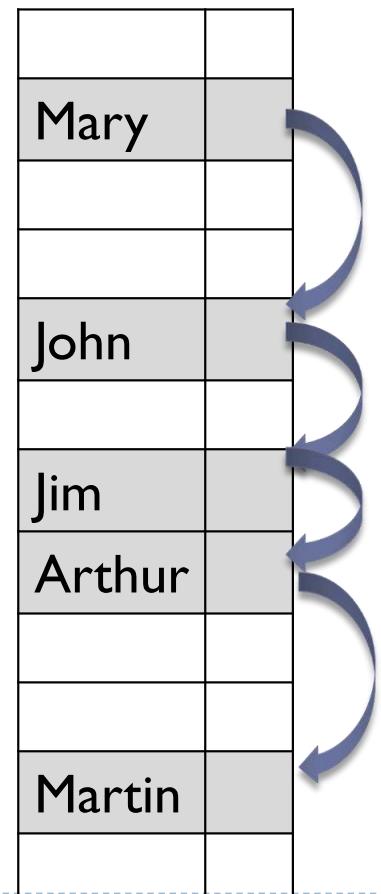
## Array vs Estructura dinámica

**Array**

Mary
John
Jim
Arthur
Martin

memoria

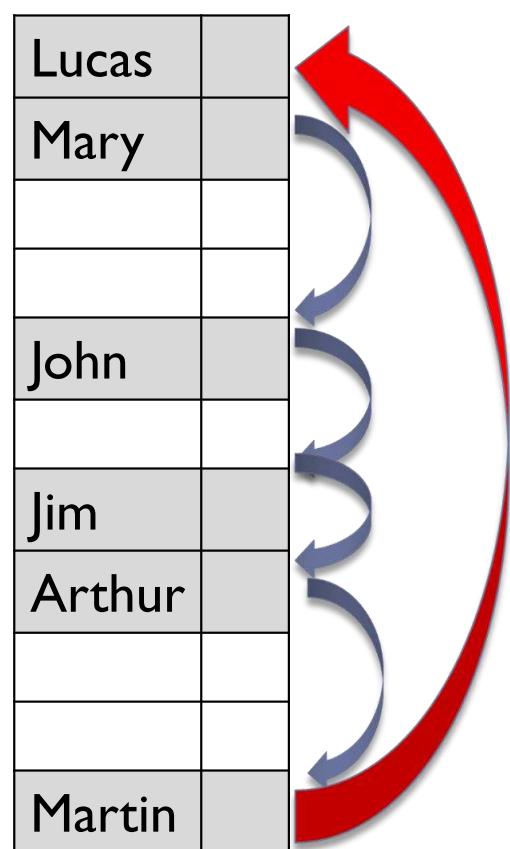
**Estructura dinámica**



## Implementación de un TAD lineal utilizando una estructura dinámica

### Estructura dinámica

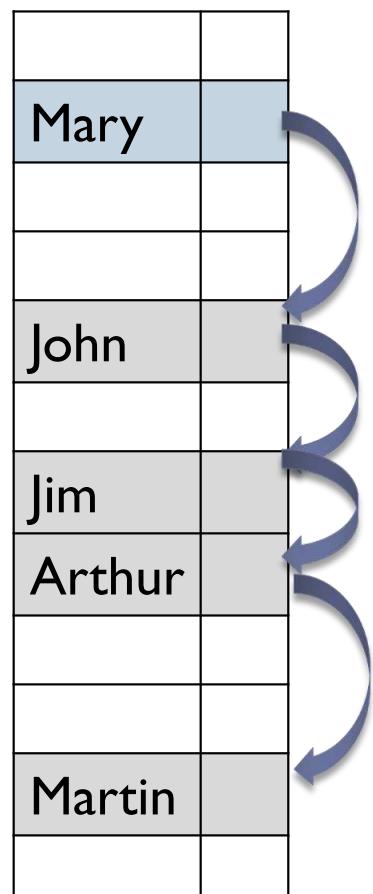
Los espacios en la memoria permiten que las órdenes físicas y lógicas puedan ser diferentes



## Implementación de un TAD lineal utilizando una estructura dinámica

### Estructura dinámica

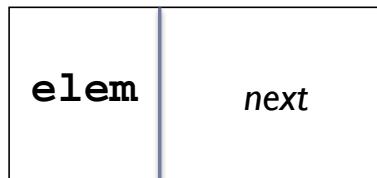
Cada ubicación no solo almacena (una referencia a) un objeto, sino también la referencia (dirección) a su sucesor en la Lista



## Implementación de un TAD lineal utilizando una estructura dinámica

- ▶ Necesitamos definir una clase, **Node**, para representar cada ubicación. La clase **Node** tiene dos atributos:
  - ▶ **elem**: es la referencia a un elemento almacenado en la Lista. Es decir, almacena la dirección de memoria donde se almacena el elemento. El tipo de datos de **elem** debe ser del mismo tipo que los elementos de la Lista
  - ▶ **next**: es la referencia al nodo que almacena el siguiente elemento en la Lista. Su tipo de datos debe ser **Node**

Node



## Class Snode (simplemente node)

```
public class SNode {  
  
    public String elem;  
    public SNode next;  
  
    public SNode(String e) {  
        elem = e;  
    }  
}
```

Permite almacenar un objeto de tipo String.  
Sin embargo, puede definir un node que  
almacene cualquier tipo de objeto

## **Tema 2.2 TAD lineales**

### **TAD Pila**

Estructura de Datos y Algoritmos (EDA)



# Contenidos

---

2.1. ¿Qué es un TAD Lineal?

**2.2. TAD Pila**

2.3. TAD Cola

2.4. TAD Lista

2.4.1 Implementación con una Lista Simplemente Enlazada

2.4.2 Implementación con una Lista Dblemente Enlazada

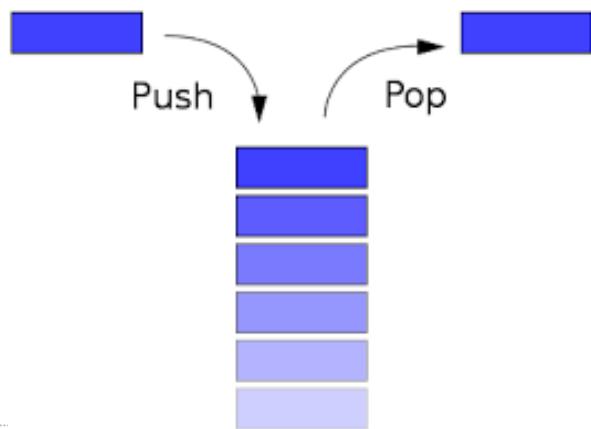
# Objetivos

---

- Al final de la clase, los estudiantes deben ser capaces de:
  1. Entender el principio LIFO (last-in, first-out)
  2. Explicar el funcionamiento de una Pila
  3. Especificar formalmente una Pila
  4. Implementar una Pila utilizando una estructura dinámica

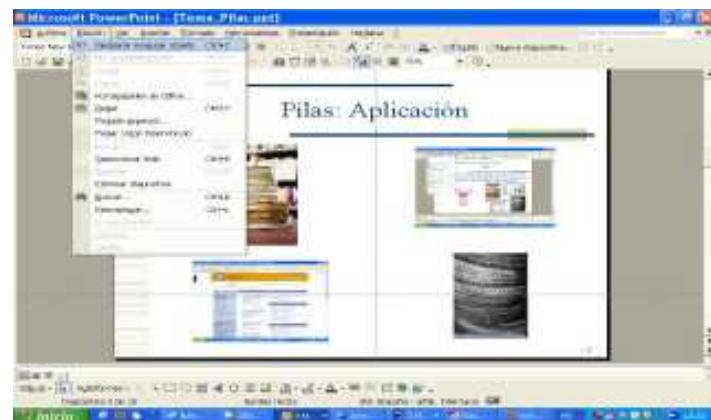
# TAD Pila: Introducción

- ▶ Estructura de datos lineal basada en el principio **LIFO (last-in, first-out)**
- ▶ Las operaciones de inserción (**push**) y borrado (**pop**) se realizan en una sola posición, que es conocido como **top** o **peak**



# Aplicación de Pilas

## Undo operaciones



## Back Navigation

# Operaciones

---

- ▶ **push(e)**: añade un objeto e a la cima de la Pila
- ▶ **pop()**: borra y devuelve el elemento en la cima de la Pila
- ▶ **top()**: devuelve el elemento de la cima de la Pila
- ▶ **size()**: devuelve el número de elementos almacenados en la Pila.
- ▶ **isEmpty()**: devuelve *true si la Pila está vacía; en otro caso, false.*

## Ejemplo

---

Operation	Stack	Output
push('h')	(h)	-
push('e')	(h,e)	-
top()	(h,e)	e
push('l')	(h,e,l)	-
push('l')	(h,e,l,l)	-
push('o')	(h,e,l,l,o)	-
top()	(h,e,l,l,o)	o
push('!')	(h,e,l,l,o,!)	-
top()	(h,e,l,l,o,!)	!
size()	(h,e,l,l,o,!)	6
isEmpty()	(h,e,l,l,o,!)	false
pop()	(h,e,l,l,o)	!

## TAD Pila

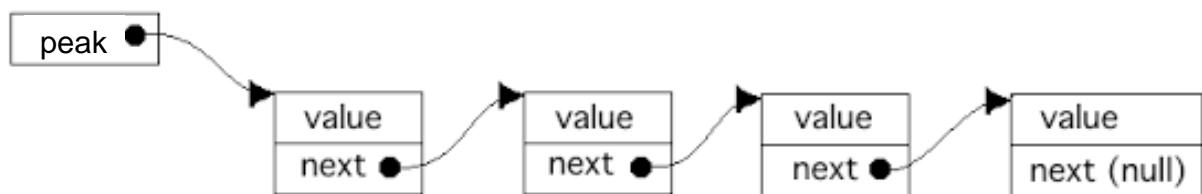
---

```
public interface IStack {  
    public boolean isEmpty();  
    public void push(Integer elem);  
    public Integer pop();  
    public Integer top();  
    public int getSize();  
}
```

# Implementación de un TAD Pila usando una estructura dinámica

---

- Una Pila se puede representar como una secuencia de nodos
  - Solo permitimos insertar, borrar o leer el primer elemento (peak) de la secuencia



## Clase SNode

---

```
public class SNode {  
  
    public Integer elem;  
    public SNode next;  
  
    public SNode(Integer e) {  
        elem = e;  
    }  
}
```

## Implementación de un TAD Pila usando una estructura dinámica

```
public class SStack implements IStack {  
    SNode peak = null;  
    int size;  
  
    public boolean isEmpty() {  
        return peak == null;  
    }
```

Java proporciona un constructor por defecto que crea una Pila vacía (peak es nulo).

## push() method

---

```
public void push(Integer newElem) {  
    SNode newNode = new SNode(newElem);  
    newNode.next = peak;  
    peak = newNode;  
    size++;  
}
```

## top() method

---

```
public Integer top() {  
    if (isEmpty()) {  
        System.out.println("The stack is empty.");  
        return null;  
    }  
    return peak.elem;  
}
```

## pop() method

---

```
public Integer pop() {  
    if (isEmpty()) {  
        System.out.println("The stack is empty.");  
        return null;  
    }  
    Integer elem = peak.elem;  
    peak = peak.next;  
    size--;  
    return elem;  
}
```

# **Tema 2.3 TAD lineales**

## **TAD Cola**

Estructura de Datos y Algoritmos (EDA)

# Contenidos

2.1. ¿Qué es un TAD Lineal?

2.2. TAD Pila

**2.3. TAD Cola**

2.4. TAD Lista

    2.4.1 Implementación con una Lista Simplemente Enlazada

    2.4.2 Implementación con una Lista Dblemente Enlazada

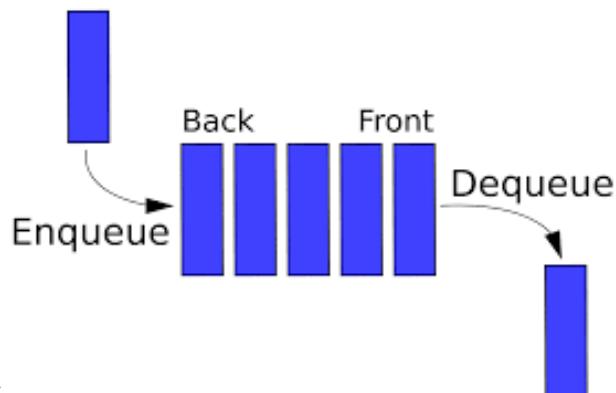
## Objetivos

---

- ▶ Al final de la clase, los estudiantes deben ser capaces de:
  - ▶ Entender el principio FIFO (first-in, first-out)
  - ▶ Explica las operaciones de una Cola.
  - ▶ Especificar formalmente una Cola
  - ▶ Implementar una Cola usando una estructura dinámica

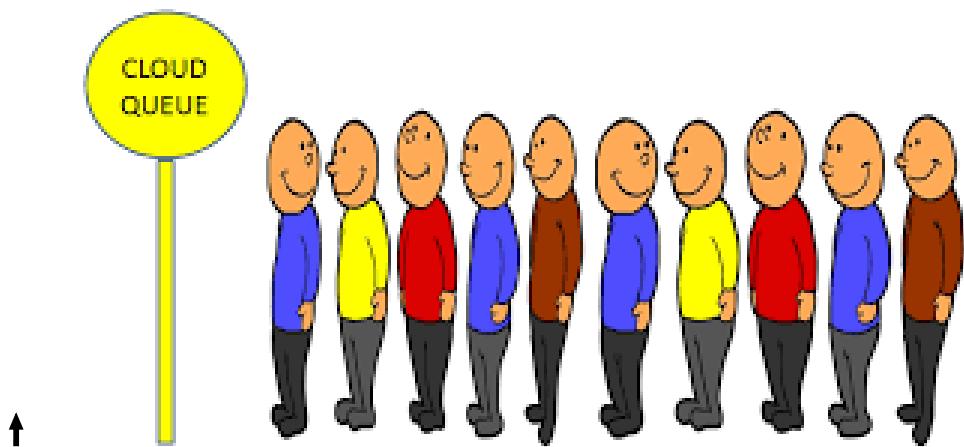
# TAD Cola: Introducción

- ▶ Estructura de datos lineal basada en el principio **FIFO (first-in, first-out)**
- ▶ Eliminamos (**dequeue**) el primer elemento de la Cola.
- ▶ Insertamos (**enqueue**) un nuevo elemento al final de la Cola



# TAD Cola

---



## TAD Cola. La Cola en impresión



Estado de impresión de documentos ([mis trabajos](#))

Archivo Trabajo Ver

Trabajo	Documento	Impresora	Tamaño	Hora de envío	Estado
2547	MARIBEL	HP-Laserj...	123k	hace un minuto	Pendiente
2546	DATOS EXC...	HP-Laserj...	145k	hace un minuto	Pendiente
2545	cuestionari...	HP-Laserj...	210k	hace un minuto	Pendiente
2544	Diario clas...	HP-Laserj...	11k	hace un minuto	Pendiente
2543	GUÍA BÁSIC...	HP-Laserj...	2476k	hace un minuto	Pendiente
2542	A9ROXbjSdh	HP-Laserj...	4325k	hace 2 minutos	Procesando



# Operaciones

---

- ▶ **enqueue (Object e)**: añade el elemento e al final de la cola
- ▶ **dequeue()**: borra y devuelve el primer elemento de la cola
- ▶ **front()**: devuelve el primer elemento de la cola
- ▶ **isEmpty()**: devuelve true si la cola está vacía; en otro caso, false
- ▶ **getSize()**: devuelve el número de elementos de la cola

# Ejemplo

<i>Operation</i>	<i>Output</i>	<i>Q</i>
enqueue(5)	-	(5)
enqueue(3)	-	(5, 3)
dequeue()	5	(3)
enqueue(7)	-	(3, 7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	(0)
dequeue()	"error"	(0)
isEmpty()	true	(0)
enqueue(9)	-	(9)
enqueue(7)	-	(9, 7)
size()	2	(9, 7)
enqueue(3)	-	(9, 7, 3)
enqueue(5)	-	(9, 7, 3, 5)
dequeue()	9	(7, 3, 5)



## Tad Cola

---

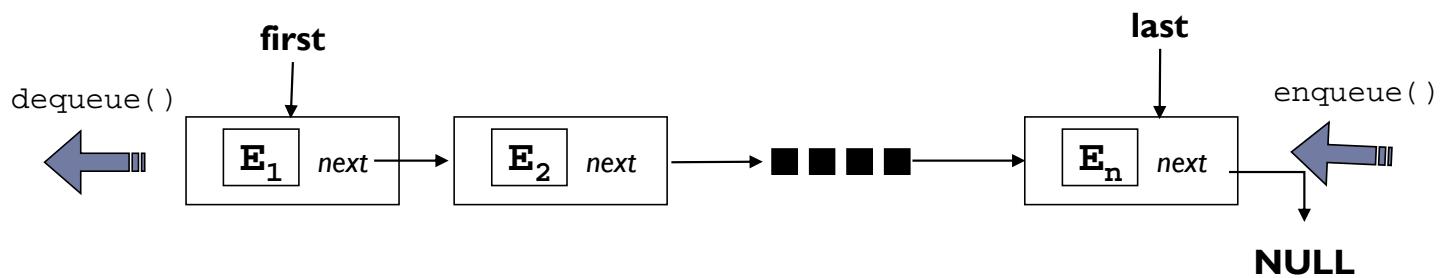
```
public interface IQueue {  
  
    public boolean isEmpty();  
    public void enqueue(String elem);  
    public String dequeue();  
    public String front();  
    public int getSize();  
}
```

En estas diapositivas, usamos una cola de objetos String, pero recuerda que puede definir una cola de cualquier tipo de datos



# Implementación de un TAD Cola usando una estructura dinámica

- Una cola puede ser representado como una secuencia de nodos
  - Solo permitimos eliminar el primer elemento en la cola
  - Solo permitimos insertar al final de la cola



## Implementación de un TAD Cola usando una estructura dinámica

---

```
public class SQueue implements IQueue {  
  
    private SNode first;  
    private SNode last;  
    int size;  
  
    public boolean isEmpty() {  
        return first == null;  
    }  
}
```



## front() method

---

```
public String front() {  
    if (isEmpty()) {  
        System.out.println("Queue is empty!");  
        return null;  
    }  
    return first.elem;  
}
```

## enqueue() method

```
public void enqueue(String elem) {  
    SNode node = new SNode(elem);  
    if (isEmpty()) {  
        first = node;  
    } else {  
        last.next = node;  
    }  
    last = node;  
    size++;  
}
```

## dequeue() method

---

```
public String dequeue() {  
    if (isEmpty()) {  
        System.out.println("Queue is empty!");  
        return null;  
    }  
    String firstElem = first.elem;  
    first = first.next;  
    if (first == null) {  
        last = null;  
    }  
    size--;  
    return firstElem;  
}
```

## **Tema 2.4.1 TAD lineales**

### **TAD Lista: Implementación con una Lista Simplemente Enlazada**

Estructura de Datos y Algoritmos (EDA)

# Contenidos

---

- ▶ 2.1. ¿Qué es un TAD Lineal?
- ▶ 2.2. TAD Pila
- ▶ 2.3. TAD Cola
- ▶ 2.4. TAD Lista
  - ▶ **2.4.1 Implementación con una Lista Simplemente Enlazada**
  - ▶ 2.4.2 Implementación con una Lista Dblemente Enlazada

## Objetivos

---

- ▶ Al final de la clase, los estudiantes deben ser capaces de:
  - ▶ Especificar formalmente un TAD Lista
  - ▶ Implementar un TAD Lista usando una Lista Simplemente Enlazada
  - ▶ Explicar qué ventajas del uso de last (referencia al último elemento (last) en la Lista)

# Clase SNode

```
public class SNode {  
  
    public String elem;  
    public SNode next;  
  
    public SNode(String e) {  
        elem = e;  
    }  
  
}
```

# TAD Lista

---

- ▶ Especificación formal:
  - ▶ Secuencia de elementos  $\{a_1, a_2, \dots, a_n\}$  donde cada elemento tiene un único predecesor (excepto el primero que no tiene predecesor) y un único sucesor (excepto el último que no tiene sucesor).
  - ▶ Las operaciones básicas de un TAD Lista son:
    - ▶ **Agregar** un elemento a la Lista
    - ▶ **Eliminar** un elemento de la Lista
    - ▶ **Consultar** un elemento de la Lista

## Especificación Formal de un TAD Lista (operaciones de agregar)

---

```
public interface IList {  
  
    public void addFirst(String newElem);  
  
    public void addLast(String newElem);  
  
    public void insertAt(int index, String newElem);
```

---

## Especificación Formal de un TAD Lista (operaciones de consulta)

```
public boolean isEmpty();  
public boolean contains(String elem);  
public int getSize();  
public int indexOf(String elem);  
public String getFirst();  
public String getLast();  
public String getAt(int index);  
public String toString();
```

## Especificación Formal de un TAD Lista (Operación de borrado)

---

`public void removeFirst();`

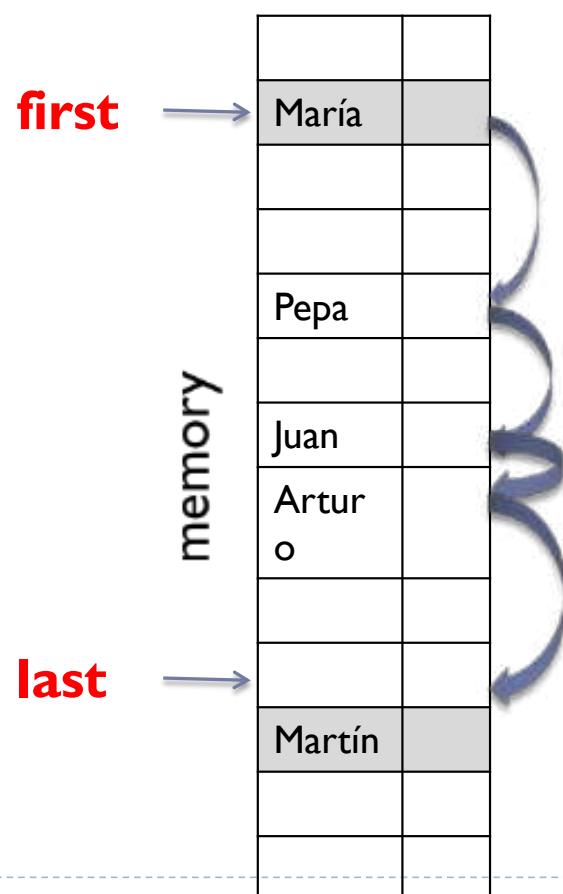
`public void removeLast();`

`public void removeAll(String elem);`

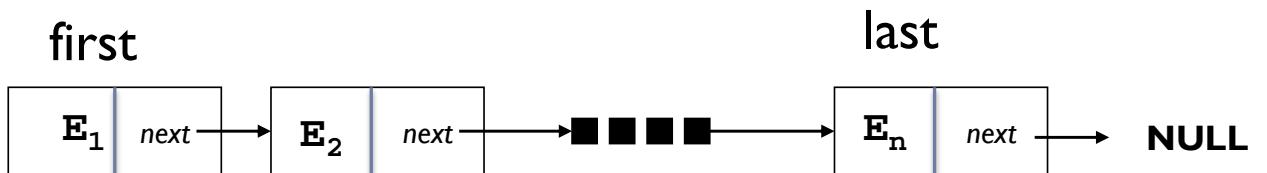
`public void removeAt(int index);`

---

# Implementación de un TAD Lista usando una Lista Simplemente Enlazada



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada



```
public class SList implements IList {  
    public SNode first;  
    public SNode last;  
    int size; //by default is 0
```



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método empty)

---

first → NULL

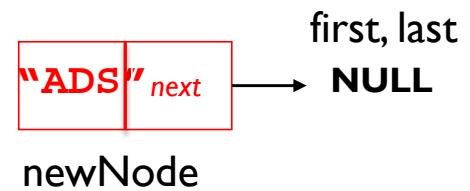
```
public boolean isEmpty() {  
    return (first == null);  
}
```



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addFirst)

Agrega “ADS” en una Lista vacía

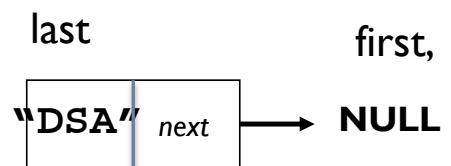
```
list.addFirst( "ADS" )
```



```
Snode newNode=new SNode( "ADS" );
newNode.next=first;
if (first==null) last=newNode;
first=newNode;
```

## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addFirst)

list.addFirst( "DSA" )

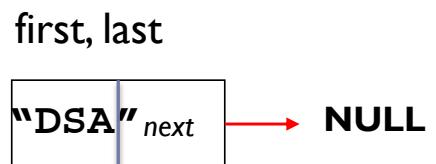


```
Snode newNode=new SNode( "DSA" );
newNode.next=first;
if (first==null) last=newNode;
first=newNode;
```

## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addFirst)

---

list.addFirst("DSA")

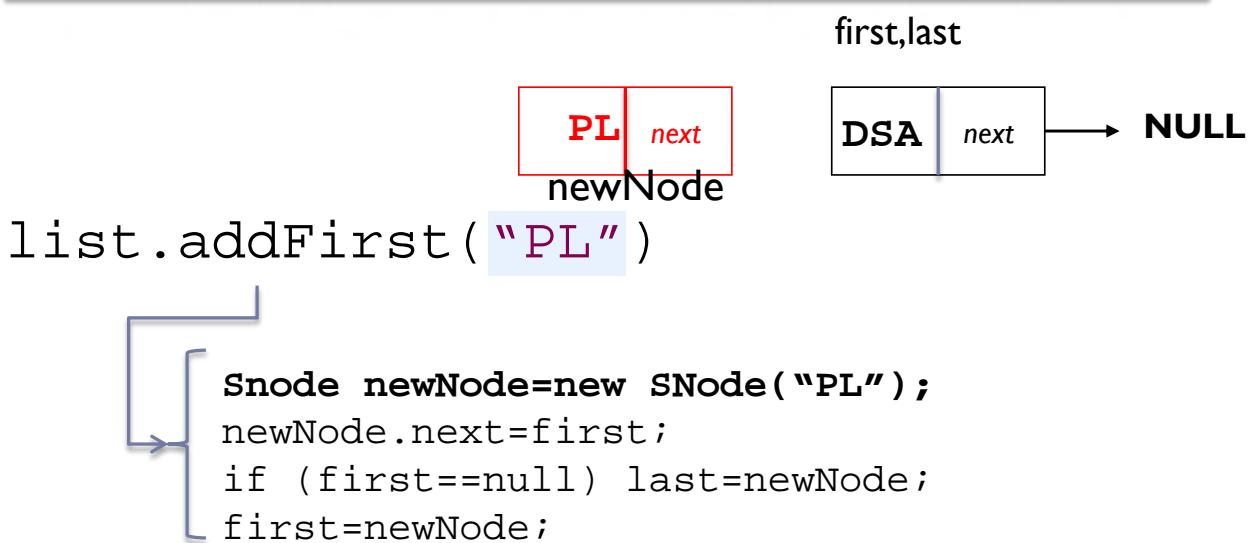


Snode newNode=new SNode( "DSA" );  
newNode.next=first;  
if (first==null) last=newNode;  
**first=newNode;**

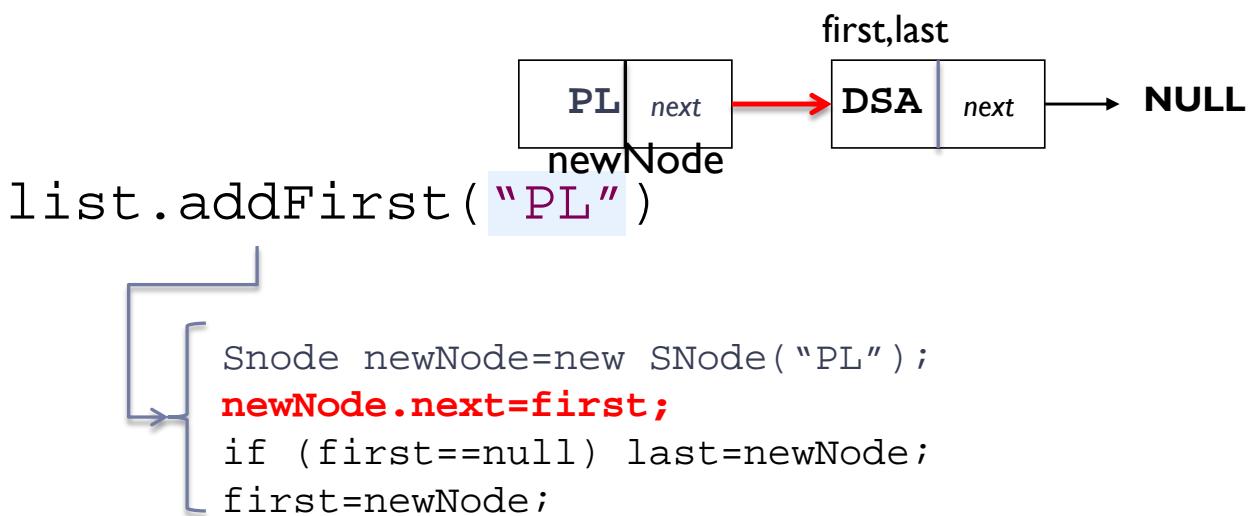


## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addFirst)

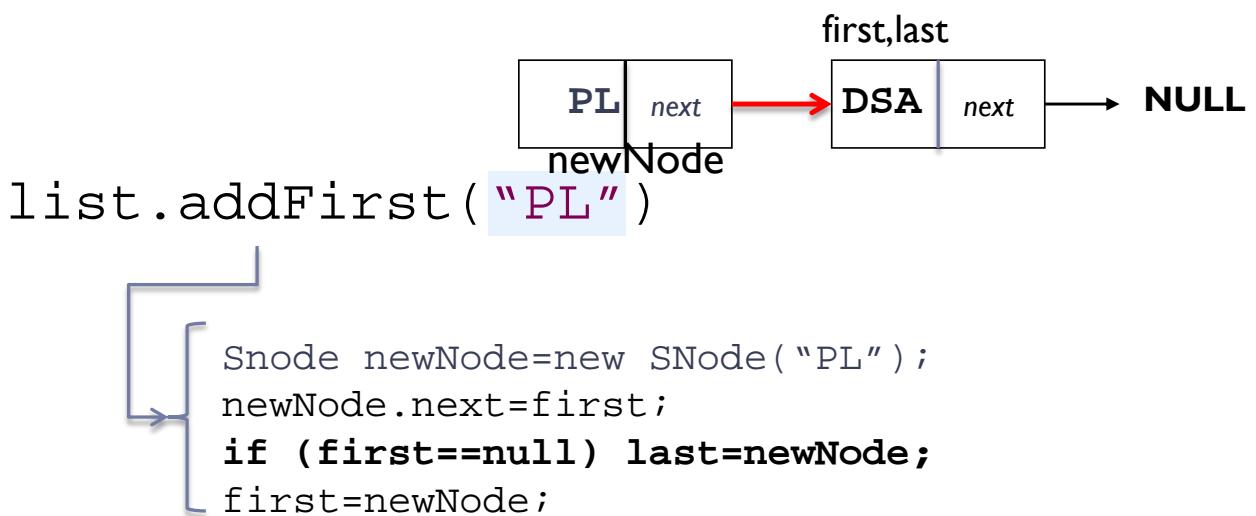
Ahora, agrega “PL” en el inicio de la Lista



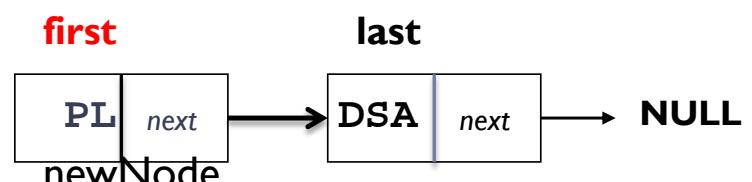
## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addFirst)



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addFirst)



# Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addFirst)



```
list.addFirst( "PL" )
```

```
Snode newNode=new SNode( "PL" );
newNode.next=first;
if (first==null) last=newNode;
first=newNode;
```

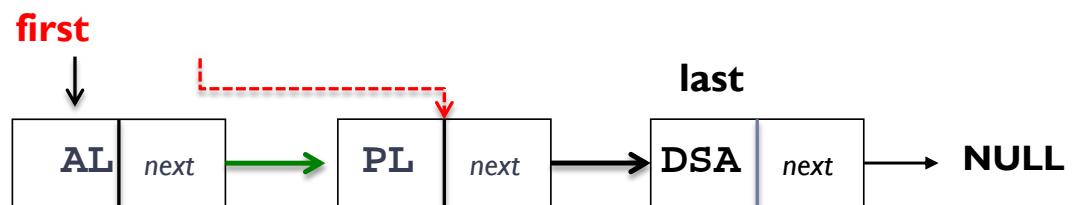
## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addFirst)

---

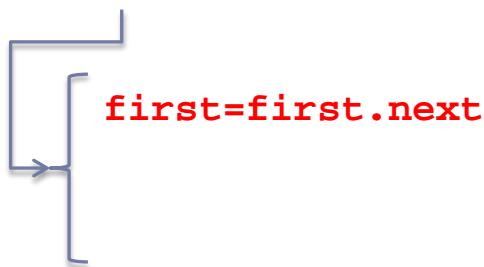
```
public void addFirst(String newElem) {  
  
    SNode newNode = new SNode(newElem);  
    newNode.next = first;  
    //If list is empty, last has also to reference to newNode  
    if (first==null) last=newNode;  
    //we set the new first node  
    first = newNode;  
    size++;  
}
```

---

## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeFirst)

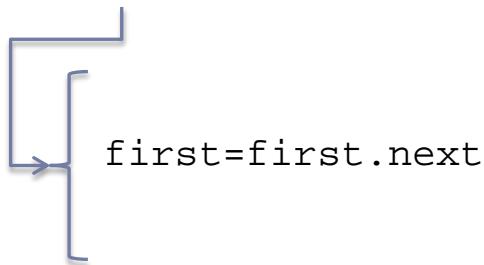


list.removeFirst( )



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeFirst)

```
list.removeFirst()
```

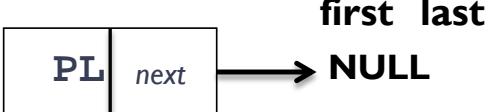


- ¿Funciona cuando la lista solo tiene un nodo?
- Si la lista está vacía, ¿qué sucede?

## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeFirst)

---

```
list.removeFirst()
```



```
first last  
PL | next → NULL
```

first=first.next  
If (first==null) last=null;

## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeFirst)

---

```
public void removeFirst() {  
    if (!isEmpty()) {  
        first = first.next;  
        if (first==null) last=null;  
        size--;  
    }  
}
```

¡¡Advertencia!! En realidad, el nodo no se elimina, solo es ignorado  
Java garballe collection lo eliminará

## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addLast)

---

```
list.addLast("HomeLand")
```

Homeland

Primer paso: crear un nuevo nodo  
para almacenar “HomeLand”

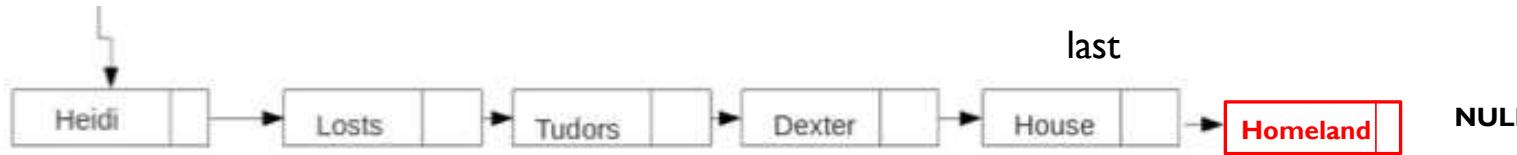
**SNode node = new SNode (newElem);**



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addLast)

```
lista.addLast("HomeLand")
```

first



Segundo paso: last.next debe hacer referencia al nuevo nodo

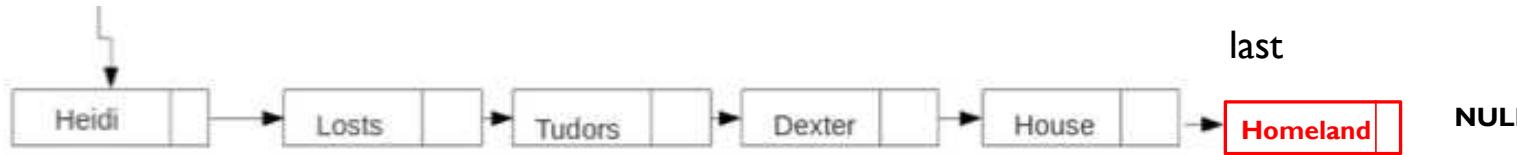
**last.next=newNode;**



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addLast)

```
lista.addLast("HomeLand")
```

first



Tercer paso: actualizar el último

**last=newNode;**

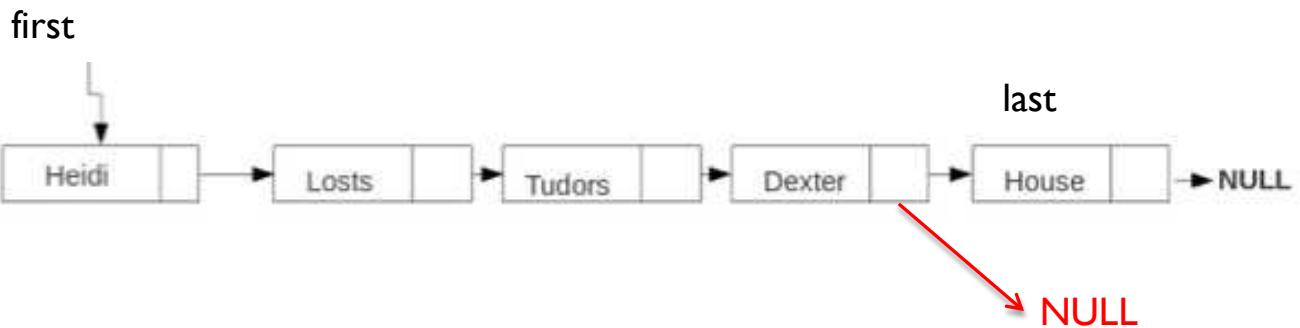


## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método addLast)

```
public void addLast(String newElem) {  
  
    SNode node = new SNode(newElem);  
    if (isEmpty()) addFirst(newElem);  
    else {  
        last.next=node;  
        last=node;  
        size++;  
    }  
}
```

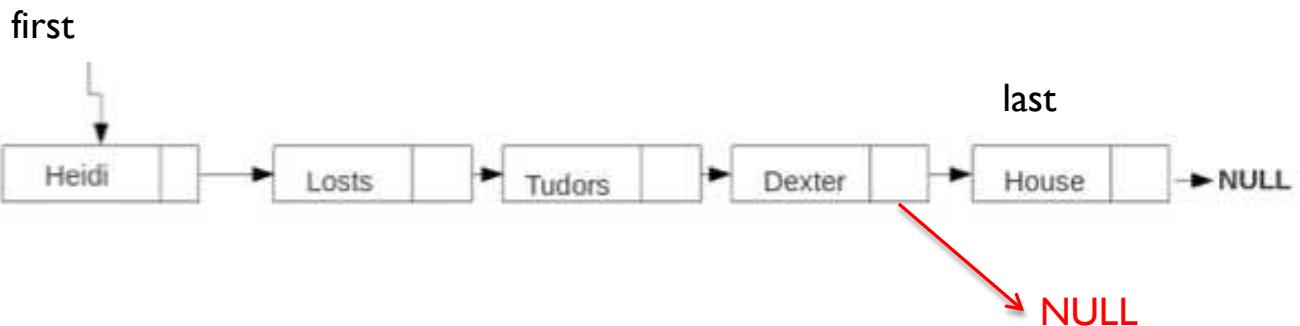
## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeLast)

```
lista.removeLast();
```



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeLast)

```
lista.removeLast();
```

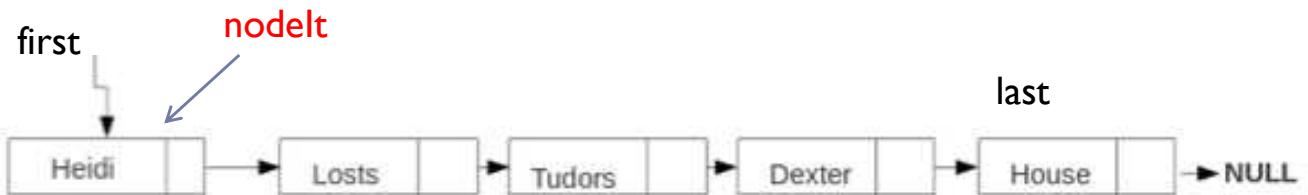


Idea: tenemos que llegar al penúltimo nodo de la lista. Cuando se alcanza este nodo, entonces tenemos que modificar su próxima referencia a NULL.



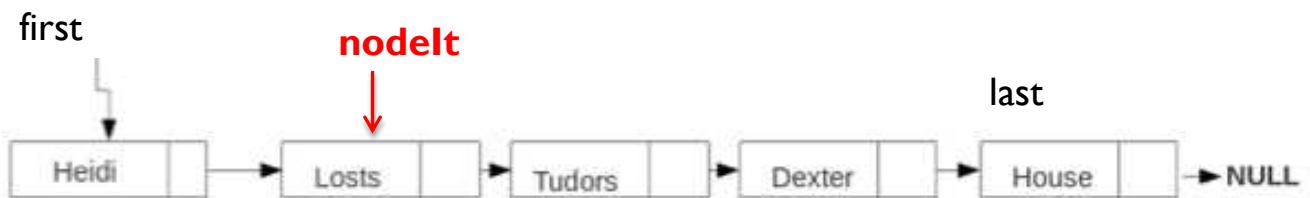
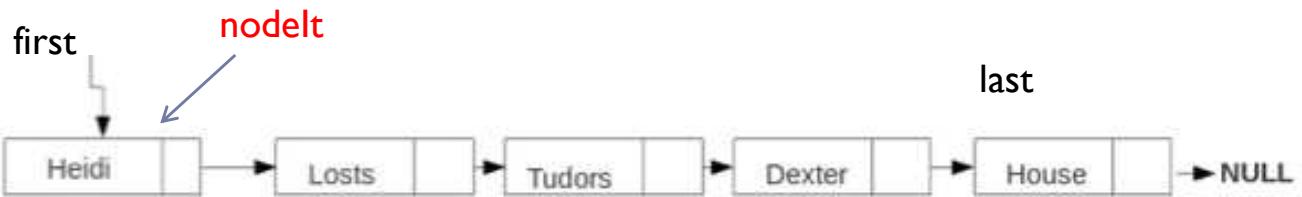
## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeLast)

`lista.removeLast()`



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeLast)

`lista.removeLast()`

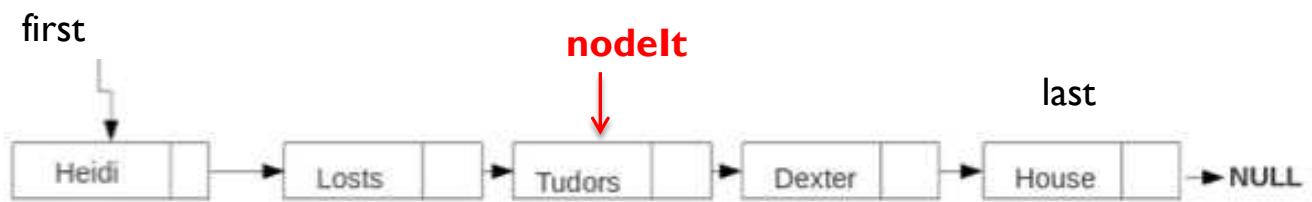


**nodeIt= nodeIt.next;**



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeLast)

`lista.removeLast()`



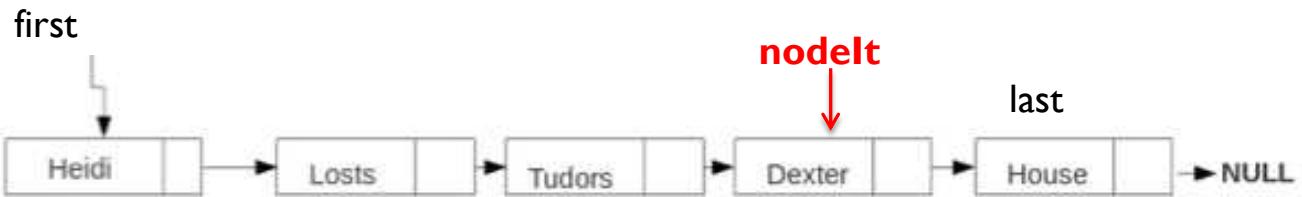
`nodelt= nodelt.next;`

¿Cuándo tenemos que parar?



## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeLast)

lista.removeLast()



```
while (nodelt!=last) {  
    nodelt= nodelt.next;  
}  
}
```

hasta que nodelt.next!=last

## Implementación de un TAD Lista usando una Lista Simplemente Enlazada (Método removeLast)

```
public void removeLast() {
    if (!isEmpty()) {
        if (size==1)
            removeFirst();
        else {
            SNode nodeIt = first;
            while (nodeIt.next!=last) {
                nodeIt = nodeIt.next;
            }
            nodeIt.next=null;
            last=nodeIt;
            size--;
        }
    }
}
```

## Ejercicio: implementa el resto de los métodos (Clase de laboratorio)

---

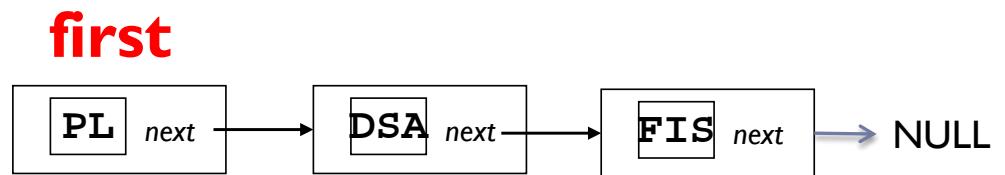
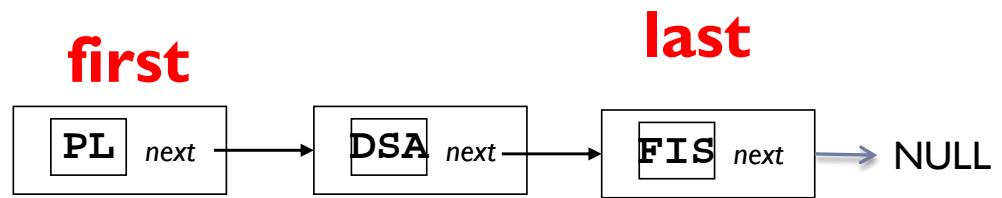
```
public String getFirst();
public String getLast();
public int getSize();
public boolean contains(String elem);
public int indexOf(String elem);

public void insertAt(int index, String newElem);
public String getAt(int index);
public void removeAll(String elem);
```



## Implementación sin last

- ▶ **Ejercicio:** Implementa una Lista Simplemente Enlazada sin usar last
- ▶ Compara ambas implementaciones. ¿Qué operaciones son mas eficientes usando last?



# **Lista Simplemente Enlazada Circular**

- ▶ Ejercicio: implementa una Lista Simplemente Enlazada circular.

**first**



## **Tema 2.4.2 TAD Lineales**

### **TAD Lista: Implementación con una Lista Dblemente Enlazada**

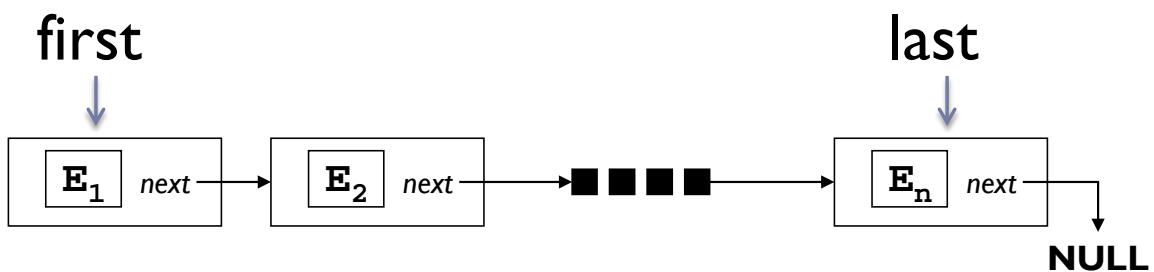
Estructura de Datos y Algoritmos (EDA)

# Outline

---

- ▶ 2.1. ¿Qué es un TAD Lineal?
- ▶ 2.2. TAD Pila
- ▶ 2.3. TAD Cola
- ▶ 2.4. TAD Lista
  - ▶ 2.4.1 Implementación con una Lista Simplemente Enlazada
  - ▶ **2.4.2 Implementación con una Lista Dblemente Enlazada**

# Implementación con un TAD Lista Simplemente Enlazada



- ▶ Un **node** es un **Objet** que almacena una referencia (**Ei**) a un objeto (el elemento de la Lista) y una referencia (**next**) al node que almacena el siguiente elemento en la Lista
- ▶ **first** y **last** permiten acceder al primer y al último elemento de la Lista respectivamente
- ▶ Para visitar un elemento, debemos comenzar a visitar el **first node** y pasar al siguiente node (**por next**) de forma iterativa, hasta llegar al node que contiene el elemento buscado

## Ventajas de Lista Simplemente Enlazada

---

- ▶ Las inserciones y eliminaciones se pueden realizar fácilmente (no es necesario mover elementos), como sucedía en los arrays
- ▶ El tamaño de la Lista no es fijo (puede ampliarse o reducirse según los requisitos)
- ▶ Uso eficiente de la memoria (el espacio no se desperdicia)

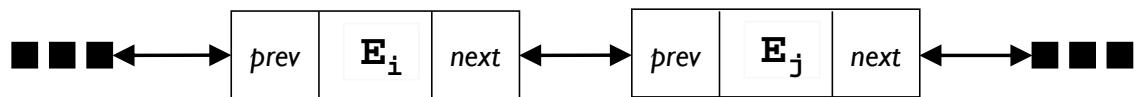
## Desventajas de Lista Simplemente Enlazada

---

- ▶ Si necesitamos llegar a un elemento en particular (por ejemplo, en el medio de la Lista), tenemos que pasar por todos los elementos que lo preceden
- ▶ En particular, si necesitamos un elemento, tenemos que llegar a su nodo predecesor
- ▶ Solo podemos atravesar la Lista desde el principio, nunca desde el final. Esto no es muy eficiente cuando buscas un elemento que está en las últimas posiciones

## ¿Cómo mejorar el acceso a los nodos? Listas Dblemente Enlazadas

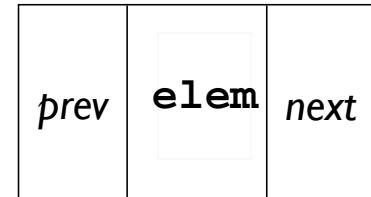
- ▶ Se permite visitar la lista de izquierda a derecha, y también a la inversa
- ▶ Además de la referencia **next**, un node necesita una referencia adicional al node anterior (**prev**).



## Clase DNode

```
public class DNode {
```

```
    Object elem;  
    DNode prev;  
    DNode next;
```



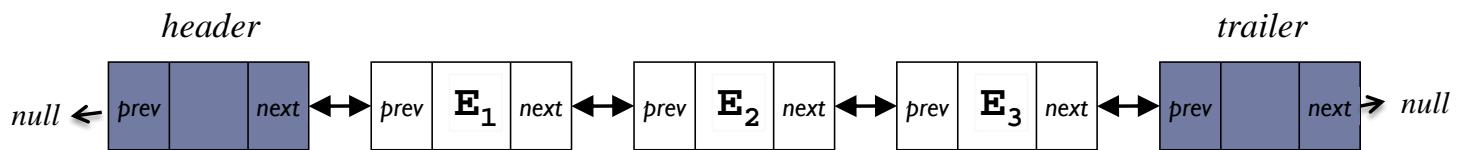
```
    public DNode(Object elem) {  
        this.elem = elem;  
    }
```

Recuerde que elem puede pertenecer a cualquier tipo de datos: String, Integer, Point, etc

## Nodos Centinelas

---

- ▶ No almacenan ningún elemento
- ▶ Situado en ambos extremos de una Lista Dblemente Enlazada.
- ▶ **header** node justo antes de la cabeza de la Lista
- ▶ **trailer** node justo después de la cola de la Lista



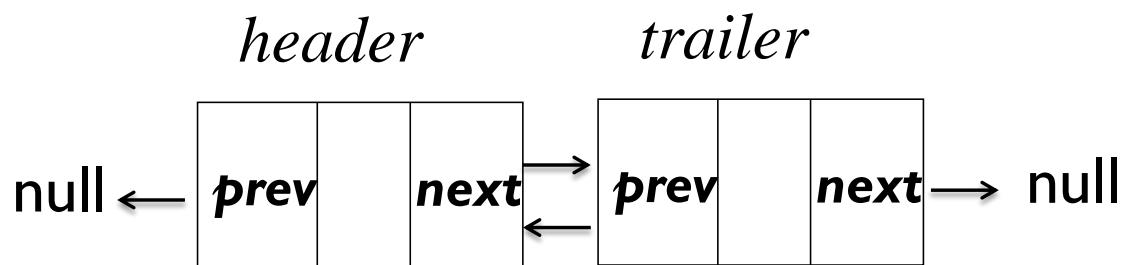
# Dlist class

---

```
/*
 * A double-linked list class with sentinel nodes
 */
public class DList implements IList {

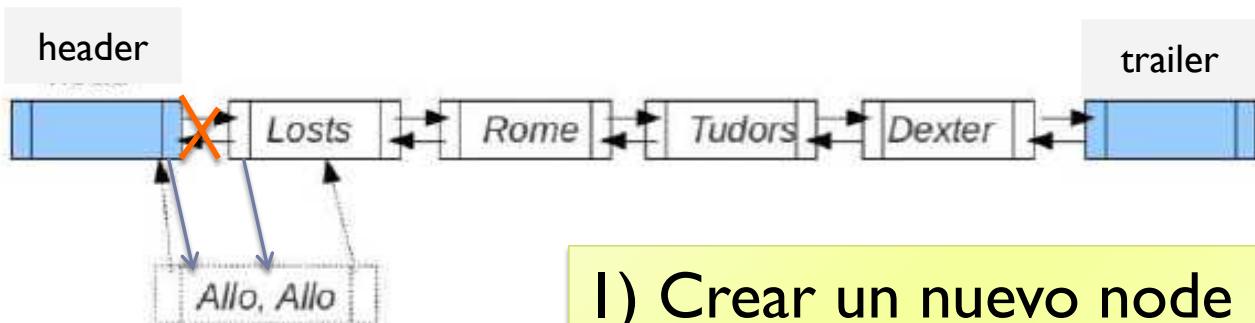
    DNode header;
    DNode trailer;
    int size=0;
```

## ¿Cómo crear una Lista vacía?



```
public DList() {  
    header = new DNode(null);  
    trailer = new DNode(null);  
    header.next = trailer;  
    trailer.prev= header;  
}
```

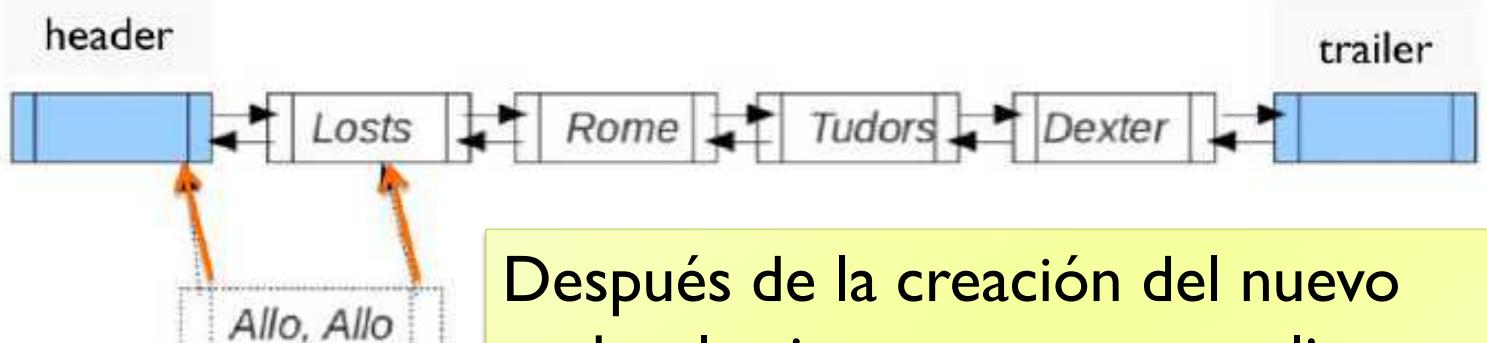
## Implementación con un TAD Lista Dblemente Enlazada (addFirst)



- 1) Crear un nuevo node
- 2) Enlazar a la lista



## Implementación con un TAD Lista Dblemente Enlazada (addFirst)



Después de la creación del nuevo node, el primer paso es actualizar sus referencias next y prev

```
DNode newNode=new DNode ("Allo,Allo");  
newNode.next=header.next;  
newNode.prev=header;
```

## Implementación con un TAD Lista Dblemente Enlazada (addFirst)

Finalmente, tenemos que actualizar las referencias de la lista para apuntar al nuevo node



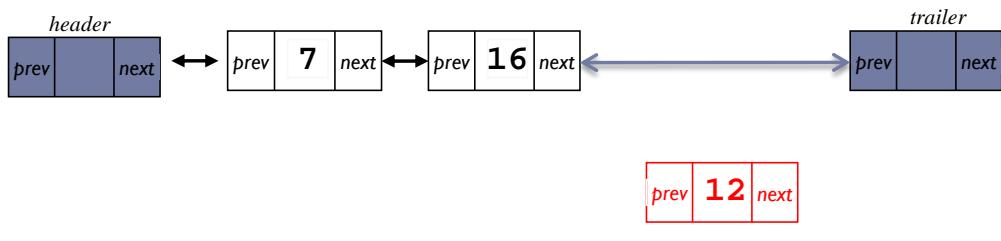
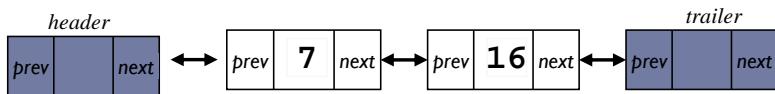
```
DNode newNode=new DNode("Allo, Allo");
newNode.next=header.next;
newNode.prev=header;
header.next.prev=newNode;
header.next=newNode;
```

## Implementación con un TAD Lista Dblemente Enlazada (addFirst)

```
public void addFirst(String elem) {  
    DNode newNode = new DNode(elem);  
    newNode.next = header.next;  
    newNode.prev= header;  
    header.next.prev= newNode;  
    header.next = newNode;  
    size++;  
}
```

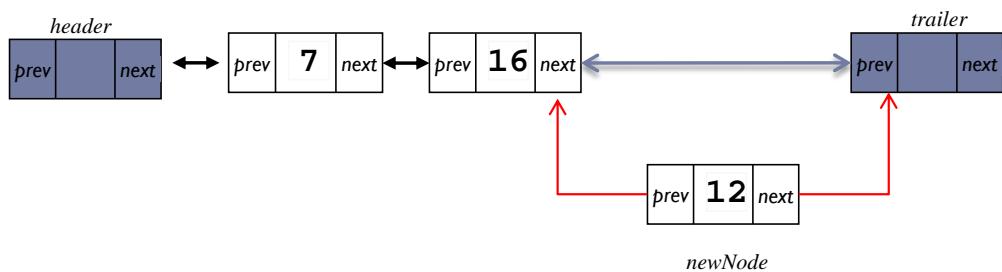
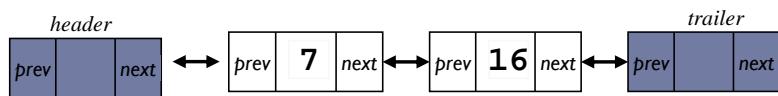
► ¿Qué sucede si cambiamos el orden de estas instrucciones?

## Implementación con un TAD Lista Dblemente Enlazada (addLast)



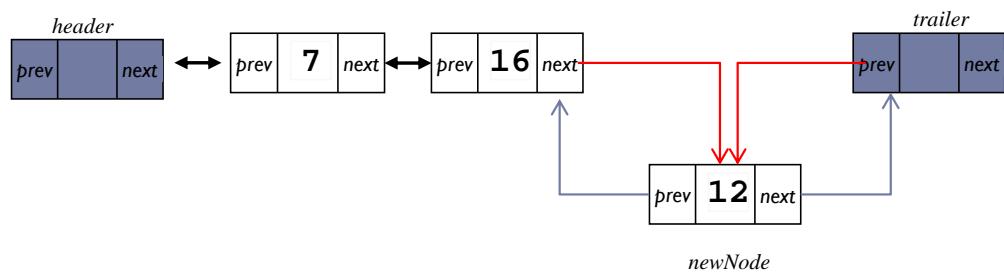
```
DNode newNode=new DNode (12);
```

## Implementación con un TAD Lista Dblemente Enlazada (addLast)



```
DNode newNode=new DNode (12);
newNode.next=tailer;
newNode.prev=tailer.prev;
```

## Implementación con un TAD Lista Dblemente Enlazada (addLast)



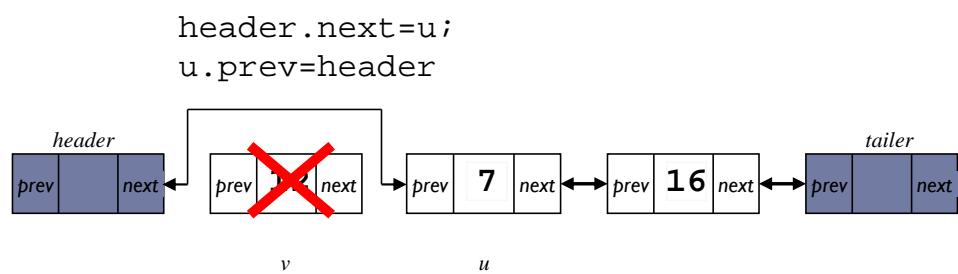
```
Dnode newNode=new Dnode(12);  
newNode.next=tailer;  
newNode.prev=tailer.prev;  
tailer.prev.next=newNode;  
tailer.prev=newNode;
```

## Implementación con un TAD Lista Dblemente Enlazada (addLast)

```
public void addLast(String elem) {  
    DNode newNode = new DNode(elem);  
    newNode.next = trailer;  
    newNode.prev= trailer.prev;  
    trailer.prev.next = newNode;  
    trailer.prev= newNode;  
    size++;  
}
```

► ¿Funciona cuando la lista está vacía?

## Implementación con un TAD Lista Dblemente Enlazada (removeFirst)



header.next=header.next.next;  
header.next.prev=header;

## Implementación con un TAD Lista Dblemente Enlazada (removeFirst)

Encuentra el error en el método:

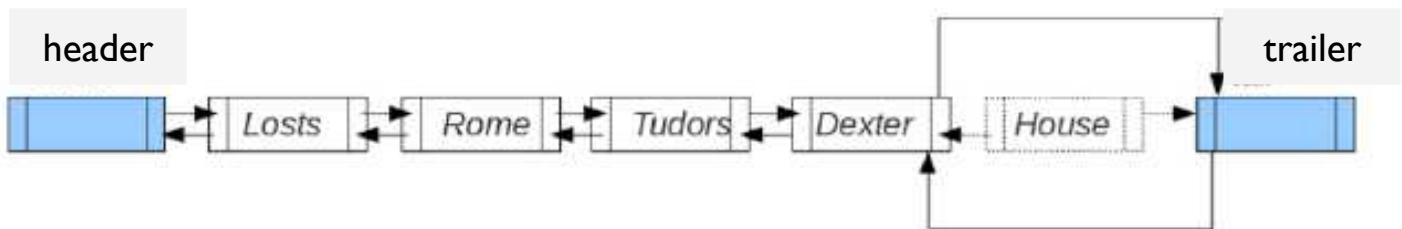
```
public void removeFirst() {  
    header.next=header.next.next;  
    header.next.prev=header;  
}
```

## Implementación con un TAD Lista Dblemente Enlazada (removeFirst)

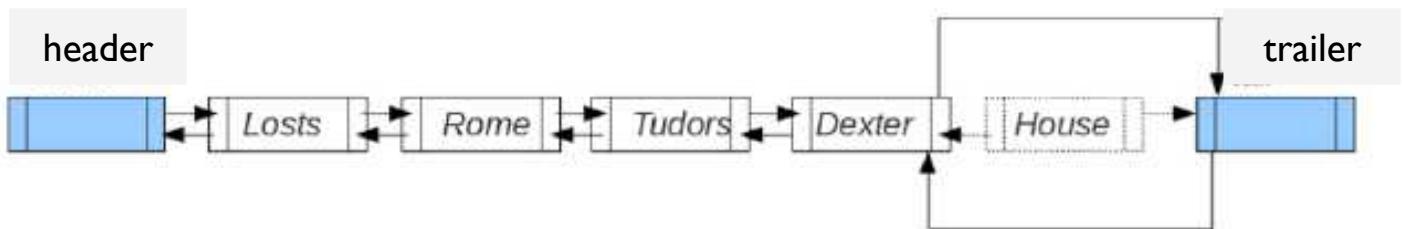
---

```
public void removeFirst() {  
    if (isEmpty()) {  
        System.out.println("DList: List is empty");  
        return;  
    }  
    header.next = header.next.next;  
    header.next.prev= header;  
    size--;  
}
```

## Implementación con un TAD Lista Dblemente Enlazada (removeLast)

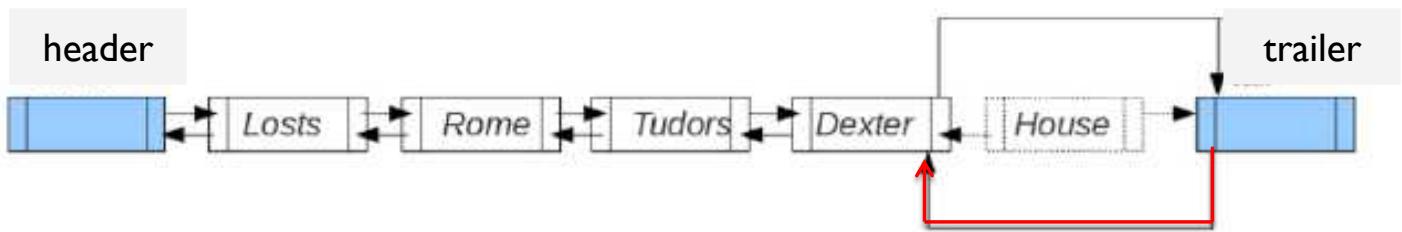


## Implementación con un TAD Lista Dblemente Enlazada (removeLast)



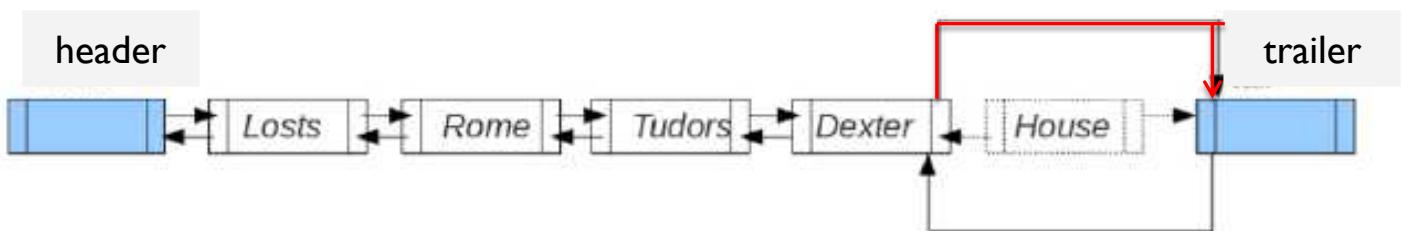
¿Tienes que recorrer la lista para encontrar el penúltimo nodo?

## Implementación con un TAD Lista Dblemente Enlazada (removeLast)



```
trailer.prev=trailer.prev.prev;
```

## Implementación con un TAD Lista Dblemente Enlazada (removeLast)



```
trailer.prev=trailer.prev.prev;  
trailer.prev.next=trailer;
```

## Implementación con un TAD Lista Dblemente Enlazada (removeLast)

Encuentra el error en el método:

```
public void removeLast() {  
    tailer.prev=tailer.prev.prev;  
    tailer.prev.next=tailer;  
}
```

## Implementación con un TAD Lista Dblemente Enlazada (removeLast)

---

```
public void removeLast() {  
    if (isEmpty()) {  
        System.out.println("DList: List is empty");  
        return;  
    }  
    trailer.prev= trailer.prev.prev;  
    trailer.prev.next = trailer;  
    size--;  
}
```

## Implementar el resto de los métodos

---

- `int getSize()`
- `void insertAt(int index, Object elem)`
- `Object getAt(int index)`
- `void removeAt()`
- `boolean isEmpty()`
- `void removeAll(Object elem)`



## **Parte V**

### **Tema 3. Algoritmos I - Análisis de algoritmos**



## Tema 3. Algoritmos I: Análisis de algoritmos

Estructura de Datos y Algoritmos (EDA)

# Objetivos

---

- ▶ Al final de la clase, los estudiantes deben ser capaces de:
- 1) Determinar empíricamente la complejidad temporal de algoritmos simples
  - 2) Determinar el orden de complejidad de los algoritmos
  - 3) Comparar y clasificar los algoritmos de acuerdo a su complejidad
  - 4) Diferenciar los conceptos mejor caso y el peor caso en el rendimiento de un algoritmo

## Contenidos

---

- ▶ Análisis de Algoritmos
  - ▶ Análisis Empírico de Algoritmos
  - ▶ Análisis Teórico de Algoritmos

# Análisis de Algoritmos

---

- ▶ Un problema puede tener varias soluciones diferentes (**algoritmos**)
- ▶ Objetivo: **elegir el algoritmo más eficiente**



# Análisis de Algoritmos

---

- ▶ Un **algoritmo** es un conjunto de pasos (instrucciones) para resolver un problema
- ▶ Debe ser correcto !!!



# Análisis de Algoritmos

---

- ▶ Un **algoritmo** es un conjunto de pasos (instrucciones) para resolver un problema
- ▶ Debe ser correcto !!!
- ▶ Debe ser eficiente !!!



# Análisis de Algoritmos

---

- ▶ Estudiar el rendimiento de los algoritmos (tiempo de ejecución y los requisitos de espacio)
  - ▶ Comparar algoritmos
  - ▶ Enfoque basado en el tiempo: ¿cómo estimar el tiempo requerido para un algoritmo?
    - ▶ Análisis empírico
    - ▶ Análisis teórico
- ▶

## Contenidos

---

- ▶ Análisis de Algoritmos
  - ▶ **Análisis Empírico de Algoritmos**
  - ▶ Análisis Teórico de Algoritmos

# Análisis Empírico de Algoritmos

---

1. Escribe el programa
2. Incluye instrucciones para medir el tiempo de ejecución
3. Ejecuta el programa con entradas de diferentes tamaños
4. Representa gráficamente los resultados

# Análisis Empírico de Algoritmos

---

- Dado un número n, desarrolle un método para sumar de 1 a n
1. Escribe el programa:

```
public static long sum(long n) {  
    long result=0;  
    for (long i=1; i<=n; i++) {  
        result = result + i;  
    }  
    return result;  
}
```

# Análisis Empírico de Algoritmos

---

2. Incluye instrucciones para medir el tiempo de ejecución

Use `System.currentTimeMillis()` or `System.nanoTime()`.

```
long startTime = System.currentTimeMillis();
```

```
//code lines whose time you want to measure
```

```
long endTime = System.currentTimeMillis();
```

```
System.out.println("Took "+(endTime - startTime) + " ms");
```

# Análisis Empírico de Algoritmos

---

2. Incluye instrucciones para medir el tiempo de ejecución

```
public static long sum(long n) {  
    long startTime = System.nanoTime();  
  
    long result=0;  
    for (long i=1; i<=n; i++) {  
        result = result + i;  
    }  
  
    long endTime = System.nanoTime();  
    long total=endTime - startTime;  
  
    System.out.println("sum("+n+") took "+total + " ns");  
    return result;  
}
```

# Análisis Empírico de Algoritmos

- Ejecuta el programa con entradas de diferentes tamaños

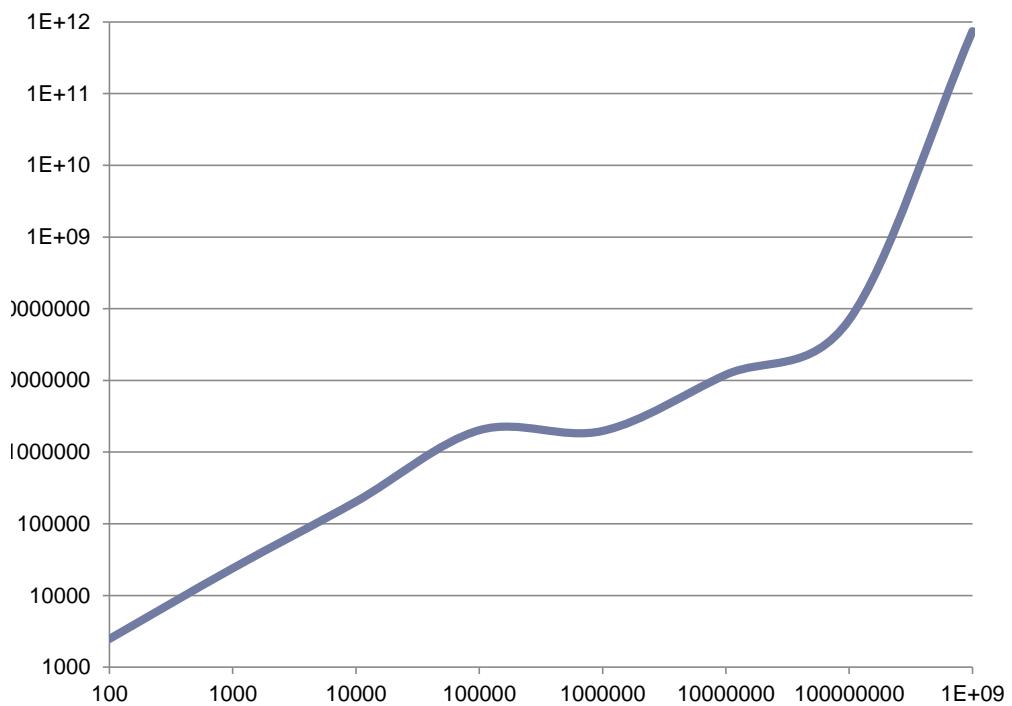
```
long MAX=1000000000;  
for (int n=1000; n<=MAX; n=n*10)  
    sum(n);
```

n	time (ns)
100	2485
1.000	23996
10.000	204102
100.000	2022441
1.000.000	1973428
10.000.000	12012791
100.000.000	69984715
1.000.000.000	743431482

# Análisis Empírico de Algoritmos

---

## 1. Representa gráficamente los resultados



# Análisis Empírico de Algoritmos

---

- Cuando se necesita mostrar rangos muy grandes (como en el ejemplo anterior), utilizar un gráfico Log-log
- El **gráfico Log-log** usa escalas logarítmicas en los ejes horizontal y vertical
- ¿Cómo se puede hacer un gráfico Log-log en Excel?
  1. En el gráfico XY (dispersión), hacer doble clic en la escala de cada eje
  2. En el cuadro Formato de ejes, seleccionar la pestaña Escala y luego verifique la escala logarítmica

# Análisis Empírico de Algoritmos

---

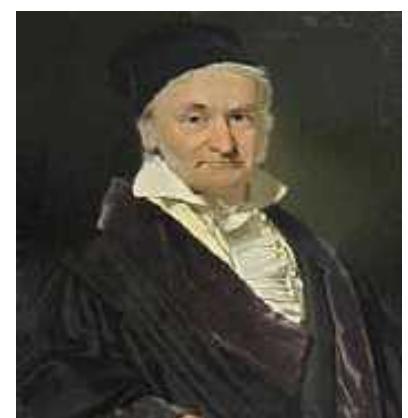
- ¿Hay otros algoritmos que resuelven este problema?

```
public static long sum(long n) {  
    long result=0;  
    for (long i=1; i<=n; i++) {  
        result = result + i;  
    }  
    return result;  
}
```



# Análisis Empírico de Algoritmos

- ▶ La solución de Gauss para sumar números del 1 al n

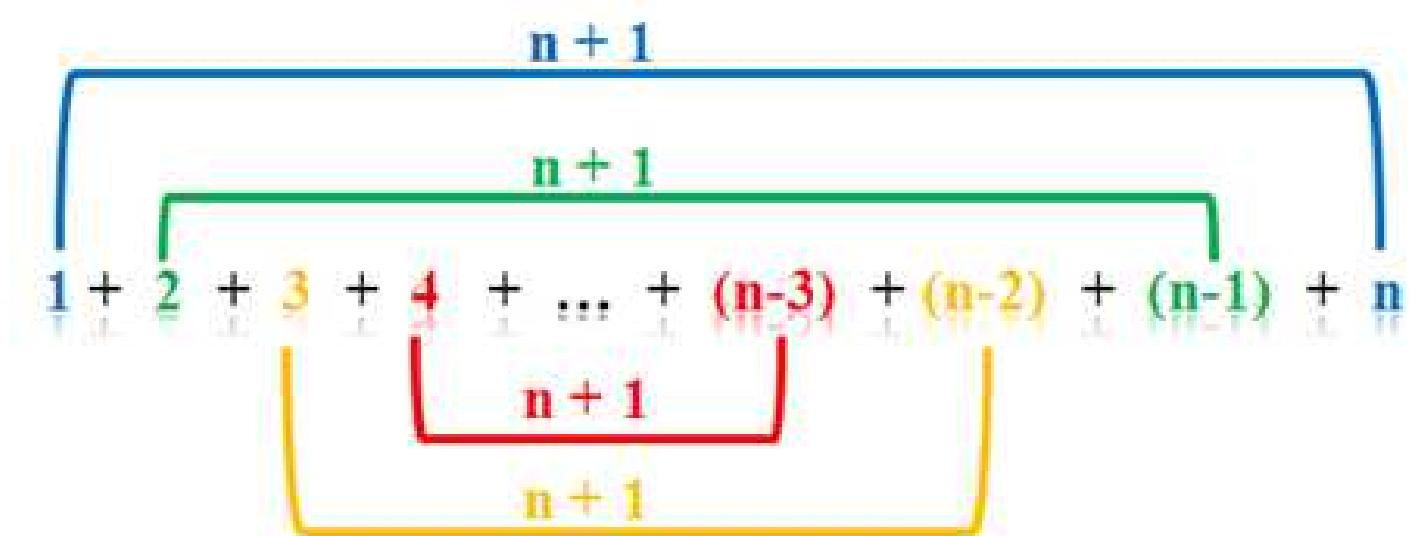


$$\sum_{k=1}^n k = \frac{n(n + 1)}{2}$$

Nota: Se puede encontrar una explicación fácil en:

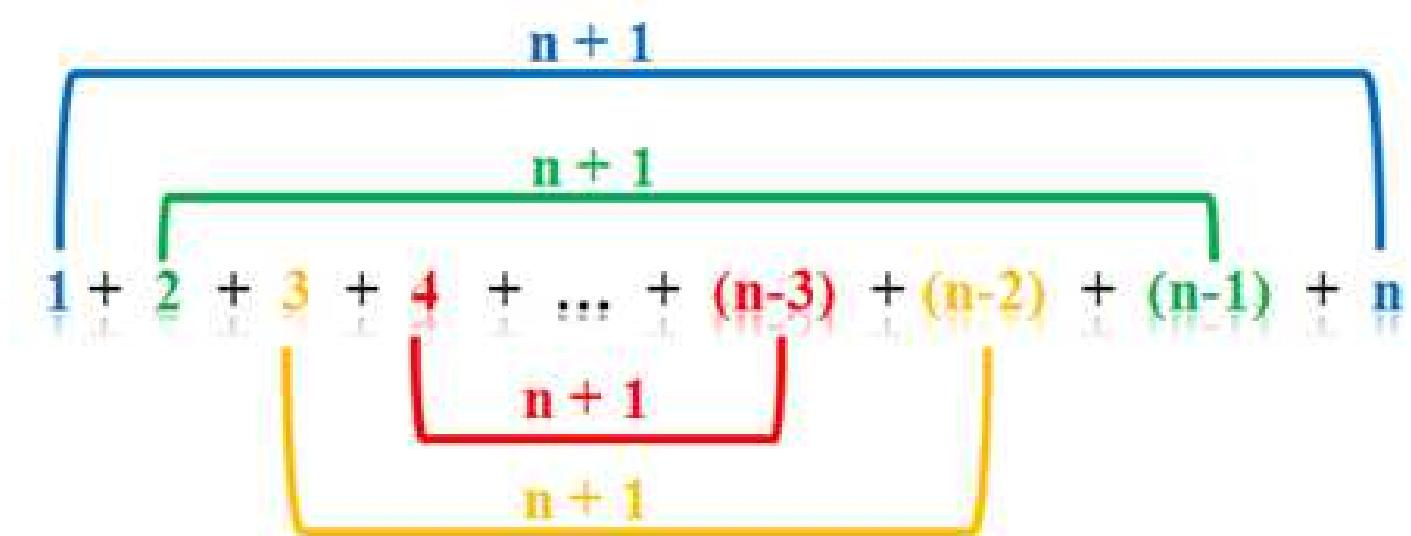
<http://mathandmultimedia.com/2010/09/15/sum-first-n-positive-integers/>

# Análisis Empírico de Algoritmos



- Cada par suma  $n + 1$
- Hay  $n / 2$  pares

# Análisis Empírico de Algoritmos



- Cada par suma  $n + 1$
  - Hay  $n / 2$  pares
- $\sum_{k=1}^n k = \frac{n(n + 1)}{2}$

## Análisis Empírico de Algoritmos

---

- ▶ Ahora, puedes implementar la solución de Gauss
- ▶ Ejecuta el programa para diferentes valores de  $n$  y mide el tiempo de ejecución
- ▶ A continuación, represente gráficamente el resultado y compáralo con la solución anterior



# Análisis Empírico de Algoritmos

---

```
public static long sumGauss(long n) {  
    long startTime = System.currentTimeMillis();  
  
    long result=n*(n+1)/2;  
  
    long endTime = System.currentTimeMillis();  
    long total=endTime - startTime;  
  
    System.out.println("sum("+n+") took "+ total + " ms");  
  
    return result;  
}
```



# Análisis Empírico de Algoritmos

---

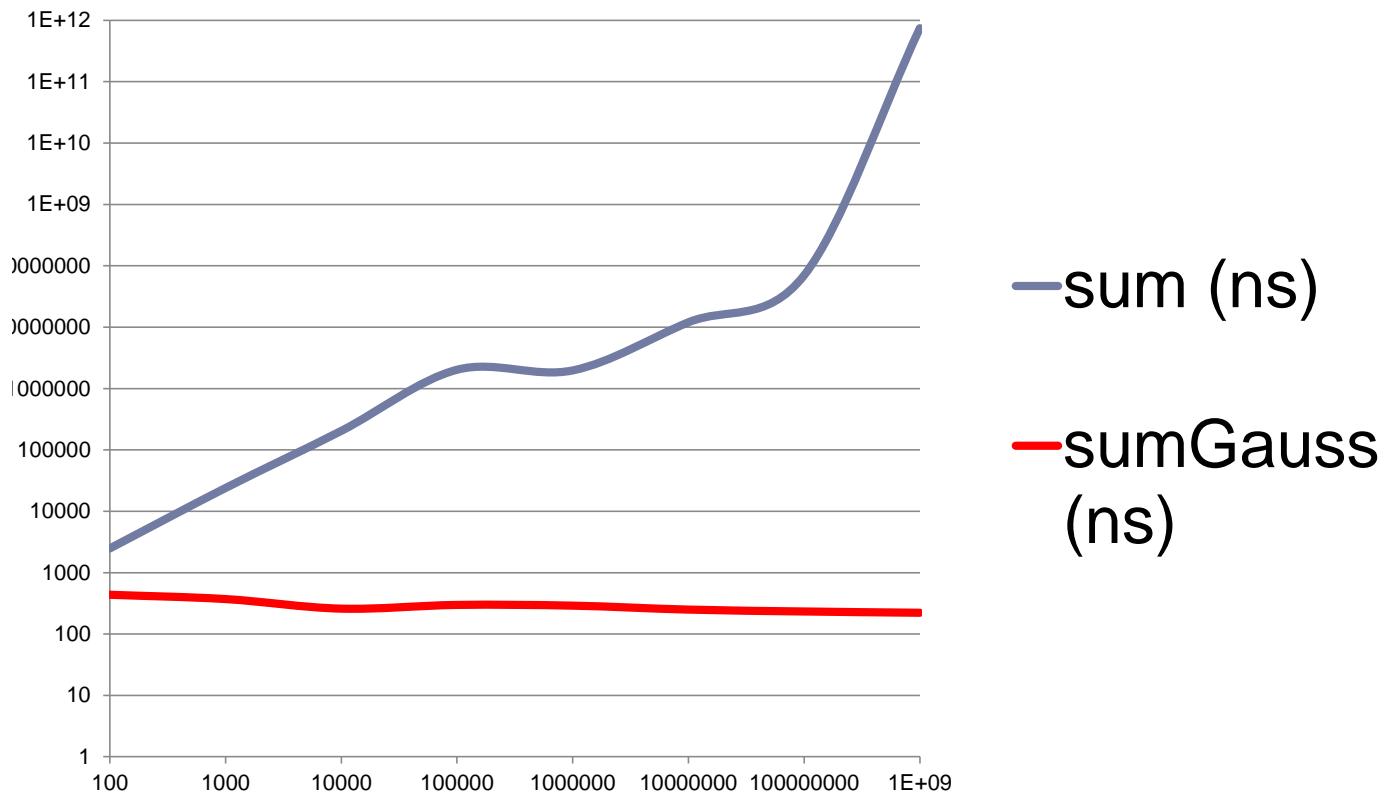
```
long MAX=1000000000;
for (int n=100; n<=MAX; n=n*10)
    sumGauss(n);
```

n	time (ns)
100	436
1.000	371
10.000	259
100.000	298
1.000.000	290
10.000.000	250
100.000.000	233
1.000.000.00	0
	222



# Análisis Empírico de Algoritmos

---



## Análisis Empírico de Algoritmos

---

- ▶ Sin embargo, algunas desventajas:
  1. Necesitas implementar el algoritmo
  2. Los resultados pueden no ser indicativos para otras entradas
  3. Mismo entorno para comparar dos algoritmos



## Outline

---

- ▶ Análisis de Algoritmos
  - ▶ Análisis Empírico de Algoritmos
  - ▶ Análisis Teórico de Algoritmos
    - ▶ Función de tiempo de ejecución

# Análisis Teórico de Algoritmos

---

- ▶ Toma en cuenta todas las entradas posibles
- ▶ **Pseudocódigo**
  - ▶ Define  $T(n)$ , función del tiempo de ejecución
  - ▶ Buscamos la independencia con el entorno de hardware / software

# Análisis Teórico de Algoritmos

---

- ▶ Ejecución de la función del tiempo  $T(n)$
- ▶ Representar el tiempo de ejecución de un algoritmo en función del tamaño de entrada
- ▶  **$T(n)=$  número de operaciones ejecutadas por un algoritmo para procesar una entrada según tamaño**

# Análisis Teórico de Algoritmos

---

- ▶ Las operaciones primitivas toman una cantidad constante de tiempo
- ▶ Ejemplos:
  - ▶ Declaring a variable: *int x;*
  - ▶ Evaluating an expression: *x+3*
  - ▶ Assigning a value to a variable: *x=2*
  - ▶ Indexing into an array: *vector[3]*
  - ▶ Calling a method: *sumGauss(n)*
  - ▶ Returning from a method: *return x;*

# Análisis Teórico de Algoritmos

- ▶ **Reglas generales para la estimación:**
  - ▶ **Declaraciones consecutivas:** solo agregar los tiempos de ejecución de las declaraciones consecutivas
  - ▶ **Bucles:** el tiempo de ejecución de un bucle es como máximo el tiempo de ejecución de las instrucciones dentro de ese bucle multiplicado por el número de iteraciones
  - ▶ **Bucles anidados:** el tiempo de ejecución de un bucle anidado que contiene una instrucción en el bucle más interno es el tiempo de ejecución de la instrucción multiplicado por el producto del tamaño de todos los bucles
  - ▶ **If/Else:** Su tiempo de ejecución es, como máximo, el tiempo de las condiciones y el mayor de los tiempos de ejecución de los bloques de instrucciones asociados

# Análisis Teórico de Algoritmos

---

```
public static long sum(long n) { # operations
    long result=0;                2
    for (long i=1; i<=n; i++) {   2+n+n
        result = result + i;      n
    }                                1
    return result;
}
```

$$T_{\text{Sum}}(n) = 3n + 5$$

Este algoritmo requiere  $3n + 6$  ns para una  
entrada de tamaño n

# Análisis Teórico de Algoritmos

---

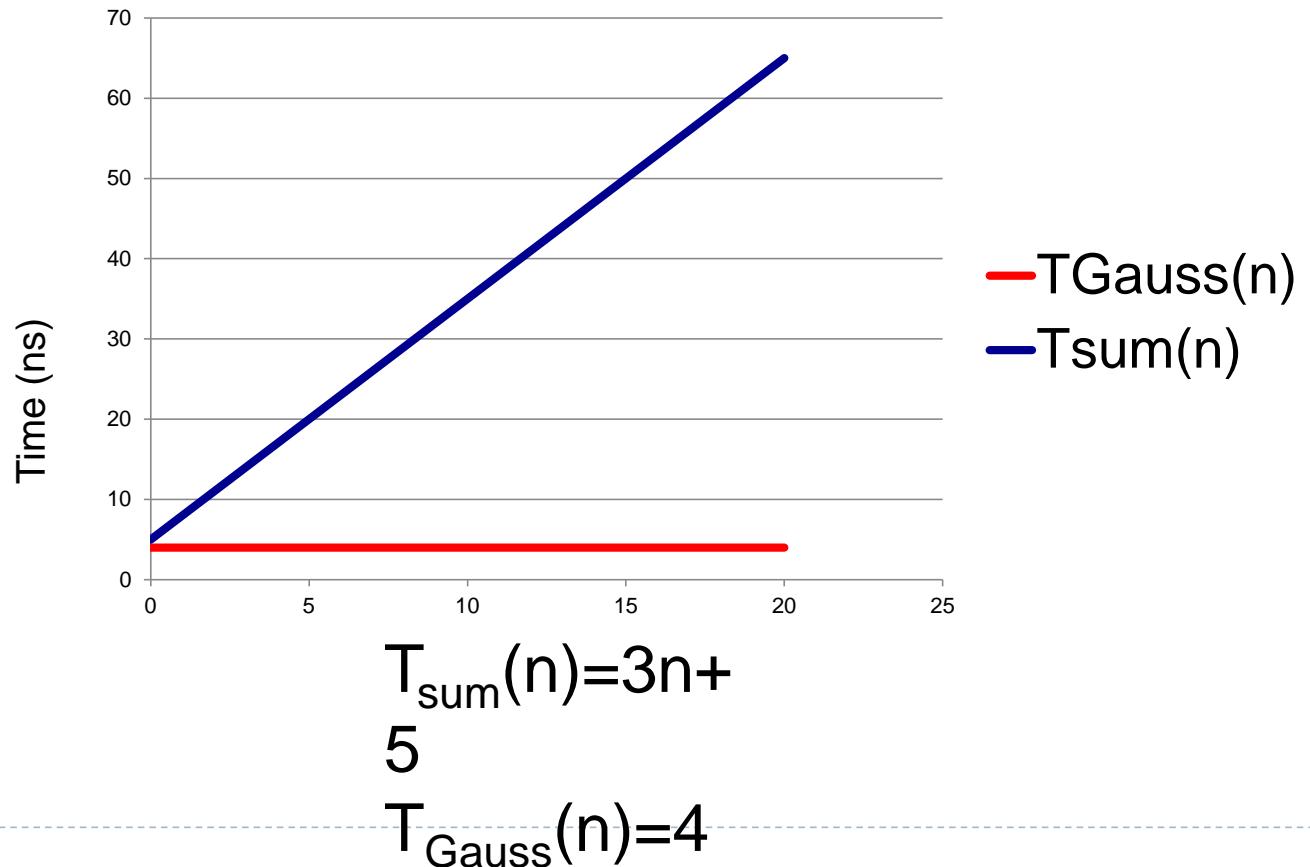
```
public static long sumGauss(long n) { # operations
    long result=n*(n+1)/2;           2
    return result;                  1
}
```

$$T_{\text{Gauss}}(n) = 3$$

La solución de Gauss requiere 3 ns para cualquier entrada

# Análisis Teórico de Algoritmos

***Los requisitos de tiempo en función del tamaño del problema n***



# Análisis Teórico de Algoritmos

---

- ▶ ¿De qué depende  $T(n)$ ?

```
// returns the index of the element x in v
public static int search(int v[], int x) {
    int index=-1;
    int n=v.length;
    for (int i=0; i<n && index== -1;i++) {
        if (x==v[i]) index=i;
    }
    return index;
}
```

# Análisis Teórico de Algoritmos

---

## ► ¿De qué depende $T(n)$ ?

```
// returns the index of the element x in v
public static int search(int v[], int x) {
    int index=-1;
    int n=v.length;
    for (int i=0; i<n && index== -1;i++) {
        if (x==v[i]) index=i;
    }
    return index; ➤ Tamaño de v
} ➤ Pero también del valor
                                de x
```

# Análisis Teórico de Algoritmos

---

## ► ¿De qué depende $T(n)$ ?

```
// returns the index of the element x in v
public static int search(int v[], int x) {
    int index=-1;
    int n=v.length;
    for (int i=0; i<n && index== -1;i++) {
        if (x==v[i]) index=i;
    }
    return index;
}
```

- Mejor-caso:  $x$  es igual a  $v[0]$
- Pero-caso:  $x$  no está en  $v$  o es igual a  $v[n-1]$ ,

# Análisis Teórico de Algoritmos

---

- ▶ Cuando el tiempo de ejecución depende de una entrada particular, definimos  **$T(n)$**  como el peor-caso de tiempo de ejecución

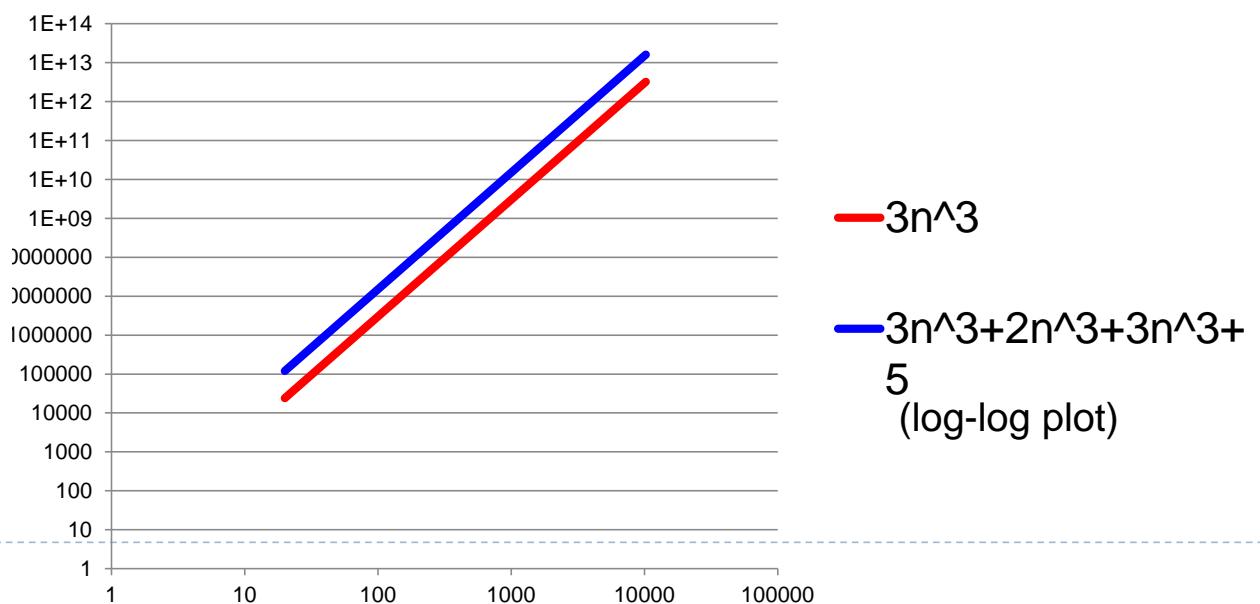
# Análisis Teórico de Algoritmos

---

- ▶  $T(n)$  también depende de:
  - 1) El ordenador en el que se ejecuta el programa
  - 2) El compilador utilizado para generar el programa
- Encontrar una función de aproximación para  $T(n)$ , una cota superior

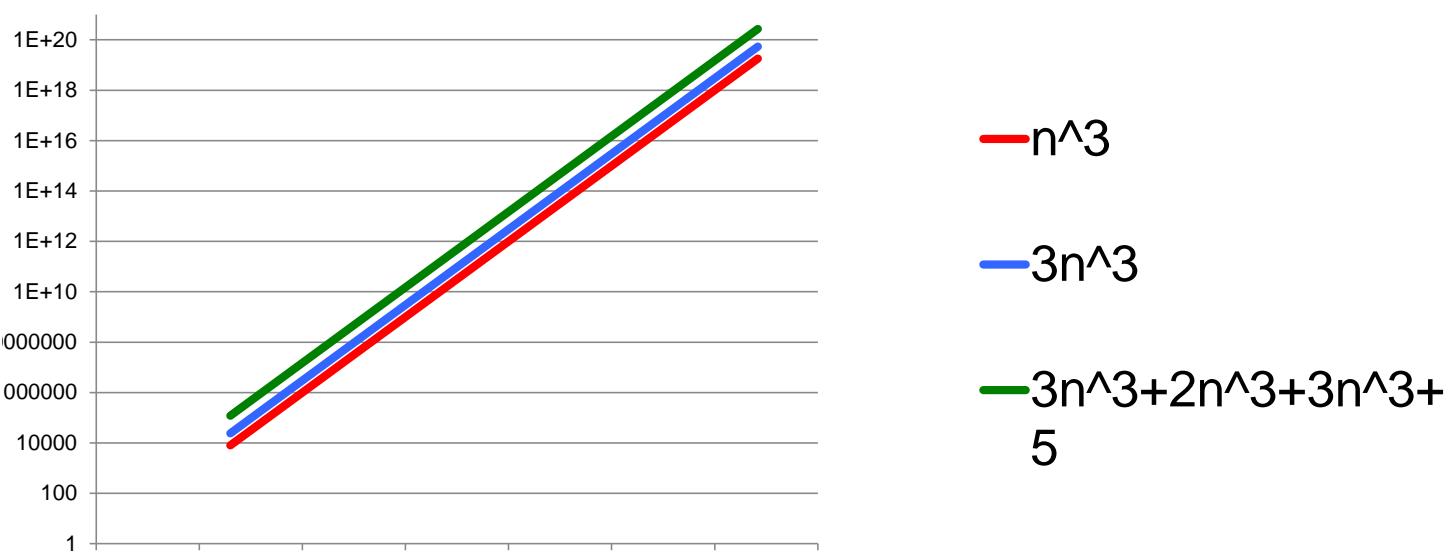
# Análisis Teórico de Algoritmos

- ▶ Ignorar términos de orden inferior :
  - ▶ Si  $n$  es pequeño, no nos importa
  - ▶ Si  $n$  es grande, los términos inferiores son insignificantes



# Análisis Teórico de Algoritmos

- ▶ Ignorar términos de orden inferior
- ▶ Establecer el coeficiente del término a1



# Análisis Teórico de Algoritmos

---

## ► Algunos ejemplos:

<b>T(n)</b>	<b>Order-of-Growth (Big-O)</b>
$n + 2$	$\sim n$
$\frac{1}{2}(n+1)(n-1)$	$\sim n^2$
$3n+\log(n)$	$\sim n$
$n(n-1)$	$\sim n^2$
$7n^4+5n^2+1$	$n^4$

# Análisis Teórico de Algoritmos

---

- ▶ Buenas noticias: un pequeño conjunto de funciones:

$$1 < \log n < n < n \log n < n^2 < n^3 < \dots < 2^n$$



# Análisis Teórico de Algoritmos

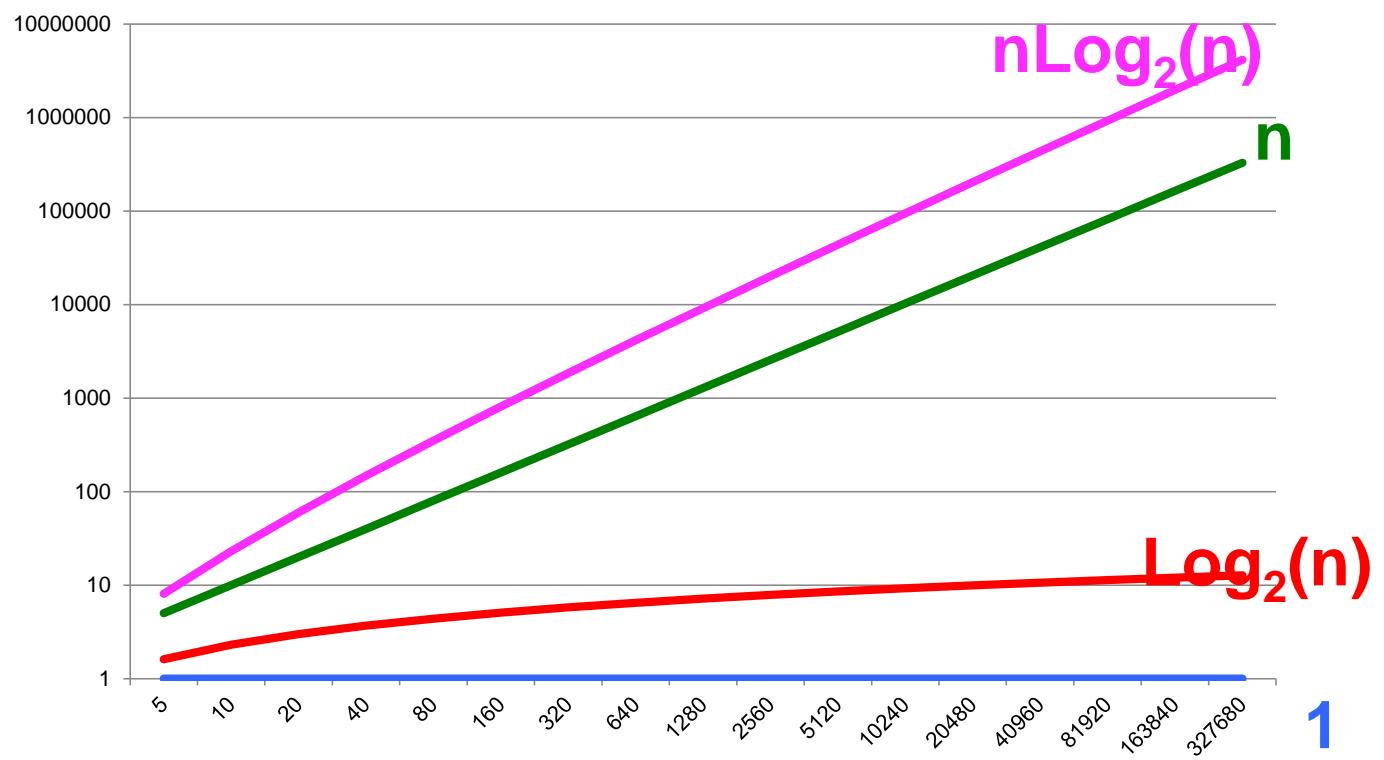
## ➤ Ordenes de complejidad eficientes:

Order	Nombre	Descripción	Ejemplo
1	Constant	Independiente del tamaño de entrada	Eliminar el primer elemento de una cola
$\log_2(n)$	Logaritmic	Dividir a la mitad	Búsqueda binaria
n	Linear	Bucle	Suma de elementos de array
$n \log_2(n)$	Linearithmic	Divide y vencerás	Mergesort, quicksort



# Análisis Teórico de Algoritmos

- **Ordenes de complejidad eficientes:**



# Análisis Teórico de Algoritmos

- Ordenes de complejidad manejables:

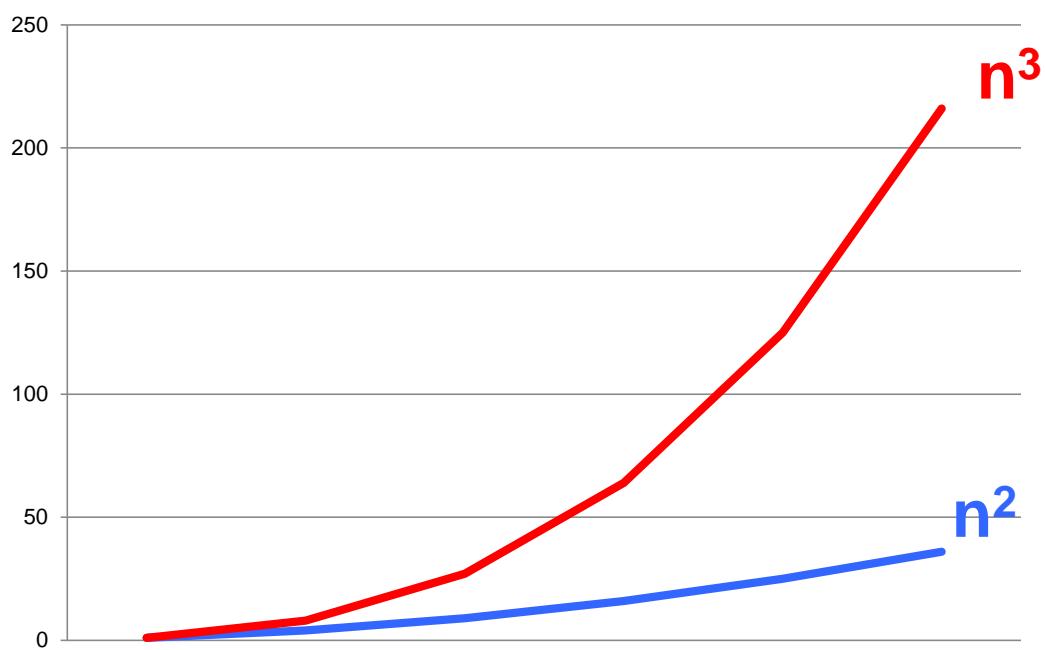
Order	Nombre	Descripción	Ejemplo
$n^2$	Cuadrático	Doble bucle	Agrega dos matrices; ordenamiento de burbuja
$n^3$	Cúbico	Triple bucle	Multiplicar dos matrices



# Análisis Teórico de Algoritmos

---

- Ordenes de complejidad manejables:



# Análisis Teórico de Algoritmos

- Ordenes de complejidad no manejables:

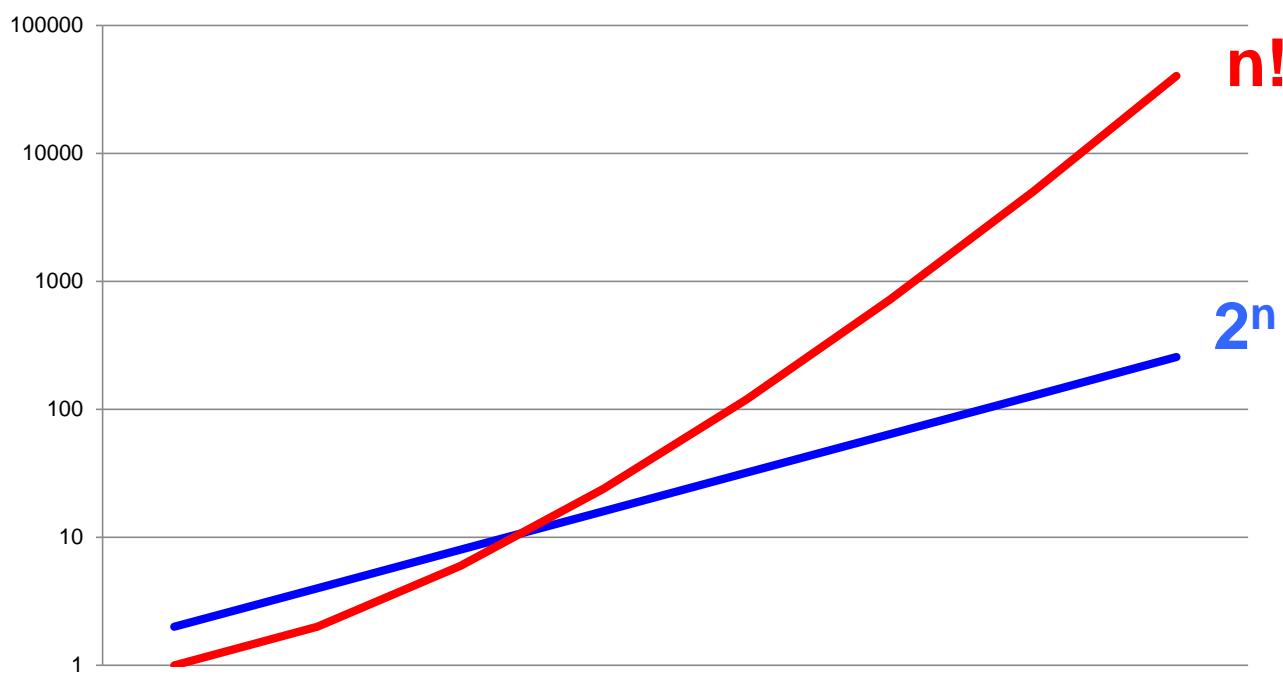
Order	Nombre	Descripción	Ejemplo
$k^n$	Exponencial	Búsqueda exhaustiva	Adivinar una contraseña
$n!$	Factorial	Búsqueda de fuerza bruta	Enumerar todas las particiones de un conjunto



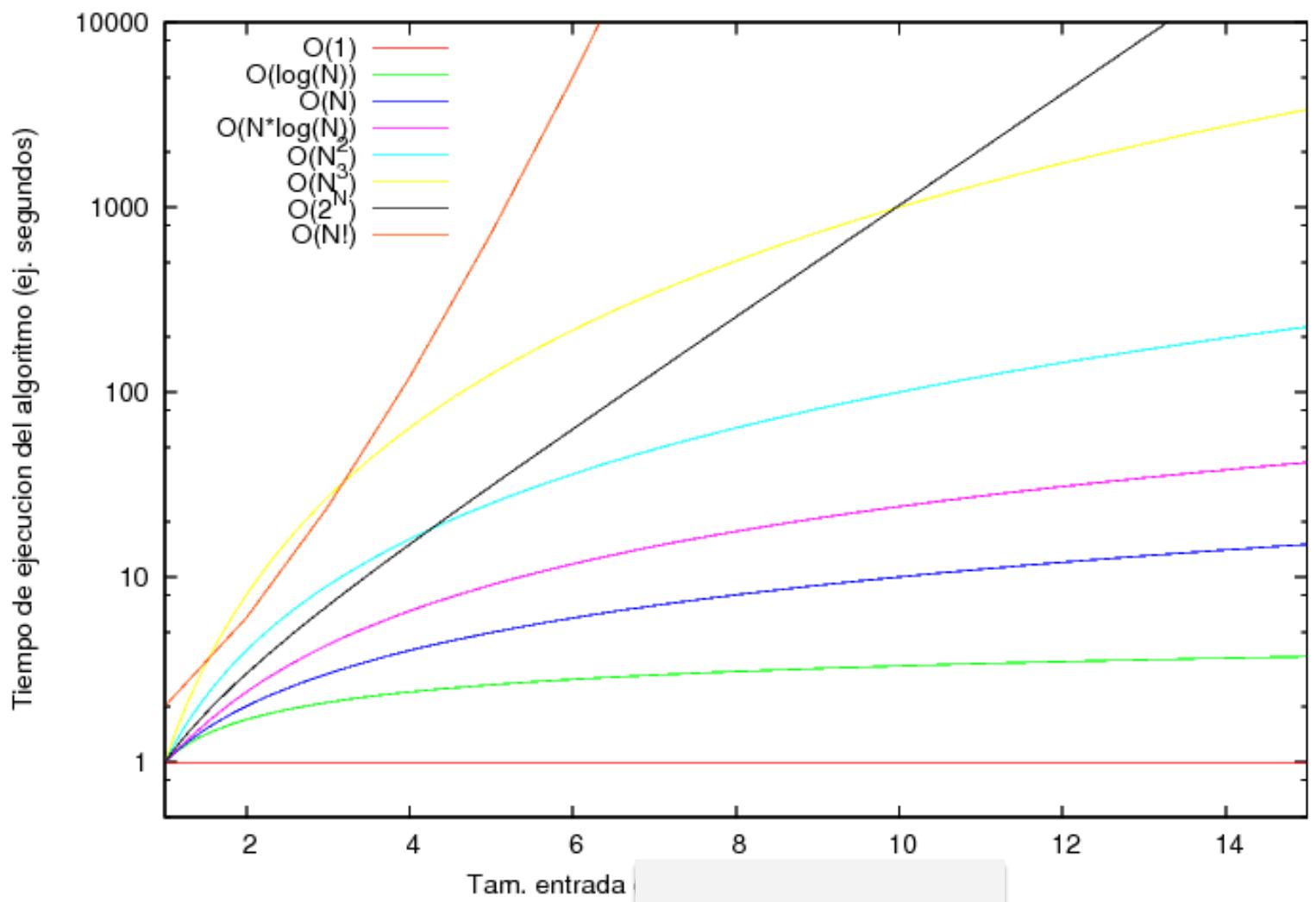
# Análisis Teórico de Algoritmos

---

- Ordenes de complejidad no manejables:



Ordenes de complejidad (escala logarítmica)



# Análisis Teórico de Algoritmos

---

## ► Más ejemplos:

- $T(n)=4 \in O(1)$ .
- $T(n)=3n+4 \in O(n)$ .
- $T(n)=5n^2 + 27n + 1005 \in O(n^2)$ .
- $T(n)=10n^3 + 2n^2 + 7n + 1 \in O(n^3)$ .
- $T(n)=n! + n^5 \in O(n!)$ .



# Ejercicios

---

## ➤ Verdadero o falso?

$$3 \in O(1)$$

$$\log n + 10 \in O(n)$$

$$5\log n \in O(\log n)$$

$$3n + 10 \in O(n)$$

$$4n + 7 \in O(n^2)$$

$$n + \log n \in O(n)$$

$$7n + n\log n \in O(n)$$

$$5n + n\log n \in O(n\log n)$$

$$4n^2 \in O(n^2)$$

$$3n^4 + 2n^3 \in O(n^4)$$

$$7n^4 + n^3 \in O(n^3)$$



## Ejercicio Calculate T(n)

---

```
//creates a matrix such as matrix[i,j]=i*j
public void createMatrix(int n) {
    int[][] matrix=new int[n][n];
    for (int i=0;i<n;i++) {
        for (int j=0;j<n;j++) {
            matrix[i][j]=i*j;
        }
    }
}
```



## Ejercicios

---

- ▶ Representar gráficamente las siguientes funciones de  $T(n)$  y discutir:
  - ▶  $T(n)=4$
  - ▶  $T(n)=3n+4.$
  - ▶  $T(n)=n^2+ 2.$
  - ▶  $T(n)=n^3+ 4n + 1.$
  - ▶  $T(n)=n!+ n^5.$
- ▶ Use Excel (tal vez tenga que usar la escala Log-log).



## Ejercicio – Búsqueda binaria

---

- ▶ Encontrar la posición de un elemento  $x$  dentro de una array ordenado  $v$ .
- ▶ Calcular su  $T(n)$  y su Big-O.



## **Parte VI**

### **Tema 4. Recursion**



## Tema 4. Recursión

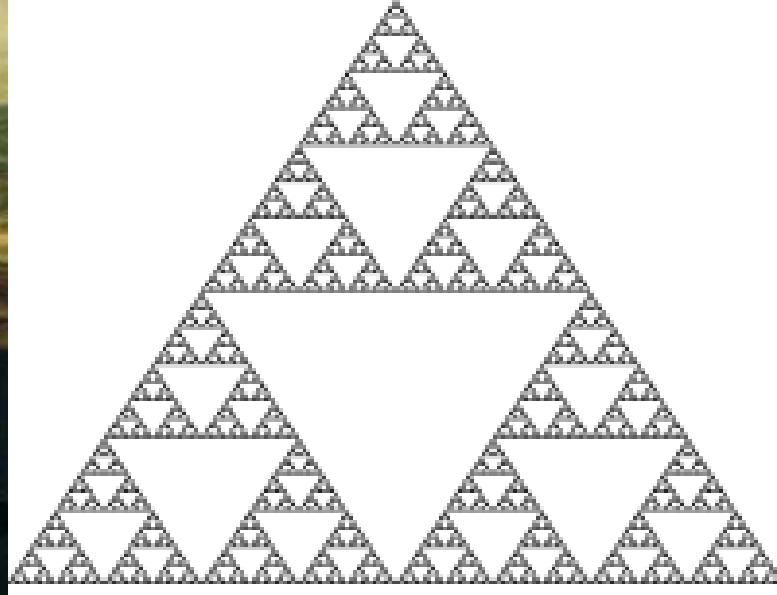
## Estructura de Datos y Algoritmos

Material elaborado por los profesores de la asignatura (Grupo LABDA)

# Objetivos

---

- ▶ Al final de la clase, los estudiantes deben ser capaces de:
  - 1) Describir el concepto de recursividad y dar ejemplos de su uso
  - 2) Identificar el caso base y el caso general de una función recursiva
  - 3) Escribir una función recursiva para resolver un problema
  - 4) Comparar las soluciones iterativas y recursivas para los problemas fundamentales



# Qué es la recursión?

---

- ▶ Un método se llama a sí mismo
- ▶ Algunas estructuras de datos pueden tener una estructura recursiva (lista o árboles)
- ▶ Cercano a la inducción matemática.

## **Los tres niveles de recursión**

---

1. Un algoritmo recursivo debe tener un caso base
2. Un algoritmo recursivo debe cambiar su estado y avanzar hacia el caso base
3. Un algoritmo recursivo debe llamarse recursivamente



# Cómo resolver problemas por recursión?

---

- ▶ Cada método recursivo tiene dos partes:
  - ▶ **CASO(S) BASE:** Caso(s) tan simple que se pueden resolver directamente
  - ▶ **CASO(S) RECURSIVO:** Caso(s) más complejo y se hace uso de la recursividad para:
    - ▶ Dividir el problema en subproblemas más pequeños y,
    - ▶ Combinar en una solución el problema más grande

# **Multiplica 2 números usando la suma**

---

$$5 \times 3 = 15 = 5 + 5 + 5$$

## Multiplica 2 números usando la suma

---

$$5 \times 3 = 15 = 5 + 5 + 5$$

```
int multiply(int x, int y) {  
    int result=0;  
    for (int i=1; i<=y;i++) {  
        result=result+x;  
    }  
    return result;  
}
```

## Multiplica 2 números usando la suma

---

$$5 \times 3 = 15 = 5 + 5 + 5$$

```
int multiplyRec(int x, int y) {
```

```
}
```

**1) Determina el caso base(s)**

## Multiplica 2 números usando la suma

---

$$5 \times 3 = 15 = 5 + 5 + 5$$

```
int multiplyRec(int x, int y) {  
    if (y==1) return x;  
}  
}
```

### 1) Determinar el caso(s) base

---

## Multiplica 2 números usando la suma

---

$$5 \times 3 = 15 = 5 + 5 + 5$$

```
int multiplyRec(int x, int y) {  
    if (y==1) return x;  
}  
}
```

### 2) Determina el caso(s) recursivo

---



## Multiplica 2 números usando la suma

---

$$5 \times 3 = 15 = 5 + 5 + 5$$

```
int multiplyRec(int x, int y) {  
    if (y==1) return x;  
    else  
        return x + multiplyRec(x,y-1);  
}
```

### 2) Determina el caso(s) recursivo

---

# Seguimiento de recursión

**multiplyRec(5,3)**



**return 5 + multiplyRec(5,2);**

```
if (y==1) return x;  
else      return x + multiplyRec(x,y-1);
```

# Seguimiento de recursión

**multiplyRec(5,3)**



```
if (y==1) return x;  
else      return x + multiplyRec(x,y-1);
```

**return 5 + multiplyRec(5,2);**



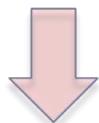
**return 5 + multiplyRec(5,1);**

# Seguimiento de recursión

**multiplyRec(5,3)**



**return 5 + multiplyRec(5,2);**



**return 5 + multiplyRec(5,1);**



**return 5;**

# Seguimiento de recursión

**multiplyRec(5,3)**

```
if (y==1) return x;  
else    return x + multiplyRec(x,y-1);
```

**return 5 + multiplyRec(5,2);**

**return 5 + multiplyRec(5,1);**



**return 5;**

Combinar desde el caso base

# Seguimiento de recursión

**multiplyRec(5,3)**

```
if (y==1) return x;  
else    return x + multiplyRec(x,y-1);
```

**return 5 + multiplyRec(5,1);**



**return 5 + 5;**



**return 5;**

Combinar desde el caso base

# Seguimiento de recursión

**multiplyRec(5,3) = 15**

**return 5 + 10;**

**Final result**



**return 5 + 5;**



**return 5;**



# Problema Factorial

---

```
int factorialRec(int n) {  
    if (n==1) return 1;  
    else return n * factorialRec(n-1);  
}
```

# Seguimiento factorial

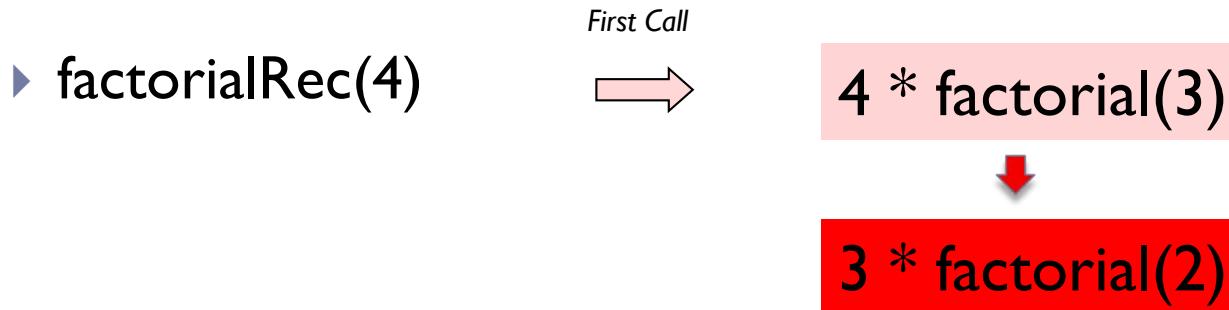
▶ factorialRec(4)  4 \* factorial(3)

*First Call*

```
int factorialRec(int n) {  
    if (n==1) return 1;  
    else return n * factorialRec(n-1);  
}
```



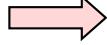
# Seguimiento factorial



```
int factorialRec(int n) {  
    if (n==1) return 1;  
    else return n * factorialRec(n-1);  
}
```



# Seguimiento factorial

▶ factorialRec(4)  4 \* factorial(3)

*First Call*



3 \* factorial(2)

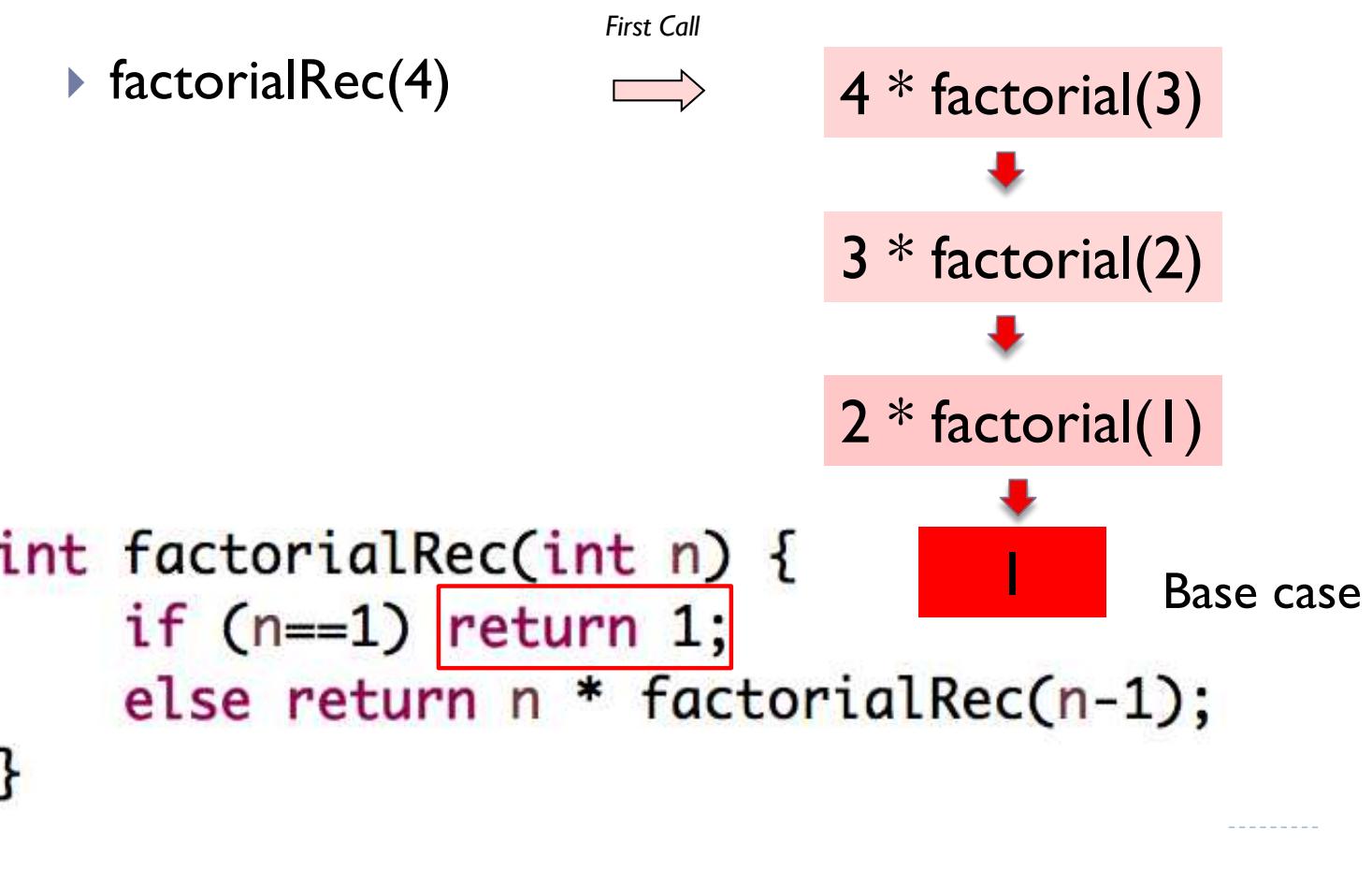


2 \* factorial(1)

```
int factorialRec(int n) {  
    if (n==1) return 1;  
    else return n * factorialRec(n-1);  
}
```



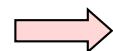
# Seguimiento factorial



# Seguimiento factorial

▶ factorialRec(4)

*First Call*



4 \* factorial(3)



3 \* factorial(2)



2 \* factorial(1)



```
int factorialRec(int n) {  
    if (n==1) return 1;  
    else return n * factorialRec(n-1);  
}
```

|

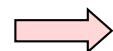


|

# Seguimiento factorial

▶ factorialRec(4)

*First Call*



4 \* factorial(3)



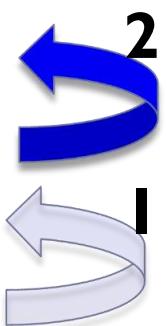
3 \* factorial(2)



2 \* 1



1



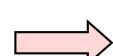
```
int factorialRec(int n) {  
    if (n==1) return 1;  
    else return n * factorialRec(n-1);  
}
```



# Tracing factorial

▶ factorialRec(4)

*First Call*



4 \* factorial(3)



3 \* 2

2

2\*1

1

```
int factorialRec(int n) {  
    if (n==1) return 1;  
    else return n * factorialRec(n-1);
```

}



# Seguimiento factorial

▶ factorialRec(4)

24  
◀

4 \* 6

6  
↶

3 \* 2

2  
↶

2 \* 1

1  
↶

1

```
int factorialRec(int n) {  
    if (n==1) return 1;  
    else return n * factorialRec(n-1);  
}
```



# Suma de una lista de números

---

```
public int sumArray(int[] data) {  
    int sum=0;  
    for (int i=0; i<data.length;i++) {  
        sum=sum + data[i];  
    }  
    return sum;  
}
```

# Suma de una lista de números

---

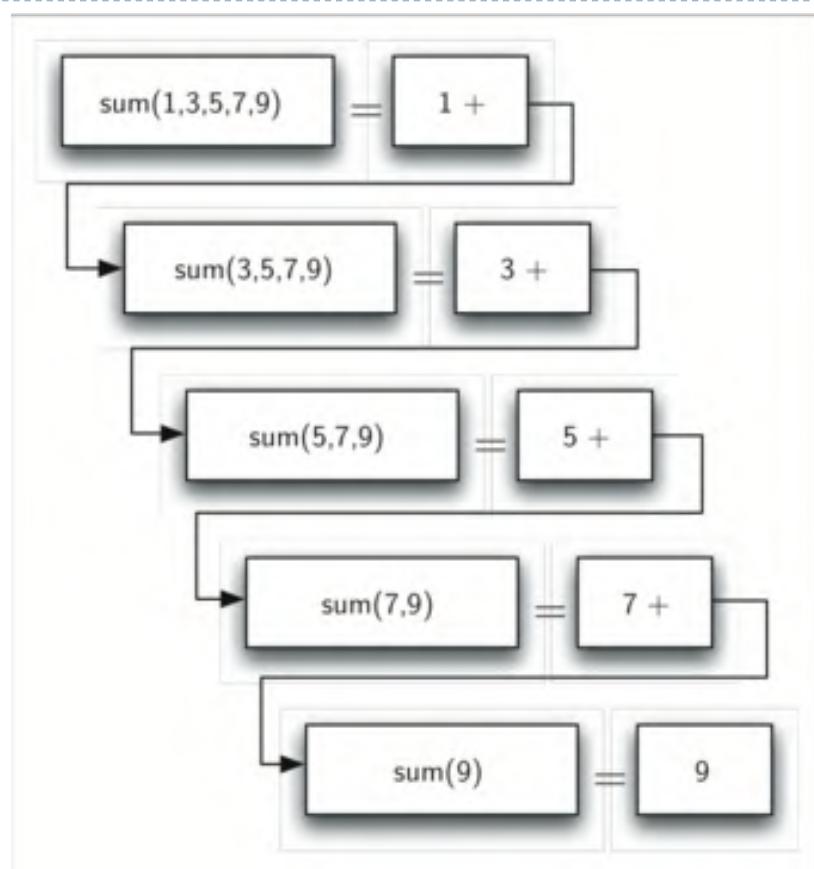
- ▶ Imaginar que no hay bucles.
- ▶ ¿Cómo se obtiene la suma de una lista de números?

## Suma una lista de números

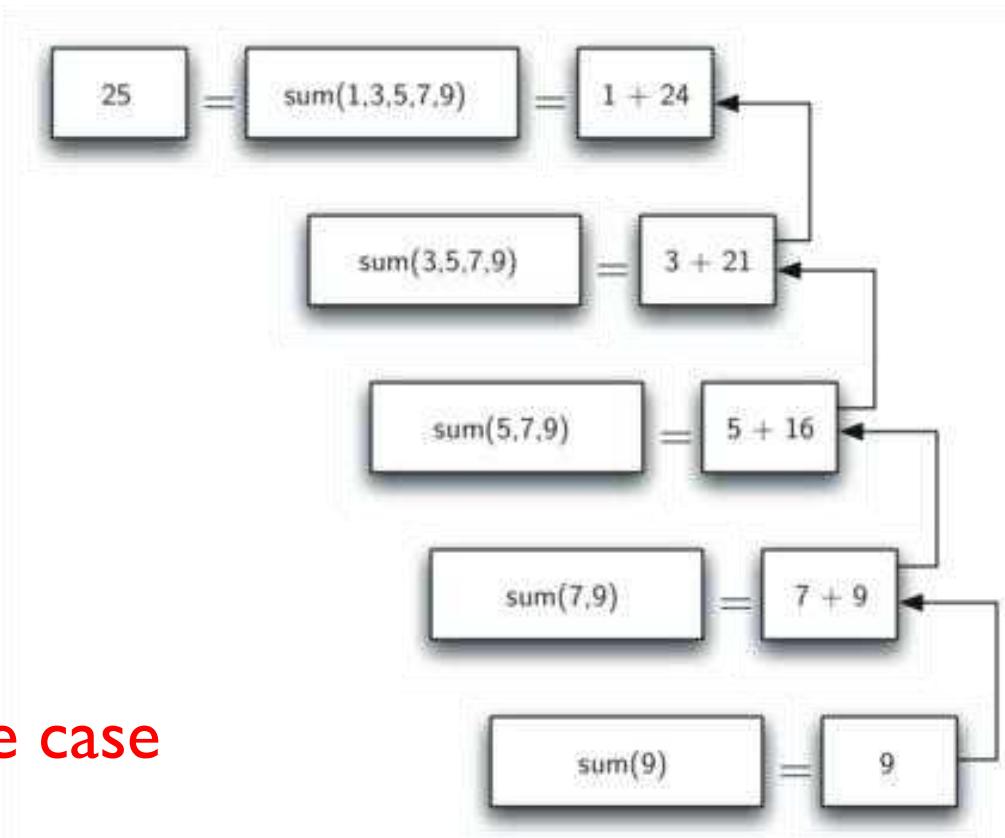
---

- ▶ Dada una lista [1,3,5,7,9]
  - ▶ La suma de sus números puede ser expresada:
- 
- ▶  $(1 + (3 + (5 + (7 + 9))))$
  - ▶  $\text{Sum} = (1 + (3 + (5 + 16)))$
  - ▶  $\text{Sum} = (1 + (3 + 21))$
  - ▶  $\text{Sum} = (1+24)$
  - ▶  $\text{Sum} = 25$

# Suma de una lista de números



# Suma una lista de números (solución recursiva)



## Suma de una lista de números (solución recursiva)

---

```
public int sumArrayRec(int[] data) {  
    return sumArrayRec(data,0);  
}  
  
public int sumArrayRec(int[] data, int i) {  
    if (i==data.length-1) return data[i];  
    else return data[i]+sumArrayRec(data,i+1);  
}
```

## Una pregunta

---

- ◆ ¿Cuántas llamadas recursivas se necesita para obtener la suma de estos números?  
2,4,6,8,10?



## Máximo común divisor (mcd)

---

- ▶ El mcd de dos enteros  $p$  y  $q$  es el mayor entero  $d$  que divide a  $p$  y  $q$ .

## Máximo común divisor (mcd)

---

$$\text{mcd}(4032, 1272)$$

$$4032 = 2^6 * 3^2 * 7$$

$$1272 = 2^3 * 3^1 * 53$$

$$\Rightarrow \text{mcd}(4032, 1272) = 2^3 * 3^1 = 24$$

## Máximo común divisor (mcd)

- ▶ Encontrar el entero mayor d que divide a p y q
- ▶ Euclid's algorithm

$$\triangleright mcd(a,b) = \begin{cases} a & \text{if } b=0 \\ mcd(b,a\%b) & \text{otherwise} \end{cases}$$

## Máximo común divisor (mcd)

---

$$\begin{aligned} \text{mcd}(4032, 1272) &= \text{mcd}(1272, 216) \\ &= \text{mcd}(216, 192) \\ &= \text{mcd}(192, 24) \\ &= \text{mcd}(24, 0) = 0 \end{aligned}$$

## Máximo común divisor (mcd)

---

```
//assume a>b, y a,b>=0
int gcd(int a, int b) {
    if (b==0) return a;
    else return gcd(b,a%b);
}
```

# Números Fibonacci

---

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377 ...

$$Fib(n) = \begin{cases} 1 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ Fib(n-1) + Fib(n-2) & \text{if } n>1 \end{cases}$$



## Números Fibonacci

---

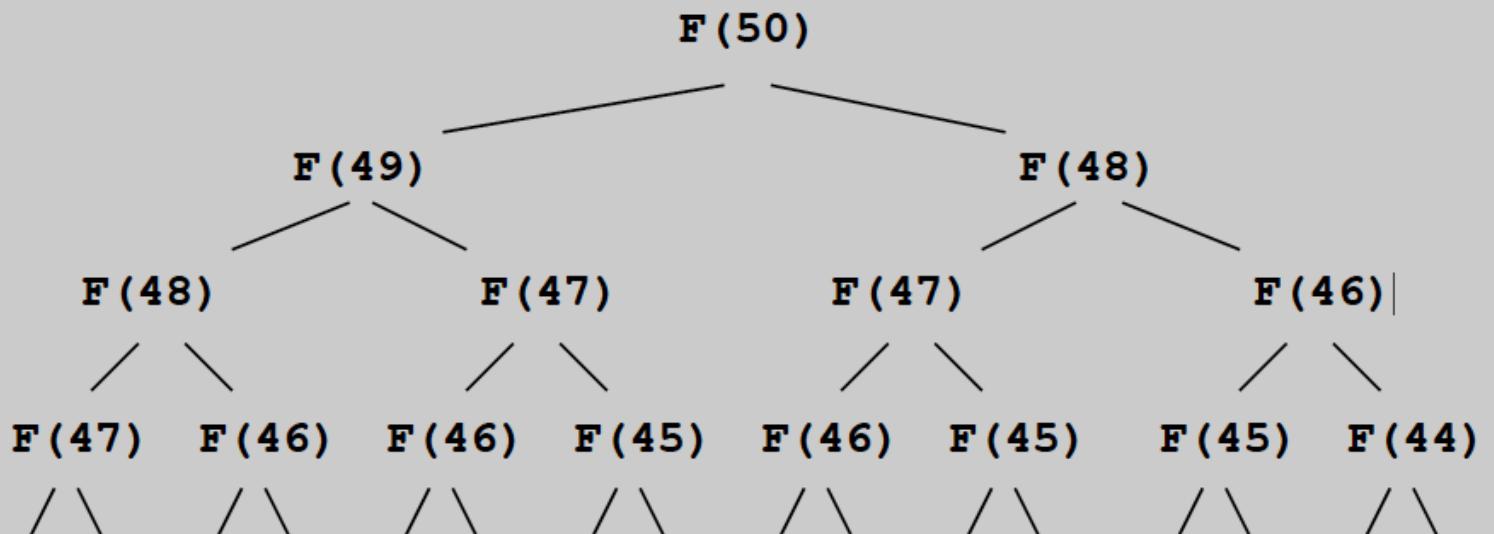
```
Int fib(int n)
{
    if ((n == 0) || (n == 1))
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

¿Es esta una forma eficiente de calcular F(50)?



# Números Fibonacci

No!! Este código es muy ineficiente



## Una forma más eficiente de calcular los números de Fibonacci

```
public static long fibo(int n) {  
    if (n == 0) return 0;  
  
    long[] F = new long[n+1];  
    F[0] = 0;  
    F[1] = 1;  
  
    for (int i = 2; i <= n; i++)  
        F[i] = F[i-1] + F[i-2];  
  
    return F[n]; Programación dinámica  
}
```

## Iteración vs Recursión

---

- ▶ Un bucle también es un proceso iterativo
- ▶ Un método recursivo es más matemáticamente elegante que usar un bucle.
- ▶ Los bucles son más eficientes que los métodos recursivos
- ▶ Todos los métodos recursivos se pueden resolver usando una solución iterativa
- ▶ No todos los problemas se pueden resolver usando recursión

# Recursión

---

- ▶ **Ventaja:** enfoque fácil y ordenado => paradigma de programación de gran alcance
- ▶ **Desventaja:** tiene peor complejidad de tiempo que los bucles (porque cada llamada de función requiere memoria múltiple para almacenar la dirección interna del método)

---

► *Iterar es humano, hacer recursividad, divino*



# **Parte VII**

## **Tema 5. Arboles**



## Tema 5. Árboles **Árboles Generales y Binarios**

Estructura de Datos y Algoritmos (EDA)

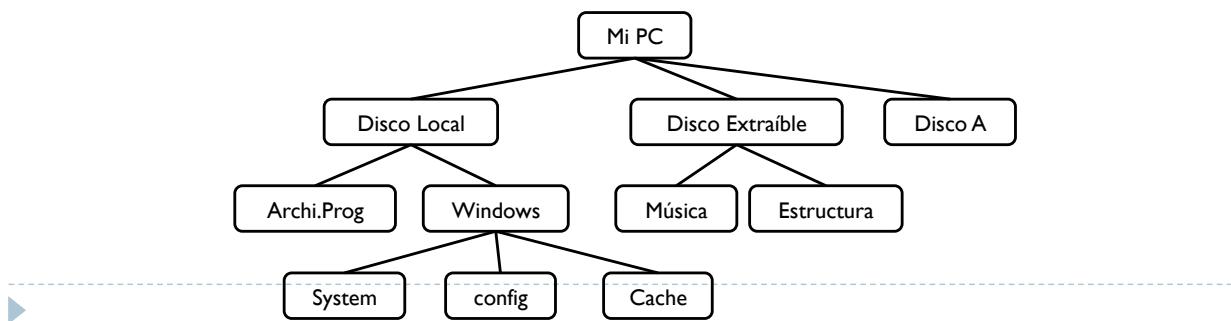
# Índice

---

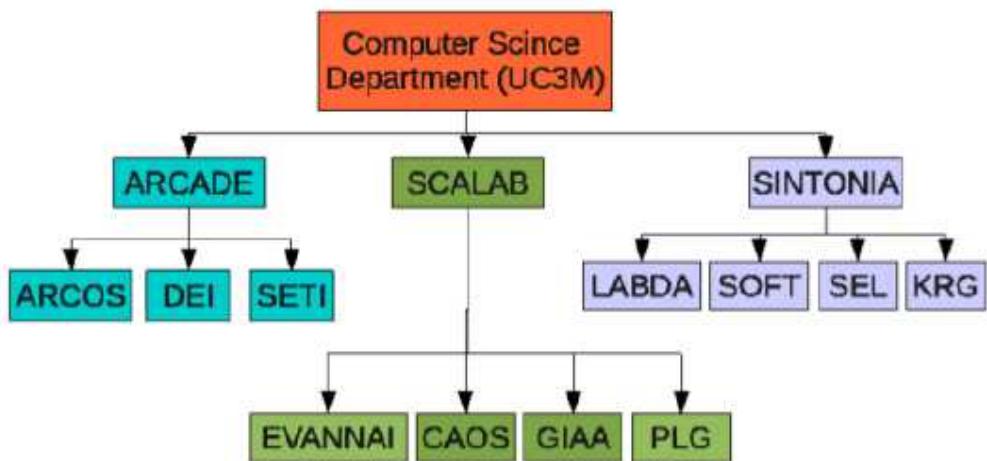
- ▶ **Conceptos básicos**
- ▶ TAD Árboles generales
- ▶ TAD Árboles binarios
- ▶ TAD Árboles binarios de búsqueda
- ▶ Equilibrado de árboles.

# ¿Qué es un árbol?

- ▶ Un árbol es un TAD que almacena elementos que tienen una relación jerárquica entre ellos (estructura jerárquica no lineal). El acceso a los elemento suele ser más rápido que en una estructura lineal.
- ▶ Relaciones padre-hijo entre nodos
- ▶ Ejemplos: **sistema de ficheros**, estructura de un libro, diagrama modular, bases de datos, interfaces gráficos, web sites, árbol genealógicos, organigramas, etc.



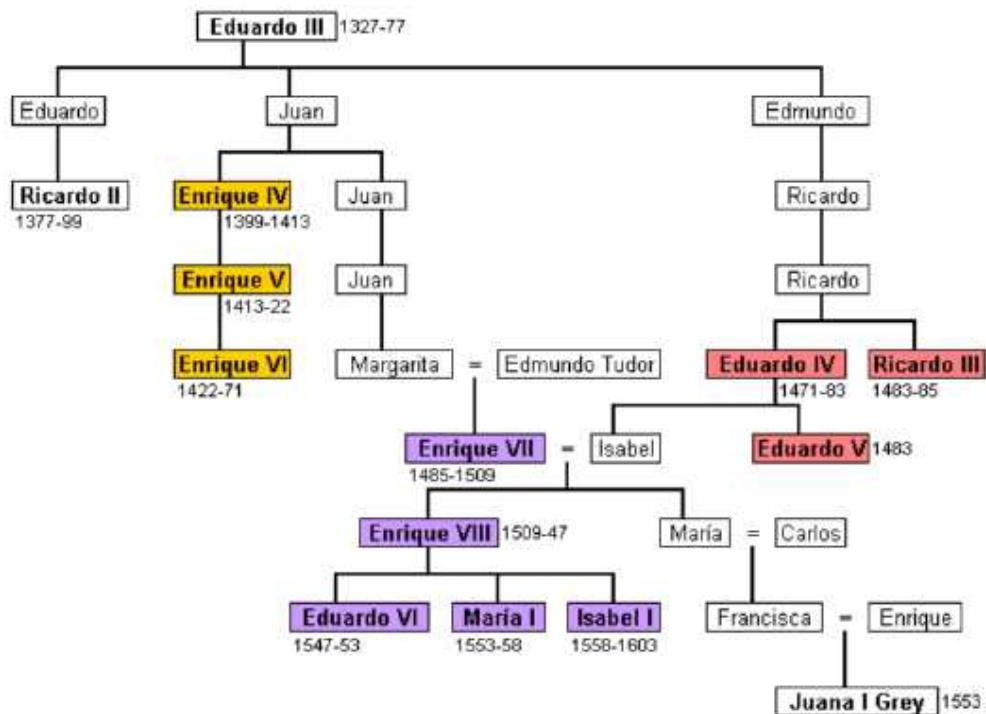
## Ejemplo: organigrama de una organización



<http://www.inf.uc3m.es/es/investigacion>

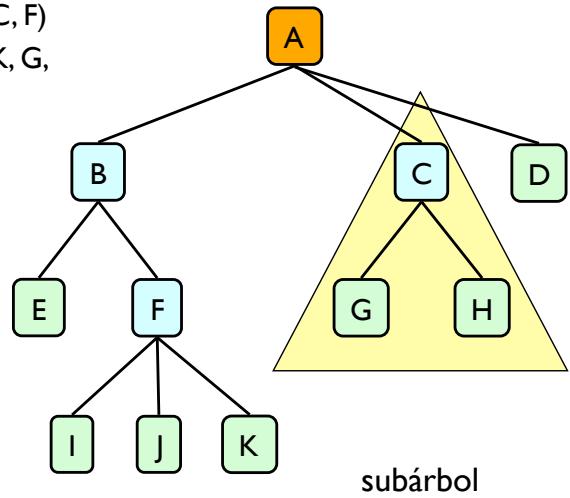
# Ejemplos: árbol genealógico

## FAMILIA TUDOR:



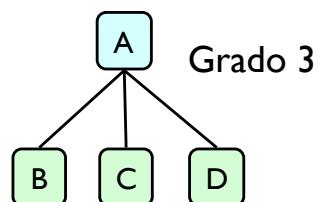
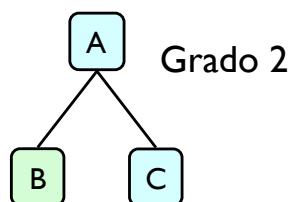
# Conceptos básicos

- ◆ **Raíz:** único nodo sin padre (A)
- ◆ **Nodo interno:** tiene al menos un hijo (A, B, C, F)
- ◆ **Nodo hoja (externo):** no tiene hijos (E, I, J, K, G, H, D)
- ◆ **Subárbol:** árbol formado por un nodo y sus descendientes
- ◆ **Ancestros y descendiente directos.**
- ◆ **Ancestros y descendientes.**



# Conceptos Básicos

- ◆ **Grado de un nodo:** número de descendientes directos

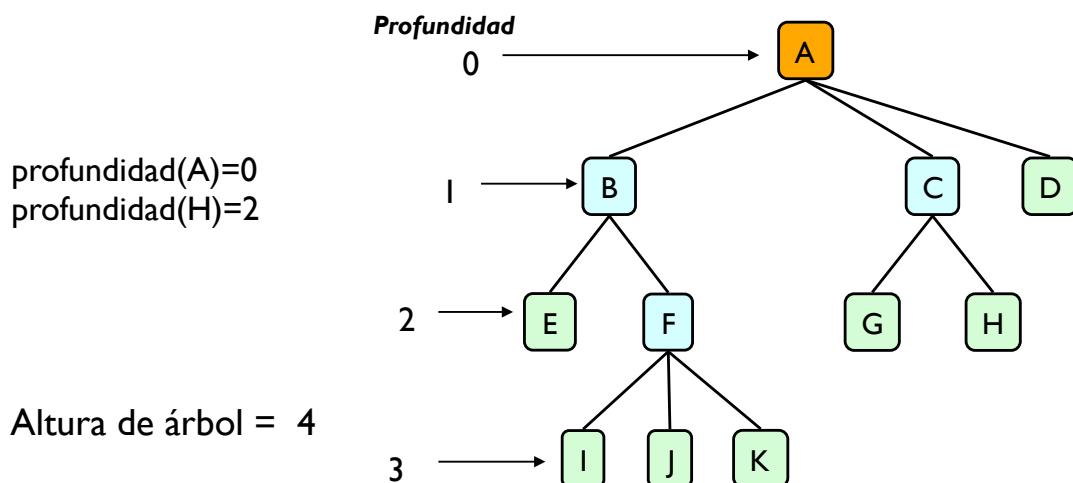


- ◆ **Grado del árbol:** mayor grado de sus nodos
- ◆ **Ejemplos:**
  - ◆ **Árbol binario:** árbol de grado 2
    - Cada nodo tiene como mucho dos descendientes directos
  - ◆ **Lista:** árbol degenerado de grado 1



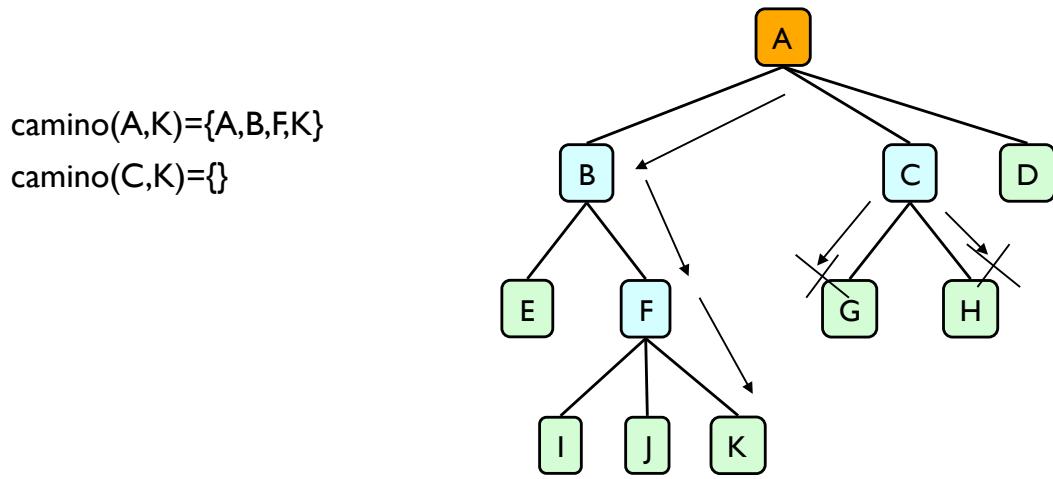
## Cónceptos básicos

- ◆ **Profundidad de un nodo:** número de predecesores
- ◆ **Altura del árbol:** longitud de la rama más larga más uno.

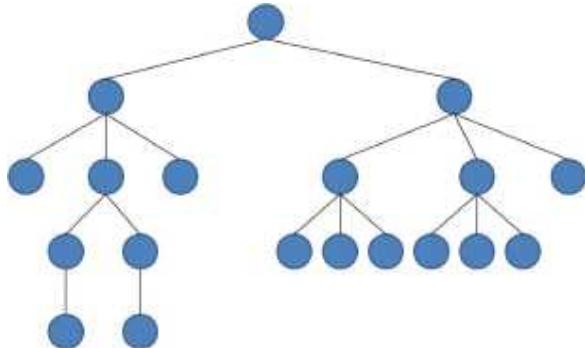


# Conceptos básicos

- ◆ **Camino:** existe un camino del nodo X al nodo Y, si existe una sucesión de nodos que permitan llegar desde X a Y. La sucesión debe tener un único sentido: ascendiente o descendiente.



# Árboles: Ejercicio I



1. Explica los valores de las principales características del árbol mostrado en la figura.
  - ¿grado del árbol?
  - ¿altura del árbol?
  - ¿número nodos del árbol?
  - ¿número de hojas?
  - ¿número de nodos internos?
2. Dadas las siguientes propiedades de un árbol, proporcione un dibujo que satisfaga las mismas:
  - Grado del árbol: 3
  - N° de Nodos: 14
  - Altura del árbol: 4
  - N° nodos con profundidad 2: 6



# Índice

---

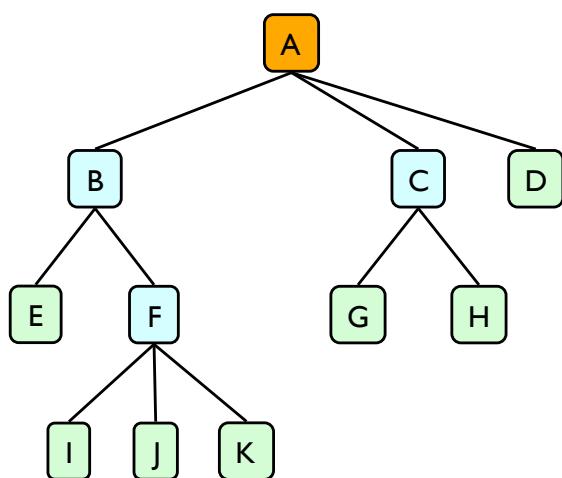
- ▶ Conceptos básicos
- ▶ **TAD Árboles generales**
  - ▶ Recorridos: post-orden, pre-orden y por niveles
- ▶ TAD Árboles binarios
- ▶ TAD Árboles binarios de búsqueda
- ▶ TAD Árboles B



## Árboles generales

---

- ▶ También llamados n-arios o multicamino
- ▶ Árboles con grado mayor a 2



## Árboles generales: Aplicación

---

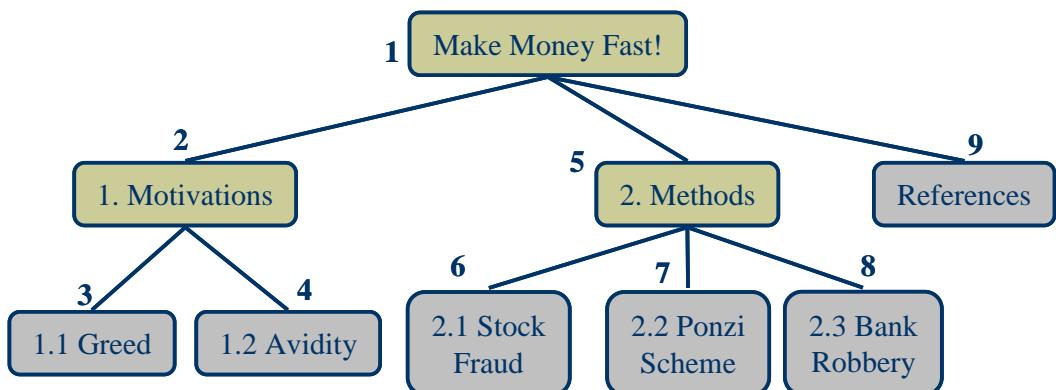
- ◆ Es la estructura utilizada para representar organizaciones jerárquicas donde cada elemento tiene un número variable de hijos.
- ◆ Ejemplos:
  - ◆ Representación de un sistema de ficheros, en el que cada directorio está enlazado con sus descendientes, ficheros o subdirectorios.
  - ◆ Representación de un árbol genealógico en el que cada persona se enlaza con sus descendientes



# Árboles generales: Recorridos

## Recorrido en pre-orden:

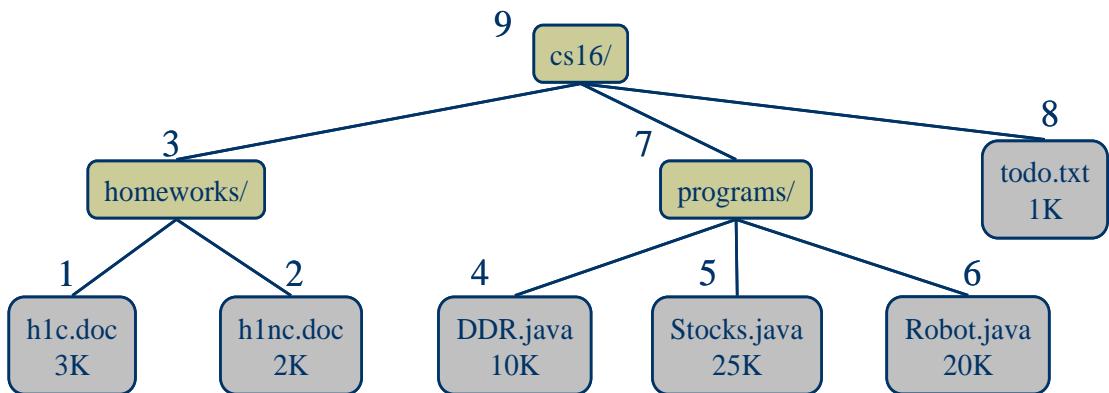
- se visita primero la raíz del árbol y luego se visitan recursivamente sus sub-árboles de izquierda a derecha (también en recorrido pre-orden)
- Ej, listar el contenido de un documento estructurado.



# Árboles generales: Recorridos

## Recorrido en post-orden:

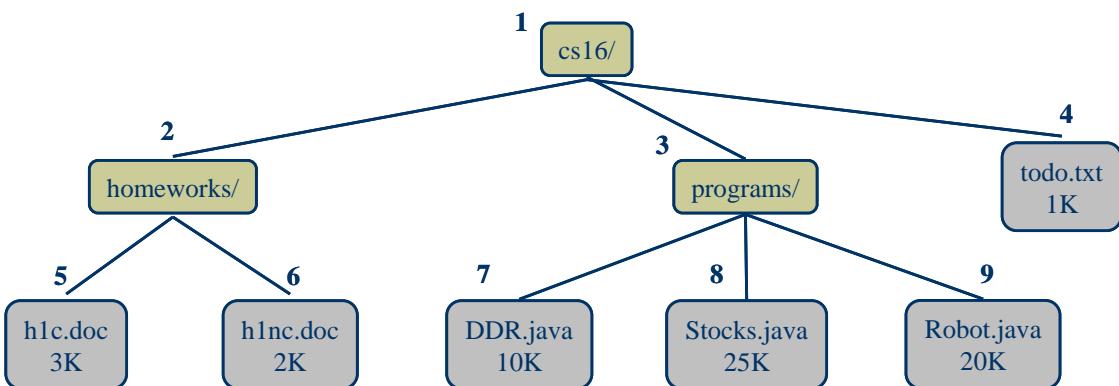
- los nodos se visitan después de haber visitado a sus hijos. Es decir, primero se recorren su sub-árboles (en recorrido post-orden) y por último se visita su raíz.
- Ej: Es útil para calcular el espacio en disco que ocupa un directorio.



# Árboles generales: Recorridos

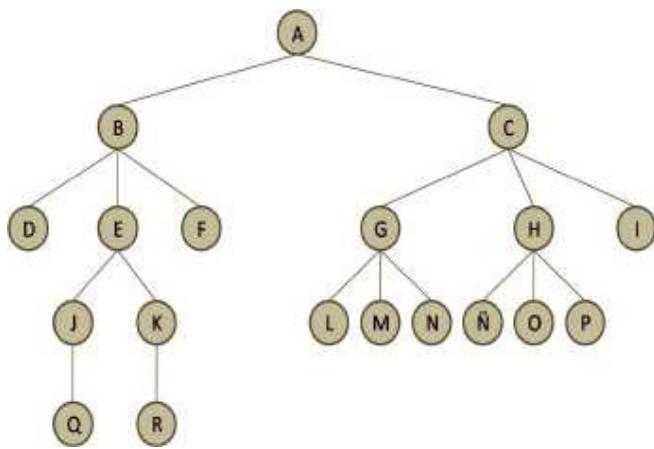
## Recorrido por niveles:

- se visita por profundidad. Es decir, vamos visitando los nodos del mismo nivel de forma descendente y de izquierda a derecha

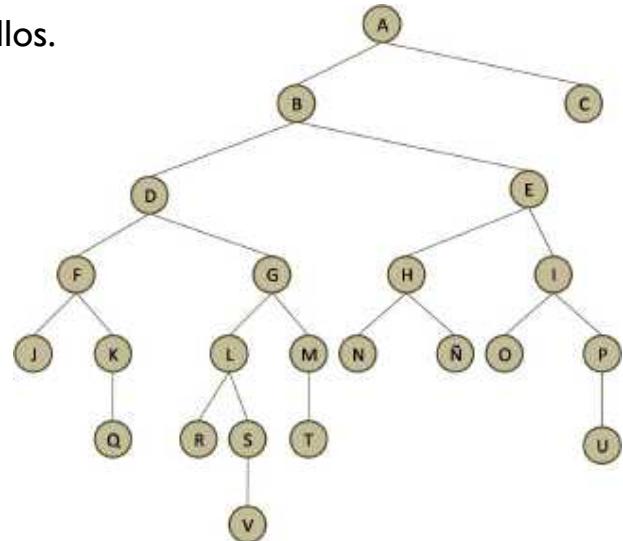


## Árboles: Ejercicio 2

Dados los siguientes árboles, escriba los recorridos pre-orden y post-orden y por niveles de cada uno de ellos.



A



B



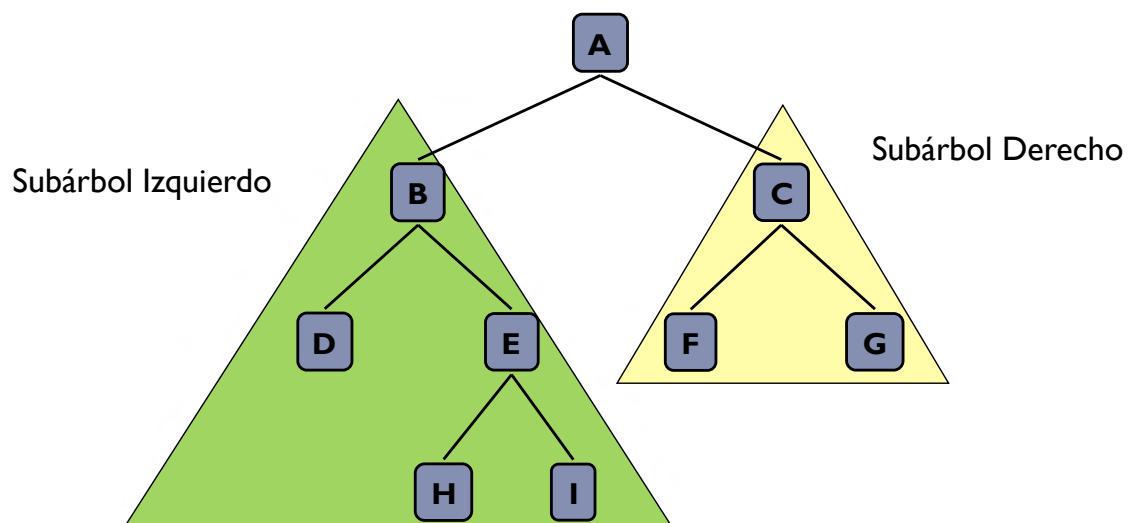
# Índice

---

- ▶ Conceptos básicos
- ▶ TAD Árboles generales
- ▶ **TAD Árboles binarios**
- ▶ TAD Árboles binarios de búsqueda
- ▶ TAD Árboles B

# Árboles Binarios

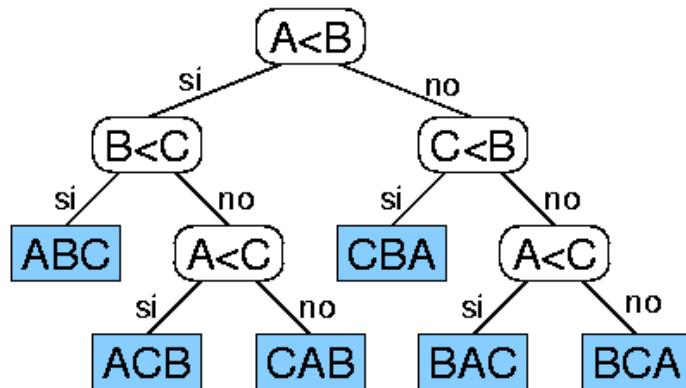
- ▶ Árbol de grado 2
  - ▶ Cada nodo tiene dos sub-árboles (pueden ser vacíos)



# Árboles binarios: Aplicación

## Ejemplo I: Árboles de decisión

- nodo interno: preguntas con respuesta si/no
- nodos hoja: decisiones



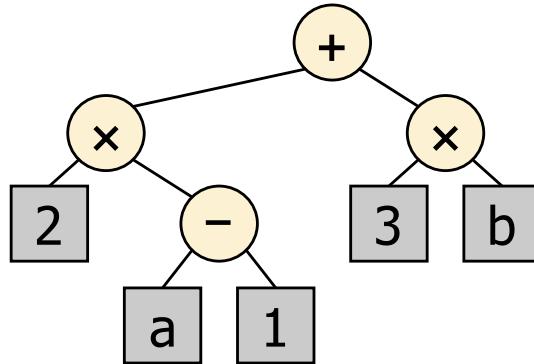
Ejemplo de un árbol de decisión para ordenar tres elementos A, B y C.

## Árboles binarios: Aplicación

---

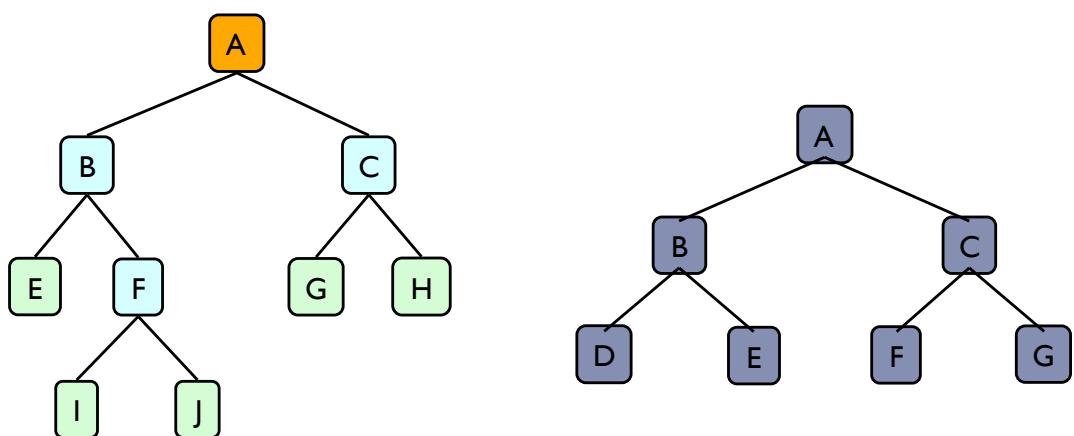
Ejemplo II: representar expresiones aritméticas

- nodo interno: operadores
- nodos hoja: operandos



# Árboles Binarios

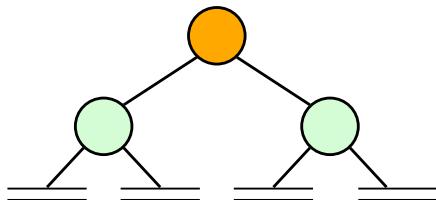
- ◆ Es **completo** si todo nodo interno (no hoja) tiene dos descendientes
- ▶ Está **lleno** si es completo y además todas sus hojas están en el mismo nivel



# Árboles binarios: propiedades

## ◆ Notación

- n: número de nodos = 3
- e: número de nodos hoja = 2
- i: número de nodos internos = 1
- h: altura del árbol = 1



## ◆ Propiedades

- $n = 2e - 1 = 3$
- $h \leq i \rightarrow 1 \leq 1$
- $h \leq (n-1)/2 \rightarrow 1 \leq 1$
- $e \leq 2^h \rightarrow 2 \leq 2$
- $h \geq \log_2 e \rightarrow 1 \geq 1$
- $h \geq \log_2(n+1) - 1 \rightarrow 1 \geq 1$

## ◆ Si es completo:

- $e = i + 1 \rightarrow 2 = 2$
- $e \geq h + 1 \rightarrow 2 \geq 2$

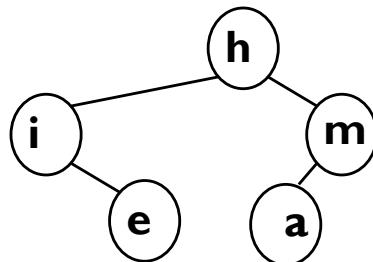


# Árboles binarios: Recorridos

## Recorrido PRE-ORDER

- primero se visita cada nodo, luego su subárbol izquierdo y finalmente el derecho (raiz, izq, der)
- Ejemplo:

pre-order: (h, i, e, m, a)

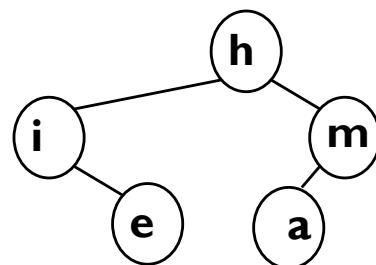


# Árboles binarios: Recorridos

## Recorrido POST-ORDEN

- cada nodo se visita después de visitar su subárbol izquierdo y después de visitar el derecho (izq, der, raiz)
- Ejemplo:

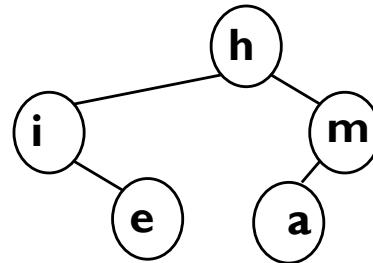
*post-order: (e, i, a, m, h)*



# Árboles binarios: Recorridos

## Recorrido POR NIVELES (LEVEL-ORDER)

- Se visitan los nodos en orden por nivel (en profundidad). Es decir, se visitan los nodos del mismo nivel de forma descendiente y de izquierda a derecha
- Ejemplo:  
*Level-order: (h,i,m,e,a)*

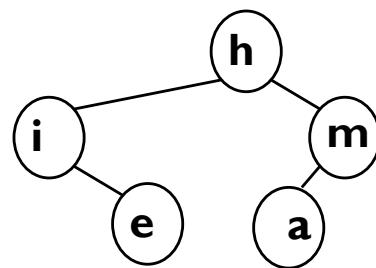


## Árboles binarios: Recorridos

### Recorrido IN-ORDER (nuevo en árboles binarios)

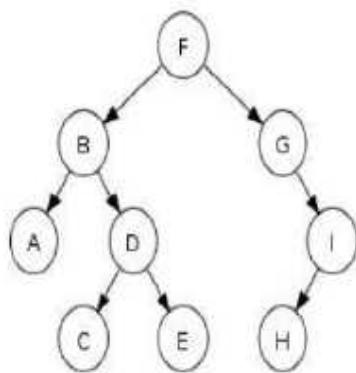
- cada nodo se visita tras visitar su subárbol izquierdo y antes de visitar el derecho (izq, raiz, der)
- Ejemplo:

*in-order: (i, e, h, a, m)*



## Ejemplo Recorridos

---



In this [binary search tree](#)

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right); note how this produces a sorted sequence
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

## Árboles binarios: Ejercicio 3

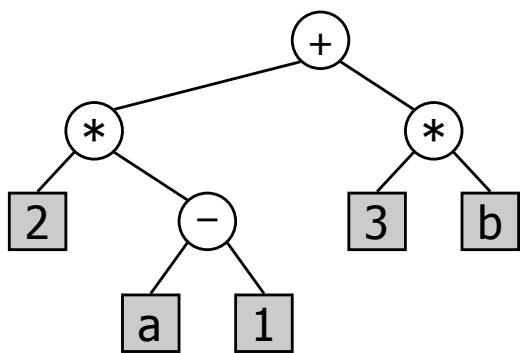
El recorrido en “pre-orden” de un árbol binario es: EXAMFUN y en “in-orden” MAFXUEN, donde cada carácter es un nodo.

- Dibujar el árbol binario.
- Dar el recorrido en post-orden.
- Dar el recorrido por niveles del árbol.



## Árboles binarios: Ejercicios de recorridos

- ◆ **Ejemplo: expresiones aritméticas**
  - pre-order: notación prefija (polaca)
  - in-order: notación normal (sin paréntesis)
  - post-order: notación polaca inversa

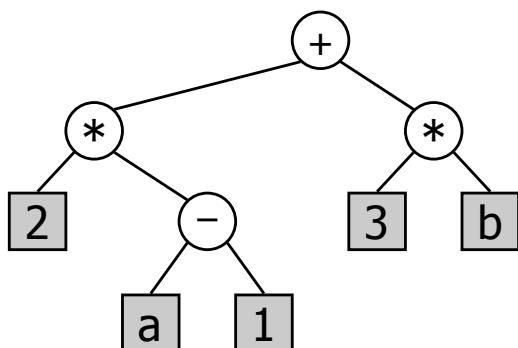


pre-order: + \* 2 - a 1 \* 3 b (raiz, izq, der)  
in-order: 2 \* a - 1 + 3 \* b (izq, raiz, der)  
post-order: 2 a 1 - \* 3 b \* + (izq, der, raiz)



## Ejercicio. Paréntesis en expresión matemática

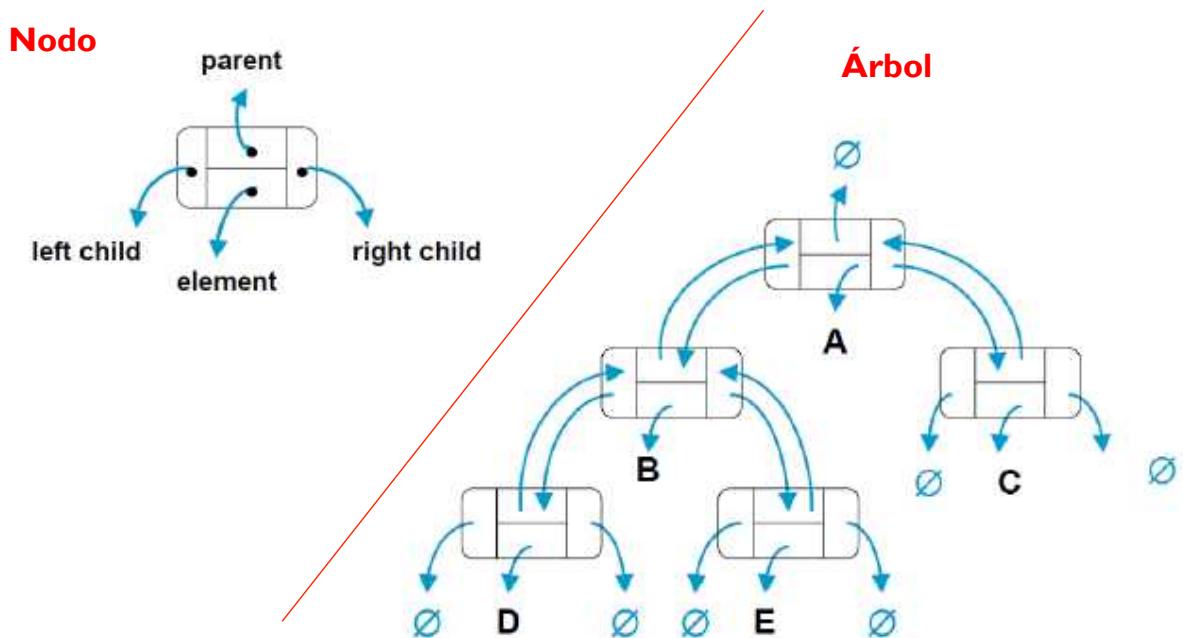
- ▶ Dada una expresión matemática en un TAD árbol binario
  - ▶ escribir (de forma teórica) el algoritmo para que el recorrido en in-order incluya los paréntesis



in-order:  $2 * (a - 1) + 3 * b$

Nota: es interesante poner todos los paréntesis, pero si sólo se añaden los necesarios para su correcto cálculo, mejor.

# Árboles binarios: Implementación



## Tema 5. Árboles Árboles Binarios de Búsqueda

Estructura de Datos y Algoritmos (EDA)

# Índice

---

- ▶ Conceptos básicos
- ▶ TAD Árboles generales
- ▶ TAD Árboles binarios
- ▶ **TAD Árboles Binarios de Búsqueda (ABB)**
- ▶ Equilibrado de árboles.

# Motivación de los ABB

---

- ▶ Los árboles binarios no ordenados son de poco interés.
  - ▶ Su única utilidad es la representación de información jerárquica (sólo grado 2!!!).
- ▶ La búsqueda en una lista ordenada es poco eficiente ( $O(n)$ ).
- ▶ Los árboles binarios de búsqueda son una solución eficiente para realizar búsquedas eficientes en colecciones ordenadas de elementos.
- ▶ En inglés: Binary Search Tree (BST)

## Árboles Binarios de Búsqueda (ABB)

---

- ▶ ABB = Árbol binario en el que TODOS sus nodos cumplen las siguientes condiciones:
  1. Cada nodo está asociado a una clave de ordenación. La clave puede ser de cualquier tipo siempre que sea un tipo comparable. Además de la clave, el nodo puede tener asociado también un elemento o valor.

Ejemplo: ABB que almacene una agenda de contactos. La clave de cada nodo puede ser el email del contacto y el valor podría ser el nombre del contacto.

**Nota:** Por simplificar, sólo representaremos las claves, que por lo general serán números enteros.



## Árboles Binarios de Búsqueda (ABB)

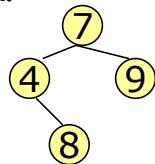
---

- ▶ ABB = Árbol binario en el que TODOS sus nodos cumplen las siguientes condiciones:
    1. Cada nodo está asociado a una clave de ordenación.
    2. Además para cada nodo, el valor de la clave de la raíz de su subárbol izquierdo es menor que el valor de la clave del nodo, y
    3. El valor de la clave raíz del subárbol derecho es mayor que el valor de la clave del nodo.
-

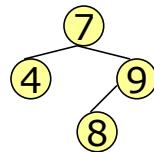
## Árboles Binarios de Búsqueda (ABB)

- ▶ ¿Cuál de estos dos árboles binarios de enteros es un ABB?

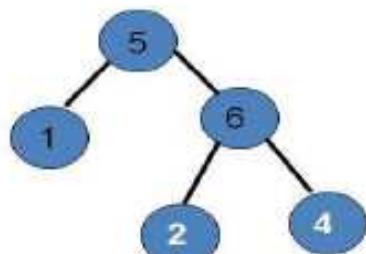
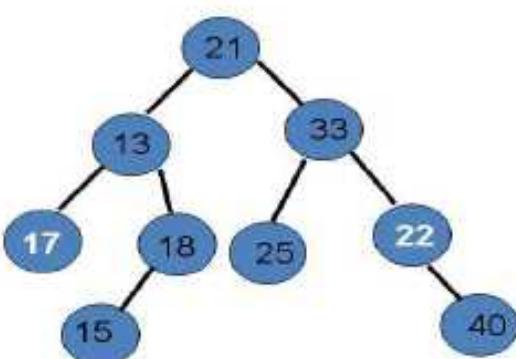
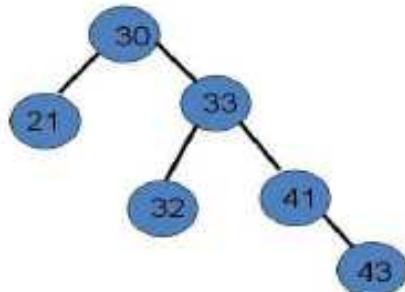
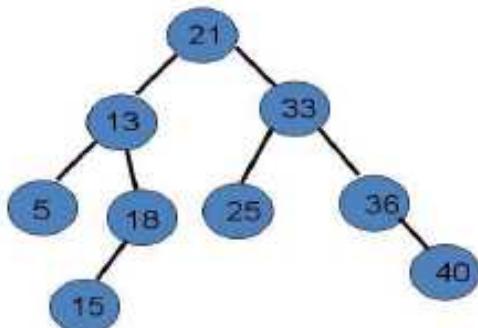
A:



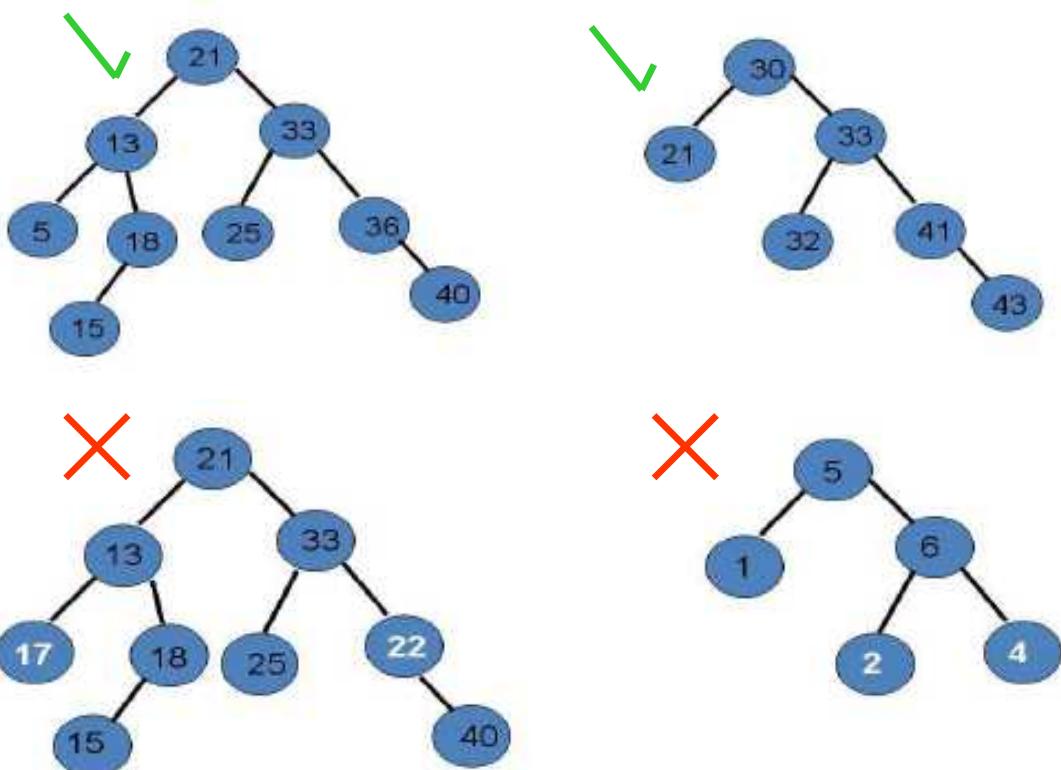
B:



# Árboles Binarios de Búsqueda (ABB)



## Árboles Binarios de Búsqueda (ABB)



# Árboles Binarios de Búsqueda (ABB)

---

- ▶ Utilidad
  - ▶ Almacenar estructuras lineales (que normalmente serían listas) **mejorando la complejidad de las búsquedas**
    - ▶ En el caso peor
    - ▶ En el caso medio
- ▶ Motivación: AB no ordenados son de poco interés
  - ▶ La falta de ordenación en un AB hace injustificable una estructura enlazada de árbol, prefiriéndose una lista
- ▶ Problema
  - ▶ un ABB puede llegar a degenerar en una lista
- ▶ Solución: **equilibrio en altura de un ABB** (**tema siguiente**)



# Árboles Binarios de Búsqueda (ABB)

---

## ► Ventajas

- Permite almacenar una colección ordenada de elementos de una forma más eficiente al mejorar **la complejidad de las búsquedas.**
- El número de accesos al árbol es menor que en una lista.



## Especificación de un ABB

---

- ▶ Aunque las claves y valores de un nodo binario de búsqueda pueden ser de cualquier tipo (las claves deben ser de un tipo comparable), por simplificar nos centraremos en un ABB con claves enteras y valores de tipo String.

## Especificación de un TAD ABB (=BST)

```
public interface IBSTree{  
  
    //number of nodes  
    public int getSize();  
  
    //length of the largest plus 1  
    public int getHeight();  
  
    //shows the preorder tree trasversal  
    public void showPreOrder();  
  
    //shows the in-order tree trasversal  
    public void showInOrder();  
  
    //shows the post-order tree trasversal  
    public void showPostOrder();  
  
    //shows the level order tree trasversal  
    public void showLevelOrder();
```

## Especificación de un TAD ABB (cont.)

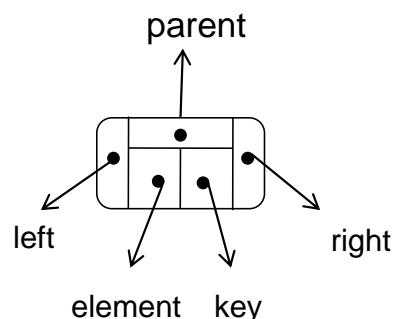
---

```
//inserts a new node
public void insert(int key, String elem);
//removes the node with key
public void remove(int key);
//searches the node with key and returns its elem
public String find(int key);
```

## Árboles binarios de búsqueda: Implementación Clase **BSTNode**

```
public class BSTNode {  
    public int key;  
    public String elem;  
  
    public BSTNode parent;  
    public BSTNode leftChild;  
    public BSTNode rightChild;  
  
    public BSTNode(int key, String element) {  
        this.key = key;  
        this.elem = element;  
    }  
}
```

La clave tiene que ser siempre de  
un tipo comparable



## Árboles binarios de búsqueda: Implementación **Clase BSTree**

---

```
public class BSTree implements IBSTree {  
  
    BSTNode root;  
  
    public BSTree() {  
  
    }  
    public BSTree(BSTNode root) {  
        this.root=root;  
    }  
}
```

## Implementación del método getSize

El tamaño del un árbol se puede ver como el tamaño del subárbol que cuelga de la raíz

```
public int getSize() {  
    return getSize(root);  
}  
  
//auxiliary recursive method to calculate the size of a subtree (node)  
public int getSize(BSTNode node) {  
    if (node == null) {  
        return 0;  
    } else {  
        int result = 1 + getSize(node.leftChild) + getSize(node.rightChild);  
        return result;  
    }  
}
```

Podemos usar un método auxiliar para definir la recursión sobre un nodo cualquiera.

## Método showInOrder()

---

```
public void showInOrder() {  
    showInOrder(root);  
}  
public void showInOrder(BSTNode node) {  
    //base case, we do not show anything  
    if (node == null) return;  
  
    //first, we visit the left child  
    showInOrder(node.leftChild);  
    //now, we visit the root.  
    System.out.println(node.elem);  
    //finally, we visit the right child  
    showInOrder(node.rightChild);  
}
```

## Problemas a resolver

---

- ▶ Ejercicio implementa el método **getHeight** que devuelve la altura de un nodo. Recuerda que la altura de un árbol es la longitud de su rama más larga + 1.
- ▶ Implementa el método que devuelve la altura del árbol.
- ▶ Implementa el método **getDepth** que devuelve la profundidad de un nodo.
- ▶ Implementa los métodos **showPreOrder** y **showPostOrder**

## TAD ABB: Búsqueda

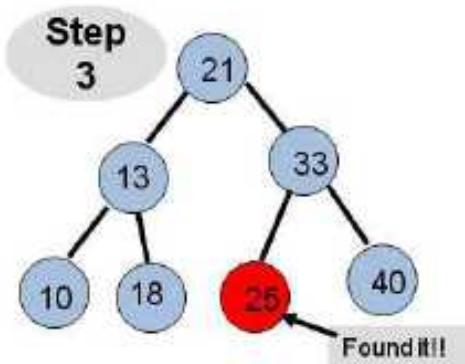
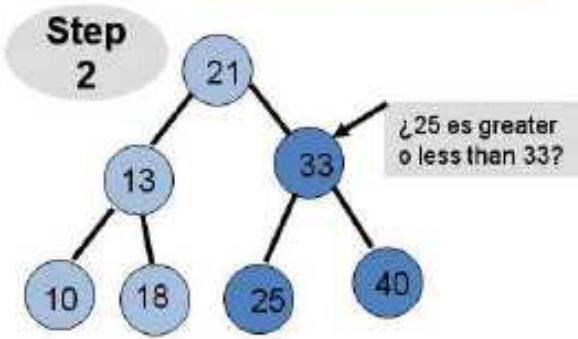
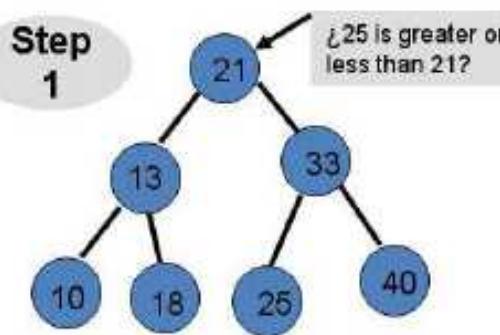
---

- ▶ Funcionamiento:
  - ▶ se va recorriendo el árbol
  - ▶ si el nodo actual no es el buscado se decide si hay que buscar por la derecha o la izquierda
  - ▶ el algoritmo para al encontrar el nodo o llegar al árbol vacío
- ▶ Puede desarrollarse:
  - ▶ como algoritmo recursivo del nodo del árbol
  - ▶ como algoritmo iterativo del árbol



## TAD ABB: ejemplo de búsqueda

Search 25



## Búsqueda (versión iterativa)

```
public String findIt(int key) {  
  
    BSTNode searchNode=root;  
    while (searchNode!=null) {  
        int keyVisit=searchNode.key;  
        if (key==keyVisit) {  
            //found it!!!  
            return searchNode.elem;  
        } else if (key<keyVisit) {  
            searchNode=searchNode.leftChild;  
        } else {  
            searchNode=searchNode.rightChild;  
        }  
    }  
    System.out.println(key + " does not exist");  
    return null;  
}
```

# Búsqueda recursiva

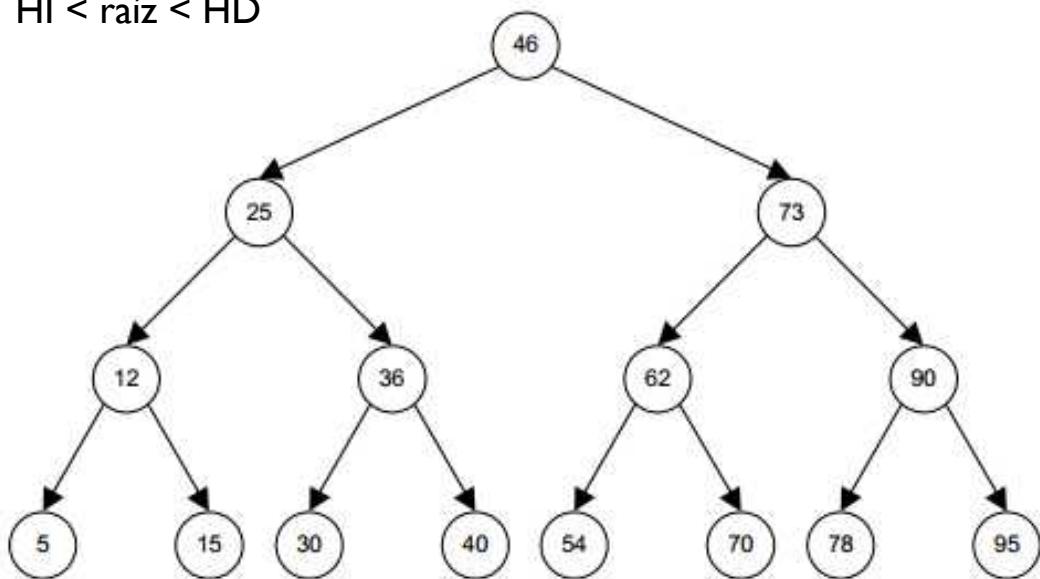
```
public String find(int key) {  
    return find(root, key);  
}  
//auxiliary recursive method to search a node  
//inside a subtree  
  
private static String find(BSTNode currentNode, Integer key) {  
    String result=null;  
    if (currentNode == null) {  
        //System.out.println(key + " does not exist!");  
    } else {  
        if (key.equals(currentNode.key))  
            result= currentNode.elem;  
        else if (key.compareTo(currentNode.key) < 0)  
            result=find(currentNode.left, key);  
        else  
            result=find(currentNode.right, key);  
    }  
    return result;  
}
```

# Árboles Binarios de Búsqueda (ABB)

**¿Cuál es la complejidad de la operación de búsqueda?**

**Buscar 70**

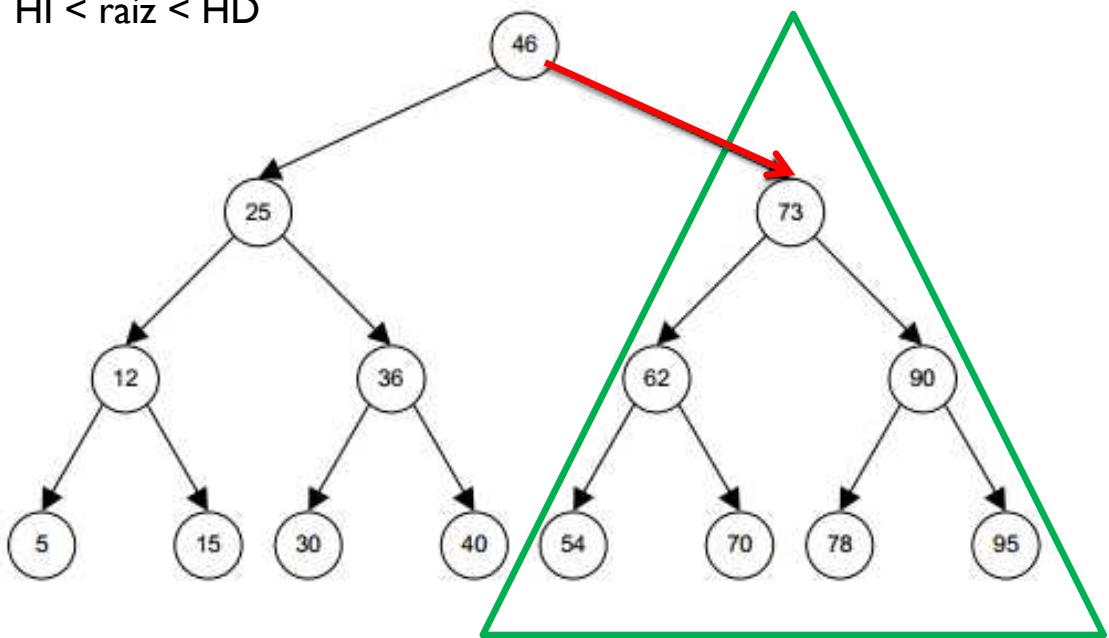
$HI < \text{raíz} < HD$



# Árboles Binarios de Búsqueda (ABB)

**Buscar 70**

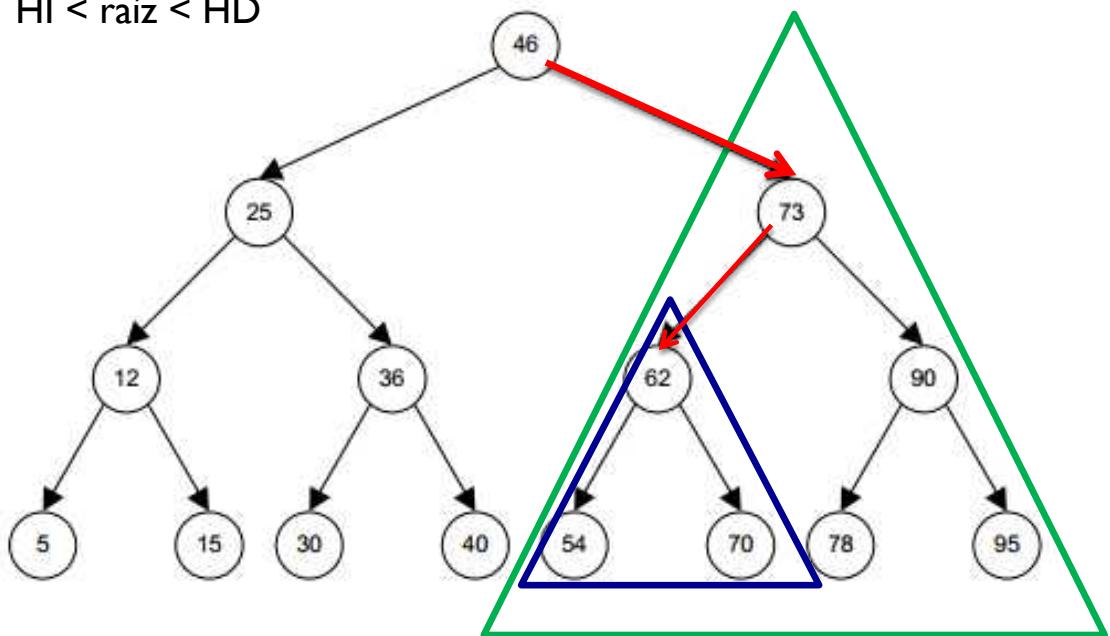
HI < raíz < HD



# Árboles Binarios de Búsqueda (ABB)

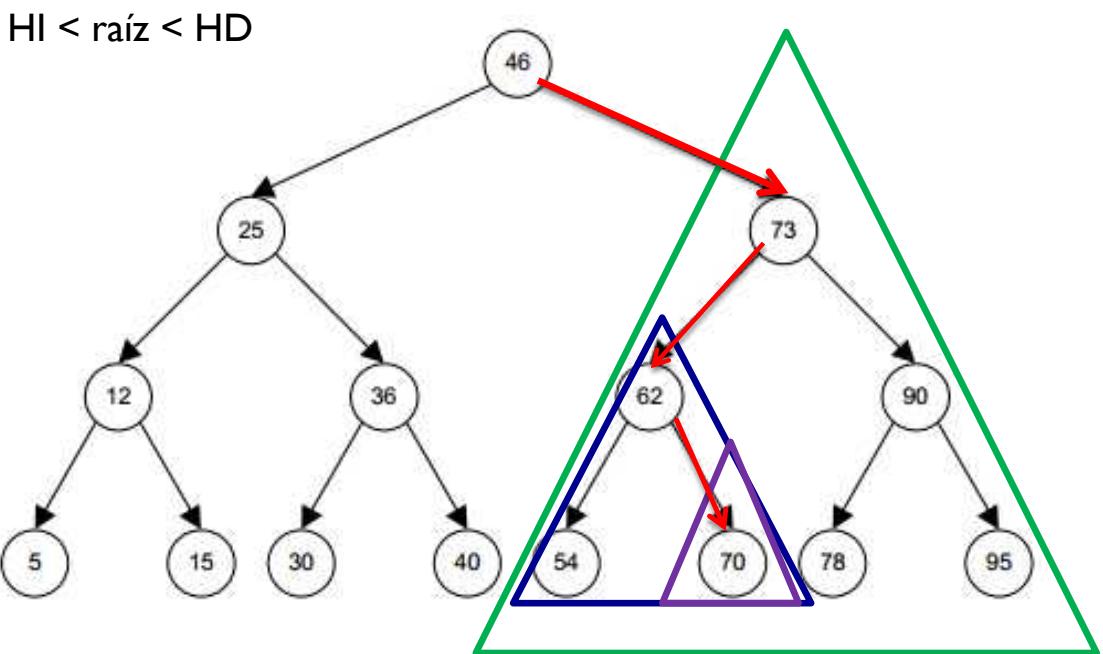
**Buscar 70**

HI < raíz < HD



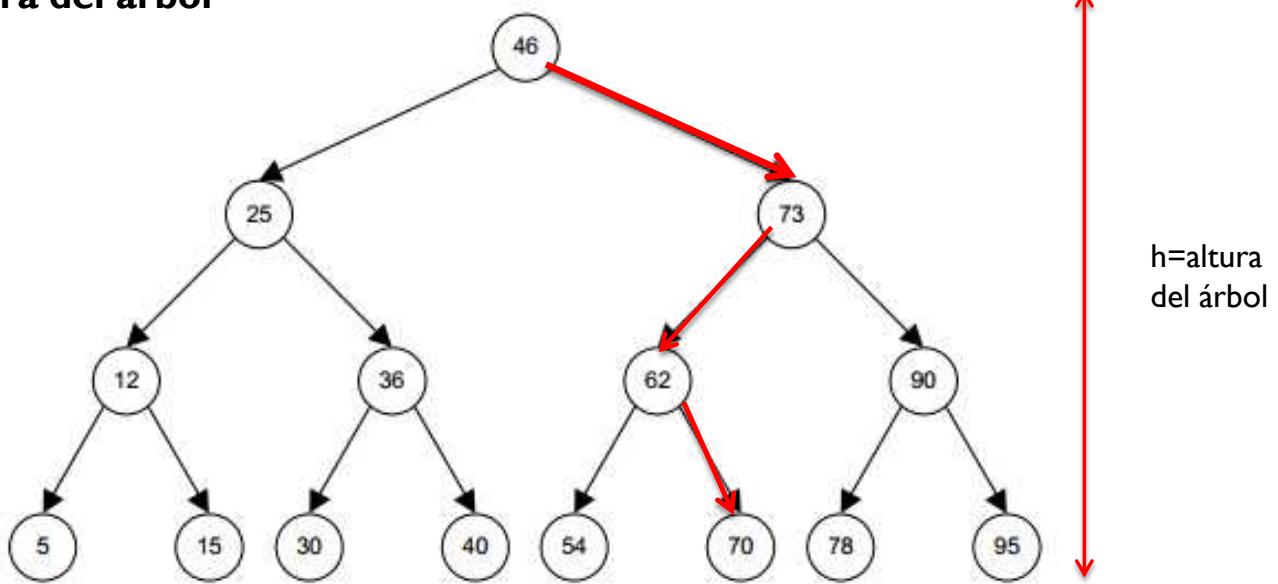
# Árboles Binarios de Búsqueda (ABB)

**Buscar 70**



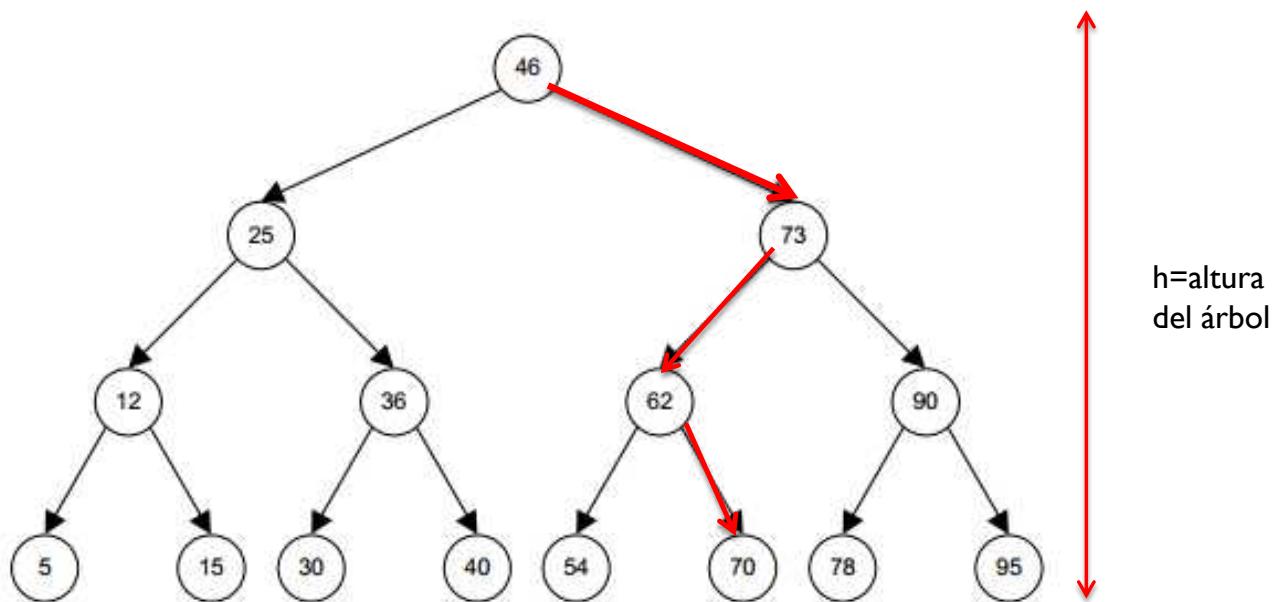
# Árboles Binarios de Búsqueda (ABB)

**En el peor de los casos, en la búsqueda se realiza  $h$  comparaciones, donde  $h$  es la altura del árbol**



# Árboles Binarios de Búsqueda (ABB)

**Por tanto, la complejidad será  $O(h)$  donde  $h$  es la altura del árbol**

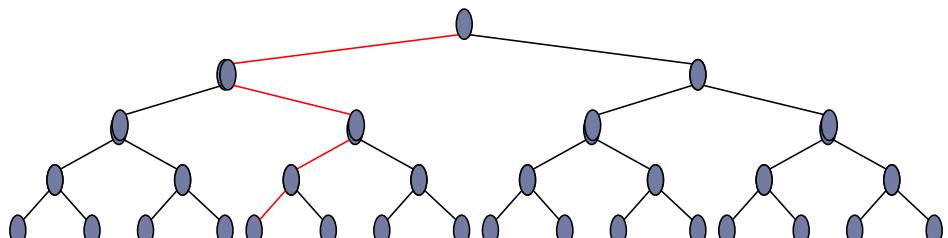


# Árboles Binarios de Búsqueda (ABB)

Siempre dividimos el espacio de búsqueda en 2

- ◆  $\text{Paso}_1 = n / 2^1$
- ◆  $\text{Paso}_2 = n / 2^2$
- ◆  $\text{Paso}_3 = n / 2^3$
- ◆ .
- ◆ .
- ◆  $\text{Paso}_h = n / 2^h = 1$ , donde  $h$  es la altura del árbol

↓  
 $n = 2^h$   
 $h = \log_2(n)$

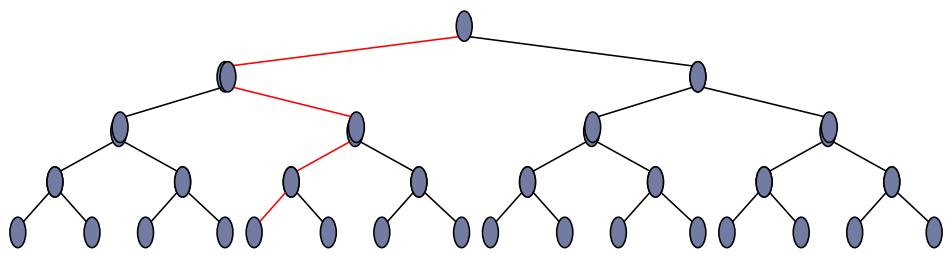


**En el peor de los casos, el número de pasos necesarios para buscar un nodo será igual a la altura del árbol = h**

# Árboles Binarios de Búsqueda (ABB)

Siempre dividimos el espacio de búsqueda en 2

- ◆  $\text{Paso}_1 = n / 2^1$
- ◆  $\text{Paso}_2 = n / 2^2$
- ◆  $\text{Paso}_3 = n / 2^3$
- ◆ .
- ◆ .
- ◆  $\text{Paso}_h = n / 2^h = 1$ , donde  $h$  es la altura del árbol



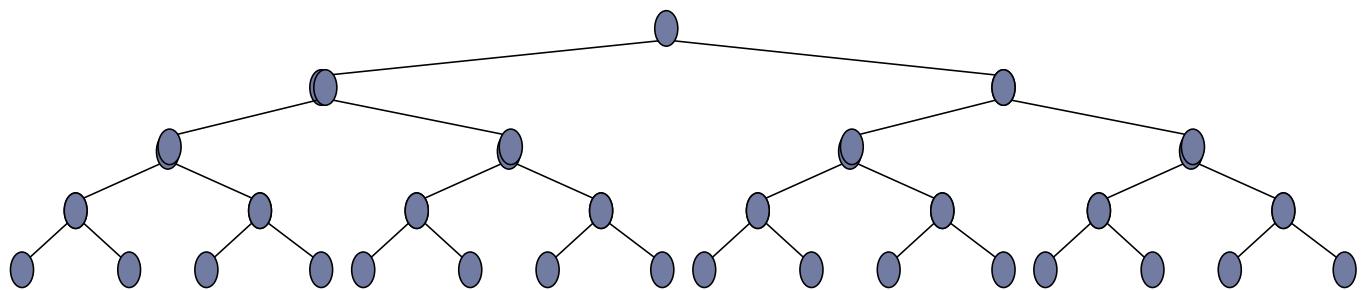
↓  
 $n = 2^h$   
 $h = \log_2(n)$

$O(h) = O(\log_2 n)$



# Árboles Binarios de Búsqueda (ABB)

Complejidad Búsqueda binaria  $O(\log_2 n)$



En un árbol de  $n$  nodos, el camino más largo que hay que recorrer es de orden  $\log n$ .

- Si  $n=32 \Rightarrow \log 32 = 5$
- Si  $n=1024 \Rightarrow \log 1024= 10$

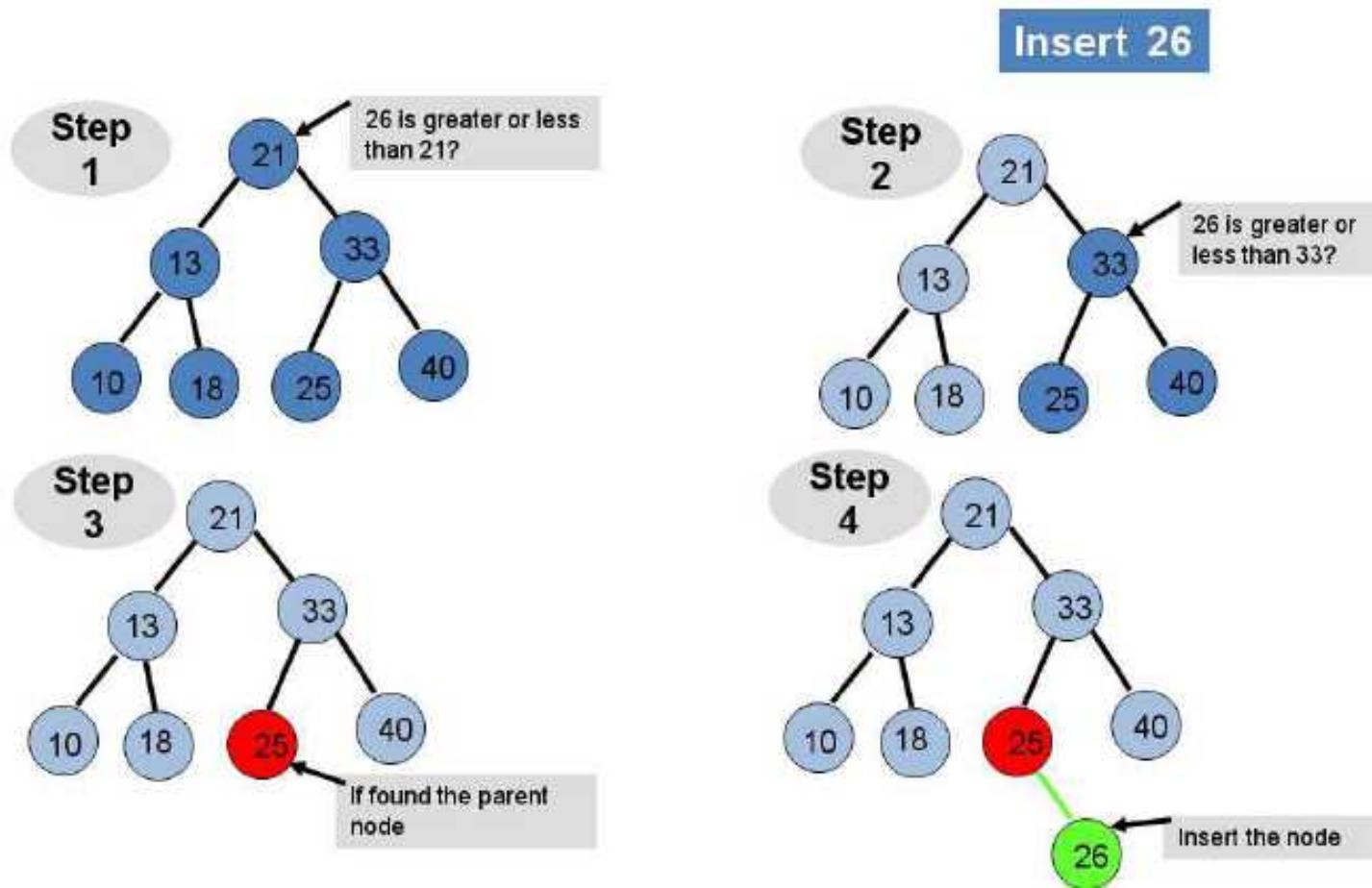
## TAD ABB: inserción

---

- ▶ Los nodos se insertan siempre como nodos hoja
- ▶ El algoritmo de inserción garantiza para cada nodo del árbol que:
  - ▶ Su subárbol izquierdo contiene claves menores
  - ▶ Su subárbol derecho contiene claves mayores
- ▶ Funcionamiento:
  - ▶ si el árbol estuviera vacío, se inserta el nodo en la raíz
  - ▶ si no, se va recorriendo el árbol:
    - ▶ en cada nodo se decide si hay que insertar a la derecha o la izquierda
    - ▶ si el subárbol en que hay que insertar es vacío, se inserta el nuevo elemento
    - ▶ si el subárbol en que hay que insertar no es vacío hay que recorrerlo hasta encontrar el lugar que le corresponde al nodo en ese subárbol
  - ▶ es un algoritmo recursivo

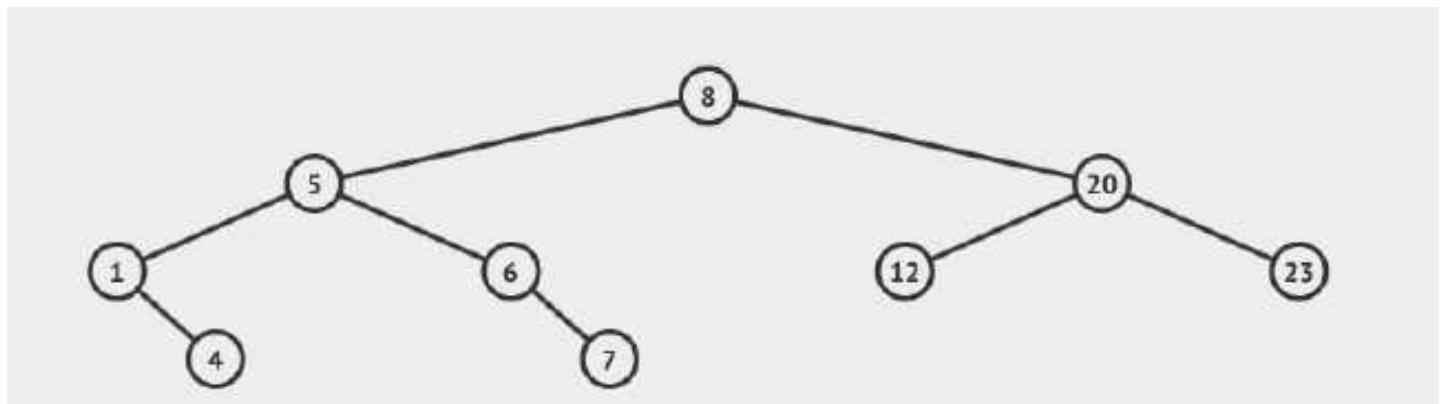


## TAD ABB: ejemplo de inserción



## TAD ABB: ejemplo de inserción

Insertar 8, 5, 1, 4, 6, 7, 20, 12, 23



## ABB: Implementación Inserción (1/2)

---

```
public void insert(int key, String element) {  
    BSTNode newNode=new BSTNode(key,element);  
    if (root==null) root=newNode;  
    else insert(newNode, root);  
}
```

## ABB: Implementación Inserción (2/2)

```
public void insert(BSTNode newNode, BSTNode node) {  
  
    int key=newNode.key;  
    if (key==node.key) {  
        System.out.println(key + " already exists. Duplicates are not allowed!!!.");  
        return;  
    }  
    if (key<node.key) {  
        if (node.leftChild==null) {  
            node.leftChild=newNode;  
            newNode.parent=node;  
        } else insert(newNode,node.leftChild);  
    } else {  
        if (node.rightChild==null) {  
            node.rightChild=newNode;  
            newNode.parent=node;  
        } else insert(newNode,node.rightChild);  
    }  
}
```

## TAD ABB: Borrado

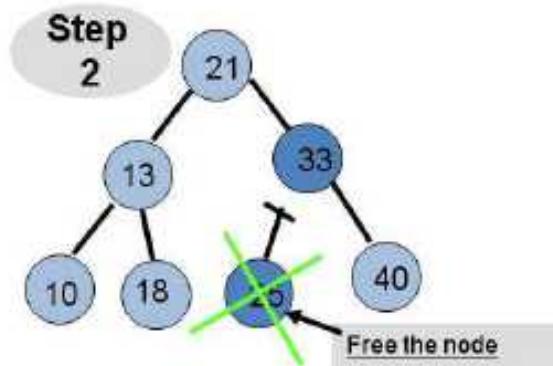
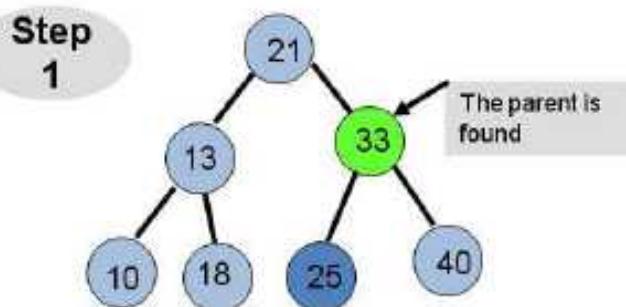
---

- ▶ Funcionamiento: Buscar el nodo a borrar,
  - 1 Si es un **nodo hoja**, basta con que su padre haga referencia a vacío
  - 2 Si **no es nodo hoja** hay que sustituirlo por otro
    - a) El nodo a borrar sólo tiene un hijo: sustituirlo por su hijo
    - b) El nodo a borrar tiene dos hijos, sustituirlo por:
      - El **mayor** de su subárbol izquierdo (predecesor) o
      - El **menor** de su subárbol derecho (sucesor)
  - En realidad, no se sustituye un nodo por otro, sino que se reemplazan la clave y valor del nodo por la clave y valor del predecesor (o sucesor, según la estrategia escogida). Además, será necesario eliminar dicho nodo predecesor (o sucesor), que siempre será un nodo hoja o un nodo con un único hijo.
- ▶ Si el nodo a borrar es la raíz, hay que modificar la raíz, aplicando el caso que corresponda



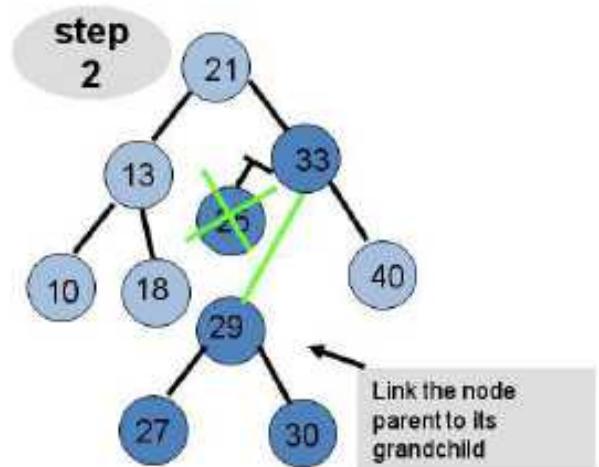
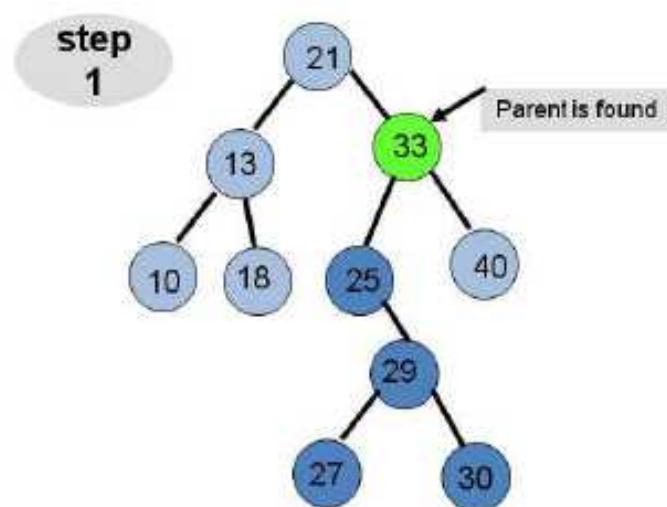
## TAD ABB: Ejemplo de borrado

Remove 25

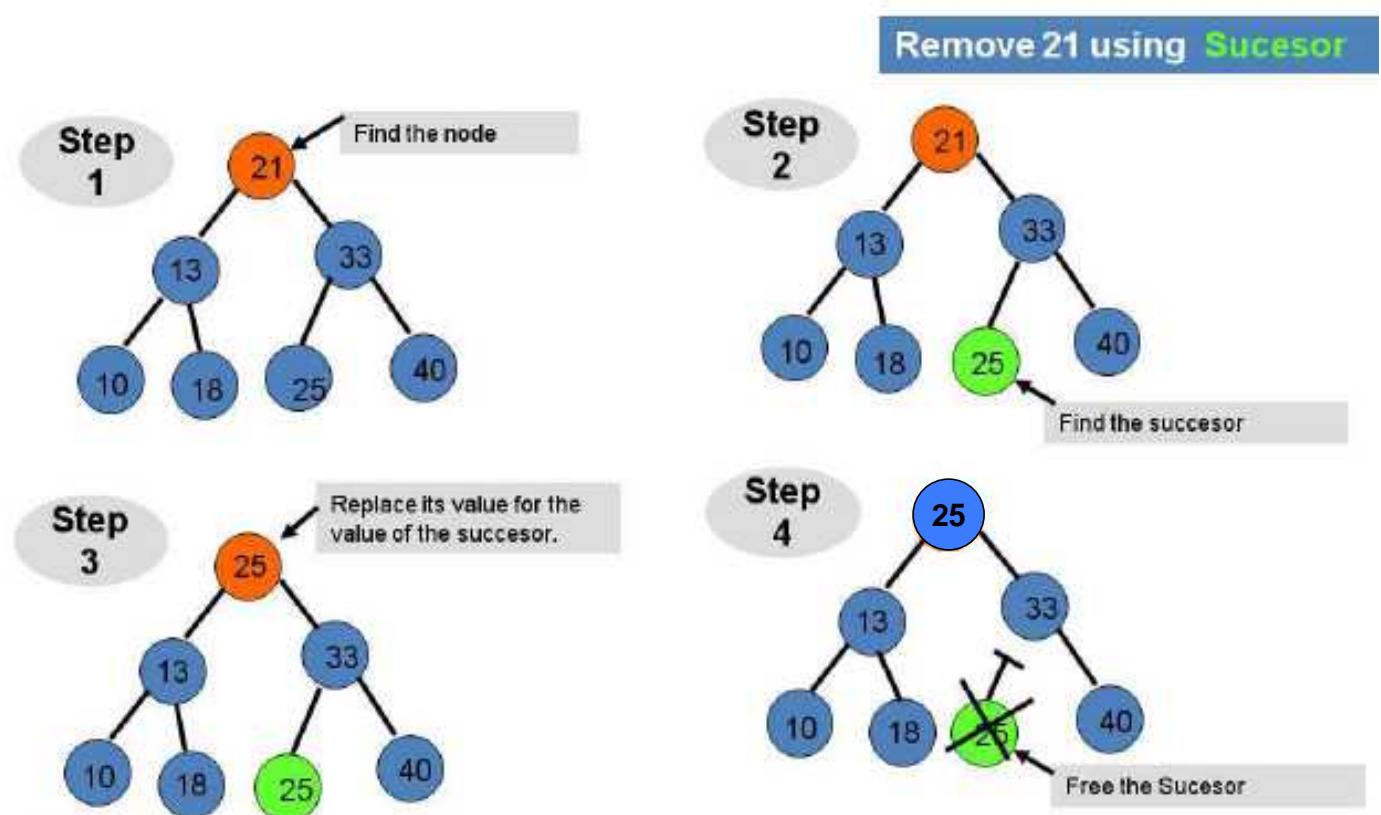


## TAD ABB: Ejemplo de borrado

Remove 25



## TAD ABB: Ejemplo de borrado



## ABB: Implementación Borrado (1/10)

---

```
public void remove(int key) {  
    if (root == null) {  
        System.out.println("Cannot remove: The tree is empty");  
        return;  
    }  
  
    //removing the root is a special case  
    if (key==root.key) removeRoot();  
    else remove(key,root);  
}
```

## ABB: Implementación Borrado (2/10)

---

```
public void removeRoot() {  
    //if the root is a leave, then root should be null  
    if (root.leftChild==null && root.rightChild==null) root=null;  
    //the root only has a child  
    else if (root.leftChild==null ||root.rightChild==null) {  
        if (root.leftChild==null) root=root.rightChild;  
        else root=root.leftChild;  
        root.parent=null;  
    } else {  
        remove(root.key,root);  
    }  
}
```

## ABB: Implementación Borrado (3/10)

Lo primero es encontrar el nodo a borrar. Eso se puede conseguir mediante llamadas recursivas del método remove sobre el subárbol izquierdo o derecho, dependiendo de si la clave buscada es menor o mayor que la clave del nodo actual.

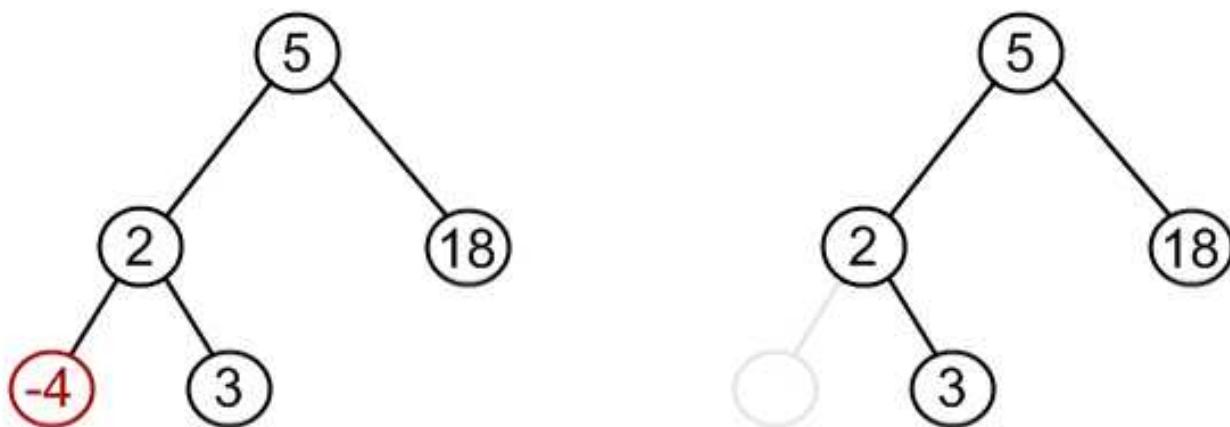
```
private boolean remove(int key, BSTNode node) {  
    if (node == null) {  
        System.out.println("Cannot remove: The key doesn't exist");  
        return false;  
    }  
  
    if (key < node.key) return remove(key, node.leftChild);  
  
    if (key > node.key) return remove(key, node.rightChild);
```

Nota que si llegamos a un nodo nulo, quiere decir que la clave no ha sido encontrada.

## ABB: Implementación Borrado (4/10)

I. Una vez localizado, distinguimos 3 casos:

**PRIMER CASO:** El nodo a borrar es una hoja => Rompemos la referencia del padre a ese nodo.



## ABB: Implementación Borrado (5/10)

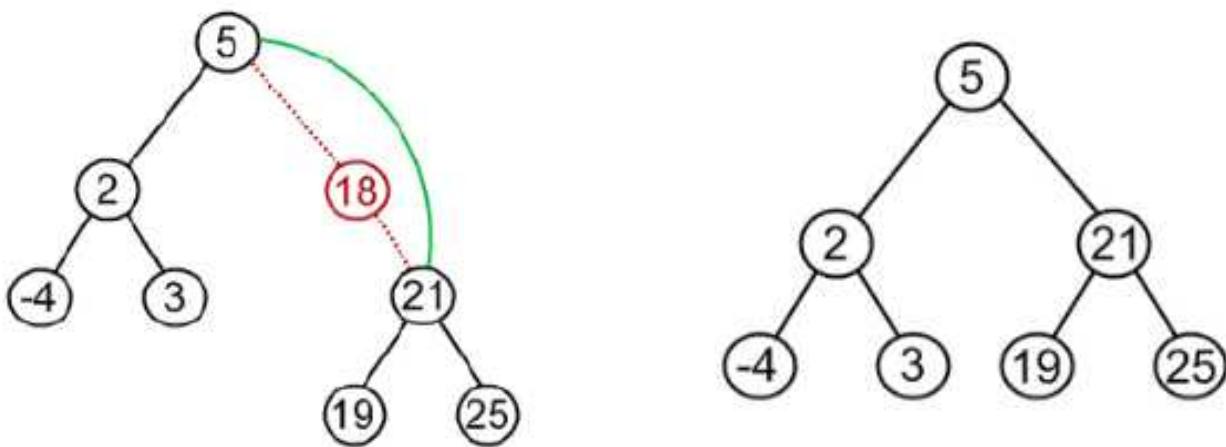
### I. Una vez localizado, distinguimos 3 casos:

**PRIMER CASO: El nodo a borrar es una hoja => Rompemos la referencia del padre a ese nodo.**

```
//First case: the node is a leaf.  
if (node.leftChild==null && node.rightChild==null) {  
    BSTNode parent=node.parent;  
  
    //we must break the references between the node and its parent.  
    //We have to find out if node is the left  
    //or right child of its parent.  
    if (parent.leftChild==node) parent.leftChild=null;  
    else parent.rightChild=null;  
    return true;  
}
```

## ABB: Implementación Borrado (6/10)

**SEGUNDO CASO: El nodo a borrar tiene un único hijo => su hijo**  
debe ser conectado directamente al padre del nodo que queremos borrar



## ABB: Implementación Borrado (7/10)

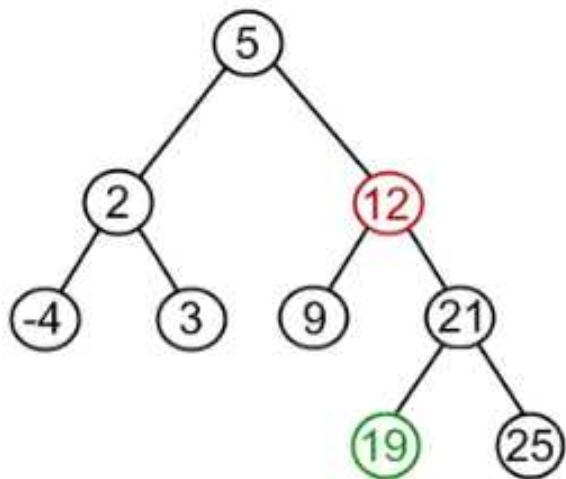
**SEGUNDO CASO: El nodo a borrar tiene un único hijo => su hijo** debe ser conectado directamente al padre del borro que queremos borrar

```
//Second case is one the node only has a child: left or right
if (node.leftChild==null || node.rightChild==null){
    //its only child is its right child
    BSTNode grandChild=null;
    if (node.leftChild==null)
        grandChild=node.rightChild;
    else
        grandChild=node.leftChild;

    BSTNode grandParent=node.parent;
    if (grandParent.leftChild==node)
        grandParent.leftChild=grandChild;
    else
        grandParent.rightChild=grandChild;
    //the grand child must point its grand parent.
    grandChild.parent=grandParent;
    return true;
}
```

## ABB: Implementación Borrado (8 / 10)

**TERCER CASO: El nodo a borrar tiene dos hijos =>**



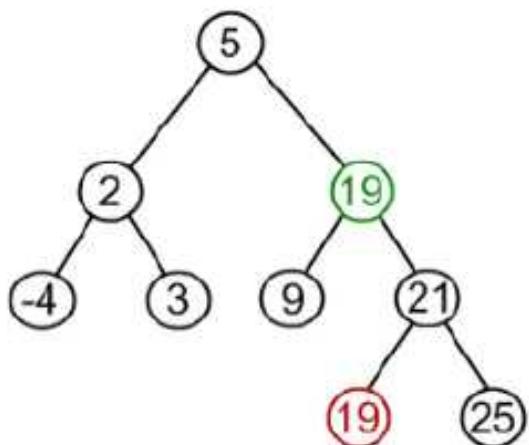
Buscamos su sucesor. También se puede usar el predecesor

```
private BSTNode findMin(BSTNode node) {  
    if (node==null) return null;  
    BSTNode minNode=node;  
    while (minNode.leftChild!=null) {  
        minNode=minNode.leftChild;  
    }  
    return minNode;  
}
```

Para encontrar el sucesor, deberemos buscar en su hijo derecho, el nodo con la key más pequeña.

## ABB: Implementación Borrado (9 / 10)

**TERCER CASO: El nodo a borrar tiene dos hijos =>**



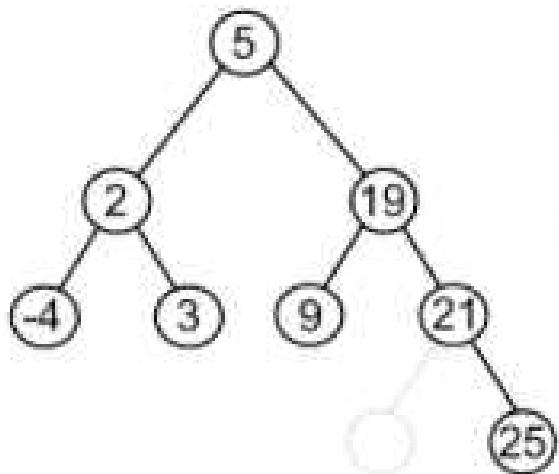
```
//Third case: node has two childs.  
//We can replace its value by  
//the minimum value in its right child  
  
BSTNode sucesorNode = findMin(node.rightChild);  
  
node.elem=sucesorNode.elem;  
node.key=sucesorNode.key;
```

Reemplazamos su clave y elemento por los del nodo sucesor.

# ABB: Implementación Borrado (10/10)

**TERCER CASO: El nodo a borrar tiene dos hijos =>**

Finalmente, borramos el sucesor



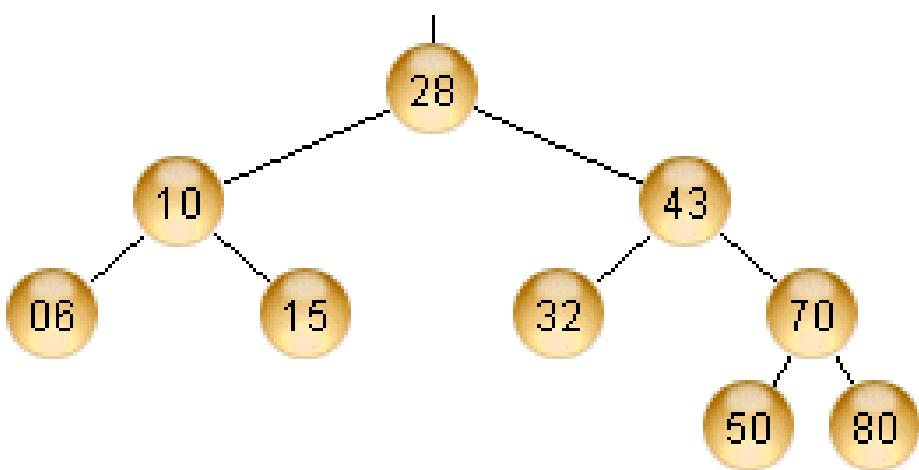
```
//Finally, we must remove the sucesorNode  
remove(sucesorNode.key, node.rightChild);  
return true;
```

Nota que el sucesor siempre será un nodo hoja o un nodo con un único hijo.

## TAD ABB: Ejercicios. Borrado

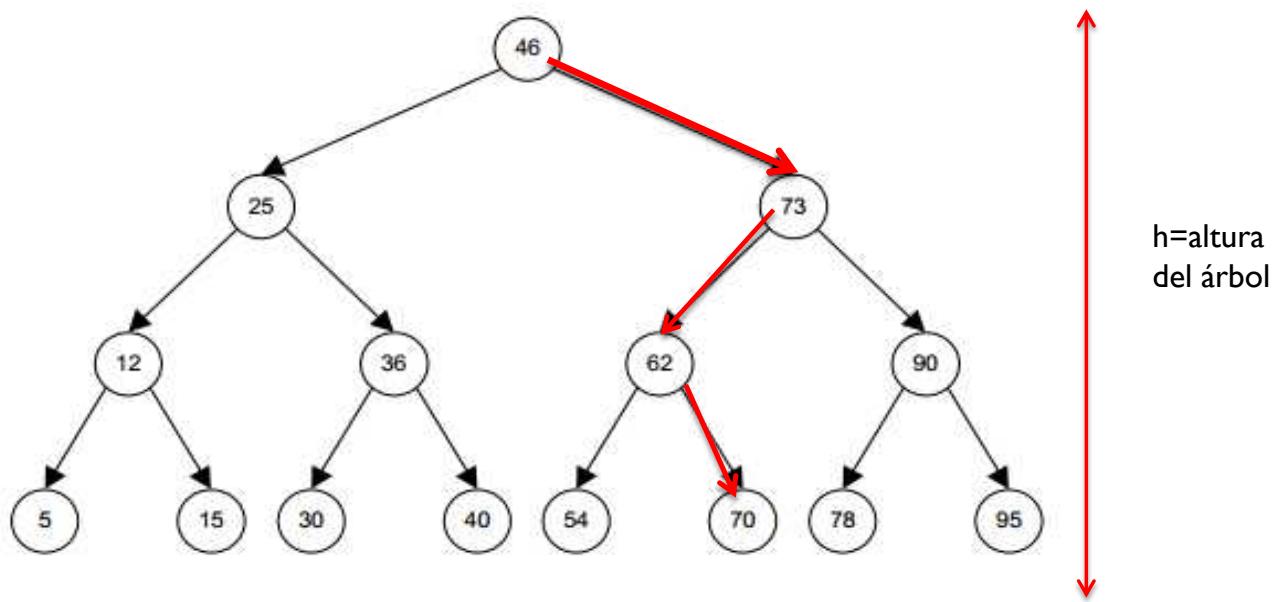
---

- ▶ Elimina la secuencia 70, 15, 28, 43, 32 del siguiente árbol



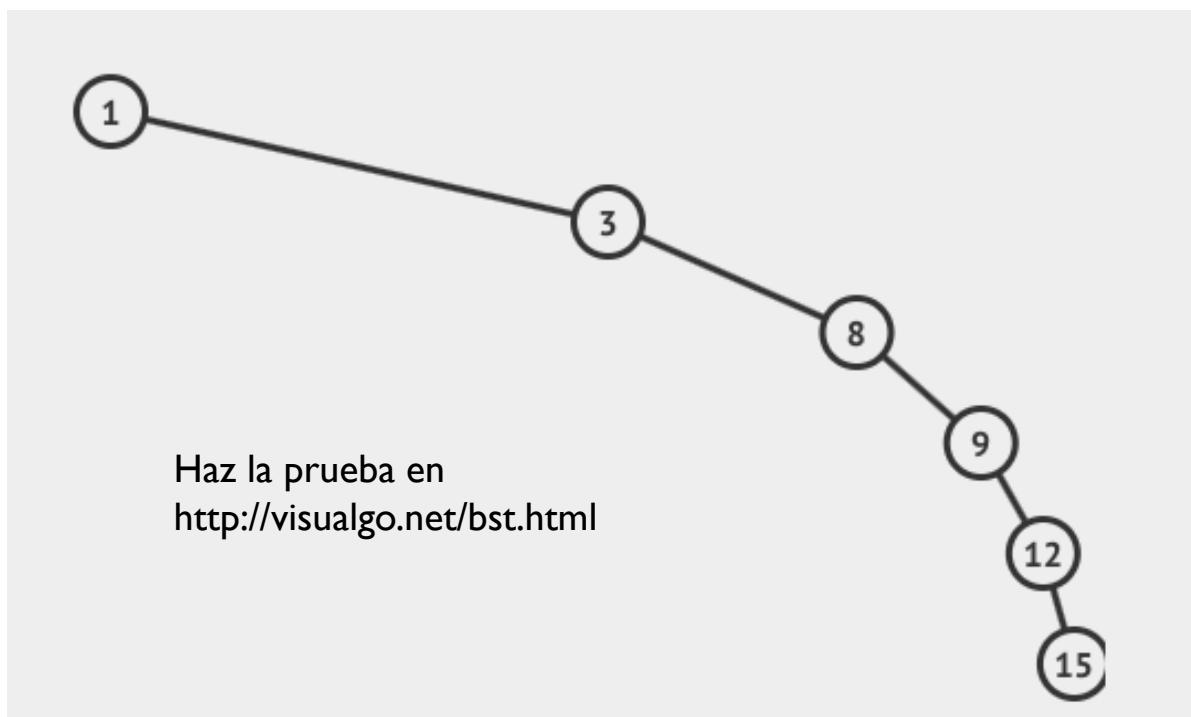
# Árboles Binarios de Búsqueda (ABB)

- ▶ La complejidad de las tres operaciones (búsqueda, inserción y borrado) es  $O(h)$  donde  $h$  es la altura del árbol. En el peor de los casos, se realizan  $h$  comparación, siendo  $h$  la altura del árbol



## TAD ABB

- ▶ La complejidad aumentará cuando  $h \rightarrow n$  (árbol degenerado).  
Complejidad  $O(n)$ . El siguiente árbol es el resultado de insertar la secuencia: 1, 3, 8, 9, 12, 15



## Tema 5.3. Equilibrado de árboles

Estructura de Datos y Algoritmos (EDA)

# Índice

---

- ▶ **5.3 Equilibrado de ABB**
  - ▶ **Definición de Árbol Perfectamente Equilibrado**
  - ▶ Definición de Árbol Equilibrado en Altura

## ABB perfectamente equilibrados

---

- ▶ Resumen de características de ABB
  - ▶  $h = \text{altura del árbol}, n = \text{número de nodos}$
  - ▶ Ventaja:
    - ▶ Inserción, borrado y búsqueda  $\sim O(h)$
  - ▶ Desventaja:
    - ▶ Se pierde eficiencia cuando  $n \approx h$ , y su complejidad se iguala a la de las listas



## ABB perfectamente equilibrados

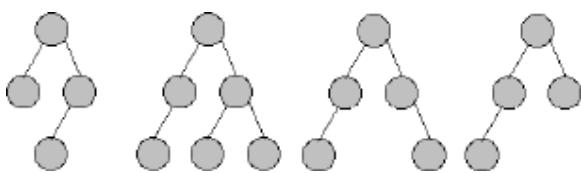
---

- ▶ Estrategia → Despues de las operaciones de inserción o borrado, debemos mantener el árbol equilibrado.
  - ▶ Equilibrio perfecto o en tamaño (árboles perfectamente equilibrados)
  - ▶ Equilibrio en altura (árboles AVL, árboles ideado por los matemáticos Adelson-Velskii y Landis)

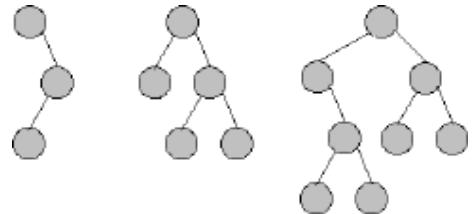


## ABB perfectamente equilibrados (en tamaño)

- ▶ Factor de equilibrio de un nodo (fe) : diferencia entre el tamaño del subárbol derecho y el del izquierdo (o viceversa)
- ▶ Para TODOS los nodos, el número de nodos del subárbol izquierdo y el número de nodos del subárbol derecho difieren como máximo en 1 unidad
- ▶ Coste alto de mantener un ABB perfectamente equilibrado,  $O(n)$



con equilibrio perfecto



sin equilibrio perfecto



## ABB perfectamente equilibrados: Algoritmo Reequilibrado

---

- ▶ Idea: desplazar la mitad de los nodos que sobran de un lado al otro del ABB
- ▶ Importante: Reequilibrado se hace desde la raíz hacia abajo (es decir, de forma **descendiente**).
- ▶ Será necesario modificar algoritmos de inserción/borrado
  - ▶ haciendo un re-equilibrado tras insertar/borrar, o bien
  - ▶ equilibrando en algún momento



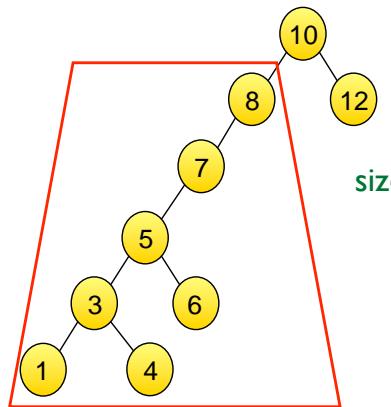
## ABB perfectamente equilibrados: Algoritmo Reequilibrado

---

- ▶ El siguiente algoritmo para desplazar nodos, asegura el nuevo árbol mantendrá la condición de ABB. Pasos:
  - ▶ Desplazar a derechas:
    1. Introducir la raíz del subárbol no equilibrado en el subárbol derecho
    2. Colocar como raíz del subárbol no equilibrado el mayor del subárbol izquierdo
    3. Repetir 1 y 2 tantas veces como número de nodos a desplazar. (Es decir, repeticiones= (fe nodo a equilibrar/2)).
  - ▶ Desplazar a izquierdas: (simétrico)



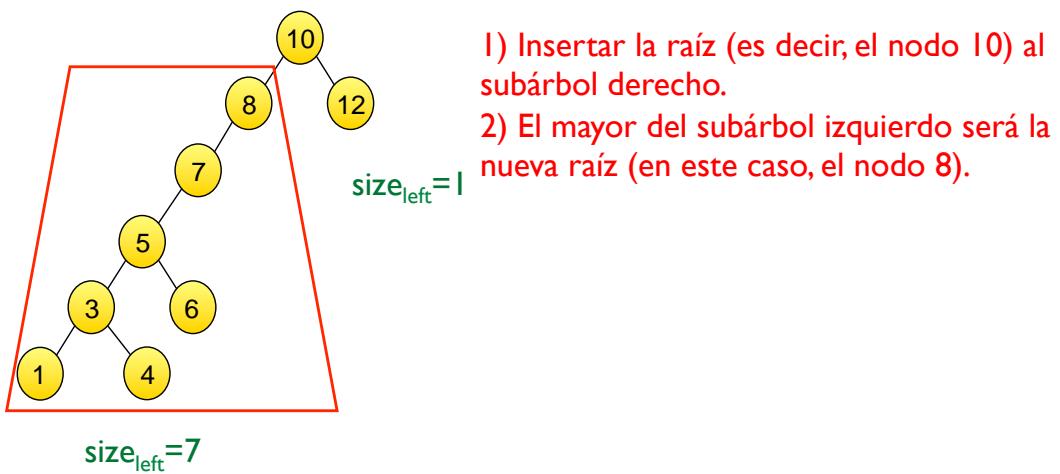
## Ejemplo 1 de equilibrado perfecto (1 de 4)



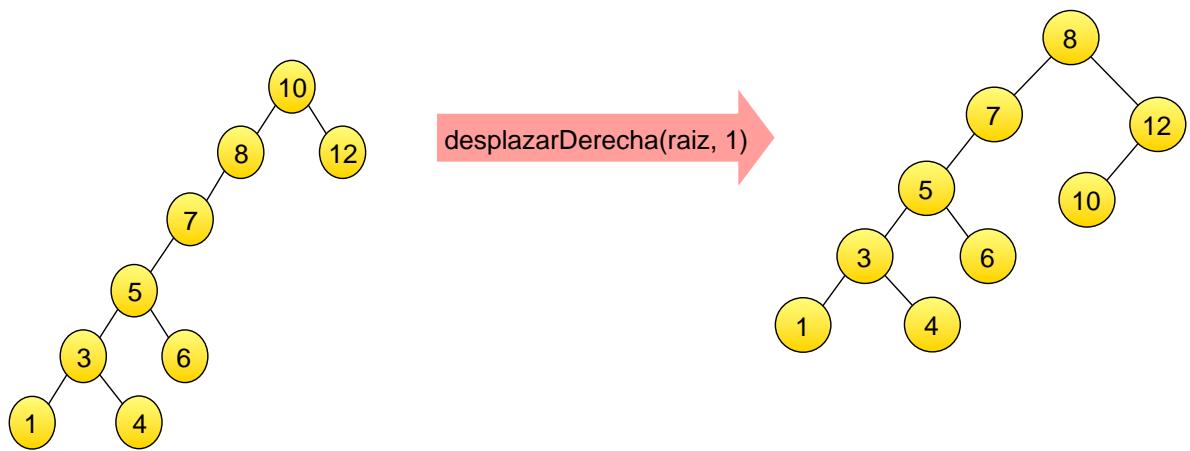
$fe(10)=7-1=6$ . Es decir, hay 6 nodos de diferencia  $\rightarrow$  desplazar 6/2 veces a la derecha



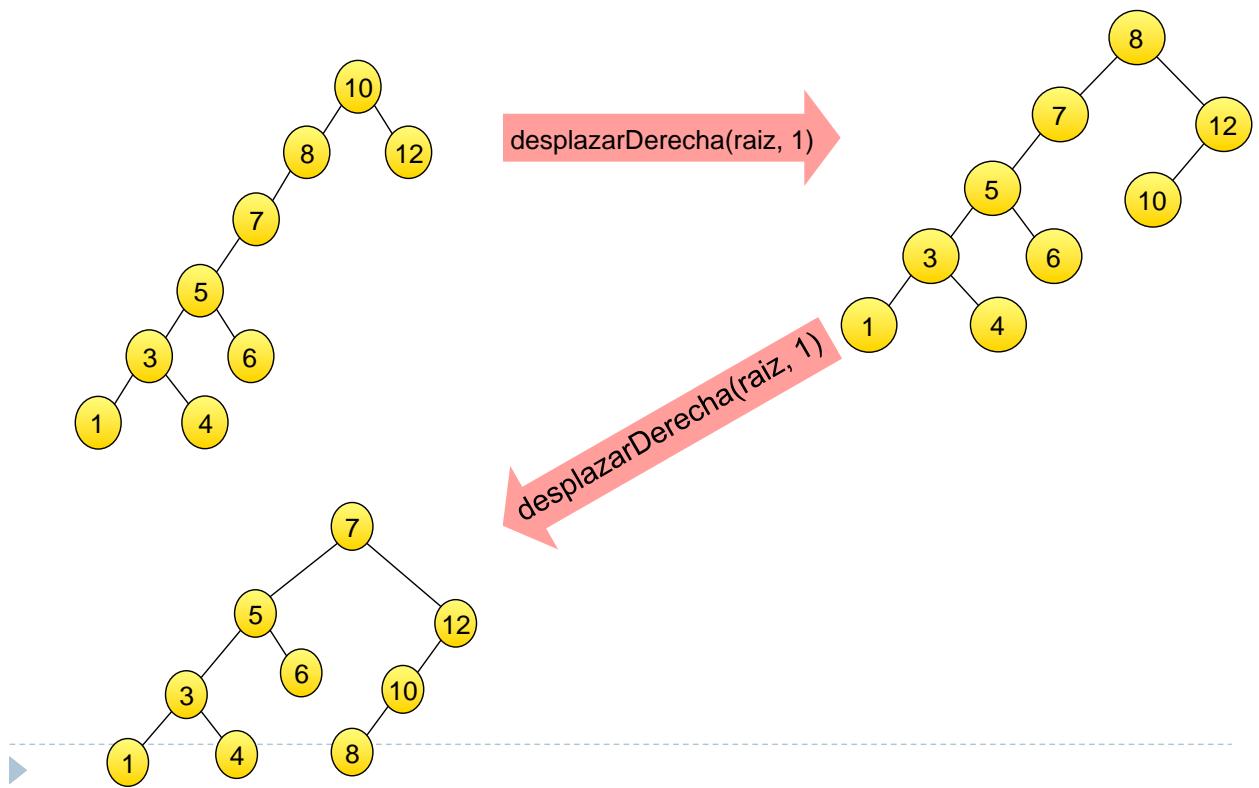
## Ejemplo 1 de equilibrado perfecto (1 de 4)



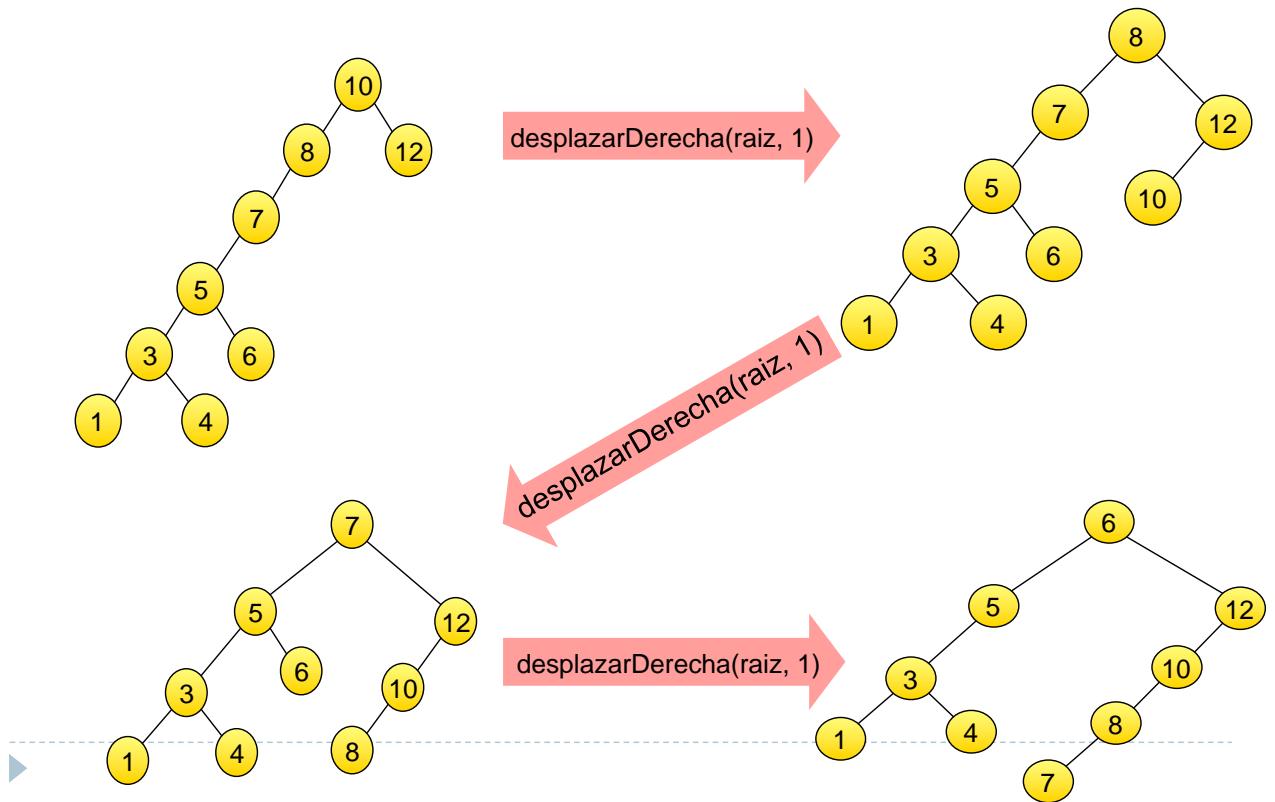
## Ejemplo 1 de equilibrado perfecto (1 de 4)



## Ejemplo 1 de equilibrado perfecto (1 de 4)

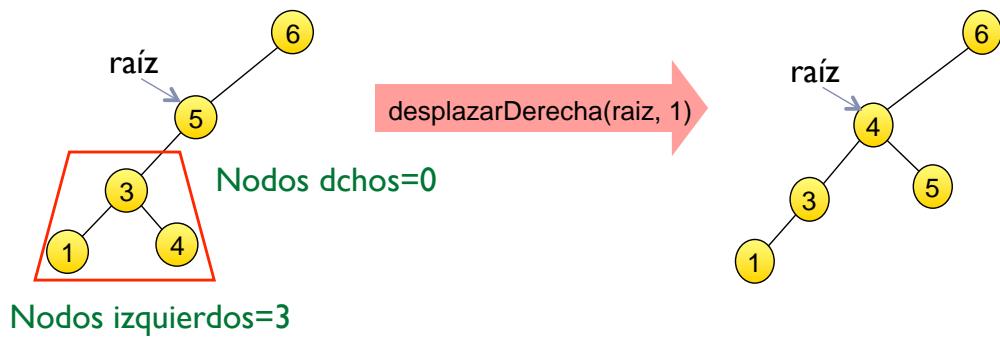


## Ejemplo 1 de equilibrado perfecto (1 de 4)



## Ejemplo 1 de equilibrado perfecto (2 de 4)

Continuamos: Aplicar equilibrio perfecto al subárbol izquierdo

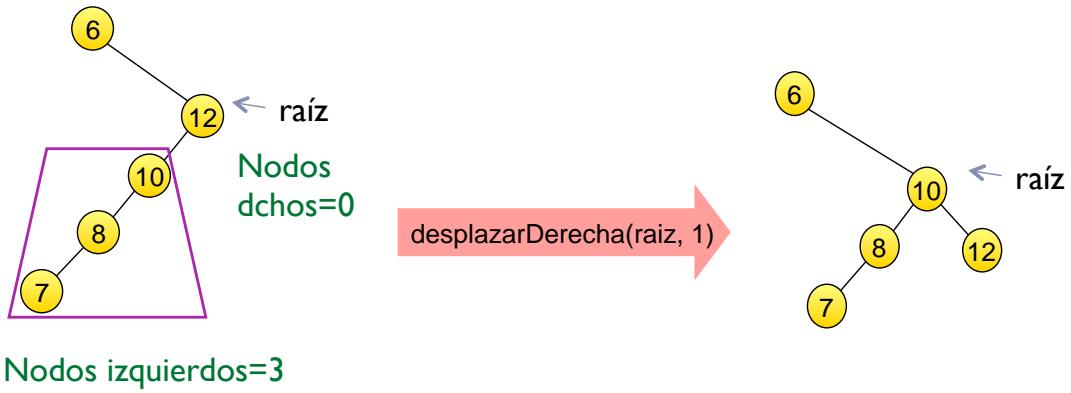


3 nodos de diferencia → desplazar 3/2 veces a la derecha



## Ejemplo 1 de equilibrado perfecto (3 de 4)

Continuamos: Aplicar equilibrio perfecto al subárbol derecho

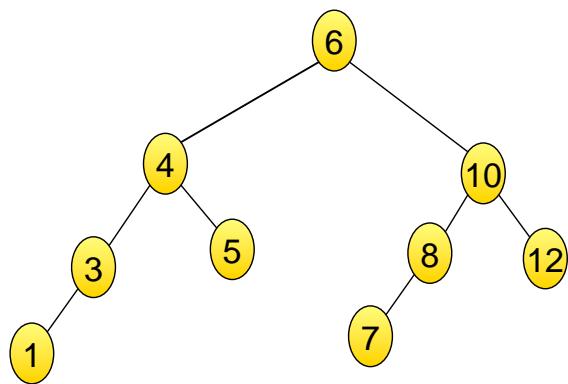


3 nodos de diferencia → desplazar 3/2 veces a la derecha

## Ejemplo 1 de equilibrado perfecto (4/4)

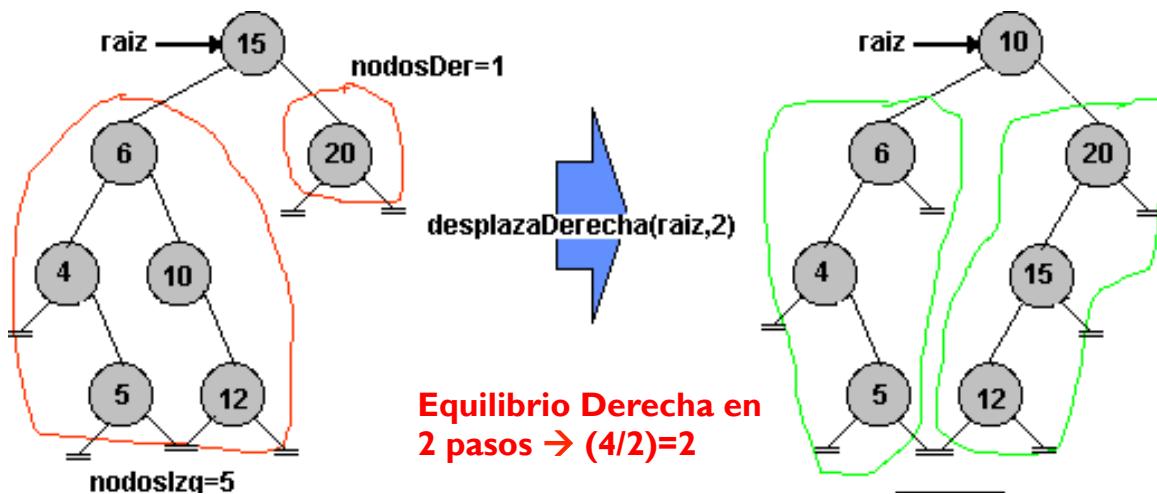
---

RESULTADO final



## Ejemplo 2 de equilibrado perfecto (1 de 4)

Primero se equilibra la raíz, pasando dos veces un nodo hacia la derecha ( $f_e = -4$ )

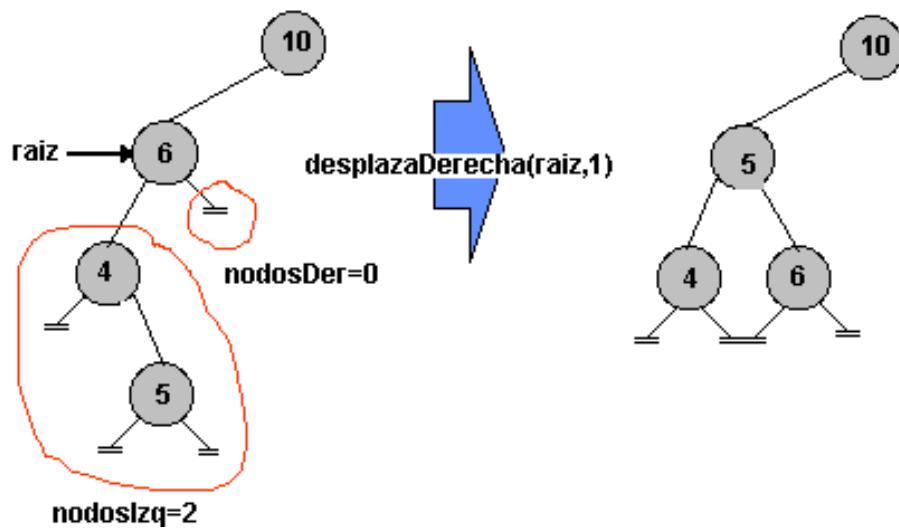


Continuamos: llamamos a equilibrar el subárbol izquierdo y luego el subárbol derecho



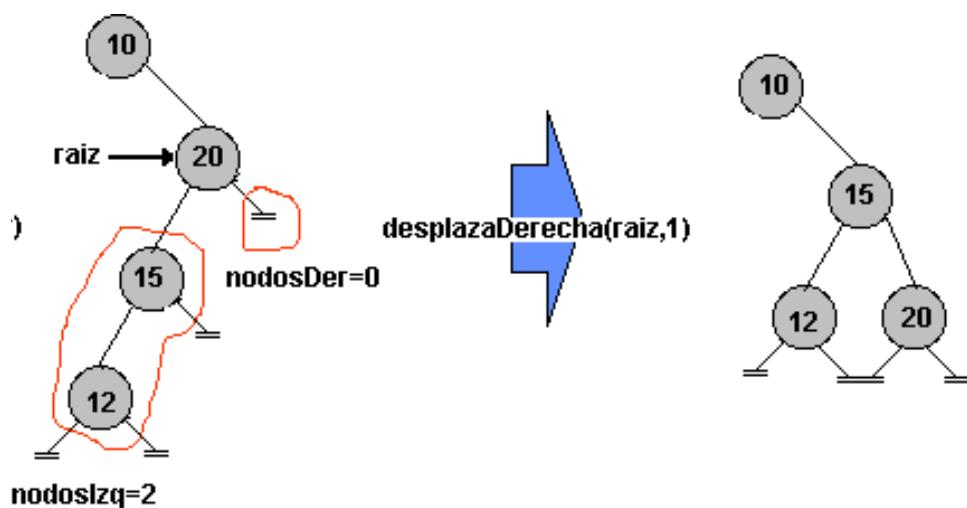
## Ejemplo 2 de equilibrado perfecto (2 de 4)

... equilibrio del subárbol izquierdo ( $f_e = -2$ )



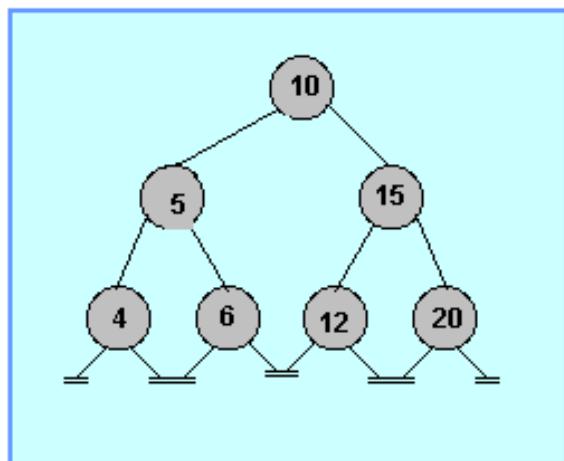
## Ejemplo 2 de equilibrado perfecto (3 de 4)

... equilibrio del subárbol derecho ( $f_e = -2$ )



## Ejemplo 2 de equilibrado perfecto (4 de 4)

RESULTADO: árbol perfectamente equilibrado



# Índice

---

- ▶ **4.3 Árboles Equilibrados**

- ▶ **Árboles Binarios Equilibrados**

- ▶ Definición de Árbol Perfectamente Equilibrado

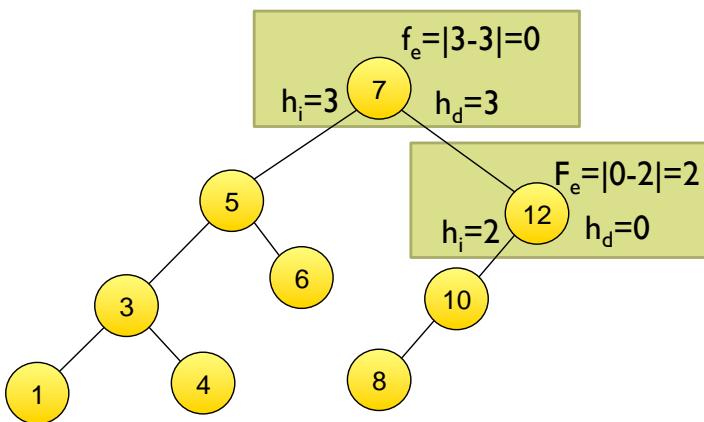
- ▶ **Árboles AVL: Definición de Árbol Equilibrado en Altura.**

- AVL: Árbol binario ideado por los matemáticos rusos Adelson-Velskii y Landis.

## ABB: Equilibrados en altura (árboles AVL)

- ▶ Factor de equilibrio en altura de un nodo (fe) :  
Diferencia entre la altura (longitud del camino máximo a una hoja) por el lado derecho y por el lado izquierdo (o viceversa)

$$fe = |hd - hi|$$

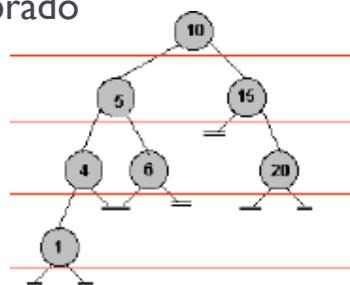


## ABB: Equilibrados en altura (árboles AVL)

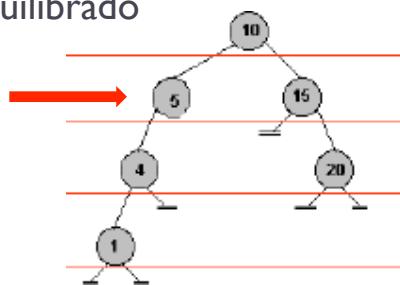
### ▶ ABB equilibrado en altura

- Para cada uno de sus nodos, las alturas de sus subárboles izquierdo y derecho difieren como máximo en 1 unidad

### ▶ ABB equilibrado



### ABB no equilibrado



- Esto es, el factor de equilibrio (en altura) de todos sus nodos es menor o igual que 1.

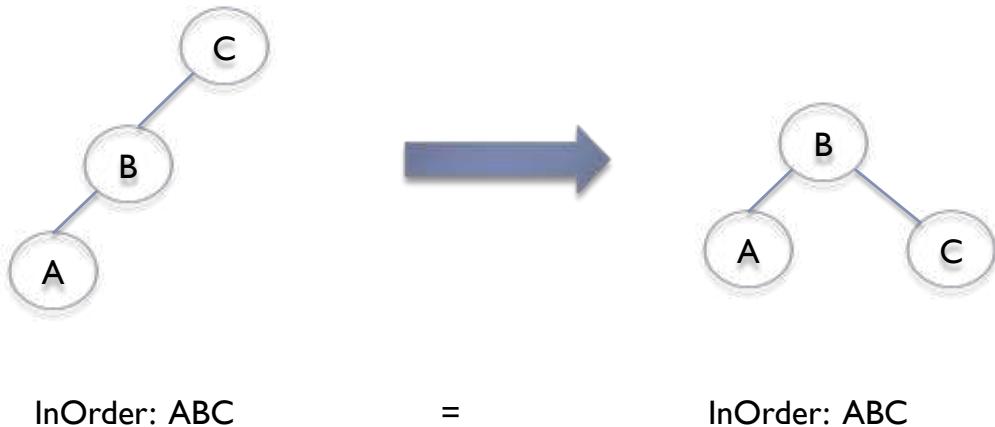
## ABB: Equilibrados en altura (árboles AVL)

---

- ▶ Idea: desplazar nodos de la rama más larga a la rama más corta.
- ▶ Importante: Equilibrado se hace en orden ascendente, es decir, siempre desde abajo, sólo en el camino desde el nodo insertado o borrado hacia la raíz.
- ▶ Nota que el árbol resultante debe seguir siendo ABB (y por tanto, tener el mismo recorrido in order).
- ▶ Rotaciones:
  - ▶ Rotación simple a la derecha
  - ▶ Rotación simple a la izquierda
  - ▶ Rotación doble (izquierda-derecha)
  - ▶ Rotación doble (derecha-izquierda)

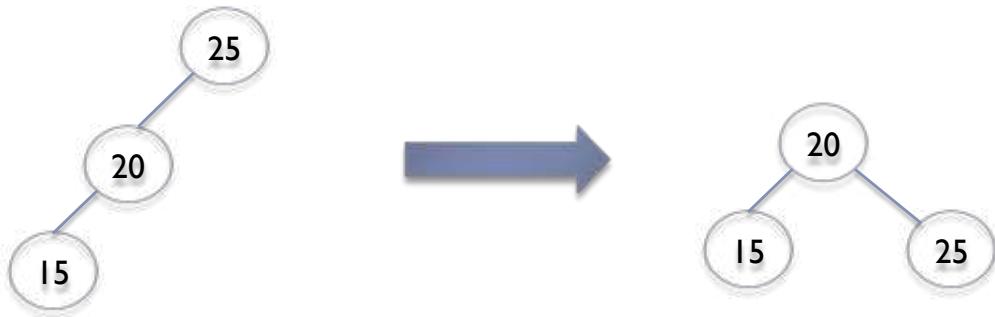
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación simple a la derecha:



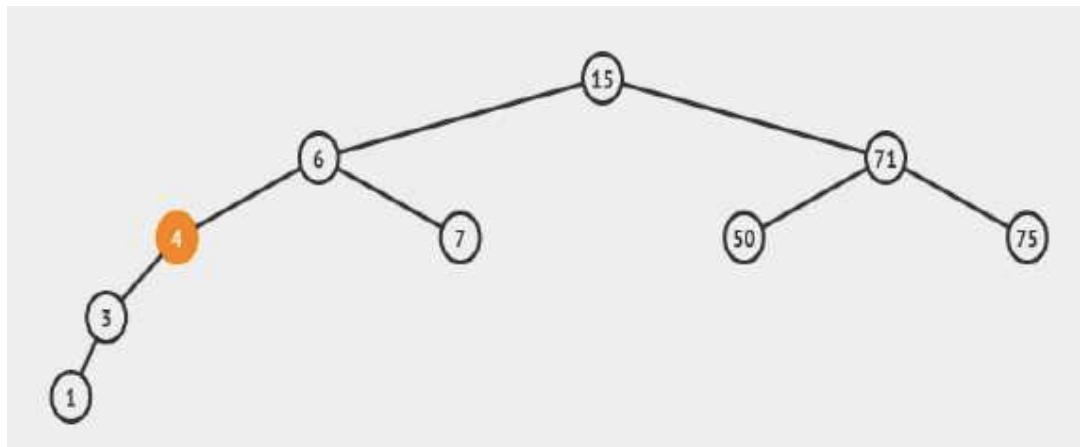
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo rotación simple a la derecha:



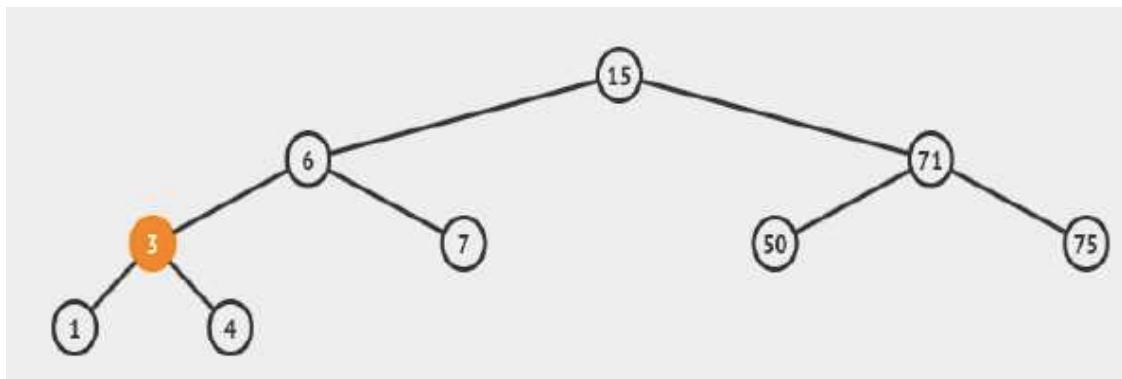
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo rotación simple a la derecha:



## ABB: Equilibrados en altura (árboles AVL)

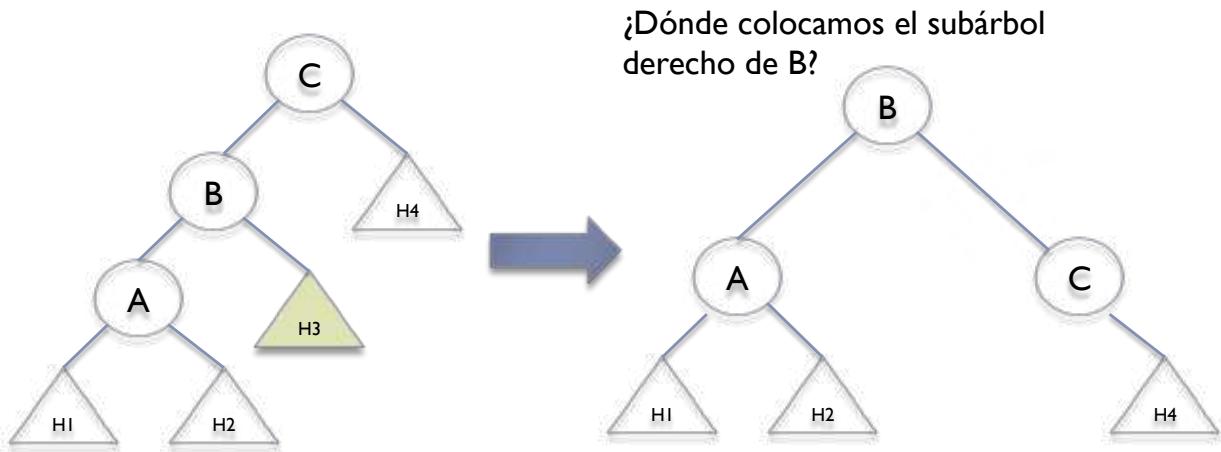
- ▶ Ejemplo rotación simple a la derecha:



<http://visualgo.net/bst.html>

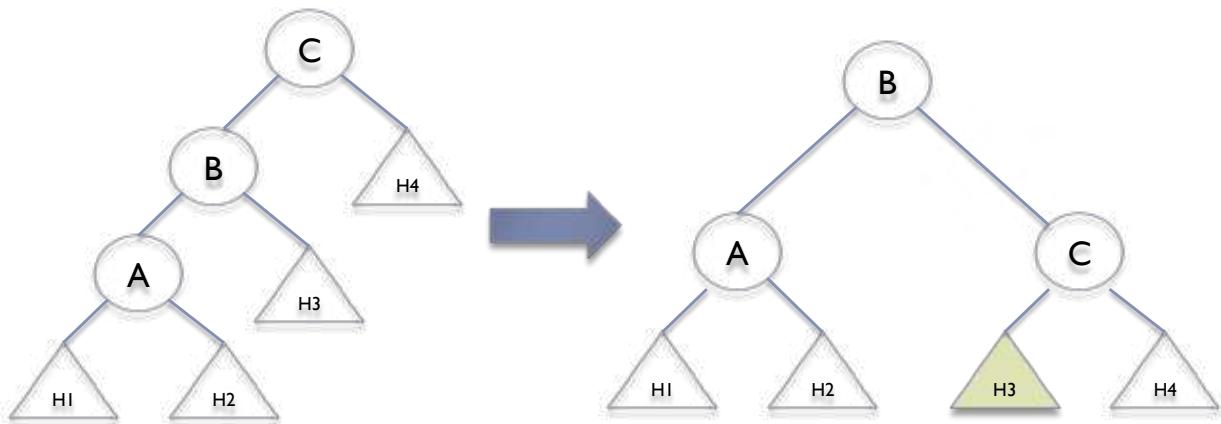
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación simple a la derecha:



## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación simple a la derecha:



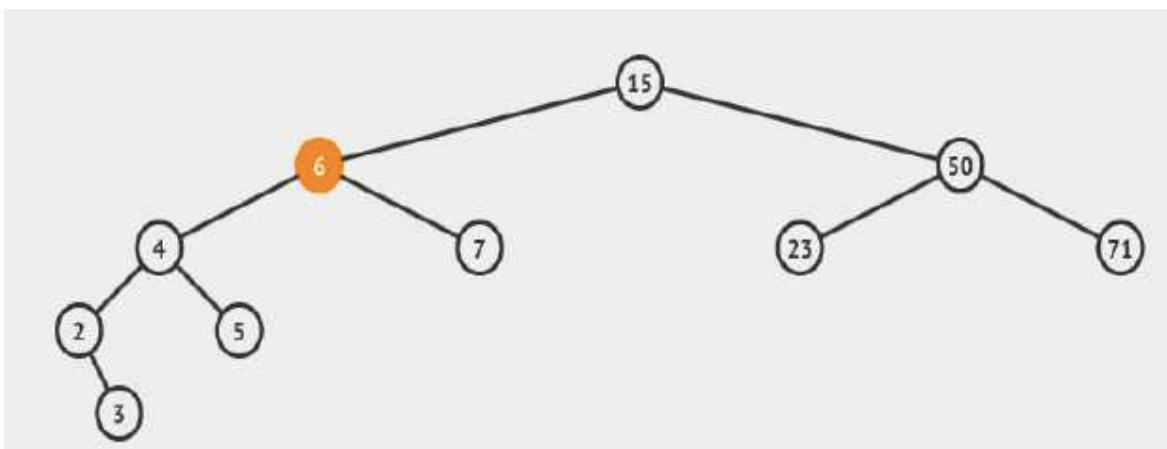
InOrder:  $H_1AH_2BH_3CH_4$

=

InOrder:  $H_1AH_2BH_3CH_4$

## ABB: Equilibrados en altura (árboles AVL)

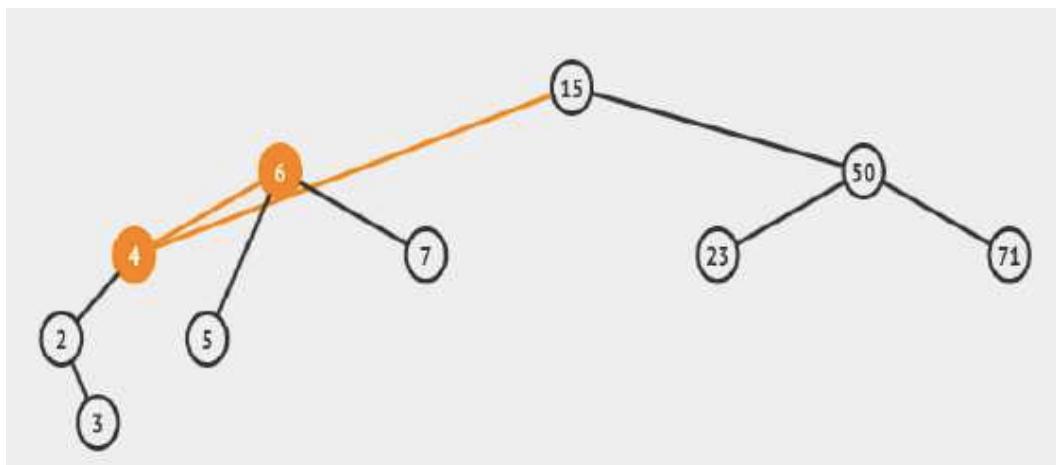
- ▶ Ejemplo (con subárboles) rotación simple a la derecha:



- El primer nodo desequilibrado es el 6.
- Podemos aplicar una rotación simple a la derecha: **mover 4 como raíz del subárbol, y 6 como su hijo derecho.**
- Nota que el nodo 4, tiene un subárbol derecho, ¿qué hacemos con el nodo 5?

## ABB: Equilibrados en altura (árboles AVL)

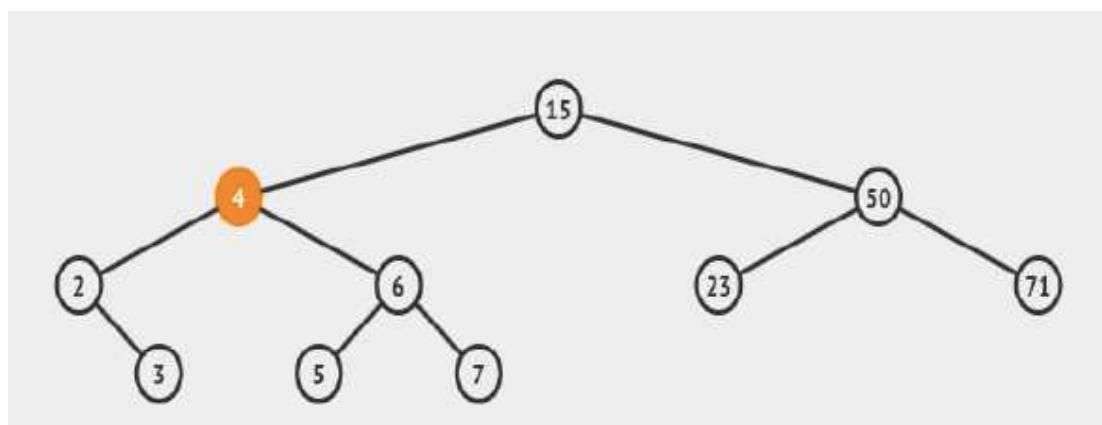
► Ejemplo (con subárboles) rotación simple a la derecha:



- El nodo 6 se rota como hijo derecho de 4, y éste pasa a ser la nueva raíz del subárbol.
- **El antiguo subárbol derecho de 4, es decir el 5, tiene que pasar ahora a ser subárbol izquierdo de 6.**

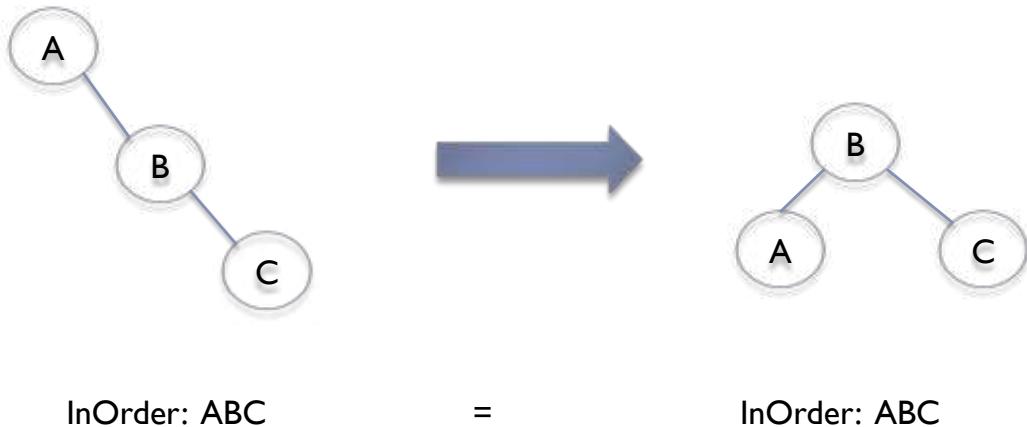
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo (con subárboles) rotación simple a la derecha:



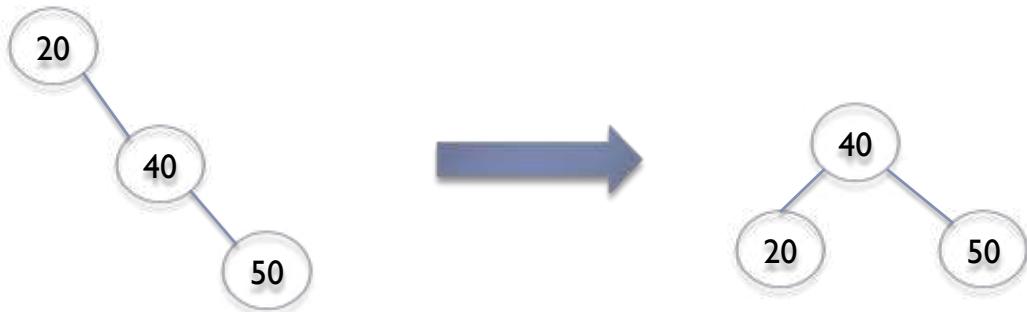
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación simple a la izquierda:



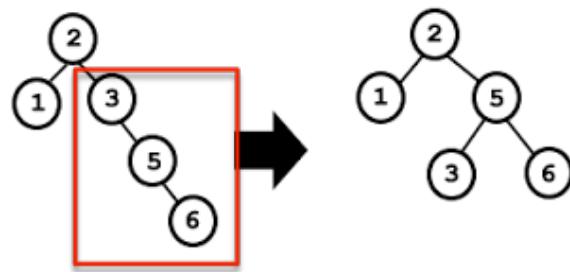
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo Rotación simple a la izquierda:



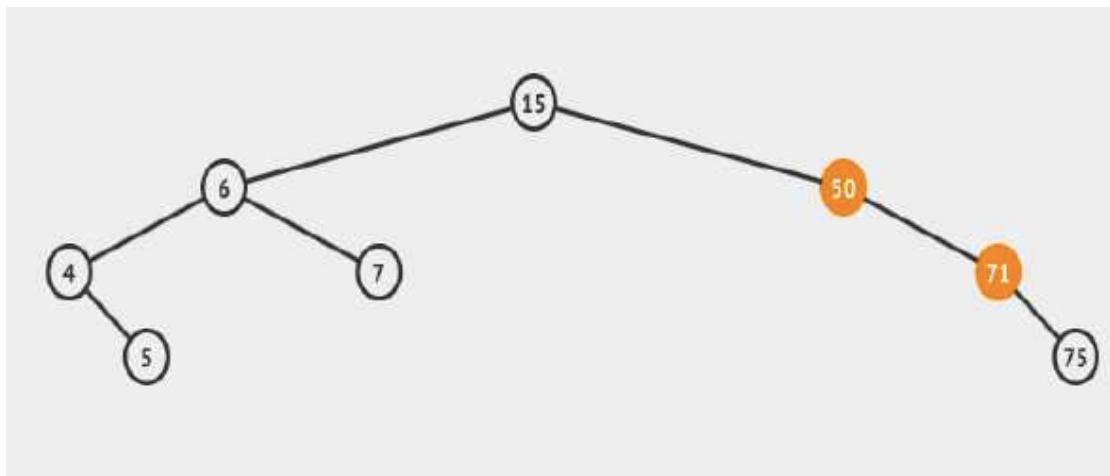
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo rotación simple a la izquierda:



## ABB: Equilibrados en altura (árboles AVL)

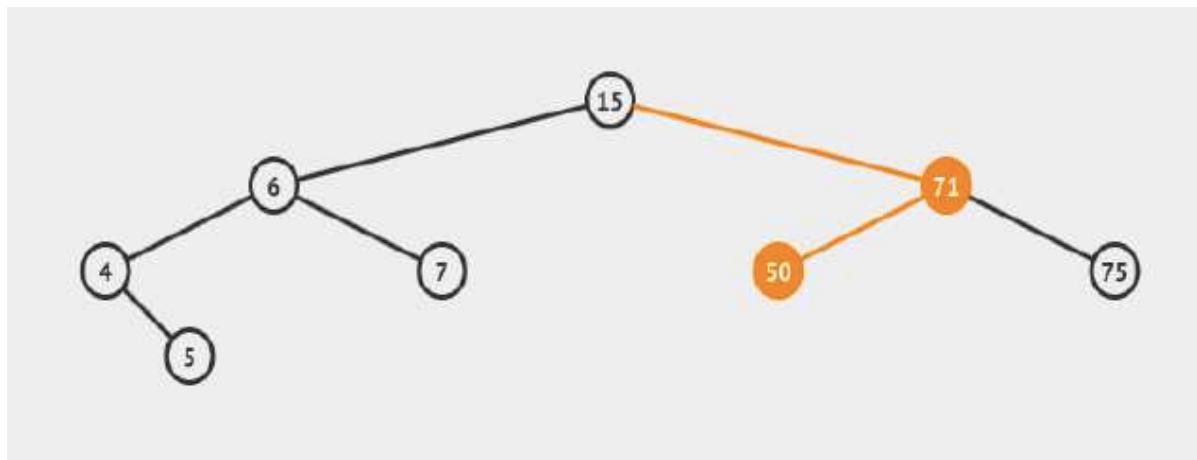
- ▶ Ejemplo rotación simple a la izquierda:



<http://visualgo.net/bst.html>

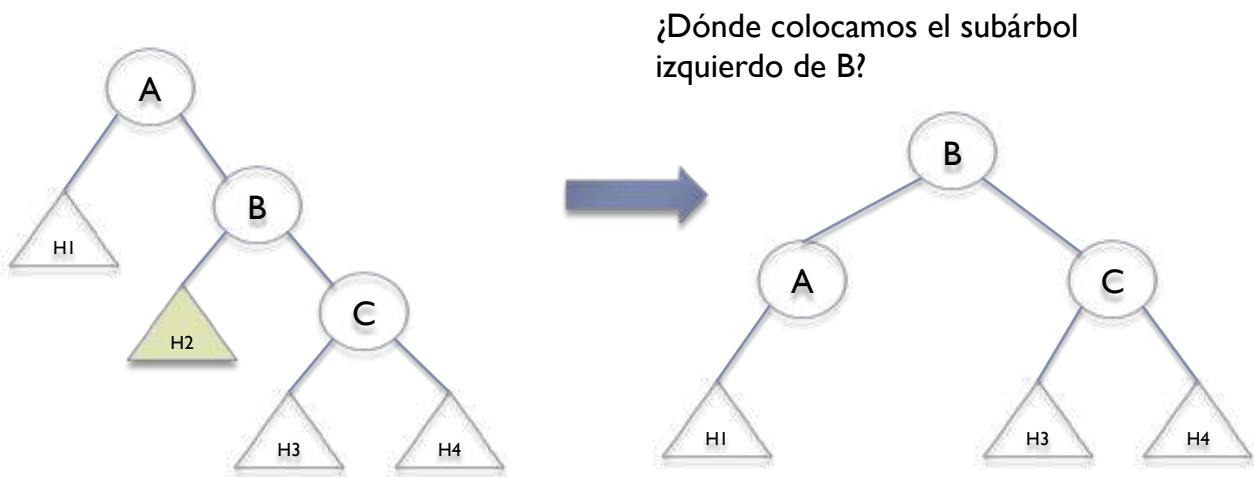
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo rotación simple a la izquierda:



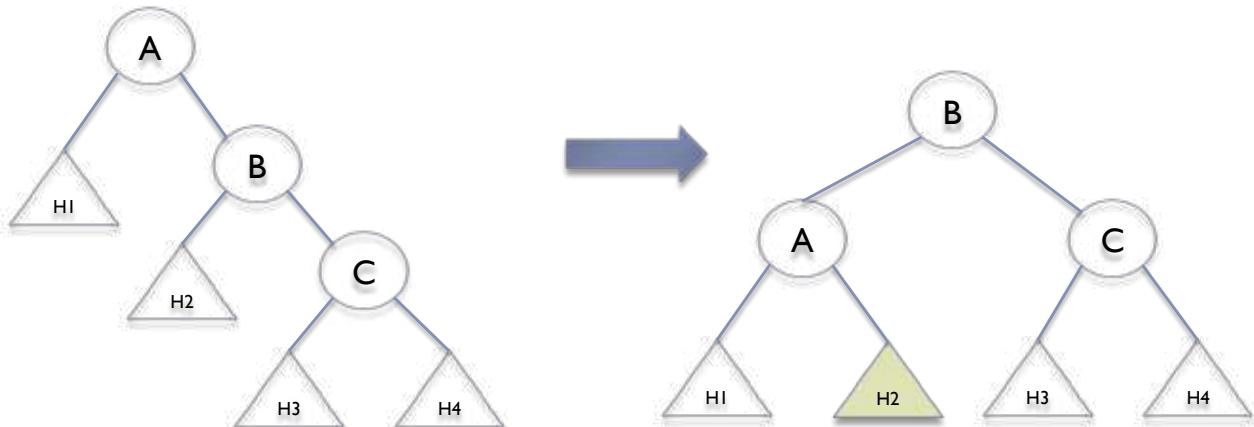
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación simple a la izquierda:



## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación simple a la izquierda:



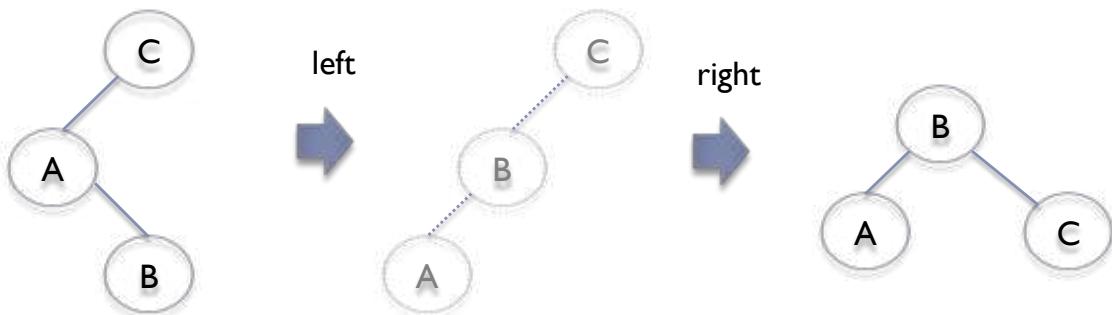
InOrder:  $H_1AH_2BH_3CH_4$

=

InOrder:  $H_1AH_2BH_3CH_4$

## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación Doble (left-right) :



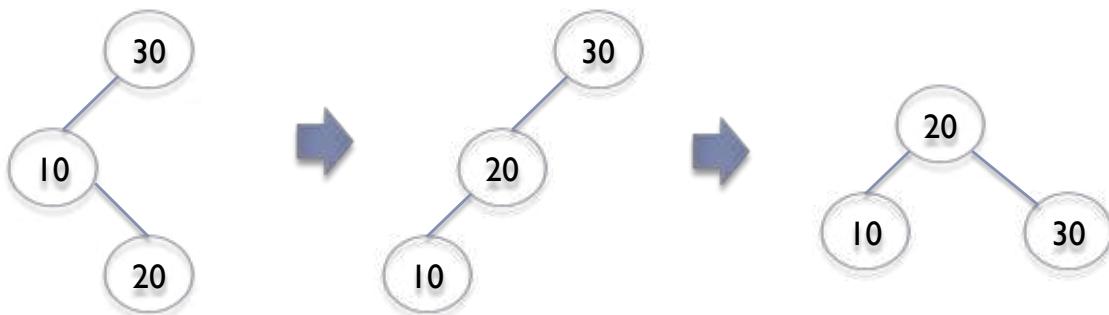
InOrder: ABC

=

InOrder: ABC

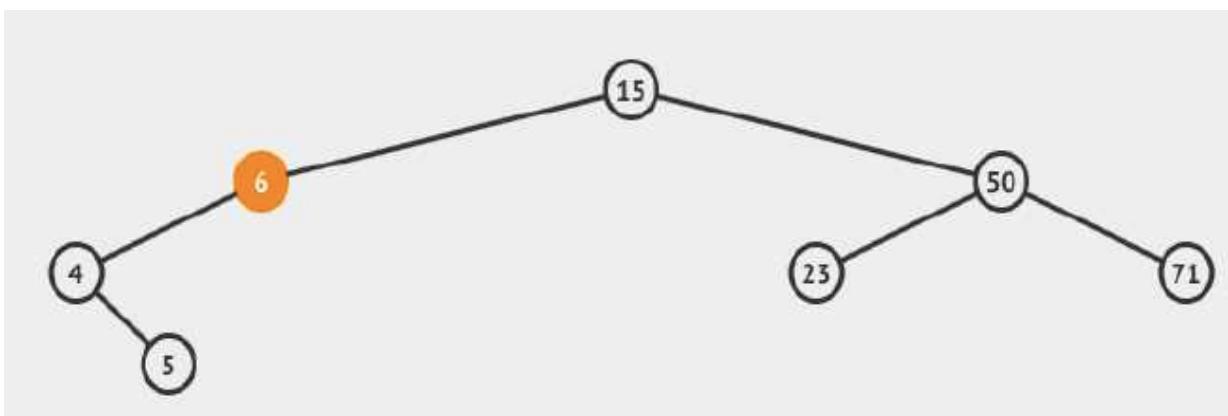
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo Rotación Doble (left-right):



## ABB: Equilibrados en altura (árboles AVL)

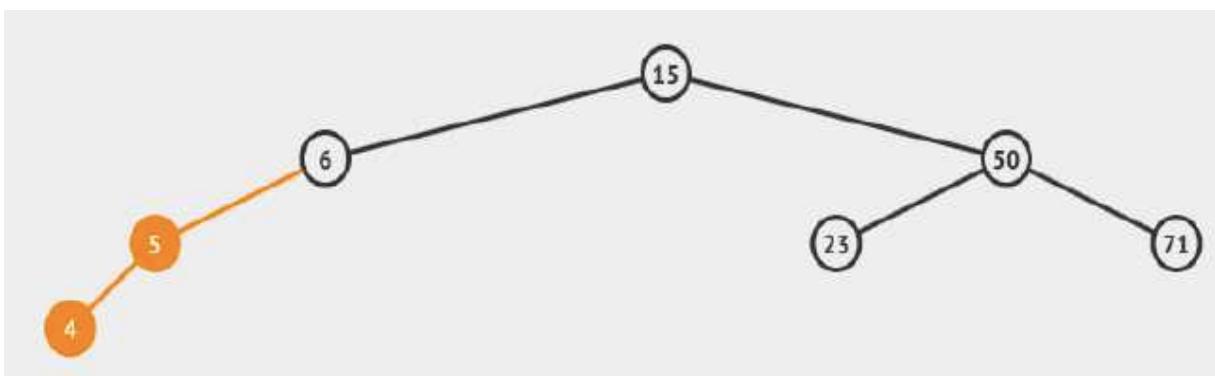
### ► Ejemplo Rotación doble (left-right):



- El nodo desequilibrado es el 6. Para equilibrar zig-zag izquierdo-derecho, tenemos que hacer dos rotaciones:
- Primera rotación a la izquierda: subimos el nodo 5 como hijo izquierdo de 6, y 4 pasa a ser hijo izquierdo de 5.
- Segunda rotación a la derecha: giramos el nodo 6 a la derecha y el nodo 5 pasa a ser la nueva raíz del subárbol.

## ABB: Equilibrados en altura (árboles AVL)

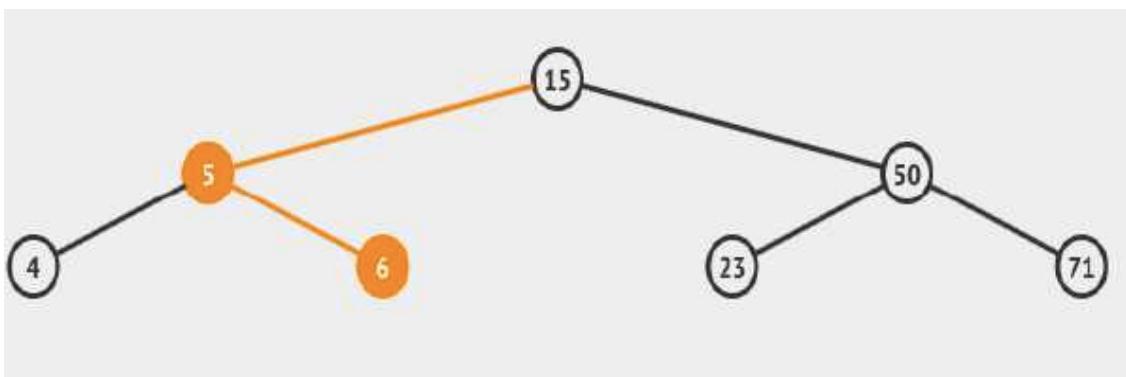
### ► Ejemplo Rotación doble (left-right):



- El nodo desequilibrado es el 6. Para equilibrar zig-zag izquierdo-derecho, tenemos que hacer dos rotaciones:
- **Primera rotación a la izquierda: subimos el nodo 5 como hijo izquierdo de 6, y 4 pasa a ser hijo izquierdo de 5.**
- Segunda rotación a la derecha: giramos el nodo 6 a la derecha y el nodo 5 pasa a ser la nueva raíz del subárbol.

## ABB: Equilibrados en altura (árboles AVL)

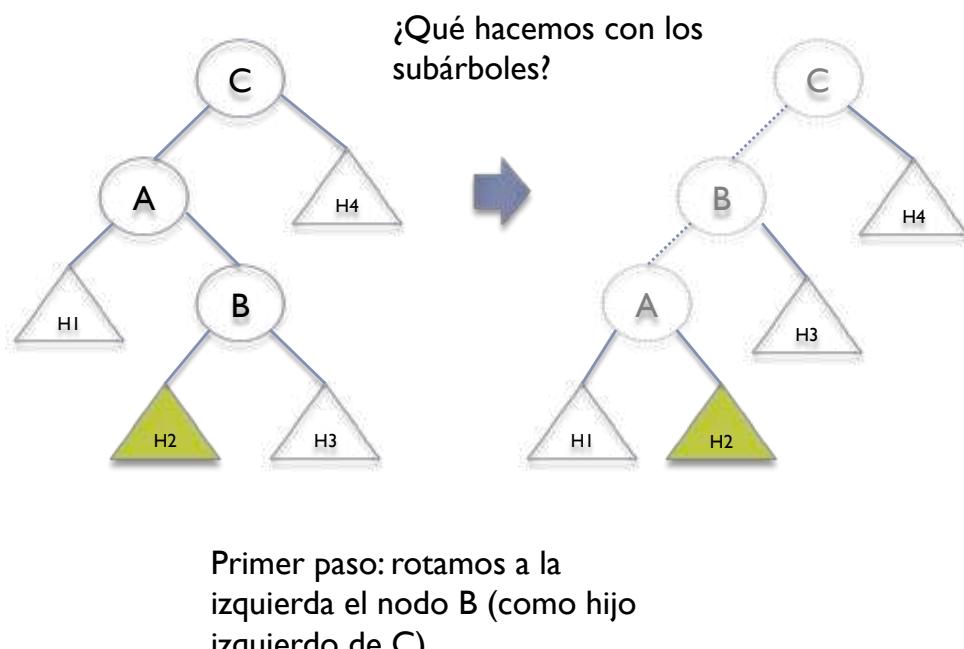
### ► Ejemplo Rotación doble (left-right):



- El nodo desequilibrado es el 6. Para equilibrar zig-zag izquierdo-derecho, tenemos que hacer dos rotaciones:
- Primera rotación a la izquierda: subimos el nodo 5 como hijo izquierdo de 6, y 4 pasa a ser hijo izquierdo de 5.
- **Segunda rotación a la derecha: giramos el nodo 6 a la derecha y el nodo 5 pasa a ser la nueva raíz del subárbol.**

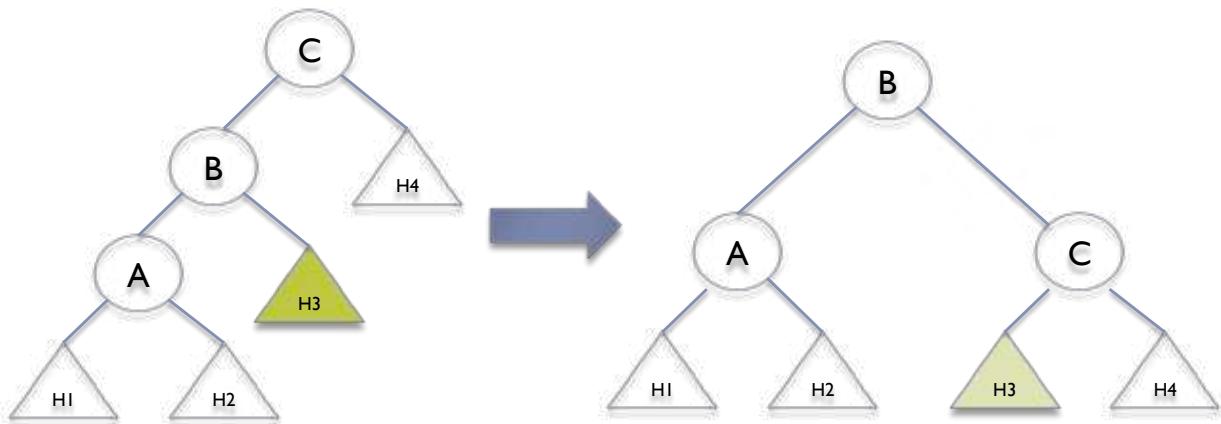
## ABB: Equilibrados en altura (árboles AVL)

### ► Rotación Doble (left-right) :



## ABB: Equilibrados en altura (árboles AVL)

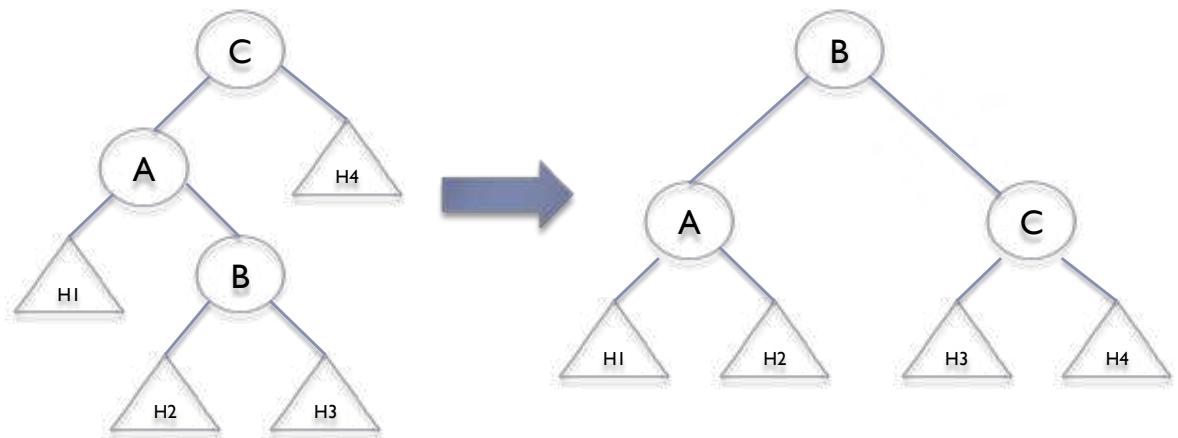
- ▶ Rotación doble (left-right) :



Segunda rotación: movemos el  
nodo C como hijo derecho de B

## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación doble (left-right):



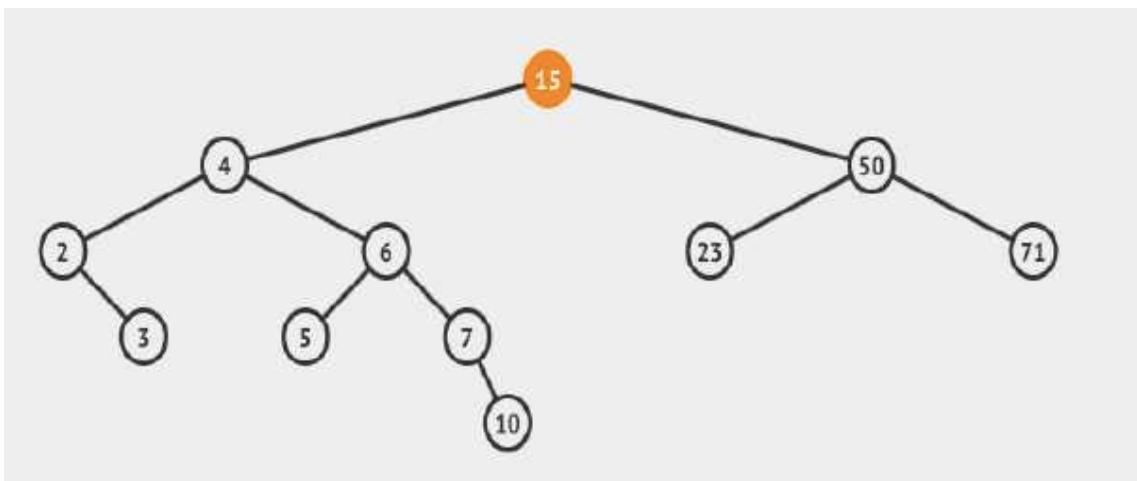
InOrder: H<sub>1</sub>AH<sub>2</sub>BH<sub>3</sub>CH<sub>4</sub>

=

InOrder: H<sub>1</sub>AH<sub>2</sub>BH<sub>3</sub>CH<sub>4</sub>

## ABB: Equilibrados en altura (árboles AVL)

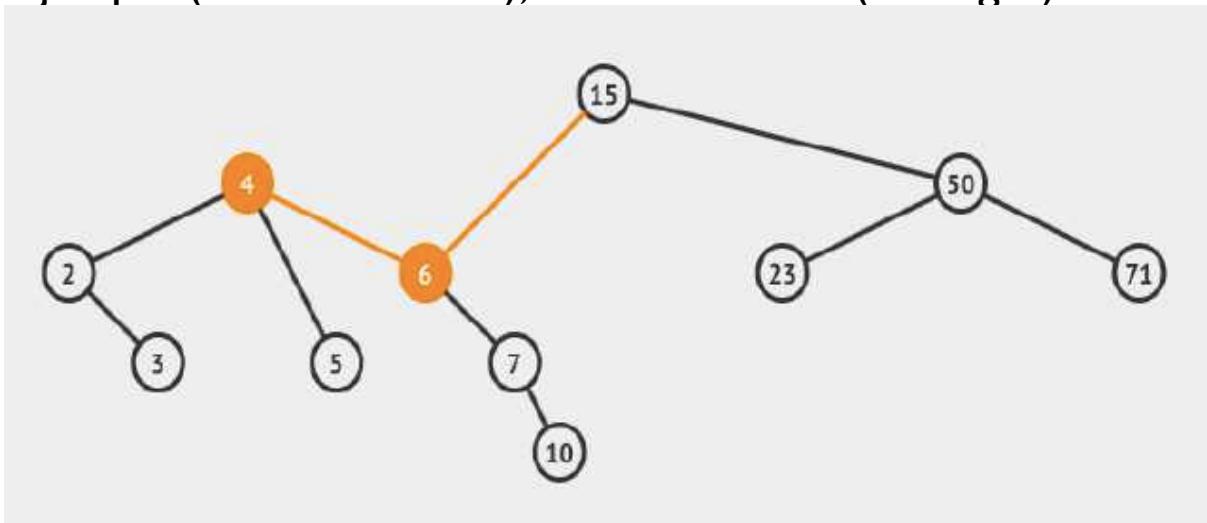
- ▶ Ejemplo (con subárboles) Rotación doble (left-right):



- El nodo 15 está desequilibrado ( $fe=2$ ). Podemos aplicar una rotación left-right.
- La primera rotación es mover el 6 como hijo izquierdo de 15.
- La segunda rotación será rotar el 15 como hijo derecho de 6 (y que éste sea la nueva raíz)

## ABB: Equilibrados en altura (árboles AVL)

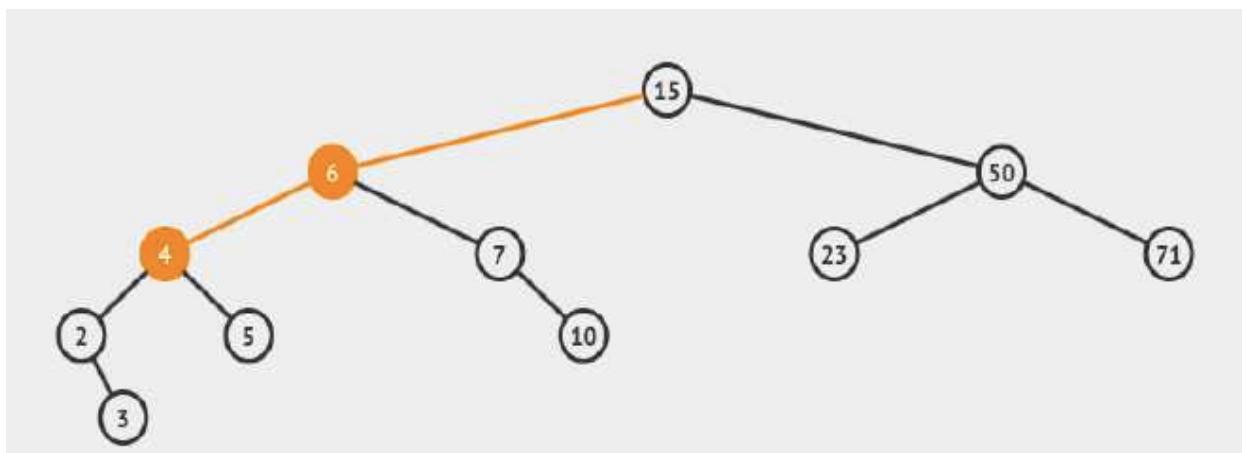
- ▶ Ejemplo (con subárboles), Rotación doble (left-right):



- El nodo 15 está desequilibrado ( $fe=2$ ). Podemos aplicar una rotación left-right.
- **La primera rotación es mover el 6 como hijo izquierdo de 15**
- La segunda rotación será rotar el 15 como hijo derecho de 6 (y que éste sea la nueva raíz)

## ABB: Equilibrados en altura (árboles AVL)

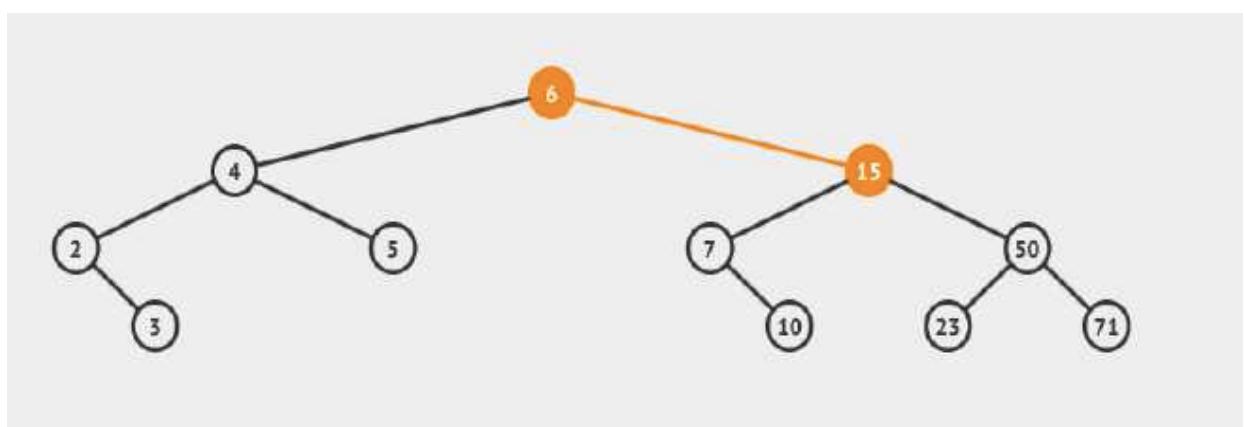
- ▶ Ejemplo (con subárboles) Rotación doble (left-right):



- El nodo 15 está desequilibrado ( $fe=2$ ). Podemos aplicar una rotación left-right.
- La primera rotación es mover el 6 como hijo izquierdo de 15
- La segunda rotación será rotar el 15 como hijo derecho de 6 (y que éste sea la nueva raíz)

## ABB: Equilibrados en altura (árboles AVL)

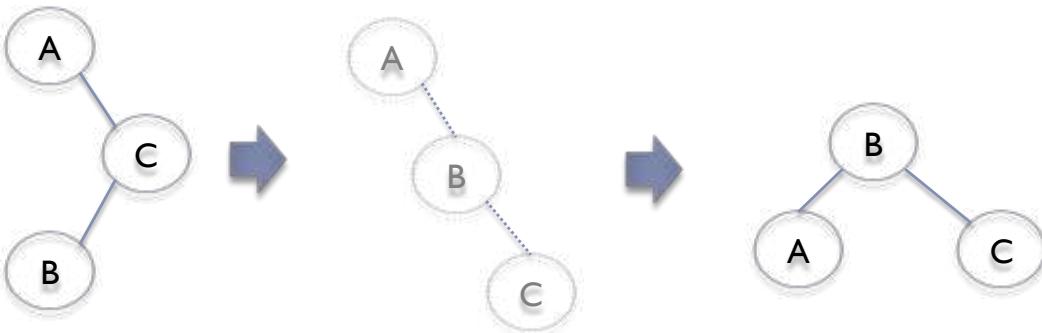
- ▶ Ejemplo (con subárboles) Rotación doble (left-right):



- El nodo 15 está desequilibrado ( $fe=2$ ). Podemos aplicar una rotación left-right.
- La primera rotación es mover el 6 como hijo izquierdo de 15
- **La segunda rotación será rotar el 15 como hijo derecho de 6 (y que éste sea la nueva raíz)**

## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación Doble (right-left):



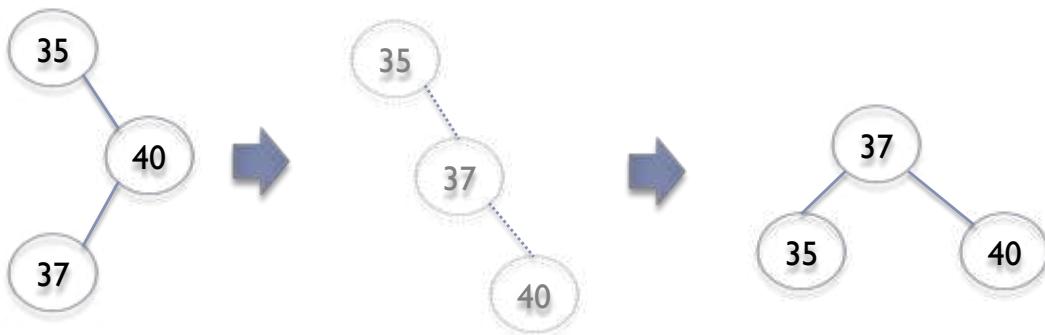
InOrder: ABC

=

InOrder: ABC

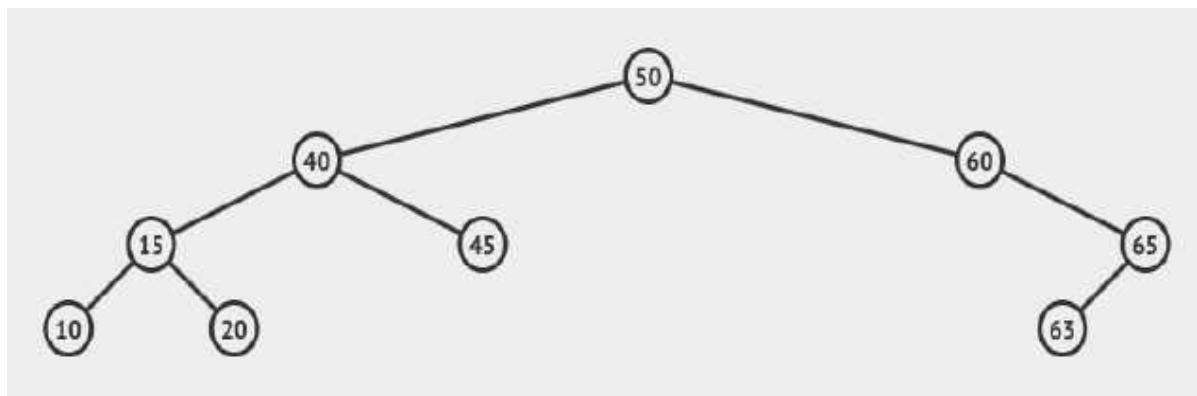
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo Rotación Doble (right-left):



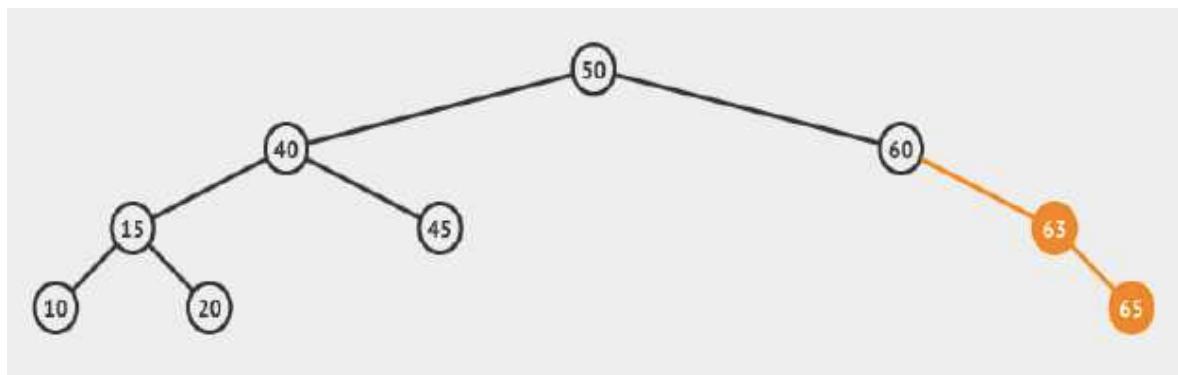
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo Rotación doble (right-left):



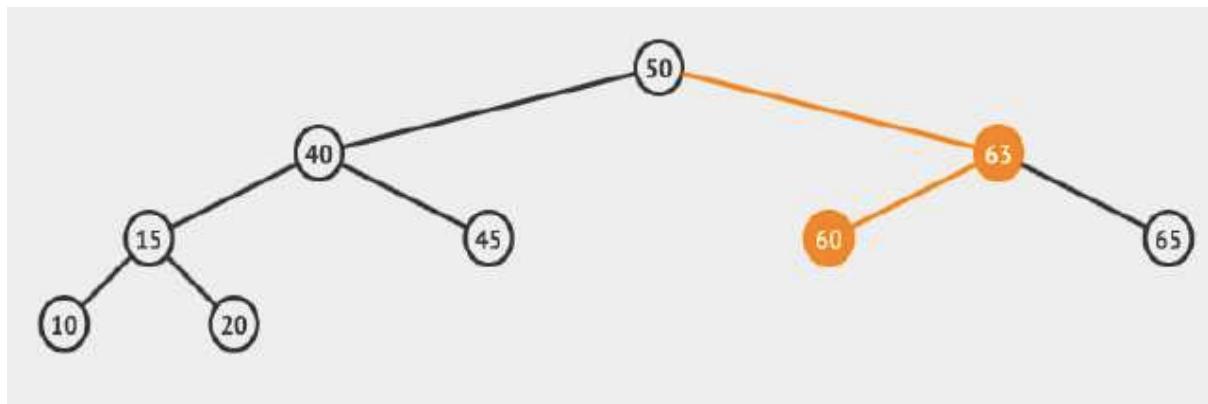
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo Rotación doble (right-left):



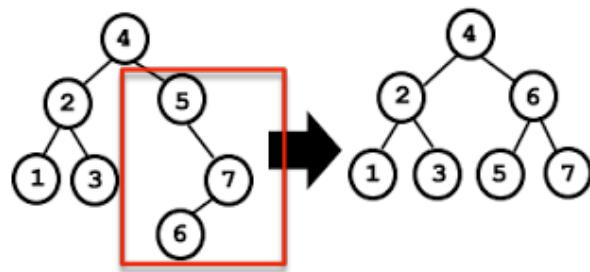
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Ejemplo Rotación doble (right-left):



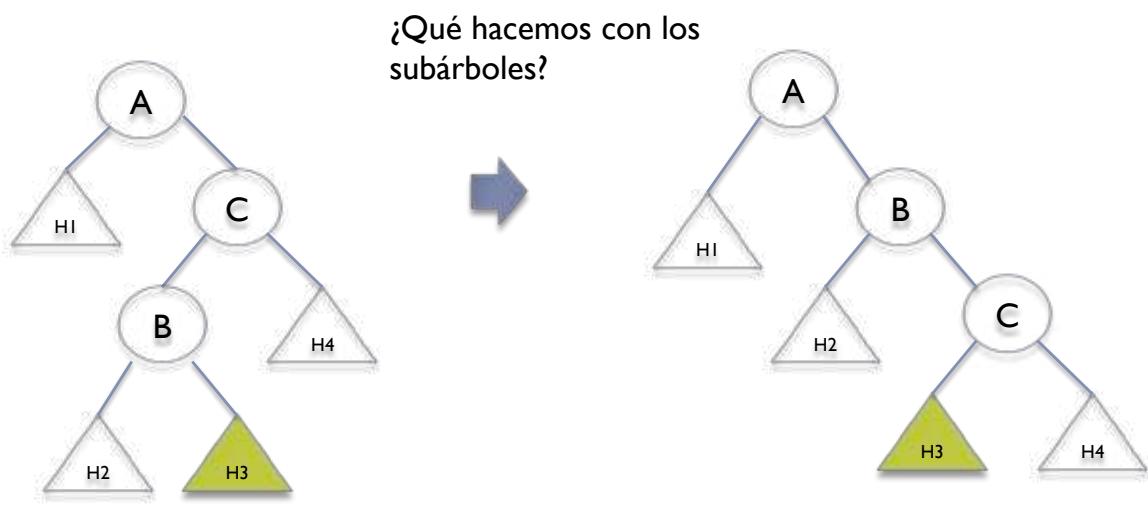
## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotaciones Dobles (ejemplo right-left):



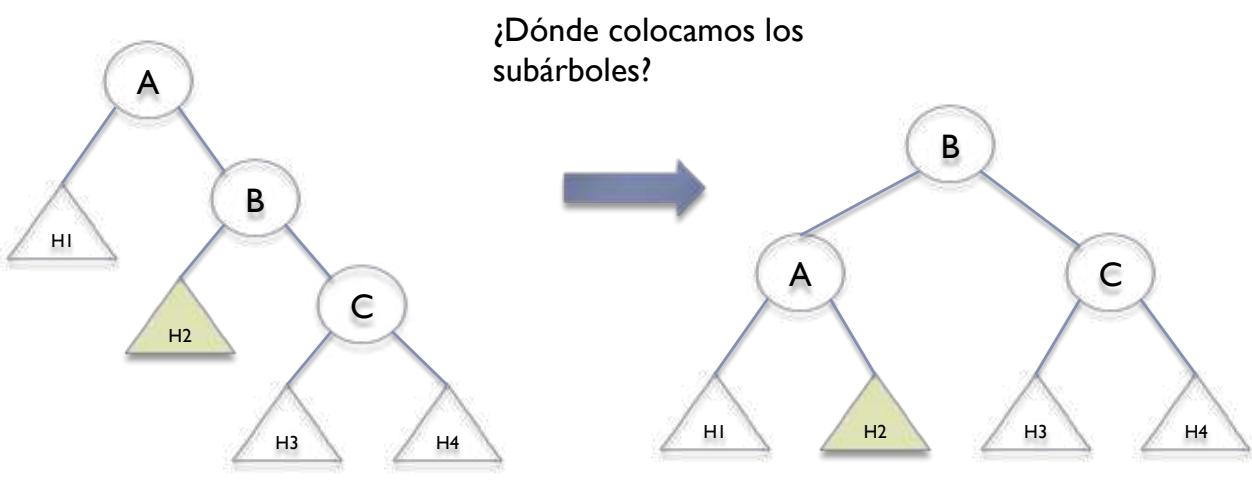
## ABB: Equilibrados en altura (árboles AVL)

### ► Rotación Doble (right-left):



## ABB: Equilibrados en altura (árboles AVL)

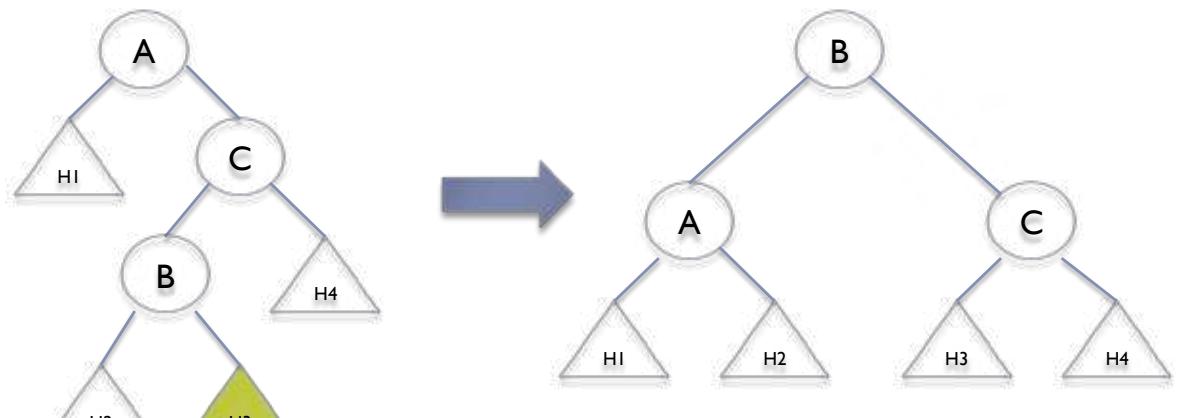
- ▶ Rotación Doble (right-left):



Segunda rotación: rotamos el nodo A como hijo izquierdo de B

## ABB: Equilibrados en altura (árboles AVL)

- ▶ Rotación doble (right-left):



InOrder:  $H_1AH_2BH_3CH_4$

=

InOrder:  $H_1AH_2BH_3CH_4$

## ABB: Equilibrados en altura (árboles AVL)

---

- ▶ Ventaja: El equilibrado se hace desde abajo, sólo en el camino desde el nodo insertado o borrado hacia la raíz
  - ▶ Por tanto, el equilibrado es  $O(\log n)$
- ▶ Desventaja: El árbol no queda tan compactado como en ABB perfectamente equilibrado. Aun así, las búsquedas también son  $O(\log n)$

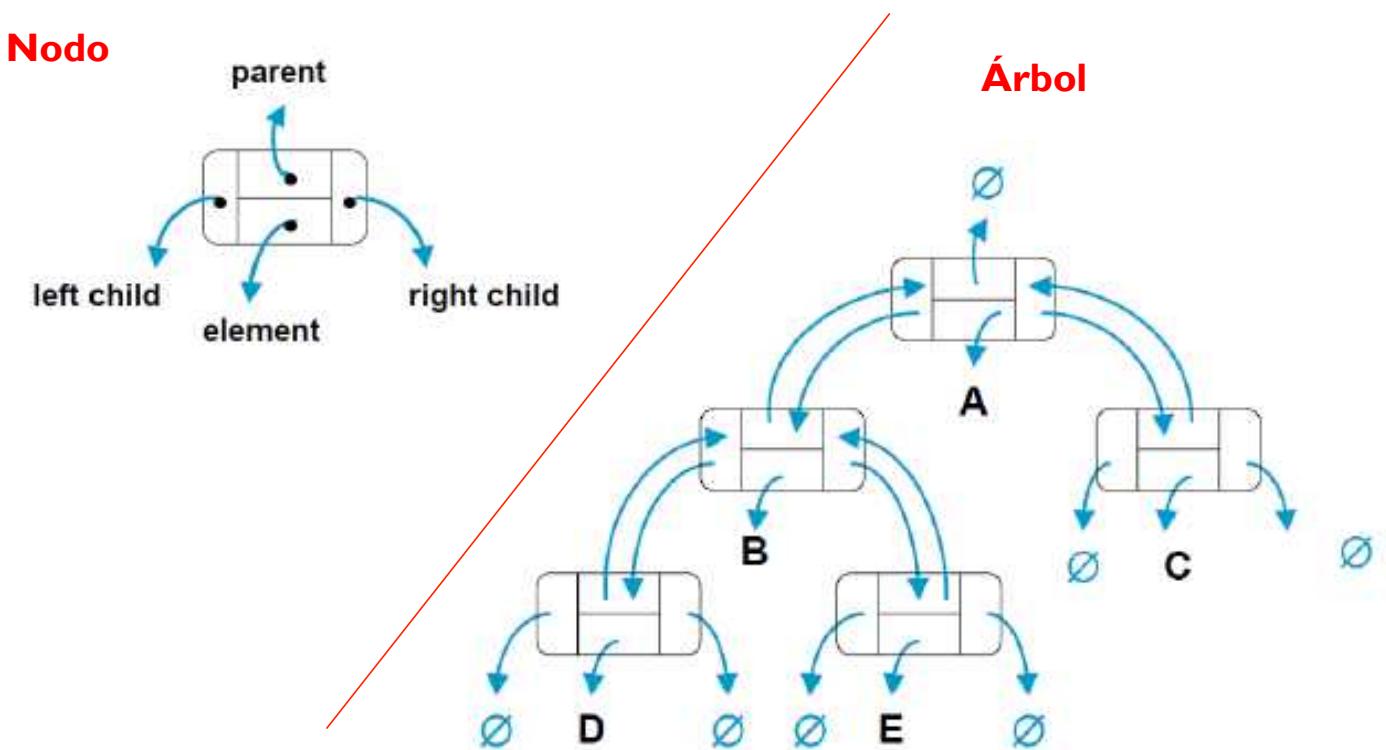


## Tema 5. Árboles **Implementando árboles binarios**



Estructura de Datos y Algoritmos (EDA)

# Árboles binarios: Implementación



## Implementando árboles binarios: BinTreeNode class (parámetros y constructor)

```
public class BinTreeNode {  
    Object elem;  
  
    BinTreeNode parent;  
    BinTreeNode left;  
    BinTreeNode right;  
  
    public BinTreeNode(Object element) {  
        elem = element;  
    }  
}
```

## Implementando árboles binarios: BinTreeNode class (método getSize)

```
public int getSize() {
    return getSize(this);
}

static int getSize(BinTreeNode subtree) {
    if (subtree == null) {
        return 0;
    } else {
        int result = 1 + getSize(subtree.left) + getSize(subtree.right);
        return result;
    }
}
```

## Implementando árboles binarios: BinTreeNode class (método getHeight)

- El método **getHeight** que devuelve la altura de un nodo.

```
public static int getHeight(BinTreeNode node) {  
    if (node == null) {  
        return 0;  
    } else {  
        int result = 1 + Math.max(getHeight(node.left), getHeight(node.right));  
        return result;  
    }  
}
```

## Implementando árboles binarios: BinTreeNode class (método `getDepth`)

- El método `getDepth` que devuelve la profundidad de un nodo.

```
public static int getDepth(BinTreeNode node) {  
    if (node==null) return -1;  
    else return 1 + getDepth(node.parent);  
}  
  
public static int getDepthIt(BinTreeNode node) {  
    if (node==null) return -1;  
    BinTreeNode nodeIt=node;  
    int level=0;  
    while (nodeIt.parent!=null) {  
        nodeIt=nodeIt.parent;  
        level++;  
    }  
    return level;  
}
```

## Implementando árboles binarios: BinTreeNode class (método getPreorder)

```
public SList getPreorder() {  
    SList list = new SList();  
    getPreorder(root, list);  
    return list;  
}  
  
public static void getPreorder(BinTreeNode node,  
                               SList list) {  
    if (node == null) return;  
    list.addLast(node.elem);  
    getPreorder(node.left, list);  
    getPreorder(node.right, list);  
}
```

## Implementando árboles binarios: BinTreeNode class (método getPostOrder)

```
public SList getPostOrder() {
    SList list = new SList();
    getPostOrder(root, list);
    return list;
}

public static void getPostOrder(BinTreeNode node, SList list) {
    if (node == null) return;

    getPostOrder(node.left, list);
    getPostOrder(node.right, list);

    list.addLast(node.elem);
    //System.out.println(node.elem)
}
```

---

## Implementando árboles binarios: BinTreeNode class (método getInOrder)

```
public SList getInOrder() {
    SList list = new SList();
    getInOrder(root, list);
    return list;
}

public static void getInOrder(BinTreeNode node, SList list) {
    if (node == null) return;

    getInOrder(node.left, list);
    list.addLast(node.elem);
    //System.out.println(node.elem)

    getInOrder(node.right, list);
}
```

## Implementando árboles binarios: Interfaz IBinTree

```
public interface IBinTree {  
    public int getSize();  
    public int getHeight();  
    public SList getPreorder();  
    public SList getPostOrder();  
    public SList getInOrder();  
}
```

## **Parte VIII**

### **Tema 6. Grafo**





## Tema 6. Grafos



Estructura de Datos y Algoritmos

# Contenidos

---

- ▶ **¿Qué es un grafo?.**
- ▶ TAD Grafo.
- ▶ Implementaciones
  - ▶ Matriz de adyacencias.
  - ▶ Lista de adyacencias
- ▶ Recorridos
  - ▶ En profundidad
  - ▶ En altura

## ¿Qué es un grafo?

- ▶ “A Graph is a way of representing relationships that exist between pairs of objects.” - Goodrich



# Los orígenes de la Teoría de Grafos

---

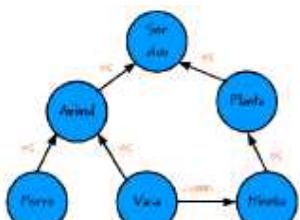
## ► El problema de los siete puentes de Königsberg (Euler)

- ¿Es posible dar un paseo comenzando desde cualquiera de estas regiones, pasando por todos los puentes, recorriendo sólo una vez cada uno, y regresando al mismo punto de partida?

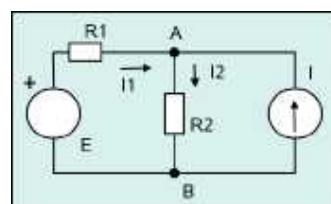


# Aplicaciones

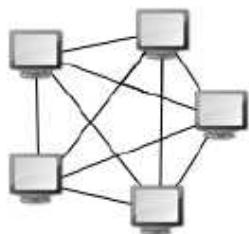
## ▶ Otras aplicaciones



Grafos conceptuales



Circuitos electrónicos



Redes de ordenadores



Organigramas



## Conceptos Básicos

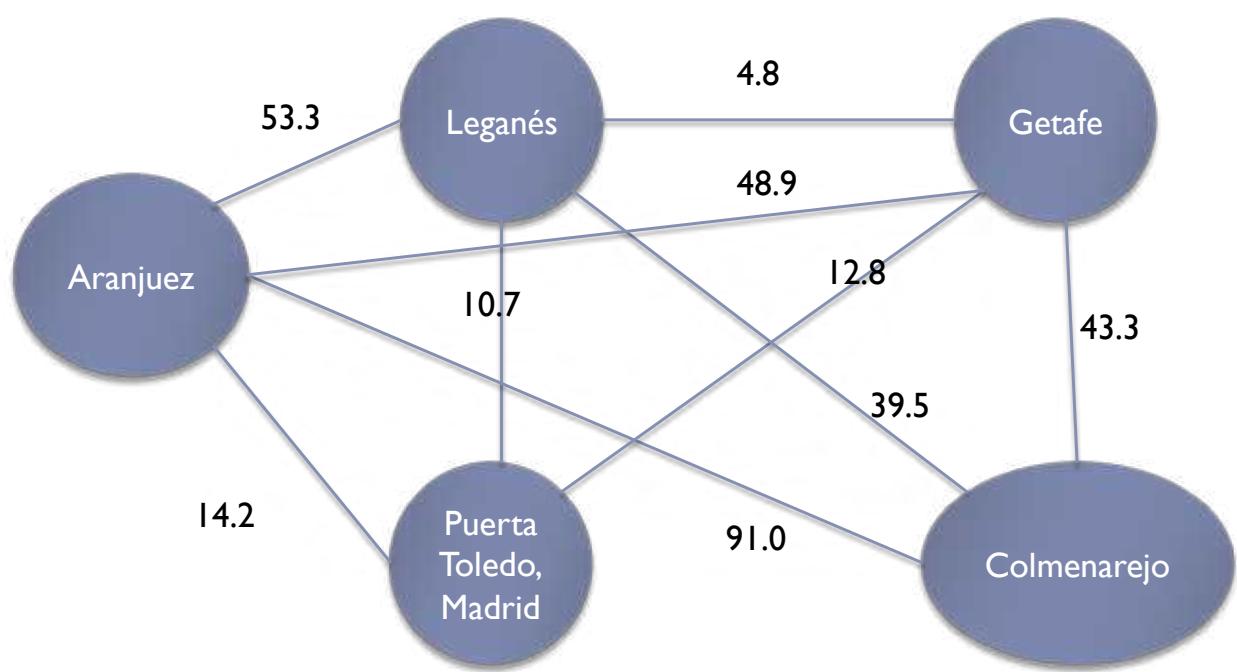
---

- ▶ Grafo  $G$  donde  $G=(V,A)$ .
  - ▶  $V$  es un conjunto de vértices (nodos)
  - ▶  $A$  es un conjunto de aristas (arcos).
    - ▶ Una arista es una conexión entre dos vértices.
    - ▶ Cada arista puede ser representada como una tupla  $(v,w)$  donde  $w,v \in V$
  - ▶ Además, cada artista puede tener un peso asociado (**grafo ponderado**). En este caso, la arista quedaría representada por una terna  $(v,w,p)$  donde  $p$  es el peso asociado a la arista entre  $v$  y  $w$ .

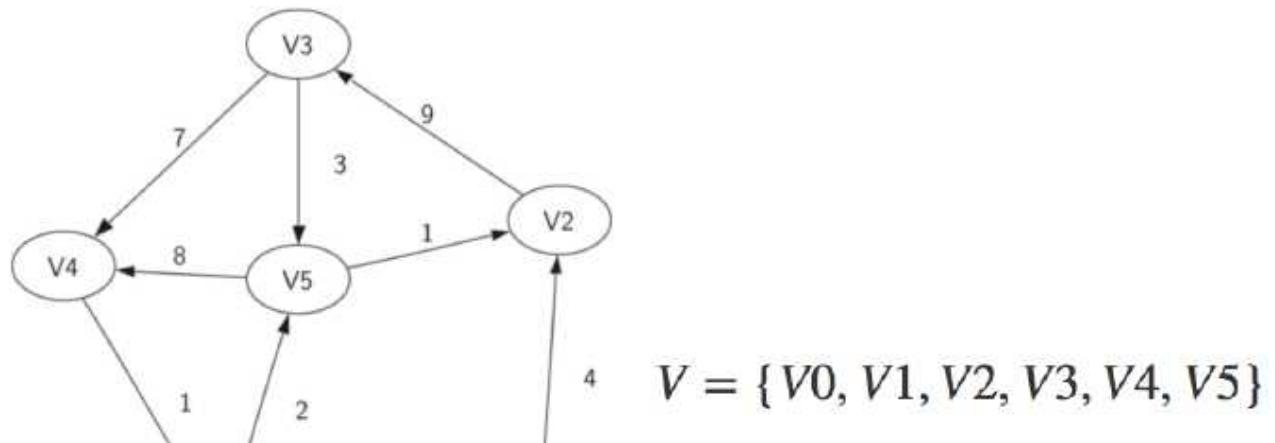


## Ejemplo I – Campus UC3M

---



## Ejemplo II

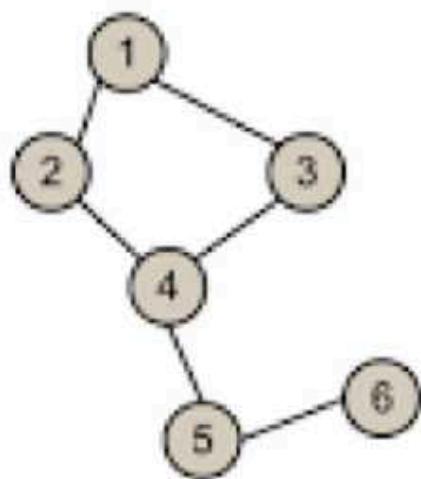


$$A = \left\{ \begin{array}{l} (v_0, v_1, 5), (v_1, v_2, 4), (v_2, v_3, 9), (v_3, v_4, 7), (v_4, v_0, 1), \\ (v_0, v_5, 2), (v_5, v_4, 8), (v_3, v_5, 3), (v_5, v_2, 1) \end{array} \right\}$$

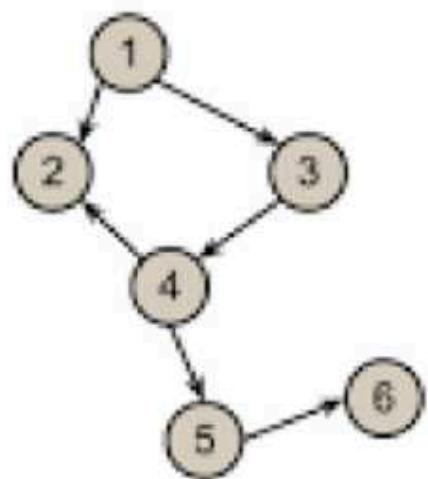


## Grafo No Dirigido vs Dirigido

---



No dirigido



dirigido

# Tipos de Grafos

---

## ▶ **Grafos no dirigidos.**

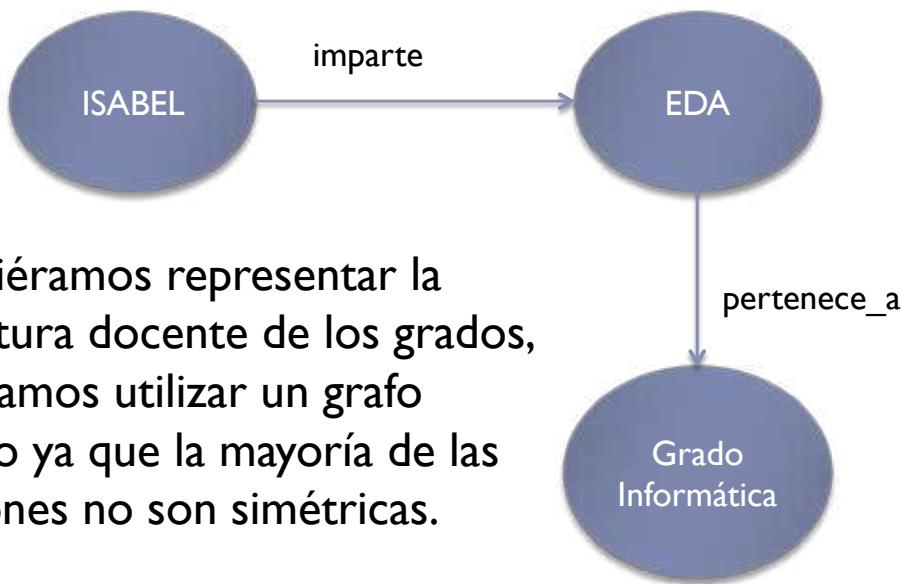
- ▶ Las aristas no tiene dirección, es decir,  $(u,v) = (v,u)$ . La arista se puede recorrer en ambos sentidos.
- ▶ Nos permiten representar relaciones simétricas y de colaboración.
- ▶ Ejemplo Grafo Campus UC3M.

## ▶ **Grafos dirigidos.**

- ▶ Cada arista  $(u,v)$  tiene una única dirección, siendo  $u$  el origen y  $v$  el vértice final.  $(u,v) \neq (v,u)$
- ▶ Nos permiten representar relaciones asimétricas y jerárquicas.

## Ejemplo I Grafo Dirigido

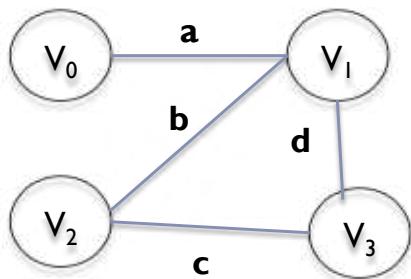
---



Si quisiéramos representar la estructura docente de los grados, deberíamos utilizar un grafo dirigido ya que la mayoría de las relaciones no son simétricas.

# Conceptos Básicos

- ▶ Dos vértices son **adyacentes** si existe una arista que los conecta.
- ▶ Una arista es **incidente** a un vértice si lo une con otro vértice.
- ▶ El **grado** de un vértice  $v$ ,  $\deg(v)$ , es el número de aristas conectadas a  $v$ .



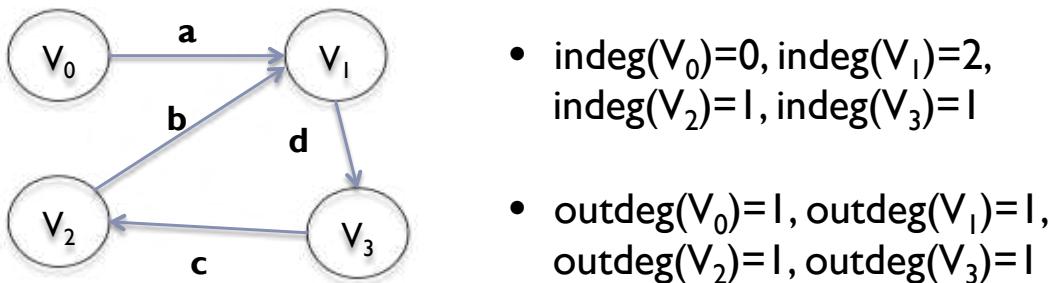
- Vértices adyacentes:  $(V_0, V_1)$ ,  $(V_1, V_2)$ ,  $(V_2, V_3)$ ,  $(V_1, V_3)$ .
- Grado de  $V_1 = 3$ .
- Las aristas a, b y d son incidentes en  $V_1$



# Conceptos Básicos

---

- ▶ En los grafos dirigidos, podemos distinguir entre:
- ▶ **Grado de entrada** de un vértice  $v$ ,  $\text{indeg}(v)$ , es el número de aristas que llegan al vértice  $v$ .
- ▶ **Grado de salida** de un vértice  $v$ ,  $\text{outdeg}(v)$ , es el número de aristas que parten del vértice  $v$ .



## Conceptos Básicos

---

- ▶ Proposición: Si  $G$  es un grafo con  $m$  aristas:

$$\sum_{v \in G} \deg(v) = 2m$$

- ▶ Proposición: Si  $G$  es un grafo dirigido con  $m$  aristas:

$$\sum_{v \in G} \text{in deg}(v) = \sum_{v \in G} \text{out deg}(v) = m$$



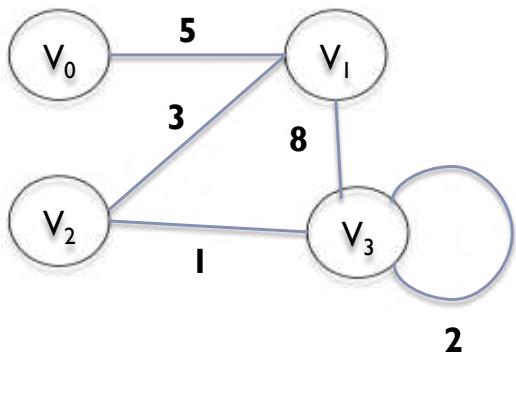
# Conceptos Básicos

---

- ▶ **Camino:** una secuencia de vértices conectados por aristas.
  - ▶ La longitud del camino sin pesos es el número de aristas en el camino.
  - ▶ La longitud del camino con pesos es la suma de los pesos de todas las aristas del camino.
- ▶ **Ciclo:** un ciclo en un grafo dirigido es una camino que empieza y termina en el mismo nodo.

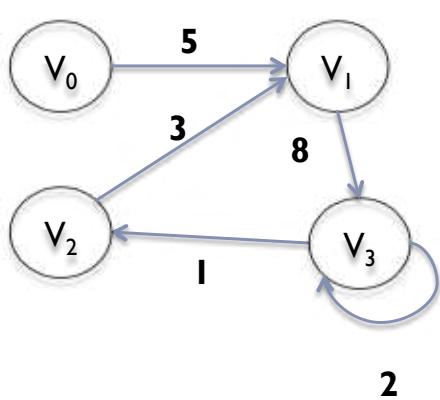


## Ejemplo camino y ciclo (grafo no dirigido)



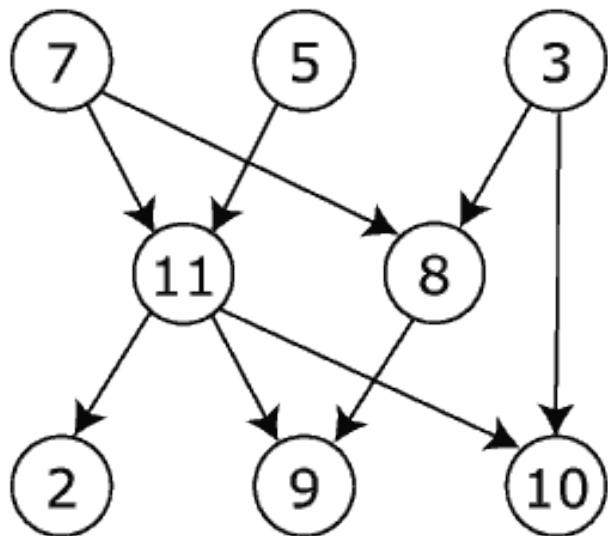
- $\langle v_0, v_1, v_2, v_3 \rangle$  camino simple (no se repiten nodos) de longitud  $5+3+1=9$
- $\langle v_0, v_1, v_2, v_3, v_1 \rangle$  camino de longitud  $5+3+1+8=17$
- $\langle v_0, v_3 \rangle$  no es un camino
- $\langle v_1, v_2, v_3, v_1 \rangle$  camino y ciclo
- $\langle v_3, v_3 \rangle$  bucle

## Ejemplo camino y ciclo (grafo dirigido)



- $\langle V_0, V_1, V_2, V_3 \rangle$  no es un camino.
- $\langle V_0, V_1, V_3, V_1 \rangle$  camino simple de longitud  $5 + 8 + 1 = 14$
- $\langle V_0, V_1, V_3, V_2, V_1 \rangle$  camino de longitud  $5+8+1+3=17$
- $\langle V_1, V_3, V_2, V_1 \rangle$  camino y ciclo
- $\langle V_3, V_3 \rangle$  bucle

## Grafo Acíclico



- ▶ Un grafo sin ciclos es un **grafo acíclico**.
- ▶ Un grafo dirigido sin ciclos se llama **grafo dirigido acíclico (DAG)**.

# Contenidos

---

- ▶ ¿Qué es un grafo?
- ▶ **TAD Grafo.**
- ▶ Implementaciones
  - ▶ Matriz de adyacencias.
  - ▶ Lista de adyacencias
- ▶ Recorridos
  - ▶ En profundidad
  - ▶ En altura

## TAD Grafo

```
//Interface for a graph whose vertices are integers
//and its weights are floats

public interface IGraph {

    //return the number of vertices
    public int sizeVertices();
    //return the number of edges
    public int sizeEdges();
    //shows the graph (vertices and edges)
    public void show();

    //returns the degree a vertex
    public int getDegree(int i);
    //return the inward-bound degree
    public int getInDegree(int i);
    //return the outward-bound degree
    public int getOutDegree(int i);
```



## TAD Grafo (cont.)

```
//create a new vertex
public void addVertex();
//create an edge between the vertices i and j
public void addEdge(int i, int j);
//create an edge between the vertices i and j with weight w
public void addEdge(int i, int j, float w);
//remove the edge between the vertices i and j
public void removeEdge(int i, int j);

//check if the pair of vertices (i,j) is an edge.
public boolean isEdge(int i, int j);
//returns the weight of the edge (i,j)
public float getWeightEdge(int i, int j);
//returns an array with the adjacent vertices of i
public int[] getAdjacents(int i);
```



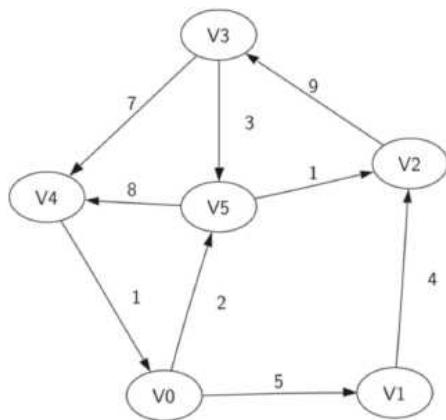
# Contenidos

---

- ▶ ¿Qué es un grafo?
- ▶ TAD Grafo.
- ▶ **Implementaciones**
  - ▶ Matriz de adyacencias.
  - ▶ Lista de adyacencias
- ▶ Recorridos
  - ▶ En profundidad
  - ▶ En altura

# Implementación basada en matriz

## ► La matriz de adyacencias

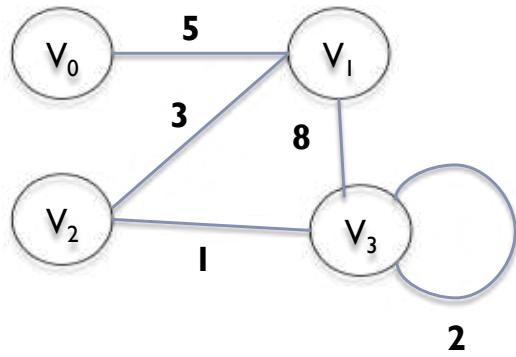


	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

► Picture taken from <http://interactivepython.org/runestone/static/pythonds/Graphs/AnAdjacencyMatrix.html>

## Implementación – Matriz de adyacencias

- ▶ Un grafo puede ser representado como una matriz cuadrada nxn, siendo n el número de vértices del grafo.
- ▶ Cada vértice v es representado por un entero (índice de v), en el rango  $\{0, 1, \dots, n-1\}$  siendo n el número de vértices

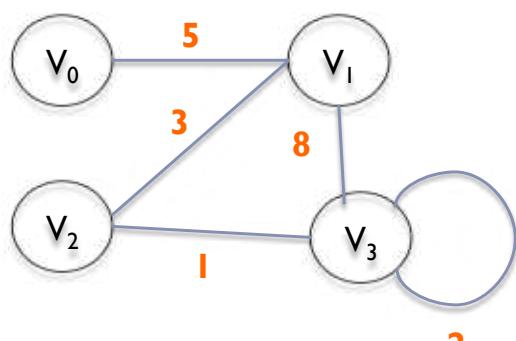


- $V_0 \rightarrow$  índice 0
- $V_1 \rightarrow$  índice 1
- $V_2 \rightarrow$  índice 2
- $V_3 \rightarrow$  índice 3



## Implementación – Matriz de adyacencias

- ▶ La matriz se puede implementar como un array bidimensional  $n \times n M$ , tal que el elemento  $M[i,j]$  guarda información sobre la arista  $(v,w)$ , si existe, donde  $v$  es el vértice con índice  $i$  y  $w$  es el vértice con índice  $j$ .

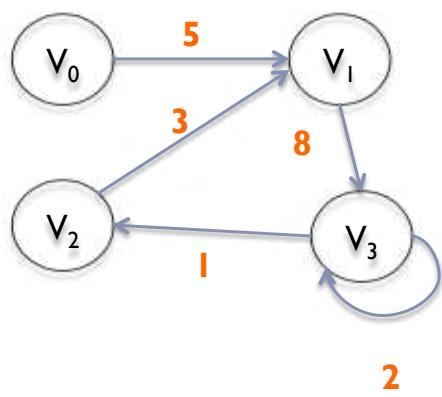


	0	1	2	3
0			5	
1	5		3	8
2		3		1
3		8	1	2

Si el grafo no es dirigido la matriz es simétrica



## Implementación – Matriz de adyacencias



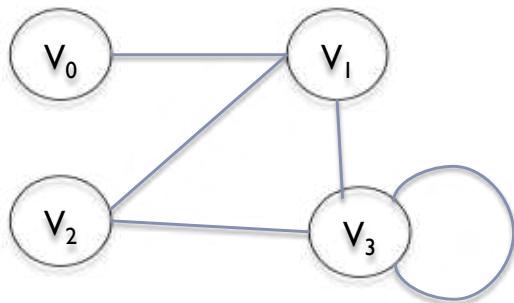
Si el grafo es dirigido la matriz NO es simétrica

	0	1	2	3
0			5	
1				8
2		3		
3			1	2



## Implementación – Matriz de adyacencias

- ▶ Si es un grafo no etiquetado, el grafo se podría representar con una matriz de booleanos,



	0	1	2	3
0	false	true	false	false
1	true	false	true	true
2	false	true	false	true
3	false	true	true	true

Si el grafo no es dirigido la matriz es simétrica



## Implementación – Matriz de adyacencias

---

- ▶ Vamos a ver una implementación para un grafo no ponderado.
- ▶ La matriz puede ser almacenada en un array bidimensional de booleanos (true indicará que existe arista y false que no existe).
- ▶ La creación de nuevos vértices podría implicar la necesidad de modificar el tamaño asignado a la matriz.
- ▶ Para evitarlo, vamos a definir un atributo que almacene el **número máximo de vértices** (en ningún caso, se permitirá añadir un nuevo vértice cuando ese umbral se haya alcanzando) y otro atributo que almacene el número actual de vértices.

## Implementación – Matriz de adyacencias

---

```
public class GraphMA implements IGraph {  
  
    boolean matrix[][];  
    //maximum number of vertices  
    int maxVertices;  
    //current number of vertices  
    int numVertices;  
    //true if the graph is directed, false eoc  
    boolean directed;
```

## Implementación – Matriz de adyacencias

```
public GraphMAFull(int n, int max, boolean d) {  
    //We checks if the values are right for the graph  
    if (max<=0)  
        throw new IllegalArgumentException("Negative maximum number of vertices!!!!");  
    if (n<=0)  
        throw new IllegalArgumentException("Negative number of vertices!!!!.");  
    if (n>max)  
        throw new IllegalArgumentException("number of vertices can never be greater than the maximum.");  
  
    maxVertices=max;  
    numVertices=n;  
    matrix=new float[maxVertices][maxVertices];  
    directed=d;  
}
```

- Primero deberemos comprobar que todos los argumentos del constructor reciben valores apropiados: tanto el número máximo como el número de vértices debe ser siempre un número positivo.
- Además, el número de vértices nunca deberá sobrepasar el número máximo de vértices.
- El constructor crea el array bidimensional. Por defecto, todas las posiciones son inicializadas a false.

## Implementación – Matriz de adyacencias

```
public void addVertex() {  
    if (numVertices==maxVertices) {  
        System.out.println("Cannot add new vertices!!!");  
        return;  
    }  
    numVertices++;  
}
```

- Lo primero que tenemos que hacer es comprobar que el nuevo número total de vértices no va a sobrepasar el número máximo permitido.
- Por último, sólo tendremos que incrementar en uno el número actual de vértices.
- No hace falta inicializar la matriz para el nuevo vértice, porque por defecto todas sus posiciones en la matriz son false.

## Implementación – Matriz de adyacencias

---

```
//check if i is a right vertex
private boolean checkVertex(int i) {
    if (i>=0 && i<numVertices) return true;
    else return false;
}
```

- Vamos a usar un método auxiliar para comprobar si un índice representa o no un vértice en el grafo.
- Para que sea un vértice del grafo siempre deberá ser positivo y menor que numVertices, porque los vértices del grafo toman valores en el rango [0, numVertices-1].

## Implementación – Matriz de adyacencias

---

```
public void addEdge(int i, int j) {  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    if (!checkVertex(j))  
        throw new IllegalArgumentException("Nonexistent vertex " + j);  
  
    matrix[i][j]=true;  
    if (!directed) matrix[j][i]=true;  
}
```

Una vez comprobadas que ambas índices son correctos, simplemente lo que tenemos que hacer es actualizar la posición `matrix[i,j]` a true. Si no es dirigido, también tendremos que poner su posición simétrica

# Implementación – Matriz de adyacencias

---

```
@Override  
public boolean isEdge(int i, int j) {  
    //checks if the indexes are right  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    if (!checkVertex(j))  
        throw new IllegalArgumentException("Nonexistent vertex " + j);  
    return matrix[i][j];  
}
```

- En primer lugar, tenemos que comprobar que los índices  $i$  y  $j$  son correctos.
- El par  $(i,j)$  es un arista si  $\text{matrix}[i,j]$  guarda true.

# Implementación – Matriz de adyacencias

```
@Override  
public void removeEdge(int i, int j) {  
    //checks if the indexes are right  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    if (!checkVertex(j))  
        throw new IllegalArgumentException("Nonexistent vertex " + j);  
    matrix[i][j]=false;  
    if (!directed) matrix[j][i]=false;  
}
```

- Primero tenemos que comprobar que son índices válidos
- Una vez comprobado que son índices válidos, basta con modificar el valor del array en esa posición (i,j) a false.
- Si no es dirigido, también tendremos que hacerlo en su elemento simétrico (j,i)

## Implementación – Matriz de adyacencias

```
public int sizeVertices() {  
    return numVertices;  
}  
  
public int sizeEdges() {  
    int numEdges=0;  
    if (directed) {  
        for (int i=0;i<numVertices;i++) {  
            for (int j=0;j<numVertices;j++) {  
                if (matrix[i][j]!=false) numEdges++;  
            }  
        }  
    } else {  
        for (int i=0;i<numVertices;i++) {  
            for (int j=i;j<numVertices;j++) {  
                if (matrix[i][j]!=false) numEdges++;  
            }  
        }  
    }  
    return numEdges;  
}
```

- Equivale a contar todos los elementos true en la matriz.
- Si no es dirigido, como la matriz es simétrica, sólo necesitaremos visitar una de las dos partes divididas por la diagonal.

## Implementación – Matriz de adyacencias

```
public int getOutDegree(int i) {  
    if (!directed) {  
        System.out.println("Graph non directed!!!");  
        return 0;  
    }  
    //checks if the vertex is right  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    int outdeg=0;  
    for (int col=0;col<numVertices;col++) {  
        if (matrix[i][col]!=false) outdeg++;  
    }  
    return outdeg;  
}
```

Las filas representan los vértices de origen  
y las columnas los vértices destino

- Incrementamos 1 por cada columna cuyo índice tenga una arista con i, es decir, matrix[i,col].

## Implementación – Matriz de adyacencias

```
public int getInDegree(int i) {  
    if (!directed) {  
        System.out.println("Graph non directed!!!");  
        return 0;  
    }  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    int indeg=0;  
    for (int row=0;row<numVertices;row++) {  
        if (matrix[row][i]!=false) indeg++;  
    }  
  
    return indeg;  
}
```

Las filas representan los vértices de origen  
y las columnas los vértices destino

- Incrementamos 1 por cada fila cuyo índice tenga una arista con i, es decir, matrix[row,i].

## Implementación – Matriz de adyacencias

---

```
public int getDegree(int i) {  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    int degree=0;  
    if (directed) degree=getInDegree(i)+getOutDegree(i);  
    else {  
        for (int row=0;row<numVertices;row++) {  
            if (matrix[row][i]!=false) degree++;  
        }  
    }  
    return degree;  
}
```

Si el grafo no es dirigido, el grado será la suma del grado de entrada y el grado de salida.

En otro caso, bastará con que contemos las aristas de entrada en ese vértice. También se podría hacer contando las aristas de salida (pero nunca ambas).

## Implementación – Matriz de adyacencias

```
//returns an array with the adjacent vertices for i
public int[] getAdjacents(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex " + i);

    //obtains the number of adjacent vertices,
    //which will be the size of the array
    int numAdjacents=0;
    if (directed) numAdjacents=getOutDegree(i);
    else numAdjacents=getDegree(i);

    int[] adjacents=new int[numAdjacents];

    if (numAdjacents>0) {
        int j=0;
        //gets the edges (i,col) and saves col into adjacents.
        for (int col=0; col<numVertices;col++) {
            if (matrix[i][col]!=null) {
                adjacents[j]=col;
                j++;
            }
        }
    }
    //return an array with the adjacent vertices of i
    return adjacents;
}
```

## Problema

---

- ▶ ¿Cómo representarías un grafo ponderado con valores reales (es decir, un grafo donde todas las aristas tienen un valor real asociado)?. Implementa una clase para representar grafos ponderados.

# Contenidos

---

- ▶ ¿Qué es un grafo?
- ▶ TAD Grafo.
- ▶ **Implementaciones**
  - ▶ Matriz de adyacencias.
  - ▶ **Lista de adyacencias**
- ▶ Recorridos
  - ▶ En profundidad
  - ▶ En altura

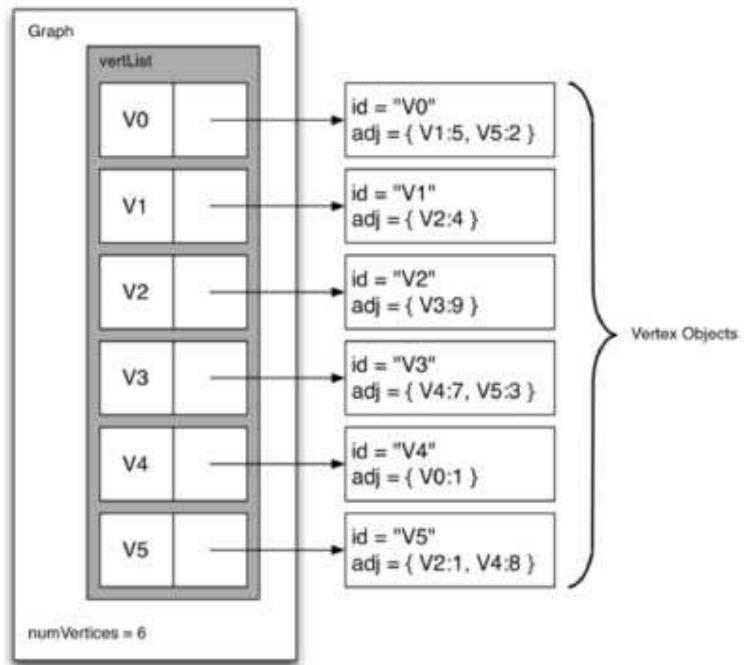
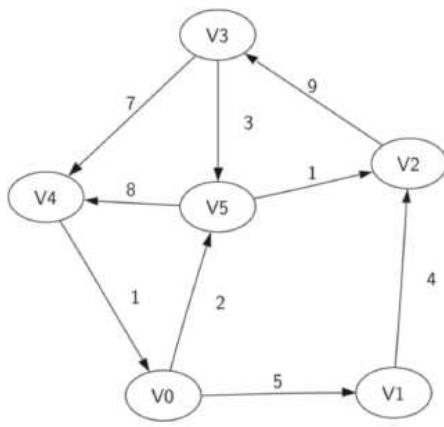
## Implementación – Lista de adyacencias

---

- ▶ La matriz de adyacencia consume memoria y la complejidad de las operaciones con la matriz es alta (por ejemplo, el método que muestra la matriz tiene complejidad cuadrática).
  - ▶ Una lista de adyacencia sólo almacena la información para los aristas existentes, en lugar de almacenar todas las posibles combinaciones como ocurría en la matriz de adyacencias.
  - ▶ Necesita menos espacio de memoria y su coste computacional es menor.
- 
- ▶ Picture taken from <http://interactivepython.org/runestone/static/pythonds/Graphs/AnAdjacencyList.html>

# Implementación – Lista de adyacencias

## ▶ Lista adyacencias



▶ Picture taken from <http://interactivepython.org/runestone/static/pythonds/Graphs/AnAdjacencyList.html>

## Implementación – Lista de adyacencias

---

- ▶ Si tenemos un número fijo de vértices, el grafo se puede representar como un array de listas enlazadas (tema 2).
- ▶ Cada posición del array representa un vértice y almacena la referencia a la lista de vértices adyacentes a dicho vértice (implementada como lista enlazada).
- ▶ Cada uno de los nodos almacenará la información sobre el vértice adyacente. Si el grafo es ponderado, también debería almacenar su peso.

---

▶ Picture taken from <http://interactivepython.org/runestone/static/pythonds/Graphs/AnAdjacencyList.html>

## Implementación – Lista de adyacencias

---

```
import dlist.DListVertex;

public class GraphLAFull implements IGraph {

    int numVertices;
    int maxVertices;

    DListVertex[] vertices;
    boolean directed;
```



## Implementación – Lista de adyacencias

---

```
public GraphLAFull(int n, int max, boolean d) {  
    if (max<=0)  
        throw new IllegalArgumentException("Negative maximum number of vertices!!!!");  
    if (n<=0)  
        throw new IllegalArgumentException("Negative number of vertices!!!!.");  
    if (n>max)  
        throw new IllegalArgumentException("number of vertices can never "  
            + "be greater than the maximum.");  
  
    maxVertices=max;  
    vertices=new DListVertex[maxVertices];  
    numVertices=n;  
    //creates each list  
    for (int i=0; i<numVertices;i++) {  
        vertices[i]=new DListVertex();  
    }  
    directed=d;  
}
```



## Implementación – Lista de adyacencias

Comprobamos que los índices son correctos.

```
public void addEdge(int i, int j, float w) {  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    if (!checkVertex(j))  
        throw new IllegalArgumentException("Nonexistent vertex " + j);  
  
    vertices[i].addLast(j,w);  
    //if it is a non-directed graph  
    if (!directed) vertices[j].addLast(i,w);  
}
```

Tenemos que añadir el vértice j a la lista de vértices adyacentes del vértice i (que está almacenada en vertices[i]).

Si el grafo no es dirigido, deberemos también almacenar el vértice i como adyacente del vértice j



## Implementación – Lista de adyacencias

```
public void removeEdge(int i, int j) {  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    if (!checkVertex(j))  
        throw new IllegalArgumentException("Nonexistent vertex " + j);  
  
    int index=vertices[i].getIndexOf(j);  
    vertices[i].removeAt(index);  
  
    if (!directed) {  
        index=vertices[j].getIndexOf(i);  
        vertices[j].removeAt(index);  
    }  
}
```

Si el grafo no es dirigido, deberemos también  
borrar el vértice i como adyacente del vértice j



## Implementación – Lista de adyacencias

---

```
public boolean isEdge(int i, int j) {  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    if (!checkVertex(j))  
        throw new IllegalArgumentException("Nonexistent vertex " + j);  
  
    boolean result=vertices[i].contains(j);  
    return result;  
}
```



## Implementación – Lista de adyacencias

```
public int getOutDegree(int i) {  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
  
    int outdegree=0;  
    outdegree=vertices[i].getSize();  
    return outdegree;  
}  
Sólo para grafos dirigidos
```

```
public int getInDegree(int i) {  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    int indegree=0;  
    for (int j=0; j<numVertices;j++) {  
        if (vertices[j].contains(i)) indegree++;  
    }  
    return indegree;  
}
```

## Implementación – Lista de adyacencias

---

```
public int getDegree(int i) {  
    int degree=0;  
    if (directed) {  
        degree=getOutDegree(i)+getInDegree(i);  
    } else degree=vertices[i].getSize();  
    return degree;  
}
```

El grado de un vértice en un grado dirigido es igual a la suma de su grado de entrada y de su grado de salida.

En un grado no dirigido, es suficiente con obtener el número de vértices adjacentes a dicho vértice.



## Implementación – Lista de adyacencias

---

```
public int[] getAdjacents(int i) {  
    if (!checkVertex(i))  
        throw new IllegalArgumentException("Nonexistent vertex " + i);  
    //gets the number of adjacent vertices  
    int numAdj=vertices[i].getSize();  
    //creates the array  
    int[] adjVertices=new int[numAdj];  
    //saves the adjacent vertices into the array  
    for (int j=0; j<numAdj; j++) {  
        adjVertices[j]=vertices[i].getVertexAt(j);  
    }  
    //return the array with the adjacent vertices of i  
    return adjVertices;  
}
```



## Problemas a resolver

---

- ▶ Añade un método que muestre el contenido del grafo. Es decir, por cada vértice, que muestra su lista vértices adyacentes.
- ▶ Implementa un método que compruebe si el grafo es dirigido o no. (No puedes usar el atributo directed!!!)

# Contenidos

---

- ▶ ¿Qué es un grafo?.
- ▶ TAD Grafo.
- ▶ Implementaciones
  - ▶ Matriz de adyacencias.
  - ▶ Lista de adyacencias
- ▶ **Recorridos**
  - ▶ **En amplitud**
  - ▶ En profundidad

## Recorrido en amplitud

---

- ▶ Recorrido basado en estructura FIFO (first in first out).
- ▶ Se toma un vértice como inicial, y se visitan todos sus adyacentes. Una vez visitados éstos, se continúa visitando los adyacentes de cada uno de ellos, hasta que no se pueden alcanzar más vértices.
- ▶ Se repite el proceso mientras haya nodos sin visitar.
- ▶ Es necesario utilizar una estructura de cola para ir almacenando los vértices a medida que se llega a ellos.
- ▶ Ver animación <http://visualgo.net/dfsbfs.html> (bfs)

# Recorrido en amplitud

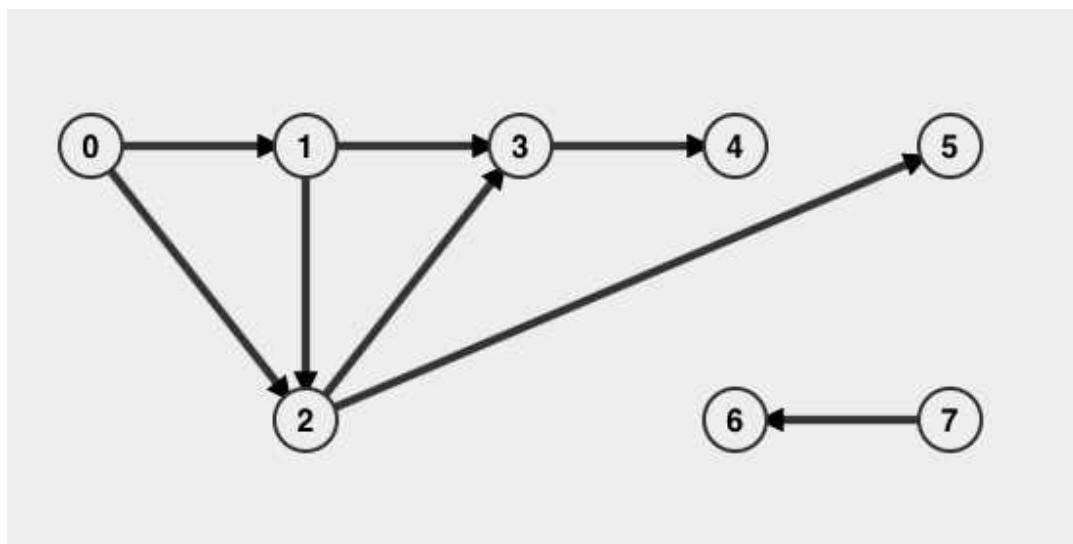
---

## ► Pasos del algoritmo:

1. Se toma un vértice  $v$  como inicial y se imprime.
2. Se visitan cada uno de sus nodos adyacentes y se almacenan en la cola.
3. Se desencola el primer elemento de la cola y se marca como visitado.
4. Se repiten los pasos 2 y 3 mientras haya elementos en la cola.
5. Si la cola no está vacía y quedan vértices sin recorrer se debe elegir un vértice no visitado y repetir los puntos 2-3-4.
6. El algoritmo termina cuando todos los vértices del grafo han sido visitados.

## Recorrido en amplitud

---



## Recorrido en amplitud

---

- Tomamos el índice inicial: 0 y lo visitamos. (Salida=0)
- Recuperamos sus adyacentes (1,2) y los encolamos: (cola=1,2)
- Desencolamos el 1 ( $c=2$ ) y lo visitamos (Salida = 0,1).
- Recuperamos sus adyacentes (2,3). Sólo encolamos el 3 porque el 2 ya fue añadido ( $c=2,3$ ).
- Desencolamos el 2 ( $c=3$ ) y lo visitamos (Salida=0,1,2).
- Recuperamos sus adyacentes (3,5). Sólo encolamos el 5 porque el 3 ya fue añadido ( $c=3,5$ ).
- Desencolamos el 3 ( $c=5$ ) y lo visitamos (Salida=0,1,2,3).
- Recuperamos sus adyacentes (4). Encolamos el 4 ( $c=5,4$ ).

## Recorrido en amplitud. Ejemplo (cont)

---

- Desencolamos el 5 ( $c=4$ ) y lo visitamos ( $\text{Salida}=0,1,2,3,5$ ).
- Como 5 no tiene adyacentes no hacemos nada y continuamos.
- Desencolamos el 4 ( $c=\text{empty}$ ) y lo visitamos ( $\text{Salida}=0,1,2,3,5,4$ ).
- La cola está vacía.
- Como en el grafo existen nodos sin visitar (6 y 7), continuamos.
- Elegimos el 6 y lo visitamos ( $\text{Salida}=0,1,2,3,5,4,6$ ). Como no tiene adyacentes no encolamos nada.
- Elegimos el 7 y lo visitamos ( $\text{Salida}=0,1,2,3,5,4,6,7$ ). Su único adyacente ya ha sido visitado.
- Hemos terminado porque no quedan nodos por visitar.

# Implementación Recorrido en amplitud

```
public void breadth() {  
    System.out.println("breadth traverse of the graph:");  
  
    //to mark when a vertex has already been shown  
    boolean visited[] = new boolean[numVertices];  
  
    //we have to traverse all vertices  
    for (int i=0; i<numVertices; i++) {  
        if (!visited[i]) { //we only process the non-visited vertex  
            breadth(i, visited);  
            System.out.println();  
        }  
    }  
}
```

Llama a otro método auxiliar que va a hacer el recorrido en amplitud de cada vértice (no visitado previamente).

Para registrar que vértices ya han sido visitados o no, usamos un array de booleanos

```
//breadth order for the vertex i
protected void breadth(int i, boolean[] visited) {
    //this array helps to mark what vertices have been stored into the queue
    boolean stored[] = new boolean[numVertices];
    System.out.println("breadth traverse for " + i);
    //we use a queue to save the adjacent vertices that we visit
    SQueue q = new SQueue();
    //enqueue the first
    q.enqueue(i);
    //while the queue is not empty
    while (!q.isEmpty()) {
        //gets the first
        int vertex = q.dequeue();
        //shows the vertex and marks it as visited
        System.out.print(vertex + "\t");
        visited[vertex] = true;
        //gets its adjacent vertices
        int[] adjacents = getAdjacents(vertex);
        for (int adjVertex : adjacents) {
            //enqueue only those that have not been visited or stored yet
            if (!visited[adjVertex] && !stored[adjVertex]) {
                q.enqueue(adjVertex);
                stored[adjVertex] = true;
            }
        }
    }
}
```

## Recorrido en profundidad

---

- ▶ Va localizando los posibles caminos y en el caso de no poder continuar, vuelve al punto donde existen nuevos caminos posibles con el fin de visitar todos los vértices.
- ▶ Ver animación <http://visualgo.net/dfsbfs.html> (dfs)

## Recorrido en profundidad

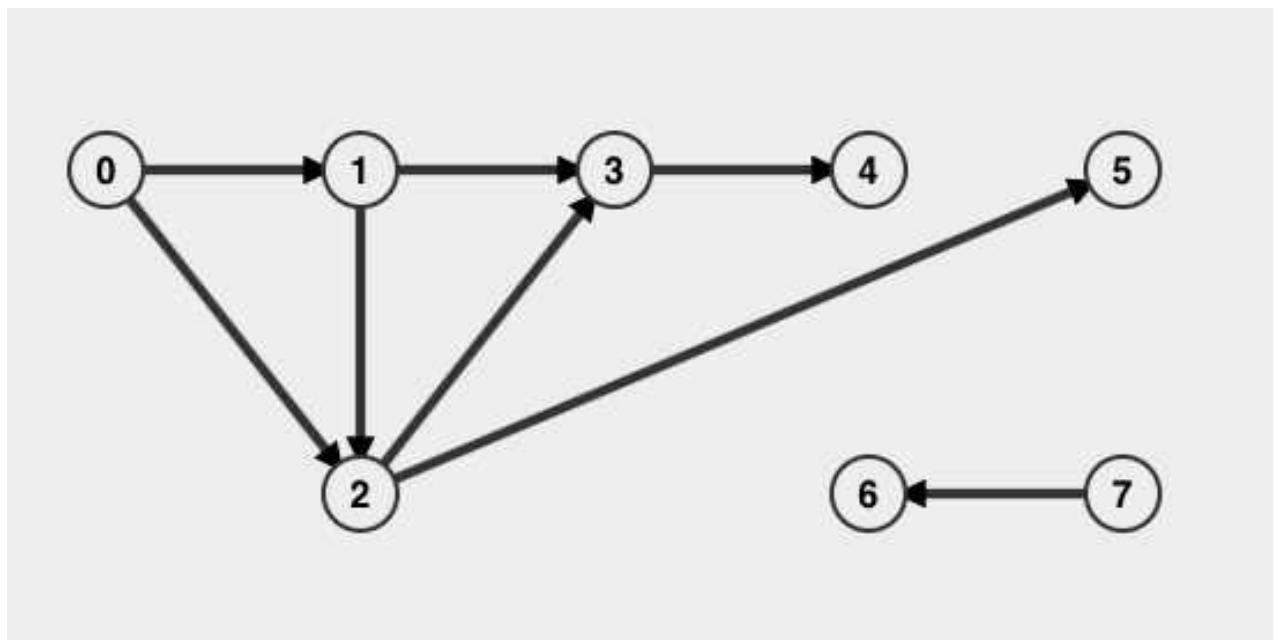
---

### ► Pasos del algoritmo:

1. Se toma un vértice  $v$  como inicial y se marca como visitado.
2. Por cada vértice  $w$ , no visitado y adyacente a  $v$ , deberemos hacer una llamada recursiva sobre  $w$ .  
Repetir mientras haya vértices no visitados y adyacentes a  $v$ .
3. Repetir 1-2 mientras haya vértices no visitados. El algoritmo termina cuando todos los vértices del grafo han sido visitados.

## Recorrido en profundidad

---



## Recorrido en profundidad

---

- ▶ Comenzamos con el vértice inicial  $v=0$ . Lo visitamos (salida =0) y obtenemos sus adyacentes son  $\{1,2\}$ .
  - ▶ Empezamos con 1, lo visitamos (salida=0,1). Obtenemos sus adyacentes  $\{2,3\}$ .
    - ▶ Tomamos el 2, lo visitamos (salida=0,1,2). Obtenemos sus adyacentes  $\{3,5\}$ .
      - Visitamos el 3 (salida =0,1,2,3) y recuperamos sus adyacentes  $\{4\}$ .
      - Visitamos al 4 (salida=0,1,2,3,4) como no tiene más adyacentes, continuamos con los adyacentes de 3. Como 3 no tiene más adyacentes, continuamos con los adyacentes de 2.
      - Visitamos el 5 (salida =0,1,2,3,4,5) y como no tiene más adyacentes regresamos al 2.
      - El 2 no tiene más adyacentes por lo que continuar, así que debemos regresar al 1.
    - ▶ El 1 tiene otro adyacente, 3, pero esté ya ha sido visitado, así que no debemos continuar por ese camino. Continuamos con los adyacentes de 1 ya no tiene más adyacentes por recorrer.
  - ▶ El otro adyacente de 0, es 2, que no tenemos que visitar porque ya ha sido visitado, y por tanto, no debemos continuar por ese camino.
  - ▶ Como aún quedan nodos por visitar, elegimos otro de los no visitados, 6 y lo visitamos. (Salida=0,1,2,3,4,5,6). 6 no tiene adyacentes.
  - ▶ Visitamos el único que queda por visitar. (salida=0,1,2,3,4,5,6,7). Tiene un adyacente (6) pero ya ha sido visitado.
  - ▶ Hemos terminado.

## Recorrido en profundidad

---

```
public void depth() {  
    System.out.println("depth traverse of the graph:");  
    //to mark when a vertex has already been shown  
    boolean visited[] = new boolean[numVertices];  
    //we have to traverse all vertices  
    for (int i=0; i<numVertices; i++) {  
        if (!visited[i]) depth(i, visited);  
    }  
}
```

Llama a otro método auxiliar que va a hacer el recorrido en profundidad de cada vértice (no visitado previamente).  
Para registrar que vértices ya han sido visitados o no, usamos un array de booleanos

## Recorrido en profundidad

---

```
protected void depth(int i,boolean[] visited) {  
    //prints the vertex and marks as visited  
    System.out.print(i+"\t");  
    visited[i]=true;  
    //gets its adjacent vertices  
    int[] adjacents=getAdjacents(i);  
    for (int adjV:adjacents) {  
        if (!visited[adjV]) {  
            //only depth traverses those adjacent vertices  
            //that have not been visited yet  
            depth(adjV,visited);  
        }  
    }  
}
```

## **Parte IX**

### **Tema 7. Estrategias de algoritmos**



# Tema 7. Algoritmos I. Estrategias de algoritmos

Estructura de datos y algoritmos (EDA)

# Algunos conceptos

---

- ▶ Un **algoritmo** es una secuencia finita y bien definida de pasos utilizada para resolver un problema bien definido
- ▶ **Estrategia del algoritmo**
  - ▶ Enfoque para resolver un problema
  - ▶ Se pueden combinar varios enfoques
- ▶ **Estructura de algoritmos:**
  - ▶ **Iterativo:** se utiliza un bucle para encontrar la solución
  - ▶ **Recursivo:** una función que se llama a sí misma

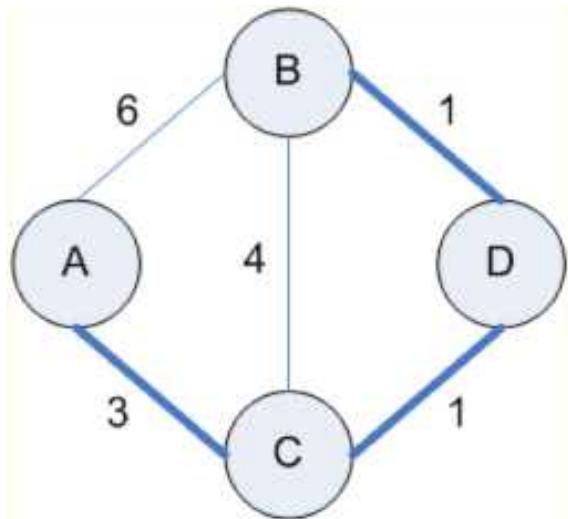
# Problema tipo

## ▶ **Se satisface**

- ▶ Encontrar cualquier solución que satisfaga
- ▶ Ej: Encontrar un camino de A a B

## ▶ **Optimización**

- ▶ Encontrar la mejor solución
- ▶ Ej: Encontrar el camino más corto de A a B



This example was taken from [http://cs.smu.ca/~porter/csc/common\\_341\\_342/notes/graphs\\_shortest\\_path.html](http://cs.smu.ca/~porter/csc/common_341_342/notes/graphs_shortest_path.html)

# Principales estrategias de algoritmos

---

- ▶ Algoritmos recursivos
  - ▶ Algoritmos de divide y vencerás
  - ▶ Algoritmos de Backtracking
  - ▶ Algoritmos de programación dinámica
  - ▶ Algoritmos ‘greedy’
  - ▶ Algoritmos de fuerza de bruta
  - ▶ Algoritmos de ramificación y poda
  - ▶ Algoritmos heurísticos
- 
- Heurística y optimización,  
Curso 3<sup>a</sup>,  
Semestre I°

## Divide y vencerás

---

- ▶ Basado en dividir el problema en sub-problemas
- ▶ Aproximación en tres pasos:
  - 1) **Dividir:** dividir el problema en sub-problemas más pequeños del mismo tipo
  - 2) **Vencer:** resolver recursivamente cada sub-problema, resolver cada sub-problema independientemente.
  - 3) **Combinar:** combinar soluciones para resolver el problema original
- ▶ Por lo general, contiene dos o más llamadas recursivas

# Divide y vencerás

---

## ► Esquema General

**divide\_venceras (p: problema)**

*dividir (p, p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>k</sub>)*

*para i = 1, 2, ..., k*

*s<sub>i</sub> = resolver (p<sub>i</sub>)*

**resolver (p<sub>i</sub>) puede ser  
recursivo siendo una nueva  
llamada a divide\_venceras**

*solución = combinar (s<sub>1</sub>, s<sub>2</sub>, ..., s<sub>k</sub>)*

## Divide y vencerás

---

- ▶ Algunos ejemplos:
  - ▶ Búsqueda binaria
  - ▶ Encontrar el elemento máximo en un array
  - ▶ Merge-sort
  - ▶ Quick-sort

## Divide y vencerás. Búsqueda binaria

---

- ▶ Dado un array  $A[]$  ordenado de enteros, escribir un método que busque un número entero  $x$  en  $A[]$
- ▶ El método tiene que devolver la (primera) posición de  $x$  en  $A[]$ . Si no existe, entonces devuelve -1

# Divide y vencerás – Búsqueda binaria

## Ejemplo

Pre-requisitos:

- El array debe tomar valores únicos
- El array debe de estar ordenado de manera ascendente

If searching for 23 in the 10-element array:



## Divide y vencerás – Búsqueda binaria

---

```
public static int searchBinary(int A[], int x) {  
    if (A==null || A.length==0) {  
        System.out.println("Error: array is empty");  
        return -1;  
    }  
    return searchBinary(A, 0, A.length-1, x);  
}
```

---

## Divide y vencerás – Búsqueda binaria

---

```
public static int searchBinary(int A[], int start, int end, int x) {  
    if (start>end || start<0 || end>=A.length) {  
        System.out.println("Error: indexed out of range");  
        return -1;  
    }  
    if (start==end) {  
        if (A[start]==x) return start;  
        else return -1;  
    } else {  
        int m = start + (end - start)/2;  
        if (x==A[m]) return m;  
        else if (x<A[m]) return searchBinary(A,start,m-1,x);  
        else return searchBinary(A,m+1,end,x);  
    }  
}
```

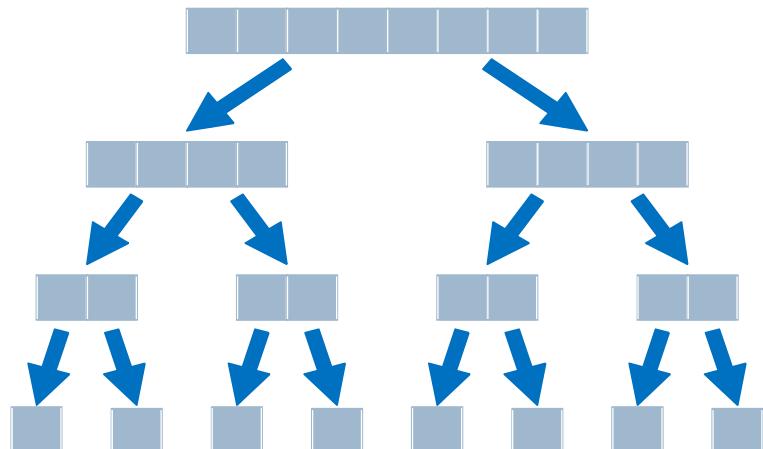
---

## Divide y vencerás – Encontrar máximo

A: 

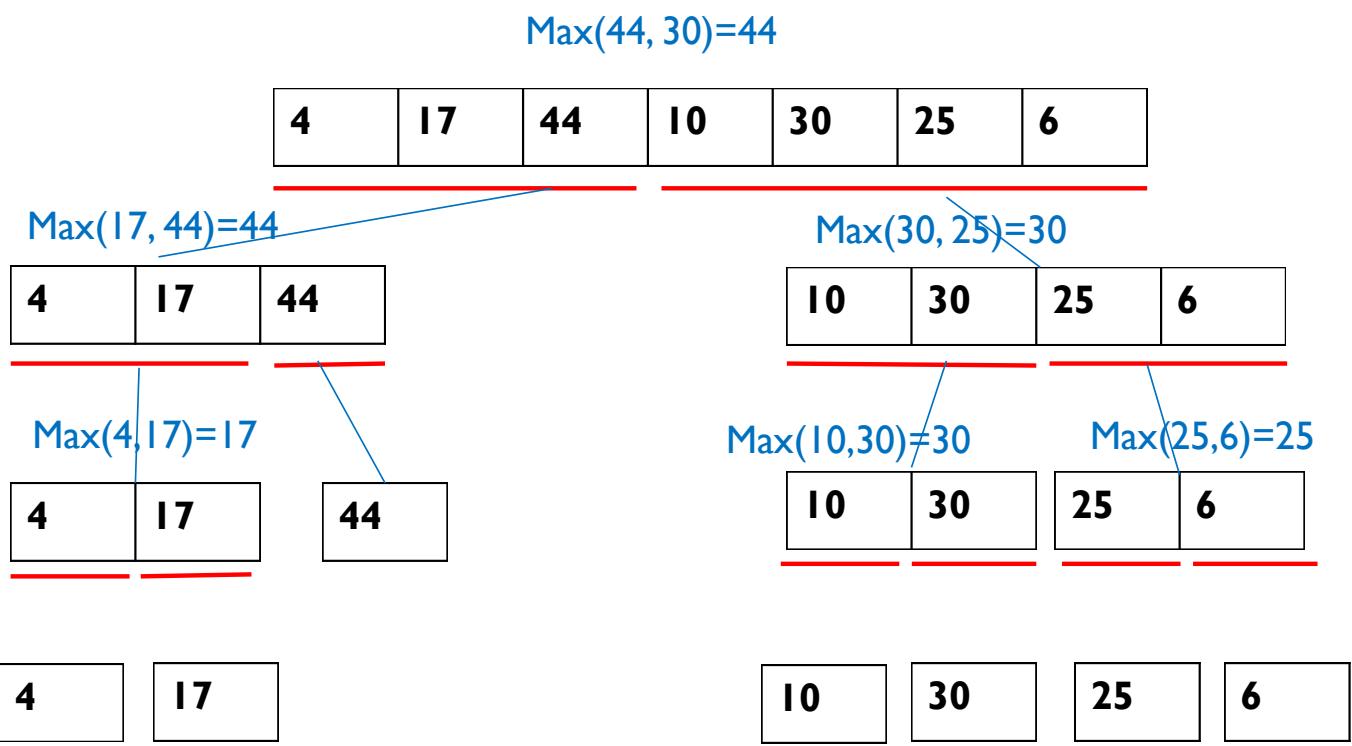
-3	24	11	-13	...	33	-7	12	1
----	----	----	-----	-----	----	----	----	---

1. Divide el array en dos partes
  2. Encontrar el máximo en cada parte
  3. Comparar ambos números y devolver del mayor.
- Es aplicado mediante llamadas recursivas



## Divide y vencerás – Encontrar máximo

### Ejemplo



# Divide y vencerás – Encontrar máximo

---

```
public static int findMax (int[] vector, int i, int j){  
    int med, max_left, max_right;  
    if (i==j)  
        return vector[i];   Si hay un único elemento, es el máximo  
    else  
        jmed = (i + j) / 2;  
        max_left = findMax (vector, i, med);      Divide a la mitad  
        max_right = findMax (vector, med+1, );  Máximo de cada mitad  
        if (max_left > max_right)  
            return max_left;  
        else  
            return max_right;  
}
```

# Divide y vencerás - Mergesort

---

- ▶ Merge-sort algoritmo: ordenar un array

## 1) **Dividir:**

- ▶ Dividir el array en dos (aproximadamente la mitad)
- ▶ Seguir dividiendo los arrays resultantes hasta que ya no pueda dividir mas (arrays de longitud uno)

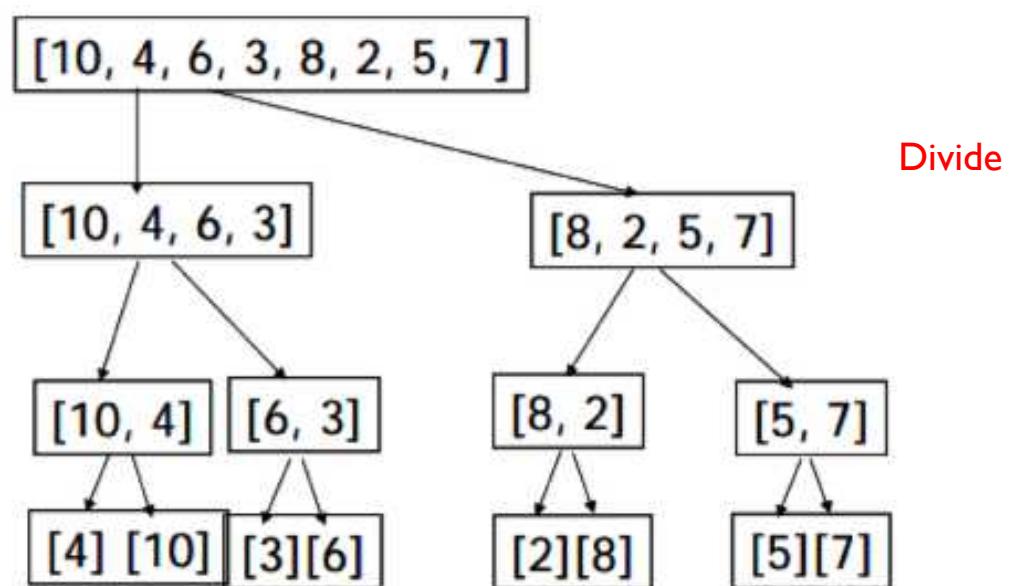
## 2) **Vencer:**

- Ordenar cada sub-array recursivamente
- Los arrays de longitud uno están ordenados

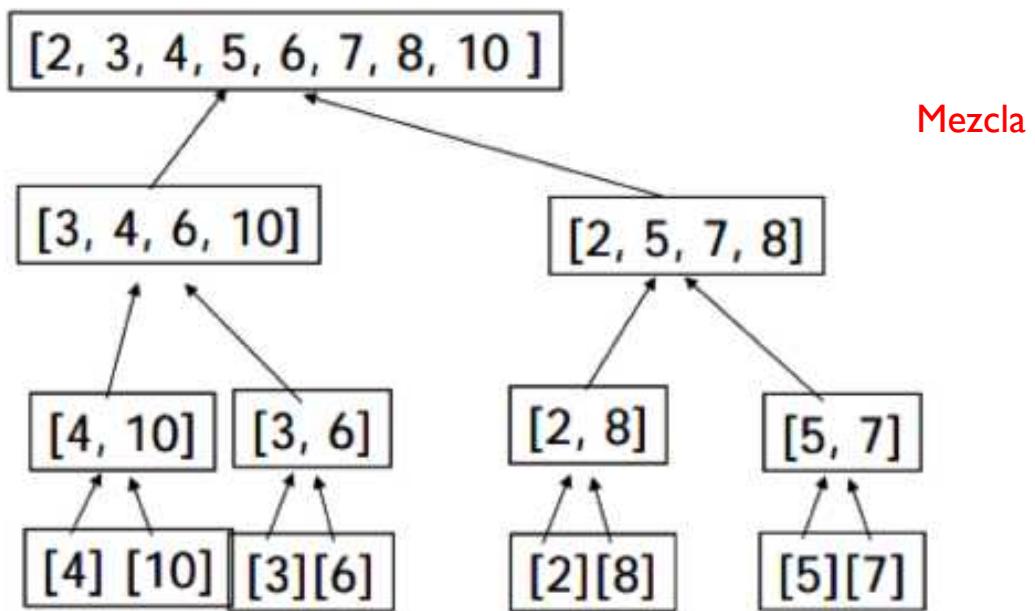
## 3) **Combinar:**

- A continuación, combinar los arrays adyacentes para formar un solo array ordenado.
- Repetir el proceso hasta que tengamos un único array ordenado

## Divide y vencerás – Mergesort Ejemplo

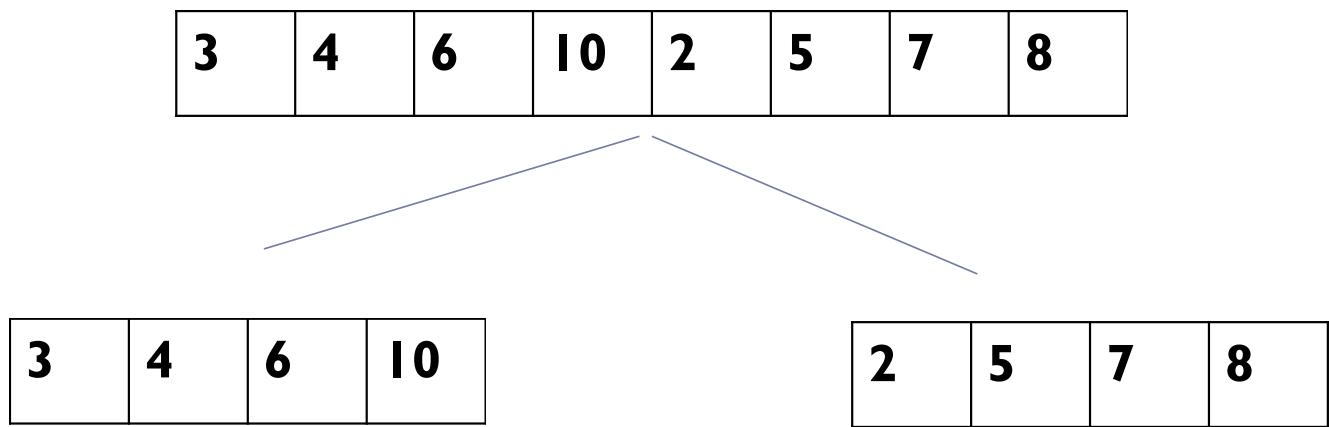


## Divide y vencerás – Mergesort Ejemplo



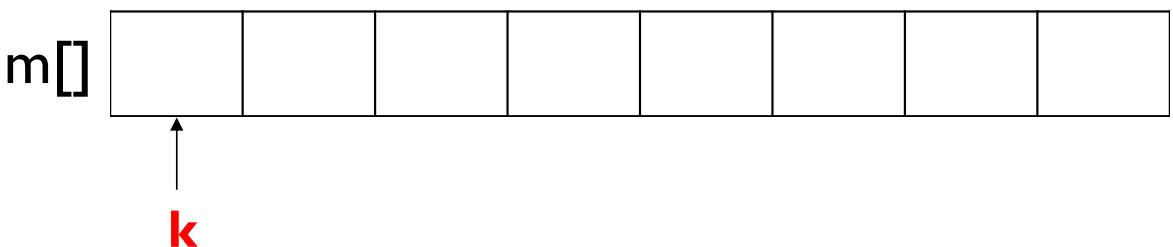
## Divide y vencerás – Mergesort Ejemplo

---



## Divide y vencerás – Mergesort Ejemplo

---



<b>3</b>	<b>4</b>	<b>6</b>	<b>10</b>
----------	----------	----------	-----------

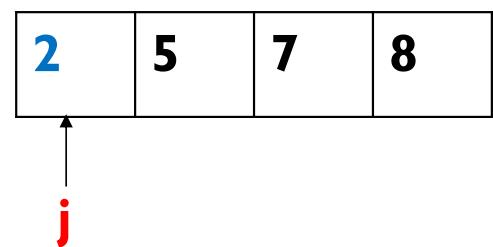
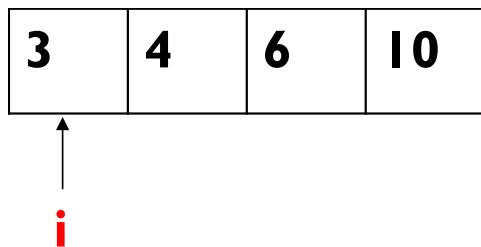
**i**

<b>2</b>	<b>5</b>	<b>7</b>	<b>8</b>
----------	----------	----------	----------

**j**

## Divide y vencerás – Mergesort Ejemplo

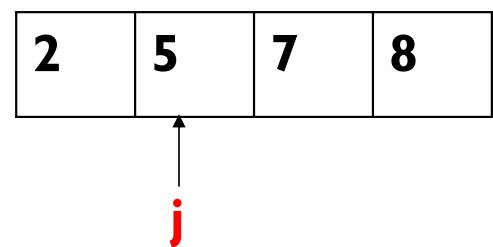
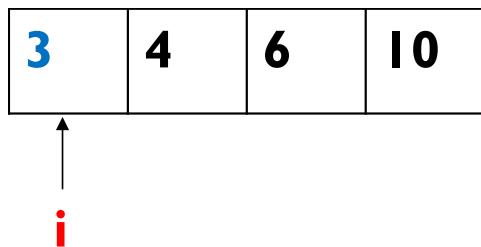
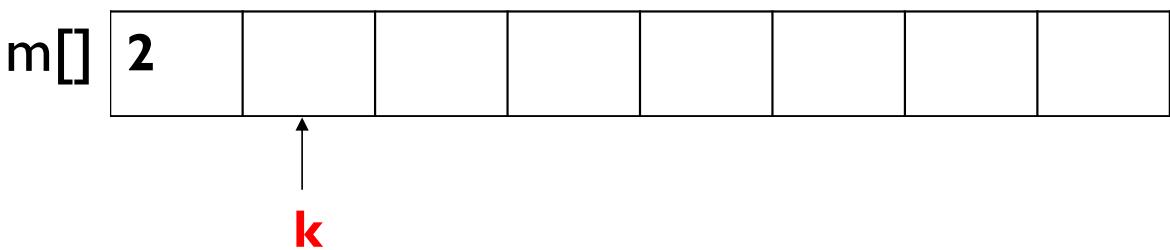
---



$2 < 3 \Rightarrow$  subo el 2 al array  $m[]$  y aumento índices  $j, k$

## Divide y vencerás – Mergesort Ejemplo

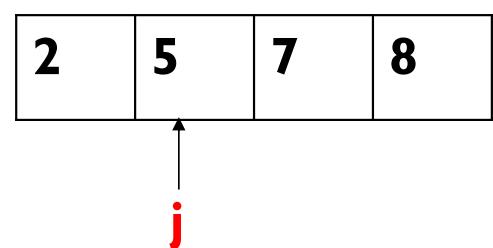
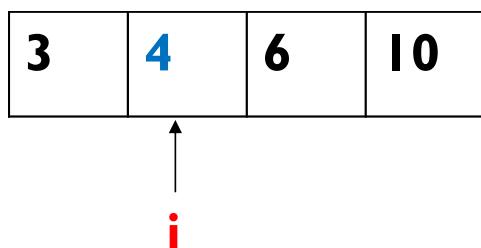
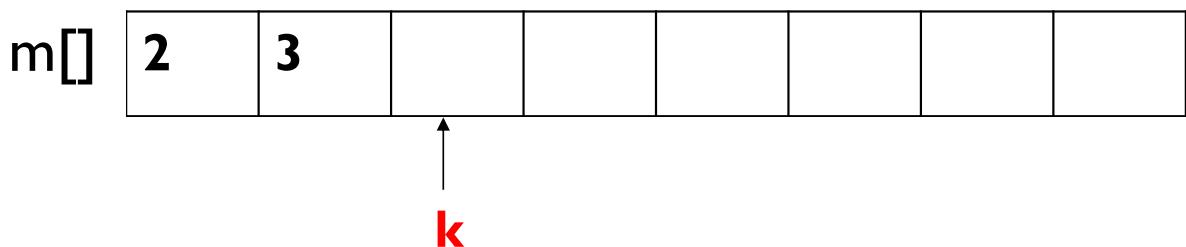
---



$3 < 5 \Rightarrow$  subo el 3 al array  $m[]$  y aumento índices i, k

## Divide y vencerás – Mergesort Ejemplo

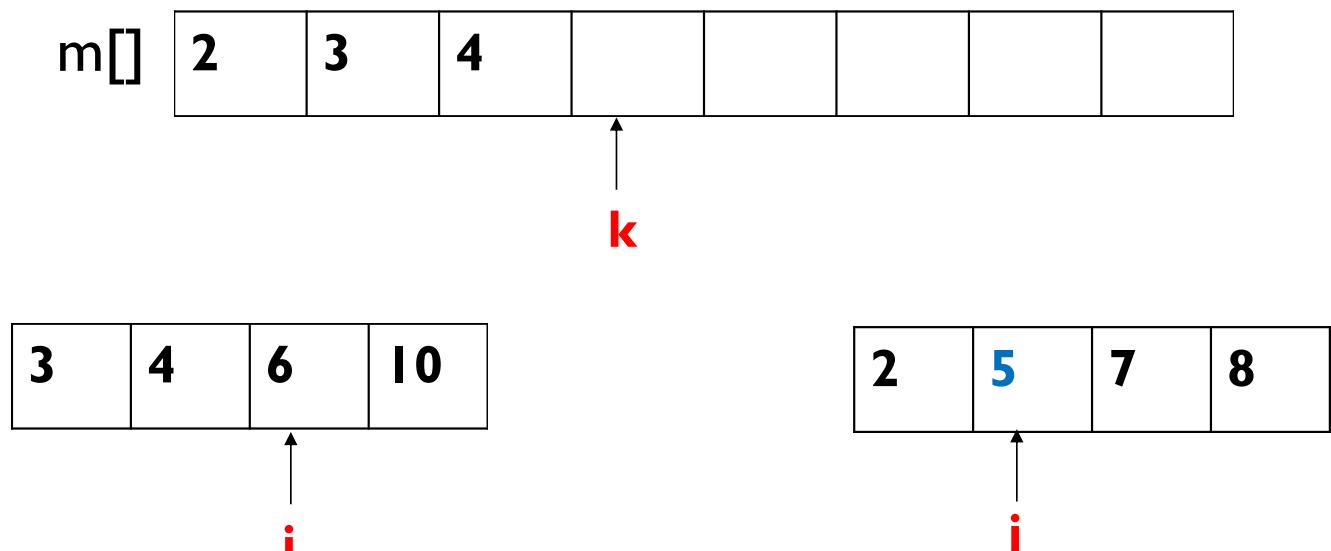
---



$4 < 5 \Rightarrow$  subo el 4 al array  $m[]$  y aumento índices i, k

## Divide y vencerás – Mergesort Ejemplo

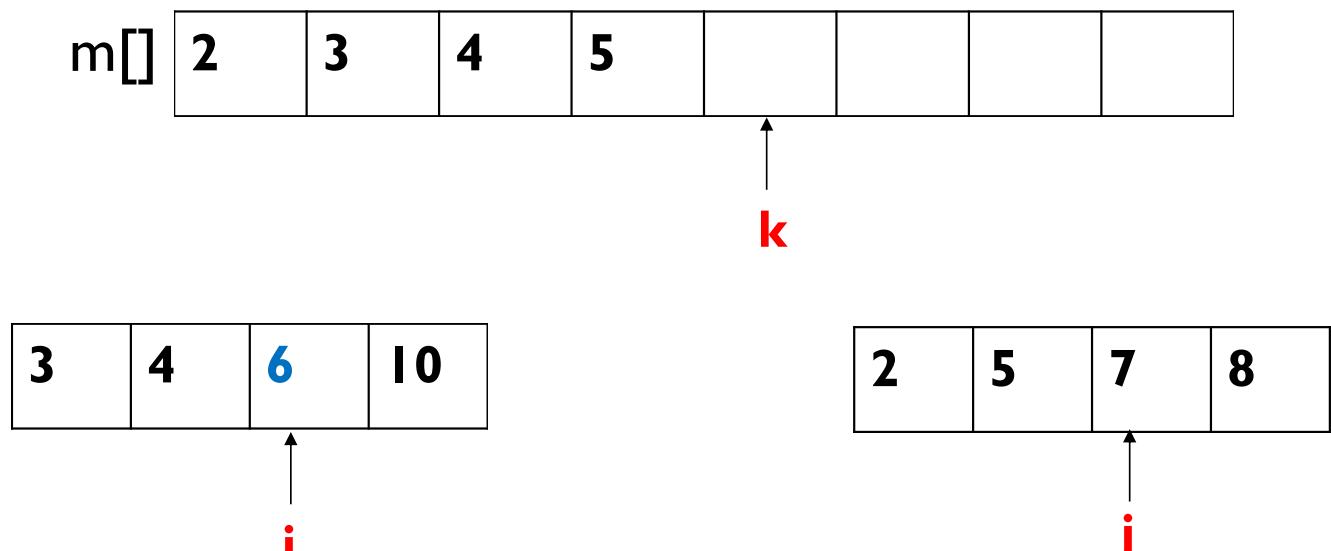
---



$5 < 6 \Rightarrow$  subo el 5 al array  $m[]$  y aumento índices  $j, k$

## Divide y vencerás – Mergesort Ejemplo

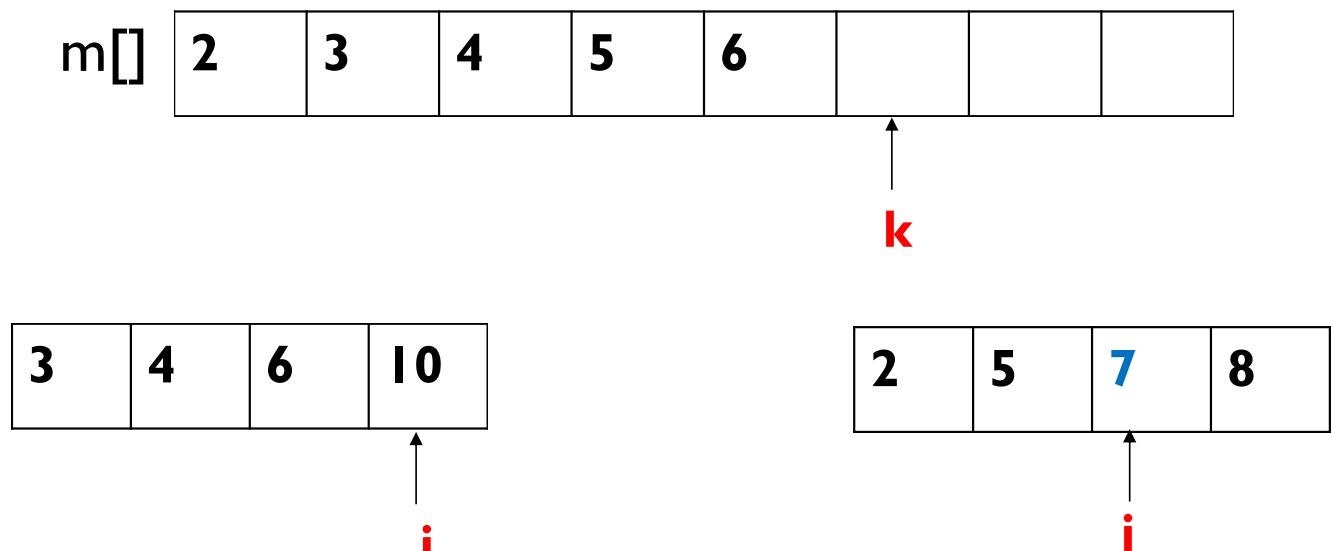
---



$6 < 7 \Rightarrow$  subo el 6 al array  $m[]$  y aumento índices i, k

## Divide y vencerás – Mergesort Ejemplo

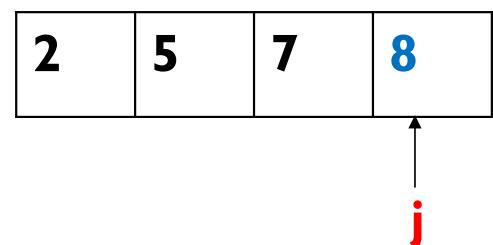
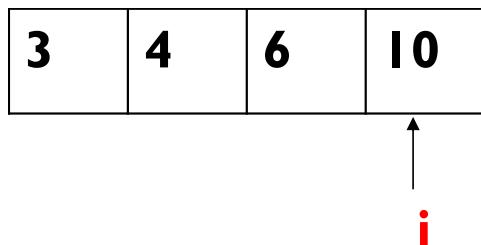
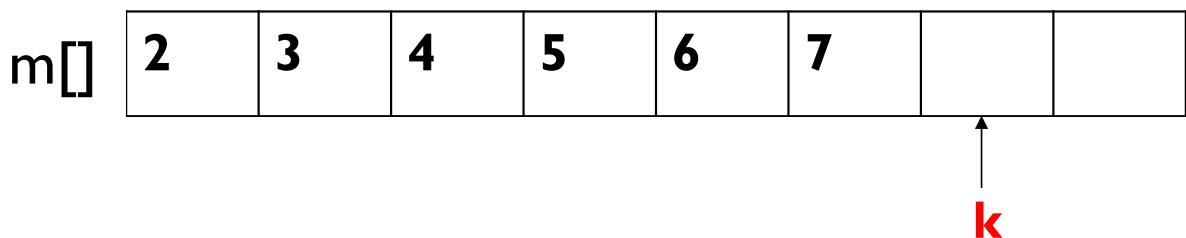
---



$7 < 10 \Rightarrow$  subo el 7 al array m[] y aumento índices j, k

## Divide y vencerás – Mergesort Ejemplo

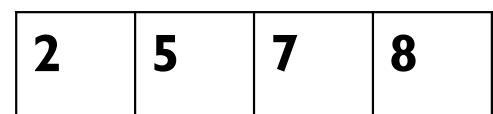
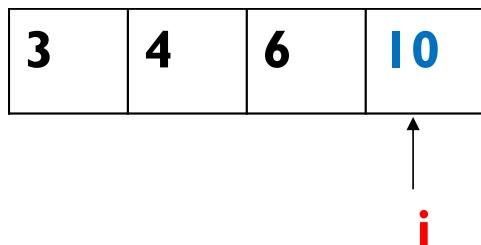
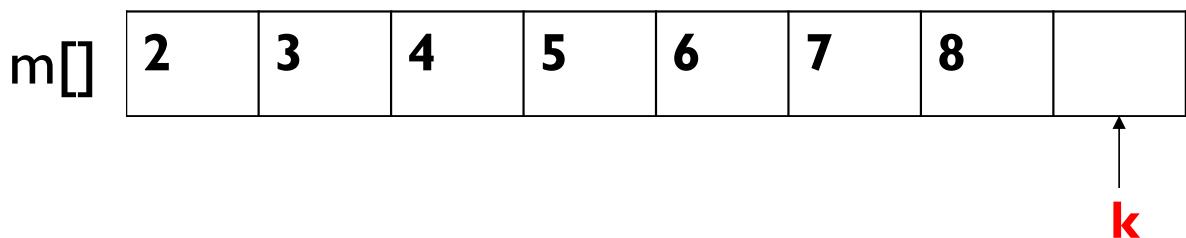
---



$8 < 10 \Rightarrow$  subo el 8 al array m[] y aumento índices j, k

## Divide y vencerás – Mergesort Ejemplo

---



subo el 10 al array  $m[]$  y aumento índices i, k

## Divide y vencerás – Mergesort Ejemplo

---

$m[]$

2	3	4	5	6	7	8	10
---	---	---	---	---	---	---	----

↑  
**k**

3	4	6	10
---	---	---	----

↑  
**i**

2	5	7	8
---	---	---	---

↑  
**j**

## Divide y vencerás – MergeSort Ejemplo

```
public static int[] mergesort(int a[]) {  
    if (a==null) {  
        System.out.println("array cannot be null!!!");  
        return null;  
    }  
    return mergesort(a,0,a.length-1);  
}  
  
public static int[] mergesort(int a[], int start, int end) {  
    if (start==end) return new int[]{a[start]};  
  
    int middle=(start+end)/2;  
    int[] m1=mergesort(a,start,middle);  
    int[] m2=mergesort(a,middle+1,end);  
    int[] m=merge(m1,m2);  
    return m;                                Divide al vector en dos partes  
}                                              iguales, y se realiza la llamada  
                                                 recursiva a un método merge que  
                                                 seguirá dividiendo y mezcla
```

# Divide y vencerás - Mergesort

Obtiene un array m[] ordenado a partir de dos sub-array ordenados a[] y b[]

```
//merges the arrays
public static int[] merge(int a[],int b[]) {
    if (a==null || b==null) {
        System.out.println("arrays cannot be null!!!!");
        return null;
    }
    int m[]=new int[a.length+b.length];
    int k=0;
    int i=0;
    int j=0;
    while (i<a.length && j<b.length) {
        if (a[i]<b[j]) {
            m[k]=a[i];
            i++;
        } else {
            m[k]=b[j];
            j++;
        }
        k++;
    }
    while (i<a.length) {
        m[k]=a[i];
        i++;
        k++;
    }
    while (j<b.length) {
        m[k]=b[j];
        j++;
        k++;
    }
    return m;
}
```

mezcla

Ya se ha recorrido b[], añadimos a m[], los elementos de a[] que están ordenados

Ya se ha recorrido a[] añadimos a m[] los elementos de b[] que están ordenados

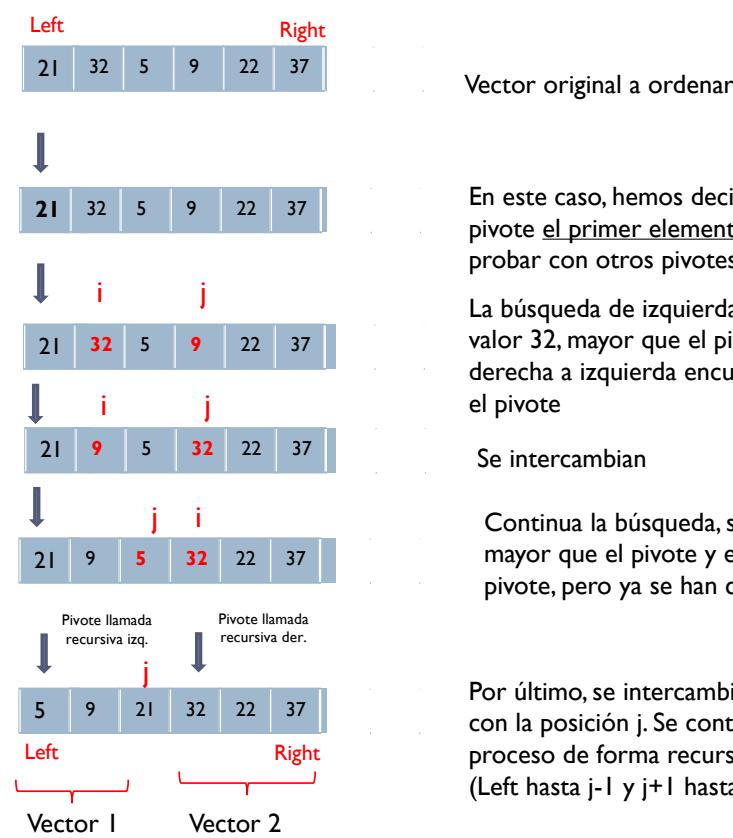
## Divide y vencerás – Quicksort

---

- ▶ Es el algoritmo de ordenación más rápido.
- ▶ Se basa en la técnica **divide y vencerás**. Consiste en dividir el array en arrays más pequeños, y ordenar éstos.
  - ▶ Se toma un valor del array como **pivote**, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha.
  - ▶ A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.
- ▶ Después de elegir el pivote se realizan dos búsquedas:
  - ▶ Izquierda a derecha, buscando un elemento mayor que el pivote
  - ▶ Derecha a izquierda, buscando un elemento menor que el pivote.
- ▶ Cuando se han encontrado los dos elementos anteriores, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.

# Divide y vencerás – Quicksort

## Ejemplo 1



# Divide y vencerás – Quicksort

## Ejemplo 1

### ARRAY DESORDENADO

- ▶ **vector[0]: 21 vector[1]: 32 vector[2]: 5 vector[3]: 9 vector[4]: 22 vector[5]: 37**
  - ▶ **Pivot:21 left: 0 right: 5**
- ▶ **vector[0]: 5 vector[1]: 9 vector[2]: 21 vector[3]: 32 vector[4]: 22 vector[5]: 37**
  - ▶ **Pivot:5 left: 0 right: 1**
- ▶ **vector[0]: 5 vector[1]: 9 vector[2]: 21 vector[3]: 32 vector[4]: 22 vector[5]: 37**
  - ▶ **Pivot:32 left: 3 right: 5**
- ▶ **vector[0]: 5 vector[1]: 9 vector[2]: 21 vector[3]: 22 vector[4]: 32 vector[5]: 37**

Llamada inicial

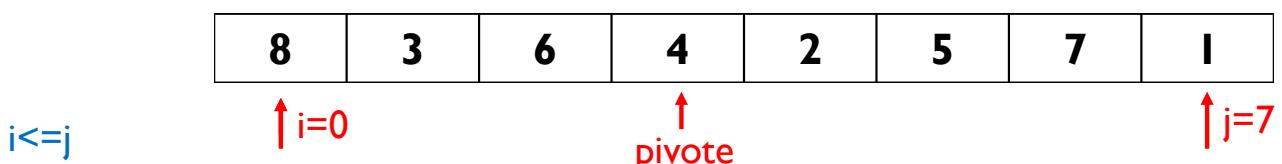
Llamada recursiva izq.

Llamada recursiva der.

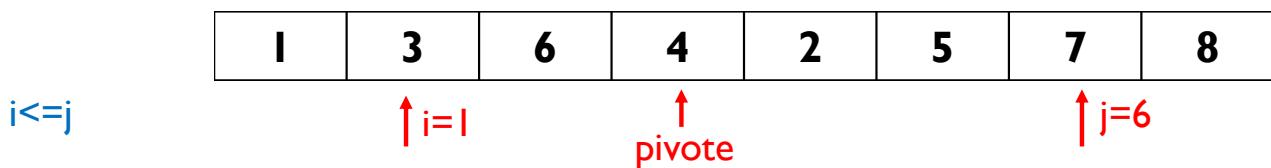
Ordenación final

### ARRAY ORDENADO

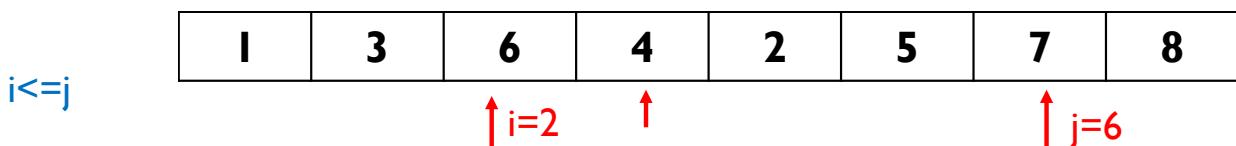
## Divide y vencerás – Quicksort Ejemplo 2



8 no es menor 4, y 1 no es mayor que 4 => intercambio e incremento índices i,j



3 es menor 4,=> incremento i



6 no es menor 4 (no incremento i)

7 es mayor que 4=>decremento j

## Divide y vencerás – Quicksort Ejemplo 2

I	3	6	4	2	5	7	8
i<=j		↑ i=2	↑ pivot		↑ j=5		

5 es mayor que 4=>decremento j

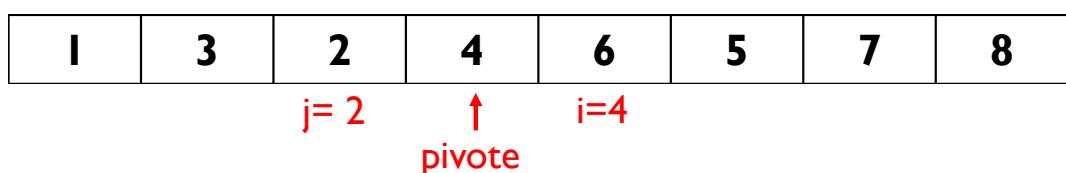
I	3	6	4	2	5	7	8
i<=j		↑ i=2	↑ pivot		↑ j=4		

2 no es mayor que 4=>intercambio 6 y 2, e incremento i y decremento j

I	3	2	4	6	5	7	8
i<=j		↑ i=3	↑↑ pivot	↑ j=3			

4 no es menor ni mayor que 4=>intercambio 4 y 4, e incremento i y decremento j

## Divide y vencerás – Quicksort Ejemplo 2



i>j => llamada recursiva a array [0,j] y [i, 7]



# Divide y vencerás – Quicksort

```
public static void quicksort(int a[]) {  
    doPartition(a,0,a.length-1);  
}
```

```
    public static void doPartition(int a[],int start, int end) {  
        if (a==null) {  
            System.out.println("array cannot be null!!!!");  
            return;  
        }  
        if (start>=end) return;  
        int middle = start + (end- start) / 2;  
        int pivate = a[middle];  
        System.out.println("pivate:"+pivate);  
        int i=start;  
        int j=end;  
  
        while (i<=j) {  
  
            while (a[i]<pivate) i++;  
            while (a[j]>pivate) j--;  
  
            if (i<=j) {  
                int x=a[i];  
                a[i]=a[j];  
                a[j]=x;  
                i++;  
                j--;  
            }  
        }  
        if (start<j) doPartition(a,start,j);  
        if (i<end) doPartition(a,i,end);  
    }  
}
```

# Resumen

---

- ▶ **Se divide el problema original en sub-problemas**
  - ▶ Recursivamente, cada sub-problema se divide de nuevo
- ▶ **Cuando el caso a resolver es lo suficientemente sencillo se resuelve utilizando un algoritmo directo (no recursivo)**
  - ▶ El algoritmo directo debe ser eficiente para problemas sencillos
  - ▶ No importa que no lo sea para problemas grandes
- ▶ **Cada sub-problema se resuelve de forma independiente**
- ▶ **Finalmente se combinan las soluciones de todos los sub-problemas para formar la solución del problema original**

## Unit 7. An overview of algorithm strategies

Data Structures and Algorithms (DSA)

# Some concepts

---

- ▶ An **algorithm** is a well-defined and finite sequence of steps used to solve a well-defined problem.
- ▶ **Algorithm strategy**
  - ▶ Approach to solving a problem
  - ▶ May combine several approaches
- ▶ **Algorithm structure:**
  - ▶ **Iterative:** uses a loop to find the solution
  - ▶ **Recursive:** a function calling itself

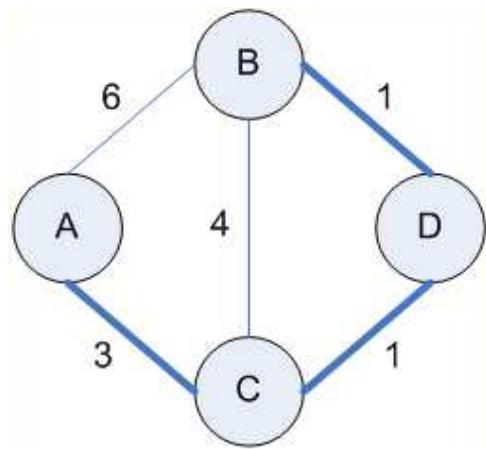
# Problem type

## ► **Satisfying**

- ▶ Find any satisfactory solution
- ▶ Ex: Find a path from A to E

## ► **Optimization**

- ▶ Find the best solution
- ▶ Ex: Find the shortest path from A to E



This example was taken from [http://cs.smu.ca/~porter/csc/common\\_341\\_342/notes/graphs\\_shortest\\_path.html](http://cs.smu.ca/~porter/csc/common_341_342/notes/graphs_shortest_path.html)

# Main Algorithm Strategies

---

- ▶ Recursive algorithms
- ▶ Divide and Conquer algorithms
- ▶ **Backtracking algorithms**
- ▶ Dynamic programming algorithms
- ▶ Greedy algorithms
- ▶ Brute force algorithms
- ▶ Branch and bound algorithms
- ▶ Heuristic algorithms

Heuristics and  
Optimization,  
Course 3<sup>a</sup>,  
Semester I°

## Get out of the labyrinth

---



- ▶ We are in a labyrinth, full of crossings, detours and somebody chasing us.
  - ▶ There are brute force solutions (e.g. always follow the wall at our left) but they are inefficient.
  - ▶ We have to create a path of decisions (right-left-straight-right-straight...) to reach the exit but skipping paths that, without following them, it is known that they are not driving us to the exit.
  - ▶ If we take a bad decision, we can go back and try a different path.
-

# Constraints

---

- ▶ Sometimes, there are no specific algorithms to solve a problem → All possible **solutions** (not decisions) must be explored.
- ▶ The solution (aprtial or global) is a vector of cases, decisions, values, etc., with a finite length.
- ▶ There is a way to know if a solution is global or partial and, hence, the algorithm must go to an end (attention to infinite loops must be paid).
- ▶ Completeness: Given a partial solution, it is possible to say if it is part of a global solution (it is “complete”) or not.

## Back Tracking

---

- ▶ General philosophy to solve problems also used in many other scopes.
  - ▶ Technique to solve problems based on exploring possible partial solutions.
  - ▶ Each partial solution is extended with more ‘complete’ solutions.
  - ▶ Partial solutions are evaluated using brute force approaches.
  - ▶ When a solution is ‘complete’, it is called ‘k-promising solution’ where k is the level in that execution point in the algorithm.
  - ▶ When a set of solutions is not ‘complete’, it is discarded..
  - ▶ Implementing these algorithms requires **recursion**.

## Five core methods

---

1. **exploreLevel(k)**: The algorithm goes one step deeper in the set of possible solutions.
2. **pendingOptions()**: Checking if there are pending options to be explored at the current level.
3. **completeSolution()**: Checking if the current solution is complete or global.
4. **processSolution()**: to show the solution.
5. **completeness()**: To evaluate if a k-promising vector can be completed.

## 3 stages method

### Start the algorithm at k level

The algorithm analyses the next level  $k$ , starting at the first level.



### Check options at k level

If there are  $n$  solutions that can be completed, the algorithm goes to the  $k+1$  level for each option

Otherwise the path is discarded



### Finalize

If there are no more options for a path, is it a global solution (return true) or not (return false)?

## When to apply backtracking

---

- ▶ It is a brute force algorithm (discarding some options), so:
  - ▶ Used when there are no more appropriate solutions
  - ▶ The solutions tree is finite and there are no loops (or loops can be avoided)
- ▶ In general:
  - ▶ The problem can be solved taking decisions at each level, so
    - ▶ The algorithm allows for a recursive design
    - ▶ It is possible to discard candidates
  - ▶ IMPORTANT: take care of memory (each recursive call requires room for all local variables)

# Designing the algorithm

---

- ▶ **Recursive algorithm**
    - ▶ At each level, the candidate solution is checked to know if it is global
    - ▶ If it is not, complete candidate solutions are evaluated and the algorithm runs over all of them
  - ▶ **Global solution decision**
    - ▶ Evaluate if a vector is a solution to the problem
  - ▶ **Completeness**
    - ▶ Evaluates if a solution “seems to be valid” or must be discarded. The completeness criterion must be determinist (“true” or “false”). E.g. If the solution must not have repeated numbers the candidate vector [3,5,1,1] can be discarded
  - ▶ **Are we looking for a solution? Or do we want all solutions?**
    - ▶ We can finish the algorithm with the first solution found or explore the k-promising remaining candidate solutions
- 
- ▶ //

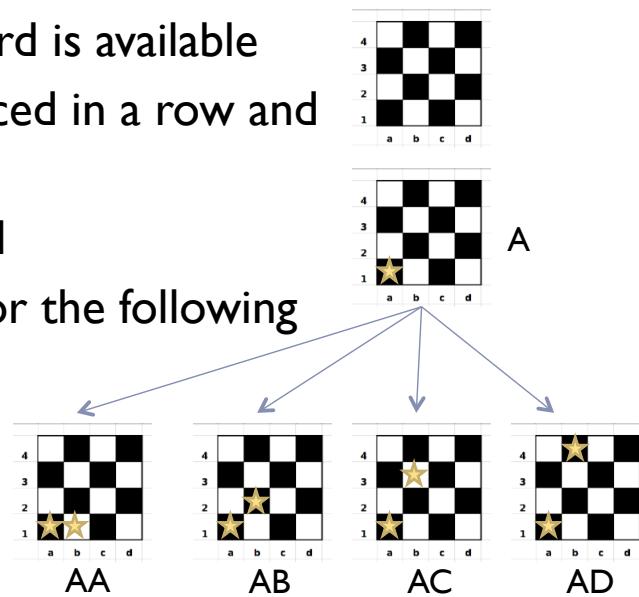
# Back tracking algorithm

## Pseudocode

```
VA(x: sequence, k: level) {  
    exploreLevel(k) 1. Search for promising candidates  
    While pendingOptions(k){  
        extend x with option vi 2. Detect global solution  
        if ({x,vi} is a global solution) {  
            processSolution()  
        } 3. Detect if it is a k-promising candidate solution  
        else {  
            if(completeness(({x,vi}))  
                VA({x,vi},k+1) 4. Launch the algorithm with the next level  
            }  
        }  
    }  
}
```

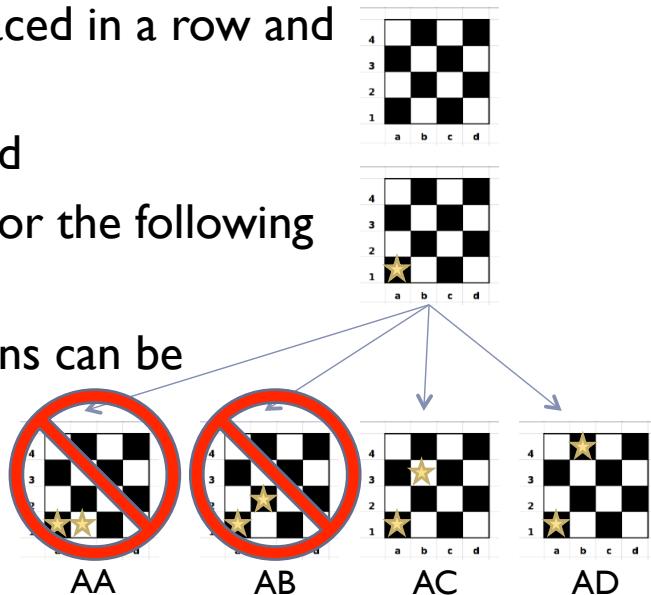
Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

- ▶ An empty 4x4 chessboard is available
- ▶ Each queen must be placed in a row and a column
- ▶ The first queen is placed
- ▶ Four different options for the following queen



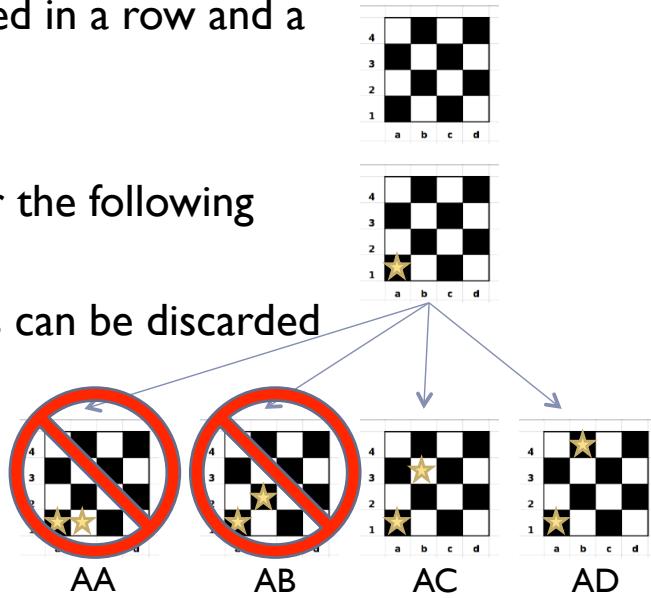
Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

- ▶ An empty 4x4 chessboard is available
- ▶ Each queen must be placed in a row and a column
- ▶ The first queen is placed
- ▶ Four different options for the following queen
- ▶ Some candidate solutions can be discarded



Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

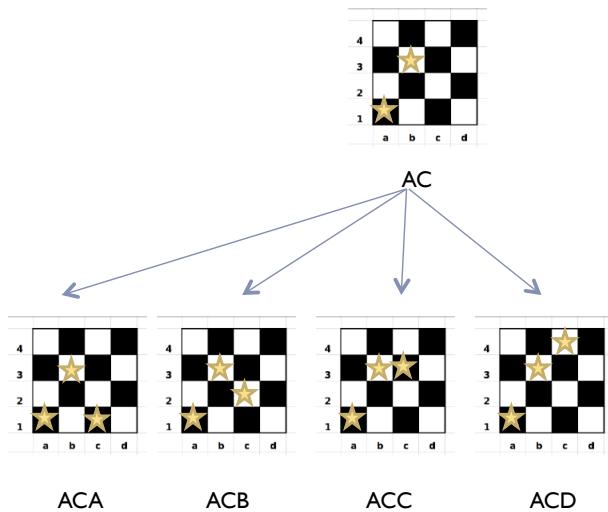
- ▶ An empty 4x4 chessboard is available
- ▶ Each queen must be placed in a row and a column
- ▶ The first queen is placed
- ▶ Four different options for the following queen
- ▶ Some candidate solutions can be discarded



- ▶ Solutions under AA and AB are not further considered, they are not valid (not complete)

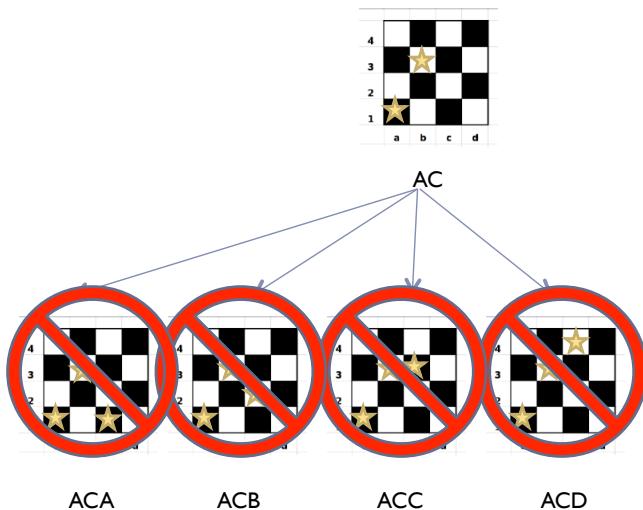
Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

► Let's develop AC candidate:



Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

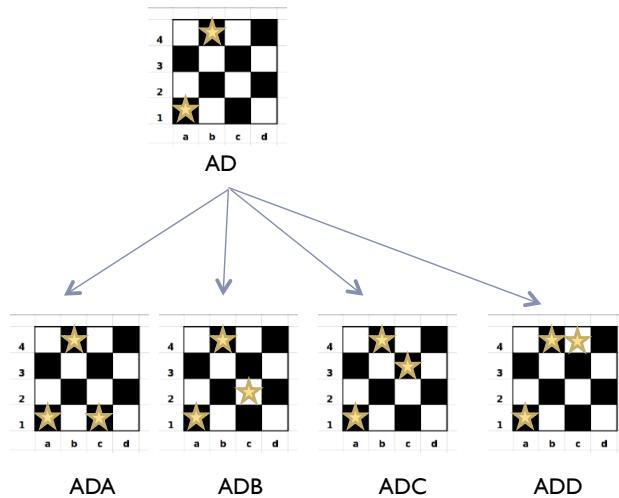
- ▶ Let's develop AC candidate:



- ▶ No valid candidates!! All options discarded ... so?

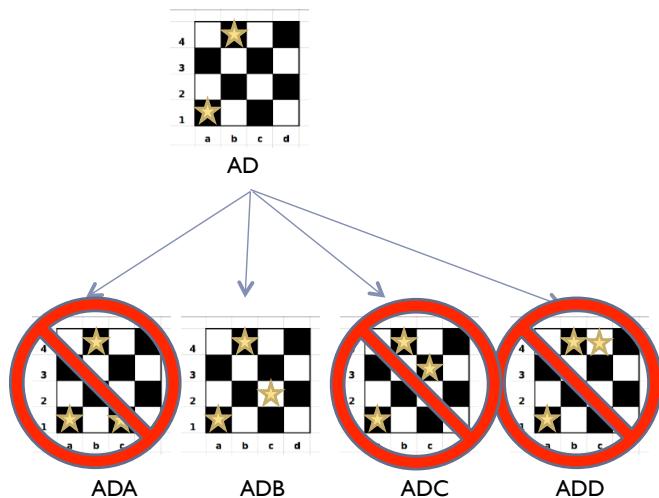
Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

- ▶ Back tracking!!, go back to AD candidate:



Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

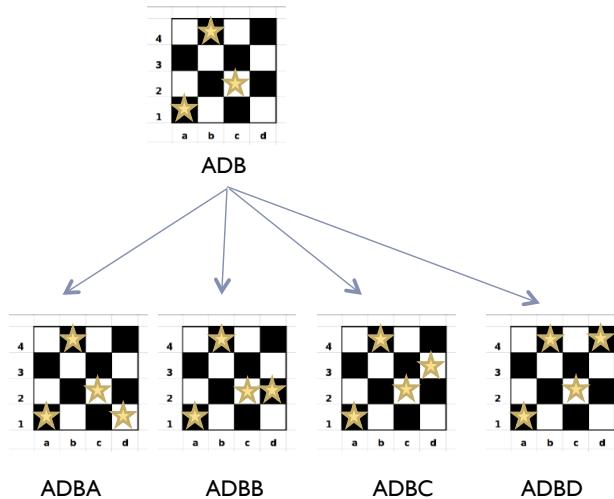
- ▶ Back tracking!!, go back to AD candidate:



- ▶ But some candidates can be discarded

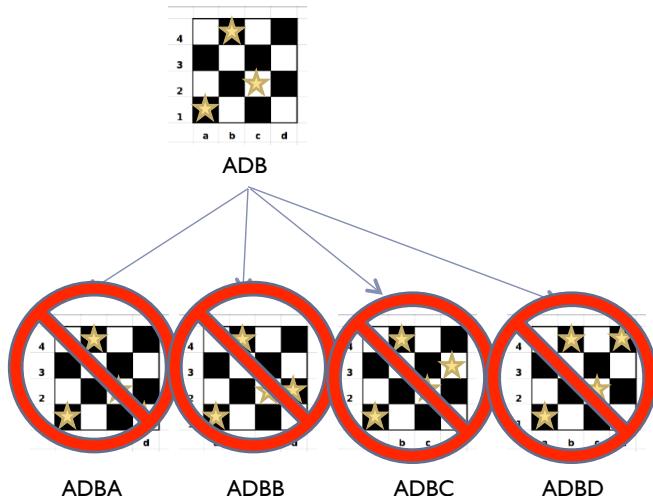
Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

► Let's develop ADB:



Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

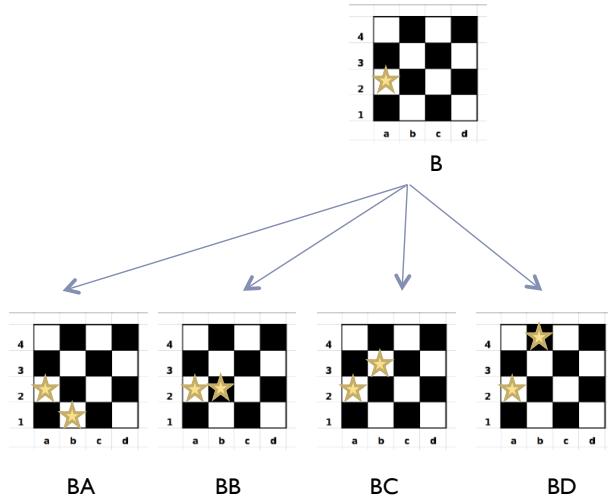
► Let's develop ADB:



► All candidates discarded!! Is there a solution?

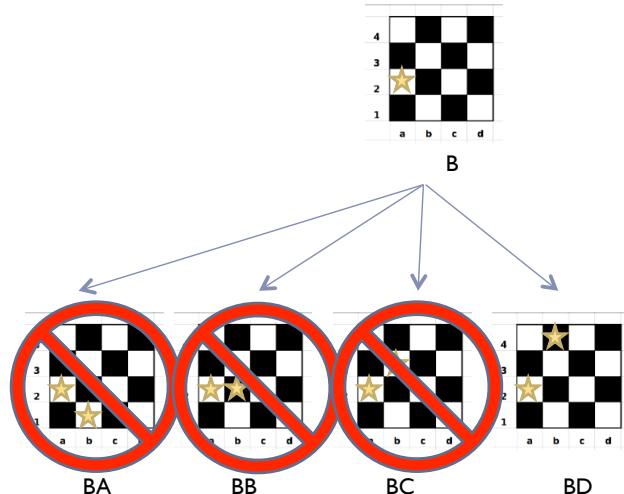
Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

- ▶ Back tracking!! Try another position for first queen:



Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

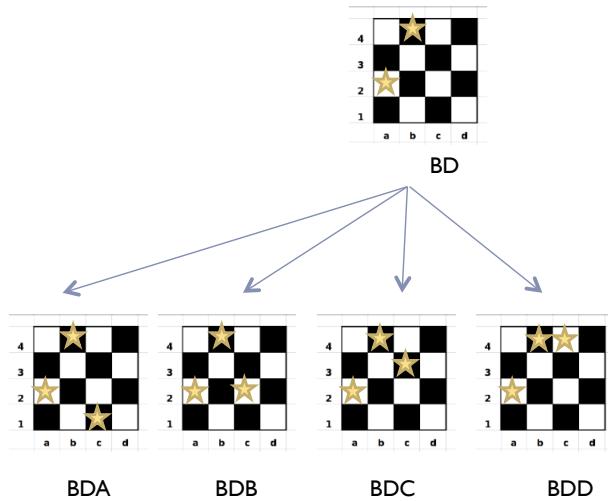
- ▶ Back tracking!! Try another position for first queen:



- ▶ Not valid candidates are discarded

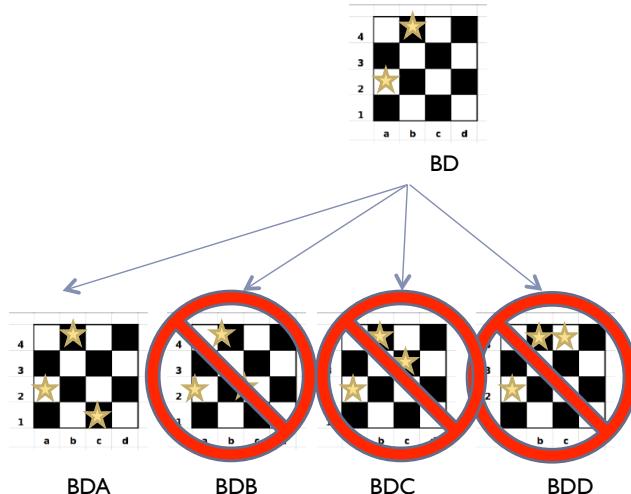
Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

► Let's evaluate BD candidate:



Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

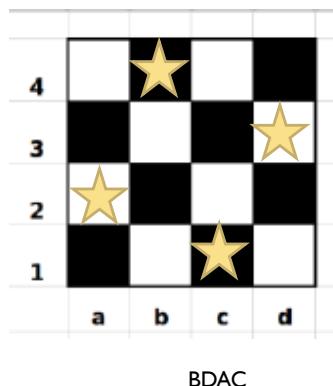
- ▶ Let's evaluate BD candidate:



- ▶ Not valid candidates are discarded

Example. Placing 4 queens in a 4x4 chessboard so that no queen attacks any other

► And so on ...

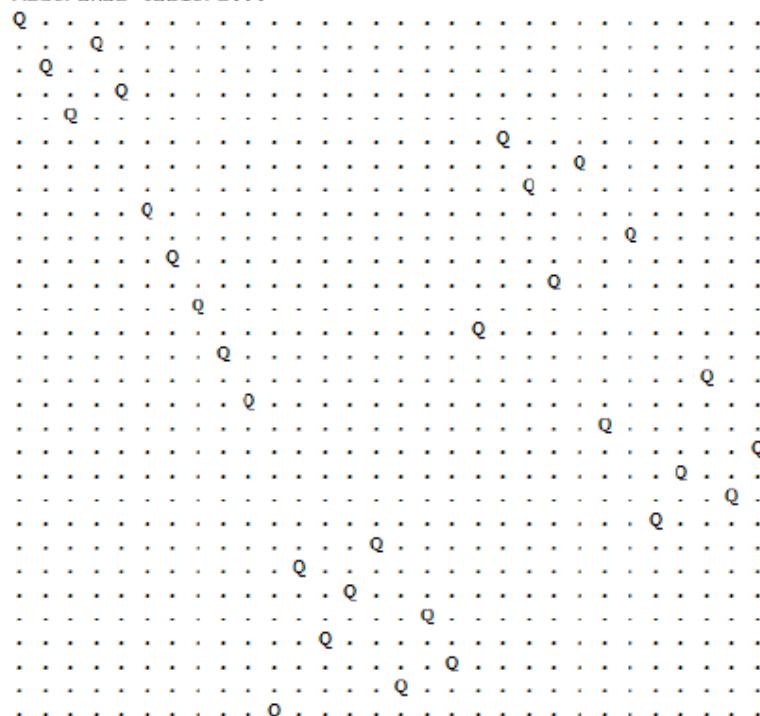


BDAC

► 26

Example. Placing 30 queens in a 30x30 chessboard so that no queen attacks any other

```
Queens resuelto en 19237575 iteraciones. Fin  
Mostrando tablero...
```



## Summary Backtracking

---

- ▶ For problems where the solution can be seen as a ‘data sequence’
  - ▶ The solution is a set of decisions among several possible candidates
  - ▶ All possible cases constitute a set that can be seen as a decision tree (a tree with conditions in internal nodes and candidates at leaves)
- ▶ At each decision, a recursive call is done and every k-promising candidate is tested
- ▶ It is possible to have clear criteria to assign to tree nodes
  - ▶ K-promising partial solutions must be evaluated as true or false
- ▶ Clear criteria to finalize the algorithm
  - ▶ Do we want all solutions or only one?
  - ▶ Identify global solution
  - ▶ Avoid infinite loops (of course)

## Backtracking applications

---

- ▶ Graphs painting
- ▶ Hamiltonian path
- ▶ Combinatorial optimizations (knapsack problem...)
- ▶ Labyrinth resolution
- ▶ Prisoner's dilemma
- ▶ Resources management,
- ▶ Chess, dominoes, cards games, scrabble, ... and  
**SUDOKUS** (the next weekly work)

# Main Algorithm Strategies

---

- ▶ Recursive algorithms
- ▶ Divide and Conquer algorithms
- ▶ Backtracking algorithms
- ▶ **Dynamic programming algorithms**
- ▶ Greedy algorithms
- ▶ Brute force algorithms
- ▶ Branch and bound algorithms
- ▶ Heuristic algorithms

Heuristics and  
Optimization,  
Course 3<sup>a</sup>,  
Semester I°

# Dynamic Programming Algorithm

---

- ▶ Based on remembering past results
- ▶ Approach:
  - ▶ Divide problem into smaller subproblems
    - ▶ Subproblems **must be of same type**
    - ▶ Subproblems **must overlap**
  - ▶ Solve each subproblem recursively
    - ▶ May simply look up solution (if previously solved)
  - ▶ Combine solutions to solve original problem
  - ▶ Store solution to problem
- ▶ For optimization problems.

```
// Fibonacci Series using Dynamic Programming
class fibonacci
{
    static int fib(int n)
    {
        /* Declare an array to store Fibonacci numbers.
        int f[] = new int[n+1];
        int i;

        /* 0th and 1st number of the series are 0 and 1*/
        f[0] = 0;
        f[1] = 1;

        for (i = 2; i <= n; i++)
        {
            /* Add the previous 2 numbers in the series
            and store it */
            f[i] = f[i-1] + f[i-2];
        }

        return f[n];
    }

    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}
```

# Divide and conquer vs Dynamic Programming

---

- ▶ Both paradigms divide the problem into subproblems, recursively solve them and combine their solutions.
- ▶ Choose Divide and Conquer when subproblems must be solved only once. For example: binary search o mergesort.
- ▶ Other wise, Dynamic Programming. For example: fibonacci.

# Main Algorithm Strategies

---

- ▶ Recursive algorithms
  - ▶ Divide and Conquer algorithms
  - ▶ Backtracking algorithms
  - ▶ Dynamic programming algorithms
  - ▶ **Greedy algorithms**
  - ▶ Brute force algorithms
  - ▶ Branch and bound algorithms
  - ▶ Heuristic algorithms
- 
- Heuristics and Optimization,  
Course 3<sup>a</sup>, Semester I°

---

These slides are based on the course CMSC 132's materials, University of Maryland

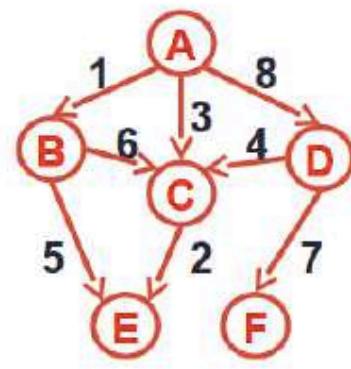
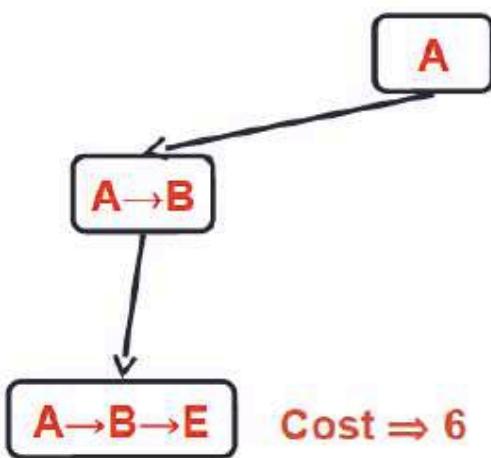
# Greedy Algorithm

---

- ▶ Based on trying best current (local) choice results
- ▶ Approach:
  - ▶ At each step of algorithm
  - ▶ Choose best local solution
- ▶ Avoid backtracking, exponential time  $O(2^n)$ .
- ▶ Hope local optimum lead to global optimum

# Greedy Algorithm

- Example (Shortest Path from A to E)
  - Choose lowest-cost neighbor



Does not obtain the global shortest path!!!

# Main Algorithm Strategies

---

- ▶ Recursive algorithms
- ▶ Divide and Conquer algorithms
- ▶ Backtracking algorithms
- ▶ Dynamic programming algorithms
- ▶ Greedy algorithms
- ▶ **Brute force algorithms**
- ▶ Branch and bound algorithms
- ▶ Heuristic algorithms

Heuristics and  
Optimization,  
Course 3<sup>a</sup>,  
Semester I°

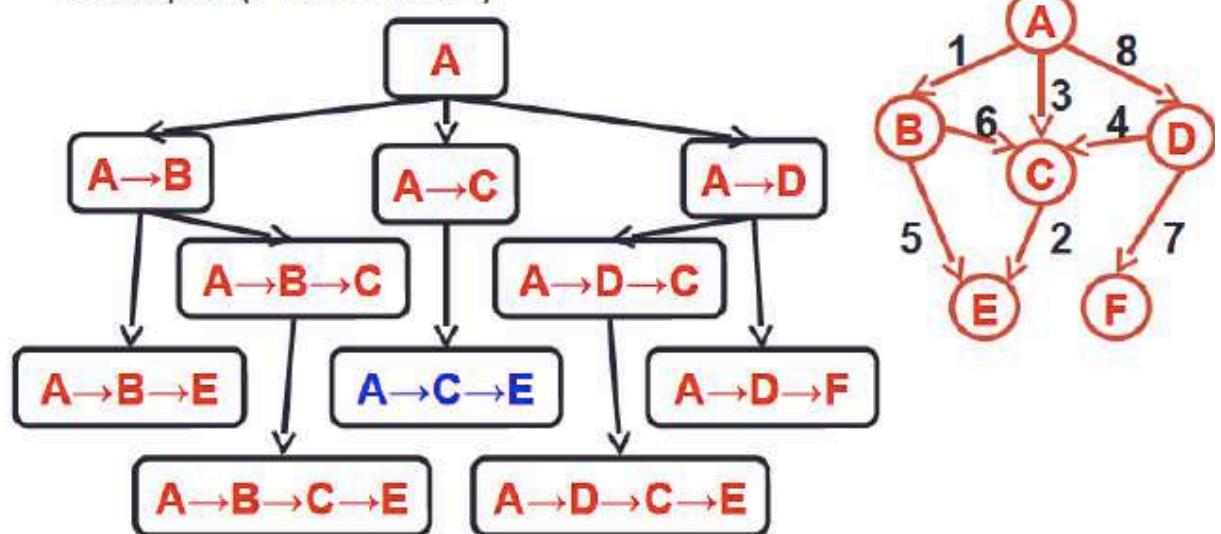
# Brute force Algorithm

---

- ▶ Based on trying all possible solutions
- ▶ Most expensive approach
- ▶ Approach:
  - ▶ Generate and evaluate possible solutions until
  - ▶ Best solution is found (if can be determined)
  - ▶ All possible solutions found
    - ▶ Return best solution
    - ▶ Return failure if no satisfactory solution

# Brute force Algorithm

- Example (From A to E)



# Main Algorithm Strategies

---

- ▶ Recursive algorithms
- ▶ Divide and Conquer algorithms
- ▶ Backtracking algorithms
- ▶ Dynamic programming algorithms
- ▶ Greedy algorithms
- ▶ Brute force algorithms
- ▶ **Branch and bound algorithms**
- ▶ Heuristic algorithms

Heuristics and  
Optimization,  
Course 3<sup>a</sup>,  
Semester I°

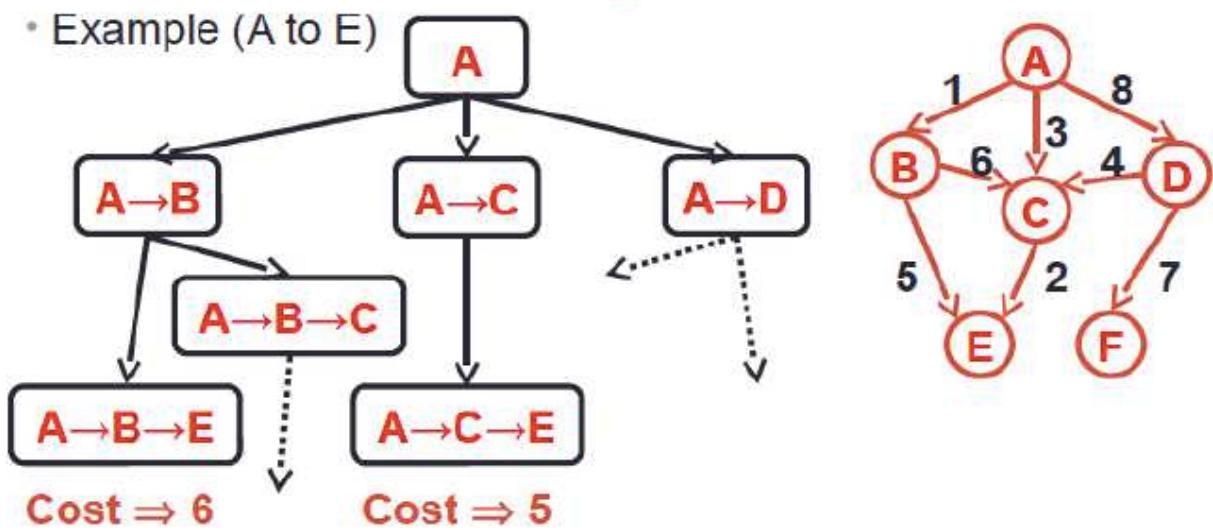
## Branch and bound Algorithm

---

- ▶ Based on limiting search using current solution
- ▶ Approach
  - ▶ Track best current solution found
  - ▶ Eliminate (prune) partial solutions that can not improve upon best current solution
- ▶ Reduces amount of backtracking
- ▶ Not guaranteed to avoid exponential time

## Branch and bound Algorithm

- Example (A to E)



# Main Algorithm Strategies

---

- ▶ Recursive algorithms
- ▶ Divide and Conquer algorithms
- ▶ Backtracking algorithms
- ▶ Dynamic programming algorithms
- ▶ Greedy algorithms
- ▶ Brute force algorithms
- ▶ Branch and bound algorithms
- ▶ **Heuristic algorithms**

Heuristics and  
Optimization,  
Course 3<sup>a</sup>,  
Semester I°

# Heuristic Algorithm

---

- ▶ Based on trying to guide search for solution
- ▶ Heuristic => “rule of thumb”
- ▶ Approach
  - ▶ Generate and evaluate possible solutions
    - ▶ Using “rule of thumb”
    - ▶ Stop if satisfactory solution is found
- ▶ Can reduce complexity
- ▶ Not guaranteed to yield best solution

# Heuristic Algorithm

- Example (A to E)

- Try only edges with cost < 5

