Programación de sistemas

Árboles

Julio Villena Román
<jvillena@it.uc3m.es>

MATERIALES CREADOS EN EL TRABAJO DE DIFERENTES AUTORES:
Carlos Delgado Kloos, M.Carmen Fernández Panadero,
Raquel M.Crespo García, Carlos Alario Hoyos

Contenidos

- **Concepto de árbol**
- **Terminología**
- **Implementación**
- Casos especiales
  - Árboles binarios de búsqueda
  - Montículos (*heaps*)

Cita

*"The structure of concepts is formally called a **hierarchy** and since ancient times has been a basic structure for all Western knowledge. Kingdoms, empires, churches, armies have all been structured into hierarchies. Tables of contents of reference material are so structured, mechanical assemblies, computer software, all scientific and technical knowledge is so structured..."*

Robert M. Pirsig:
Zen and the Art of Motorcycle Maintenance

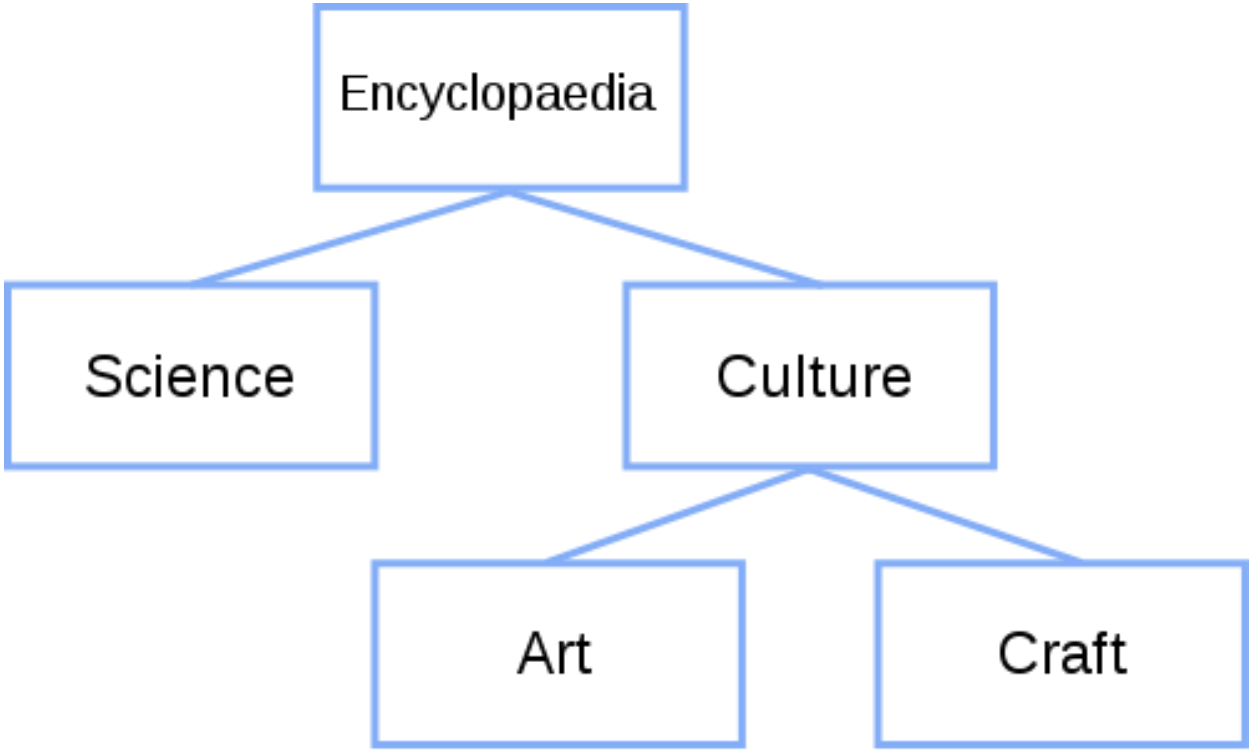
Concepto de árbol

Un **árbol** es una estructura de datos **no lineal** que almacena los elementos **jerárquicamente**

(generalización de las listas)

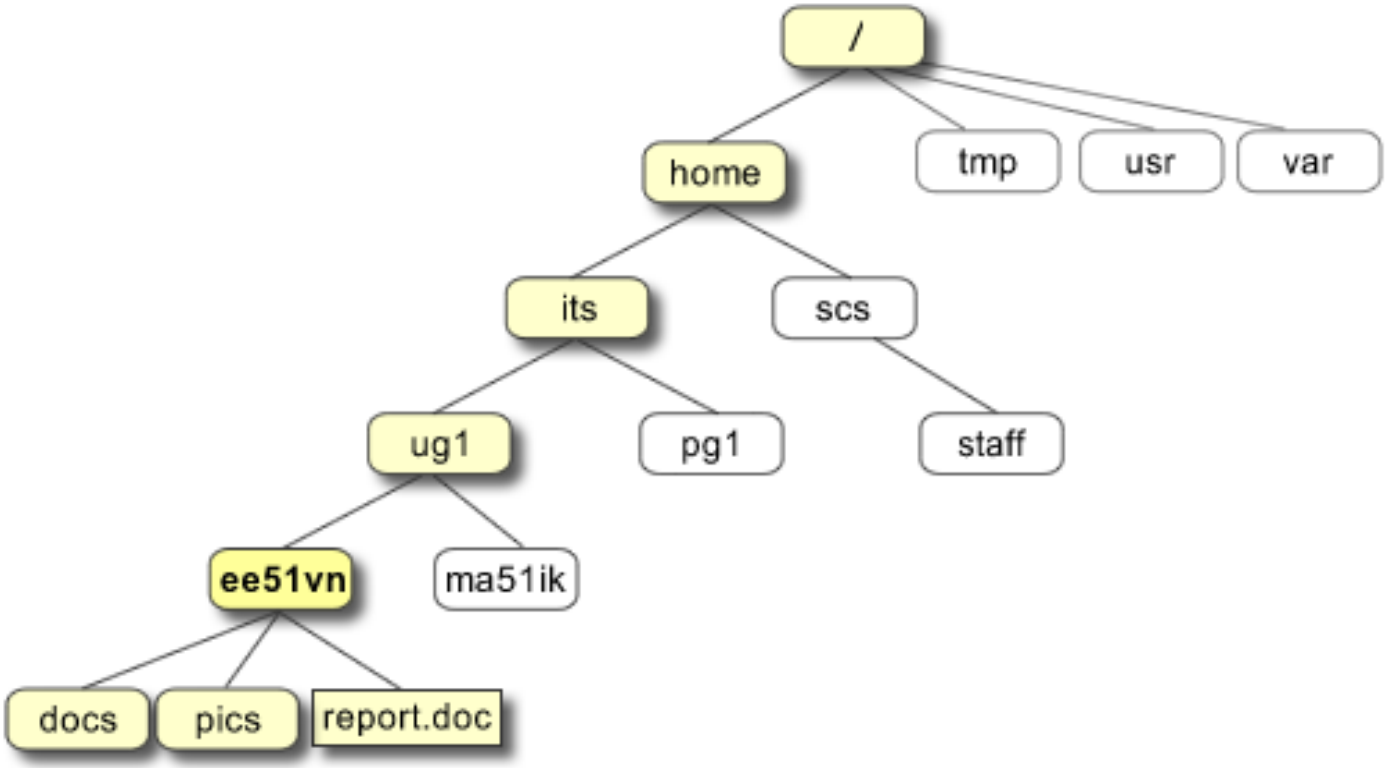
Ejemplos

1. **Clasificación de la información en una enciclopedia**



Ejemplos

2. Sistema de ficheros



Ejemplos

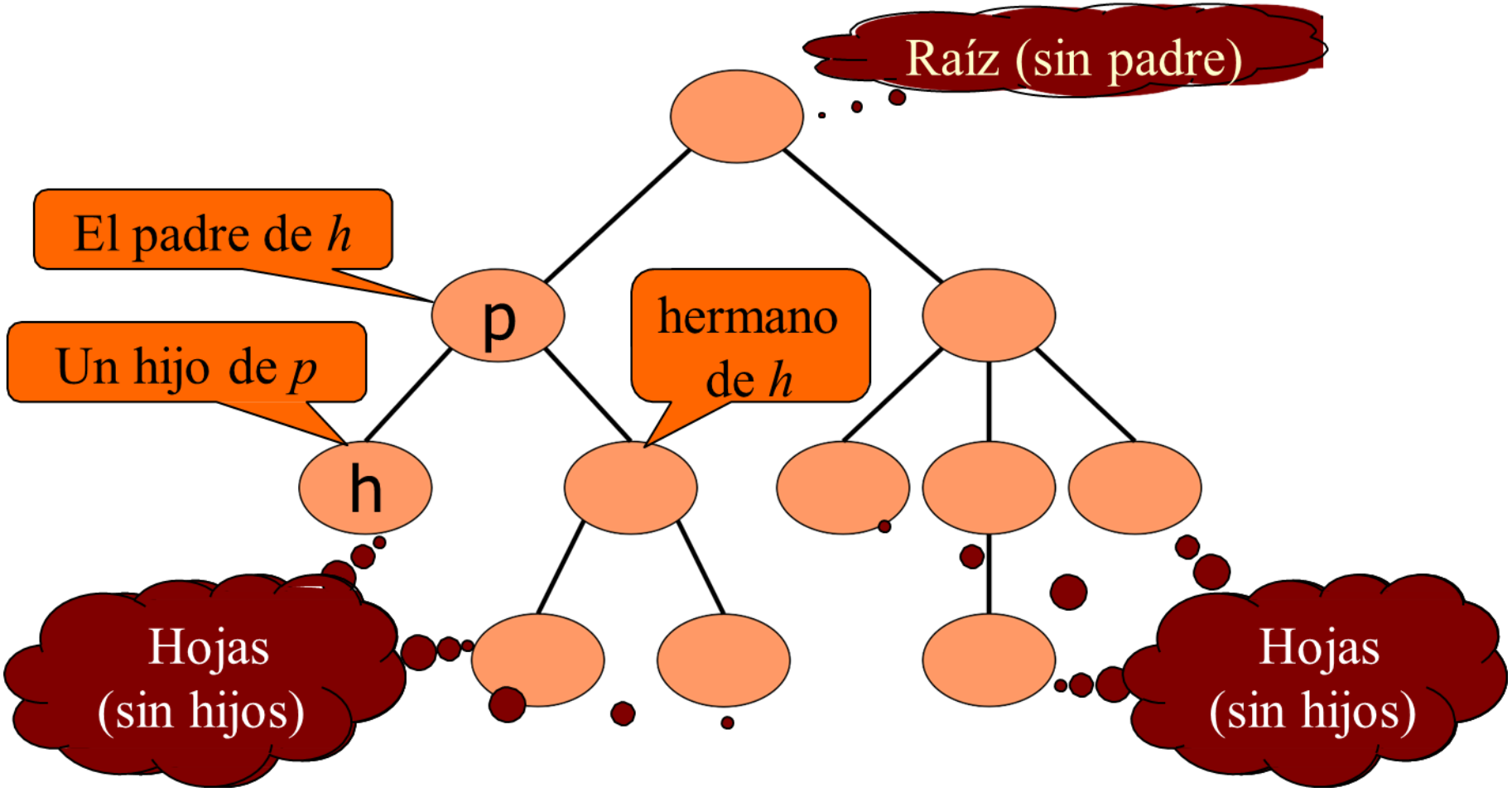
3. **Estructura organizativa de una empresa**
4. **Estructura de rangos del ejército**
5. **Estructura de un libro**

…

Definición no recursiva

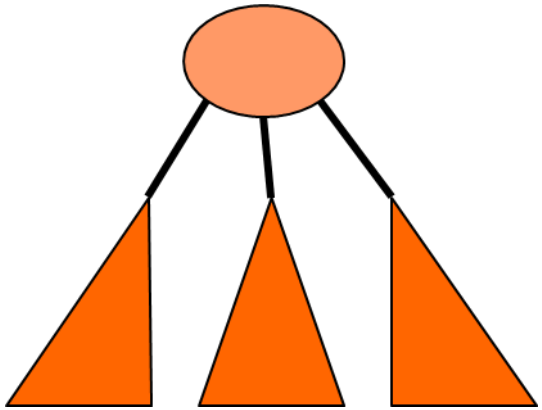
- Un árbol consiste en un conjunto de **nodos** y un conjunto de **aristas**, de forma que:
  - Se distingue un nodo llamado **raíz**
  - A cada nodo h (**hijo**), excepto la raíz, le llega una arista de otro nodo p (**padre**)
  - Para cada nodo hay un **camino** (secuencia de aristas) único desde la raíz
  - Los nodos que no tienen hijos se denominan **hojas**

Definición no recursiva



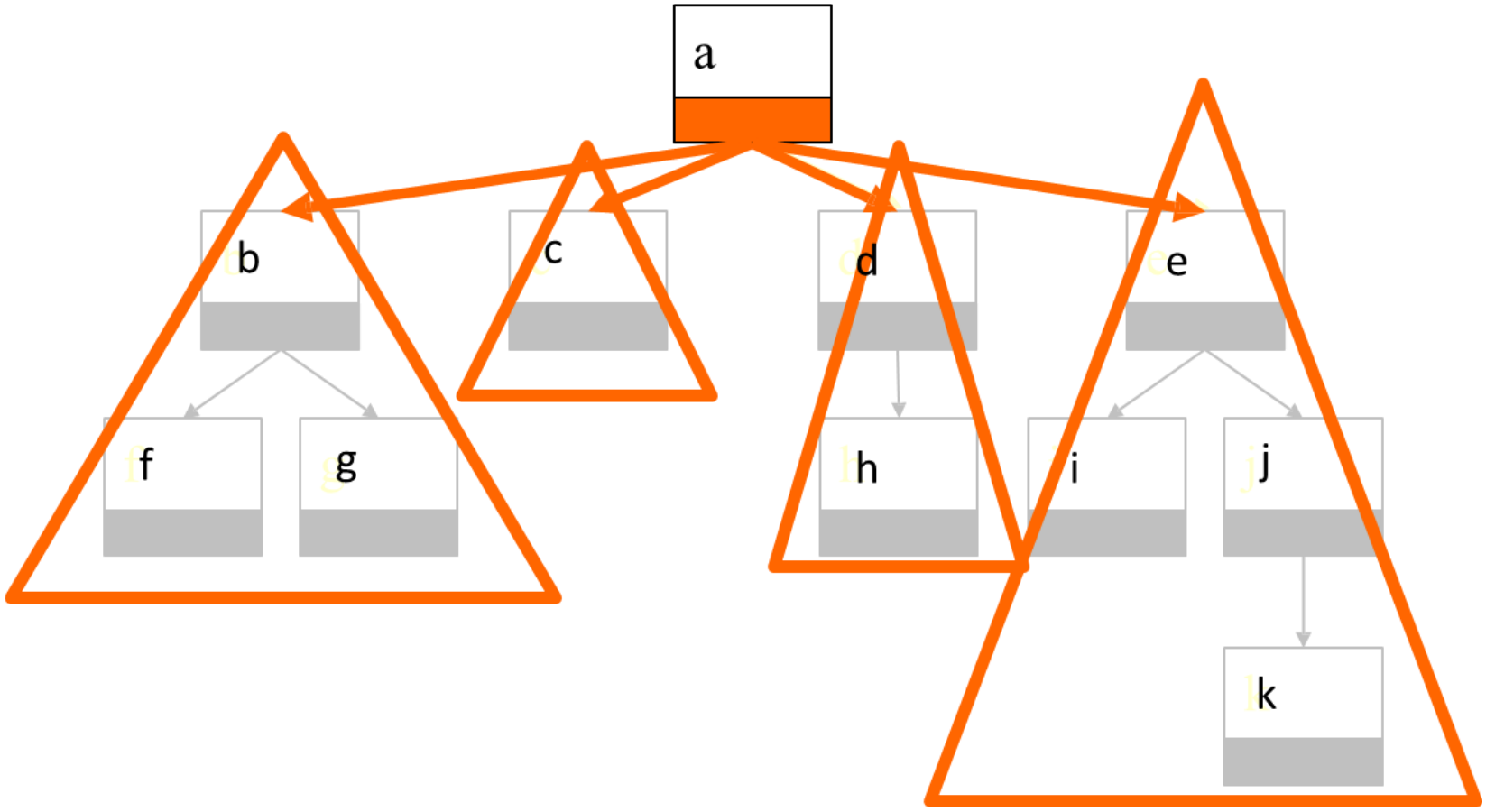
Definición recursiva

- Un árbol es:
  - Vacío
  - Un nodo y cero o más árboles conectados al nodo mediante una arista

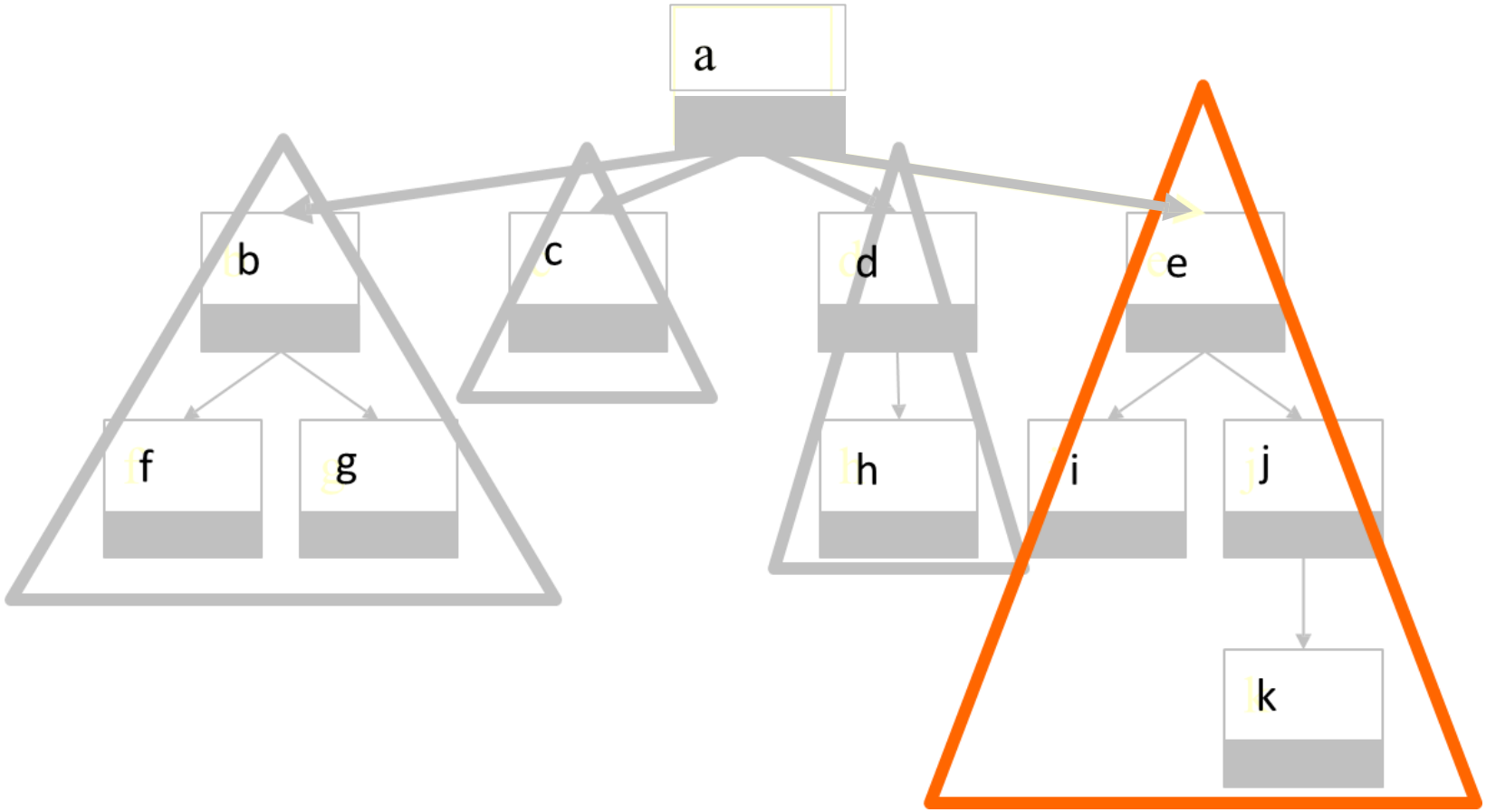


* A los árboles que se conectan al nodo raíz los denominaremos también “subárboles”

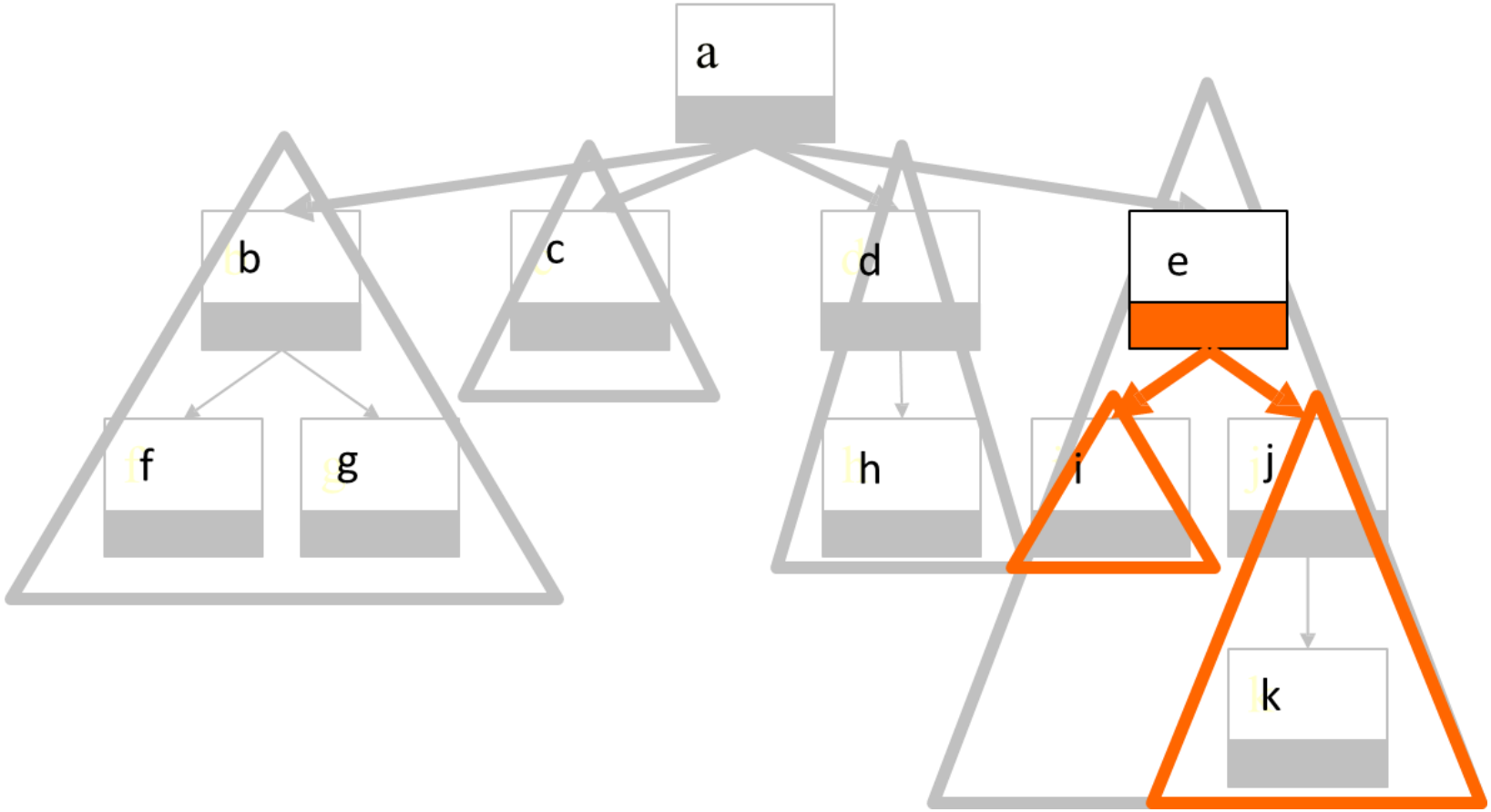
Definición recursiva



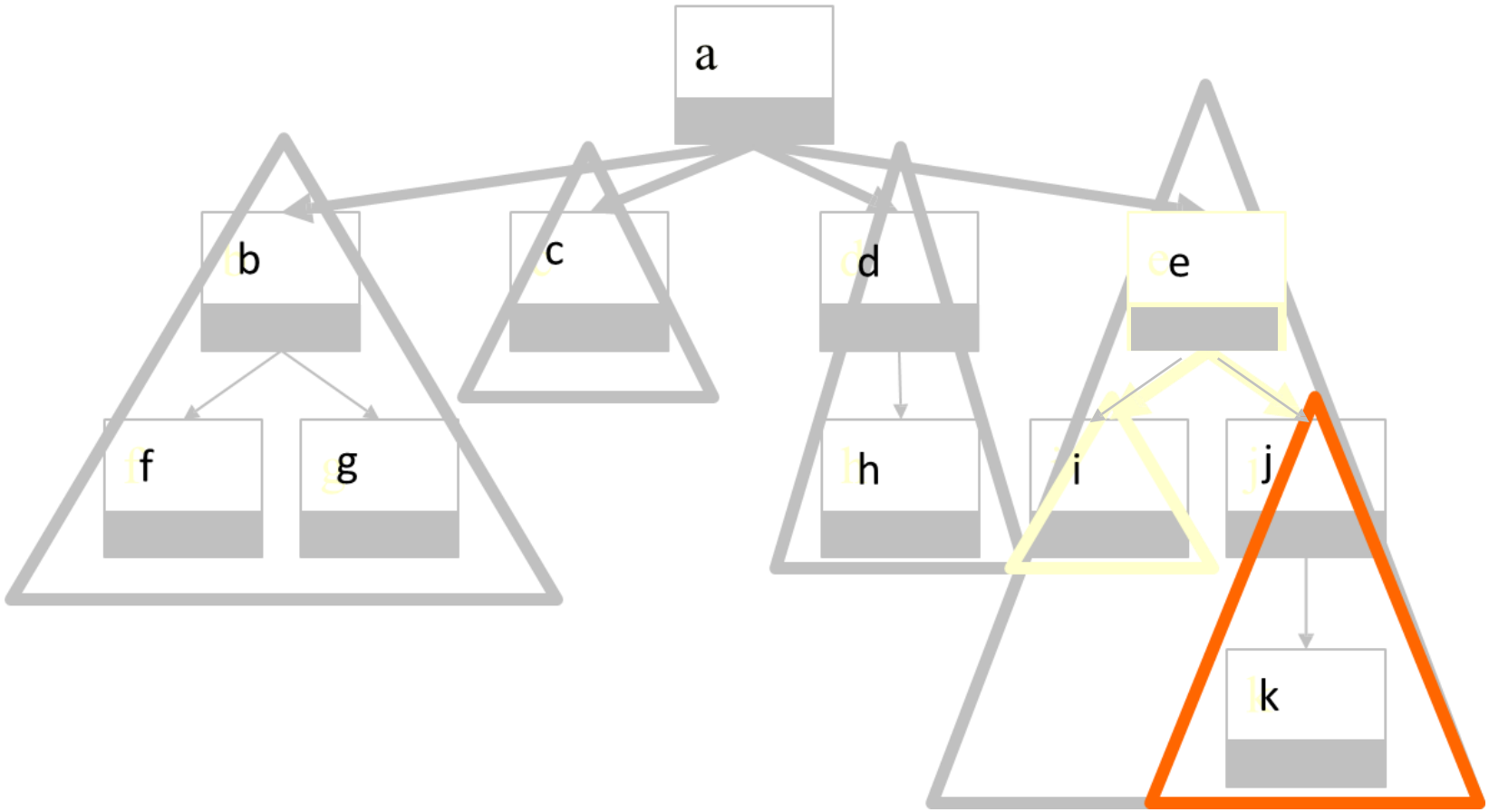
Definición recursiva



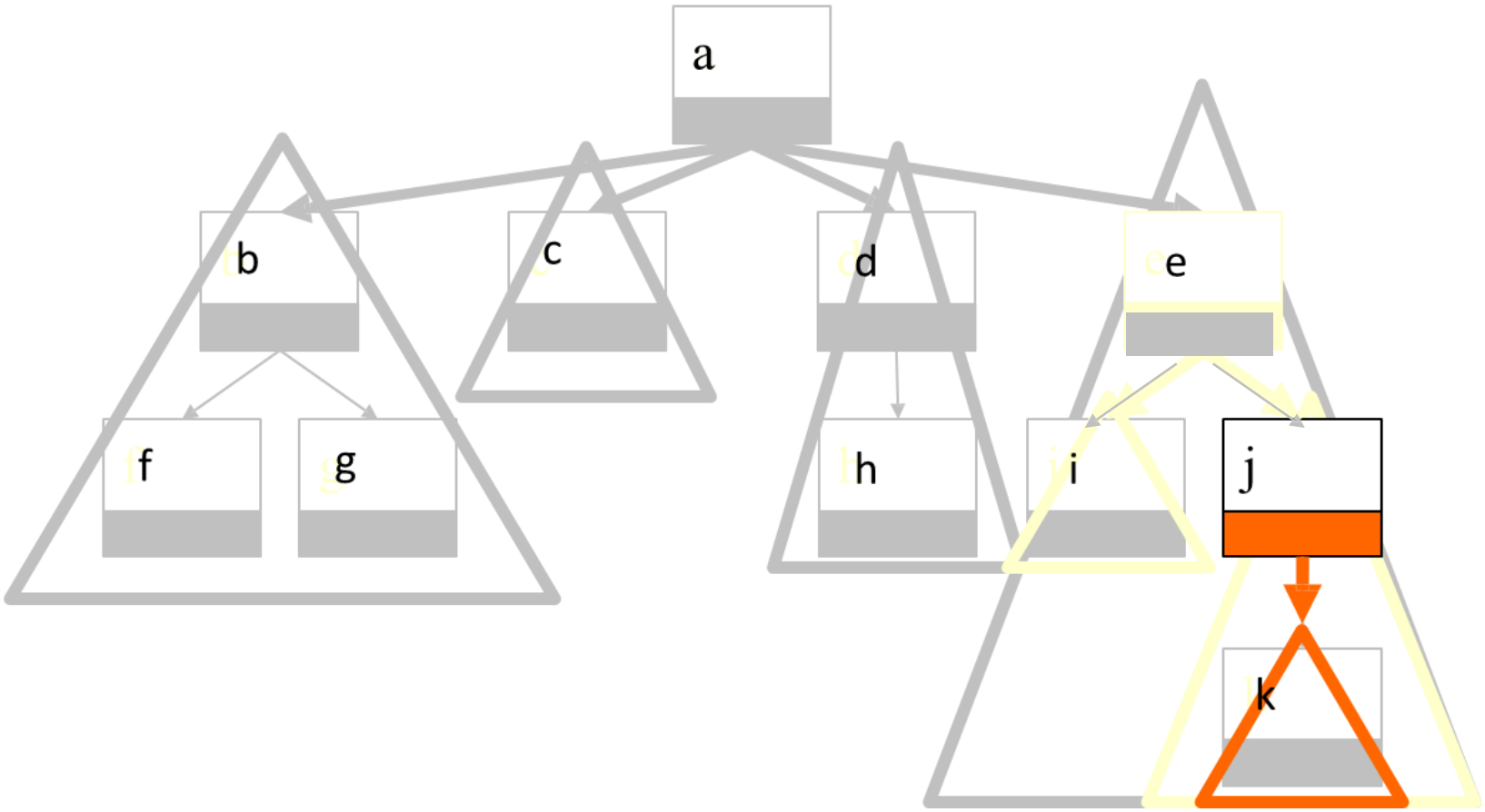
Definición recursiva



Definición recursiva



Definición recursiva



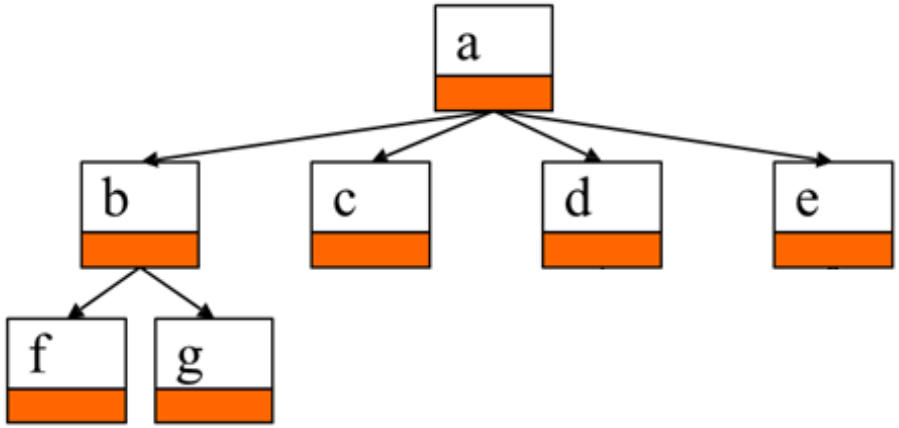
Terminología

- Un nodo es **externo**, si no tiene hijos (es *hoja*)
  - Según la definición recursiva: si todos los subárboles conectados a ese nodo están vacíos
- Un nodo es **interno**, si tiene uno o más hijos
  - Según la definición recursiva: si alguno de los subárboles conectados a ese nodo no está vacío
- Un nodo es **ascendiente** de otro, si es padre suyo o ascendiente de su padre
- Un nodo es **descendiente** de otro, si este último es ascendiente del primero
  - Los descendientes de un nodo forman un subárbol en el que ese nodo hace de raíz

Terminología

- Un **camino** de un nodo a otro, es una secuencia de aristas consecutivas que llevan del primero al segundo
  - La **longitud** del camino es el número de aristas
- La **profundidad** de un nodo es la longitud del camino de la raíz a ese nodo
- La **altura** de un árbol es el valor de la profundidad del nodo más profundo
- El **tamaño** de un árbol es el número de nodos que contiene

Ejemplo

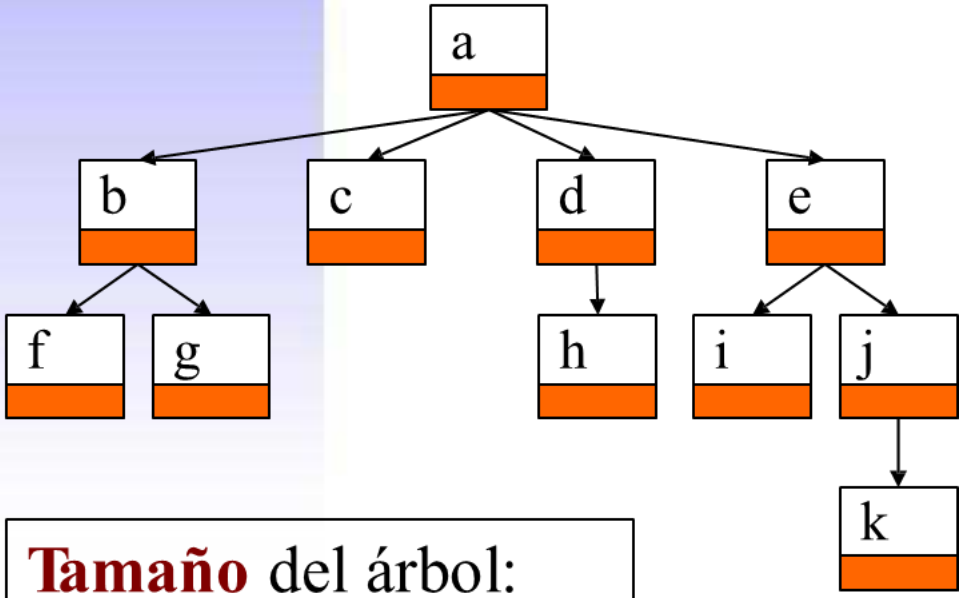


Tamaño del árbol: 7
Altura del árbol: **2**

Nodo	Altura	Profundidad	Tamaño	Int./Ext.
a	**2**	0	**7**	Interno
b	1	1	3	Interno
c	0	1	1	Externo
d	0	1	1	Externo
e	0	1	1	Externo
f	0	2	1	Externo
g	0	2	1	Externo

Ejercicio 1

- Completa la tabla para el siguiente árbol

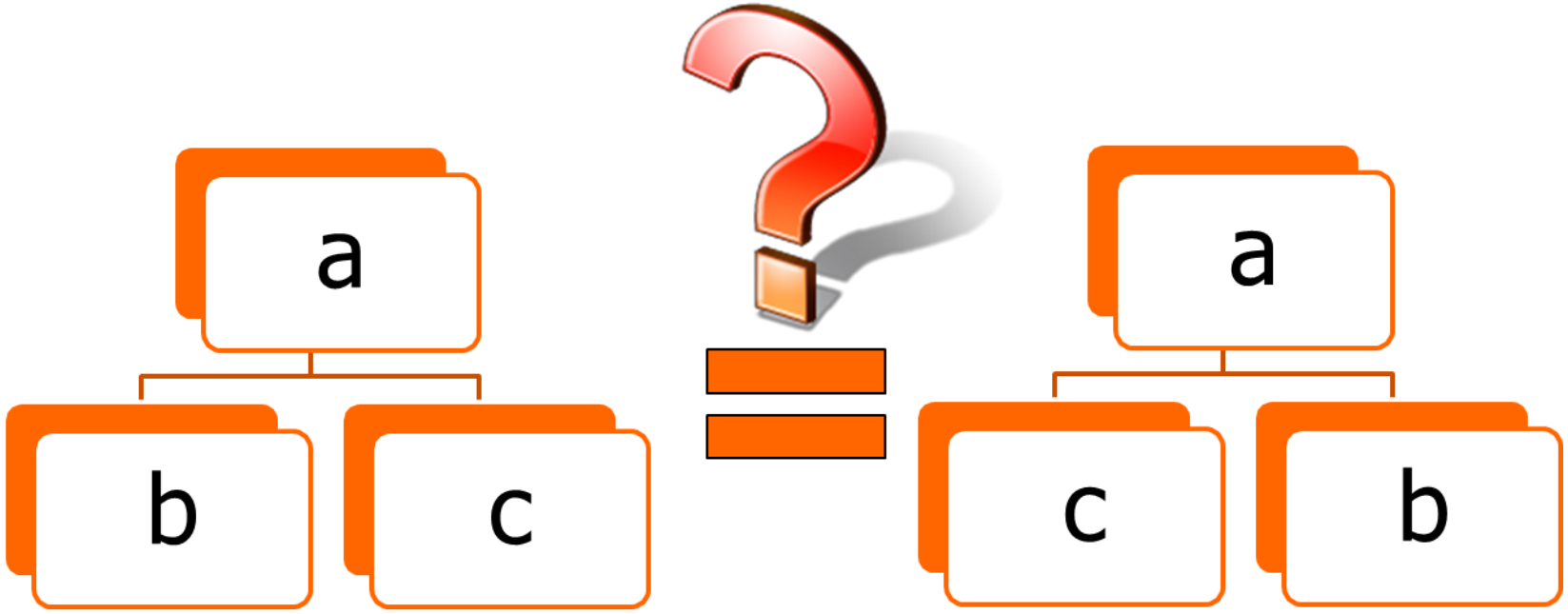


Tamaño del árbol:
Altura del árbol:

Nodo	Altura	Profundidad	Tamaño	Int./Ext.
a				Interno
b	1	1	3	Interno
c	0	1	1	Externo
d				
e				
f	0	2	1	Externo
g	0	2	1	Externo
h				
i				
j				
k				

Terminología: Árbol ordenado

- Un árbol es **ordenado**, si para cada nodo existe un orden lineal para todos sus hijos



Terminología: Árbol binario

- Un árbol **binario** es un árbol ordenado en el que cada nodo tiene 2 árboles (izquierdo y derecho).
  - Árbol binario según la definición recursiva de árbol
  - Los árboles izquierdo y/o derecho pueden estar vacíos

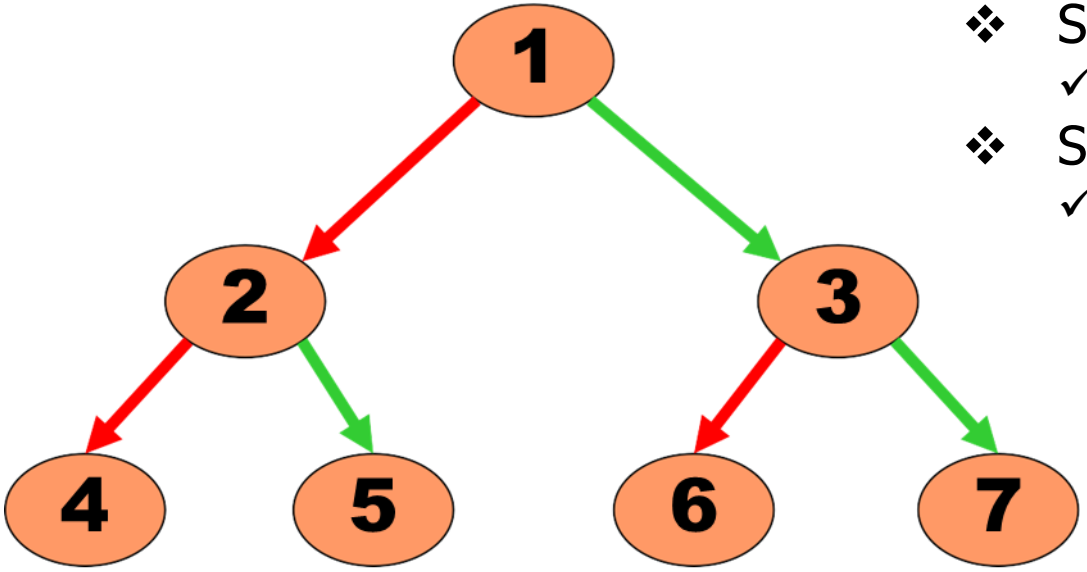
* En general, suponemos árboles binarios para simplificar la implementación de los árboles

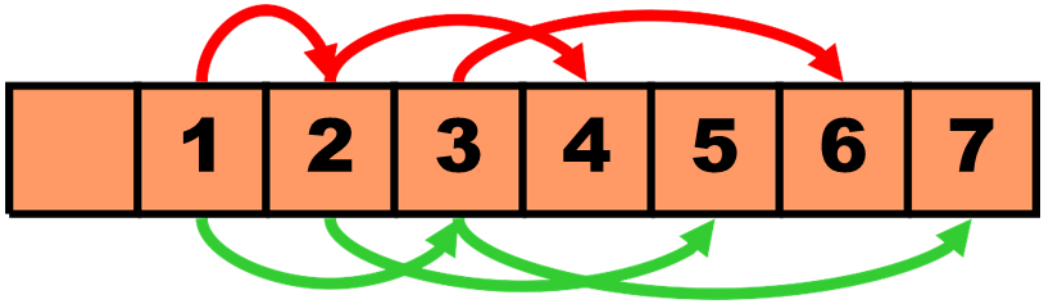
La interfaz BTree

```
public interface BTree<E> {

    static final int LEFT = 0;
    static final int RIGHT = 1;

    boolean isEmpty();
    E getInfo() throws BTreeException;
    BTree<E> getLeft() throws BTreeException;
    BTree<E> getRight() throws BTreeException;

    void insert(BTree<E> tree, int side) throws BTreeException;
    BTree<E> extract(int side) throws BTreeException;

    String toStringPreOrder();
    String toStringInOrder();
    String toStringPostOrder();
    String toString(); // preorder

    int size();
    int height();

    boolean equals(BTree<E> tree);
    boolean find(BTree<E> tree);
}
```

Una interfaz varias implementaciones

- Implementación basada en arrays

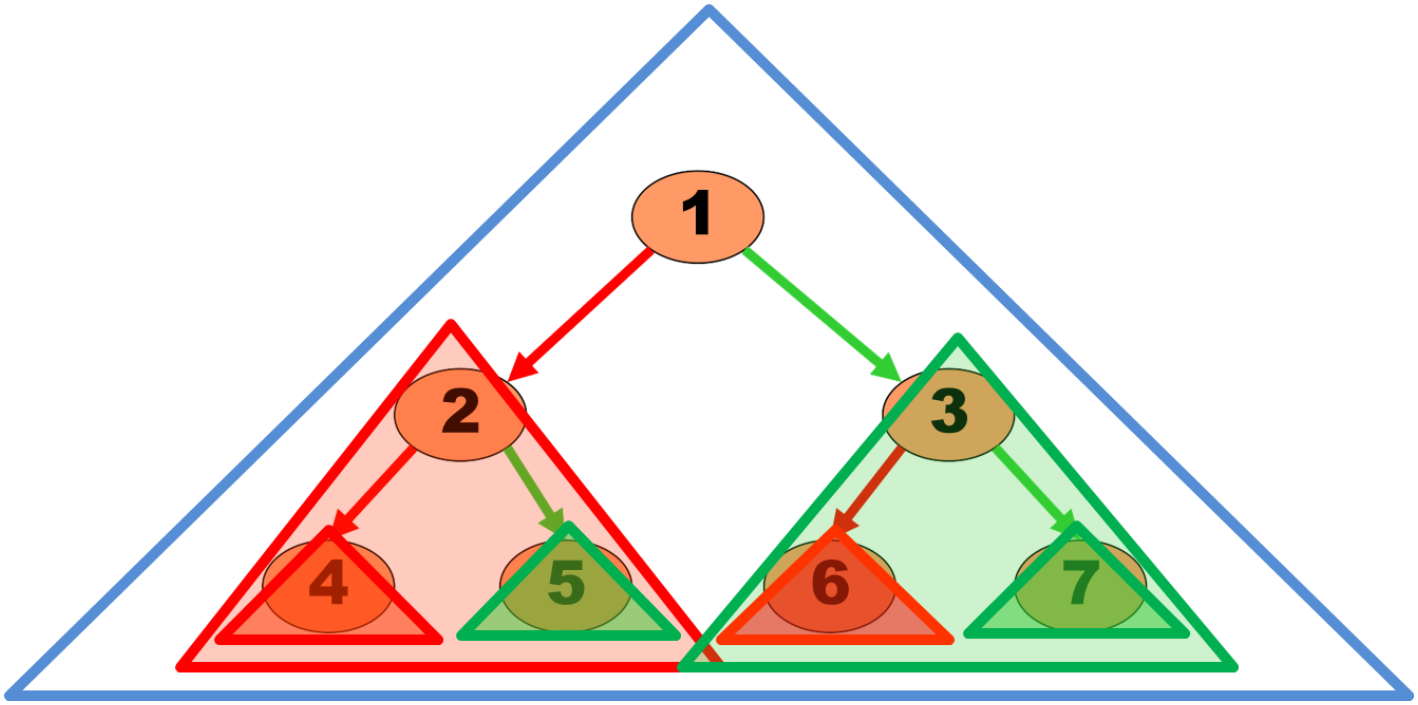




- Subárbol izquierdo
  - ✓ Posición nodo raíz * 2
- Subárbol derecho
  - ✓ Posición nodo raíz * 2 +1

Una interfaz varias implementaciones

- Implementación basada en enlaces
  - *Linked Binary Node (LBNode)*
  - *Linked Binary Tree (LBTree)*
  - ✓ Cada árbol (*LBTree)* tiene un nodo raíz (atributo *LBNode*)
  - ✓ Cada nodo raíz *LBNode* apunta a dos árboles (atributos *LBTree*), los cuales pueden estar vacíos (null)



La clase LBNode

```
public class LBNode<E> {

    private E info;
    private BTree<E> left;
    private BTree<E> right;

    public LBNode(E info, BTree<E> left, BTree<E> right) {…}
    public E getInfo() {…}
    public void setInfo(E info) {…}
    public BTree<E> getLeft() {…}
    public void setLeft(BTree<E> left) {…}
    public BTree<E> getRight() {…}
    public void setRight(BTree<E> right){…}
}
```

Ejercicio 2

- Completa la implementación de la clase **`LBNode`**.

La clase LBTree

```
public class LBTree<E> implements BTree<E> {

    private LBNode<E> root;

    public LBTree() {
        root = null;
    }

    public LBTree(E info) {
        root = new LBNode<E>(info, new LBTree<E>(), new LBTree<E>());
    }

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

La clase LBTree

```
...
    public E getInfo() throws BTreeException {
        if (isEmpty()) {
            throw new BTreeException("empty trees do not have info");
        }
        return root.getInfo();
    }

    public BTree<E> getLeft() throws BTreeException {
        if (isEmpty()) {
            throw new BTreeException("empty trees do not have a left child");
        }
        return root.getLeft();
    }

    public BTree<E> getRight() throws BTreeException {
        if (isEmpty()) {
            throw new BTreeException("empty trees do not have a right child");
        }
        return root.getRight();
    }
...
```

Algoritmos básicos

- Tamaño: **size()**
- Altura: **height()**
- Recorridos
  - Pre-orden: **toStringPreOrder()**
  - In-orden: **toStringInOrder()**
  - Post-orden: **toStringPostOrder()**

La clase LBTree: size()

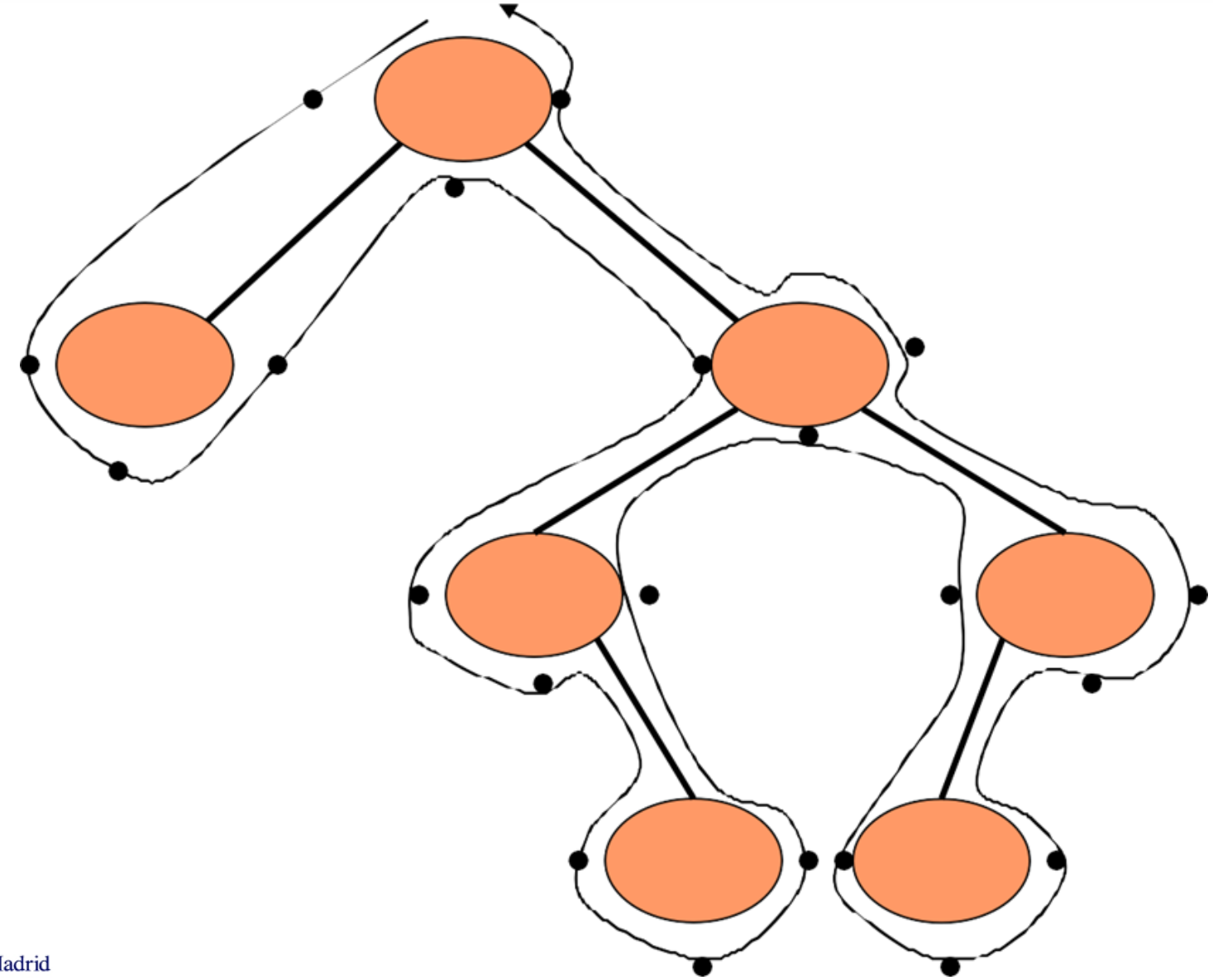
```
public int size() {
    if (isEmpty()) {
        return 0;
    } else {
        return 1 + root.getLeft().size()
            + root.getRight().size();
    }
}
```

La clase LBTree: height()

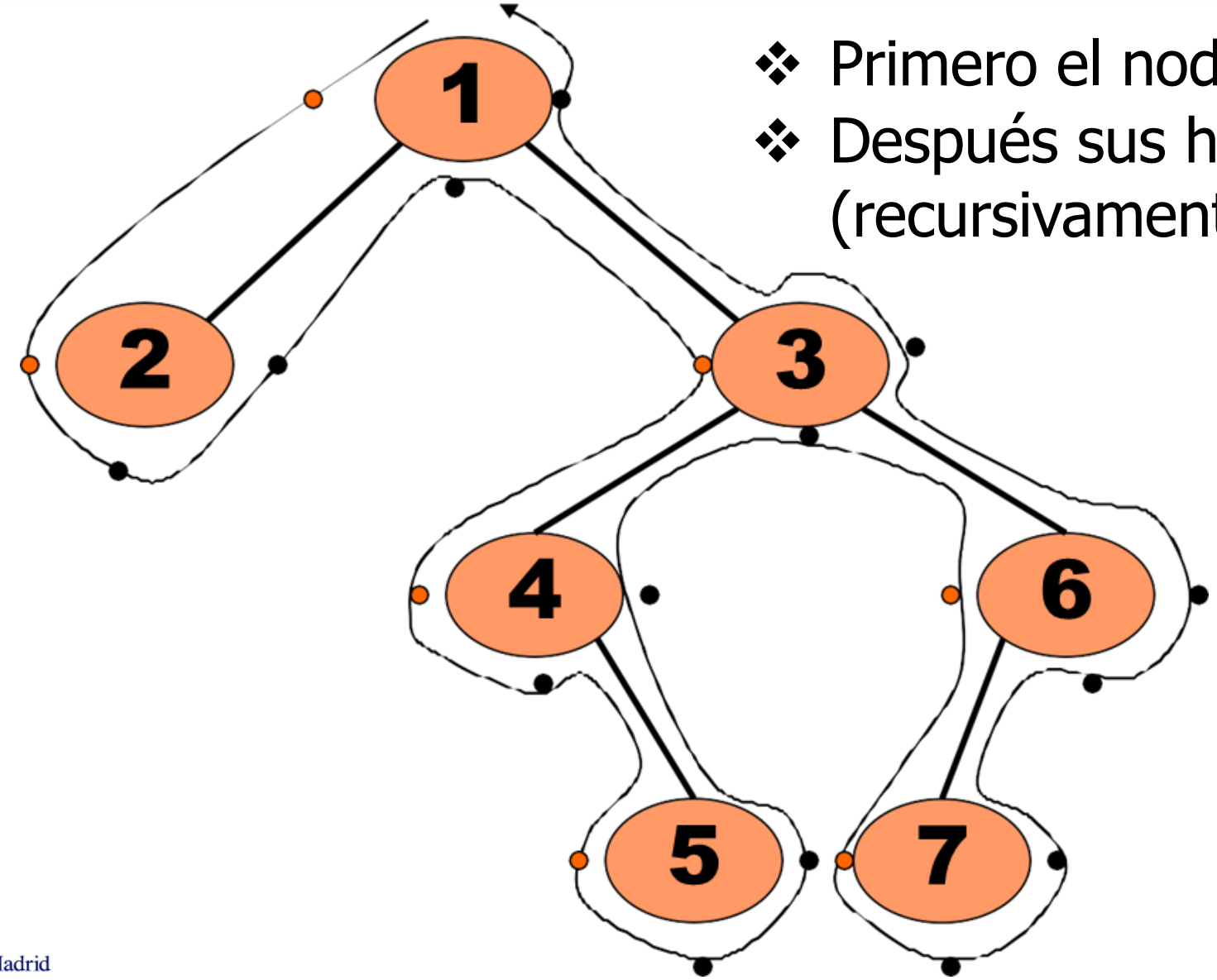
```
public int height() {
    if (isEmpty()) {
        return -1;
    } else {
        int leftHeight = root.getLeft().height();
        int rightHeight = root.getRight().height();
        if (leftHeight > rightHeight) {
            return 1 + leftHeight;
        } else {
            return 1 + rightHeight;
        }
    }
}
```

```
1+Math.max(leftHeight,
           rightHeight);
```

Recorrido de Euler



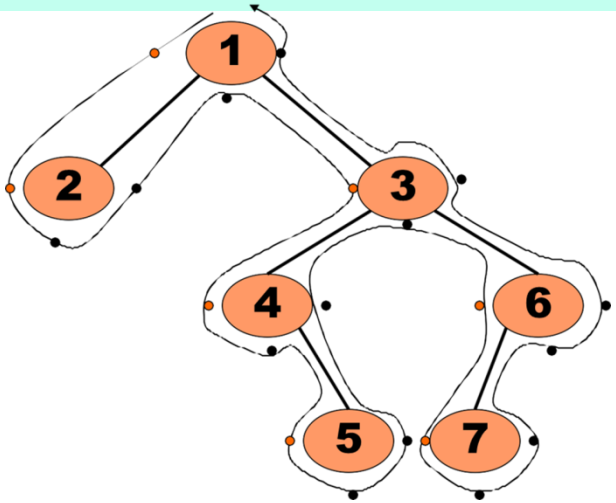
Recorrido Pre-orden



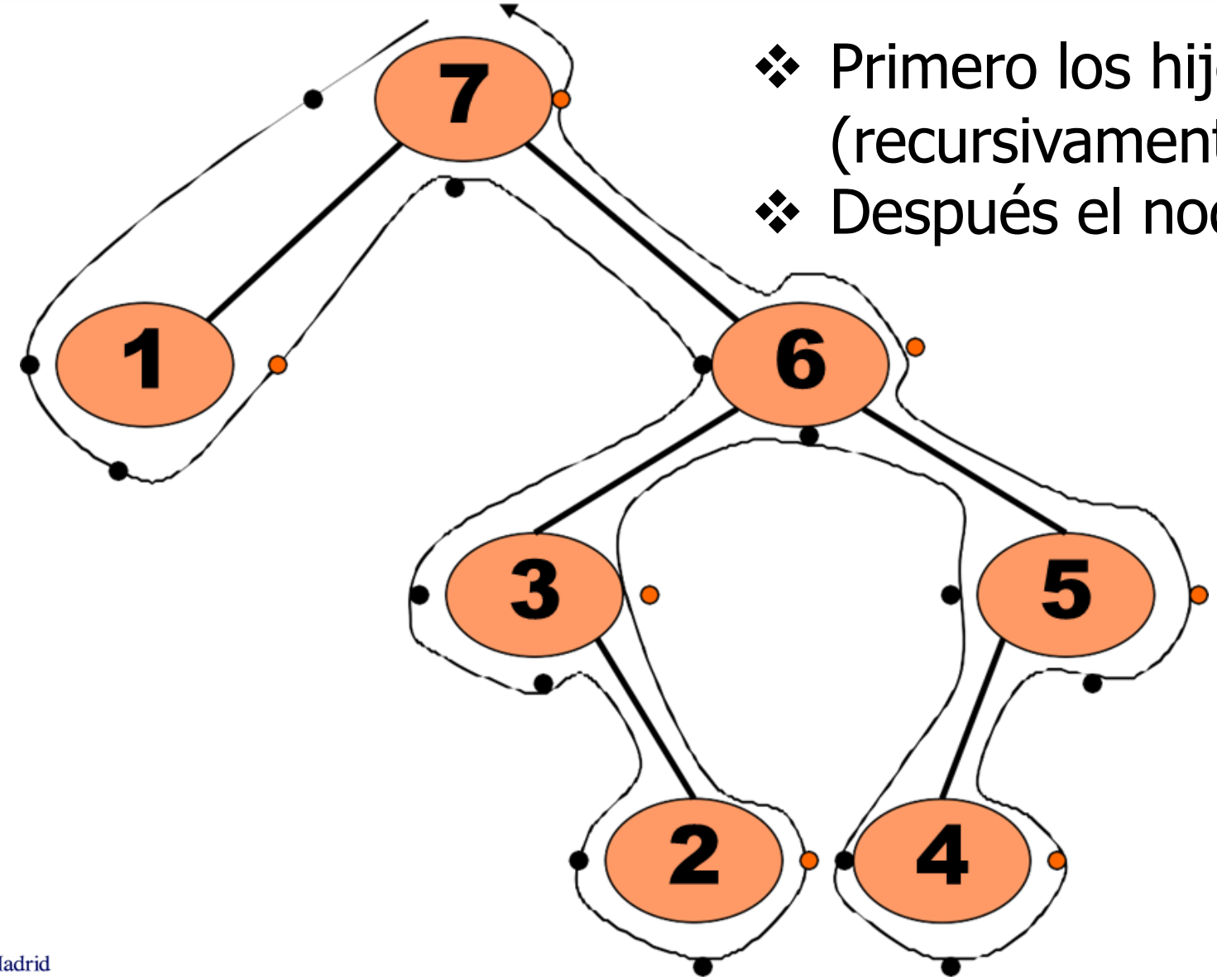
- Primero el nodo raíz
- Después sus hijos (recursivamente)

La clase LBTree: toStringPreOrder()

```
public String toStringPreOrder() {
    if (isEmpty()) {
        return "";
    } else {
        return root.getInfo().toString() + " " +
            root.getLeft().toStringPreOrder() +
            root.getRight().toStringPreOrder();
    }
}
```



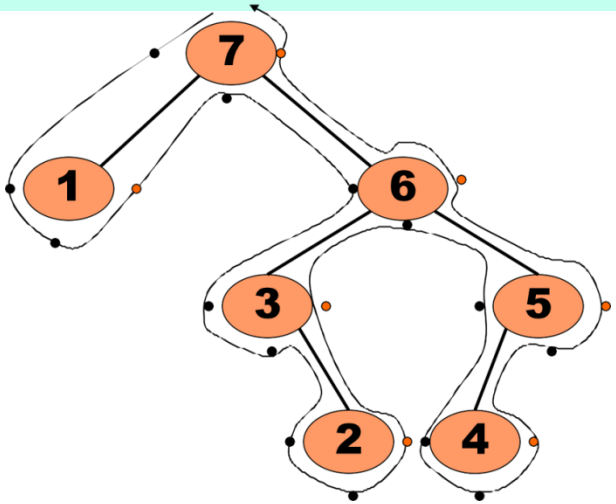
Recorrido Post-orden



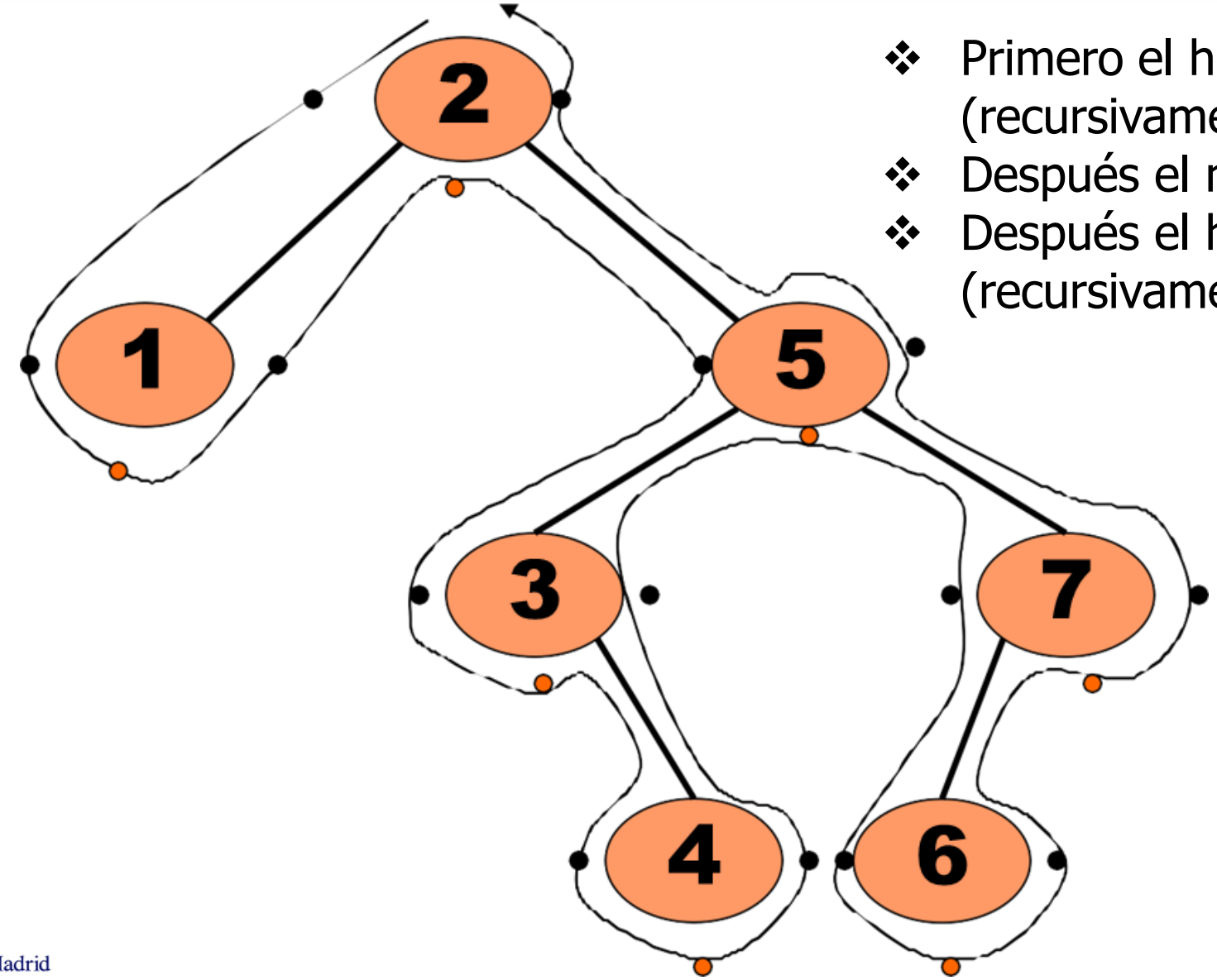
- Primero los hijos (recursivamente)
- Después el nodo raíz

La clase LBTree: toStringPostOrder()

```
public String toStringPostOrder() {
    if (isEmpty()) {
        return "";
    } else {
        return root.getLeft().toStringPostOrder() +
            root.getRight().toStringPostOrder() +
            root.getInfo().toString() + " ";
    }
}
```



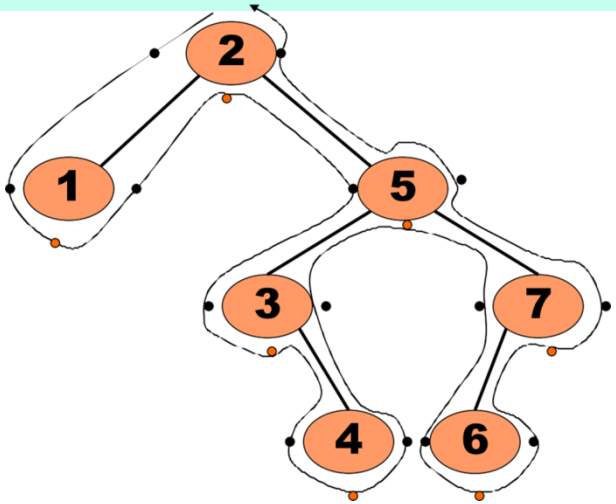
Recorrido In-orden (simétrico)



- Primero el hijo izquierdo (recursivamente)
- Después el nodo raíz
- Después el hijo derecho (recursivamente)

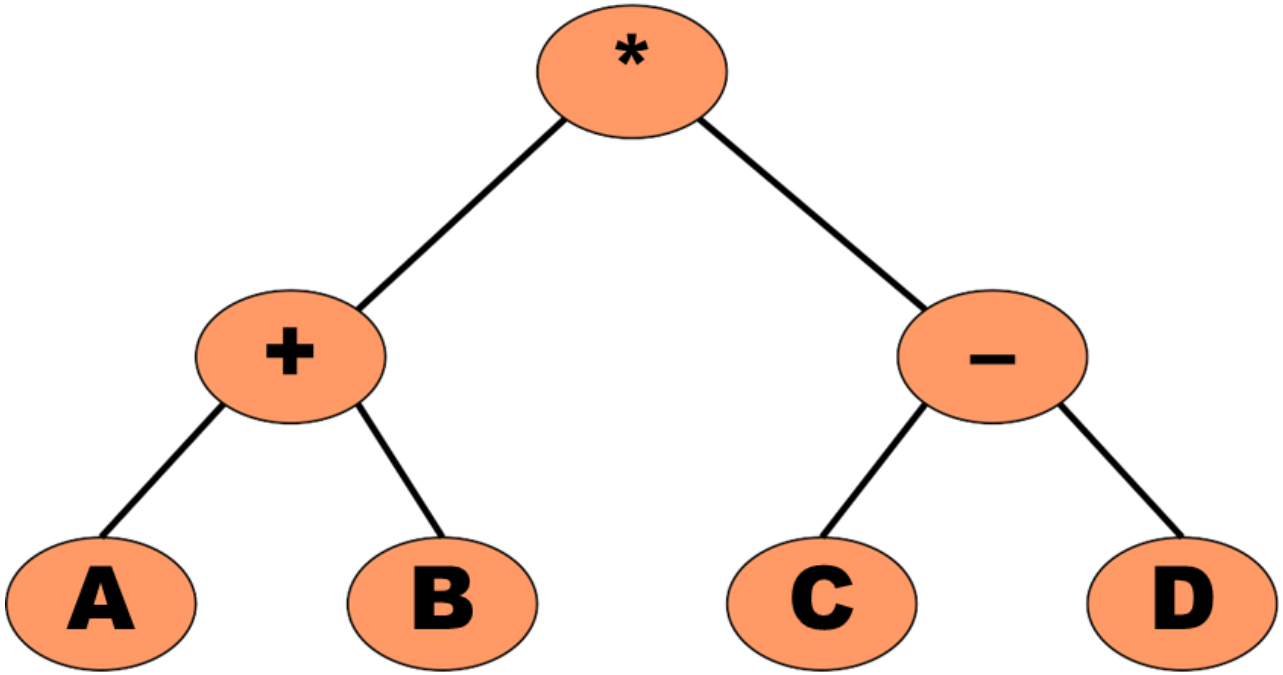
La clase LBTree: toStringInOrder()

```
public String toStringInOrder() {
    if (isEmpty()) {
        return "";
    } else {
        return root.getLeft().toStringInOrder() +
            root.getInfo().toString() + " " +
            root.getRight().toStringInOrder();
    }
}
```



Ejercicio 3

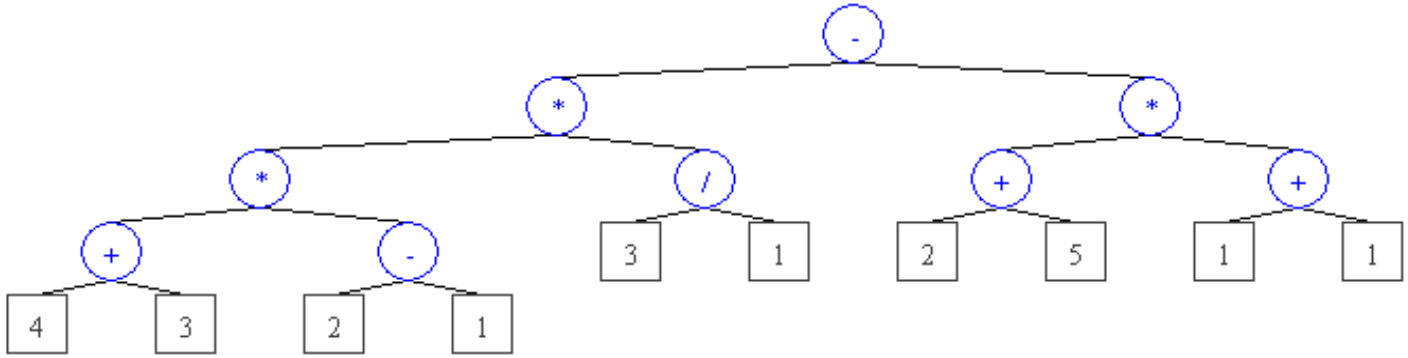
- Dado el siguiente árbol binario, indica qué recorrido (pre-orden, in-orden, post-orden) produce el resultado (A+B)*(C-D).



Diferente notación matemática

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B–C	–+ABC	AB+C–
(A+B)*(C–D)	*+AB–CD	AB+CD–*

Actividad



Expresiones matemáticas como árboles:
http://www.cs.jhu.edu/~goodrich/dsa/05trees/Demo1/

Evaluate

((((4+3)*(2-1))*(3/1))-((2+5)*(1+1)))

Result is 7

Programación de sistemas

Árboles (II)

Julio Villena Román
<jvillena@it.uc3m.es>

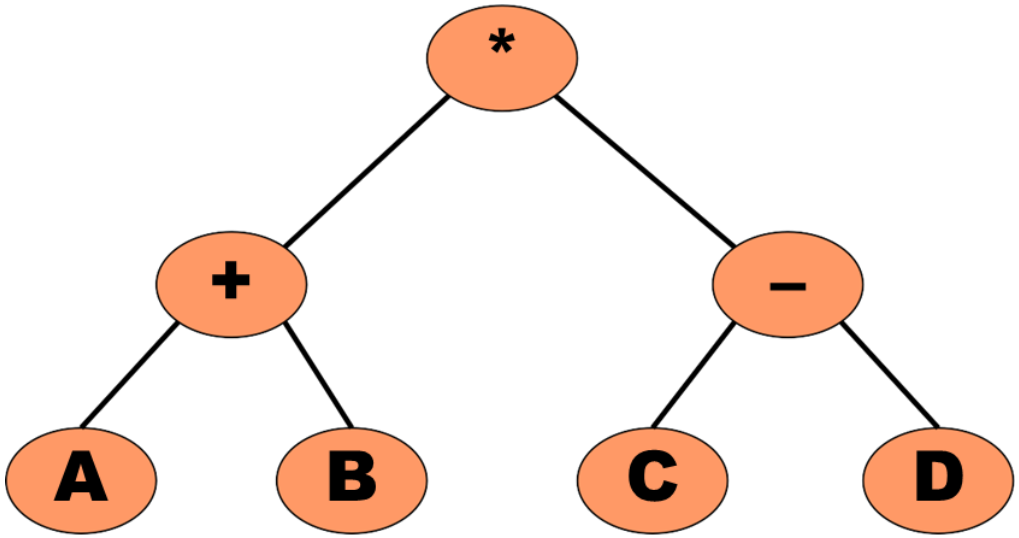
MATERIALES BASADOS EN EL TRABAJO DE DIFERENTES AUTORES:
Carlos Delgado Kloos, M.Carmen Fernández Panadero,
Raquel M.Crespo García, Carlos Alario Hoyos

Contenidos

- Concepto de árbol
- Terminología
- Implementación
- **Casos especiales**
  - **Árboles binarios de búsqueda**
  - **Montículos (*heaps*)**

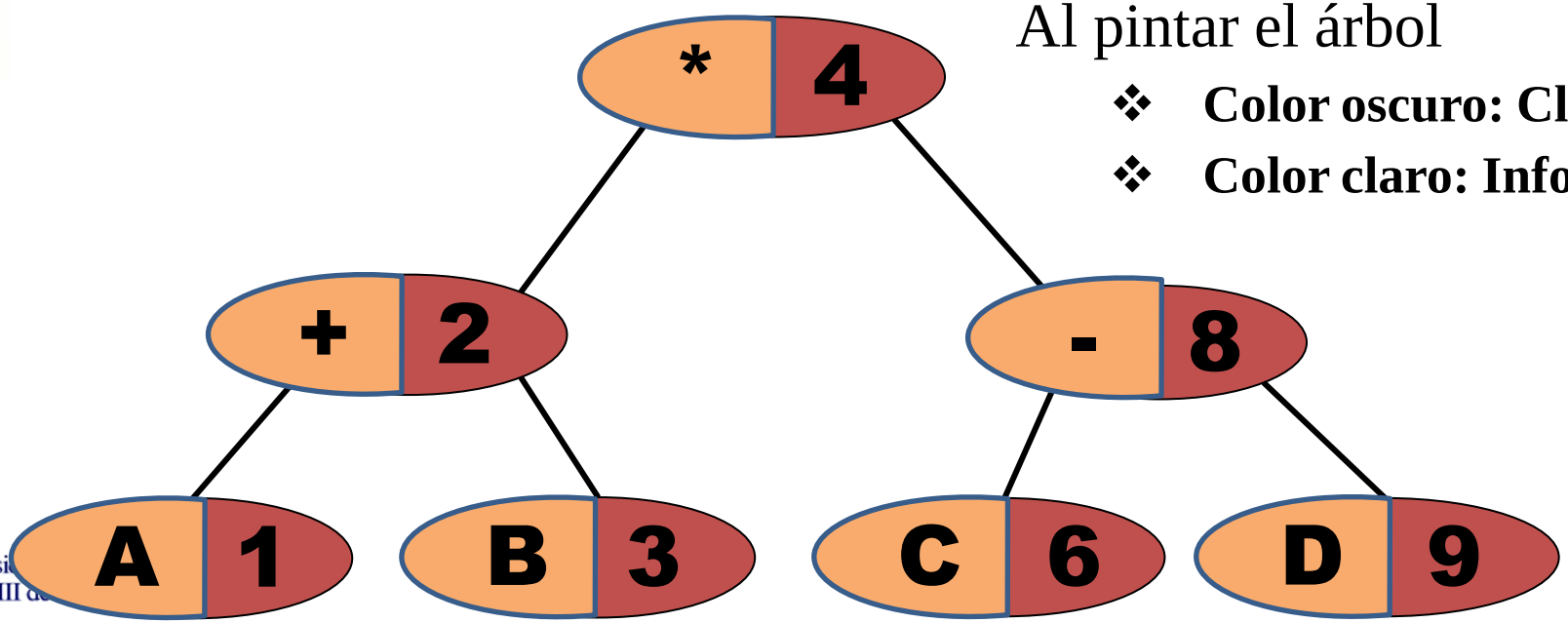
Notación

- Hasta ahora:
  - **Un nodo, tres atributos**: información almacenada, subárbol izquierdo y subárbol derecho
  - **Un árbol, un atributo**: nodo raíz
- Al pintar un árbol representamos la información almacenada como el contenido de cada nodo



Notación

- A partir de ahora:
  - Añadimos un atributo más: **la clave (*key*)**
    - Facilita la utilización del árbol en operaciones de búsqueda, inserción, eliminación
    - Dependiendo de la implementación, la clave puede añadirse como atributo del nodo o como atributo del árbol

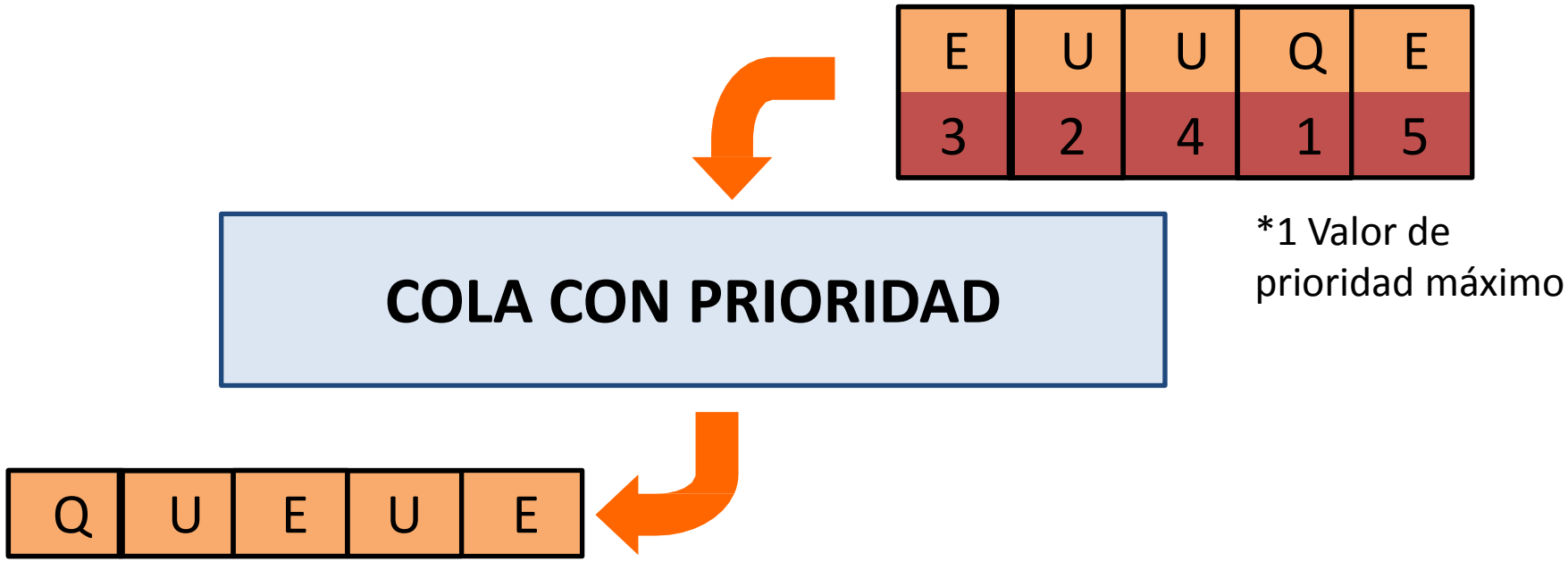


Al pintar el árbol

- **Color oscuro: Clave**
- **Color claro: Información**

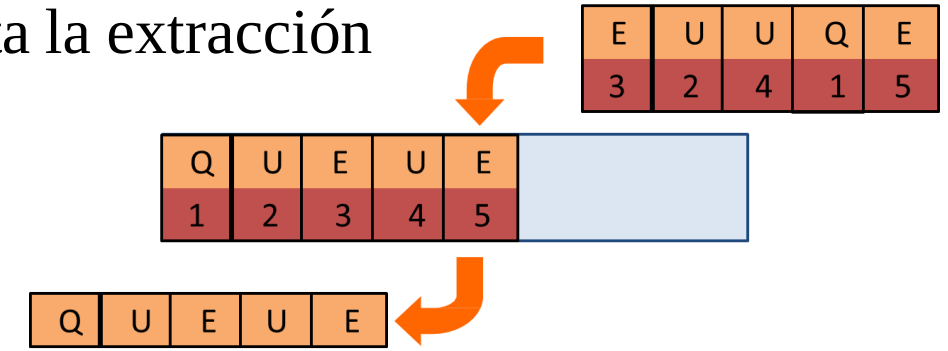
Ejemplo de uso de claves

- **Colas con prioridad**
  - Estructura de datos lineal que devuelve los elementos de acuerdo con el valor de una clave que determina la prioridad y no al orden en que fueron insertados

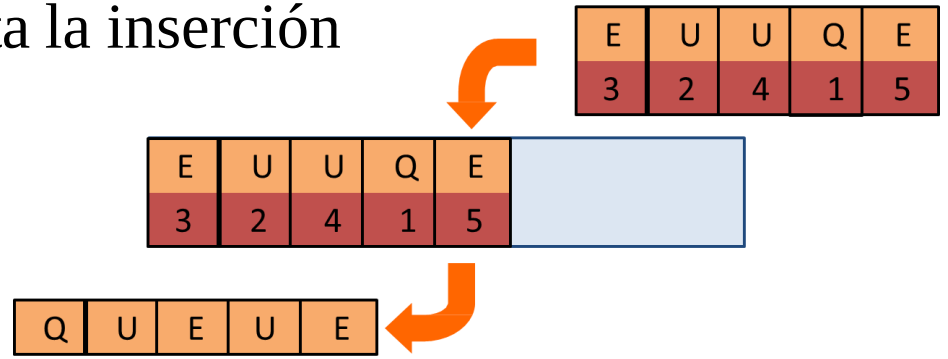


Ejemplo de uso de claves

- **Colas con prioridad**
  - Implementación 1: Comparar al insertar
    - ✓ Facilita la extracción



  - Implementación 2: Comparar al extraer
    - ✓ Facilita la inserción



Árboles binarios de búsqueda

- Concepto de árbol binario de búsqueda
- Operaciones
  - Búsqueda
  - Inserción
  - Eliminación

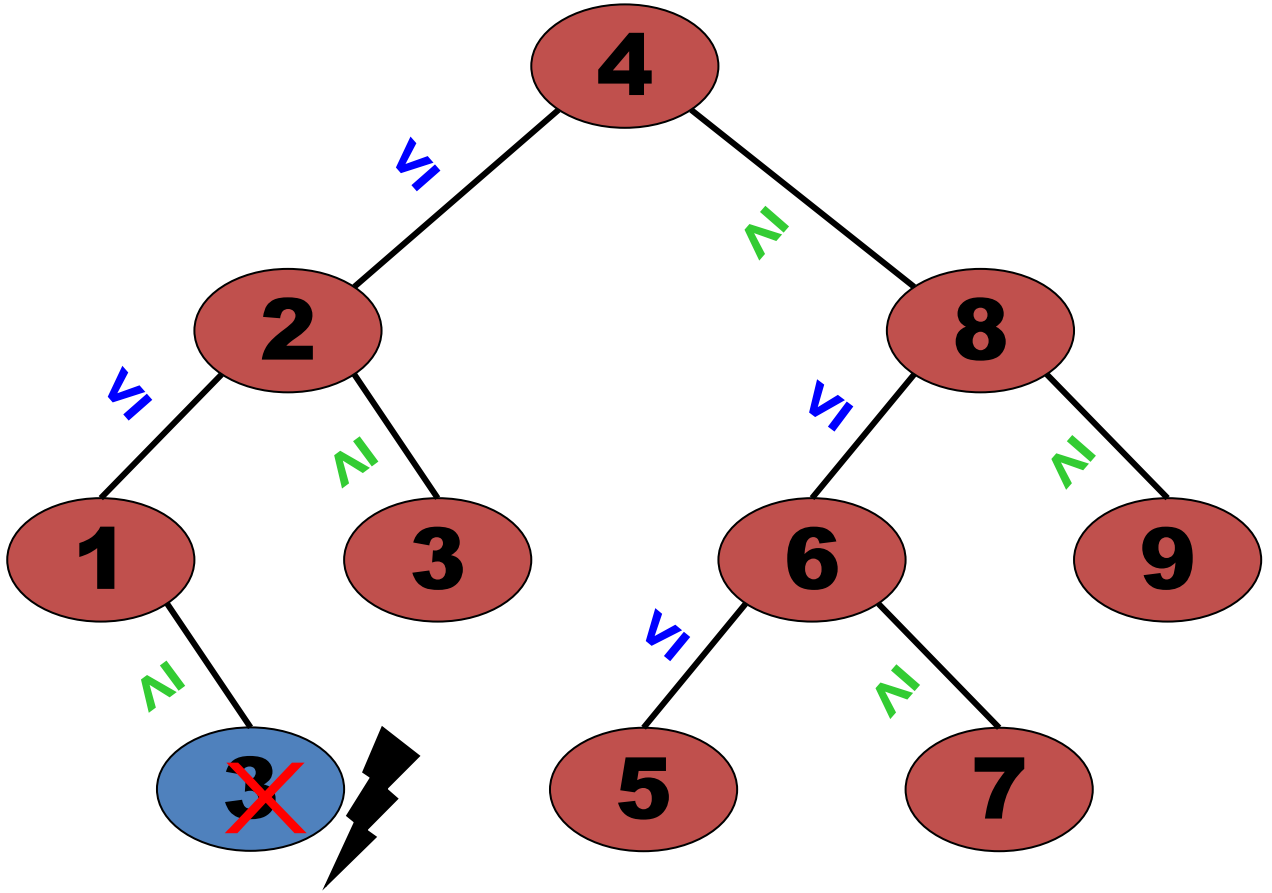
Árboles binarios de búsqueda: Concepto

Un **árbol binario de búsqueda** es un árbol binario en el que para cada nodo *n*,

- todas las claves de los nodos del subárbol **izquierdo** son **menores** que la clave de *n* (o iguales)

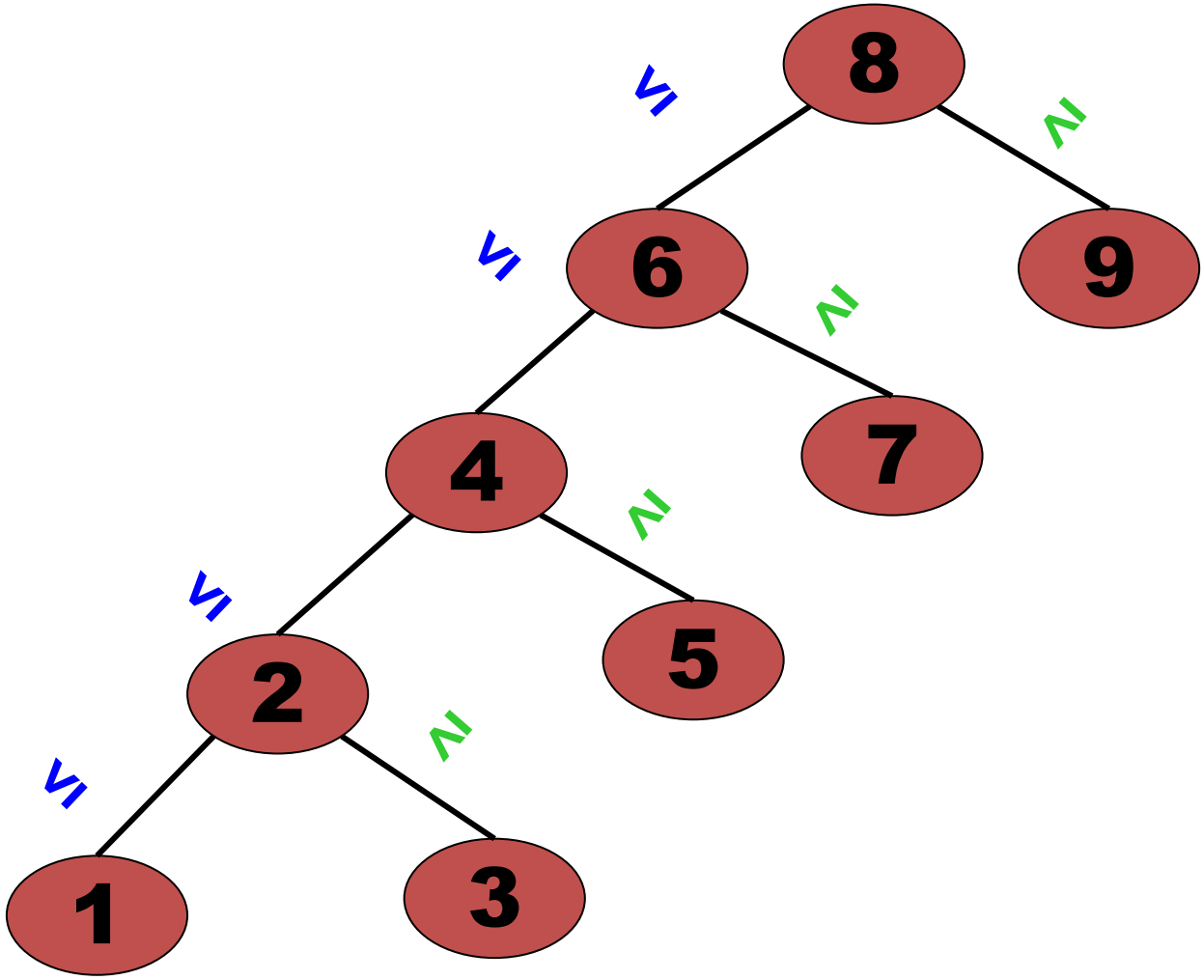
- y todas las del subárbol **derecho** **mayores** (o iguales).

Ejemplo (I)



- Representación únicamente de las claves (no se incluye la información almacenada)
- Implementación errónea (en el subárbol izquierdo de "2" hay una clave mayor que 2)

Ejemplo (II)

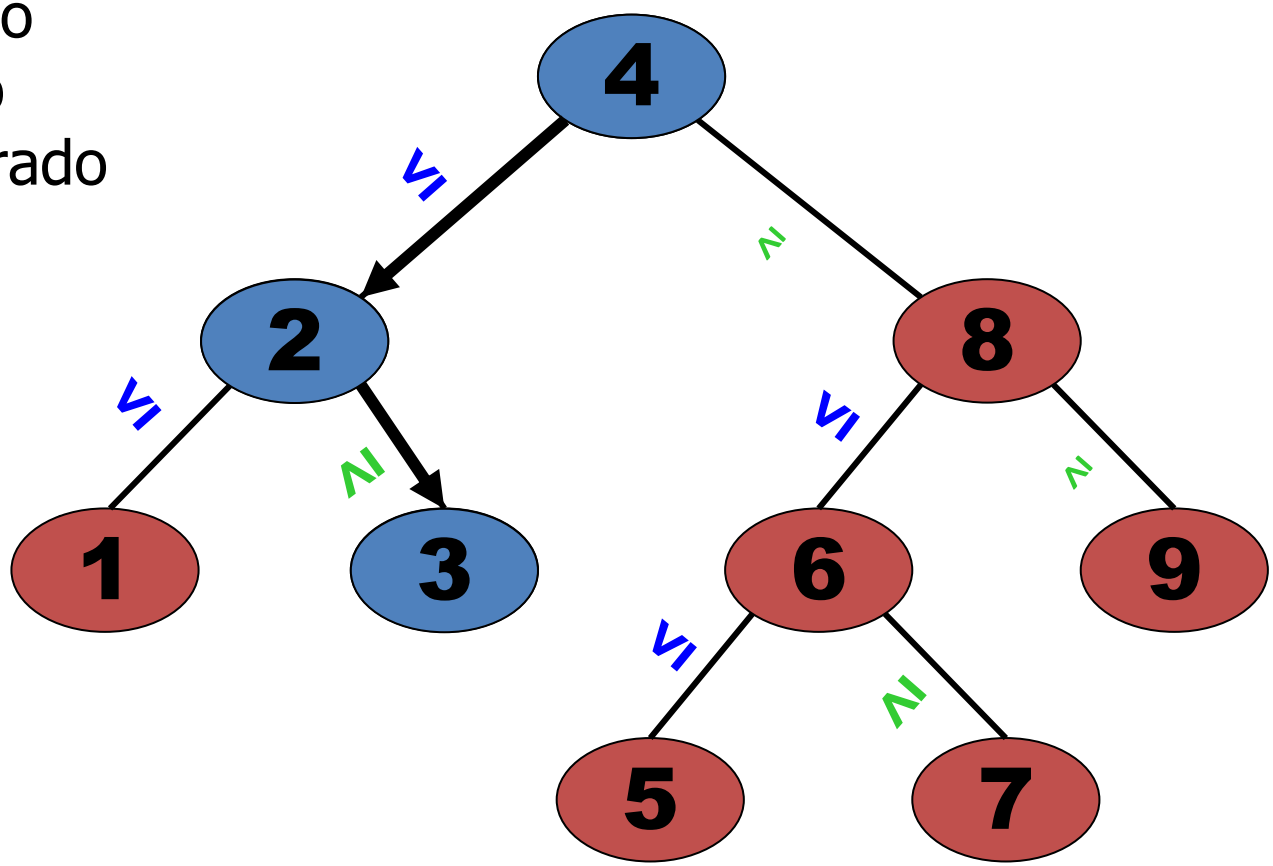


- Implementación correcta de árbol binario de búsqueda

Operación: Búsqueda

Buscamos el "3":

- 3<4: Subárbol izquierdo
- 3>2: Subárbol derecho
- 3=3: Elemento encontrado

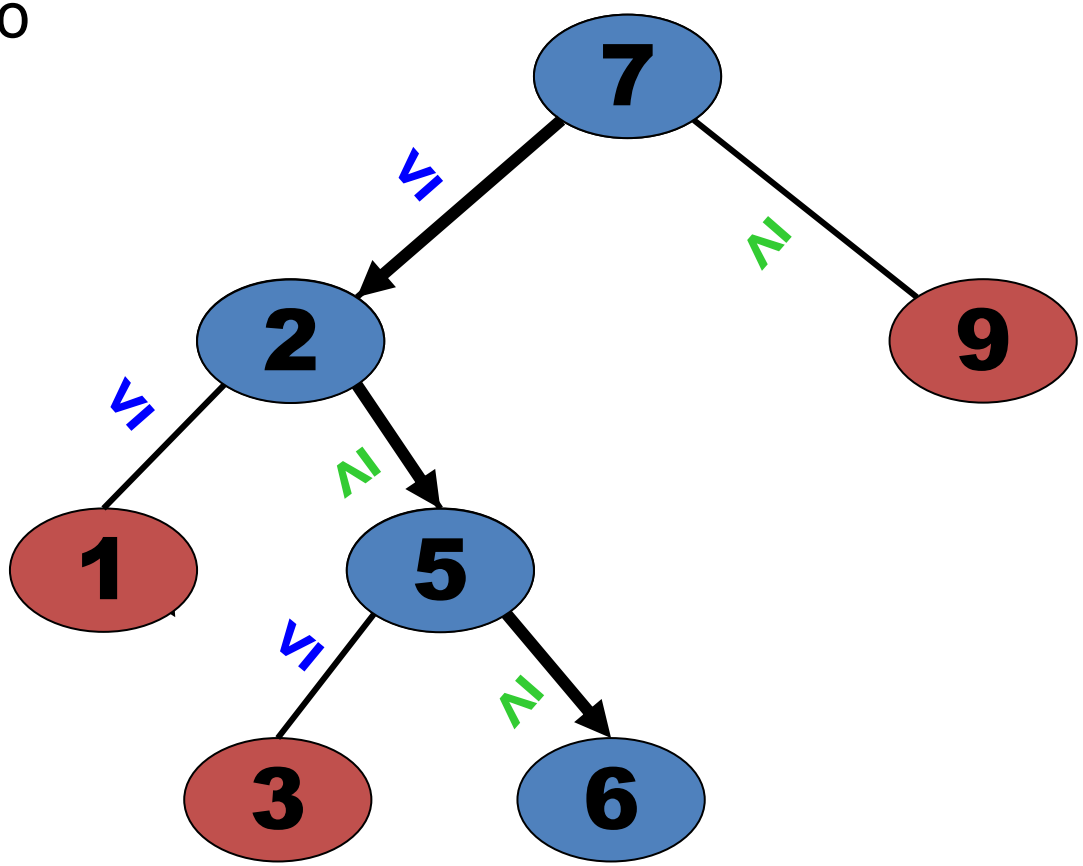


http://www.cosc.canterbury.ac.nz/mukundan/dsal/BST.html

Operación: Inserción

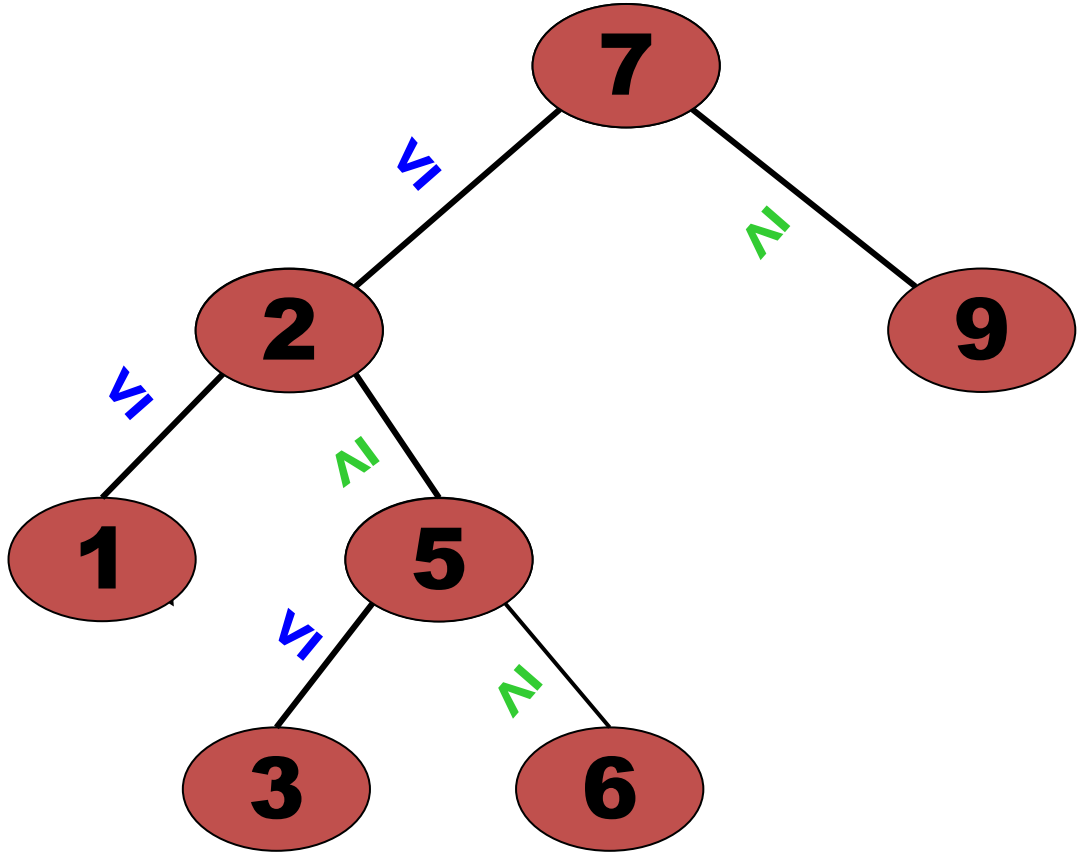
Insertar "6":

- 6<7: Subárbol izquierdo
- 6>2: Subárbol derecho
- 6>5: Subárbol derecho
- Hueco libre: insertar



Ejercicio 4

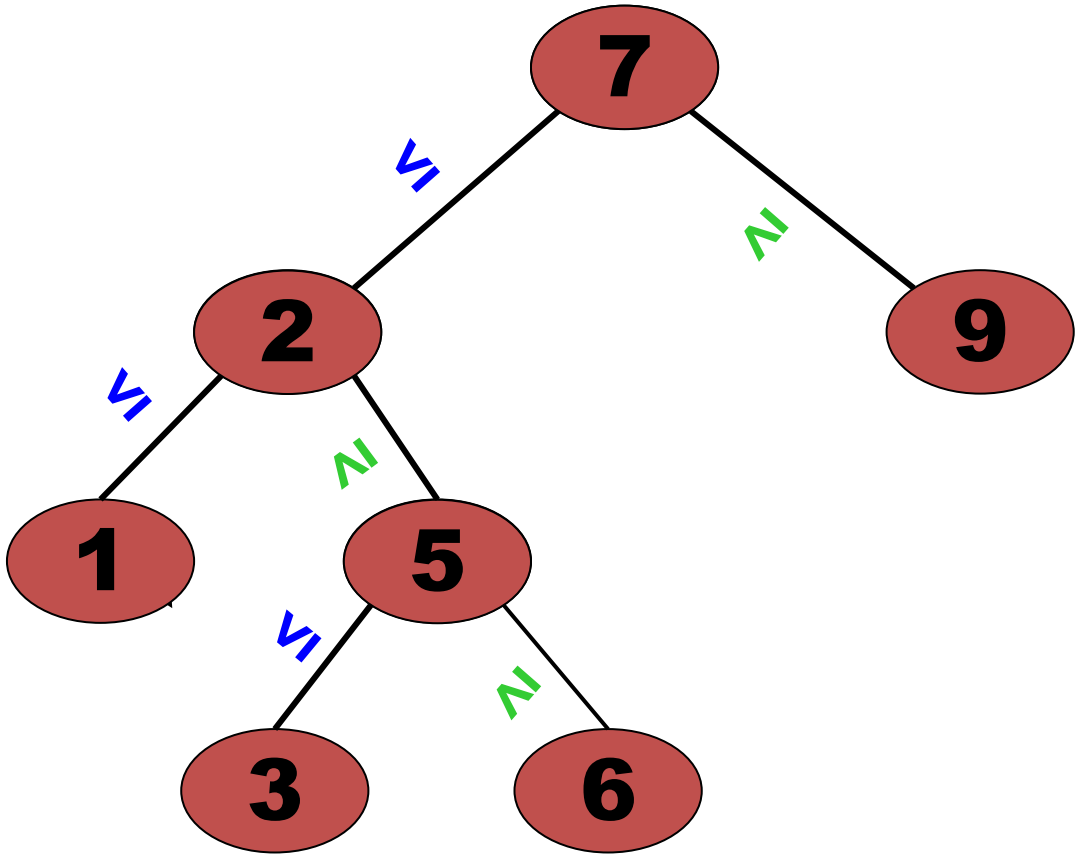
- Dado el siguiente árbol binario de búsqueda, inserta en orden tres elementos que contienen las siguientes claves: 4, 8 y 5



Operación: Eliminar (I)

Si los subárboles izquierdo y derecho están vacíos ("hoja")

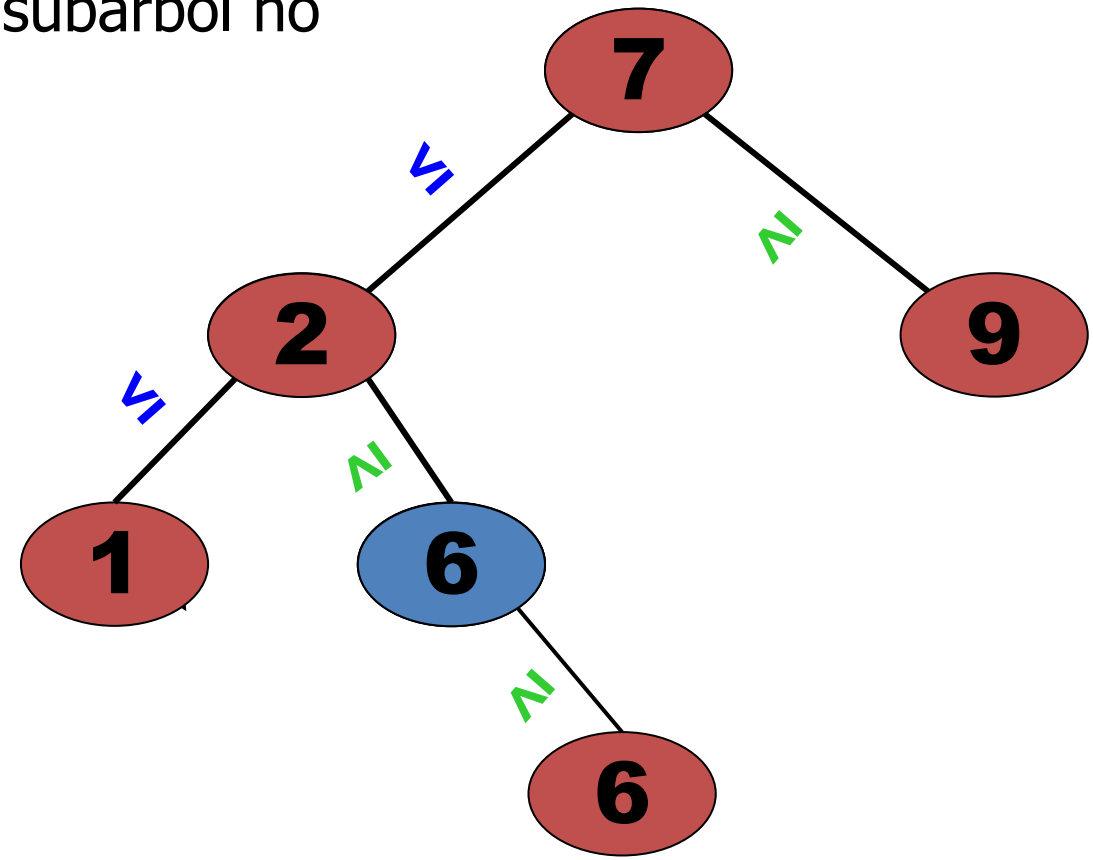
- Eliminar directamente
- Por ejemplo, eliminar "3"



Operación: Eliminar (II)

Si uno de los dos subárboles está vacío

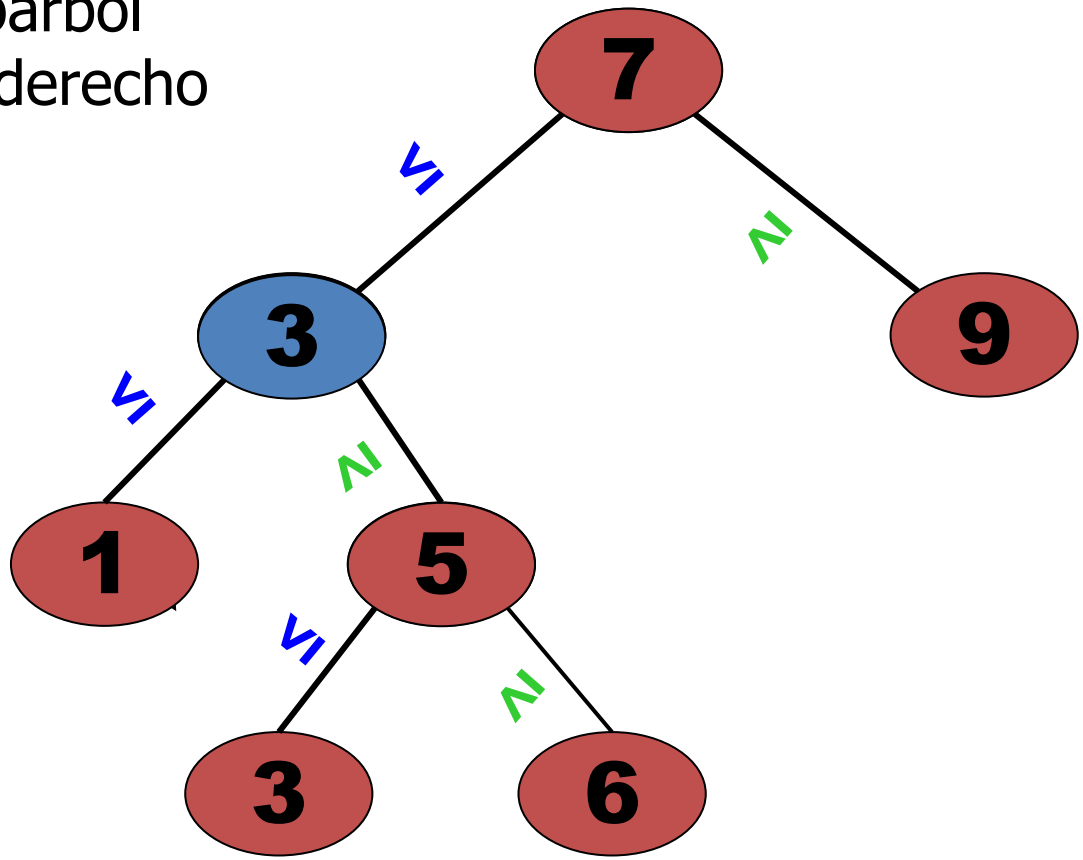
- Reemplazar por el nodo raíz del subárbol no vacío
- Por ejemplo, eliminar "5"



Operación: Eliminar (III)

Si ninguno de los subárboles está vacío

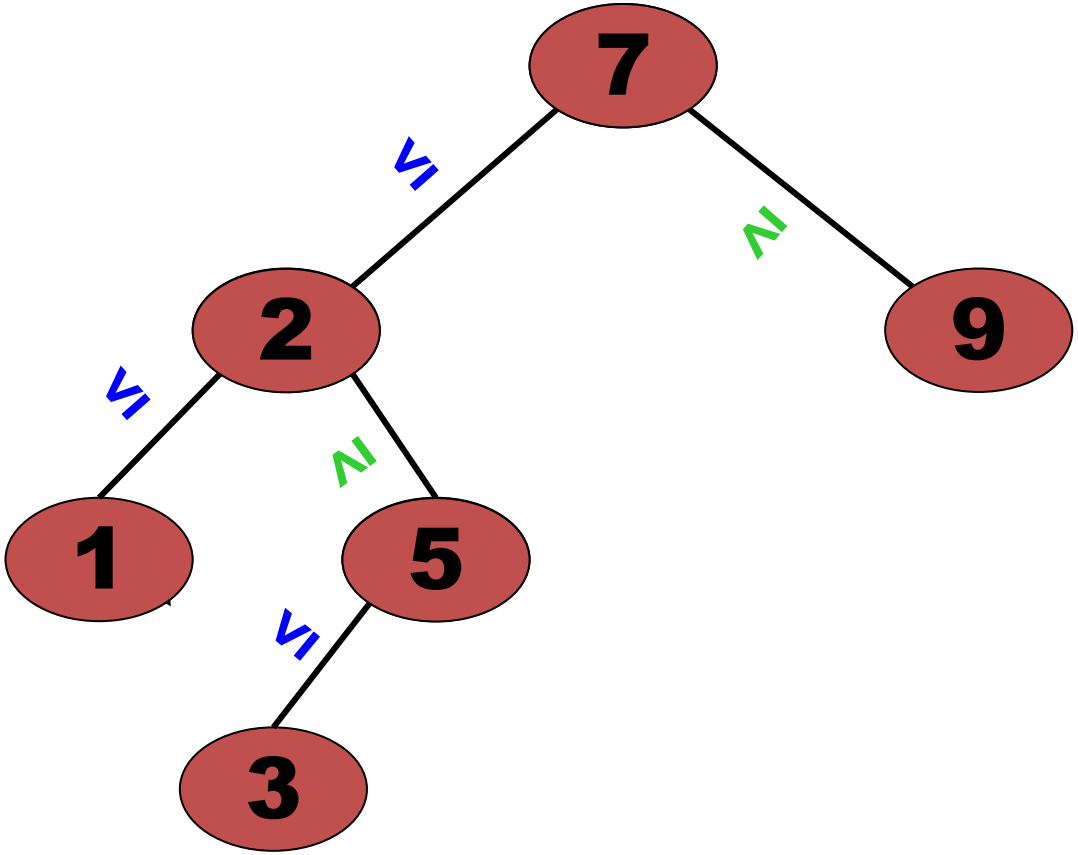
- Reemplazar por el mayor del subárbol izquierdo o el menor del subárbol derecho
- Por ejemplo, eliminar "2"
  - Reemplazar por "1" o por "3"



http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/BST-Example.html

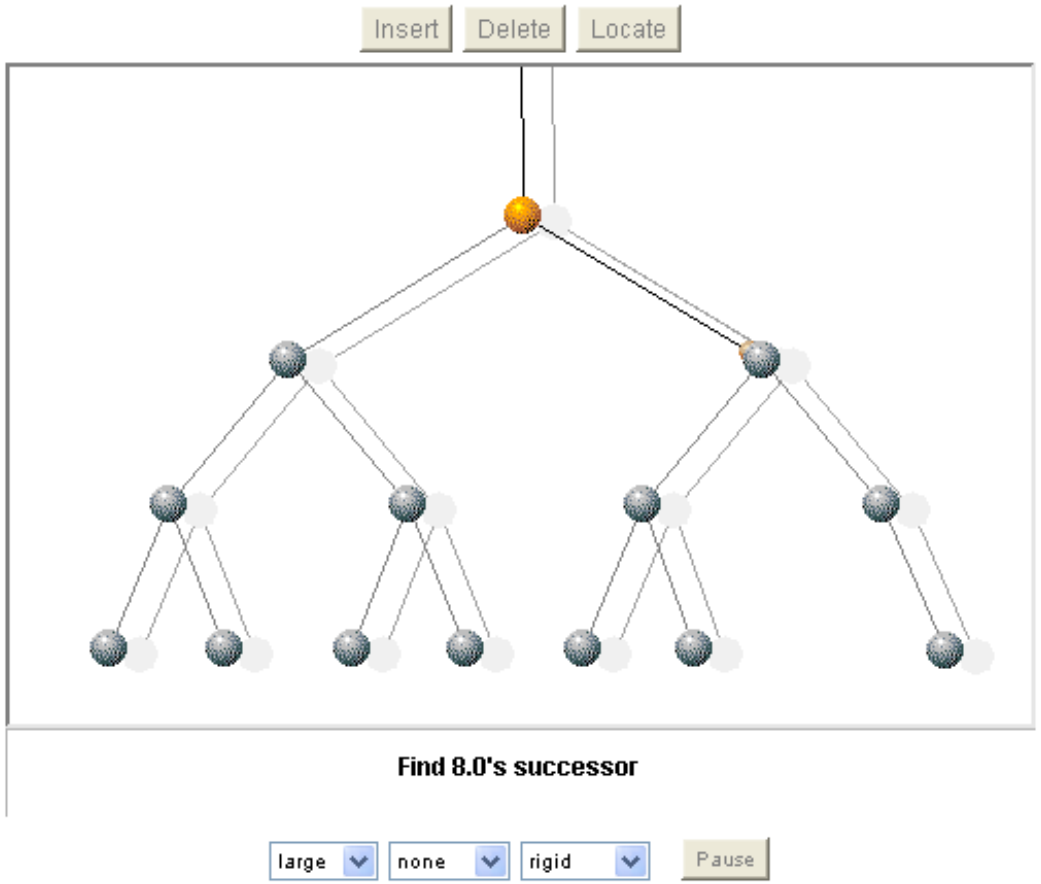
Ejercicio 5

- Dado el siguiente árbol binario de búsqueda, elimina el elemento cuya clave es 7. Propón dos formas de realizar dicha operación



Actividad

http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/BST-Example.html



Montículos (*heaps*)

- Un **montículo** (binario)* es un árbol binario *completo* en el que cada nodo tiene una clave mayor** o igual que la de su padre.

* normalmente se sobreentiende montículo *binario*
** también podría definirse como menor o igual (el orden es arbitrario)

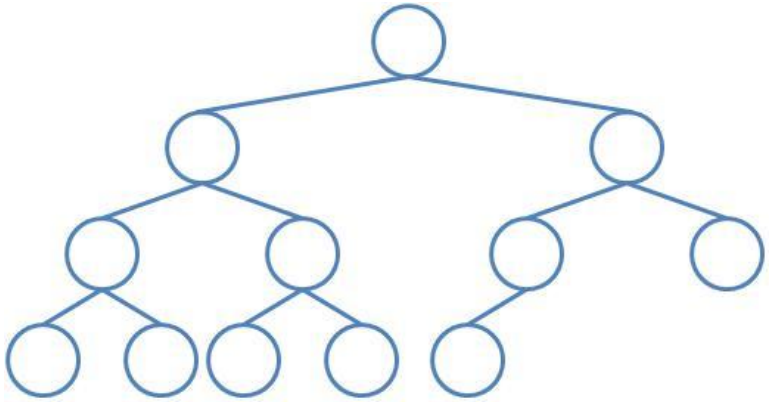
Aplicaciones:

- Colas con prioridad
- Ordenación

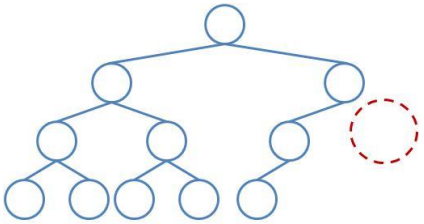
Montículos: propiedades

1. Para cada nodo *n* (excepto para el raíz), su clave es mayor o igual que la de su padre.
2. Completo

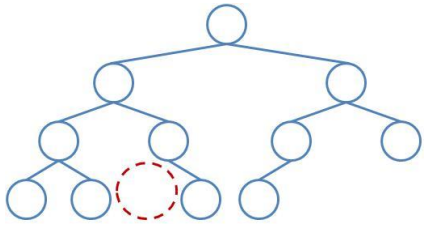
Para una profundidad K, todos los subárboles hasta K-1 están no vacíos y en K los árboles no vacíos están colocados de izquierda a derecha



Completo

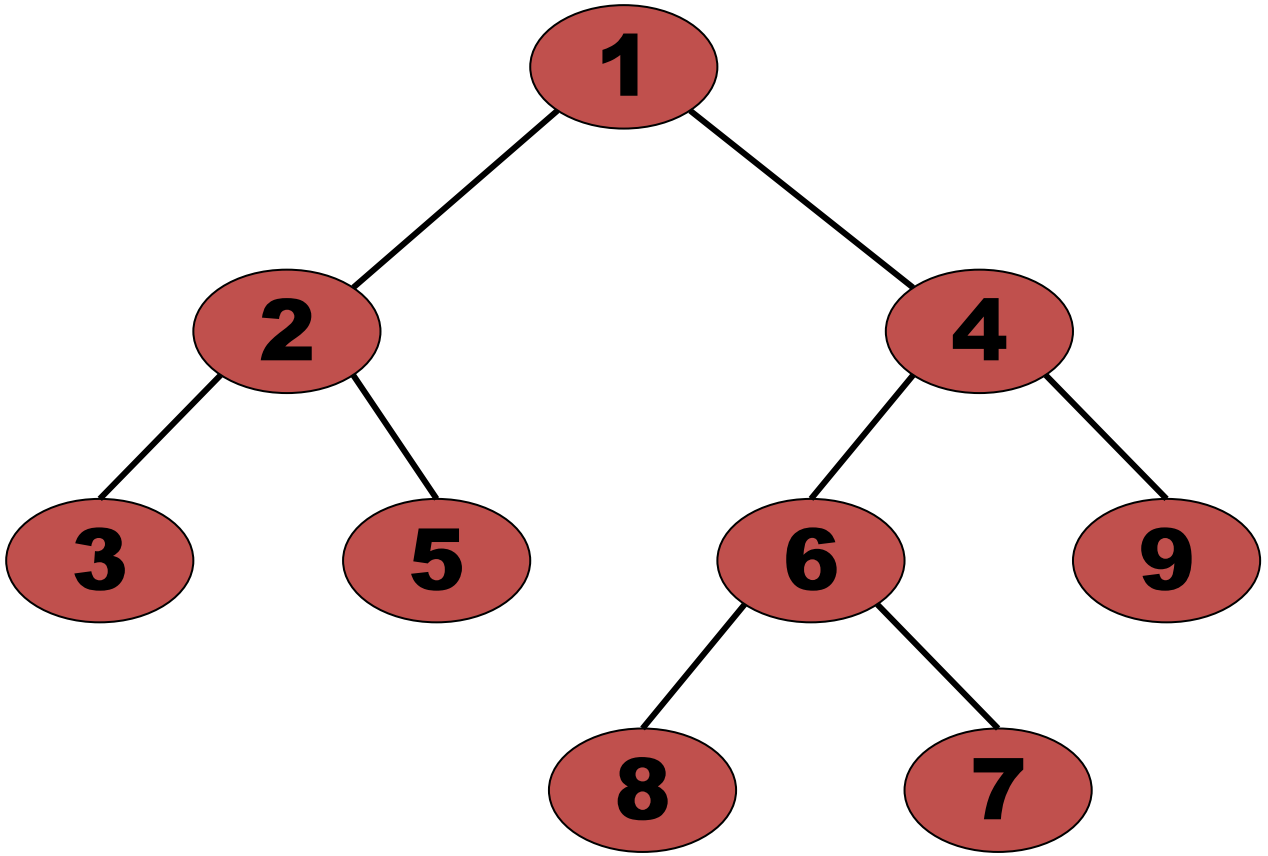


No Completo



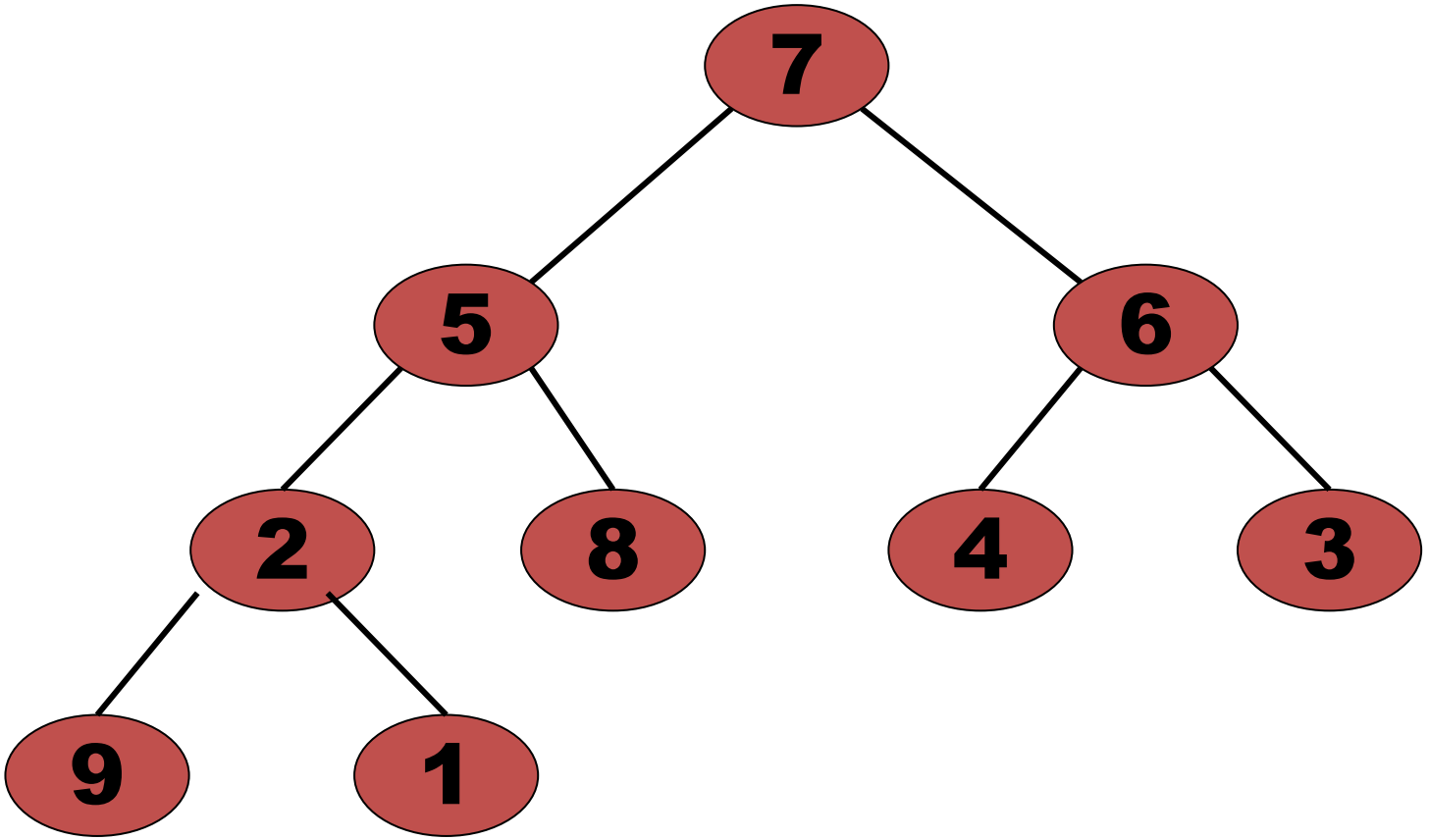
No Completo

Ejemplo (1)



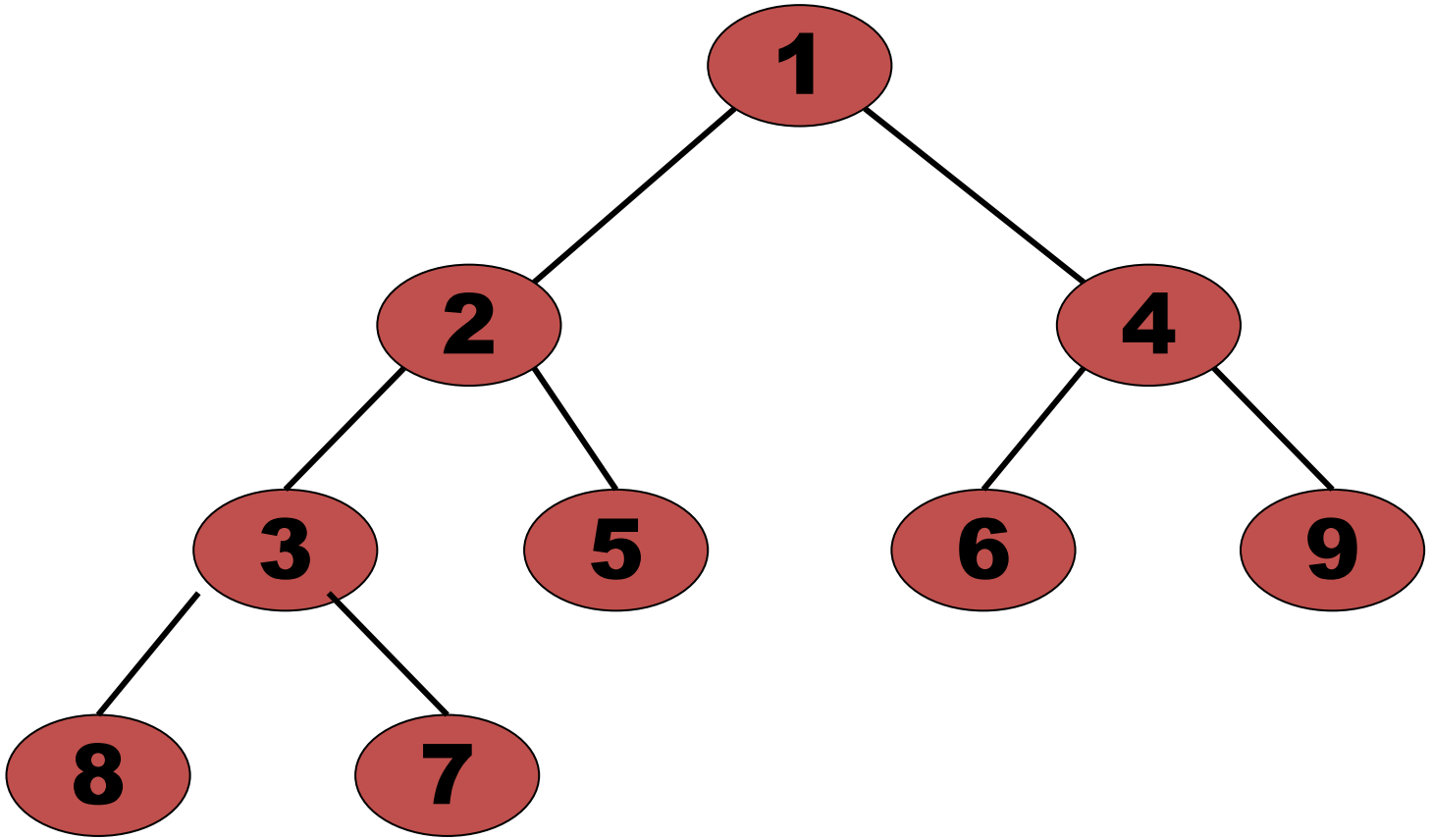
No es un montículo. Cumple la propiedad 1 pero no la 2

Ejemplo (2)



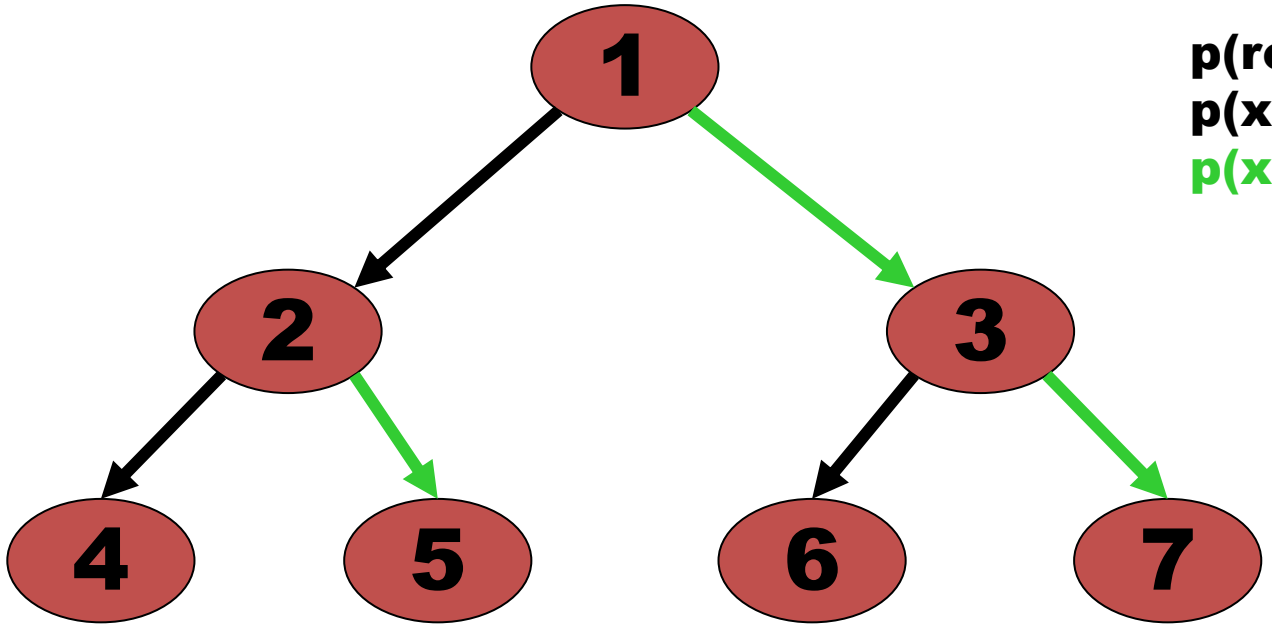
No es un montículo. Cumple la propiedad 2 pero no la 1

Ejemplo (3)

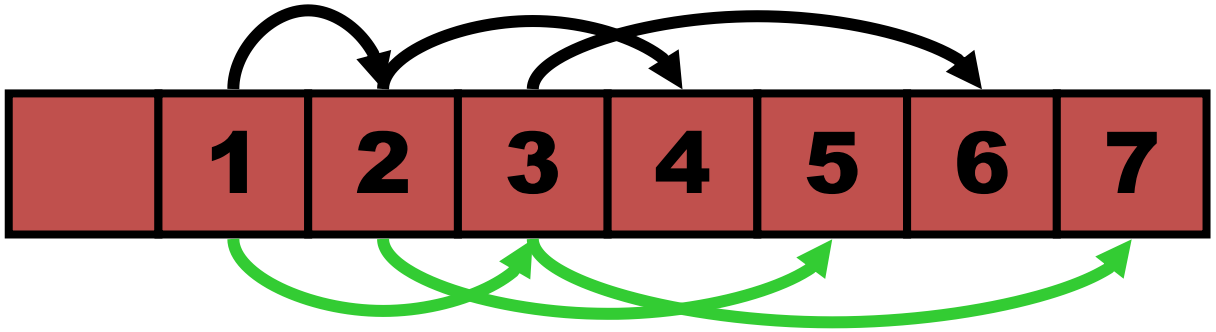


Sí es un montículo

Implementación basada en secuencias

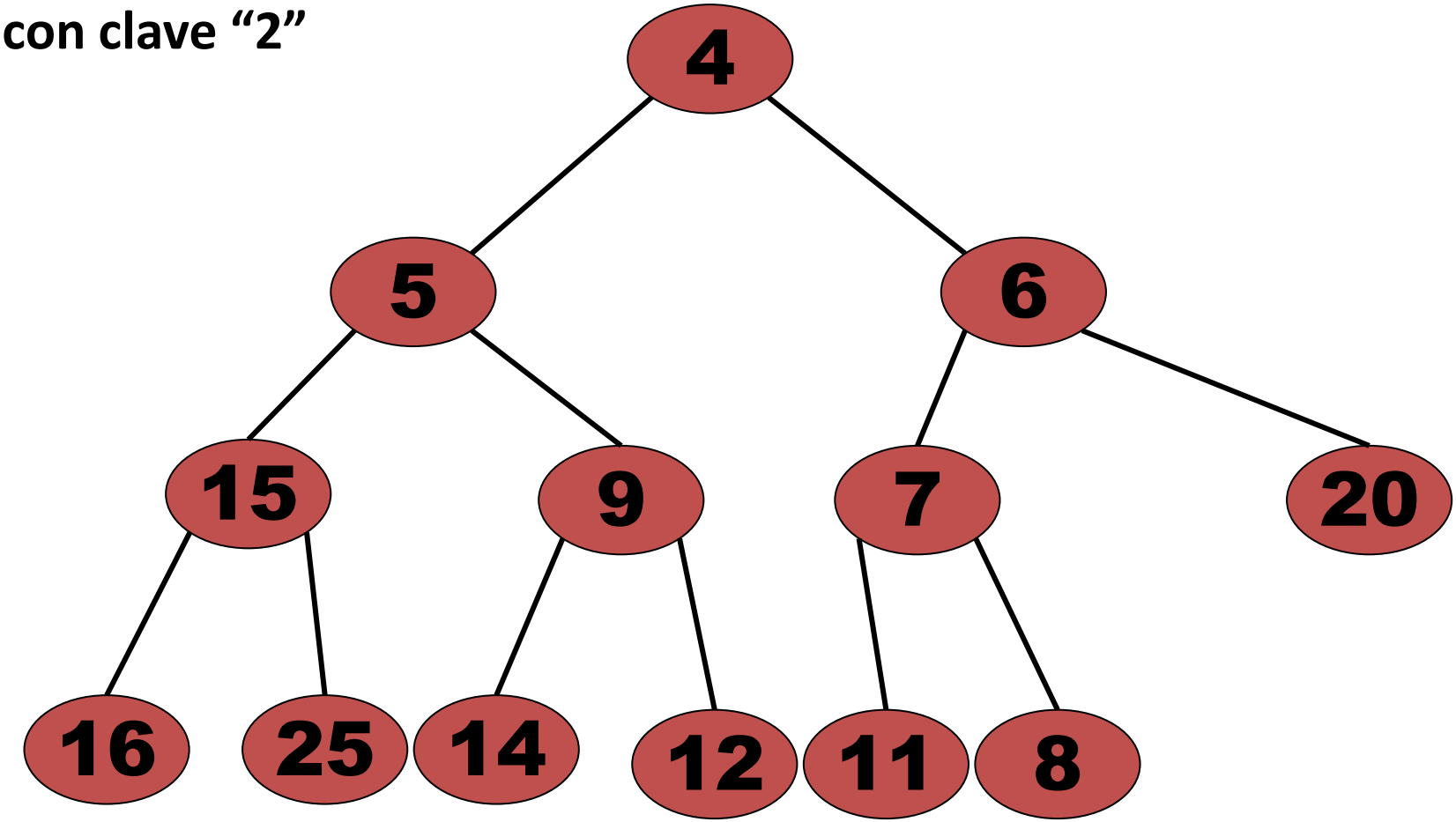


p(root)=1
p(x.left)=2*p(x)
p(x.right)=2*p(x)+1



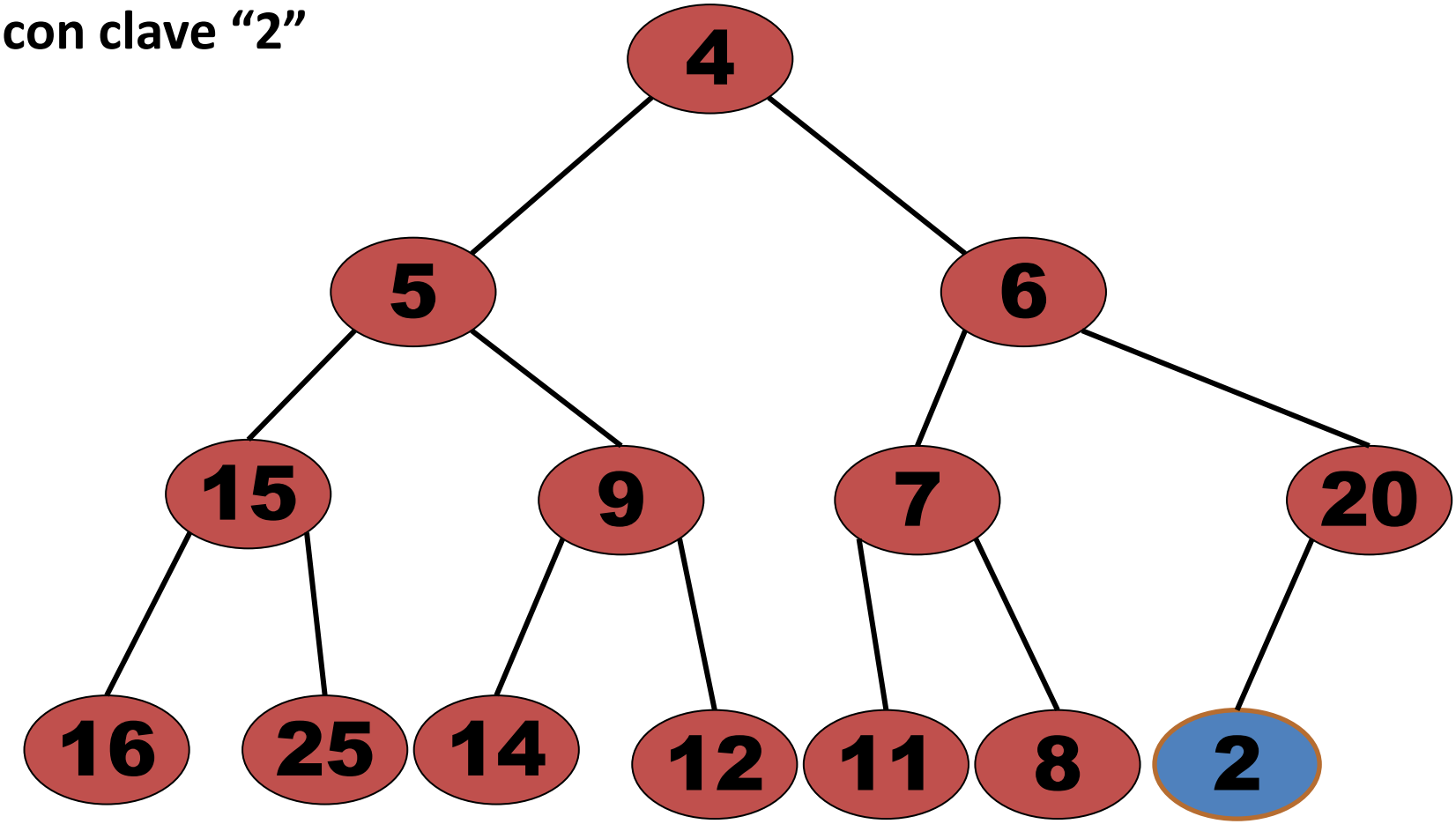
Insertar

Insertar elemento con clave “2”



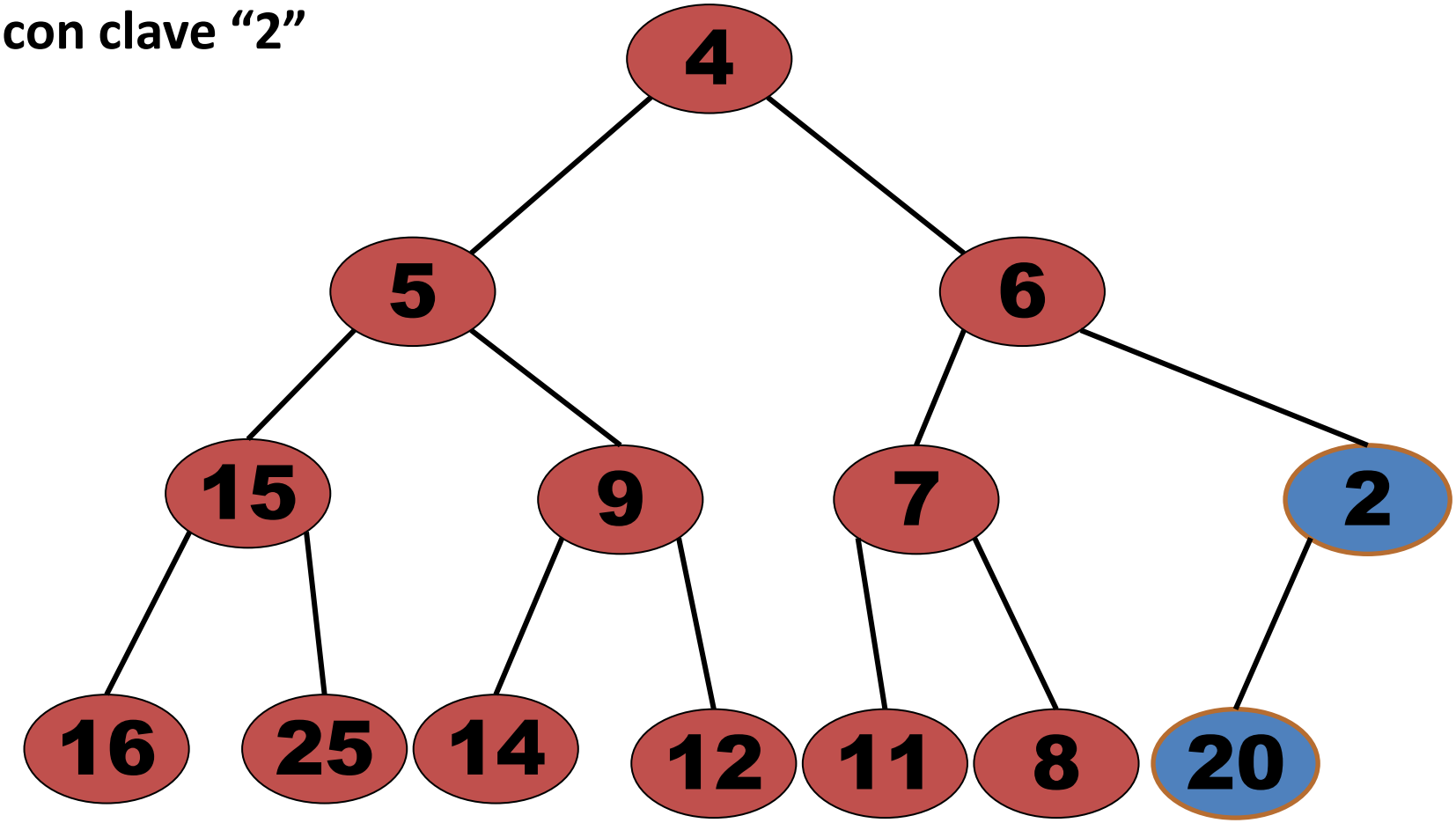
Insertar

Insertar elemento con clave “2”



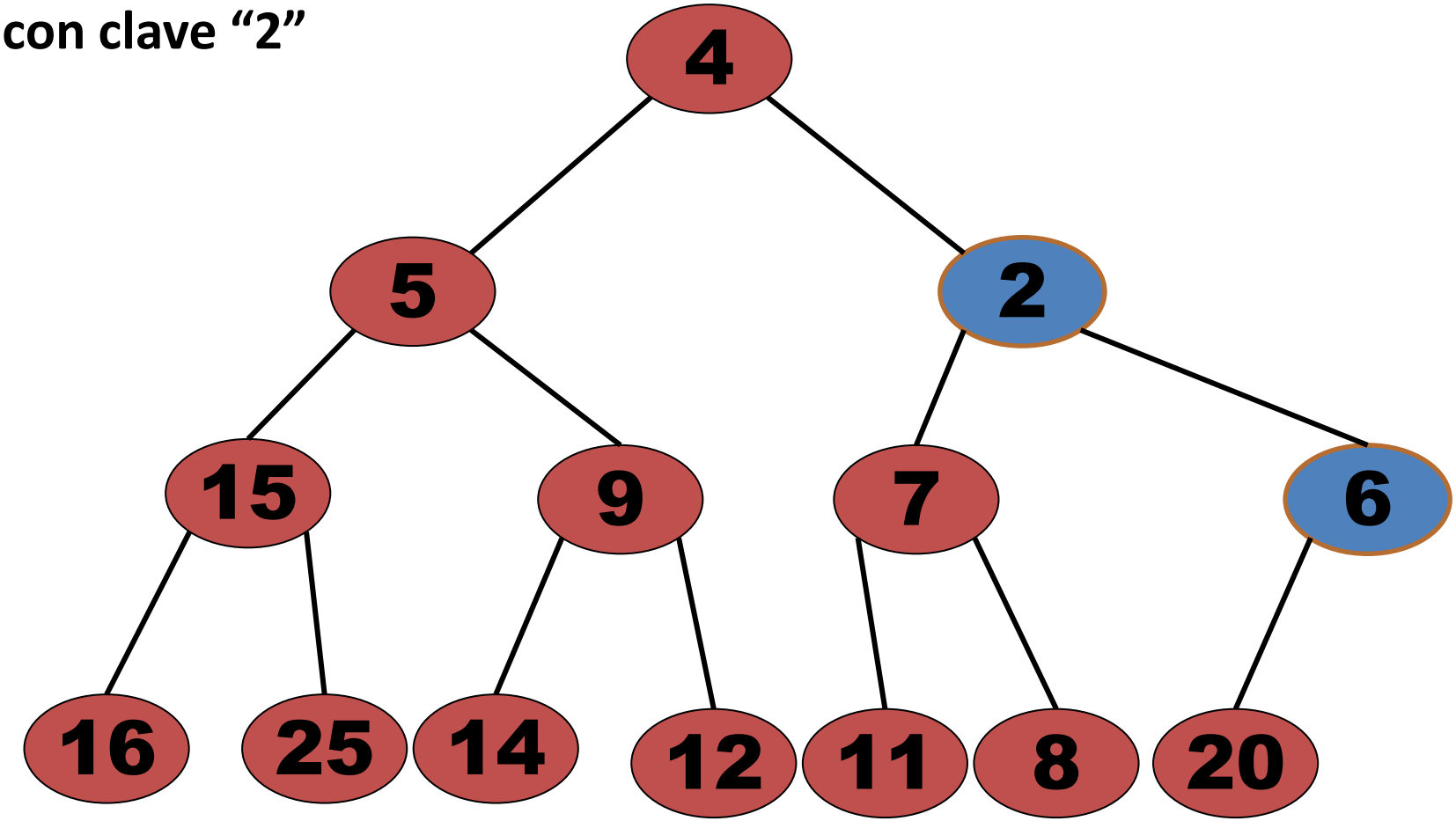
Insertar

Insertar elemento con clave “2”



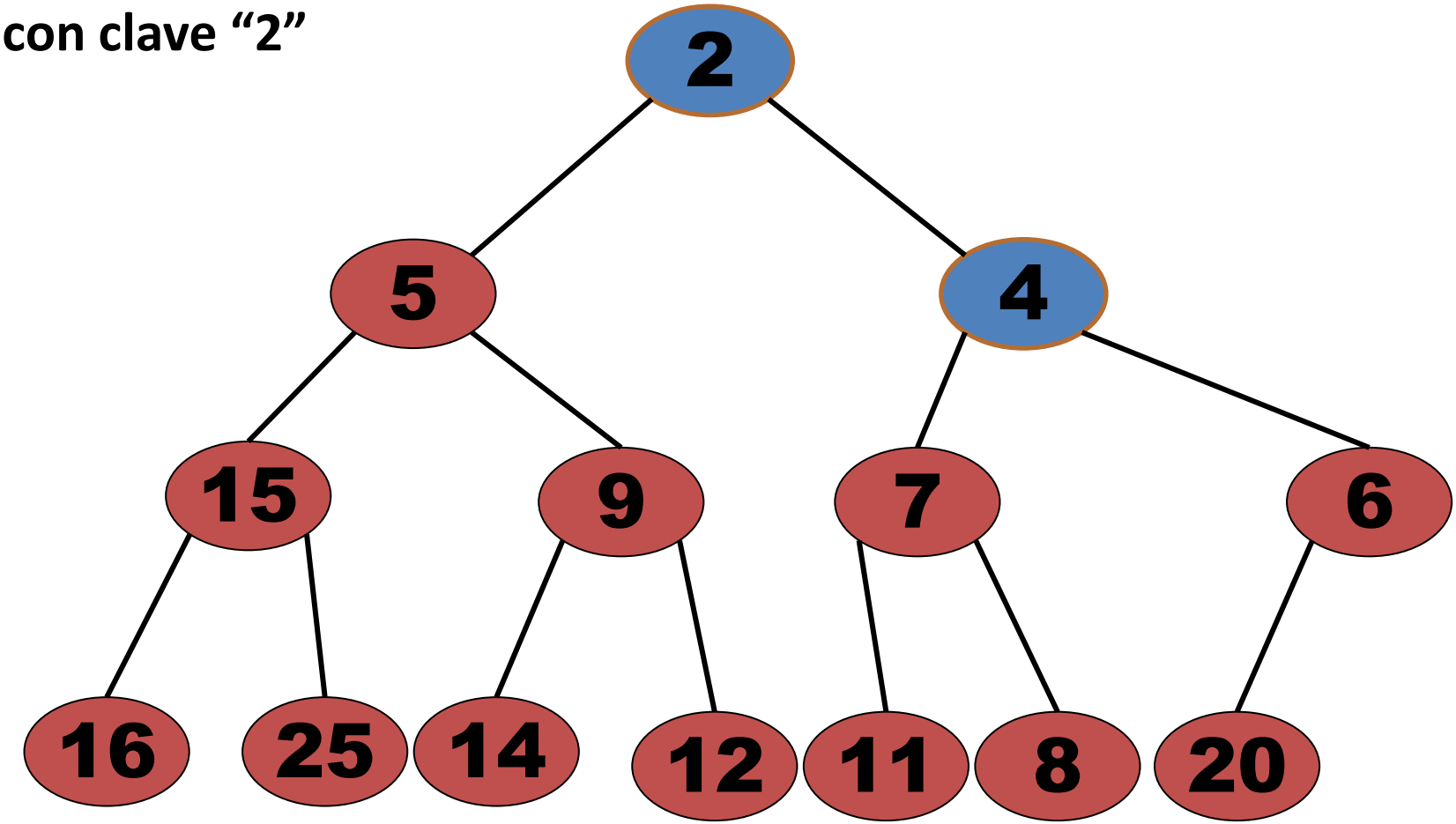
Insertar

Insertar elemento con clave “2”



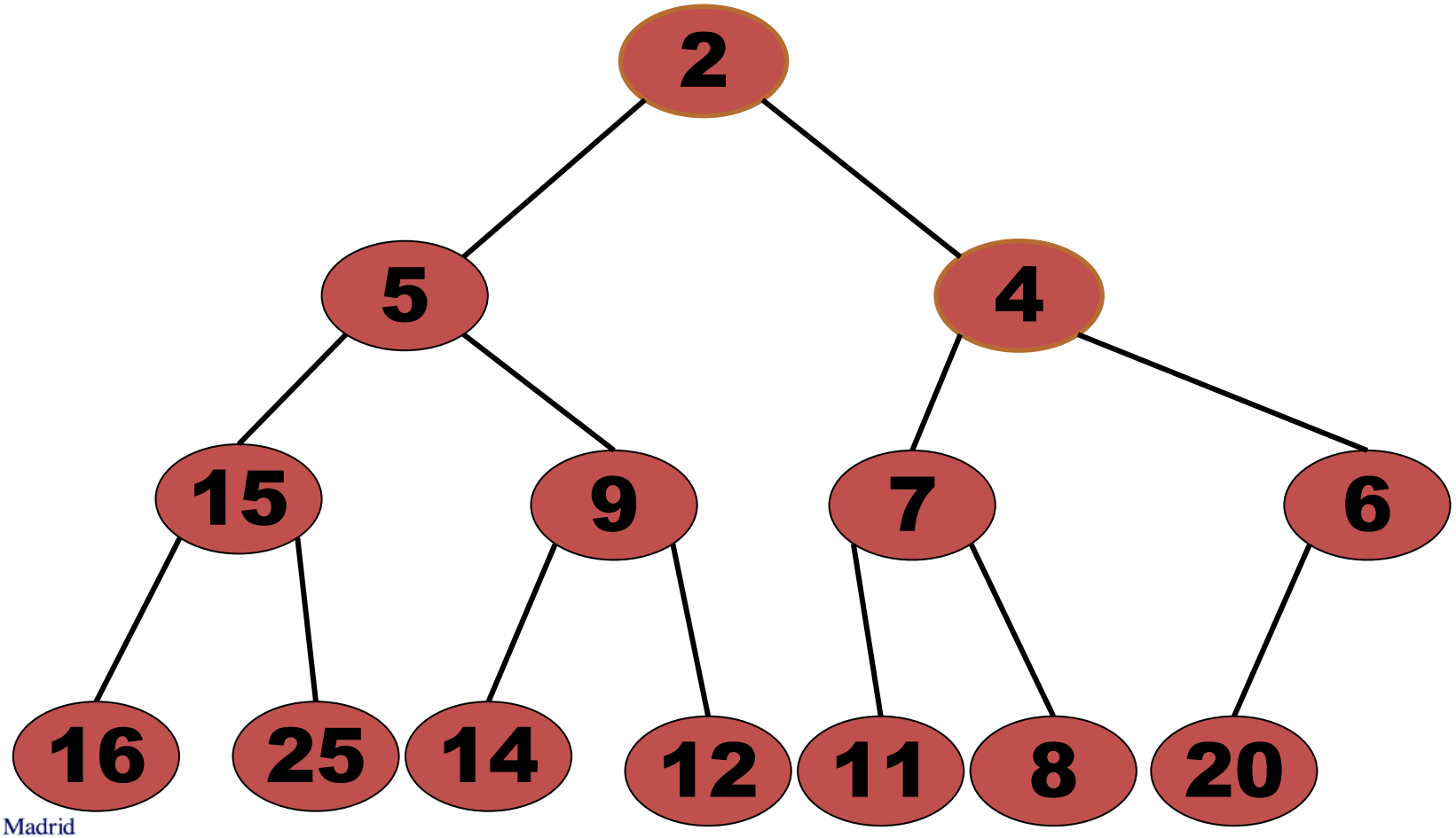
Insertar

Insertar elemento con clave “2”



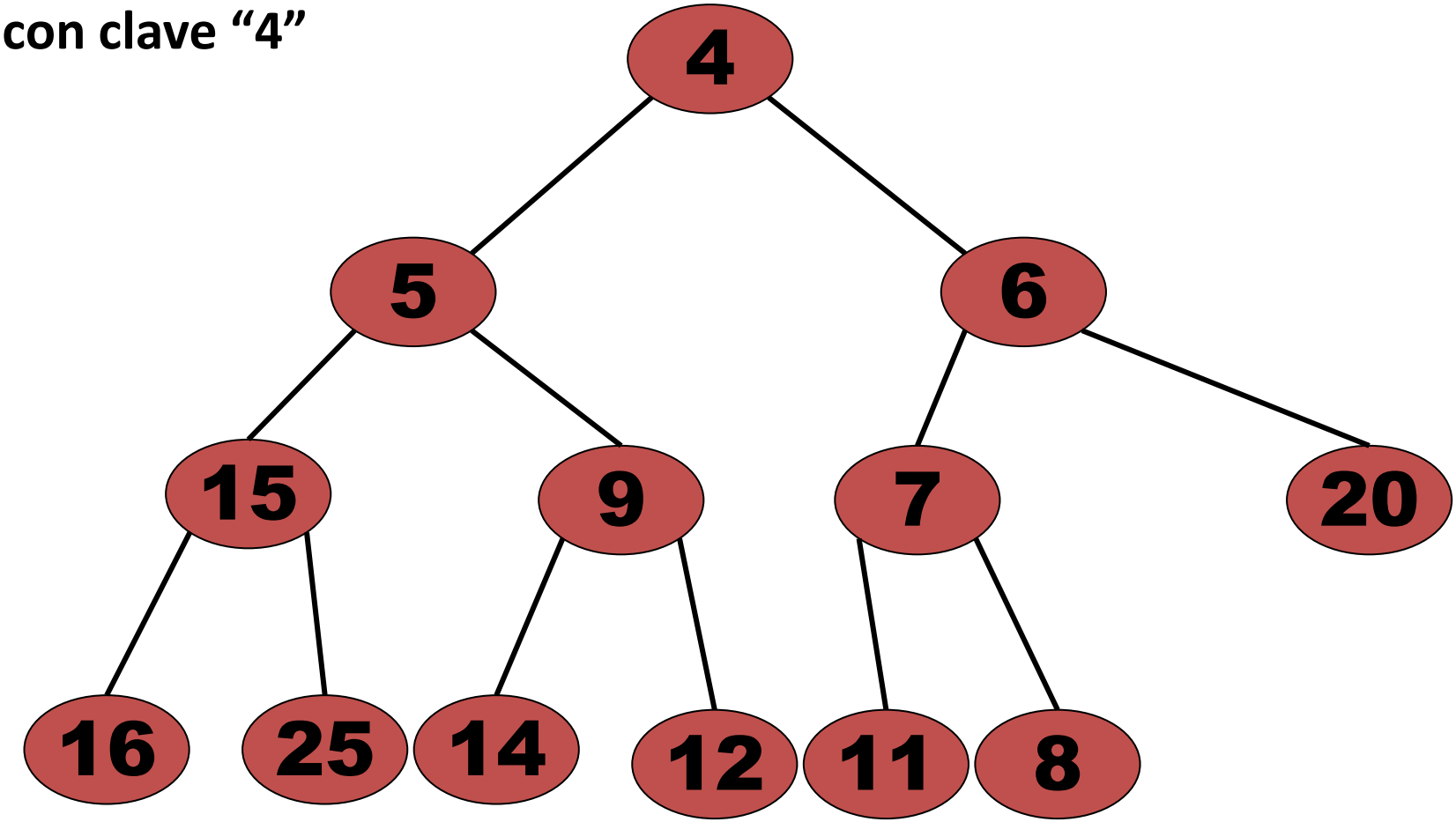
Ejercicio 6

- Dado el siguiente montículo, inserta tres elementos con claves 3, 8 y 1.

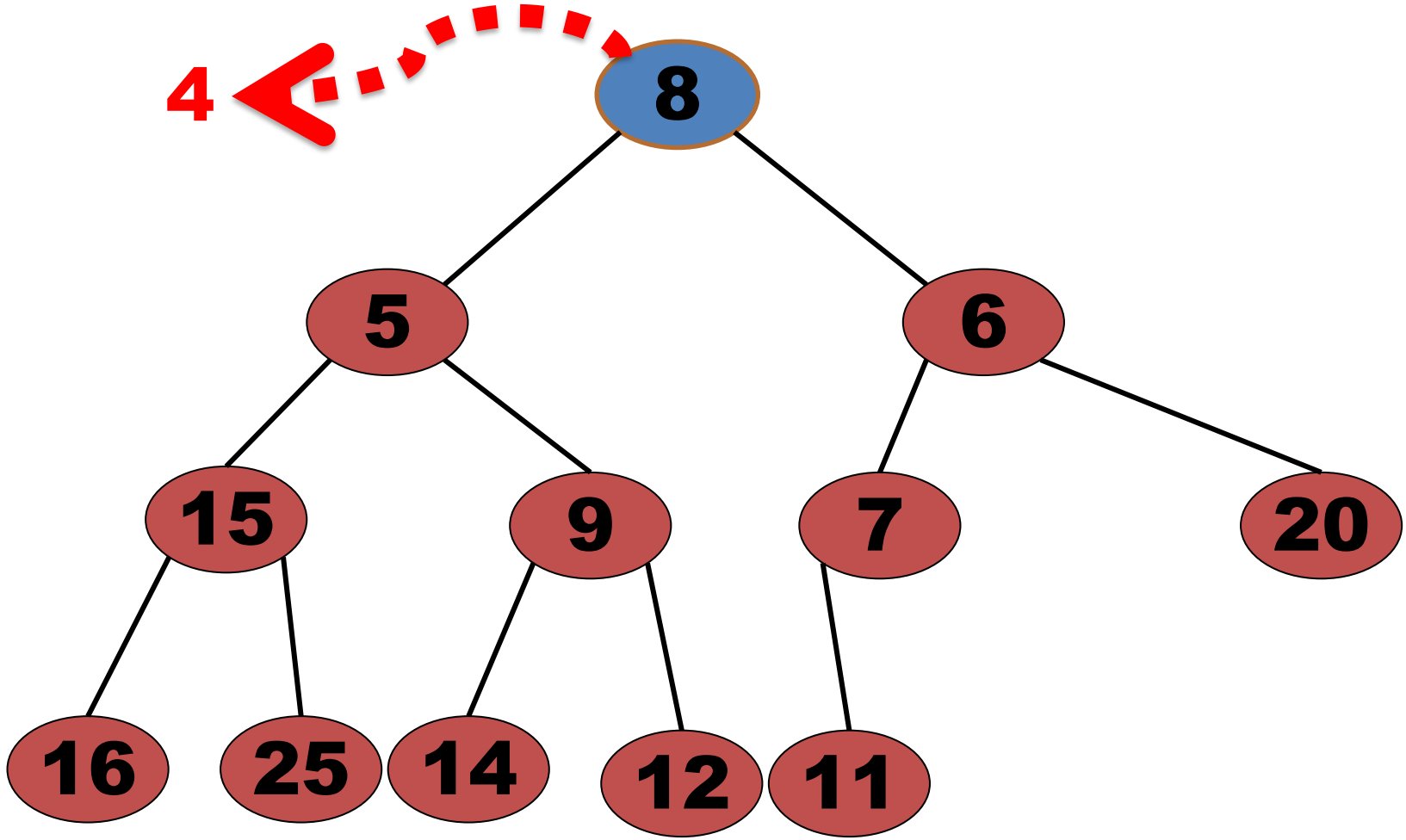


Eliminar

Eliminar elemento con clave “4”

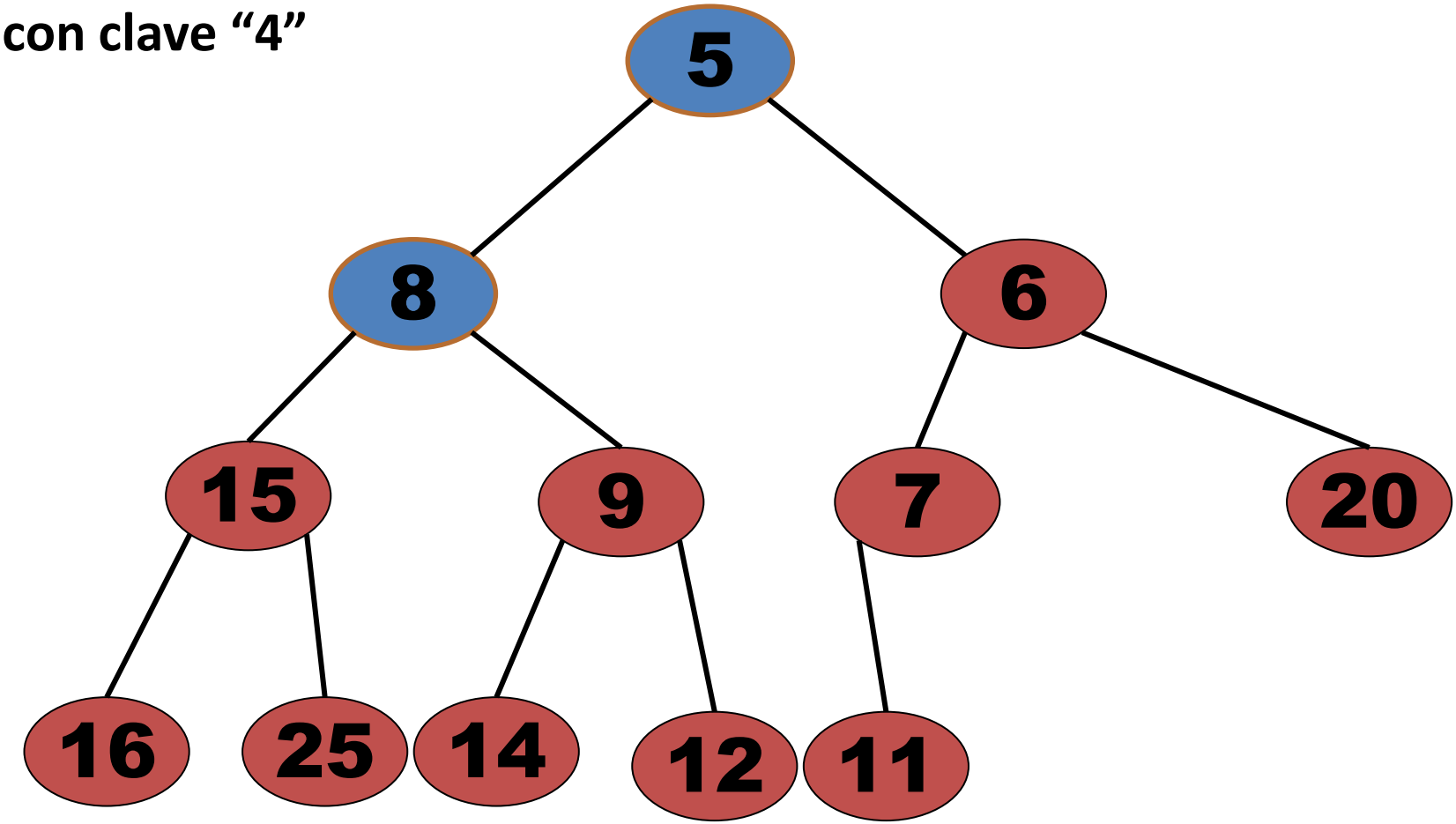


Eliminar



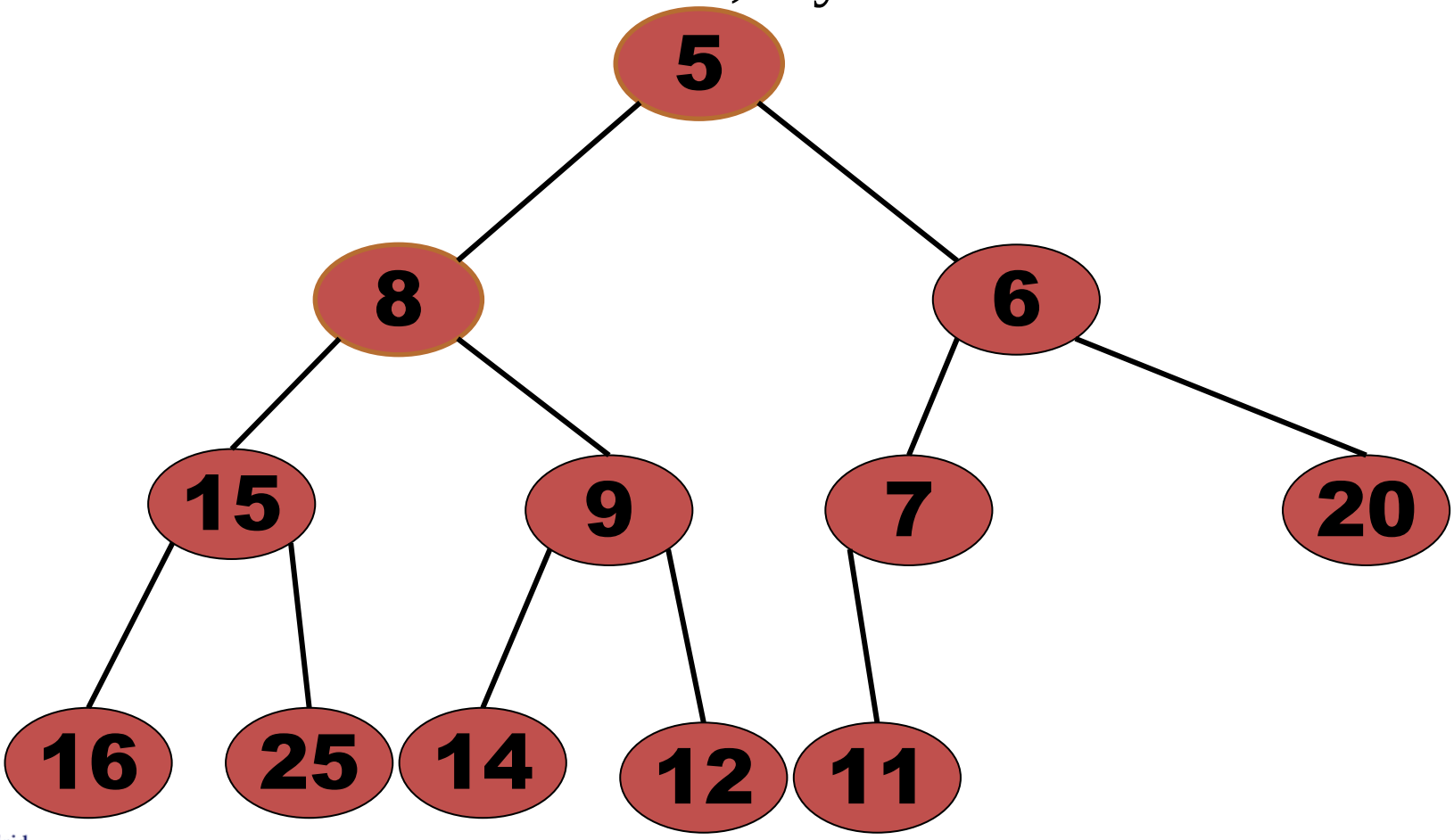
Eliminar

Eliminar elemento con clave “4”

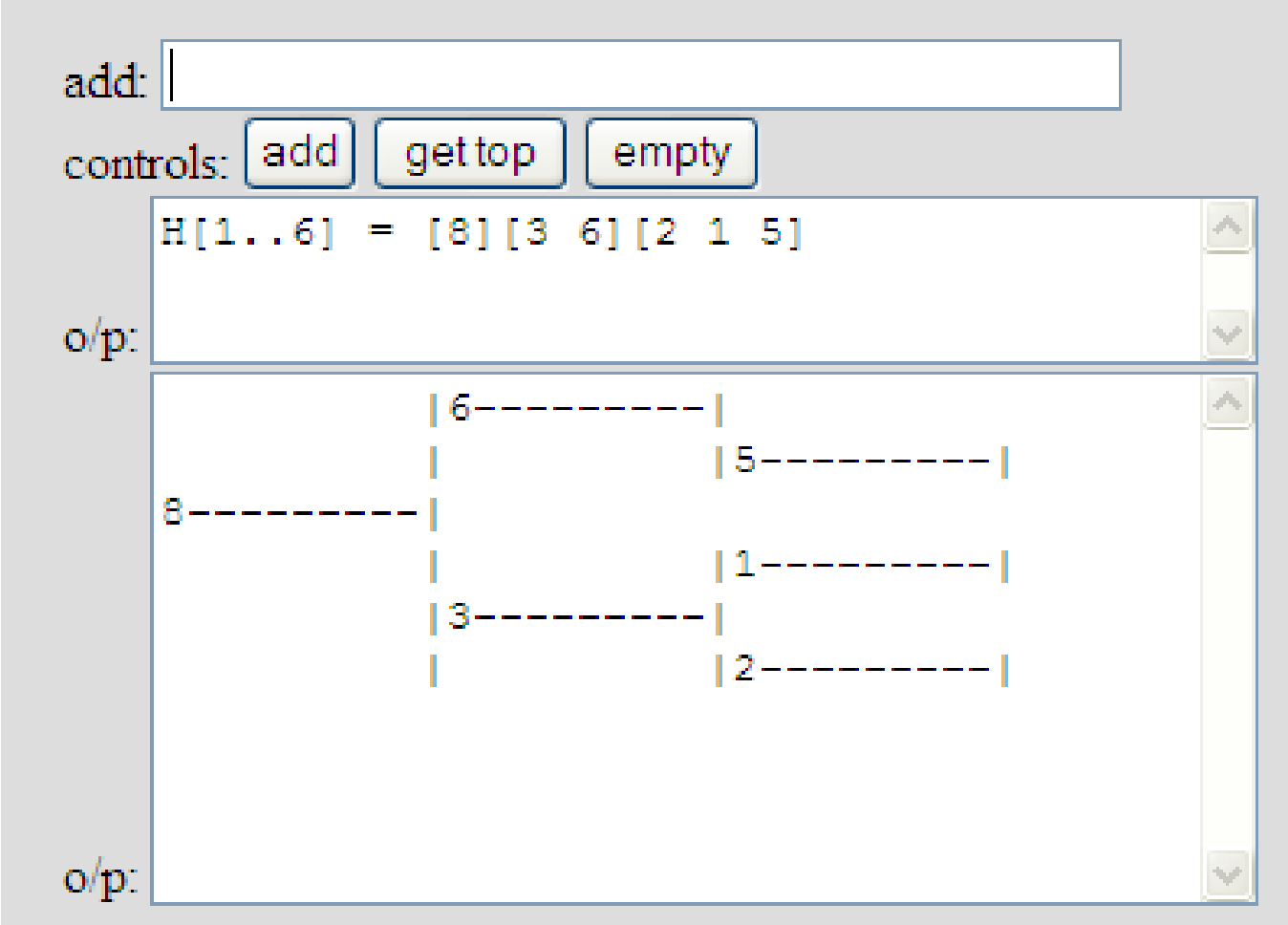


Ejercicio 7

- Dado el siguiente montículo, elimina tres elementos con claves 15, 5 y 7.

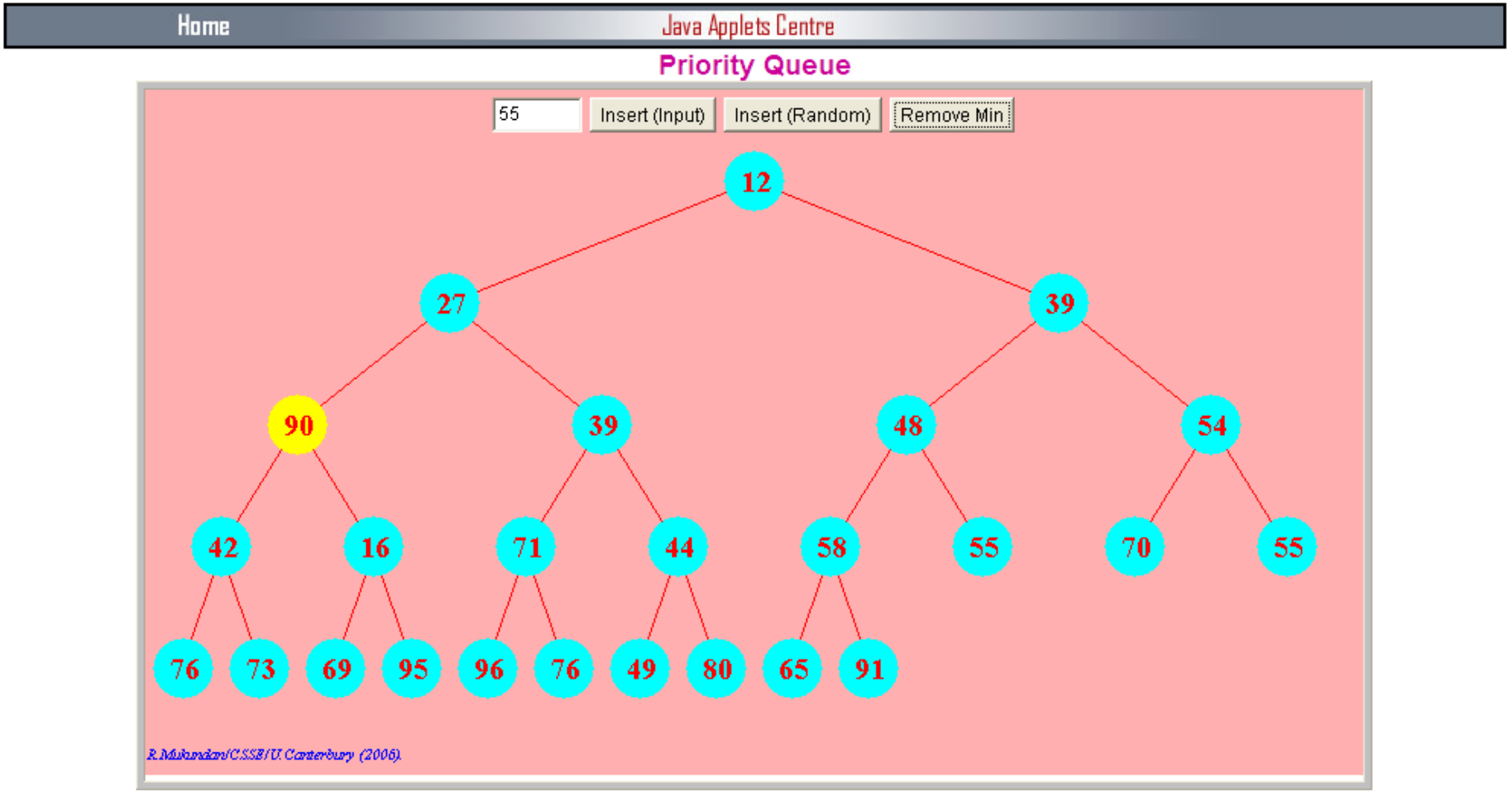


Ejemplo
Montículo - Cola con prioridad



http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Priority-Q/

Ejemplo
Montículo - Cola con prioridad



http://www.cosc.canterbury.ac.nz/mukundan/dsal/MinHeapAppl.html