

# Tema 4

## Map Ordenado y Árbol Binario de Búsqueda (ABB)

1. Conceptos de árboles
2. Árbol Binario de Búsqueda (Equilibrado)
  - 2.1. Estudio del Árbol Binario y las condiciones de Búsqueda eficiente sobre él
  - 2.2. Estudio del Árbol Binario de Búsqueda (ABB) y la relación entre su eficiencia y grado de equilibrio
3. Implementación de un ABB: las clases ABB y NodoABB
  - 3.1. Representación en memoria: atributos de las clases
  - 3.2. Implementación de sus operaciones: diseño y análisis del coste de los métodos de la clase ABB
4. El modelo Map Ordenado: definición, coste estimado y ejemplos de uso
5. Implementaciones

# Bibliografía

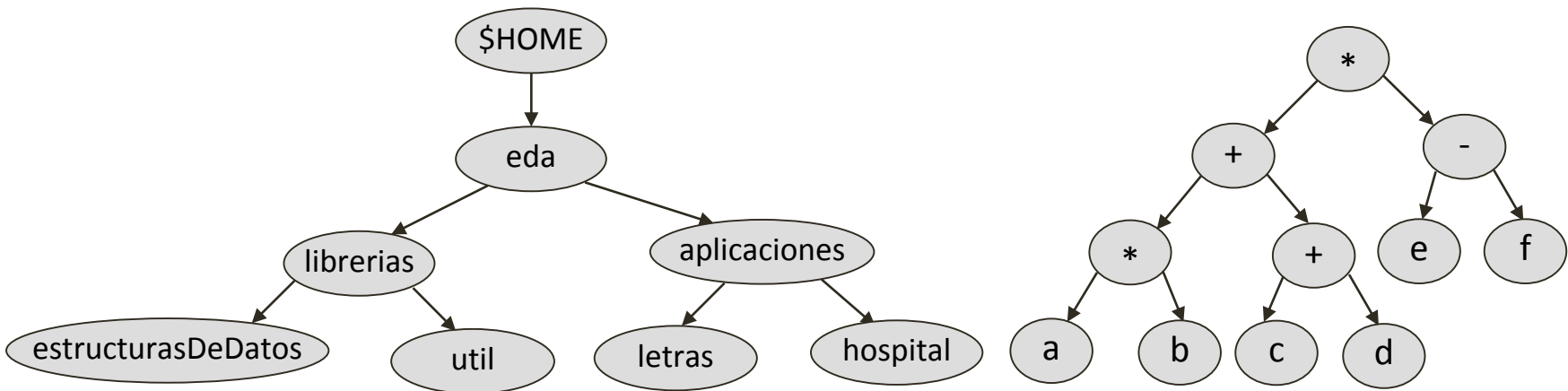
- Weiss, M.A. Estructuras de Datos en Java. Addison-Wesley, 2000. Capítulo 18, apartados del 1 al 3, ambos inclusive.
- Sahni, S. Data Structures, Algorithms, and Applications in Java. McGraw-Hill, 2000. Capítulo 15.
- Michael T. Goodrich and Roberto Tamassia. Data Structures and Algorithms in Java (4th edition). John Wiley & Sons, Inc., 2010. Capítulos 9, apartado 9.5.2, y Capítulo 10.

# 1. Conceptos de árboles - *Modelos Lineales vs. Modelos Jerárquicos*

- Con una Representación Lineal de los datos (solo) podemos representar relaciones muy simples entre ellos: [elSiguiente](#) o [elAnterior](#)
- Con una Representación Jerárquica o en Árbol de los datos podemos representar relaciones más complejas, jerárquicas, entre ellos: [padreDe](#), [hijoDe](#), [ascendienteDe](#), [descendienteDe](#), etc.

Estructura de directorios de las prácticas de EDA

Expresión aritmética  $((a*b)+(c+d))*(e-f)$



# 1. Conceptos de árboles

Un **árbol** es una estructura jerárquica que se define como:

- Un conjunto de **nodos** (uno de los cuales es distinguido como la **raíz** del árbol) y
- Un conjunto de **aristas** tal que cualquier nodo  $H$ , a excepción del raíz, está conectado por medio de una arista a un único nodo  $P$ . Se dice que  $P$  es el nodo **padre** y  $H$  el **hijo**
- ✓ Un nodo sin hijos se denomina **hoja**
- ✓ Un nodo que no es hoja se denomina **nodo interno**
- ✓ El **grado** de un nodo es el número de hijos que tiene

# 1. Conceptos de árboles - *Ejemplo*

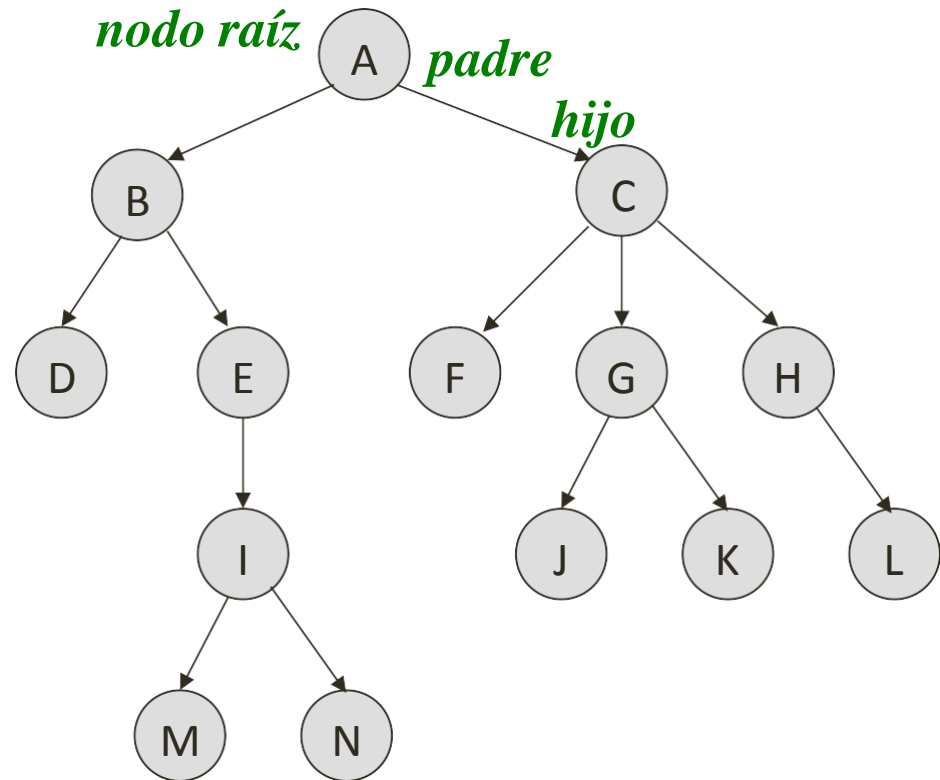
Nodo raíz: A

Nodos hoja:

{D, M, N, F, J, K, L}

Nodos internos:

{A, B, E, I, C, G, H}

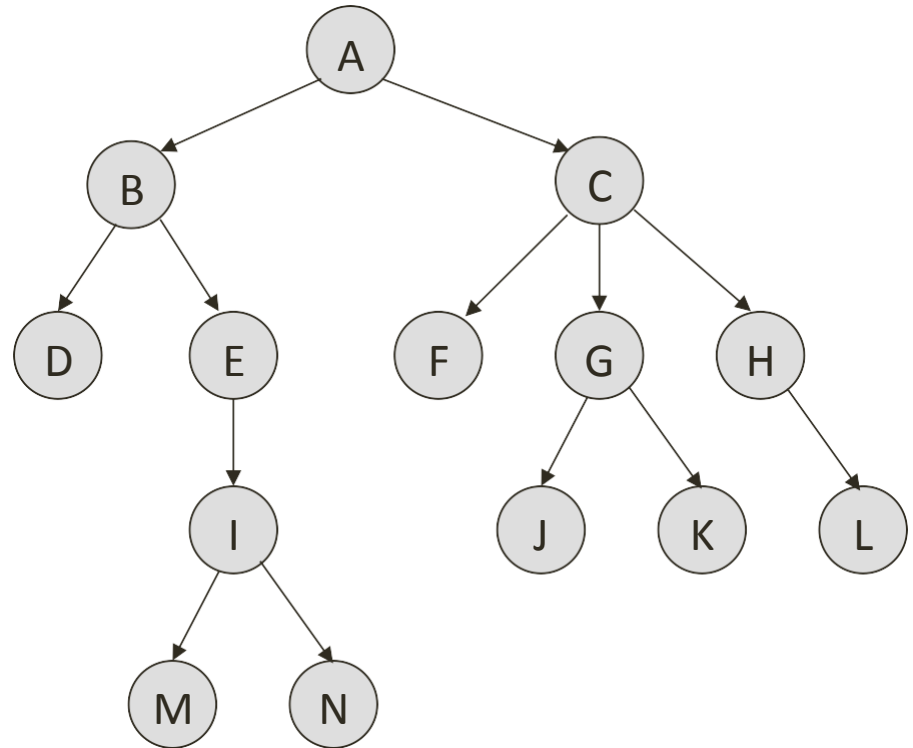


# 1. Conceptos de árboles - *Longitud, profundidad y altura*

- En un árbol hay un único **camino** desde la raíz a cada nodo
- El número de aristas que componen un camino se denomina **longitud** del camino
- **Profundidad** de un nodo: longitud del camino que va desde la raíz a ese nodo
  - La profundidad de la raíz es 0
  - Se dice que todos los nodos que están a la misma profundidad están en el mismo **nivel**
- **Altura** de un nodo: longitud del camino que va desde ese nodo hasta la hoja más profunda bajo él
- Altura de un árbol = Altura de la raíz

# 1. Conceptos de árboles

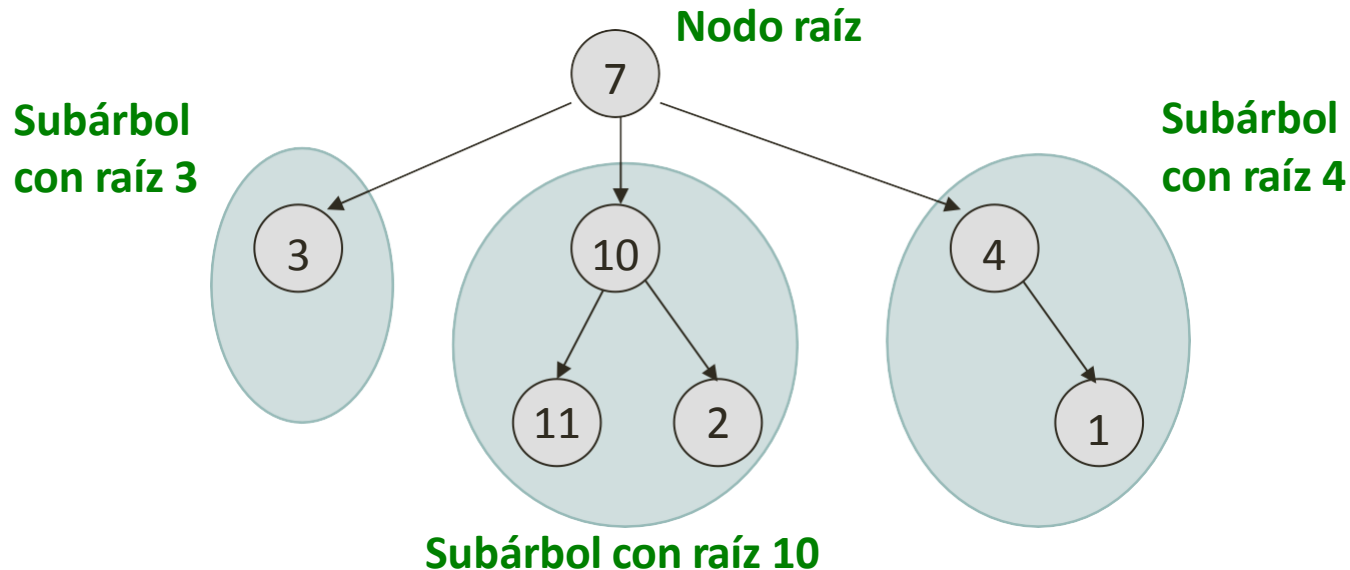
- a) ¿Cuántas aristas tiene un árbol con  $N$  nodos?
- b) ¿Longitud de A a D?
- c) ¿Longitud de C a K?
- d) ¿Longitud de B a N?
- e) ¿Longitud de B a B?
- f) ¿Profundidad de A, B, C y F?
- g) ¿Altura de B, C, I, F y del árbol?



# 1. Conceptos de árboles - *Definición recursiva de árbol*

Un árbol es:

- Un conjunto vacío (sin nodos ni aristas) , o
- Un nodo raíz y cero o más subárboles no vacíos donde cada una de sus raíces está conectada mediante una arista con el nodo raíz



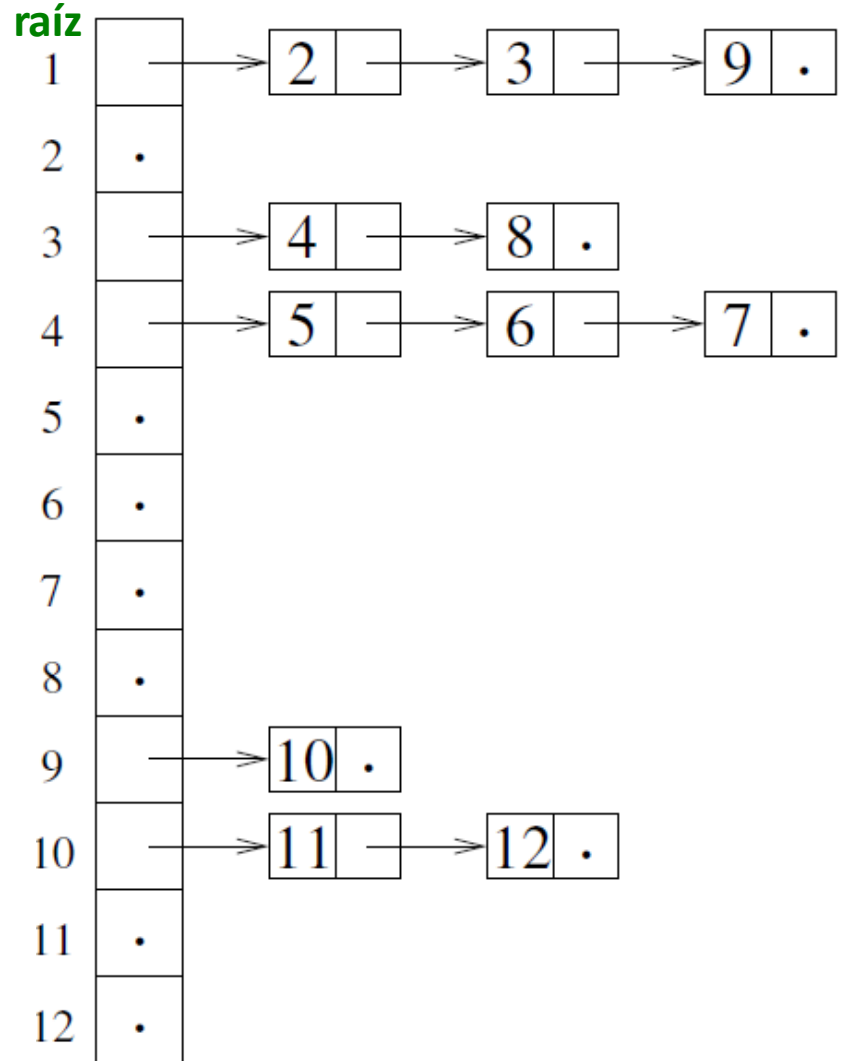
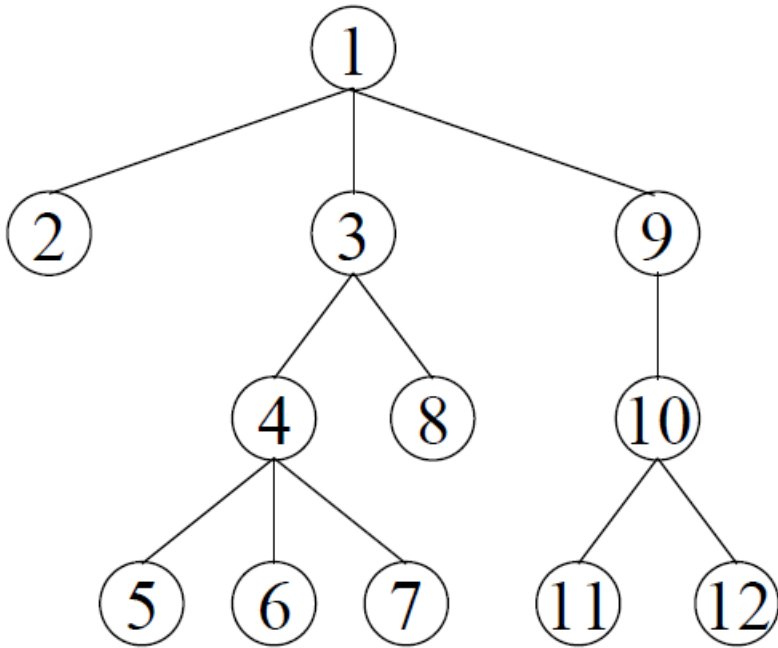


# 1. Conceptos de árboles - *Representación de árboles generales*

- Representación de árboles generales (número de hijos de los nodos sin acotar:
  - Listas (ordenadas) de hijos
  - Hijo más a la izquierda – hermano derecho
  - Con vectores y referencias al padre (*mf-sets*)
  - Otras...

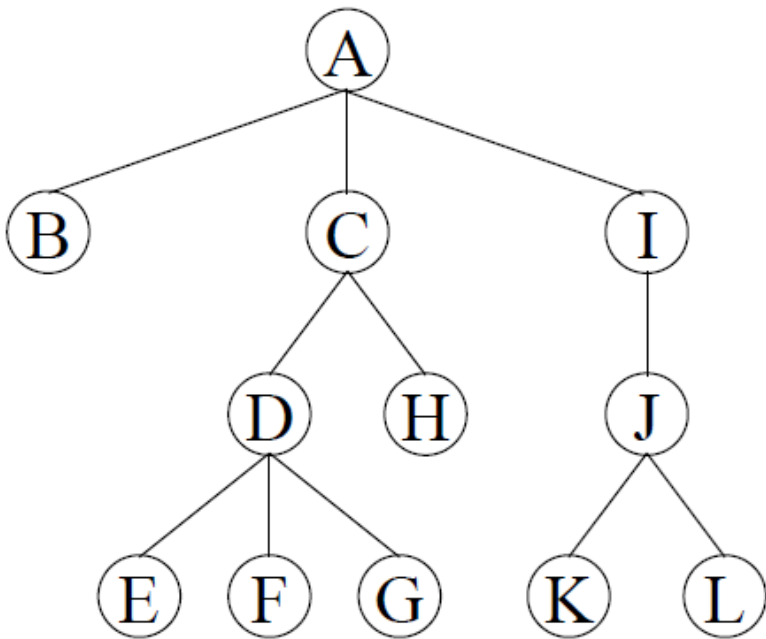
# 1. Conceptos de árboles - *Representación de árboles generales*

**Ejemplo:** listas ordenadas de hijos

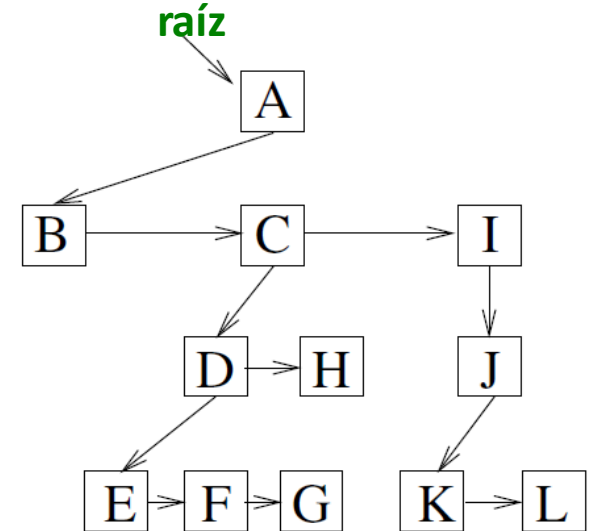


# 1. Conceptos de árboles - *Representación de árboles generales*

**Ejemplo:** hijo más a la izquierda-hermano derecho



	hijo Izq.	her. dato	der.
2	3	C	10
3	17	D	9
4	8	A	.
....			
8	.	B	2
9	.	H	.
10	13	I	.
....			
12	.	G	.
13	14	J	.
14	.	K	16
15	.	F	12
16	.	L	7
17	.	E	15
....			



# 1. Conceptos de árboles - Árboles binarios

- Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos hijos (hijo izquierdo e hijo derecho)

- **Propiedades:**

- El número máximo de nodos del nivel  $i$  es  $2^i$

- En un árbol de altura  $H$ , el número máximo de nodos es:

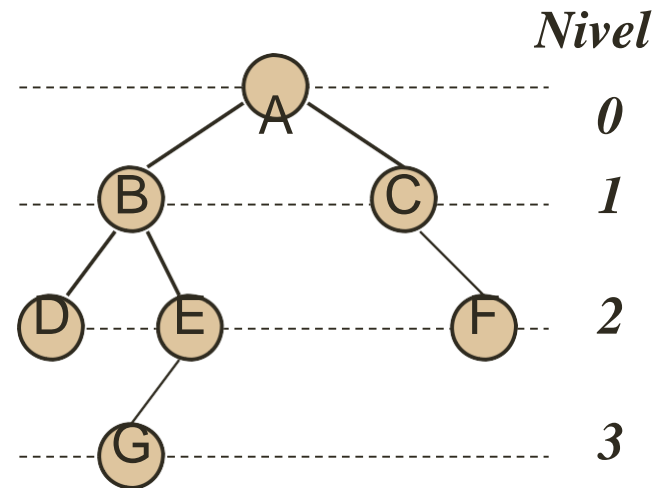
$$\sum_{i=0..H} 2^i = 2^{H+1} - 1$$

- El número máximo de hojas es:

$$(2^{H+1} - 1) - (\sum_{i=0..H-1} 2^i) = 2^H$$

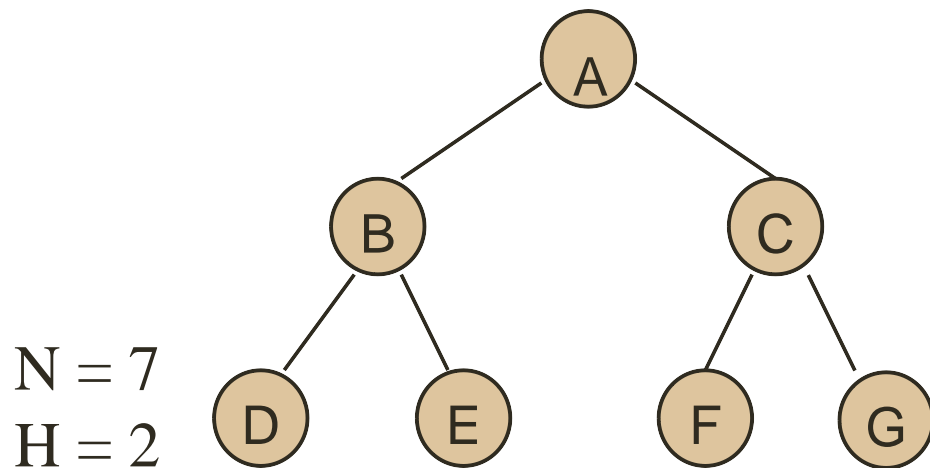
- El número máximo de nodos internos es:

$$(2^{H+1}-1) - (2^H) = 2^H - 1$$



# 1. Conceptos de árboles - Árboles binarios

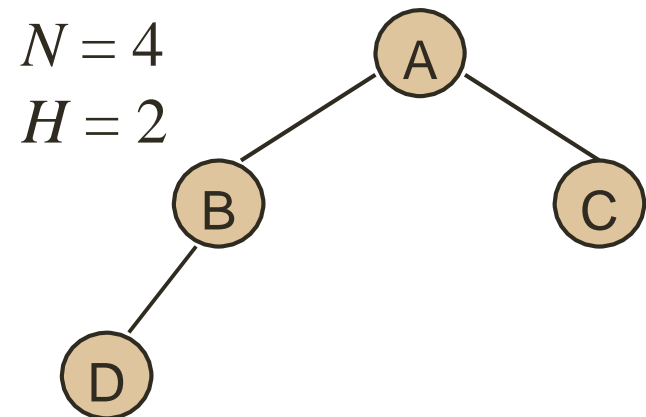
- Un **árbol binario** es **lleno** si tiene todos sus niveles completos
- **Propiedades:** sea  $H$  su altura y  $N$  su tamaño (número de nodos)
  - $H = \lfloor \log_2 N \rfloor$
  - $N = 2^{H+1} - 1$



# 1. Conceptos de árboles - Árboles binarios

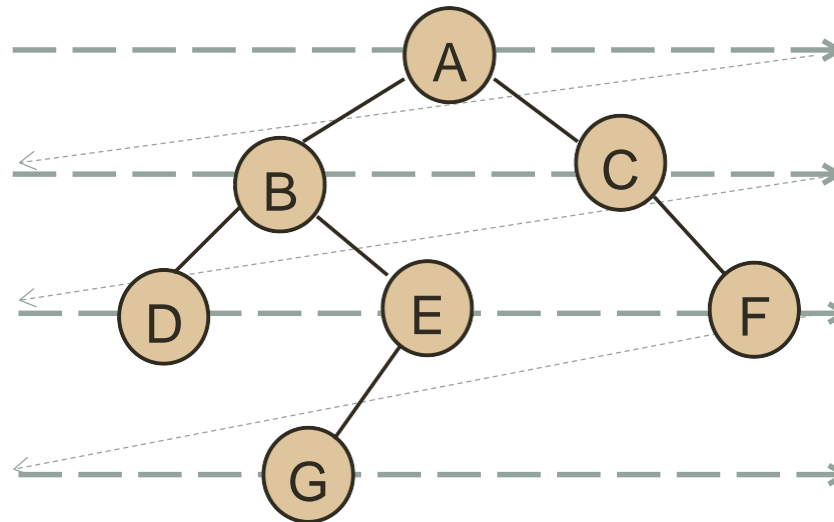
- Un **árbol binario** es **completo** tiene todos sus niveles completos, a excepción quizás del último en el cuál todas las hojas están tan a la izquierda como sea posible
- **Propiedades:** sea  $H$  su altura y  $N$  su tamaño (número de nodos)
  - $H \leq \lceil \log_2 N \rceil \rightarrow$  es un árbol **equilibrado**
  - $2^H \leq N \leq 2^{H+1} - 1$

**Nota:** un árbol es *equilibrado* si los subárboles izquierdo y derecho de cualquier nodo tienen alturas que difieren como mucho en 1



# 1. Conceptos de árboles - Árboles binarios

- En un recorrido en **anchura** (*por niveles*) de un árbol binario los nodos se visitan nivel a nivel y, dentro de cada nivel, de izquierda a derecha



Por niveles: ABCDEFG

# 1. Conceptos de árboles - Árboles binarios

○ **En profundidad:** los nodos se visitan bajando por las ramas del árbol

- ***Pre-Orden:***

1º) raíz, 2º) sub-árbol izquierdo, 3º) sub-árbol derecho

- ***In-Orden:***

1º) sub-árbol izquierdo, 2º) raíz, 3º) sub-árbol derecho

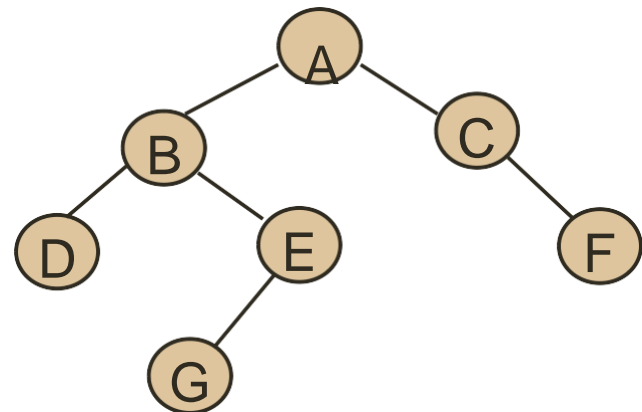
- ***Post-Orden:***

1º) sub-árbol izquierdo, 2º) sub-árbol derecho, 3º) raíz

Pre-Orden: ABDEGCF

In-Orden: DBGEACF

Post-Orden: DGEBFCA

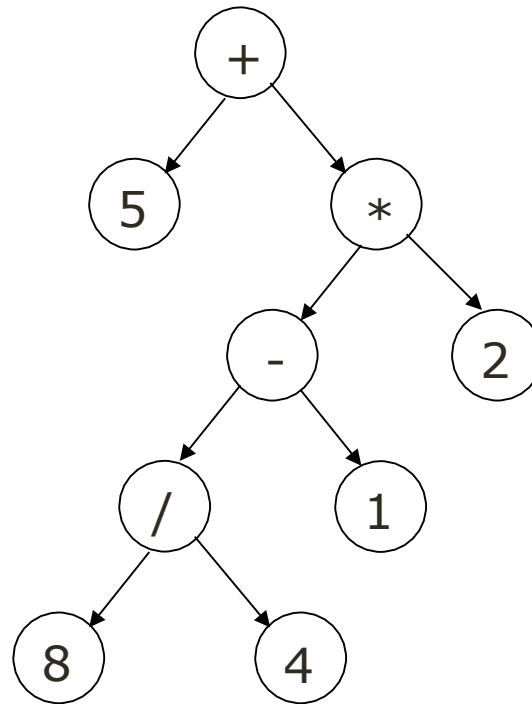




# 1. Conceptos de árboles - Árboles binarios

## *Ejercicio*

- Muestra el resultado de recorrer en pre-orden, in-orden, post-orden y por niveles el siguiente árbol



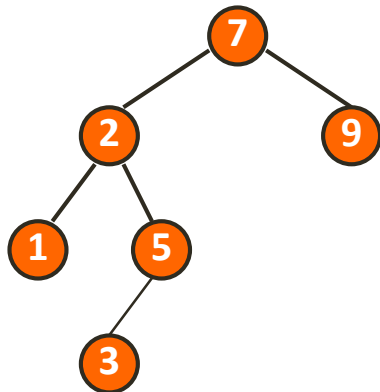
## 2. ABB (Equilibrado) - *Definición*

Un Árbol Binario de Búsqueda (ABB) Equilibrado es un Árbol Binario (AB) tal que ...

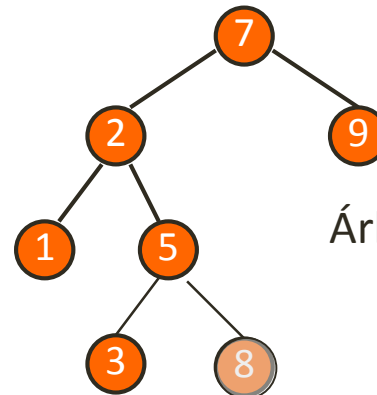
1. **PROPIEDAD ESTRUCTURAL:** AB **Equilibrado**
2. **PROPIEDAD DE ORDENACIÓN:** Propiedad de **Búsqueda Ordenada**

Un AB es **de Búsqueda** o cumple la **propiedad de Búsqueda Ordenada** si:

- Todos los datos de su subÁrbol Izquierdo son menores -o iguales- que el que ocupa su Nodo Raíz
- Todos los Datos de su subÁrbol Derecho son mayores que el que ocupa su Nodo Raíz
- Sus subÁrboles Izquierdo y Derecho también son ABB



Árbol Binario *de Búsqueda*



Árbol Binario, *¡a secas!*

### 3. ABB (Equilibrado) - *Propiedades y operaciones*

- **Localización de un Dato  $d$  en  $O(H)$ , i.e.  $O(\log N)$  en un ABB Equilibrado**

pues sólo se debe explorar en Profundidad **uno** de sus Caminos desde la Raíz

→ Operaciones en las que es posible: `recuperar(e)`, `insertar(e)`, `eliminar(e)`, `recuperarMin()`, `eliminarMin()`, `recuperarMax()`, `eliminarMax()`, `sucesor(e)`, `predecesor(e)`, etc.

- **Ordenación Ascendente de sus datos en  $\Theta(N)$** , pues basta recorrerlo In-Orden

- **Cálculo del tamaño de un ABB y el de cada uno de sus Nodos en  $\Theta(1)$** , pues al existir un único Camino de inserción y borrado se puede calcular el tamaño de sus Nodos conforme se va construyendo el ABB

- al insertar el Elemento  $e$  se crea el Nodo Hoja que lo contiene con tamaño 1
- al insertar/eliminar un Descendiente propio de un Nodo dado de un ABB su tamaño se incrementa/decrementa en 1

→ **Selección del  $k$ -ésimo menor elemento de una Colección en  $O(H)$**

→ **Operaciones de rango en  $O(H)$  i.e.  $O(\log N)$  en un ABB Equilibrado**

### 3. ABB (Equilibrado) - *Eficiencia y Grado de Equilibrio*

Un Árbol Binario de Búsqueda (ABB) Equilibrado es un Árbol Binario (AB) tal que ...

1. PROPIEDAD ESTRUCTURAL:

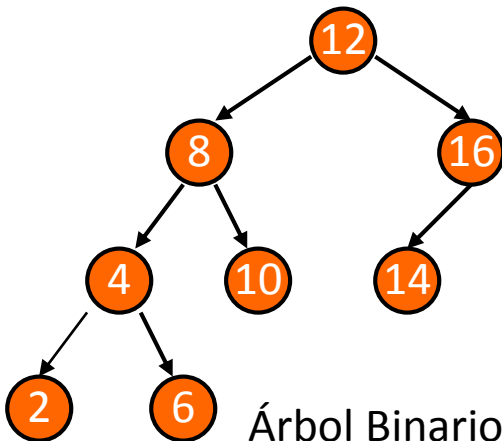
AB **Equilibrado**

2. PROPIEDAD DE ORDENACIÓN:

Elementos **Comparable**

Propiedad de **Búsqueda Ordenada**, lo que permite encontrar **CUALQUIER** dato del AB explorando tan solo UNO de sus caminos

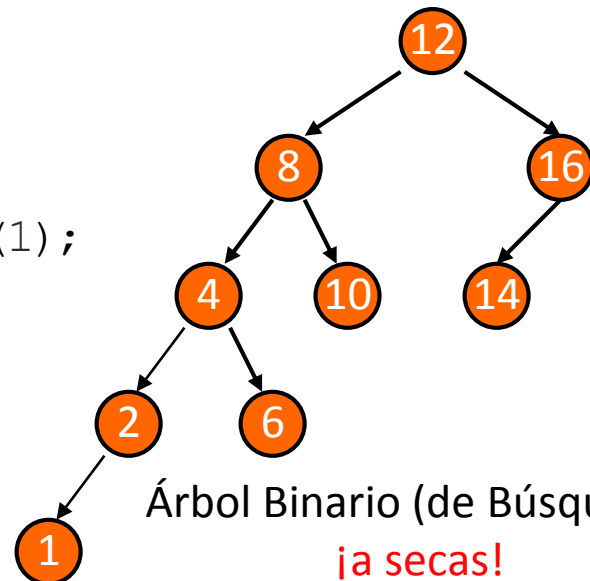
Un AB es **Equilibrado** si la diferencia de alturas entre sus Hijos Izquierdo y Derecho es **como máximo 1**



Árbol Binario (de Búsqueda)

**Equilibrado**

`insertar(1);`



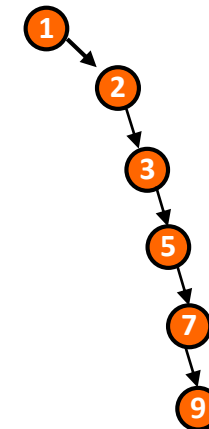
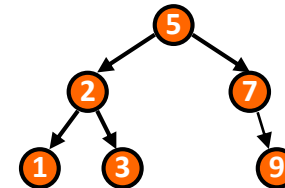
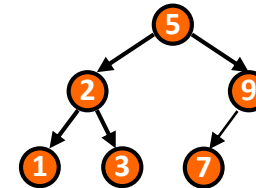
Árbol Binario (de Búsqueda)

**¡a secas!**

### 3. ABB (Equilibrado) - *Eficiencia y Grado de Equilibrio* (cont.)

El grado de equilibrio de un ABB depende fuertemente del orden de inserción de sus datos (como en *Quick Sort*). Por ejemplo, si los datos que contiene un ABB son 1, 2, 3, 5, 7 y 9

- Si su secuencia de inserción es 5, 2, 1, 3, 9 y 7 se obtiene un ABB ... ¡Completo!, luego Equilibrado
- Si su secuencia de inserción es 5, 2, 7, 1, 3 y 9 se obtiene un ABB ... ¡Equilibrado!, pero no Completo
- Si su secuencia de inserción es 1, 2, 3, 5, 7, 9 se obtiene un ABB ... ¡Completamente Degenerado!



### 3. ABB (Equilibrado) - *Eficiencia y Grado de Equilibrio* (cont.)

Como se explica en Weiss, apartado 18.3, se puede demostrar que:

- (a) La altura media de un ABB de talla  $N$  es  $\log_2 N$  si se ha generado como resultado de una secuencia aleatoria de  $N$  inserciones (sin operaciones eliminar); en concreto, en promedio la altura de un ABB así construido es sólo un 38% peor que en el Mejor de los Casos
- (b) Si después de la construcción aleatoria de un ABB se realizan  $N^2$  pares de combinaciones aleatorias de inserciones y borrados (en las que el orden de insertar y eliminar sea también aleatorio) el ABB resultante tiene una profundidad  $O(\sqrt{N})$

Desafortunadamente, ello no elimina la posibilidad de que se dé el Peor de los Casos, cuando se trabaja con secuencias de datos ordenadas o que contienen subsecuencias largas no aleatorias.

### 3. ABB (Equilibrado) - *Eficiencia y Grado de Equilibrio* (cont.)

Para imponer la condición de Equilibrio...

- (1) **ABB Bien Equilibrado**, o ABB al que se impone una Condición de Equilibrio tal que  $H \approx \lfloor \log N \rfloor$ , como por ejemplo los **Árboles “AVL”** (Adelson, Velskii y Landis) o los **“Rojinegros”**

→ Ello obliga a sofisticar la Representación del ABB y la implementación de sus operaciones de inserción y eliminación

- (2) **Construir un ABB Equilibrado a partir de un array  $v$  que contenga los datos a insertar ordenados Ascendentemente**, como sigue:

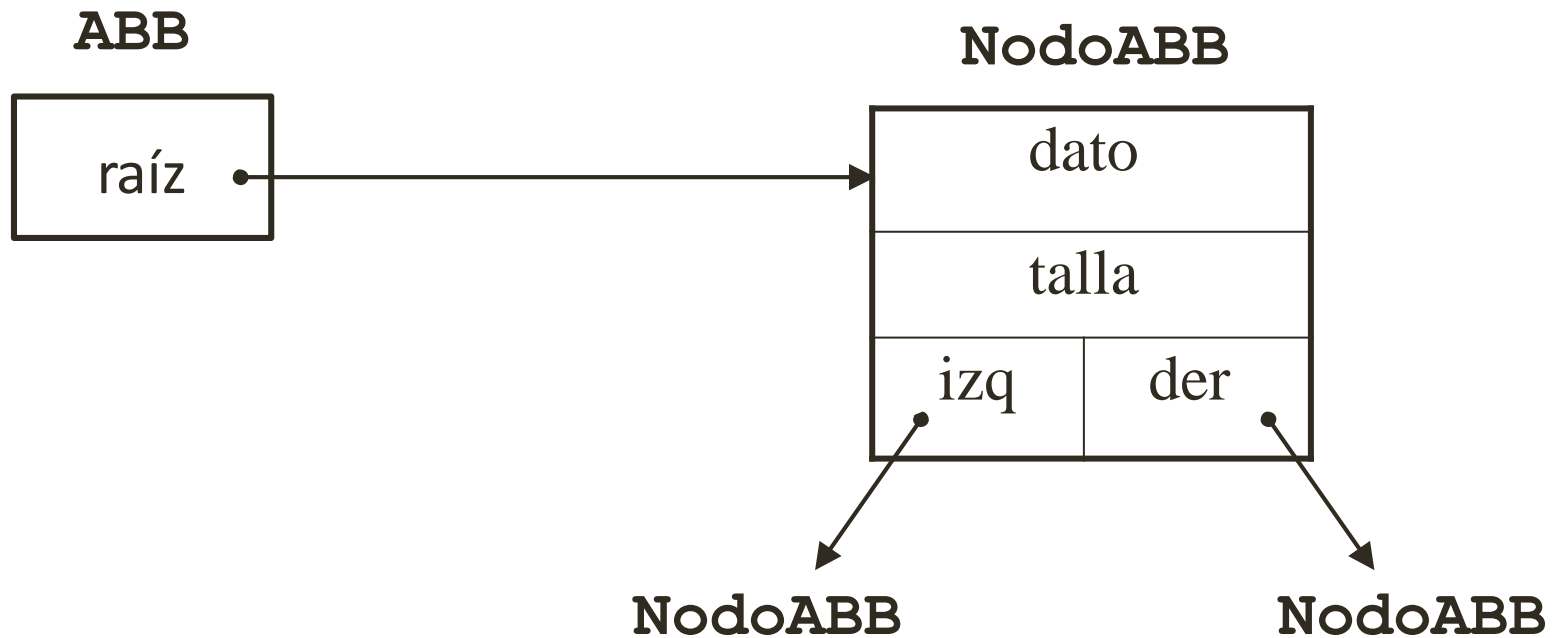
**2.1.** El dato en  $v[\text{mitad}]$ , la mediana de los datos de  $v$ , se sitúa como Raíz del ABB en construcción

**2.2.-** Se construye el Hijo Izquierdo del ABB aplicando recursivamente el punto 2.1 a los datos del (sub)array  $v[0, \text{mitad} - 1]$  hasta vaciarlo

**2.3.-** Se construye el Hijo Derecho del ABB aplicando recursivamente el punto 2.1 a los Datos del (sub)array  $v[\text{mitad} + 1, v.\text{length} - 1]$  hasta vaciarlo

### 3. Implementación de un ABB - *Las clases*

*NodoABB y ABB*





### 3. Implementación de un ABB - *Las clases* *NodoABB y ABB (cont.)*

```
package librerias.estructurasDeDatos.jerarquicos;

public class ABB<E extends Comparable<E>> {
    protected NodoABB<E> raiz;

    public ABB() { this.raiz = null; }
    public boolean esVacio() { return this.raiz == null; }
    public talla() { return talla(this.raiz); }
    protected talla(NodoABB<E> actual) {
        if (actual == null) { return 0; }
        else { return actual.talla; }
    }
    ...
}

class NodoABB<E> {
    protected E dato;
    protected NodoABB<E> izq, der;
    int talla;

    NodoABB(E e) {
        this.dato = e;
        this.izq = null; this.der = null;
        talla = 1;
    }
}
```

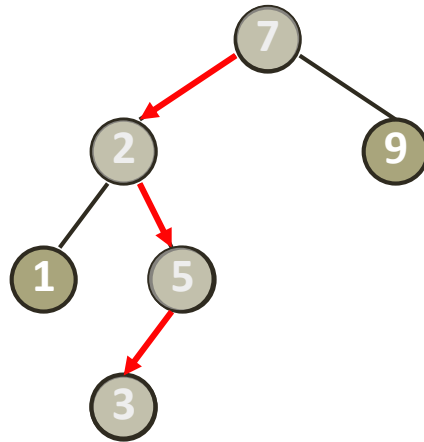
### 3. ABB - *Recuperar*

```
protected NodoABB<E> recuperar(E e, NodoABB<E> actual) {  
    NodoABB<E> res = actual;  
    if (actual != null) {  
        int resC = actual.dato.compareTo(e);  
        if (resC > 0) { res = recuperar(e, actual.izq); } //dato>e  
        else if (resC < 0) { res = recuperar(e, actual.der); } //dato<e  
        // else NO hacer nada porque res se ha inicializado a actual (dato=e)  
    }  
    return res;  
}
```

```
public E recuperar(E e) {  
    NodoABB<E> res = recuperar(e, this.raiz);  
    if (res == null) { return null; }  
    else { return res.dato; }  
}
```

### 3. ABB - Recuperar (cont.)

recuperar( $e$ ) , con  $e = 3$



Búsqueda de  $e$  en un **único Camino del ABB** (tamaño  $N$  y altura  $H$ )

- Si  $e$  está en el Nodo Raíz del ABB (caso mejor):  $T_{\text{recuperar}}(N) \in \Omega(1)$
- Si  $e$  no está en el ABB y se busca en su Camino Más Largo (caso peor):

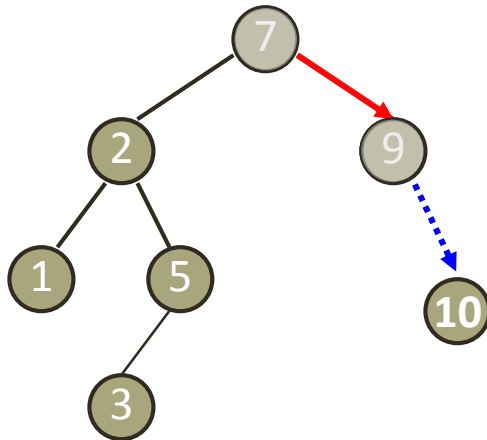
(a) ABB Equilibrado:  $T_{\text{recuperar}}(N) \in O(H = \log N)$

(b) ABB Completamente Degenerado:  $T_{\text{recuperar}}(N) \in O(H = N)$

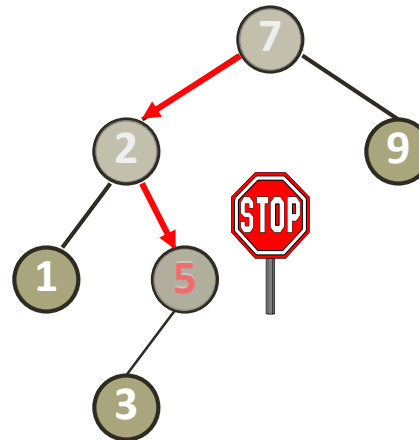
$$T_{\text{recuperar}}^{\mu}(N) \in O(\log N = H)$$

### 3. ABB – Insertar (sin duplicados)

insertar (10), que NO está



insertar (5), que SÍ está



Operación en **dos** pasos:

1. Búsqueda **del lugar de inserción** de  $e$  en un único Camino del ABB
2. **Resolución de la Búsqueda:** si  $e$  está **actualizar** y si no está **insertar** una nueva Hoja que lo contenga

$$T_{\text{insertar}}^{\mu}(N) \in O(\log N = H)$$

### 3. ABB - Insertar (cont.)

```
protected NodoABB<E> insertar(E e, NodoABB<E> actual) {  
    NodoABB<E> res = actual;  
    if (actual != null) {  
        int resC = actual.dato.compareTo(e);  
        if (resC > 0) { res.izq = insertar(e, actual.izq); }  
        else if (resC < 0) { res.der = insertar(e, actual.der); }  
        else { res.dato = e; }  
        res.talla = 1 + talla(res.izq) + talla(res.der);  
    }  
    else { res = new NodoABB<E>(e); }  
    return res;  
}
```

Se actualiza la talla de todos los nodos del camino

Se inicializa la talla a 1

```
public void insertar(E e) {  
    // devuelve el Nodo Raíz que se le pasa como parámetro,  
    // con un Nodo más si e NO estaba en él  
    this.raiz = insertar(e, this.raiz)  
}
```

### 3. ABB - Insertar (cont.)

¿En qué se diferencian insertar y recuperar? (en verde)

```
protected NodoABB<E> recuperar(E e, NodoABB<E> actual) {  
    NodoABB<E> res = actual;  
    if (actual != null) {  
        int resC = actual.dato.compareTo(e);  
        if (resC > 0) { res = recuperar(e, actual.izq); }  
        else if (resC < 0) { res = recuperar(e, actual.der); }  
        // else NO hacer nada porque res se ha inicializado a actual  
    }  
    return res;  
}
```

```
protected NodoABB<E> insertar(E e, NodoABB<E> actual) {  
    NodoABB<E> res = actual;  
    if (actual != null) {  
        int resC = actual.dato.compareTo(e);  
        if (resC > 0) { res.izq = insertar(e, actual.izq); }  
        else if (resC < 0) { res.der = insertar(e, actual.der); }  
        else { res.dato = e; }  
        res.talla = 1 + talla(res.izq) + talla(res.der);  
    }  
    else { res = new NodoABB<E>(e); }  
    return res;  
}
```

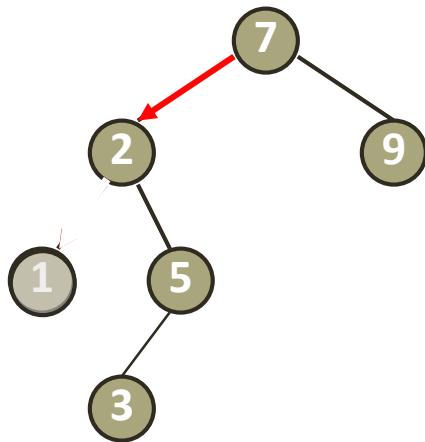
Se actualiza la talla

Se inicializa la talla a 1

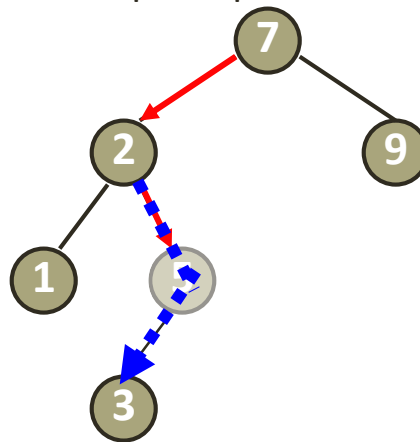
### 3. ABB – Eliminar

¿Dónde y cómo se elimina un dato en un ABB?

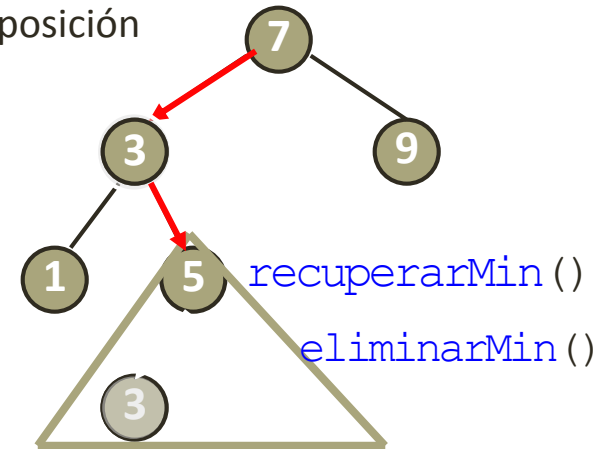
`eliminar(1),`  
**Hoja:** se elimina



`eliminar(5),`  
**Nodo con 1 Hijo:** su hijo ocupa su posición



`eliminar(2),`  
**Nodo con 2 Hijos:** el mínimo de su subárbol derecho ocupa su posición




Operación en **dos** pasos:

1. Búsqueda de  $e$  en un único Camino del ABB
2. **Resolución de la Búsqueda:** si  $e$  está, borrar el Nodo que lo contiene; sino NO hacer nada

$$T_{\text{eliminar}}^{\mu}(N) \in O(\log N)$$

### 3. ABB – *Eliminar (cont.)*

```
protected NodoABB<E> eliminar(E e, NodoABB<E> actual) {
    NodoABB<E> res = actual;
    if (actual != null) {
        int resC = actual.dato.compareTo(e);
        if (resC > 0) { res.izq = eliminar(e, actual.izq); }
        else if (resC < 0) { res.der = eliminar(e, actual.der); }
        else { // eliminar actual
            if (actual.izq == null) { return actual.der; }
            else if (actual.der == null) { return actual.izq; }
            else {
                res.dato = recuperarMin(actual.der).dato;
                res.der = eliminarMin(actual.der);
            }
        }
        res.talla = 1 + talla(res.izq) + talla(res.der);
    }
    return res;
```



Se actualiza la talla de todos los nodos del camino

```
public void eliminar(E e) {
    // devuelve el Nodo Raíz que se le pasa como parámetro,
    // con un Nodo menos si e SÍ estaba en él
    this.raiz = eliminar(e, this.raiz)
}
```

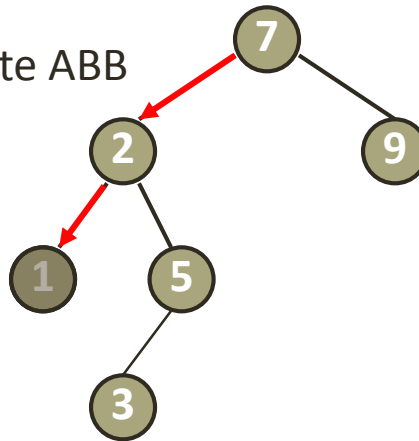


### 3. Implementación de un ABB – *RecuperarMin*

El mínimo en un ABB no tiene hijo izquierdo y no pertenece a ningún subárbol derecho de ningún nodo. (El máximo es el caso simétrico.)

`recuperarMin`(actual)

siendo actual el Nodo Raíz de este ABB



**Recorrido del Camino más a la izquierda del ABB** hasta alcanzar su último Nodo, que puede no ser una Hoja, y devolverlo:  $T^{\mu}_{\text{recuperarMin}}(N) \in O(\log N)$

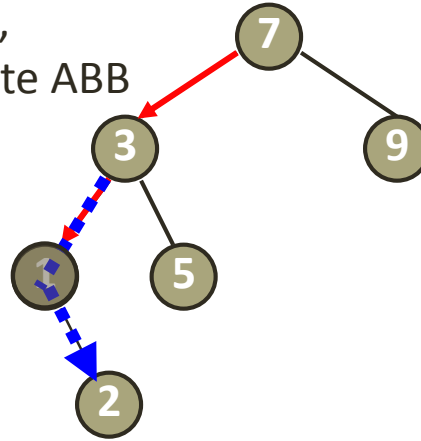
```
//SII actual != null: devuelve el Nodo de actual que contiene a su mínimo
protected NodoABB<E> recuperarMin(NodoABB<E> actual) {
    if (actual.izq == null) { return actual; }
    return recuperarMin(actual.izq);
}
```

```
// SII !esVacio()
public E recuperarMin() {
    return recuperarMin(raiz).dato;
}
```

### 3. Implementación de un ABB – *EliminarMin*

El método `eliminarMin` de ABB, diferencias con `recuperarMin` y coste

**Ejemplo:** `eliminarMin(actual)`,  
siendo `actual` el Nodo Raíz de este ABB



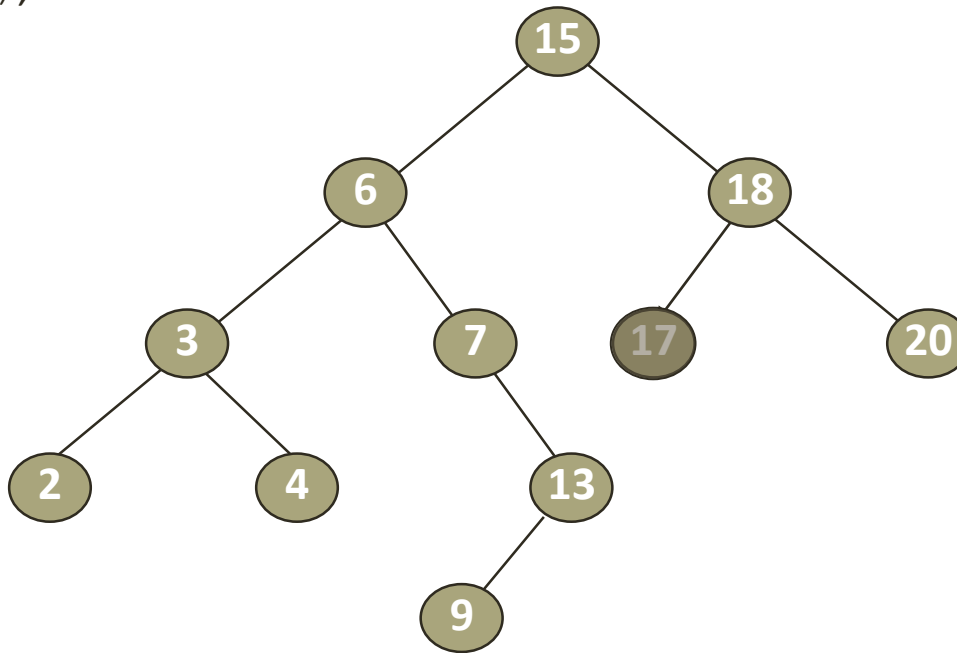
Recorrido del Camino más a la izquierda del ABB hasta alcanzar su último Nodo, que puede no ser una Hoja, y **borrarlo**:  $T^{\mu}_{\text{eliminarMin}}(N) \in O(\log N)$

```
//SII actual != null: devuelve el Nodo actual tras eliminar su mínimo
protected NodoABB<E> eliminarMin(NodoABB<E> actual) {
    if (actual.izq == null) { return actual.der; }
    actual.izq = eliminarMin(actual.izq);
    actual.talla--;
    return actual;
}
```

```
// SII !esVacio()
public E eliminarMin() {
    E res = recuperarMin();
    this.raiz = eliminarMin(this.raiz);
    return res;
}
```

### 3. Implementación de un ABB – *Sucesor*

*sucesor*(e), con e = 15

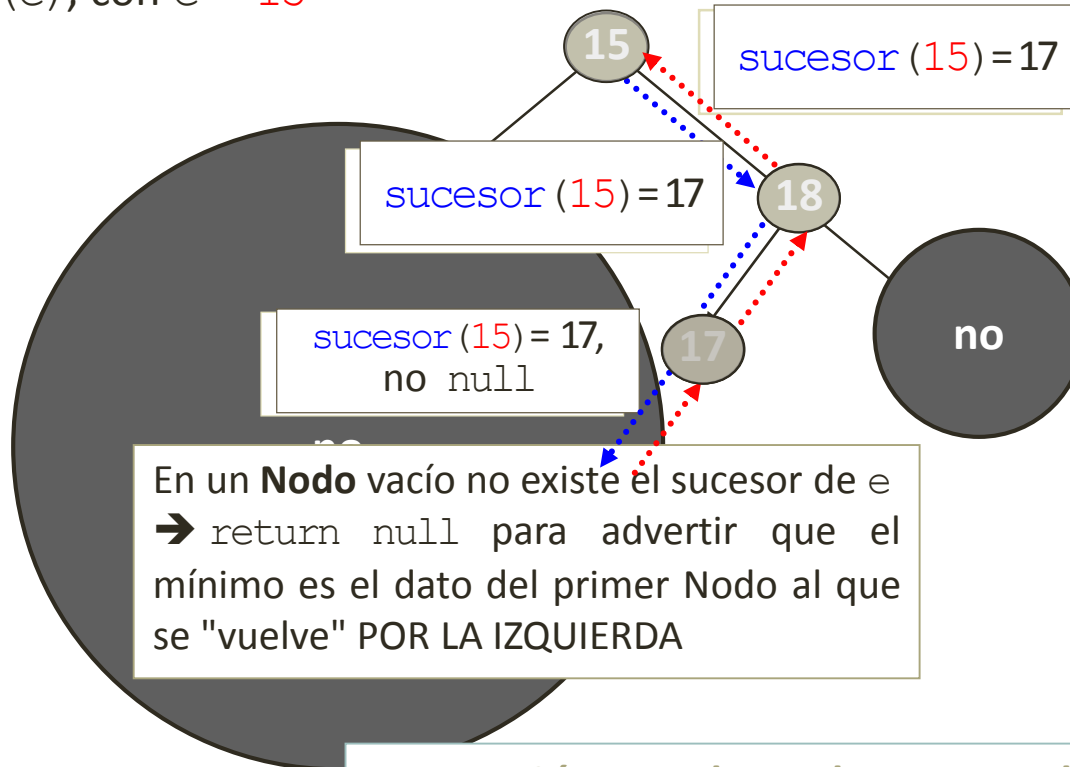


¿Cómo explorar el ABB para obtener el **17**?

Si un Nodo tiene Hijo Derecho, por definición, el sucesor del dato que contiene es el mínimo de su Hijo Derecho

### 3. ABB – Sucesor (cont.)

sucesor(e), con  $e = 15$



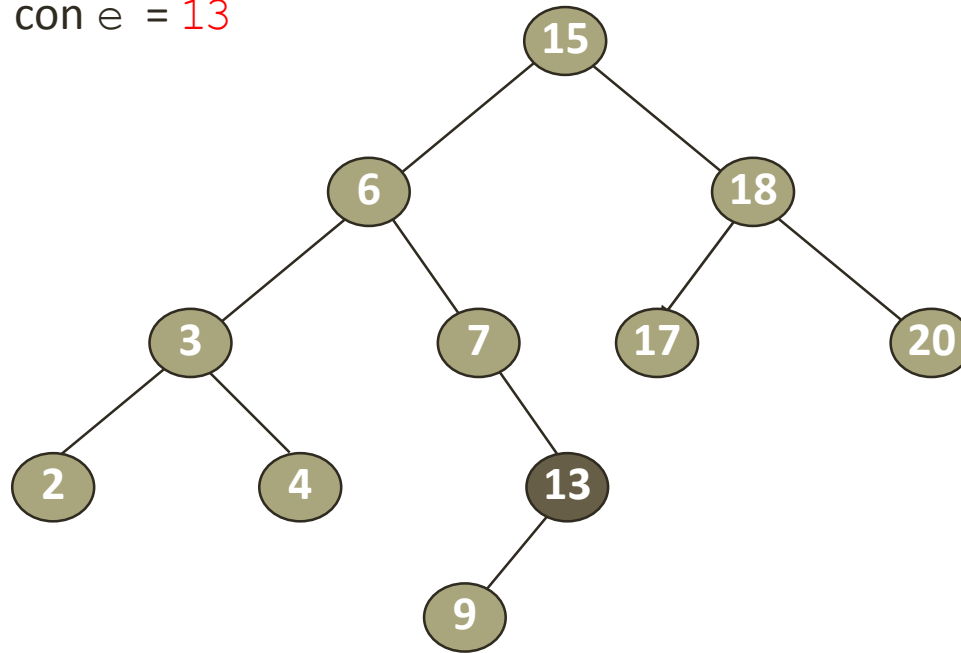
¿Cómo explorar el ABB para obtener el 17?

**Como siempre:** aprovechamos la propiedad de Ordenación del ABB para seguir el único Camino donde puede estar el dato a recuperar, de altura promedio  $\log N$

**Sí, pero este es un caso fácil:** si un Nodo tiene Hijo Derecho, por definición, el sucesor del dato que contiene es el mínimo de su Hijo Derecho. **¿Y si no tiene hijo derecho?**

### 3. ABB – Sucesor (cont.)

sucesor (e), con e = 13

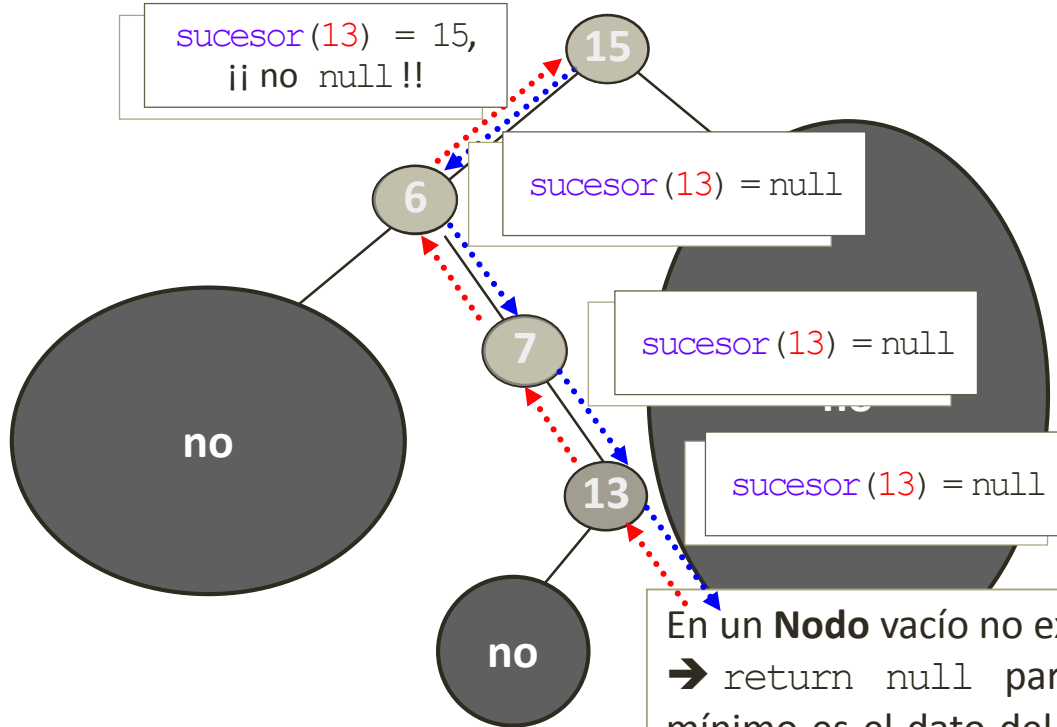


¿Cómo explorar el ABB para obtener el 15?

Si un Nodo no tiene Hijo Derecho, el sucesor del dato que contiene es su Ascendiente por la Derecha más próximo.

### 3. ABB – Sucesor (cont.)

`sucesor(e)`, con  $e = 13$



¿Cómo explorar el ABB para obtener el 15?

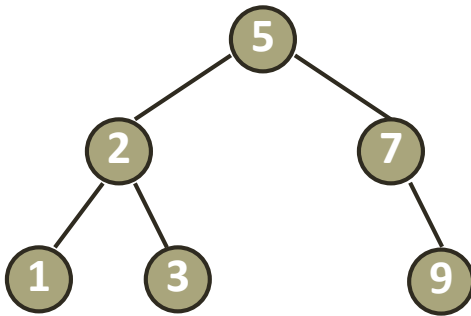
En un **Nodo** vacío no existe el sucesor de  $e$   
→ return null para advertir que el mínimo es el dato del primer Nodo al que se "vuelve" POR LA IZQUIERDA

**De nuevo, como siempre:** aprovechando la propiedad de Ordenación del ABB

Si un Nodo no tiene Hijo Derecho, el sucesor del dato que contiene es su Ascendiente por la Derecha más próximo.

### 3. ABB – Sucesor (cont.)

El sucesor de un nodo equivale al siguiente nodo que se obtendría en un recorrido InOrden del árbol



`sucesor(5)` = 7

`sucesor(1)` = 2

`sucesor(3)` = 5

`sucesor(9)` = null

`Sucesor(7)` = 9

El sucesor es el primero *al que se vuelve* por la izquierda (desde el caso base)

### 3. ABB – *Sucesor (cont.)*

¿En qué se diferencian sucesor y recuperar? (en verde)

```
protected NodoABB<E> sucesor(E e, NodoABB<E> actual) {
    NodoABB<E> res = null;
    if (actual != null) {
        int resC = actual.dato.compareTo(e);
        if (resC > 0) {
            res = sucesor(e, actual.izq); //dato>e: va a la izq
            //vuelve de la izq, donde siempre está el sucesor
            if (res == null) { res = actual; } //actualiza sucesor
        }
        else { res = sucesor(e, actual.der); } //va a la dra
            //vuelve de la dra, luego el sucesor no varía
        }
    return res;
}
```

```
public E sucesor(E e) {
    NodoABB<E> res = sucesor(e, this.raiz);
    if (res == null) { return null; }
    else { return res.dato; }
}
```

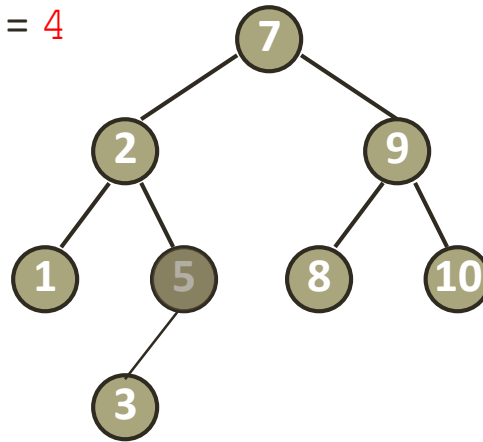
Reducción Logarítmica:  $T_{\text{sucesor}}^{\mu}(N) \in O(\log N)$



### 3. ABB – $K$ -ésimo mínimo

El método **seleccionar( $k$ )**: cómo resolver eficientemente el problema de la Selección en una colección de talla  $N$  (calcular su  $k$ -ésimo mínimo,  $1 \leq k \leq N$ , **revisitado**)

**Ejemplo:** **seleccionar**( $k$ ), con  $k = 4$



¿Cómo explorar la Raíz del ABB para obtener el **5**?

Acceder al  **$k$ -ésimo elemento de su Recorrido In-Orden**, con el mismo coste ( $O(N)$ ) que cuando los datos están sobre un array (recuerda el método DyV `seleccionRapida`)

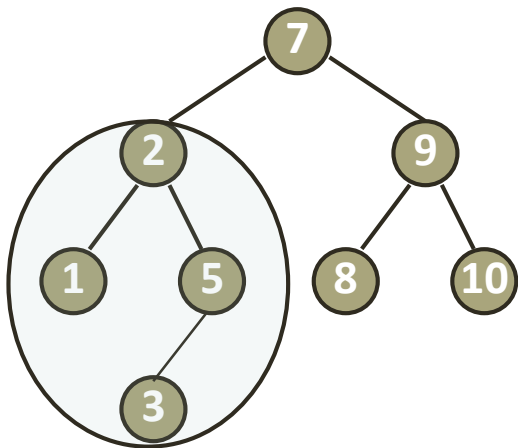
.... ¿No se puede usar otra propiedad del ABB para mejorar este coste? ...

**PISTA:** recuerda que usas un **ABB con Rango** y que el (primer) **mínimo** del ABB ( $k=1$ ) está en su **Hijo Izquierdo**

### 3. ABB – *K*-ésimo mínimo (cont.)

Ejemplos de `seleccionar(k, actual)`, con `talla(actual.izq) = 4`

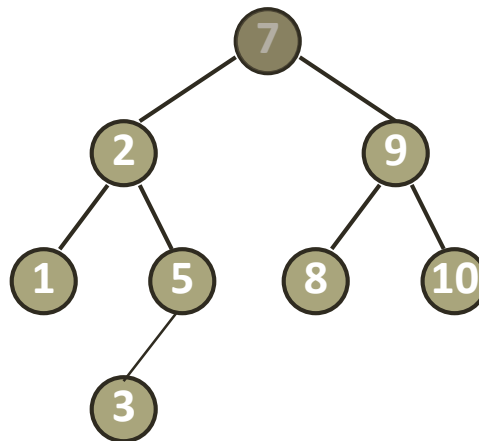
con  $k = 4$



si ( $k \leq \text{talla}(\text{actual.izq})$ )

el  $k$ -ésimo está en `actual.izq`

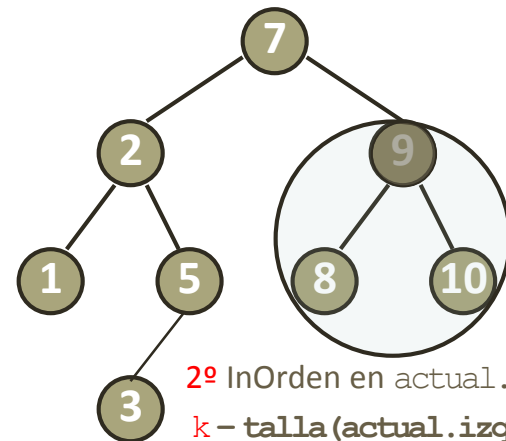
con  $k = 5$



si ( $k = \text{talla}(\text{actual.izq}) + 1$ )

el  $k$ -ésimo es `actual.dato`

con  $k = 7$



2º InOrden en `actual.der`  
 $k - \text{talla}(\text{actual.izq}) - 1$

si ( $k > \text{talla}(\text{actual.izq}) + 1$ )

el  $k$ -ésimo está en `actual.der`

¿Cómo explorar el `Nodo actual` para obtener su  $k$ -ésimo mínimo?

**PISTA para la solución óptima:** recuerda que usas un **ABB con Rango** y que el (primer) **mínimo** del ABB ( $k=1$ ) está en su **Hijo Izquierdo**

### 3. ABB – *K-ésimo mínimo (cont.)*

#### El método **seleccionar(k)** de ABB y su coste

```
protected NodoABB<E> seleccionar(int k, NodoABB<E> actual) {  
    int tallaI = talla(actual.izq);  
    if (k == tallaI + 1) {  
        return actual;  
    }  
    else if (k <= tallaI) {  
        return seleccionar(k, actual.izq);  
    }  
    else {  
        return seleccionar(k - tallaI - 1, actual.der);  
    }  
}  
  
// SII !esVacio() AND 1 <= k <= talla()  
public E seleccionar(int k) {  
    return seleccionar(k, this.raiz).dato;  
}
```

**Reducción Logarítmica + Rango :**  $T^{\mu}_{\text{seleccionar}}(N) \in O(\log N)$

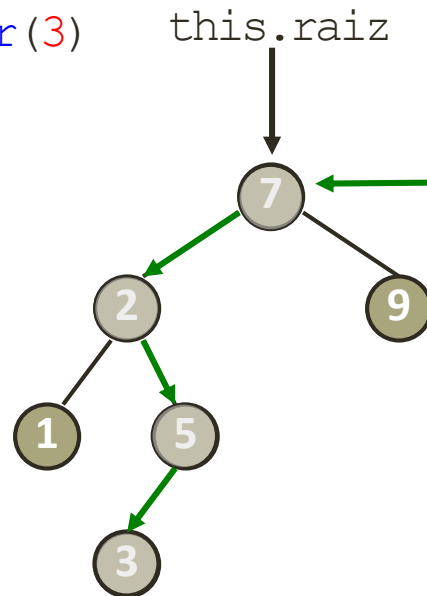
### 3. ABB – *Iterativa*

El método `recuperar` de la clase ABB (Equilibrado), y cualquier otro de **Búsqueda Dinámica**, implementa una Exploración Recursiva.

➔ Para **ahorrar la sobrecarga espacial** que supone la Pila de la Recursión, se debe realizar una sencilla **transformación recursivo-iterativa**

¿Cómo?

Ejemplo: `recuperar(3)`

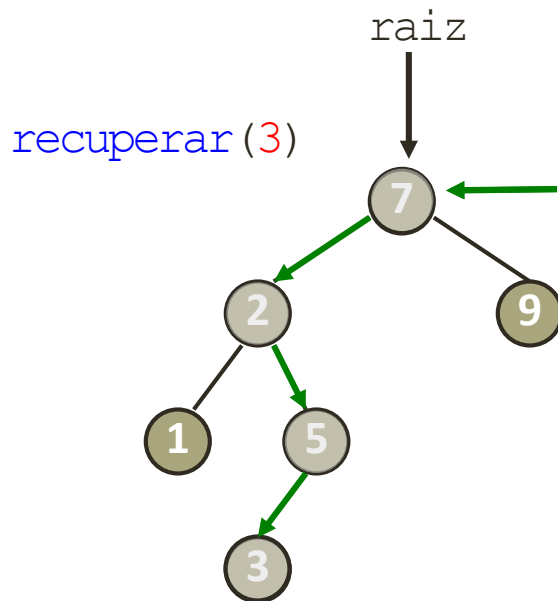


`aux`, enlace auxiliar

- se inicializa a `this.raiz`
- en función del resultado del `compareTo` sobre cada dato de cada Nodo, **avanza en cada iteración sobre el Hijo Izquierdo o el Derecho**

### 3. ABB – *Iterativa: Métodos consultores*

```
public E recuperar(E e) {  
    NodoABB<E> aux = this.raiz;  
    while (aux != null) {  
        int resC = aux.dato.compareTo(e);  
        if (resC == 0)      { return aux.dato; } //dato=e  
        else if (resC > 0) { aux = aux.izq;   } //dato>e  
        else                { aux = aux.der;   } //dato<e  
    }  
    return null;  
}
```

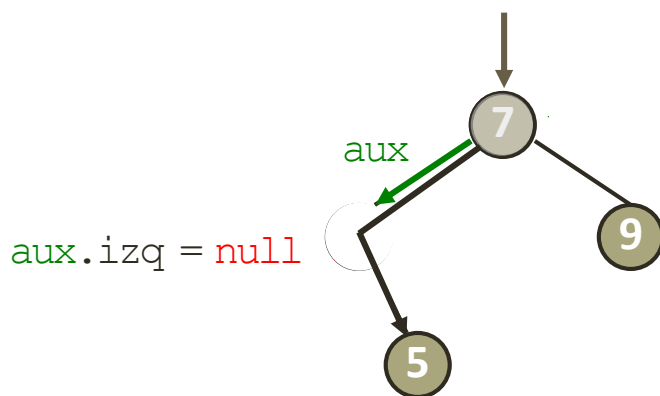


### 3. ABB – *Iterativa: Métodos modificadores*

La transformación a iterativo de cualquier método **modificador** de un ABB presenta 2 puntos más a tener en cuenta, comparada con la de un método consultor:

- ① Manejar aquellas situaciones en las que `aux`, o alguno de sus Hijos, sea `null`, como “desenlazar” el Nodo a borrar del resto del árbol o “enlazar” el Nodo a insertar al árbol existente
- ② Actualizar el tamaño de los nodos del camino de inserción o borrado, si es que realmente se ha modificado el árbol

**Ejemplo:** `eliminarMin()`    `this.raiz`



¿Cómo “borrar” el Nodo que contiene al mínimo?

Sólo con `aux`, IMPOSIBLE! Hay que usar un puntero más, que siempre apunte al padre de `aux`:  
`padreAux.izq = aux.der;`

### 3. ABB – *Iterativa: Métodos modificadores (cont.)*

Se puede modificar el método `protected eliminarMin` para que, además de devolver el `Nodo` que se le pasa como parámetro SIN su mínimo, **recupere el dato que contiene mínimo**.

```
// SII actual != null: devuelve el Nodo actual tras eliminar su mínimo,  
// copiando el dato que este contiene en nodoMin (paso por referencia)  
protected NodoABB<E> eliminarMin(NodoABB<E> actual, NodoABB<E> nodoMin) {  
    NodoABB<E> aux = actual, padreAux = null;  
    while (aux.izq != null) {  
        aux.talla--;  
        padreAux = aux; aux = aux.izq;  
    }  
    nodoMin.dato = aux.dato;  
    if (padreAux == null) { actual = actual.der; }  
    else { padreAux.izq = aux.der; }  
    return actual;  
}
```

```
// SII !esVacio()  
public E eliminarMin() {  
    NodoABB<E> nodoRes = new NodoABB<E>(null);  
    this.raiz = eliminarMin(this.raiz, nodoRes);  
    E res = nodoRes.dato;  
    return res;  
}
```

## 4. Map Ordenado - *Modelo*

Un Map Ordenado es un conjunto dinámico de **x** entradas que soportan **eficientemente** no solo las operaciones de un Map sino también las típicas de un **conjunto ordenado por Clave** (siguiente en el orden o sucesor, anterior en el orden o predecesor, máximo, mínimo...). Precisamente porque maximiza las ventajas de Map y Lista Ordenada y, a la vez, minimiza sus inconvenientes, el coste máximo estimado de una de sus operaciones básicas es  $\log x$ , el de la Búsqueda Binaria en un conjunto estático y ordenado.

En el estándar Java este modelo es SortedMap, de la Jerarquía Collection

```
package librerias.estructurasDeDatos.modelos;

public interface MapOrdenado<C extends Comparable<C>, V> extends Map<C, V> {
    EntradaMap<C, V> recuperarEntradaMin(); // Entrada de clave mínima
    C recuperarMin();
    EntradaMap<C, V> recuperarEntradaMax(); // Entrada de clave máxima
    C recuperarMax(); // clave máxima

    EntradaMap<C, V> sucesorEntrada(C c); // siguiente Entrada a c en orden
    C sucesor(C c);

    EntradaMap<C, V> predecesorEntrada(C c); // Entrada anterior a c en orden
    C predecesor(C c);
    ...
}
```

**INDISPENSABLE** garantizar la consistencia de los métodos equals y compareTo para cualquier par de claves c1 y c2:

$c1.compareTo(c2) == 0$  **SII**  $c1.equals(c2)$



## 4. Map Ordenado - ABB (Equilibrado)

*¿Qué tipo de Árboles Binarios usar para Representar un Map Ordenado?*

Coste Promedio Representación					
	recuperar	insertar eliminar	sucesor predecesor	eliminarMin eliminarMax	recuperarMin recuperarMax
Lineal	$\Theta(x)$	$\Theta(1)$ $\Theta(x)$	$\Theta(x)$	$\Theta(x)$	$\Theta(x)$
Lineal Contigua Ordenada	$\Theta(\log x)$	$\Theta(x)$	$\Theta(\log x)$	$\Theta(1)$	$\Theta(1)$
Tabla Hash	$\Theta(1)$	$\Theta(1)$	$\Theta(x)$	$\Theta(x)$	$\Theta(x)$
Árbol Binario de Búsqueda Equilibrado	$\Theta(\log x)$	$\Theta(\log x)$	$\Theta(\log x)$	$\Theta(\log x)$	$\Theta(\log x)$

## 4. Map Ordenado - *Ejemplos de uso*

Existen dos grandes tipos de aplicaciones que usan un Map Ordenado:

### (a) Aplicaciones con restricciones temporales fuertes (*Time-sensitive*)

- Sistema de reservas de la pista de aterrizaje de un aeropuerto, que solo acepta la petición de aterrizaje de cualquier avión en un tiempo dado  $t$  SI no hay planificado ningún otro aterrizaje en los  $k$  minutos previos o posteriores a  $t$  y, obviamente, en el tiempo actual ( $t_{\text{actual}}$ ). **Por ejemplo**, para  $t_{\text{actual}}=37$ ,  $k=3$  y una lista de reservas  $R=[41.2', 49', 56.3']$ , la petición  $t=44'$  NO será aceptada; sin embargo, si se aceptará la petición para  $t=53'$ .
- El *Completely Fair Scheduler* (CFS) que usa el actual kernel de Linux. Se mantiene una lista de tareas ordenada Asc. por la cantidad de tiempo de ejecución (en nanoseg.) que ya ha utilizado la tarea. Cuanto menor sea dicho tiempo para una tarea, mayor será su necesidad de acceder al procesador y, por tanto, más cerca del principio de la dicha lista estará.
- Problemas de Geometría Computacional, como la generación de Diagramas de Voronoi o el problema de encontrar los dos puntos más cercanos de un conjunto en un espacio métrico (*Closest Pair Problem*).

### (b) Aplicaciones en las que las claves pertenecen a un conjunto completamente ordenado (por ejemplo, el de los `String`) y, por tanto, donde se pueden realizar búsquedas basadas tanto en dichas claves como en sus “vecinos” más cercanos (predecesores y/o sucesores) o en sus sufijos y/o prefijos. **Por ejemplo**: Routers de Internet, Correctores Ortográficos sencillos (verificadores), la función “Autocompletar” de un Editor Predictivo, etc.

## 4. Map Ordenado - *Ejemplos de uso (cont.)*

Los siguientes ejercicios enuncian problemas de interés, conocidos en su mayoría, cuya solución se puede (re)plantear en términos de las operaciones de un Map Ordenado.

- **Problema 1: Listar las Entradas de un Map en orden Asc. (Desc.)**

Diseña un método estático, genérico e iterativo `entradas` que devuelva una `ListaConPI` con las Entradas de un Map `m` ordenadas Asc. (Desc.)

- **Problema 2: Ordenación**

Diseña un método estático, genérico e iterativo `mapSort` que, con la ayuda de un `MapOrdenado`, ordene los elementos (`Comparable`) de un array `v`

- **Problema 3: 2-SUM (simplificación del problema de la suma de subconjuntos)**

Diseña un método estático, genérico e iterativo `hayDosQueSuman` que, dados un array `v` de enteros y un entero `k`, determine si existen en `v` dos números cuya suma sea `k`; usa un Map Ordenado como EDA auxiliar

## 4. Map Ordenado - *Ejemplos de uso (cont.)*

- **Problema 1: Listar las Entradas de un Map en orden Asc.**

Diseña un método estático, genérico e iterativo `entradas()` que devuelva una `ListaConPI` con las Entradas de un Map `m` ordenadas Asc.

**Pista** –¿quién es el sucesor del mínimo de un conjunto? ¿Y el sucesor del sucesor del mínimo? ¿Y el sucesor del sucesor del sucesor del mínimo?

```
public static <C extends Comparable<C>> ListaConPI<EntradaMap<C, V>> entradas (MapOrdenado<C, V> m) {  
    ListaConPI<EntradaMap<C, V>> l = new LEGListaConPI<EntradaMap<C, V>>();  
    EntradaMap<C, V> e = m.recuperarEntradaMin();  
    l.insertar(e);  
    for (int i = 1; i < m.talla(); i++) {  
        e = m.sucesorEntrada(e.getClave());  
        l.insertar(e);  
    }  
    return l;  
}
```

## 4. Map Ordenado - *Ejemplos de uso (cont.)*

- **Problema 2: Ordenación**

Diseña un método estático, genérico e iterativo `mapSort` que, con la ayuda de un `MapOrdenado`, ordene los elementos (`Comparable`) de un array `v`. Usa la pista del problema anterior para resolver este.

```
public static <C extends Comparable <C>> void mapSort (C[] v) {  
    MapOrdenado<C, C> m = new ABBMapOrdenado<C, C>();  
    for (int i = 0; i < v.length; i++) { m.insertar(v[i], v[i]); }  
    C x = m.recuperarMin(); v[0] = x;  
    for (int i = 1; i < v.length; i++) {  
        x = m.sucesor(x);  
        v[i] = x;  
    }  
}
```

## 4. Map Ordenado - *Ejemplos de uso (cont.)*

- **Problema 3: 2-SUM**

Diseña un método estático, genérico e iterativo `hayDosQueSuman` que, dados un array `v` de enteros y un entero `k`, determine si existen en `v` dos números cuya suma sea `k`. Usa un Map Ordenado como EDA auxiliar.

**Pista** – Ya sabes quién es el sucesor del mínimo de un conjunto. Consulta ahora la definición de predecesor de un elemento y responde: ¿quién es el predecesor del máximo de un conjunto? ¿Y el predecesor del predecesor del máximo? ¿Y el predecesor del predecesor del predecesor del máximo?

```
public static boolean hayDosQueSuman(Integer[] v, int k) {
    MapOrdenado<Integer, Integer> m = new ABBMapOrdenado<Integer, Integer>();
    for (int i = 0; i < v.length; i++) { m.insertar(v[i], i); }
    Integer min = m.recuperarMin(), max = m.recuperarMax();
    boolean esta = false;
    for (int i = 0; i < v.length - 1 && !esta; i++) {
        int suma = min + max;
        if (suma == k) { esta = true; }
        else if (suma < k) { min = m.sucesor(min); }
        else { max = m.predecesor(max); }
    }
    return esta;
}
```

**Ejercicio:** en la clase ABB, diseña un método toListaConPI que devuelva una Lista Con PI con los datos de un ABB en In-Orden, i.e. ordenados Asc.

**Solución:** modificar el método toStringInOrden() de la clase ABB, que es un **Recorrido** de los  $x = N$  nodos de un ABB en In-Orden con coste es  $\Theta(N)$

```
public ListaConPI<E> toListaConPI() {
    ListaConPI<E> res = new LEGListaConPI<E>();
    if (raiz != null) { toListaConPI(raiz, res); }
    return res;
}

// SII actual != null: actualiza res con los Datos del Nodo actual
// en In-Orden (Recorrido In-Orden con caso base Nodo Hoja implícito)

protected void toListaConPI(NodoABB<E> actual, ListaConPI<E> res) {
    if (actual.izq != null) {
        toListaConPI(actual.izq, res);
    }
    res.insertar(actual.dato);
    if (actual.der != null) {
        toListaConPI(actual.der, res);
    }
}
```

## 5. Implementaciones - *Relación entre modelos e implementaciones*

