Tema 4

Map Ordenado y Árbol Binario de Búsqueda (ABB)

Objetivos

- Aprender los conceptos básicos de árboles, árboles binarios y árboles binarios de búsqueda.
- Aprender los conceptos de recorridos de árboles y las operaciones básicas sobre árboles binarios de búsqueda
- Conocer los árboles equilibrados.

Contenidos

1. El modelo Map Ordenado: definición, coste estimado y ejemplos de uso
2. Conceptos sobre árboles
3. Árboles binarios. Recorridos
4. Árbol Binario de Búsqueda (Equilibrado)
5. La clase ABB
6. Diseño de ABBColaPrioridad y ABBMapOrdenado
7. Árboles equilibrados

1. Map Ordenado

- 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, etc.)
  - El coste máximo estimado de sus operaciones básicas es $\log_2 x$
  - En el estándar Java este modelo es *SortedMap*, de la Jerarquía *Collection*

1. Entradas en un Map Ordenado

```
public class EntradaMap<C extends Comparable<C>, V>
        implements Comparable<EntradaMap<C, V>> {
  private C clave;                          // Atributos: clave y valor de la entrada
  private V valor;
  public EntradaMap(C c, V v) { clave = c; valor = v; }    // Constructor
  public C getClave() { return clave; }                    // Consultores
  public V getValor() { return valor; }
  public void setClave(C nueva) { clave = nueva; }         // Modificadores
  public void setValor(V nuevo) { valor = nuevo; }
  @SuppressWarnings("unchecked")     // 2 entradas son iguales si tienen la misma clave
  public boolean equals(Object otra) {
    return this.clave.equals(((EntradaMap<C,V>) otra).clave);
  }                                  // Comparación de 2 entradas según sus claves
  public int compareTo(EntradaMap<C, V> otra) {
    return clave.compareTo(otra.clave);
  }
  public String toString() {     // Descripción de la entrada
    return "(" + this.clave + ", " + this.valor + ")";
  }
}
```

1. Map Ordenado

```
public interface MapOrdenado<C extends Comparable<C>, V>
       extends Map<C, V> {

  EntradaMap<C, V> recuperarEntradaMin(); // Entrada de clave mínima

  C recuperarMin(); // Clave mínima

  EntradaMap<C, V> recuperarEntradaMax(); // Entrada de clave máxima

  C recuperarMax(); // Clave máxima

  EntradaMap<C, V> sucesorEntrada(C c); // Siguiente entrada a c en el orden

  C sucesor(C c); // Siguiente clave a c en el orden

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

  C predecesor(C c); // Clave anterior a c en el orden

  ...
}
```

2. Conceptos de árboles

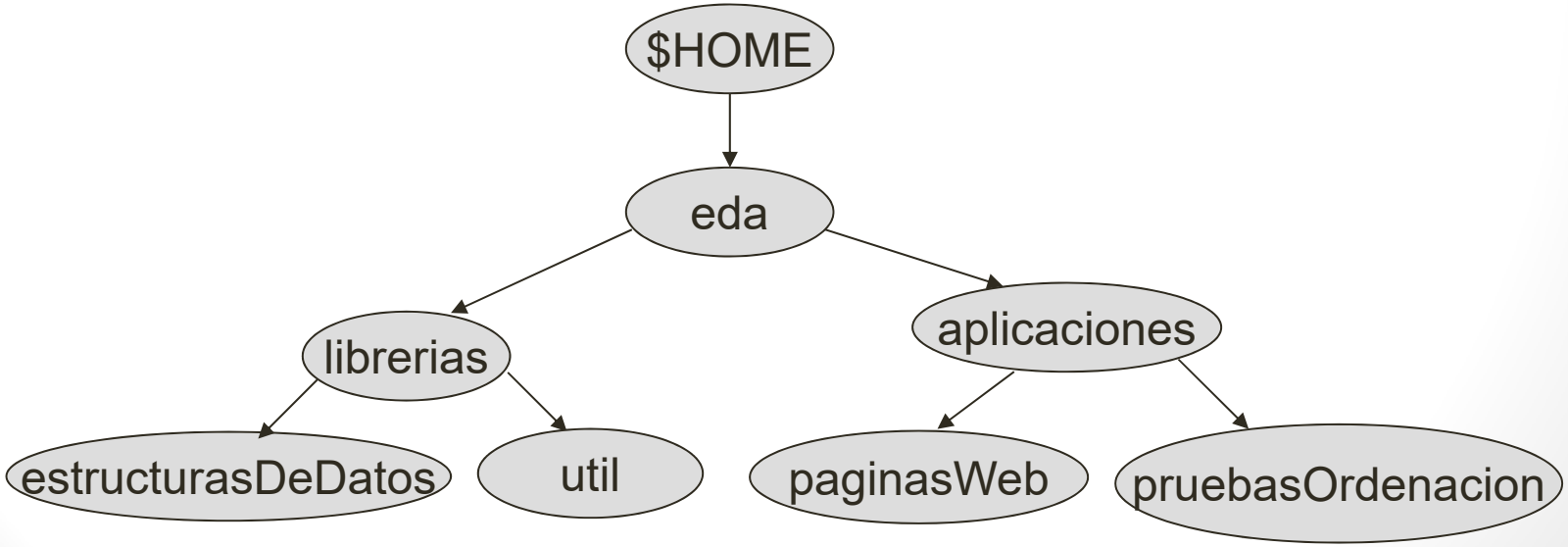
Modelos lineales vs. jerárquicos

- Las estructuras de datos lineales permiten describir conjuntos de datos que mantienen entre ellos relaciones de sucesión (o de predecesión).
  - Ejemplo: lista de clientes de una empresa, trabajos en la cola de impresión, etc.
- Los Árboles permiten representar estructuras jerárquicas entre conjuntos de datos.
  - Ejemplo: estructura de directorios, árbol genealógico, expresiones aritméticas, etc.

2. Conceptos de árboles

Estructuras jerárquicas

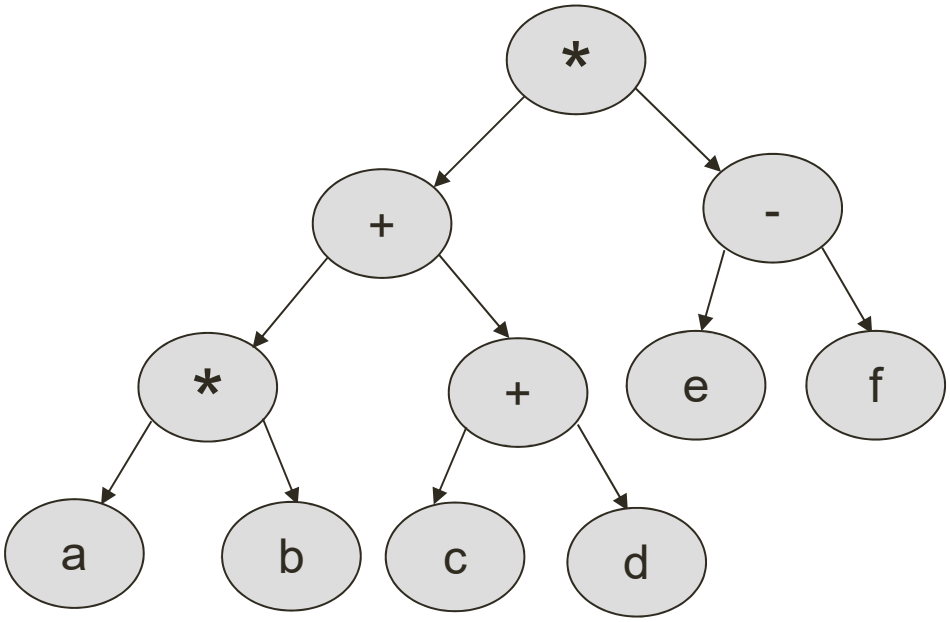
- En ocasiones los datos de una colección mantienen relaciones de tipo jerárquico, que no es posible expresar con una representación lineal.
  - Ejemplo 1: colección de directorios para las prácticas de EDA



2. Conceptos de árboles

Estructuras jerárquicas

- Ejemplo 2: el árbol que se muestra a continuación representa la expresión aritmética (((a*b)+(c+d))*(e-f)):



- Los Árboles son estructuras básicas para problemas de búsqueda y optimización (ajedrez, damas, sudoku, etc.)

2. Conceptos de árboles

Conceptos básicos sobre árboles

- Un *árbol* es una estructura jerárquica que se puede definir por medio de 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

2. Conceptos de árboles

Conceptos básicos sobre árboles

Nodo raíz:

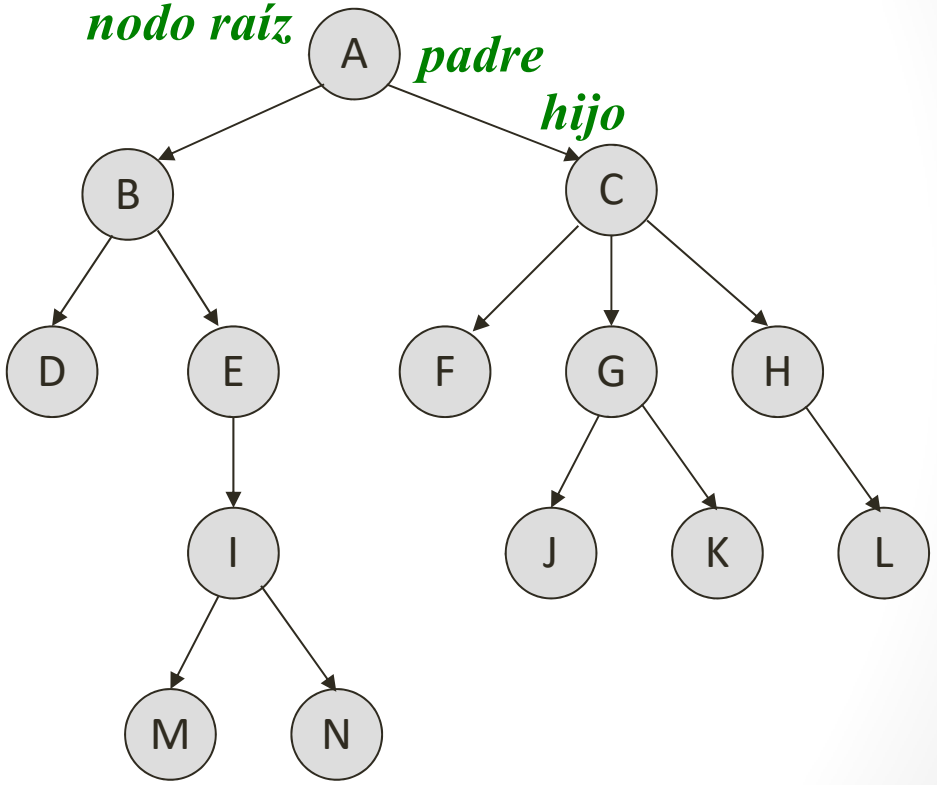
A

Nodos hoja:

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

Nodos internos:

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



2. Conceptos de árboles

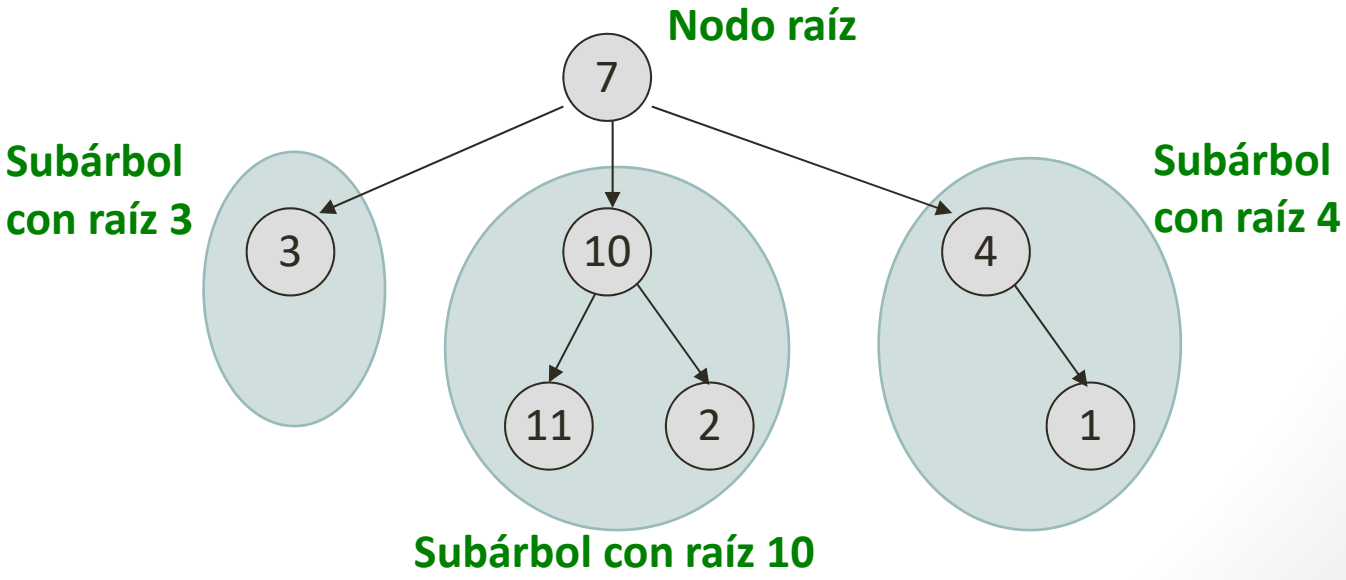
Longitud, profundidad y altura

- 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

2. 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



2. Conceptos de árboles

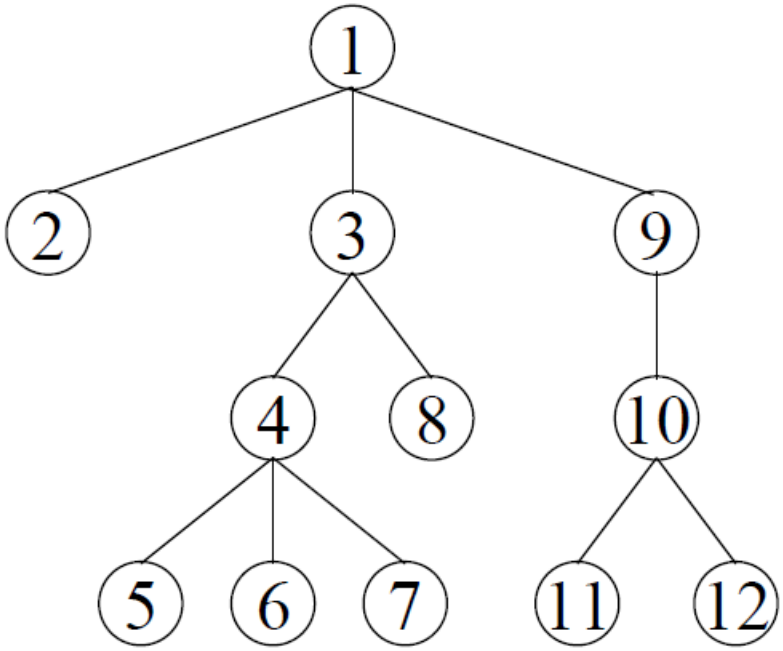
Representación de árboles generales

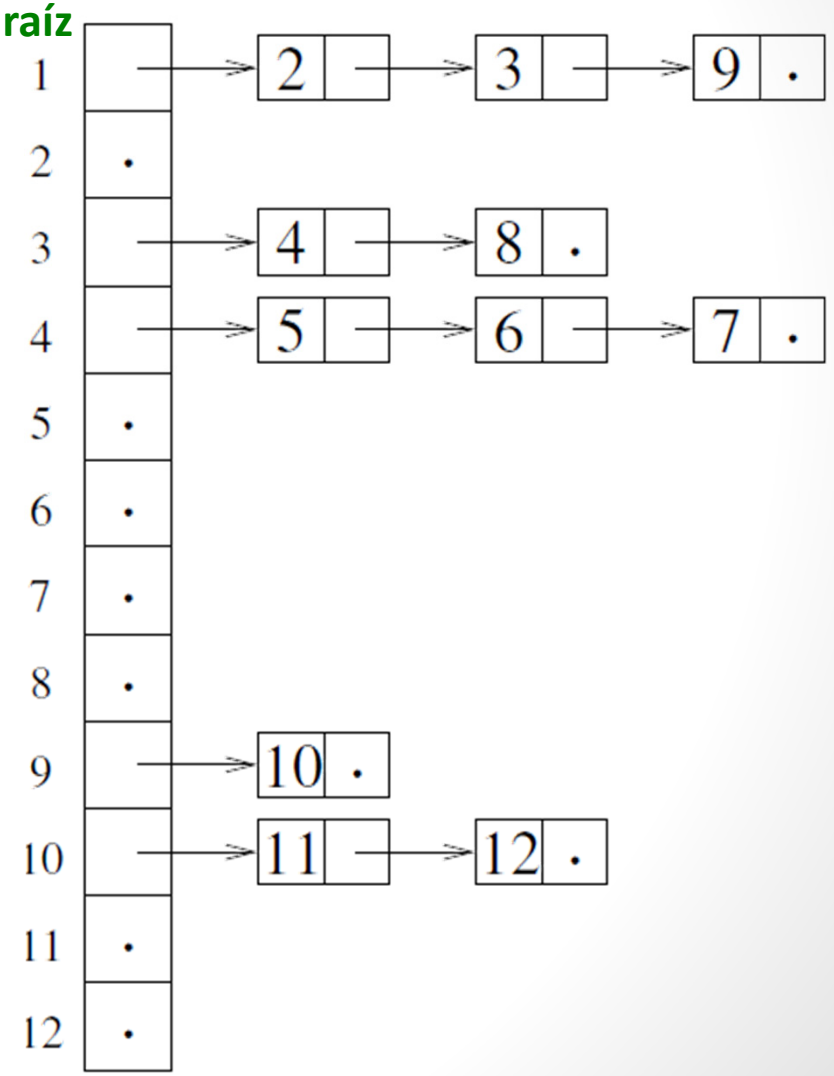
- Representación de árboles generales (número de hijos de los nodos sin acotar:
  - Listas (ordenadas) de hijos
  - Hijo más a la izquierda – hermano derecho
  - Con vectores y referencias al padre (*mf-sets*)
  - Otras…

2. Conceptos de árboles

Representación de árboles generales

- Ejemplo: listas ordenadas de hijos:

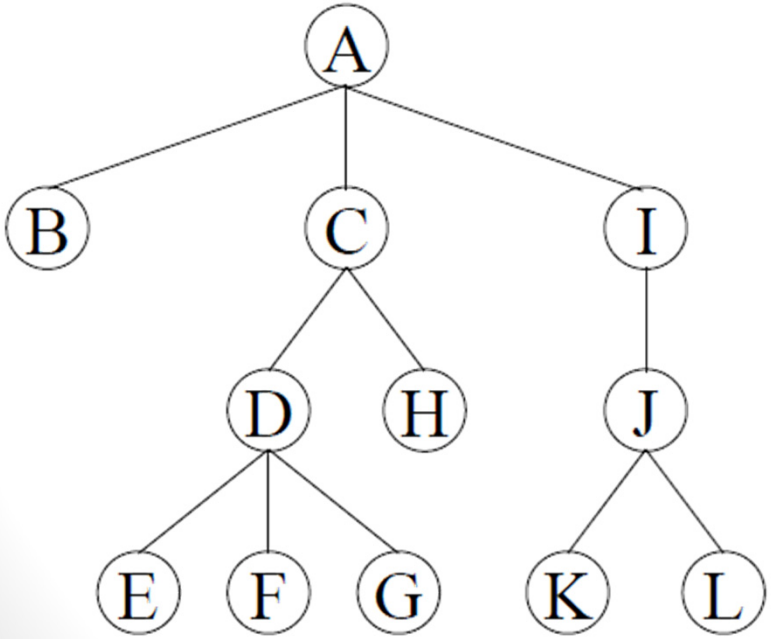




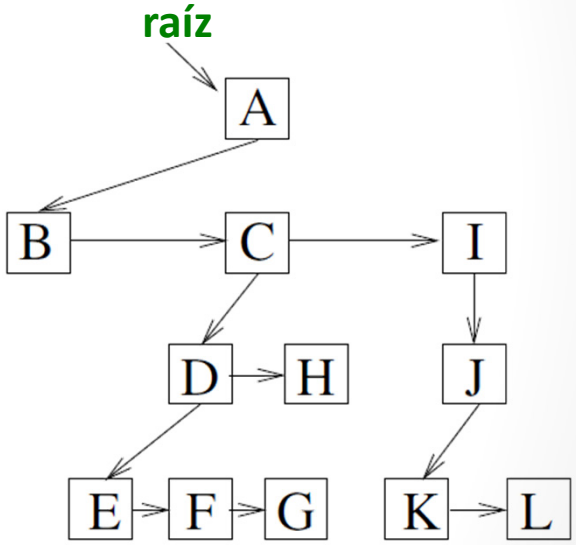
2. Conceptos de árboles

Representación

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



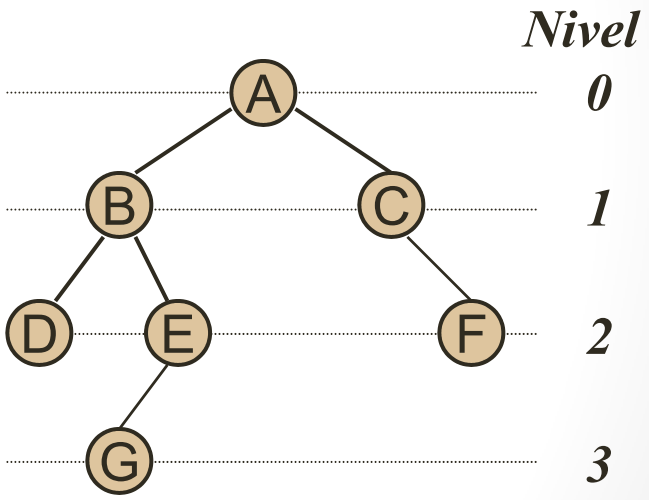
	hijo izq.	dato	her. der.
2	3	C	10
3	17	D	9
raíz → 4	8	A	.
....			
8	.	B	2
9	.	H	.
10	13	I	.
....			
12	.	G	.
13	14	J	.
14	.	K	16
15	.	F	12
16	.	L	7
17	.	E	15
....			



3. Árboles binarios

Definición

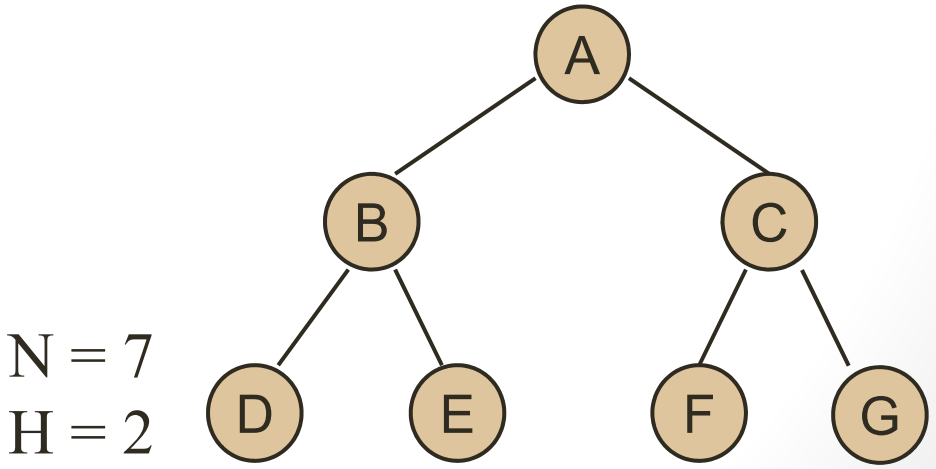
- Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos hijos (hijo izquierdo e hijo derecho)
- Propiedades:
  - El número máximo de nodos del nivel **i** es $\mathbf{2^i}$
  - En un árbol de altura **H**, el número máximo de nodos es: $\sum_{i=0..H} 2^i = 2^{H+1} - 1$
  - El 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: $(2^{H+1}-1) - (2^H) = 2^H - 1$



3. Árboles binarios

Conceptos básicos

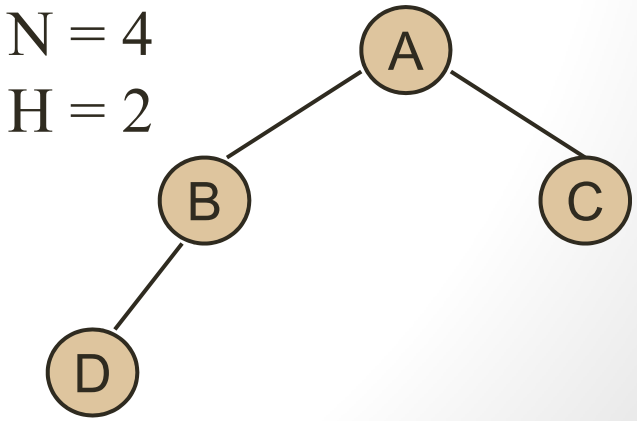
- Un **árbol binario** es **lleno** si tiene todos sus niveles completos
- Propiedades: sea *H* su altura y *N* su tamaño (número de nodos)
  - $H = \lfloor \log_2 N \rfloor$
  - $N = 2^{H+1} - 1$



3. Árboles binarios

Conceptos básicos

- Un **árbol binario *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)
  - $H \leq \lfloor \log_2 N \rfloor \rightarrow$ es un árbol ***equilibrado***
  - $2^H \leq N \leq 2^{H+1} - 1$

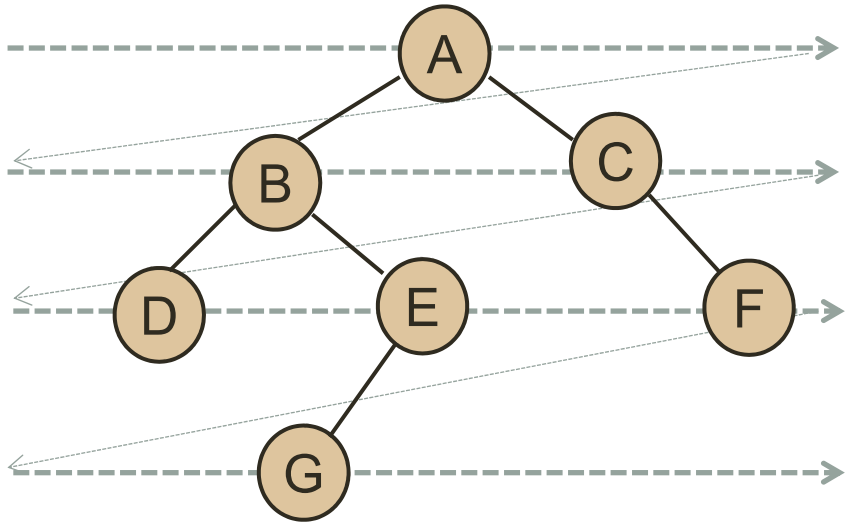


Nota: un árbol es *equilibrado* si los subárboles izquierdo y derecho de cualquier nodo tienen alturas que difieren como mucho en 1

3. Árboles binarios

Recorrido en anchura

- 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



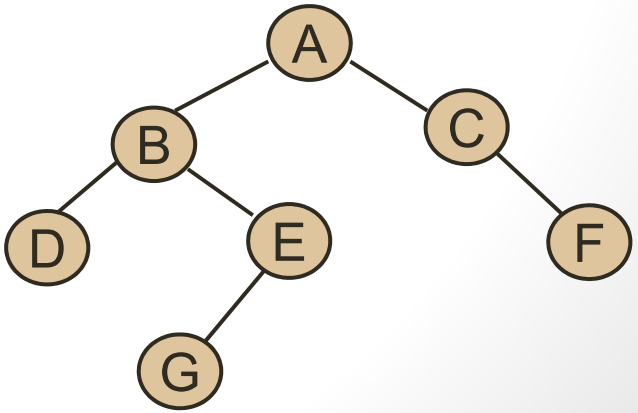
Por niveles: ABCDEFG

3. Árboles binarios

Recorridos en profundidad

- **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



4. Árboles binarios de búsqueda

Conceptos básicos

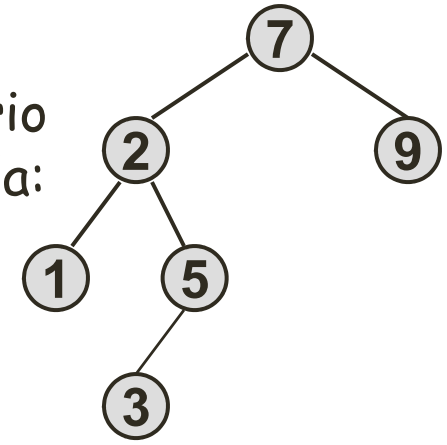
- Estructura de datos muy versátil, sirve para la implementación de diccionarios y de colas de prioridad
- Es una generalización de la búsqueda dicotómica
- Soporta eficientemente las operaciones de búsqueda, localizar el mínimo, el máximo, el predecesor y el successor
- También soporta eficientemente las operaciones de inserción y borrado

4. Árboles binarios de búsqueda

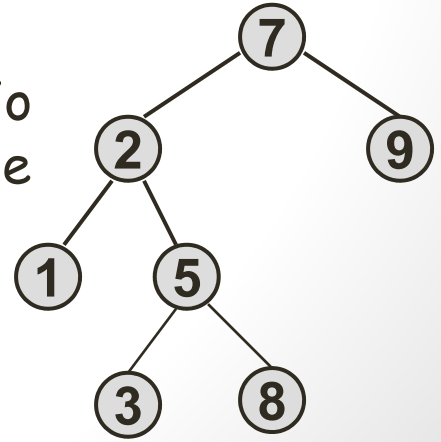
Definición

- Un árbol binario es de ***búsqueda*** si:
  - Los datos de su subárbol izquierdo son menores que el raíz
  - Los datos de su subárbol derecho son mayores que el raíz
  - Los subárboles izquierdo y derecho también son árboles binarios de búsqueda
- Si se imprime en in-orden resulta una secuencia ordenada

Árbol Binario de Búsqueda:

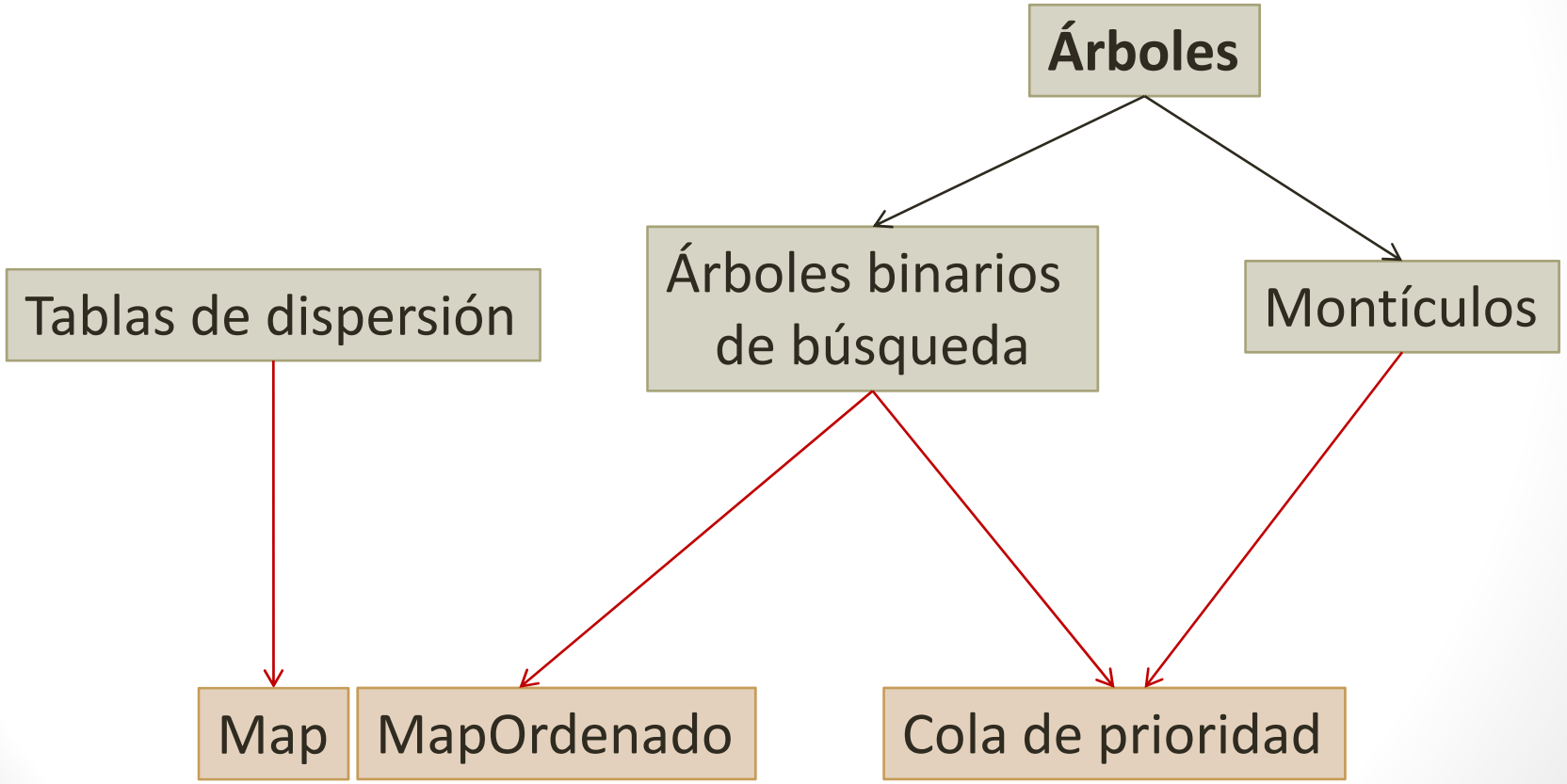


Árbol Binario (que no es de Búsqueda):



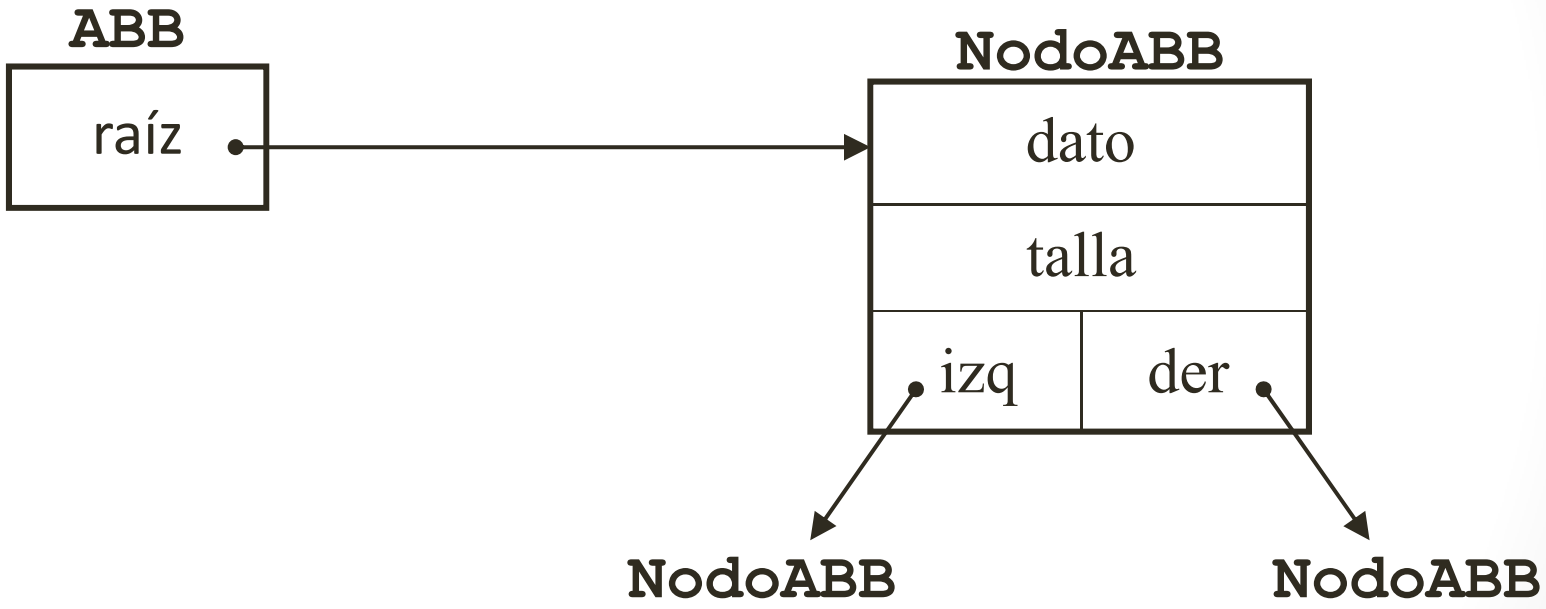
4. Árboles binarios de búsqueda

Relación entre modelos e implementaciones



5. Árboles binarios de búsqueda

Representación enlazada



5. Árboles binarios de búsqueda

La clase NodoABB

```
class NodoABB<E> {  // Nodos de un ABB
  E dato;           // Dato que contiene el nodo
  int talla;        // Tamaño del nodo (opcional)
  NodoABB<E> izq;   // Hijo izquierdo
  NodoABB<E> der;   // Hijo derecho
  // Constructor de un nodo sin hijos
  public NodoABB(E valor) {
    this(valor, null, null);
  }
  // Constructor de un nodo con un hijo izquierdo y derecho
  public NodoABB(E valor, NodoABB<E> izq, NodoABB<E> der) {
    dato = valor;
    this.izq = izq;
    this.der = der;
    talla = 1;
    if (izq != null) talla += izq.talla;
    if (der != null) talla += der.talla;
  }
}
```

5. Árboles binarios de búsqueda

La clase ABB

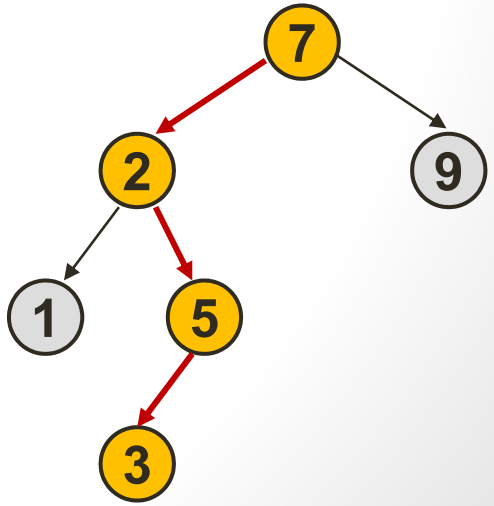
```
package librerias.estructurasDeDatos.jerarquicos;

public class ABB<E extends Comparable<E>> {
  // Atributos
  protected NodoABB<E> raiz;  // Nodo raíz del árbol

  /** Constructor de un ABB vacio **/
  public ABB() {
    raiz = null;
  }

  …
```

5. Árboles binarios de búsqueda

Búsqueda en un ABB

```
// Busca el dato x en el ABB y lo devuelve.
// Si no lo encuentra devuelve null
public E recuperar(E x) {
  NodoABB<E> nodo = raiz;
  while (nodo != null) {
    int resC = x.compareTo(nodo.dato);
    if (resC == 0) return nodo.dato;
    nodo = resC < 0 ? nodo.izq : nodo.der;
  }
  return null;
}
```

Ejemplo:
búsqueda
del dato 3

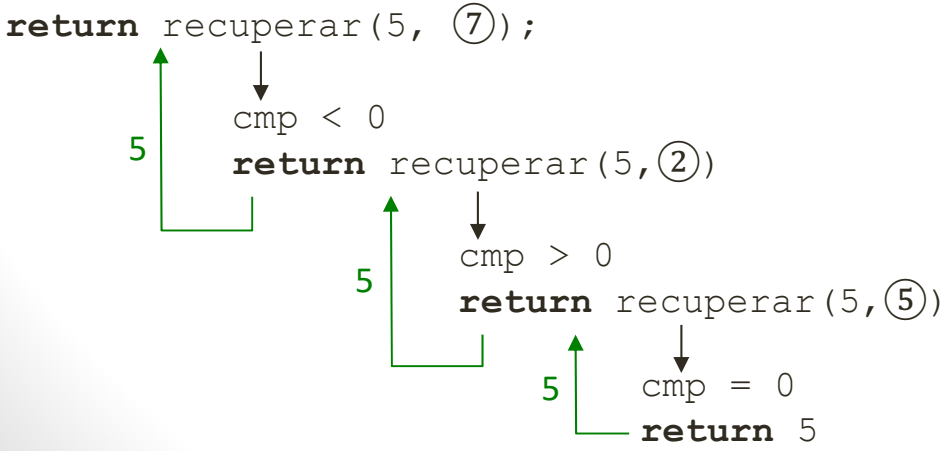


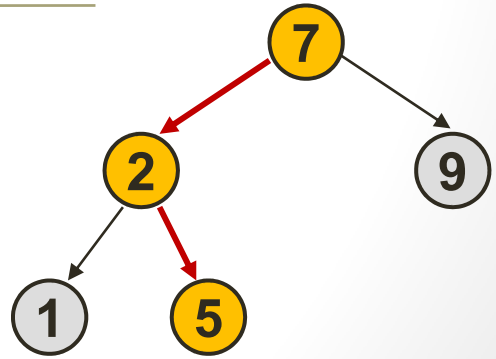
5. Árboles binarios de búsqueda

Búsqueda en un ABB (versión recursiva)

```
public E recuperar(E x) {  // Método lanzadera
  return recuperar(x, raiz);
}
protected E recuperar(E x, NodoABB<E> nodo) { // Recursivo
  if (nodo == null) return null;            // No encontrado
  int cmp = x.compareTo(nodo.dato);
  if (cmp < 0) return recuperar(x, nodo.izq);
  else if (cmp > 0) return recuperar(x, nodo.der);
  else return nodo.dato;                    // Dato x encontrado
}
```

Llamada inicial (buscamos el dato x=5):





Ejemplo: búsqueda del dato 5

5. Árboles binarios de búsqueda

Tamaño de un ABB

```
// Devuelve el número de elementos en el ABB
 public int talla() {
    return talla(raiz);
}

protected int talla(NodoABB<E> nodo) {
   if (nodo == null) return 0;
   else return nodo.talla;
}

// Indica si el ABB está vacío
public boolean esVacio() {
   return raiz == null;
}
```

Si no tuviéramos el atributo *talla* en los nodos, necesitaríamos un atributo *talla* en el *ABB*.

5. Árboles binarios de búsqueda

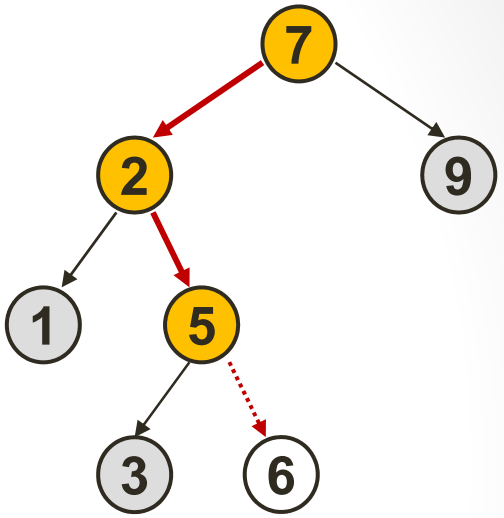
Inserción en un ABB (versión recursiva)

```
// Actualiza el dato x en el ABB, si no está lo inserta
public void insertar(E x) {        // Lanzadera
  raiz = insertar(x, raiz);
}

protected NodoABB<E> insertar(E x, NodoABB<E> nodo) {
  if (nodo == null) return new NodoABB<E>(x);
  int cmp = x.compareTo(nodo.dato);
  if (cmp < 0) nodo.izq = insertar(x, nodo.izq);
  else if (cmp > 0) nodo.der = insertar(x, nodo.der);
  else nodo.dato = x;
  nodo.talla = 1 + talla(nodo.izq) + talla(nodo.der);
  return nodo;
}
```

5. Árboles binarios de búsqueda

Inserción en un ABB

- Ejemplo: insertar el 6

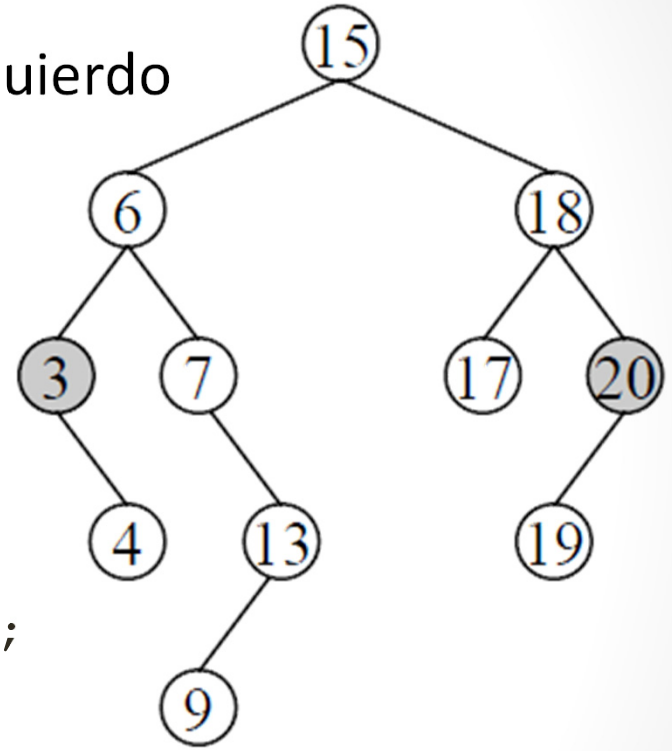



```
raiz = insertar(6,⑦);
        cmp < 0
        ⑦.izq = insertar(6,②)
                cmp > 0
                ②.der = insertar(6,⑤)
                        cmp > 0
                        ⑤.der = insertar(6,null)
                                return ⑥
                        ⑤.talla=3
                        return ⑤
                ②.talla=5
                return ②
        ⑦.talla=7
        return ⑦
```

5. Árboles binarios de búsqueda

Mínimo y máximo en un ABB

- 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.*

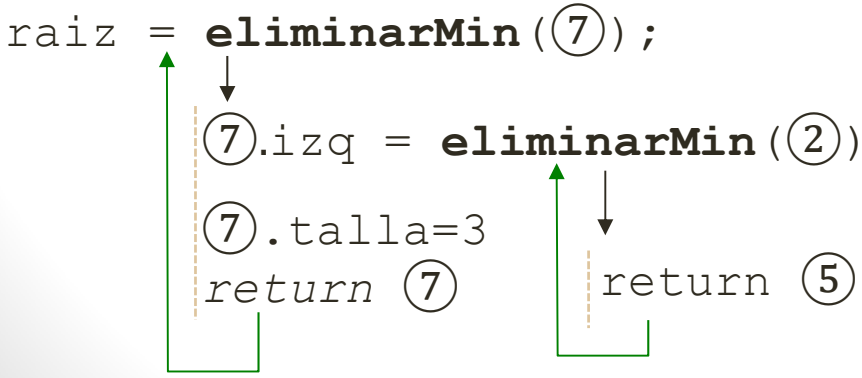


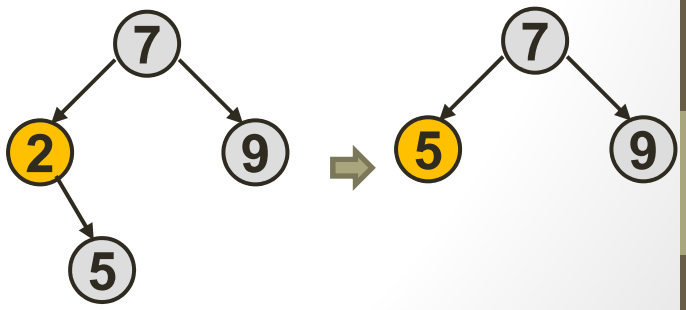

```
// Devuelve el mínimo
public E recuperarMin() {
  if (raiz == null) return null;
  return recuperarMin(raiz).dato;
}

protected NodoABB<E> recuperarMin(NodoABB<E> nodo) {
  if (nodo.izq == null) return nodo;
  else return recuperarMin(nodo.izq);
}
```

5. Árboles binarios de búsqueda

Borrado del mínimo en un ABB

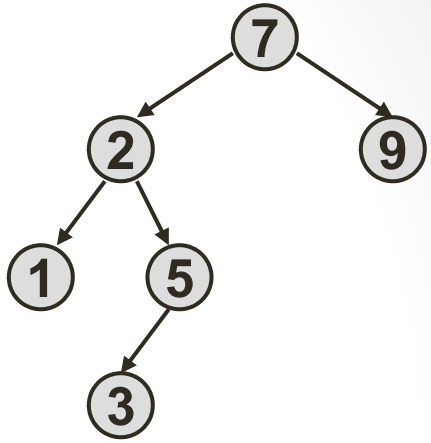
```
public E eliminarMin() {
   E min = recuperarMin();
   if (min != null) raiz = eliminarMin(raiz);
   return min;
}

protected NodoABB<E> eliminarMin(NodoABB<E> nodo) {
   if (nodo.izq == null) return nodo.der;
   nodo.izq = eliminarMin(nodo.izq);
   nodo.talla--;
   return nodo;
}
```





5. Árboles binarios de búsqueda

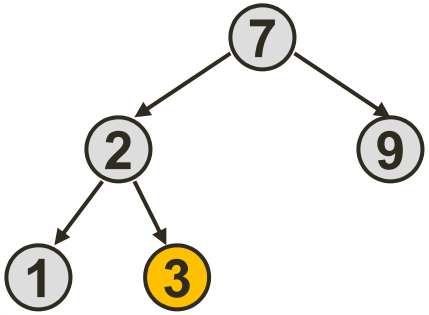
Borrado en un ABB



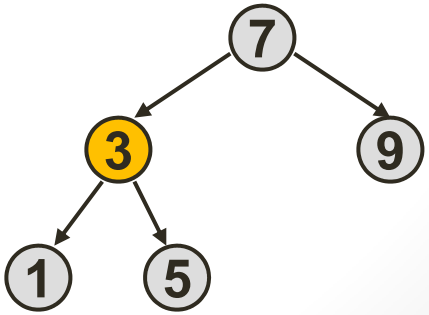
Posibles casos:

a) El nodo a eliminar no tiene hijos
Ejemplo: el 3 (se elimina sin problemas)

b) El nodo a eliminar tiene un hijo
Ejemplo: el 5
Su hijo ocupa su posición:



c) El nodo a eliminar tiene dos hijos
Ejemplo: el 2
El mínimo de su subárbol derecho ocupa su posición:



5. Árboles binarios de búsqueda

Borrado en un ABB

```
// Elimina el nodo que contiene el dato x
public void eliminar(E x) {
  raiz = eliminar(x, raiz);
}

protected NodoABB<E> eliminar(E x, NodoABB<E> nodo) {
  if (nodo == null) return nodo;     // Dato x no encontrado
  int cmp = x.compareTo(nodo.dato);
  if (cmp < 0) nodo.izq  = eliminar(x, nodo.izq);
  else if (cmp > 0) nodo.der = eliminar(x, nodo.der);
  else {              // Dato x encontrado -> borramos el nodo
    if (nodo.der == null) return nodo.izq;  // Solo un hijo
    if (nodo.izq == null) return nodo.der;  // Solo un hijo
    nodo.dato = recuperarMin(nodo.der).dato;   // Dos hijos
    nodo.der = eliminarMin(nodo.der);
  }
  nodo.talla = 1 + talla(nodo.izq) + talla(nodo.der);
  return nodo;
}
```

5. Árboles binarios de búsqueda

Recorridos en profundidad

- La forma natural de implementar los recorridos en profundidad es recursiva:

```
public String preOrden() {     // Lanzadera
  return preOrden(raiz);
}
private String preOrden(NodoABB<E> nodo) {
  if (nodo == null) return "";
  return actual.dato.toString() + "\n" +
    preOrden(actual.izq) + preOrden(actual.der);
}
```

- Los métodos para imprimir en Post-Orden y en In-Orden son muy similares (sólo cambia el orden de las instrucciones)

5. Árboles binarios de búsqueda

Recorrido en anchura

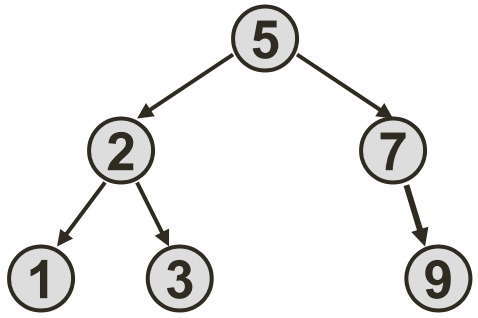
- El recorrido por niveles, al ser iterativo, utiliza una *Cola* como estructura auxiliar

```
public String porNiveles() {
  if (raiz == null) return "";
  Cola<NodoABB<E>> q = new ArrayCola<NodoABB<E>>();
  q.encolar(raiz);
  String res = "";
  while (!q.esVacia()) {
    NodoABB<E> nodo = q.desencolar();
    res += nodo.dato.toString() + "\n";
    if (nodo.izq != null) q.encolar(nodo.izq);
    if (nodo.der != null) q.encolar(nodo.der);
  }
  return res;
}
```

5. Árboles binarios de búsqueda

Cálculo del sucesor/predecesor

- Si un nodo tiene subárbol derecho, el sucesor de dicho nodo es el mínimo de su subárbol derecho
- Sino, el sucesor es el ascendiente por la derecha más cercano
- El sucesor de un nodo equivale al siguiente nodo que se obtendría en un recorrido in-orden del árbol:



sucesor(5) = 7
sucesor(1) = 2
sucesor(3) = 5
sucesor(9) = *null*

5. Árboles binarios de búsqueda

Cálculo del sucesor/predecesor

```
/** Obtiene el sucesor de e en un ABB.
 *  Si no existe tal sucesor, devuelve null para
 *  advertirlo */
public E sucesor(E e) {
  E sucesor = null;
  NodoABB<E> nodo = this.raiz;
  while (nodo != null) {
    int cmp = nodo.dato.compareTo(e);
    if (cmp > 0) {
      sucesor = nodo.dato;
      nodo = nodo.izq;
    } else nodo = nodo.der;
  }
  return sucesor;
}
```

5. Árboles binarios de búsqueda

Coste de las operaciones

Coste promedio / *EDA*	buscar(x)	insertar(x)	minimo ()	eliminarMin()
Lista Enlazada / Array	$\Theta(N)$	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$
LEG Ordenada	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$
Array Ordenado	$\Theta(\log N)$	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$
ABB	$\Theta(\log N)$	$\Theta(\log N)$	$\Theta(\log N)$	$\Theta(\log N)$

- El coste de las operaciones en un ABB está en función de la altura del árbol (*h*)
- La altura *h* está entre $\Omega(\log_2 n)$ y O(n)
- En el peor caso (ABB desbalanceado), los costes son lineales

6. La clase ABBMapOrdenado

Propuesta de implementación

- Una posible implementación de *ABBMapOrdenado* vía **composición**, sería ...

```
public class ABBMapOrdenado<C extends Comparable<C>, V>
        implements MapOrdenado<C, V> {
  protected ABB<EntradaMap<C, V>> abb;
  public ABBMapOrdenado() { abb = new ABB<EntradaMap<C, V>>(); }
  public boolean esVacio() { return abb.esVacio(); }
  public int talla() { return abb.talla(); }
  public V recuperar(C c) {
    EntradaMap<C, V> e;
    e = abb.recuperar(new EntradaMap<C,V>(c, null));
    return e != null ? e.getValor() : null;
  }
```

6. La clase ABBMapOrdenado

Propuesta de implementación

```
public V insertar(C c, V v) {
  EntradaMap<C,V> nuevo = new EntradaMap<C,V>(c,v);
  EntradaMap<C,V> ant = abb.recuperar(nuevo);
  abb.insertar(nuevo);
  return ant == null ? null : ant.getValor();
}
public V eliminar(C c) {
  EntradaMap<C, V> e;
  e = abb.recuperar(new EntradaMap<C,V>(c, null));
  if (e == null) return null;
  abb.eliminar(e);
  return e.getValor();
}
...
```

6. La clase ABBColaPrioridad

Propuesta de implementación

- Los métodos *eliminarMin* y *recuperarMin* se heredan tal cual de la clase *ABB*
- Es necesario definir el método *esVacia* ya que cambia de género (en *ABB* se llama *esVacio*)
- Hace falta sobrescribir el método *insertar* para permitir la inserción de elementos duplicados

```
public interface
  ColaPrioridad<E extends Comparable<E>> {
  void  insertar(E e);
  E  eliminarMin();
  E  recuperarMin();
  boolean esVacia();
}
```

6. La clase ABBColaPrioridad

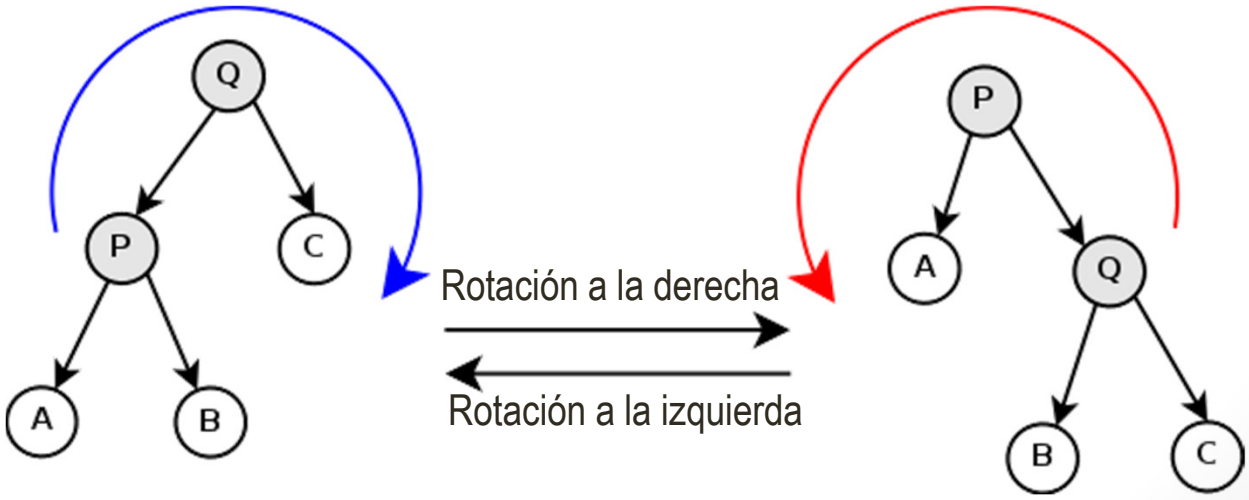
Propuesta de implementación

```
public class ABBColaPrioridad<E extends Comparable<E>>
        extends ABB<E>
        implements ColaPrioridad<E> {
  public boolean esVacia() { return super.esVacio(); }
  public void insertar(E x) { // Inserción con duplicados
    raiz = insertar(x, raiz);
  }
  protected NodoABB<E> insertar(E x, NodoABB<E> nodo) {
    if (nodo == null) return new NodoABB<E>(x);
    int cmp = x.compareTo(nodo.dato);
    if (cmp <= 0) nodo.izq = insertar(x, nodo.izq);
    else nodo.der = insertar(x, nodo.der);
    nodo.talla++;
    return nodo;
  }
}
```

7. Árboles equilibrados

Introducción

- Los árboles balanceados son estructuras de datos basadas en árboles que incluyen información y/o operaciones adicionales para conseguir un equilibrio en los árboles
- Su funcionamiento se basa en efectuar rotaciones: se intercambian nodos y subárboles en un árbol binario de búsqueda para conseguir otro equivalente



7. Árboles equilibrados

Los árboles balanceados más conocidos

- Árboles AVL
  - Almacenan el *factor de equilibrio* en cada nodo
  - Permanecen balanceados en todo momento
- Árboles rojo-negro
  - Cada nodo tiene un atributo indicando su color (rojo o negro)
  - Al igual que los AVL, permanecen equilibrados en todo momento
- Splay-trees
  - Suben (mediante rotaciones) el elemento insertado/buscado a la raíz y con ello se mantiene balanceado
- Day–Stout–Warren (DSW)
  - Consigue balancear el *ABB* (esté como esté previamente) en O(n)

Bibliografía

- Data structures, algorithms, and applications in Java, *Sahni* (capítulos 12 y 15)
- Estructuras de datos en Java, *Weiss* (capítulos 17 y 18)
- Data Structures and Algorithms in Java (4th edition), *Goodrich y Tamassia* (capítulo 10)