

## Tema 6

# Árboles de búsqueda

### 6.1. El TAD Árbol Binario de Búsqueda

### 6.2. Implementación de árboles binarios de búsqueda

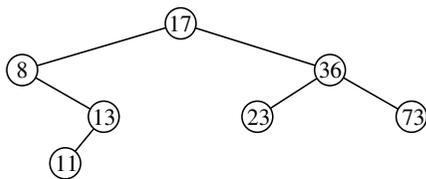
### 6.3. Árboles balanceados: Árbol AVL

## El TAD Árbol Binario de Búsqueda

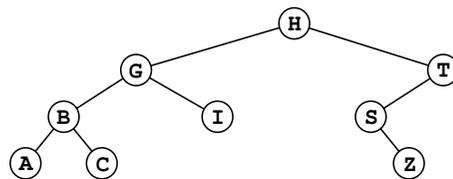
Un **árbol binario de búsqueda** es un árbol binario con una **propiedad** adicional:

“para todo nodo del árbol, los elementos de su subárbol izquierdo son menores que el elemento del nodo, y los del derecho son mayores”.

Esta propiedad necesita que para todo par de posibles elementos del árbol se pueda determinar cuál es mayor y cuál menor.



árbol binario de búsqueda



árbol binario que no es de búsqueda

## El TAD Árbol Binario de Búsqueda

Existen estructuras de datos y TADs especialmente apropiados para realizar búsquedas de información.

En ellos, la representación de los datos está orientada a que al menos las operaciones de búsqueda e inserción sean lo más eficientes posible.

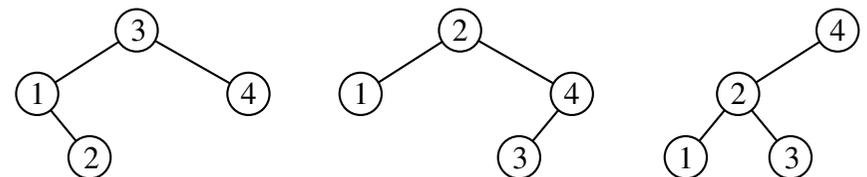
Los árboles de búsqueda constituyen el grupo mayoritario de tales estructuras y TADs.

Un uso importante de los árboles de búsqueda es la construcción de índices que hacen eficiente el acceso a información almacenada en ficheros y bases de datos.

Los árboles binarios de búsqueda son el tipo más básico de árbol de búsqueda.

## El TAD Árbol Binario de Búsqueda

Un conjunto de elementos puede representarse mediante distintos árboles binarios de búsqueda.



Un recorrido en inorden de un árbol binario de búsqueda produce la secuencia ordenada de los elementos de los nodos.

El recorrido en inorden de los distintos árboles que representan un mismo conjunto produce la misma secuencia de nodos.

## El TAD Árbol Binario de Búsqueda

**TAD** Árbol Binario de Búsqueda

**Elementos:**  $\mathcal{AB} = \{ \emptyset \} \cup \{ (I, x, D) \mid x \in \mathcal{X} \wedge I, D \in \mathcal{AB} \}$

$$\forall A \in \mathcal{AB}, X(A) = \begin{cases} \emptyset, & \text{si } A = \emptyset \\ \{ x \} \cup X(I) \cup X(D), & \text{si } A = (I, x, D) \neq \emptyset \end{cases}$$

En  $\mathcal{X}$  existe un **orden lineal**  $<$ :

$$\forall x, y \in \mathcal{X}, x \neq y \Rightarrow (x < y \wedge y \not< x) \vee (y < x \wedge x \not< y).$$

$$\mathcal{ABB} = \{ \emptyset \} \cup$$

$$\{ (I, x, D) \in \mathcal{AB} \mid I, D \in \mathcal{ABB} \wedge y < x, y \in X(I) \wedge x < z, z \in X(D) \}$$

Esta definición permite una aparición como máximo de cada elemento de  $\mathcal{X}$  en un árbol binario de búsqueda. Podrían considerarse múltiples ocurrencias.

## El TAD Árbol Binario de Búsqueda

**Operaciones:** Dados  $A \in \mathcal{ABB}$  y  $x \in \mathcal{X}$ :

**Crear()** : Devuelve  $\emptyset$ .

**Destruir(A)** : Elimina  $A$ .

**EstáVacío(A)** : Devuelve **cierto** si  $A = \emptyset$ , y **falso** si  $A \neq \emptyset$ .

**Buscar(A,x)** : Devuelve  $x$ , si  $x \in X(A)$ .

**Mínimo(A)** : Devuelve  $z \in X(A)$  tal que  $z < y, y \in X(A) - \{ z \}$ , si  $A \neq \emptyset$ .

**Máximo(A)** : Devuelve  $z \in X(A)$  tal que  $y < z, y \in X(A) - \{ z \}$ , si  $A \neq \emptyset$ .

Si  $A = \emptyset$  ó  $x \notin X(A)$ , **Buscar**, **Mínimo** y **Máximo** podrían devolver **falso** ó  $c \notin \mathcal{X}$ .

**Insertar(A,x)** : Devuelve  $A' \in \mathcal{ABB}$  tal que  $X(A') = X(A) \cup \{ x \}$ .

**Eliminar(A,x)** : Devuelve  $A' \in \mathcal{ABB}$  tal que  $X(A') = X(A) - \{ x \}$ .

## EL TAD Árbol Binario de Búsqueda

Los algoritmos de **Buscar**, **Mínimo** y **Máximo** se derivan directamente de la propiedad de los árboles binarios de búsqueda:

**Buscar:** Si un (sub)árbol no está vacío, el elemento buscado está:

- ▷ en la raíz,
  - ▷ en su subárbol izquierdo, o
  - ▷ en su subárbol derecho,
- en función de la comparación del elemento buscado con el elemento de la raíz.

**Mínimo:** Si un (sub)árbol no está vacío, su elemento mínimo está:

- ▷ en la raíz, si su subárbol izquierdo está vacío, o
- ▷ en su subárbol izquierdo, en caso contrario.

**Máximo:** Si un (sub)árbol no está vacío, su elemento máximo está:

- ▷ en la raíz, si su subárbol derecho está vacío, o
- ▷ en su subárbol derecho, en caso contrario.

## EL TAD Árbol Binario de Búsqueda

- ▷ Repitiendo sucesivamente estos procesos en cada nodo resultante de la comparación anterior, se encuentra el elemento o éste no está en el árbol.

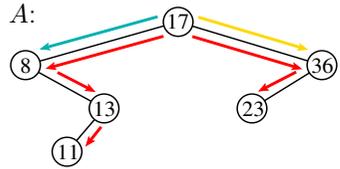
Las operaciones de **Insertar** y **Eliminar** se pueden realizar de diferentes maneras, en función de las propiedades que se desee exigir a los árboles resultantes.

- ▷ Primero, veremos sus algoritmos básicos, que son muy eficientes, pero sólo garantizan que los árboles resultantes de su ejecución serán binarios de búsqueda.
- ▷ Después, veremos como garantizar alguna propiedad adicional en los árboles resultantes.

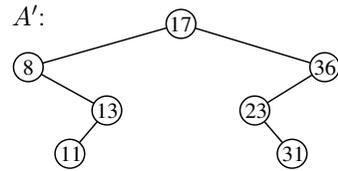
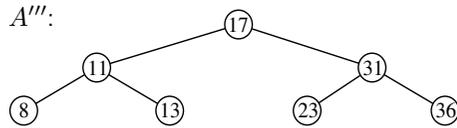
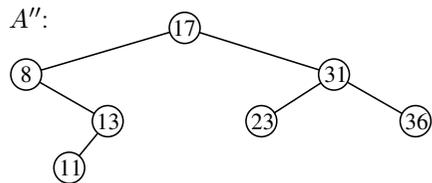
## El TAD Árbol Binario de Búsqueda

Ejemplos:

Mínimo(A) Buscar(A,11)  
Máximo(A) Buscar(A,23)

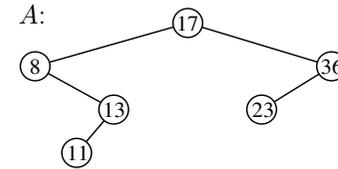


Insertar(A,31)

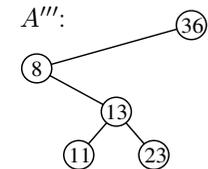
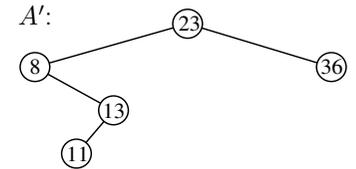
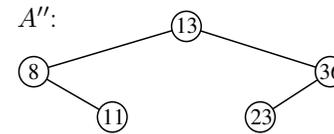


## El TAD Árbol Binario de Búsqueda

Ejemplos:



Eliminar(A,17)



## Implementación de árboles binarios de búsqueda

En la implementación siguiente, como hasta ahora, se define el tipo base de la clase `ArbolBB` como:

```
typedef int TBABB;
```

Los valores de este tipo serán la información que se almacenará en los nodos de un árbol binario de búsqueda.

Más aún, los valores de `TBABB` van a ser utilizados por los algoritmos para guiar sus acciones, a diferencia de lo que ocurría hasta ahora, que tan sólo se almacenaban como información de los nodos.

Con estos valores se van a realizar comparaciones, que podrían ser de más tipos, pero se van a reducir sólo a `<` (las restantes pueden expresarse mediante este operador).

Como `TBABB` es `int` estas comparaciones las resuelve el operador predefinido de `C++`. Y también resuelve las de los demás tipos básicos.

Pero, ¿y si se desea que la información de los nodos de un árbol binario de búsqueda sean objetos más generales?

## Implementación de árboles binarios de búsqueda

Se necesitará que estos puedan compararse con la misma sintaxis que los valores de los tipos predefinidos, para no tener que modificar el código de las operaciones.

En concreto, se necesitará poder escribir `x < y` para comprobar si `x` es menor que `y`, cuando `x` e `y` sean valores de tipos predefinidos u objetos cualesquiera.

Para poder reutilizar el código con cualquier clase que se quiera utilizar como `TBABB`, se debe sobrecargar el operador `<` en la clase:

```
class Fecha {
public:
    bool operator<(const Fecha & f) const
        {return(a<f.a || a==f.a && m<f.m || a==f.a && m==f.m && d<f.d);}
private:
    int d, m, a;
};
```

Cada clase debe establecer su propio orden de los objetos en función de los valores de sus atributos.

Cambiando a `typedef Fecha TBABB`; el resto del código puede reutilizarse.

## Implementación de árboles binarios de búsqueda

```
1 class ArbolBB {                                // arbol_bb.h
2     public:
3         typedef int TBABB;
4         static const TBABB NO_ESTA;
5         ArbolBB();
6         ~ArbolBB();
7         ArbolBB(const ArbolBB & der);
8         ArbolBB & operator=(const ArbolBB & der);
9
9         void Vaciar();
10        bool EstaVacio() const;
11        const TBABB & Buscar(const TBABB & item) const;
12        const TBABB & Minimo() const;
13        const TBABB & Maximo() const;
14        void Insertar(const TBABB & item);
15        void Eliminar(const TBABB & item);
```

## Implementación de árboles binarios de búsqueda

Buscar, Minimo y Maximo devuelven una referencia constante a un objeto TBABB:

- ▷ el encontrado, si la búsqueda tiene éxito, o
- ▷ uno especial NO\_ESTA, si ésta fracasa.

Para ello, en la definición de la clase ArbolBB, se declara la **constante de clase** NO\_ESTA de tipo TBABB.

Su valor será un valor no válido entre los valores del tipo TBABB que vayan a representarse en la aplicación, y éste se define en arbol\_bb.c++.

Las **variables y constantes de clase se comparten** por todos los objetos de una clase. Hay un único espacio en memoria para cada variable o constante de clase.

⇒ Los cambios que realice un objeto, afectan a todos.

Se declaran como atributos en la definición de la clase, precediéndolas de `static`.

Y en el caso de las constantes, su valor se asigna fuera de la definición de la clase (en el fichero de implementación de los métodos de la clase).

## Implementación de árboles binarios de búsqueda

```
16     private:
17         struct Nodo {
18             TBABB elem;    Nodo * hizq, * hder;
19             explicit Nodo(const TBABB & dato, Nodo * hi=0, Nodo * hd=0)
20                 { elem = dato; hizq = hi; hder = hd; }
21         };
22         typedef Nodo * Enlace;
23         Enlace raiz;
24         void Vaciar(Enlace p);
25         Enlace Copiar(Enlace p);
26         Enlace Buscar(Enlace p, const TBABB & item) const;
27         Enlace Minimo(Enlace p) const;
28         Enlace Maximo(Enlace p) const;
29         void Insertar(Enlace & p, const TBABB & item);
30         void Eliminar(Enlace & p, const TBABB & item);
31     };
```

## Implementación de árboles binarios de búsqueda

Acceso a una variable o constante de clase:

- ▷ Dentro de la clase (definición y métodos), como cualquier otro atributo.
- ▷ En el código ajeno a la clase, si la variable o constante de clase es pública, o privada pero el código ajeno es amigo:

```
ArbolBB A, *B; ... A.NO_ESTA ... B->NO_ESTA      ArbolBB::NO_ESTA
```

Cada operación del TAD Árbol Binario de Búsqueda se implementa mediante un par de métodos sobrecargados, uno publico y otro privado.

Cada método publico recibe como argumentos la información externa necesaria para realizar sus funciones y llama a su correspondiente método privado, pasándole esta información y el puntero a la raíz del árbol.

Los métodos privados están implementados recursivamente, ya que resulta más cómodo hacerlo así dada la naturaleza recursiva de los árboles.

- ▷ En general, no cabe esperar un desbordamiento de la pila del sistema.
- ▷ No obstante, algunos árboles degenerados podrían llegar a producirlo.

## Implementación de árboles binarios de búsqueda

```
1 #include "arbol_bb.h" // arbol_bb.c++
2 const ArbolBB::TBABB ArbolBB::NO_ESTA = -1;
3 ArbolBB::ArbolBB() { raiz = 0; } t_ArbolBB(n) = Θ(1)

4 const ArbolBB::TBABB & ArbolBB::Buscar(const TBABB & item) const {
5     Enlace p = Buscar(raiz,item);
6     if (p == 0) Tamaño del problema
7         return(NO_ESTA); n: número de nodos del árbol binario
8     else
9         return(p->elem); t_Buscar(n) = Ω(1), O(n)
10 }
11 ArbolBB::Enlace ArbolBB::Buscar(Enlace p, const TBABB & item) const {
12     if (p == 0) return(0);
13     else if (item < p->elem) return(Buscar(p->hizq,item));
14     else if (p->elem < item) return(Buscar(p->hder,item));
15     else return(p);
16 }
```

## Implementación de árboles binarios de búsqueda

La implementación básica de Insertar comienza buscando el elemento en el árbol.

Si se encuentra, no se hace nada más ya que estamos considerando que sólo puede haber una ocurrencia como mucho de cada elemento.

Si no se encuentra, el puntero nulo que finaliza la búsqueda sería una rama en el camino desde la raíz al nodo con dicho elemento si:

- ▷ el elemento se encontrase en el árbol, y
- ▷ el árbol mantuviese intacta toda su estructura, excepto el subárbol que se ramificaría desde ese puntero nulo.

Así, creando un nodo para el elemento y enlazándolo con ese puntero nulo, se realiza una inserción sencilla y el árbol resultante sigue siendo un árbol binario de búsqueda.

Si se consideran árboles binarios de búsqueda que puedan contener más de una ocurrencia de cada elemento, pueden implementarse con un nodo para cada ocurrencia o un único nodo para todas las ocurrencias con un contador de ocurrencias (si es preciso, otra estructura almacenará la información diferente de cada ocurrencia).

## Implementación de árboles binarios de búsqueda

```
17 const ArbolBB::TBABB & ArbolBB::Minimo() const {
18     if (raiz == 0)
19         return(NO_ESTA);
20     else
21         return(Minimo(raiz)->elem); t_Minimo(n) = Ω(1), O(n)
22 }
23 ArbolBB::Enlace ArbolBB::Minimo(Enlace p) const
24     { return(p->hizq == 0 ? p : Minimo(p->hizq)); }

25 const ArbolBB::TBABB & ArbolBB::Maximo() const {
26     if (raiz == 0)
27         return(NO_ESTA);
28     else
29         return(Maximo(raiz)->elem); t_Maximo(n) = Ω(1), O(n)
30 }
31 ArbolBB::Enlace ArbolBB::Maximo(Enlace p) const
32     { return(p->hder == 0 ? p : Maximo(p->hder)); }
```

## Implementación de árboles binarios de búsqueda

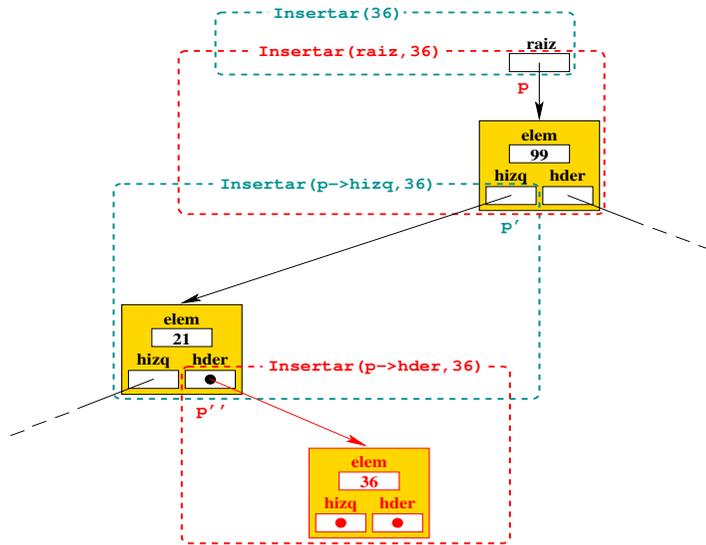
```
33 void ArbolBB::Insertar(const TBABB & item)
34     { Insertar(raiz,item); } t_Insertar(n) = Ω(1), O(n)

35 void ArbolBB::Insertar(Enlace & p, const TBABB & item) {
36     if (p == 0) p = new Nodo(item);
37     else if (item < p->elem) Insertar(p->hizq,item);
38     else if (p->elem < item) Insertar(p->hder,item);
39     // else // El item ya está en el árbol
40 }
```

El parámetro formal p del método privado Insertar es una referencia a un puntero, es decir, un alias para el parámetro real raiz, p->hizq o p->hder empleado en la llamada.

Asignando el puntero devuelto por new a p, éste se asigna físicamente al espacio en memoria del parámetro real (que es el mismo que el de p).

## Implementación de árboles binarios de búsqueda



## Implementación de árboles binarios de búsqueda

La implementación básica de Eliminar es un poco más compleja que las anteriores.

Su coste temporal es:  $t_{\text{Eliminar}}(n) = \Omega(1), \mathcal{O}(n)$

También empieza buscando el elemento en el árbol. Si no lo encuentra, no hace nada.

Si lo encuentra, pueden darse tres situaciones distintas (que pueden reducirse a dos):

**Caso 1.** El nodo a eliminar es una hoja.

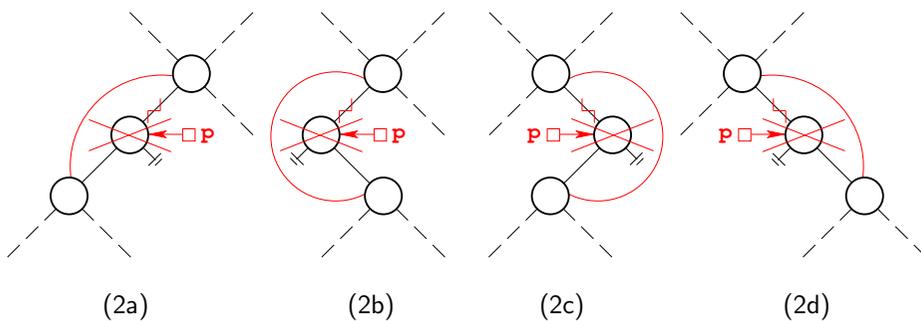
- ▷ Simplemente se libera el nodo y se asigna el puntero nulo en el correspondiente puntero hizq o hder del padre del nodo o en raiz.



## Implementación de árboles binarios de búsqueda

**Caso 2.** El nodo a eliminar tiene sólo un hijo (izquierdo o derecho).

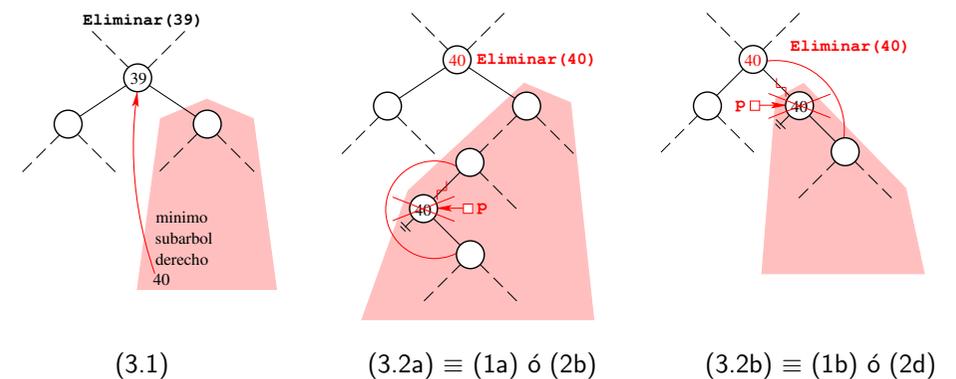
- ▷ El puntero hizq o hder del padre del nodo a eliminar, o el puntero raiz, pasa a apuntar a su único nieto (hizq o hder). Después se libera el nodo a eliminar.



## Implementación de árboles binarios de búsqueda

**Caso 3.** El nodo a borrar tiene dos hijos.

- ▷ Se sustituye el elemento del nodo a eliminar por el mínimo de su subárbol derecho.
- ▷ Así, ya no se elimina este nodo, y pasa a eliminarse el nodo del elemento mínimo de su subárbol derecho, que será una eliminación en el caso 1 ó 2.



## Implementación de árboles binarios de búsqueda

En el caso 1, si el elemento a eliminar está en una hoja, no tiene descendientes. No hay descendientes que resituar respecto a sus antepasados.

En el caso 2, si el nodo a eliminar tiene un sólo hijo, el nodo es:

- (2a) Predecesor inmediato de su padre y sucesor inmediato de sus descendientes.
- (2b) Predecesor inmediato de sus descendientes, que son predecesores inmediatos de su padre.
- (2c) Sucesor inmediato de sus descendientes, que son sucesores inmediatos de su padre.
- (2d) Sucesor inmediato de su padre y predecesor inmediato de sus descendientes.

La reasignación de punteros:

- ▷ mantiene la situación relativa entre el padre y los descendientes del nodo eliminado,
- ▷ y provoca cambios mínimos en la estructura del árbol (realizando pocas acciones).

## Implementación de árboles binarios de búsqueda

```

41 ArbolBB::~ArbolBB() { Vaciar(); }
42 void ArbolBB::Vaciar() {
43     if (raiz != 0) {
44         if (raiz->hizq != 0) Vaciar(raiz->hizq);
45         if (raiz->hder != 0) Vaciar(raiz->hder);
46         delete raiz;
47         raiz = 0;
48     }
49 }
50 void ArbolBB::Vaciar(Enlace p) {
51     if (p->hizq != 0) Vaciar(p->hizq);
52     if (p->hder != 0) Vaciar(p->hder);
53     delete p;
54 }
55 bool ArbolBB::EstaVacio() const
56     { return(raiz == 0); }
    
```

$t_{\text{ArbolBB}}(n) = \Theta(n)$

Sólo es necesario asignar el puntero nulo a raiz.

No es necesario asignar el puntero nulo a los punteros hizq y hder (tras liberar los nodos a los que apuntan), ya que el nodo que los contiene se libera a continuación.

$t_{\text{Vaciar}}(n) = \Theta(n)$

$t_{\text{EstaVacio}}(n) = \Theta(1)$

## Implementación de árboles binarios de búsqueda

En el caso 3, el elemento a eliminar se sustituye en el nodo por su sucesor en inorden y se conserva el nodo.

Después se elimina el elemento y el nodo sucesor en inorden, produciéndose una reestructuración del árbol como en los casos 1 ó 2.

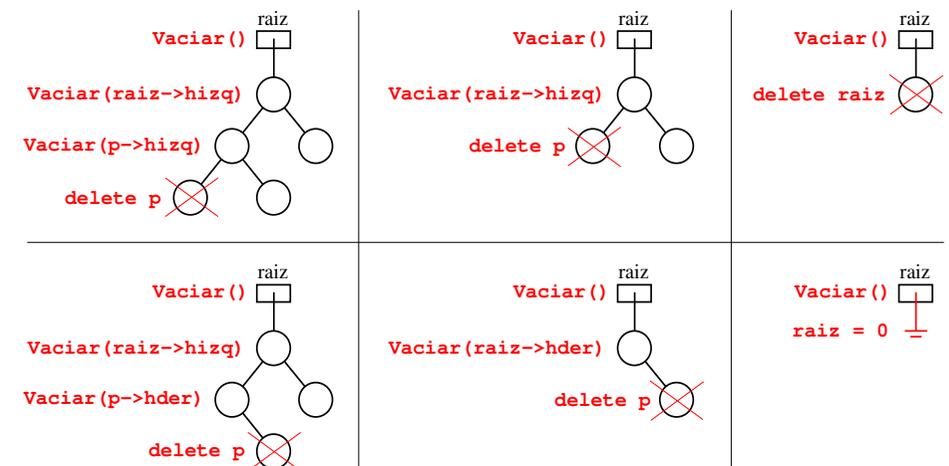
Estos algoritmos básicos de inserción y eliminación en un árbol binario de búsqueda:

- ▷ Son muy eficientes, porque realizan muy pocas acciones (encontrando la posición de inserción o el nodo a eliminar, y modificando sólo un puntero).
- ▷ Conservan intacta la estructura del árbol inicial (excepto la mínima modificación asociada al nodo insertado o eliminado).

No obstante, sólo garantizan que el árbol resultante será binario de búsqueda.

Según el orden de inserción y eliminación de los elementos, podrían producirse árboles degenerados de gran altura, lo que actúa en contra de su pretendida eficiencia.

## Implementación de árboles binarios de búsqueda



## Implementación de árboles binarios de búsqueda

```

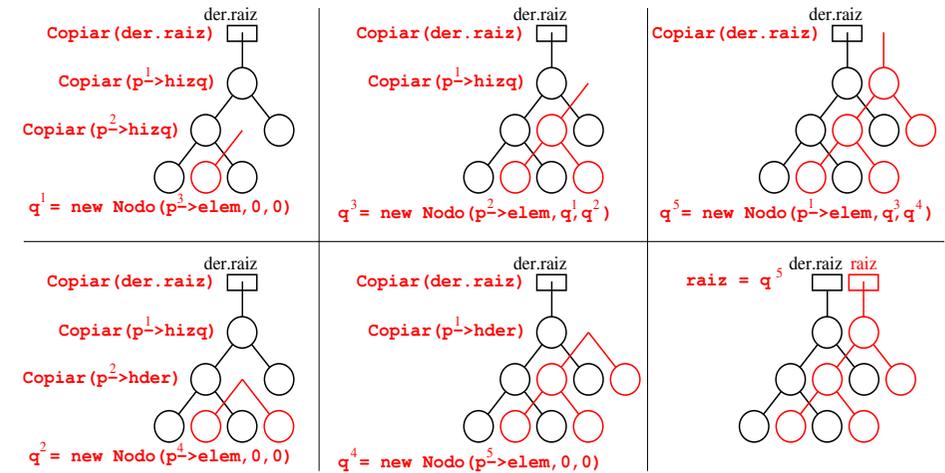
57 ArbolBB::ArbolBB(const ArbolBB & der)
58   { raiz = 0; operator=(der); }            $t_{\text{ArbolBB}}(n) = \Theta(n)$ 

59 ArbolBB & ArbolBB::operator=(const ArbolBB & der) {
60   if (this != &der) {
61     Vaciar();
62     raiz = der.raiz != 0 ? Copiar(der.raiz) : 0;
63   }
64   return(*this);            $t_{\text{operator}}(n) = \Omega(1), \mathcal{O}(n)$ 
65 }

66 ArbolBB::Enlace ArbolBB::Copiar(Enlace p) {
67   Enlace hi = p->hizq != 0 ? Copiar(p->hizq) : 0;
68   Enlace hd = p->hder != 0 ? Copiar(p->hder) : 0;
69   Enlace q = new Nodo(p->elem,hi,hd);
70   return(q);
71 }            $t_{\text{Copiar}}(n) = \Theta(n)$ 

```

## Implementación de árboles binarios de búsqueda



## Árboles balanceados: Árbol AVL

El coste en el peor de los casos de las operaciones Buscar, Mínimo, Máximo, Insertar y Eliminar en un árbol binario de búsqueda de  $n$  nodos es  $\mathcal{O}(h)$ , donde  $h$  es la altura del árbol y  $\lfloor \log_2 n \rfloor \leq h \leq n - 1$ .

Está demostrado que la altura promedio de un nodo de un árbol binario de búsqueda es  $\mathcal{O}(\log n)$ .

Estrictamente, esta cota para el caso promedio es válida:

- ▷ para árboles binarios de búsqueda formados sólo con inserciones (sin eliminaciones),
- ▷ y asumiendo que todas las secuencias de inserción son equiprobables.

No está claro que mediante inserciones y eliminaciones se obtenga la misma distribución de árboles binarios de búsqueda que sólo mediante inserciones.

- ▷ El algoritmo de eliminación dado favorece la obtención de subárboles izquierdos más profundos que los derechos (reemplaza un nodo con otro del subárbol derecho).

## Árboles balanceados: Árbol AVL

La realización de eliminaciones anula la garantía de la cota promedio logarítmica, aunque en general cabe esperar operaciones eficientes en un árbol binario de búsqueda.

Aún asumiendo sólo búsquedas e inserciones, en la práctica pueden encontrarse secuencias de entrada que conducen a árboles con excesiva altura:

- ▷ valores ordenados (de menor a mayor o viceversa),
- ▷ alternando sucesivamente valores máximos y mínimos, etc.

Interesa conseguir árboles binarios de búsqueda equilibrados o balanceados, independientemente de la secuencia de entrada, para que las operaciones sean eficientes.

El equilibrio en un árbol binario de búsqueda puede definirse de formas diversas.

- ▷ El ideal de árbol equilibrado, que las distintas definiciones tratan de aproximar, es un árbol binario que resulta completo al eliminar las hojas del último nivel.

## Árboles balanceados: Árbol AVL

Realizando acciones adicionales, a la inserción y eliminación estándar, se pueden lograr árboles binarios de búsqueda equilibrados.

**Árboles binarios de búsqueda aleatorizados:** Al azar se decide cuándo un elemento se inserta según el método estándar y cuándo se fuerza a que se convierta en la raíz del árbol.

Equivale a reordenar aleatoriamente la secuencia de entrada al principio, y después realizar las inserciones mediante la operación estándar, con lo que se obtiene la garantía de la cota promedio logarítmica.

**Árboles biselados:** Cada vez que se accede a un nodo se reestructura el árbol para que éste se convierta en la raíz. Con ello, se tiende a equilibrar el árbol.

Algunas reestructuraciones del árbol requerirán mucho esfuerzo, si el nodo accedido es muy profundo, pero éste esfuerzo se amortizará en las siguientes reestructuraciones, que requerirán poco o ningún esfuerzo.

## Árboles balanceados: Árbol AVL

Se puede garantizar que el coste amortizado de  $m$  operaciones sobre un árbol de  $n$  nodos será  $\mathcal{O}(m \log n)$ , aunque individualmente el coste de cada una de ellas no está acotado por  $\mathcal{O}(\log n)$ .

**Árboles binarios de búsqueda con condiciones estructurales adicionales:**

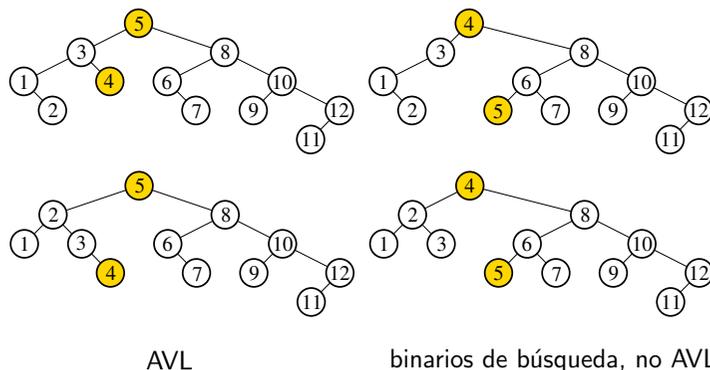
- ▷ Árboles AVL,
- ▷ Árboles 2-3,
- ▷ Árboles rojos-negros, etc.

Las condiciones estructurales adicionales obligan a mantener en todo momento el equilibrio en los árboles, con lo que consiguen que el coste en el peor de los casos de cada operación sea  $\mathcal{O}(\log n)$ .

## Árboles balanceados: Árbol AVL

Un **árbol AVL** es un árbol binario de búsqueda con una condición de equilibrio adicional: **en cada nodo, la altura de sus subárboles izquierdo y derecho puede diferir como mucho en 1**. (Adelson-Velskii y Landis, 1962)

La altura de un árbol vacío se establece a  $-1$ .



## Árboles balanceados: Árbol AVL

Está demostrado que **la altura de un árbol AVL de  $n$  nodos es  $\mathcal{O}(\log n)$** .

- ▷ Por tanto, todo acceso a un nodo para realizar alguna operación tiene la garantía de un coste logarítmico en el peor de los casos.

La cuestión esencial que aparece cuando se realizan operaciones con árboles AVL es la siguiente: **tras insertar o eliminar un nodo en un árbol AVL** (con los métodos estándar), **¿el árbol binario de búsqueda resultante es AVL?**

- ▷ No siempre. A veces la condición de equilibrio se mantendrá en todos los nodos.
- ▷ Pero, otras veces se romperá la condición de equilibrio en algunos nodos.

No obstante, **la condición de equilibrio puede restaurarse**, en todos los nodos en los que se haya perdido, mediante unas operaciones conocidas como **rotaciones**.

## Árboles balanceados: Árbol AVL

### Inserción

Tras una inserción, sólo los nodos del camino desde el nodo insertado a la raíz pueden haber perdido su equilibrio (ya que sólo estos han modificado sus subárboles).

Tras insertar el nodo, se recorre de abajo a arriba dicho camino comprobando la condición de equilibrio en cada nodo.

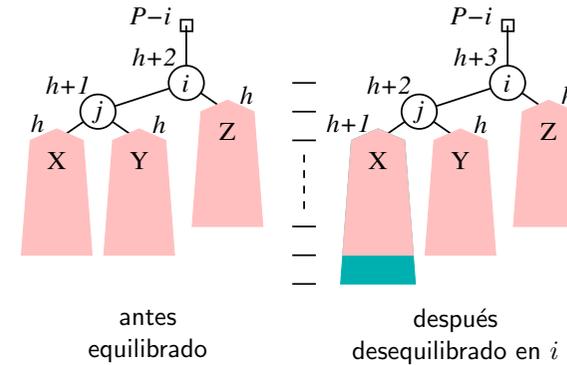
▷ Cuando se encuentra el primer nodo que la incumple, se debe reequilibrar el subárbol cuya raíz es este nodo.

Esta operación tiene el efecto de restaurar la condición de equilibrio no sólo en este nodo, sino en todo el árbol. Es el único reequilibrado que hay que realizar.

Hay distintos casos en los que se puede romper el equilibrio en un nodo, y cada uno tiene su solución.

## Árboles balanceados: Árbol AVL

### Caso 1. Inserción en el subárbol izquierdo del hijo izquierdo de $i$



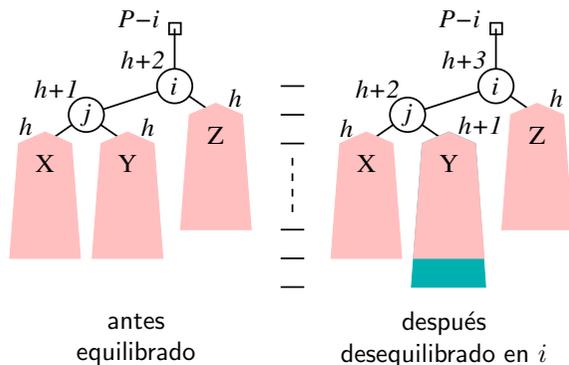
Se inserta un nodo en X que incrementa su altura.

⇒ El hijo izquierdo de  $i$  es dos niveles más alto que el hijo derecho.

El equilibrio se restaura con una **rotación simple a la derecha**.

## Árboles balanceados: Árbol AVL

### Caso 2. Inserción en el subárbol derecho del hijo izquierdo de $i$



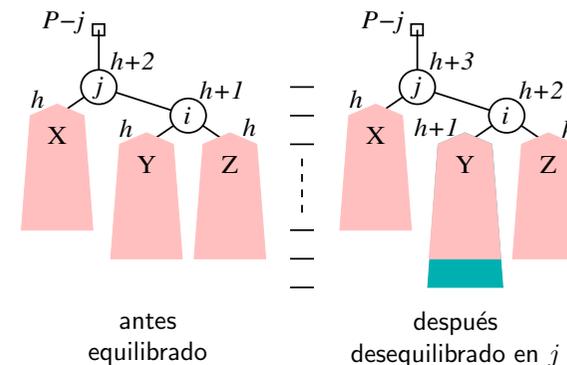
Se inserta un nodo en Y que incrementa su altura.

⇒ El hijo izquierdo de  $i$  es dos niveles más alto que el hijo derecho.

El equilibrio se restaura con una **rotación doble izquierda-derecha**.

## Árboles balanceados: Árbol AVL

### Caso 3. Inserción en el subárbol izquierdo del hijo derecho de $j$ (simétrico a 2)



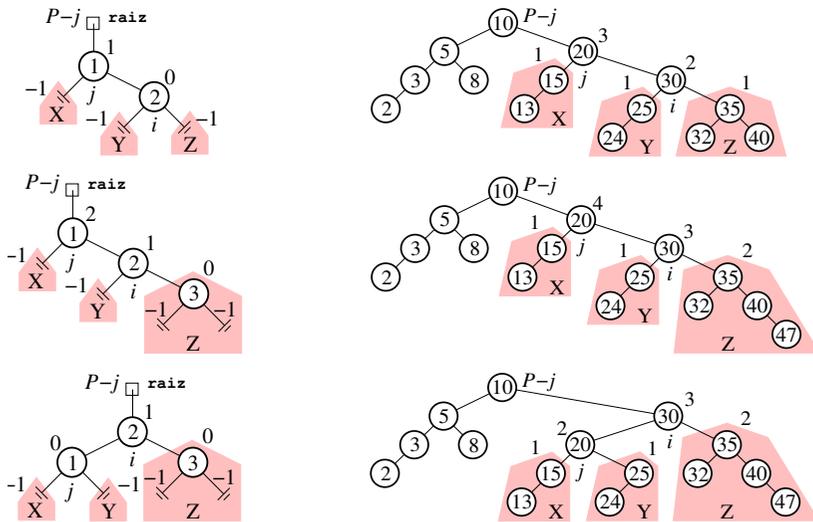
Se inserta un nodo en Y que incrementa su altura.

⇒ El hijo derecho de  $j$  es dos niveles más alto que el hijo izquierdo.

El equilibrio se restaura con una **rotación doble derecha-izquierda**.

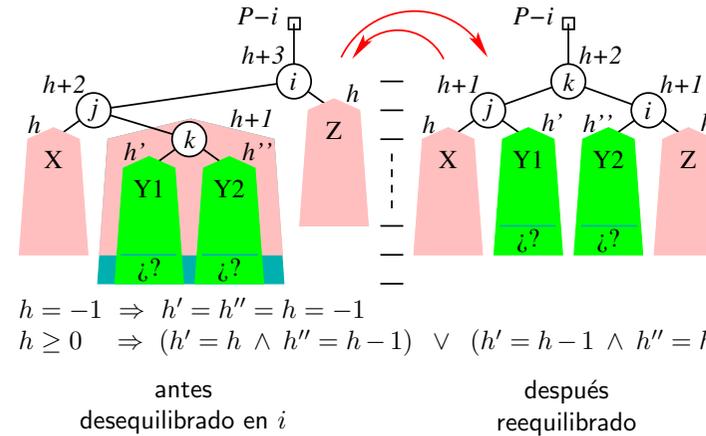


## Árboles balanceados: Árbol AVL



## Árboles balanceados: Árbol AVL

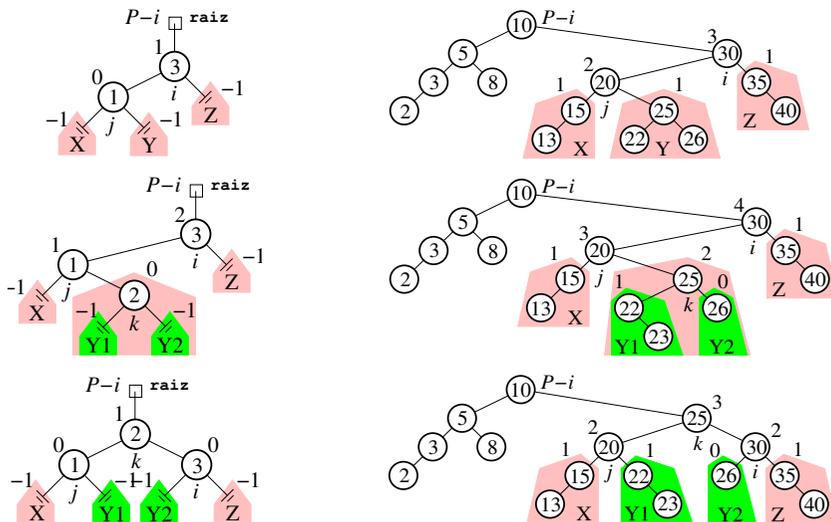
### Rotación doble izquierda-derecha (caso 2)



Equivale a:

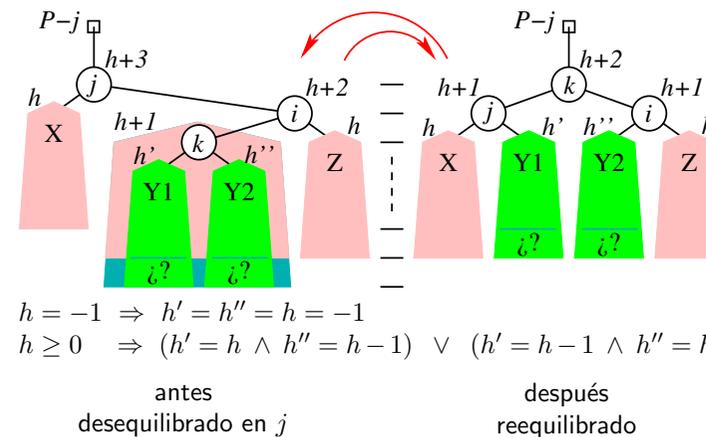
- 1) una rotación simple a la izquierda entre  $k$  y  $j$ , y a continuación,
- 2) una rotación simple a la derecha entre  $k$  e  $i$ .

## Árboles balanceados: Árbol AVL



## Árboles balanceados: Árbol AVL

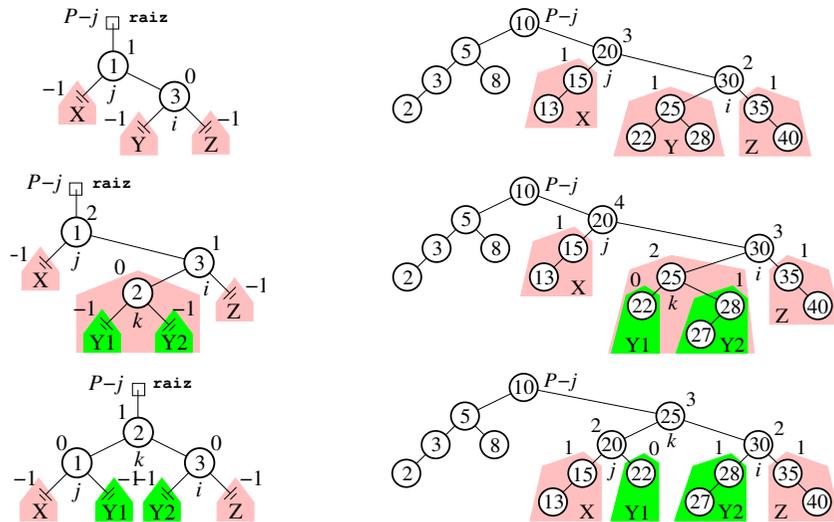
### Rotación doble derecha-izquierda (caso 3)



Equivale a:

- 1) una rotación simple a la derecha entre  $k$  e  $i$ , y a continuación,
- 2) una rotación simple a la izquierda entre  $k$  y  $j$ .

## Árboles balanceados: Árbol AVL



## Árboles balanceados: Árbol AVL

Como **resultado de reequilibrar el subárbol** cuya raíz es el primer nodo que incumple la condición de equilibrio, en el camino desde el nodo insertado a la raíz, este subárbol:

- ▷ cumple la condición de equilibrio en sus subárboles izquierdo y derecho, y
- ▷ además, **su altura es la misma que tenía antes de la inserción.**

Por tanto, **todas las roturas de equilibrio en el resto del camino hasta la raíz quedan automáticamente restauradas.**

- ▷ La causa de su rotura de equilibrio estaba en el incremento de altura en este subárbol producido por la inserción del nuevo nodo.

## Árboles balanceados: Árbol AVL

### Implementación de árboles AVL

Se debe guardar la altura de cada subárbol como atributo del nodo raíz del mismo.

- ▷ Calcularla cada vez que se necesite destruye el coste logarítmico de las operaciones.
- ▷ En la práctica, con 1 byte (`char`) es suficiente. Un árbol AVL con el mínimo número de nodos para una altura 127 tendrá más de  $4 \cdot 10^{26}$  nodos.
- ▷ Por otra parte, no es preciso almacenar exactamente la altura.

Basta con almacenar el **factor de equilibrio**: +1, 0 ó -1, para representar la diferencia de altura entre el subárbol izquierdo y el derecho, que cabe en 2 bits.

```
struct Nodo {
    TBAVL elem;    Nodo * hizq, * hder;    char altura;
    explicit Nodo(const TBAVL & dato, Nodo * hi=0, Nodo * hd=0, char h=0)
        { elem = dato; hizq = hi; hder = hd; altura = h; }
};
typedef Nodo * Enlace;
```

## Árboles balanceados: Árbol AVL

### Implementación de la inserción

Se puede partir de la función recursiva dada para insertar en un árbol binario de búsqueda para implementar la inserción en un árbol AVL.

- ▷ No obstante, será más eficiente una inserción iterativa optimizada.

El siguiente código ilustra las acciones a realizar en cada nodo del camino de vuelta hacia la raíz desde el nuevo nodo insertado con el elemento `item`, tras insertarlo.

- ▷ Comprueba el equilibrio del nodo y, si está equilibrado, actualiza su altura, y si no, determina el caso aplicable y lo reequilibra.
- ▷ En este momento `t->hizq` y `t->hder`, si existen, habrán actualizado sus alturas.

```
1 char altizq = t->hizq ? t->hizq->altura : -1;
2 char altder = t->hder ? t->hder->altura : -1;
3 if (altizq - altder <= 1 && altder - altizq <= 1)
4     t->altura = altizq > altder ? altizq+1 : altder+1;
```

## Árboles balanceados: Árbol AVL

```

5 else if (item < t->elem) {
6   if (item < t->hizq->elem) // CASO 1
7     RotacionSimpleDerecha(t);
8   else { // CASO 2
9     RotacionDobleIzquierdaDerecha(t);
10    t->hizq->altura--; t->altura++;
11  }
12  t->hder->altura--;
13 }
14 else if (t->elem < item) {
15   if (t->hder->elem < item) // CASO 4
16     RotacionSimpleIzquierda(t);
17   else { // CASO 3
18     RotacionDobleDerechaIzquierda(t);
19     t->hder->altura--; t->altura++;
20   }
21   t->hizq->altura--;
22 }

```

El caso se determina averiguando por qué ramas se bifurcó el camino de inserción de item.

Puede hacerse mediante comparaciones con los elementos de t, t->hizq o t->hder.

También puede averiguarse con los valores del factor de equilibrio:

- (1) +2 en t y +1 en t->hizq
- (2) +2 en t y -1 en t->hizq
- (3) -2 en t y +1 en t->hder
- (4) -2 en t y -1 en t->hder

## Árboles balanceados: Árbol AVL

```

23 void RotacionSimpleDerecha(Enlace & pi) {
24   Enlace pj = pi->hizq;
25   pi->hizq = pj->hder;
26   pj->hder = pi;
27   pi = pj;
28 }

```



```

29 void RotacionSimpleIzquierda(Enlace & pj) {
30   Enlace pi = pj->hder;
31   pj->hder = pi->hizq;
32   pi->hizq = pj;
33   pj = pi;
34 }

```

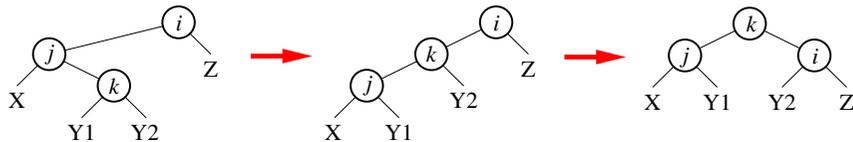


## Árboles balanceados: Árbol AVL

```

35 void RotacionDobleIzquierdaDerecha(Enlace & pi) {
36   RotacionSimpleIzquierda(pi->hizq);
37   RotacionSimpleDerecha(pi);
38 }

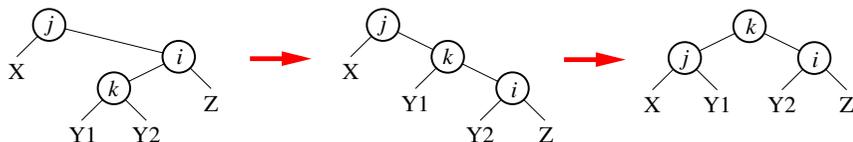
```



```

39 void RotacionDobleDerechaIzquierda(Enlace & pj) {
40   RotacionSimpleDerecha(pj->hder);
41   RotacionSimpleIzquierda(pj);
42 }

```



## Árboles balanceados: Árbol AVL

### Eliminación

Al igual que ocurre en la inserción, tras eliminar un nodo de la forma estándar:

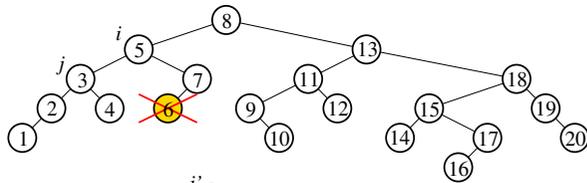
- ▷ Sólo los nodos del camino desde el nodo eliminado a la raíz pueden haber perdido su equilibrio (ya que sólo estos han modificado sus subárboles).
- ▷ Se recorre de abajo a arriba dicho camino comprobando la condición de equilibrio en cada nodo.

Pero, a diferencia de la inserción:

- ▷ **puede ser necesario reequilibrar más de un subárbol** en este camino de vuelta, ya que **tras reequilibrar no se conserva la altura** (previa a la eliminación) del subárbol reequilibrado.

# Árboles balanceados: Árbol AVL

rotación simple  
a la derecha



rotación simple  
a la izquierda

