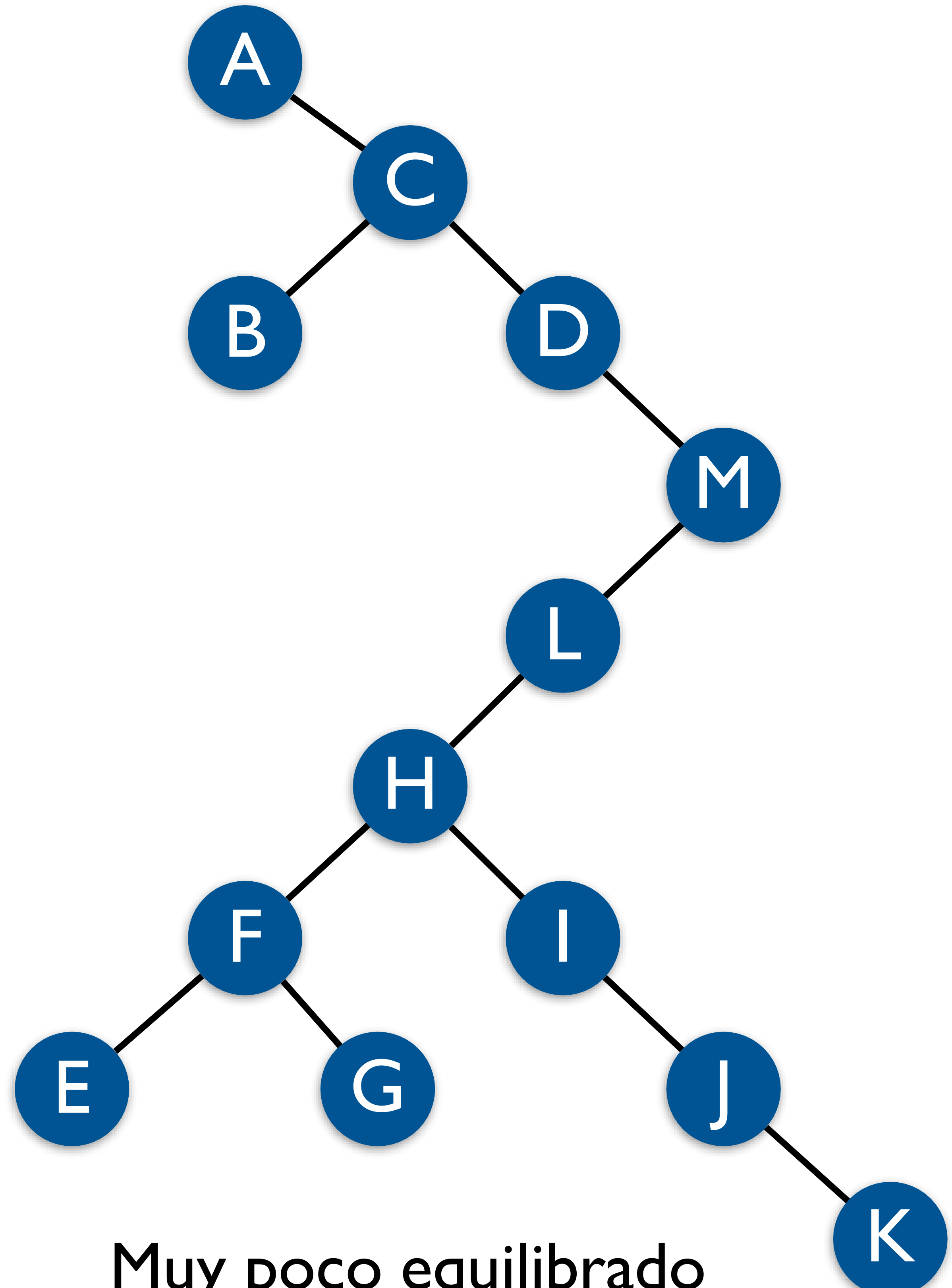
Árboles binarios equilibrados
Árboles AVL

Joaquín Fernández-Valdivia
Javier Abad
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

DECSAI

Motivación

- En ocasiones, la construcción de los ABB conduce a árboles con características muy pobres para la búsqueda



Muy poco equilibrado

IDEA

Construir ABB equilibrados, impidiendo que en ningún nodo las alturas de los subárboles izquierdo y derecho difieran en más de una unidad

Árboles AVL

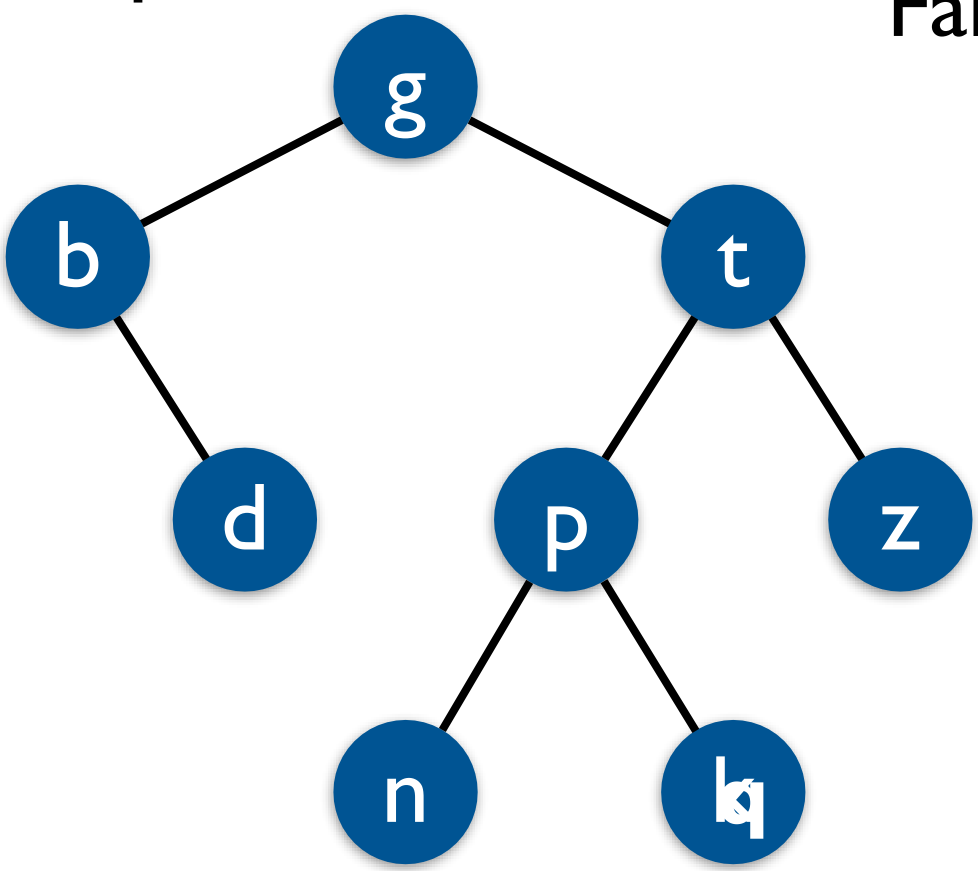
- Diremos que un árbol binario de búsqueda es un AVL (o que está equilibrado en el sentido de Addelson-Velski-Landis) si, para cada uno de sus nodos, se cumple que las alturas de sus dos subárboles difieren como máximo en 1

- Es decir, si T_i tiene altura h, T_d puede tener como máximo altura h+1 y viceversa

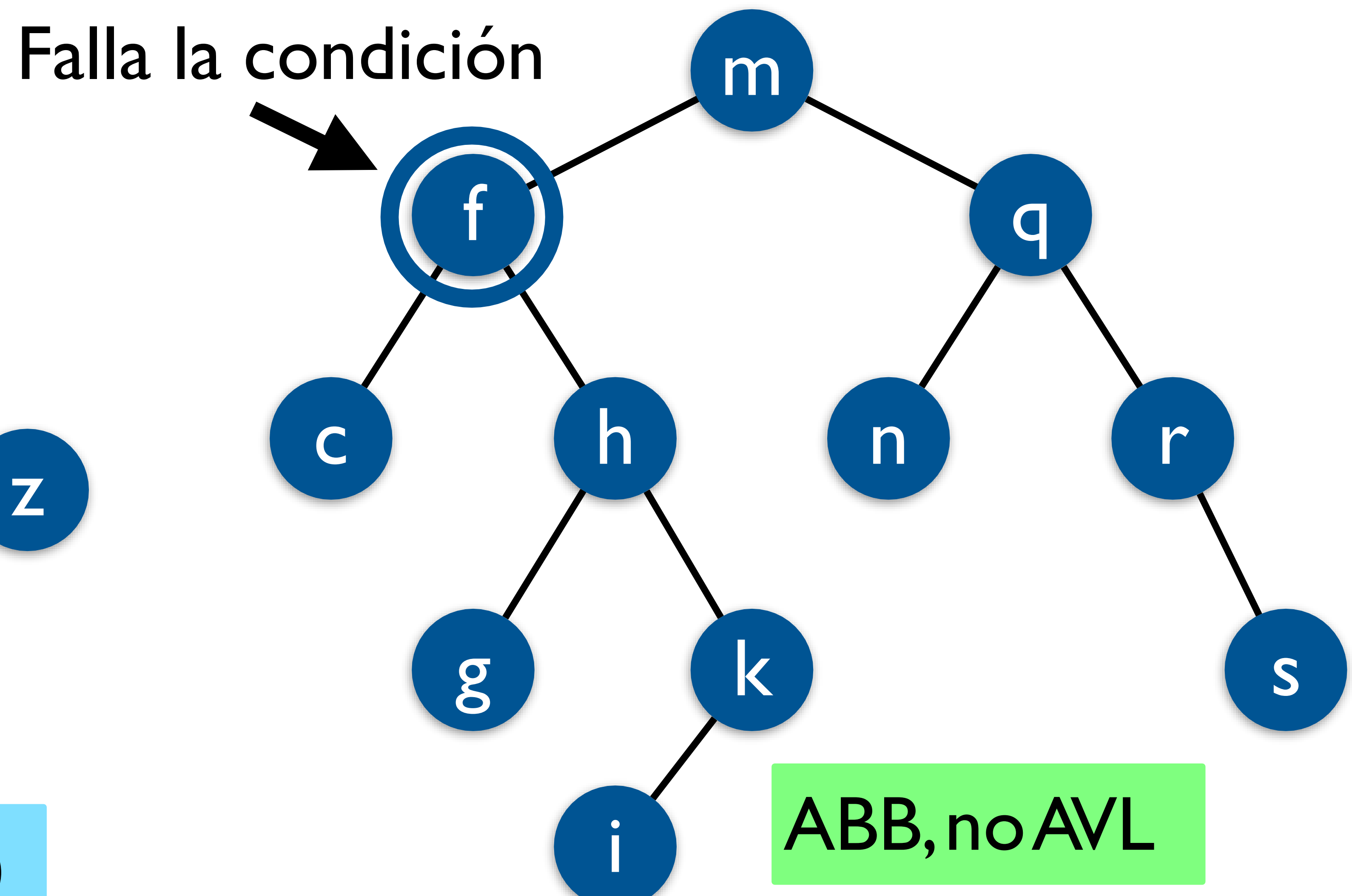
- Los árboles que cumplen esta condición son denominados como árboles AVL

Árboles AVL

- Los AVL cumplen dos condiciones en cada nodo:
  - Analítica: la misma que los ABB
  - Geométrica: las alturas de sus dos subárboles difieren como mucho en 1
- Ejemplo:



AVL (ABB + Equilibrio)



ABB, no AVL

Eficiencia

- La altura de un árbol AVL está acotada por

$$\log_2(n+1) \leq h \leq 1.44 \log_2(n+2) - 0.33$$

- La altura de un AVL (esto es, la longitud de sus caminos de búsqueda) con *n* nodos nunca excede al 44% de la longitud de los caminos (o la altura) de un árbol completamente equilibrado con *n* nodos

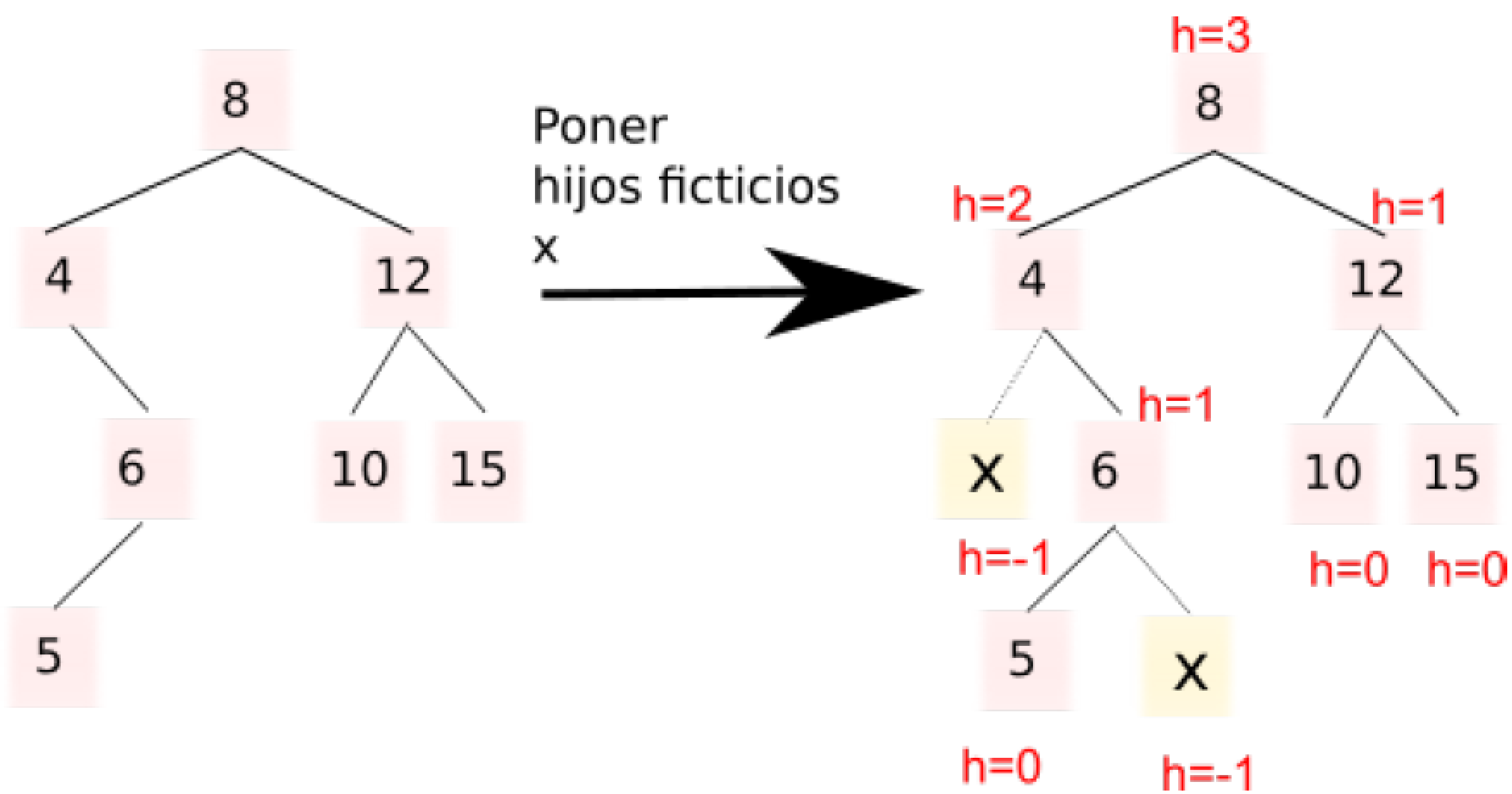
- Consecuencia: en el peor de los casos, la búsqueda se puede realizar en $O(\log_2 n)$

Eficiencia

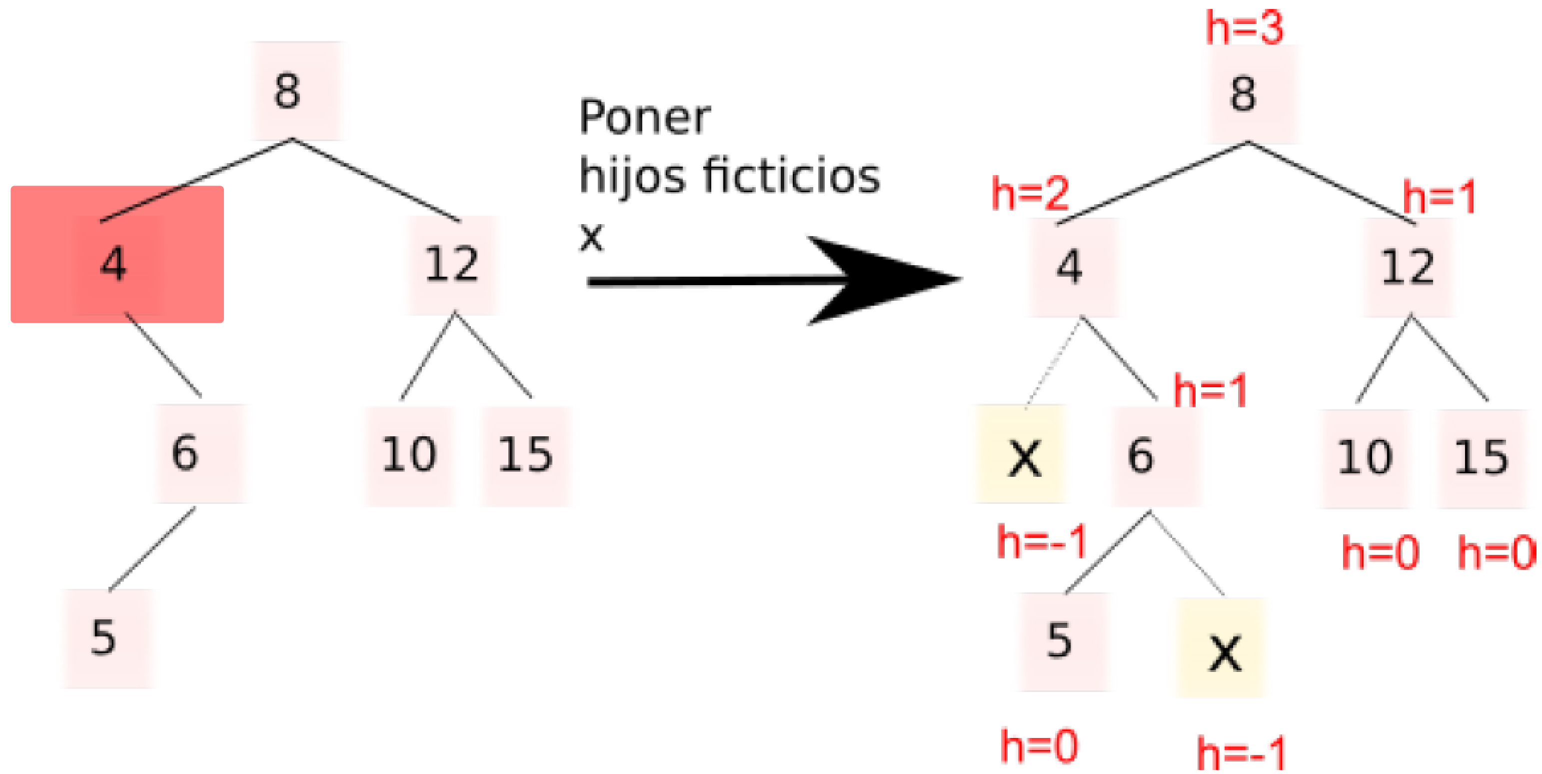
- Si un ABB está muy desequilibrado, los tiempos de búsqueda **no** son $\log_2(n)$
- En el peor de los casos podría ser O(n)
- Lo ideal sería tener en cada nodo aproximadamente el mismo número de nodos para, en cada iteración, descartar la mitad (o casi) de nodos y tener un tiempo de búsqueda de $\log_2(n)$

Ejemplo

- Dado el árbol a la izquierda vamos a obtener su altura
- Antes vamos a transformarlo en el árbol de la derecha
- Este árbol se obtiene añadiendo al árbol de la izquierda el hijo que le falta cuando un nodo tiene un sólo hijo
- A este hijo ficticio le hemos puesto etiqueta x

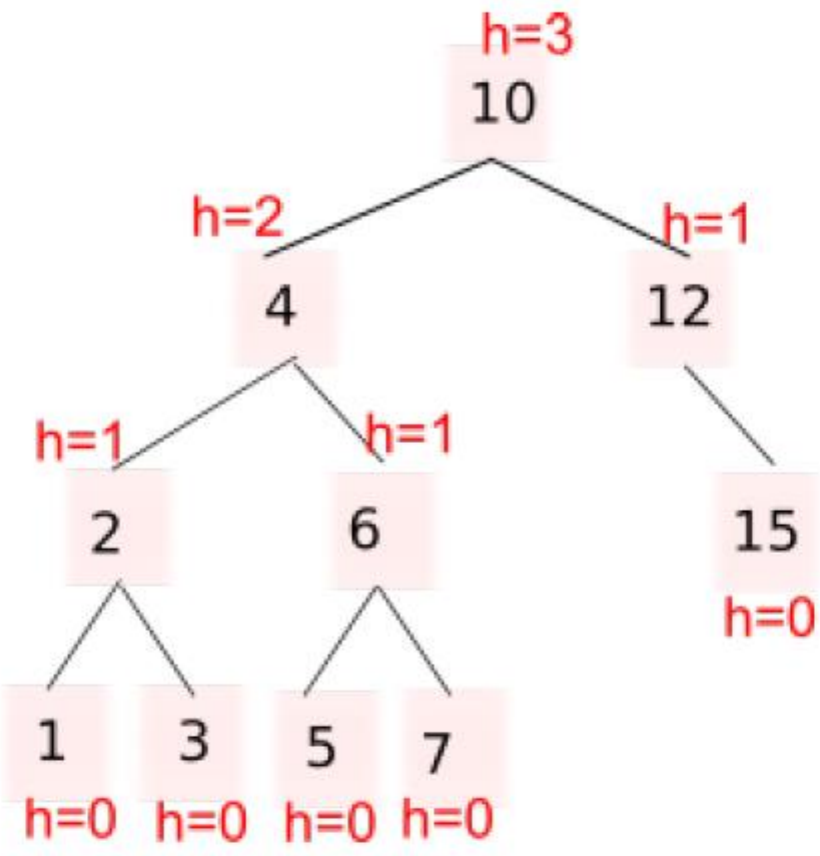


Ejemplo



1. Los nodos que no existen (x) tienen altura h = -1
2. Las hoja tienen altura h = 0
3. Por ejemplo, el nodo 6 tiene altura 1, ya que sería la altura máxima de sus hijos más 1
4. Tenemos un desequilibrio en el 4, ya que sus hijos tienen h(x)=-1 y h(2)=1, la altura de ambos difiere en más de 1, por lo que no es AVL

Ejemplo



- Este árbol sí está equilibrado, aunque no tengamos el mismo número de nodos en T_i y T_d ya que se cumple la definición de AVL

Árboles AVL

- Nos interesan funciones para las operaciones de:
  - Pertenencia
  - Inserción
  - Borrado

- Debemos tener en cuenta que tendremos que diseñar funciones auxiliares que permitan realizar estas operaciones manteniendo el árbol equilibrado

Equilibrio en inserciones y borrados

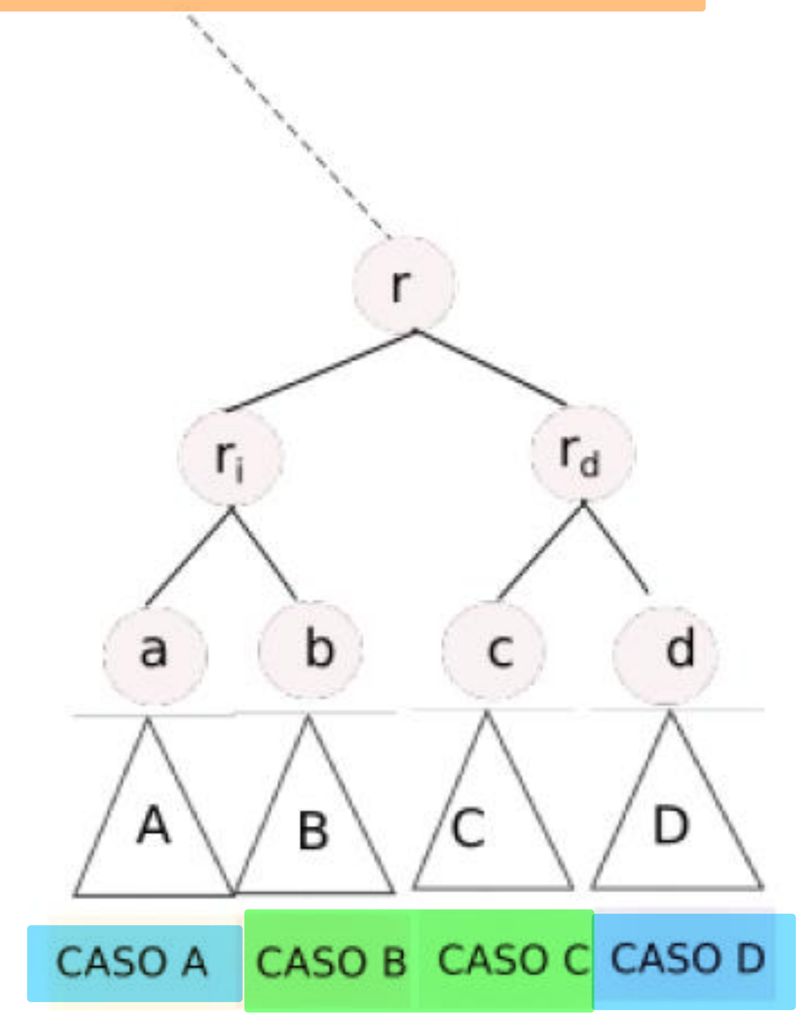
- En el proceso de inserción podemos desequilibrar el árbol, por tanto, debemos volver a hacer que esté equilibrado

- Los pasos a seguir para insertar un elemento en un AVL serían:
  1. Buscar dónde insertar el nuevo elemento
  2. Insertarlo
  3. Equilibrar el árbol

Equilibrio en inserciones y borrados

- El desequilibrio ocurre en el nodo *r*
- Pero puede ocurrir porque se haya realizado la nueva inserción en el subárbol A, B, C o D
- El procedimiento para volver a equilibrar el árbol depende de dónde se haya hecho la nueva inserción

- Para lograr el equilibrio se aplicarán rotaciones simples (ocurren cuando la nueva inserción se ha hecho en el subárbol A o D), o rotaciones dobles que ocurren cuando se hace la nueva inserción en los subárboles B y C



Equilibrio en inserciones y borrados

Idea: Usar un campo altura en el registro que represente cada uno de los nodos del AVL para determinar el factor de equilibrio (diferencia de altura entre los subárboles izquierdo y derecho), de forma que cuando esa diferencia sea >1 se hagan los reajustes necesarios en los punteros para que tenga una diferencia de alturas ≤ 1

Representación

```
template <class T>
struct info_nodo_AVL {
      T et;
      info_nodo_AVL<T> * hijoizq, * hijoder, * padre;
      int altura;
};

// El proceso de busqueda es identico al ABB normal.
```

- Cada nodo del árbol almacena su altura
- Al realizar una inserción en el árbol la altura del nodo puede verse afectada

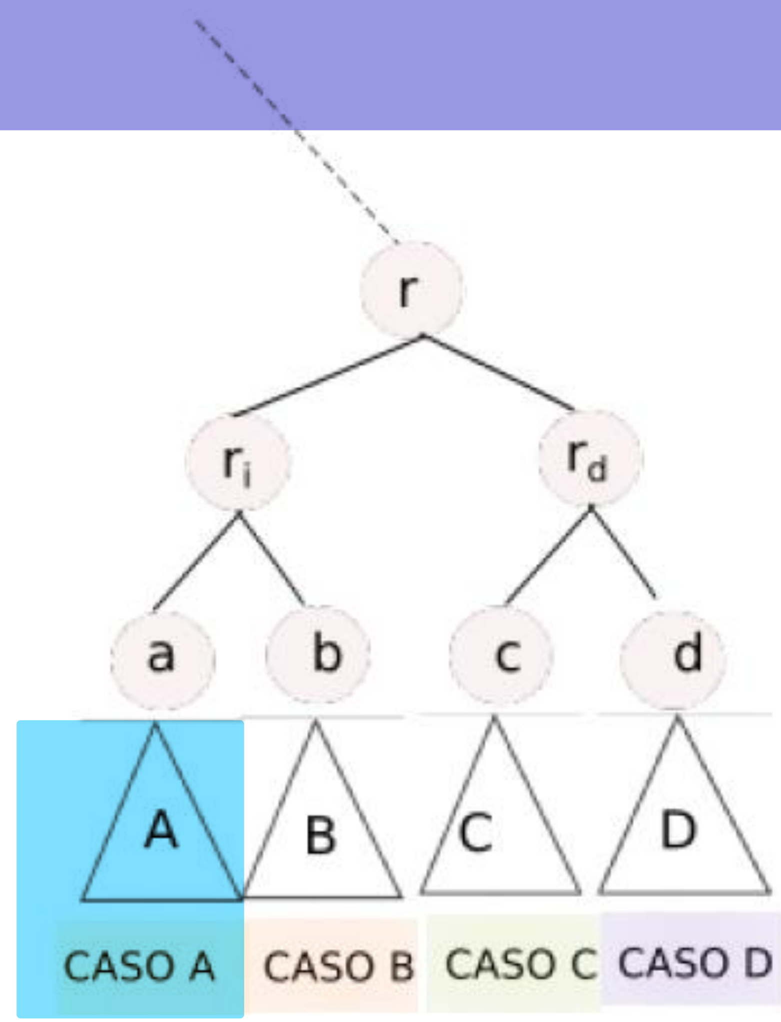
Equilibrio en inserciones y borrados

- Notaremos los subárboles como T_k, anotando entre paréntesis su altura (la altura de su raíz)
- Notaremos el factor de equilibrio como un valor con signo ubicado entre corchetes junto a cada padre o hijo
- Las dos situaciones posibles que pueden representarse son:
  - Rotaciones simples
  - Rotaciones dobles

Rotaciones Simples

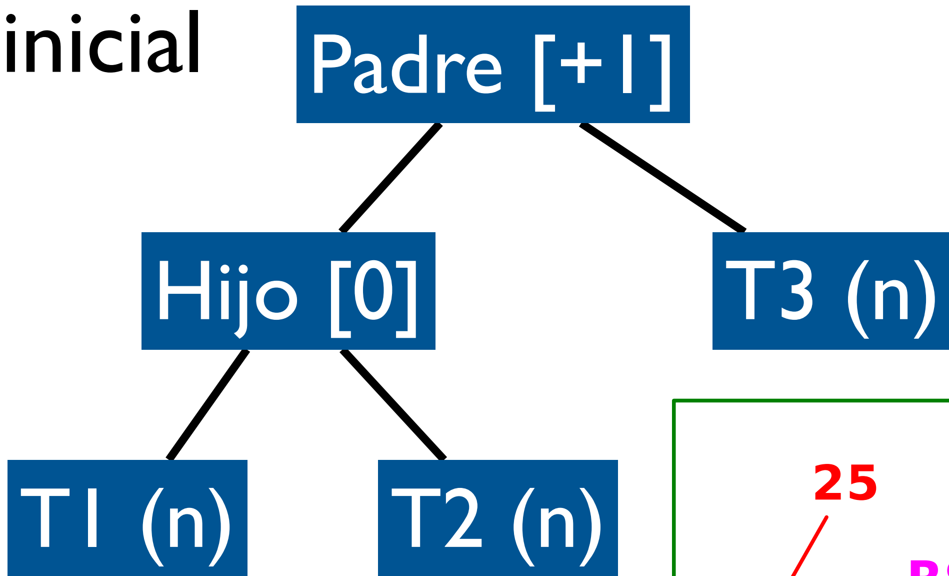
CASO A

- El desequilibrio se produce al insertar un nuevo elemento en la parte más a la izquierda del árbol (subárbol A)
- Para equilibrarlo de nuevo, se hace una *rotación simple a la derecha*

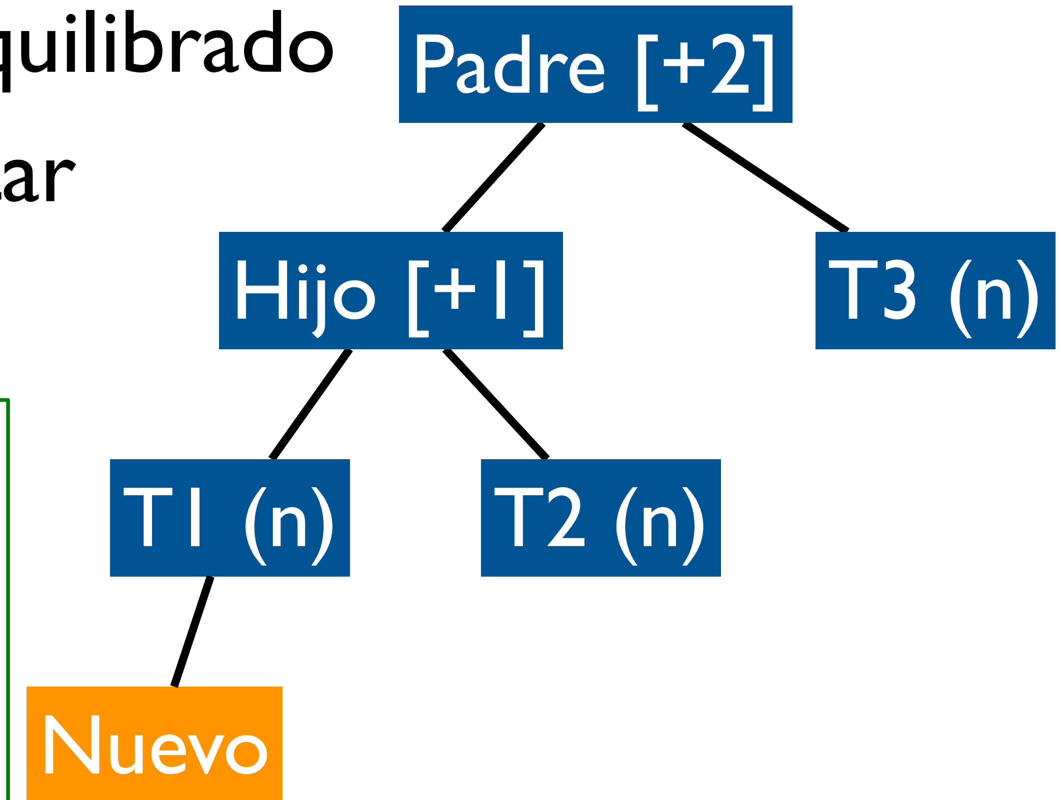


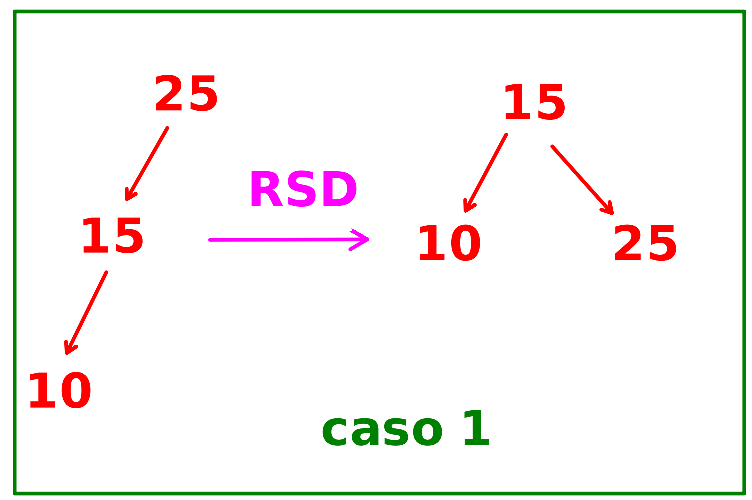
Rotación simple a la derecha

AVL inicial

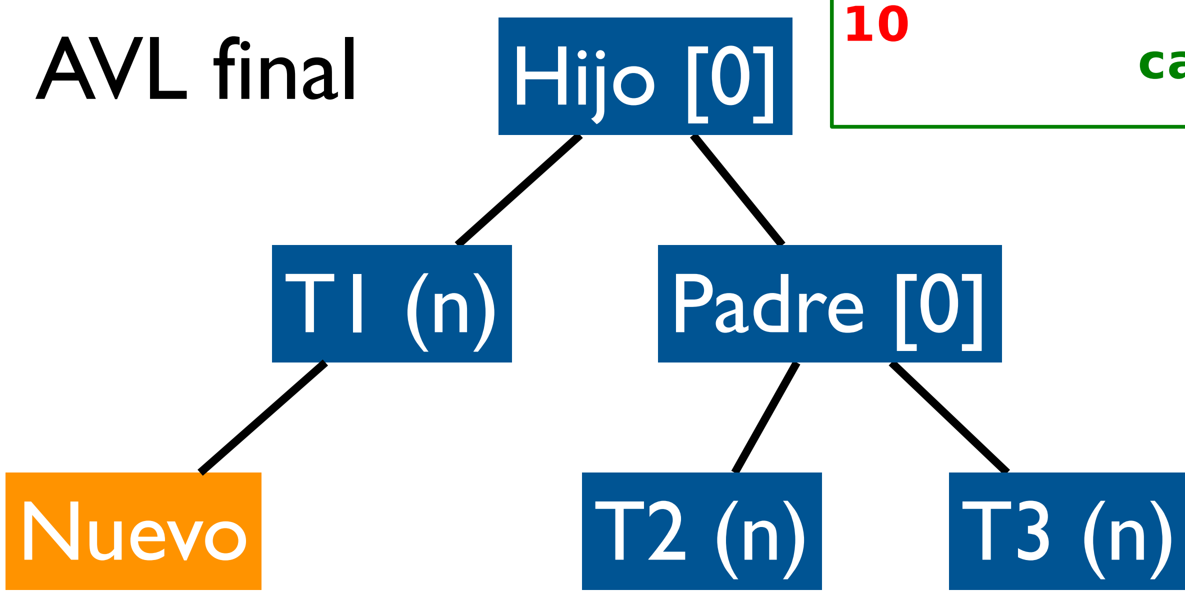


AVL desequilibrado tras insertar





AVL final



Reajustes necesarios para la rotación

padre→izqda = hijo→drcha

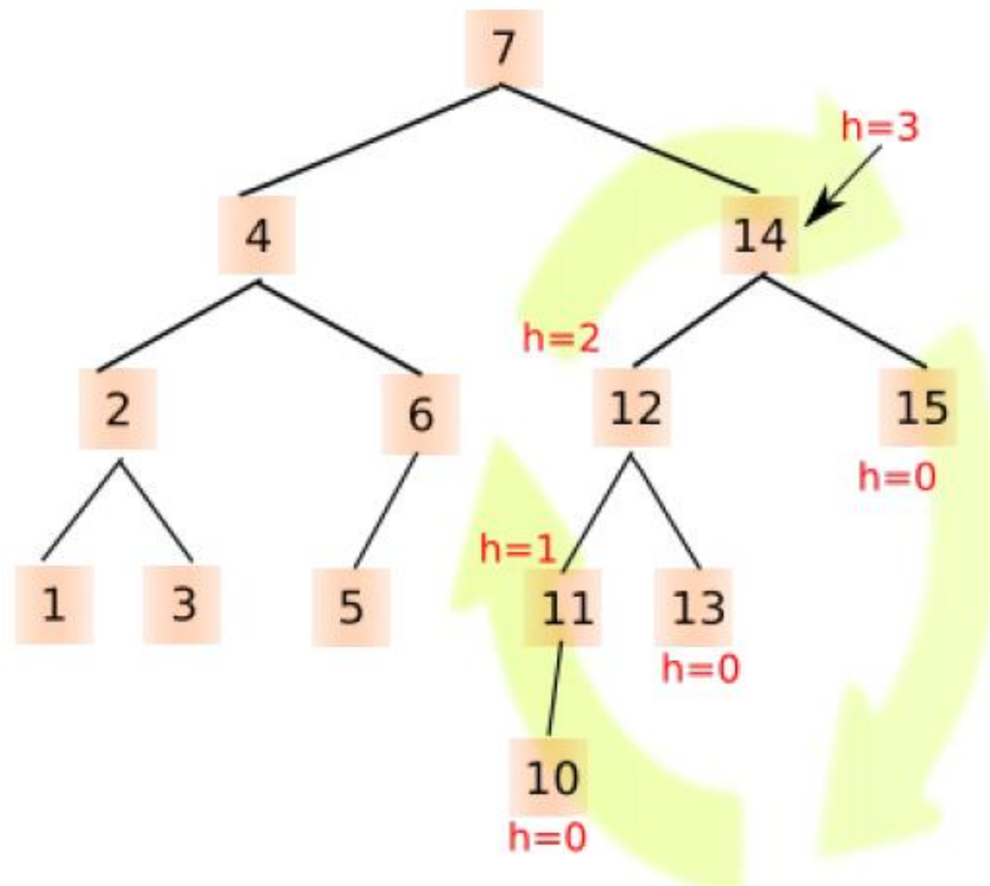
hijo→drcha = padre

a) Se preserva el inorden

b) Altura del árbol final = altura arbol inicial

Rotaciones Simples

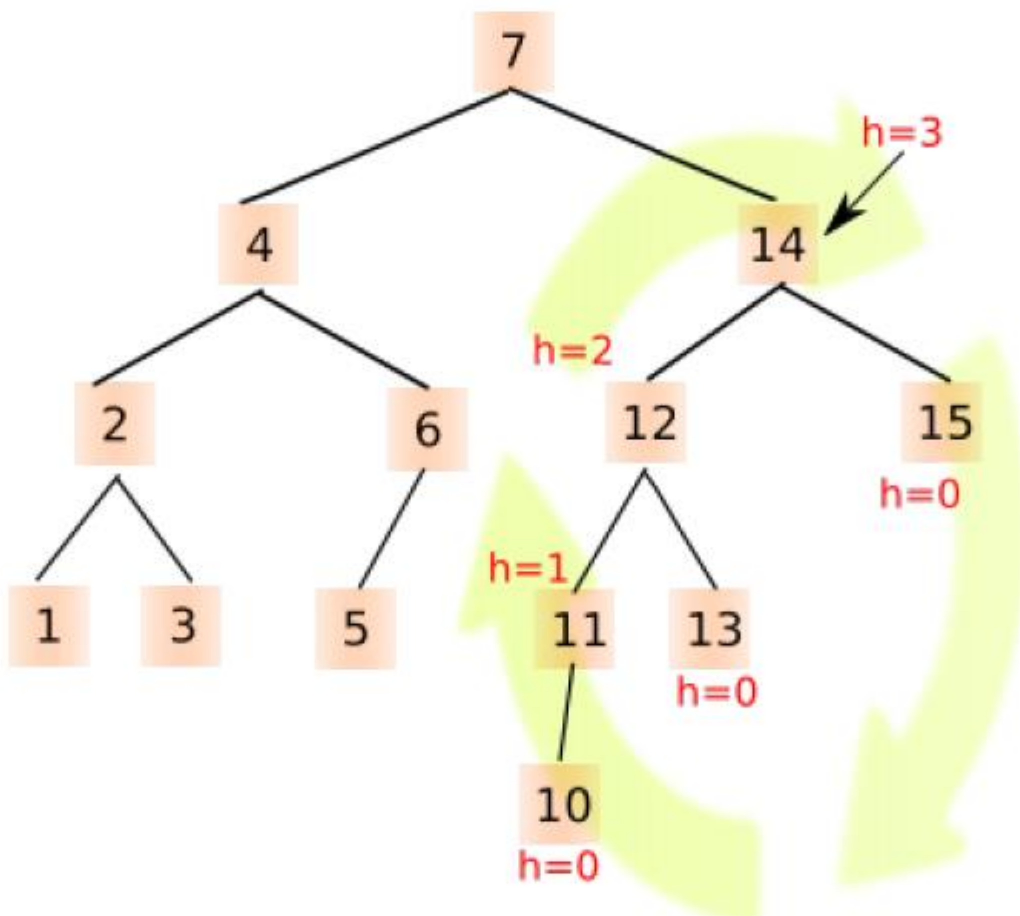
CASO A

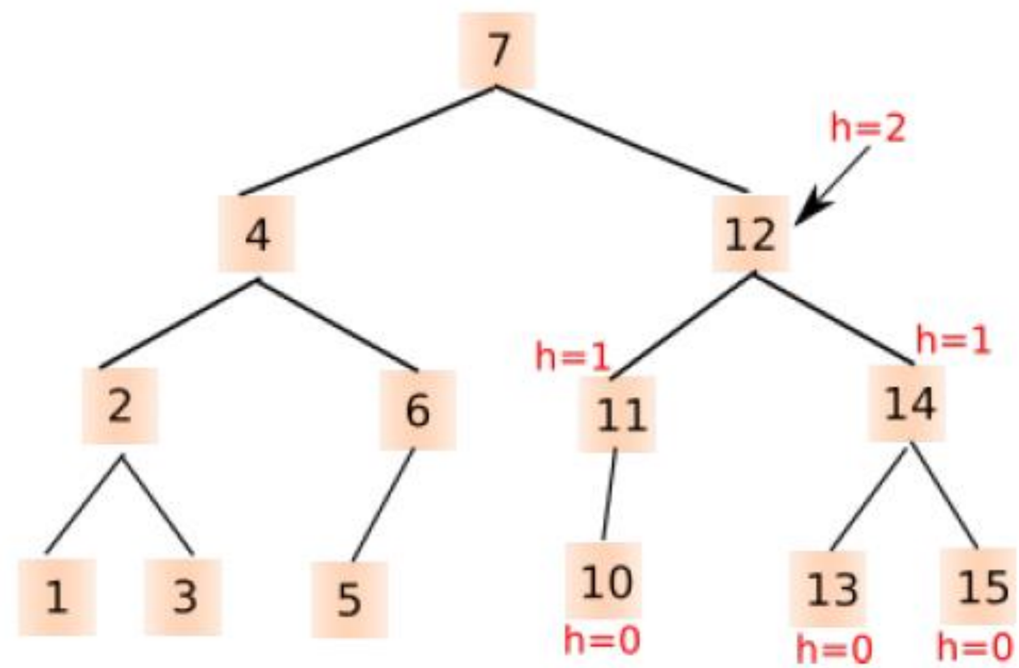


- Supongamos que el árbol estaba equilibrado y se inserta la clave 10 dando lugar al árbol de la figura
- En este caso, el desequilibrio está en 14, pues la altura de 12 es h = 2 y la de 15, h = 0
- Al hacer la rotación simple a la derecha, el árbol queda ya equilibrado

Rotaciones Simples

CASO A





Rotaciones Simples

CASO A

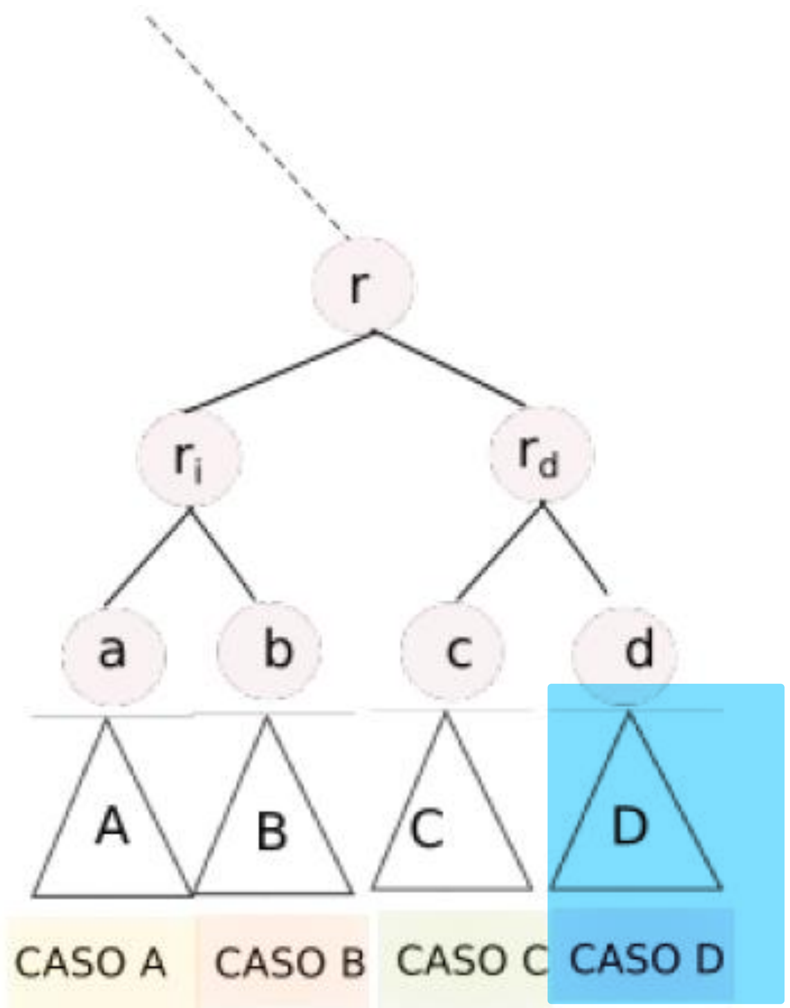
```
template <class T>
void SimpleDerecha (info_nodo_AVL<T> * & n) { // n = 14 en el ejemplo
        info_nodo_AVL<T> * aux = n->hijoizq; // 12 en el ejemplo
        info_nodo_AVL<T> * padre = n->padre; // 7
        // a 14 le ponemos como hijo izquierdo 13
        n->hijoizq = aux->hijoder;
        if (n->hijoizq != 0)
          // el padre de 13 pasa a ser 14
            n->hijoizq->padre = n;
        n->padre = aux;//el padre de 14 pasa a ser 12
        aux->padre = padre;// el padre de 12 es 7
        aux->hder = n;// 12 tiene como hijo derecho a 14
        n = aux;
        ActualizarAltura(n->hder);
}

// Esta funcion Actualiza el campo n->altura
template <class T>
void ActualizarAltura (info_nodo_AVL<T> * & n) {
    if (n != 0) {
        n->altura = std::max(Altura(n->hijoizq), Altura(n->hijoder))+1;
        // La funcion Altura devuelve n->altura si es
        // distinta de 0 y -1 si es 0
        ActualizarAltura(n->padre);
    }
}
```

Rotaciones Simples

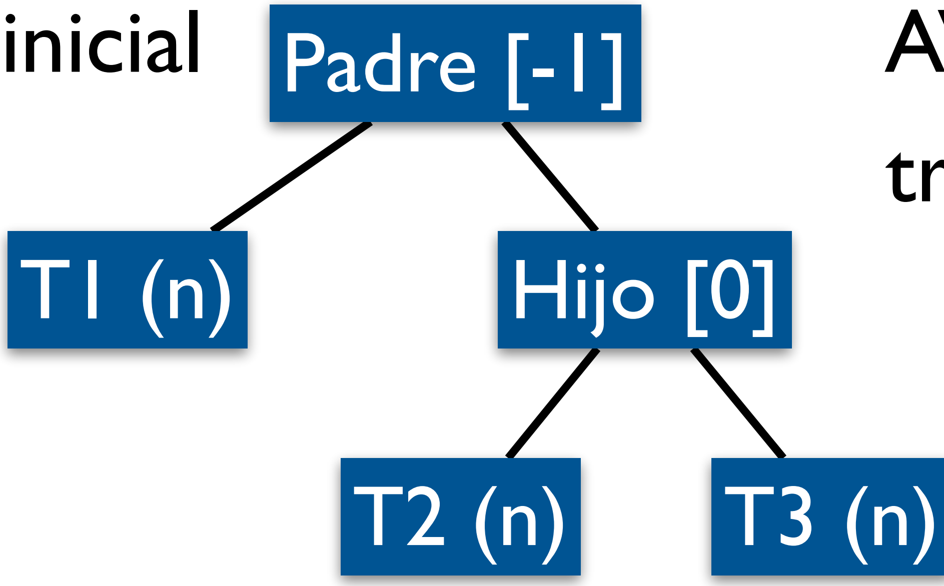
CASO D

- El nodo que produce el desequilibrio se inserta en el subárbol D, para volver a equilibrarlo se hace una *rotación simple a la izquierda*

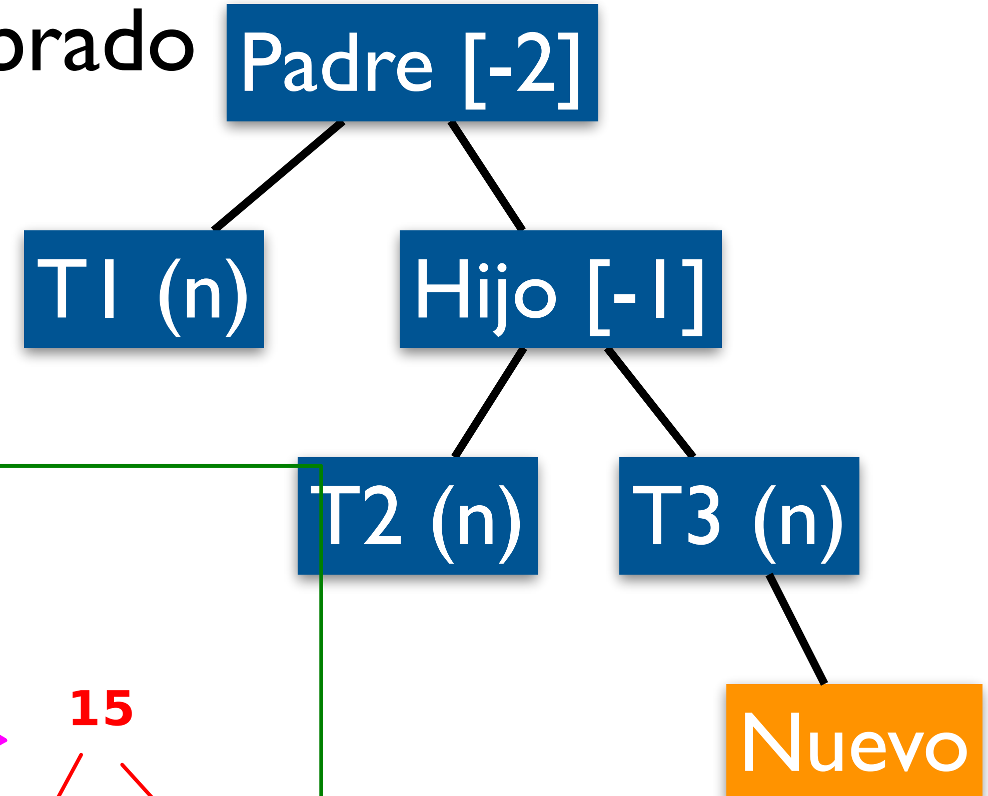


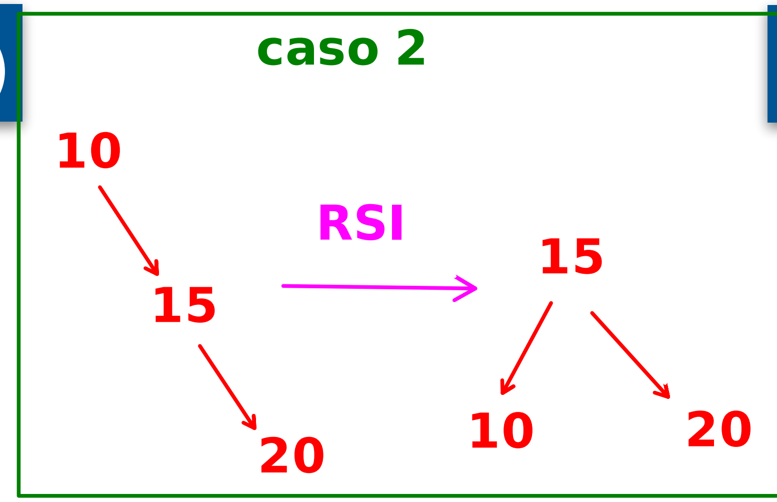
Rotación simple a la izquierda

AVL inicial

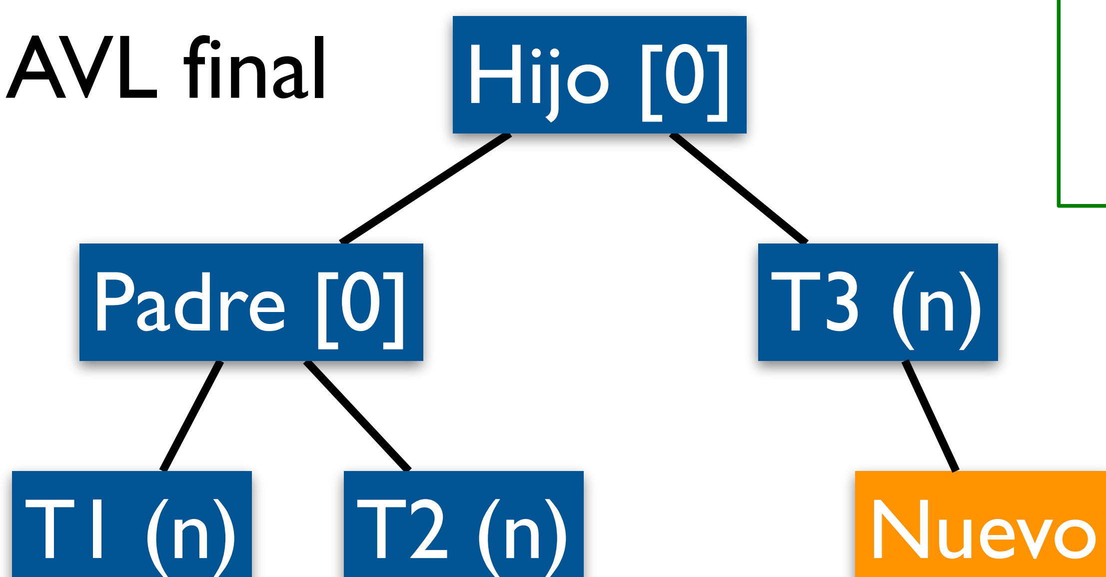


AVL desequilibrado tras insertar





AVL final



Reajustes necesarios para la rotación

padre→drcha = hijo→izqda

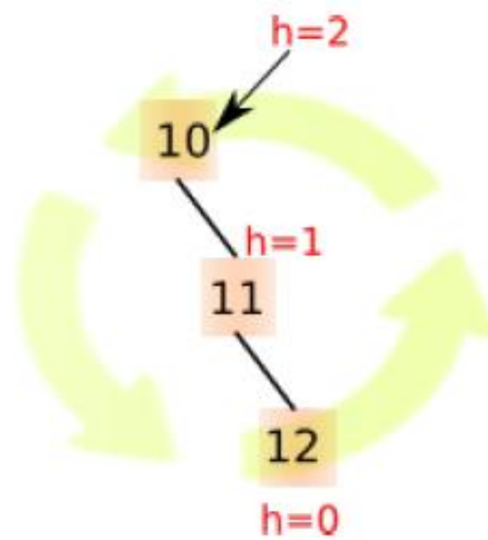
hijo→izqda = padre

a) Se preserva el inorden

b) Altura del árbol final = altura arbol inicial

Rotaciones Simples

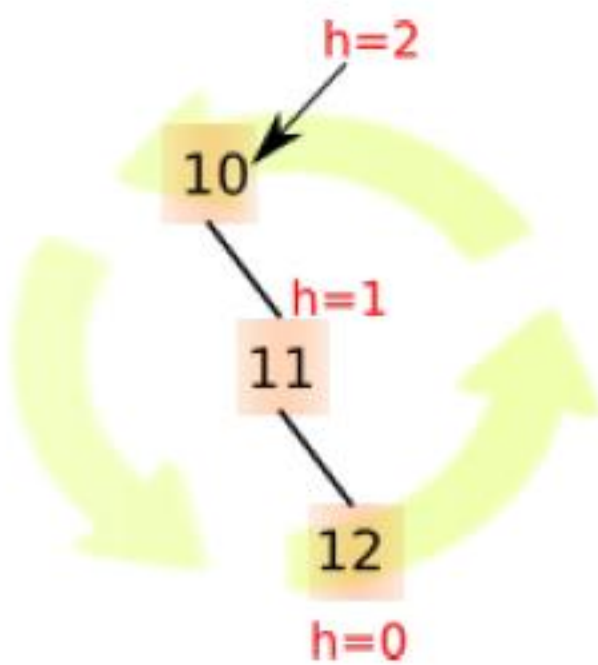
CASO D

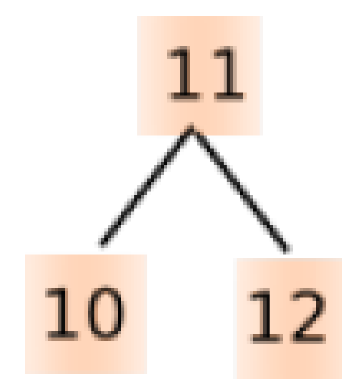


- En este caso, el desequilibrio se encuentra en 10
- Si falta el hijo a la izquierda de 10, creamos un nodo ficticio que tiene altura -1 (hermano de 11) por lo tanto la diferencia es 2 en altura

Rotaciones Simples

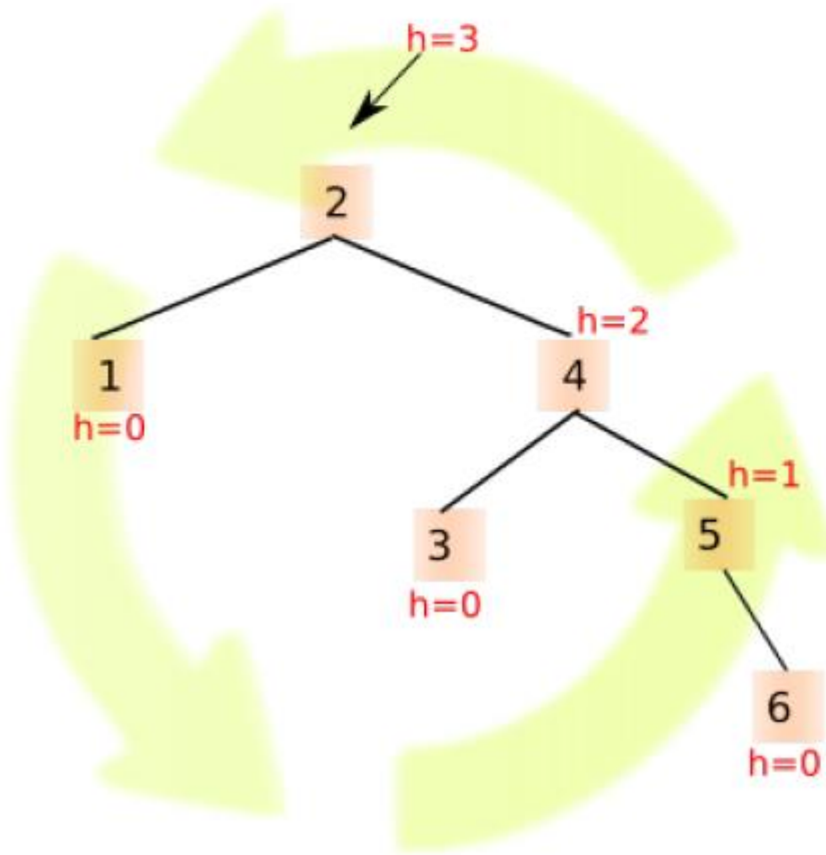
CASO D





Rotaciones Simples

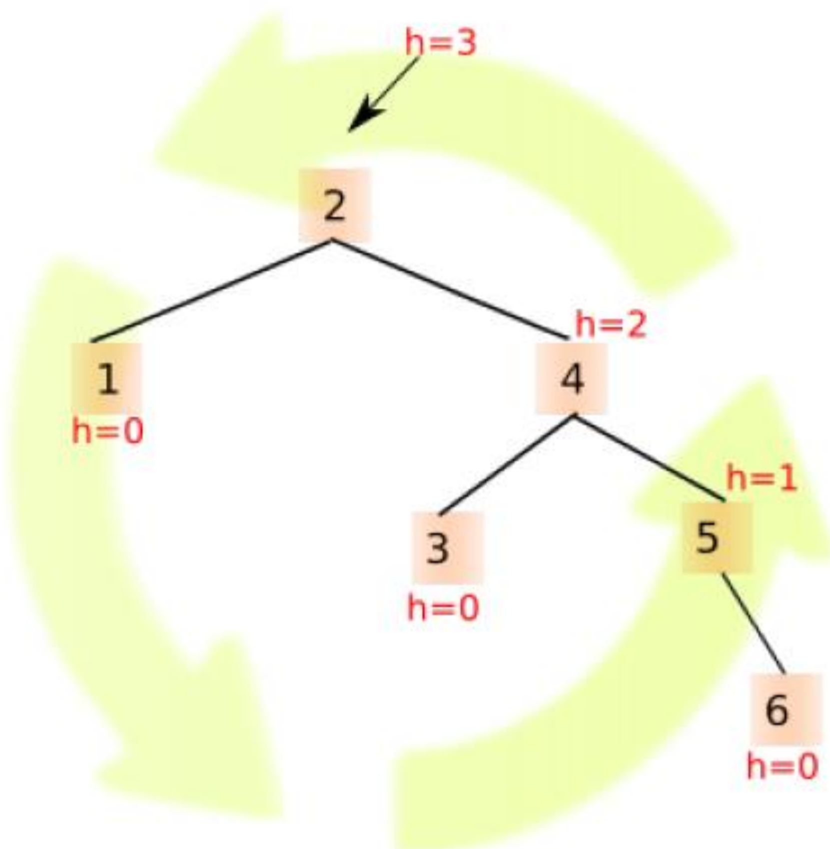
CASO D

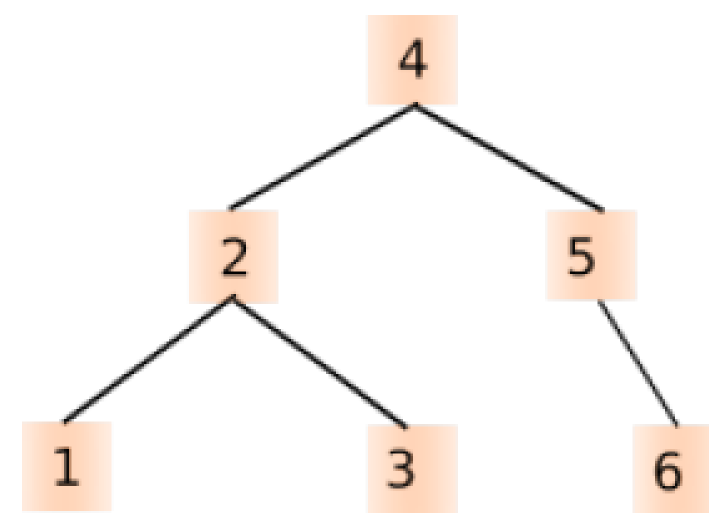


- En este caso el desequilibrio estaría en el 2, ya que su hijo derecho tiene altura h = 2 y el izquierdo, h = 0

Rotaciones Simples

CASO D





Rotaciones Simples

CASO D

```
template <class T>
void SimpleIzquierda (info_nodo_AVL<T> * & n) { // n = 2
        info_nodo_AVL<T> * aux = n->hijoder;        // 4 en el ejemplo
        info_nodo_AVL<T> * padre = n->padre;        // nulo
        n->hder = aux->hijoizq; // a 2 se le pone a 3 como hijo derecho
        if (n->hder!=0)
          n->hder->padre = n;// el padre de 3 pasa a ser 2

        n->padre = aux;     // el padre de 2 es 4
        aux->padre = padre; // el padre de 4 es nulo, porque es la raiz
        aux->hijoizq = n;  // el hijo izquierdo de 4 es 2
        n = aux;
        ActualizarAltura(n->hijoizq);
}
```

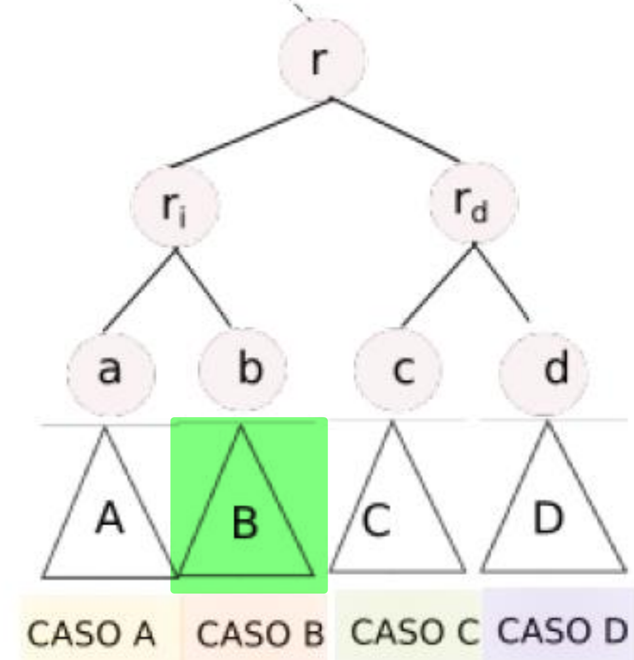
Rotaciones Dobles

CASO B

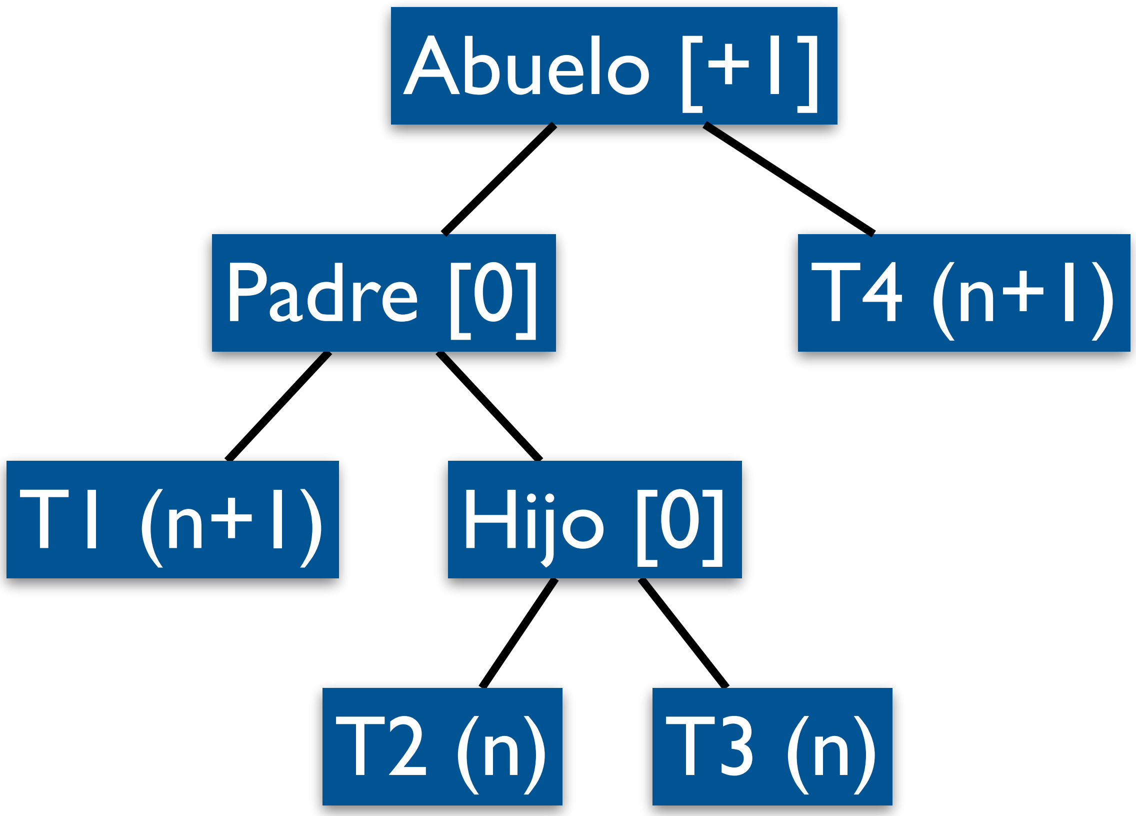
El desequilibrio se produce al insertar un nodo en el subárbol B

Para equilibrar el árbol debemos seguir dos pasos:

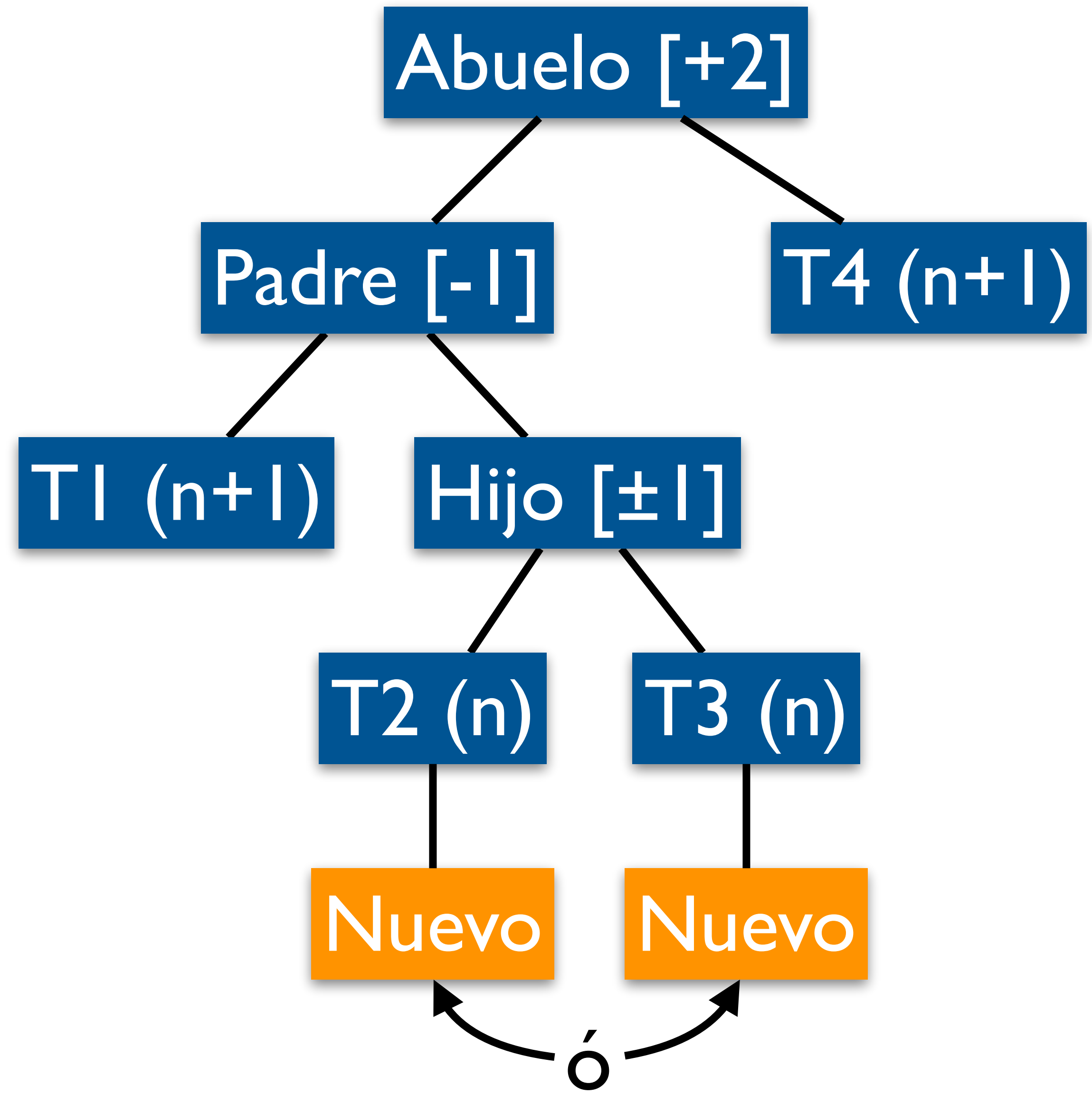
1. Hacer una rotación simple a la izquierda sobre el hijo izquierdo del nodo donde se produzca el desequilibrio
2. Hacer una rotación simple a la derecha sobre el nodo donde surge el desequilibrio



Rotación doble a la derecha

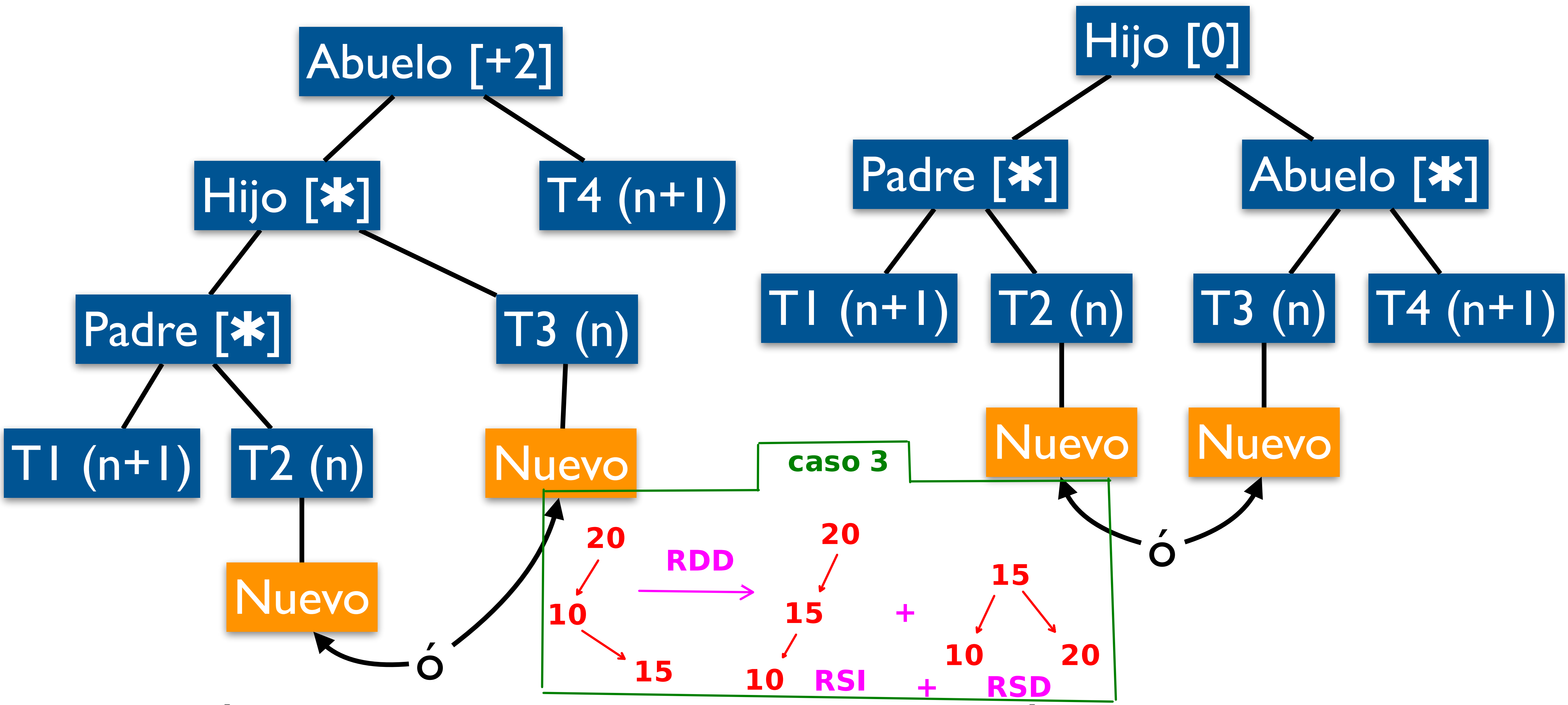


AVL inicial



AVL desequilibrado tras insertar

Rotación doble a la derecha

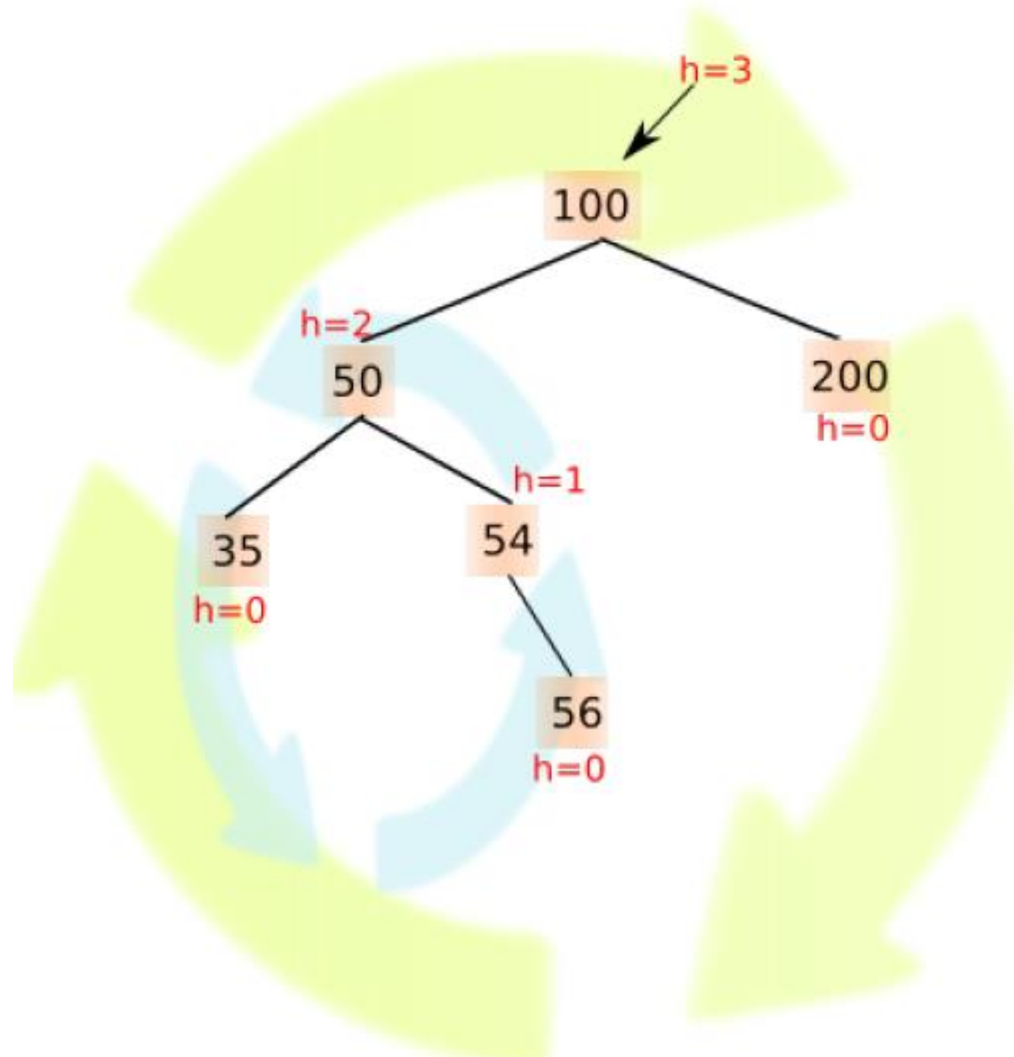


Rotación simple a izquierda
en padre

Rotación simple a derecha
en abuelo

Rotaciones Dobles

CASO B

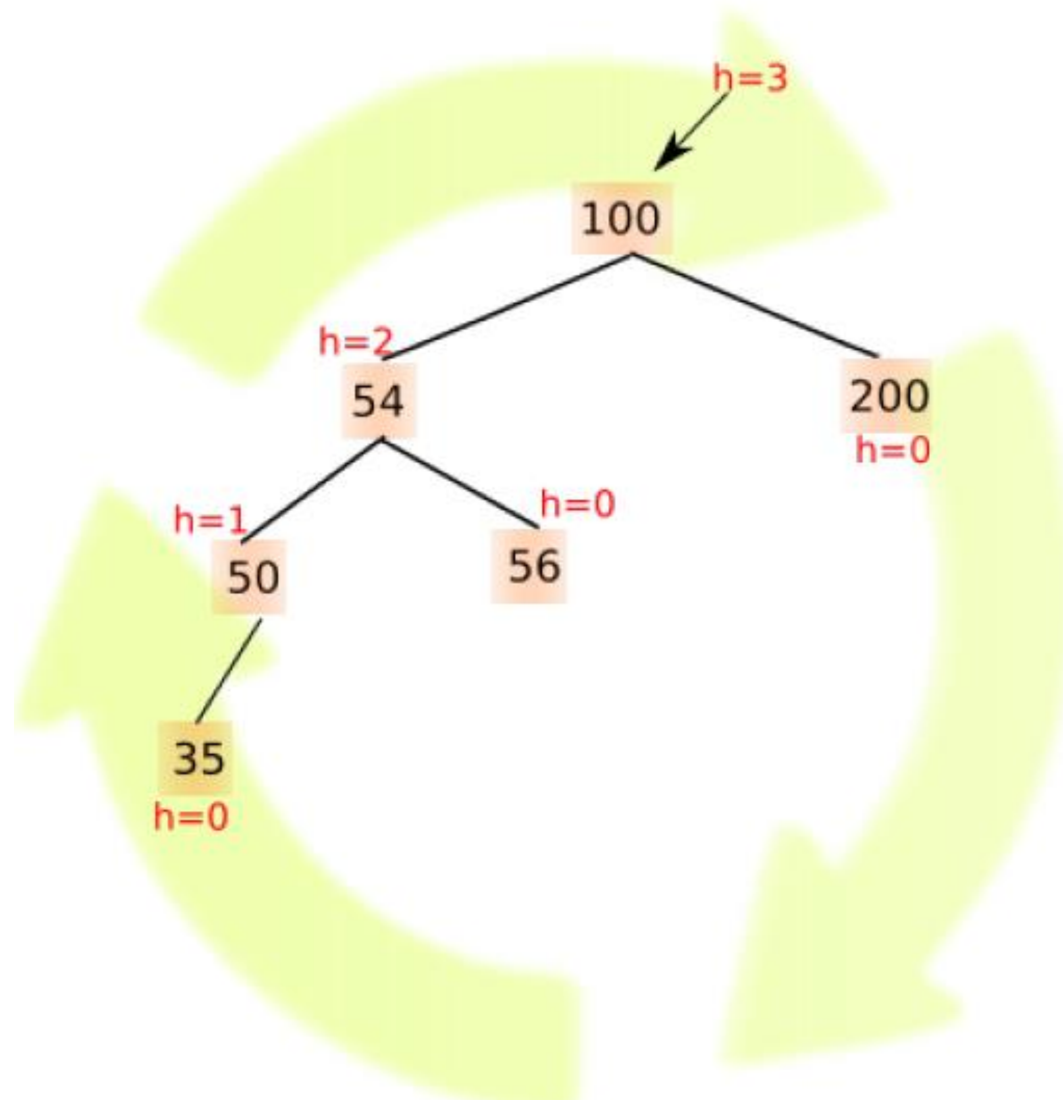


- El desequilibrio se da en el nodo con etiqueta 100, en este caso, para equilibrarlo debemos dar dos pasos

Rotaciones Dobles

CASO B

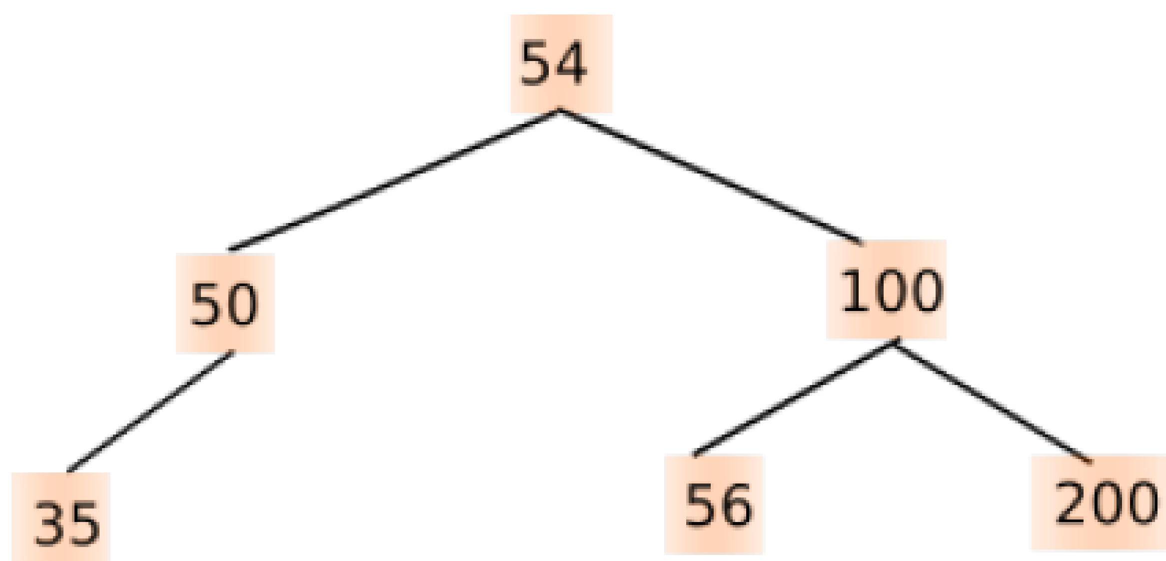
1. En primer lugar debemos hacer una rotación simple a la izquierda en el nodo de etiqueta 50:



Rotaciones Dobles

CASO B

2. Pero, el árbol aún no está equilibrado, falta el último paso que sería hacer una rotación simple a la derecha sobre 100:



Rotaciones Dobles

CASO B

La rotación doble consistiría, por tanto, en llamar a las funciones de rotación simples pasando como argumento los nodos correspondientes:

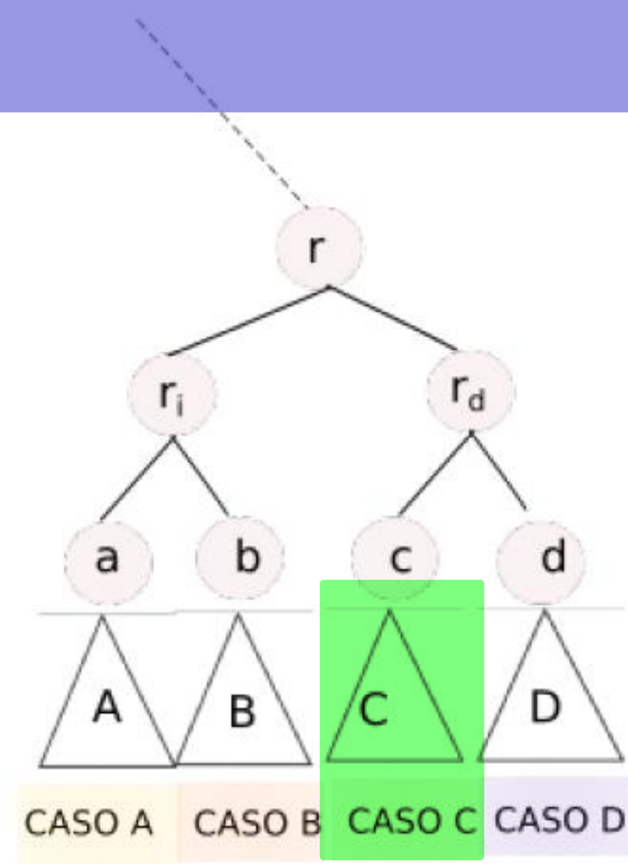
```
template <class T>
void Doble_IzquierdaDerecha (info_nodo_AVL<T> * & n) {
        SimpleIzquierda (n->hijoizq);
        SimpleDerecha(n);
}
```

Rotaciones Dobles

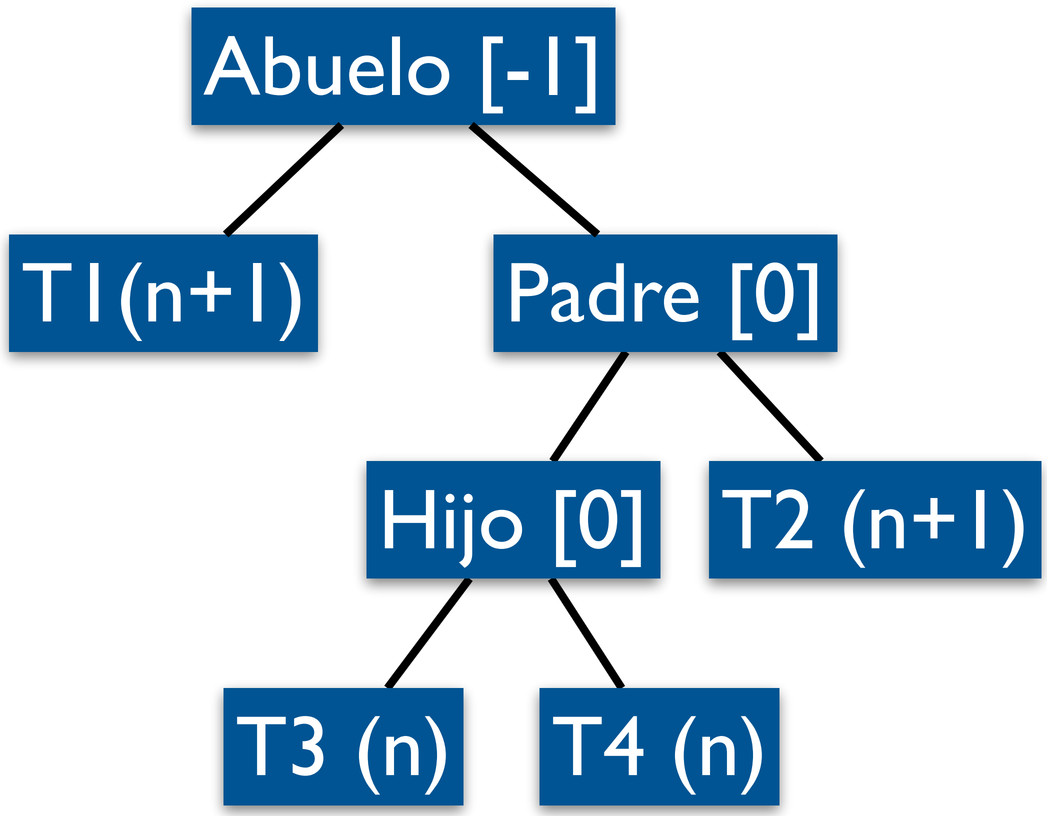
CASO C

Es equivalente pero hay que hacerlo al contrario, es decir, los pasos a seguir serían:

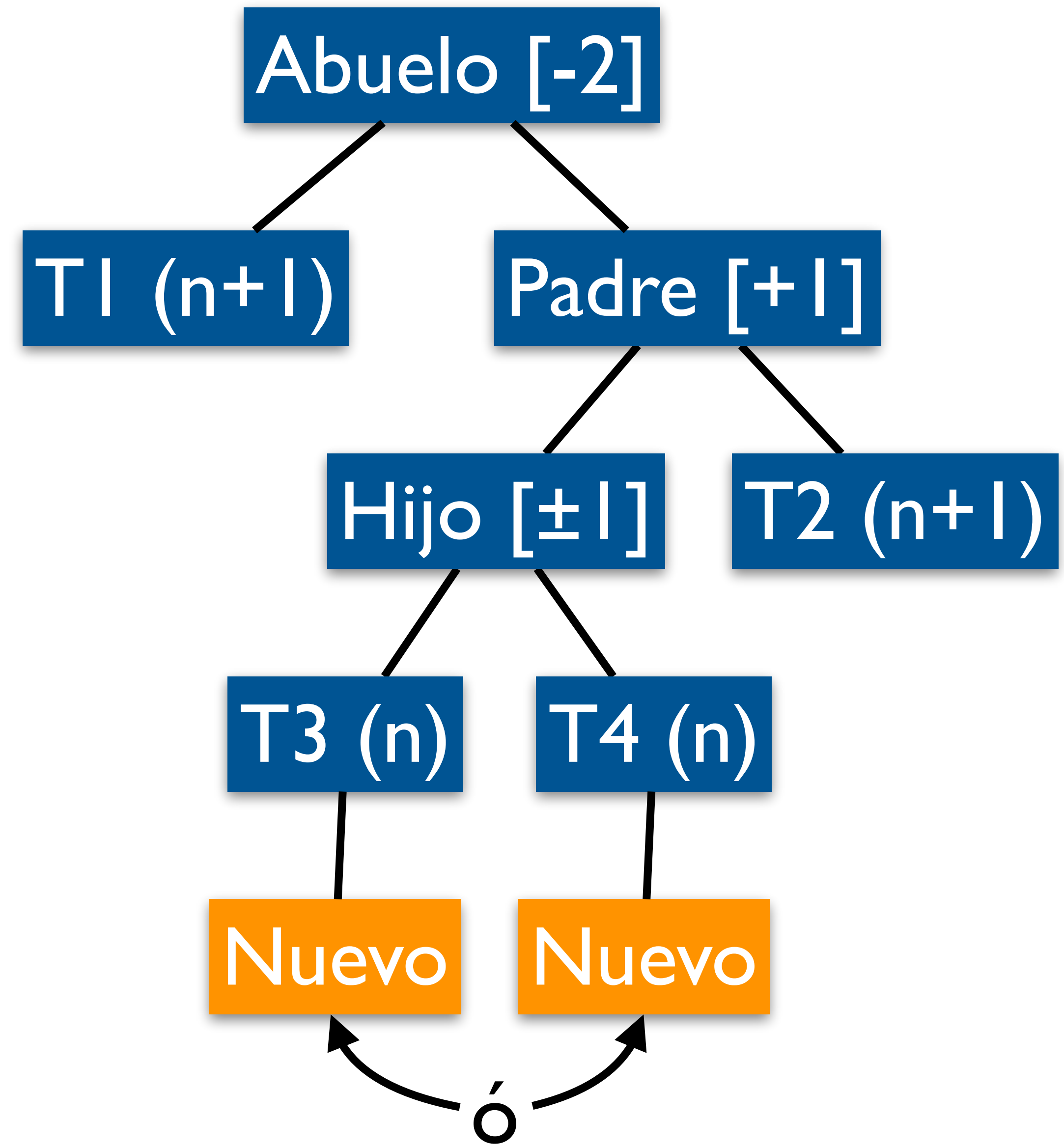
1. Hacer una rotación simple a la derecha sobre el hijo derecho del nodo que tenga desequilibrio
2. Y hacer una rotación simple a la izquierda sobre el dicho nodo



Rotación doble a la izquierda

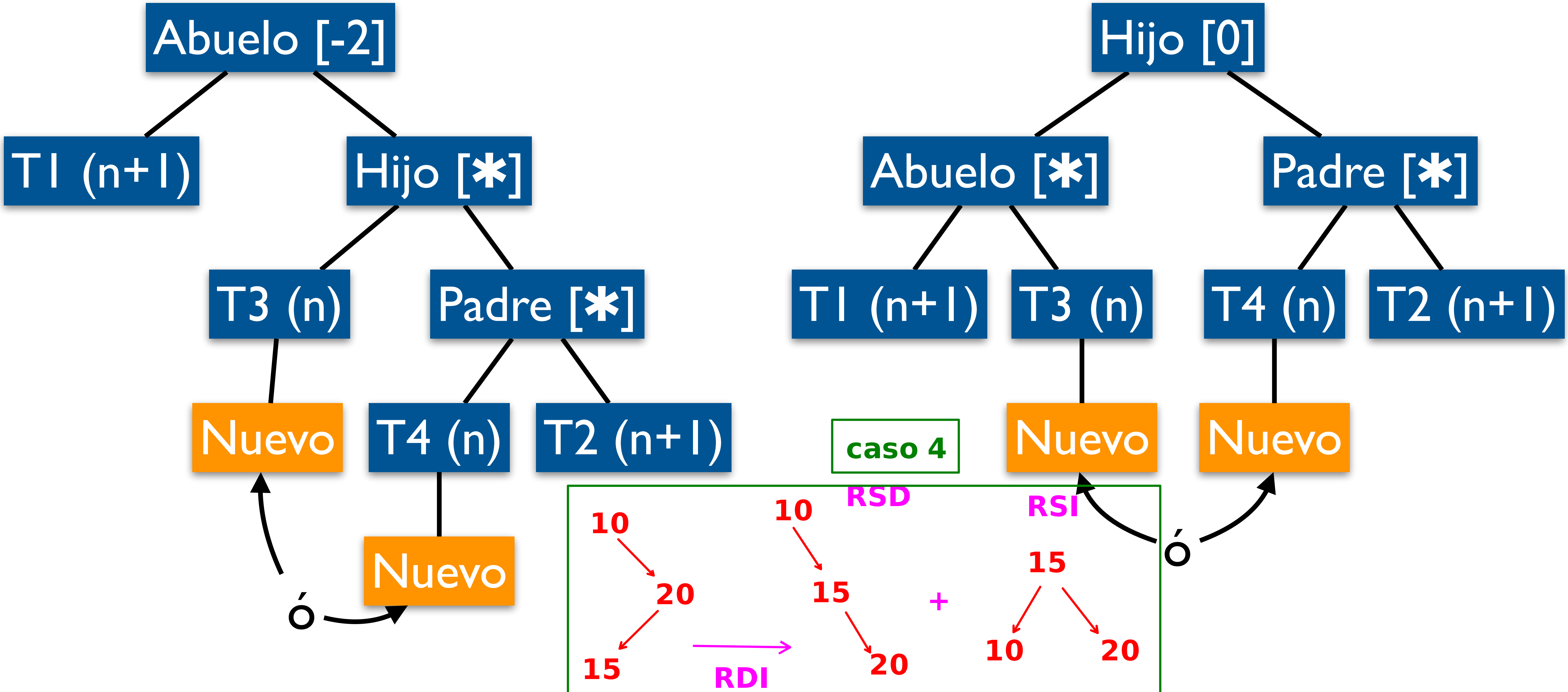


AVL inicial



AVL desequilibrado
tras insertar

Rotación doble a la izquierda

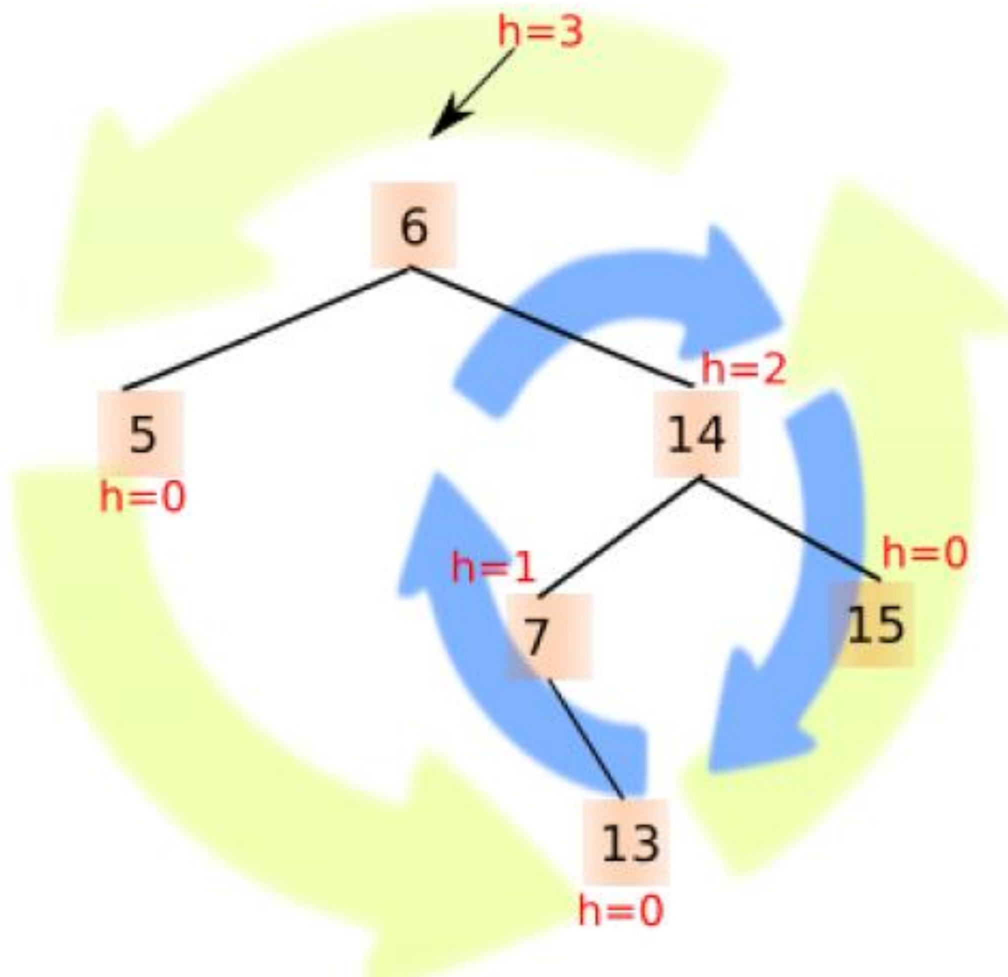


Rotación simple a derecha en padre

Rotación simple a izquierda en abuelo

Rotaciones Dobles

CASO C

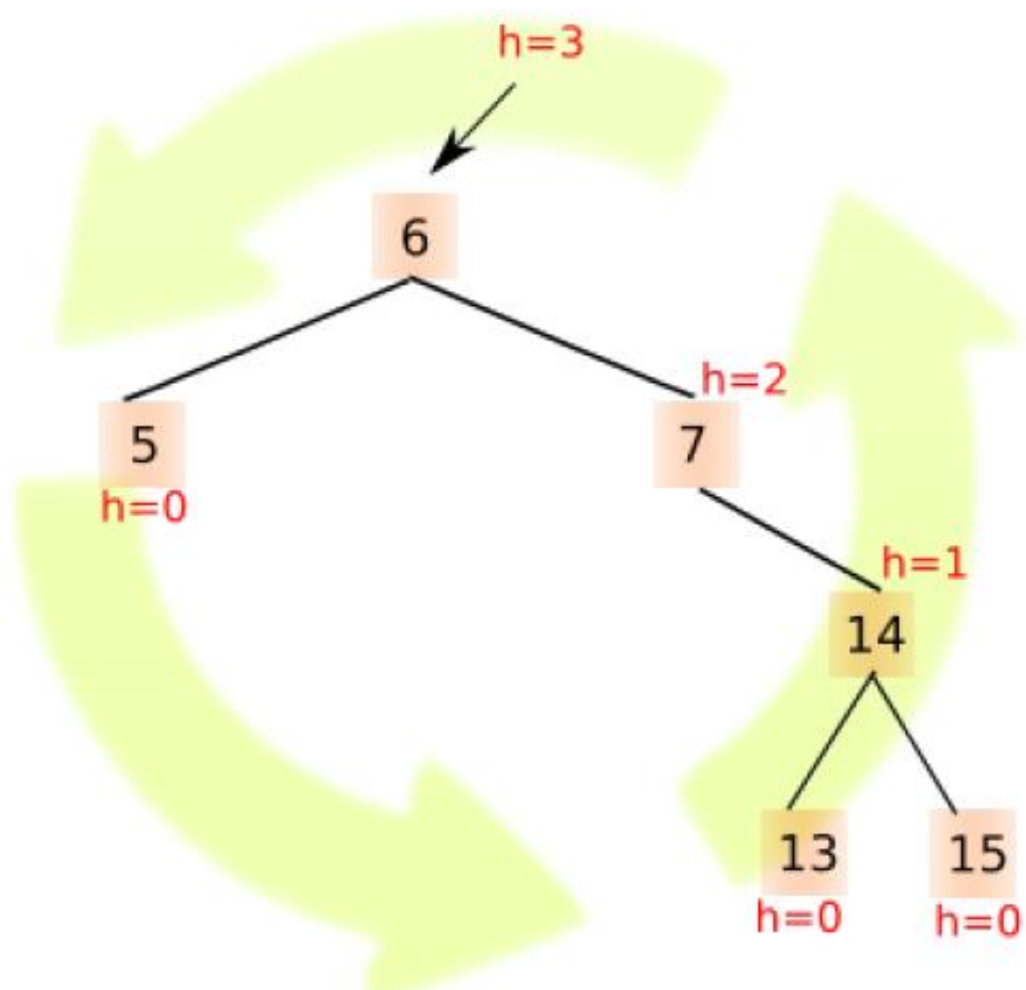


- El desequilibrio está en el nodo de etiqueta 6, para equilibrar el árbol debemos hacerlo en dos pasos

Rotaciones Dobles

CASO C

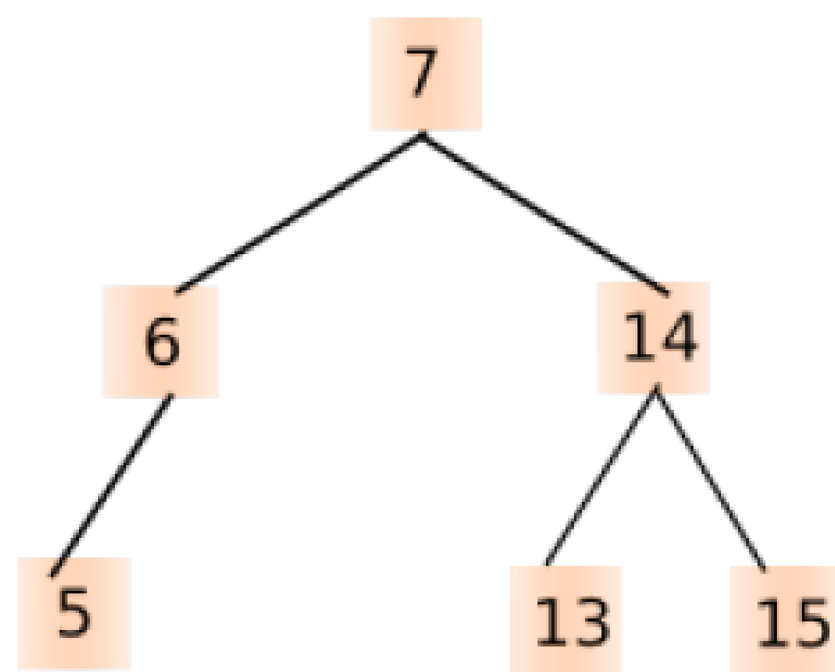
1. En primer lugar, haremos una rotación simple a la derecha sobre 14:



Rotaciones Dobles

CASO C

2. Y por último, hacemos una rotación simple a la izquierda sobre el nodo desequilibrado, 6:



Rotaciones Dobles

CASO C

La rotación doble consistiría, por tanto, en llamar a las funciones de rotación simples pasando como argumento los nodos correspondientes:

```
template <class T>
void Doble_DerechaIzquierda (info_nodo_AVL<T> * & n) {
        SimpleDerecha(n->hijoder);
        SimpleIzquierda (n);
}
```

¿Qué rotación utilizar?

Si la inserción se realiza en:

- el hijo izquierdo del hijo izquierdo del nodo desequilibrado $\Rightarrow$RSD
- el hijo derecho del hijo derecho del nodo desequilibrado $\Rightarrow$RSI
- el hijo derecho del hijo izquierdo del nodo desequilibrado $\Rightarrow$RDD
- el hijo izquierdo del hijo derecho del nodo desequilibrado $\Rightarrow$RDI

Ejemplos

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

https://es.wikipedia.org/wiki/%C3%81rbol_AVL

Ejemplo

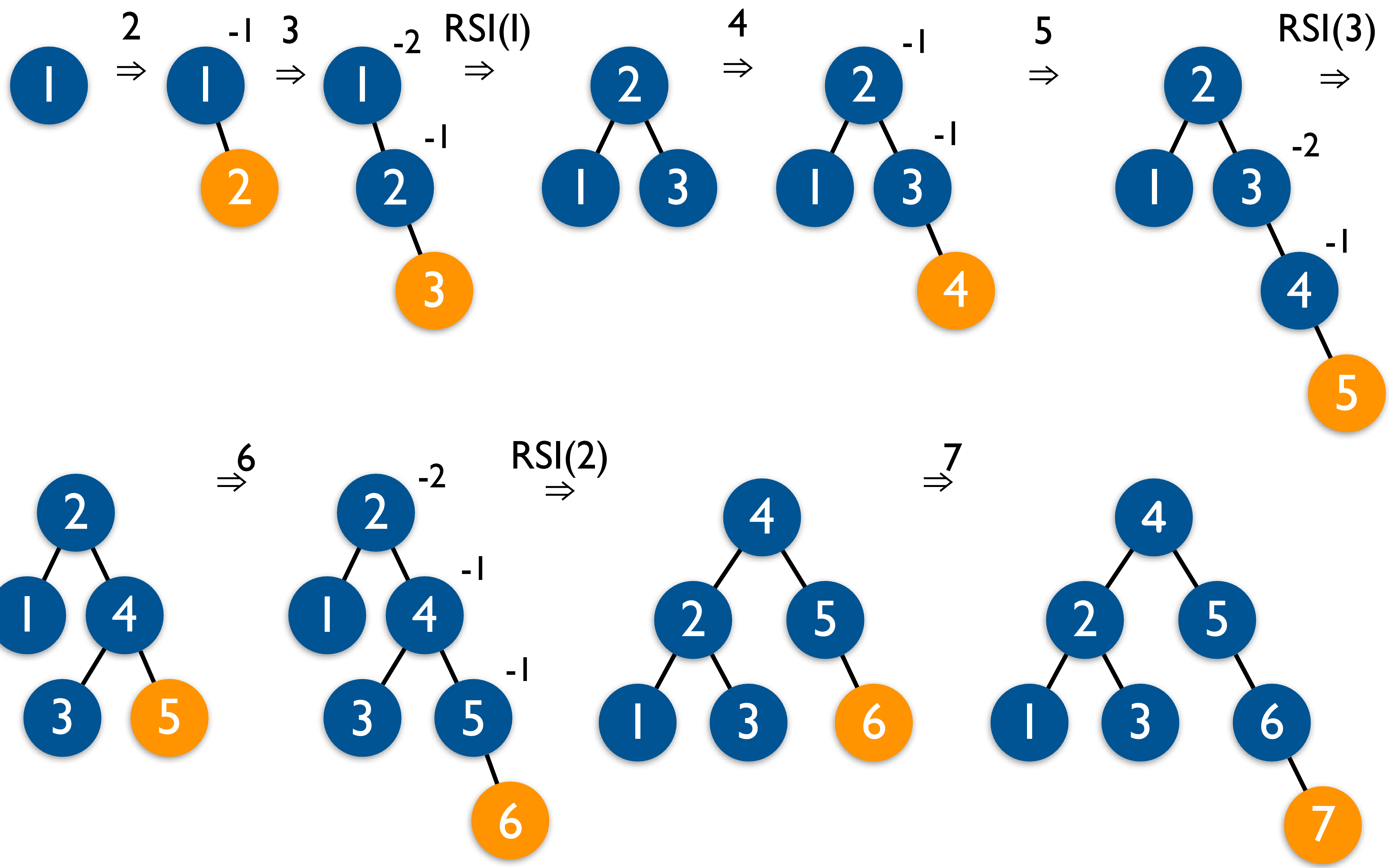
Dada una lista de números, crear un árbol binario de búsqueda AVL:

{1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9}

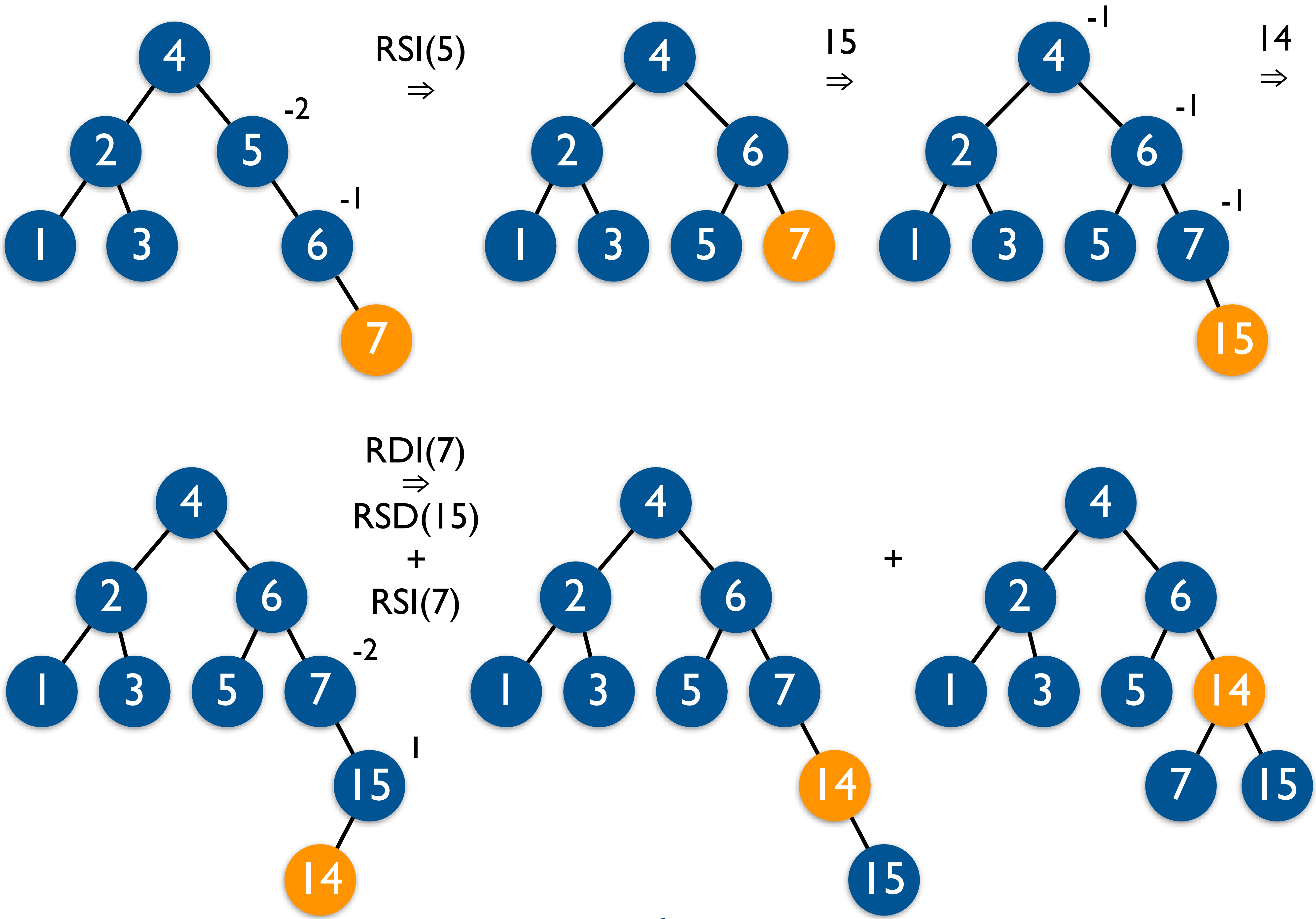
Los pasos para resolver este ejercicio son, elemento a elemento:

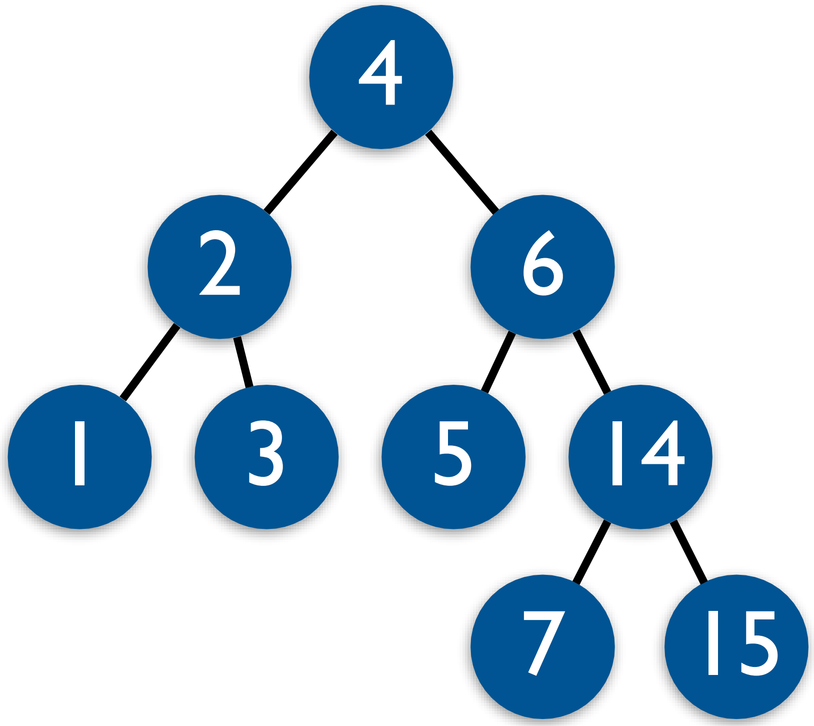
1. Insertar el elemento que corresponda
2. Comprobar que el árbol está equilibrado
   a) Si lo está, seguimos insertando elementos donde corresponda
   b) Si no lo está, lo equilibramos antes de seguir insertando

Ejemplo

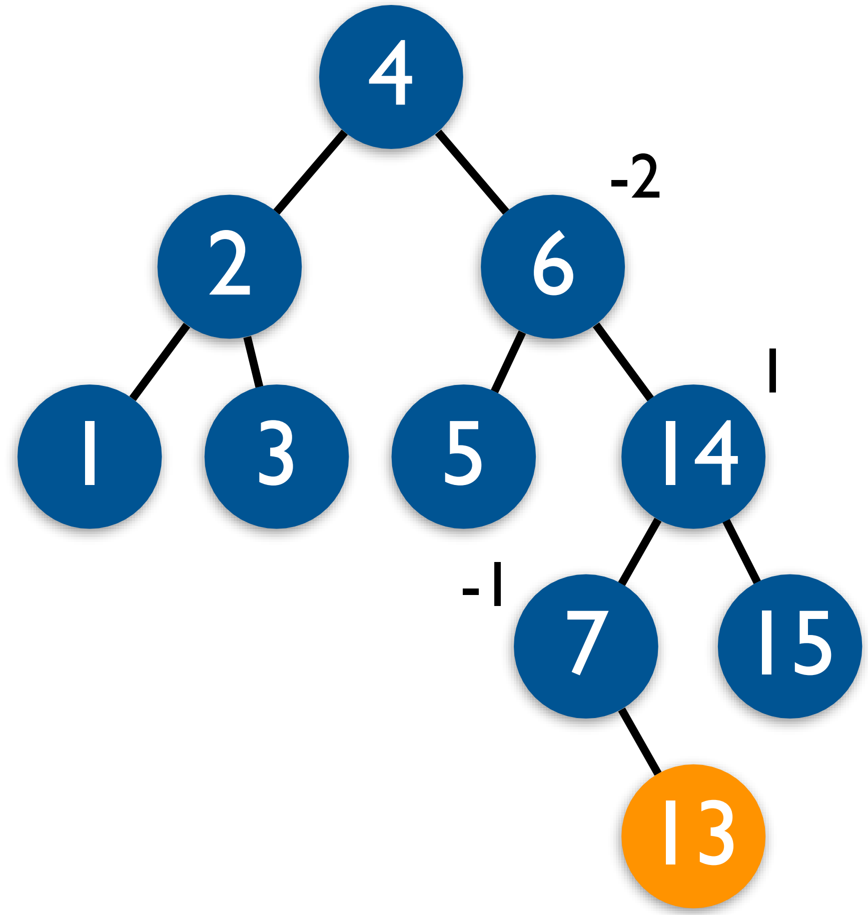


2
⇒
-1
3
⇒
-2
-1
RSI(1)
⇒
4
⇒
-1
-1
5
⇒
RSI(3)
⇒
-2
-1
1
1
2
1
2
3
2
1
3
2
1
3
4
2
1
3
4
5
6
⇒
-2
-1
-1
RSI(2)
⇒
7
⇒
2
1
4
3
5
2
1
4
3
5
6
4
2
5
1
3
6
4
2
5
1
3
6
7

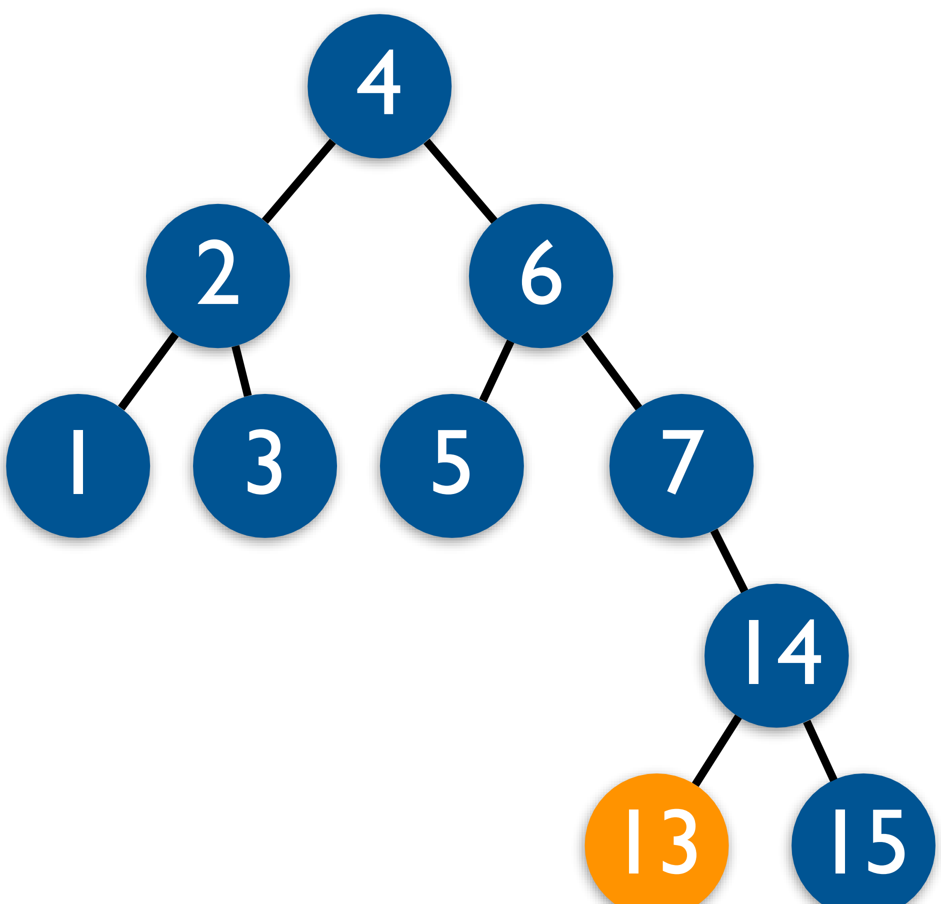

RSI(5)
⇒
-2
-1
15
⇒
-1
-1
-1
14
⇒
RDI(7)
⇒
RSD(15)
+
RSI(7)
-2
1
+



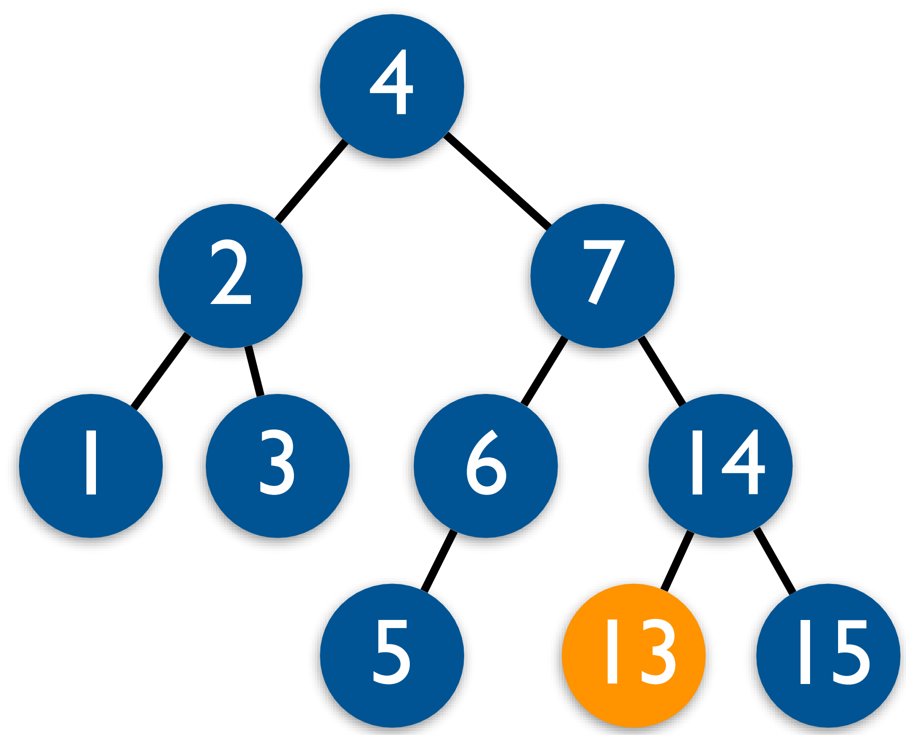
13
$\Rightarrow$



RDI(6)
$\Rightarrow$
RSD(14)
+
RSI(6)

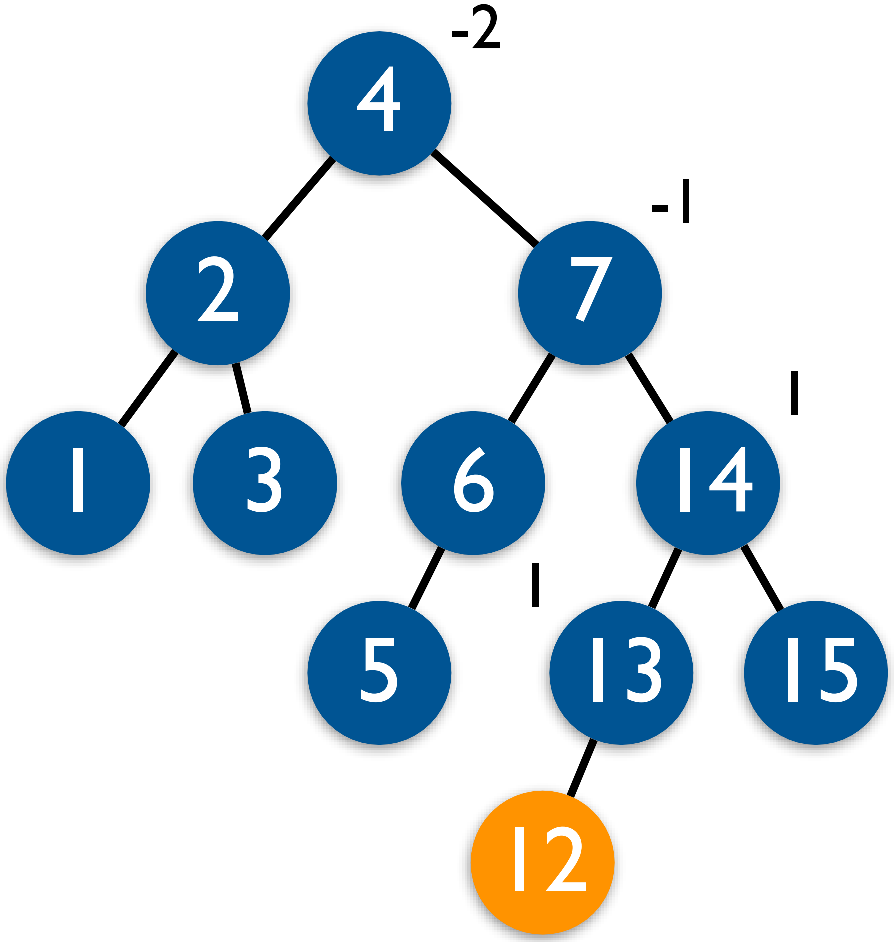


+

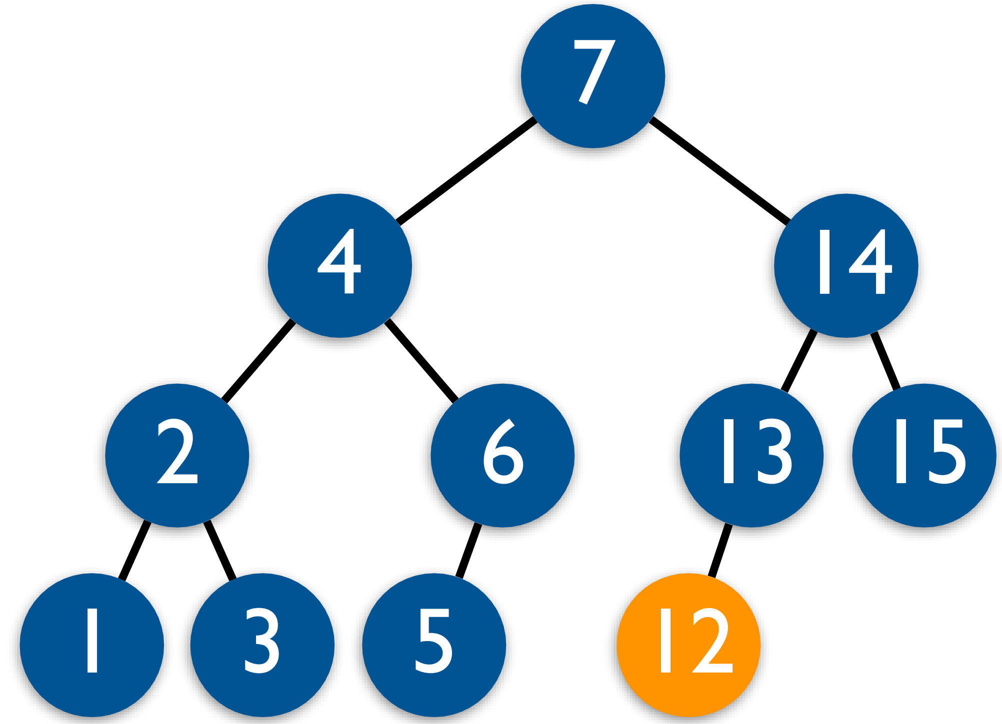


12
$\Rightarrow$

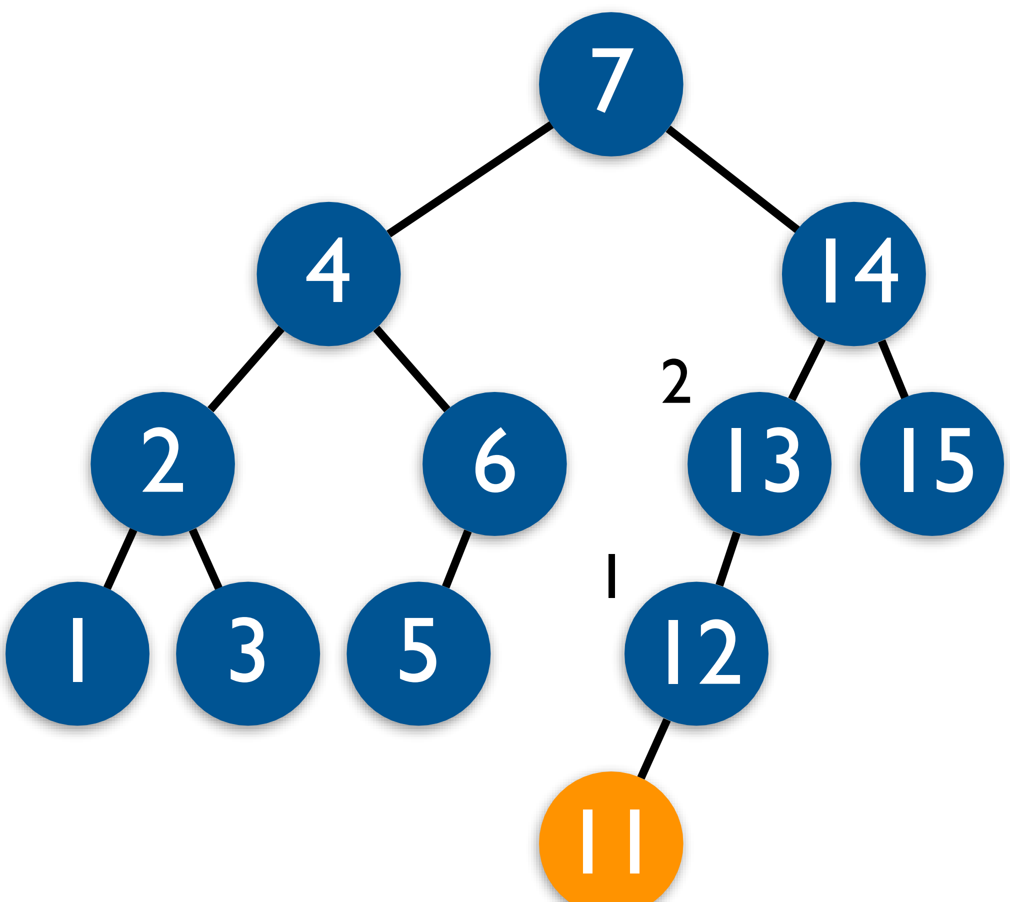
12 $\Rightarrow$



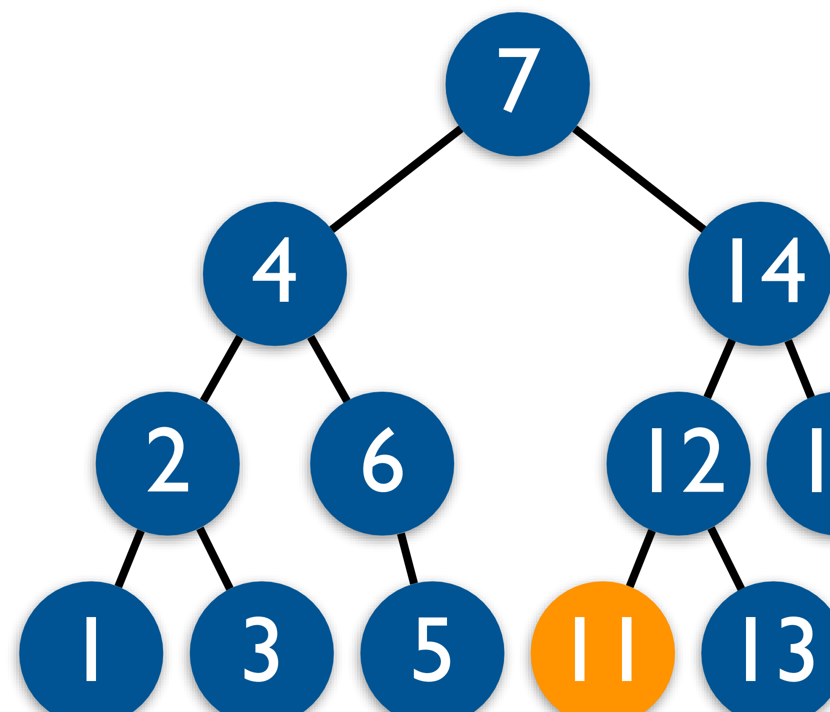
RSI(4) $\Rightarrow$



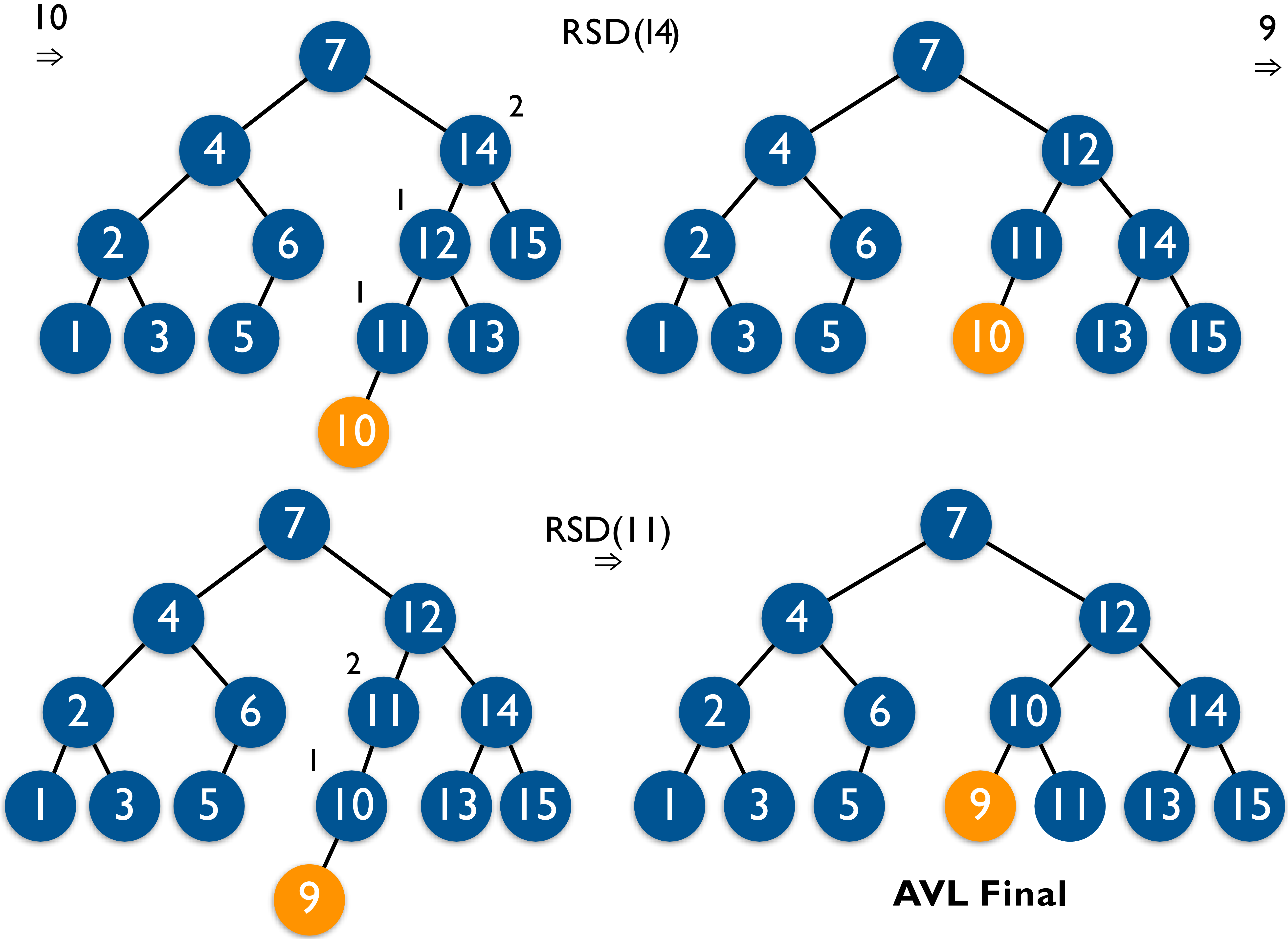
11 $\Rightarrow$



RSD(13) $\Rightarrow$



10 $\Rightarrow$


10 ⇒
RSD(14)
9 ⇒
RSD(11) ⇒
AVL Final

Inserción

```
template <class T>
bool InsertarAVL (info_nodo_AVL<T> * & raiz, T x) {
  if (raiz == 0) {
    // constructor: crea un nodo vacio con et=x
    raiz = new info_nodo_AVL(x);
    raiz->altura = 0;
    return true; // el nodo ha podido insertarse con exito
  }
  else {
    if (x < raiz->et) {
      if (raiz->hijoder != 0)
      raiz->hijoder->padre = raiz;
      if (InsertarAVL(raiz->hijoizq, x)) { // el arbol ha crecido,
      // vemos la diferencia de altura entre ambos hijos
      switch (Altura(raiz->hijoizq) - Altura(raiz->hijoder)) {
      // los valores del switch deben ser 0, 1 o 2, si son mayores
      // en algun momento no hemos insertado un elemento bien y
      // nuestro arbol no esta equilibrado.
      case 0:
      return false; // el arbol no ha crecido

      case 1: // ha crecido por la izquierda, sumamos 1 a la altura
      raiz->altura++;    // de la raiz
      return true;
```

Inserción

```
        case 2:
        /* CASO A*/
        if (Altura(raiz->hijoizq->hijoizq) >
        Altura(raiz->hijoizq->hijoder))
        SimpleDerecha(raiz);

        /*CASO B*/
        else
        Doble_IzquierdaDerecha(raiz);

        return false; // la altura no crece porque hemos
              // equilibrado el arbol
        }
        }
      }
```

Inserción

```
else { // x es mayor que la etiqueta
  if (raiz->hijoizq != 0)
  raiz->hijoizq->padre = raiz;

  if (InsertarAVL(raiz->hijoder, x)) {
  switch (Altura(raiz->hijoder) - Altura(raiz->hijoizq)) {
  case 0:
  return false; // el arbol no ha crecido

  case 1: // ha crecido por la izquierda, sumamos 1 a la altura
  raiz->altura++;   // de la raiz
  return true;
```

Inserción

```
            case 2:
            /* CASO D */
            if (Altura(raiz->hijoder->hijoder) >
            Altura(raiz->hijoder->hijoizq))
            SimpleIzquierda(raiz);

            /* CASO C */
            else
            Doble_DerechoIzquierda(raiz);

            return false;
            }
            }
        }
    }
}
```

Árboles equilibrados AVL

- Son árboles binarios de búsqueda equilibrados. Las operaciones de inserción y borrado tienen un orden de eficiencia logarítmico.
- Se caracterizan porque para cada nodo se cumple que la diferencia de la altura de sus dos hijos es como mucho de una unidad.
- La especificación coincide con la del Árbol binario de búsqueda.
- La implementación varía en las operaciones que modifican la altura de un nodo: insertar y borrar.

Implementación

```
template <class Tbase>
void AVL<Tbase>::ajustarArbol
   (ArbolBinario<Tbase>::Nodo &n)
{
   int aIzda;
   int aDcha;
   ArbolBinario<Tbase>::Nodo hIzda, hDcha;

   // Ajustamos desde n hasta la raíz del árbol
   while (n!=ArbolBinario<Tbase>::NODO_NULO) {
     aIzda = altura(arbolb.HijoIzqda(n));
     aDcha = altura(arbolb.HijoDrcha(n));

     if (abs(aIzda-aDcha)>1) // Hay que ajustar
       if (aIzda>aDcha) {
         hIzda = arbolb.HijoIzqda(n);
         if (altura(arbolb.HijoIzqda(hIzda)) >
     altura(arbolb.HijoDrcha(hIzda)))
           rotarHijoIzqda(n);
         else {
           rotarHijoDrcha(hIzda);
```

```
            rotarHijoIzqda(n);
        }
      }
      else { // Exceso de altura por la dcha
        hDcha = arbolb.HijoDrcha(n);
        if (altura(arbolb.HijoIzqda(hDcha)) >
  altura(arbolb.HijoDrcha(hDcha))) {
          rotarHijoIzqda(hDcha);
          rotarHijoDrcha(n);
        }
        else
          rotarHijoDrcha(n);
      }

    n = arbolb.Padre(n);
  }
}
```

```
template <class Tbase>
void AVL<Tbase>::rotarHijoIzqda
   (ArbolBinario<Tbase>::Nodo &n)
{
  assert(n!=ArbolBinario<Tbase>::NODO_NULO);

  char que_hijo;
```

simétrico rotarHijoDrcha

```
ArbolBinario<Tbase>::Nodo elPadre =
  arbolb.Padre(n);

ArbolBinario<Tbase> A;
arbolb.PodarHijoIzqda(n, A);

ArbolBinario<Tbase> Aux;
if (elPadre!=ArbolBinario<Tbase>::NODO_NULO)
  if (arbolb.HijoIzqda(elPadre)==n) {
    arbolb.PodarHijoIzqda(elPadre, Aux);
    que_hijo = IZDA;
  }
  else {
    arbolb.PodarHijoDrcha(elPadre, Aux);
    que_hijo = DCHA;
  }
else
  Aux = arbolb;

ArbolBinario<Tbase> B;
A.PodarHijoDrcha(A.Raiz(), B);

Aux.InsertarHijoIzqda(Aux.Raiz(), B);
A.InsertarHijoDrcha(A.Raiz(), Aux);
```

doble o simple ?

```
    if (elPadre!=ArbolBinario<Tbase>::NODO_NULO) {
      if (que_hijo==IZDA) {
        arbolb.InsertarHijoIzqda(elPadre, A);
        n = arbolb.HijoIzqda(elPadre);
      }
      else {
        arbolb.InsertarHijoDrcha(elPadre, A);
        n = arbolb.HijoDrcha(elPadre);
      }
    }
    else {
      arbolb = A;
      n = arbolb.Raiz();
    }
}
```

```
template <class Tbase>
void AVL<Tbase>::rotarHijoDrcha
  (ArbolBinario<Tbase>::Nodo &n)
{
  assert(n!=ArbolBinario<Tbase>::NODO_NULO);

  char que_hijo;
```

```
ArbolBinario<Tbase>::Nodo elPadre =
  arbolb.Padre(n);

ArbolBinario<Tbase> A;
arbolb.PodarHijoDrcha(n, A);

ArbolBinario<Tbase> Aux;
if (elPadre!=ArbolBinario<Tbase>::NODO_NULO)
  if (arbolb.HijoIzqda(elPadre)==n) {
    que_hijo = IZDA;
    arbolb.PodarHijoIzqda(elPadre, Aux);
  }
  else {
    que_hijo = DCHA;
    arbolb.PodarHijoDrcha(elPadre, Aux);
  }
else
  Aux = arbolb;

ArbolBinario<Tbase> B;
A.PodarHijoIzqda(A.Raiz(), B);

Aux.InsertarHijoDrcha(Aux.Raiz(), B);
A.InsertarHijoIzqda(A.Raiz(), Aux);
```

```
  if (elPadre!=ArbolBinario<Tbase>::NODO_NULO) {
    if (que_hijo==IZDA) {
      arbolb.InsertarHijoIzqda(elPadre, A);
      n = arbolb.HijoIzqda(elPadre);
    }
    else {
      arbolb.InsertarHijoDrcha(elPadre, A);
      n = arbolb.HijoDrcha(elPadre);
    }
  }
  else {
    arbolb = A;
    n = arbolb.Raiz();
  }
}
```