Árboles balanceados: AVLs

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025

Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

UNRC
CREER... CREAR... CRECER

CONICET

AVL’s

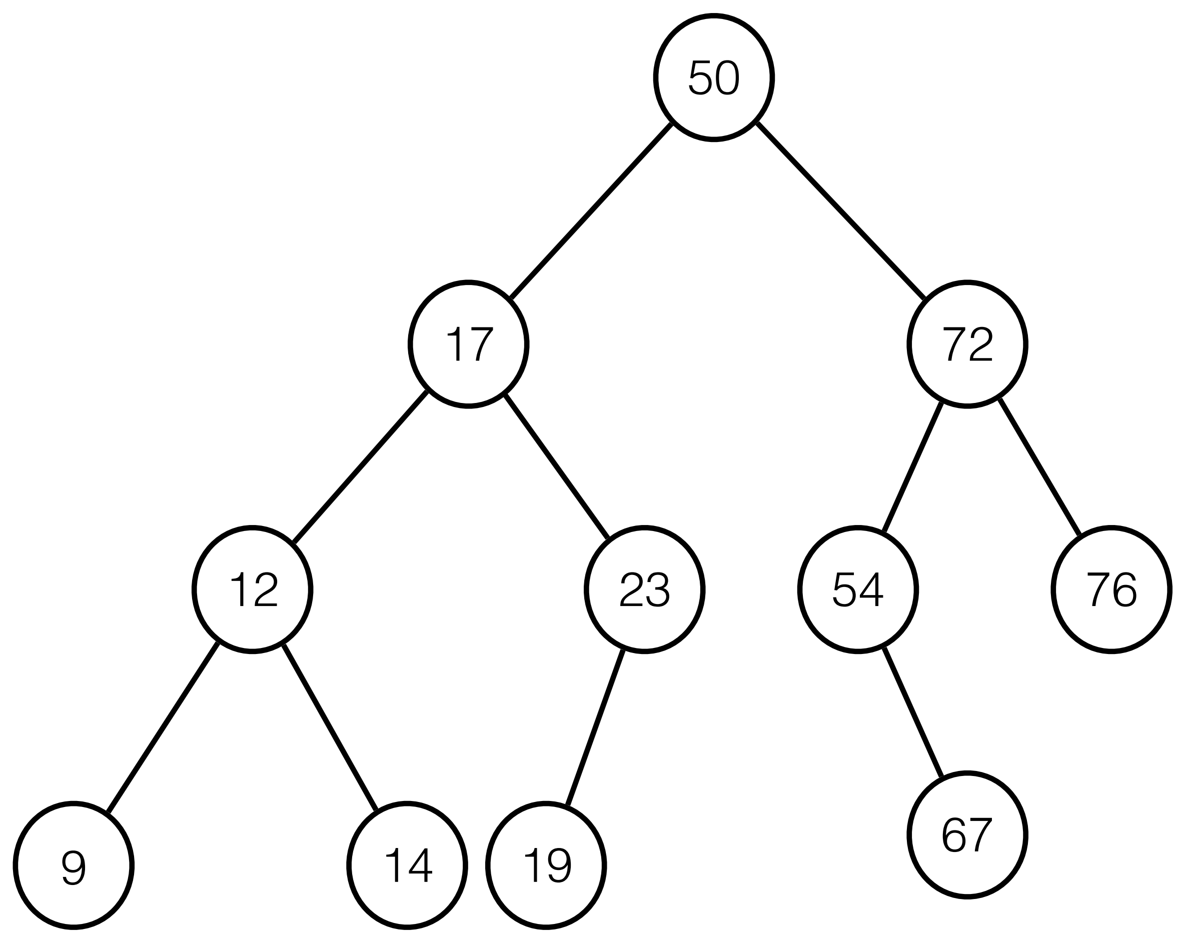
Los AVL’s solucionan algunos de los problemas de los ABB’s.

- AVL = Adelson-Velskii y Landis (los autores),
- Son ABB’s pero con ciertas restricciones que aseguran los árboles se mantienen balanceados:
  - La altura de sus hijos puede diferir en a lo sumo 1
- Insertar, eliminar y buscar son O(log n).

Pedir que sean exactamente balanceados no tiene sentido ya que no podríamos insertar ni eliminar.

Un Ejemplo

Un ejemplo de AVL:



Para todo nodo, las alturas de los subárboles izquierdo y derecho difieren por a lo sumo 1

Implementando AVLs

Podemos implementar los AVL's de la siguiente forma:

- Cada nodo lleva la altura,
- El factor de balance es la diferencia entre la altura del hijo izquierdo y el hijo derecho, y se calcula como:

  FB = altura(hi) - altura(hd)   *(= -1,0,1 para AVLs)*

- Si el valor es -1 el hijo der. tiene más altura, si es 0 los dos tienen igual altura, si es 1 el hijo izq. tiene más altura

Implementación en Java

```java
/**
 * AVLSet is an implementation of unbounded sets of
 * objects of type T, based on AVLs.
 * A typical AVLSet is {o1, . . . , on}.
 *
 * AVLSet requires that the key type T implements the
 * Comparable interface. AVLSet calls the compareTo method
 * to compare two keys in many of the operations.
 *
 * The methods use compareTo to determine equality of elements.
 */
public class AVLSet<T extends Comparable<? super T>> implements SortedSet<T>
{
    private Node root;          // root of BST
    private int size;           // number of nodes in subtree

    private class Node {
        private final T key;       // the key
        private int height;        // height of the subtree
        private Node left;         // left subtree
        private Node right;        // right subtree

        public Node(T key, int height) {
            this.key = key;
            this.height = height;
        }
```

Implementación en Java

```java
/**
 * AVLSet is an implementation of unbounde
 * objects of type T, based on AVLs.
 * A typical AVLSet is {o1, . . . , on}.
 *
 * AVLSet requires that the key type T im
 * Comparable interface. AVLSet calls the
 * to compare two keys in many of the ope
 *
 * The methods use compareTo to determine
 */
public class AVLSet<T extends Comparable<
{
    private Node root;          // root of
    private int size;           // number

    private class Node {
        private final T key;      // the key
        private int height;       // height of the subtree
        private Node left;        // left subtree
        private Node right;       // right subtree

        public Node(T key, int height) {
            this.key = key;
            this.height = height;
        }
```

```java
/**
 * @post Returns the height of the AVL tree.
 * It is assumed that the height of an empty tree is -1
 * and the height of a tree with just one node is 0.
 */
public int height() {
    return height(root);
}
```

```java
/**
 * @post Returns the height of the subtree with root x.
 */
private int height(Node x) {
    if (x == null)
        return 0;
    return x.height;
}
```

Implementación en Java

```java
/**
 * AVLSet is an implementation of unbounde
 * objects of type T, based on AVLs.
 * A typical AVLSet is {o1, . . . , on}.
 *
 * AVLSet requires that the key type T im
 * Comparable interface. AVLSet calls the
 * to compare two keys in many of the ope
 *
 * The methods use compareTo to determine
 */
public class AVLSet<T extends Comparable<
{
    private Node root;          // root of
    private int size;           // number 

    private class Node {
        private final T key;        // the key
        private int height;         // height
        private Node left;          // left su
        private Node right;         // right s

        public Node(T key, int height) {
            this.key = key;
            this.height = height;
        }
```

```java
/**
 * @post Returns the height of the AVL tree.
 * It is assumed that the height of an empty tree is -1
 * and the height of a tree with just one node is 0.
 */
public int height() {
    return height(root);
}
```

```java
/**
 * @post Returns the height of the subtree with root x.
 */
private int height(Node x) {
    if (x == null)
        return 0;
    return x.height;
}
```

```java
/**
 * @post Returns the balance factor of the subtree.
 *   The balance factor is defined as the difference
 *   in height of the left subtree and right subtree,
 *   in this order. Therefore, a subtree with a balance
 *   factor of -1, 0 or 1 has the AVL property since
 *   the heights of the two child subtrees differ by at
 *   most one.
 */
private int balanceFactor(Node x) {
    return height(x.left) - height(x.right);
}
```

Implementación en Java

```java
/**
 * AVLSet is an implementation of unbound
 * objects of type T, based on AVLs.
 * A typical AVLSet is {o1, . . . , on}.
 *
 * AVLSet requires that the key type T im
 * Comparable interface. AVLSet calls the
 * to compare two keys in many of the ope
 *
 * The methods use compareTo to determine
 */
public class AVLSet<T extends Comparable<
{
                                   // root of
                                   // number

    pr
        private final T key;       // the key
        private int height;        // height
        private Node left;         // left su
        private Node right;        // right s

        public Node(T key, int height) {
            this.key = key;
            this.height = height;
        }
```

Guardamos la altura de cada subárbol en los nodos

```java
/**
 * @post Returns the height of the AVL tree.
 * It is assumed that the height of an empty tree is -1
 * and the height of a tree with just one node is 0.
 */
public int height() {
    return height(root);
}
/**
 * @post Returns the height of the subtree with root x.
 */
private int height(Node x) {
    if (x == null)
        return 0;
    return x.height;
}
/**
 * @post Returns the balance factor of the subtree.
 *   The balance factor is defined as the difference
 *   in height of the left subtree and right subtree,
 *   in this order. Therefore, a subtree with a balance
 *   factor of -1, 0 or 1 has the AVL property since
 *   the heights of the two child subtrees differ by at
 *   most one.
 */
private int balanceFactor(Node x) {
    return height(x.left) - height(x.right);
}
```

Implementación en Java

```java
/**
 * AVLSet is an implementation of unbounde
 * objects of type T, based on AVLs.
 * A typical AVLSet is {o1, . . . , on}.
 *
 * AVLSet requires that the key type T im
 * Comparable interface. AVLSet calls the
 * to compare two keys in many of the ope
 *
 * The methods use compareTo to determine
 */
public class AVLSet<T extends Comparable<
{
                               // root of
                               // number
    pr
        private final T key;      // the key
        private int height;       // height
        private Node left;        // left su
                ight;             // right s

                       height) {

                       ght;
```

Guardamos la altura de cada subárbol en los nodos

Computaremos la altura a medida que vamos agregando/quitando nodos

```java
/**
 * @post Returns the height of the AVL tree.
 * It is assumed that the height of an empty tree is -1
 * and the height of a tree with just one node is 0.
 */
public int height() {
    return height(root);
}
```

```java
/**
 * @post Returns the height of the subtree with root x.
 */
private int height(Node x) {
    if (x == null)
        return 0;
    return x.height;
}
```

```java
/**
 * @post Returns the balance factor of the subtree.
 *   The balance factor is defined as the difference
 *   in height of the left subtree and right subtree,
 *   in this order. Therefore, a subtree with a balance
 *   factor of -1, 0 or 1 has the AVL property since
 *   the heights of the two child subtrees differ by at
 *   most one.
 */
private int balanceFactor(Node x) {
    return height(x.left) - height(x.right);
}
```

Implementación en Java

```java
/**
 * AVLSet is an implementation of unbound
 * objects of type T, based on AVLs.
 * A typical AVLSet is {o1, . . . , on}.
 *
 * AVLSet requires that the key type T im
 * Comparable interface. AVLSet calls the
 * to compare two keys in many of the ope
 *
 * The methods use compareTo to determine
 */
public class AVLSet<T extends Comparable<
{
                                  // root of
                                  // number
    pr
        private final T key;       // the key
        private int height;        // height
        private Node left;         // left su
                     ight;         // right s

                            height) {

                            ght;
```

Guardamos la altura de cada subárbol en los nodos

Computaremos la altura a medida que vamos agregando/quitando nodos

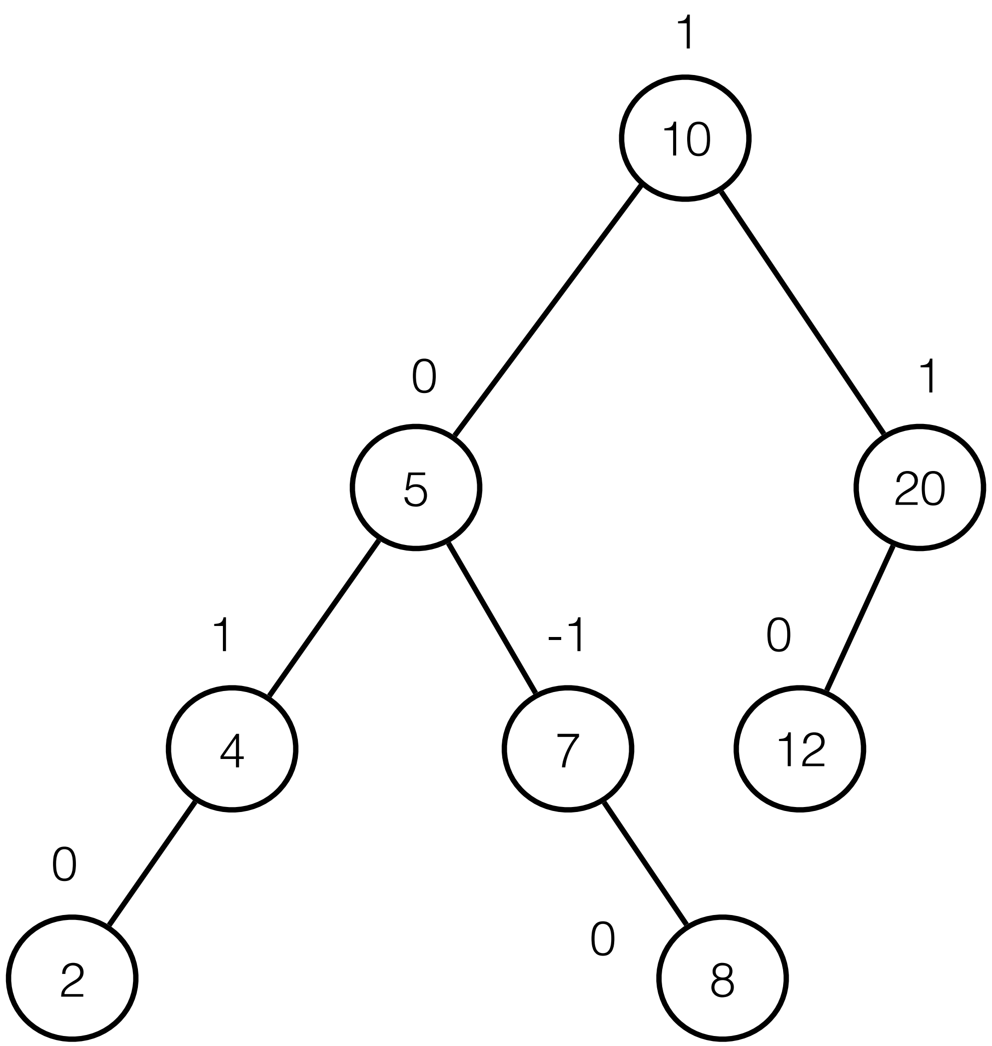
```java
/**
 * @post Returns the height of the AVL tree.
 * It is assumed that the height of an empty tree is -1
 * and the height of a tree with just one node is 0.
 */
public int height() {
    return height(root);
}
```

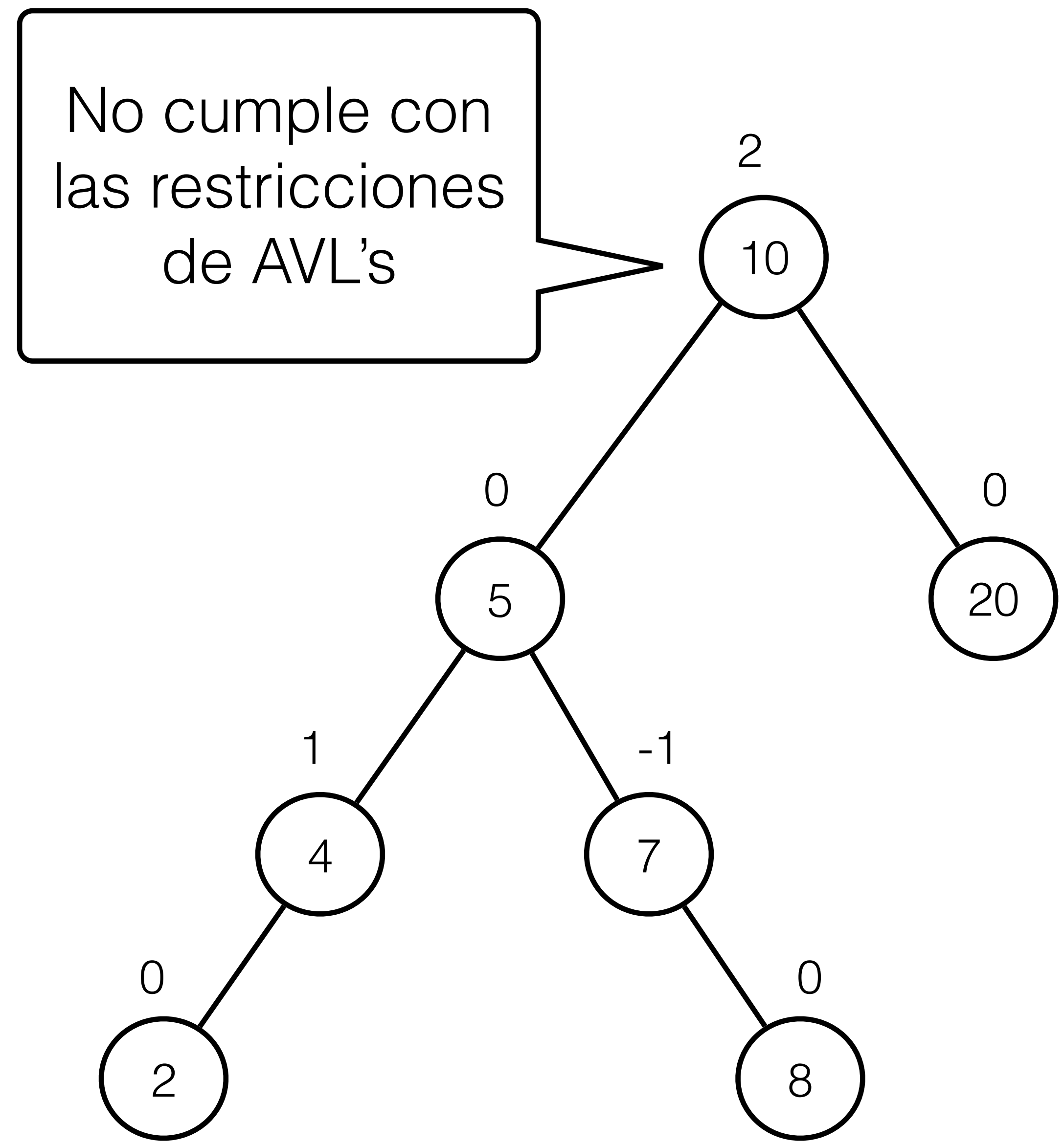
```java
/**
 * @post Returns the height of the subtree with root x.
 */
private int height(Node x) {
    if (x == null)
        return 0;
    return x.height;
}
```

```java
/**
 * @post Returns the balance factor of the subtree.
 *   The balance factor is defined as the difference
 *   in height of the left subtree and right subtree,
 *   in this order. Therefore, a                    a balance
 *   factor of -1, 0 or 1                             e
 *   the heights of the t                             at
 *   most one.
 */
private int balanceFactor(Node x) {
    return height(x.left) - height(x.right);
}
```

Retornar la altura toma tiempo constante

Ejemplos


1
10
0
1
5
20
1
-1
0
4
7
12
0
0
2
8


No cumple con las restricciones de AVL's
2
10
0
0
5
20
1
-1
4
7
0
0
2
8

Búsqueda en AVLs

- La búsqueda en AVL es igual que en ABB:
  - Comenzar desde el root
  - Si key es menor que la clave del root, buscar recursivamente en el subárbol izquierdo
  - Si key es mayor que la clave del root, buscar recursivamente en el subárbol derecho
- Casos base:
  - Si llegamos a un nodo que contiene key, retornamos true
  - Si llegamos a null, la clave no está en el ABB, y retornamos false
- La búsqueda es O(h), donde h es la altura
- Para AVLs, el tiempo de la búsqueda es O(log n)

```java
/**
 * @post Returns true iff 'key' is in 'this'.
 */
public boolean contains(T key) {
    return get(root, key) != null;
}
```

```java
/**
 * @post Returns true iff 'key' can be reached
 *    from node 'x'.
 */
private Node get(Node x, T key) {
    if (x == null)
        return null;

    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        return get(x.left, key);
    else if (cmp > 0)
        return get(x.right, key);
    else
        return x;
}
```

Inserción en AVL

La principal diferencia con la inserción en ABBs es que tenemos que preservar el balance

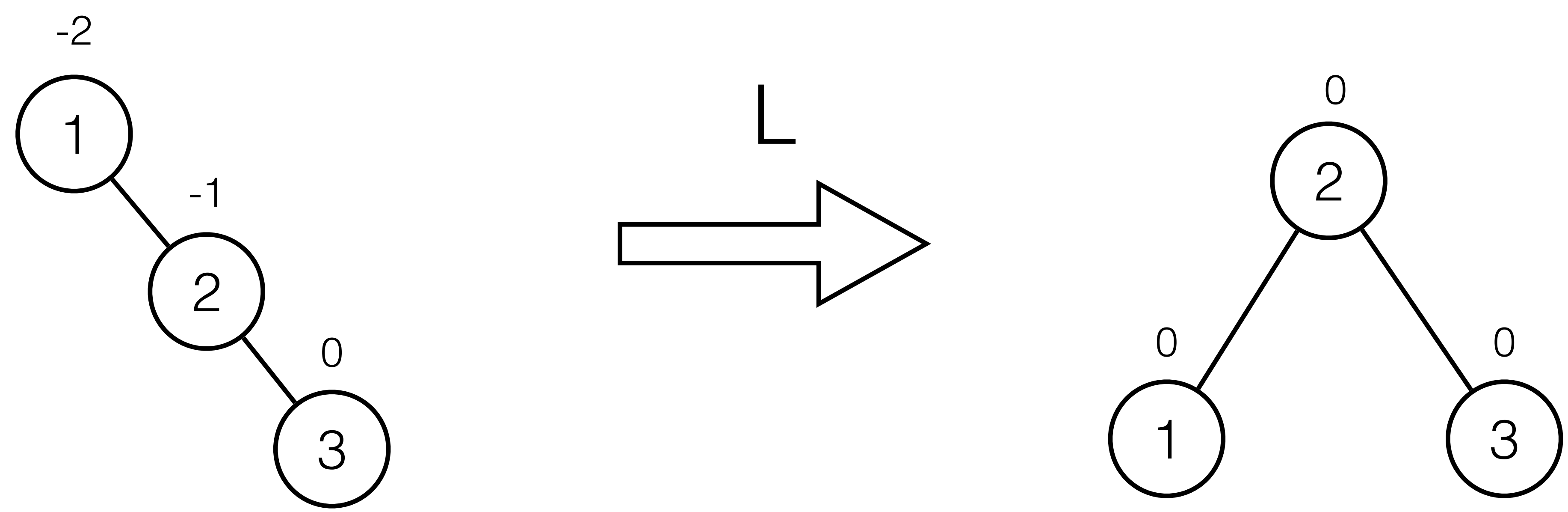
- Procedemos como los ABBs, buscamos el lugar en donde insertar el elemento en una hoja
- Creamos el nodo y vamos actualizando las referencias hacia arriba hasta la raíz
- La diferencia con ABBs, es que tenemos que ir rebalanceando los subárboles en caso de ser necesario
    - Esto es, cuando encontramos un factor de balance 2 o -2

Rotaciones en AVL’s

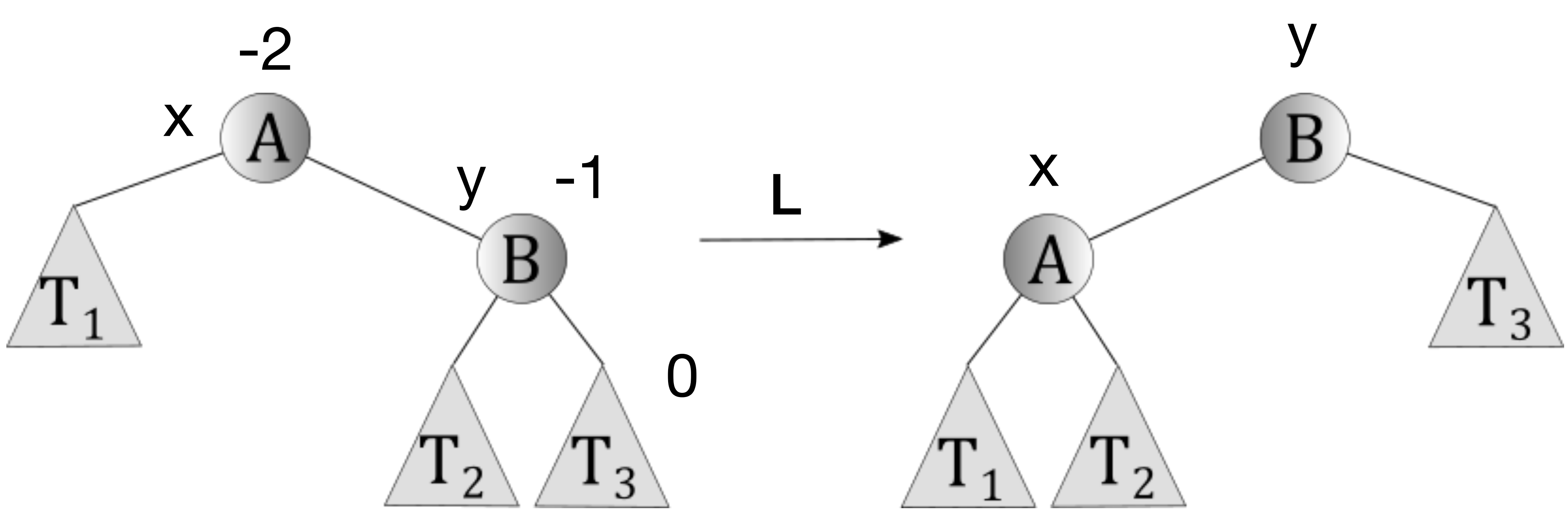
Tenemos cuatro tipos de transformaciones que permiten rebalancear un AVL:

- Rotación a la izquierda (L-rotación),
- Rotación a la derecha (R-rotación),
- Rotación izquierda-derecha (LR-rotación)
- Rotación derecha-izquierda (RL-rotación)

Rotación a la izquierda



Rotación a la izquierda

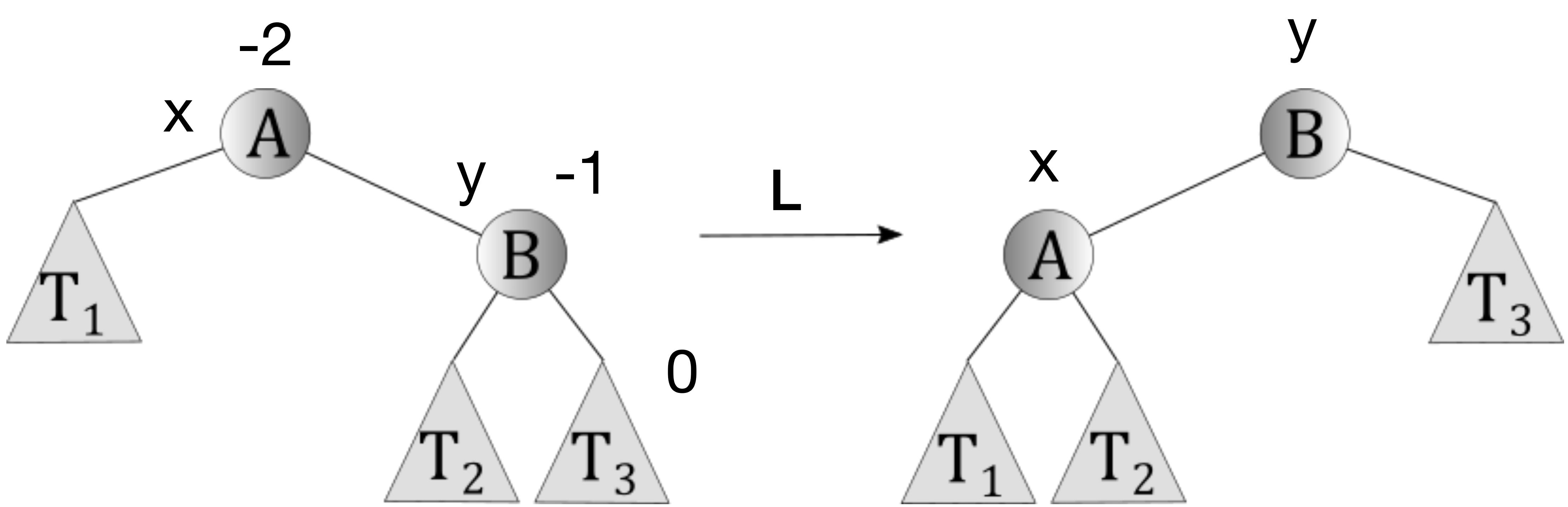


Todos los nodos en T2 > A

Todos los nodos en T2 < B

A < B

Rotación a la izquierda



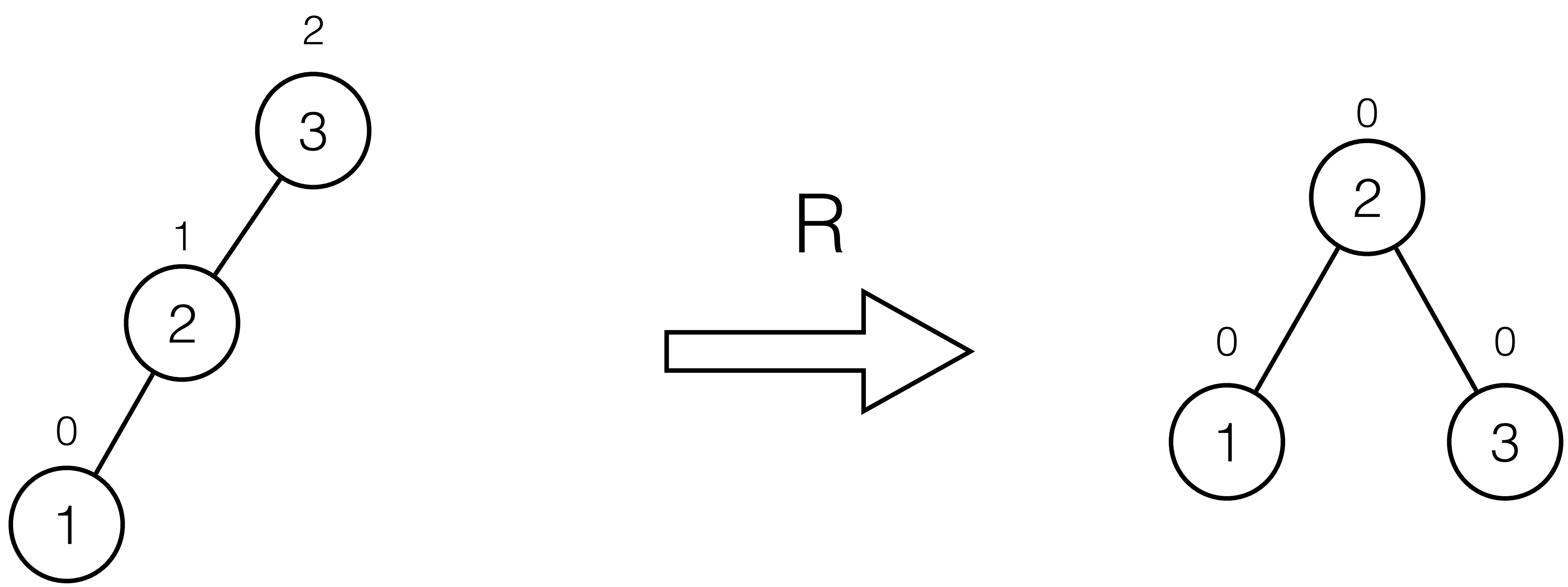
Todos los nodos en T2 > A

Todos los nodos en T2 < B

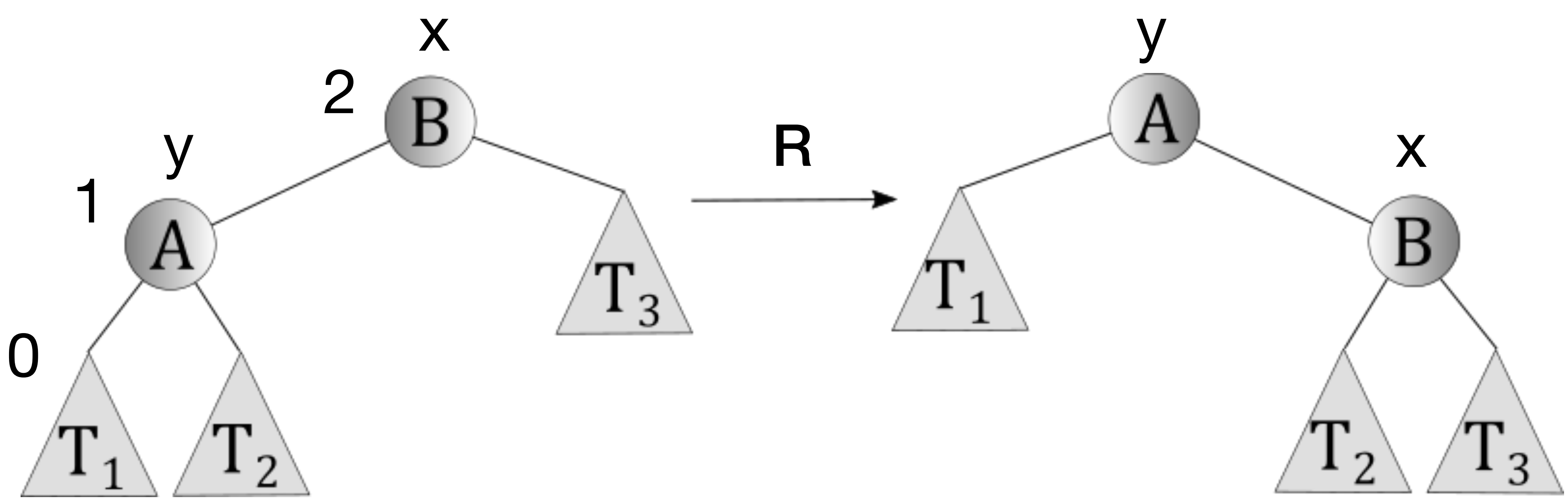
A < B

```java
/**
 * @post Rotates the given subtree to the left, and
 *   returns the root of the resulting tree.
 */
private Node rotateLeft(Node x) {
    Node y = x.right;
    x.right = y.left;
    y.left = x;
    x.height = 1 + Math.max(height(x.left), height(x.right));
    y.height = 1 + Math.max(height(y.left), height(y.right));
    return y;
}
```

Rotación a la derecha



Rotación a la derecha

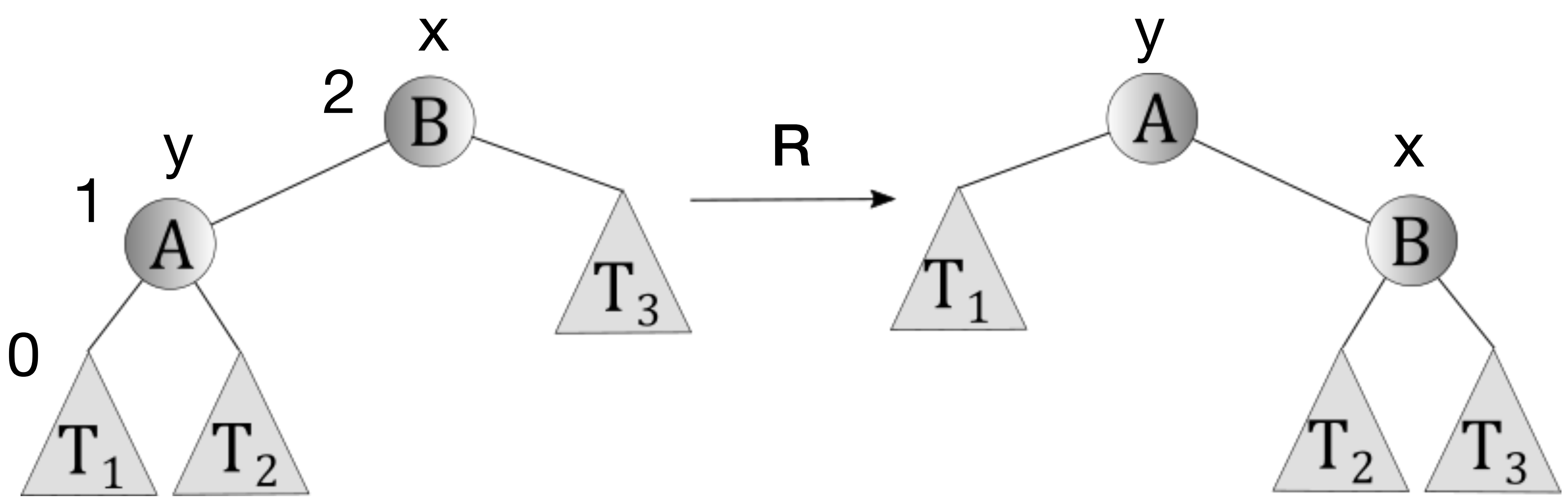


Todos los nodos en T2 > A

Todos los nodos en T2 < B

A < B

Rotación a la derecha



Todos los nodos en T2 > A

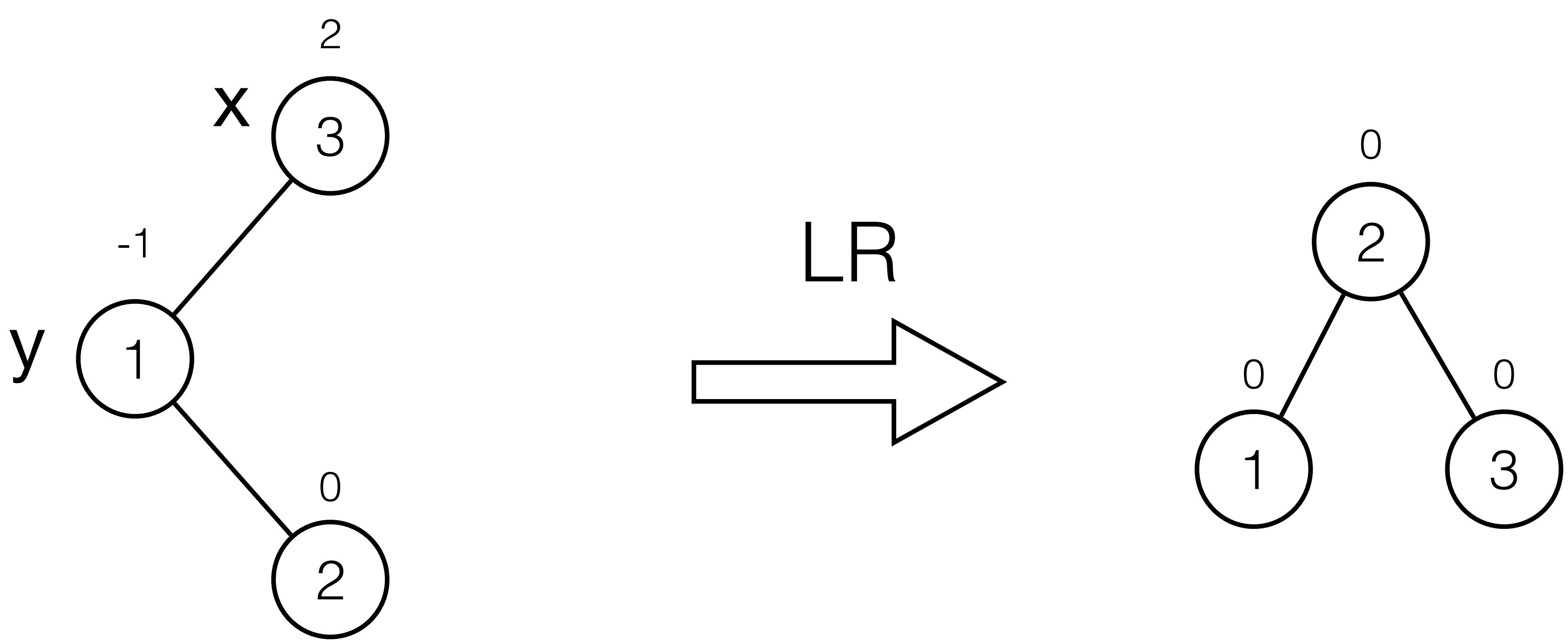
Todos los nodos en T2 < B

A < B

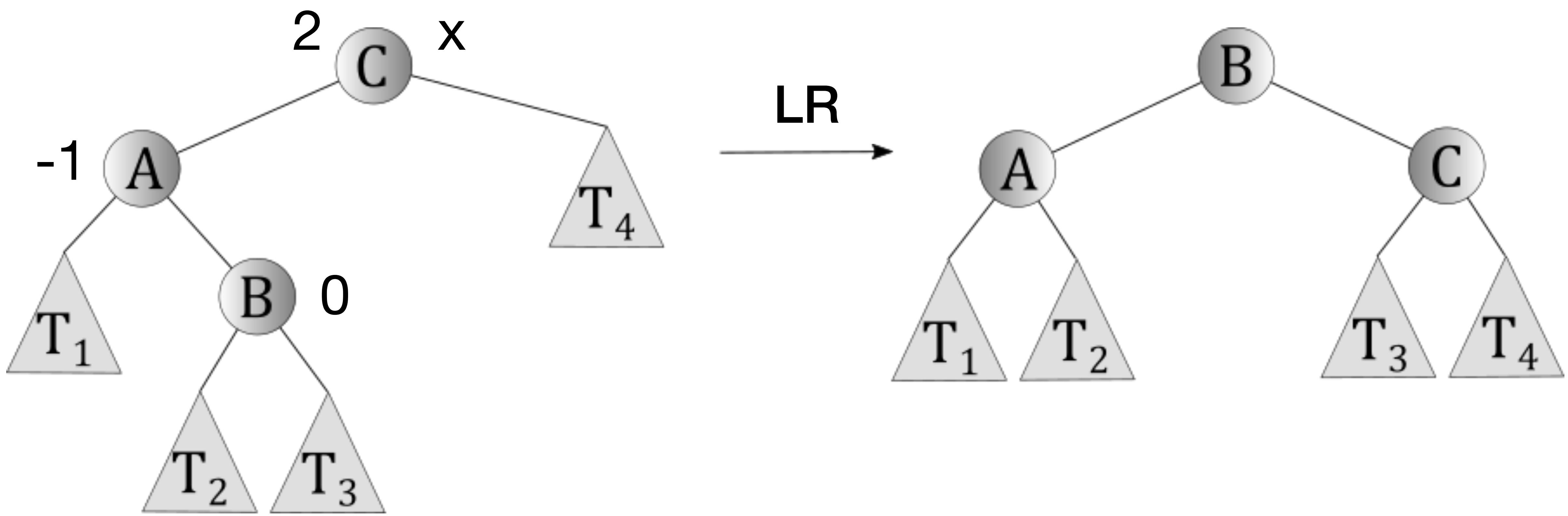
```
/**
 * @post Rotates the given subtree to the right, and
 *   returns the root of the resulting tree.
 */
private Node rotateRight(Node x) {
    Node y = x.left;
    x.left = y.right;
    y.right = x;
    x.height = 1 + Math.max(height(x.left), height(x.right));
    y.height = 1 + Math.max(height(y.left), height(y.right));
    return y;
}
```

Rotación izquierda-derecha

Doble rotación: Primero rotar a la izquierda el subárbol izquierdo (con raíz y), y luego rotar el árbol resultante (con raíz x) a la derecha



Rotación izquierda-derecha



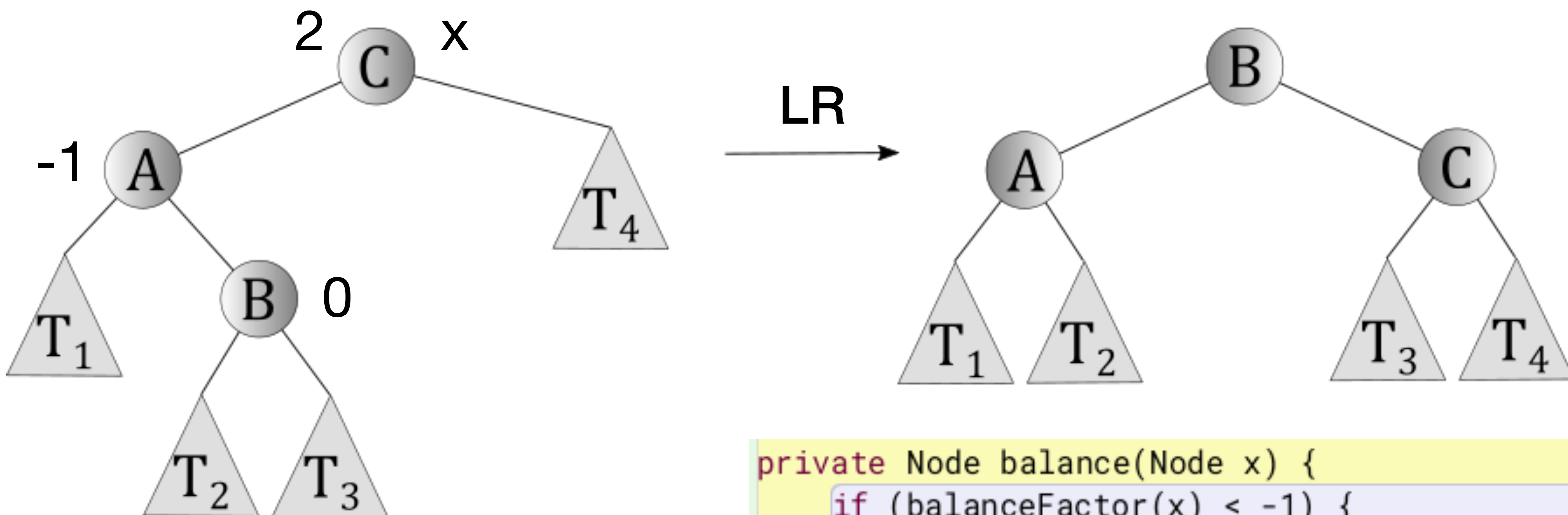
Todos los nodos en T2 > A

Todos los nodos en T3 < C

A < B

- Primero rotar a la izquierda el subárbol izquierdo x.left
- Luego rotar el árbol resultante (con raíz x) a la derecha

Rotación izquierda-derecha



Todos los nodos en T2 > A

Todos los nodos en T3 < C

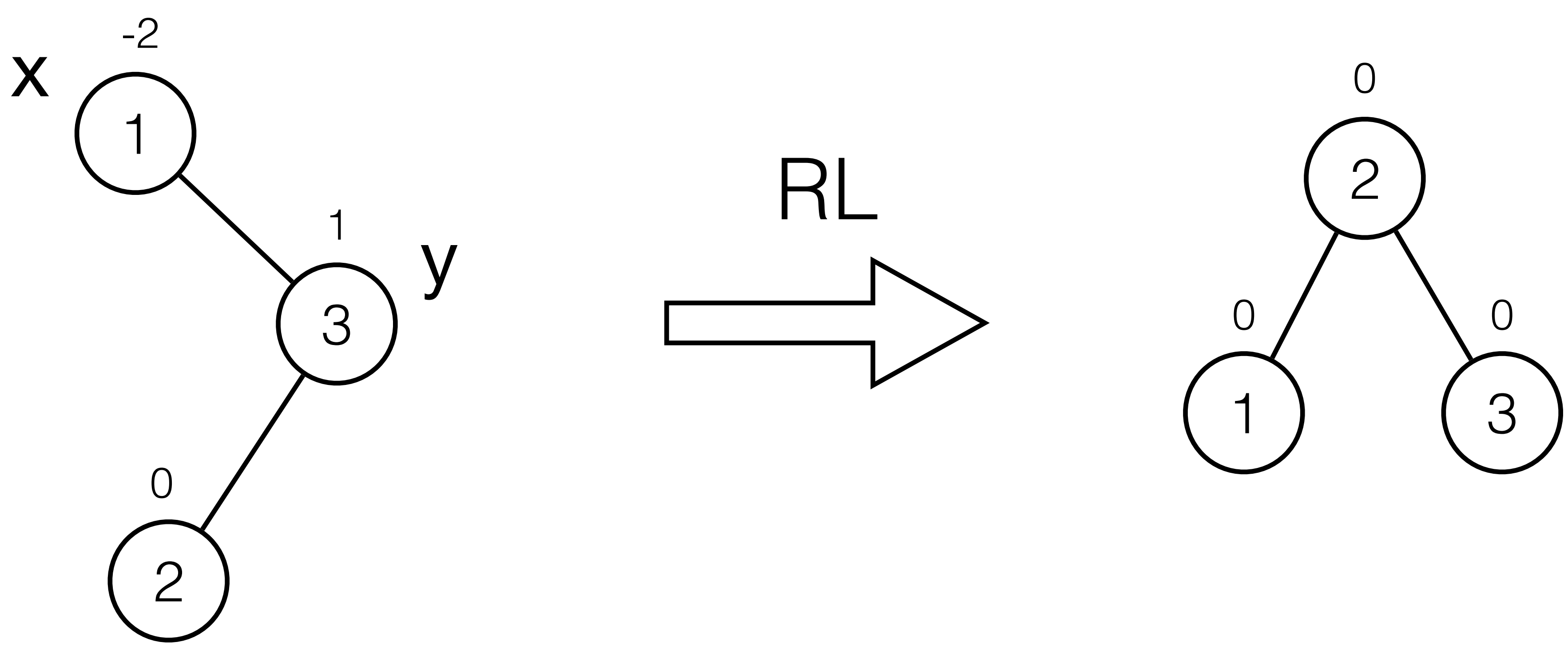
A < B

- Primero rotar a la izquierda el subárbol izquierdo x.left
- Luego rotar el árbol resultante (con raíz x) a la derecha

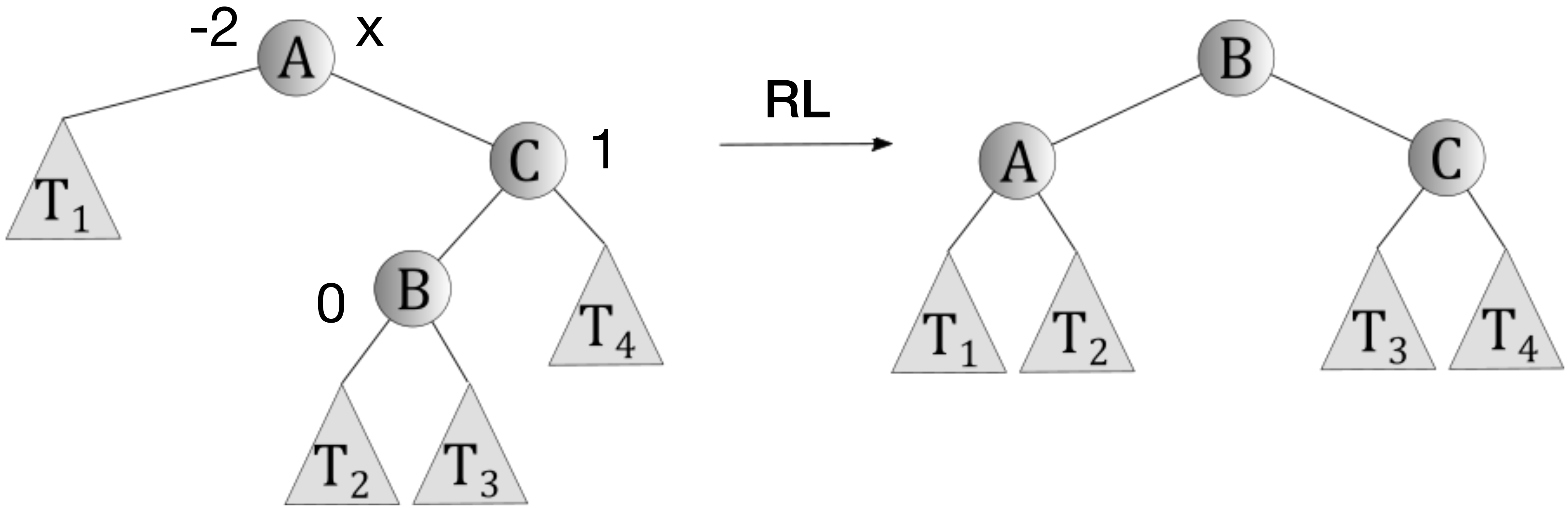
```
private Node balance(Node x) {
    if (balanceFactor(x) < -1) {
        if (balanceFactor(x.right) > 0) {
            x.right = rotateRight(x.right);
        }
        x = rotateLeft(x);
    }
    else if (balanceFactor(x) > 1) {
        if (balanceFactor(x.left) < 0) {
            x.left = rotateLeft(x.left);
        }
        x = rotateRight(x);
    }
    return x;
}
```

Rotación derecha-izquierda

Doble rotación: Primero rotar a la derecha el subárbol derecho (con raíz y), y luego rotar el árbol resultante (con raíz x) a la izquierda



Rotación derecha-izquierda



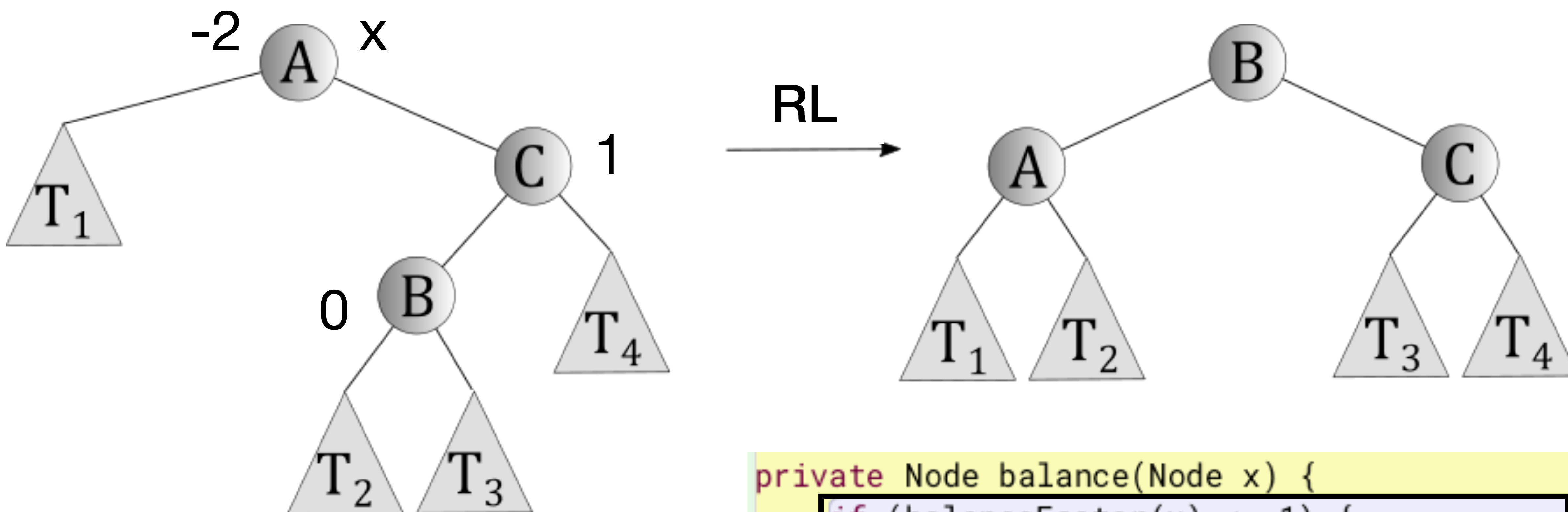
Todos los nodos en T3 < C

Todos los nodos en T2 > A

B < C

- Primero rotar a la derecha el subárbol derecho x.right
- Luego rotar el árbol resultante (con raíz x) a la izquierda

Rotación derecha-izquierda



Todos los nodos en T3 < C

Todos los nodos en T2 > A

B < C

- Primero rotar a la derecha el subárbol derecho x.right
- Luego rotar el árbol resultante (con raíz x) a la izquierda

```java
private Node balance(Node x) {
    if (balanceFactor(x) < -1) {
        if (balanceFactor(x.right) > 0) {
            x.right = rotateRight(x.right);
        }
        x = rotateLeft(x);
    }
    else if (balanceFactor(x) > 1) {
        if (balanceFactor(x.left) < 0) {
            x.left = rotateLeft(x.left);
        }
        x = rotateRight(x);
    }
    return x;
}
```

Inserción en AVLs

- Misma idea que en BST, pero tenemos que actualizar la altura y rebalancear al final

```
/**
 * @post Adds 'key' to the elements of 'this'.
 *   Returns true iff 'key' was added. The
 *   tree is rebalanced after the insertion.
 *   More formally, it satisfies:
 *      result = !(key in old(this)) &&
 *        this = old(this) U {key}.
 */
public boolean add(T key) {
    if (contains(key))
        return false;
    root = add(root, key);
    size++;
    return true;
}
```

Inserción en AVLs

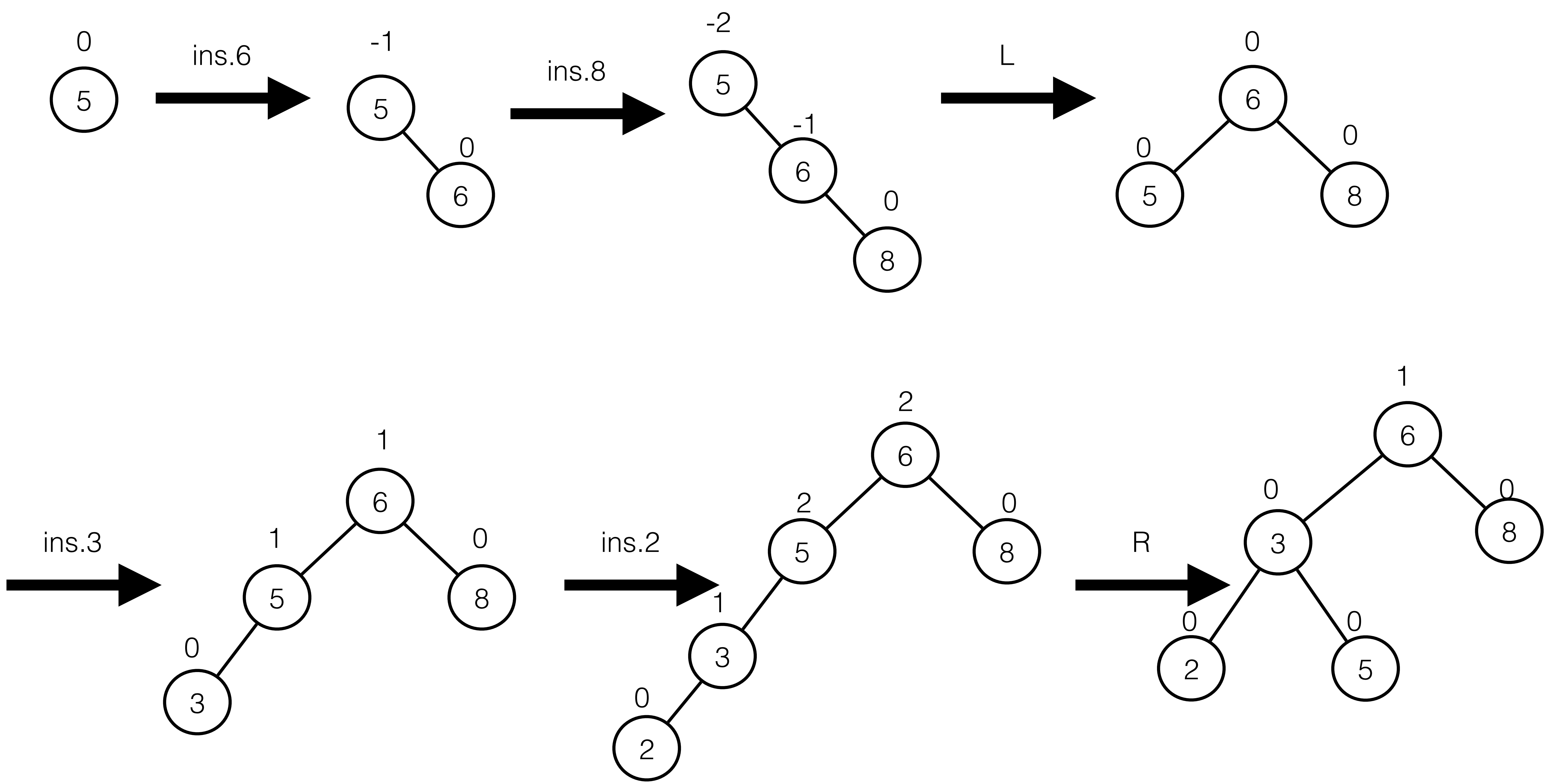
- Misma idea que en BST, pero tenemos que actualizar la altura y rebalancear al final

```java
/**
 * @post Adds 'key' to the elemen
 *   Returns true iff 'key' was a
 *   tree is rebalanced after the
 *   More formally, it satisfies:
 *      result = !(key in old(thi
 *        this = old(this) U {key
 */
public boolean add(T key) {
    if (contains(key))
        return false;
    root = add(root, key);
    size++;
    return true;
}
```

```java
/**
 * @post Inserts 'key' to the tree with root 'x',
 *   and returns the root of the resulting tree. The
 *   tree is rebalanced after the insertion.
 */
private Node add(Node x, T key) {
    if (x == null)
        return new Node(key, 1);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0)
        x.left  = add(x.left, key);
    else if (cmp > 0)
        x.right = add(x.right, key);
    else
        // Should never happen!
        assert false;

    x.height = 1 + Math.max(height(x.left), height(x.right));
    return balance(x);
}
```

Un Ejemplo



0
5
ins.6
-1
5
0
6
ins.8
-2
5
-1
6
0
8
L
0
6
0
5
0
8
ins.3
1
6
1
5
0
8
0
3
ins.2
2
6
2
5
0
8
1
3
0
2
R
1
6
0
3
0
8
0
2
0
5

Eliminar el mínimo

- Para eliminar el mínimo también procedemos como en ABBs, sólo que al final corregimos las alturas y balanceamos el árbol
- Eliminar el máximo es dual
- min y max son idénticas a las operaciones de ABBs

```java
/**
 * @pre !isEmpty()
 * @post Deletes the smallest element of 'this'. The
 *   tree is rebalanced after the removal.
 */
public void removeMin() {
    if (isEmpty())
        throw new NoSuchElementException("Empty tree");
    root = removeMin(root);
    size--;
}
```

```java
/**
 * @post Deletes the smallest element of the tree with root 'x',
 *   and returns the root of the resulting tree. The
 *   tree is rebalanced after the removal.
 */
private Node removeMin(Node x) {
    if (x.left == null)
        return x.right;
    x.left = removeMin(x.left);
    x.height = 1 + Math.max(height(x.left), height(x.right));
    return balance(x);
}
```

Eliminación en AVLs

- Para el borrado de un elemento se procede como en los ABBs,
- Buscamos el nodo a borrar, y reemplazamos por el nodo correspondiente,
- Vamos de abajo hacia arriba corrigiendo desbalances con rotaciones

A lo sumo O(log n) rotaciones

Eliminación en AVLs

- Misma idea que en BST, pero tenemos que actualizar la altura y rebalancear al final

```java
/**
 * @post Removes 'x' from 'this'. Returns
 *   true iff 'x' was removed.  The
 *   tree is rebalanced after the removal.
 *   More formally, it satisfies:
 *      result = (e in old(this)) && this = old(this) \ {e}.
 */
public boolean remove(T key) {
    if (!contains(key))
        return false;
    root = remove(root, key);
    size--;
    return true;
}
```

Eliminación en AVLs

- Misma idea que en BST, rebalancear al final

```
/**
 * @post Removes 'x' from 'this'. Retur
 *   true iff 'x' was removed.  The
 *   tree is rebalanced after the remov
 *   More formally, it satisfies:
 *      result = (e in old(this)) && th
 */
public boolean remove(T key) {
    if (!contains(key))
        return false;
    root = remove(root, key);
    size--;
    return true;
}
```

```
/**
 * @pre key belongs to the tree with root x.
 * @post Removes element key from the tree with root 'x',
 *   and returns the root of the resulting tree. The
 *   tree is rebalanced after the removal.
 */
private Node remove(Node x, T key) {
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left  = remove(x.left,  key);
    else if (cmp > 0)
        x.right = remove(x.right, key);
    else {
        if (x.right == null)
            return x.left;
        if (x.left  == null)
            return x.right;
        Node t = x;
        x = min(t.right);
        x.right = removeMin(t.right);
        x.left = t.left;
    }
    x.height = 1 + Math.max(height(x.left), height(x.right));
    return balance(x);
}
```

Invariante de representación

```java
/**
 * @post Returns true if and only if the structure is a
 *   valid AVL.
*/
public boolean repOK() {
    return isBST(root, null, null) && isAVL();
}
```

```java
/**
 * @post Returns true iff all the keys in the subtree with
 *   root x are larger than min and smaller than max.
 */
private boolean isBST(Node x, T min, T max) {
    if (x == null)
        return true;

    if (min != null && x.key.compareTo(min) <= 0)
        return false;
    if (max != null && x.key.compareTo(max) >= 0)
        return false;

    return isBST(x.left, min, x.key) &&
           isBST(x.right, x.key, max);
}
```

repOK: El árbol tiene que ser un ABB y tiene que ser balanceado

Invariante de representación

```java
/**
 * @post Returns true if and only if the structure is a
 *   valid AVL.
*/
public boolean repOK() {
    return isBST(root, null, null) && isAVL();
}
```

repOK: El árbol tiene que ser un ABB y tiene que ser balanceado

```java
/**
 * @post Returns true iff all the keys in the subt
 *   root x are larger than min and smaller than m
 */
private boolean isBST(Node x, T min, T max) {
    if (x == null)
        return true;

    if (min != null && x.key.compareTo(min) <= 0)
        return false;
    if (max != null && x.key.compareTo(max) >= 0)
        return false;

    return isBST(x.left, min, x.key) &&
           isBST(x.right, x.key, max);
}
```

```java
/**
 * @post Returns true iff the tree satisfies the AVL
 *   balance property.
 */
private boolean isAVL() {
    return isAVL(root);
}
```

```java
/**
 * @post Returns true iff the tree with root x satisfies
 *   the AVL balance property.
 */
private boolean isAVL(Node x) {
    if (x == null)
        return true;
    int bf = balanceFactor(x);
    if (bf > 1 || bf < -1)
        return false;
    return isAVL(x.left) && isAVL(x.right);
}
```

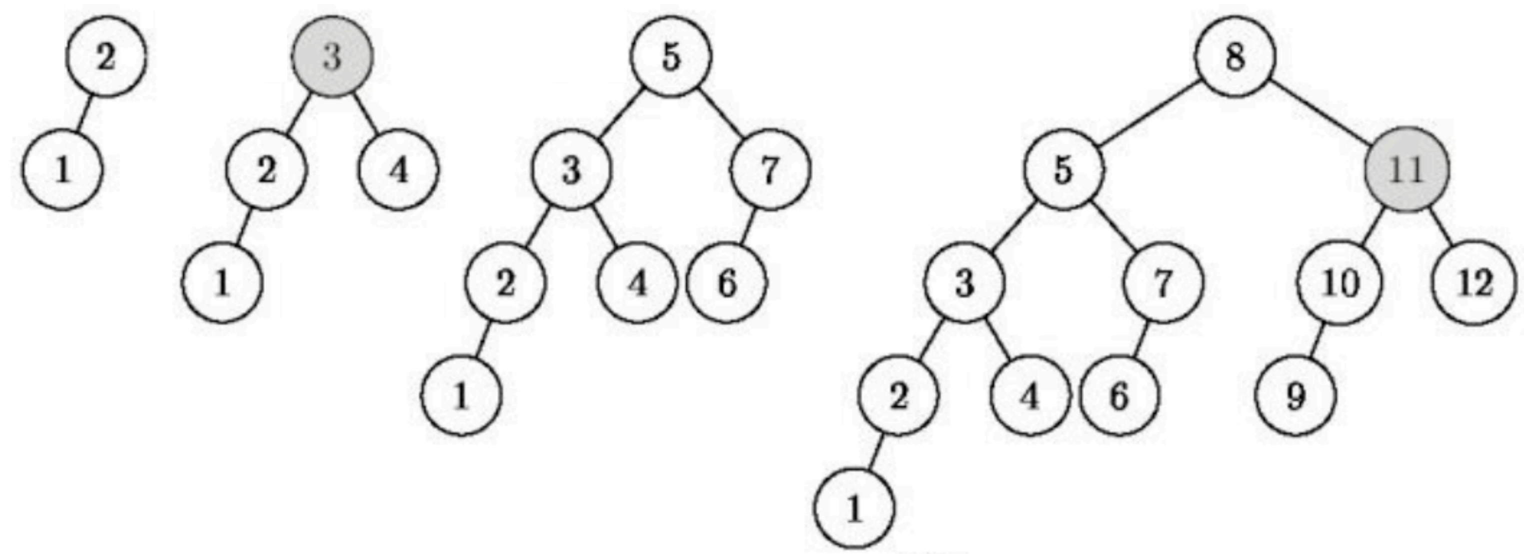
Altura de AVLs

Tenemos el siguiente teorema:

La altura de cualquier AVL es $O(\log n)$, donde n es la cantidad de nodos

Altura de AVLs: Demostración

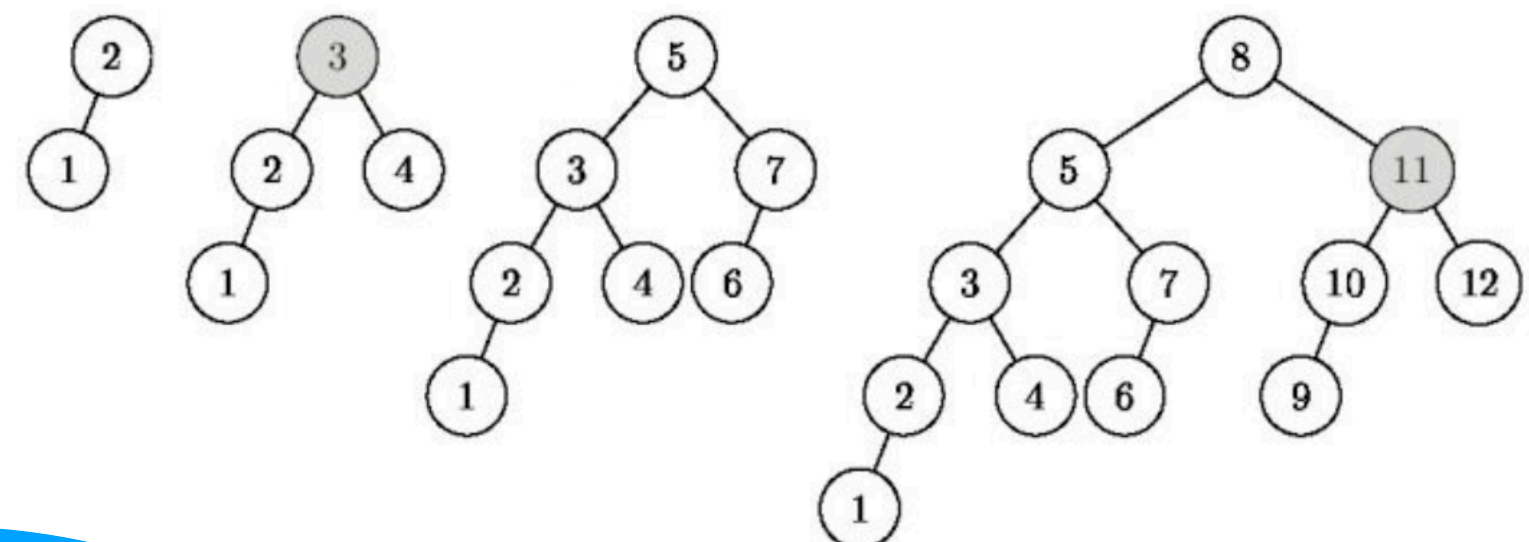
- Sea $N(h)$ la cantidad mínima de nodos en un AVL de altura h
- Los AVLs con mínima cantidad de nodos son el peor caso
  - Todas las alturas de los subárboles difieren en 1 o -1
  - Es decir, están lo más "desbalanceado" que puede estar un AVL
  - Ejemplos:



- Asumamos que el subárbol izquierdo es más grande que el derecho (el caso contrario es idéntico)
- Podemos definir la siguiente ecuación de recurrencia para $N(h)$:
  - $N(1) = 1$
  - $N(h) = 1 + N(h-1) + N(h-2)$

Altura de AVLs: Demostración

- Sea $N(h)$ la cantidad mínima de nodos en un AVL de altura h
- Los AVLs con mínima cantidad de nodos son el peor caso
    - Todas las alturas de los subárboles difieren en 1 o -1
    - Es decir, están lo más "desbalanceado" que puede estar un AVL
    - Ejemplos:



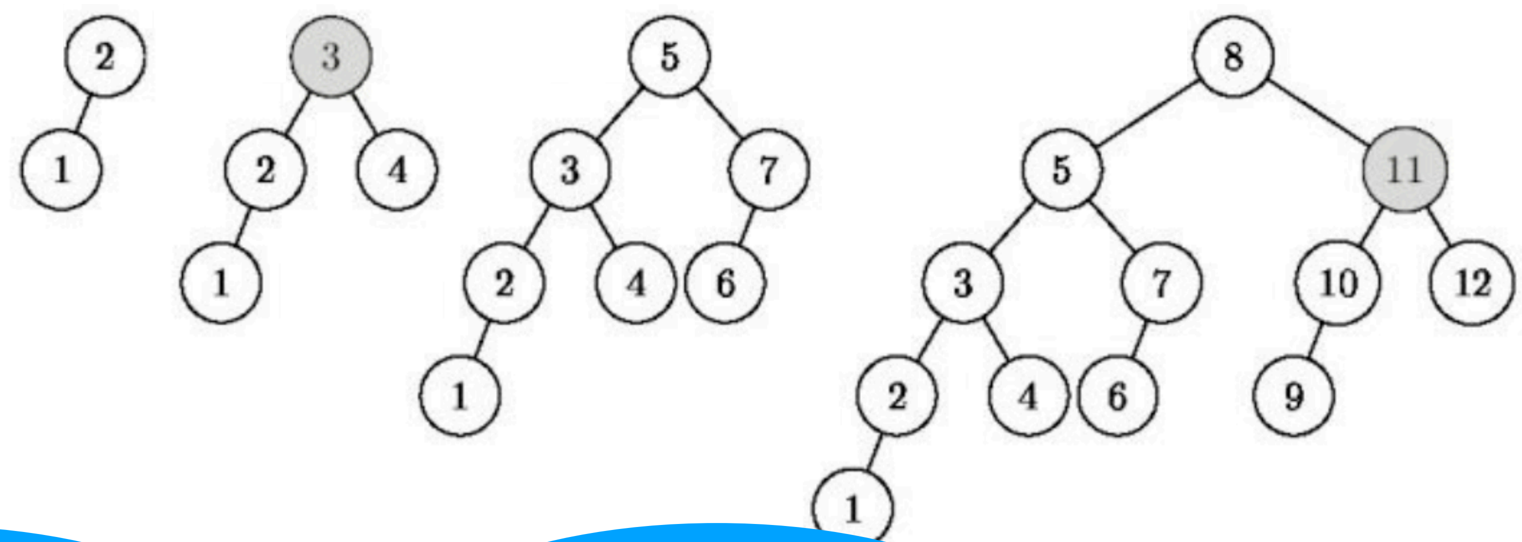
- As[illegible] árbol izquierdo es más grande que el derecho (el caso

cantidad mínima de nodos en el subárbol izquierdo

- [illegible]ente ecuación de recurrencia para $N(h)$:
    - $N(1) = 1$
    - $N(h) = 1 + N(h-1) + N(h-2)$

Altura de AVLs: Demostración

- Sea $N(h)$ la cantidad mínima de nodos en un AVL de altura h
- Los AVLs con mínima cantidad de nodos son el peor caso
  - Todas las alturas de los subárboles difieren en 1 o -1
  - Es decir, están lo más "desbalanceado" que puede estar un AVL
  - Ejemplos:



- As árbol izqu que el derecho (el caso

cantidad mínima de nodos en el subárbol izquierdo

cantidad mínima de nodos en el subárbol derecho

- ente e para $N(h)$:
  - $N(1) = 1$
  - $N(h) = 1 + N(h-1) + N(h-2)$

Altura de AVLs: Demostración

- Resolviendo en $N(h) = 1 + N(h-1) + N(h-2)$
  - $N(h) \geq 1 + 2N(h-2) \geq 2N(h-2)$
- Luego:
  - $N(h) \geq 2 * N(h-2)$
    $\{N(h-2) \geq 2 * N(h-4)\}$
  - $N(h) \geq 2 * 2 * N(h-4)$
    $\{N(h-4) \geq 2 * N(h-6)\}$
  - $N(h) \geq 2 * 2 * 2 * N(h-6)$
    $\{N(h-6) \geq 2 * N(h-8)\}$
  - $N(h) \geq 2 * 2 * 2 * 2 * N(h-8)$
- La forma general es:
  - $N(h) \geq 2^i * N(h - 2 * i)$
- Para que $h - 2 * i = 1$ se debe cumplir que $i = (h-1)/2$. Reemplazando...
  - $N(h) \geq 2^{(h-1)/2} \iff log_2 N(h) \geq log_2 2^{(h-1)/2} \iff 2 * log_2 N(h) + 1 \geq h$
- Por lo tanto, $h \in O(log_2 N(h))$ y como en el caso general $n \geq N(h)$, $h \in O(log_2 n)$

Sobre la eficiencia de las implementaciones de sets y maps

- Como los AVLs tienen altura O(log n), las operaciones de inserción, eliminación y búsqueda son O(log n) en el peor caso
  - Esto se debe a que las rotaciones toman tiempo constante (solo involucran cambios de referencias)
  - Y que a lo sumo se realiza una cantidad logarítmica de rotaciones en las operaciones de insertar y eliminar
- Usando árboles balanceados podemos implementar sets y maps eficientemente: insertar, eliminar y buscar son O(log n) en el peor caso
    - Si usamos listas estas operaciones son O(n) en el peor caso
- Si para nuestro problema no se requiere almacenar elementos repetidos, y tenemos una noción de orden es conveniente usar sets/maps en lugar de listas

Sobre la eficiencia de las implementaciones de sets y maps

- Existen otras formas de implementar sets/maps con árboles balanceados, como por ejemplo, usando Red-Black Trees (RBTs) o árboles 2-3
  - Las operaciones de RBTs tienen el mismo tiempo que las de AVL en el peor caso
  - Los RBTs son más permisivos en términos de balance que los AVLs, por lo que son más rápidos para insertar y eliminar elementos
    - Hacen una cantidad constante de rotaciones en lugar de logarítmica
  - Como los AVLs mantienen un balance más restrictivo, típicamente son árboles de menor altura, y la búsqueda en AVLs suele ser más rápida
  - Los AVLs requieren más memoria porque debemos guardar la altura en cada nodo
  - Las clases java.util.TreeSet y java.util.TreeMap implementan conjuntos y maps, respectivamente, usando Red-Black Trees

Bibliografía

- "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016
- "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009
- "Data Structures and Algorithms". A. Aho, J. Hopcroft, J. Ullman. Addison-Wesley. 1983