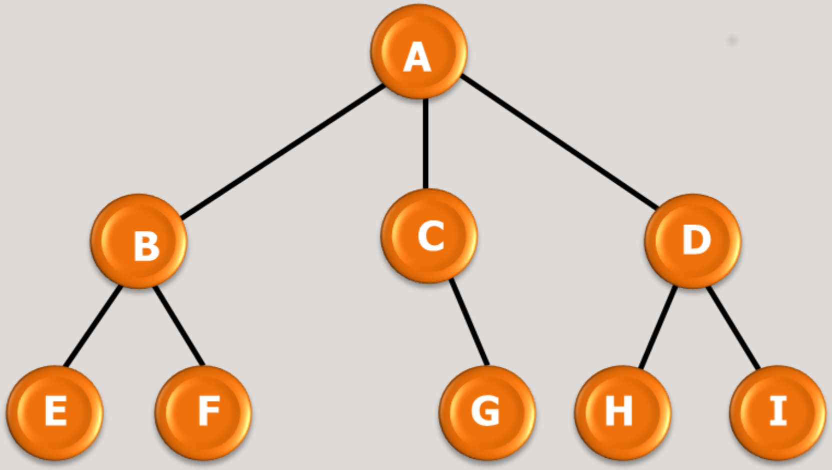
ESTRUCTURAS NO LINEALES

Árboles



Estructuras no Lineales

- Tipos de datos para jerarquías de elementos
  - Árboles
  - Árboles de búsqueda

Árboles (1)

- Definición de **árbol ordenado** (*ordered tree*)
  a) Un nodo, que contiene una información conocida como *etiqueta*. Este nodo es la parte del árbol denominada *raíz*.
  b) Una secuencia (lista) de 0 o más árboles, cuyos nodos raíz son los *hijos* de la raíz del árbol.
    - Cada nodo del árbol es hijo de a lo sumo un nodo, su *padre*.
    - Los hijos de cualquiera de los nodos son *hermanos* unos de otros.

El orden es *posicional*, como en las listas, y no debe confundirse con un *sorted tree* (un árbol de búsqueda)

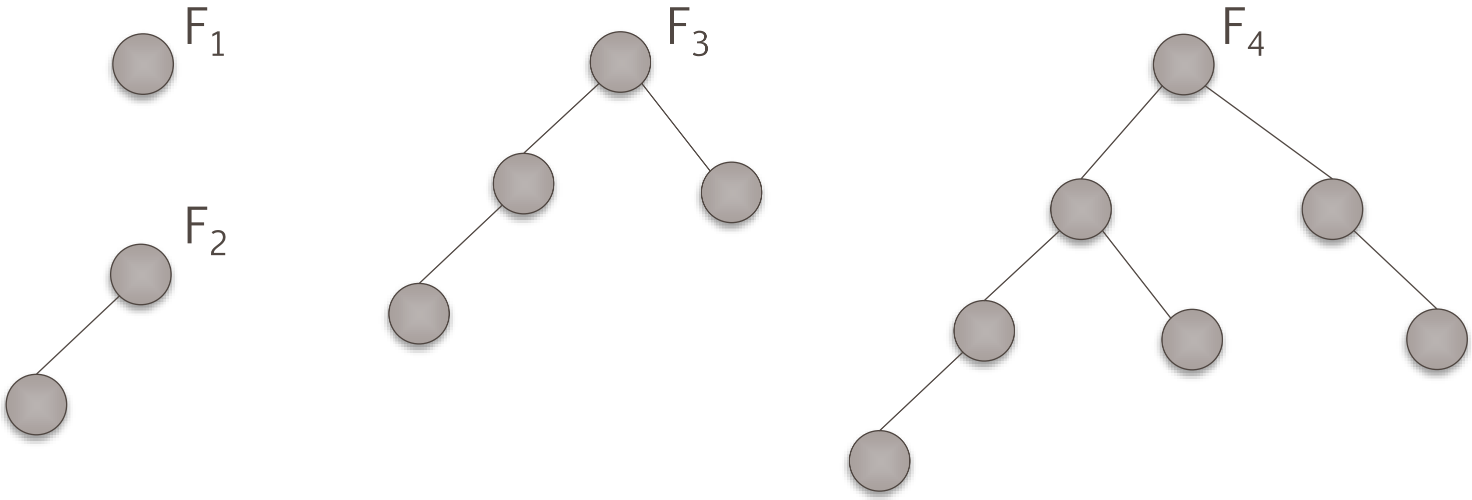
Árboles (2)

- En un árbol **no** ordenado los hijos de un nodo cualquiera constituyen un *conjunto de nodos*, en lugar de una *secuencia* (o lista).
  - En informática es habitual que los árboles estén ordenados y serán éstos los que se verán a continuación.

- Tipos de árboles ordenados
  - Árboles k-arios
  - Árboles de fibonacci
  - Árboles binomiales

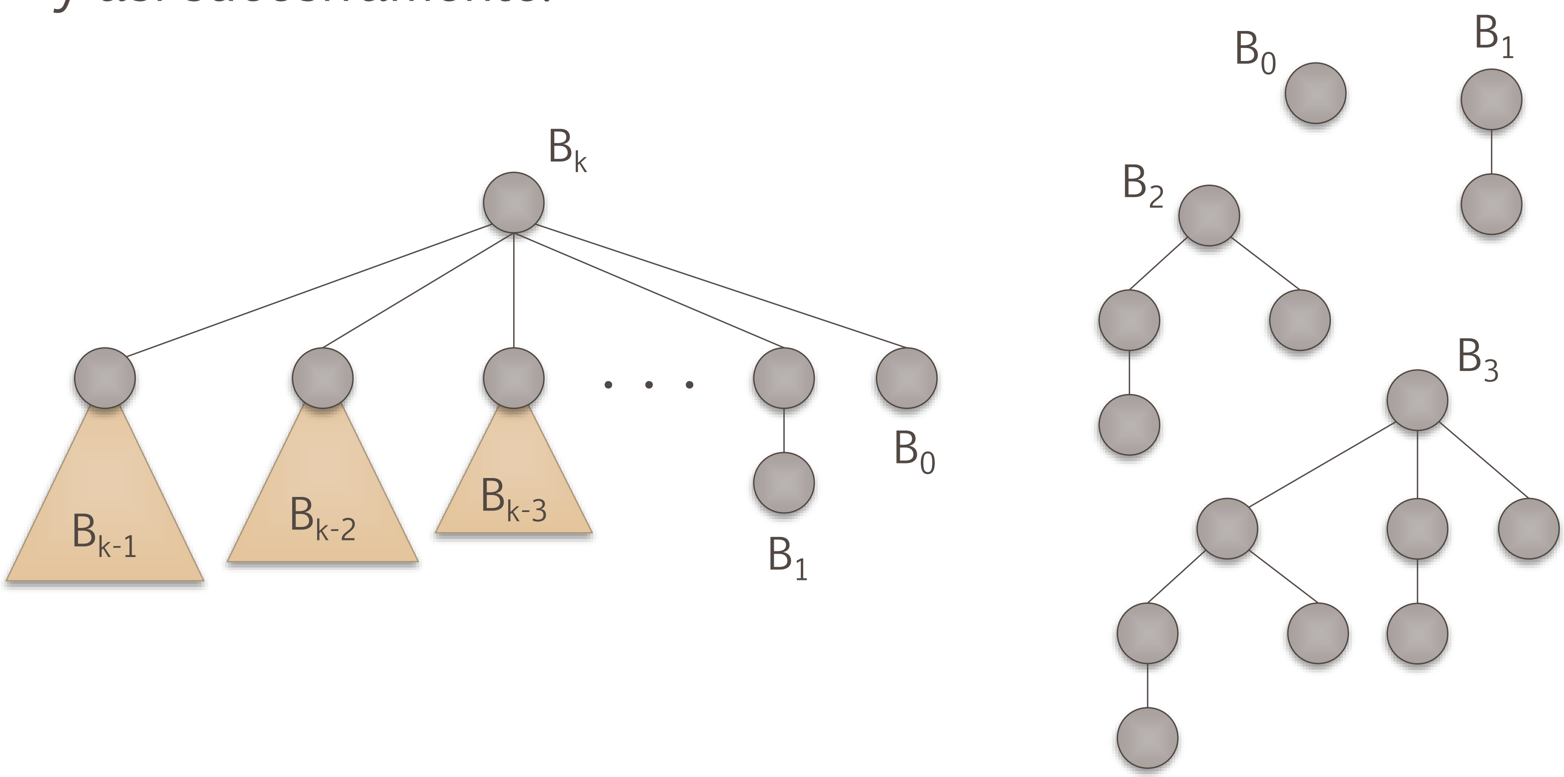
Árboles (3)

- Árboles de Fibonacci
  - F_0 es el árbol vacío
  - F_1 es un árbol con un único nodo
  - F_{k+2} es un nodo cuyo subárbol izquierdo es el árbol F_{k+1} y cuyo subárbol derecho es el árbol F_k



Árboles (4)

- Árboles binomiales
  - El árbol binomial B_k consta de un nodo con *k* hijos. El primer hijo es la raíz de B_{k-1} , el segundo es la raíz de B_{k-2} y así sucesivamente.

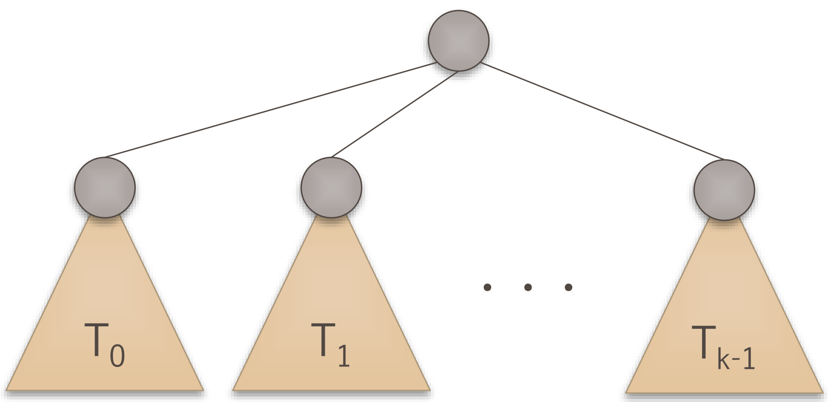


Árboles (5)

- Definición de árbol *k*-ario
  - Un árbol *k*-ario es un árbol en el que los hijos de un nodo ocupan distintas posiciones en el rango 0...*k*-1. Por tanto, el número máximo de hijos para un nodo es *k*.
  - Nombres especiales de algunos árboles *k*-arios:
    - Los árboles 2-ario se denominan **árboles binarios**.
    - Los árboles 3-ario se denominan **árboles ternarios**.
    - Los árboles 1-ario son las **listas**.

Árboles (6)

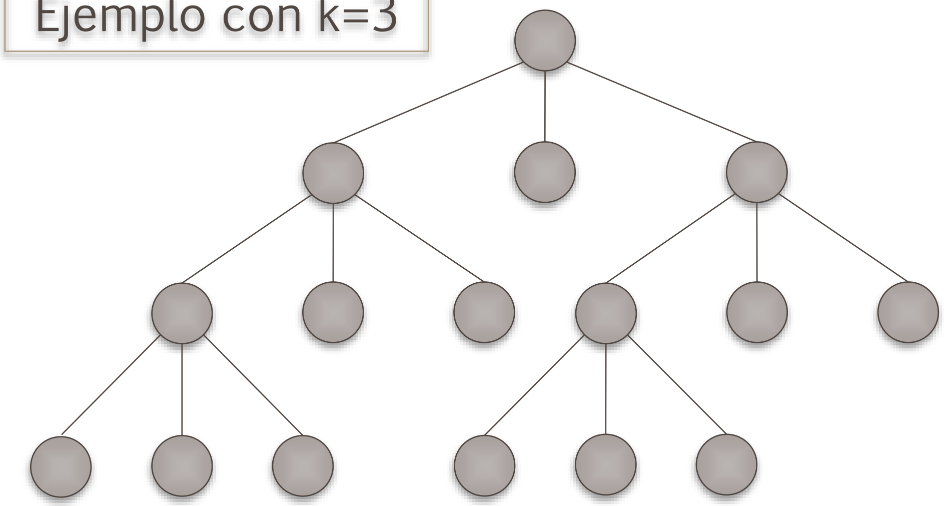
- Definición recurrente de **árbol k-ario**
  a) Un árbol vacío (sin nodos) es un árbo k-ario.
  b) Dado un nodo (generalmente etiquetado) y los árboles k-arios T_0, T_1, …, T_{k-1} (que pueden ser vacíos), el árbol de raíz el nodo dado y que tiene por hijos las raíces de los árboles dados (en las posiciones 0…*k*-1) es un árbol k-ario.



Árboles (7)

- Árboles *k-arios* característicos
  - Árbol *k-ario* **lleno**
    - Todos sus nodos internos son de grado *k* (cada nodo tiene 0 o *k* hijos).

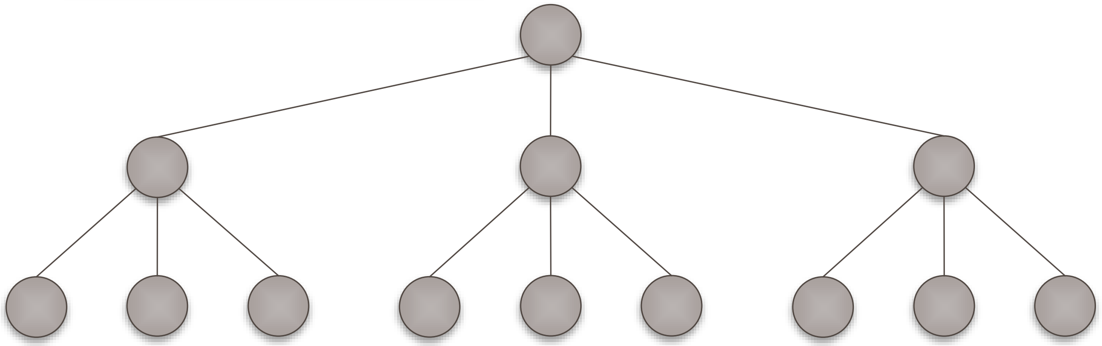
Ejemplo con k=3



Árboles (8)

- Árbol *k-ario* **perfecto**
  - Si está **lleno** y todas sus hojas tienen la misma profundidad (o nivel).

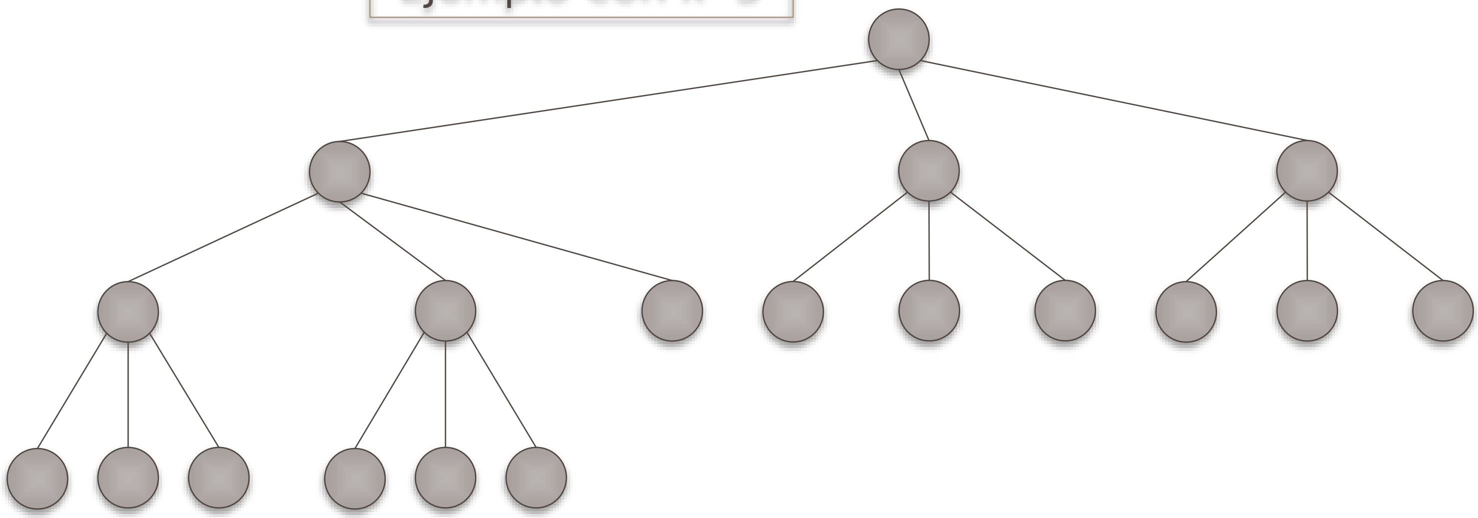
Ejemplo con k=3



Árboles (9)

- Árbol *k*-ario **completo**
  - Si está lleno y los nodos hojas del último nivel se presentan de izquierda a derecha sin huecos.

Ejemplo con k=3



Árboles (10)

- Propiedades de los árboles *k*-arios
  - El número máximo de hojas en un árbol de altura h es k^h
    - k^h es el número de hojas que tiene un árbol perfecto de altura h
  - El número total de nodos en un árbol *k*-ario perfecto de altura h es $(k^{h+1} - 1)/(k - 1)$
    - En cada nivel i $(0 \leq i \leq h)$ tiene k^i nodos y en total:

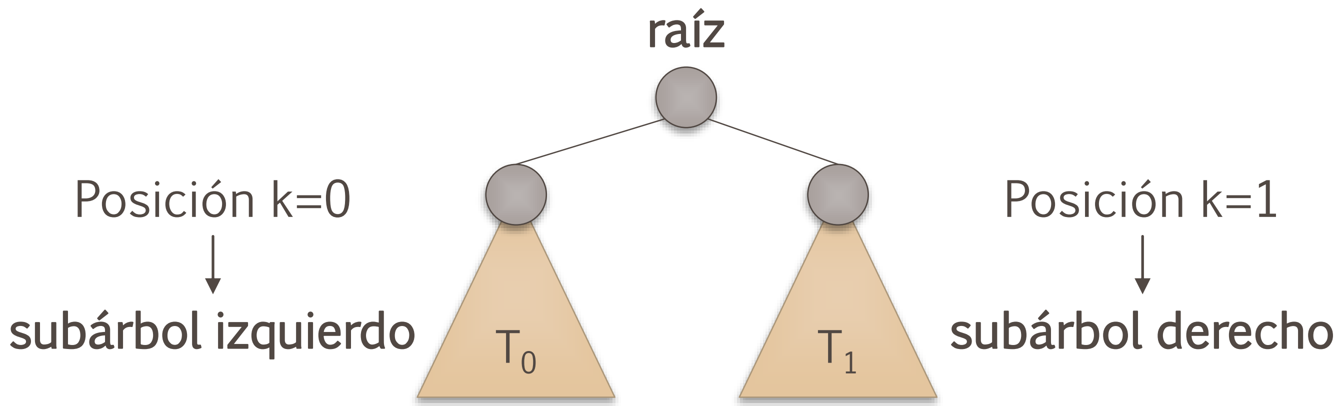
      $$k^0 + k^1 + k^2 + \ldots + k^{h-1} + k^h$$

      Progresión geométrica de razón k y de primer término 1.
  - En un árbol *k*-ario perfecto cuyo número de nodos es n la altura del árbol es

    $$\log_k(k-1) + \log_k(n-1) \Rightarrow O(\log(n))$$

Árboles binarios (1)

- Árbol binario: árbol *2-ario*
  - Es la clase de árbol *k-ario* más utilizada.



No debe confundirse un árbol ordenado cualquiera de grado dos con un árbol binario (en este último, es relevante la posición a la **izquierda** o a la **derecha** del nodo padre).

Árboles binarios (2)

- Interfaz `BinaryTree<E>` (no es de la biblioteca de Java)

```
public interface BinaryTree<E> {
    /**
     * Retorna cierto si este árbol binario es vacío.
     * @return {@code true} si es el árbol vacío
     */
    boolean isEmpty();
    /**
     * Retorna la etiqueta de la raíz de este árbol.
     * @return la etiqueta de la raíz
     * @throws IllegalStateException si este árbol es vacío
     */
    E label();
    /**
     * Retorna el subárbol izquierdo de este árbol.
     * @return el subárbol izquierdo del árbol
     * @throws IllegalStateException si este árbol es vacío
     */
    BinaryTree<E> left();
```

Árboles binarios (3)

```
/**
 * Retorna el subárbol derecho de este árbol.
 * @return el subárbol derecho del árbol
 * @throws IllegalStateException si este árbol es vacío
 */
BinaryTree<E> right();

/**
 * Cambia la etiqueta de la raíz de este árbo por la
 * especificada (operación opcional).
 * @param e la nueva etiqueta de la raíz
 * @throws IllegalStateException si este árbol es vacío
 * @throws NullPointerException si el árbol no admite
 * etiquetas de valor {@code null}
 */
default void setLabel(E e) {
    throw new UnsupportedOperationException();
}
```

Árboles binarios (4)

```java
    /**
     * Cambia el subárbol izquierdo de este árbol por el
     * especificado (operación opcional).
     * @param left el nuevo subárbol izquierdo
     * @throws IllegalStateException si este árbol es vacío
     */
    default void setLeft(BinaryTree<E> left) {
        throw new UnsupportedOperationException();
    }
    /**
     * Cambia el subárbol derecho de este árbol por el
     * especificado (operación opcional).
     * @param right el nuevo subárbol derecho
     * @throws IllegalStateException si este árbol es vacío
     */
    default void setRight(BinaryTree<E> right) {
        throw new UnsupportedOperationException();
    }
}
```

Árboles binarios (5)

- Ejemplo de uso
  - Número de nodos de un árbol binario

```
/**
 * Retorna el número de nodos del árbol binario especificado.
 * @param <E> el tipo de las etiquetas de los nodos del árbol
 * @param bt el árbol binario
 * @return el número de nodos del árbol binario
 */
public static <E> int numNodos(BinaryTree<E> bt) {
    if (bt.isEmpty()) {
        return 0;
    }

    return 1 + numNodos(bt.left()) + numNodos(bt.right());
}
```

Árboles binarios (6)

- Tipo de dato `BinaryTreeImp<E>`
  - Implementa la interfaz `BinaryTree<E>`
  - Representación

```
private E label;                    // etiqueta del nodo
private BinaryTree<E> left;         // subárbol izquierdo
private BinaryTree <E> right;       // subárbol derecho
```

    - Adicionalmente, se podría incluir una referencia al nodo padre para soportar de forma eficiente la operación que retorna el padre de un nodo (`parent()`)

Árboles binarios (7)

- Representación para árboles binarios completos
  - Una alternativa para este tipo de árboles es utilizar un *array* de etiquetas de tipo `E` (parámetro de tipo), de la forma siguiente:
    - Un nodo i si tiene hijo izquierdo está en $2 * i + 1$ y si tiene hijo derecho en $2 * i + 2$.
    - La raíz del árbol binario está en la posición 0. Para el resto de nodos si i es la posición del *array* que le corresponde entonces su padre está en $(i - 1)/2$.
  - Esta representación es extensible a los árboles k-arios completos.
    - Los hijos de un nodo i estarán entre las posiciones

$$k * i + 1 \quad \text{y} \quad k * (i + 1)$$

Árboles binarios. Recorrido (1)

- Formas de recorrido exhaustivo de árboles binarios

Recorrido en profundidad (recurrente)	Recorrido en anchura (iterativo)
preorden	o por niveles
inorden	
postorden	

- Recorridos en profundidad
  - **Preorden**
    - Se visita la raíz del árbol
    - Se recorre en *preorden* el subárbol izquierdo
    - Se recorre en *preorden* el subárbol derecho

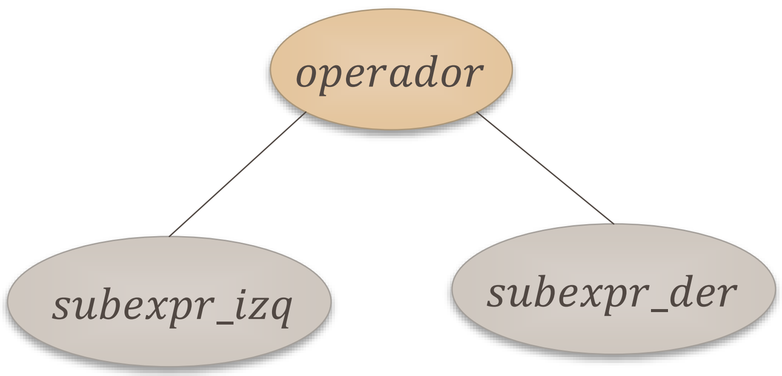
Árboles binarios. Recorrido (2)

- Inorden
  - Se recorre en *inorden* el subárbol izquierdo
  - Se visita la raíz del árbol
  - Se recorre en *inorden* el subárbol derecho
- Postorden
  - Se recorre en *postorden* el subárbol izquierdo
  - Se recorre en *postorden* de subárbol derecho
  - Se visita la raíz del árbol

Para los recorridos en profundidad, resulta bastante más sencillo implementar un *iterador interno* recursivo que implementar un *iterador externo*. Ya que para este último habría que disponer, necesariamente, de un algoritmo iterativo equivalente al recursivo y resultaría bastante más complejo.

Árboles binarios. Recorrido (3)

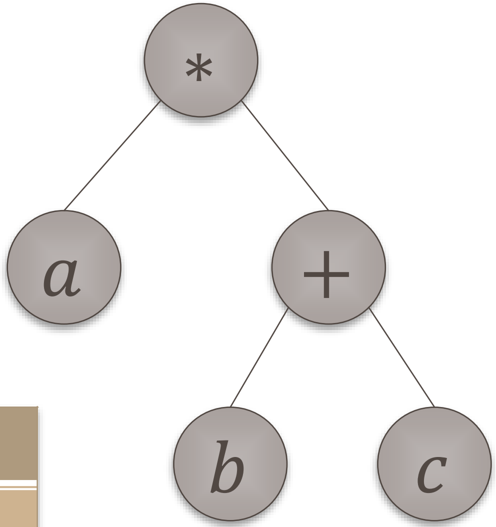
- Relación con la notación de expresiones aritméticas
  - Cualquier expresión: *subexpr_izq operador subexpr_der* se puede representar mediante el árbol



  - y los recorridos en profundidad se corresponden con las tres posibles formas de escribir la expresión (se asume que todos los operadores son binarios)

Árboles binarios. Recorrido (4)

Las notaciones polacas no requieren del uso de paréntesis



Recorrido	Notación	Expresión
Inorden	Infija	$a * (b + c)$
Preorden	Polaca o prefija	$* \; a + b \; c$
Postorden	Polaca inversa o postfija	$a \; b \; c + *$

Árboles binarios. Recorrido (5)

- Se pasa una **acción** a realizar con la etiqueta de cada nodo
- Se pasa un **filtro**: condición que ha de cumplir la etiqueta para aplicar la acción

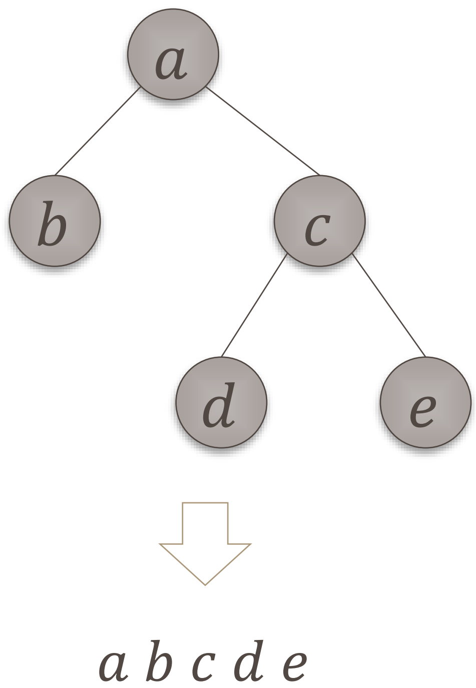
Iterador interno

```
static <E> void inorder (BinaryTree<E> bt,
                         Consumer<? super E> action,
                         Predicate<? super E> filter) {
    if (!bt.isEmpty()) {
        inorder(bt.left(), action, filter);
        E e = bt.label();

        if (filter.test(e)) {
            action.accept(e);
        }

        inorder(bt.right(), action, filter);
    }
}
```

Para realizar la acción con todos los nodos, se indica el filtro (lambda-expr):
`e -> true`

Árboles binarios. Recorrido (6)

- Recorrido en anchura (o por niveles)
  - Los nodos se visitan del primer nivel al último nivel y, en cada nivel, de izquierda a derecha.



- Se requiere un almacenamiento auxiliar, una cola FIFO.
  - Inicialmente se añade a la cola la raíz
  - Nodos en cola visitando el nivel `i`:
    - Todos los nodos no visitados del nivel `i`, de izquierda a derecha
    - A continuación, los hijos de los nodos visitados del nivel `i` de izquierda a derecha

Árboles binarios. Recorrido (7)

- Algoritmo de recorrido en anchura:

hasNext()

```
Añadir la raíz del árbol a la cola;
mientras la cola no esté vacía hacer
    e = elemento extraído de la cola;
    visitar el elemento e;
    añadir a la cola los hijos de e;
fin_mientras
```

next() ⇨ e

A diferencia de los recorridos en profundidad, el recorrido por niveles es intrínsicamente iterativo, por lo que puede obtenerse un *iterador externo* o un *iterador interno* con una dificultad análoga.

Árboles ordenados (1)

- Interfaz `Tree<E>` (no es de la biblioteca de Java)

```
public interface Tree<E> {
    /**
     * Retorna cierto si la raíz de este árbol es una hoja
     * @return {@code true} si la raíz de este árbol es una hoja
     */
    boolean isLeaf();
    /**
     * Retorna la etiqueta de la raíz de este árbol.
     * @return la raíz del árbol
     */
    E label();


    /**
     * Iterador de los subárboles hijos de la raíz de este árbol.
     * @return un iterador de los nodos hijos de la raíz de este árbol
     */
    ChildrenIterator<Tree<E>> childrenIterator();
```

Árboles ordenados (2)

```java
/**
 * Cambia la etiqueta de la raíz de este árbol (operación opcional)
 * @param e la nueva etiqueta de la raíz
 * @throws UnsupportedOperationException si la operación no está
 * soportada por este árbol.
 * @throws NullPointerException si el árbol no admite etiquetas
 * de valor {@code null}
 */
default void setLabel(E e) {
    throw new UnsupportedOperationException();
}

}
```

Árboles ordenados (3)

- Interfaz `ChildrenIterator<E>`

```
public interface ChildrenIterator<E> extends Iterator<E> {

    /**
     * Reemplaza el último elemento retornado por {@code next()} por
     * el elemento especificado (operación opcional).
     * @param e el elemeto de reemplazo
     * @throws IllegalStateException si no se ha llamado a {@code next()}
     * o se ha reemplazado o borrado o añadido un elemento después de la
     * última llamada a {@code next()}
     * @throws NullPointerException si el árbol no admite etiquetas
     * de valor {@code null}
     */
     default void set(E e) {
         throw new UnsupportedOperationException();
     }
```

Árboles ordenados (4)

```
    /**
     * Añade el elemento especificado antes del elemento que
     * proporcionará la operación {@code next()} (operación opcional).
     * @param e el elemeto a añadir
     * @throws NullPointerException si el árbol no admite etiquetas
     * de valor {@code null}
     */
     default void add(E e) {
         throw new UnsupportedOperationException();
     }

}
```

Árboles ordenados (5)

- Ejemplo de uso
  - Camino de la raíz a un nodo con la etiqueta dada

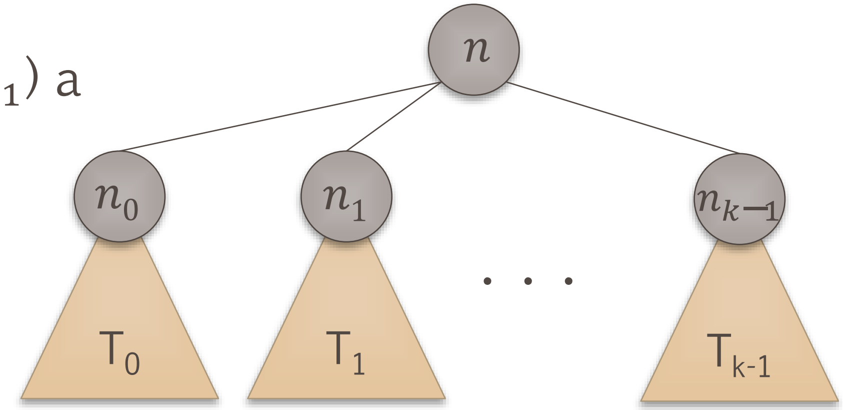
```
public static <E> List<E> camino(Tree<E> t, E e) {
    List<E> l = new LinkedList<>();
    ChildrenIterator<Tree<E>> itr = t.childrenIterator();

    while (itr.hasNext() && l.isEmpty()) {
        l = camino(itr.next(), e);
    }

    if (l.isEmpty()) { // ¿ t.label() = e ?
        if (t.label().equals(e)) {
            l.add(e);
        }
    } else { // añadir la etiqueta de la raíz al principio
        l.add(0, t.label());
    }

    return l;

}
```

Árboles ordenados (6)

- Tipo de dato `LCRSTree<E>`
  - Implementa la interfaz `Tree<E>`
  - Representación basada en árboles binarios: representación *Left-Child, Right Sibling* (LCRS).

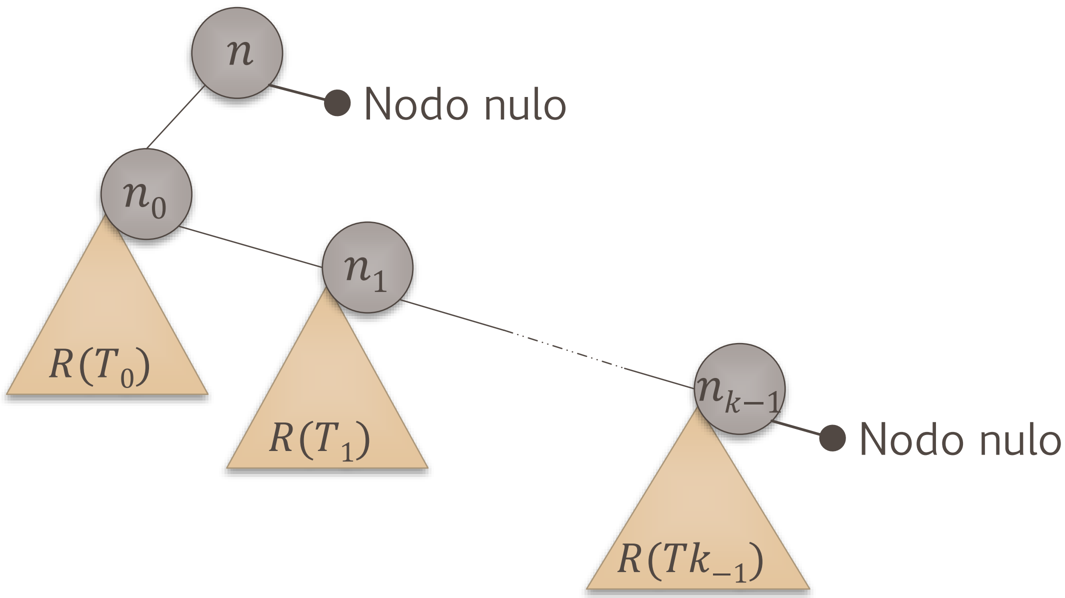
```
private BinaryTree<E> theRoot;    // árbol binario
```

  - Primer hijo de la raíz, n_0, a la izquierda de la raíz n
  - Resto de hijos $(n_1, \ldots, n_{k-1})$ a la derecha de n_0



Árboles ordenados (7)

- Representación LCRS



  - $R(Ti)$ es la representación LCRS del subárbol T_i de raíz n_i $(0 \leq i < k)$

Árboles ordenados. Recorridos (1)

- Recorrido exhaustivo
  - Análogo al ya visto para árboles binarios, tanto en profundidad (*preorden*, *inorden* y *postorden*) como en anchura (o por niveles)
    - Aclaración sobre el recorrido en **inorden**:
      - Primero se visita en *inorden* el subárbol T_0 cuya raíz es el primer hijo de la raíz del árbol (n_0)
      - Se visita la raíz del árbol (n)
      - Por último, se visitan en *inorden* los subárboles $T_1, \ldots, T_{k-1}$ cuyas raíces son los hermanos de n_0 ($n_1, \ldots, n_{k-1}$ respectivamente).

Árboles ordenados. Recorrido (2)

```
static <E> void levelorder (Tree<E> tree,
                            Consumer<? super E> action,
                            Predicate<? super E> filter) {
    Queue<Tree<E>> queue = new LinkedList<>();
    // añadir el árbol a la cola
    queue.add(tree);
    while (!queue.isEmpty()) {
        Tree<E> current = queue.remove();
        if (filter.test(current.label())) {
            action.accept(current.label());
        }
        ChildrenIterator<Tree<E>> itr =
            current.childrenIterator();
        while (itr.hasNext()) { queue.add(itr.next()); }
    }
}
```

Iterador interno

Para realizar la acción con todos los nodos, se indica el filtro (lambda-expr):

`e -> true`

Árboles Binarios de Búsqueda (ABB-1)

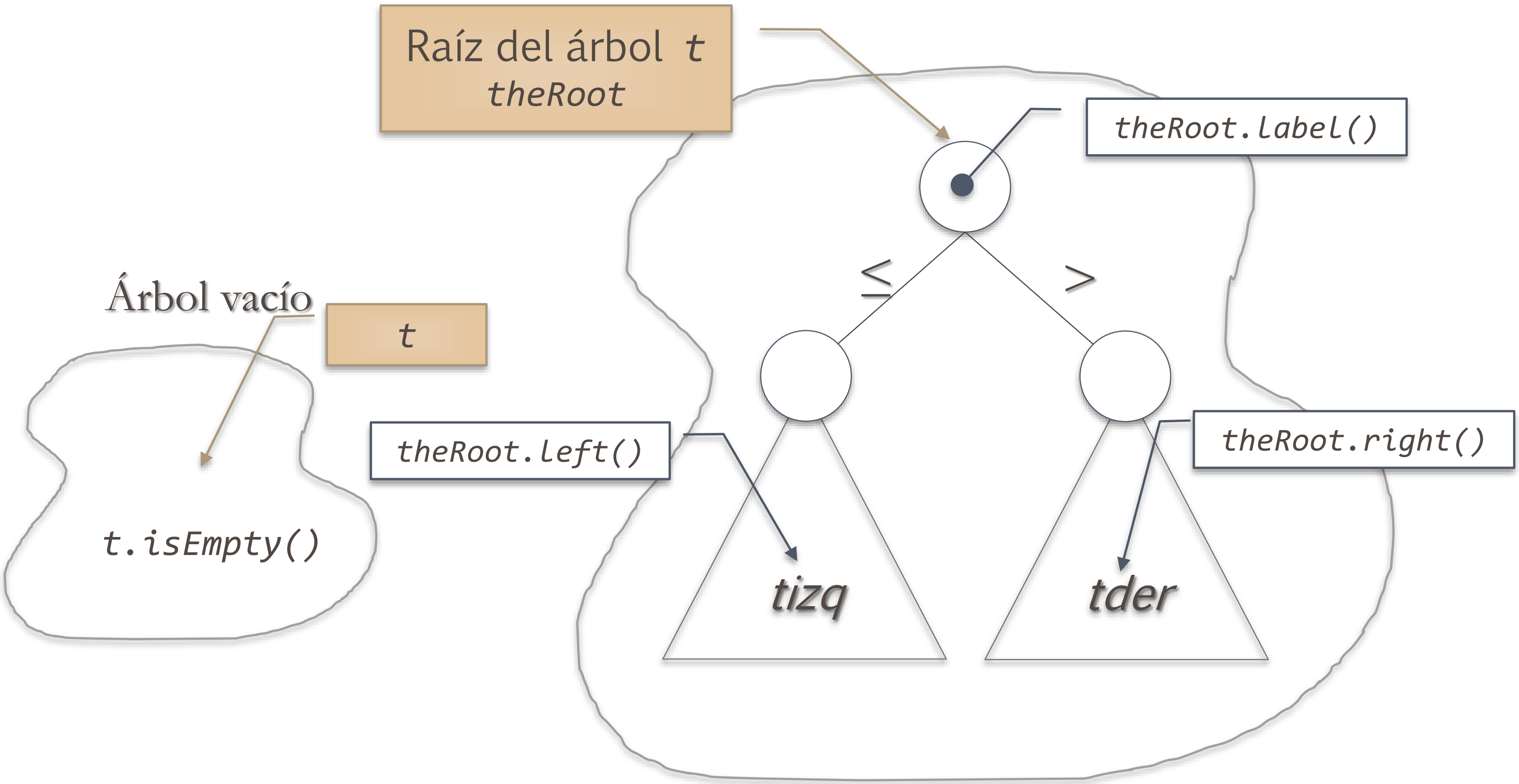
- ABB
  - Un árbol binario de búsqueda es una **colección de elementos** jerarquizada según el orden entre éstos.
    - Se requiere que los elementos se puedan comparar
  - Operaciones principales:
    - Inserción (`add(e)`)
    - Extracción (`remove(o)`)
    - Búsqueda (`contains(o)`)

A diferencia de los árboles ordenados, los ABB son colecciones de elementos con las operaciones habituales para colecciones.

Árboles Binarios de Búsqueda (ABB-2)

```
public class LinkedBST<E> extends AbstractCollection<E> {
    private BinaryTreee<E> theRoot;      // árbol binario
    private Comparator<? super E> cmp;  // comparador
    private int size;                    // num. elementos

    public LinkedBST() {
        theRoot = new BinaryTreeImp<>();
        cmp = null;
        size = 0;
    }

    public LinkedBST(Comparator<? super E> cmp) {
        this();
        this.cmp = cmp;
    }

    ...
```

Árbol Binario de Búsqueda (ABB-3)



Árbol Binario de Búsqueda (ABB-4)

- Función de comparación

Se puede prescindir del *if* y la ejecución dará la excepción igualmente si *a* no es de tipo Comparable<E>

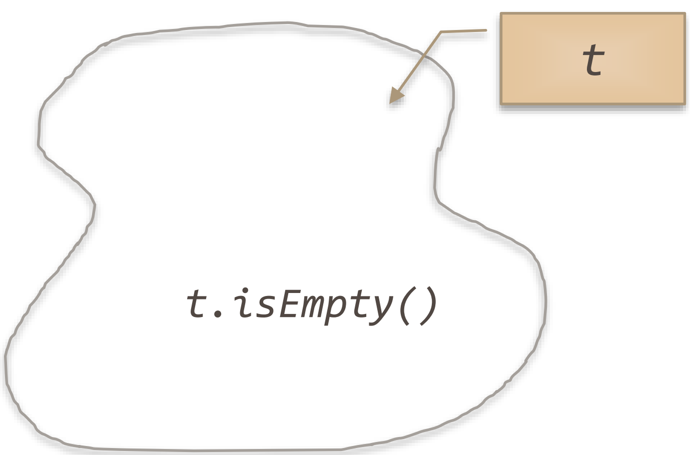
Por defecto (si no se proporciona un comparador), se asume que el tipo E es comparable.

```
public int compare(E a, E b) {
    if (cmp == null && !(a instanceof Comparable<?>)) {
        throw new ClassCastException(
            String.format("El tipo %s no es comparable.\n",
                               a.getClass().getName()));
    }

    return cmp == null ? ((Comparable<? super E>) a).compareTo((E) b)
                       : cmp.compare(a, b);
}
```

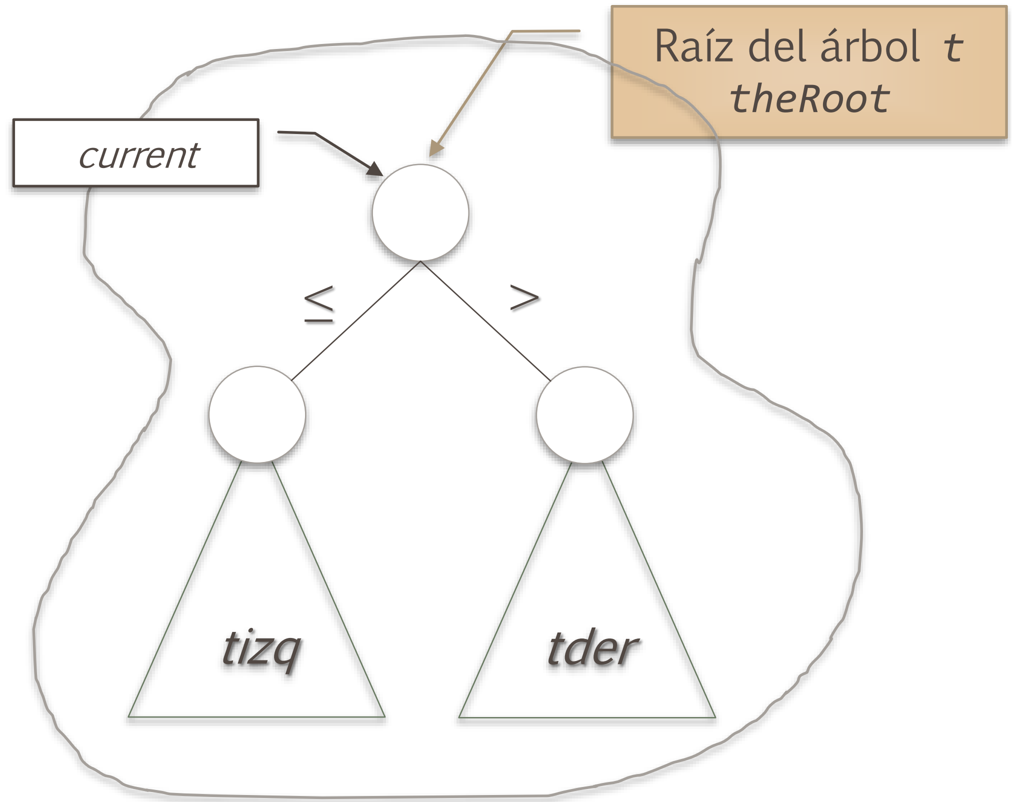
ABB. Operación de búsqueda (ABB-5)

t.contains(o)

Árbol vacío



return *false*



return ¿?

ABB. Operación de búsqueda (ABB-6)

```
x = compare(o, current.label())

x == 0 → etiqueta encontrada
x < 0 → current = current.left()
x > 0 → current = current.right()
```

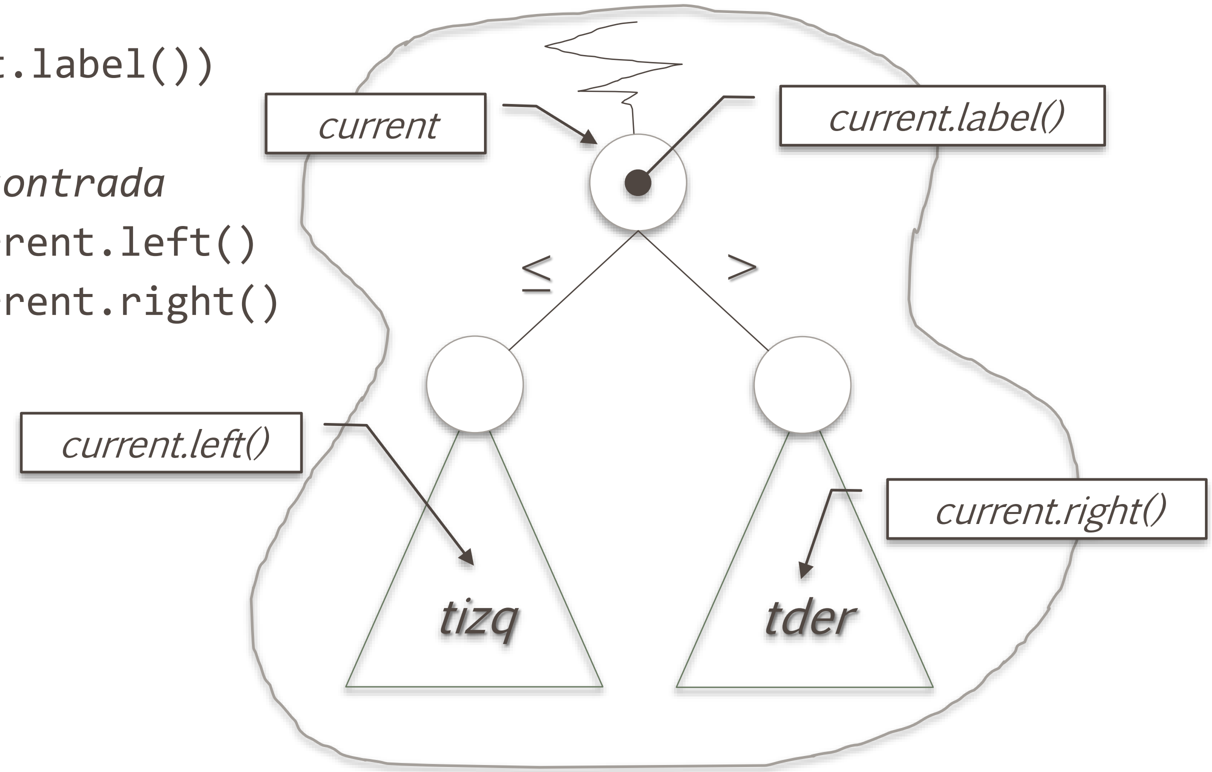


ABB. Operación de búsqueda (ABB-7)

```
public boolean contains(Object o) {
    BinaryTree<E> current = theRoot;
    while (!current.isEmpty()) {
        int x = compare(o, current.label);
        if (x == 0) {
            return true;
        }
        if (x < 0) {  // subárbol izquierdo
            current = current.left();
        } else {      // subárbol derecho
            current = current.right();
        }
    }
    return false;
}
```

ABB. Operación de inserción (ABB-8)

t.add(e)

- Si el árbol está vacío se crea el nodo raíz de etiqueta `e`.
- En caso contrario, se busca la posición de inserción y se añade una nueva hoja al ABB de etiqueta `e`.
  - La búsqueda se realiza de manera análoga a la ya vista para la operación `contains`, partiendo de la raíz del árbol y buscando un nodo (`current`) que cumpla lo siguiente:

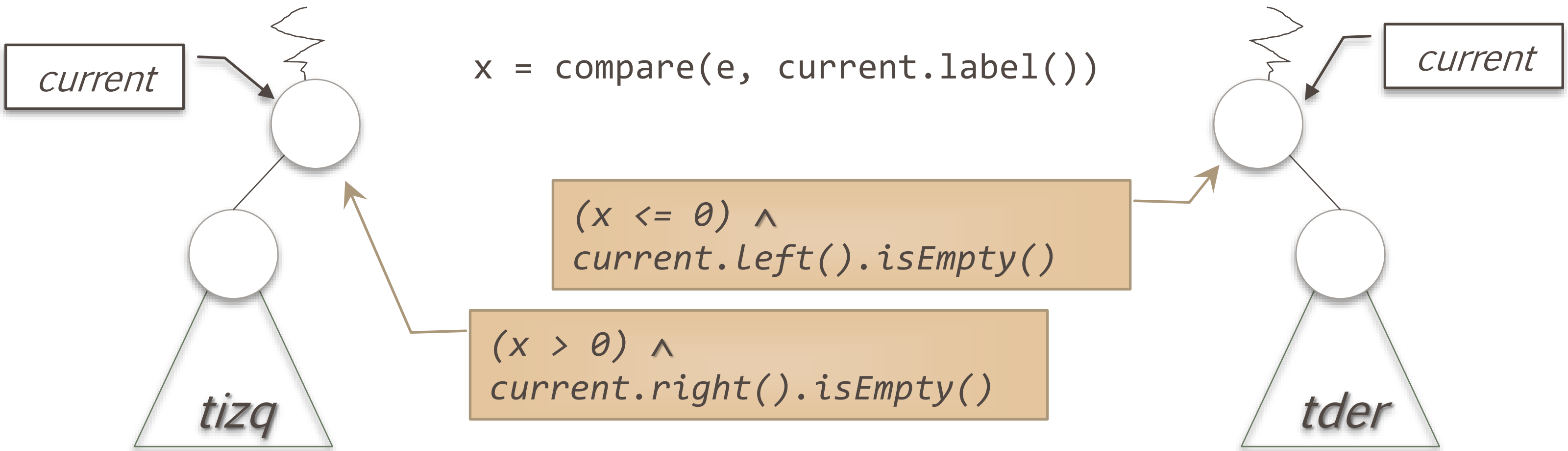
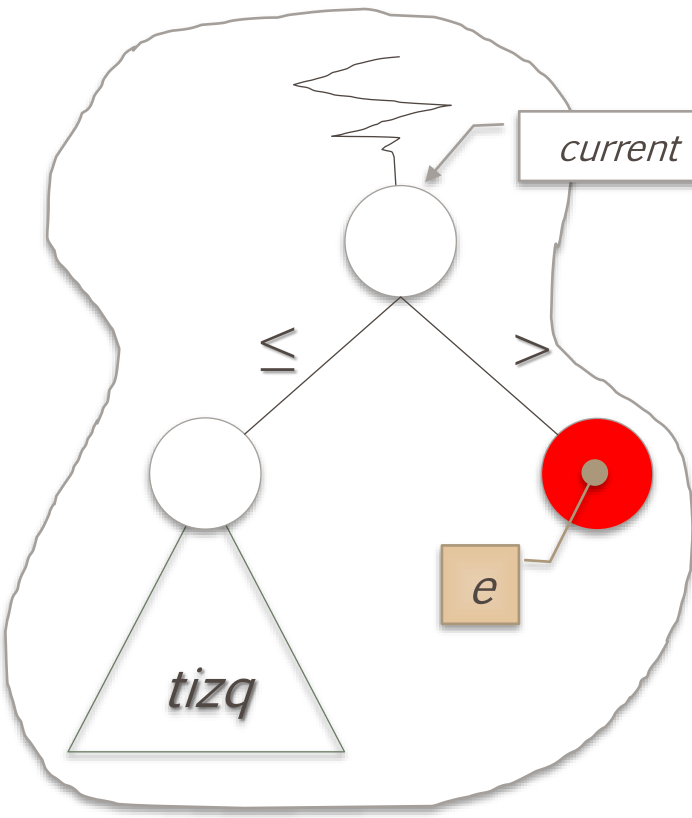


ABB. Operación de inserción (ABB-9)

```
(x > 0) ∧ current.right().isEmpty()
→ current.setRight(
        new BinaryTreeImp<>(e))
```




```
(x <= 0) ∧ current.left().isEmpty()
→ current.setLeft(
        new BinaryTreeImp<>(e))
```

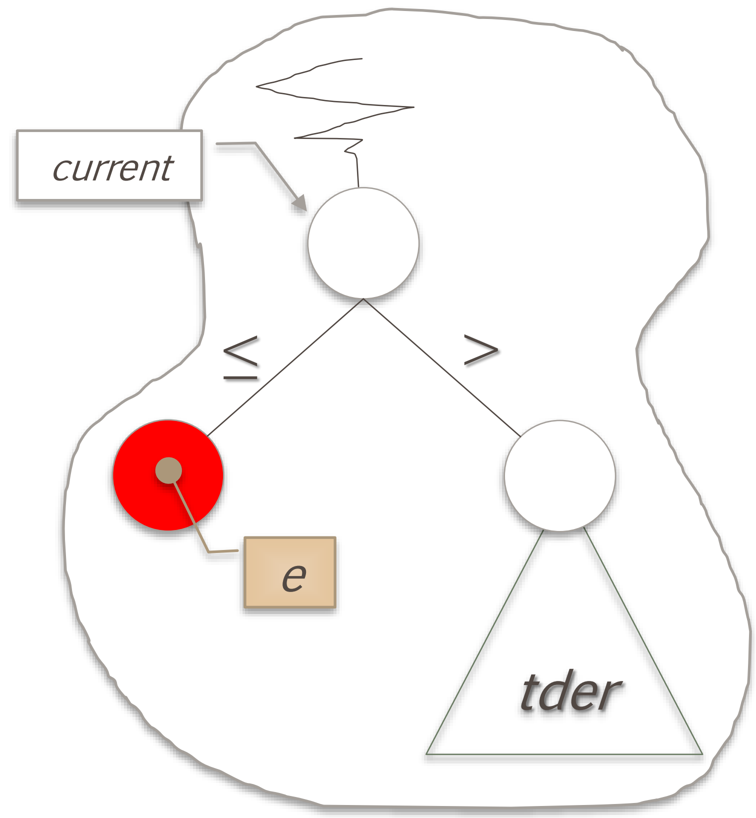


ABB. Operación de inserción (ABB-10)

```
public boolean add(E e) {
    if (isEmpty()) {
        theRoot = new BinaryTreeImp<>(e);
    } else {
        int x = 0;
        BinaryTree<E> current = null;
        BinaryTree<E> child = theRoot;
        while (!child.isEmpty()) {
            current = child;
            x = compare(e, current.label());
            if (x > 0) { // añadir  al subárbol derecho
                child = current.right();
            } else {     // añadir al subárbol izquierdo
                child = current.left();
            }
        }
```

ABB. Operación de inserción (ABB-11)

```
    if (x>0) {
            current.setRight(new BinaryTreeImp<>(e));
        } else {
            current.setLeft(new BinaryTreeImp<>(e));
        }
    }

    size++;
    return true;
}
```

ABB. Operación de extracción (ABB-12)

`t.remove(o)`

- Si el árbol está vacío no hay que hacer nada.
- En caso contrario, se busca la posición del nodo *child* que contiene la etiqueta *o*.

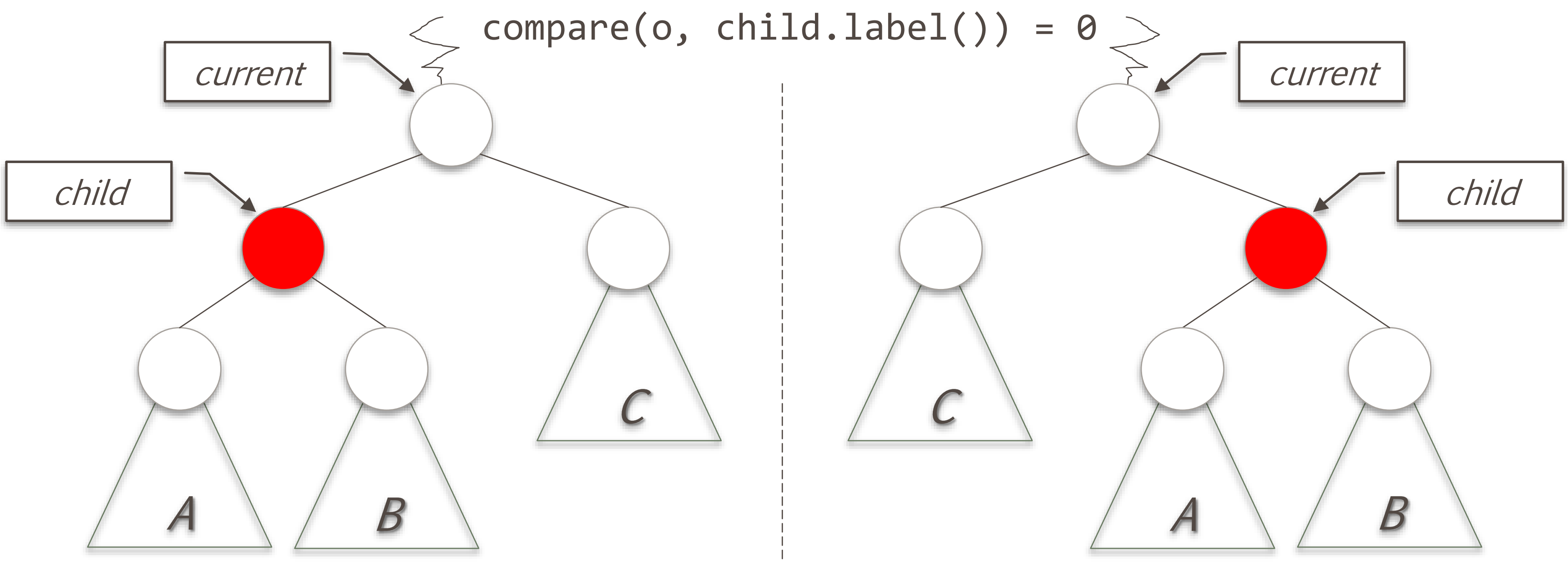
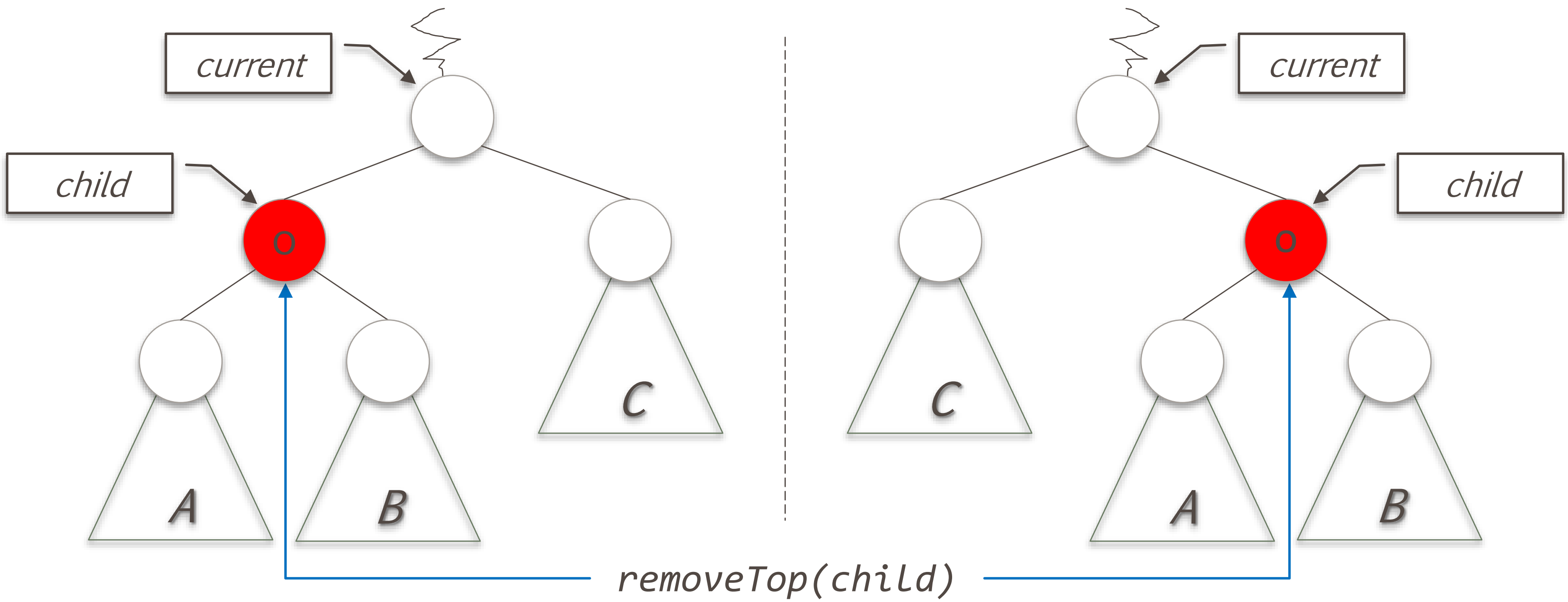


ABB. Operación de extracción (ABB-13)



- Por lo general, eliminar el nodo en el que se encuentra el elemento **o** es un problema, ya que podría tener dos hijos. Así que se eliminará un nodo descendiente menos problemático (uno conveniente que a lo sumo tenga un hijo):
  - El nodo que retorne la operación interna `removeTop(child)`

ABB. Operación de extracción (ABB-14)

- La raíz no tiene padre, así que es necesario tratar ésta de forma particular

```java
public boolean remove(Object o) {
    if (isEmpty()) { // caso 1: el árbol está vacío
         return false;
    }

    // BST no vacío
    if (compare(o, theRoot.label()) == 0) {
        // caso 2: el elemento o está en la raíz
        theRoot = removeTop(theRoot);
        size--;
        return true;
    }
```

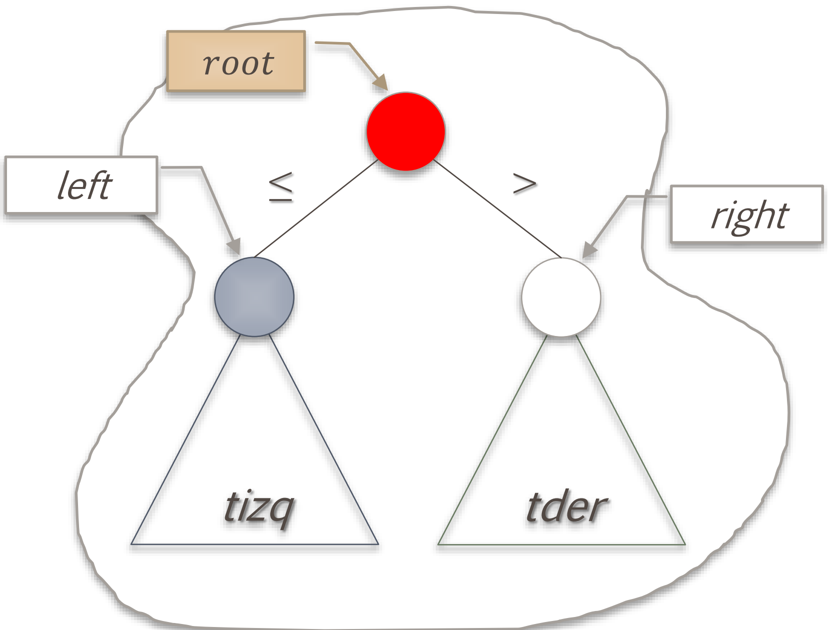
ABB. Operación de extracción (ABB-15)

```
// caso general: el elemento o no está en la raíz
BinaryTree<E> current = null;
BinaryTree<E> child = theRoot;
while (!child.isEmpty()) {
    current = child;
    int x = compare(o, child.label());
    if (x > 0) { // buscar en el subárbol derecho
        child = child.right();
        if (!child.isEmpty()
            && compare(o,  child.label()) == 0) {
            current.setRight(removeTop(child));
            size--;
            return true;
        }
    } else { // buscar en el subárbol izquierdo
```

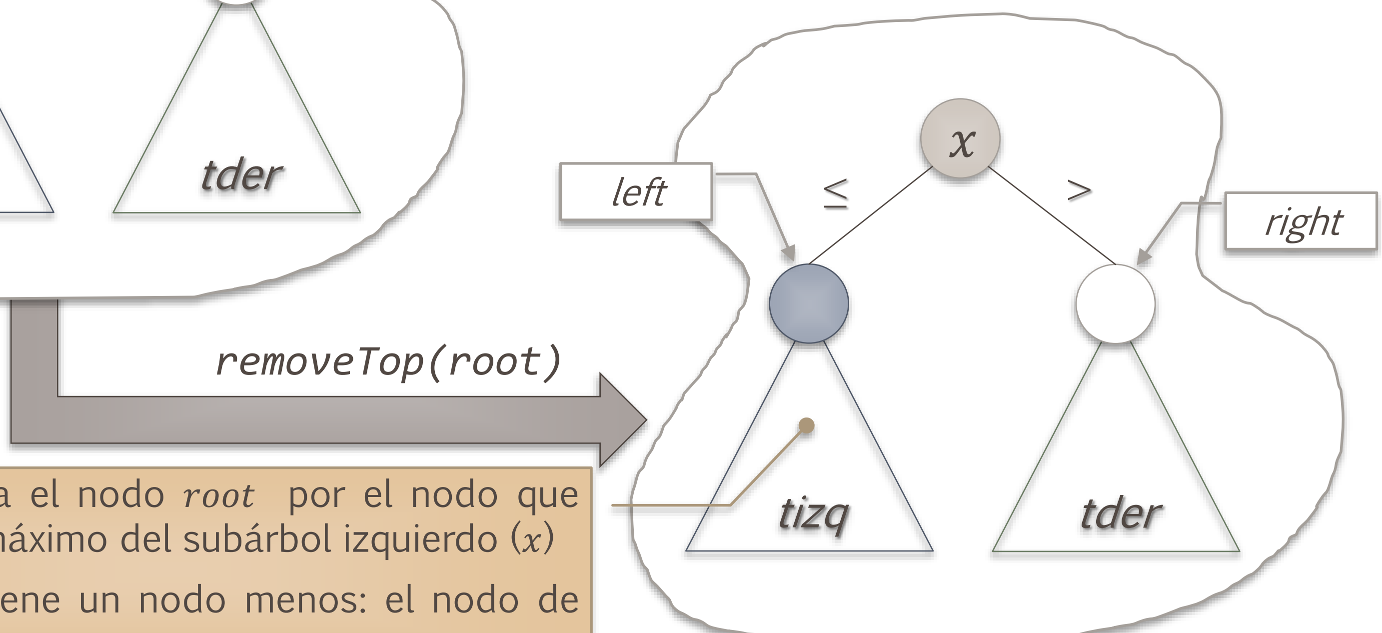
ABB. Operación de extracción (ABB-16)

```
            // buscar en el subárbol izquierdo
            child = child.left();
            if (!child.isEmpty()
                && compare(o, child.label()) == 0) {
                current.setLeft(removeTop(child));
                size--;
                return true;
            }
        }
    }

    return false;
}
```

ABB. Operación `removeTop(root)` (ABB-17)



Hay que reemplazar la raíz del subárbol recibido como argumento (*root*). Su etiqueta es el elemento a eliminar en el ABB



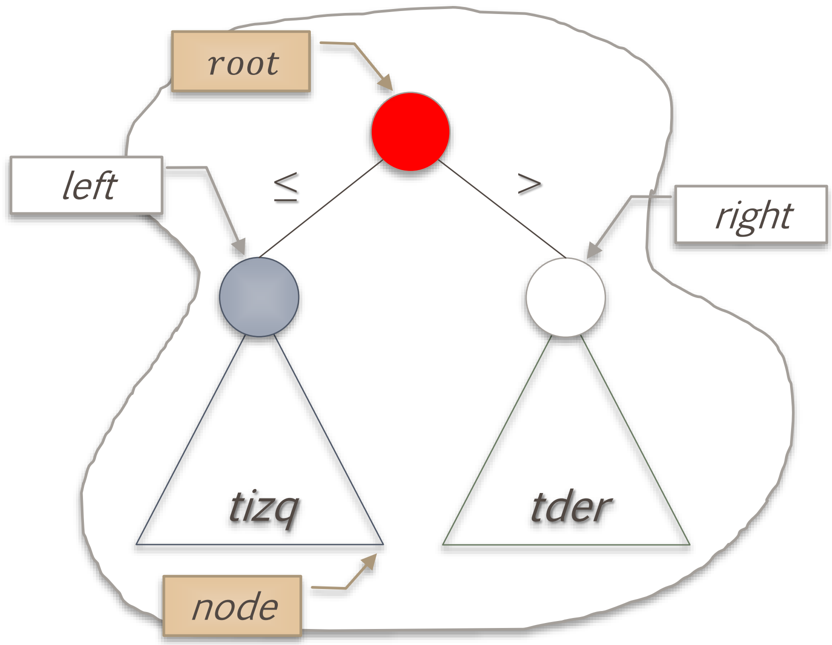
Se reemplaza el nodo *root* por el nodo que contiene el máximo del subárbol izquierdo (x)

Ahora *tizq* tiene un nodo menos: el nodo de etiqueta x.

ABB. Operación `removeTop(root)` (ABB-18)

- Hay que reemplazar la raíz del subárbol recibido (`root`)

```
Node<E> left = root.left();
Node<E> right = root.right();
```



En general, la nueva raíz será el nodo, node, de mayor valor (*máximo*) del subárbol izquierdo, *tizq*.

Si el árbol no pudiera contener elementos repetidos, otra opción sería el nodo de menor valor (*mínimo*) del subárbol derecho, *tder*.

ABB. Operación `removeTop(root)` (ABB-19)

- Descender siempre por la derecha de *tizq* hasta alcanzar un nodo, `current`, que no tenga subárbol derecho. Su etiqueta, `x`, es el máximo del subárbol de raíz `left`.

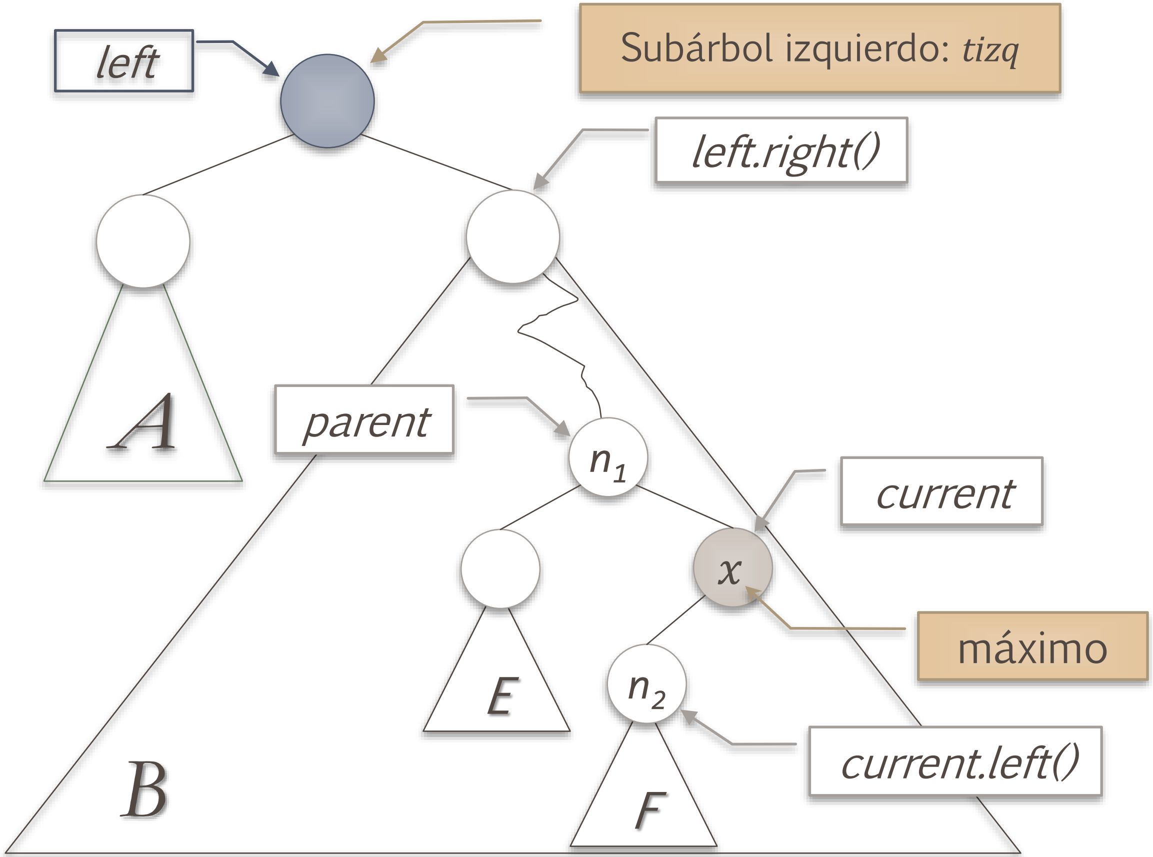
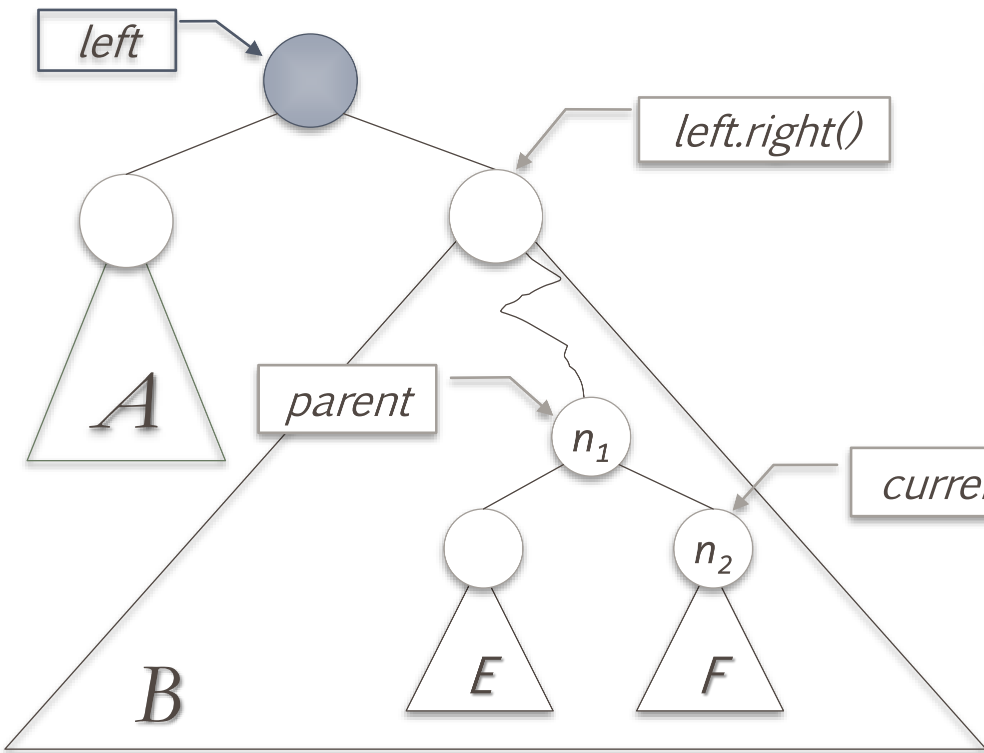


ABB. Operación `removeTop(root)` (ABB-20)

- El nodo `current` de etiqueta `x` es el nodo que va a reemplazar al nodo `root`



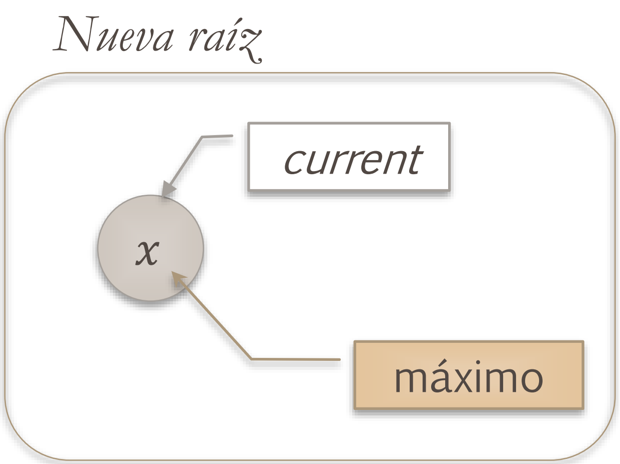


ABB. Operación `removeTop(root)` (ABB-21)

- Caso particular 1 (`left` es el árbol vacío)

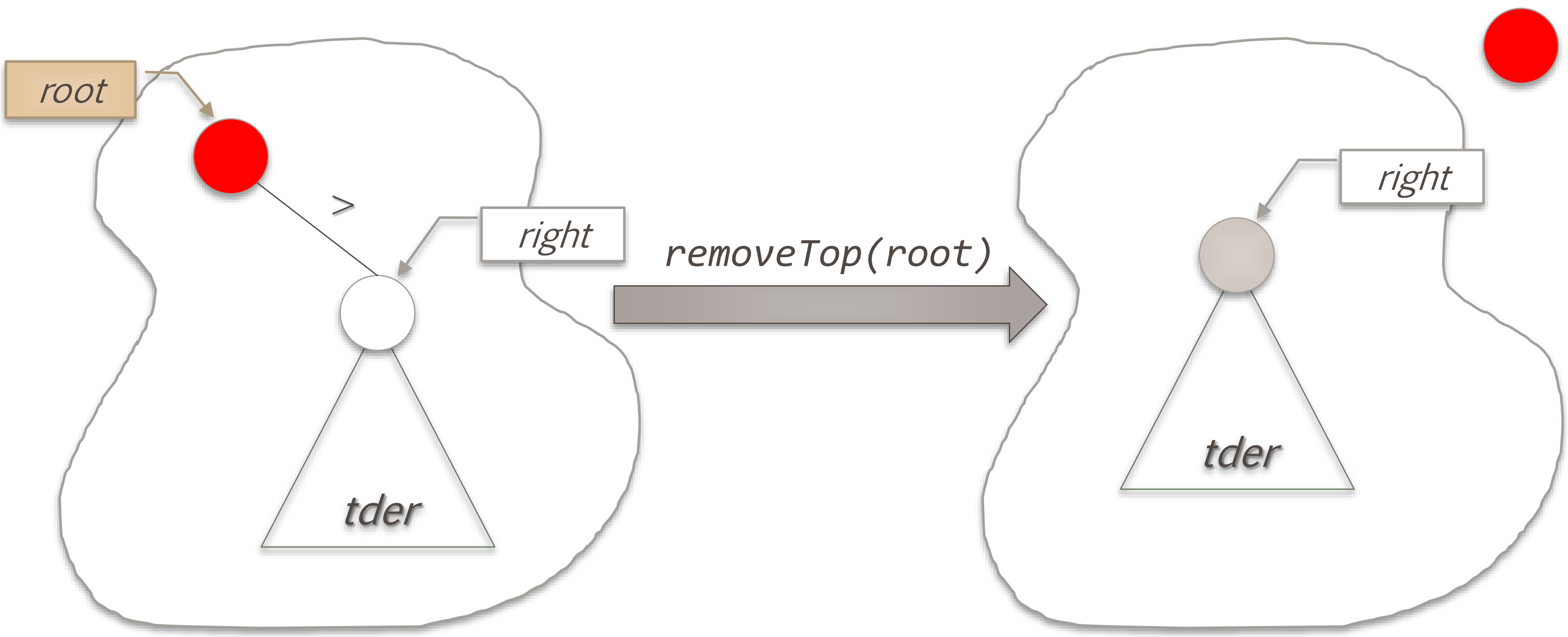


ABB. Operación `removeTop(root)` (ABB-22)

- Caso particular 2 (`right` es el árbol vacío)
  - Se puede resolver con el caso general, pero resulta más simple solucionarlo de forma simétrica al caso 1.

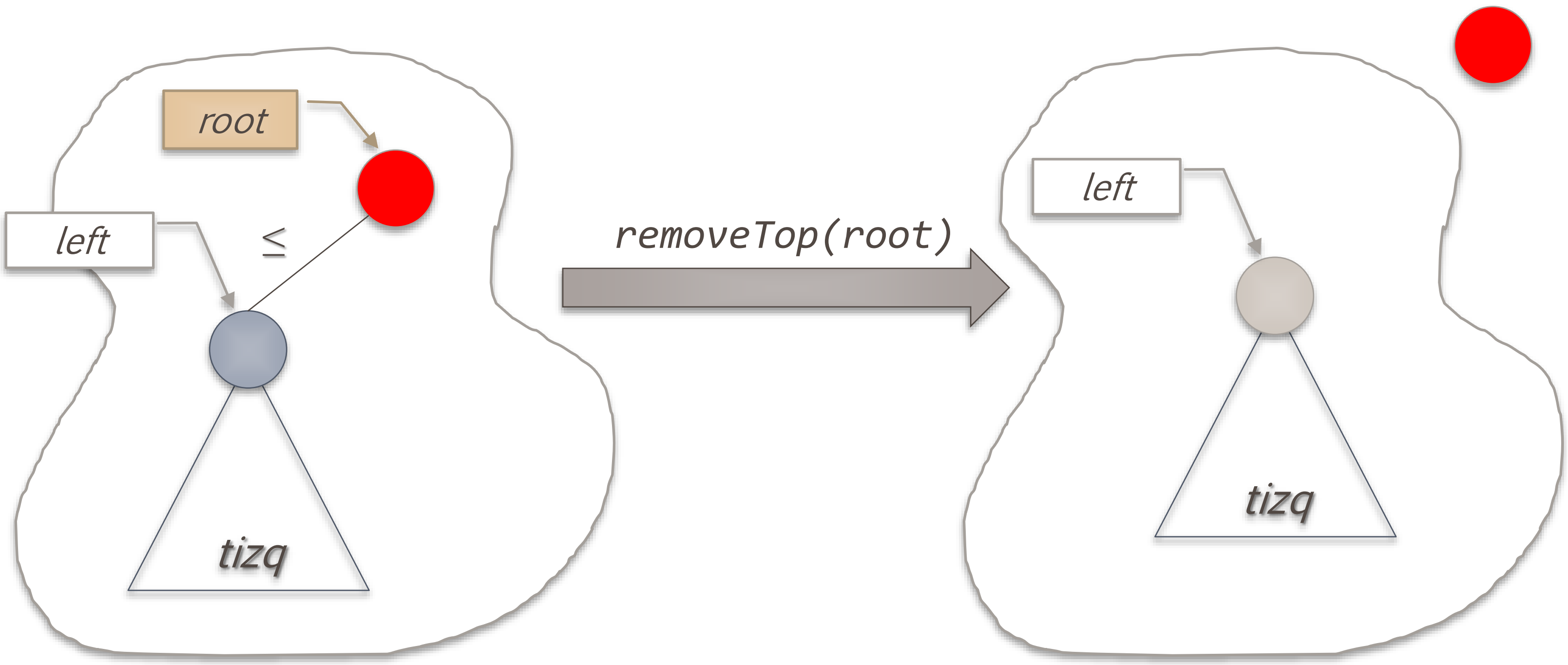


ABB. Operación `removeTop(root)` (ABB-23)

- Caso particular 3 (`left.right()` es el árbol vacío)
  - La etiqueta x del nodo `left` es el máximo del subárbol izquierdo

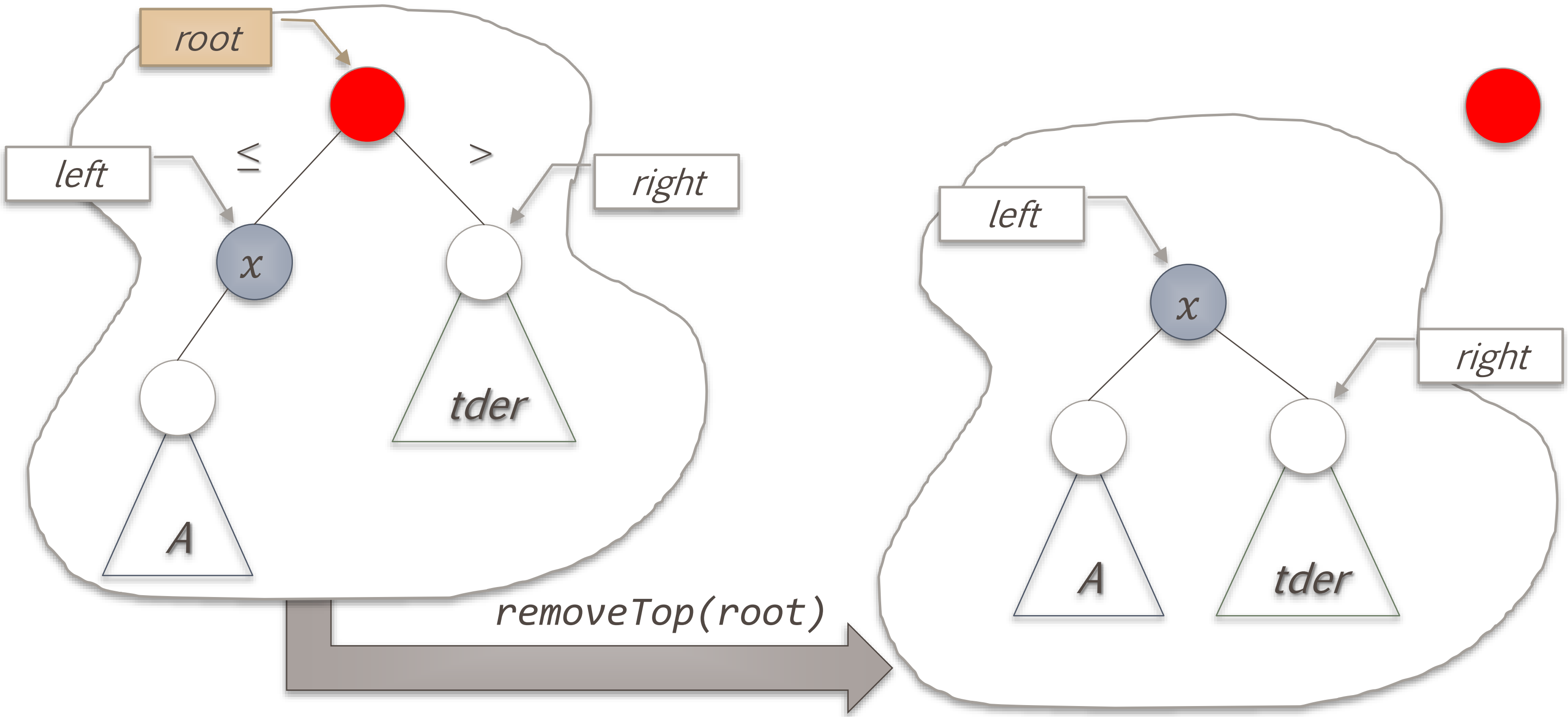


ABB. Operación `removeTop(root)` (ABB-24)

```
private Node<E> removeTop(Node<E> root) {
    BinaryTree<E> left = root.left();
    BinaryTree<E> right = root.right();

    if (left.isEmpty() {               // caso 1
        return right;
    }

    if (right.isEmpty()) {             // caso 2
        return left;
    }

    if (left.right().isEmpty()) {       // caso 3
        left.setRight(right);
        return left;
    }
```

ABB. Operación `removeTop(root)` (ABB-25)

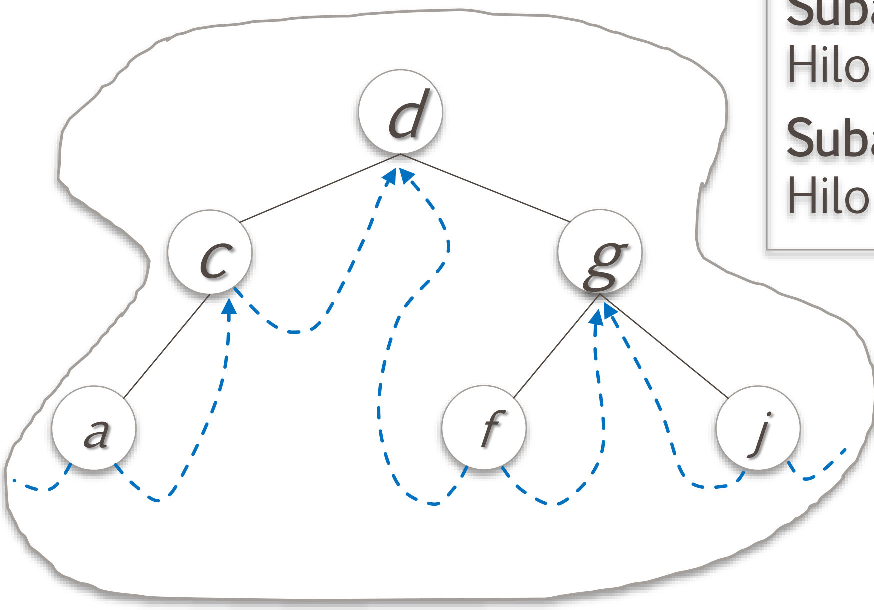
```
    // Caso general
    BinaryTree<E> parent = root;
    BinaryTree<E> current = left;
    while (!current.right().isEmpty()) {
        parent = current;
        current = current.right();
    }

    parent.setRight(current.left());
    current.setLeft(left);
    current.setRight(right);

    return current;
}
```

ABB. Iterador (ABB-26)

- Iterador en un ABB
  - En lo árboles binarios de búsqueda es relevante el recorrido en profundidad en **inorden**.
    - Este recorrido obtiene la secuencia de etiquetas de los nodos del ABB ordenadas entre sí en sentido creciente.
  - Para implementar el iterador en inorden se require una estructura auxiliar para almacenar los nodos del árbol (una cola LIFO o pila).
    - En la pila se guardarán todos los nodos del camino que va desde la raíz del subárbol que se visita en inorden, hasta el nodo hoja situado más a la izquierda (que es el primero que se visita en inorden para este subárbol).

ABB enhebrados. (ABB-27)

- Árboles binarios de búsqueda enhebrados
  - Se reemplazan árboles vacíos por hilos
  - Es necesario añadir al área de datos un par de booleanos para distinguir los subárboles (izquierdo y derecho) de los hilos.



Subárbol izquierdo vacío
Hilo al nodo predecesor en inorden
Subárbol derecho vacío
Hilo al nodo sucesor en inorden

ABB enhebrados. Iterador (ABB-28)

- Ventaja
  - Al tener los nodos enhebrados se puede implementar el iterador de recorrido en inorden sin necesidad de una estructura auxiliar (una pila)
  - Si el único objetivo de utilizar hilos es realizar el iterador `Iterator<E>`, es suficiente con mantener los hilos derechos.
    - En todo caso, si se incluyen también los hilos izquierdos se puede proporcionar un iterador para recorrer los nodos del árbol en sentido inverso: de mayor a menor.

ABB+ equilibrados

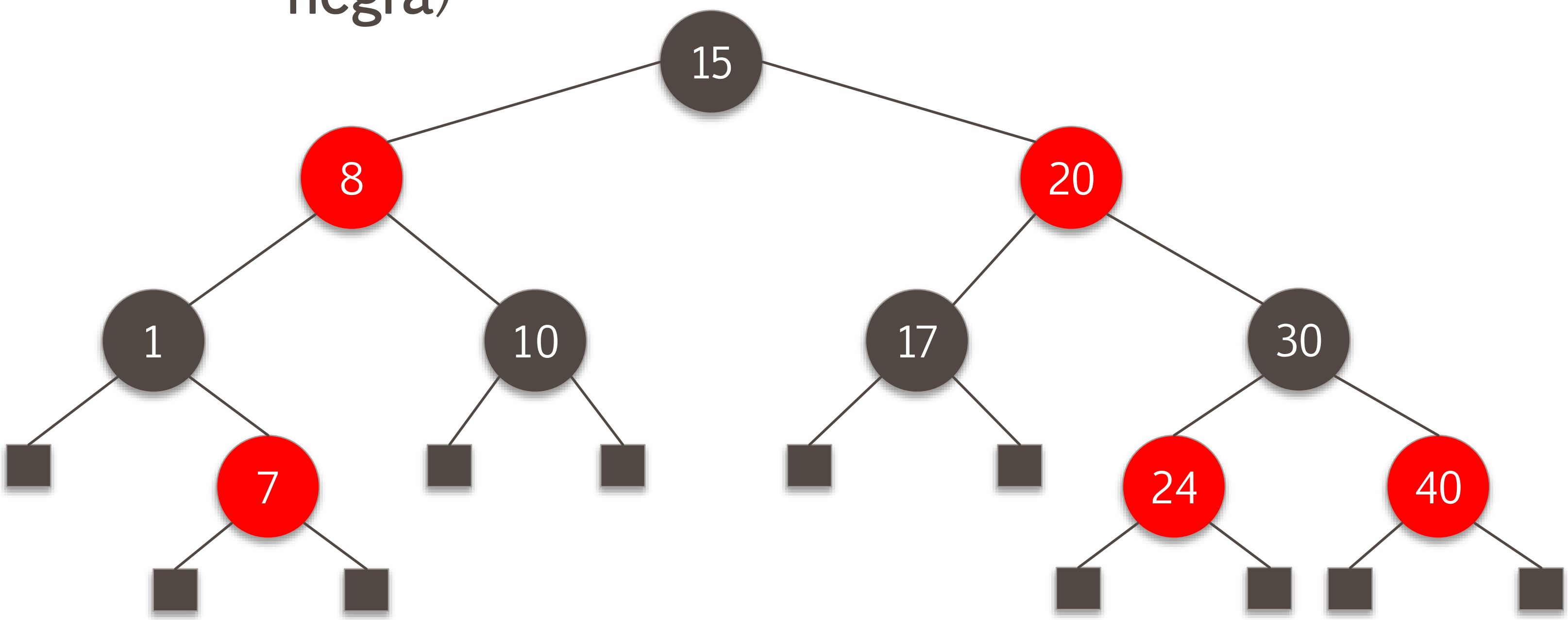
- Restauración del equilibrio
  - Se maximiza el número de nodos para cada altura del árbol, de forma que si n es el número de nodos del árbol su altura es $k \cdot \log(n)$.
    - Se garantiza que las operaciones de búsqueda, inserción y borrado es de $O(\log(n))$
- Tipos
  - Árboles AVL
    - Se define un factor de equilibrio para cada nodo: $|h_{izq} - h_{der}| \leq 1$

Árboles Rojo-Negro (1)

- Árboles rojo-negro
  - Árbol binario estricto
    - Los nodos nulos se tienen en cuenta en la definición de las operaciones
    - Todo nodo hoja es nulo
  - Cada nodo tiene estado *rojo* o *negro*
  - Los nodos hoja (nulos) son *negros*
  - La raíz es *negra*

Árboles Rojo-Negro (2)

- Condiciones
  1. Un nodo *rojo* tiene dos hijos *negros*
  2. Todo camino de la raíz a cualquier hoja pasa por el mismo número de nodos *negros* (**altura negra**)



Árboles Rojo-Negro (3)

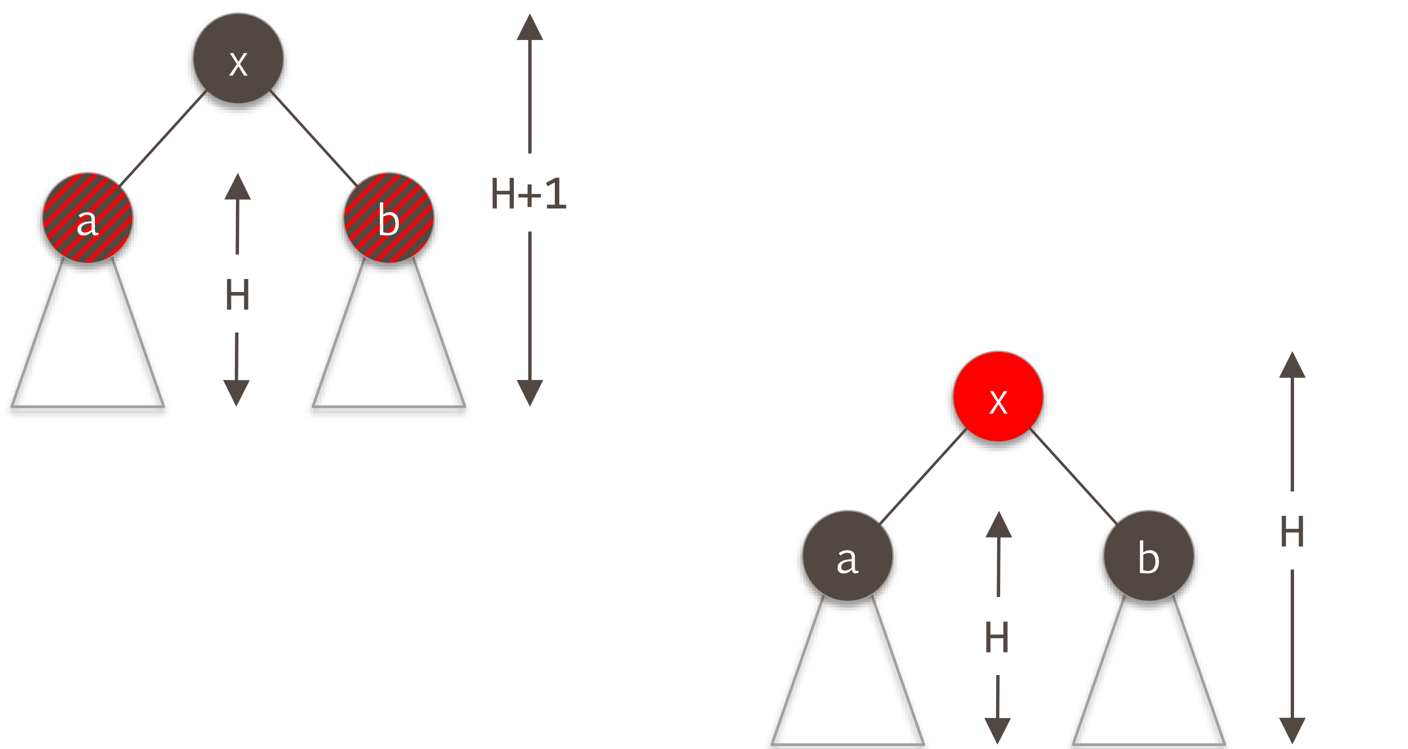
- Altura negra

$$H(n) = \begin{cases} \max\left(H(n_{izq}), H(n_{der})\right) + 1 & \text{si } n \text{ es negro} \\ \max\left(H(n_{izq}), H(n_{der})\right) & \text{si } n \text{ es rojo} \end{cases}$$

  - La segunda condición de los árboles rojo-negro se puede expresar en la forma:

> Para todo nodo interno (no nulo), la altura negra de su hijo izquierdo es igual a la altura negra de su hijo derecho

Árboles Rojo-Negro (4)



Árboles Rojo-Negro (5)

- Propiedades
  - Cambiar un nodo de *rojo* a *negro* no afecta a la condición 1, pero si a la condición 2 (la altura negra se incrementa en todos los nodos ascendientes)
  - Cambiar un nodo de *negro* a *rojo* puede afectar a la condición 1 (si el padre o alguno de los hijos es *rojo*) y también a la condición 2 (la altura negra se decrementa en todos los nodos descendientes)
  - Si como resultado de una operación la raíz pasa a ser *rojo*, se puede cambiar a *negro* directamente sin afectar a las condiciones
  - Borrar un nodo *rojo* no afecta a las condiciones, pero borrar un nodo *negro* sí (la altura negra decrece en sus ascendientes)

Árboles Rojo-Negro (6)

- Inserción de un nodo
  - Se realiza igual que en un ABB y al nuevo nodo se le da el color *rojo*
    - No se viola la condición 2, pero se puede violar la 1 (si el padre del nodo insertado también es *rojo*)
  - El equilibrio se restablece según el caso
  - Caso 0 (trivial)
    - Si el nodo padre del insertado es ***negro***, no se realiza ningún ajuste

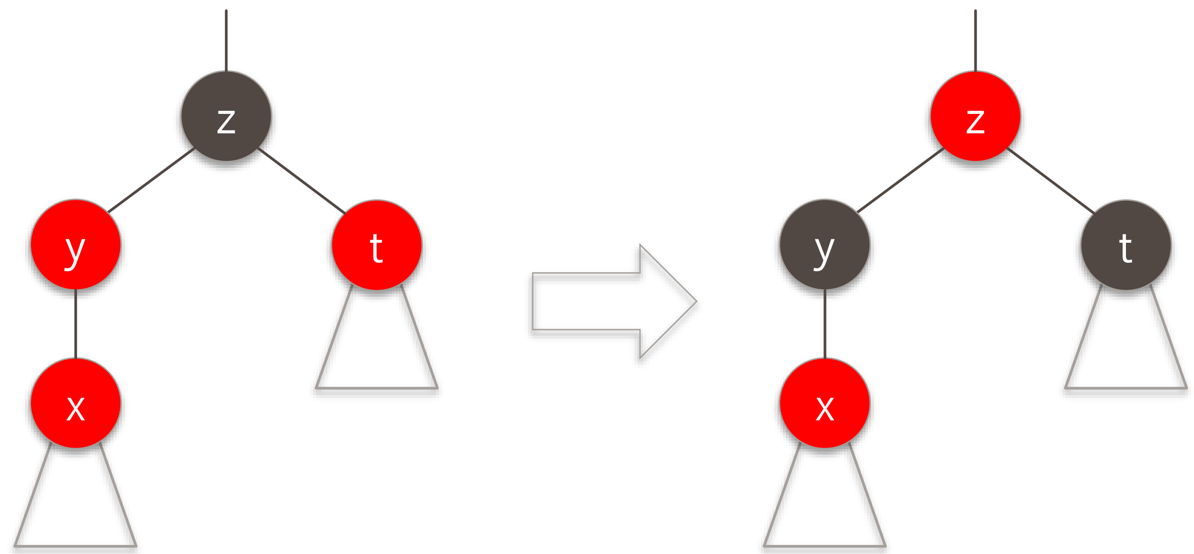
Árboles Rojo-Negro (7)

- Resto de casos (un bucle)
  - Donde x representa el nodo a comprobar y es *rojo*
    - Si el padre de x fuera *negro* termina el bucle
    - Si x no tuviera padre (x es el nodo raíz) se cambia a *negro* y termina el bucle
    - En cualquier otro caso se realiza cierta operación, x pasa a ser otro nodo *rojo* y continua el bucle

A continuación se van a presentar los distintos casos cuando la inserción del nodo x tiene lugar en el subárbol izquierdo, pero no se presentan los casos para la inserción en el subárbol derecho que serían simétricos de los primeros.

Árboles Rojo-Negro (8)

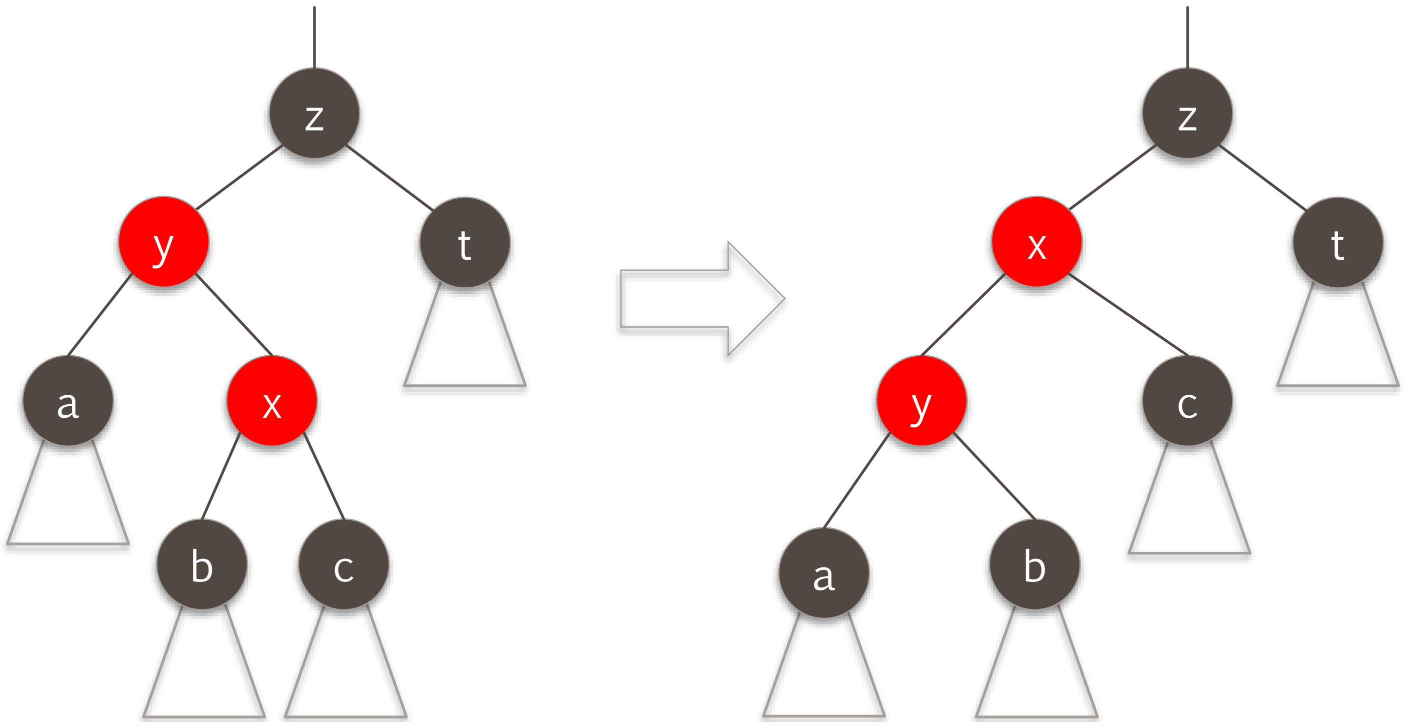
- Caso 1. Tío *rojo*, nodo x a la izquierda o a la derecha



- Se cambian los colores de y, z, y t. En la siguiente iteración z pasa a ser el nodo x

Árboles Rojo-Negro (9)

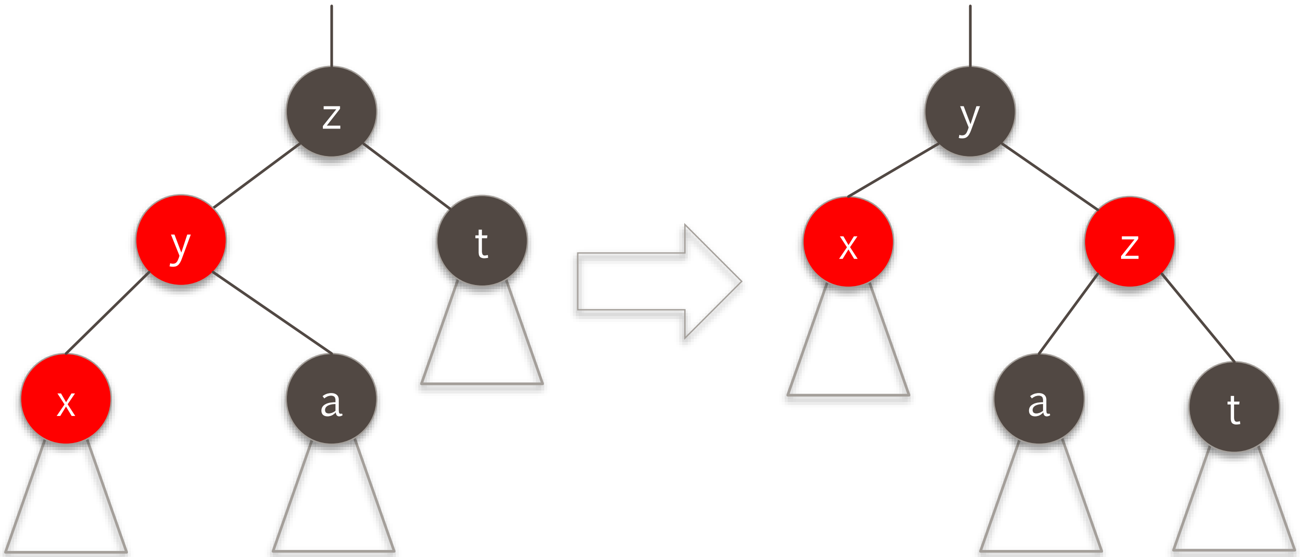
- Caso 2. Tío *negro*, nodo x a la derecha



- Se gira a la izquierda el subárbol de raíz y. En la siguiente iteración y pasa a ser el nodo x `(caso 3)`

Árboles Rojo-Negro (10)

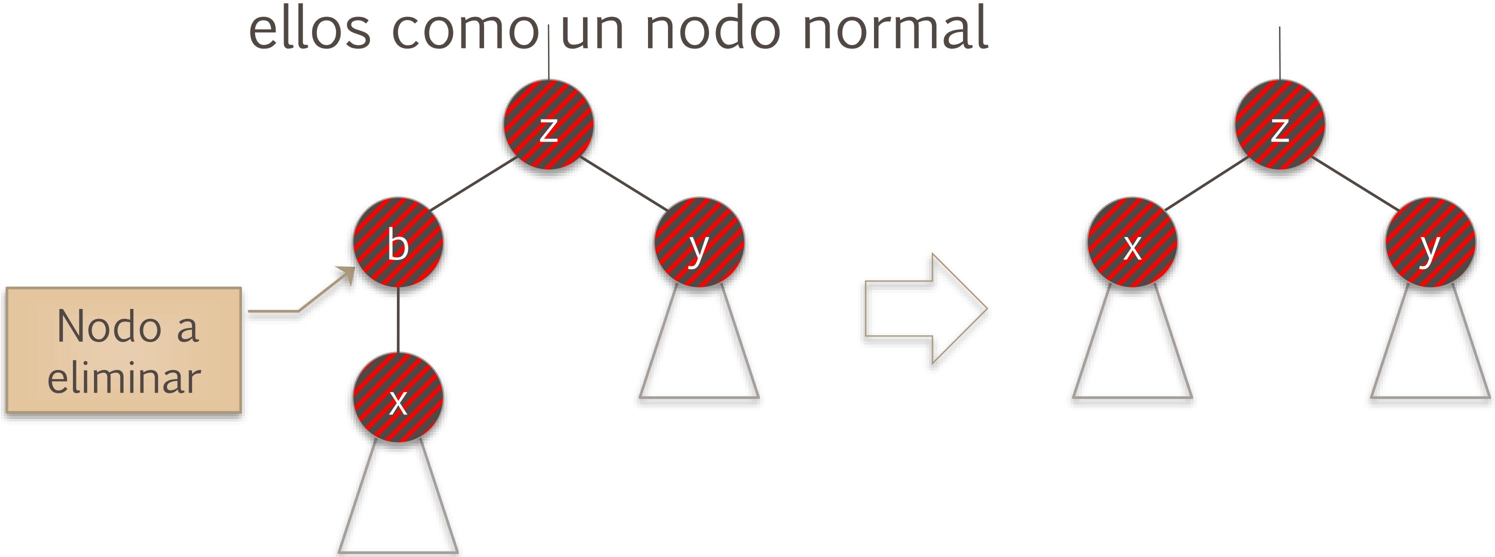
- Caso 3. Tío *negro*, nodo x a la izquierda



- Se gira a la derecha el subárbol de raíz z y se cambian de color los nodos z e y. El bucle termina

Árboles Rojo-Negro (11)

- Borrado de un nodo
  - Se realiza igual que en un ABB
    - El nodo a eliminar tendrá dos hijos y al menos uno de ellos será un nodo nulo
    - En el caso de que el nodo a eliminar tenga dos hijos nulos, se considerará uno cualquiera de ellos como un nodo normal



Árboles Rojo-Negro (12)

- En el esquema de borrado previo
  - b es el nodo que se borra
  - x es el hijo no nulo (o uno cualquiera de los hijos si ambos son nulos)
  - z es el padre del nodo borrado
  - y es el nodo hermano del nodo borrado
- Para restablecer el equilibrio, es necesario conocer los nodos x y z. El nodo x puede ser nulo, al igual que el y, pero el nodo z debe existir.

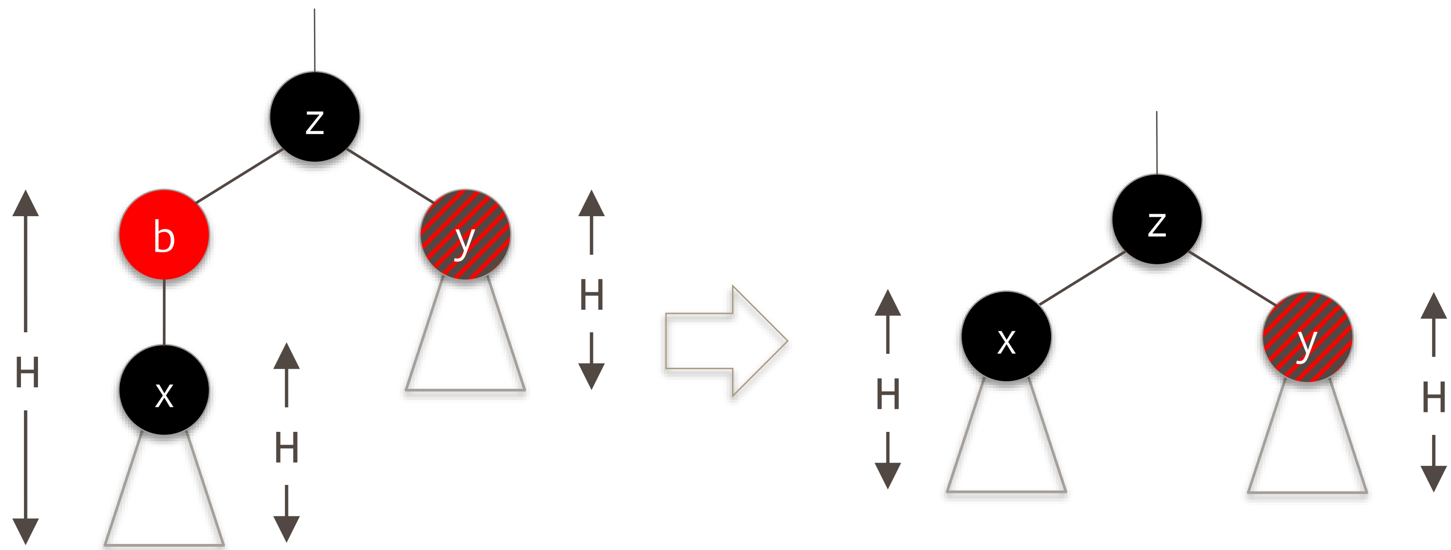
Árboles Rojo-Negro (13)

- Por tanto, eliminar el nodo raíz es un caso especial:
  - Se elimina la raíz y si la nueva raíz es de color rojo, se cambia de color a negro.

A continuación, se van a presentar los distintos casos cuando el nodo b a borrar es un hijo izquierdo, pero no se presentan los casos en que b es un hijo derecho que serían simétricos de los primeros.

Árboles Rojo-Negro (14)

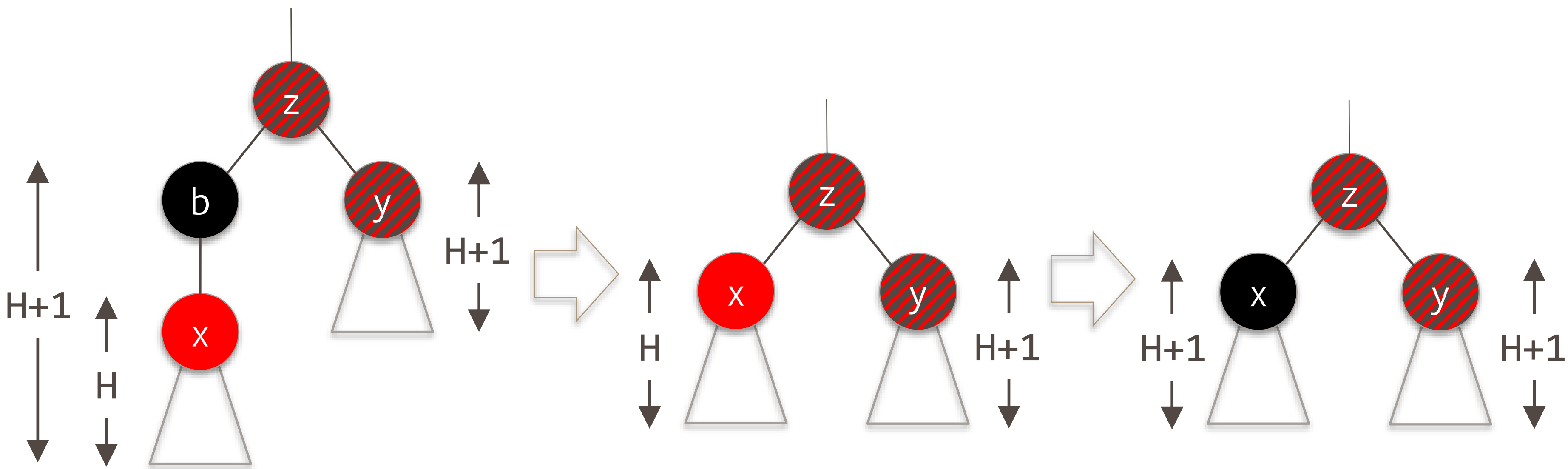
- Casos triviales
  - El nodo b a eliminar es rojo
    - El árbol sigue siendo rojo-negro



En adelante el nodo b será negro

Árboles Rojo-Negro (15)

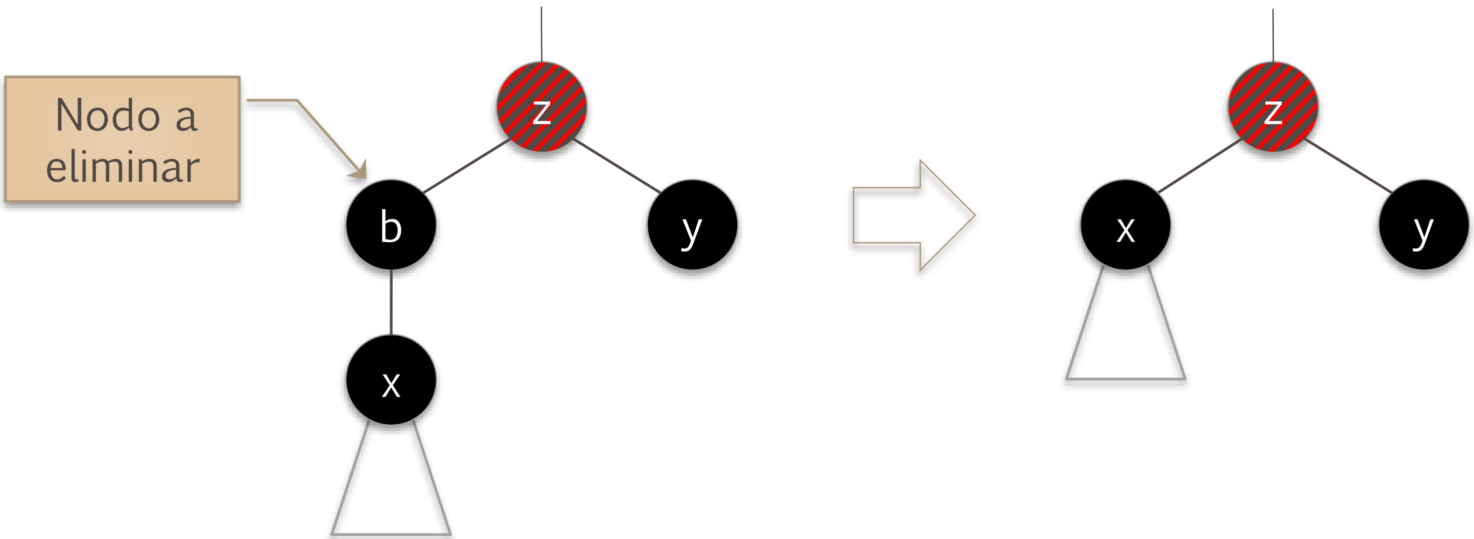
- El hijo x es rojo
  - Los hijos del nodo z no tienen la misma altura negra, con cambiar el color de x a negro se restablece la condición



En adelante los nodos b y x serán negros

Árboles Rojo-Negro (16)

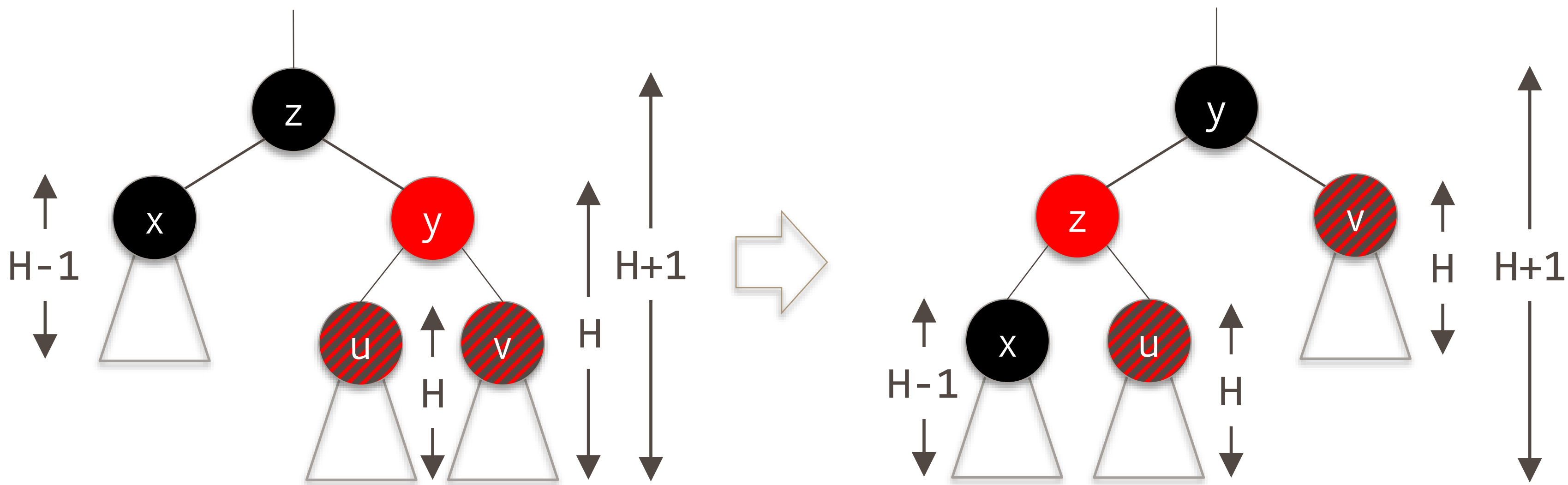
- Caso imposible
  - Nodo hermano y nulo (negro)



No es posible que x sea negro y esté desequilibrado respecto a un hermano nulo

Árboles Rojo-Negro (17)

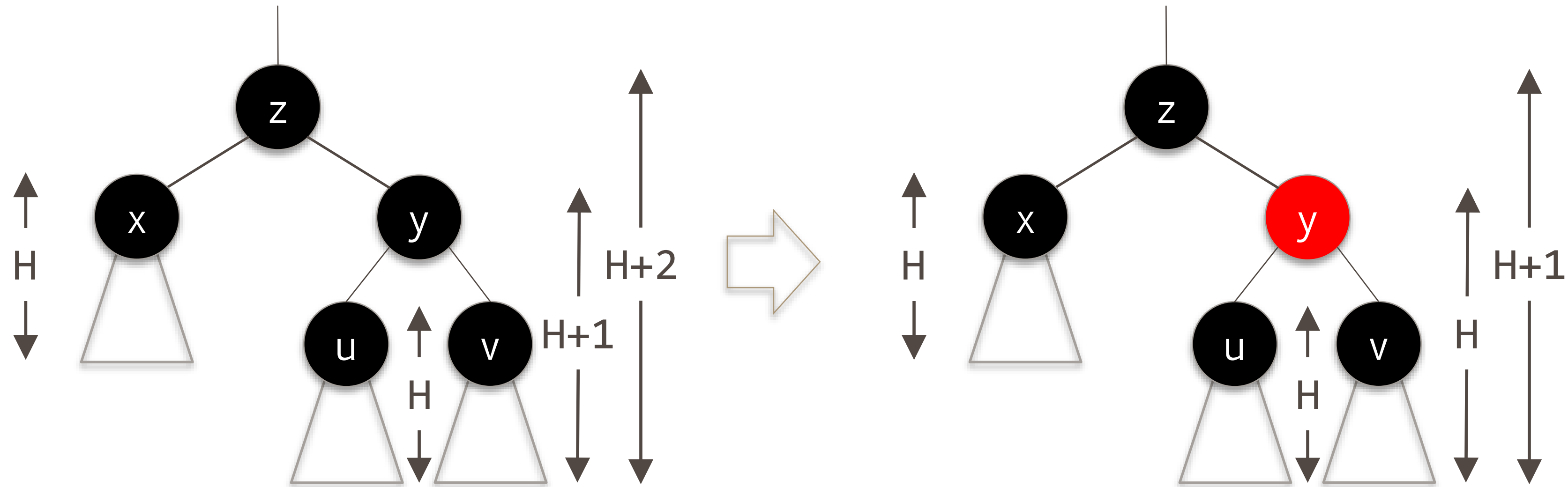
- Caso 1
  - Hermano *y* rojo, padre *z* negro



- Rotación a la izquierda padre-hermano y se cambian los colores. El nodo *x* sigue teniendo una altura menos que su hermano, pero ahora su padre es rojo (caso 3, 4 o 5)
- En la iteración siguiente se comprueban los mismos nodos

Árboles Rojo-Negro (18)

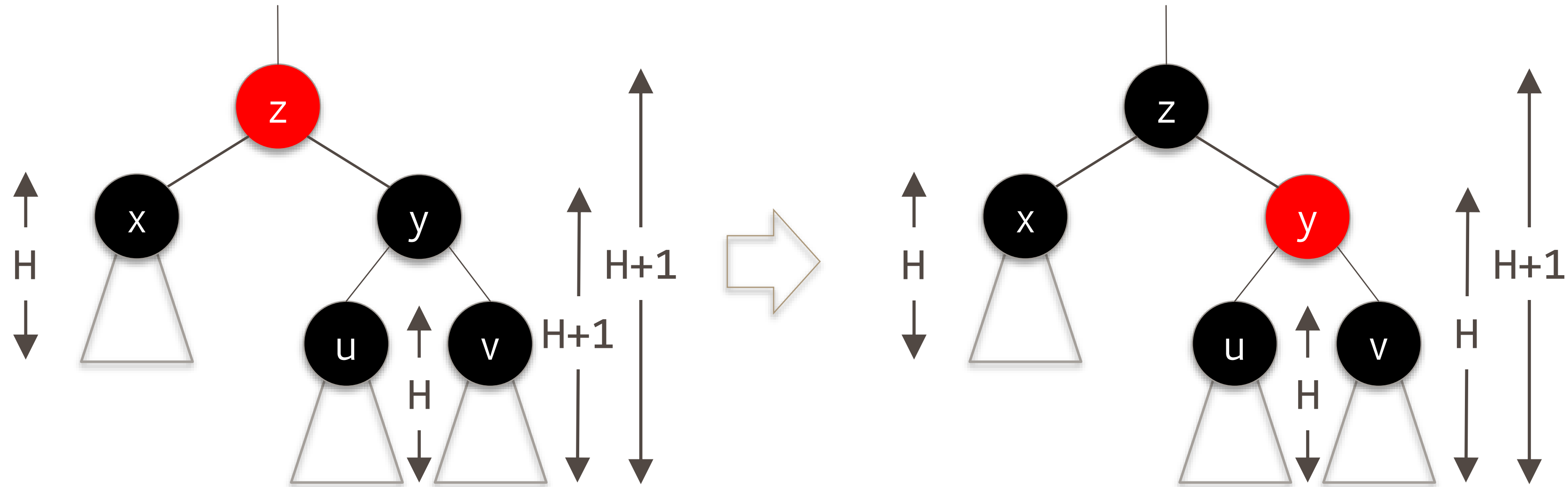
- Caso 2
  - Hermano y negro, padre z negro y sobrinos negros



- Se cambia el color de y a rojo. Disminuye la altura del nodo z
- En la siguiente iteración el nodo z pasa a ser el llamado x y el nodo llamado z sería el padre del actual nodo z. Si el nodo z es la raíz del árbol se cumplen todas las condiciones y termina el bucle

Árboles Rojo-Negro (19)

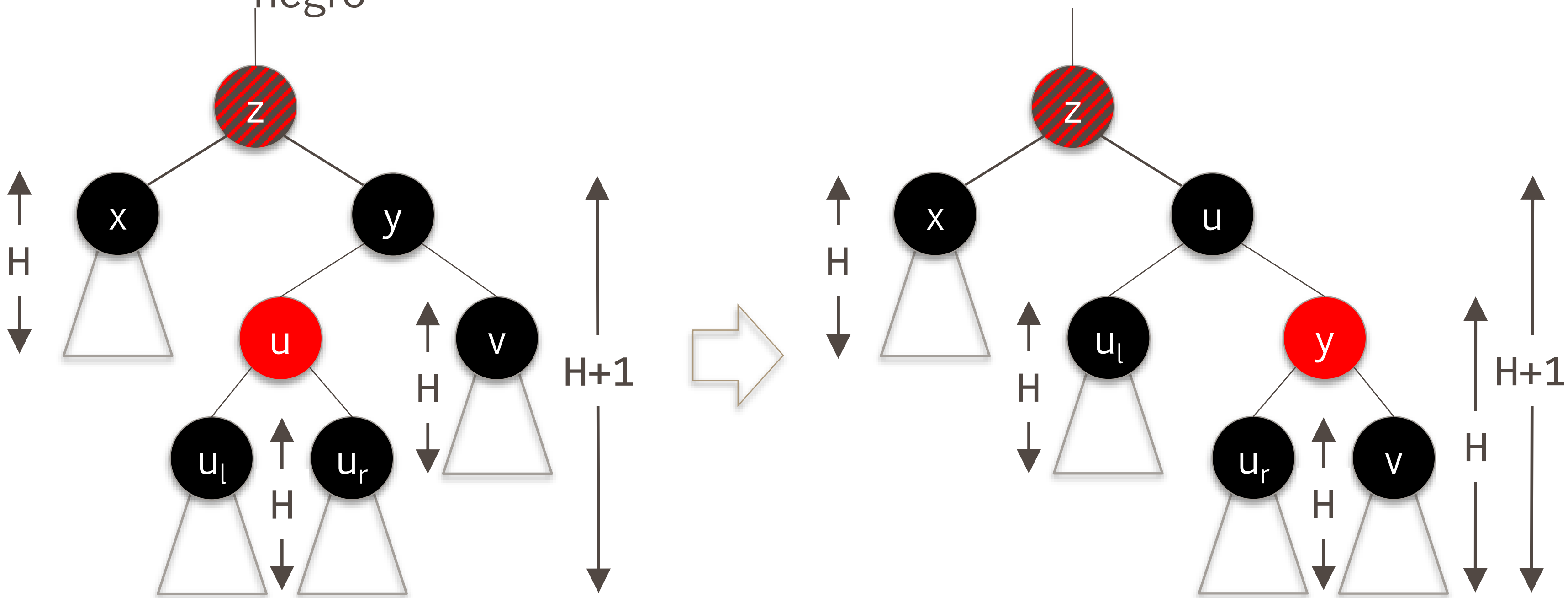
- Caso 3
  - Hermano *y* negro, padre *z* rojo y sobrinos negros



- Se intercambian los colores de los nodos *y* y *z*.
- Se cumplen todas las condiciones y el bucle termina

Árboles Rojo-Negro (20)

- Caso 4
  - Hermano *y* negro, padre *z* rojo/negro y sobrinos rojo y negro

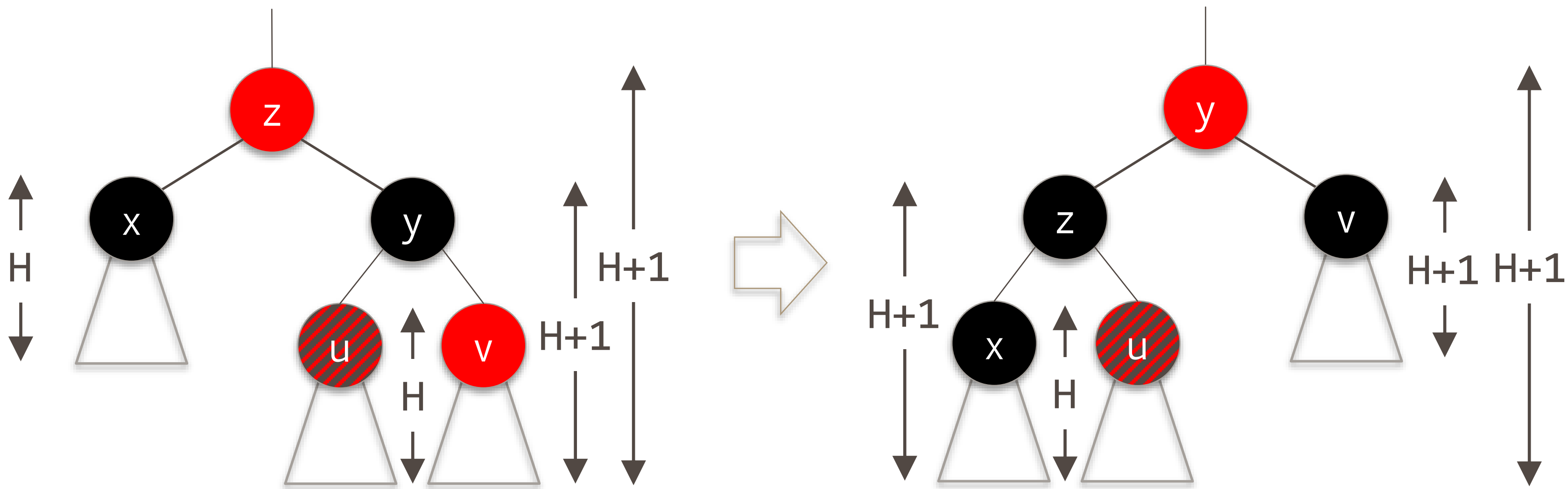


  - Se realiza una rotación a la derecha del hermano y sobrino izquierdo y se cambian sus colores.

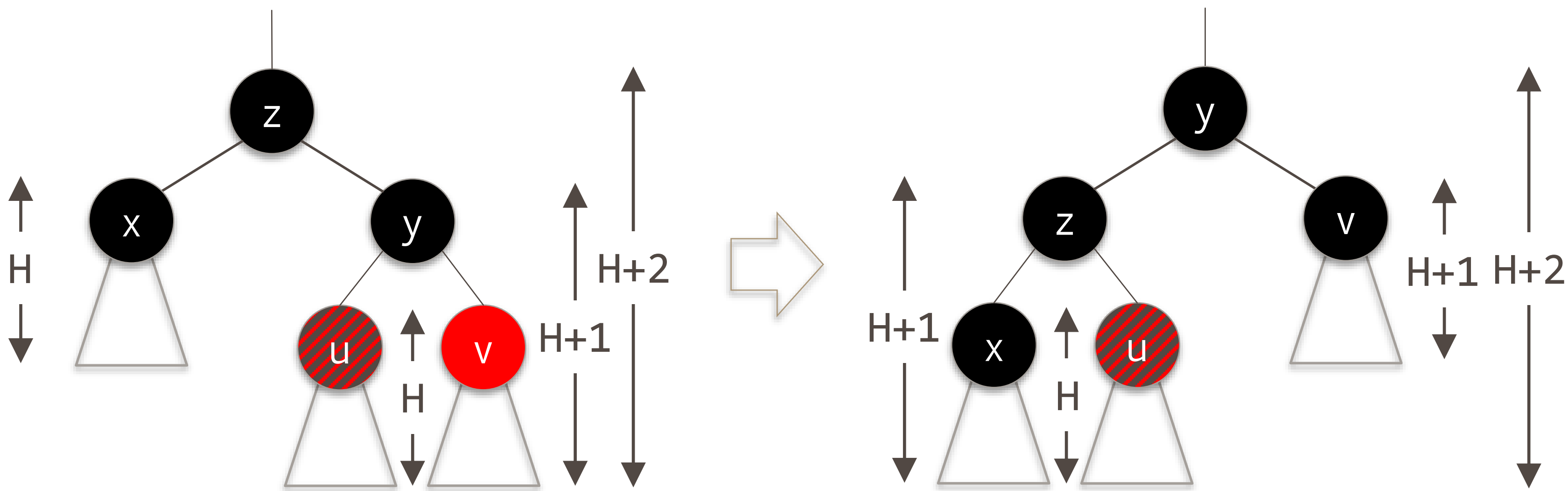
Árboles Rojo-Negro (20)

- En la siguiente iteración se comprueban los mismos nodos, cayendo en el quinto y último caso

- Caso 5
  - Hermano y negro, padre z rojo/negro y sobrinos rojo/negro y rojo



Árboles Rojo-Negro (20)



- Se realiza una rotación a la izquierda padre-hermano. Si el nodo z es rojo cambia a negro, el nodo y cambia al color original del nodo z y el sobrino rojo pasar a ser negro.
- Se cumplen todas las condiciones y termina el bucle.

En ninguno de los cinco casos se cambia el color negro del nodo x, por lo que esté puede ser un nodo nulo