Árboles de búsqueda Rojo-Negros

Red-black binary search trees

Introducción

- Permiten una implementación sencilla y eficiente de los árboles 2-3, utilizando únicamente 2-nodes.
- Los 3-nodes se representan por dos 2-nodes unidos por un enlace “rojo” siempre hacia la izquierda.
- Los 2-nodes se conservan y sus enlaces son los enlaces “negros”.

Árbol rojo-negro

Definición

Es un árbol binario de búsqueda en el que:

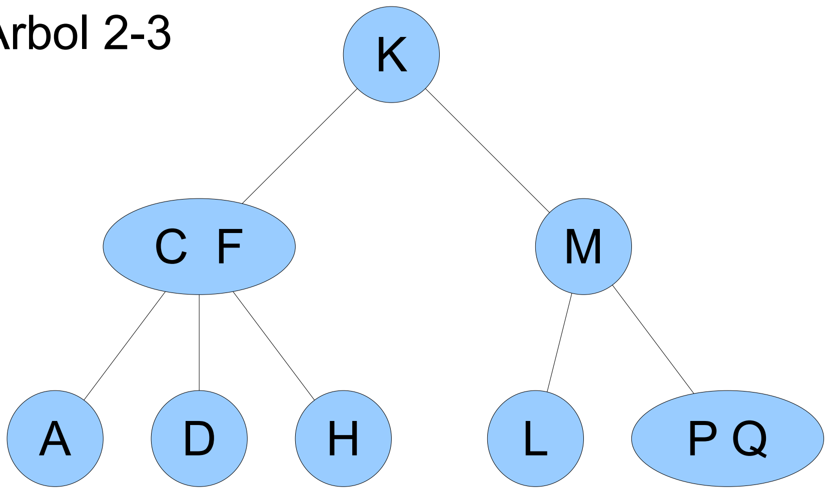
- Los enlaces rojos siempre van hacia la izquierda.
- Ningún nodo está conectado a dos enlaces rojos.
- El árbol está perfectamente balanceado con respecto a los enlaces negros. Es decir, sin contar enlaces rojos en la ruta de la raíz a una hoja.

Correspondencia con los árboles 2-3

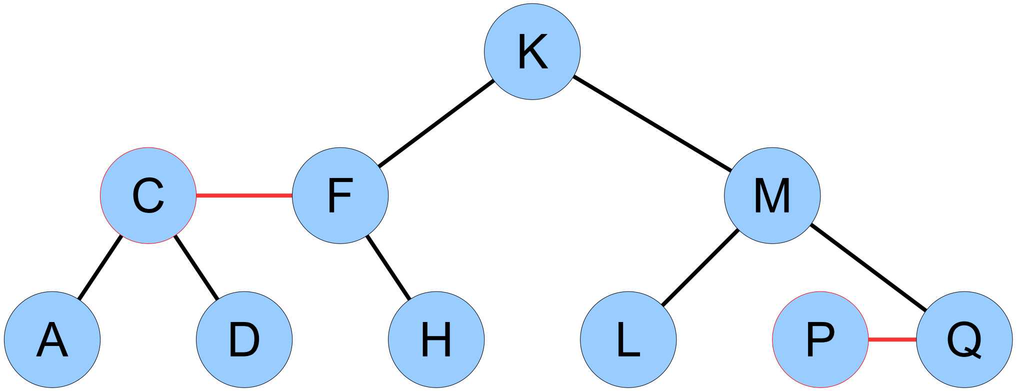
- Hay una correspondencia uno-a-uno entre todo árbol 2-3 y su correspondiente árbol rojo-negro.
- La implementación de los árboles rojo-negros resulta más sencilla, por lo que se prefiere esta representación.
- Para efectos de la representación el color se códifica en el nodo: Si el enlace al padre es rojo, decimos que el nodo es rojo.

Correspondencia 2-3 a rojo-negro

Árbol 2-3



Árbol rojo-negro



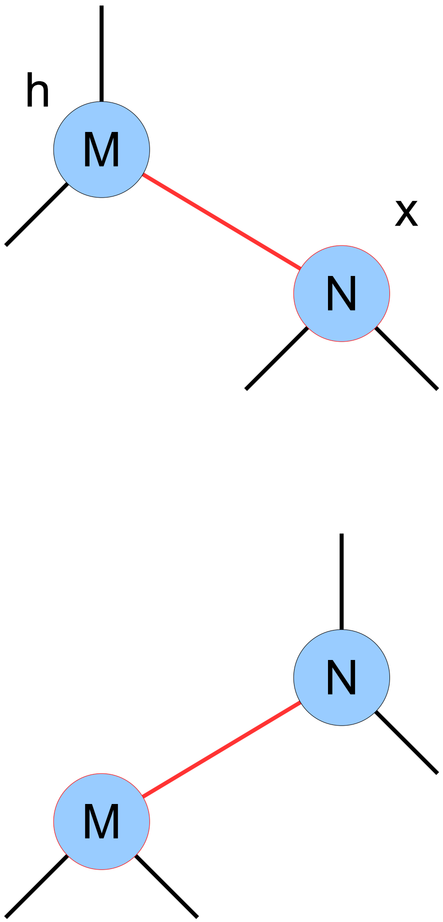
Estructura básica

```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node {
    private Key key;           // key
    private Value val;         // associated data
    private Node left, right;  // links to left and right subtrees
    private boolean color;     // color of parent link
    private int N;             // subtree count
    public Node(Key key, Value val, boolean color, int N) {
        this.key = key;
        this.val = val;
        this.color = color;
        this.N = N;
    }
}

private boolean isRed(Node x) {
    if (x == null) return BLACK;
    return x.color == RED;
}
```

Rotaciones

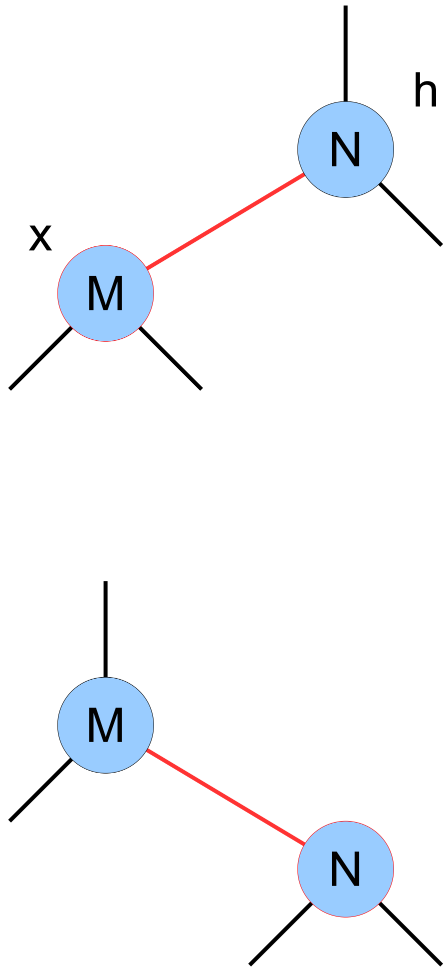
- Operaciones que permiten cambiar el orden de un grupo de nodos para asegurar que se cumplan las condiciones del árbol rojo-negro.
- La rotación devuelve la nueva referencia a la raíz del subárbol.
- Rotación a la izquierda: La menor llave se convierte en el hijo izquierdo de la mayor llave y su enlace en un enlace rojo.
- Rotación a la derecha: La mayor llave se convierte en el hijo derecho de la menor. El enlace a la mayor se hace rojo.

Rotación izquierda




```
private Node rotateLeft(Node h) {
    // assert (h != null) && isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = size(h.left) + size(h.right) + 1;
    return x;
}
```

Rotación derecha




```
private Node rotateRight(Node h) {
    // assert (h != null) && isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = size(h.left) + size(h.right) + 1;
    return x;
}
```

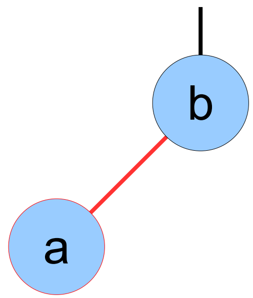
Operaciones de inserción

- Se implementan de forma que haciendo uso de operaciones de rotación se mantengan las propiedades:
  - Correspondencia 1-a-1 con un árbol 2-3
  - Orden de llaves y balance de negro.
  - Enlaces rojos a la izquierda y ningún nodo tiene 2 enlaces rojos.
- El nodo insertado siempre se conecta con enlace rojo.

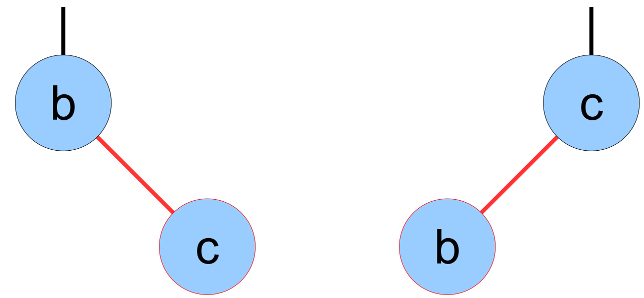
Inserción en un 2-node

- Si el valor a insertar es menor, se agrega al enlace izquierdo y el enlace se hace rojo.
- Si el valor es mayor, se agrega al enlace derecho (con enlace rojo) y se hace rotación izquierda para restaurar el árbol.

put(a,value) :



put(c,value) + root = rotateLeft(root) :



Inserción en un 3-node
Caso 1 : llave mayor a las otras dos

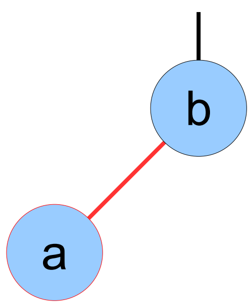
- Se agrega el nuevo nodo conectado al enlace derecho (con color rojo).
- El nodo padre queda inválido por tener dos enlaces rojos. Se hace una operación flipColors.
- Como resultado se convierte el 3-node en dos 2-nodes y la altura crece en 1.

Operación flipColors

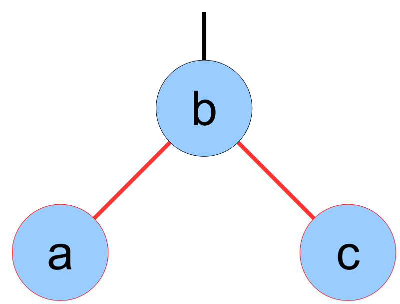
```
private void flipColors(Node h) {
    h.color = !h.color;
    h.left.color = !h.left.color;
    h.right.color = !h.right.color;
}
```

Inserción en 3-node: caso 1

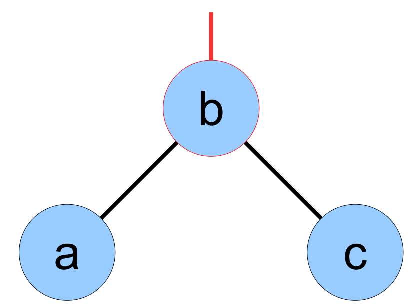
3-node inicial



put(c,value)



flipColors(b)

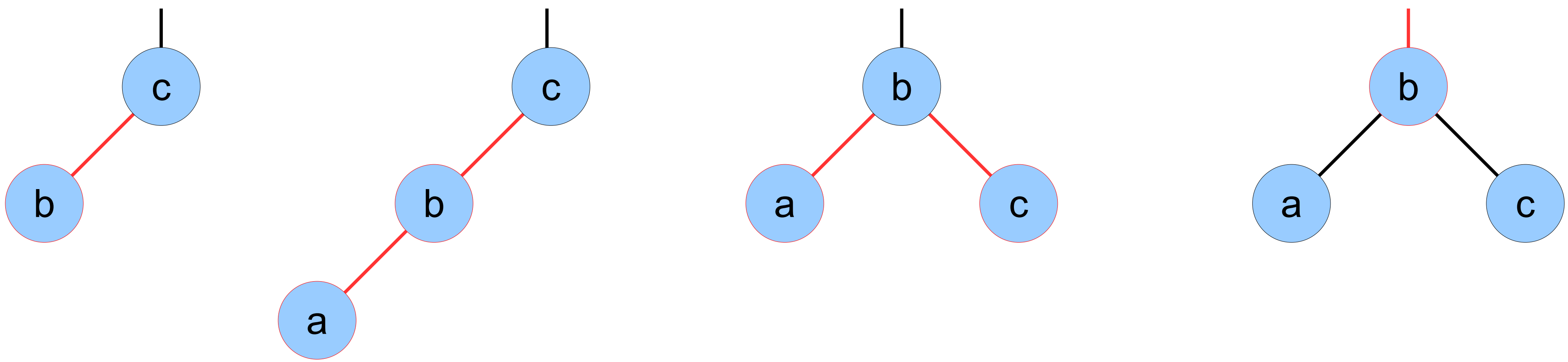


Inserción en un 3-node
Caso 2 : llave menor que las otras dos

- Se agrega como hijo izquierdo con enlace rojo.
- Quedan dos enlaces rojos. Se hace una rotación a la derecha y se convierte en el caso 1.
- Se hace una operación `flipColors` y se restauran las propiedades del árbol.

Inserción en 3-node: caso 2

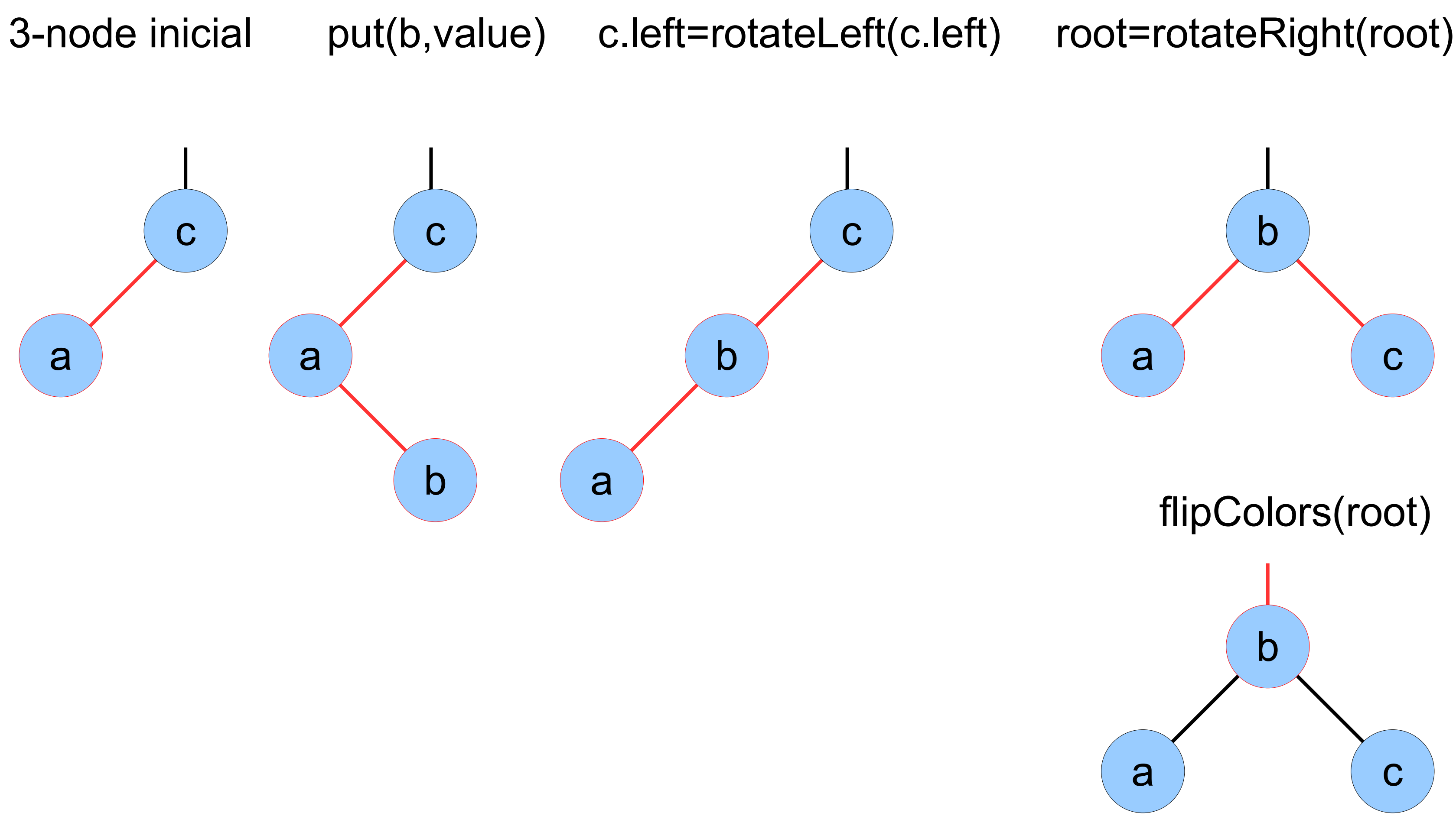
3-node inicial　put(a,value)　root=rotateRight(root)　flipColors(root)



Inserción en un 3-node
Caso 3 : llave en el rango de las otras dos

- Se agrega con enlace rojo a la derecha del hijo izquierdo.
- Se hace rotación izquierda del hijo izquierdo, y luego rotación derecha del padre. Queda de nuevo el caso 1.
- Se restaura con la operación `flipColors`.

Inserción en 3-node: caso 3



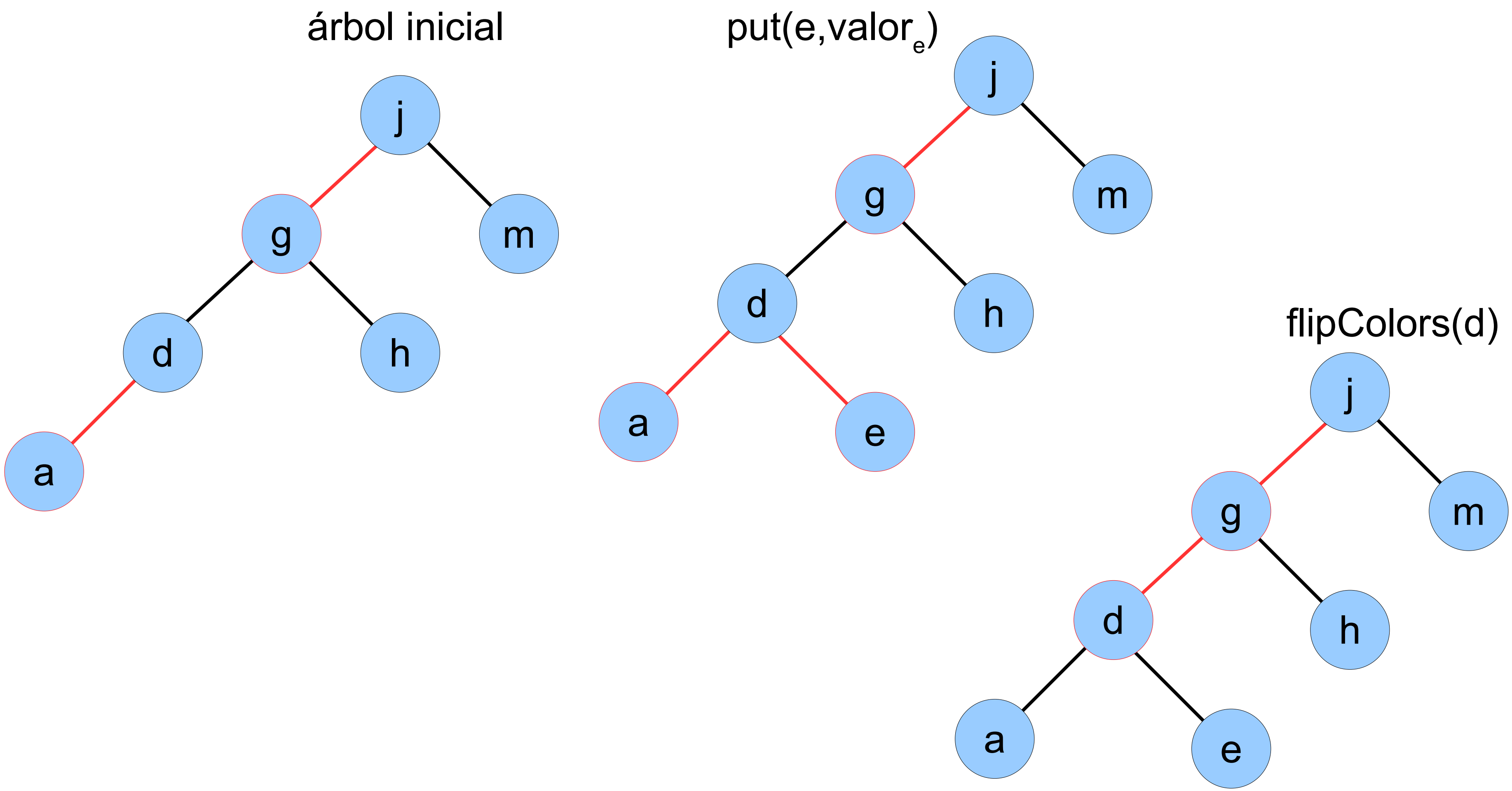
Color de la raíz

- Las operaciones de inserción en un 3-node pueden hacer la raíz roja.
- Observar que se puede hacer la raíz de todo el árbol negra después de cada inserción, sin afectar las características del árbol rojo-negro.

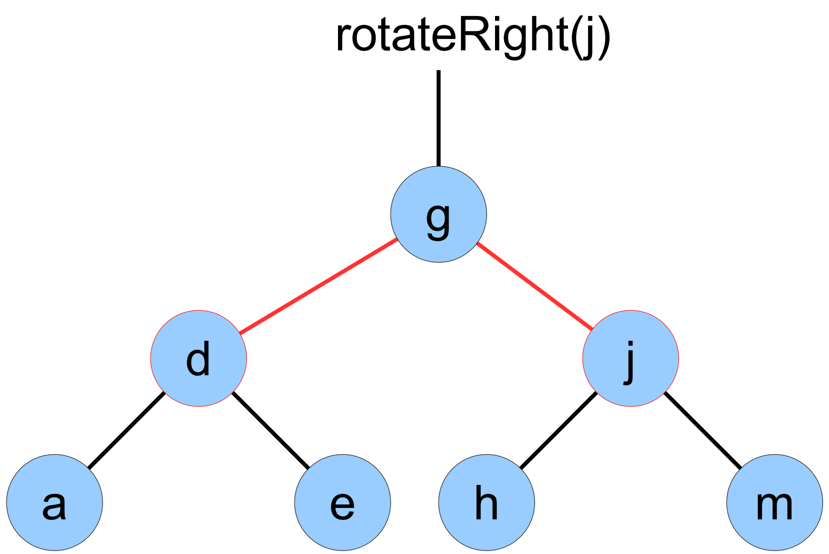
Inserción en nivel inferior

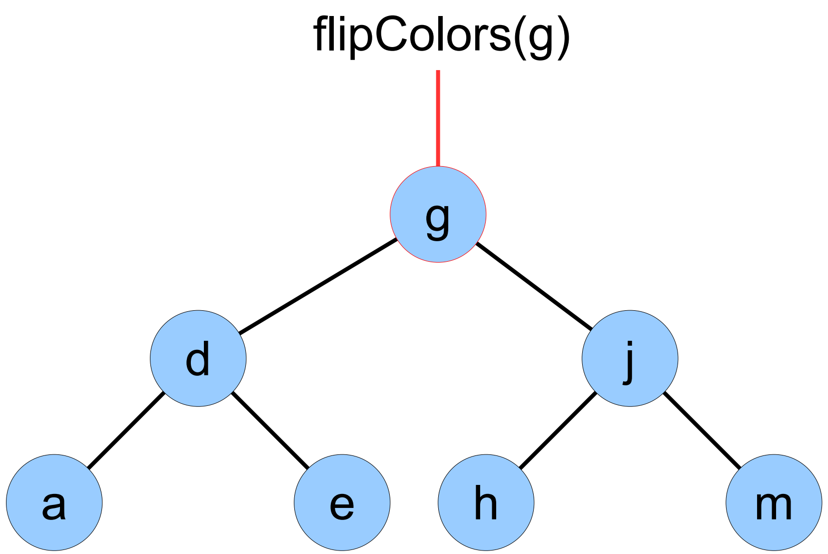
- Cuando se inserta en un nivel inferior, cambia el color del nodo padre.
- Esto puede afectar las propiedades del árbol rojo-negro. En realidad, se presentan los mismos 3 casos de la inserción en un 3-node.
- Se corrigen aplicando exactamente las mismas reglas y propagando los cambios de color recursivamente hacia arriba.

Ejemplo: Inserción nivel inferior



Continuación





Finalmente, la raíz siempre se vuelve negra.

Implementación

```java
public void put(Key key, Value val) {
    if (key == null) throw new NullPointerException("key is null");
    if (val == null) {
        delete(key);
        return;
    }
    root = put(root, key, val);
    root.color = BLACK;
}

private Node put(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val, RED, 1);
    int cmp = key.compareTo(h.key);
    if      (cmp < 0) h.left  = put(h.left,  key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else              h.val   = val;
    // fix-up any right-leaning links
    if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
    if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
    h.N = size(h.left) + size(h.right) + 1;
    return h;
}
```

Borrado en árboles rojo-negros

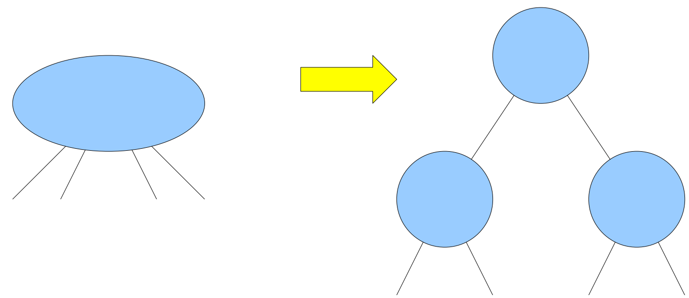
- La operación borrado de los BST no mantiene el invariante del balance perfecto de enlaces negros.
- Se requiere un algoritmo un poco más intrincado porque hay que realizar transformaciones tanto en el recorrido hacia abajo, como en el recorrido hacia arriba.

Árboles 2-3-4

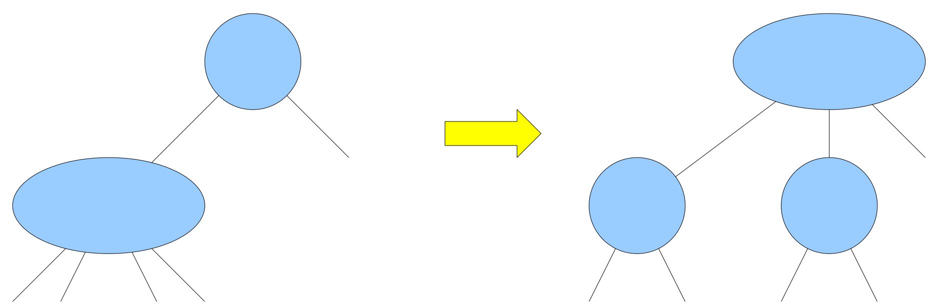
- Similar a los árboles 2-3, pero se mantienen temporalmente los 4-nodes.
- Para insertar una llave, se hace un recorrido hacia abajo, <u>garantizando que el nodo actual no sea un 4-node</u>. Se utilizan las mismas transformaciones ya vistas en árboles 2-3.
- Al llegar a un nulo, siempre se pueden agregar la nueva llave: 2-node→3-node ó 3-node→4-node.

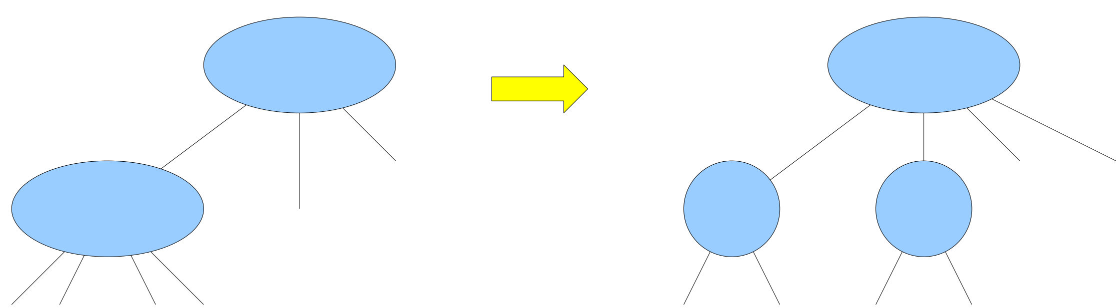
Inserción en un árbol 2-3-4

Raíz



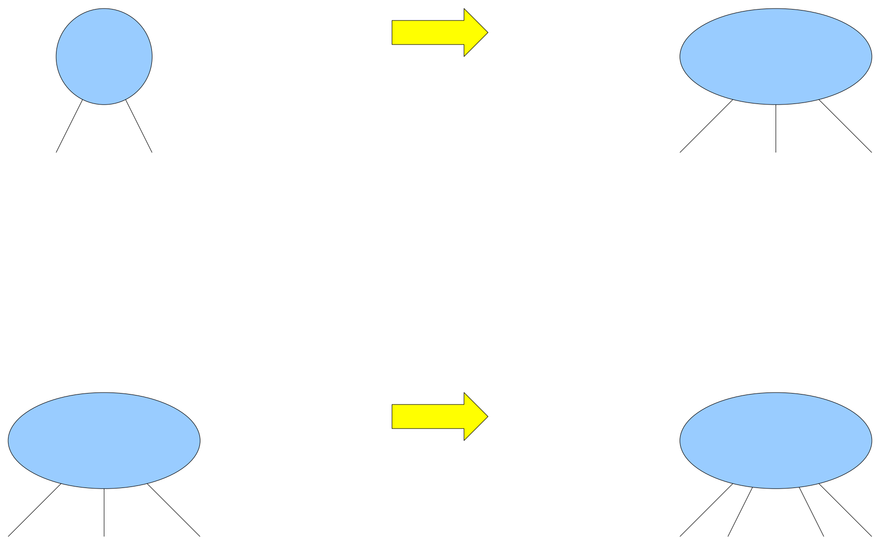
Recorrido hacia abajo





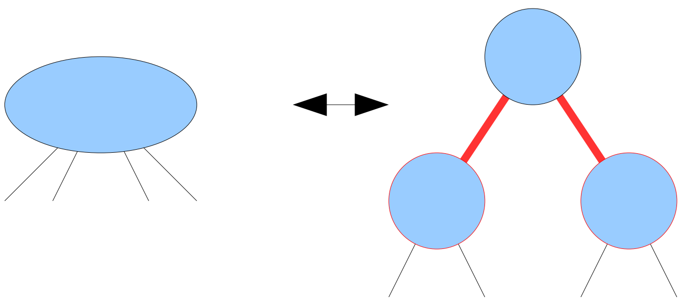
Inserción en un árbol 2-3-4

Inserción en el nivel inferior



Árboles 2-3-4

- Los 4-nodes se representan en el árbol rojo-negro como 3 2-nodes donde ambos hijos tienen enlace rojo.



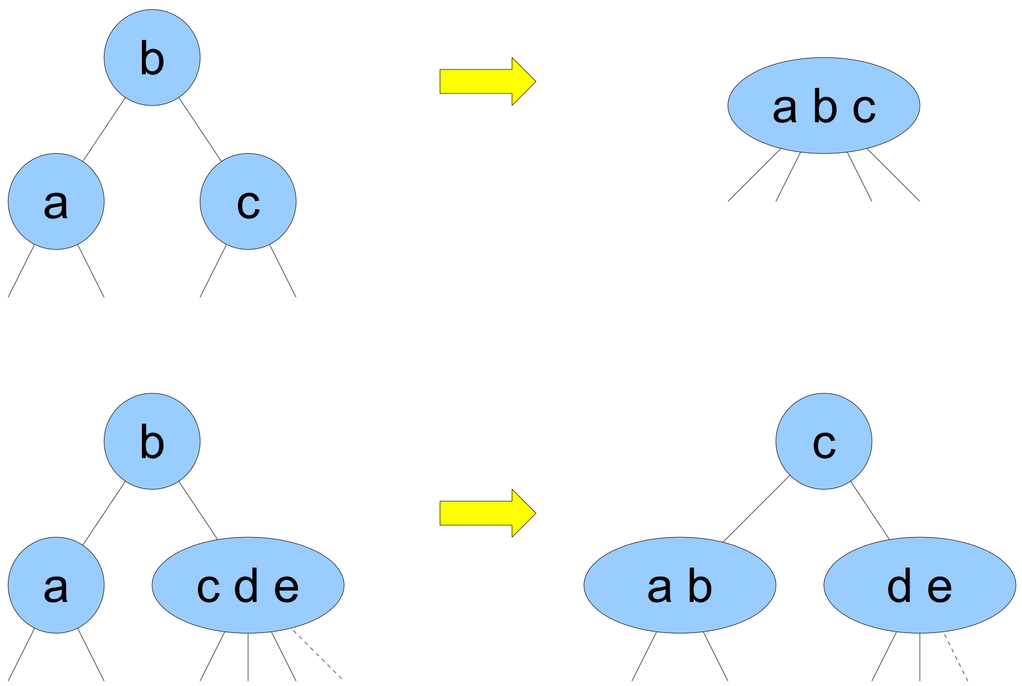
- Se dividen los 4-nodes en el recorrido hacia abajo haciendo flip de color.
- Se balancean los 4-nodes en el recorrido hacia arriba haciendo las mismas rotaciones que para los árboles rojo-negros.

Operación `deleteMin`

- Si el nodo que contiene el mínimo es un 3-node o un 4-node, se puede borrar la llave. No cambia la altura del árbol.
- Si el nodo es un 2-node, no se puede borrar inmediatamente pues invalida la propiedad de balance del árbol. Se debe hacer un recorrido hacia abajo, para transformar el nodo izquierdo en un 3-node ó un 4-node.

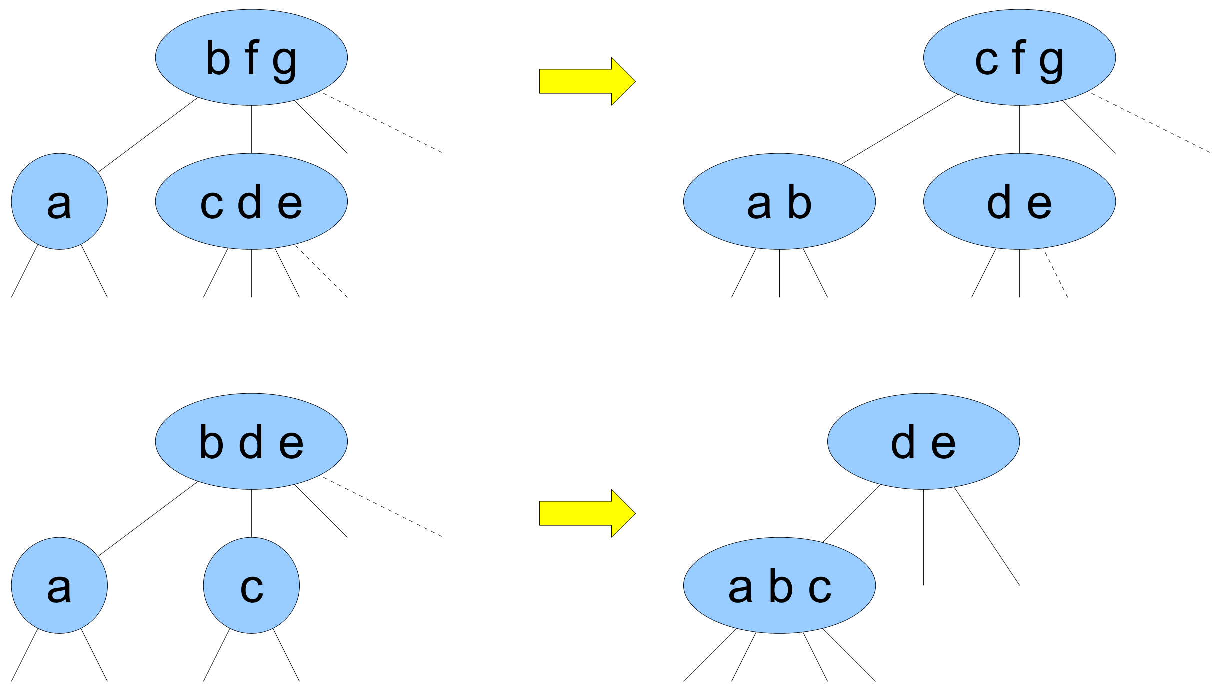
Transformaciones para borrar mínimo

En la raíz



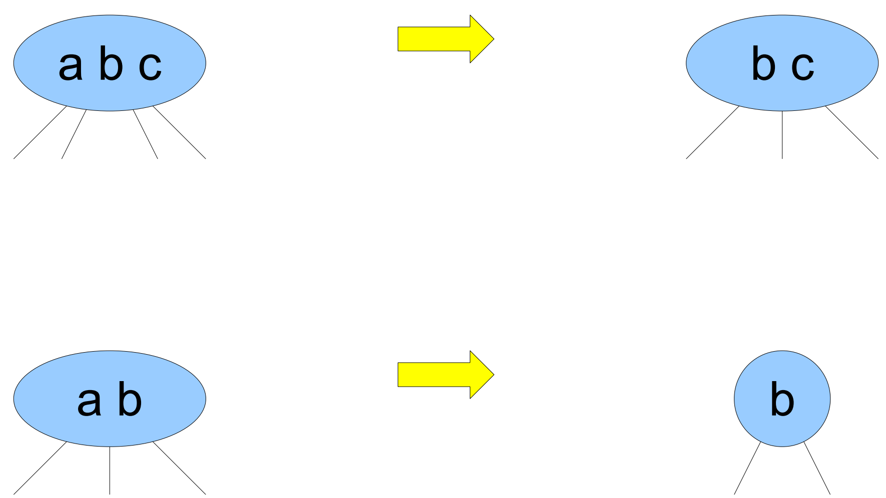
Transformaciones para borrar mínimo

En el recorrido hacia abajo



Transformaciones para borrar mínimo

En el nivel inferior



Operación `delete`

- Las mismas transformaciones consideradas en el caso de `deleteMin`.
- Se ajustan para hacer las transformaciones en el camino descendente hacia el nodo a borrar.

Altura del árbol rojo-negro

Proposición

La altura de un árbol rojo-negro de N nodos es como máximo ~2 lg(N).

Esto es incluyendo incluso los enlaces rojos.

La altura promedio de un árbol rojo-negro es ~lg(N).

Eficiencia de operaciones

- En el árbol rojo-negro el algoritmo get es exactamente el mismo del BST. Siendo la altura del árbol ~2 lg(N), este sería a su vez el número de comparaciones de peor caso.
- En el caso medio, sería ~lg(N).

Otras operaciones

- Operaciones como put, deleteMin, delete, min, max, floor, ceil, select, rank hacen como máximo un número constante de recorridos de la altura del árbol: ~K lg(N).

Comparativo bibliotecas

Texto guía RedBlackBST<Key extends Comparable<Key>, Value>	**Bibliotecas Java** java.util.TreeMap<K,V>
Value get(k)	V get(k)
void put(k,v)	void put(k,v)
void delete(k)	V remove(k)
int size() boolean isEmpty()	int size() boolean isEmpty() void clear()
Key min() Key max()	K firstKey() K lastKey()
Key floor() Key ceil()	K ceilingKey(k) K floorKey(k)
	K lowerKey(k) K higherKey(k)
int rank(k)	
Key select(i)	
int size(lo,hi)	
Iterable<Key> keys()	Set<K> keySet() Collection<V> values()
Iterable<Key> keys(lo,hi)	SortedMap<K,V> subMap(from,to)