Árvores rubro-negras

Referências

- BSTs rubro-negras (PF),
- Balanced Search Trees (S&W),
- slides (S&W)

Vídeo

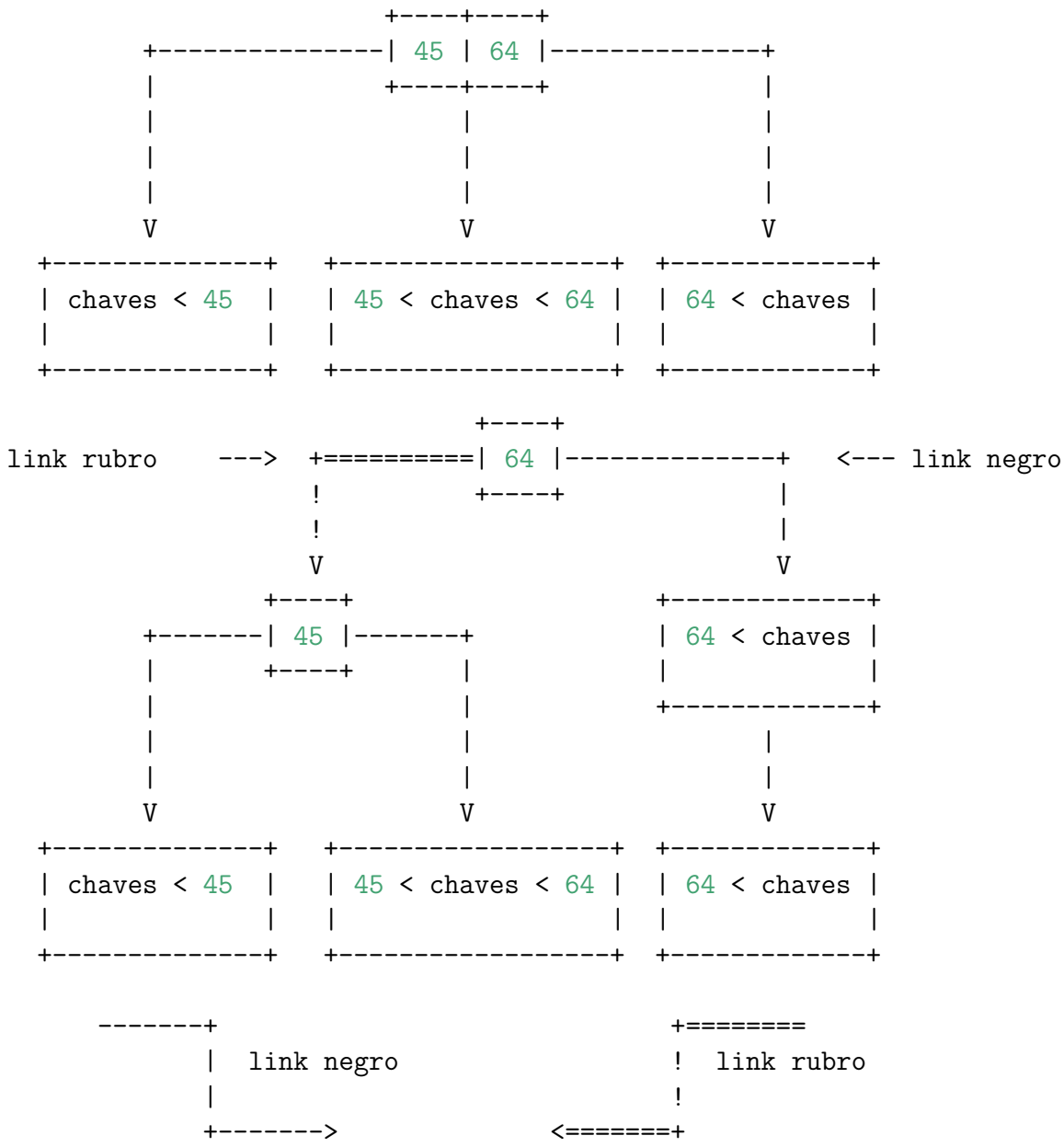
Balanced Search Trees (S&W)

BSTs rubro-negras

Uma BST rubro-negra (*red-black BST*) é uma BST que simula uma árvores 2-3.

Cada 3-nó da árvore 2-3 é representado por dois 2-nós ligados por um link rubro.

Nossas BSTs são esquerdistas (*left-leaning*), pois os links rubros são **sempre** inclinados para a esquerda.

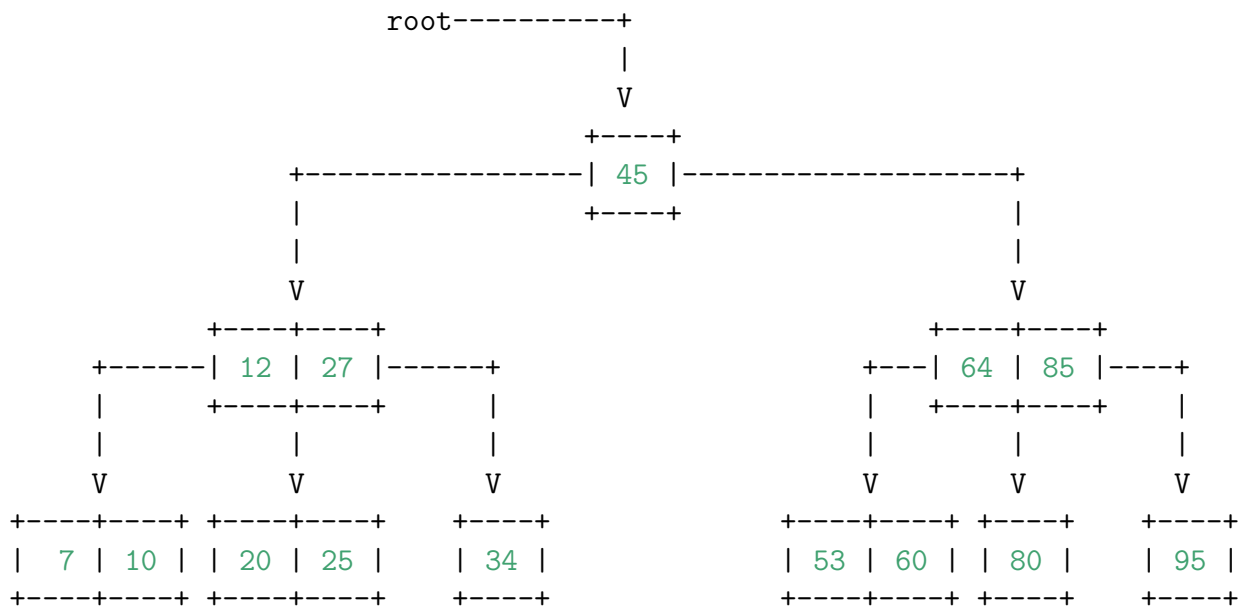


Definição: uma **BST rubro-negra** é uma BST cujos links são negros e rubros e têm as seguintes popriedades:

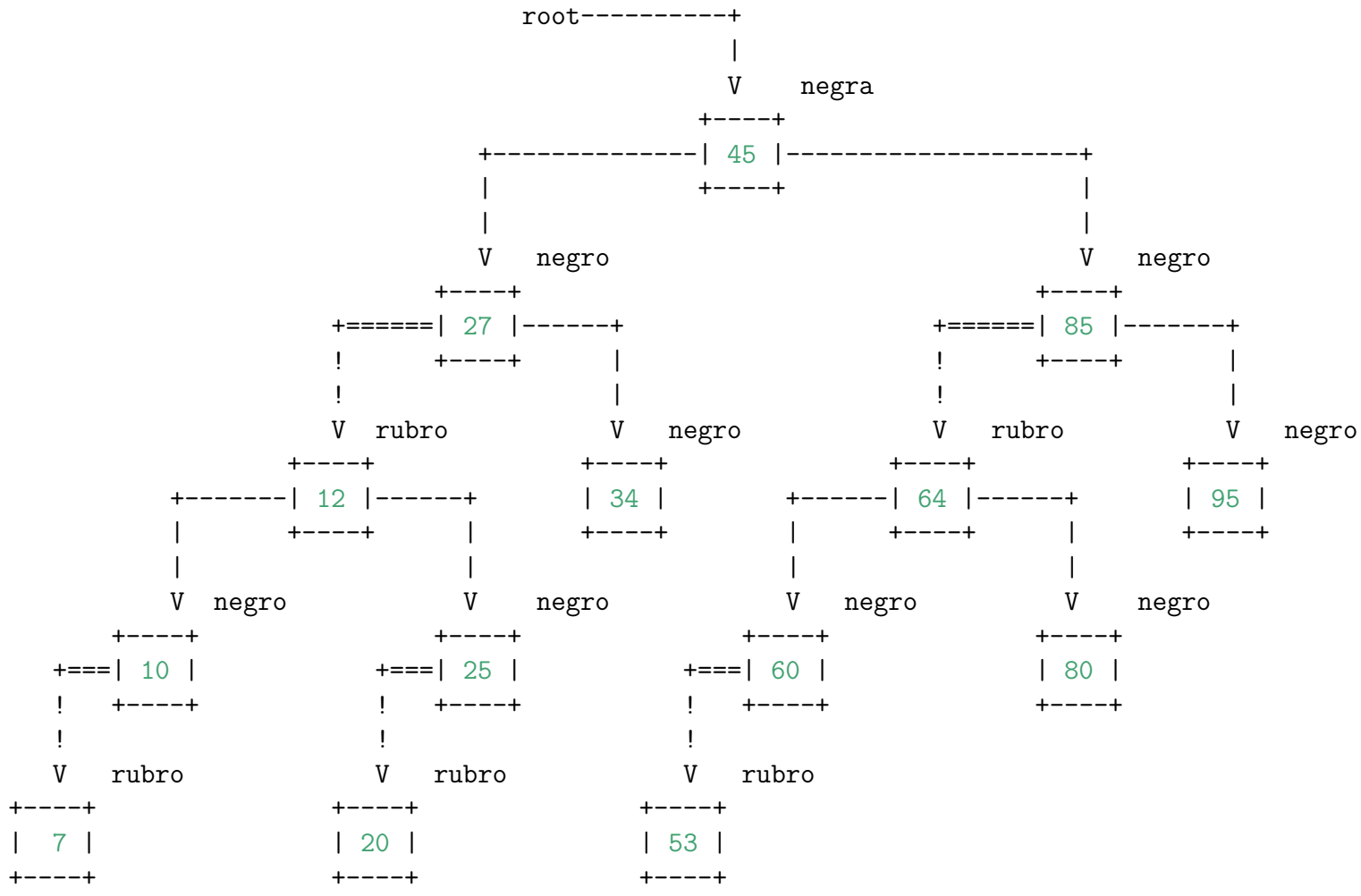
- links rubros se inclinam para a esquerda;
- nenhum nó incide em dois links rubros;
- balanceamento negro perfeito: todo caminho da raiz até um link null tem o mesmo número de links negros.

Se os links rubros forem desenhados horizontalmente e depois contraídos, teremos uma árvore 2-3:

Árvore 2-3



Árvore rubro-negra



O balanço negro perfeito vem do fato de que os links negros correspondem aos links da árvore 2-3.

Nota. No CLRS as [arvores rubro-negras têm nós rubros e negros:

- nós rubros são os referenciados por links rubros.
- nós negros são os referenciados por links negros.

A **profundidade negra** de um nó `x` é o número de links negros no caminho da raiz até x.

A **altura negra** da árvore é o máximo da profundidade negra de todos os nós.

Representação dos nós de uma BST rubro-negra

É inconveniente armazenar a cor de um link na estrutura de dados; é mais simples armazenar essa informação nos nós. A cor de um nó é a cor do único link que entra nele. A raiz é considerada negra.

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

private class Node {
   Key     key;
   Value   val;
   Node    left, right;
   int     n           // número de nós nesta subárvore
   boolean color;     // cor do link que aponta para este nó

   Node(Key key, Value val, int n, boolean color) {
      this.key   = key;
      this.val   = val;
      this.N     = N;
      this.color = color;
   }
}

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

Buscas

O código de busca (= `get()`) para BSTs rubro-negras é exatamente igual ao das BSTs comuns!

```
public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
    return get(root, key);
}

// value associated with the given key in subtree rooted at x; null if no such key
private Value get(Node x, Key key) {
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else              return x.val;
    }
    return null;
}
```

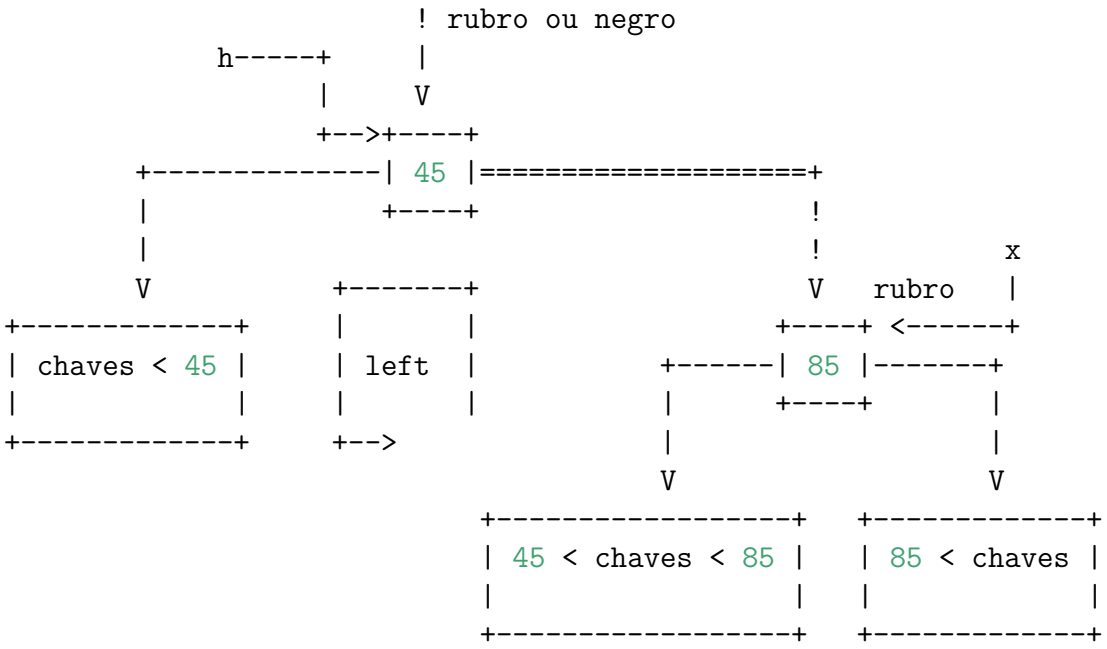
Rotações

O código de inserção (= `put()`) é complicado; ele depende de operações de rotação.

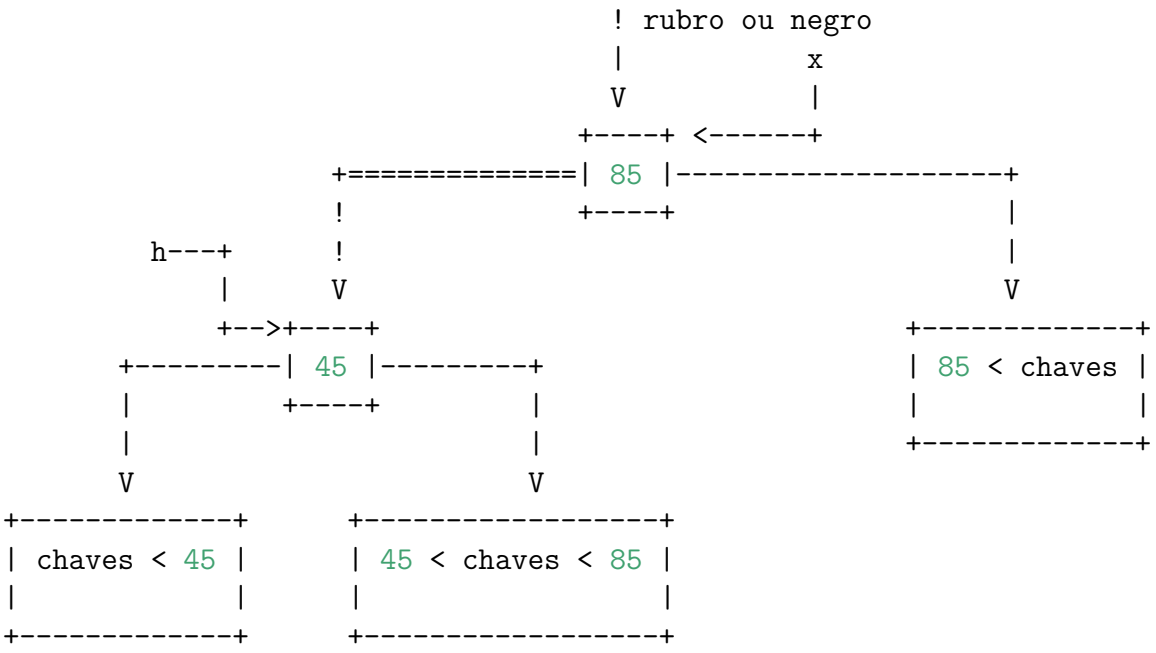
Veja o este vídeo de S&W que ilustra a inserção da sequência de chaves `S E A R C H E X A M P L E` em uma BST rubro-negra.

Durante uma operação de inserção, podemos ter, temporariamente, um link rubro inclinado para a direita ou dois links rubros incidindo no mesmo nó. Para corrigir isso, usamos rotações e *flipping colors*.

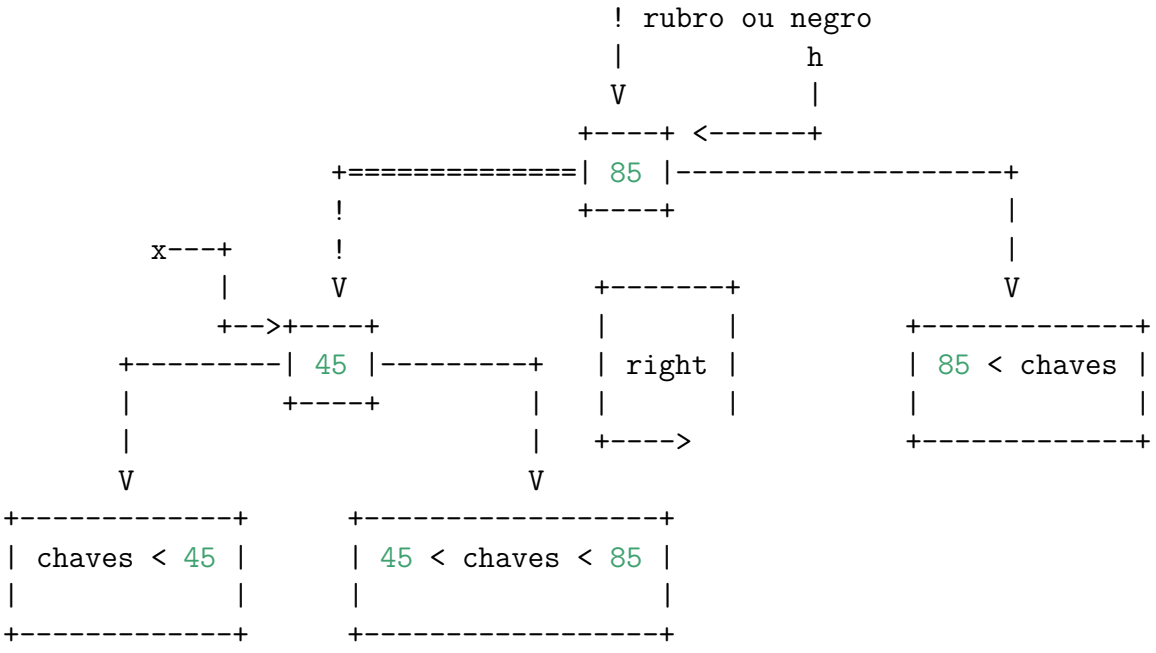
Rotação esquerda (ou horária) em torno de um nó `h`: o filho direito de `h` "sobe" e adota `h` como seu filho esquerdo. Continuamos tendo uma BST com os mesmos nós, mas raiz diferente:



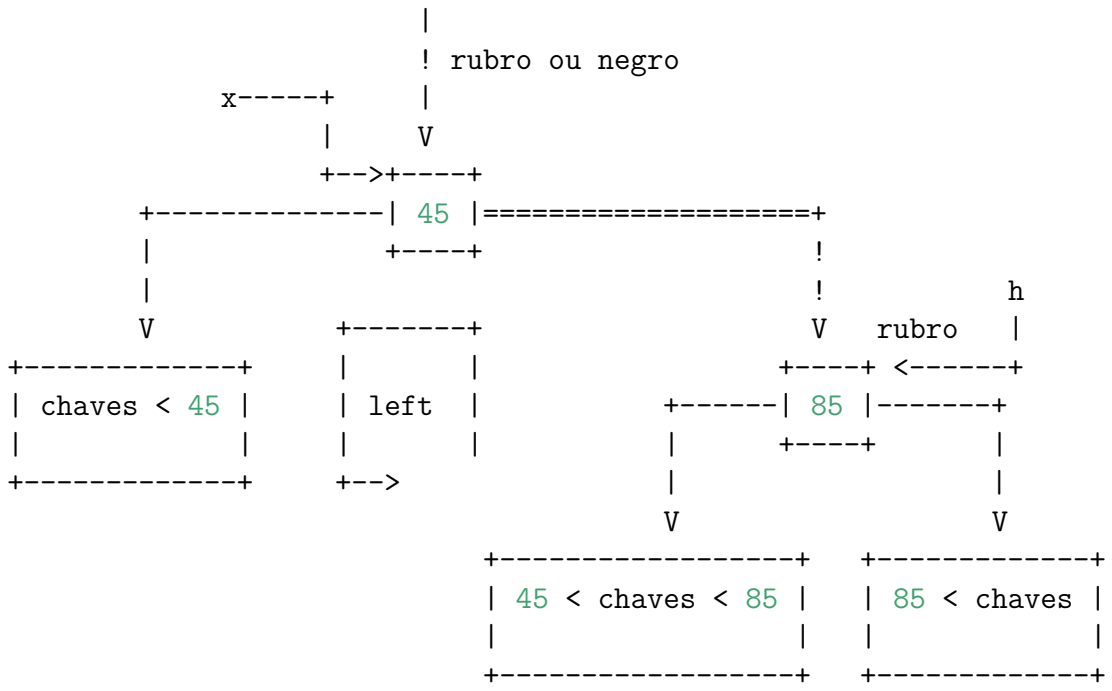

```
private Node rotateLeft(Node h) {
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left) + size(h.right);
    return x;
 }
```



Rotação direita (ou horária) em torno de um nó `h`: o filho esquerdo de `h` "sobe" e adota `h` como seu filho direito. Continuamos tendo uma BST com os mesmos nós, mas raiz diferente:




```
  private Node rotateRight(Node h) {
     Node x = h.left;
     h.left = x.right;
     x.right = h;
     x.color = h.color;
     h.color = RED;
     x.N = h.N;
     h.N = 1 + size(h.left) + size(h.right);
     return x;
  }
```



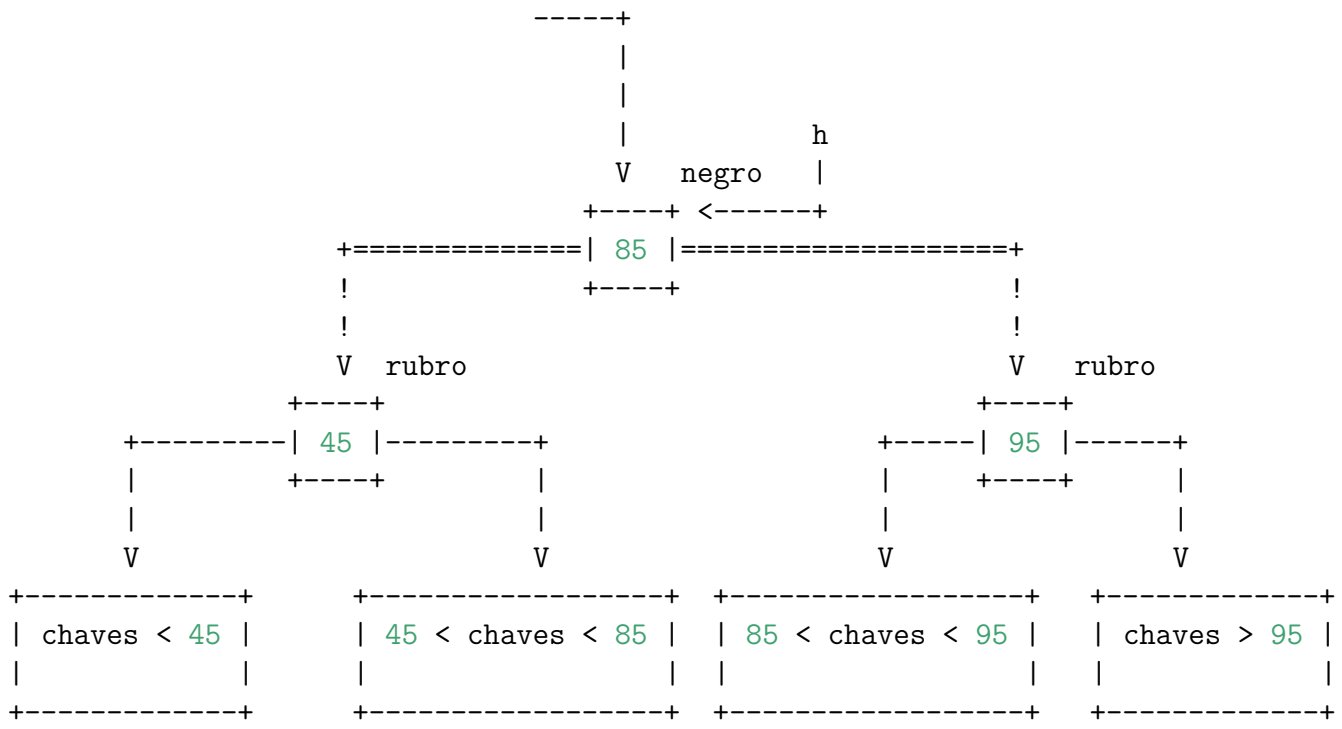
As operações de rotação são locais.

Depois de uma rotação, continuamos tendo uma BST com balanceamento negro perfeito.

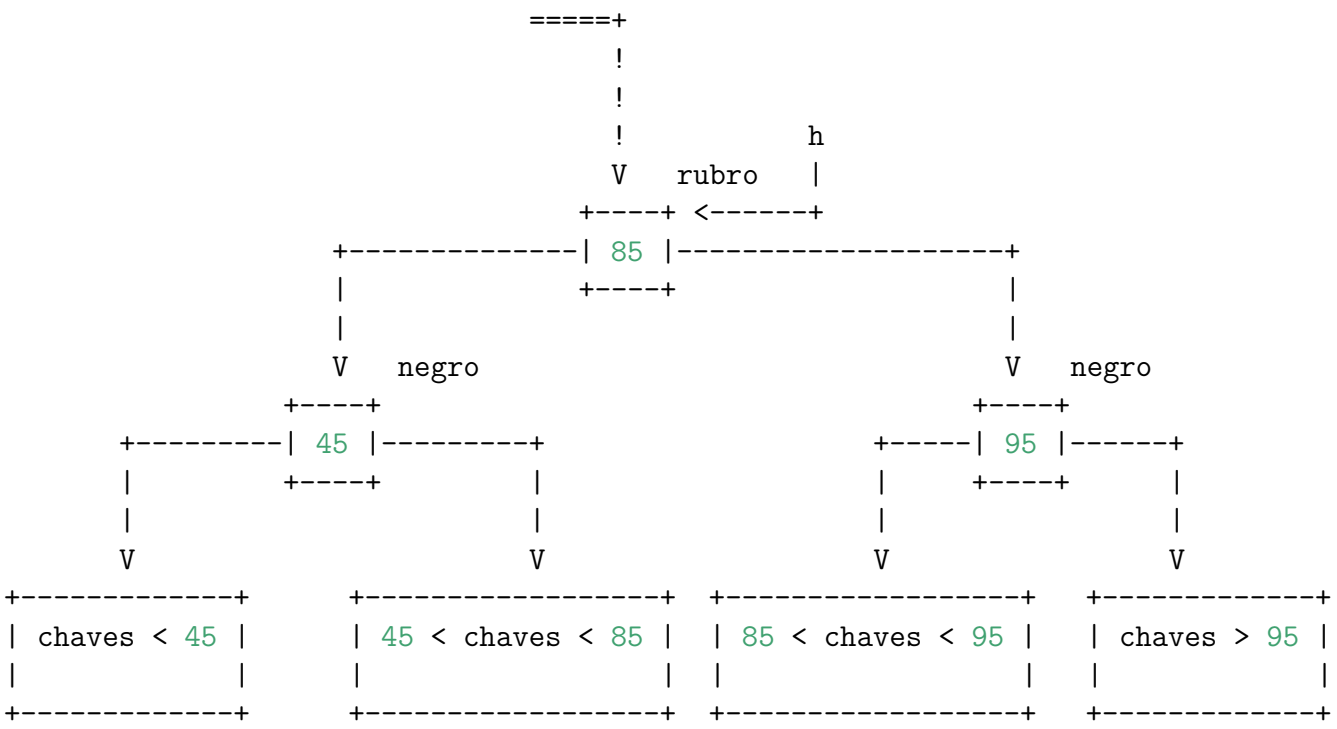
Mas a operação pode ter criado um link rubro inclinado para a lado errado ou dois links rubros seguidos. Isso deverá ser corrigido.

Flipping colors

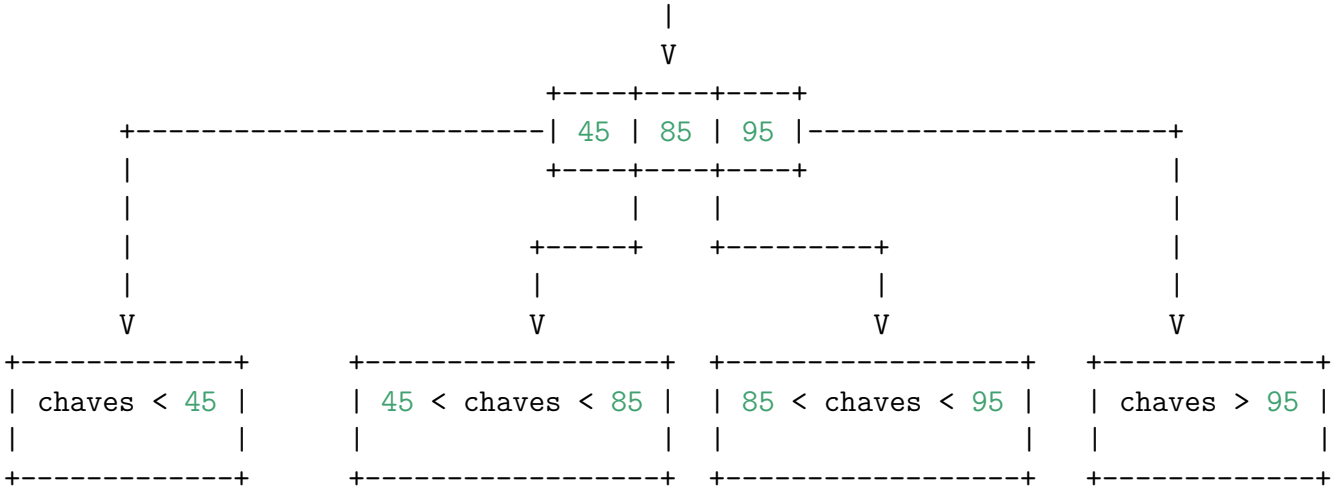
Na árvore 2-3 a operação de *flipping colors* corresponderá a espatifar um 4-nó e subir a chave do meio para o nó pai.

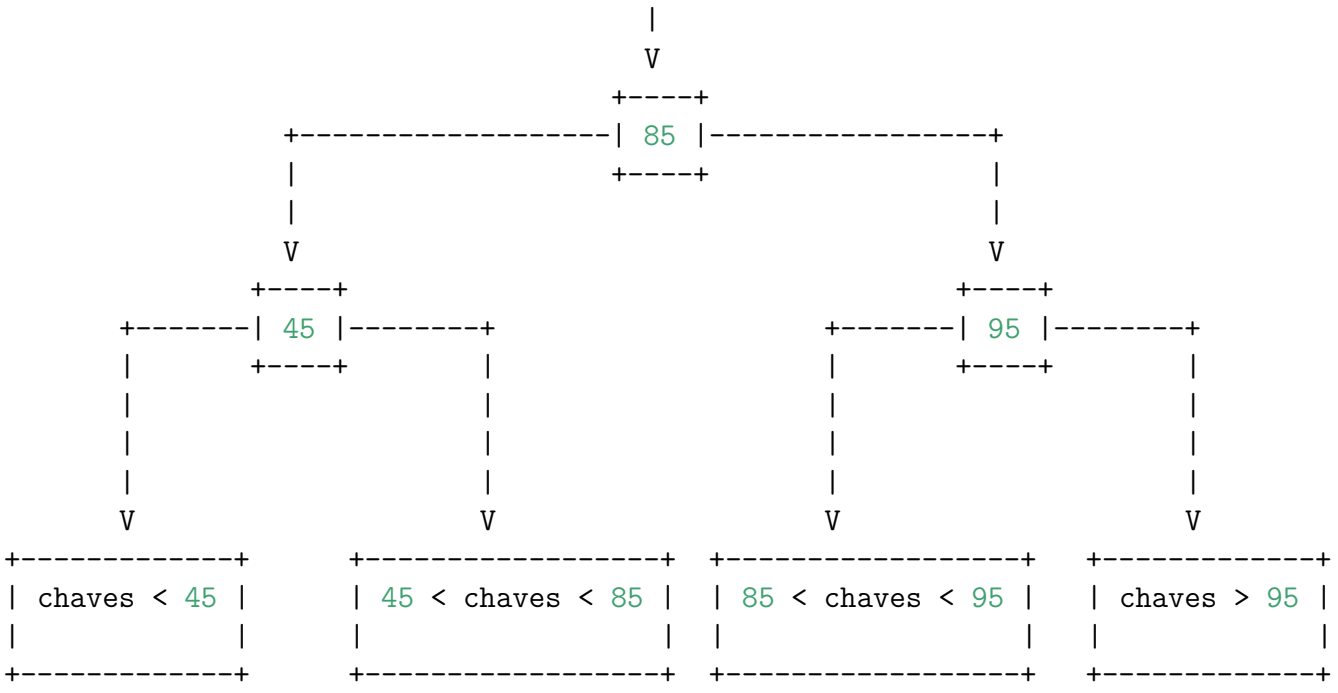


```
private void flipColor(Node h) {
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```



Na árvore 2-3 temos



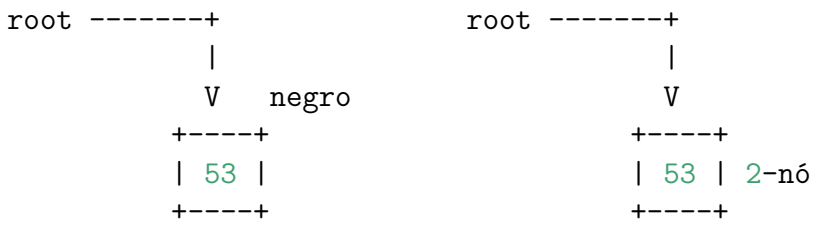


(SW 3.3.38, p.452) **Teorema fundamental das rotações.** Mostre que qualquer BST pode ser transformada em qualquer outra sobre o mesmo conjunto de chaves por uma sequência apropriada de rotações.

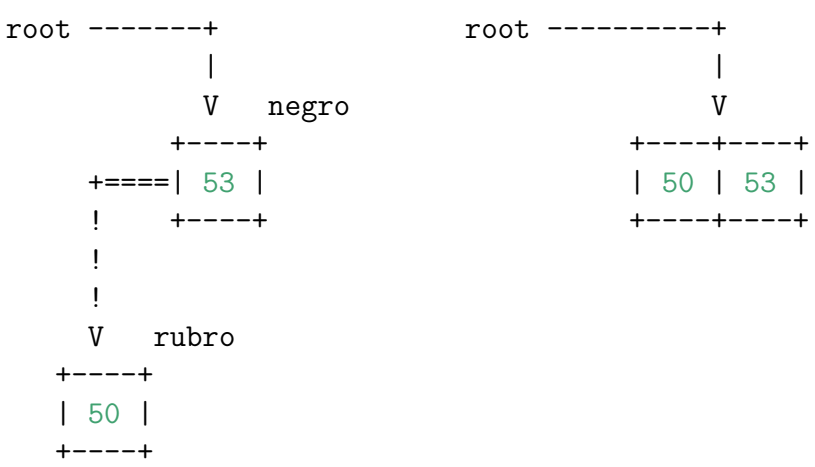
Inserção

Inserção em um 2-nó

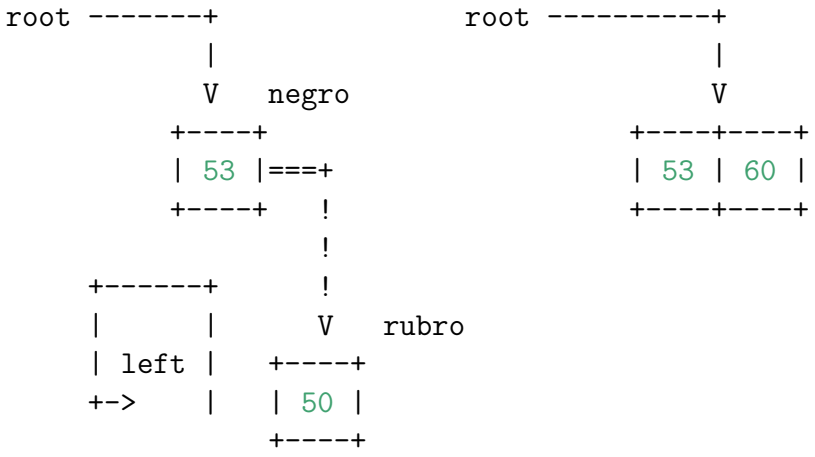
Considere que a árvore é formada por apenas um 2-nó



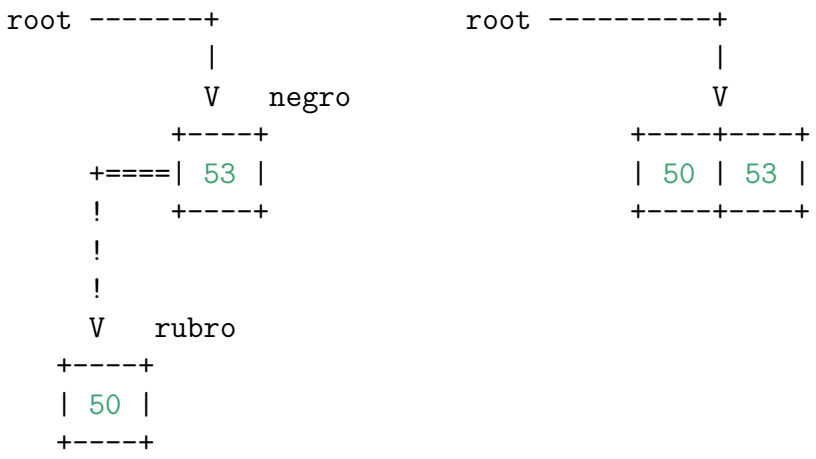

```
put(50)
```



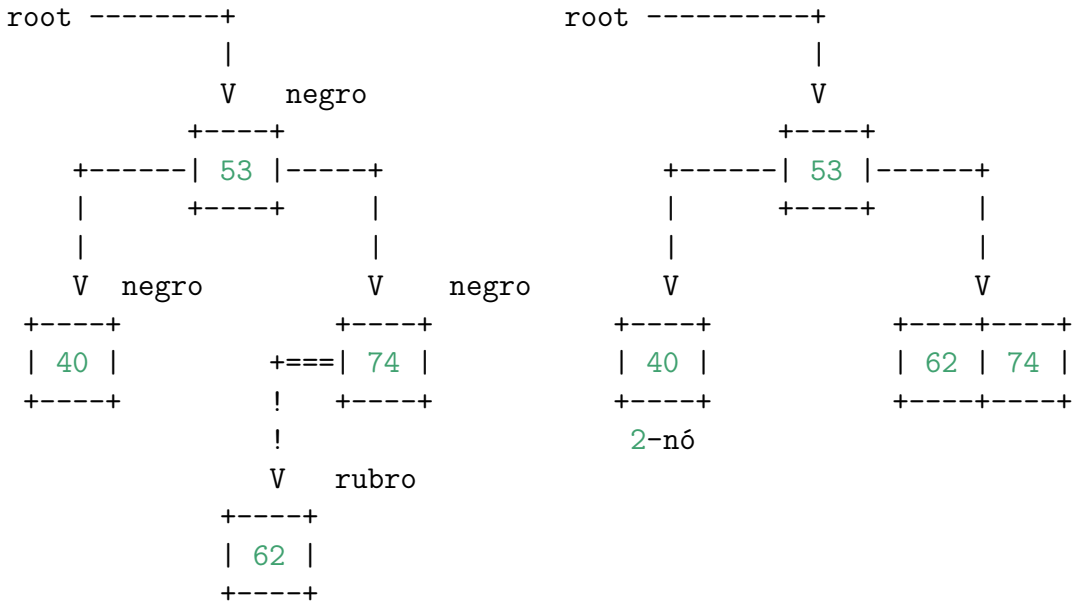

```
put(60)
```



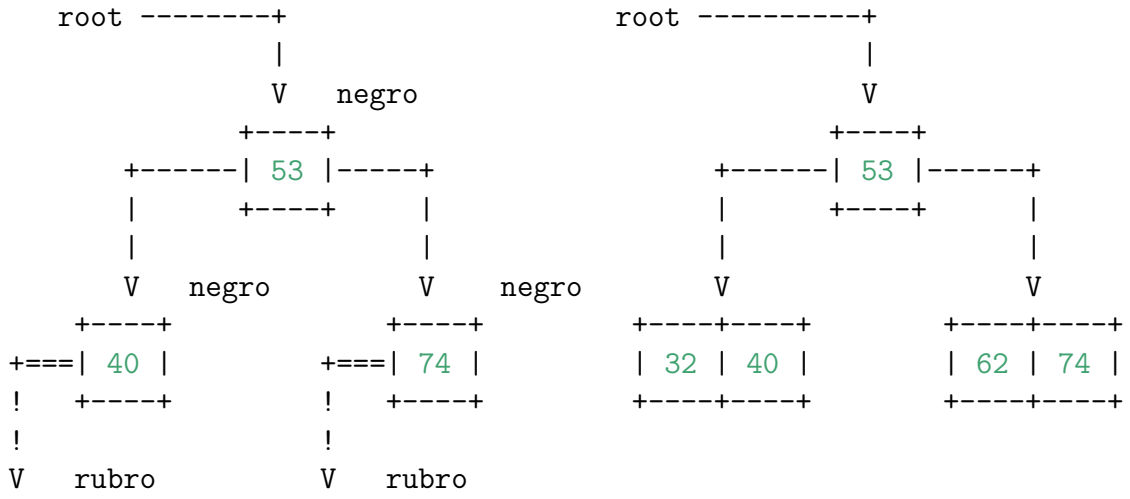

```
root = rotateLeft(root);
```



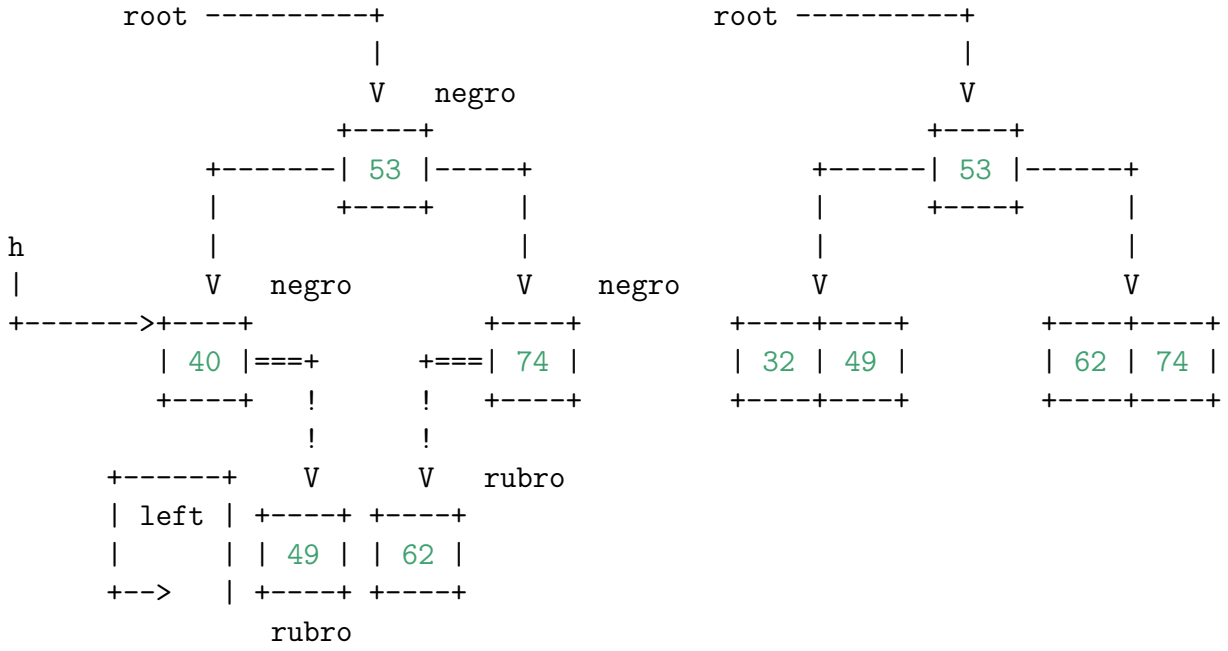
Inserção em um 2-nó qualquer



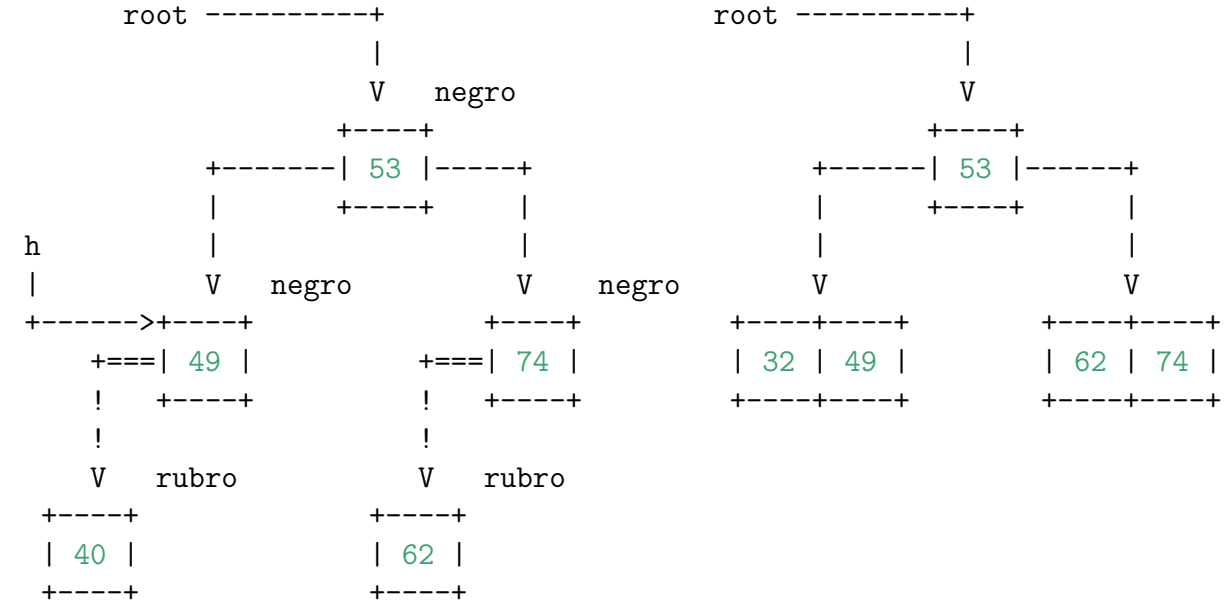

```
put(32)
```




```
put(49)
```

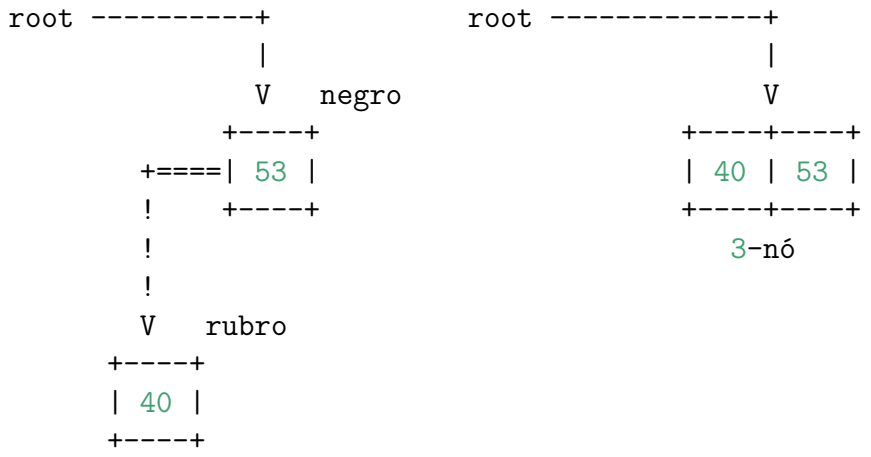



```
h = rotateLeft(h);
```



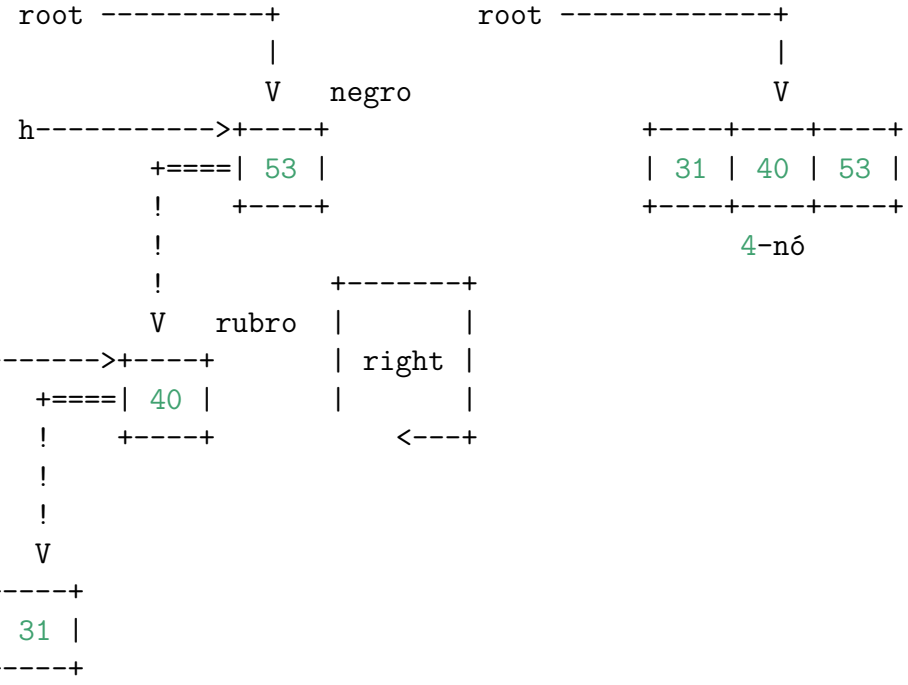
Inserção em um 3-nó

Árvore 2-3 consiste apenas de um nó

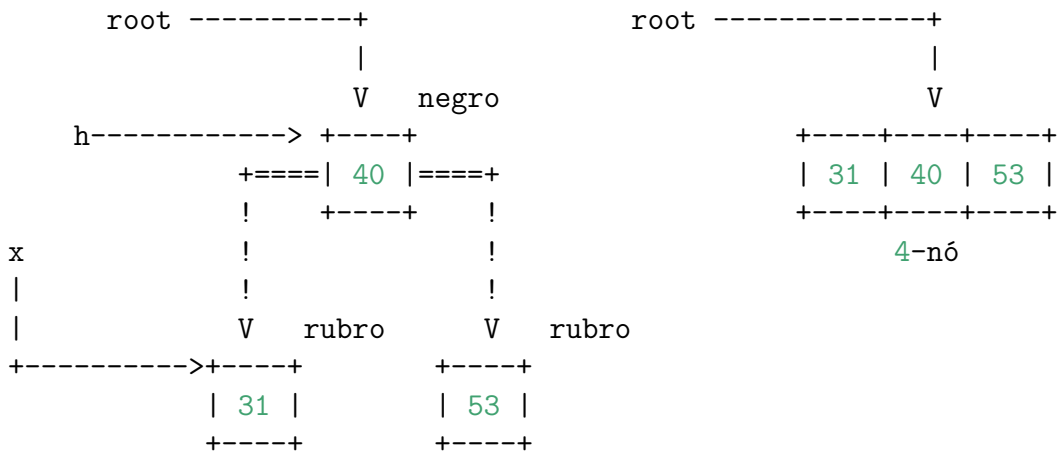


Caso 1. chave é inserida é menor do 3-nó

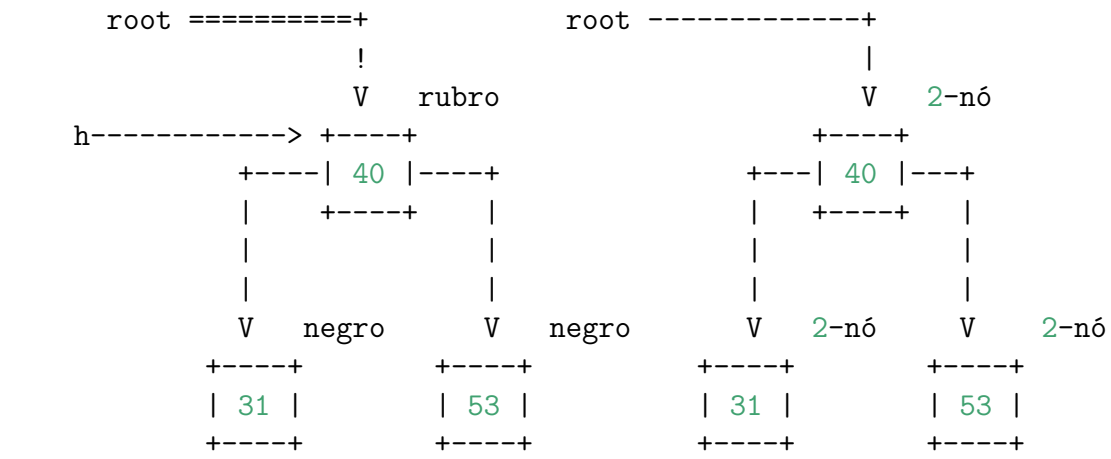
```
put(31)
```




```
x = rotateRight(x);
```

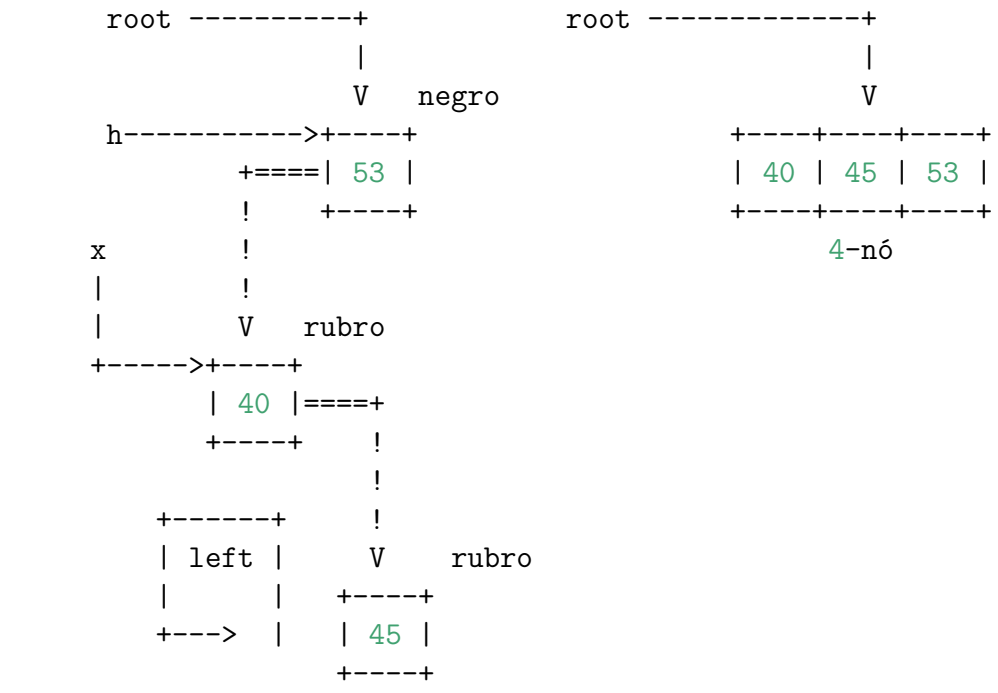



```
flipColors(h);
```

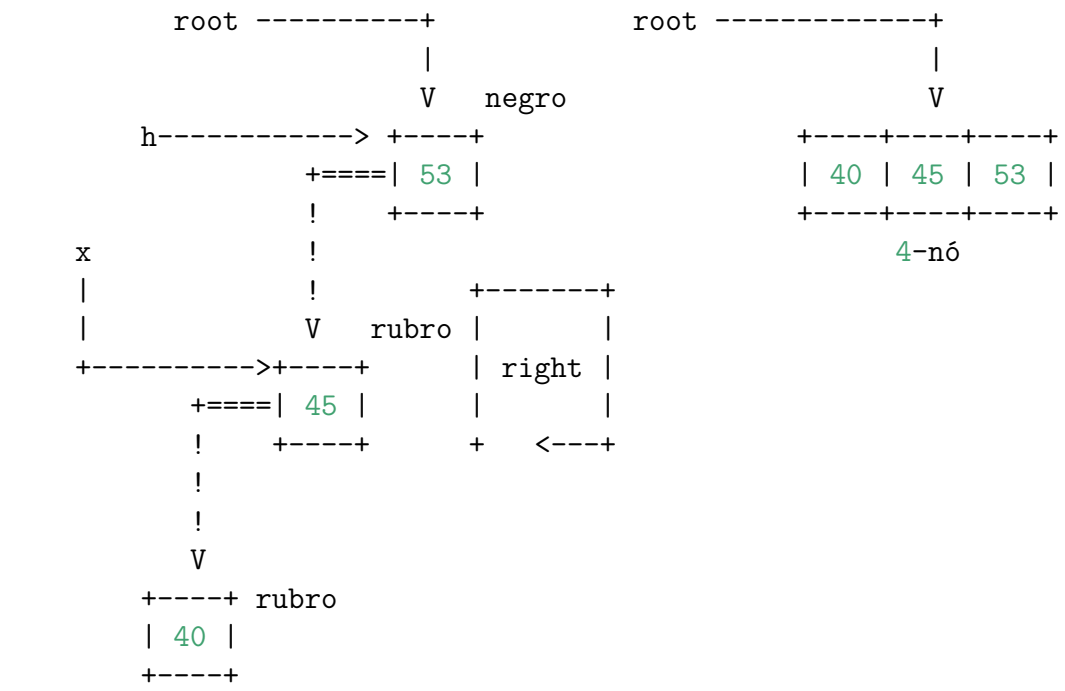


Caso 2. chave é inserida entre as chaves do 3-nó

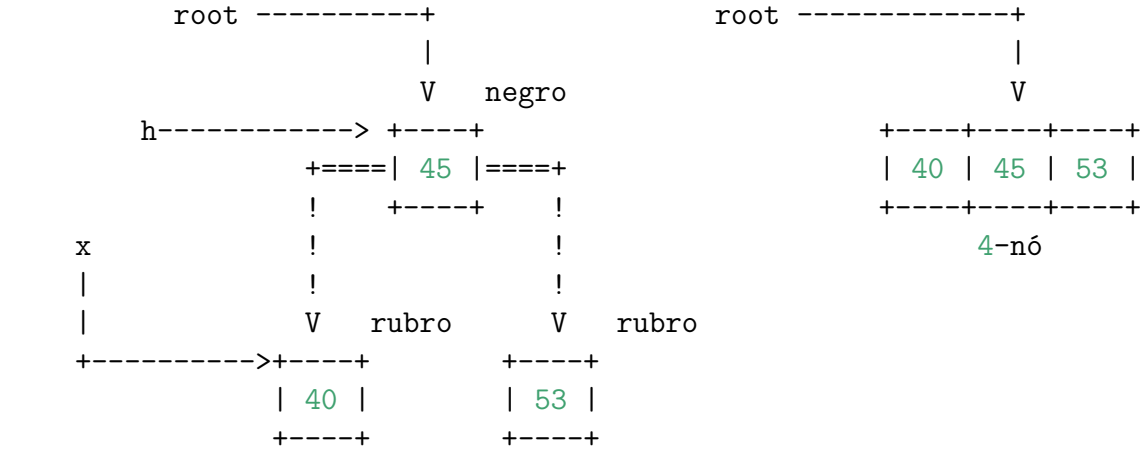
```
put(45)
```



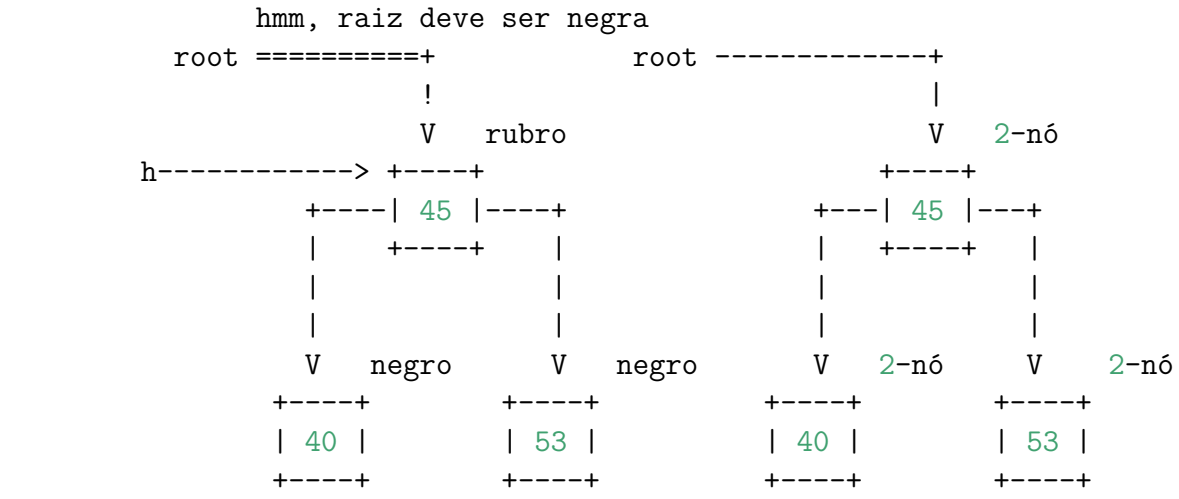

```
x = rotateLeft(x);
```




```
h = rotateRight(h);
```

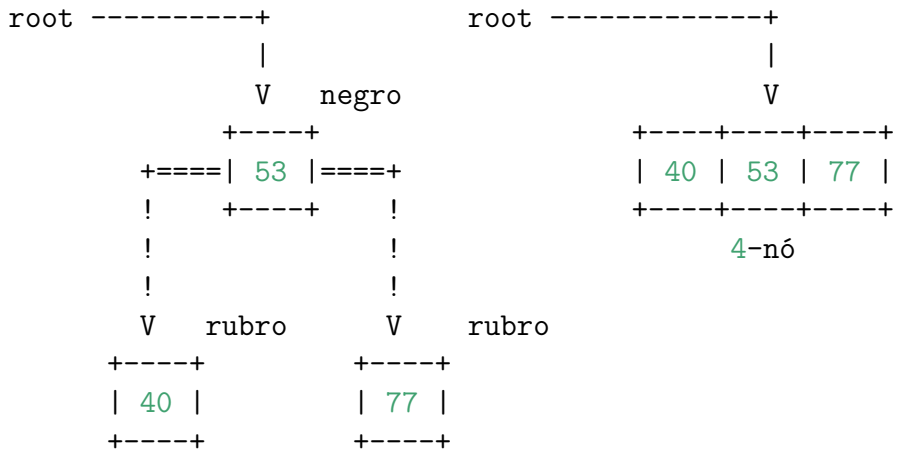



```
flipColors(h);
```



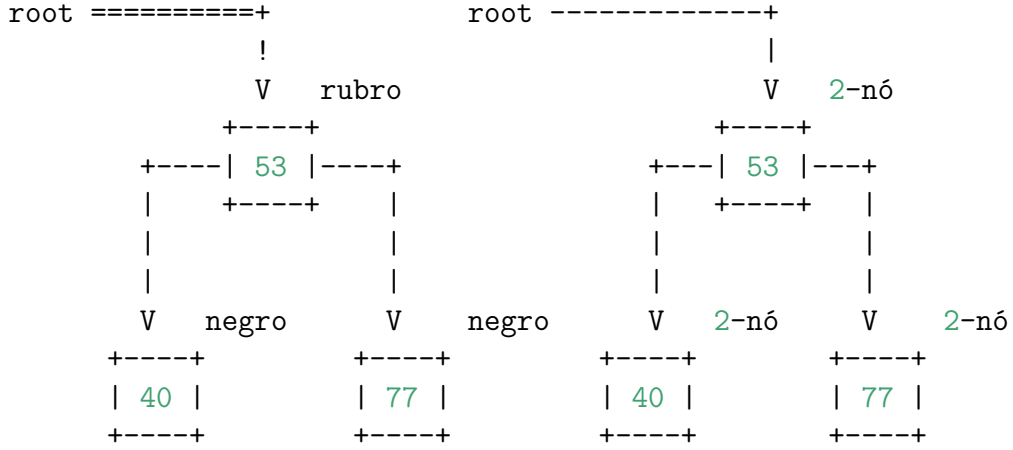
Caso 3. chave inserida é maior que todas as chaves no 3-nó

`put(77)`

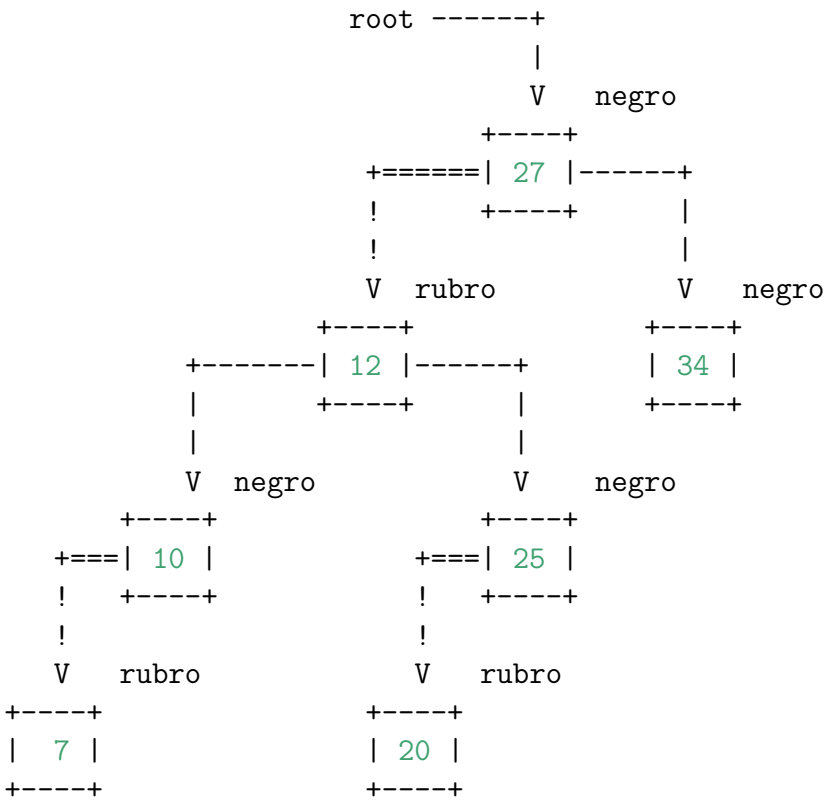


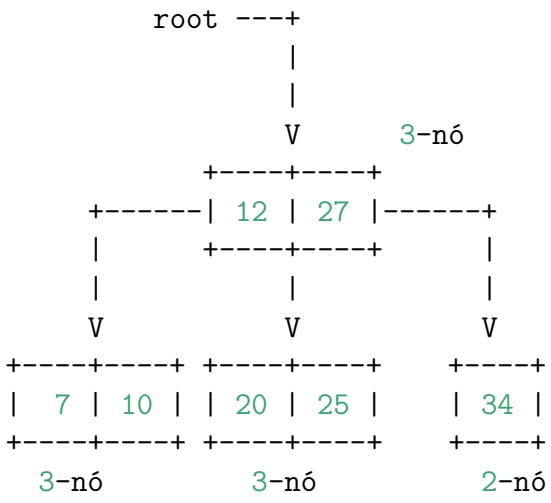

```
flipColors(root);
```

hmmm. raiz deve ser negra

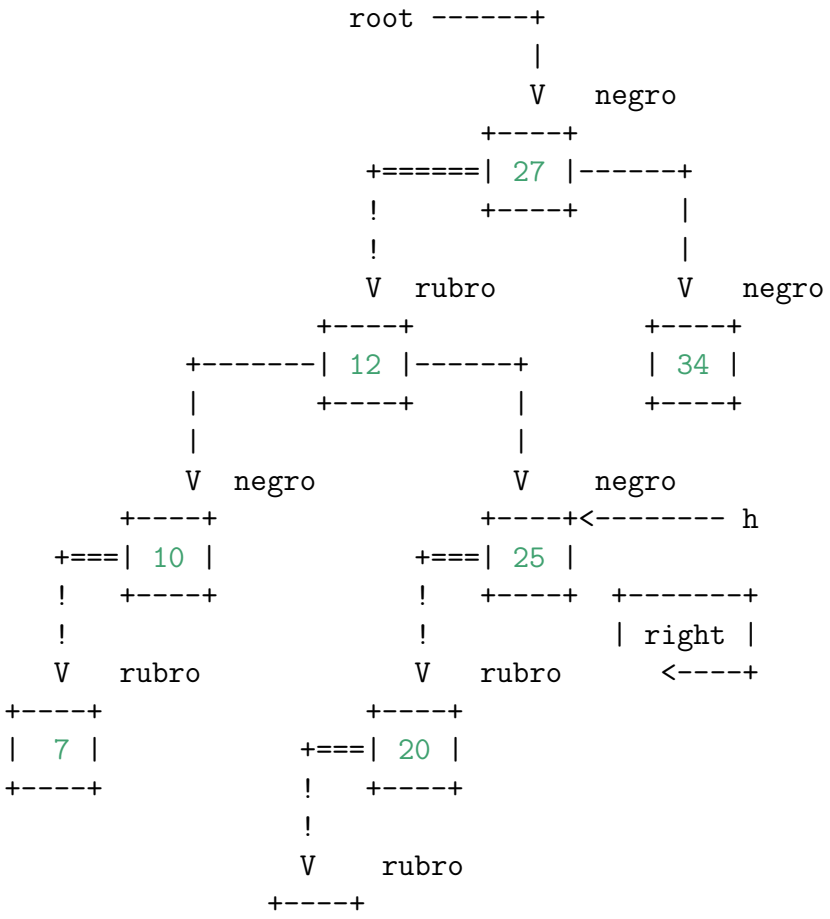


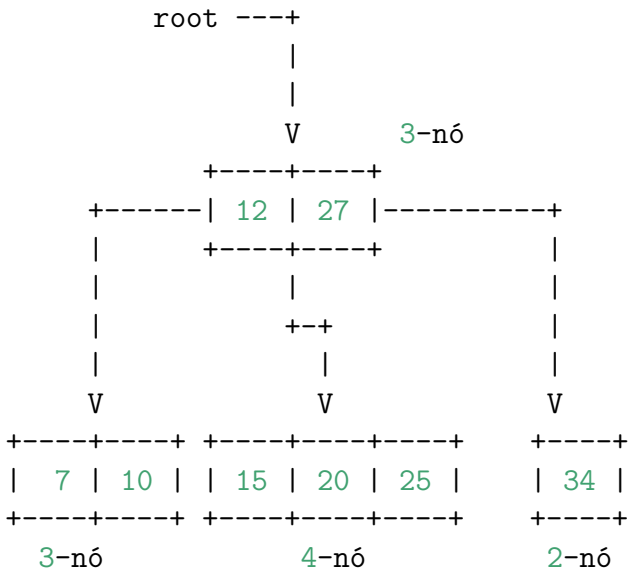
Inserção em um 3-nó qualquer

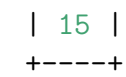




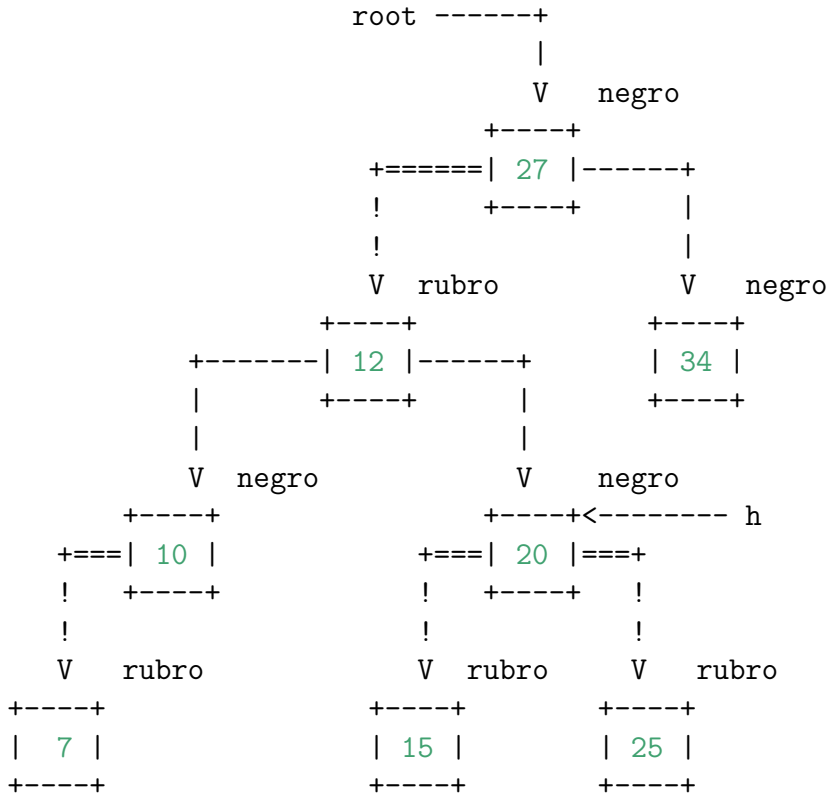

```
put(15)
```

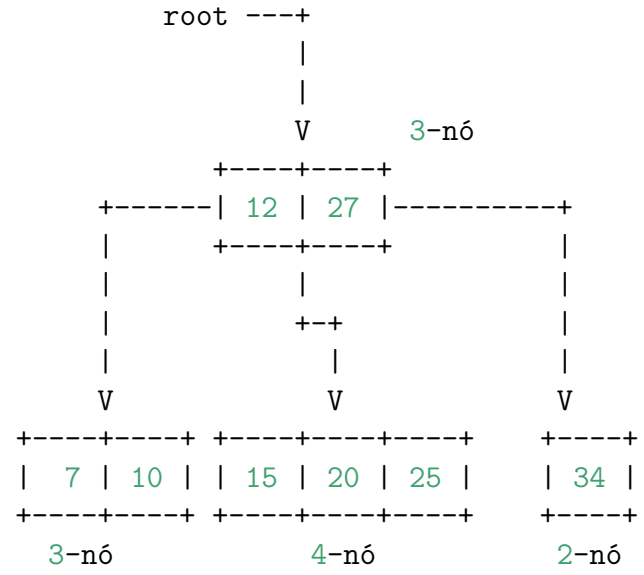




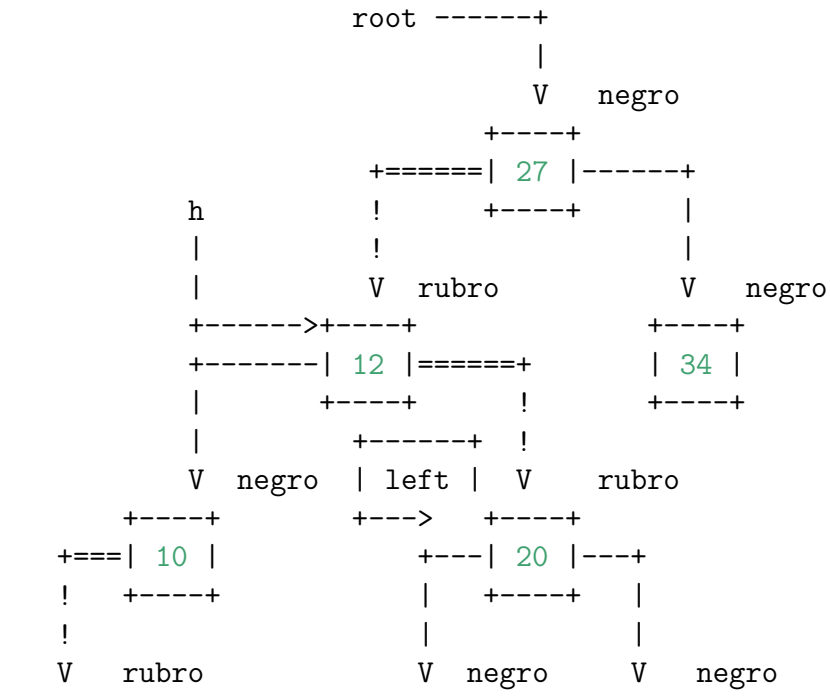


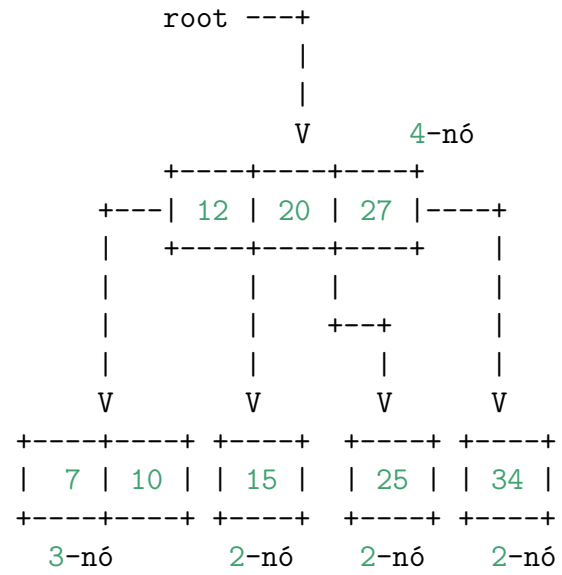

```
h = rotateRight(h);
```

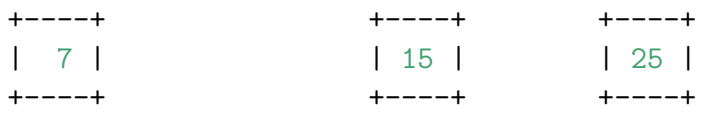




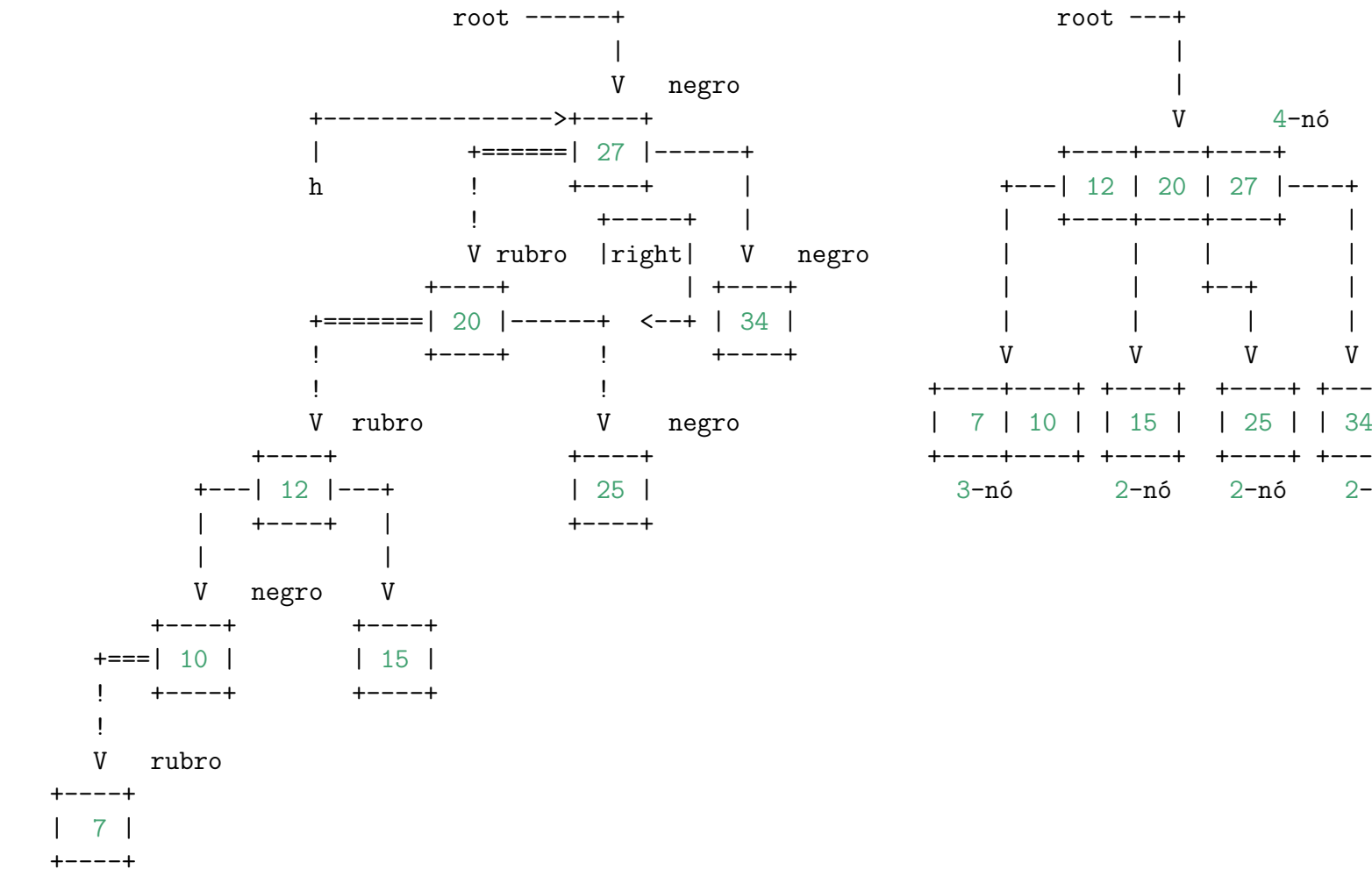

```
flipColors(h);
```



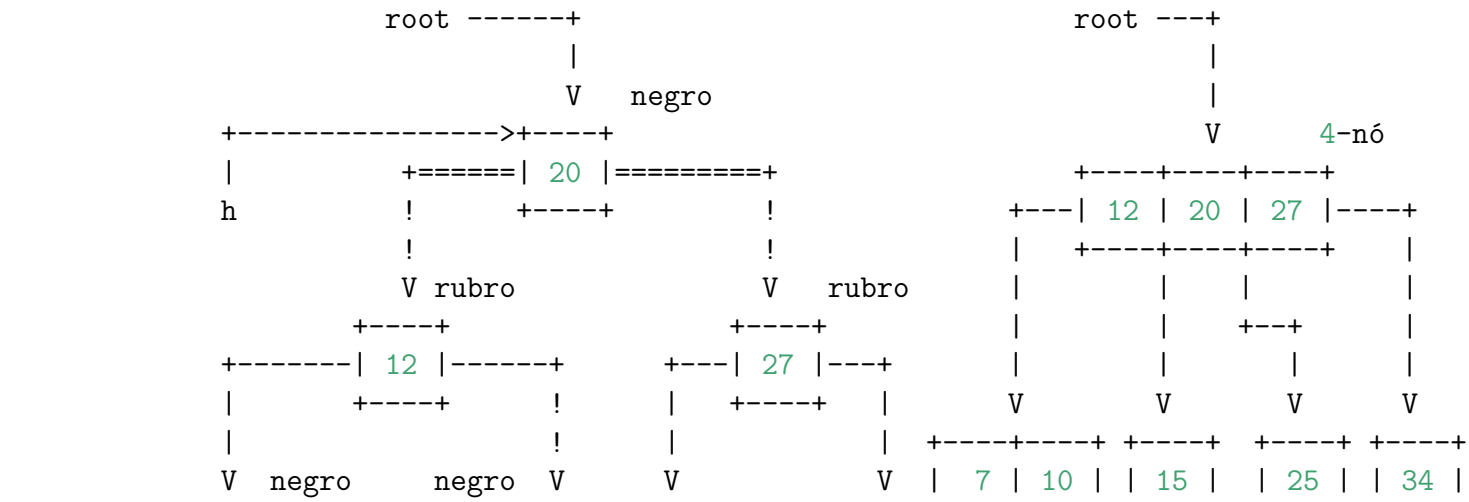


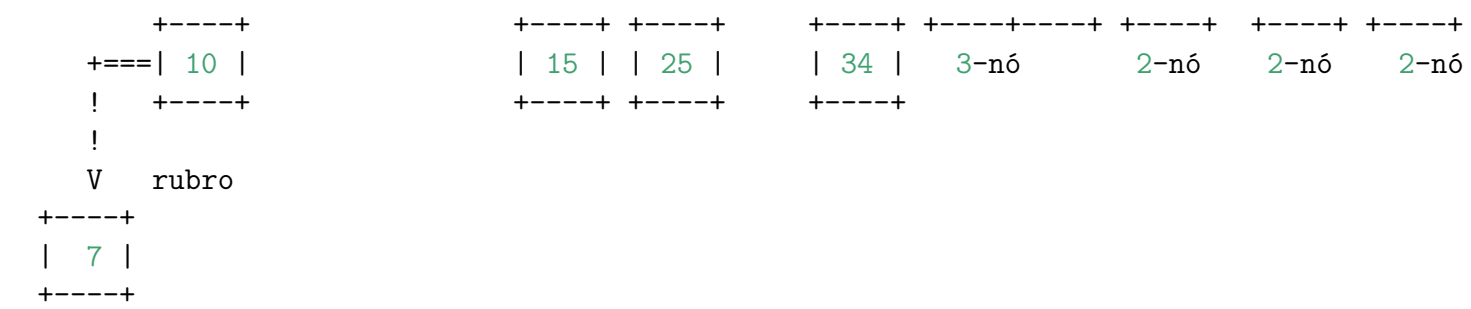



```
h = rotateLeft(h);
```

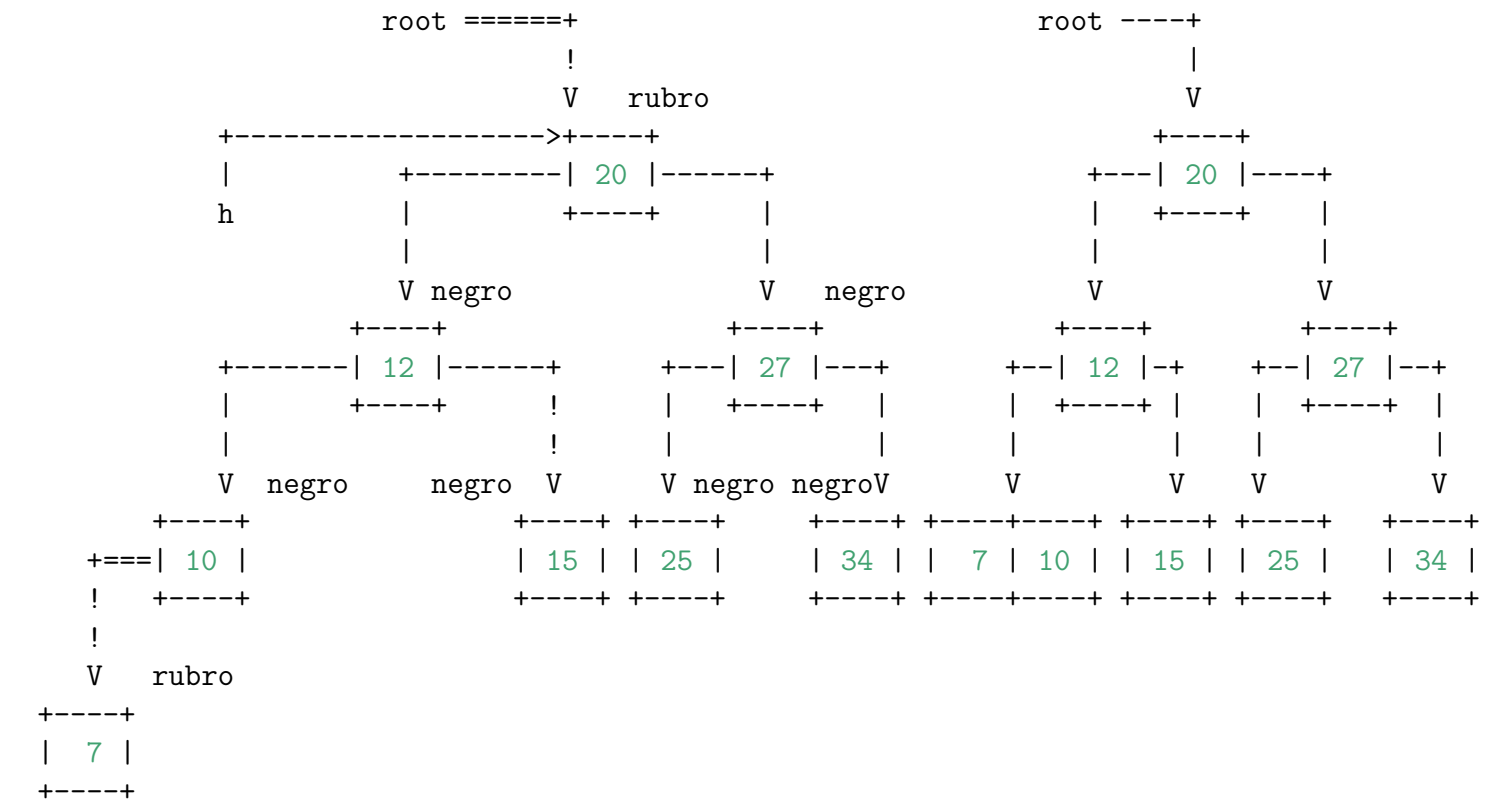



```
h = rotateRight(h);
```






```
flipColors(h);
```




```
root.color = BLACK; // manter BLACK o link para a raiz.
```

Implementação

Algoritmo 3.4: classe `RedBlackBST`:

```
    public class RedBlackBST<Key extends Comparable<Key>, Value> {

       private Node root;

       private class Node                // veja acima

       private boolean isRed(Node h)     // veja acima

       private Node rotateLeft(Node h)   // veja acima

       private Node rotateRight(Node h) // veja acima

       private void flipColors(Node h)   // veja acima

       private int size()                // veja na página sobre BSTs

       public void put(Key key, Value val) {
          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, 1, RED);

          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;

          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;
       }
    }
```