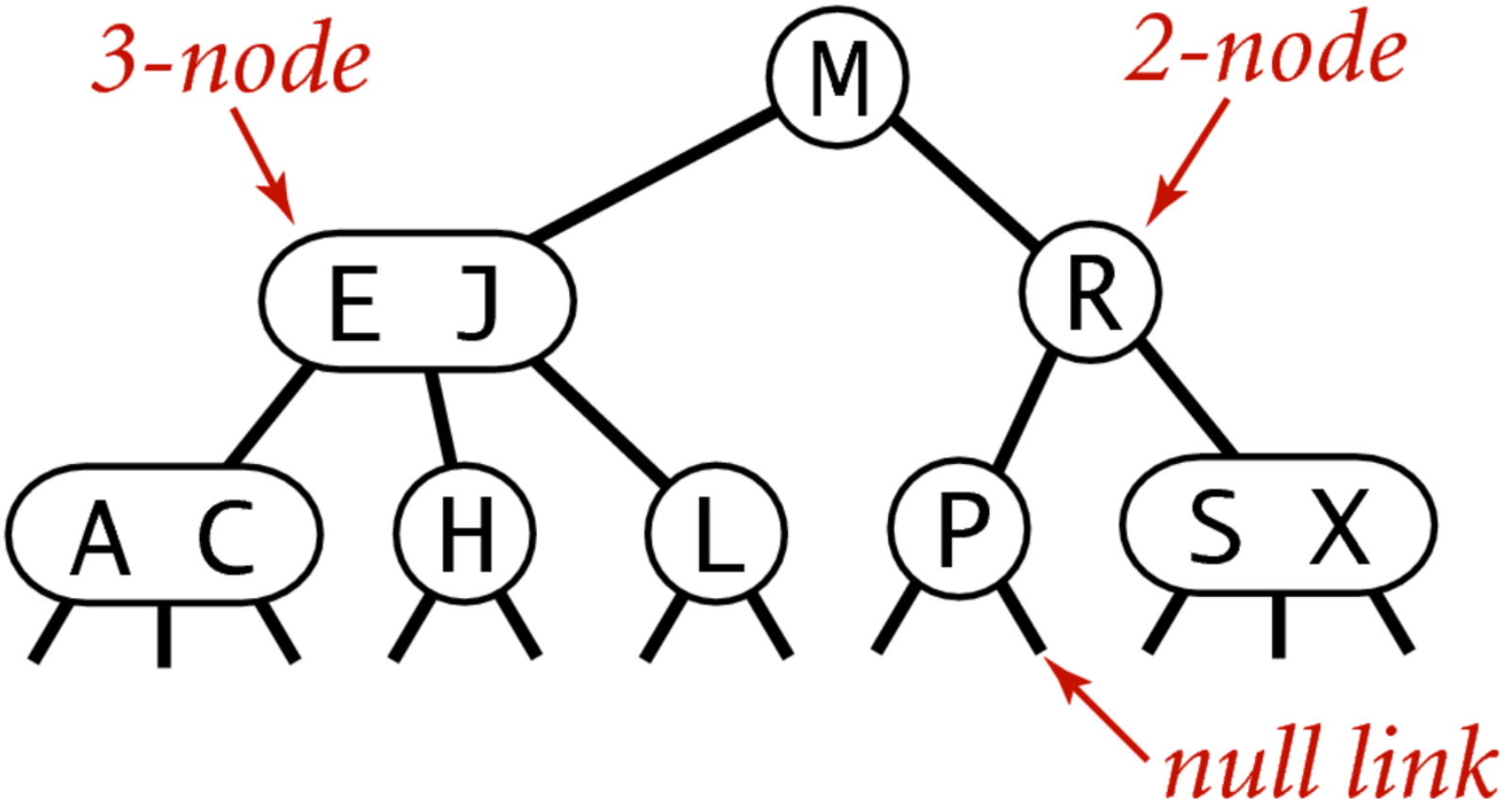
Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 11

Anatomia de uma árvore 2-3 de busca



Anatomy of a 2-3 search tree

Fonte: algs4

Árvores 2-3

Uma **árvore 2-3** é:

- uma árvore vazia;
- ou um nó simples com **2 links**:
  - um link `left` para uma árvore 2-3;
  - um link `right` para uma árvore 2-3;
- ou um nó duplo com **3 links**:
  - um link `left` para uma árvore 2-3;
  - um link `mid` para uma árvore 2-3; e
  - um link `right` para uma árvore 2-3.

Árvores 2-3 são **perfeitamente balanceada**: todos links `null` estão no mesmo nível.

Estrutura

Fato. Toda árvore 2-3 de altura $\mathtt{h}$ tem no mínimo $2^{\mathtt{h}+1}-1$ nós e no máximo $3^{\mathtt{h}+1}-1$ nós.

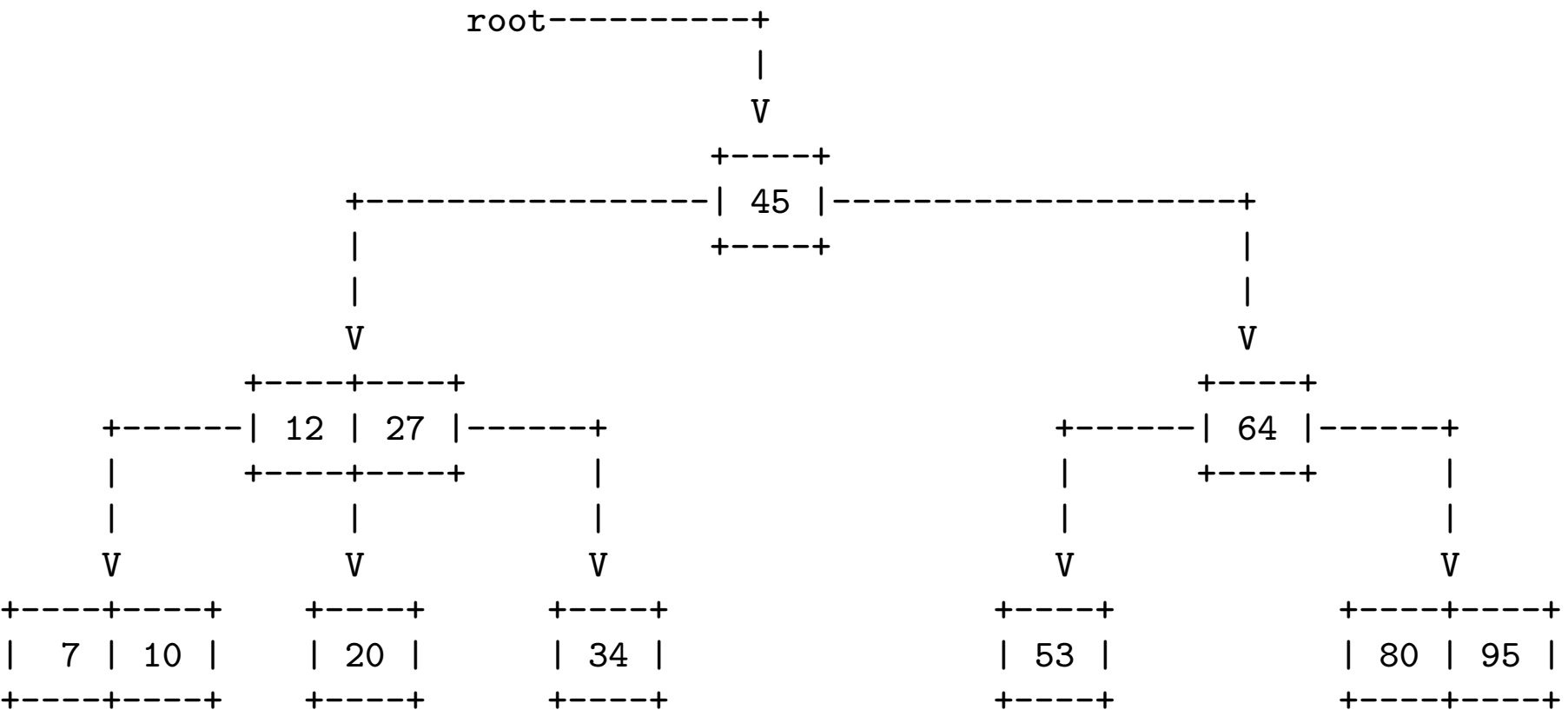
Consequência. Toda árvore 2-3 com $\mathtt{n}$ nós tem altura não superior a $\lg(\mathtt{n}+1)-1$ e não inferior a $\log_3(\mathtt{n}+1)-1$.

Árvore 2-3 de busca

Uma árvore 2-3 de **busca** (*2-3 search tree*) é:

- uma árvore vazia;
- ou um nó simples com uma chave e **2 links**:
  - um link `left` para uma árvore 2-3 que tem chaves menores que a chave do nó e
  - um link `right` para uma árvore 2-3 que tem chaves maiores;
- ou um nó duplo com duas chave e **3 links**:
  - um link `left` para uma árvore 2-3 que tem chaves menores;
  - um link `mid` para uma árvore 2-3 que tem chaves entre as duas chaves do nó; e
  - um link `right` para uma árvore 2-3 que tem chaves maiores.

Exemplo de árvore 2-3 de busca



Consumo de tempo

Numa árvore 2-3 com n nós, busca e inserção nunca visitam mais que $\lg(n+1)$. Cada visita faz no máximo 2 comparações de chaves.

BSTs rubro-negras

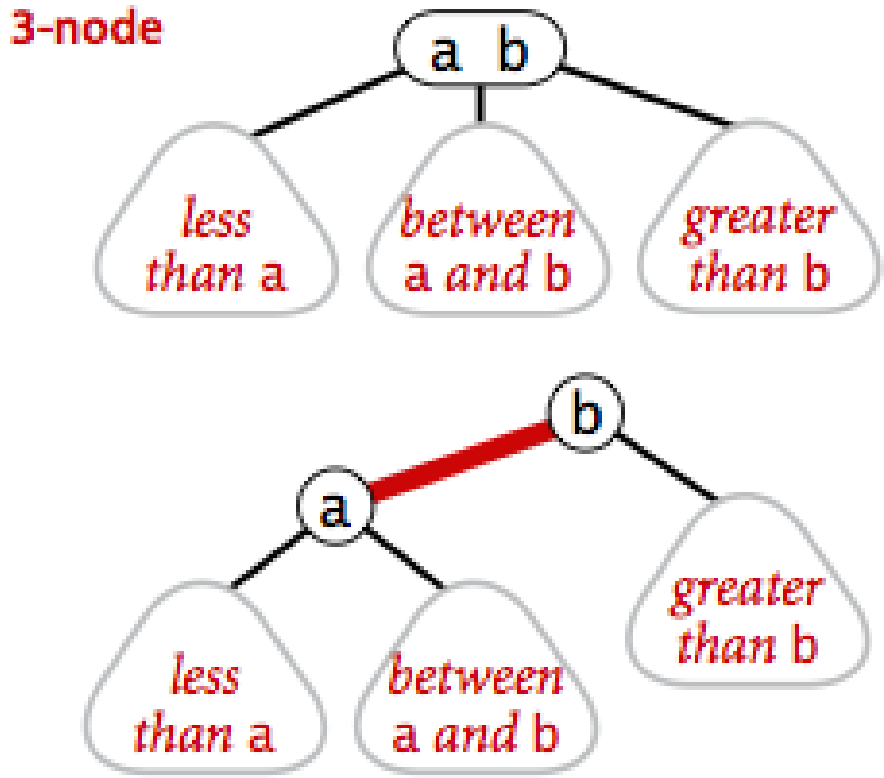
Fonte: http://scottlobdell.me/

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

BSTs rubro-negras

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

Links rubros são **sempre** inclinados para a esquerda.



BSTs rubro-negras

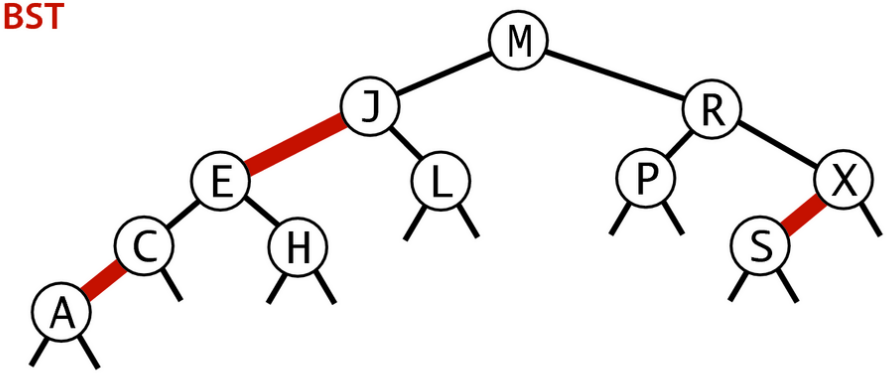
Uma **BST rubro-negra** é uma `BST` cujos links são negros e rubros e:

- 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.

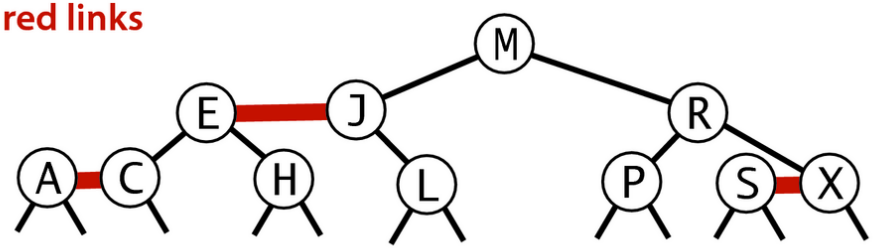
Se os links rubros forem desenhados horizontalmente e depois contraídos, teremos uma árvore 2-3

Anatomia de uma árvore rubro-negra

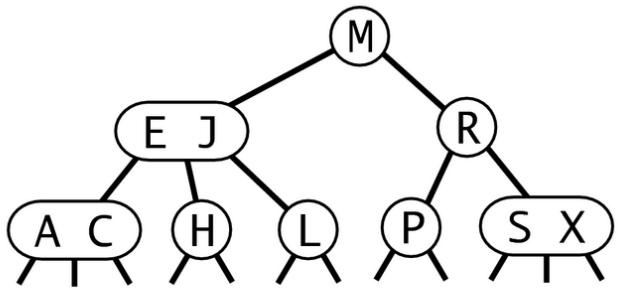
red-black BST



horizontal red links



2-3 tree

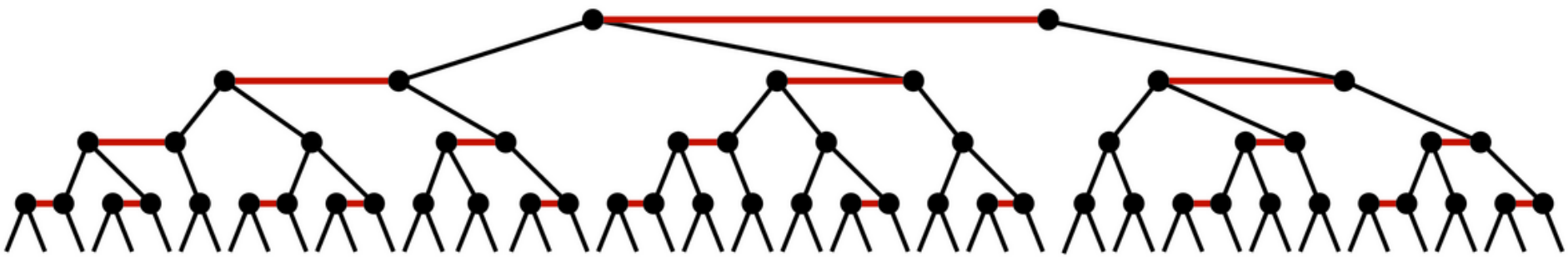


Fonte: algs4

1-1 correspondence between red-black BSTs and 2-3 trees

Árvore 2-3 para rubro-negra

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



A red-black tree with horizontal red links is a 2-3 tree

Fonte: algs4

Nós de uma BST rubro-negra

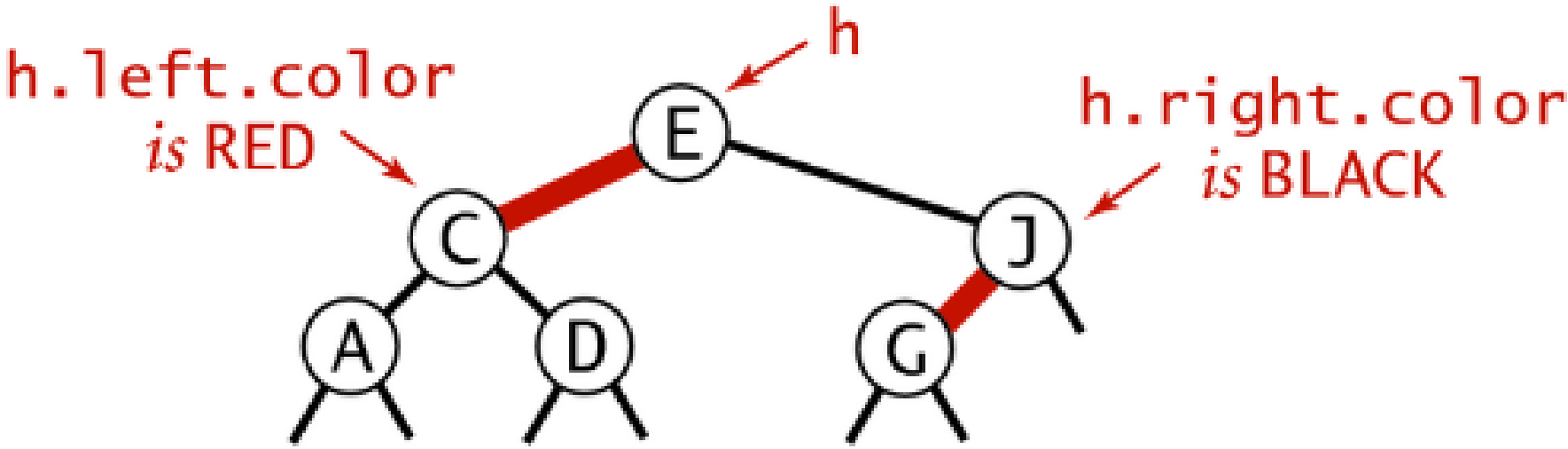
É inconveniente armazenar a cor de um link na estrutura de dados; é mais simples armazenar essa informação nos nós.

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;
```

Nós de uma BST rubro-negra



Nós de uma BST rubro-negra

```
private class Node{
   Key key; Value val;
   Node left, right;
   int n; // número de nós nesta subárvore
   boolean color; // cor do link 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;
}
```

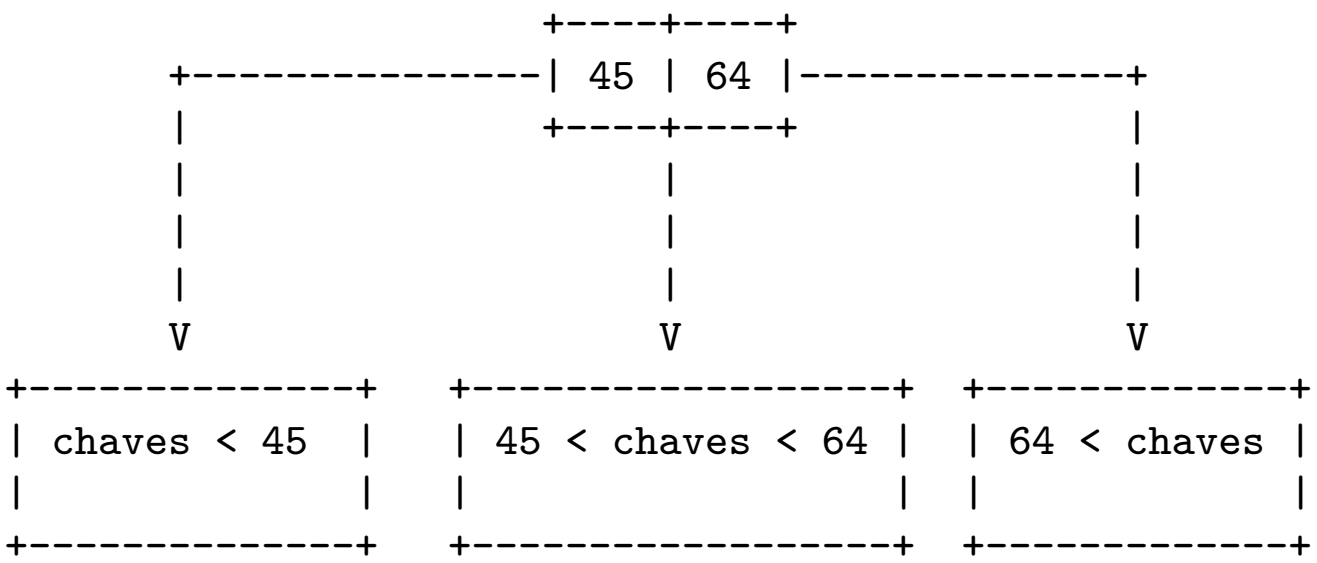
AULA 12

BSTs rubro-negras: `put()`

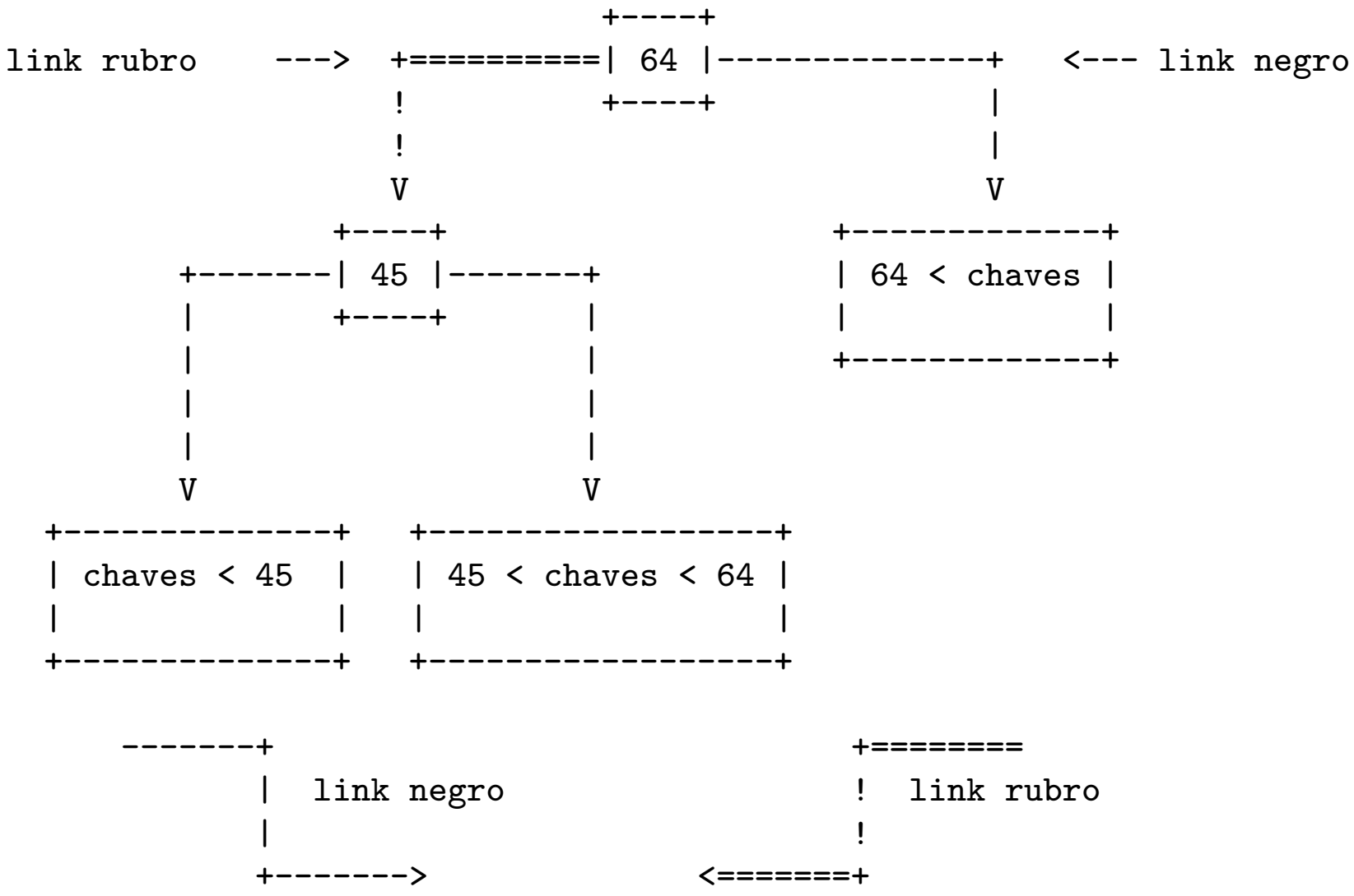
Fonte: `...-crimson-vermillion-Chinese/`

Referências: BSTs rubro-negras (PF); Balanced Search Trees (S&W); slides (S&W)

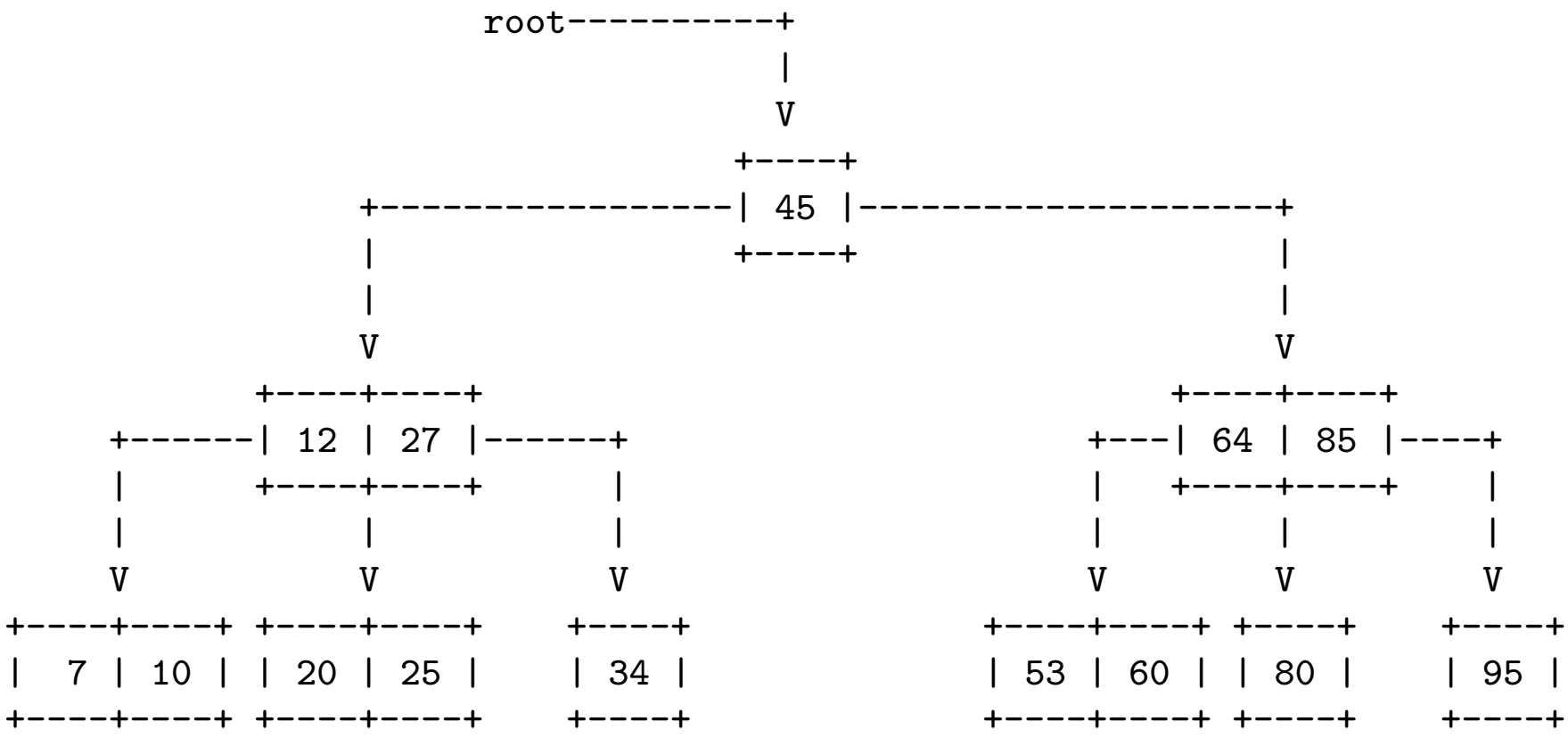
Árvore 2-3 para rubro-negra



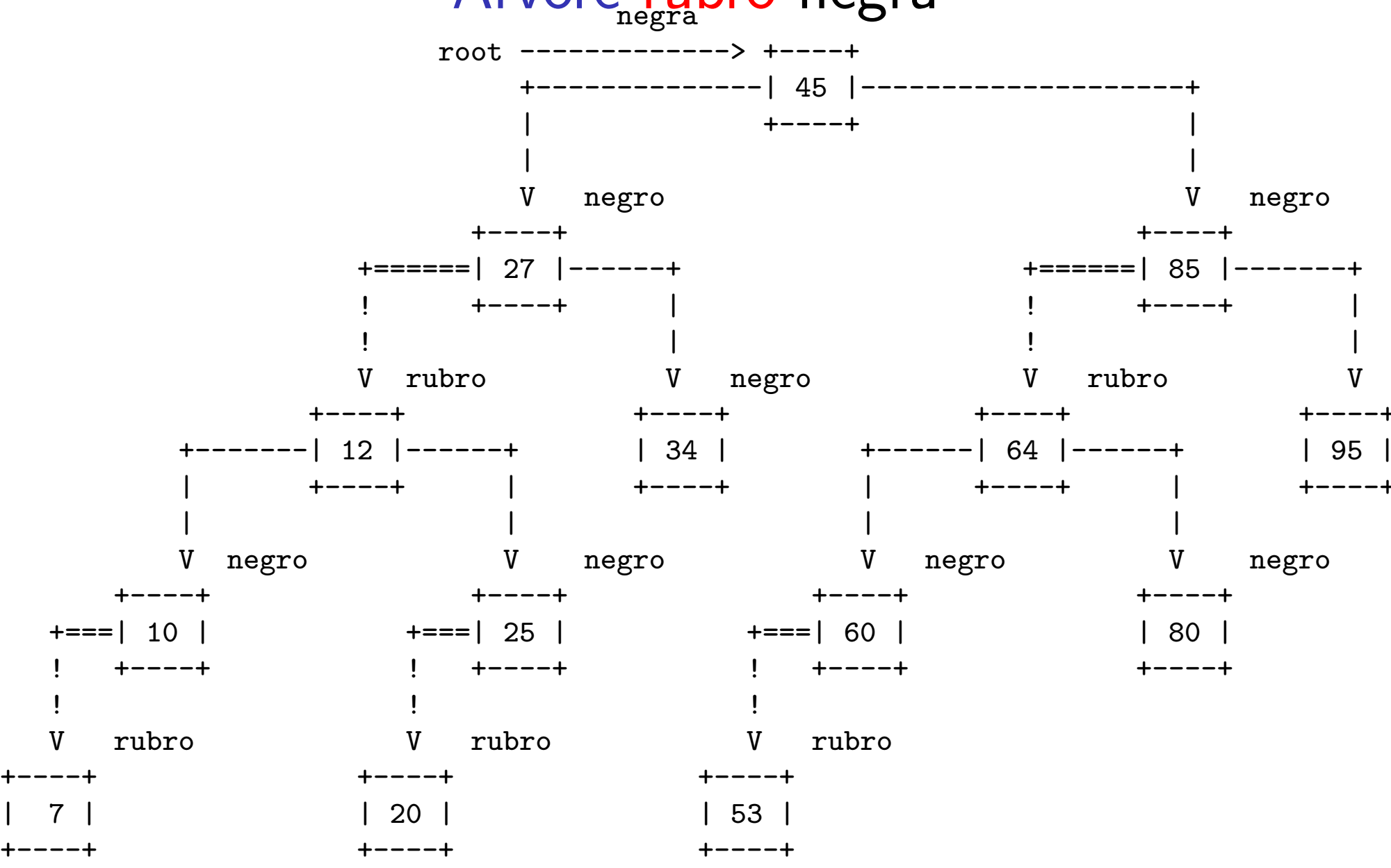
Árvore 2-3 para rubro-negra



Árvore 2-3

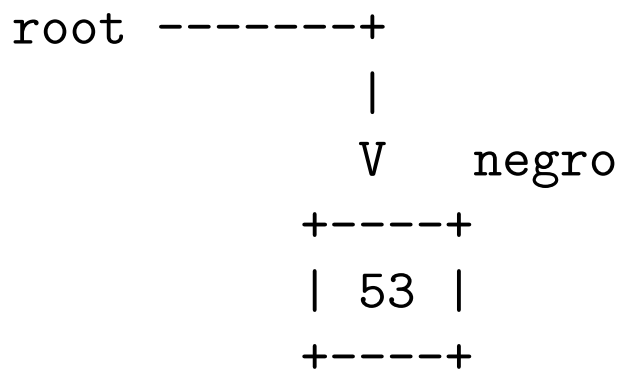


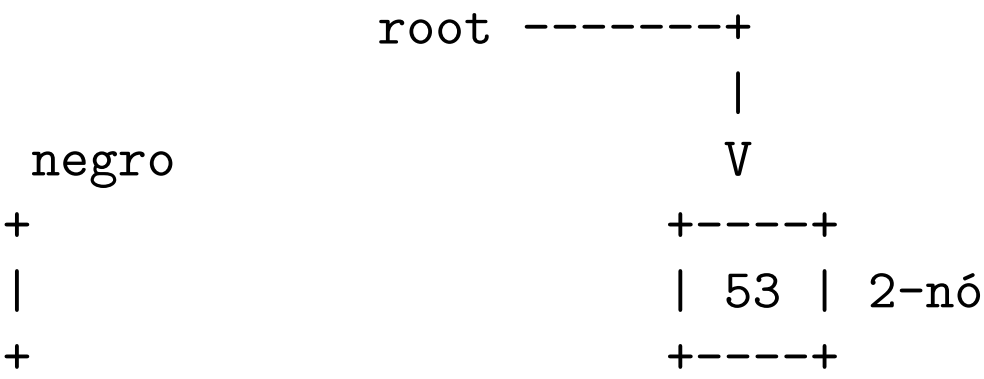
Árvore rubro-negra



Inserção em um 2-nó

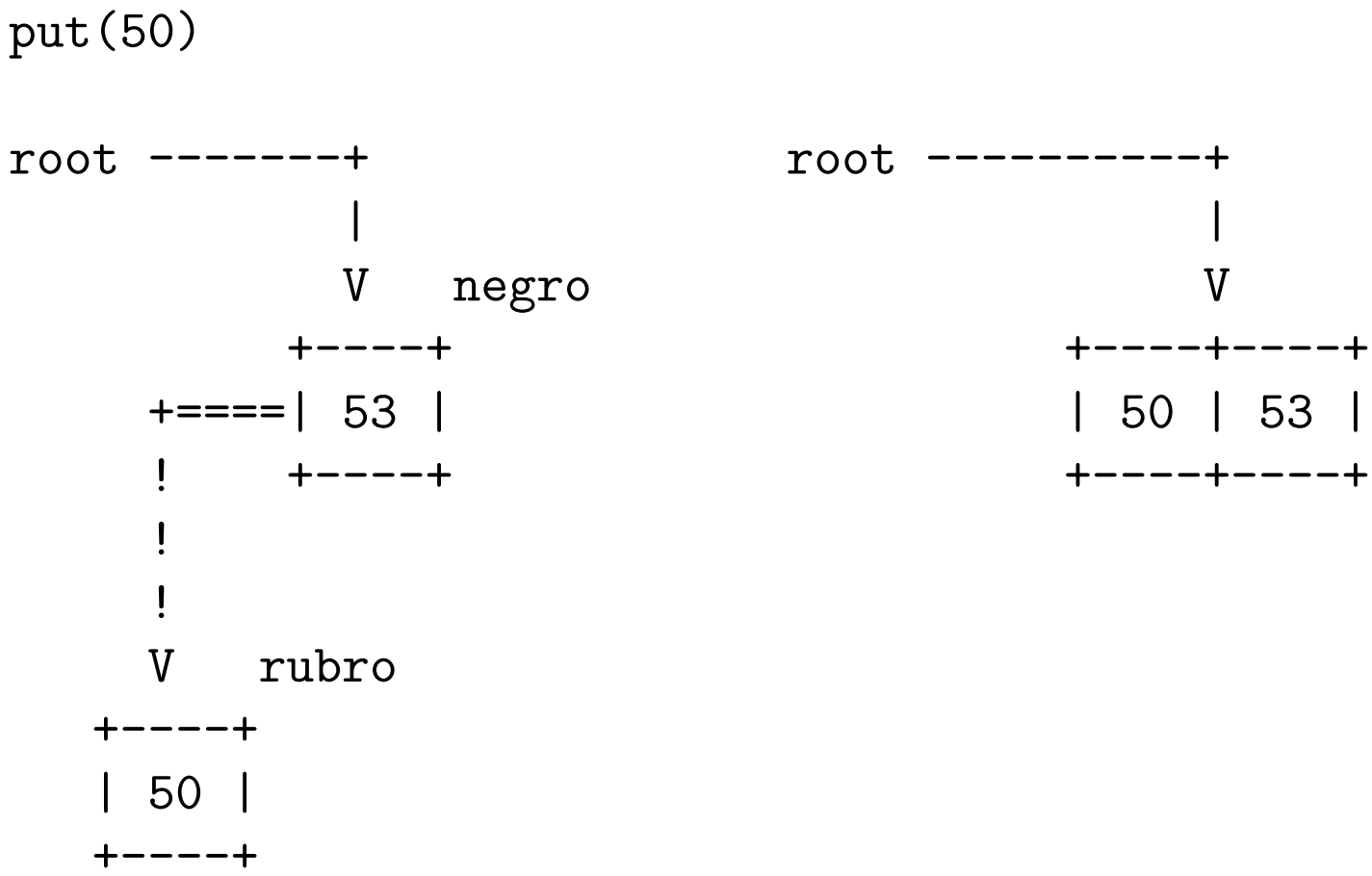
Árvore formada por apenas um 2-nó





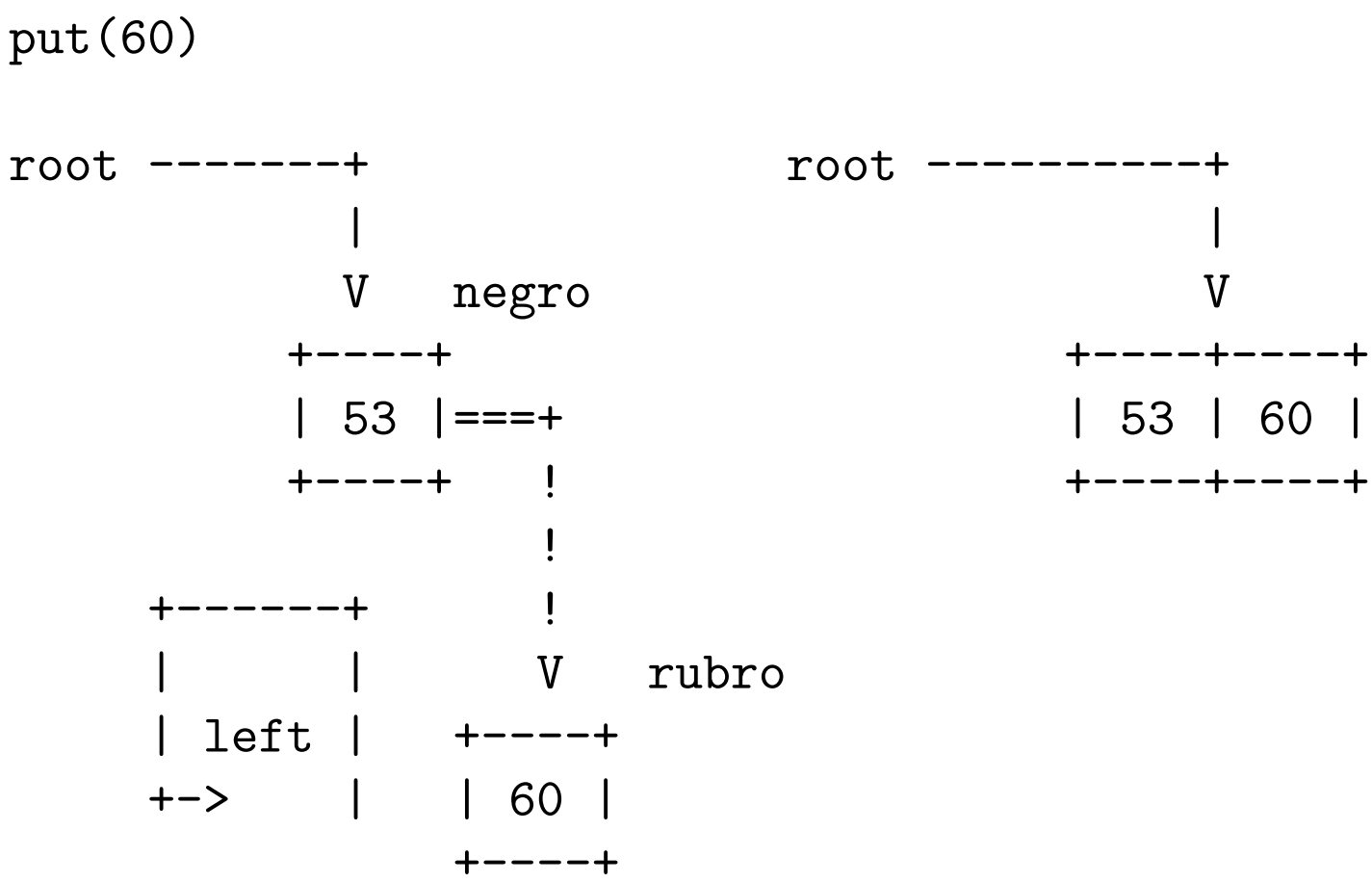
Inserção em um 2-nó

Árvore formada por apenas um 2-nó



Inserção em um 2-nó

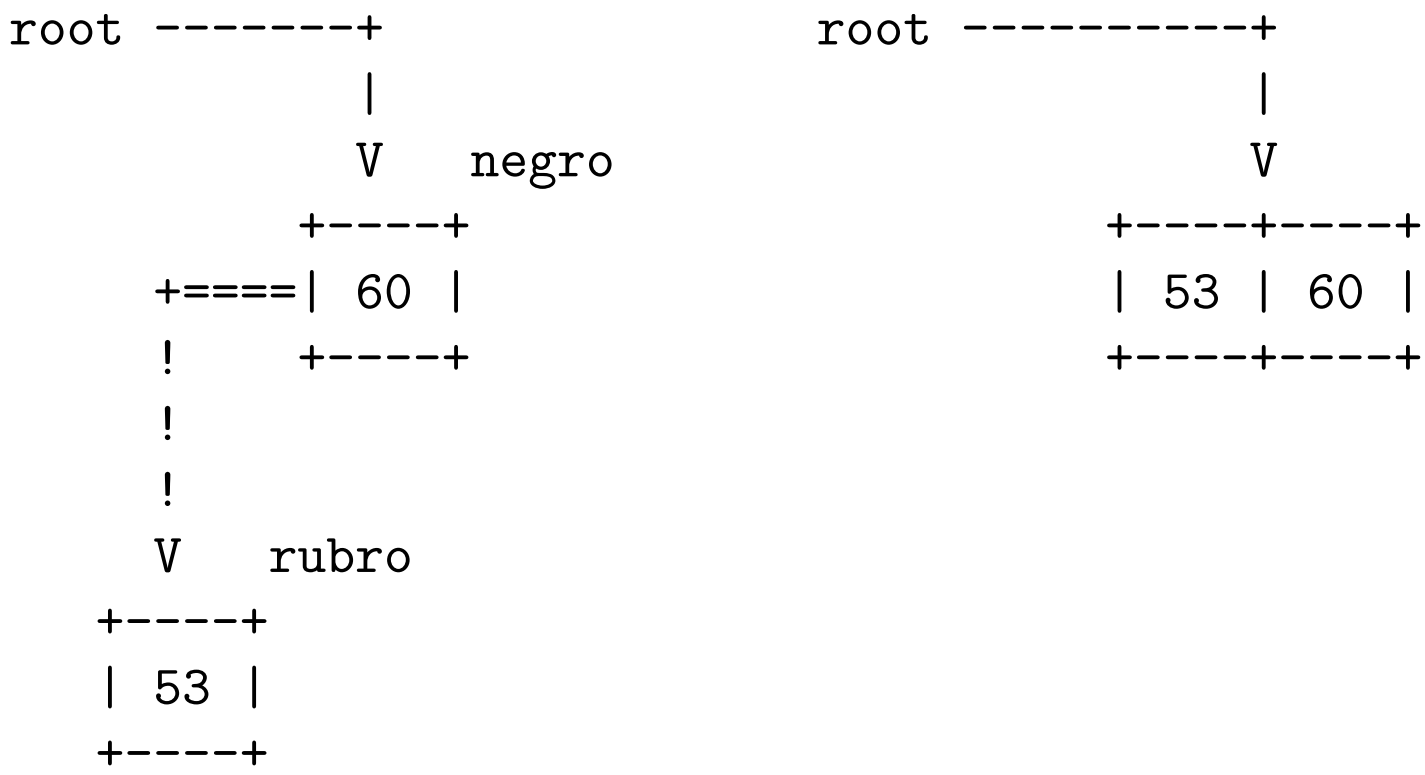
Árvore formada por apenas um 2-nó



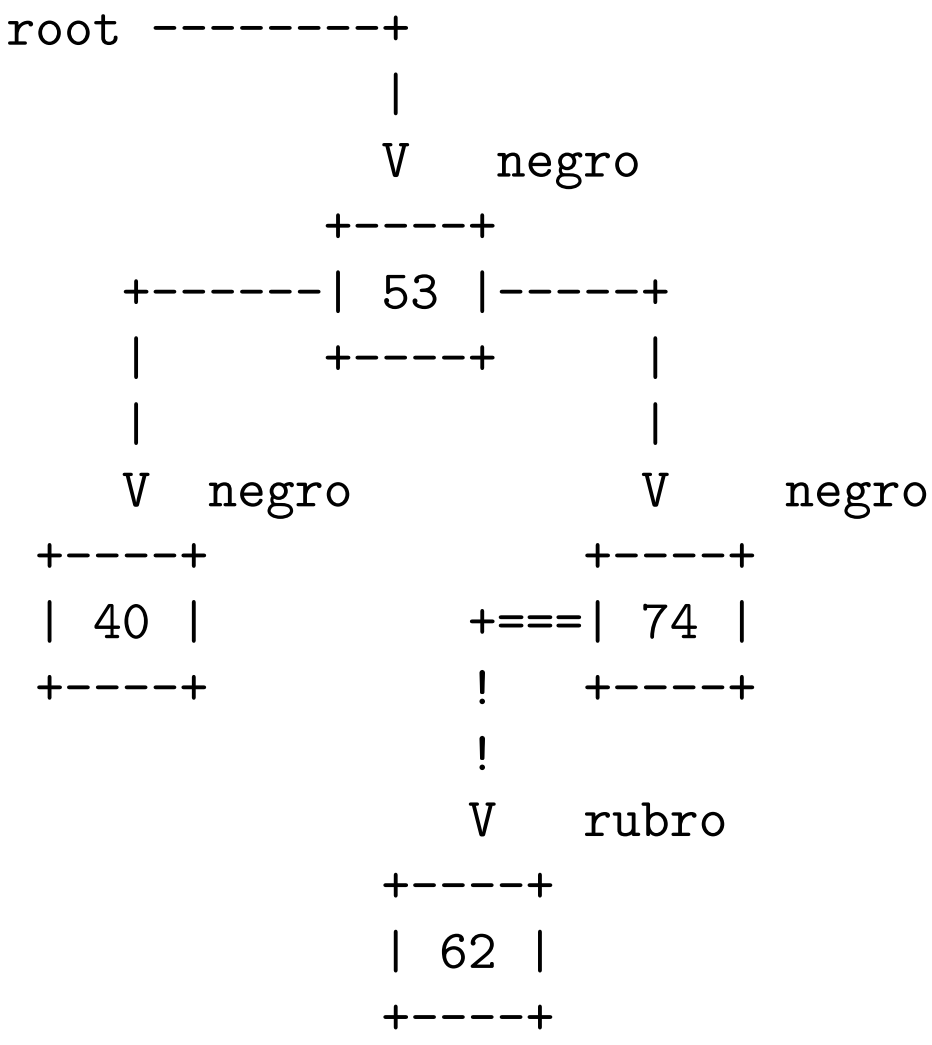
Inserção em um 2-nó

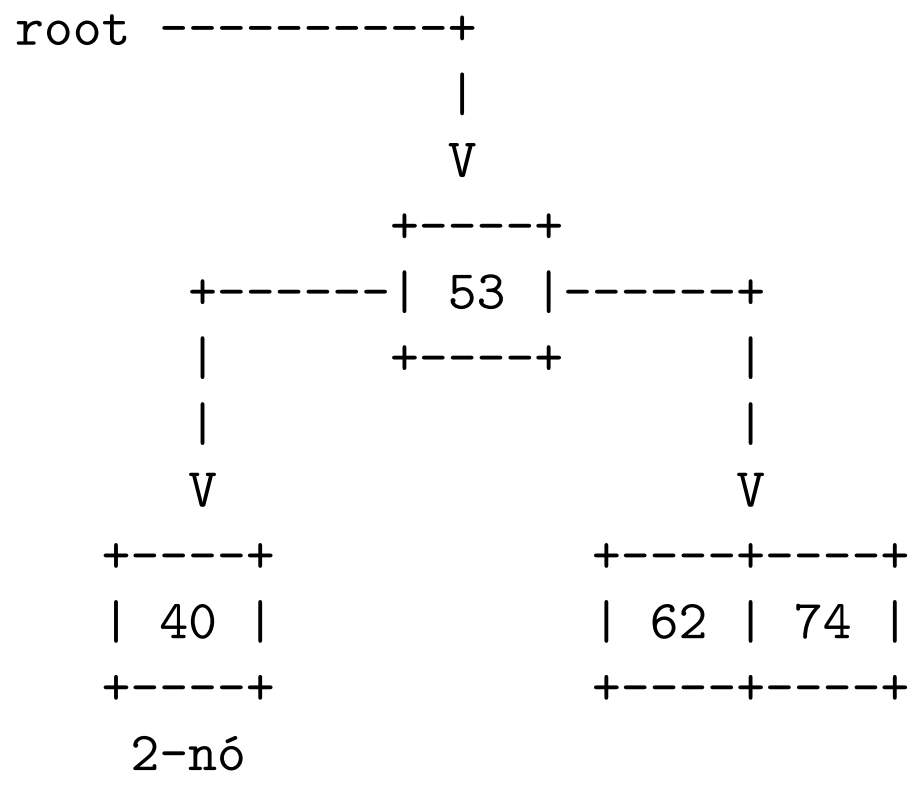
Árvore formada por apenas um 2-nó

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

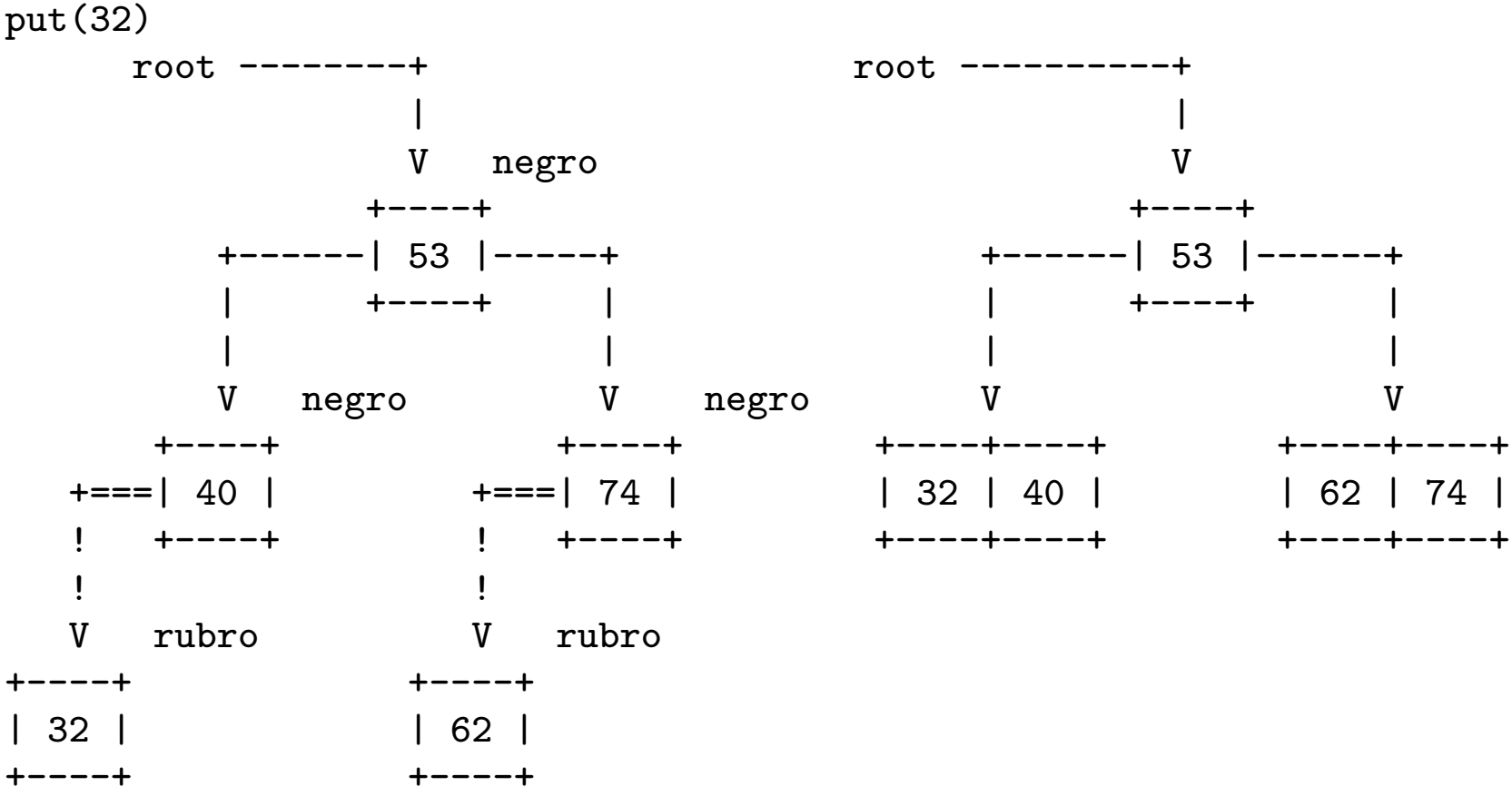


Inserção em um 2-nó qualquer

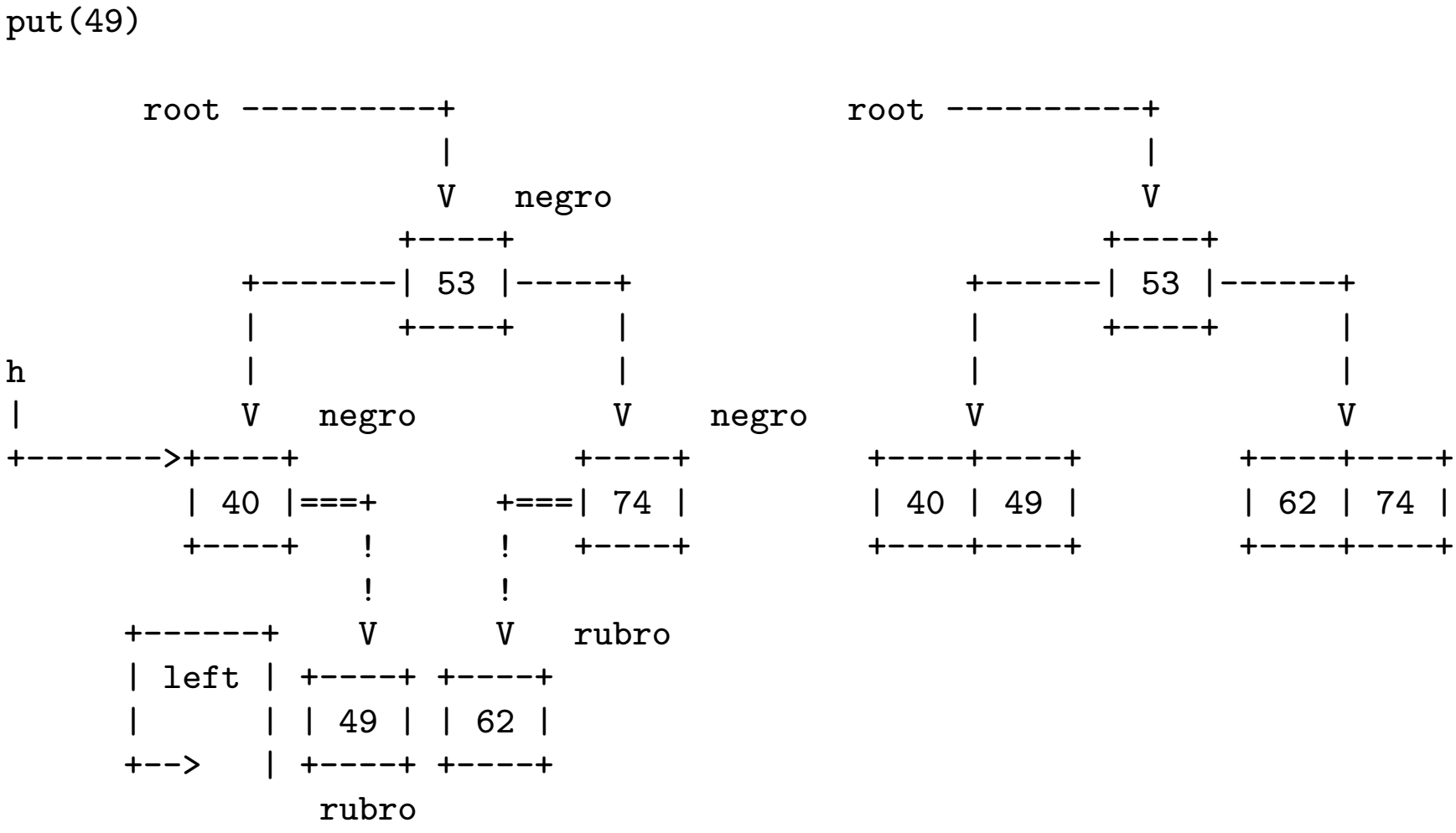




Inserção em um 2-nó qualquer

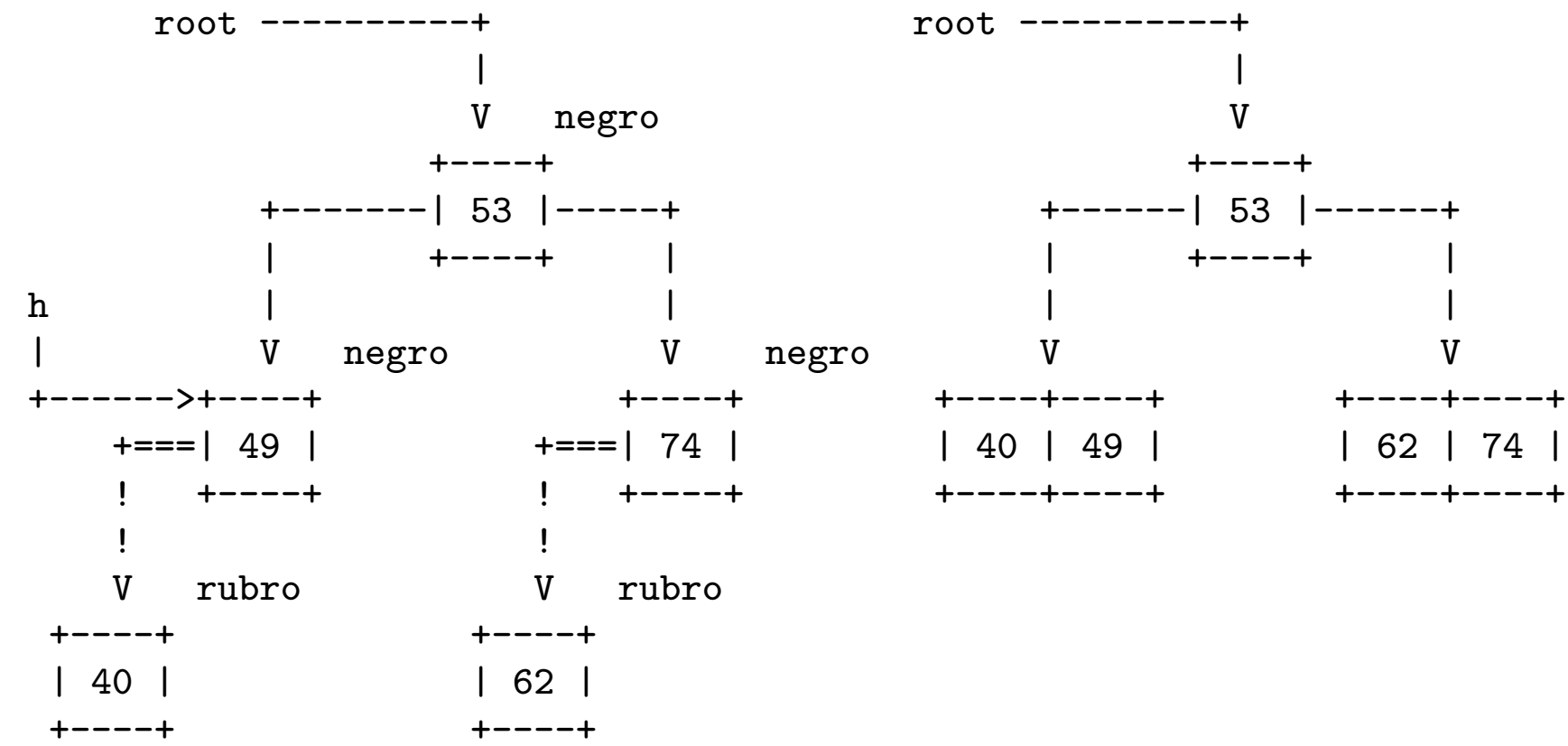


Inserção em um 2-nó qualquer

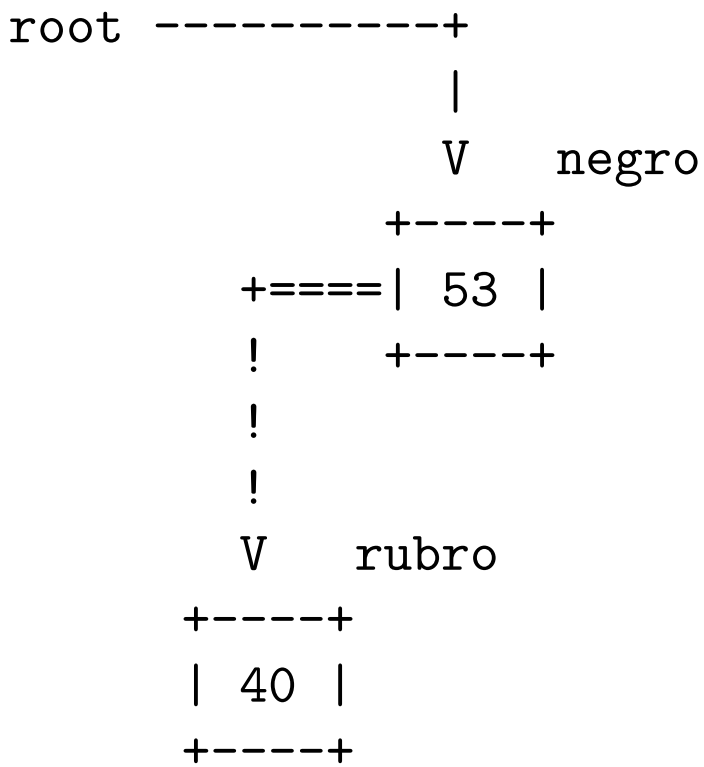


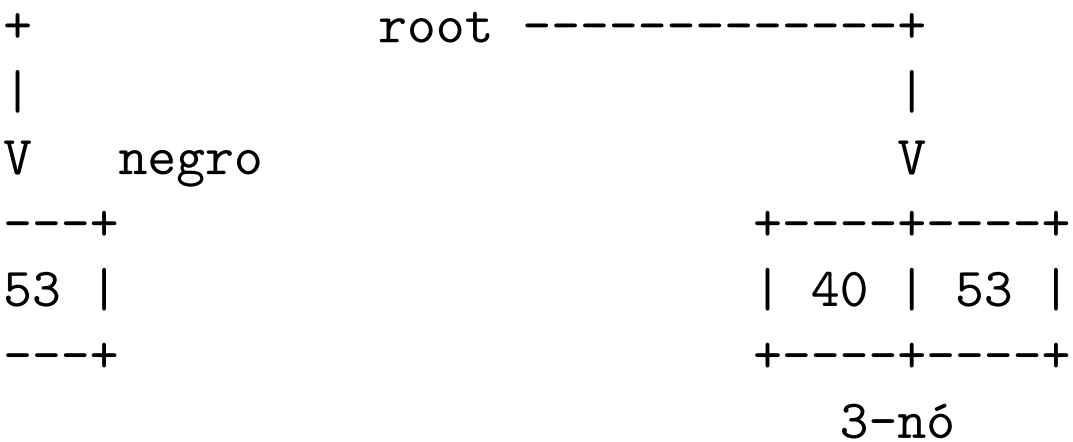
Inserção em um 2-nó qualquer

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

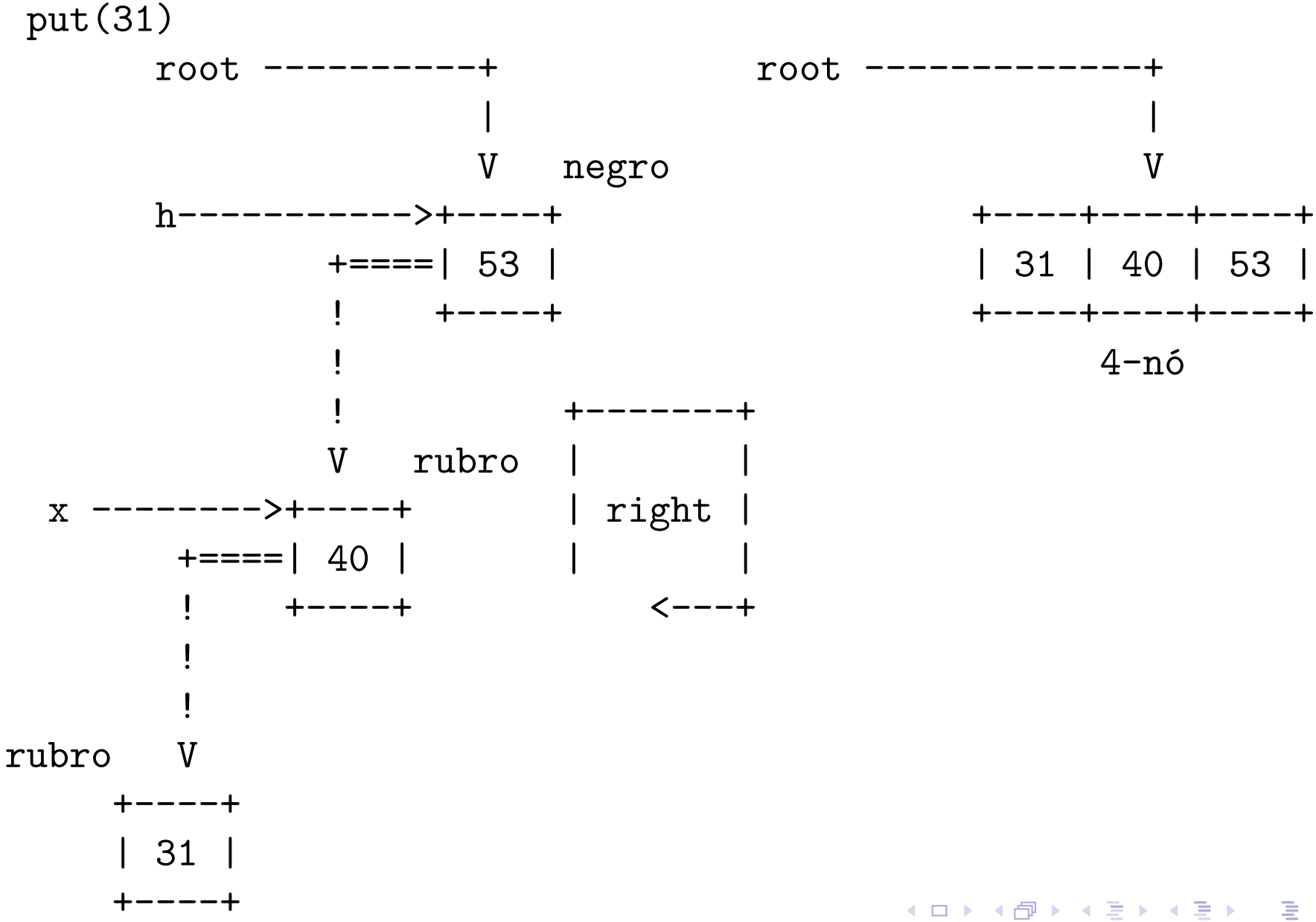


Inserção em um 3-nó



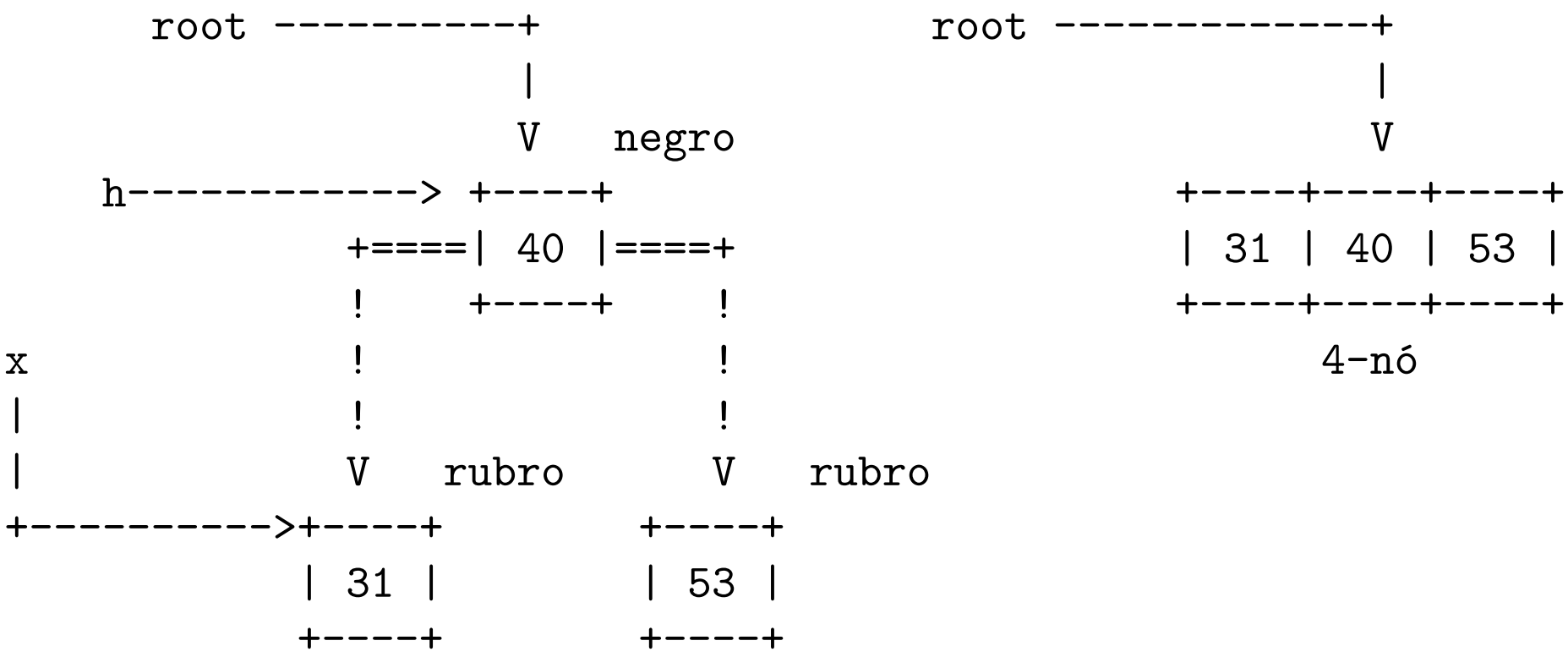


chave é inserida é menor do 3-nó



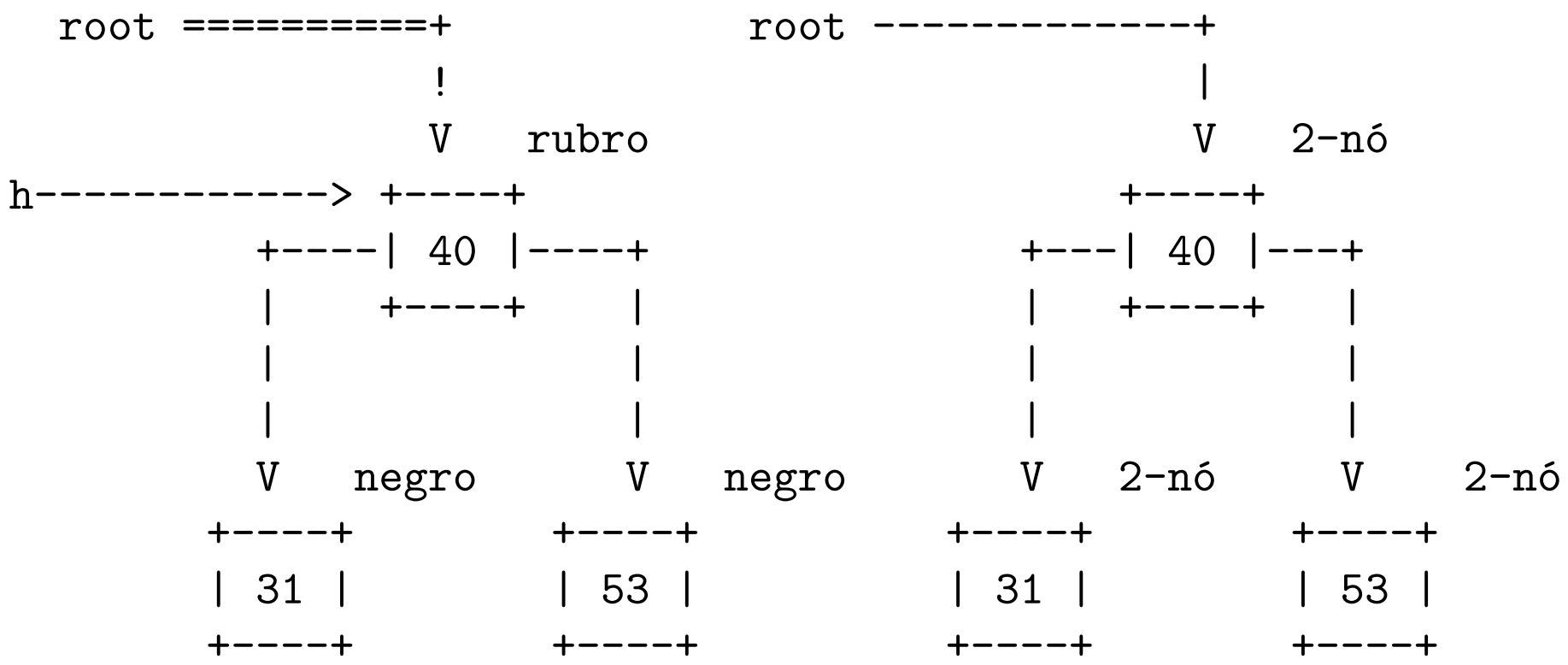
chave é inserida é menor do 3-nó

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

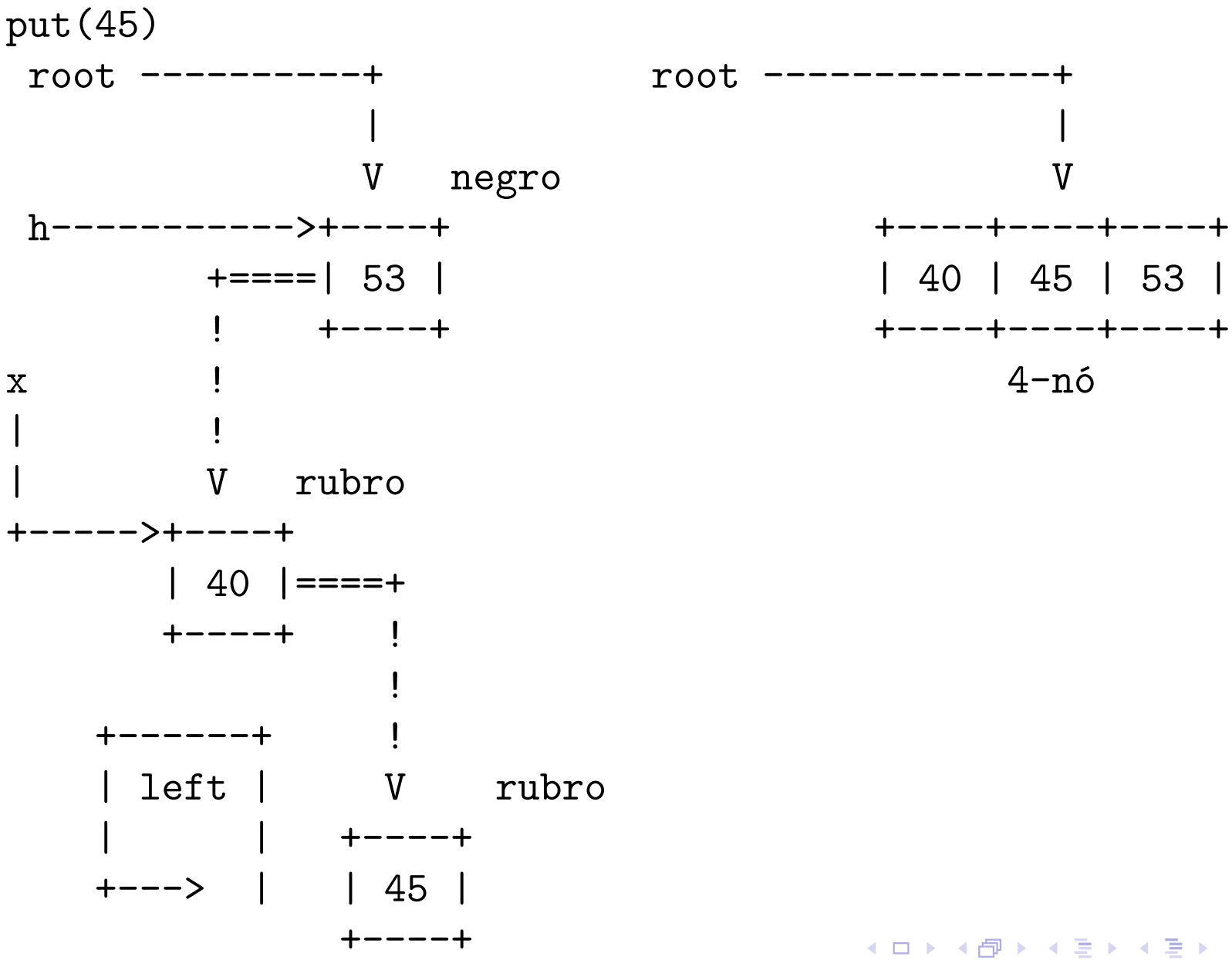


chave é inserida é menor do 3-nó

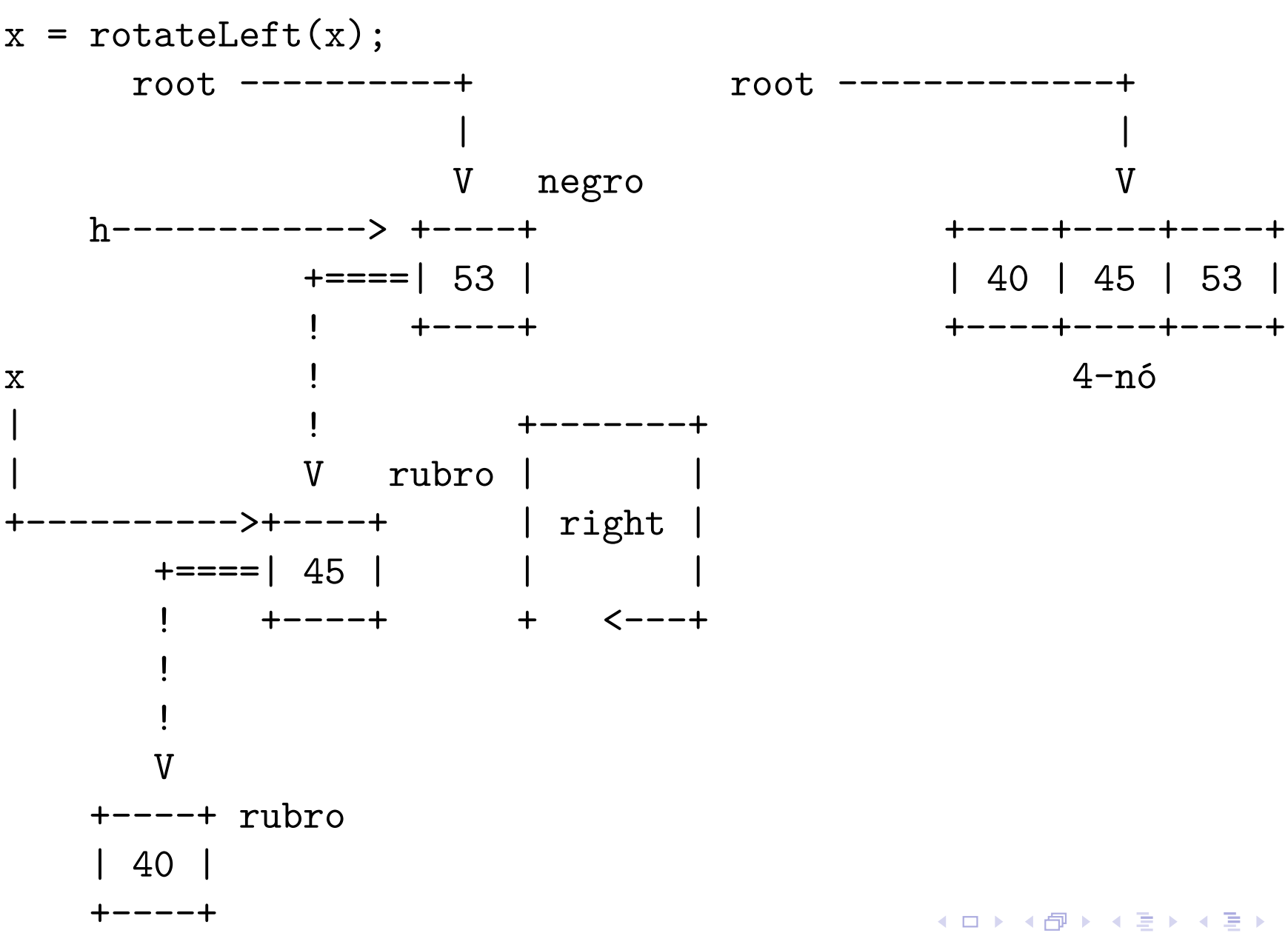
```
flipColors(h);
```



chave é inserida entre as chaves do 3-nó

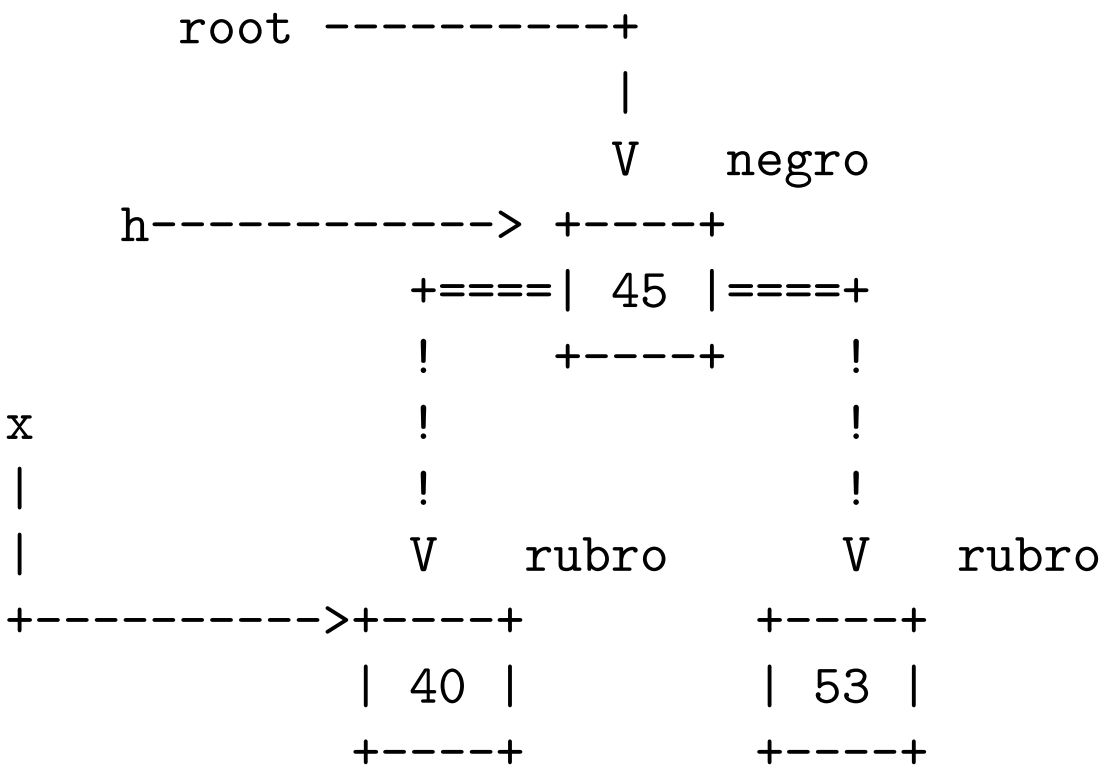


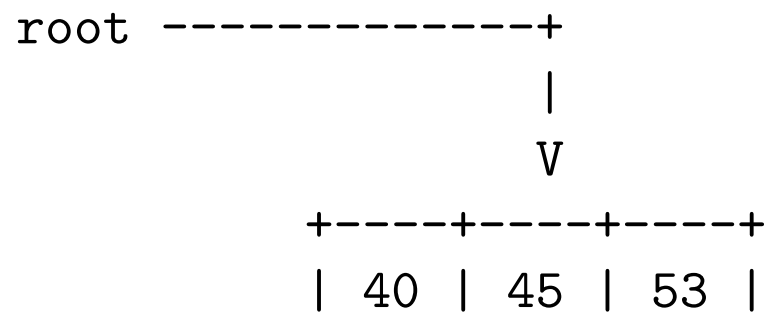
chave é inserida entre as chaves do 3-nó



chave é inserida entre as chaves do 3-nó

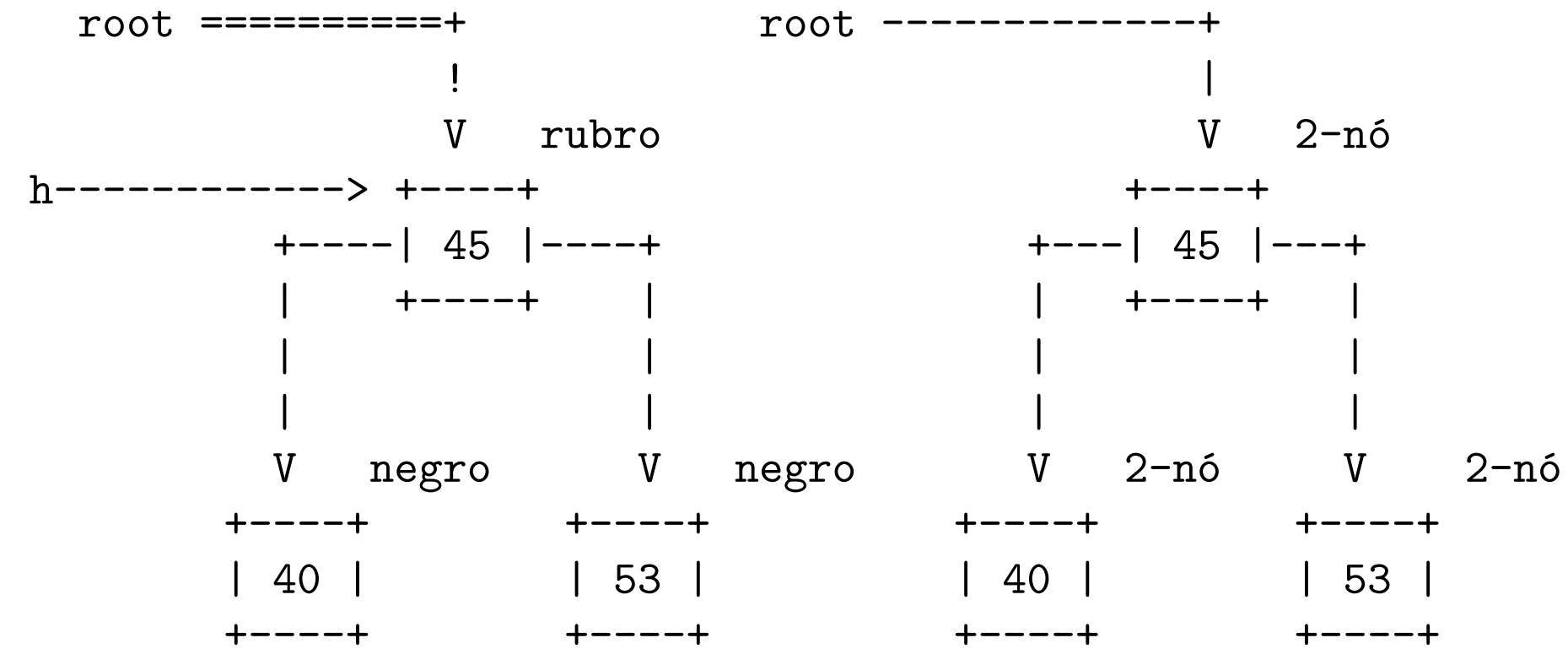
```
h = rotateRight(h);
```



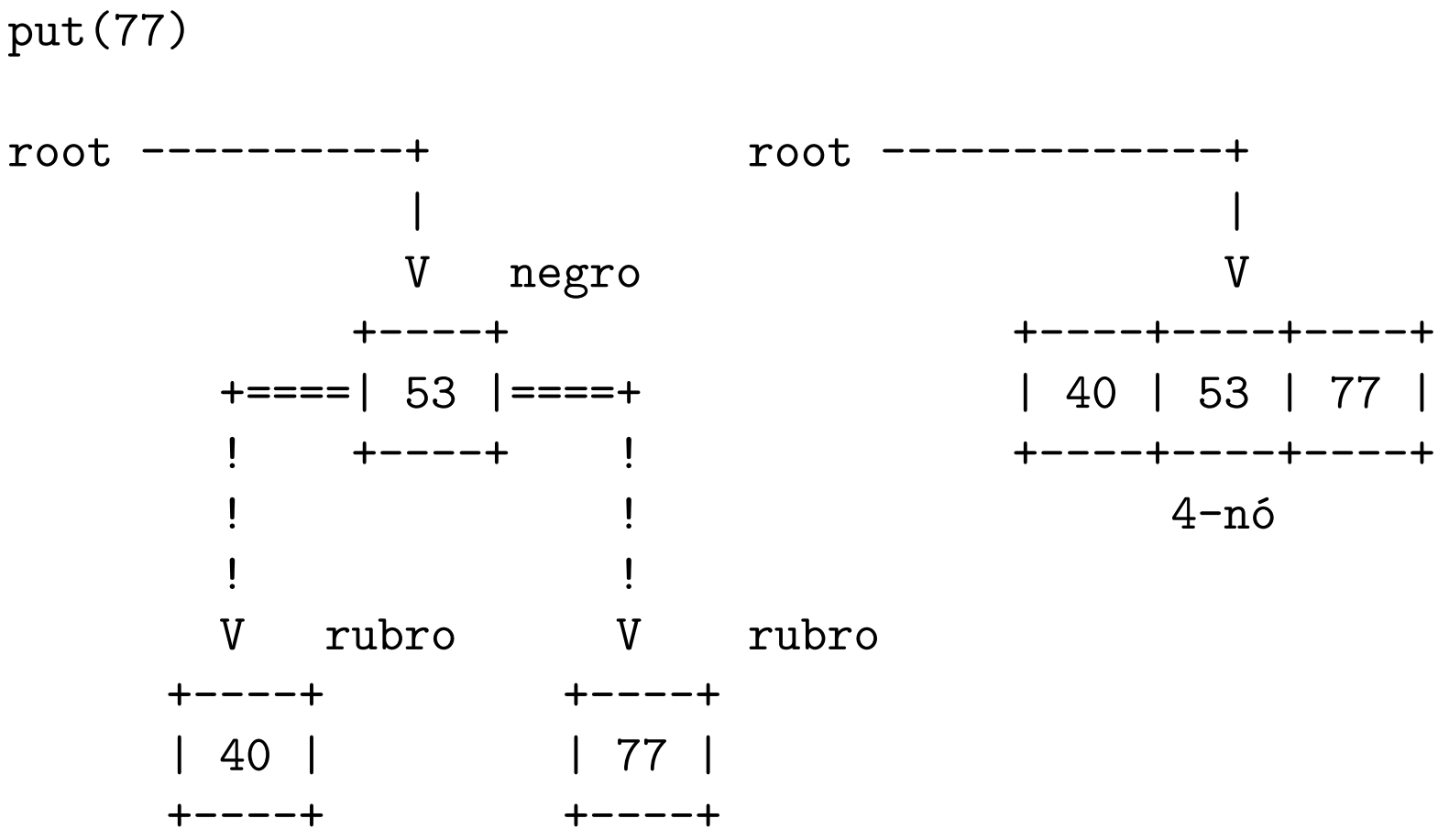


chave é inserida entre as chaves do 3-nó

flipColors(h); hmmm. raiz deve ser negra

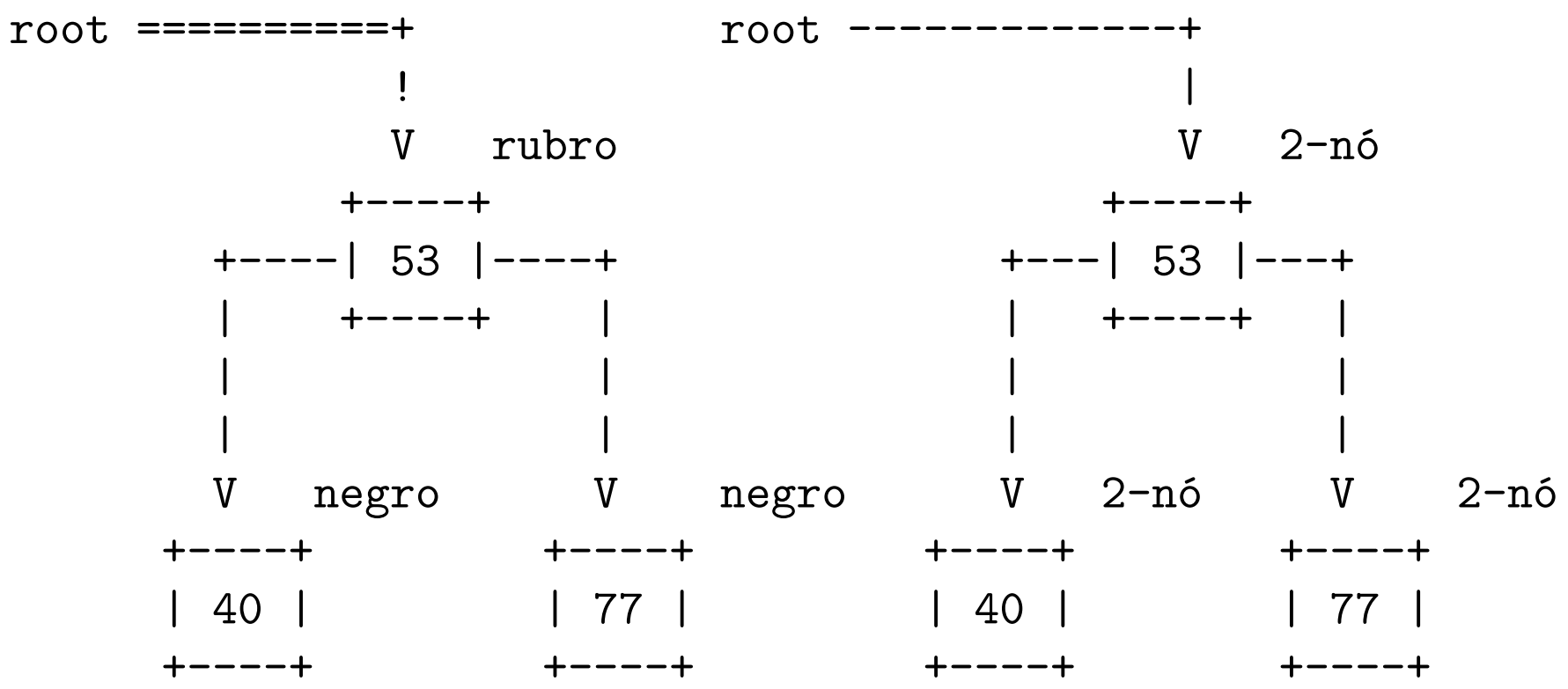


chave inserida é maior que todas do 3-nó

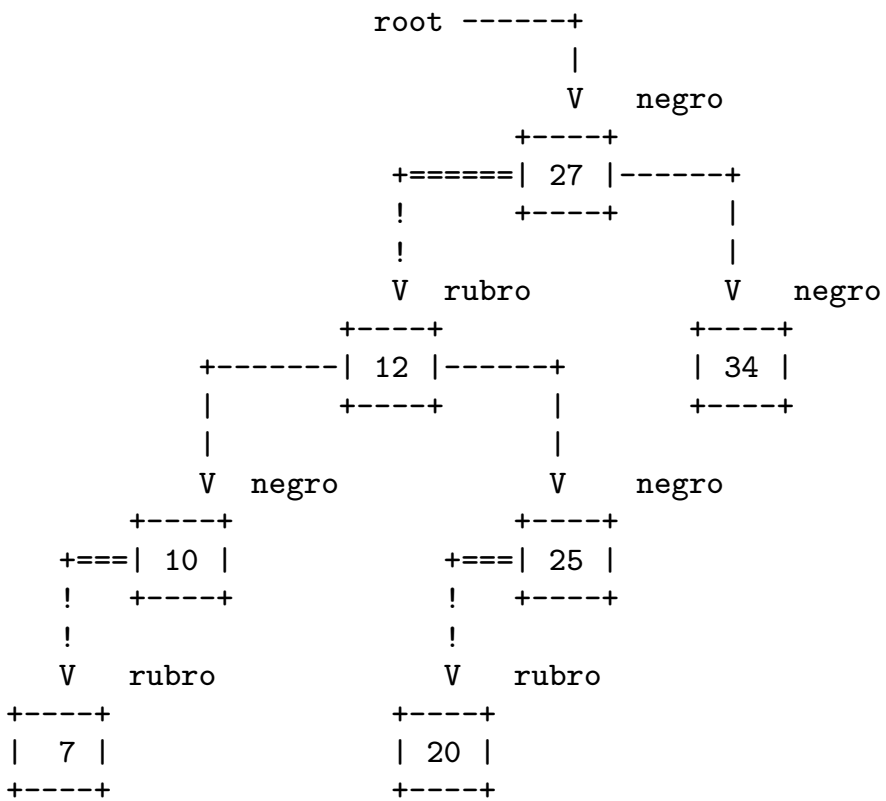


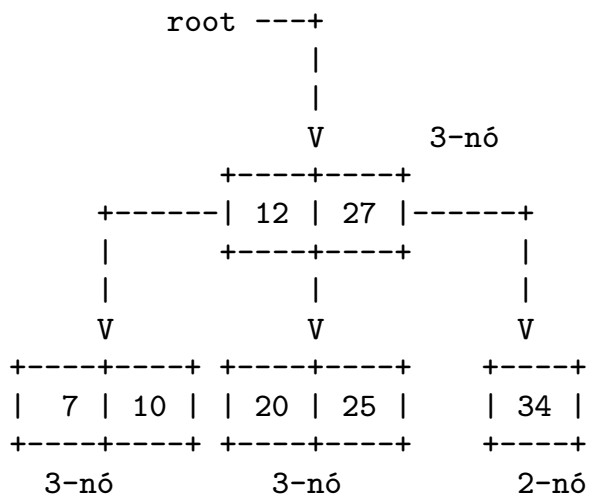
chave inserida é maior que todas do 3-nó

```
flipColors(root);  hmmm. raiz deve ser negra
```

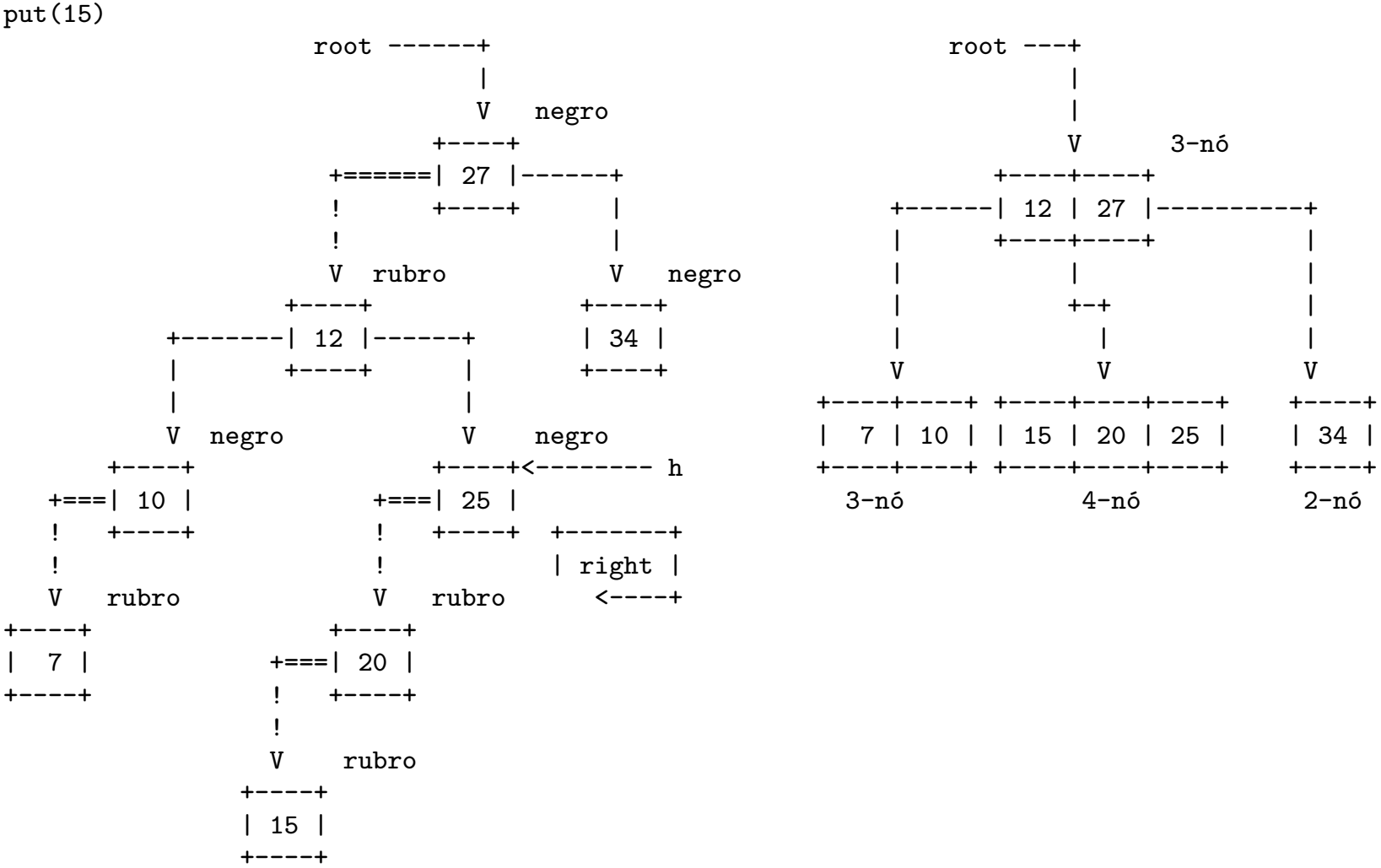


chave é inserida em um 3-nó qualquer



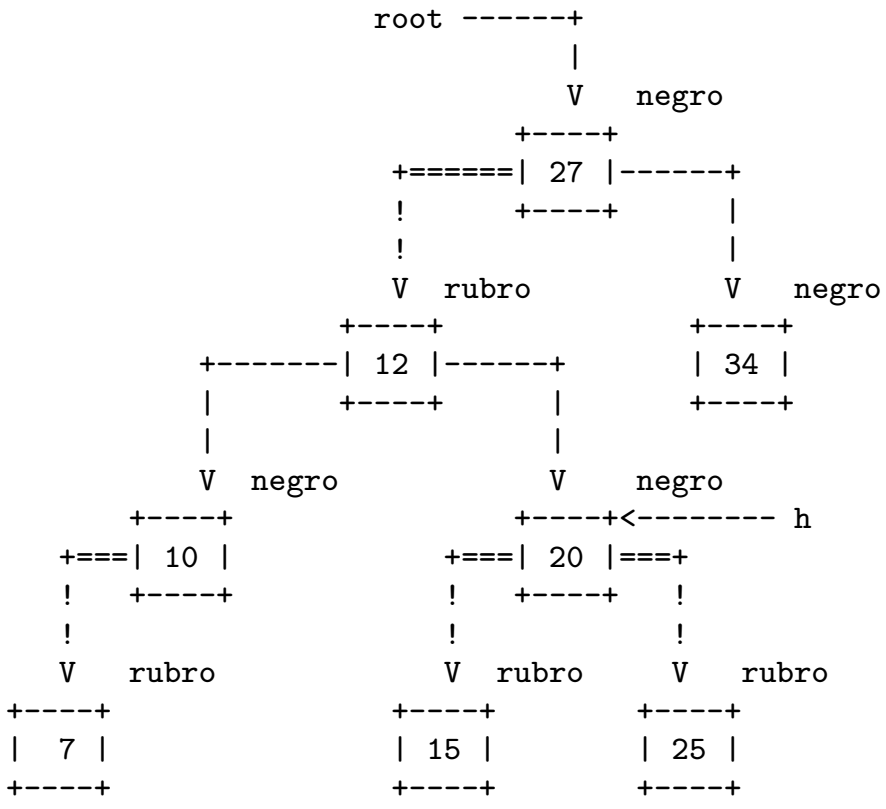


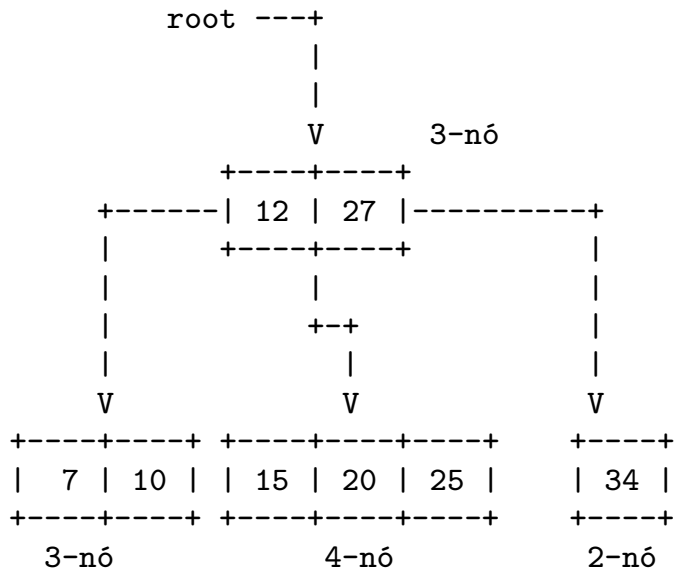
chave é inserida em um 3-nó qualquer



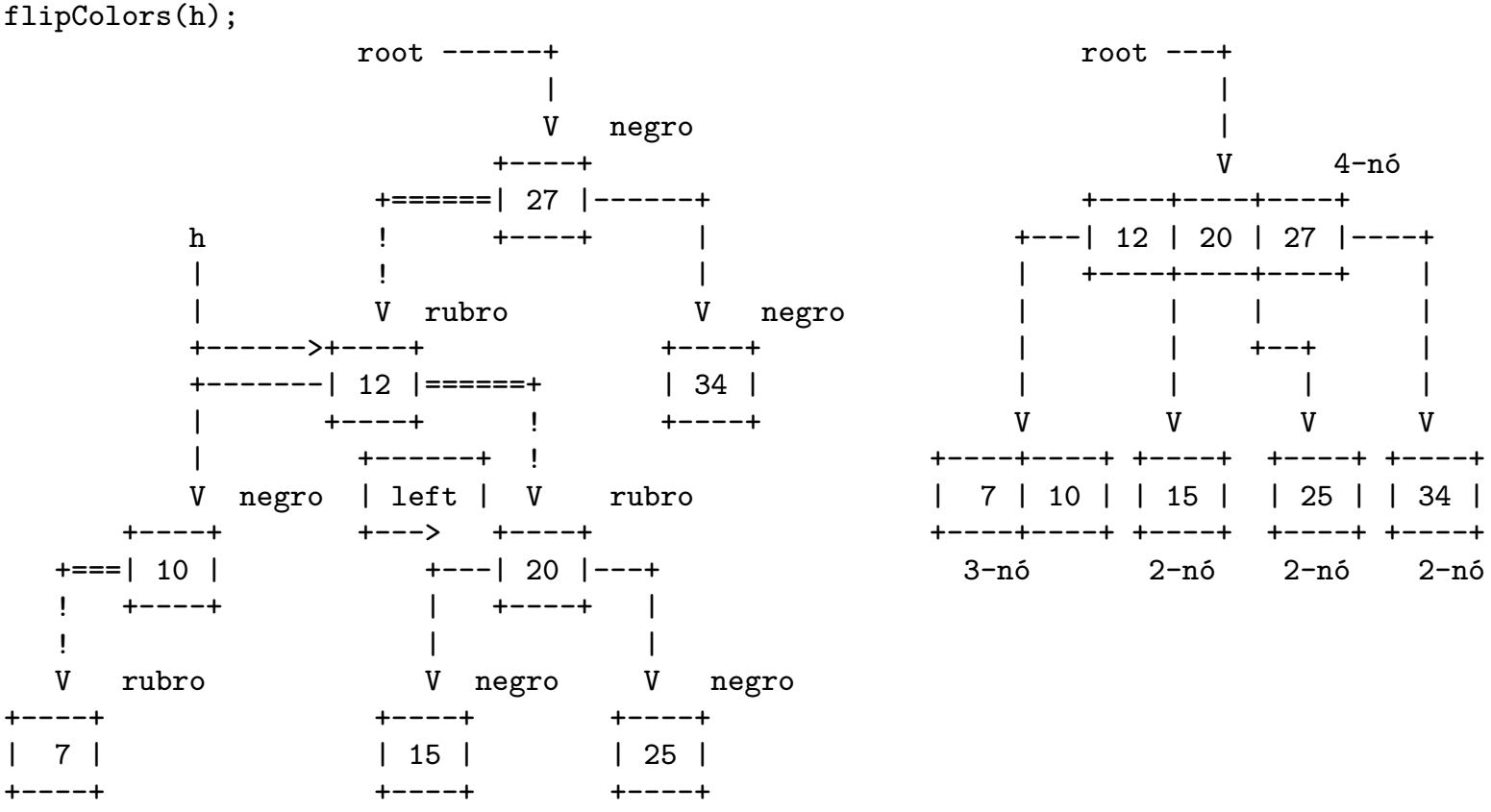
chave é inserida em um 3-nó qualquer

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



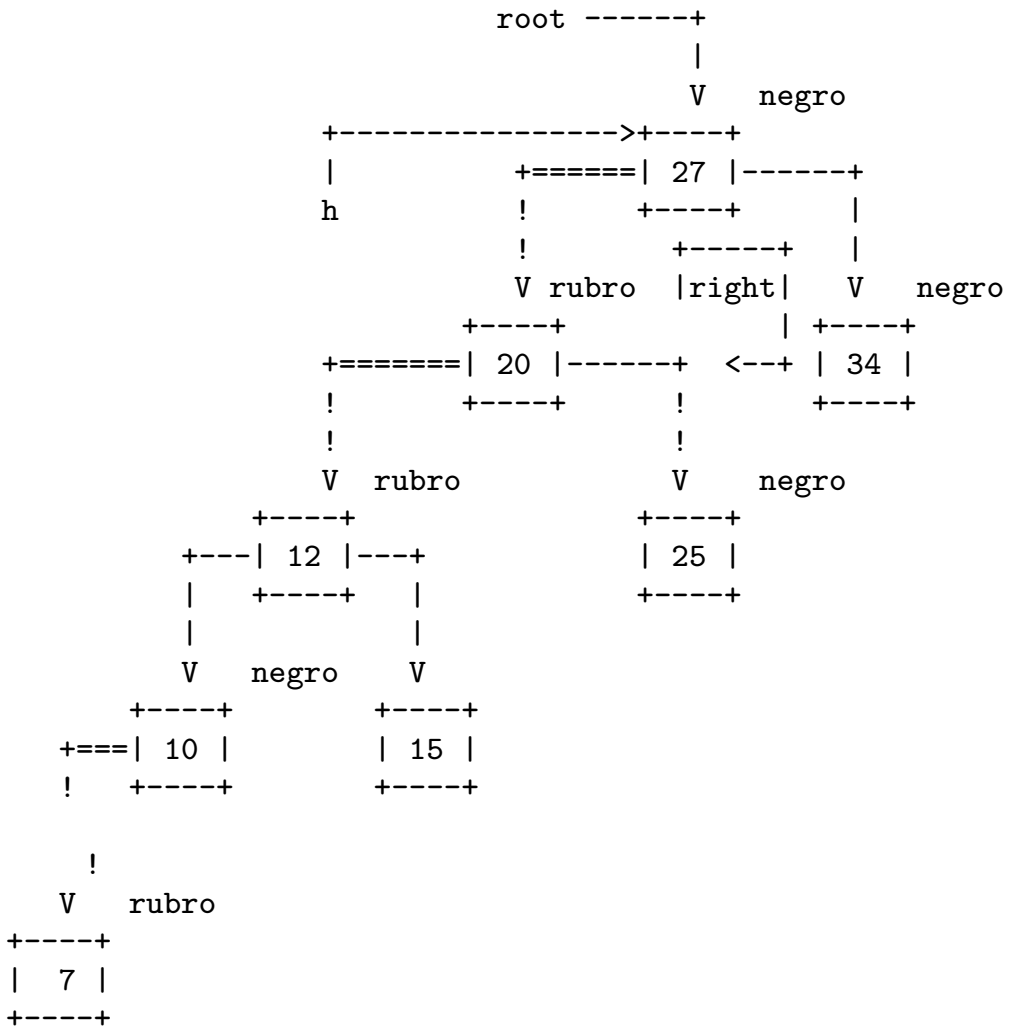


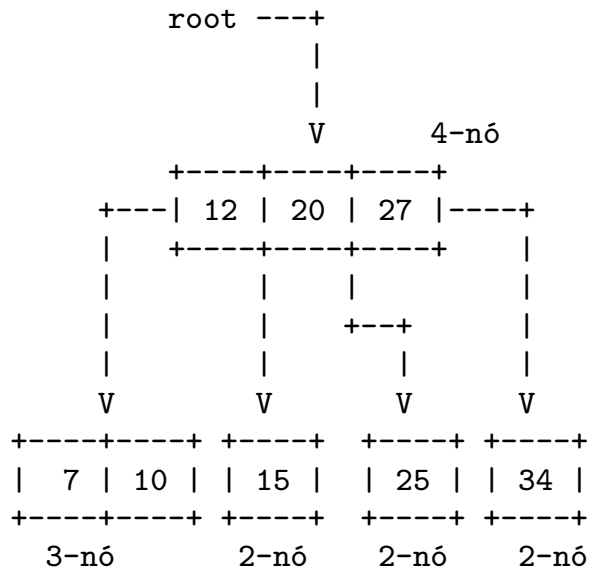
chave é inserida em um 3-nó qualquer



chave é inserida em um 3-nó qualquer

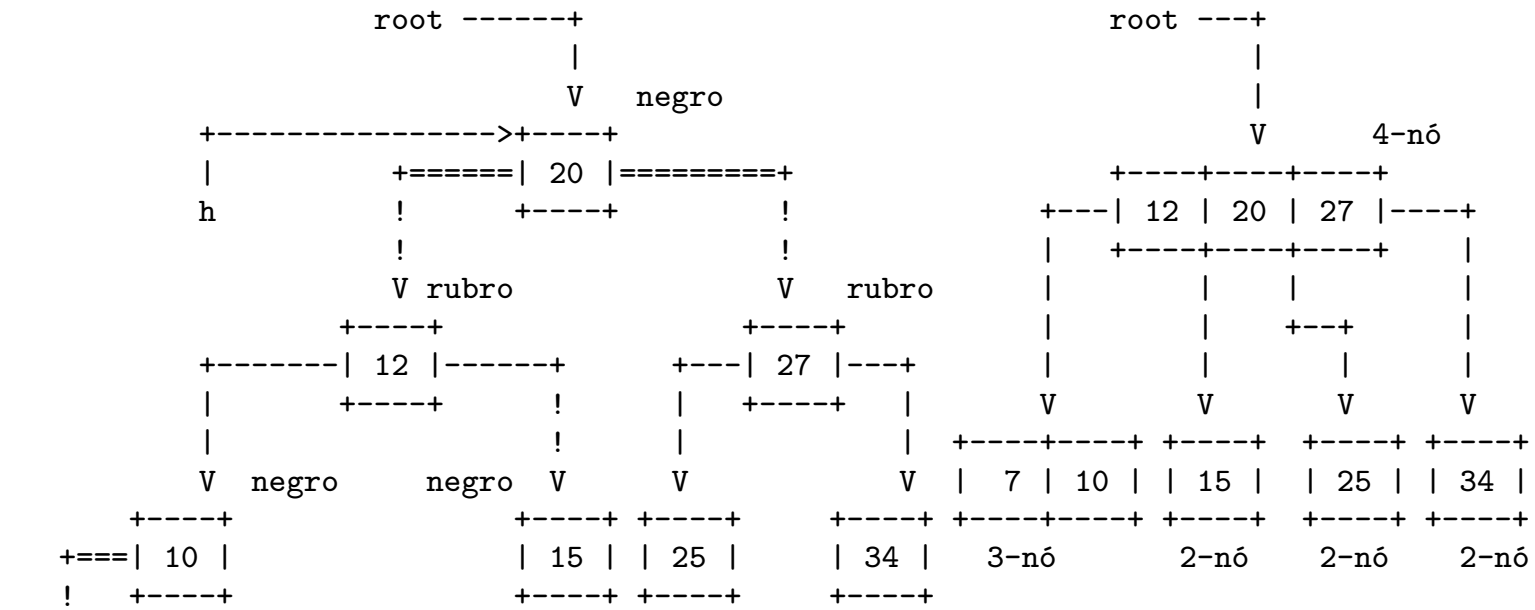
```
h = rotateLeft(h);
```





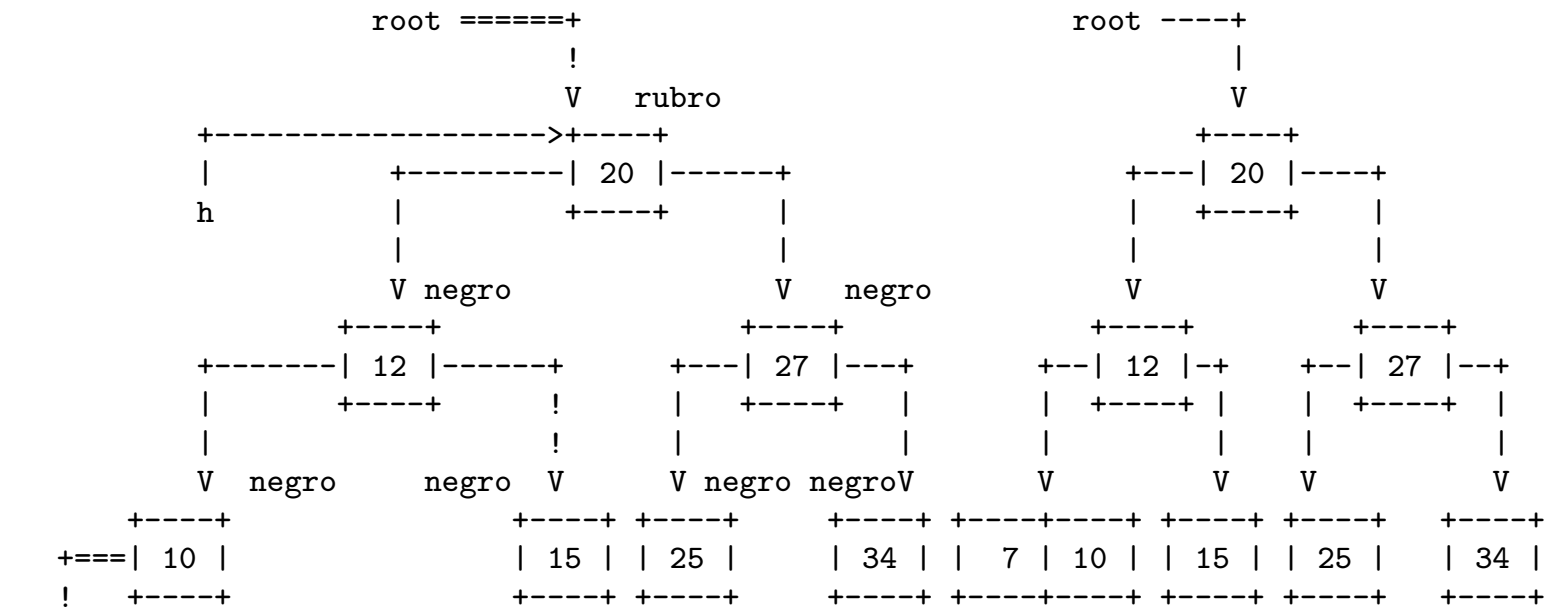
chave é inserida em um 3-nó qualquer

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



chave é inserida em um 3-nó qualquer

```
flipColors(h);
```




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

Rotações

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

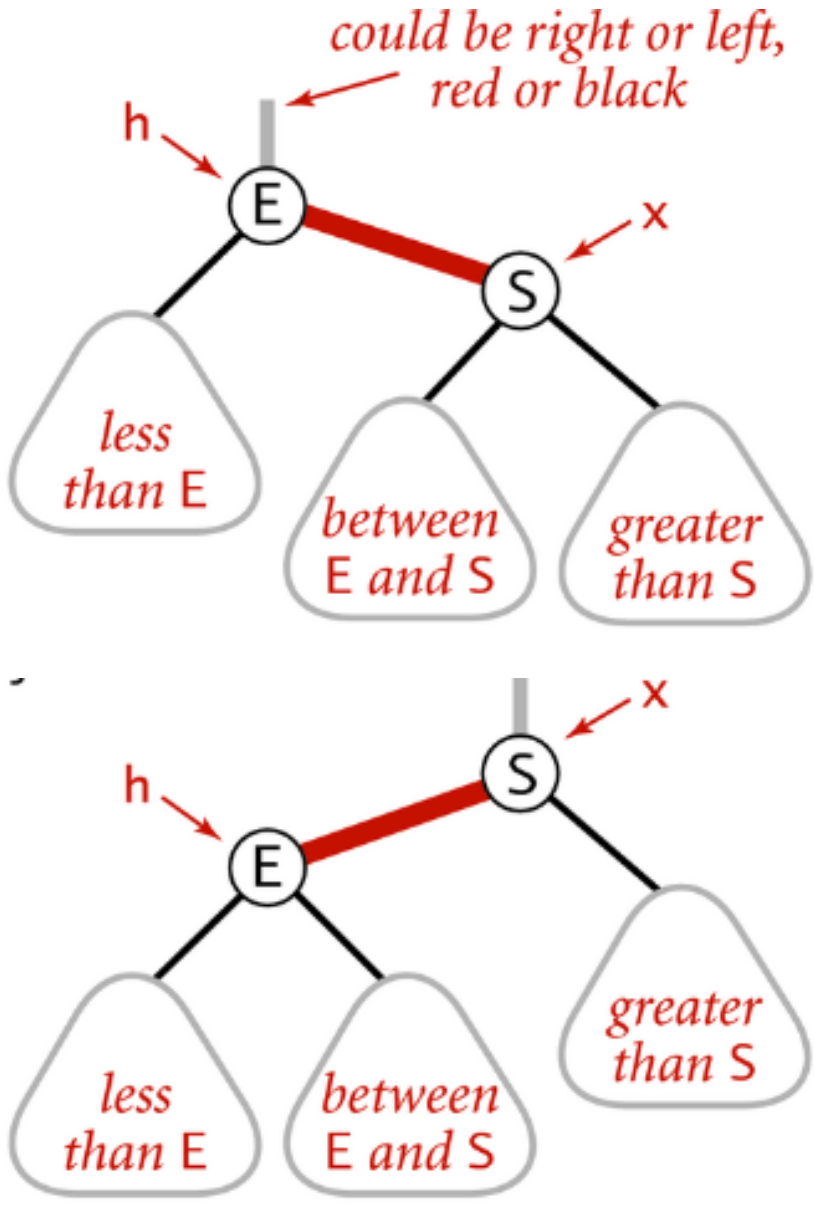
Durante uma operação de inserção, podemos ter, temporariamente, um link rubro inclinado para a direita ou dois links rubros incidindo no mesmo nó.

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.

Rotação esquerda



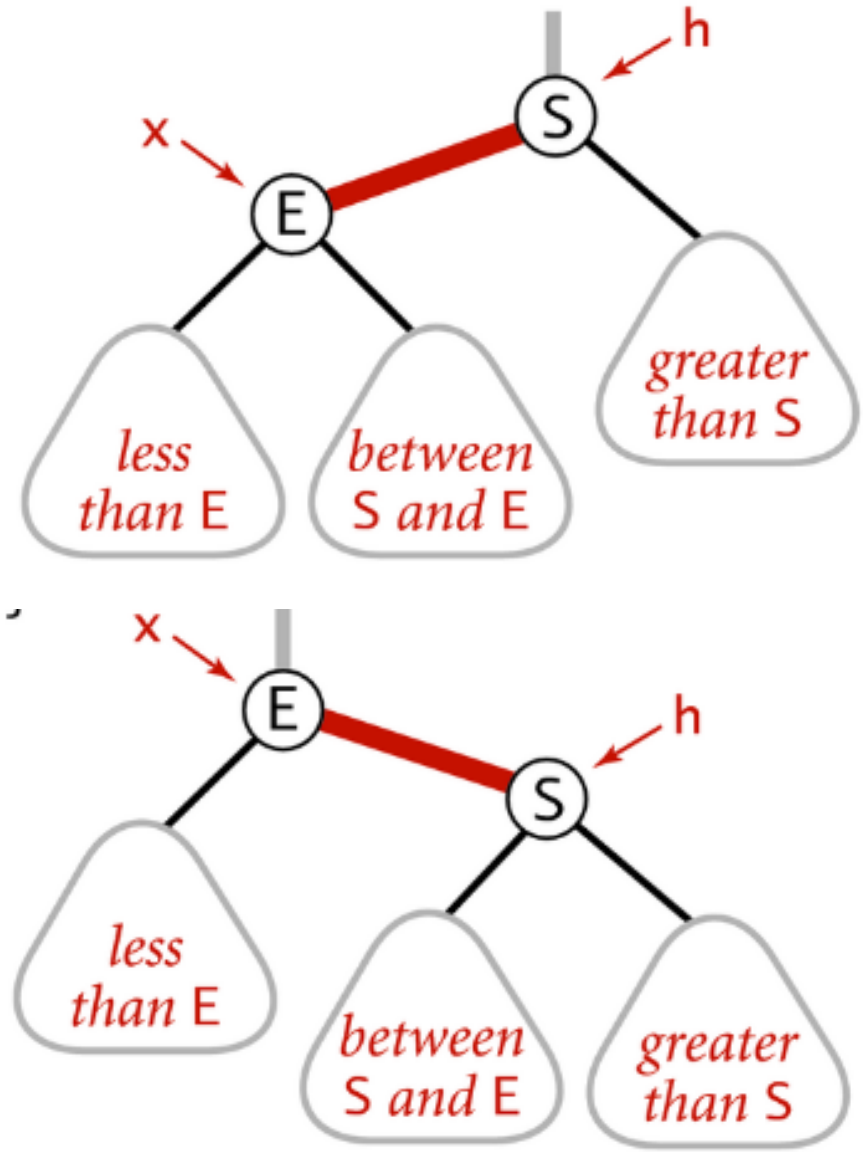
Left rotate (right link of h)

Fonte: algs4

Rotação esquerda

```
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



Right rotate (left link of h)

Fonte: algs4

Rotação direita

```
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;
}
```

Flipping colors

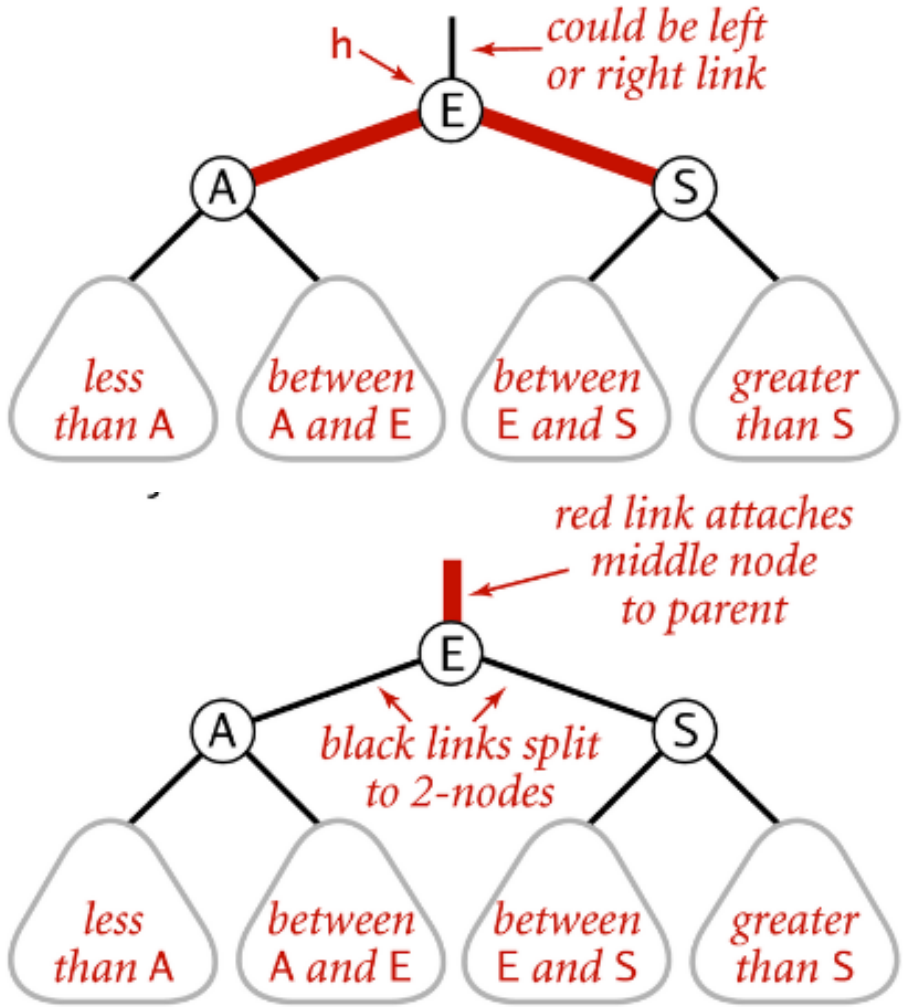
As operações de rotação são locais.

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.

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.

Flipping colors



Flipping colors to split a 4-node

Fonte: algs4

Flipping Colors

Troca as cores de um nó e dos seus filhos.

A cor de h deve ser diferente da de seus dois filhos.

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

RedBlackBST

```
public class RedBlackBST<Key extends
                Comparable<Key>, Value> {
   private Node r;
   private class Node {...}
   private boolean isRed(Node h) {...}
   private Node rotateLeft(Node h) {...}
   private Node rotateRight(Node h) {...}
   private void flipColors(Node h) {...}
   private int size() {...}
```

RedBlackBST

```
public void put(Key key, Value val) {
   r = put(r, key, val);
   r.color = BLACK;
}
```

RedBlackBST

```
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;
   h = balance(h); // mantenha rubro-negra
   return h;
}
```

RedBlackBST

Verifica invariante rubro-negro quando estamos voltando da recursão.

```
private Node balance(Node h) {
   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;
}
```

BSTs rubro-negras: `delete()`

Fonte: `.../only-one/red-leaves-black-tree/`

Referências: BSTs rubro-negras (PF); Balanced Search Trees (S&W); slides (S&W)

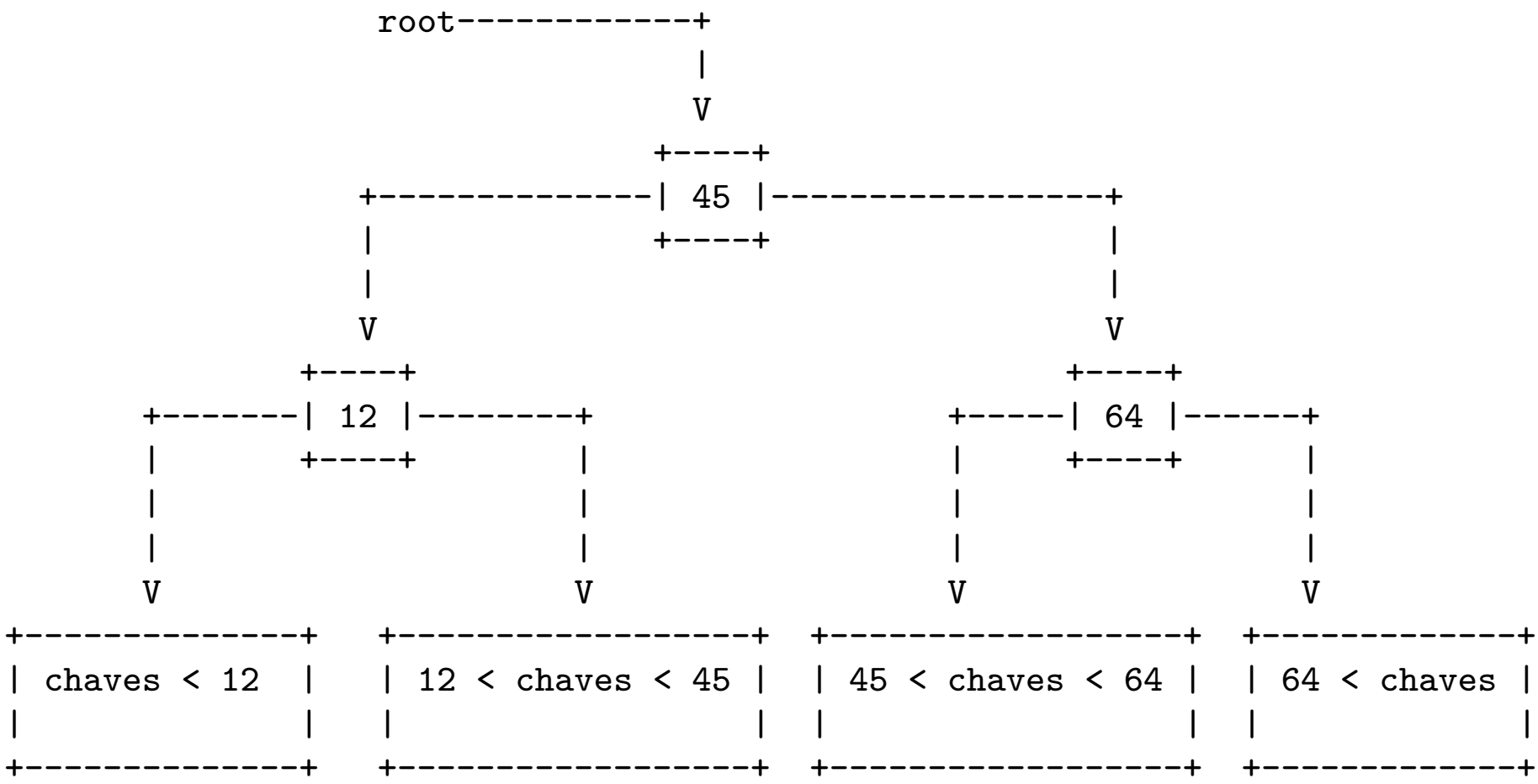
Remoção em árvore 2-3

No caminho até a chave a ser removida o algoritmo mantém a relação invariante em relação à árvore 2-3:

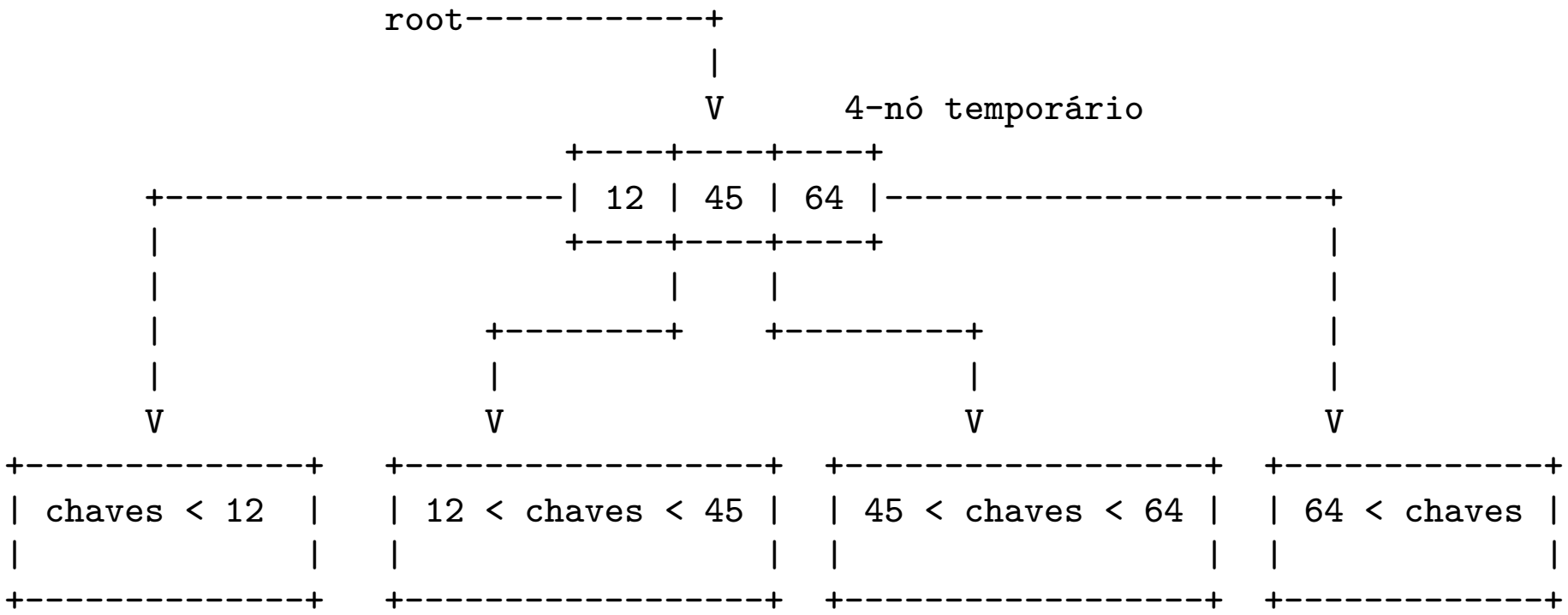
> *o nó sendo examinado é um 3-nó ou um 4-nó (temporário)*

`deleteMin()`: comecemos com a raiz quando os dois filhos são 2-nós.

Os dois filhos da raiz são 2-nós

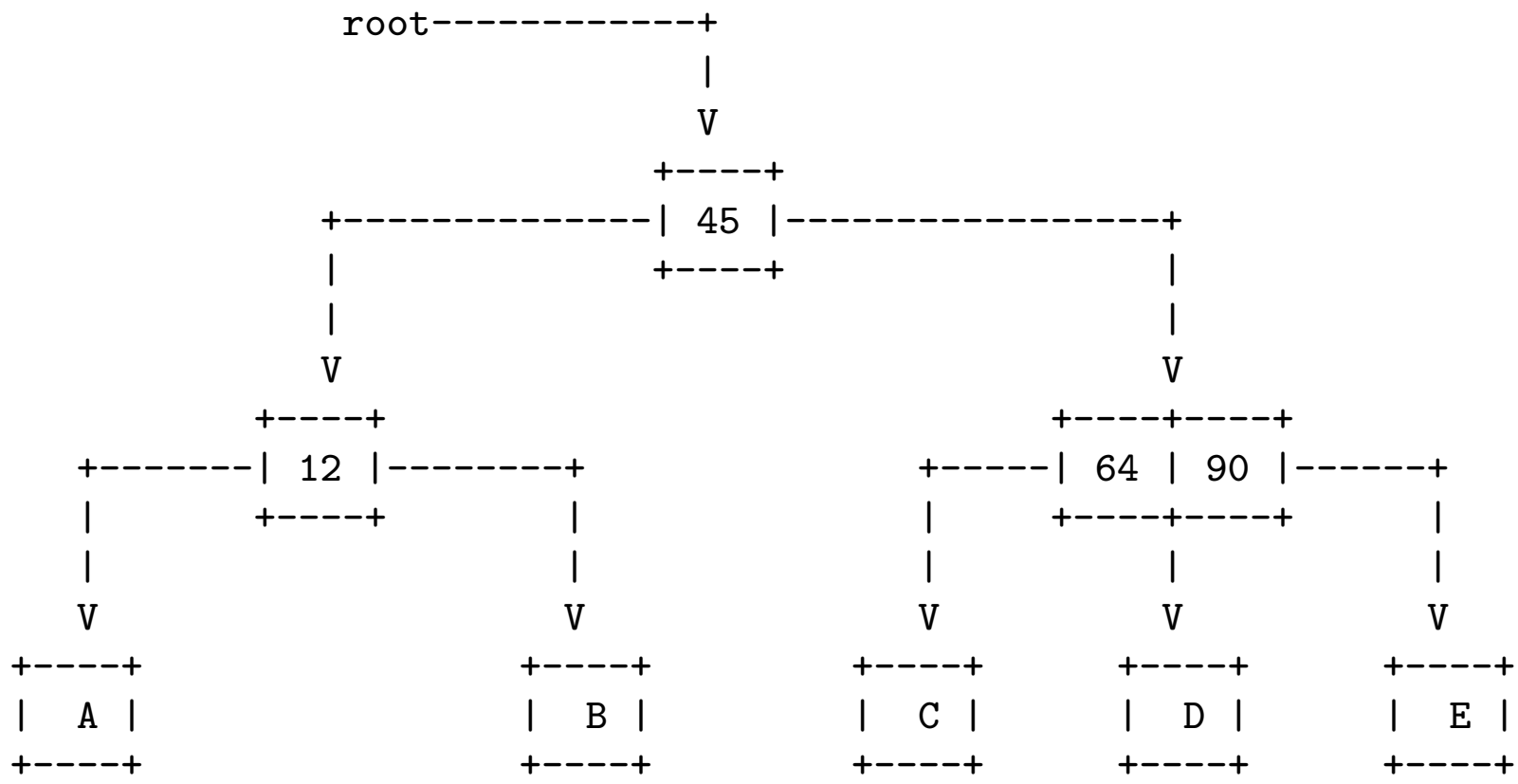


Os dois filhos da raiz são 2-nós

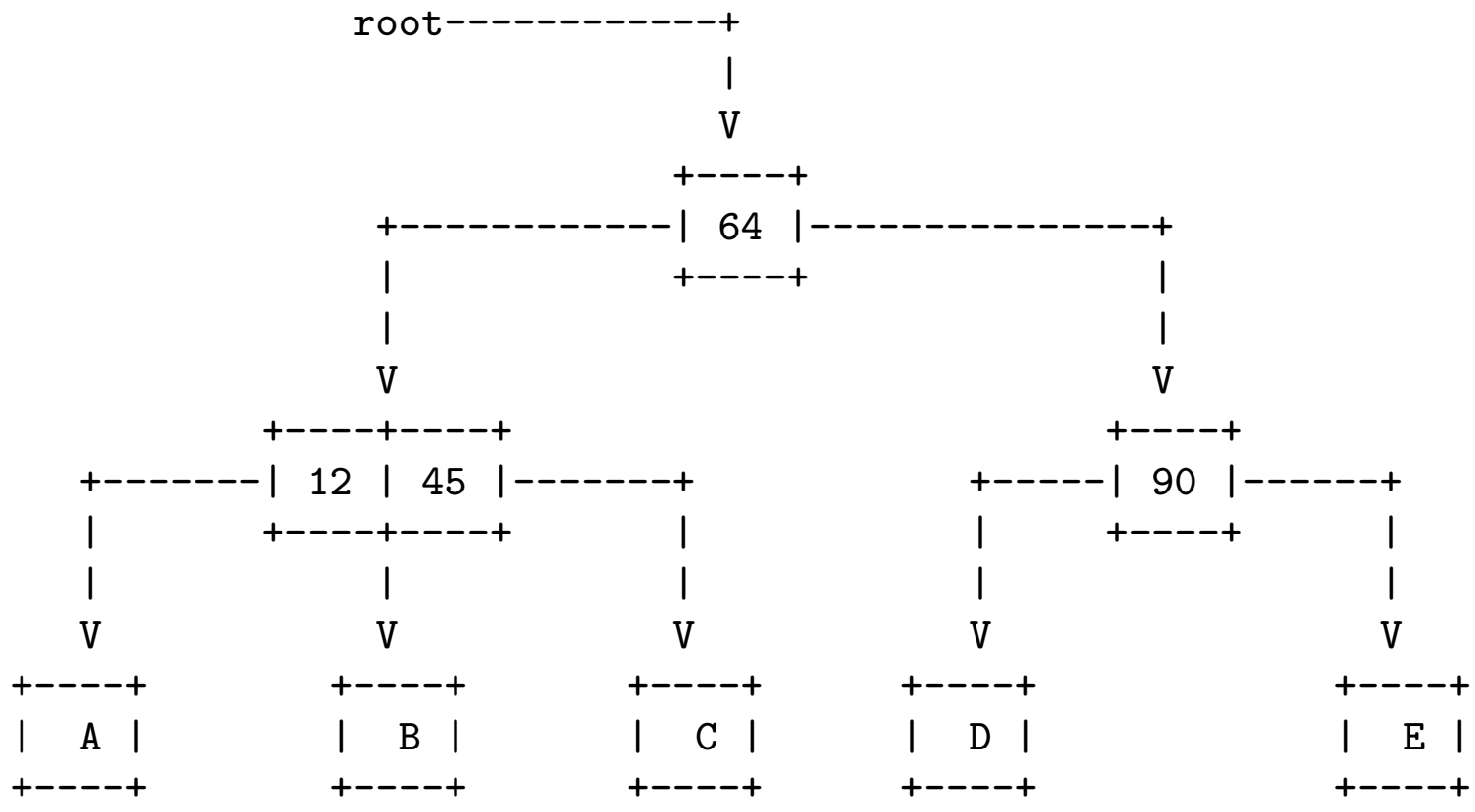


Agora passemos a raiz quando apenas o nó esquerdo é um 2-nó.

Filhos esquerdo da raiz é 2-nós



Filhos esquerdo da raiz é 2-nós



Mover para esquerda

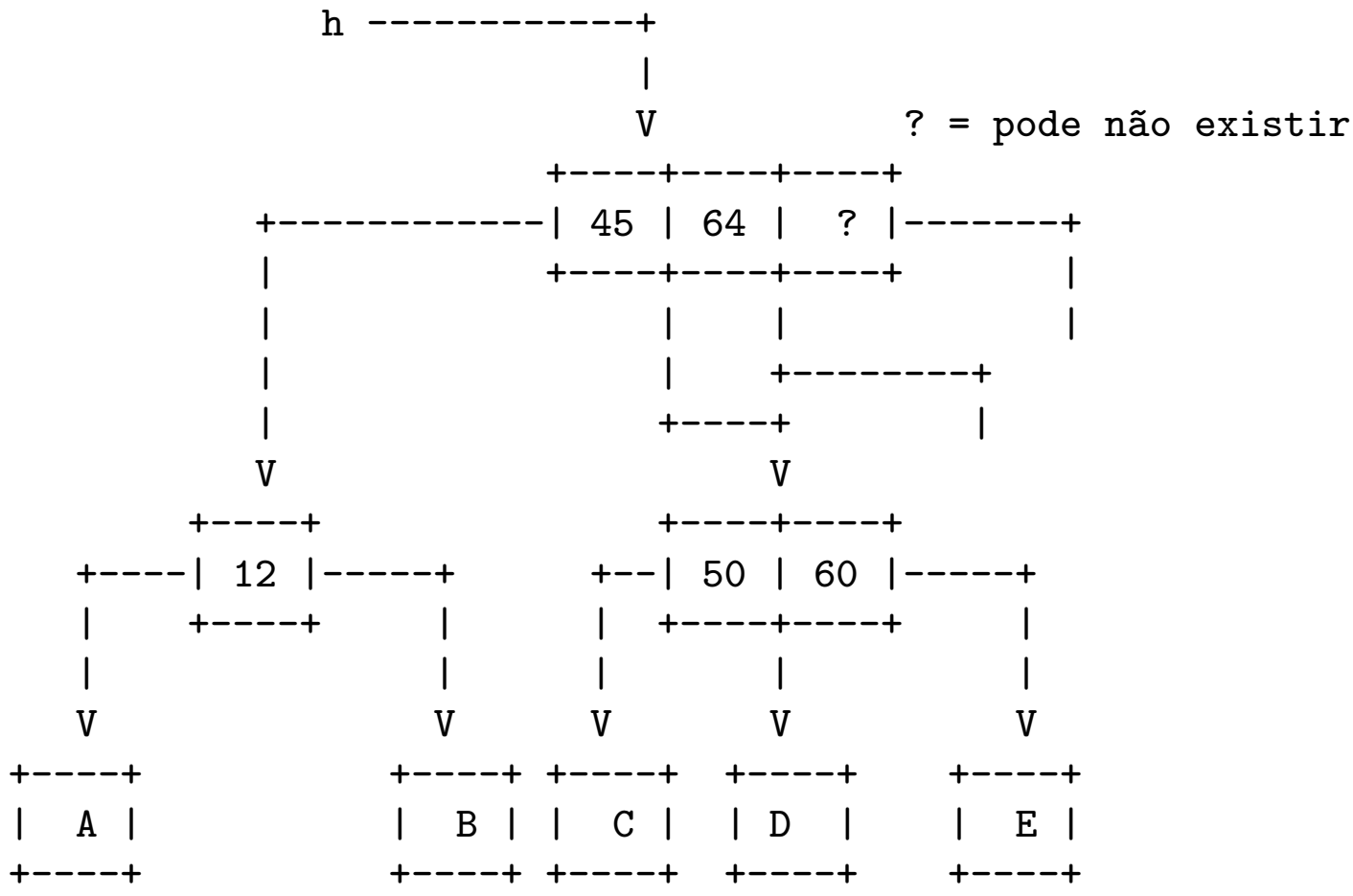
No meio do caminho, sabemos que o nó corrente `h` é um 3-nó ou um 4-nó.

Antes de movermos para o nó mais à esquerda precisamos nos certificar que esse nó é um 3-nó ou 4-nó.

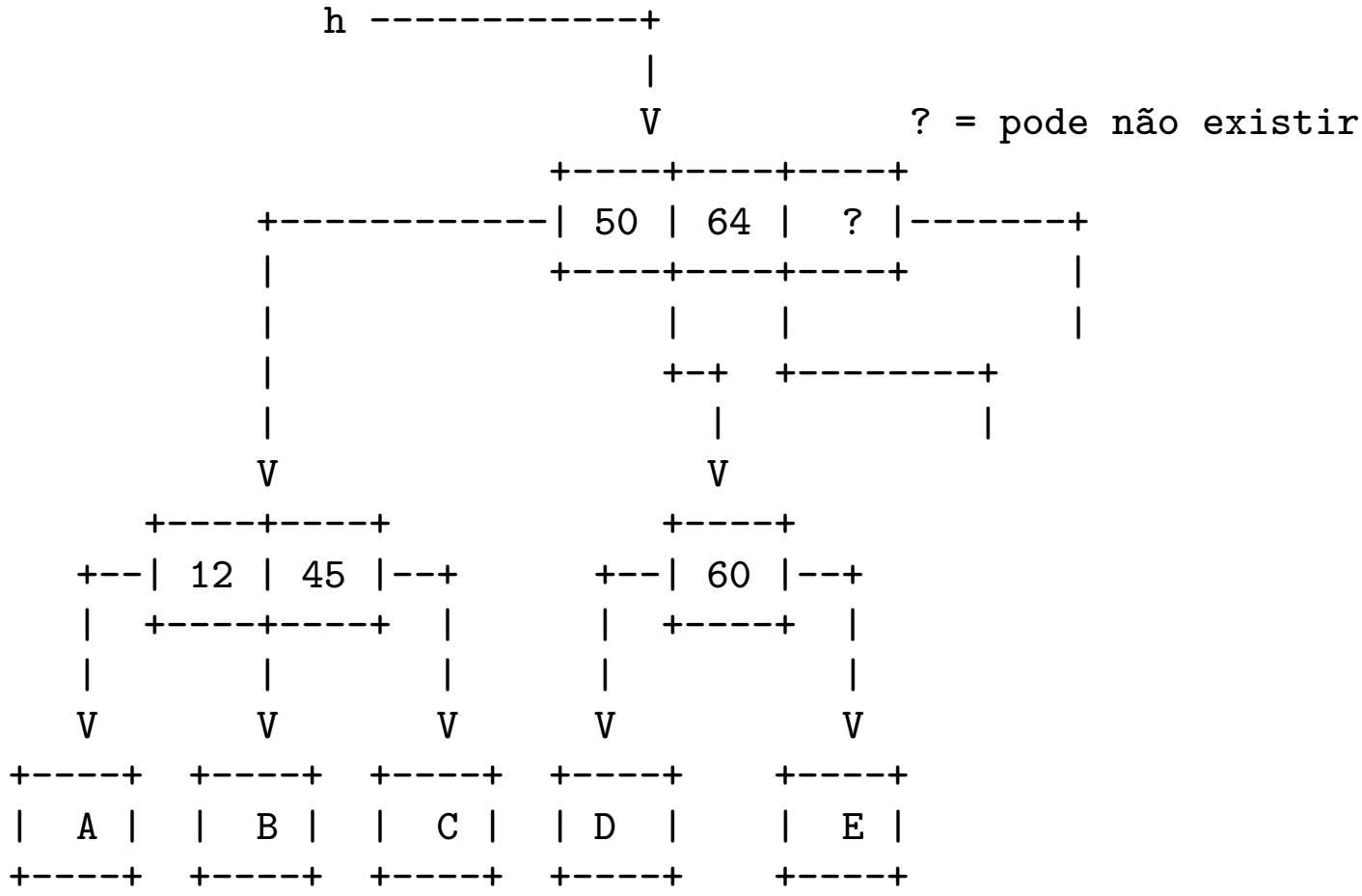
Se ele já é um 3-nó, não precisamos fazer nada.

Comecemos com o caso em que o nó mais a esquerda de `h` é um 2-nó.

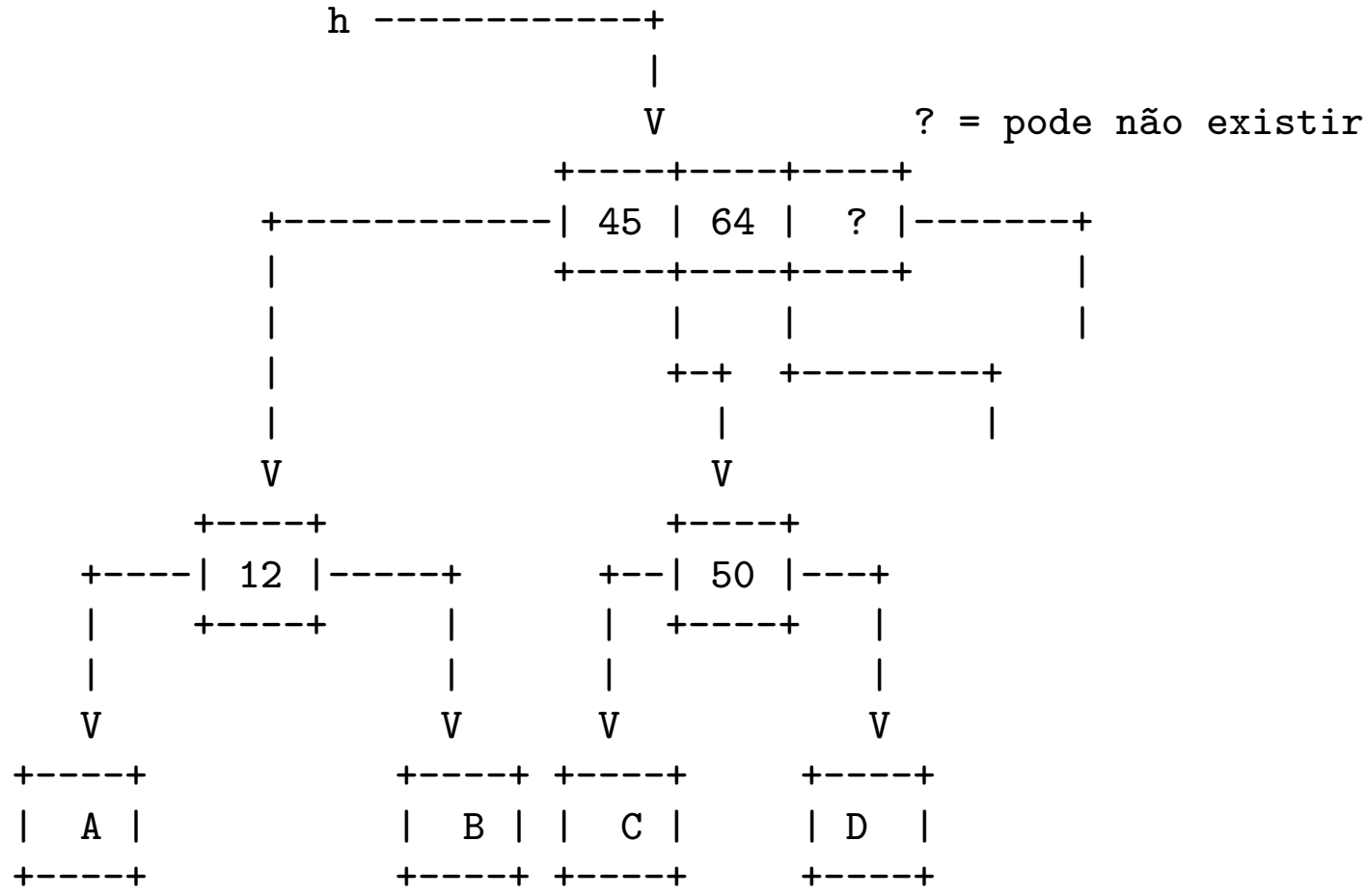
Filho mais a esquerda de h é 2-nó



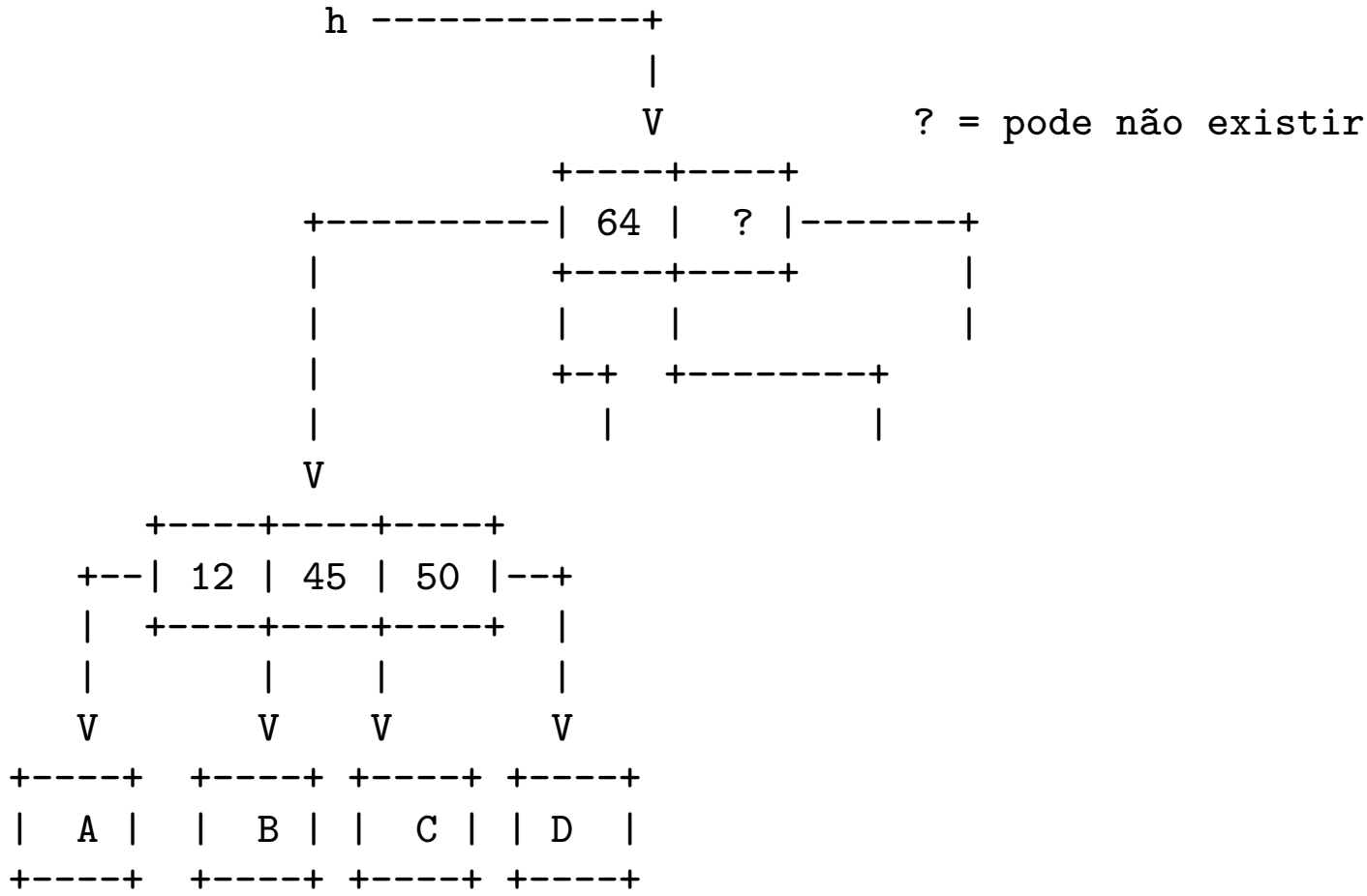
Filho mais a esquerda de h é 2-nó



Dois filhos mais à esquerda de h são 2-nós



Dois filhos mais à esquerda de h são 2-nós



Finalmente...

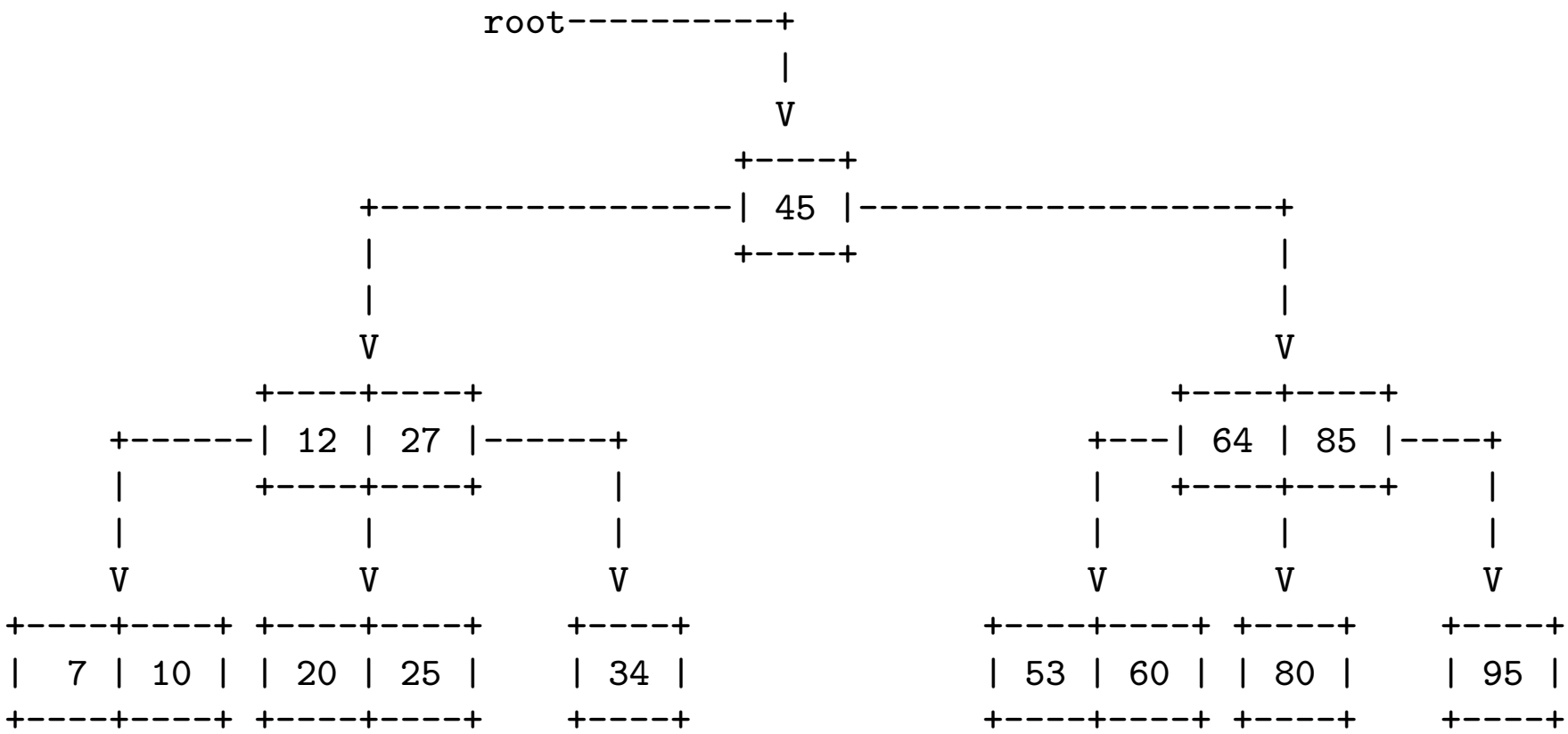
Desta forma quando atingirmos a folha mais a esquerda dessa árvore 2-3-4 teremos chegado a um 3-nó ou 4-no.

Removendo a item mais à esquerda teremos um 2-nó ou 3-nó.

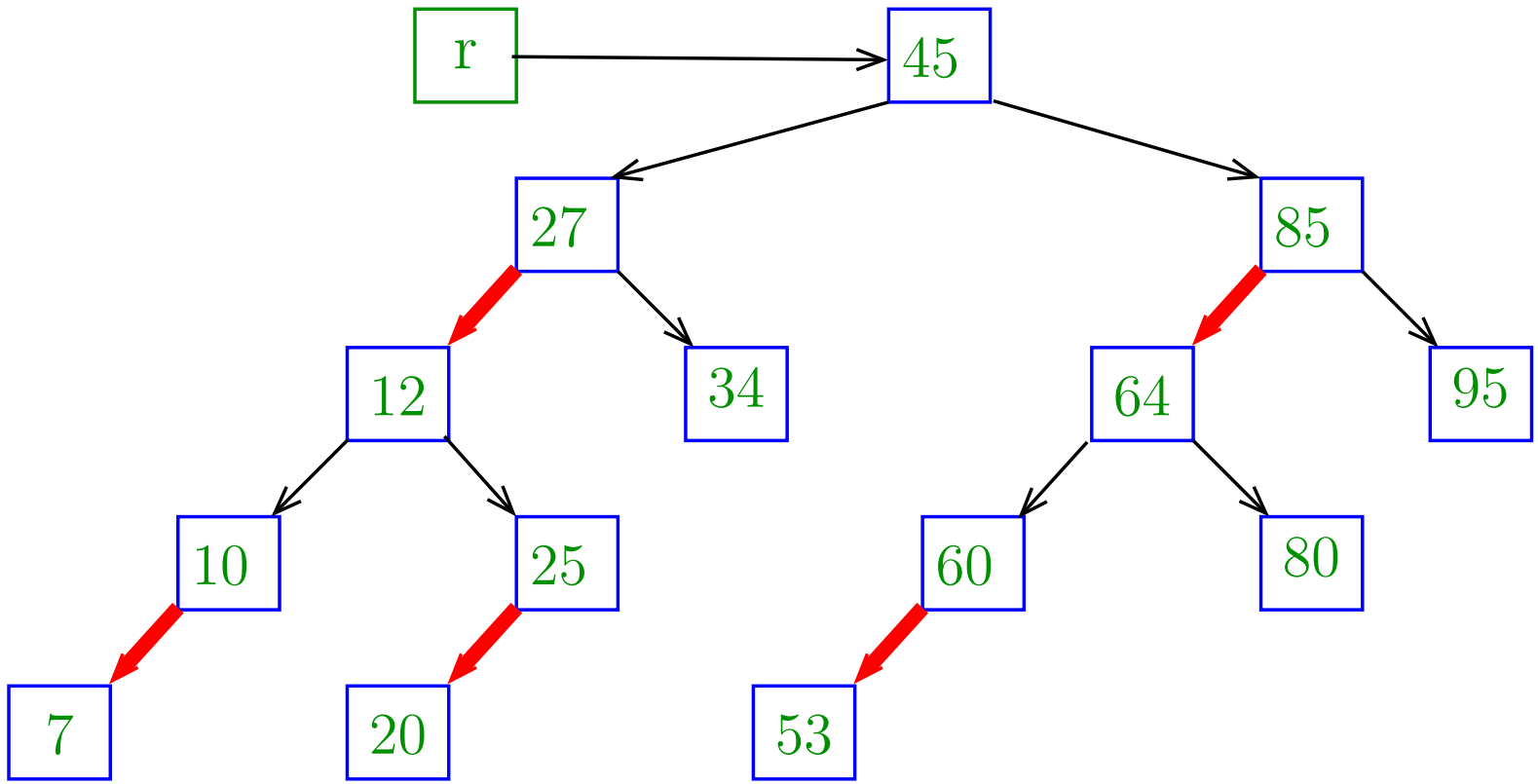
Depois devemos voltar "espatifando" os 4-nós que por ventura deixamos pelo caminho.

Hmm. Talvez seja importante notar que o pai de um 4-nó deixado pelo caminho, que não seja a raiz, é um 2-nó ou 3-nó.

Árvore 2-3: `deleteMin()`

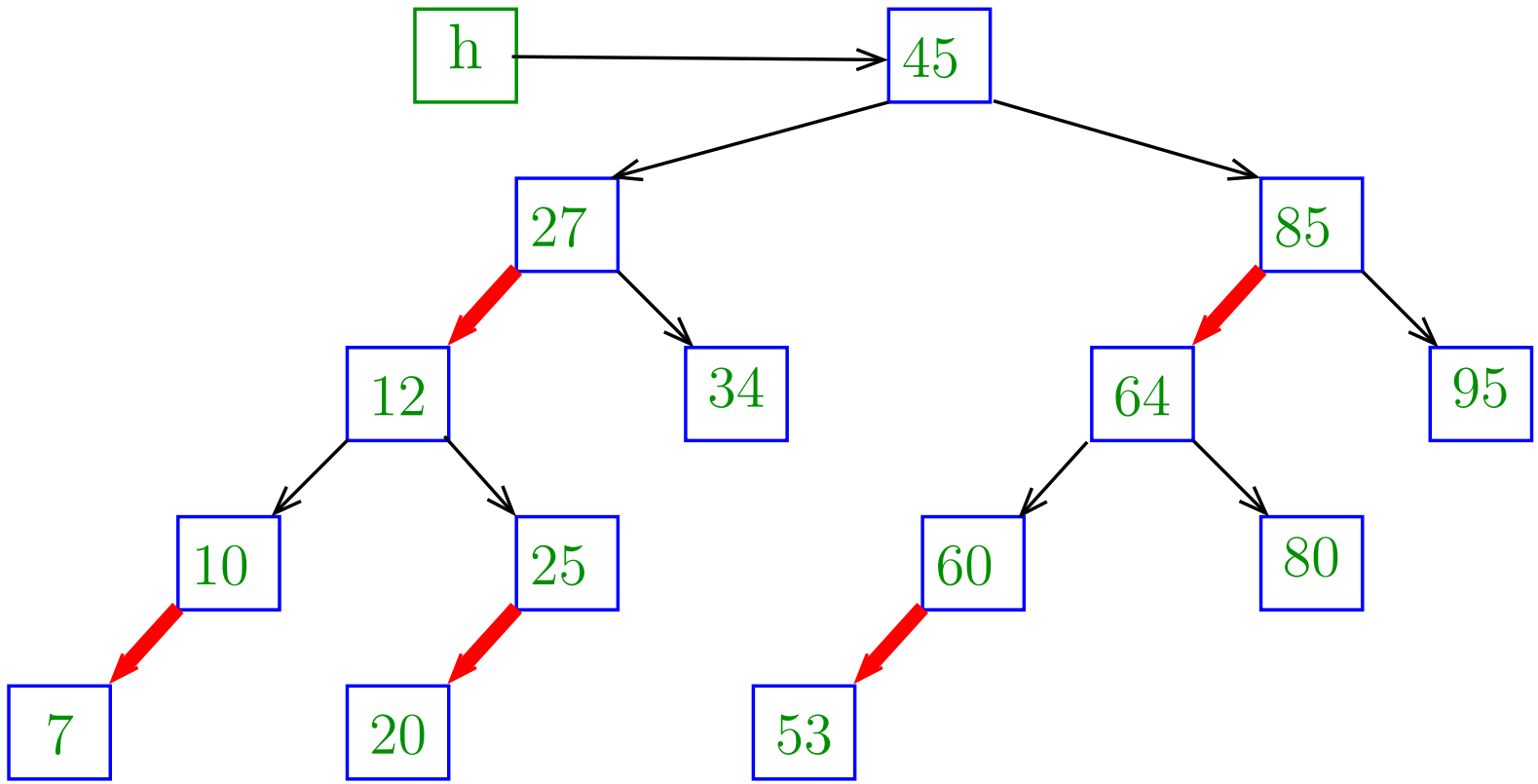


rubro-negra: `deleteMin()`



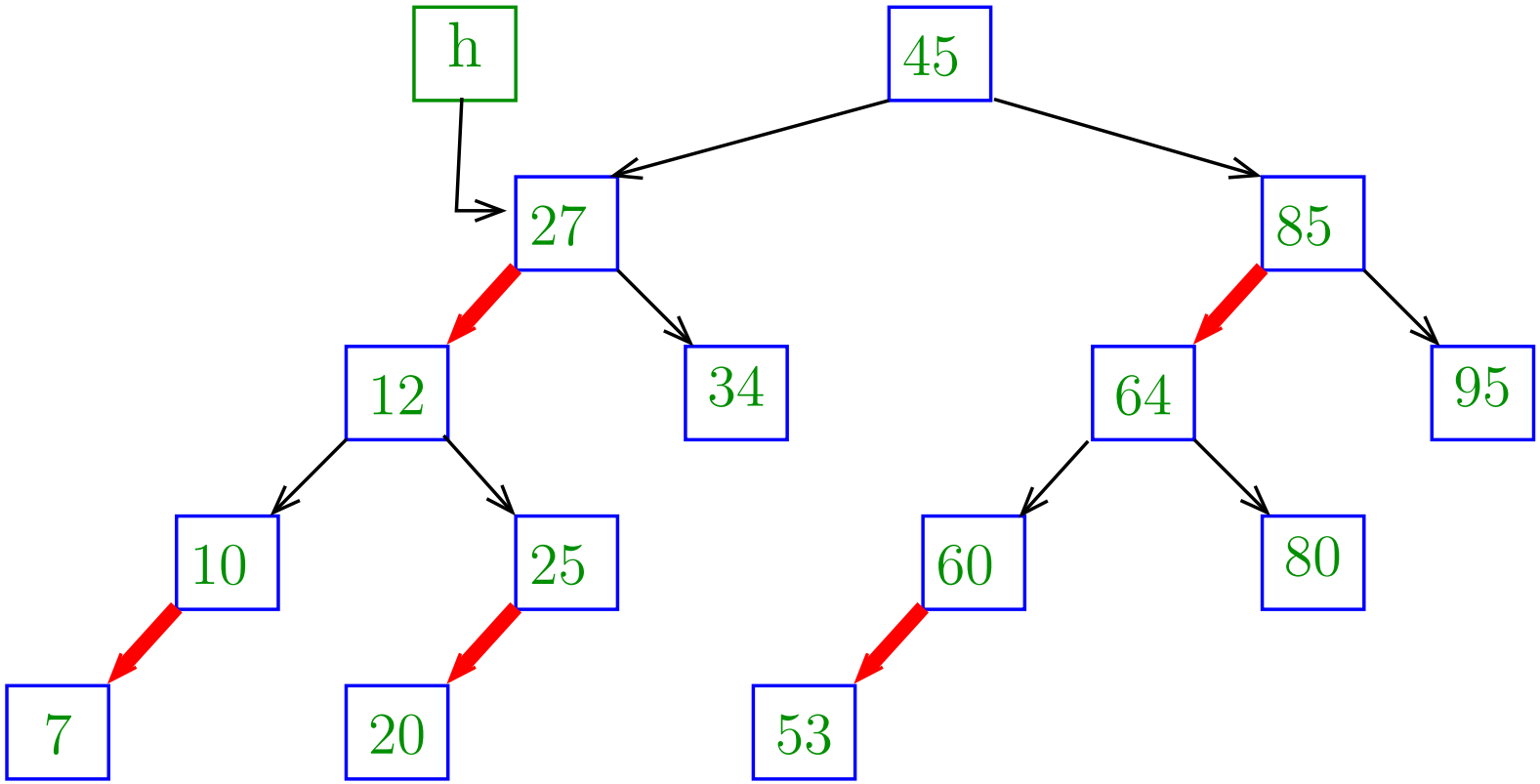
`deleteMin(r);`

rubro-negra: deleteMin()



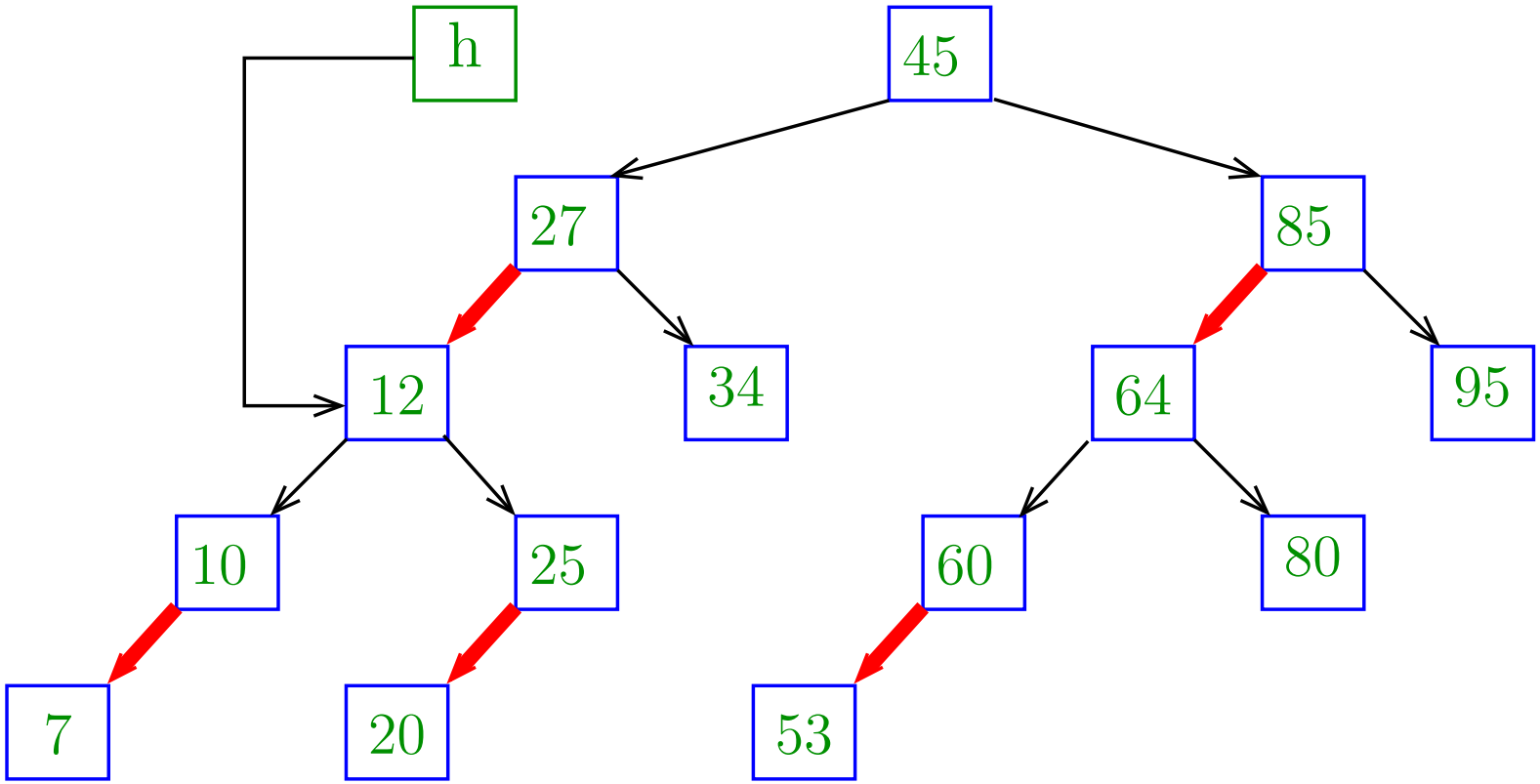

```
h.left = deleteMin(h.left);
```

rubro-negra: deleteMin()



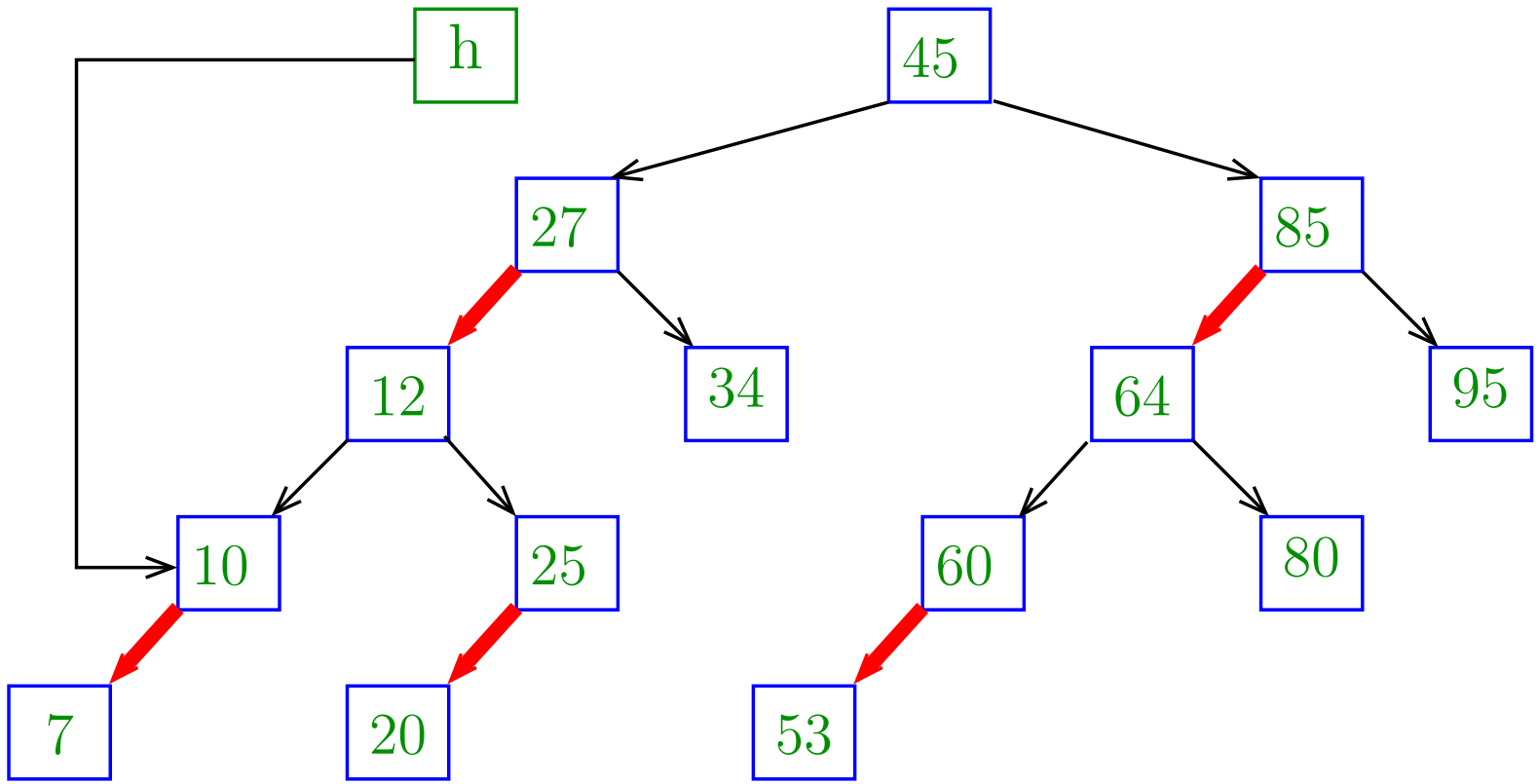

```
h.left = deleteMin(h.left);
```

rubro-negra: deleteMin()



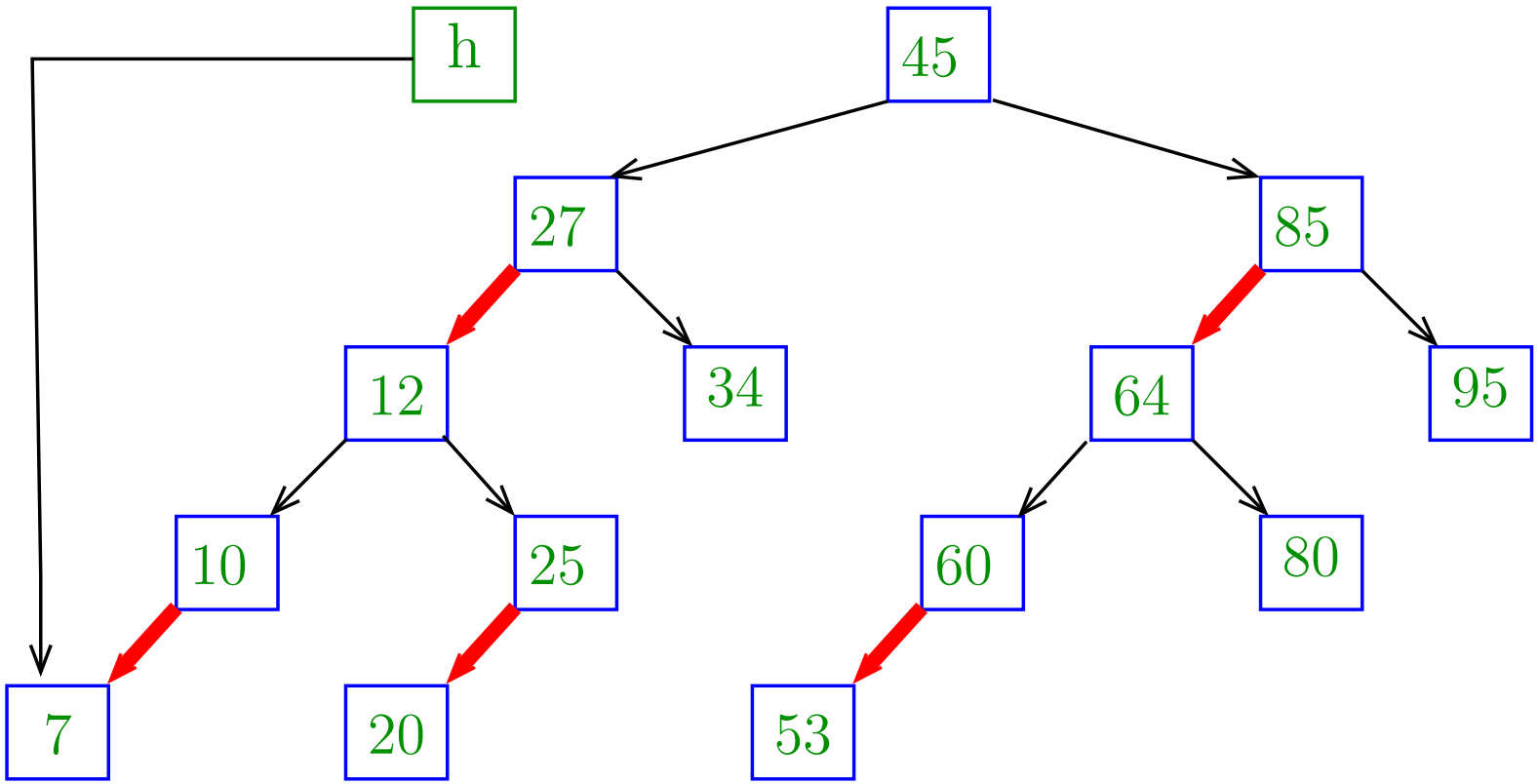

```
h.left = deleteMin(h.left);
```

rubro-negra: deleteMin()



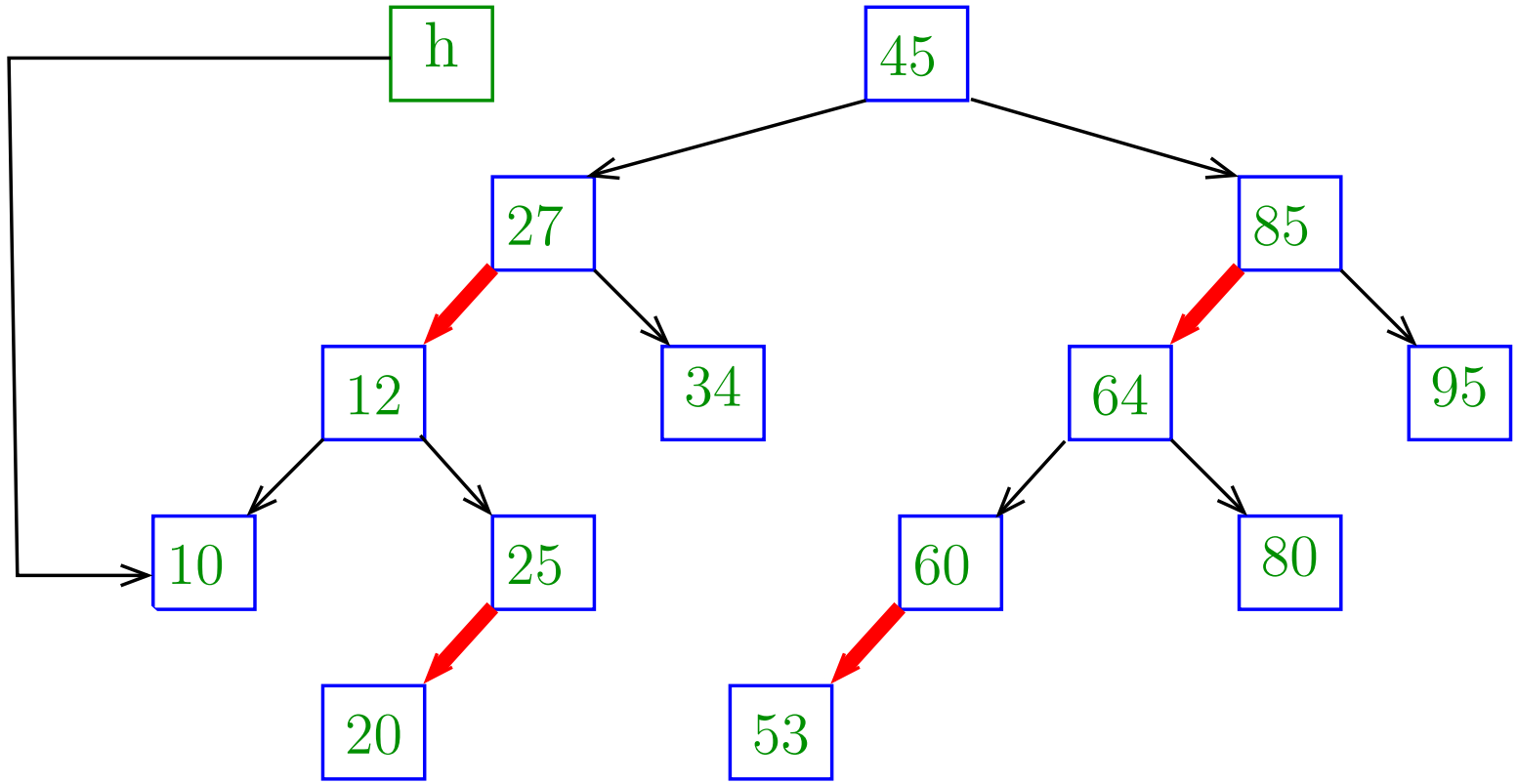

```
h.left = deleteMin(h.left);
```

rubro-negra: deleteMin()



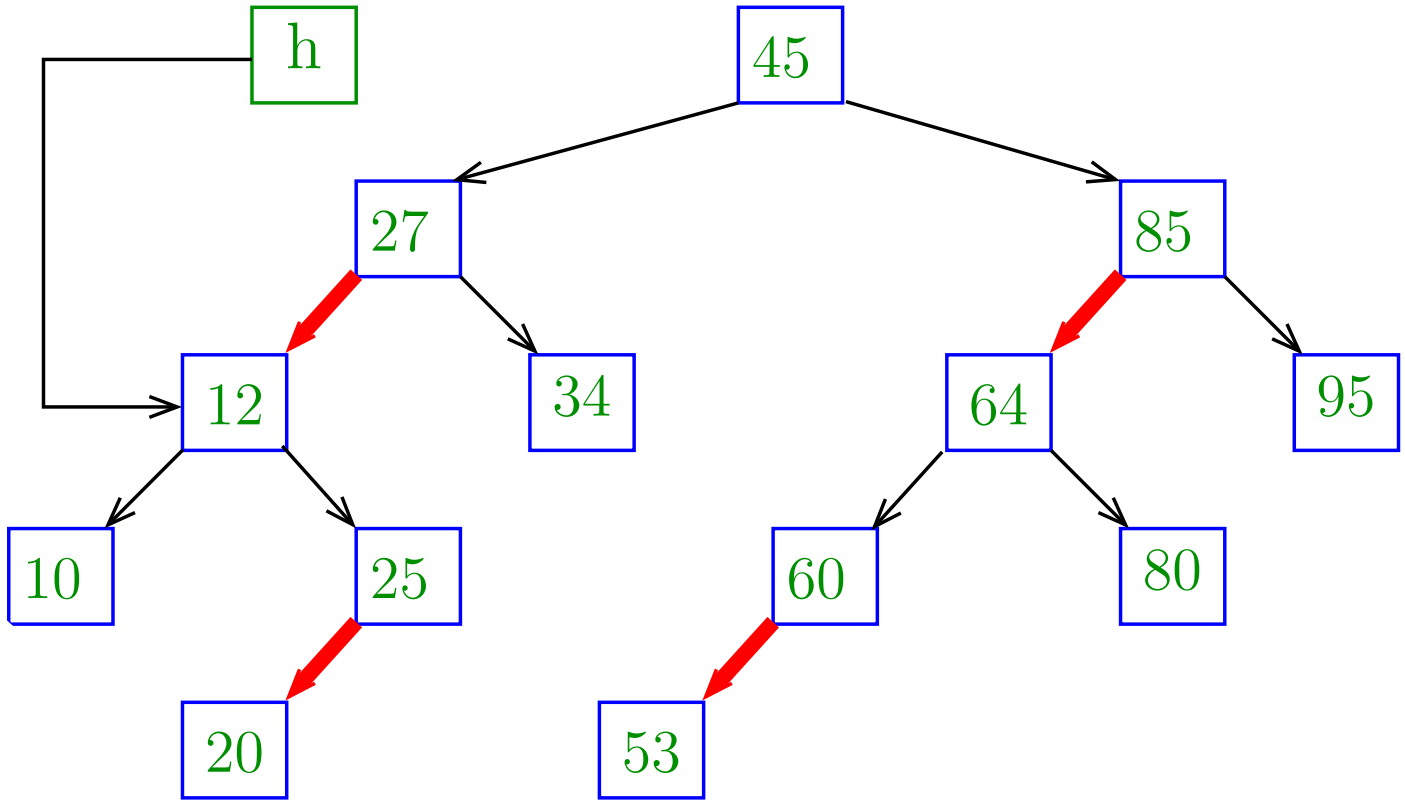

```
return null;
```

rubro-negra: deleteMin()



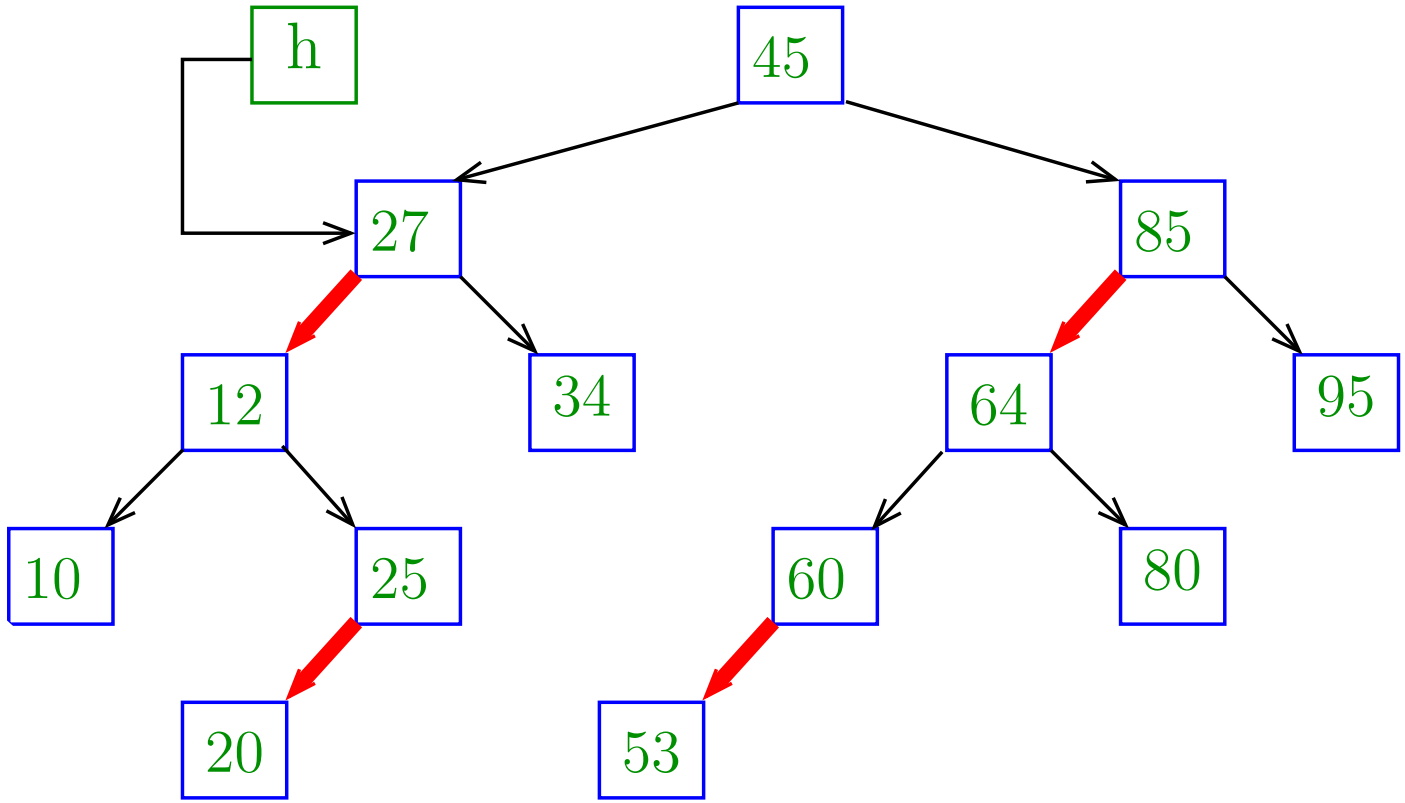

```
return balance(h);
```

rubro-negra: deleteMin()



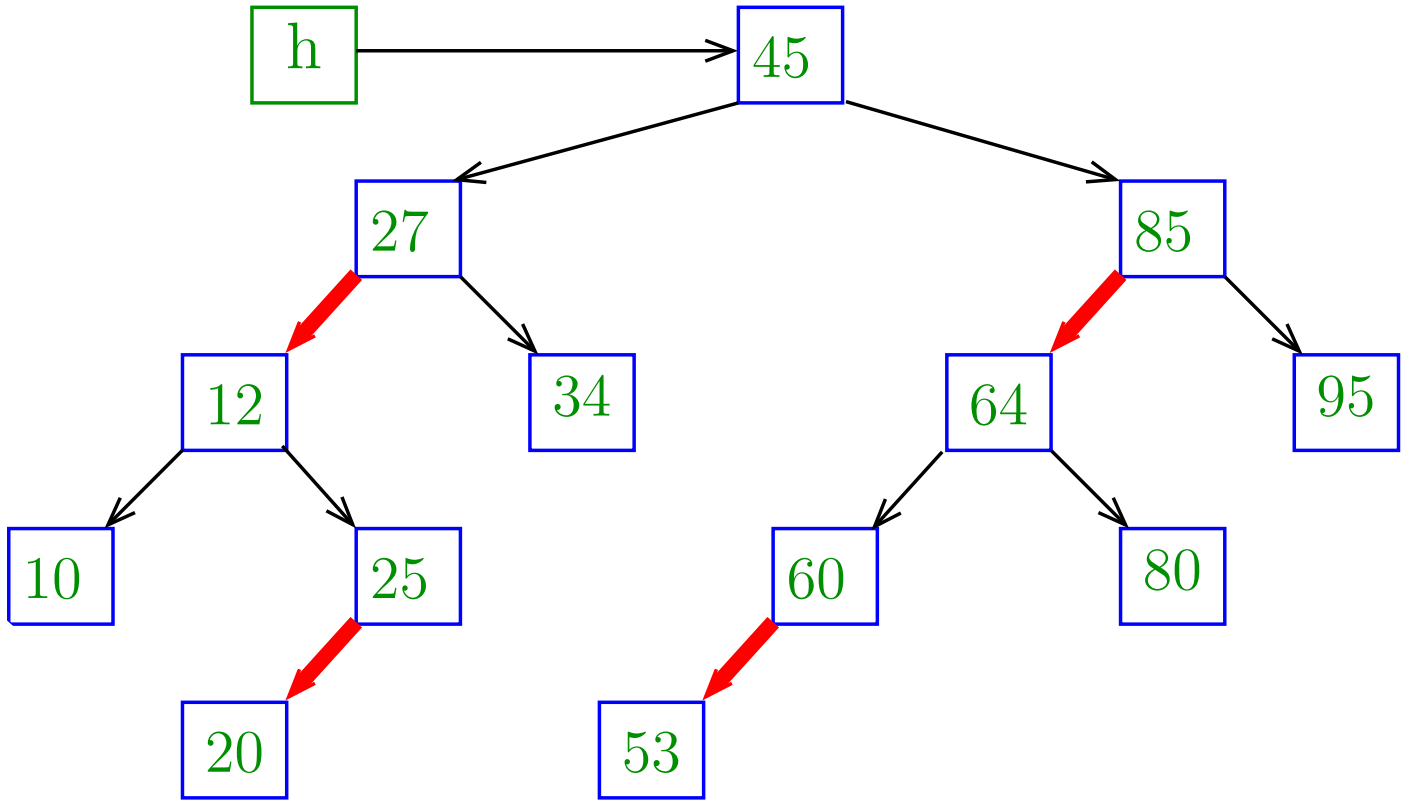

```
return balance(h);
```

rubro-negra: deleteMin()



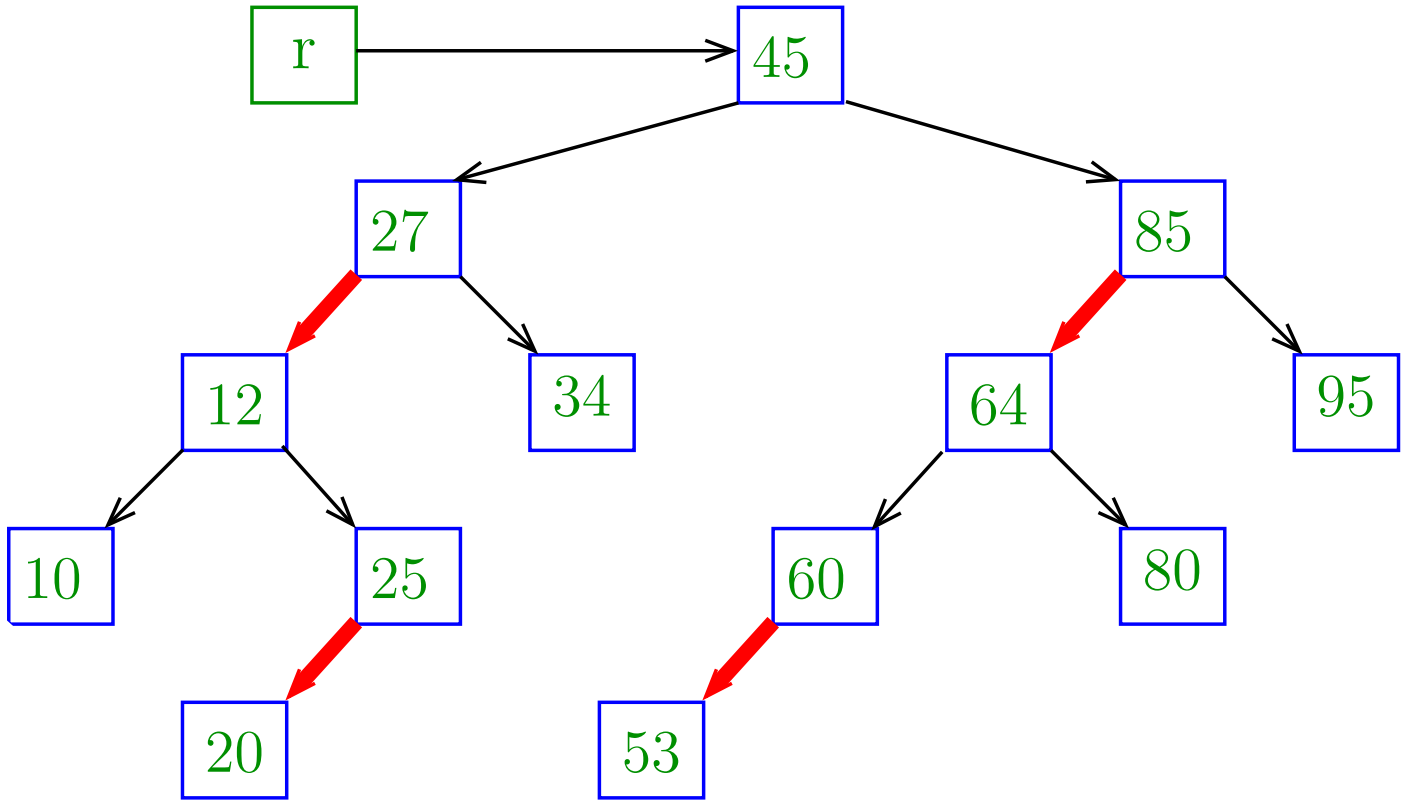

```
return balance(h);
```

rubro-negra: deleteMin()

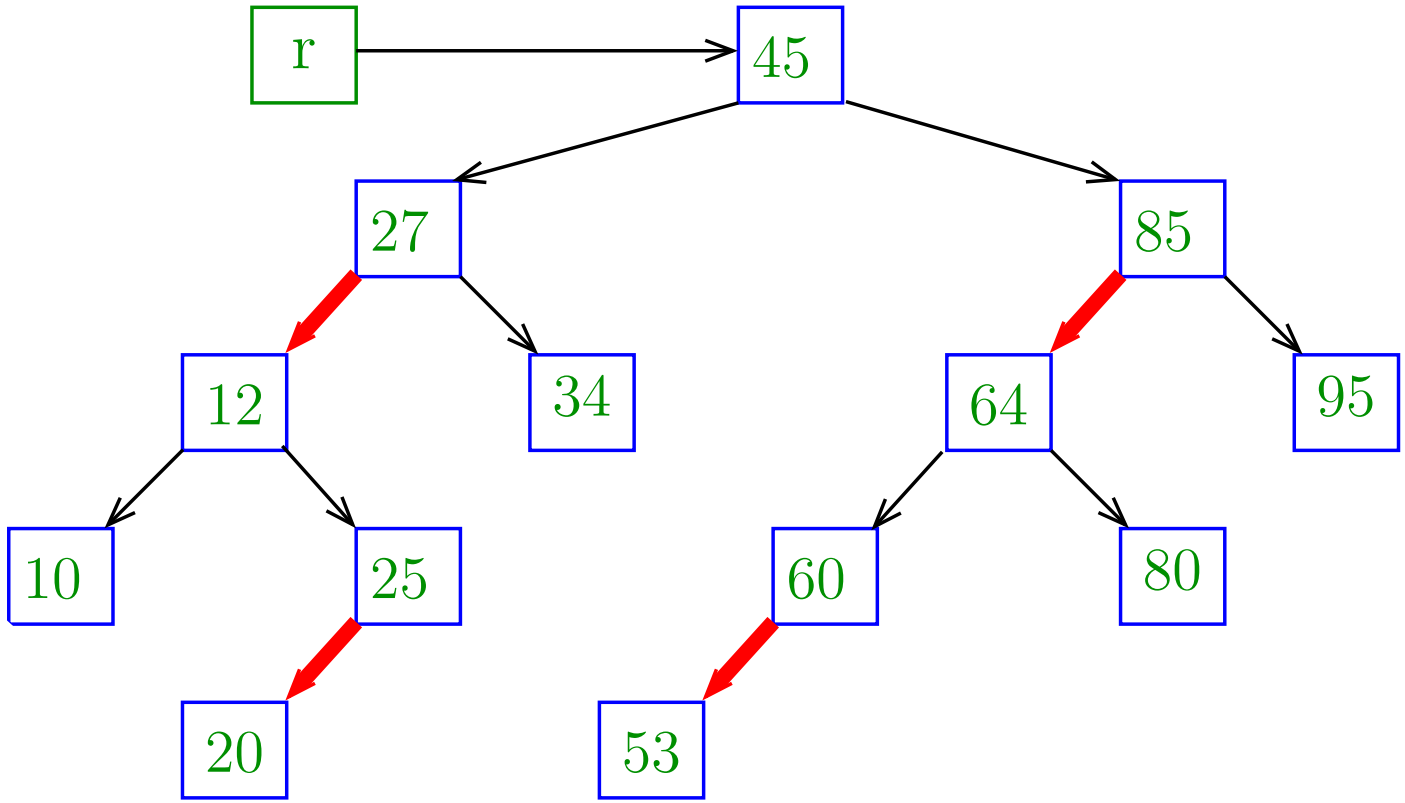



```
return balance(h);
```

rubro-negra: `deleteMin()`

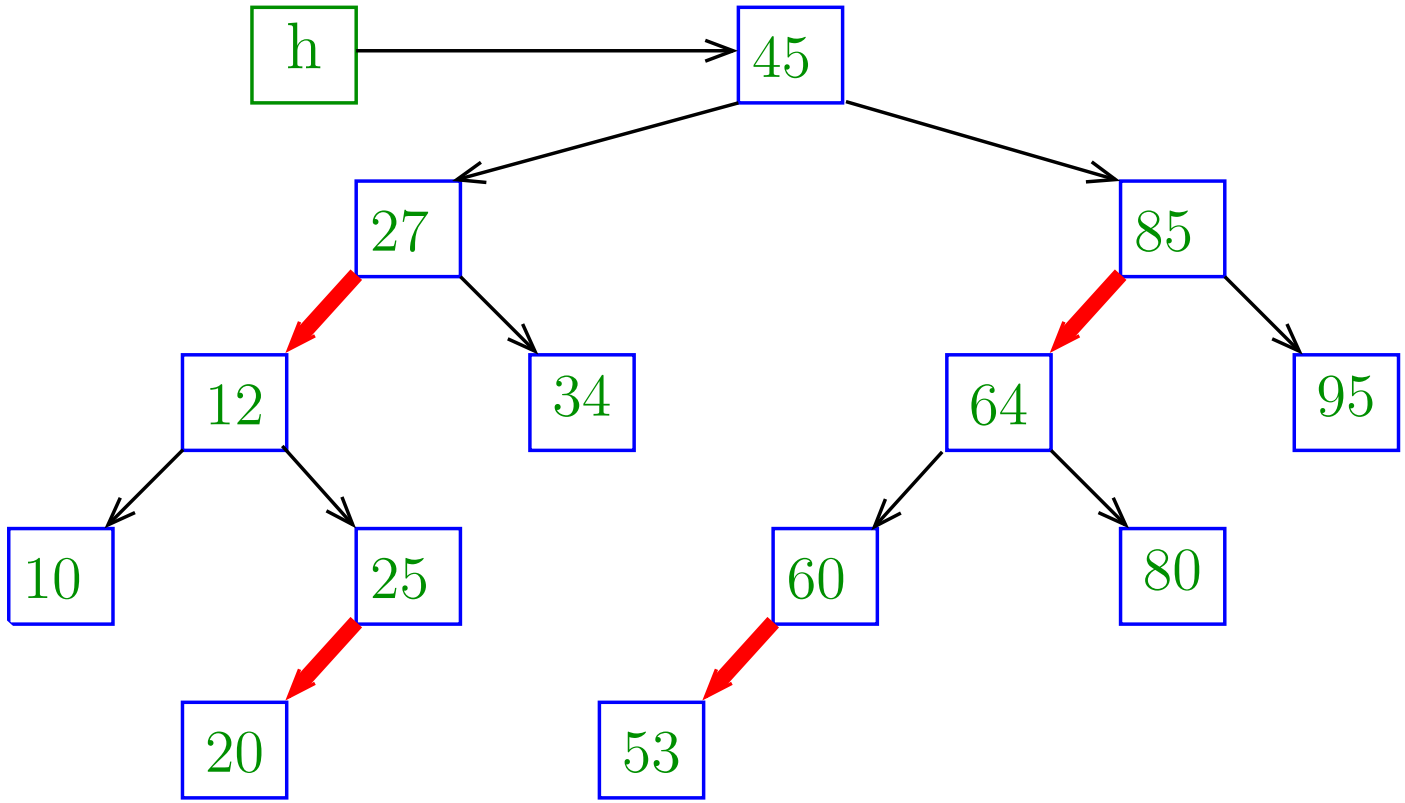


rubro-negra: outro `deleteMin()`



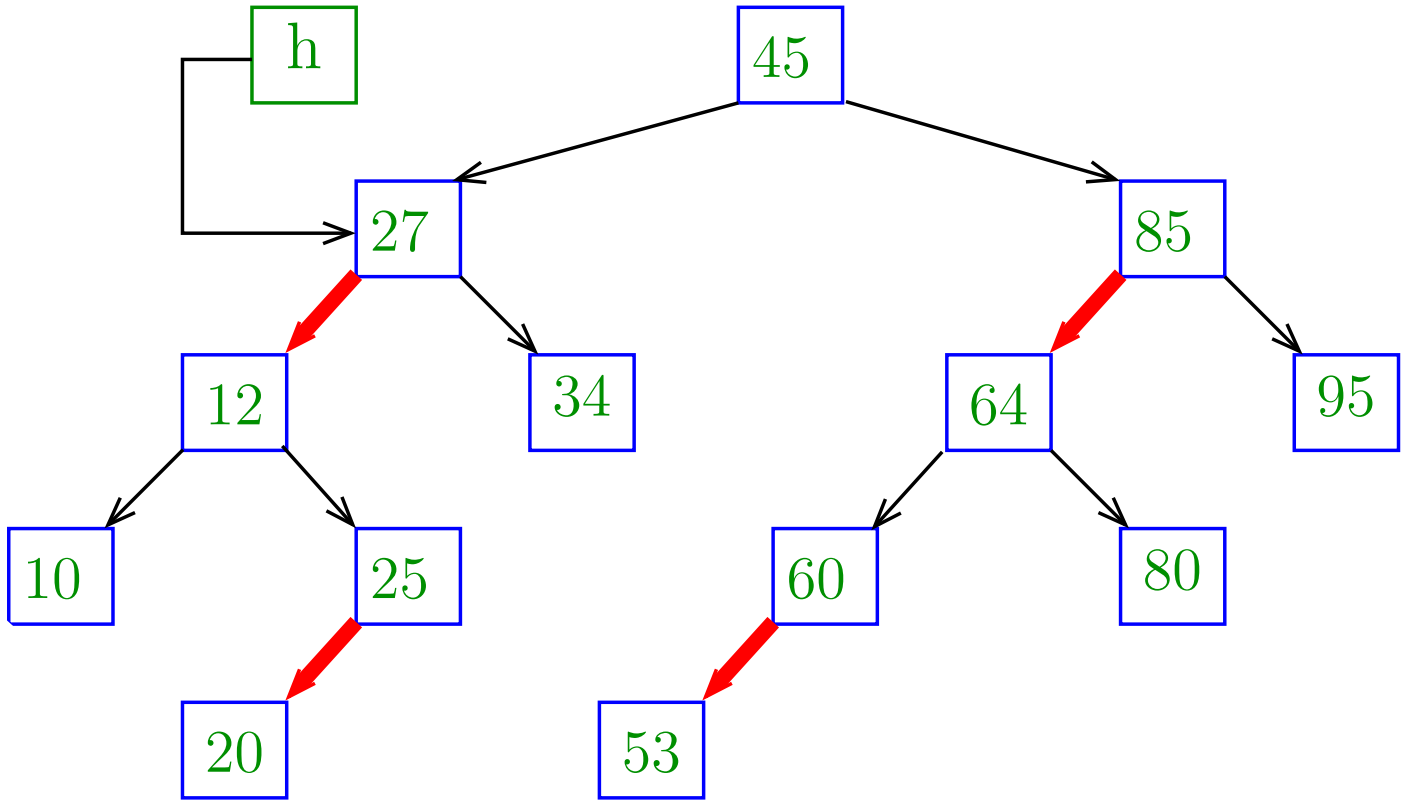
`deleteMin(r);`

rubro-negra: outro `deleteMin()`



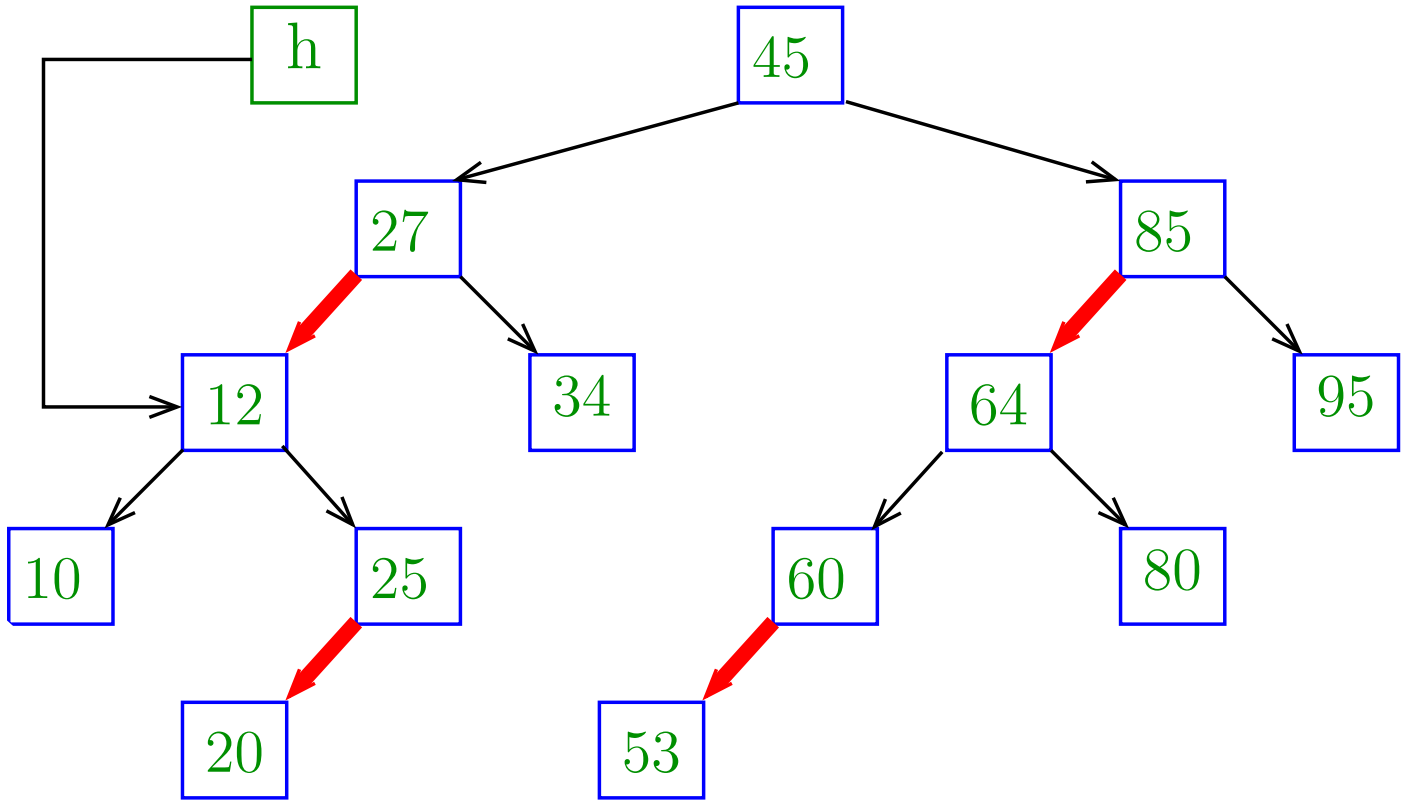

```
h.left = deleteMin(h.left);
```

rubro-negra: outro `deleteMin()`



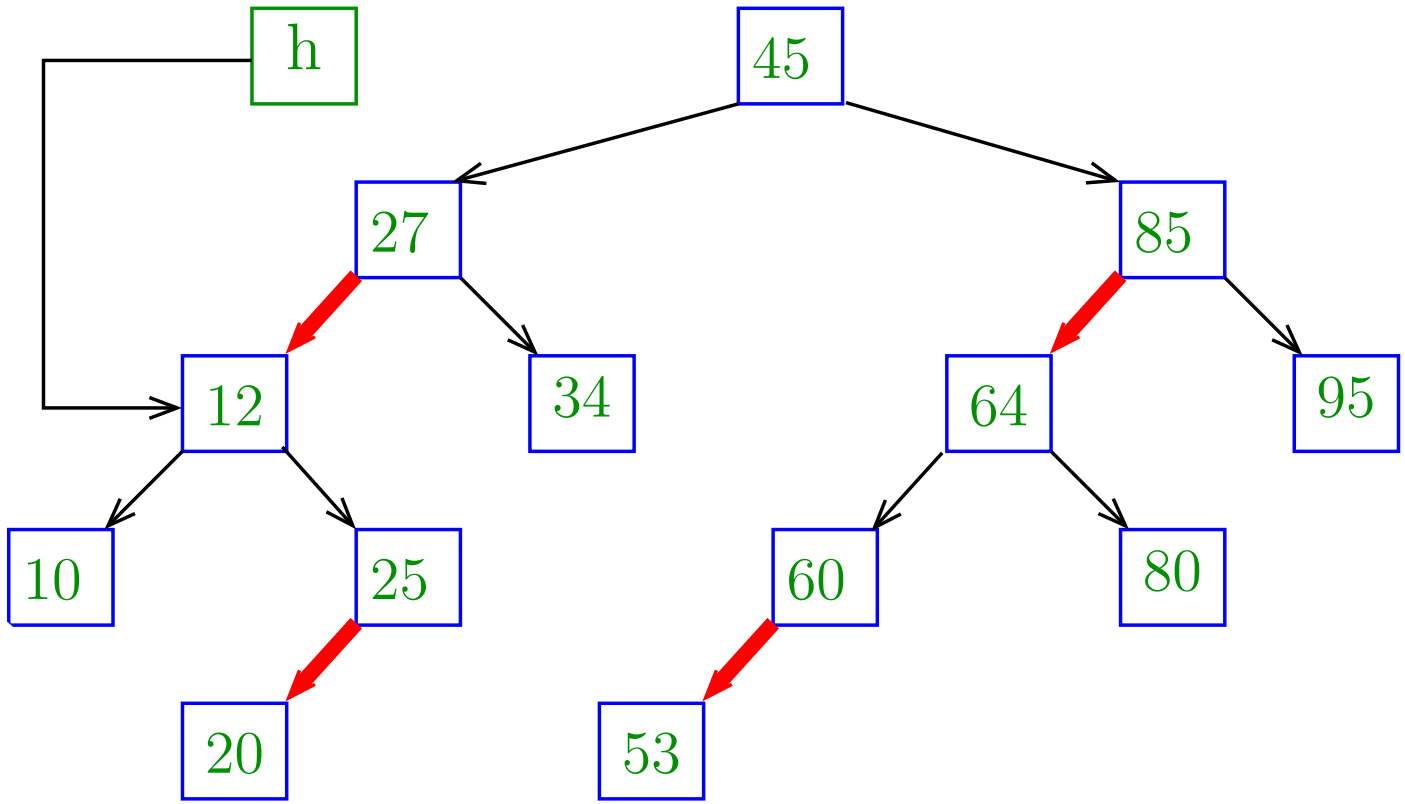

```
h.left = deleteMin(h.left);
```

rubro-negra: outro `deleteMin()`



`h = moveRedLeft(h);`

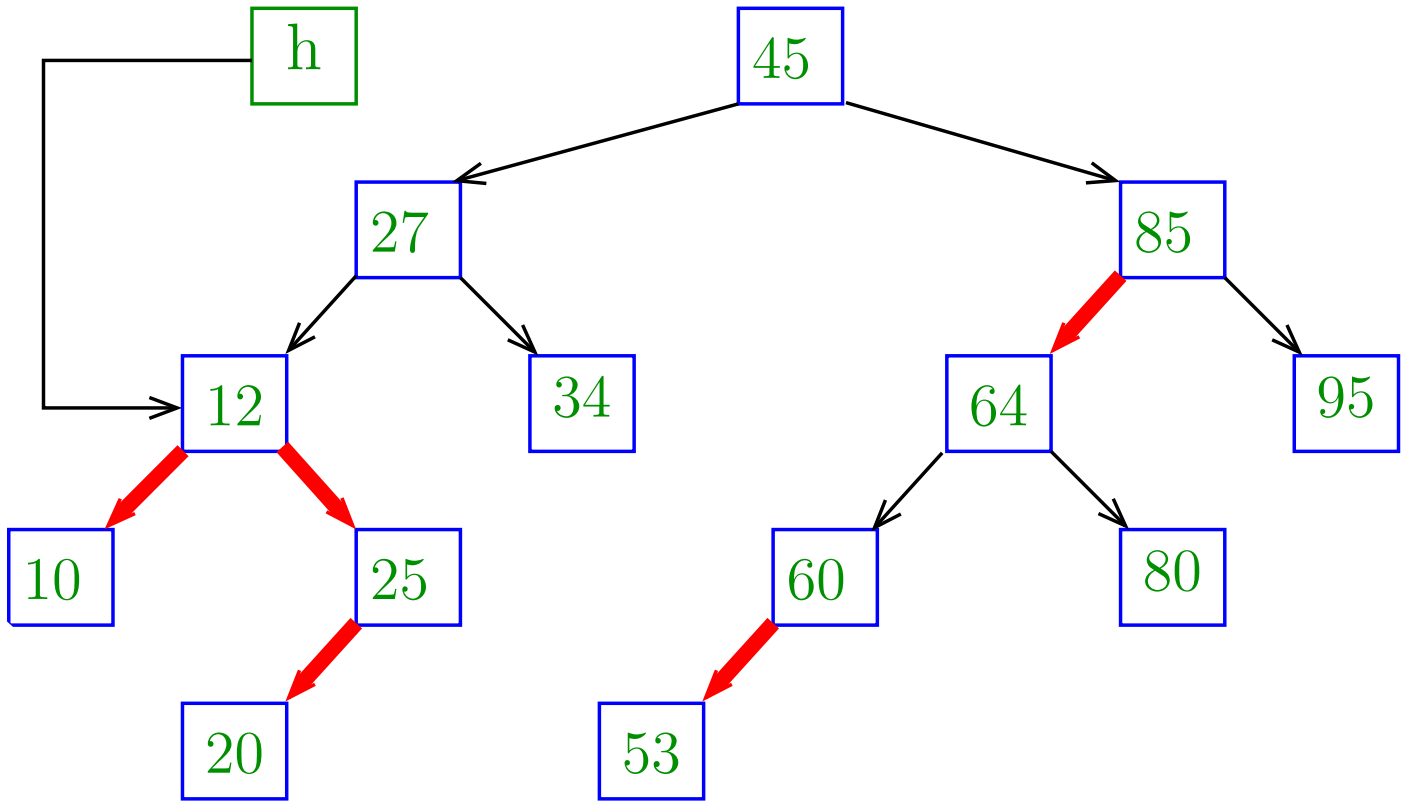
rubro-negra: outro `deleteMin()`



`flipColors(h);`

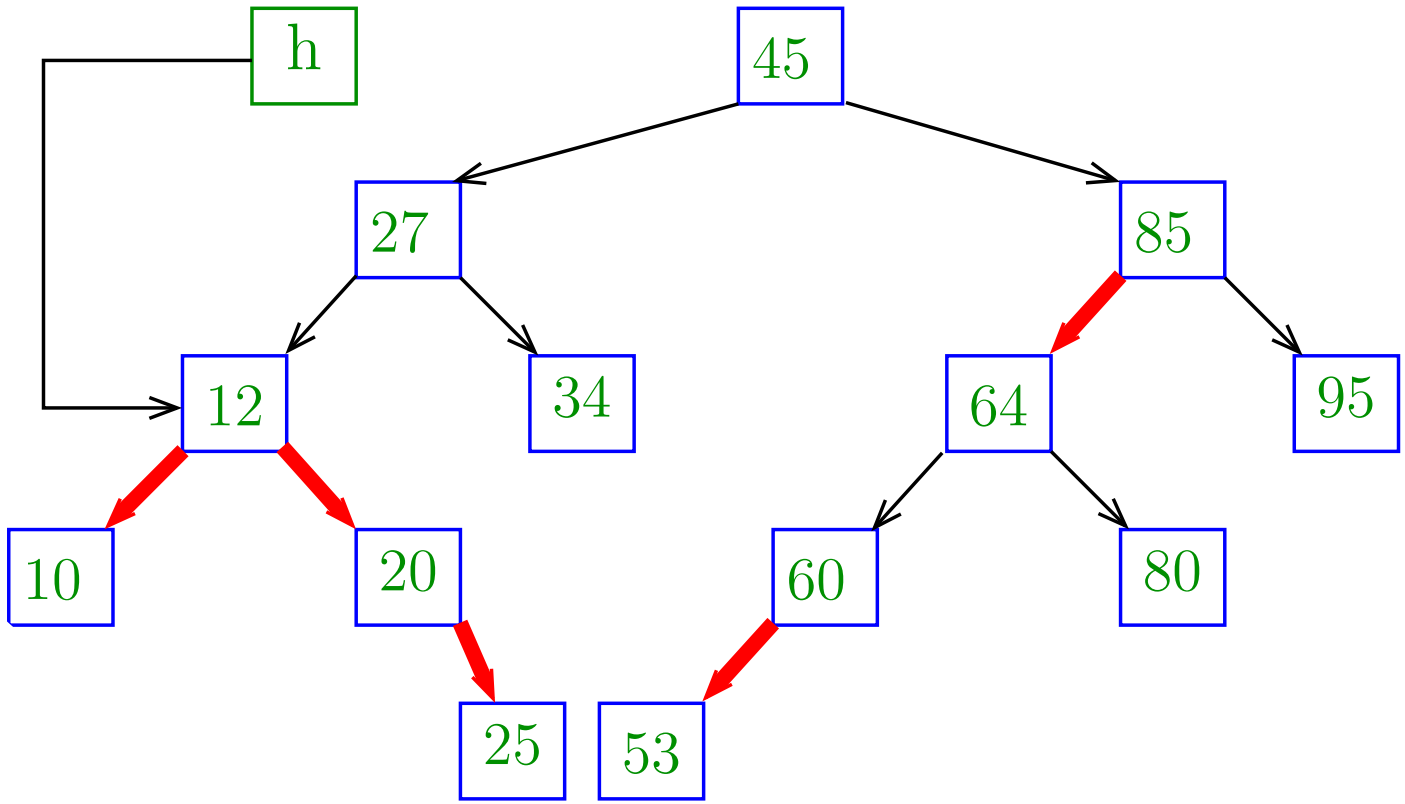
[`moveRedLeft(h);`]

rubro-negra: outro `deleteMin()`



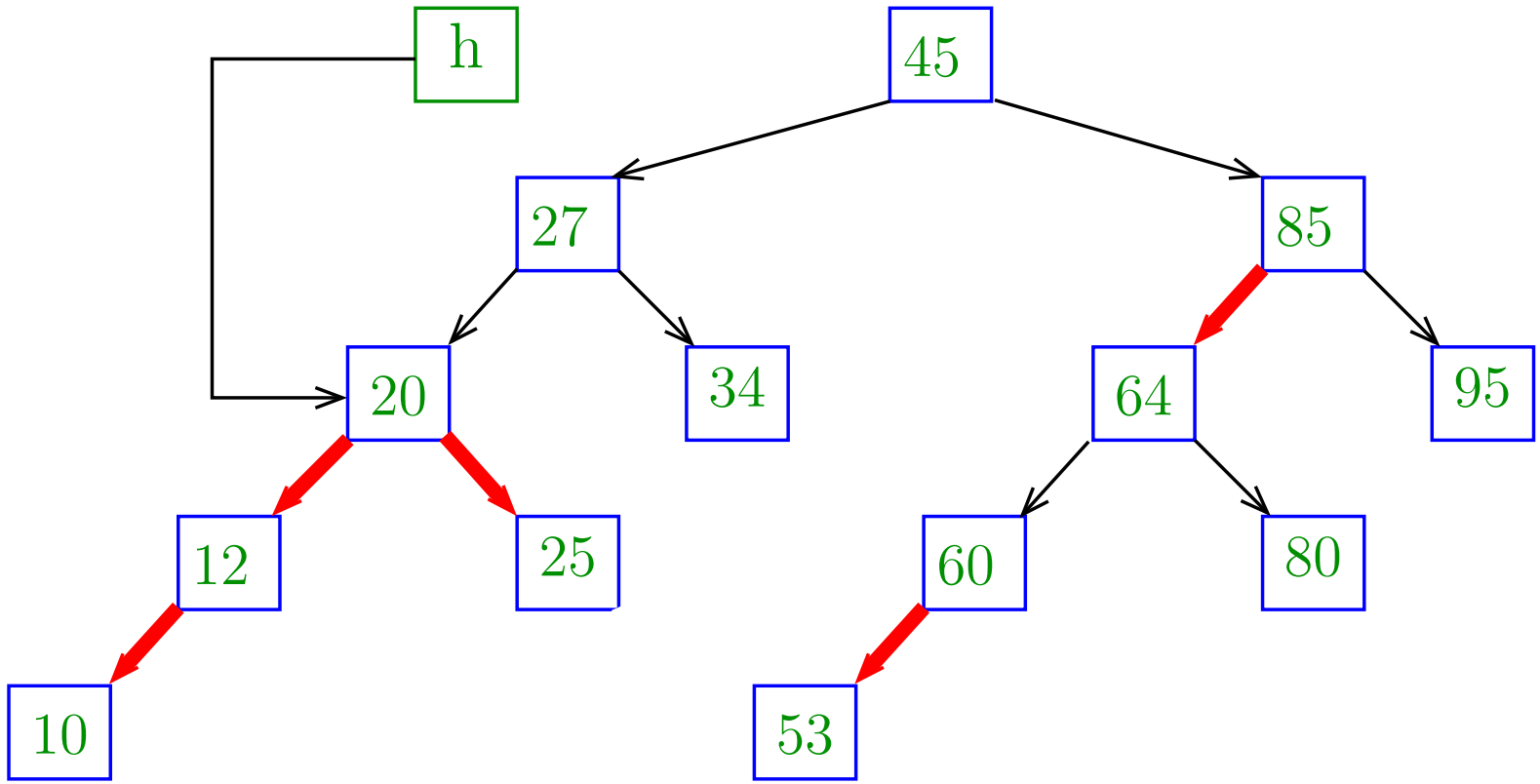
`h.right = rotateRight(h.right);` [`moveRedLeft(h);`]

rubro-negra: outro `deleteMin()`



`h = rotateLeft(h);` [`moveRedLeft(h);`]

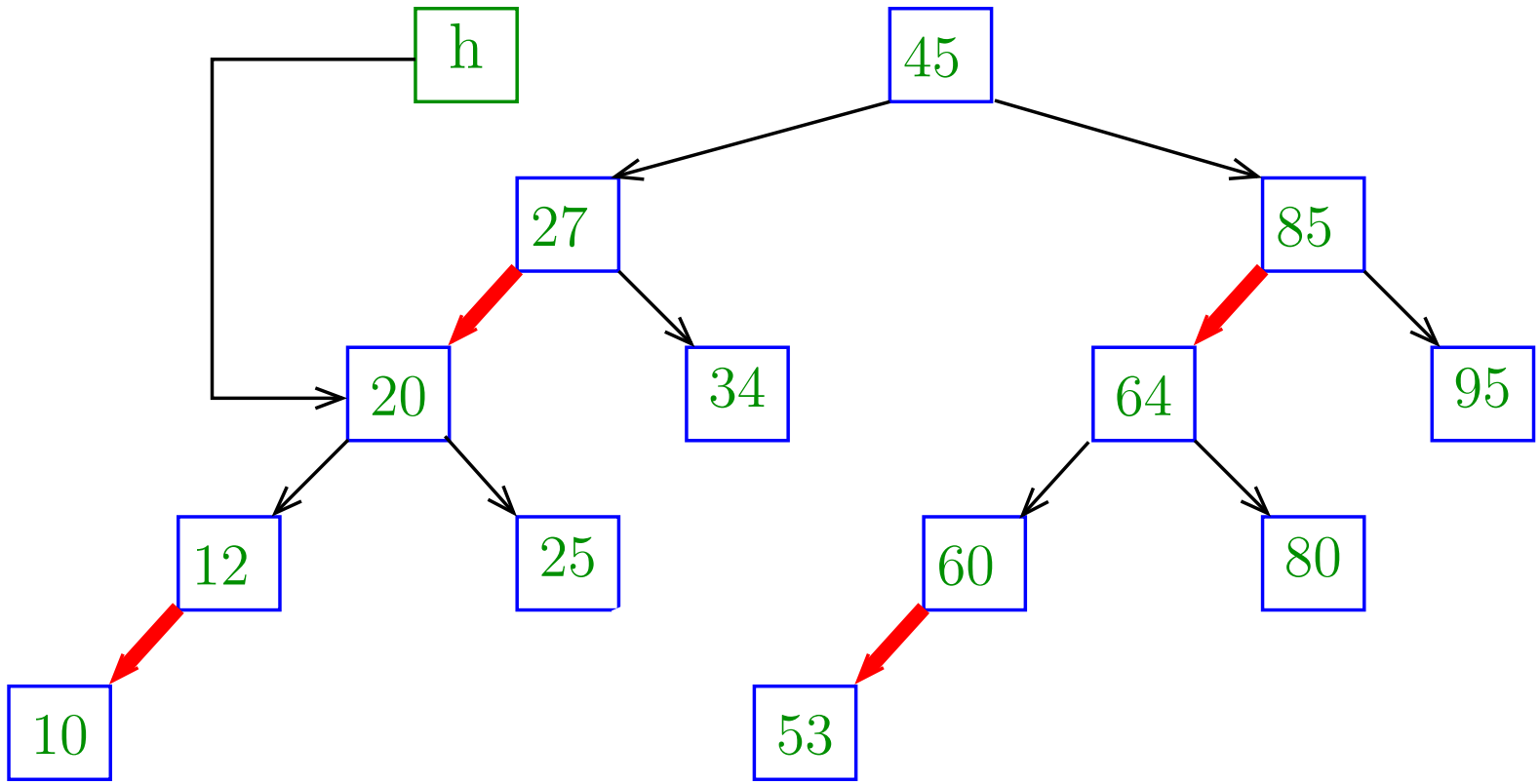
rubro-negra: outro `deleteMin()`



`flipColors(h);`

[`moveRedLeft(h);`]

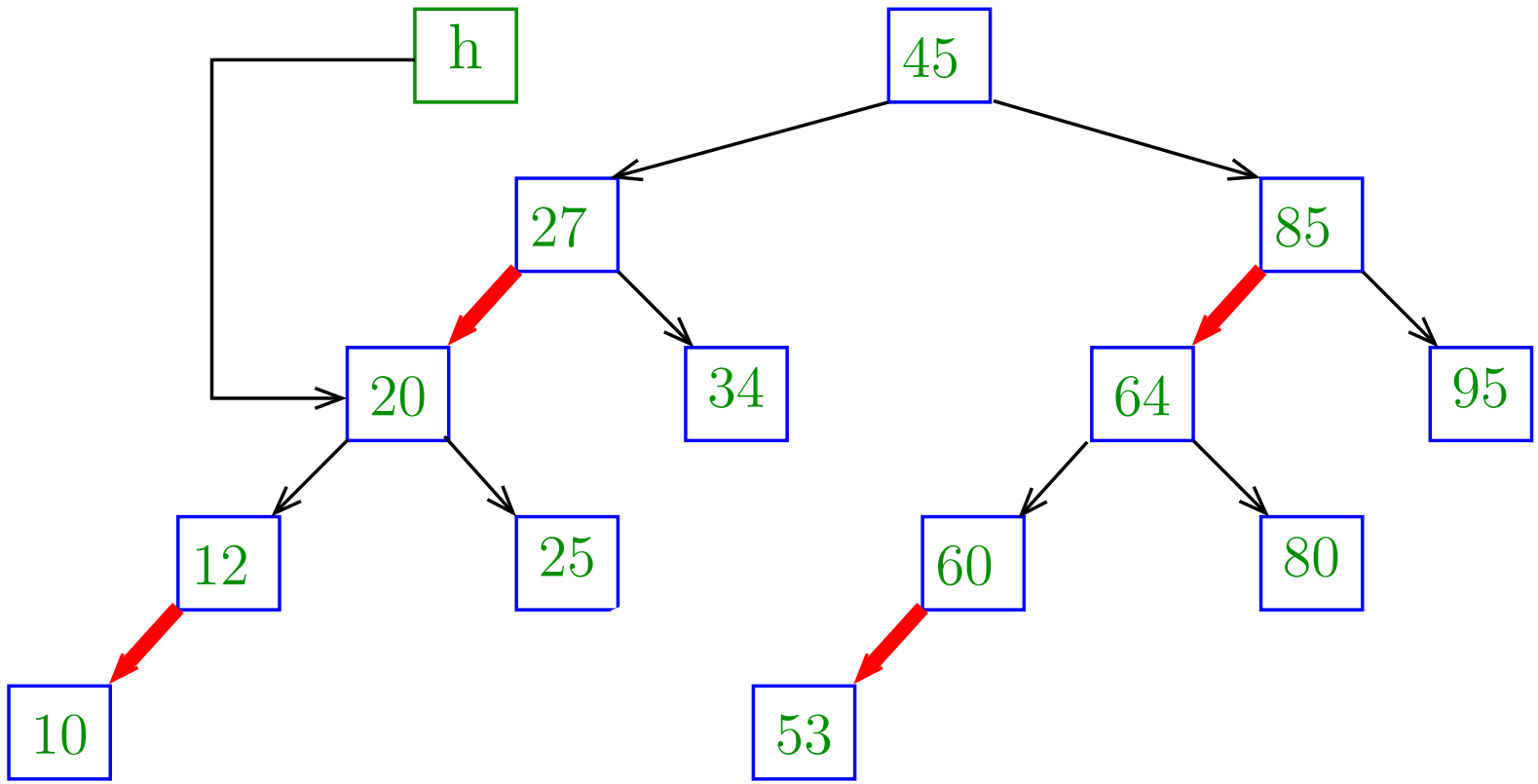
rubro-negra: outro `deleteMin()`



`return h;`

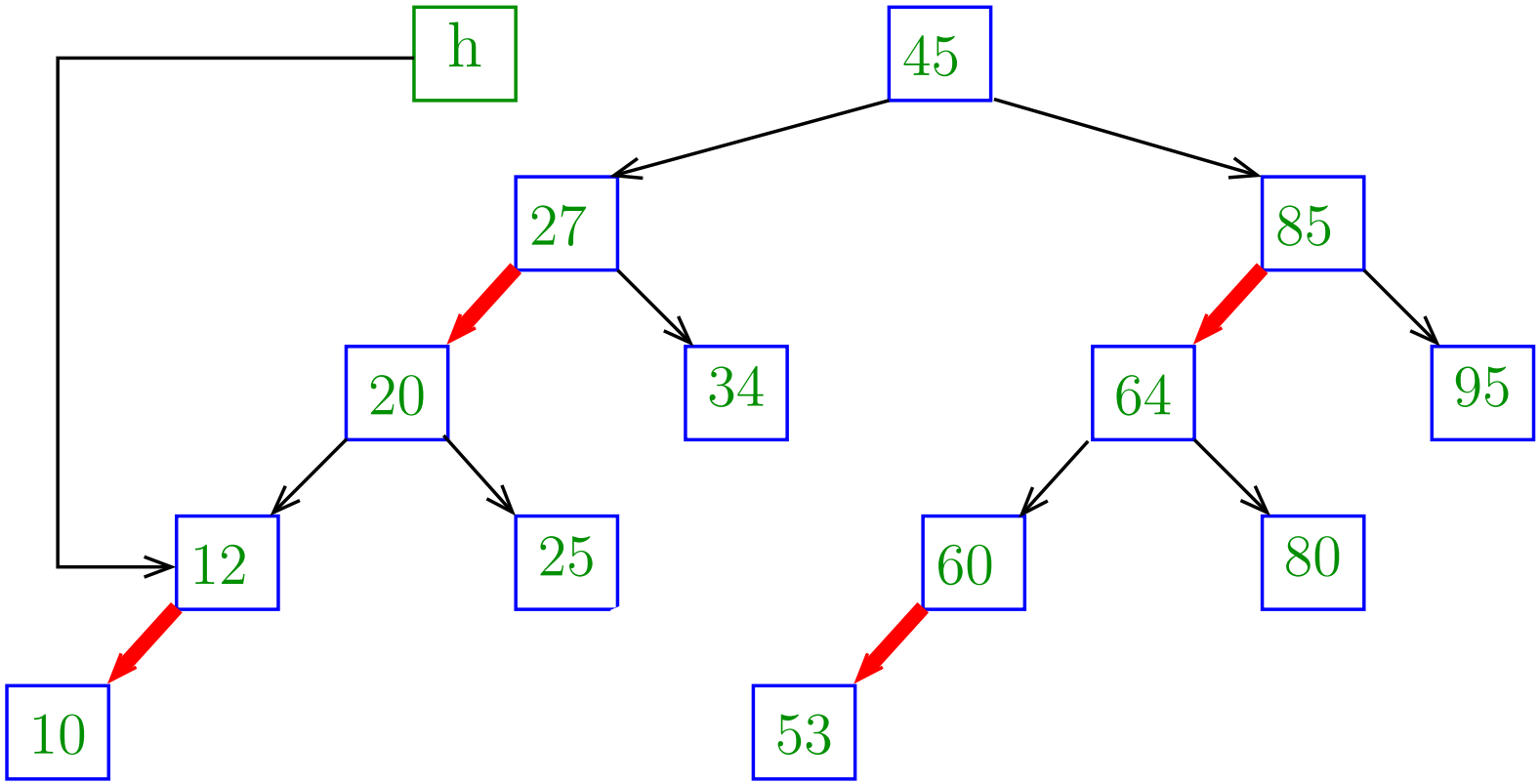
[`moveRedLeft(h);`]

rubro-negra: outro `deleteMin()`



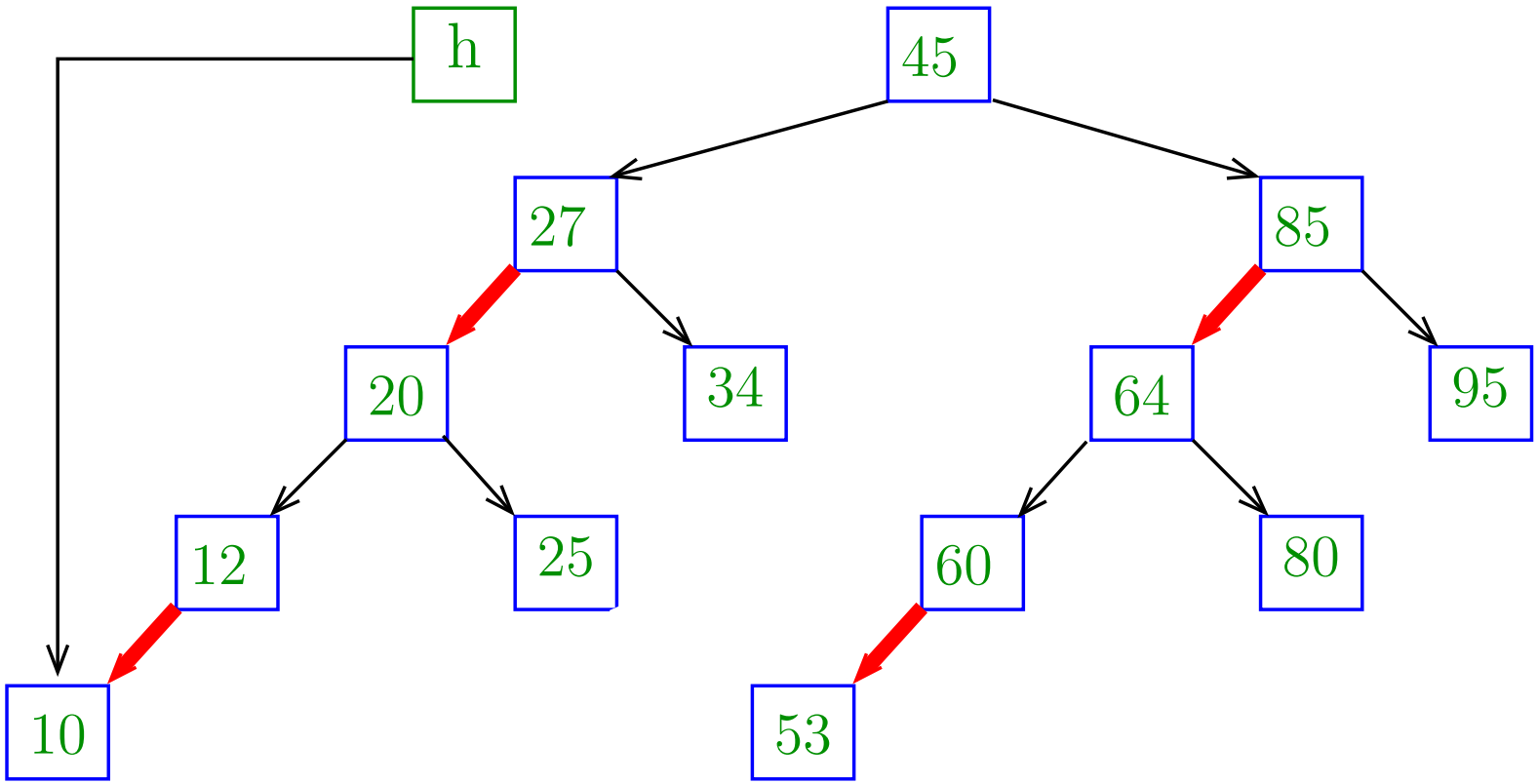

```
h.left = deleteMin(h.left);
```

rubro-negra: outro `deleteMin()`



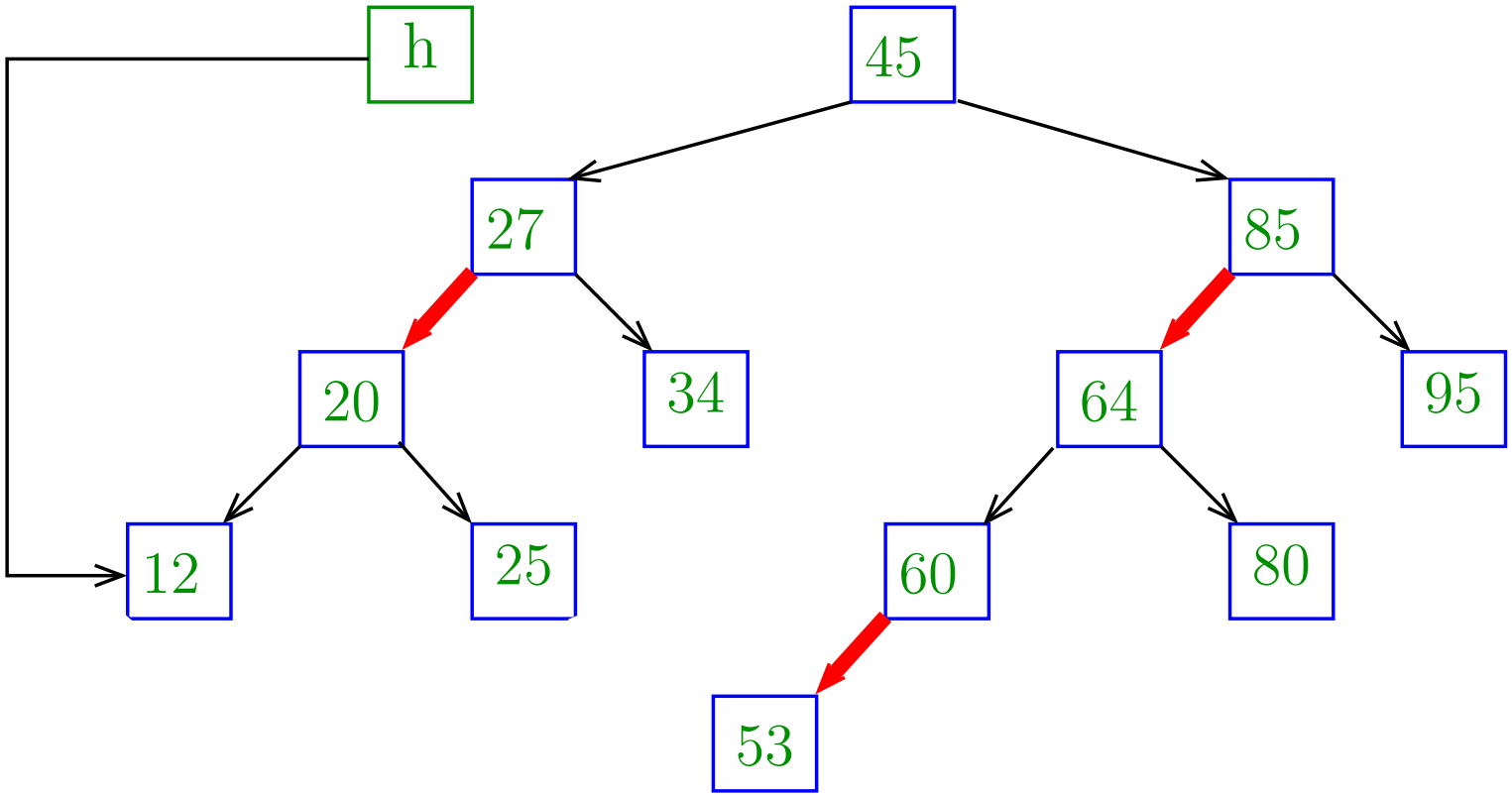

```
h.left = deleteMin(h.left);
```

rubro-negra: outro `deleteMin()`



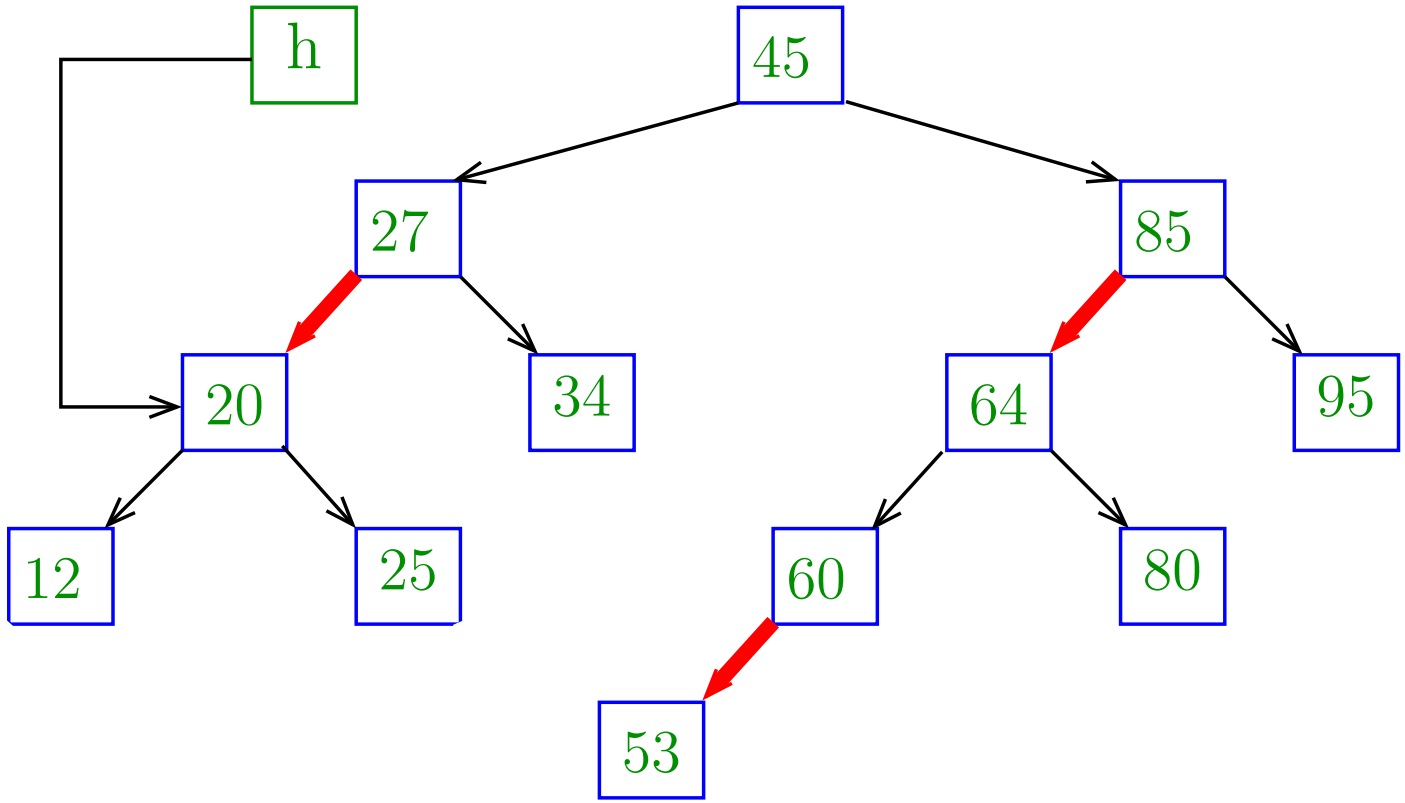

```
return null;
```

rubro-negra: outro `deleteMin()`



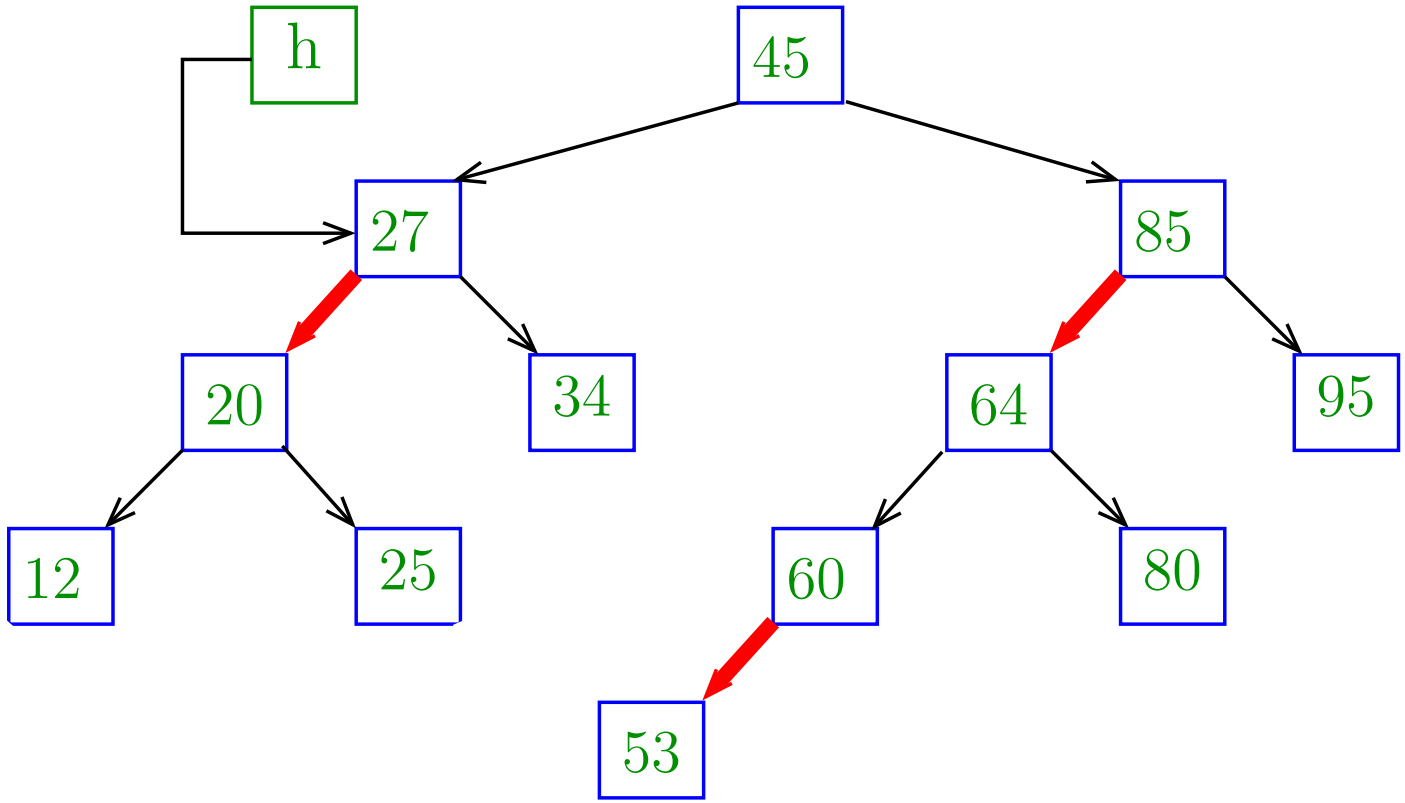

```
return balance(h);
```

rubro-negra: outro `deleteMin()`



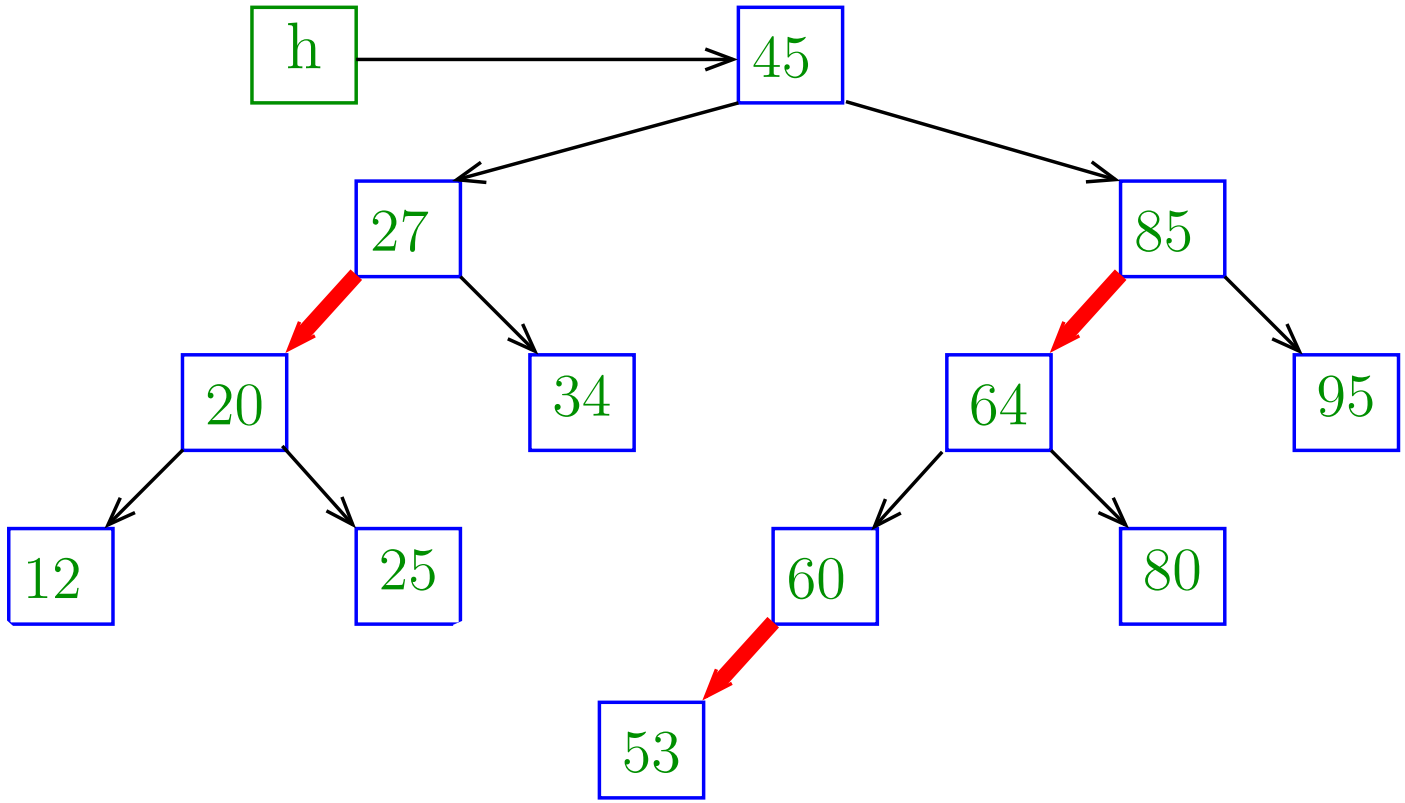

```
return balance(h);
```

rubro-negra: outro `deleteMin()`



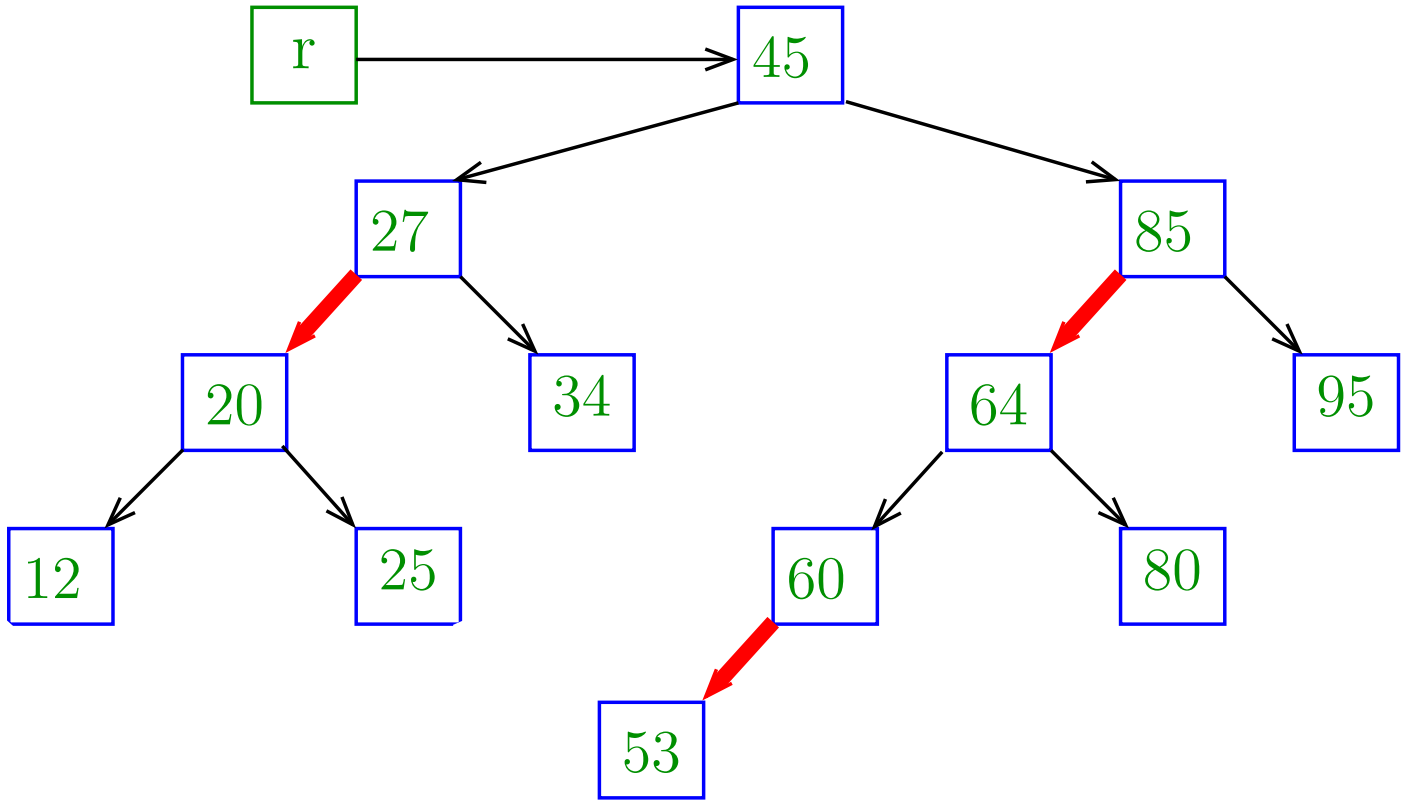

```
return balance(h);
```

rubro-negra: outro `deleteMin()`

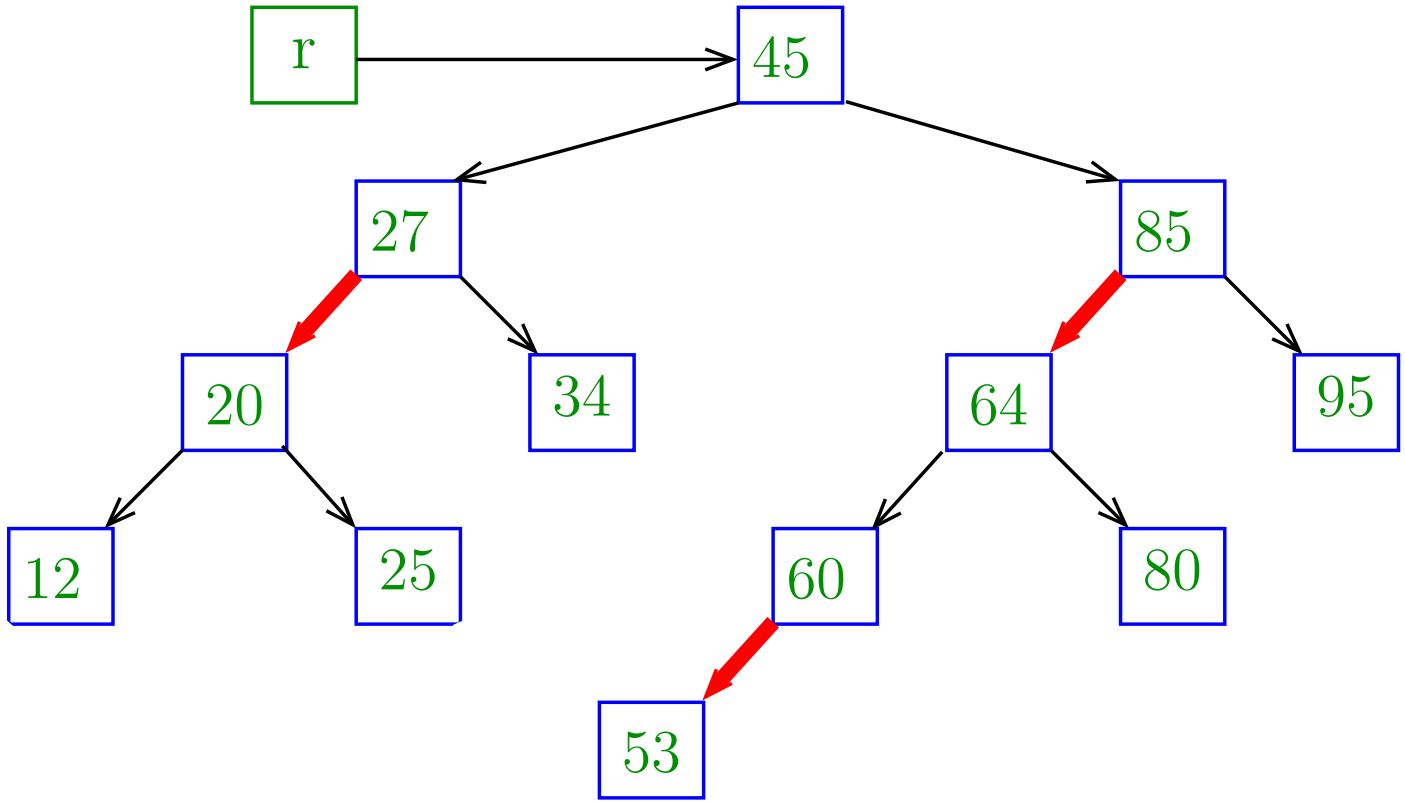



```
return balance(h);
```

rubro-negra: outro `deleteMin()`

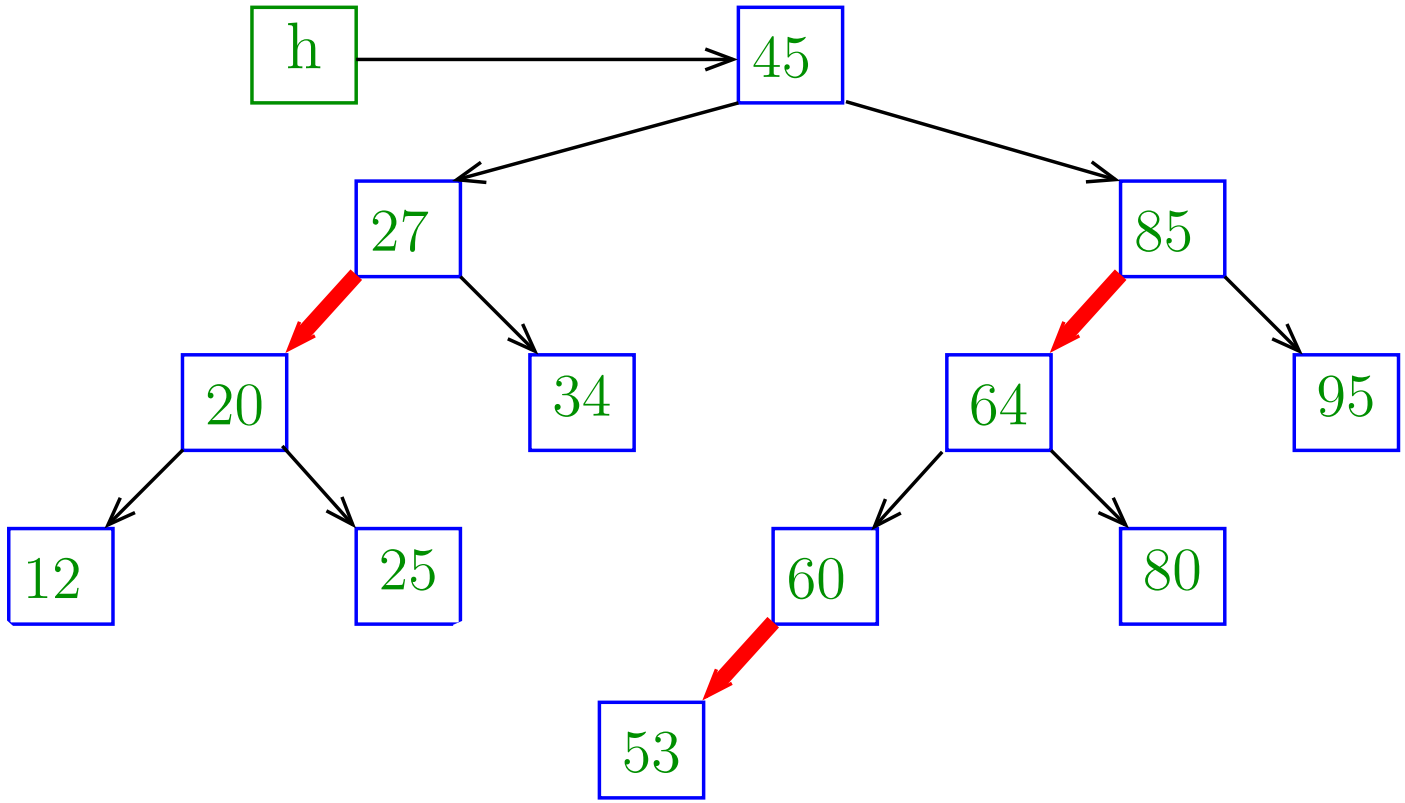


rubro-negra: mais outro `deleteMin()`



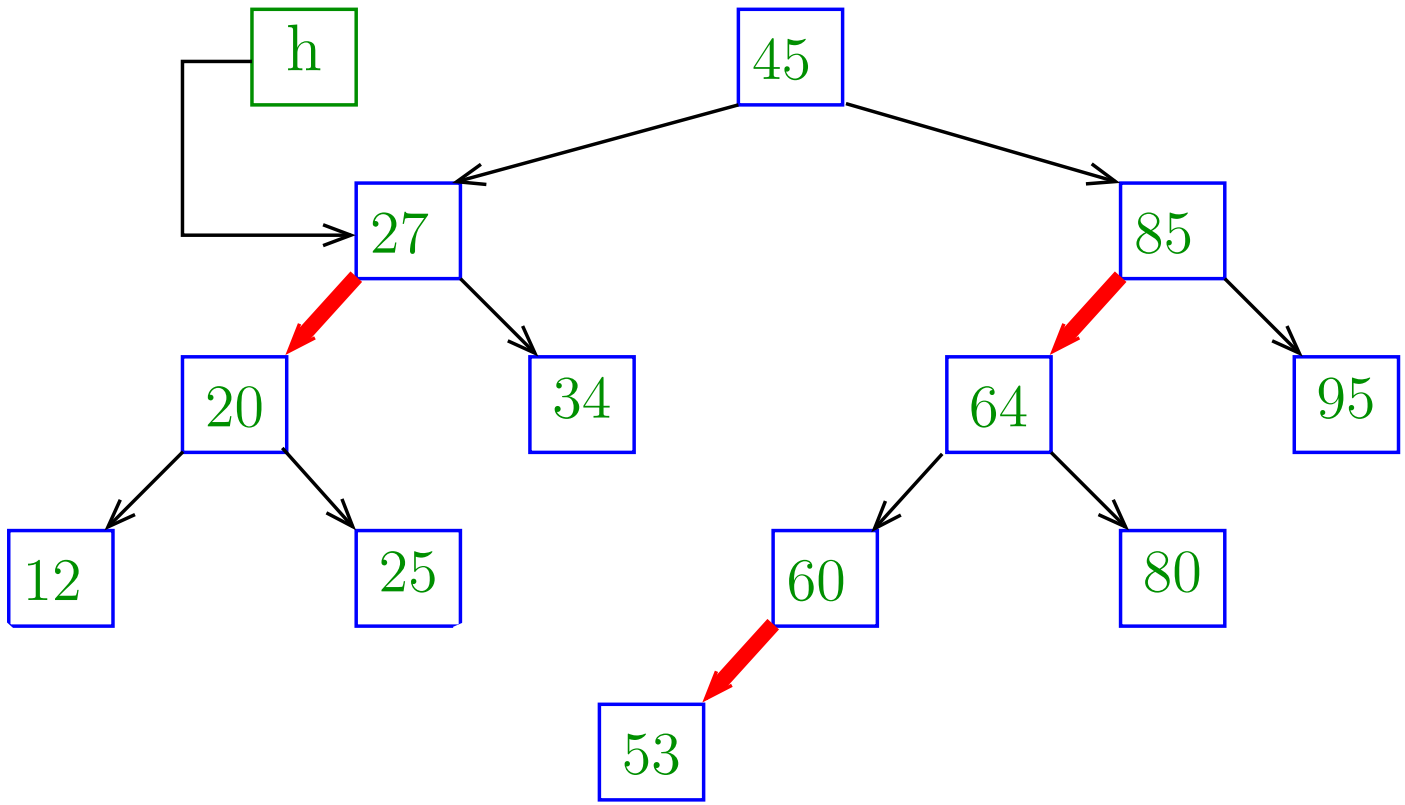
`deleteMin(r);`

rubro-negra: mais outro `deleteMin()`



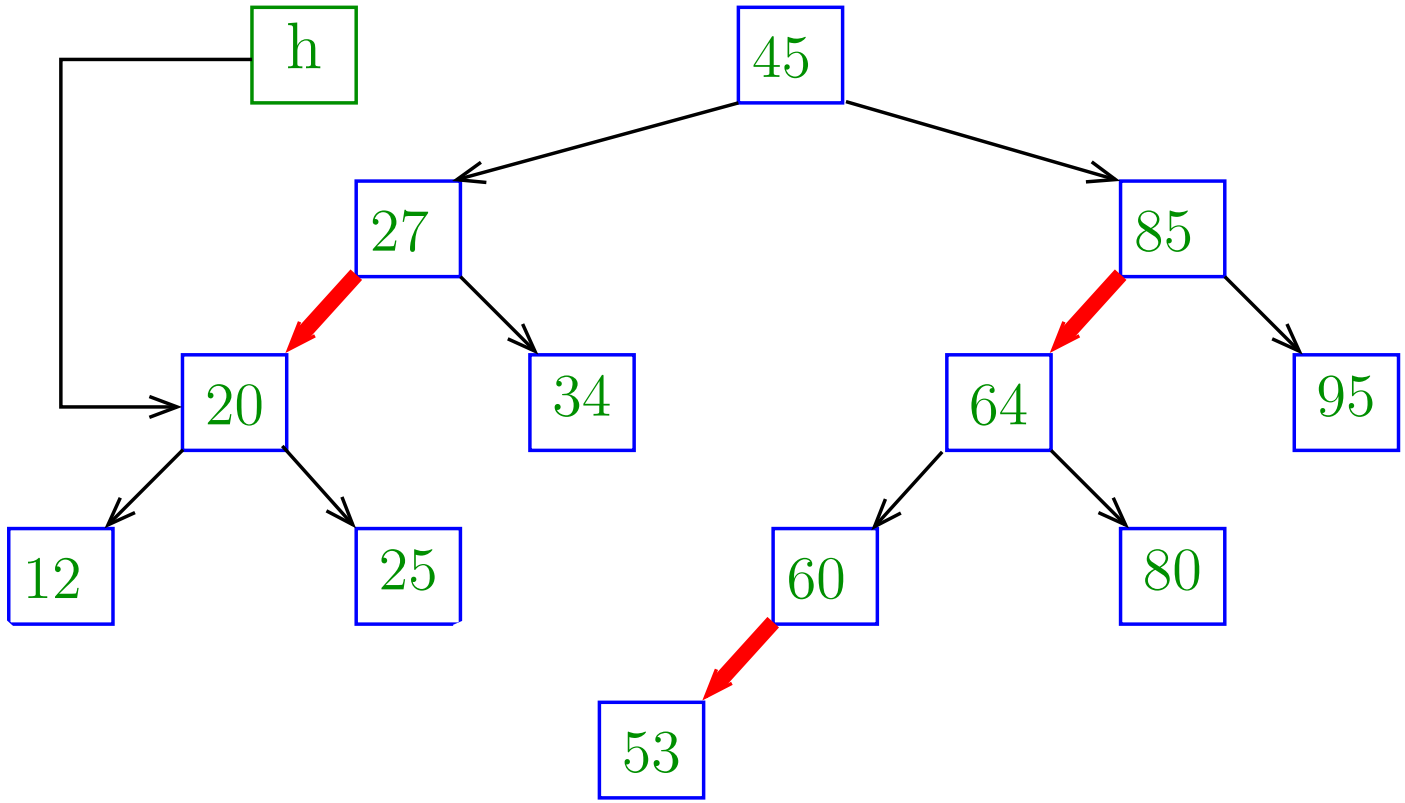

```
h.left = deleteMin(h.left);
```

rubro-negra: mais outro `deleteMin()`



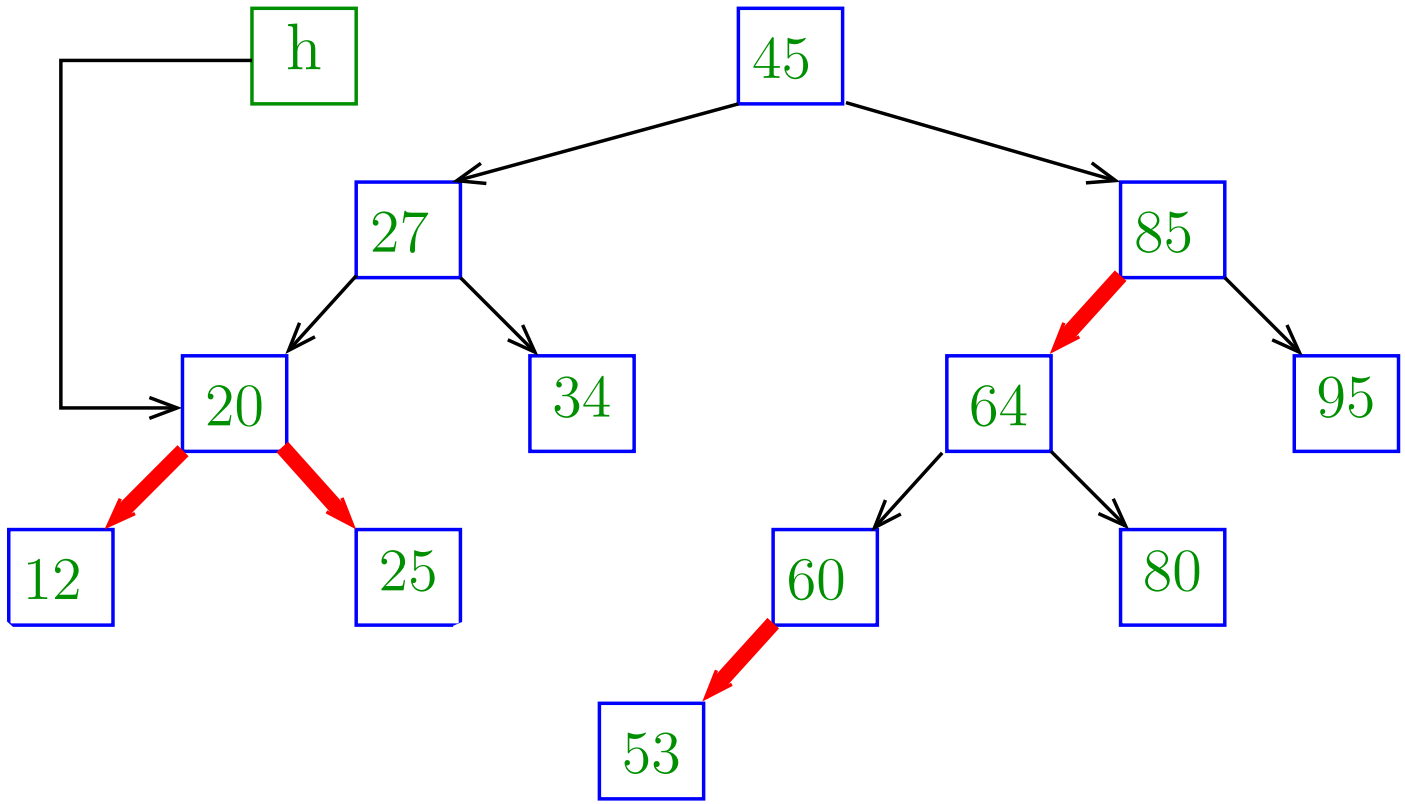

```
h.left = deleteMin(h.left);
```

rubro-negra: mais outro `deleteMin()`



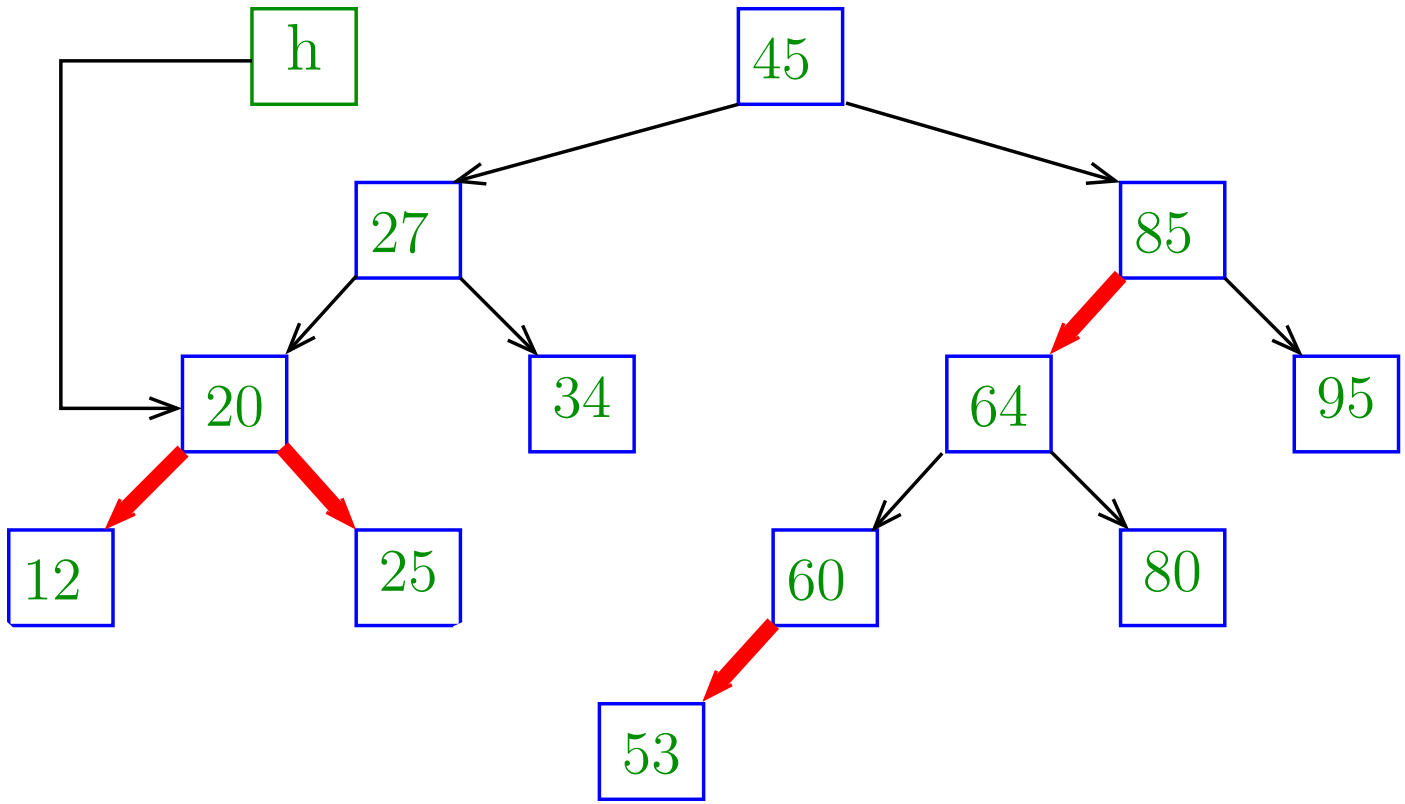
`h = moveRedLeft(h);`

rubro-negra: mais outro `deleteMin()`



`flipColors(h);` [`moveRedLeft(h);`]

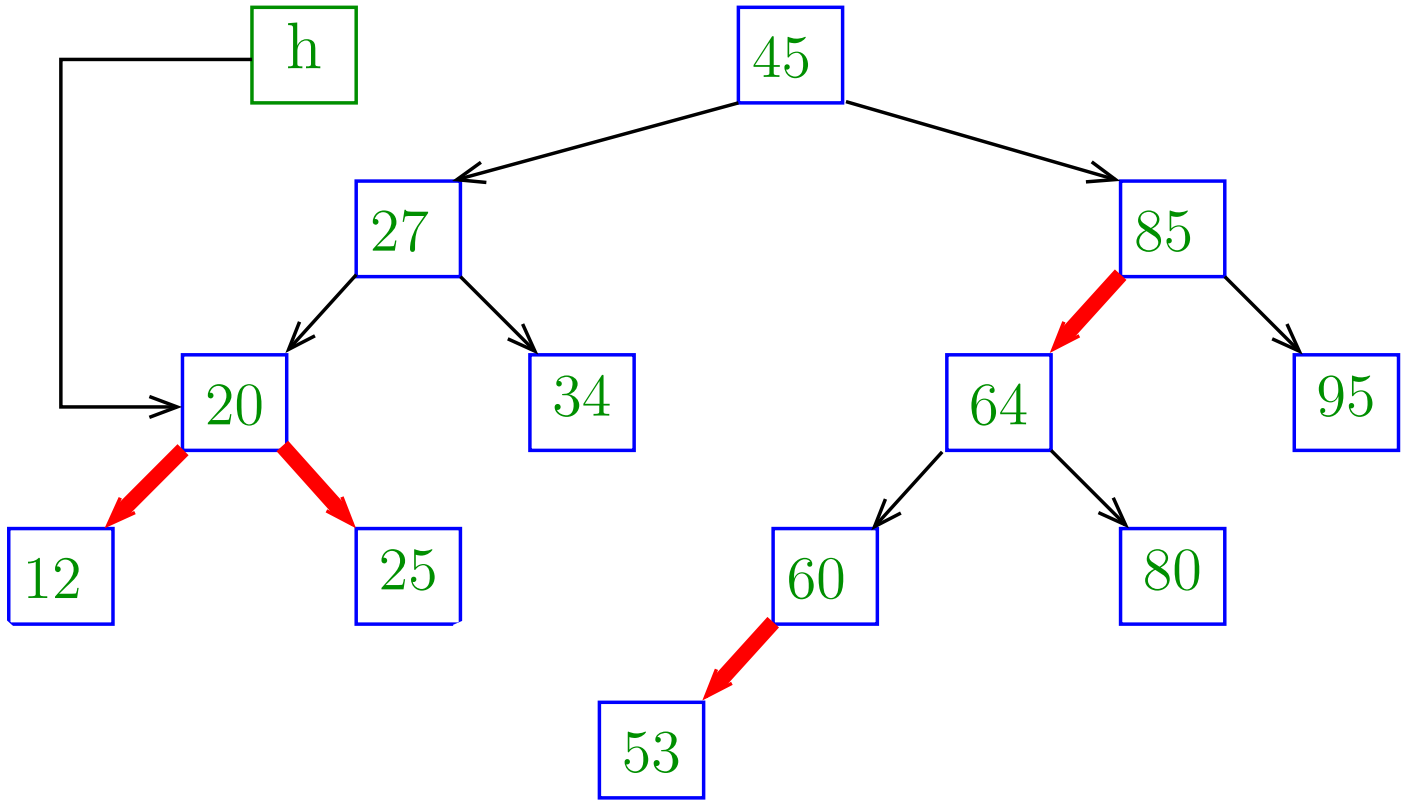
rubro-negra: mais outro `deleteMin()`



`return h`

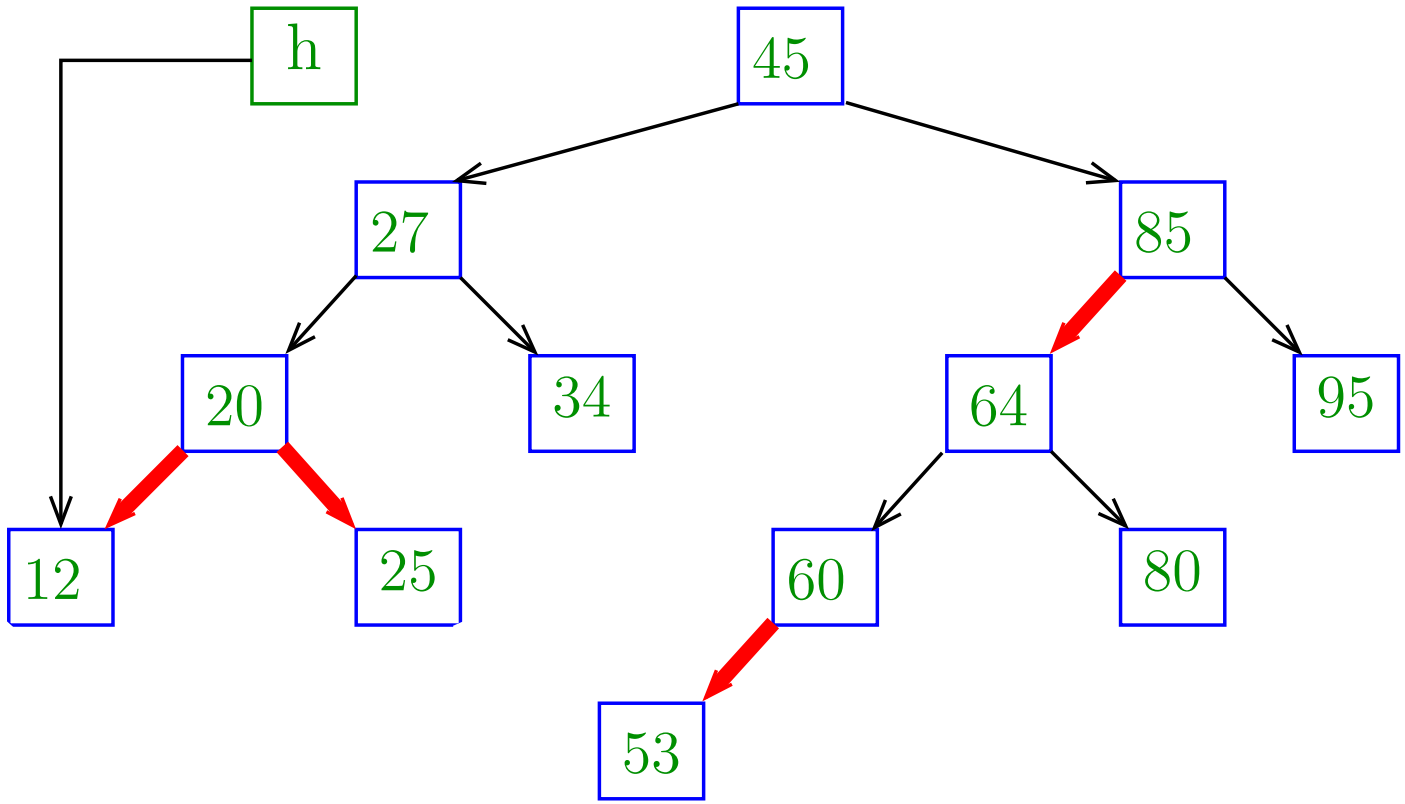
[`moveRedLeft(h);`]

rubro-negra: mais outro `deleteMin()`



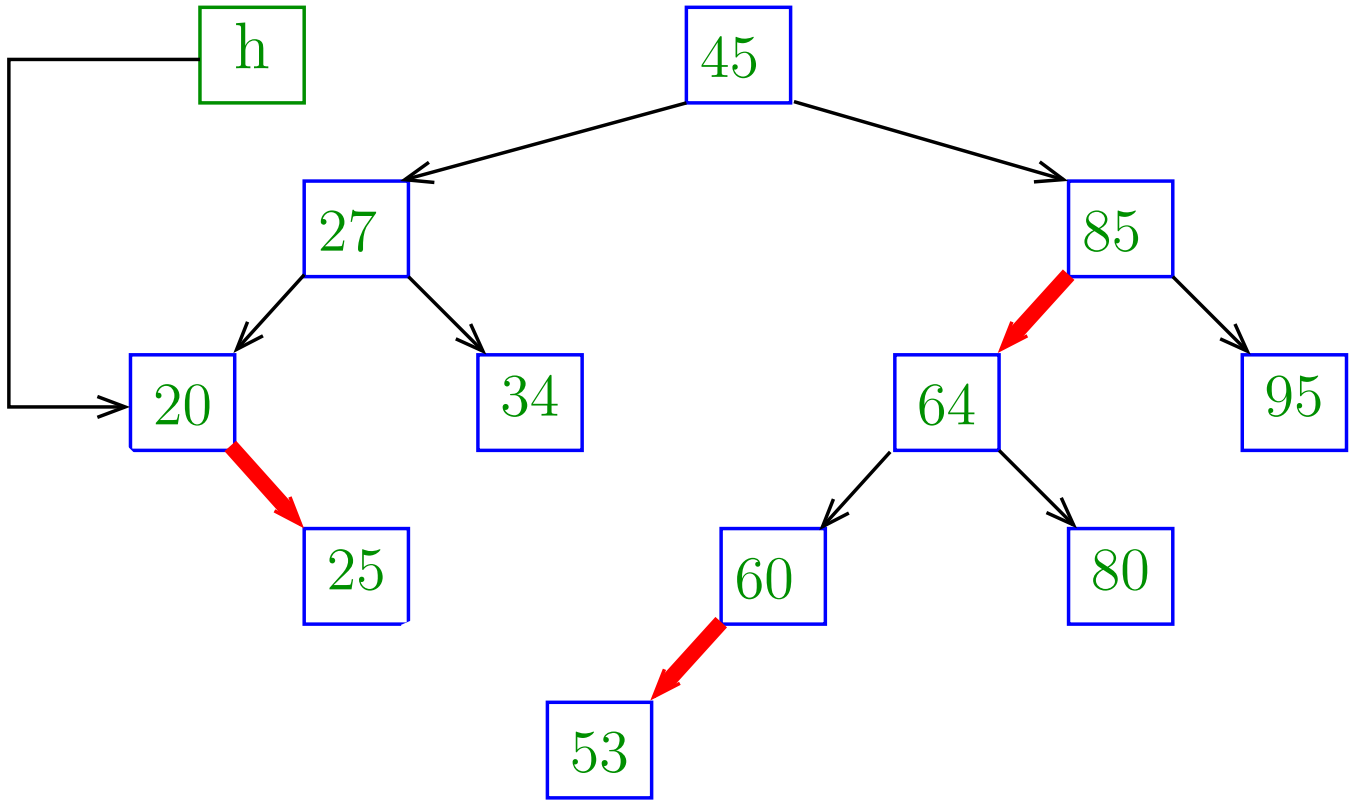

```
h.left = deleteMin(h.left);
```

rubro-negra: mais outro `deleteMin()`



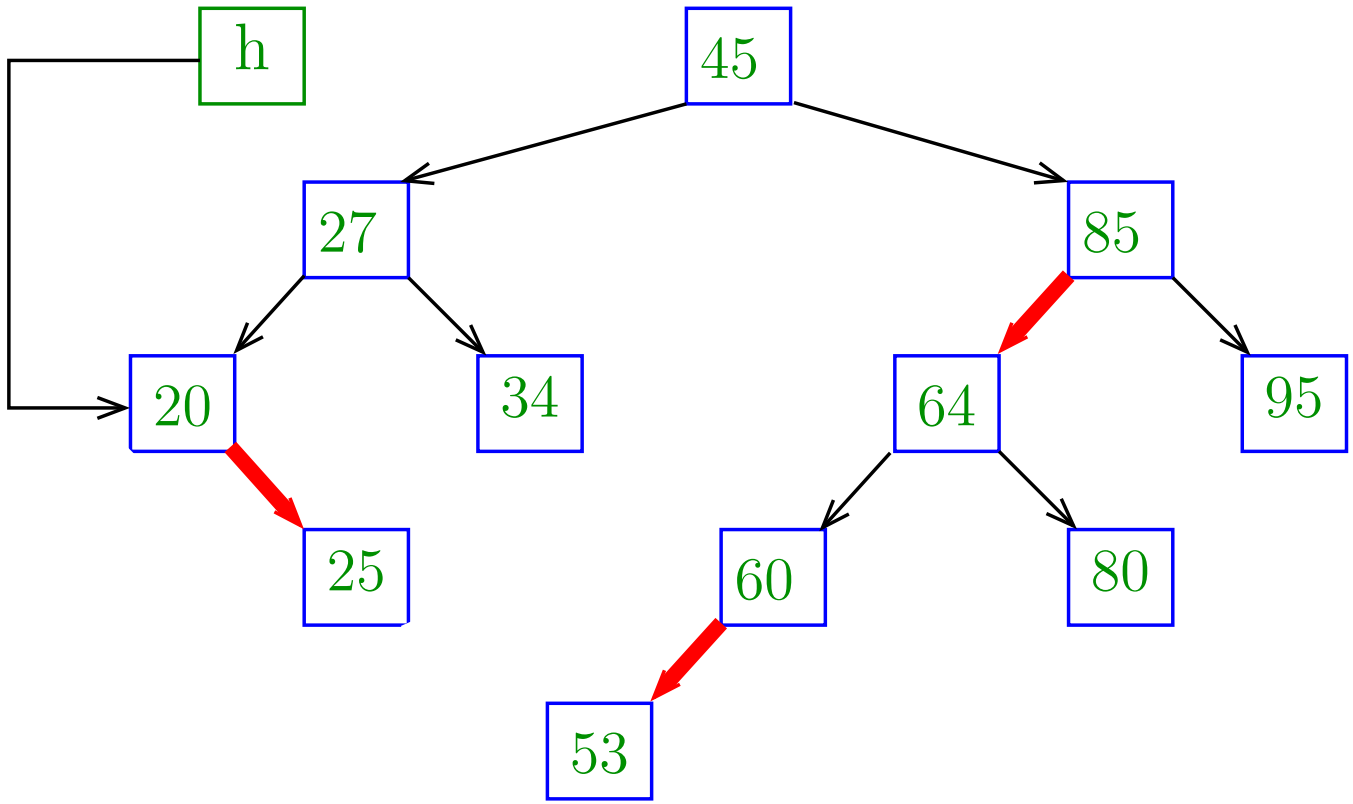
`return null`

rubro-negra: mais outro `deleteMin()`



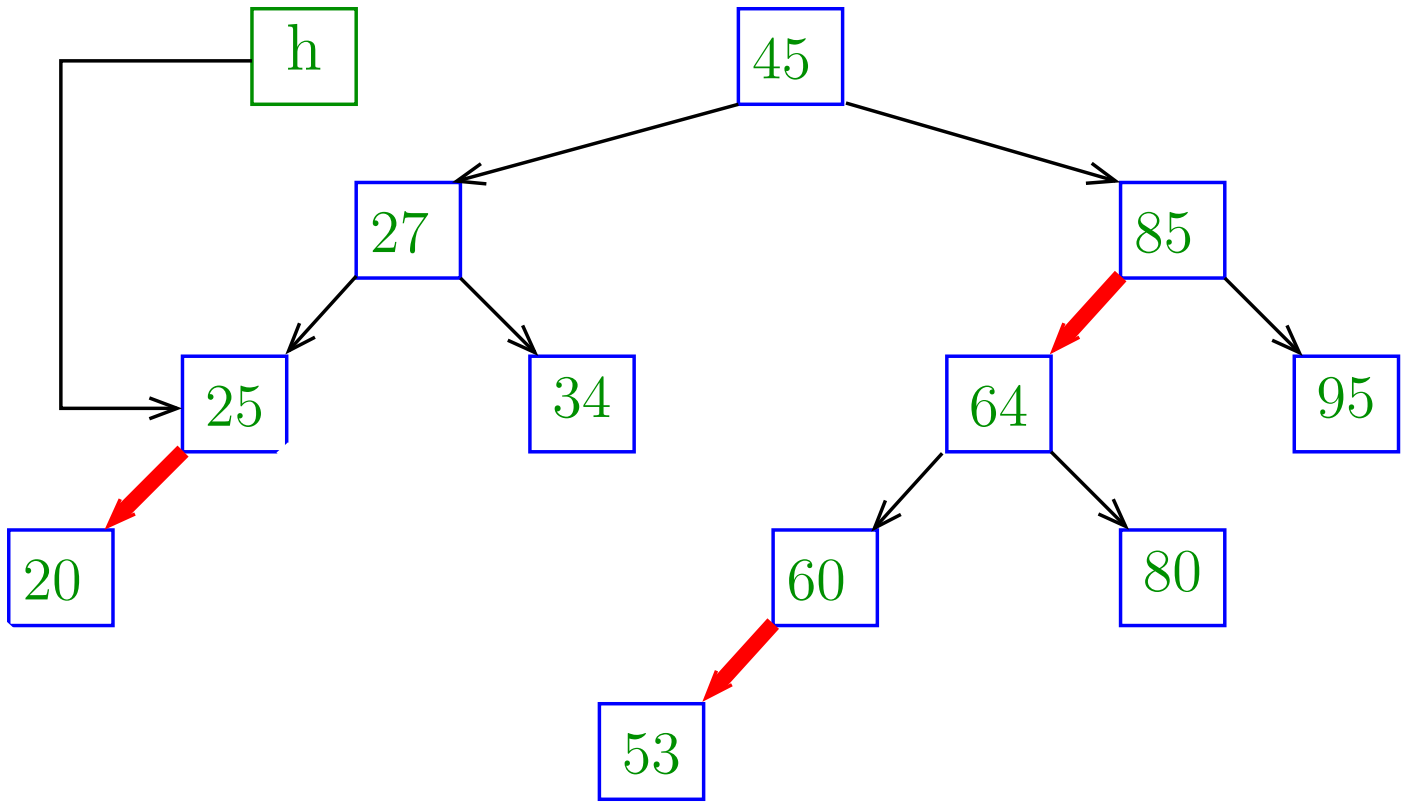

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()`



`h = rotateLeft(h);` [`balance(h);`]

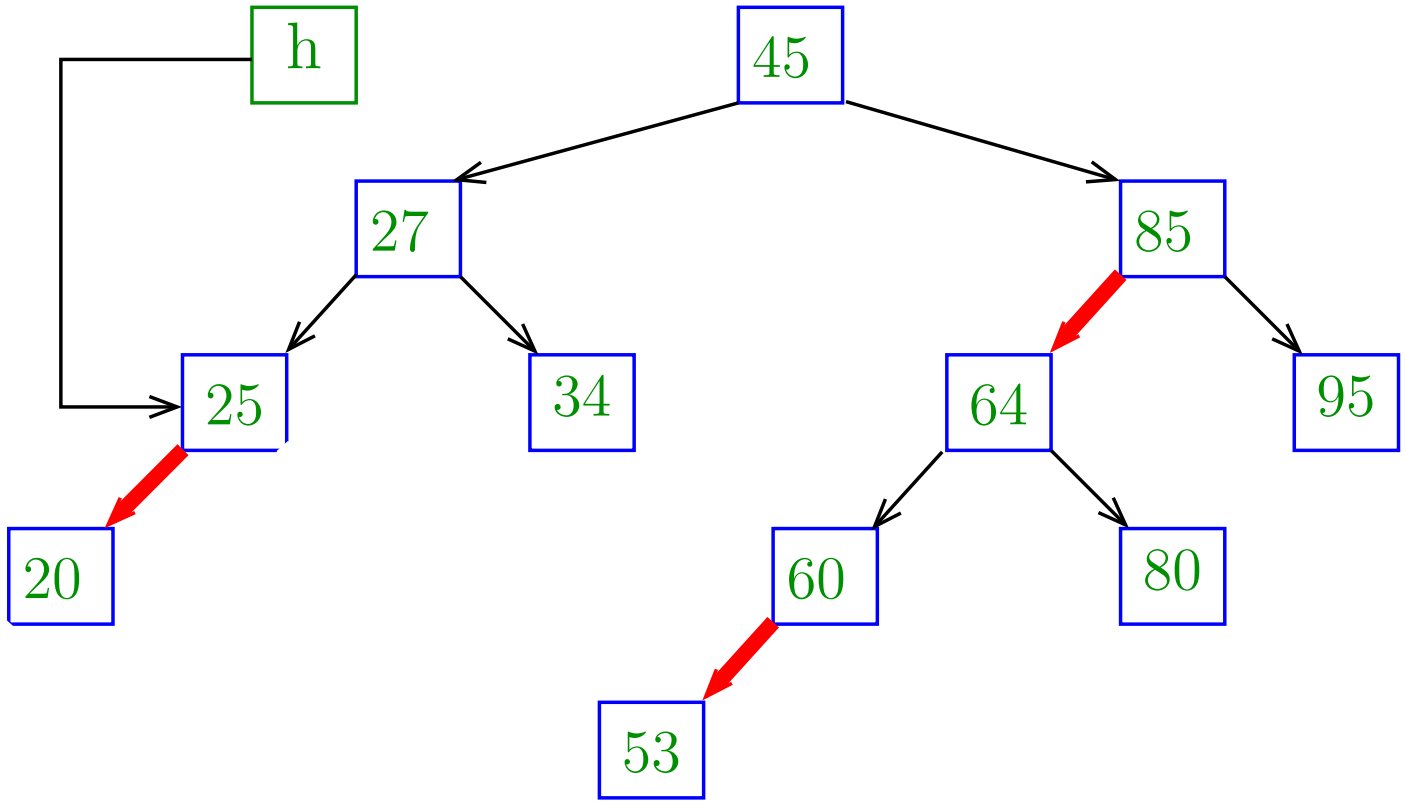
rubro-negra: mais outro `deleteMin()`



`return h;`

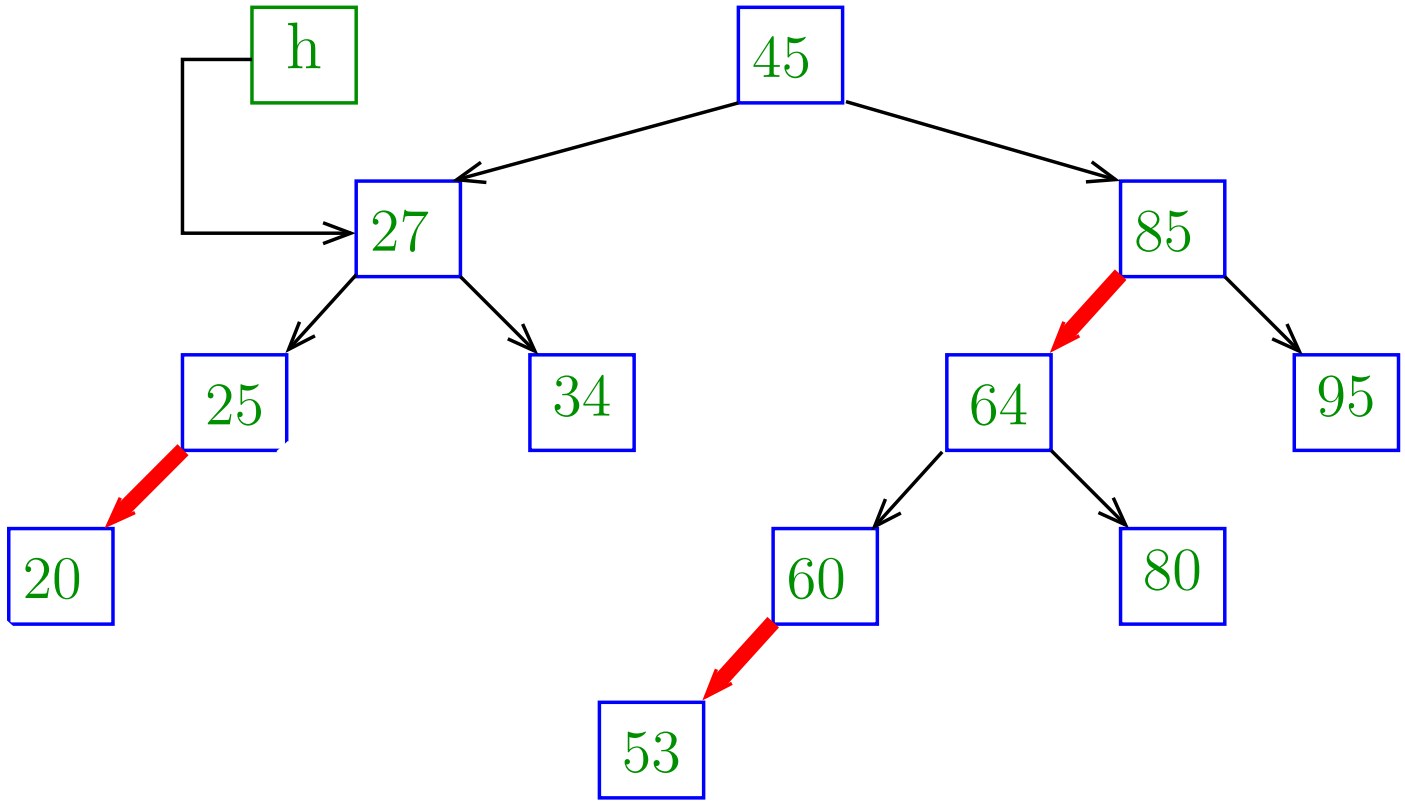
[`balance(h);`]

rubro-negra: mais outro `deleteMin()`



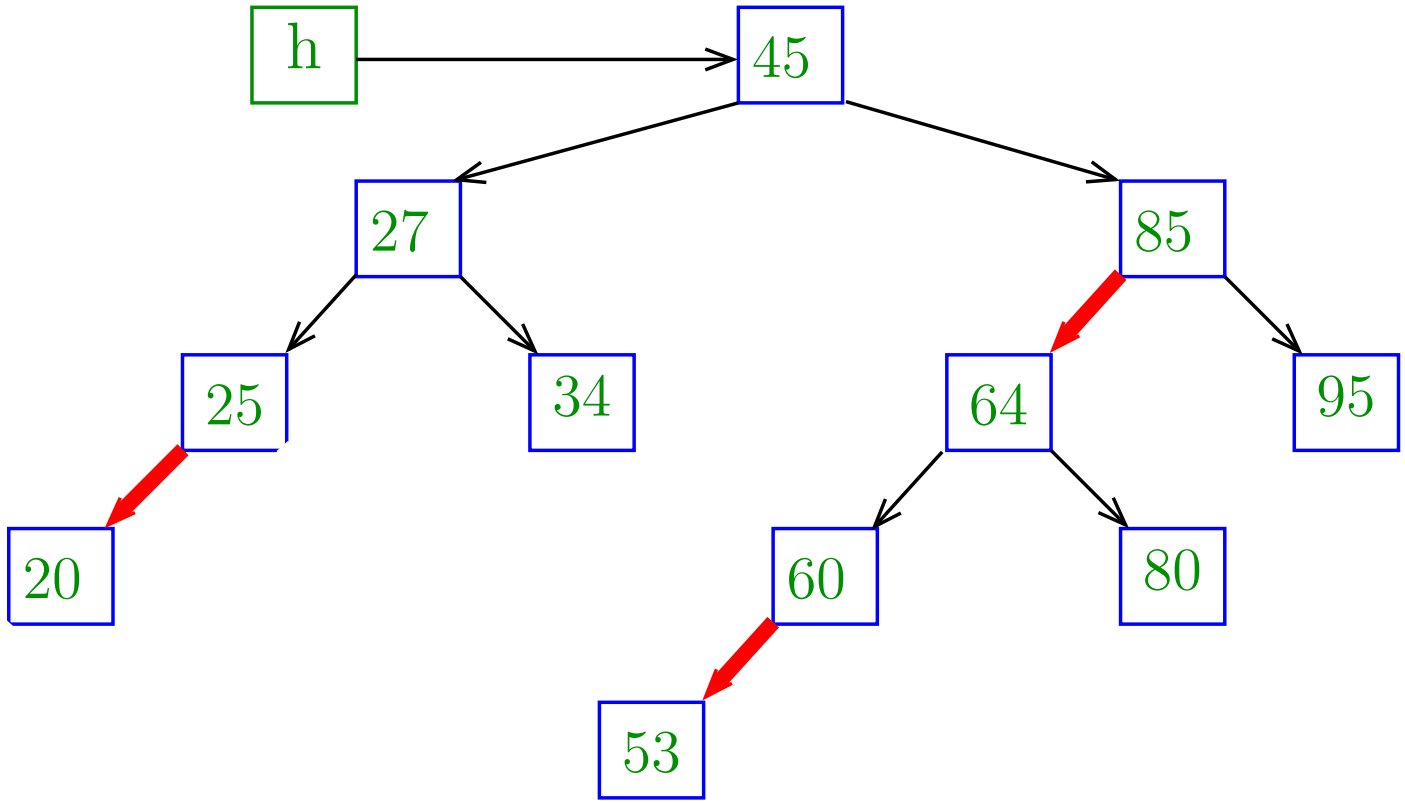

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()`



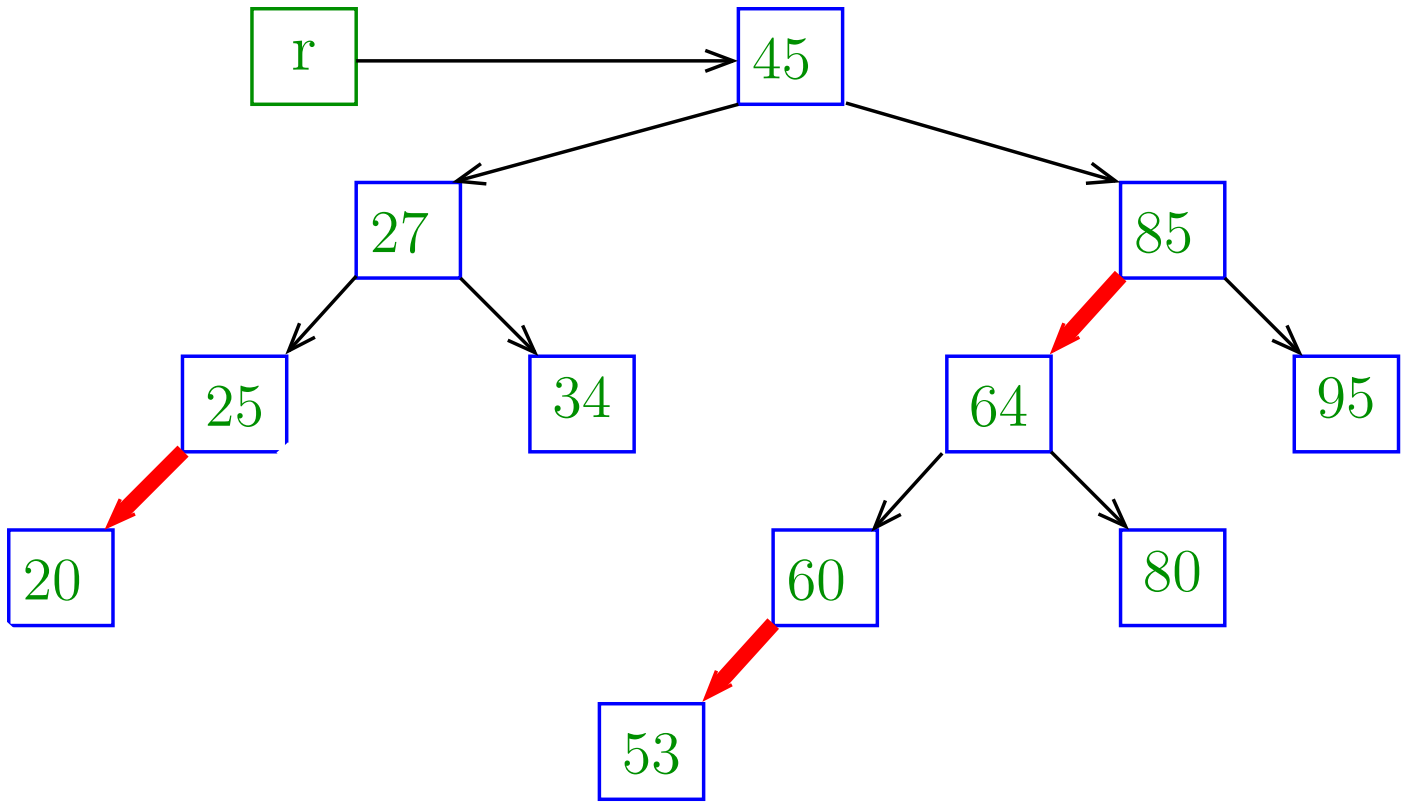

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()`

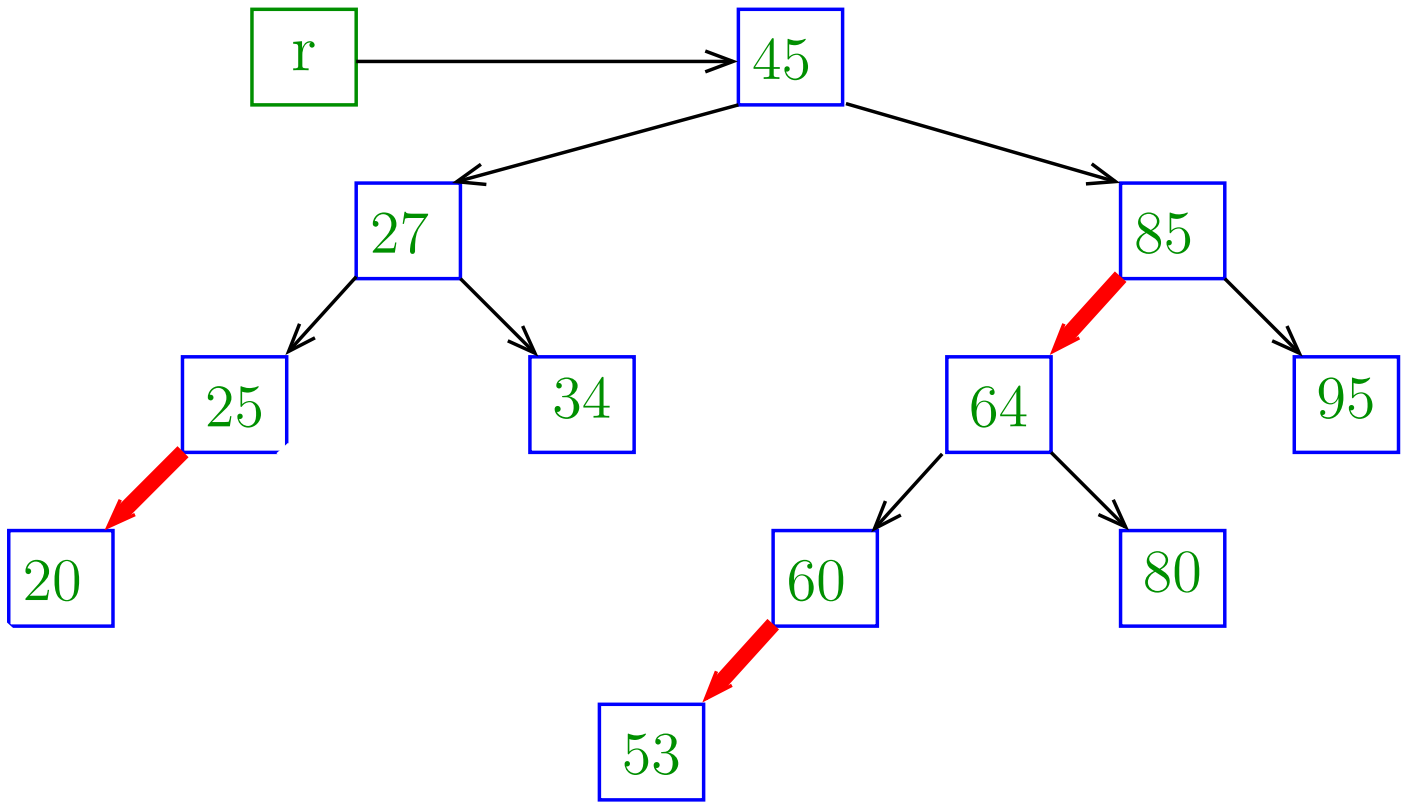



```
return balance(h);
```

rubro-negra: mais outro `deleteMin()`

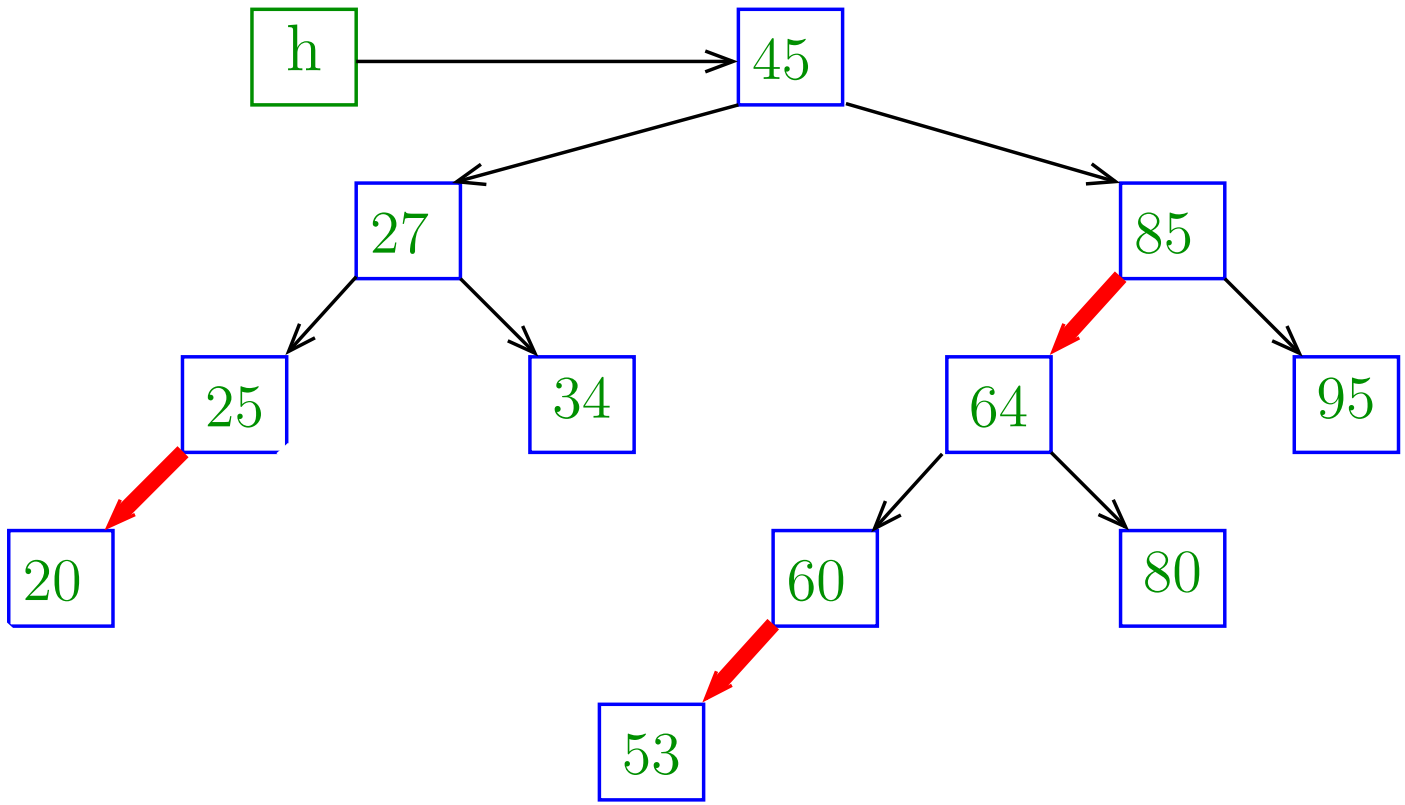


rubro-negra: mais outro `deleteMin()` ainda



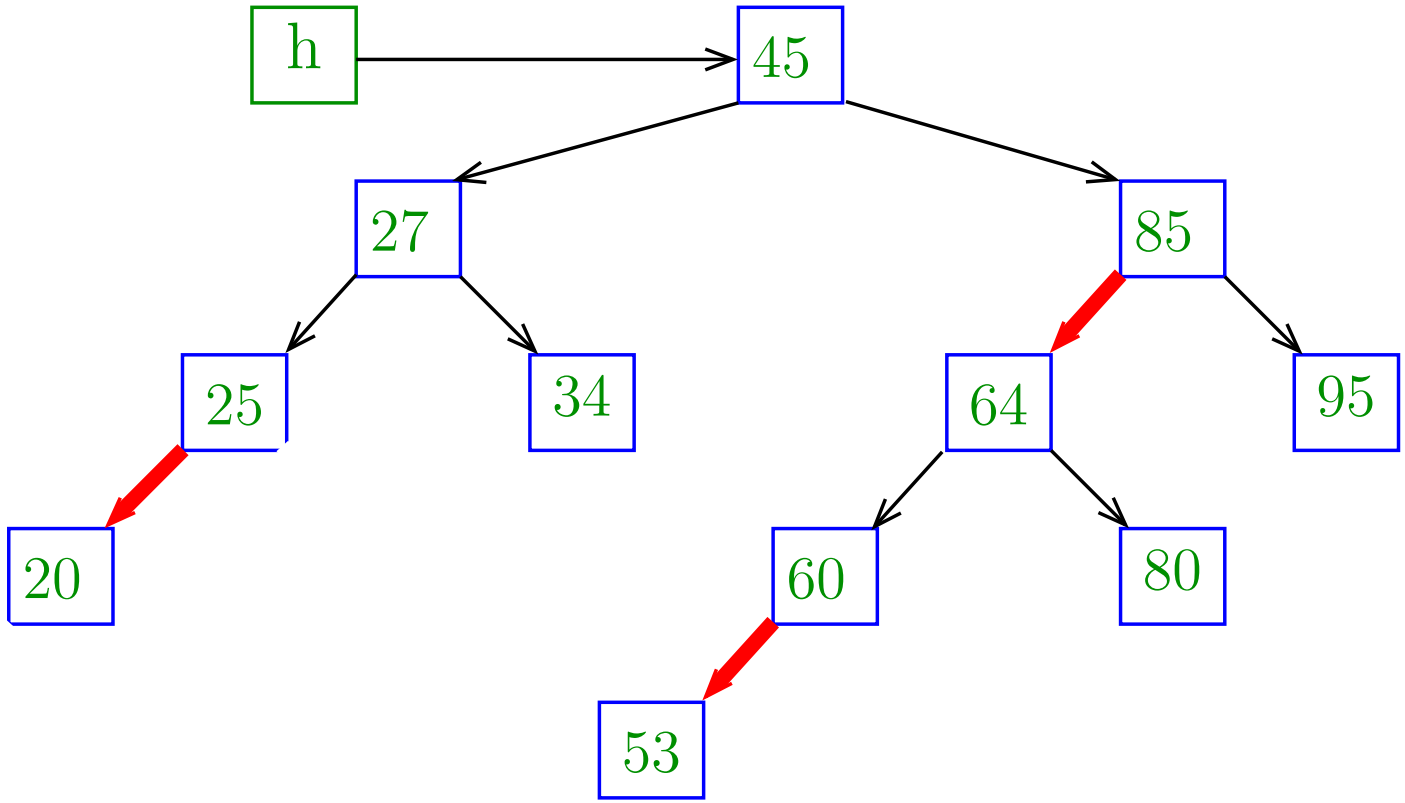
`deleteMin(r);`

rubro-negra: mais outro `deleteMin()` ainda



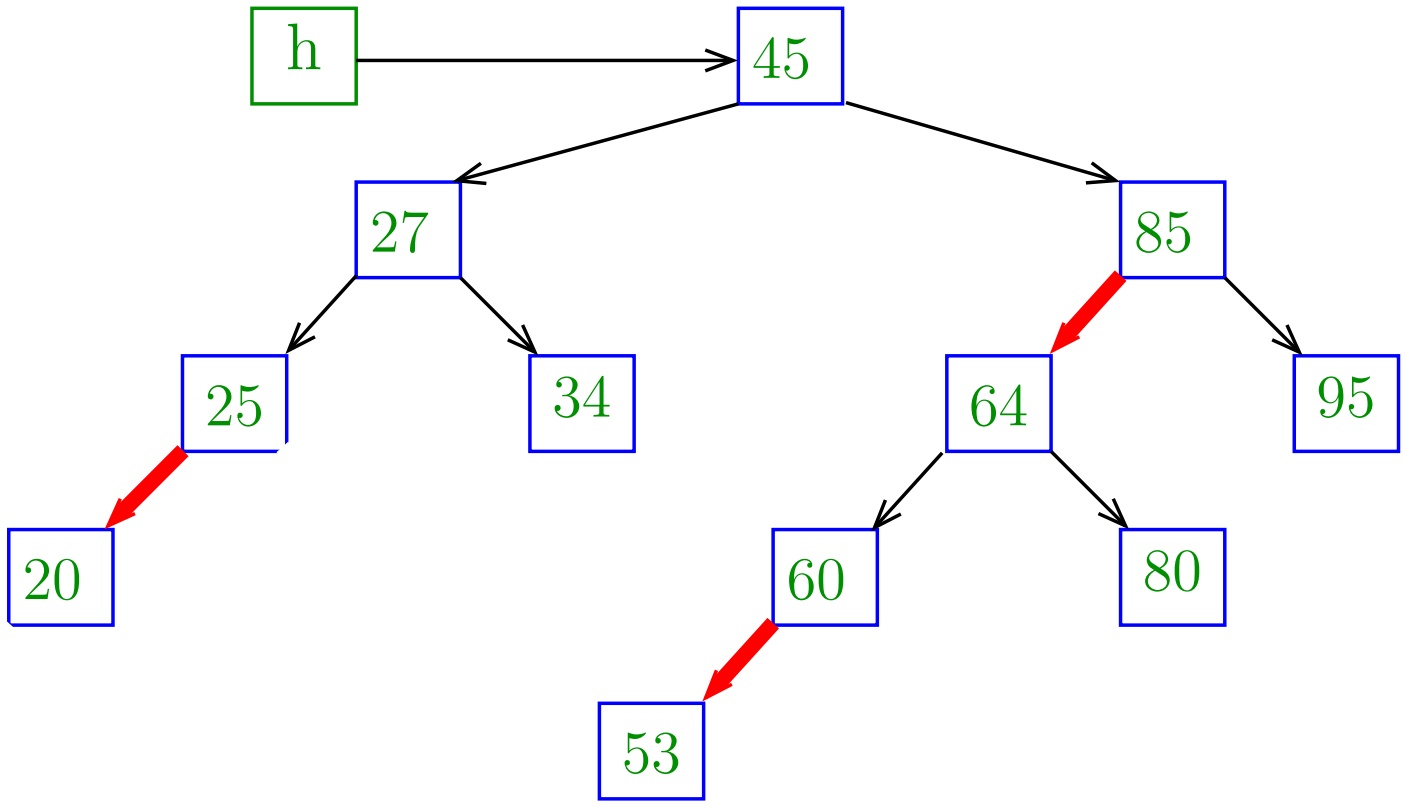
`h.left` = `deleteMin(h.left);`

rubro-negra: mais outro `deleteMin()` ainda



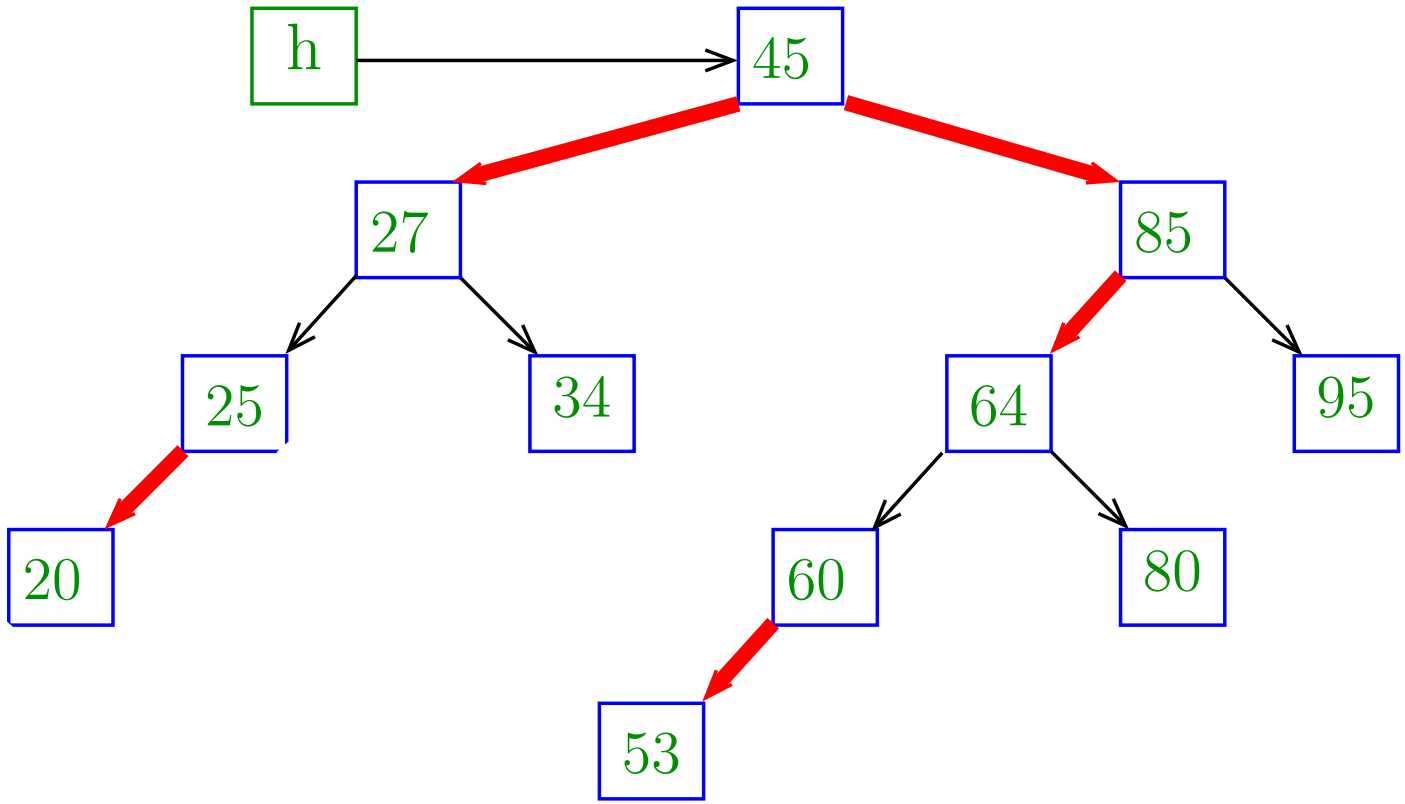
`h= moveRedLeft(h);`

rubro-negra: mais outro `deleteMin()` ainda



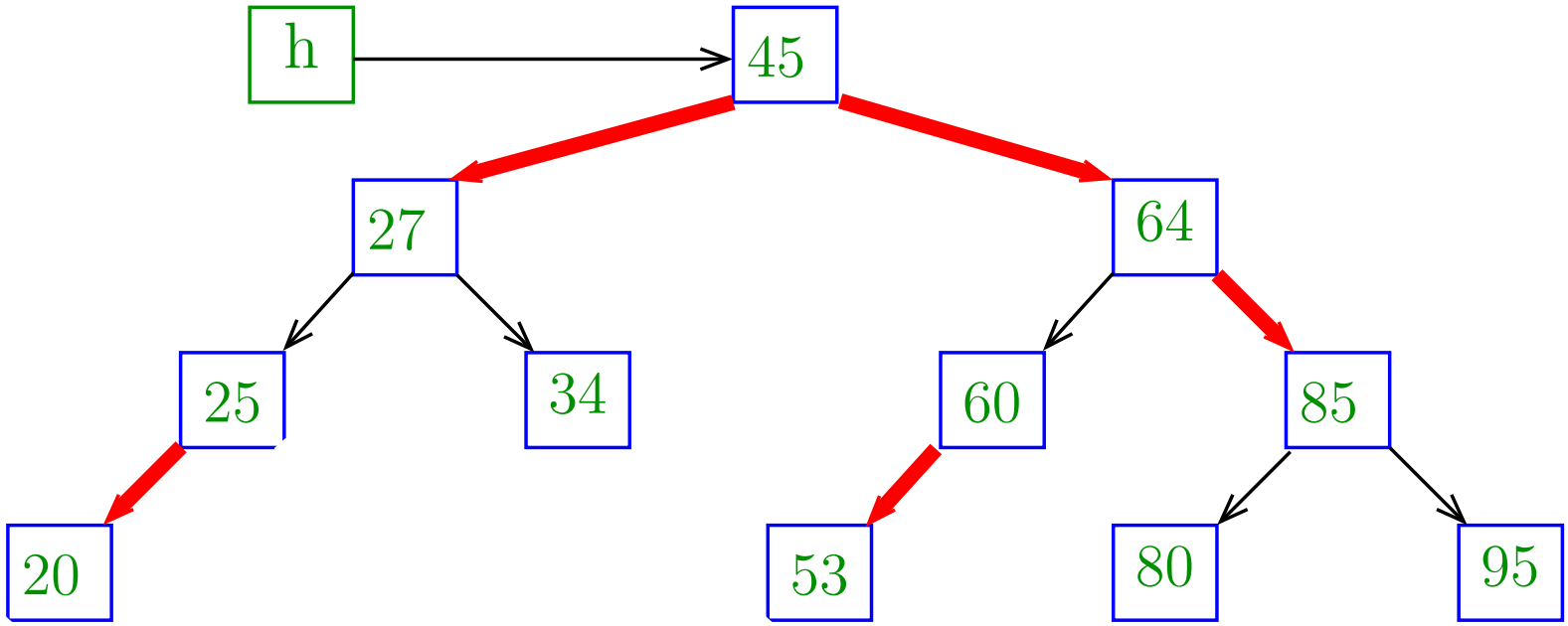
`flipColors(h)` [`moveRedLeft(h)`];

rubro-negra: mais outro `deleteMin()` ainda



`h.right = rotateRight(h.right);` [`moveRedLeft(h);`]

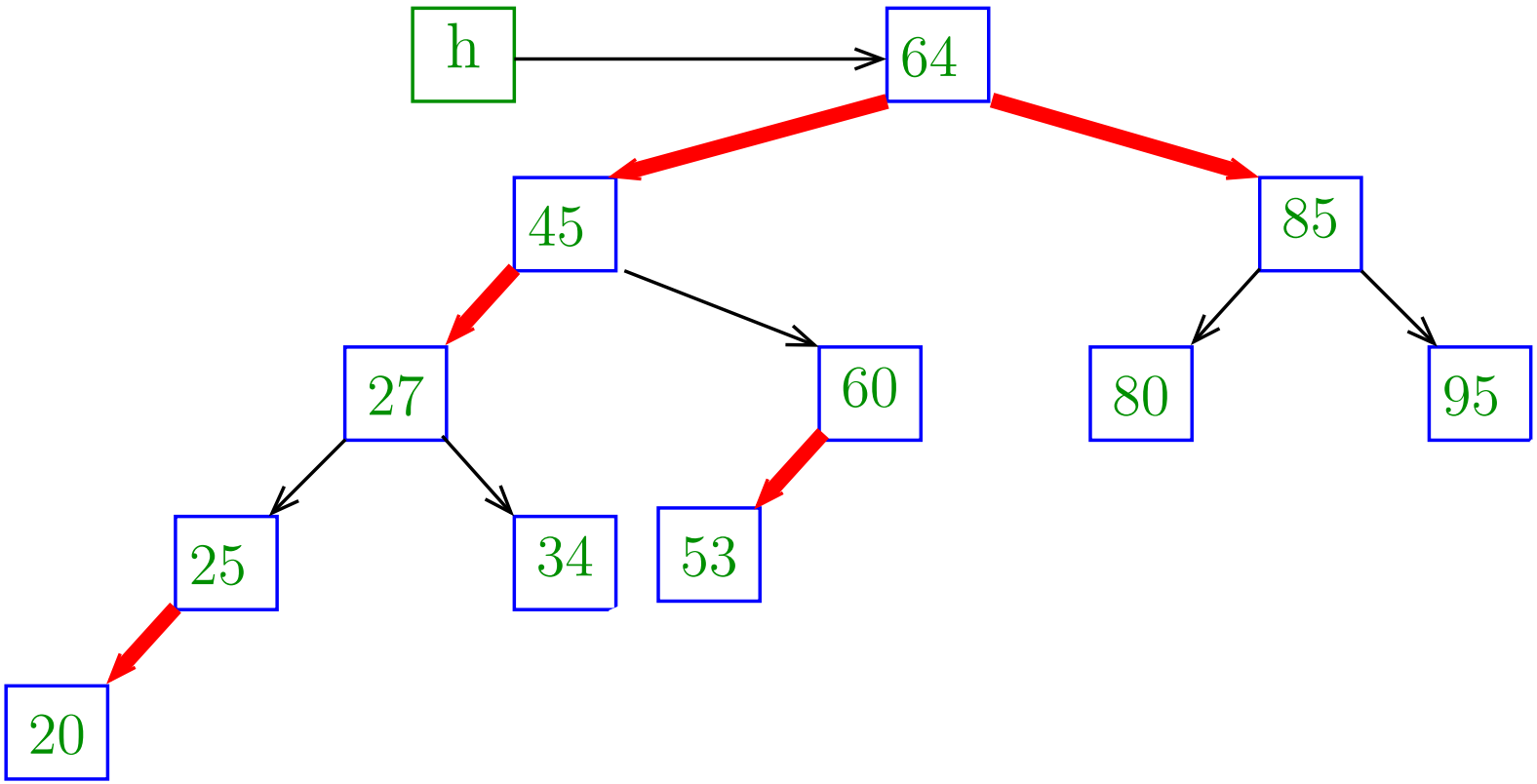
rubro-negra: mais outro `deleteMin()` ainda



`h = rotateLeft(h);`

[`moveRedLeft(h);`]

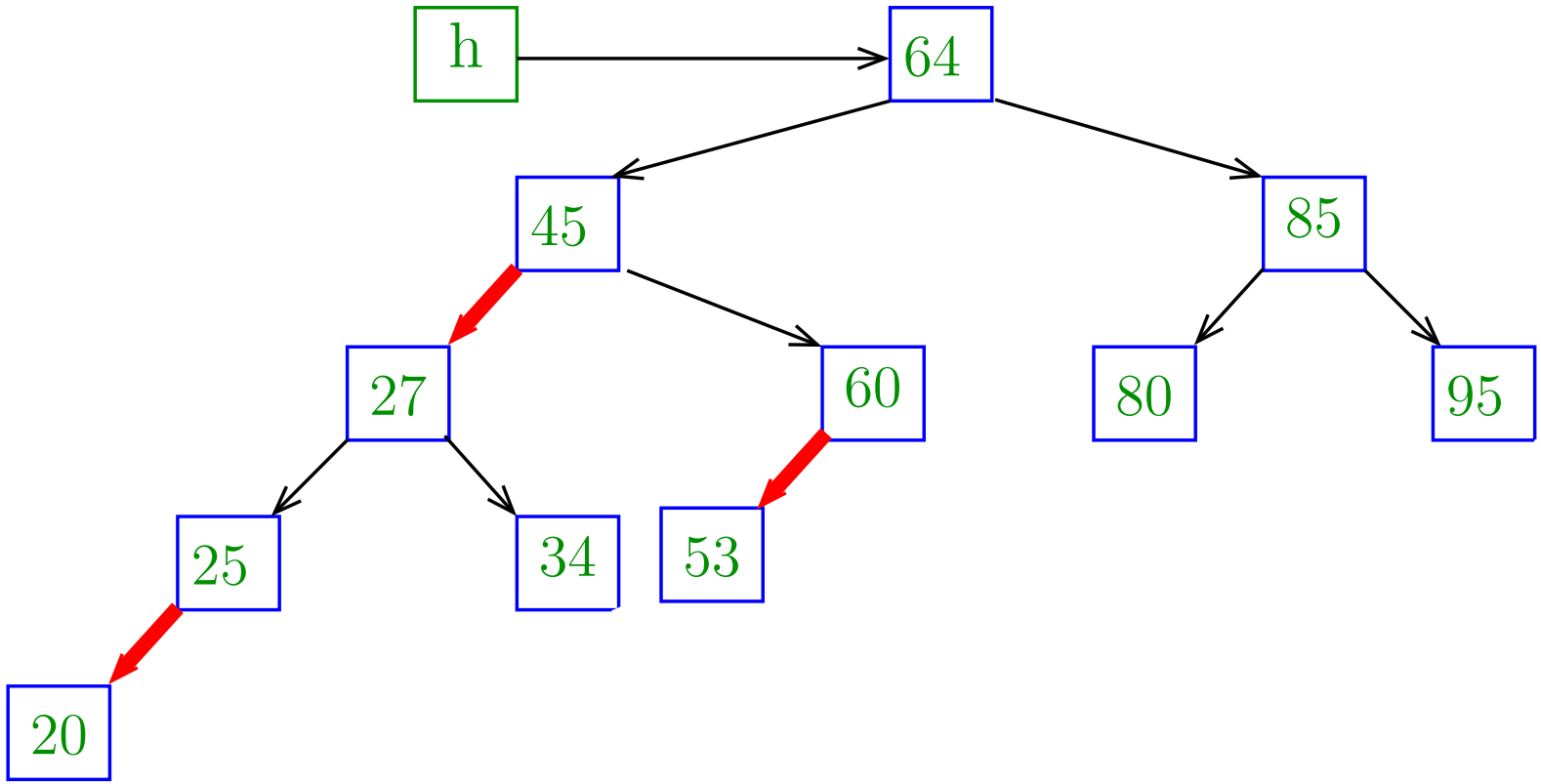
rubro-negra: mais outro `deleteMin()` ainda



`flipColors(h)`

[`moveRedLeft(h)`];

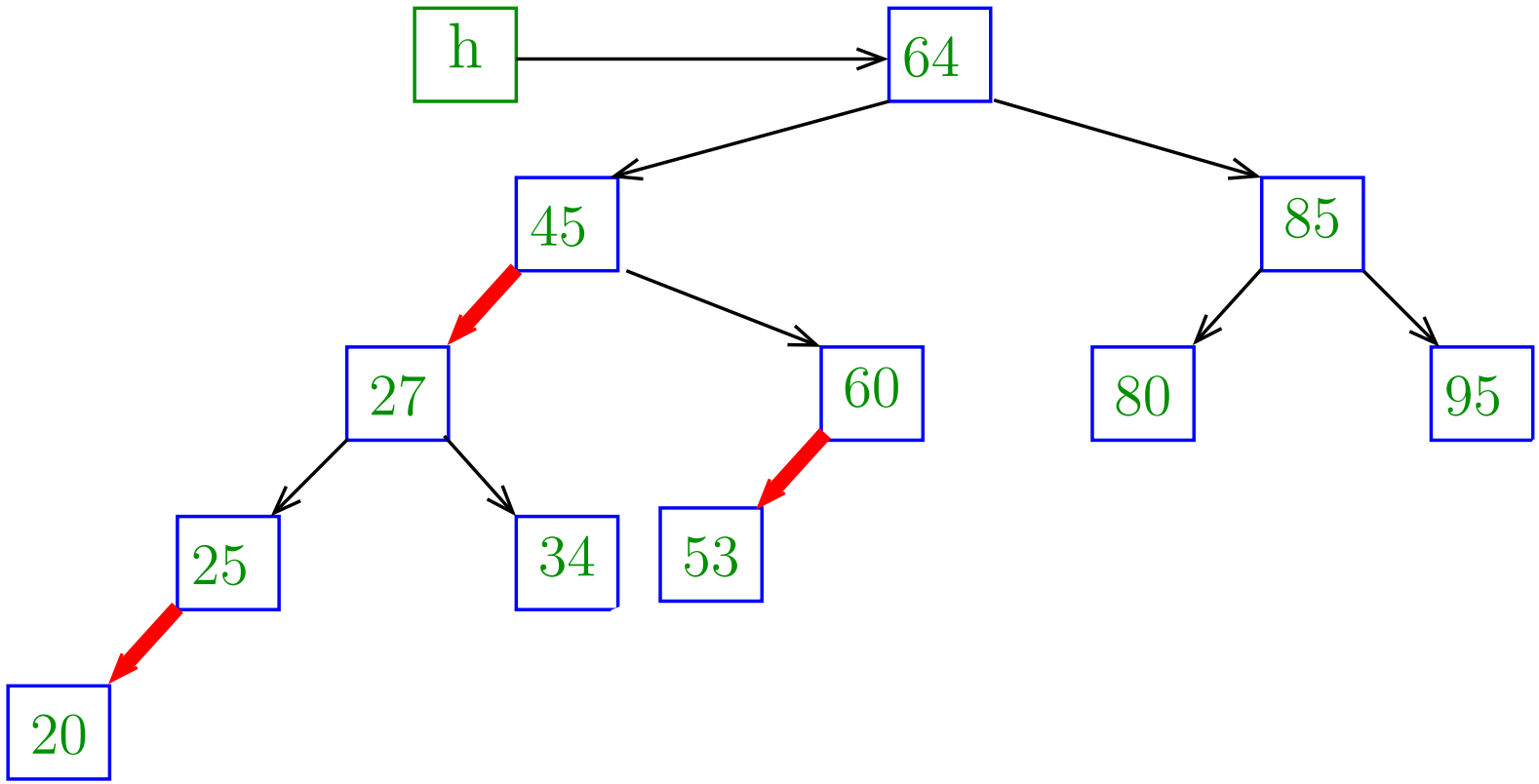
rubro-negra: mais outro `deleteMin()` ainda



`return h;`

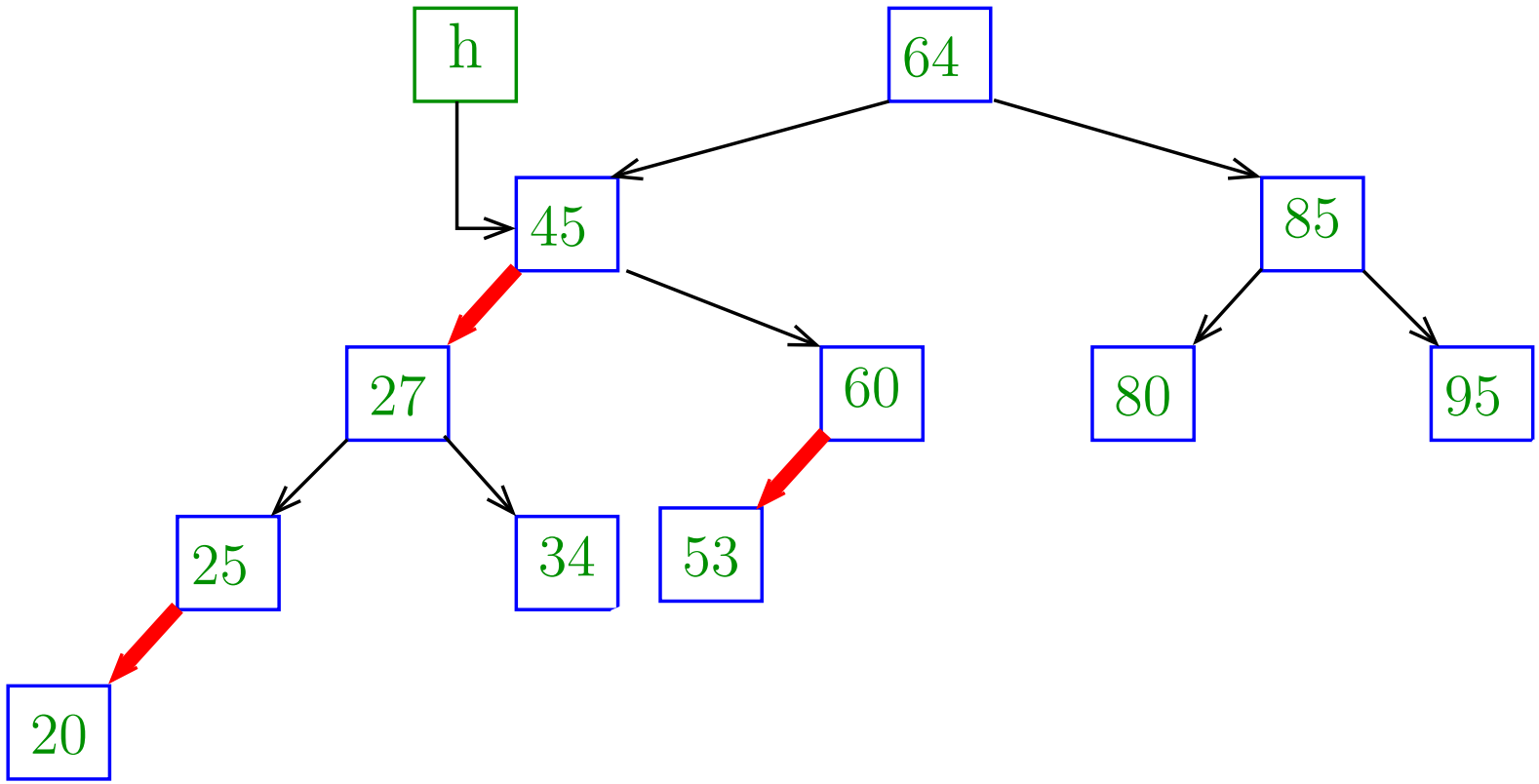
[`moveRedLeft(h)`];

rubro-negra: mais outro `deleteMin()` ainda



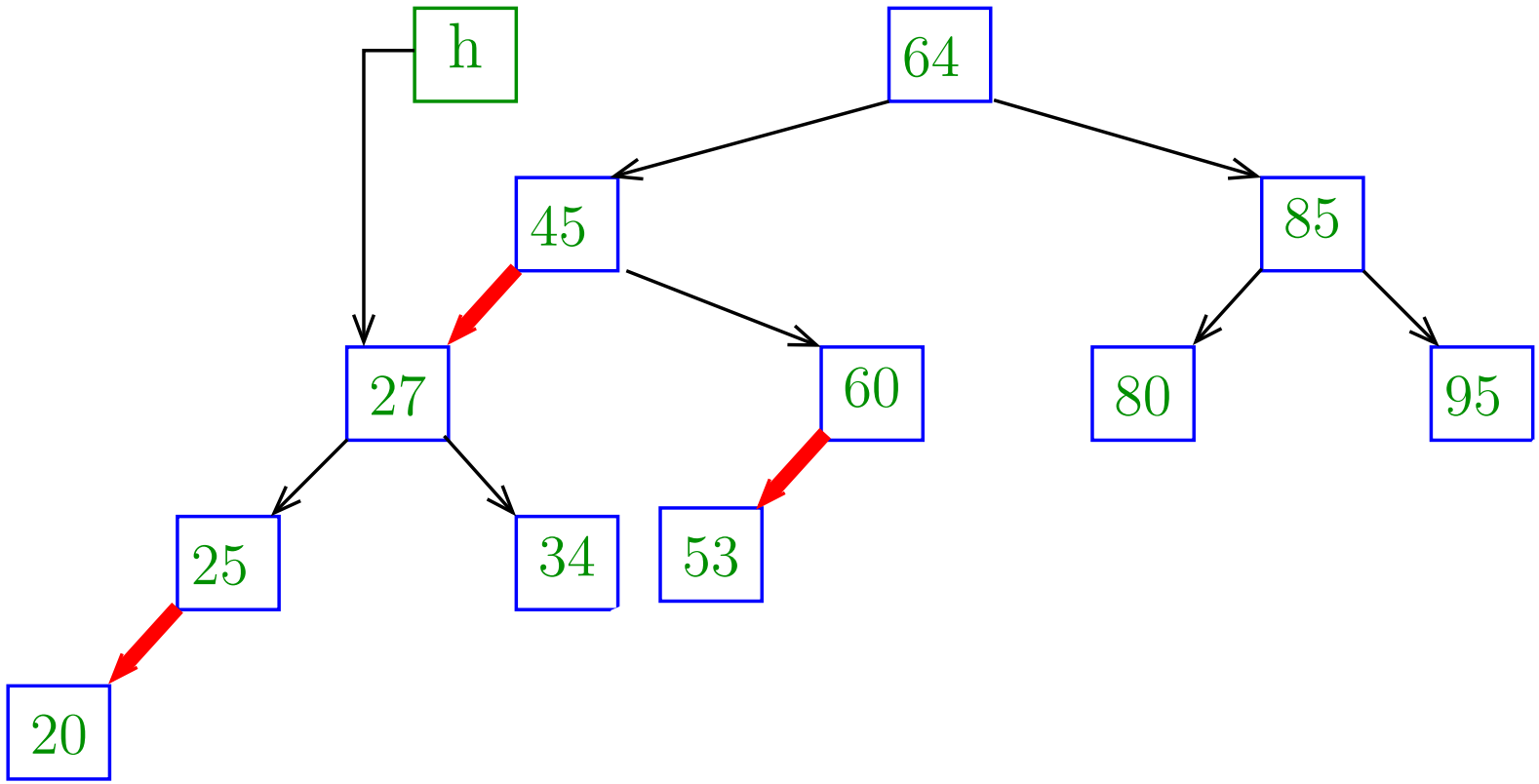

```
h.left = deleteMin(h.left);
```

rubro-negra: mais outro `deleteMin()` ainda



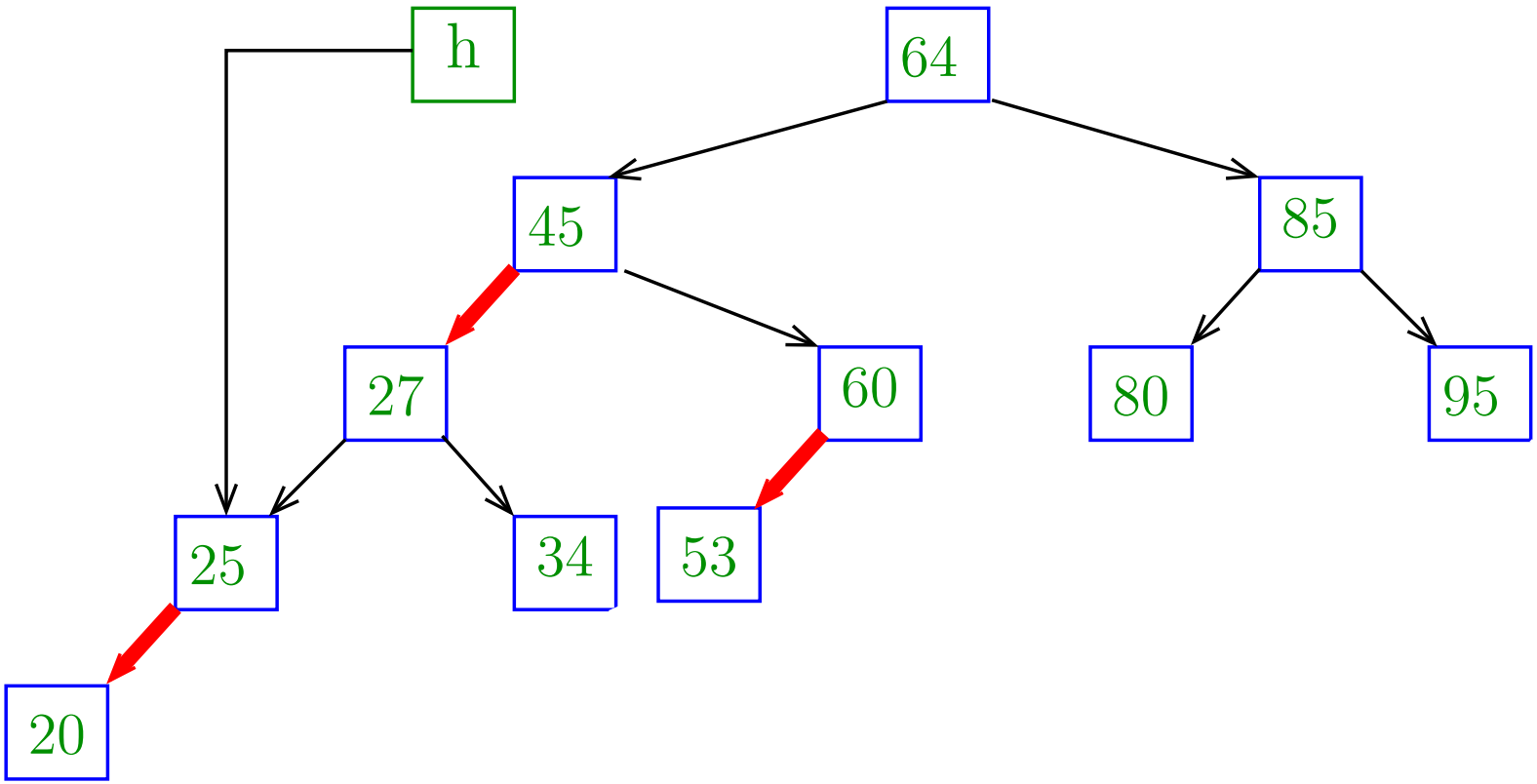
`h.left = deleteMin(h.left);`

rubro-negra: mais outro `deleteMin()` ainda



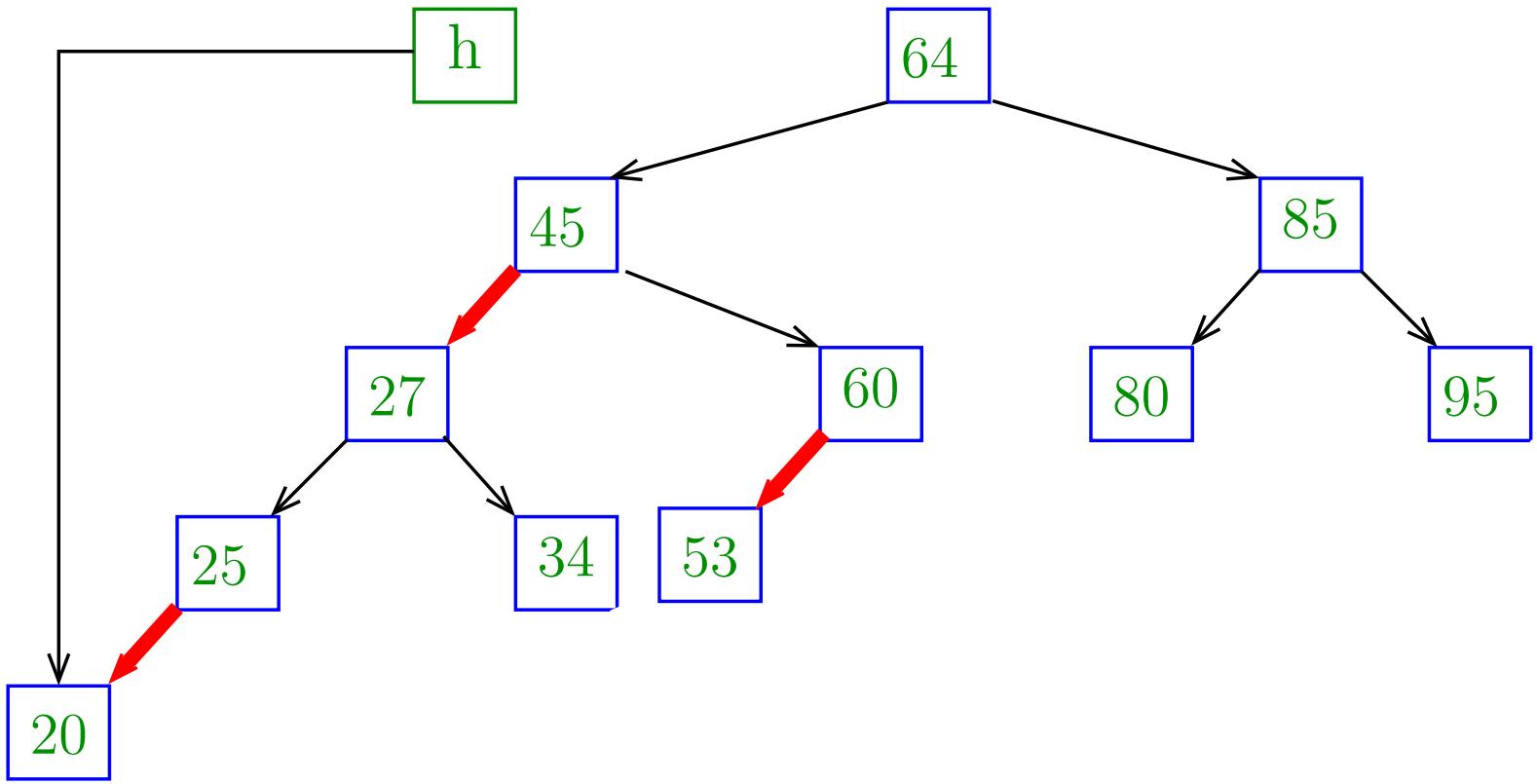
`h.left = deleteMin(h.left);`

rubro-negra: mais outro `deleteMin()` ainda



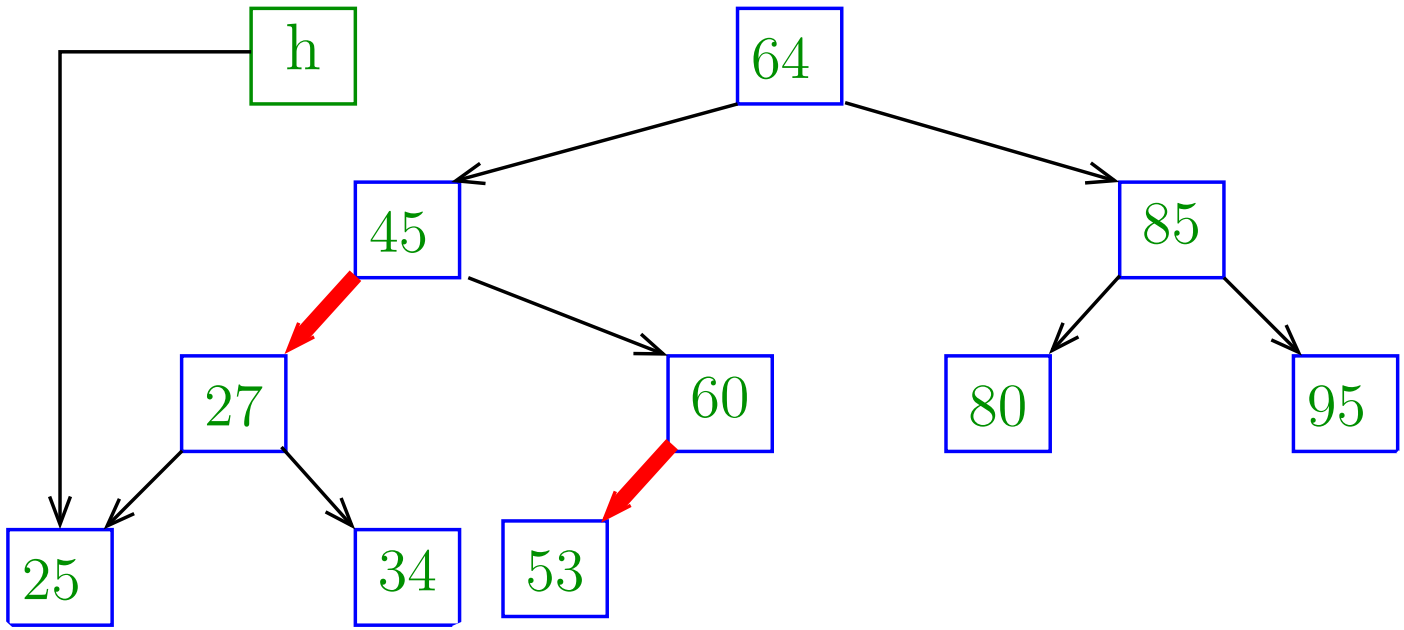

```
h.left = deleteMin(h.left);
```

rubro-negra: mais outro `deleteMin()` ainda



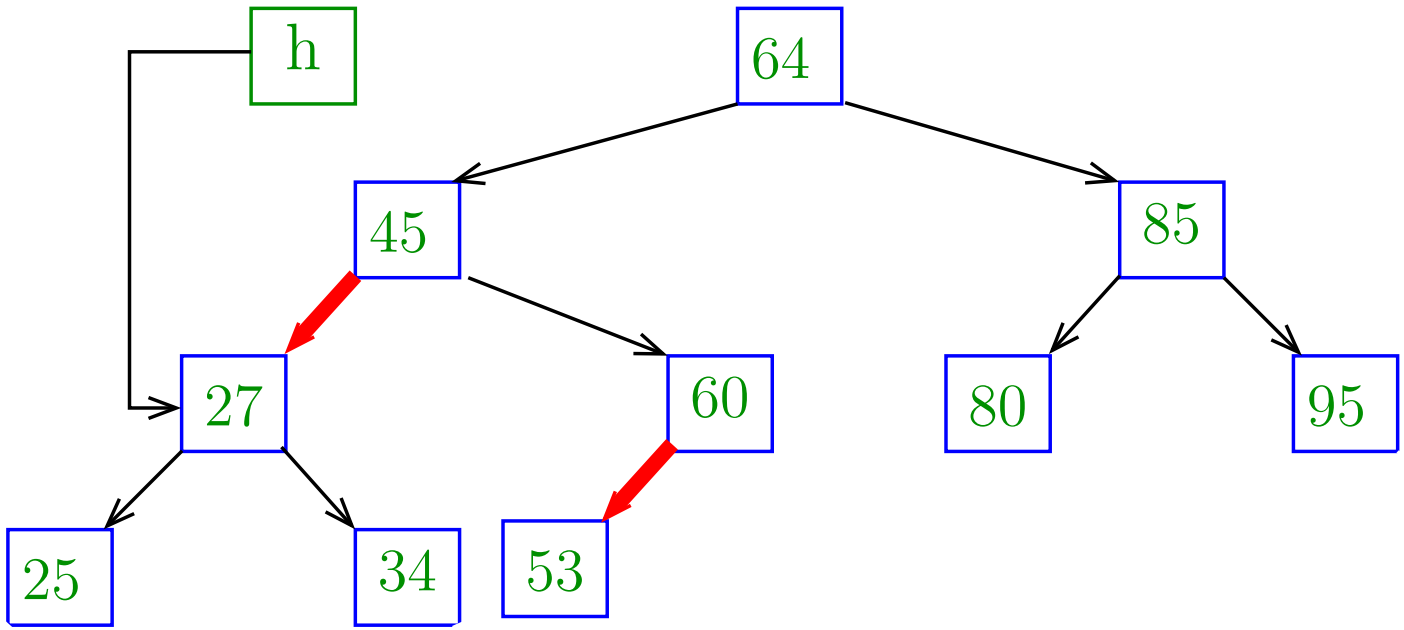

```
return null;
```

rubro-negra: mais outro `deleteMin()` ainda



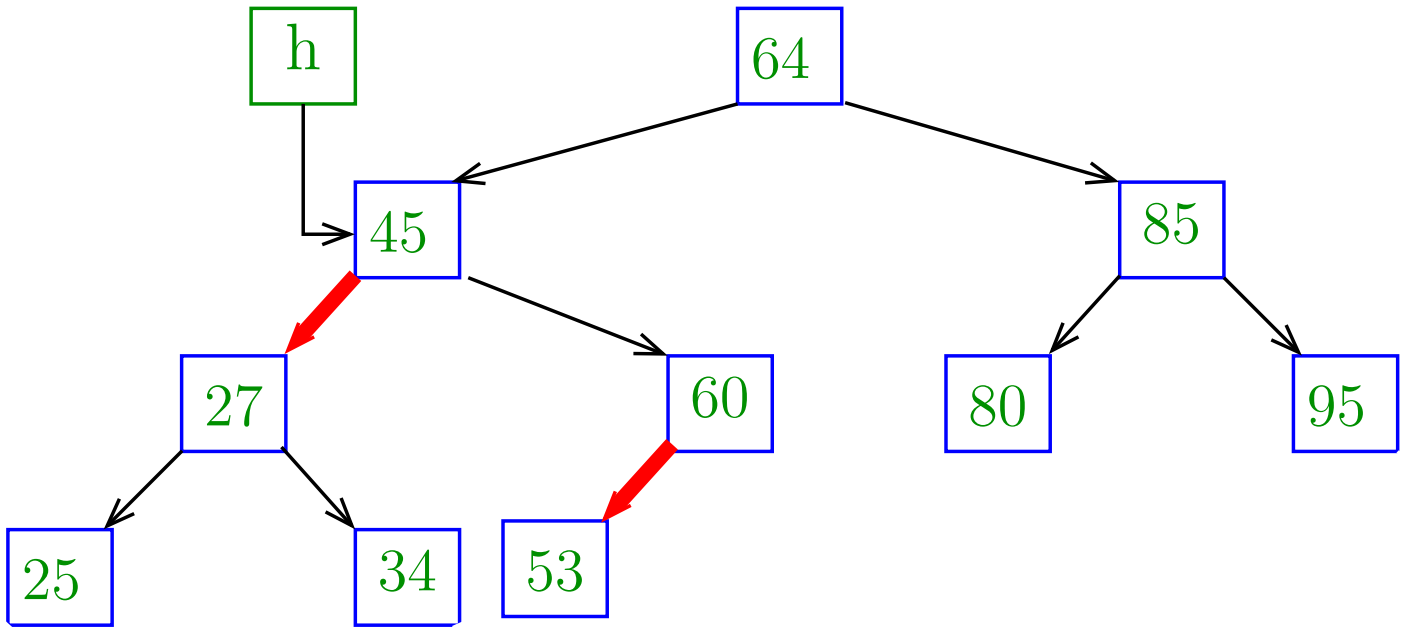

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()` ainda



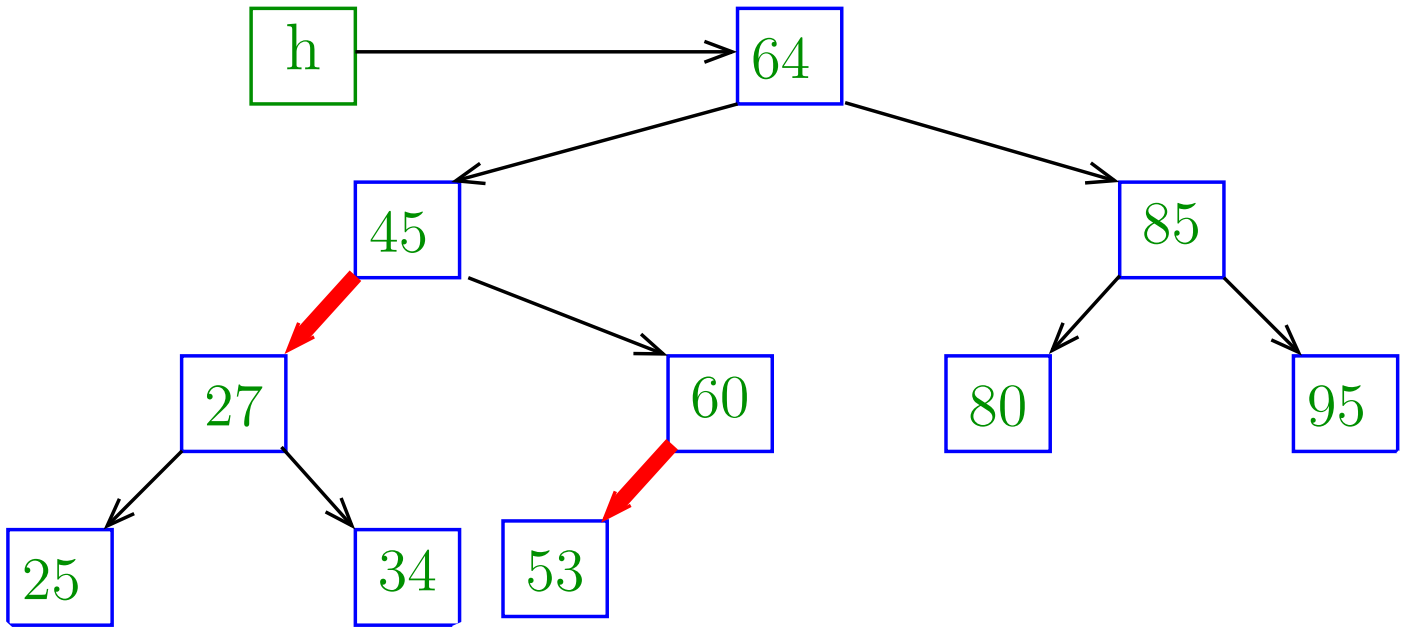

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()` ainda



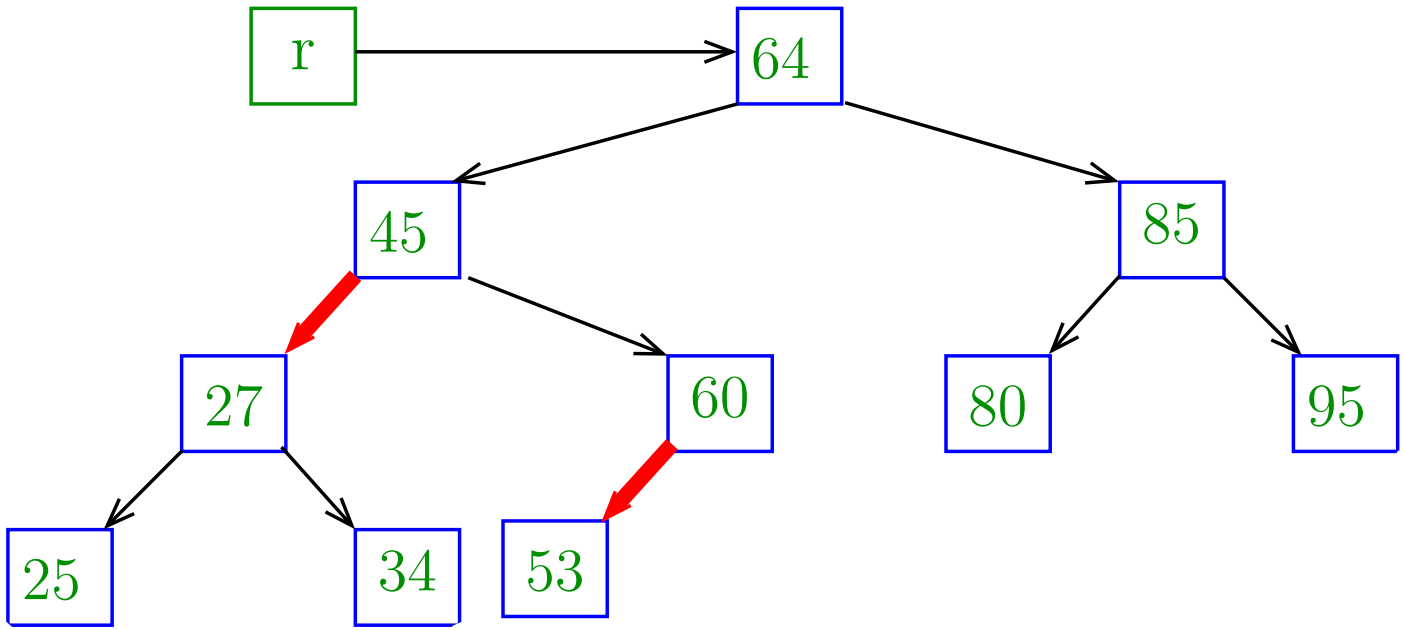

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()` ainda




```
return balance(h);
```

rubro-negra: mais outro `deleteMin()` ainda



deleteMin()

Se ambos os filhos da raiz são **negros** faz a razia rubra.

```
public void deleteMin() {
   if(r == null) return;
   if (!isRed(r.left) && !isRed(r.right))
       r.color = RED;
   r = deleteMin(r);
   if (!isEmpty()) r.color = BLACK;
}
```

deleteMin()

```
private Node deleteMin(Node h) {
   if (h.left == null)
       return null;
   if(!isRed(h.left)&&!isRed(h.left.left))
       h = moveRedLeft(h);
   h.left = deleteMin(h.left);
   return balance(h);
}
```

deleteMin()

Supõe que h é RED e h.left e h.left.left são BLACK.

```
private Node moveRedLeft(Node h) {
   flipColors(h);
   if (isRed(h.right.left)) {
       h.right = rotateRight(h.right);
       h = rotateLeft(h);
       flipColors(h);
   }
   return h;
}
```

deleteMin()

Volta o invariante rubro-**negro**.

```
private Node balance(Node h) {
   if (isRed(h.right))
       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;
}
```

delete()

```
public void delete(Key key) {
   if (!isRed(r.left) && !isRed(r.right))
       r.color = RED;
   r = delete(r, key);
   if (!isEmpty()) r.color = BLACK;
}
```

delete()

```
private Node delete(Node h, Key key) {
   if (key.compareTo(h.key) < 0) {
       if(!isRed(h.left)&&!isRed(h.left.left))
           h = moveRedLeft(h);
       h.left = delete(h.left, key);
   }
   else {
       if (isRed(h.left))
           h = rotateRight(h);
       if (key.equals(h.key) &&
                  (h.right == null))
           return null;
```

delete()

```
    if(!isRed(h.right)&&!isRed(h.right.left))
        h = moveRedRight(h);
    if (key.equals(h.key)) {
        Node x = min(h.right);
        h.key = x.key;
        h.val = x.val;
        h.right = deleteMin(h.right);
    }
    else h.right = delete(h.right, key);
}
return balance(h);
}
```

delete()

Supõe que h é RED e h.right e h.right.left são BLACK.

```
private Node moveRedRight(Node h) {
   flipColors(h);
   if (isRed(h.left.left)) {
       h = rotateRight(h);
       flipColors(h);
   }
   return h;
}
```

Observação

BSTs são estruturas ordenadas.

Como as chaves de uma BST são comparáveis, podemos perguntar pela chave mínima e pela chave máxima.

Já fizemos isso.

O **piso** (`floor()`) de uma chave `key` na BST é a maior chave da BST que é menor que ou igual a `key`.

O **teto** (`ceiling()`) de uma chave `key` na BST é a menor chave da BST que é maior que ou igual a `key`.

Os métodos `min()`, `max()`, `floor()` e `ceiling()` são exatamente os mesmos das BSTs ordinárias!

floor()

Devolve `null` se `key` não tem piso nesta `BST`.

```
public Key floor(Key key) {
   Node x = floor(r, key);
   if (x == null) return null;
   return x.key;
}
```

floor()

Devolve o nó que contém o piso de key na subárvore com raiz x. Devolve null se esse piso não existe.

```
private Node floor(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if (cmp == 0) return x;
   if (cmp < 0) return floor(x.left, key);
   Node t = floor(x.right, key);
   if (t != null) return t;
   else return x;
}
```