MAC0323 Algoritmos e Estruturas de Dados II

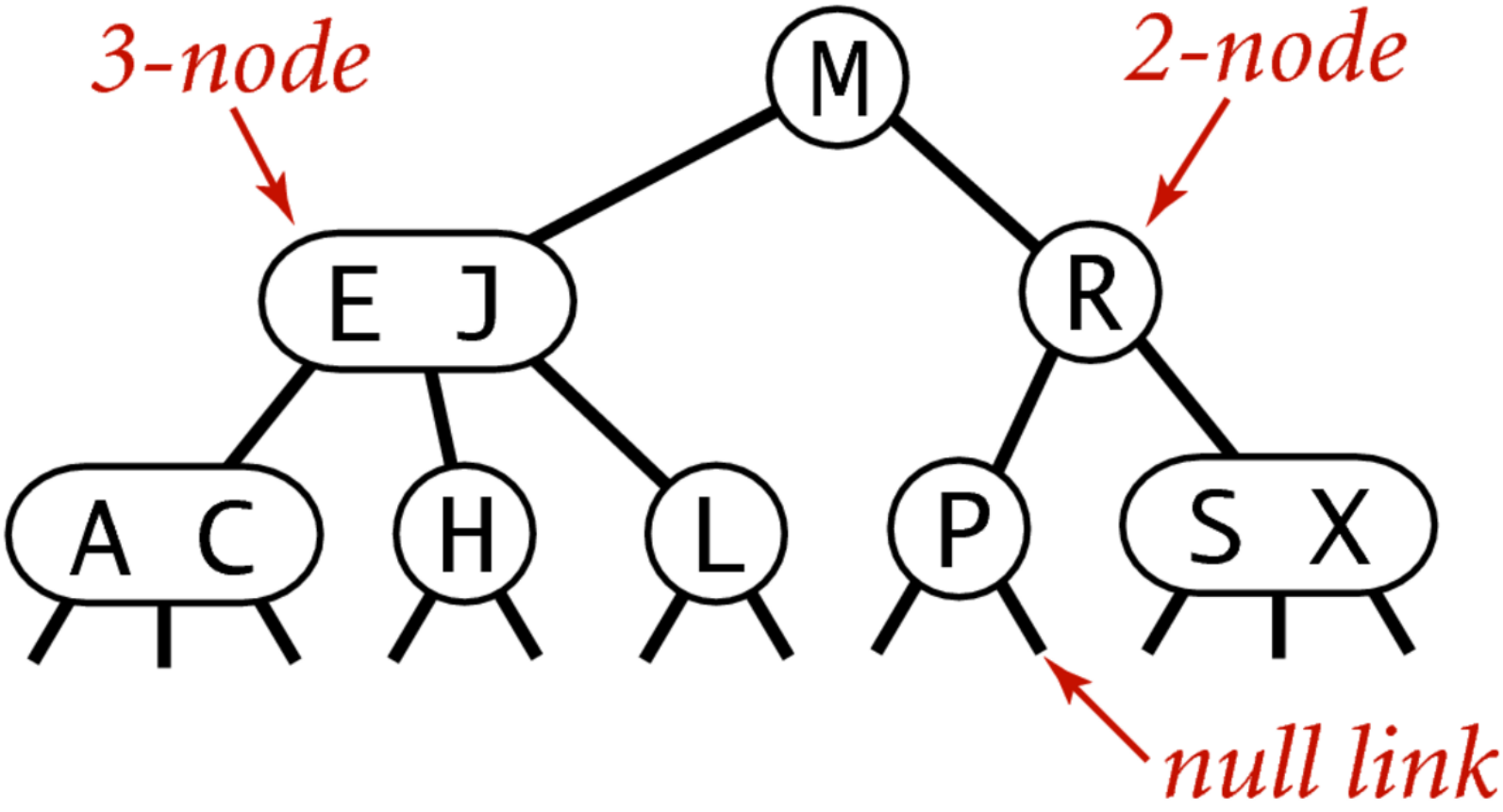
Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 12

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.

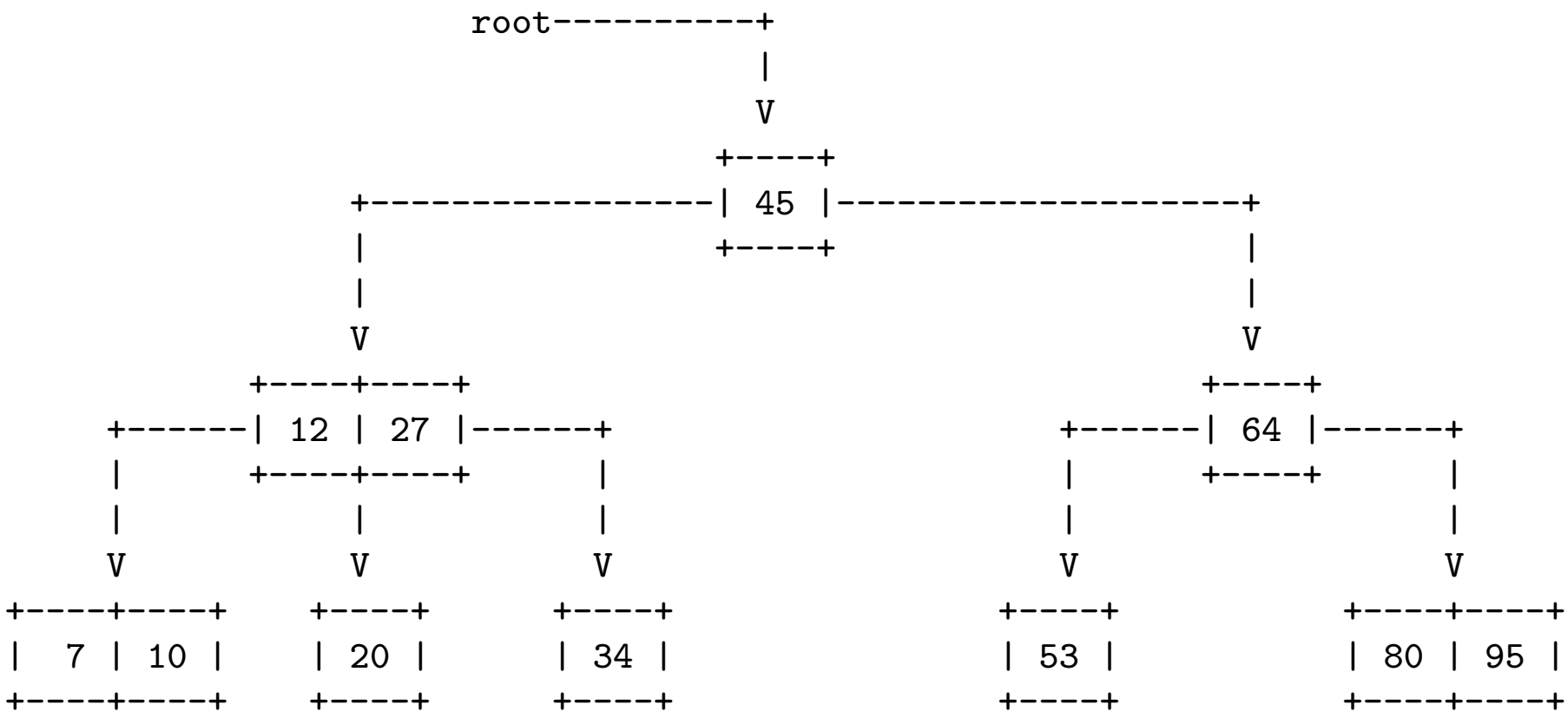
Consequência. Toda árvore 2-3 com $\mathtt{n}$ nós tem altura não superior a $\lg(\mathtt{n}+1)-1$ e não inferior a $\log_3(\mathtt{n}+1)-1$.

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

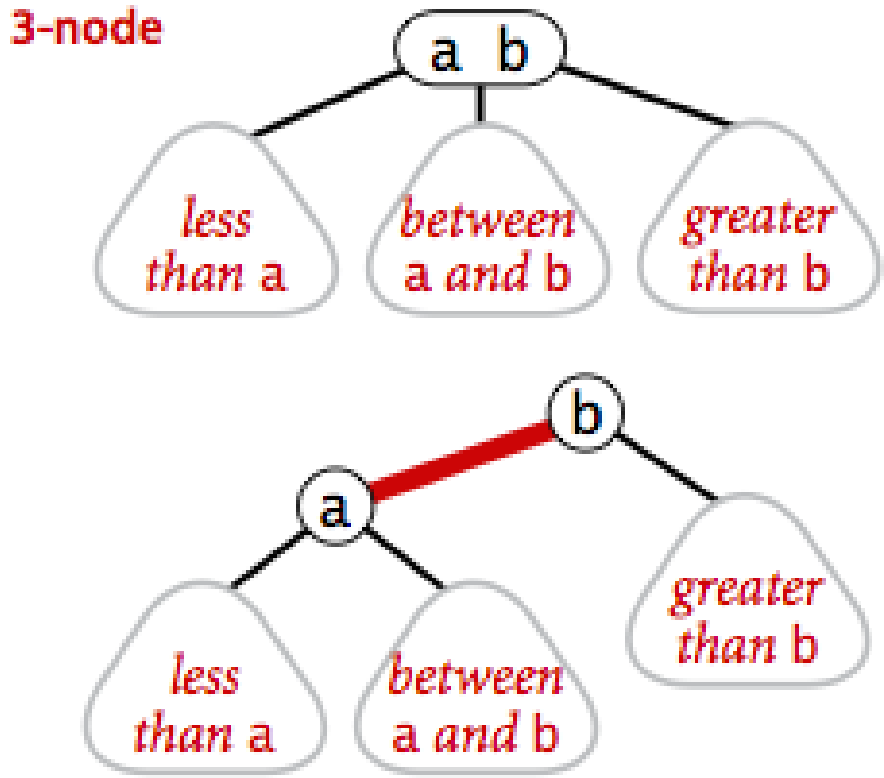
Fonte: http://scottlobdell.me/

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

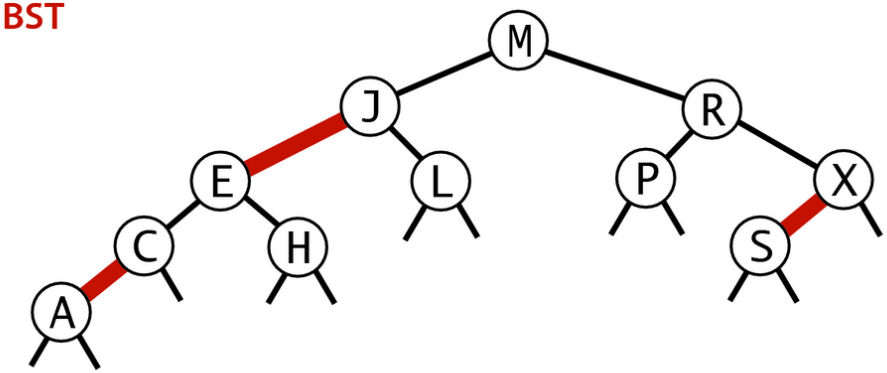
Uma **BST rubro-negra** é uma `BST`
cujos links são negros e rubros e:

- 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.

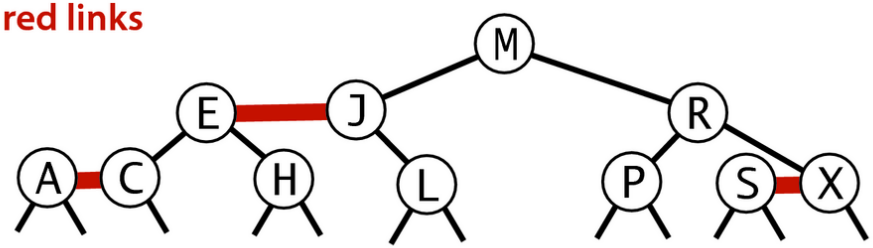
Se os links rubros forem desenhados horizontalmente e depois contraídos, teremos uma árvore 2-3

Anatomia de uma árvore rubro-negra

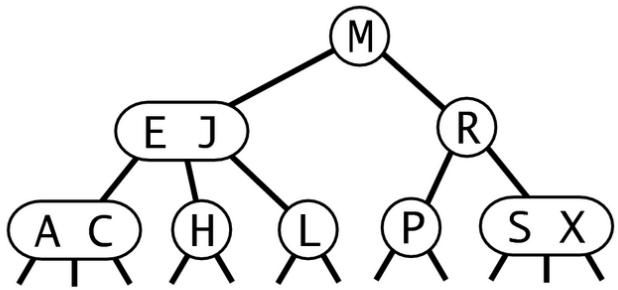
red-black BST



horizontal red links



2-3 tree

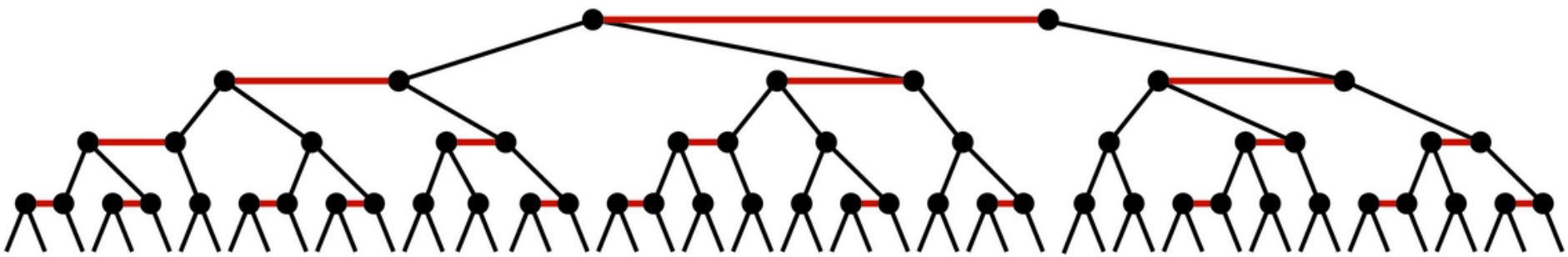


Fonte: algs4

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

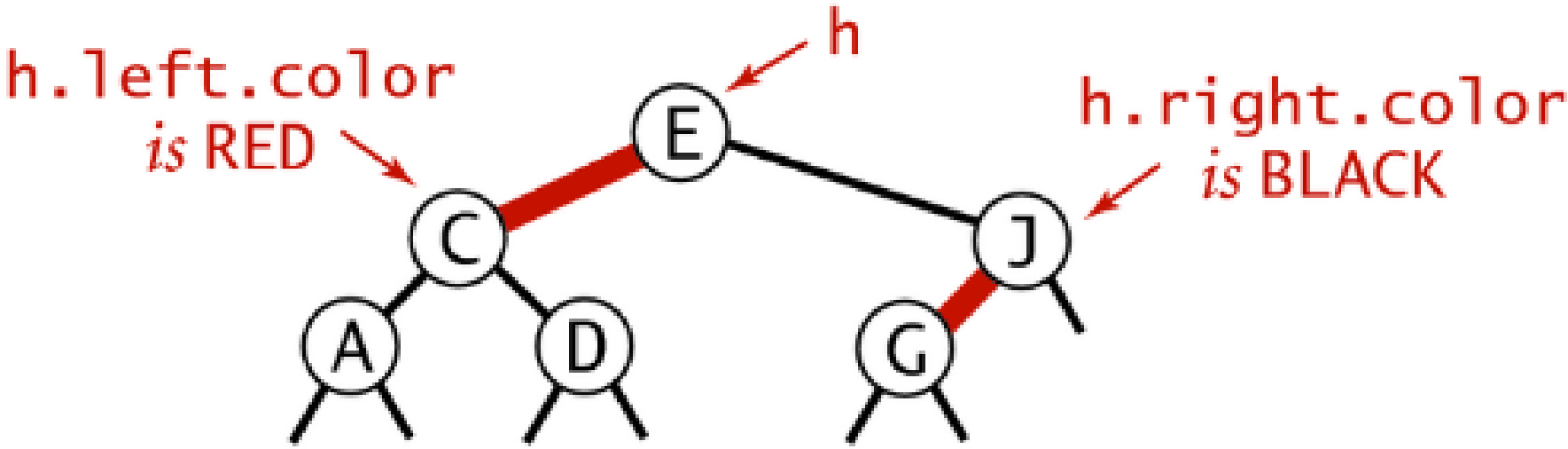
É inconveniente armazenar a cor de um link na estrutura de dados; é mais simples armazenar essa informação nos nós.

A cor de um nó é a cor do único link que entra nele.

A raiz é considerada **negra**.

```
static bool RED = true;
static bool BLACK = false;
```

Nós de uma BST rubro-negra



Nós de uma BST rubro-negra

```
typedef struct node *Node;

struct node {
    Key key;    Value val;
    int n; /* número de nós nesta subárvore */
    bool color; /* cor do link para este nó */
    Node left, right;
}

Node newNode(Key key, Value val,
                    int n, bool color) {
    Node p = mallocSafe(sizeof(*p));
    p->key = key;       p->val = val;
    p->n = n;           p->color = color;
    p->left = NULL;     p->right = NULL;
    return p;
}
```

Nós de uma BST rubro-negra

```
typedef struct node *Node;

struct node {
   Key key;   Value val;
   int n; /* número de nós nesta subárvore */
   bool color; /* cor do link para este nó */
   Node left, right;
}

static bool isRed(Node x) {
   if (x == NULL) return false;
   return x->color == RED;
}
```

AULA 13

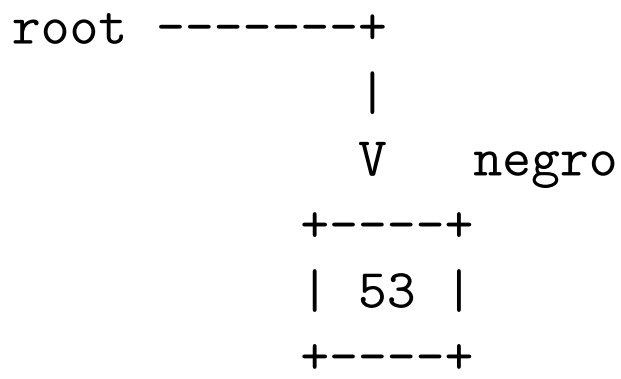
BSTs rubro-negras: `put()`

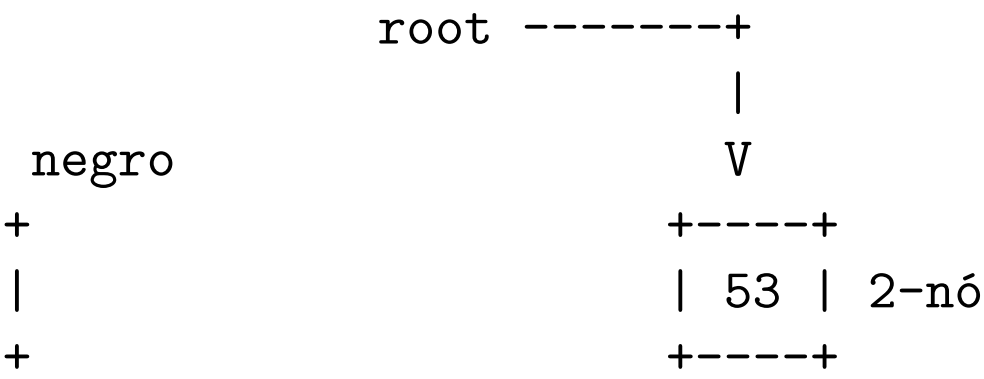
Fonte: `...-crimson-vermillion-Chinese/`

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

Inserção em um 2-nó

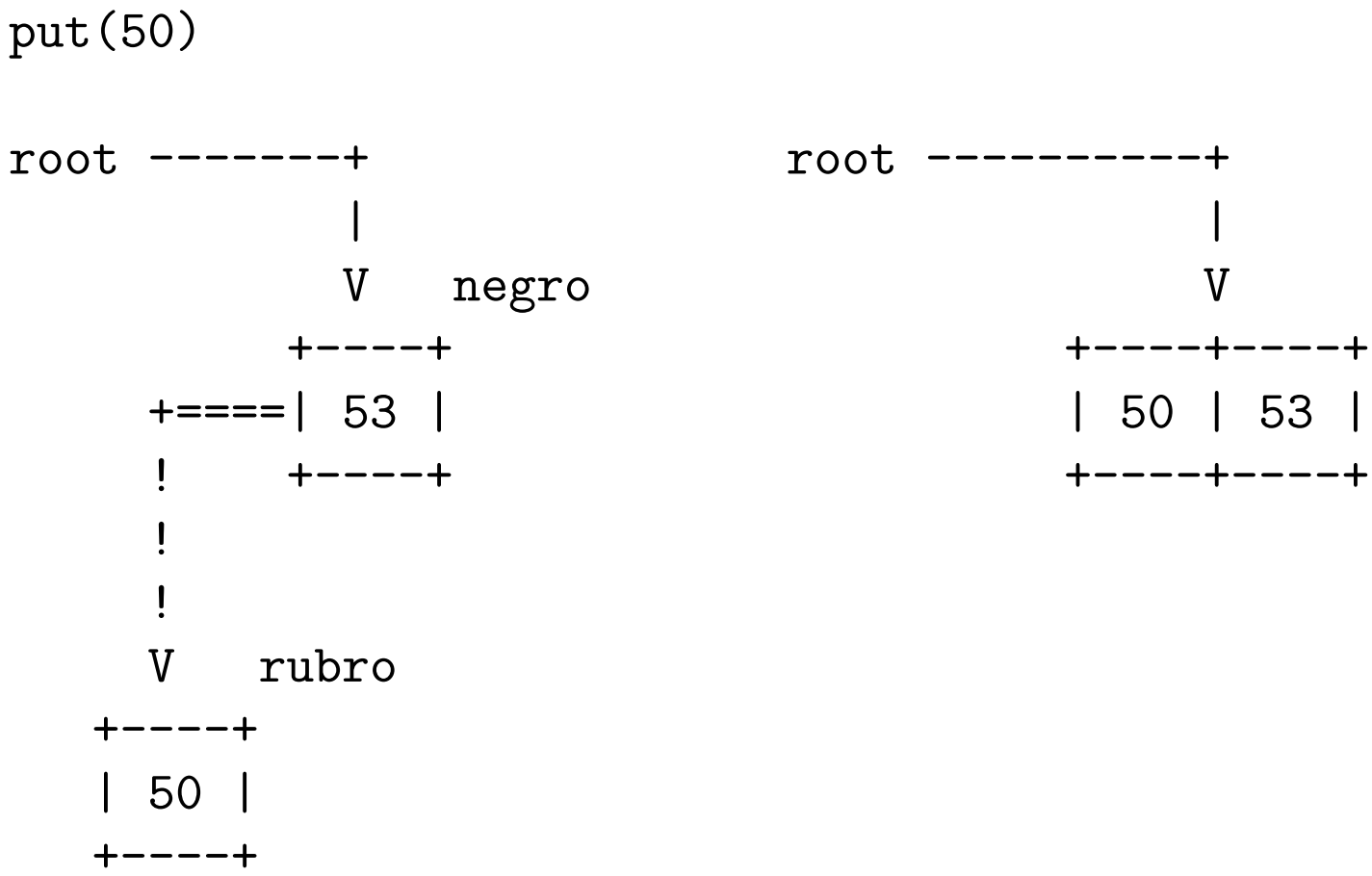
Árvore formada por apenas um 2-nó





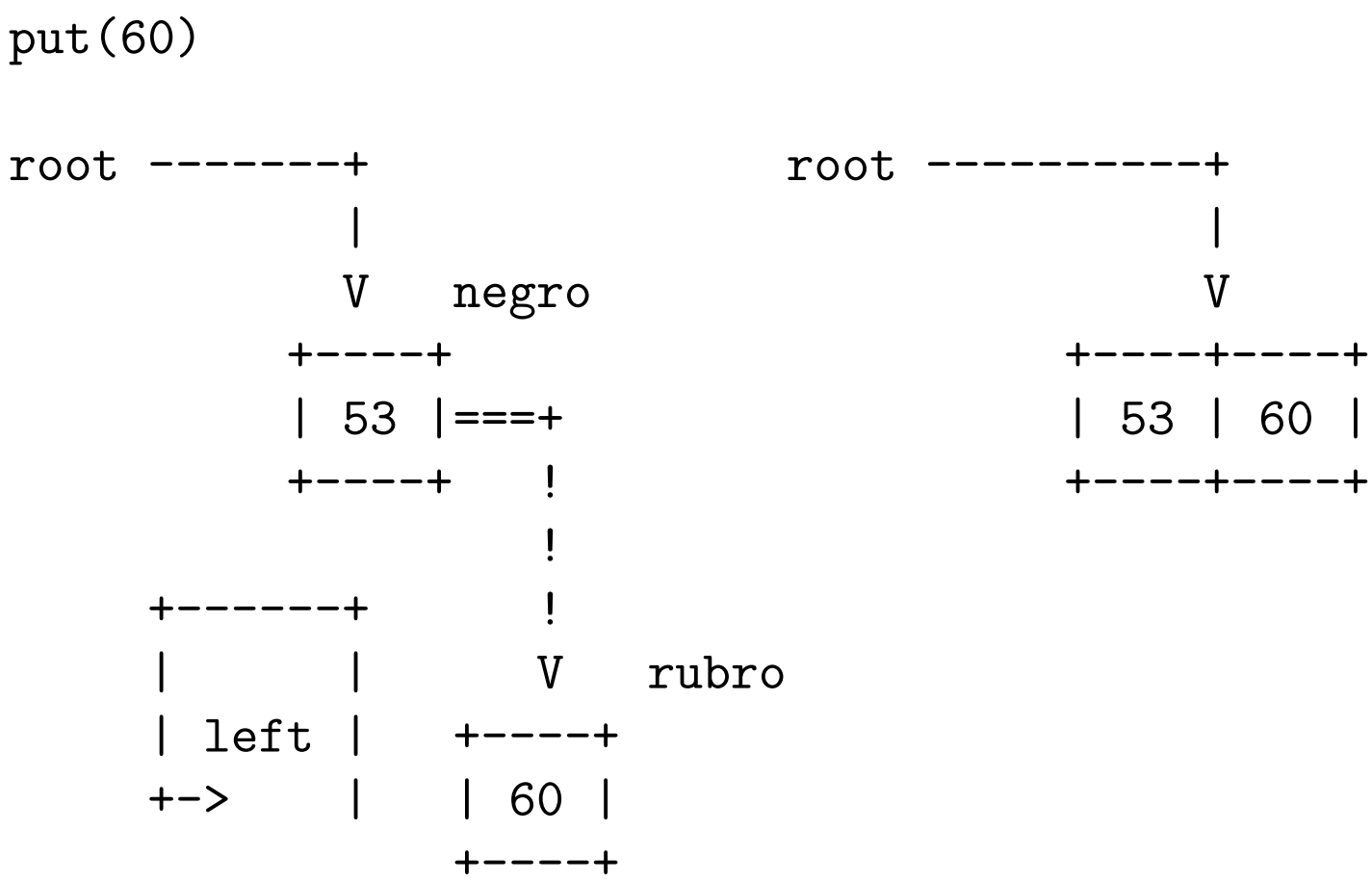
Inserção em um 2-nó

Árvore formada por apenas um 2-nó



Inserção em um 2-nó

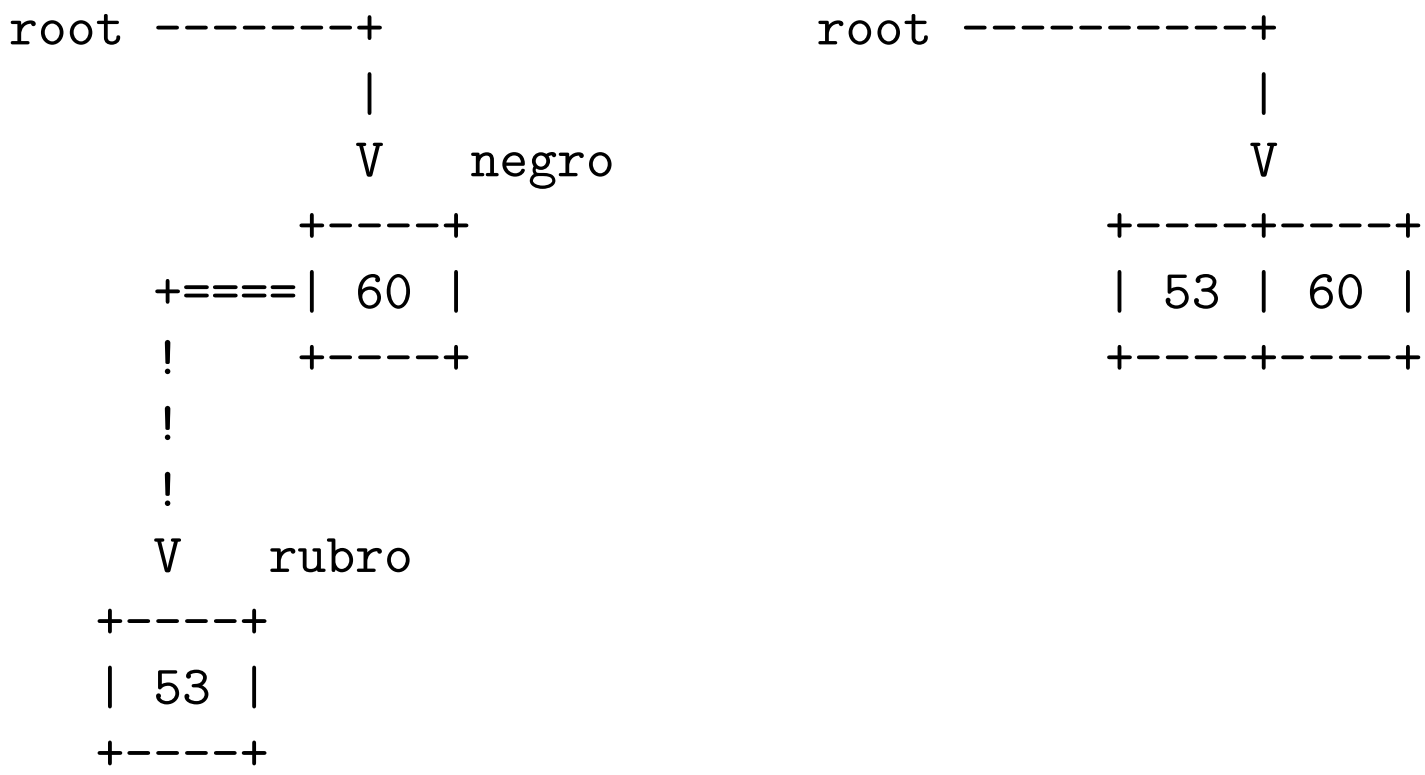
Árvore formada por apenas um 2-nó



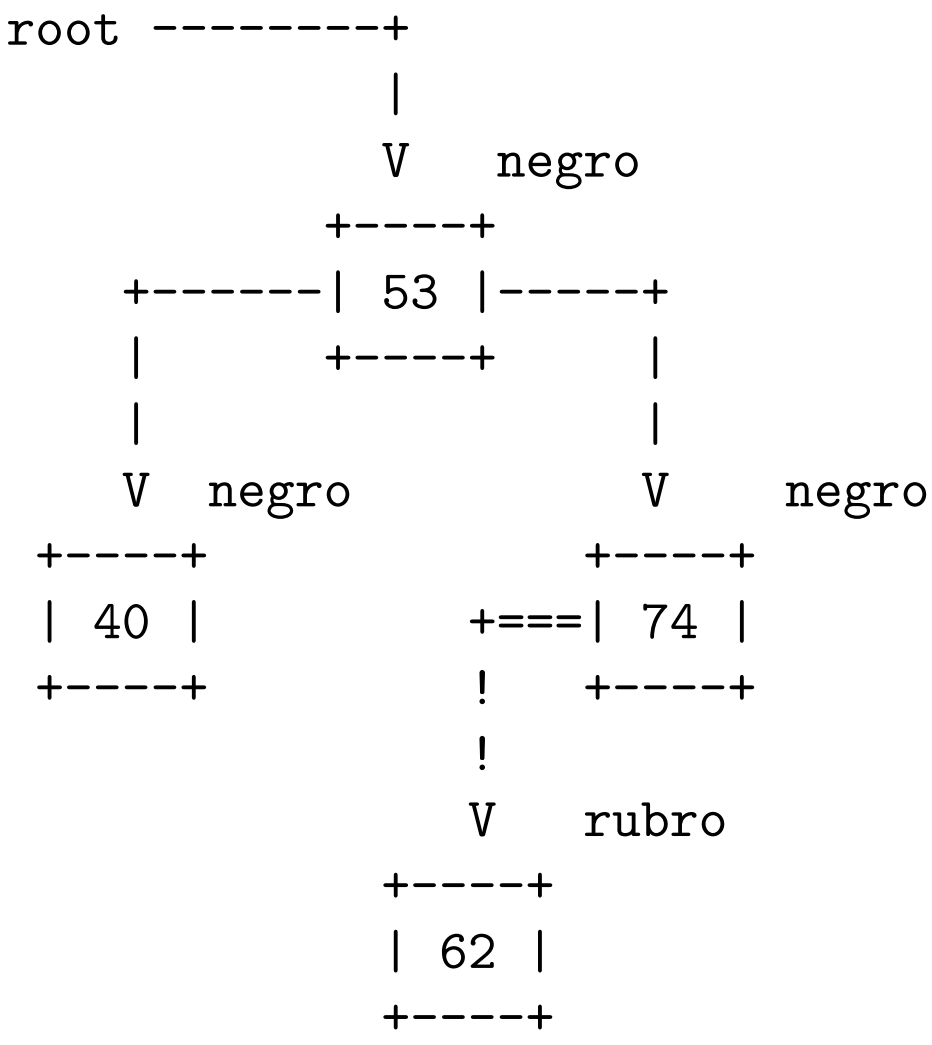
Inserção em um 2-nó

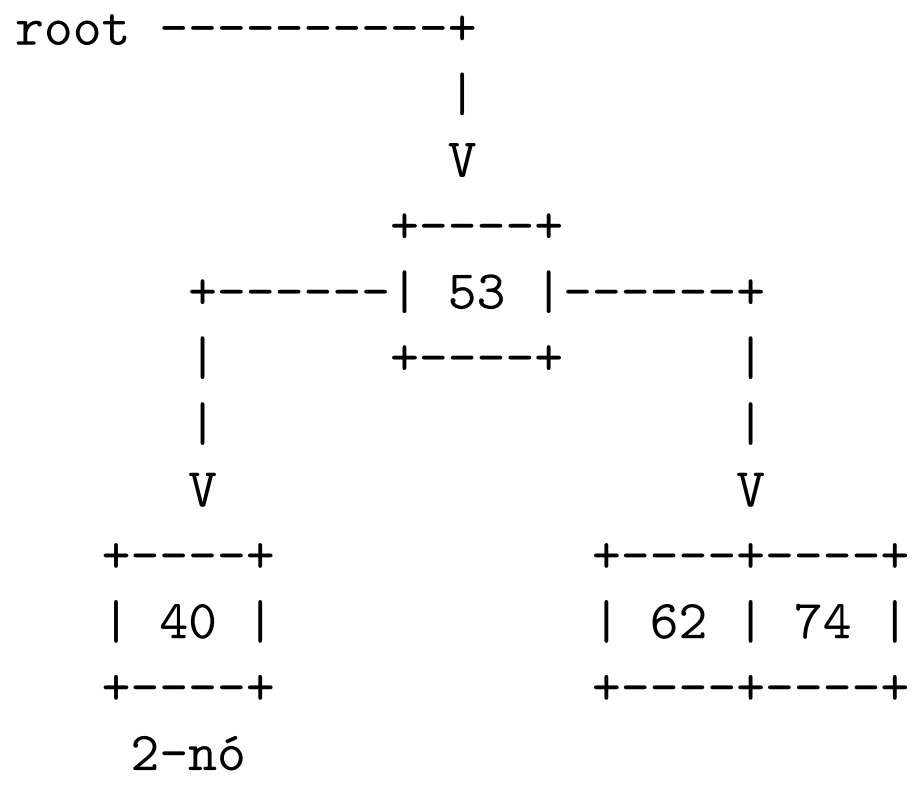
Árvore formada por apenas um 2-nó

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

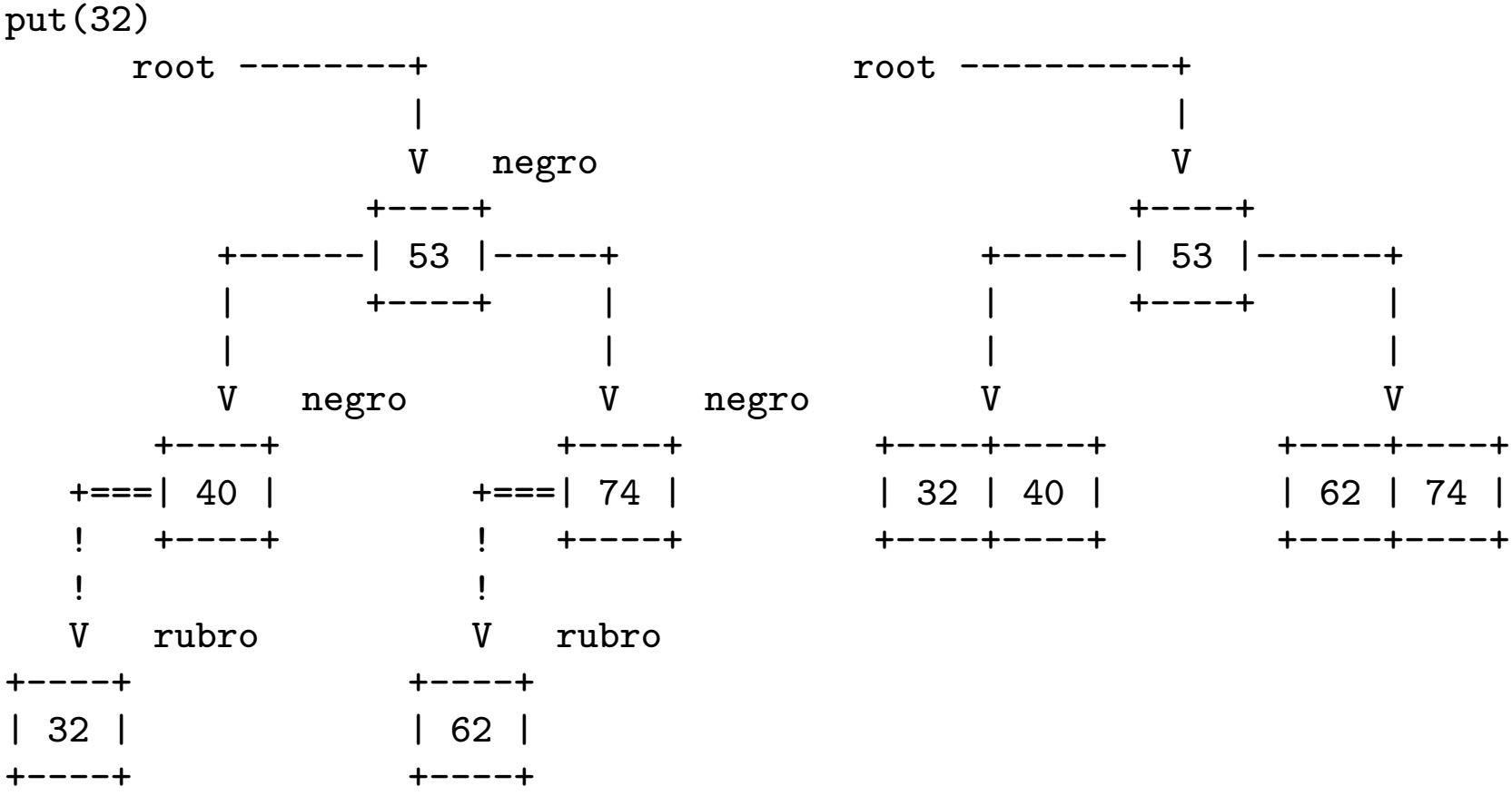


Inserção em um 2-nó qualquer

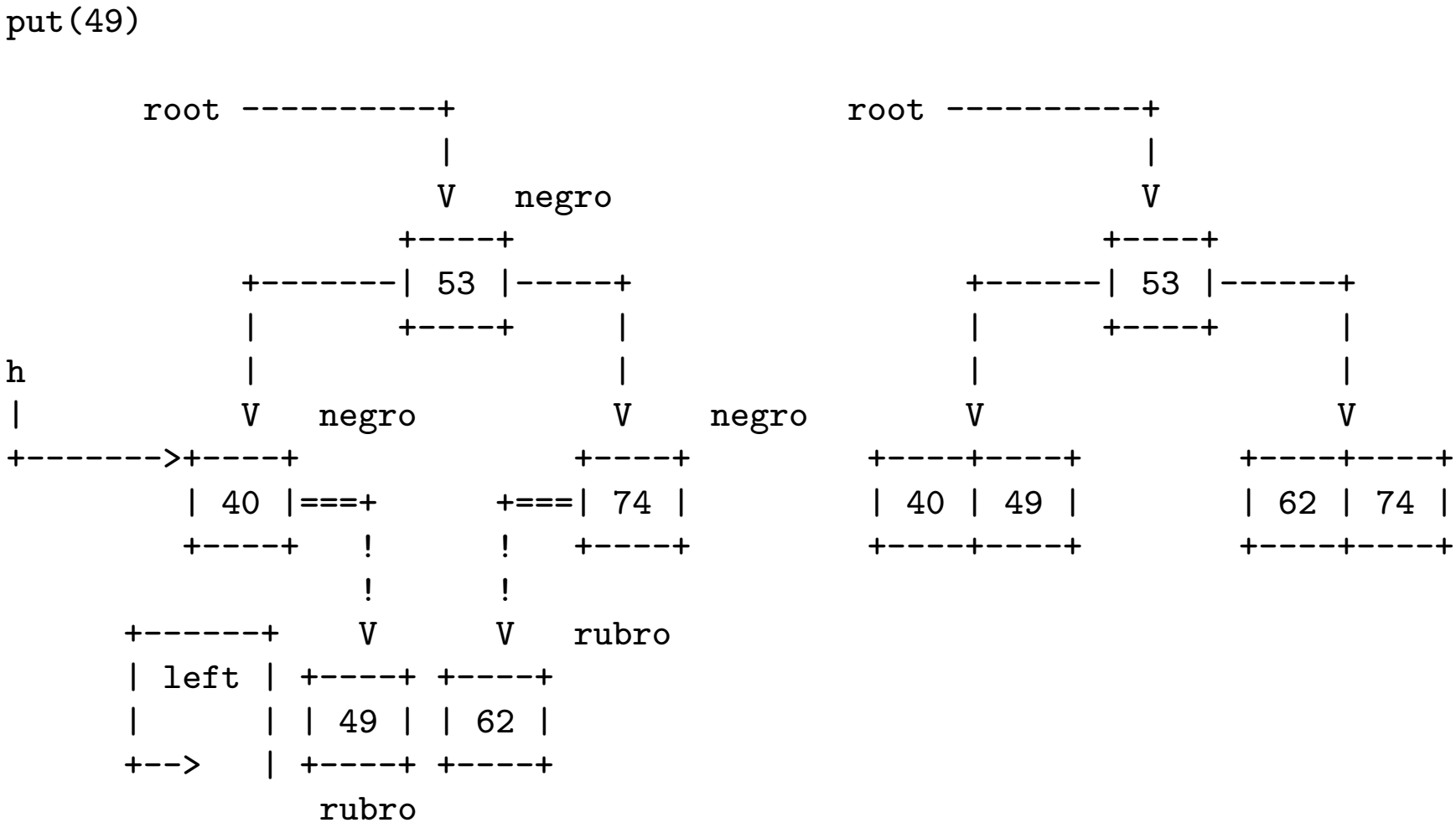




Inserção em um 2-nó qualquer

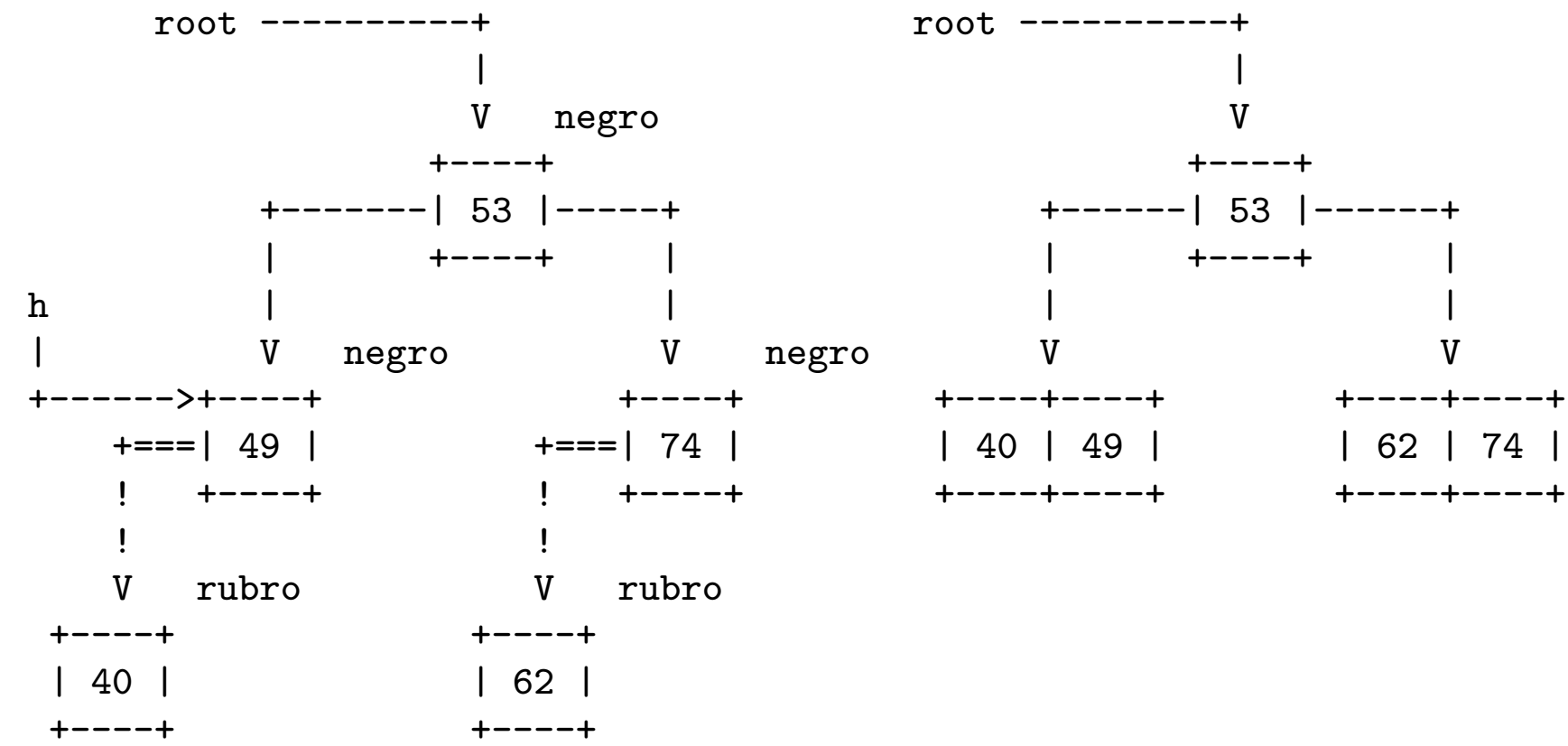


Inserção em um 2-nó qualquer

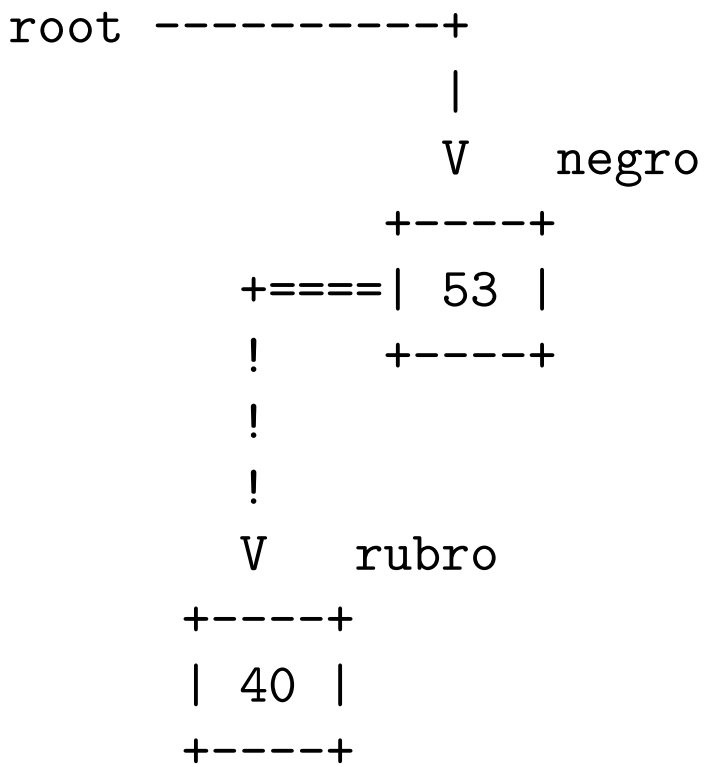


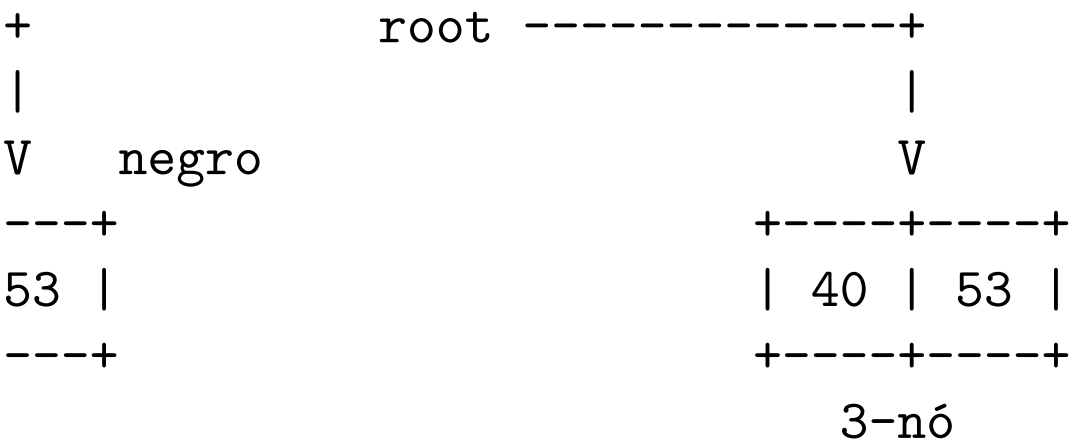
Inserção em um 2-nó qualquer

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

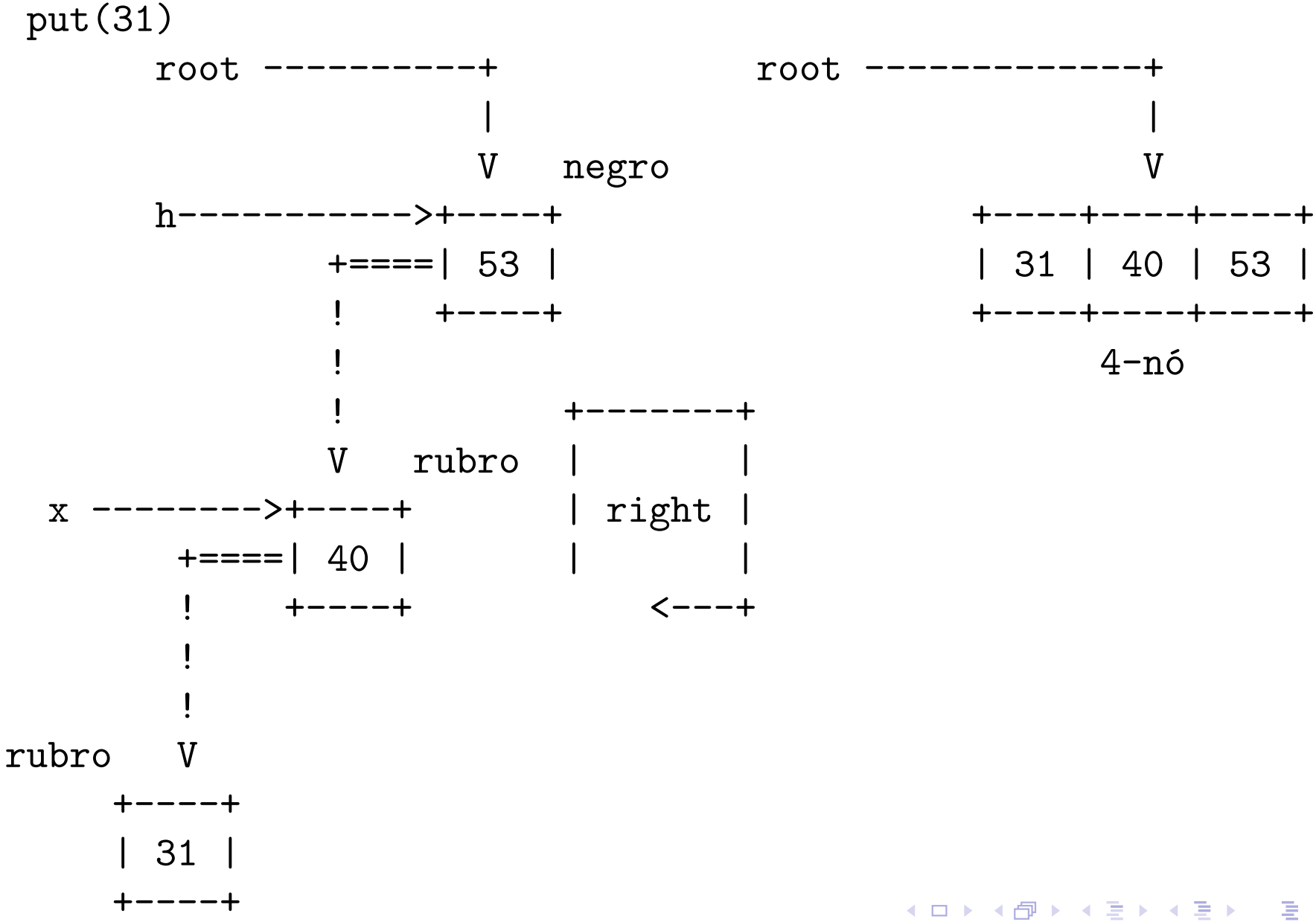


Inserção em um 3-nó



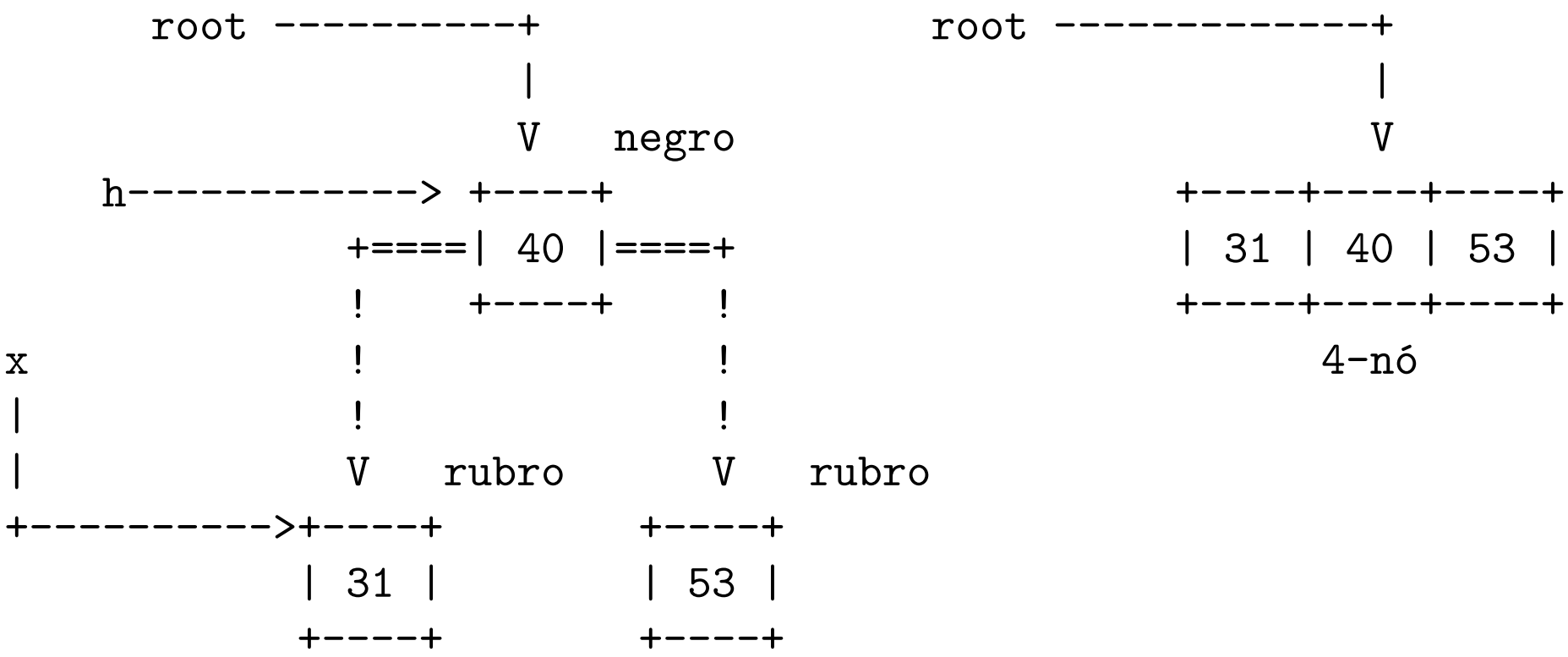


chave é inserida é menor do 3-nó



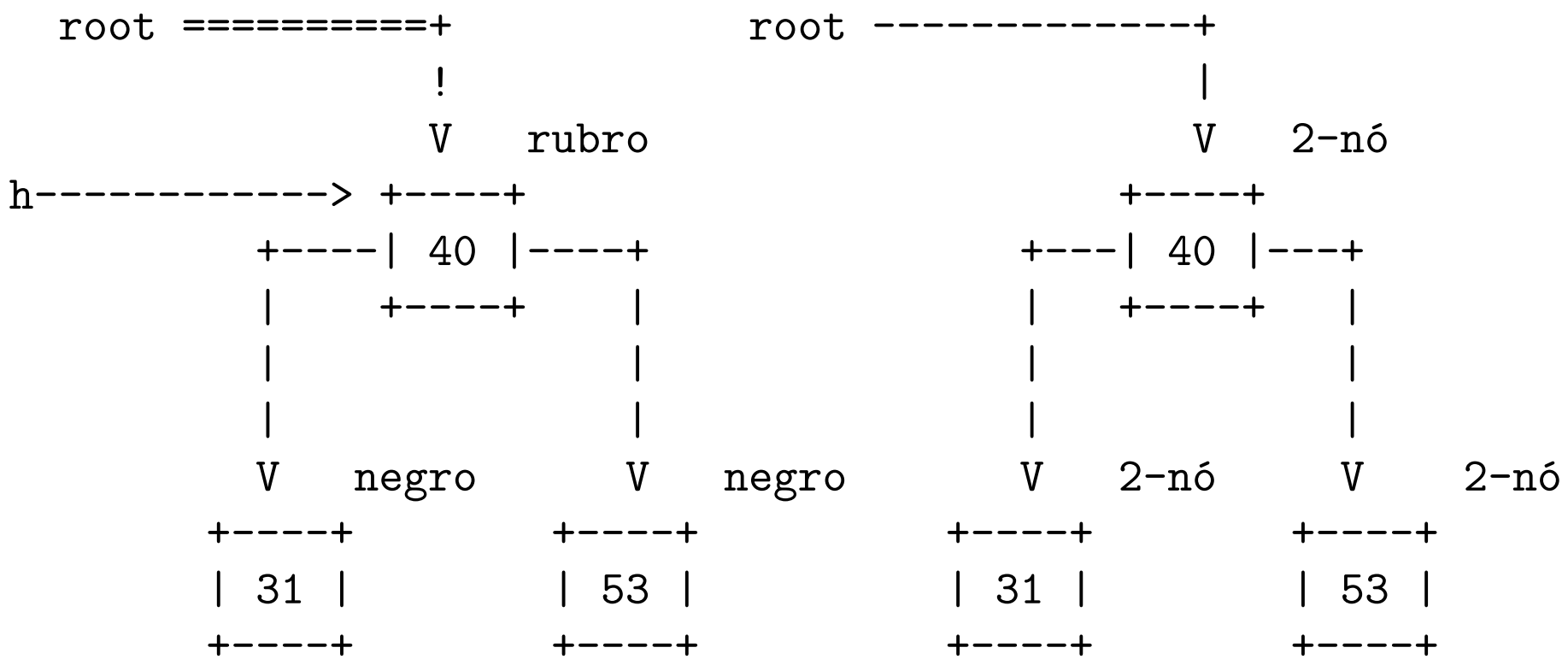
chave é inserida é menor do 3-nó

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

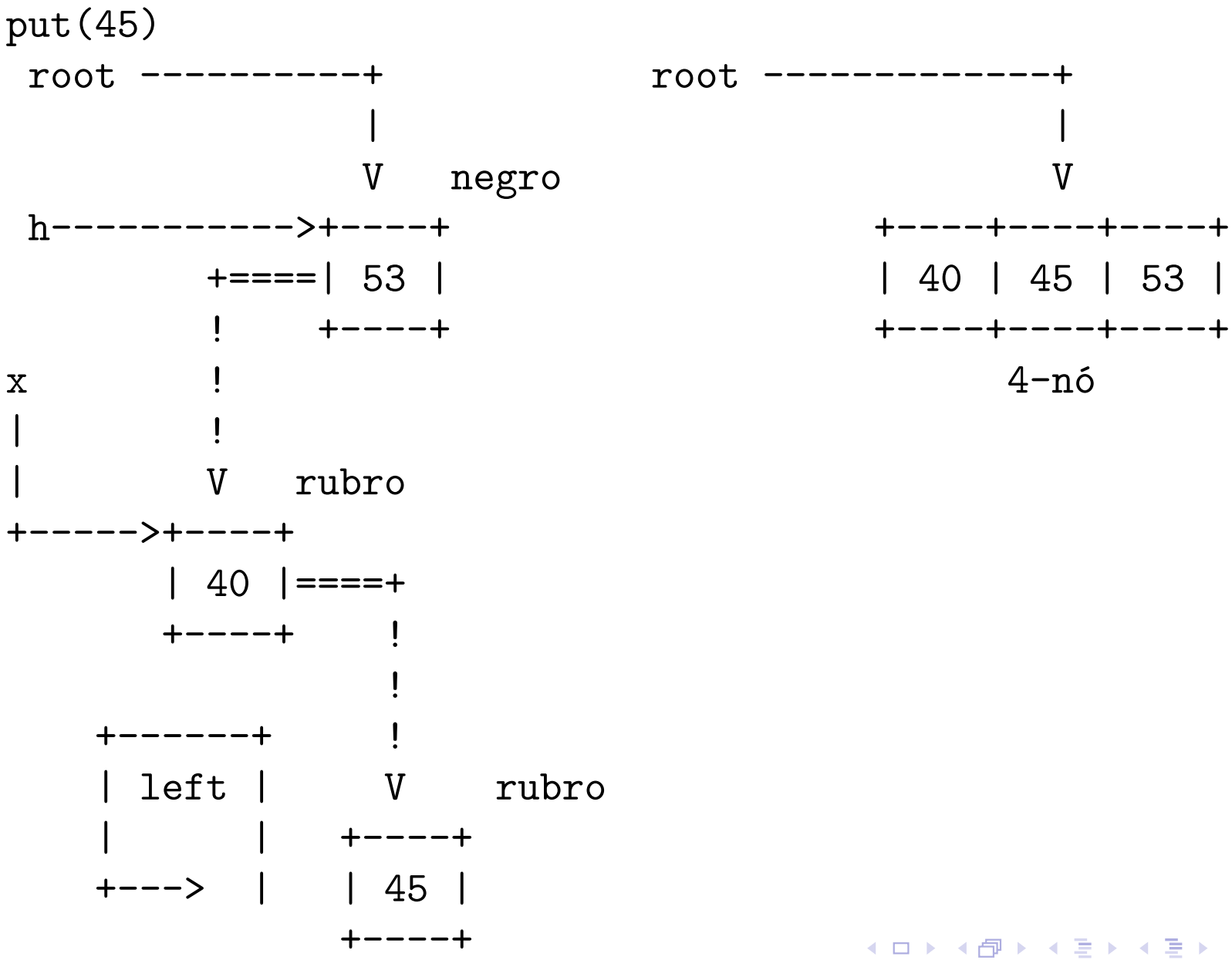


chave é inserida é menor do 3-nó

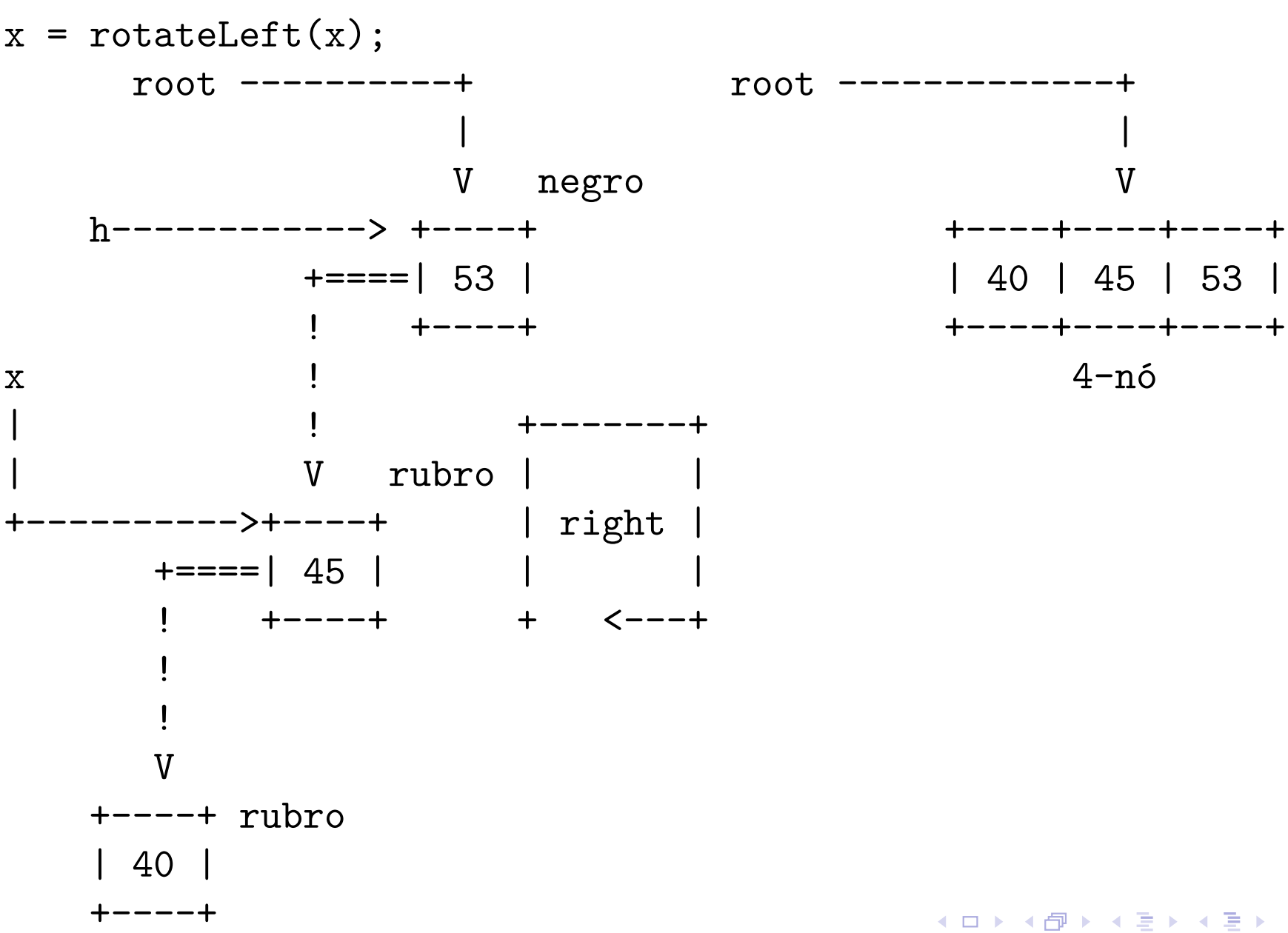
```
flipColors(h);
```



chave é inserida entre as chaves do 3-nó

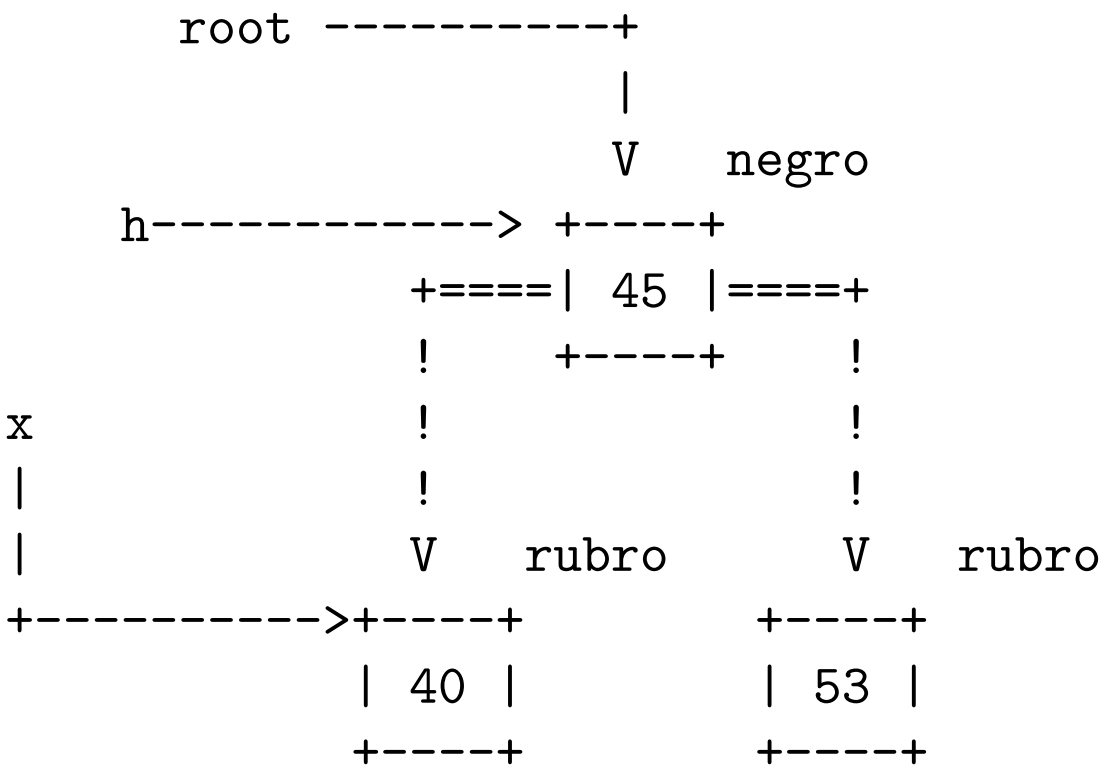


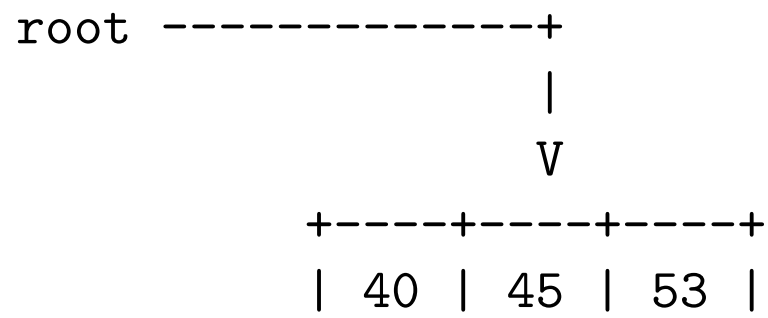
chave é inserida entre as chaves do 3-nó



chave é inserida entre as chaves do 3-nó

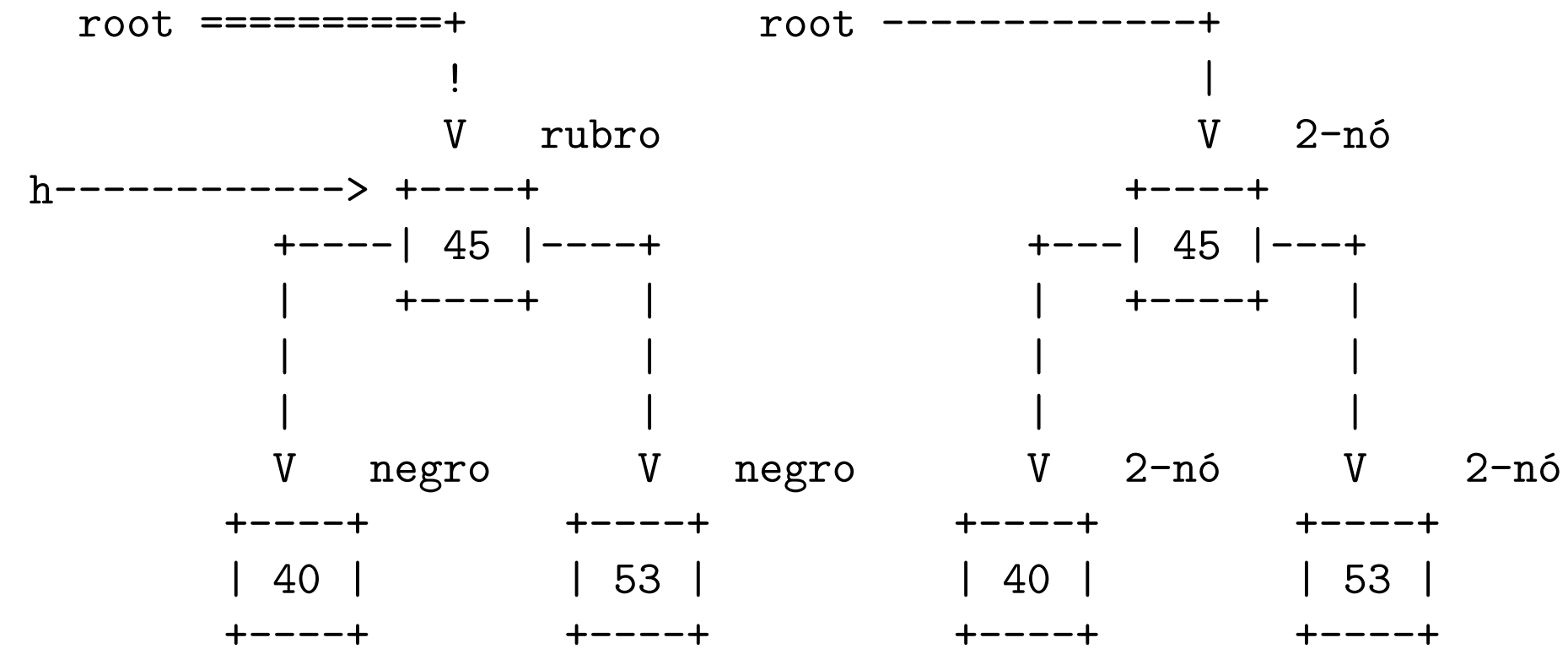
```
h = rotateRight(h);
```



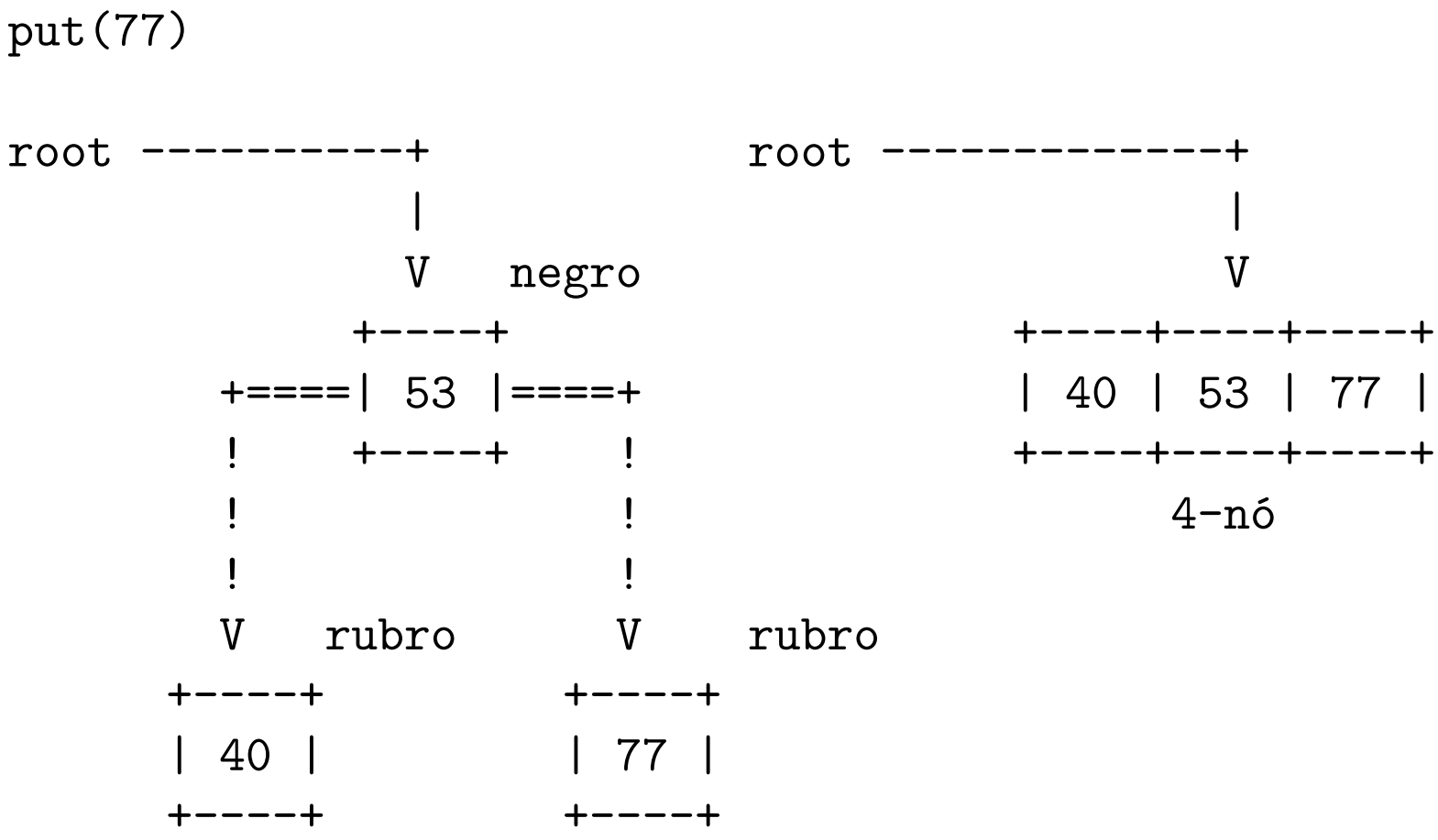


chave é inserida entre as chaves do 3-nó

flipColors(h); hmmm. raiz deve ser negra

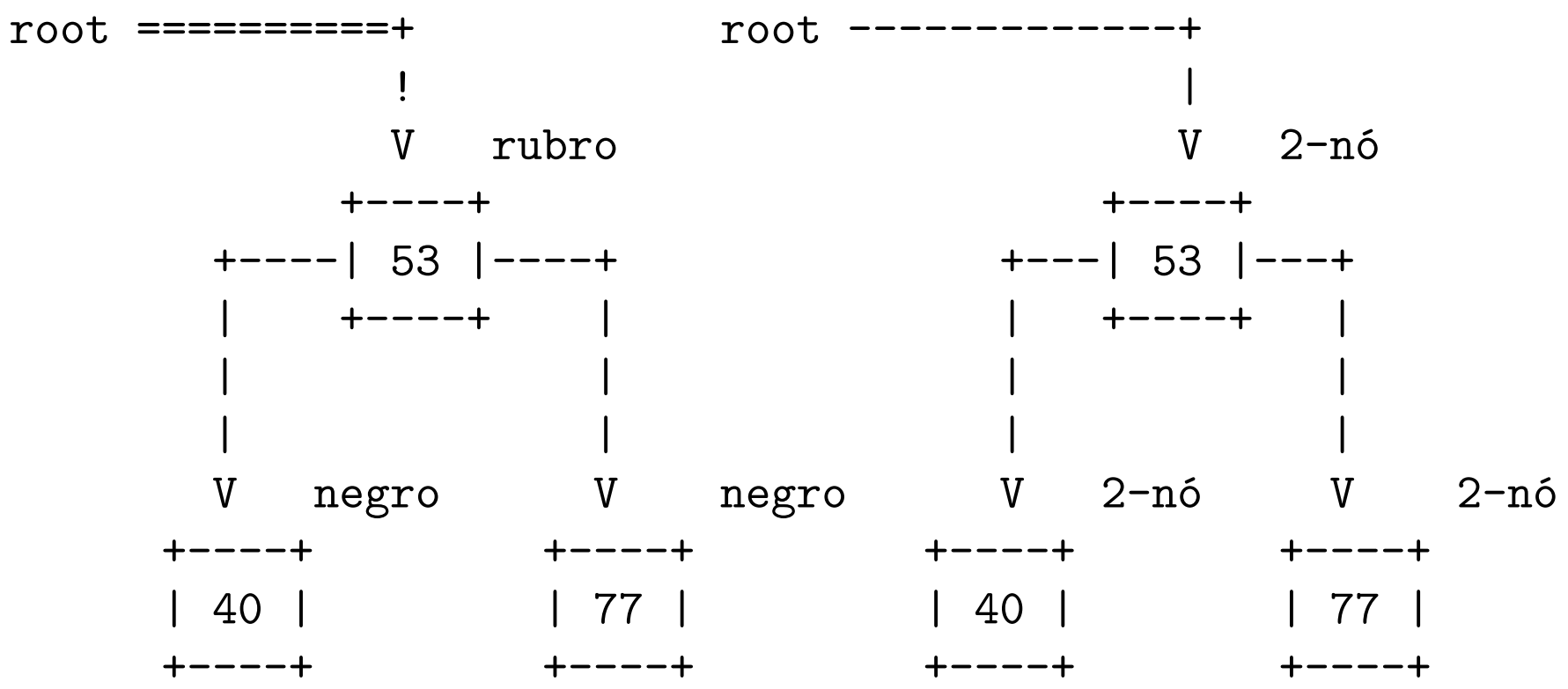


chave inserida é maior que todas do 3-nó

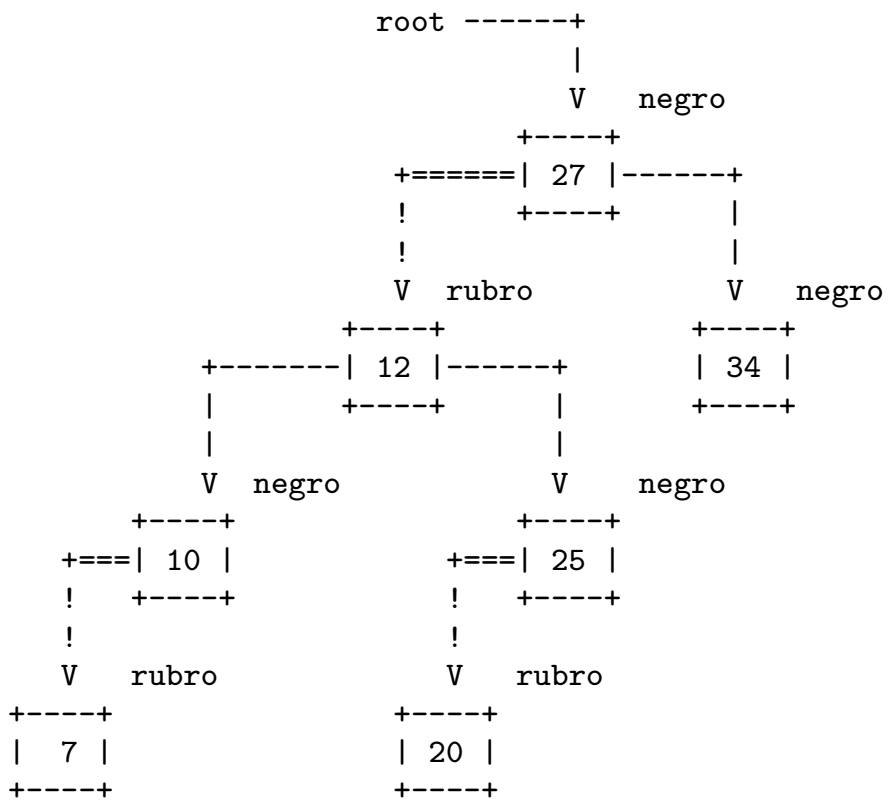


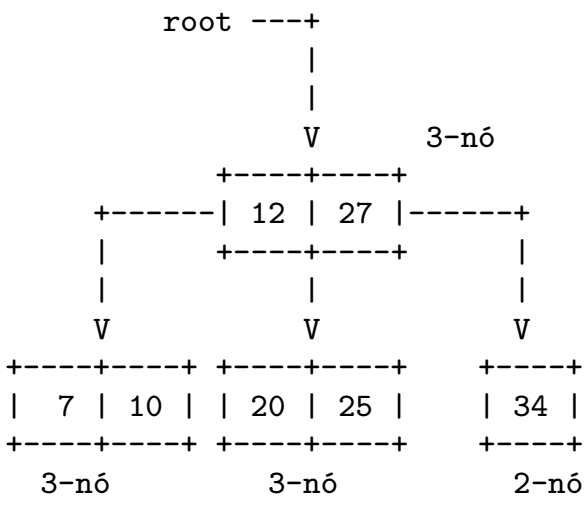
chave inserida é maior que todas do 3-nó

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

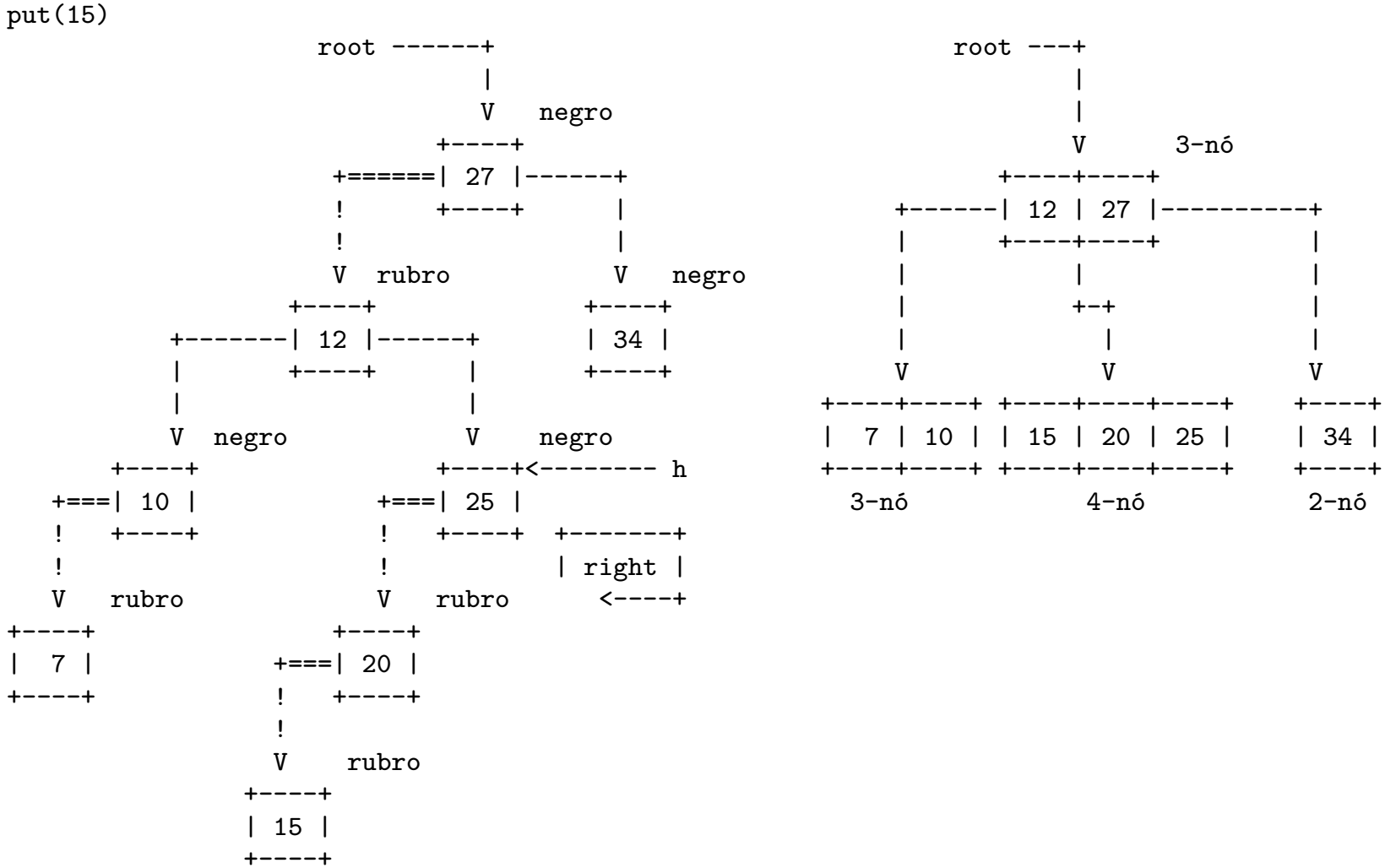


chave é inserida em um 3-nó qualquer

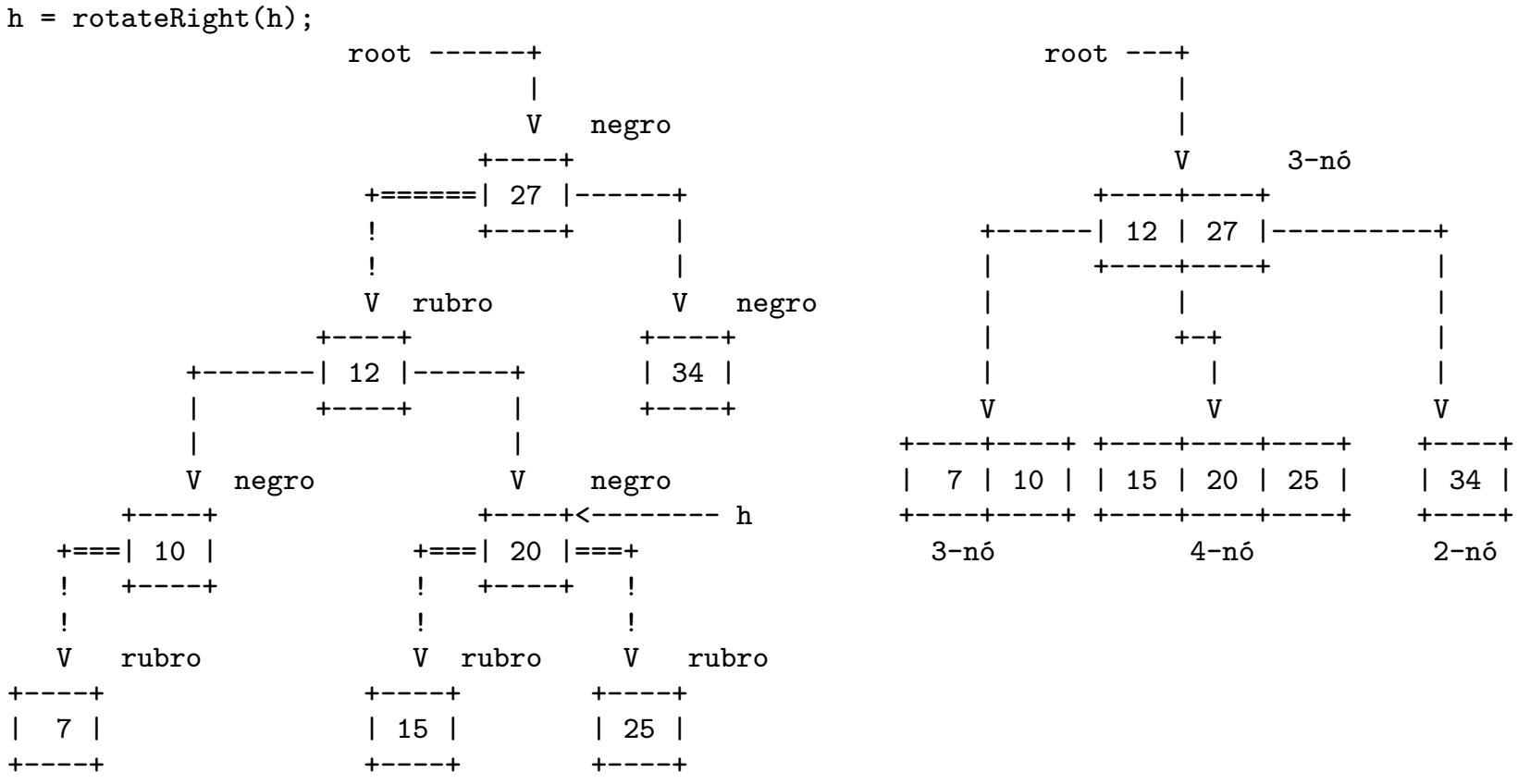




chave é inserida em um 3-nó qualquer

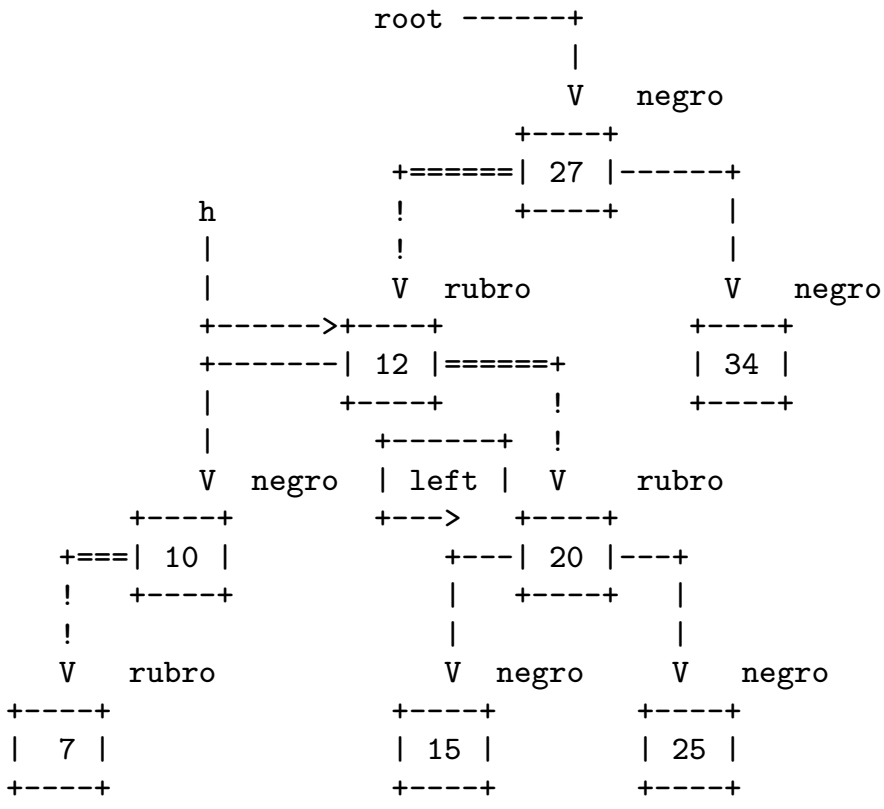


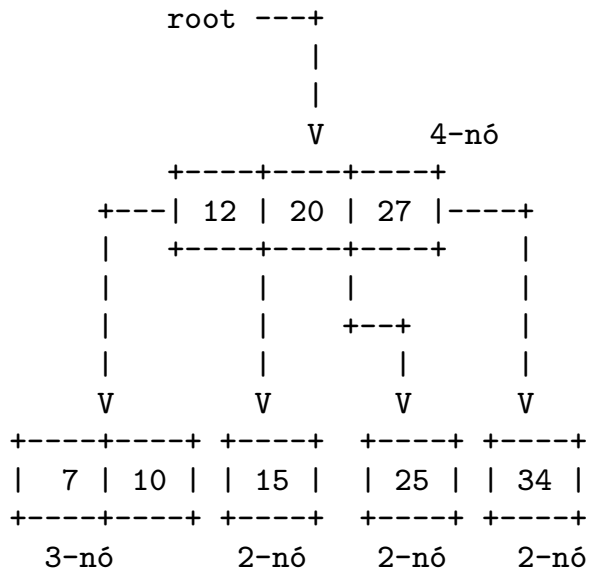
chave é inserida em um 3-nó qualquer



chave é inserida em um 3-nó qualquer

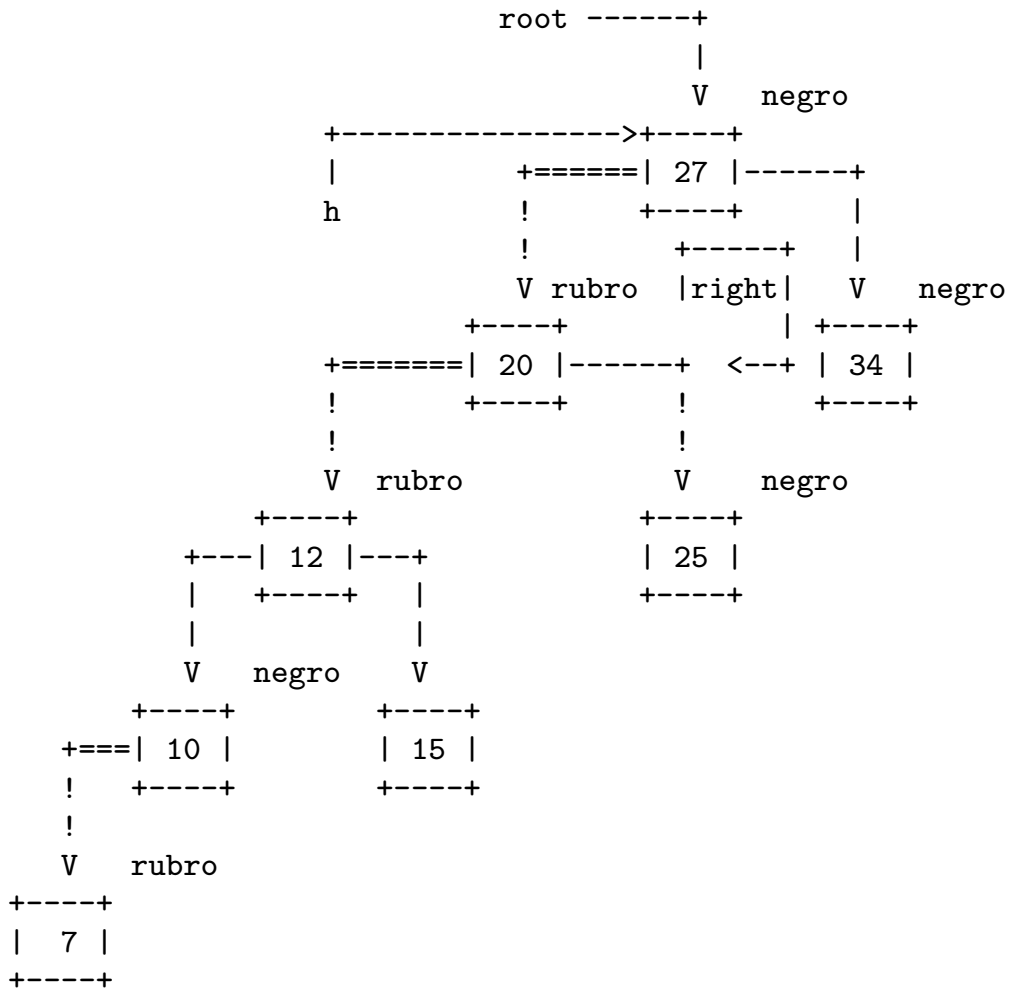
```
flipColors(h);
```

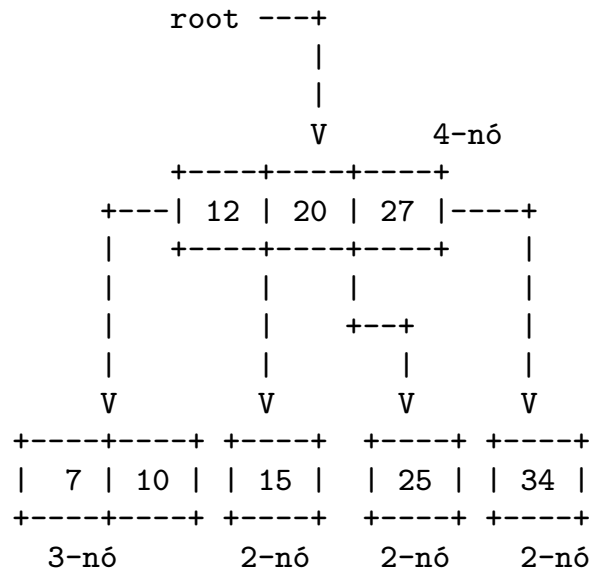




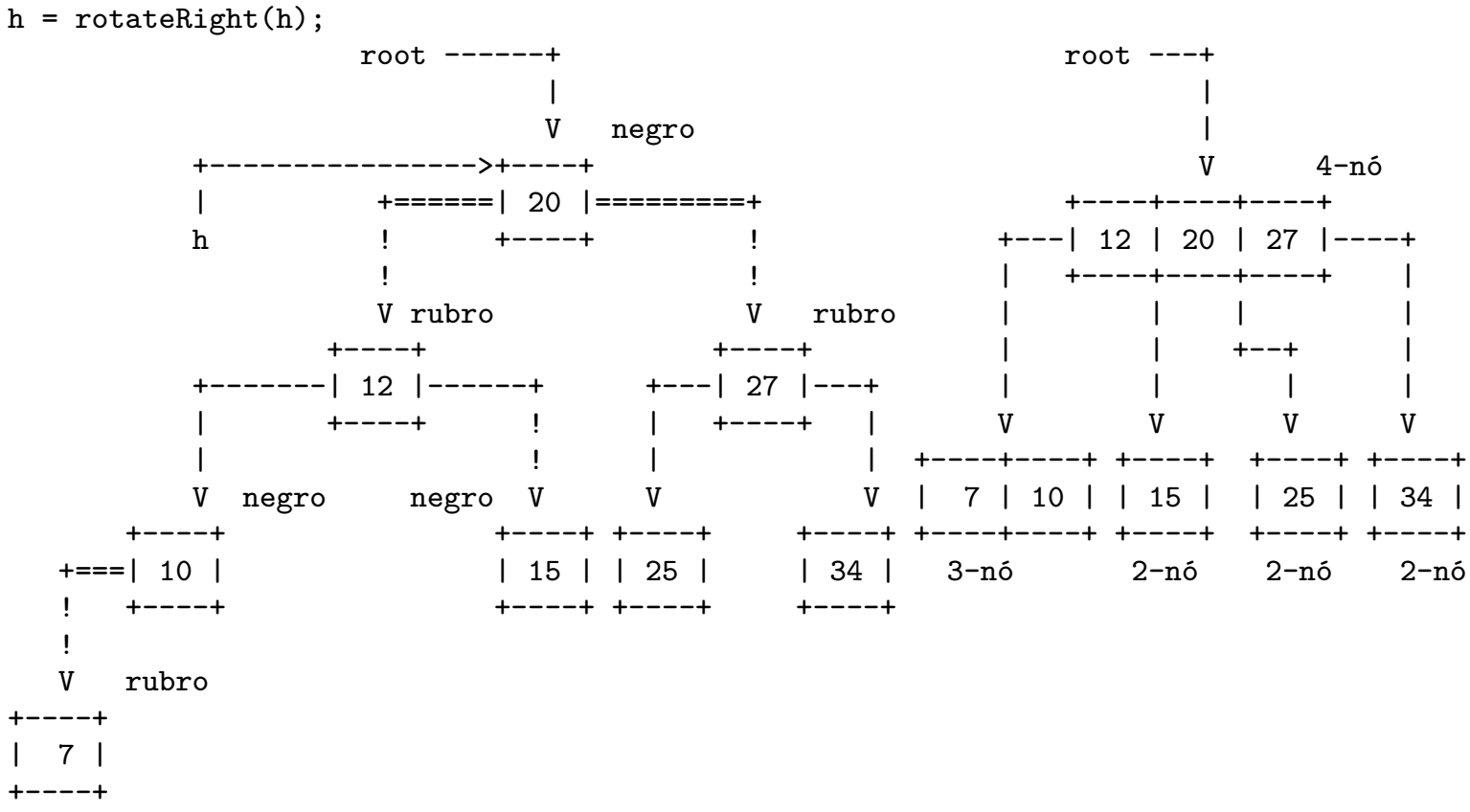
chave é inserida em um 3-nó qualquer

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





chave é inserida em um 3-nó qualquer



chave é inserida em um 3-nó qualquer

```
flipColors(h);
                    root ======+                                  root ----+
                               !                                           |
                               V   rubro                                   V
          +------------------->+----+                                  +----+
          |          +---------| 20 |------+                        +---| 20 |----+
          h          |         +----+      |                        |   +----+    |
                     |                     |                        |             |
                     V negro               V   negro                V             V
                   +----+                  +----+                   +----+        +----+
          +--------| 12 |------+      +---| 27 |---+       +--| 12 |-+    +--| 27 |--+
          |        +----+      !      |   +----+   |       |  +----+ |    |  +----+  |
          |                    !      |            |       |         |    |          |
          V  negro     negro  V      V negro negroV       V         V    V          V
       +----+                 +----+ +----+     +----+ +----+----+ +----+ +----+   +----+
   +===| 10 |                 | 15 | | 25 |     | 34 | |  7 | 10 | | 15 | | 25 |   | 34 |
   !   +----+                 +----+ +----+     +----+ +----+----+ +----+ +----+   +----+
   !
   V   rubro
+----+
|  7 |
+----+
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*.

Rotações

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

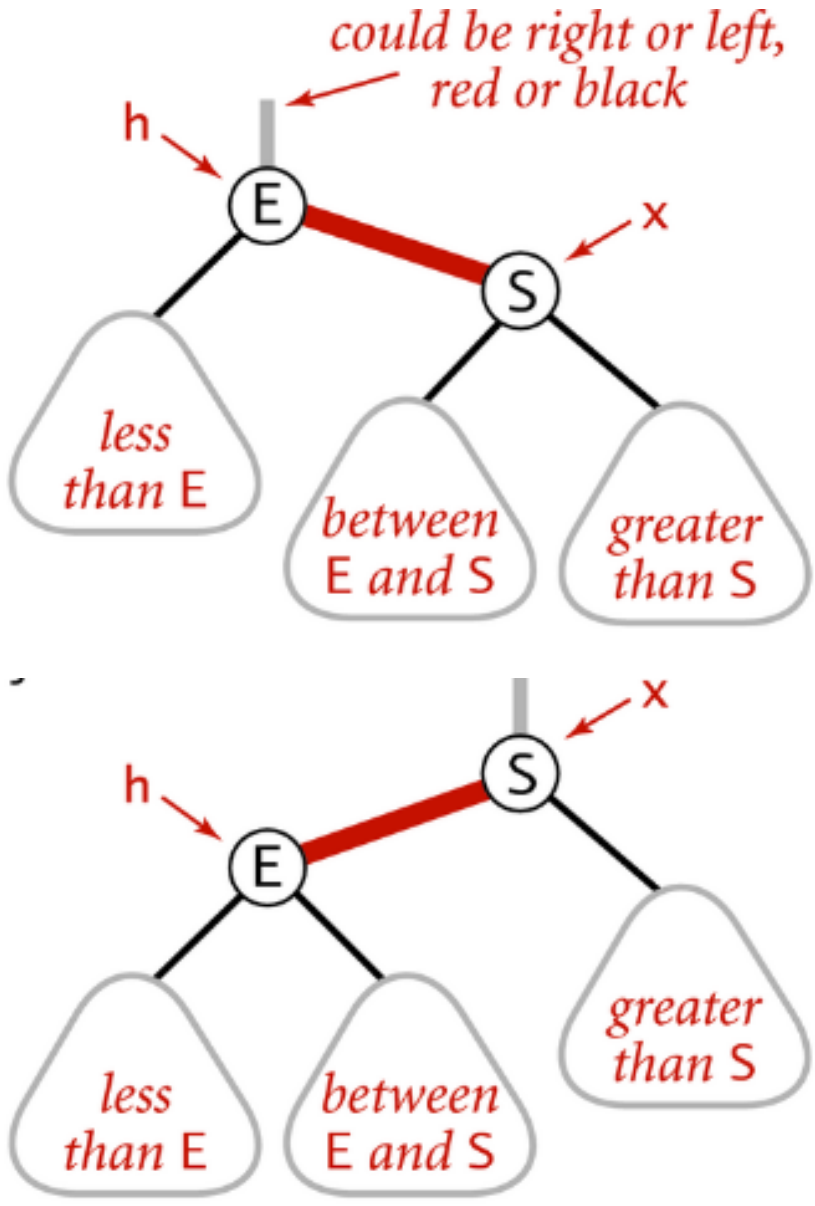
Durante uma operação de inserção, podemos ter, temporariamente, um link rubro inclinado para a direita ou dois links rubros incidindo no mesmo nó.

Para corrigir isso, usamos rotações e *flipping colors*.

Rotação esquerda (ou anti-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



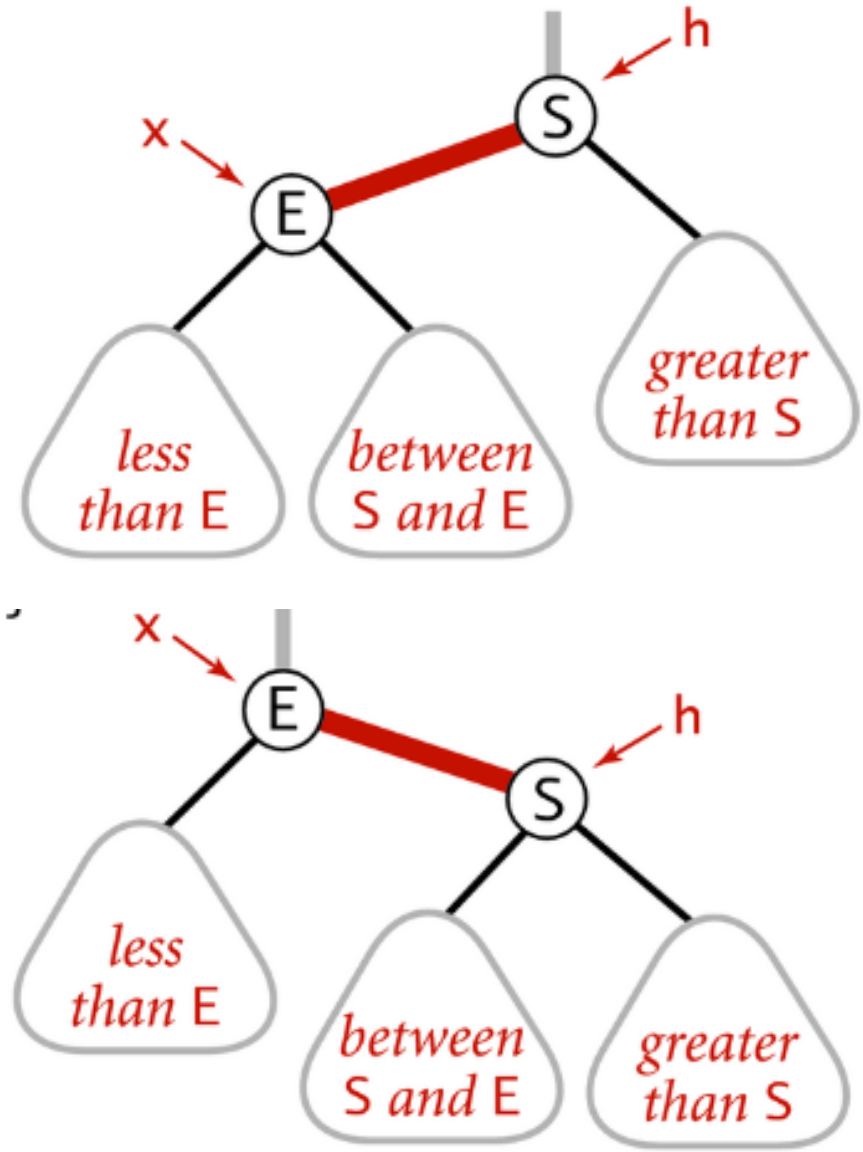
Left rotate (right link of h)

Fonte: algs4

Rotação esquerda

```
static 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

```
static 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**.

Flipping colors

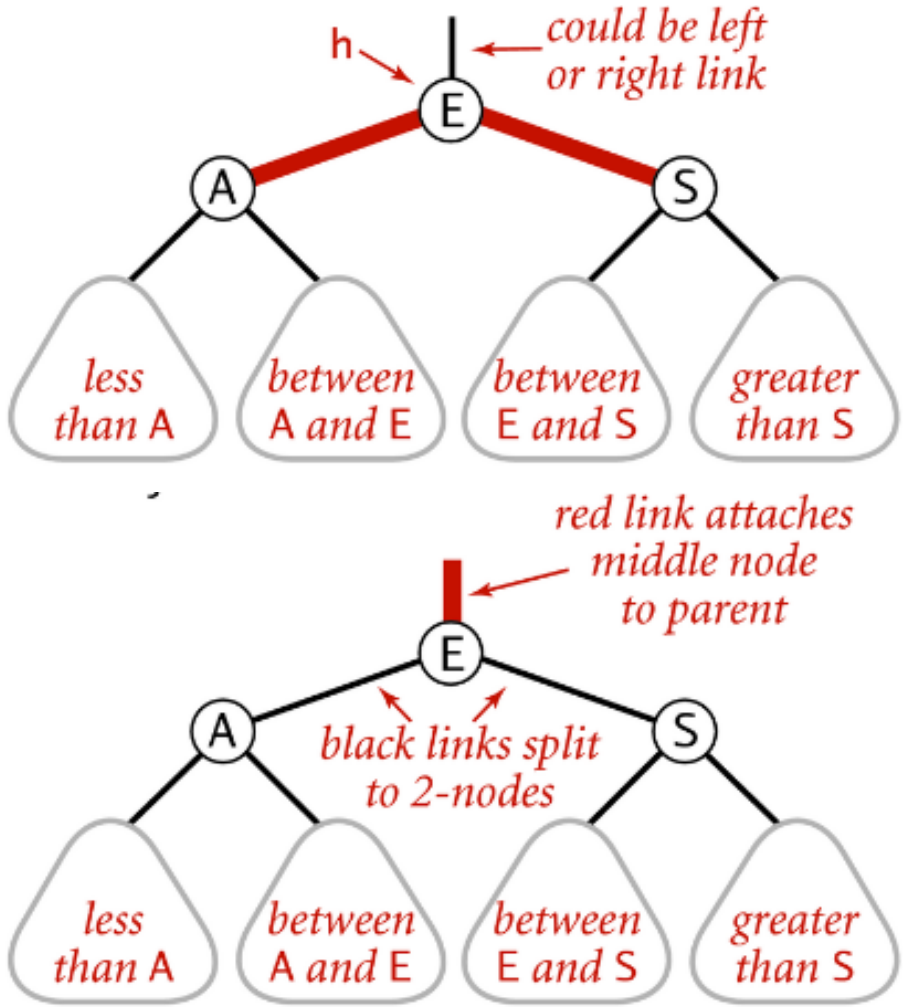
As operações de rotação são locais.

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

Mas a operação pode ter criado um link rubro inclinado para o 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.

```
static void flipColors(Node h) {
   h->color = RED;
   h->left->color  = BLACK;
   h->right->color = BLACK;
}
```

RedBlackBST: put()

```
void put(Key key, Value val) {
   r = putTree(r, key, val);
   r->color = BLACK;
}
```

RedBlackBST: put()

```
static
Node putTree(Node h, Key key, Value val) {
   if (h == NULL)
      return newNode(key, val, 1, RED);
   int cmp = compare(key, h->key);
   if (cmp < 0)
      h->left  = putTree(h->left, key, val);
   else if (cmp > 0)
      h->right = putTree(h->right, key, val);
   else h->val = val;
   h = balance(h);
   return h;
}
```

RedBlackBST: put()

Verifica invariante rubro-negro
quando estamos voltando da recursão.

```
static 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);
   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)

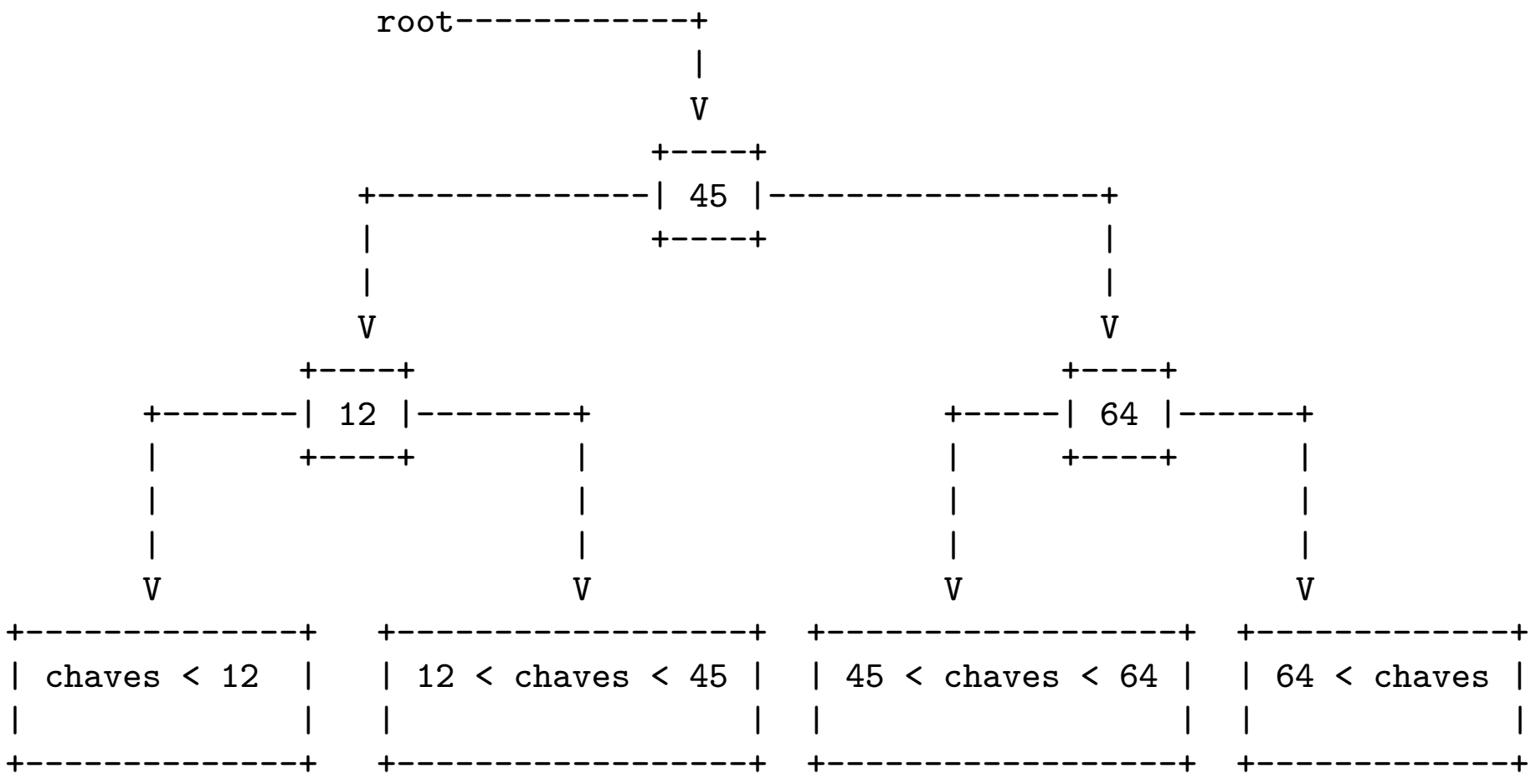
Remoção em árvore 2-3

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

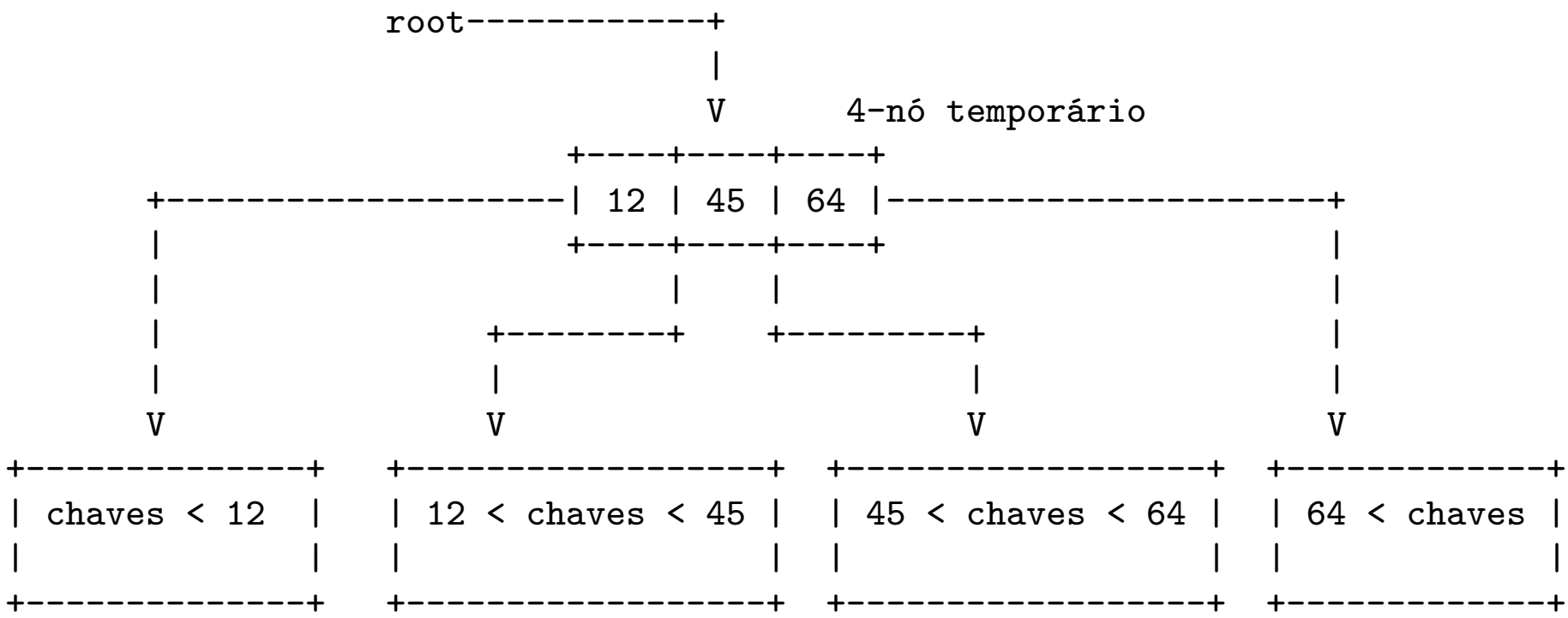
> *o nó sendo examinado é um 3-nó*
> *ou um 4-nó (temporário)*

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

Os dois filhos da raiz são 2-nós

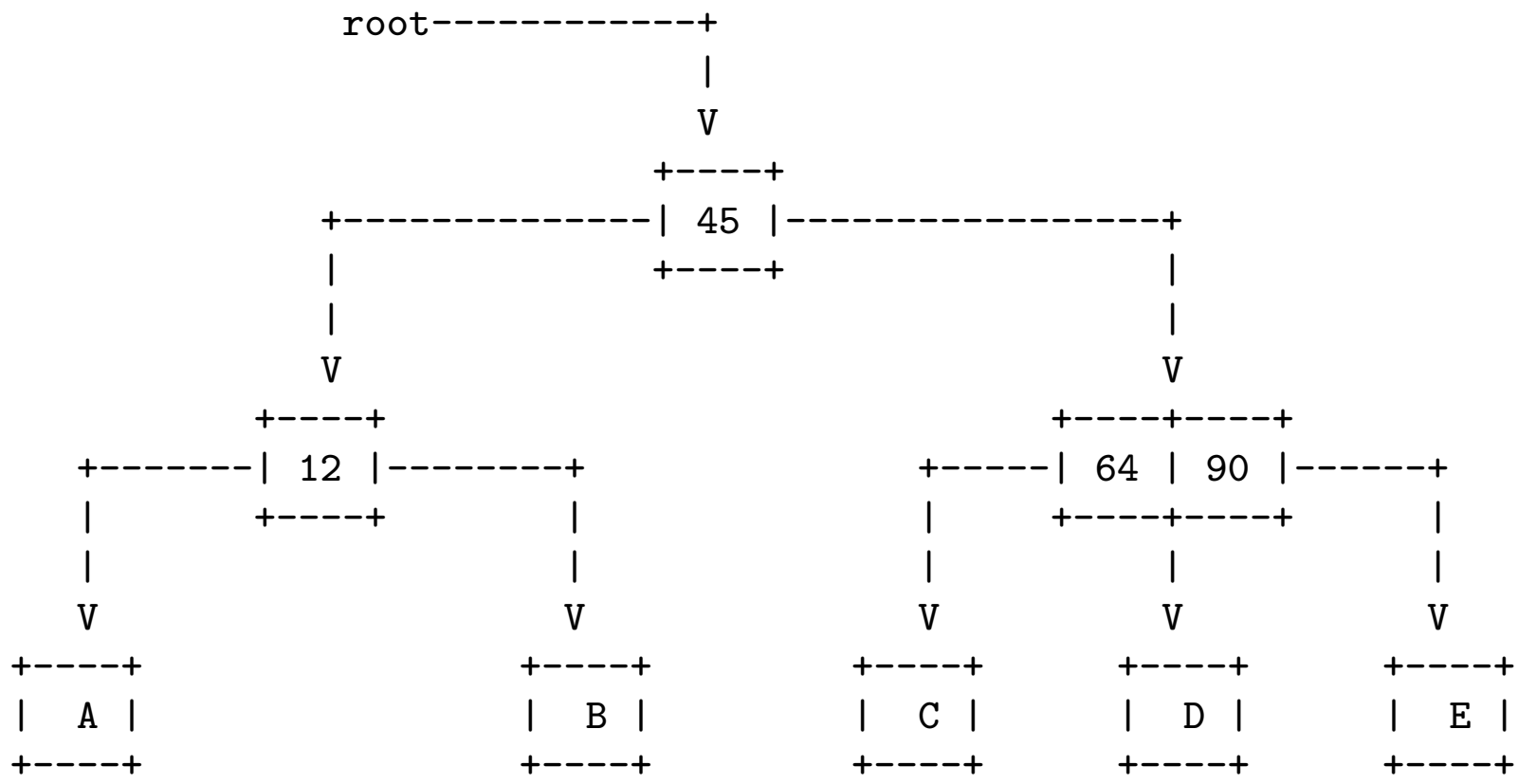


Os dois filhos da raiz são 2-nós

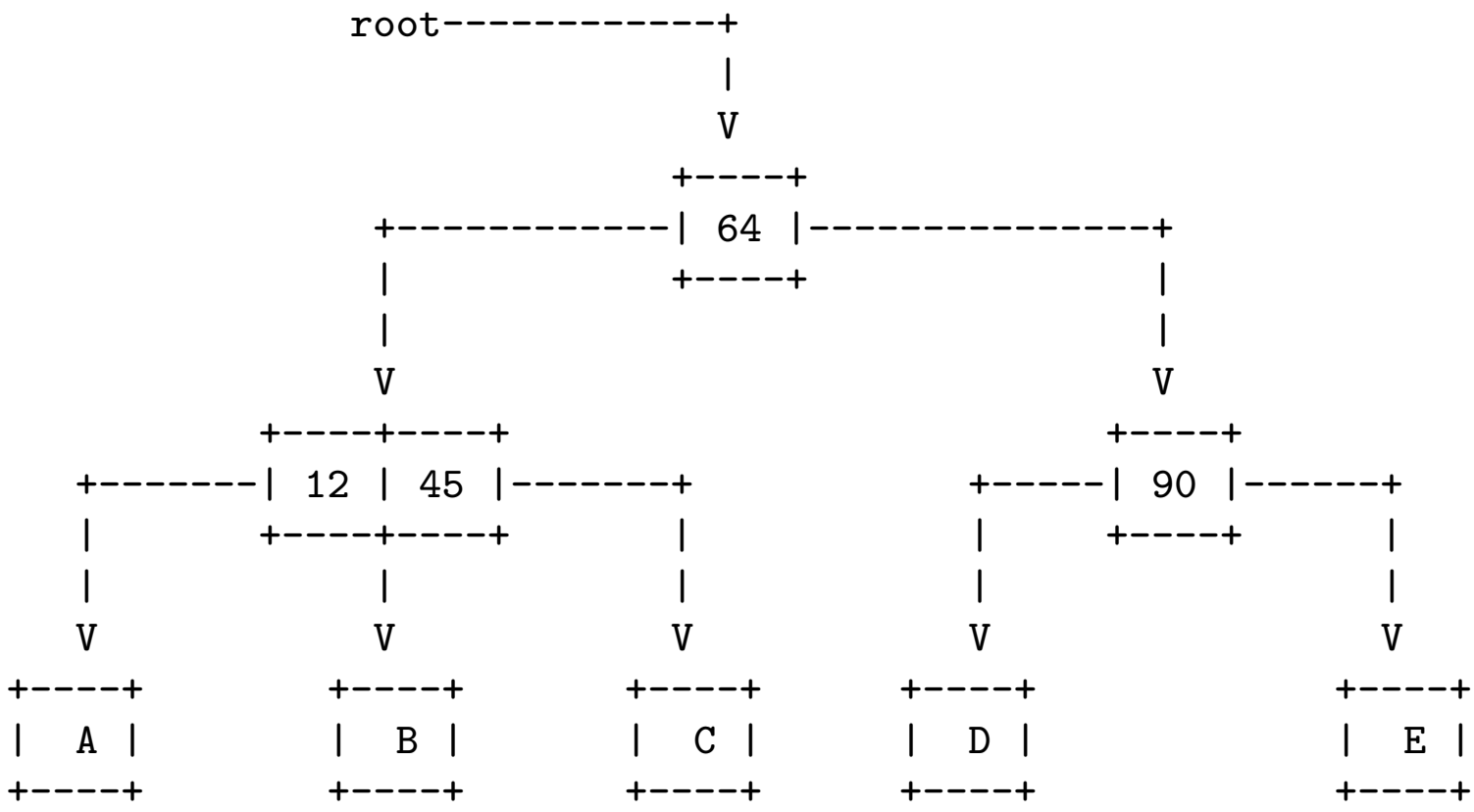


Agora passemos a raiz quando apenas o nó esquerdo é um 2-nó.

Filho esquerdo da raiz é 2-nó



... vira filho direito da raiz é 2-nó



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.

Mover para esquerda

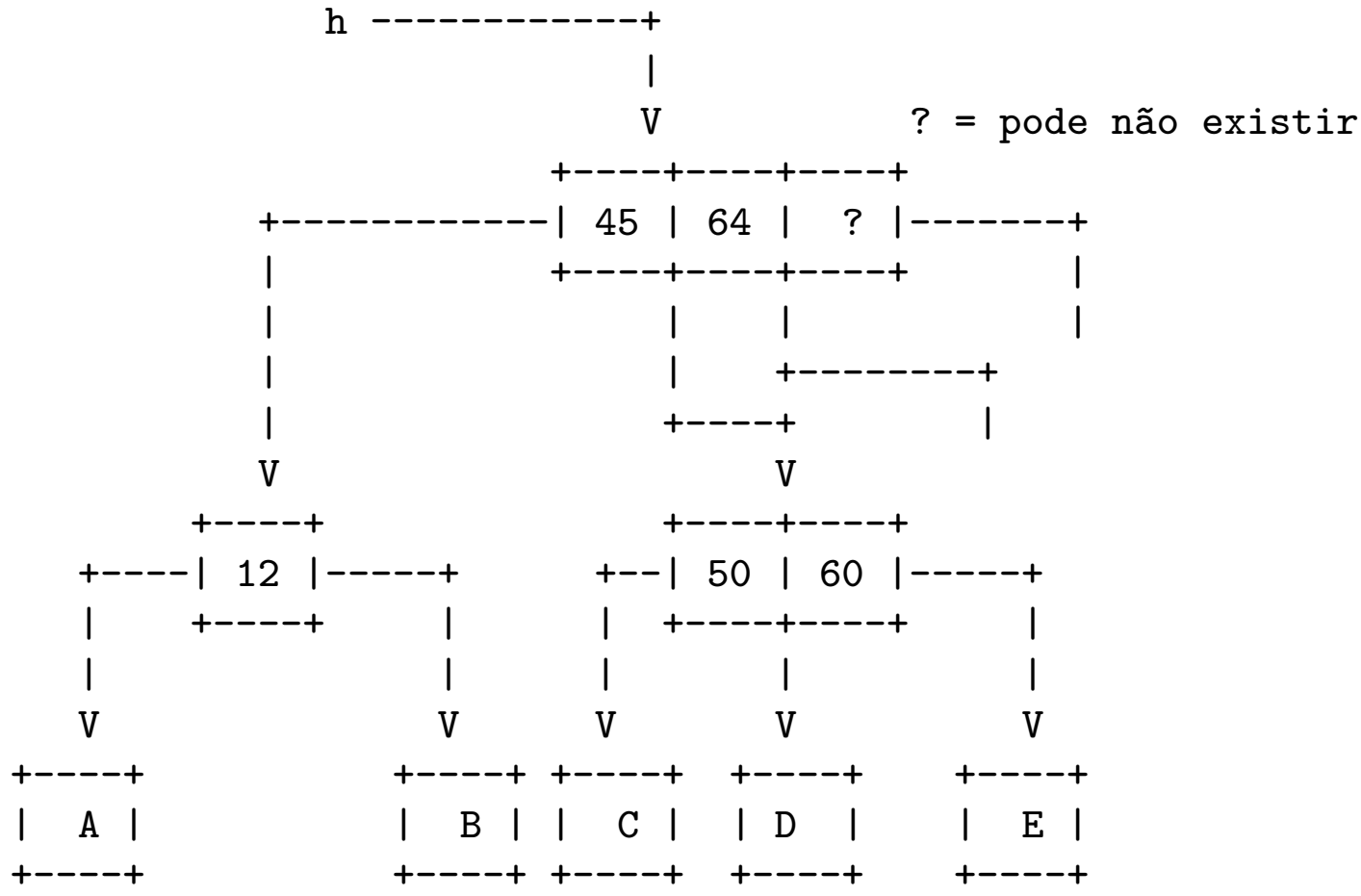
No meio do caminho, sabemos que
o nó corrente `h` é um 3-nó ou um 4-nó.

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.

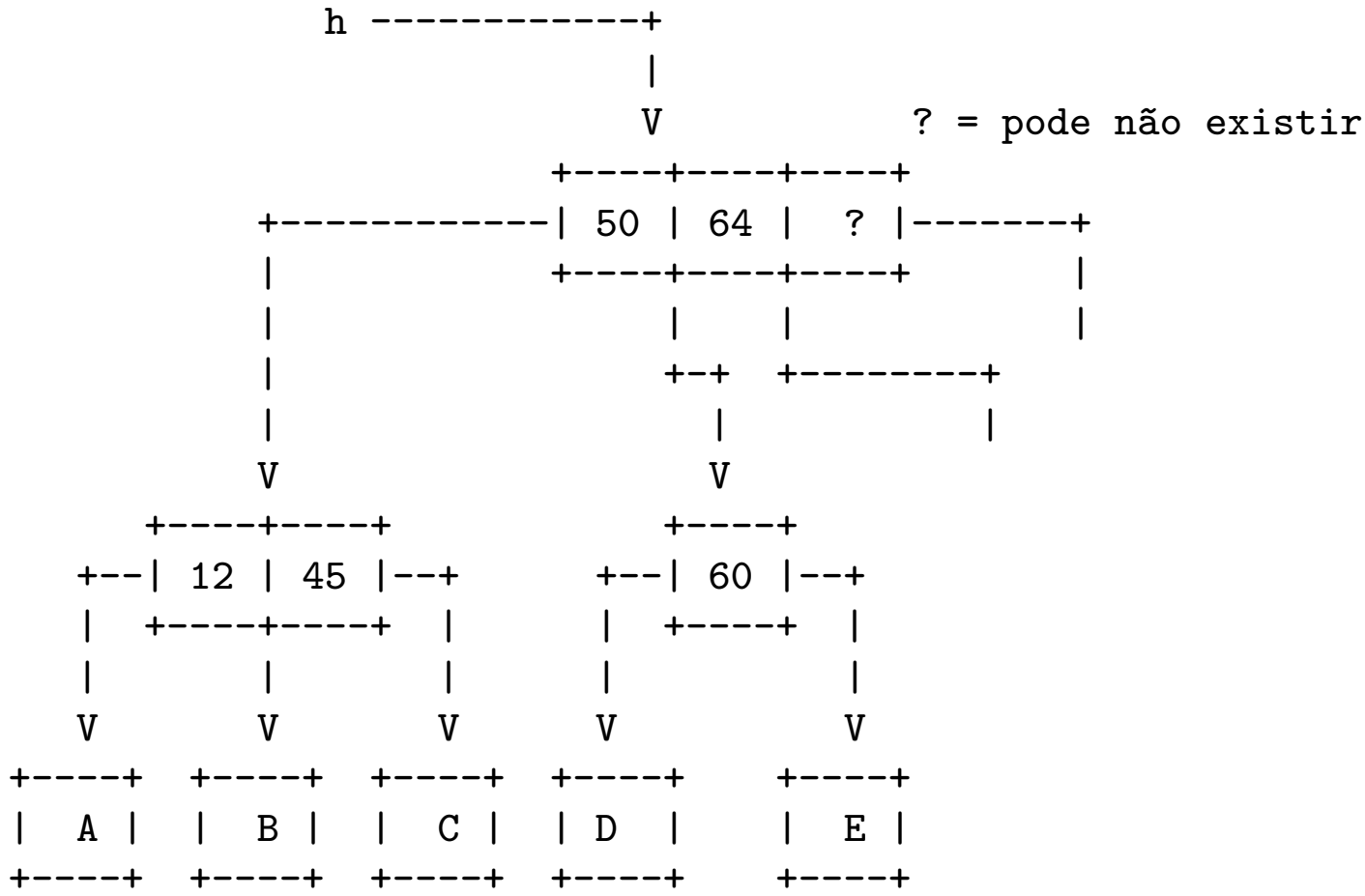
Comecemos com o caso em que
o filho esquerdo de `h` é um 2-nó.

Filho esquerdo de h é 2-nó



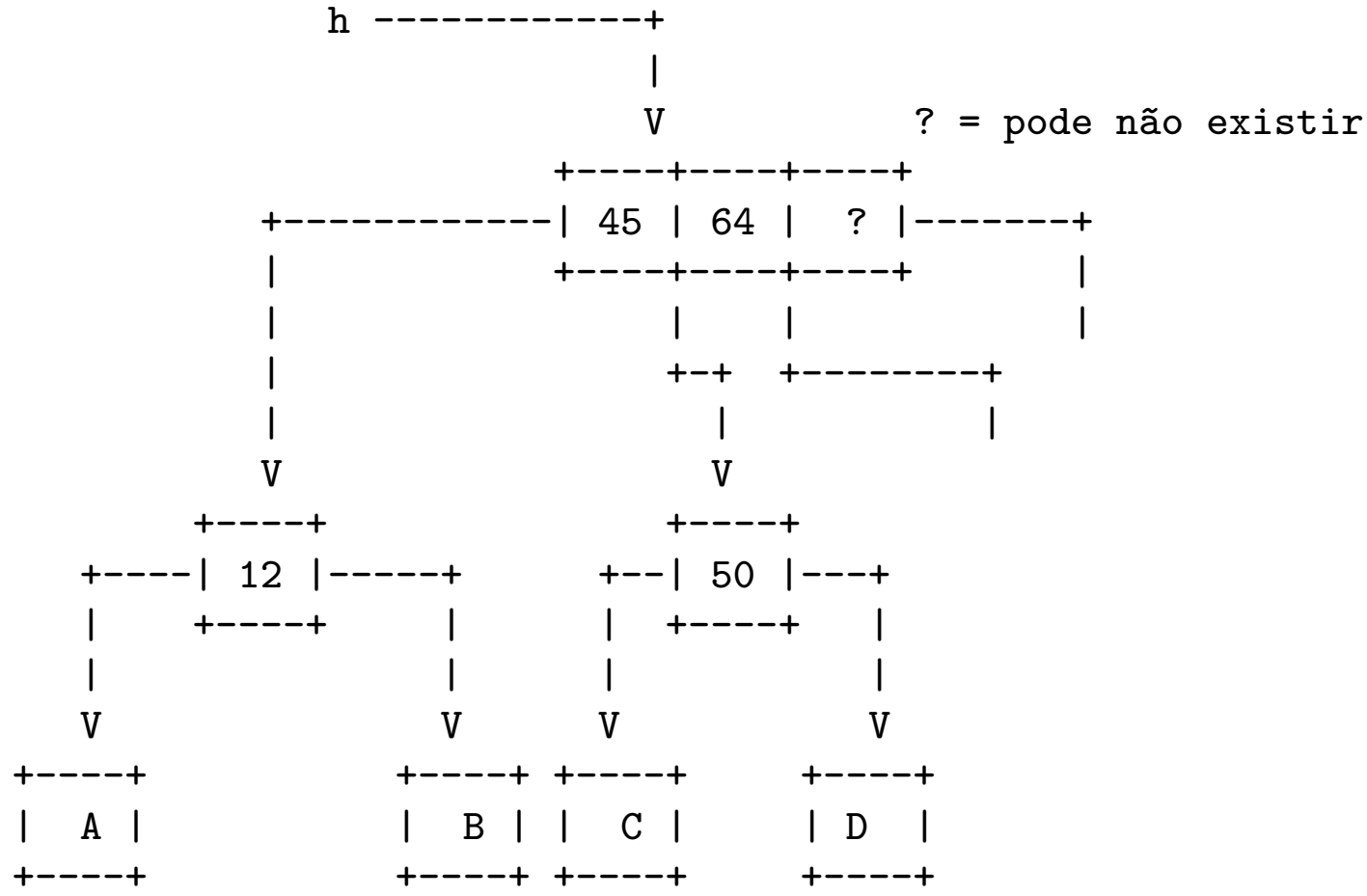
e filho do meio é 3-nó.

Filho esquerdo de h é 2-nó

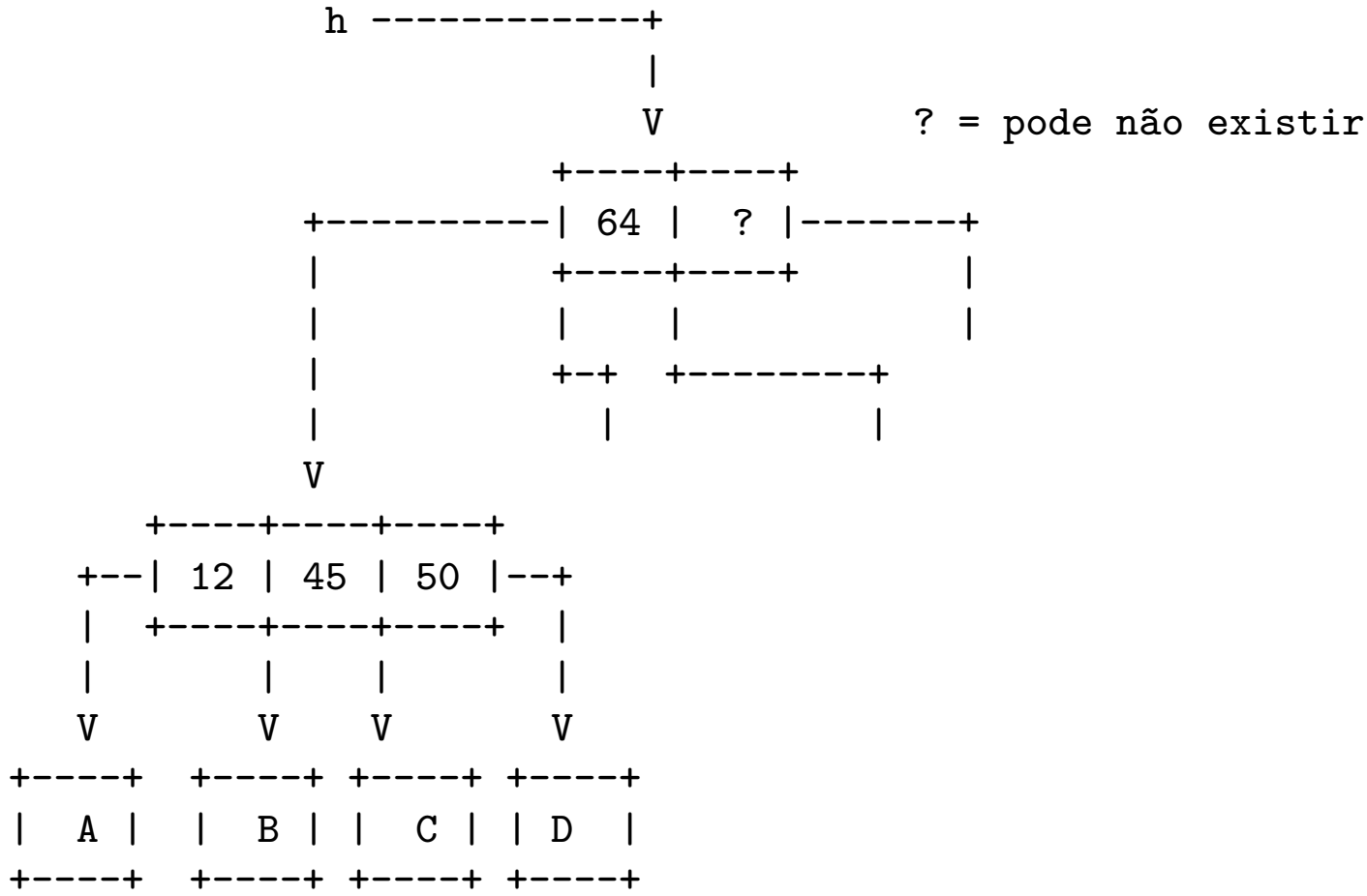


e filho do meio é 3-nó.

Dois filhos mais à esquerda de h são 2-nós



Dois filhos mais à esquerda de h são 2-nós



Finalmente...

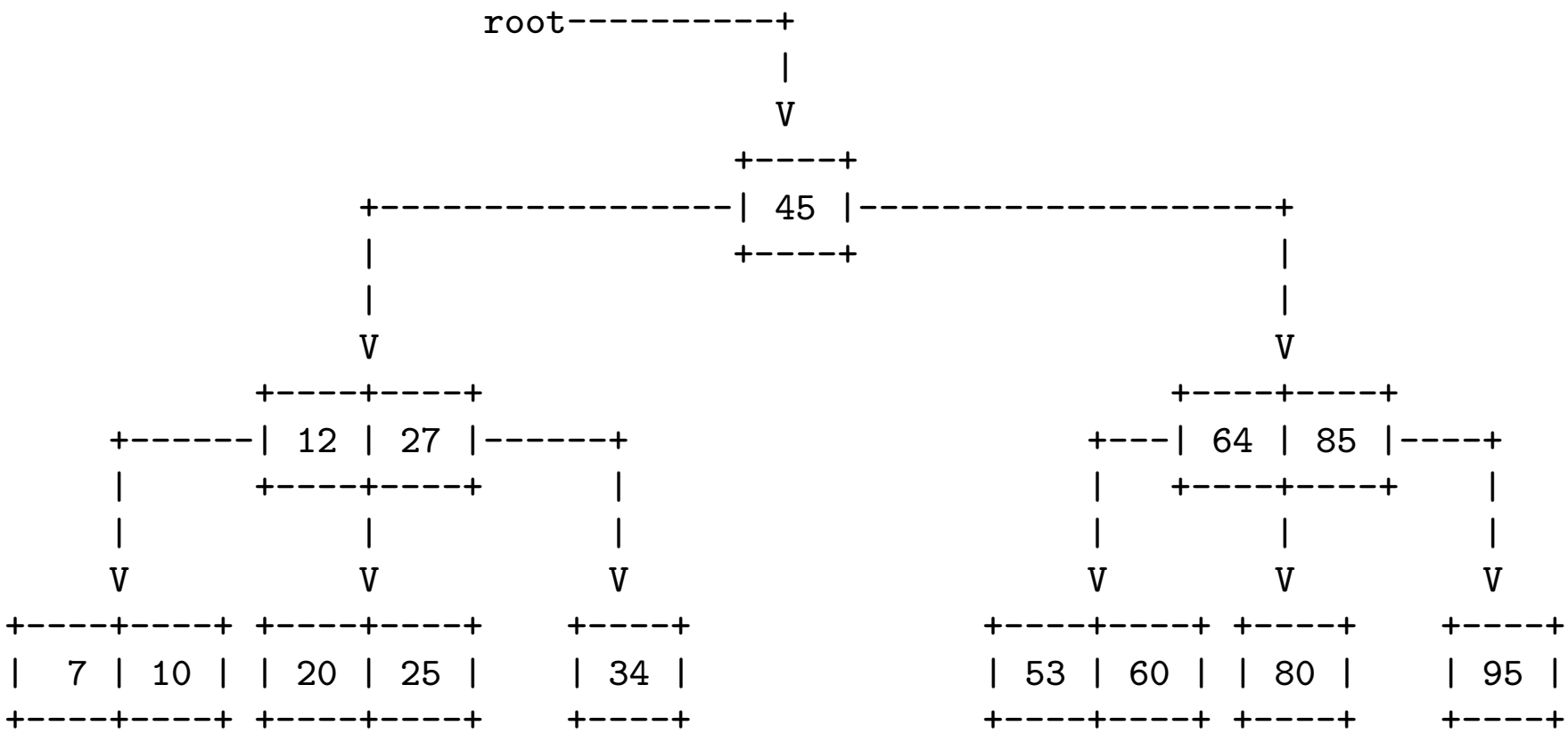
Desta forma, quando atingirmos a folha mais à esquerda dessa árvore 2-3-4, teremos chegado a um 3-nó ou 4-nó.

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

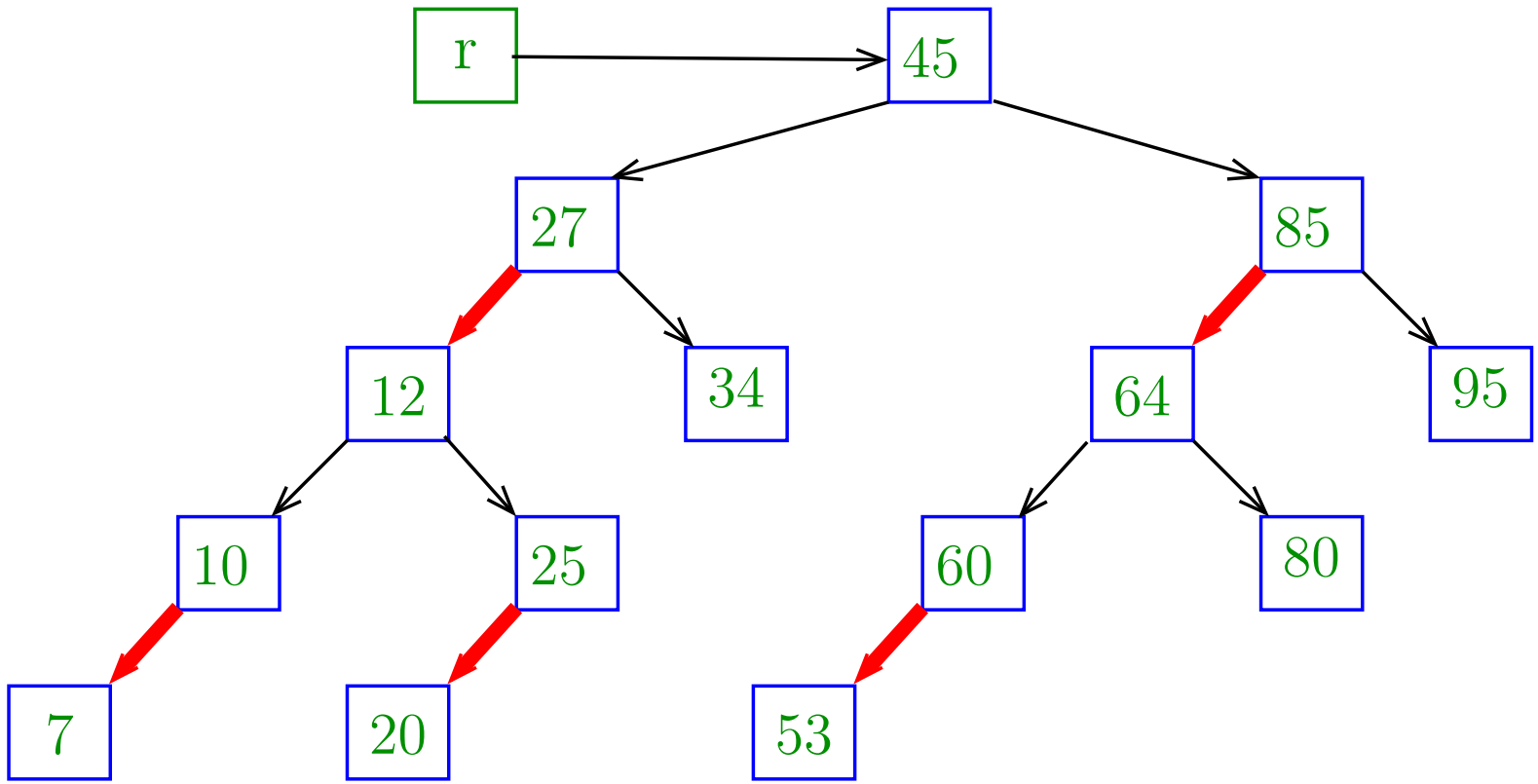
Depois devemos voltar "espatifando" os 4-nós que por ventura deixamos pelo caminho.

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()`

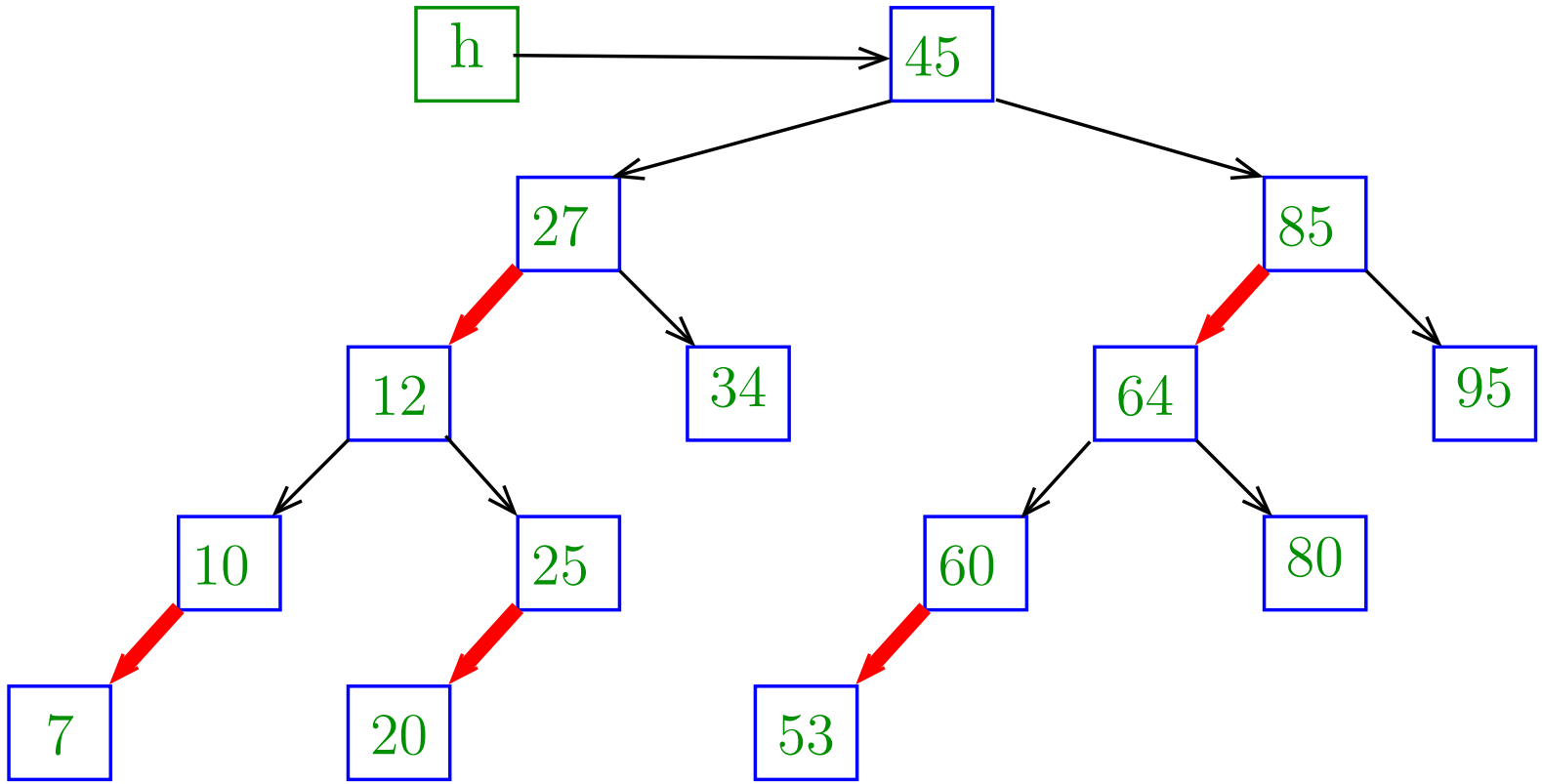


rubro-negra: `deleteMin()`



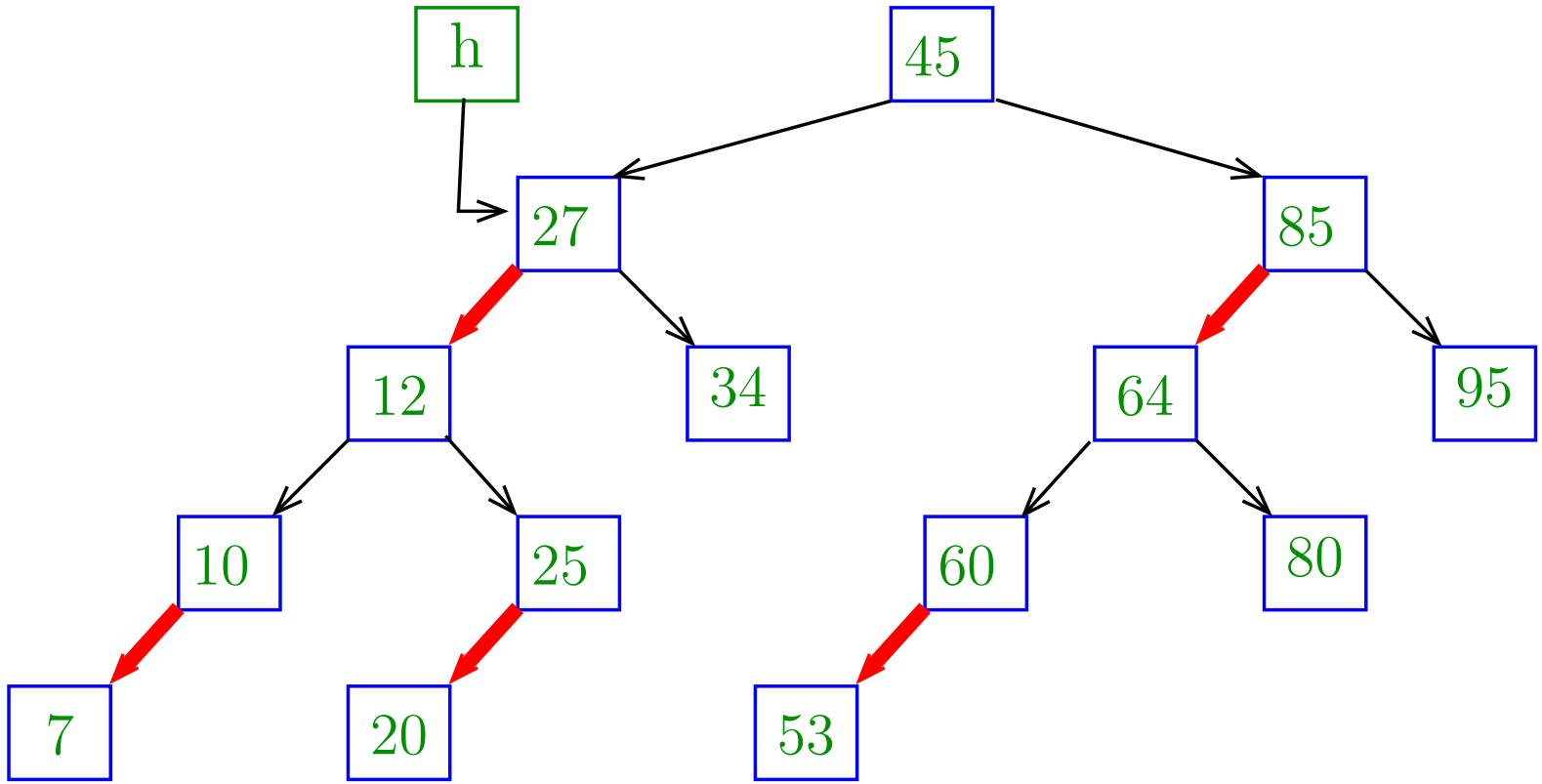
`deleteMin(r);`

rubro-negra: deleteMin()



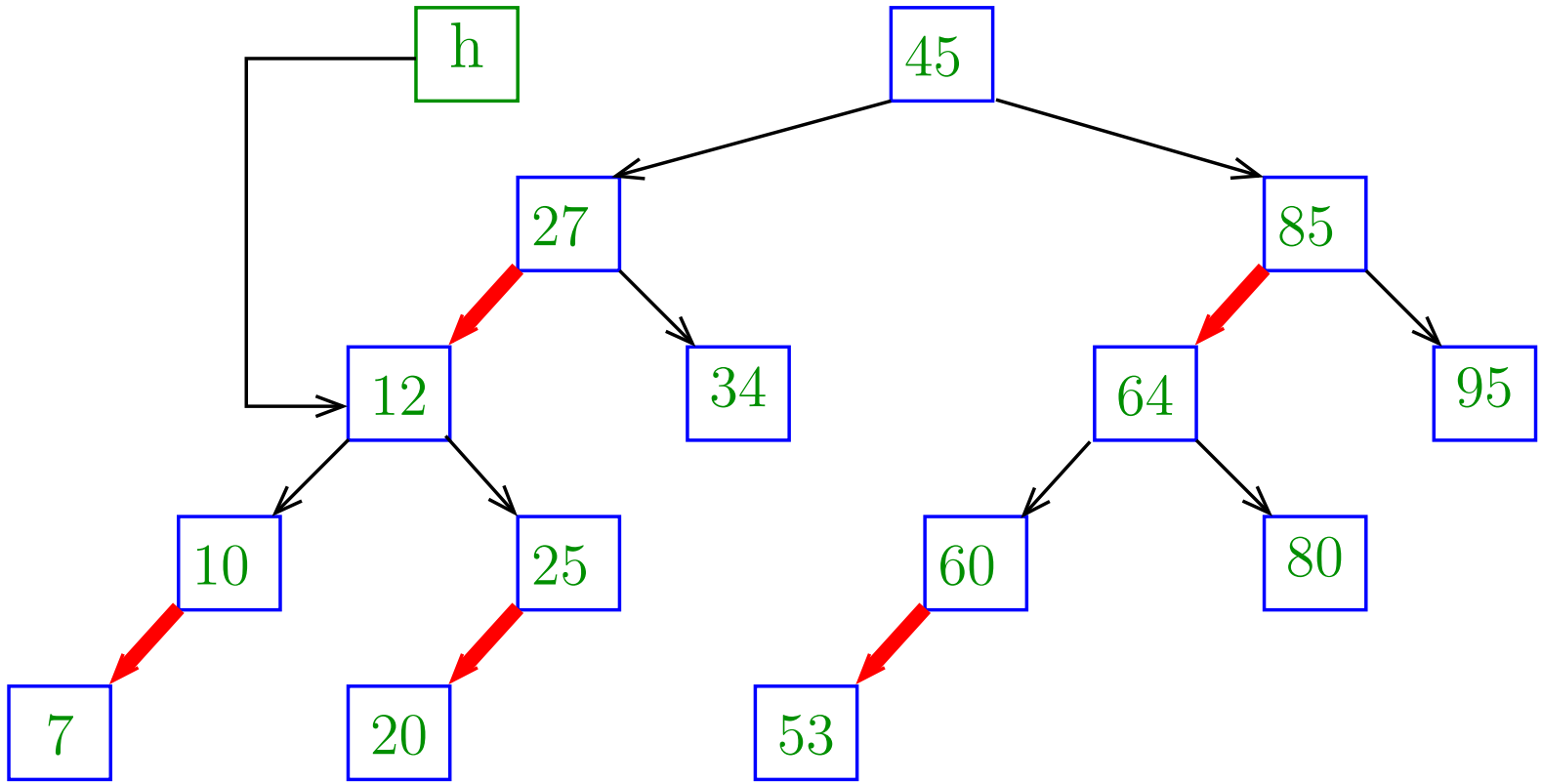

```
h->left = deleteMin(h->left);
```

rubro-negra: deleteMin()



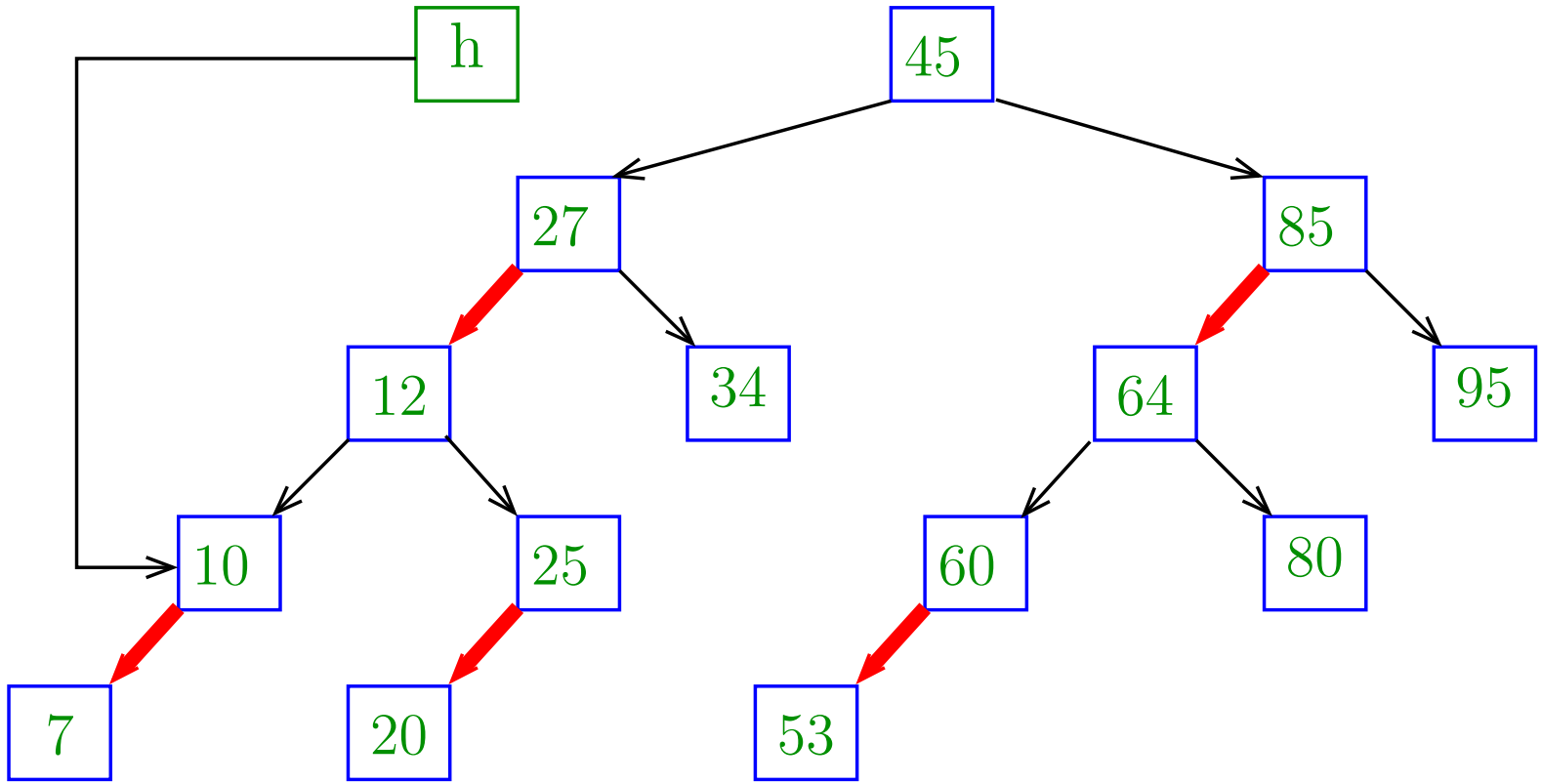

```
h->left = deleteMin(h->left);
```

rubro-negra: deleteMin()



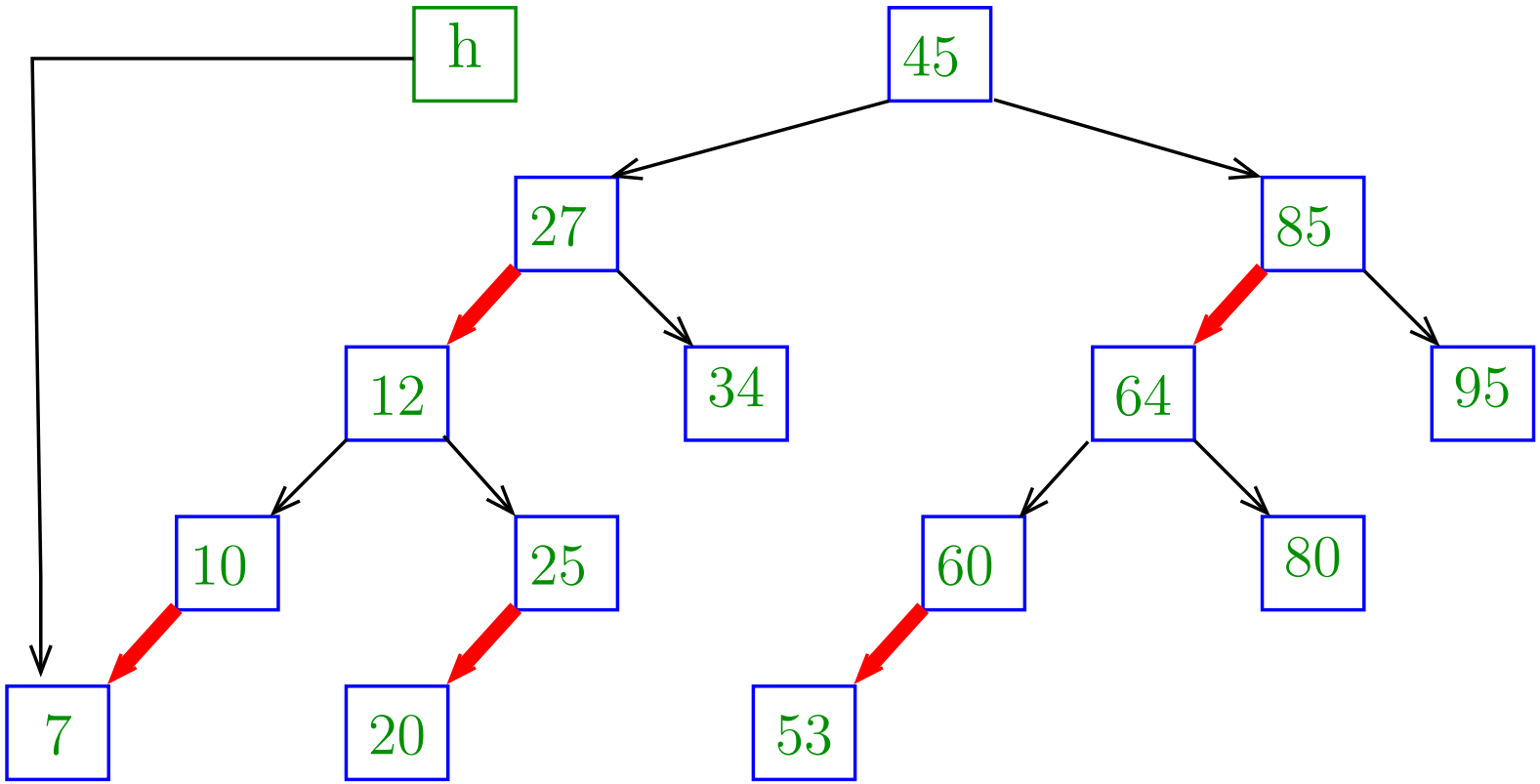

```
h->left = deleteMin(h->left);
```

rubro-negra: deleteMin()



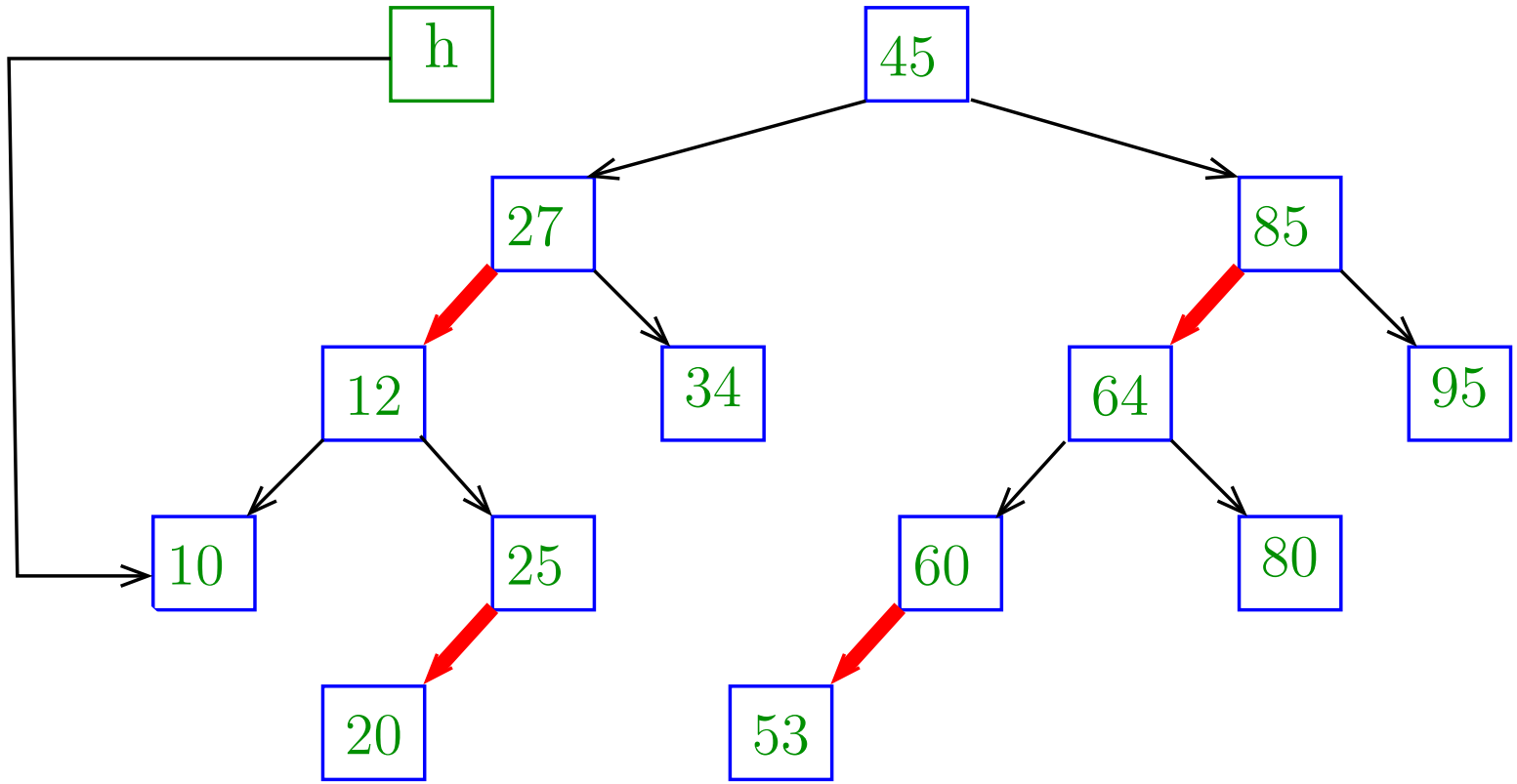

```
h->left = deleteMin(h->left);
```

rubro-negra: deleteMin()



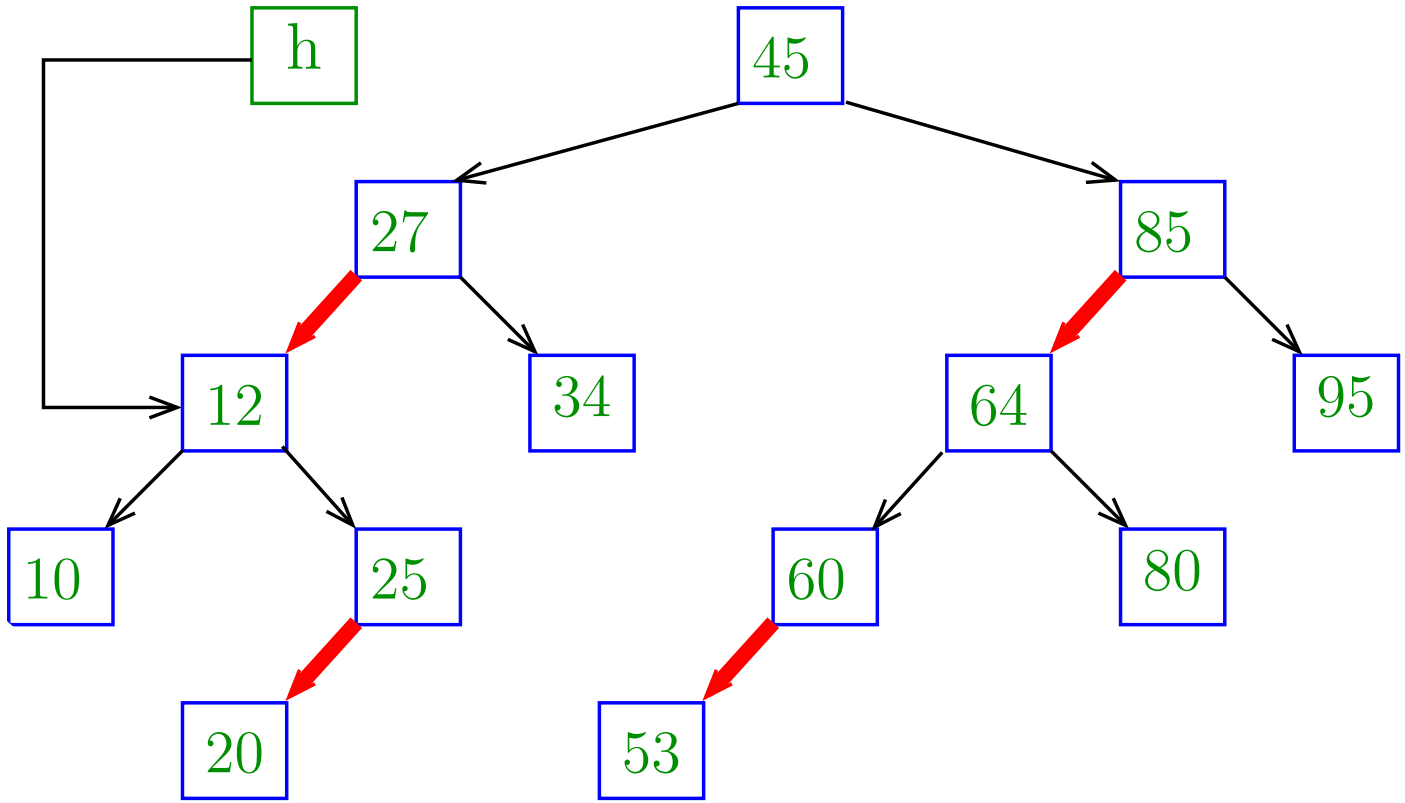

```
return NULL;
```

rubro-negra: deleteMin()



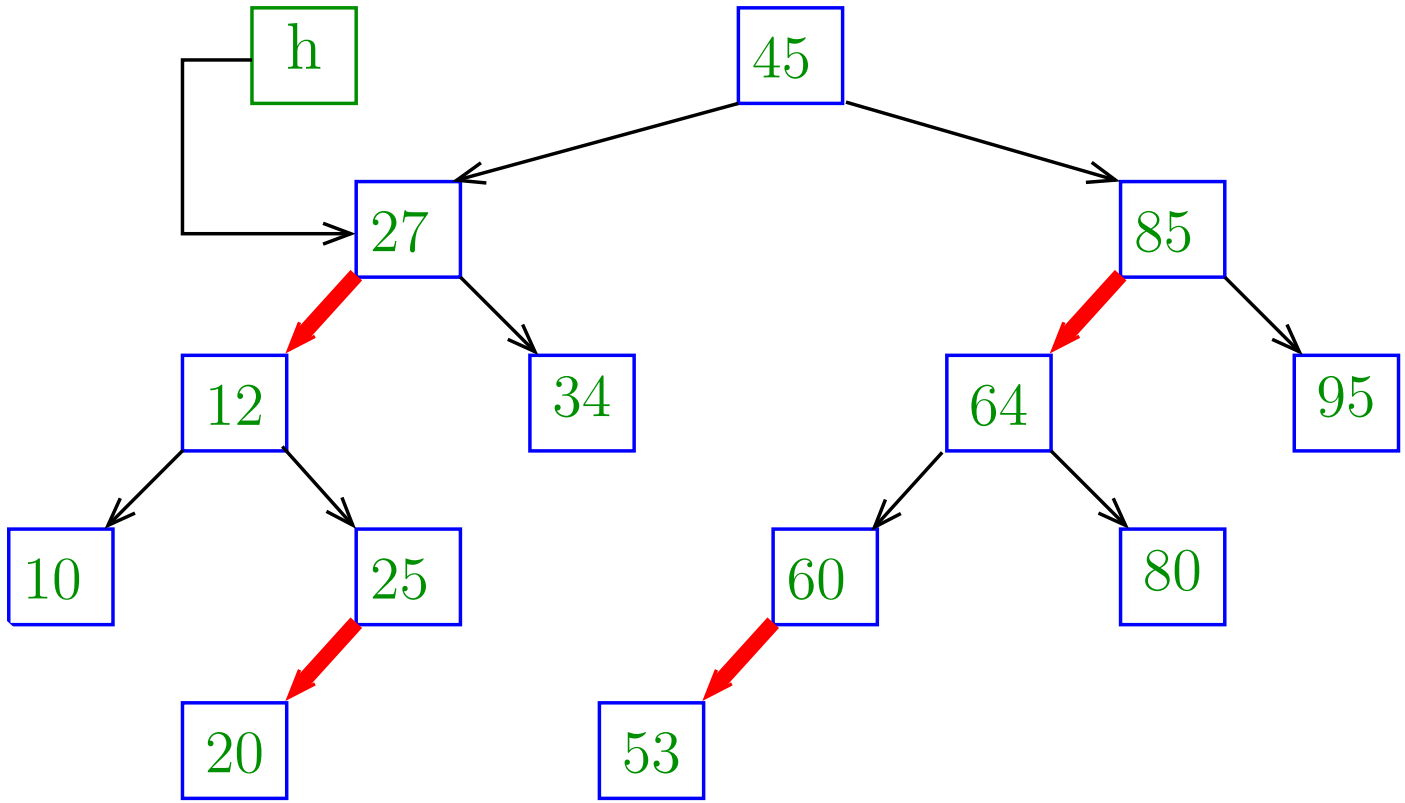

```
return balance(h);
```

rubro-negra: deleteMin()



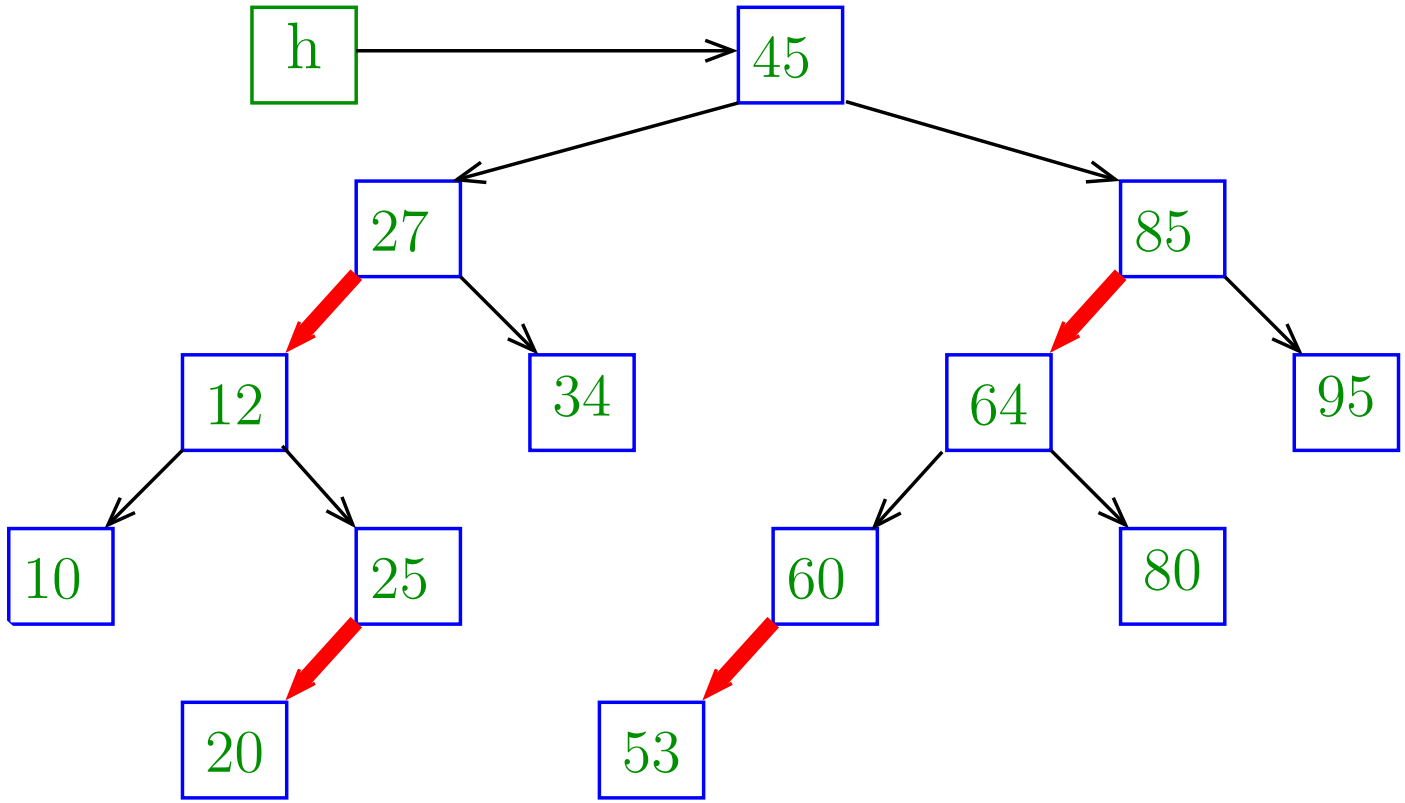

```
return balance(h);
```

rubro-negra: deleteMin()



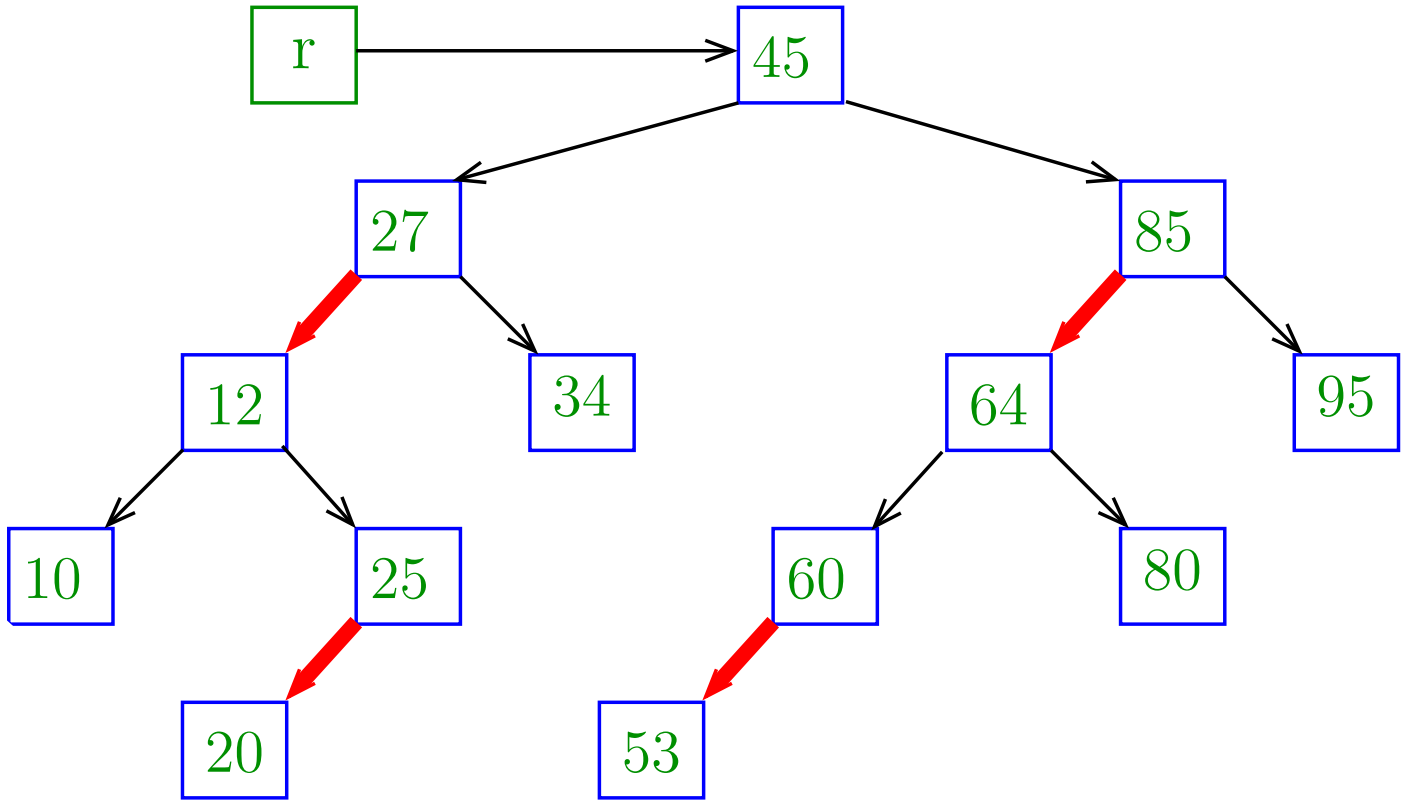

```
return balance(h);
```

rubro-negra: deleteMin()

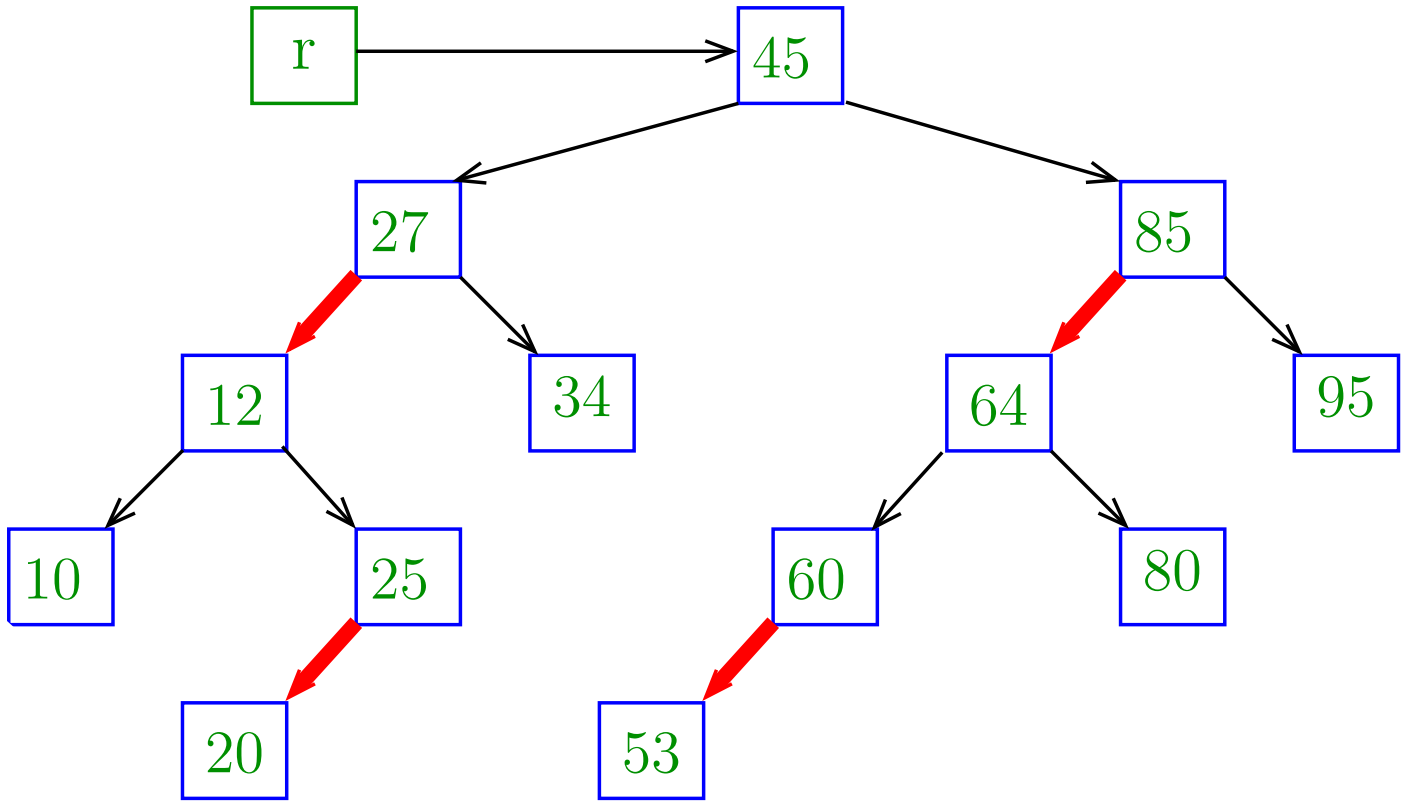



```
return balance(h);
```

rubro-negra: `deleteMin()`

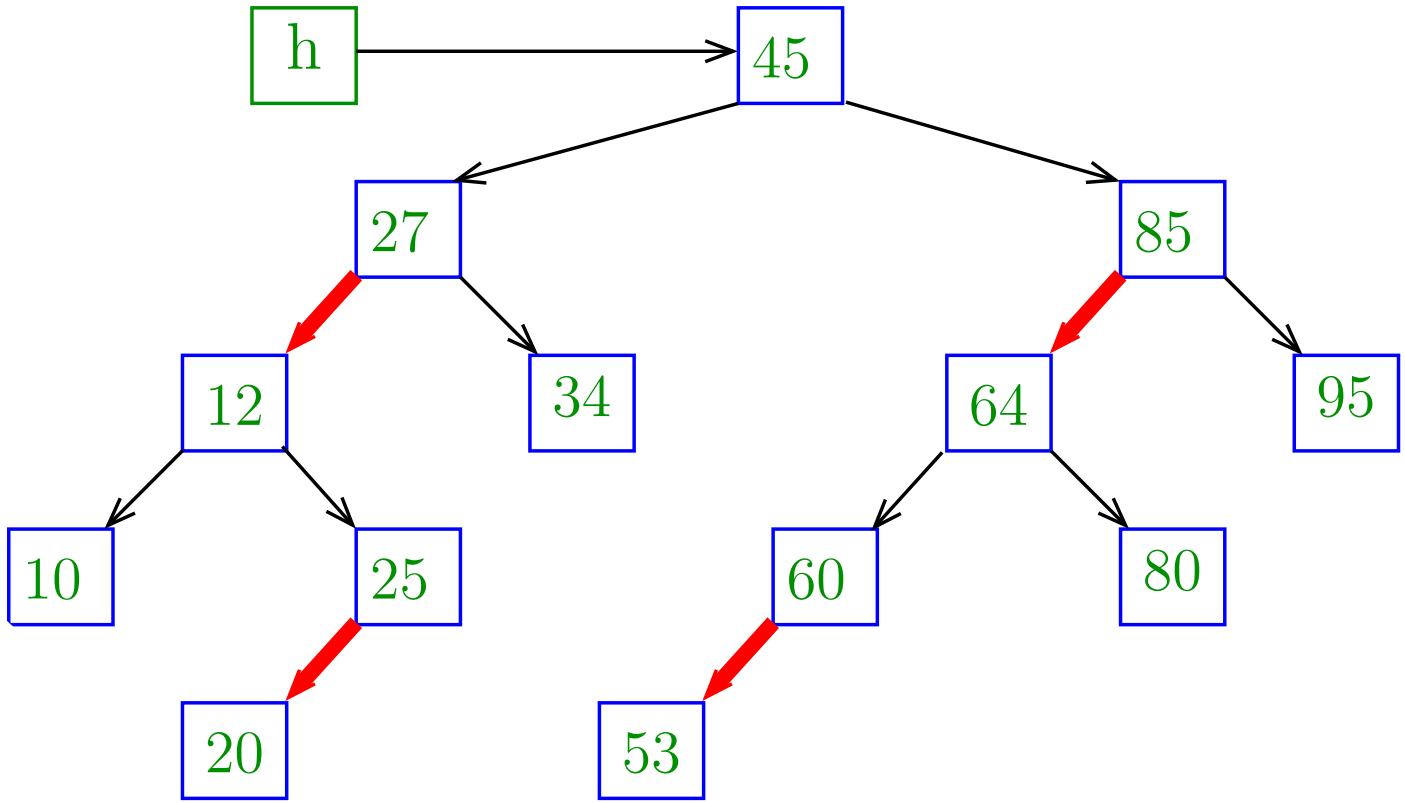


rubro-negra: outro `deleteMin()`



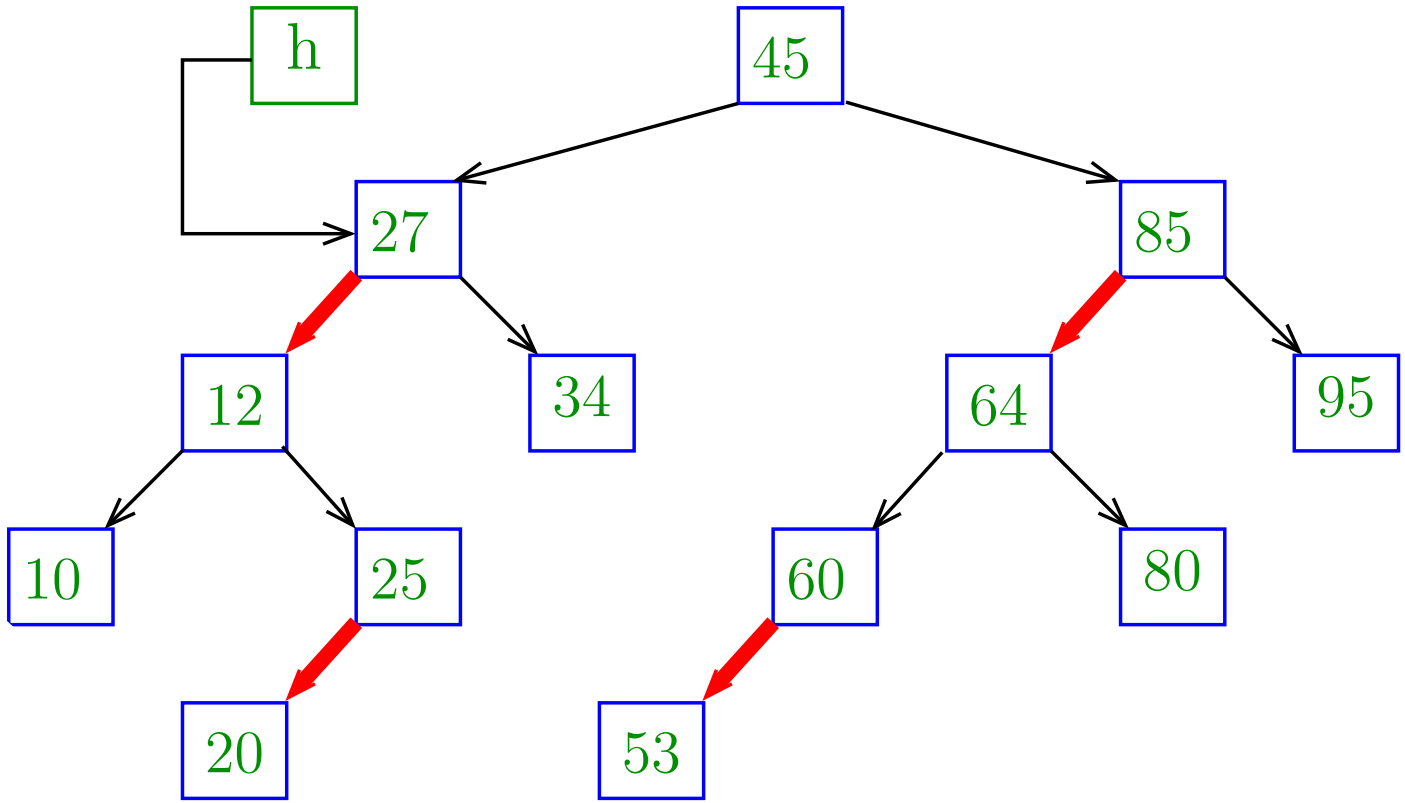

```
deleteMin(r);
```

rubro-negra: outro `deleteMin()`



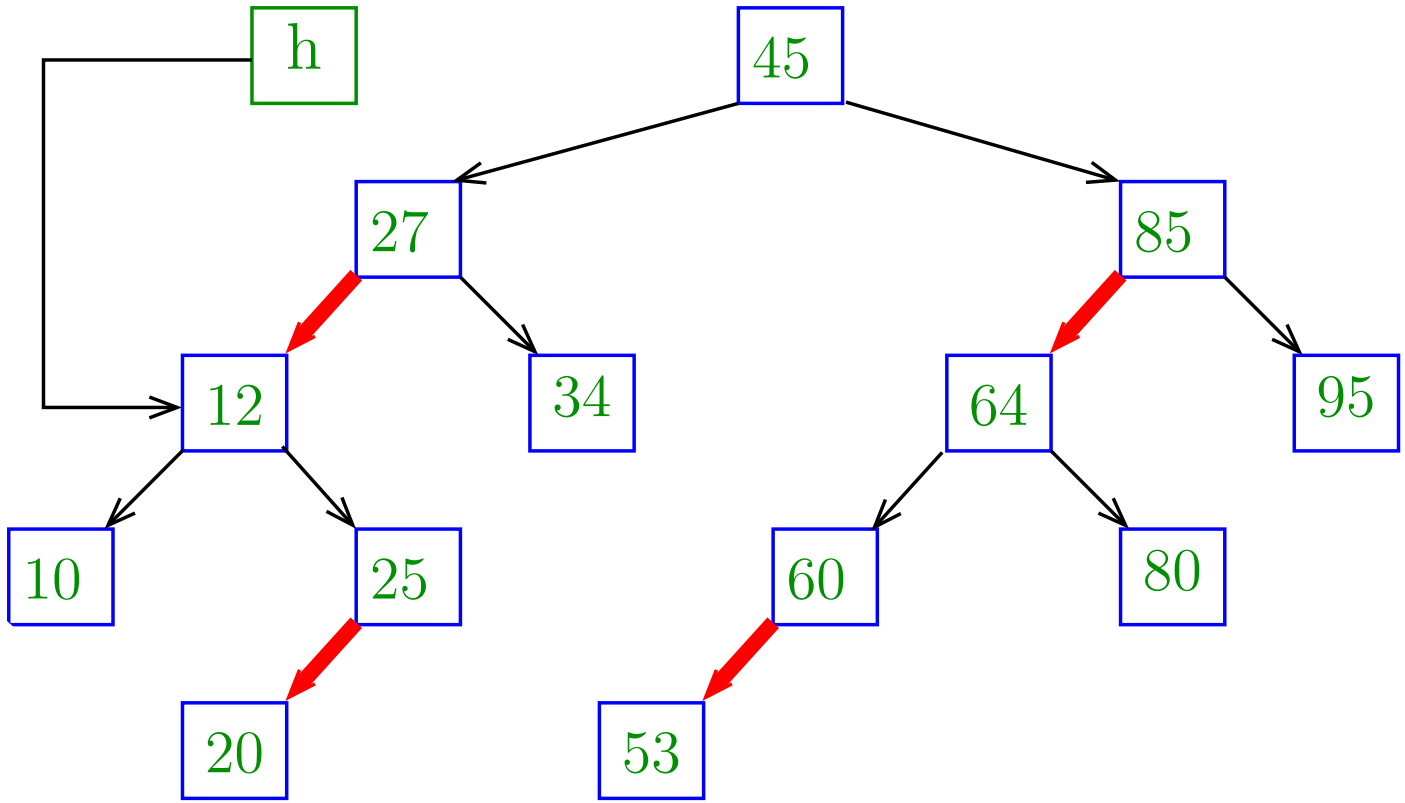

```
h->left = deleteMin(h->left);
```

rubro-negra: outro `deleteMin()`



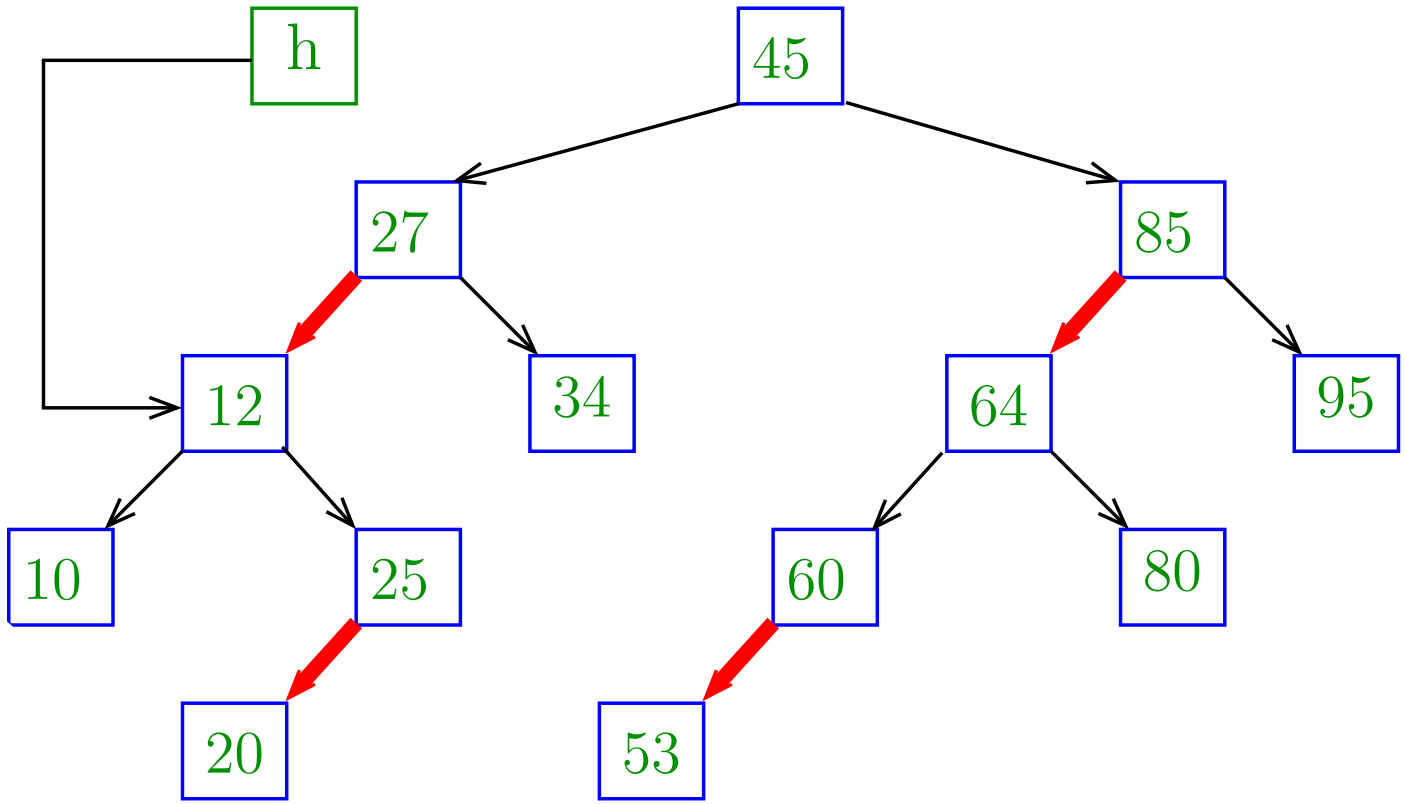

```
h->left = deleteMin(h->left);
```

rubro-negra: outro `deleteMin()`



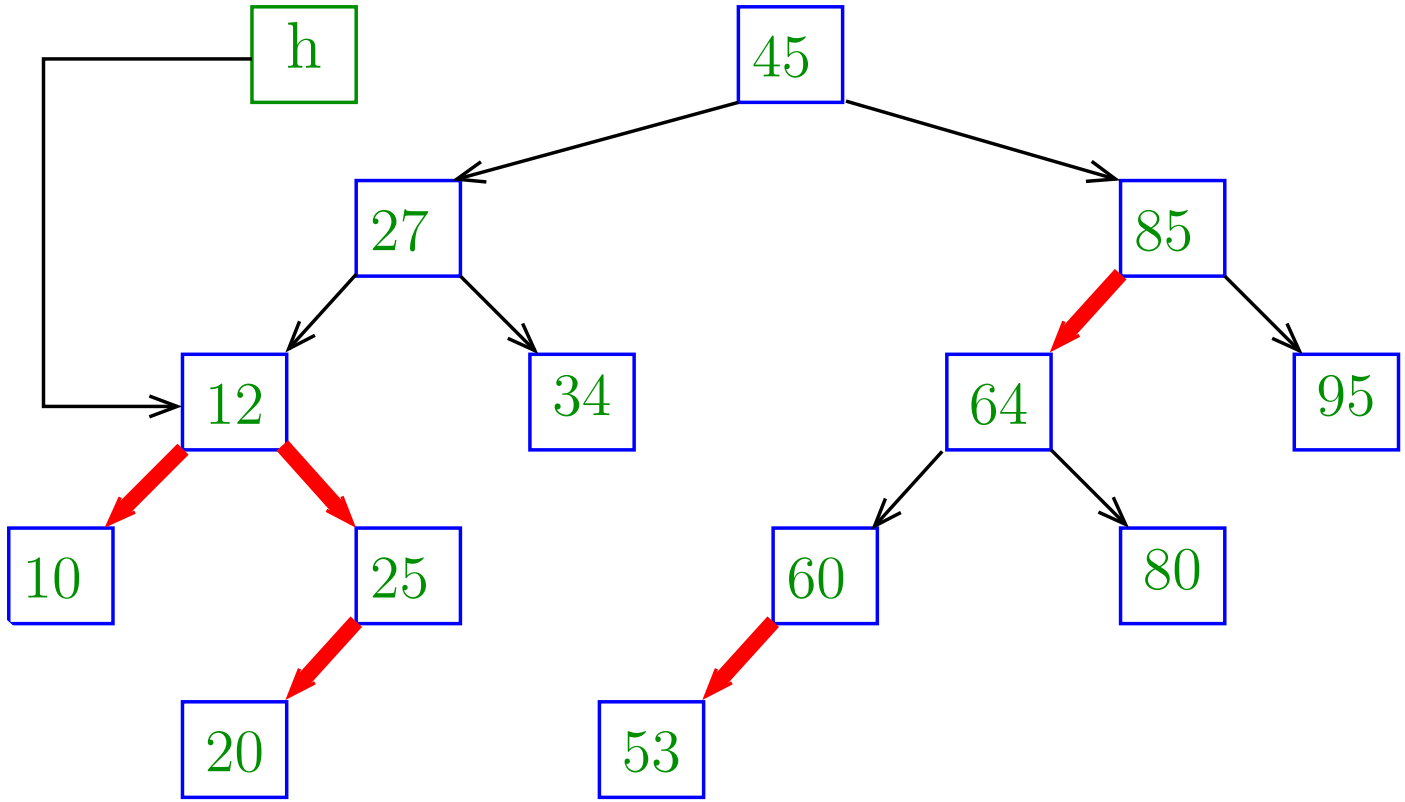

```
h = moveRedLeft(h);
```

rubro-negra: outro `deleteMin()`



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

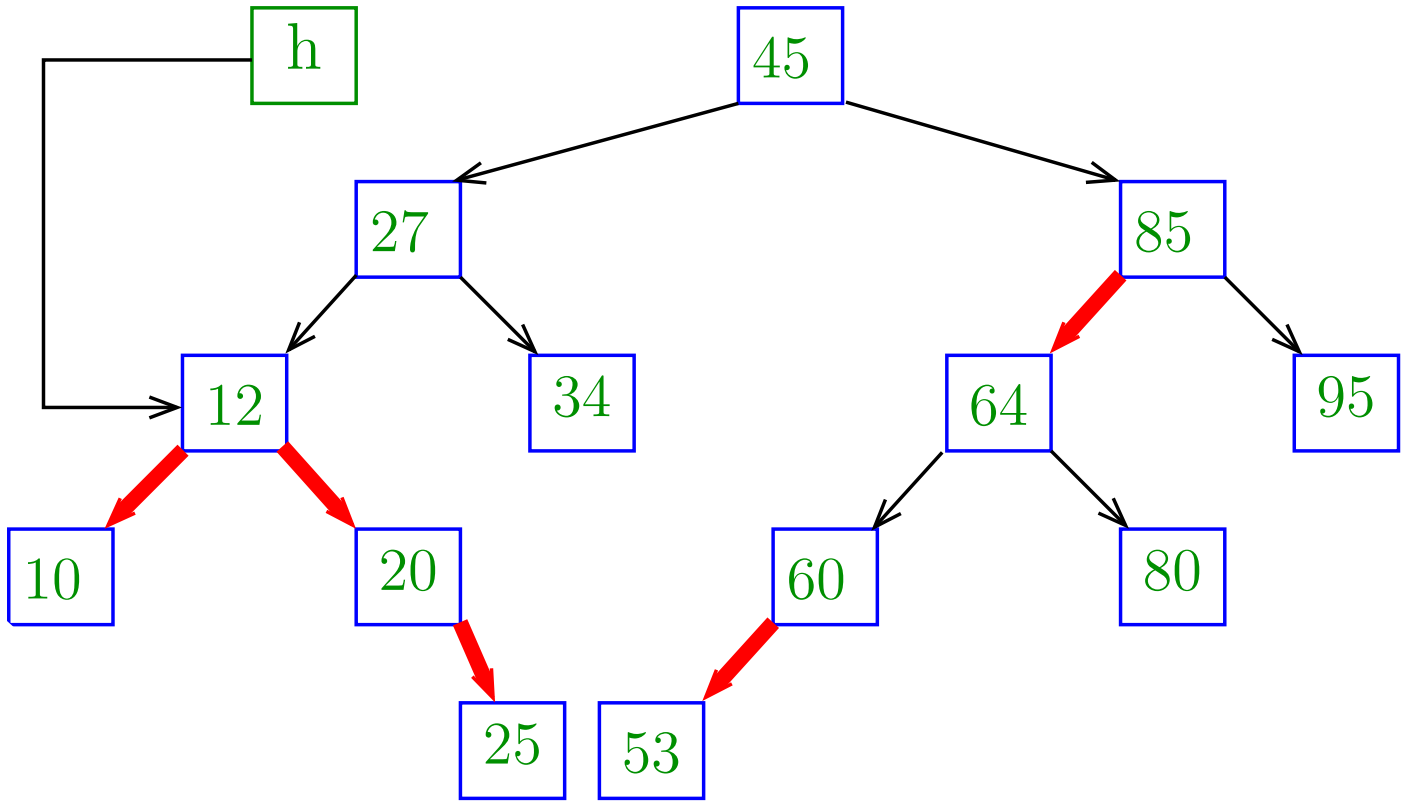
rubro-negra: outro deleteMin()




```
h->right = rotateRight(h->right);
```

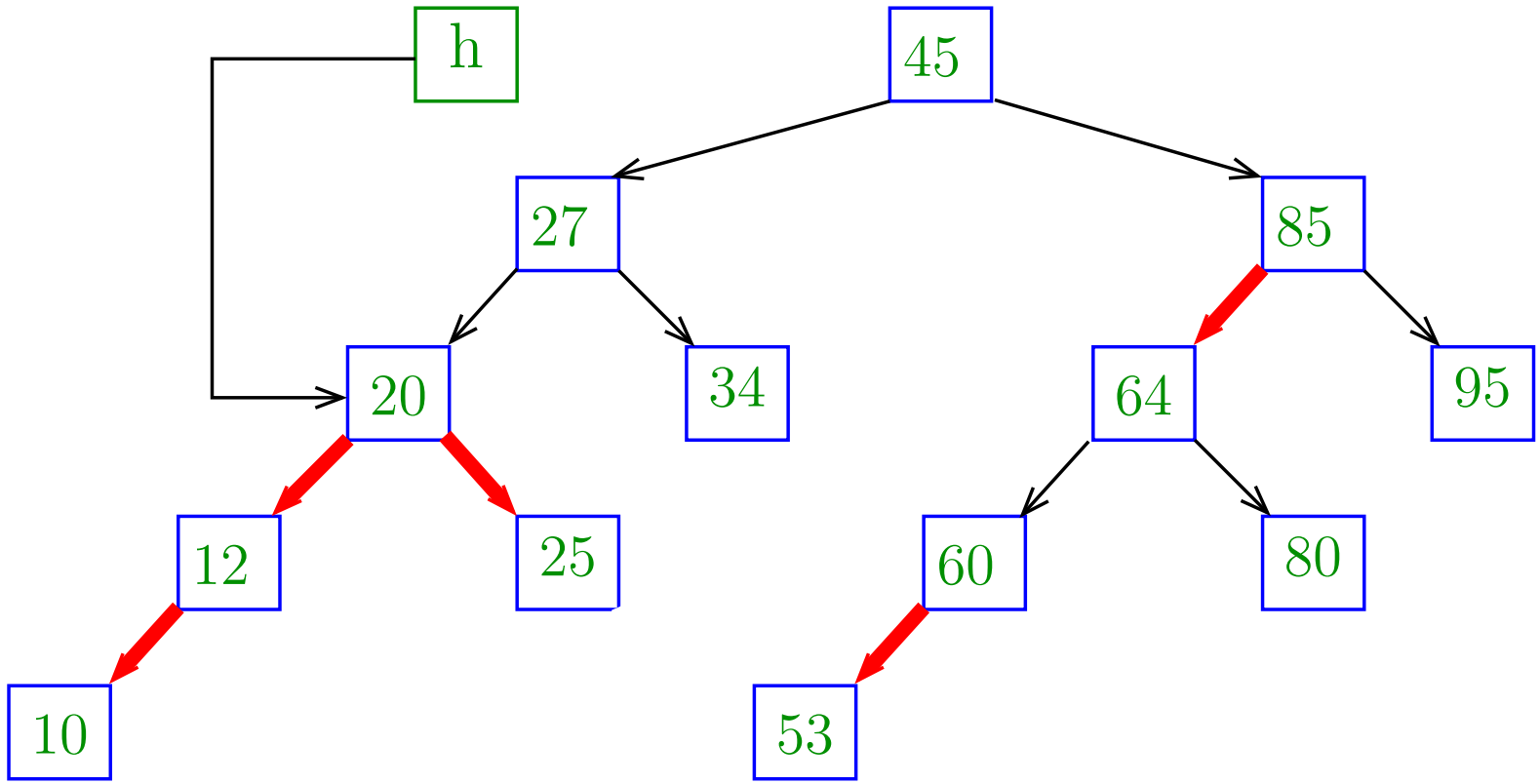
[moveRedLeft(h);]

rubro-negra: outro `deleteMin()`



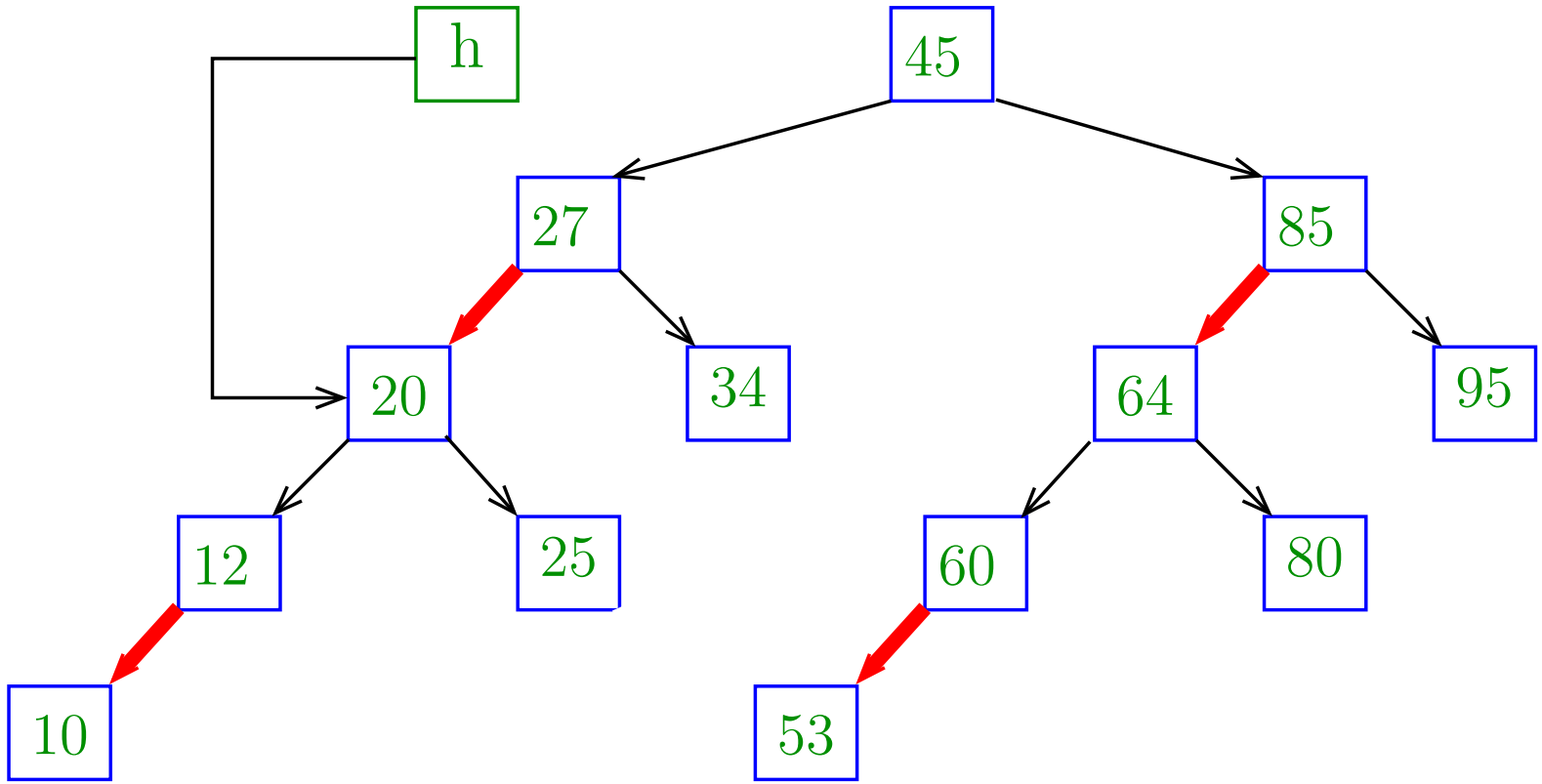
`h = rotateLeft(h);` `[moveRedLeft(h);]`

rubro-negra: outro `deleteMin()`



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

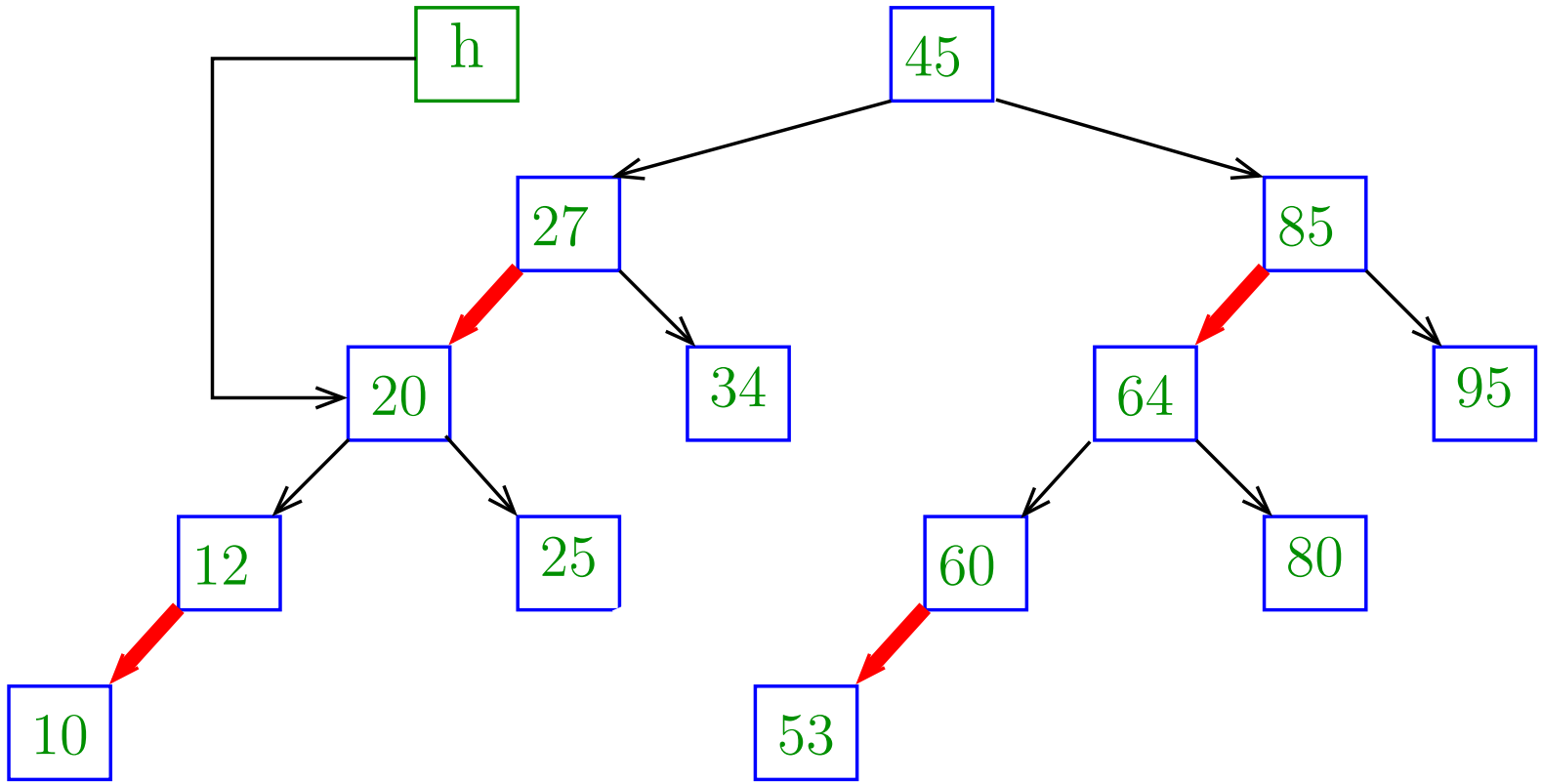
rubro-negra: outro `deleteMin()`




```
return h;
```

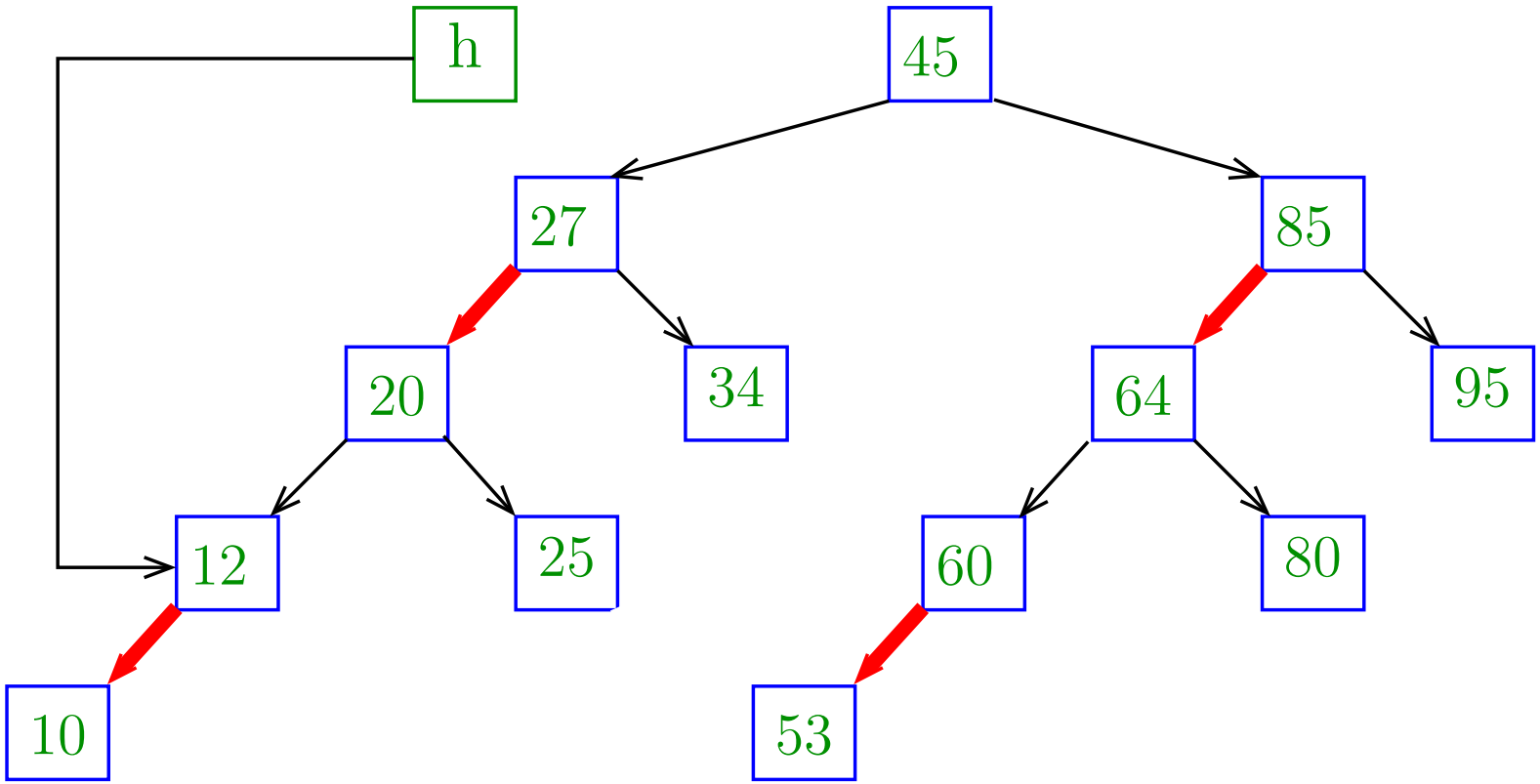
```
[moveRedLeft(h);]
```

rubro-negra: outro `deleteMin()`



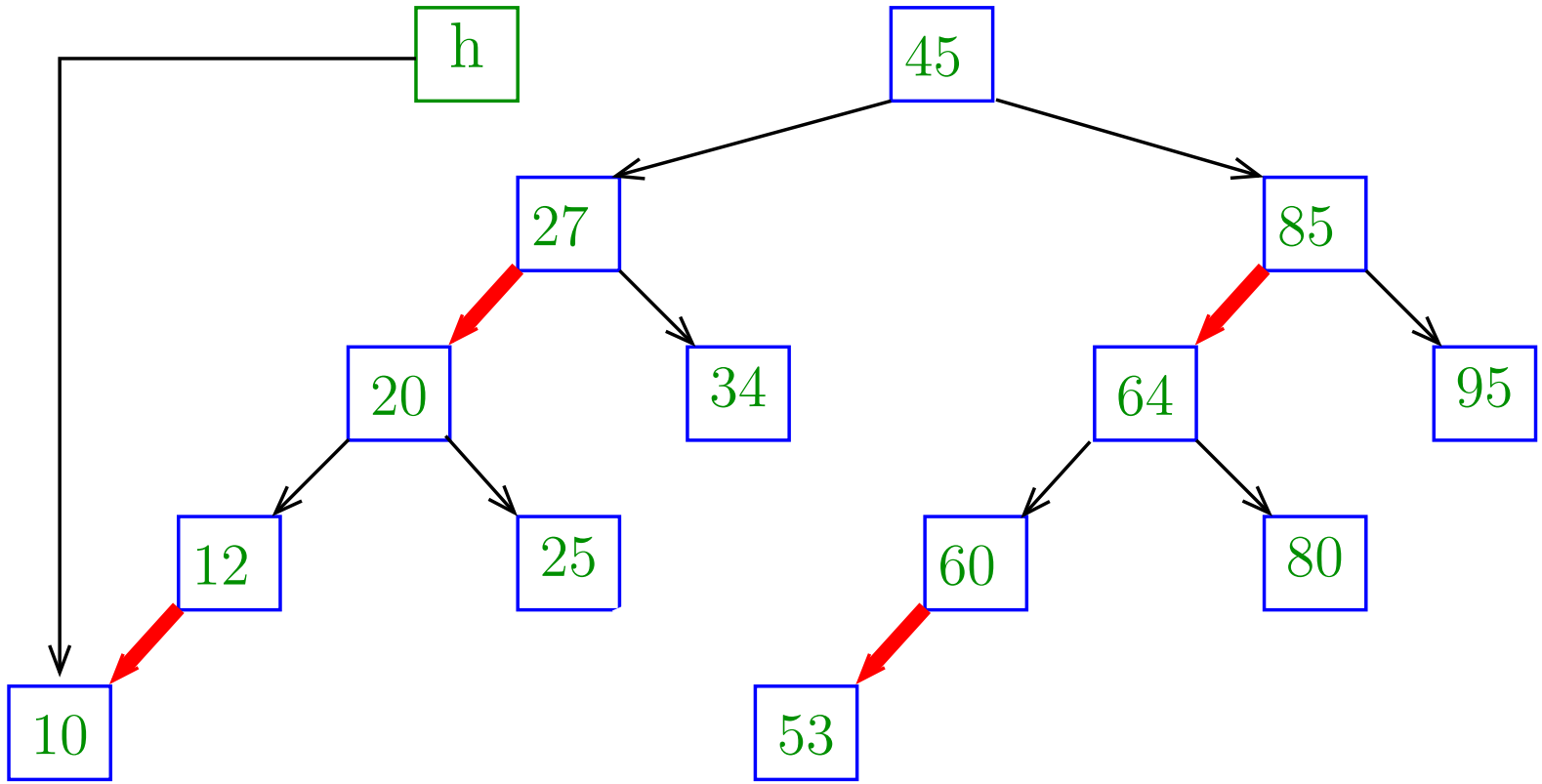

```
h->left = deleteMin(h->left);
```

rubro-negra: outro `deleteMin()`



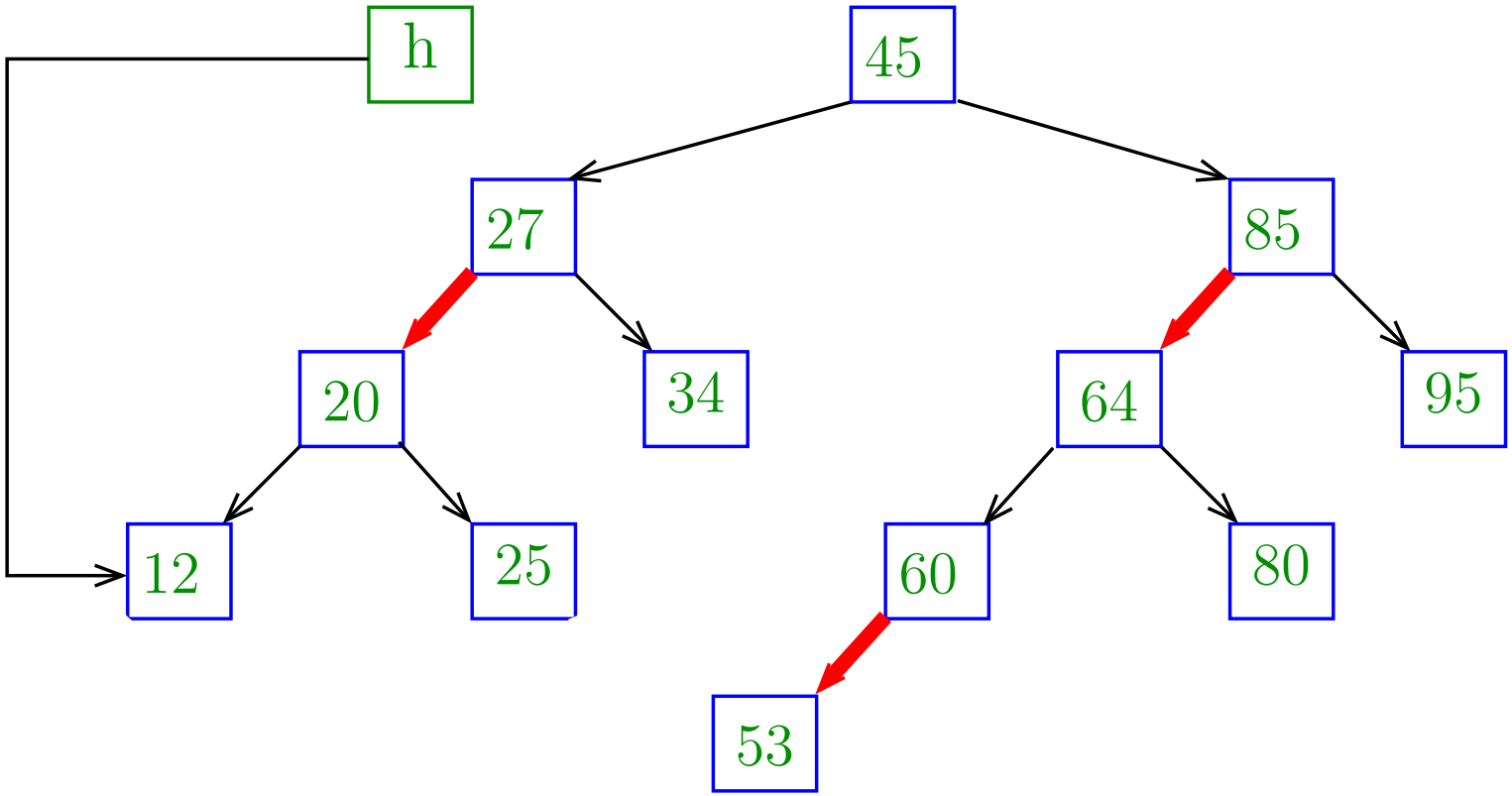

```
h->left = deleteMin(h->left);
```

rubro-negra: outro `deleteMin()`



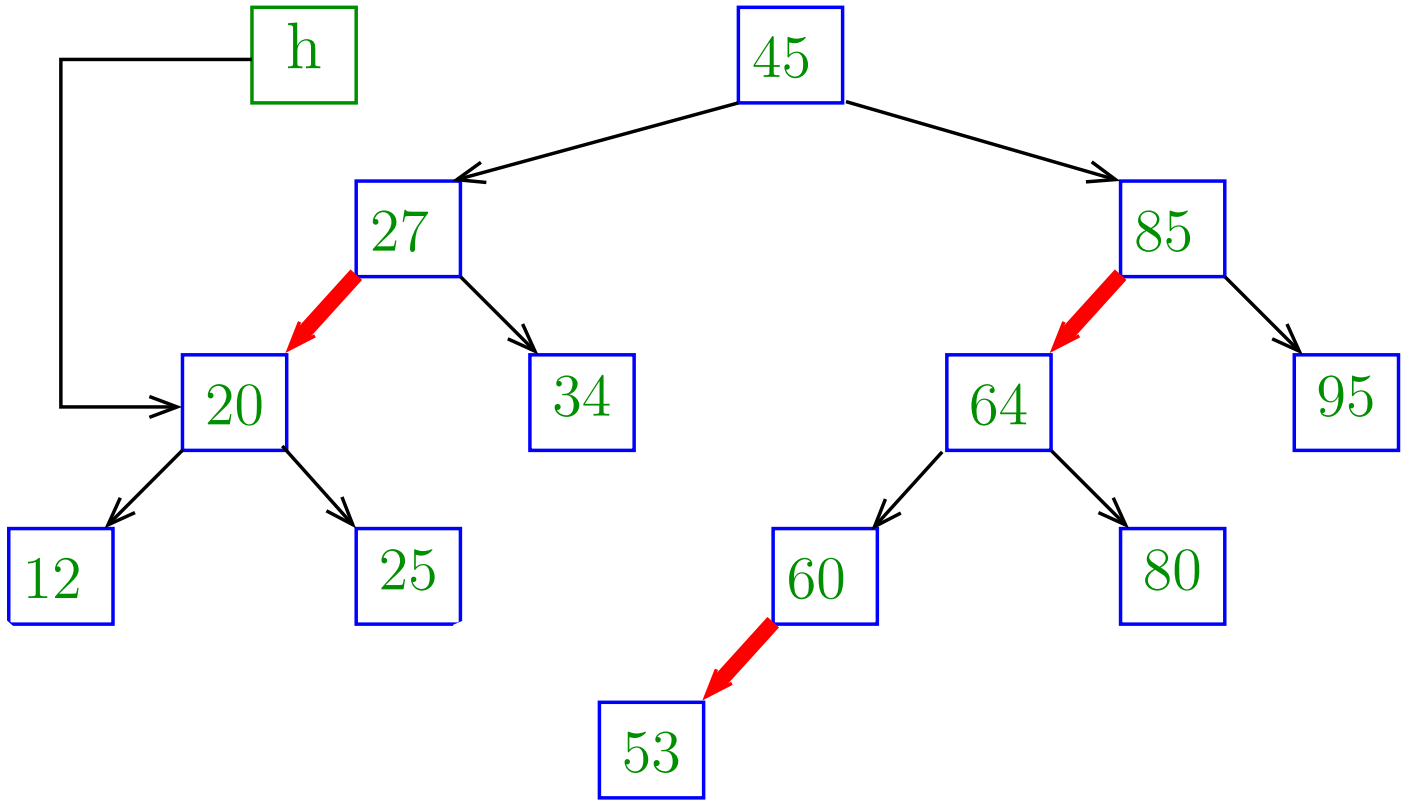

```
return NULL;
```

rubro-negra: outro `deleteMin()`



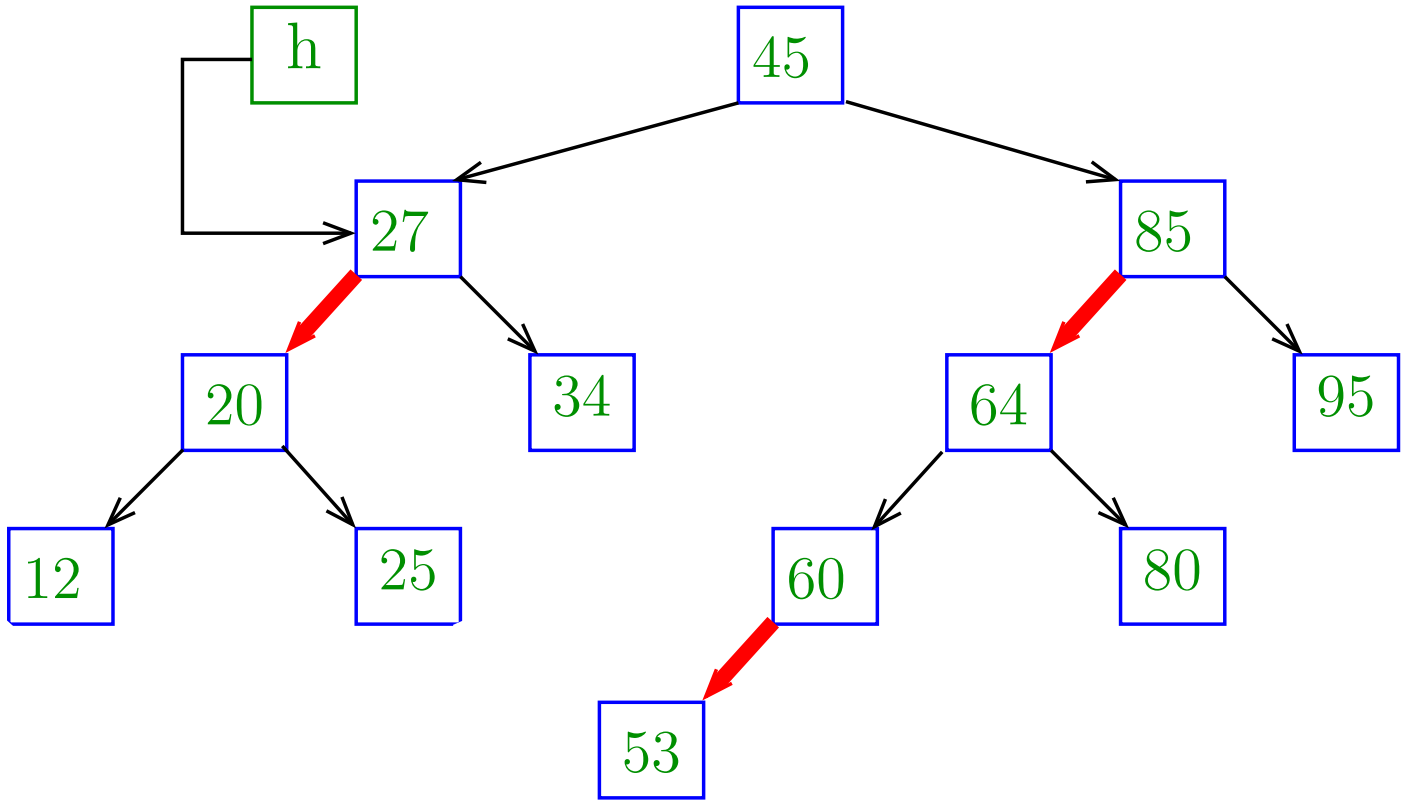

```
return balance(h);
```

rubro-negra: outro `deleteMin()`



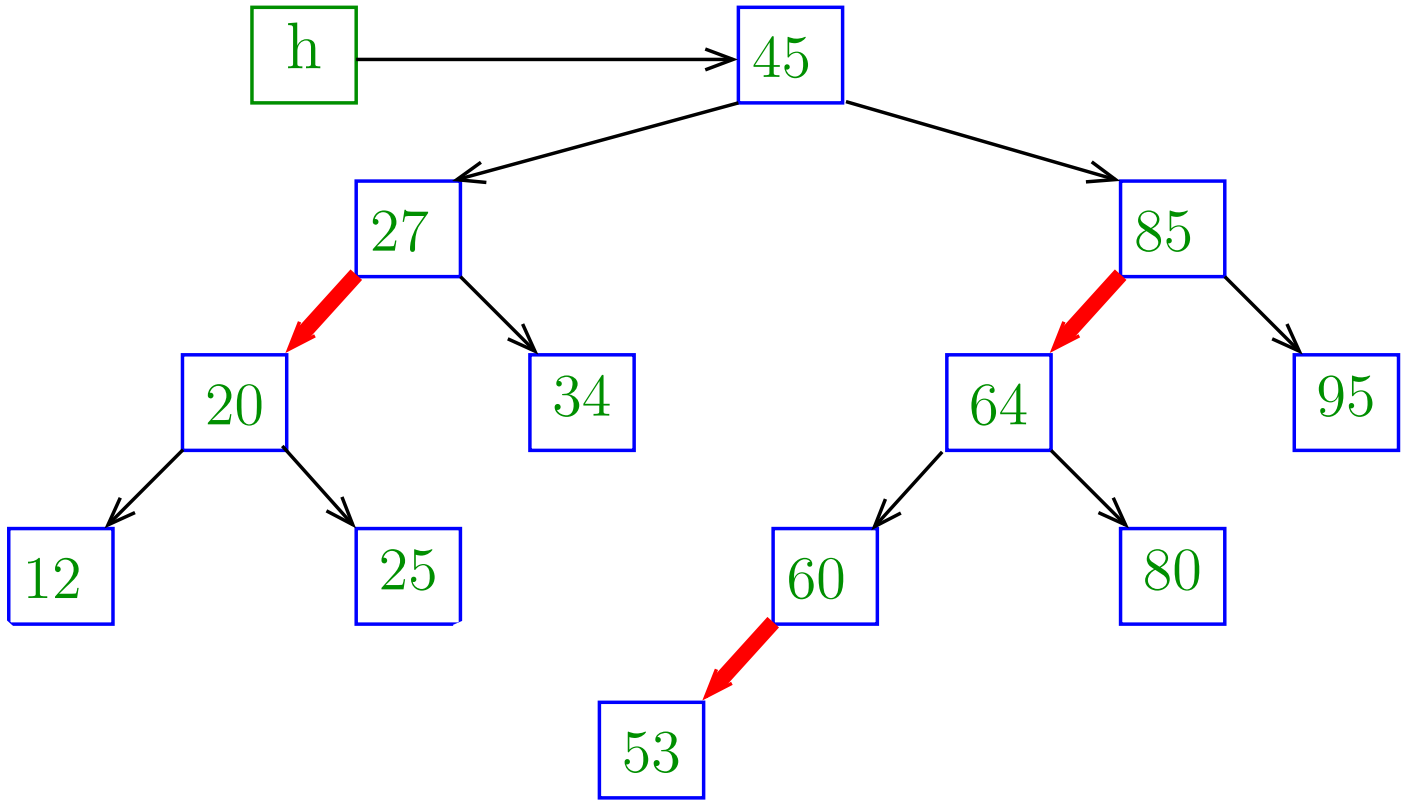

```
return balance(h);
```

rubro-negra: outro `deleteMin()`



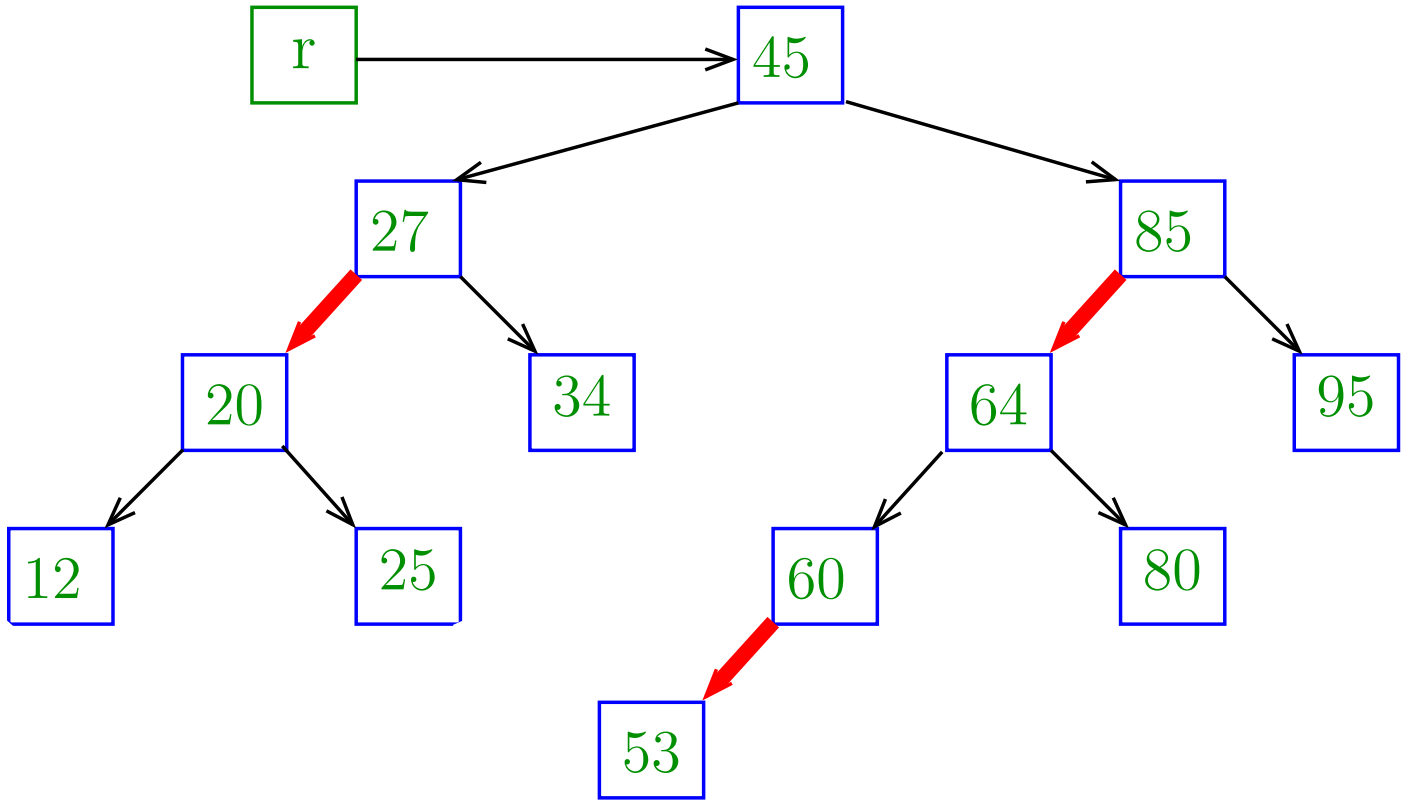

```
return balance(h);
```

rubro-negra: outro `deleteMin()`

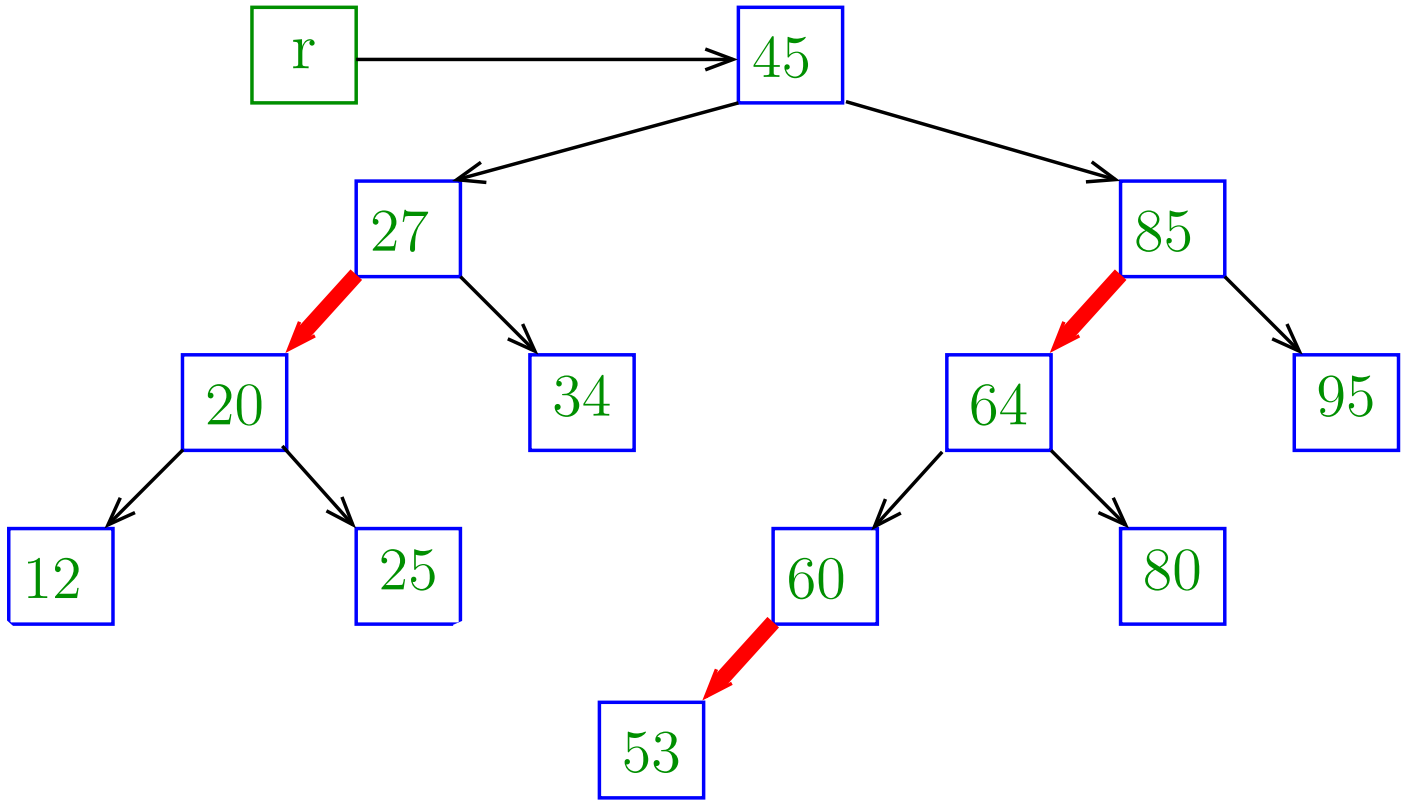



```
return balance(h);
```

rubro-negra: outro `deleteMin()`

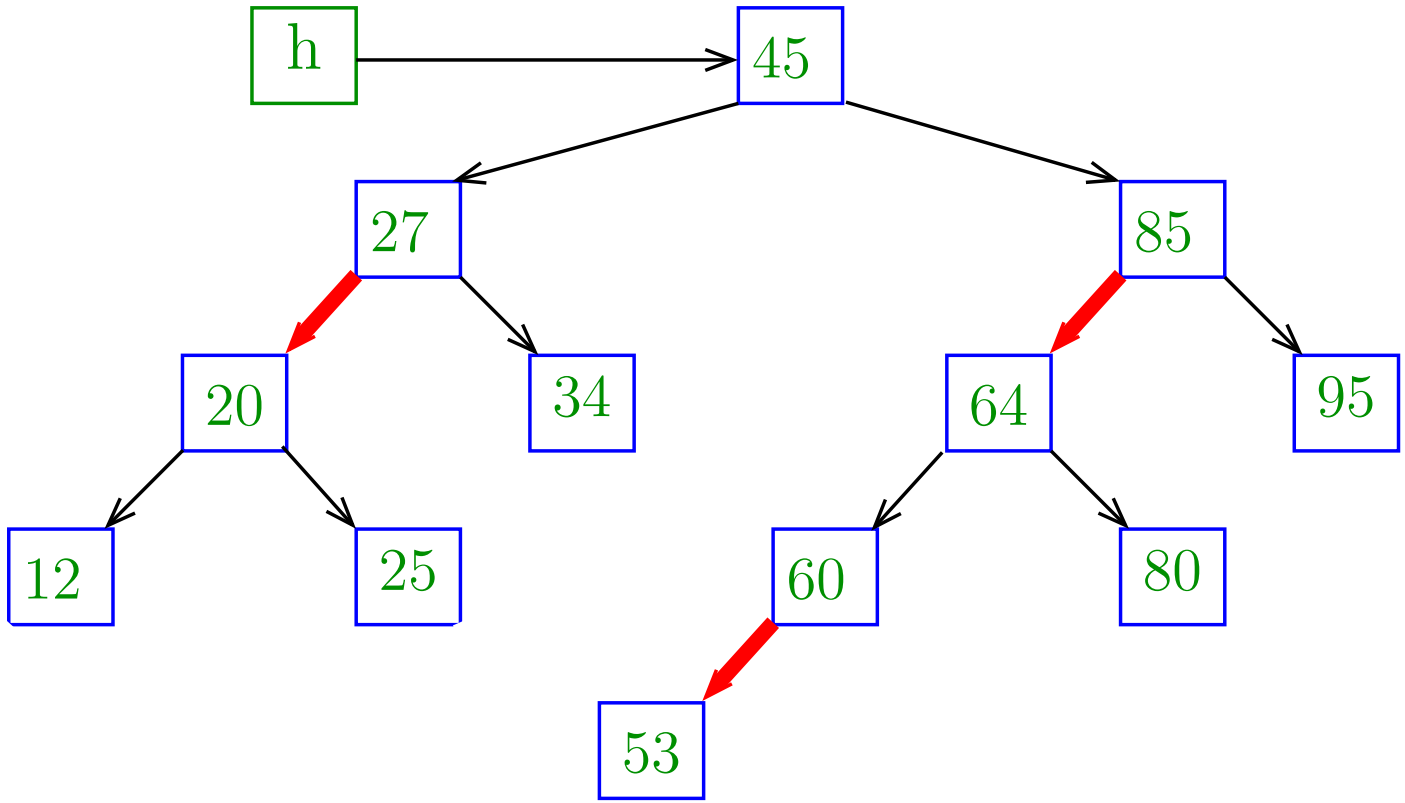


rubro-negra: mais outro `deleteMin()`



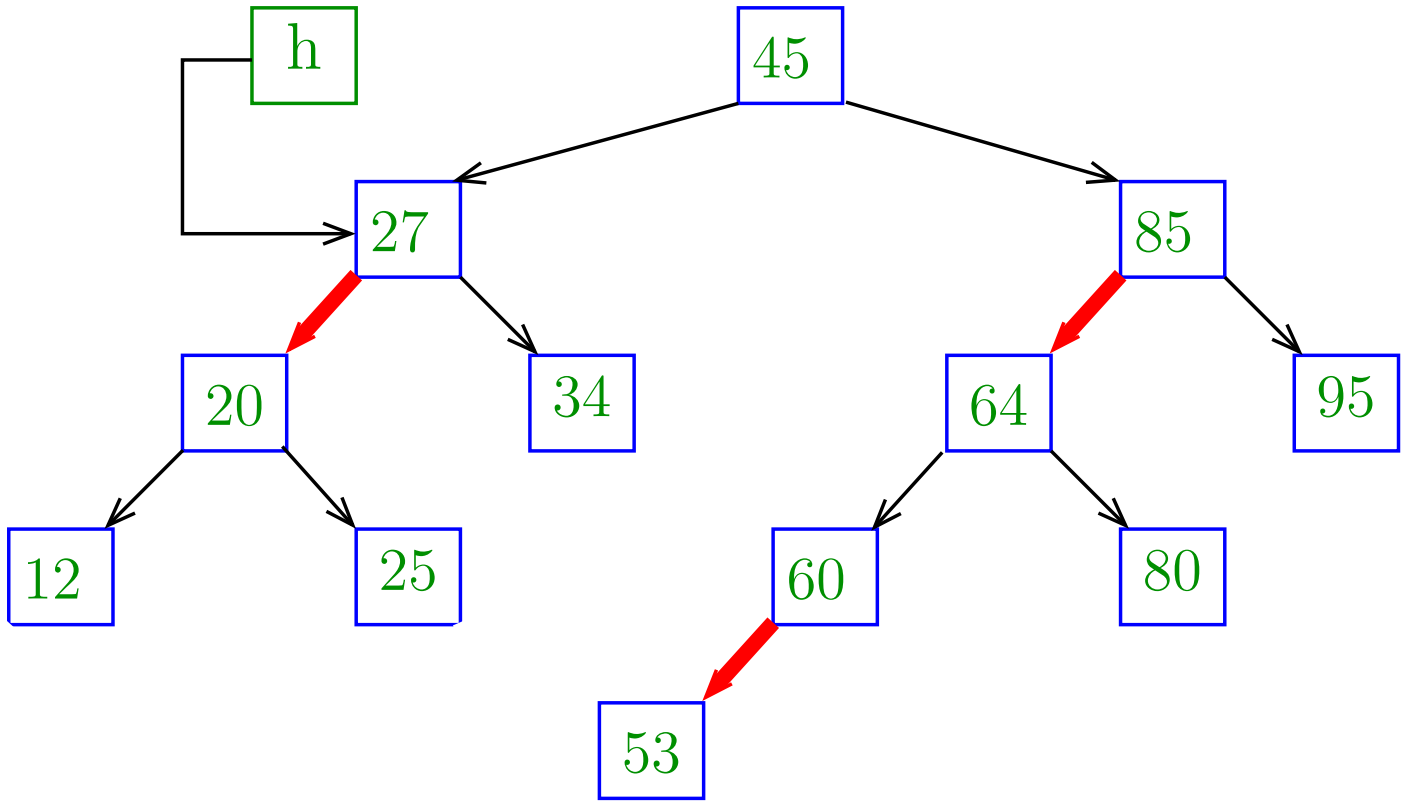
`deleteMin(r);`

rubro-negra: mais outro `deleteMin()`



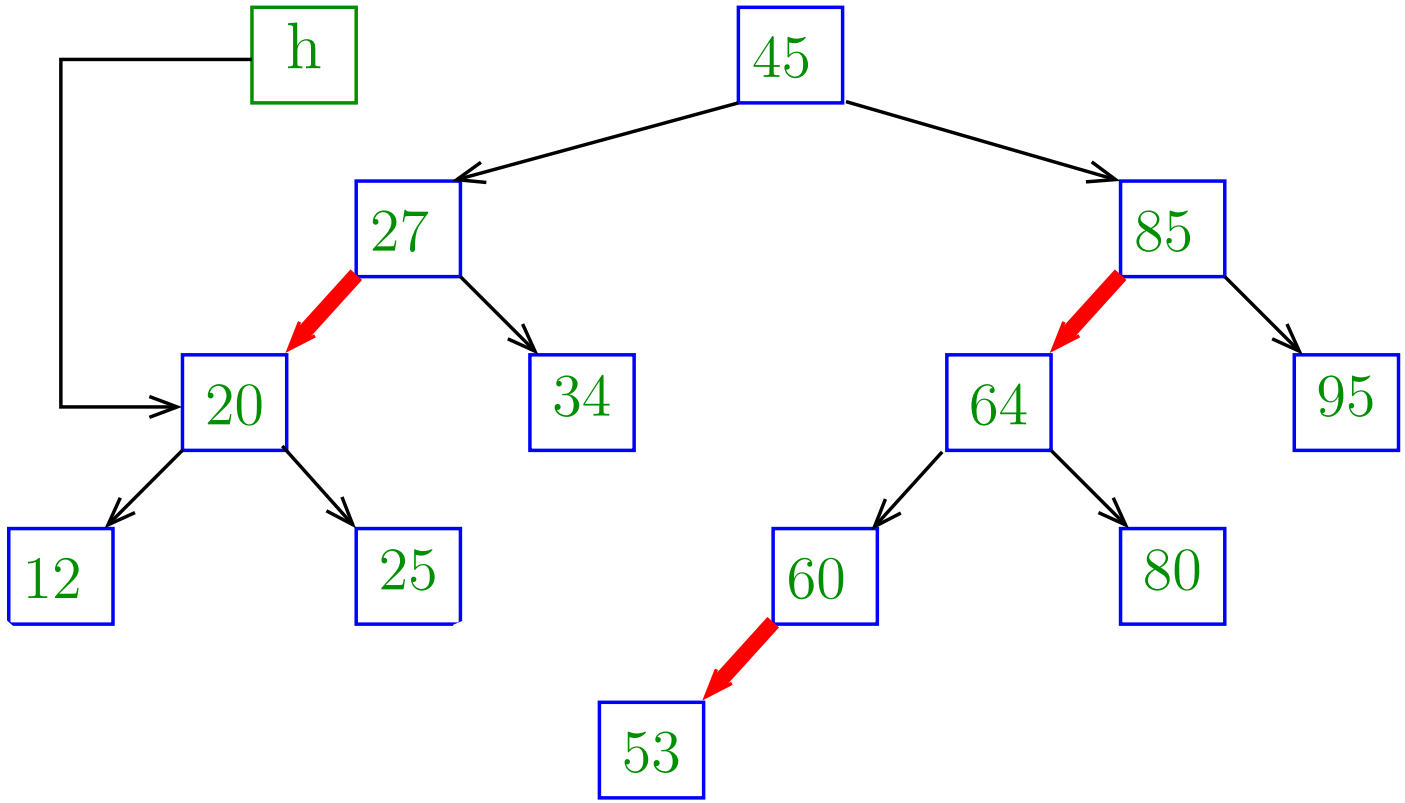

```
h->left = deleteMin(h->left);
```

rubro-negra: mais outro `deleteMin()`



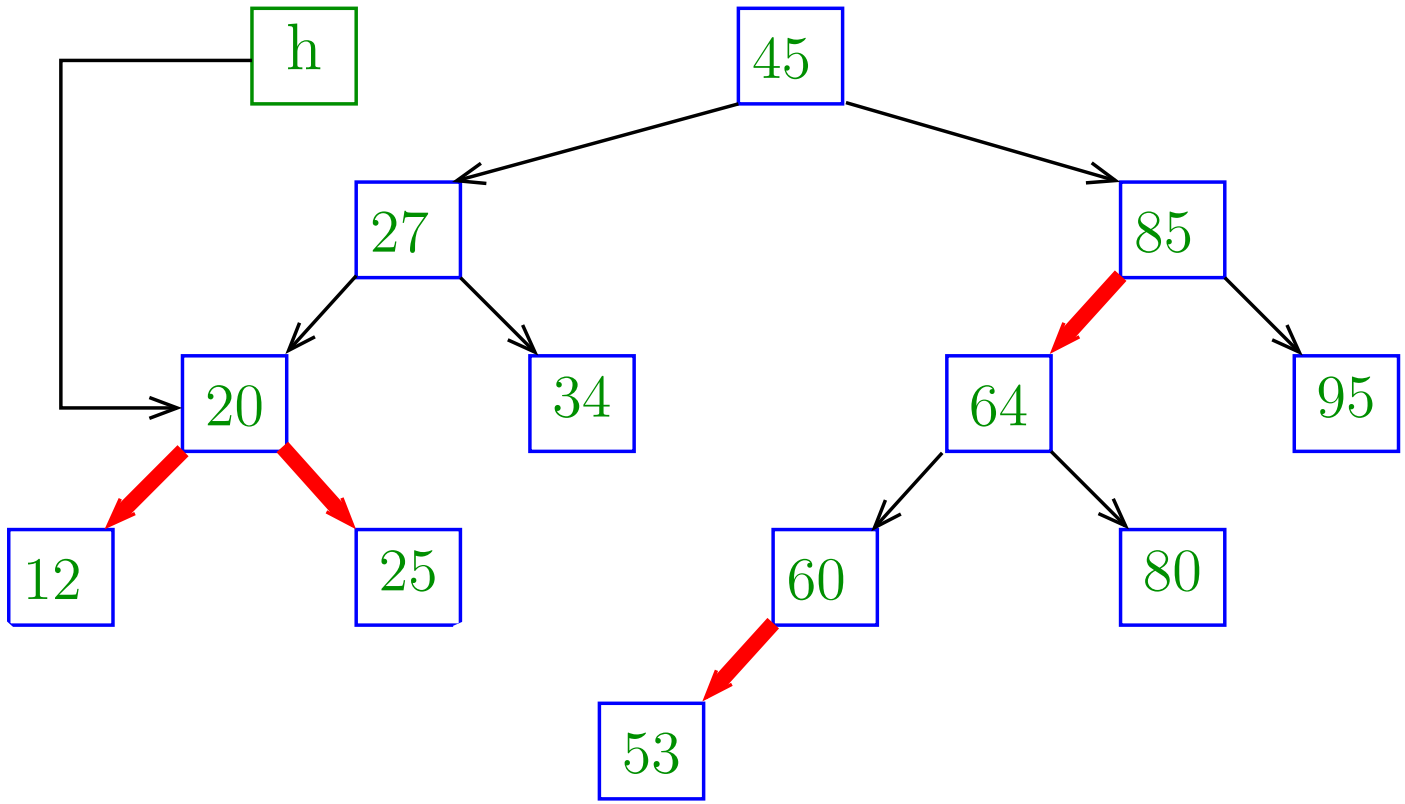

```
h->left = deleteMin(h->left);
```

rubro-negra: mais outro `deleteMin()`



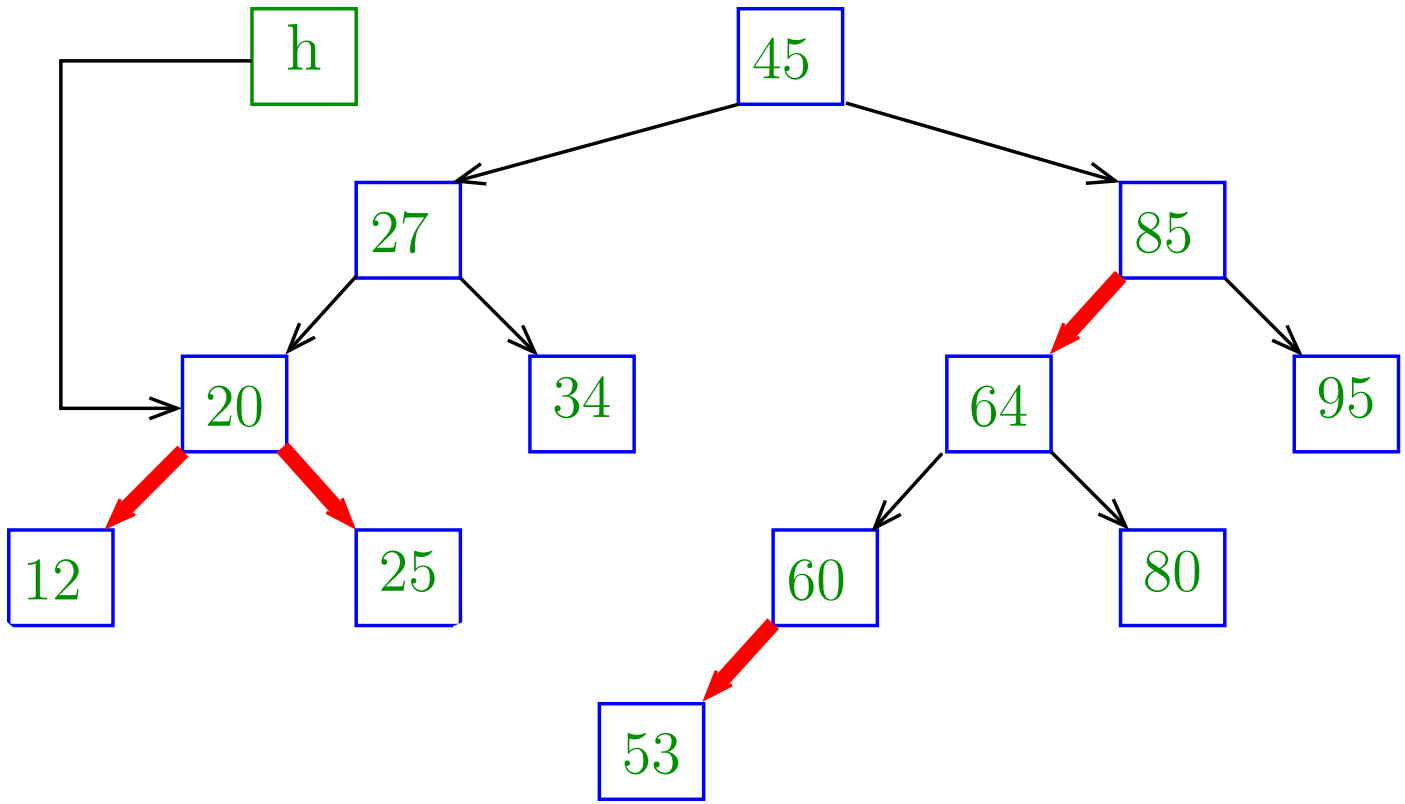
`h = moveRedLeft(h);`

rubro-negra: mais outro `deleteMin()`



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

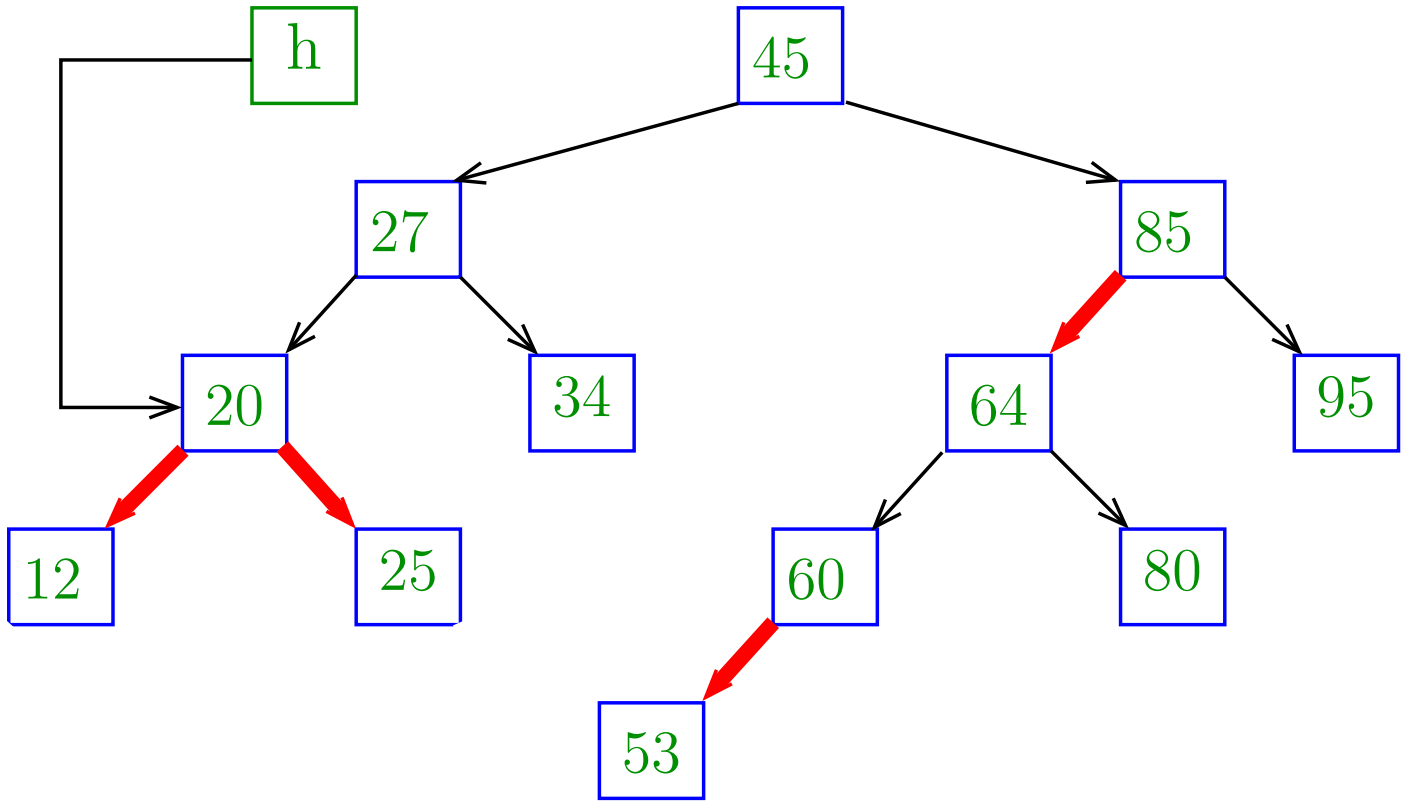
rubro-negra: mais outro `deleteMin()`



`return h`

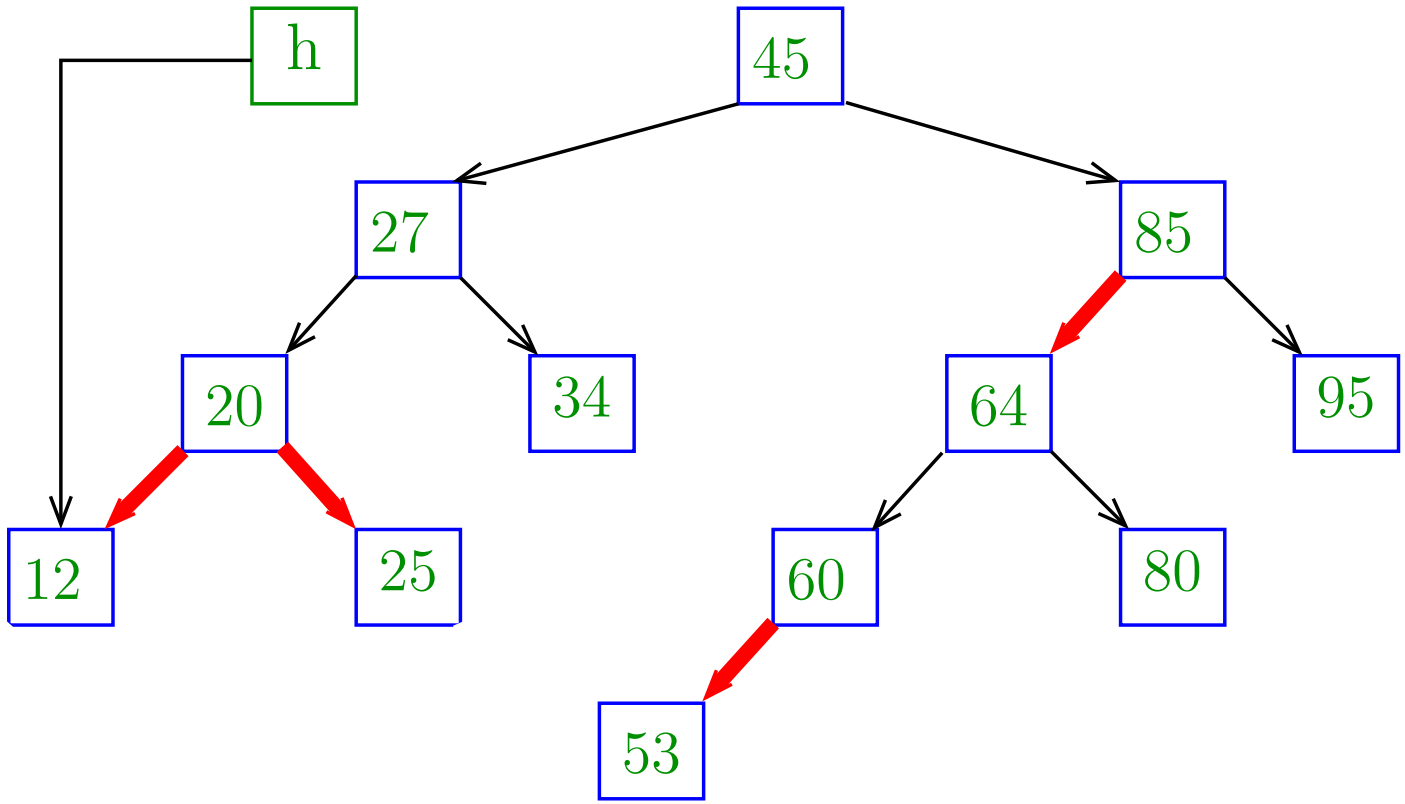
[`moveRedLeft(h);`]

rubro-negra: mais outro `deleteMin()`



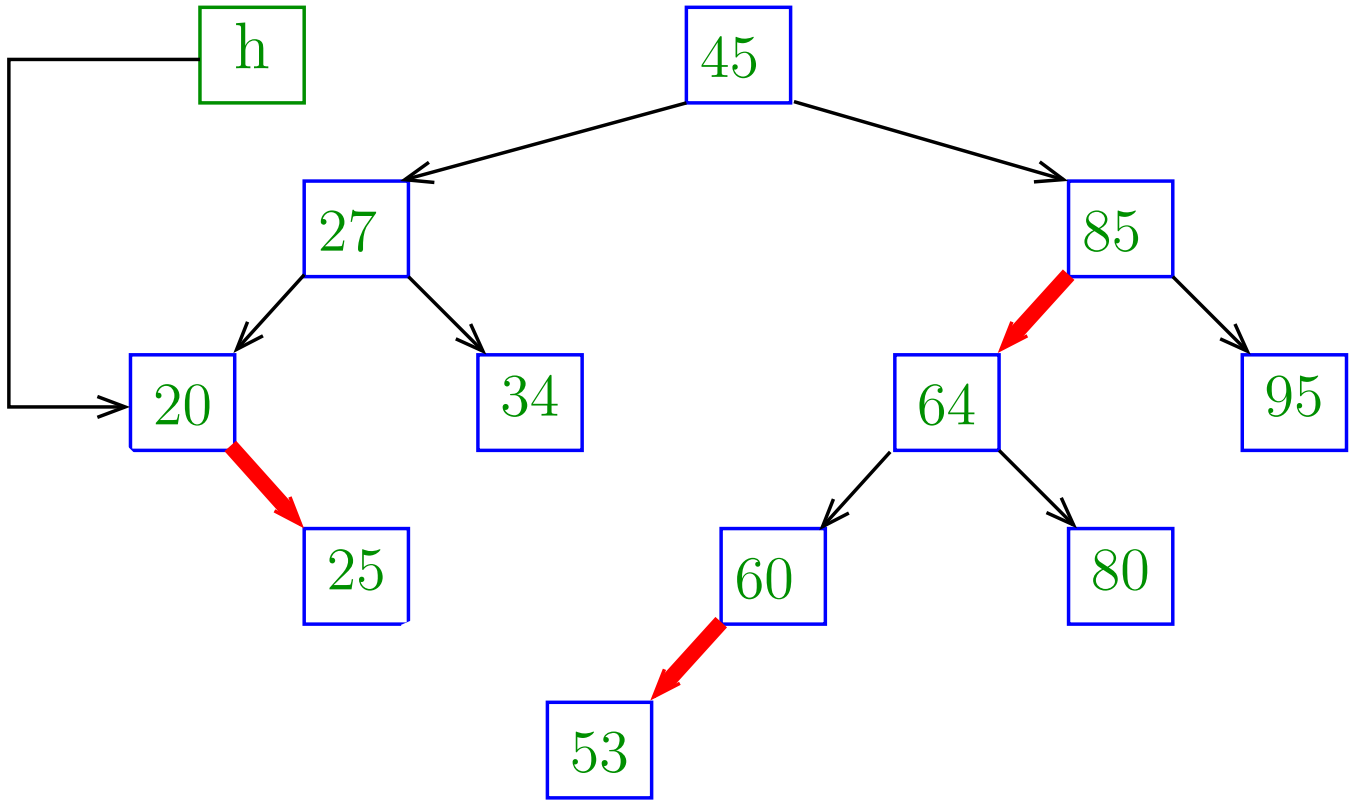

```
h->left = deleteMin(h->left);
```

rubro-negra: mais outro `deleteMin()`



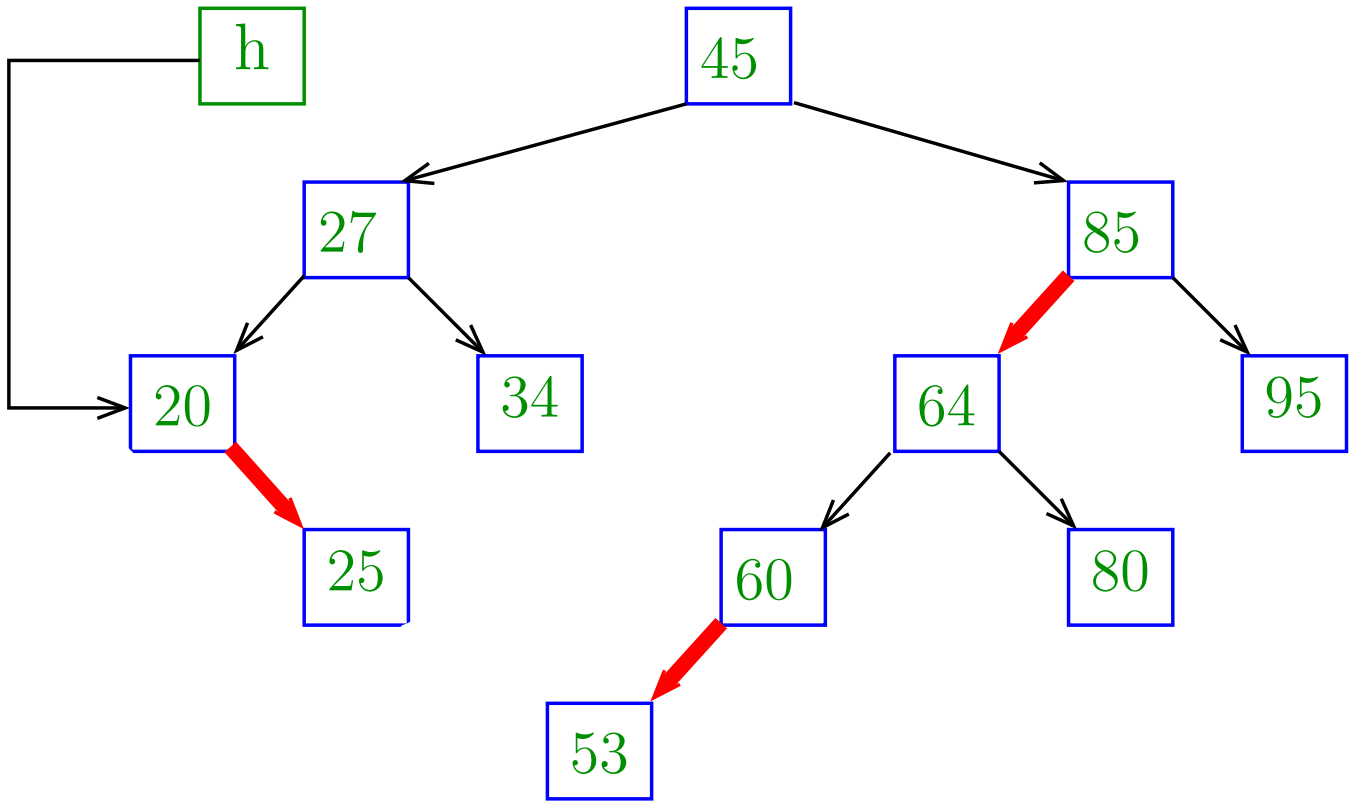
`return NULL`

rubro-negra: mais outro `deleteMin()`



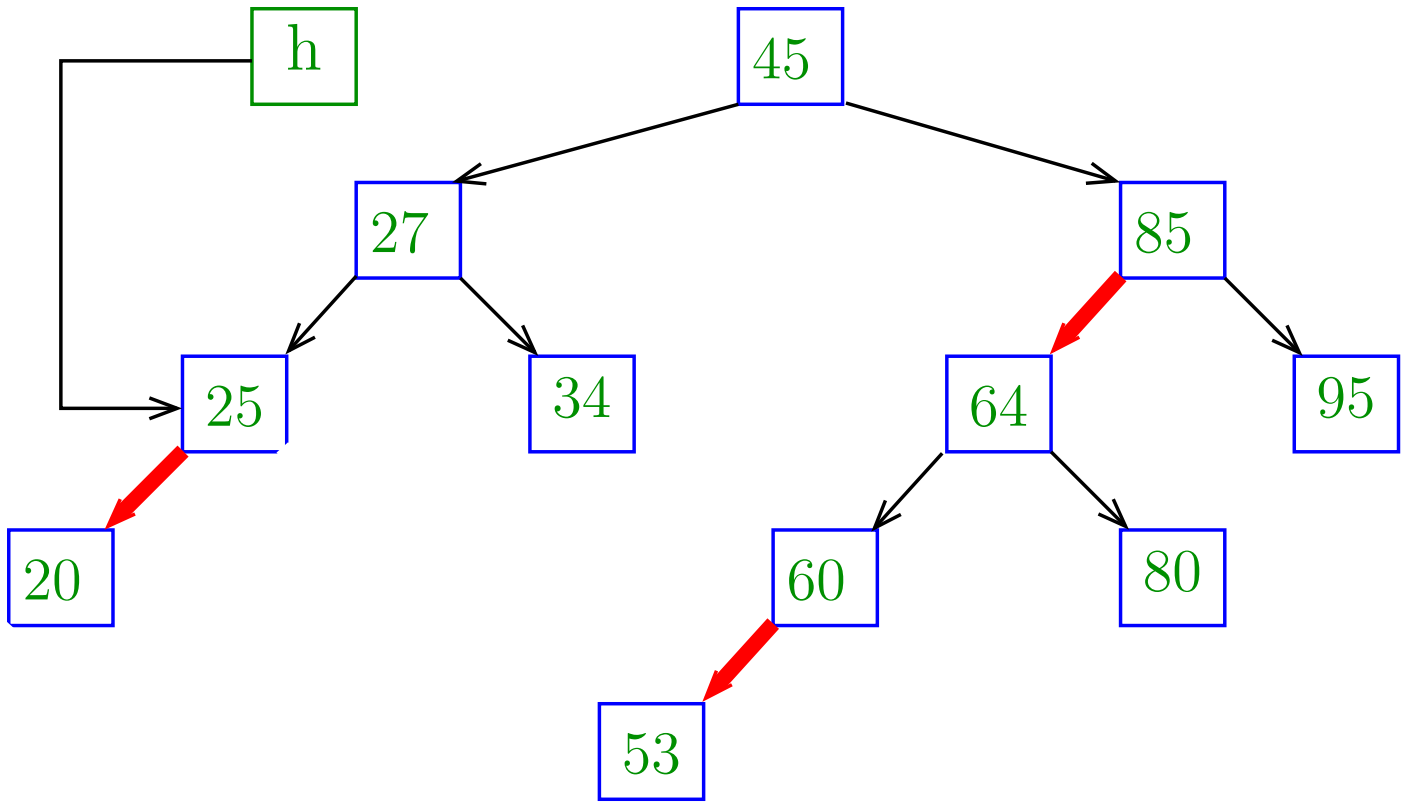

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()`



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

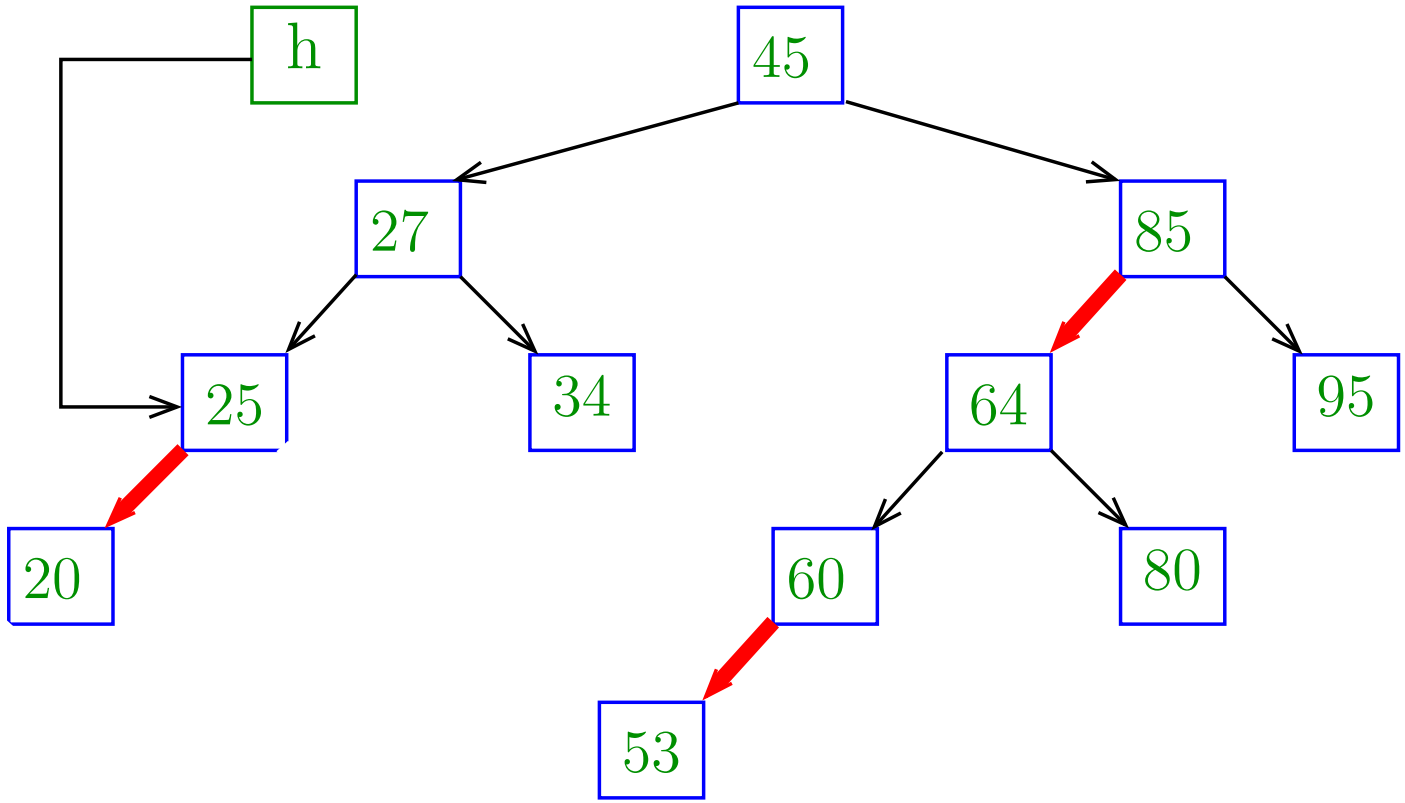
rubro-negra: mais outro `deleteMin()`



`return h;`

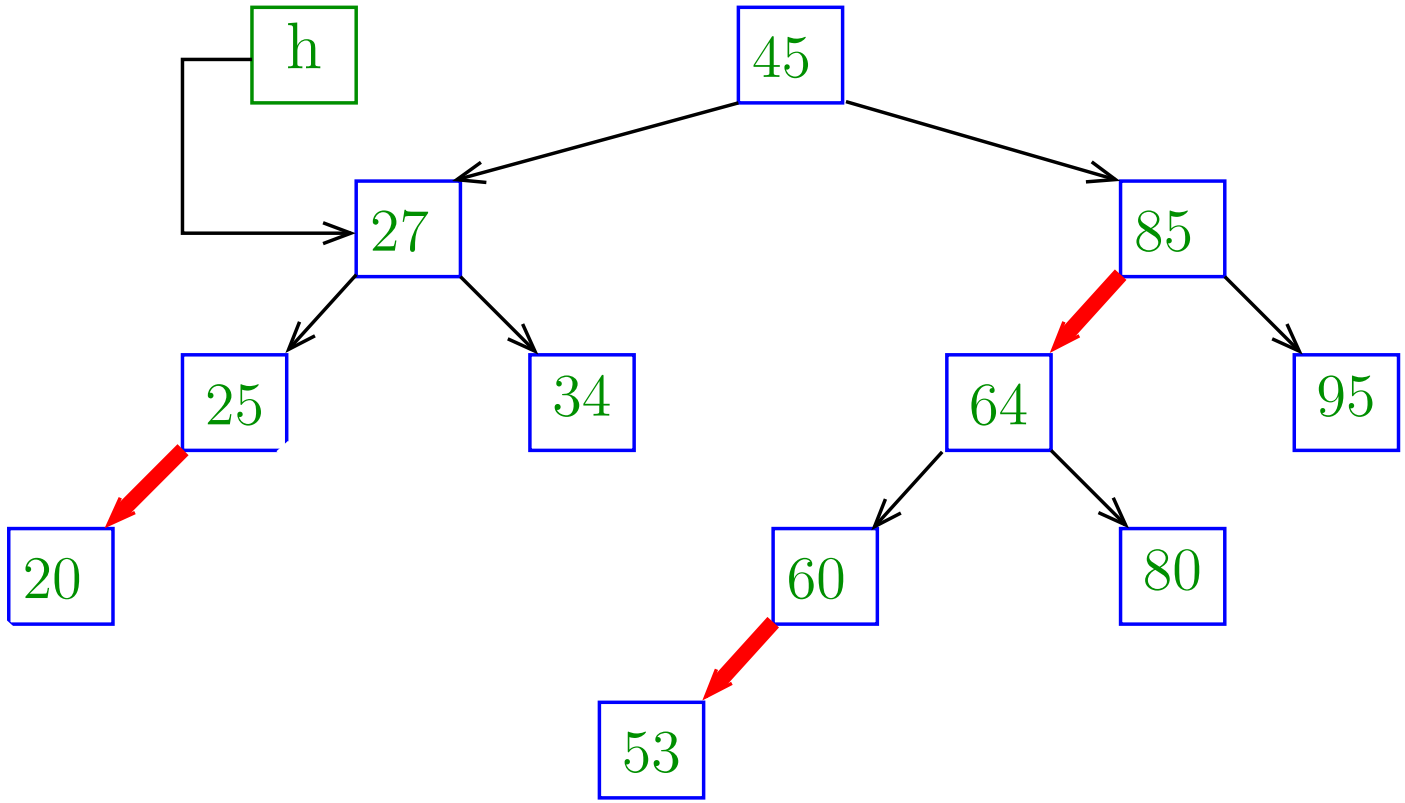
[`balance(h);`]

rubro-negra: mais outro `deleteMin()`



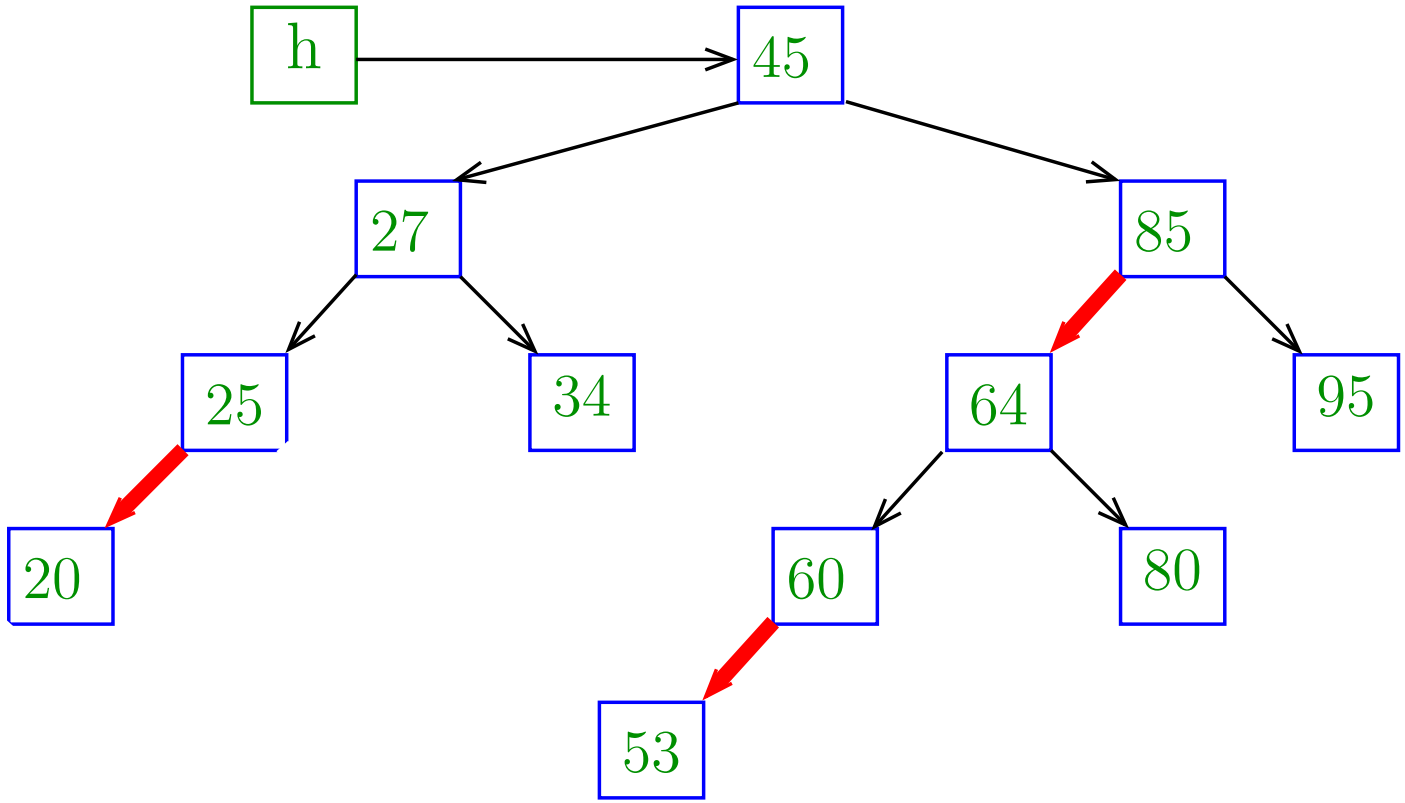

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()`



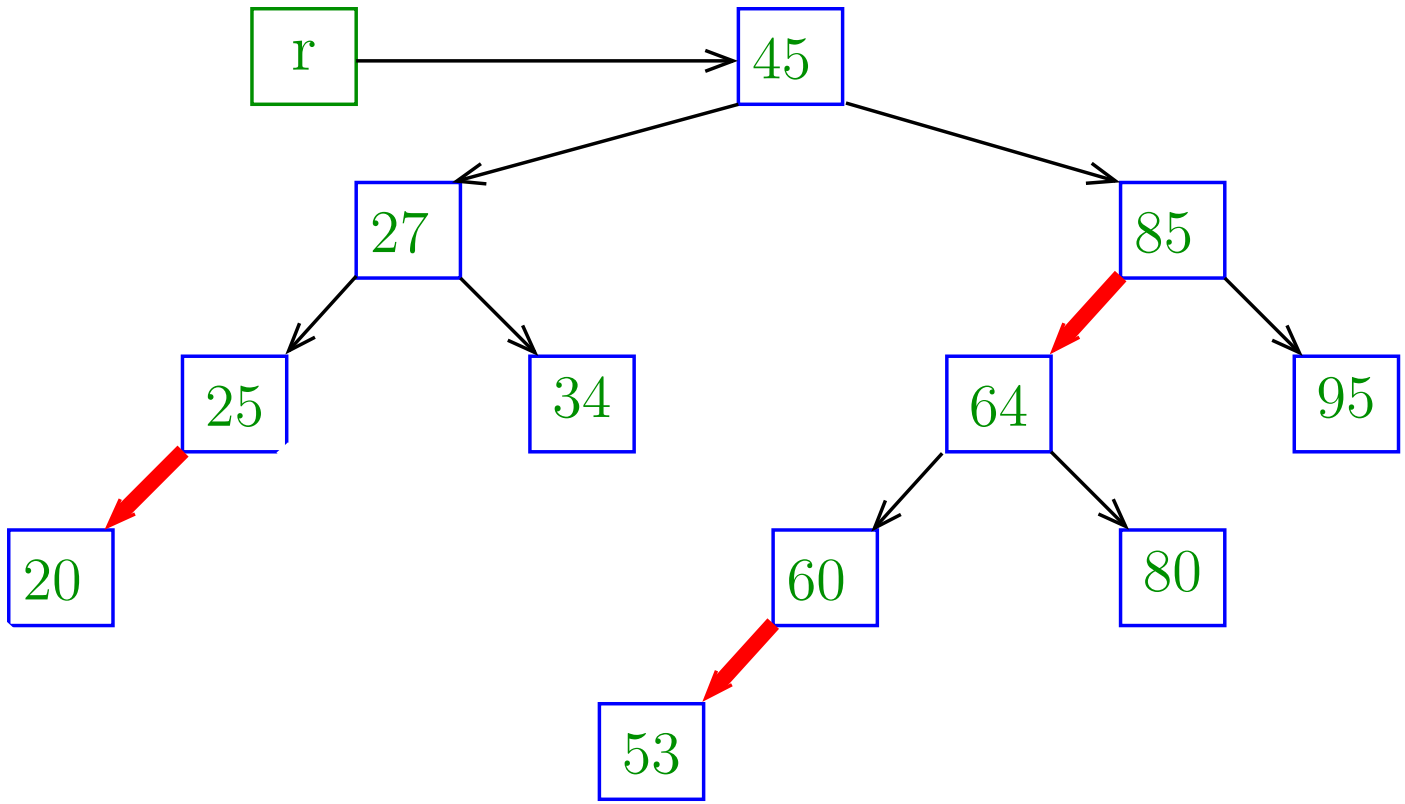

```
return balance(h);
```

rubro-negra: mais outro `deleteMin()`

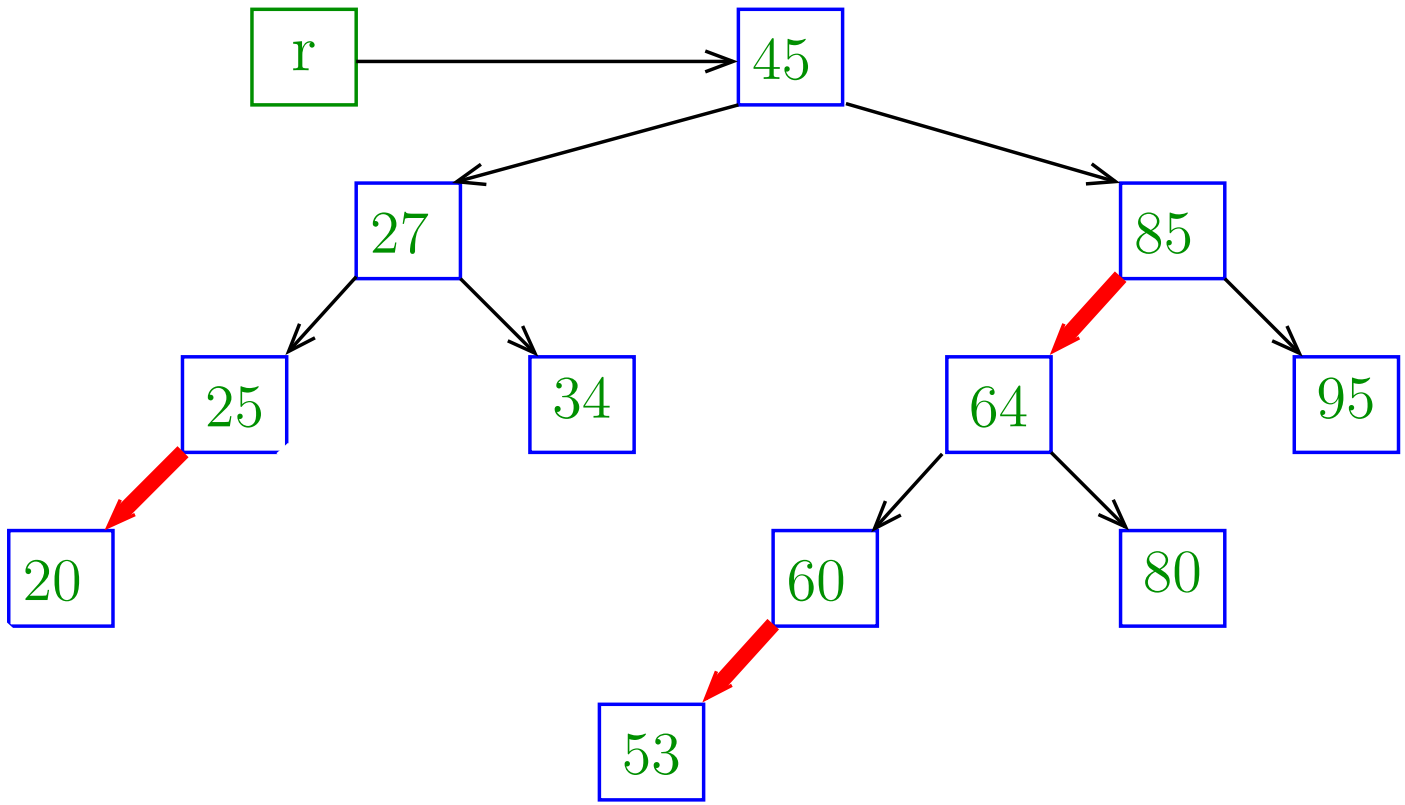



```
return balance(h);
```

rubro-negra: mais outro `deleteMin()`

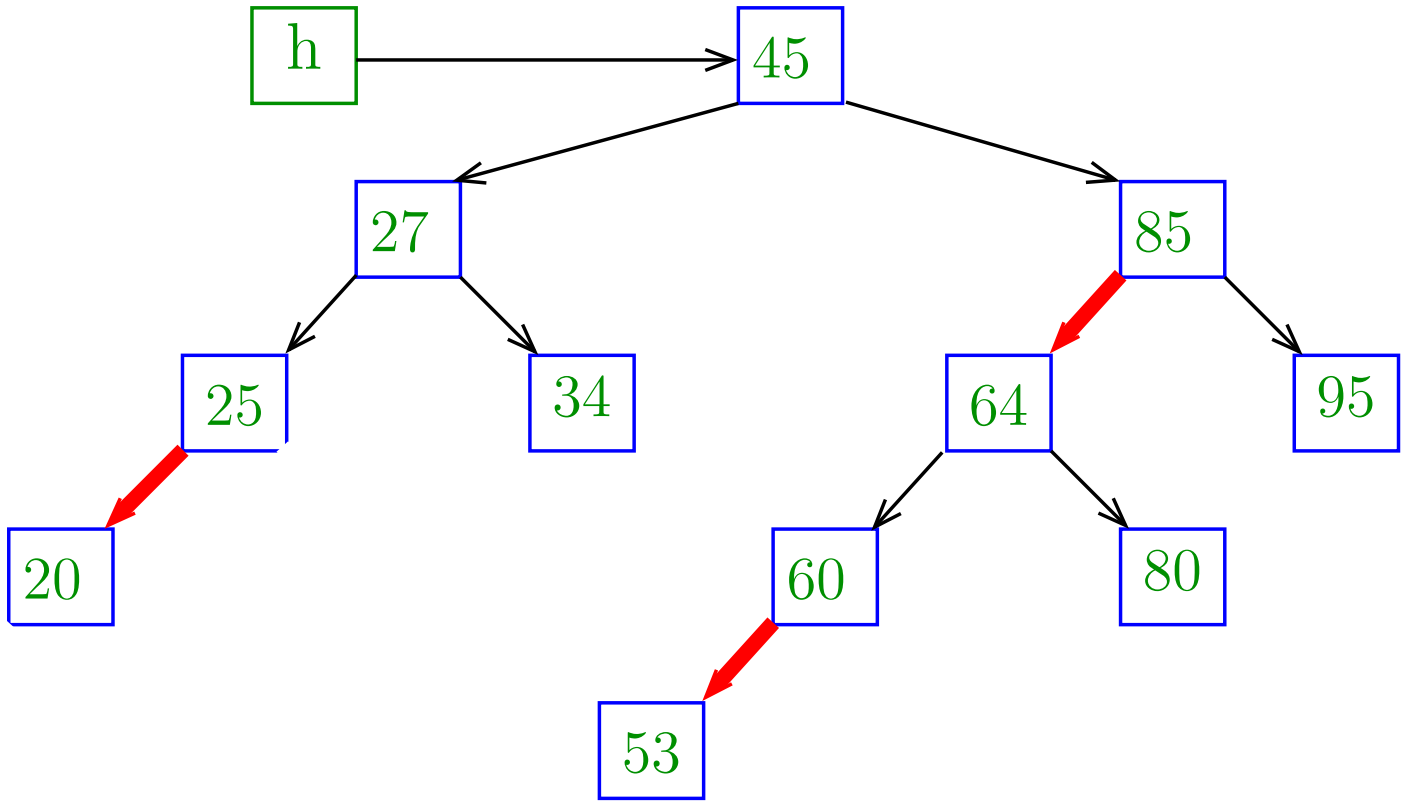


rubro-negra: mais outro `deleteMin()` ainda



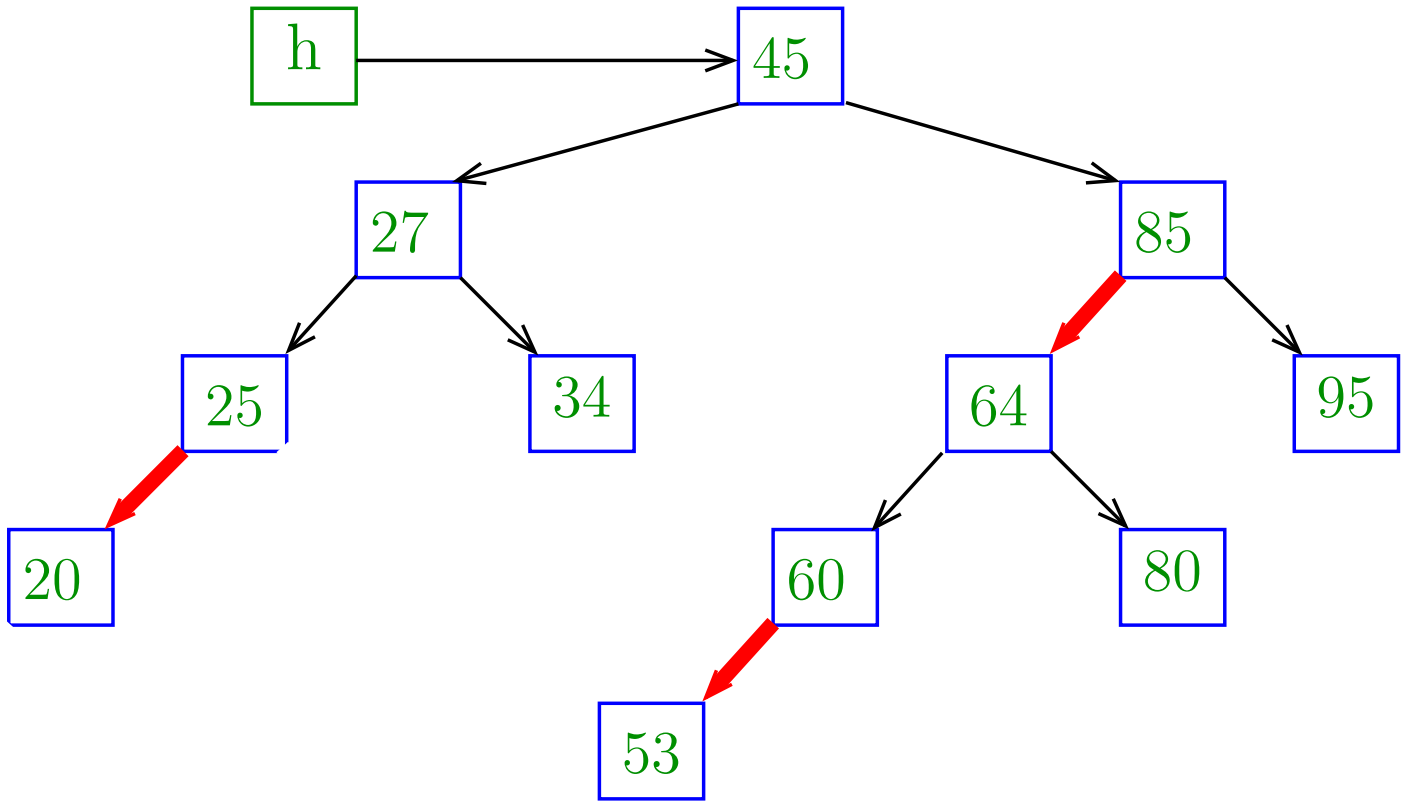
`deleteMin(r);`

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



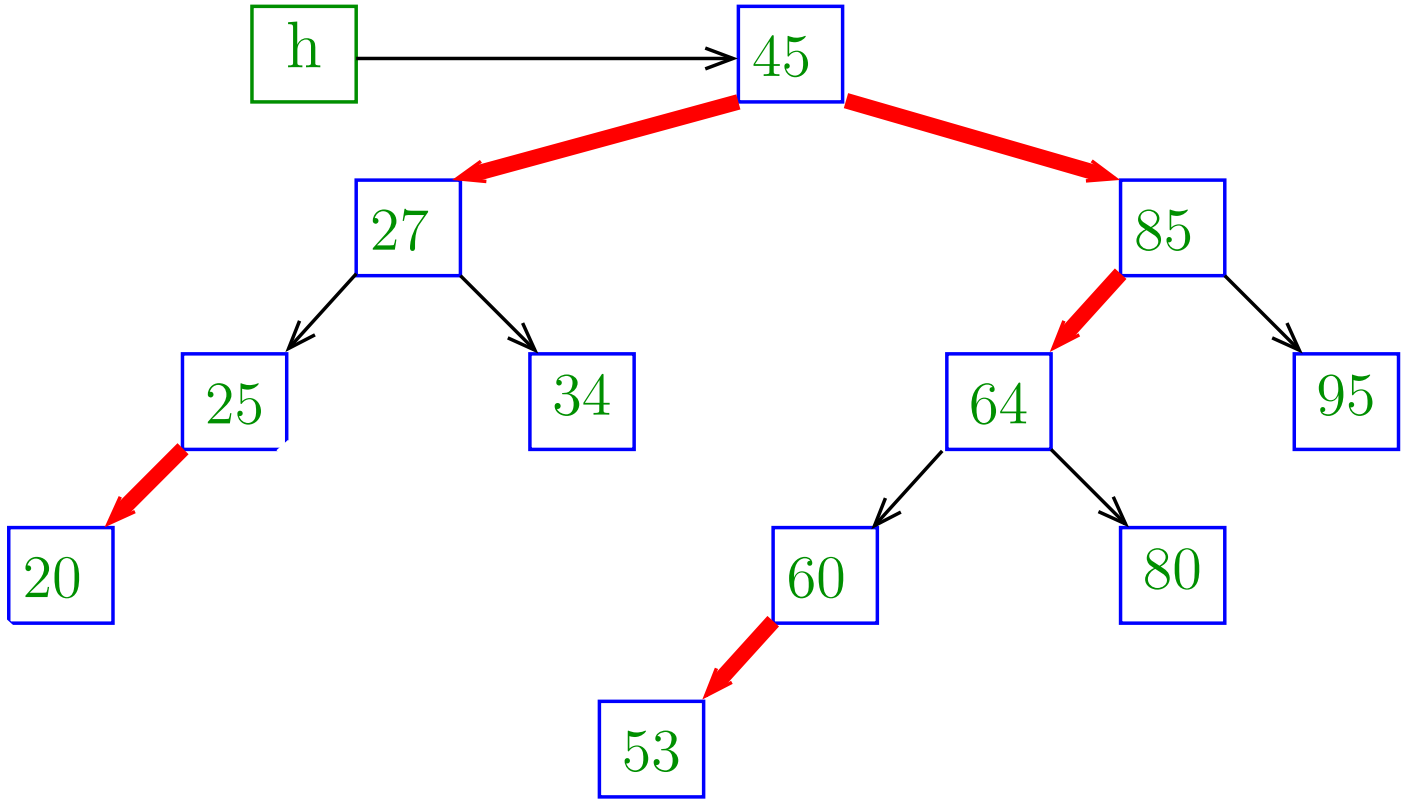

```
h = moveRedLeft(h);
```

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



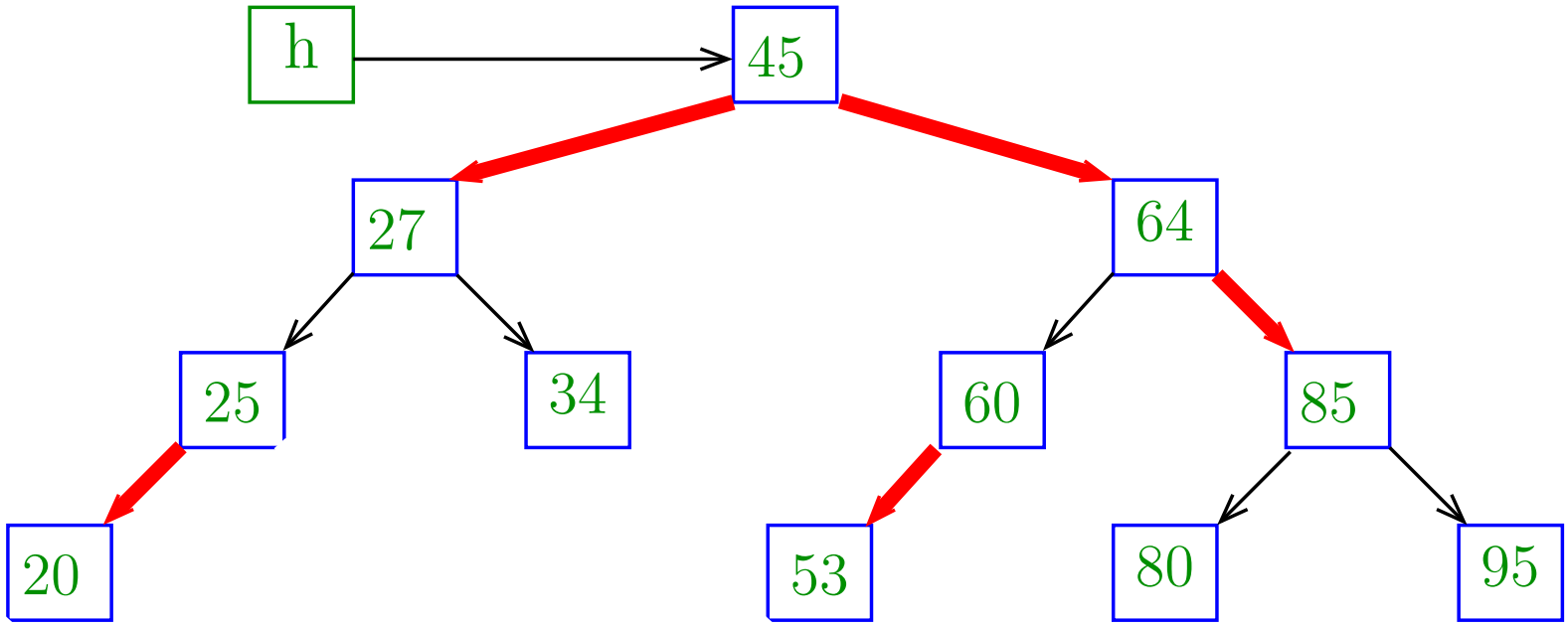
`flipColors(h)` [`moveRedLeft(h)`];

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



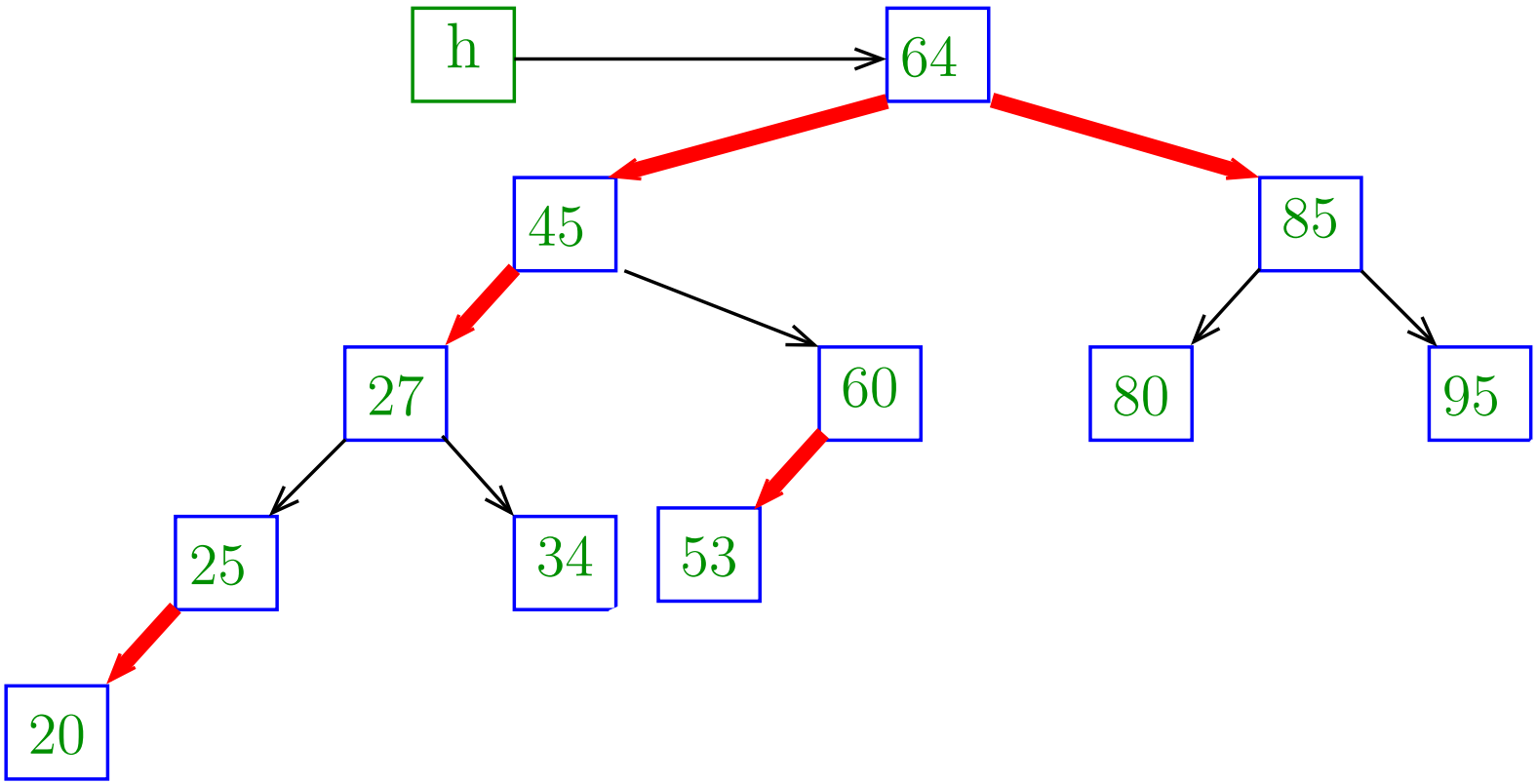

```
h->right = rotateRight(h->right);
[moveRedLeft(h);]
```

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



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

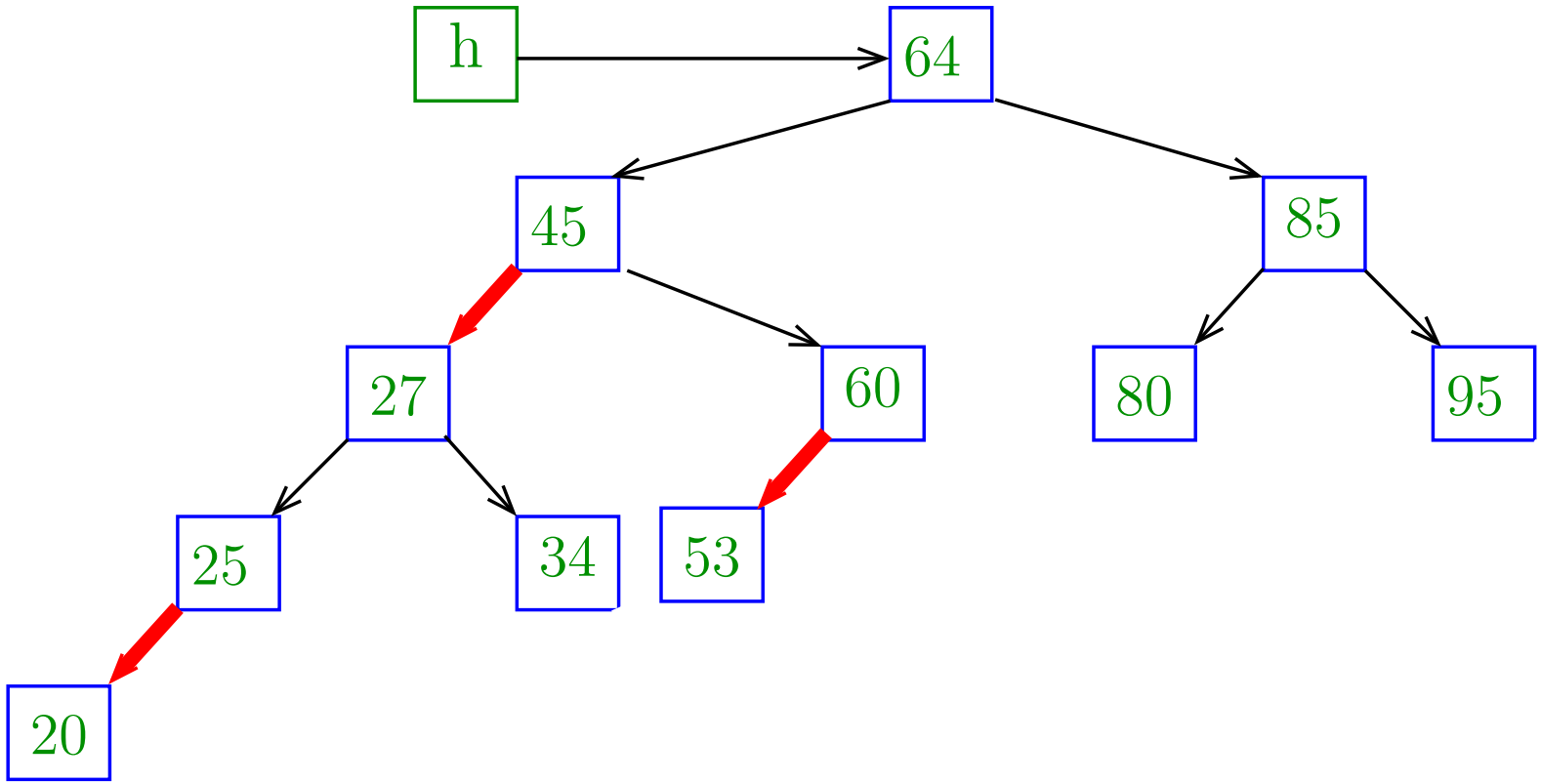
rubro-negra: mais outro `deleteMin()` ainda



`flipColors(h)`

[`moveRedLeft(h)`];

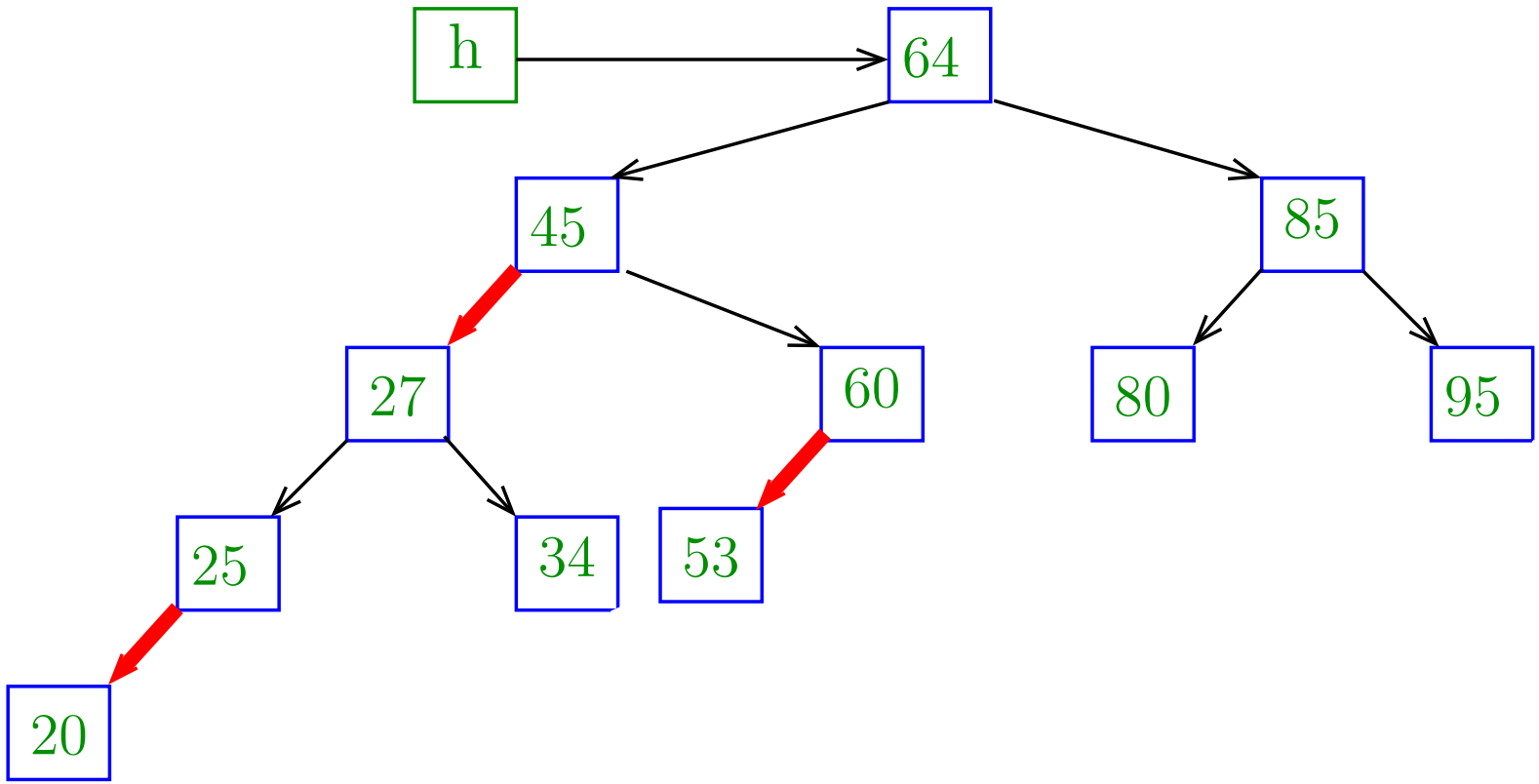
rubro-negra: mais outro `deleteMin()` ainda



`return h;`

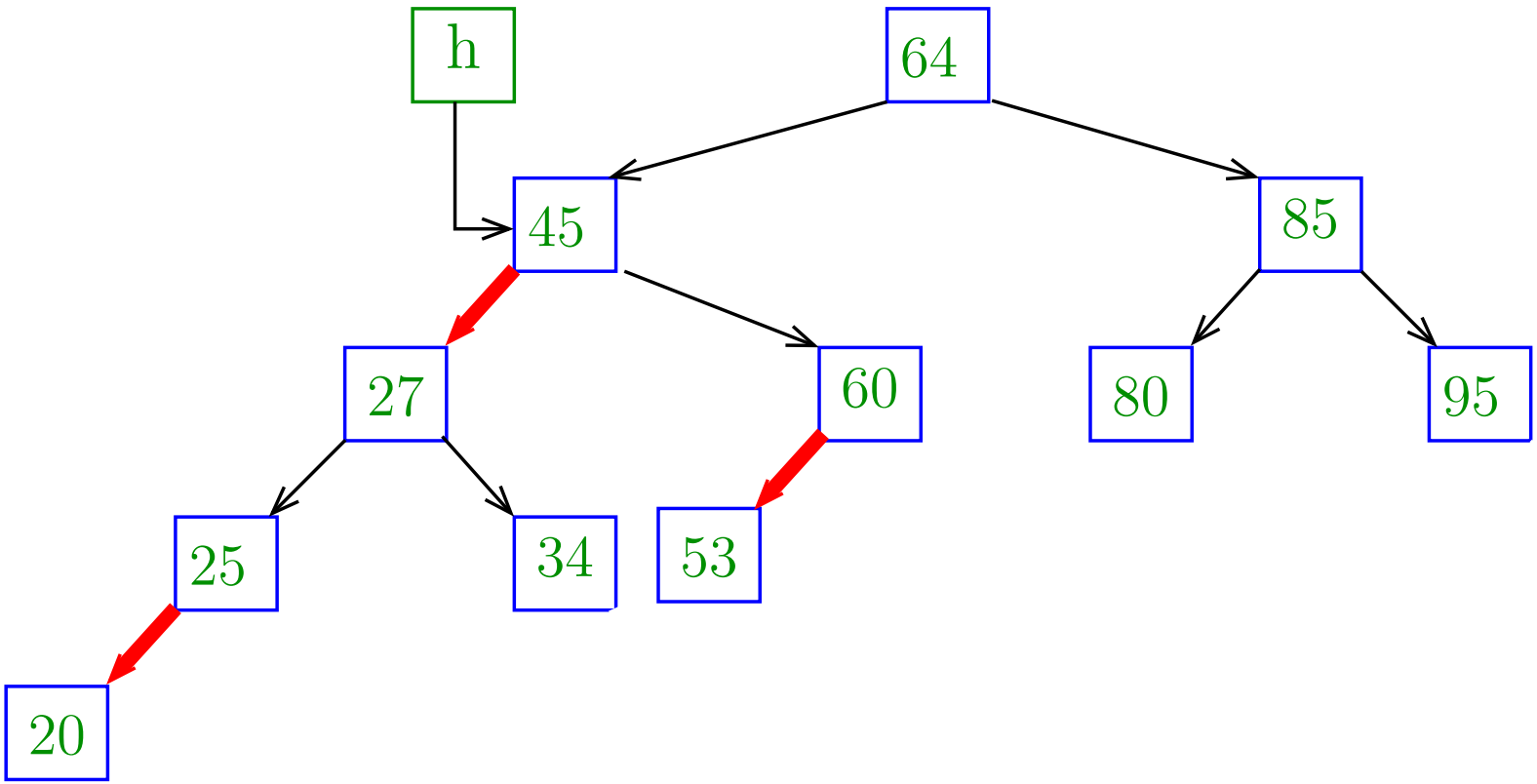
[`moveRedLeft(h)`];

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



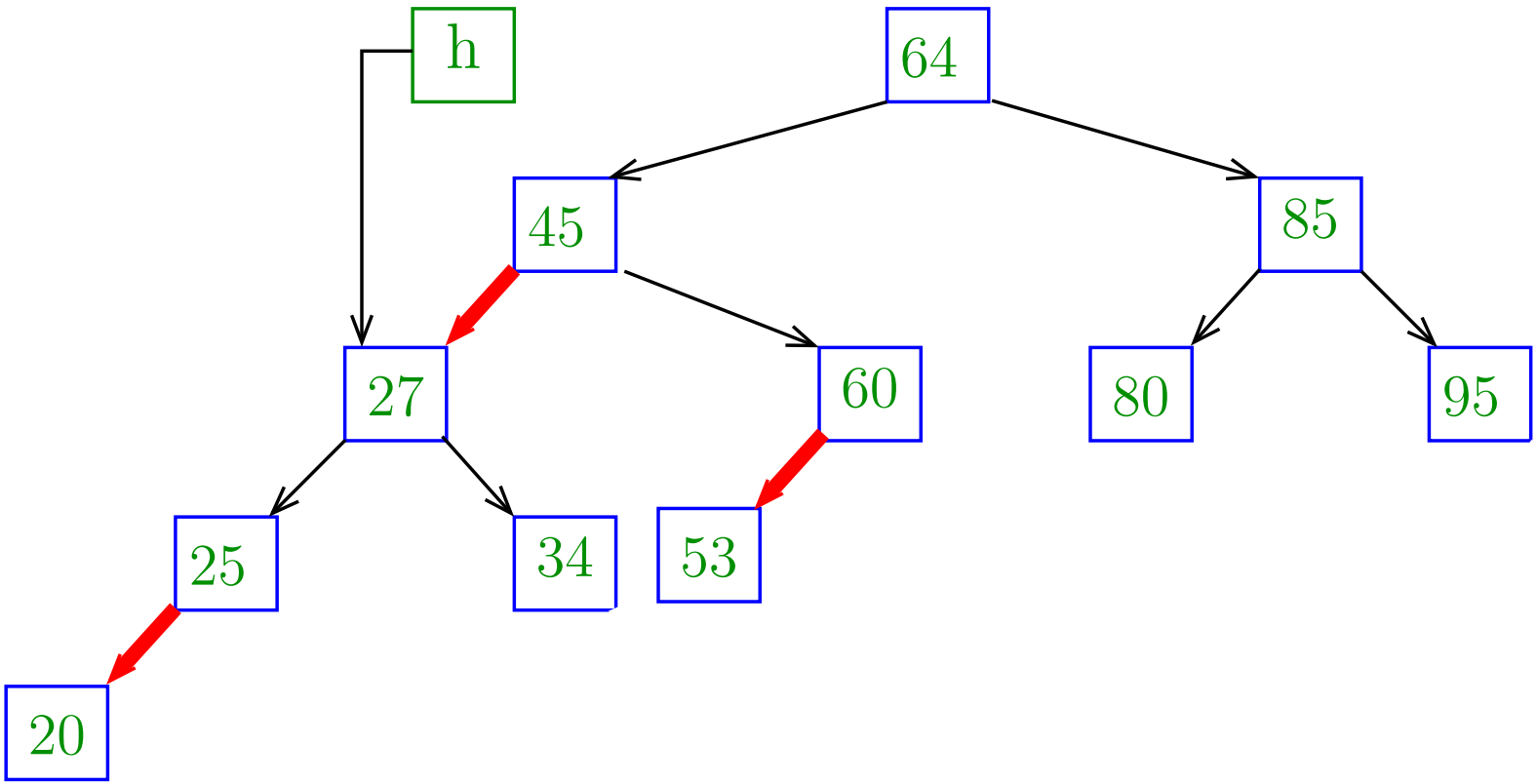

```
h->left = deleteMin(h->left);
```

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



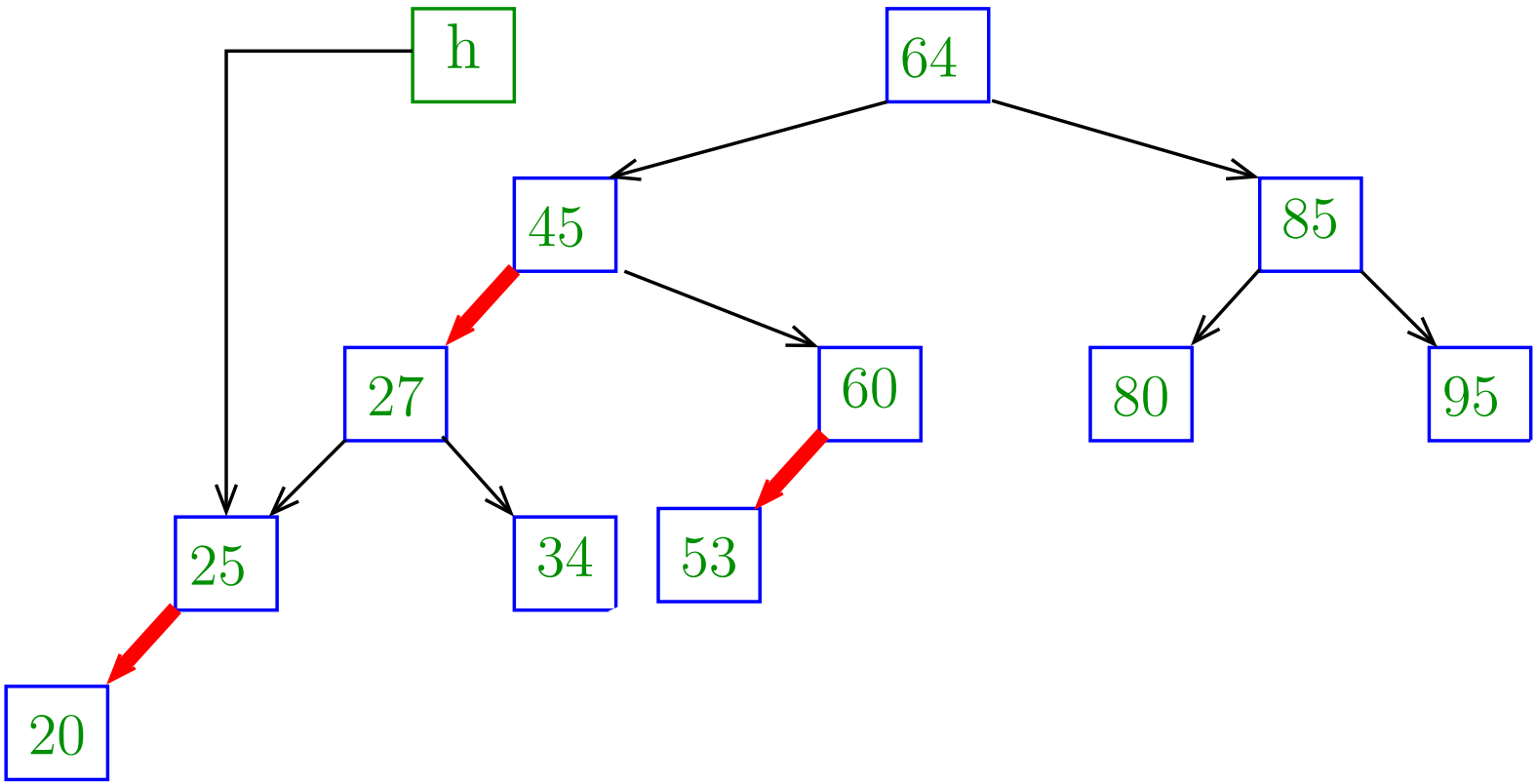

```
h->left = deleteMin(h->left);
```

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



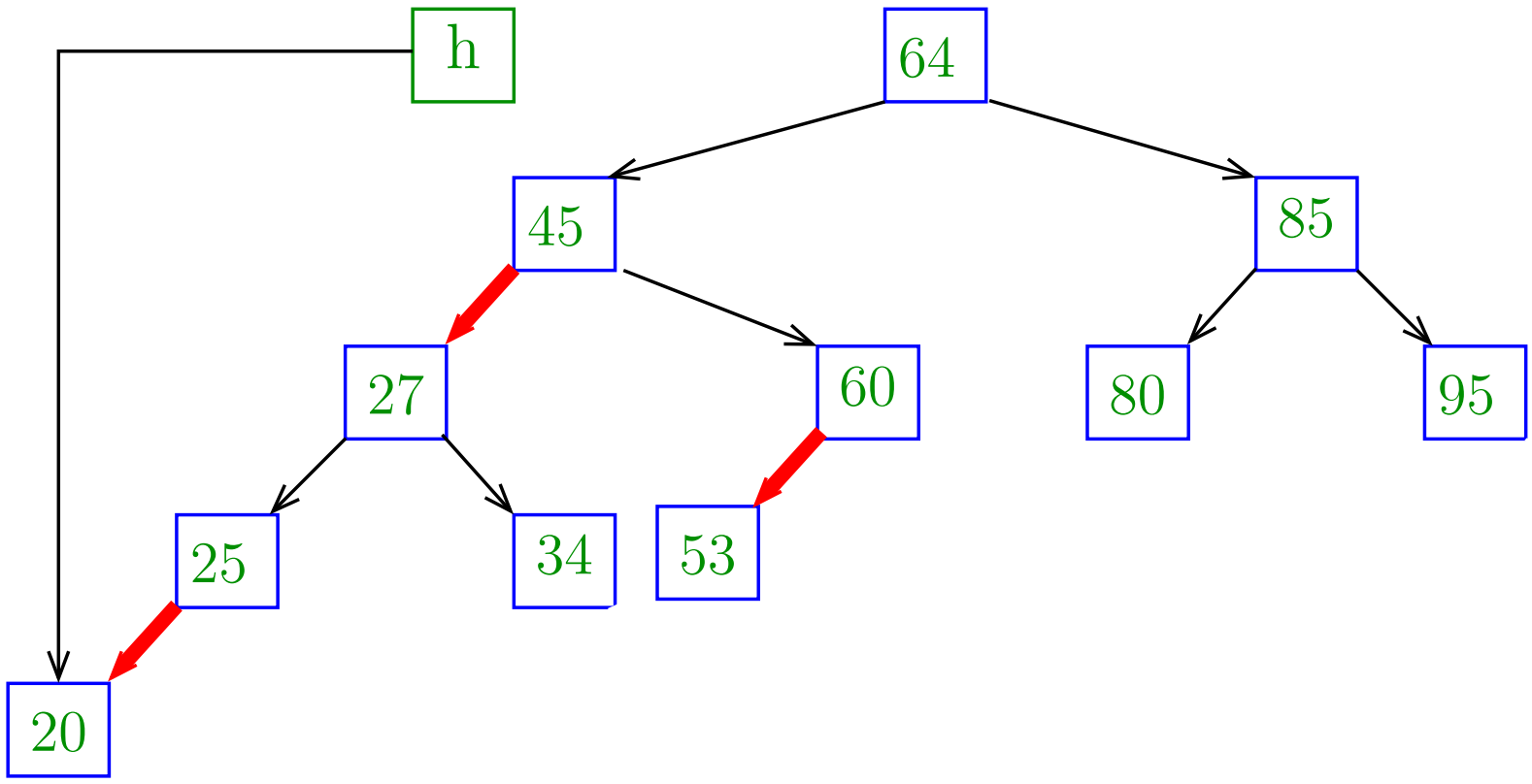

```
h->left = deleteMin(h->left);
```

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



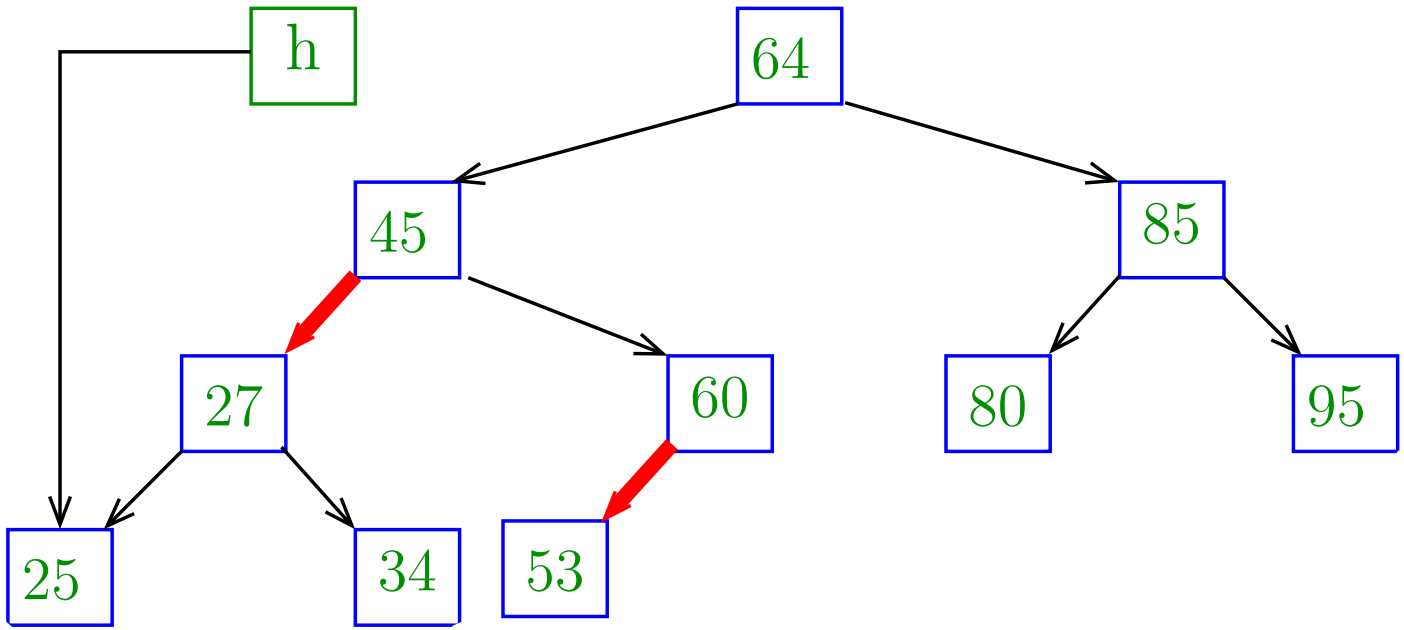

```
h->left = deleteMin(h->left);
```

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



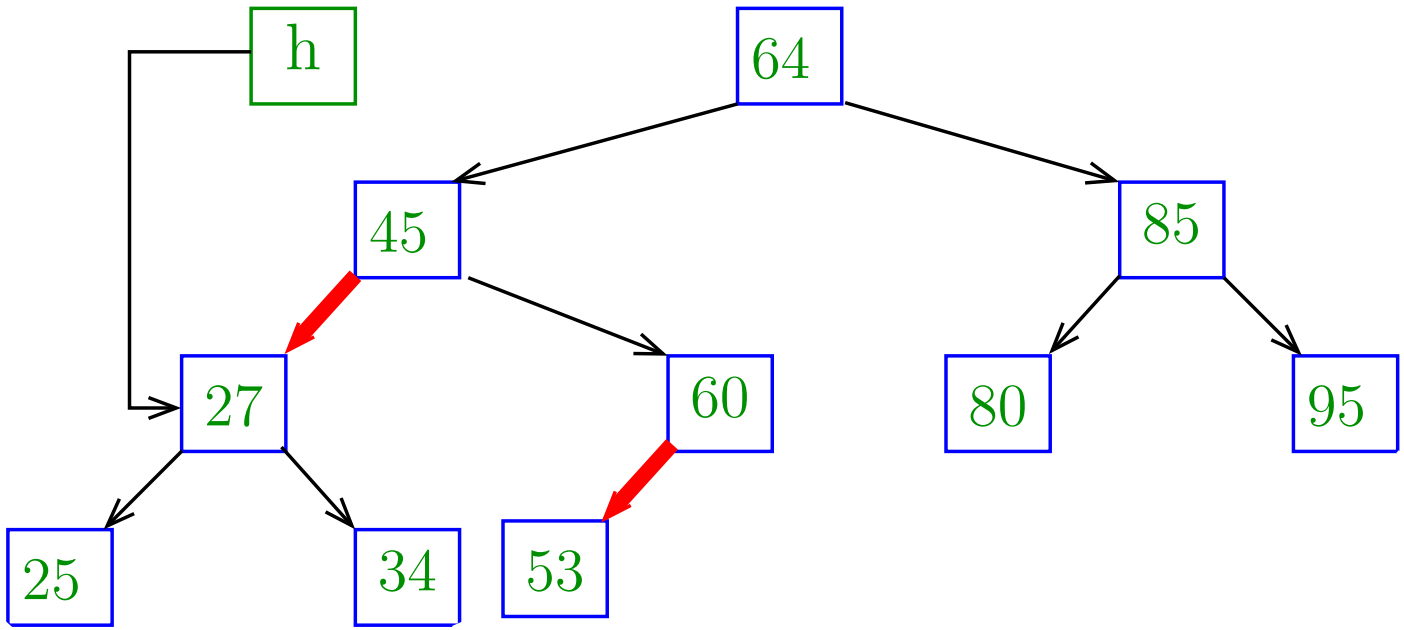

```
return NULL;
```

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



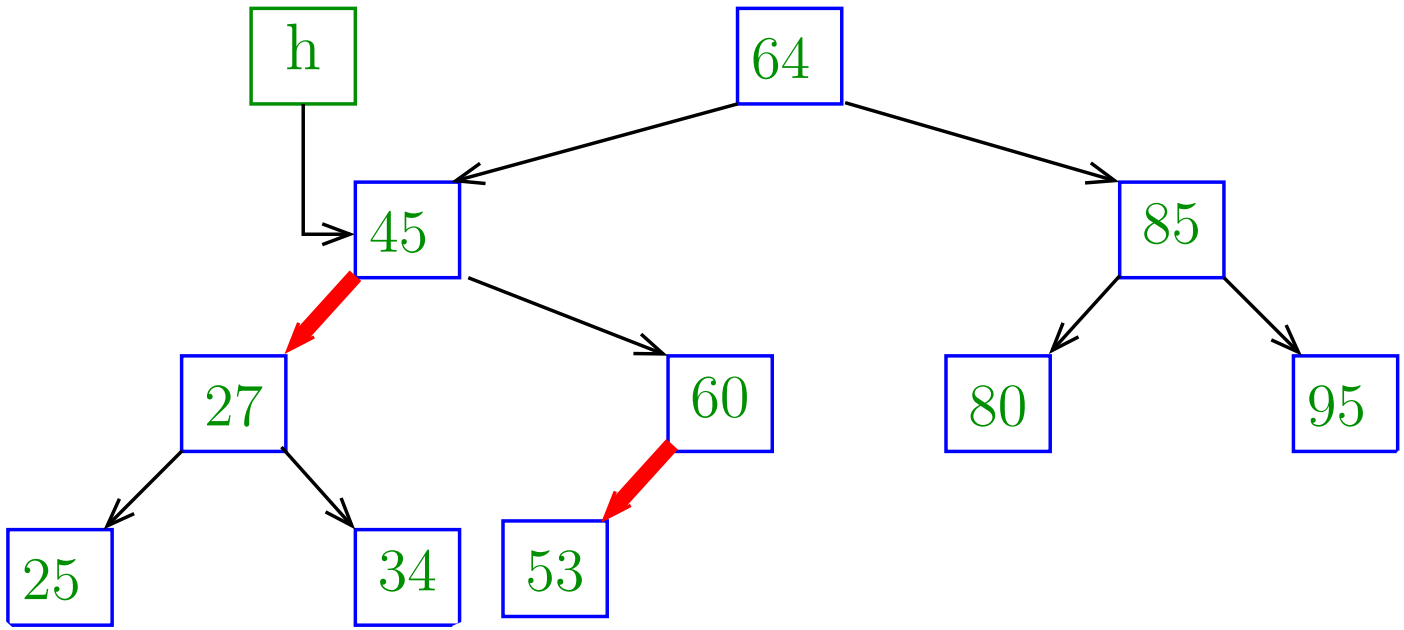

```
return balance(h);
```

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



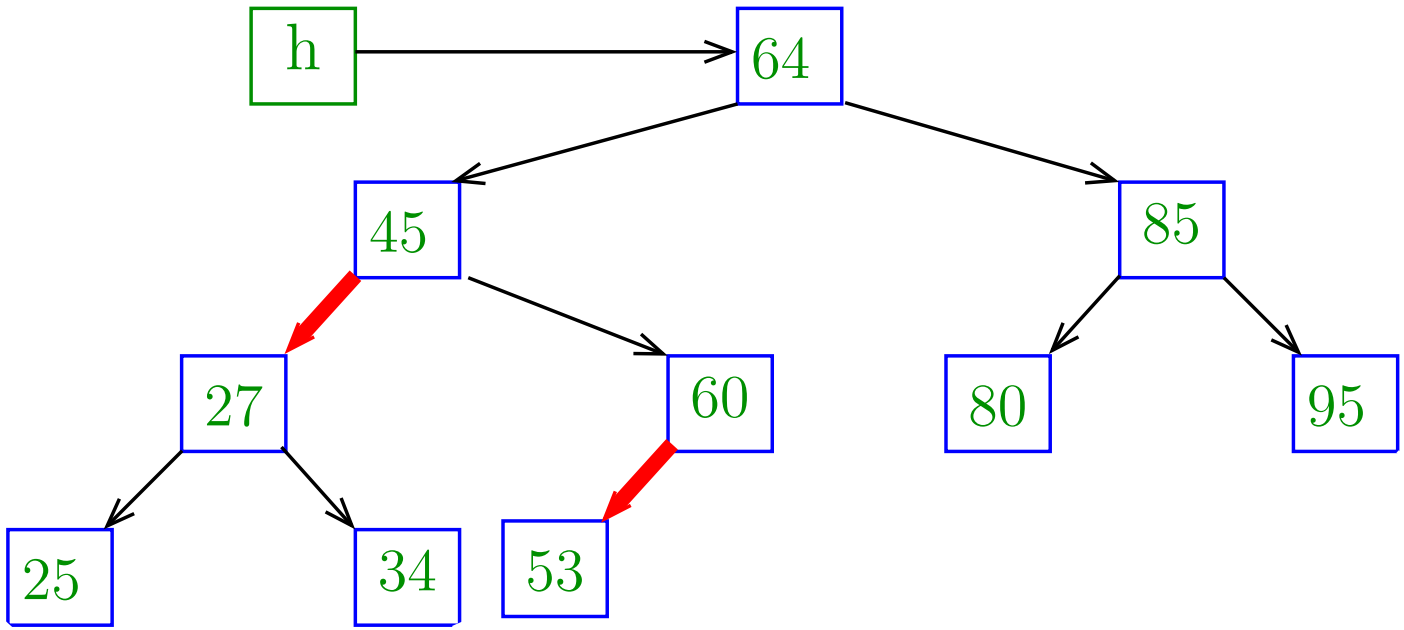

```
return balance(h);
```

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



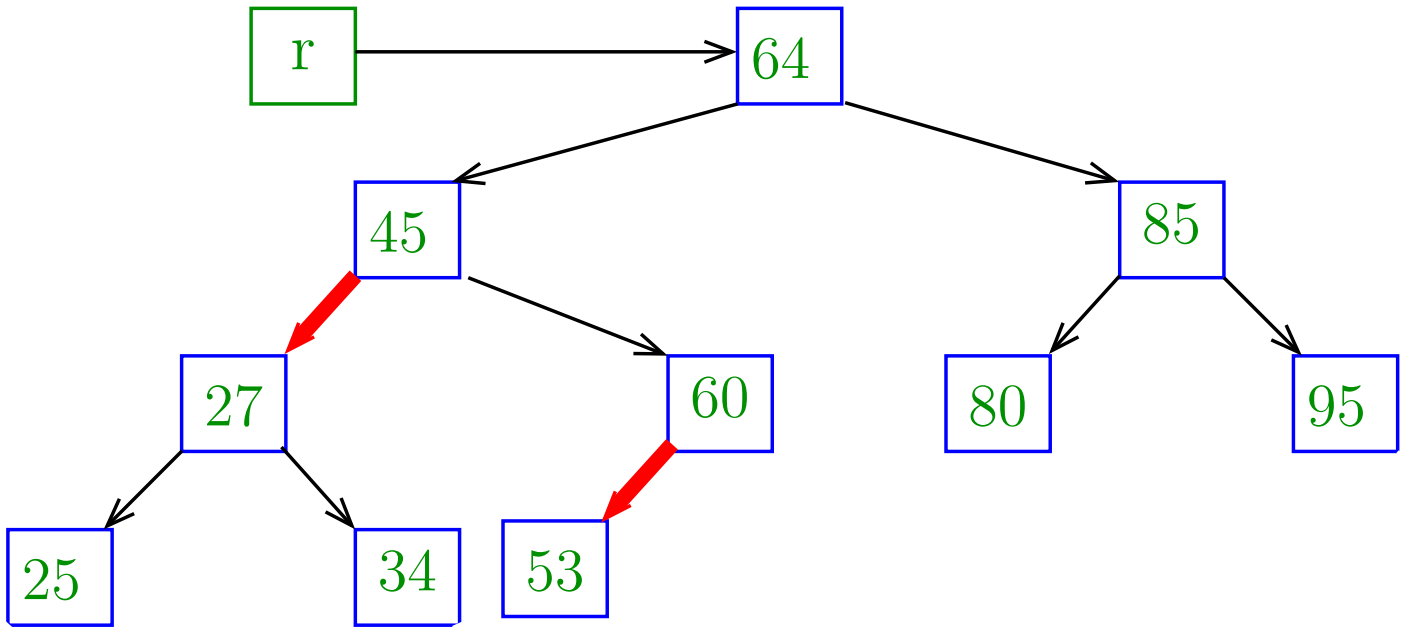

```
return balance(h);
```

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 raiz rubra.

```
void deleteMin() {
   if(r == NULL) return;
   if (!isRed(r->left) && !isRed(r->right))
      r->color = RED;
   r = deleteMinTree(r);
   if (!isEmpty()) r->color = BLACK;
}
```

deleteMin()

```
static Node deleteMinTree(Node h) {
   if (h->left == NULL)
      return NULL;
   if(!isRed(h->left) && !isRed(h->left->left))
      h = moveRedLeft(h);
   h->left = deleteMinTree(h->left);
   return balance(h);
}
```

deleteMin()

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

```
static 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**. Note a alteração.

```
static 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;
}
```