

BSTs rubro-negras: put()

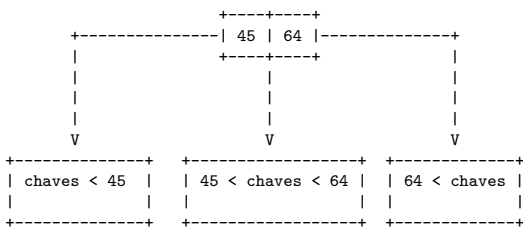


Fonte: ...-crimson-vermillion-Chinese/

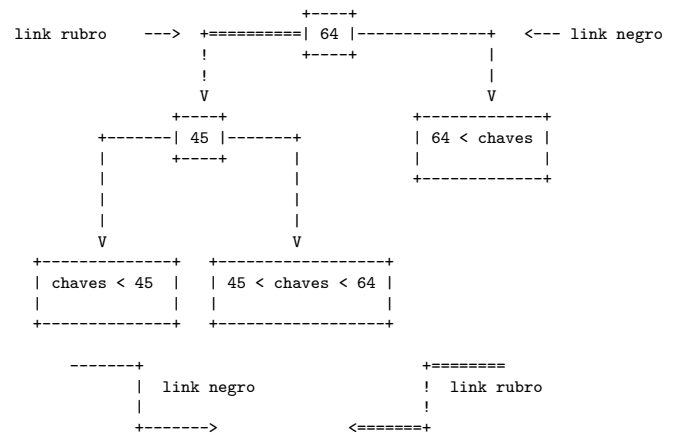
Referências: BSTs rubro-negras (PF); Balanced Search Trees (S&W); slides (S&W)

AULA 12

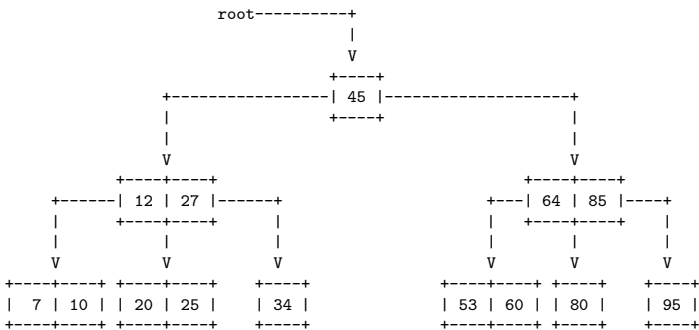
Árvore 2-3 para rubro-negra



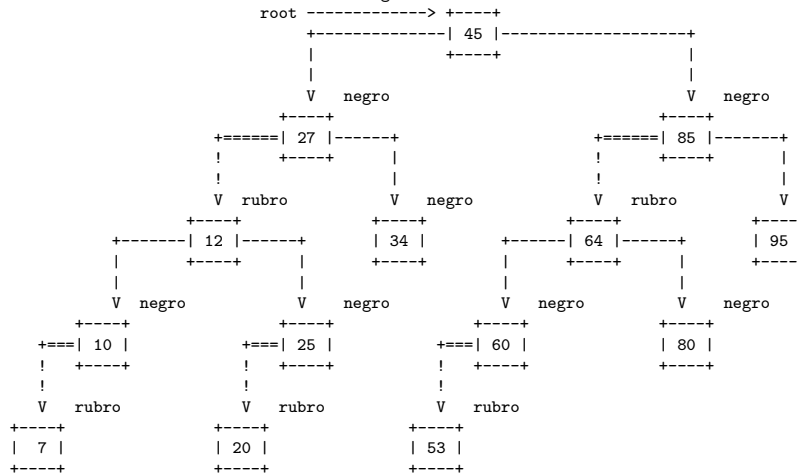
Árvore 2-3 para rubro-negra



Árvore 2-3

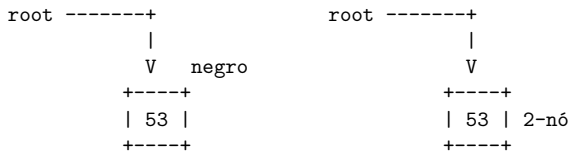


Árvore rubro-negra



Inserção em um 2-nó

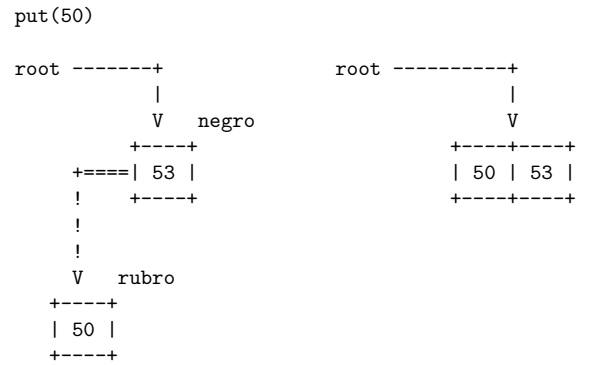
Árvore formada por apenas um 2-nó



Navigation icons

Inserção em um 2-nó

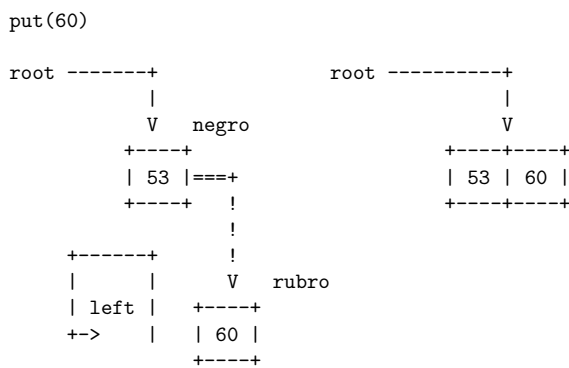
Árvore formada por apenas um 2-nó



Navigation icons

Inserção em um 2-nó

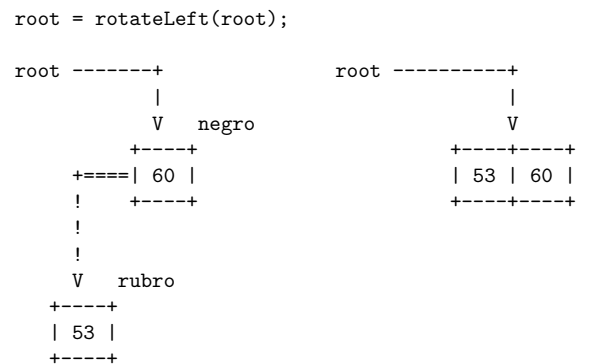
Árvore formada por apenas um 2-nó



Navigation icons

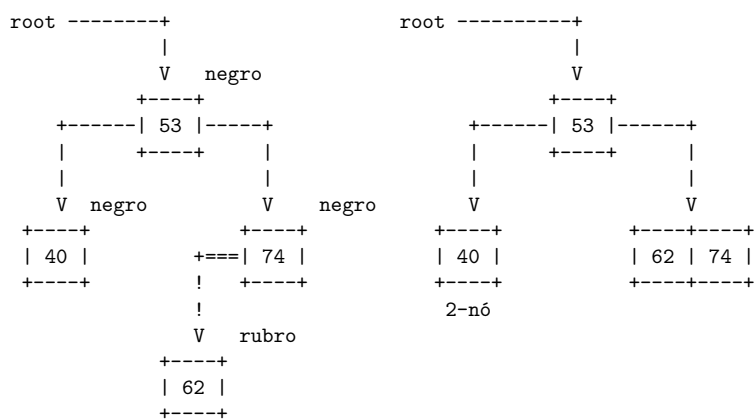
Inserção em um 2-nó

Árvore formada por apenas um 2-nó



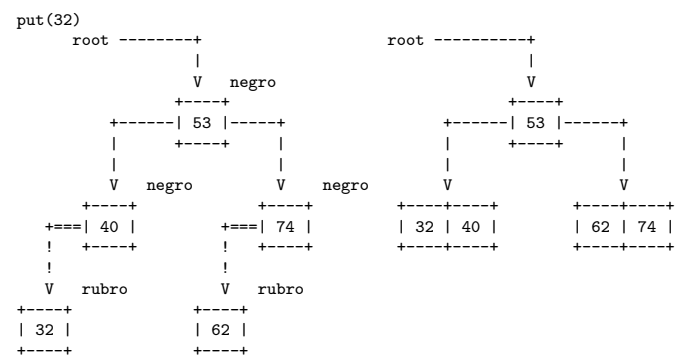
Navigation icons

Inserção em um 2-nó qualquer



Navigation icons

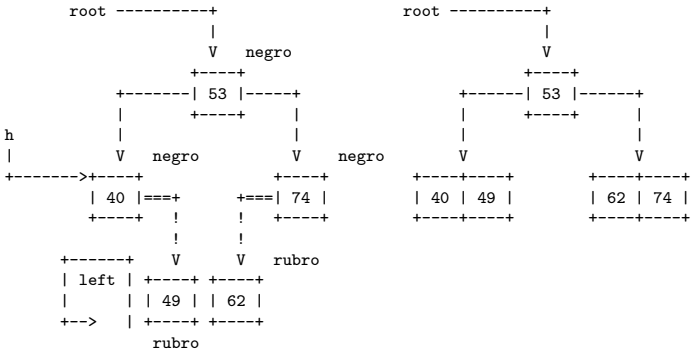
Inserção em um 2-nó qualquer



Navigation icons

Inserção em um 2-nó qualquer

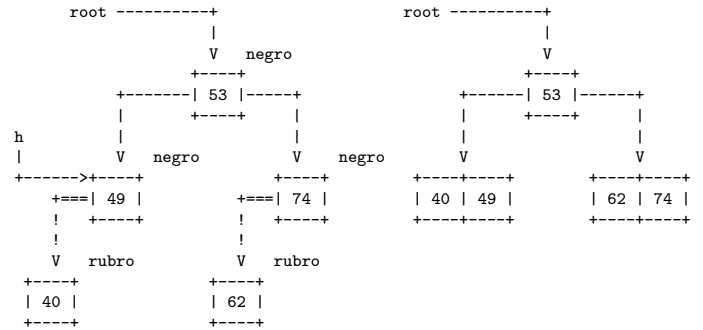
put(49)



Navigation icons

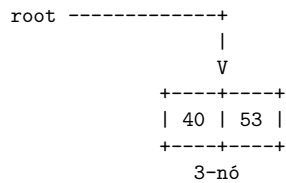
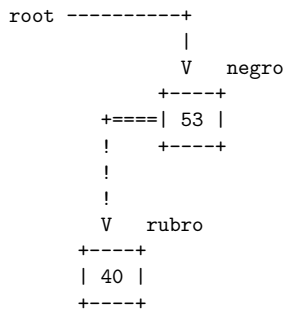
Inserção em um 2-nó qualquer

h = rotateLeft(h);



Navigation icons

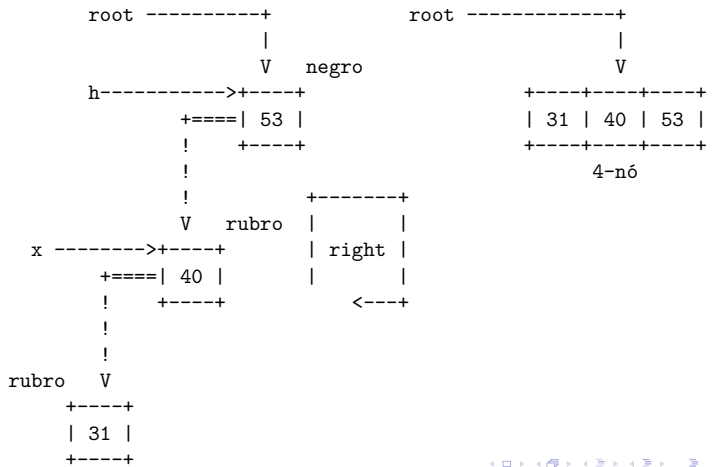
Inserção em um 3-nó



Navigation icons

chave é inserida é menor do 3-nó

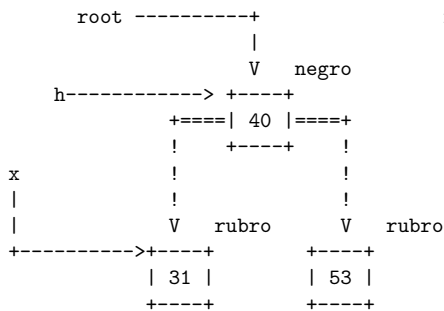
put(31)



Navigation icons

chave é inserida é menor do 3-nó

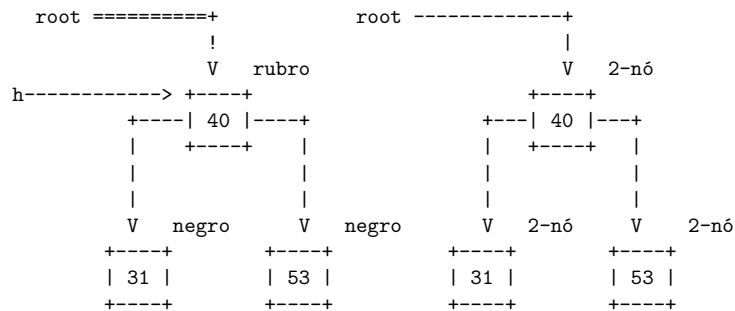
x = rotateRight(x);



Navigation icons

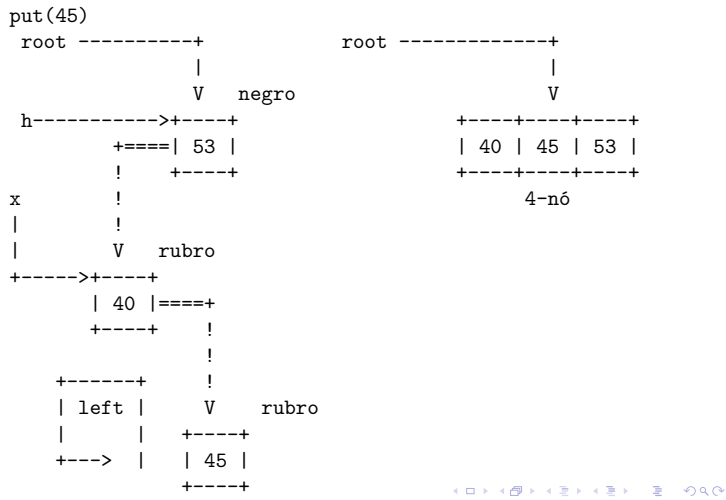
chave é inserida é menor do 3-nó

flipColors(h);

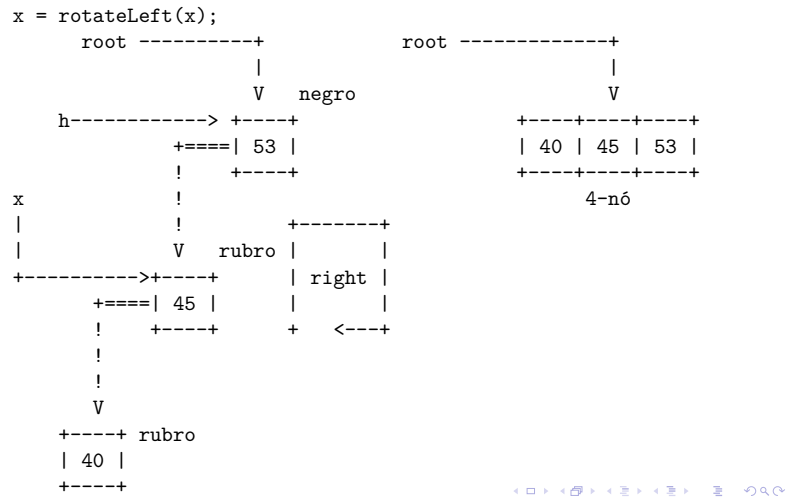


Navigation icons

chave é inserida entre as chaves do 3-nó

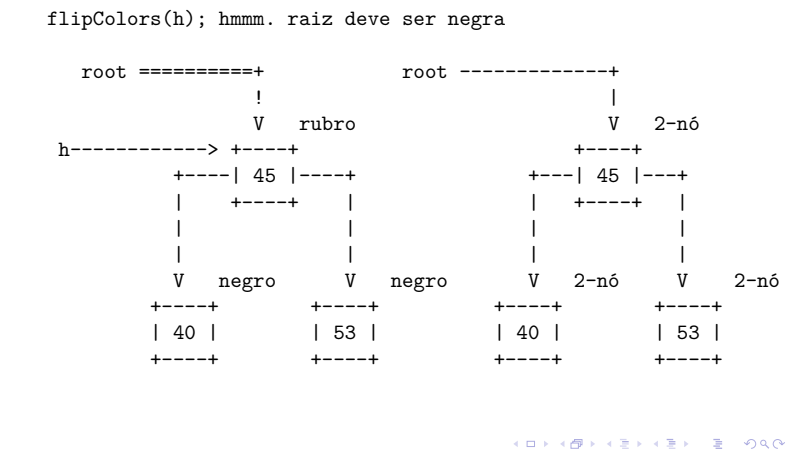
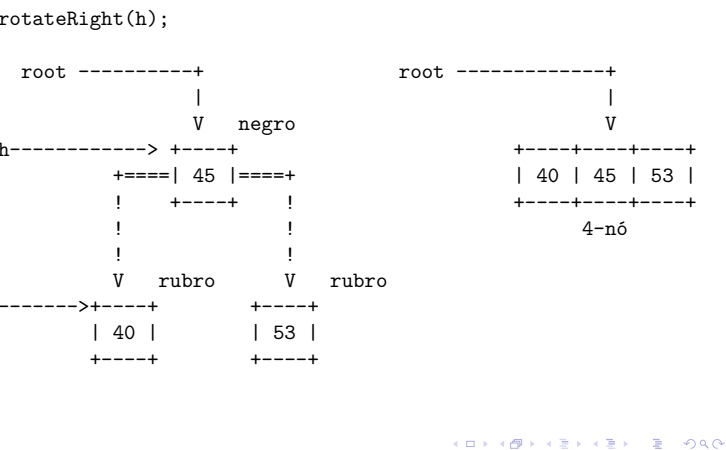


chave é inserida entre as chaves do 3-nó



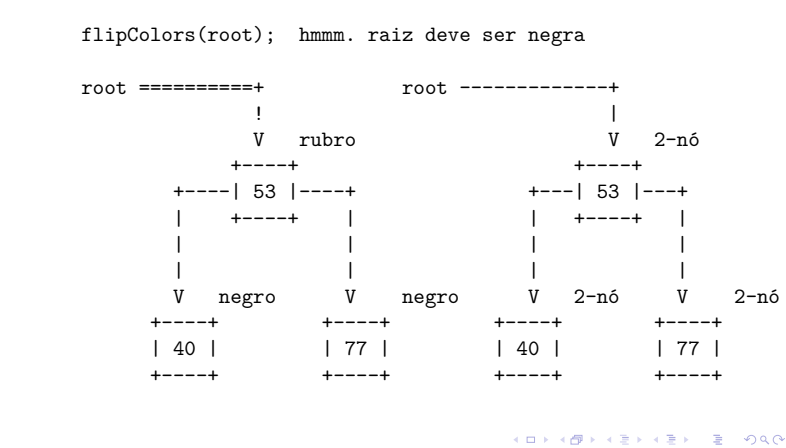
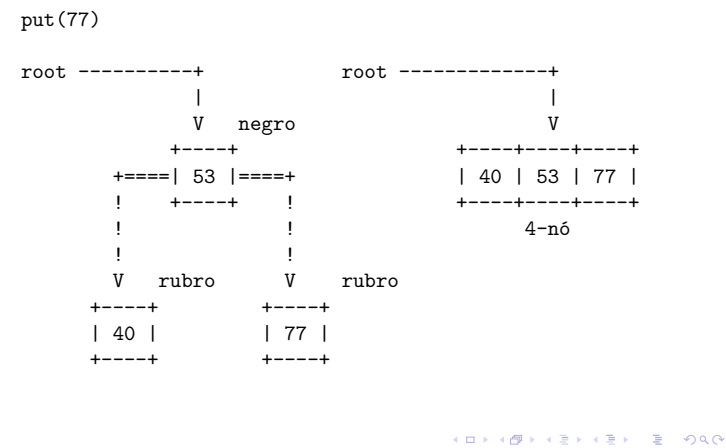
chave é inserida entre as chaves do 3-nó

chave é inserida entre as chaves do 3-nó

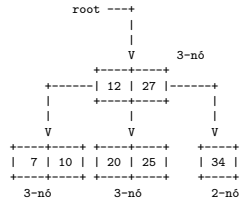
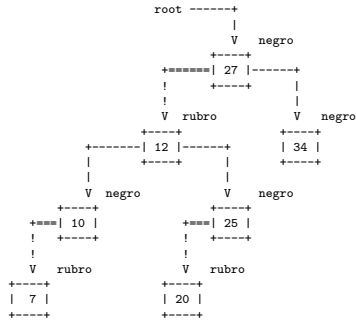


chave inserida é maior que todas do 3-nó

chave inserida é maior que todas do 3-nó

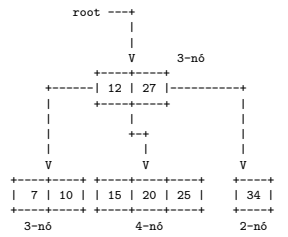
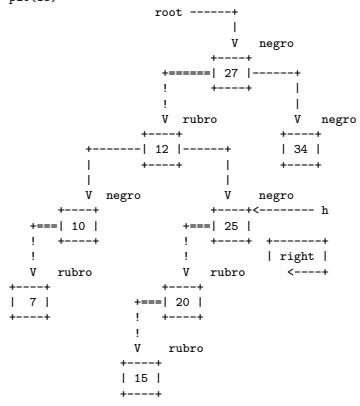


chave é inserida em um 3-nó qualquer



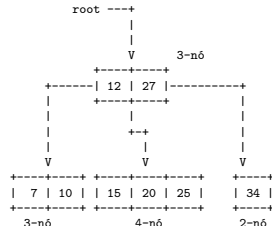
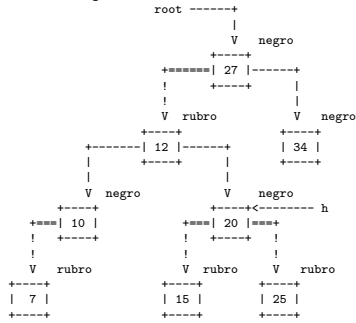
chave é inserida em um 3-nó qualquer

put(15)



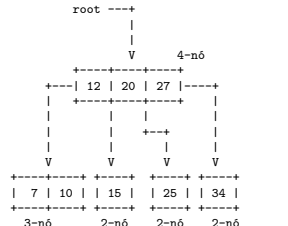
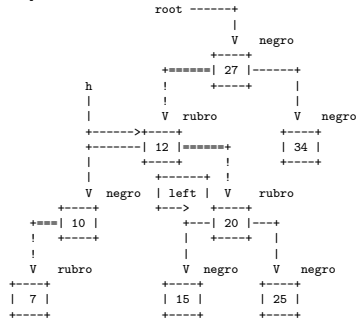
chave é inserida em um 3-nó qualquer

h = rotateRight(h);



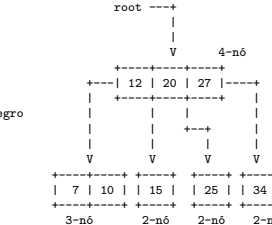
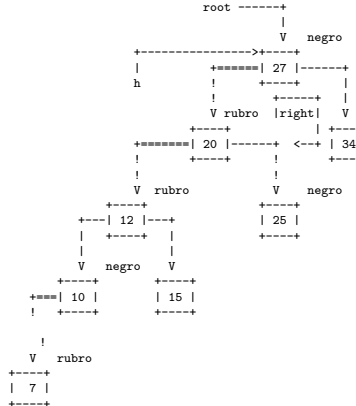
chave é inserida em um 3-nó qualquer

flipColors(h);



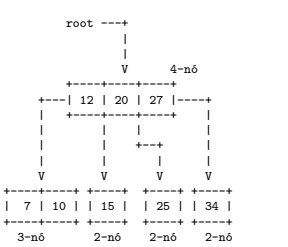
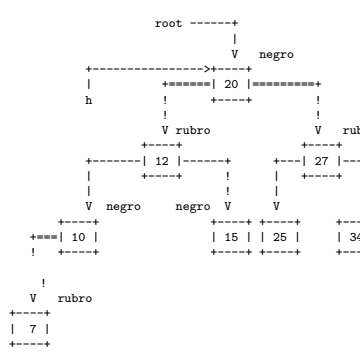
chave é inserida em um 3-nó qualquer

h = rotateLeft(h);



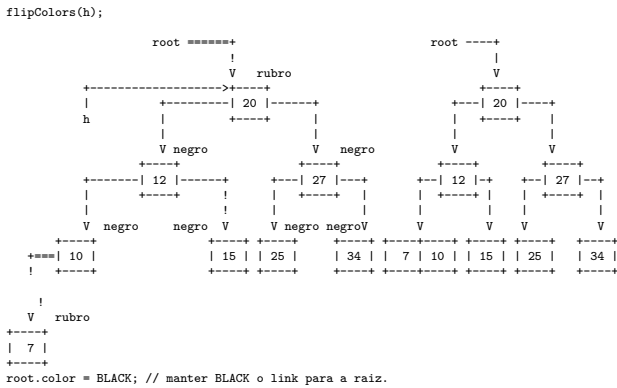
chave é inserida em um 3-nó qualquer

h = rotateRight(h);



chave é inserida em um 3-nó qualquer

rotações



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

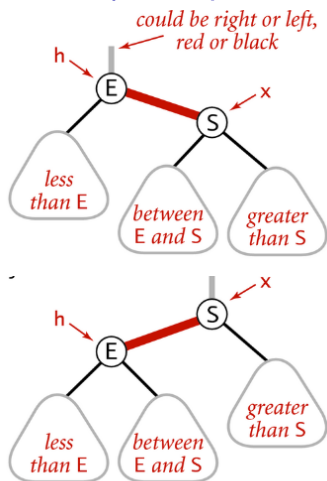
Durante uma operação de inserção, podemos ter, temporariamente, um **link rubro inclinado para a direita** ou **dois links rubros incidindo no mesmo nó**.

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.

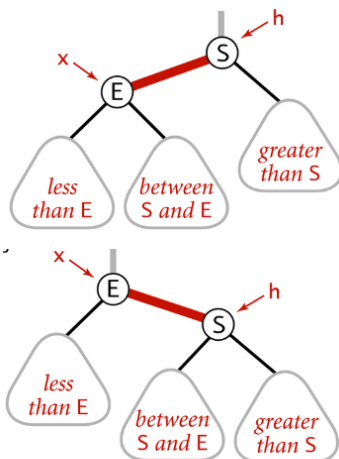
Rotação esquerda



Fonte: [algs4](#)

Left rotate (right link of h)

Rotação direita



Fonte: [algs4](#)

Right rotate (left link of h)

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

```
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

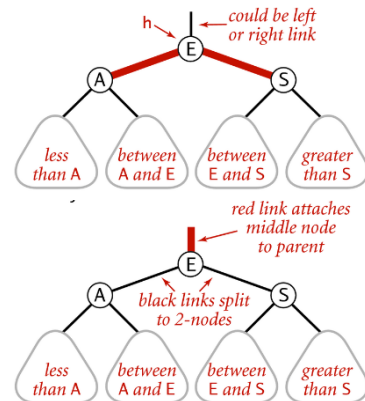
As operações de **rotação** são **locais**.

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;
}
```



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.



BSTs rubro-negras: delete()

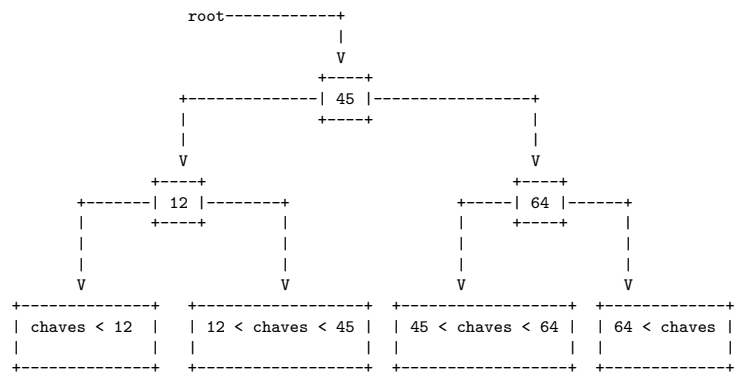


Fonte: [.../only-one/red-leaves-black-tree/](https://www.pexels.com/photo/autumn-forest-1000000000/)

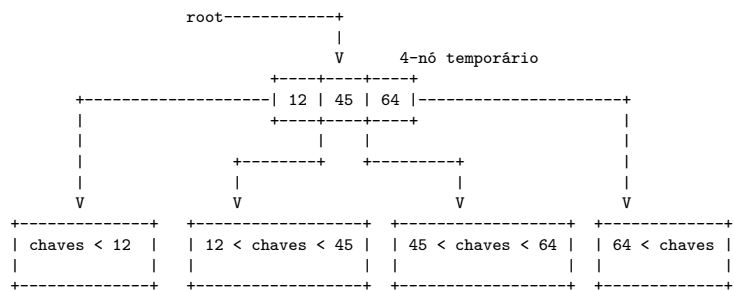
Referências: BSTs rubro-negras (PF); Balanced Search Trees (S&W); slides (S&W)



Os dois filhos da raiz são 2-nós



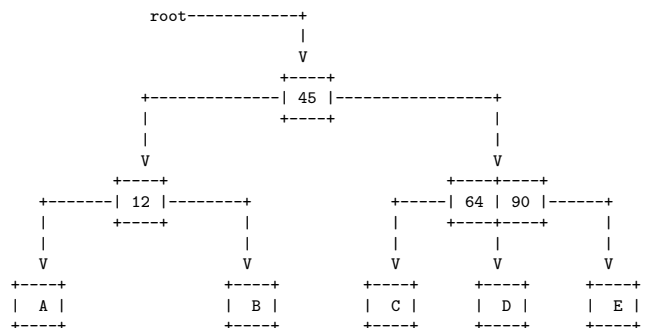
Os dois filhos da raiz são 2-nós



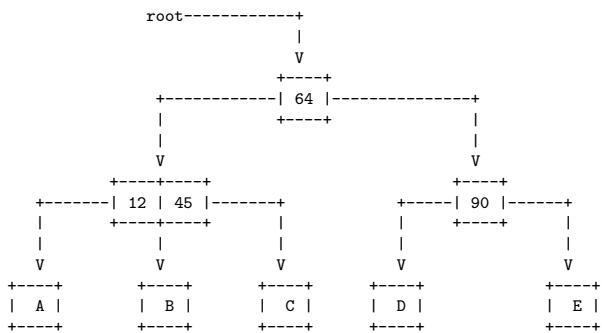
Agora passemos a raiz quando apenas o nó esquerdo é um 2-nó.



Filhos esquerdo da raiz é 2-nós



Filhos esquerdo da raiz é 2-nós



Mover para esquerda

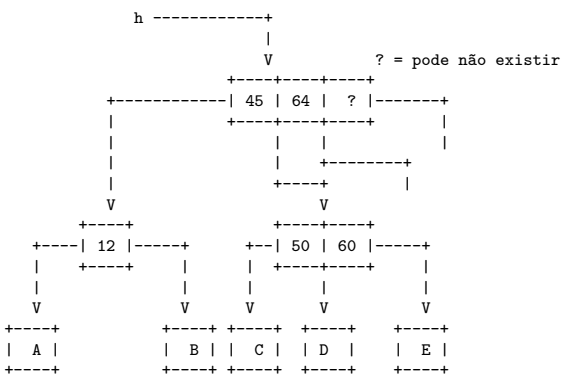
No meio do caminho, sabemos que o nó corrente **h** é um 3-nó ou um 4-nó.

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

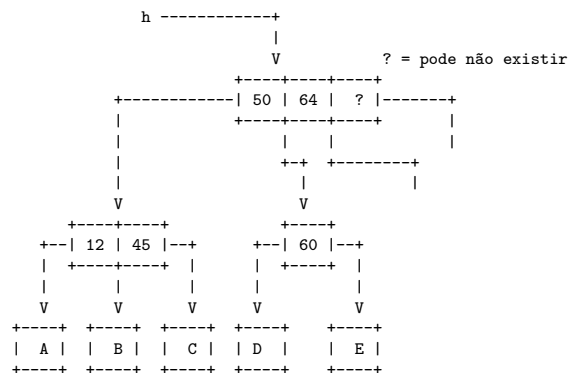
Se ele já é um 3-nó, não precisamos fazer nada.

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

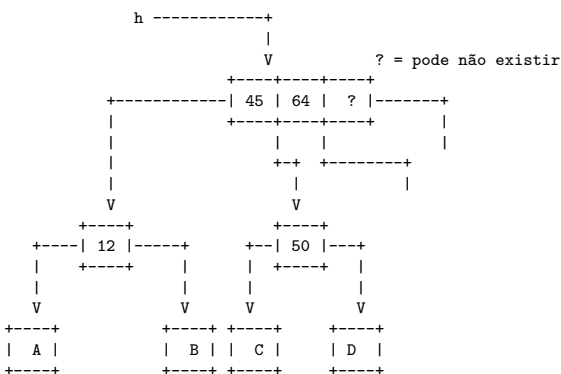
Filho mais a esquerda de h é 2-nó



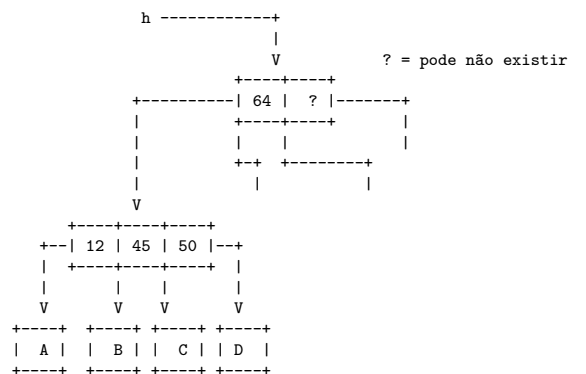
Filho mais a esquerda de h é 2-nó



Dois filhos mais à esquerda de h são 2-nós



Dois filhos mais à esquerda de h são 2-nós



Finalmente...

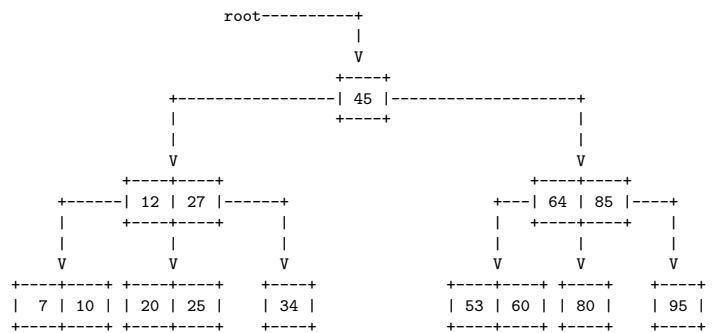
Desta forma quando atingirmos a folha mais a esquerda dessa **árvore 2-3-4** teremos chegado a um **3-nó** ou **4-no**.

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

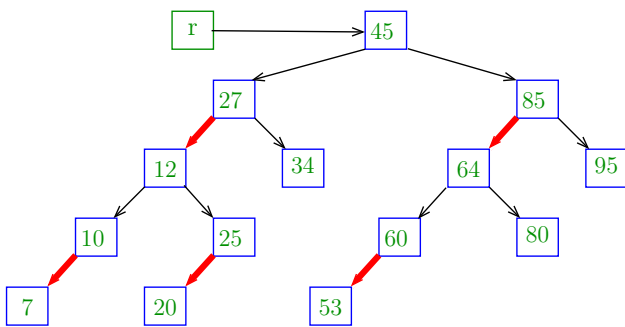
Depois devemos voltar "espatifando" os **4-nós** que por ventura deixamos pelo caminho.

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()

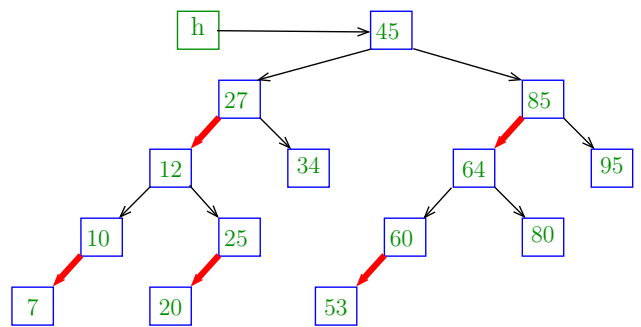


rubro-negra: deleteMin()



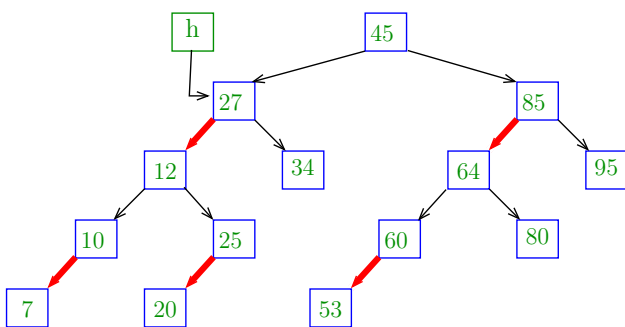
`deleteMin(r);`

rubro-negra: deleteMin()



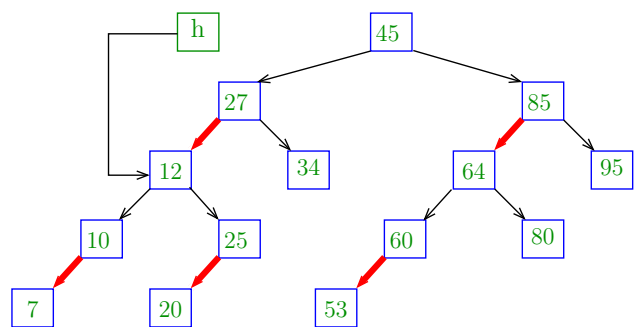
`h.left = deleteMin(h.left);`

rubro-negra: deleteMin()



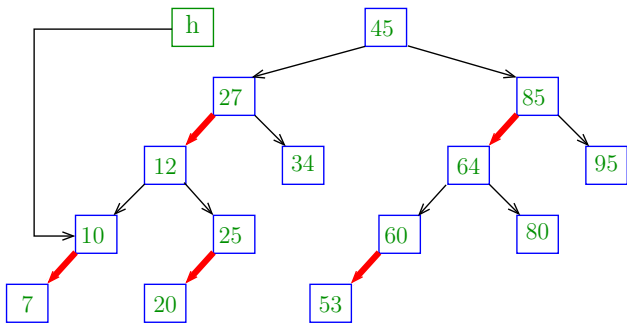
`h.left = deleteMin(h.left);`

rubro-negra: deleteMin()



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

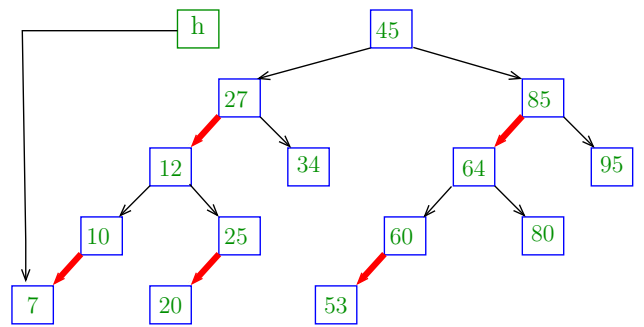
rubro-negra: deleteMin()



h.left = deleteMin(h.left);



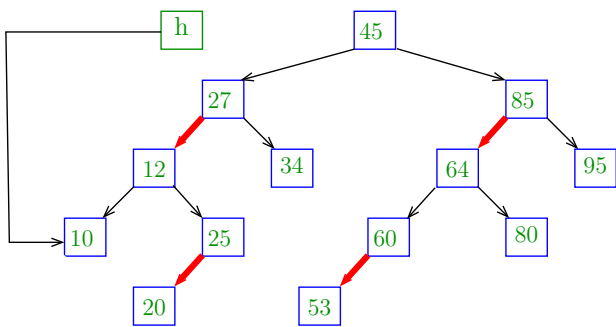
rubro-negra: deleteMin()



return null;



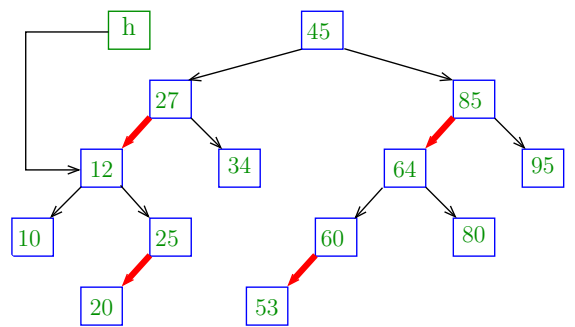
rubro-negra: deleteMin()



return balance(h);



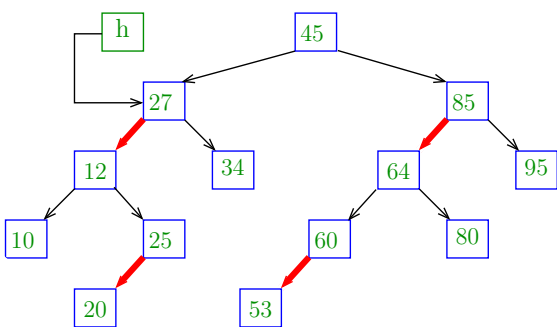
rubro-negra: deleteMin()



return balance(h);



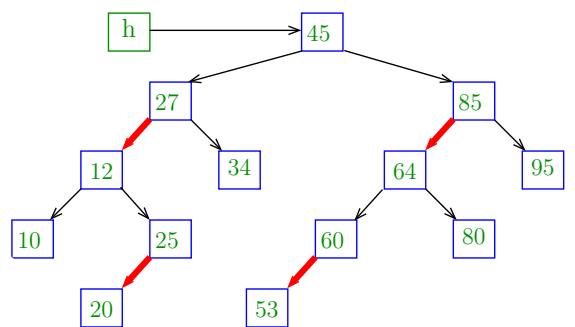
rubro-negra: deleteMin()



return balance(h);



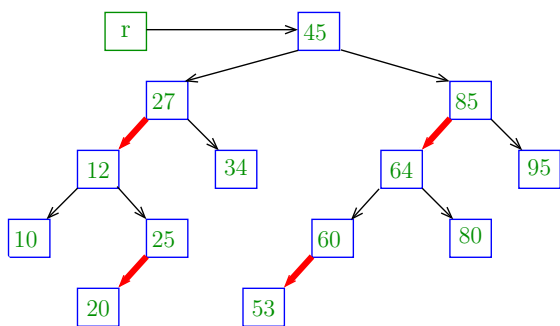
rubro-negra: deleteMin()



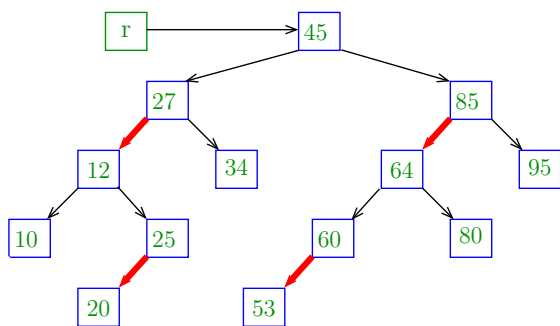
return balance(h);



rubro-negra: deleteMin()

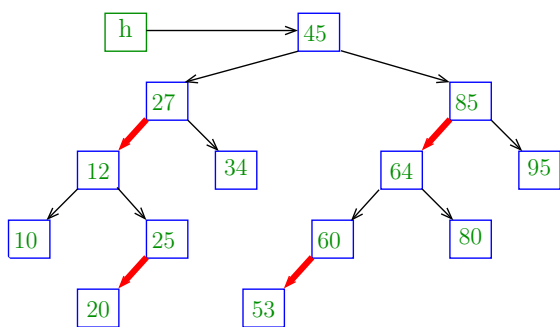


rubro-negra: outro deleteMin()



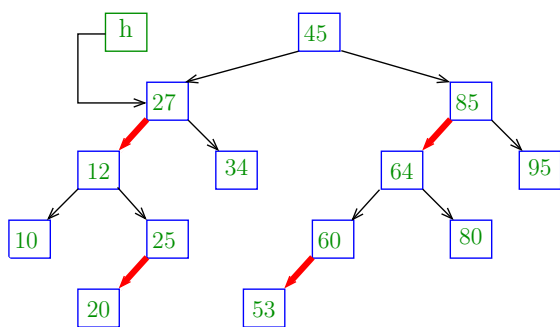
deleteMin(r);

rubro-negra: outro deleteMin()



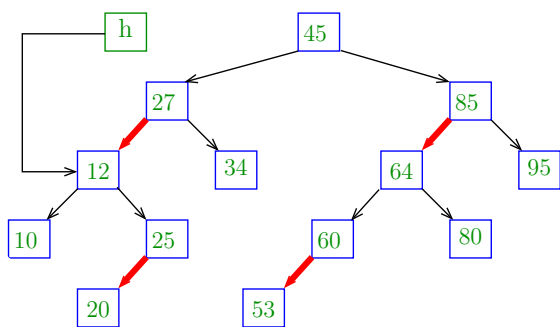
h.left = deleteMin(h.left);

rubro-negra: outro deleteMin()



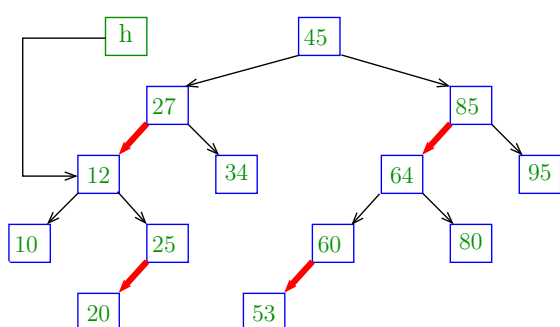
h.left = deleteMin(h.left);

rubro-negra: outro deleteMin()



h = moveRedLeft(h);

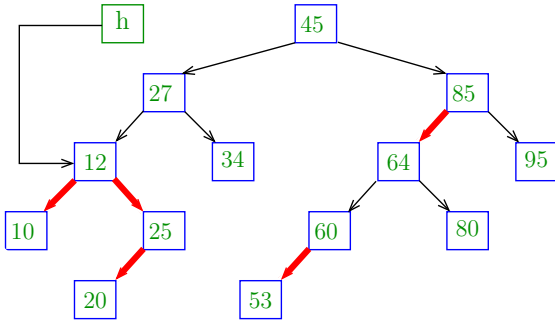
rubro-negra: outro deleteMin()



flipColors(h);

[moveRedLeft(h);]

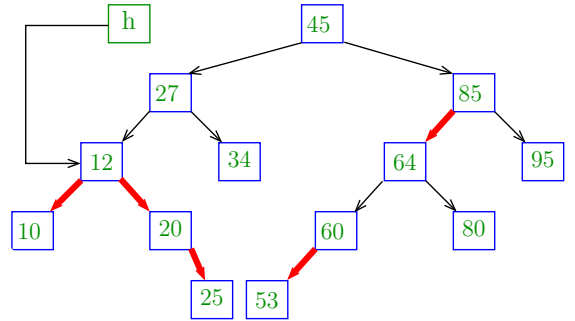
rubro-negra: outro deleteMin()



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

Navigation icons

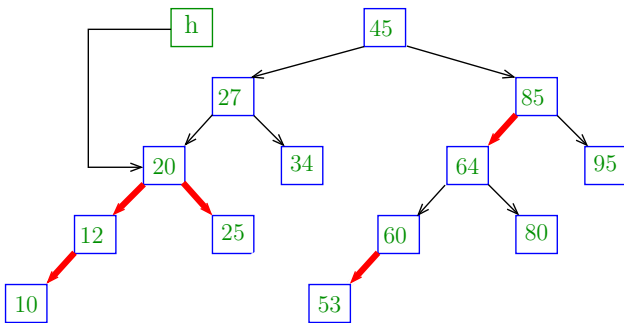
rubro-negra: outro deleteMin()



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

Navigation icons

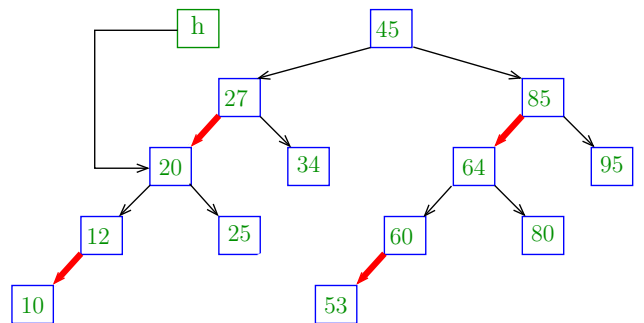
rubro-negra: outro deleteMin()



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

Navigation icons

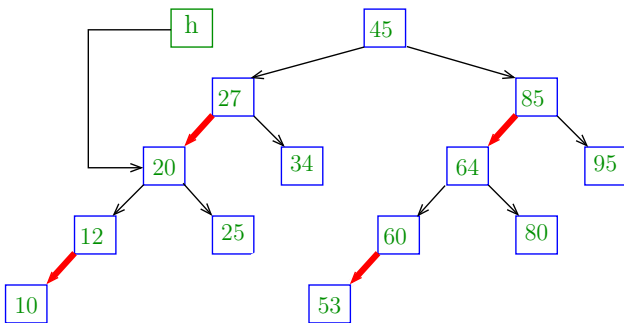
rubro-negra: outro deleteMin()



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

Navigation icons

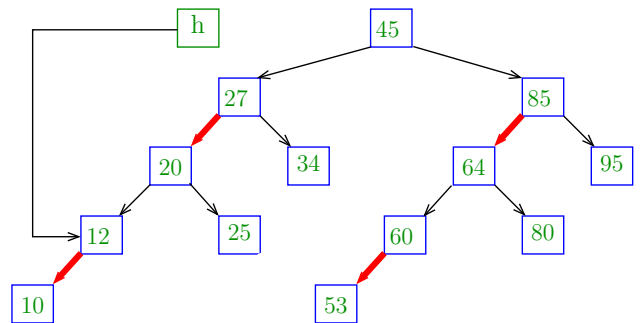
rubro-negra: outro deleteMin()



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

Navigation icons

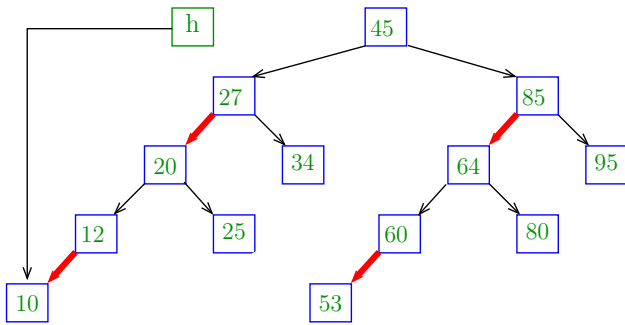
rubro-negra: outro deleteMin()



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

Navigation icons

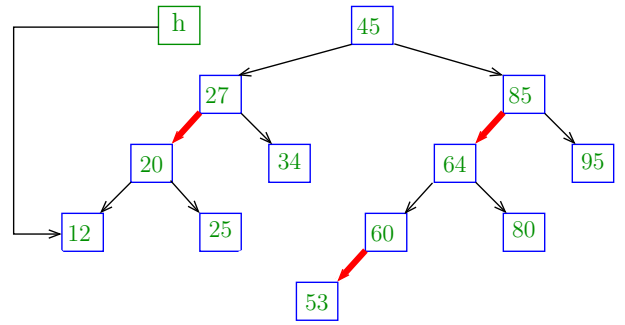
rubro-negra: outro deleteMin()



return null;



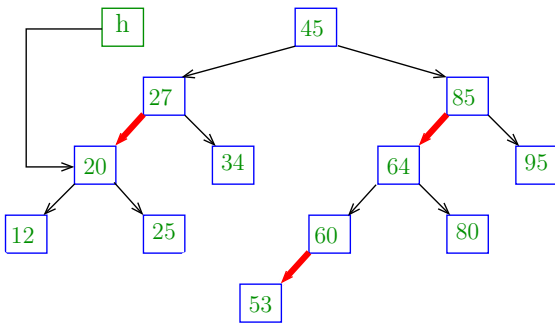
rubro-negra: outro deleteMin()



return balance(h);



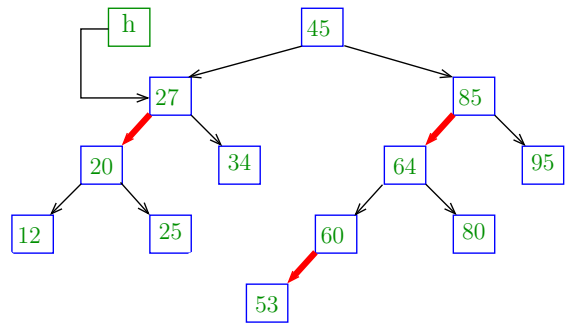
rubro-negra: outro deleteMin()



return balance(h);



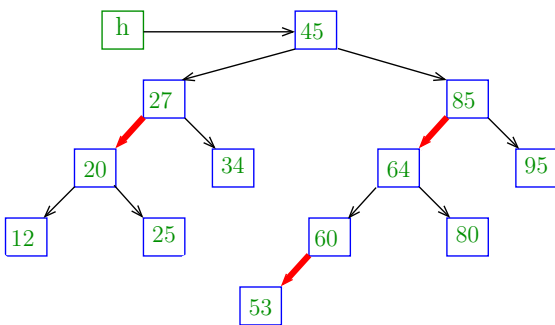
rubro-negra: outro deleteMin()



return balance(h);



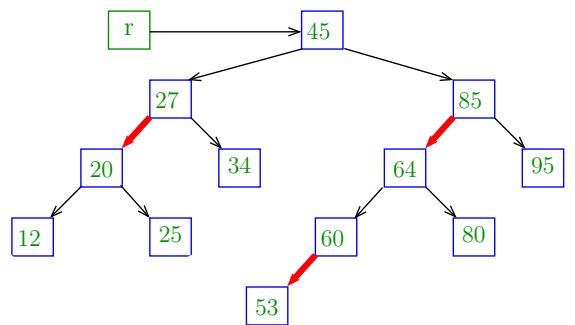
rubro-negra: outro deleteMin()



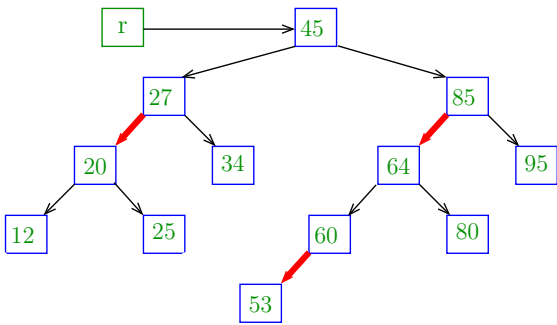
return balance(h);



rubro-negra: outro deleteMin()



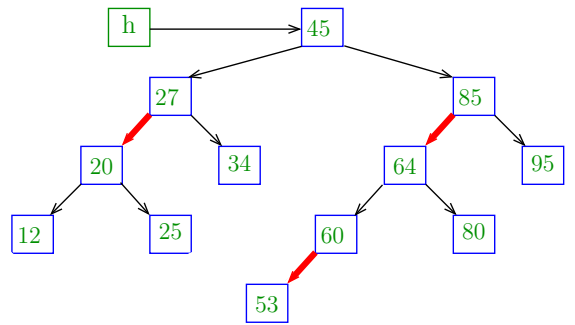
rubro-negra: mais outro deleteMin()



`deleteMin(r);`

Navigation icons: back, forward, search, etc.

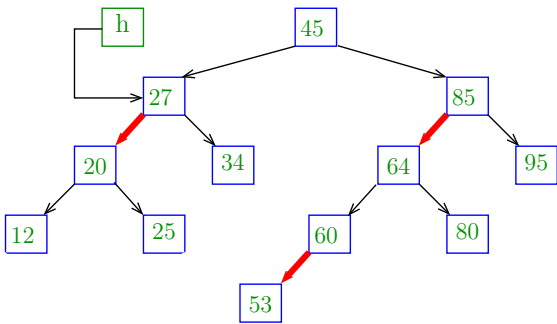
rubro-negra: mais outro deleteMin()



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

Navigation icons: back, forward, search, etc.

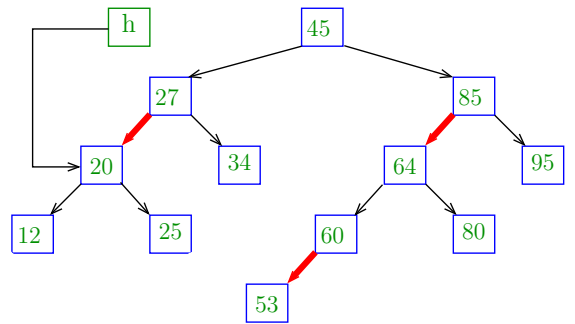
rubro-negra: mais outro deleteMin()



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

Navigation icons: back, forward, search, etc.

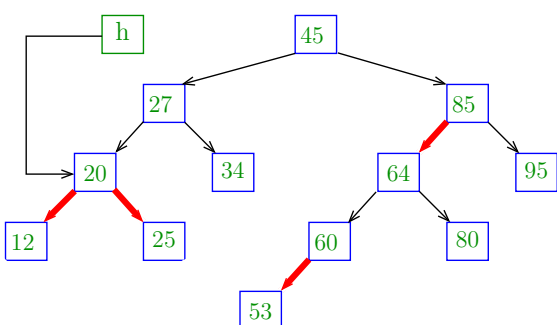
rubro-negra: mais outro deleteMin()



`h = moveRedLeft(h);`

Navigation icons: back, forward, search, etc.

rubro-negra: mais outro deleteMin()

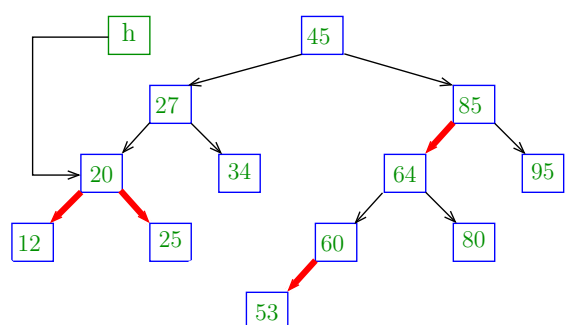


`flipColors(h);`

`[moveRedLeft(h);]`

Navigation icons: back, forward, search, etc.

rubro-negra: mais outro deleteMin()

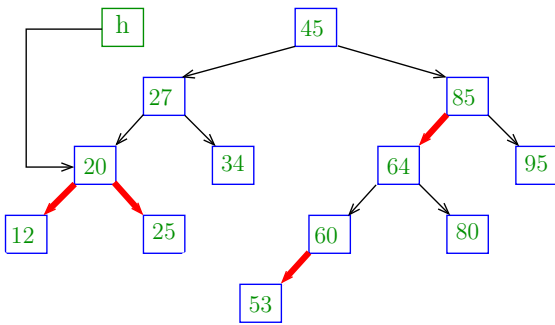


`return h`

`[moveRedLeft(h);]`

Navigation icons: back, forward, search, etc.

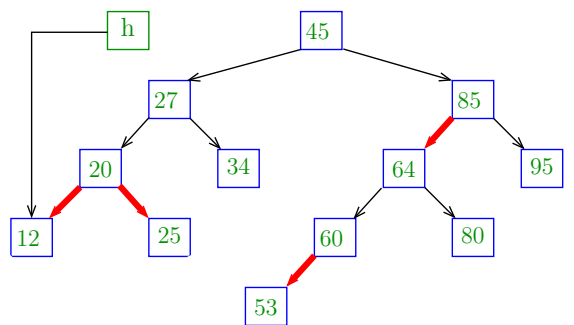
rubro-negra: mais outro deleteMin()



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

Navigation icons

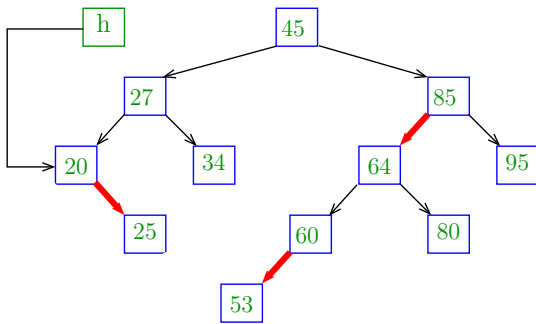
rubro-negra: mais outro deleteMin()



```
return null
```

Navigation icons

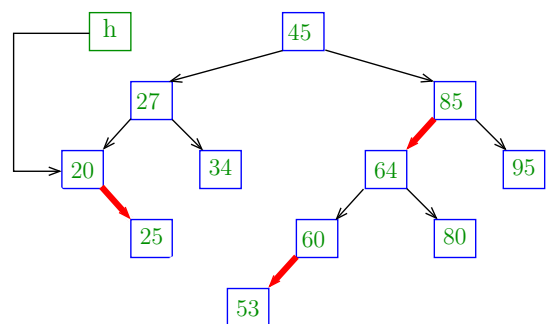
rubro-negra: mais outro deleteMin()



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

Navigation icons

rubro-negra: mais outro deleteMin()

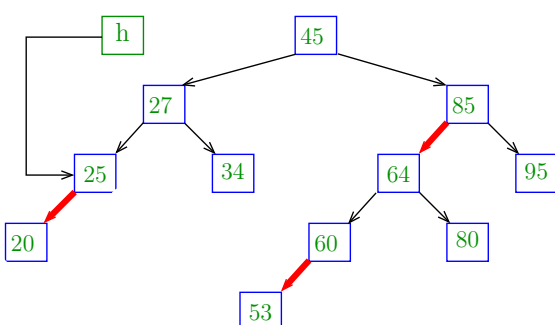


```
h = rotateLeft(h);
```

[balance(h);]

Navigation icons

rubro-negra: mais outro deleteMin()

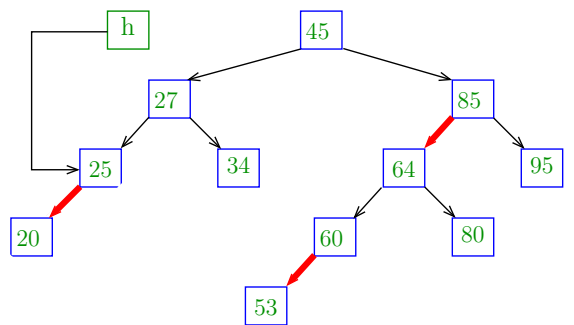


```
return h;
```

[balance(h);]

Navigation icons

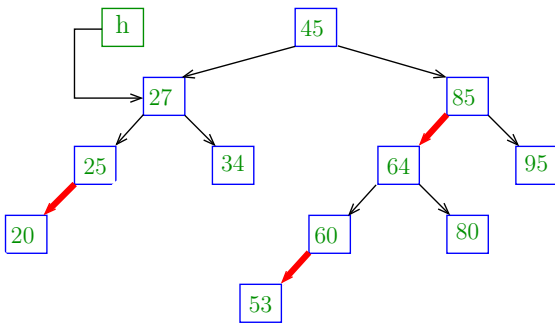
rubro-negra: mais outro deleteMin()



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

Navigation icons

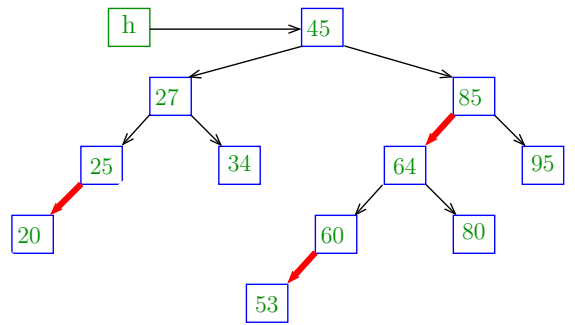
rubro-negra: mais outro deleteMin()



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

Navigation icons

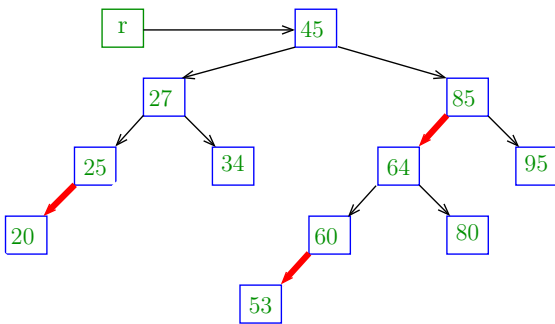
rubro-negra: mais outro deleteMin()



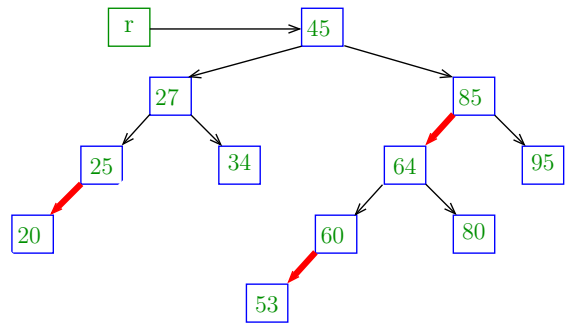
```
return balance(h);
```

Navigation icons

rubro-negra: mais outro deleteMin()



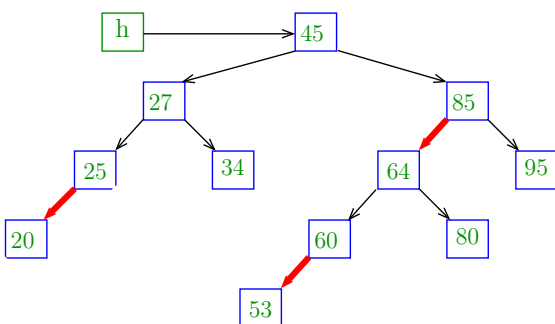
rubro-negra: mais outro deleteMin() ainda



```
deleteMin(r);
```

Navigation icons

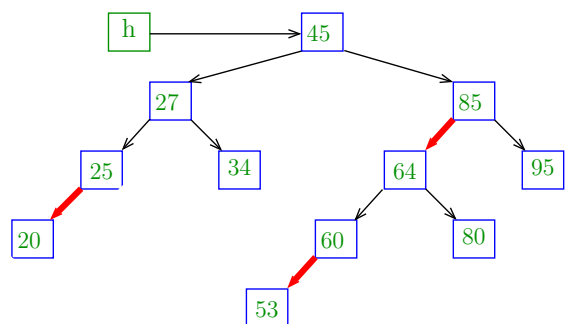
rubro-negra: mais outro deleteMin() ainda



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

Navigation icons

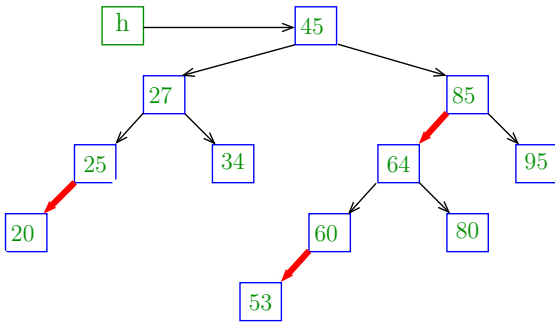
rubro-negra: mais outro deleteMin() ainda



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

Navigation icons

rubro-negra: mais outro deleteMin() ainda

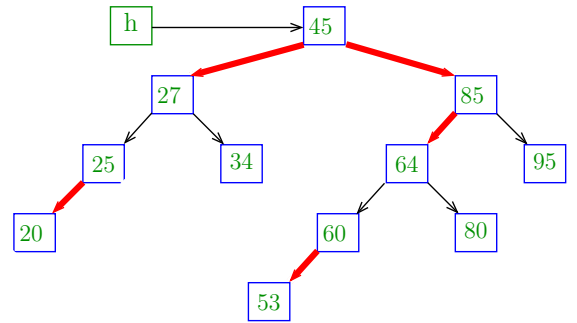


flipColors(h)

[moveRedLeft(h);



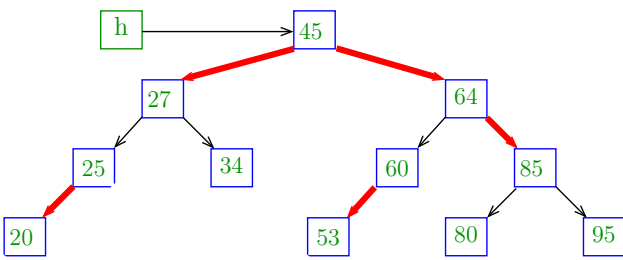
rubro-negra: mais outro deleteMin() ainda



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



rubro-negra: mais outro deleteMin() ainda

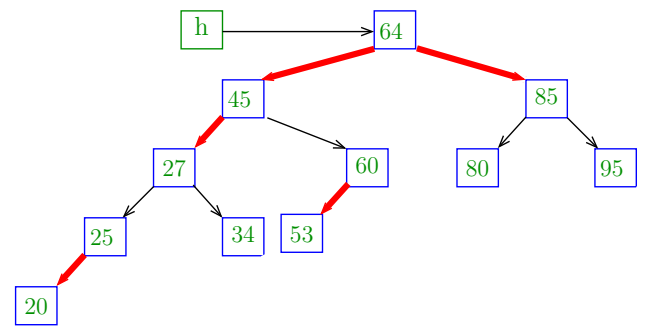


h = rotateLeft(h);

[moveRedLeft(h);



rubro-negra: mais outro deleteMin() ainda

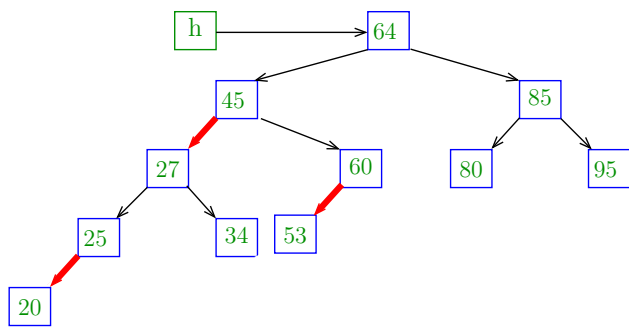


flipColors(h)

[moveRedLeft(h);



rubro-negra: mais outro deleteMin() ainda

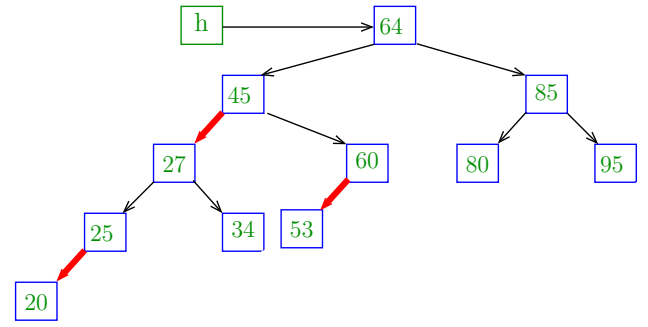


return h;

[moveRedLeft(h);



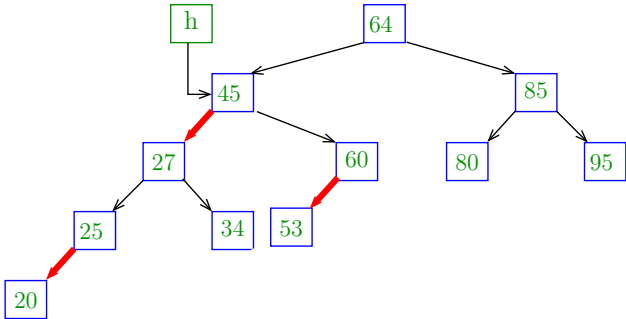
rubro-negra: mais outro deleteMin() ainda



h.left = deleteMin(h.left);



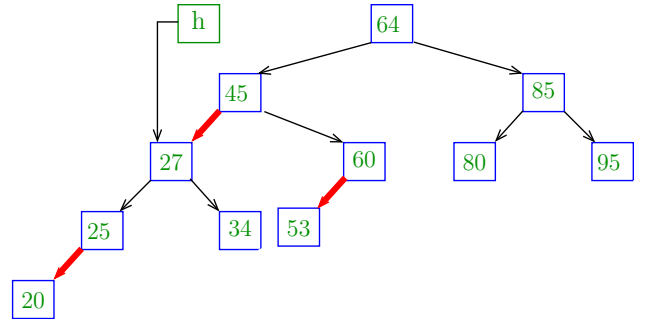
rubro-negra: mais outro deleteMin() ainda



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



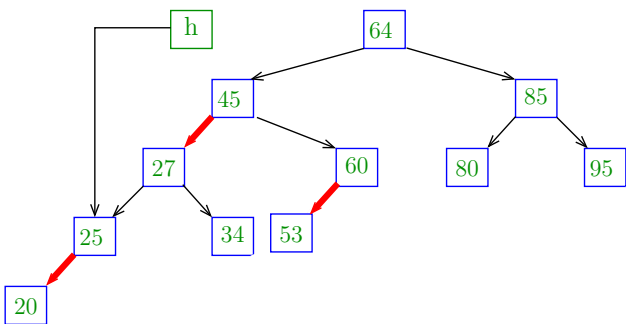
rubro-negra: mais outro deleteMin() ainda



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



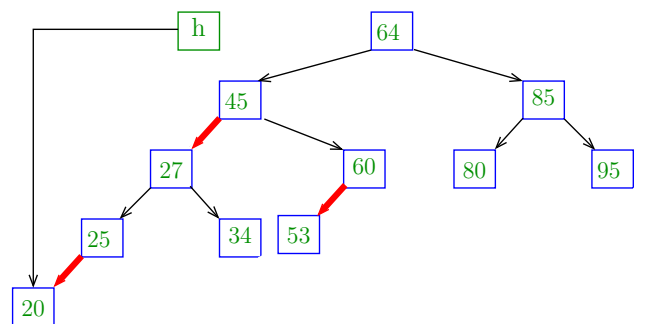
rubro-negra: mais outro deleteMin() ainda



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



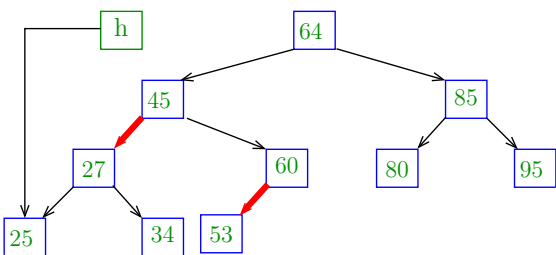
rubro-negra: mais outro deleteMin() ainda



```
return null;
```



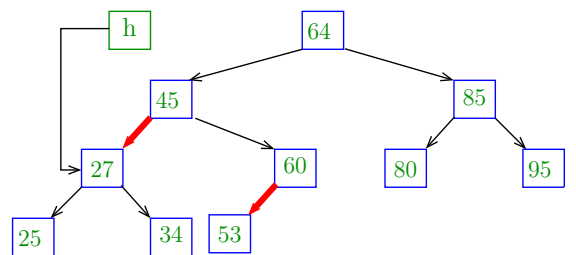
rubro-negra: mais outro deleteMin() ainda



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



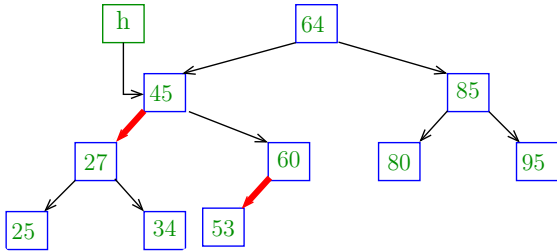
rubro-negra: mais outro deleteMin() ainda



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



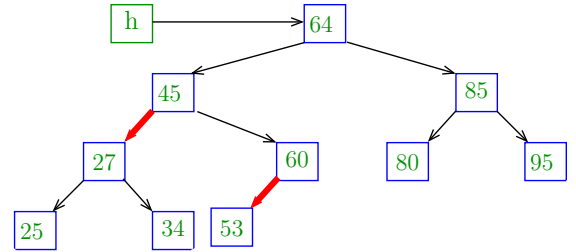
rubro-negra: mais outro deleteMin() ainda



return balance(h);



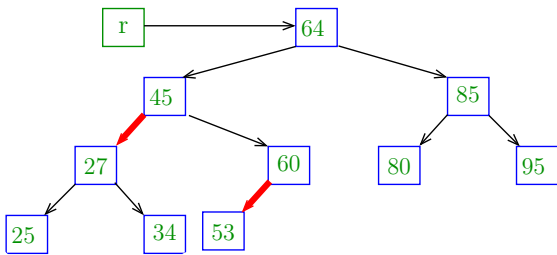
rubro-negra: mais outro deleteMin() ainda



return balance(h);



rubro-negra: mais outro deleteMin() ainda



deleteMin()

```
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()

Se ambos os filhos da raiz são **negros** faz a raiz **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()

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;
}
```

Navigation icons

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;
}
```

Navigation icons

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;
    }
}
```

Navigation icons

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);
}
```

Navigation icons

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;
}
```

Navigation icons

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!

Navigation icons

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;
}
```

