



Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 11

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.

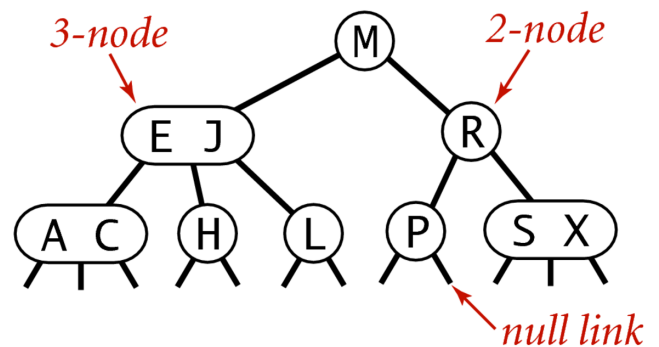
Árvores 2-3 são **perfeitamente balanceada**: todos links **null** estão no **mesmo nível**.

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**.

Anatomia de uma árvore 2-3 de busca



Anatomy of a 2-3 search tree

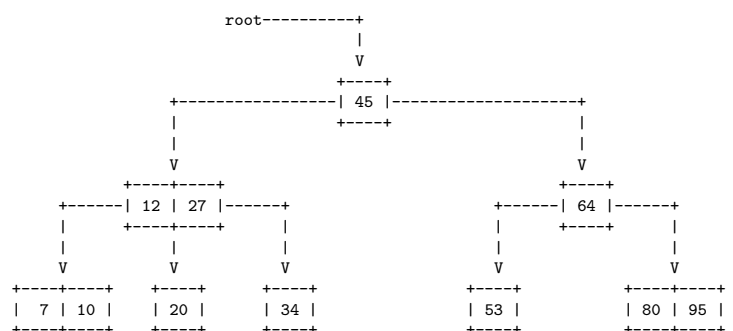
Fonte: algs4

Estrutura

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

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

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



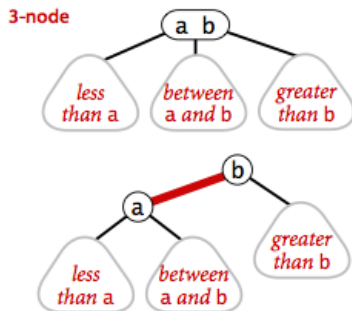
Fonte: <http://scottlobdell.me/>

Uma **BST rubro-negra** (*red-black BST*) é uma **BST** que **simula** uma **árvore 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.



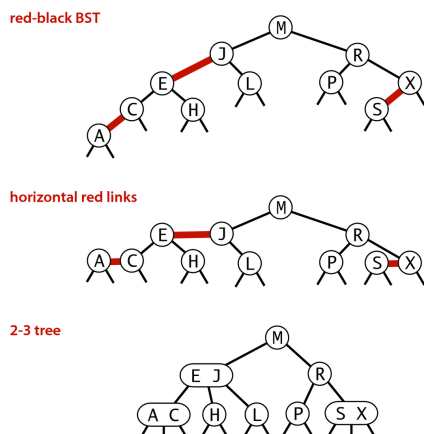
BSTs rubro-negras

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

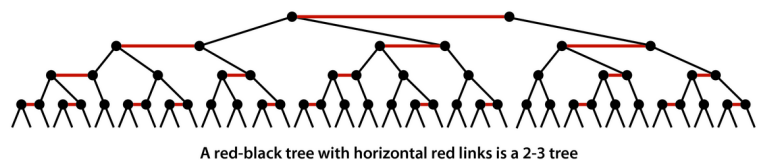


Fonte: algs4

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**:



Fonte: algs4

Nós de uma BST rubro-negra

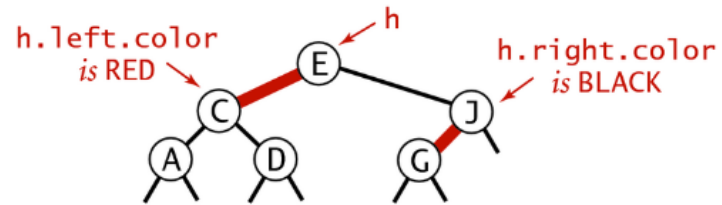
É inconveniente armazenar a **cor** de um link na estrutura de dados; é mais simples armazenar essa informação nos nós.

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



Navigation icons

Navigation icons

Nós de uma BST rubro-negra

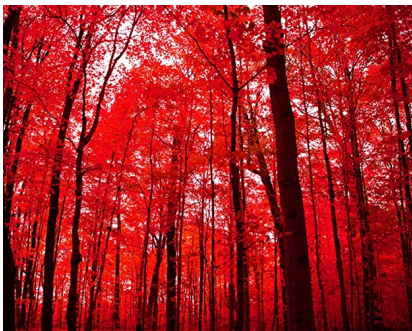
```
private class Node{
    Key key; Value val;
    Node left, right;
    int n; // número de nós nesta subárvore
    boolean color; // cor do link para este nó
    Node(Key key, Value val, int n,
         boolean color) {
        this.key = key; this.val = val;
        this.n = n; this.color = color;
    }
}
private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
}
```

Navigation icons

Navigation icons

AULA 12

BSTs rubro-negras: put()

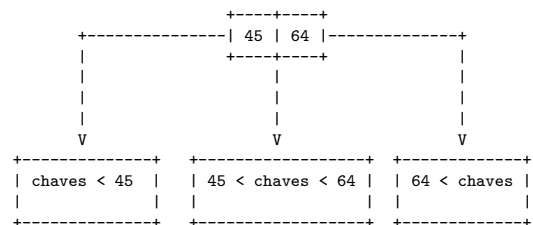


Fonte: ...-crimson-vermillion-Chinese/

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

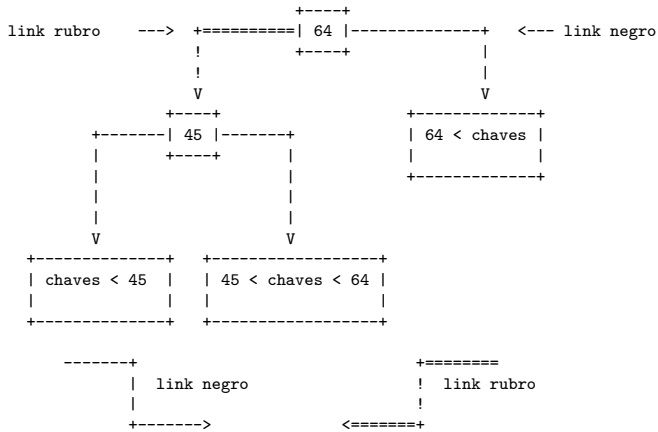
Navigation icons

Árvore 2-3 para rubro-negra

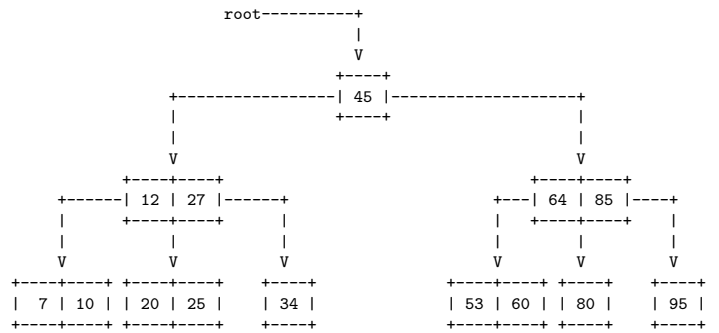


Navigation icons

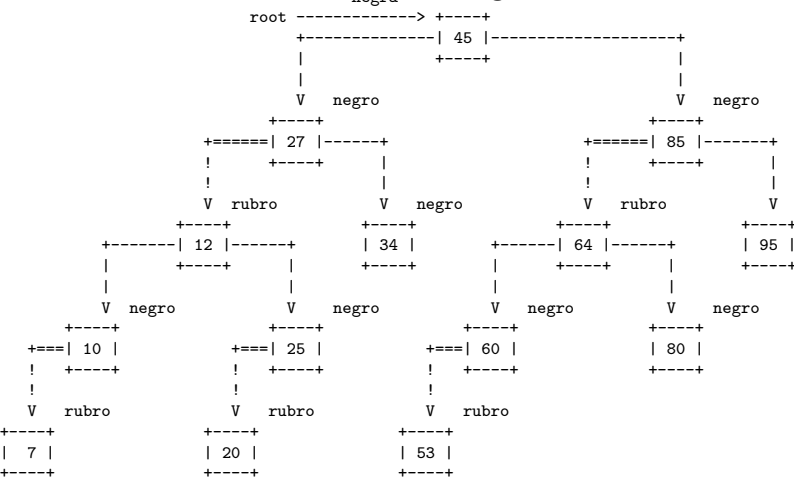
Árvore 2-3 para rubro-negra



Árvore 2-3

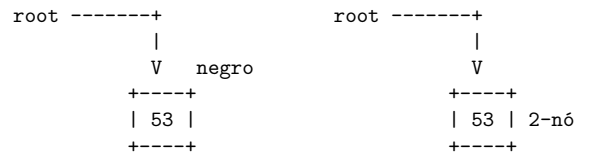


Árvore rubro-negra



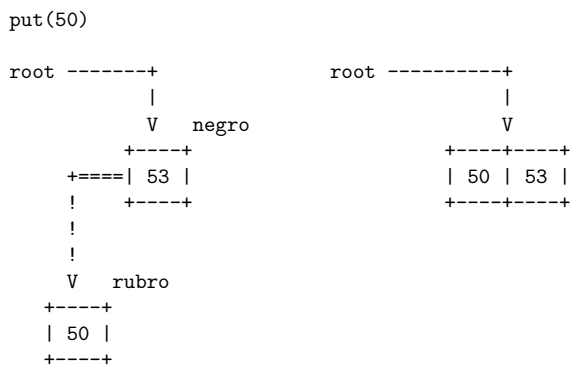
Inserção em um 2-nó

Árvore formada por apenas um 2-nó



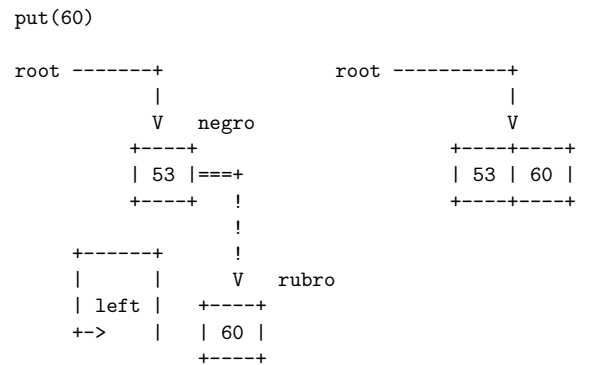
Inserção em um 2-nó

Árvore formada por apenas um 2-nó



Inserção em um 2-nó

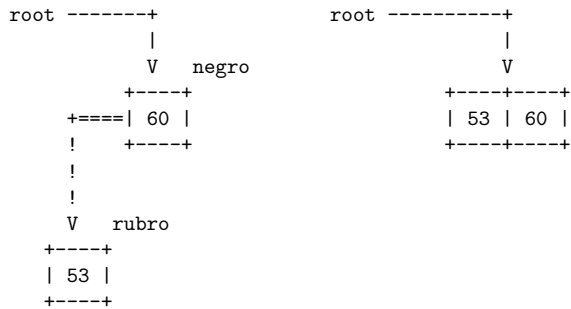
Árvore formada por apenas um 2-nó



Inserção em um 2-nó

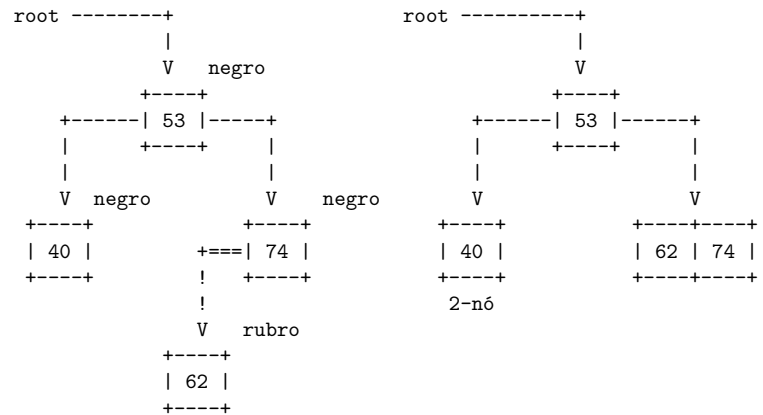
Árvore formada por apenas um 2-nó

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



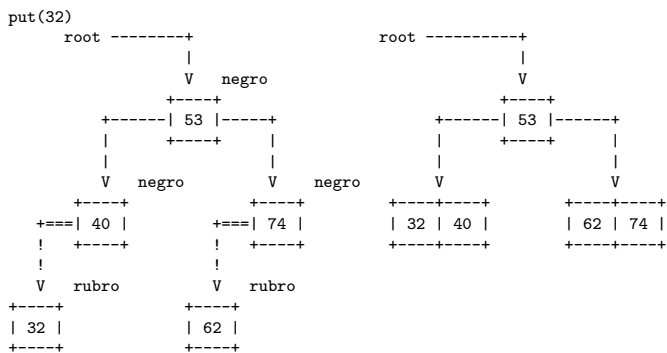
Navigation icons: back, forward, search, etc.

Inserção em um 2-nó qualquer



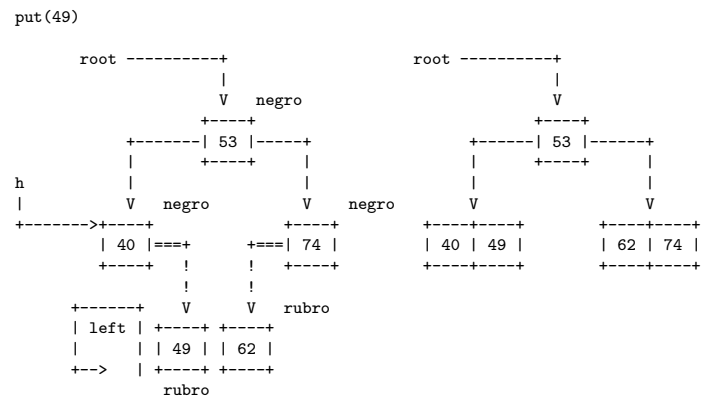
Navigation icons: back, forward, search, etc.

Inserção em um 2-nó qualquer



Navigation icons: back, forward, search, etc.

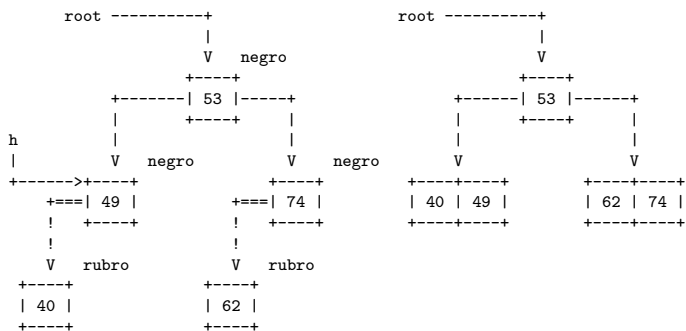
Inserção em um 2-nó qualquer



Navigation icons: back, forward, search, etc.

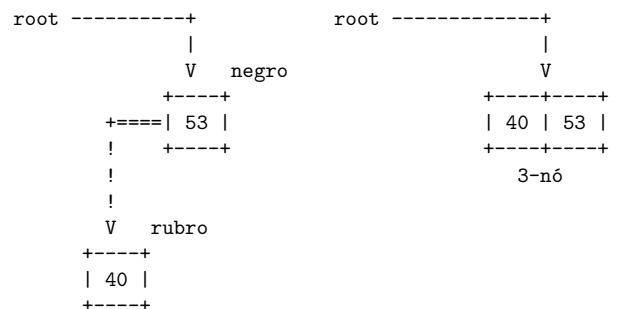
Inserção em um 2-nó qualquer

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



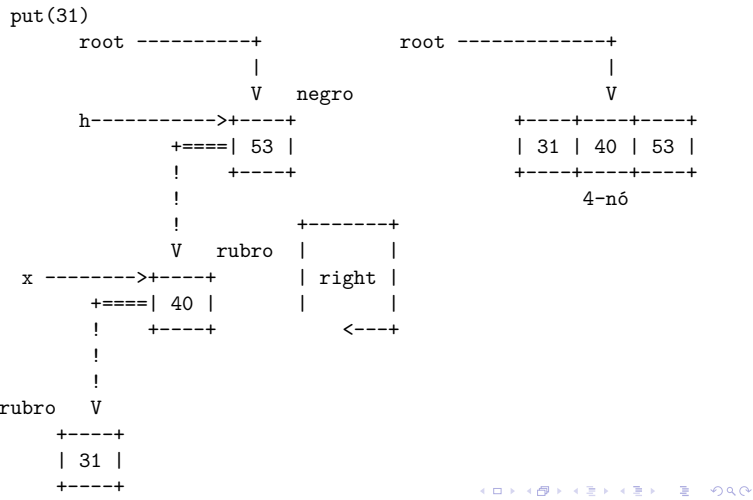
Navigation icons: back, forward, search, etc.

Inserção em um 3-nó

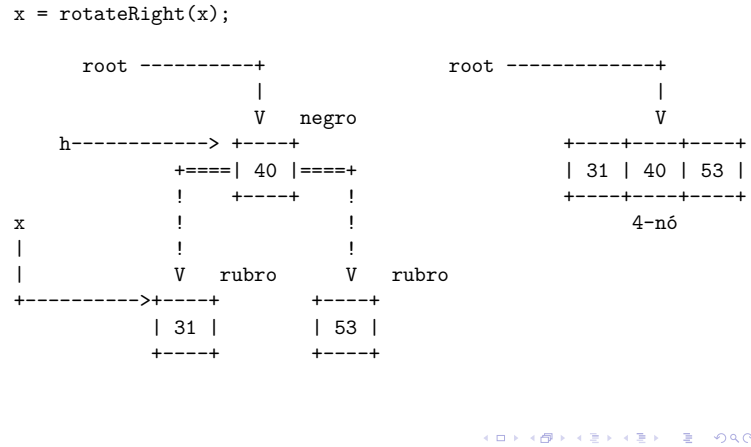


Navigation icons: back, forward, search, etc.

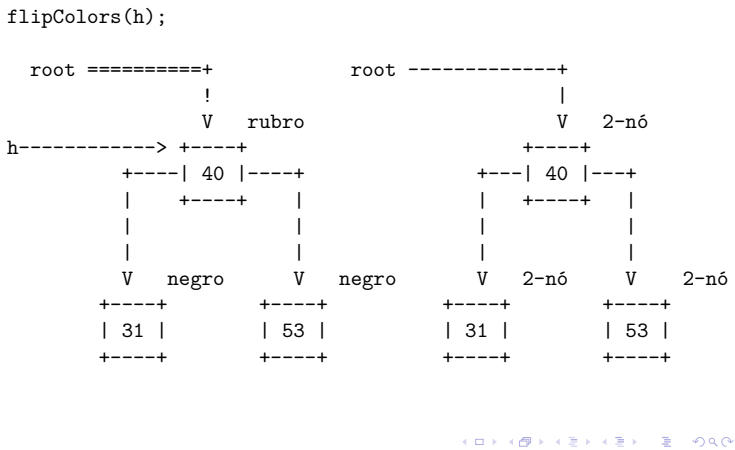
chave é inserida é menor do 3-nó



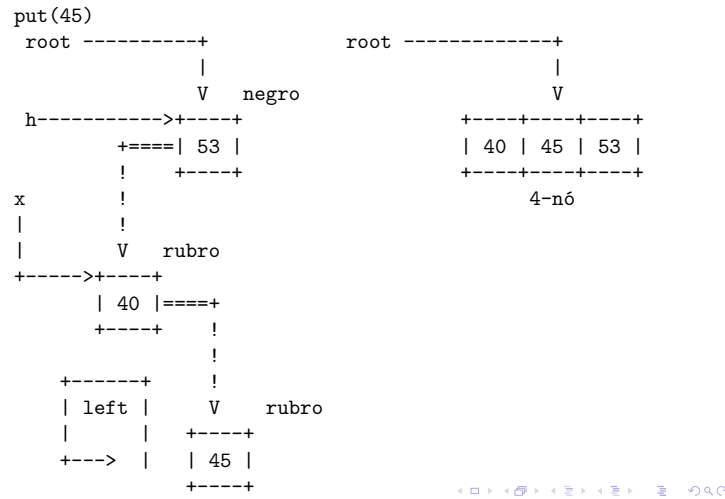
chave é inserida é menor do 3-nó



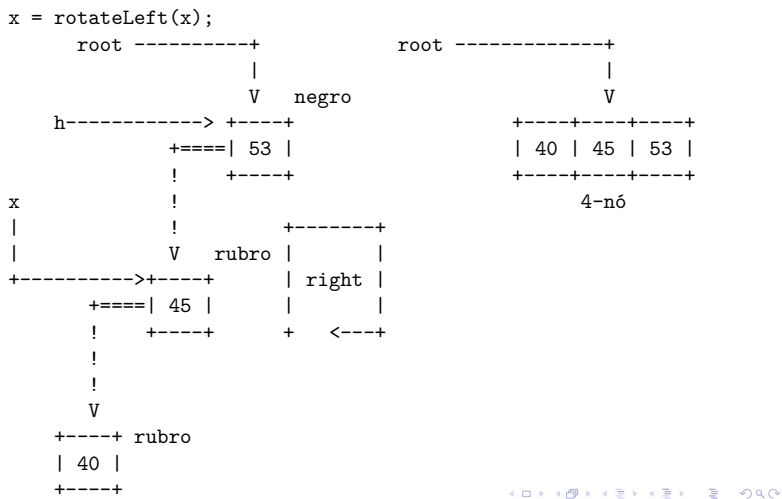
chave é inserida é menor do 3-nó



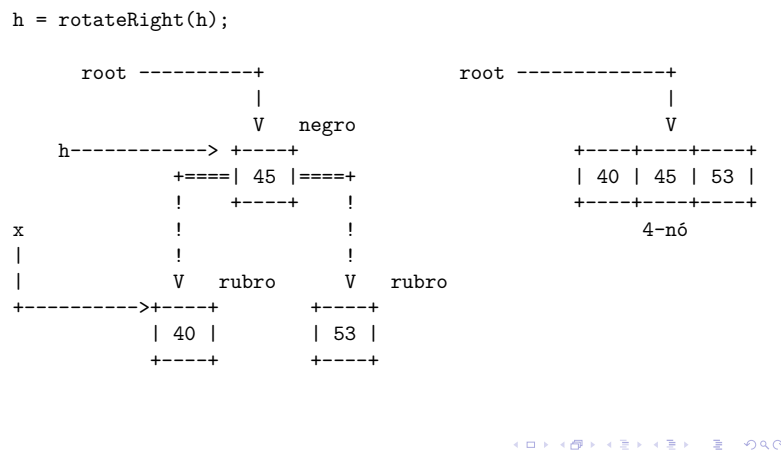
chave é inserida entre as chaves do 3-nó



chave é inserida entre as chaves do 3-nó

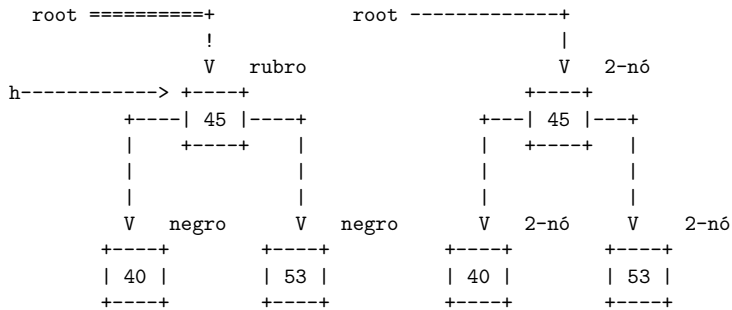


chave é inserida entre as chaves do 3-nó



chave é inserida entre as chaves do 3-nó

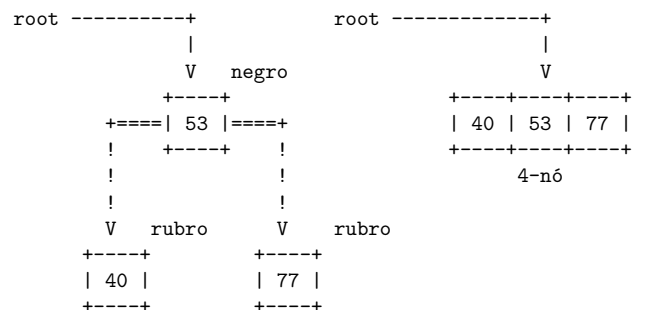
flipColors(h); hmmm. raiz deve ser negra



Navigation icons

chave inserida é maior que todas do 3-nó

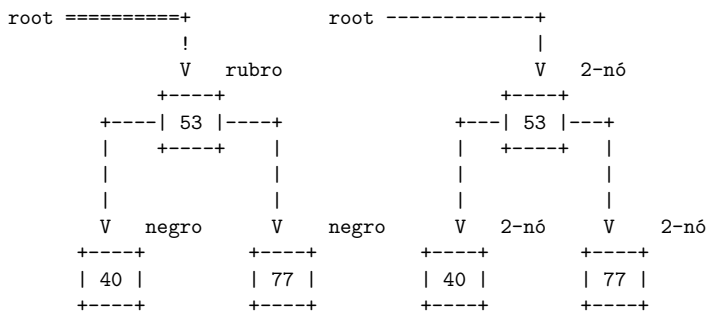
put(77)



Navigation icons

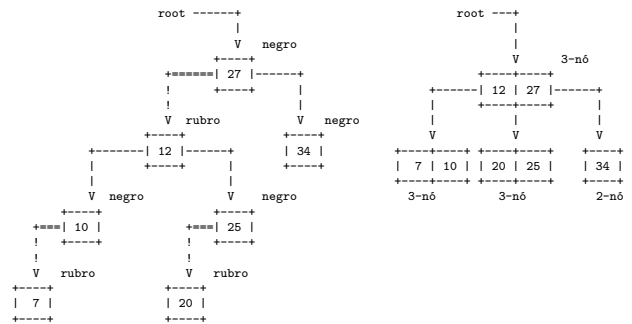
chave inserida é maior que todas do 3-nó

flipColors(root); hmmm. raiz deve ser negra



Navigation icons

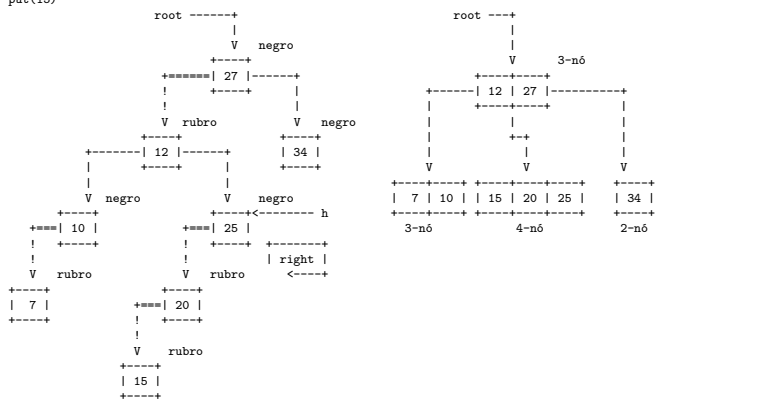
chave é inserida em um 3-nó qualquer



Navigation icons

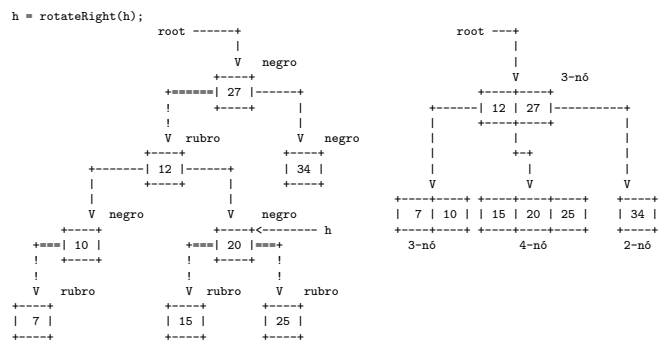
chave é inserida em um 3-nó qualquer

put(15)



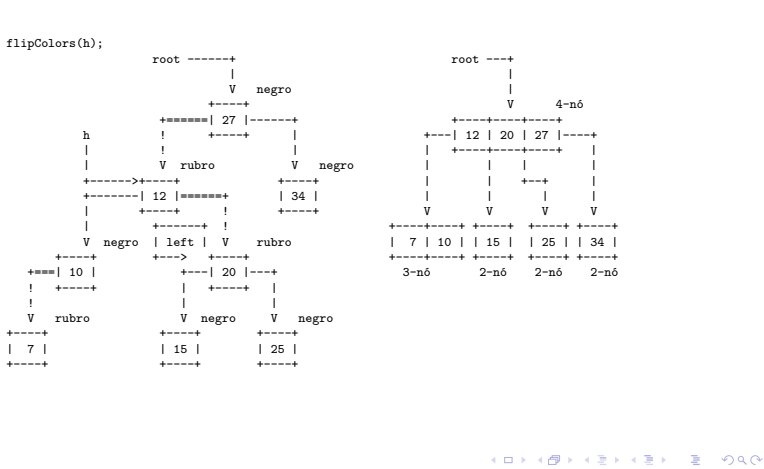
Navigation icons

chave é inserida em um 3-nó qualquer

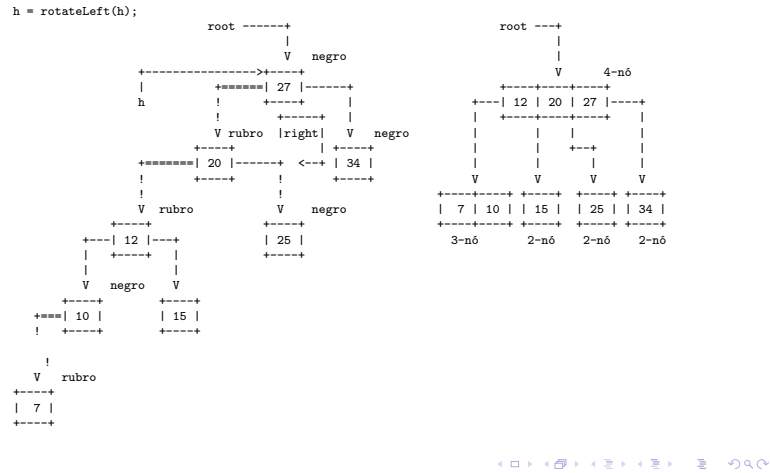


Navigation icons

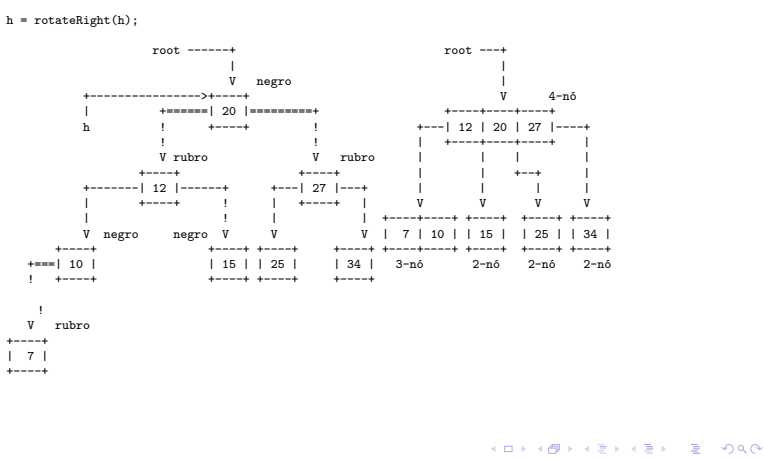
chave é inserida em um 3-nó qualquer



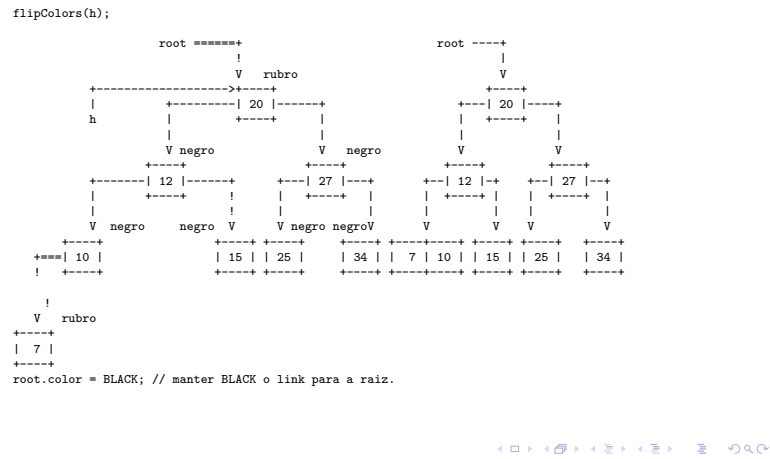
chave é inserida em um 3-nó qualquer



chave é inserida em um 3-nó qualquer



chave é inserida em um 3-nó qualquer



rotações

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

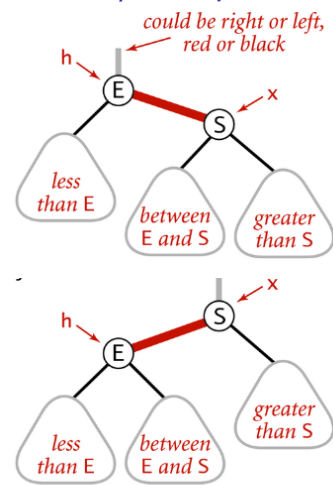
Durante uma operação de inserção, podemos ter, temporariamente, um **link rubro inclinado para a direita** ou **dois links rubros incidindo no mesmo nó**.

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;
}
```

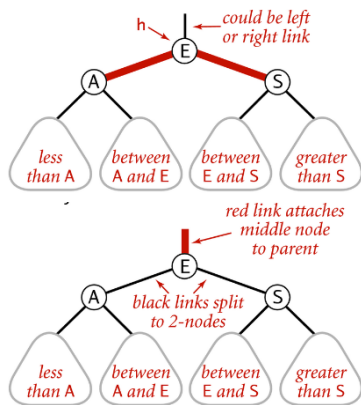
Navigation icons

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;
}
```

Navigation icons

Flipping colors

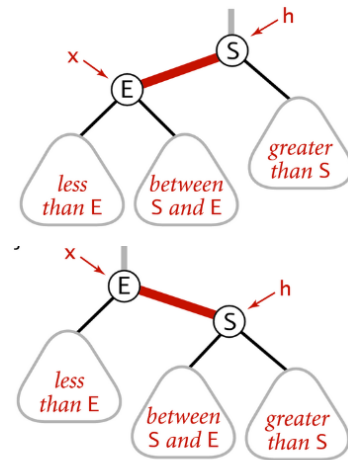


Flipping colors to split a 4-node

Fonte: [algs4](#)

Navigation icons

Rotação direita



Right rotate (left link of h)

Fonte: [algs4](#)

Navigation icons

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.

Navigation icons

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;
}
```

Navigation icons

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() {...}
}
```

Navigation icons

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;
}
```

Navigation icons

BSTs rubro-negras: delete()



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

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

Navigation icons

RedBlackBST

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

Navigation icons

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;
}
```

Navigation icons

Remoção em árvore 2-3

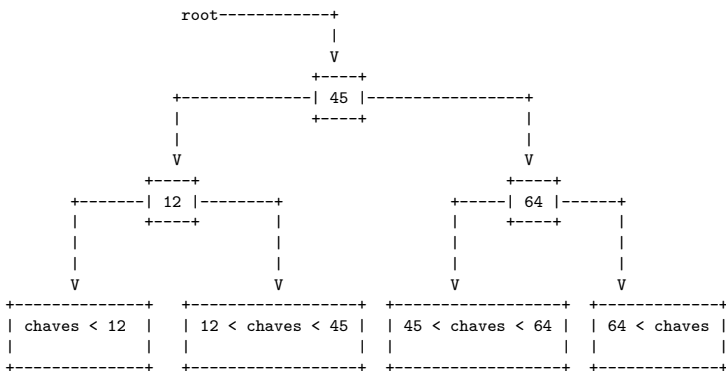
No caminho até a chave a ser **removida** o algoritmo mantém a relação invariante em relação à **árvore 2-3**:

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.

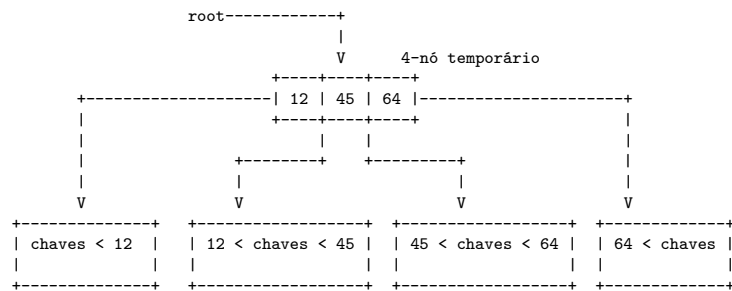
Navigation icons

Os dois filhos da raiz são 2-nós



Navigation icons: back, forward, search, etc.

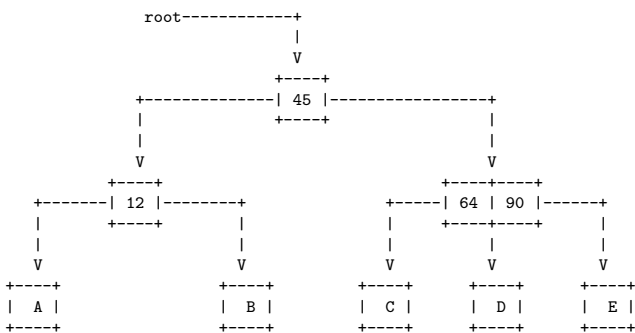
Os dois filhos da raiz são 2-nós



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

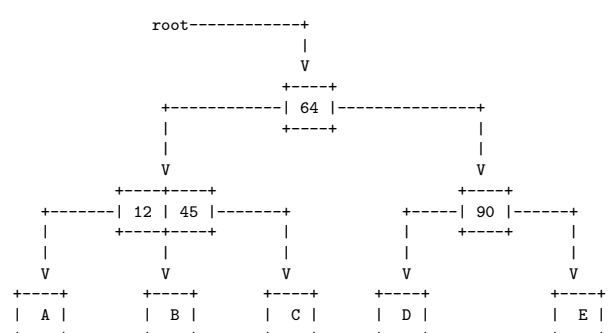
Navigation icons: back, forward, search, etc.

Filhos esquerdo da raiz é 2-nós



Navigation icons: back, forward, search, etc.

Filhos esquerdo da raiz é 2-nós



Navigation icons: back, forward, search, etc.

Mover para esquerda

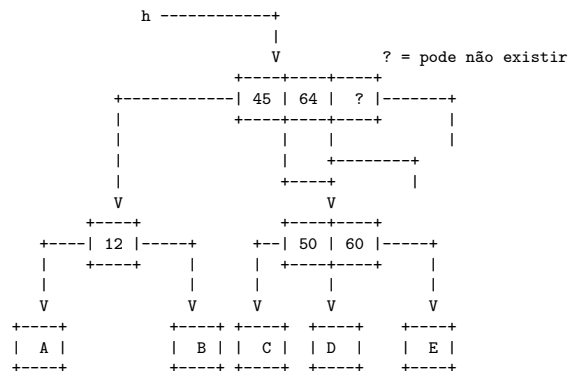
No meio do caminho, sabemos que o nó corrente **h** é um 3-nó ou um 4-nó.

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.

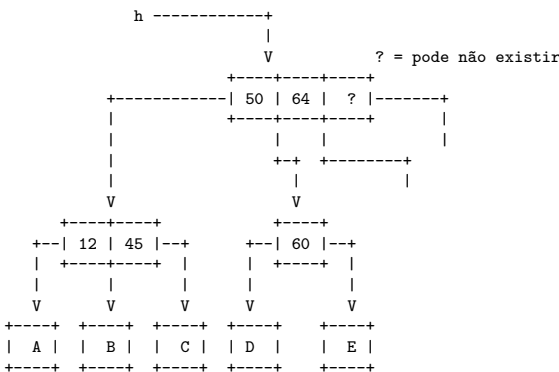
Filho mais a esquerda de h é 2-nó



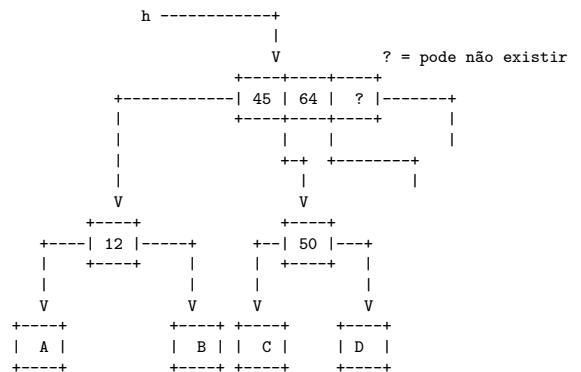
Navigation icons: back, forward, search, etc.

Navigation icons: back, forward, search, etc.

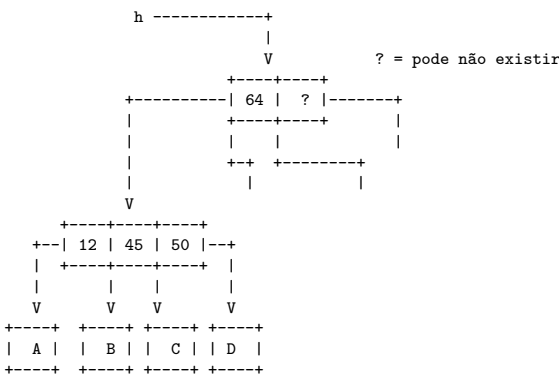
Filho mais a esquerda de h é 2-nó



Dois filhos mais à esquerda de h são 2-nós



Dois filhos mais à esquerda de h são 2-nós



Finalmente...

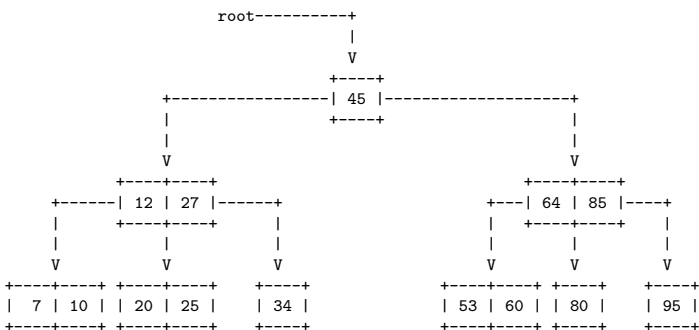
Desta forma quando atingirmos a folha mais a esquerda dessa **árvore 2-3-4** teremos chegado a um **3-nó** ou **4-no**.

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

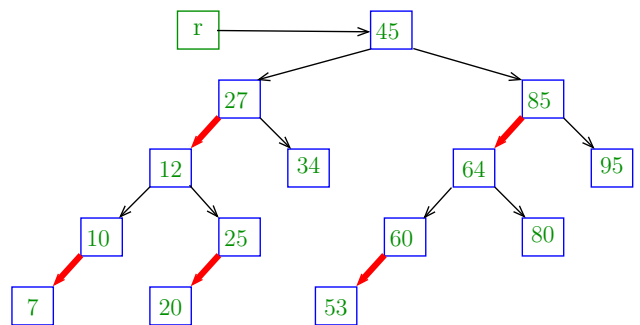
Depois devemos voltar "espatifando" os **4-nós** que por ventura deixamos pelo caminho.

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()

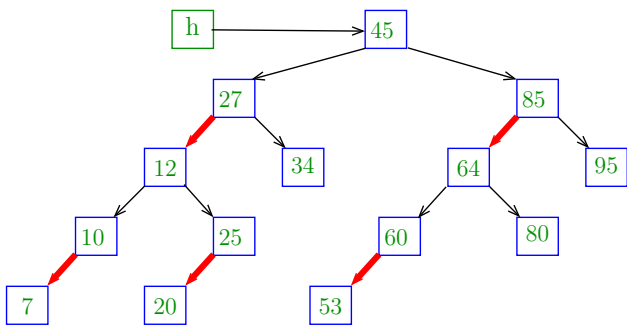


rubro-negra: deleteMin()



`deleteMin(r);`

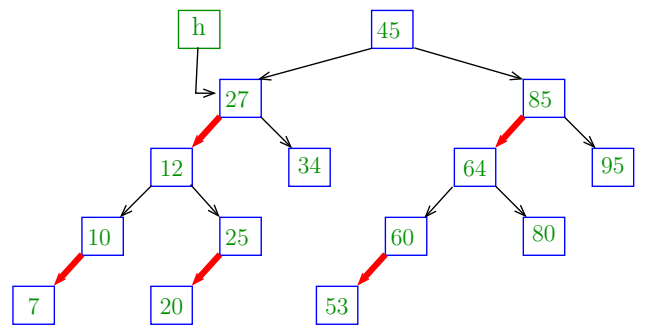
rubro-negra: deleteMin()



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



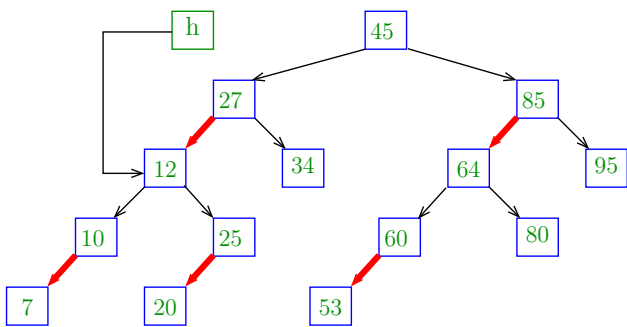
rubro-negra: deleteMin()



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



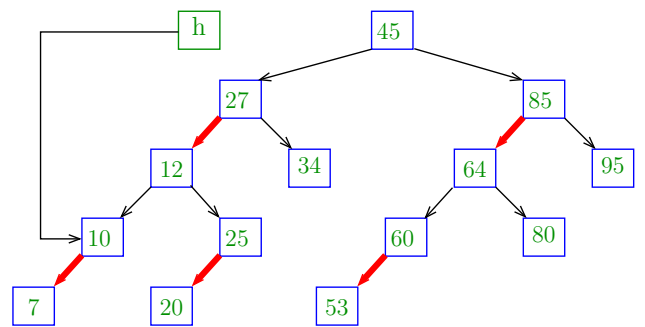
rubro-negra: deleteMin()



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



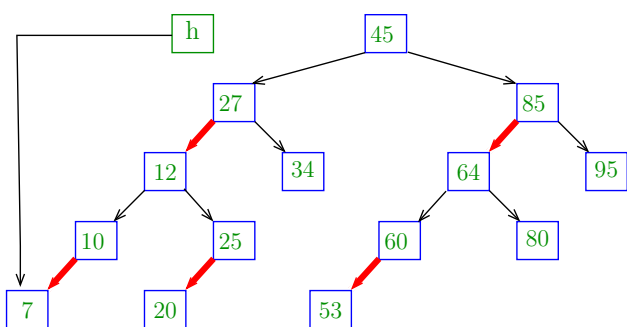
rubro-negra: deleteMin()



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



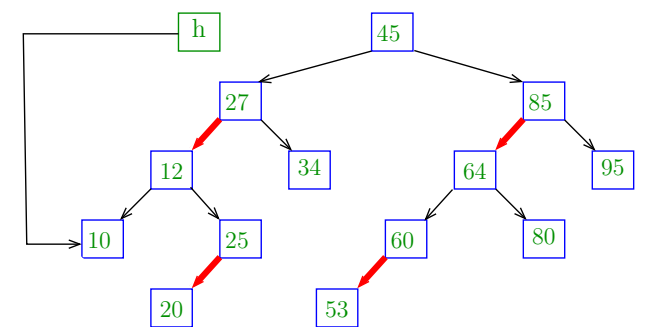
rubro-negra: deleteMin()



```
return null;
```



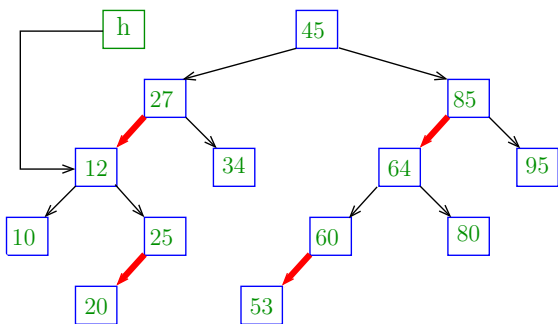
rubro-negra: deleteMin()



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



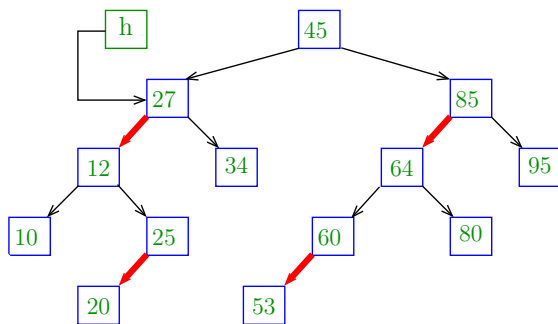
rubro-negra: deleteMin()



return balance(h);

Navigation icons

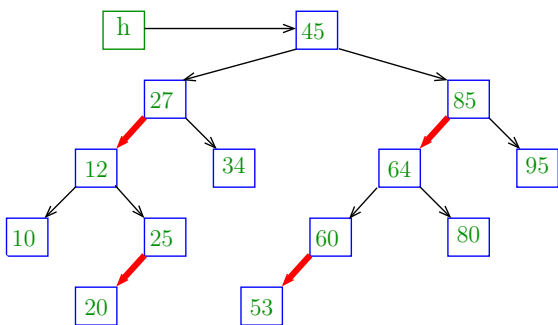
rubro-negra: deleteMin()



return balance(h);

Navigation icons

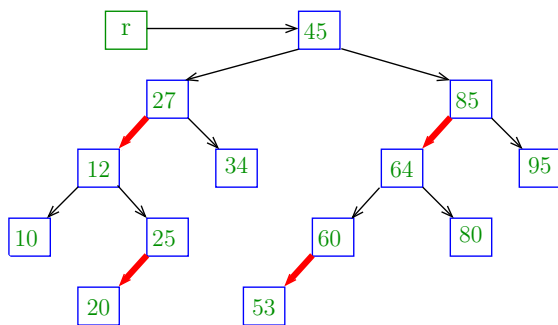
rubro-negra: deleteMin()



return balance(h);

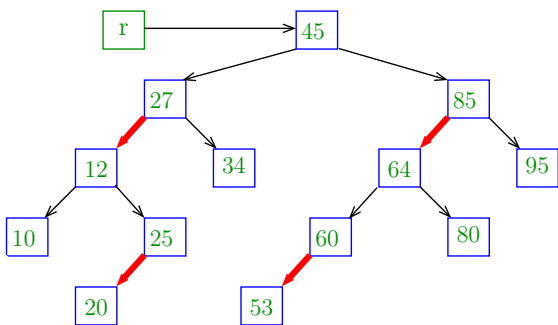
Navigation icons

rubro-negra: deleteMin()



Navigation icons

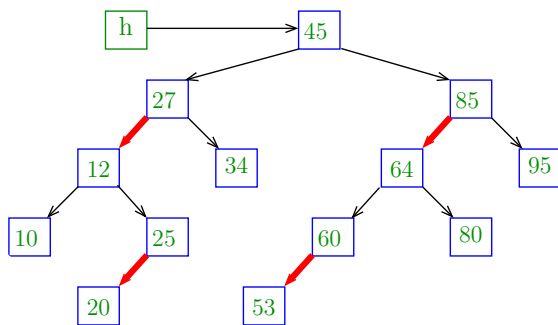
rubro-negra: outro deleteMin()



deleteMin(r);

Navigation icons

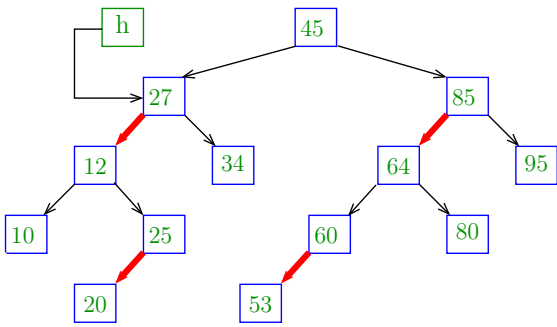
rubro-negra: outro deleteMin()



h.left = deleteMin(h.left);

Navigation icons

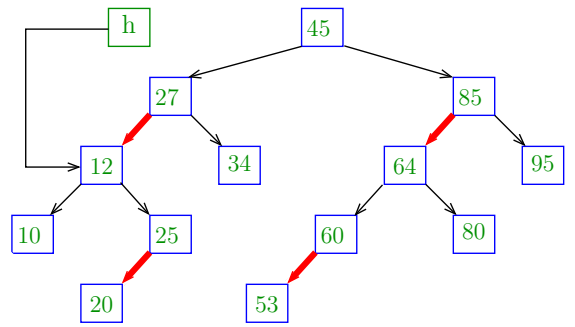
rubro-negra: outro deleteMin()



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



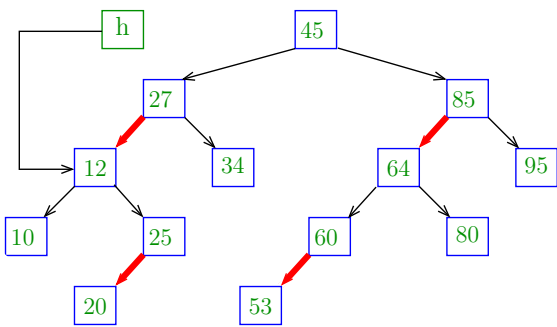
rubro-negra: outro deleteMin()



`h = moveRedLeft(h);`



rubro-negra: outro deleteMin()

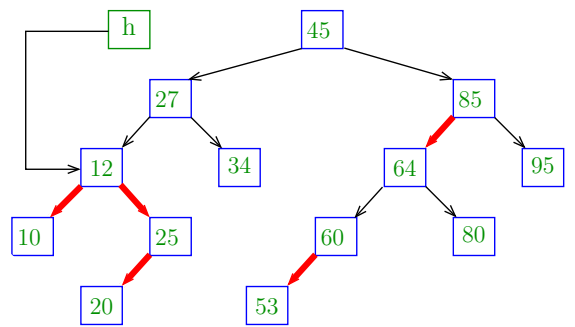


`flipColors(h);`

`[moveRedLeft(h);`



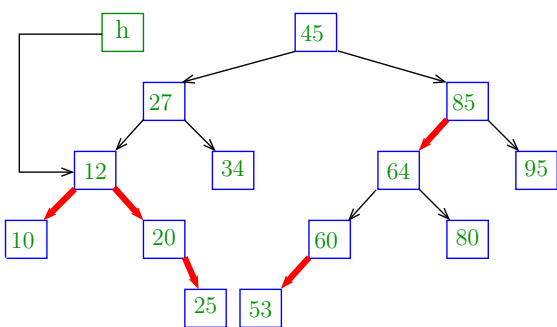
rubro-negra: outro deleteMin()



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



rubro-negra: outro deleteMin()

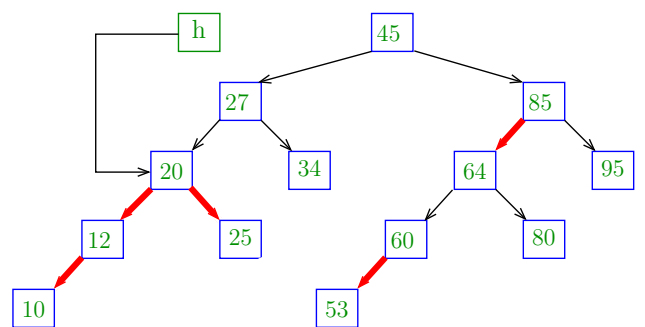


`h = rotateLeft(h);`

`[moveRedLeft(h);`



rubro-negra: outro deleteMin()

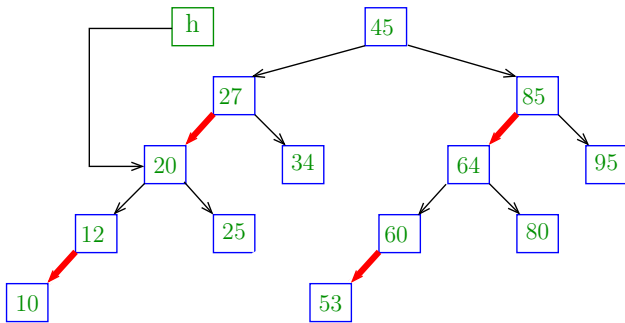


`flipColors(h);`

`[moveRedLeft(h);`



rubro-negra: outro deleteMin()

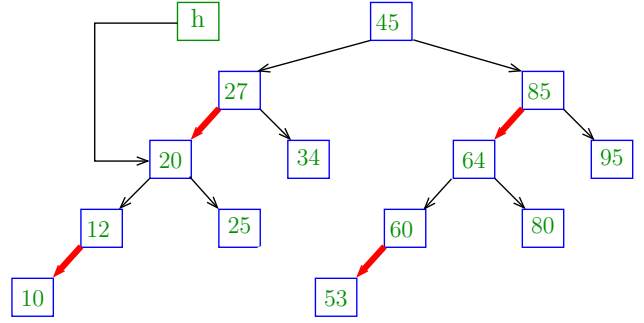


return h;

[moveRedLeft(h);]

Navigation icons

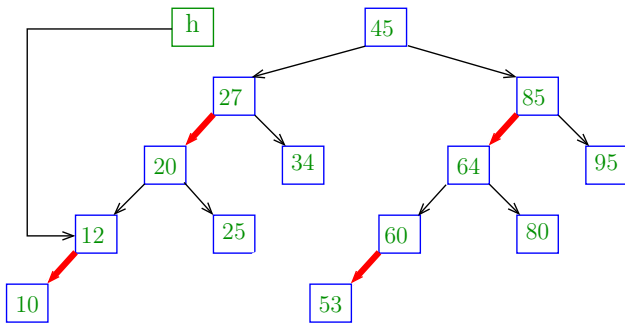
rubro-negra: outro deleteMin()



h.left = deleteMin(h.left);

Navigation icons

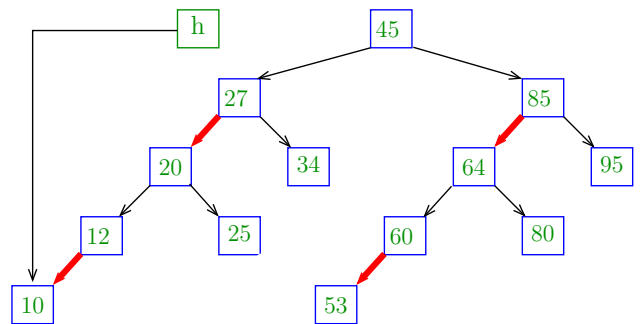
rubro-negra: outro deleteMin()



h.left = deleteMin(h.left);

Navigation icons

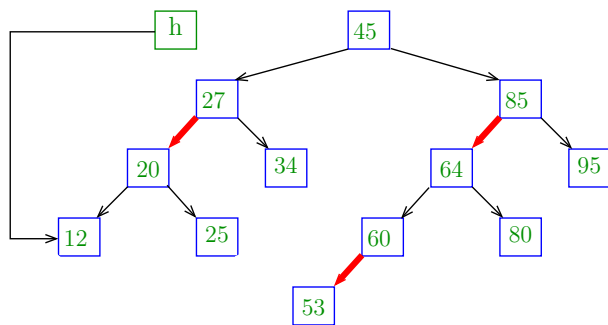
rubro-negra: outro deleteMin()



return null;

Navigation icons

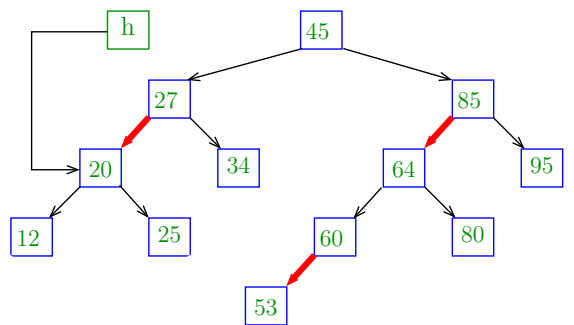
rubro-negra: outro deleteMin()



return balance(h);

Navigation icons

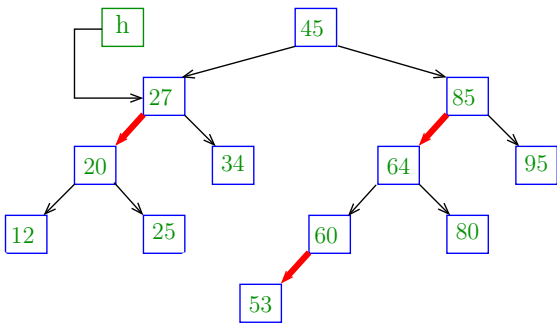
rubro-negra: outro deleteMin()



return balance(h);

Navigation icons

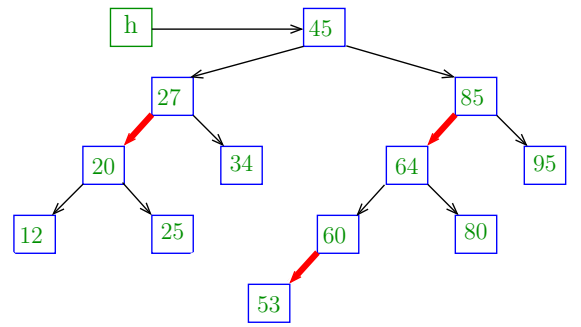
rubro-negra: outro deleteMin()



return balance(h);



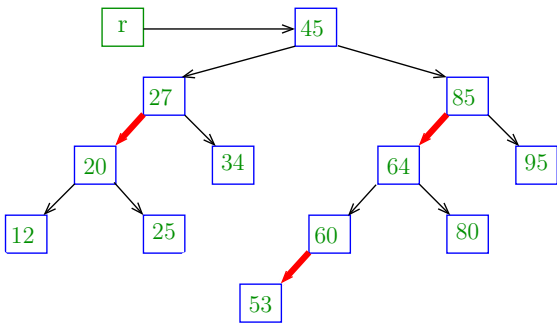
rubro-negra: outro deleteMin()



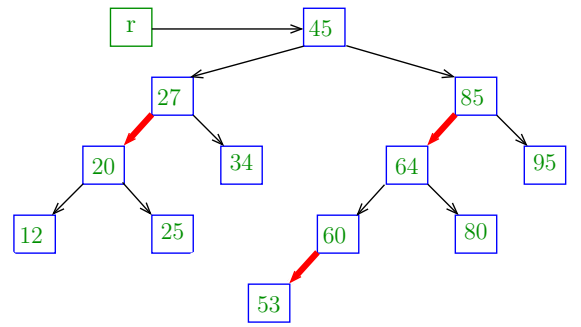
return balance(h);



rubro-negra: outro deleteMin()



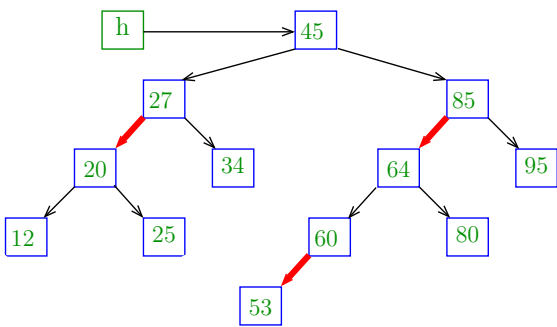
rubro-negra: mais outro deleteMin()



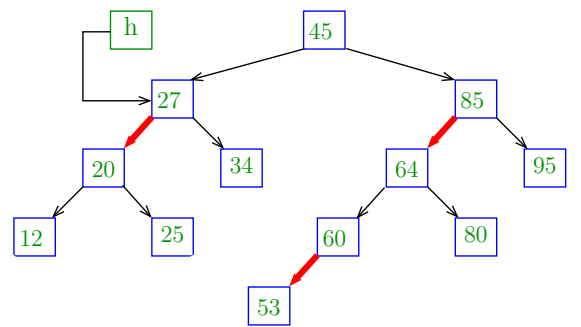
deleteMin(r);



rubro-negra: mais outro deleteMin()



rubro-negra: mais outro deleteMin()



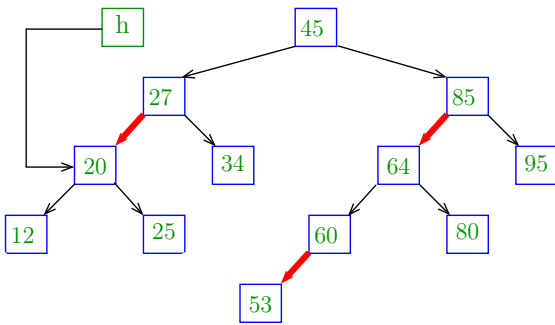
h.left = deleteMin(h.left);



h.left = deleteMin(h.left);



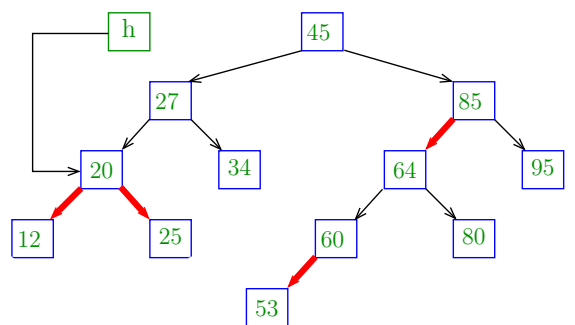
rubro-negra: mais outro deleteMin()



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

Navigation icons

rubro-negra: mais outro deleteMin()

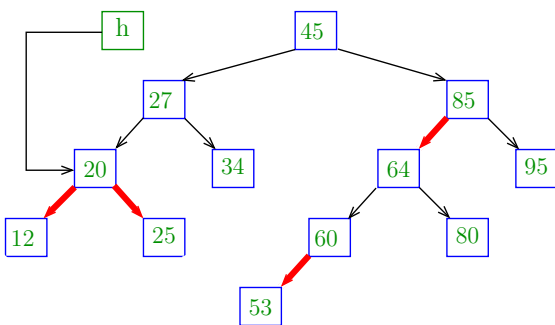


```
flipColors(h);
```

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

Navigation icons

rubro-negra: mais outro deleteMin()

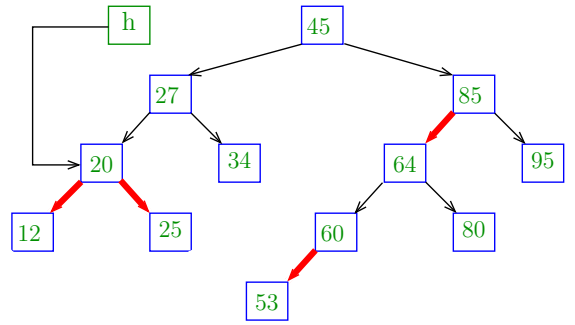


```
return h
```

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

Navigation icons

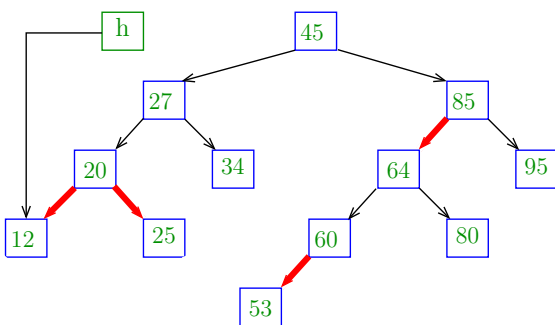
rubro-negra: mais outro deleteMin()



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

Navigation icons

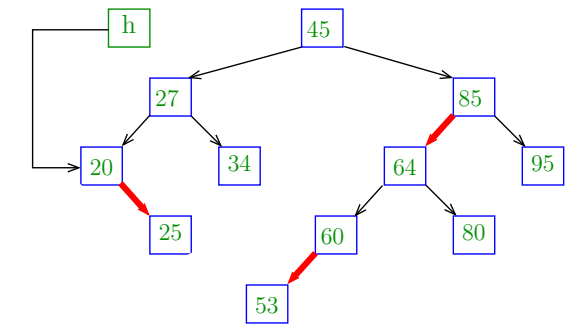
rubro-negra: mais outro deleteMin()



```
return null
```

Navigation icons

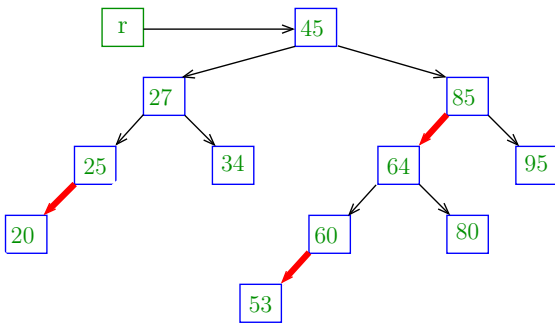
rubro-negra: mais outro deleteMin()



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

Navigation icons

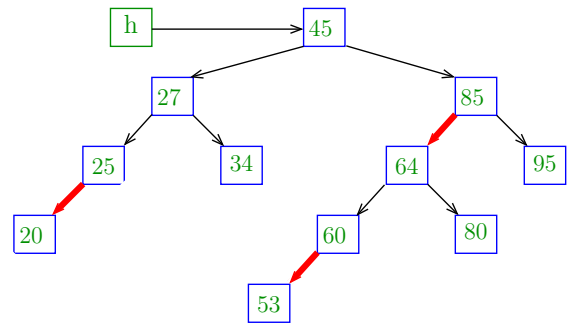
rubro-negra: mais outro deleteMin() ainda



deleteMin(r);



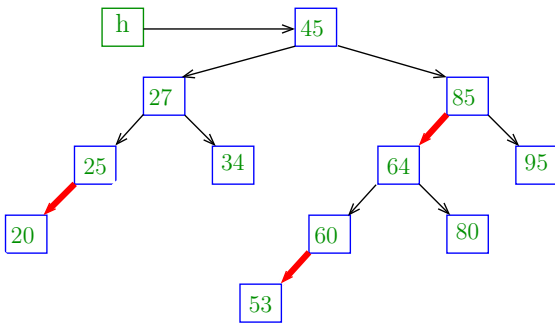
rubro-negra: mais outro deleteMin() ainda



h.left = deleteMin(h.left);



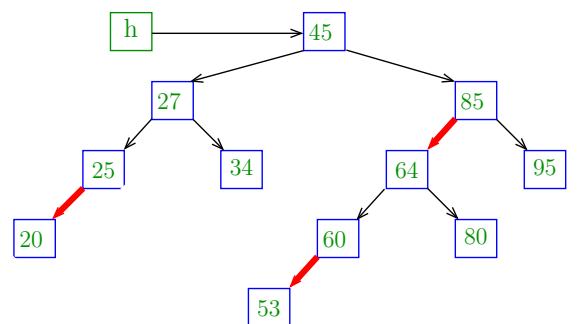
rubro-negra: mais outro deleteMin() ainda



h = moveRedLeft(h);



rubro-negra: mais outro deleteMin() ainda

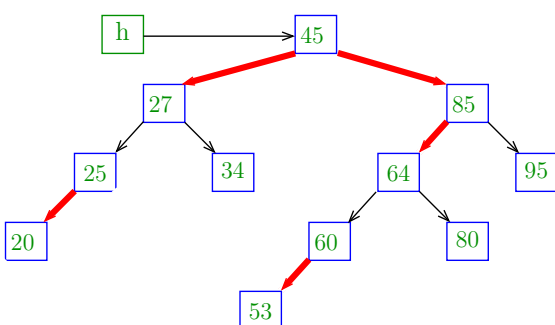


flipColors(h)

[moveRedLeft(h);



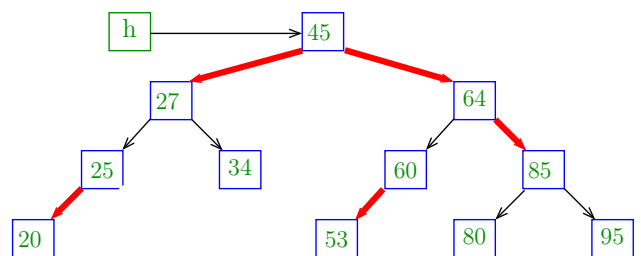
rubro-negra: mais outro deleteMin() ainda



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



rubro-negra: mais outro deleteMin() ainda

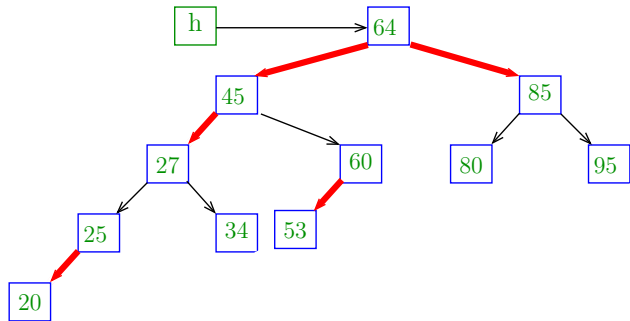


h = rotateLeft(h);

[moveRedLeft(h);



rubro-negra: mais outro deleteMin() ainda

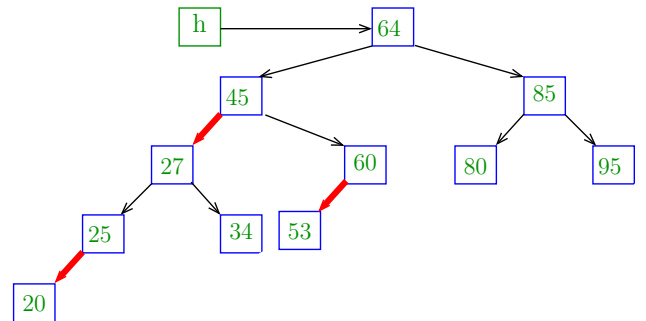


flipColors(h)

[moveRedLeft(h)];



rubro-negra: mais outro deleteMin() ainda

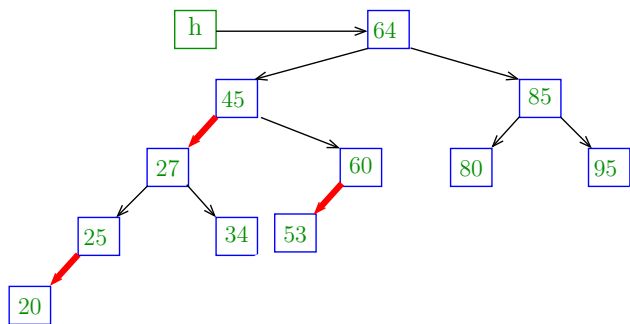


return h;

[moveRedLeft(h)];



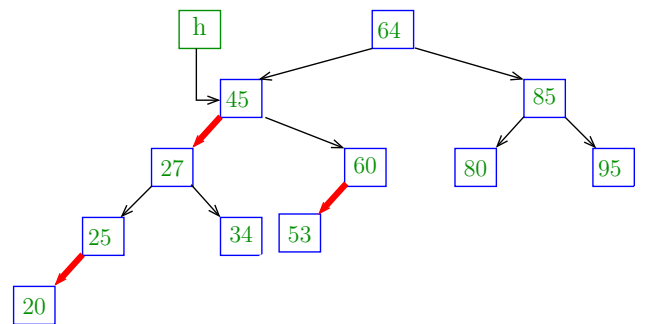
rubro-negra: mais outro deleteMin() ainda



h.left = deleteMin(h.left);



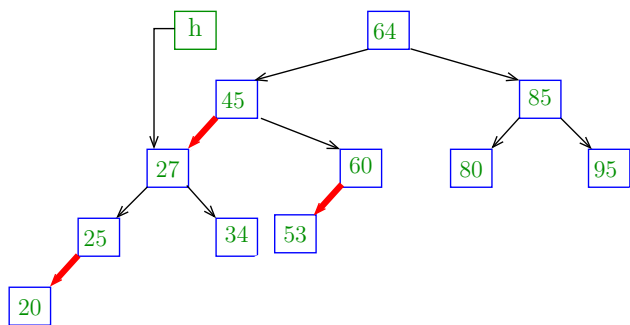
rubro-negra: mais outro deleteMin() ainda



h.left = deleteMin(h.left);



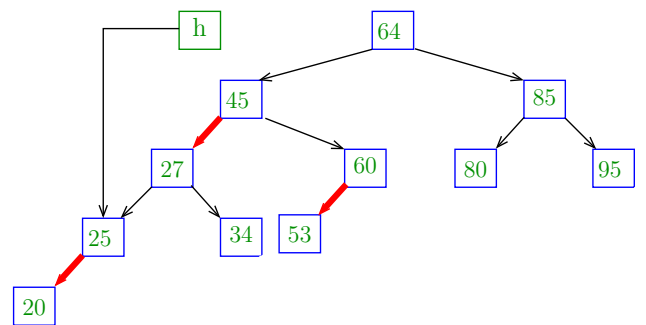
rubro-negra: mais outro deleteMin() ainda



h.left = deleteMin(h.left);



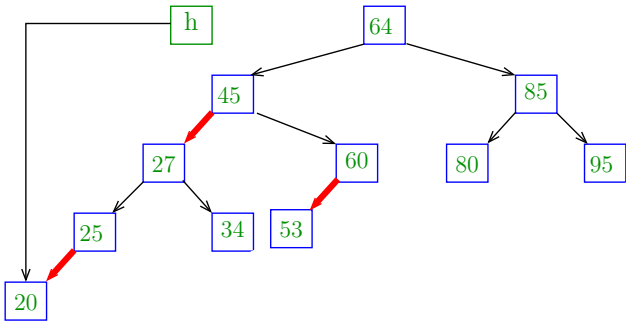
rubro-negra: mais outro deleteMin() ainda



h.left = deleteMin(h.left);



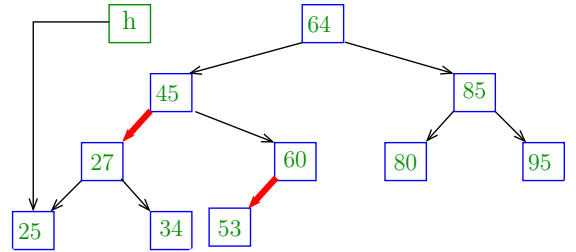
rubro-negra: mais outro deleteMin() ainda



return null;



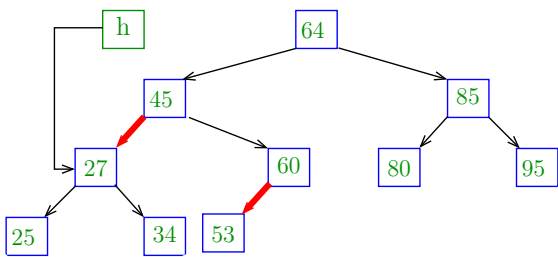
rubro-negra: mais outro deleteMin() ainda



return balance(h);



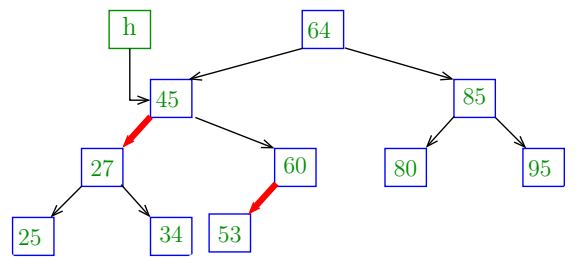
rubro-negra: mais outro deleteMin() ainda



return balance(h);



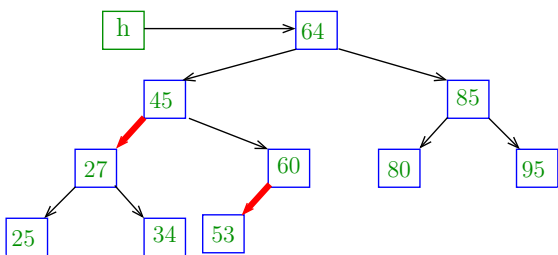
rubro-negra: mais outro deleteMin() ainda



return balance(h);



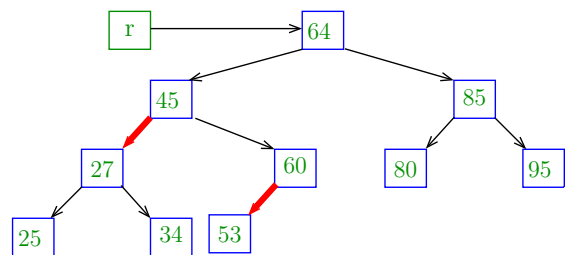
rubro-negra: mais outro deleteMin() ainda



return balance(h);



rubro-negra: mais outro deleteMin() ainda



deleteMin()

Se ambos os filhos da raiz são **negros** faz a 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;
}
```

Navigation icons: back, forward, search, etc.

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);
}
```

Navigation icons: back, forward, search, etc.

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;
}
```

Navigation icons: back, forward, search, etc.

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: back, forward, search, etc.

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: back, forward, search, etc.

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: back, forward, search, etc.

