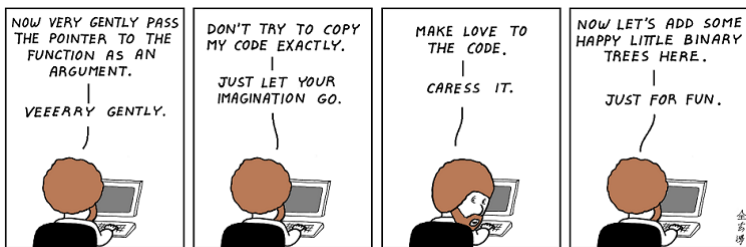


AULA 11

Árvores 2-3



The Joy of Programming
with Bob Ross

Fonte: <https://br.pinterest.com/>

Referências: Árvores 2-3 (PF); Balanced Search Trees (S&W); slides (S&W)

Árvores 2-3

Como implementar uma **tabela de símbolos** em uma **BST** de modo que a árvore permaneça **aproximadamente balanceada**?

Desejamos que a **BST** tenha altura próxima de $\lg n$, sendo n o número de nós, qualquer que seja a sequência de buscas e inserções aplicada à árvore.

Veremos **árvores 2-3** que resolvem o problema em princípio.

A implementação da ideia, usando **árvores rubro-negras**, ainda será discutida.

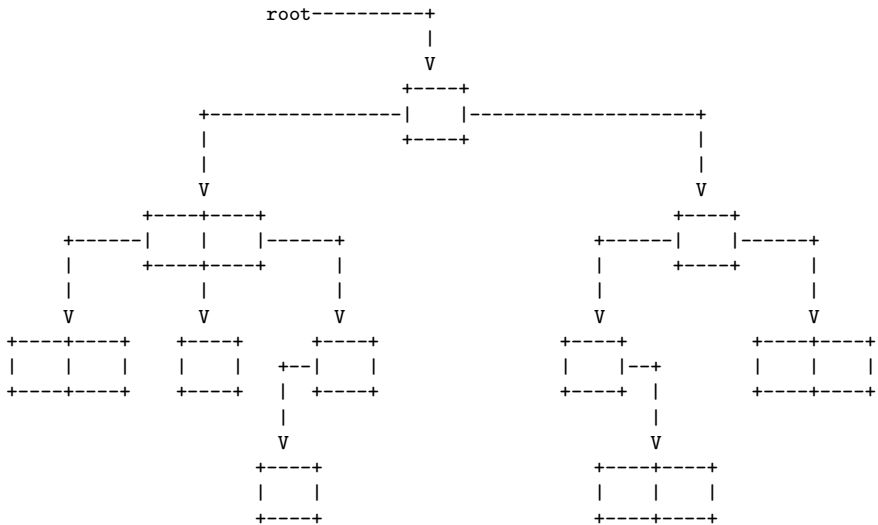
Árvore 2-3

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.

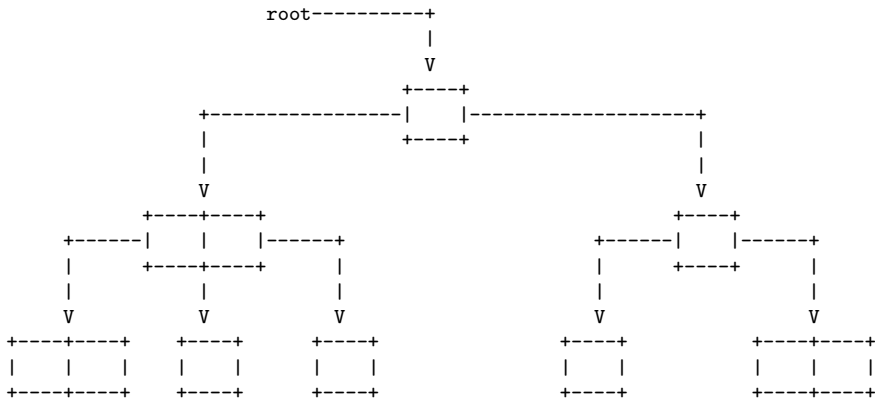
Árvores 2-3 têm esse nome porque cada nó tem **2** ou **3 links**.

Ilustração de árvore 2-3



Árvore 2-3 perfeitamente balanceadas

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



Estrutura

Importante. Para nós **árvore 2-3** é **sinônimo** de **árvore 2-3 perfeitamente balanceada**.

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

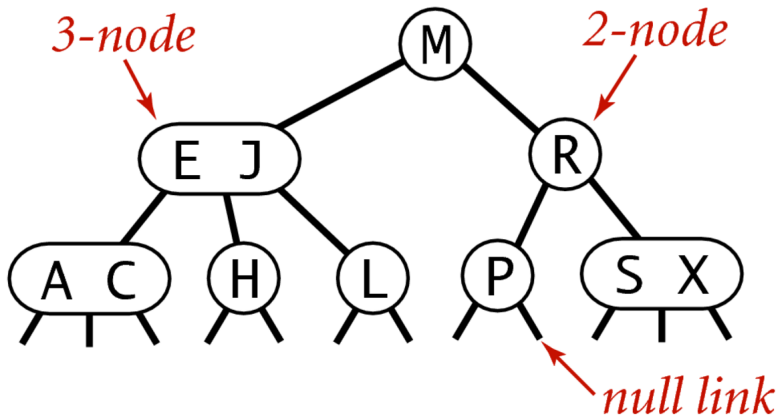
Consequência. Toda **árvore 2-3** com **n** nós tem altura **não superior** a $\lg(n + 1) - 1$ e **não inferior** a $\log_3(n + 1) - 1$.

Árvore 2-3 de busca

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

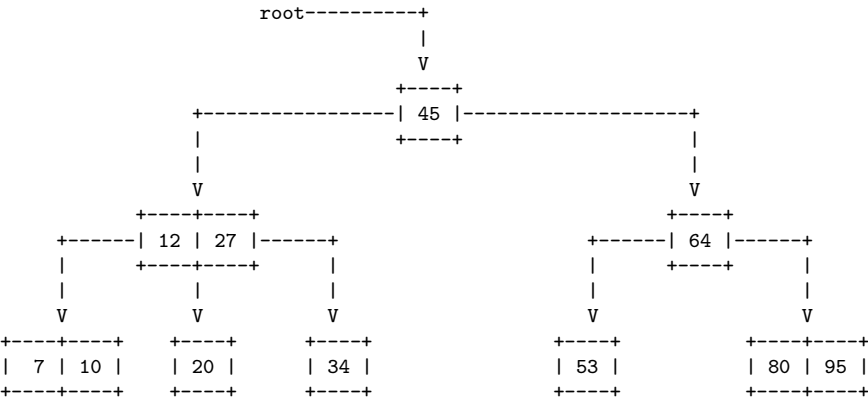
- ▶ uma árvore vazia;
- ▶ ou um nó simples com uma chave e **2 links**:
 - ▶ um link **left** para uma árvore 2-3 que tem **chaves menores** que a chave do nó e
 - ▶ um link **right** para uma árvore 2-3 que tem **chaves maiores**;
- ▶ ou um **nó duplo** com duas chave e **3 links**:
 - ▶ um link **left**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

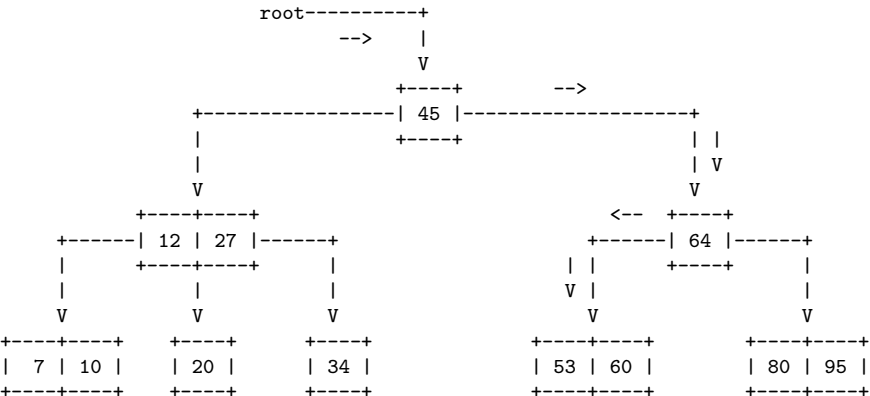
Exemplo de árvore 2-3 de busca



Missão

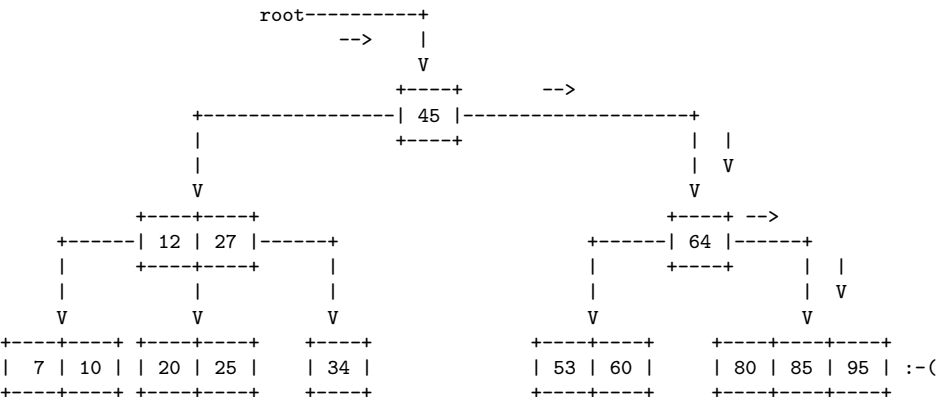
Missão. Manter uma **árvore 2-3 de busca** sujeita a operações de atualização como `put()`, `deleteMin()`, `delete()`,

put(60)



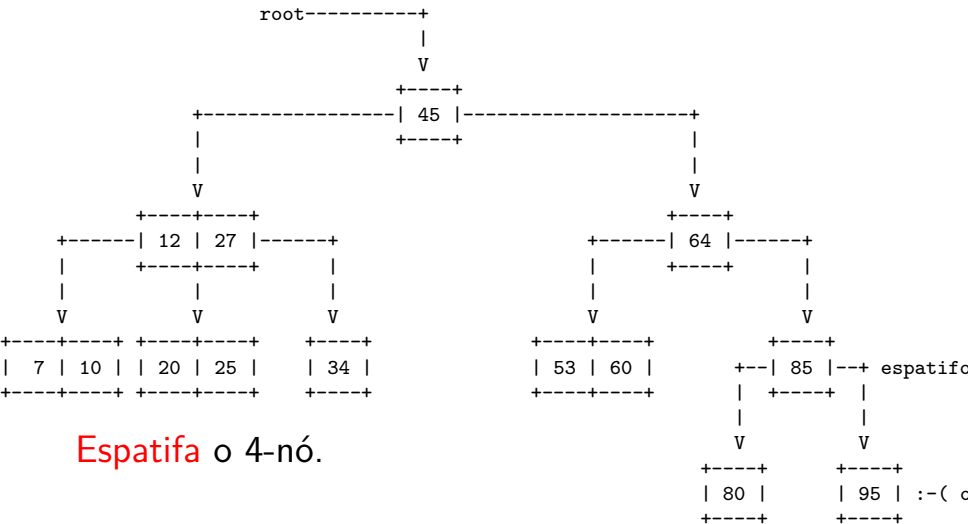
Inserção em um nó simples (**não** estraga estrutura):
Procura 60 e **transforma** um **2-nó** em **3-nó**.

put(85)

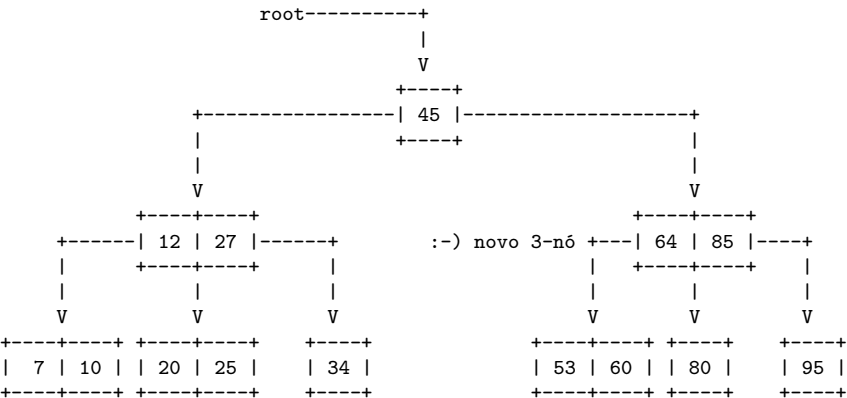


Inserção em um nó duplo (**estraga** estrutura):
procura 85 e **transforma** um 3-nó em 4-nó.

put(85)

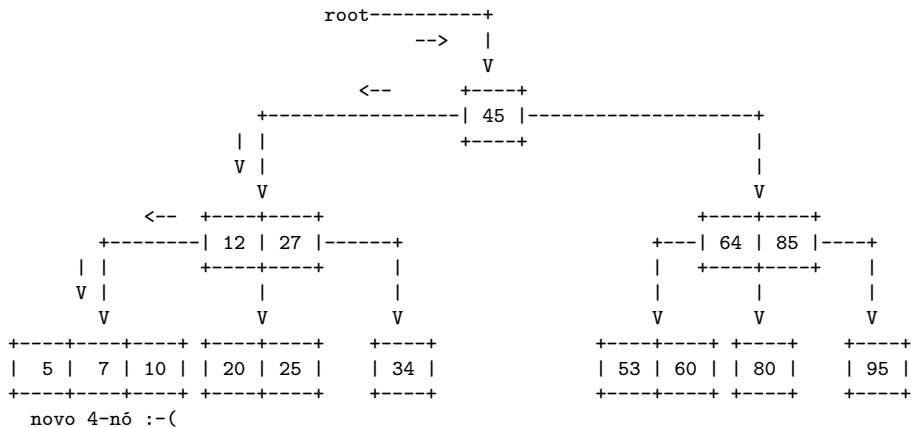


put(85)



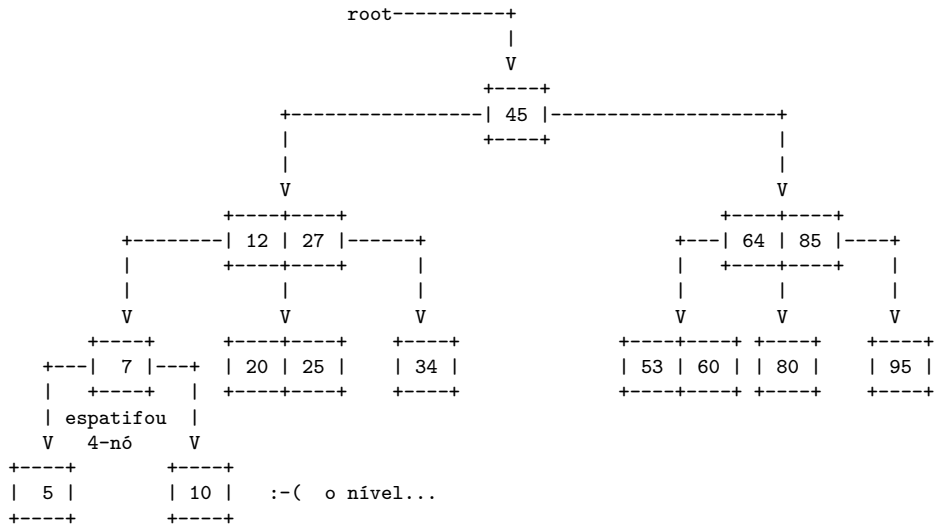
Transforma o 2-nó pai em 3-nó.

put(5)



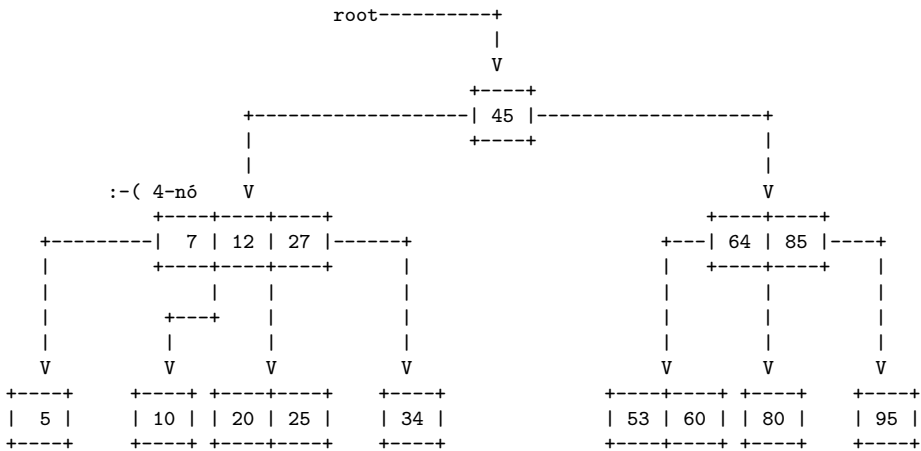
Procura 5 e transforma um 3-nó em 4-nó.

put(5)



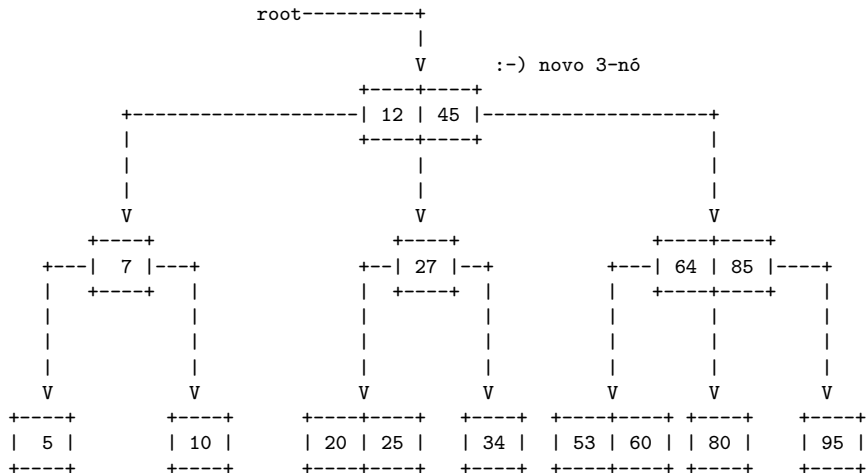
Espatifa o 4-nó.

put(5)



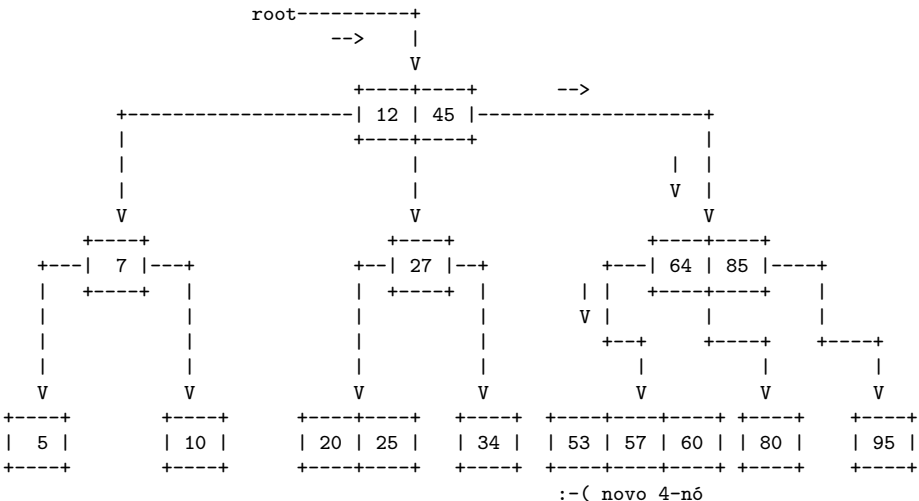
Transforma o 3-nó pai em 4-nó.

put(5)



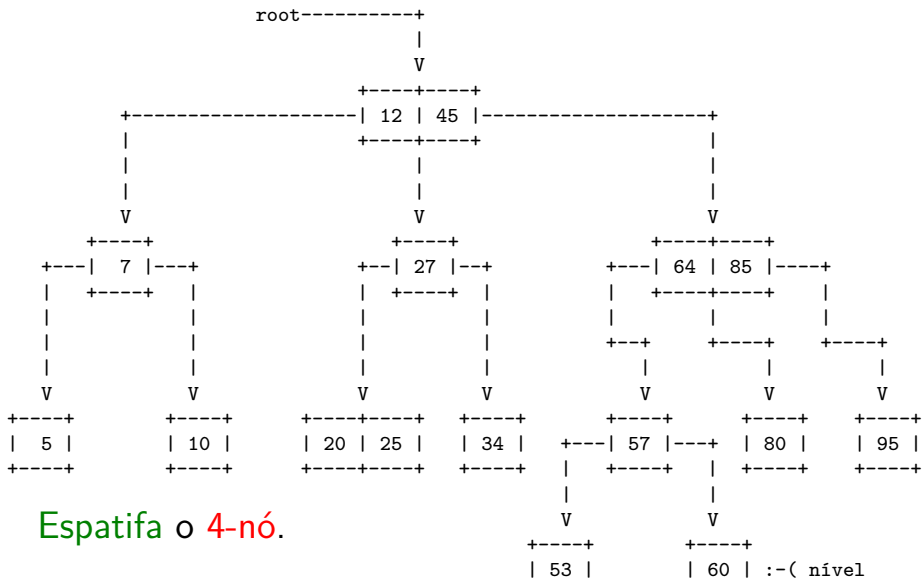
Transforma o 2-nó pai em 3-nó.

put(57)

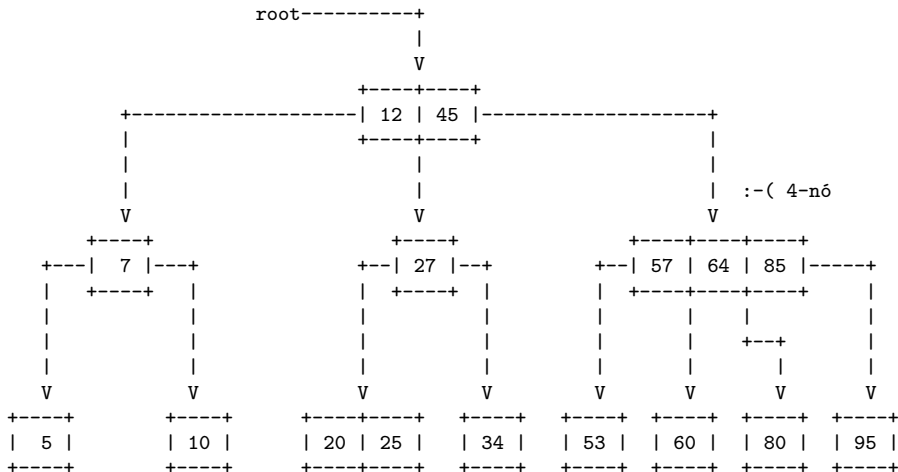


Procura 57 e transforma o 3-nó em 4-nó.

put(57)

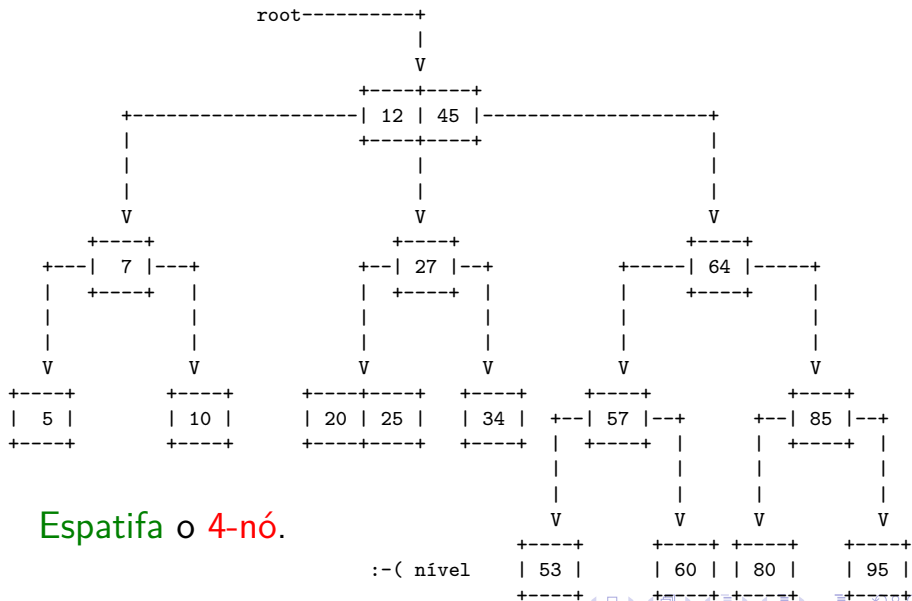


put(57)



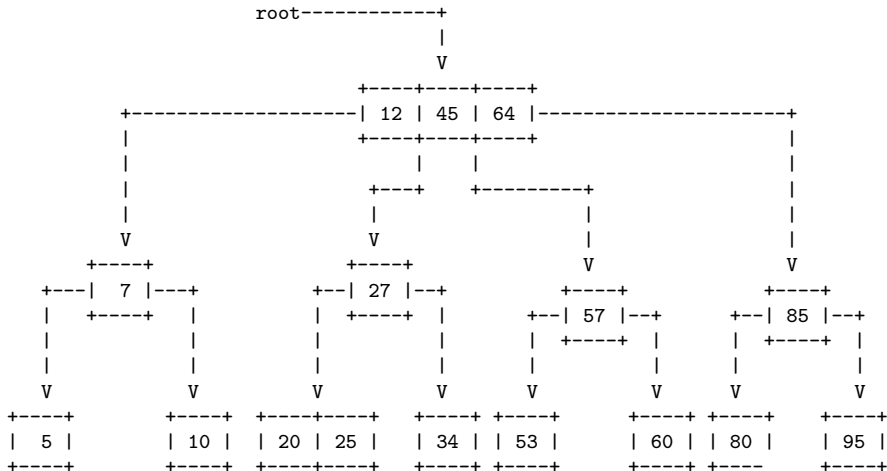
Transforma o 3-nó pai em 4-nó.

put(57)



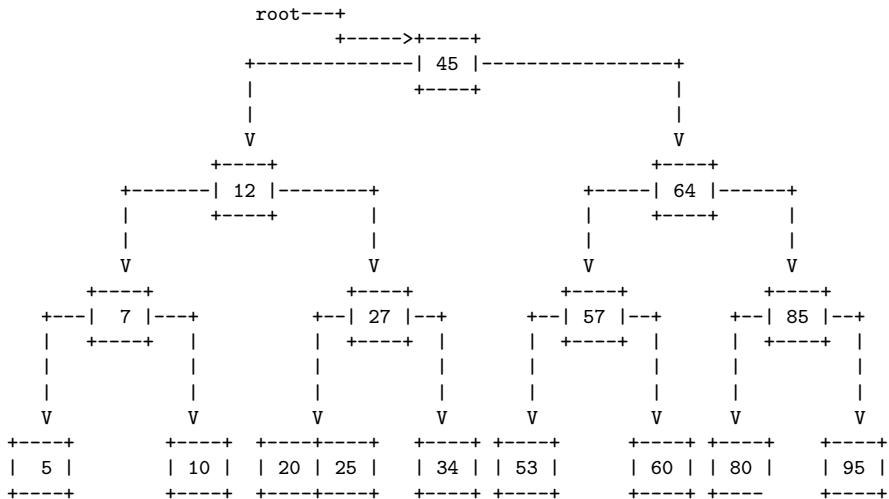
Espatifa o 4-nó.

put(57)



Transforma o 3-nó pai em um 4-nó.

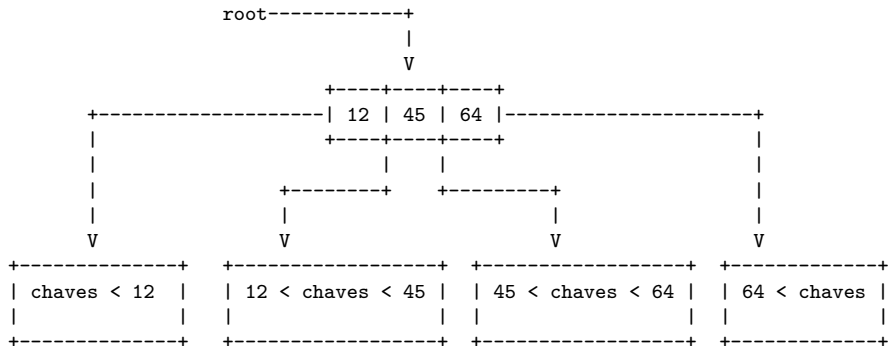
put(57)



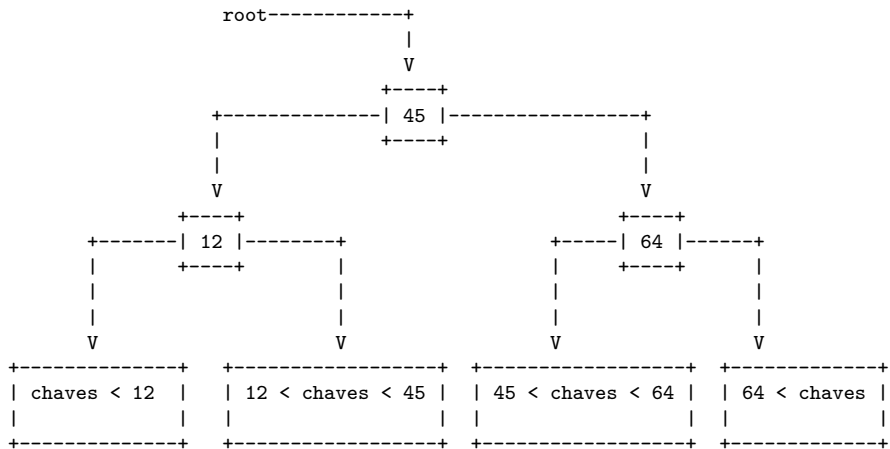
Espatifa o 4-nó.

Altura foi incrementada!

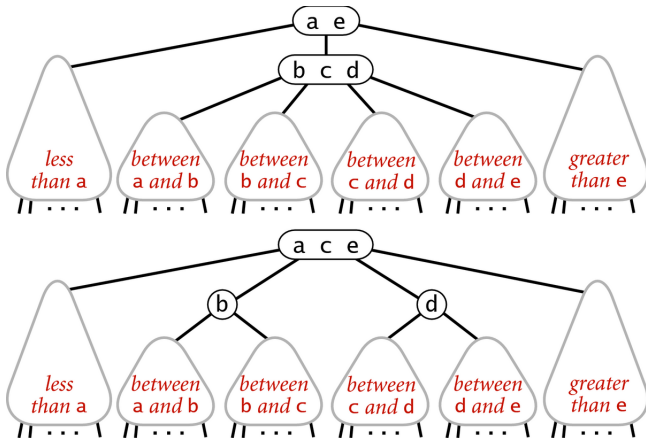
Tranformações preservam propriedades



Tranformações preservam propriedades



Tranformações preservam propriedades



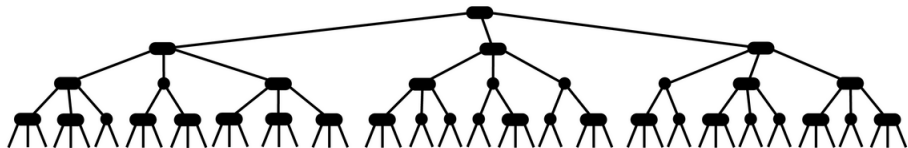
Splitting a 4-node is a local transformation that preserves order and perfect balance

Fonte: [algs4](#)

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.

Árvore 2-3 aleatória



Typical 2-3 tree built from random keys

Fonte: [algs4](#)

Implementação

Usaremos **BSTs** (**árvores binária de busca**) para simular **árvores 2-3**.

BSTs rubro-negras



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

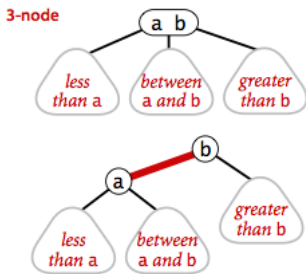
Referências: BSTs rubro-negras (PF); Balanced Search Trees (S&W); slides (S&W)

BSTs rubro-negras

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

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

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



BSTs rubro-negras

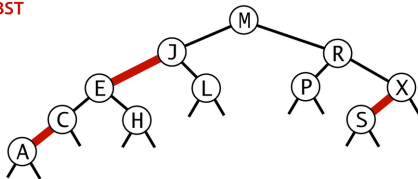
Uma **BST rubro-negra** é uma **BST** cujos links são negros e **rubros** e:

- ▶ **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.

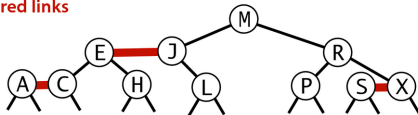
Se os **links rubros** forem desenhados horizontalmente e depois contraídos, teremos uma **árvore 2-3**

Anatomia de uma árvore **rubro**-negra

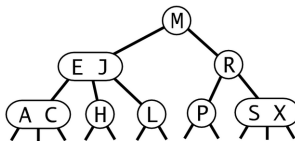
red-black BST



horizontal red links



2-3 tree

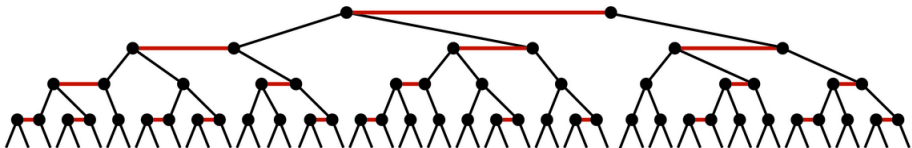


Fonte: [algs4](#)

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

Árvore 2-3 para rubro-negra

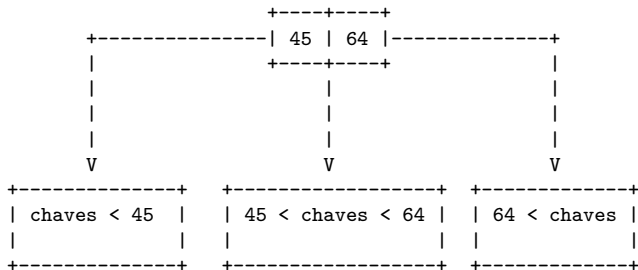
Se os **links rubros** forem desenhados horizontalmente e depois contraídos, teremos uma **árvore 2-3**:



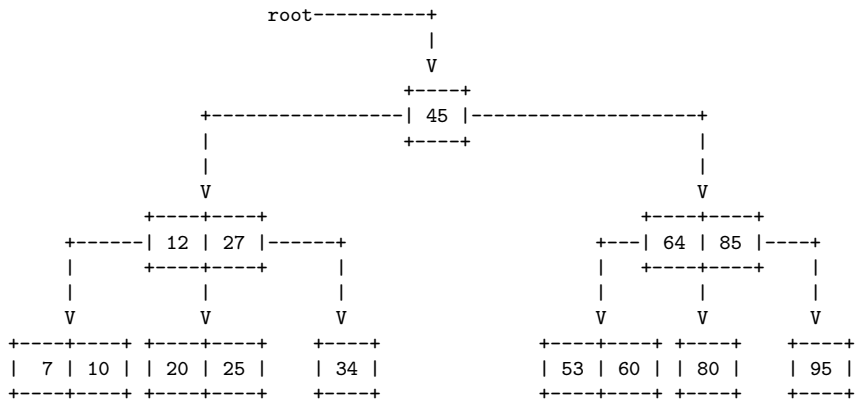
A red-black tree with horizontal red links is a 2-3 tree

Fonte: [algs4](#)

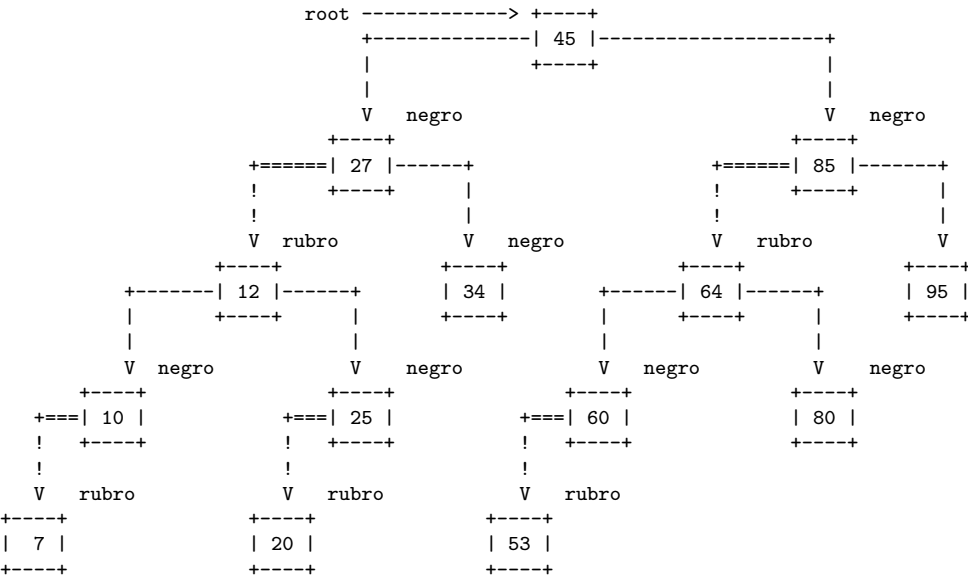
Árvore 2-3 para rubro-negra



Árvore 2-3



Árvore rubro-negra



Balanço e profundidade

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

Nota. No CLRS as árvores **rubro-negras** têm **nós rubros** e negros:

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

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

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

Nós de uma **BST rubro-negra**

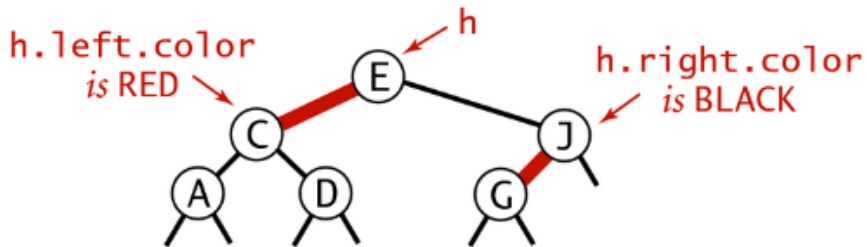
É inconveniente armazenar a **cor** de um link na estrutura de dados; é mais simples armazenar essa informação nos nós.

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

A raiz é considerada **negra**.

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

Nós de uma **BST rubro-negra**



Nós de uma BST rubro-negra

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

get(key)

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

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

get(key)

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

```
private Node get(Node x, Key key) {
    // Considera subárvore que tem raiz x
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        return get(x.left, key);
    else if (cmp > 0)
        return get(x.right, key);
    else return x;
}
```

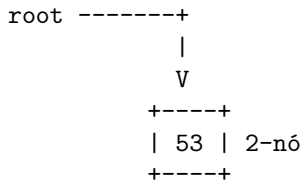
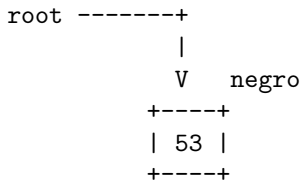

get(key) versão iterativa

Recebe uma chave `key` e retorna o valor `val` associado `key`; se `key` não está na `BST`, retorna `null`.

```
private Node get(Node x, Key key) {
    if (x == null) return null;
    while (x != null && !x.key.equals(key))
        int cmp = key.compareTo(x.key);
        if (cmp > 0)
            x = x.left;
        else
            x = x.right;
    if (x != null) return x.val;
    return null;
}
```

Inserção em um 2-nó

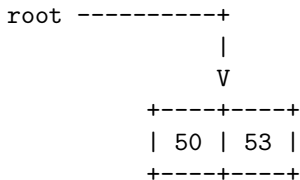
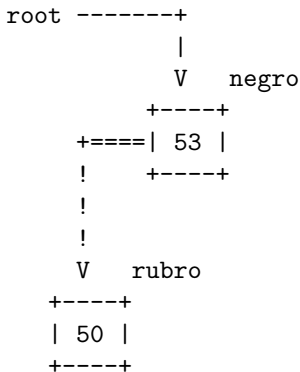
Árvore formada por apenas um 2-nó



Inserção em um 2-nó

Árvore formada por apenas um 2-nó

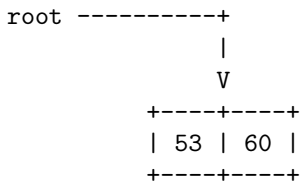
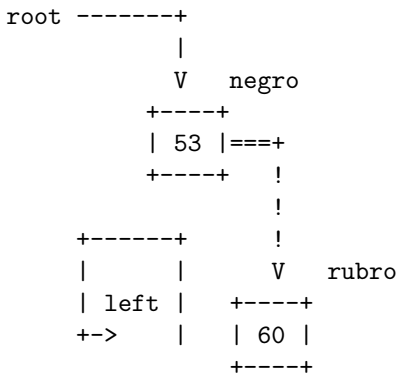
```
put(50)
```



Inserção em um 2-nó

Árvore formada por apenas um 2-nó

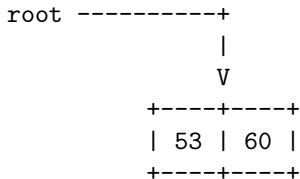
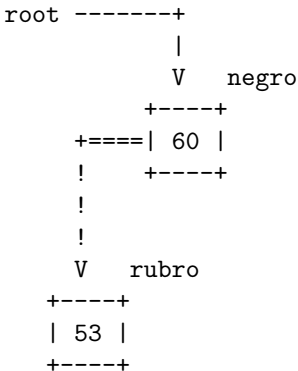
put(60)



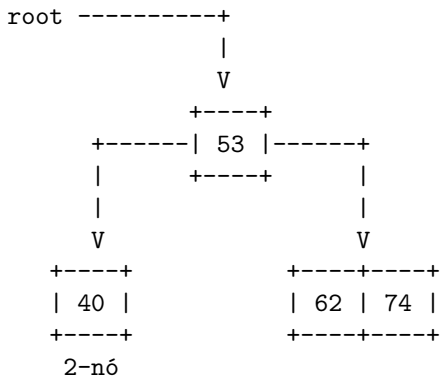
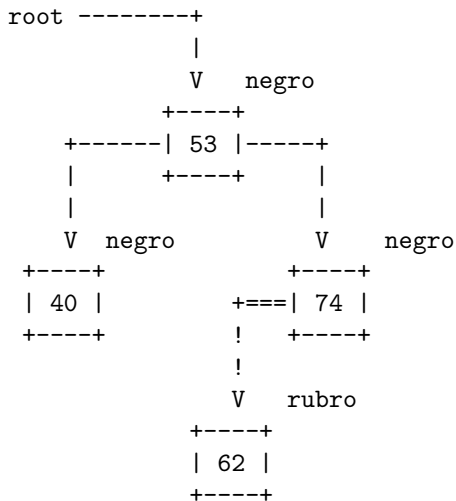
Inserção em um 2-nó

Árvore formada por apenas um 2-nó

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

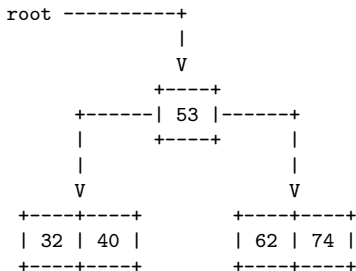
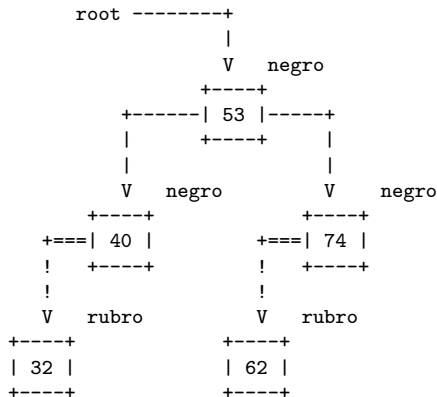


Inserção em um 2-nó qualquer



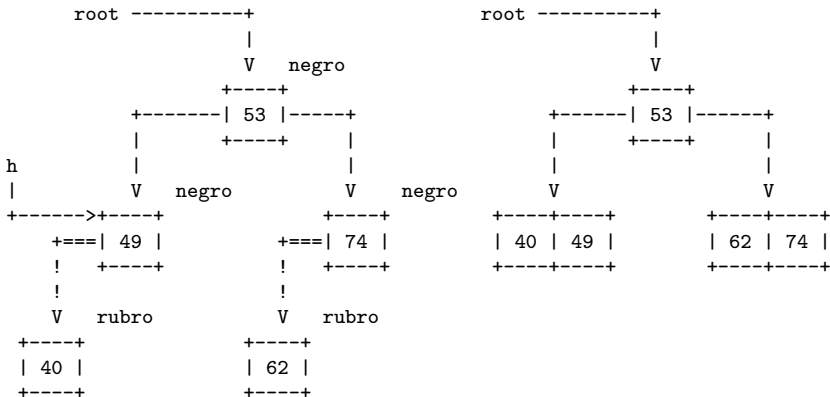
Inserção em um 2-nó qualquer

put(32)

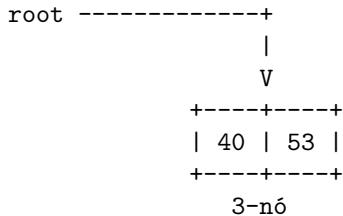
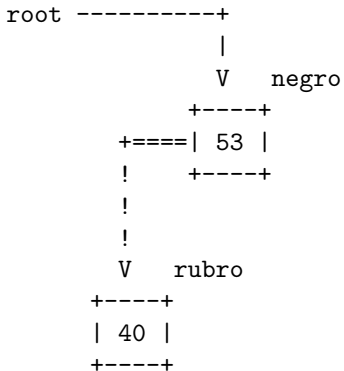


Inserção em um 2-nó qualquer

h = rotateLeft(h);

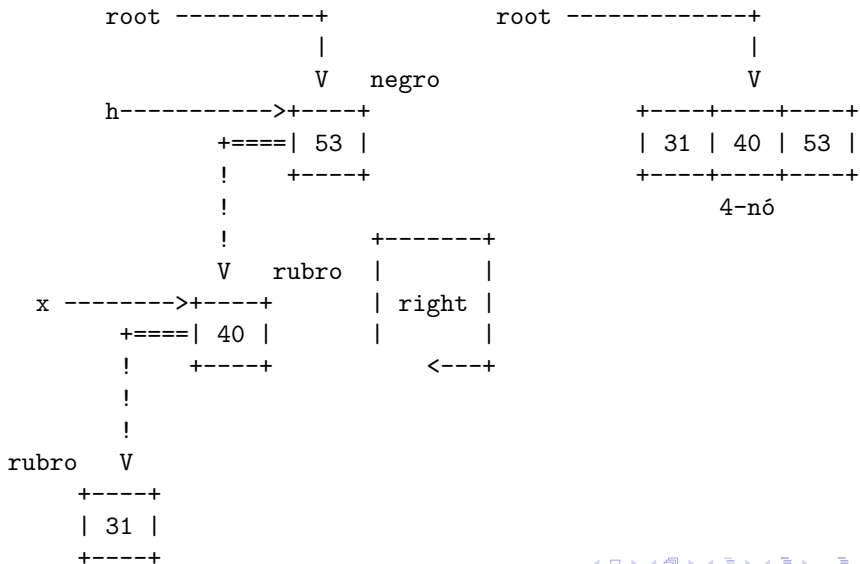


Inserção em um 3-nó



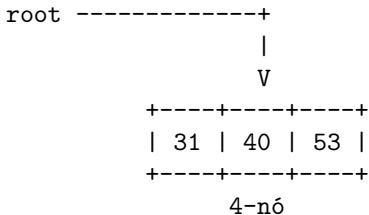
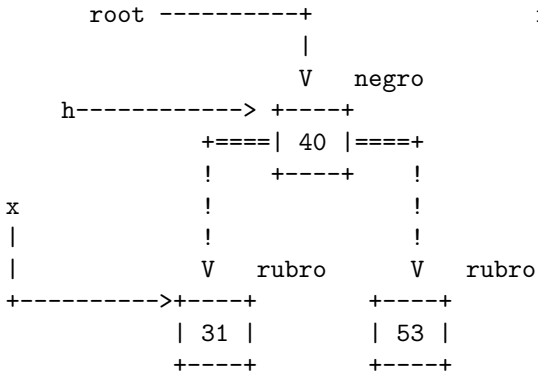
chave é inserida é menor do 3-nó

put(31)



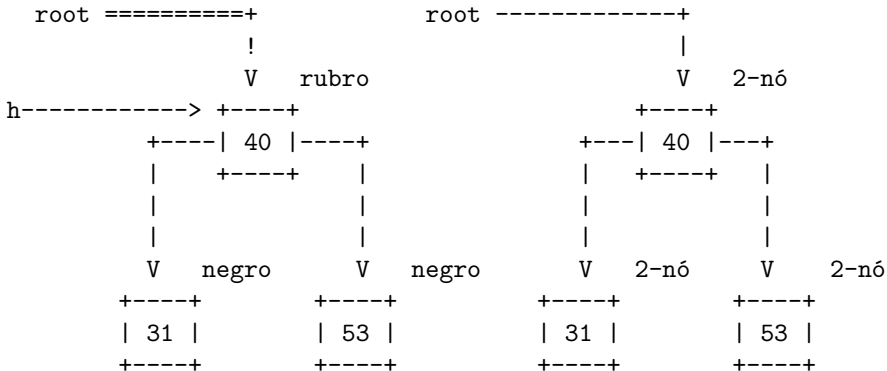
chave é inserida é menor do 3-nó

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



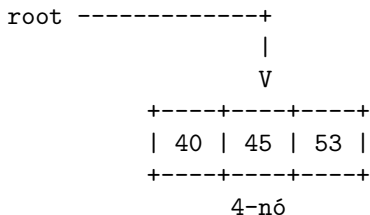
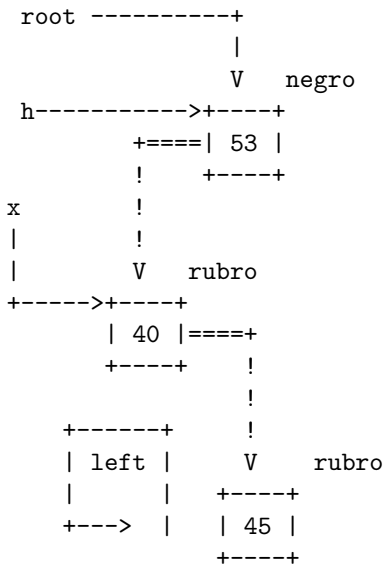
chave é inserida é menor do 3-nó

```
flipColors(h);
```



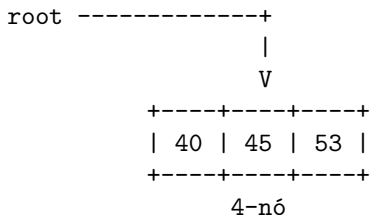
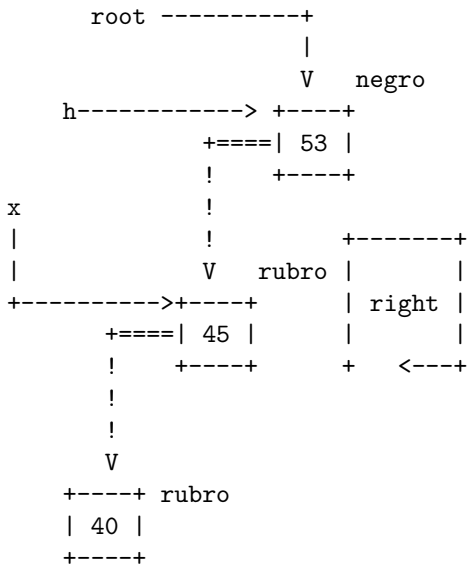
chave é inserida entre as chaves do 3-nó

put(45)



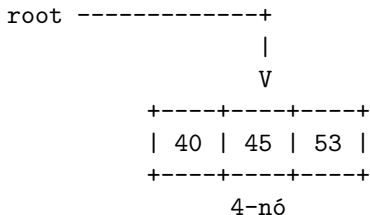
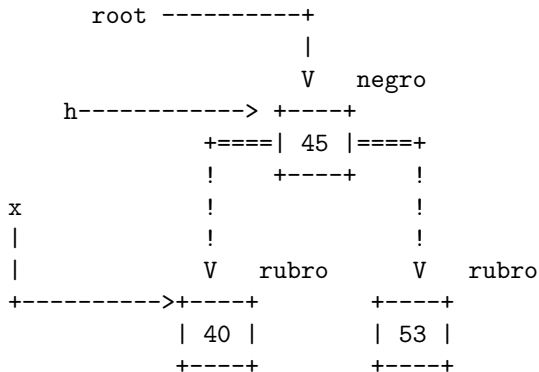
chave é inserida entre as chaves do 3-nó

x = rotateLeft(x);



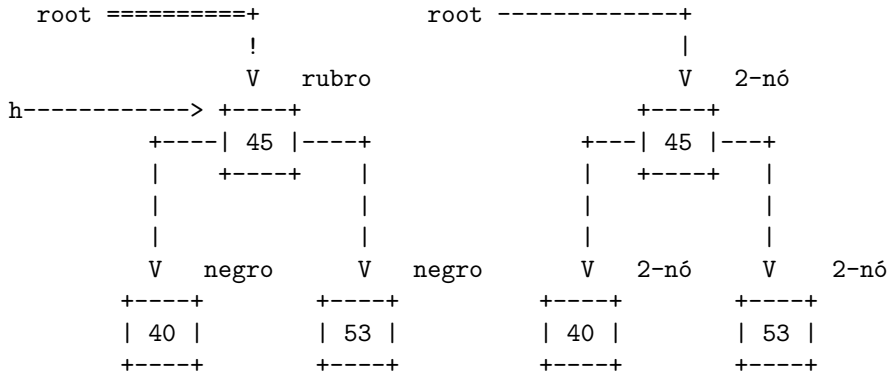
chave é inserida entre as chaves do 3-nó

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



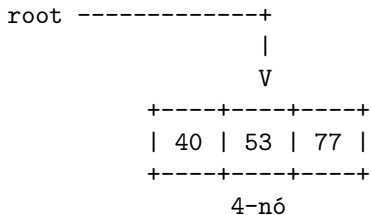
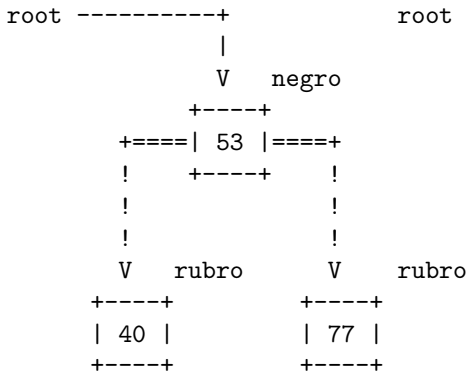
chave é inserida entre as chaves do 3-nó

flipColors(h); hmmm. raiz deve ser negra



chave inserida é maior que todas do 3-nó

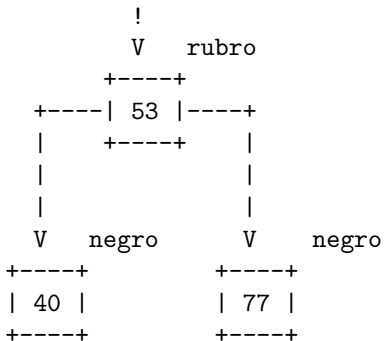
put(77)



chave inserida é maior que todas do 3-nó

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

root =====+



root -----+

