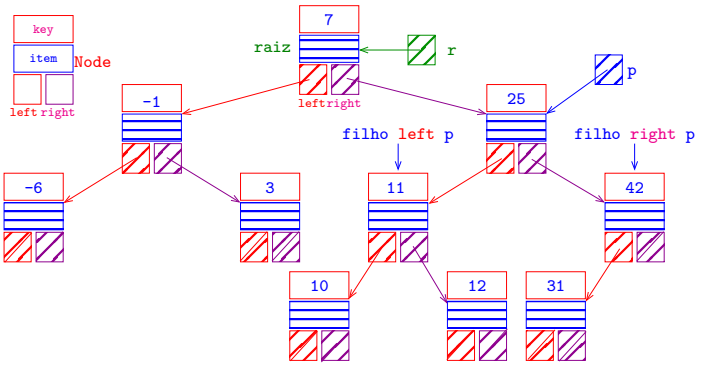
Fonte: ash.atozviews.com

Compacto dos melhores momentos

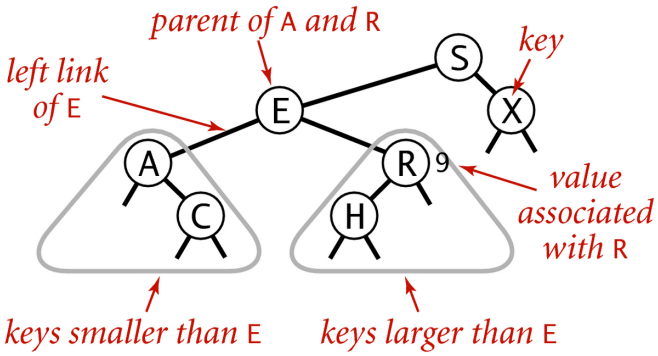
AULA 10

Árvore binárias de busca



in-ordem (e-r-d): -6 -1 3 7 10 11 12 25 31 42

Anatomia de uma árvore binária de busca



Anatomy of a binary search tree

Fonte: algs4

Consumo de tempo

O consumo de tempo das funções `get()`, `put()` e `delete()` é, no pior caso, proporcional à altura da árvore.

Conclusão: interessa trabalhar com árvores balanceadas: árvores AVL, árvores rubro-negras, árvores . . .

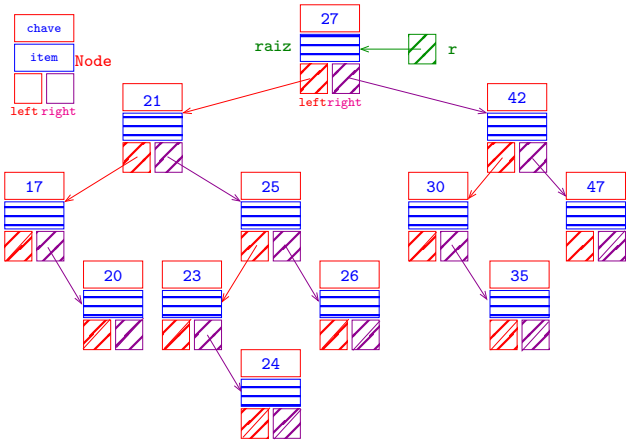
Mais experimentos

Consumo de tempo para se criar um ST em que a chaves são as palavras em `les_miserables.txt` e os valores o número de ocorrências.

estrutura	ST	tempo
vetor	não-ordenada	59.5
vetor MTF	não-ordenada	7.6
vetor	ordenada	1.5
lista ligada	não-ordenada	147.1
lista ligada MTF	não-ordenada	15.3
lista ligada	ordenada	115.2
skiplist	ordenada	1.1
arvore binária de busca ♡	ordenada	0.7

Tempos em segundos obtidos com `StopWatch`.

Desempenho esperado



Desempenho esperado

Na BST acima:

- a busca por 27 requer 1+0 comparações
- a busca por 21 requer 1+1 comparações
- a busca por 42 requer 1+1 comparações
- a busca por 17 requer 1+2 comparações
- a busca por 25 requer 1+2 comparações
- a busca por 30 requer 1+2 comparações
- a busca por 47 requer 1+2 comparações
- a busca por 20 requer 1+3 comparações
- a busca por 23 requer 1+3 comparações
- a busca por 26 requer 1+3 comparações
- a busca por 35 requer 1+3 comparações
- a busca por 24 requer 1+4 comparações

Desempenho esperado

Assim, o número médio de comparações necessários para uma busca com sucesso na BST acima é

$$(1+2+2+3+3+3+3+4+4+4+4+5)/12 \approx 3.17$$

O **comprimento interno** (= *internal path length*) de uma BT é a soma das profundidades dos seus nós.

O comprimento interno da árvore mostrada anteriormente é

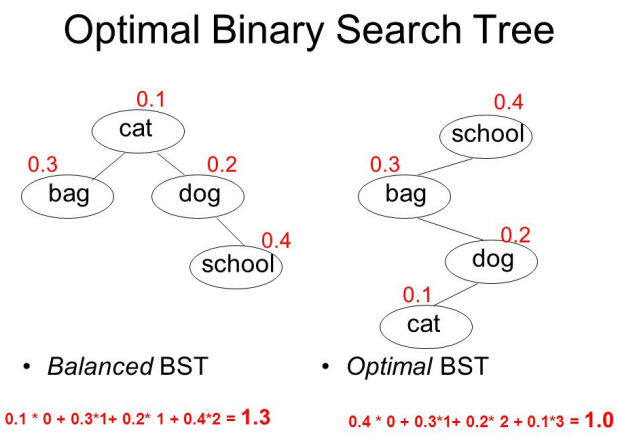
$$0+1+1+2+2+2+2+3+3+3+3+4 = 26$$

Consumo de tempo

O número esperado de nós visitados durante uma busca em uma BST aleatória não passa de $2 \lg n$

AULA 11

Árvores binárias de busca ótima



Árvores binárias de busca ótima

Devemos realizar um série de operação `get()` em uma BST com chaves

$$\texttt{key}[0] < \texttt{key}[1] < \texttt{key}[2] < \ldots < \texttt{key}[n\text{-}1].$$

Busca bem-sucedida: suponha que `p[i]` é a probabilidade de `key[i]` ser argumento de `get()`.

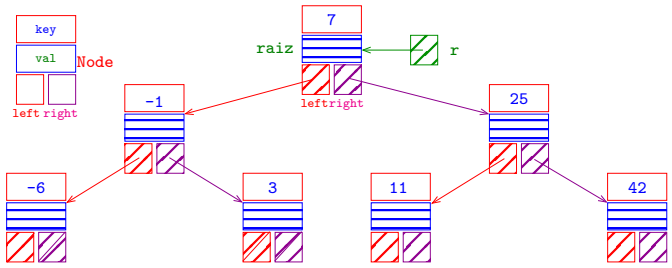
Problema. construir uma BST que minimize o número esperado de comparações:

$$\begin{aligned} &\texttt{p}[0](\texttt{prf}[0]+1) + \texttt{p}[1](\texttt{prf}[1]+1) \\ &\quad + \cdots \\ &\quad + \texttt{p}[n-1](\texttt{prf}[n-1]+1) \end{aligned}$$

onde `prf[i]` é a profundidade da chave `key[i]`.

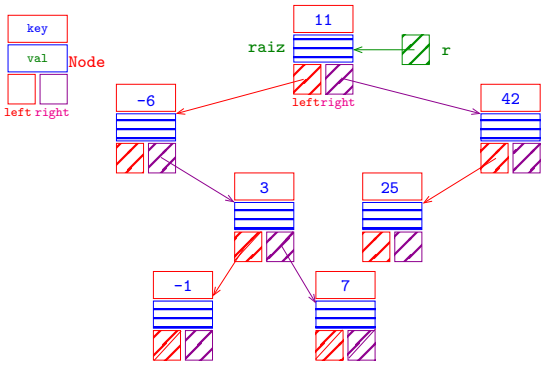
Exemplo: BST balanceada

key	-6	-1	3	7	11	25	42
p	0.22	0.18	0.20	0.05	0.25	0.02	0.08



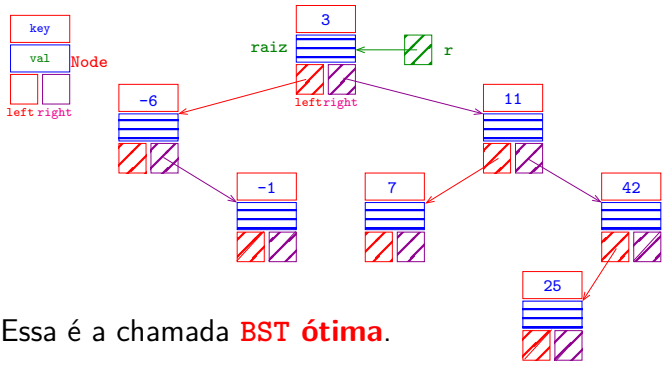
$0.05+2(0.18+0.02)+3(0.22+0.20+0.25+0.08) = 2.7$ comparações

Exemplo: BST gulosa



$0.25+2(0.22+0.08)+3(0.2+0.02)+4(0.18+0.05) = 2.43$ comparações

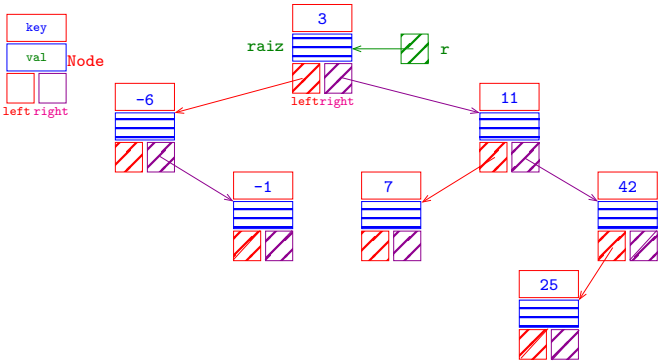
Exemplo: BST ótima



Essa é a chamada **BST ótima**.

$0.2+2(0.22+0.25)+3(0.18+0.05+0.08)+4(0.02) = 2.15$ comparações

Propriedade: subestrutura ótima



Toda subárvore de uma árvore ótima é **ótima**.

Solução indutiva

Seja `C[i,j]` o número esperado de comparações de uma BST ótima com as chaves

$$\texttt{key[i]} < \texttt{key[i+1]} < \cdots < \texttt{key[j-1]}$$

Suponha que a BST ótima das chaves

$$\texttt{key[i]} < \texttt{key[i+1]} < \cdots < \texttt{key[j-1]}$$

tem na sua raiz `key[r]`.

Nesse caso,

$$\texttt{C[i,j]} = \texttt{p[r]} + \texttt{C[i,r-1]} + \texttt{C[r+1,j]} + \sum_{\texttt{k=i}}^{\texttt{r-1}} \texttt{p[k]} + \sum_{\texttt{k=r+1}}^{\texttt{j-1}} \texttt{p[k]}$$

Solução indutiva

Na expressão para `C[i,j]` temos que

- `key[r]` contribui com `p[r]` pois `prf[r]` = 0;
- `C[i,r-1]` corresponde as comparações da subárvore esquerda;
- `C[r+1,j]` corresponde as comparações da subárvore direita;
- fazer `key[r]` raiz dessas subárvores faz as suas profundidades aumentarem de 1. Isso corresponde aos somatórios.

Reescrevendo obtemos

$$\texttt{C[i,j]} = \texttt{C[i,r-1]} + \texttt{C[r+1,j]} + \sum_{\texttt{k=i}}^{\texttt{j-1}} \texttt{p[k]}$$

Recorrência

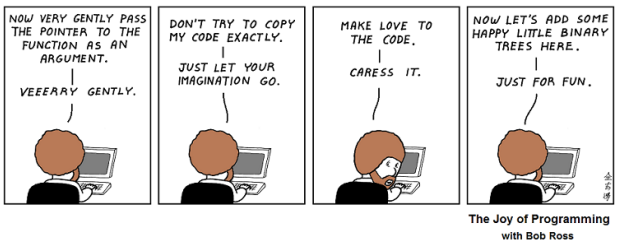
Finalmente, como não conhecemos `key[r]`, testamos todos os candidatos

$$\begin{aligned} C[i,i] &= 0 \\ C[i,i+1] &= p[i] \\ C[i,j] &= \min_{i \le r < j} \{C[i,r-1] + C[r+1,j] + \sum_{k=i}^{j-1} p[k]\} \end{aligned}$$

Consumo de tempo

O consumo de tempo para construir uma BST ótima é $O(n^3)$ onde n é o número de chaves na árvore.

Árvores 2-3



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.

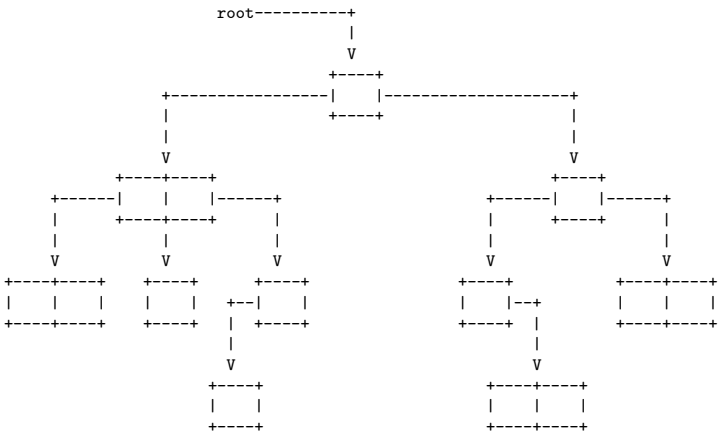
Árvore 2-3

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.

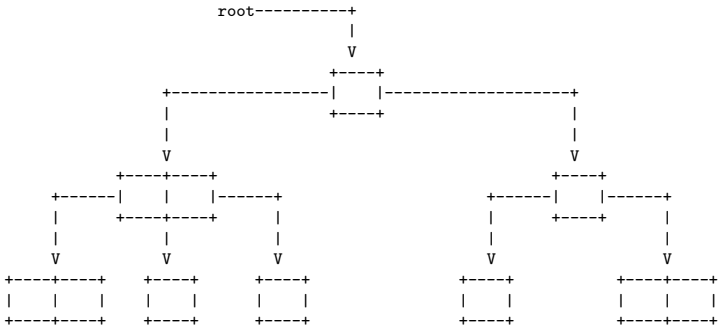
Árvores 2-3 têm esse nome porque cada nó tem **2 ou 3 links**.

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.

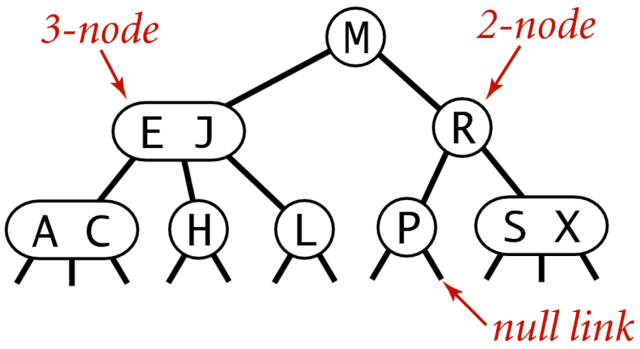
Consequência. Toda árvore 2-3 com `n` nós tem altura não superior a $\lg(n+1)-1$ e não inferior a $\log_3(n+1)-1$.

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.

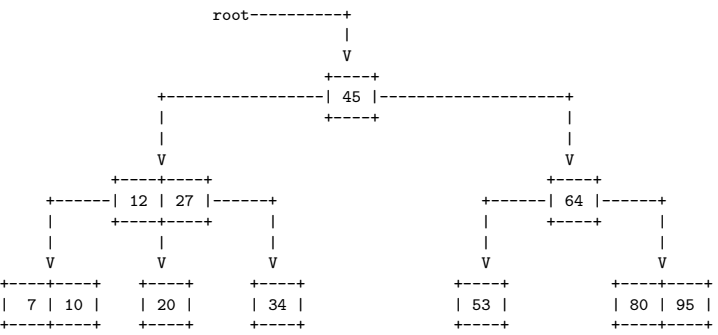
Anatomia de uma árvore 2-3 de busca



Anatomy of a 2-3 search tree

Fonte: `algs4`

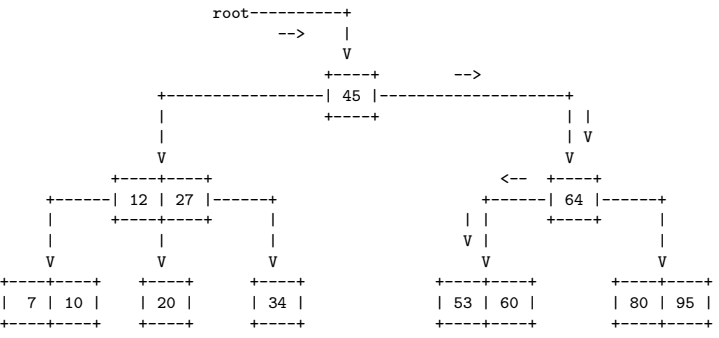
Exemplo de árvore 2-3 de busca



Missão

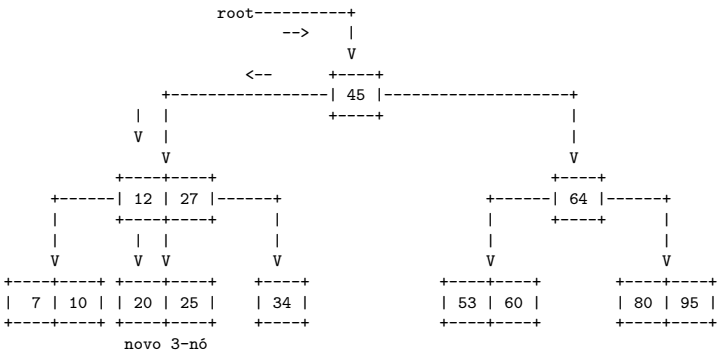
Missão. Manter uma árvore 2-3 de busca sujeita a operações de atualização como `put`(), `deleteMin`(), `delete`(), ...

put(60)



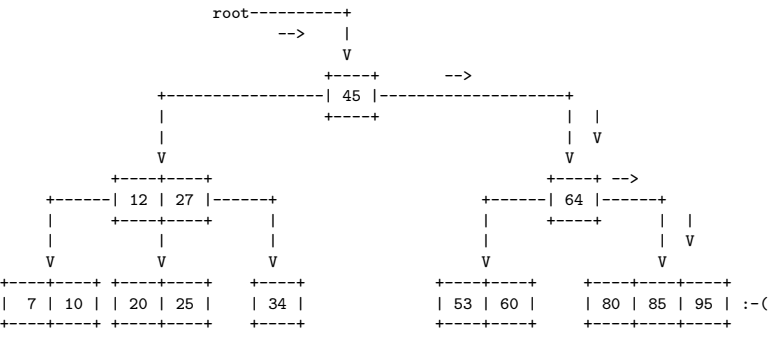
Inserção em um nó simples (**não** estraga estrutura):
Procura 60 e transforma um 2-nó em 3-nó.

put(25)



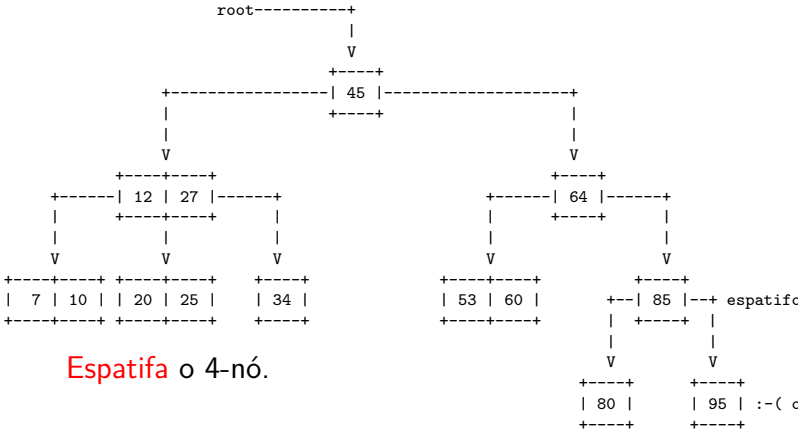
Procura 25 e transforma um 2-nó em 3-nó.

put(85)



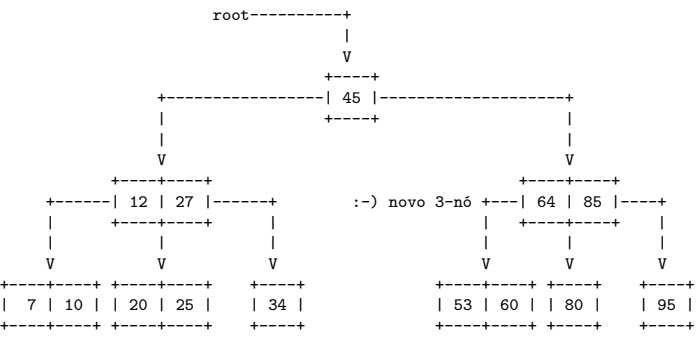
Inserção em um nó duplo (**estraga** estrutura):
procura 85 e transforma um 3-nó em 4-nó.

put(85)



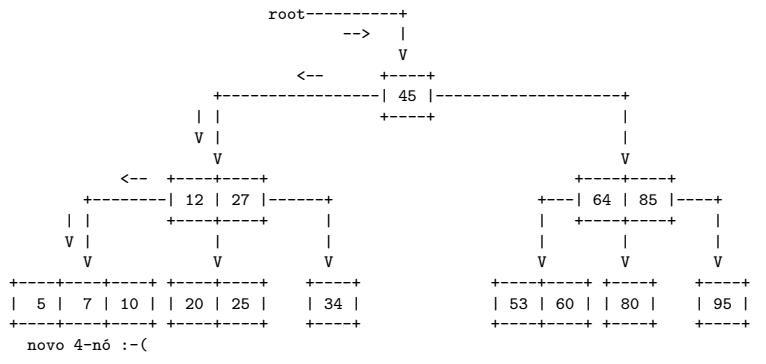
Espatifa o 4-nó.

put(85)



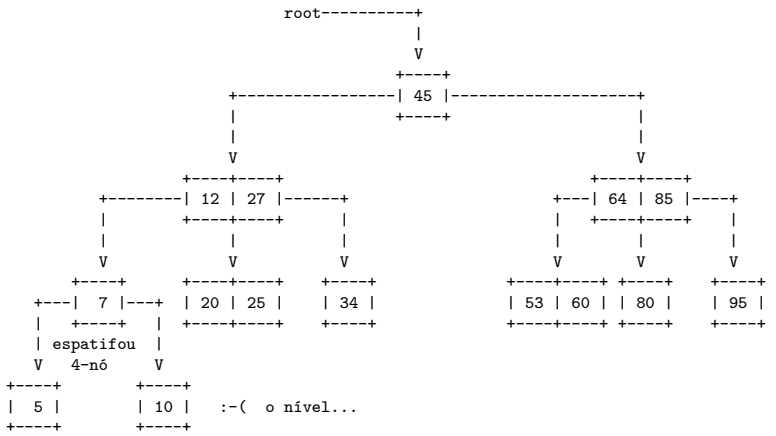
Transforma o 2-nó pai em 3-nó.

put(5)



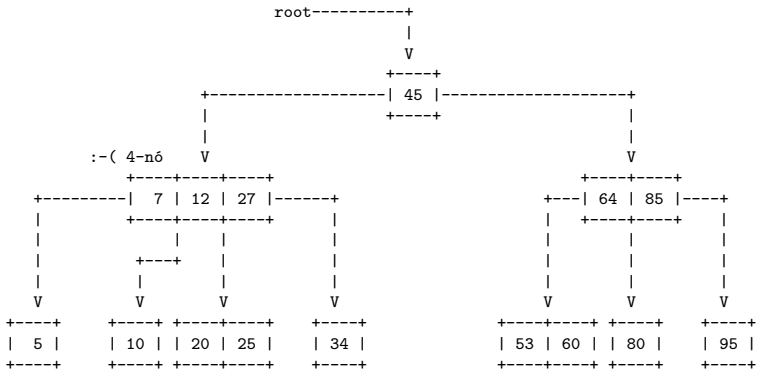
Procura 5 e transforma um 3-nó em 4-nó.

put(5)



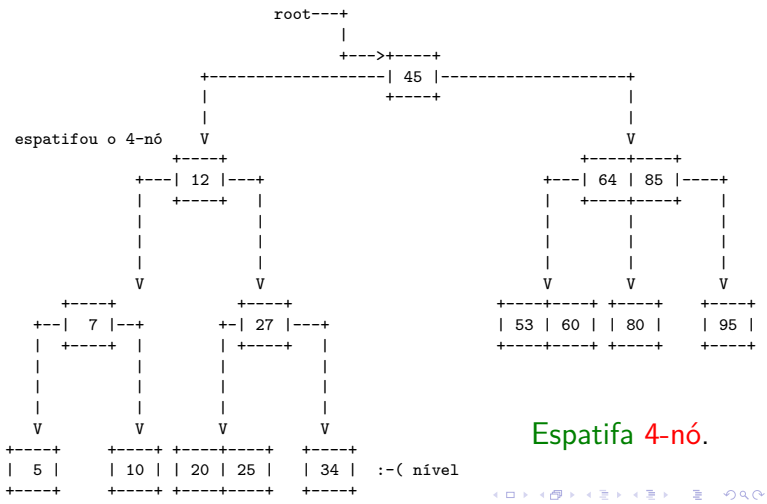
Espatifa o 4-nó.

put(5)



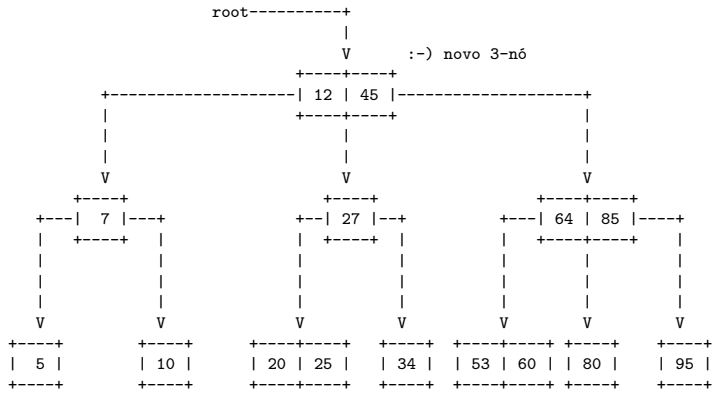
Transforma o 3-nó pai em 4-nó.

put(5)



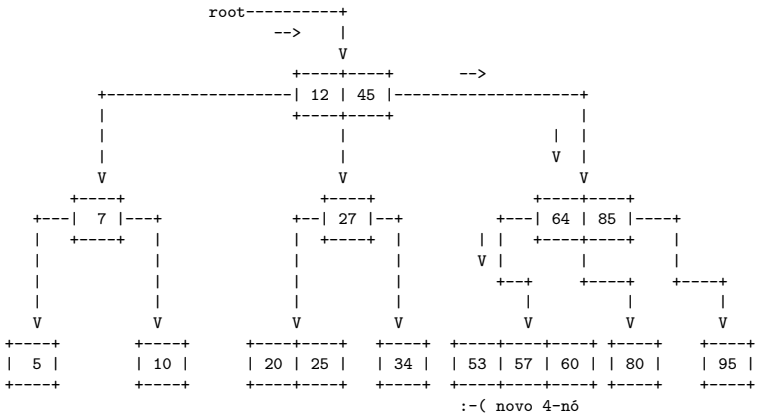
Espatifa 4-nó.

put(5)



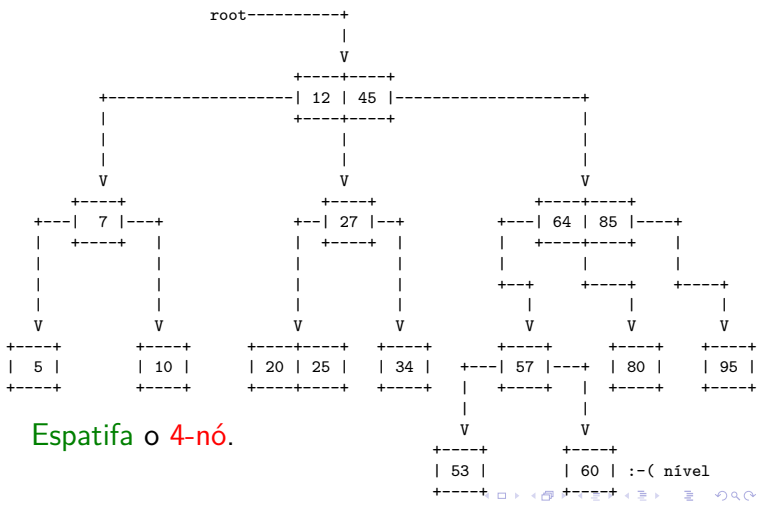
Transforma o 2-nó pai em 3-nó.

put(57)



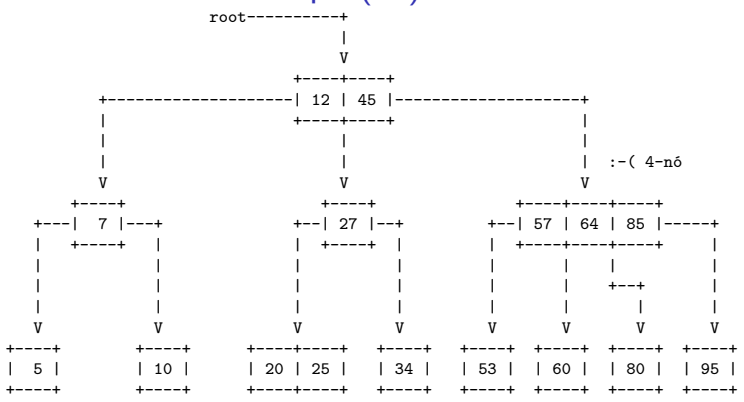
Procura 57 e transforma o 3-nó em 4-nó.

put(57)



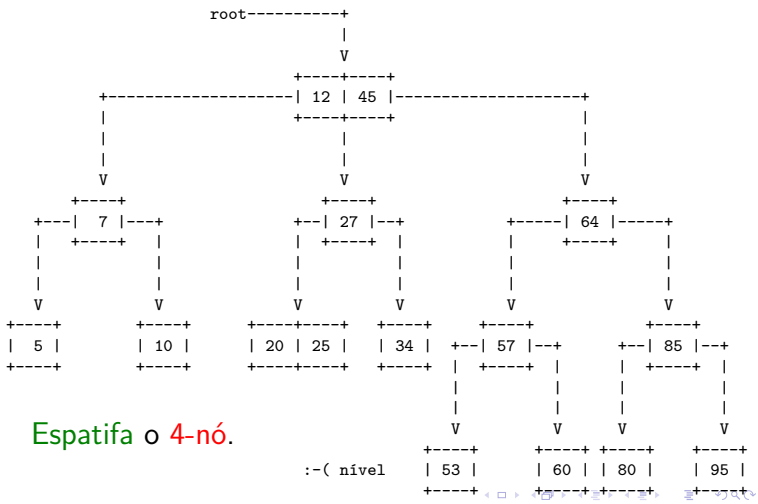
Espatifa o 4-nó.

put(57)



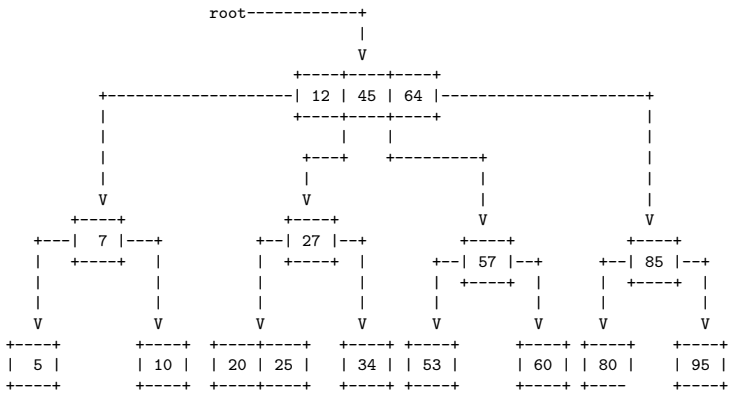
Transforma o 3-nó pai em 4-nó.

put(57)



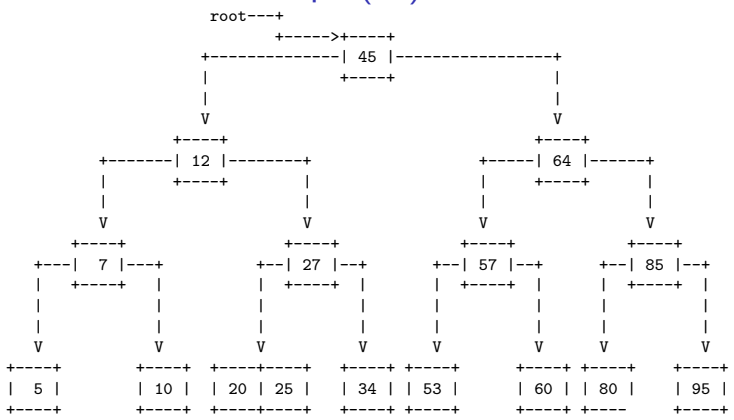
Espatifa o 4-nó.

put(57)



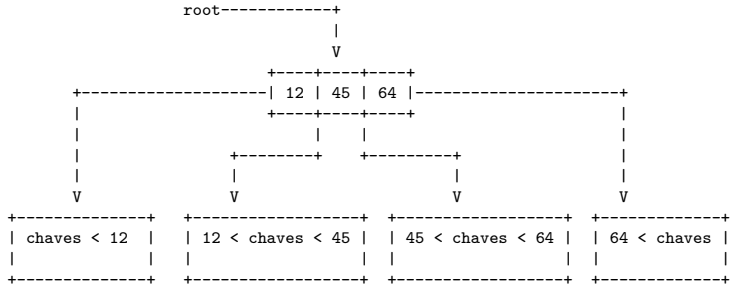
Tranforma o 3-nó pai em um 4-nó.

put(57)

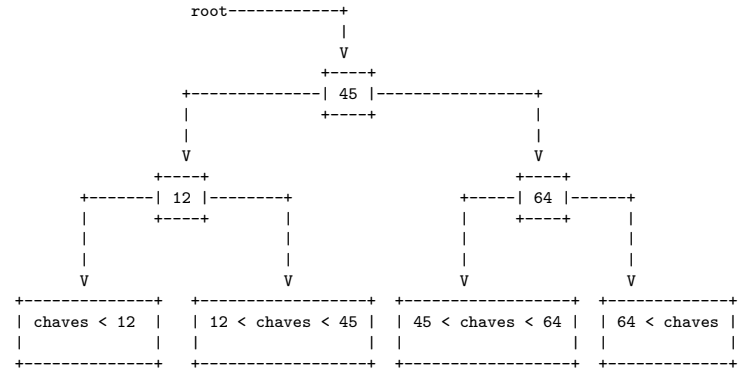


Espatifa o 4-nó. **Altura foi incrementada!**

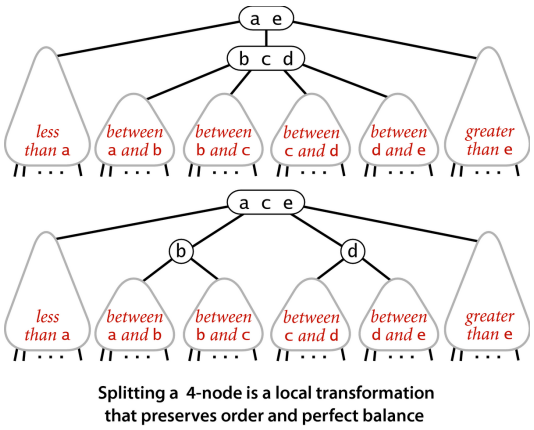
Tranformações preservam propriedades



Tranformações preservam propriedades



Tranformações preservam propriedades



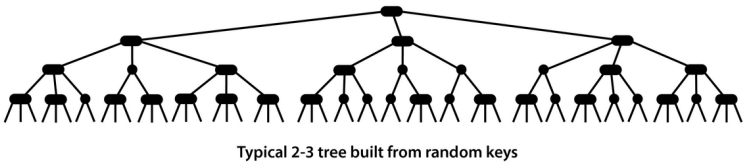
Splitting a 4-node is a local transformation that preserves order and perfect balance

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 simultar árvores 2-3.

BSTs rubro-negras

Fonte: http://scottlobdell.me/

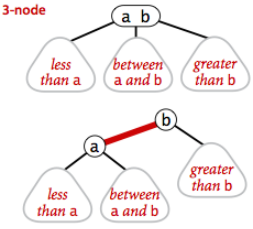
Referências: BSTs rubro-negras (PF); Balanced Search Trees (S&W); slides (S&W) Hashing Functions (S&W)

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.



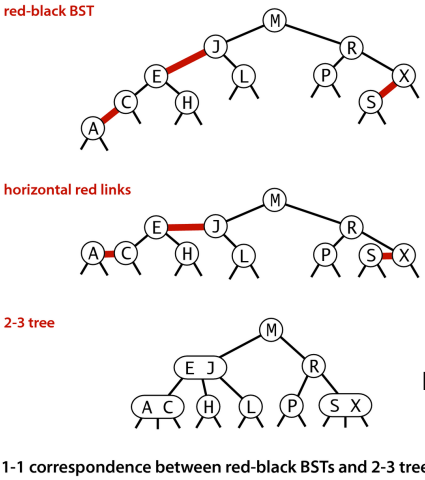
BSTs rubro-negras

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

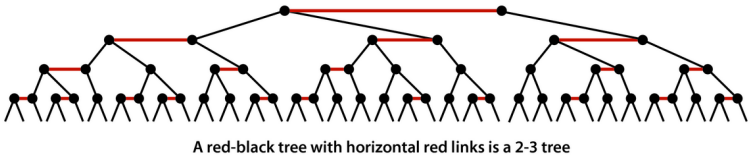


Fonte: algs4

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

Árvore 2-3 para rubro-negra

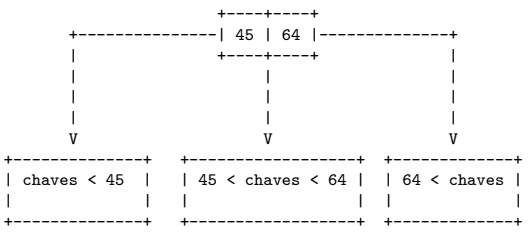
Se os links rubros forem desenhados horizontalmente e depois contraídos, teremos uma árvore 2-3:



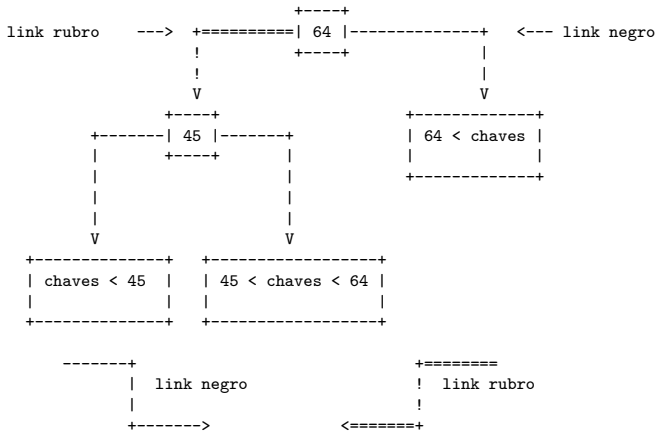
A red-black tree with horizontal red links is a 2-3 tree

Fonte: algs4

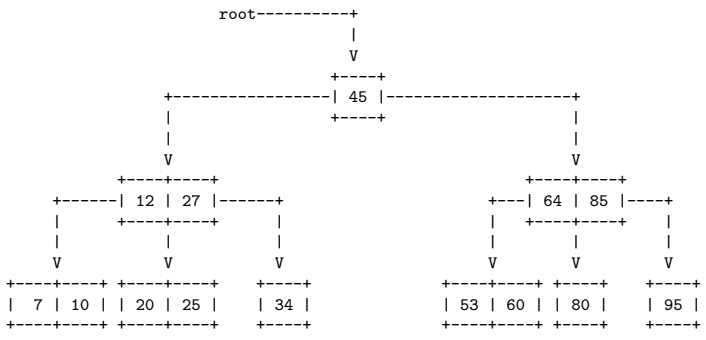
Árvore 2-3 para rubro-negra



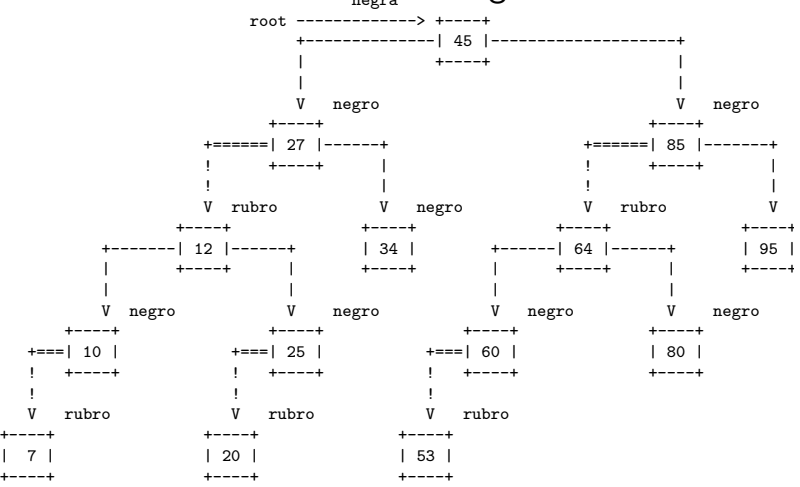
Árvore 2-3 para rubro-negra



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

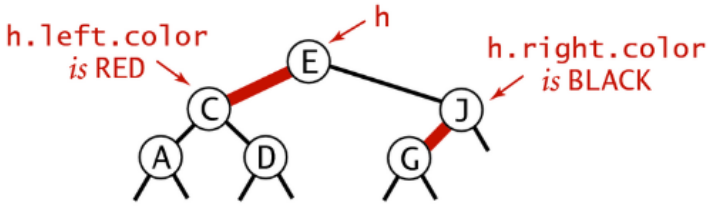
É inconveniente armazenar a cor de um link na estrutura de dados; é mais simples armazenar essa informação nos nós.

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;
}
```

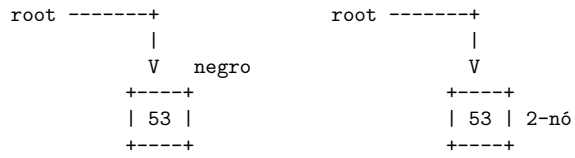
get(key) versão iterativa

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 > k)
          x = x.left;
       else
          x = x.right;
   if (x != null) return x.val;
   return null;
}
```

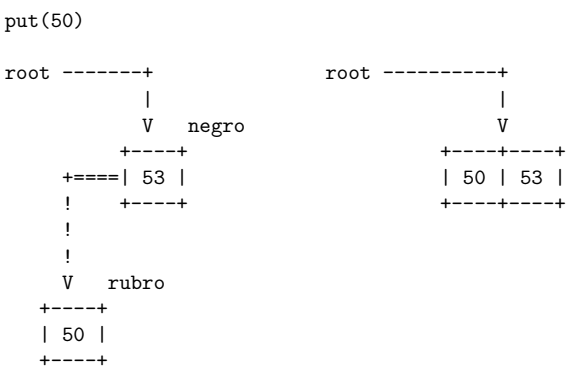
Inserção em um 2-nó

Árvore formada por apenas um 2-nó



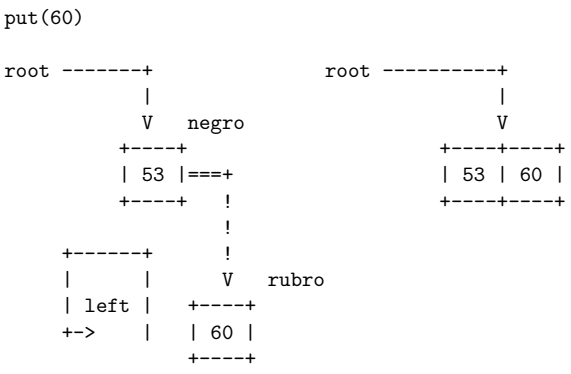
Inserção em um 2-nó

Árvore formada por apenas um 2-nó



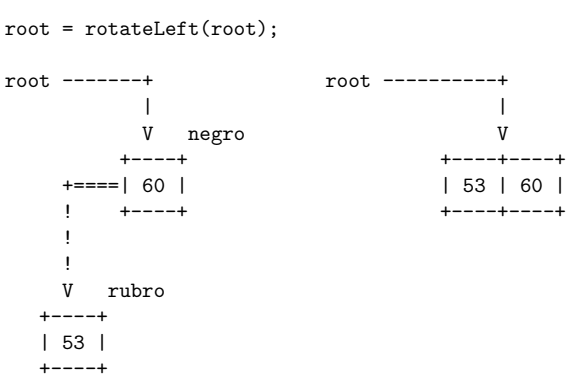
Inserção em um 2-nó

Árvore formada por apenas um 2-nó

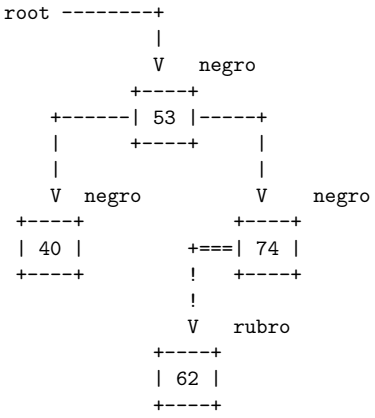


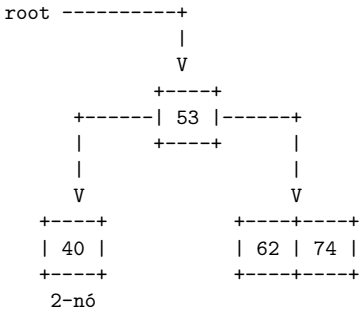
Inserção em um 2-nó

Árvore formada por apenas um 2-nó

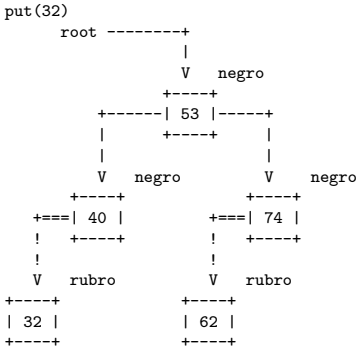


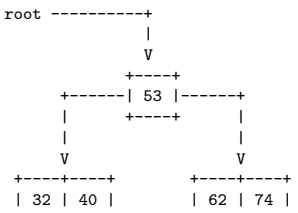
Inserção em um 2-nó qualquer



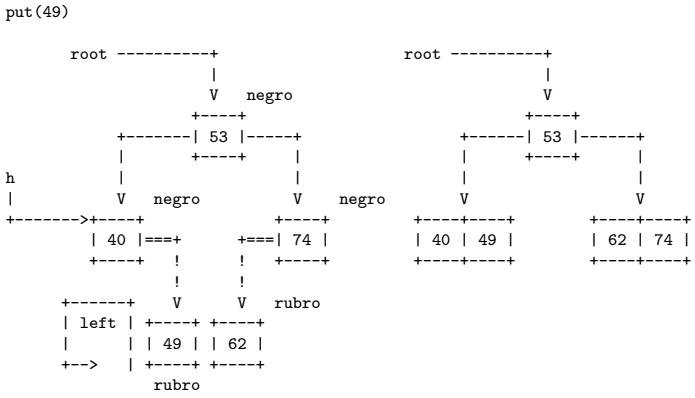


Inserção em um 2-nó qualquer

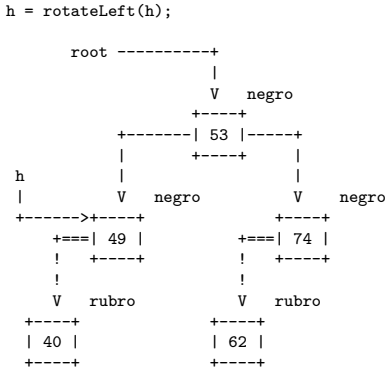


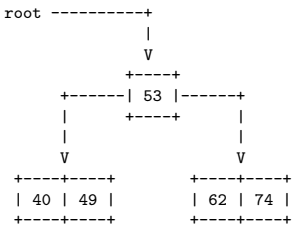


Inserção em um 2-nó qualquer

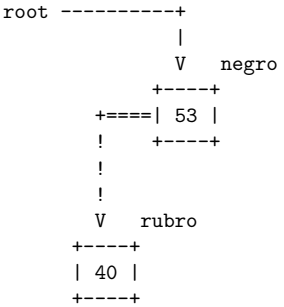


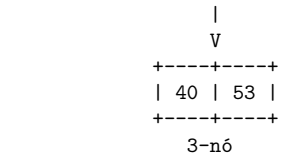
Inserção em um 2-nó qualquer



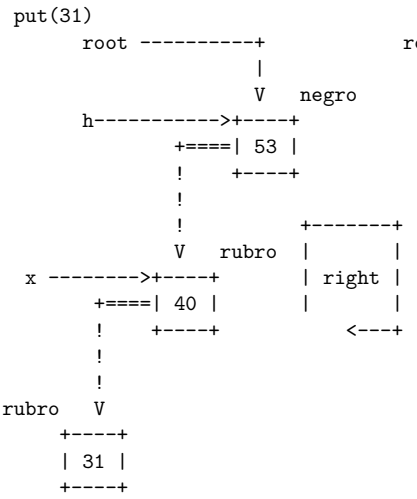


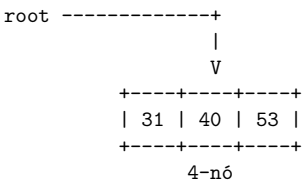
Inserção em um 3-nó





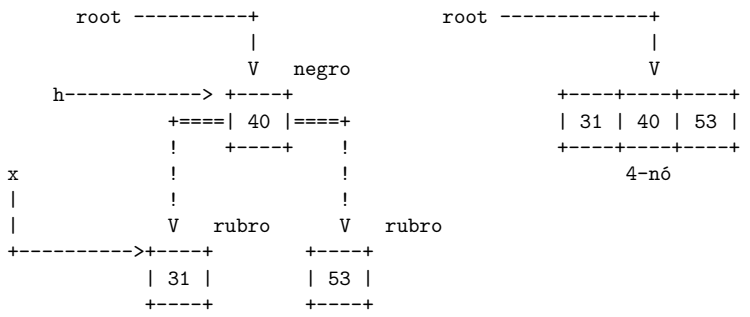
chave é inserida é menor do 3-nó





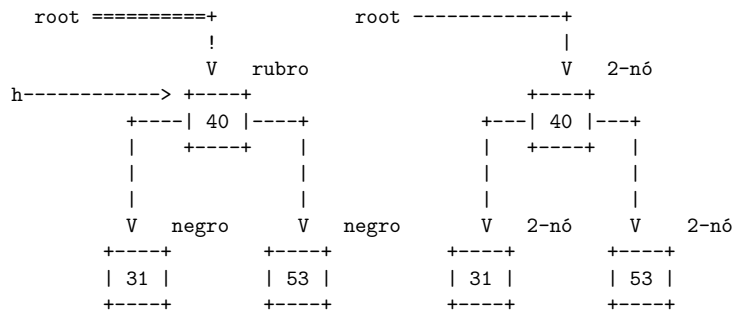
chave é inserida é menor do 3-nó

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



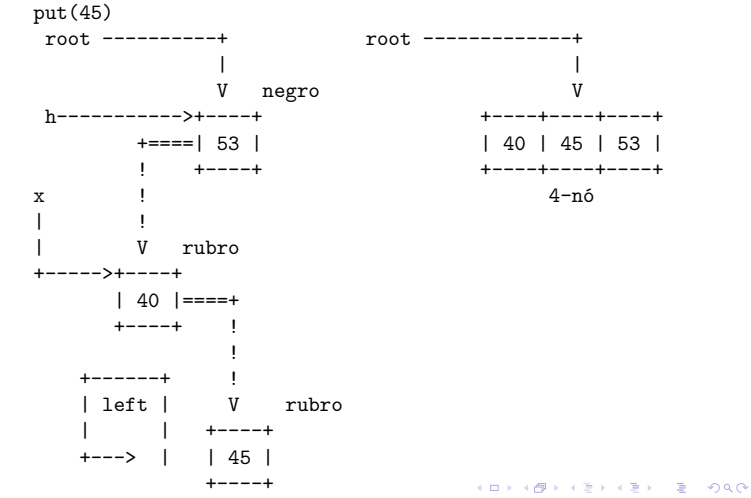
chave é inserida é menor do 3-nó

```
flipColors(h);
```



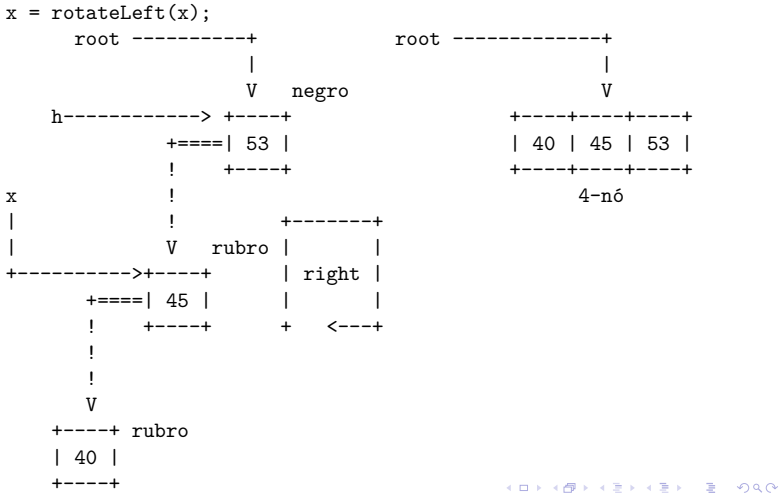
chave é inserida entre as chaves do 3-nó

```
put(45)
```



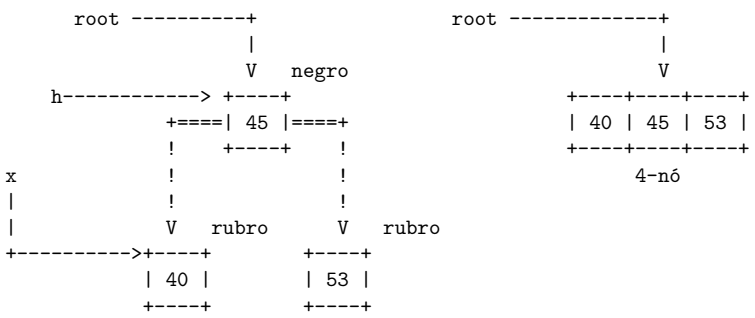
chave é inserida entre as chaves do 3-nó

```
x = rotateLeft(x);
```



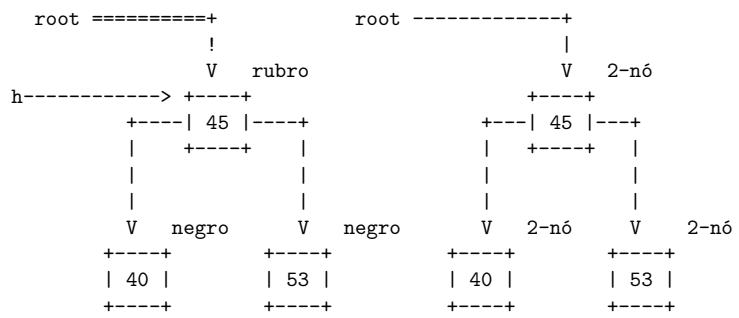
chave é inserida entre as chaves do 3-nó

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



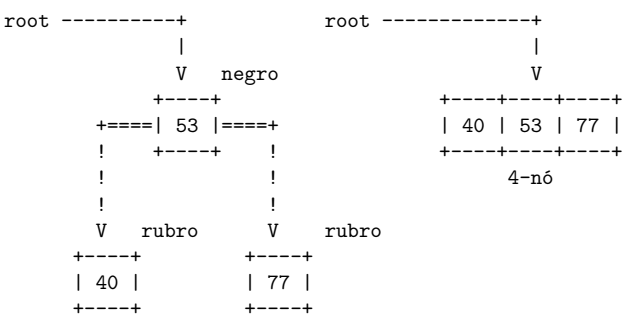
chave é inserida entre as chaves do 3-nó

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



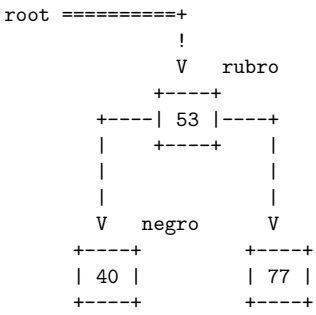
chave inserida é maior que todas do 3-nó

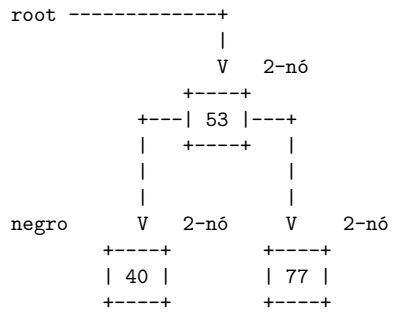
```
put(77)
```



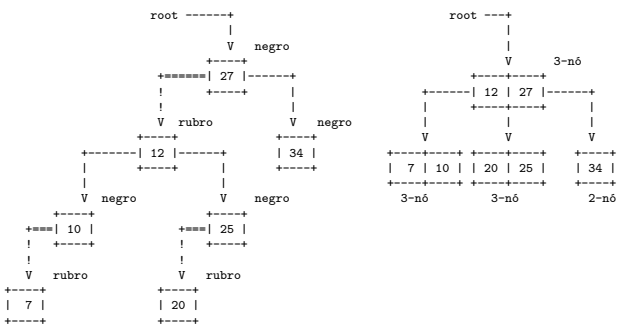
chave inserida é maior que todas do 3-nó

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



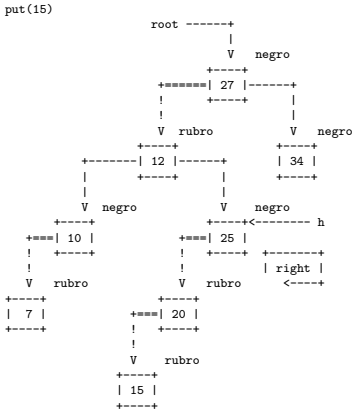


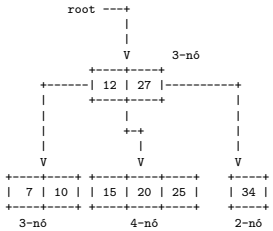
chave é inserida em um 3-nó qualquer



chave é inserida em um 3-nó qualquer

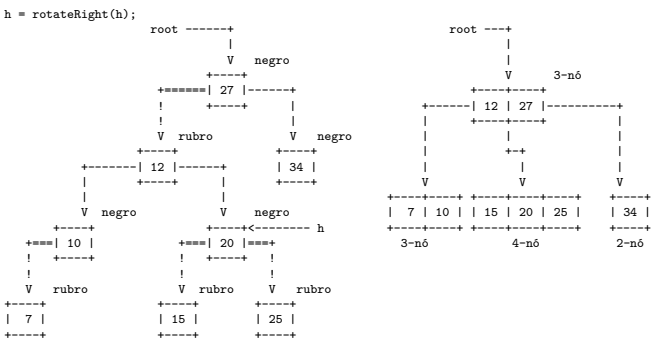
```
put(15)
```





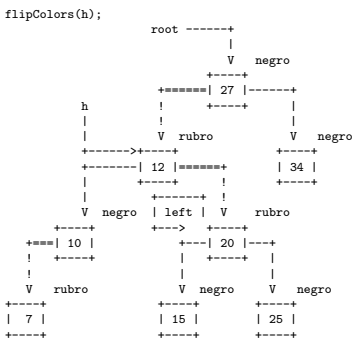
chave é inserida em um 3-nó qualquer

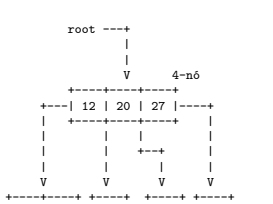
```
h = rotateRight(h);
```



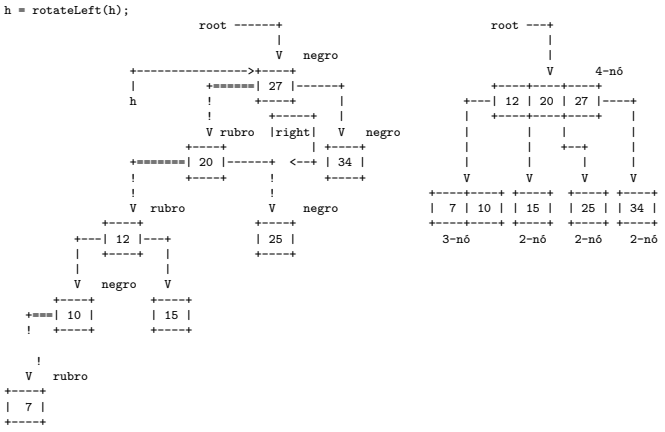
chave é inserida em um 3-nó qualquer

```
flipColors(h);
```

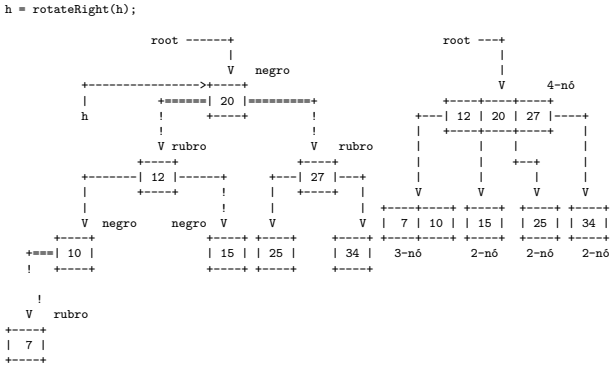




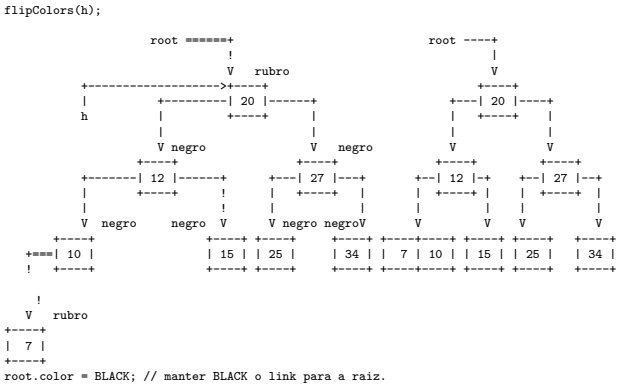
chave é inserida em um 3-nó qualquer



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.

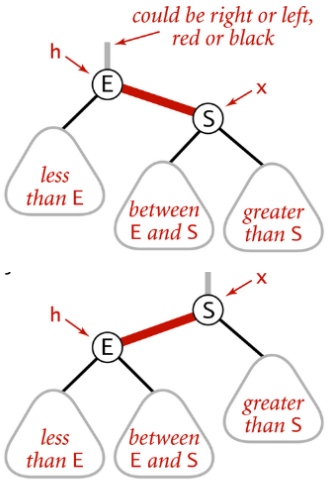
Durante uma operação de inserção, podemos ter, temporariamente, um link rubro inclinado para a direita ou dois links rubros incidindo no mesmo nó.

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



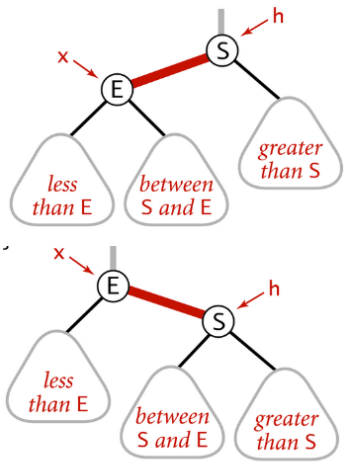
Fonte: algs4

Left rotate (right 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



Right rotate (left link of h)

Fonte: algs4

Rotação direita

```
private Node rotateRight(Node h) {
  Node x = h.left;
  h.left = x.right;
  x.right = h;
  x.color = h.color;
  h.color = RED;
  x.n = h.n;
  h.n = 1 + size(h.left) + size(h.right);
  return x;
}
```

Flipping colors

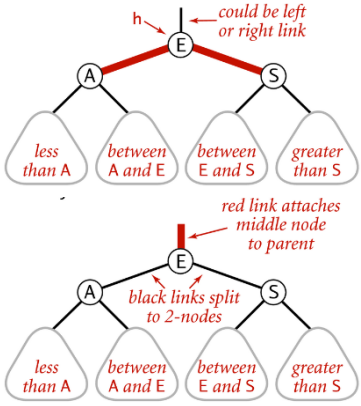
As operações de rotação são locais.

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

Mas a operação pode ter criado um link rubro inclinado para a lado errado ou dois links rubros seguidos. Isso deverá ser corrigido.

Na árvore 2-3 a operação de **flipping colors** corresponderá a espatifar um 4-nó e subir a chave do meio para o nó pai.

Flipping colors



Flipping colors to split a 4-node

Fonte: algs4

Flipping Colors

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

RedBlackBST

```
public class RedBlackBST<Keyextends
          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);
   return h;
}
```

RedBlackBST

```
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;
}
```