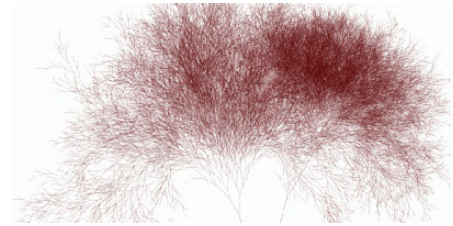


# Árvores binárias de busca



Fonte: <http://infosthetics.com/archives/>

## AULA 10

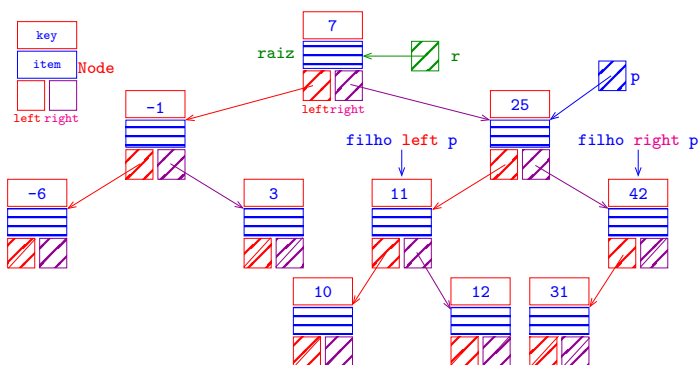
Referências: Árvores binárias de busca (PF); Binary Search Trees (S&W); slides (S&W)

### Árvore binárias de busca

Considere uma **árvore binária** cujos nós têm um campo **chave** (**key** como **int** ou **String**, por exemplo).

```
public class BST <Key extends
    Comparable<Key>, Value> {
    private Node r; // raiz
    private class Node {
        private Key key;
        private Value val;
        private Node left, right;
        public Node(Key key, Value val) {
            this.key = key; this.val = val;
        }
    }
}
```

### Ilustração de uma árvore binária de busca



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

### Árvore binárias de busca

Uma **árvore binária** deste tipo é de **busca** (em relação ao campo **key**) se para cada nó **x**: **x.key** é

1. **maior ou igual** à chave de qualquer nó na subárvore **esquerda** de **x** e
2. **menor** à chave de qualquer nó na subárvore **direita** de **x**.

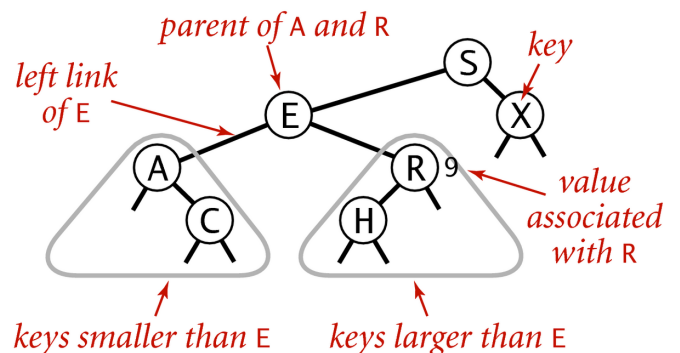
Assim, se **p** é um nó qualquer então vale que

$$q.key.compareTo(p.key) \leq 0 \text{ e}$$

$$p.key.compareTo(t.key) < 0$$

para todo nó **q** na subárvore **esquerda** de **p** e todo nó **t** na subárvore **direita** de **p**.

### Anatomia de uma árvore binária de busca



### Anatomy of a binary search tree

Fonte: [algs4](#)

## get(key)

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

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

Navigation icons

## get(key)

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) {
    // 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;
}
```

Navigation icons

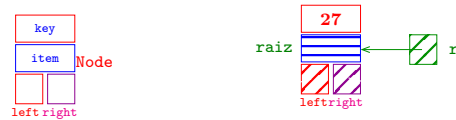
## get(key) versão iterativa

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;
}
```

Navigation icons

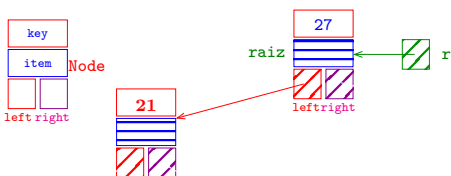
## Ilustração de put()



ordem put(): 27

Navigation icons

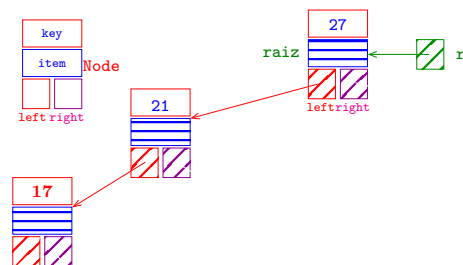
## Ilustração de put()



ordem put(): 27 21

Navigation icons

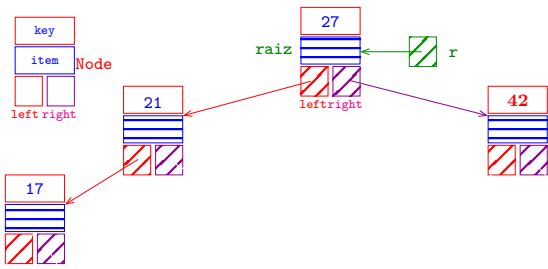
## Ilustração de put()



ordem put(): 27 21 17

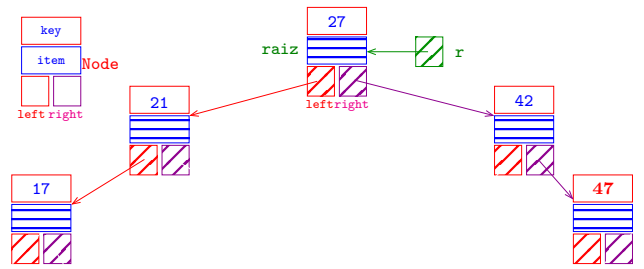
Navigation icons

### Ilustração de put ()



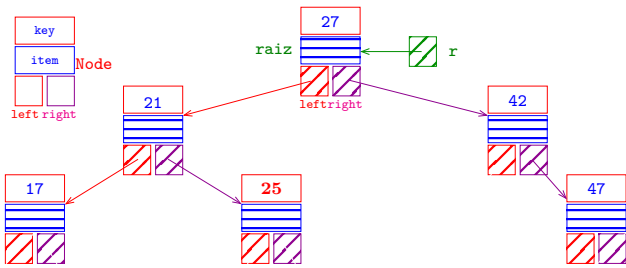
ordem put(): 27 21 17 42

### Ilustração de put ()



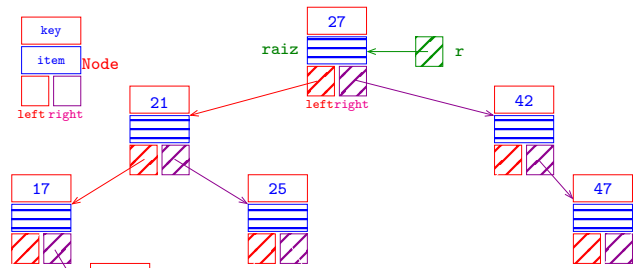
ordem put(): 27 21 17 42 47

### Ilustração de put ()



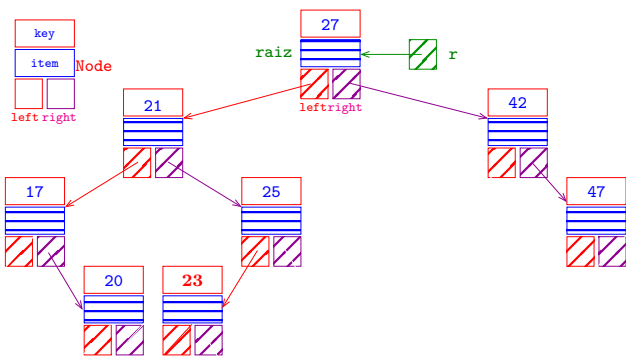
ordem put(): 27 21 17 42 47 25

### Ilustração de put ()



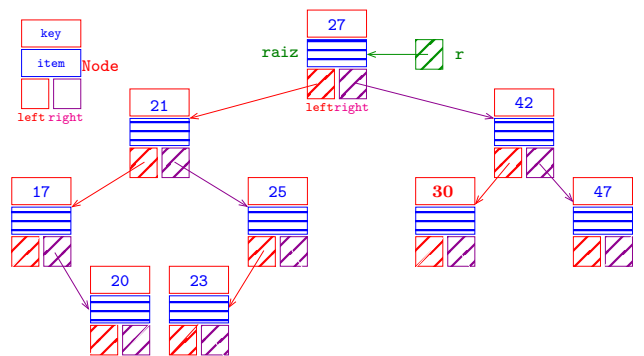
ordem put(): 27 21 17 42 47 25 20

### Ilustração de put ()



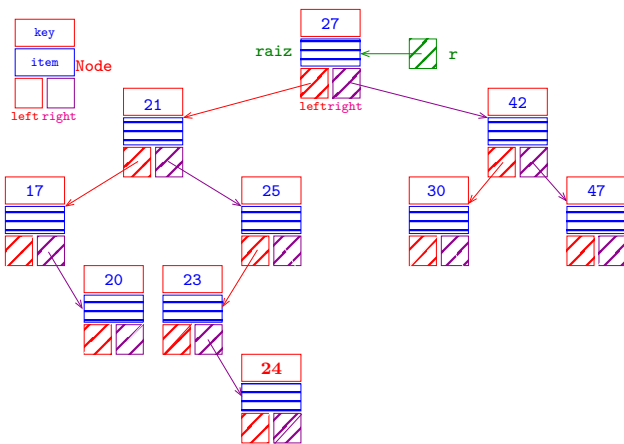
ordem put(): 27 21 17 42 47 25 20 23

### Ilustração de put ()



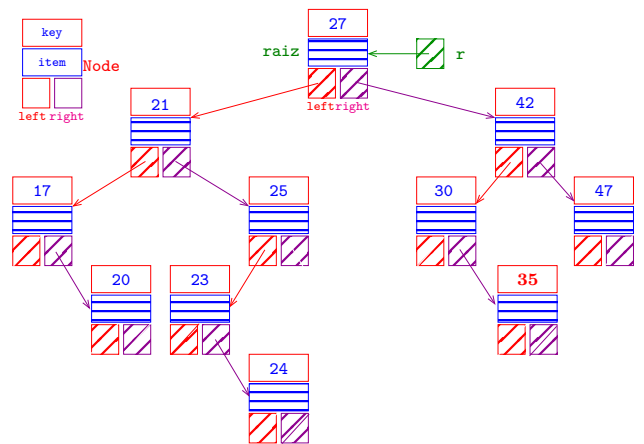
ordem put(): 27 21 17 42 47 25 20 23 30

### Ilustração de put ()



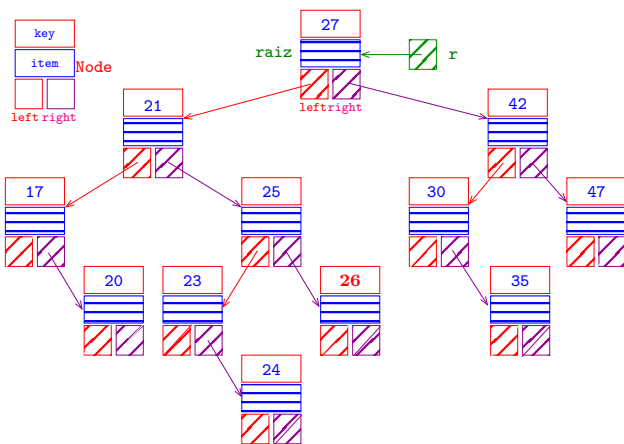
ordem put(): 27 21 17 42 47 25 20 23 30 24

### Ilustração de put ()



ordem put(): 27 21 17 42 47 25 20 23 30 24 35

### Ilustração de put ()



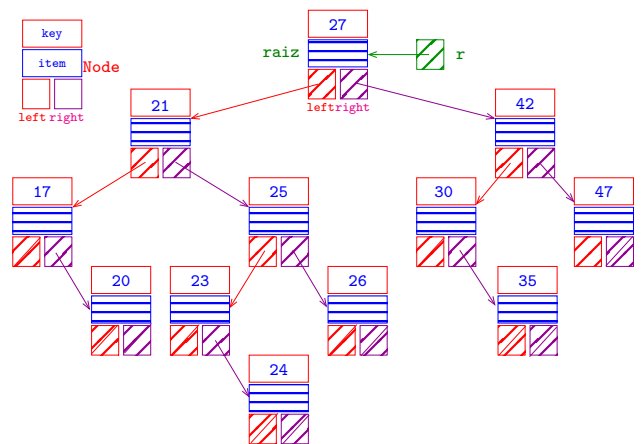
ordem put(): 27 21 17 42 47 25 20 23 30 24 35 26

put(key, val)

Recebe key e val r e insere um novo nó no lugar correto da árvore de modo que a árvore continue sendo de busca. Se key está na BST o seu valor é atualizado.

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

### Ilustração de put ()

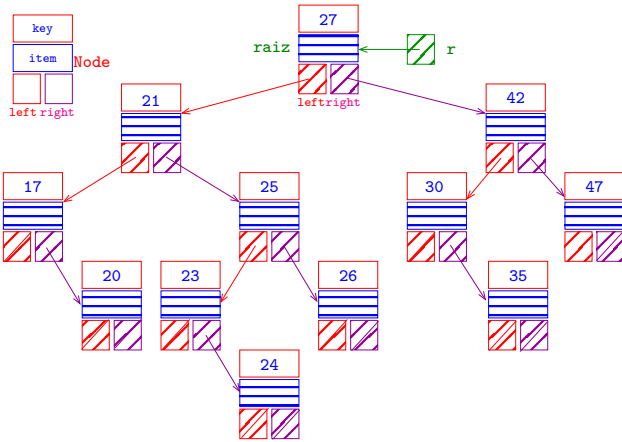


ordem put(): 27 21 17 42 47 25 20 23 30 24 35 26

put(key, val)

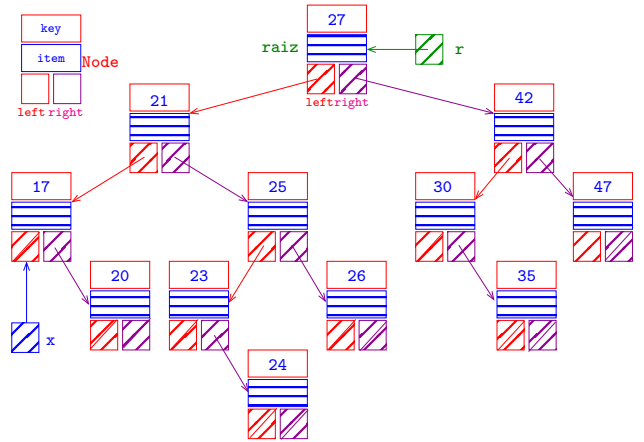
```
private Node put(Node x, Key key,
                 Value val) {
    if (x == null)
        return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = put(x.left, key, val);
    else if (cmp > 0)
        x.right = put(x.right, key, val);
    else x.val = val;
    return x;
}
```

### Ilustração de min()



Navigation icons: back, forward, search, etc.

### Ilustração de min()



Navigation icons: back, forward, search, etc.

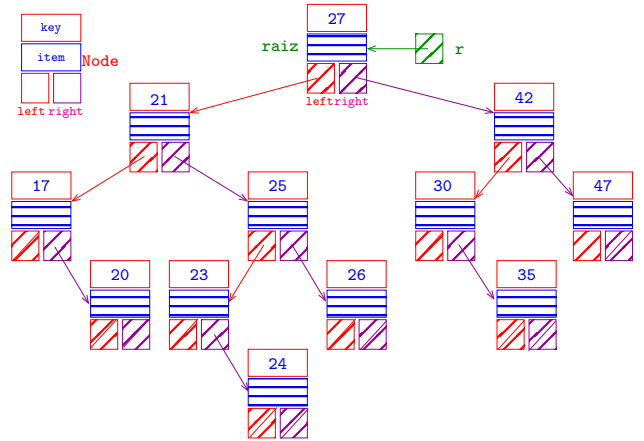
### min(key)

```
// Retorna a menor chave na ST.
public Key min() {
    if (r == null) return null;
    return min(r).key;
}

// Retorna o nó da menor chave da
// subárvore cuja raiz é x.
// Supõe que x não é null.
private Node min(Node x) {
    if (x.left == null) return x;
    return min(x.left);
}
```

Navigation icons: back, forward, search, etc.

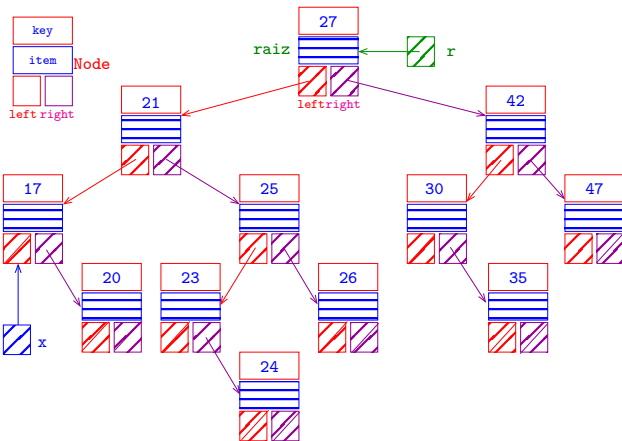
### Ilustração de deleteMin()



Navigation icons: back, forward, search, etc.

x ← nó com a menor chave

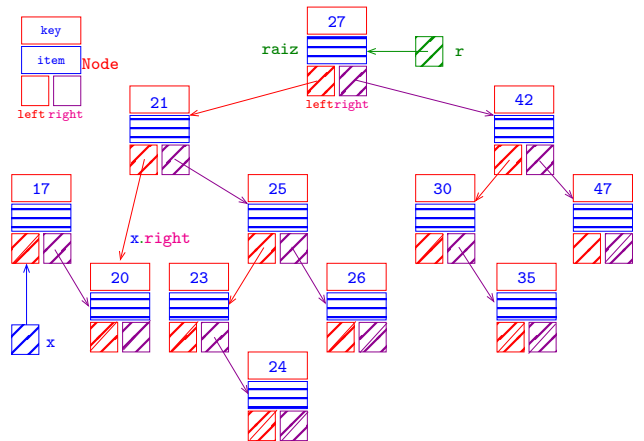
### Ilustração de deleteMin()



remove x da BST

Navigation icons: back, forward, search, etc.

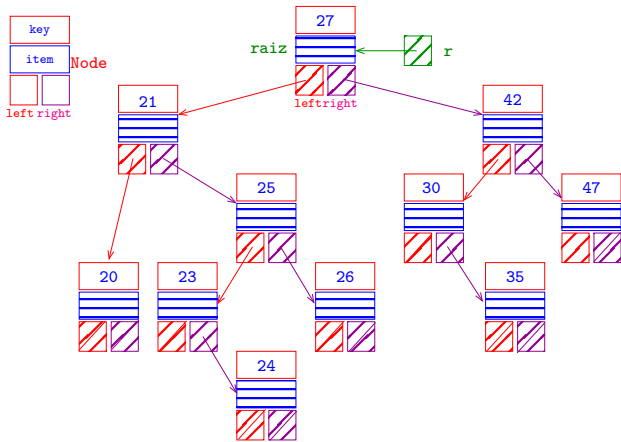
### Ilustração de deleteMin()



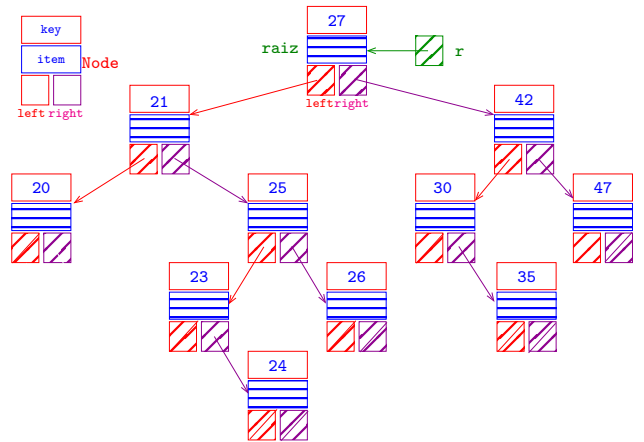
coletor de lixo em ação

Navigation icons: back, forward, search, etc.

### Ilustração de deleteMin()



### Ilustração de deleteMin()

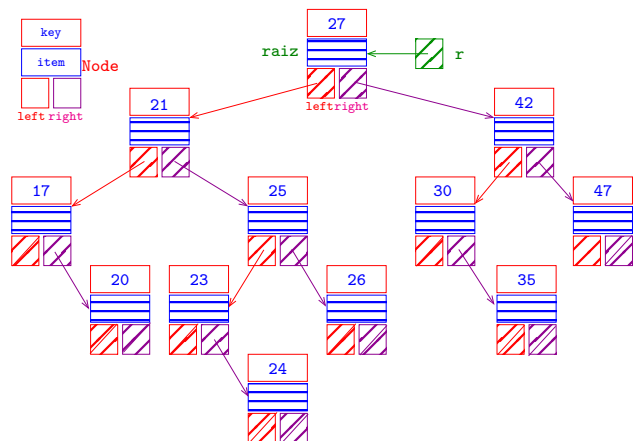


### deleteMin()

```
// Remove o nó que tem a menor chave.
public void deleteMin() {
    if (r == null) return;
    r = deleteMin(r);
}

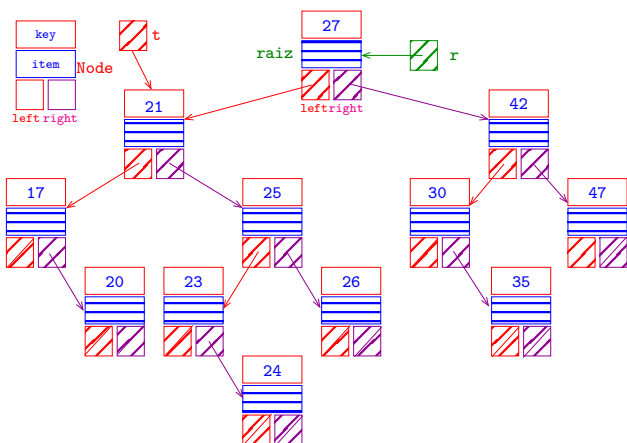
// Remove o nó que tem a menor chave
// na subárvore (não vazia) cuja raiz é x
// e retorna a raiz da sub. resultante.
private Node deleteMin(Node x) {
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    return x;
}
```

### Ilustração de delete(21)



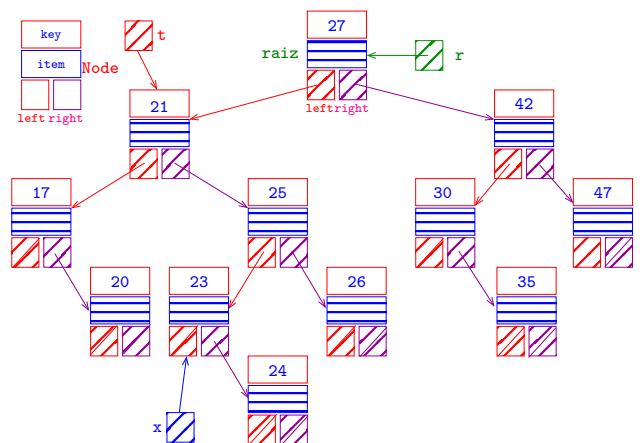
t ← nó com 21

### Ilustração de delete(21)



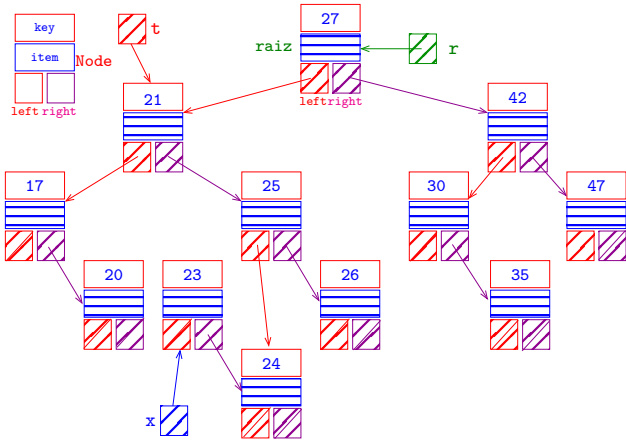
x ← menor nó em t.right (=min(t.right))

### Ilustração de delete(21)



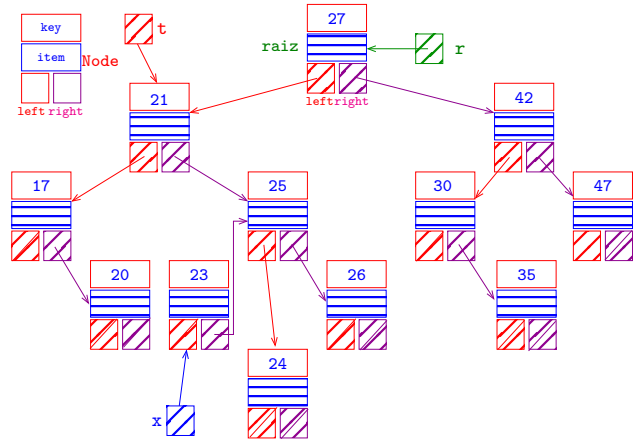
remove x da árvore (=deleteMin(t.right))

### Ilustração de delete(21)



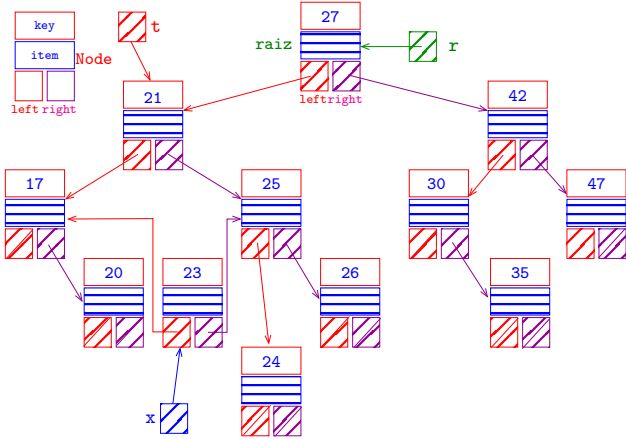
ajuste  $x.right$  ( $x.right \leftarrow deleteMin(t.right)$ )

### Ilustração de delete(21)



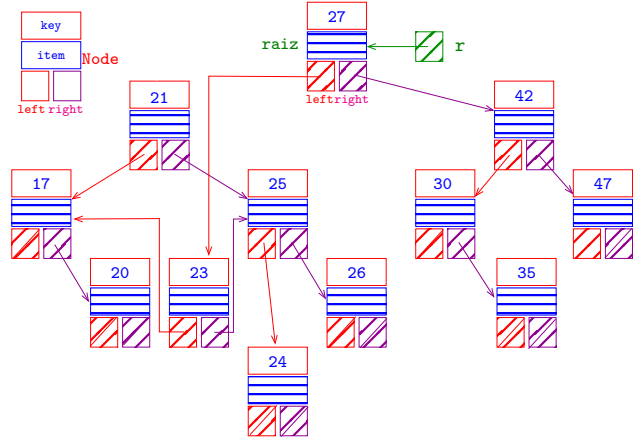
ajuste  $x.left$  ( $x.left \leftarrow t.left$ )

### Ilustração de delete(21)



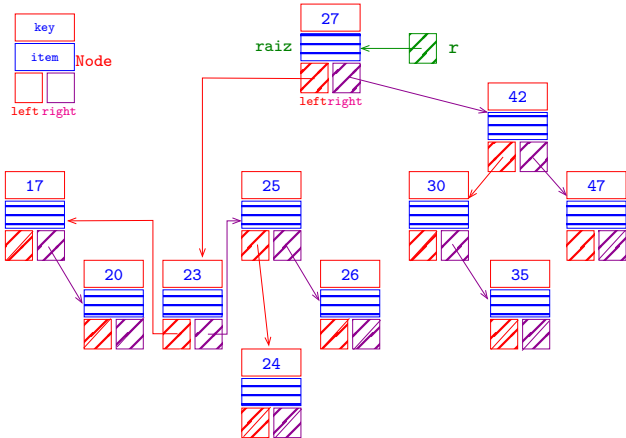
retorne  $x$  para ajustar a referência que era para  $t$

### Ilustração de delete(21)

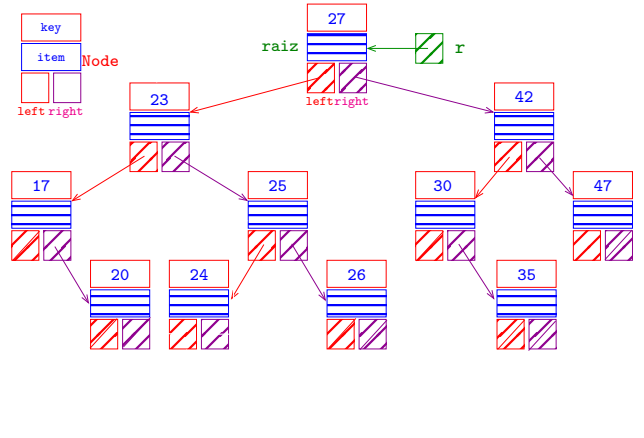


coletor de lixo em ação

### Ilustração de delete(21)



### Ilustração de delete(21)







## 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

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

◀ ▶ ⏪ ⏩ 🔍 ↺

## BST aleatória

**Fato.** Buscas com **sucesso** numa **BST aleatória** com **n** chaves requer cerca de  $2 \lg n$  comparações na média.

**Demonstração:** O número de comparações para encontrados uma dada chave é 1 mais a **profundidade do nó que contém a chave**. Se somarmos todas as profundidades dos nós de uma árvore obtemos o comprimento interno **C** da árvore.

O **número médio de comparações** para uma busca com sucesso nessa árvore é portanto  $1 + C/n$ .

◀ ▶ ⏪ ⏩ 🔍 ↺

## BST aleatória

Agora vem um **truque meio manjado**. **Primeiro**, trocando **n** por **n-1** na última igualdade temos que

$$(n-1)C_{n-1} = 2(C_0 + C_1 + \dots + C_{n-2}) + (n-1)(n-2).$$

**Segundo**, fazemos

$$nC_n - (n-1)C_{n-1} = 2C_{n-1} + 2(n-1).$$

**Rearranjando** os termos e dividindo por  $n(n+1)$  obtemos

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2(n-1)}{n(n+1)} < \frac{C_{n-1}}{n} + 2\frac{1}{n}$$

◀ ▶ ⏪ ⏩ 🔍 ↺

## BST aleatória

Uma **BST aleatória** é uma **BST** que se obtém inserindo **n** chaves **distintas** em **ordem aleatória** numa árvore inicialmente vazia.

Qual é o **número esperado** de comparações em uma busca com sucesso em uma **BST aleatória** com **n** chaves?

◀ ▶ ⏪ ⏩ 🔍 ↺

## BST aleatória

Seja  $C_n$  o **comprimento interno** de uma **BST aleatória** com **n** chaves. Temos que  $C_0 = 0$ ,  $C_1 = 0$  e

$$\begin{aligned} C_n &= ((C_0 + C_{n-1} + n-1) \\ &+ (C_1 + C_{n-2} + n-1) \\ &+ \dots \\ &+ (C_{n-1} + C_0 + n-1))/n. \end{aligned}$$

Cada  $n-1$  é devido a **contribuição da raiz** às profundidades das duas subárvores.

Reescrevendo a terceira igualdade obtemos

$$nC_n = 2(C_0 + C_1 + \dots + C_{n-1}) + n(n-1).$$

◀ ▶ ⏪ ⏩ 🔍 ↺

## BST aleatória

A última **recorrência** que pode ser "**desenrolada**" até o caso base

$$\frac{C_n}{n+1} < \frac{C_1}{2} + 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n-1} + \frac{1}{n}\right).$$

Assim,

$$C_n < 2(n+1)\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n-1} + \frac{1}{n}\right).$$

Sabemos que

$$\sum_2^n \frac{1}{n} < \int_1^n \frac{1}{x} dx = \ln n - \ln 1 = \ln n.$$

◀ ▶ ⏪ ⏩ 🔍 ↺

## BST aleatória

Logo, como  $2/\lg e = 1.3862\dots$

$$C_n < 2(n+1) \ln n = 2(n+1) \frac{\lg n}{\lg e} < 1.39(n+1) \lg n$$

Portanto, concluímos que na média o número de comparações em uma BST aleatória é

$$1 + \frac{1.39(n+1) \lg n}{n} \sim 2 \lg n.$$

## Consumo de tempo

O número esperado de nós visitados durante uma busca em uma BST aleatória não passa de  $3 \lg n$