



Fonte: [ash.atozviews.com](http://ash.atozviews.com)

## Nos episódios anteriores ...

Navigation icons: back, forward, search, etc.

### Tabelas de símbolos

Uma **tabela de símbolos** (*ST = symbol table*) é um **ADT** que consiste em um conjunto de itens, sendo cada item um par **chave-valor** ou **key-value**, munido de duas operações fundamentais:

- ▶ `put()`, que **insere** um novo item na **ST**, e
- ▶ `get()`, que **busca** o valor associado a uma dada chave.

Navigation icons: back, forward, search, etc.

### API ST

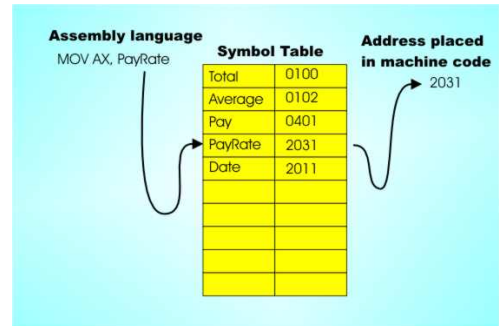
```
public class ST<Key, Value>
```

```
public class ST
```

	<code>ST()</code>	cria uma ST
<code>void</code>	<code>put(Key key, Value val)</code>	insere (key, val)
<code>Value</code>	<code>get(Key key)</code>	busca o valor associado a key
	<code>delete(Key key)</code>	remove (key, val)
<code>void</code>	<code>rank(Key key)</code>	no. de keys menor que key
<code>int</code>		ST está vazia?
<code>boolean</code>	<code>isEmpty()</code>	a key está na ST?
<code>boolean</code>	<code>contains(Key key)</code>	lista todas as chaves na ST
<code>Iterable&lt;Key&gt;</code>	<code>keys()</code>	

Navigation icons: back, forward, search, etc.

## Tabelas de Símbolos



Fonte: <http://www.i-programmer.info/>

### Tabelas de símbolos (PF) Elementary Symbol Tables (S&W)

Navigation icons: back, forward, search, etc.

### Tabelas de símbolos

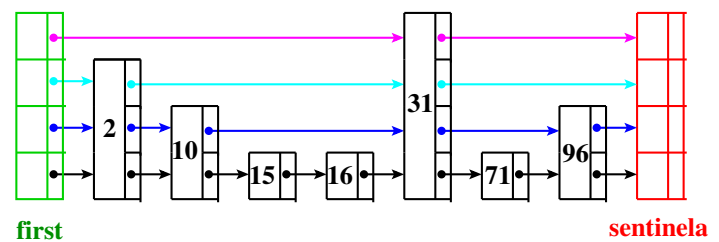
Convenções sobre **STs**:

- ▶ **não há** chaves repetidas (as chaves são duas a duas distintas),
- ▶ **null nunca** é usado como **key**,
- ▶ **null nunca** é usado como **value** associado a uma **key**.

**STs** são também chamadas de **dictionaries**, **maps** e **associative arrays**.

Navigation icons: back, forward, search, etc.

### Aleatorização: skip list



Navigation icons: back, forward, search, etc.

## 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

Tempos em segundos obtidos com **StopWatch**.

## Doubling Method

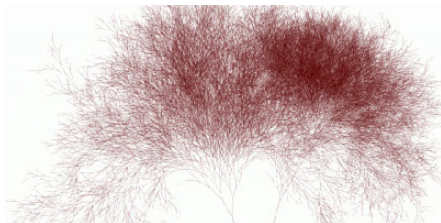
**Hipótese.** O consumo de tempo do programa é  $T(n) = an^b$ .

**Consequência** Quando  $n$  crescem  $T(2n)/T(n)$  se aproxima de  $2^b$ .

Doubling method:

- ▶ **comece** com um  $n$  moderado
- ▶ **registre** o consumo de tempo
- ▶ **dobre** o valor de  $n$
- ▶ **repita** enquanto for possível
- ▶ **verifique** que a razão de tempos consecutivos se aproxima de  $2^b$
- ▶ **prediga e extrapole:** multiplique por  $2^b$  para estimar  $T(2n)$

## Árvores binárias de busca



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

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

## Análise de desempenho

Estamos interessados em consumo de **tempo** e **memória**.

Para sabermos se podemos resolver problemas **MAIORES**.

**Motivação:**

- ▶ **prever** o comportamento de programas
- ▶ **comparar** algoritmos e suas implementações
- ▶ **compreender** o problema para desenvolver novos algoritmos

**Histórias de sucesso:** *Discrete Fourier Transform*: processamento de sinais

## AULA 10

## 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;
        }
    }
}
```

## 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**.

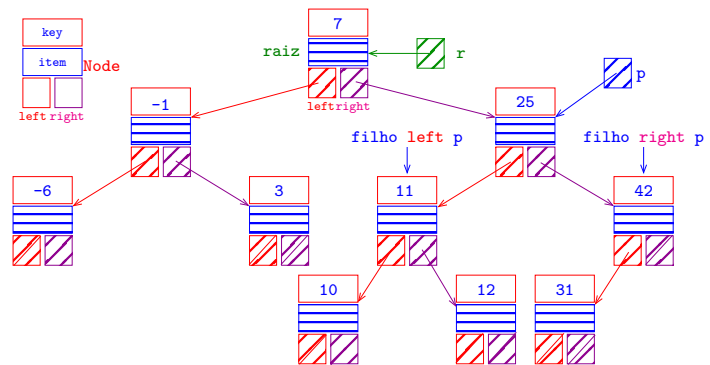
Assim, se **p** é um nó qualquer então vale que

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

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

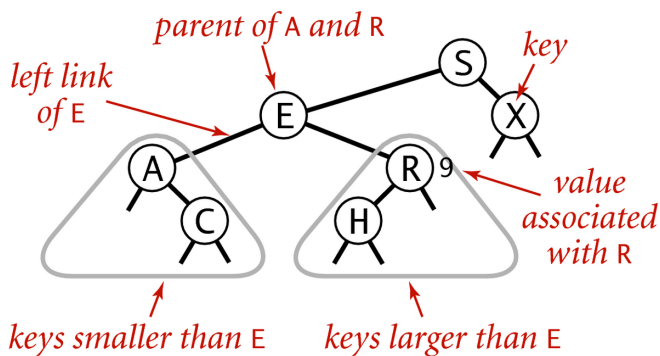
para todo nó **q** na subárvore **esquerda** de **p** e todo nó **t** na subárvore **direita** de **p**.

## Ilustração de uma árvore binária 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](#)

## BST: 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;
}
```

## BST: 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);
    if (cmp > 0) return get(x.right, key);
    return x;
}
```

## BST: get(key) versão iterativa

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

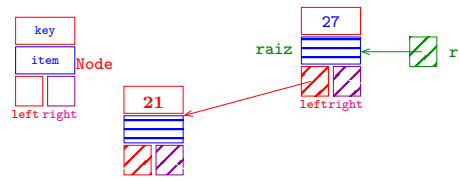
```
public Value get(Node x, Key key) {
    while (x != null && !x.key.equals(key)) {
        int cmp = key.compareTo(x.key);
        if (cmp > 0) x = x.left;
        else x = x.right;
    }
    if (x == null) return null;
    return x.val;
}
```

### Ilustração de put ()



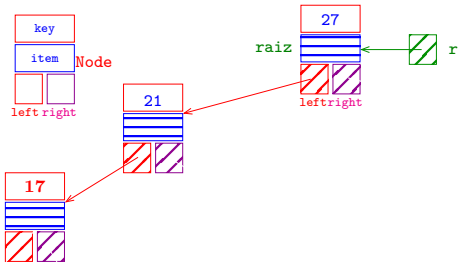
ordem put(): 27

### Ilustração de put ()



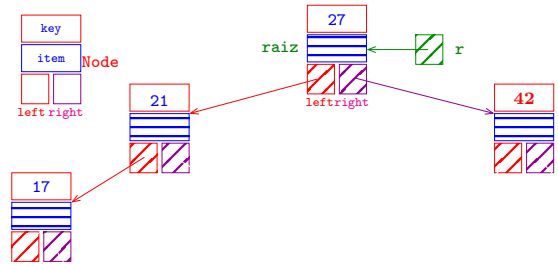
ordem put(): 27 21

### Ilustração de put ()



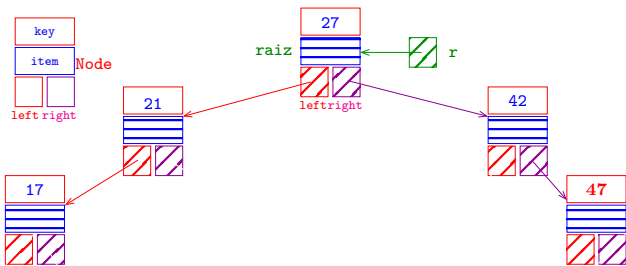
ordem put(): 27 21 17

### Ilustração de put ()



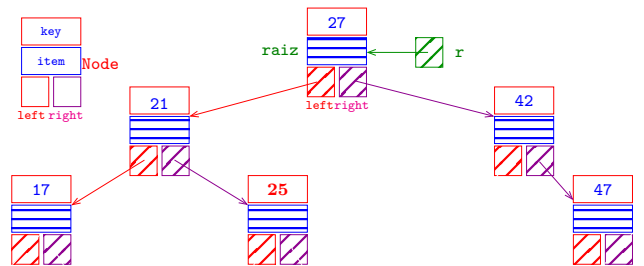
ordem put(): 27 21 17 42

### Ilustração de put ()



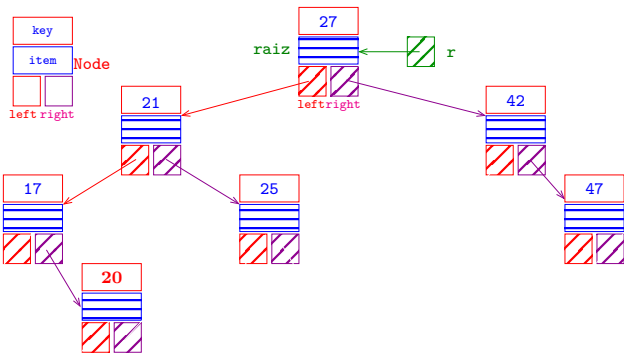
ordem put(): 27 21 17 42 47

### Ilustração de put ()



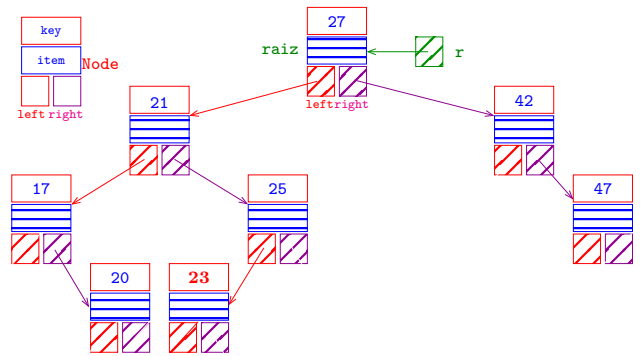
ordem put(): 27 21 17 42 47 25

### Ilustração de put ()



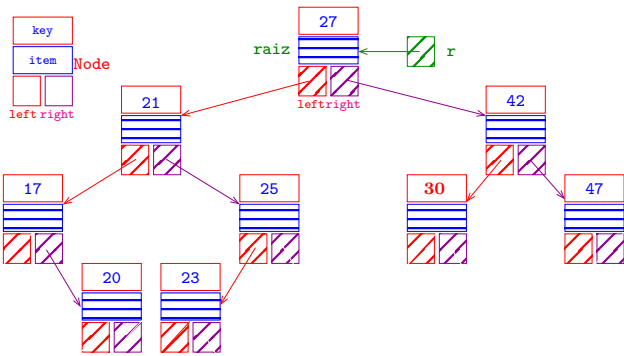
ordem put(): 27 21 17 42 47 25 **20**

### Ilustração de put ()



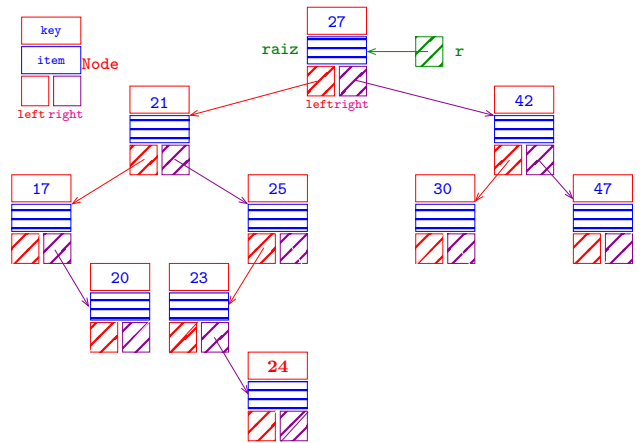
ordem put(): 27 21 17 42 47 25 20 **23**

### Ilustração de put ()



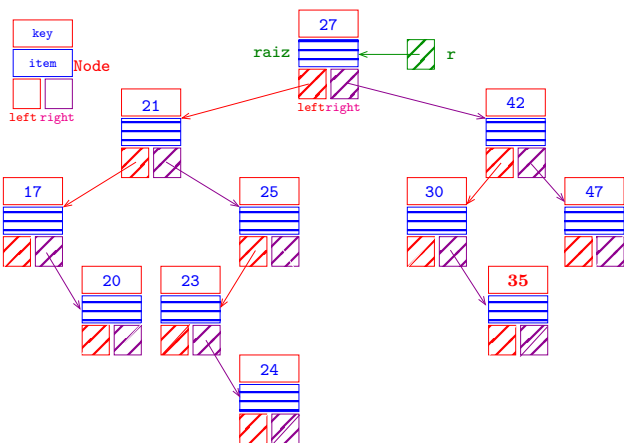
ordem put(): 27 21 17 42 47 25 20 23 **30**

### Ilustração de put ()



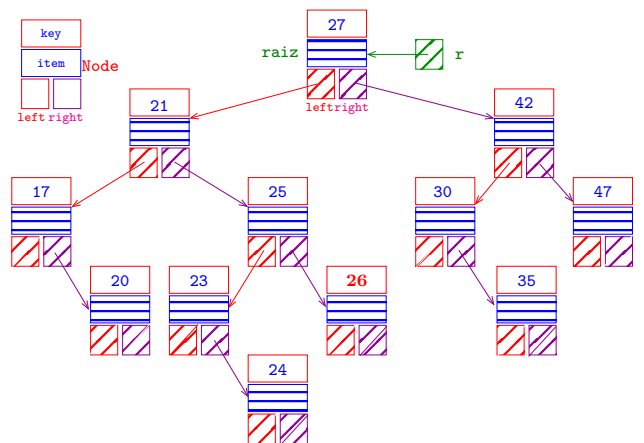
ordem put(): 27 21 17 42 47 25 20 23 30 **24**

### Ilustração de put ()



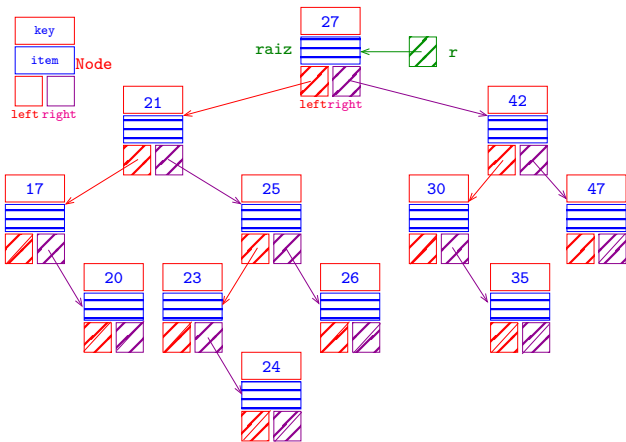
ordem put(): 27 21 17 42 47 25 20 23 30 24 **35**

### Ilustração de put ()



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

## Ilustração de put()



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

## BST: put(key, val)

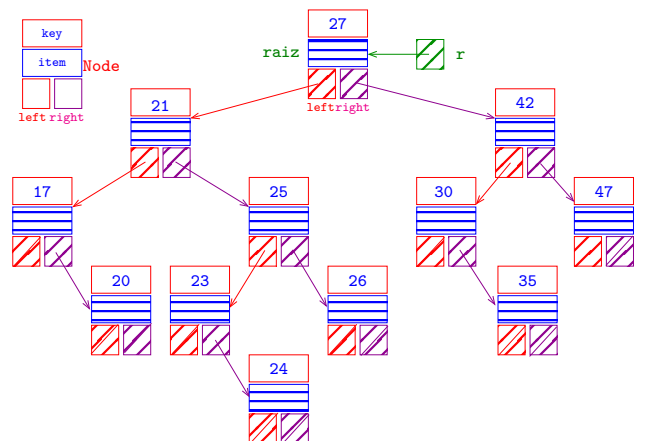
Recebe **key** e **val** e **insere** um novo nó no lugar correto da árvore de modo que a árvore **continue** sendo de busca. Se **key** está na **BST** o seu valor é atualizado.

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

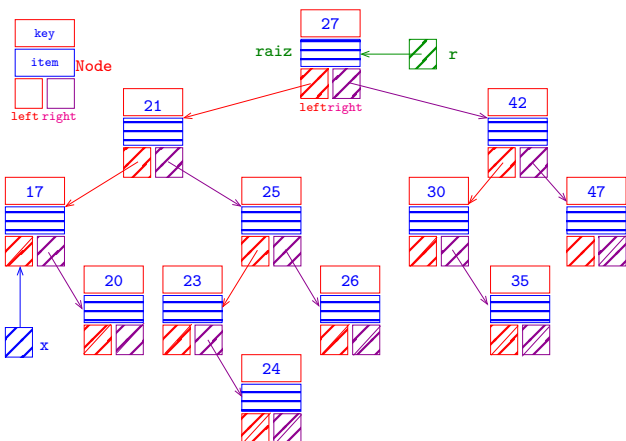
## BST: put(key, val)

```
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()



## Ilustração de min()

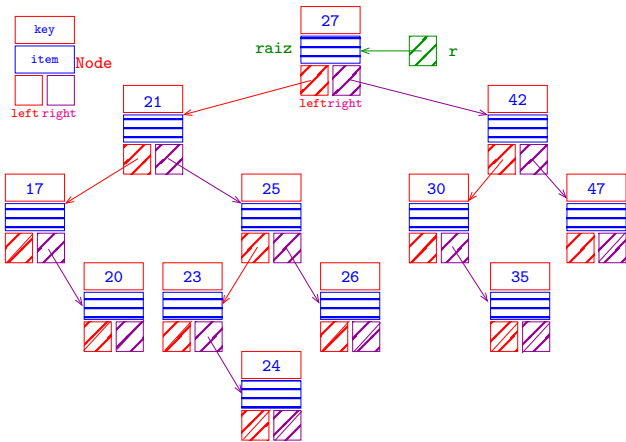


## BST: min(key)

```
// 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.
private Node min(Node x) {
    if (x == null) return null;
    if (x.left == null) return x;
    return min(x.left);
}
```

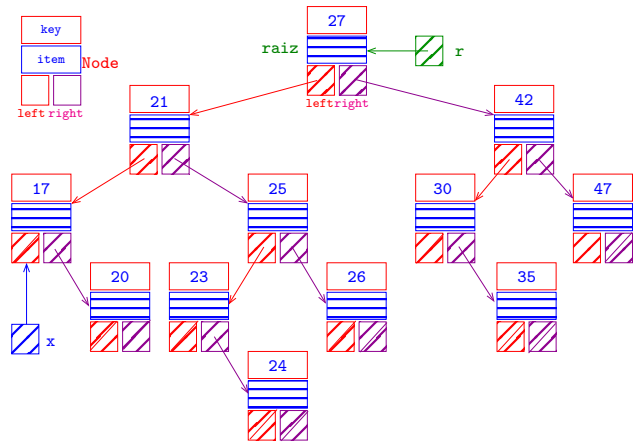
### Ilustração de deleteMin()



x ← nó com a menor chave



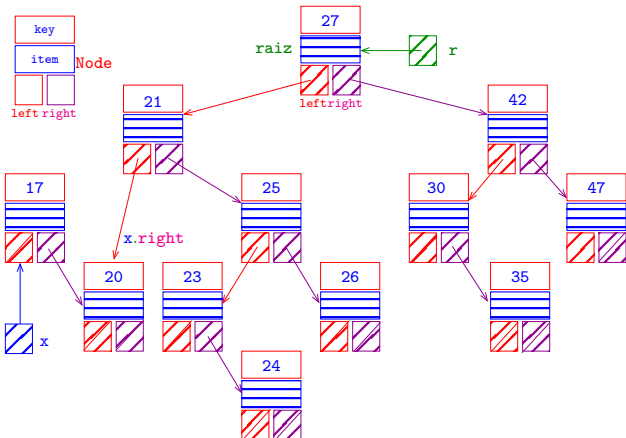
### Ilustração de deleteMin()



remove x da BST



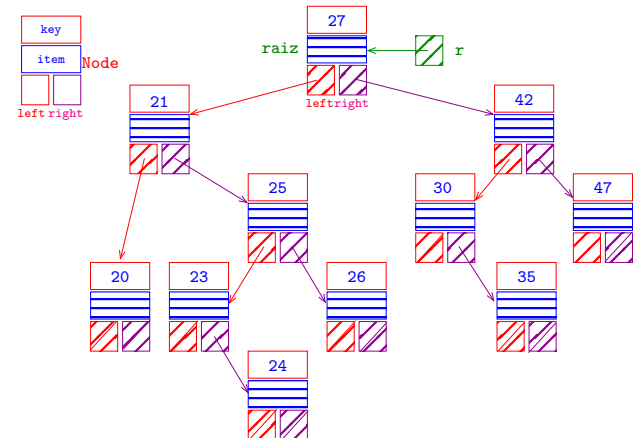
### Ilustração de deleteMin()



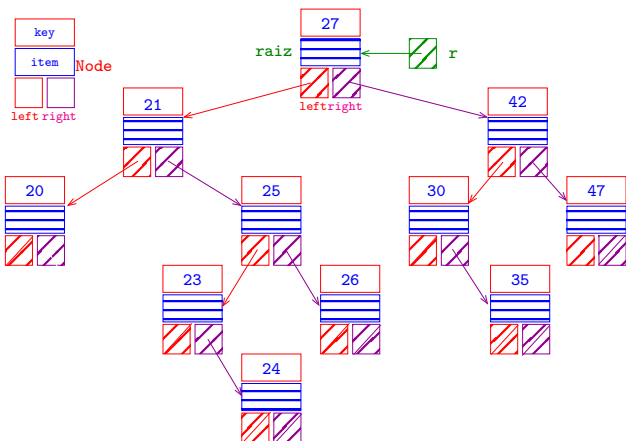
coletor de lixo em ação



### Ilustração de deleteMin()



### Ilustração de deleteMin()

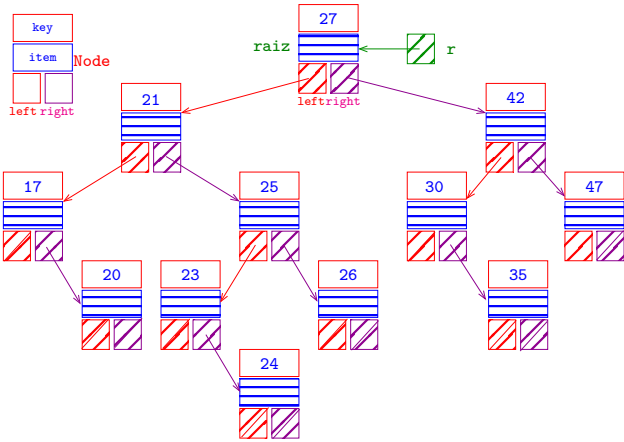


### BST: deleteMin()

```
// 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 == null) return null;
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    return x;
}
```

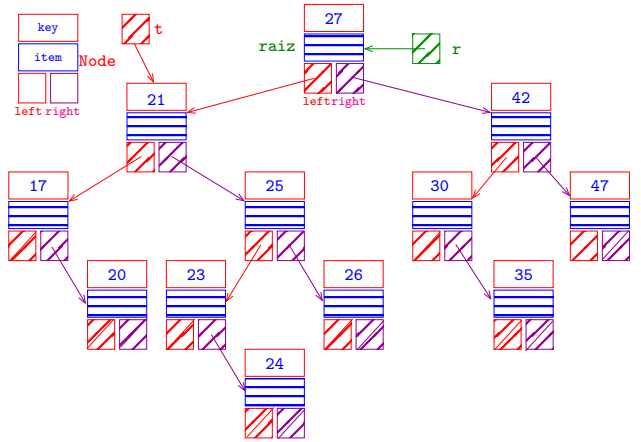
### Ilustração de delete(21)



$t \leftarrow$  nó com 21



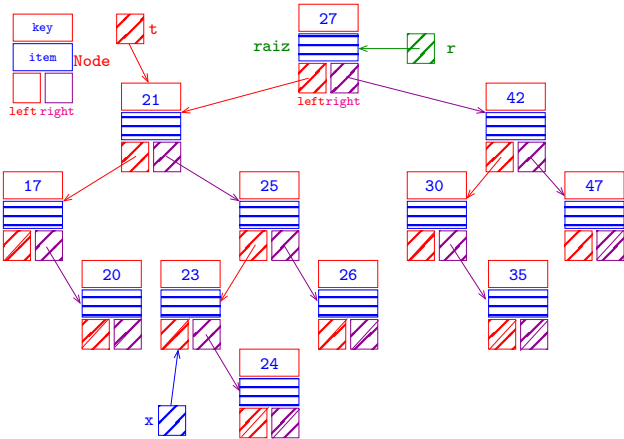
### Ilustração de delete(21)



$x \leftarrow$  menor nó em  $t.right$  ( $=\min(t.right)$ )



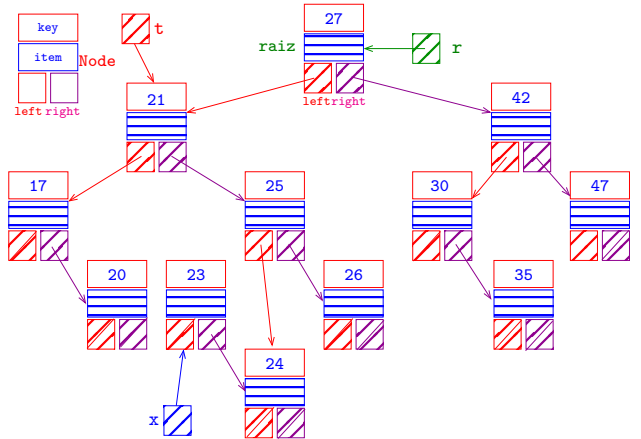
### Ilustração de delete(21)



remova  $x$  da árvore ( $=\text{deleteMin}(t.right)$ )



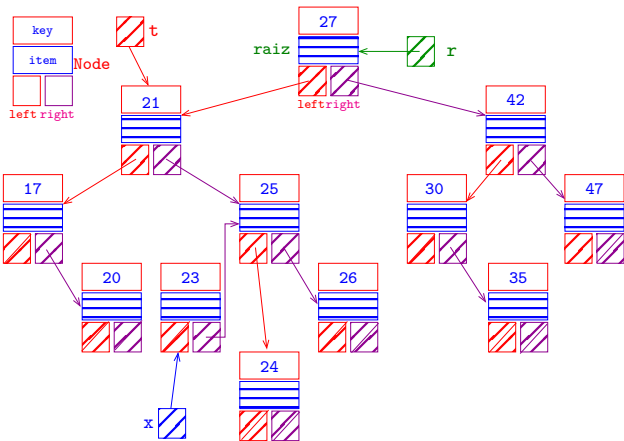
### Ilustração de delete(21)



ajuste  $x.right$  ( $x.right \leftarrow \text{deleteMin}(t.right)$ )



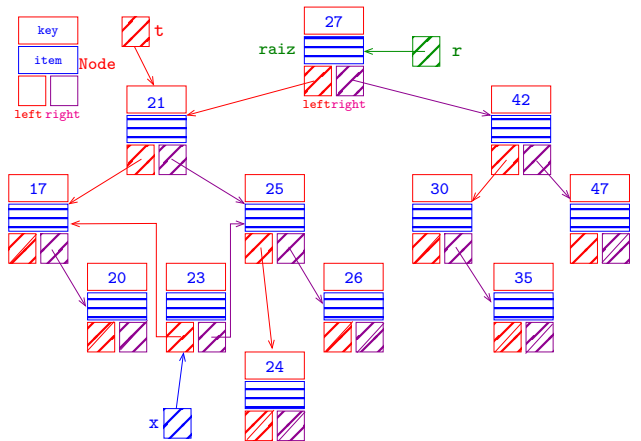
### Ilustração de delete(21)



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



### Ilustração de delete(21)

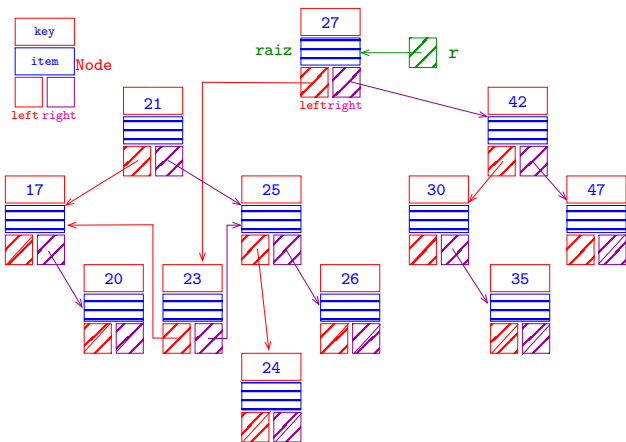


retorne  $x$  para ajustar a referência que era para  $t$





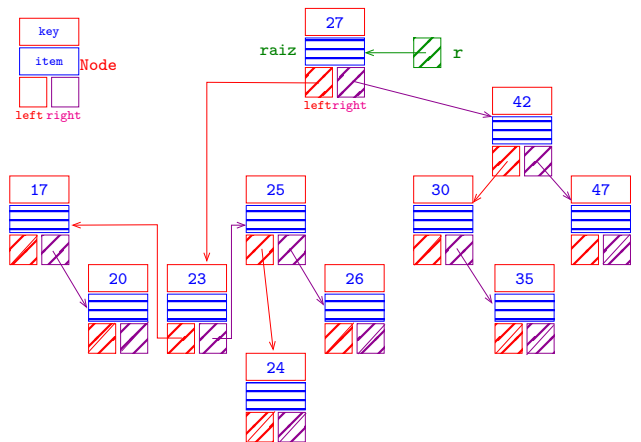
### Ilustração de delete(21)



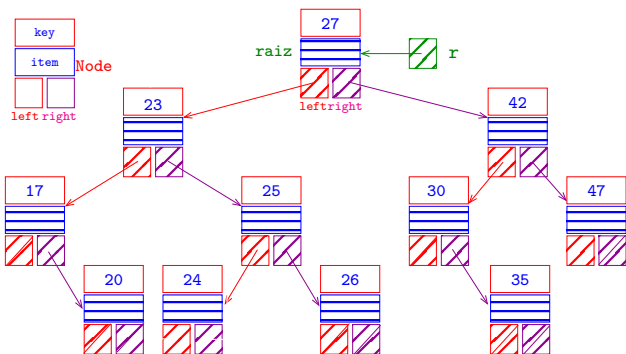
coletor de lixo em ação



### Ilustração de delete(21)



### Ilustração de delete(21)



### BST: delete(key)

Remove o nó que contém a chave `key`.  
Se nenhum nó contém `key`, não faz nada.

```
public void delete(Key key) {
    r = delete(r, key);
}
```



### BST: delete(key)

```
private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = delete(x.left, key);
    else if (cmp > 0)
        x.right = delete(x.right, key);
    else{
        // não tem algum filho
        if (x.right == null) return x.left;
        if (x.left == null) return x.right;
```



### BST: delete(key)

```
// x tem ambos os filhos
Node t = x;
x = min(t.right); // x.left == null
x.right = deleteMin(t.right);
x.left = t.left;
}
return x;
}
```



### BST aumentada

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

### BST: rank()

Retorna o número de chaves estritamente menores que `key`.

```
public int rank(Key key) {
    return rank(key, r);
}
```

### BST: size()

Retorna o número de pares `key-val` na BST.

```
public int size() {
    return size(r);
}

// retorna o número de nós na BST
// de raiz x
private int size(Node x) {
    if (x == null) return 0;
    return x.n;
}
```

### BST: rank()

Retorna o número de chaves estritamente menores que `key`.

```
private int rank(Key key, Node x) {
    if (x == null) return 0;
    int cmp = x.key.compareTo(key);
    if (cmp > 0) return rank(key, x.left);
    if (cmp < 0)
        return 1 + size(x.left)
            + rank(key, x.right);
    return size(x.left);
}
```

### BST: put(key, val) aumentado

```
private Node put(Node x, Key key,
    Value val) {
    if (x == null)
        return new Node(key, val, 1);
    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;
    x.n = 1 + size(x.left)
        + size(x.right);
    return x;
}
```

### BST: put(key, val) aumentado

```
private Node deleteMin(Node x) {
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.n = size(x.left) + size(x.right) + 1;
    return x;
}
```

## 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

A **ideia** de **BST** é realizarmos uma espécie de "**busca binária dinâmica**".

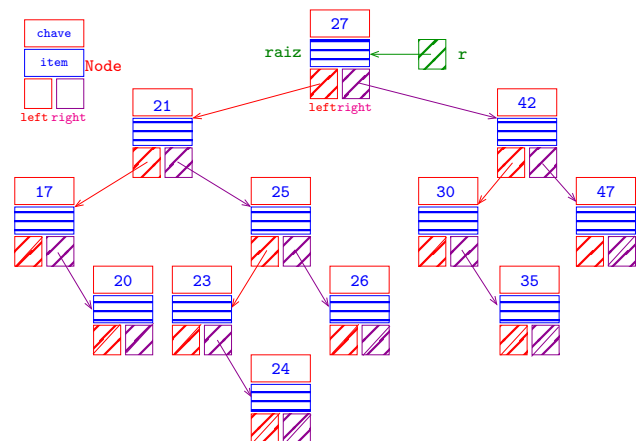
Gostaríamos de combinar a eficiência da busca por chaves ordenados (`get()`) com a eficiência da inserção (`put()`) e remoção (`delete()`).

Qual é o número de **chaves examinadas/comparadas** em uma busca com sucesso?

**Busca com sucesso** = busca em que a chave procurada está na **BST**.

Toda operação de busca ou inserção visita  $1 + p$  nós, sendo  $p$  a **profundidade** do **último nó visitado**.

## 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

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

## 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

**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

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  $2 \lg n$