SBVORIN: Organização e Recuperação da Informação

Aula 05: Revisão de Árvores Binárias de Busca Elementares, Hibbard Deletion e Árvore Binária de Busca Balanceada AVL (Adelson-Velskii e Landis)

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

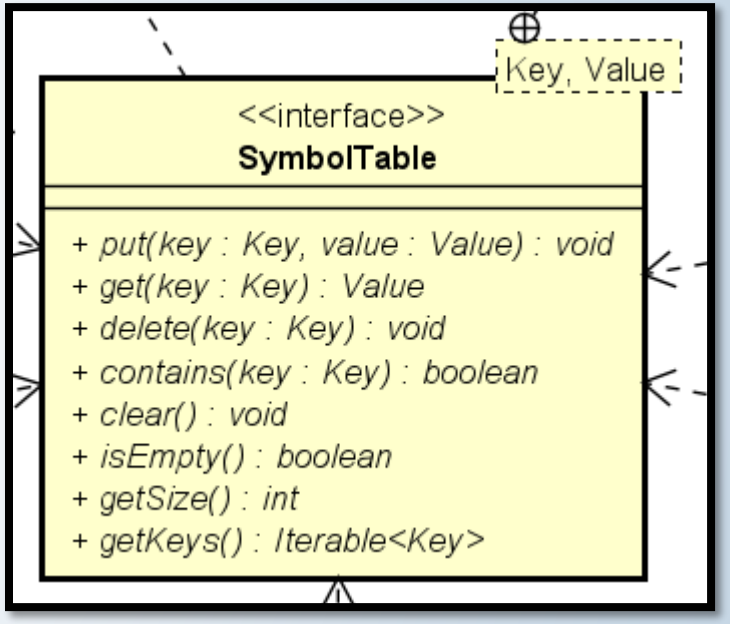
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Árvore Binária de Busca

- **Invariante/Condição de existência:**
  - Todos os nós de uma subárvore direita são maiores que o nó raiz;
  - Todos os nós de uma subárvore esquerda são menores que o nó raiz;
  - Cada subárvore é também uma árvore binária de busca.

Árvore Binária de Busca

- **Implementação:**
  - Veremos agora a ideia por trás das operações de inserção (**`put`**) e remoção (**`delete`**) de nós de uma árvore binária de busca;
  - Na nossa implementação real, contida na classe **`BinarySearchTree`**, as operações são feitas, em sua maioria, usando uma chave (**`key`**) como identificador do nó, tendo um valor associado à ela;
  - As operações inerentes à nossa implementação real estão atreladas a API da tabela de símbolos;
  - Nos próximos slides consideraremos, por questão de simplicidade (por enquanto), que a árvore armazena apenas as chaves.



Árvore Binária de Busca

Inserção (`put`)

Árvore Binária de Busca

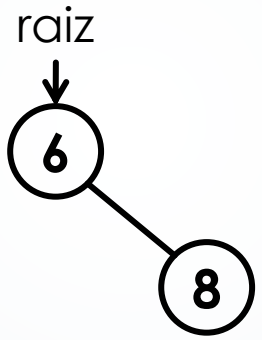
Inserção (`put`)

```
abb.put( 6 );
```

Árvore Binária de Busca

Inserção (`put`)

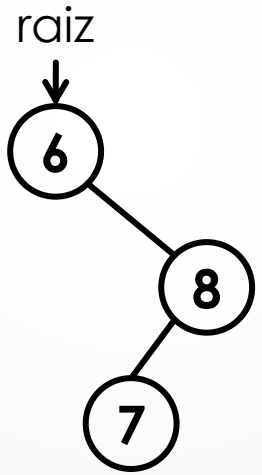
```
abb.put( 8 );
```



Árvore Binária de Busca

Inserção (`put`)

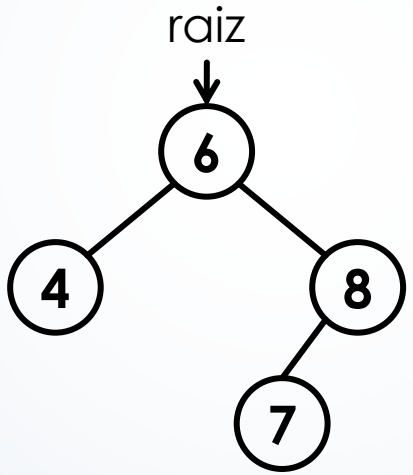
```
abb.put( 7 );
```



Árvore Binária de Busca

Inserção (`put`)

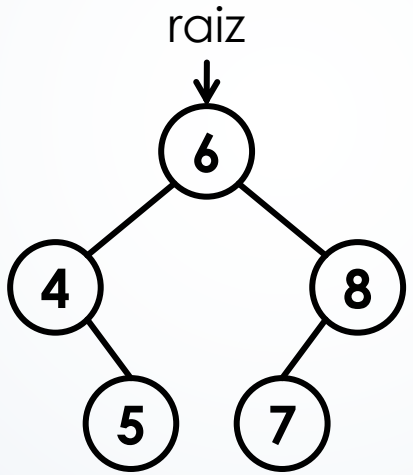
```
abb.put( 4 );
```



Árvore Binária de Busca

Inserção (`put`)

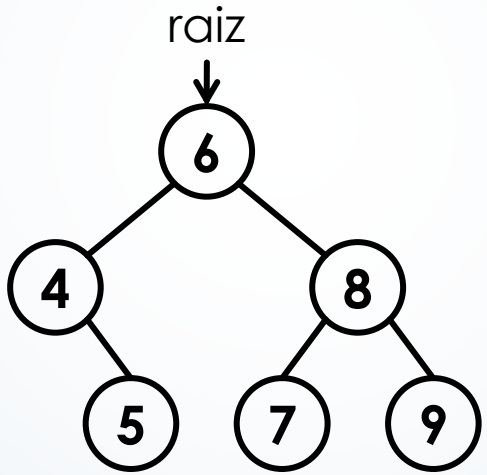
```
abb.put( 5 );
```



Árvore Binária de Busca

Inserção (`put`)

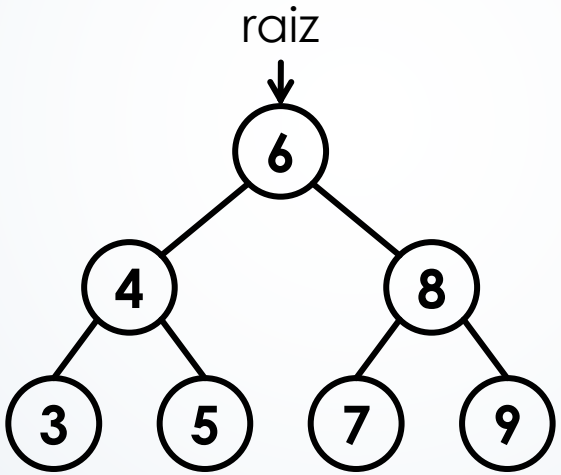
```
abb.put( 9 );
```



Árvore Binária de Busca

Inserção (`put`)

```
abb.put( 3 );
```



Árvore Binária de Busca

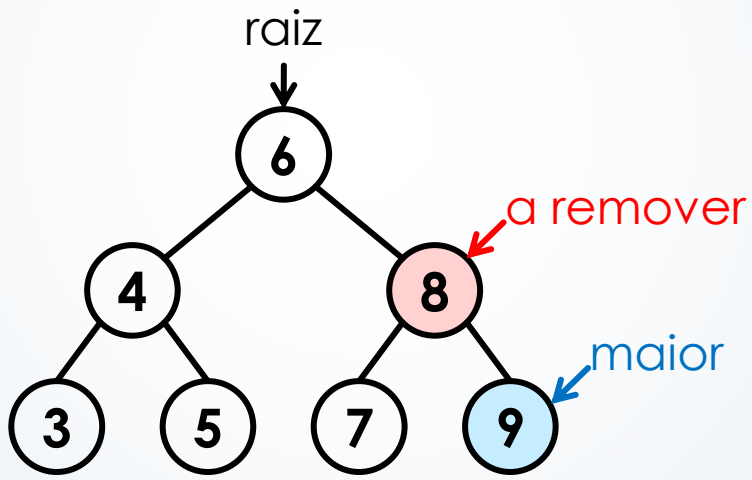
Remoção (`delete`)

- Se:
  - Todos os nós de uma subárvore direita são maiores que o nó raiz;
  - Todos os nós de uma subárvore esquerda são menores que o nó raiz;
  - Cada subárvore é também uma árvore binária de busca;
- Então:
  - A remoção de um nó com filhos faz com que o nó maior que ele (à sua direita) ocupe sua posição;
  - Se o nó à direita não existe, então o nó à esquerda ocupa sua posição;
  - Caso o nó seja um nó folha, não há a necessidade de reestruturação da árvore;
- Essas condições são necessárias para manter a invariante das árvores binárias de busca. Note que pode-se realizar a mesma operação de forma reflexiva, ou seja, para o outro lado.

Árvore Binária de Busca

Remoção (`delete`)

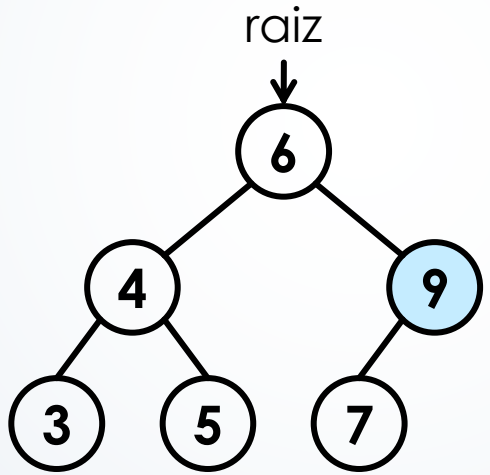
```
abb.delete( 8 );
```



Árvore Binária de Busca

Remoção (`delete`)

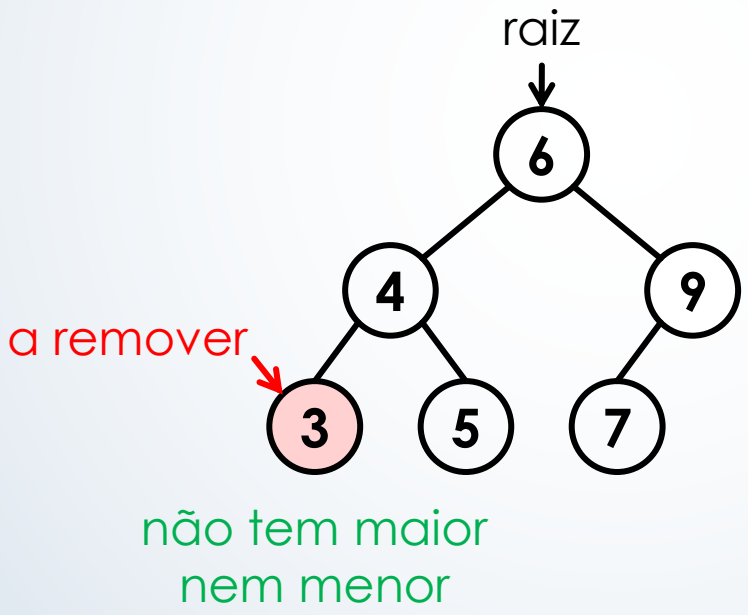
```
abb.delete( 8 );
```



Árvore Binária de Busca

Remoção (`delete`)

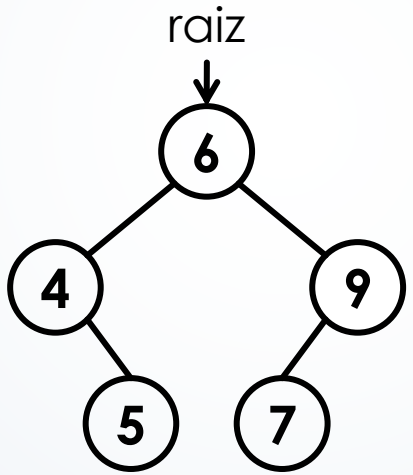
```
abb.delete( 3 );
```



Árvore Binária de Busca

Remoção (`delete`)

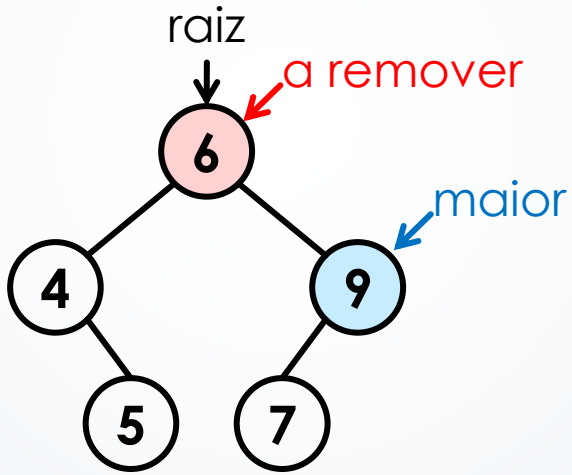
```
abb.delete( 3 );
```



Árvore Binária de Busca

Remoção (`delete`)

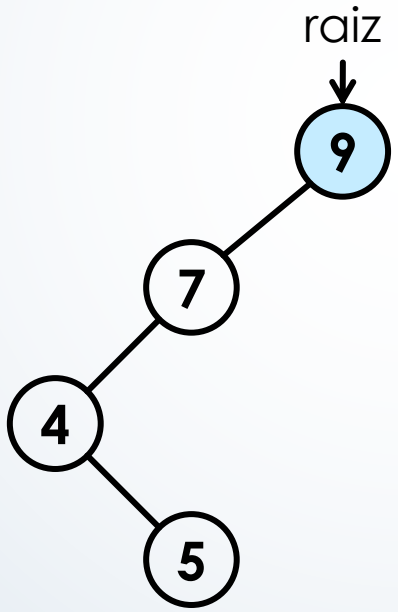
```
abb.delete( 6 );
```



Árvore Binária de Busca

Remoção (`delete`)

```
abb.delete( 6 );
```



Importante!

A política de remoção de nós, onde o maior nó ocupa a posição do nó excluído, faz com que, quando o nó removido possui filhos para os dois lados, a altura da árvore aumente! Esse algoritmo de remoção é denominado Hibbard Deletion.

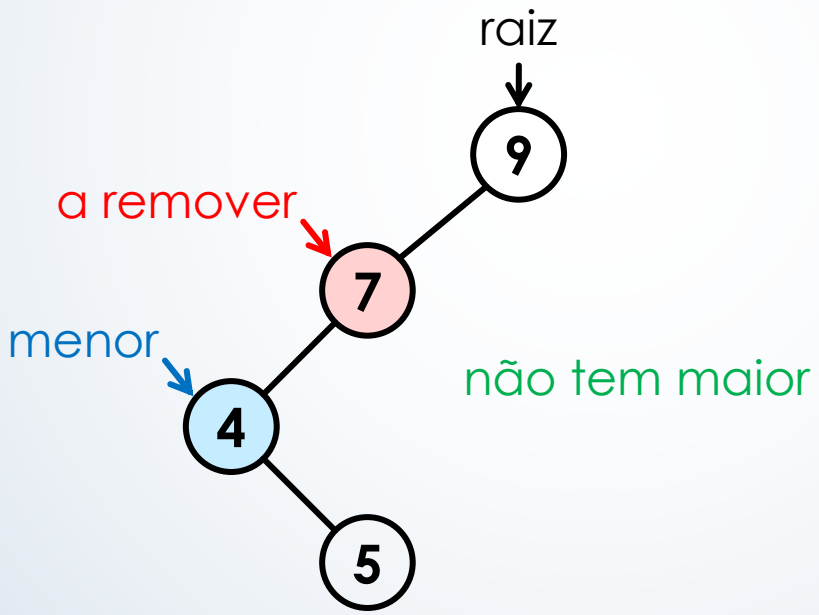
Para pensar:

- Isso é bom ou ruim?
- Há alguma situação análoga na inserção?

Árvore Binária de Busca

Remoção (`delete`)

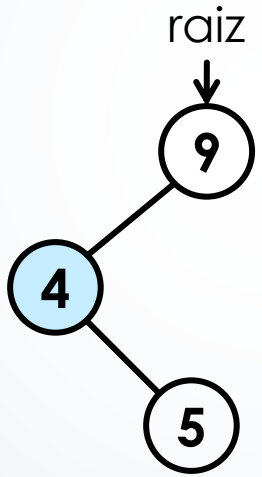
```
abb.delete( 7 );
```



Árvore Binária de Busca

Remoção (`delete`)

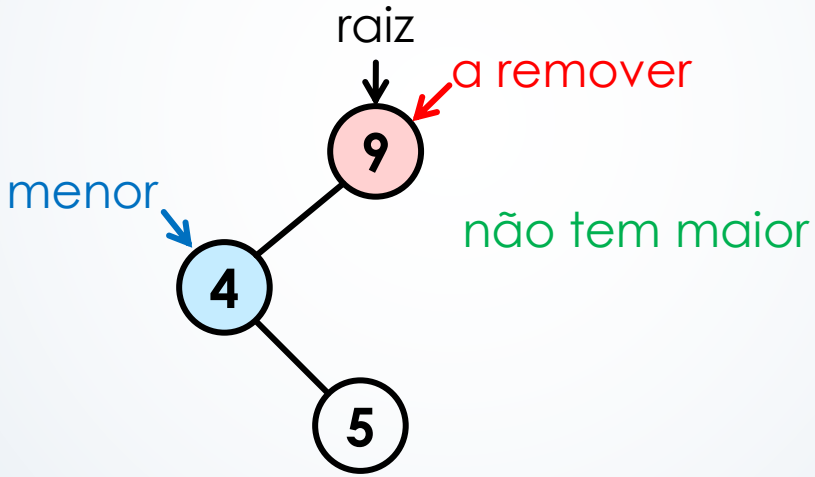
```
abb.delete( 7 );
```



Árvore Binária de Busca

Remoção (`delete`)

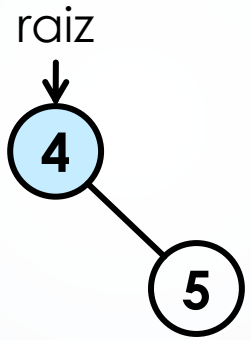
```
abb.delete( 9 );
```



Árvore Binária de Busca

Remoção (`delete`)

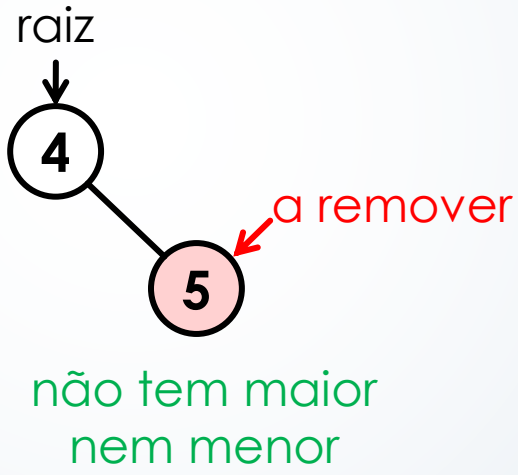
```
abb.delete( 9 );
```



Árvore Binária de Busca

Remoção (`delete`)

```
abb.delete( 5 );
```



Árvore Binária de Busca

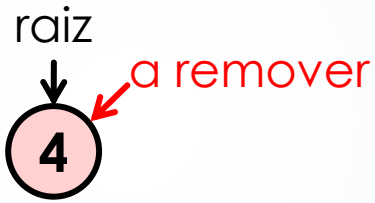
Remoção (`delete`)

```
abb.delete( 5 );
```

Árvore Binária de Busca

Remoção (`delete`)

```
abb.delete( 4 );
```



não tem maior
nem menor

Árvore Binária de Busca

Remoção (`delete`)

```
abb.delete( 4 );
```

raiz

Árvore Binária

Para pensar

- As árvores binárias de busca, da forma que foram apresentadas até agora, tem um grave defeito. Você é capaz de identificá-lo? Pense... se uma árvore binária de busca será usada como infraestrutura para permitir que uma tabela de símbolos seja implementada, queremos sempre que as operações realizadas, como inserção, exclusão e consulta sejam o mais rápidas possíveis. Dada a natureza da construção das árvores binárias de busca, qual seria então essa falha?
- Não conseguiu? Construa uma árvore binária de busca com os seguintes valores, nessa exata ordem: 1, 2, 3, 4, 5, 6 e 7.
- O que você nota?
- Extrapole a ideia para a construção com mil elementos em ordem. Qual a ordem de crescimento da inserção, remoção e consulta no pior caso?

Árvore Binária

Percursos

- **Percurso:** forma de percorrer uma estrutura de dados, no caso, uma árvore binária;
- Principais tipos de percurso em árvores binárias:
  1. **Pré-ordem:** primeiramente processa-se a raiz, depois percorre-se a subárvore esquerda e então percorre-se a subárvore direita. Chamado de ***preorder traversal*** em inglês;
  2. **Em ordem ou ordem simétrica:** percorre-se primeiramente a subárvore esquerda, ao terminar processa-se a raiz e então percorre-se a subárvore direita. Chamado de ***inorder traversal*** em inglês;
  3. **Pós-ordem ou ordem final:** percorre-se primeiramente a subárvore esquerda, depois percorre-se a subárvore direita e então processa-se a raiz. Chamado de ***postorder traversal*** em inglês;
  4. **Em nível:** iniciando no nível zero, progride-se nível a nível, processando cada raiz, da esquerda para a direita. Chamado de ***level-order traversal*** em inglês;

Árvore Binária

Percursos

- **Observações:**
  - Os percursos pré-ordem, em ordem e pós-ordem são análogos à Busca em Profundidade (***Depth-First Search***) em grafos;
  - O percurso em nível é análogo à Busca em Largura (***Breadth-First Search***) em grafos;
- Aplicações:
  1. **Pré-ordem:** cópia de uma árvore binária de busca, criação de hierarquias, como índices em documentos de texto, ordenação topológica (grafos), obtenção de expressão pré-fixada em uma árvore de expressão etc;
  2. **Em ordem:** processa os nós de forma crescente, podendo ser empregado, por exemplo, em ordenações, obtenção de expressão infixada em uma árvore de expressão etc;
  3. **Pós-ordem:** ordenação topológica (grafos), obtenção de expressão pós-fixada em uma árvore de expressão etc;
  4. **Em nível:** obtenção da distância entre nós.

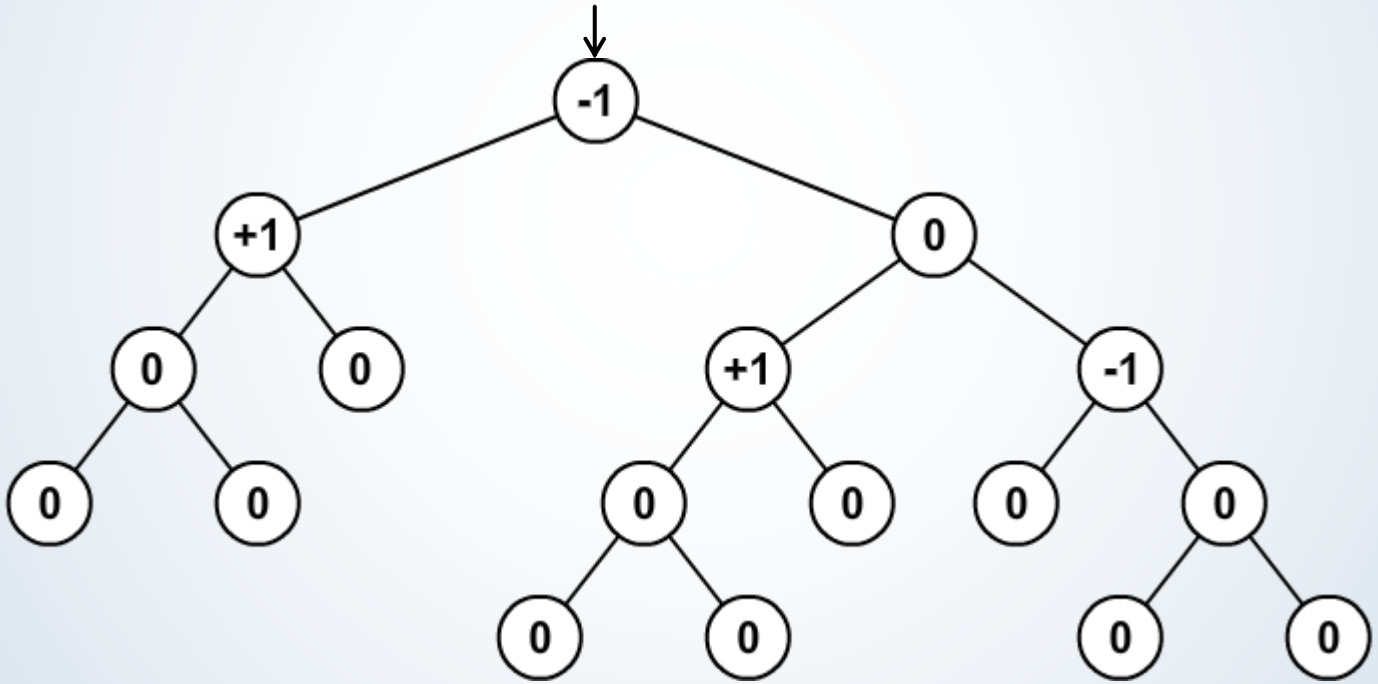
Árvores AVL

- São árvores binárias de busca balanceadas propostas em 1962 pelos matemáticos soviéticos Georgy Maximovich **A**delson-**V**elsky e Evgenii Mikhailovich **L**andis;
- A altura balanceada implica em operações com ordem de crescimento proporcionais a $O(\lg n)$.

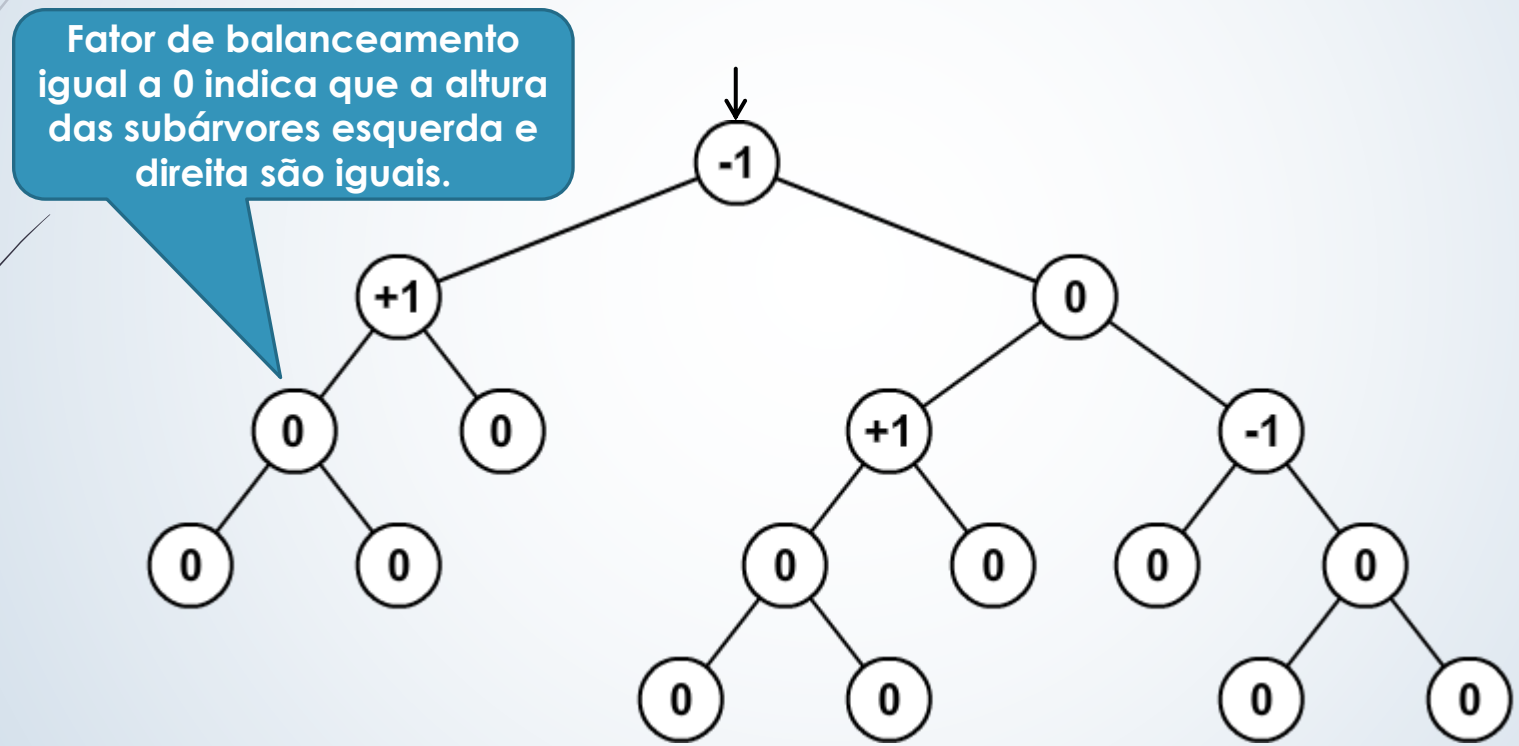
Árvores AVL
Fator de Balanceamento

- Dado um nó **T**, seu fator de balanceamento é dado por $h_e - h_d$;
- Os fatores de balanceamento aceitáveis para que **T** seja um nó de uma árvore AVL são -1, 0 ou +1;
- Se **T** for um nó folha, seu fator de balanceamento é 0.

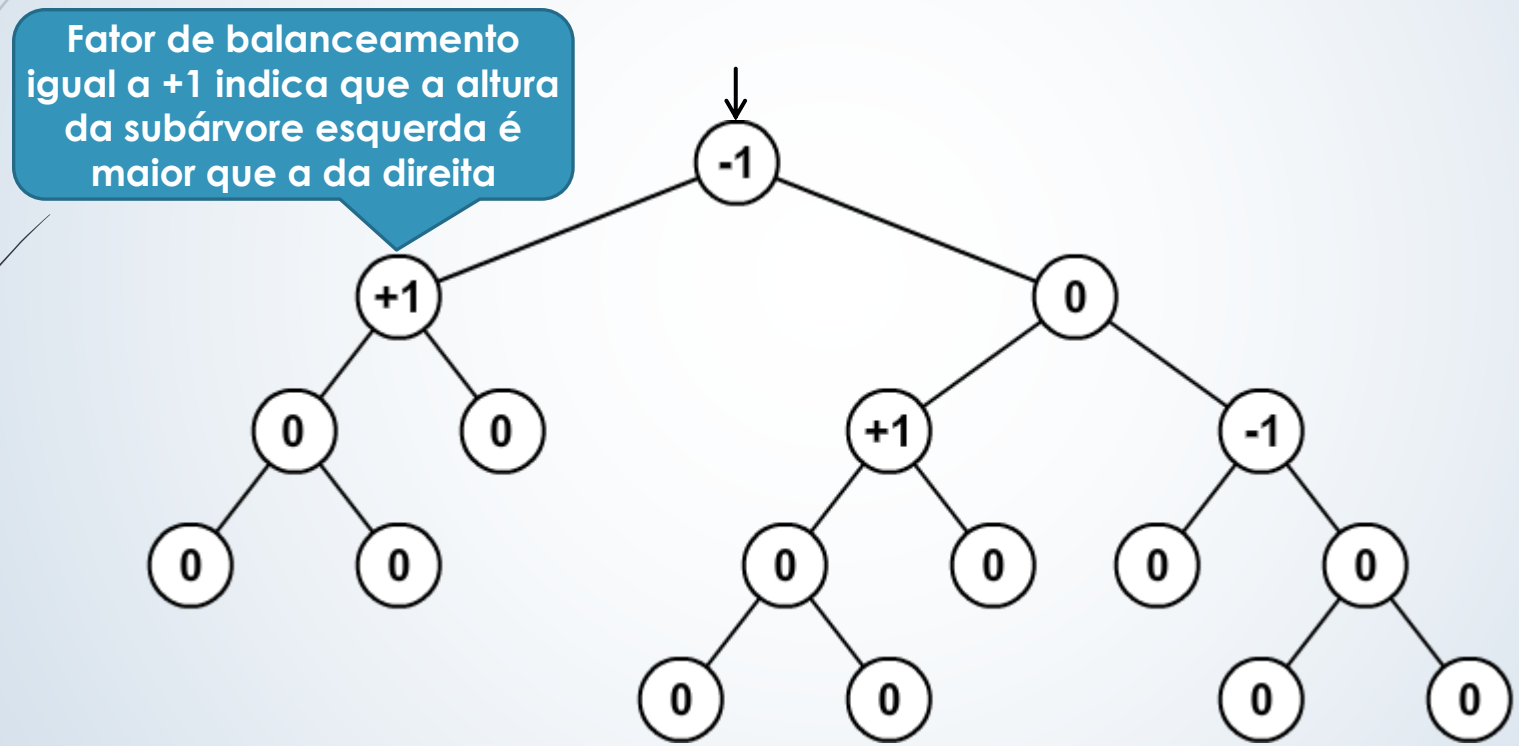
Árvores AVL
Fator de Balanceamento



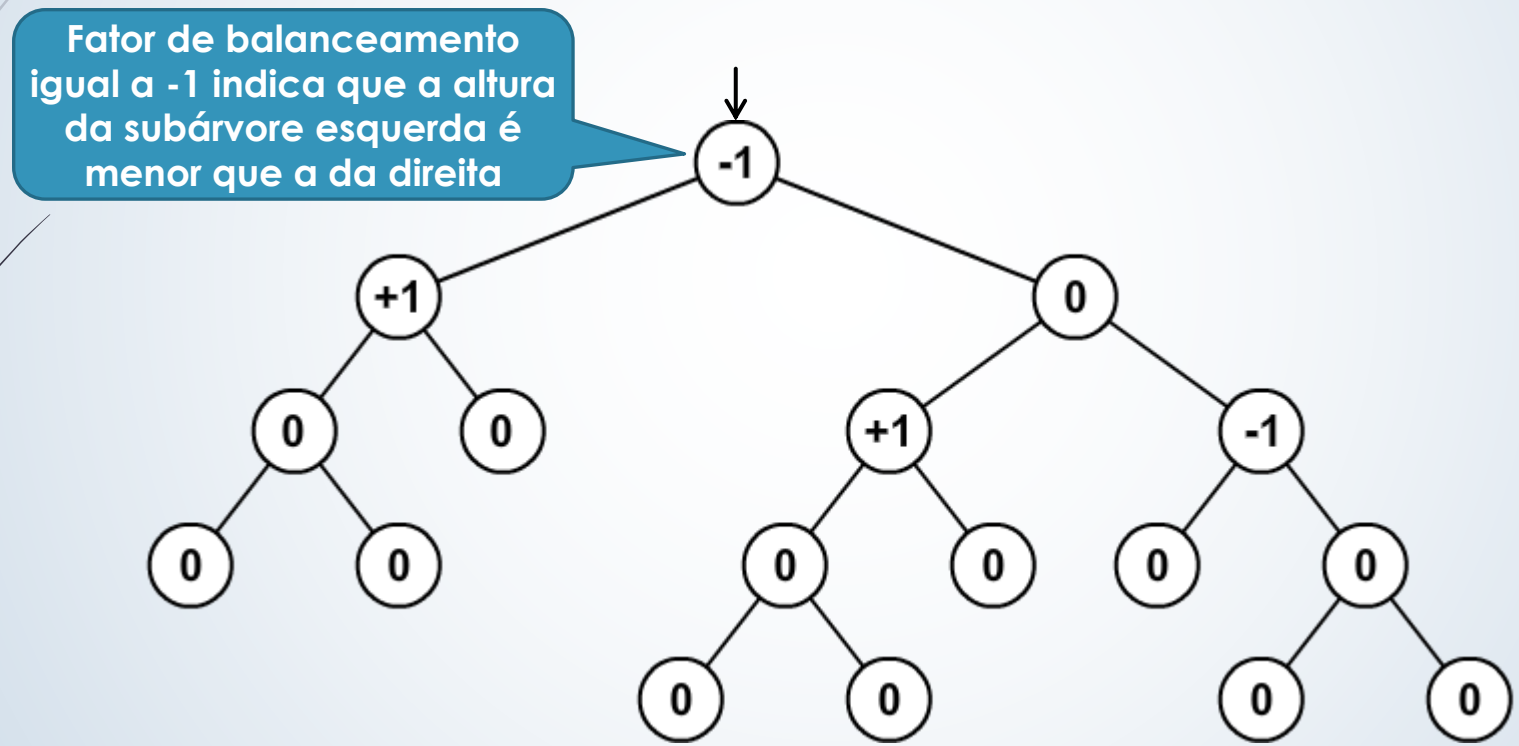
Árvores AVL
Fator de Balanceamento



Árvores AVL
Fator de Balanceamento



Árvores AVL
Fator de Balanceamento



Árvores AVL
Rotações

- O rebalanceamento utiliza quatro tipos de rotações: EE, DD, ED e DE
  - EE e DD são simétricas entre si;
  - ED e DE são simétricas entre si;
- As rotações tomam como base o fator de balanceamento do nó **A**, o ancestral mais próximo de **N**, o novo nó inserido. Para haver uma rotação o fator de balanceamento de **A** deve ser +2 ou -2.

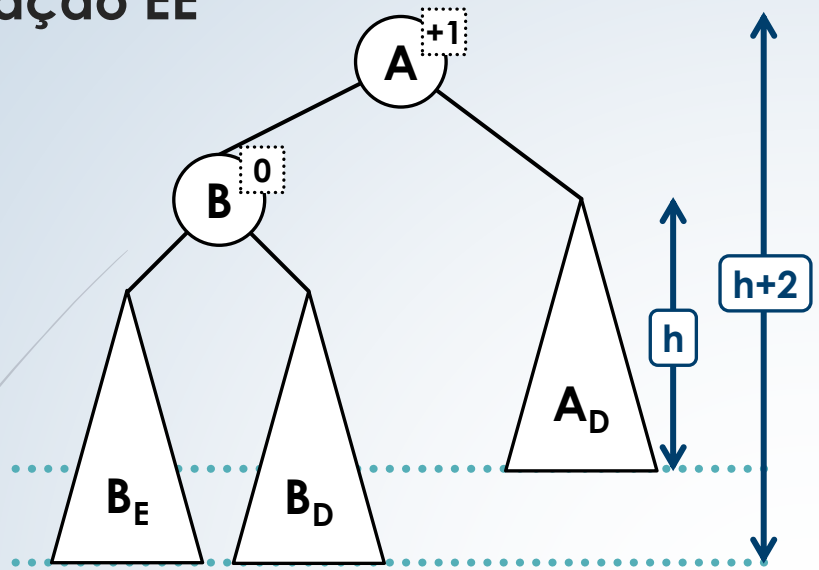
Árvores AVL
Rotações

- **EE: N** foi inserido na subárvore **esquerda** da subárvore **esquerda** de **A**;
- **ED: N** foi inserido na subárvore **direita** da subárvore **esquerda** de **A**;
- **DD: N** foi inserido na subárvore **direita** da subárvore **direita** de **A**;
- **DE: N** foi inserido na subárvore **esquerda** da subárvore **direita** de **A**;

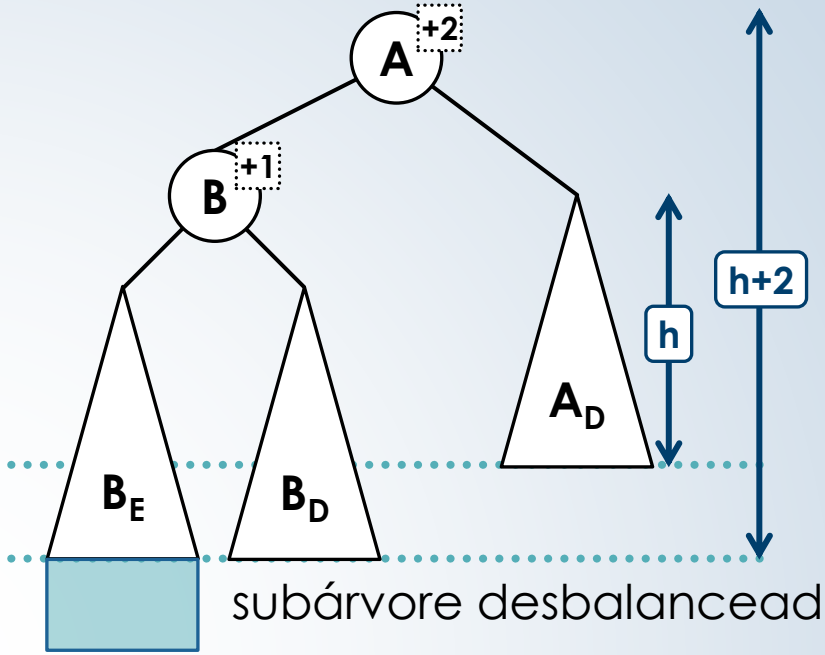
Árvores AVL
Rotações

- Se **B** é o filho de **A** aonde ocorreu a inserção de **N**:
  - **EE: A = +2; B = +1;**
  - **ED: A = +2; B = -1;**
  - **DD: A = -2; B = -1;**
  - **DE: A = -2; B = +1;**
- **C** é o filho de **B** aonde ocorreu a inserção de **N**.

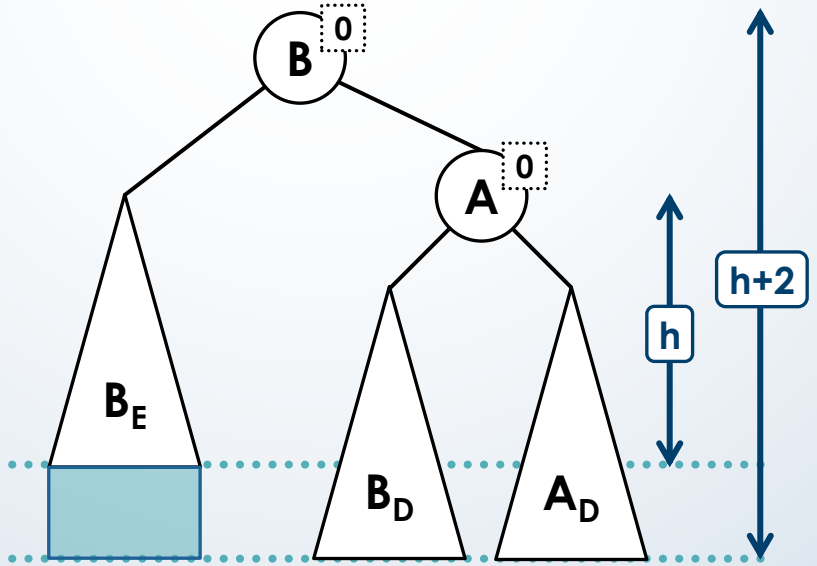
Rotação EE



subárvore balanceada

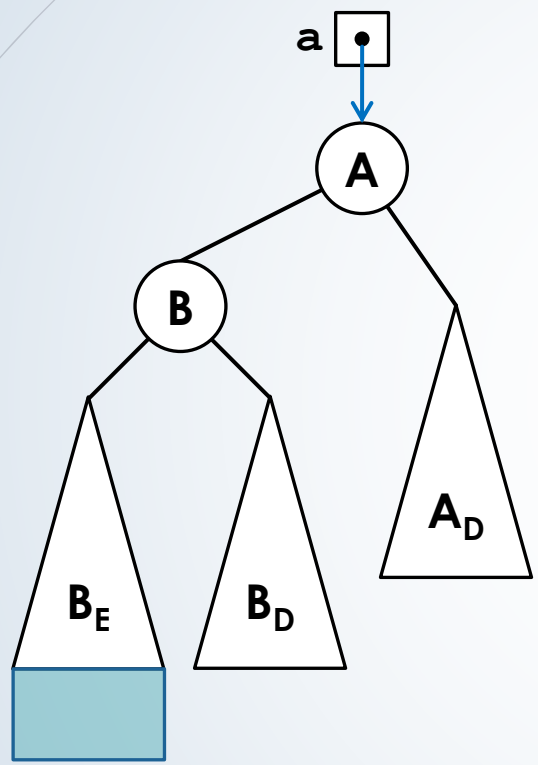


subárvore desbalanceada



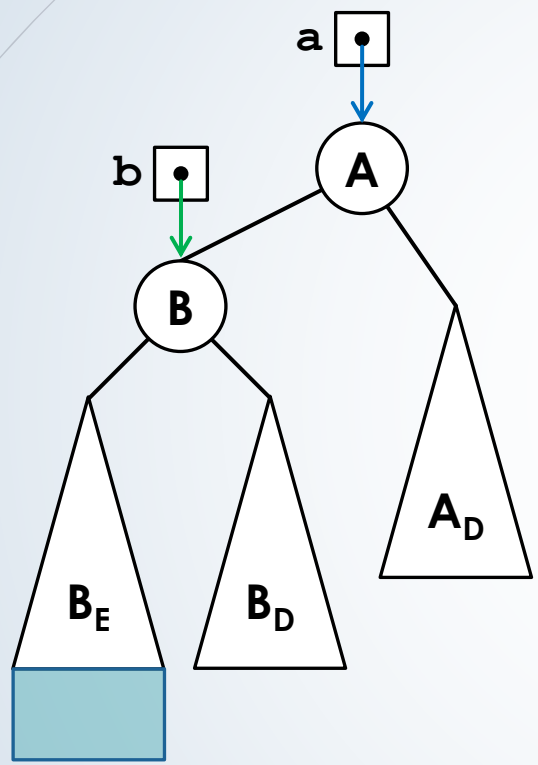
subárvore rebalanceada

Rotação EE



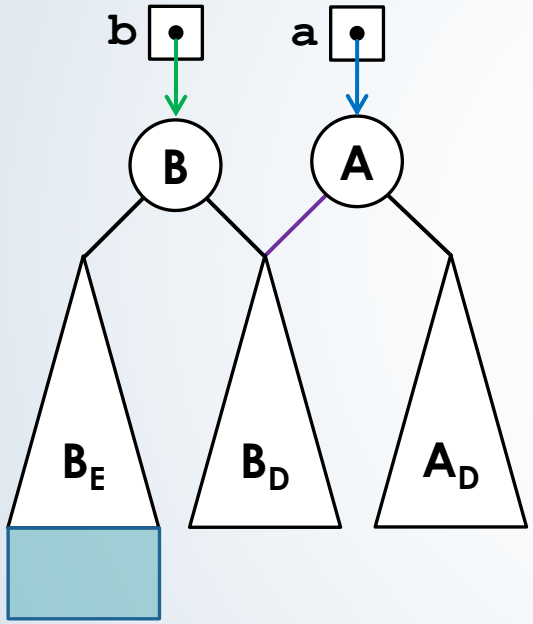

```
private No<Tipo> ee( No<Tipo> a ) {
    No<Tipo> b = a.esquerda;
    a.esquerda = b.direita;
    b.direita = a;
    return b;
}
```

Rotação EE



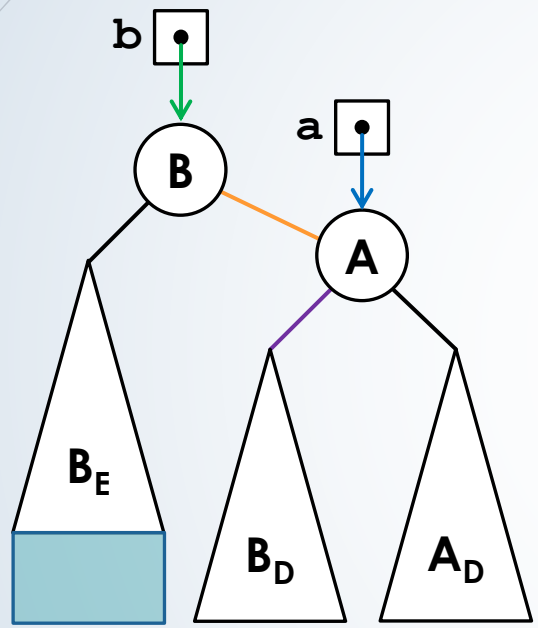

```
private No<Tipo> ee( No<Tipo> a ) {
    No<Tipo> b = a.esquerda;
    a.esquerda = b.direita;
    b.direita = a;
    return b;
}
```

Rotação EE



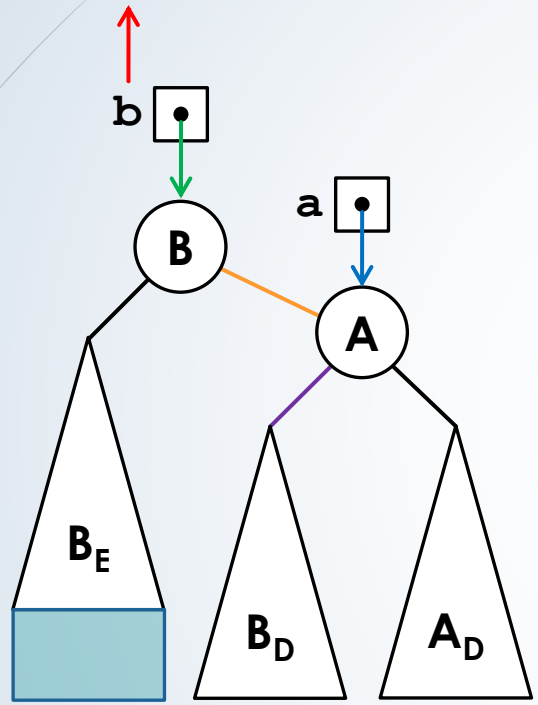

```
private No<Tipo> ee( No<Tipo> a ) {
    No<Tipo> b = a.esquerda;
    a.esquerda = b.direita;
    b.direita = a;
    return b;
}
```

Rotação EE



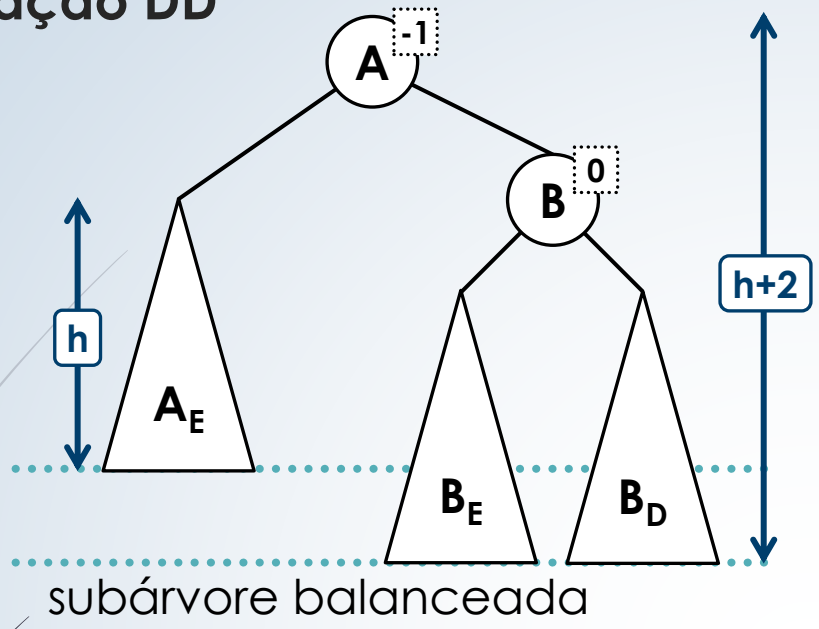

```
private No<Tipo> ee( No<Tipo> a ) {
    No<Tipo> b = a.esquerda;
    a.esquerda = b.direita;
    b.direita = a;
    return b;
}
```

Rotação EE

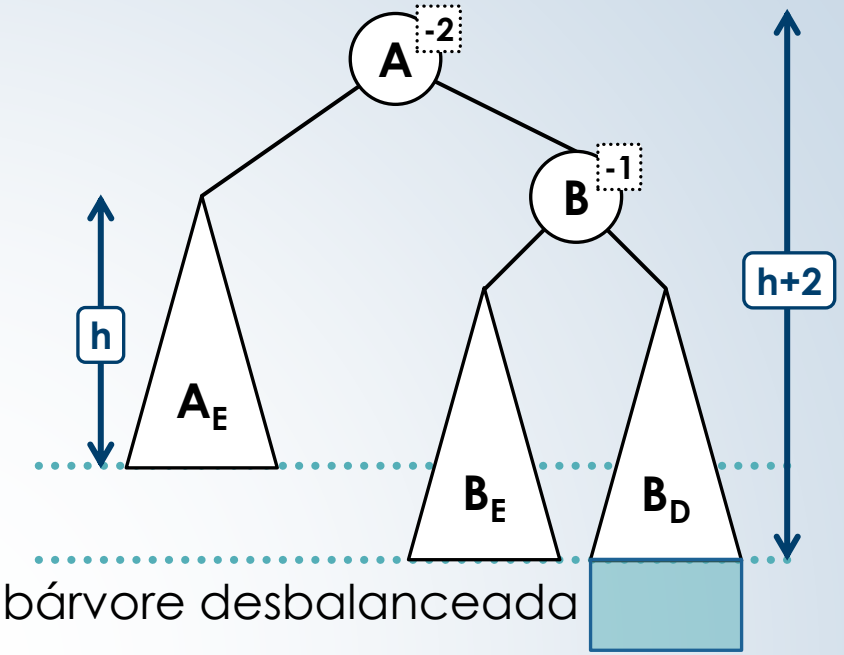



```
private No<Tipo> ee( No<Tipo> a ) {
    No<Tipo> b = a.esquerda;
    a.esquerda = b.direita;
    b.direita = a;
    return b;
}
```

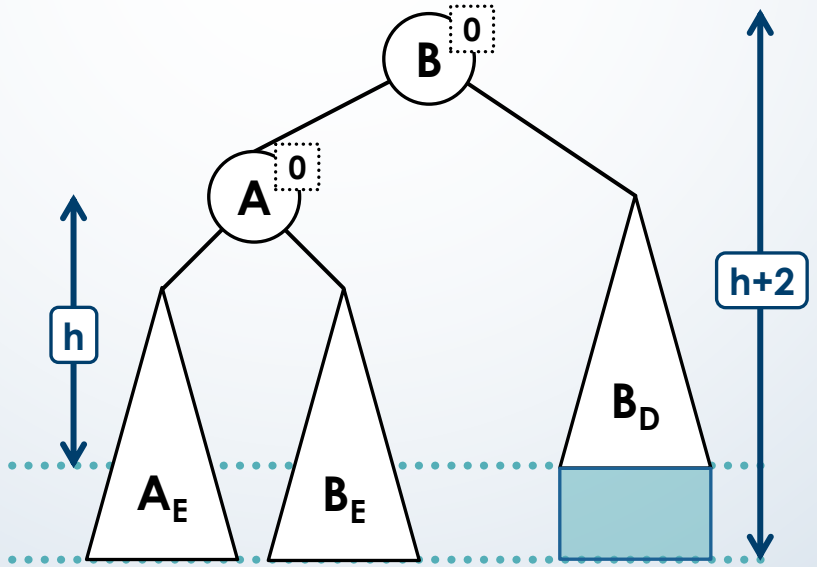
Rotação DD



subárvore balanceada

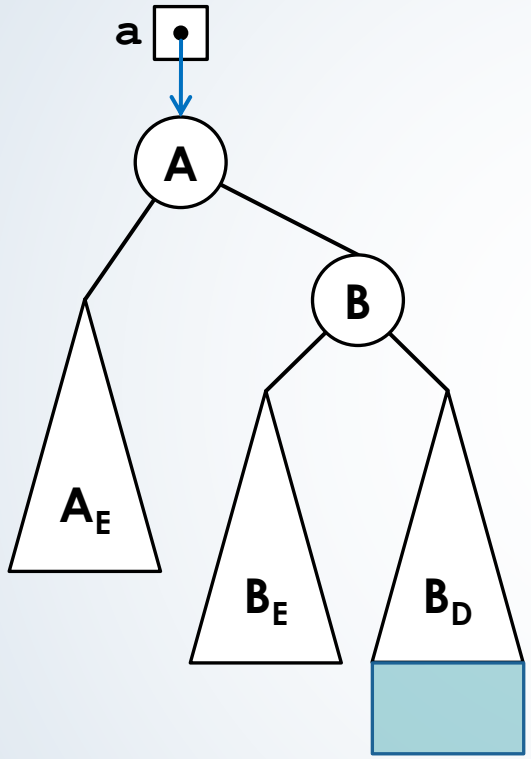


subárvore desbalanceada



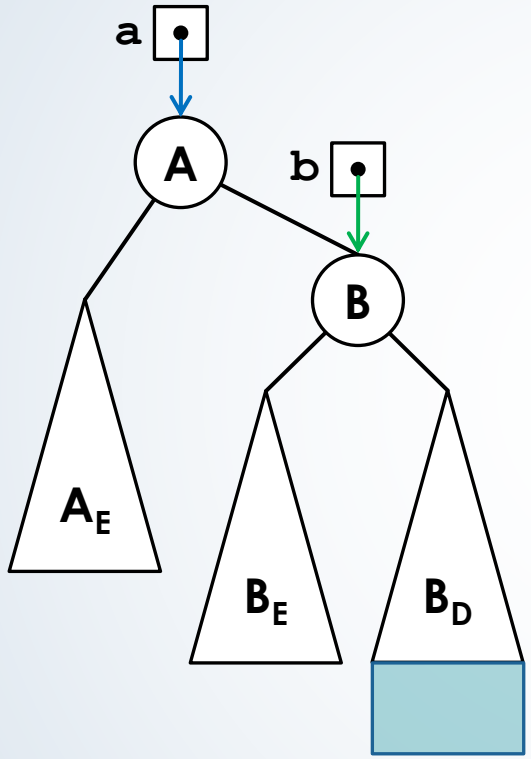
subárvore rebalanceada

Rotação DD



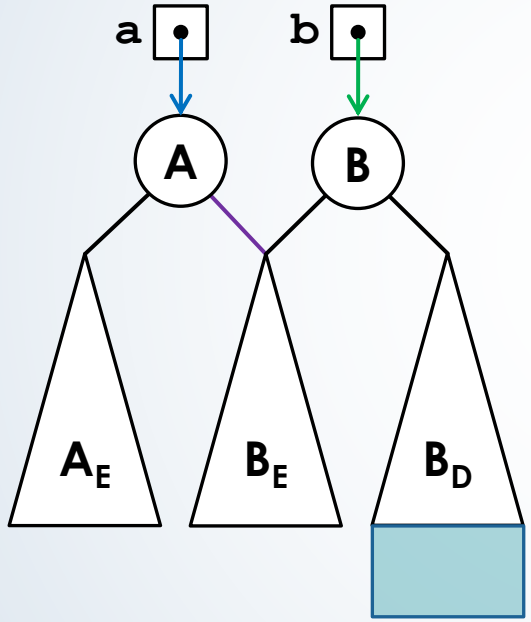

```
private No<Tipo> dd( No<Tipo> a ) {
    No<Tipo> b = a.direita;
    a.direita = b.esquerda;
    b.esquerda = a;
    return b;
}
```

Rotação DD



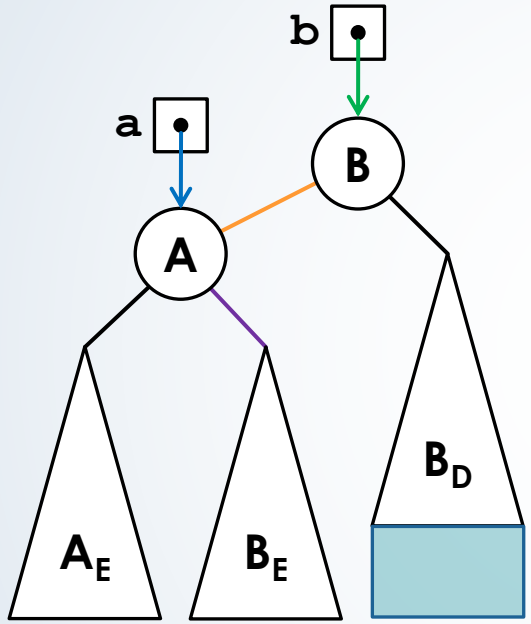

```
private No<Tipo> dd( No<Tipo> a ) {
    No<Tipo> b = a.direita;
    a.direita = b.esquerda;
    b.esquerda = a;
    return b;
}
```

Rotação DD



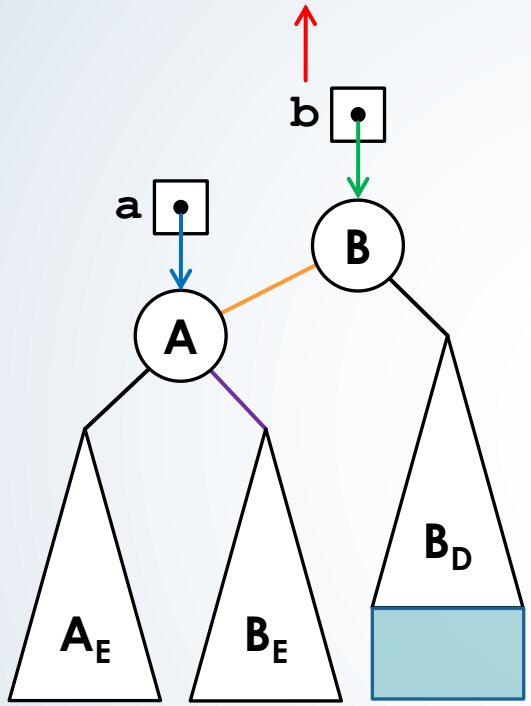

```
private No<Tipo> dd( No<Tipo> a ) {
    No<Tipo> b = a.direita;
    a.direita = b.esquerda;
    b.esquerda = a;
    return b;
}
```

Rotação DD



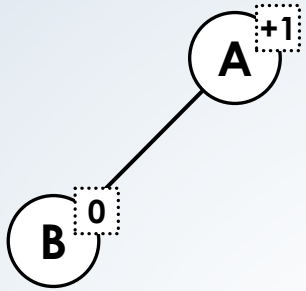

```
private No<Tipo> dd( No<Tipo> a ) {
    No<Tipo> b = a.direita;
    a.direita = b.esquerda;
    b.esquerda = a;
    return b;
}
```

Rotação DD

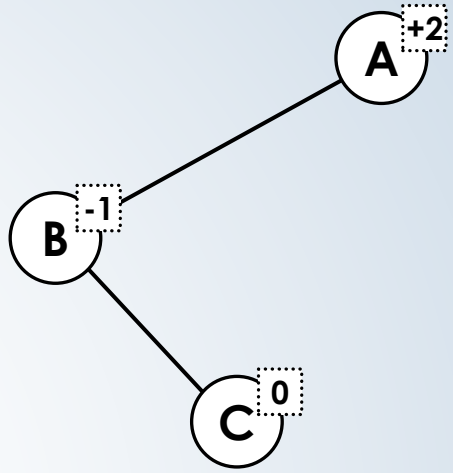



```
private No<Tipo> dd( No<Tipo> a ) {
    No<Tipo> b = a.direita;
    a.direita = b.esquerda;
    b.esquerda = a;
    return b;
}
```

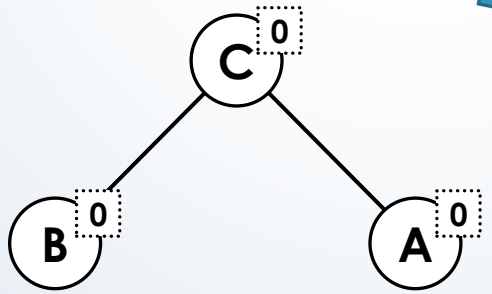
Rotação ED(a)



subárvore balanceada

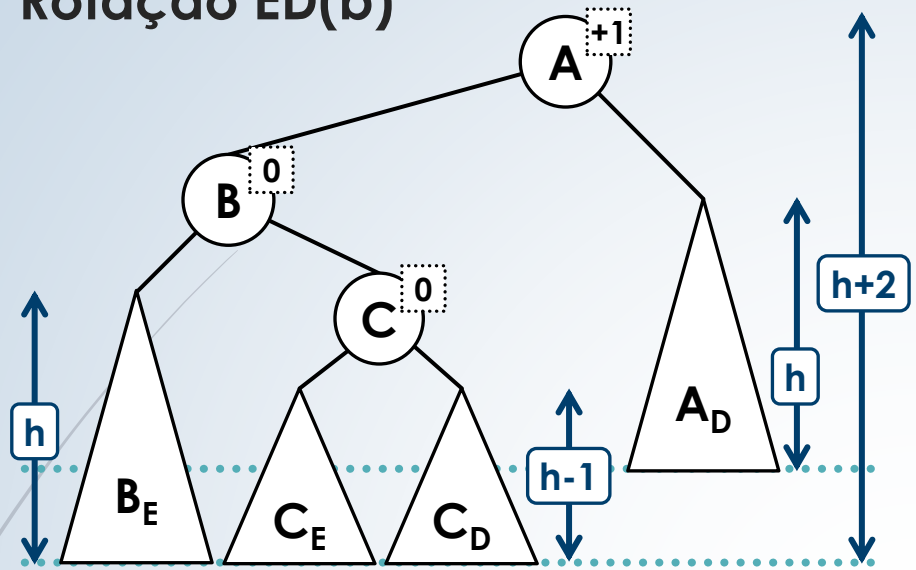


subárvore desbalanceada

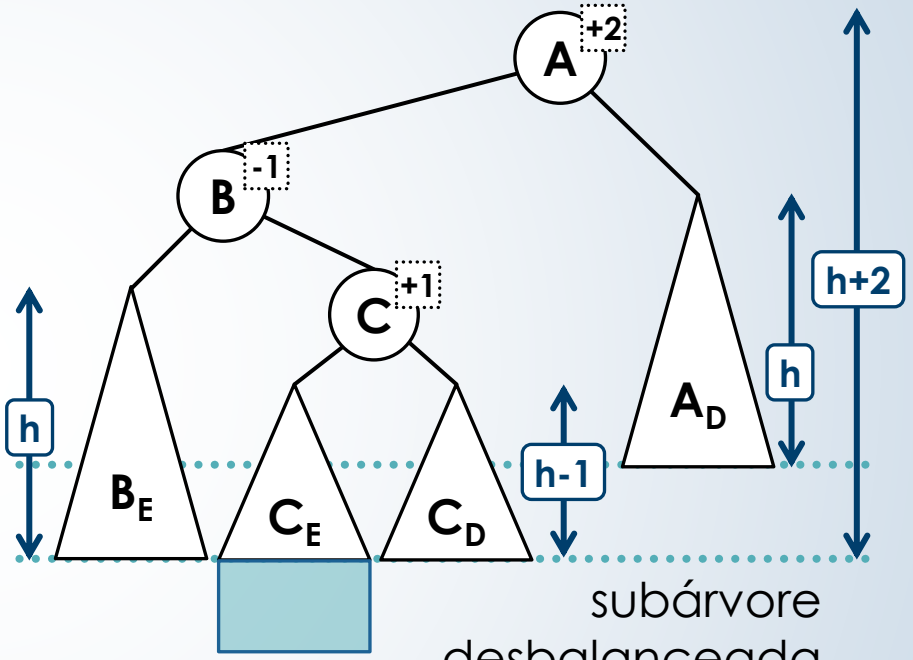


subárvore rebalanceada

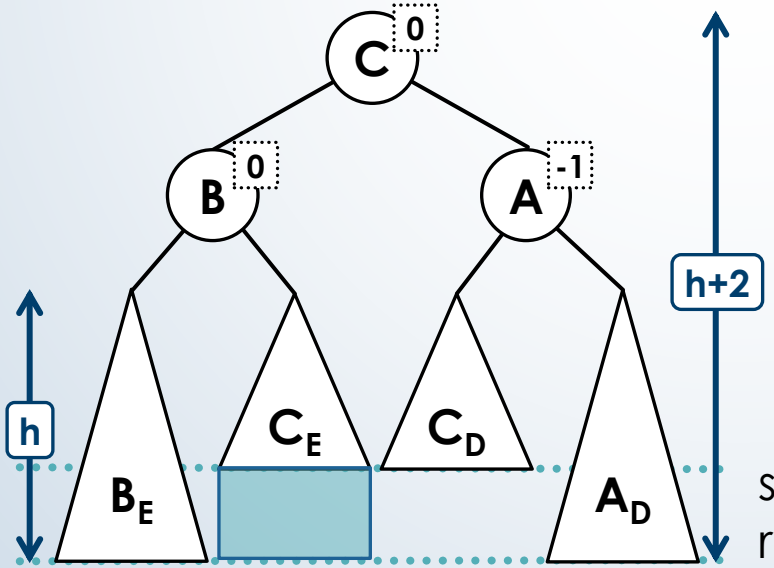
Rotação ED(b)



subárvore balanceada

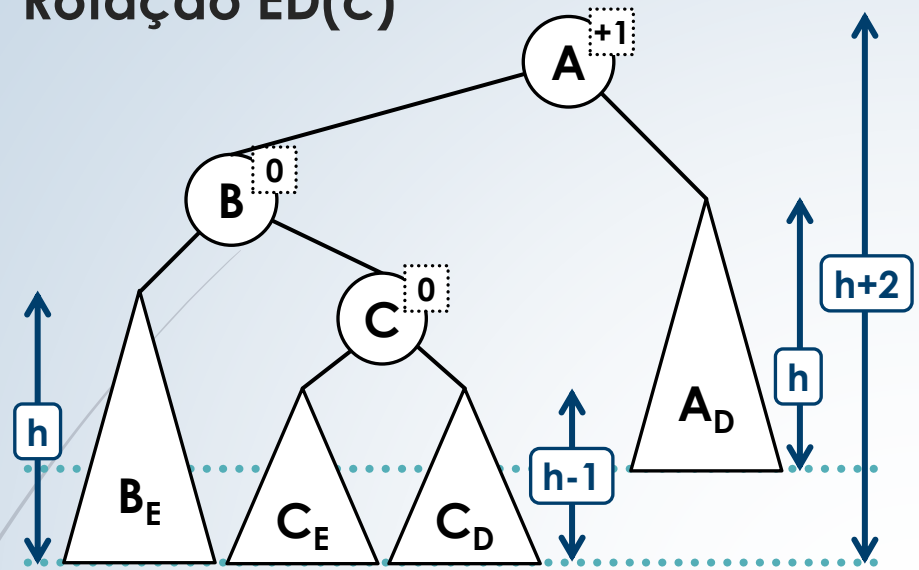


subárvore desbalanceada

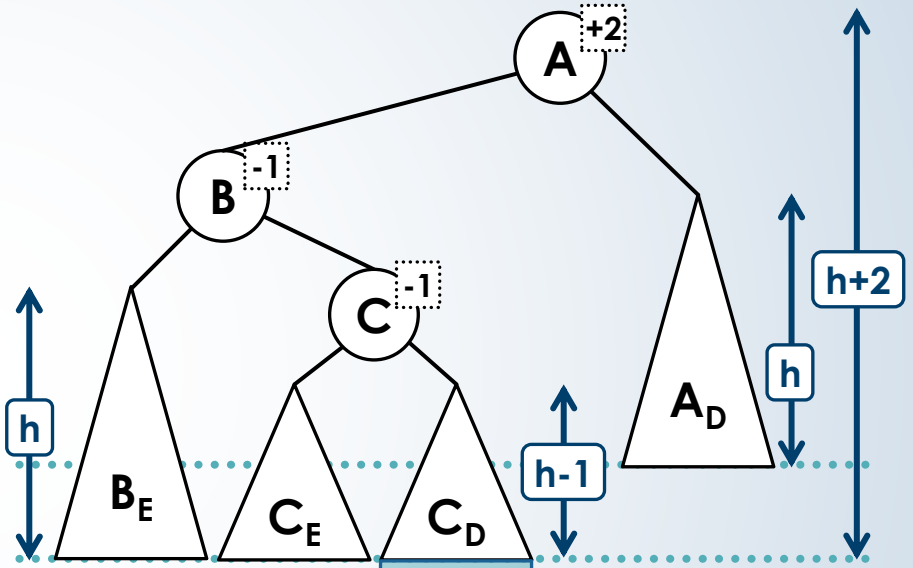


subárvore rebalanceada

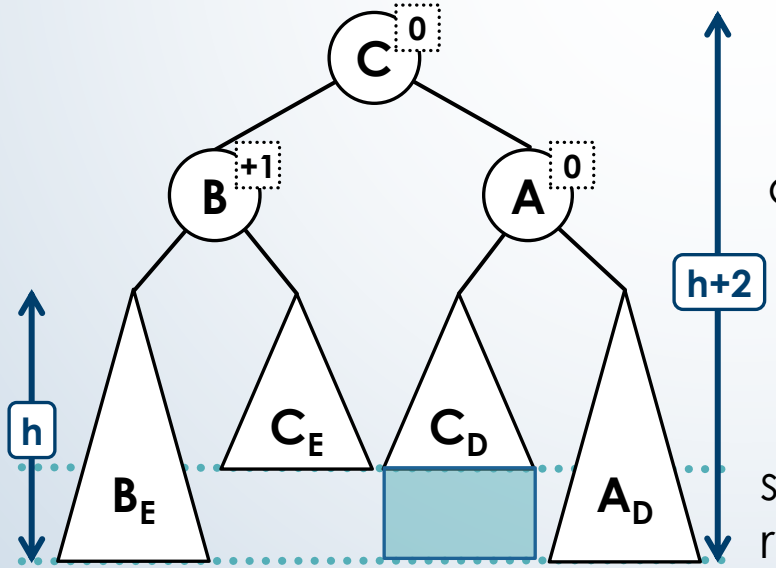
Rotação ED(c)



subárvore balanceada

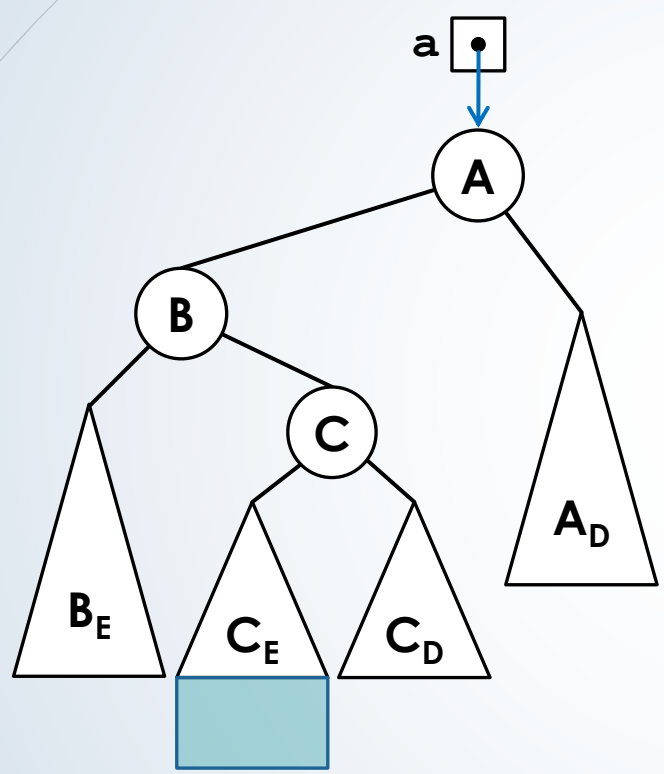


subárvore desbalanceada



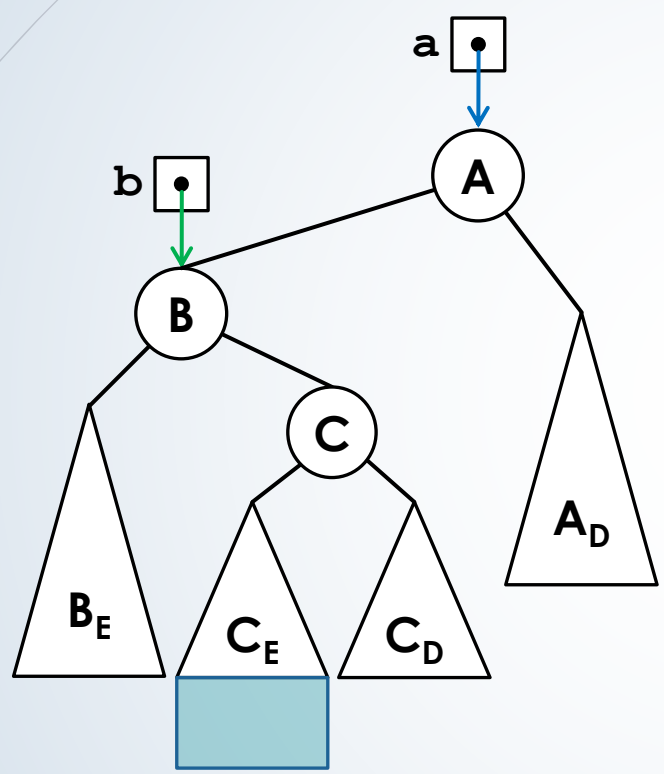
subárvore rebalanceada

Rotação ED



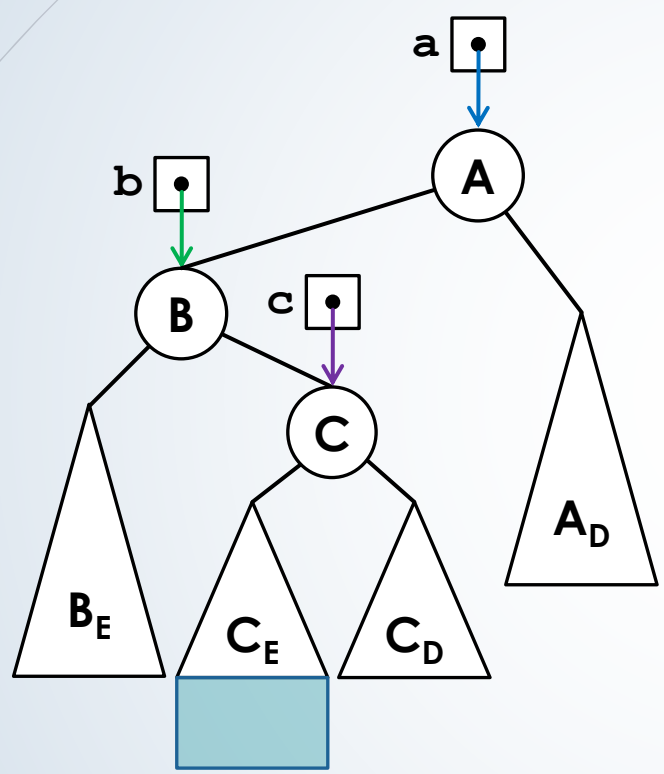

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



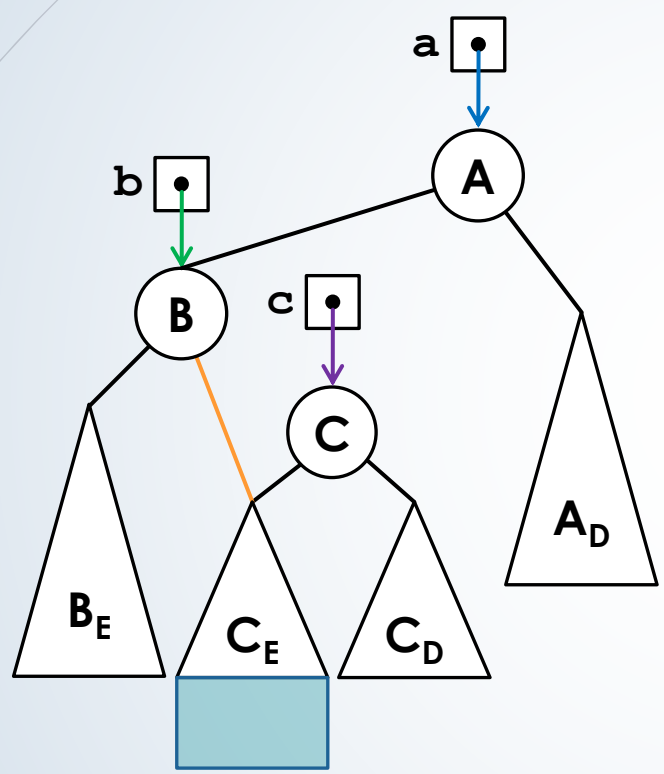

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



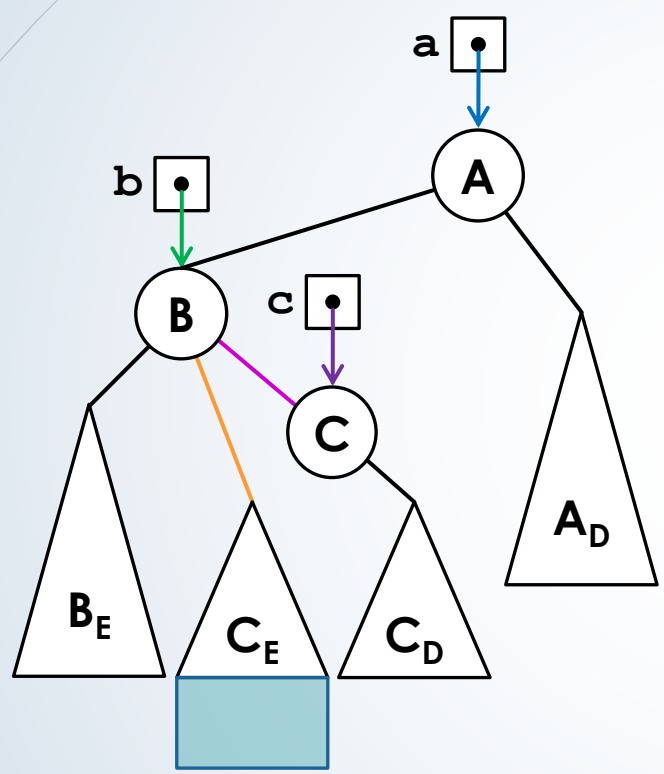

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



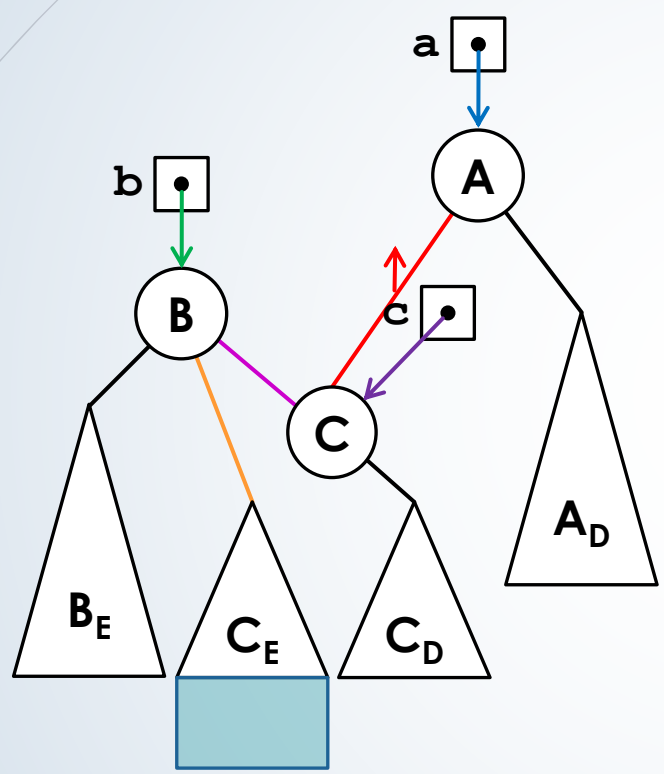

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



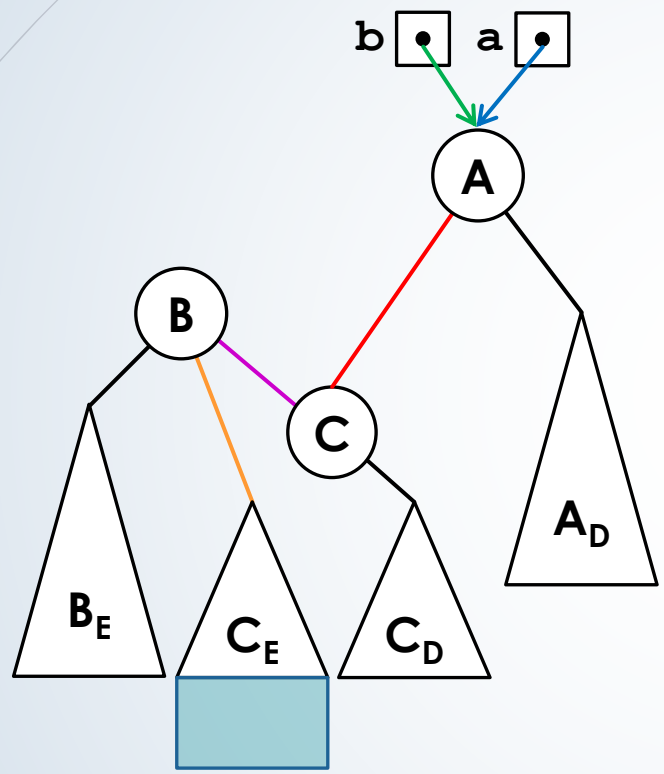

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



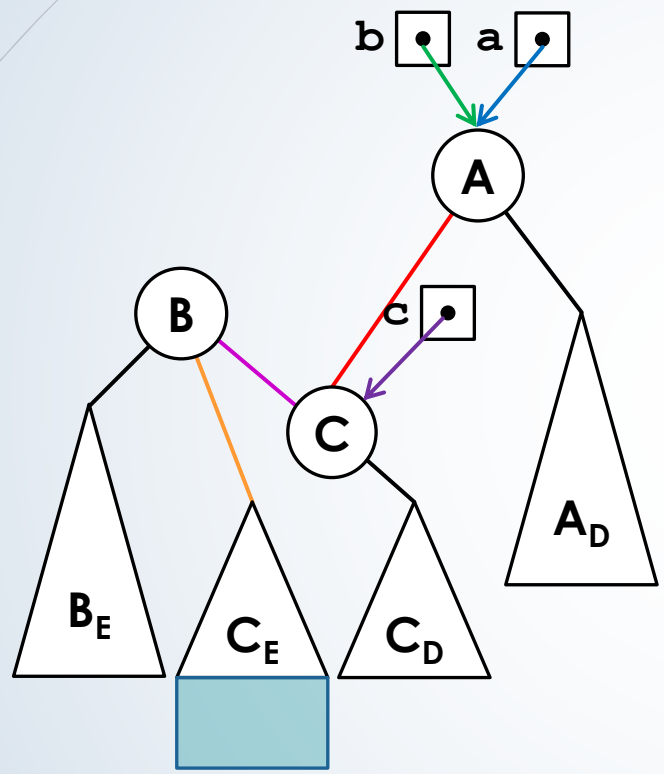

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



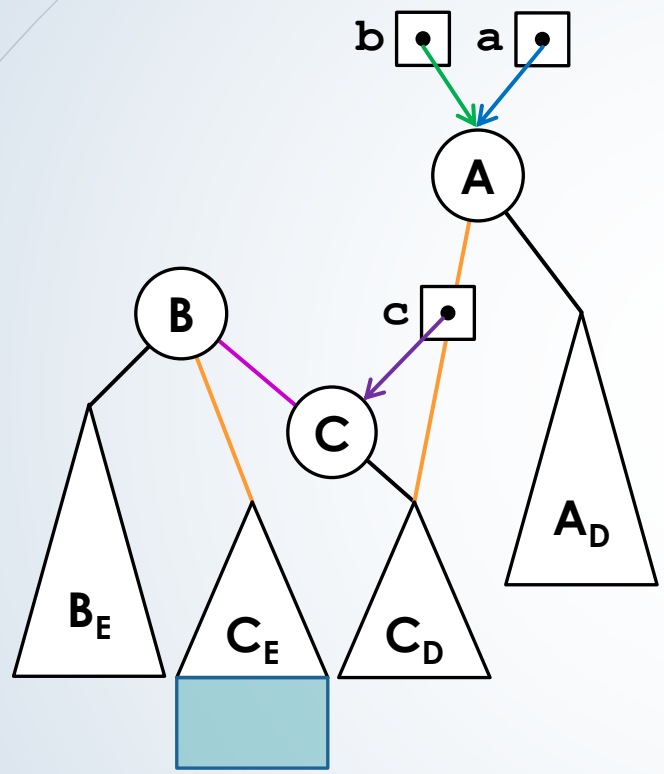

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



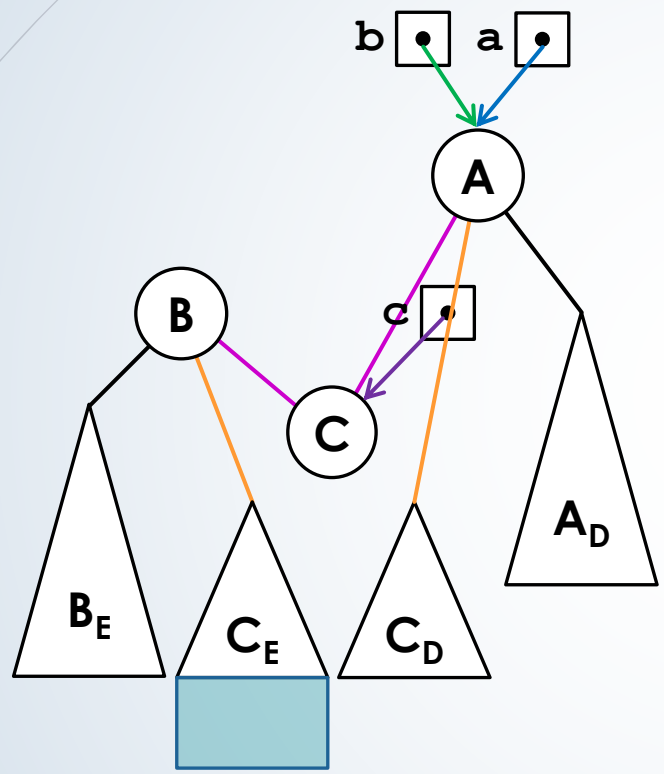

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



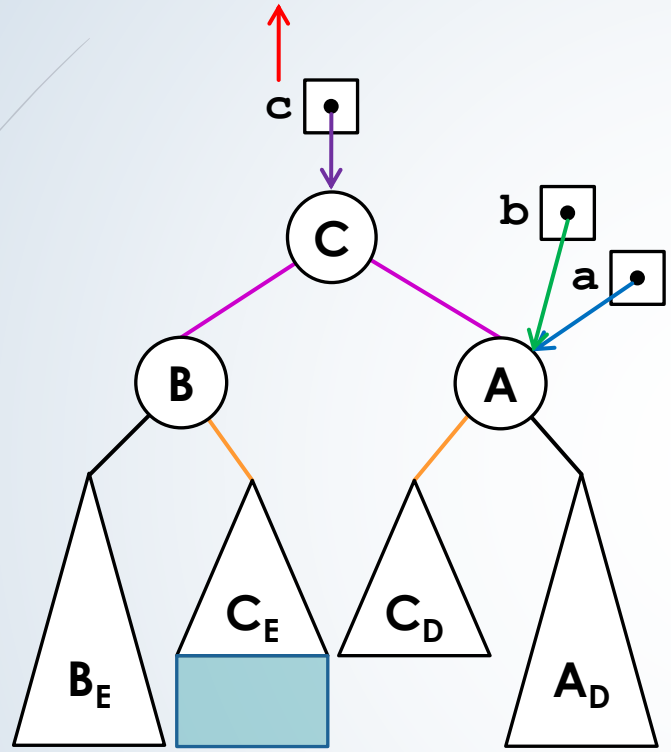

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED



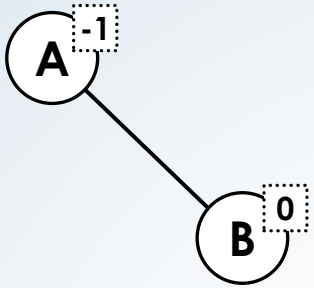

```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

Rotação ED

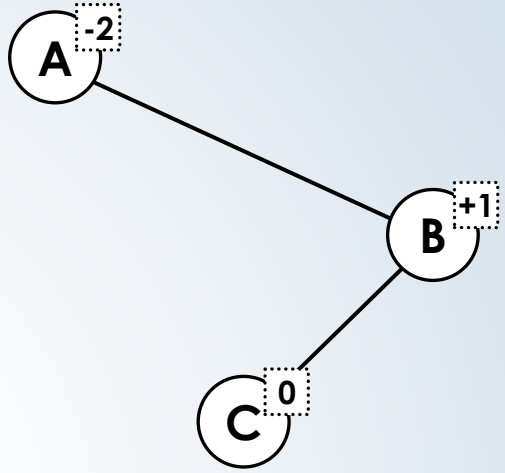



```
private No<Tipo> ed( No<Tipo> a ) {
    a.esquerda = dd( a.esquerda );
    return ee( a );
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}
```

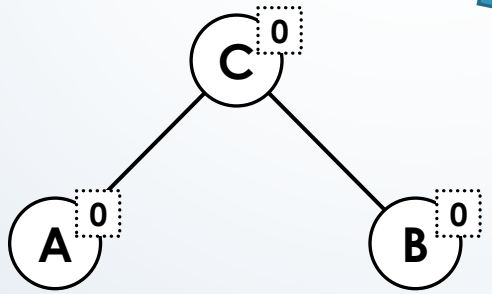
Rotação DE(a)



subárvore balanceada

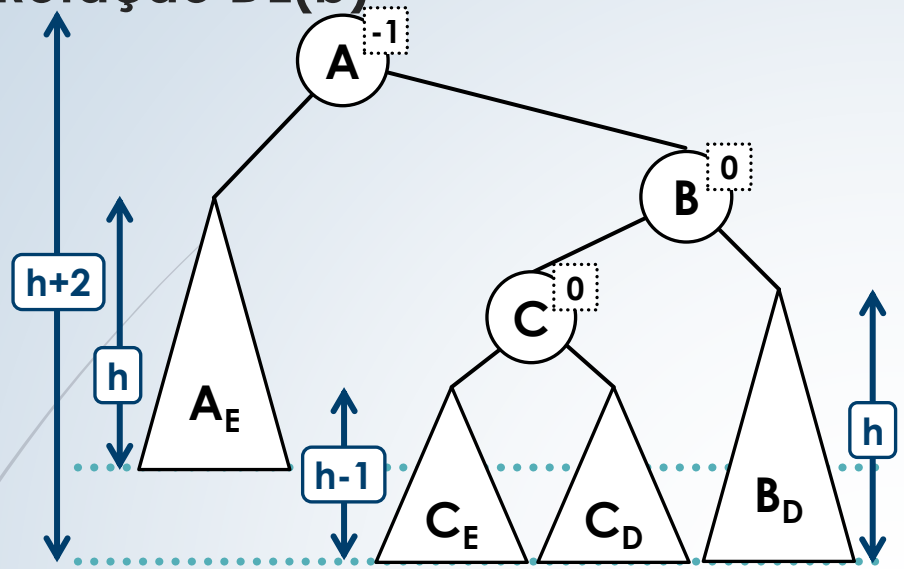


subárvore desbalanceada

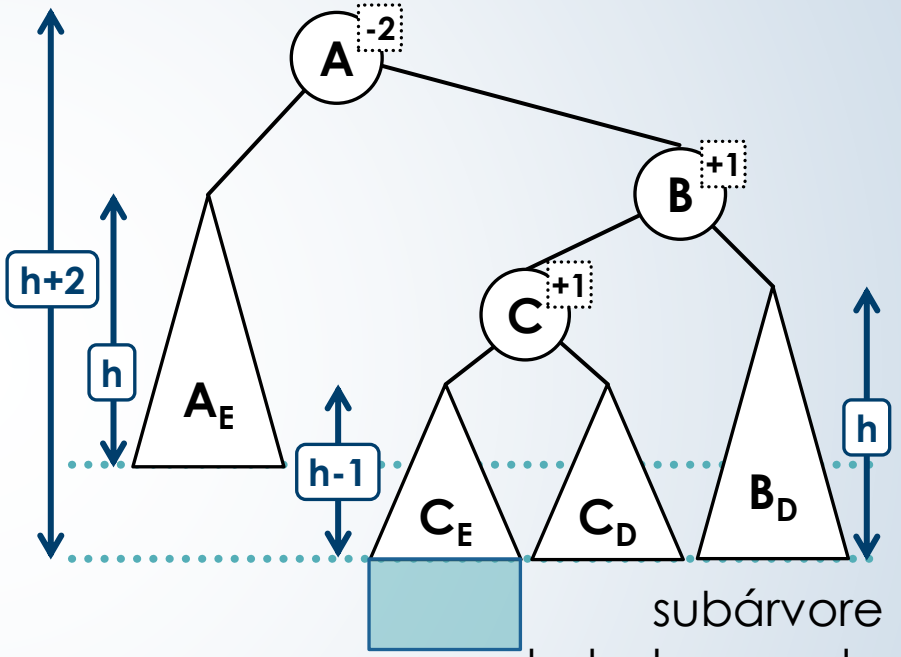


subárvore rebalanceada

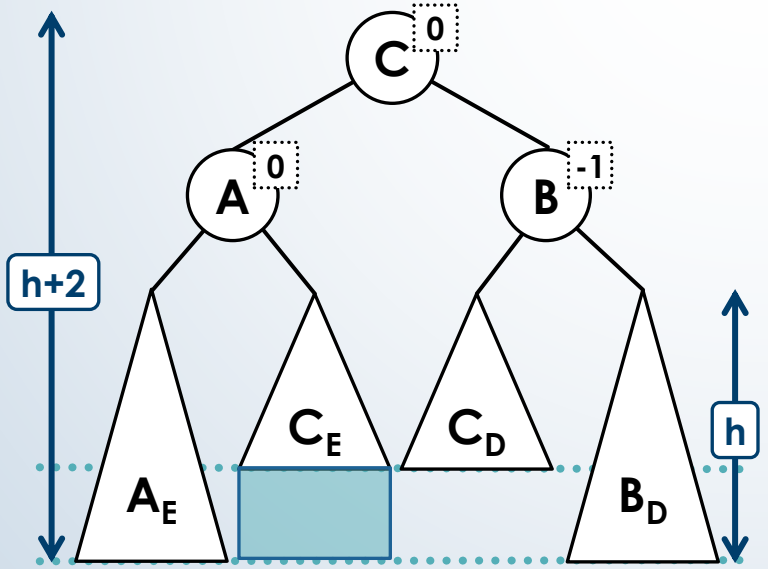
Rotação DE(b)



subárvore balanceada

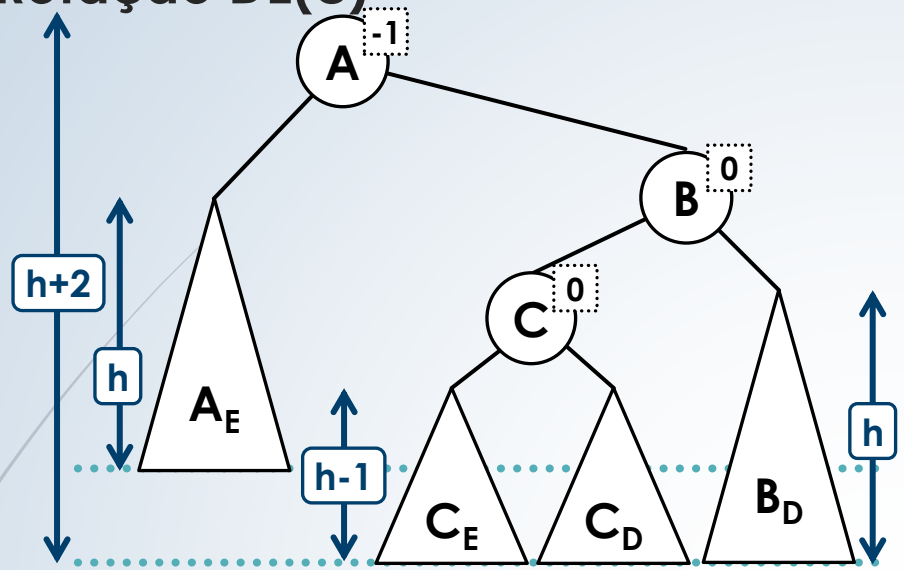


subárvore desbalanceada

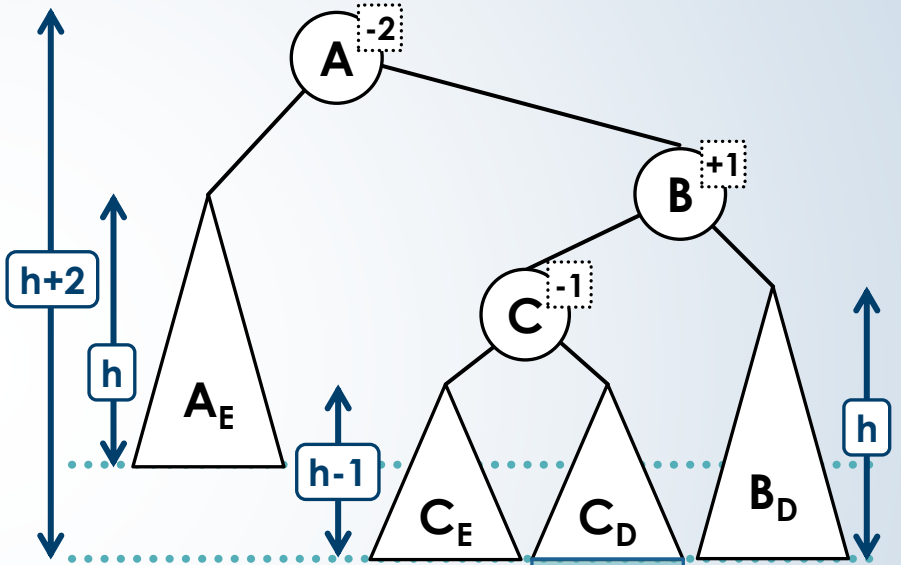


subárvore rebalanceada

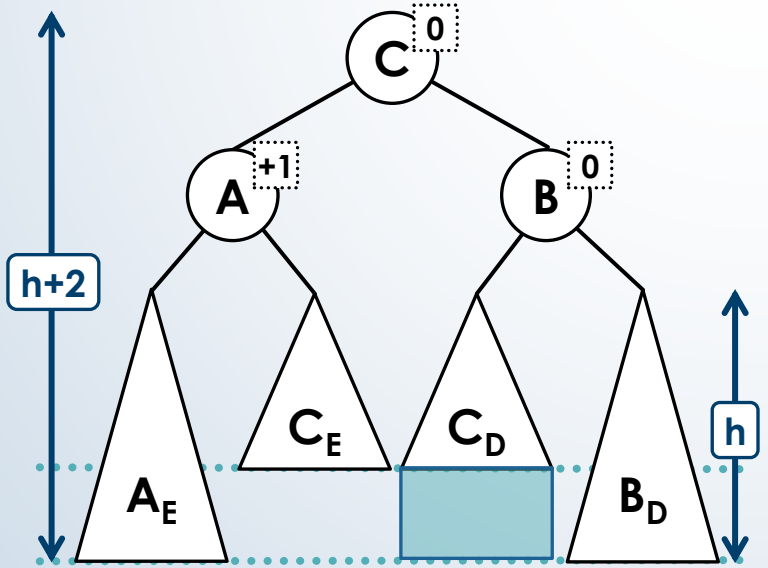
Rotação DE(c)



subárvore balanceada

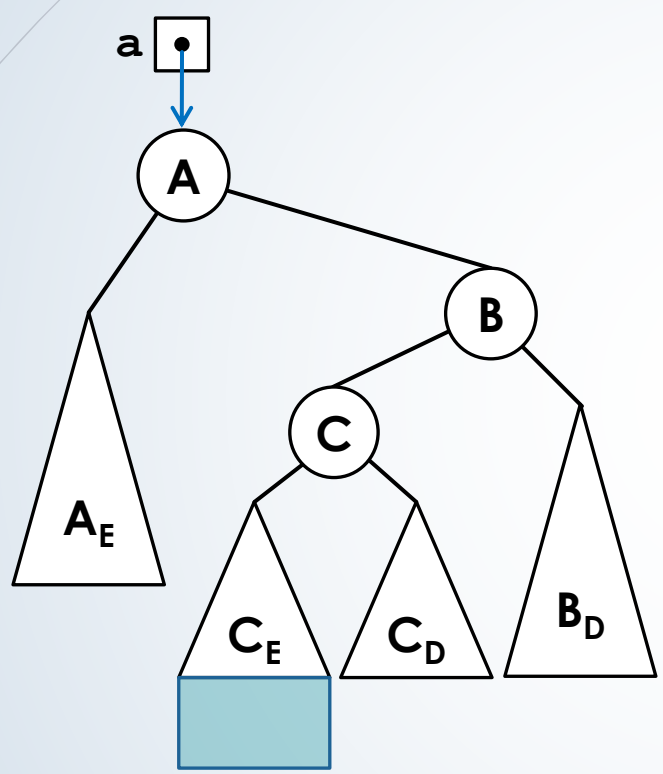


subárvore desbalanceada



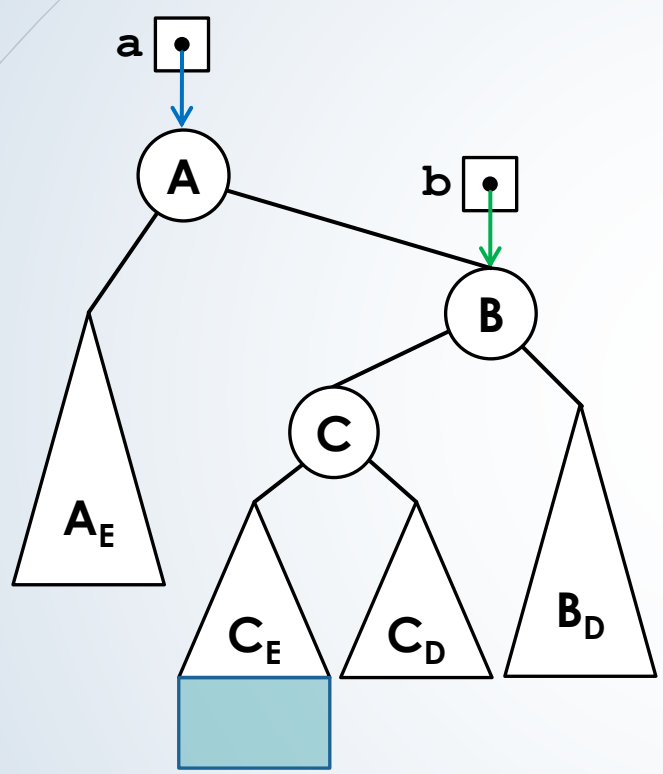
subárvore rebalanceada

Rotação DE



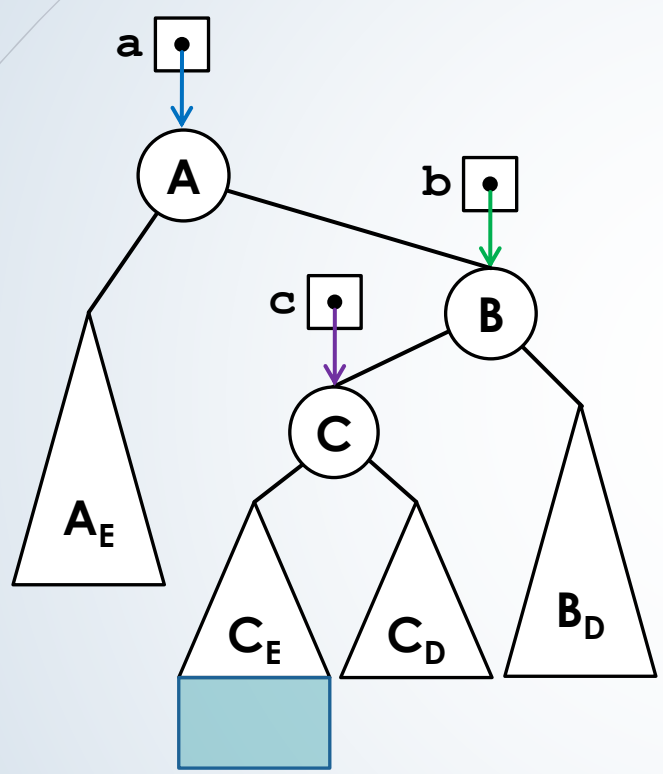

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE



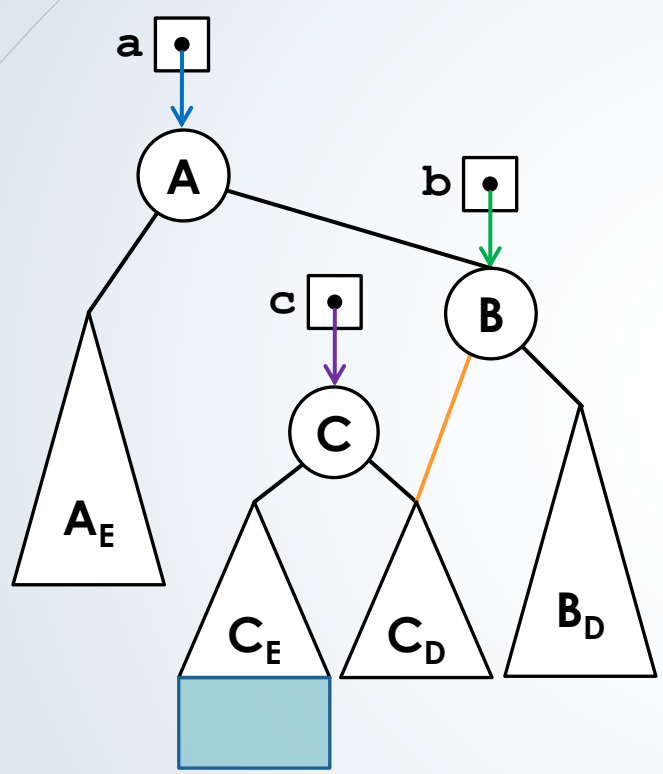

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE



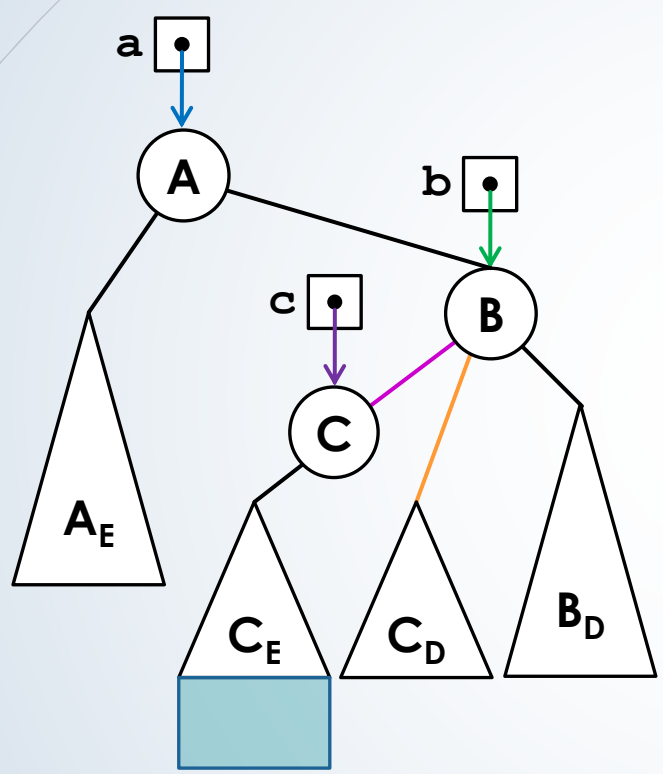

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE



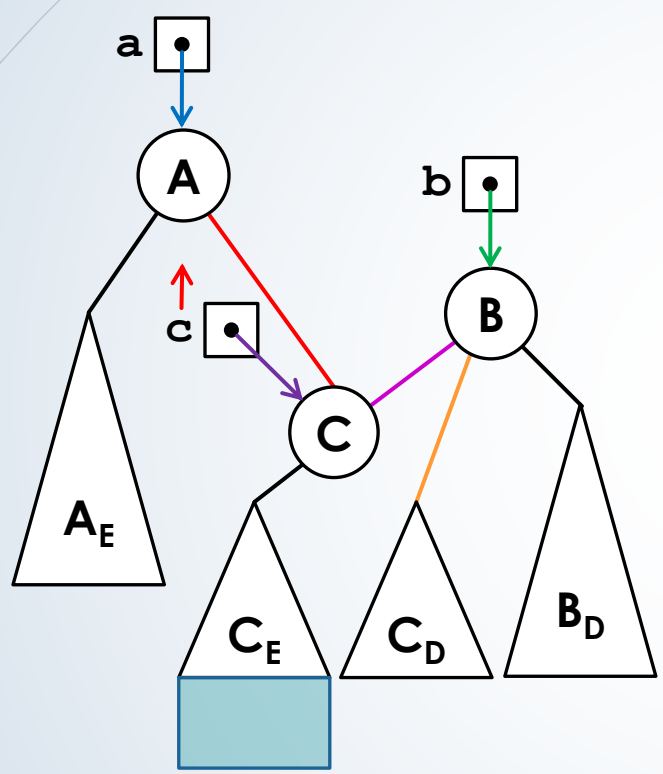

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE



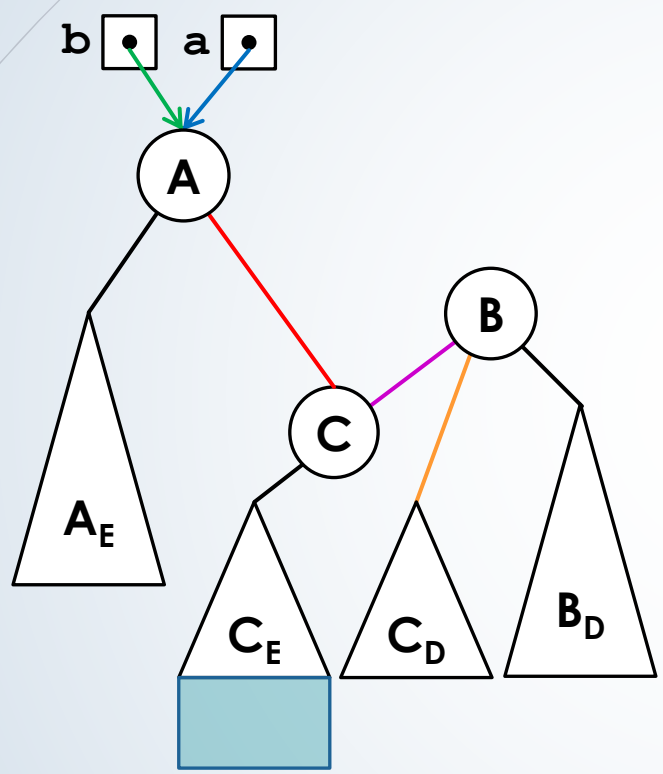

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE



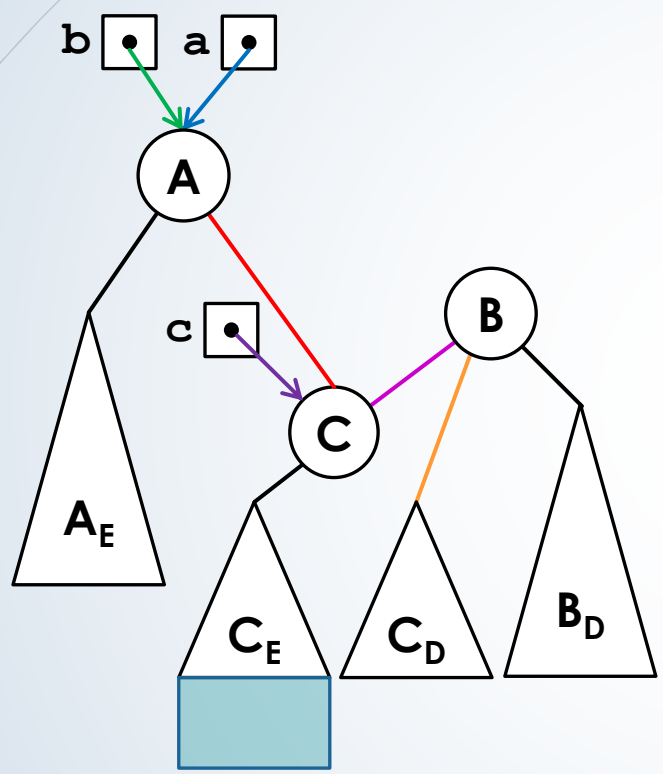

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE



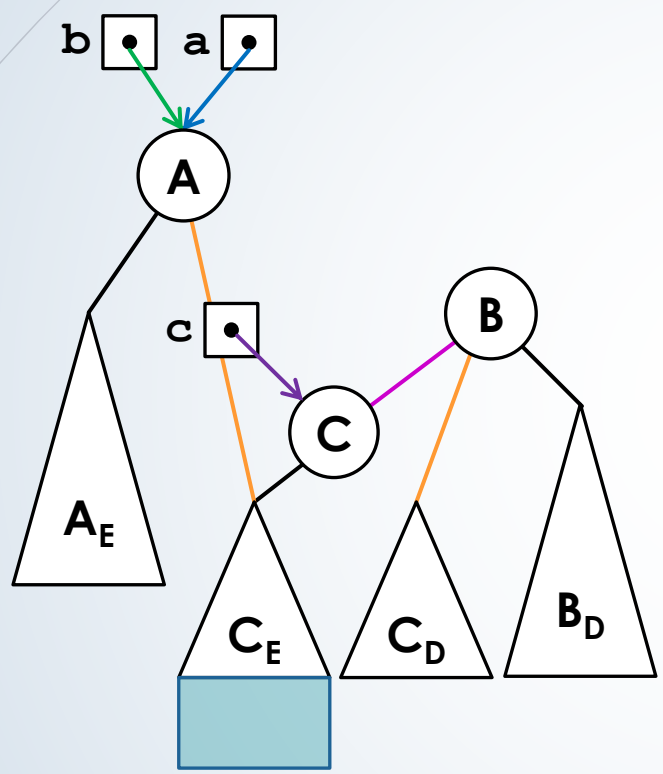

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE



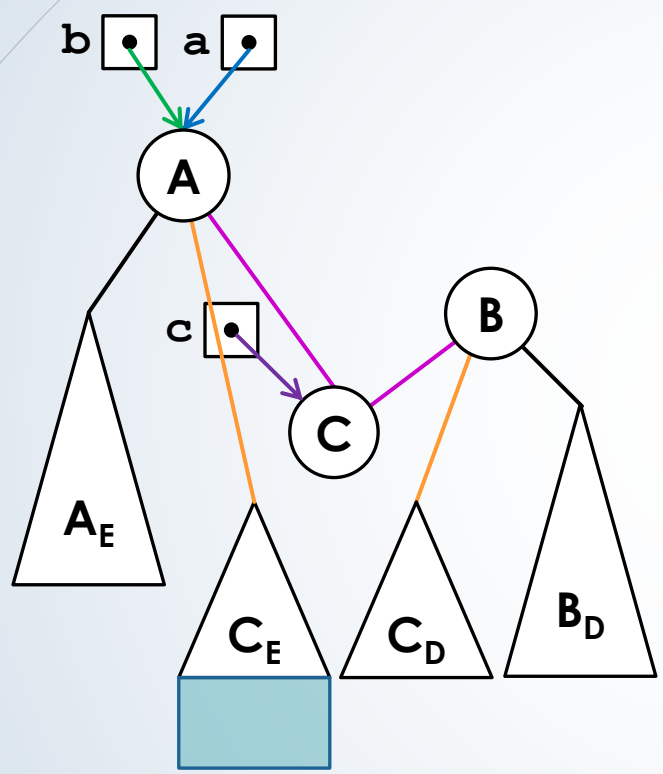

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE



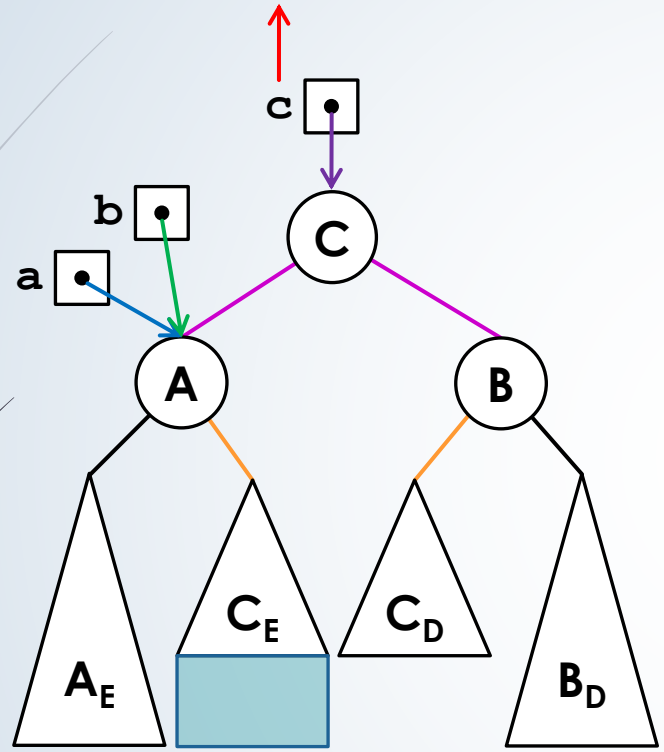

```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE




```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Rotação DE




```
private No<Tipo> de( No<Tipo> a ) {
    a.direita = ee( a.direita );
    return dd( a );
}

private No<Tipo> ee( No<Tipo> b ) {
    No<Tipo> c = b.esquerda;
    b.esquerda = c.direita;
    c.direita = b;
    return c;
}

private No<Tipo> dd( No<Tipo> b ) {
    No<Tipo> c = b.direita;
    b.direita = c.esquerda;
    c.esquerda = b;
    return c;
}
```

Árvores AVL
Construção

- Inserir na ordem:
  - **H**, **I**, **J**, **B**, **A**, **E**, **C**, **F**, **D**, **G**, **K** e **L**

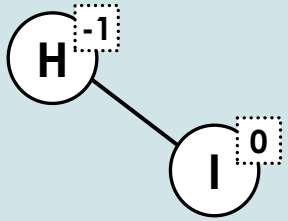
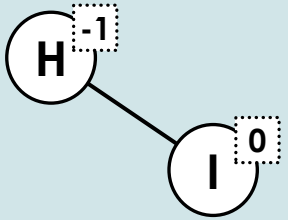
Árvores AVL
Construção

```
avl.put( "H" );
```

Inserção	Rebalanceamento
	 **Não há necessidade de rebalanceamento!**

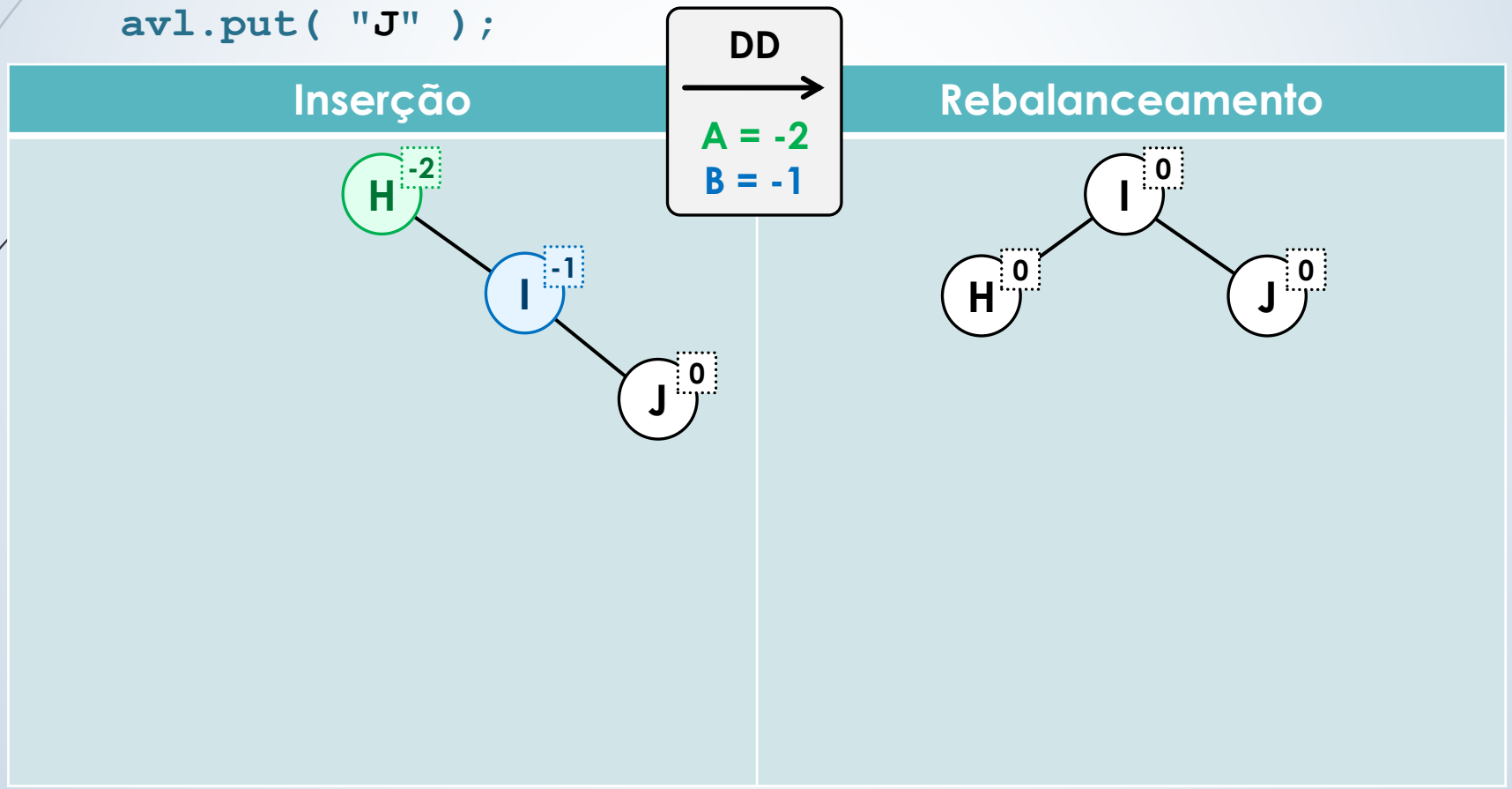
Árvores AVL
Construção

```
avl.put( "I" );
```

Inserção	Rebalanceamento
	
	Não há necessidade de rebalanceamento!

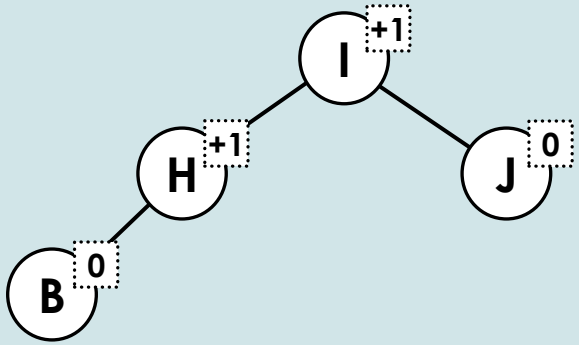
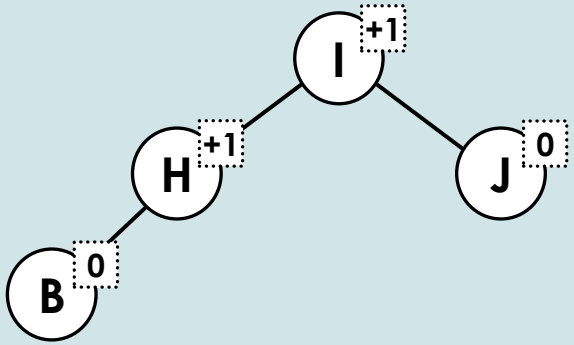
Árvores AVL
Construção

```
avl.put( "J" );
```



Árvores AVL
Construção

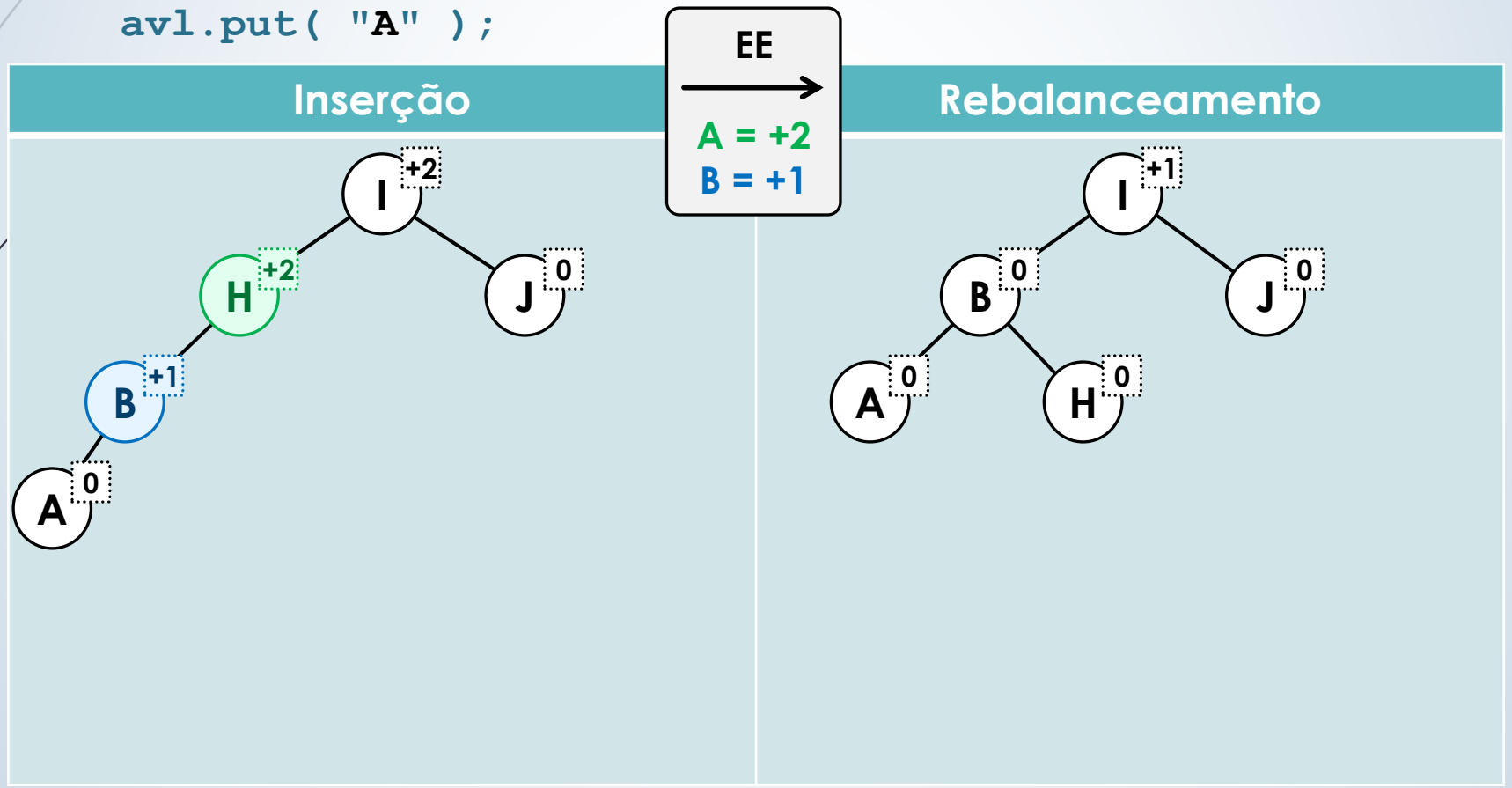
```
avl.put( "B" );
```

Inserção	Rebalanceamento
	

Não há necessidade de rebalanceamento!

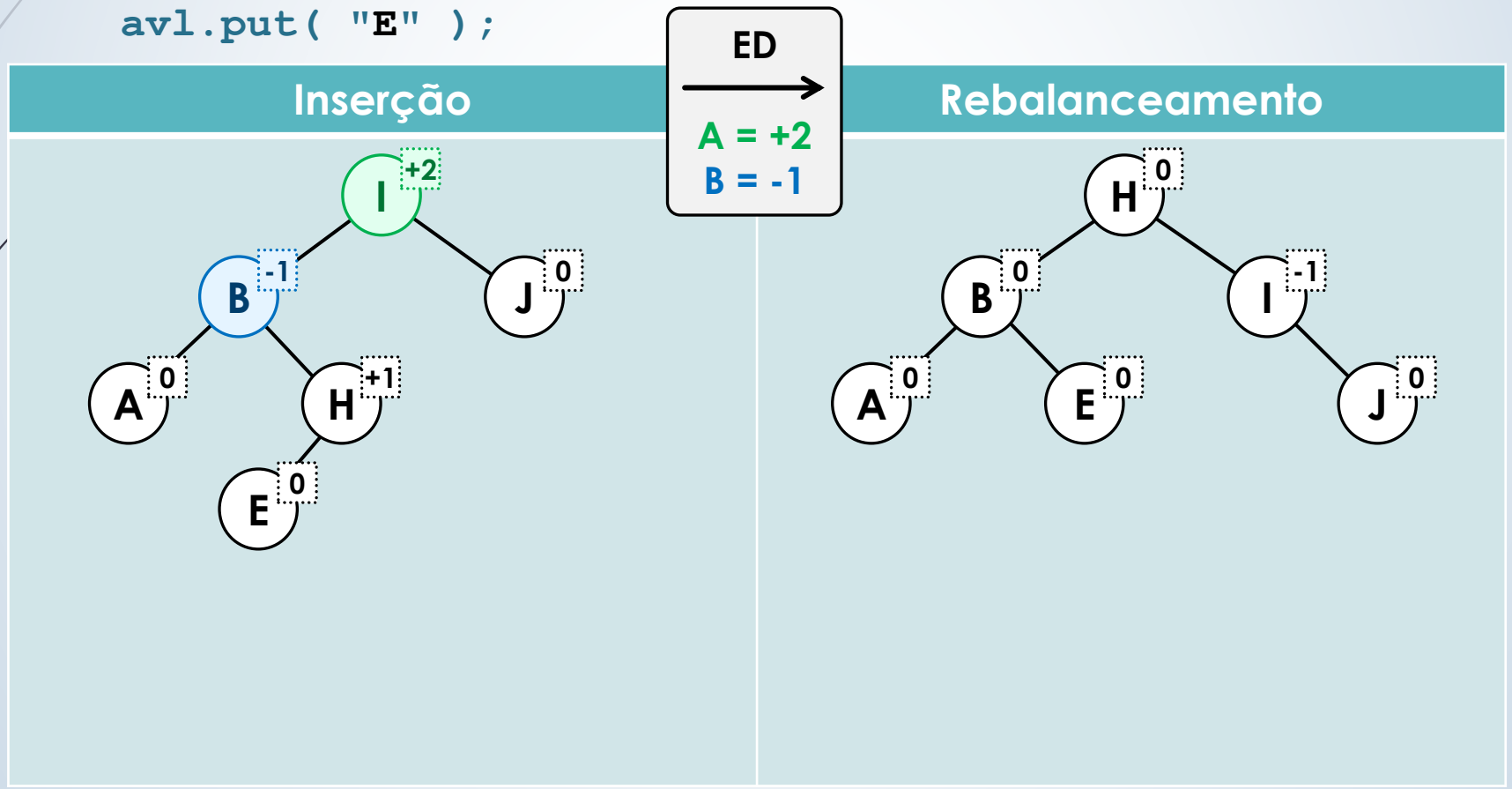
Árvores AVL
Construção

```
avl.put( "A" );
```



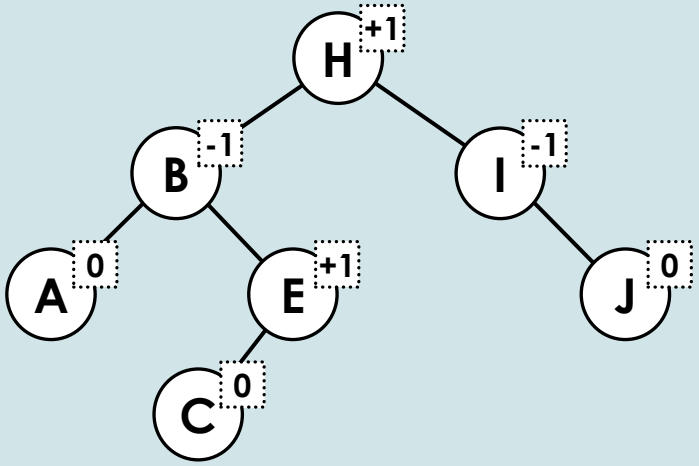
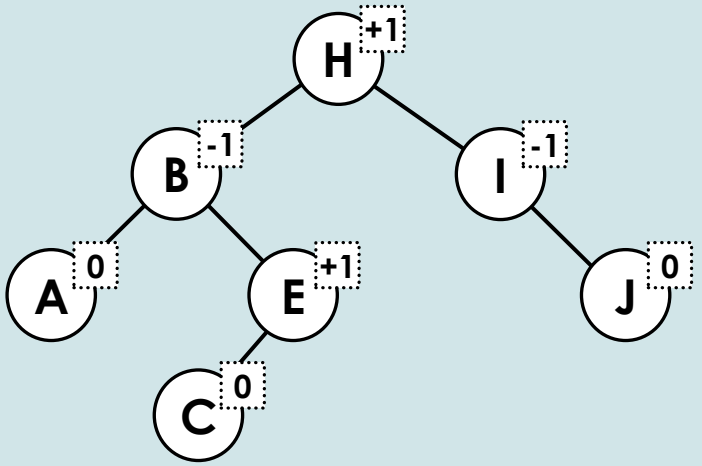
Árvores AVL
Construção

```
avl.put( "E" );
```



Árvores AVL
Construção

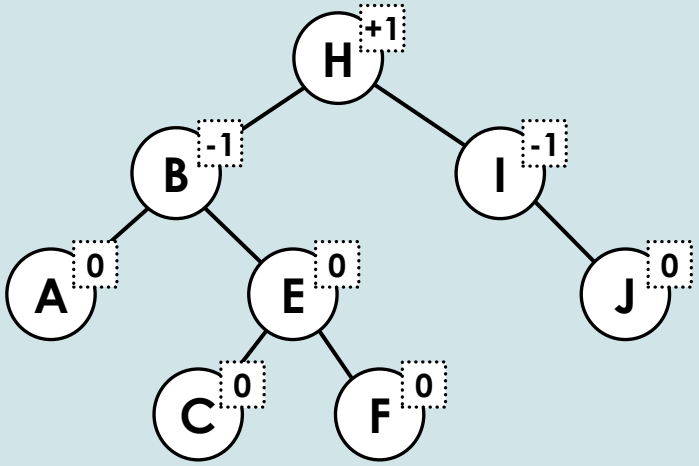
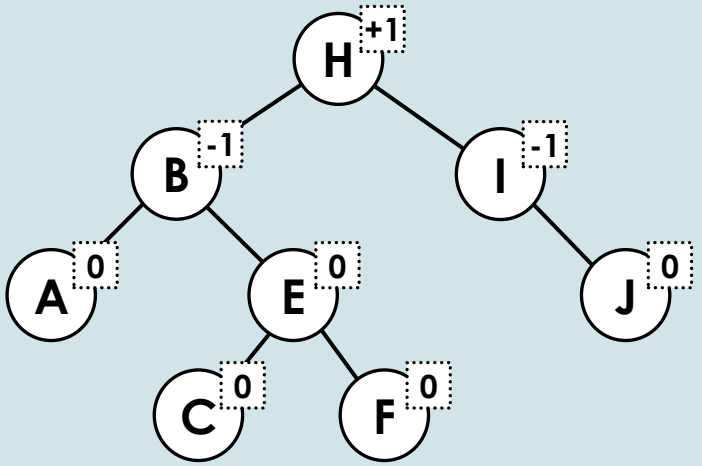
```
avl.put( "C" );
```

Inserção	Rebalanceamento
	

Não há necessidade de rebalanceamento!

Árvores AVL
Construção

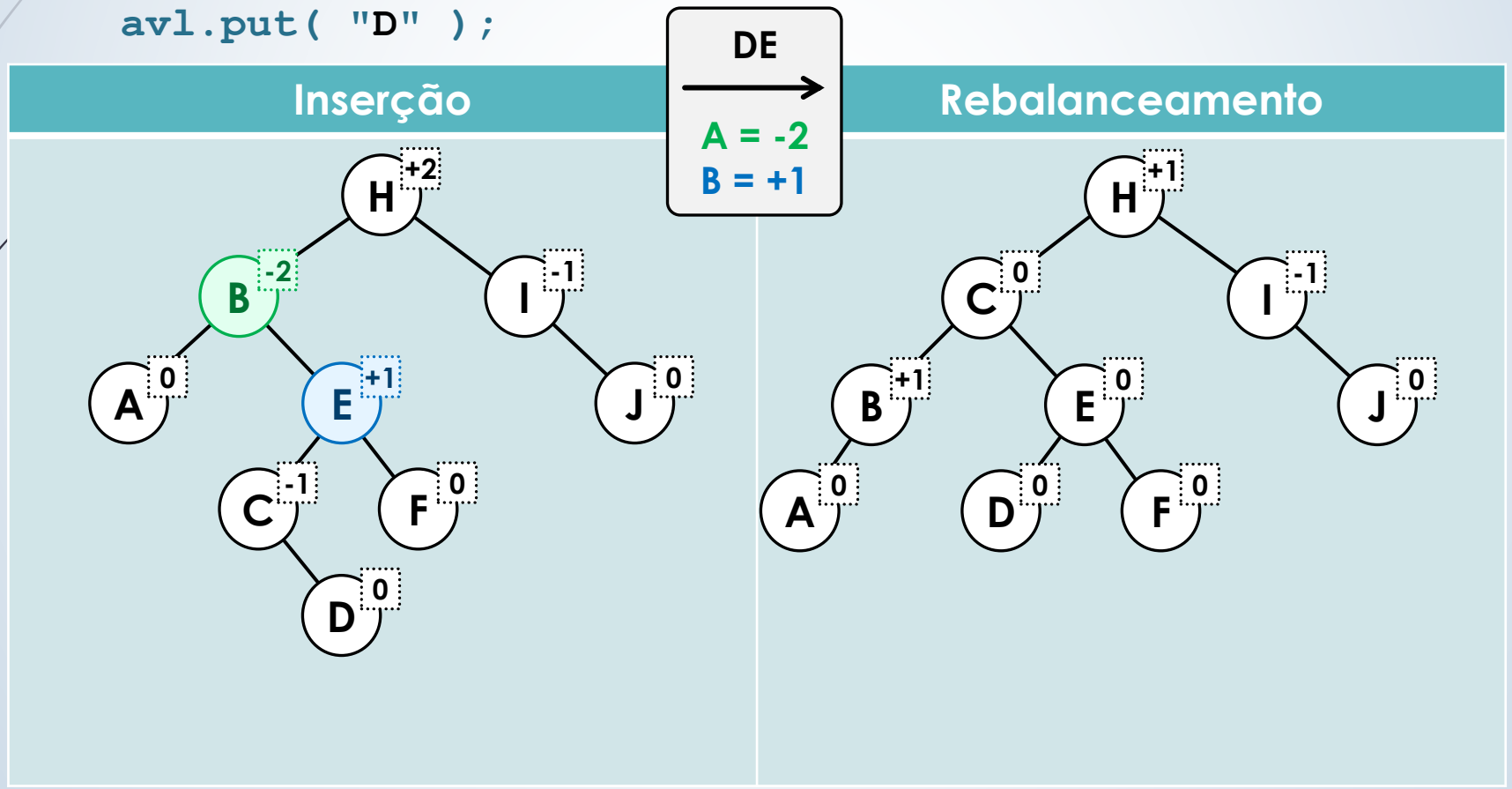
```
avl.put( "F" );
```

Inserção	Rebalanceamento
	

Não há necessidade de rebalanceamento!

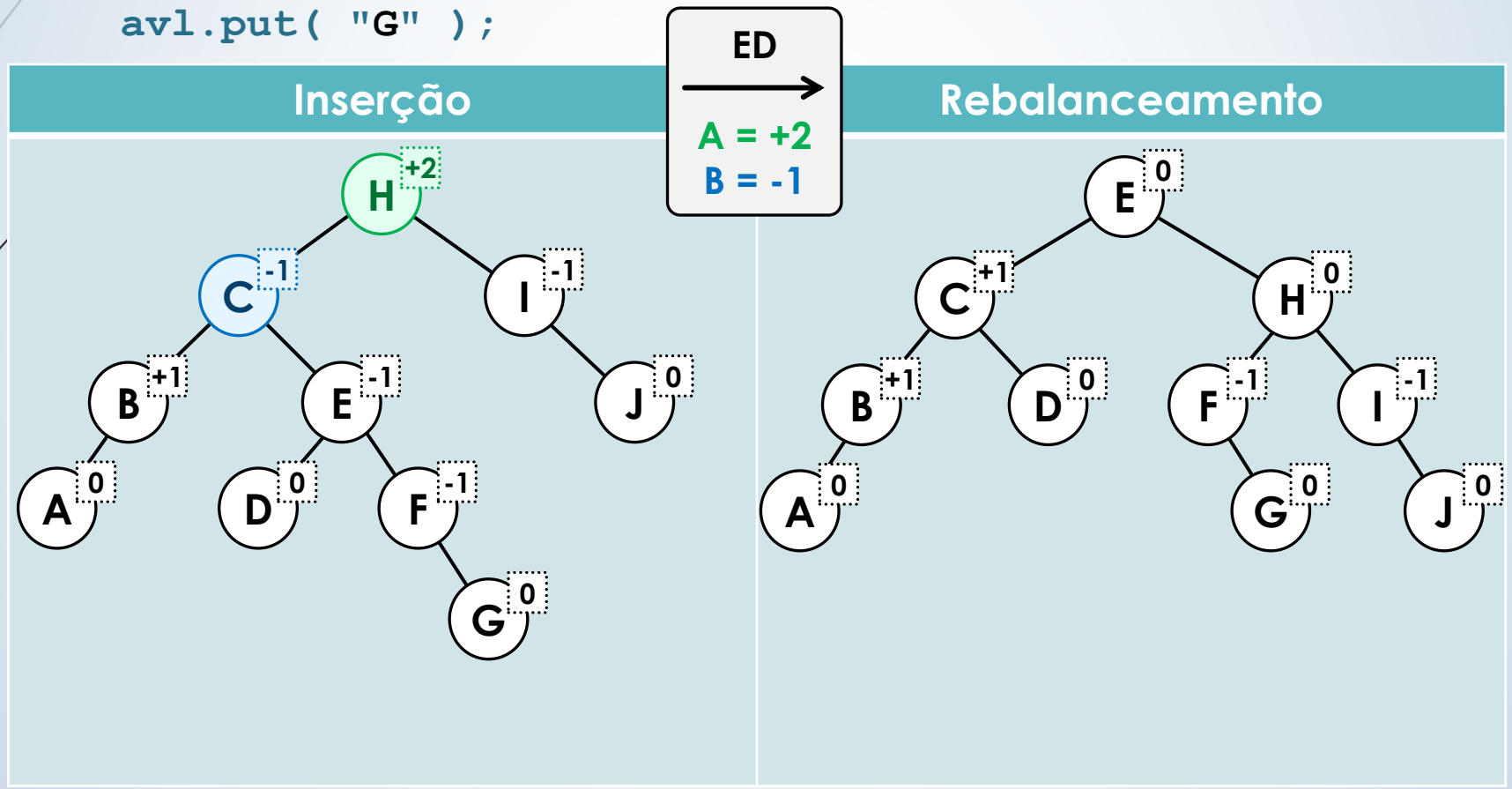
Árvores AVL
Construção

```
avl.put( "D" );
```



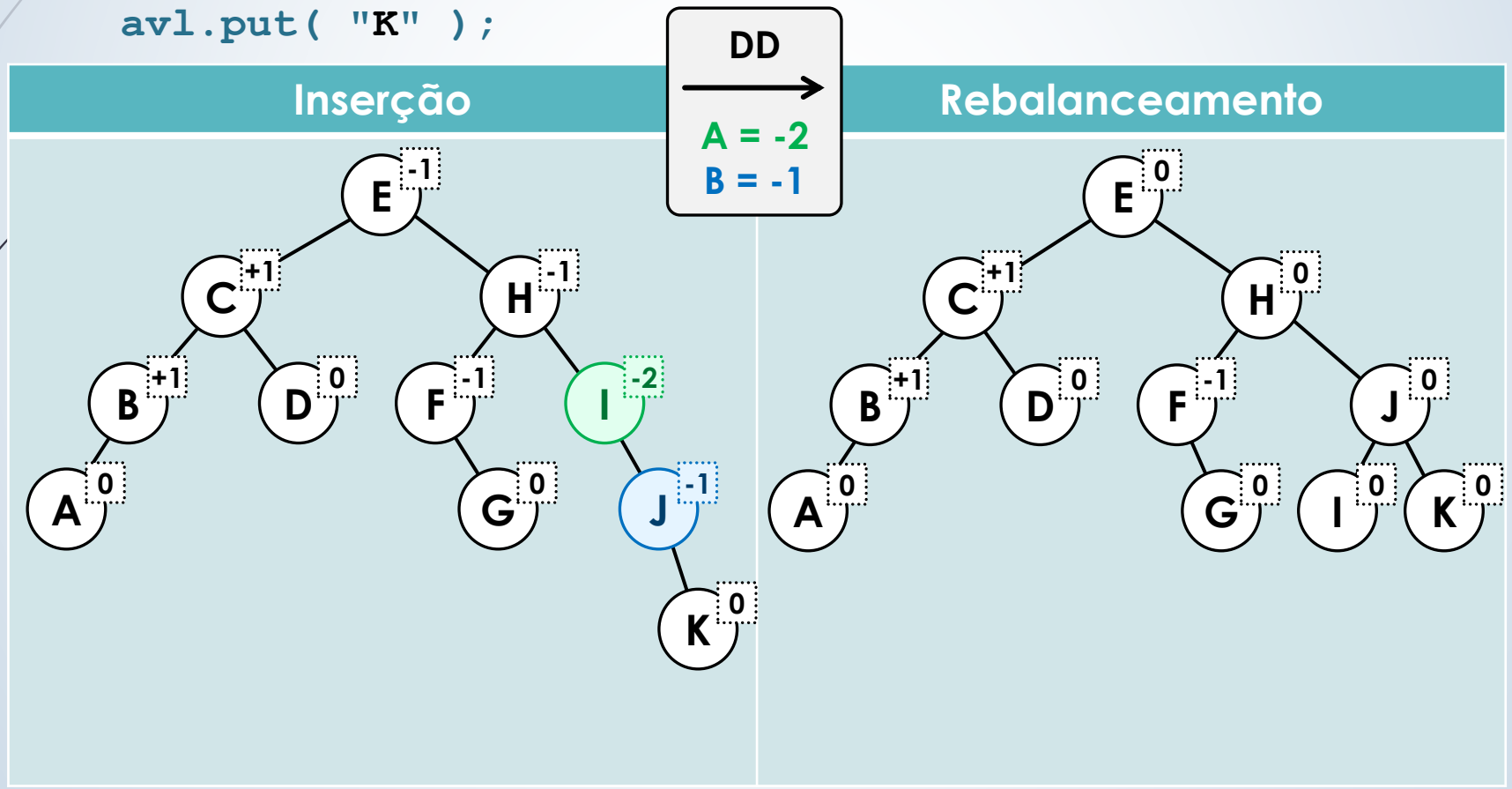
Árvores AVL
Construção

```
avl.put( "G" );
```



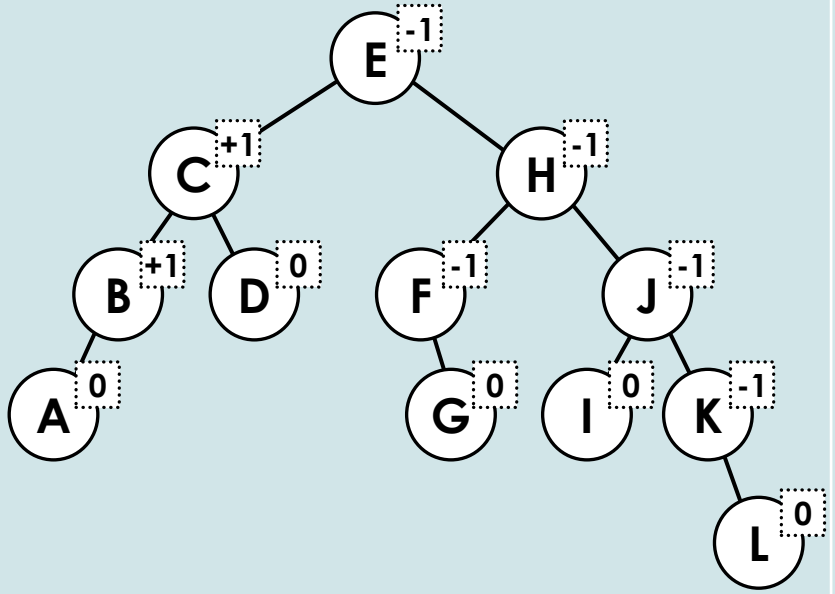
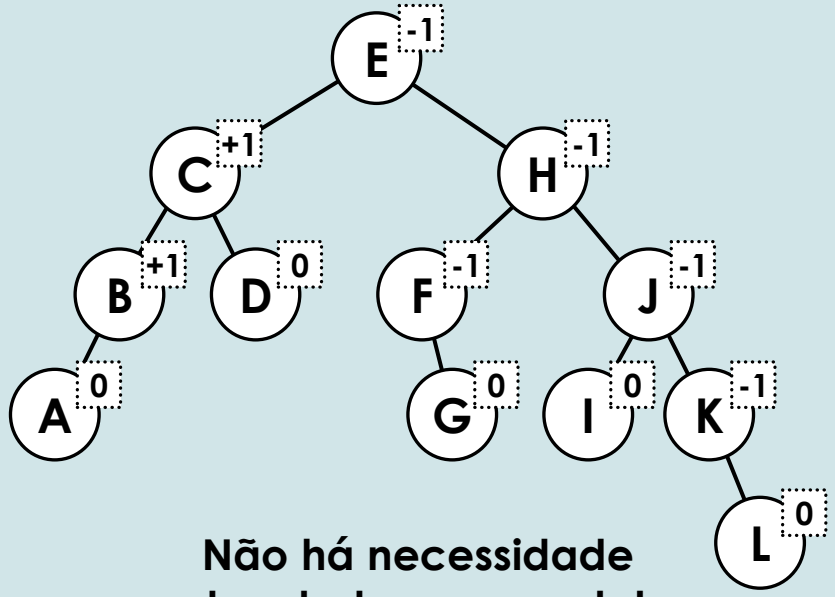
Árvores AVL
Construção

```
avl.put( "K" );
```



Árvores AVL
Construção

```
avl.put( "L" );
```

Inserção	Rebalanceamento
	

Não há necessidade de rebalanceamento!

Bibliografia

SEDGEWICK, R.; WAYNE, K. **Algorithms**. 4. ed. Boston: Pearson Education, 2011. 955 p.

WEISS, M. A. Data Structures and **Algorithm Analysis in Java**. 3. ed. Pearson Education: New Jersey, 2012. 614 p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos – Teoria e Prática**. 3. ed. São Paulo: GEN LTC, 2012. 1292 p.