

# Árvores AVL

Prof. Luiz Gustavo Almeida Martins

# Árvores binárias de busca (ABB)

---

**Árvore binária ordenada** na qual os dados são distribuídos pelos nós para **facilitar a pesquisa**

# Árvores binárias de busca (ABB)

---

**Árvore binária ordenada** na qual os dados são distribuídos pelos nós para **facilitar a pesquisa**

**Distribuição dos dados** deve obedecer o critério de ordenação (**ex:** ordem crescente):

**Elementos à esquerda (SAE) < raiz**

**Elementos à direita (SAD) > raiz**

# Árvores binárias de busca (ABB)

---

**Árvore binária ordenada** na qual os dados são distribuídos pelos nós para **facilitar a pesquisa**

**Distribuição dos dados** deve obedecer o critério de ordenação (**ex:** ordem crescente):

**Elementos à esquerda (SAE) < raiz**

**Elementos à direita (SAD) > raiz**

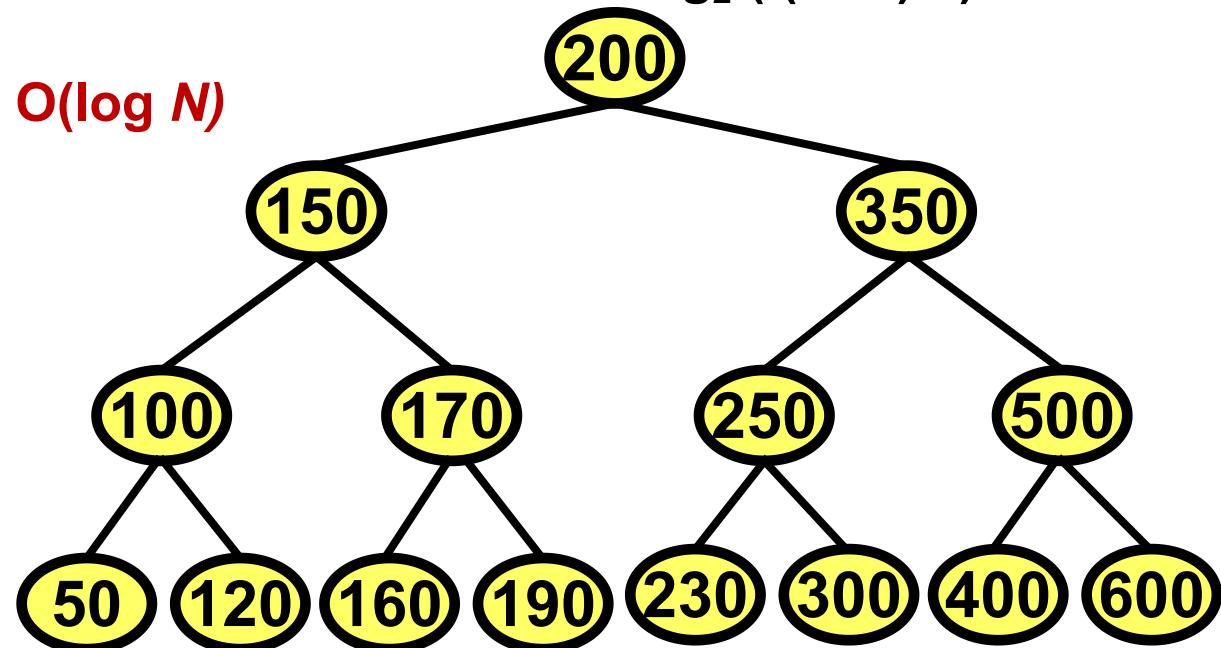
**Eficiência** está relacionada com a **altura da árvore**

Operações básicas (inserção, remoção e busca) levam **tempo proporcional a quantidade de níveis** da árvore

# Exemplo de árvore binária de busca

## Árvore cheia (melhor caso):

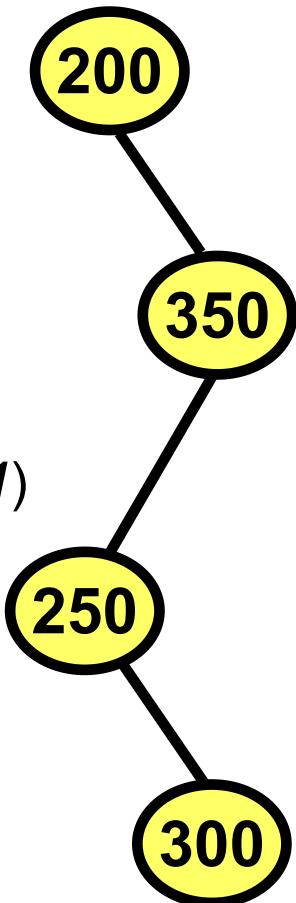
- Cada nível  $X$  tem o número máximo de nós ( $2^X$ )
- Árvore armazena o **maior** número possível de nós ( $2^{h+1} - 1$ )
- **Nº divisões da busca** = altura  $h$  da árvore =  $\log_2 ( (N+1)/2 )$
- **Custo da busca** =  $O(\log N)$



# Exemplo de árvore binária de busca

## Árvore degenerada (pior caso):

- Influenciada pela **ordem de entrada dos elementos (chaves ordenadas)** ou por **sucessivas remoções**
- Cada nível da árvore tem apenas **1 nó**
- Árvore armazena o **menor** número possível de nós ( $h+1$ )
- **Nº divisões da busca** = altura  $h$  da árvore =  $N-1$
- **Custo da busca =  $O(N)$**



# Balanceamento

---

Proporciona uma **distribuição equilibrada dos nós**

Diminui o número médio de comparações

Otimiza as operações básicas

Custo na ordem de  $O(\log N)$

# Balanceamento

---

Proporciona uma **distribuição equilibrada dos nós**

Diminui o número médio de comparações

Otimiza as operações básicas

Custo na ordem de  $O(\log N)$

## Etapas:

**Identificar o desequilibrio** dos nós entre as subárvore

**Corrigir o desequilibrio** (balancear a árvore)

# Balanceamento

Proporciona uma **distribuição equilibrada dos nós**

Diminui o número médio de comparações

Otimiza as operações básicas

Custo na ordem de  $O(\log N)$

## Etapas:

**Identificar o desequilibrio** dos nós entre as subárvore

**Corrigir o desequilibrio** (balancear a árvore)

Algoritmo para manter as árvores **perfeitamente balanceadas** (com altura mínima) possui **alto custo**

**Solução:** árvores "**quase**" balanceadas (**controle + flexível**)

# Balanceamento

---

## Técnicas de balanceamento:

Árvores AVL

Árvores rubro-negras ou vermelho-preto (*red-black*)

Árvores 2-3-4

# Balanceamento

---

## Técnicas de balanceamento:

Árvores AVL

Árvores rubro-negras ou vermelho-preto (*red-black*)

Árvores 2-3-4

Modificam as operações de inserção e remoção para  
**garantir o balanceamento**

# Balanceamento

---

## Técnicas de balanceamento:

Árvores AVL

Árvores rubro-negras ou vermelho-preto (*red-black*)

Árvores 2-3-4

Modificam as operações de inserção e remoção para  
**garantir o balanceamento**

## Problema: **custo do balanceamento**

Frequentes reorganizações da árvore para manter o balanceamento (manutenção onerosa)

**Ganho de desempenho > custo do balanceamento**

# Árvores AVL

---

**Árvore AVL** é uma **árvore binária de busca (ABB)** onde, para todo nó, **as alturas de suas subárvores (SAE e SAD) diferem no máximo em 1 unidade**

Criada por **Adelson-Velskii** e **Landis** em 1962

Realiza um **balanceamento local**

# Árvores AVL

---

**Árvore AVL** é uma **árvore binária de busca (ABB)** onde, para todo nó, **as alturas de suas subárvores (SAE e SAD) diferem no máximo em 1 unidade**

Criada por **Adelson-Velskii** e **Landis** em 1962

Realiza um **balanceamento local**

**Árvore está balanceada** quando suas subárvores tem alturas iguais ou próximas (**diferença de 1 nível**)

Toda árvore perfeitamente balanceada é uma AVL

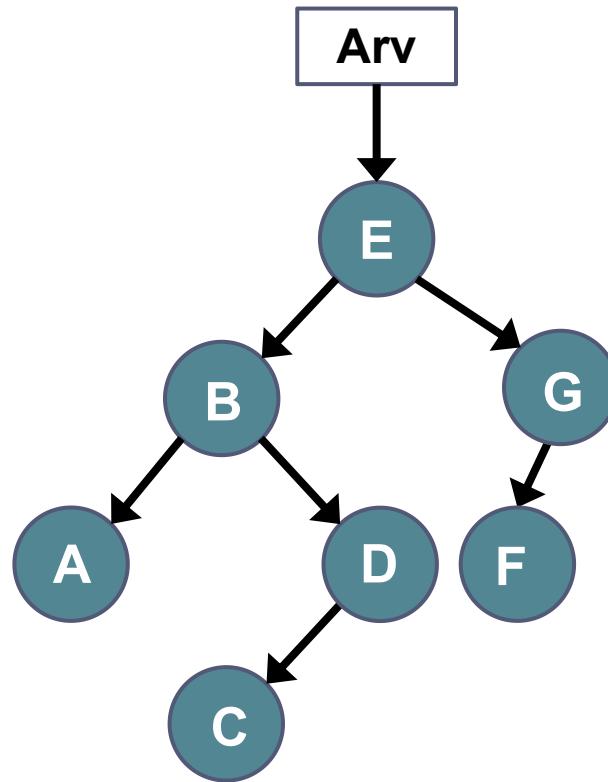
**Ex:** árvores cheias e completas

Nem toda AVL é perfeitamente balanceada

**Ex:** árvores quase completas

# Revisão

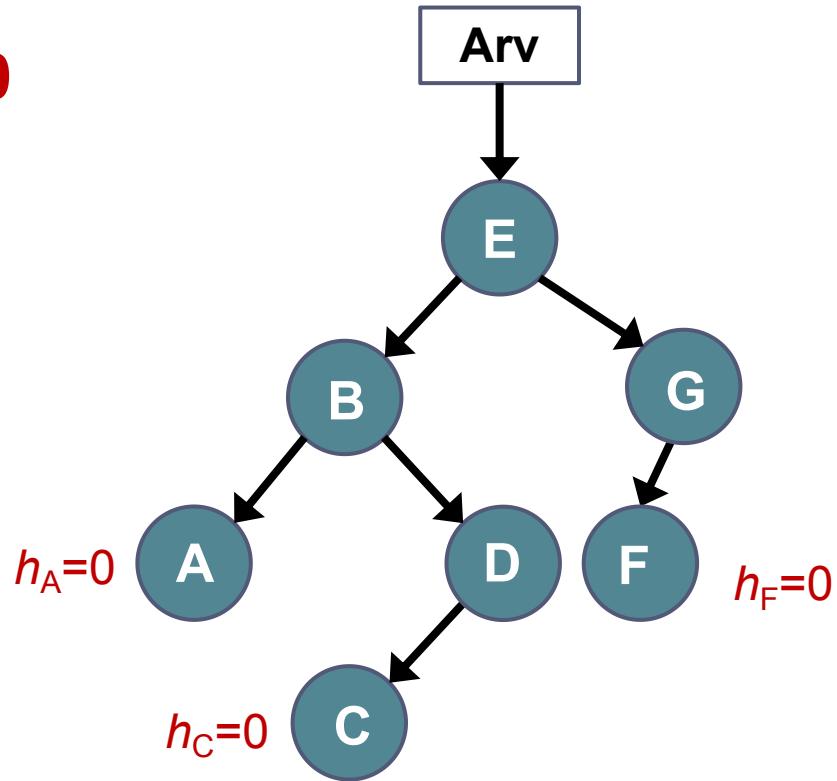
**Altura de um nó é o comprimento do maior caminho do nó até um descendente folha**



# Revisão

**Altura de um nó é o comprimento do maior caminho do nó até um descendente folha**

**Nós folha têm altura = 0**

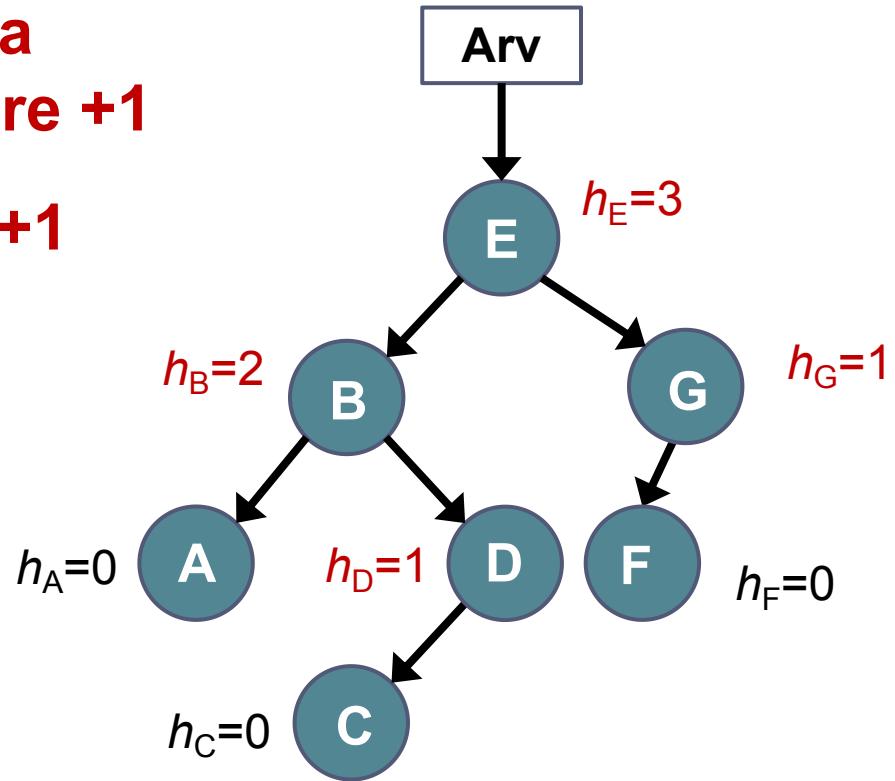


# Revisão

**Altura de um nó é o comprimento do maior caminho do nó até um descendente folha**

**Nós de derivação terão a altura da maior subárvore +1**

$$h_{\text{nó}} = \max(h_{\text{SAE}}, h_{\text{SAD}}) + 1$$

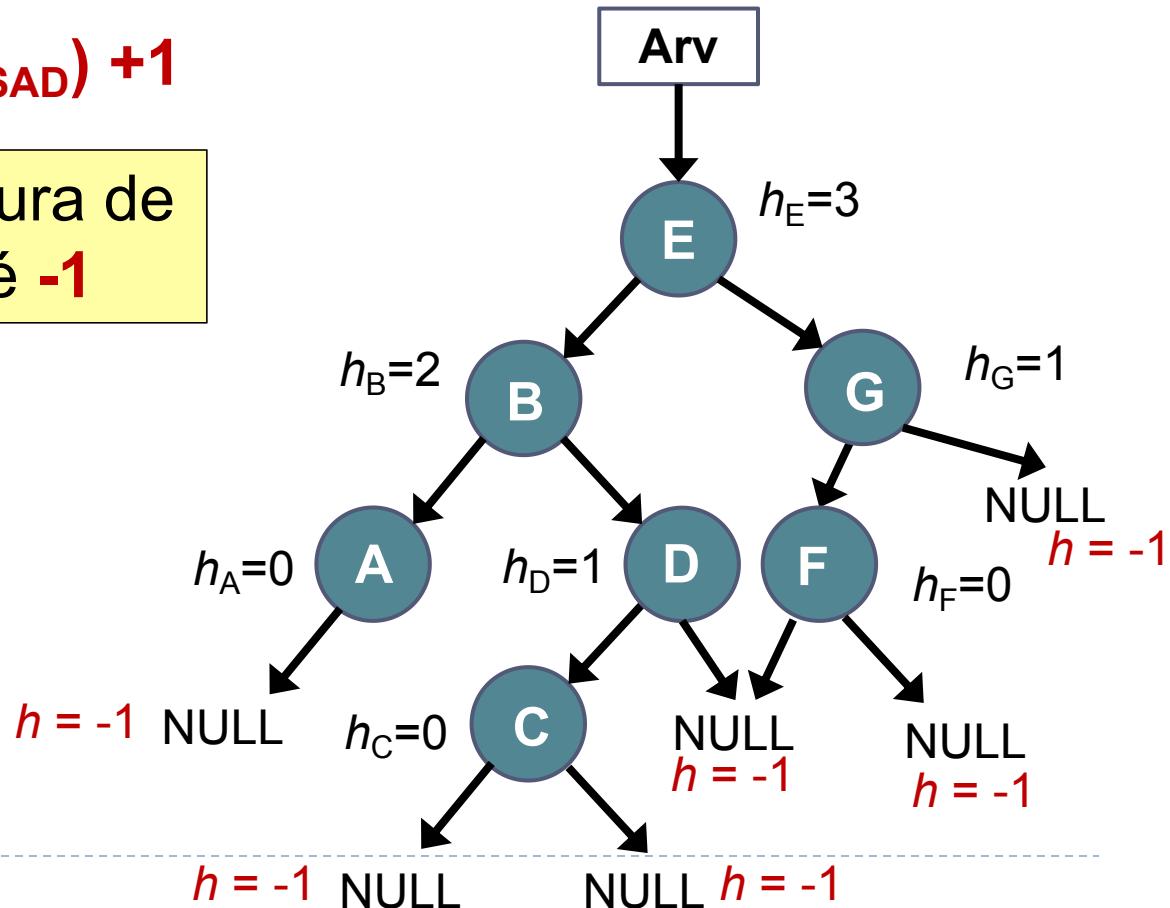


# Revisão

**Altura de um nó é o comprimento do maior caminho do nó até um descendente folha**

$$h_{\text{nó}} = \max(h_{\text{SAE}}, h_{\text{SAD}}) + 1$$

**Generalização:** altura de uma **árvore vazia** é **-1**



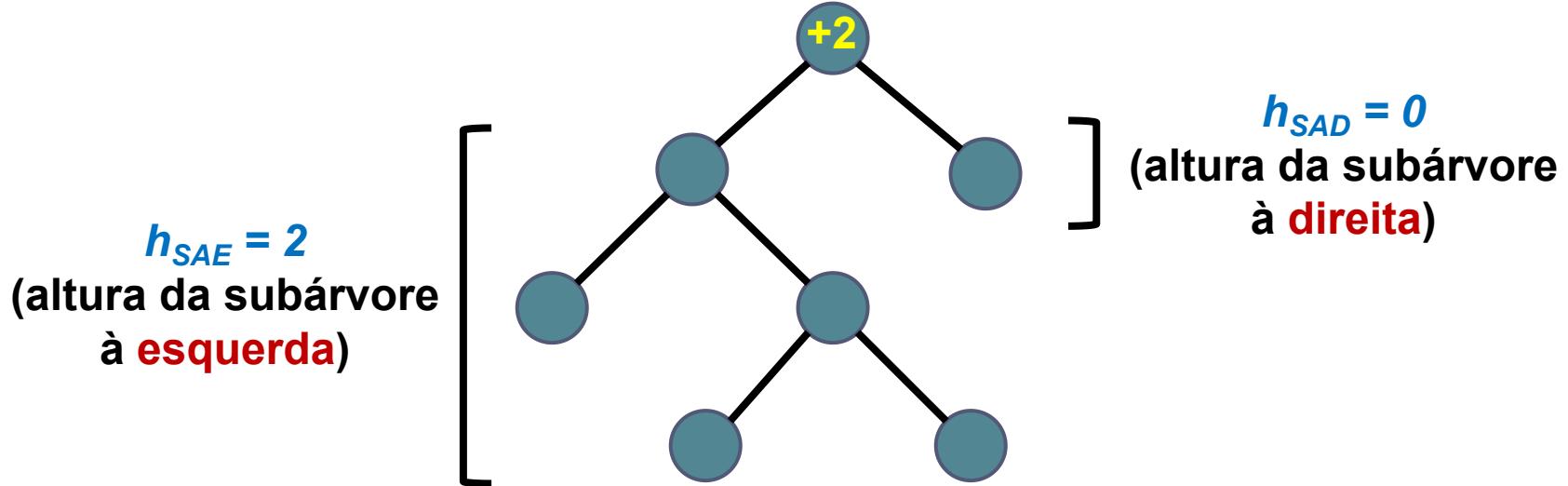
# Árvores AVL

**Questão:** Como identificar o desequilíbrio (**necessidade de balanceamento**)?

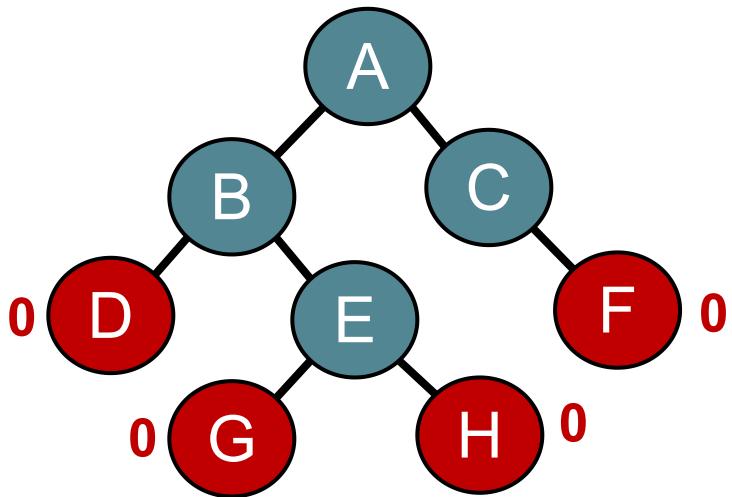
**R:** determinar a diferença de altura entre as subárvore à esquerda e à direita (**fator de balanceamento** ou **FB**)

Raízes das subárvore devem ter **FB entre 1 e -1**

$$\text{FB} = h_{SAE} - h_{SAD} = +2$$



# Exemplo: cálculo FB



Nós folha não possuem subárvore:

$$h = 0$$

$$FB = -1 - (-1) = -1 + 1 = 0$$

indica uma  
subárvore nula

Fator de Balanceamento:

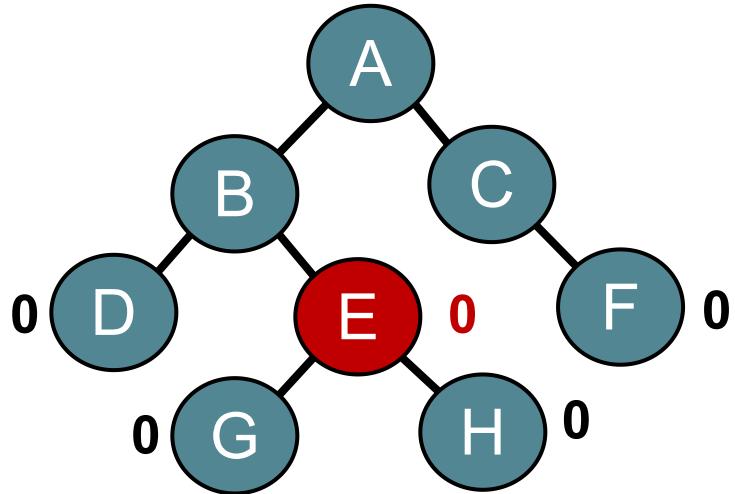
$$FB = h_{SAE} - h_{SAD}$$

# Exemplo: cálculo FB

Nó E possui 2 subárvore:

$$h = 1$$

$$\text{FB} = h(\text{SAE}) - h(\text{SAD}) = 0 - 0 = 0$$



Fator de Balanceamento:

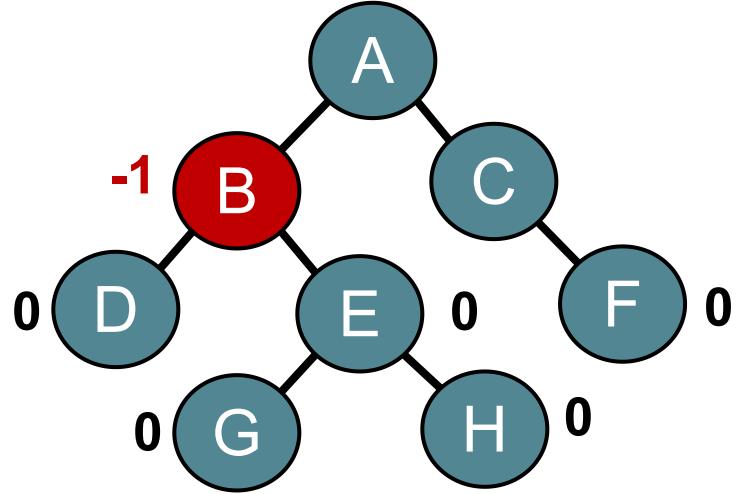
$$\text{FB} = h_{\text{SAE}} - h_{\text{SAD}}$$

# Exemplo: cálculo FB

Nó B possui 2 subárvores:

$$h = 2$$

$$\text{FB} = h(\text{SAE}) - h(\text{SAD}) = 0 - 1 = -1$$



Fator de Balanceamento:

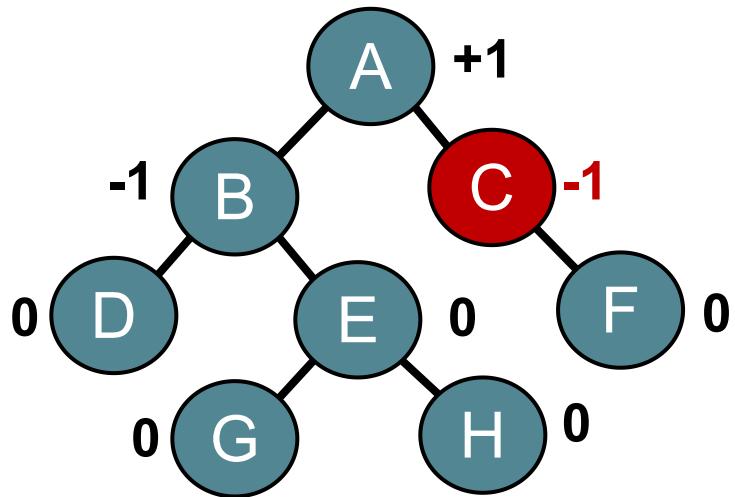
$$\text{FB} = h_{\text{SAE}} - h_{\text{SAD}}$$

# Exemplo: cálculo FB

Nó C possui 1 subárvore (SAD):

$$h = 1$$

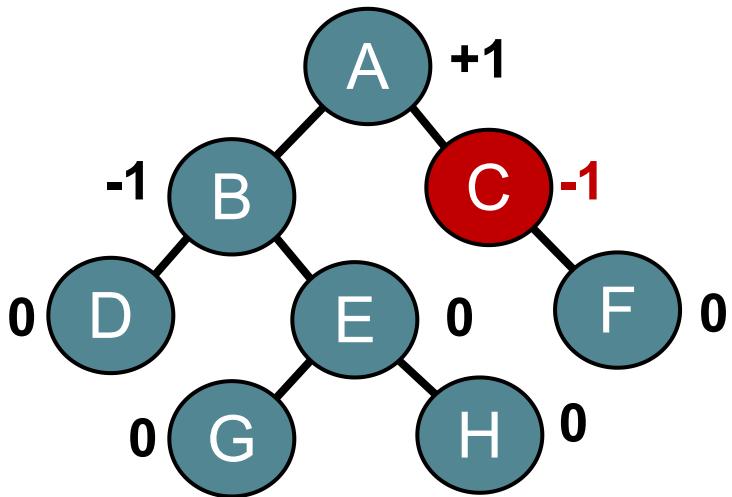
$$\text{FB} = h(\text{SAE}) - h(\text{SAD}) = -1 - 0 = -1$$



Fator de Balanceamento:

$$\text{FB} = h_{\text{SAE}} - h_{\text{SAD}}$$

# Exemplo: cálculo FB



Nó C possui 1 subárvore (SAD):

$$h = 1$$

$$\text{FB} = h(\text{SAE}) - h(\text{SAD}) = \textcircled{-1} - 0 = -1$$

indica uma subárvore nula

Fator de Balanceamento:

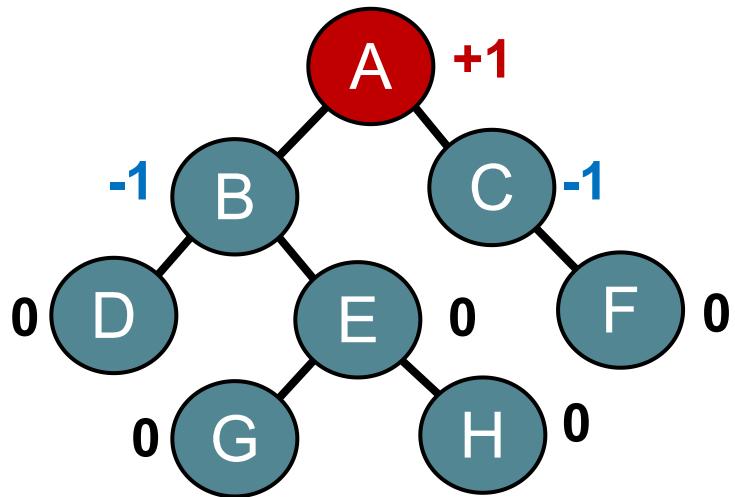
$$\text{FB} = h_{\text{SAE}} - h_{\text{SAD}}$$

# Exemplo: cálculo FB

Nó A possui 2 subárvore:

$$h = 3$$

$$\text{FB} = h(\text{SAE}) - h(\text{SAD}) = 2-1 = +1$$



Fator de Balanceamento:

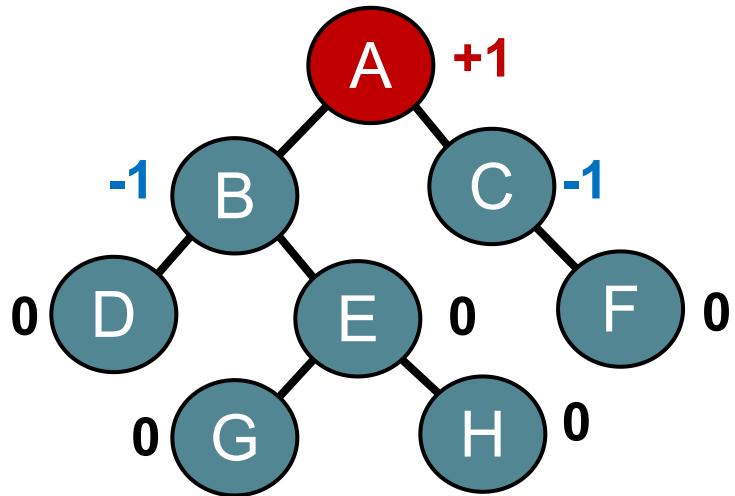
$$\text{FB} = h_{\text{SAE}} - h_{\text{SAD}}$$

# Exemplo: cálculo FB

Nó A possui 2 subárvore:

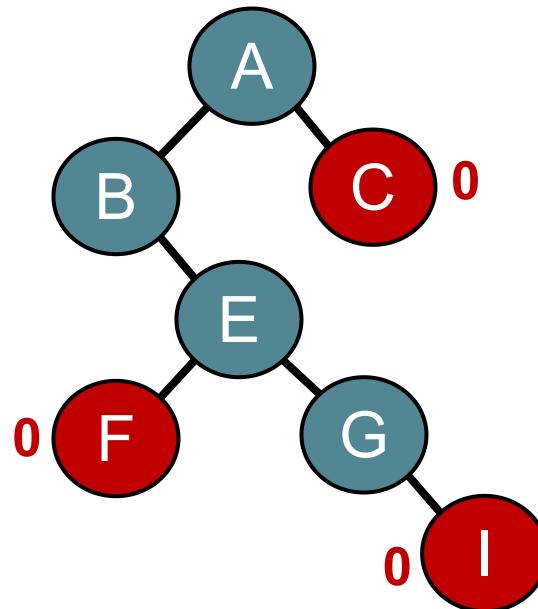
$$h = 3$$

$$\text{FB} = h(\text{SAE}) - h(\text{SAD}) = 2-1 = +1$$



Todos os FB = -1, 0 ou +1  
(árvore AVL)

# Exemplo: cálculo FB



Nós folha não possuem subárvore:

$$h = 0$$

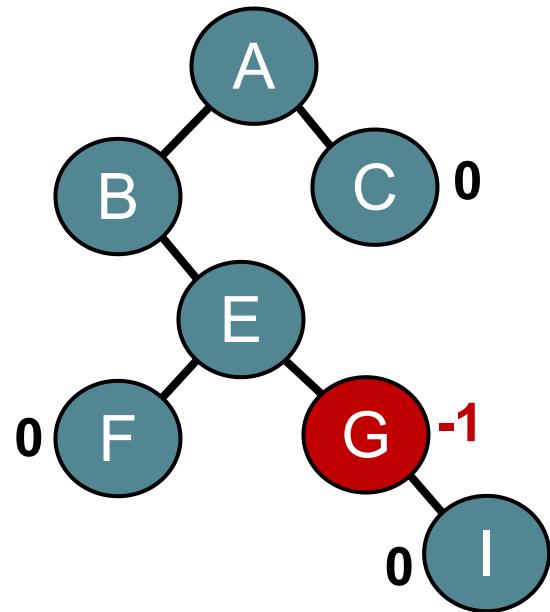
$$FB = -1 - (-1) = -1 + 1 = 0$$

indica uma  
subárvore nula

Fator de Balanceamento:

$$FB = h_{SAE} - h_{SAD}$$

# Exemplo: cálculo FB



Nó **G** possui 1 subárvore (SAD):

$$h = 1$$

$$\text{FB} = h(\text{SAE}) - h(\text{SAD}) = \textcircled{-1} - 0 = -1$$

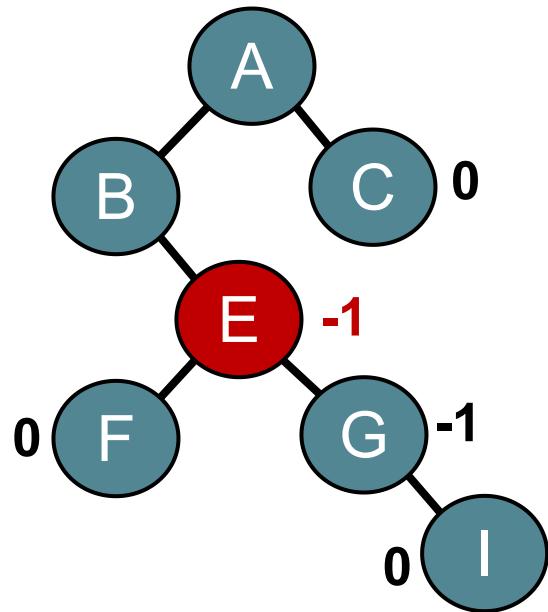
indica uma subárvore nula

Fator de Balanceamento:

$$\text{FB} = h_{\text{SAE}} - h_{\text{SAD}}$$

# Exemplo: cálculo FB

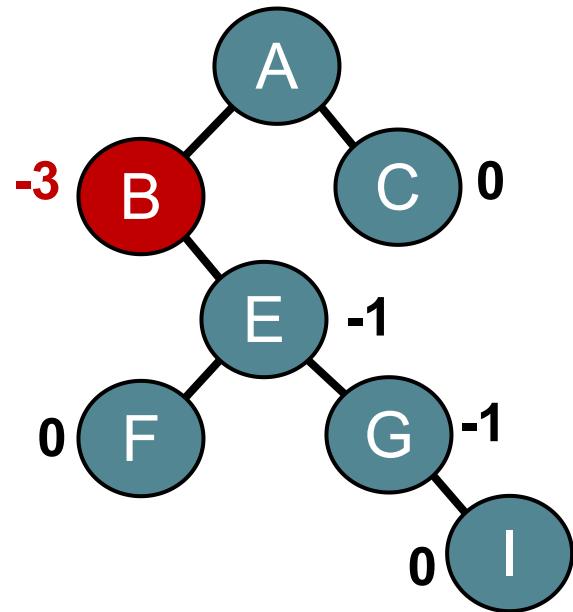
Nó E possui 2 subárvore:  
 $h = 2$   
 $FB = h(\text{SAE}) - h(\text{SAD}) = 0 - (1) = -1$



Fator de Balanceamento:

$$FB = h_{\text{SAE}} - h_{\text{SAD}}$$

# Exemplo: cálculo FB



Nó B possui 1 subárvore (SAD):

$$h = 3$$

$$FB = h(\text{SAE}) - h(\text{SAD}) = \textcircled{-1} - 2 = -3$$

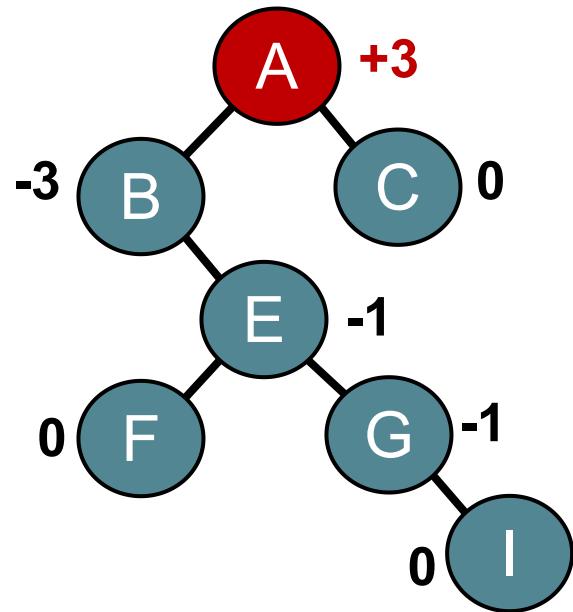
indica uma subárvore nula

Fator de Balanceamento:

$$FB = h_{\text{SAE}} - h_{\text{SAD}}$$

# Exemplo: cálculo FB

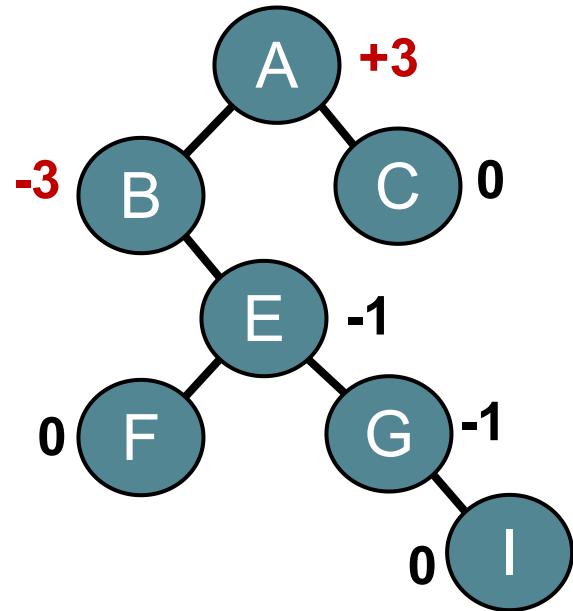
Nó A possui 2 subárvore:  
 $h = 4$   
 $FB = h(\text{SAE}) - h(\text{SAD}) = 3 - (0) = +3$



Fator de Balanceamento:

$$FB = h_{\text{SAE}} - h_{\text{SAD}}$$

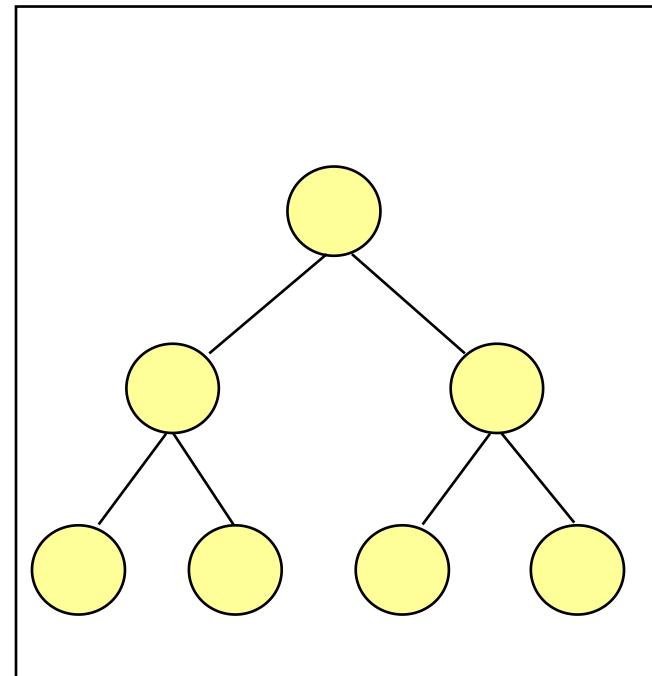
# Exemplo: cálculo FB



Existe  $\text{FB} < -1$  e  $\text{FB} > +1$   
(árvore NÃO é AVL)

# Exemplos de árvores

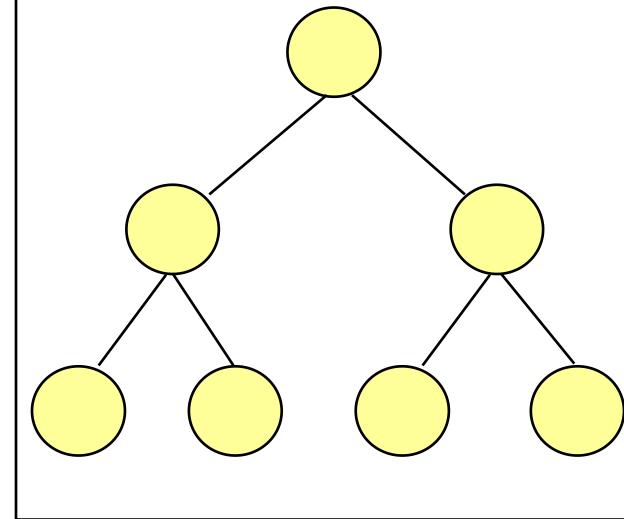
---



# Exemplos de árvores

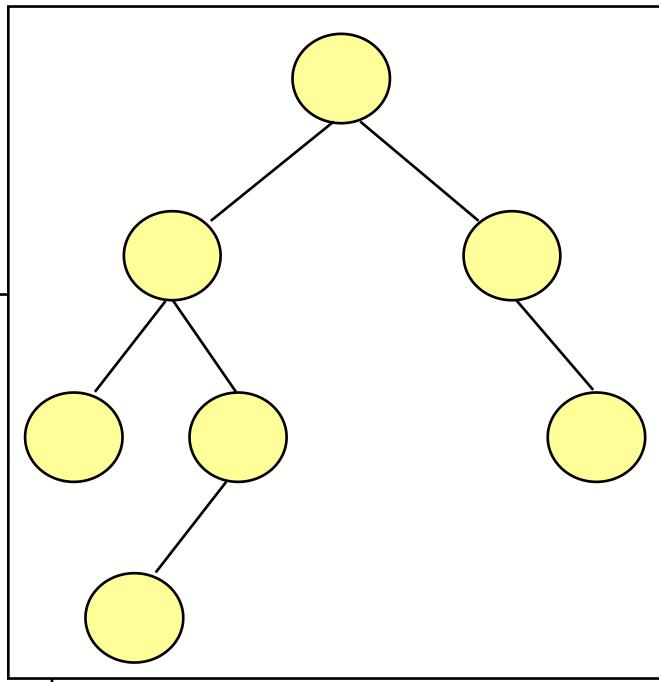
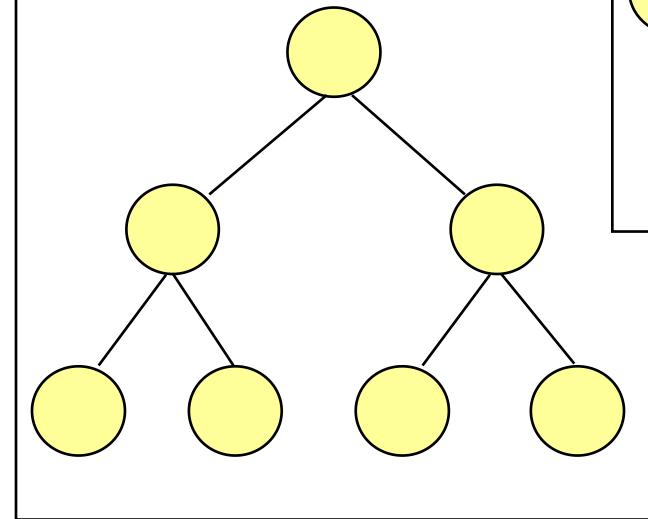
---

**AVL e cheia**



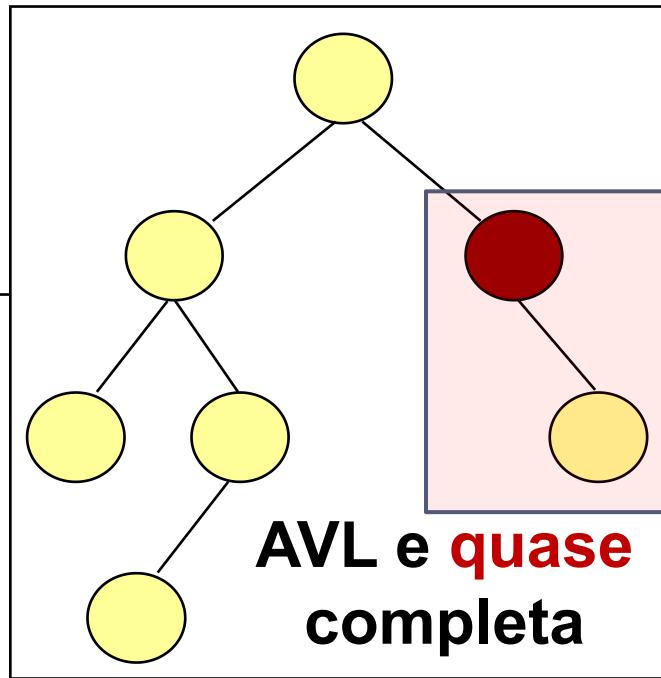
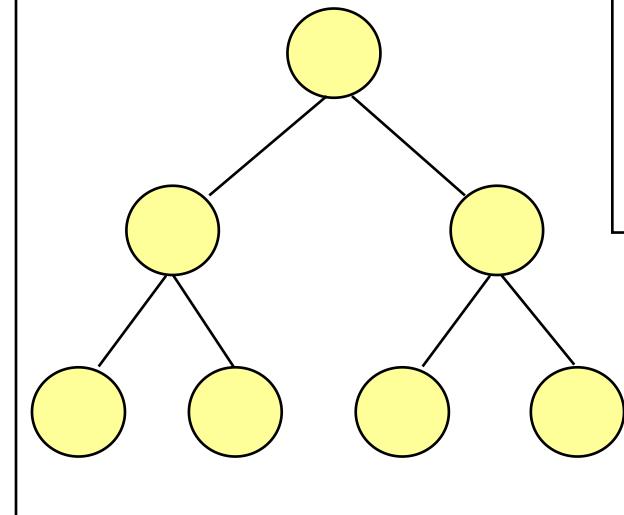
# Exemplos de árvores

**AVL e cheia**



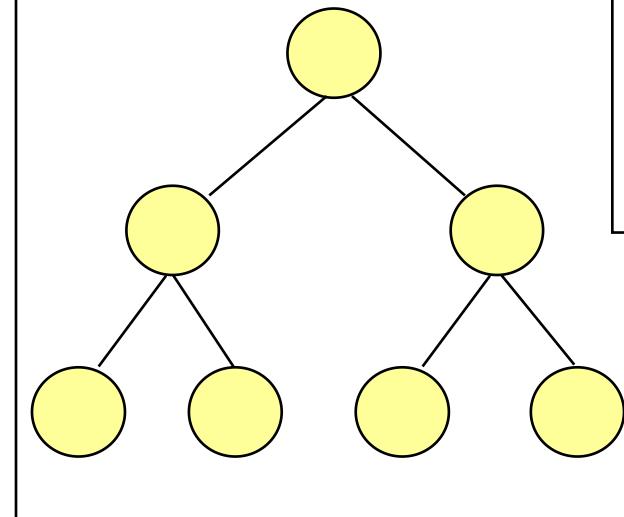
# Exemplos de árvores

AVL e cheia

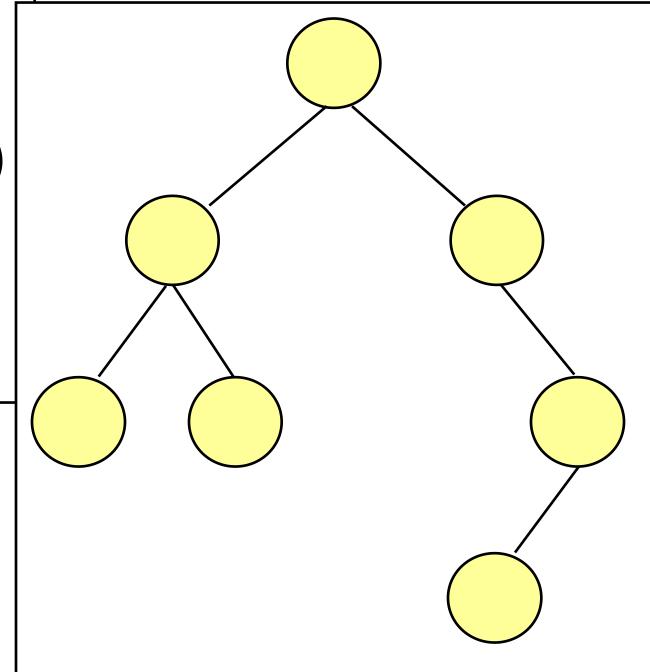
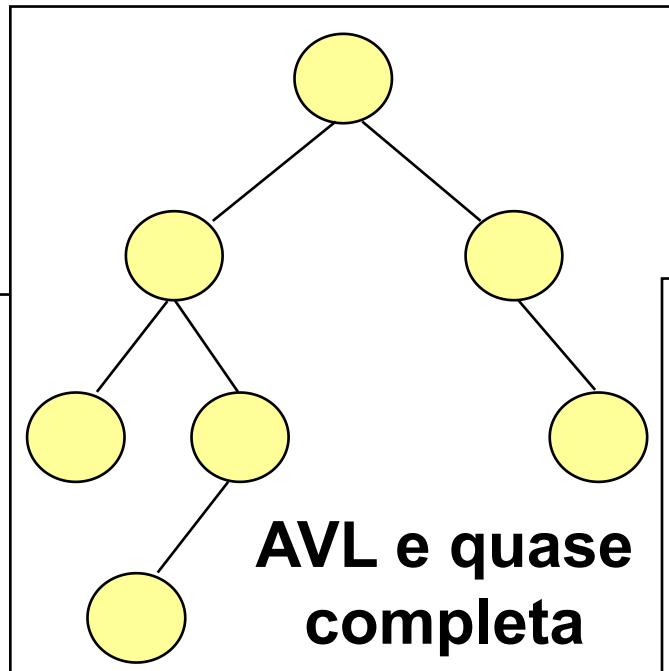


# Exemplos de árvores

AVL e cheia

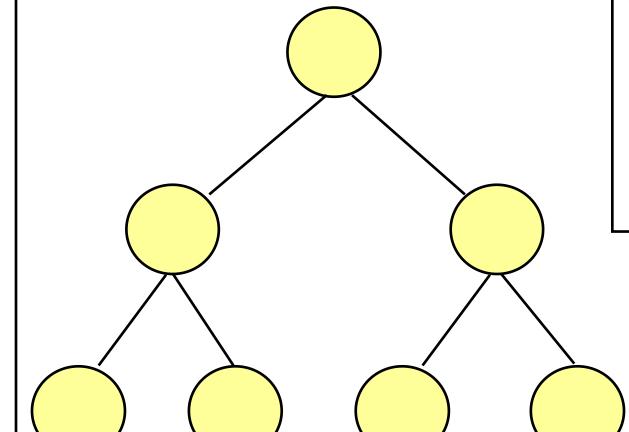


AVL e quase completa

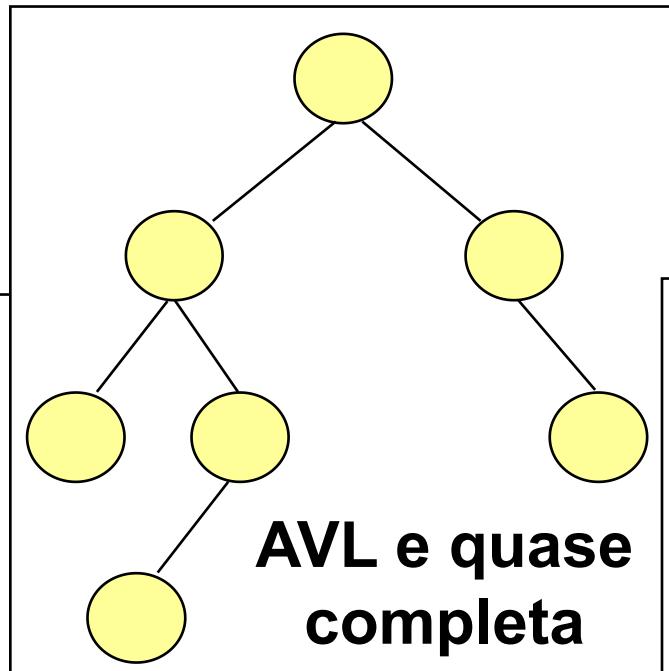


# Exemplos de árvores

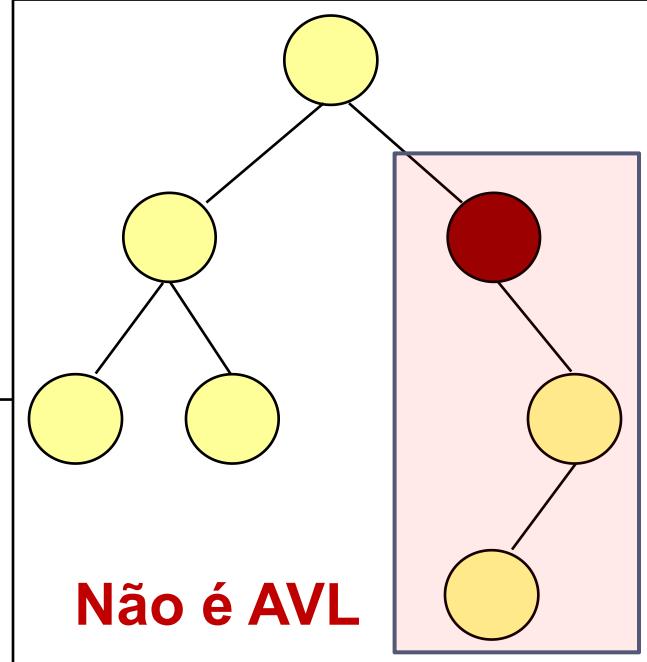
AVL e cheia



AVL e quase completa



Não é AVL



# Estrutura de representação

---

**Questão:** Como saber se uma árvore está balanceada?

# Estrutura de representação

---

**Questão:** Como saber se uma árvore está balanceada?

**R:** Verificar se existe algum **nó desequilibrado**

# Estrutura de representação

---

**Questão:** Como saber se uma árvore está balanceada?

**R:** Verificar se existe algum **nó desequilibrado**

**Questão:** Como determinar desequilíbrio de um nó?

# Estrutura de representação

---

**Questão:** Como saber se uma árvore está balanceada?

R: Verificar se existe algum **nó desequilibrado**

**Questão:** Como determinar desequilíbrio de um nó?

R: Determinar as **alturas das subárvore**s e calcular o **fator de balanceamento (FB)**

Processo com **custo alto** → **operações lentas**

# Estrutura de representação

---

**Questão:** Como saber se uma árvore está balanceada?

R: Verificar se existe algum **nó desequilibrado**

**Questão:** Como determinar desequilíbrio de um nó?

R: Determinar as **alturas das subárvore**s e calcular o **fator de balanceamento (FB)**

Processo com **custo alto** → **operações lentas**

**Questão:** Como ser mais eficiente?

# Estrutura de representação

---

**Questão:** Como saber se uma árvore está balanceada?

R: Verificar se existe algum **nó desequilibrado**

**Questão:** Como determinar desequilíbrio de um nó?

R: Determinar as **alturas das subárvore**s e calcular o **fator de balanceamento (FB)**

Processo com **custo alto** → **operações lentas**

**Questão:** Como ser mais eficiente?

R: Guardar o fator de balanceamento de cada nó

**Mudança na estrutura de representação**

# Estrutura de representação

---

## Estrutura do nó:

**INFO:** dado armazenado no nó (**registro do dicionário**)

**SAE:** endereço da subárvore à esquerda

**SAD:** endereço da subárvore à direita

**FB:** fator de balanceamento do nó (**NOVIDADE**)

SAE	INFO	FB	SAD
-----	------	----	-----

# Estrutura de representação

## Estrutura do nó:

**INFO:** dado armazenado no nó (**registro do dicionário**)

**SAE:** endereço da subárvore à esquerda

**SAD:** endereço da subárvore à direita

**FB:** fator de balanceamento do nó (**NOVIDADE**)

## Implementação em C:

***struct no {***

***Reg info;***

***struct no \* sae;***

***struct no \* sad;***

***int fb;***

***};***

<b>SAE</b>	<b>INFO</b>	<b>FB</b>	<b>SAD</b>
------------	-------------	-----------	------------

# Estrutura de representação

## Estrutura do nó:

**INFO:** dado armazenado no nó (**registro do dicionário**)

**SAE:** endereço da subárvore à esquerda

**SAD:** endereço da subárvore à direita

**h:** altura da subárvore da qual o nó é raiz

## Implementação em C:

```
struct no {
```

```
    Reg info;
```

```
    struct no * sae;
```

```
    struct no * sad;
```

```
    int h;
```

```
};
```

SAE	INFO	<b>h</b>	SAD
-----	------	----------	-----

Outra alternativa é guardar a **altura da subárvore** ao invés do FB

# Árvores AVL

---

**Questão:** Como corrigir o desequilibrio em árvores AVL?

# Árvores AVL

---

**Questão:** Como corrigir o desequilíbrio em árvores AVL?

**R:** Aplicar **rotações** **após as inserções e remoções**, quando necessário, para manter propriedade AVL

# Árvores AVL

---

**Questão:** Como corrigir o desequilíbrio em árvores AVL?

**R:** Aplicar **rotações** **após as inserções e remoções**, quando necessário, para manter propriedade AVL

**Sentido da rotação** é da subárvore maior para a menor

Objetivo é **diminuir a subárvore maior**

Possui **custo constante**

# Árvores AVL

**Questão:** Como corrigir o desequilíbrio em árvores AVL?

**R:** Aplicar **rotações** **após as inserções e remoções**, quando necessário, para manter propriedade AVL

**Sentido da rotação** é da subárvore maior para a menor

Objetivo é **diminuir a subárvore maior**

Possui **custo constante**

Em árvores AVL são aplicadas **4 tipos de rotação**:

**Rotações simples:** ramo desbalanceado possui sentido único de inclinação que é contrário à rotação

**Rotação à direita (LL) ou rotação à esquerda (RR)**

# Árvores AVL

**Questão:** Como corrigir o desequilíbrio em árvores AVL?

**R:** Aplicar **rotações** após as inserções e remoções, quando necessário, para manter propriedade AVL

**Sentido da rotação** é da subárvore maior para a menor

Objetivo é **diminuir a subárvore maior**

Possui **custo constante**

Em árvores AVL são aplicadas **4 tipos de rotação**:

**Rotações simples:** ramo desbalanceado possui sentido único de inclinação que é contrário à rotação

**Rotação à direita (LL) ou rotação à esquerda (RR)**

**Rotações duplas:** ramo desbalanceado possui inclinação nos dois sentidos

**Rotação direita-esquerda (RL) ou rotação esquerda-direita (LR)**

# Rotação à direita (LL)

---

Consiste em "**trocar a posição**" de um nó **X** com seu pai à direita **Y** (**X** é filho à esquerda de **Y**)

Nó **Y** passa a ser filho à direita de **X**

Diminui  $h_{SAE}$  (podendo aumentar  $h_{SAD}$ )

Deve **manter a ordenação** da ABB: **SAE  $\leq$  Raiz  $<$  SAD**

# Rotação à direita (LL)

Consiste em "**trocar a posição**" de um nó **X** com seu pai à direita **Y** (**X** é filho à esquerda de **Y**)

Nó **Y** passa a ser filho à direita de **X**

Diminui  $h_{\text{SAE}}$  (podendo aumentar  $h_{\text{SAD}}$ )

Deve **manter a ordenação** da ABB: **SAE  $\leq$  Raiz  $<$  SAD**

Ex:



# Rotação à direita (LL)

---

**Processamento genérico de uma rotação à direita:**

Nó raiz da SAE ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

# Rotação à direita (LL)

---

**Processamento genérico de uma rotação à direita:**

Nó raiz da SAE ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

Nó raiz ( $R_1$ ) passa a ser filho à direita de  $R_2$ , ou seja, passa a ser nó raiz da SAD

$$R_2 \leq R_1$$

# Rotação à direita (LL)

---

## Processamento genérico de uma rotação à direita:

Nó raiz da SAE ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

Nó raiz ( $R_1$ ) passa a ser filho à direita de  $R_2$ , ou seja, passa a ser nó raiz da SAD

$$R_2 \leq R_1$$

SAD da árvore original ( $SAD_1$ ) permanece à direita de  $R_1$

$$R_1 < SAD_1$$

# Rotação à direita (LL)

---

## Processamento genérico de uma rotação à direita:

Nó raiz da SAE ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

Nó raiz ( $R_1$ ) passa a ser filho à direita de  $R_2$ , ou seja, passa a ser nó raiz da SAD

$$R_2 \leq R_1$$

SAD da árvore original ( $SAD_1$ ) permanece à direita de  $R_1$

$$R_1 < SAD_1$$

SAE da subárvore à esquerda ( $SAE_2$ ) permanece à esquerda de  $R_2$

$$SAE_2 \leq R_2$$

# Rotação à direita (LL)

## Processamento genérico de uma rotação à direita:

Nó raiz da SAE ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

Nó raiz ( $R_1$ ) passa a ser filho à direita de  $R_2$ , ou seja, passa a ser nó raiz da SAD

$$R_2 \leq R_1$$

SAD da árvore original ( $SAD_1$ ) permanece à direita de  $R_1$

$$R_1 < SAD_1$$

SAE da subárvore à esquerda ( $SAE_2$ ) permanece à esquerda de  $R_2$

$$SAE_2 \leq R_2$$

SAD da subárvore à esquerda ( $SAD_2$ ) passa a ser SAE de  $R_1$

Mantém posição de  $SAD_2$  em relação à  $R_1$  e  $R_2$  ( $R_2 < SAD_2 \leq R_1$ )

# Rotação à direita (LL)

Esquematicamente:



$$SAE_2 \leq R_2 < SAD_2 \leq R_1 < SAD_1$$

# Rotação à direita (LL)

## Implementação em C:

```
int rot_dir(Arv * pai) {  
    if (*pai != NULL && (*pai)->sae != NULL) {  
        Arv temp = (*pai)->sae; // Temp = SAE1  
        (*pai)->sae = temp->sad; // SAE1 = SAD2  
        temp->sad = *pai; // SAD2 = Raiz  
        (*pai)->fb = 0;  
        temp->fb = 0;  
        *pai = temp; // Raiz = SAE1  
        return 1;  
    }  
    return 0;  
}
```

# Rotação à direita (LL)

---

## Quando aplicar?

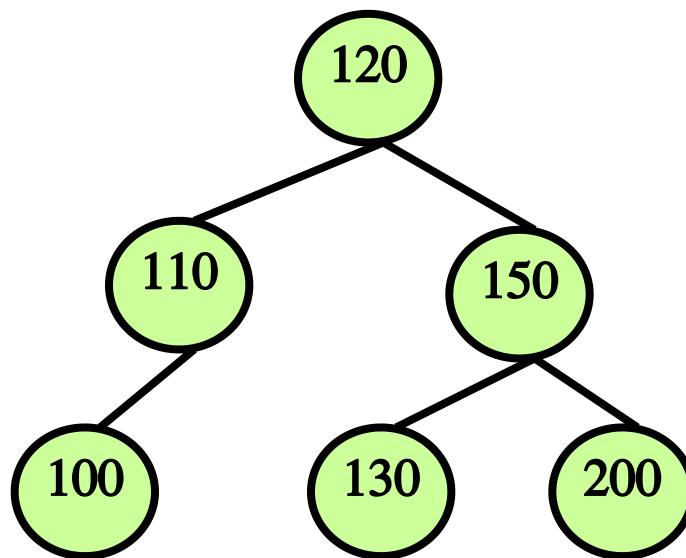
Nó raiz (**pai**) ficou com **FB = +2** (**SAE<sub>1</sub>** é mais alta) e  
Raiz da **SAE<sub>1</sub>** (**filho**) ficou com **FB = +1** (**SAE<sub>2</sub>** é mais alta)

# Rotação à direita (LL) Quando aplicar? Nó raiz (**pai**) ficou com **FB = +2** (**SAE<sub>1</sub>** é mais alta) e Raiz da **SAE<sub>1</sub>** (**filho**) ficou com **FB = +1** (**SAE<sub>2</sub>** é mais alta) **Exemplo:** inserção de um nó na subárvore esquerda de um ascendente esquierdo com FB = +1 A diagram showing a binary tree node labeled 'B' at the bottom left. Above it, another node labeled 'C' is connected by a horizontal line to the right side of node 'B'. To the left of node 'B', the text 'FB = 0' is written. To the right of node 'C', the text 'FB = +1' is written. ▶ 63

# Rotação à direita (LL) Quando aplicar? Nó raiz (**pai**) ficou com **FB = +2** (SAE<sub>1</sub> é mais alta) e Raiz da SAE<sub>1</sub> (**filho**) ficou com **FB = +1** (SAE<sub>2</sub> é mais alta) **Exemplo:** inserção de um nó na subárvore esquerda de um ascendente esquierdo com FB = +1 The diagram illustrates a left-left (LL) rotation in a 2-3 tree. It shows two stages of a node insertion. **Initial State (Left):** A red circle labeled "A" has a blue circle labeled "B" as its child. Circle "B" has a blue circle labeled "C" as its child. The label "FB = +2" is above the connection between A and B. The label "FB = +1" is above the connection between B and C. The label "FB = 0" is below each of the three circles. **Final State (Right):** A red circle labeled "A" is now the root. It has a blue circle labeled "B" as its child, and a blue circle labeled "C" as its right sibling. The label "FB = 0" is below each of the three circles. A curved red arrow labeled "Rotação de B à direita" points from the initial state to the final state. ▶ 64

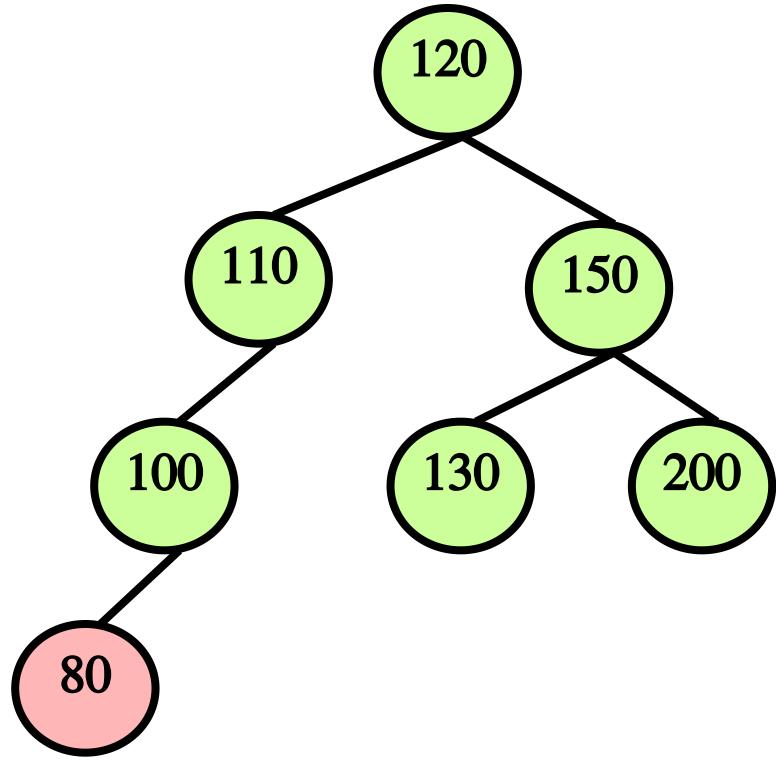
# Exercício

Insira o elemento **80** na árvore AVL abaixo e faça a rotação necessária para manter a ordenação e o balanceamento.



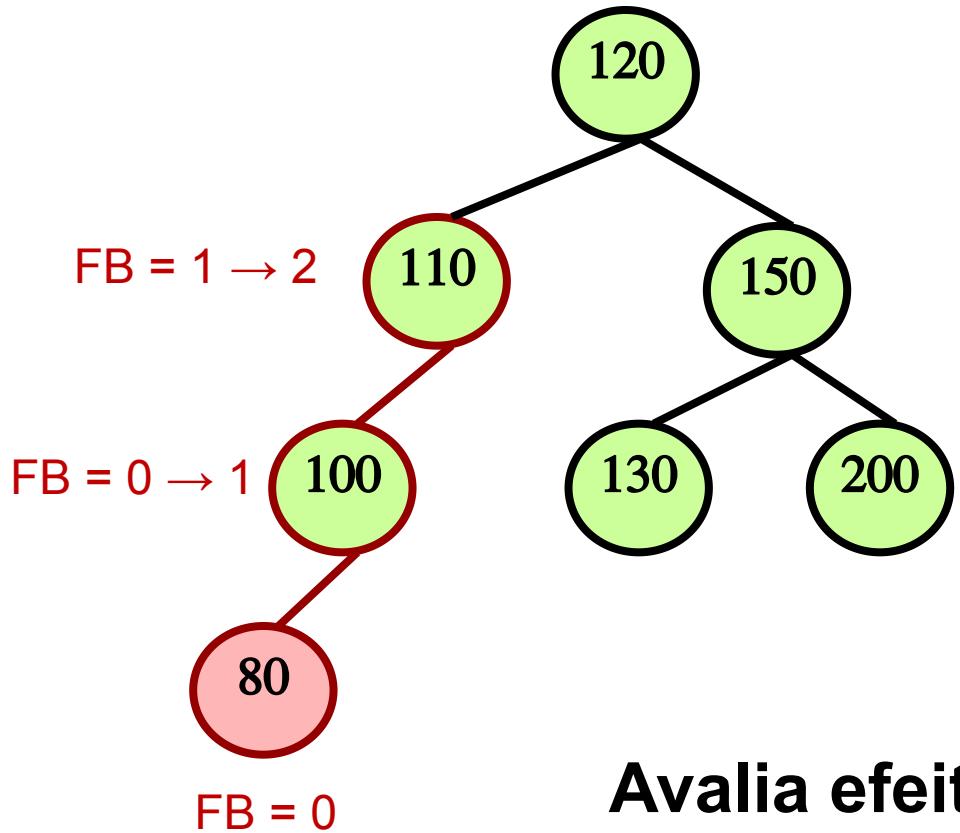
# Resolução

---



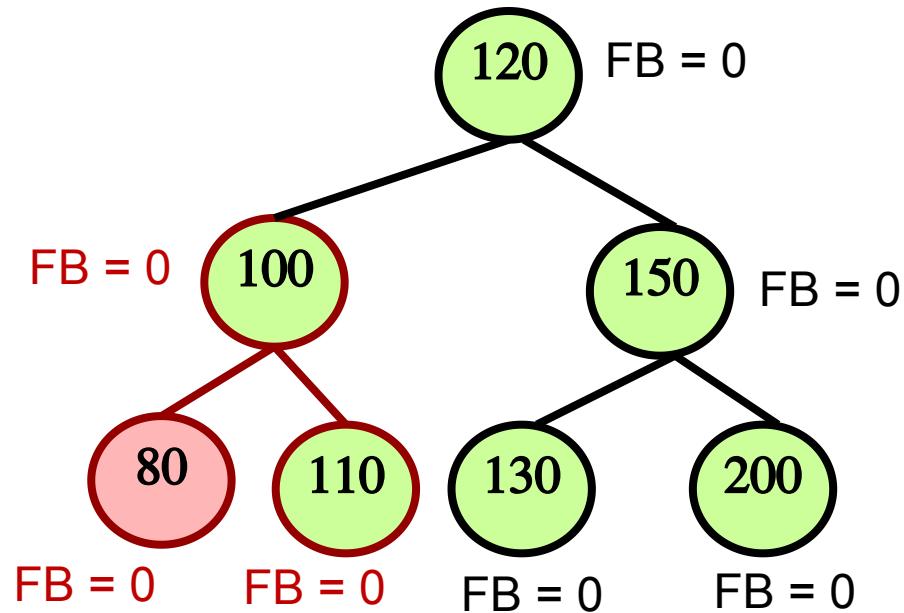
**Inseção de um *novo* nó  
na árvore com valor 80**

# Resolução



**Avalia efeito nos nós antecedentes  
do novo nó até identificar ponto de  
balanceamento**

# Resolução



**Rotaciona subárvore  
selecionada para a direita**

# Rotação à esquerda (RR)

---

Consiste em "**trocar a posição**" de um nó  $X$  com seu pai à esquerda  $Y$  ( $X$  é filho à direita de  $Y$ )

Nó  $Y$  passa a ser filho à esquerda de  $X$

Diminui  $h_{SAD}$  (podendo aumentar  $h_{SAE}$ )

Deve **manter a ordenação** da ABB: **SAE  $\leq$  Raiz  $<$  SAD**

# Rotação à esquerda (RR)

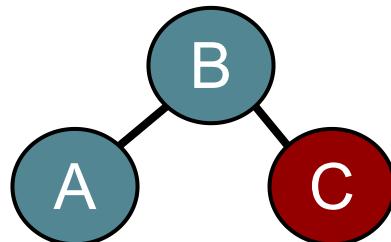
Consiste em "**trocar a posição**" de um nó **X** com seu pai à esquerda **Y** (**X** é filho à direita de **Y**)

Nó **Y** passa a ser filho à esquerda de **X**

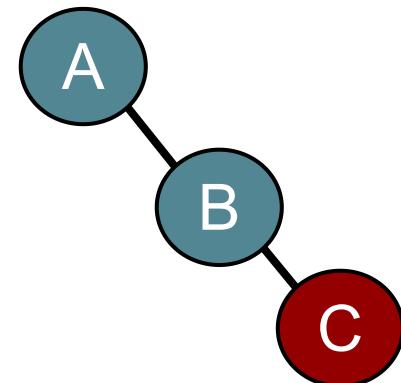
Diminui  $h_{SAD}$  (podendo aumentar  $h_{SAE}$ )

Deve **manter a ordenação** da ABB: **SAE  $\leq$  Raiz  $<$  SAD**

Ex:



 **Rotação de C à esquerda**



# Rotação à esquerda (RR)

---

**Processamento genérico de uma rotação à esquerda:**

Nó raiz da SAD ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

# Rotação à esquerda (RR)

---

**Processamento genérico de uma rotação à esquerda:**

Nó raiz da SAD ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

Nó raiz ( $R_1$ ) passa a ser filho à esquerda de  $R_2$ , ou seja, passa a ser nó raiz da SAE

$$R_1 < R_2$$

# Rotação à esquerda (RR)

---

**Processamento genérico de uma rotação à esquerda:**

Nó raiz da SAD ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

Nó raiz ( $R_1$ ) passa a ser filho à esquerda de  $R_2$ , ou seja, passa a ser nó raiz da SAE

$$R_1 < R_2$$

SAE da árvore original ( $SAE_1$ ) permanece à esquerda de  $R_1$

$$SAE_1 \leq R_1$$

# Rotação à esquerda (RR)

---

## Processamento genérico de uma rotação à esquerda:

Nó raiz da SAD ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

Nó raiz ( $R_1$ ) passa a ser filho à esquerda de  $R_2$ , ou seja, passa a ser nó raiz da SAE

$$R_1 < R_2$$

SAE da árvore original ( $SAE_1$ ) permanece à esquerda de  $R_1$

$$SAE_1 \leq R_1$$

SAD da subárvore à direita ( $SAD_2$ ) permanece à direita de  $R_2$

$$R_2 < SAD_2$$

# Rotação à esquerda (RR)

---

**Processamento genérico de uma rotação à esquerda:**

Nó raiz da SAD ( $R_2$ ) vai para a posição do nó raiz ( $R_1$ )

Nó raiz ( $R_1$ ) passa a ser filho à esquerda de  $R_2$ , ou seja, passa a ser nó raiz da SAE

$$R_1 < R_2$$

SAE da árvore original ( $SAE_1$ ) permanece à esquerda de  $R_1$

$$SAE_1 \leq R_1$$

SAD da subárvore à direita ( $SAD_2$ ) permanece à direita de  $R_2$

$$R_2 < SAD_2$$

SAE da subárvore à direita ( $SAE_2$ ) passa a ser SAD de  $R_1$ ,

Mantém posição de  $SAE_2$  em relação à  $R_1$  e  $R_2$  ( $R_1 < SAE_2 \leq R_2$ )

# Rotação à esquerda (RR)

Esquematicamente:



$$SAE_1 \leq R_1 < SAE_2 \leq R_2 < SAD_2$$

# Rotação à esquerda (RR)

## Implementação em C:

```
int rot_esq(Arv *pai) {  
    if (*pai != NULL && (*pai)->sad != NULL) {  
        Arv temp = (*pai)->sad; // Temp = SAD1  
        (*pai)->sad = temp->sae; // SAD1 = SAE2  
        temp->sae = *pai; // SAE2 = Raiz  
        (*pai)->fb = 0;  
        temp->fb = 0;  
        *pai = temp; // Raiz = SAD1  
        return 1;  
    }  
    return 0;  
}
```

# Rotação à esquerda (RR)

---

## Quando aplicar?

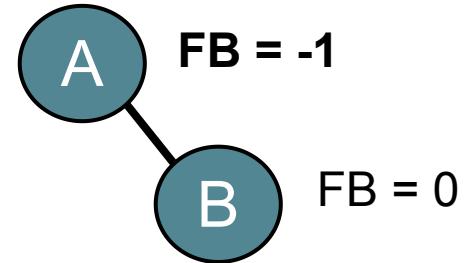
Nó raiz (**pai**) ficou com **FB = -2** (**SAD<sub>1</sub>** é mais alta) e  
Raiz da **SAD<sub>1</sub>** (**filho**) ficou com **FB = -1** (**SAD<sub>2</sub>** é mais alta)

# Rotação à esquerda (RR)

## Quando aplicar?

Nó raiz (**pai**) ficou com **FB = -2** (**SAD<sub>1</sub>** é mais alta) e  
Raiz da **SAD<sub>1</sub>** (**filho**) ficou com **FB = -1** (**SAD<sub>2</sub>** é mais alta)

**Exemplo:** inserção de um nó na subárvore direita de um ascendente direito com FB = -1

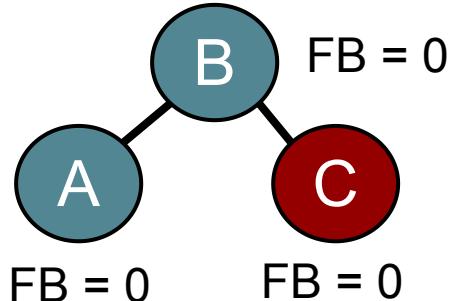


# Rotação à esquerda (RR)

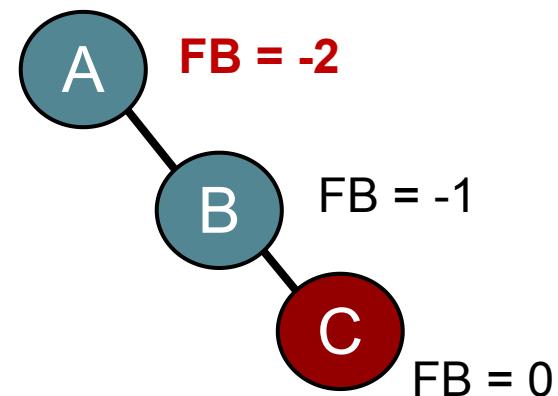
## Quando aplicar?

Nó raiz (**pai**) ficou com **FB = -2** (**SAD<sub>1</sub>** é mais alta) e  
Raiz da **SAD<sub>1</sub>** (**filho**) ficou com **FB = -1** (**SAD<sub>2</sub>** é mais alta)

**Exemplo:** inserção de um nó na subárvore direita de um ascendente direito com FB = -1

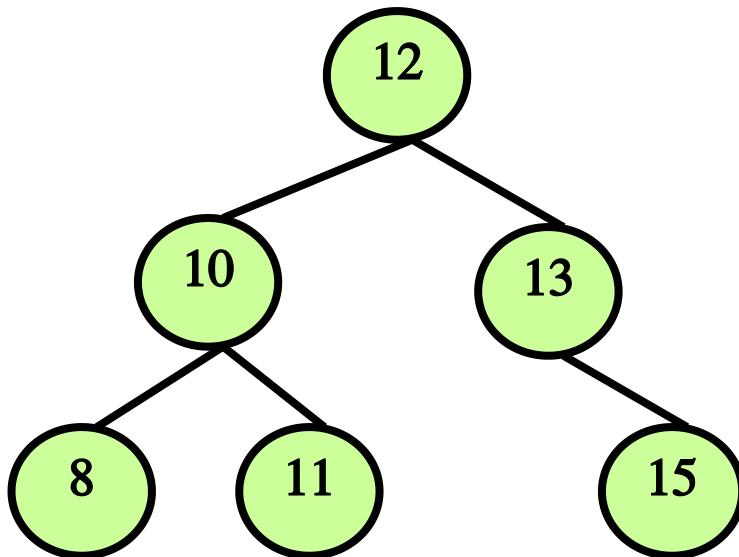


➡️ **Rotação de B à esquerda**



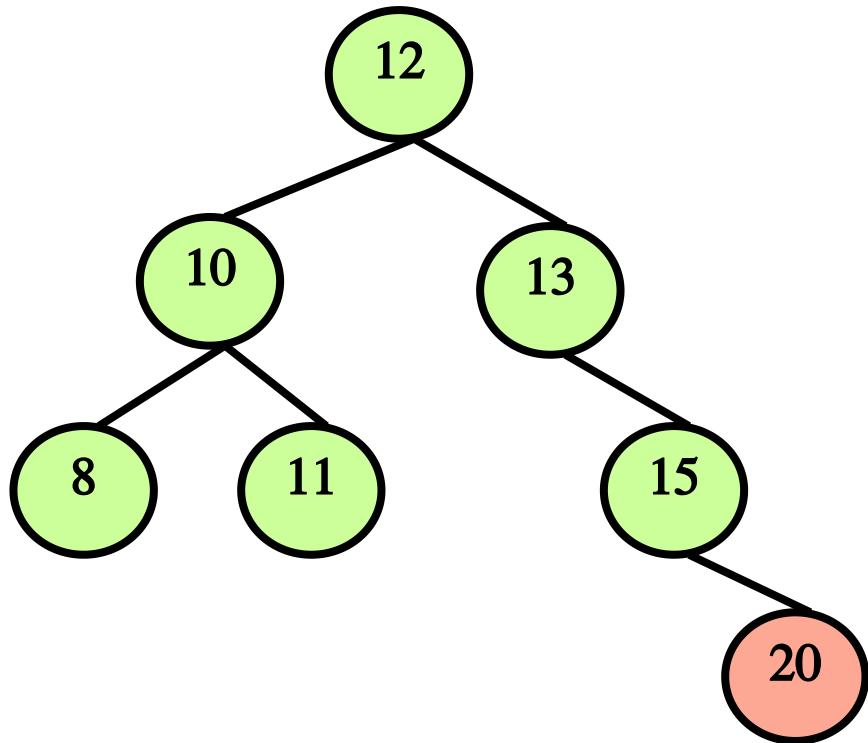
# Exercício

Insira o elemento **20** na árvore AVL abaixo e faça a rotação necessária para manter a ordenação e o balanceamento.



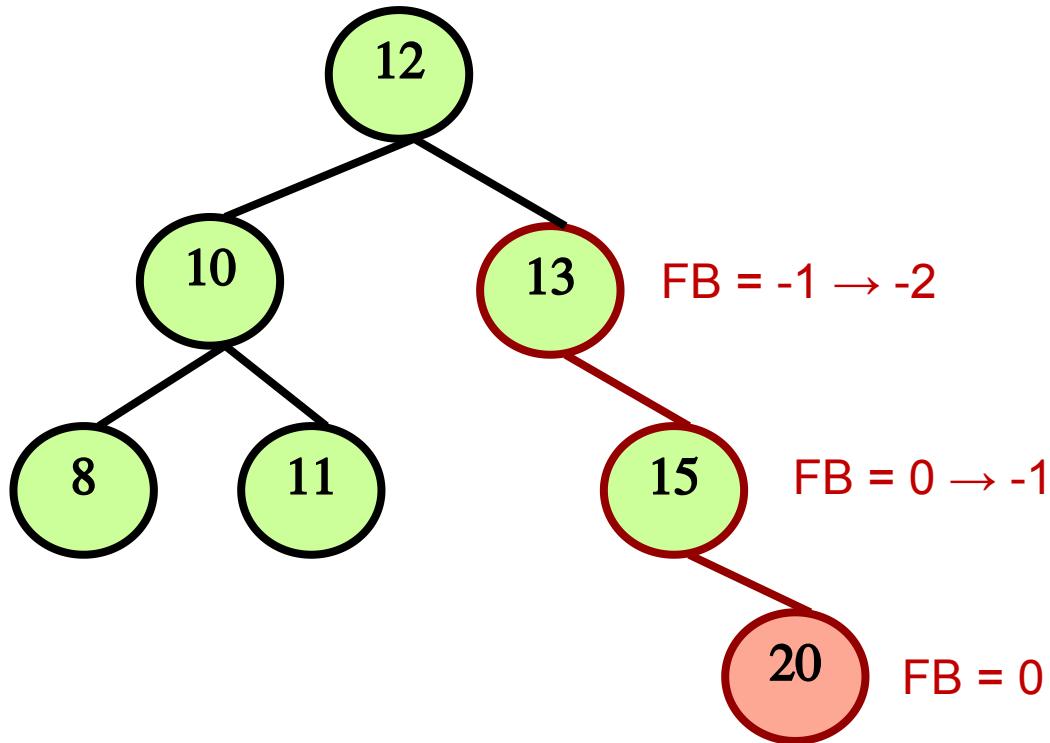
# Resolução

---



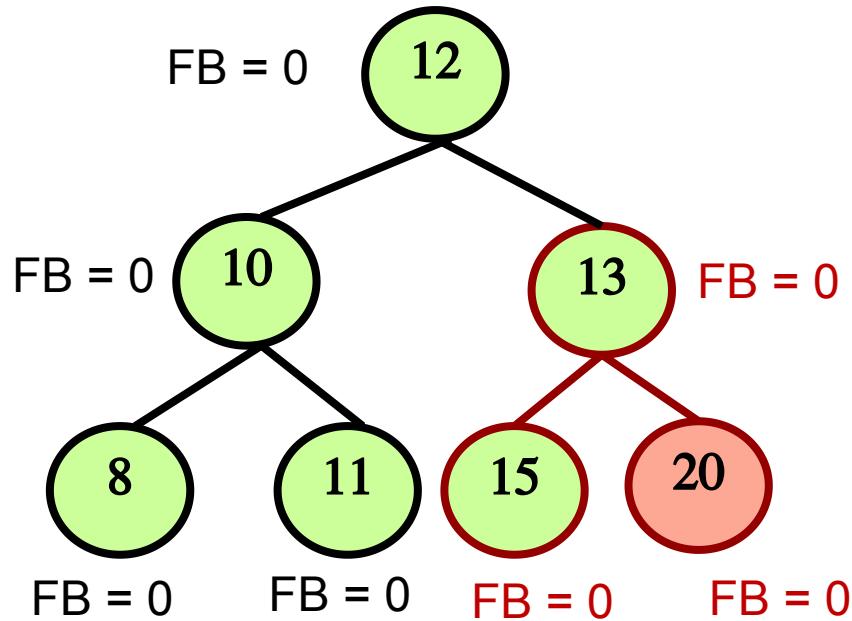
**Inseção de um *novo nó*  
na árvore com valor 20**

# Resolução



**Avalia efeito nos nós antecedentes  
do novo nó até identificar ponto de  
balanceamento**

# Resolução



**Rotaciona subárvore  
selecionada à esquerda**

# Rotação dupla direita-esquerda (RL)

---

Nome deriva do **resultado final** da rotação:

**Nó pai** fica à **direita** e

**Nó filho** (neto do pai) fica à **esquerda**

# Rotação dupla direita-esquerda (RL)

---

Nome deriva do **resultado final** da rotação:

**Nó pai** fica à **direita** e

**Nó filho** (neto do pai) fica à **esquerda**

Consiste em realizar uma **rotação à esquerda seguida de uma rotação à direita**

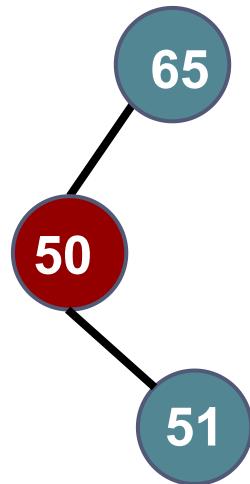
**1<sup>a</sup> rotação (para esquerda)** é feita sobre a subárvore à esquerda (**filho à direita** de SAE **passa a ser pai**)

**2<sup>a</sup> rotação (para direita)** é feita sobre a árvore (**nó raiz passa a ser irmão à direita**)

# Rotação dupla direita-esquerda (RL)

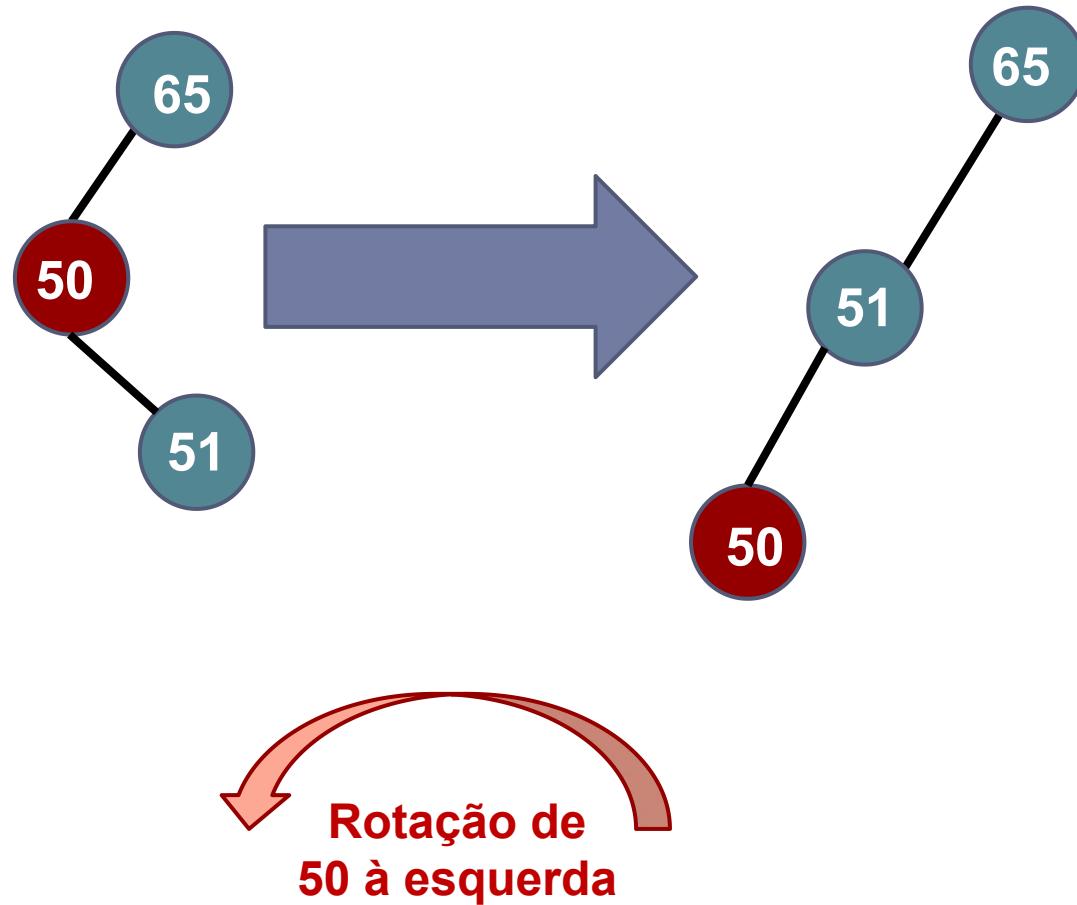
---

**Exemplo:**



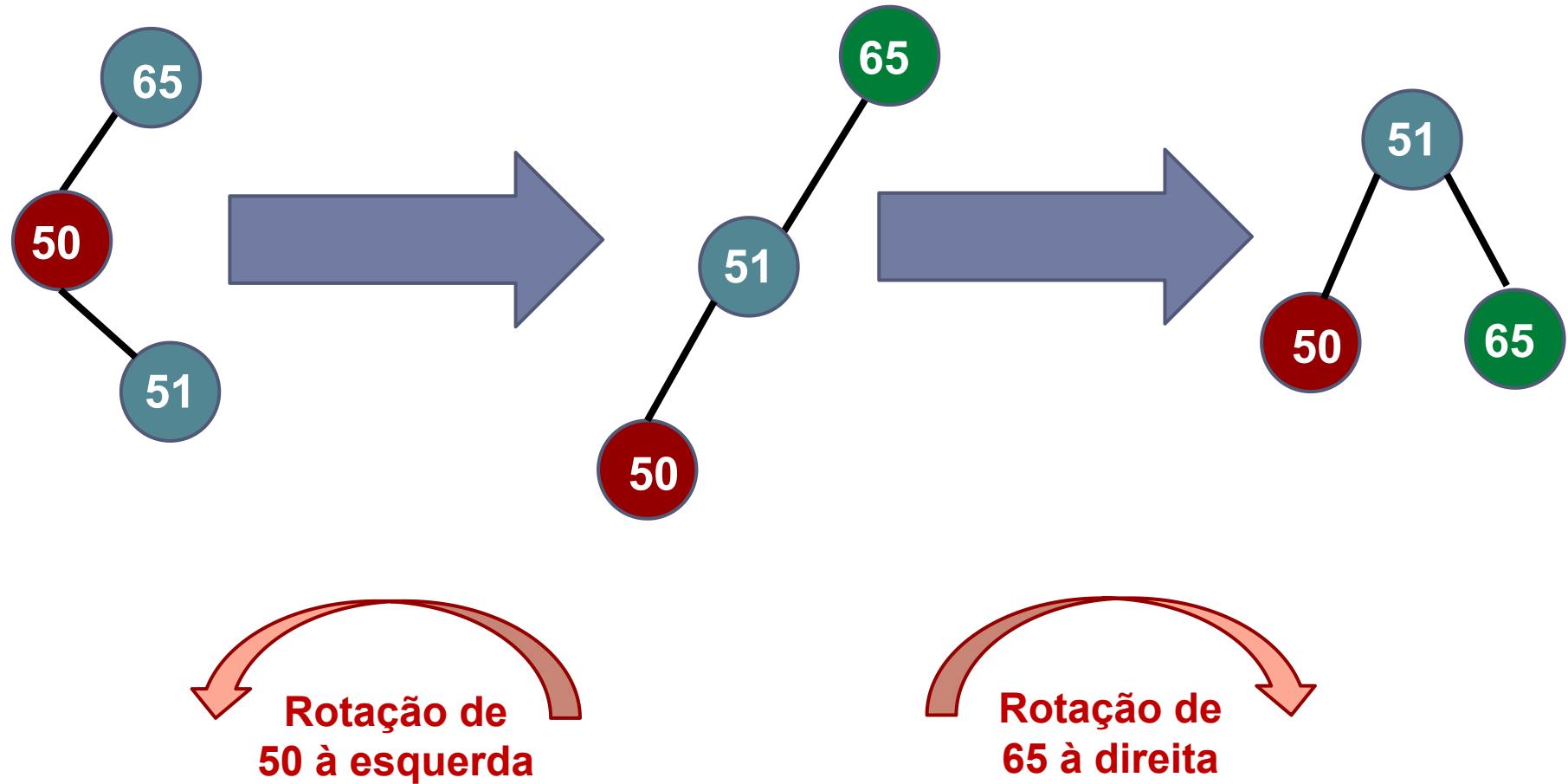
# Rotação dupla direita-esquerda (RL)

**Exemplo:**



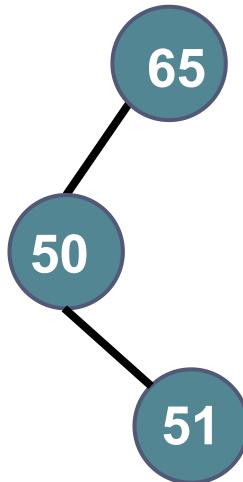
# Rotação dupla direita-esquerda (RL)

**Exemplo:**



# Rotação dupla direita-esquerda (RL)

**Por que preciso da rotação RL?**



# Rotação dupla direita-esquerda (RL)

**Por que preciso da rotação RL?**



# Rotação dupla direita-esquerda (RL)

Por que preciso da rotação RL?



**Não mudou a altura da árvore**

# Rotação dupla direita-esquerda (RL)

## Implementação em C:

```
int rot_dir_esq(Arv *pai) {  
    if (*pai != NULL && (*pai)->sae != NULL) {  
        // Rotação à direita a partir da raiz da árvore  
        Arv filho_esq = (*pai)->sae;  
  
        // Rotação à esquerda a partir da raiz de SAE (filho_esq)  
        Arv neto_dir = filho_esq->sad;  
        filho_esq->sad = neto_dir->sae;  
        neto_dir->sae = filho_esq;  
  
        (*pai)->sae = neto_dir->sad;  
        neto_dir->sad = *pai;  
    }  
}
```

...

# Rotação dupla direita-esquerda (RL)

---

...

```
// Ajuste dos fatores de balanceamento (FB)
if (neto_dir->fb == -1) {
    (*pai)->fb = 0;
    filho_esq->fb = 1;
} else if (neto_dir->fb == 1) {
    (*pai)->fb = -1;
    filho_esq->fb = 0;
} else { // neto_dir->fb == 0
    (*pai)->fb = 0;
    filho_esq->fb = 0;
}
neto_dir->fb = 0; *pai = neto_dir; return 1;
}
```

# Rotação dupla direita-esquerda (RL)

---

## Quando aplicar?

Nó raiz (**pai**) ficou com **FB = +2** (**SAE<sub>1</sub>** é mais alta) e  
Raiz da **SAE<sub>1</sub>** (**filho**) ficou com **FB = -1** (**SAD<sub>2</sub>** é mais alta)

# Rotação dupla esquerda\_direita (LR)

---

Nome deriva do **resultado final** da rotação:

**Nó pai** fica à **esquerda** e

**Nó filho** (neto do pai) fica à **direita**

# Rotação dupla esquerda\_direita (LR)

---

Nome deriva do **resultado final** da rotação:

**Nó pai** fica à **esquerda** e

**Nó filho** (neto do pai) fica à **direita**

Consiste em realizar uma **rotação à direita seguida de uma rotação à esquerda** da árvore

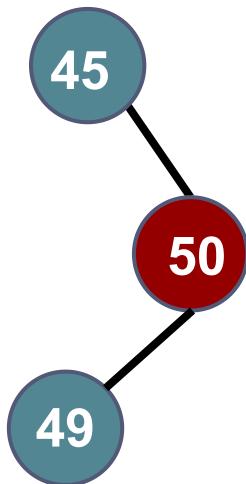
**1<sup>a</sup> rotação (para direita)** é feita sobre a subárvore à direita (**filho à esquerda de SAD passa a ser pai**)

**2<sup>a</sup> rotação (para esquerda)** é feita sobre a árvore (**nó raiz passa a ser irmão à esquerda**)

# Rotação dupla esquerda\_direita (LR)

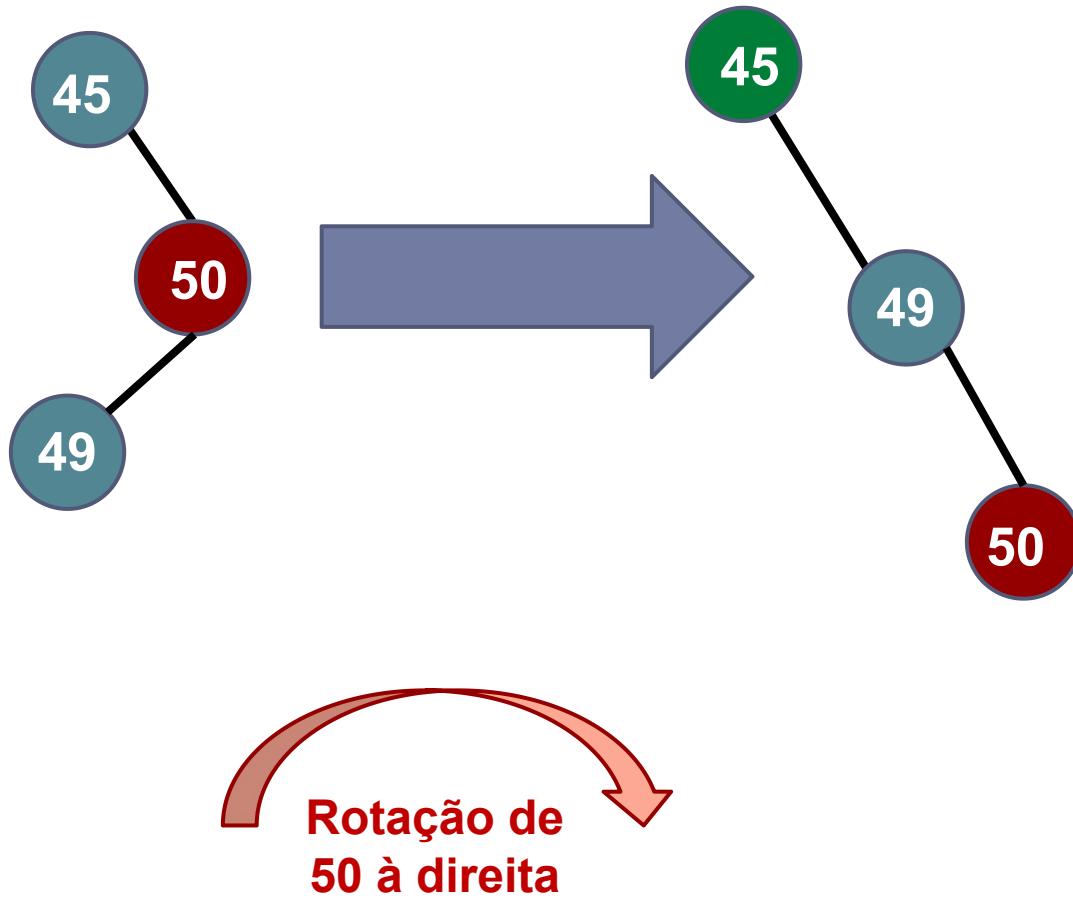
---

**Exemplo:**



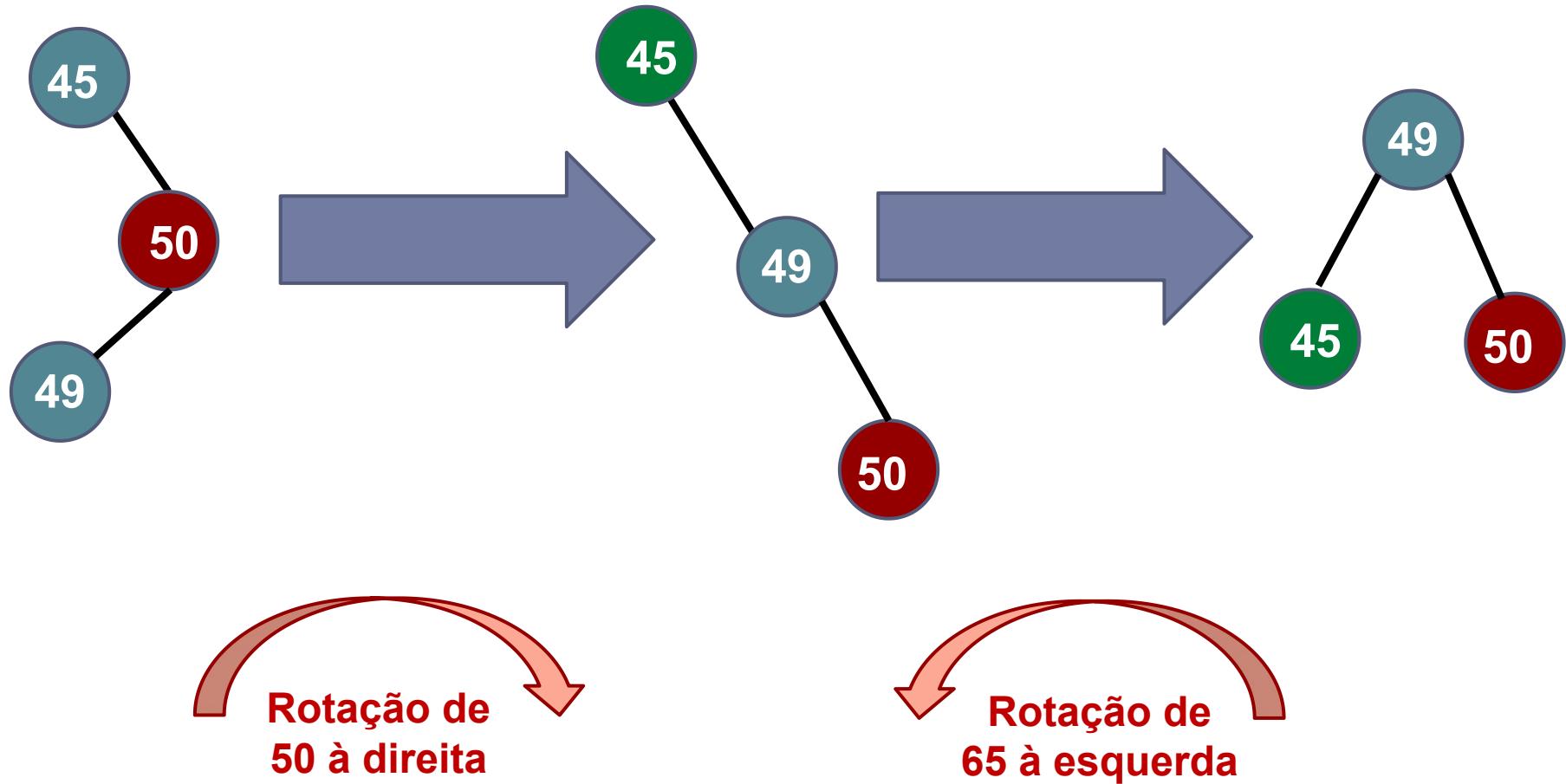
# Rotação dupla esquerda\_direita (LR)

**Exemplo:**



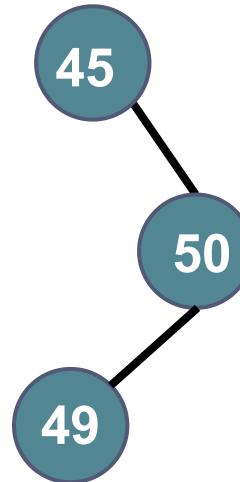
# Rotação dupla esquerda\_direita (LR)

Exemplo:



# Rotação dupla esquerda\_direita (LR)

**Por que preciso da rotação LR?**



# Rotação dupla esquerda\_direita (LR)

**Por que preciso da rotação LR?**



# Rotação dupla esquerda\_direita (LR)

Por que preciso da rotação LR?



**Não mudou a altura da árvore**

# Rotação dupla esquerda\_direita (LR)

## Implementação em C:

```
int rot_esq_dir(Arv *pai) {  
    if (*pai != NULL && (*pai)->sad != NULL) {  
        // Rotação à esquerda a partir da raiz da árvore  
        Arv filho_dir = (*pai)->sad;  
  
        // Rotação à direita a partir da raiz de SAD (filho_dir)  
        Arv neto_esq = filho_dir->sae;  
        filho_dir->sae = neto_esq->sad;  
        neto_esq->sad = filho_dir;  
  
        (*pai)->sad = neto_esq->sae;  
        neto_esq->sae = *pai;  
    }  
}
```

...

# Rotação dupla esquerda\_direita (LR)

---

...

```
// Ajuste dos fatores de balanceamento (FB)
if (neto_esq->fb == -1) {
    (*pai)->fb = 1;
    filho_dir->fb = 0;
} else if (neto_esq->fb == 1) {
    (*pai)->fb = 0;
    filho_dir->fb = -1;
} else { // neto_esq->fb == 0
    (*pai)->fb = 0;
    filho_dir->fb = 0;
}
neto_esq->fb = 0; *pai = neto_esq; return 1;
}
```

# Rotação dupla esquerda\_direita (LR)

---

## Quando aplicar?

Nó raiz (**pai**) ficou com **FB = -2** (**SAE<sub>1</sub>** é mais alta) e  
Raiz da **SAE<sub>1</sub>** (**filho**) ficou com **FB = +1** (**SAD<sub>2</sub>** é mais alta)

# Árvores AVL: operações básicas

---

## Listas de operações:

***cria\_arvore***: retorna uma AVL vazia

***arvore\_vazia***: verifica se a AVL está vazia

***insere\_AVL***: insere um novo nó na AVL de modo a garantir a ordenação e o balanceamento

***remove\_AVL***: remove um elemento da AVL, preservando a ordenação e o balanceamento

***busca\_bin***: busca um elemento na ABB, aproveitando-se da ordenação

***exibe\_arvore***: percorre a ABB e imprime cada nó

***exibe\_ordenado***: percorre a ABB e imprime os nós de forma ordenada

***libera\_arvore***: libera todo o espaço de memória alocado pela ABB

# Árvores AVL: operações básicas

---

## Listas de operações:

**cria\_arvore:** retorna uma AVL vazia

**arvore\_vazia:** verifica se a AVL está vazia

**insere\_AVL:** insere um novo nó na AVL de modo a garantir a ordenação e o **balanceamento**

**remove\_AVL:** remove um elemento da AVL, preservando a ordenação e o **balanceamento**

**busca\_bin:** busca um elemento na ABB, aproveitando-se da ordenação

**exibe\_arvore:** percorre a ABB e imprime cada nó

**exibe\_ordenado:** percorre a ABB e imprime os nós de forma ordenada

**libera\_arvore:** libera todo o espaço de memória alocado pela ABB

# Especificação do TAD AVL

---

Operação ***insere\_AVL***:

**Entrada:** endereço do endereço da árvore (**referência**), o elemento a ser inserido e o endereço de um flag

**Pré-condição:** a árvore existir

**Processo:** insere o elemento, de acordo com valor da sua chave, como nó folha; verifica o fator de balanceamento dos nós antecessores, efetuando o balanceamento para que a árvore continue AVL, caso seja necessário.

**Saída:** 1 - se operação bem sucedida ou 0 - caso contrário

**Pós-condição:** árvore de entrada com um nó a mais

# Árvores AVL: implementação

*int insere\_AVL(Arv \* A, reg elem, int \* Bal)*

**SE** árvore inválida **ENTÃO** //  $A = \text{NULL}$

Retorna 0;

**FIM\_SE**

**SE** árvore vazia **ENTÃO**

Aloca memória para o novo nó;

Campo **info** do novo nó = *elem*;

Campos **sae** e **sad** do novo nó = **NULL**;

Campo **fb** do novo nó = 0;

Faz a árvore apontar para o novo nó; //  $*A = \text{Novo}$

Conteúdo de *Bal* = 1; // Ativa verificação de balanceamento

•••

# Árvores AVL: implementação

---

•••

**SENÃO SE** chave de elem  $\leq$  chave da raiz **ENTÃO**

*Inserir elemento na subárvore à esquerda;*

**SE** Bal retornado = 1 **ENTÃO** // Verifica necessidade de balanceamento

**SE** campo **fb** da raiz = 1 **ENTÃO**

*Balancear a esquerda;*

*Conteúdo de Bal = 0; // Árvore balanceada*

**SENÃO SE** campo fb da raiz = 0 **ENTÃO**

*Campo fb da raiz = 1;*

**SENÃO** // Campo fb da raiz = -1

*Campo fb da raiz = 0;*

*Conteúdo de Bal = 0; // Árvore balanceada*

**FIM\_SE**

**FIM\_SE**

•••

# Árvores AVL: implementação

---

•••

**SENÃO SE** chave de elem ≤ chave da raiz **ENTÃO**

*Inserir elemento na subárvore à esquerda;*

**SE** Bal retornado = 1 **ENTÃO** // Verifica necessidade de balanceamento

**SE** campo **fb** da raiz = 1 **ENTÃO**

*Balancear a esquerda;*

*Conteúdo de Bal = 0; // Árvore balanceada*

**SENÃO SE** campo fb da raiz = 0 **ENTÃO**

*Campo fb da raiz = 1;*

**SENÃO** // Campo fb da raiz = -1

*Campo fb da raiz = 0;*

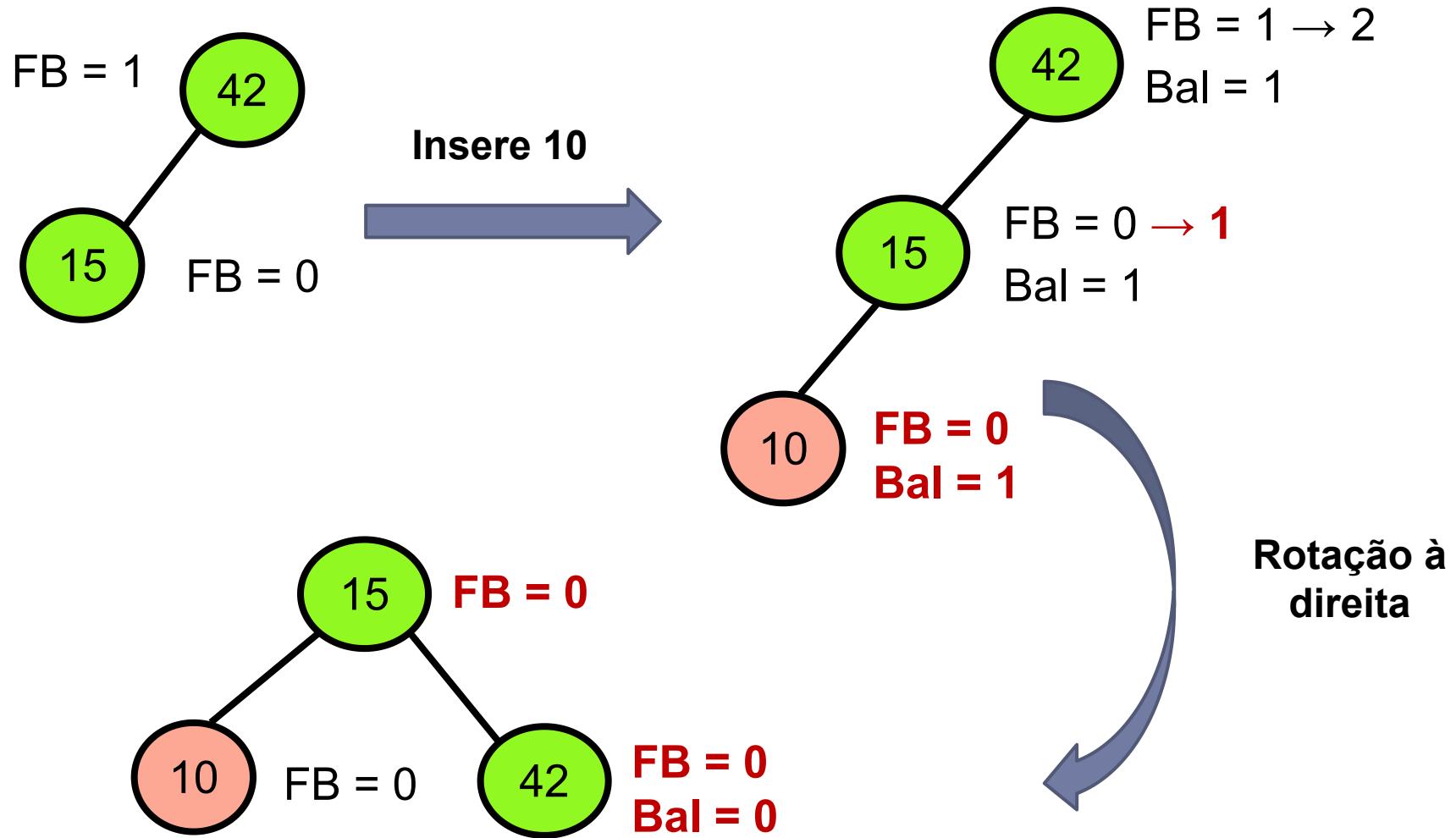
*Conteúdo de Bal = 0; // Árvore balanceada*

**FIM\_SE**

**FIM\_SE**

•••

# Árvores AVL: implementação



# Árvores AVL: implementação

---

•••

**SENÃO SE** chave de elem ≤ chave da raiz **ENTÃO**

*Inserir elemento na subárvore à esquerda;*

**SE** Bal retornado = 1 **ENTÃO** // Verifica necessidade de balanceamento

**SE** campo **fb** da raiz = 1 **ENTÃO**

*Balancear a esquerda;*

*Conteúdo de Bal = 0; // Árvore balanceada*

**SENÃO SE** campo **fb** da raiz = 0 **ENTÃO**

*Campo **fb** da raiz = 1;*

**SENÃO** // Campo **fb** da raiz = -1

*Campo **fb** da raiz = 0;*

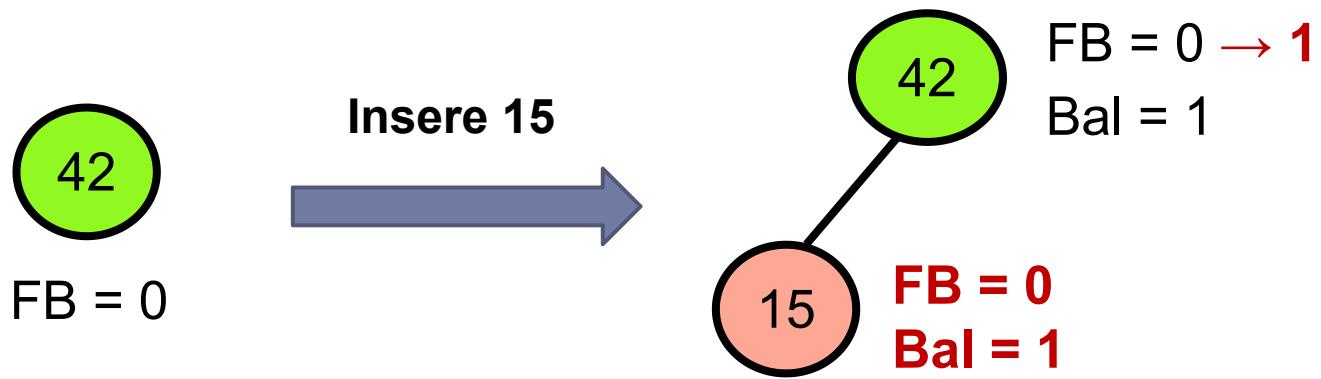
*Conteúdo de Bal = 0; // Árvore balanceada*

**FIM\_SE**

**FIM\_SE**

•••

# Árvores AVL: implementação



# Árvores AVL: implementação

---

•••

**SENÃO SE** chave de elem ≤ chave da raiz **ENTÃO**

*Inserir elemento na subárvore à esquerda;*

**SE** Bal retornado = 1 **ENTÃO** // Verifica necessidade de balanceamento

**SE** campo **fb** da raiz = 1 **ENTÃO**

*Balancear a esquerda;*

*Conteúdo de Bal = 0; // Árvore balanceada*

**SENÃO SE** campo fb da raiz = 0 **ENTÃO**

*Campo fb da raiz = 1;*

**SENÃO** // Campo fb da raiz = -1

*Campo fb da raiz = 0;*

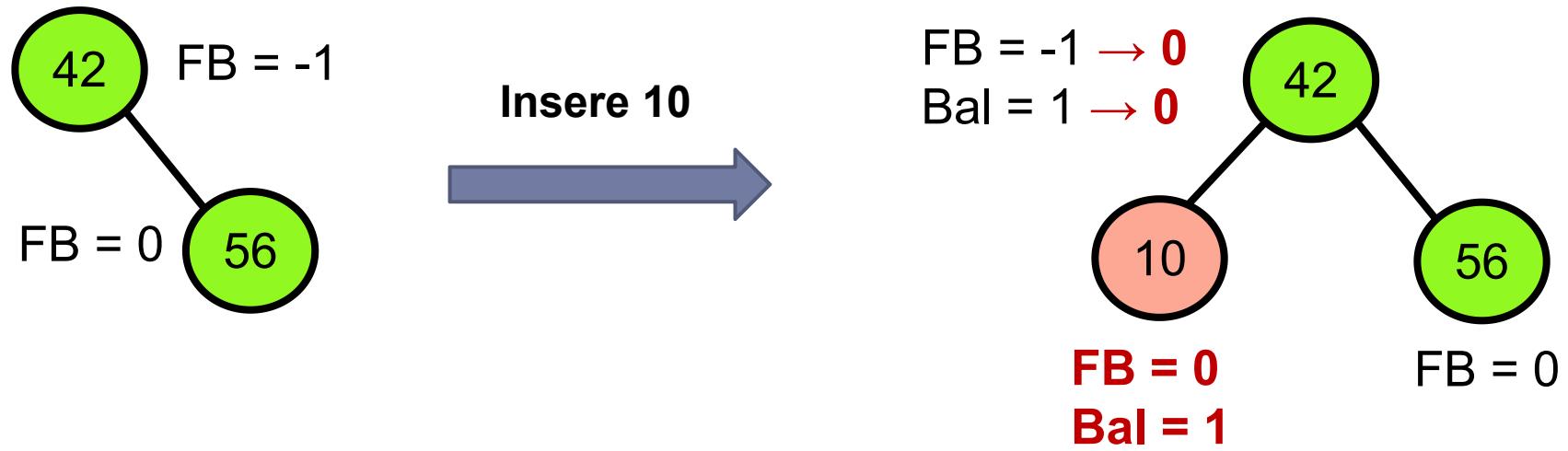
*Conteúdo de Bal = 0; // Árvore balanceada*

**FIM\_SE**

**FIM\_SE**

•••

# Árvores AVL: implementação



# Árvores AVL: implementação

---

...

**SENÃO** // chave do elem > chave da raiz  
**Inserir elemento na subárvore à direita;**  
**SE** Bal retornado = 1 **ENTÃO** // Verifica necessidade de balanceamento  
**SE** campo **fb** da raiz = -1 **ENTÃO**  
**Balancear a direita;**  
Conteúdo de Bal = 0; // Árvore balanceada  
**SENÃO SE** campo fb da raiz = 0 **ENTÃO**  
    Campo **fb** da raiz = -1;  
**SENÃO** // Campo fb da raiz = 1  
    Campo **fb** da raiz = 0;  
    Conteúdo de Bal = 0; // Árvore balanceada  
**FIM\_SE**  
**FIM\_SE**  
**FIM\_SE**  
Retorna 1;  
**FIM**

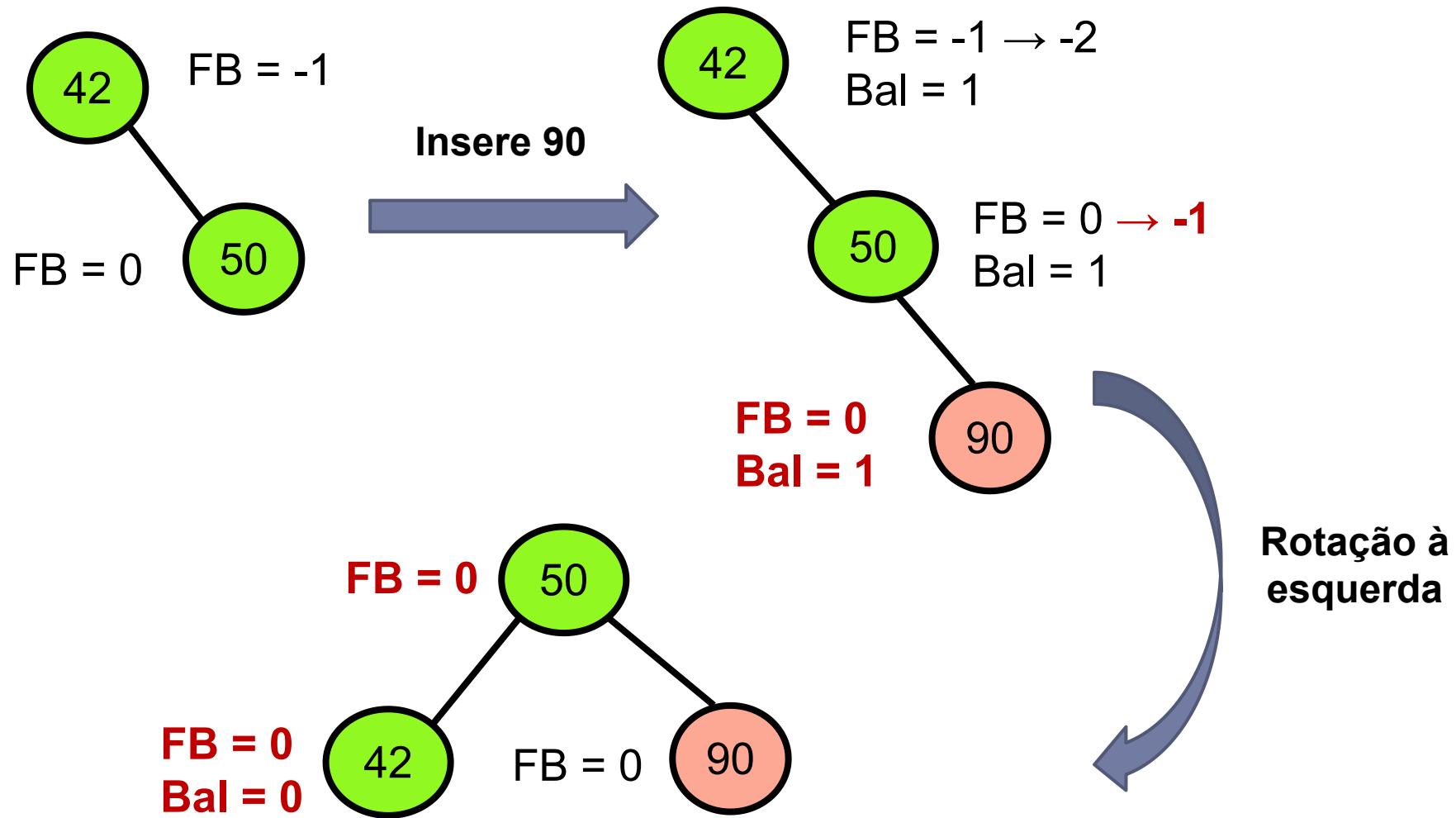
# Árvores AVL: implementação

---

...

**SENÃO** // chave do elem > chave da raiz  
*Inserir elemento na subárvore à direita;*  
**SE** Bal retornado = 1 **ENTÃO** // Verifica necessidade de balanceamento  
**SE** campo **fb** da raiz = -1 **ENTÃO**  
*Balancear a direita;*  
*Conteúdo de Bal = 0; // Árvore balanceada*  
**SENÃO SE** campo fb da raiz = 0 **ENTÃO**  
*Campo **fb** da raiz = -1;*  
**SENÃO** // Campo fb da raiz = 1  
*Campo **fb** da raiz = 0;*  
*Conteúdo de Bal = 0; // Árvore balanceada*  
**FIM\_SE**  
**FIM\_SE**  
**FIM\_SE**  
*Retorna 1;*  
**FIM**

# Árvores AVL: implementação



# Árvores AVL: implementação

---

...

**SENÃO** // chave do elem > chave da raiz

**Inserir elemento na subárvore à direita;**

**SE** Bal retornado = 1 **ENTÃO** // Verifica necessidade de balanceamento

**SE** campo **fb** da raiz = -1 **ENTÃO**

Balancear a direita;

Conteúdo de Bal = 0; // Árvore balanceada

**SENÃO SE** campo **fb** da raiz = 0 **ENTÃO**

Campo **fb** da raiz = -1;

**SENÃO** // Campo **fb** da raiz = 1

Campo **fb** da raiz = 0;

Conteúdo de Bal = 0; // Árvore balanceada

**FIM\_SE**

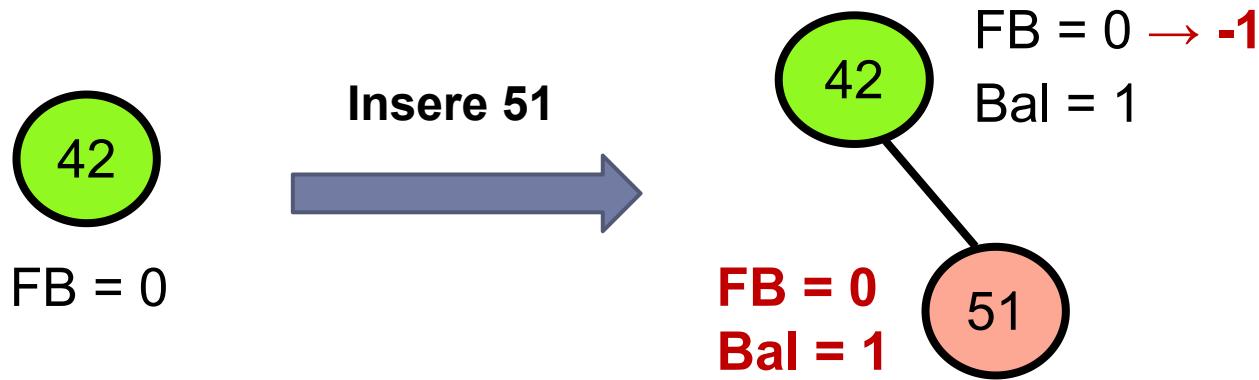
**FIM\_SE**

**FIM\_SE**

Retorna 1;

**FIM**

# Árvores AVL: implementação



# Árvores AVL: implementação

---

...

**SENÃO** // chave do elem > chave da raiz

**Inserir elemento na subárvore à direita;**

**SE** Bal retornado = 1 **ENTÃO** // Verifica necessidade de balanceamento

**SE** campo **fb** da raiz = -1 **ENTÃO**

Balancear a direita;

Conteúdo de Bal = 0; // Árvore balanceada

**SENÃO SE** campo fb da raiz = 0 **ENTÃO**

Campo **fb** da raiz = -1;

**SENÃO** // Campo fb da raiz = 1

Campo **fb** da raiz = 0;

Conteúdo de Bal = 0; // Árvore balanceada

**FIM\_SE**

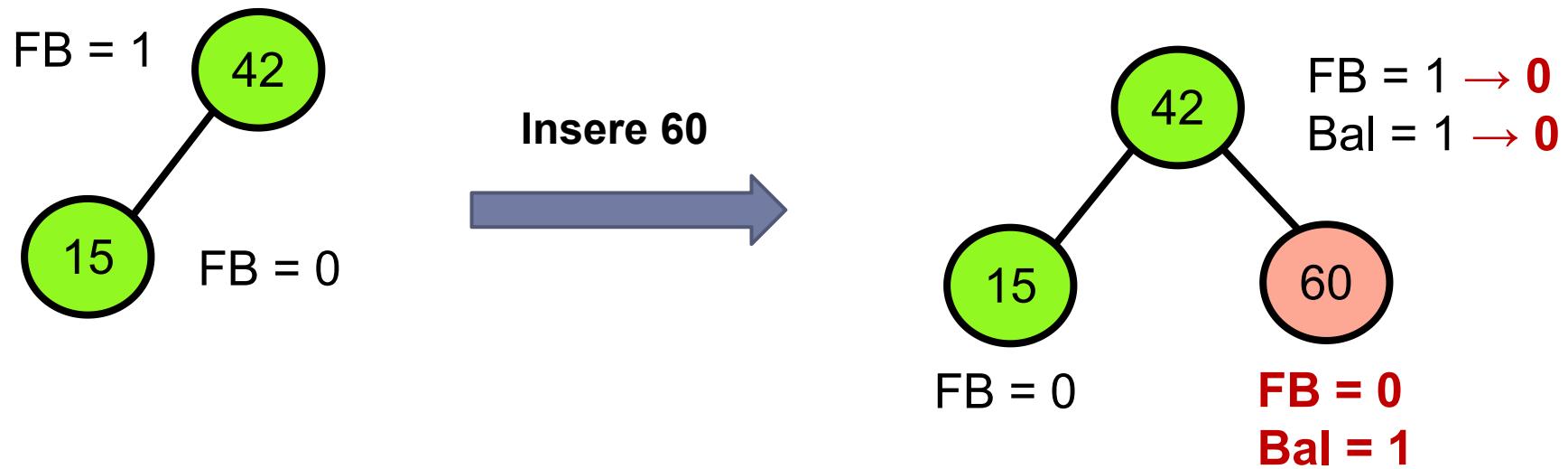
**FIM\_SE**

**FIM\_SE**

Retorna 1;

**FIM**

# Árvores AVL: implementação



# Árvores AVL: implementação

```
int balancear_esq (Arv * A) {
    if ( (*A)->esq->fb > 0 )
        return rot_dir(A);          // FBpai positivo e FBfilho_esq positivo
    else
        return rot_dir_esq(A);    // FBpai positivo e FBfilho_esq negativo
}
```

# Árvores AVL: implementação

```
int balancear_esq (Arv * A) {
    if ( (*A)->esq->fb > 0 )
        return rot_dir(A);          // FBpai positivo e FBfilho_esq positivo
    else
        return rot_dir_esq(A);    // FBpai positivo e FBfilho_esq negativo
}
```

```
int balancear_dir (Arv * A) {
    if ( (*A)->dir->fb < 0 )
        return rot_esq(A);        // FBpai negativo e FBfilho_dir negativo
    else
        return rot_esq_dir(A);   // FBpai negativo e FBfilho_dir positivo
}
```

# Árvores AVL: inserção

---

**Questão:** Como seria a inserção usando a altura no lugar do FB?

# Árvores AVL: inserção

---

**Questão:** Como seria a inserção usando a altura no lugar do FB?

**Executa a inserção ordenada como em uma ABB:**

Se árvore é vazia (raiz = ***NULL***), insere o elemento como raiz

Se chave  $\leq$  raiz da árvore, insere o elemento na SAE

Se chave  $>$  raiz da árvore, insere o elemento na SAD

# Árvores AVL: inserção

**Questão:** Como seria a inserção usando a altura no lugar do FB?

**Executa a inserção ordenada como em uma ABB:**

Se árvore é vazia (raiz = **NULL**), insere o elemento como raiz

Se chave  $\leq$  raiz da árvore, insere o elemento na SAE

Se chave  $>$  raiz da árvore, insere o elemento na SAD

**Realiza o balanceamento no retorno da recursão:**

Calcula a altura das subárvores afetadas pela inserção

$$\heartsuit \quad h_{raiz} = \max(h_{sae}, h_{sad}) + 1$$

Calcula o fator de平衡amento (FB) do nó ( $FB = h_{sae} - h_{sad}$ )

Aplica rotação necessária quando o FB for +2 ou -2

# Árvores AVL: implementação

*int **insere\_AVL2**(Arv \* A, reg elem, int \* Bal)*

**SE** árvore inválida **ENTÃO** // A = NULL

Retorna 0;

**FIM\_SE**

**SE** árvore vazia **ENTÃO**

Aloca memória para o novo nó;

Campo **info** do novo nó = elem;

Campos **sae** e **sad** do novo nó = **NULL**;

Campo **h** do novo nó = 0;

Faz a árvore apontar para o novo nó; // \*A = Novo  
retorna 1; // Sucesso na inserção

•••

# Árvores AVL: implementação

---

•••

**SENÃO SE** chave de elem  $\leq$  chave da raiz **ENTÃO**

*Inserir elemento na subárvore à esquerda;*

**SE** retorno = 0 **ENTÃO** // Falha na inserção

Retorna 0;

**FIM\_SE**

Calcular o FB da raiz;

**SE** FB da raiz  $\geq 2$  **ENTÃO** // Precisa balanceamento

**SE** chave do elem  $\leq$  chave da SAE **ENTÃO**

*Fazer rotação à direita;*

**SENÃO**

*Fazer rotação esquerda-direita;*

**FIM\_SE**

**FIM\_SE**

•••

# Árvores AVL: implementação

...

**SENÃO** // chave de elem > chave da raiz

**Inserir elemento na subárvore à direita;**

**SE** retorno = 0 **ENTÃO** // Falha na inserção

Retorna 0;

**FIM\_SE**

Calcular o FB da raiz;

**SE** FB da raiz ≤ -2 **ENTÃO** // Precisa balanceamento

**SE** chave do elem > chave da SAD **ENTÃO**

**Fazer rotação à esquerda;**

**SENÃO**

**Fazer rotação direita-esquerda;**

**FIM\_SE**

**FIM\_SE**

**FIM\_SE**

...

...

Calcular altura da raiz;

Retorna 1;

**FIM**

# Exercício

---

Considere o seguinte trecho de código do programa principal:

```
int main() {  
    Arv A = cria_arvore();  
    if (A != NULL) {  
        insere_AVL2(&A, 1); insere_AVL2(&A, 2); insere_AVL2(&A, 3);  
        insere_AVL2(&A, 10); insere_AVL2(&A, 4); insere_AVL2(&A, 5);  
        insere_AVL2(&A, 9); insere_AVL2(&A, 0); insere_AVL2(&A, -1);  
    }  
    ...  
}
```

Apresente a árvore gerada após cada comando e, quando houver necessidade de balanceamento, os passos da rotação realizada.

# Resolução

---

*A = cria\_arvore();*

**A → NULL**

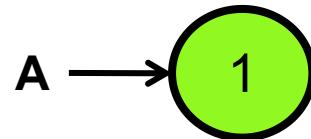
# Resolução

---

*A = cria\_arvore();*

**A → NULL**

*insere\_AVL2(&A, 1);*

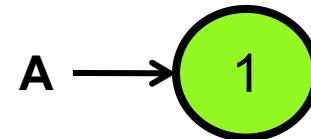


# Resolução

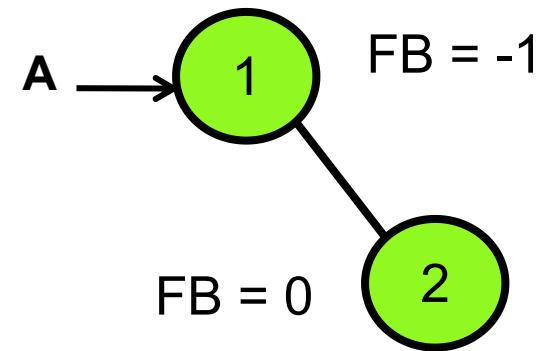
$A = \text{cria\_arvore}();$

$A \longrightarrow \text{NULL}$

$\text{insere\_AVL2}(\&A, 1);$

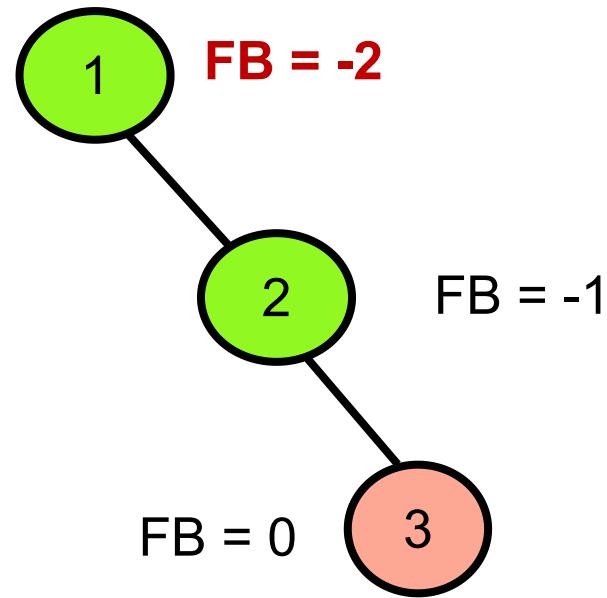


$\text{insere\_AVL2}(\&A, 2);$



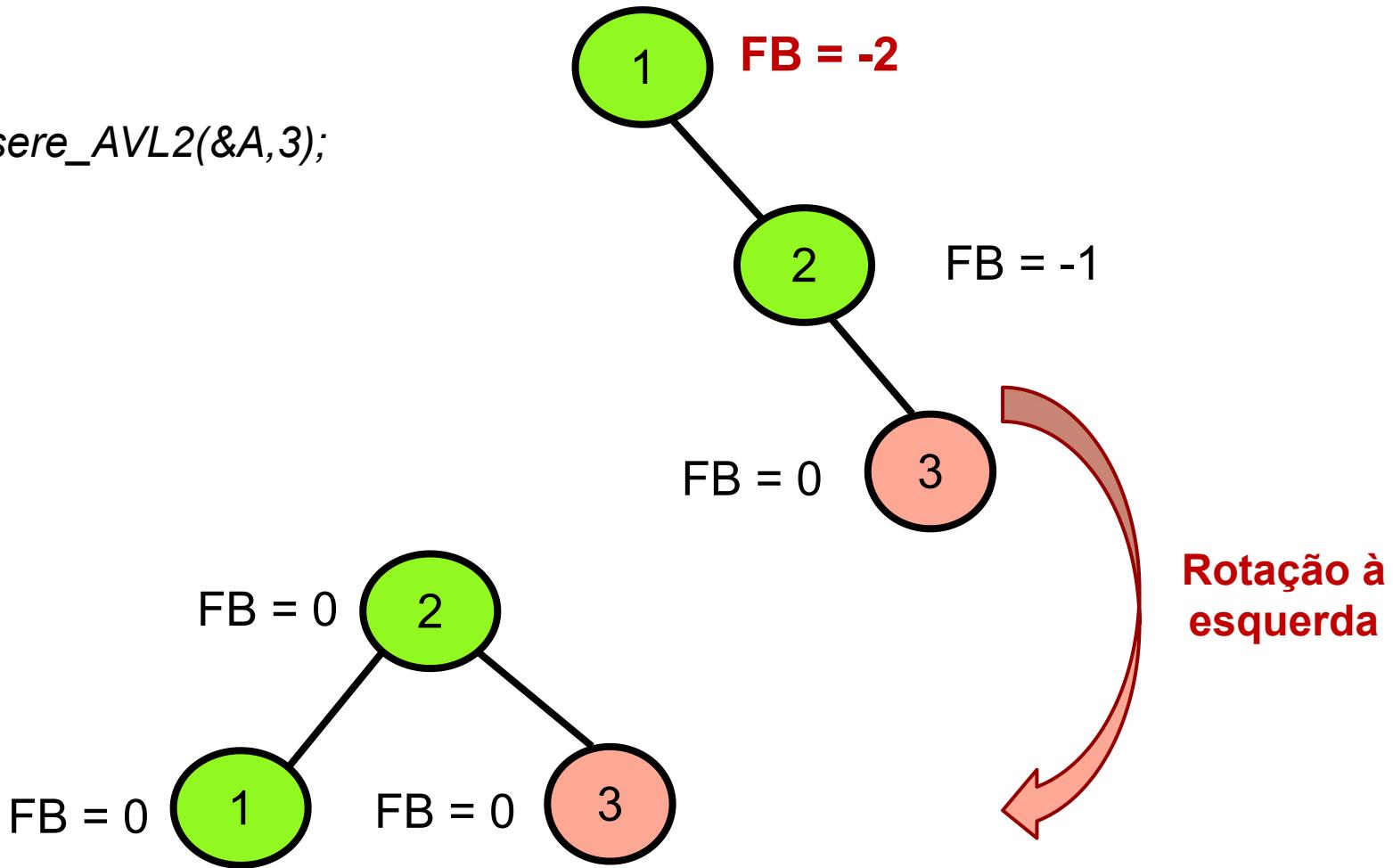
# Resolução

*insere\_AVL2(&A, 3);*



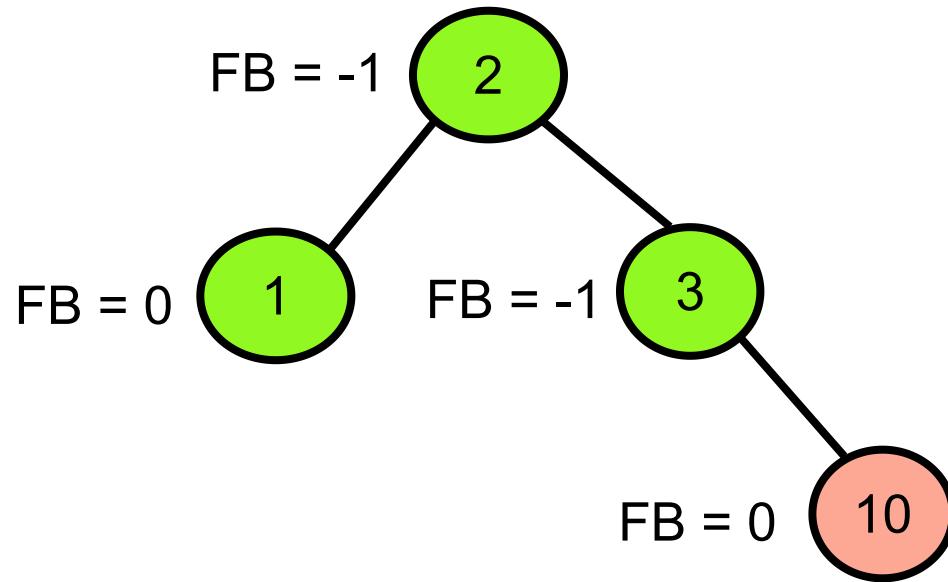
# Resolução

*insere\_AVL2(&A, 3);*



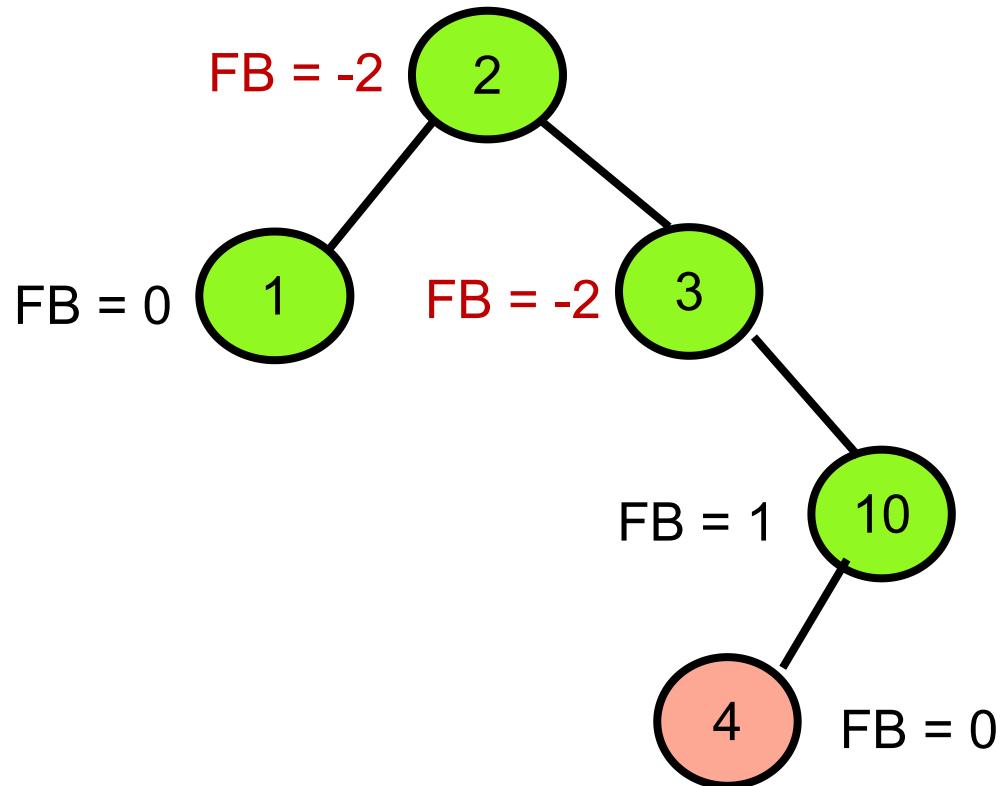
# Resolução

*insere\_AVL2(&A, 10);*



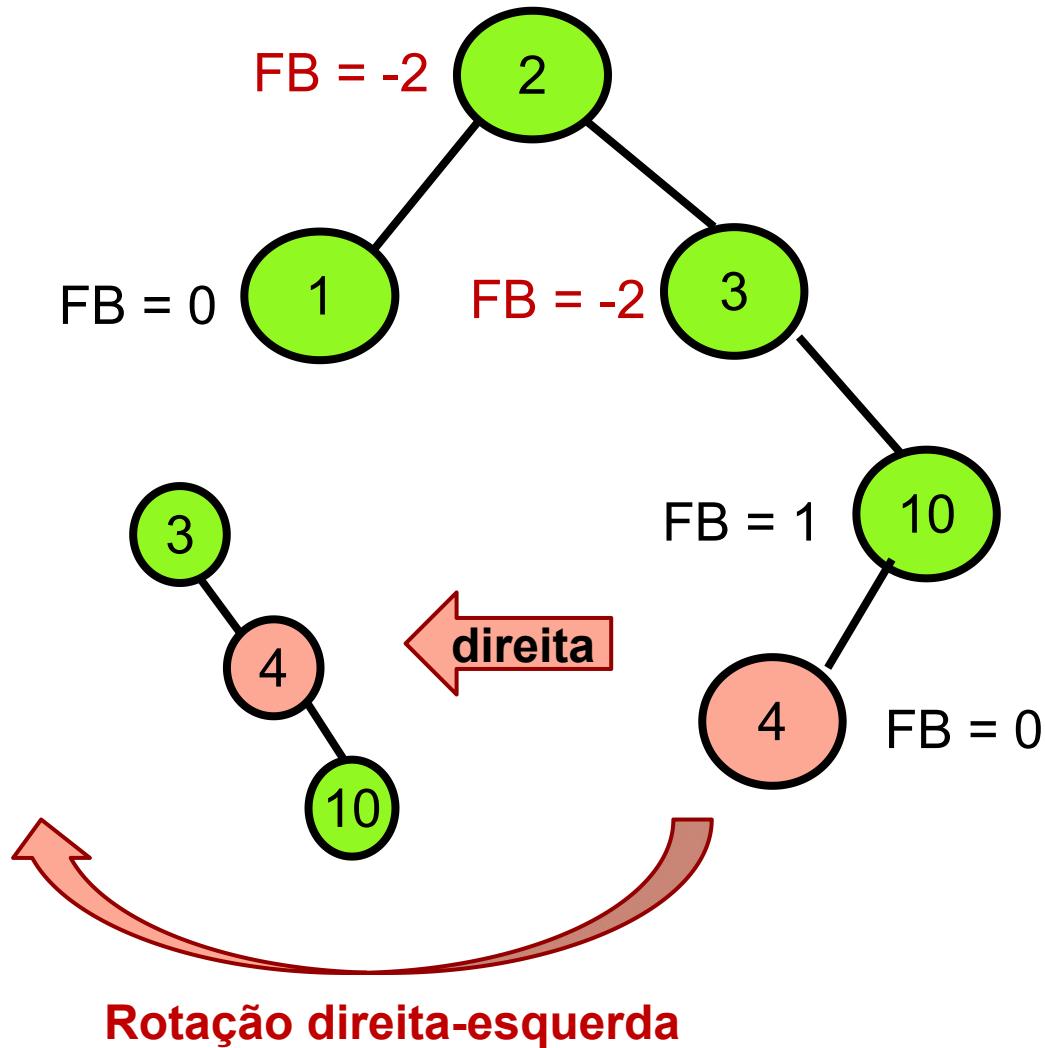
# Resolução

*insere\_AVL2(&A, 10);*



# Resolução

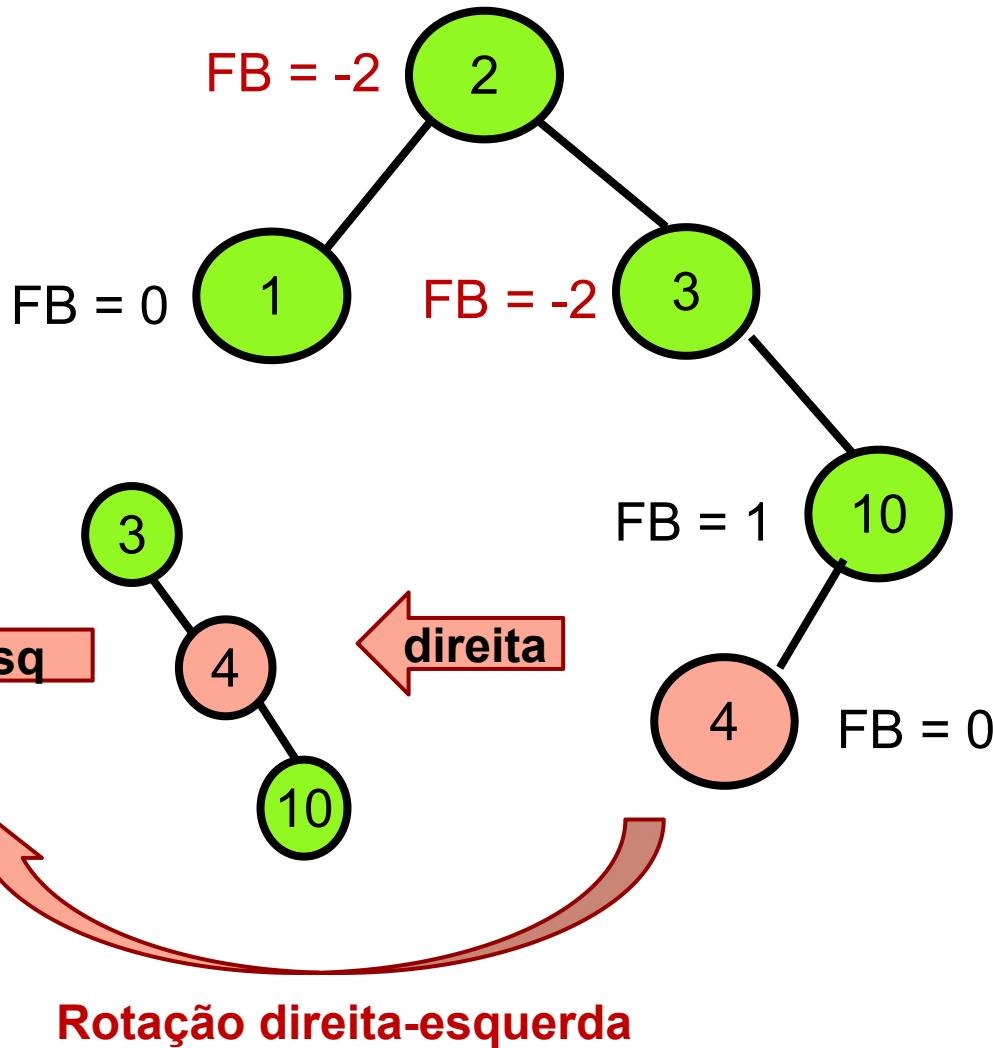
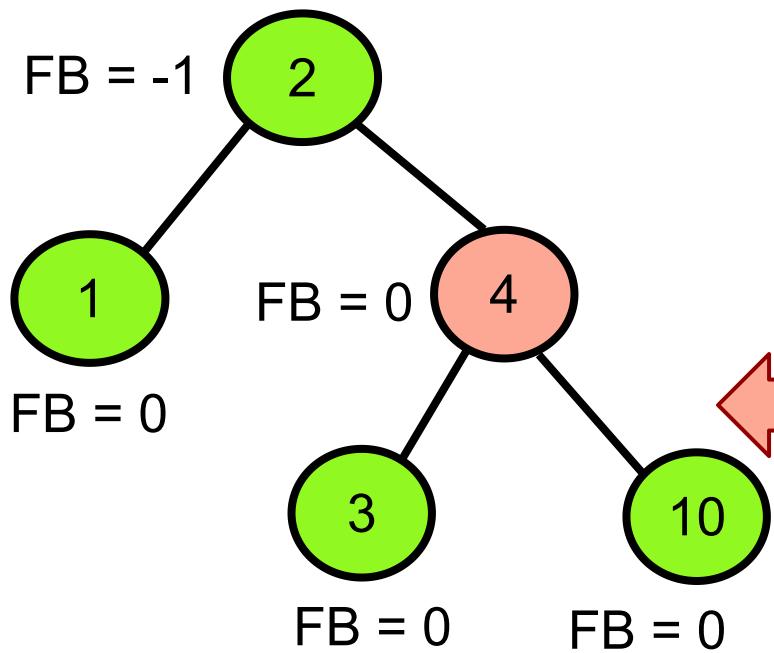
`insere_AVL2(&A, 10);`



Rotação direita-esquerda

# Resolução

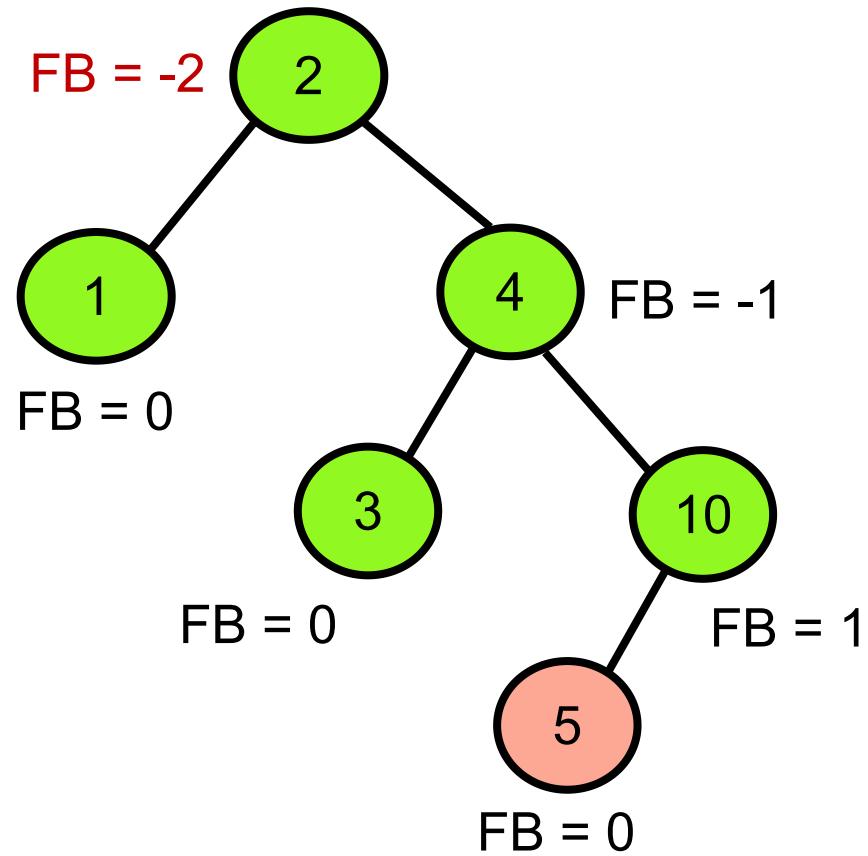
`insere_AVL2(&A, 10);`



Rotação direita-esquerda

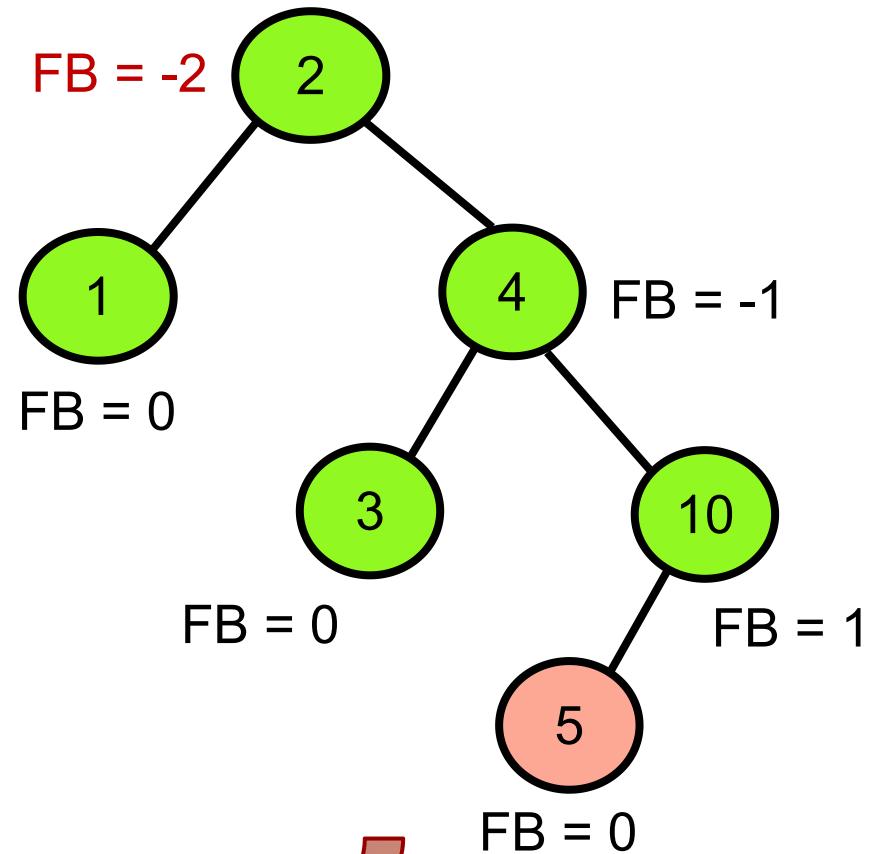
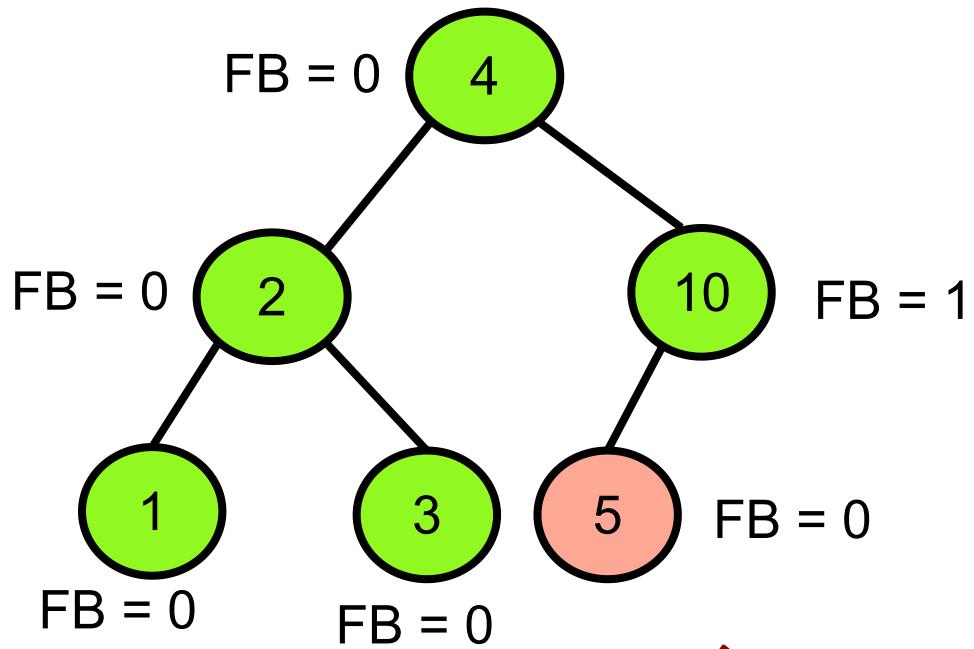
# Resolução

*insere\_AVL2(&A, 5);*



# Resolução

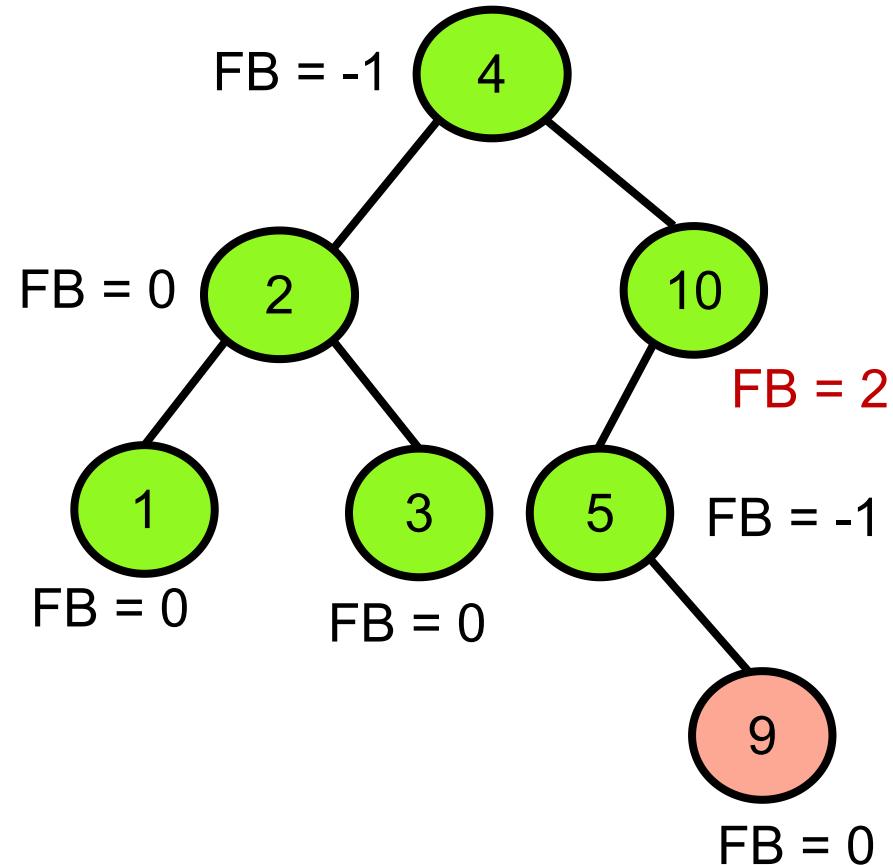
*insere\_AVL2(&A, 5);*



Rotação à esquerda

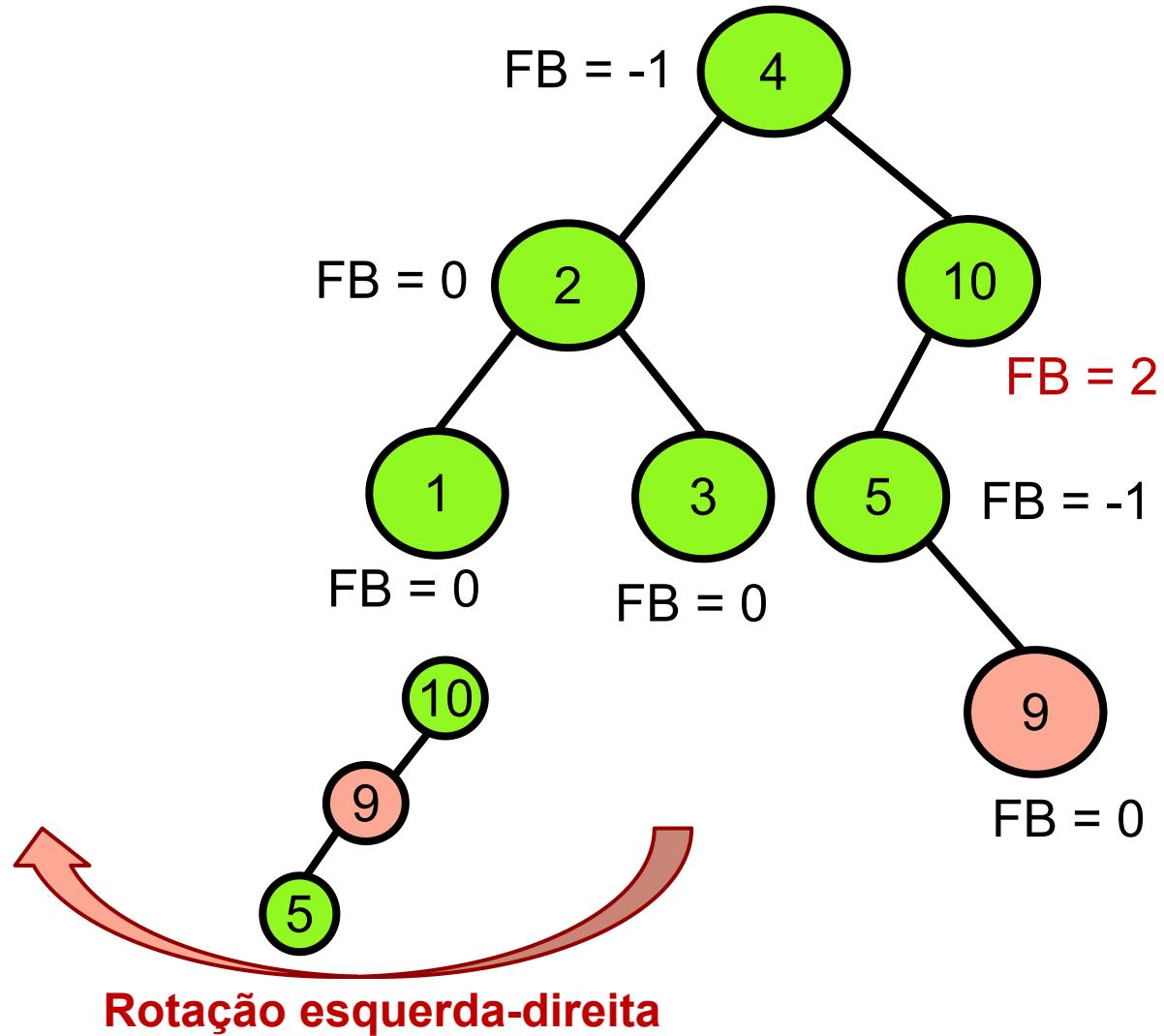
# Resolução

*insere\_AVL2(&A, 9);*



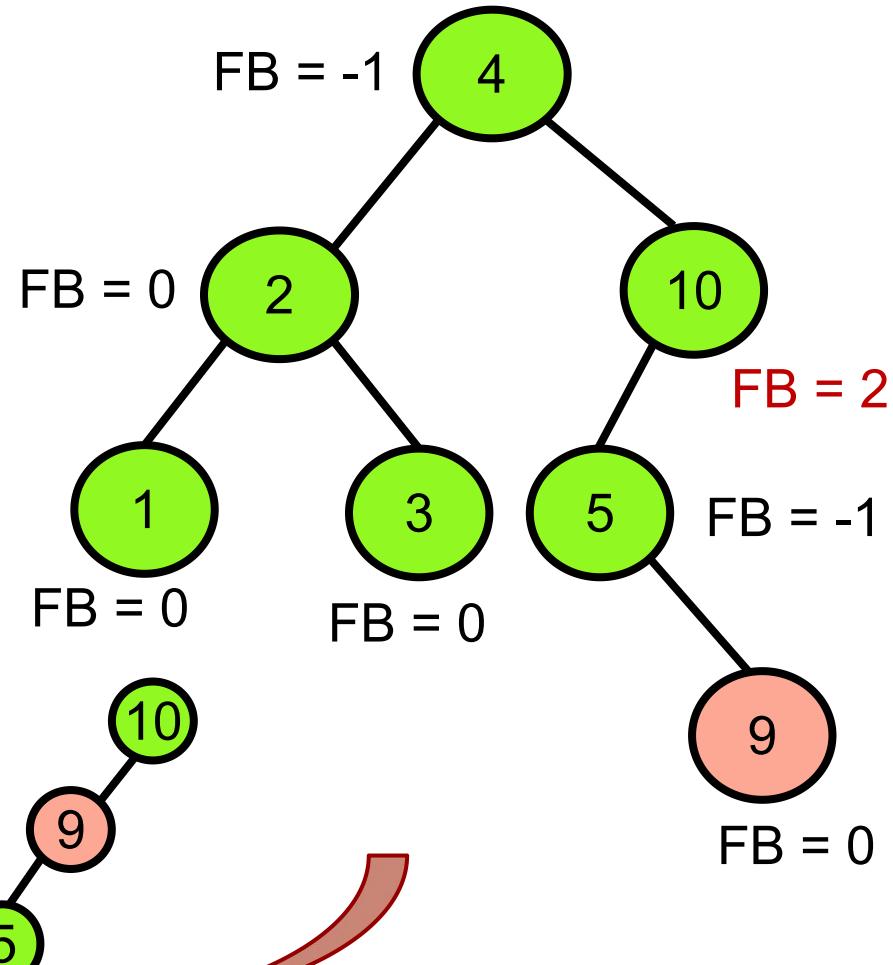
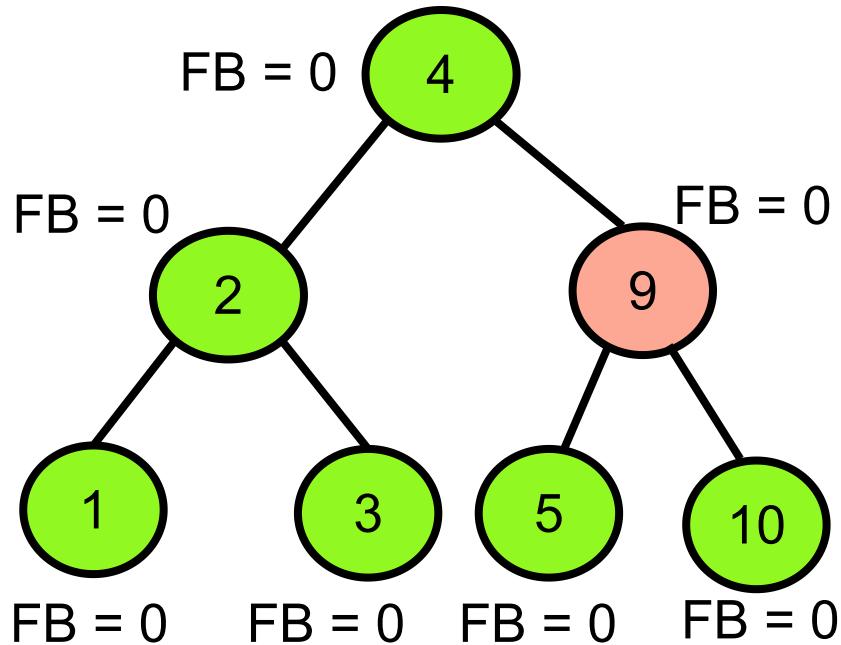
# Resolução

*insere\_AVL2(&A, 9);*



# Resolução

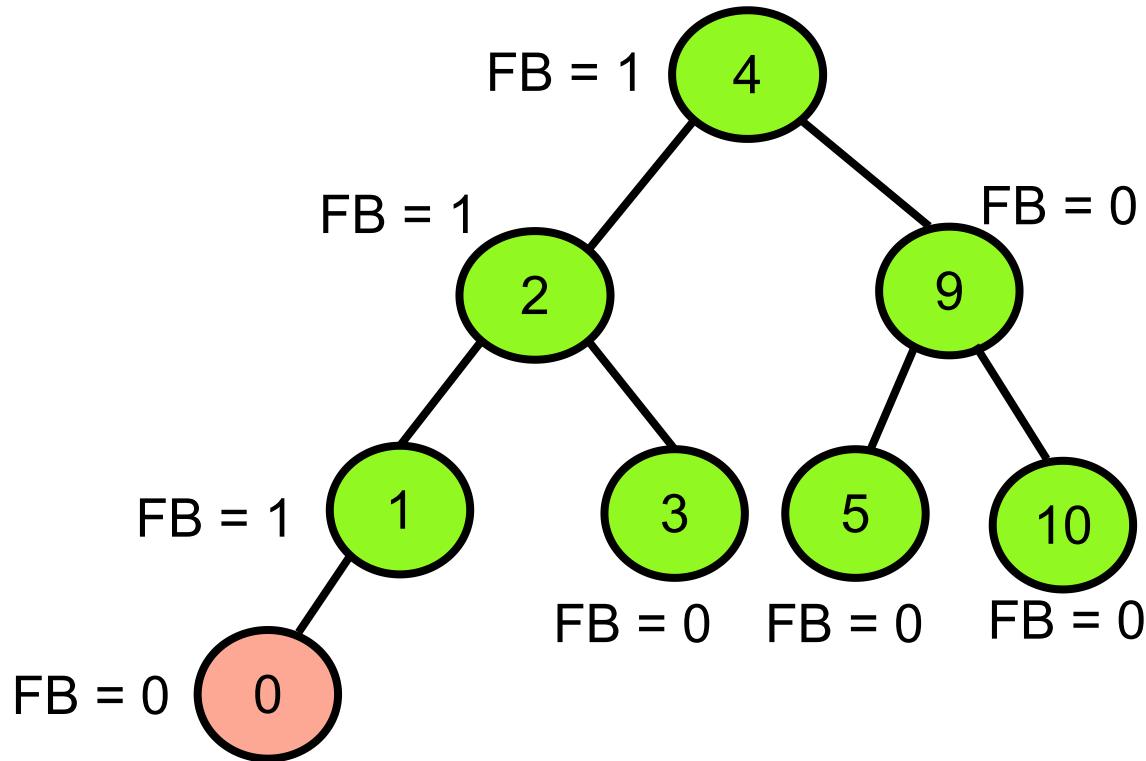
*insere\_AVL2(&A, 9);*



Rotação esquerda-direita

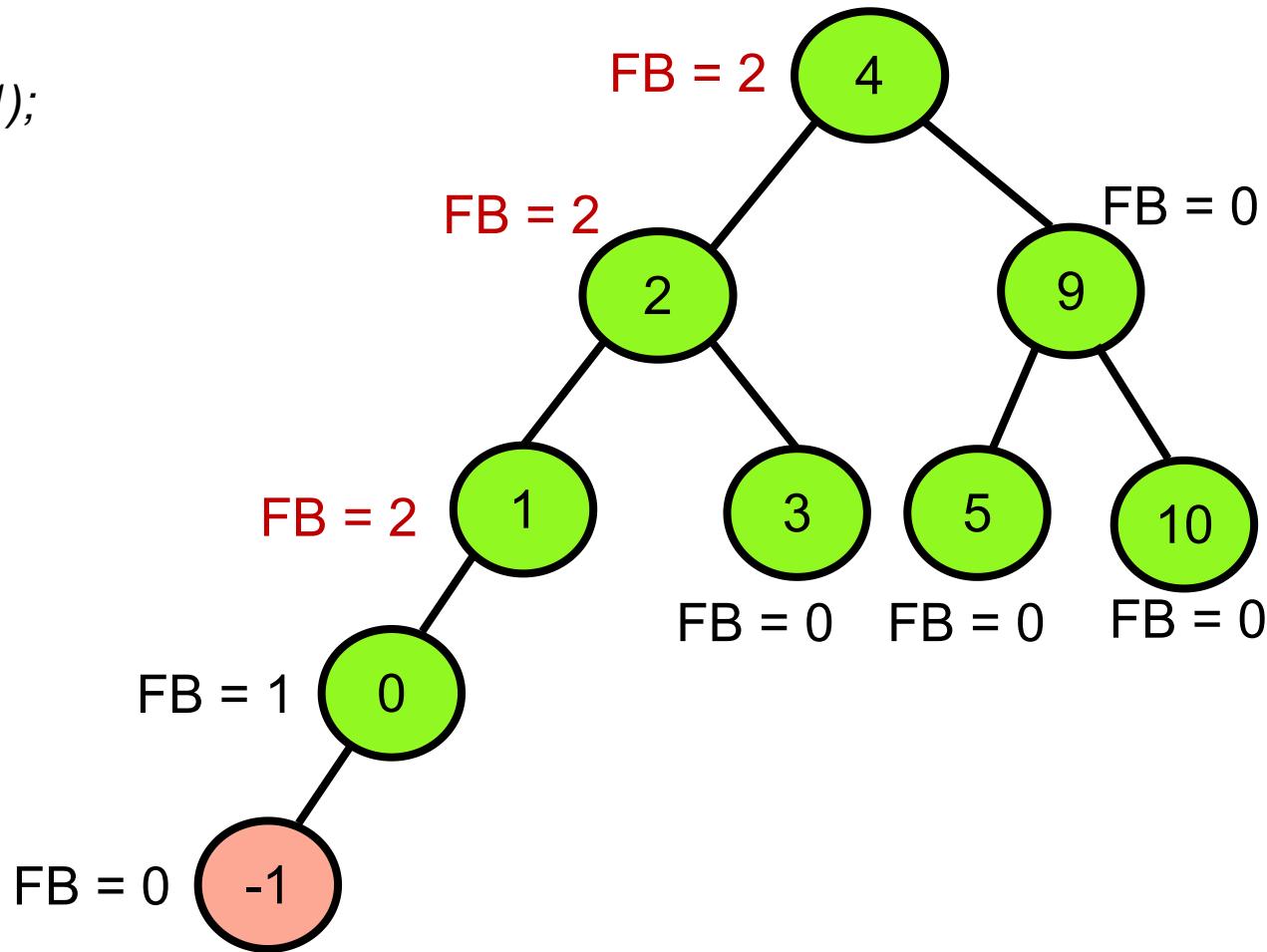
# Resolução

```
insere_AVL2(&A, 0);
```



# Resolução

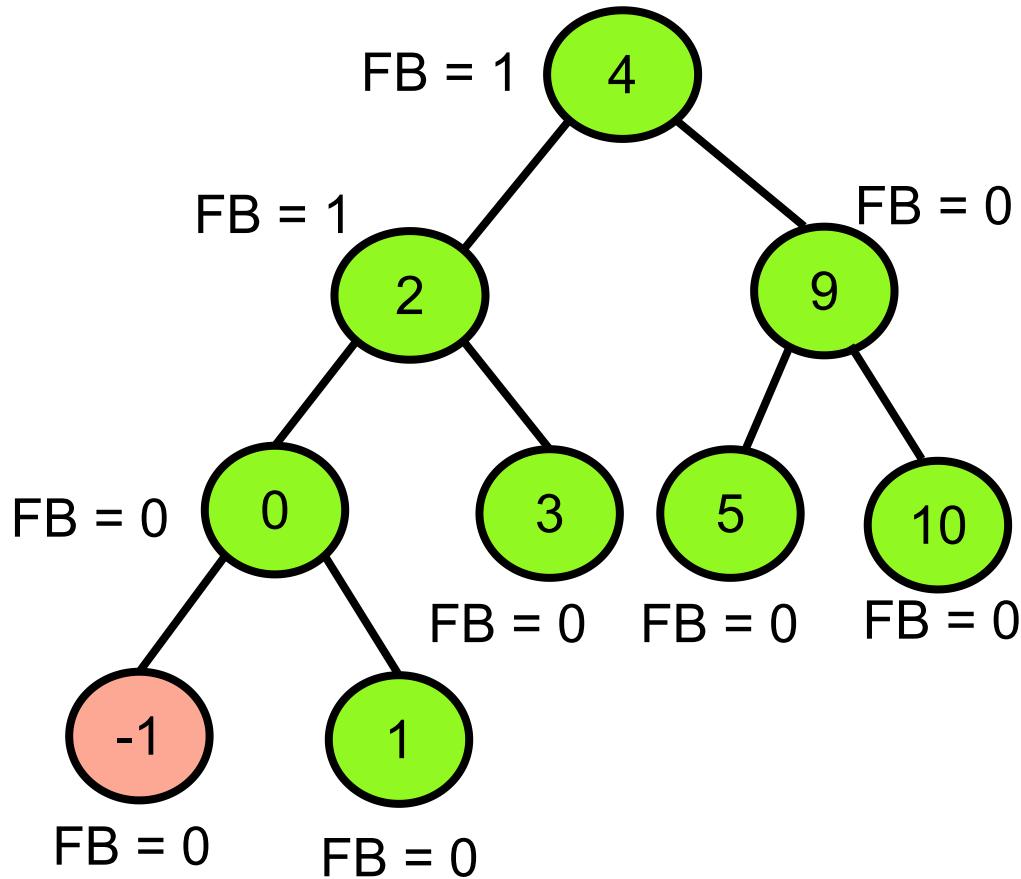
*insere\_AVL2(&A,-1);*



# Resolução

*insere\_AVL2(&A,-1);*

Rotação à  
direita



# Árvores AVL: remoção

---

## **Processo similar à inserção:**

Realiza a remoção ordenada da ABB

Realiza o balanceamento quando necessário

# Árvores AVL: remoção

---

## **Processo similar à inserção:**

Realiza a remoção ordenada da ABB

Realiza o balanceamento quando necessário

## **Tipos de remoção (revisão):**

Árvore vazia (não tem remoção)

Remoção de um nó folha (caso mais simples)

Remoção de um nó com 1 filho (filho assume lugar do pai)

Remoção de um nó com 2 filhos (caso mais complexo)

# Árvores AVL: remoção

---

Valem as **mesmas regras de balanceamento** empregadas na inserção

## Diferença:

Remover de uma subárvore = inserir na outra subárvore

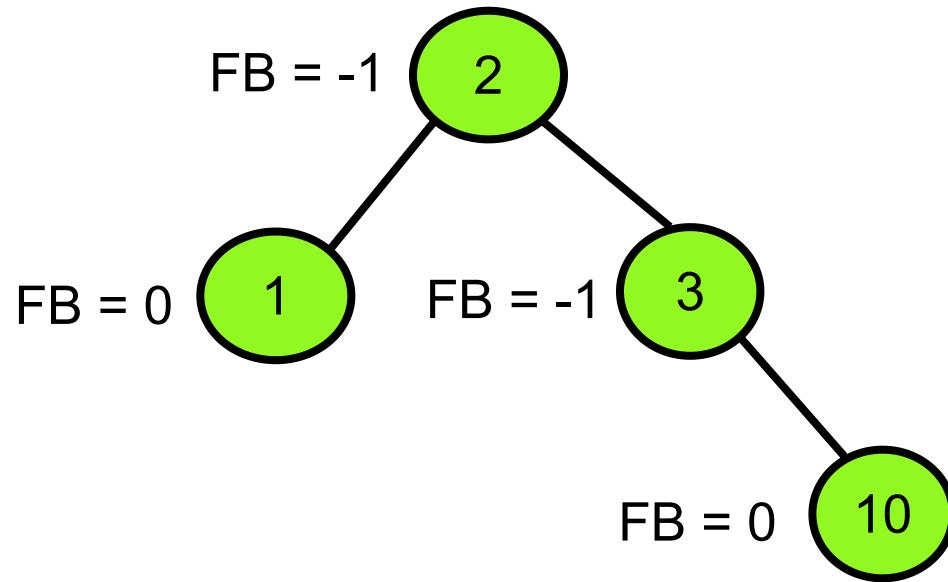
**Ex:** remover da SAE = inserir na SAD

Se houver desbalanceamento, o nó foi removido da menor subárvore (**rotação é na sua direção**)

Rotação ocorrerá da subárvore maior para subárvore menor

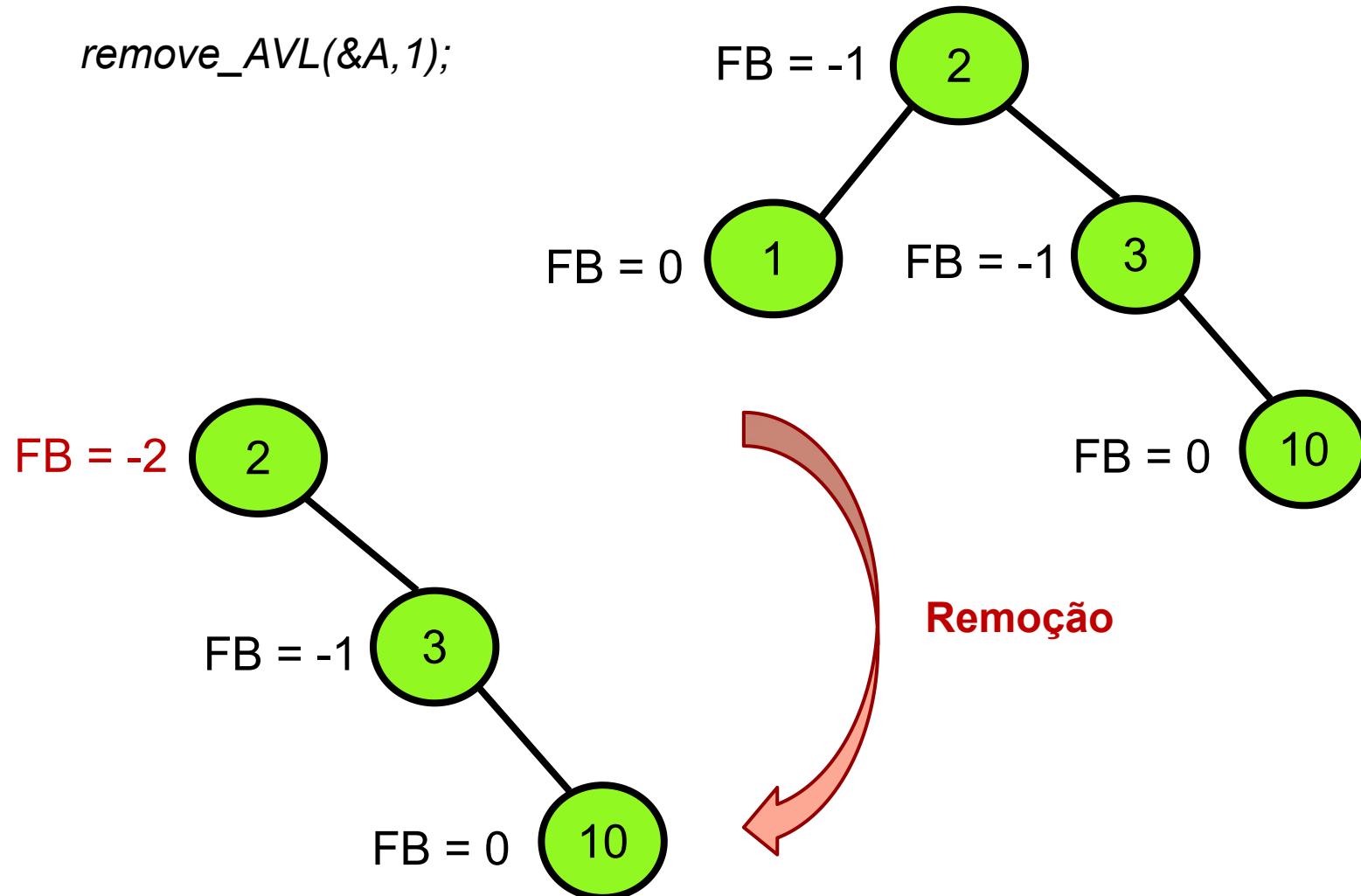
# Exemplo de remoção

*remove\_AVL(&A, 1);*



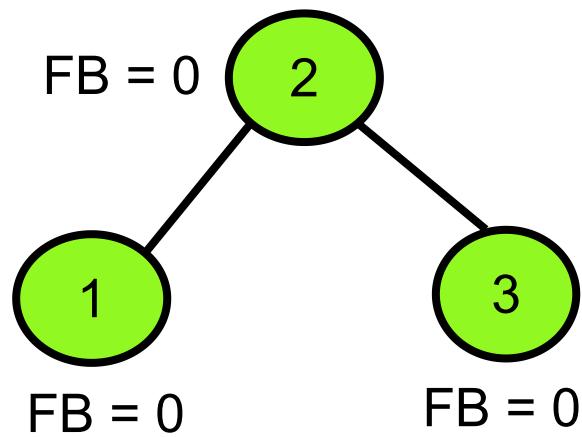
# Exemplo de remoção

`remove_AVL(&A, 1);`



# Exemplo de remoção

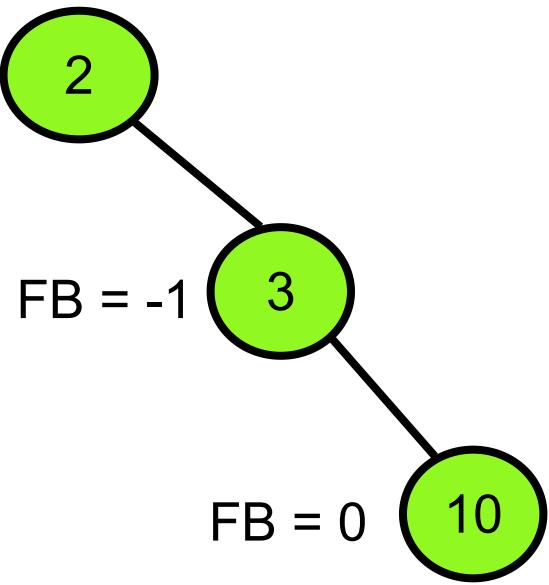
`remove_AVL(&A, 1);`



**FB = -2**

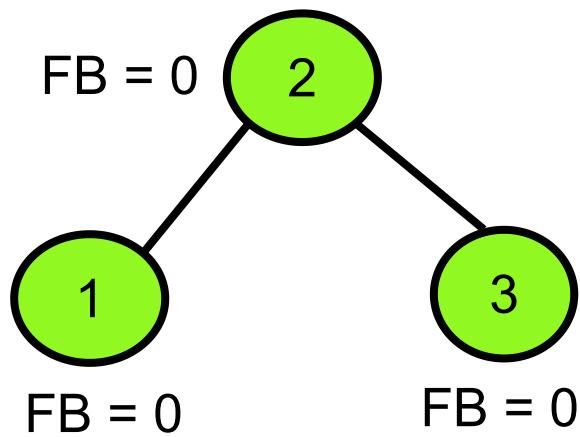
**Rotação à esquerda**

A diagram illustrating a left rotation around node 2. A red curved arrow points from node 2 towards node 3, indicating the direction of the rotation. Above the arrow, the text 'Rotação à esquerda' is written in red. To the left of the arrow, the text 'FB = -2' is displayed in red, likely referring to the balance factor of node 2 after the removal of node 1.



# Exemplo de remoção

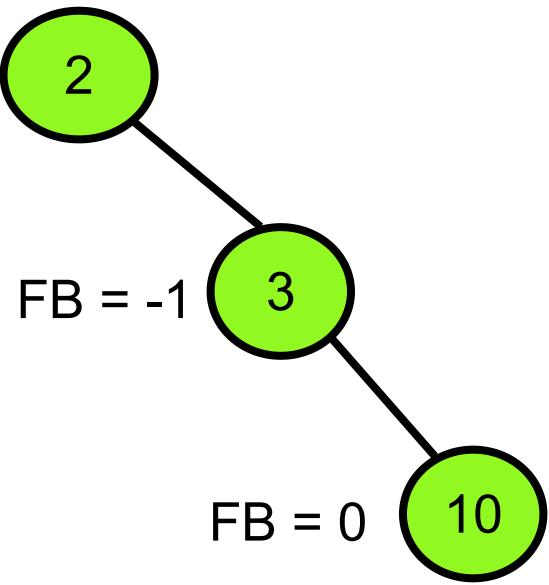
`remove_AVL(&A, 1);`



$FB = -2$

**Rotação à esquerda**

A curved red arrow points from the text "Rotação à esquerda" towards the tree, indicating a left rotation centered at node 2. The text "FB = -2" is displayed above the arrow.



**Implementação da remoção fica  
como exercício**

# Exercícios

1. Dadas a notação textual abaixo, represente a árvore graficamente, calcule o fator de balanceamento de cada nó, identificando se elas são AVL ou não. Para aquelas que não são, indique o que deve ser feito para corrigir o desequilíbrio da árvore.

```
<A <B <D>><>> <E <G>><>> <>> > <C <K>><>> <F <H >> <I>><>> > <>> > >
```

1. Insira os valores {70, 20, 80, 90, 100, 40, 30, 110, 120, 130, 140, 150} em uma árvore AVL na sequência informada. A cada inserção, desenhe a árvore gerada e, quando houver rotações, desenhe cada passo da rotação na árvore.
2. Implemente as operações de rotação e utilize-as para manter o balanceamento após a operação de inserção em uma árvore AVL.

# Exercícios

---

4. Refaça a implementação do exercício anterior, modificando a estrutura do nó para guardar a altura da subárvore na qual o nó é raiz ao invés do seu fator de平衡amento (FB)
5. Implemente a operação de remoção em uma árvore AVL para os dois tipos de estrutura (com FB e com a altura)

# Bibliografia

---

Slides adaptados do material da Profa. Dra. Gina Maira Barbosa de Oliveira, da Profa. Dra. Denise Guliato e do Prof. Dr. Bruno Travençolo.

BACKES, A. Linguagem C Descomplicada: portal de vídeo-aulas para estudo de programação. Disponível em:

<https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/>

EDELWEISS, N; GALANTE, R. Estruturas de dados (Série Livros Didáticos Informática UFRGS, v. 18), Bookman, 2008.

CORMEN, T.H. et al. Algoritmos: Teoria e Prática, Campus, 2002

ZIVIANI, N. Projeto de algoritmos: com implementações em Pascal e C (2<sup>a</sup> ed.), Thomson, 2004