

# Árvore AVL

## Estrutura de Dados Avançada — QXD0115



**UNIVERSIDADE  
FEDERAL DO CEARÁ**  
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz  
[gomes.atilio@ufc.br](mailto:gomes.atilio@ufc.br)

Universidade Federal do Ceará

1º semestre/2025



# Introdução

- **Contexto:** Temos um conjunto de  $n$  chaves  $S = \{s_1, s_2, \dots, s_n\}$  com probabilidade de acesso idênticas entre si.

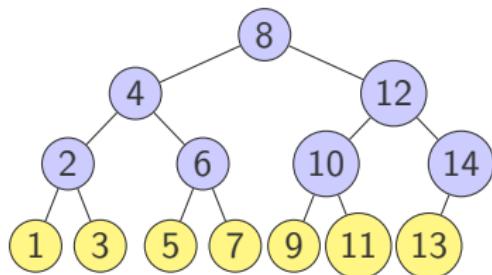
# Introdução

- **Contexto:** Temos um conjunto de  $n$  chaves  $S = \{s_1, s_2, \dots, s_n\}$  com probabilidade de acesso idênticas entre si.
- **Dentre TODAS as árvores binárias de busca com  $n$  nós,** as árvores binárias de busca **completas** são aquelas que minimizam o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrência idênticas.
  - Uma árvore binária completa com  $n > 0$  nós tem altura  $h = 1 + \lfloor \lg n \rfloor$ .

# Introdução

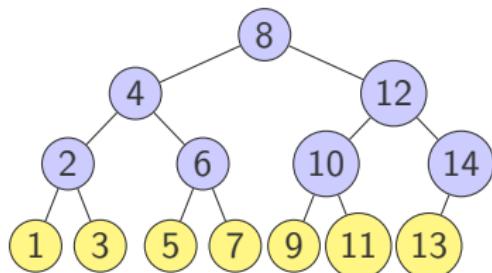
- **Contexto:** Temos um conjunto de  $n$  chaves  $S = \{s_1, s_2, \dots, s_n\}$  com probabilidade de acesso idênticas entre si.
- **Dentre TODAS as árvores binárias de busca com  $n$  nós,** as árvores binárias de busca **completas** são aquelas que minimizam o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrência idênticas.
  - Uma árvore binária completa com  $n > 0$  nós tem altura  $h = 1 + \lfloor \lg n \rfloor$ .
- **Alguns questionamentos:**
  - Seria possível manter a árvore **sempre completa** após consecutivas remoções ou inclusões?
  - Quanto custa isso? Vale o esforço?

# Um exemplo ruim para o restabelecimento de árvores completas

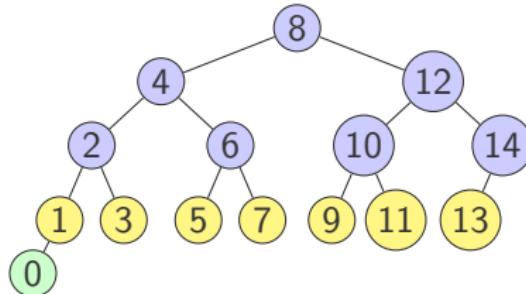


Vamos incluir 0

# Um exemplo ruim para o restabelecimento de árvores completas

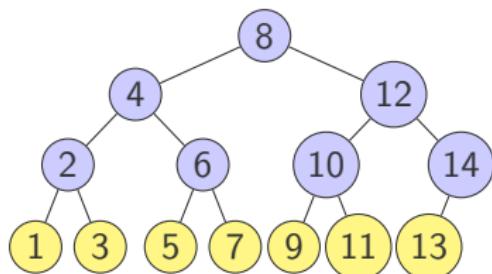


Vamos incluir 0

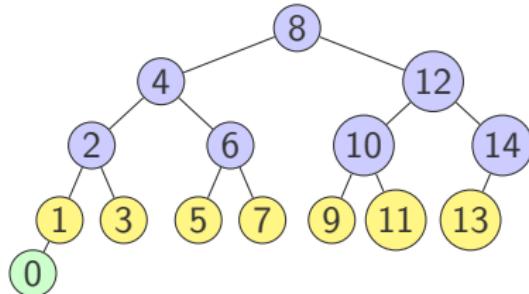


Não é mais completa

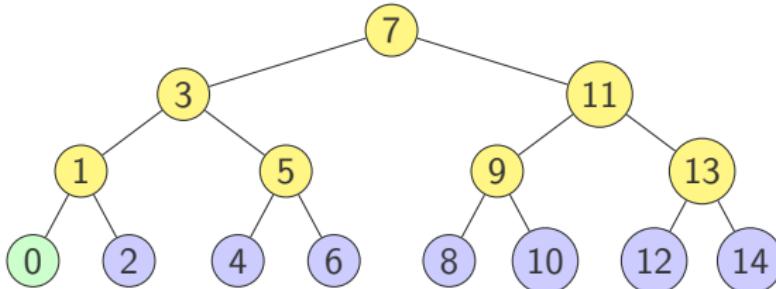
# Um exemplo ruim para o restabelecimento de árvores completas



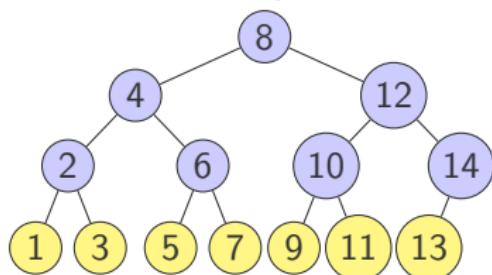
Vamos incluir 0



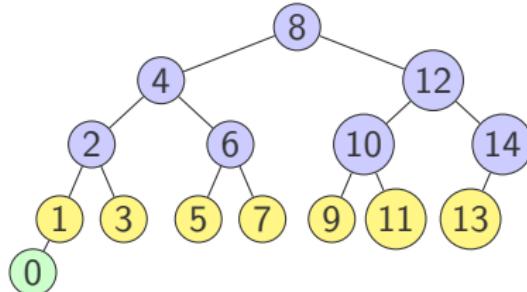
Não é mais completa



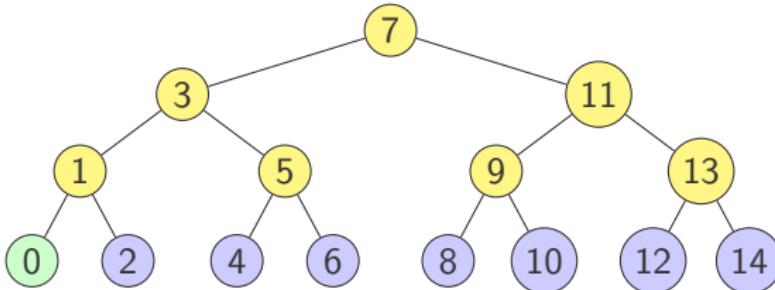
# Um exemplo ruim para o restabelecimento de árvores completas



Vamos incluir 0



Não é mais completa



O algoritmo de restabelecimento requer, pelo menos,  $\Omega(n)$  passos.

**Árvores completas não são recomendadas para aplicações dinâmicas.**

# Alternativa — Árvores Balanceadas

- Uma árvore binária é **balanceada** se sua altura é da ordem de  $O(\lg n)$  e, além disso, esta propriedade se estende a todas as suas subárvores:

# Alternativa — Árvores Balanceadas

- Uma árvore binária é **balanceada** se sua altura é da ordem de  $O(\lg n)$  e, além disso, esta propriedade se estende a todas as suas subárvores:
  - Cada subárvore que contém  $m$  nós deve possuir altura  $O(\lg m)$ .

# Alternativa — Árvores Balanceadas

- Uma árvore binária é **balanceada** se sua altura é da ordem de  $O(\lg n)$  e, além disso, esta propriedade se estende a todas as suas subárvores:
  - Cada subárvore que contém  $m$  nós deve possuir altura  $O(\lg m)$ .
- **Nossa esperança:** Como a forma de uma árvore balanceada é menos rígida que a de uma completa, torna-se “mais fácil” o seu rebalanceamento.



# Árvore AVL



# Árvore AVL

- Primeira árvore binária de busca a garantir tempo de execução  $O(\log n)$  para inserção, busca e remoção no pior caso.
- Criada pelos soviéticos **Adelson-Vělsky** e **Landis**, em 1962.



Adelson-Vělsky



Landis

# Árvore AVL

## Definição

Uma árvore binária  $T$  é do tipo AVL se, **para todo** nó  $v$  de  $T$ , as alturas de suas duas subárvore, esquerda e direita, diferem em módulo de até uma unidade.

## Definição

Uma árvore binária  $T$  é do tipo AVL se, **para todo** nó  $v$  de  $T$ , as alturas de suas duas subárvore, esquerda e direita, diferem em módulo de até uma unidade.

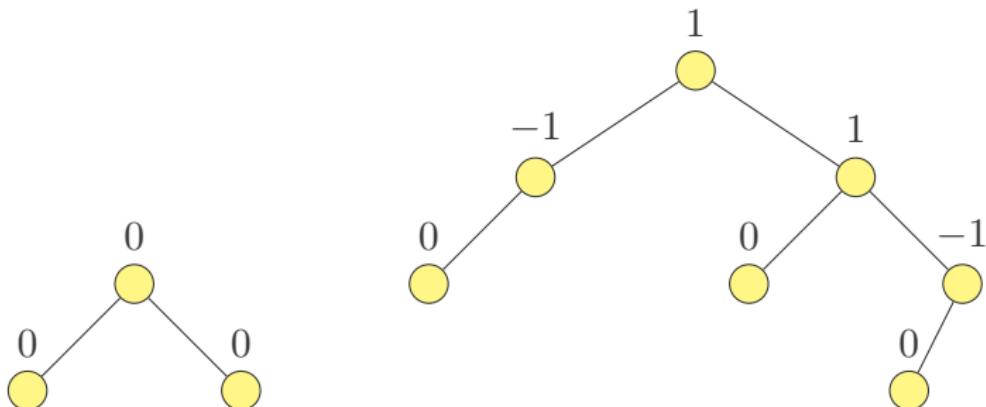
- Um nó  $v$  que satisfaz essa propriedade é dito **regulado**; caso contrário,  $v$  é dito **desregulado**.  
Uma árvore que contém nó desregulado é dita **desregulada**.

## Definição

Uma árvore binária  $T$  é do tipo AVL se, **para todo** nó  $v$  de  $T$ , as alturas de suas duas subárvore, esquerda e direita, diferem em módulo de até uma unidade.

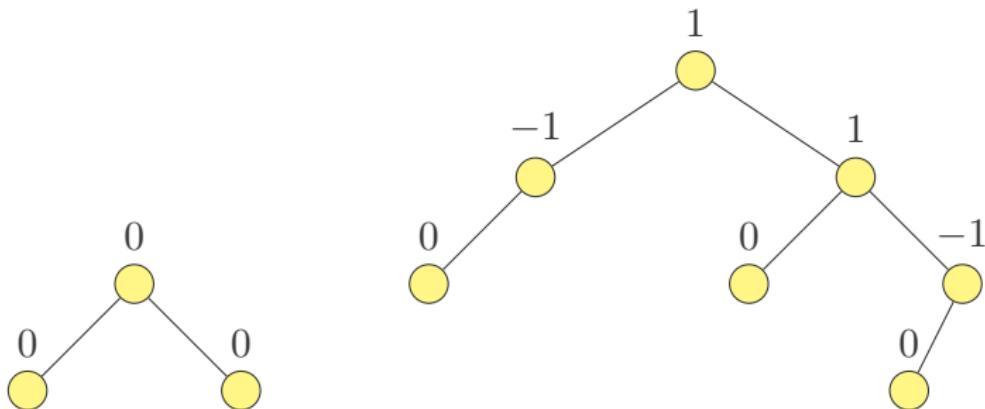
- Um nó  $v$  que satisfaz essa propriedade é dito **regulado**; caso contrário,  $v$  é dito **desregulado**.  
Uma árvore que contém nó desregulado é dita **desregulada**.
- **Fator de balanceamento** ( $fb(v)$ ): a diferença entre as alturas direita e esquerda de  $v$ , ou seja,  $fb(v) = h_D(v) - h_E(v)$ .

# Exemplos de árvores binárias AVL



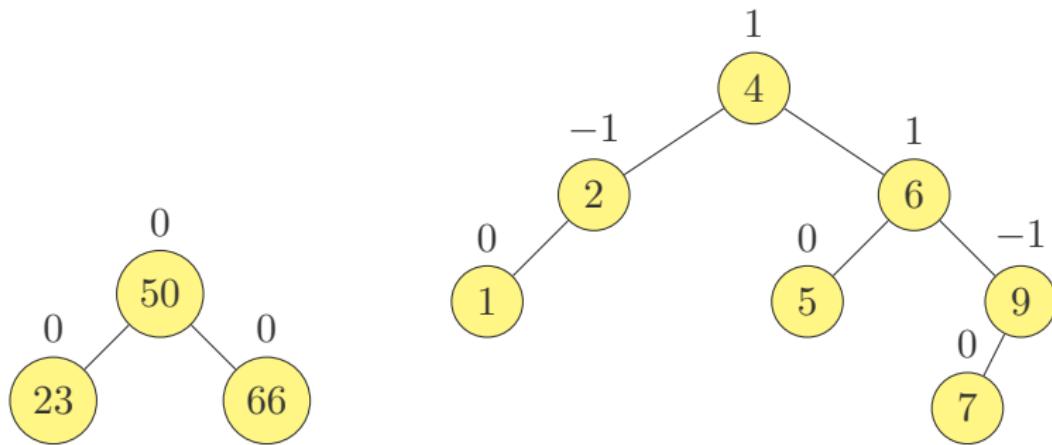
- Todos os nós nestas duas árvores estão **regulados**.

# Exemplos de árvores binárias AVL



- Todos os nós nestas duas árvores estão **regulados**.
- Ou seja, para todo nó  $v$ , temos  $|h_D(v) - h_E(v)| \leq 1$ .

# Exemplos de árvores binárias de busca AVL



- Além de possuirem a propriedade AVL, essas árvores possuem a propriedade de serem binárias de busca.

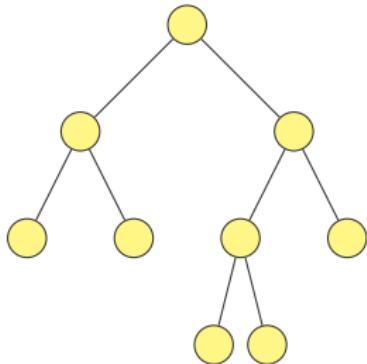
# Árvore completa × Árvore AVL

**Fato:** Toda árvore completa é AVL, mas nem toda árvore AVL é completa.



# Árvore completa × Árvore AVL

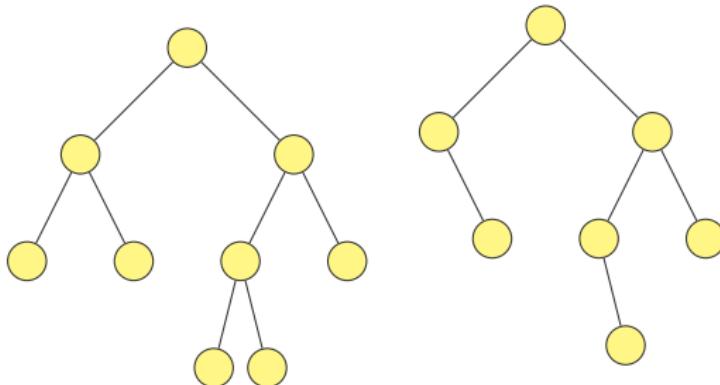
**Fato:** Toda árvore completa é AVL, mas nem toda árvore AVL é completa. □



Árvore completa

# Árvore completa × Árvore AVL

**Fato:** Toda árvore completa é AVL, mas nem toda árvore AVL é completa. □

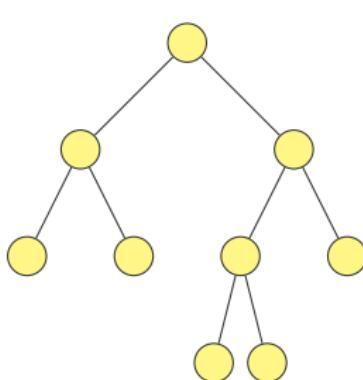


Árvore completa

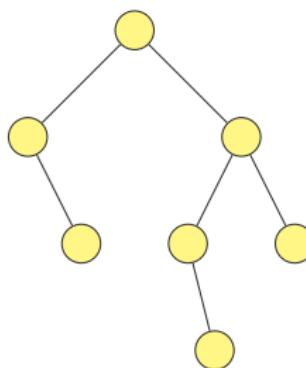
É AVL mas não é completa. **Por quê?**

# Árvore completa × Árvore AVL

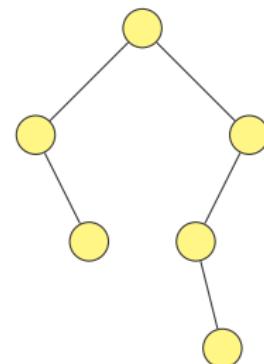
**Fato:** Toda árvore completa é AVL, mas nem toda árvore AVL é completa. □



Árvore completa



É AVL mas não é completa. **Por quê?**



Não é completa e nem AVL. **Por quê?**



## Prova do balanceamento



# Balanceamento de árvores AVL

- **Ideia:** Dada uma árvore AVL  $T$ , vamos fixar a altura  $h$  e determinar o valor mínimo do número  $n$  de nós.

# Balanceamento de árvores AVL

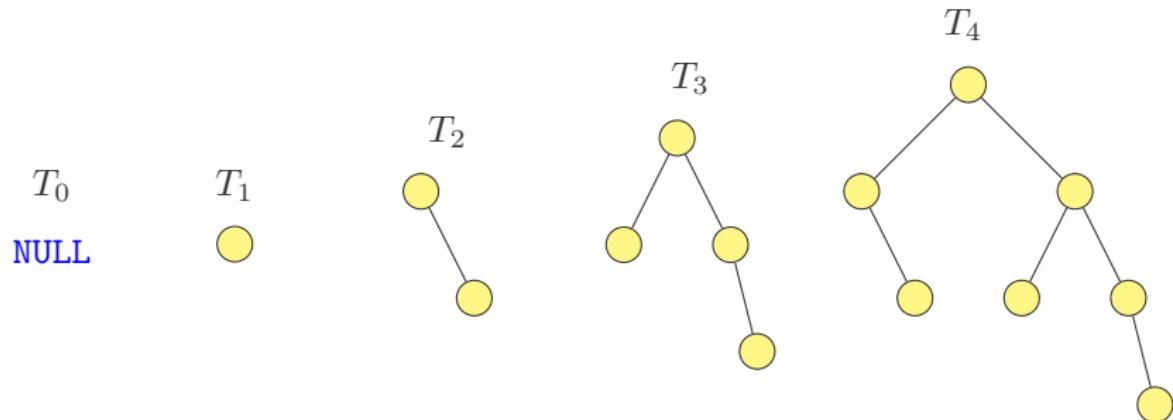
- **Ideia:** Dada uma árvore AVL  $T$ , vamos fixar a altura  $h$  e determinar o valor mínimo do número  $n$  de nós.

## Problema

Dada uma árvore AVL de altura  $h$ , qual seria o valor mínimo possível para  $n$ ?

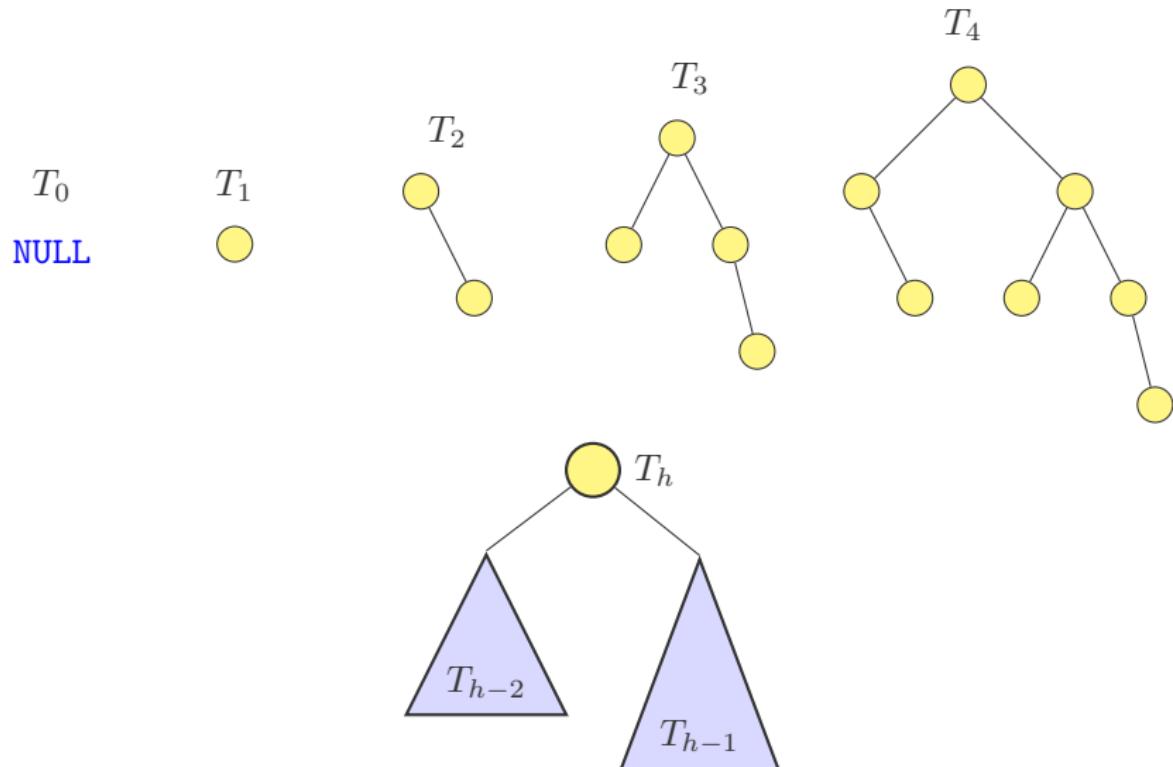
# Árvores de Fibonacci

- Árvores AVL com os piores fatores de平衡amento.



# Árvores de Fibonacci

- Árvores AVL com os piores fatores de平衡amento.



# Número de nós — Árvores de Fibonacci

- Denotamos por  $N(T)$  o número de nós de uma árvore  $T$

## Número de nós — Árvores de Fibonacci

- Denotamos por  $N(T)$  o número de nós de uma árvore  $T$
- Número de nós de  $T_h$ :

$$N(T_h) = \begin{cases} 0 & \text{se } h = 0; \\ 1 & \text{se } h = 1; \\ 1 + N(T_{h-1}) + N(T_{h-2}) & \text{se } h > 1. \end{cases}$$

# Número de nós — Árvores de Fibonacci

- Denotamos por  $N(T)$  o número de nós de uma árvore  $T$
- Número de nós de  $T_h$ :

$$N(T_h) = \begin{cases} 0 & \text{se } h = 0; \\ 1 & \text{se } h = 1; \\ 1 + N(T_{h-1}) + N(T_{h-2}) & \text{se } h > 1. \end{cases}$$

- A fórmula acima, lembra a fórmula do  $h$ -ésimo termo da **sequência de Fibonacci**:

$$F(h) = \begin{cases} 0 & \text{se } h = 0; \\ 1 & \text{se } h = 1; \\ F(h - 1) + F(h - 2) & \text{se } h > 1. \end{cases}$$

# Número de nós — Árvores de Fibonacci

- Denotamos por  $N(T)$  o número de nós de uma árvore  $T$
- Número de nós de  $T_h$ :

$$N(T_h) = \begin{cases} 0 & \text{se } h = 0; \\ 1 & \text{se } h = 1; \\ 1 + N(T_{h-1}) + N(T_{h-2}) & \text{se } h > 1. \end{cases}$$

- A fórmula acima, lembra a fórmula do  $h$ -ésimo termo da **sequência de Fibonacci**:

$$F(h) = \begin{cases} 0 & \text{se } h = 0; \\ 1 & \text{se } h = 1; \\ F(h - 1) + F(h - 2) & \text{se } h > 1. \end{cases}$$

- Logo,  $N(T_h) \geq F(h)$ , para todo  $h \geq 0$ .

# Número de nós — Árvores de Fibonacci

**Fato:** Dada uma árvore AVL  $T$  de altura  $h$ , temos que:  
 $N(T) \geq N(T_h) \geq F(h)$ .

# Número de nós — Árvores de Fibonacci

**Fato:** Dada uma árvore AVL  $T$  de altura  $h$ , temos que:  
 $N(T) \geq N(T_h) \geq F(h)$ .

Fórmula do  $h$ -ésimo termo da sequência de Fibonacci:

$$F(h) = \begin{cases} 0 & \text{se } h = 0; \\ 1 & \text{se } h = 1; \\ F(h - 1) + F(h - 2) & \text{se } h > 1. \end{cases}$$

**Fato:** Para  $h > 1$ , o  $h$ -ésimo termo da sequência de Fibonacci é dado por:

$$F(h) = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^h - \left( \frac{1 - \sqrt{5}}{2} \right)^h \right]$$

# Prova $h = O(\lg n)$

Se  $T$  é uma árvore AVL com altura  $h$  e com  $n$  nós, então  $h = O(\lg n)$ .

## Prova $h = O(\lg n)$

Se  $T$  é uma árvore AVL com altura  $h$  e com  $n$  nós, então  $h = O(\lg n)$ .

**Prova:** Do slide anterior, temos que  $n \geq F(h) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h$ .

## Prova $h = O(\lg n)$

Se  $T$  é uma árvore AVL com altura  $h$  e com  $n$  nós, então  $h = O(\lg n)$ .

**Prova:** Do slide anterior, temos que  $n \geq F(h) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h$ .

Como  $h > 0$ , temos que  $\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h < 1$ .

## Prova $h = O(\lg n)$

Se  $T$  é uma árvore AVL com altura  $h$  e com  $n$  nós, então  $h = O(\lg n)$ .

**Prova:** Do slide anterior, temos que  $n \geq F(h) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h$ .

Como  $h > 0$ , temos que  $\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h < 1$ .

Portanto,  $n \geq F(h) > \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - 1$ .

## Prova $h = O(\lg n)$

Se  $T$  é uma árvore AVL com altura  $h$  e com  $n$  nós, então  $h = O(\lg n)$ .

**Prova:** Do slide anterior, temos que  $n \geq F(h) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h$ .

Como  $h > 0$ , temos que  $\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h < 1$ .

Portanto,  $n \geq F(h) > \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - 1$ .

Fazendo  $a = \frac{1+\sqrt{5}}{2}$ , tem-se  $n > \frac{1}{\sqrt{5}}a^h - 1$ . O que implica,  $n + 1 > \frac{a^h}{\sqrt{5}}$ .

# Prova $h = O(\lg n)$

Se  $T$  é uma árvore AVL com altura  $h$  e com  $n$  nós, então  $h = O(\lg n)$ .

**Prova:** Do slide anterior, temos que  $n \geq F(h) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h$ .

Como  $h > 0$ , temos que  $\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h < 1$ .

Portanto,  $n \geq F(h) > \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - 1$ .

Fazendo  $a = \frac{1+\sqrt{5}}{2}$ , tem-se  $n > \frac{1}{\sqrt{5}}a^h - 1$ . O que implica,  $n + 1 > \frac{a^h}{\sqrt{5}}$ .

Aplicando logaritmo na base  $a$  em ambos os lados, temos:

$$\log_a(n + 1) > \log_a a^h - \log_a \sqrt{5}$$

$$\log_a(n + 1) > h - \log_a \sqrt{5}$$

$$h < \log_a(n + 1) + \log_a \sqrt{5}$$

$$h < \frac{1}{\log_2 a} \log_2(n + 1) + \log_a \sqrt{5} \quad (\text{mudança de base})$$

$$h < 1.44 \cdot \log_2(n + 1) + 1.67 = O(\lg n).$$



## Inserção em árvores AVL



# Inserção em Árvores AVL

- **Ideia:** Após cada inserção, verificar se algum nó  $p$  se encontra desregulado.
  - Em caso positivo, aplicar transformações apropriadas para regulá-lo.

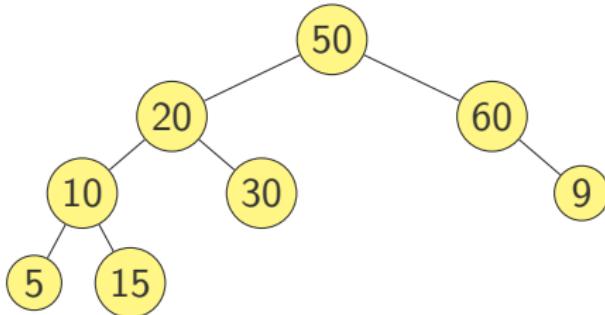
# Inserção em Árvores AVL

- **Ideia:** Após cada inserção, verificar se algum nó  $p$  se encontra desregulado.
  - Em caso positivo, aplicar transformações apropriadas para regulá-lo.
- **Pergunta:** Após a inserção de um nó, quais nós podem ter se tornado desregulados?

# Inserção em Árvores AVL

- **Ideia:** Após cada inserção, verificar se algum nó  $p$  se encontra desregulado.
  - Em caso positivo, aplicar transformações apropriadas para regulá-lo.
- **Pergunta:** Após a inserção de um nó, quais nós podem ter se tornado desregulados?

**Exemplo:** Inserir 3 na árvore abaixo.



# Inserção em Árvores AVL

- Em alguns casos, a inserção de um novo nó  $x$  na árvore AVL pode vir a modificar o fator de balanceamento de algum nó no caminho que vai de  $x$  até a raiz.

# Inserção em Árvores AVL

- Em alguns casos, a inserção de um novo nó  $x$  na árvore AVL pode vir a modificar o fator de balanceamento de algum nó no caminho que vai de  $x$  até a raiz.
- No caso em que algum nó  $v$  neste caminho ficar desregulado, uma **operação de regulagem** do nó  $v$  deve ser realizada **a fim de regular o nó**.

# Inserção em Árvores AVL

- Em alguns casos, a inserção de um novo nó  $x$  na árvore AVL pode vir a modificar o fator de balanceamento de algum nó no caminho que vai de  $x$  até a raiz.
- No caso em que algum nó  $v$  neste caminho ficar desregulado, uma **operação de regulagem** do nó  $v$  deve ser realizada **a fim de regular o nó**.
  - A essa operação de regulagem do nó  $v$  chamaremos de **rotação do nó  $v$** .

# Inserção em Árvores AVL

- Em alguns casos, a inserção de um novo nó  $x$  na árvore AVL pode vir a modificar o fator de balanceamento de algum nó no caminho que vai de  $x$  até a raiz.
- No caso em que algum nó  $v$  neste caminho ficar desregulado, uma **operação de regulagem** do nó  $v$  deve ser realizada **a fim de regular o nó**.
  - A essa operação de regulagem do nó  $v$  chamaremos de **rotação do nó  $v$** .
- Usaremos basicamente quatro tipos de rotações:

# Inserção em Árvores AVL

- Em alguns casos, a inserção de um novo nó  $x$  na árvore AVL pode vir a modificar o fator de balanceamento de algum nó no caminho que vai de  $x$  até a raiz.
- No caso em que algum nó  $v$  neste caminho ficar desregulado, uma **operação de regulagem** do nó  $v$  deve ser realizada **a fim de regular o nó**.
  - A essa operação de regulagem do nó  $v$  chamaremos de **rotação do nó  $v$** .
- Usaremos basicamente quatro tipos de rotações:
  - Rotação esquerda

# Inserção em Árvores AVL

- Em alguns casos, a inserção de um novo nó  $x$  na árvore AVL pode vir a modificar o fator de balanceamento de algum nó no caminho que vai de  $x$  até a raiz.
- No caso em que algum nó  $v$  neste caminho ficar desregulado, uma **operação de regulagem** do nó  $v$  deve ser realizada **a fim de regular o nó**.
  - A essa operação de regulagem do nó  $v$  chamaremos de **rotação do nó  $v$** .
- Usaremos basicamente quatro tipos de rotações:
  - Rotação esquerda
  - Rotação direita

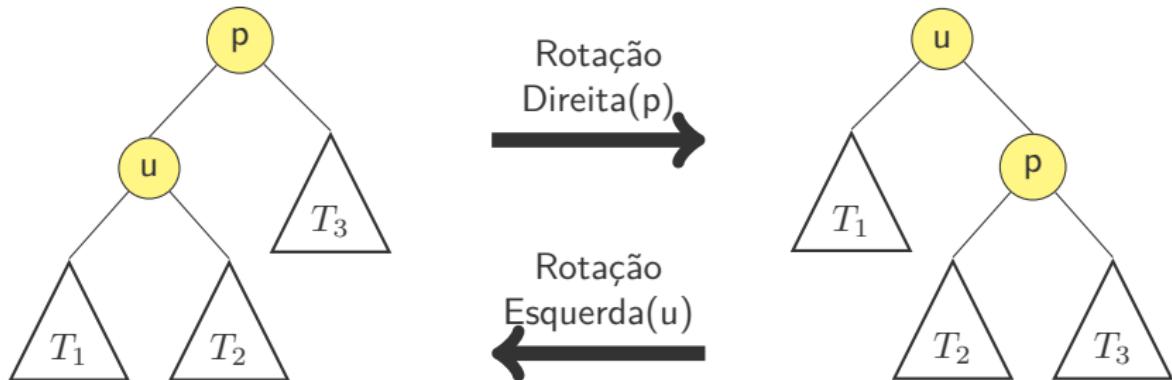
# Inserção em Árvores AVL

- Em alguns casos, a inserção de um novo nó  $x$  na árvore AVL pode vir a modificar o fator de balanceamento de algum nó no caminho que vai de  $x$  até a raiz.
- No caso em que algum nó  $v$  neste caminho ficar desregulado, uma **operação de regulagem** do nó  $v$  deve ser realizada **a fim de regular o nó**.
  - A essa operação de regulagem do nó  $v$  chamaremos de **rotação do nó  $v$** .
- Usaremos basicamente quatro tipos de rotações:
  - Rotação esquerda
  - Rotação direita
  - Rotação dupla à esquerda

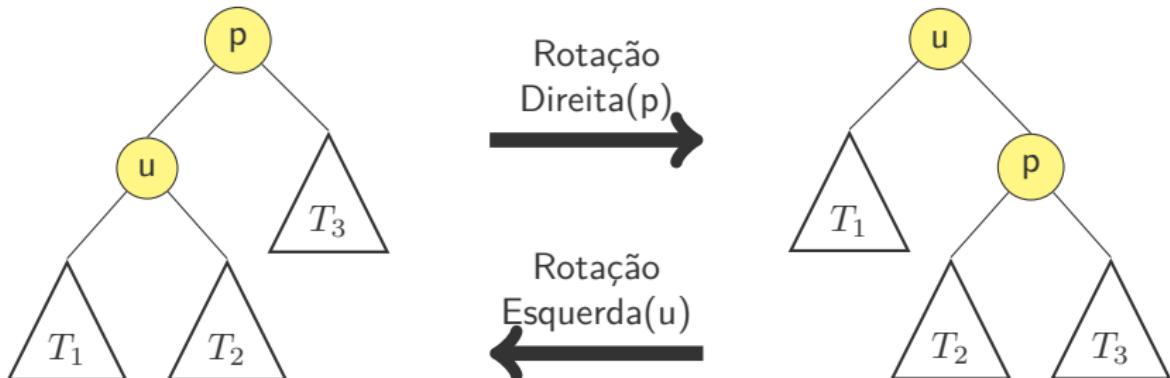
# Inserção em Árvores AVL

- Em alguns casos, a inserção de um novo nó  $x$  na árvore AVL pode vir a modificar o fator de balanceamento de algum nó no caminho que vai de  $x$  até a raiz.
- No caso em que algum nó  $v$  neste caminho ficar desregulado, uma **operação de regulagem** do nó  $v$  deve ser realizada **a fim de regular o nó**.
  - A essa operação de regulagem do nó  $v$  chamaremos de **rotação do nó  $v$** .
- Usaremos basicamente quatro tipos de rotações:
  - Rotação esquerda
  - Rotação direita
  - Rotação dupla à esquerda
  - Rotação dupla à direita

# Rotação Esquerda e Rotação Direita



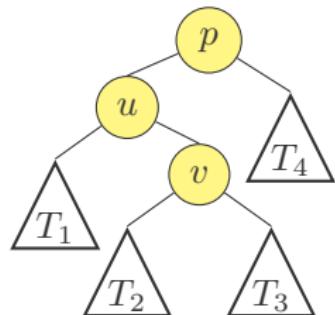
# Rotação Esquerda e Rotação Direita



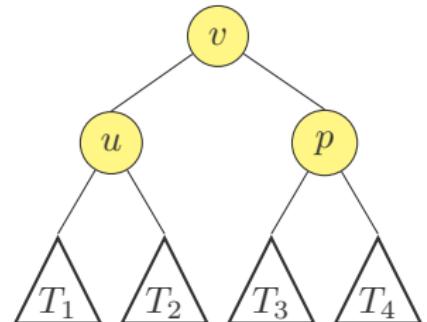
## Propriedade 1

As rotações preservam a natureza da árvore como sendo binária de busca. □

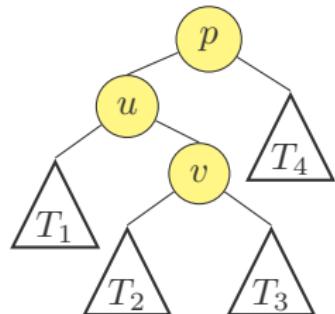
# Rotação Dupla à Direita



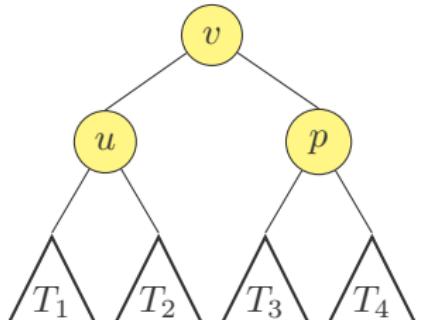
Rotação Dupla  
à Direita ( $p$ )



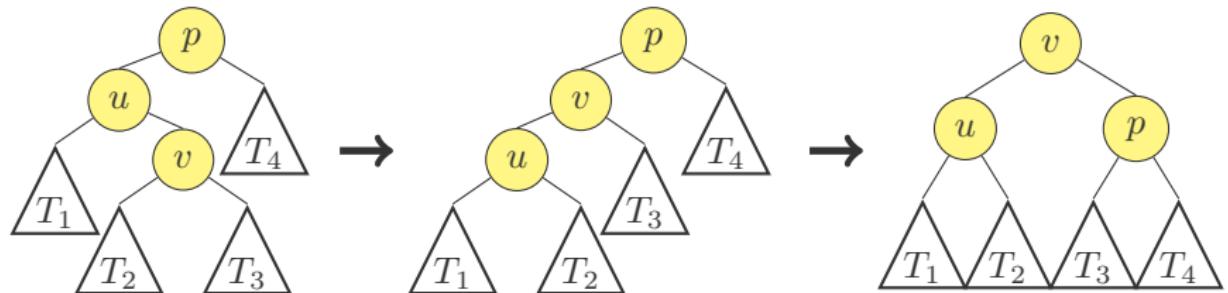
# Rotação Dupla à Direita



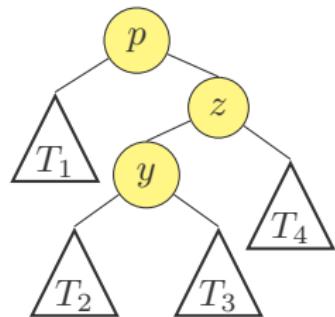
Rotação Dupla  
à Direita ( $p$ )



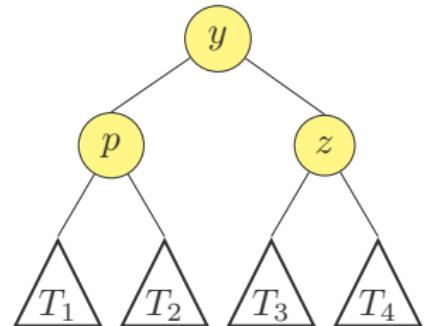
- Rotação Dupla à Direita = Rot. Esquerda( $u$ ) + Rot. Direita( $p$ )



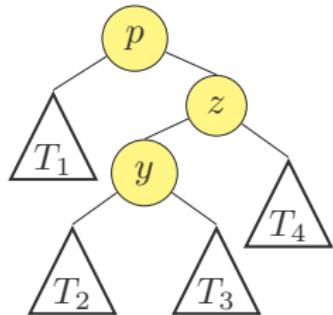
# Rotação Dupla à Esquerda



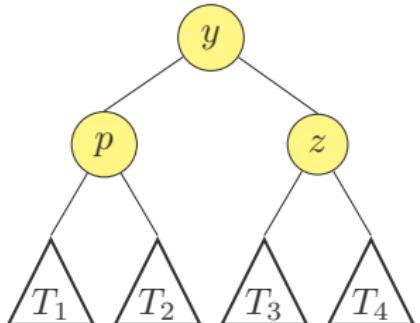
Rotação Dupla  
à Esquerda (p)



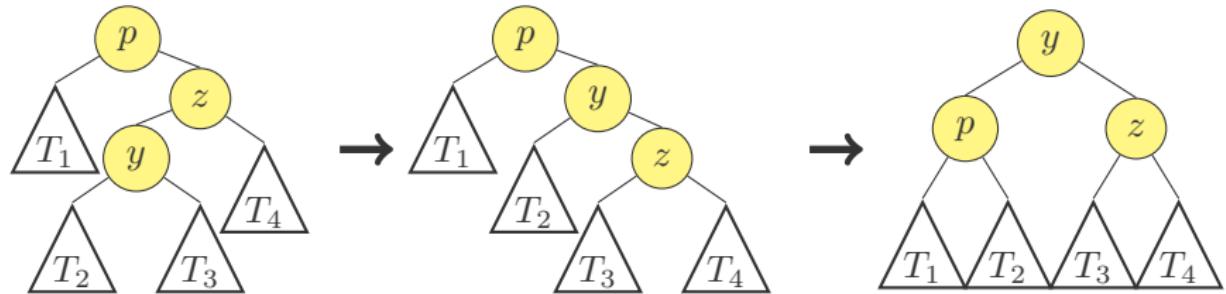
# Rotação Dupla à Esquerda



Rotação Dupla  
à Esquerda ( $p$ )

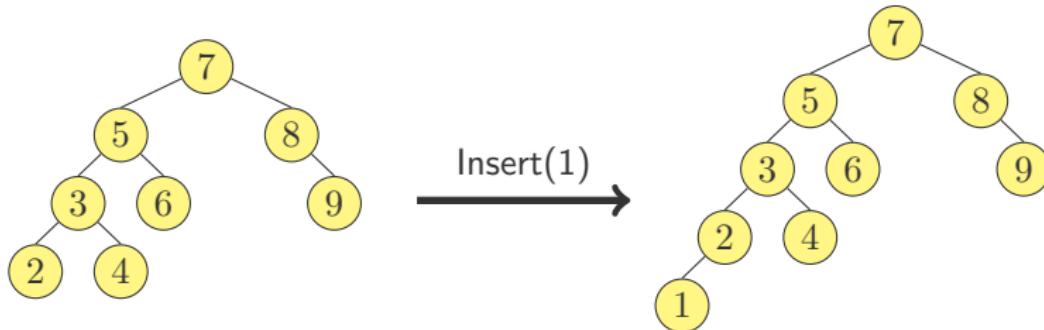



- Rotação Dupla à Esquerda = Rot. Direita( $z$ ) + Rot. Esquerda( $p$ )



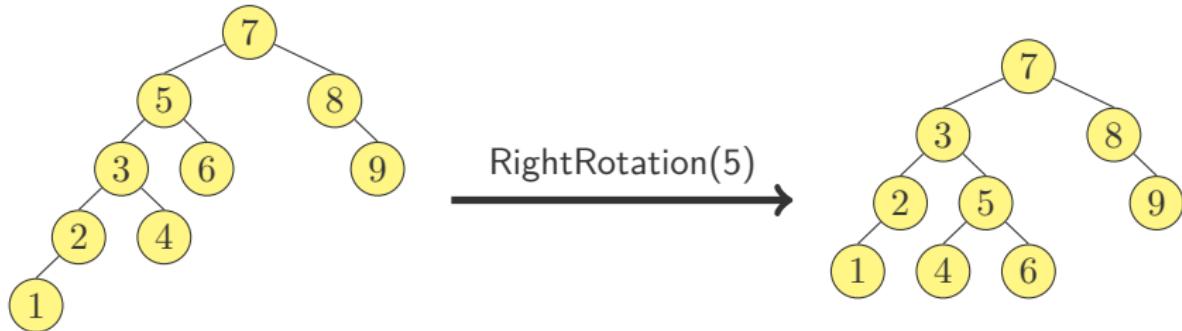
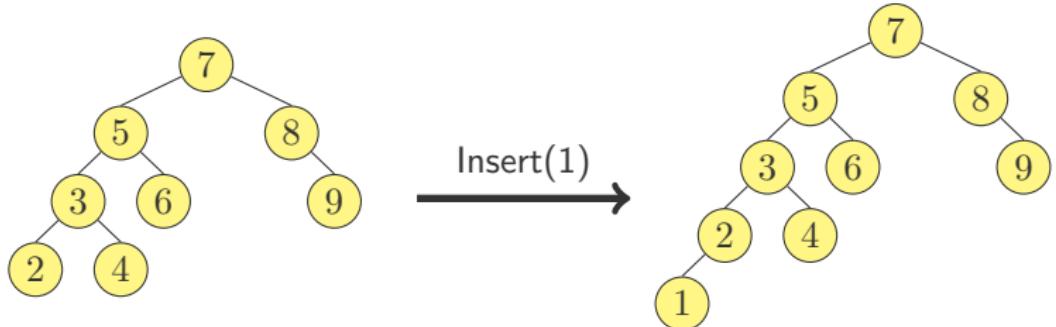
## Exemplo

Inclusão da chave 1 na árvore e posterior efeito de uma rotação direita do nó 5, que tornou-se desregulado após a inclusão de 1.



## Exemplo

Inclusão da chave 1 na árvore e posterior efeito de uma rotação direita do nó 5, que tornou-se desregulado após a inclusão de 1.





## Análise da operação de inserção



# Análise da Inserção

- Após a inserção de um nó  $x$  na árvore, as chamadas recursivas vão se “desenrolando” e todos os vértices no caminho de  $x$  até a raiz da árvore devem ter seus **fatores de balanceamento** devidamente checados.

# Análise da Inserção

- Após a inserção de um nó  $x$  na árvore, as chamadas recursivas vão se “desenrolando” e todos os vértices no caminho de  $x$  até a raiz da árvore devem ter seus **fatores de balanceamento** devidamente checados.
- Vamos provar que, uma vez que um nó  $p$  torna-se desregulado, a regulagem de  $p$  é restabelecida pela aplicação de uma das rotações AVL estudadas.

# Análise da Inserção

- Suponha que o nó  $x$  acabou de ser inserido em  $T$ .

## Análise da Inserção

- Suponha que o nó  $x$  acabou de ser inserido em  $T$ .
- Se após a inclusão de  $x$  todos os ancestrais de  $x$  mantiveram-se regulados, então a árvore manteve-se AVL e não há nada o que efetuar.

## Análise da Inserção

- Suponha que o nó  $x$  acabou de ser inserido em  $T$ .
- Se após a inclusão de  $x$  todos os ancestrais de  $x$  mantiveram-se regulados, então a árvore manteve-se AVL e não há nada o que efetuar.
- Caso contrário, seja  $p$  o ancestral de  $x$  mais próximo que se tornou desregulado.

# Análise da Inserção

- Suponha que o nó  $x$  acabou de ser inserido em  $T$ .
- Se após a inclusão de  $x$  todos os ancestrais de  $x$  mantiveram-se regulados, então a árvore manteve-se AVL e não há nada o que efetuar.
- Caso contrário, seja  $p$  o ancestral de  $x$  mais próximo que se tornou desregulado.
  - Temos que  $|h_D(p) - h_E(p)| = 2$  pois  $T$  era uma árvore AVL antes da inclusão de  $x$  e, além disso, a inclusão de um nó não pode aumentar em mais de uma unidade a altura de qualquer subárvore.

# Análise da Inserção

- Suponha que o nó  $x$  acabou de ser inserido em  $T$ .
- Se após a inclusão de  $x$  todos os ancestrais de  $x$  mantiveram-se regulados, então a árvore manteve-se AVL e não há nada o que efetuar.
- Caso contrário, seja  $p$  o ancestral de  $x$  mais próximo que se tornou desregulado.
  - Temos que  $|h_D(p) - h_E(p)| = 2$  pois  $T$  era uma árvore AVL antes da inclusão de  $x$  e, além disso, a inclusão de um nó não pode aumentar em mais de uma unidade a altura de qualquer subárvore.
- Há exatamente dois casos a considerar:

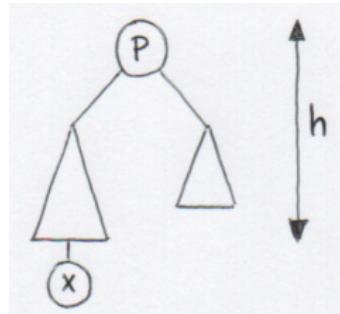
# Análise da Inserção

- Suponha que o nó  $x$  acabou de ser inserido em  $T$ .
- Se após a inclusão de  $x$  todos os ancestrais de  $x$  mantiveram-se regulados, então a árvore manteve-se AVL e não há nada o que efetuar.
- Caso contrário, seja  $p$  o ancestral de  $x$  mais próximo que se tornou desregulado.
  - Temos que  $|h_D(p) - h_E(p)| = 2$  pois  $T$  era uma árvore AVL antes da inclusão de  $x$  e, além disso, a inclusão de um nó não pode aumentar em mais de uma unidade a altura de qualquer subárvore.
- Há exatamente dois casos a considerar:
  - **Caso (1):**  $h_E(p) > h_D(p)$

# Análise da Inserção

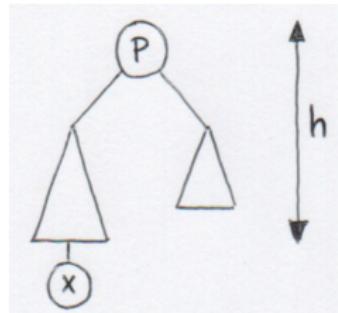
- Suponha que o nó  $x$  acabou de ser inserido em  $T$ .
- Se após a inclusão de  $x$  todos os ancestrais de  $x$  mantiveram-se regulados, então a árvore manteve-se AVL e não há nada o que efetuar.
- Caso contrário, seja  $p$  o ancestral de  $x$  mais próximo que se tornou desregulado.
  - Temos que  $|h_D(p) - h_E(p)| = 2$  pois  $T$  era uma árvore AVL antes da inclusão de  $x$  e, além disso, a inclusão de um nó não pode aumentar em mais de uma unidade a altura de qualquer subárvore.
- Há exatamente dois casos a considerar:
  - **Caso (1):**  $h_E(p) > h_D(p)$
  - **Caso (2):**  $h_E(p) < h_D(p)$

## Caso 1: $h_E(p) > h_D(p)$



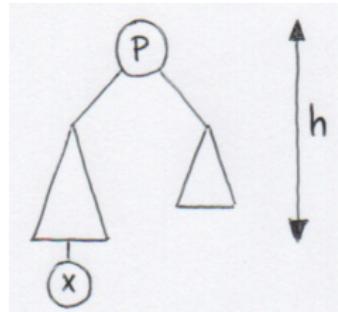
- O nó  $x$  foi inserido na subárvore esquerda de  $p$ .

## Caso 1: $h_E(p) > h_D(p)$



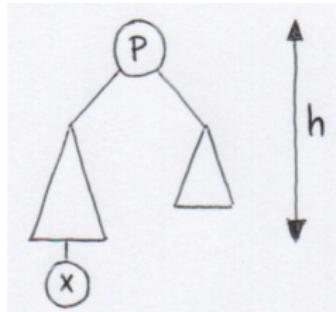
- O nó  $x$  foi inserido na subárvore esquerda de  $p$ .
- $p$  possui o filho esquerdo  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.

## Caso 1: $h_E(p) > h_D(p)$



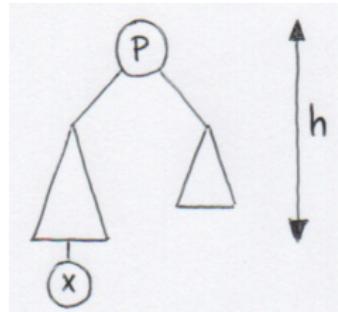
- O nó  $x$  foi inserido na subárvore esquerda de  $p$ .
- $p$  possui o filho esquerdo  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.
- Por esse mesmo motivo, sabe-se que  $h_E(u) \neq h_D(u)$ .

## Caso 1: $h_E(p) > h_D(p)$



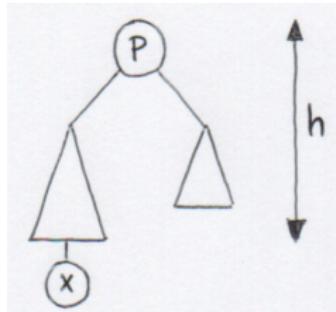
- O nó  $x$  foi inserido na subárvore esquerda de  $p$ .
- $p$  possui o filho esquerdo  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.
- Por esse mesmo motivo, sabe-se que  $h_E(u) \neq h_D(u)$ .
- Há dois subcasos a considerar:

## Caso 1: $h_E(p) > h_D(p)$



- O nó  $x$  foi inserido na subárvore esquerda de  $p$ .
- $p$  possui o filho esquerdo  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.
- Por esse mesmo motivo, sabe-se que  $h_E(u) \neq h_D(u)$ .
- Há dois subcasos a considerar:
  - **Caso 1(a):**  $h_E(u) > h_D(u)$ .

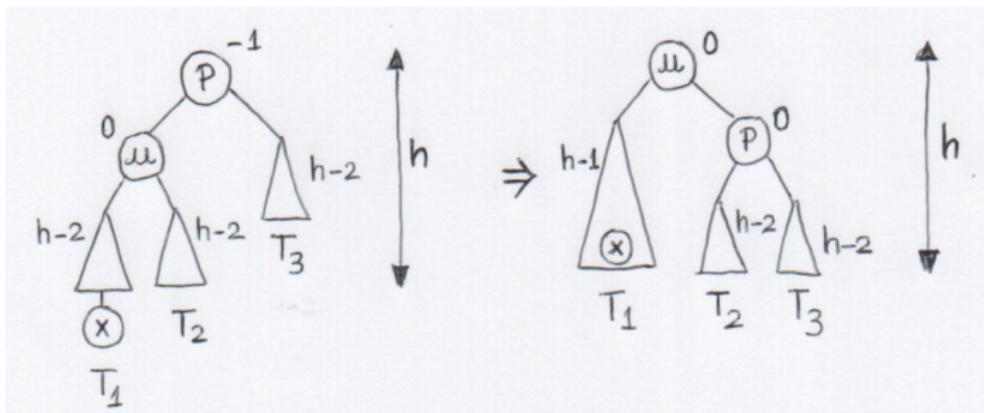
## Caso 1: $h_E(p) > h_D(p)$



- O nó  $x$  foi inserido na subárvore esquerda de  $p$ .
- $p$  possui o filho esquerdo  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.
- Por esse mesmo motivo, sabe-se que  $h_E(u) \neq h_D(u)$ .
- Há dois subcasos a considerar:
  - **Caso 1(a):**  $h_E(u) > h_D(u)$ .
  - **Caso 1(b):**  $h_E(u) < h_D(u)$ .

## Caso 1(a): $h_E(u) > h_D(u)$

**Solução:** Rotação direita simples em  $p$ .



O nó  $x$  é inserido à esquerda de  $u$ .

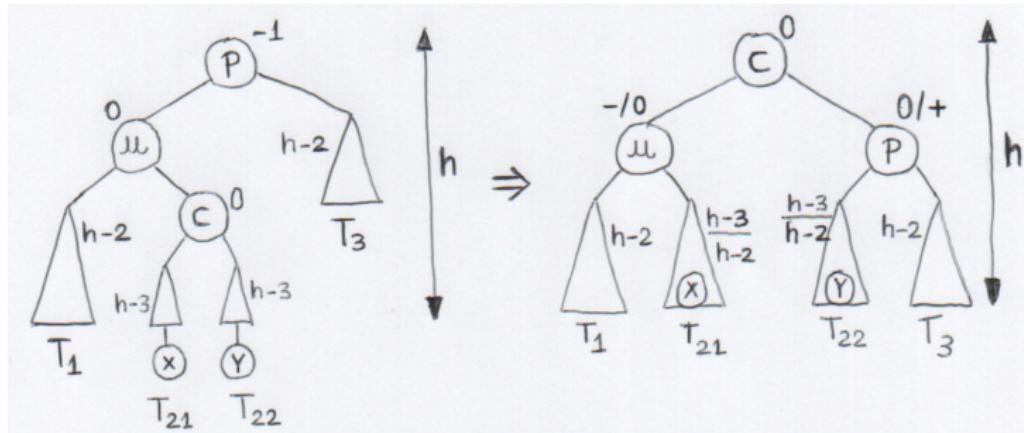
Note que  $h(T_1) = h(T_2) + 1$ .

Após a rotação simples, a altura final permanece inalterada.

**Nenhuma modificação futura é necessária.**

## Caso 1(b): $h_E(u) < h_D(u)$

Solução: Rotação dupla direita em  $p$ .



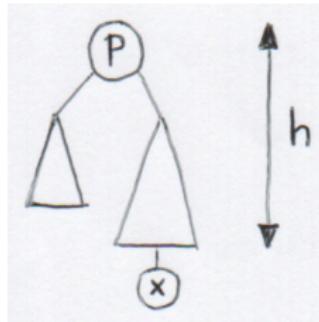
O nó inserido pode ser  $X$  ou  $Y$ .

Quando  $h = 2$ , as árvores  $T_1$  e  $T_3$  são vazias, e o nó inserido é o próprio nó  $C$ ; neste caso as árvores  $T_{21}$  e  $T_{22}$  são vazias.

A altura final permanece inalterada.

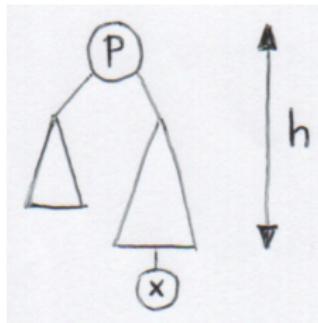
**Nenhuma modificação futura é necessária.**

## Caso 2: $h_E(p) < h_D(p)$



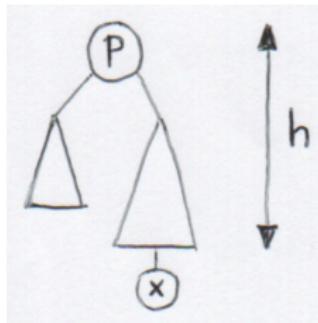
- O nó  $x$  foi inserido na subárvore direita de  $p$ .

## Caso 2: $h_E(p) < h_D(p)$



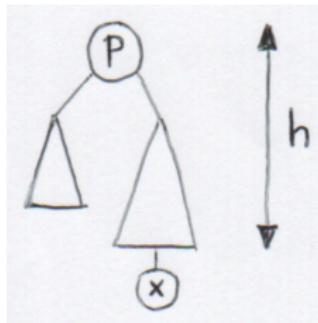
- O nó  $x$  foi inserido na subárvore direita de  $p$ .
- $p$  possui o filho direito  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.

## Caso 2: $h_E(p) < h_D(p)$



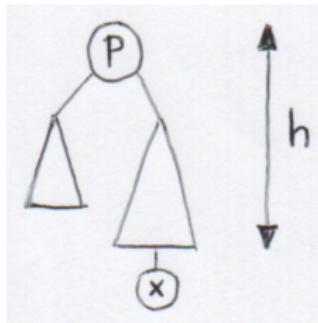
- O nó  $x$  foi inserido na subárvore direita de  $p$ .
- $p$  possui o filho direito  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.
- Por esse mesmo motivo, sabe-se que  $h_E(u) \neq h_D(u)$ .

## Caso 2: $h_E(p) < h_D(p)$



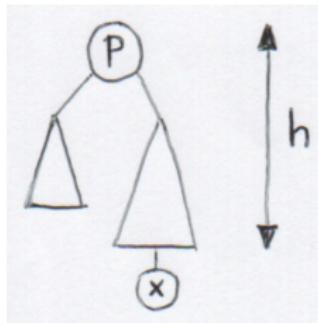
- O nó  $x$  foi inserido na subárvore direita de  $p$ .
- $p$  possui o filho direito  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.
- Por esse mesmo motivo, sabe-se que  $h_E(u) \neq h_D(u)$ .
- Há dois subcasos a considerar:

## Caso 2: $h_E(p) < h_D(p)$



- O nó  $x$  foi inserido na subárvore direita de  $p$ .
- $p$  possui o filho direito  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.
- Por esse mesmo motivo, sabe-se que  $h_E(u) \neq h_D(u)$ .
- Há dois subcasos a considerar:
  - **Caso 2(a):**  $h_E(u) < h_D(u)$ .

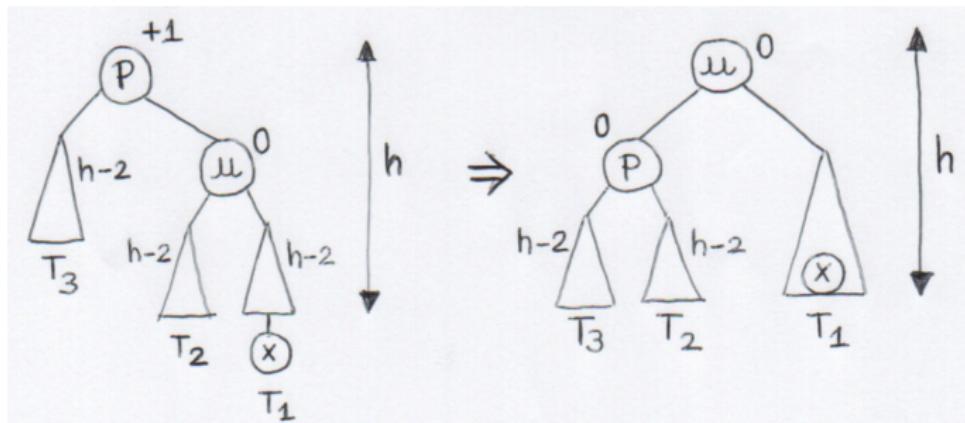
## Caso 2: $h_E(p) < h_D(p)$



- O nó  $x$  foi inserido na subárvore direita de  $p$ .
- $p$  possui o filho direito  $u$ ,  $u \neq x$ . Pois caso contrário,  $p$  não estaria desregulado.
- Por esse mesmo motivo, sabe-se que  $h_E(u) \neq h_D(u)$ .
- Há dois subcasos a considerar:
  - **Caso 2(a):**  $h_E(u) < h_D(u)$ .
  - **Caso 2(b):**  $h_E(u) > h_D(u)$ .

## Caso 2(a): $h_E(u) < h_D(u)$

**Solução:** Rotação esquerda simples em  $p$ .



O nó  $x$  é inserido à direita de  $u$ .

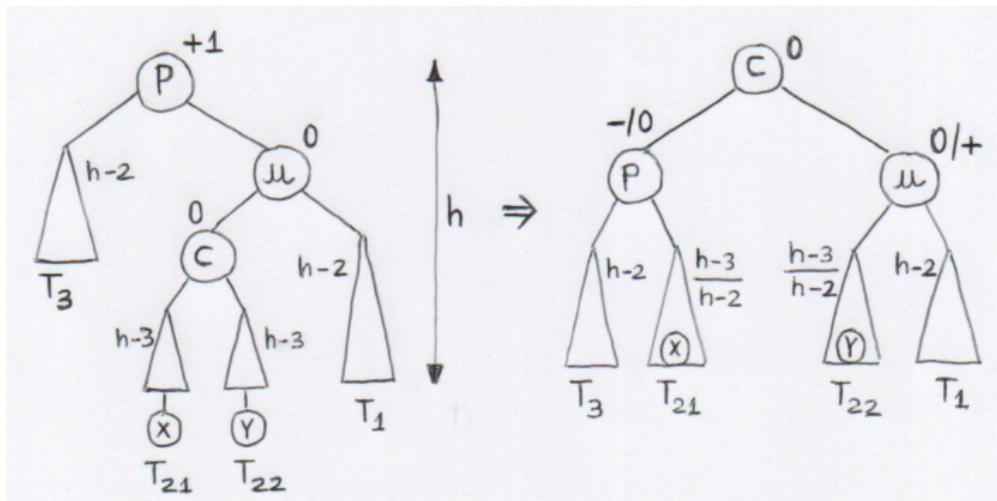
Note que  $h(T_1) = h(T_2) + 1$ .

Após a rotação, a altura final permanece inalterada.

Nenhuma modificação futura é necessária.

## Caso 2(b): $h_E(u) > h_D(u)$

Solução: Rotação dupla esquerda em  $p$ .



O nó inserido pode ser  $X$  ou  $Y$ .

Quando  $h = 2$ , as árvores  $T_1$  e  $T_3$  são vazias, e o nó inserido é o próprio nó  $C$ ; neste caso as árvores  $T_{21}$  e  $T_{22}$  são vazias.

A altura final permanece inalterada.

Nenhuma modificação futura é necessária.

# Propriedades das rotações

## Propriedade 1

As rotações preservam a natureza da árvore como sendo binária de busca.



# Propriedades das rotações

## Propriedade 1

As rotações preservam a natureza da árvore como sendo binária de busca.



## Propriedade 2

Uma vez que um nó  $p$  torna-se desregulado, a regulagem de  $p$  é restabelecida pela aplicação de uma das 4 rotações vistas.



# Propriedades das rotações

## Propriedade 1

As rotações preservam a natureza da árvore como sendo binária de busca.



## Propriedade 2

Uma vez que um nó  $p$  torna-se desregulado, a regulagem de  $p$  é restabelecida pela aplicação de uma das 4 rotações vistas.



## Propriedade 3

Dado um nó  $p$  desregulado, uma rotação apropriada de  $p$  assegura a regulagem de TODOS os nós ancestrais de  $p$ .





# Atividade

Mostre o passo-a-passo da inserção das chaves 1,2,3,4,5,6,7 em uma árvore AVL inicialmente vazia. Em cada passo, ilustre o valor do **fator de balanceamento** de cada nó, assim como as rotações realizadas.



## Implementação da inserção



# Como determinar o fator de balanço de um nó?

Como verificar se algum nó  $v$  de  $T$  se tornou desregulado após a inclusão?

- Basta calcular as alturas de suas subárvores e subtrair uma da outra.

# Como determinar o fator de balanço de um nó?

Como verificar se algum nó  $v$  de  $T$  se tornou desregulado após a inclusão?

- Basta calcular as alturas de suas subárvores e subtrair uma da outra.
- Precisamos fazer isso mantendo o tempo de inserção em  $O(\log n)$ .
  - É possível?

# Como determinar o fator de balanço de um nó?

Como verificar se algum nó  $v$  de  $T$  se tornou desregulado após a inclusão?

- Basta calcular as alturas de suas subárvores e subtrair uma da outra.
- Precisamos fazer isso mantendo o tempo de inserção em  $O(\log n)$ .
  - É possível?

**Ideia:** Cada nó  $v$  da árvore terá um campo adicional chamado `height` que guardará a altura da árvore enraizada em  $v$ .

- Assim, não será preciso percorrer a árvore enraizada em  $v$ .
- Poderemos calcular o fator de balanceamento do nó  $v$  em tempo constante  $O(1)$ .

# Como a altura de cada nó é determinada?

- Cada nó da árvore possui o campo `height`, que guarda sua altura.
- Assim que um nó  $p$  é inserido na árvore ele é um nó folha.
  - A altura de  $p$  é igual a 1 logo após sua inserção.
  - Fazendo `p->height = 1` assim que o nó  $p$  é inserido, determinamos sua altura em tempo  $O(1)$ .

# Como a altura de cada nó é determinada?

- Cada nó da árvore possui o campo `height`, que guarda sua altura.
- Assim que um nó  $p$  é inserido na árvore ele é um nó folha.
  - A altura de  $p$  é igual a 1 logo após sua inserção.
  - Fazendo `p->height = 1` assim que o nó  $p$  é inserido, determinamos sua altura em tempo  $O(1)$ .
- **Observação:** A partir deste momento, os únicos nós da árvore que podem ter alturas modificadas são os nós no caminho de  $p$  até a raiz.

# Como a altura de cada nó é determinada?

- Cada nó da árvore possui o campo `height`, que guarda sua altura.
- Assim que um nó  $p$  é inserido na árvore ele é um nó folha.
  - A altura de  $p$  é igual a 1 logo após sua inserção.
  - Fazendo `p->height = 1` assim que o nó  $p$  é inserido, determinamos sua altura em tempo  $O(1)$ .
- **Observação:** A partir deste momento, os únicos nós da árvore que podem ter alturas modificadas são os nós no caminho de  $p$  até a raiz.
  - Todos eles devem ser verificados e ter seus campos `height` corretamente atualizados.

# Como a altura de cada nó é determinada?

- Cada nó da árvore possui o campo `height`, que guarda sua altura.
- Assim que um nó  $p$  é inserido na árvore ele é um nó folha.
  - A altura de  $p$  é igual a 1 logo após sua inserção.
  - Fazendo `p->height = 1` assim que o nó  $p$  é inserido, determinamos sua altura em tempo  $O(1)$ .
- **Observação:** A partir deste momento, os únicos nós da árvore que podem ter alturas modificadas são os nós no caminho de  $p$  até a raiz.
  - Todos eles devem ser verificados e ter seus campos `height` corretamente atualizados.
  - Existem  $O(\log n)$  destes nós e essa atualização pode ser feita em tempo constante à medida que as chamadas recursivas “se desenrolam”.

# Arquivo Node.h

```
1 #ifndef NODE_H
2 #define NODE_H
3
4 struct Node {
5     int key;
6     int height;
7     Node *left;
8     Node *right;
9 };
10
11 #endif
```

# Arquivo Tree.h (com código inicial)

```
1 #ifndef TREE_H
2 #define TREE_H
3 #include "Node.h"
4
5 class Tree {
6 public:
7     Tree() = default;
8     void add(int key);
9     ~Tree();
10
11 private:
12     Node *root {nullptr};
13     int height(Node *node);
14     int balance(Node *node);
15     Node* rightRotation(Node *p);
16     Node* leftRotation(Node *p);
17     Node* add(Node *p, int key);
18     Node* fixup_node(Node *p, int key);
19 };
20
21 #endif
```

# Determinando fator de balanceamento de um nó

Calculamos o fator de平衡amento de um nó v através da subtração das alturas das subárvores esquerda e direita do nó:

# Determinando fator de balanceamento de um nó

Calculamos o fator de balanceamento de um nó v através da subtração das alturas das subárvores esquerda e direita do nó:

```
1 int Tree::height(Node *node) {  
2     return (node == nullptr) ? 0 : node->height;  
3 }
```

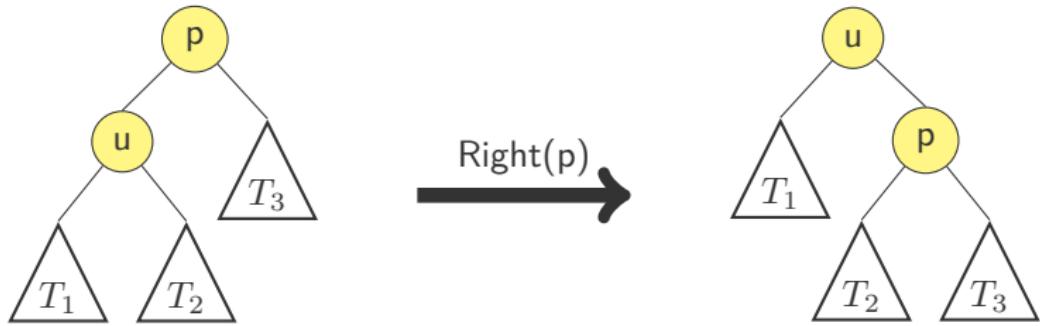
# Determinando fator de balanceamento de um nó

Calculamos o fator de balanceamento de um nó v através da subtração das alturas das subárvores esquerda e direita do nó:

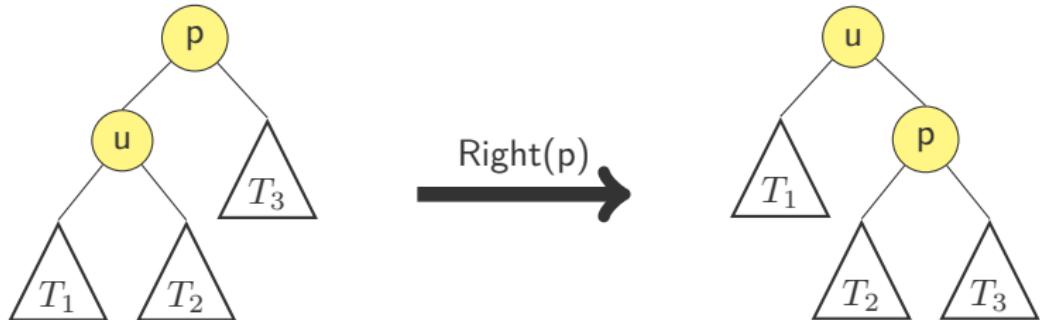
```
1 int Tree::height(Node *node) {
2     return (node == nullptr) ? 0 : node->height;
3 }

1 int Tree::balance(Node *node) {
2     return height(node->right) - height(node->left);
3 }
```

# Rotação Direita



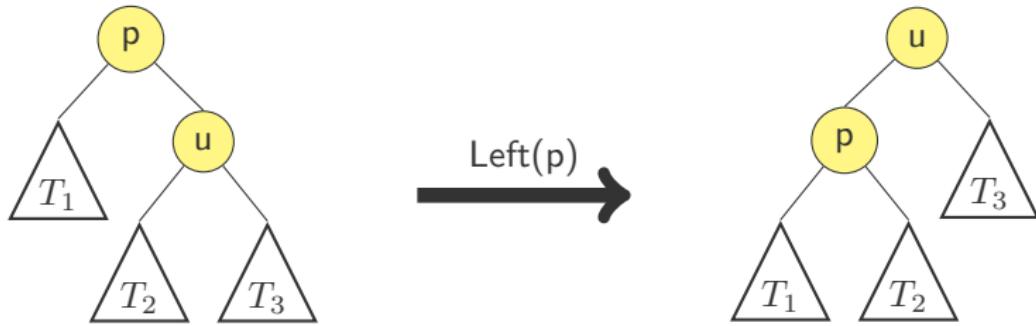
# Rotação Direita



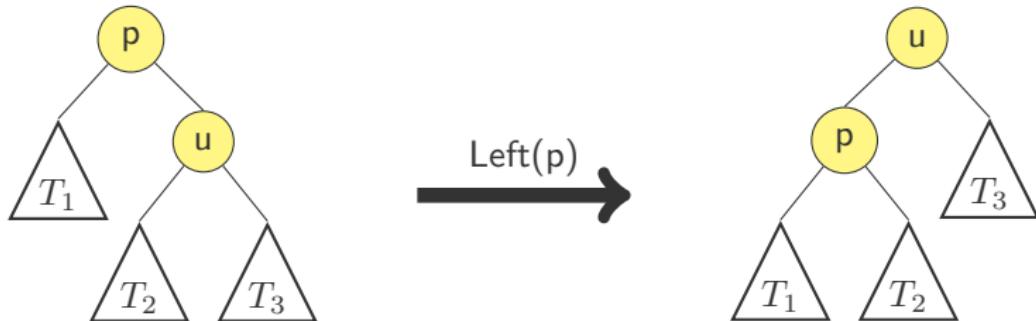
```

1 Node* Tree::rightRotation(Node *p) {
2     Node *u = p->left;
3     p->left = u->right;
4     u->right = p;
5     // atualiza altura dos nodes
6     p->height = 1 + max(height(p->left),height(p->right));
7     u->height = 1 + max(height(u->left),height(u->right));
8     return u; // nova raiz
9 }
```

# Rotação Esquerda



# Rotação Esquerda



```

1 Node* Tree::leftRotation(Node *p) {
2     Node *u = p->right;
3     p->right = u->left;
4     u->left = p;
5     // atualiza altura dos nodes
6     p->height = 1 + max(height(p->left),height(p->right));
7     u->height = 1 + max(height(u->left),height(u->right));
8     return u; // nova raiz
9 }
```

# Inserção

Função pública:

```
1 void Tree::add(int key) {  
2     root = add(root, key);  
3 }
```

# Inserção

Função privada:

```
1 Node* Tree::add(Node *p, int key) {
2     if(p == nullptr) // subarvore vazia
3         return new Node{key, 1, nullptr, nullptr};
```

# Inserção

Função privada:

```
1 Node* Tree::add(Node *p, int key) {
2     if(p == nullptr) // subarvore vazia
3         return new Node(key, 1, nullptr, nullptr);
4     if(key == p->key) // chave repetida
5         return p;
```

# Inserção

Função privada:

```
1 Node* Tree::add(Node *p, int key) {
2     if(p == nullptr) // subarvore vazia
3         return new Node(key, 1, nullptr, nullptr);
4     if(key == p->key) // chave repetida
5         return p;
6     if(key < p->key)
7         p->left = add(p->left, key);
8     else
9         p->right = add(p->right, key);
```

# Inserção

Função privada:

```
1 Node* Tree::add(Node *p, int key) {
2     if(p == nullptr) // subarvore vazia
3         return new Node(key, 1, nullptr, nullptr);
4     if(key == p->key) // chave repetida
5         return p;
6     if(key < p->key)
7         p->left = add(p->left, key);
8     else
9         p->right = add(p->right, key);
10    p = fixup_node(p, key); // regula o node p
11
12
13    return p;
14 }
```

# Inserção

```
1 Node* Tree::fixup_node(Node *p, int key) {
2     // obtém balanço de p
3     int bal = balance(p);
```



# Inserção

```
1 Node* Tree::fixup_node(Node *p, int key) {
2     // obtém balanço de p
3     int bal = balance(p);
4     // Caso 1(a): rotacao direita
5     if(bal < -1 && key < p->left->key)
6         return rightRotation(p);
```



# Inserção

```
1 Node* Tree::fixup_node(Node *p, int key) {
2     // obtém balanço de p
3     int bal = balance(p);
4     // Caso 1(a): rotacao direita
5     if(bal < -1 && key < p->left->key)
6         return rightRotation(p);
7     // Caso 1(b): rotacao dupla direita
8     else if(bal < -1 && key > p->left->key) {
9         p->left = leftRotation(p->left);
10        return rightRotation(p);
11    }
```



# Inserção

```
1 Node* Tree::fixup_node(Node *p, int key) {
2     // obtém balanço de p
3     int bal = balance(p);
4     // Caso 1(a): rotacao direita
5     if(bal < -1 && key < p->left->key)
6         return rightRotation(p);
7     // Caso 1(b): rotacao dupla direita
8     else if(bal < -1 && key > p->left->key) {
9         p->left = leftRotation(p->left);
10        return rightRotation(p);
11    }
12    // Caso 2(a): rotacao esquerda
13    else if(bal > 1 && key > p->right->key)
14        return leftRotation(p);
```

# Inserção

```
1 Node* Tree::fixup_node(Node *p, int key) {
2     // obtém balanço de p
3     int bal = balance(p);
4     // Caso 1(a): rotacao direita
5     if(bal < -1 && key < p->left->key)
6         return rightRotation(p);
7     // Caso 1(b): rotacao dupla direita
8     else if(bal < -1 && key > p->left->key) {
9         p->left = leftRotation(p->left);
10        return rightRotation(p);
11    }
12    // Caso 2(a): rotacao esquerda
13    else if(bal > 1 && key > p->right->key)
14        return leftRotation(p);
15    // Caso 2(b): rotacao dupla esquerda
16    else if(bal > 1 && key < p->right->key) {
17        p->right = rightRotation(p->right);
18        return leftRotation(p);
19    }
```

# Inserção

```
1 Node* Tree::fixup_node(Node *p, int key) {
2     // obtém balanço de p
3     int bal = balance(p);
4     // Caso 1(a): rotacao direita
5     if(bal < -1 && key < p->left->key)
6         return rightRotation(p);
7     // Caso 1(b): rotacao dupla direita
8     else if(bal < -1 && key > p->left->key) {
9         p->left = leftRotation(p->left);
10        return rightRotation(p);
11    }
12    // Caso 2(a): rotacao esquerda
13    else if(bal > 1 && key > p->right->key)
14        return leftRotation(p);
15    // Caso 2(b): rotacao dupla esquerda
16    else if(bal > 1 && key < p->right->key) {
17        p->right = rightRotation(p->right);
18        return leftRotation(p);
19    }
20    // atualiza altura deste node ancestral p
21    p->height = 1 + max(height(p->left),height(p->right));
```

# Inserção

```
1 Node* Tree::fixup_node(Node *p, int key) {
2     // obtém balanço de p
3     int bal = balance(p);
4     // Caso 1(a): rotacao direita
5     if(bal < -1 && key < p->left->key)
6         return rightRotation(p);
7     // Caso 1(b): rotacao dupla direita
8     else if(bal < -1 && key > p->left->key) {
9         p->left = leftRotation(p->left);
10        return rightRotation(p);
11    }
12    // Caso 2(a): rotacao esquerda
13    else if(bal > 1 && key > p->right->key)
14        return leftRotation(p);
15    // Caso 2(b): rotacao dupla esquerda
16    else if(bal > 1 && key < p->right->key) {
17        p->right = rightRotation(p->right);
18        return leftRotation(p);
19    }
20    // atualiza altura deste node ancestral p
21    p->height = 1 + max(height(p->left),height(p->right));
22    return p;
23 }
```



# Remoção



# Remoção em árvores AVL

- A remoção também pode ser feita em  $O(\log n)$ .
- Após a exclusão da chave, verificamos se a árvore se tornou desregulada.
- Assim como na inserção, os nós a serem examinados pertencem ao caminho da raiz até uma de suas folhas.
- Ao contrário da inserção, agora o número de rotações necessárias para a regulagem da árvore pode atingir  $O(\log n)$ .

# Algoritmo de remoção em árvores AVL

1. Fazemos uma busca pelo nó a ser removido.

# Algoritmo de remoção em árvores AVL

1. Fazemos uma busca pelo nó a ser removido.
2. Se o nó encontrado for nulo, então a árvore é vazia ou a chave não existe na árvore. Não há o que remover neste caso.

# Algoritmo de remoção em árvores AVL

1. Fazemos uma busca pelo nó a ser removido.
2. Se o nó encontrado for nulo, então a árvore é vazia ou a chave não existe na árvore. Não há o que remover neste caso.
3. Caso contrário, uma vez encontrado o nó  $x$  com a chave desejada, tratamos de removê-lo da árvore. Há somente dois casos a considerar:

# Algoritmo de remoção em árvores AVL

1. Fazemos uma busca pelo nó a ser removido.
2. Se o nó encontrado for nulo, então a árvore é vazia ou a chave não existe na árvore. Não há o que remover neste caso.
3. Caso contrário, uma vez encontrado o nó  $x$  com a chave desejada, tratamos de removê-lo da árvore. Há somente dois casos a considerar:
  - (a) Se o nó  $x$  não tiver filho direito, então o seu filho esquerdo (seja ele vazio ou não) assume o papel de  $x$  e o nó  $x$  é liberado.

# Algoritmo de remoção em árvores AVL

1. Fazemos uma busca pelo nó a ser removido.
2. Se o nó encontrado for nulo, então a árvore é vazia ou a chave não existe na árvore. Não há o que remover neste caso.
3. Caso contrário, uma vez encontrado o nó  $x$  com a chave desejada, tratamos de removê-lo da árvore. Há somente dois casos a considerar:
  - (a) Se o nó  $x$  não tiver filho direito, então o seu filho esquerdo (seja ele vazio ou não) assume o papel de  $x$  e o nó  $x$  é liberado.
    - Todos os ancestrais de  $x$  devem ter suas alturas atualizadas e devem ser regulados, caso necessário, por meio de rotação apropriada.

# Algoritmo de remoção em árvores AVL

1. Fazemos uma busca pelo nó a ser removido.
2. Se o nó encontrado for nulo, então a árvore é vazia ou a chave não existe na árvore. Não há o que remover neste caso.
3. Caso contrário, uma vez encontrado o nó  $x$  com a chave desejada, tratamos de removê-lo da árvore. Há somente dois casos a considerar:
  - (a) Se o nó  $x$  não tiver filho direito, então o seu filho esquerdo (seja ele vazio ou não) assume o papel de  $x$  e o nó  $x$  é liberado.
    - Todos os ancestrais de  $x$  devem ter suas alturas atualizadas e devem ser regulados, caso necessário, por meio de rotação apropriada.
  - (b) Se o nó  $x$  tiver filho direito, **trocamos a chave** de  $x$  com a chave do seu nó sucessor e o nó sucessor é liberado.

# Algoritmo de remoção em árvores AVL

1. Fazemos uma busca pelo nó a ser removido.
2. Se o nó encontrado for nulo, então a árvore é vazia ou a chave não existe na árvore. Não há o que remover neste caso.
3. Caso contrário, uma vez encontrado o nó  $x$  com a chave desejada, tratamos de removê-lo da árvore. Há somente dois casos a considerar:
  - (a) Se o nó  $x$  não tiver filho direito, então o seu filho esquerdo (seja ele vazio ou não) assume o papel de  $x$  e o nó  $x$  é liberado.
    - Todos os ancestrais de  $x$  devem ter suas alturas atualizadas e devem ser regulados, caso necessário, por meio de rotação apropriada.
  - (b) Se o nó  $x$  tiver filho direito, **trocamos a chave** de  $x$  com a chave do seu nó sucessor e o nó sucessor é liberado.
    - Todos os nós que estiverem no caminho do antigo pai do sucessor de  $x$  até a raiz da árvore devem ter suas alturas atualizadas e devem ser regulados, caso necessário, por meio de rotação apropriada.

# Análise do balanceamento na remoção

- Os únicos nós que podem ter se tornado desregulados após a remoção são os ancestrais do nó **fisicamente** removido.
- Analisaremos os casos que podem influenciar no fator de平衡amento de um nó  $p$  quando um nó  $x$  é removido do lado esquerdo de  $p$ .

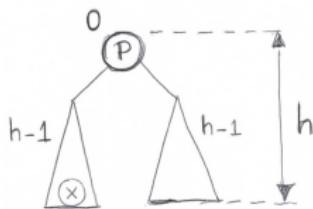
# Análise do balanceamento na remoção

- Os únicos nós que podem ter se tornado desregulados após a remoção são os ancestrais do nó **fisicamente** removido.
- Analisaremos os casos que podem influenciar no fator de平衡amento de um nó  $p$  quando um nó  $x$  é removido do lado esquerdo de  $p$ .

**Atenção:** Os casos em que o nó  $x$  é removido do lado direito de  $p$  são simétricos aos apresentados nestes slides, e sua análise e exame será deixada como exercício para casa.

# Análise do balanceamento após a remoção

**Caso 1:**  $h_E(p) = h_D(p)$  antes da remoção de  $x$ , e o nó  $x$  foi removido da subárvore esquerda de  $p$ .

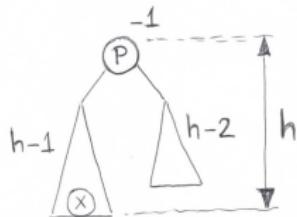


Neste caso, a altura do nó  $p$  permanece inalterada, e  $fb(p) \in \{0, +1\}$  após a remoção de  $x$ .

**Nenhuma regulagem é necessária no nó  $p$  e nem em nenhum de seus ancestrais.**

# Análise do balanceamento após a remoção

**Caso 2:**  $h_E(p) > h_D(p)$  antes da remoção do nó  $x$ , e o nó  $x$  foi removido da subárvore esquerda de  $p$ .



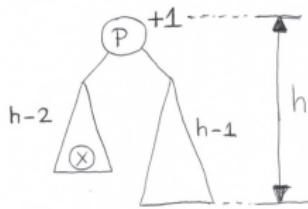
Neste caso, a subárvore esquerda pode ou não diminuir de tamanho e  $fb(p) \in \{0, -1\}$  após a remoção de  $x$ .

**Nenhuma regulagem é necessária no nó  $p$ .** Porém, caso a altura de  $p$  venha diminuir, algum ancestral de  $p$  pode ter se tornado desregulado.

# Análise do balanceamento após a remoção

**Caso 3:**  $h_E(p) < h_D(p)$  e o nó  $x$  é removido da subárvore esquerda de  $p$ .

Antes da remoção de  $x$ ,  $fb(p) = +1$  e, após a remoção,  $fb(p) \in \{+1, +2\}$ .

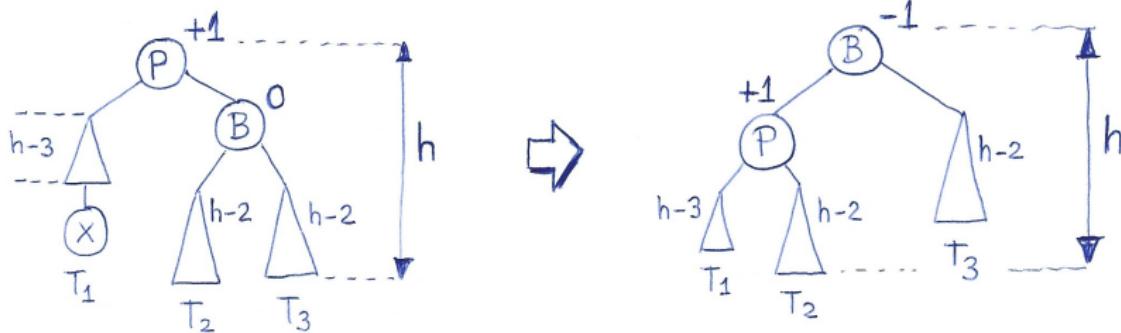


No caso em que  $fb(p) = +2$  após a remoção, há três subcasos a considerar, dependendo do fator de平衡amento do filho direito da raiz  $p$ , que pode ser 0, +1 ou -1.

# Análise do balanceamento após a remoção

Caso 3(a): Filho direito de  $p$  tem balanço = 0.

Antes da remoção de  $x$ , balanço( $p$ ) = +1 e, após, balanço( $p$ ) = +2.



**Solução:** Rotação esquerda em  $p$ .

Os balanços são ajustados!

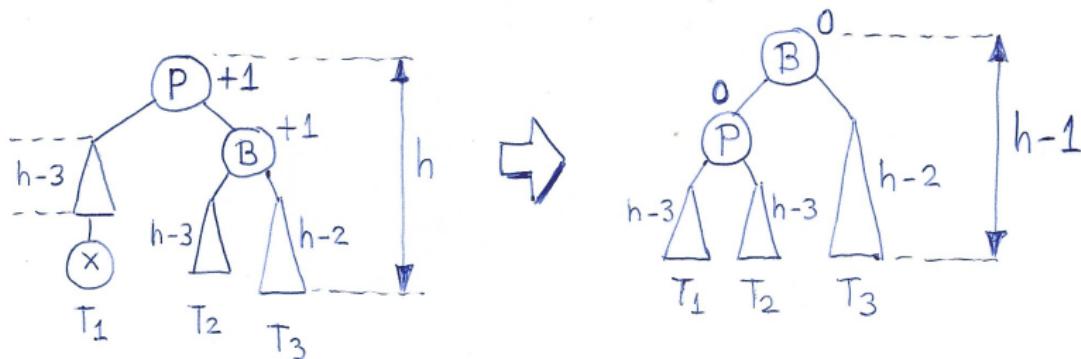
A altura permanece inalterada.

Nenhuma regulagem acontecerá mais.

# Análise do balanceamento após a remoção

**Caso 3(b):** Filho direito de  $p$  tem balanço = +1.

Antes da remoção de  $x$ , balanço( $p$ ) = +1 e, após, balanço( $p$ ) = +2.



**Solução:** Rotação esquerda em  $p$ .

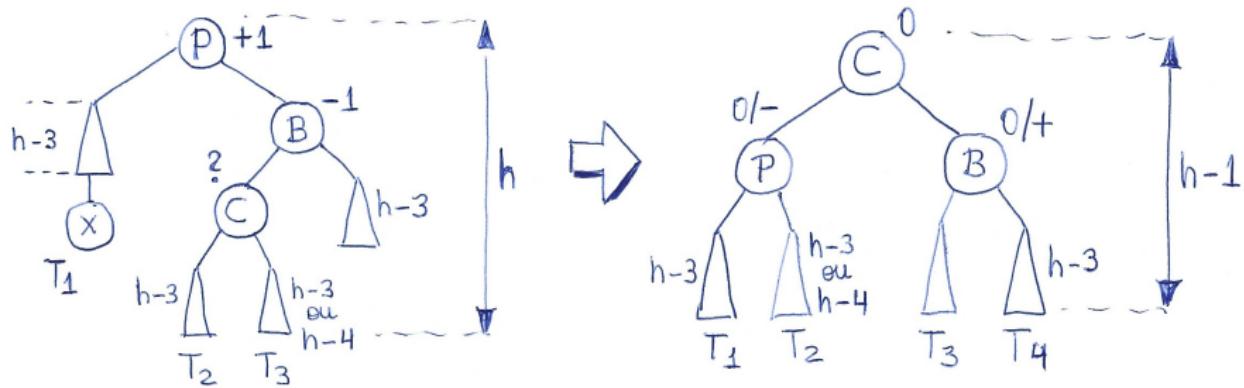
Os balanços são ajustados!

A altura diminui (algum ancestral pode ter se tornado desregulado).

# Análise do balanceamento após a remoção

**Caso 3(c):** Filho direito de  $p$  tem balanço = -1.

Antes da remoção de  $x$ , balanço( $p$ ) = +1 e, após, balanço( $p$ ) = +2.



**Solução:** Rotação esquerda dupla no  $p$ .

Os balanços são ajustados!

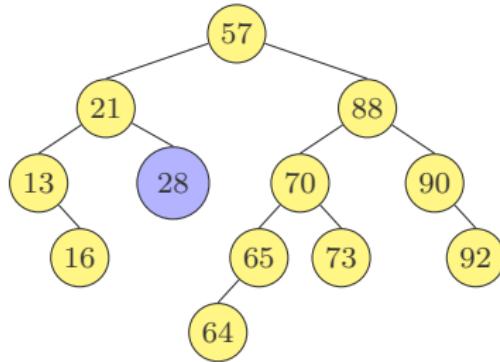
A altura diminui (algum ancestral pode ter se tornado desregulado).

# Remoção — Função privada de rebalanceamento

```
1 Node* avl_tree::fixup_deletion(Node *node) {
2     int bal = balance(node);
3     // node pode estar desregulado, ha 4 casos a considerar
4     if(bal > 1 && balance(node->right) >= 0) {
5         return leftRotation(node);
6     }
7     else if(bal > 1 && balance(node->right) < 0) {
8         node->right = rightRotation(node->right);
9         return leftRotation(node);
10    }
11    else if(bal < -1 && balance(node->left) <= 0) {
12        return rightRotation(node);
13    }
14    else if(bal < -1 && balance(node->left) > 0) {
15        node->left = leftRotation(node->left);
16        return rightRotation(node);
17    }
18    node->height =
19        1 + max(height(node->left),height(node->right));
20    return node;
21 }
```

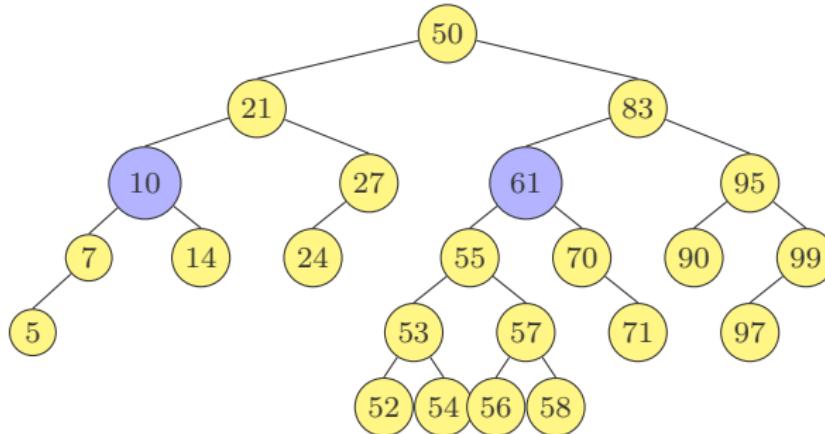
# Remoção — Exercício 1

Excluir o nó 28 e fazer as regulagens para manter a árvore AVL.



## Remoção — Exercício 2

Excluir a chave 61 e, depois, a chave 10 da árvore abaixo e fazer as regulagens necessárias.



# Remoção — Função Pública

```
1 void avl_tree::remove(int key) {
2     root = remove(root, key);
3 }
```

# Remoção – Função privada que busca o nó

```
1 Node* avl_tree::remove(Node *node, int key) {
2     if(node == nullptr) // node não encontrado
3         return nullptr;
```

# Remoção – Função privada que busca o nó

```
1 Node* avl_tree::remove(Node *node, int key) {
2     if(node == nullptr) // node não encontrado
3         return nullptr;
4     if(key < node->key)
5         node->left = remove(node->left, key);
6     else if(key > node->key)
7         node->right = remove(node->right, key);
```

# Remoção – Função privada que busca o nó

```
1 Node* avl_tree::remove(Node *node, int key) {
2     if(node == nullptr) // node não encontrado
3         return nullptr;
4     if(key < node->key)
5         node->left = remove(node->left, key);
6     else if(key > node->key)
7         node->right = remove(node->right, key);
8     // encontramos no node
9     else if(node->right == nullptr) { // sem filho direito
10         Node *child = node->left;
11         delete node;
12         return child;
13     }
14     else // tem filho direito: troca pelo sucessor
15         node->right = remove_successor(node, node->right);
```

# Remoção – Função privada que busca o nó

```
1 Node* avl_tree::remove(Node *node, int key) {
2     if(node == nullptr) // node não encontrado
3         return nullptr;
4     if(key < node->key)
5         node->left = remove(node->left, key);
6     else if(key > node->key)
7         node->right = remove(node->right, key);
8     // encontramos no node
9     else if(node->right == nullptr) { // sem filho direito
10         Node *child = node->left;
11         delete node;
12         return child;
13     }
14     else // tem filho direito: troca pelo sucessor
15         node->right = remove_successor(node, node->right);
16
17     // Atualiza a altura do node e regula o node
18     node = fixup_deletion(node);
19     return node;
20 }
```

# Remoção

Função privada que remove o sucessor

```
1 Node* avl_tree::remove_successor(Node *root, Node *node) {
2     if(node->left != nullptr)
3         node->left = remove_successor(root, node->left);
4     else {
5         root->key = node->key;
6         Node *aux = node->right;
7         delete node;
8         return aux;
9     }
10    // Atualiza a altura do node e regula o node
11    node = fixup_deletion(node);
12    return node;
13 }
```



FIM

