DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

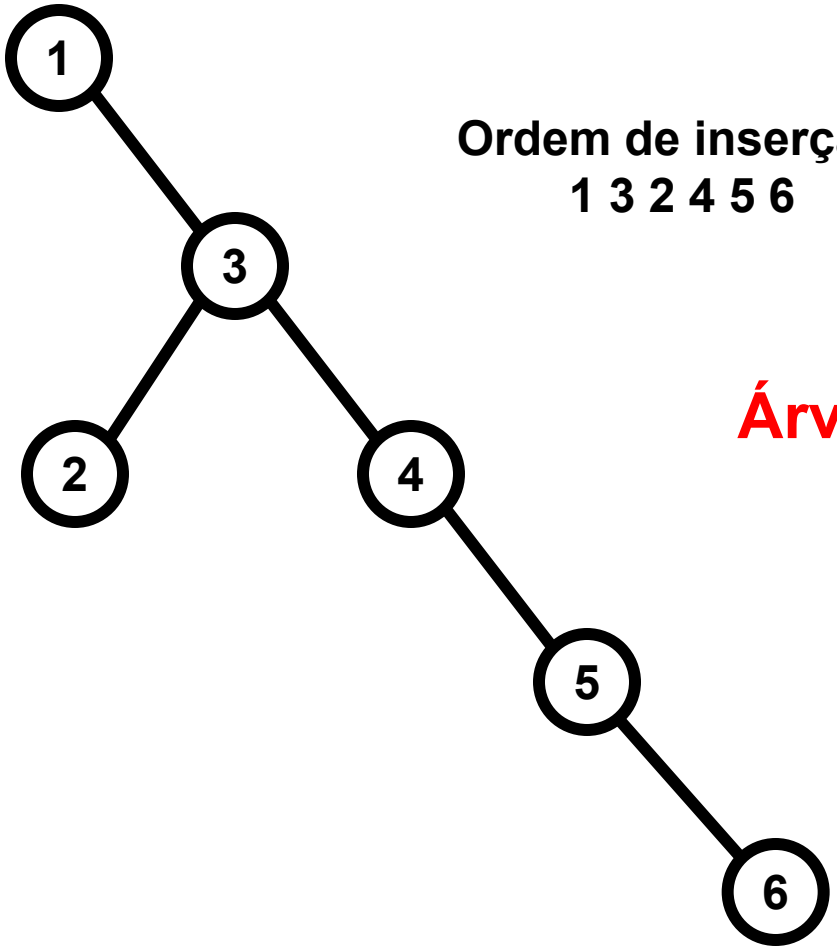
Estrutura de Dados

Árvores de Pesquisa Balanceadas

Professores: Anisio Lacerda e Wagner Meira Jr.

Árvores binárias de pesquisa

- Pior caso para uma busca é O(n)



Ordem de inserção:
1 3 2 4 5 6

Árvore desbalanceada!

Árvore completamente balanceada

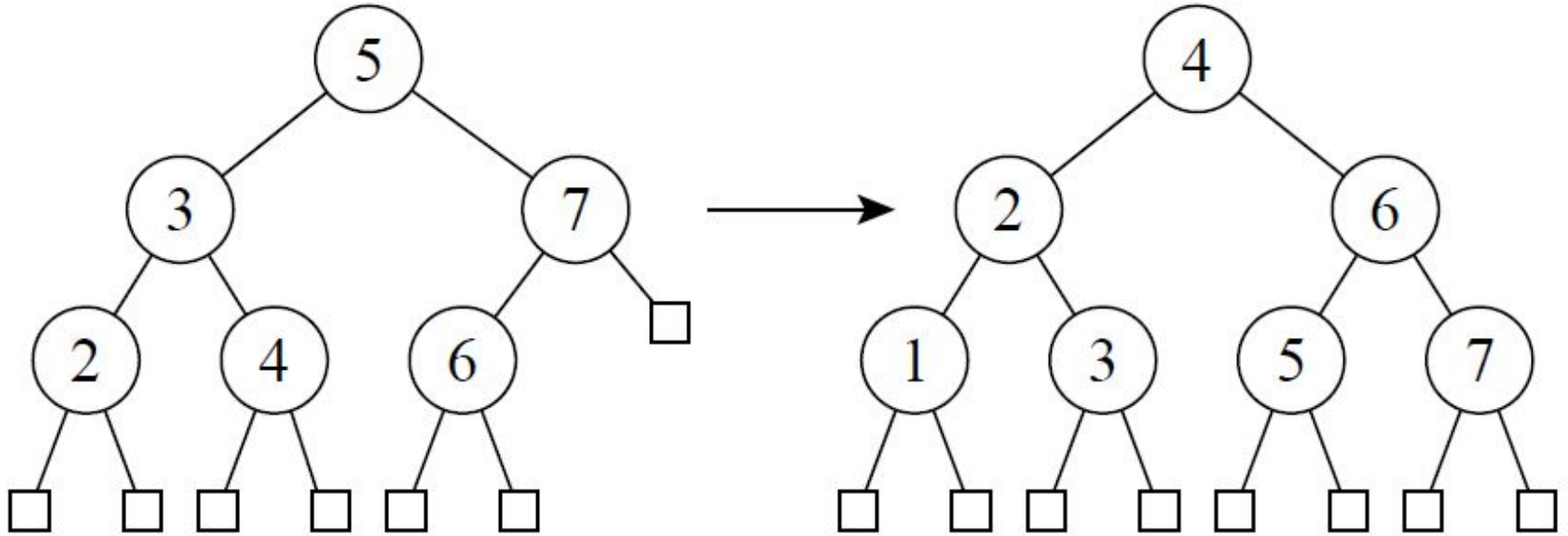
- Nós folha (externos) aparecem em no máximo dois níveis diferentes

- Minimiza o tempo médio de pesquisa
  - Assumindo distribuição uniforme das chaves

- Problema: manter árvore completamente balanceada após cada inserção é muito caro

Árvore completamente balanceada

- Para inserir a chave 1 na árvore à esquerda e manter a árvore completamente balanceada precisamos movimentar todos os nós



Árvores Balanceadas

- A solução é criar estruturas que mantenham um certo balanceamento mas que não exijam o balanceamento completo
  - Tradeoff entre o custo de pesquisa e manutenção
  - Garantias de custos máximos
- Exemplos
  - Árvores 2-3-4
  - Árvores Vermelha e Preta
  - Árvores SBB
  - Árvores AVL

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estrutura de Dados

Árvore AVL

Professores: Luiz Chaimowicz e Raquel Prates

Árvore AVL

- Árvore Binária de Pesquisa Balanceada
- Foi proposta por Georgy Adelson-Velsky e Yevgeniy Landis em 1962
- Ideia: À medida em que as operações de inserção e remoção são efetuadas a árvore é balanceada

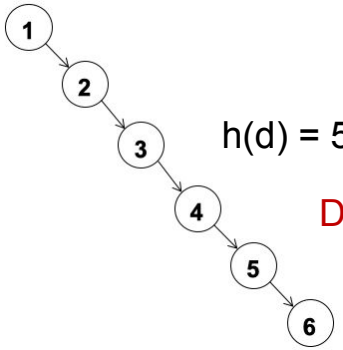
AVL – CONCEITOS E OPERAÇÕES BÁSICAS

Balanceamento

- Considera a altura da árvore
- Fator de Balanceamento (FB): Subtrai a altura da subárvore da direita da altura da subárvore da esquerda
  - Se o Fator de Balanceamento for: -1<=0<=1 → Balanceada
  - Se for <-1 ou >1 → Desbalanceada

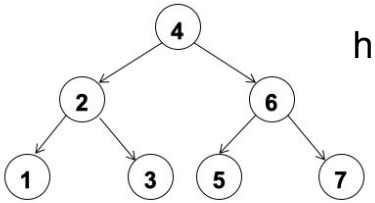
Balanceamento - Exemplo

<-1 ou >1 → Desbalanceada
-1<=0<=1 → Balanceada



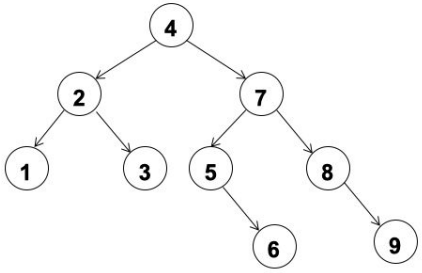
h(d) = 5 e h(e) = 0 → FB = 5

Desbalanceada!



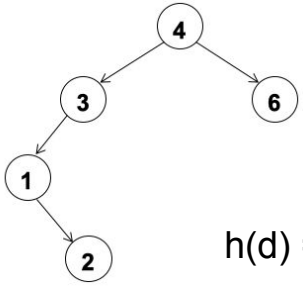
h(d) = 2 e h(e) = 2 → FB = 0

Balanceada!



h(d) = 3 e h(e) =2 → FB = 1

Balanceada!

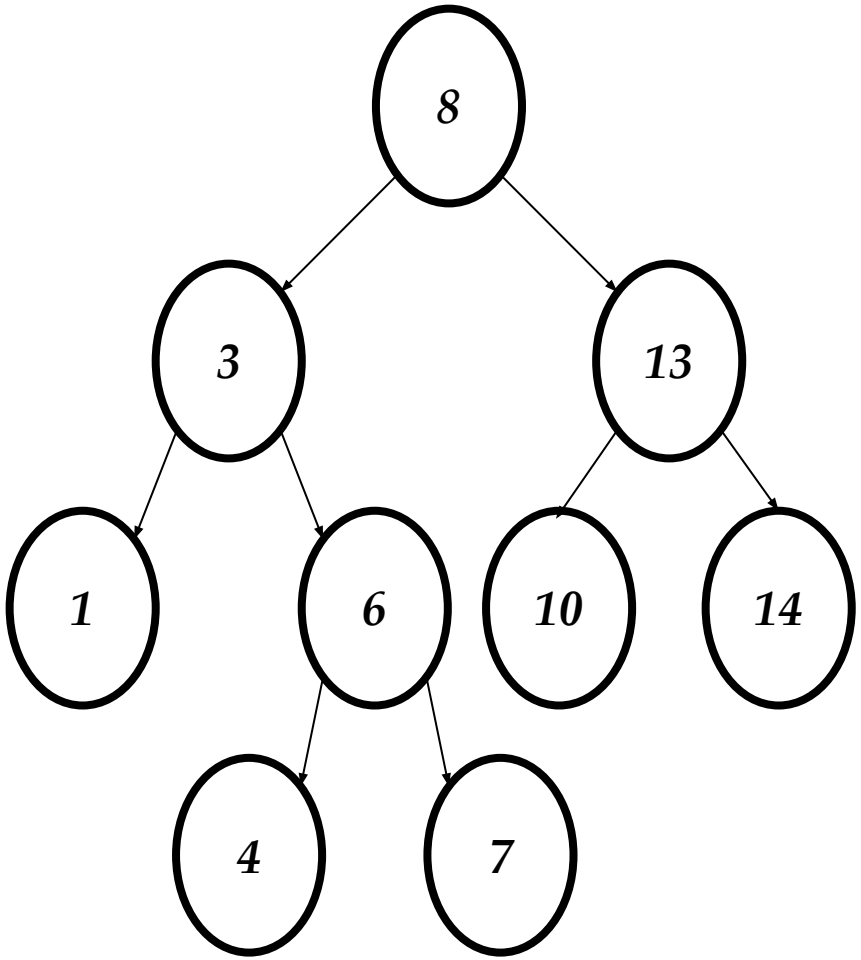


h(d) = 1 e h(e) = 3 → FB = -2

Desbalanceada!

Árvores Balanceadas

- Como computar o balanceamento de um nó?

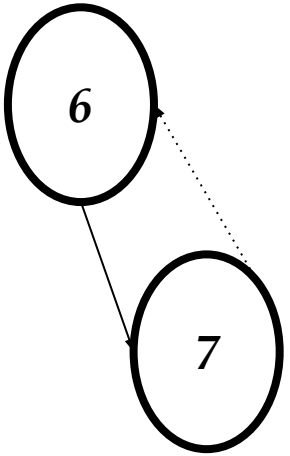


Representando o nó

- Por simplicidade
  - Vamos assumir que é fácil achar o ascendente direto de um nó
  - Para isto podemos apenas colocar um ponteiro para cima
  - Outra opção é uma função parent

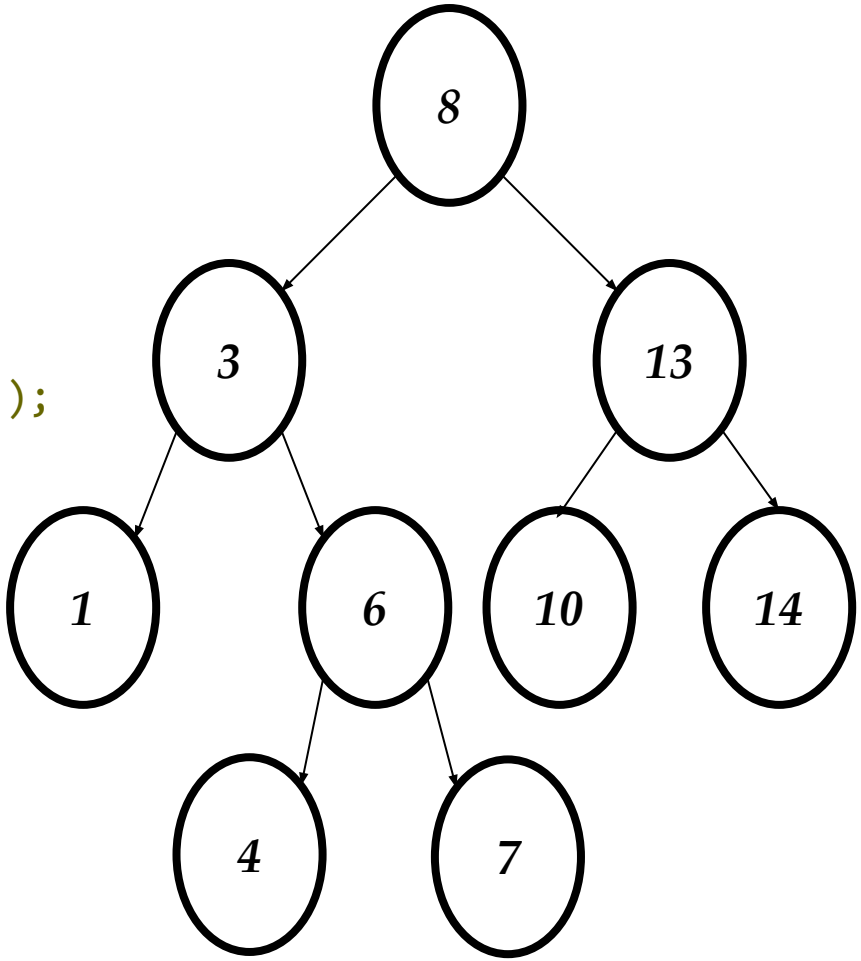
```
typedef struct node {
  int value;
  struct node *leftChild;
  struct node *rightChild;
  struct node *parent;
} node_t;
```

- Não mostramos os links para cima nas figuras
  - Pois podemos fazer tudo sem eles se necessário



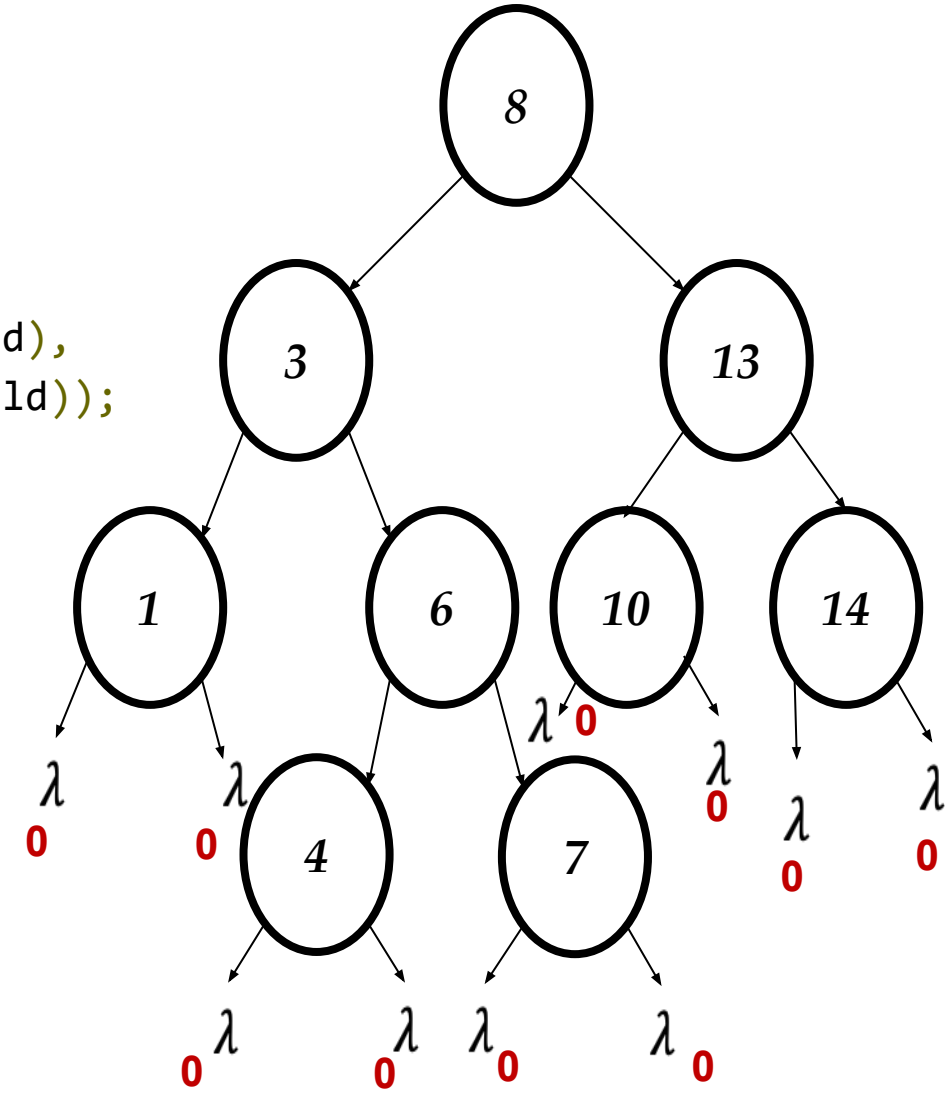
Altura de um Nó

```
int height(node_t *node) {
  if (node == NULL) return 0;
  return 1 + max(height(node->leftChild),
                 height(node->rightChild));
}
```



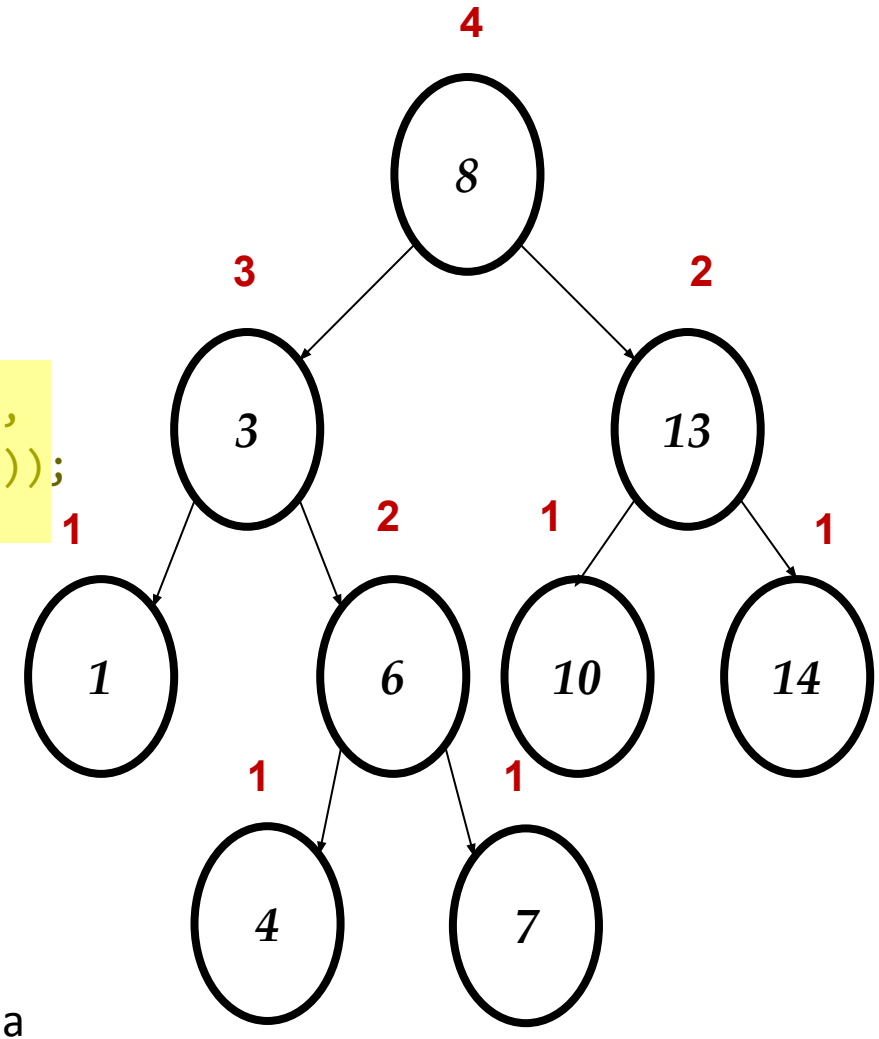
Altura de um Nó

```
int height(node_t *node) {
  if (node == NULL) return 0;
  return 1 + max(height(node->leftChild),
                 height(node->rightChild));
}
```



Altura de um Nó

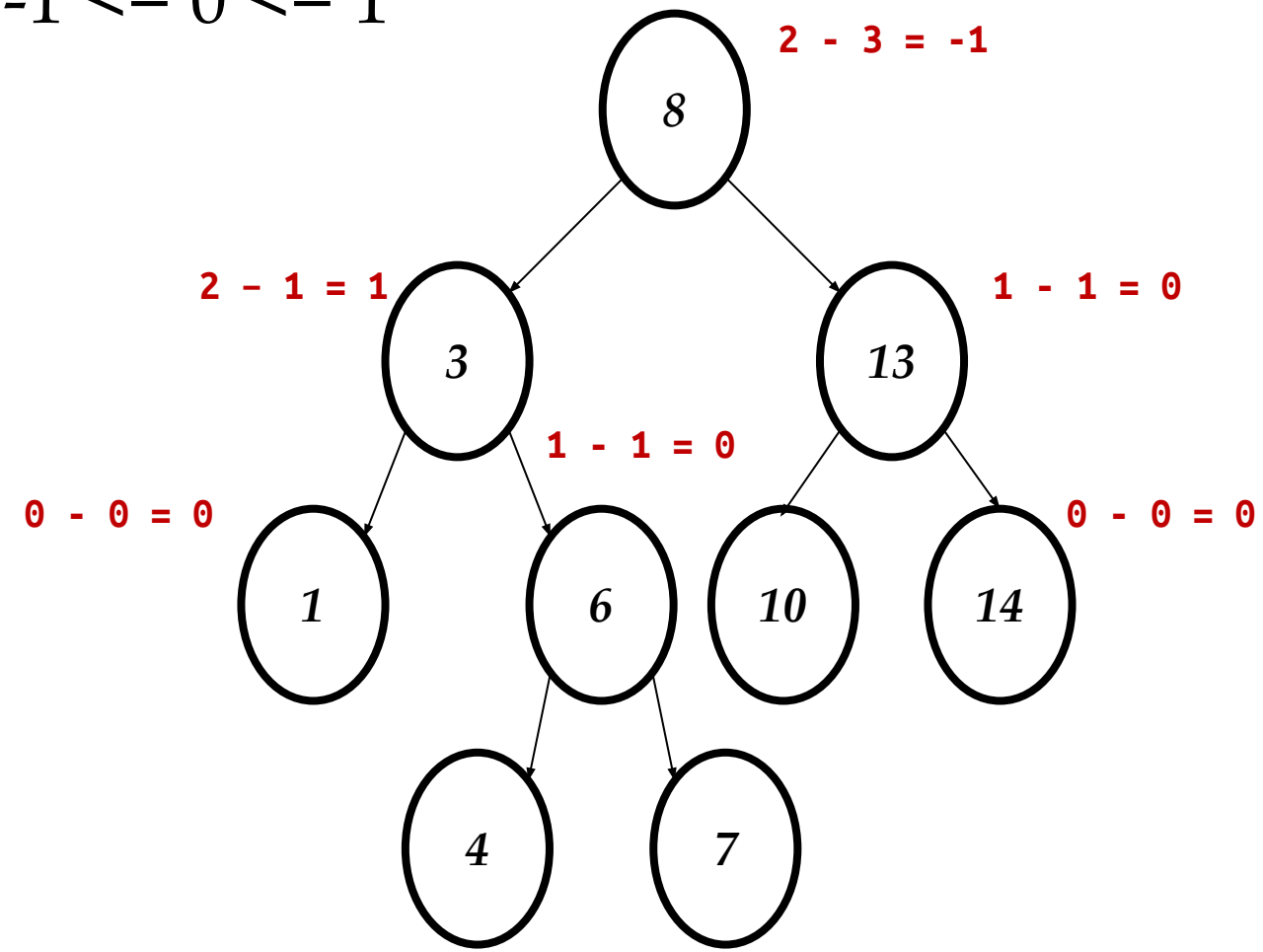
```
int height(node_t *node) {
  if (node == NULL) return 0;
  return 1 + max(height(node->leftChild),
                 height(node->rightChild));
}
```



- Observação:
  - Normalmente conta-se a altura da folha como sendo 0
  - Basta reduzir -1 de todos os casos acima

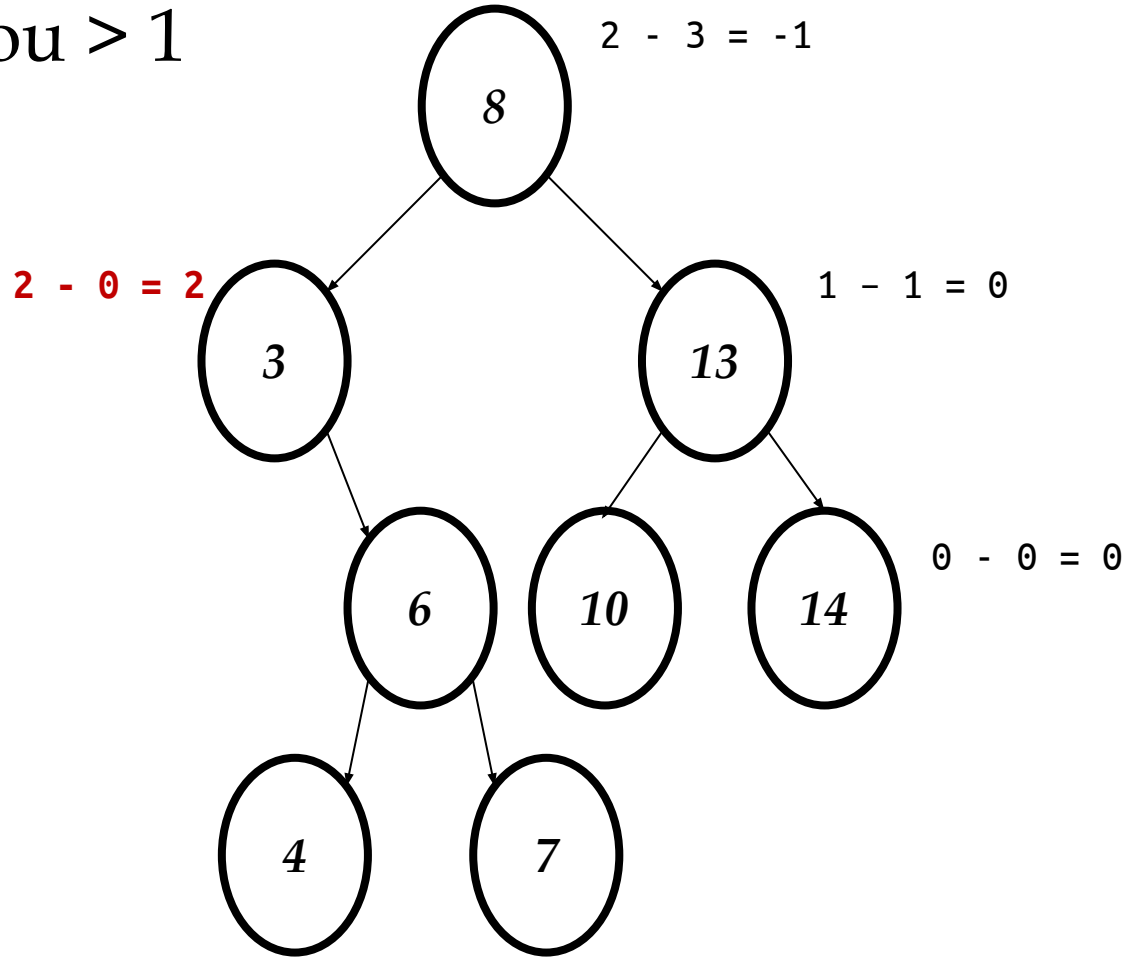
Balanceamento

- Se for -1 <= 0 <= 1

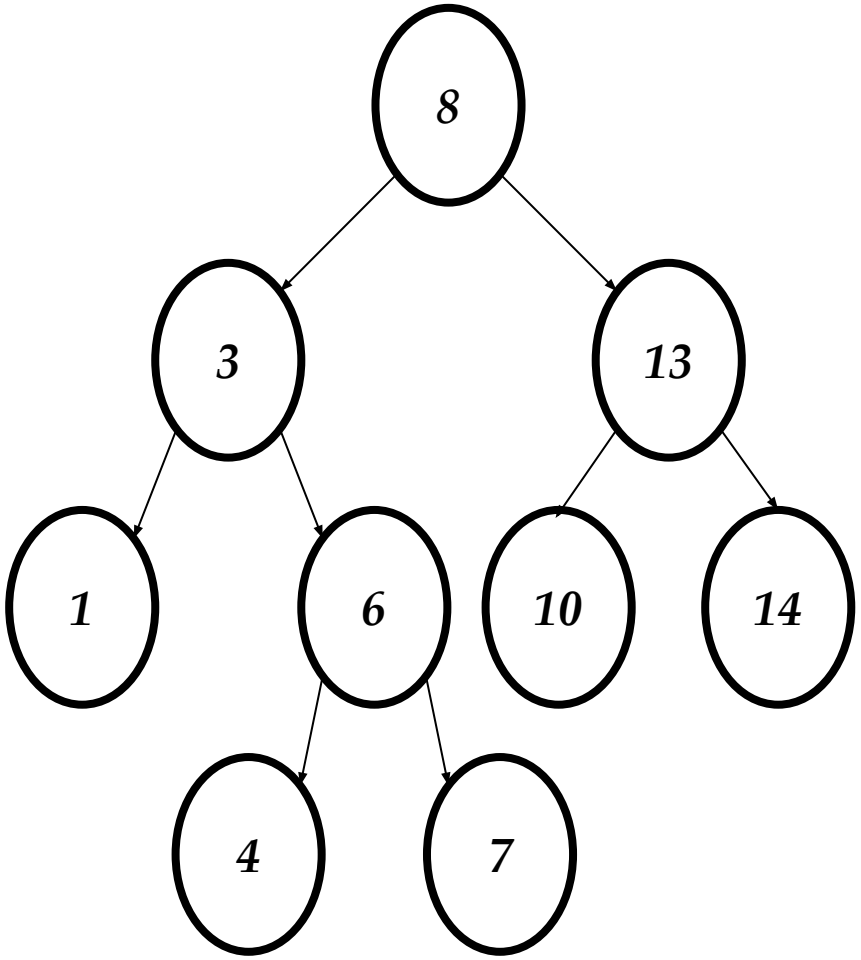


Desbalanceamento

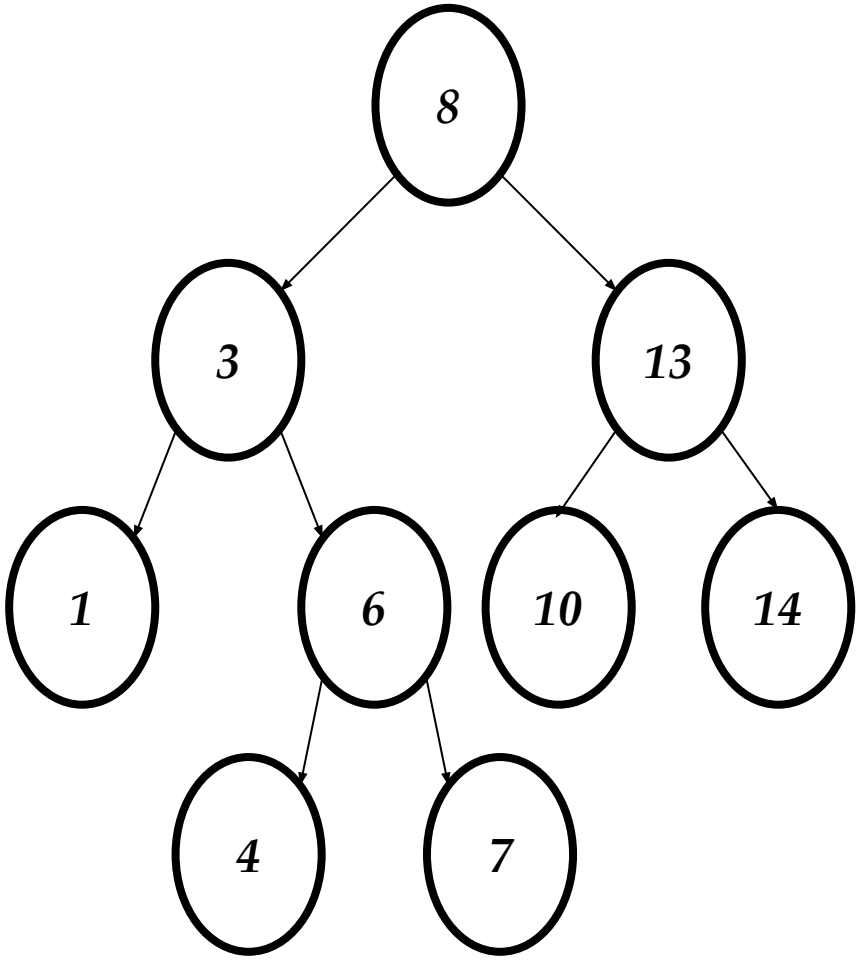
- Se for <-1 ou > 1



```c
int max(int v1, int v2) {
  if (v1 > v2) return v1;
  return v2;
}

int height(node_t *node) {
  if (node == NULL) return 0;
  return 1 + max(height(node->leftChild),
                 height(node->rightChild));
}

int balanceFactor(node_t *node) {
  if (node == NULL) return 0;

  int balance = height(node->rightChild) -
                height(node->leftChild);
  return balance;
}
```



```c
int max(int v1, int v2) {
  if (v1 > v2) return v1;
  return v2;
}

int height(node_t *node) {
  if (node == NULL) return 0;
  return 1 + max(height(node->leftChild),
                 height(node->rightChild));
}

int balanceFactor(node_t *node) {
  if (node == NULL) return 0;

  int balance = height(node->rightChild) -
                height(node->leftChild);
  return balance;
}
```

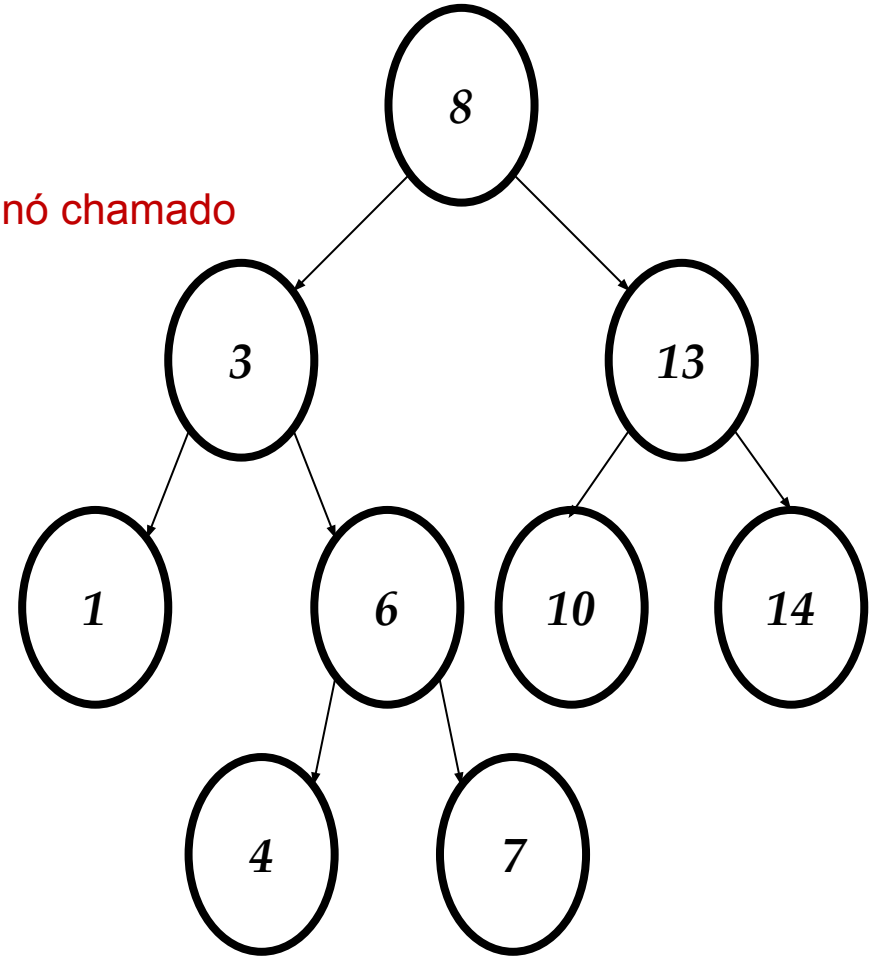


```c
int max(int v1, int v2) {
  if (v1 > v2) return v1;
  return v2;
}
```

Calcula apenas para o nó chamado

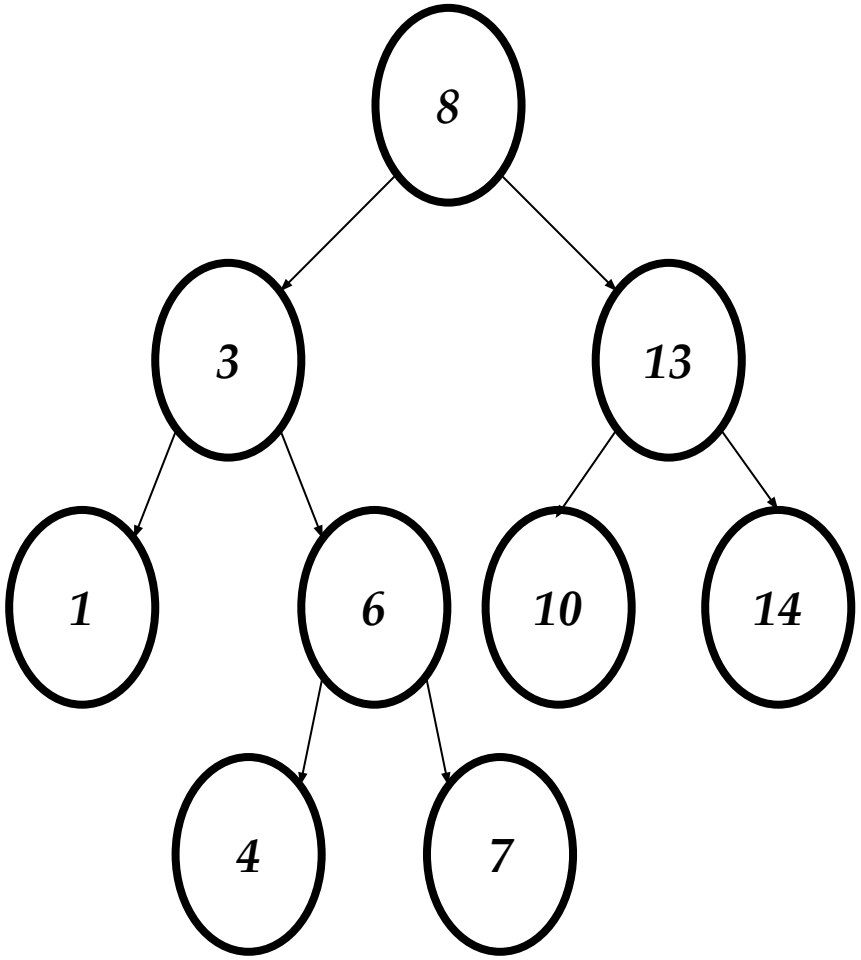
```c
int height(node_t *node) {
  if (node == NULL) return 0;
  return 1 + max(height(node->leftChild),
                 height(node->rightChild));
}
```

```c
int balanceFactor(node_t *node) {
  if (node == NULL) return 0;

  int balance = height(node->rightChild) -
                height(node->leftChild);
  return balance;
}
```



```c
int max(int v1, int v2) {
  if (v1 > v2) return v1;
  return v2;
}

int height(node_t *node) {
  if (node == NULL) return 0;
  return 1 + max(height(node->leftChild),
                 height(node->rightChild));
}
```

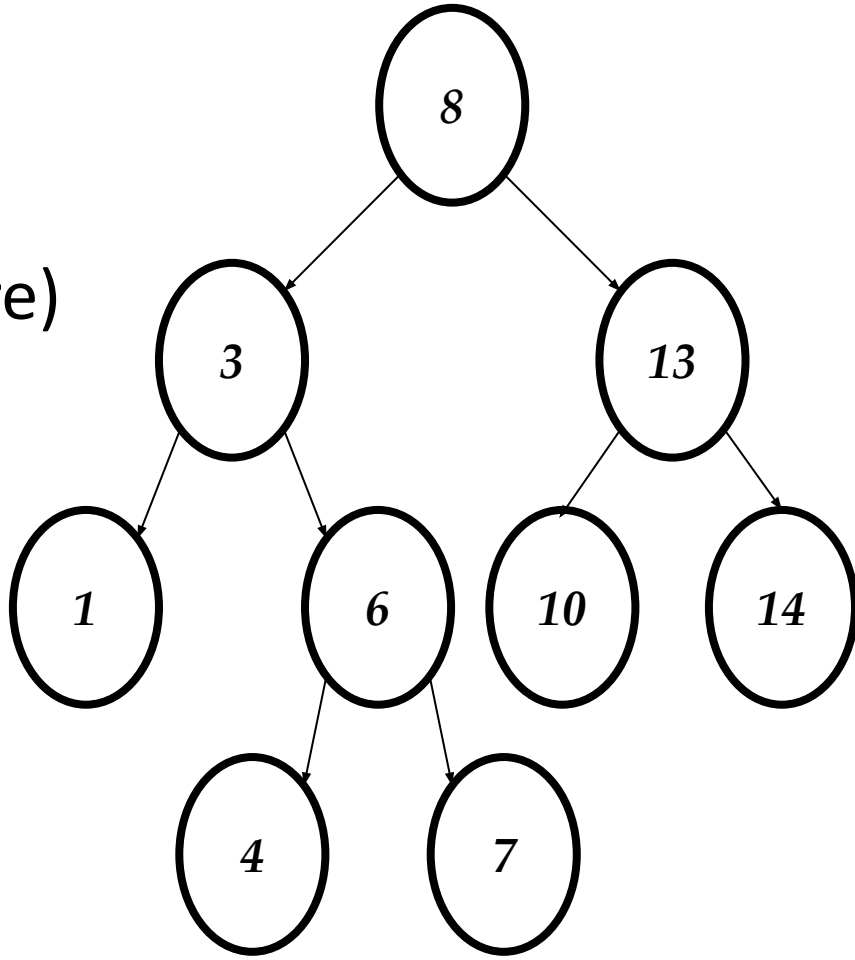
```c
int balanceFactor(node_t *node) {
  if (node == NULL) return 0;

  int balance = height(node->rightChild) -
                height(node->leftChild);
  return balance;
}
```



Fator de balanceamento: Depende da altura das sub-árvores

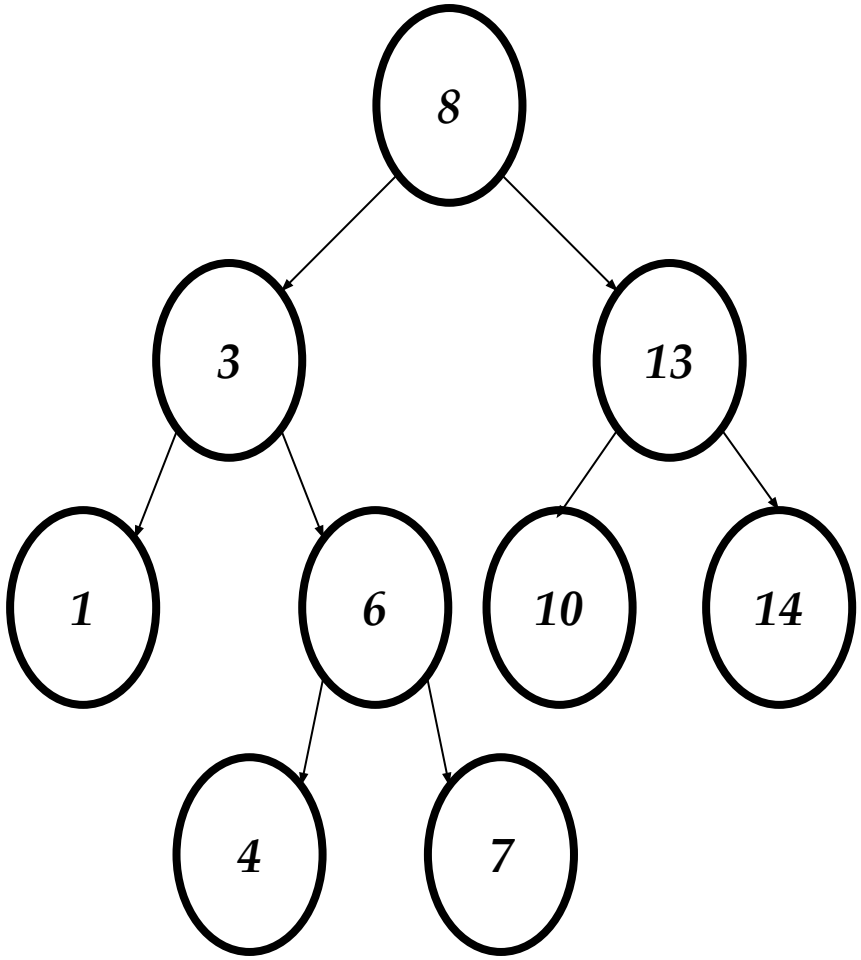
Altura

- Podemos armazenar a altura do nó (e não calculá-la sempre)
- Calculo apenas ao inserir o nó
- Necessário atualizar a altura do antecedentes ao inserir novo nó



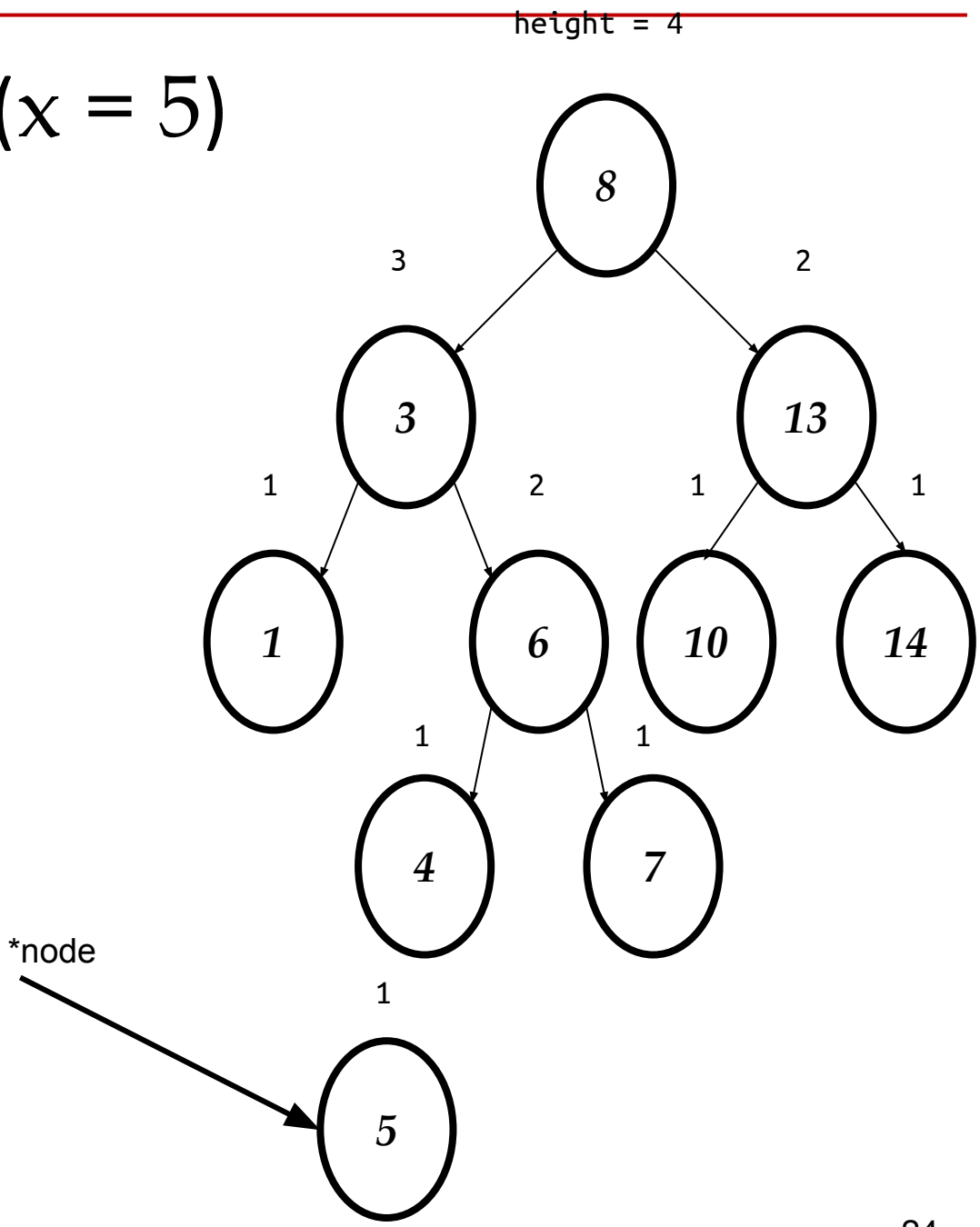
Pre-Computando Alturas

```
typedef struct node {
  int value;
  int height;
  struct node *leftChild;
  struct node *rightChild;
  struct node *parent;
} node_t;
```



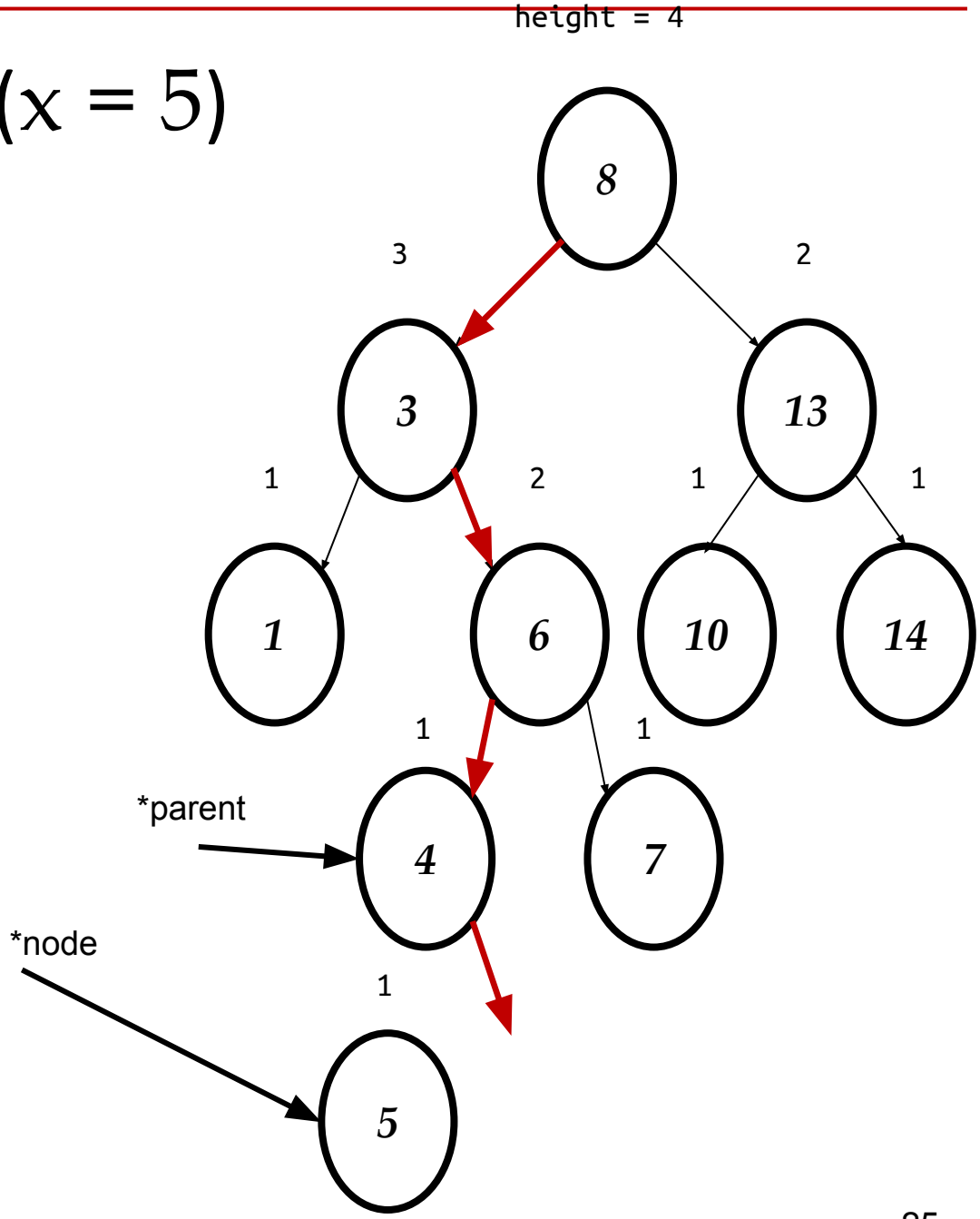
Altura ao Inserir (x = 5)

- Inicializa o struct
  - `value = 5;`
  - `leftChild = NULL;`
  - `rightChild = NULL;`
  - `height = 1;`



Altura ao Inserir (x = 5)

- Acha a posição dele na árvore

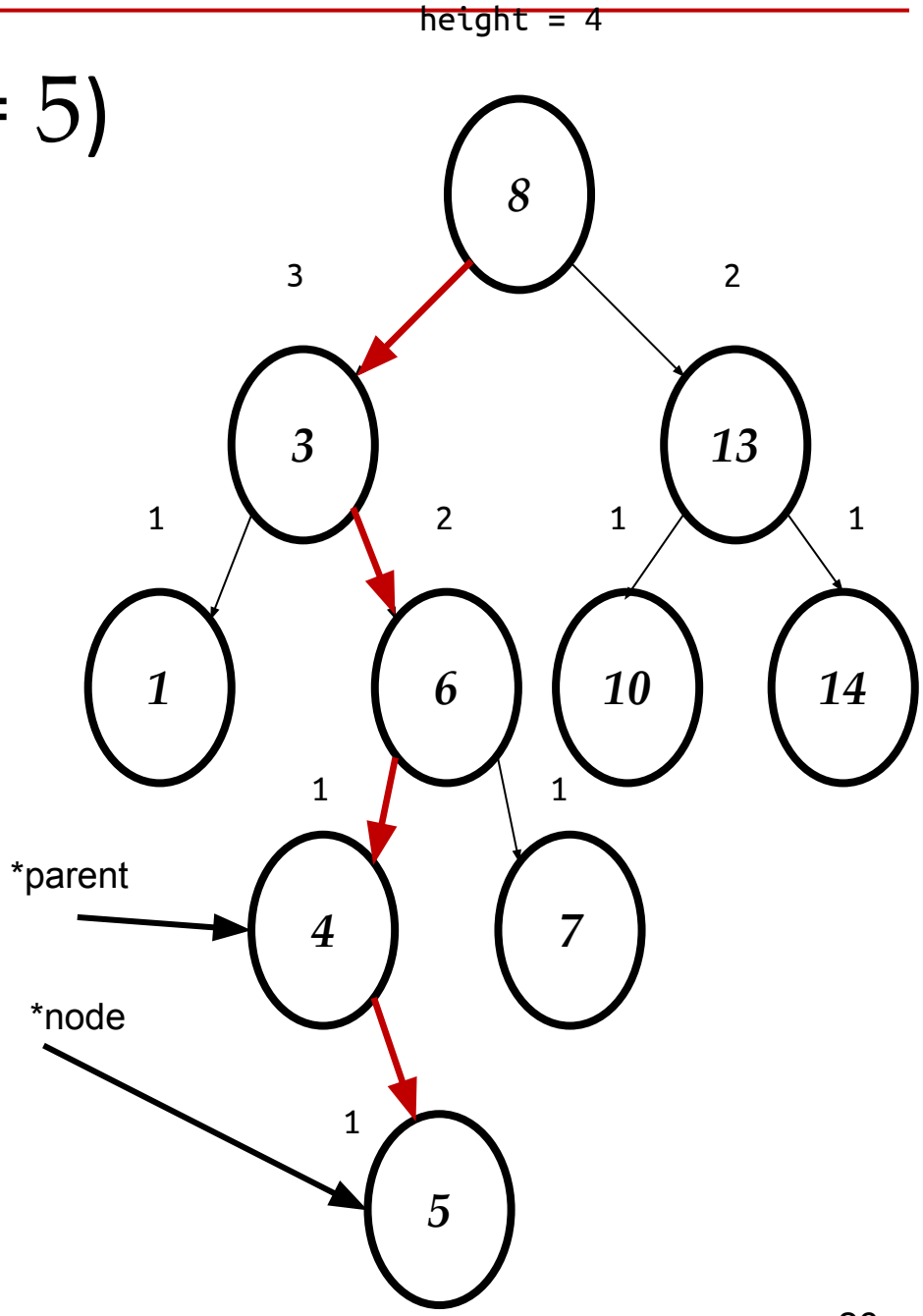


Altura ao Inserir (x = 5)

- Insere ele na posição

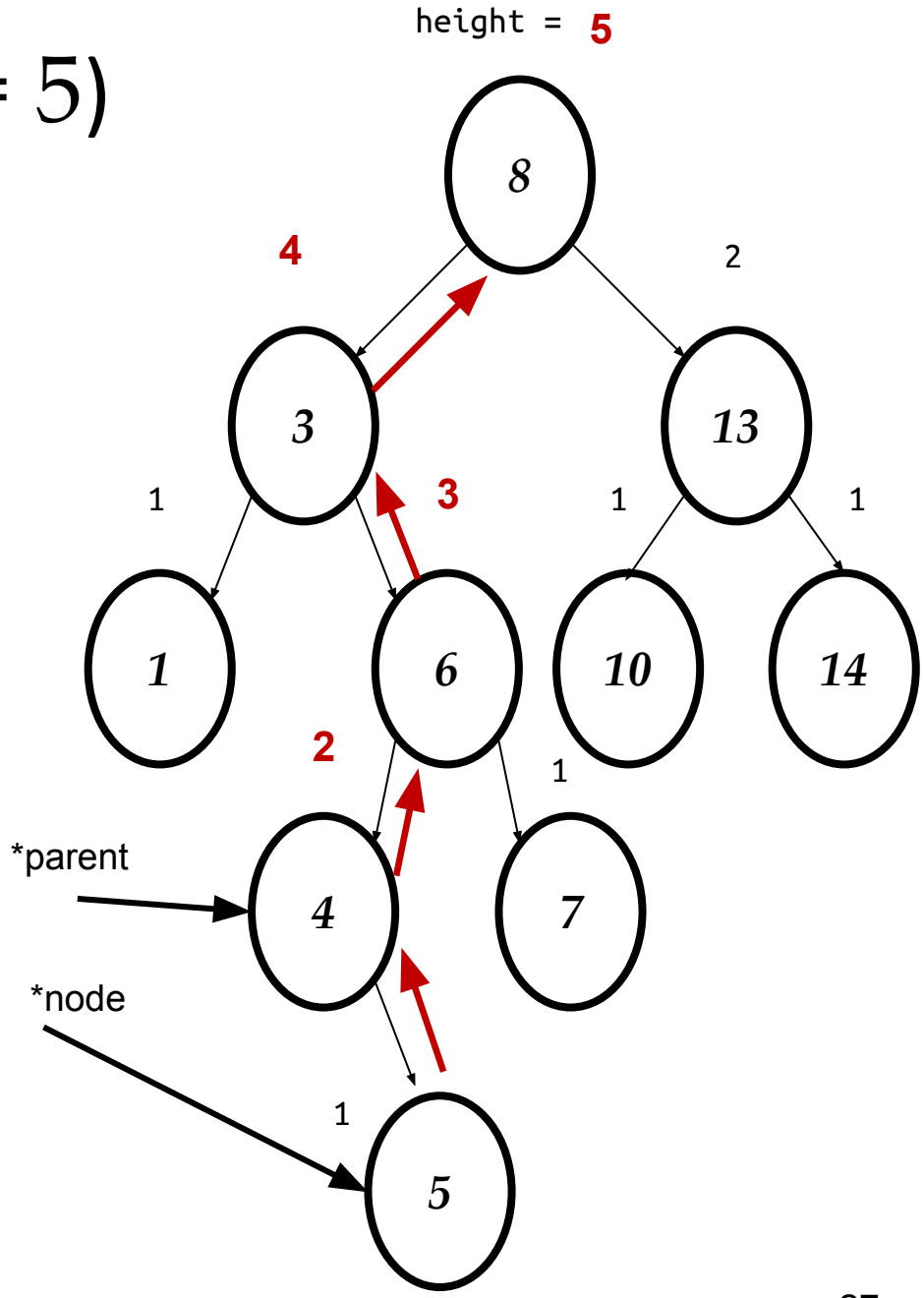
Nó:

```
value = 5;
leftChild = NULL;
rightChild = NULL;
height = 1;
node->parent = nó 4;
```



Altura ao Inserir (x = 5)

- Caminhar para cima (até a raiz), atualizando os antecedentes
- Quando atualizar:
  - Quando pai não tem nenhum filho
  - Quando altura do pai = altura do filho



Sabendo Computar Altura

Balanceamento é fácil

```
int balanceFactor(node_t *node) {
  if (node == NULL) return 0;
  int balance = node->rightChild->height - node->leftChild->height;
  return balance;
}
```

Até Agora

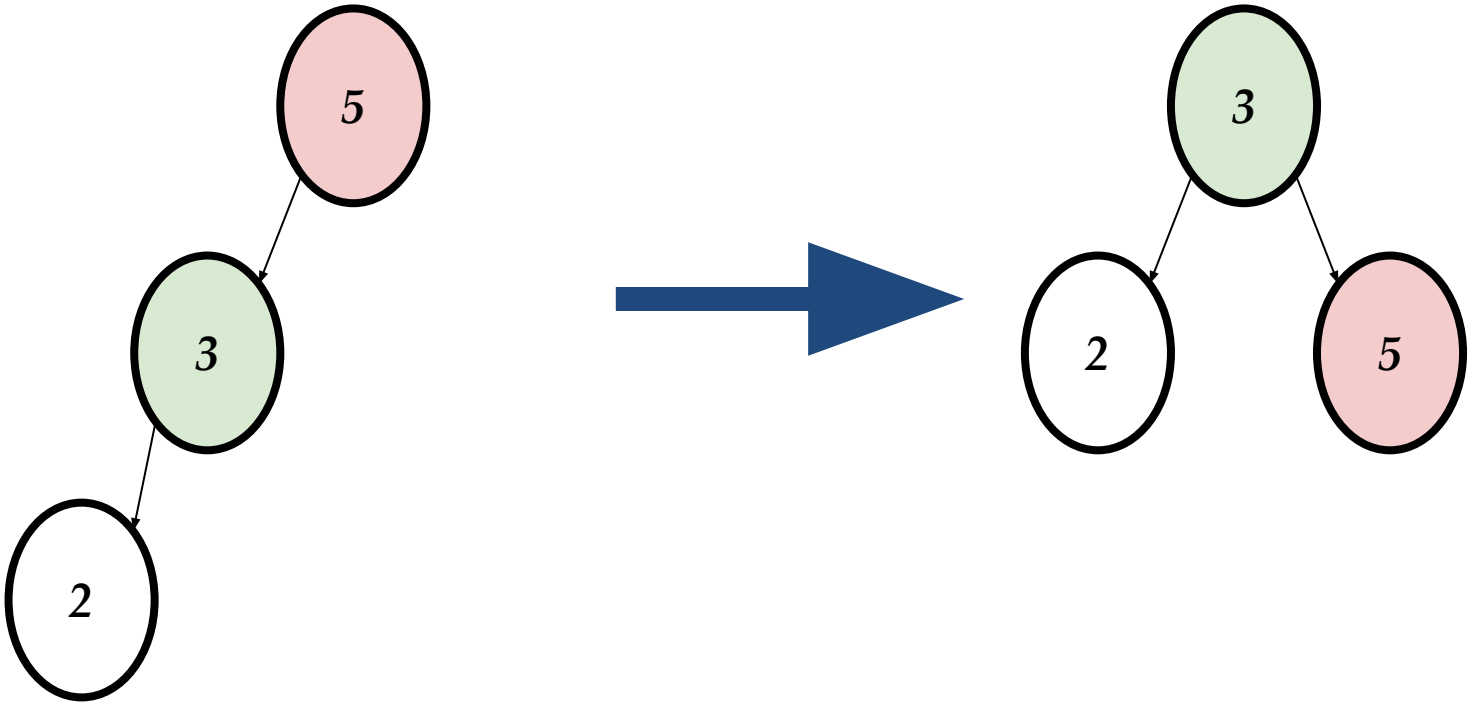
- Sabemos computar a altura de um nó
- Sabemos manter a altura computada ao inserir
- Sabemos computar o balanceamento de um nó

ROTAÇÕES

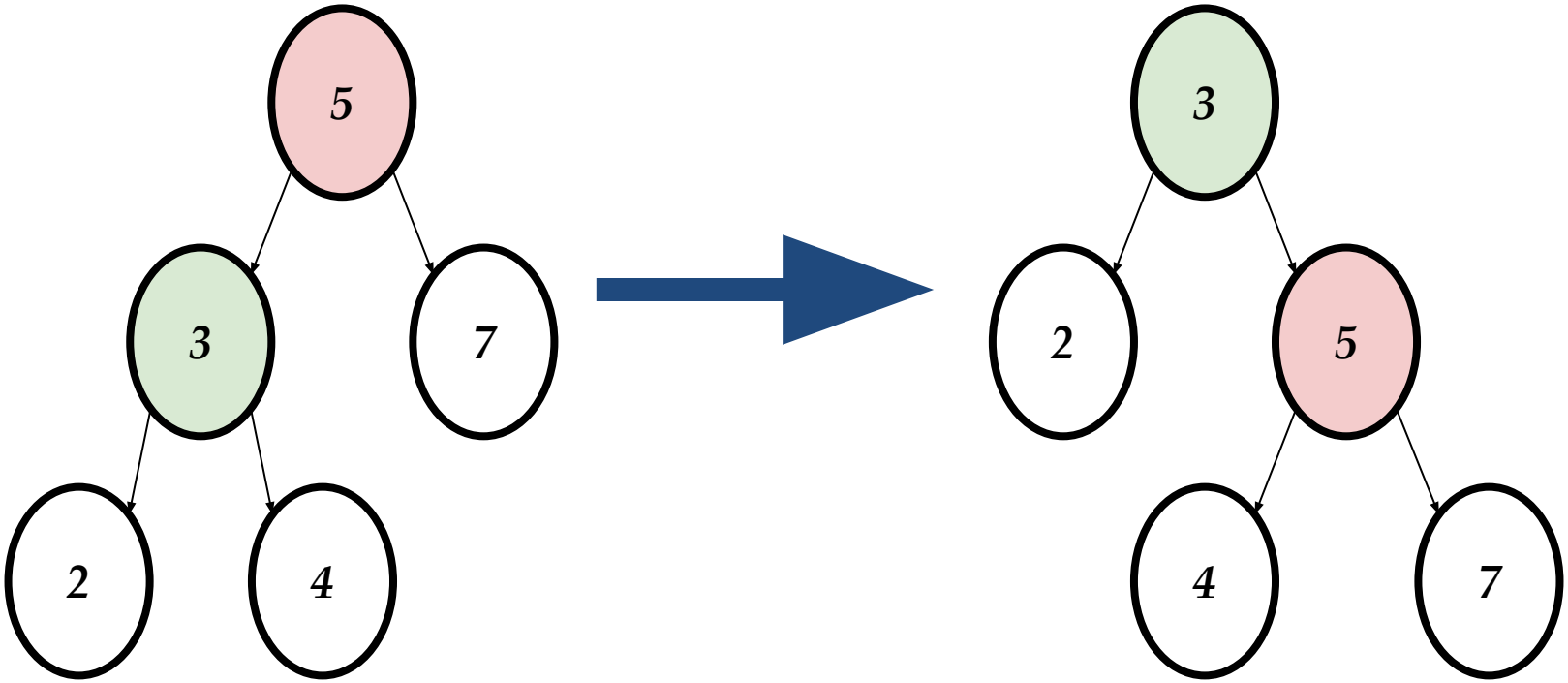
Rotações

- Rotação é uma operação utilizada na árvore AVL para manter o balanceamento

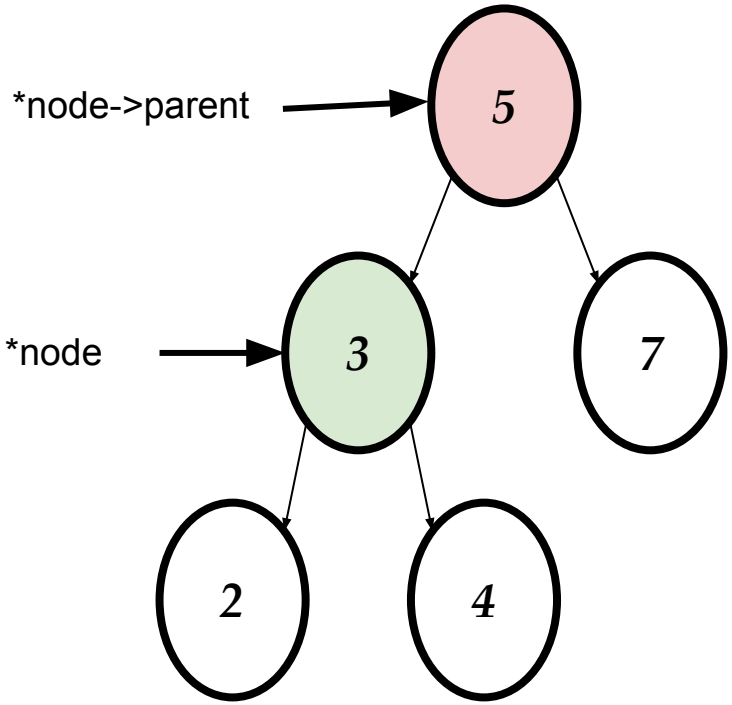
Rotação para Direita



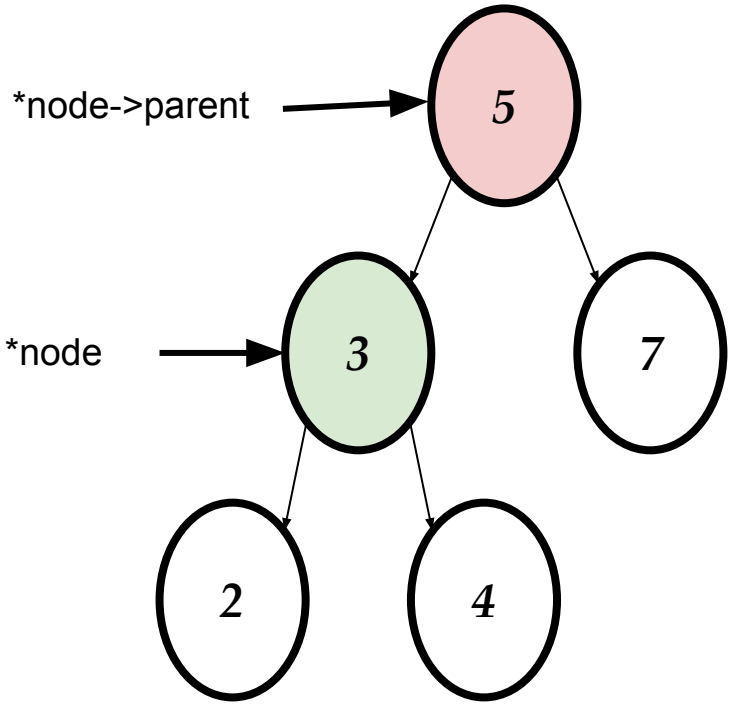
Resultado



Rotação para Direita

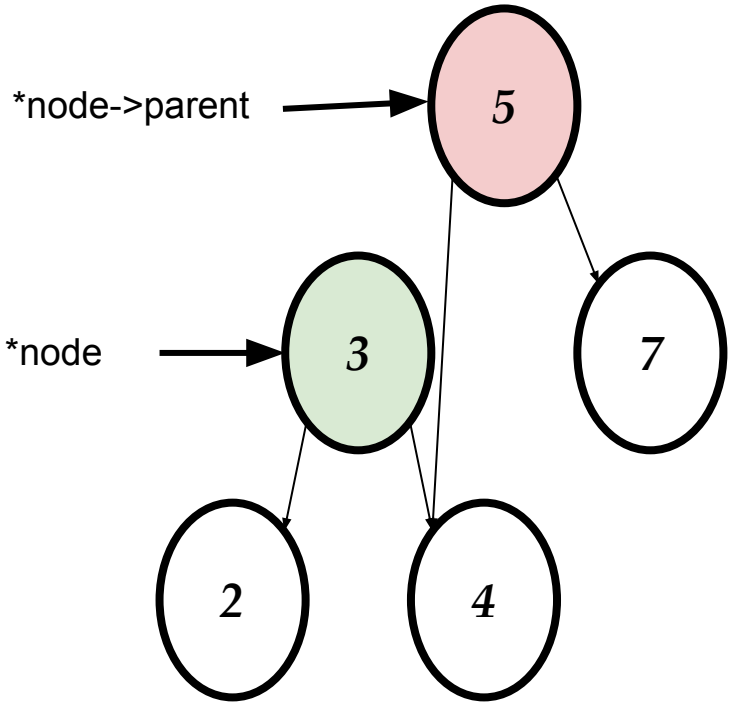


Rotação para Direita



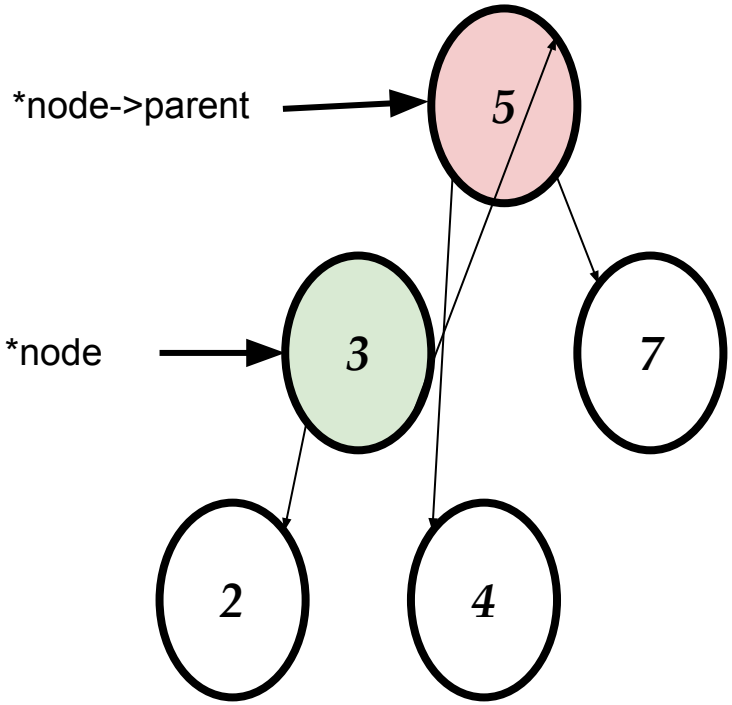
Como 5 é maior do que 3
tem que ir para direita

Rotação para Direita



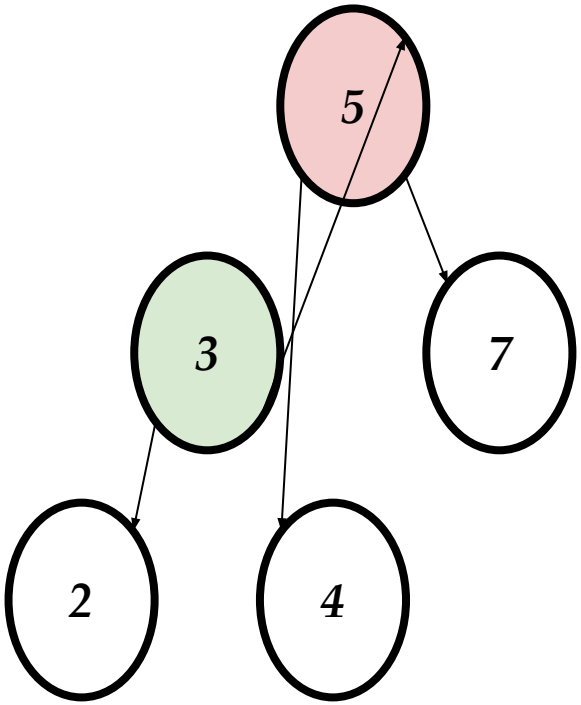

```
node->parent->leftChild = node->rightChild;
```

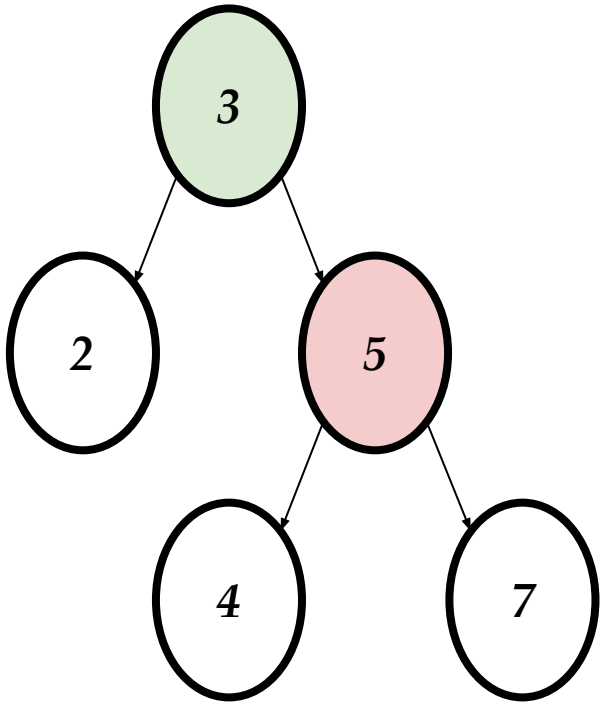
Rotação para Direita



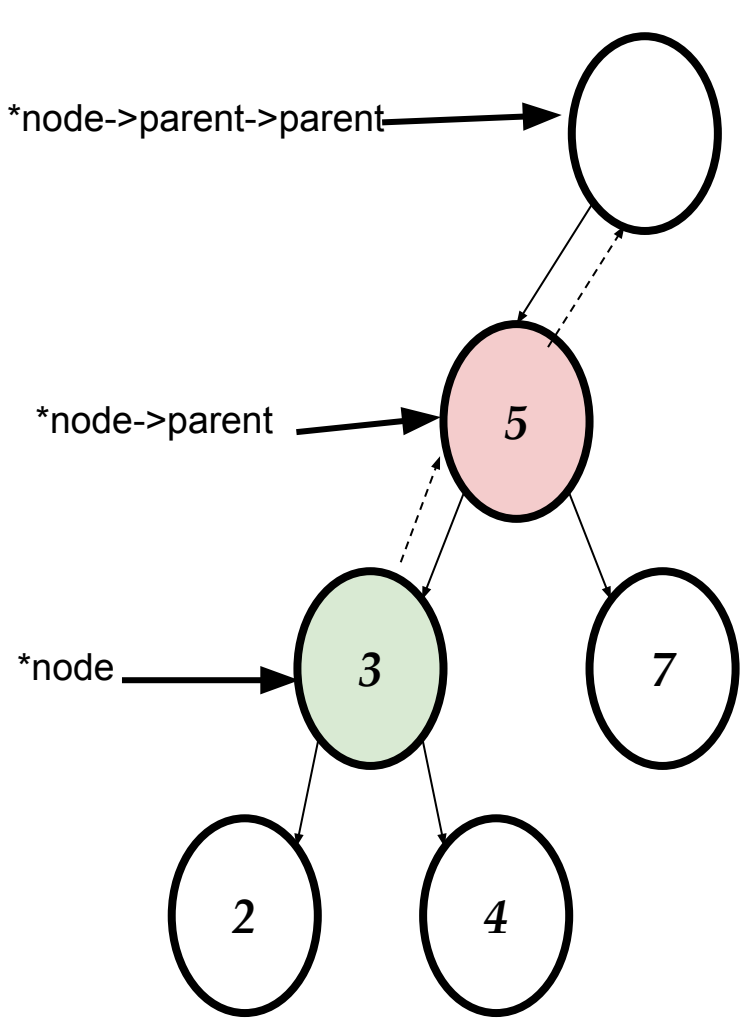

```
node->parent->leftChild = node->rightChild;

node->rightChild = node->parent;
```

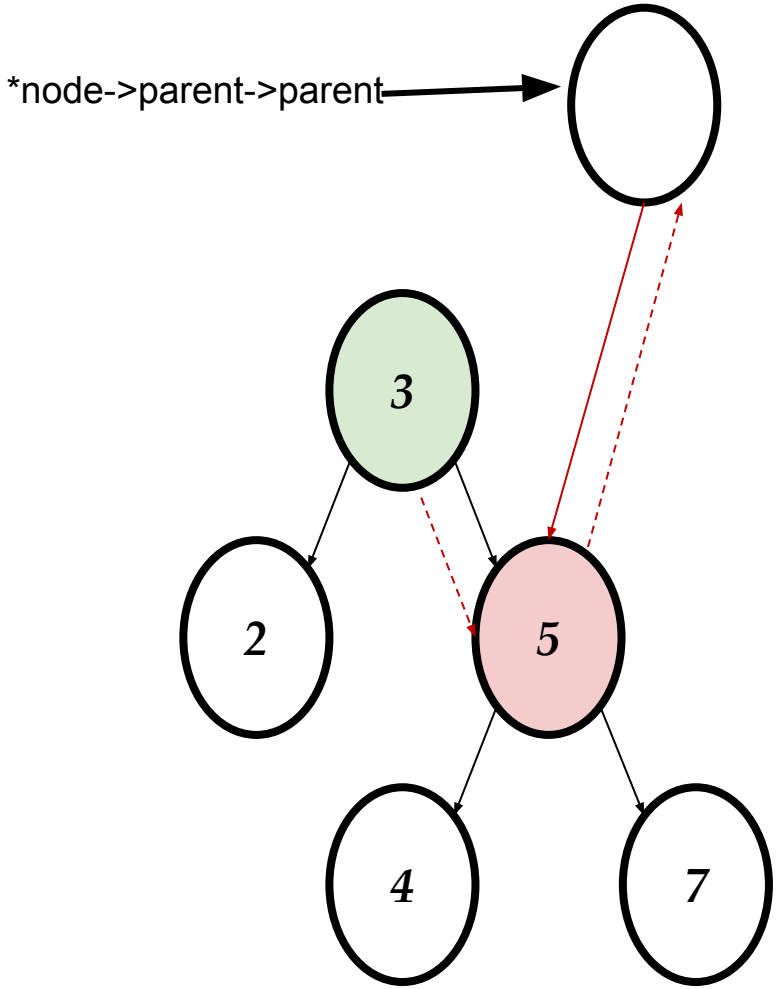
Resultado é este, só redesenhar



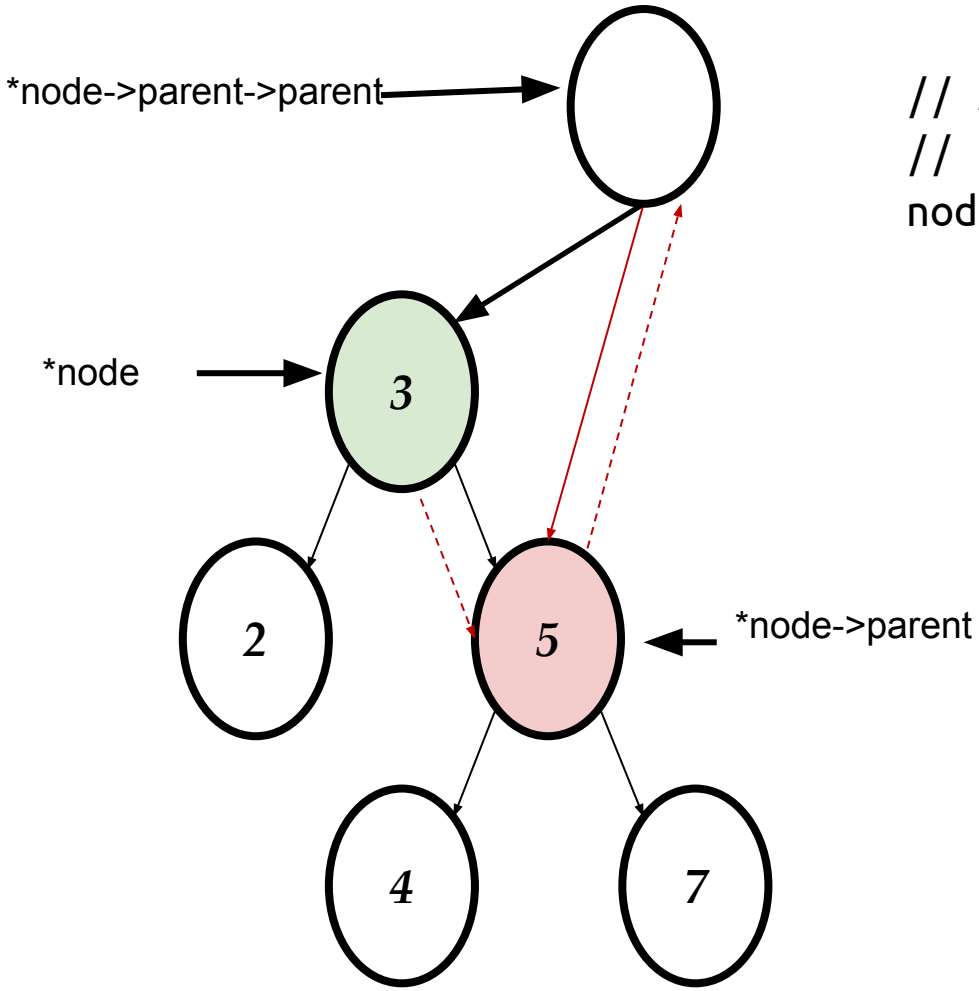


[Caso exista] Atualizar ponteiros do parent



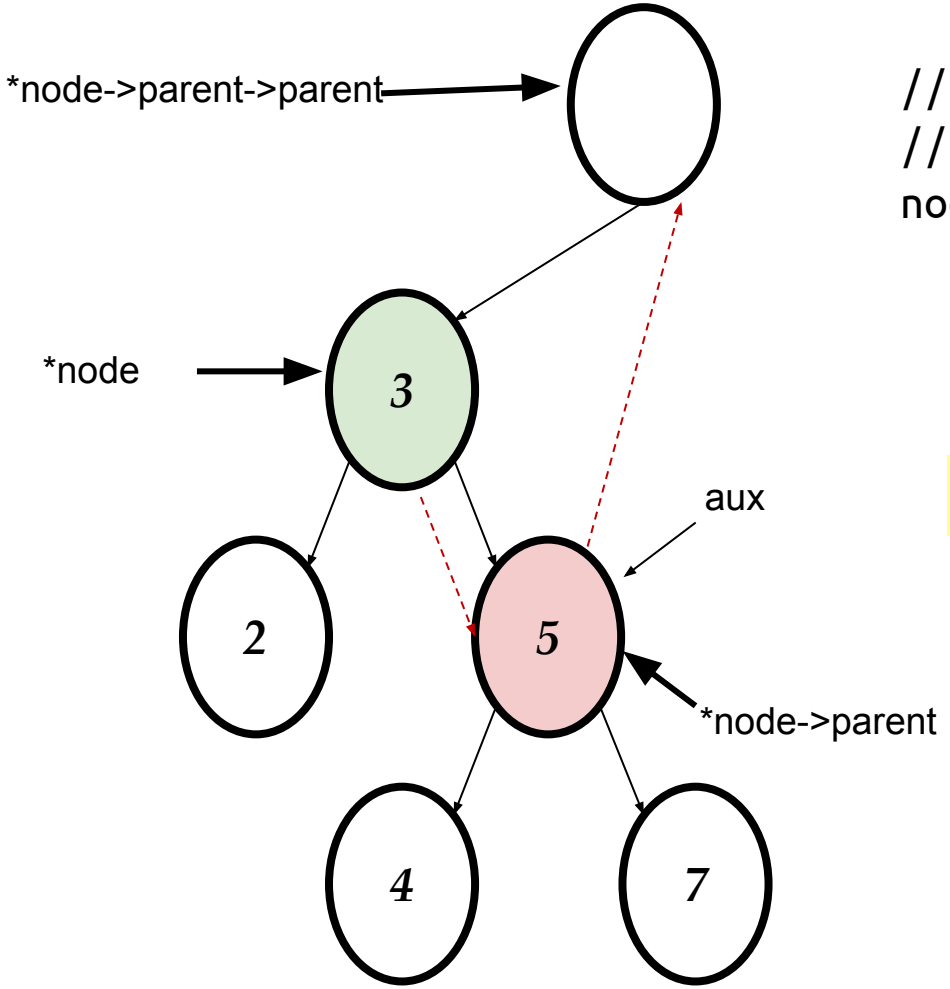


[Caso exista] Atualizar ponteiros do parent



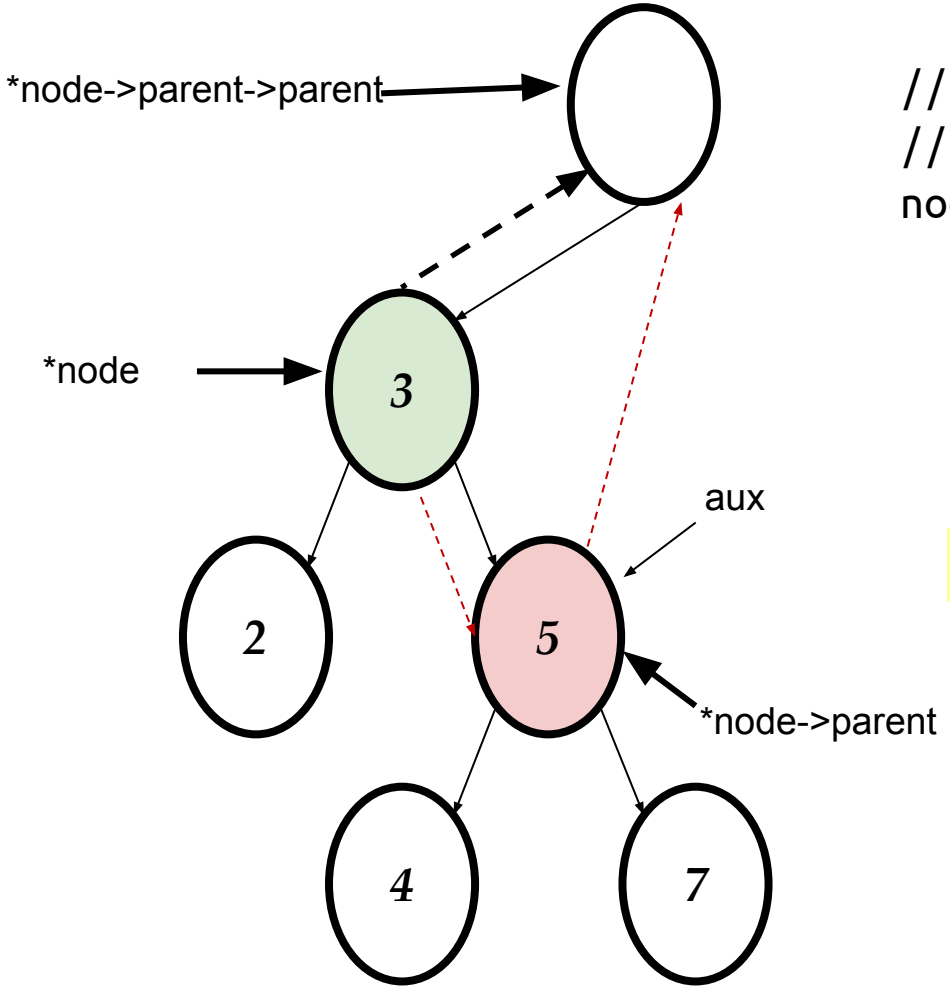

```
// Assumindo que estamos na
// esquerda (teríamos que identificar)
node->parent->parent->leftChild = node
```

[Caso exista] Atualizar ponteiros do parent



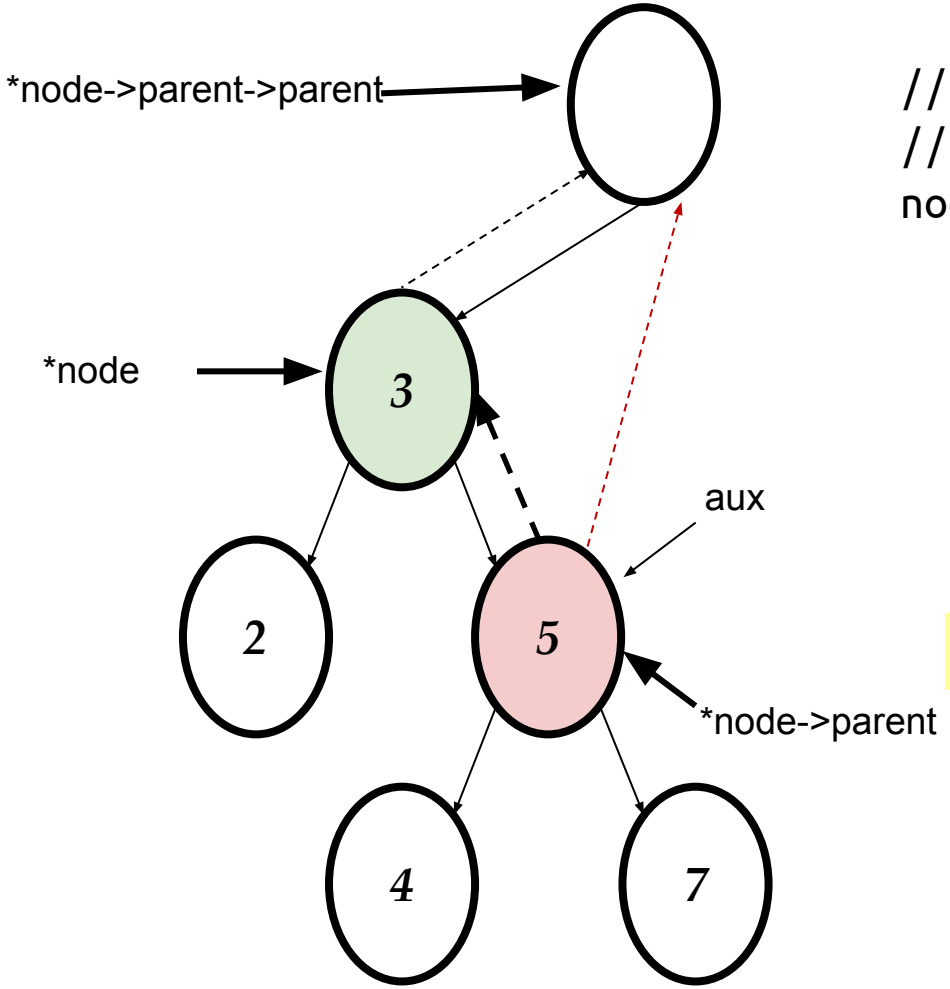

```
// Assumindo que estamos na
// esquerda (teríamos que identificar)
node->parent->parent->leftChild = node

  // Caso use ponteiro para cima
  aux = node->parent;
  node->parent = node->parent->parent;
  aux->parent = node;

   // Parent do cinco vira o 3
   aux->parent = node;
```

[Caso exista] Atualizar ponteiros do parent



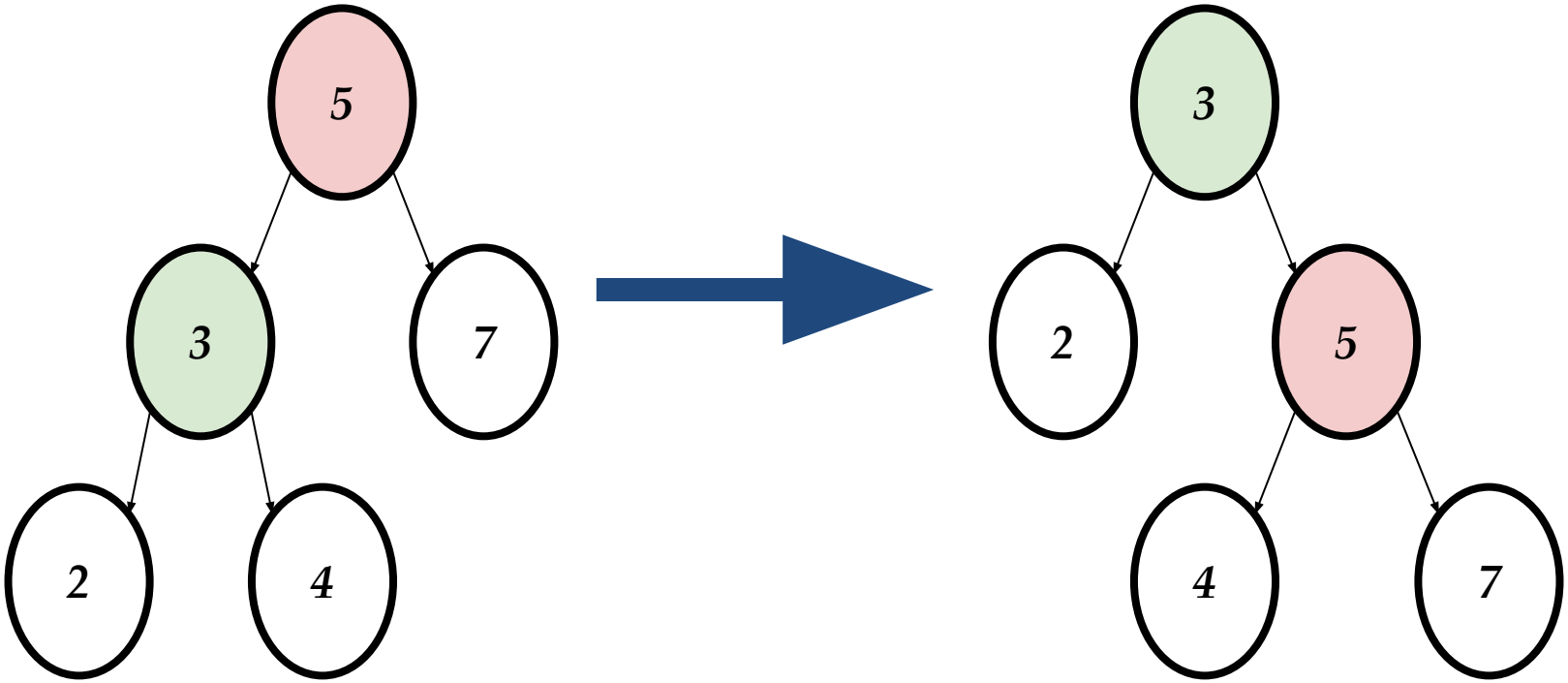

```
// Assumindo que estamos na
// esquerda (teríamos que identificar)
node->parent->parent->leftChild = node

  // Caso use ponteiro para cima
  aux = node->parent;
  node->parent = node->parent->parent;
  aux->parent = node;

  // Parent do cinco vira o 3
  aux->parent = node;
```

[Caso exista] Atualizar ponteiros do parent

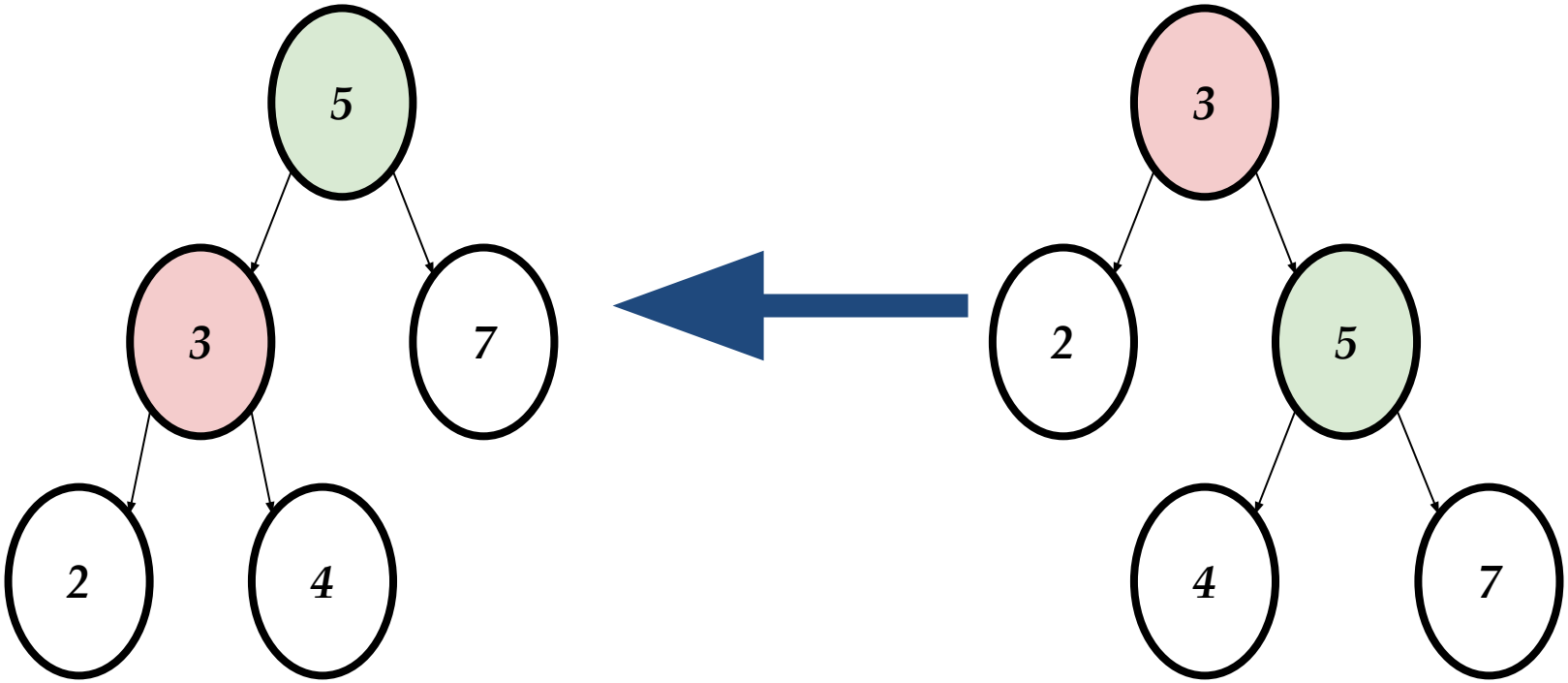



```
// Assumindo que estamos na
// esquerda (teríamos que identificar)
node->parent->parent->leftChild = node

  // Caso use ponteiro para cima
  aux = node->parent;
  node->parent = node->parent->parent;
  aux->parent = node;

  // Parent do cinco vira o 3
  aux->parent = node;
```

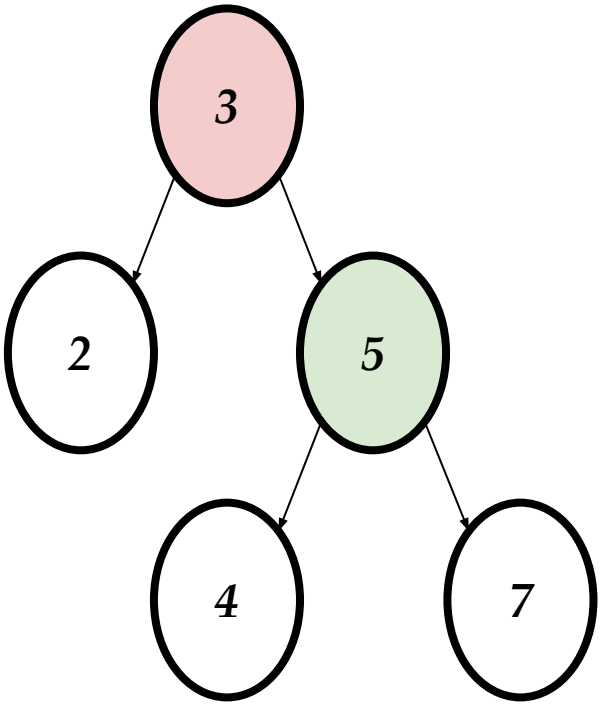
Resultado da Rotação para Direita



Rotação para Esquerda

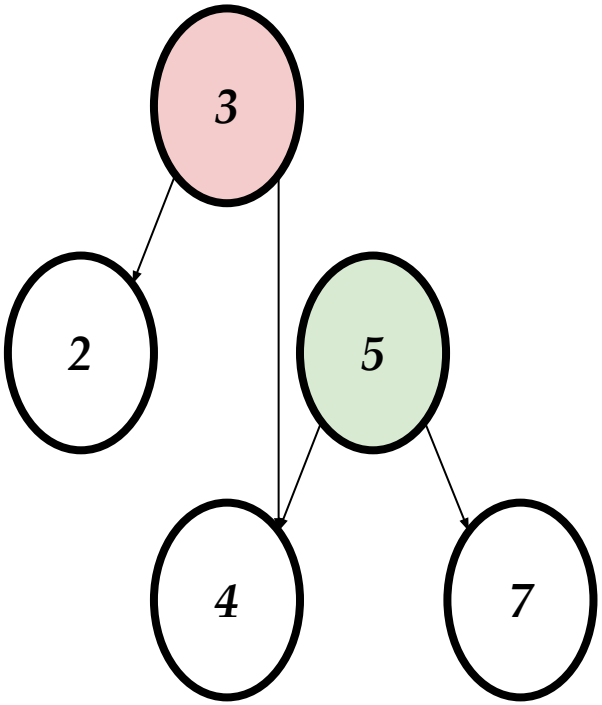


Mesma Ideia



Atualiza filho da direita do parent

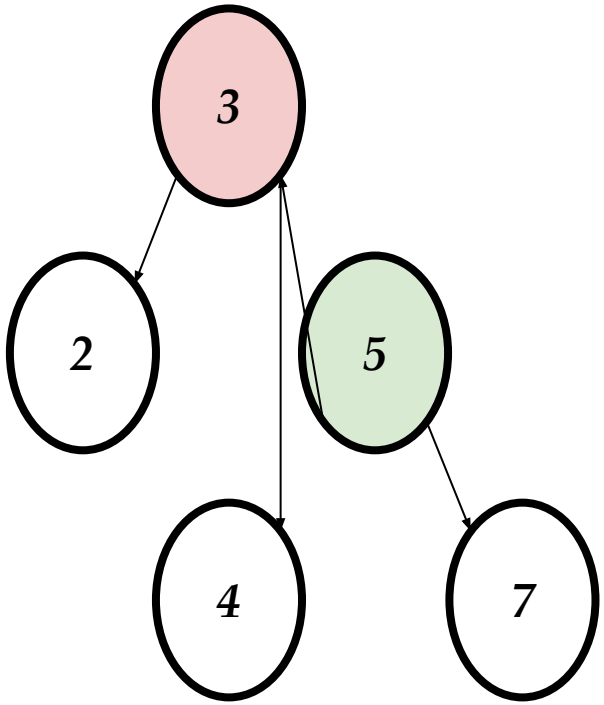
```
node->parent->rightChild = node->leftChild;
```



Atualiza filho da direita do parent

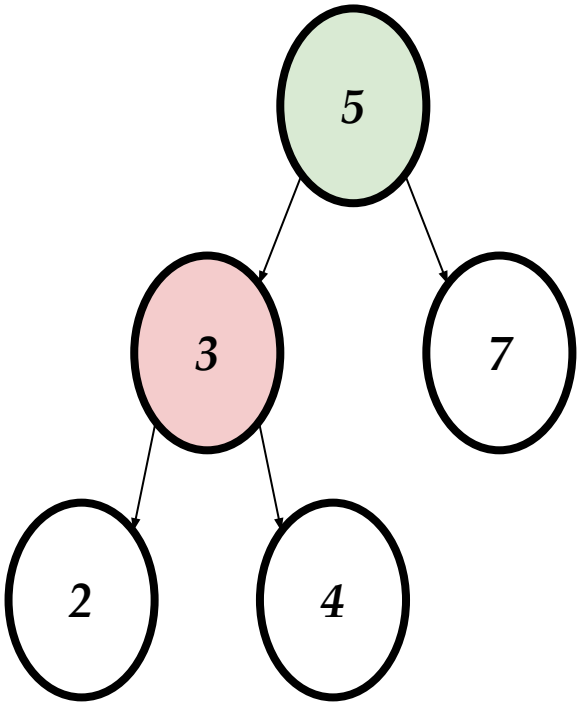
```
node->parent->rightChild = node->leftChild;
```

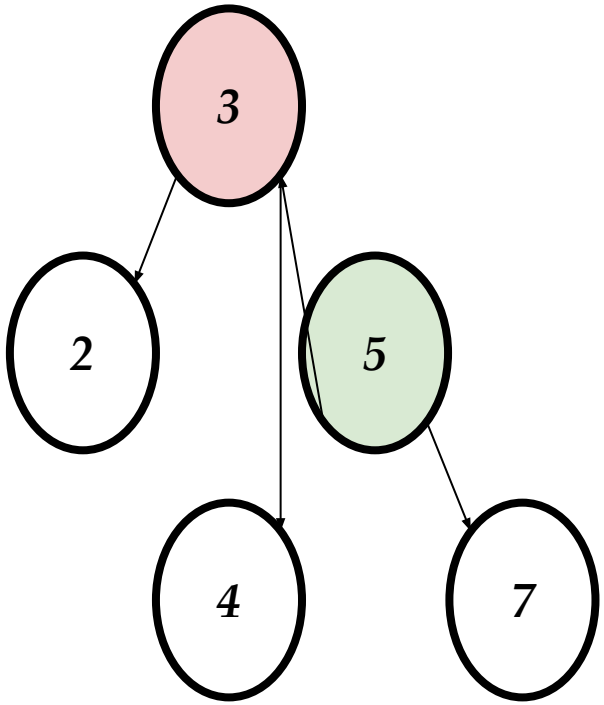
```
node->leftChild = node->parent;
```



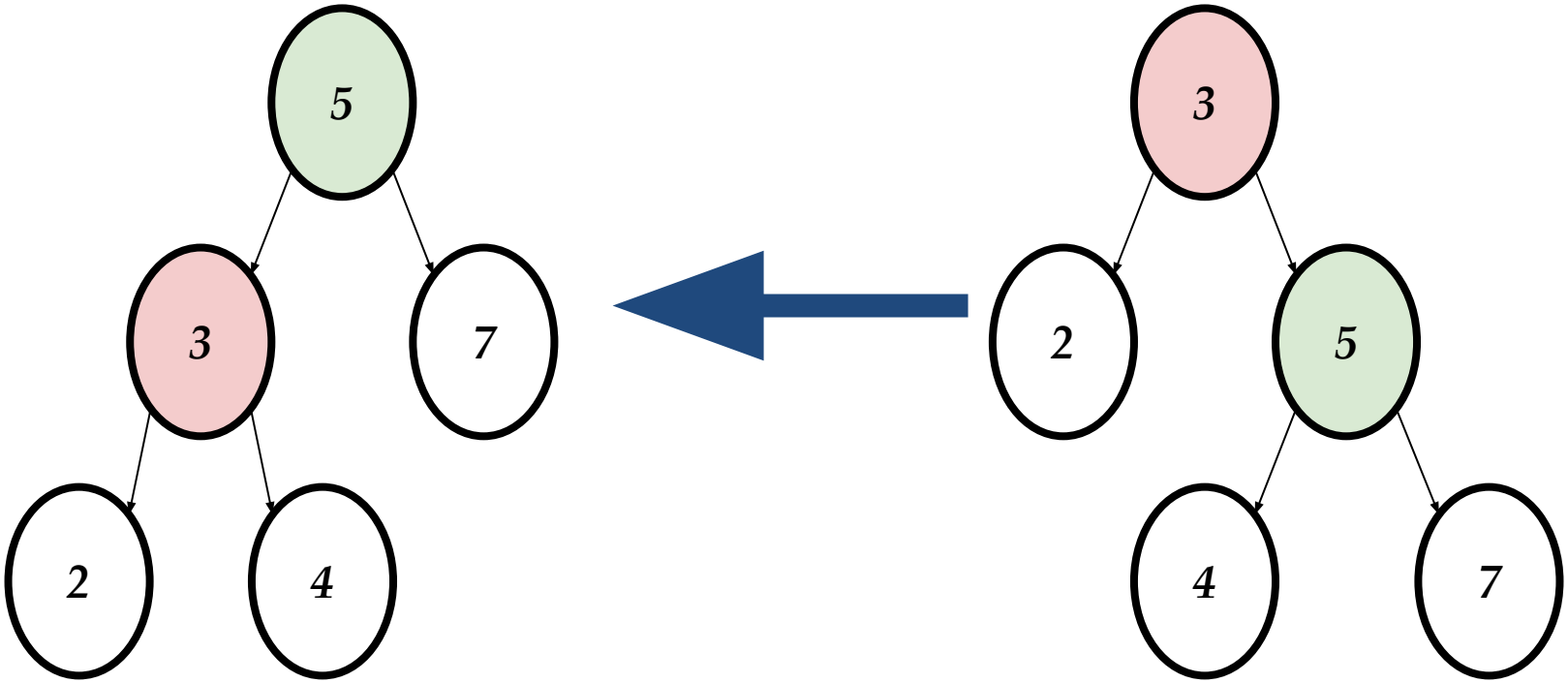
Resultado é este, só redesenhar

(assumindo que atualizamos os parents)





Resultado da Rotação para Esquerda



Outros Cuidados

- Se em alguma rotação o parent for o root
- Atualizar o root também

Custo de Rotações

- O(1)
- Estamos apenas definindo valores de ponteiros

AVL – MANTENDO O BALANCEAMENTO

Árvore AVL

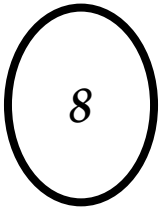
- Sempre mantém a árvore balanceada
- Faz uso do fator de balanceamento e de rotações

Até Agora

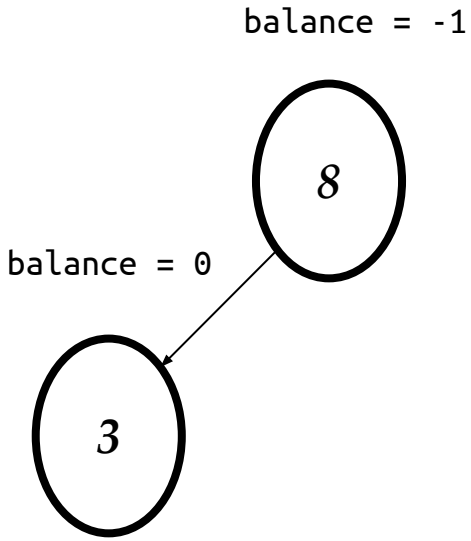
- Sabemos computar a altura de um nó
- Sabemos manter a altura computada ao inserir
- Sabemos computar o balanceamento de um nó
- Sabemos rotacionar

Inserção

balance = 0

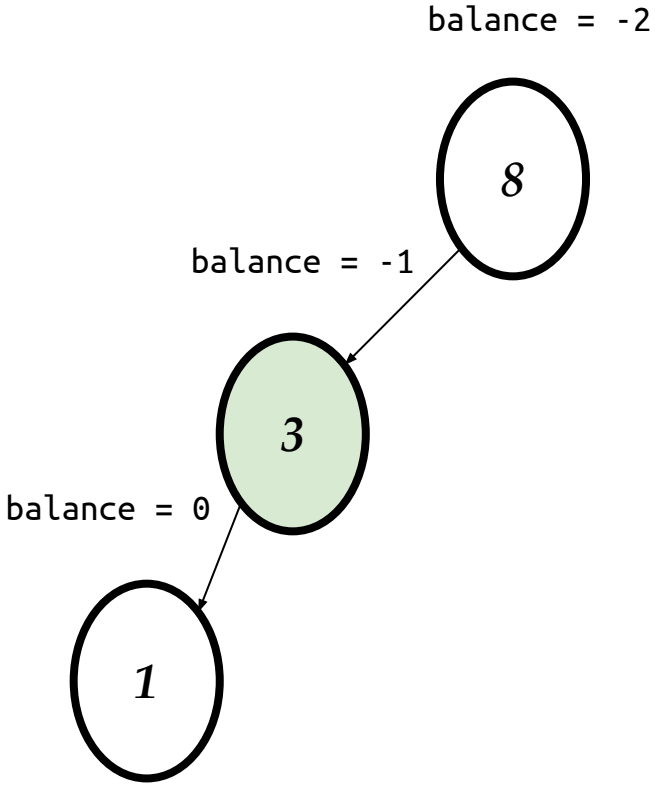


Inserção



Inserção

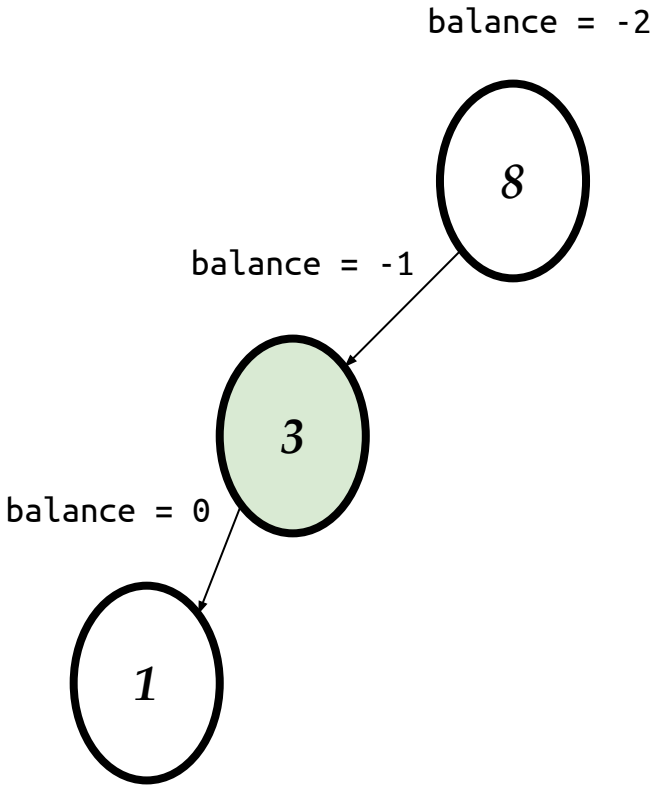
- Sempre inserimos embaixo de um nó folha



Inserção

- Sempre inserimos embaixo de um nó folha

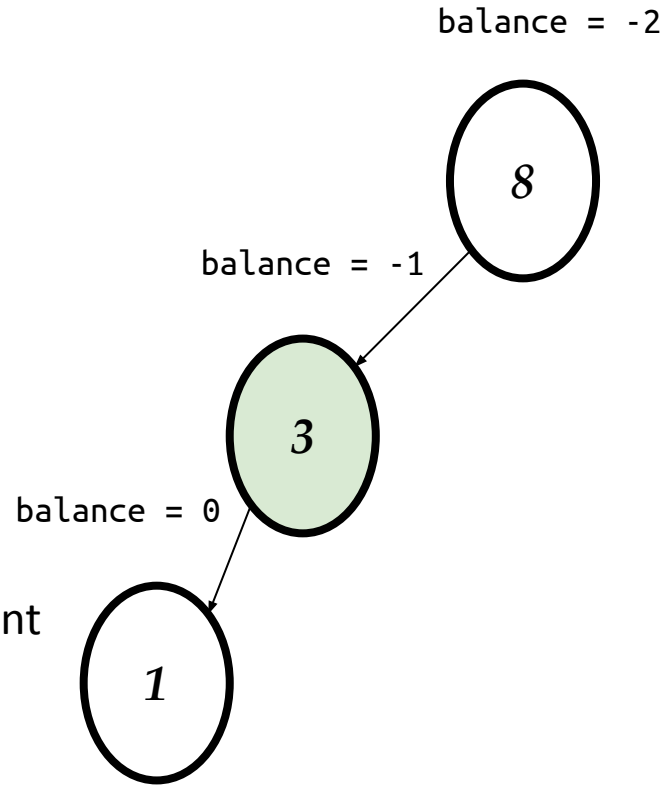
- Por definição:
  - O fator de balanceamento do parent vai ficar entre [-1, 1]



Inserção

- Sempre inserimos embaixo de um nó folha

- Por definição:
  - O fator de balanceamento do parent vai ficar entre [-1, 1]
    - Se era 0 (sem filhos), vira 1 ou -1
    - Se era 1 ou -1 (com 1 filho) vira 0

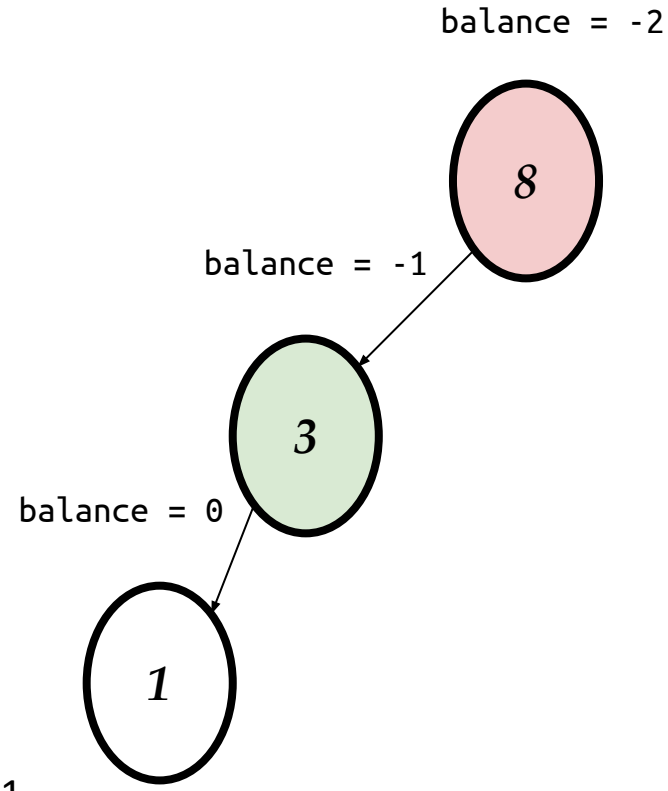


Inserção

- Sempre inserimos embaixo de um nó folha

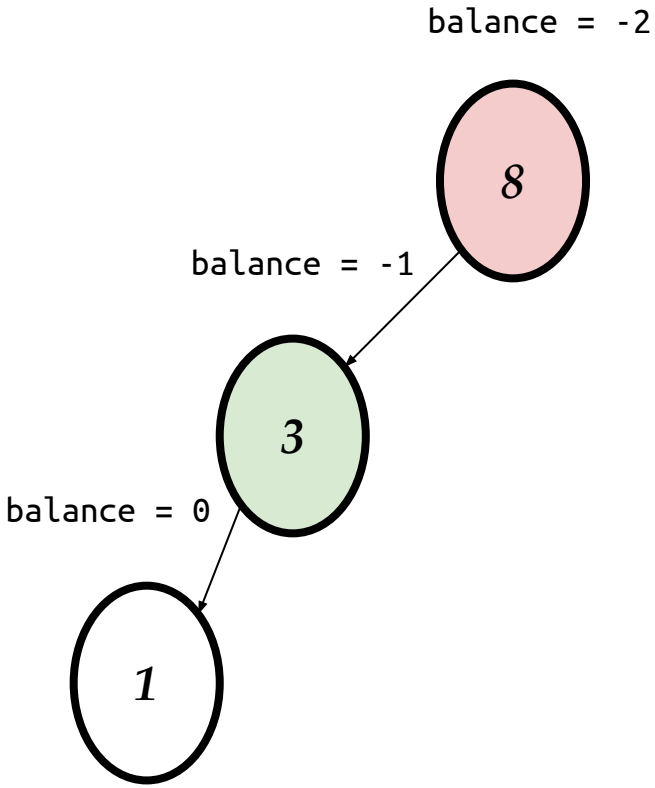
- Por definição:
  - O fator de balanceamento do parent
    vai ficar entre [-1, 1]
    - Se era 0 (sem filhos), vira 1 ou -1
    - Se era 1 ou -1 (com 1 filho) vira 0

- O "avô", por consequência pode desbalancear



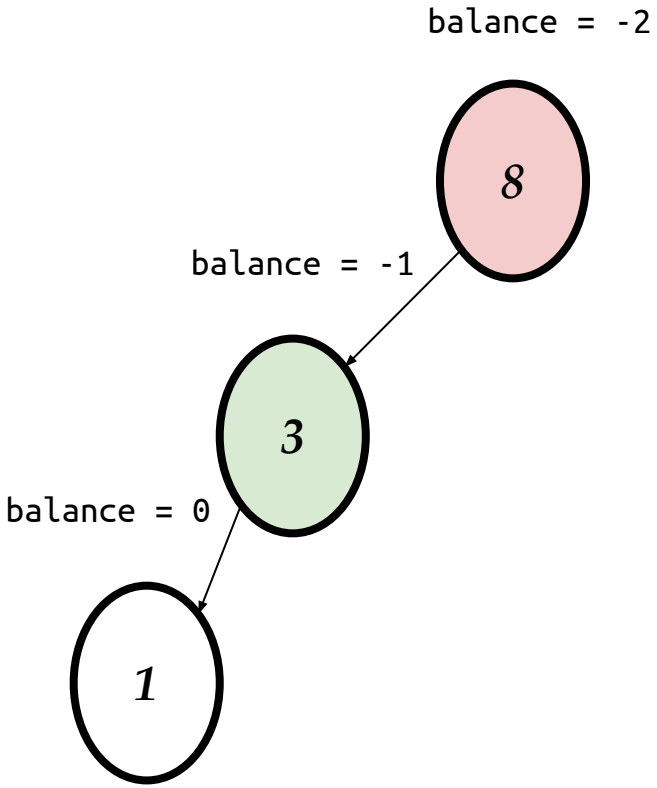
Inserção

- Ao inserir:
  - Atualizar alturas
  - Atualizar balanceamentos
- Olhar para o nó "avô"

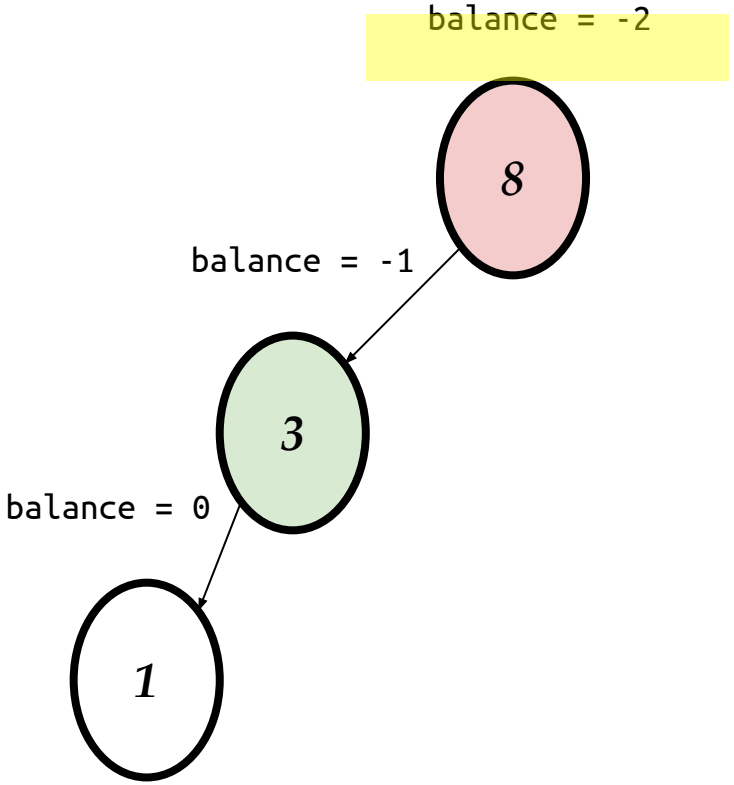


Inserção

- Ao inserir:
  - Atualizar alturas
  - Atualizar balanceamentos
- Olhar para o nó "avô"
  - O mesmo desbalanceou?



Inserção

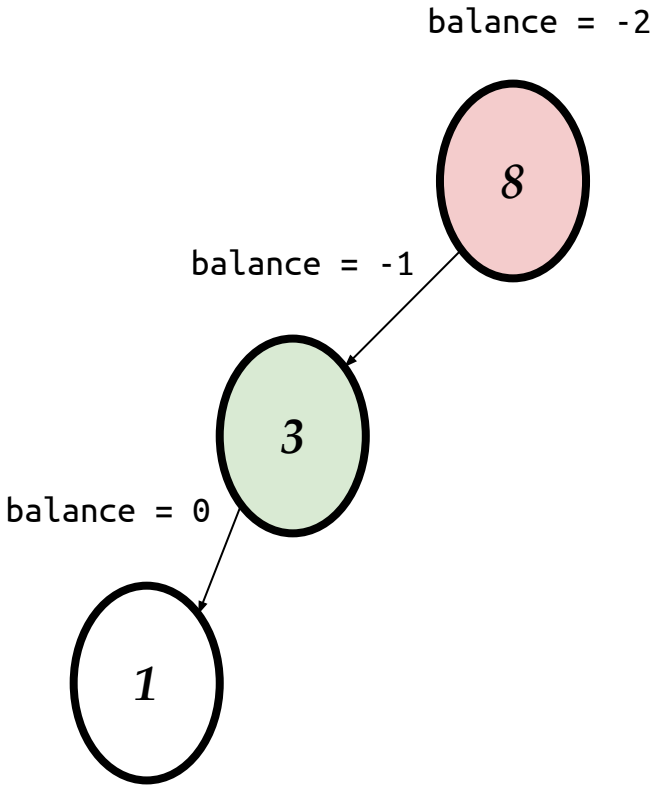


- Ao inserir:
  - Atualizar alturas
  - Atualizar balanceamentos
- Olhar para o nó "avô"
  - O mesmo desbalanceou?

Sim!

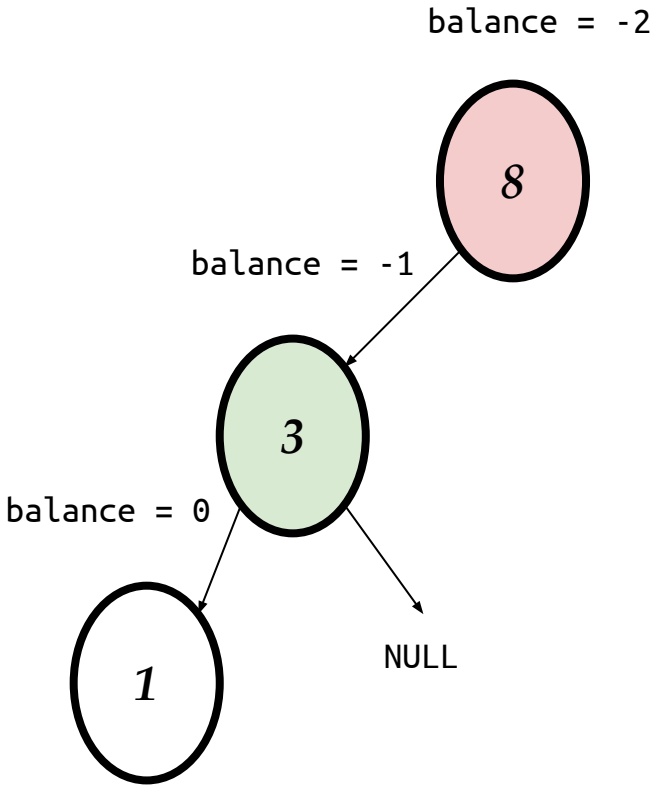
Inserção

- Ao inserir:
  - Atualizar alturas
  - Atualizar balanceamentos
- Olhar para o nó "avô"
  - O mesmo desbalanceou?
  - Sim!
- Consertamos com uma rotação para a **direita**



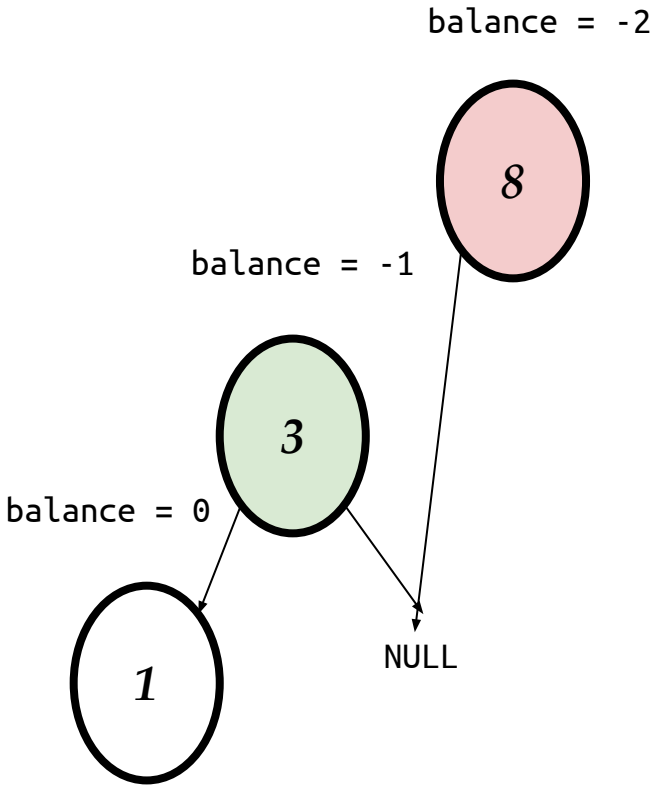
Inserção

- Ao inserir:
  - Atualizar alturas
  - Atualizar balanceamentos
- Olhar para o nó "avô"
  - O mesmo desbalanceou?
  - Sim!
- Consertamos com uma rotação para a **direita**



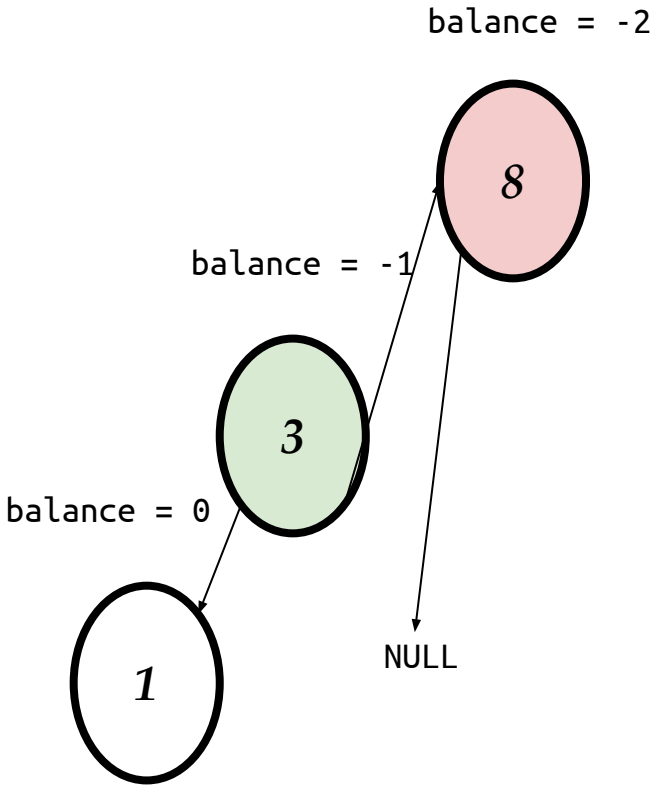
Inserção

- Ao inserir:
  - Atualizar alturas
  - Atualizar balanceamentos
- Olhar para o nó "avô"
  - O mesmo desbalanceou?
  - Sim!
- Consertamos com uma rotação para a **direita**



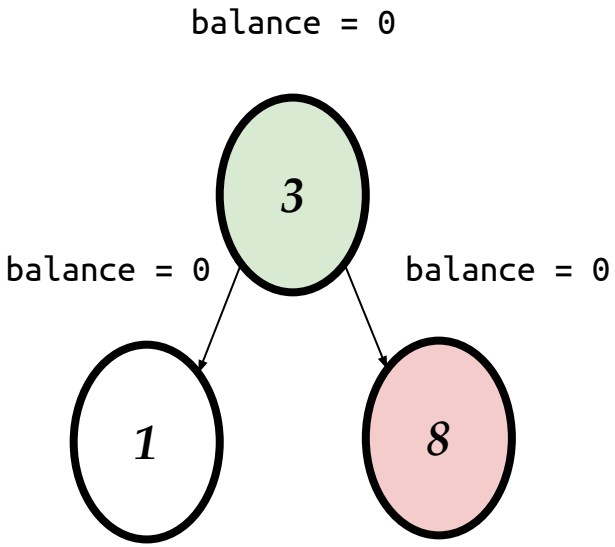
Inserção

- Ao inserir:
  - Atualizar alturas
  - Atualizar balanceamentos
- Olhar para o nó "avô"
  - O mesmo desbalanceou?
  - Sim!
- Consertamos com uma rotação para a **direita**



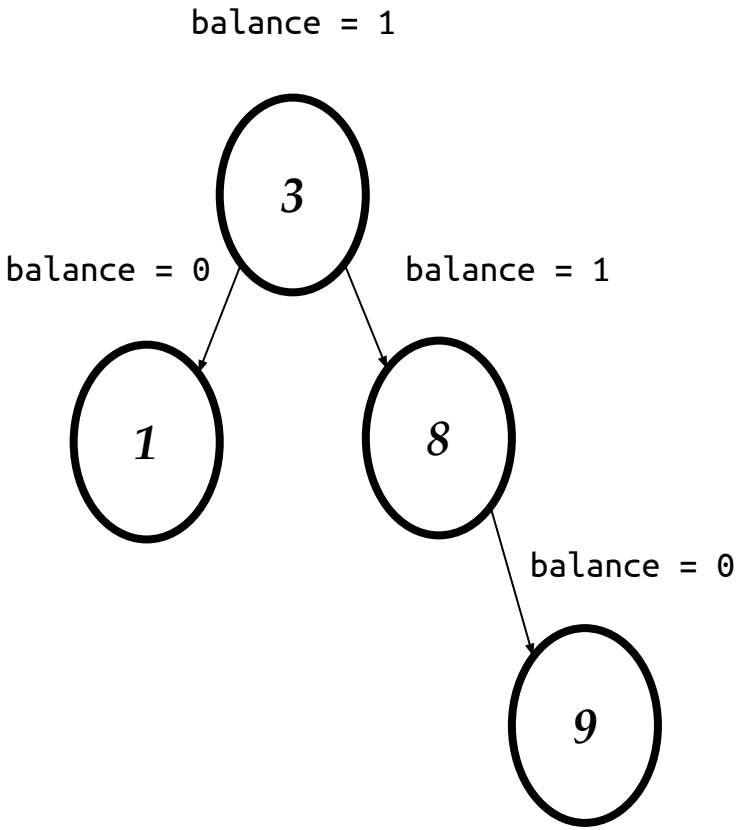
Inserção

- Ao inserir:
  - Atualizar alturas
  - Atualizar balanceamentos
- Olhar para o nó "avô"
  - O mesmo desbalanceou?
  - Sim!
- Consertamos com uma rotação para a **direita**



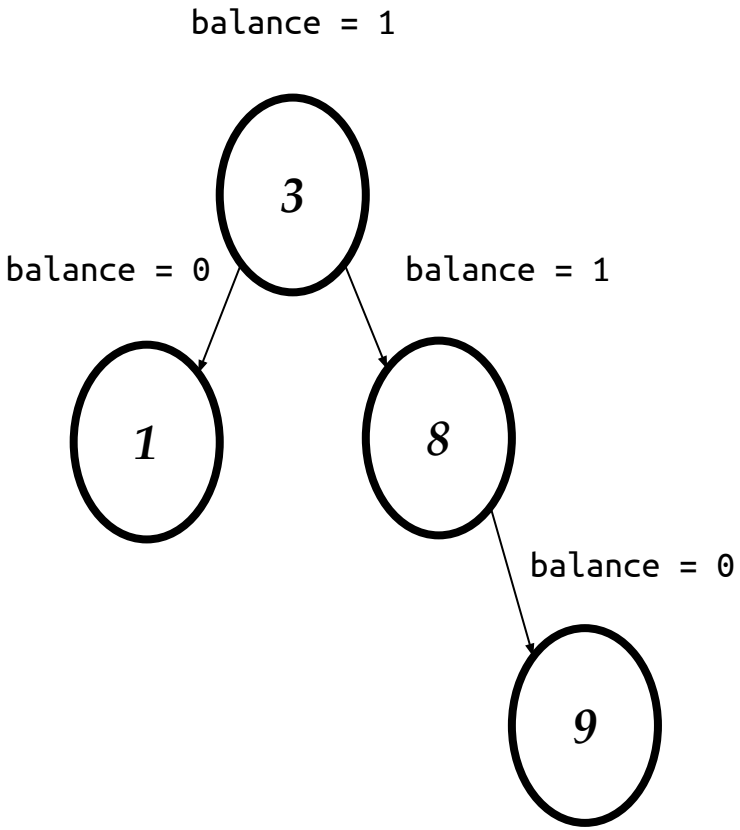
Inserção: Outro Caso

- Inserindo nó 9



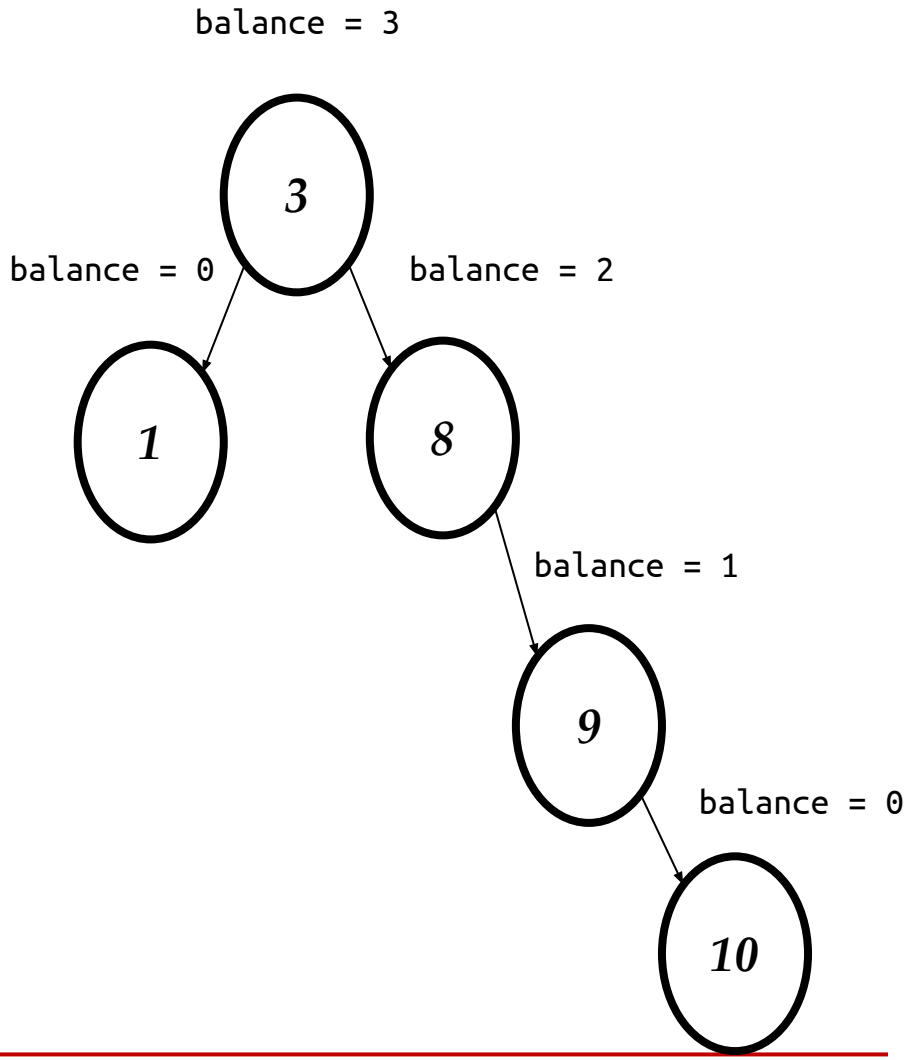
Inserção: Outro Caso

- Inserindo nó 9
- Inserindo nó 10



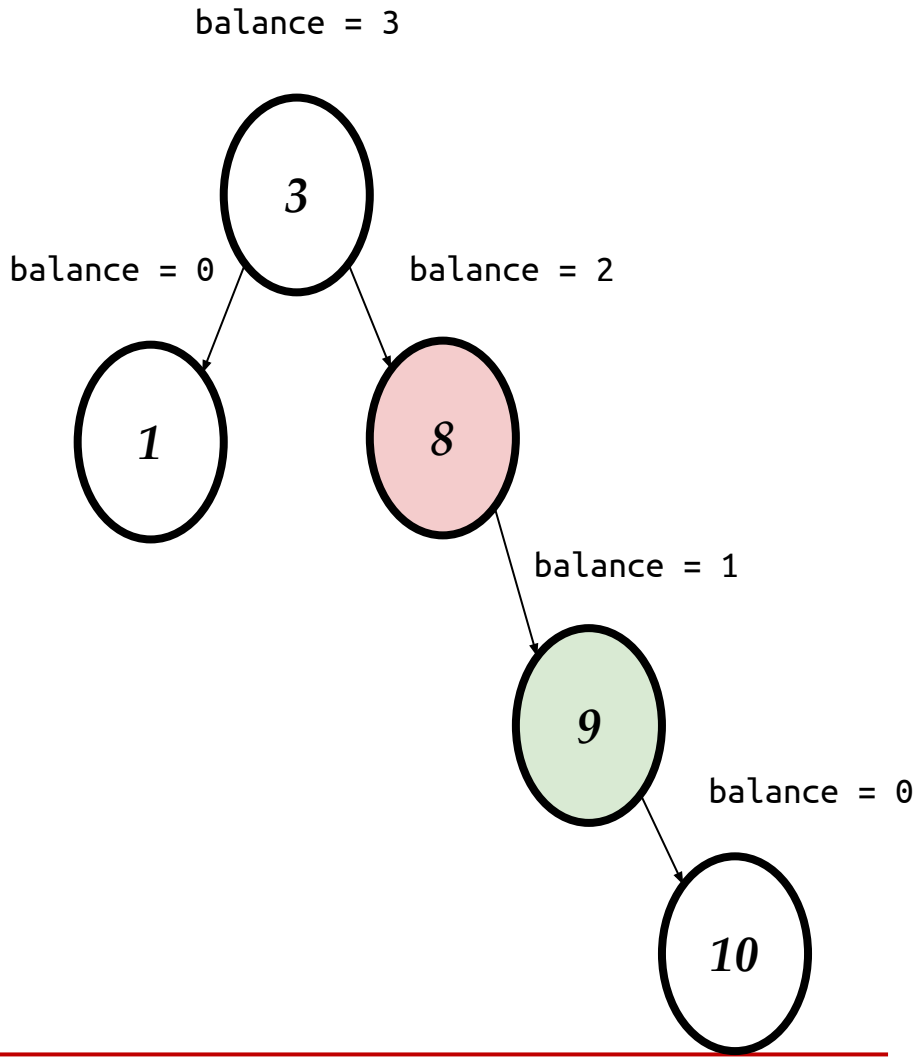
Inserção: Outro Caso

- Inserindo nó 9
- Inserindo nó 10



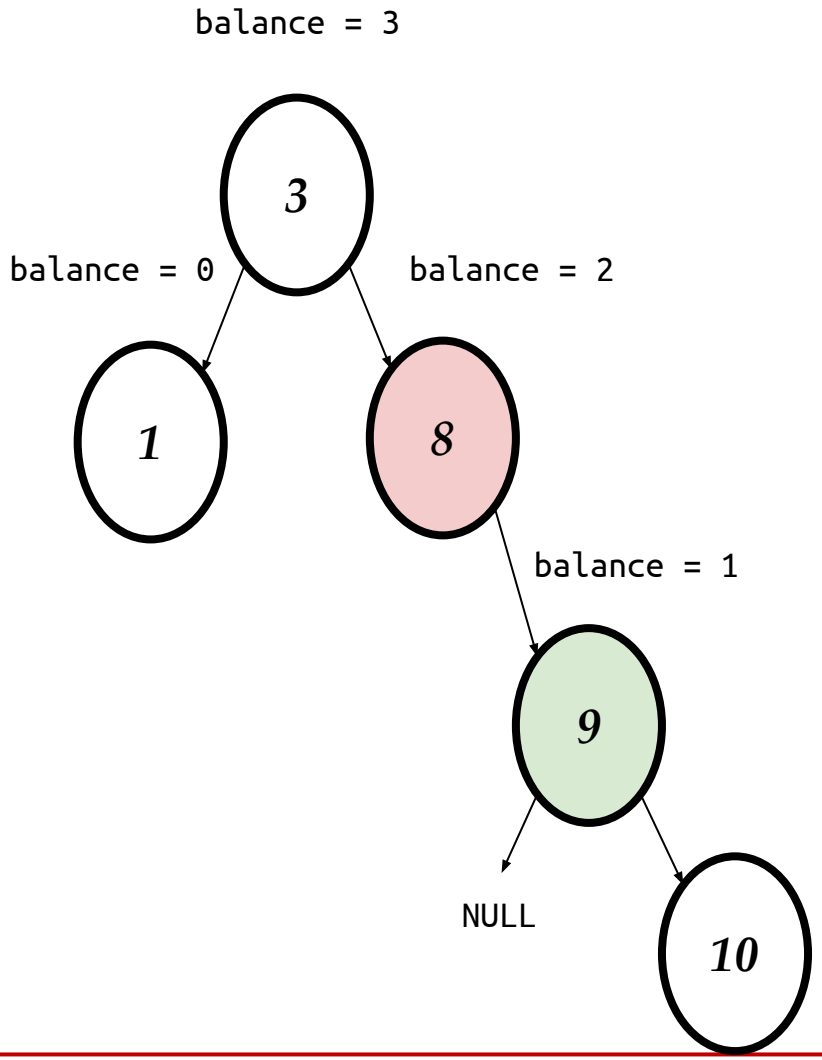
Rotação!

Consertamos com uma rotação para a **esquerda**



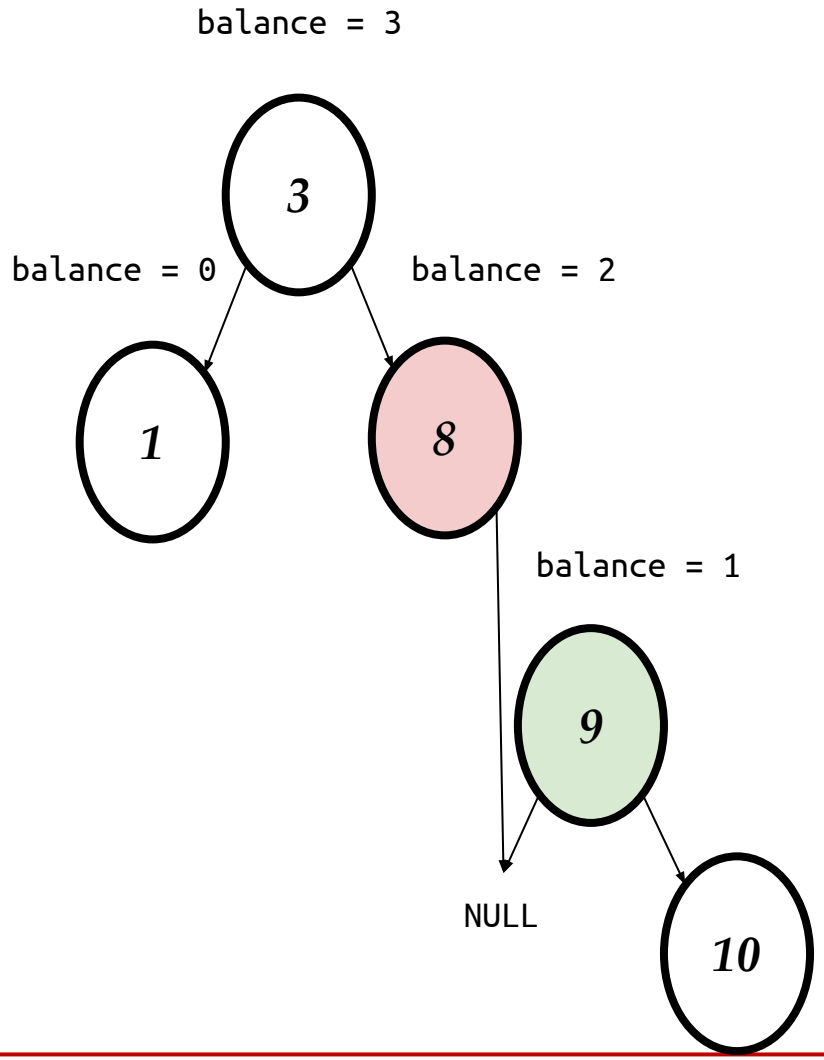
Rotação!

Consertamos com uma rotação para a **esquerda**



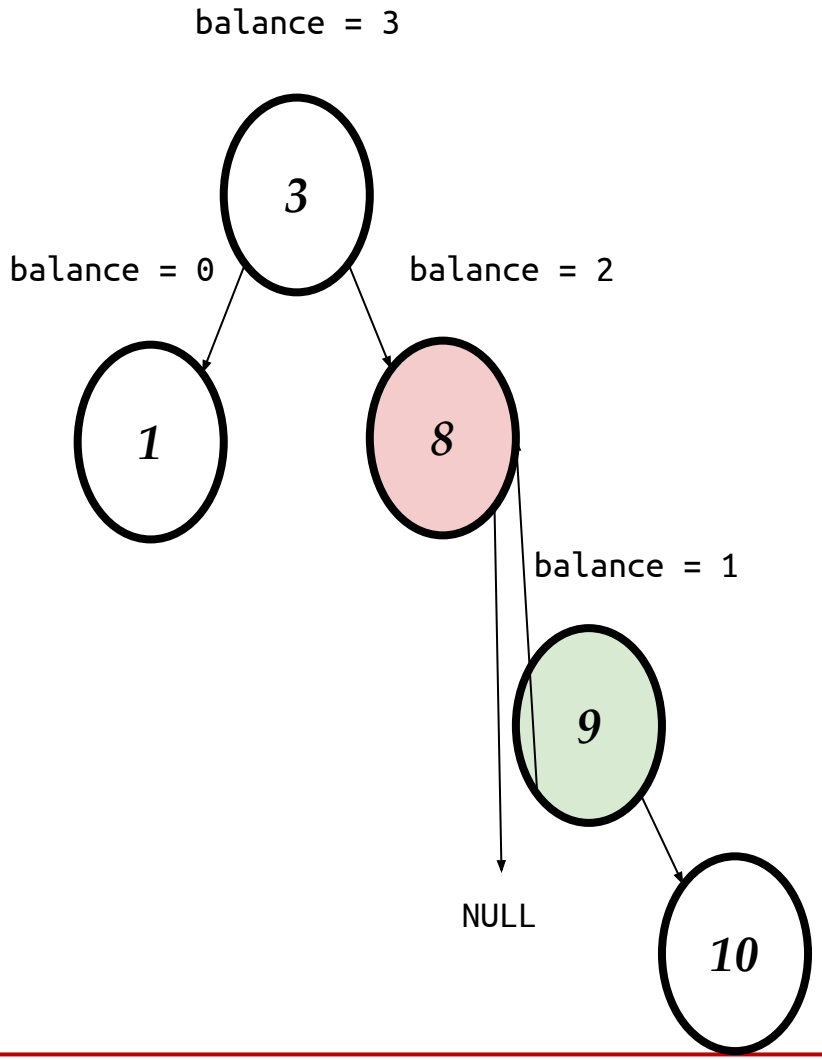
Rotação!

Consertamos com uma rotação para a **esquerda**



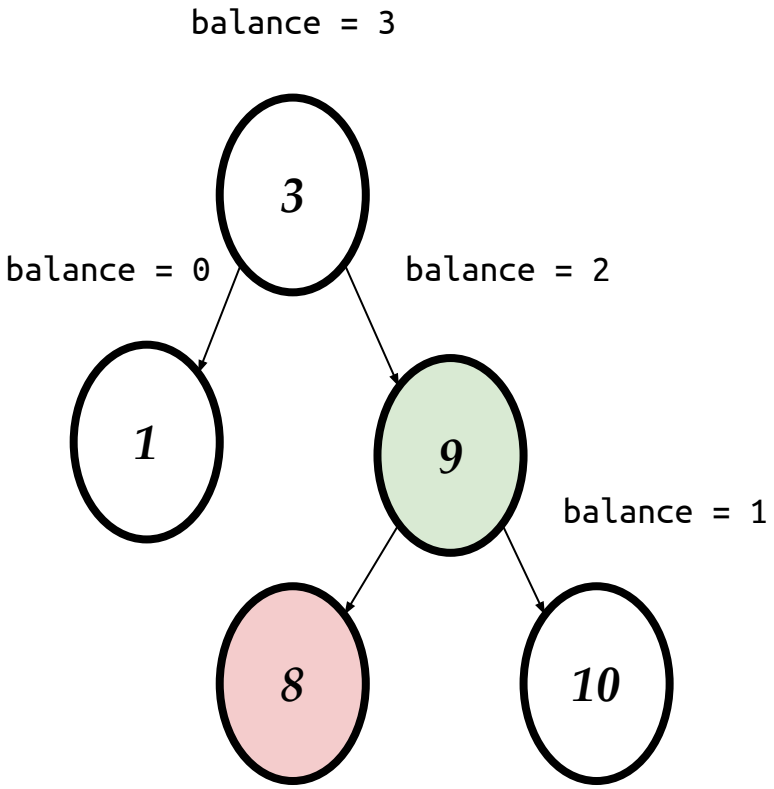
Rotação!

Consertamos com uma rotação para a **esquerda**



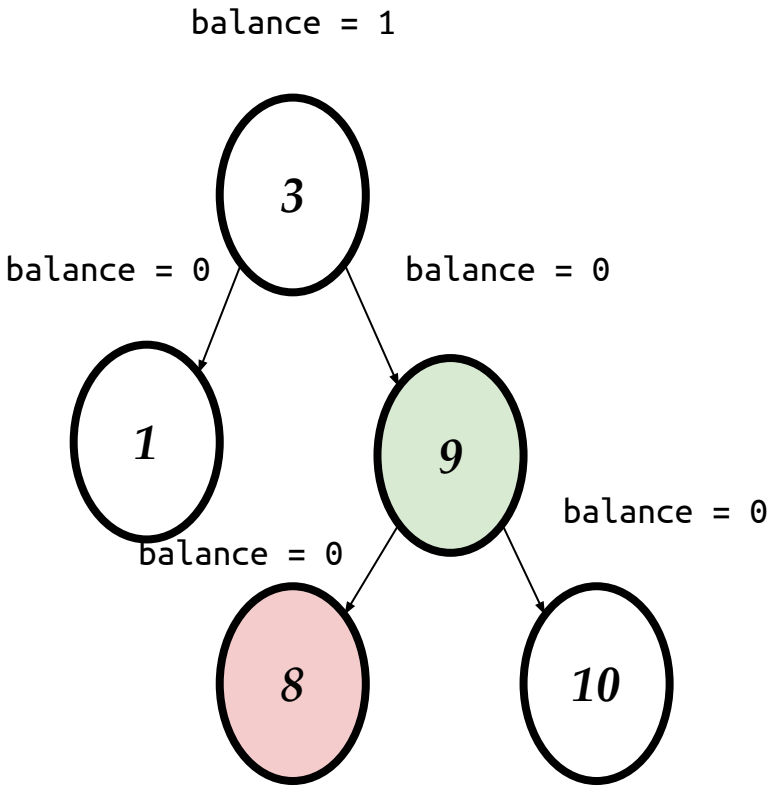
Rotação!

Consertamos com uma rotação para a **esquerda**



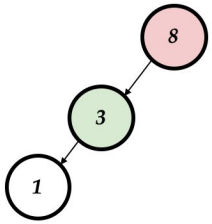
Rotação!

Consertamos com uma rotação para a **esquerda**



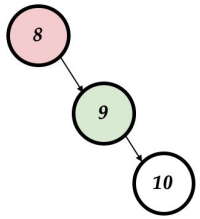
Inserção – Vimos 2 Casos

1. Caso 1: Inserimos na **esquerda** do parent e avô tinha desbalanceamento **negativo**



Solução: Rotação para a direita

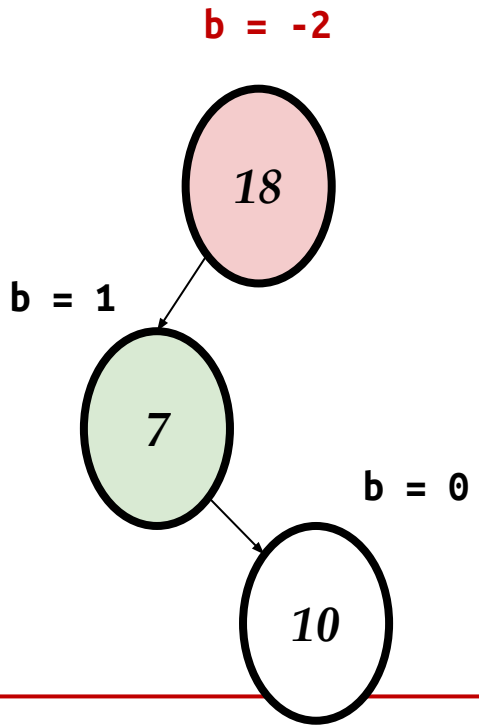
1. Caso 2: Inserimos na **direita** do parent e avô tinha desbalanceamento **positivo**



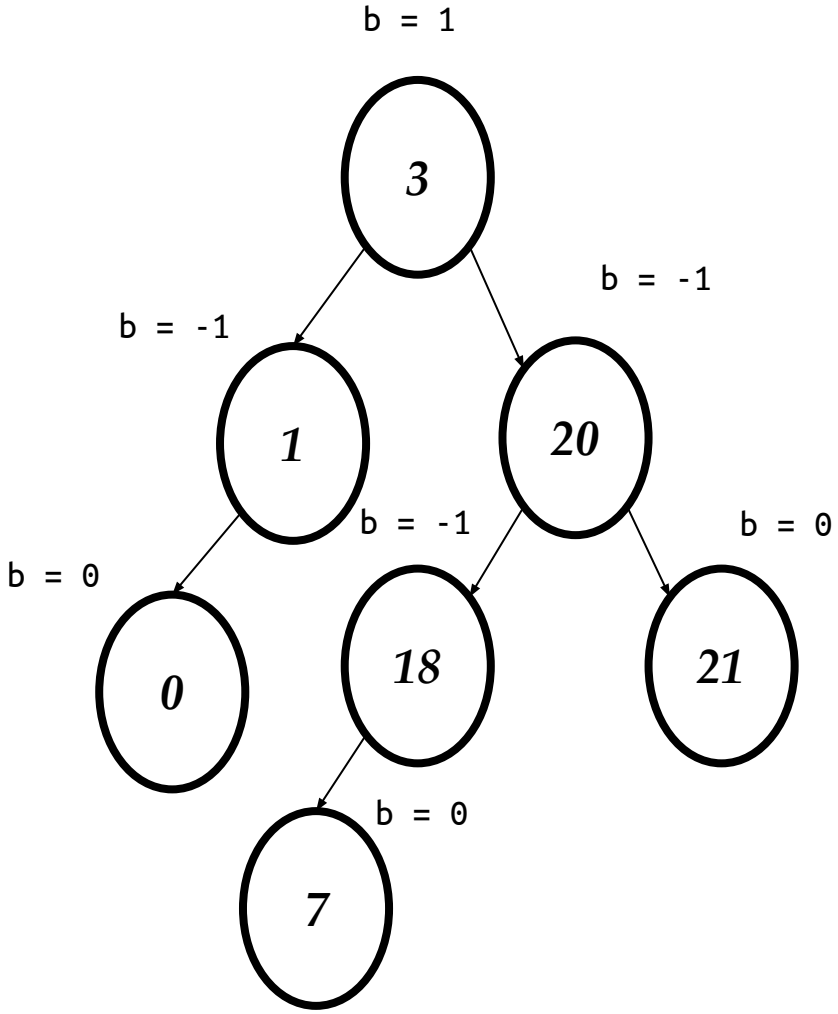
Solução: Rotação para a esquerda

Inserção – Caso 3

- Insere na **direita** do parent e avô fica com desbalanceamento **negativo**

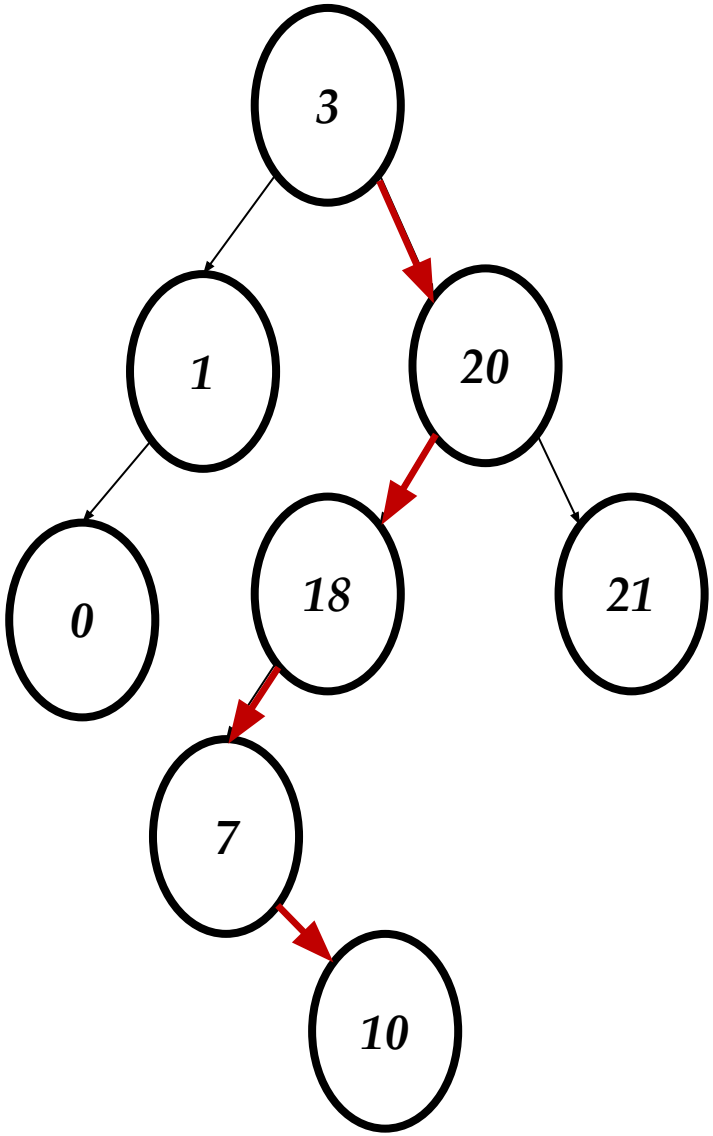


Exemplo – Caso 3



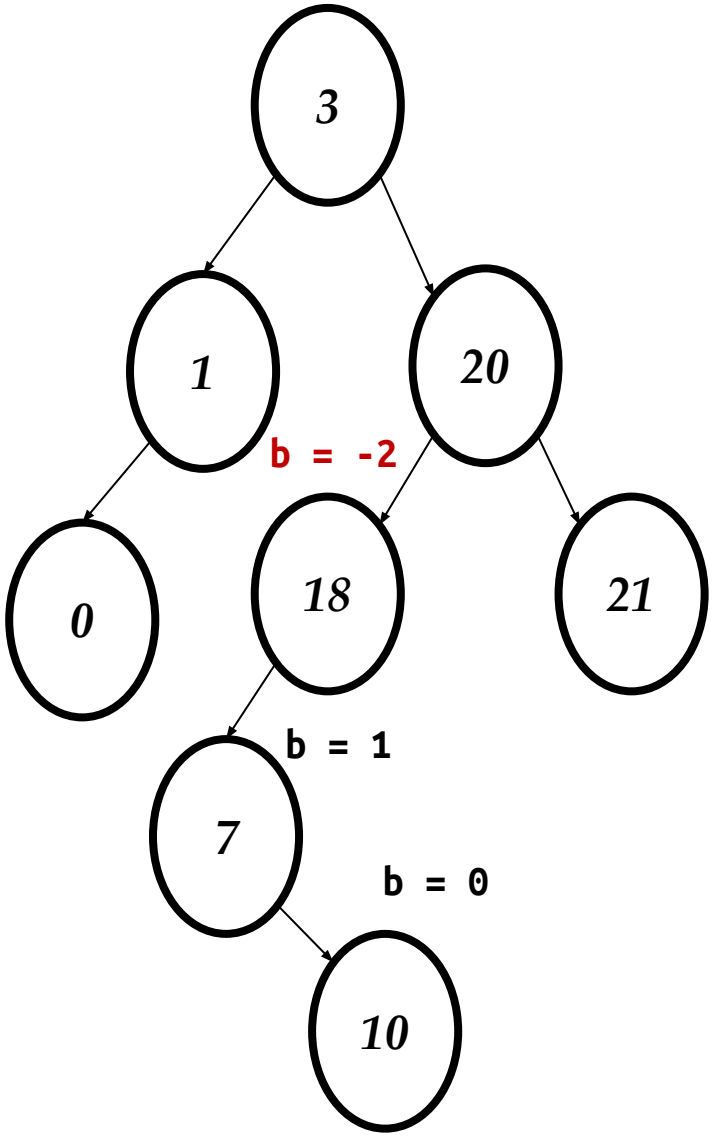
Exemplo – Caso 3

- Inserindo nó 10



Exemplo – Caso 3

- Inserindo nó 10

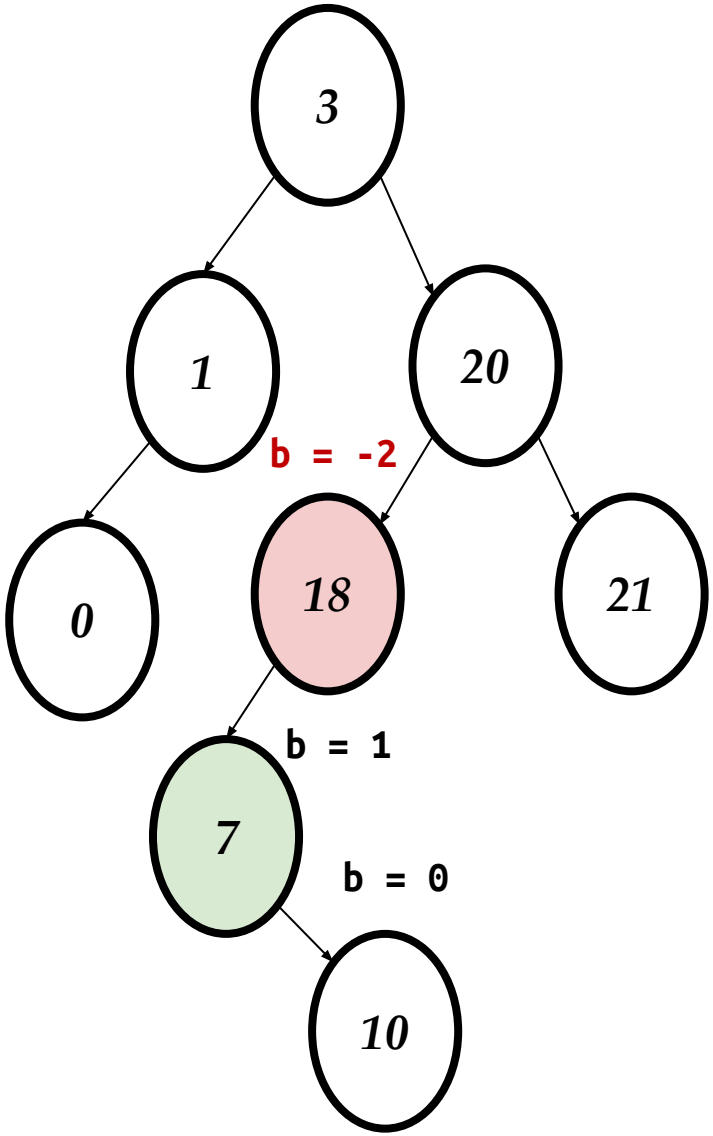


Exemplo – Caso 3

- Inserindo nó 10

Caso 1 – Avô com desbalanceamento negatis o ra
filho da lho
(todos neg

Solução: direita

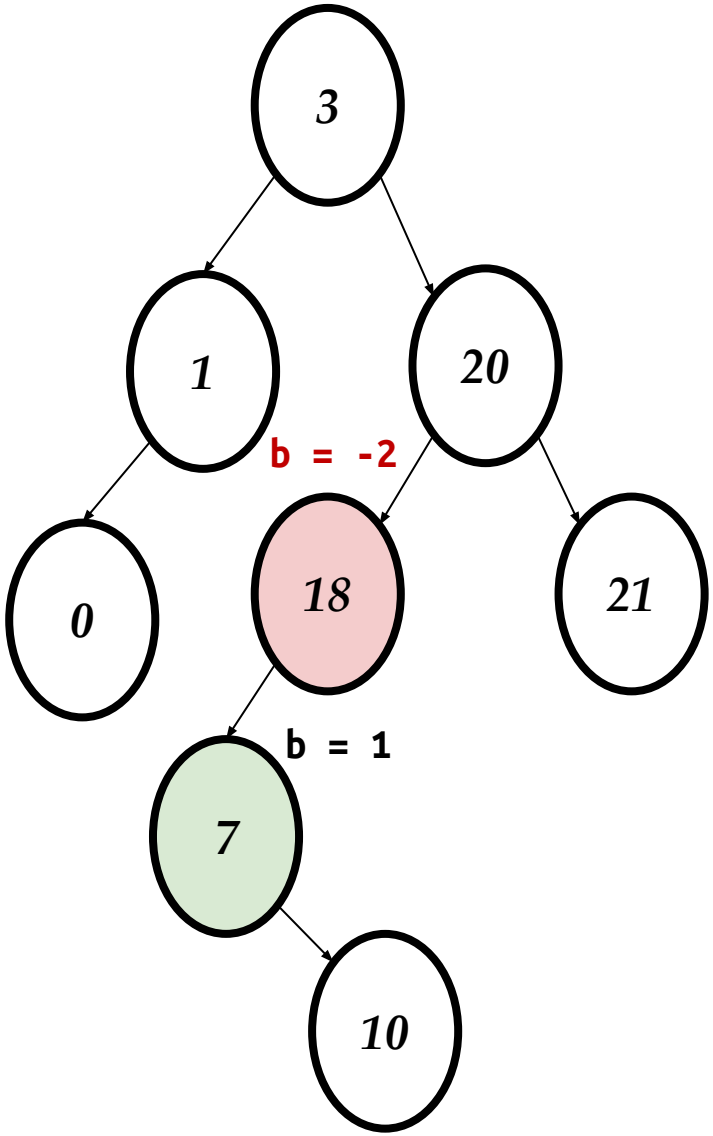


Como rotacionar?

Exemplo – Caso 3

- Solução: Transformar em um dos casos conhecidos
- 2 rotações

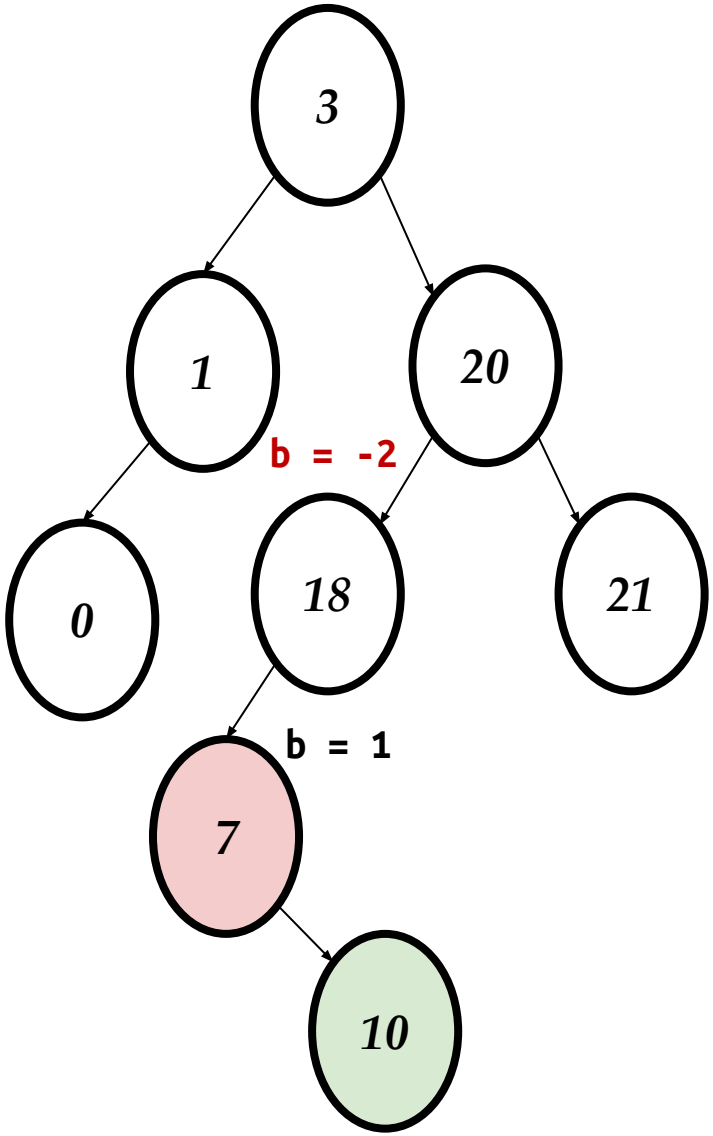
1. Rotação para esquerda no 7



Exemplo – Caso 3

- Solução: Transformar em um dos casos conhecidos
- 2 rotações

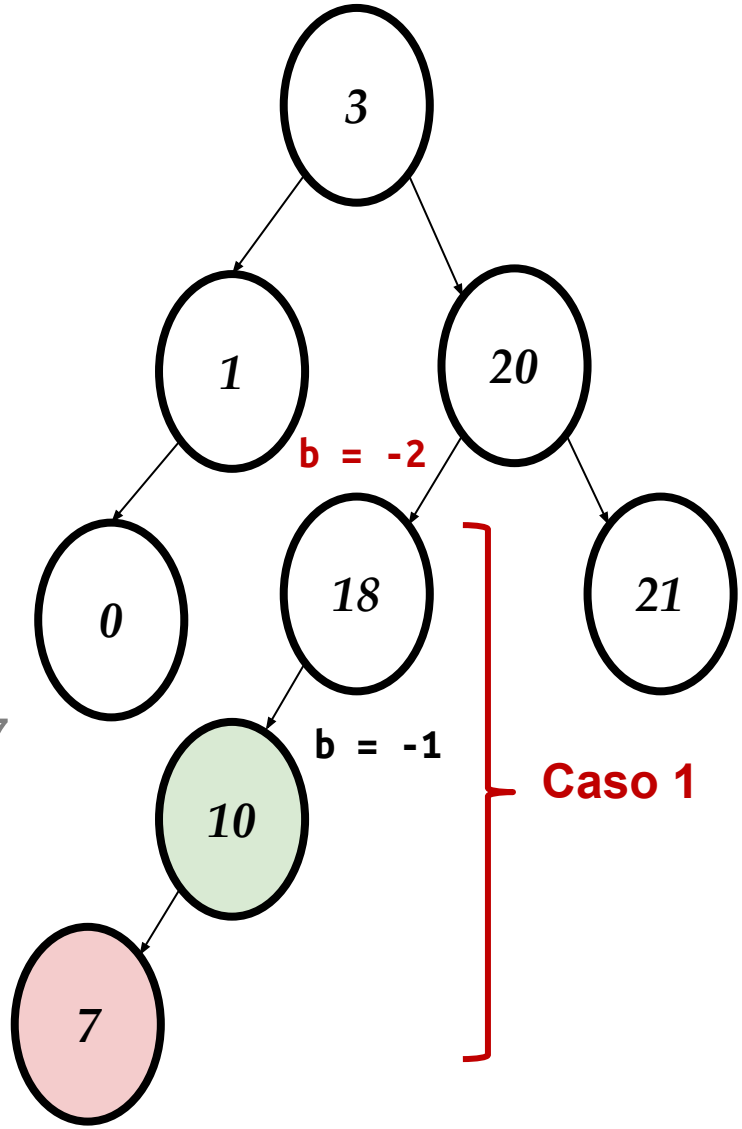
1. Rotação para esquerda no 7



Exemplo – Caso 3

- Solução: Transformar em um dos casos conhecidos
- 2 rotações

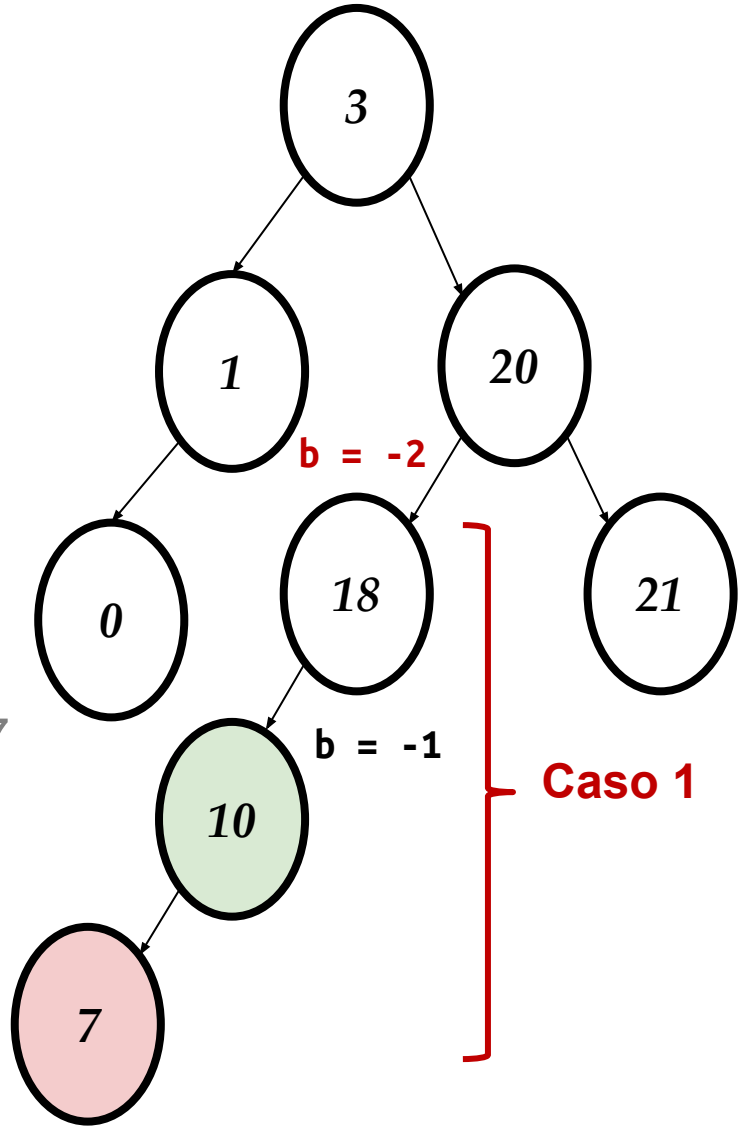
1. Rotação para esquerda no 7



Exemplo – Caso 3

- Solução: Transformar em um dos casos conhecidos
- 2 rotações

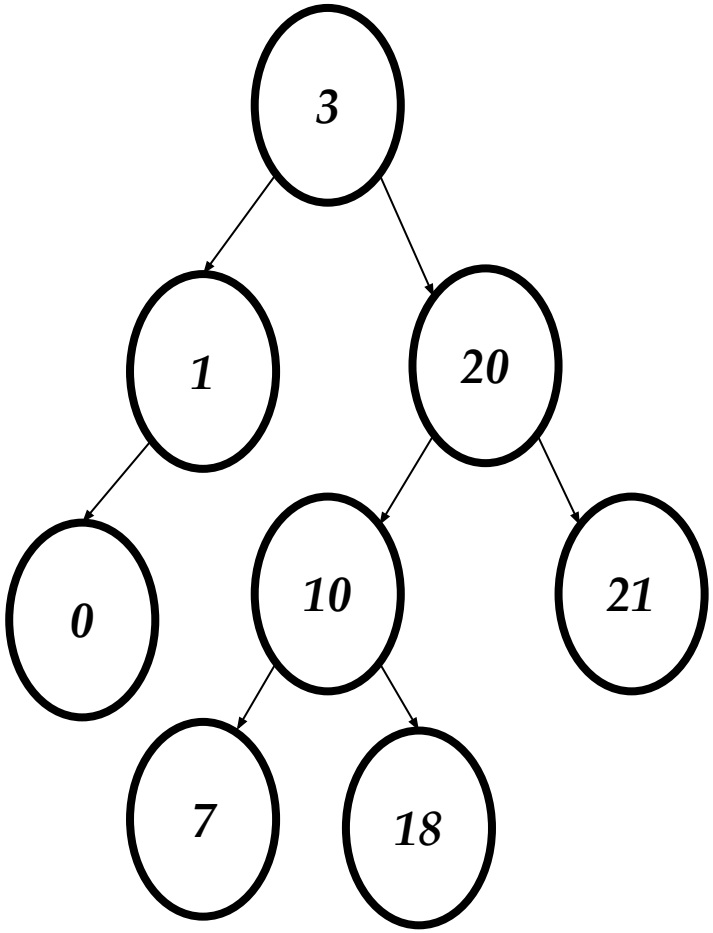
1. Rotação para esquerda no 7
2. Rotação para direita no 10



Exemplo – Caso 3

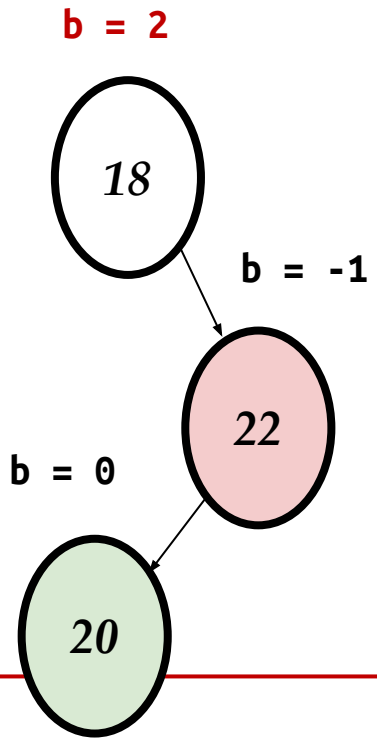
- Solução: Transformar em um dos casos conhecidos
- 2 rotações

1. Rotação para esquerda no 7
2. Rotação para direita no 10

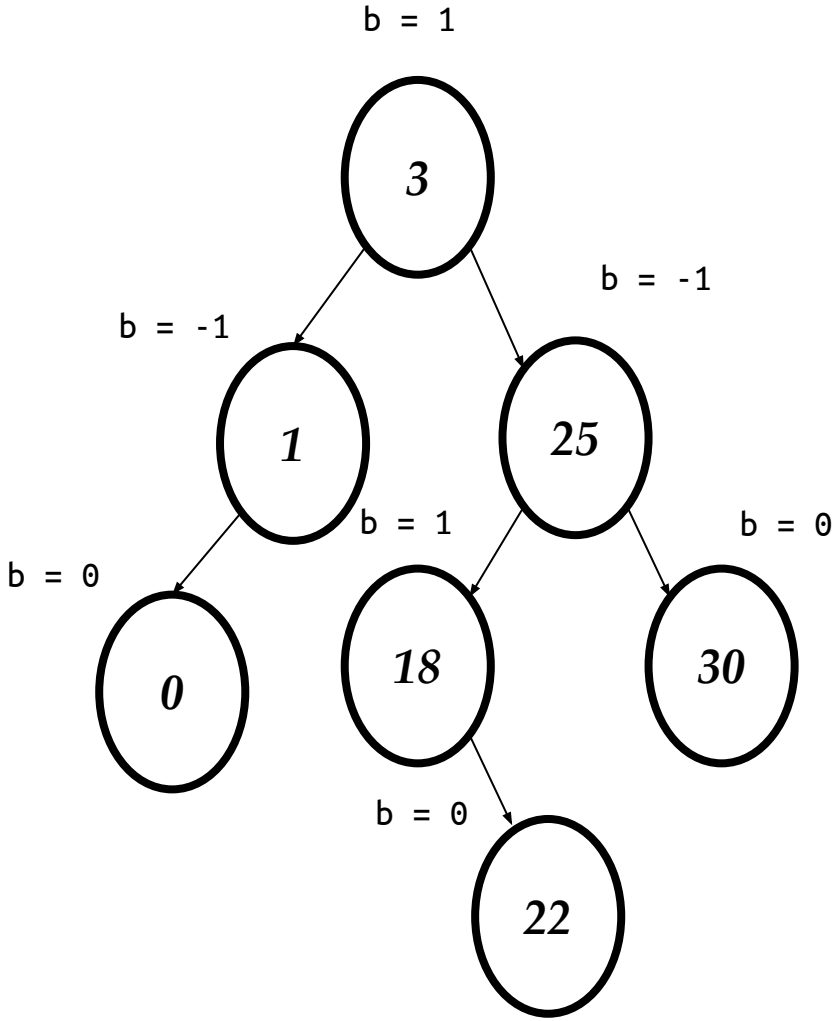


Inserção – Caso 4

- Insere na **esquerda** do parent e avô fica com desbalanceamento **positivo**

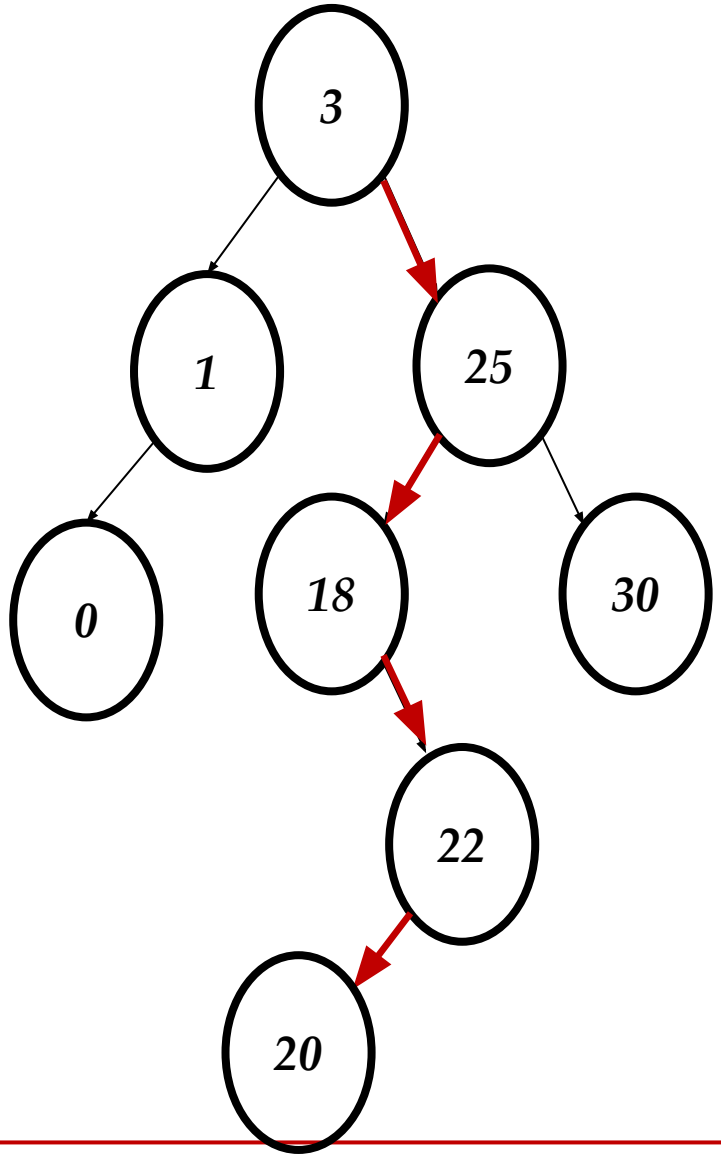


Exemplo – Caso 4

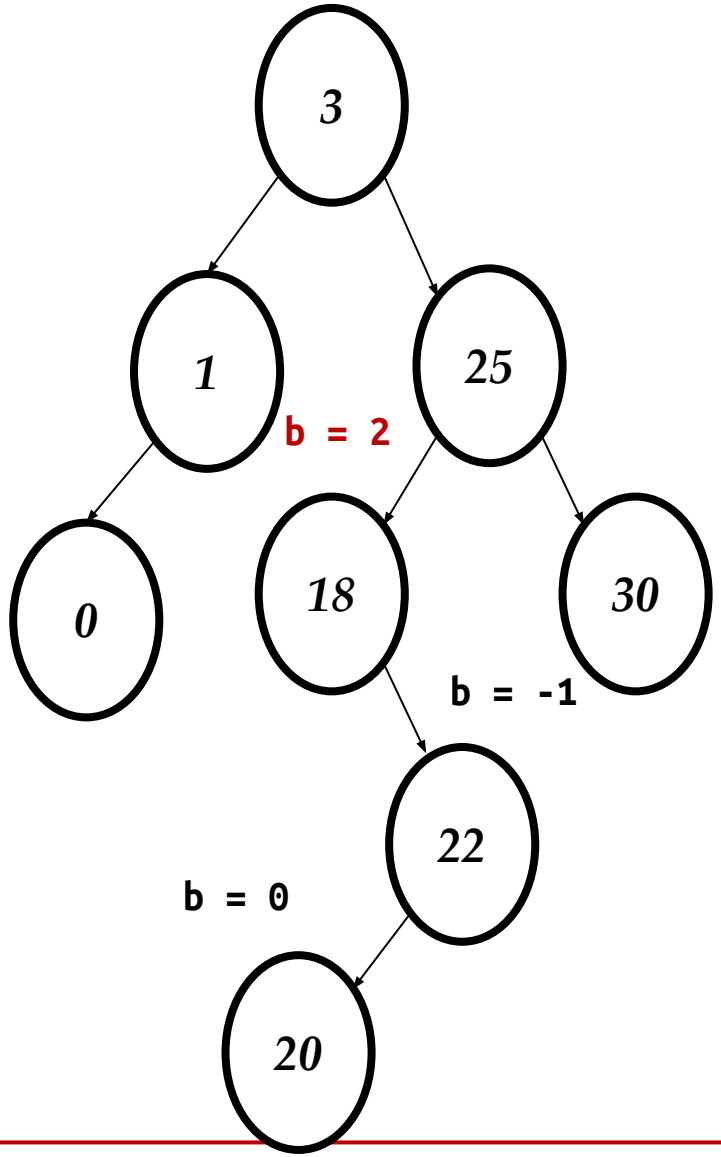


Exemplo – Caso 4

- Inserindo nó 20

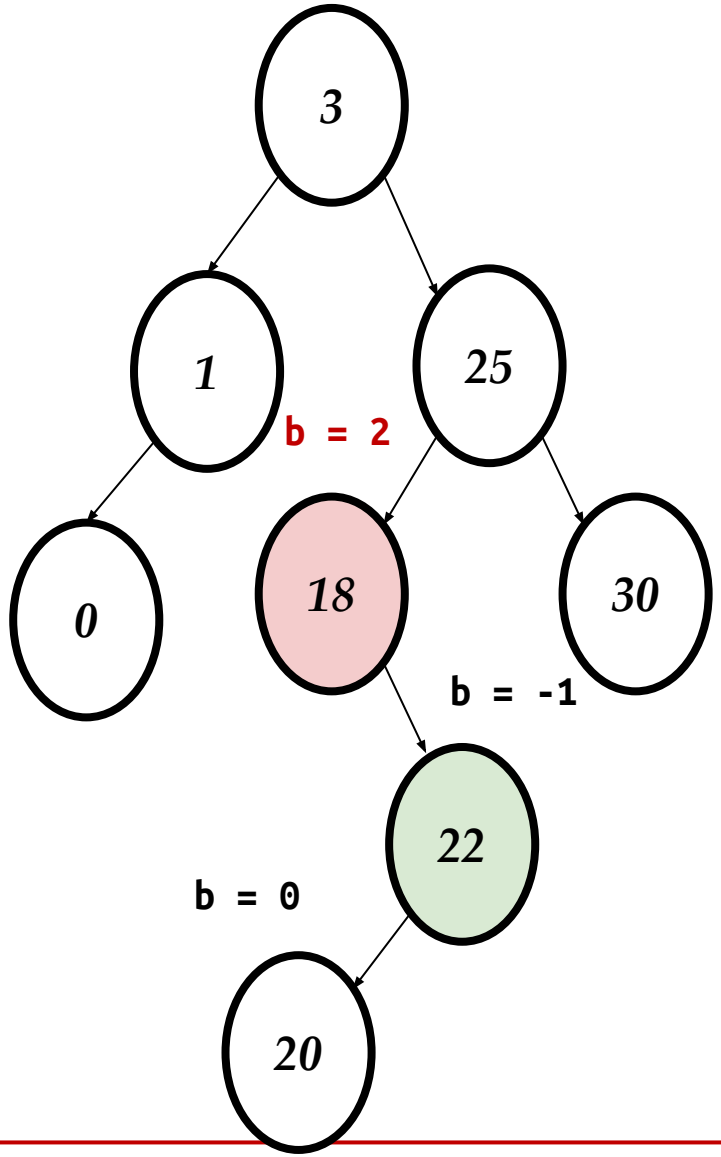


Exemplo – Caso 4



Exemplo – Caso 4

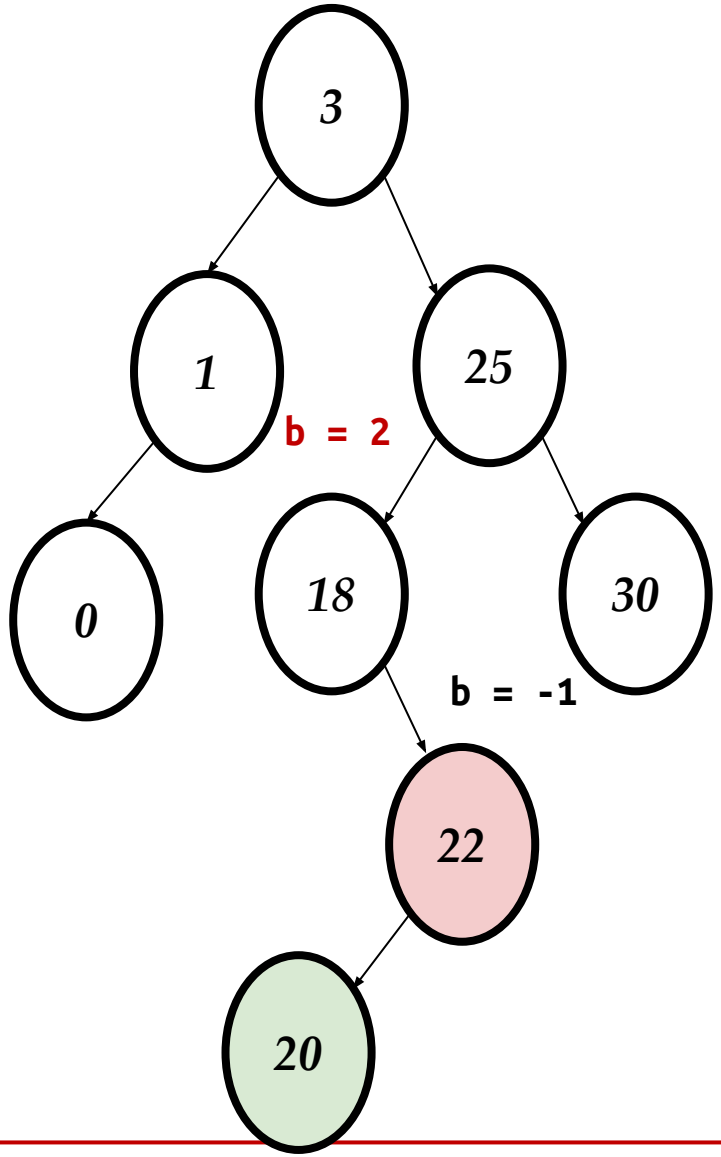
- Solução: Transformar em um dos casos conhecidos (**caso 2**)
- 2 rotações



Exemplo – Caso 4

- Solução: Transformar em um dos casos conhecidos (**caso 2**)
- 2 rotações

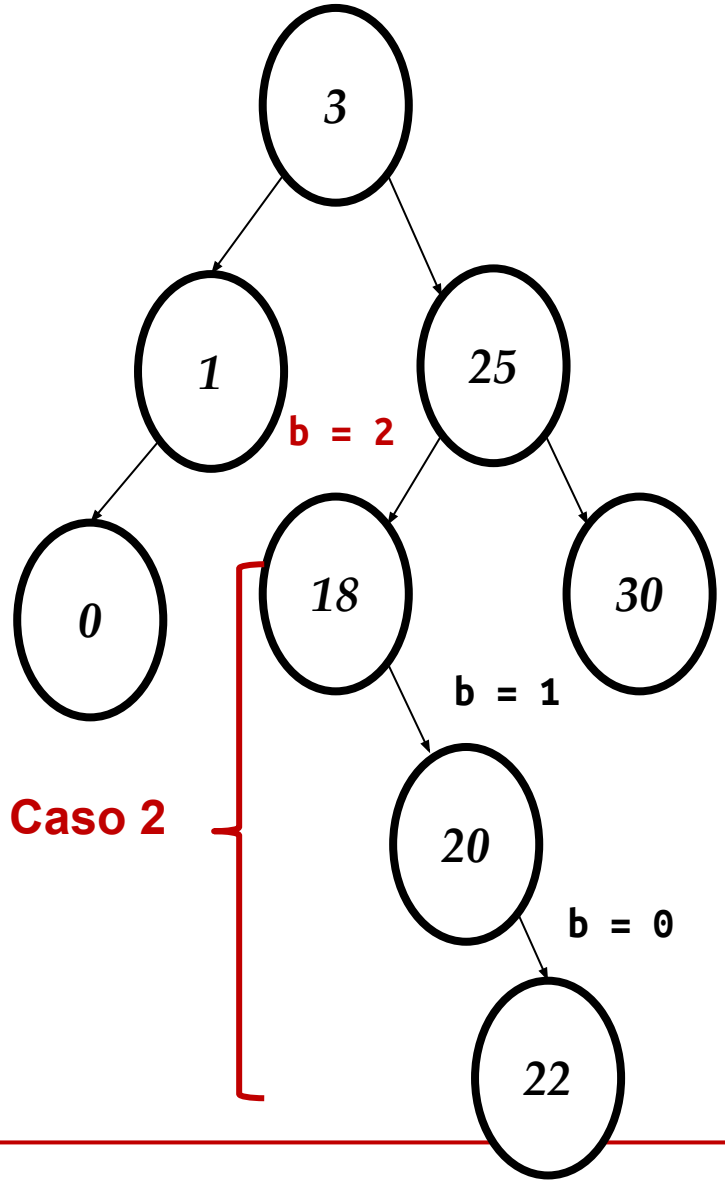
1. Rotação para direita no 22



Exemplo – Caso 4

- Solução: Transformar em um dos casos conhecidos (**caso 2**)
- 2 rotações

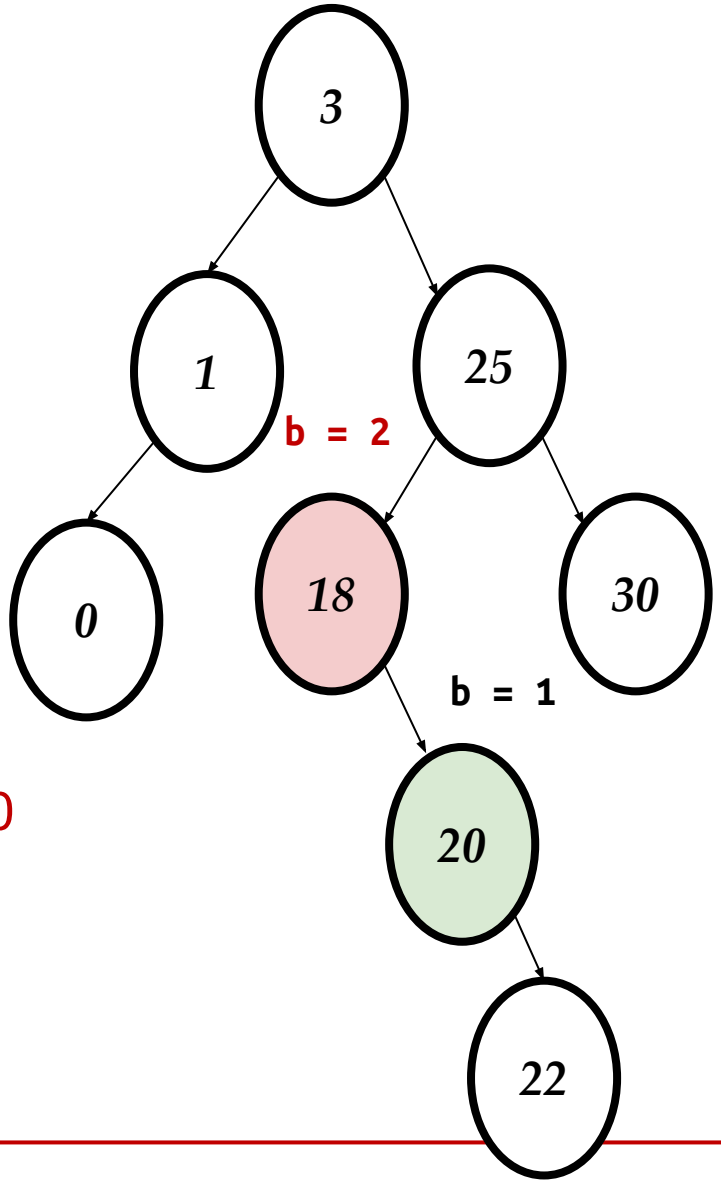
1. Rotação para direita no 22



Exemplo – Caso 4

- Solução: Transformar em um dos casos conhecidos (**caso 2**)
- 2 rotações

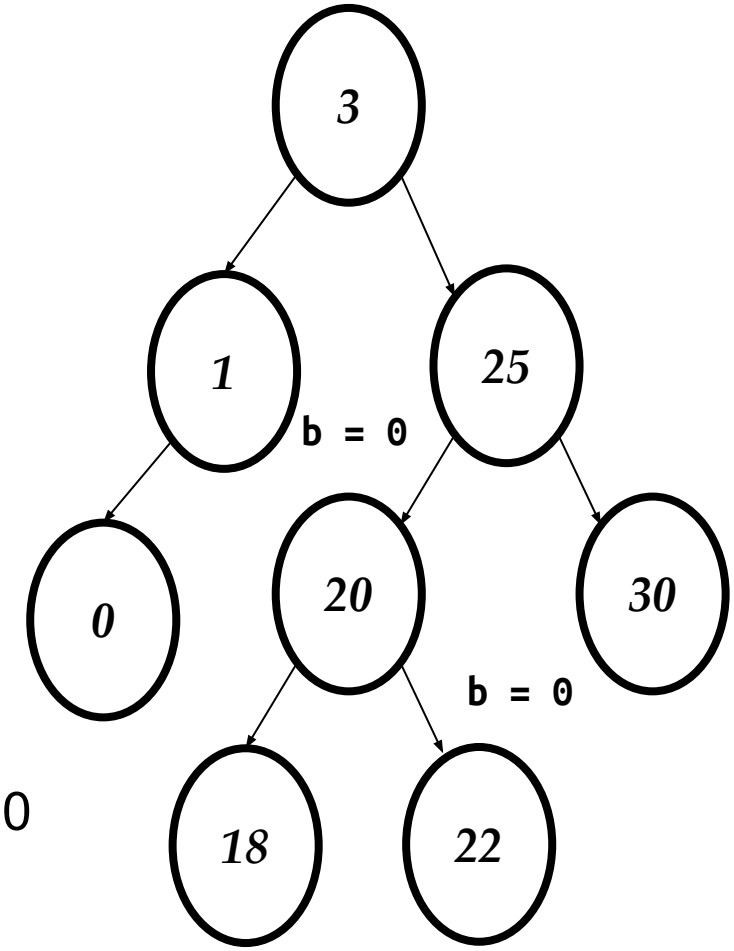
1. Rotação para direita no 22
2. Rotação para esquerda no 20



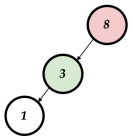
Exemplo – Caso 4

- Solução: Transformar em um dos casos conhecidos (**caso 2**)
- 2 rotações

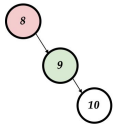
1. Rotação para direita no 22
2. Rotação para esquerda no 20



Inserção – 4 Casos Possíveis:

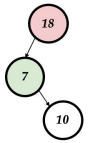


- **Caso 1**: Insere na **esquerda** do parent e avô fica com desbalanceamento **negativo**

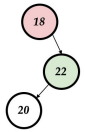


- **Caso 2**: Insere na **direita** do parent e avô fica com desbalanceamento **positivo**

1 rotacão



- **Caso 3**: Insere na **direita** do parent e avô fica com desbalanceamento **negativo**



- **Caso 4**: Insere na **esquerda** do parent e avô fica com desbalanceamento **positivo**

2 rotacões

Inserção - Código

```
node * insert(node * T, int x) {
  if (T == NULL) { initialize(T);
  } else
      if (x > T->data) { T->right = insert(T->right, x);
            if (BF(T) == -2)
                  if (x > T->right->data) T = RR(T);
                  else T = RL(T);
      } else if (x < T->data) { T->left = insert(T->left, x);
            if (BF(T) == 2)
                  if (x < T->left->data) T = LL(T);
                  else T = LR(T);
      }
  T->ht = height(T);
  return (T);
}
```

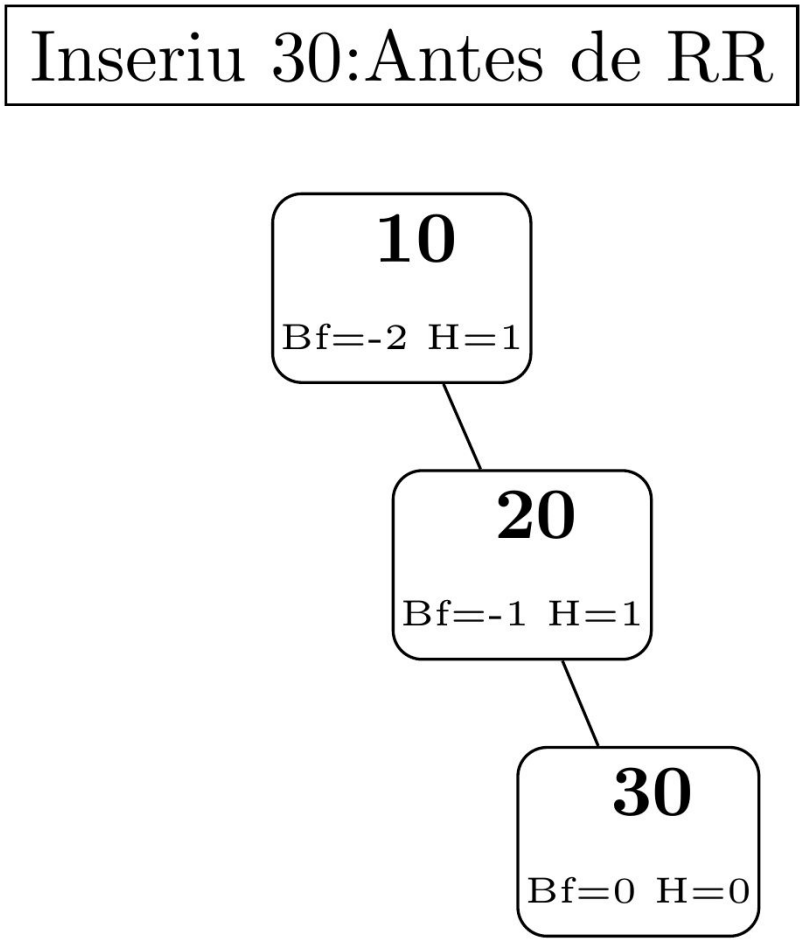
Análise da Inserção - Custos

- Achar o local do nó $O(\log(n))$
- Atualizar alturas $O(\log(n))$
- Cada rotação é $O(1)$

- **Custo Total**

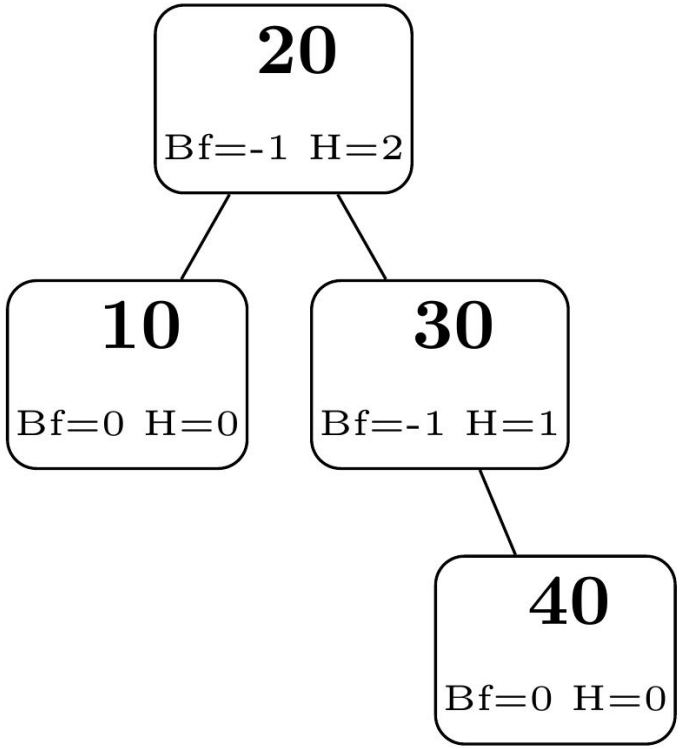
  $O(\log(n)) + O(\log(n)) + O(1) + O(1)$ **= O(log(n))**

Inserção



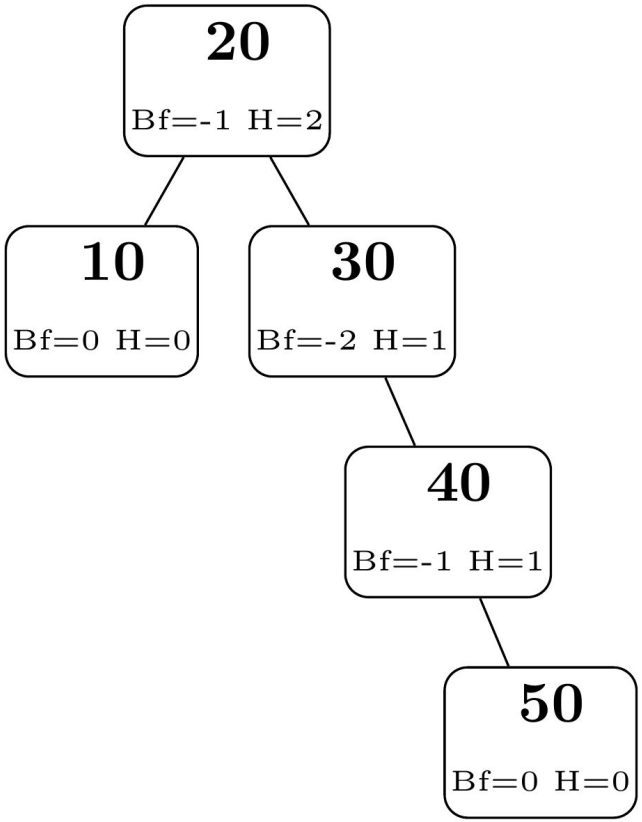
Inserção

Inseriu 10,20,30,40

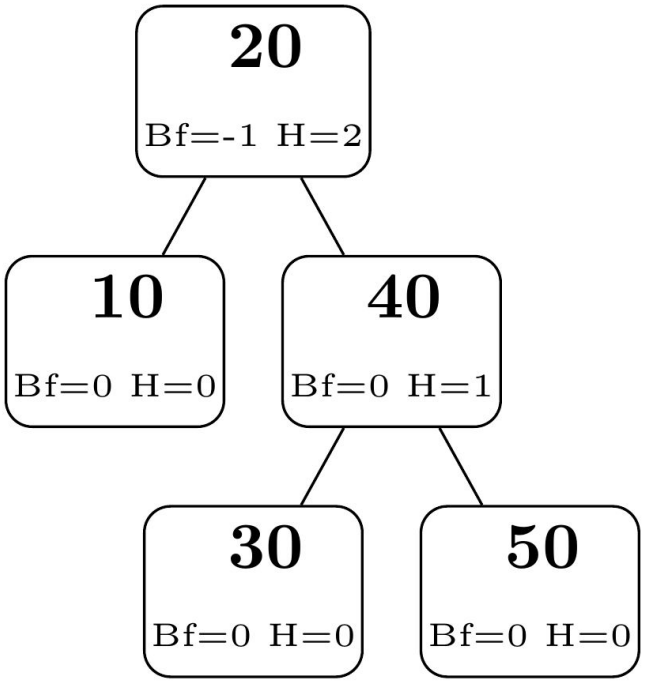


Inserção

Inseriu 50:Antes de RR

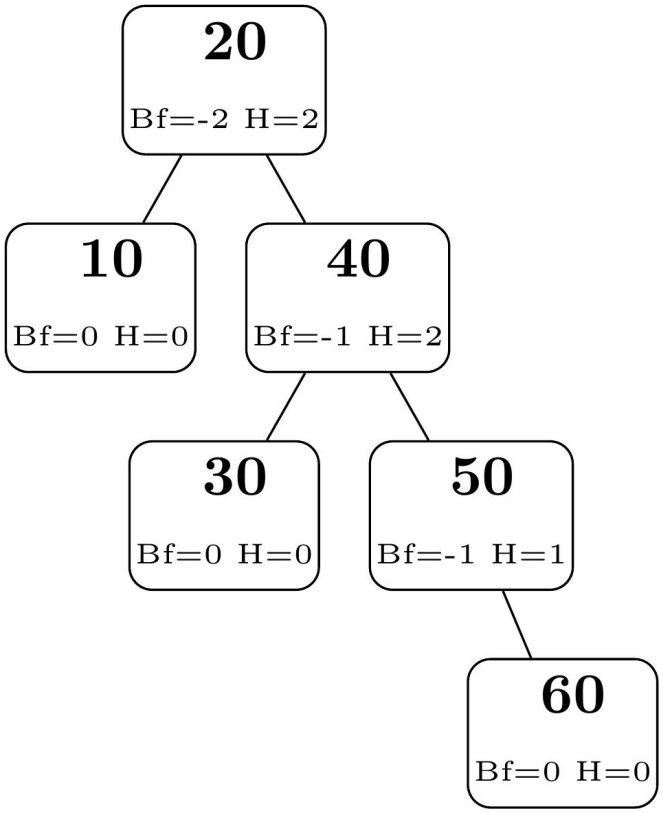


Inseriu 50

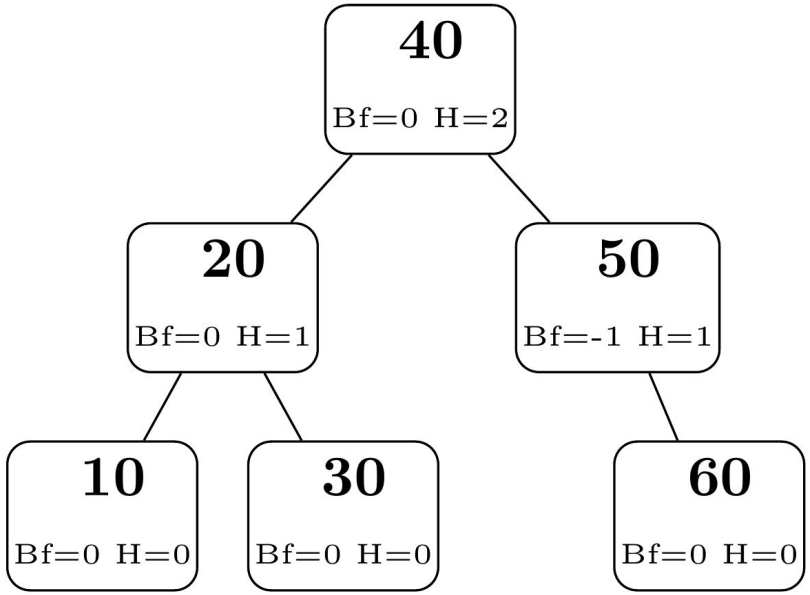


Inserção

Inseriu 60:Antes de RR

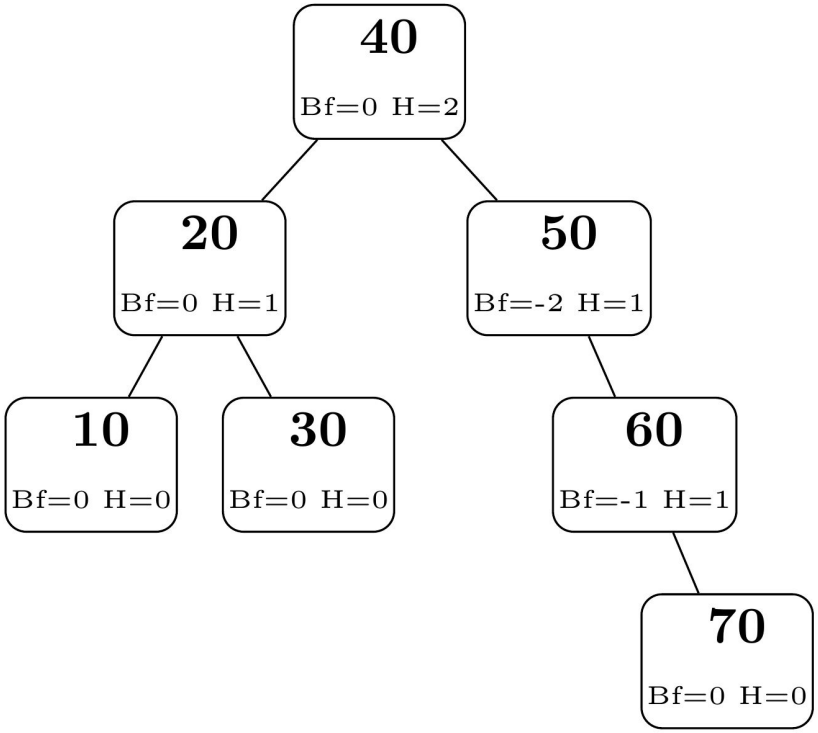


Inseriu 60

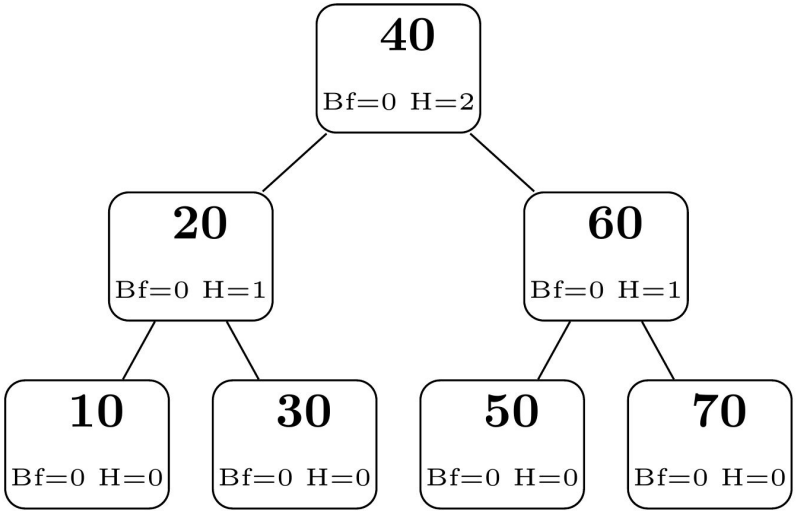


Inserção

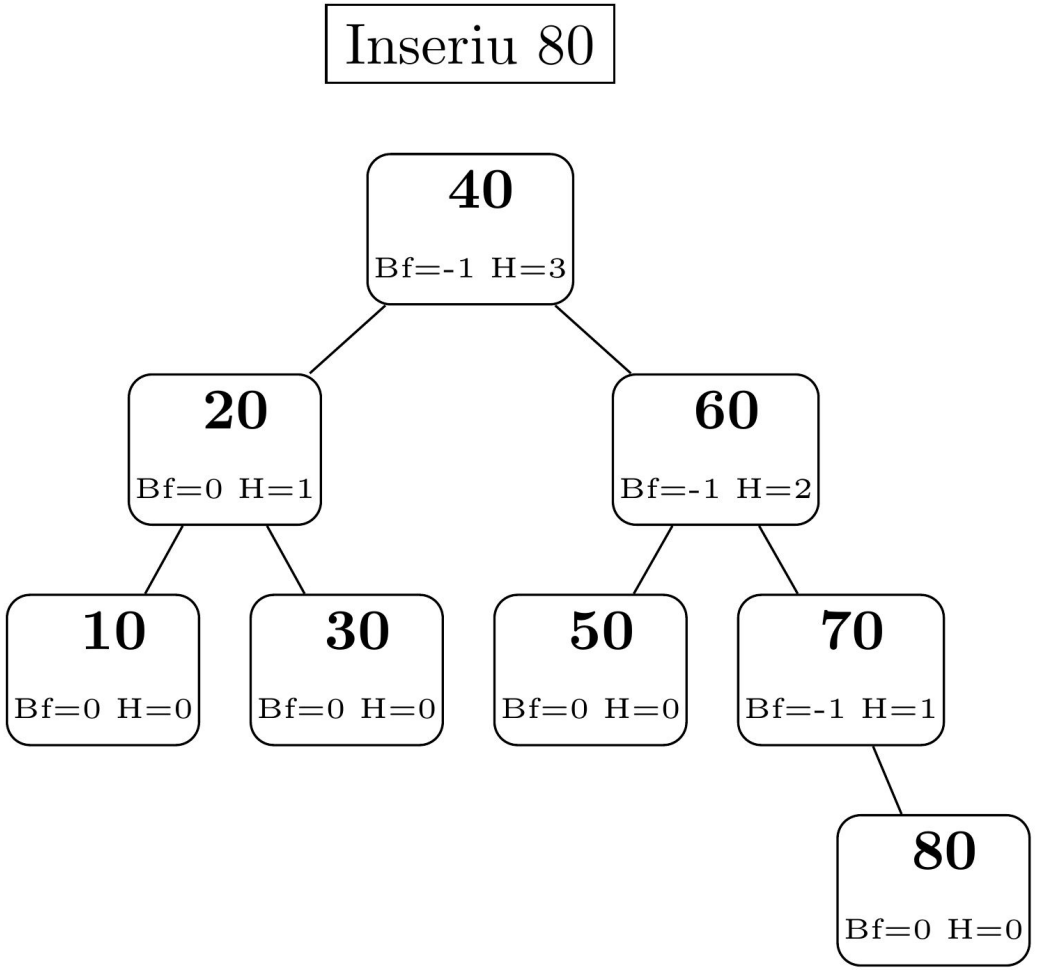
Inseriu 70:Antes de RR



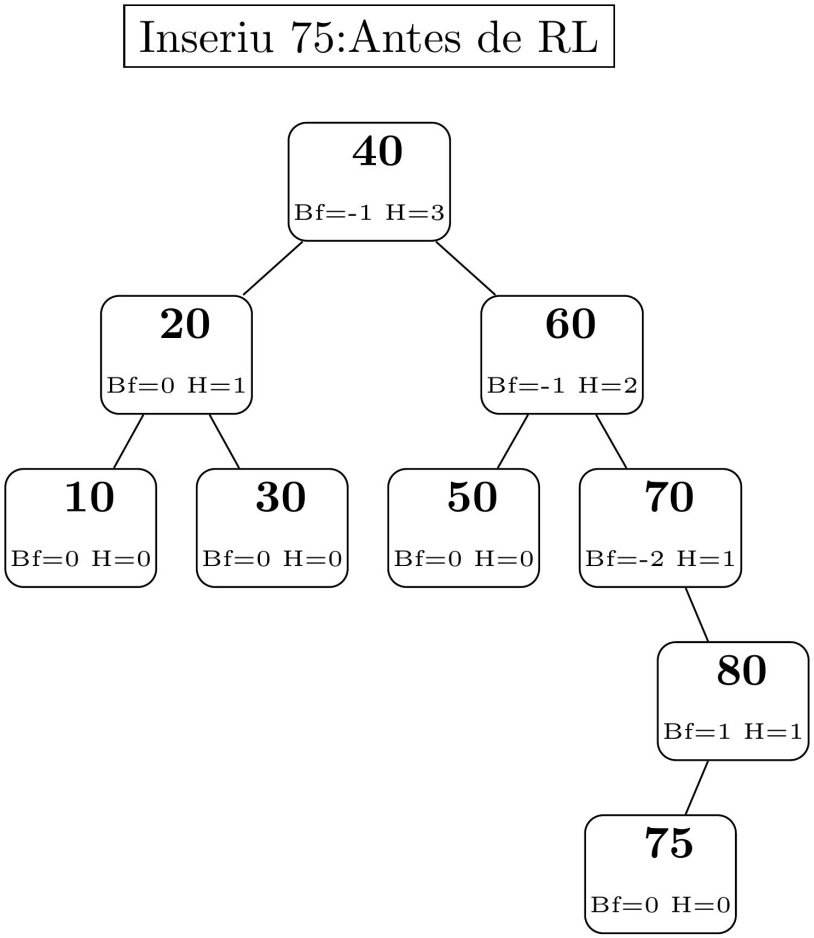
Inseriu 70

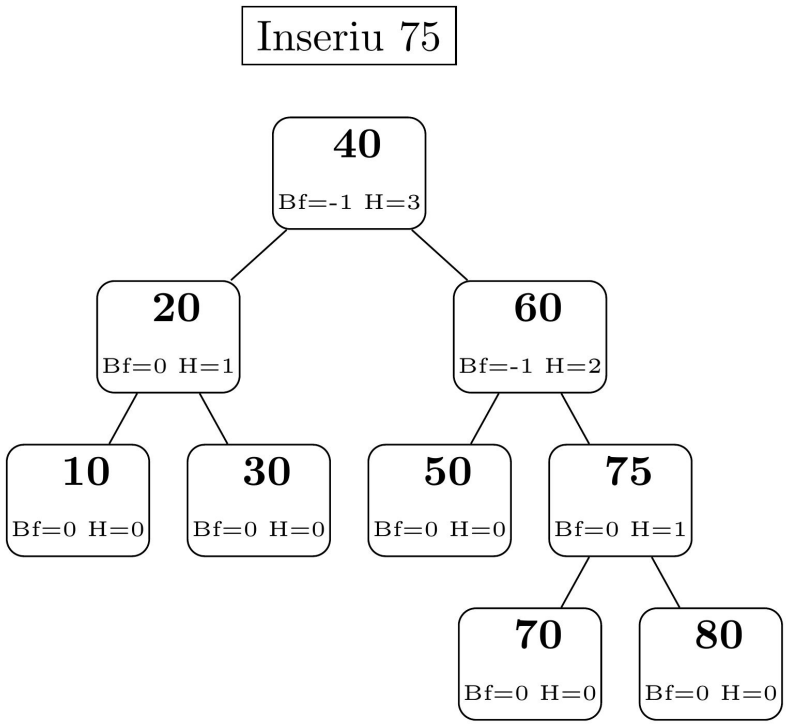


Inserção

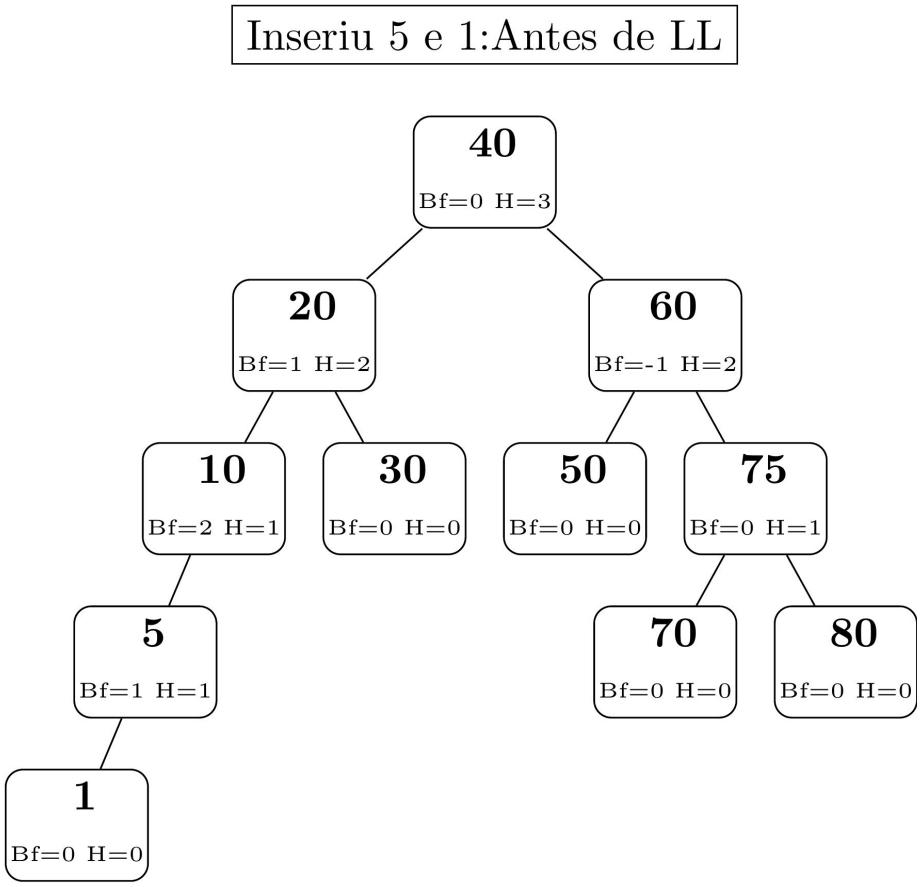

Inseriu 80
40
Bf=-1 H=3
20
Bf=0 H=1
60
Bf=-1 H=2
10
Bf=0 H=0
30
Bf=0 H=0
50
Bf=0 H=0
70
Bf=-1 H=1
80
Bf=0 H=0

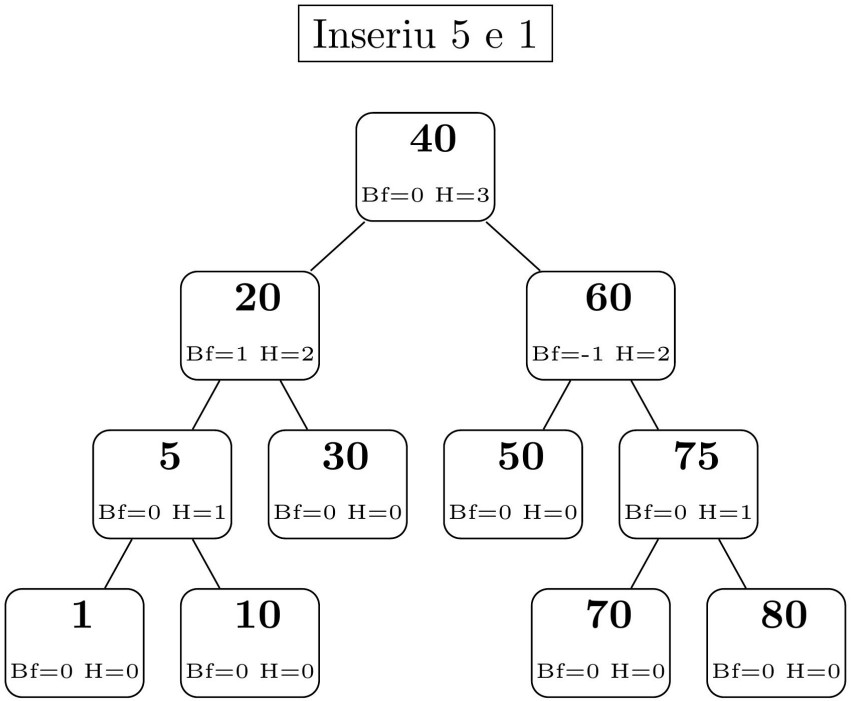
Inserção


Inseriu 75:Antes de RL
40
Bf=-1 H=3
20
Bf=0 H=1
60
Bf=-1 H=2
10
Bf=0 H=0
30
Bf=0 H=0
50
Bf=0 H=0
70
Bf=-2 H=1
80
Bf=1 H=1
75
Bf=0 H=0

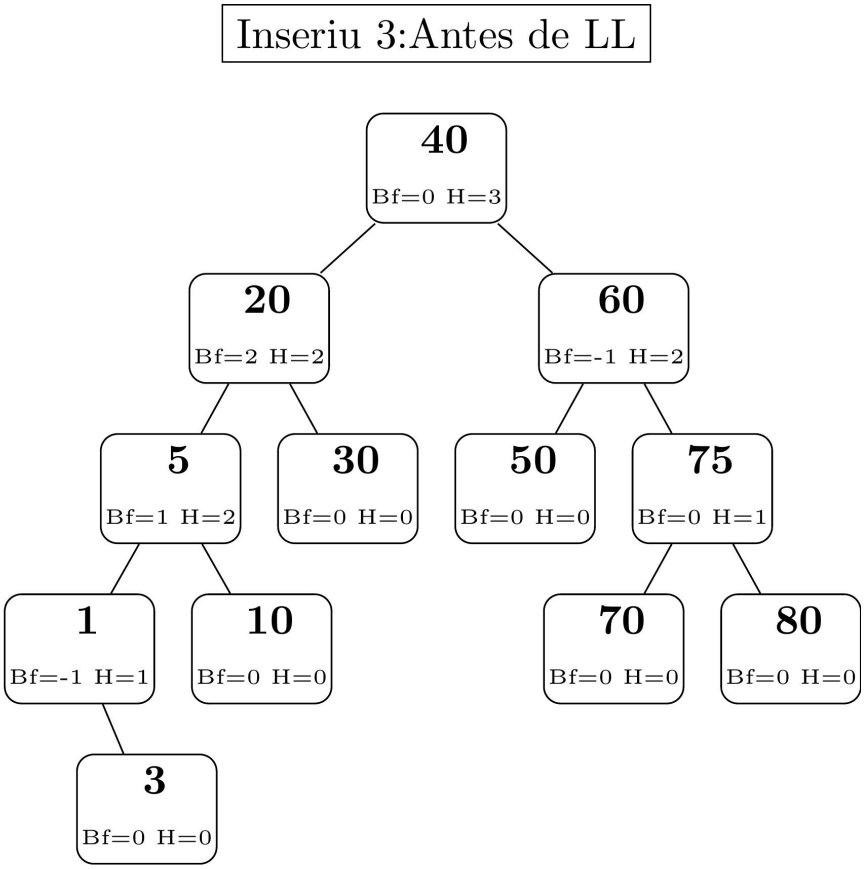

Inseriu 75
40
Bf=-1 H=3
20
Bf=0 H=1
60
Bf=-1 H=2
10
Bf=0 H=0
30
Bf=0 H=0
50
Bf=0 H=0
75
Bf=0 H=1
70
Bf=0 H=0
80
Bf=0 H=0

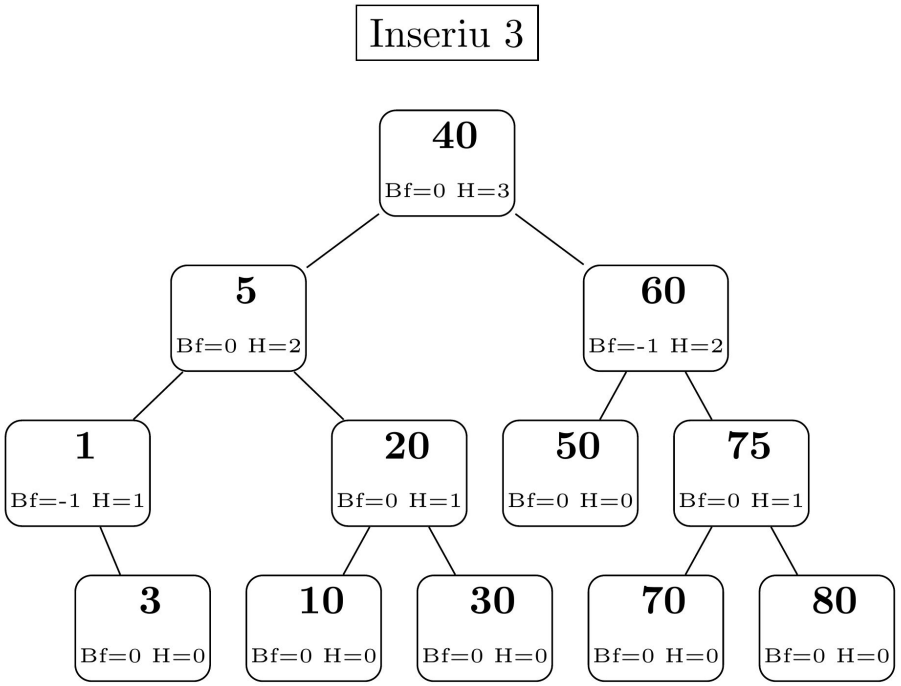
Inserção


Inseriu 5 e 1:Antes de LL
40
Bf=0 H=3
20
Bf=1 H=2
60
Bf=-1 H=2
10
Bf=2 H=1
30
Bf=0 H=0
50
Bf=0 H=0
75
Bf=0 H=1
5
Bf=1 H=1
70
Bf=0 H=0
80
Bf=0 H=0
1
Bf=0 H=0


Inseriu 5 e 1
40
Bf=0 H=3
20
Bf=1 H=2
60
Bf=-1 H=2
5
Bf=0 H=1
30
Bf=0 H=0
50
Bf=0 H=0
75
Bf=0 H=1
1
Bf=0 H=0
10
Bf=0 H=0
70
Bf=0 H=0
80
Bf=0 H=0

Inserção


Inseriu 3:Antes de LL
40
Bf=0 H=3
20
Bf=2 H=2
60
Bf=-1 H=2
5
Bf=1 H=2
30
Bf=0 H=0
50
Bf=0 H=0
75
Bf=0 H=1
1
Bf=-1 H=1
10
Bf=0 H=0
70
Bf=0 H=0
80
Bf=0 H=0
3
Bf=0 H=0


Inseriu 3
40
Bf=0 H=3
5
Bf=0 H=2
60
Bf=-1 H=2
1
Bf=-1 H=1
20
Bf=0 H=1
50
Bf=0 H=0
75
Bf=0 H=1
3
Bf=0 H=0
10
Bf=0 H=0
30
Bf=0 H=0
70
Bf=0 H=0
80
Bf=0 H=0

Passo a Passo

https://visualgo.net/en/bst

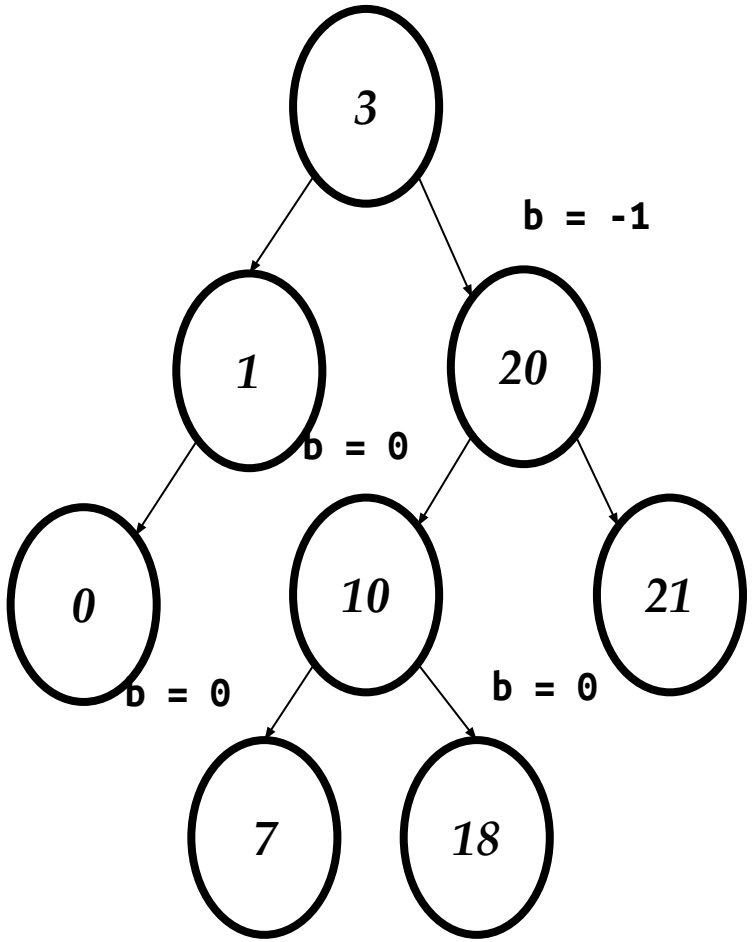
ÁRVORE AVL - REMOÇÃO

Remoção

- Lembrando de árvores binárias de pesquisa
- Existem 3 casos de remoção
  - Nó folha
  - Nó tem 1 filho
  - Nó tem 2 filhos
- Na AVL: ao remover um nó temos que corrigir a árvore

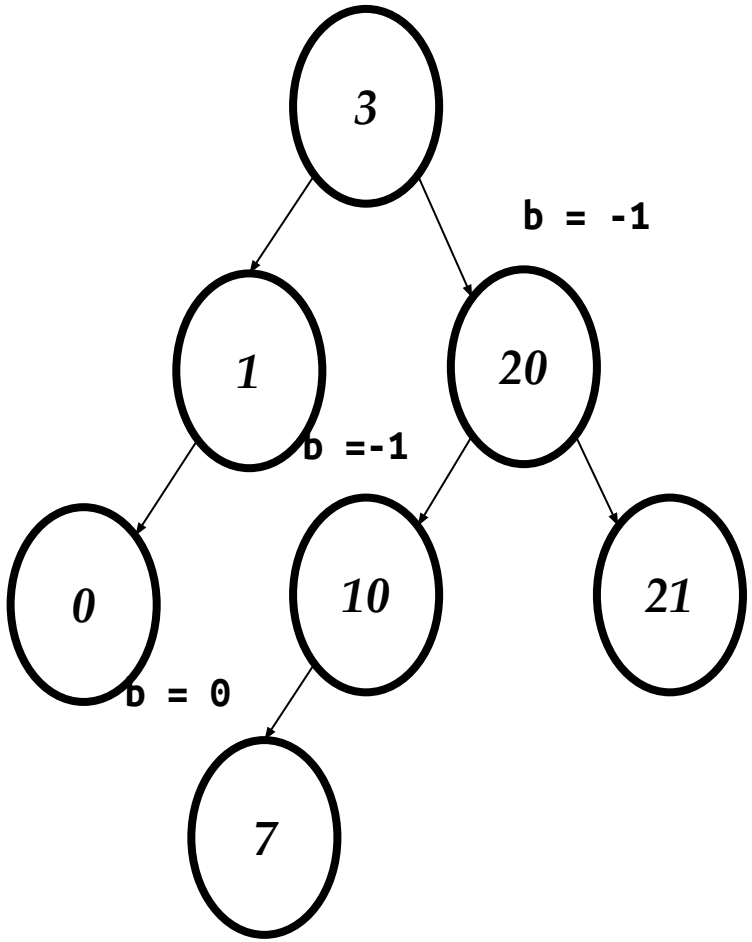
Remoção – Nó folha

- Retira o 18



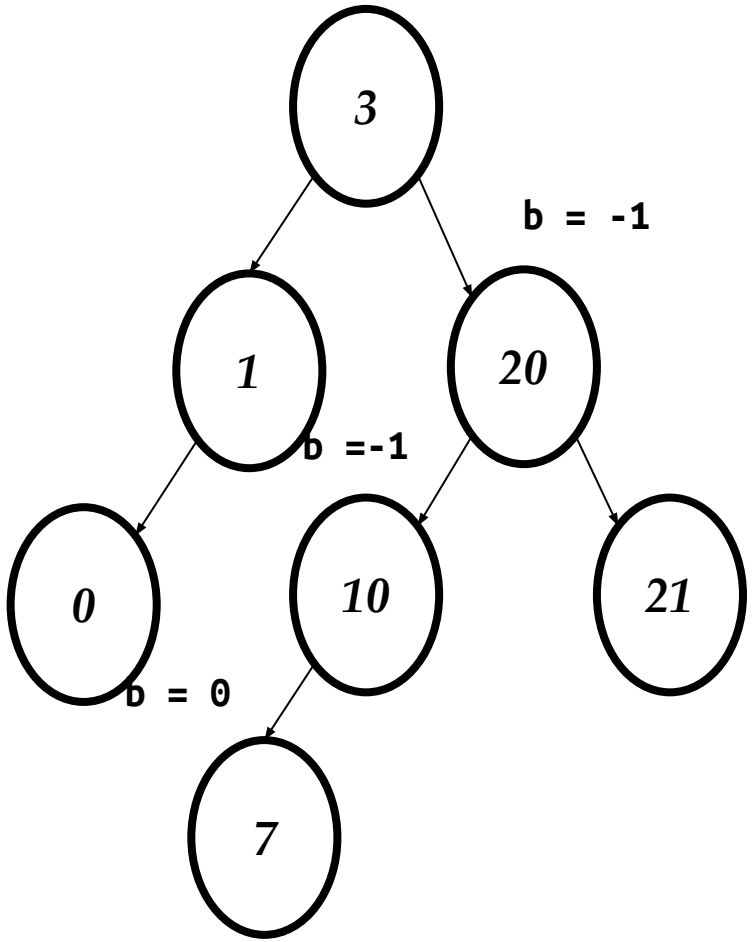
Remoção – Nó folha

- Retira o 18
- Tudo OK



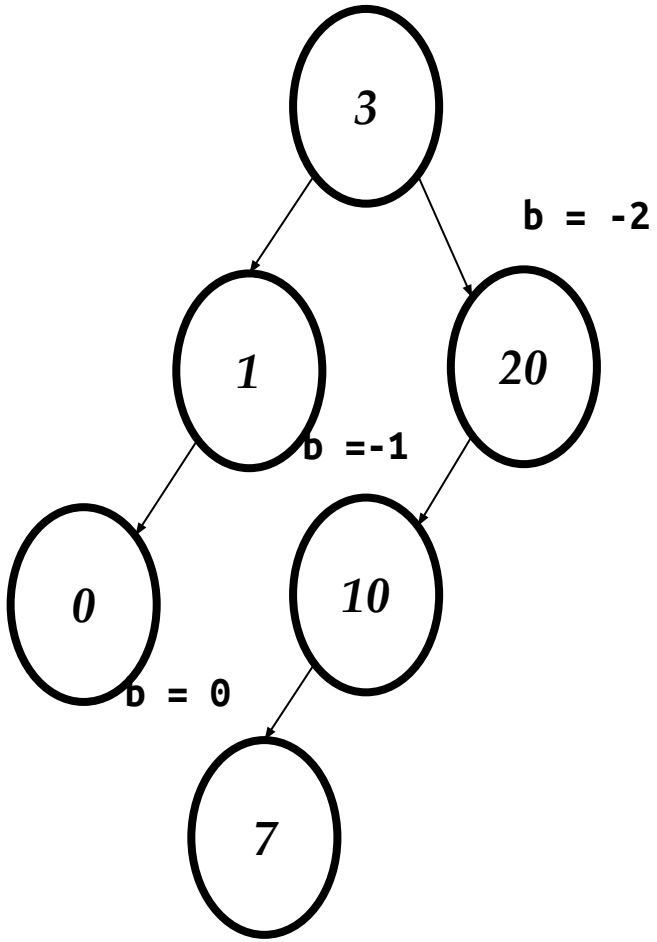
Remoção – Nó folha

- Retira o 21



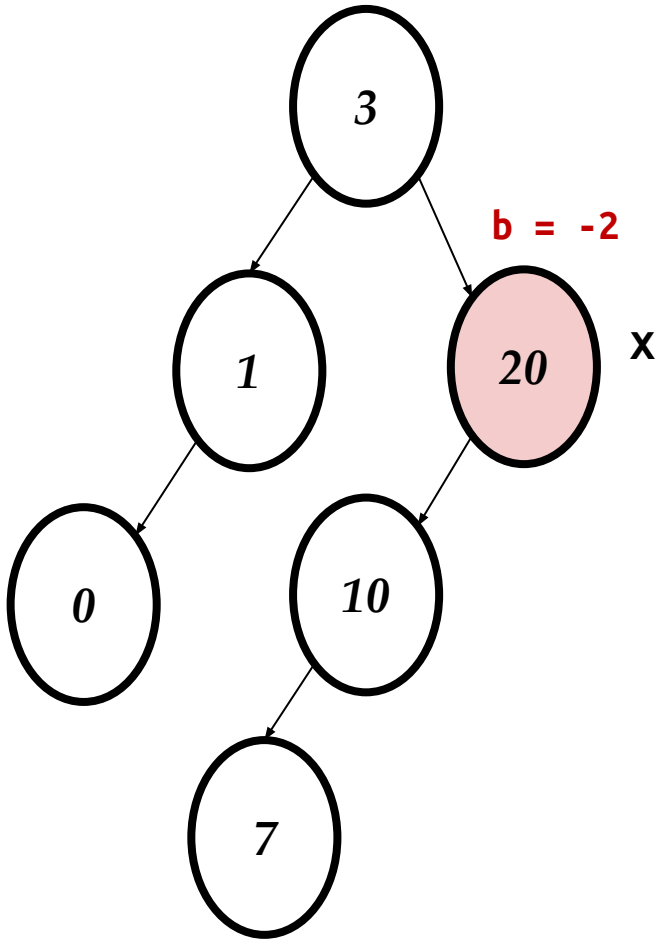
Remoção – Nó folha

- Retira o 21
- Olhamos para o balanceamento do pai
  - Caso esteja ok, continue subindo



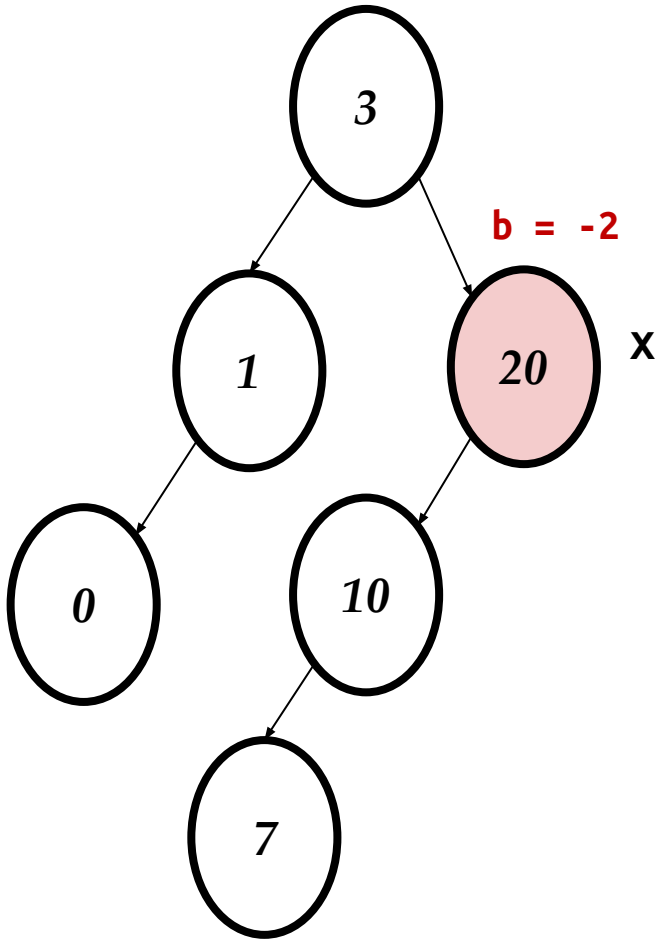
Remoção – Nó folha

- Achamos caso problema
- Vamos chamar tal nó de **X**



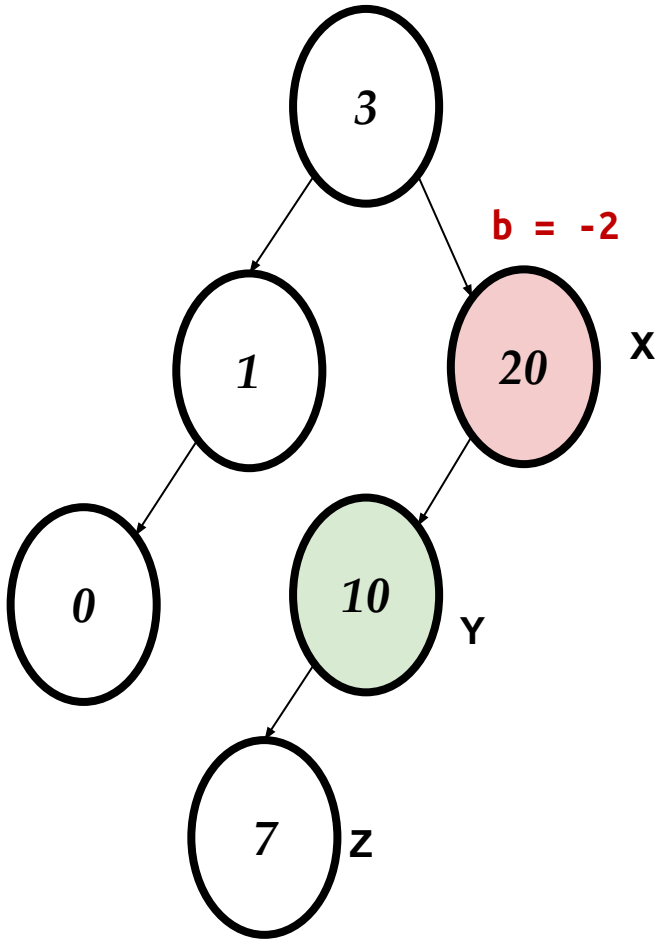
Remoção – Nó folha

- Achamos caso problema
- Vamos chamar tal nó de **X**
- Identificamos um filho de tal nó com maior altura
  - Chamamos de **Y**



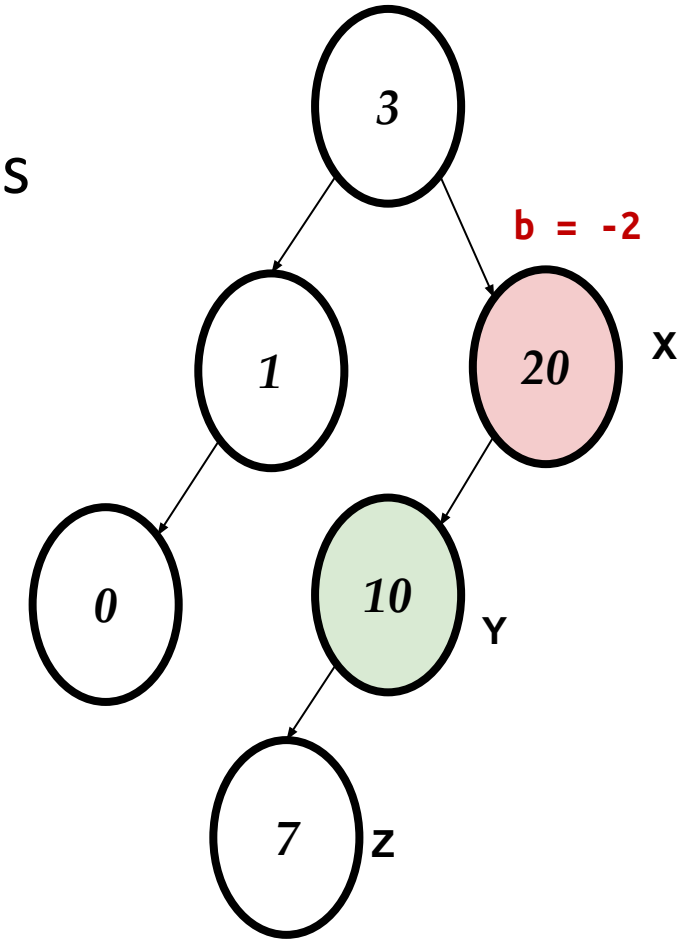
Remoção – Nó folha

- Achamos caso problema
- Vamos chamar tal nó de **X**
- Identificamos um filho de tal nó com maior altura
  - Chamamos de **Y**
- Achamos o neto de maior altura
  - Chamamos de **Z**



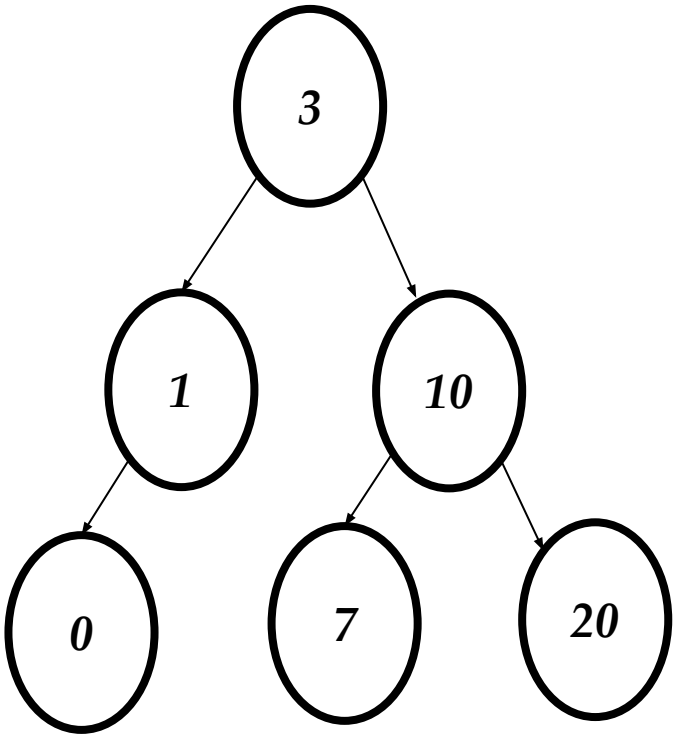
Remoção – Nó folha

- Olhando para **X, Y e Z**
  - Caímos em um dos casos de desbalanceamento
  - Caso 1



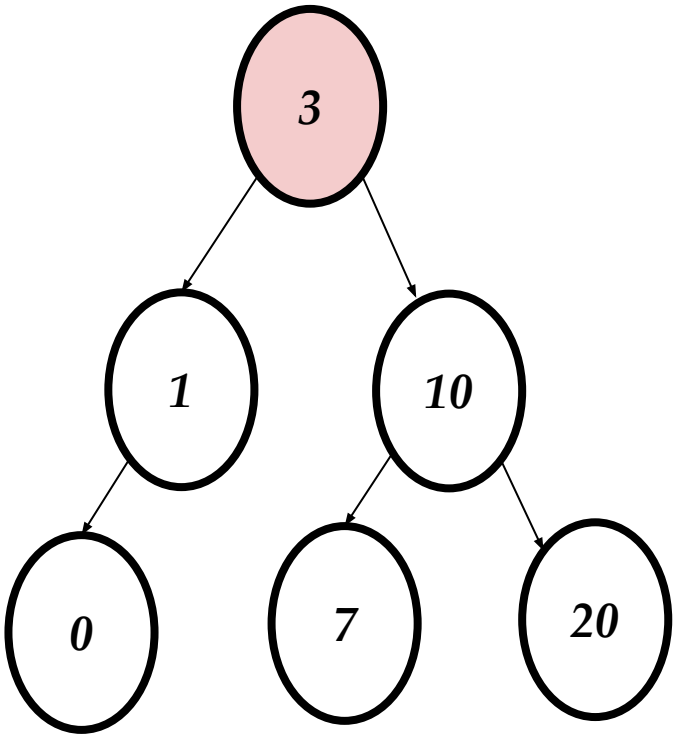
Remoção

- Olhando para **X, Y e Z**
  - Caímos em um dos casos de desbalanceamento
  - Caso 1



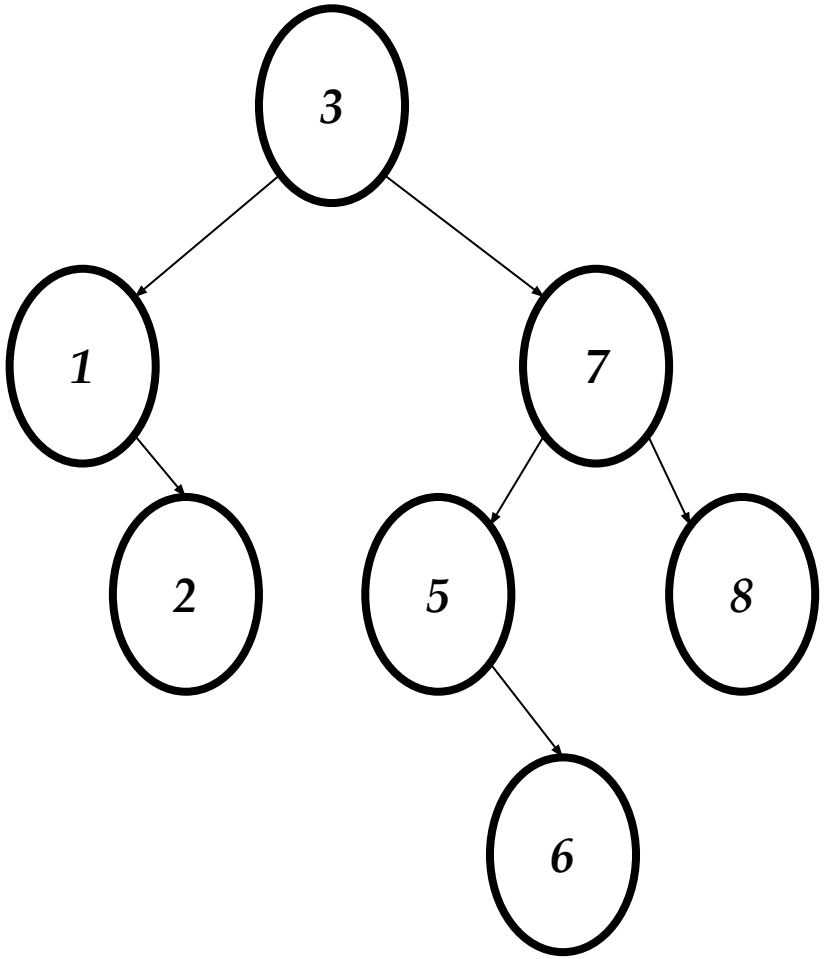
Remoção

- Olhando para **X, Y e Z**
  - Caímos em um dos casos de desbalanceamento
  - Caso 1
- Repetimos tudo olhando para o avô
  - Nó 3
  - Neste caso tudo OK



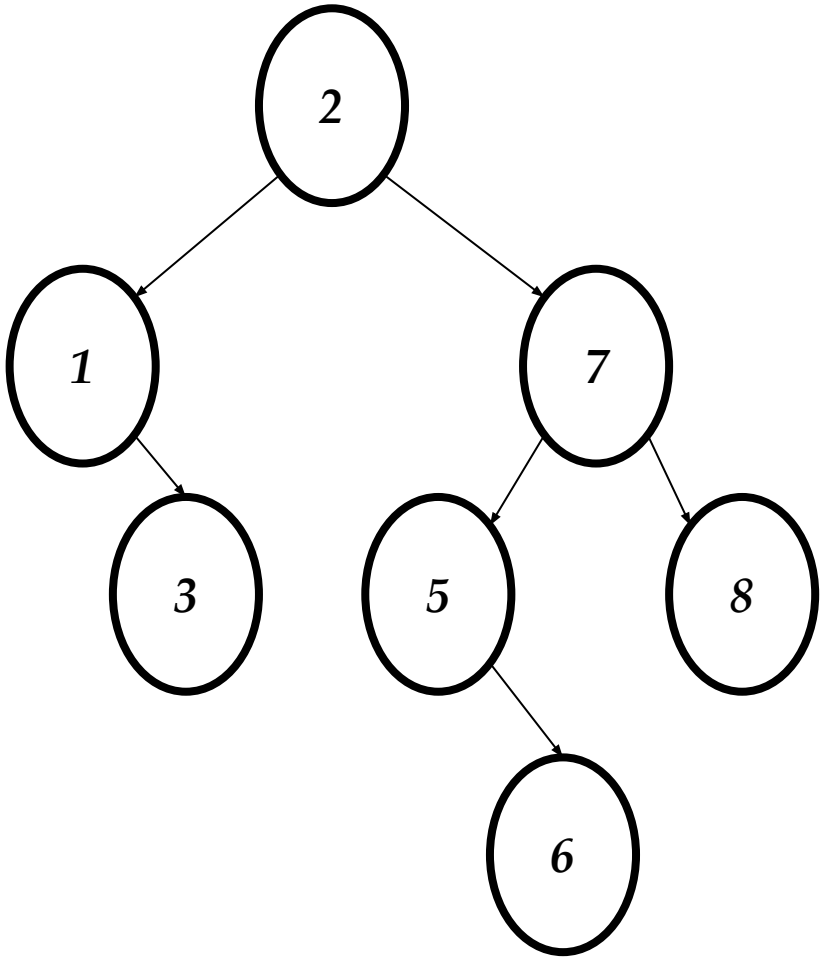
Caso mais Complicado – Nó tem 2 filhos

- Remover o nó 3
- Lembrando de árvores binárias de pesquisa, trocamos com o antecessor (nó 2)



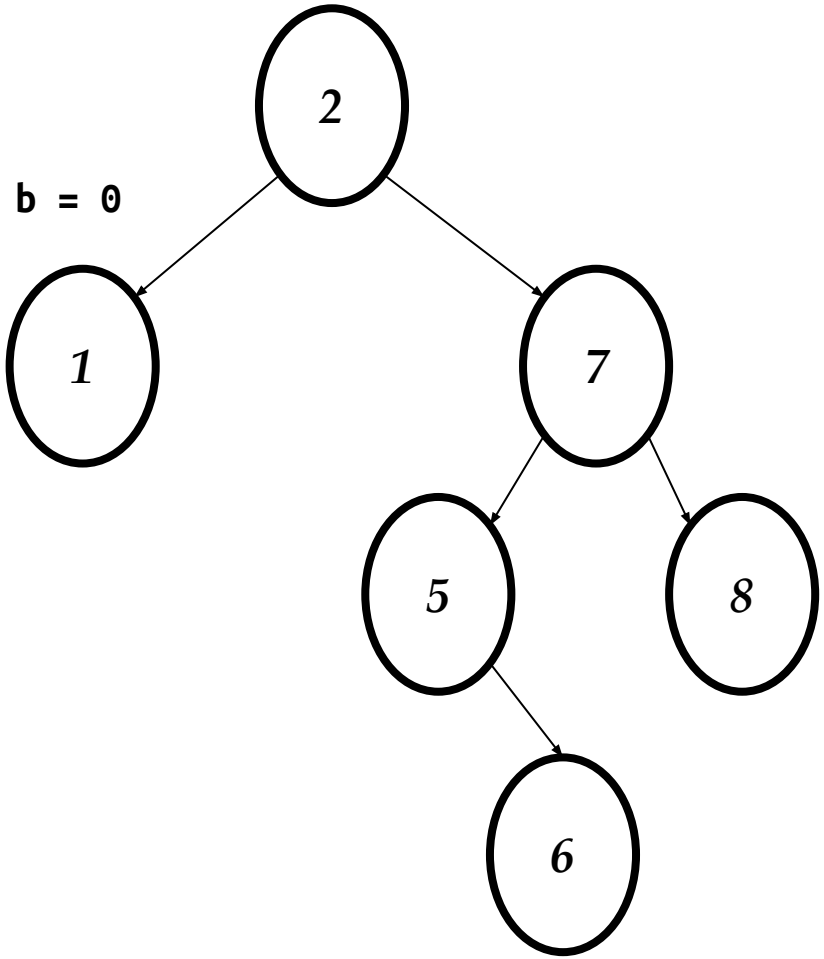
Caso mais Complicado

- Remover o nó 3
- Trocamos com o nó 2
- A remoção do mesmo ocorre em um nó folha



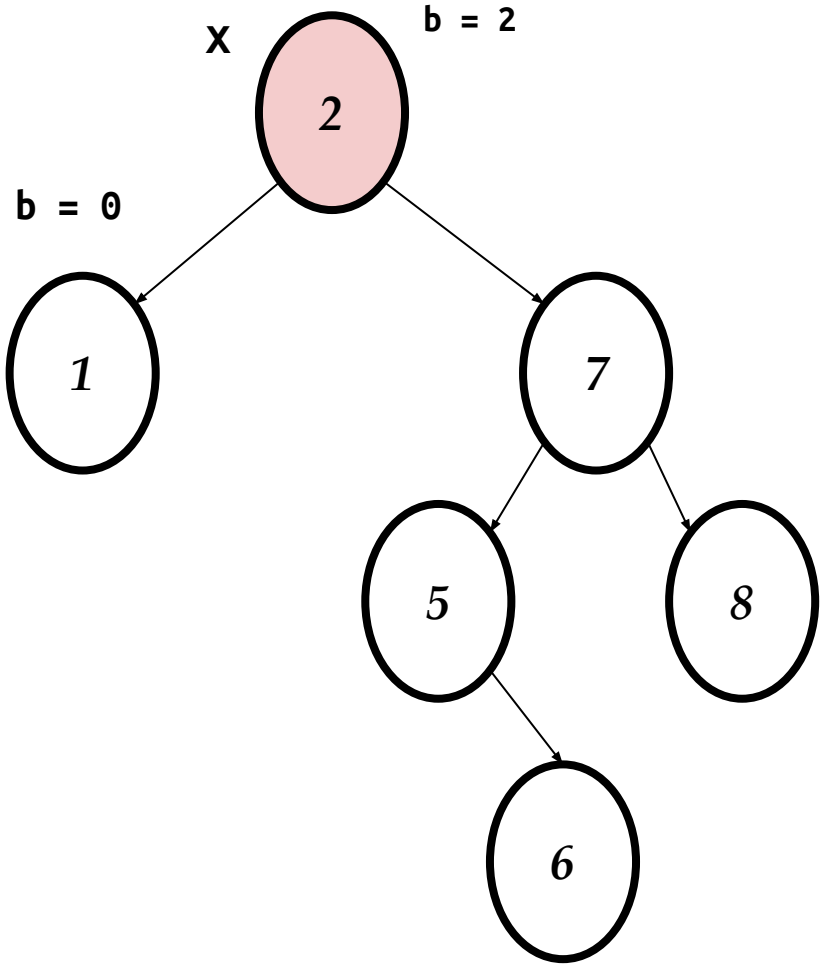
Caso mais Complicado

- Removemos o nó 3 (agora folha)
- O nó 1 continua ok, vamos olhar para cima



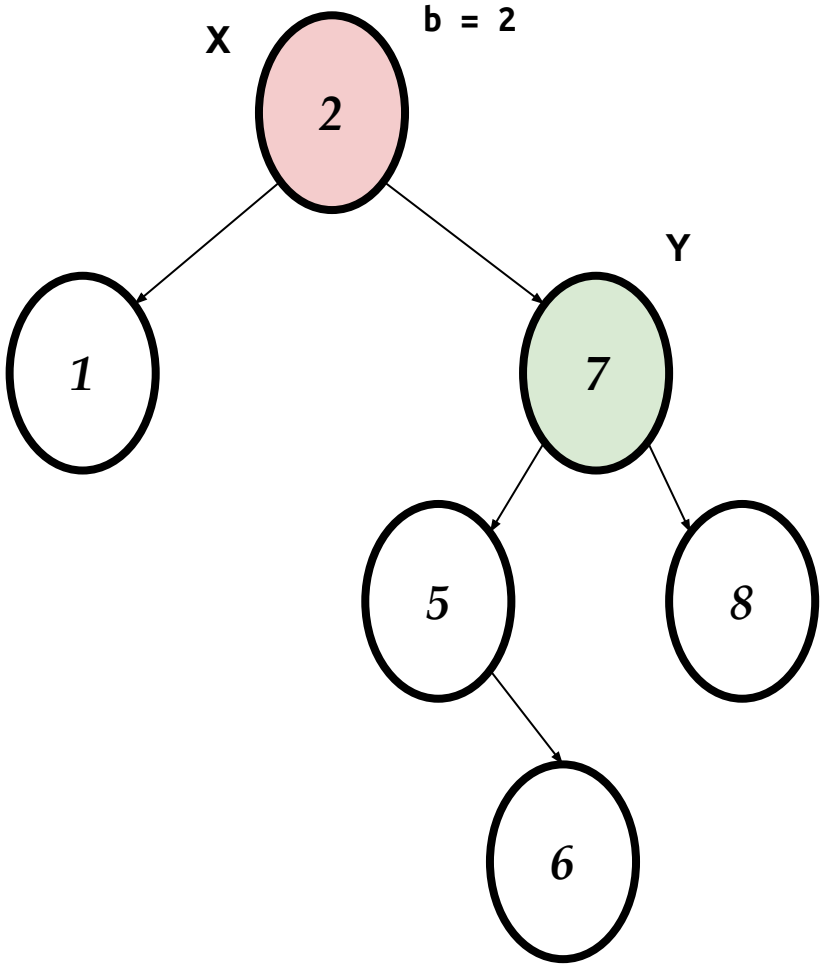
Caso mais Complicado

- Removemos o nó 3 (agora folha)
- O nó 1 continua ok, vamos olhar para cima
- O nó 2 não está ok
- Chamamos de nó X



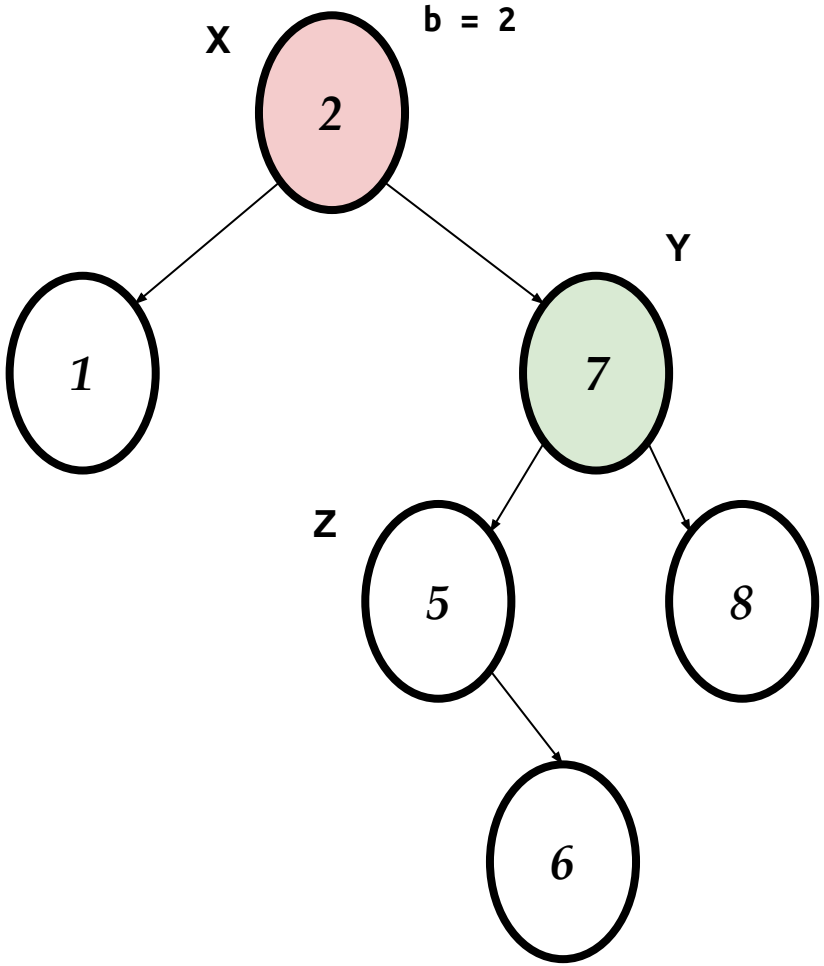
Caso mais Complicado

- O nó 2 é X
- O filho de maior altura é **Y**



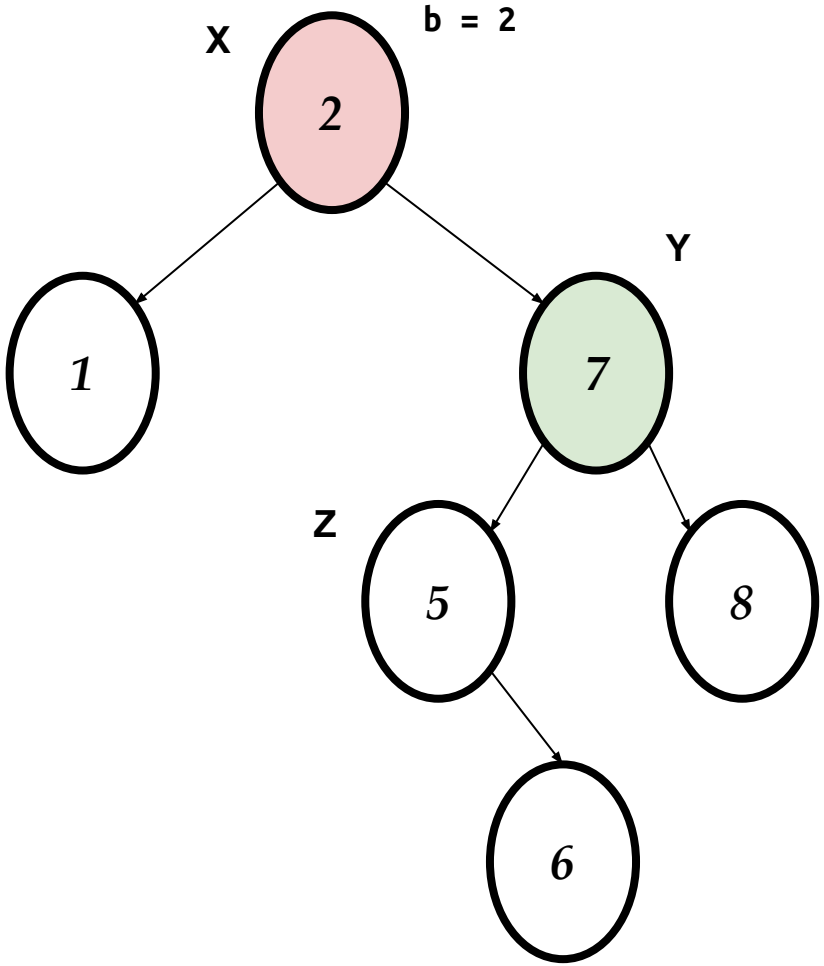
Caso mais Complicado

- O nó 2 é X
- O filho de maior altura é **Y**
- O neto de maior altura é **Z**



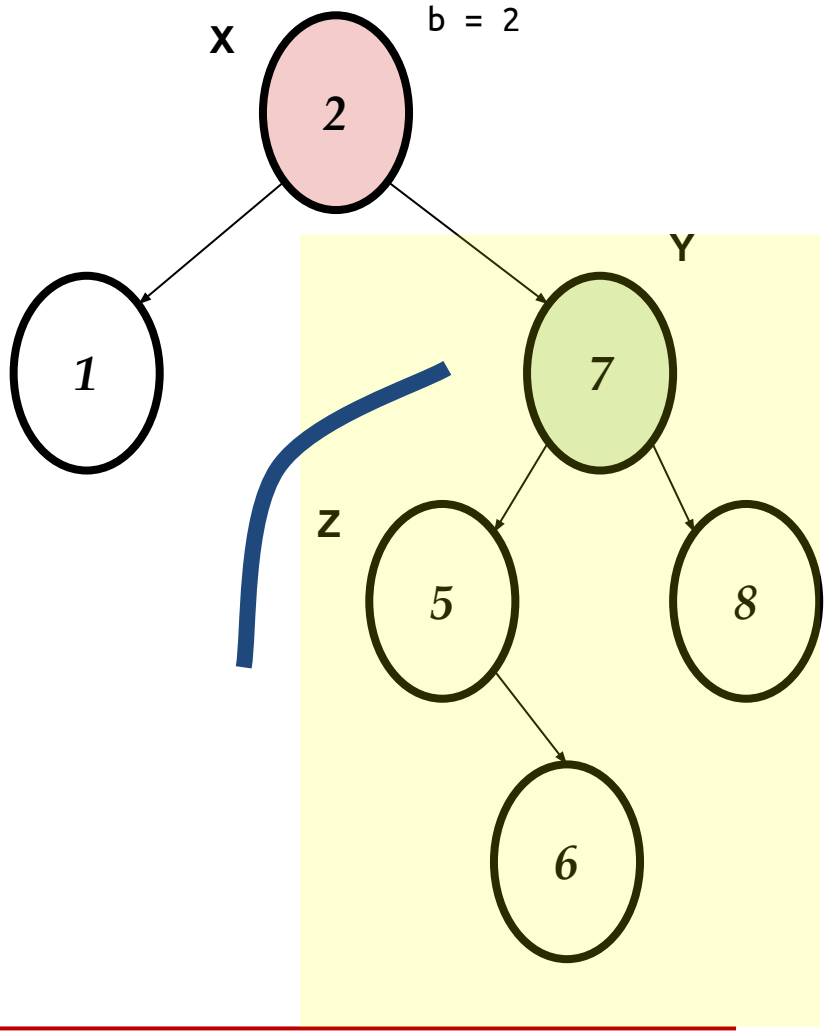
Caso mais Complicado

- Estamos no Caso 4



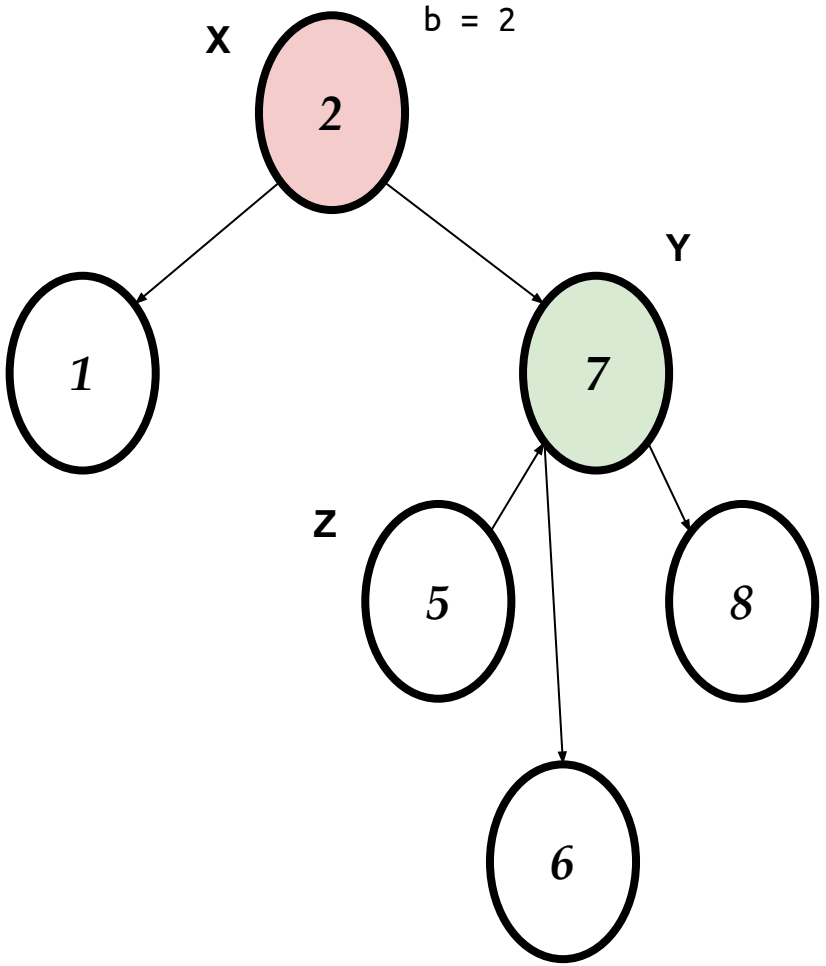
Caso mais Complicado

- Estamos no Caso 4
- Rotação para **direita**



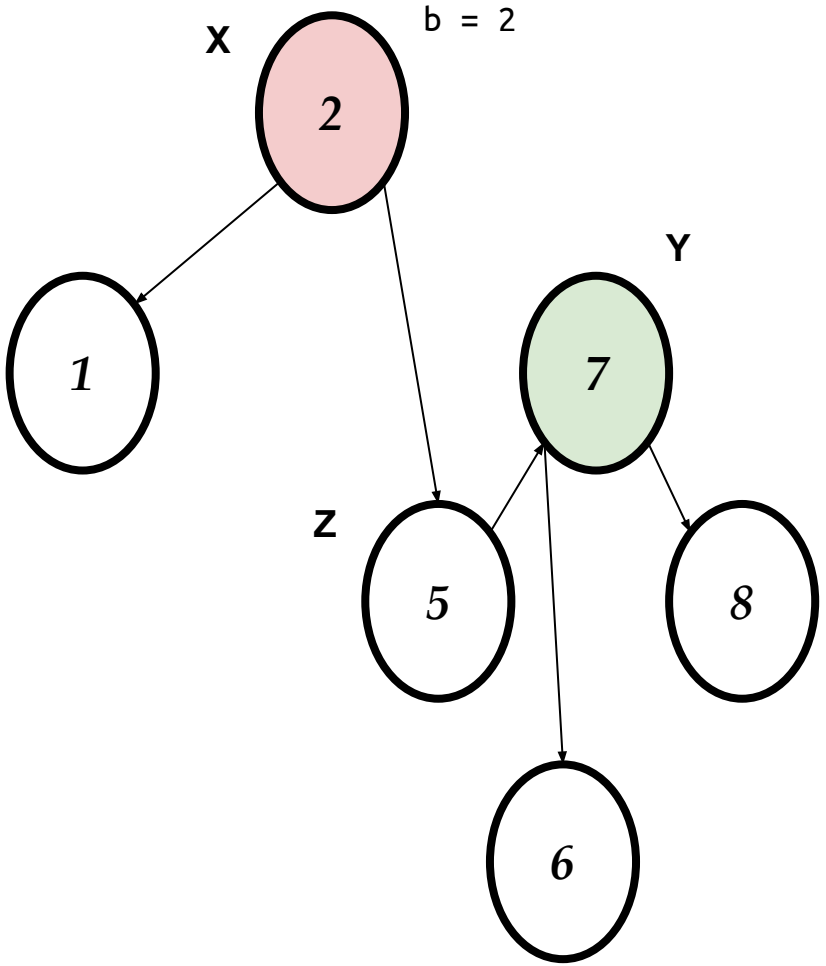
Caso mais Complicado

- Estamos no Caso 4
- Rotação para **direita**



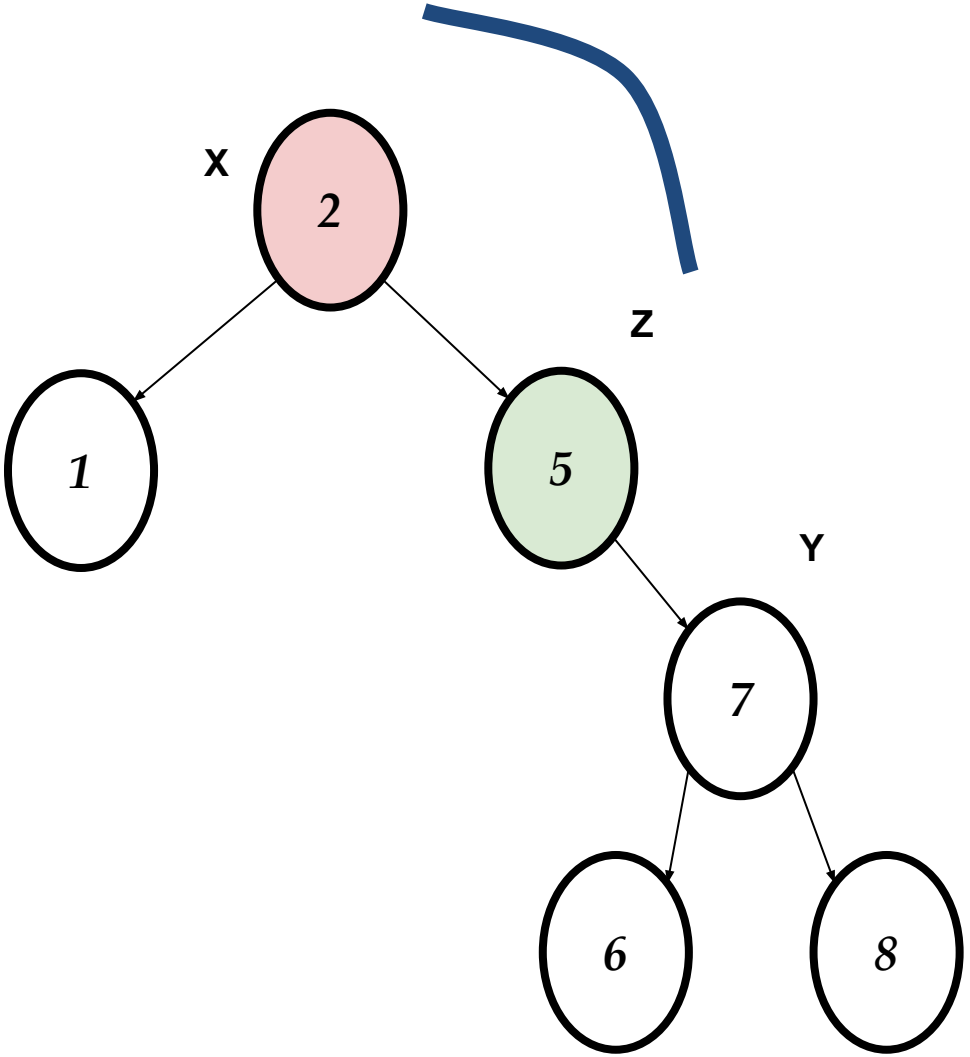
Caso mais Complicado

- Estamos no Caso 4
- Rotação para **direita**



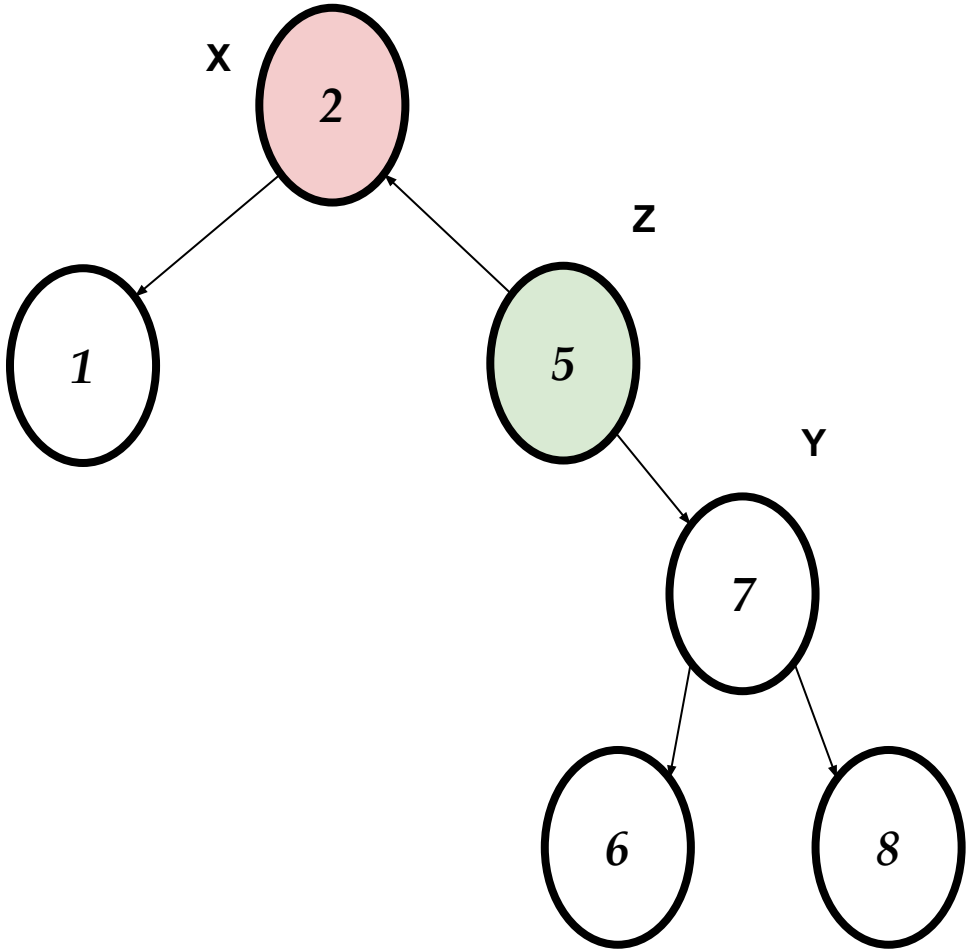
Caso mais Complicado

- Estamos no Caso 4
- Rotação para direita
- **Agora para esquerda**



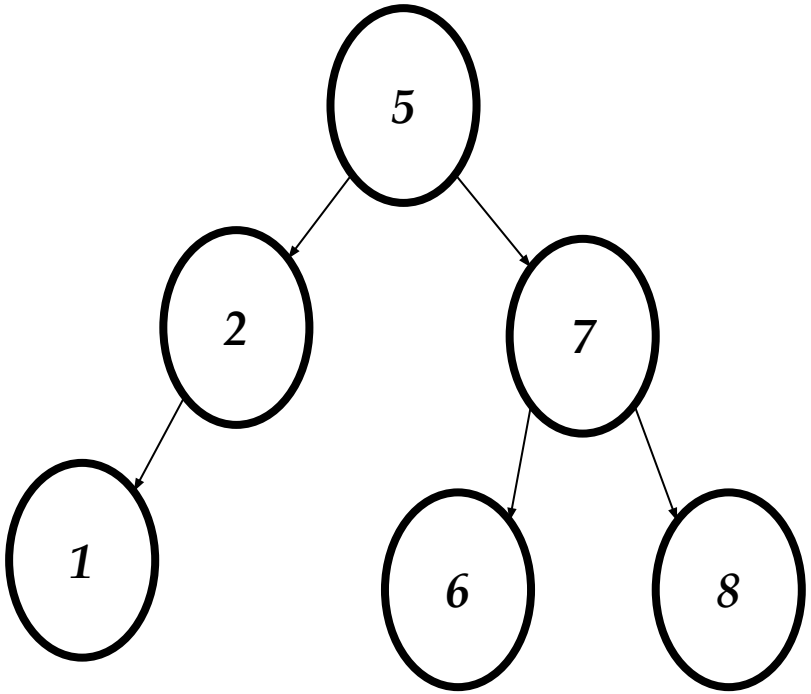
Caso mais Complicado

- Estamos no Caso 4
- Rotação para direita
- Agora para **esquerda**



Caso mais Complicado

- Estamos no Caso 4
- Rotação para direita
- Agora para esquerda



Intuição para Remoção

1. Achar um nó desbalanceado
   a. **X**
2. Achar a subárvore com maior altura
   a. Raiz dessa subárvore é **Y**
   b. É intuitivo perceber que é esta subárvore que causa o desbalanceamento
      i. balance = altura(esq)-altura(dir). A maior das 2 faz o valor <> -1, 1, 0
3. Repetir a mesma coisa a partir de **Y**
   a. Achamos o nó **Z**
4. Corrigir com os casos de desbalanceamento (inserção)

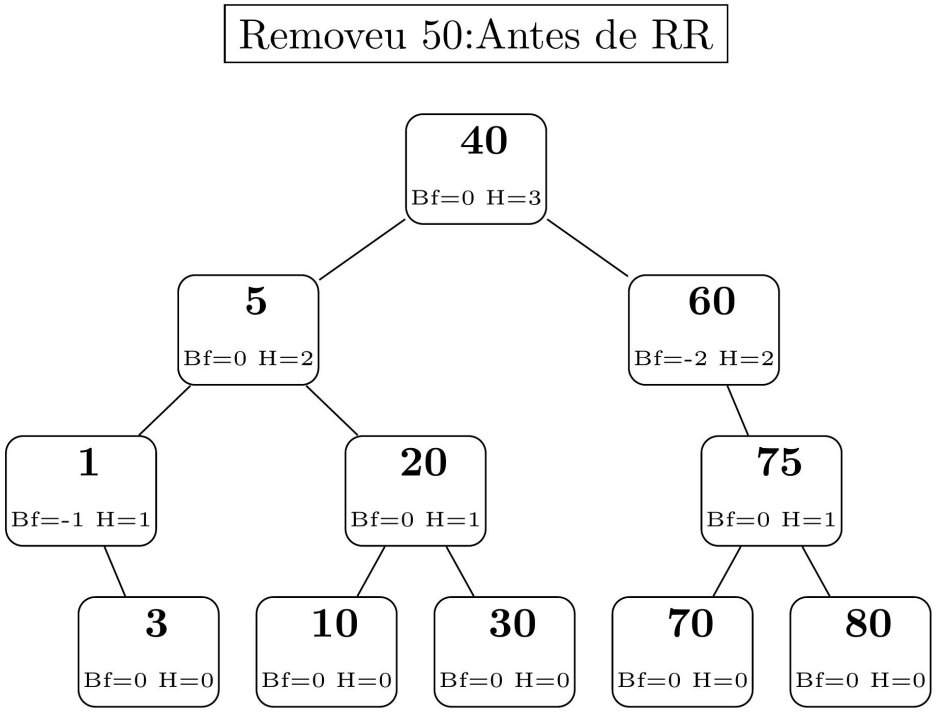
Em alguns casos temos que continuar corrigindo

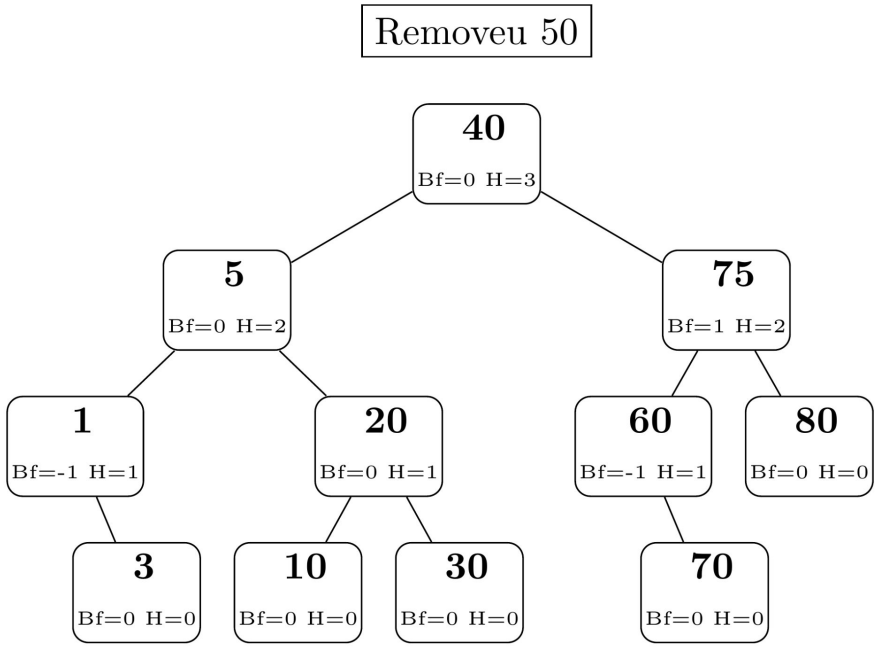
- Enquanto algum nó está desbalanceado
  - Anda para cima
  - Verifica
  - Corrige

Remoção - Código

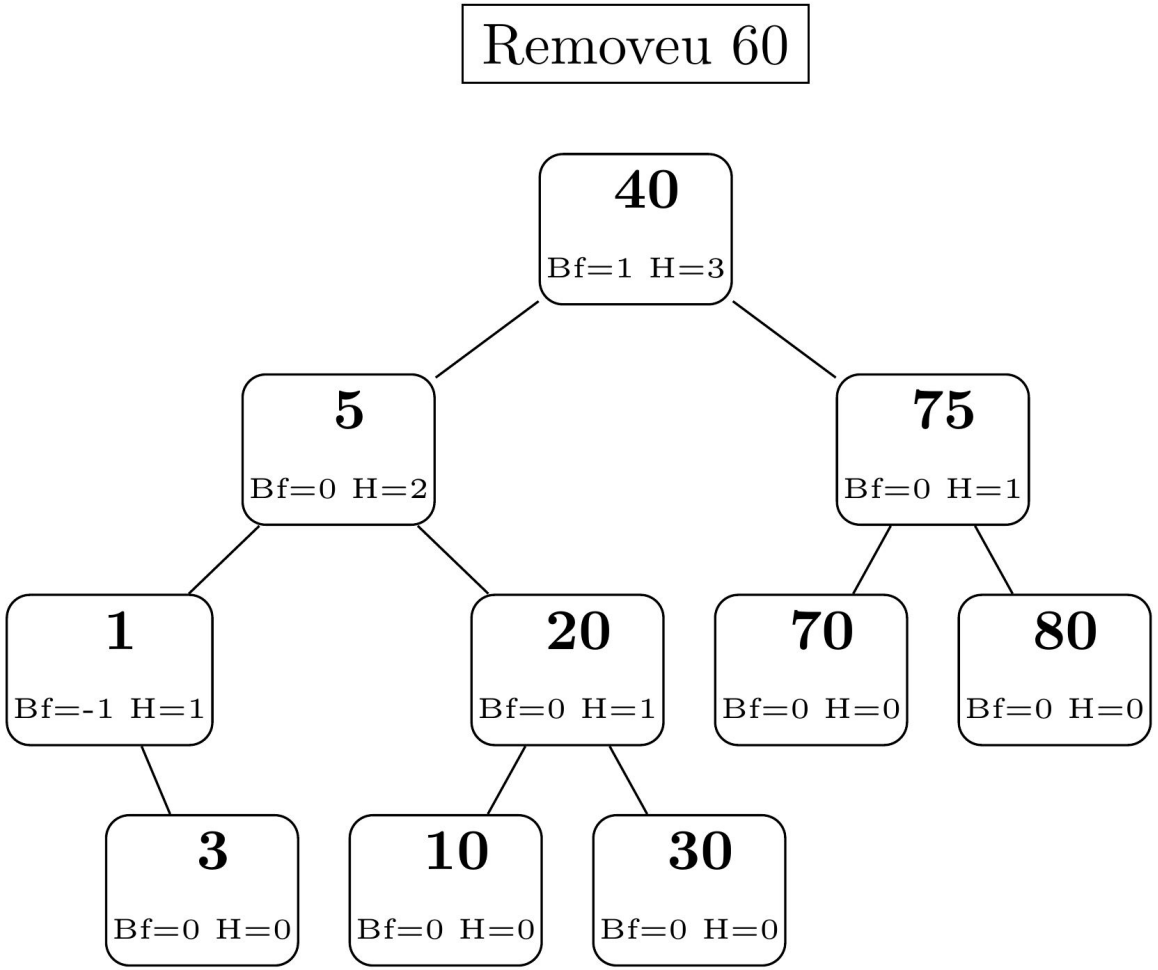
```
node * Delete(node * T, int x) {
  node * p;
  if (T == NULL) { return NULL;
  } else
  if (x > T -> data) { T -> right = Delete(T -> right, x);
        if (BF(T) == 2) // rebalancear
              if (BF(T -> left) >= 0) T = LL(T); else T = LR(T);
  } else {
  if (x < T -> data) { T -> left = Delete(T -> left, x);
        if (BF(T) == -2) // rebalancear
              if (BF(T -> right) <= 0) T = RR(T); else T = RL(T);
  } else {
        if (T -> right != NULL) { //remove sucessor
              p = T -> right; while (p -> left != NULL) p = p -> left;
              T -> data = p -> data;
              T -> right = Delete(T -> right, p -> data);
              if (BF(T) == 2) // rebalancear
                    if (BF(T -> left) >= 0) T = LL(T); else T = LR(T);
        } else return (T -> left);
  }
  T -> ht = height(T);
  return (T);
}
```

Remoção

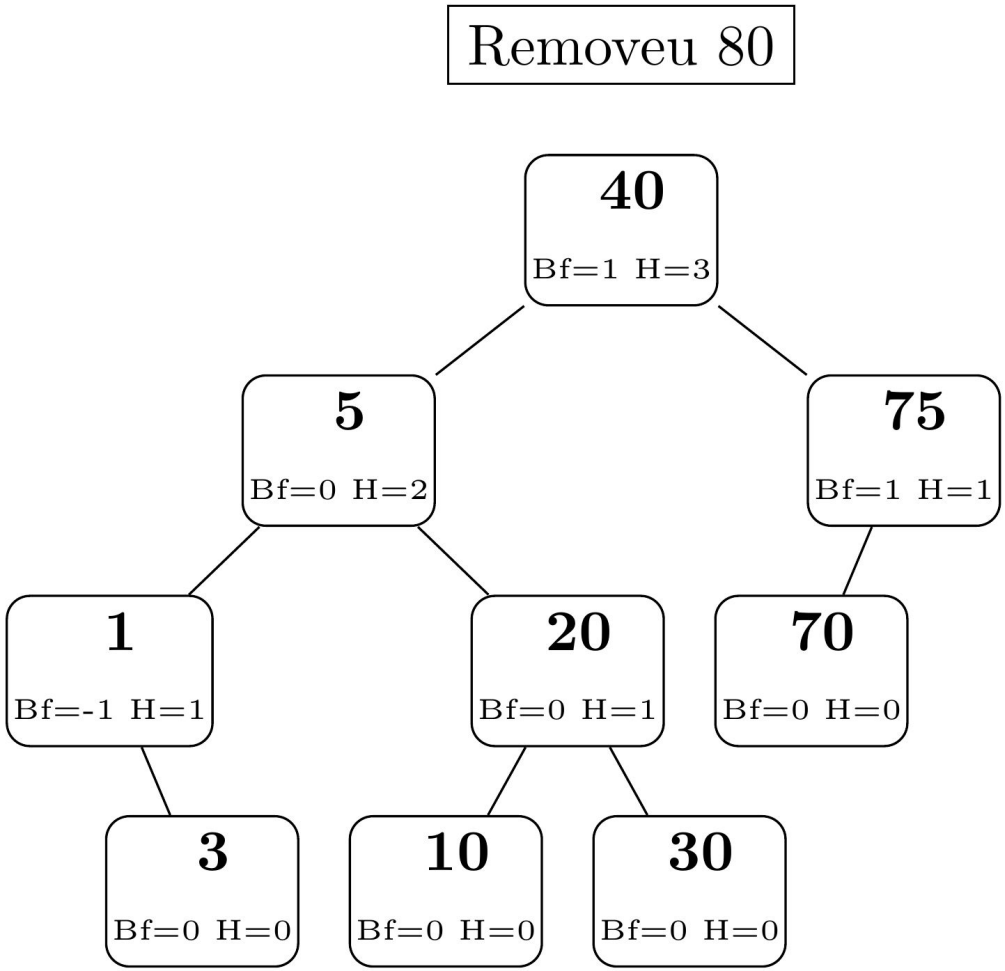

Removeu 50:Antes de RR
40
Bf=0 H=3
5
Bf=0 H=2
60
Bf=-2 H=2
1
Bf=-1 H=1
20
Bf=0 H=1
75
Bf=0 H=1
3
Bf=0 H=0
10
Bf=0 H=0
30
Bf=0 H=0
70
Bf=0 H=0
80
Bf=0 H=0


Removeu 50
40
Bf=0 H=3
5
Bf=0 H=2
75
Bf=1 H=2
1
Bf=-1 H=1
20
Bf=0 H=1
60
Bf=-1 H=1
80
Bf=0 H=0
3
Bf=0 H=0
10
Bf=0 H=0
30
Bf=0 H=0
70
Bf=0 H=0

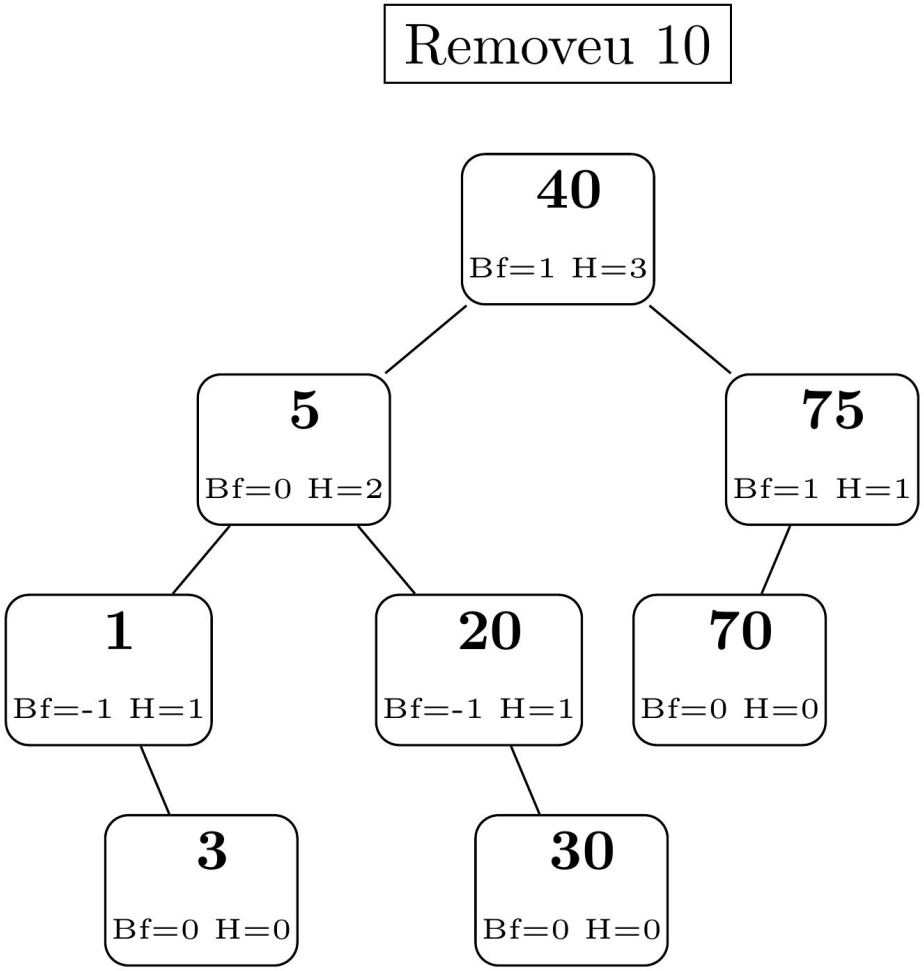
Remoção


Removeu 60
40
Bf=1 H=3
5
Bf=0 H=2
75
Bf=0 H=1
1
Bf=-1 H=1
20
Bf=0 H=1
70
Bf=0 H=0
80
Bf=0 H=0
3
Bf=0 H=0
10
Bf=0 H=0
30
Bf=0 H=0

Remoção

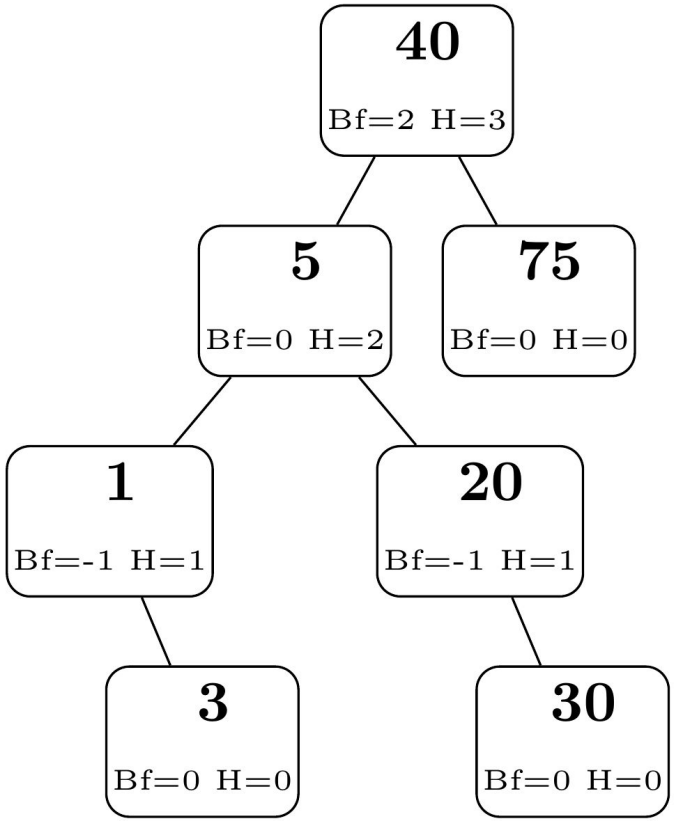

Removeu 80
40
Bf=1 H=3
5
Bf=0 H=2
75
Bf=1 H=1
1
Bf=-1 H=1
20
Bf=0 H=1
70
Bf=0 H=0
3
Bf=0 H=0
10
Bf=0 H=0
30
Bf=0 H=0

Remoção

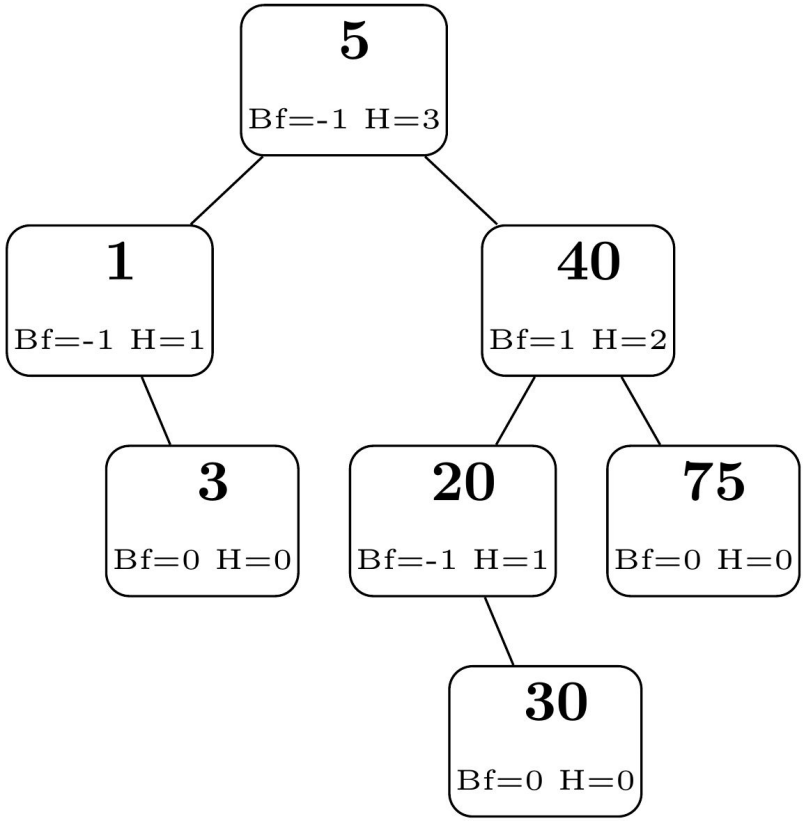

Removeu 10
40
Bf=1 H=3
5
Bf=0 H=2
75
Bf=1 H=1
1
Bf=-1 H=1
20
Bf=-1 H=1
70
Bf=0 H=0
3
Bf=0 H=0
30
Bf=0 H=0

Remoção

Removeu 70:Antes de LL



Removeu 70



Análise da Remoção - Custos

- Achar o antecessor (ou sucessor) é $O(\log(n))$
- Realizar a retirada $O(\log(n)) + O(1)$
- Verificar se tem desbalanceamento $O(\log(n))$
  - Para cada desbalanceamento – corrigir $O(1) + O(1)$

- **Custo Total**

  $O(\log(n)) + O(\log(n) \times O(1))$ **= $O(\log(n))$**