

Árvores Binárias IV

Joaquim Madeira

20/05/2021

Ficheiro ZIP

- Está disponível no Moodle um **ficheiro ZIP** de suporte aos tópicos de hoje
- O tipo abstrato **Árvore AVL**
- **Funções incompletas**, que permitem trabalho autónomo de desenvolvimento e teste

Sumário

- Recap
- Árvores equilibradas em altura – **Balanced Trees**
- Árvores de Fibonacci
- Árvores de Adelson-Velskii e Landis (AVL)
- Operações de rotação para manutenção da condição de equilíbrio
- Algoritmo para **adição** de um elemento
- Algoritmo para **remoção** de um elemento – Breve referência
- Análise do desempenho: ABPs **vs** AVLs

Let's
RECAP

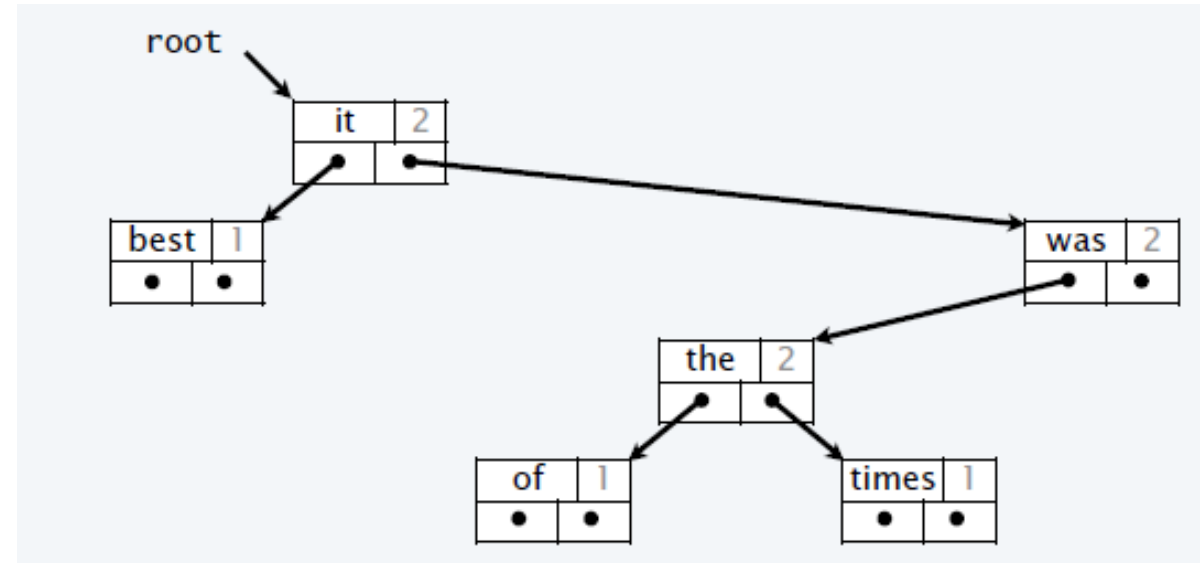
Recapitulação

TAD **Árvore Binária de Procura**

- Conjunto de **elementos** do **mesmo tipo**
- Armazenados **em-ordem**
- Procura / inserção / remoção / substituição
- Pertença
- **search() / insert() / remove() / replace()**
- **size() / isEmpty() / contains()**
- **create() / destroy()**

Critério de ordem – Definição recursiva

- Para cada nó, os elementos da sua **subárvore esquerda** são **inferiores** ao nó
- E os elementos da sua **subárvore direita** são **superiores** ao nó
- **Não** há elementos **repetidos** !!
- A organização da árvore depende da **sequência de inserção** dos elementos



[Sedgewick & Wayne]

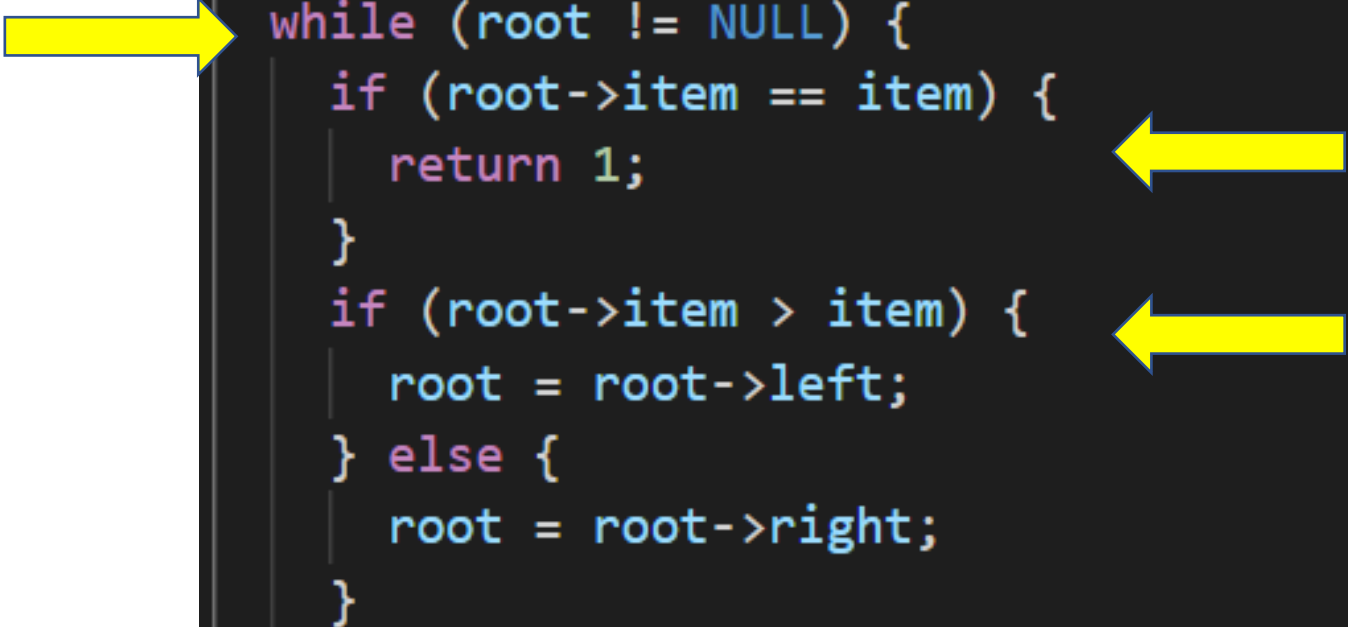
getMin()



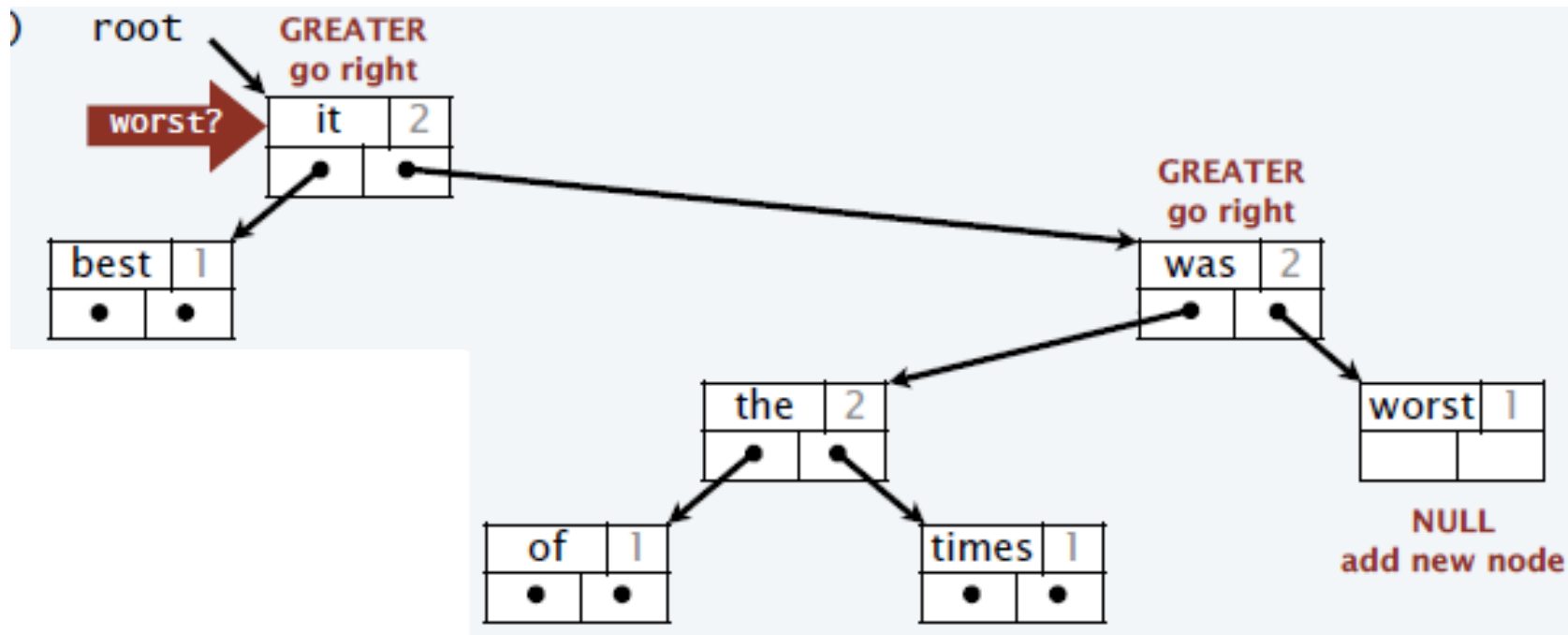
```
ItemType BSTreeGetMin(const BSTree* root) {  
    if (root == NULL) {  
        return NO_ITEM;  
    }  
    if (root->left == NULL) {  
        return root->item;  
    }  
    return BSTreeGetMin(root->left);  
}
```

Procurar – Versão iterativa

```
int BSTreeContains(const BSTree* root, const ItemType item) {  
    while (root != NULL) {  
        if (root->item == item) {  
            return 1;  
        }  
        if (root->item > item) {  
            root = root->left;  
        } else {  
            root = root->right;  
        }  
    }  
    return 0;  
}
```

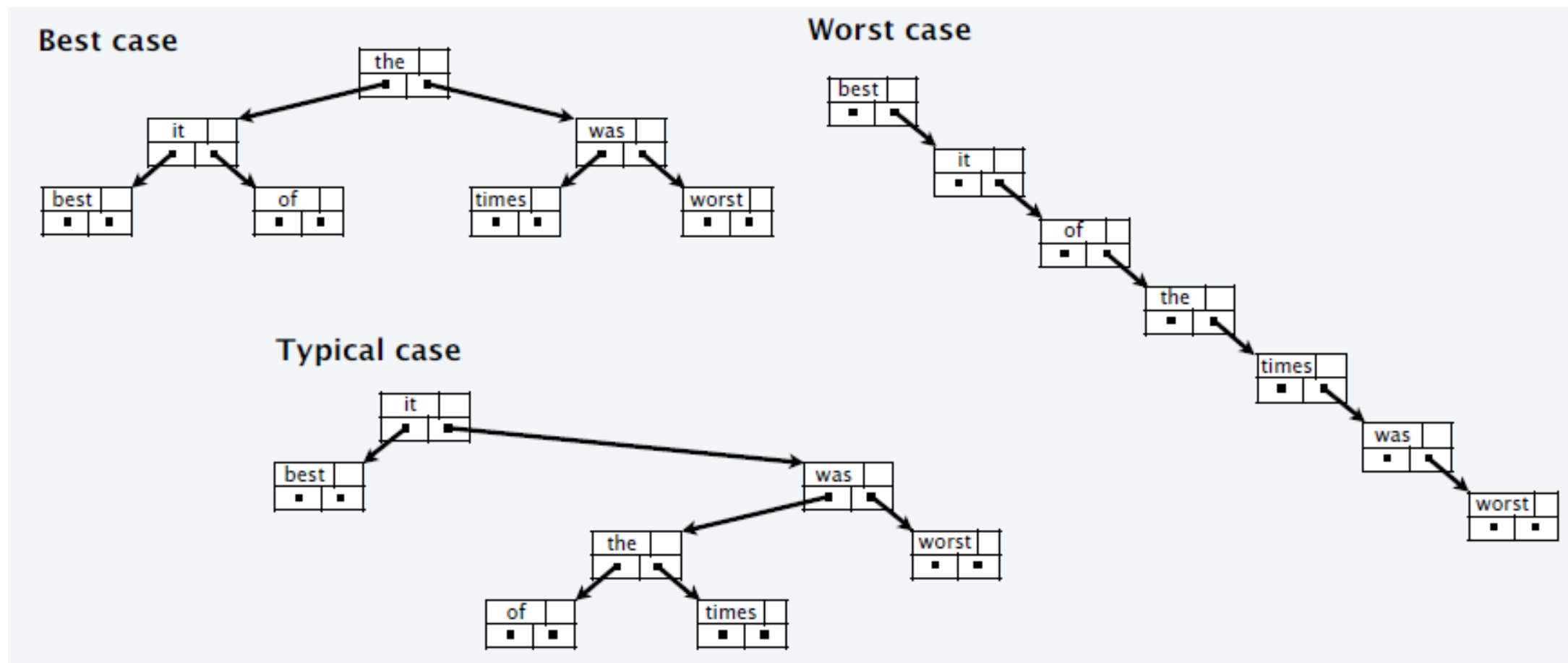


Adicionar como **folha**, mantendo a **ordem**



[Sedgewick & Wayne]

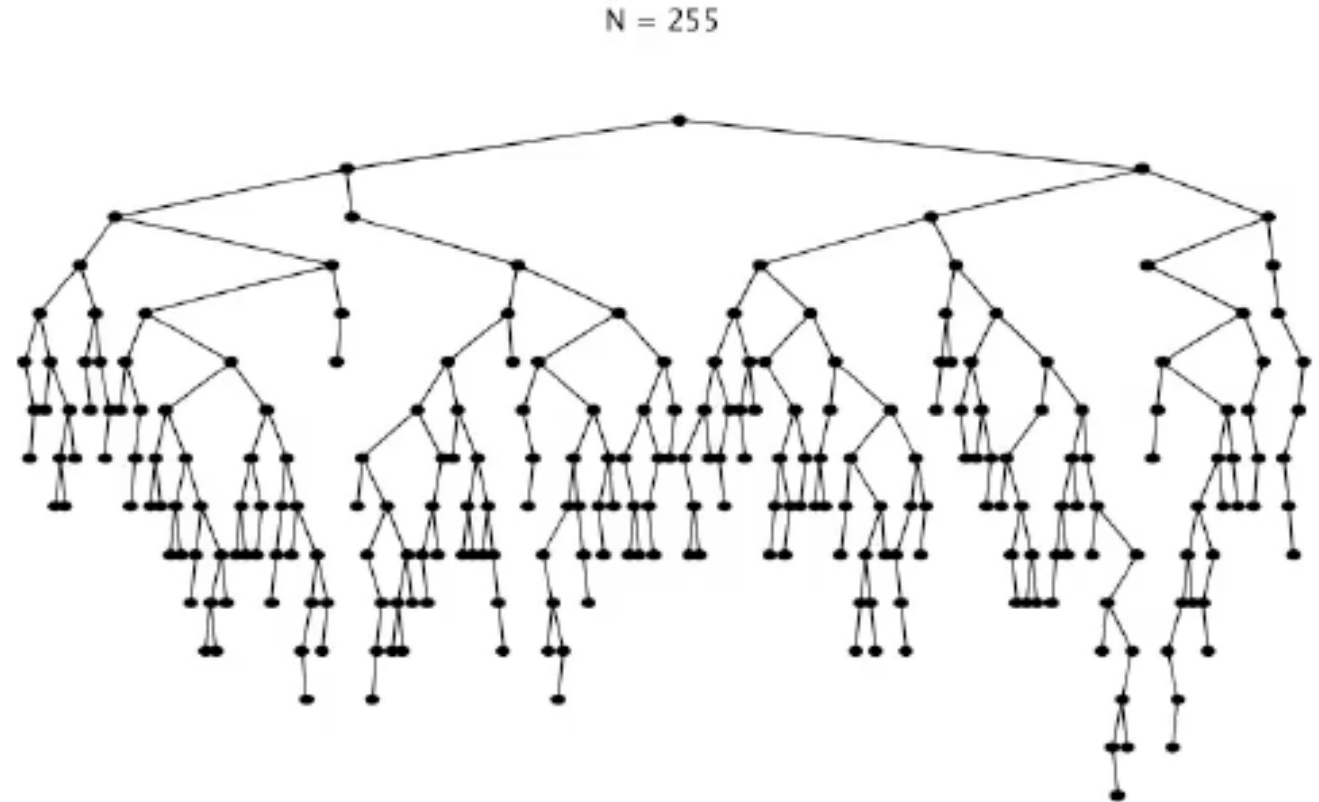
Diferentes **sequências de inserção**



[Sedgewick & Wayne]

Adição numa ordem aleatória

- Árvore **aprox.** equilibrada
- Mantém essa **tendência**

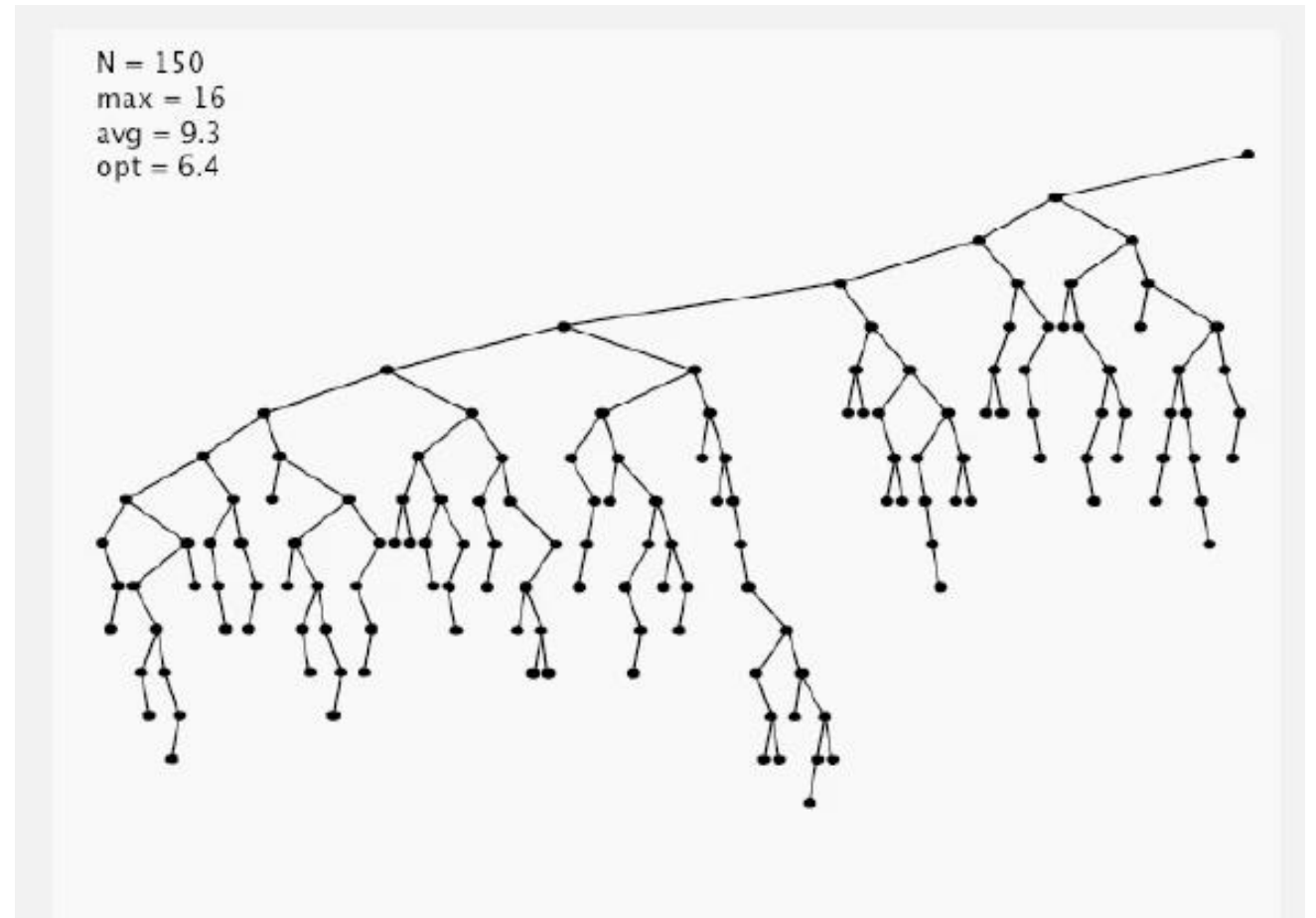


[Sedgewick & Wayne]

Após muitos apagamentos

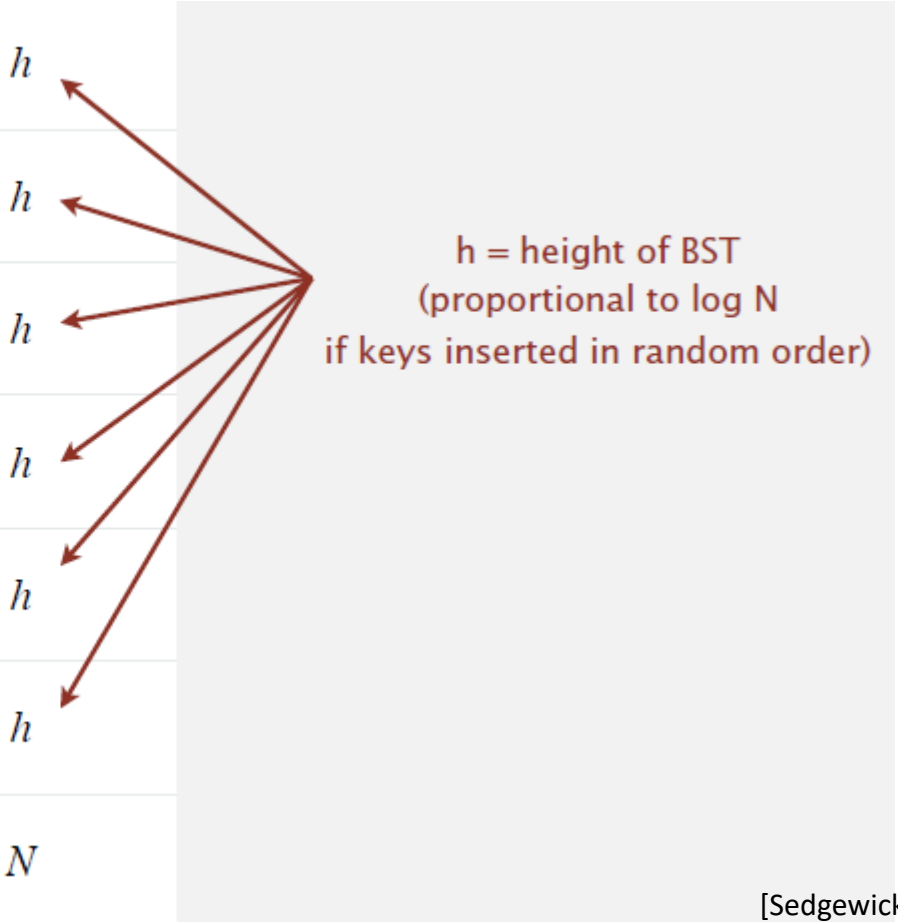
- Árvore perde alguma “simetria” !!
- Consequências ?

[Sedgewick & Wayne]



Lista ligada / Array ordenado / ABP

search	N	$\lg N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N



h = height of BST
(proportional to $\log N$
if keys inserted in random order)

[Sedgewick & Wayne]

Problema prático

- Os itens podem não ser adicionados de modo aleatório
 - Por exemplo, **adição ordenada** !!
- Como evitar o **pior caso / casos maus** ?
- **Árvores equilibradas** em altura !!
 - São ABPs de altura “aceitável”
 - **Árvores AVL** (1962)
 - **Red-black trees** – Java **TreeMap**

Árvores equilibradas em altura – Balanced Trees

Motivação

- **Esforço** computacional das operações habituais sobre ABPs depende do **comprimento do caminho** a partir da raiz da árvore
- **Evitar** que uma ABP tenha uma **altura “exagerada”**, para assegurar um bom “comportamento” – **Altura $\in O(\log n)$**
- **O que fazer ?**
- Assegurar que, para cada nó, a **altura** das suas duas **subárvores** não é “muito diferente” – **Critério de equilíbrio**

Quando fazer? / Como fazer?

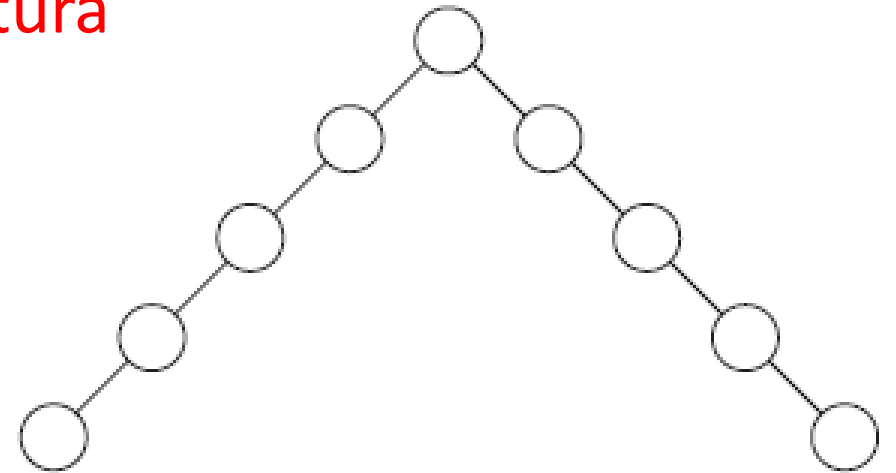
- **Assegurar** o **critério de equilíbrio** sempre que se adiciona ou remove um nó
- Tem de ser **fácil** de **verificar** e de **manter** !!
- **Reposicionar** nós / subárvores quando **falha** !!
- MAS, **manter** o **critério de ordem** da ABP !!
- Basta fazer a verificação / reposicionamento ao longo do **caminho** entre a raiz e o nó – **traceback**

Critério de equilíbrio?

- 1ª ideia
- As duas **subárvores da raiz** têm a **mesma altura**
- Boa ou má ideia ?

Critério de equilíbrio?

- 1ª ideia
- As duas **subárvores da raiz** têm a **mesma altura**
- **Má ideia...**
- É insuficiente
- Árvores desnecessariamente altas !!



[Weiss]

Critério de equilíbrio?

- 2ª ideia
- As duas **subárvores de cada nó** têm a **mesma altura**
- Boa ou má ideia ?

Critério de equilíbrio?

- 2ª ideia
- As duas subárvores de cada nó têm a mesma altura
- Má ideia...
- Demasiado restritiva !!
- Só seriam possíveis árvores completas, com $2^k - 1$ nós

Critério de equilíbrio?

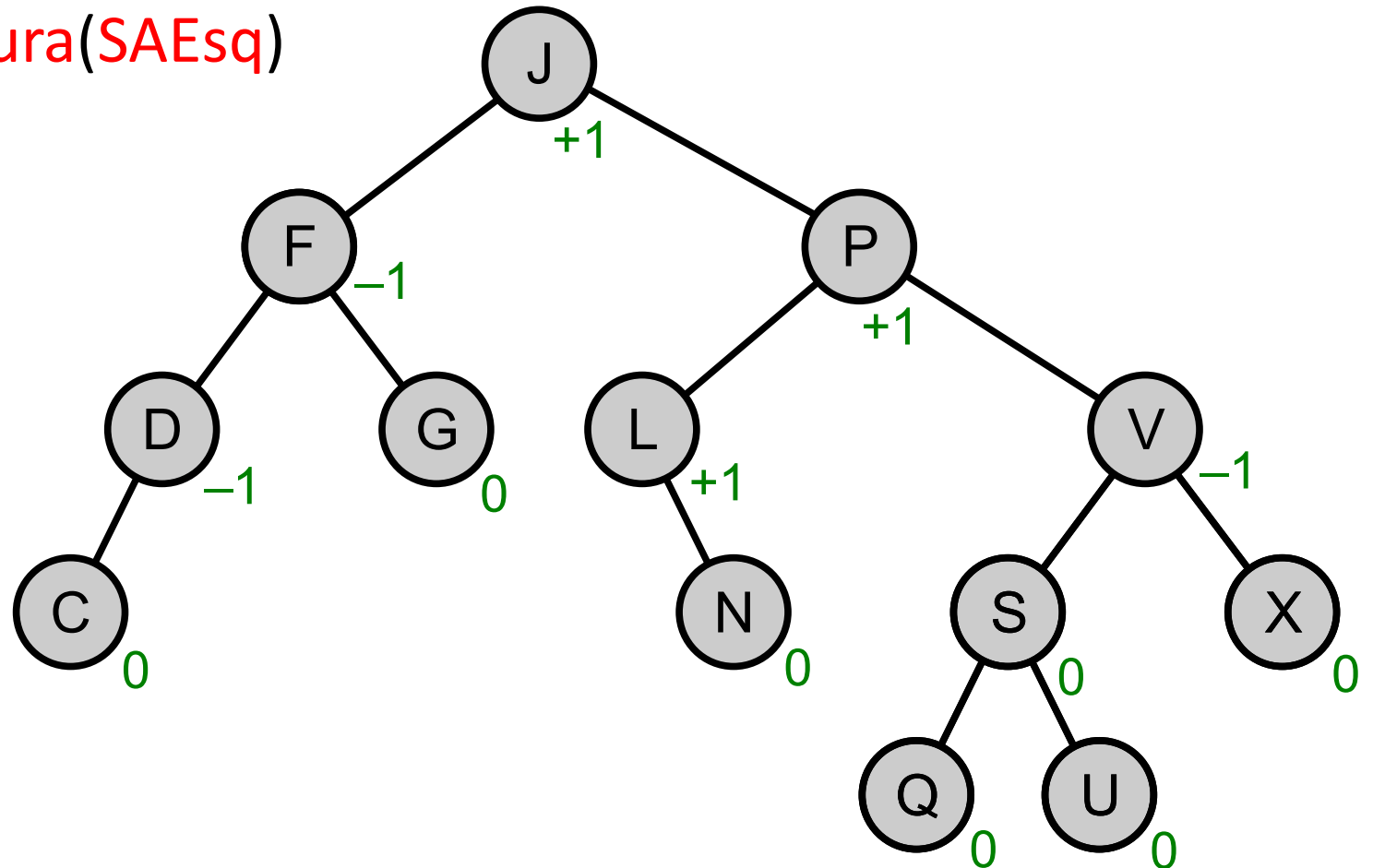
- 3ª ideia
- A **altura** das duas **subárvores de cada nó** difere, **quando muito**, de uma unidade (**0 ou ± 1**)
- Boa ou má ideia ?

Critério de equilíbrio?

- 3ª ideia
- A **altura** das duas **subárvores de cada nó** difere, **quando muito**, de uma unidade (**0 ou ± 1**)
- **Boa ideia !!**
- **Fácil de verificar e de manter**
- **Adicionar** a cada nó um **campo** com a sua **altura**

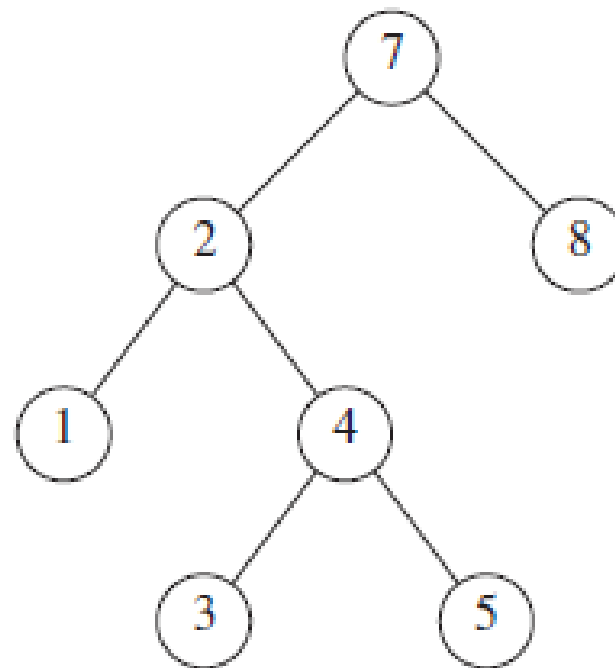
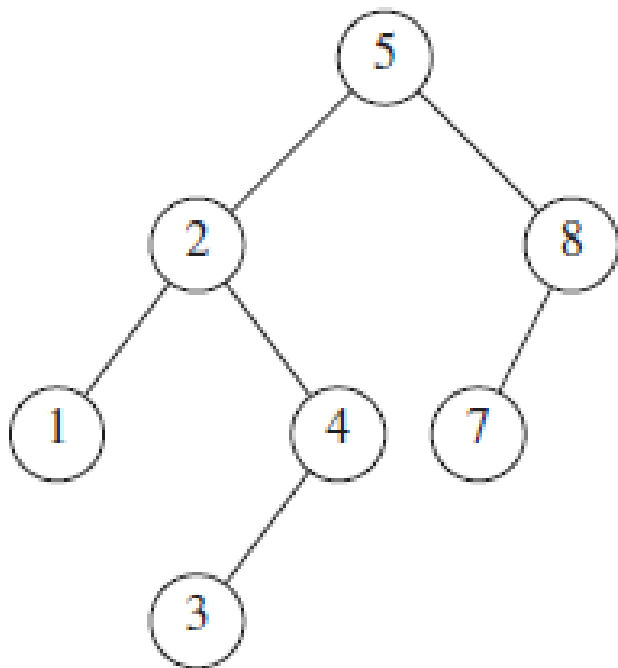
Fator de equilíbrio de um nó

- $F = \text{altura}(\text{SADir}) - \text{altura}(\text{SAEsq})$



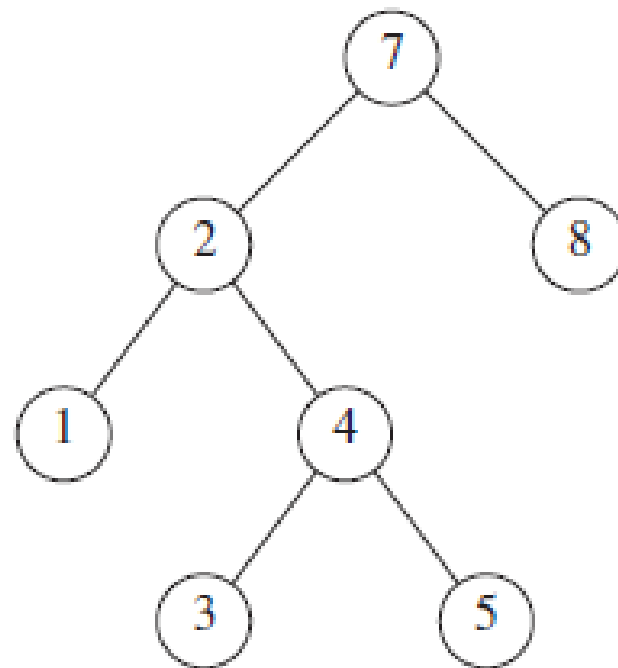
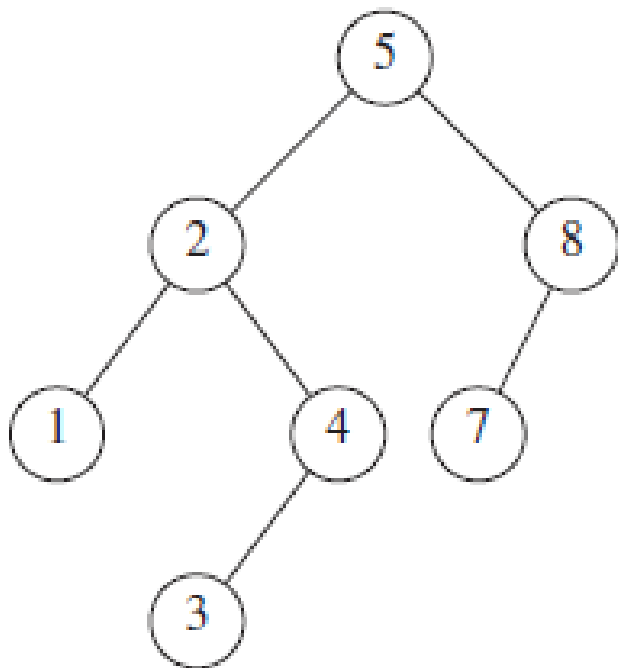
[Wikipedia]

ABPs equilibradas ?



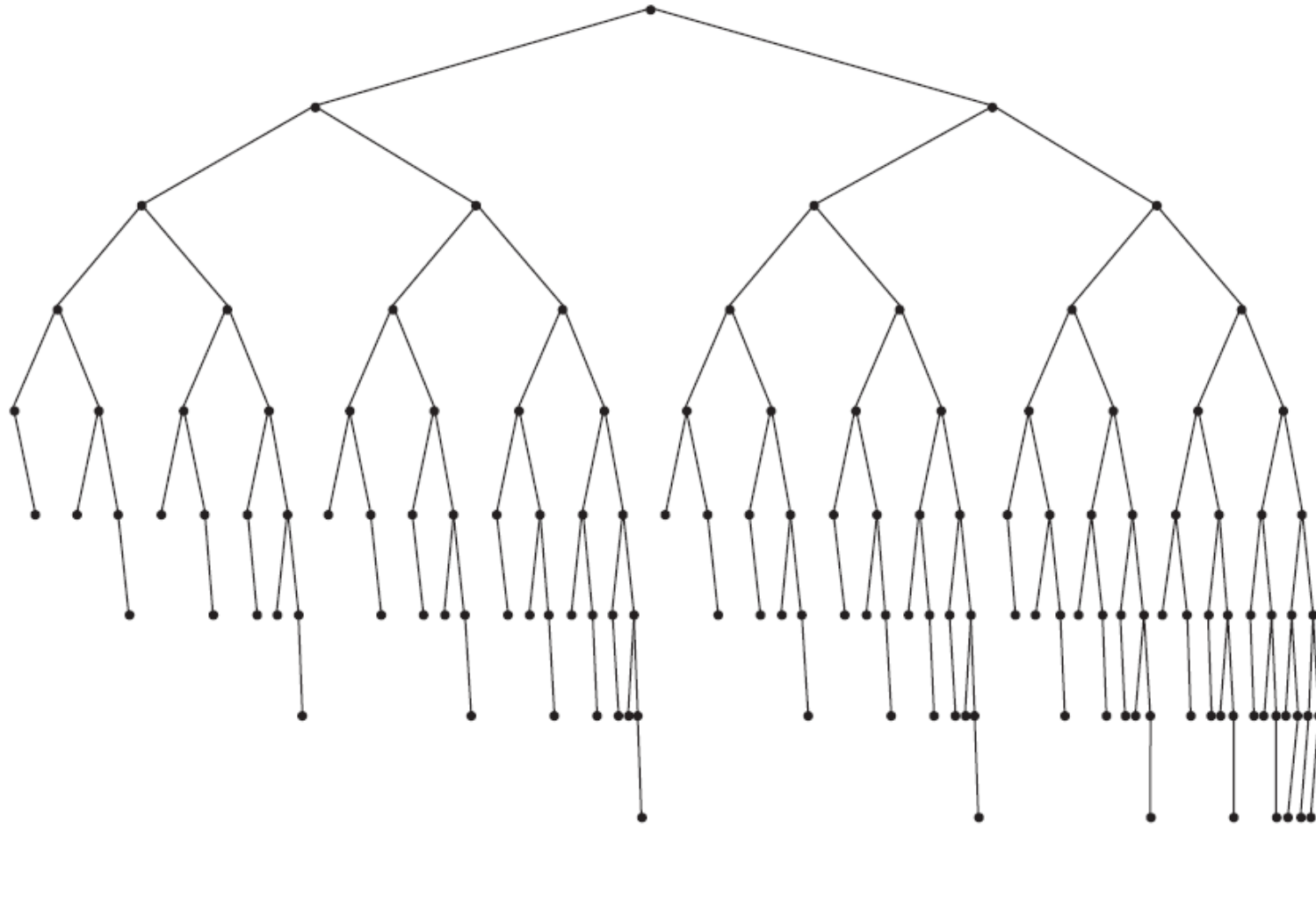
[Weiss]

Que nós falham a condição ?



[Weiss]

Árvore equilibrada – Qual é a sua altura ?



[Weiss]

Melhor caso ? / Pior caso ?

- Para uma dada altura h
- Qual é a árvore equilibrada que tem mais nós ?
- Qual é a árvore equilibrada que tem menos nós ?

Melhor caso ?

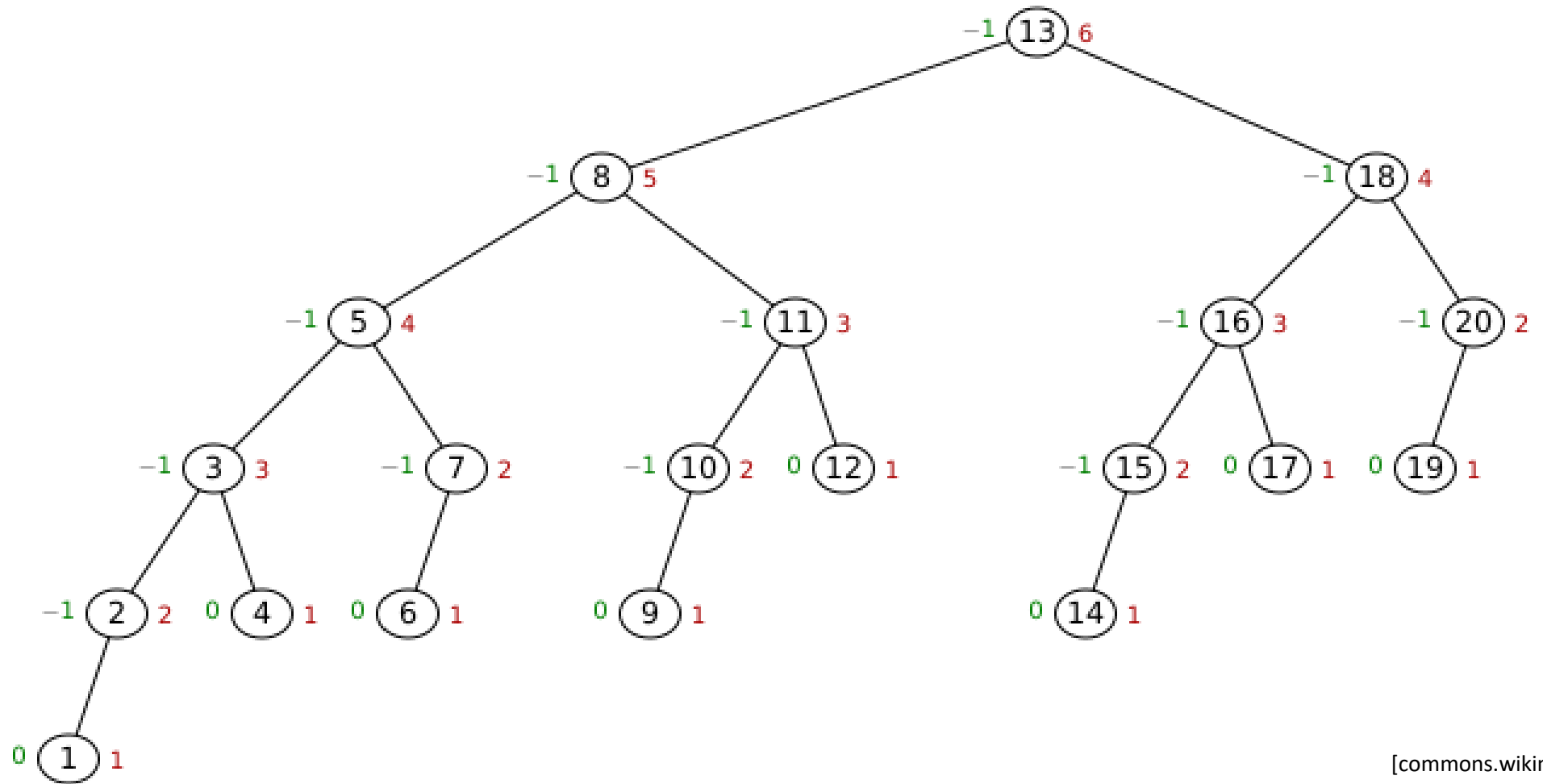
- Para uma dada altura h
- Qual é a árvore equilibrada que tem mais nós ?
- Uma árvore completa – todos os níveis preenchidos
- $h = \log_2(n+1) - 1$

Pior caso ?

- Para uma dada altura h
- Qual é a árvore equilibrada que tem menos nós ?
 - I.e., é o mais desequilibrada possível ?
- Uma árvore em que a altura das subárvores de cada nó não-terminal difere de 1
- E para cada folha tem, obviamente, duas subárvores (vazias) com a mesma altura
- $h = ?$

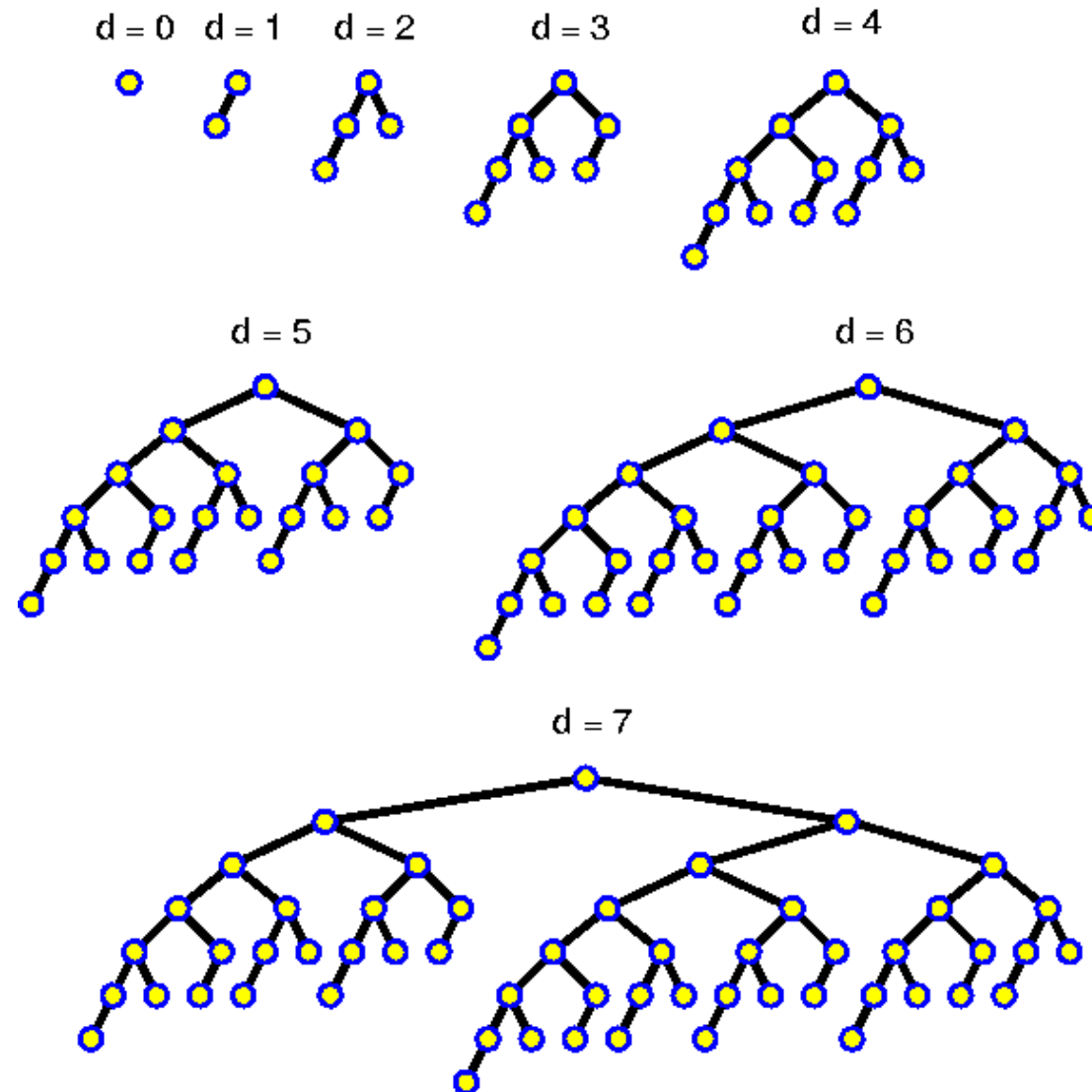
Árvores de Fibonacci

Árvore de Fibonacci



[commons.wikimedia.org]

Árvores de Fibonacci – Sucessivas alturas



[rikuti.de]

Altura **vs** Número de nós

- **N_h** = nº de nós de uma árvore de Fibonacci de altura h
- $N_0 = 1$
- $N_1 = 2$
- ...
- **$N_h = 1 + N_{h-1} + N_{h-2}$**

Altura vs Número de nós – Tarefas

- $N_h = 1 + N_{h-1} + N_{h-2}$
- **Tarefas:**
 - Fazer uma **tabela** e relacionar com os **números de Fibonacci**
 - Obter uma **solução simplificada** para a eq. de recorrência
 - Relacionar a **altura** com o **nº de nós**
 - Obter a **ordem de complexidade** para a **altura**

Árvores AVL

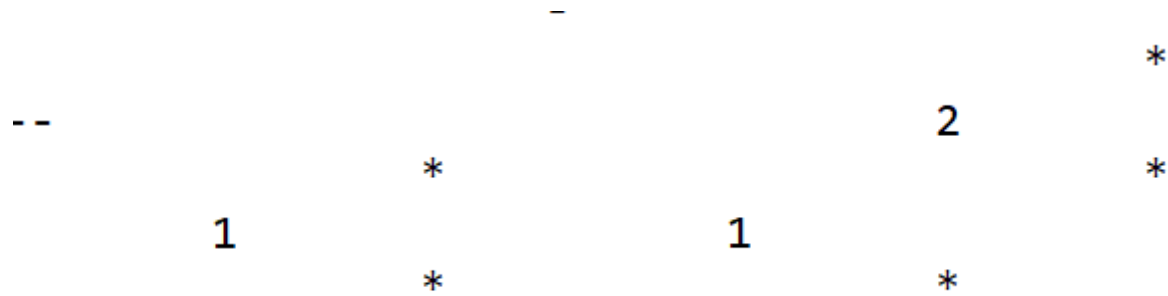
Árvore AVL – Inserir + Equilibrar, se necessário

[Wikipedia]

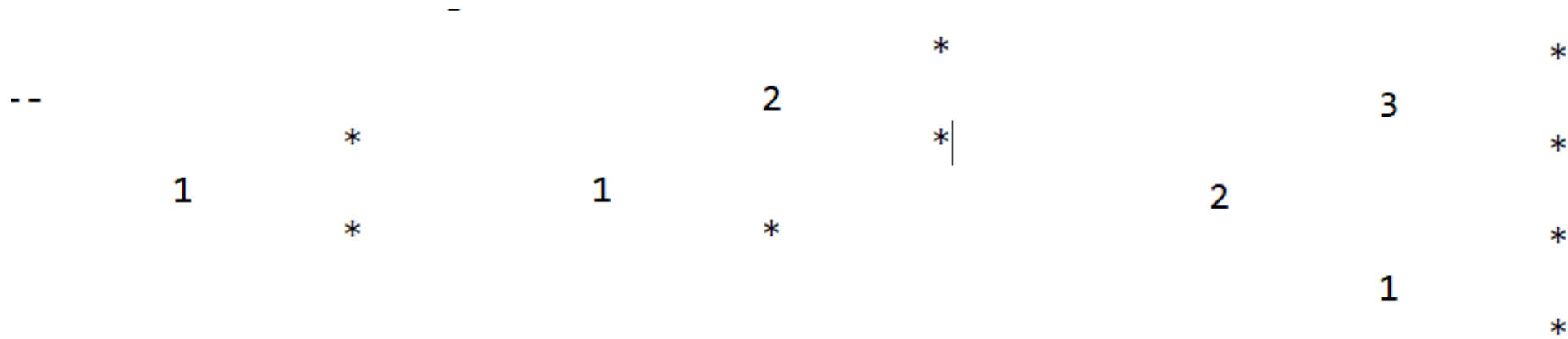
Sequência após **inserir** + **equilibrar**

--
1 *
 *

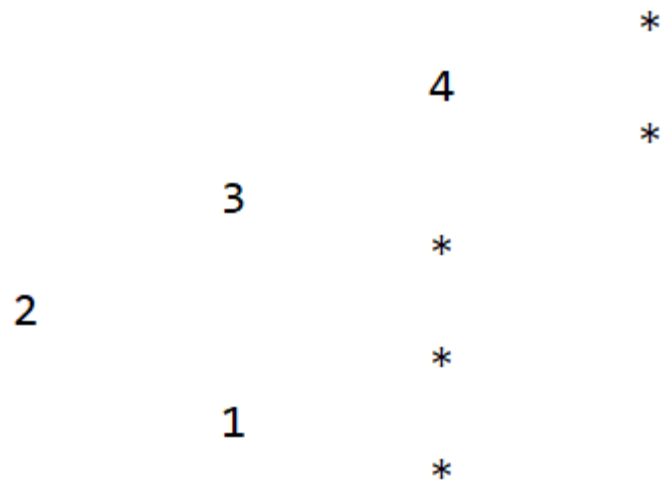
Sequência após **inserir** + **equilibrar**



Sequência após **inserir** + **equilibrar**



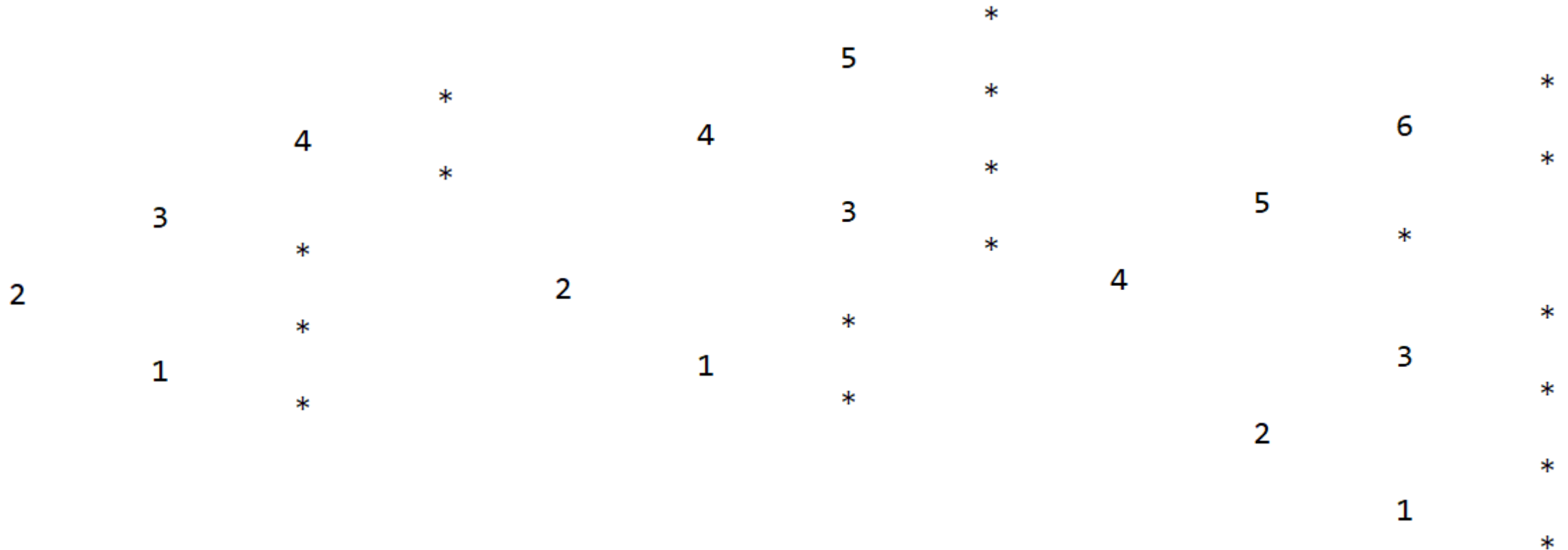
Sequência após **inserir** + **equilibrar**



Sequência após **inserir** + **equilibrar**



Sequência após **inserir** + **equilibrar**

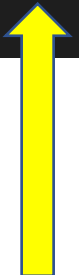



Fator de equilíbrio

- Para cada nó
- As duas sub-árvores têm a mesma altura
- Ou a sua altura difere de 1
- $F = \text{Altura}(\text{SADireita}) - \text{Altura}(\text{SAEsquerda})$
- $F = -1, 0, 1$
- Se uma árvore estiver equilibrada, a **adição/remoção** de um nó pode forçar **F** a tomar o valor **+2** ou **-2**
- Podemos usar para **identificar** os nós “**desequilibrados**” !!

Nó de uma árvore AVL – Altura


```
struct _AVLTreeNode {  
    ItemType item;  
    struct _AVLTreeNode* left;  
    struct _AVLTreeNode* right;  
    int height;  
};
```



```
int AVLTreeGetHeight(const AVLTree* root) {  
    if (root == NULL) return -1;  
    return root->height;   
}
```

Atualizar após inserir / remover um nó

```
static void _updateNodeHeight(AVLTree* t) {  
    assert(t != NULL);  
  
    int leftHeight = AVLTreeGetHeight(t->left);  
  
    int rightHeight = AVLTreeGetHeight(t->right);  
  
    if (leftHeight >= rightHeight) {  
        t->height = leftHeight + 1;  
    } else {  
        t->height = rightHeight + 1;  
    }  
}
```

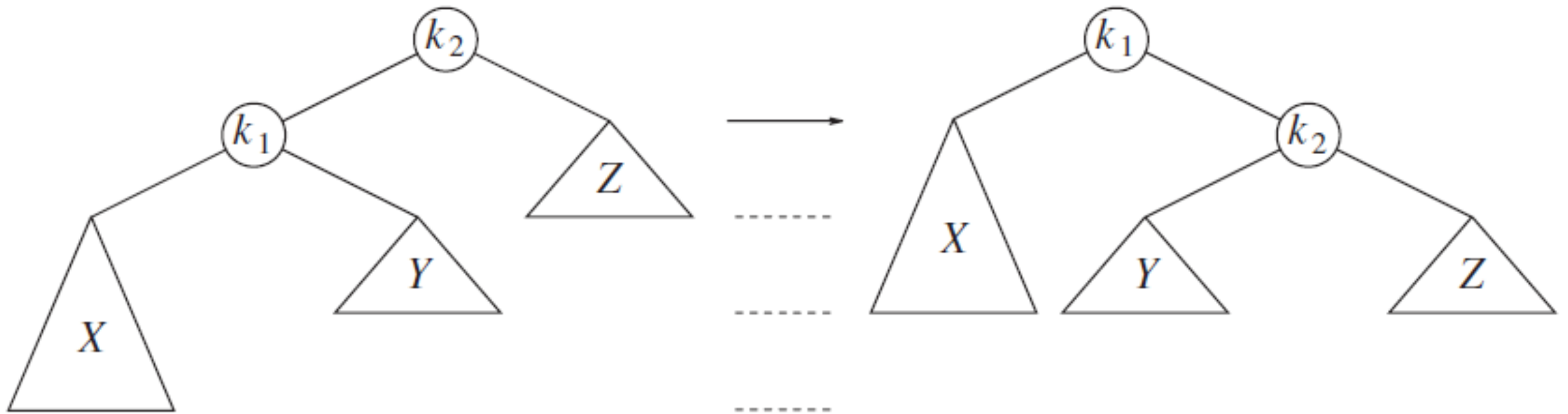


Como **corrigir / equilibrar**, se necessário ?

- Efetuando **operações de rotação** – 4 possibilidades
- MAS, assegurando o **critério de ordem** das ABPs
- Apenas **troca de ponteiros** !!

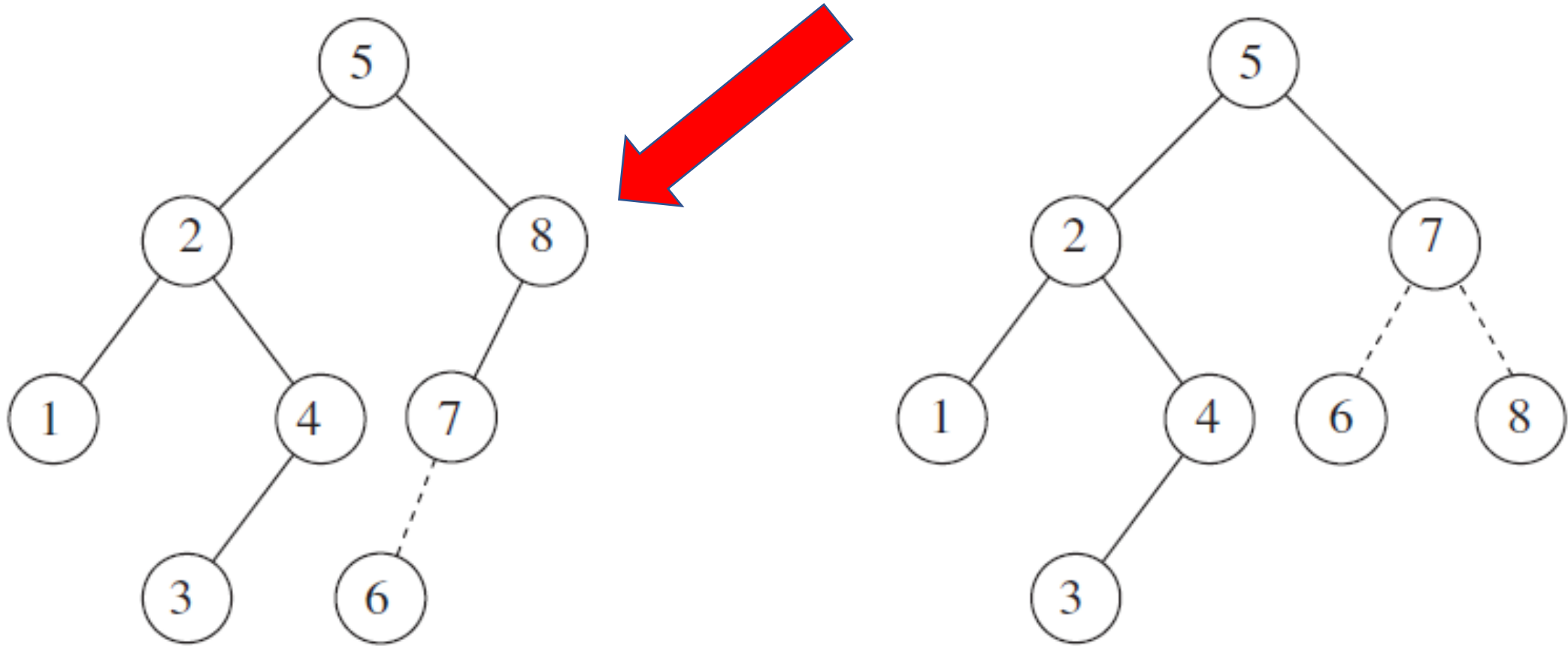
- **Rotações simples** à esquerda ou à direita
- **Rotações duplas** à esquerda ou à direita
 - Sequência de duas rotações simples

Rotação simples à esquerda : $F(k_2) = -2$



[Weiss]

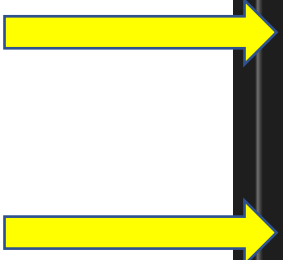
Rotação simples à esquerda : $F(8) = -2$



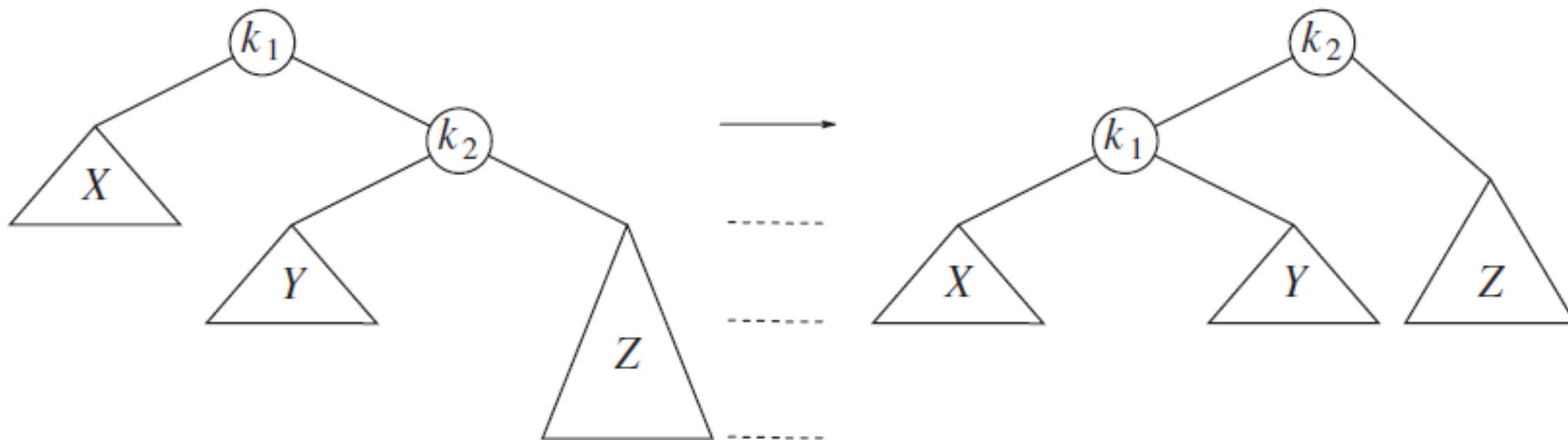
[Weiss]

Rotação simples à esquerda

```
static void _singleRotateWithLeftChild(AVLTree** p) {  
    AVLTree* pLeft = (*p)->left;  
  
    (*p)->left = pLeft->right;  
    pLeft->right = *p;  
  
    _updateNodeHeight(*p);  
    _updateNodeHeight(pLeft);  
  
    *p = pLeft;  
}
```

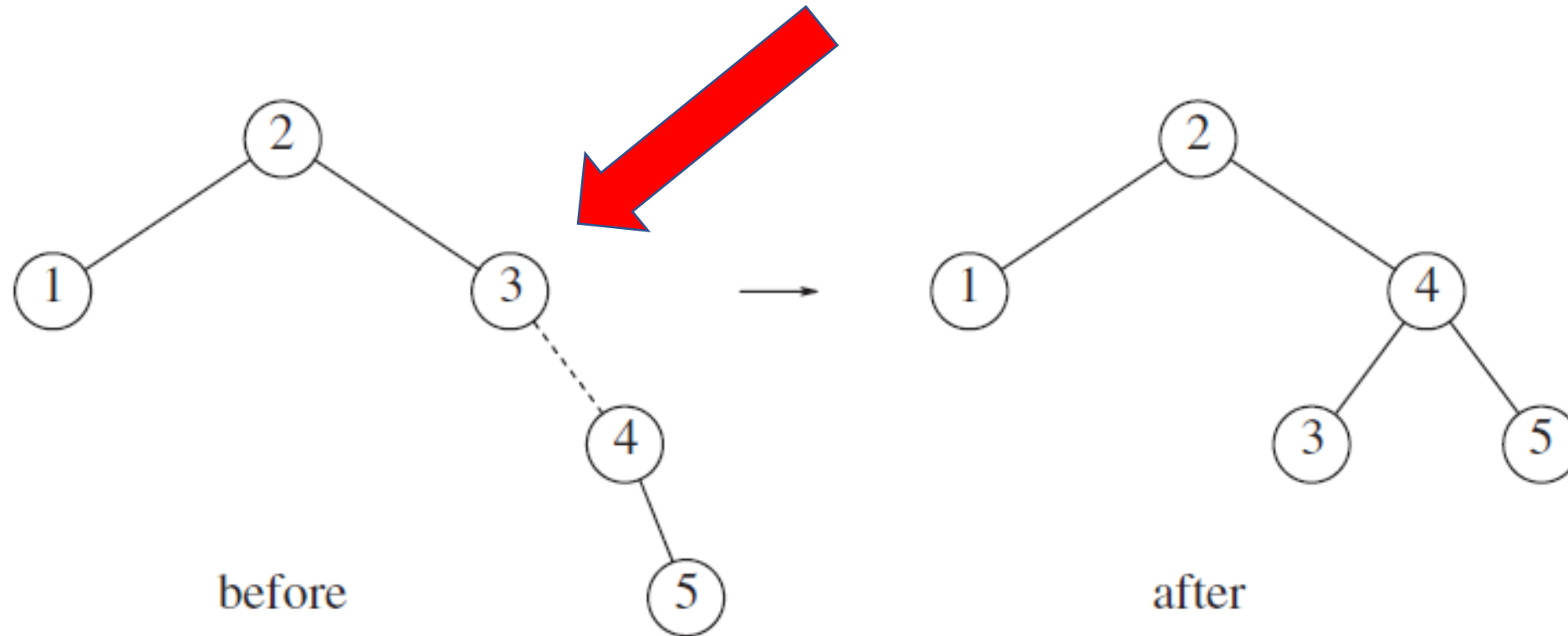


Rotação simples à direita : $F(k_1) = +2$



[Weiss]


Rotação simples à direita : $F(3) = +2$



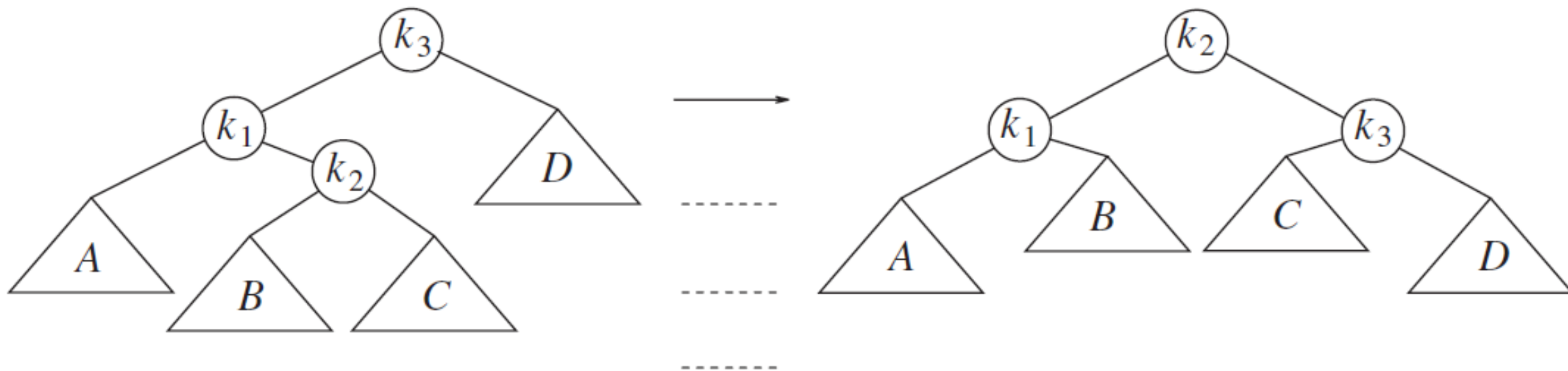
[Weiss]

Rotação simples à direita

```
static void _singleRotateWithRightChild(AVLTree** p) {  
    AVLTree* pRight = (*p)->right;  
  
    (*p)->right = pRight->left;  
    pRight->left = *p;  
  
    _updateNodeHeight(*p);  
    _updateNodeHeight(pRight);  
  
    *p = pRight;  
}
```




Rotação dupla à esquerda – Como identificar?



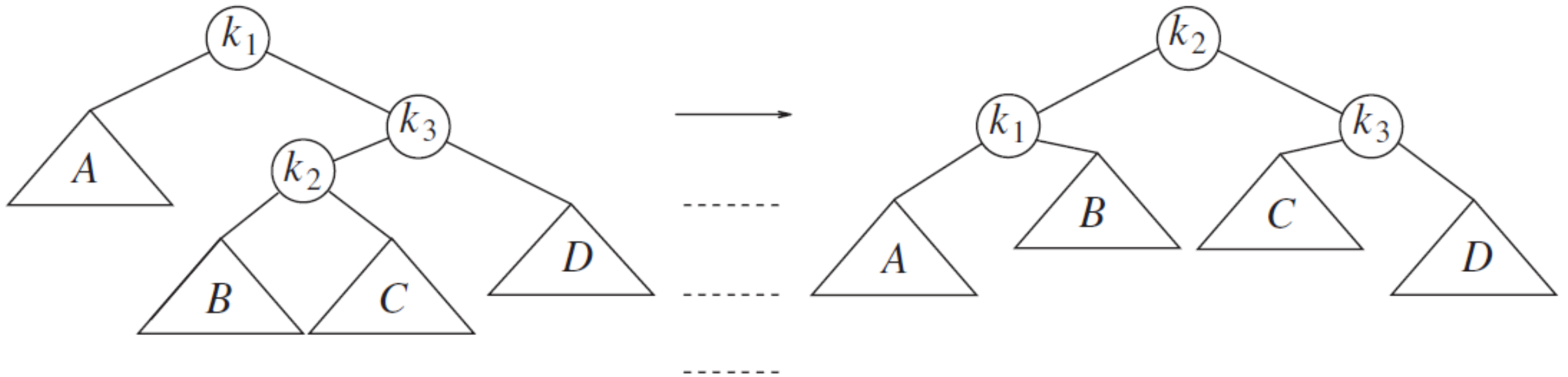
[Weiss]

Rotação dupla à esquerda




```
static void _doubleRotateWithLeftChild(AVLTree** p) {  
    _singleRotateWithRightChild(&(*p)->left);  
    _singleRotateWithLeftChild(p);  
}
```

Rotação dupla à direita – Como identificar?



[Weiss]

Rotação dupla à direita




```
static void _doubleRotateWithRightChild(AVLTree** p) {  
    _singleRotateWithLeftChild(&(*p)->right);  
    _singleRotateWithRightChild(p);  
}
```

Inserir um novo nó

- O novo nó é adicionado como uma **folha**
- Respeitando o **critério de ordem**
- Ao fazer o **traceback** das chamadas recursivas, verificar se há algum **nó desequilibrado** ao longo do **caminho de retorno à raiz**
- Identificar que **tipo de rotação** é necessário efetuar
- **TAREFA:** **analisar o código** da função que adiciona um novo nó

Inserir um novo nó



```
int AVLTreeAdd(AVLTree** pRoot, const ItemType item) {
    AVLTree* root = *pRoot;

    if (root == NULL) {
        root = (struct _AVLTreeNode*)malloc(sizeof(*root));
        assert(root != NULL);

        root->item = item;
        root->left = root->right = NULL;
        root->height = 0;

        *pRoot = root;
        return 1;
    }
}
```

Inserir um novo nó

```
if (item < root->item) {  
    // Try to insert on the left  
    if (AVLTreeAdd(&(root->left), item) == 0) {  
        // No success  
        return 0;  
    }  
  
    // Unbalanced on the left ?  
    if (AVLTreeGetHeight(root->left) - AVLTreeGetHeight(root->right) == 2) {  
        if (item < root->left->item) {  
            _singleRotateWithLeftChild(pRoot);  
        } else {  
            _doubleRotateWithLeftChild(pRoot);  
        }  
    }  
    _updateNodeHeight(root);  
    return 1;  
}
```

Remover um nó

- O nó é removido usando o algoritmo desenvolvido para as ABPs
- Mantendo o **critério de ordem**
- Ao fazer o **traceback** das chamadas recursivas, verificar se há algum **nó desequilibrado** ao longo do **caminho de retorno à raiz**
- Usar uma **função auxiliar** para efetuar o equilíbrio
 - Estratégia distinta neste caso
- **TAREFA:** **analisar o código** – onde é chamada a função auxiliar ?


Função auxiliar

```
static void _balanceNode(AVLTree** pRoot) {
    int leftHeight, rightHeight;

    if (*pRoot == NULL) return;
    leftHeight = AVLTreeGetHeight((*pRoot)->left);
    rightHeight = AVLTreeGetHeight((*pRoot)->right);

    if (leftHeight - rightHeight == 2) {
        leftHeight = AVLTreeGetHeight((*pRoot)->left->left);
        rightHeight = AVLTreeGetHeight((*pRoot)->left->right);
        if (leftHeight >= rightHeight)
            _singleRotateWithLeftChild(pRoot);
        else
            _doubleRotateWithLeftChild(pRoot);
    }
}
```

Função auxiliar



```
    } else if (rightHeight - leftHeight == 2) {  
        rightHeight = AVLTreeGetHeight((*pRoot)->right->right);  
        leftHeight = AVLTreeGetHeight((*pRoot)->right->left);  
        if (rightHeight >= leftHeight)  
            _singleRotateWithRightChild(pRoot);  
        else  
            _doubleRotateWithRightChild(pRoot);  
    } else  
        (*pRoot)->height =  
            leftHeight > rightHeight ? leftHeight + 1 : rightHeight + 1;  
}
```

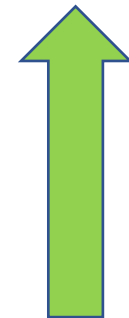
ABP vs AVL

1ª experiência

- **Criar** uma árvore vazia
- **Inserir** ordenadamente sucessivos números pares: 2, 4, 6, ...
- **Procurar** cada um desses números pares na árvore
- **Procurar** sucessivos inteiros positivos (ímpares + pares) na árvore
- **Contar** o número de **comparações** efetuadas em cada nó
 - 1 ou 2 **comparações** por nó visitado

Procurar os sucessivos números pares

nós	Altura ABP	Nº médio comps	Altura AVL	Nº médio comps
5000	4999	5001	12	17,69
10000	9999	10001	13	19,19
20000	19999	20001	14	20,69
40000	39999	40001	15	22,19



Procurar sucessivos números ímpares e pares

nós	Altura ABP	Nº médio comps	Altura AVL	Nº médio comps
5000	4999	5000,5	12	18,19
10000	9999	10000,5	13	19,69
20000	19999	20000,5	14	21,19
40000	39999	40000,5	15	22,69



2ª experiência

- Criar uma árvore vazia
- Inserir uma sequência de números aleatórios
- Procurar cada um desses números na árvore
- Contar o número de comparações efetuadas em cada nó
 - 1 ou 2 comparações por nó visitado

Procurar os sucessivos números aleatórios

nós	Altura ABP	Nº médio comps	Altura AVL	Nº médio comps
2500	27	19,64	12	16,18
5000	25	22,10	14	17,66
10000	30	25,72	15	18,85
20000	28	25,83	16	19,83
40000	32	26,73	16	20,91

