

# Árvores Binárias III


13/11/2024

# Ficheiro ZIP

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

# Sumário

Uma árvore é equilibrada quando não é excessiva. Ou seja, quando há um critério.

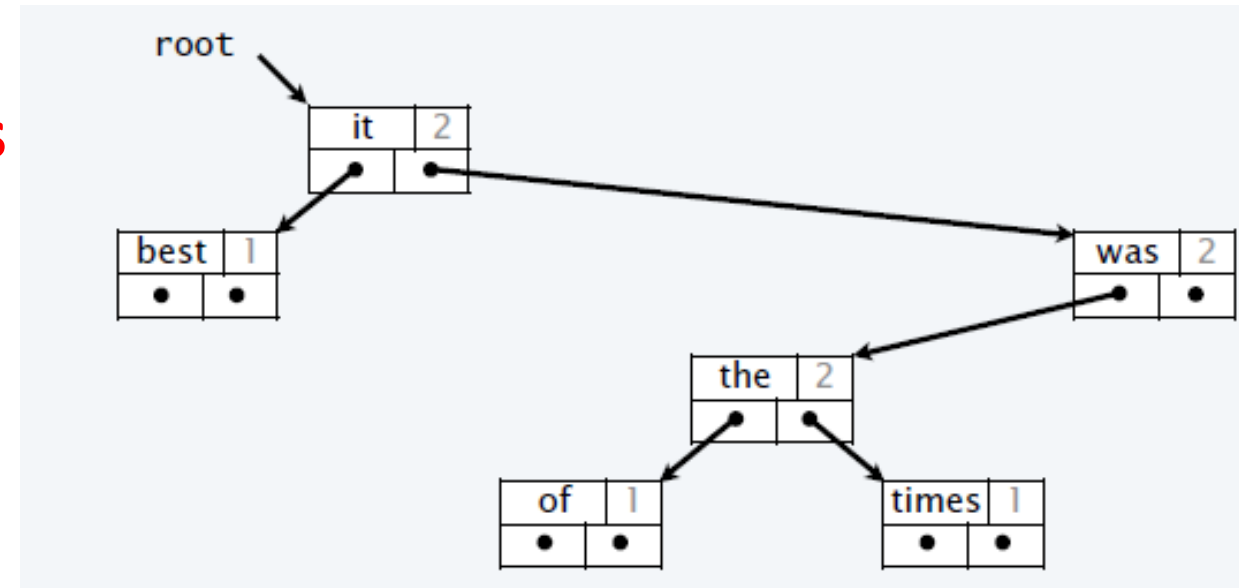
- O TAD **Árvore Binária de Procura** (conclusão)
- Análise do **desempenho** das operações habituais sobre ABPs
- **Árvores equilibradas em altura – Árvores AVL**
- Operações de rotação para manutenção da condição de equilíbrio
- Análise do **desempenho**: ABPs **vs** AVLs
- Exercícios / Tarefas 

# Árvores Binárias de Procura (ABP)

- Binary Search Trees (BST)

# ABP – Critério de **ordem** – Definição recursiva

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



[Sedgewick & Wayne]

- Qual a **ordem de inserção dos nós** nesta árvore ?

# ABP – Procurar elemento – 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;  
}
```

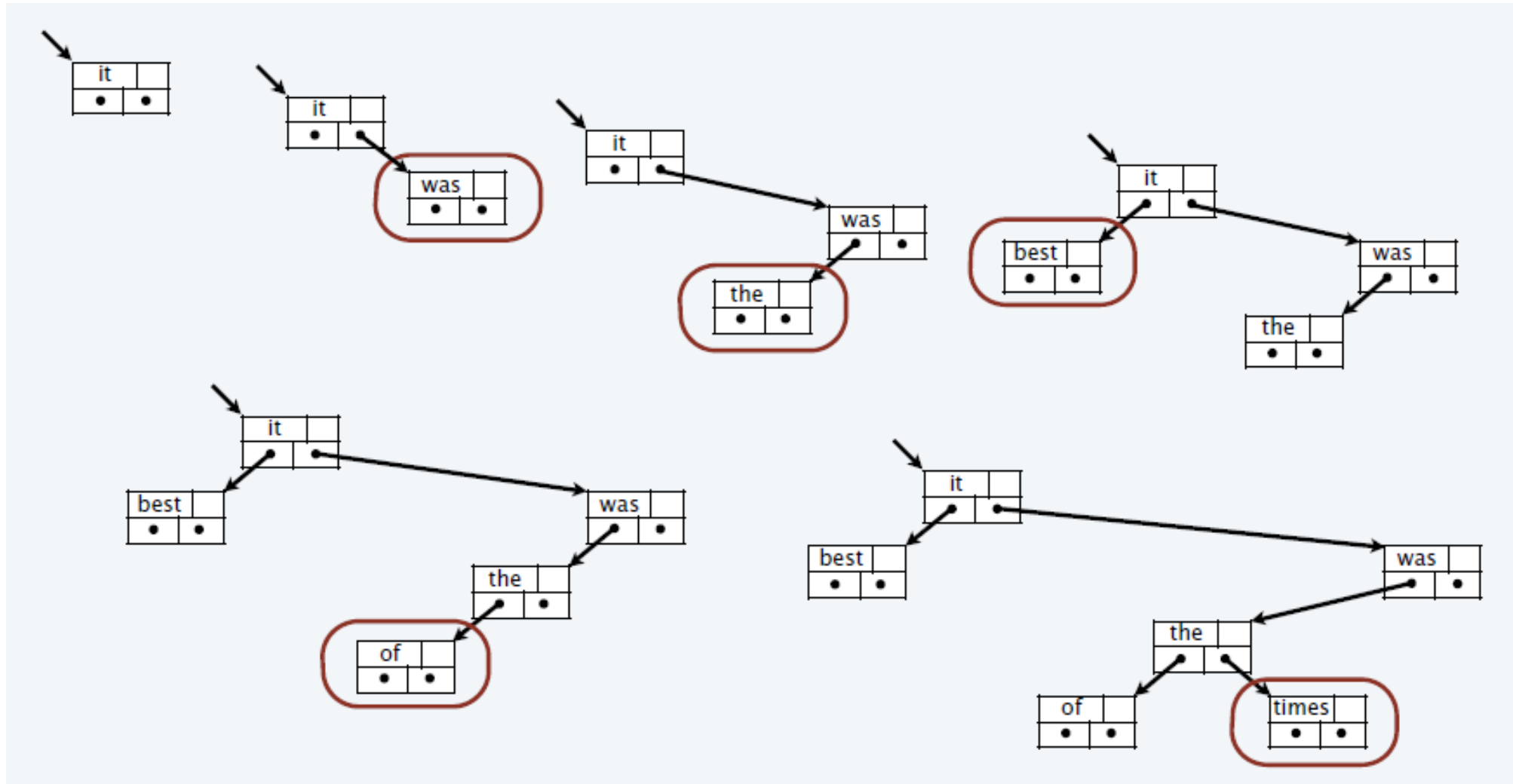
# ABP – Adicionar um elemento

# ABP – Adicionar um elemento

- Restrição : não adicionar duplicados !!
- Restrição : manter a ordem dos elementos !!
- Como fazer ?
- Inserir o novo item como folha da árvore, na posição correta



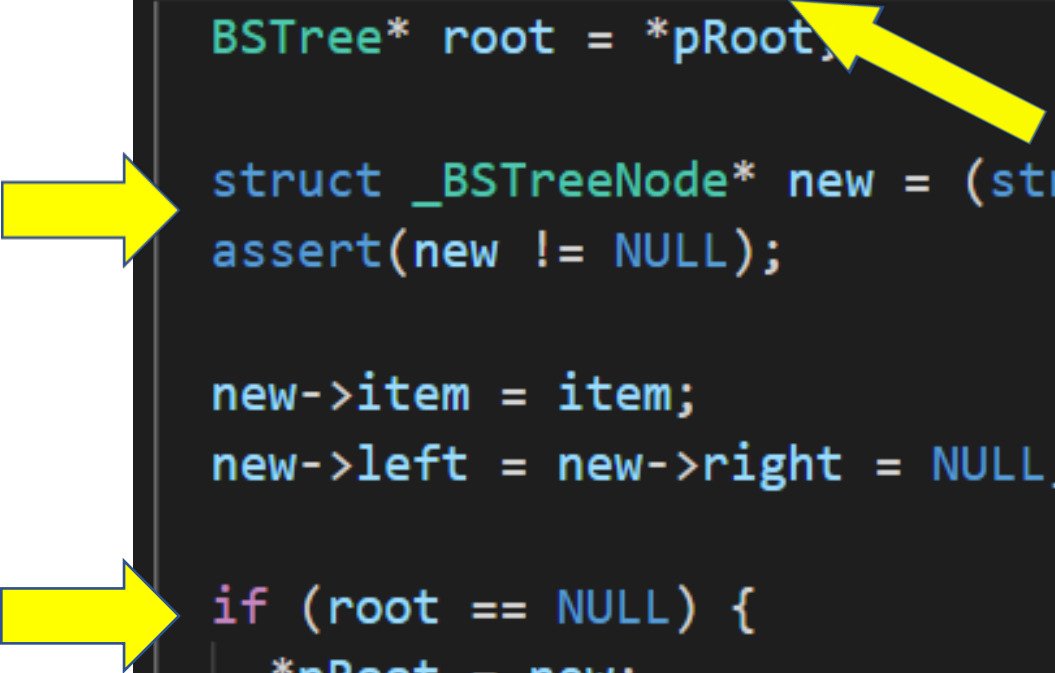
# ABP – Adicionar como folha, manter a ordem



[Sedgewick & Wayne]

# Adicionar – Versão iterativa – Criar o novo nó

```
int BSTreeAdd(BSTree** pRoot, const ItemType item) {  
    BSTree* root = *pRoot;  
  
    struct _BSTreeNode* new = (struct _BSTreeNode*)malloc(sizeof(*new));  
    assert(new != NULL);  
  
    new->item = item;  
    new->left = new->right = NULL;  
  
    if (root == NULL) {  
        *pRoot = new;  
        return 1;  
    }  
}
```



# Adicionar – Procurar a posição da nova folha

```
struct _BSTreeNode* prev = NULL;
struct _BSTreeNode* current = root;

while (current != NULL) {
    if (current->item == item) {
        free(new);
        return 0;
    } // Not allowed

    prev = current;
    if (current->item > item) {
        current = current->left;
    } else {
        current = current->right;
    }
}
```

- Usar 2 ponteiros auxiliares para percorrer a árvore
- Se o elemento já pertence à árvore, não fazer nada !
- A que subárvore pertencerá o novo elemento ?

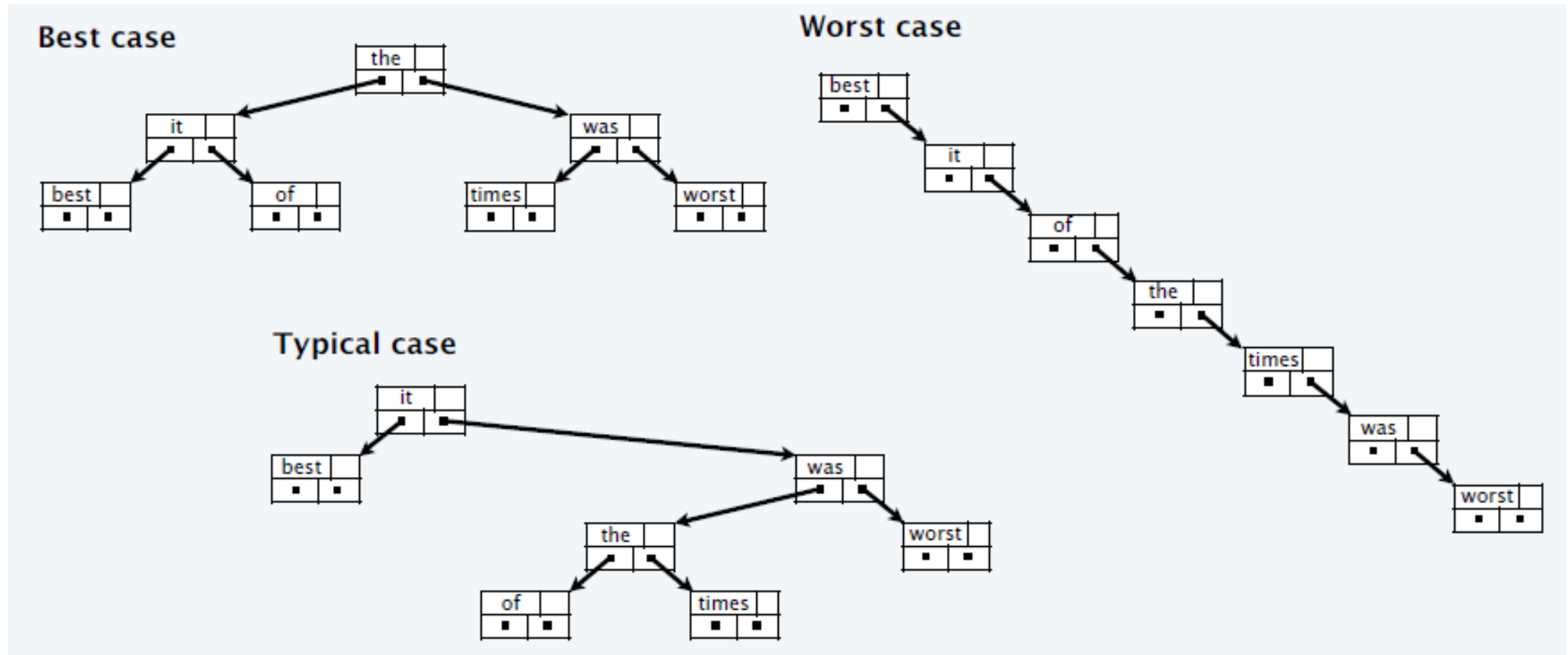
# Adicionar – Ancorar a nova folha



```
if (prev->item > item) {  
    prev->left = new;  
} else {  
    prev->right = new;  
}  
return 0;  
}
```

- A nova folha é menor ou maior do que o seu nó pai ?
- Ancorar como filho esquerdo ou filho direito !

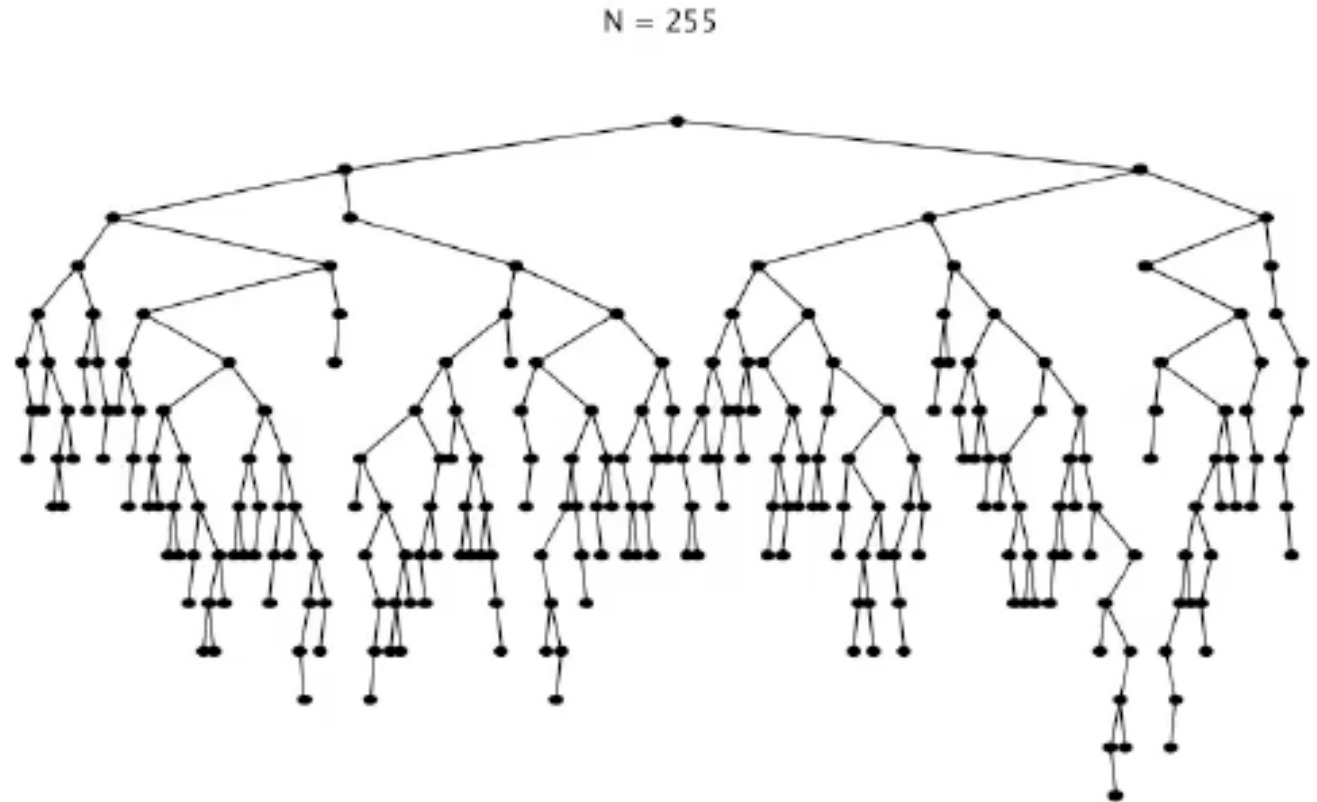
# Altura – Diferentes sequências de inserção



[Sedgewick & Wayne]

# Altura – Adição numa ordem aleatória

- Inserir **muitos elementos** numa **ordem aleatória**
- Árvore **aprox.** equilibrada !!



[Sedgewick & Wayne]

# ABP – Remove um elemento

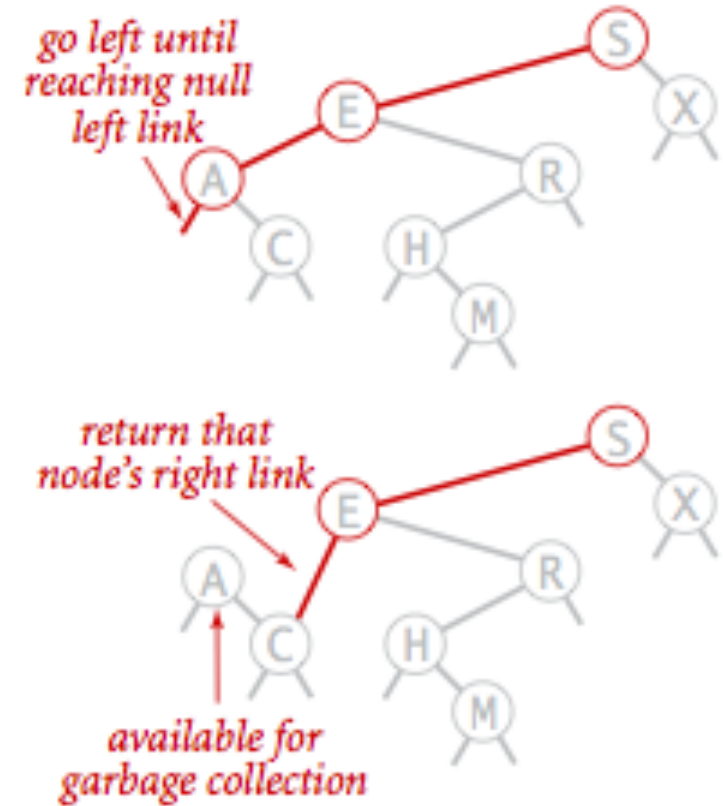
# ABP – Remove um elemento

- Restrição : **manter a ordem** dos elementos após a remoção !!
- Como fazer ?
- Como remover o **menor / maior elemento** de uma árvore ?
  - Casos **simples**
- Remover um **qualquer elemento** : a estratégia de **Hibbard**



# ABP – Remove o **menor** elemento

- O menor elemento está no “**nó mais à esquerda**” !
- **Folha** ?
- Nó só **com subárvore direita** ?
- E se for a **raiz** ?



[Sedgewick & Wayne]

# ABP – **Remove** – A estratégia de **Hibbard**

- Dado um **elemento a remover**, **procurar** o respetivo **nó**
- **Caso 1** : é uma **folha** – FÁCIL !!
- **Caso 2** : só tem **subárvore esquerda** – FÁCIL !!
- **Caso 3** : só tem **subárvore direita** – FÁCIL !!
- **Caso 4** : tem **2 subárvores**

# Procurar recursivamente o nó a remover

```
int BSTreeRemove(BSTree** pRoot, const ItemType item) {
    BSTree* root = *pRoot;

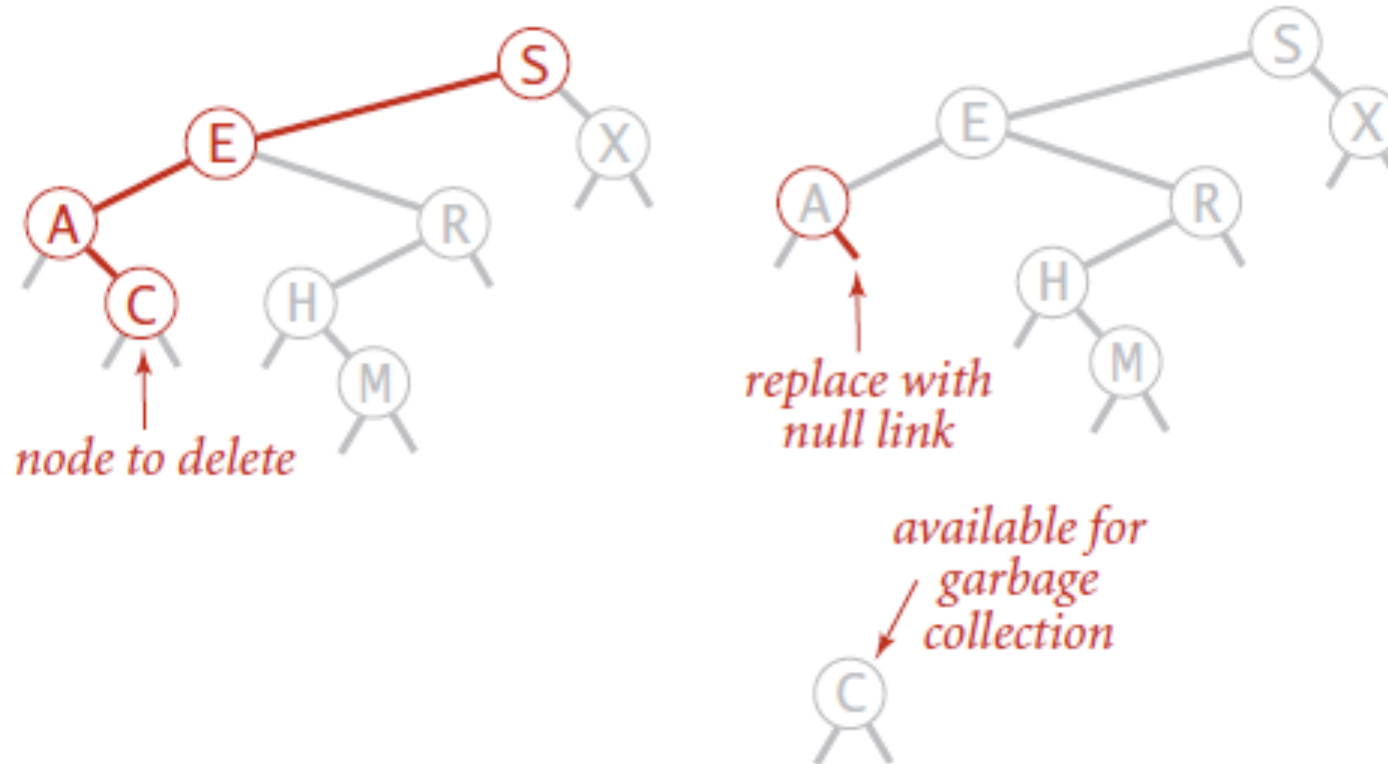
    if (root == NULL) {
        return 0;
    }
    if (root->item == item) {
        _removeNode(pRoot);
        return 1;
    }
    if (root->item > item) {
        return BSTreeRemove(&(root->left), item);
    }

    return BSTreeRemove(&(root->right), item);
}
```

- Elemento procurado **não pertence** a uma (sub-)árvore vazia
- Elemento **encontrado**
- **Remover** usando uma função auxiliar
- A que **subárvore** pertence o elemento a remover ?

# Caso 1 – Remover um nó que é uma folha

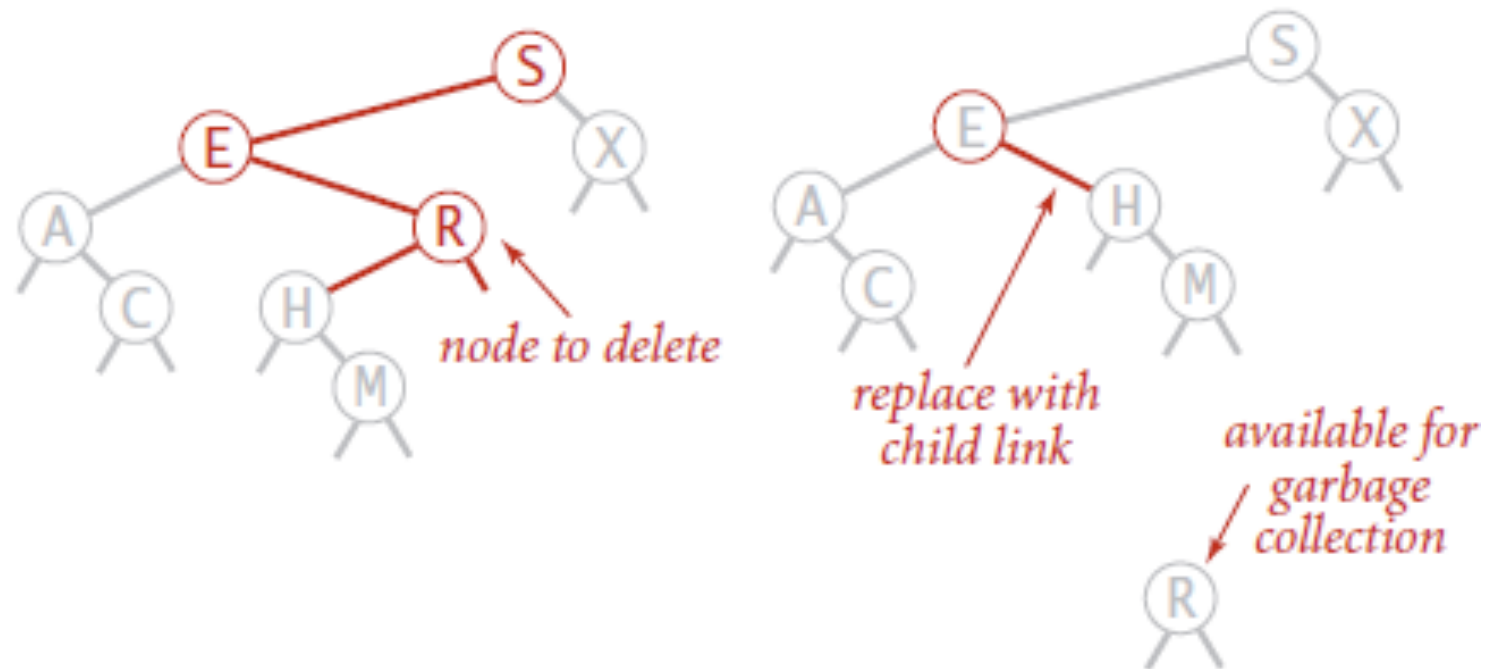
deleting C



[Sedgewick & Wayne]

# Casos 2 e 3 – Remover nó com um só filho

deleting R

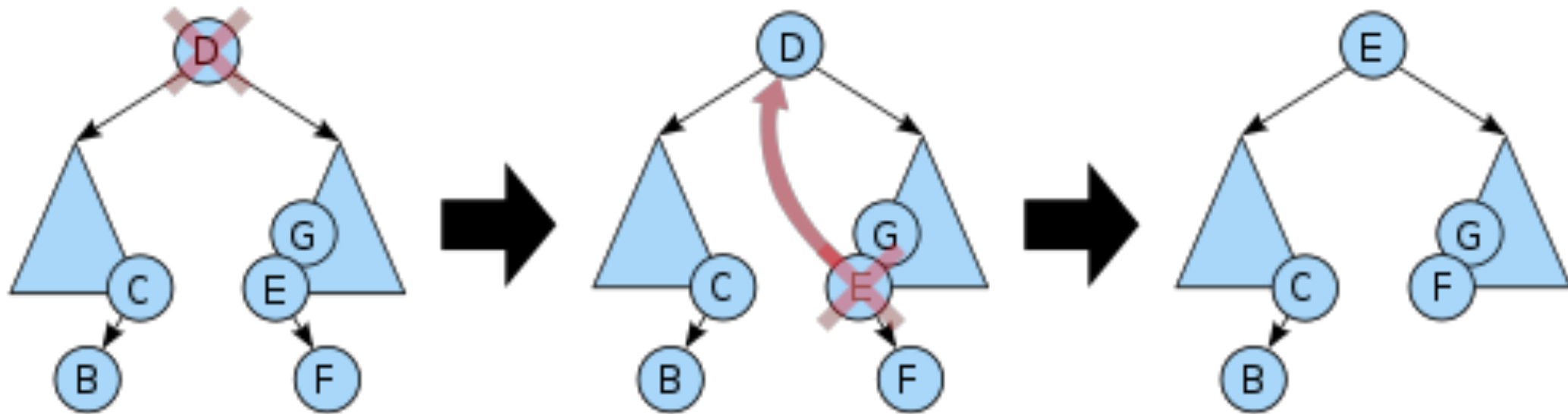


[Sedgewick & Wayne]

## Caso 4 – Remover um nó com dois filhos

- Manter a **ordem** !!
- **Substituir** o **item** do nó pelo seu **sucessor** OU pelo seu **predecessor**
  - Vamos usar o sucessor !!
- **Encontrar** o **sucessor** e copiar o seu valor
- **Substituir** o **item** pelo seu **sucessor**
- **Apagar** o nó do **sucessor** – é o **menor elemento** da **subárvore direita**

# Remove – Substituir pelo **sucessor** e **apagá-lo**



[Wikipedia]

# Tarefa

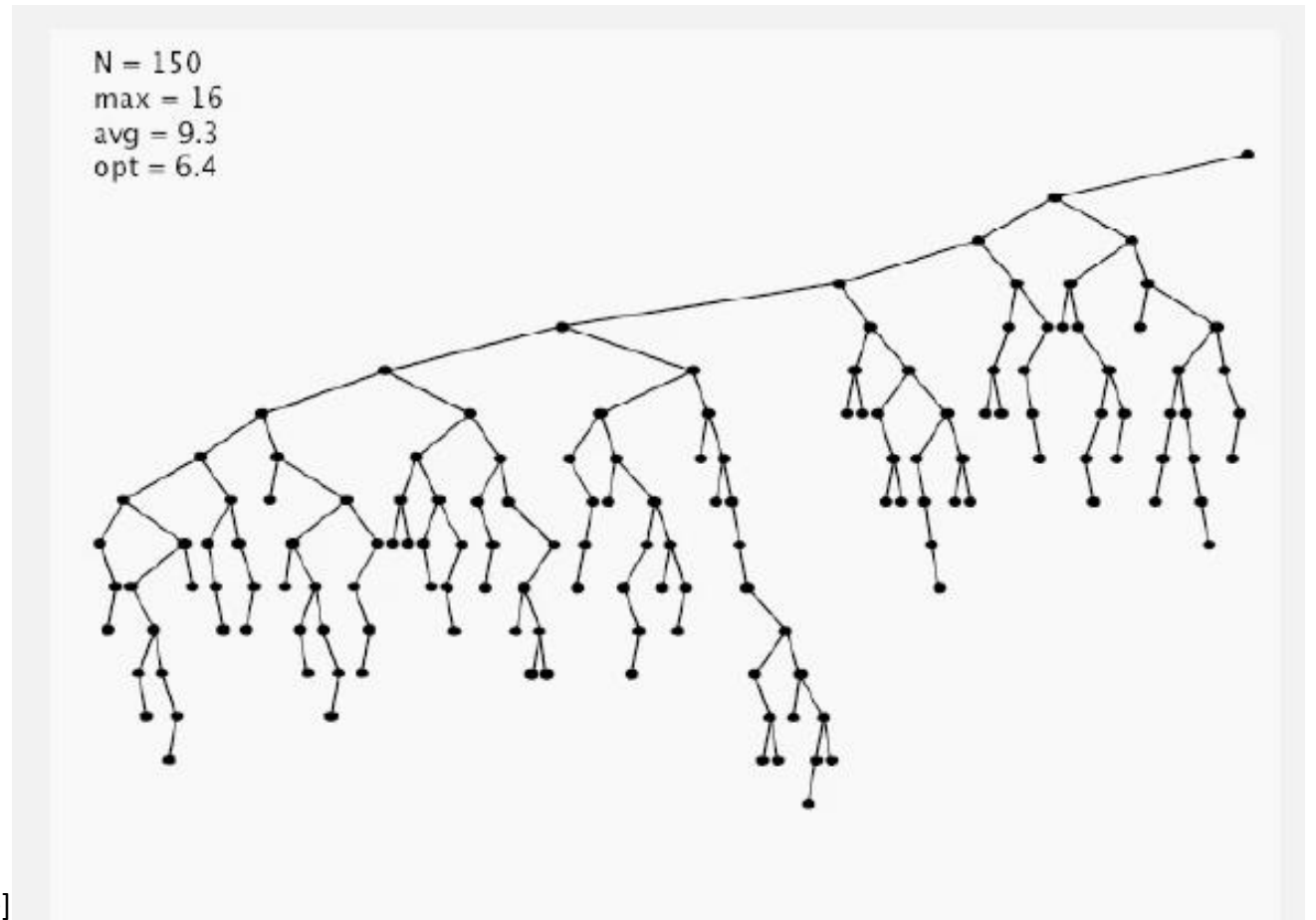
- **Analisar** o código das **funções auxiliares** que permitem remover de uma ABP o elemento procurado



# ABP – Após muitos apagamentos

- Remover muitos elementos, numa ordem qualquer
- Árvore perde alguma “simetria” !!
- Consequência ?
- Mais difícil procurar alguns elementos do que outros

[Sedgewick & Wayne]



# Eficiência Computacional

# Eficiência - Lista ligada / Array ordenado / ABP

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

$h$

$h$

$h$

$h$

$h$

$h$

$N$

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

Para garantir que  
a árvore é equilibrada  
a complexidade é  $O(\log_2 N)$

[Sedgewick & Wayne]

# Eficiência - Lista ligada / Array ordenado / ABP

Pior caso

implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	$N$	$N$	$N$	$\frac{1}{2} N$	$N$	$\frac{1}{2} N$		equals()
binary search (ordered array)	$\lg N$	$N$	$N$	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	compareTo()
BST	$N$	$N$	$N$	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✓	compareTo()

other operations also become  $\sqrt{N}$  if deletions allowed

[Sedgewick & Wayne]

# ABP – Problema – Árvores “desequilibradas”

- Os elementos podem não ser adicionados de modo aleatório
  - Por exemplo, **adição ordenada** !!
- Ou já ter ocorrido um grande número de **apagamentos**
- Como evitar o **pior caso / casos maus** ?
- **Árvores equilibradas** em altura !!
  - São ABPs de **altura “aceitável”**, com **procuras “pouco demoradas”**
  - **Árvores AVL** (1962)
  - **Red-black trees** – Java **TreeMap**

# Árvores equilibradas em altura – Balanced Trees

# Árvores equilibradas em altura

- **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**

# Critério de equilíbrio

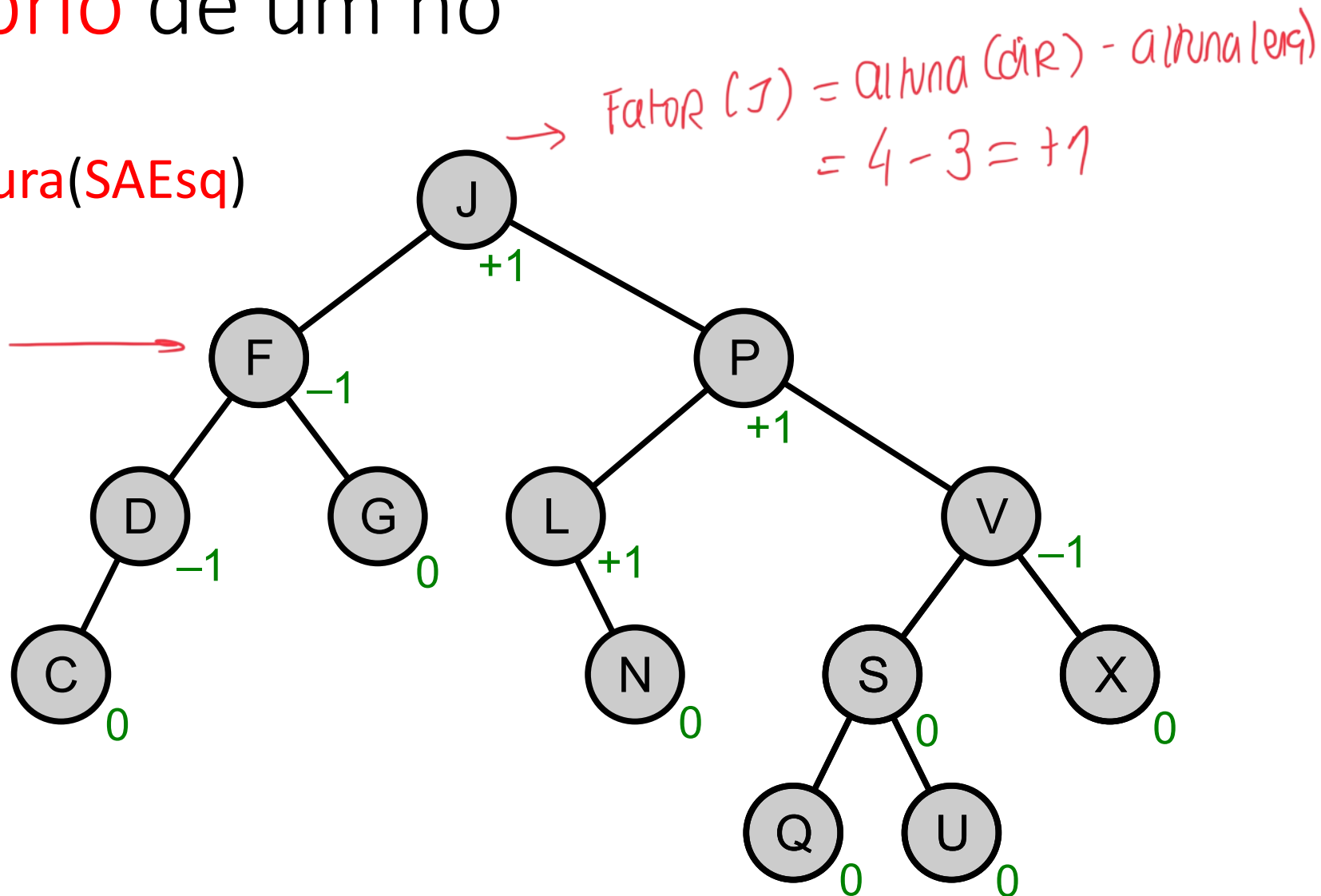
- A altura das duas subárvores de cada nó difere, quando muito, de uma unidade (0 ou  $\pm 1$ )
- Boa ideia !!
- Fácil de verificar e de manter
- Adicionar a cada nó um atributo com a altura da (sub-)árvore da qual é raiz



# Fator de equilíbrio de um nó

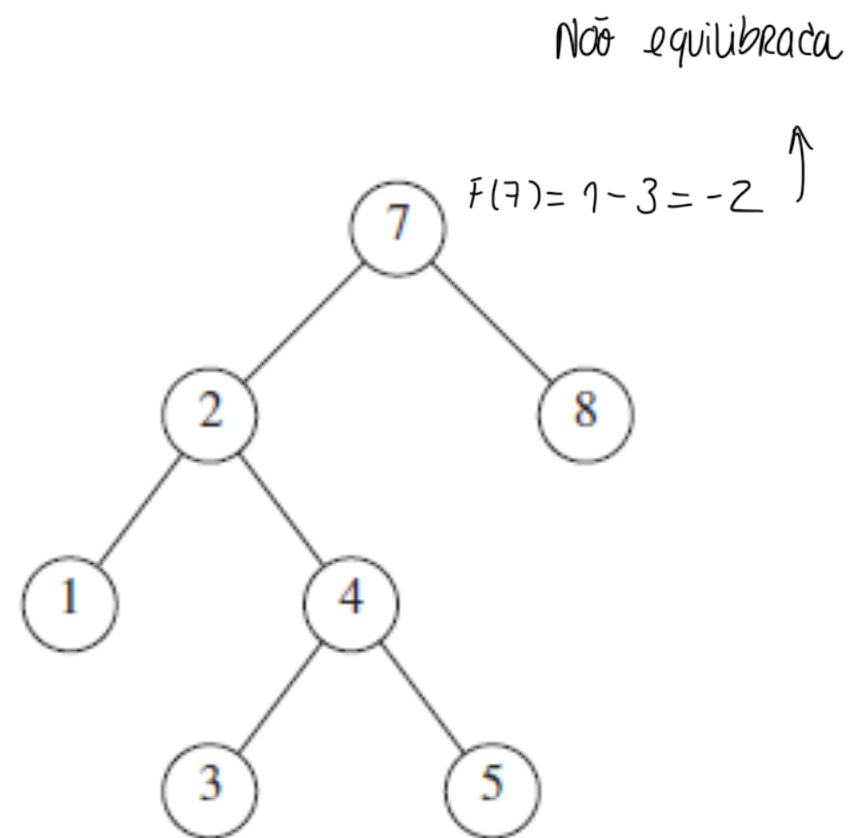
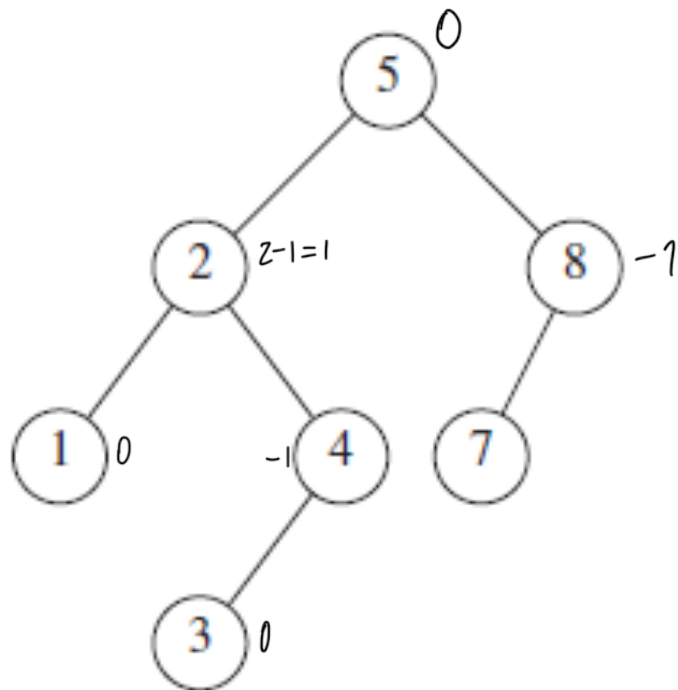
- $F = \text{altura}(\text{SADir}) - \text{altura}(\text{SAEsq})$

$$F(J) = \text{altura}(\text{dir}) - \text{altura}(\text{esq}) \\ = 1 - 2 = -1$$



[Wikipedia]

# Árvores equilibradas ?



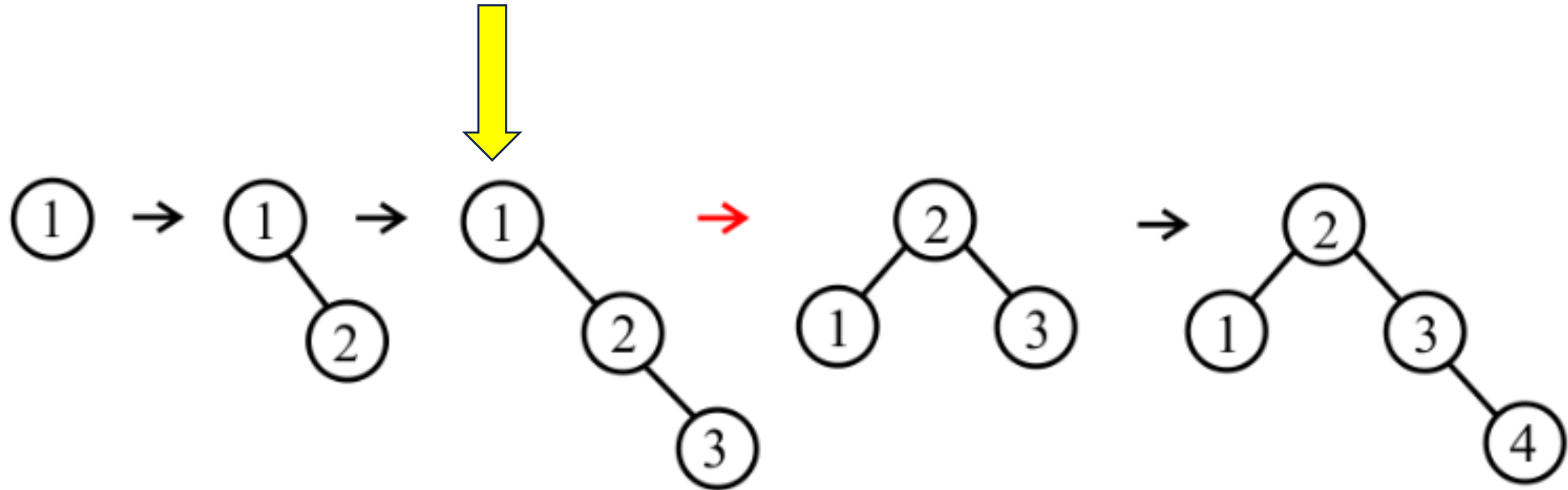
[Weiss]

# Árvores AVL – Árvores de Adelson-Velskii e Landis

# Árvores AVL – Manter altura equilibrada

- 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 o critério de equilíbrio **falha** !!
- MAS, **manter** o **critério de ordem** da ABP !!
- Basta fazer a verificação / reposicionamento ao longo do **caminho** entre a **raiz** e um **nó** que tenha sido “alterado” – **traceback**

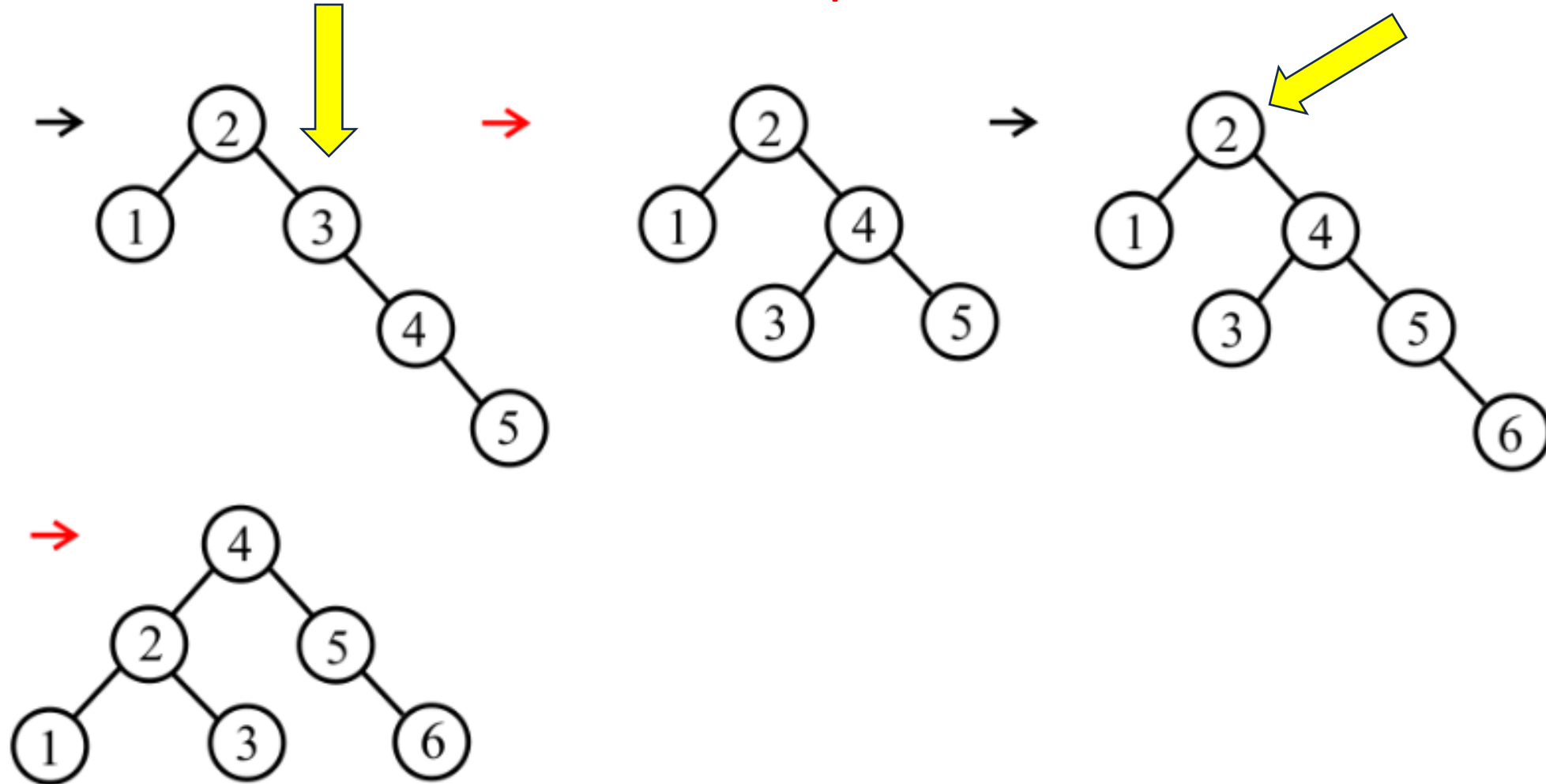
# Árvore AVL – Inserir + Equilibrar, se necessário



[cs.ecu.edu]

- Após **adicionar** o **nó 3**, o fator de equilíbrio do **nó 1** é 2, e esse nó **falha** o **critério de equilíbrio**
- **Nó 1** é **reposicionado**

# Árvore AVL – Inserir + Equilibrar, se necessário



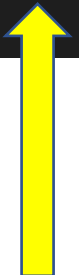
[cs.ecu.edu]


# Fator de equilíbrio

- Para cada nó de uma árvore equilibrada em altura
- As duas subárvores têm a mesma altura
- Ou a sua altura difere de 1
- $F = \text{Altura(SADireita)} - \text{Altura(SAEsquerda)}$
- $F = -1, 0, 1$
- 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**” !!

# AVL – Altura de um nó da árvore

```
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 altura após inserir / remover um nó

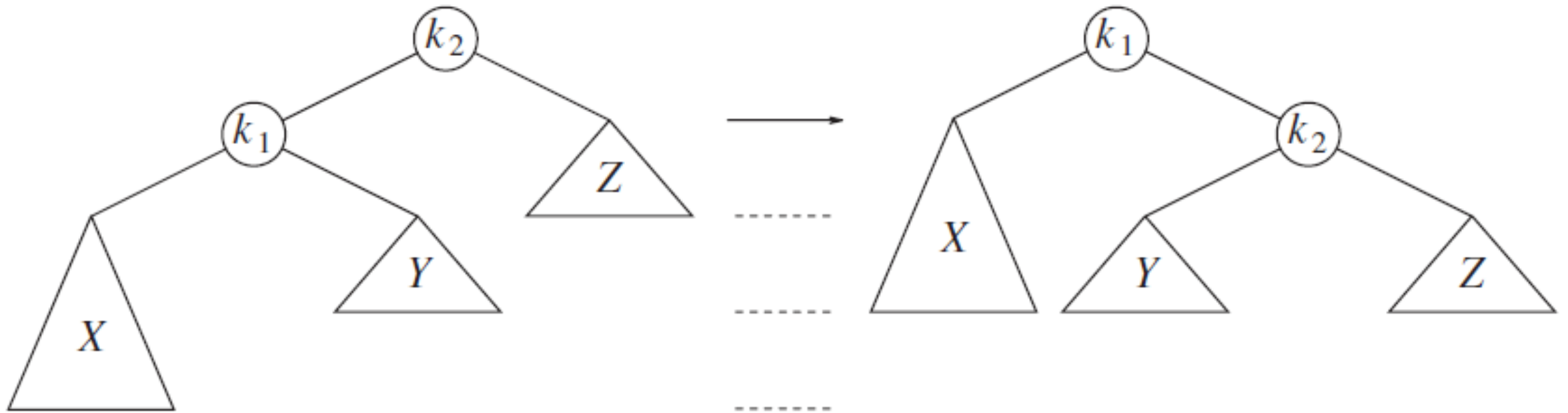
```
static void _updateNodeHeight(AVLTree* t) {  
    assert(t != NULL);  
  
    int leftHeight = AVLTreeGetHeight(t->left);  
  
    int rightHeight = AVLTreeGetHeight(t->right);  
  
    if (leftHeight >= rightHeight) {  
        t->height = leftHeight + 1;  
    } else {  
        t->height = rightHeight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



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

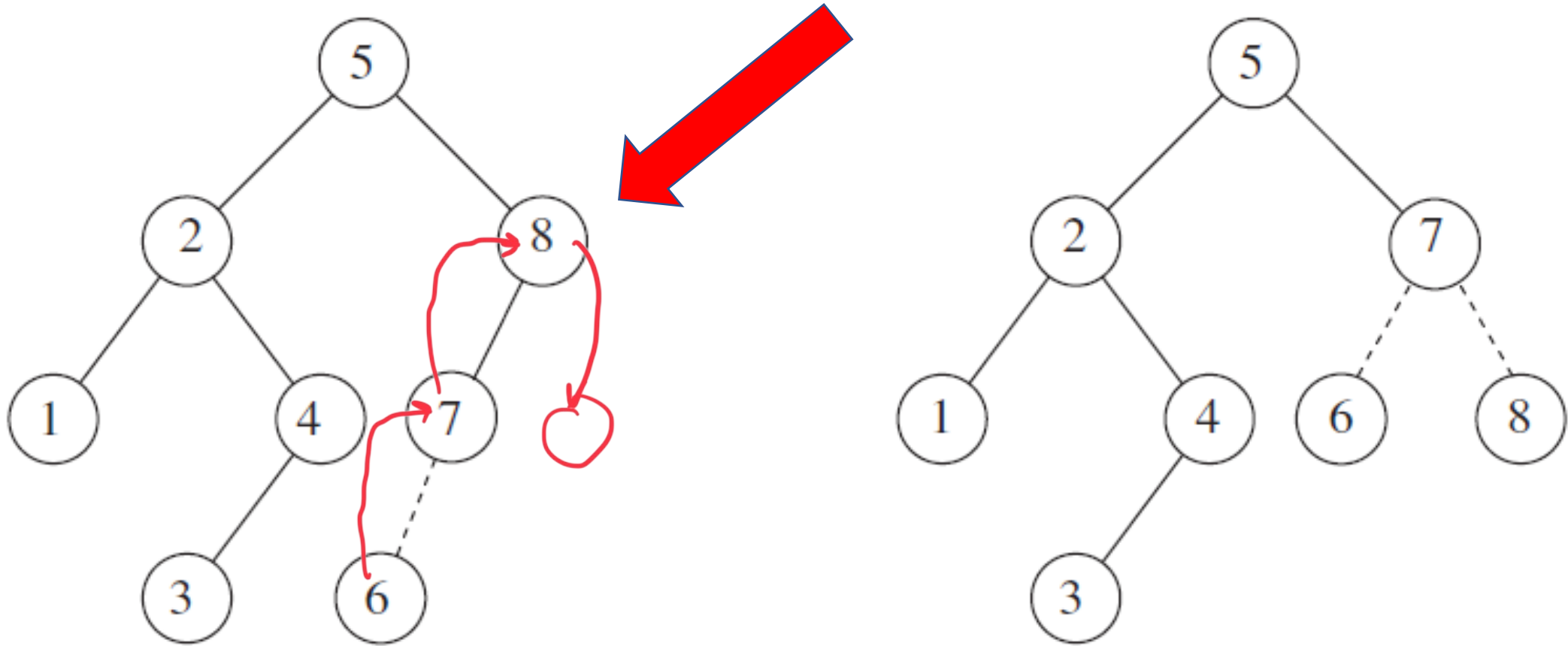
- Efetuando **operações de rotação** – 4 possibilidades
- MAS, assegurando o **critério de ordem** das ABPs
- Apenas **troca de ponteiros** para termos **operações rápidas** !
- **Rotações simples** à esquerda ou à direita
- **Rotações duplas** à esquerda ou à direita
  - Sequência de duas rotações simples

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



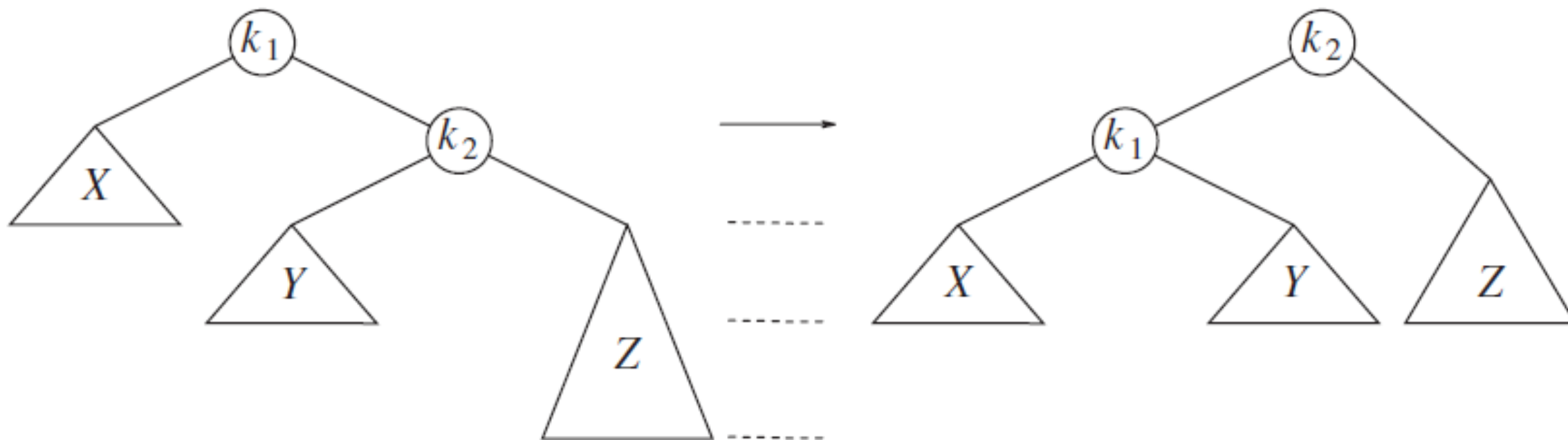
[Weiss]

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



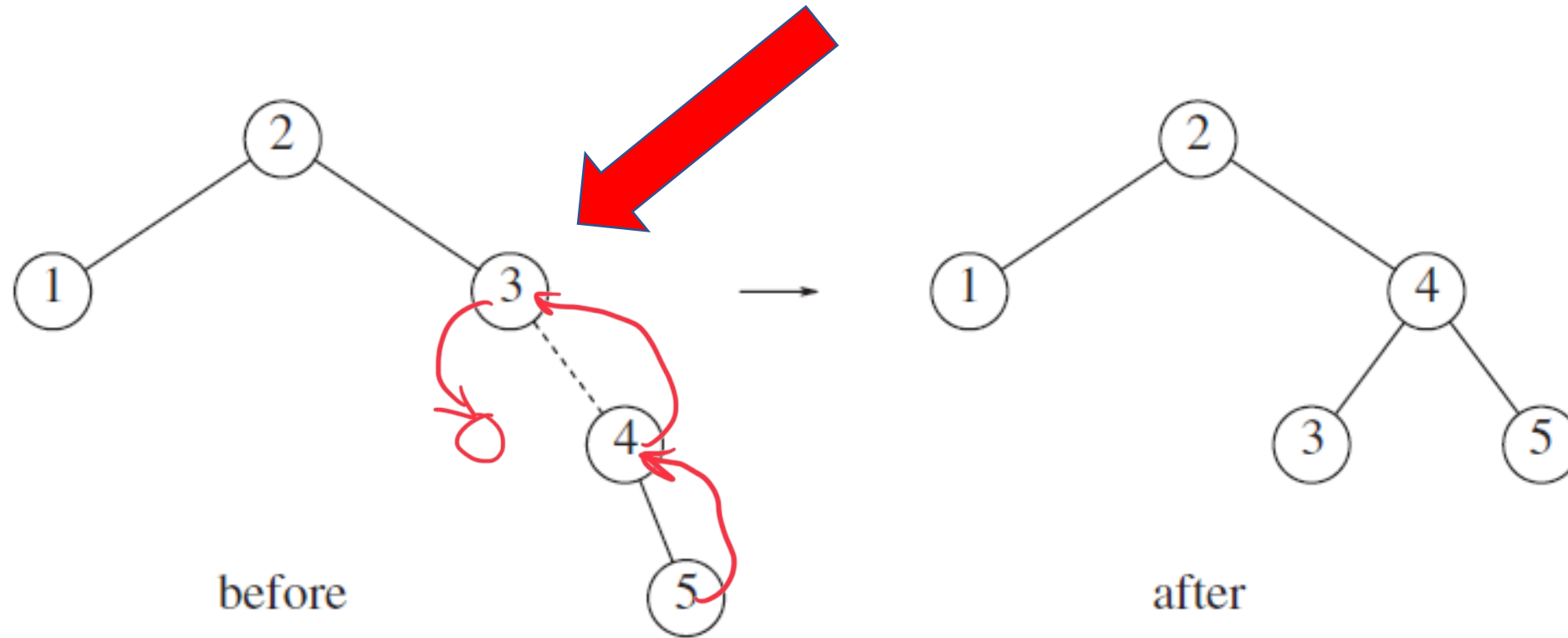
[Weiss]

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



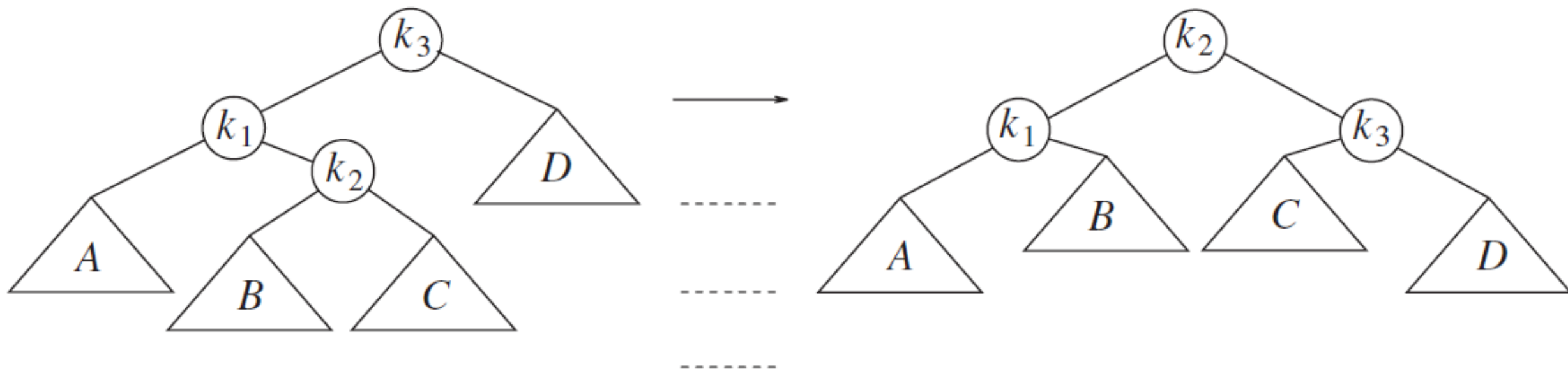
[Weiss]

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



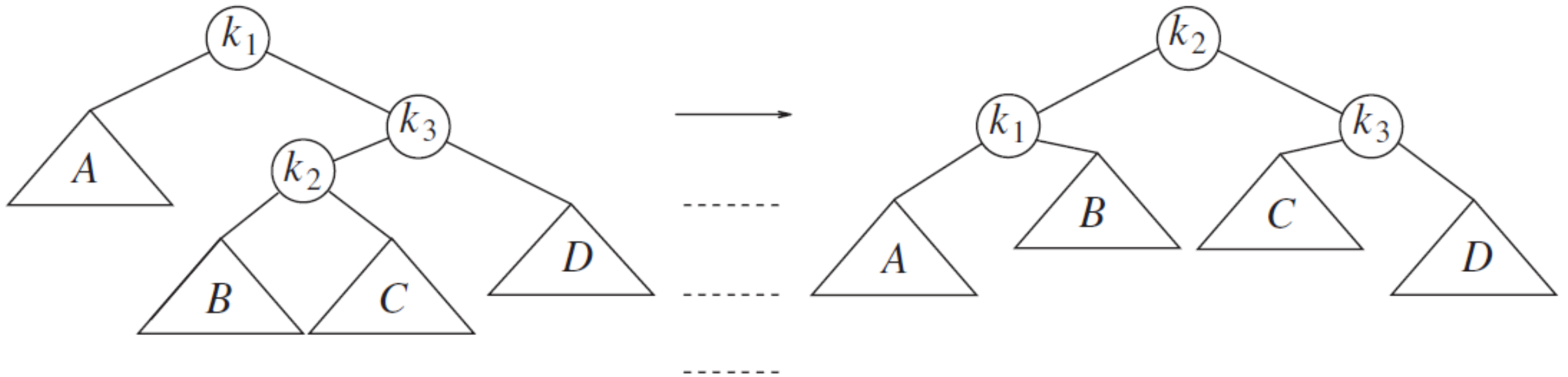
[Weiss]

# Rotação dupla à esquerda – Como identificar?



[Weiss]

# Rotação dupla à direita – Como identificar?



[Weiss]



# AVL – Inserir um novo nó e equilibrar

- 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

# AVL – Remover um nó e equilibrar

- 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 ?

# Árvores ABP **vs** Árvores AVL

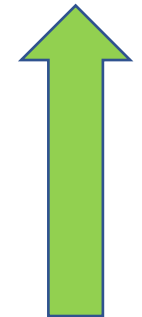
## – Experiências computacionais

# Eficiência – 1ª experiência computacional

- **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

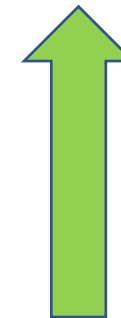
# Procurar os sucessivos números pares

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

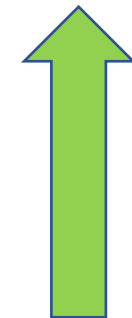
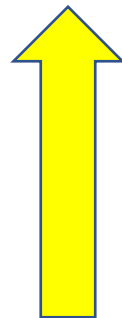
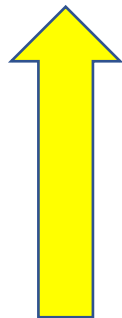


# Eficiência – 2ª experiência computacional

- 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







# Exercícios / Tarefas

# Exercício 1 – Escolha múltipla

Seja dada uma árvore binária armazenando, de modo **não-ordenado**,  $n$  números inteiros.

- a) Determinar o menor elemento armazenado na árvore é uma operação de complexidade  $O(\log n)$ .  $\rightarrow O(n)$
- ☒ b) No pior caso, verificar se um dado número pertence à árvore é uma operação de complexidade  $O(n)$ .
- ☒ c) Determinar a altura da árvore é uma operação de complexidade  $O(\log n)$ .
- d) Todas estão corretas.

## Exercício 2 – Escolha múltipla

Seja dada uma **árvore binária** de altura **equilibrada** que armazena, de modo **ordenado**,  $n$  números inteiros.

- ✓ a) No pior caso, determinar o valor do menor elemento armazenado na árvore é uma operação de complexidade  $O(\log n)$ .
- ✓ b) No pior caso, concluir que um dado número não pertence à árvore é uma operação de complexidade  $O(\log n)$ .
- ☒ c) Ambas estão corretas.
- d) Nenhuma está correta.

# Tarefa 1 : ABP – Adicionar um elemento

- Desenvolva uma função **recursiva** que adiciona um **novo elemento** a uma **Árvore Binária de Procura (ABP) / Binary Search Tree (BST)** que armazena números inteiros
- Se esse elemento **já pertence** à árvore, deve ser indicado que a operação falhou

```
int AddElement ( Bstree ** root, int Element ) {
```

```
    Bstree* root = *pRoot;
```

```
    if ( root == NULL ) {
```

```
        root = (Bstree *) malloc (size of (Bstree));
```

```
        root → item = element;
```

```
        root → right = root → left = NULL;
```

```
    {
```

```
        if ( root → item == element ) {
```

```
            free (root)
```

```
            return -1;
```

```
        {
```

```
            if ( element < root → item ) return AddElement ( &(root → left), element );
```

```
            Return AddElement ( &(root → right), element );
```

```
    }
```

## Tarefa 2 : ABP – Remover o menor elemento

- Desenvolva uma função **recursiva** que remove o **menor elemento** de uma **Árvore Binária de Procura (ABP) / Binary Search Tree (BST)** que armazena números inteiros

```
int RemoveSmallest Element ( Bstree** pRoot ) {
```

```
    Bstree* root = *pRoot;
```

```
    if (root == NULL) return -1;
```

```
    if ( root->left == NULL ) {
```

```
        _RemoveNode (pRoot);
```

```
        return 1;
```

```
{
```

```
    return RemoveSmallest Element (&(root->left));
```

## Tarefa 3 : ABP – Remover o maior elemento

- Desenvolva uma função **recursiva** que remove o **maior elemento** de uma **Árvore Binária de Procura (ABP) / Binary Search Tree (BST)** que armazena números inteiros

Menma coisa que a função de cima mas para a direita.