

Árvores Binárias III

13/11/2023

Ficheiro ZIP

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

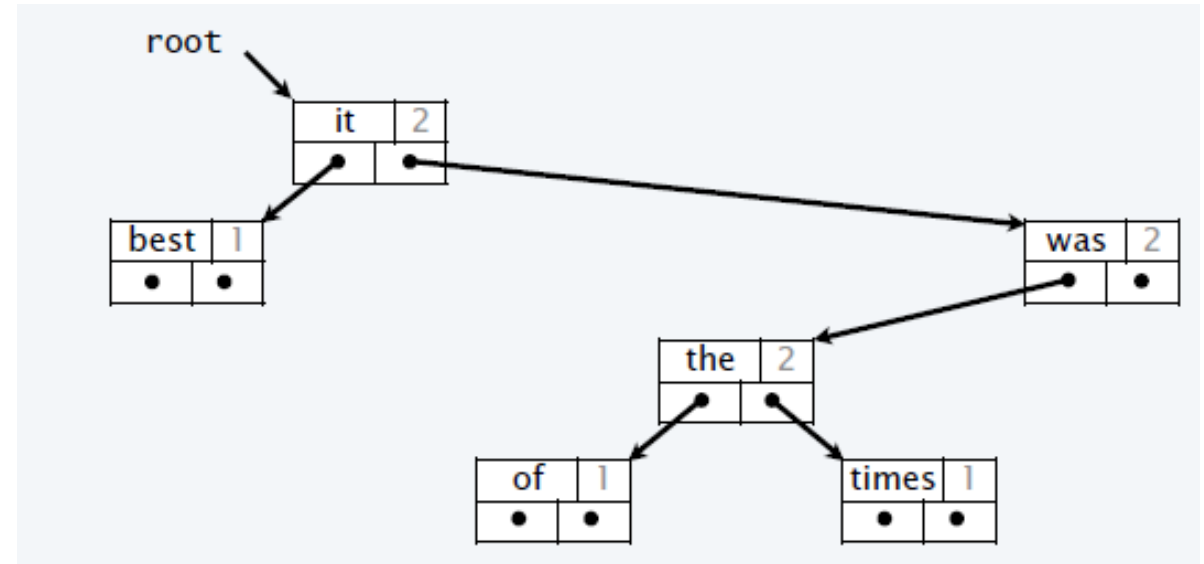
Sumário

- 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

Árvores Binárias de Procura (ABP) – Binary Search Trees (BST)

Critério de ordem – Definição recursiva

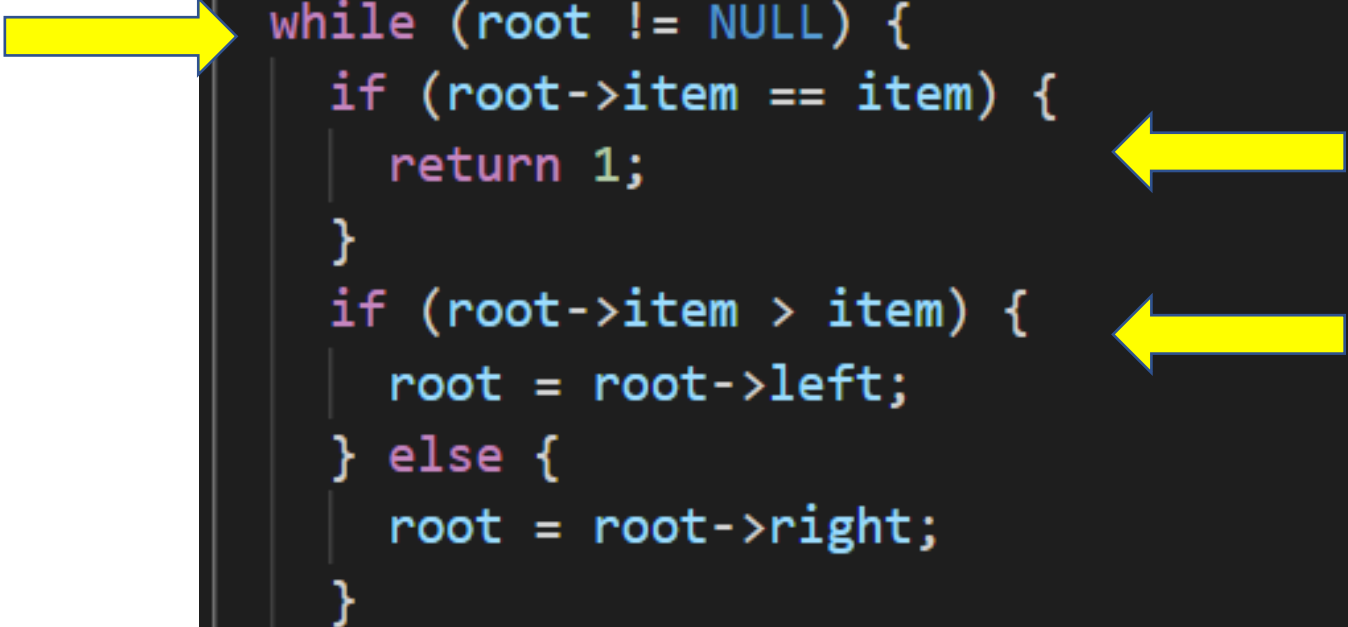
- Para cada nó, os elementos da sua **subárvore esquerda** são **inferiores** ao nó
- E os elementos da sua **subárvore direita** são **superiores** ao nó
- **Não** há elementos **repetidos** !!
- A organização da árvore depende da **sequência de inserção** dos elementos



[Sedgewick & Wayne]

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;  
}
```

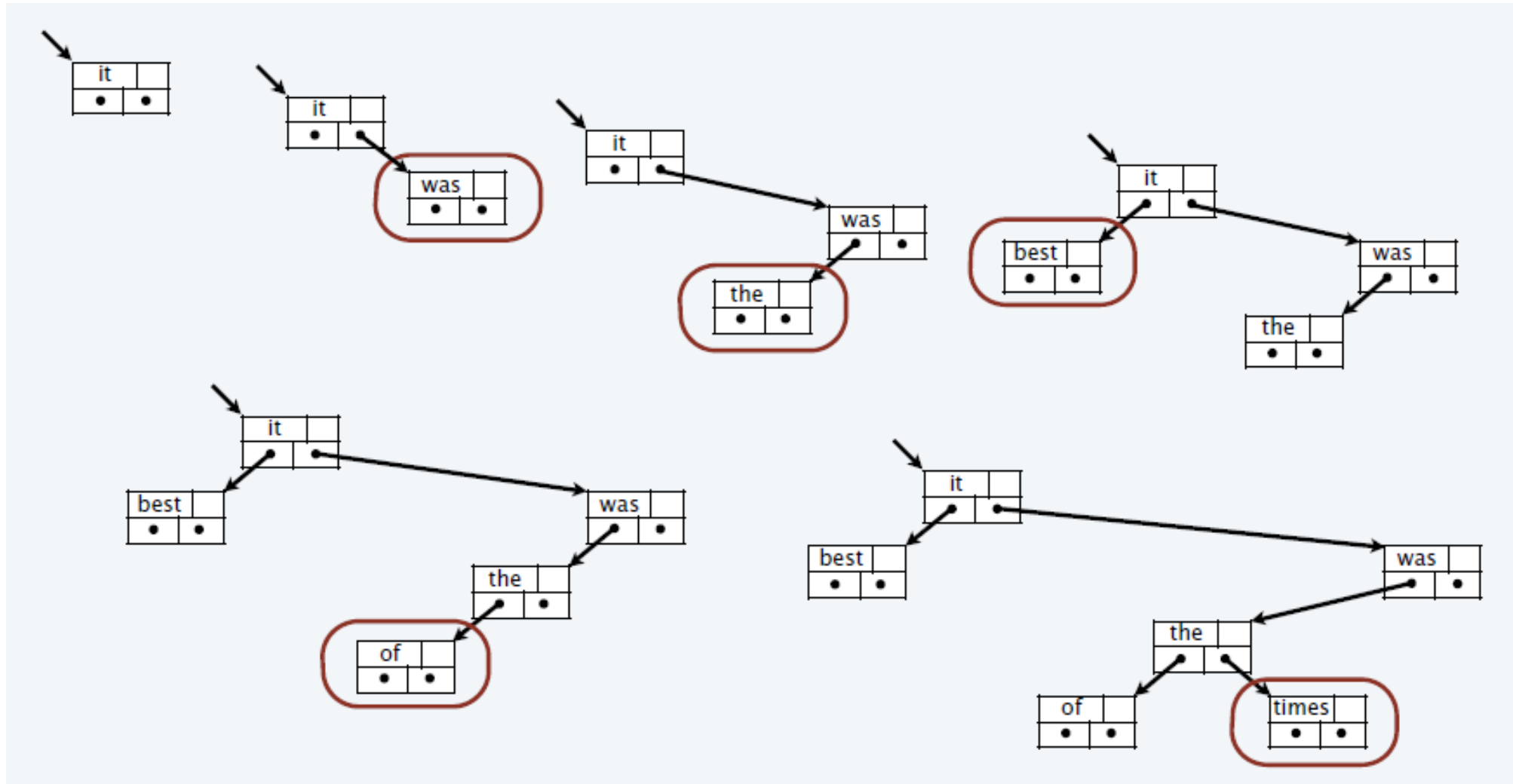


Adicionar um item

Adicionar

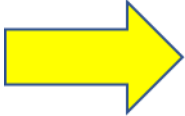
- Restrição : não adicionar duplicados !!
- Restrição : manter a ordem dos itens !!
- Como fazer ?
- Inserir o novo item como folha da árvore, na posição correta

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


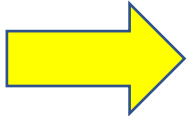


[Sedgewick & Wayne]

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;  
    }  
}
```

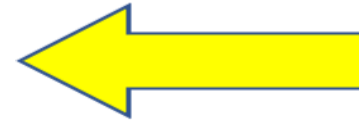


Adicionar – Procurar a posição

```
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;
    }
}
```



Adicionar – Ancorar o novo nó como folha

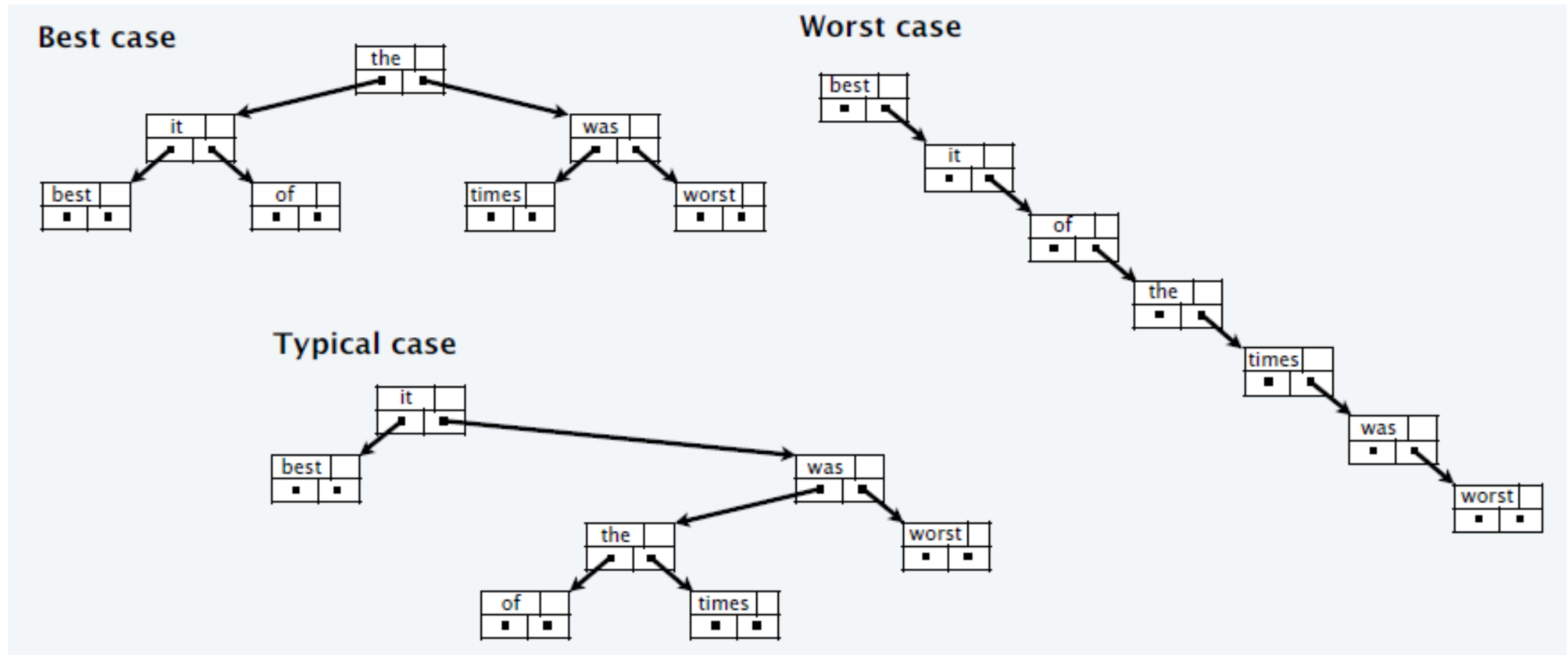


```
if (prev->item > item) {  
    prev->left = new;  
} else {  
    prev->right = new;  
}  
return 0;  
}
```

Adicionar – Versão recursiva

- Tarefa
- Versão recursiva

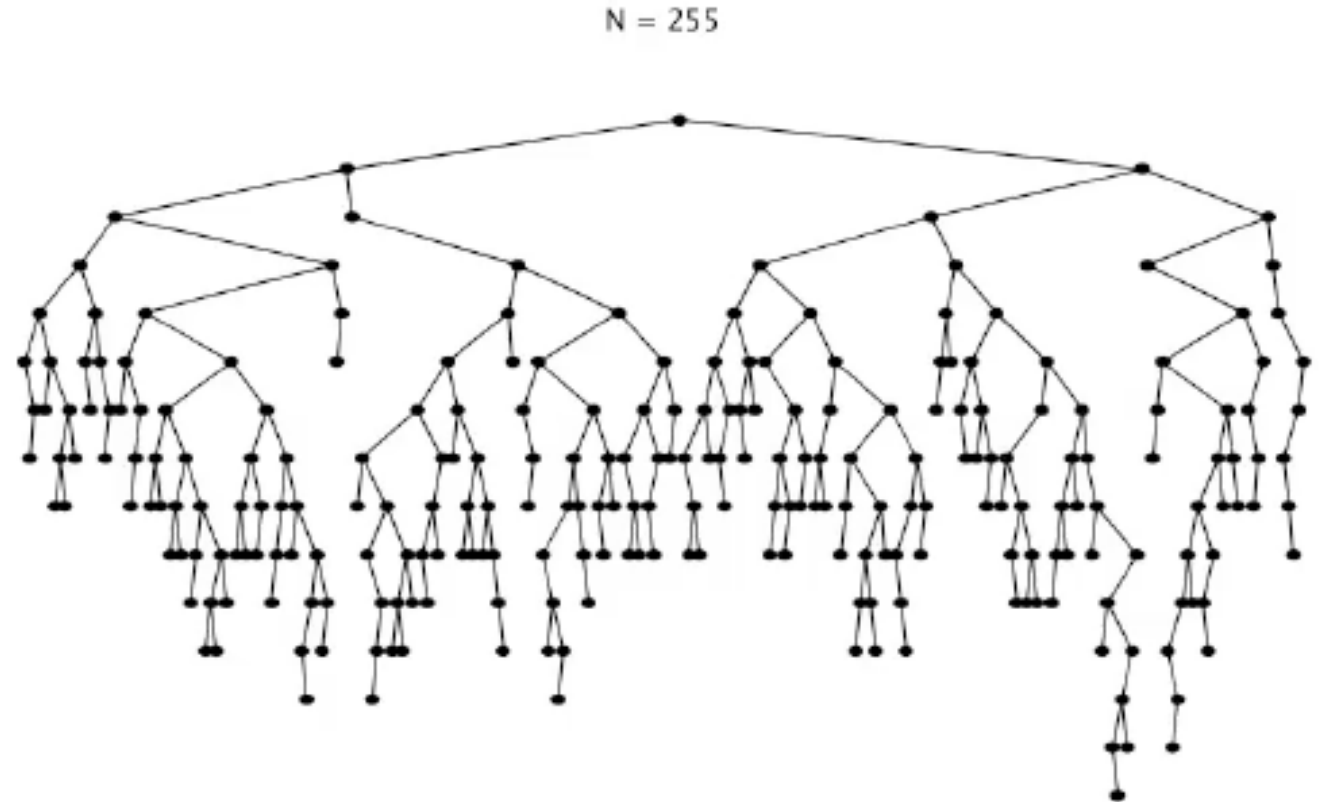
Altura – Diferentes sequências de inserção



[Sedgewick & Wayne]

Altura – Adição numa ordem aleatória

- Árvore **aprox.** equilibrada



[Sedgewick & Wayne]

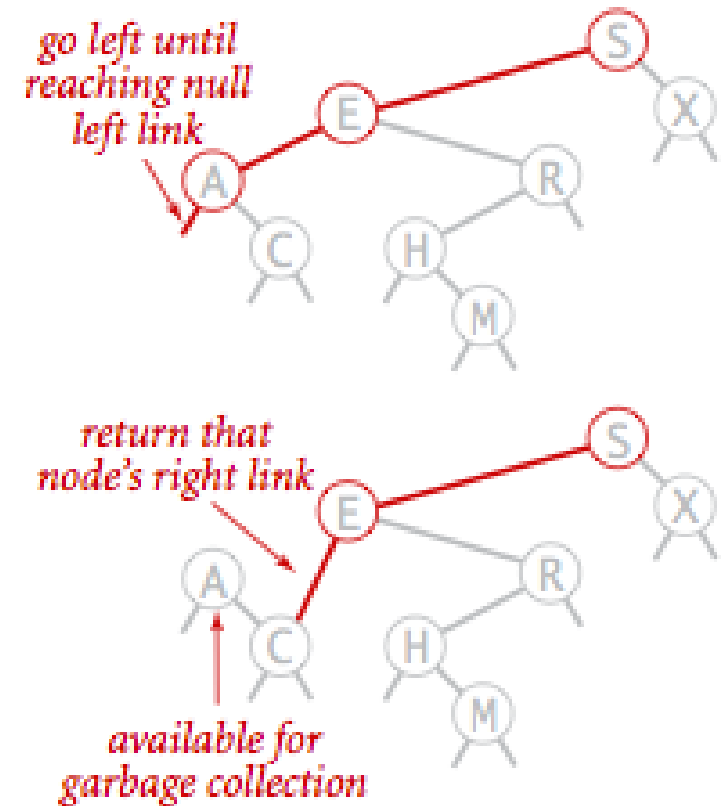
Remover um item

Remove

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

Remover 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]

Remover o menor elemento

- TAREFA : fazer uma função recursiva

Remover o maior elemento

- TAREFA : fazer uma função recursiva

Remover – A estratégia de Hibbard

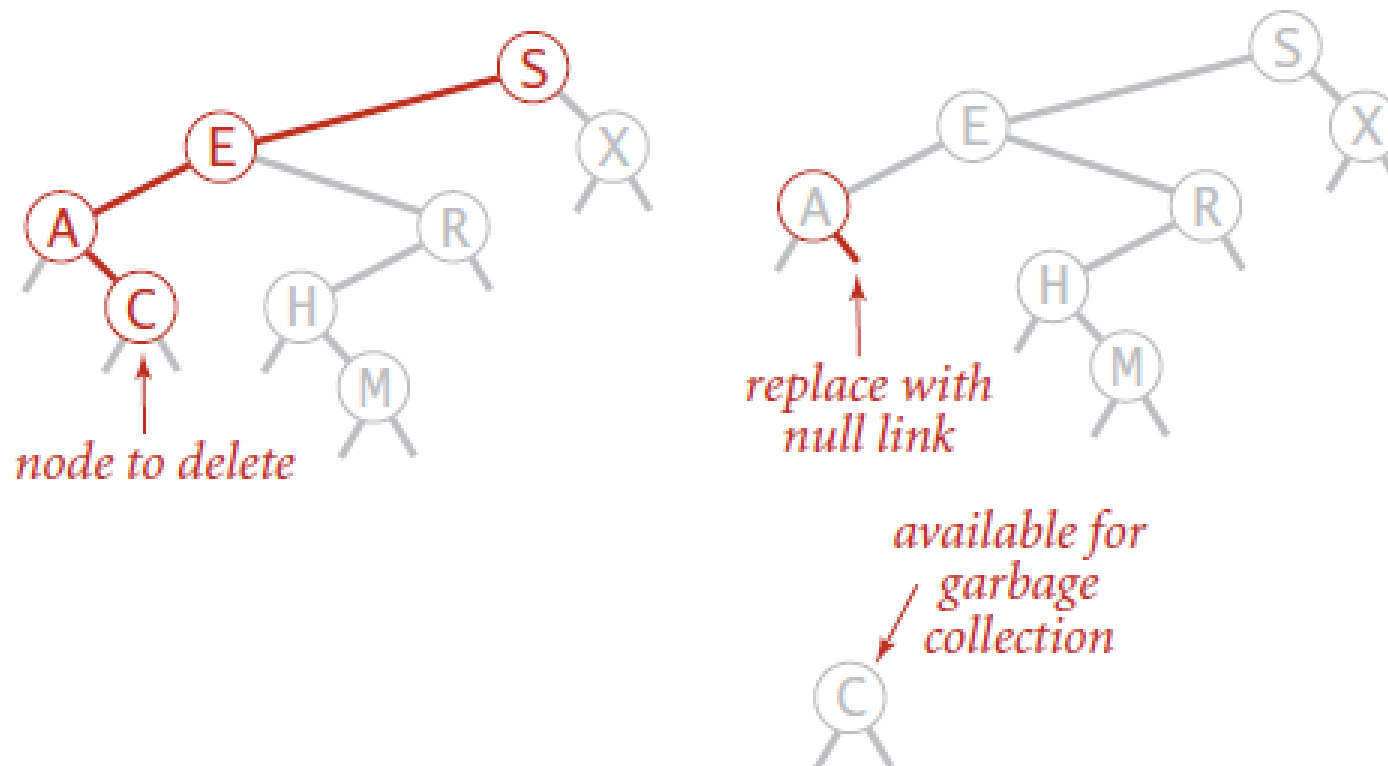
- Dado **item a remover**, **procurar** o respetivo **nó**
- **Caso 1** : é uma **folha** – FÁCIL !!
- **Caso 2** : só tem **subárvore esquerda** – FÁCIL !!
- **Caso 3** : só tem **subárvore direita** – FÁCIL !!
- **Caso 4** : tem **2 subárvores**

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);  
}
```

Remover um nó que é **uma folha**

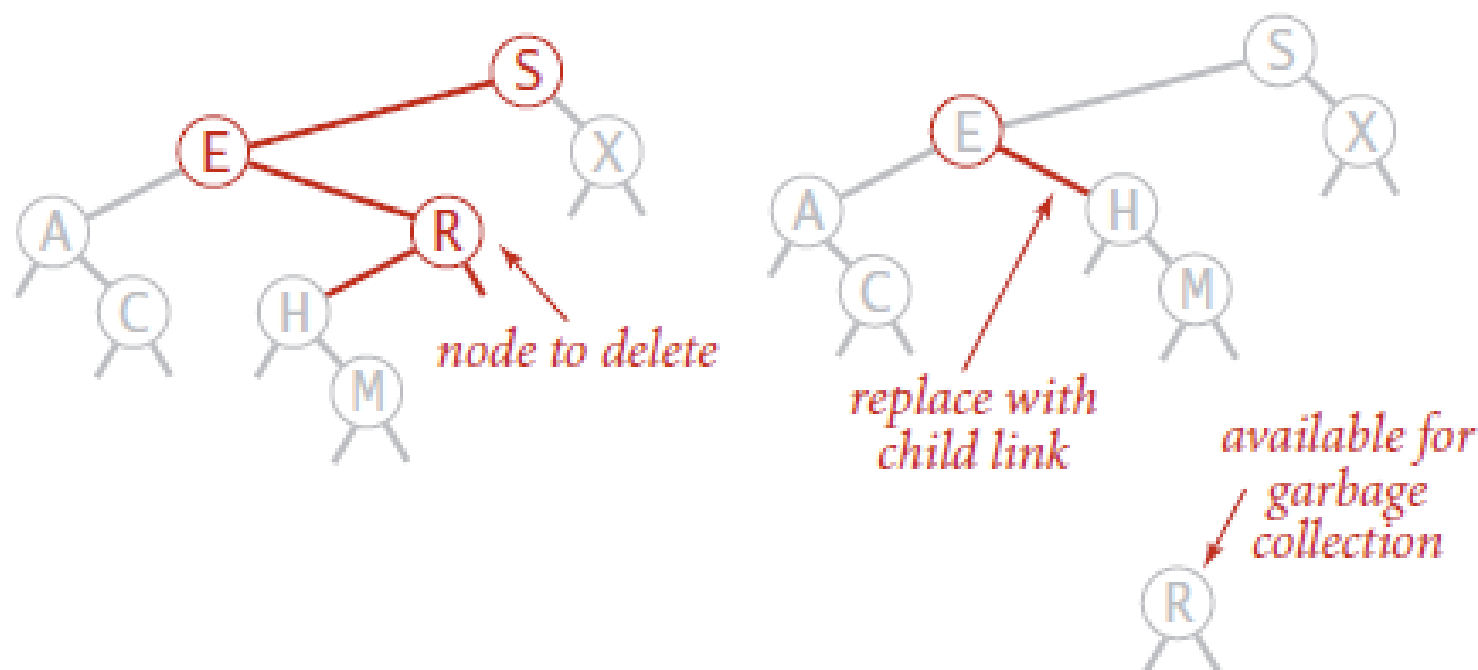
deleting C



[Sedgewick & Wayne]

Remover um **nó** que só tem um filho

deleting R

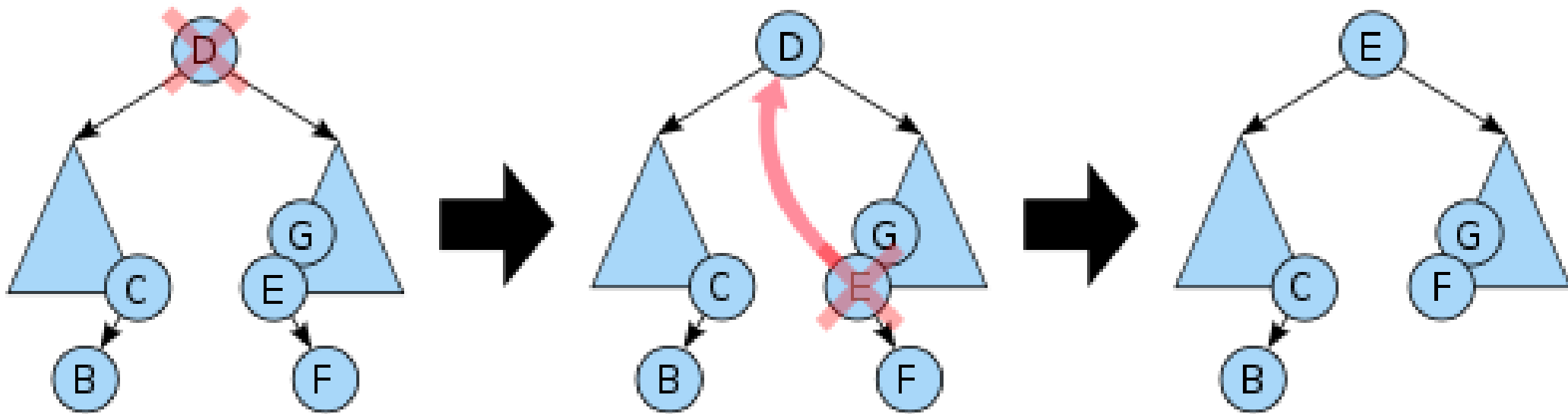


[Sedgewick & Wayne]

Remover um nó que tem dois filhos

- Manter a **ordem** !!
- **Substituir** o item do nó pelo seu **predecessor** OU pelo seu **sucessor**
 - 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** da **subárvore direita**

Substituir pelo sucessor e apagá-lo



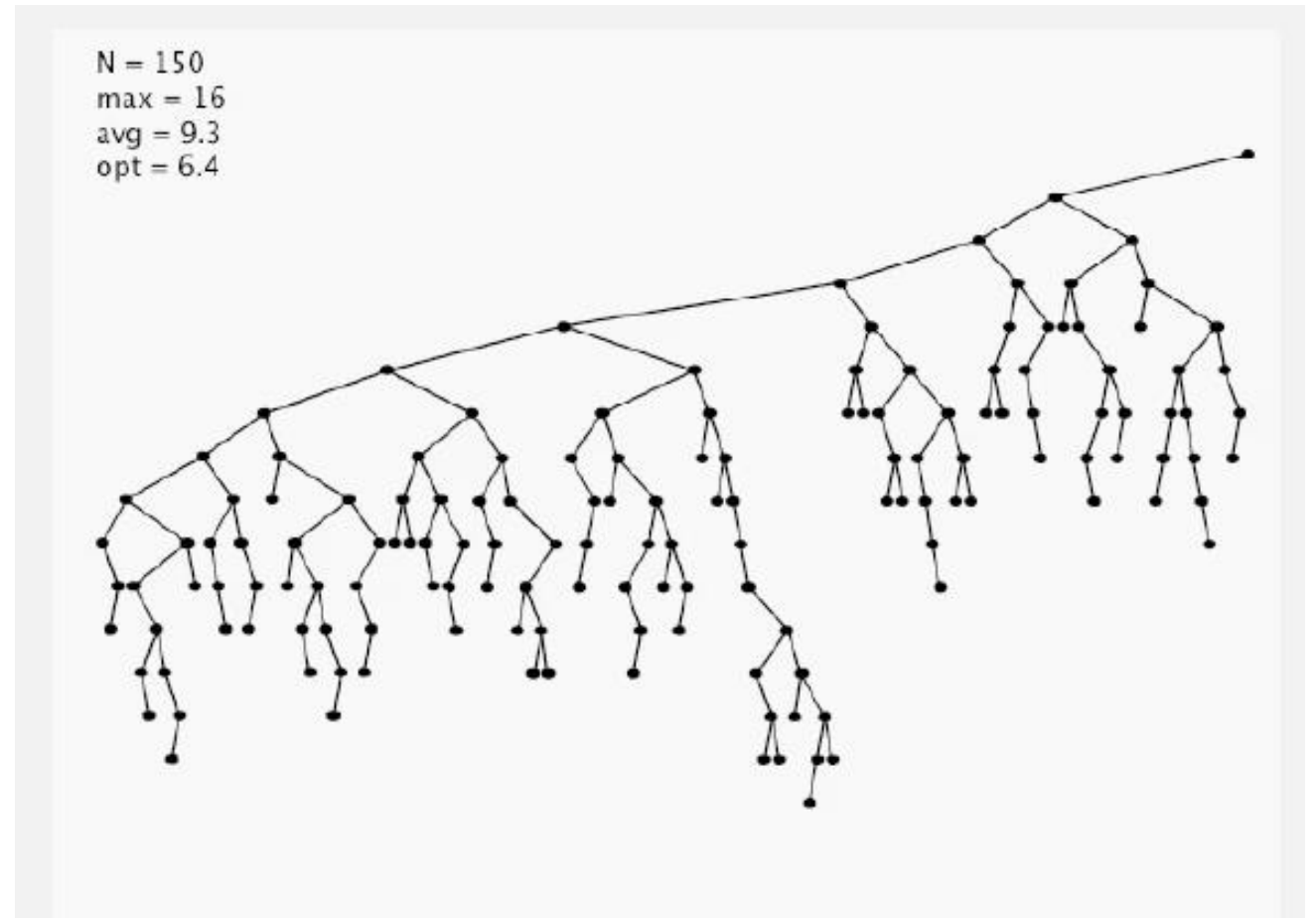
[Wikipedia]

Tarefa

- **Analisar** o código das **funções auxiliares** que efetuam a remoção

Após muitos apagamentos

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




[Sedgewick & Wayne]

Eficiência Computacional

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]

Lista ligada / Array ordenado / ABP

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]

ABPs – Problema

- 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

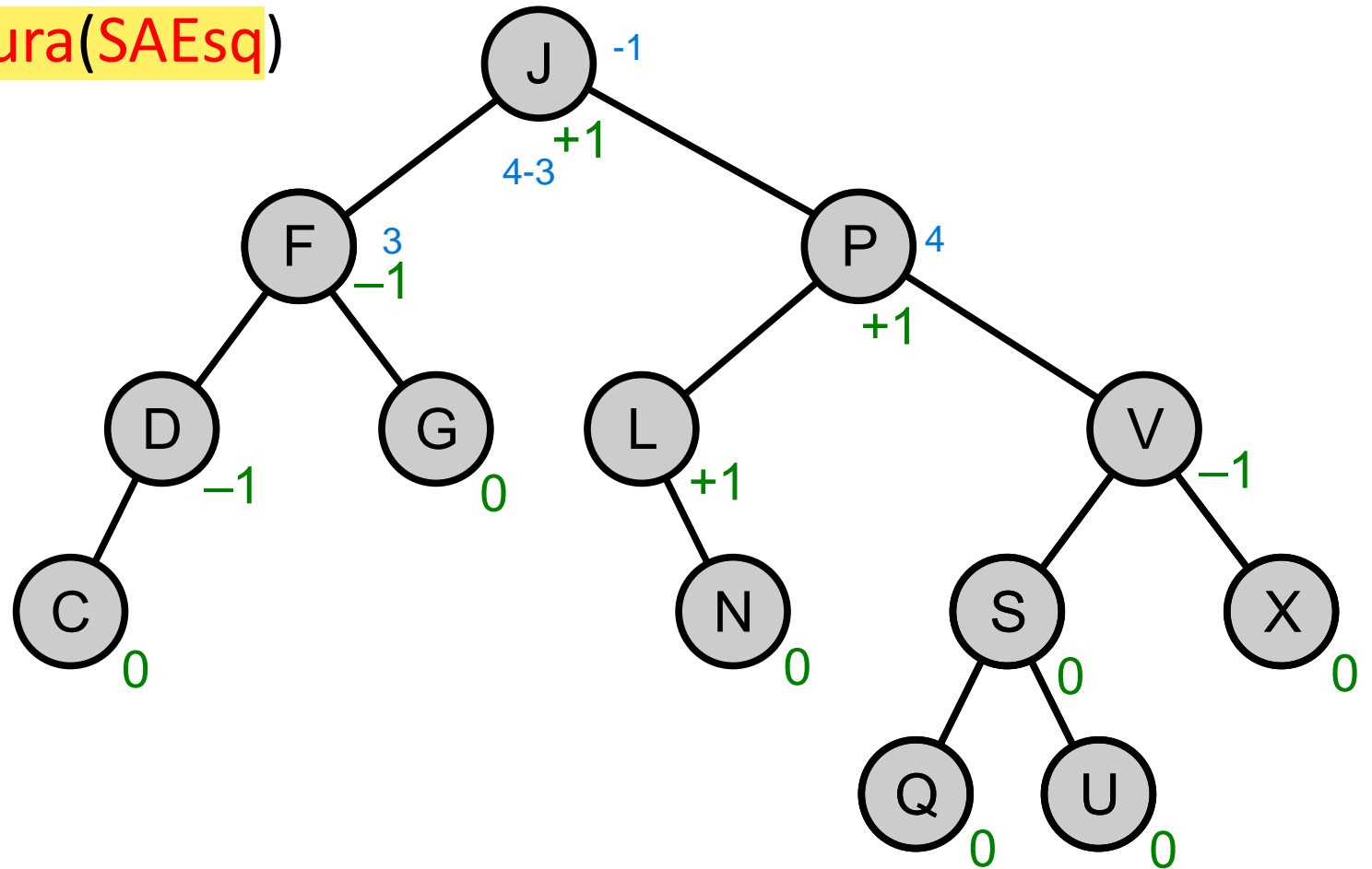
- **Esforço** computacional das operações habituais sobre ABPs depende do **comprimento do caminho** a partir da raiz da árvore
- **Evitar** que uma ABP tenha uma **altura “exagerada”**, para assegurar um bom “comportamento” – **Altura $\in O(\log n)$**
- **O que fazer ?**
- Assegurar que, para cada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

Critério de equilíbrio

- 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 atributo com a altura da sua (sub-)árvore

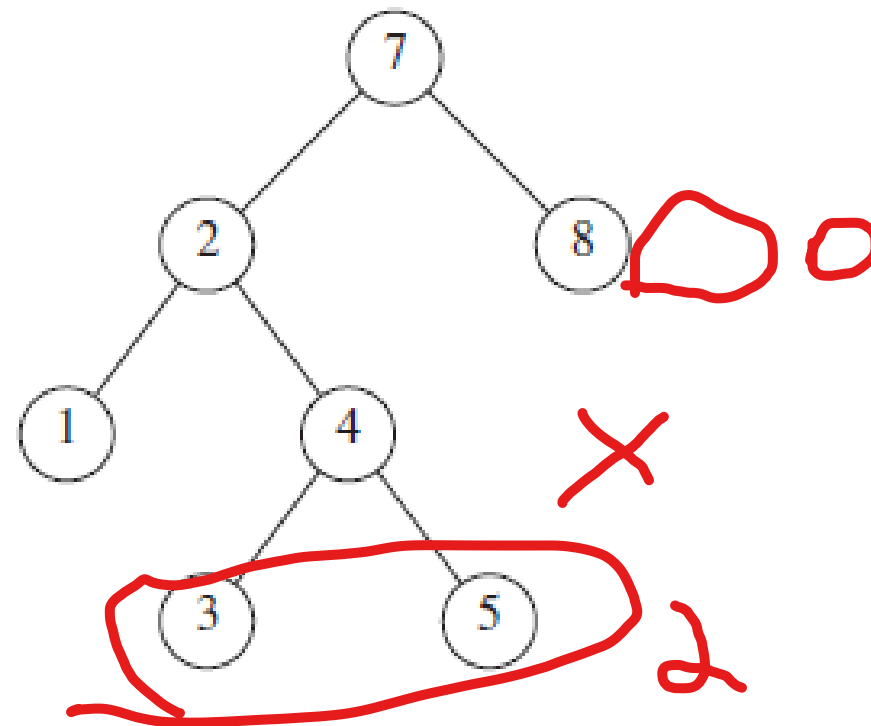
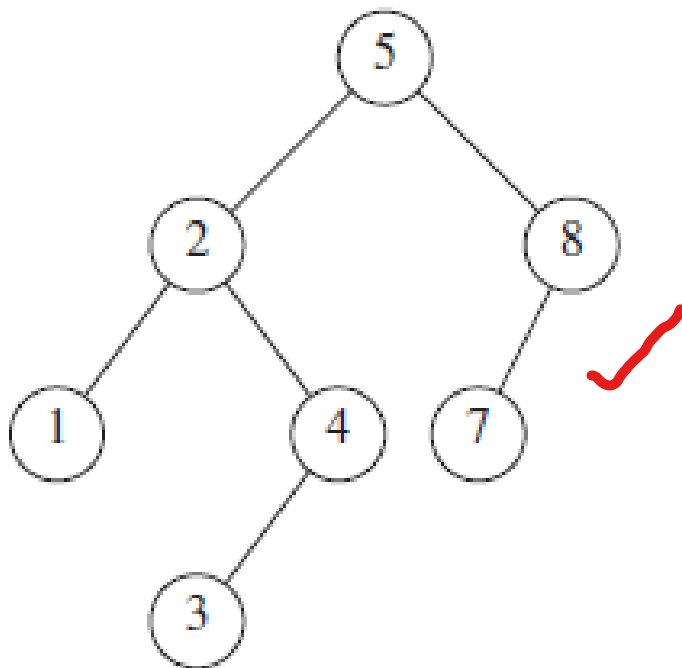
Fator de equilíbrio de um nó

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



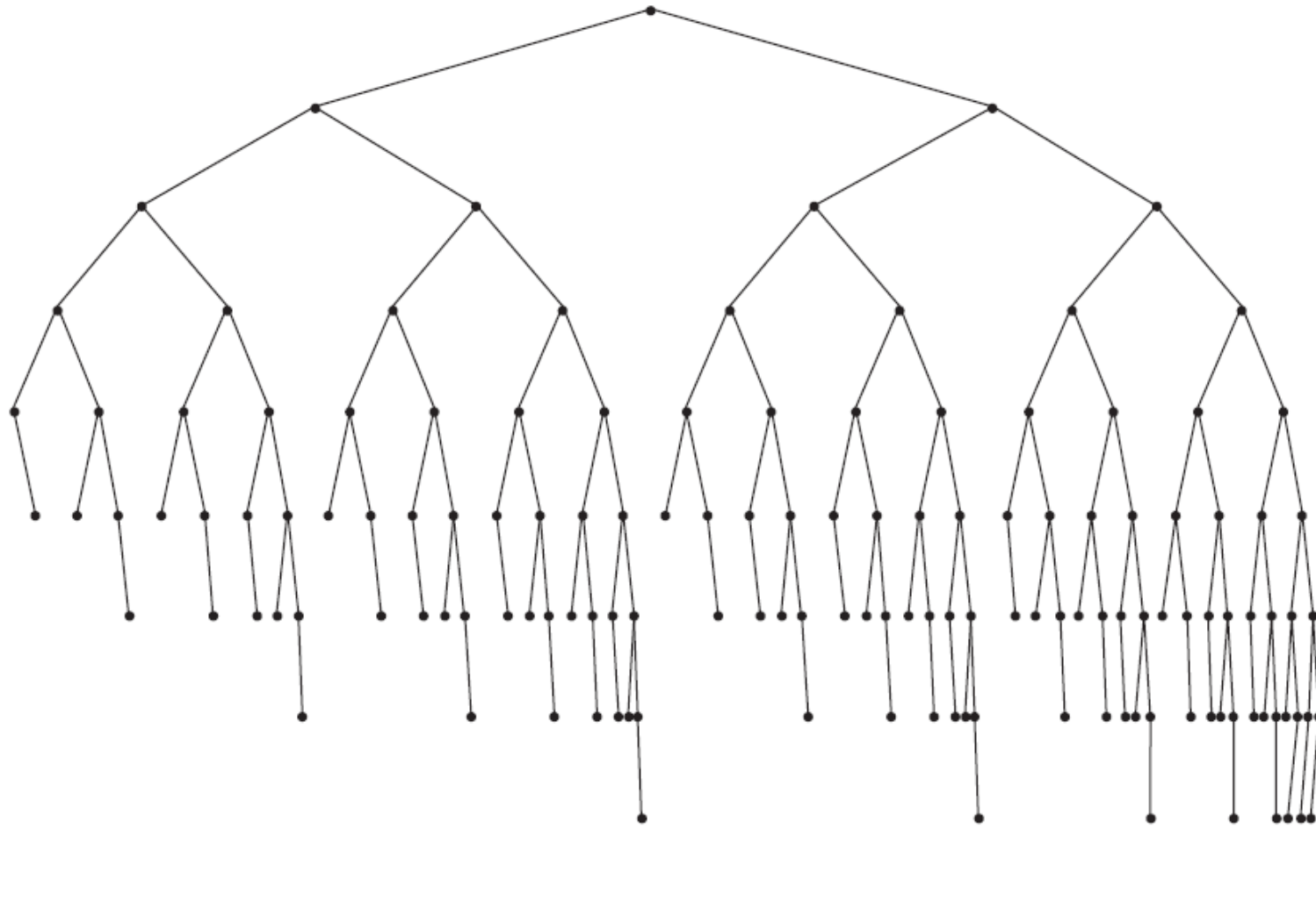
[Wikipedia]

ABPs equilibradas ?



[Weiss]

Árvore equilibrada – Qual é a sua altura ?



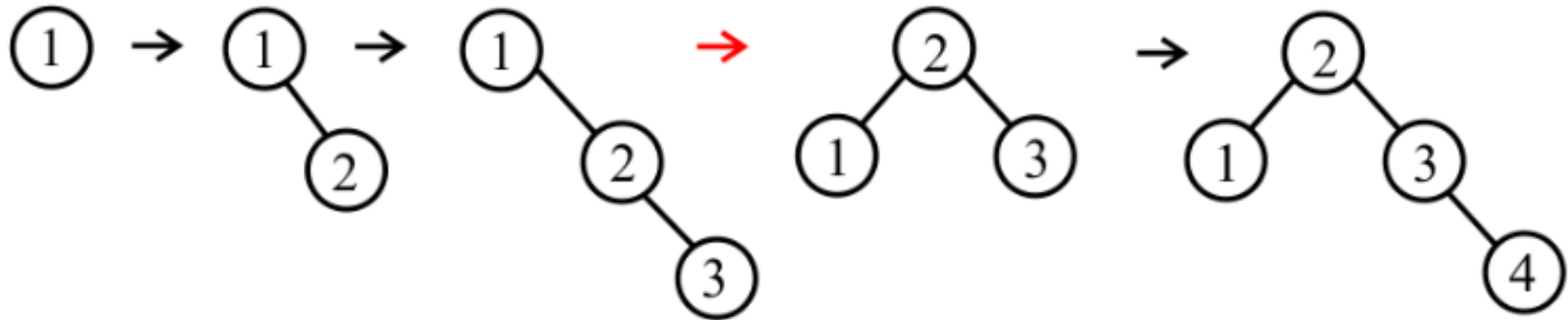
[Weiss]

Árvores de Adelson-Velskii e Landis

Estratégia

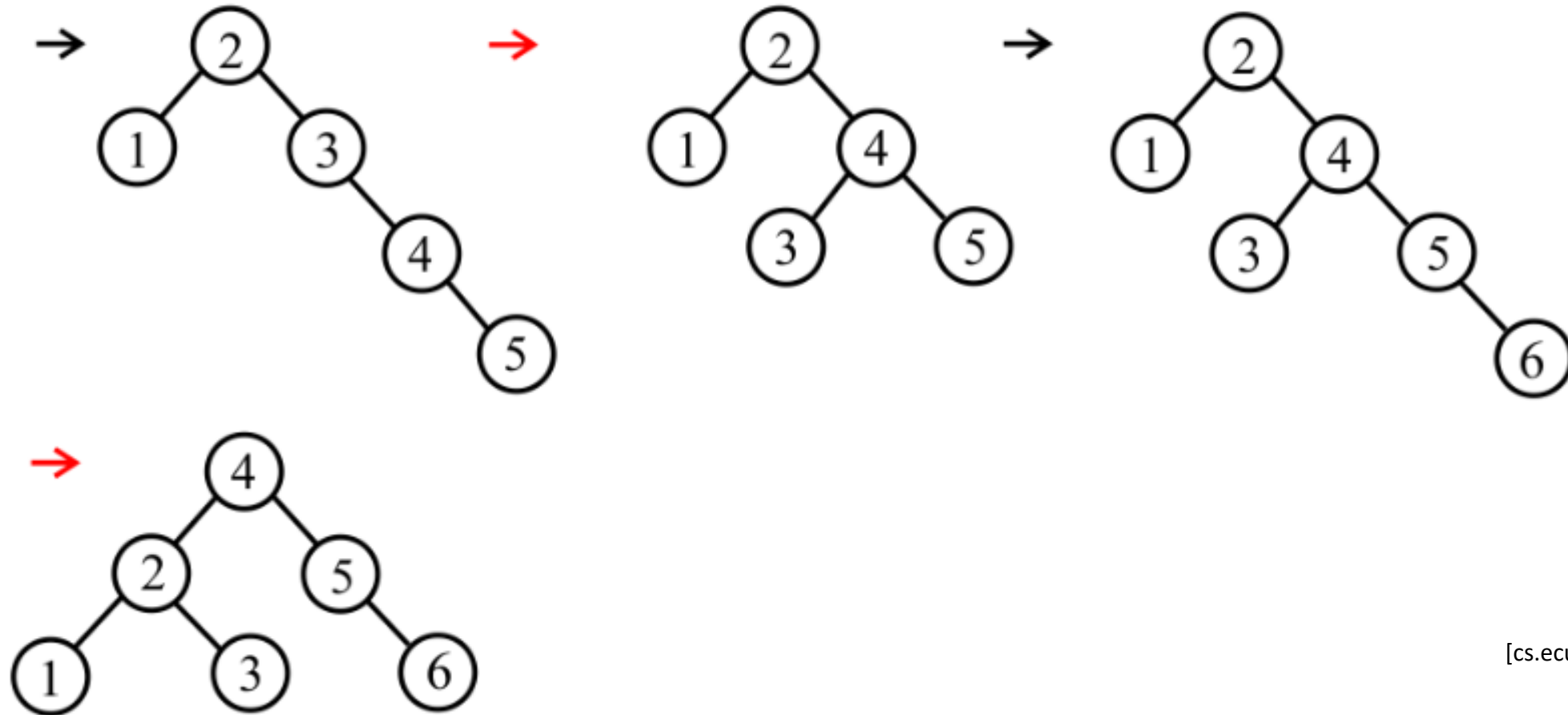
- **Assegurar** o **critério de equilíbrio** sempre que se adiciona ou remove um nó
- Tem de ser **fácil** de **verificar** e de **manter** !!
- **Reposicionar** nós / subárvores quando o critério de equilíbrio **falha** !!
- MAS, **manter** o **critério de ordem** da ABP !!
- Basta fazer a verificação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

Árvore AVL – Inserir + Equilibrar, se necessário



[cs.ecu.edu]

Árvore AVL – Inserir + Equilibrar, se necessário



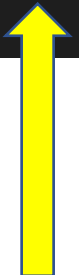
[cs.ecu.edu]


Fator de equilíbrio

- Para cada nó
- As duas sub-árvores 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**” !!

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;  
    }  
}
```

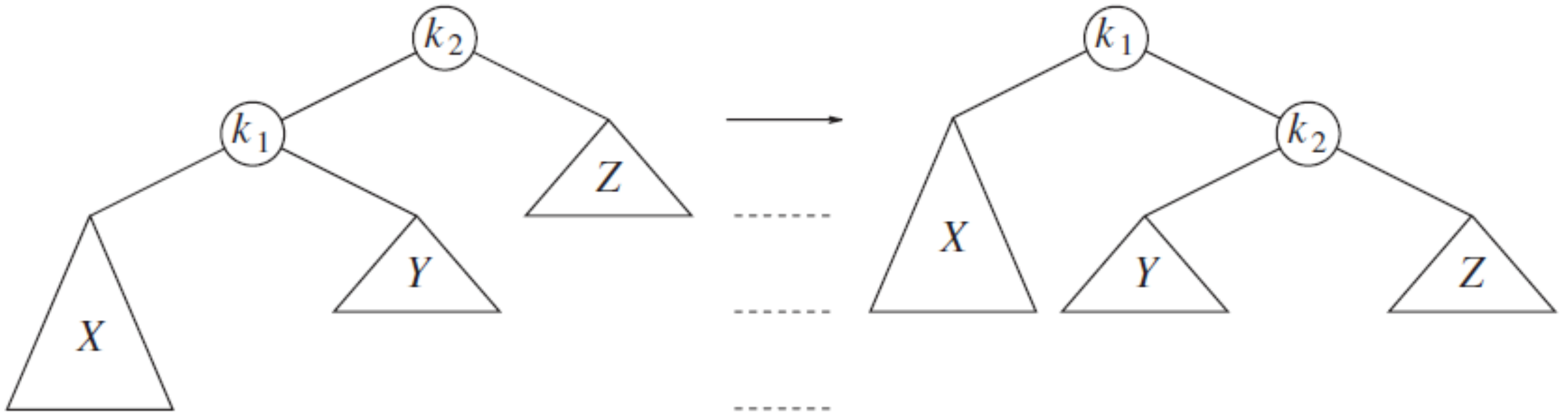


Como corrigir / equilibrar, se necessário ?

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

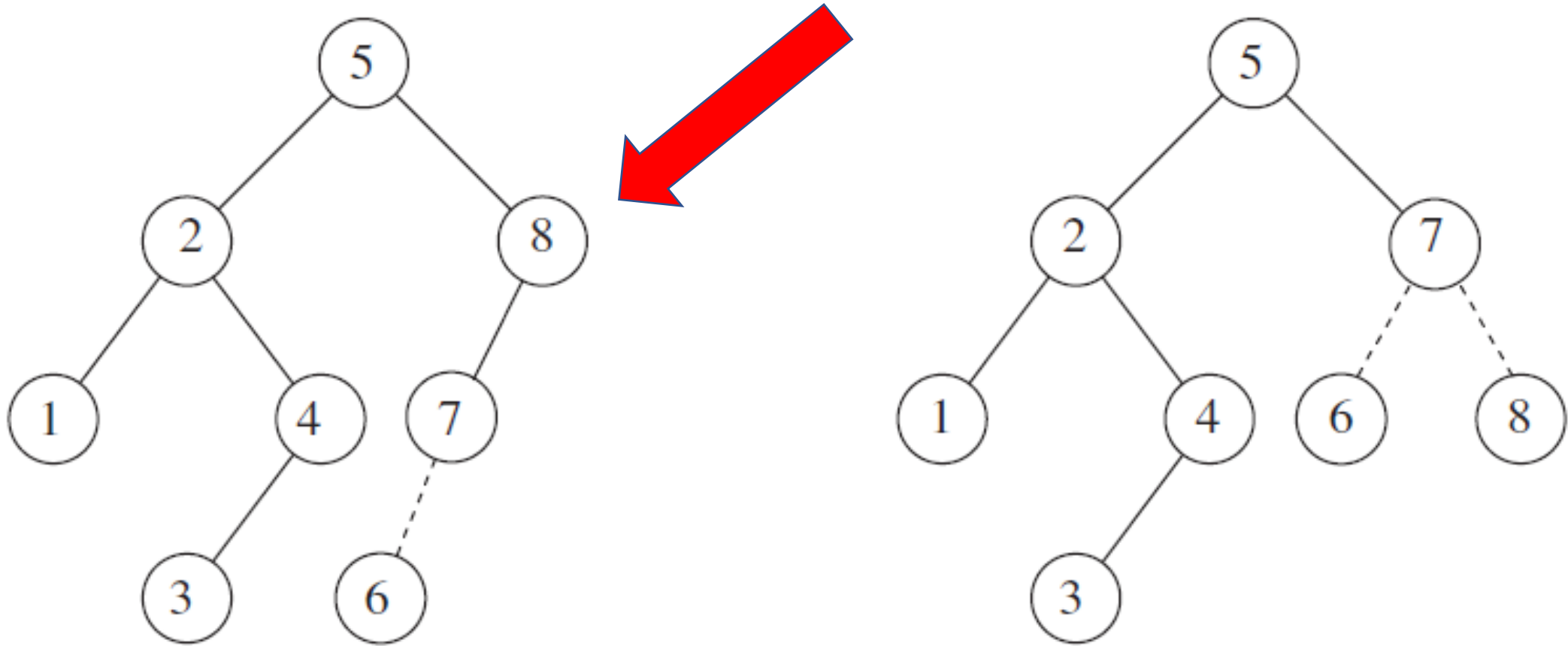
- **Rotações simples** à esquerda ou à direita
- **Rotações duplas** à esquerda ou à direita
 - Sequência de duas rotações simples

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



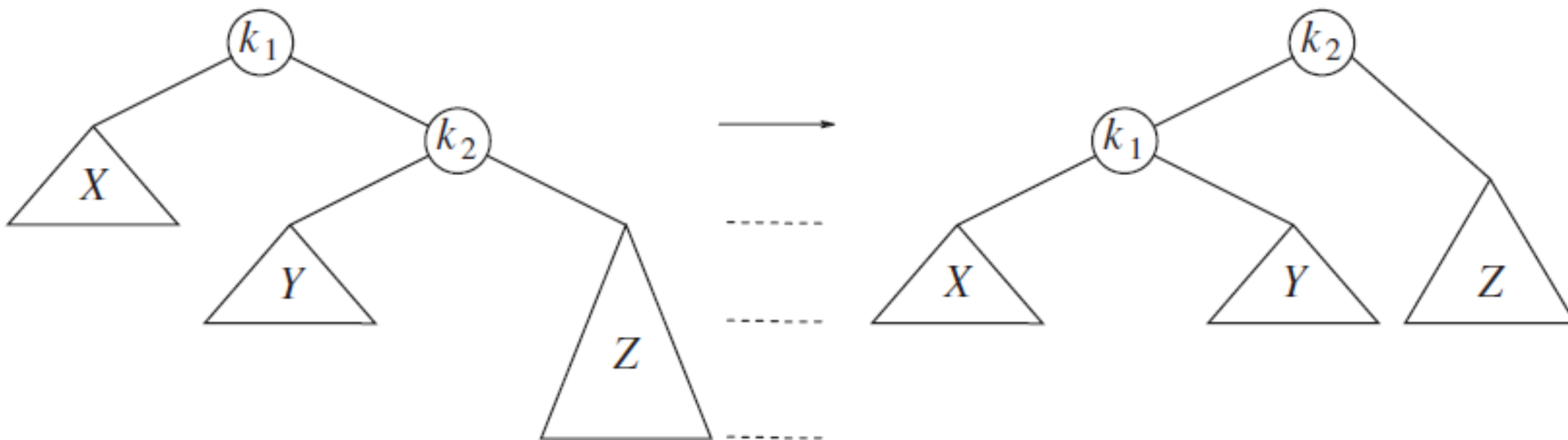
[Weiss]

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



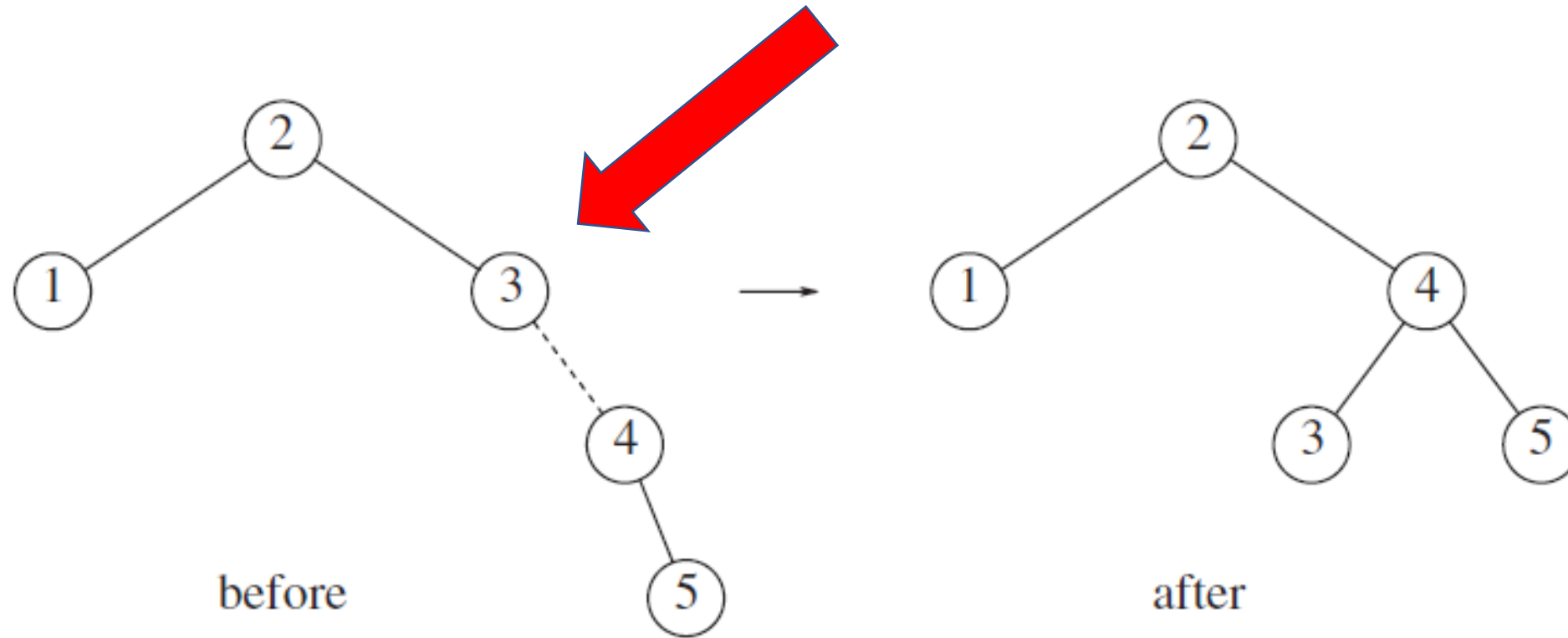
[Weiss]

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



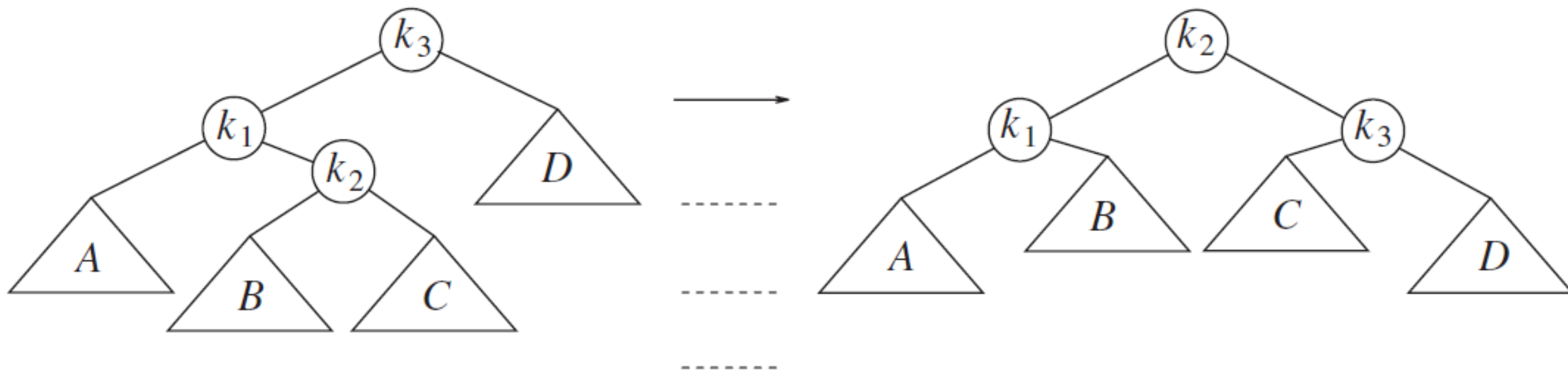
[Weiss]

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



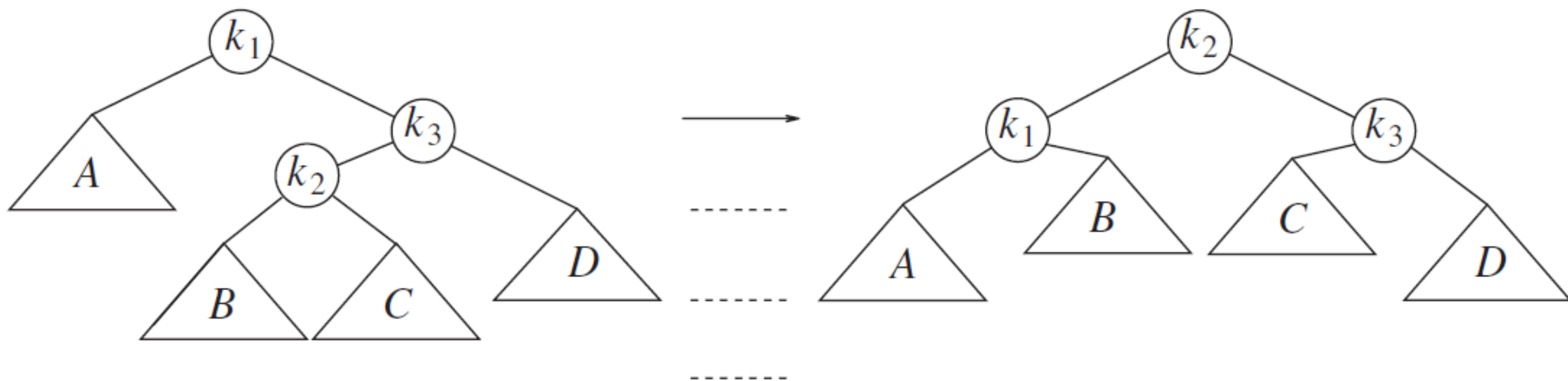
[Weiss]

Rotação dupla à esquerda – Como identificar?



[Weiss]

Rotação dupla à direita – Como identificar?



[Weiss]

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

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 ?

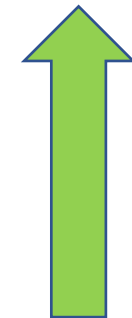
Árvores ABP vs Árvores AVL

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

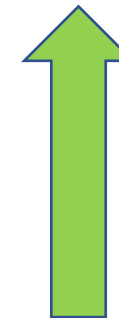
Procurar os sucessivos números pares

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

