

# Árvores Binárias III

Joaquim Madeira

05/05/2020

# Ficheiro ZIP

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

# Sumário

- Recap
- Árvores Binárias de Procura (ABP) – **Binary Search Trees** (BST)
- **Procura** de um elemento – Algs. recursivo e iterativo
- **Adição** de um elemento – Algs. recursivo e iterativo
- Procura e **remoção** de um elemento – Desenvolvimento “top-down”
- Análise do desempenho das operações habituais sobre ABPs

Let's  
RECAP

# Recapitulação

# TAD Árvore Binária – Funcionalidades

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

# Travessias recursivas

- Travessia em **pré-ordem** (**NLR**)

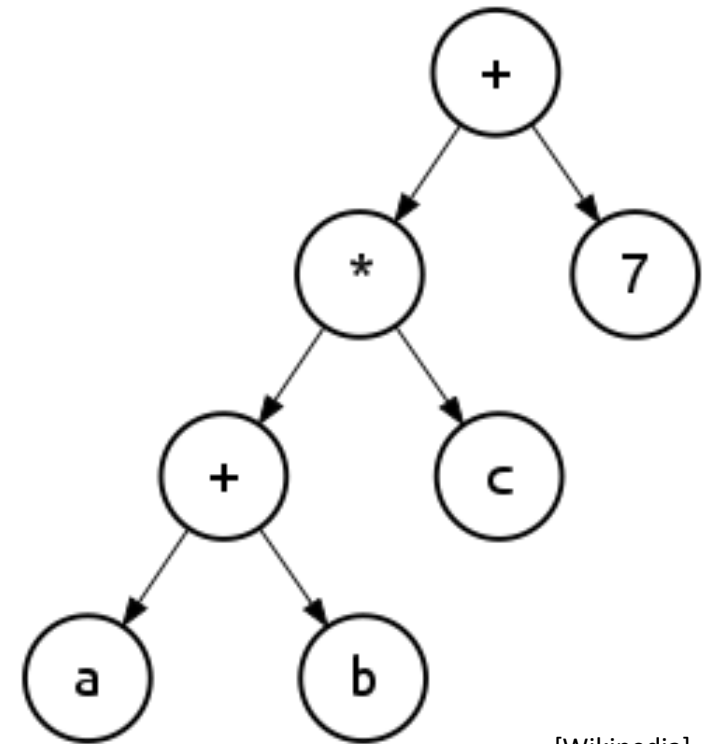
+ \* + a b c 7

- Travessia **em-ordem** (**LNR**)

a + b \* c + 7

- Travessia em **pós-ordem** (**LRN**)

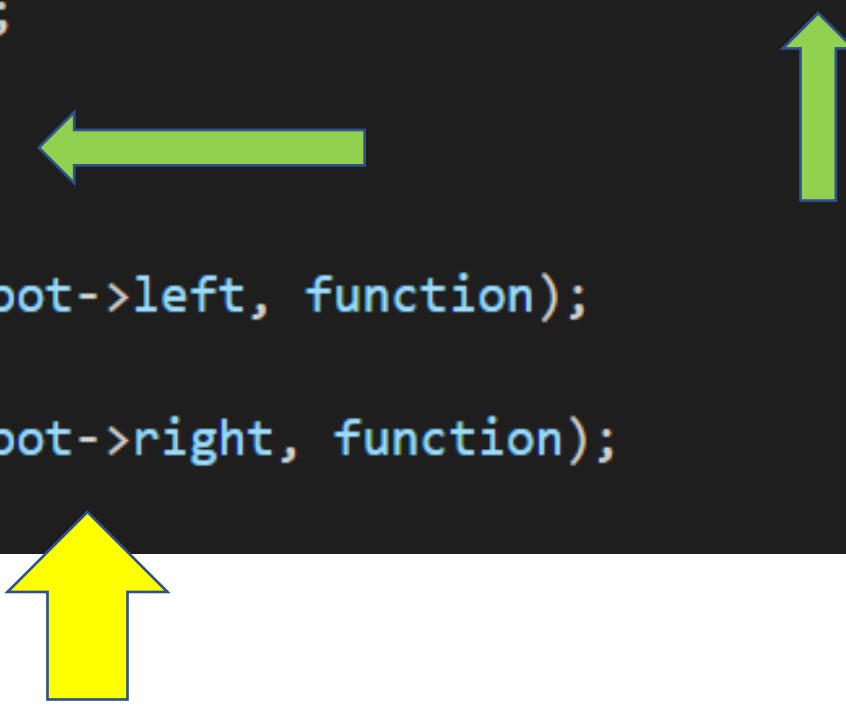
a b + c \* 7 +



[Wikipedia]

# Travessia em pré-ordem

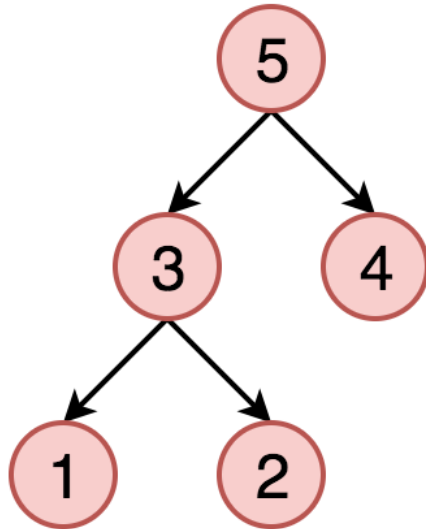
```
void TreeTraverseInPREOrder(Tree* root, void (*function)(ItemType* p)) {  
    if (root == NULL) return;  
  
    function(&(root->item));  
  
    TreeTraverseInPREOrder(root->left, function);  
  
    TreeTraverseInPREOrder(root->right, function);  
}
```



# Ordem / Travessias

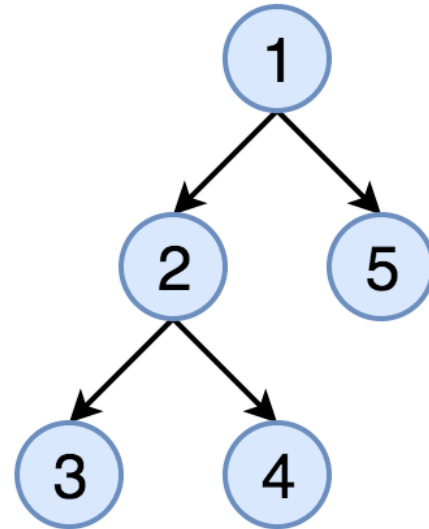
## DFS Postorder

Bottom -> Top  
Left -> Right



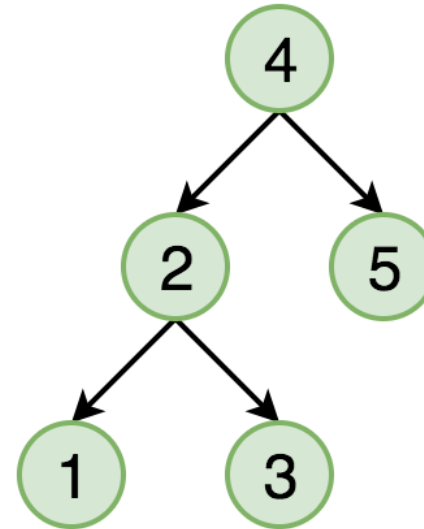
## DFS Preorder

Top -> Bottom  
Left -> Right



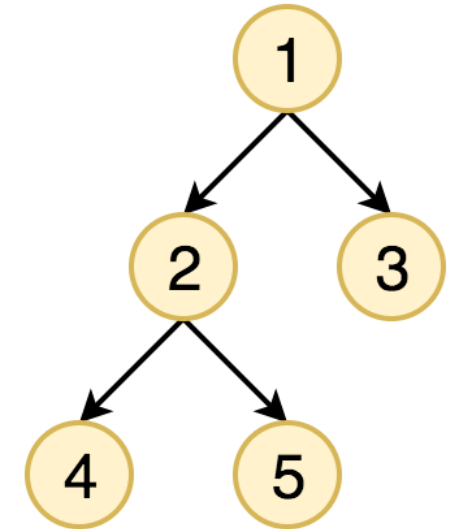
## DFS Inorder

Left -> Node -> Right



## BFS

Left -> Right  
Top -> Bottom



[zhang-xiao-mu.blog]



# Travessias iterativas

- Usar uma estrutura de dados auxiliar : **QUEUE** ou **STACK**
- Armazenar **ponteiros** para os próximos nós a processar
- **QUEUE** : **Breadth-First** – por níveis
- **STACK** : **Depth-First** – em profundidade
  - Pré-Ordem / Em-Ordem / Pós-Ordem

# Travessias por níveis – QUEUE

```
void TreeTraverseLevelByLevelWithQUEUE(Tree* root,  
                                         void (*function)(ItemType* p)) {  
    if (root == NULL) {  
        return;  
    }  
  
    // Not checking for queue errors !!  
    // Create the QUEUE for storing POINTERS  
  
    Queue* queue = QueueCreate();  
  
    QueueEnqueue(queue, root);
```

# Travessias por níveis – QUEUE

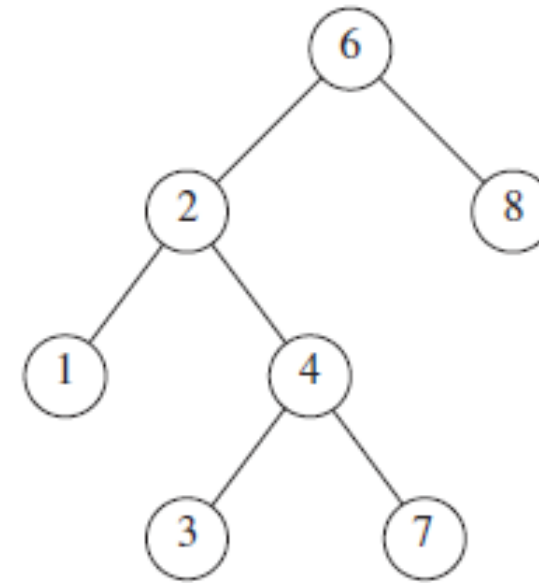
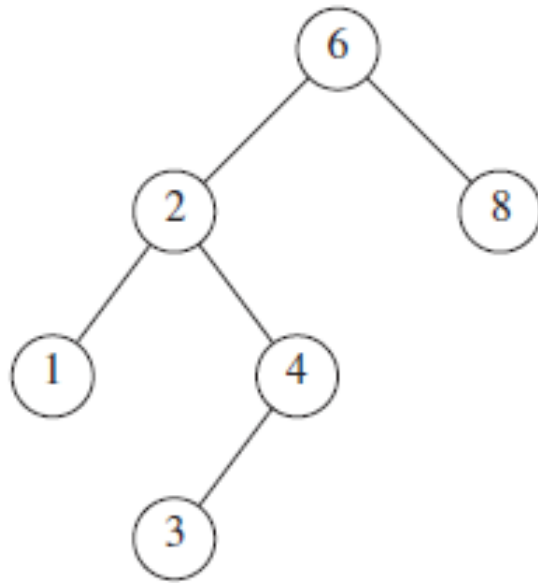
```
while (QueueIsEmpty(queue) == 0) {  
    Tree* p = QueueDequeue(queue);  
  
    function(&(p->item));  
  
    if (p->left != NULL) {  
        QueueEnqueue(queue, p->left);  
    }  
    if (p->right != NULL) {  
        QueueEnqueue(queue, p->right);  
    }  
}  
  
QueueDestroy(&queue);  
}
```

# Árvores Binárias de Procura – Binary Search Trees

# 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()**

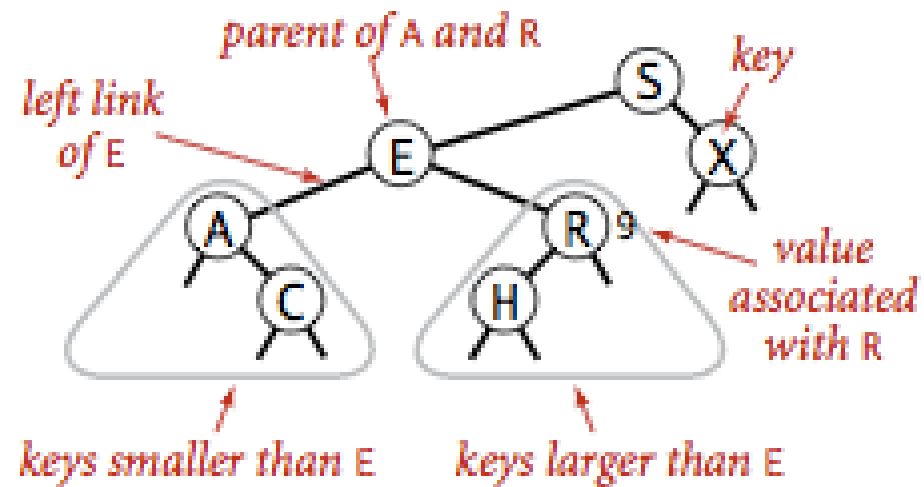
# Critério de **ordem**



[Weiss]

- Qual das árvores está **ordenada** ?
- O que se modifica / simplifica por existir uma ordem ?

# Organização de uma ABP

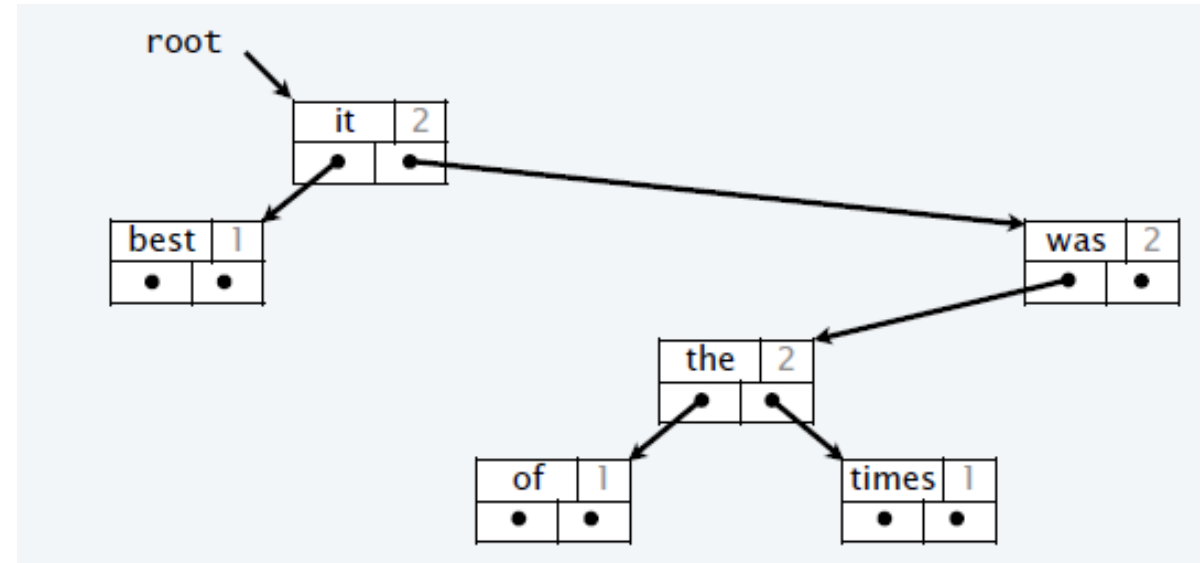


**Anatomy of a binary search tree**

[Sedgewick & Wayne]

# Critério de ordem – Definição recursiva

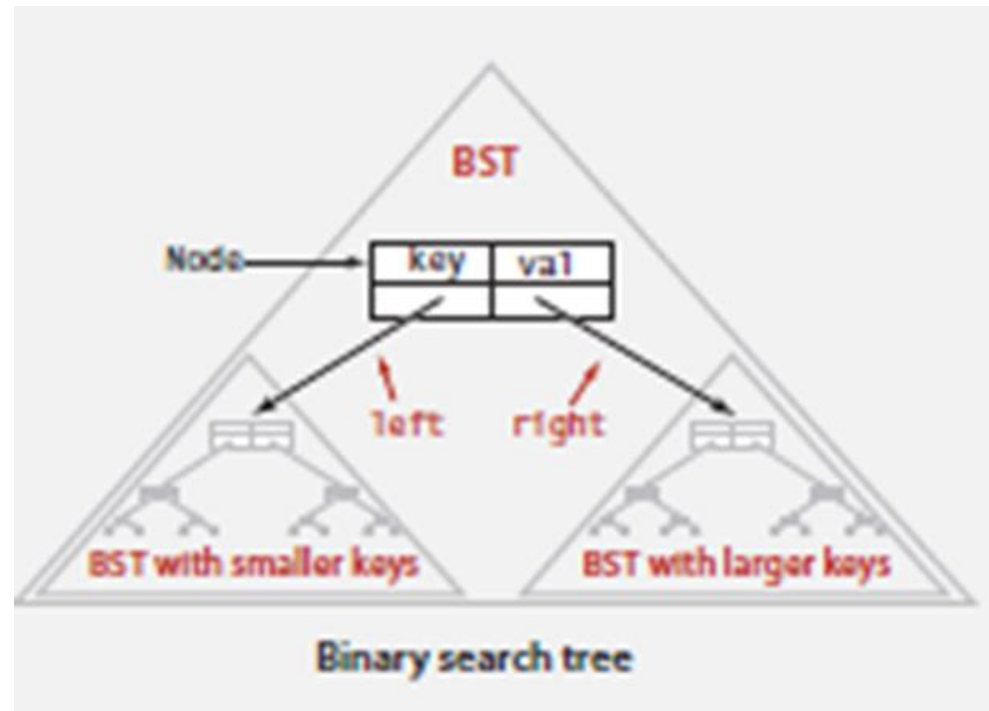
- Para cada nó, os elementos da sua **subárvore esquerda** são **inferiores** ao nó
- E os elementos da sua **subárvore direita** são **superiores** ao nó
- **Não** há elementos **repetidos** !!
- A organização da árvore depende da **sequência de inserção** dos elementos



[Sedgewick & Wayne]



# Definição Recursiva

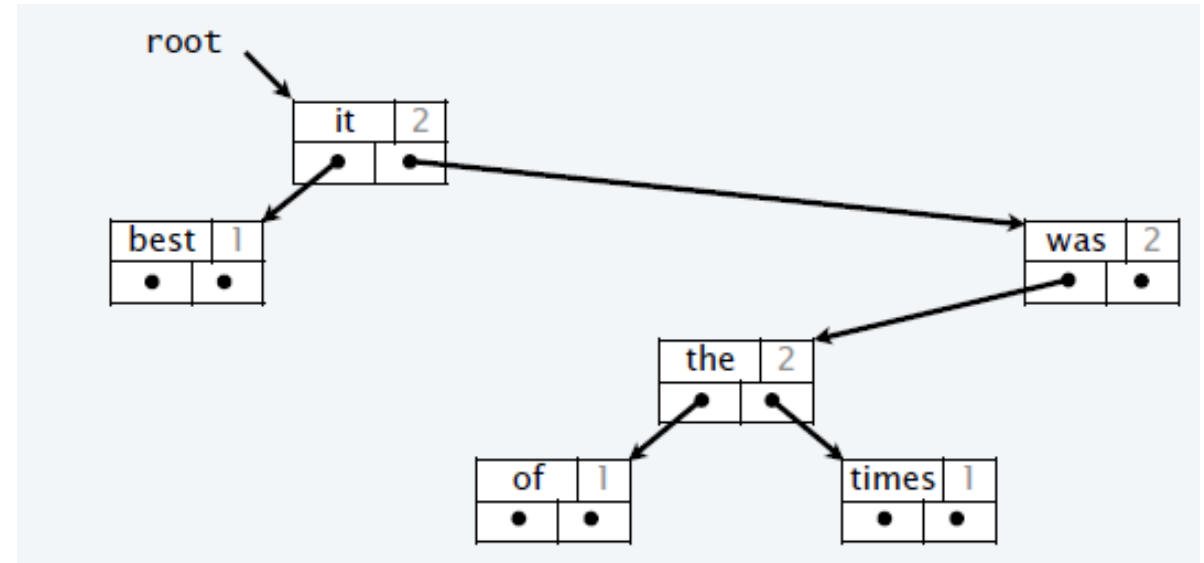


[Sedgewick & Wayne]

# Operações habituais

- O **item** armazenado em cada nó é, em geral, um par (**chave, valor**)
- Procurar
- Adicionar
- Alterar
- Remover
- Visitar em-ordem

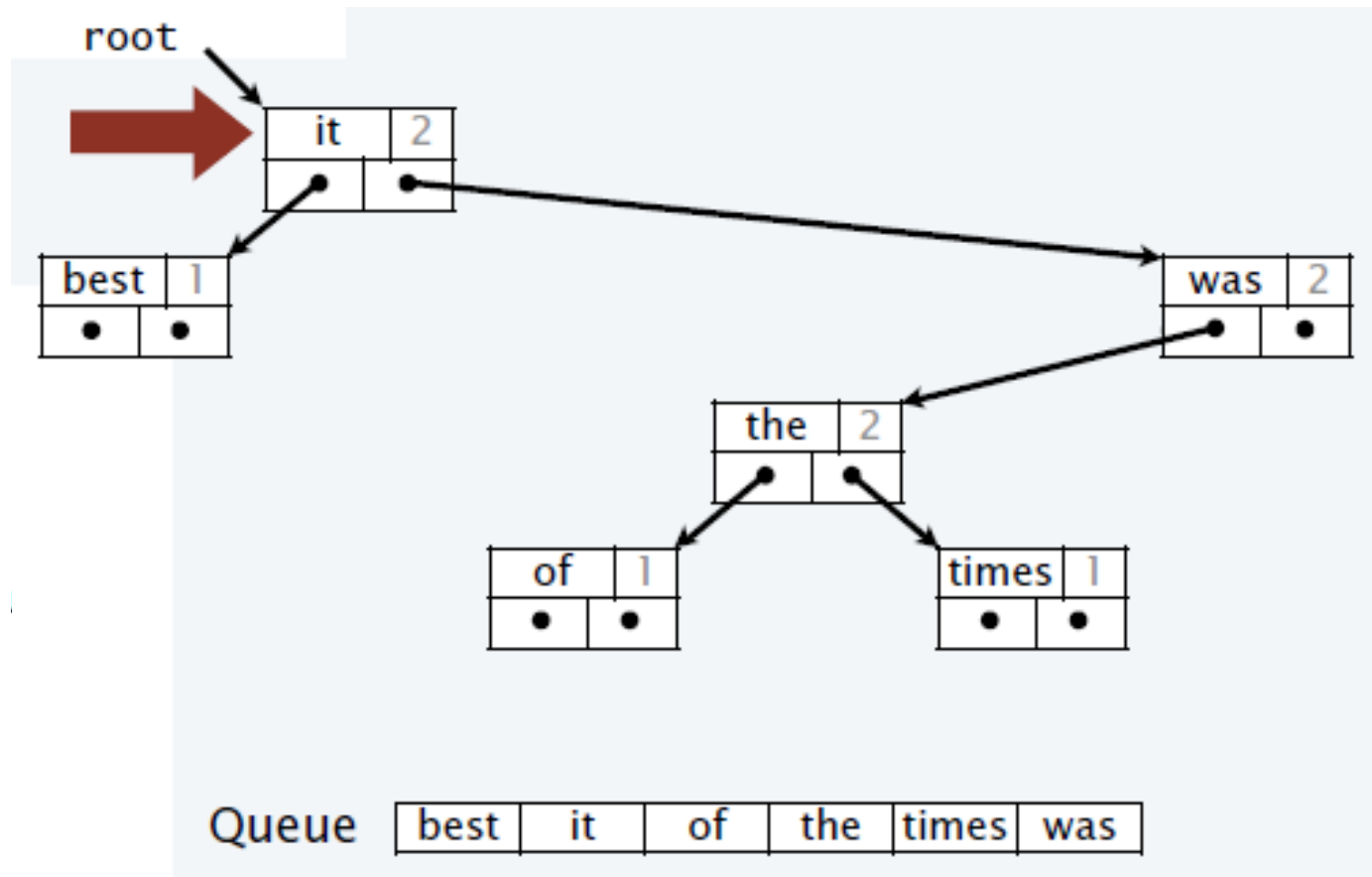
✓



[Sedgewick & Wayne]

# Travessia em-ordem

- Exemplo: preencher uma fila com os itens ordenados

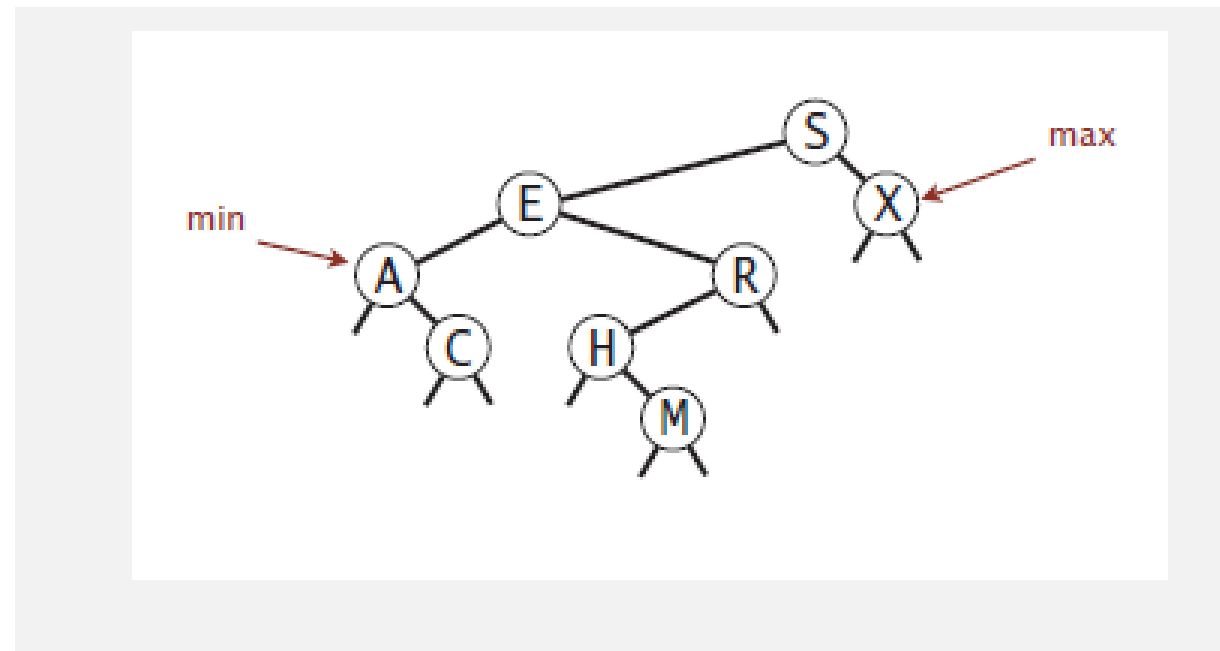


[Sedgewick & Wayne]

# Algumas funções simples

# Menor item ? / Maior item ?

- Como fazer ?



[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);  
}
```


# getMin()

- Tarefa
- Versão iterativa

# getMax()



```
ItemType BSTreeGetMax(const BSTree* root) {  
    if (root == NULL) {  
        return NO_ITEM;  
    }  
  
    while (root->right != NULL) {  
        root = root->right;  
    }  
    return root->item;  
}
```



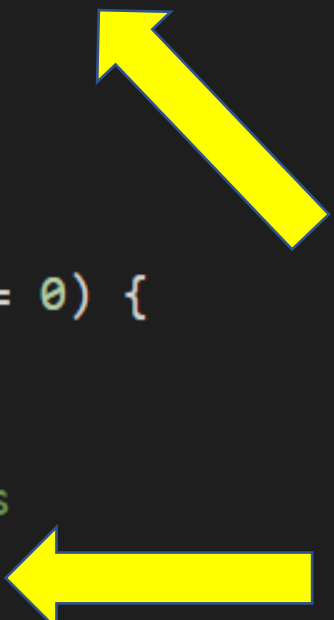


# getMax()

- Tarefa
- Versão recursiva

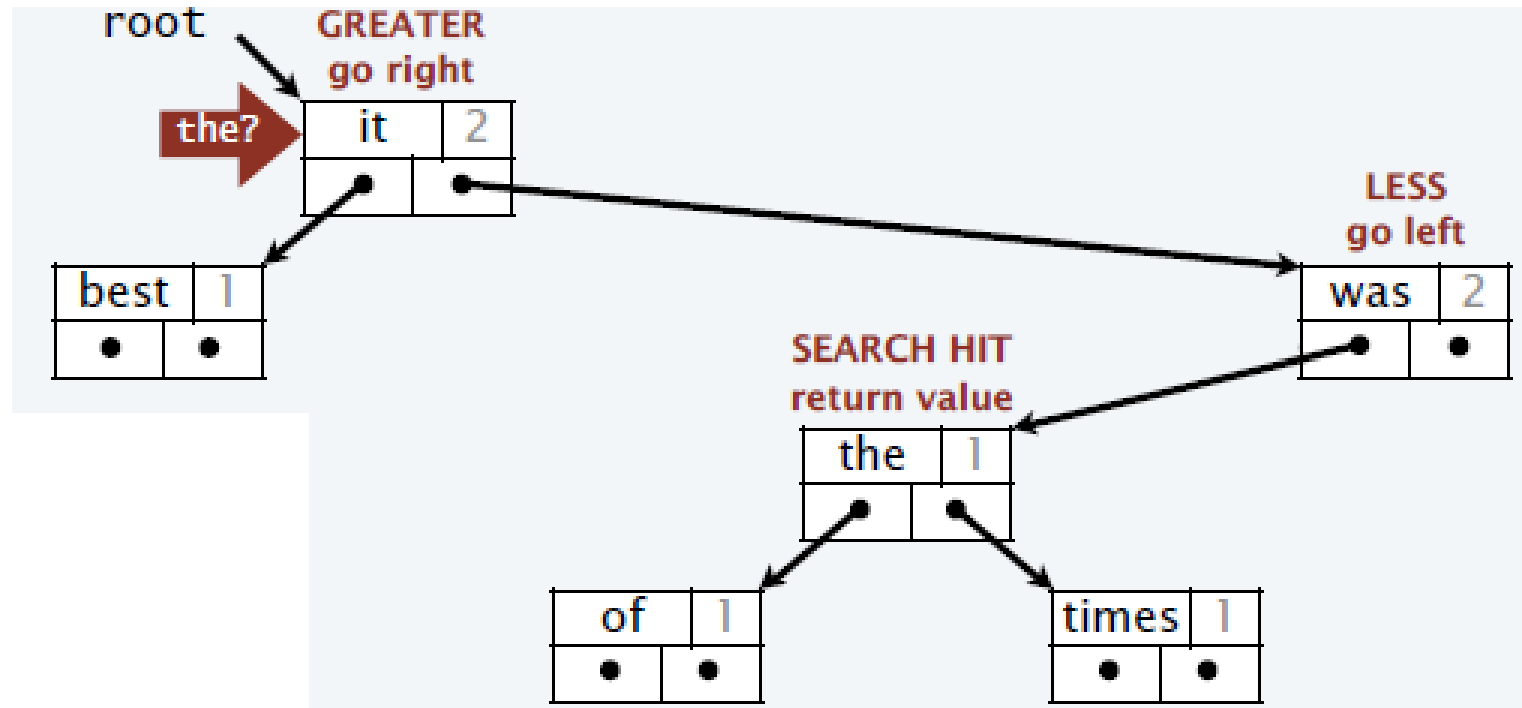
# isBST()

```
int _isBST(const BSTree* root, ItemType* prevItem) {  
    if (root == NULL) {  
        return 1;  
    }  
    // IN_ORDER TRAVERSAL  
    if (_isBST(root->left, prevItem) == 0) {  
        return 0;  
    }  
    // Allow only distinct valued nodes  
    if (root->item <= *prevItem) {  
        return 0;  
    }  
    // Update prevValue to current  
    *prevItem = root->item;  
    return _isBST(root->right, prevItem);  
}
```

Two yellow arrows are present. One arrow points from the right towards the 'prevItem' parameter in the function signature 'int \_isBST(const BSTree\* root, ItemType\* prevItem)'. The second arrow points from the right towards the comparison 'root->item <= \*prevItem' in the code.

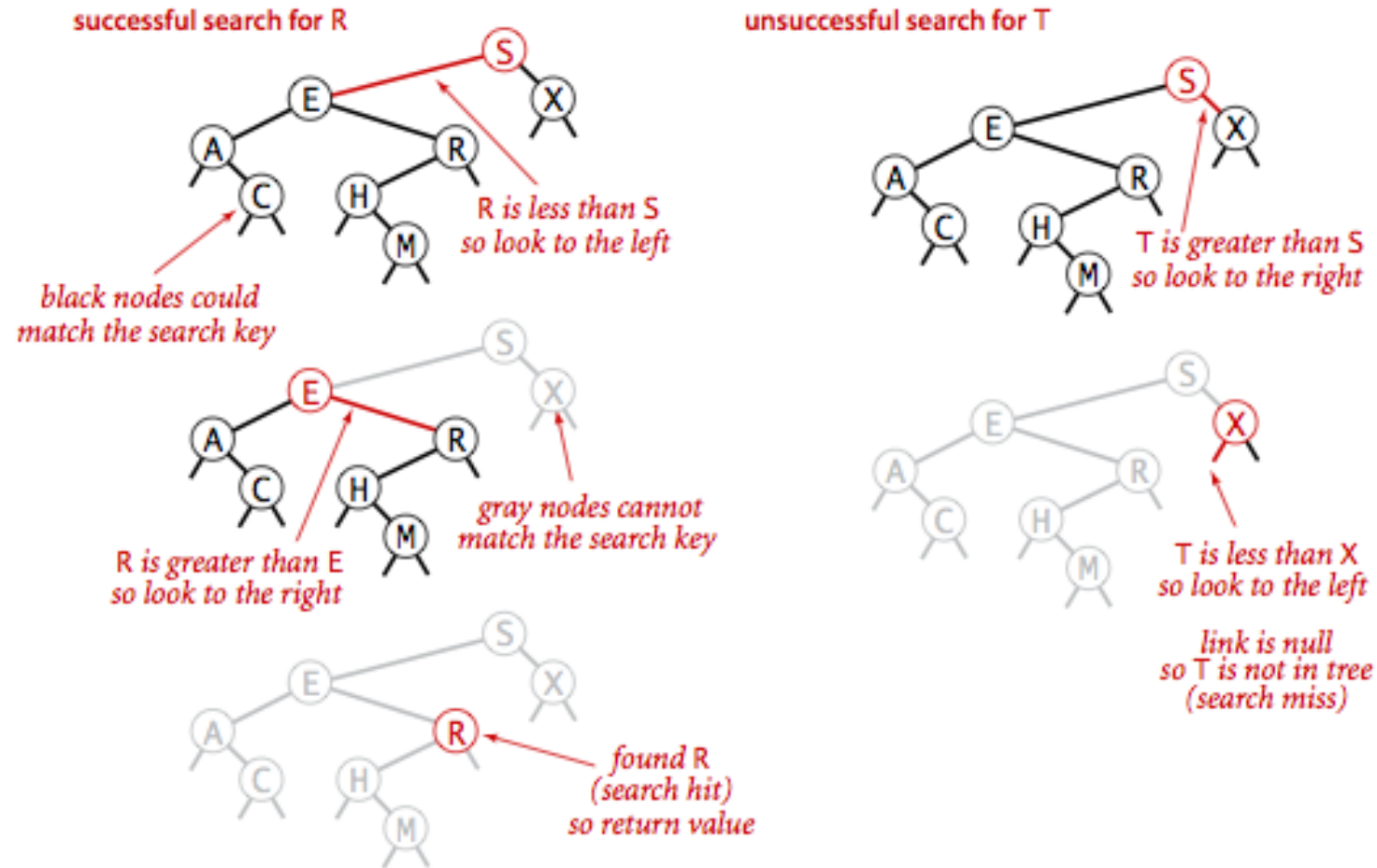
# Procurar um item

# Procurar



[Sedgewick & Wayne]

# Procurar



[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;  
}
```

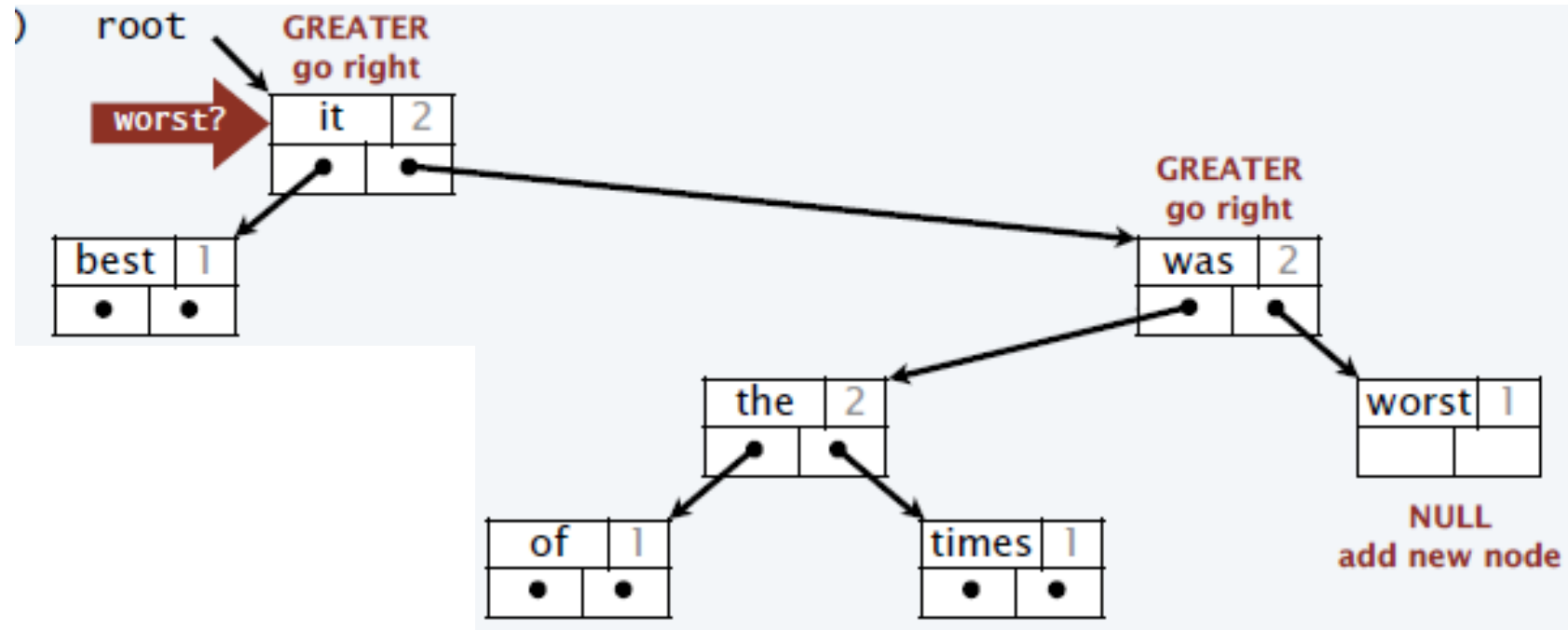
# Procurar – Versão recursiva

- Tarefa
- Versão recursiva

# Adicionar um item



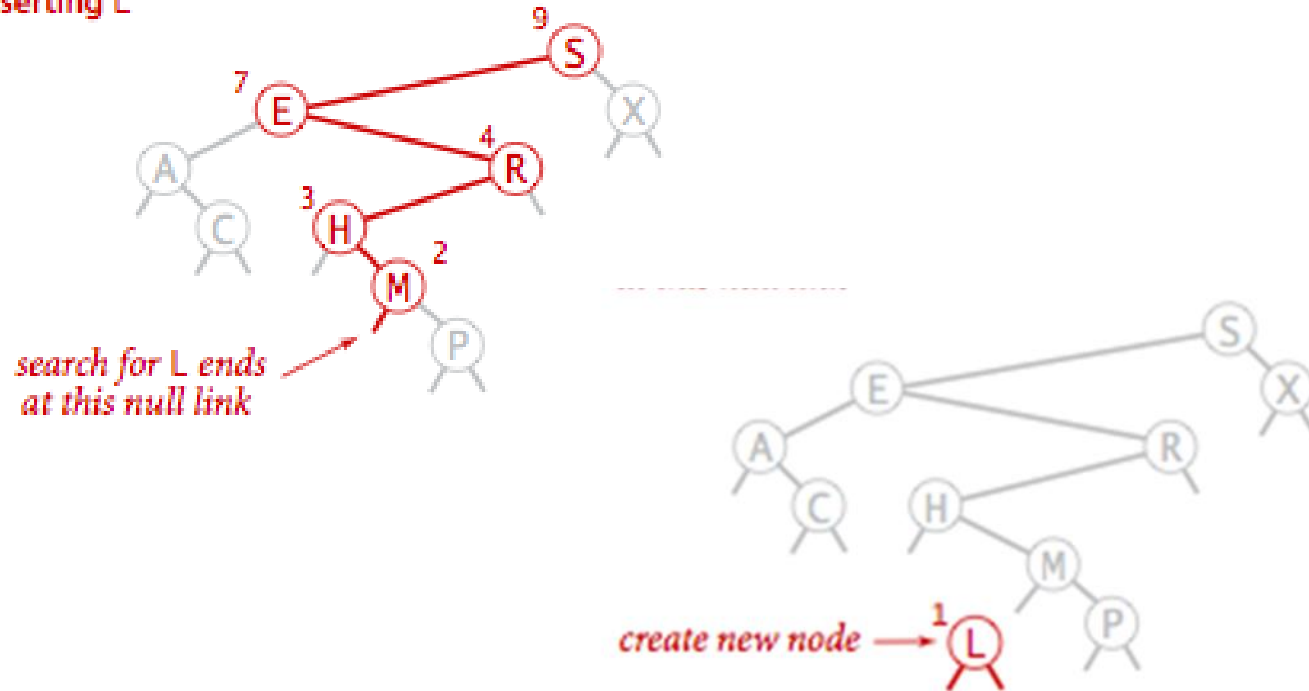
# Adicionar



[Sedgewick & Wayne]

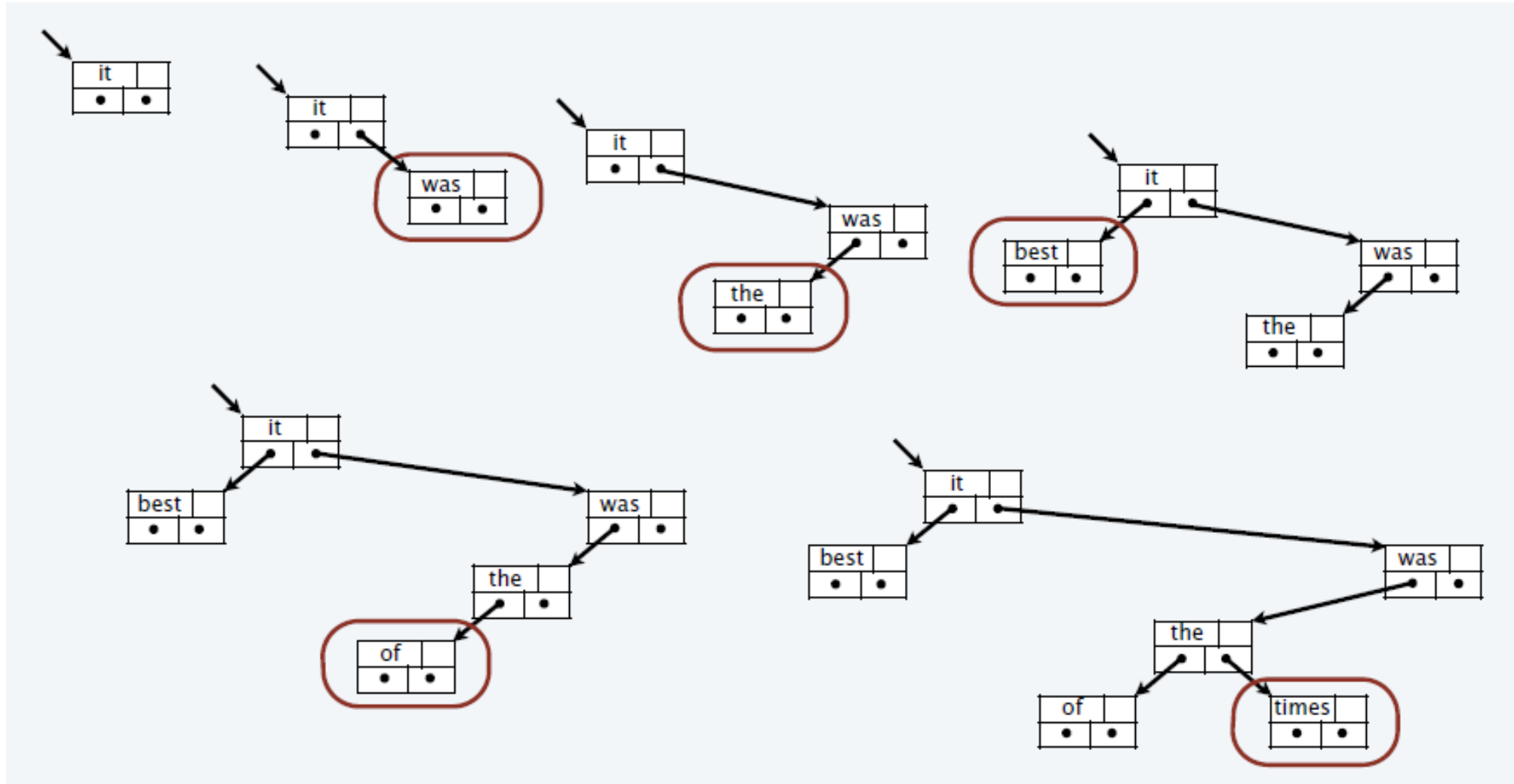
# Adicionar – Outro exemplo

inserting L



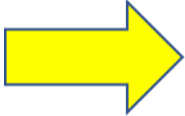
[Sedgewick & Wayne]

# Adicionar – Sequência de operações


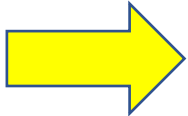


[Sedgewick & Wayne]

## 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;  
    }  
}
```

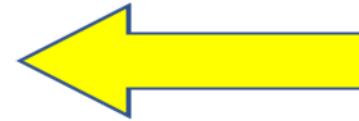


# Adicionar – Procurar a posição

```
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

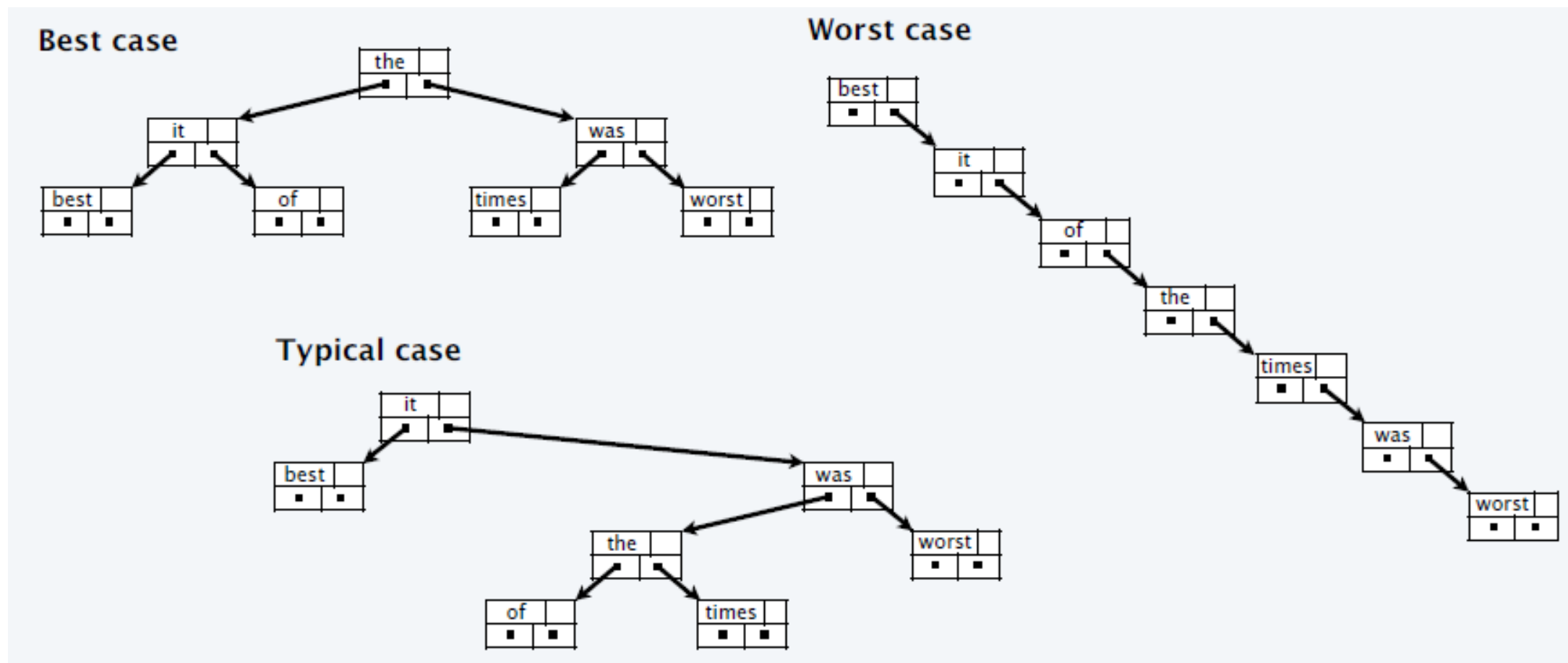


```
if (prev->item > item) {  
    prev->left = new;  
} else {  
    prev->right = new;  
}  
return 0;  
}
```

# Adicionar – Versão recursiva

- Tarefa
- Versão recursiva

# Possíveis árvores

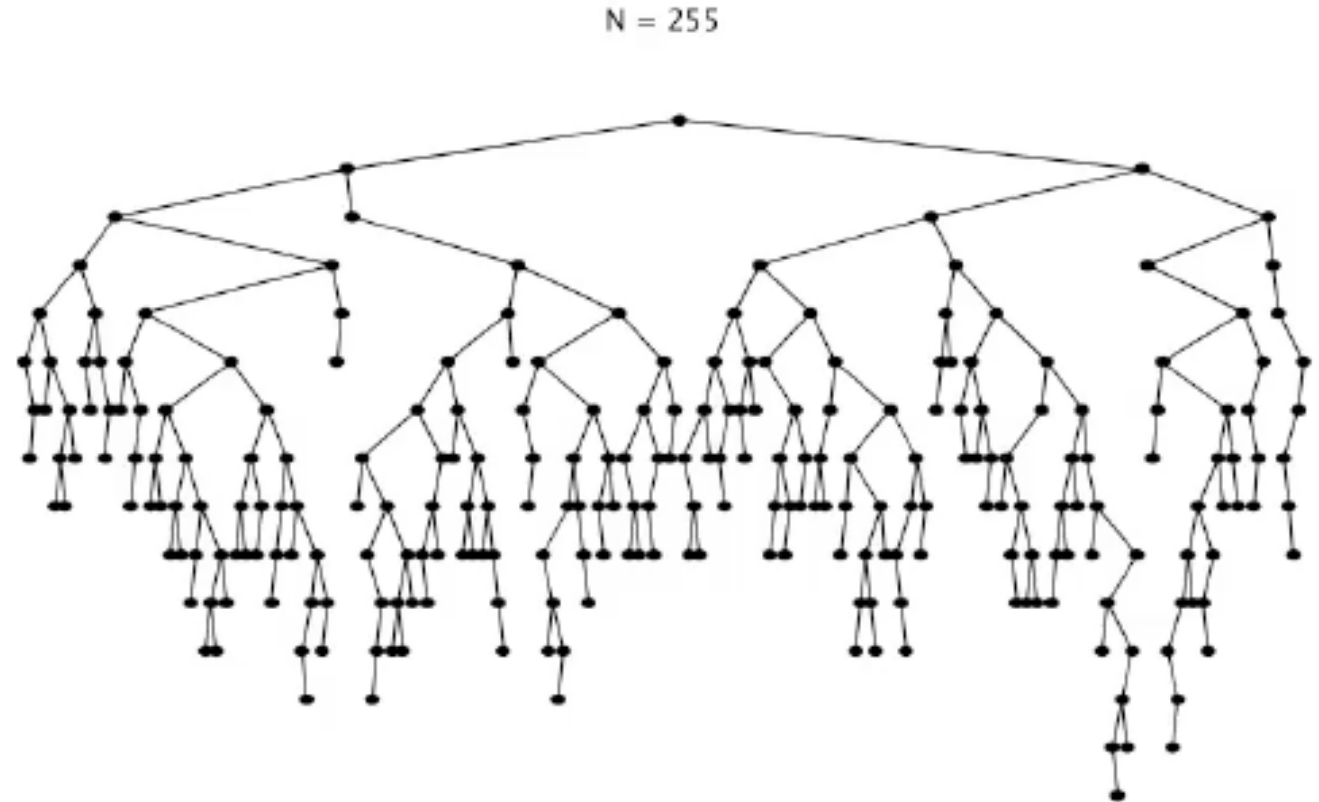


[Sedgewick & Wayne]



# Adição numa ordem aleatória

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



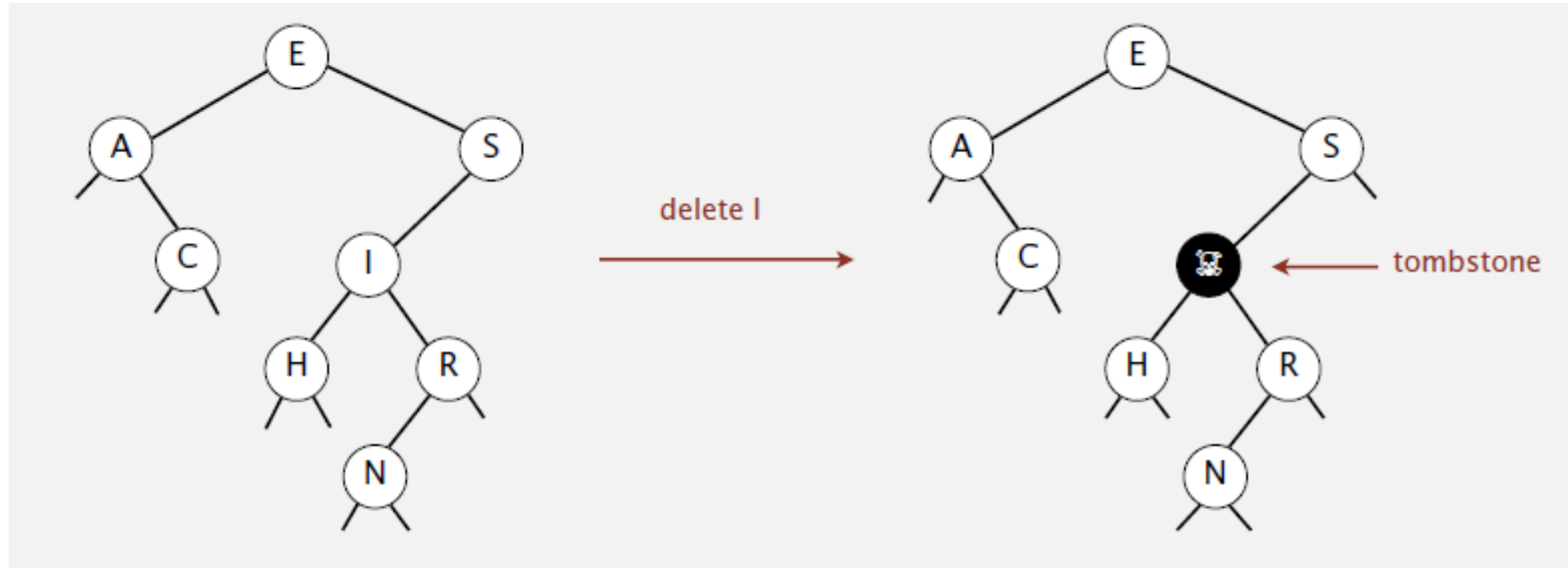
[Sedgewick & Wayne]

# Remover um item

# Remove

- **Restrição** : manter a ordem dos itens após a remoção !!
- Como fazer ?
- “**Lazy** deletion”
- Remove o **menor** item
- O método de **Hibbard**

# “Lazy deletion”



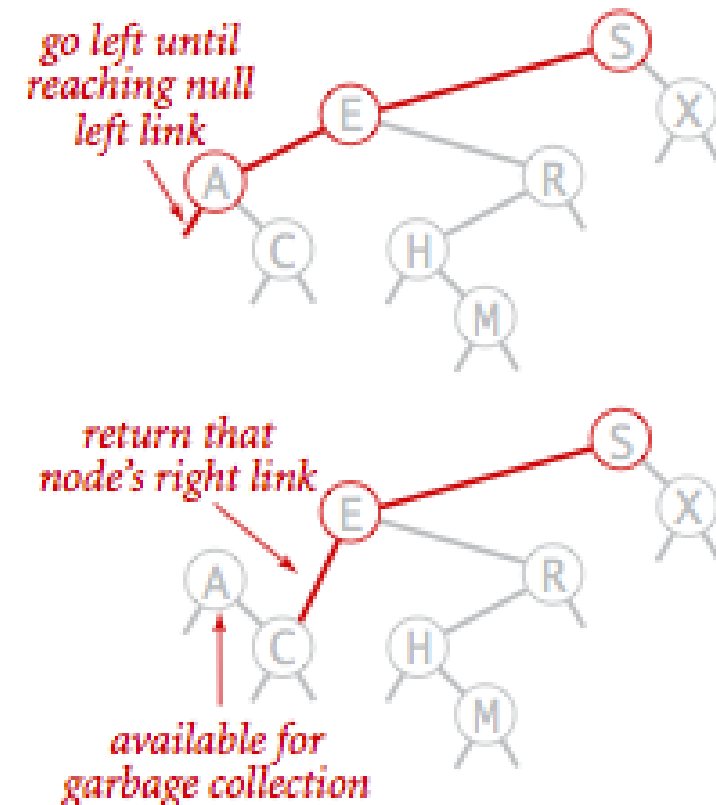
[Sedgewick & Wayne]

# “Lazy deletion”

- Procurar o item
- Marcar o seu nó como “apagado”
- MAS, mantê-lo na árvore
- Alterar o modo de comparação em todas as operações !!
- VANTAGEM : rapidez
- DESVANTAGEM : gasto desnecessário de memória !!

# Remover o **menor** item

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



[Sedgewick & Wayne]

# Remover o menor item

- TAREFA : fazer uma função recursiva

# Remover o maior item

- TAREFA : fazer uma função recursiva



# Remover um nó – Método de Hibbard

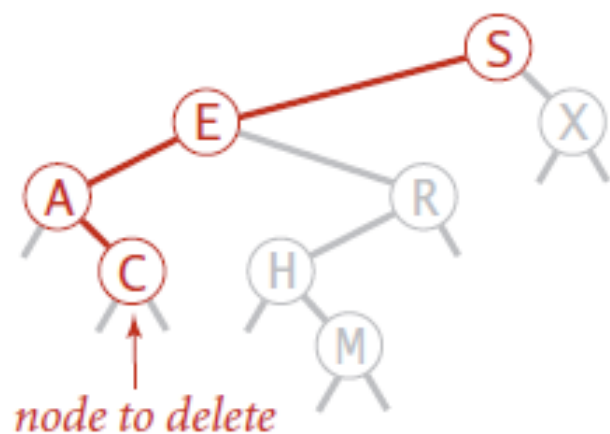
- Dada a chave / o item **k**, **procurar** o respetivo **nó**
- **Caso 1** : é uma **folha** – FÁCIL !!
- **Caso 2** : só tem **subárvore esquerda**
- **Caso 3** : só tem **subárvore direita**
- **Caso 4** : tem **2 subárvores**

# Procurar 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



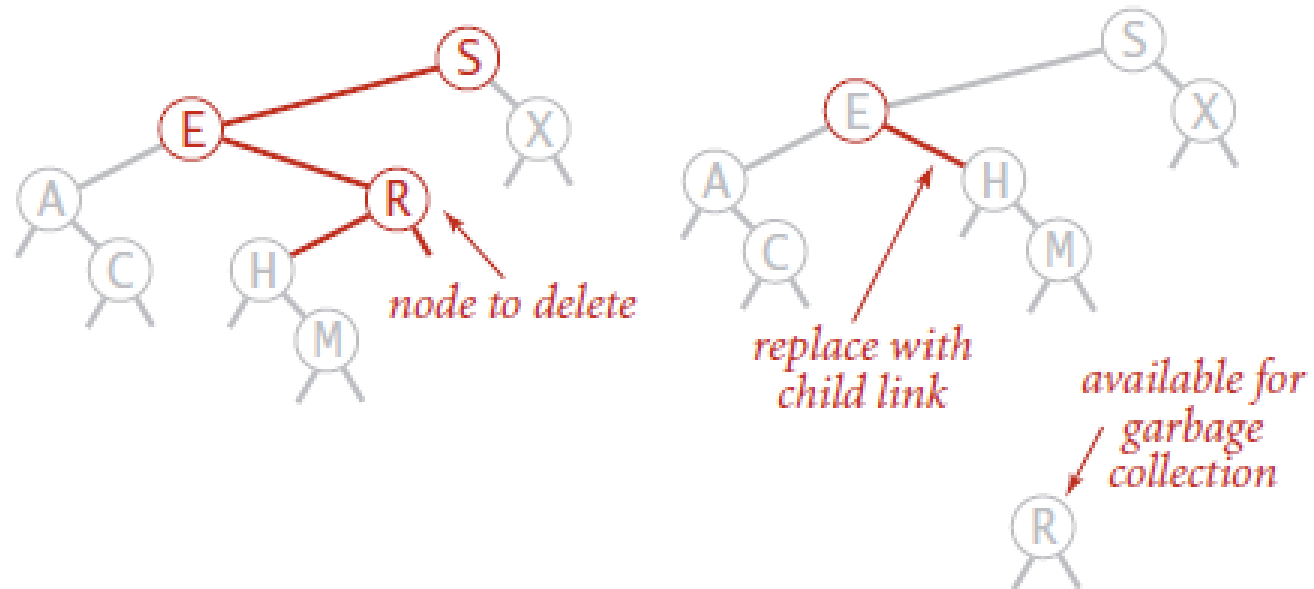
[Sedgewick & Wayne]

# Remover um nó que é uma folha

```
void _removeNode(BSTree** pPointer) {  
    BSTree* nodePointer = *pPointer;  
  
    if ((nodePointer->left == NULL) && (nodePointer->right == NULL)) {  
        /* A LEAF node */  
  
        free(nodePointer);  
  
        *pPointer = NULL;  
  
        return;  
    }  
}
```

# Remover um nó que só tem um filho

deleting R



[Sedgewick & Wayne]

# Remover um nó que só tem um filho

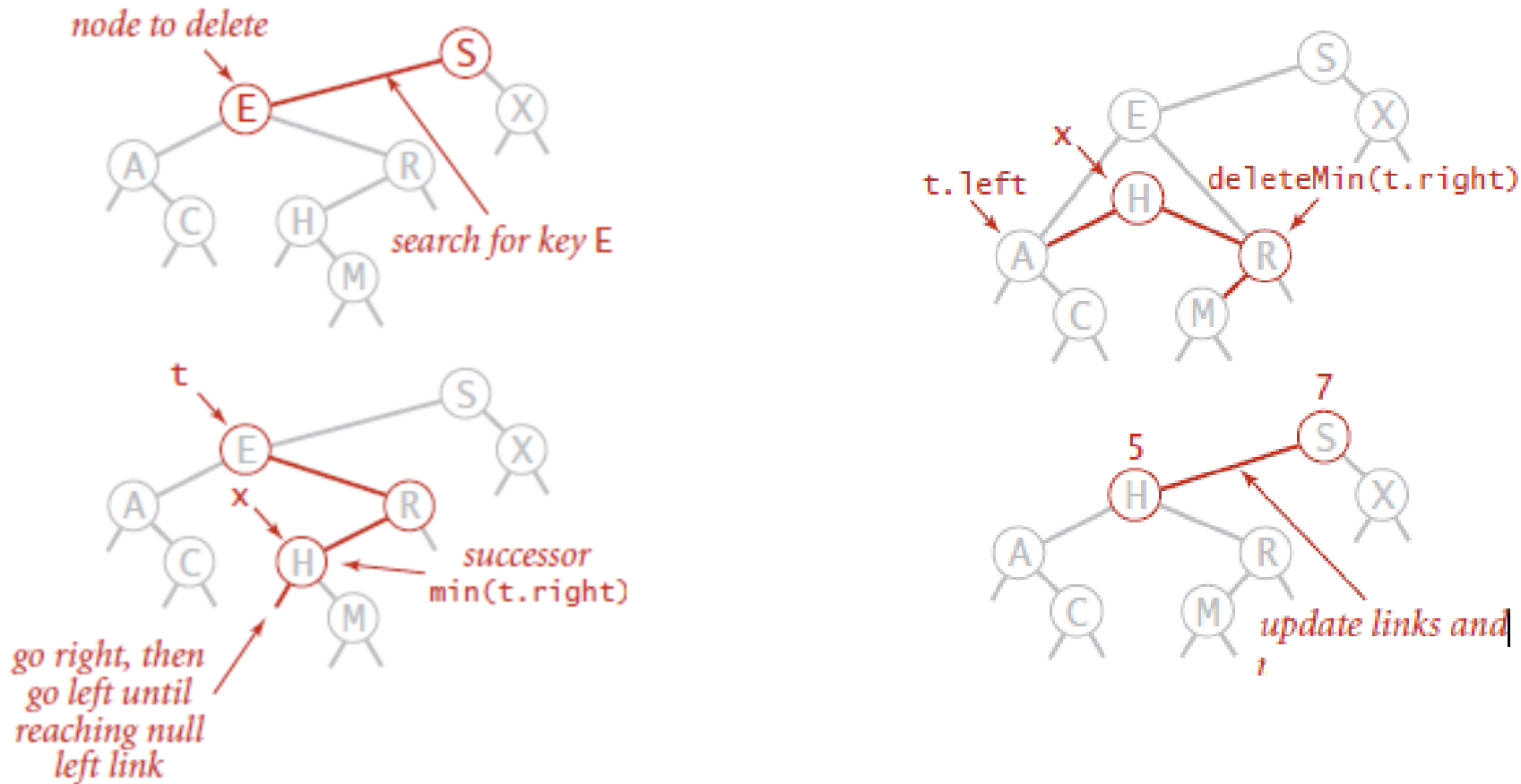
```
if (nodePointer->left == NULL) {  
    /* It has only a RIGHT sub-tree */  
  
    *pPointer = nodePointer->right;  
  
    free(nodePointer);  
  
    return;  
}
```

```
if (nodePointer->right == NULL) {  
    /* It has only a LEFT sub-tree */  
  
    *pPointer = nodePointer->left;  
  
    free(nodePointer);  
  
    return;  
}
```

# 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**

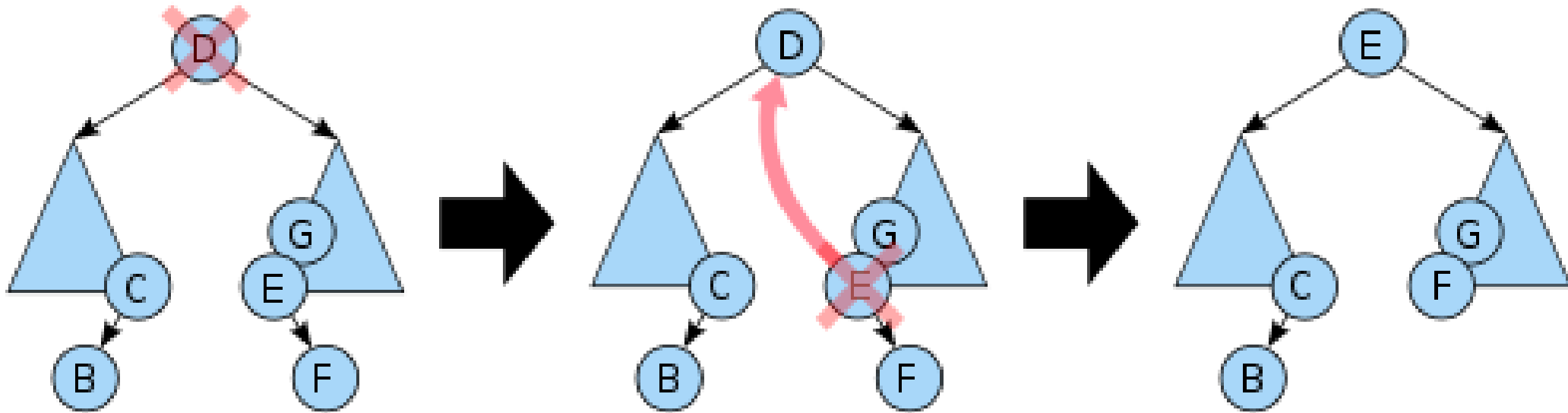
# Remover um nó que tem dois filhos



[Sedgewick & Wayne]

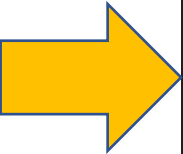


# Substituir pelo sucessor e apagá-lo



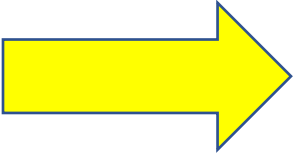
[Wikipedia]

# Substituir pelo sucessor e apagá-lo

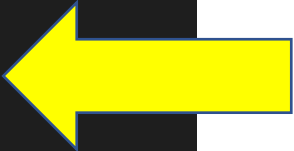


```
/* Node has TWO CHILDREN */  
/* Replace its item with the item of the next node in-order */  
/* And remove that node */  
  
_deleteNextNode(&(nodePointer->right), &(nodePointer->item));  
}
```

# Substituir pelo sucessor e apagá-lo



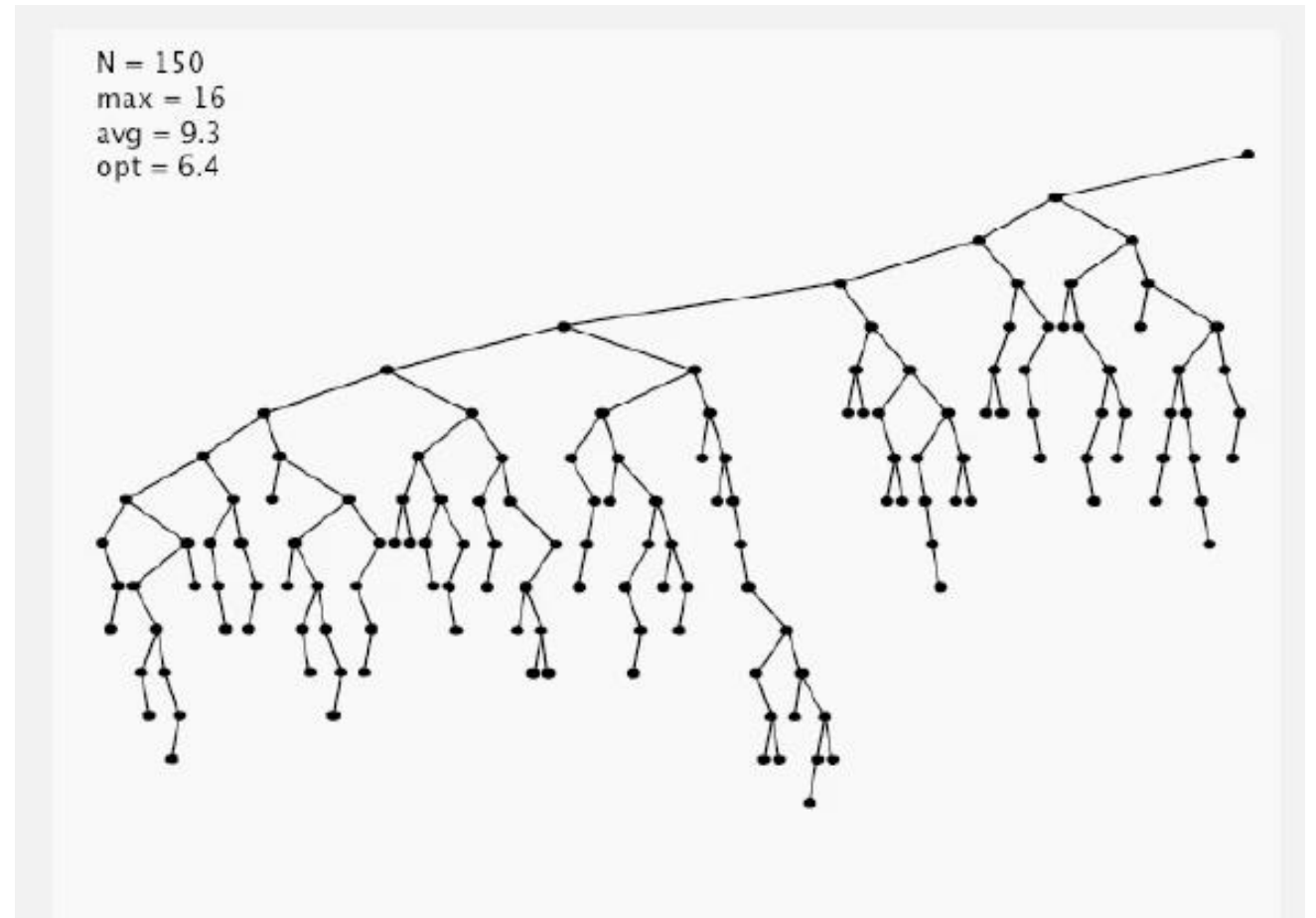
```
void _deleteNextNode(BSTree** pRoot, ItemType* pItem) {  
    if ((*pRoot)->left == NULL) {  
        // FOUND IT  
        BSTree* auxPointer = *pRoot;  
  
        *pItem = auxPointer->item;  
  
        *pRoot = auxPointer->right;  
  
        free(auxPointer);  
    } else {  
        _deleteNextNode(&((*pRoot)->left), pItem);  
    }  
}
```



# Após muitos apagamentos

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

[Sedgewick & Wayne]



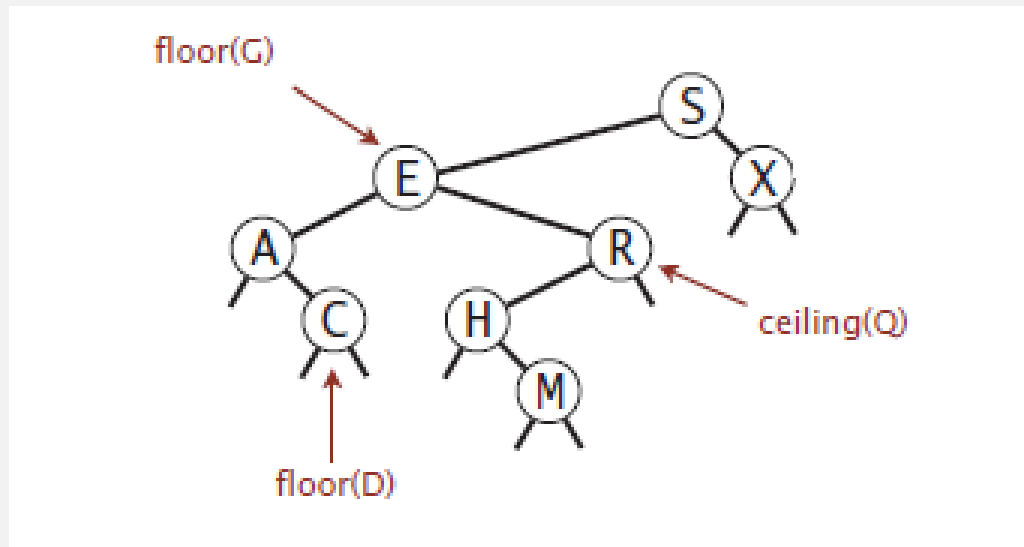
# Sugestões adicionais

– *Floor(k)* e *Ceiling(k)*

# Floor / Ceiling

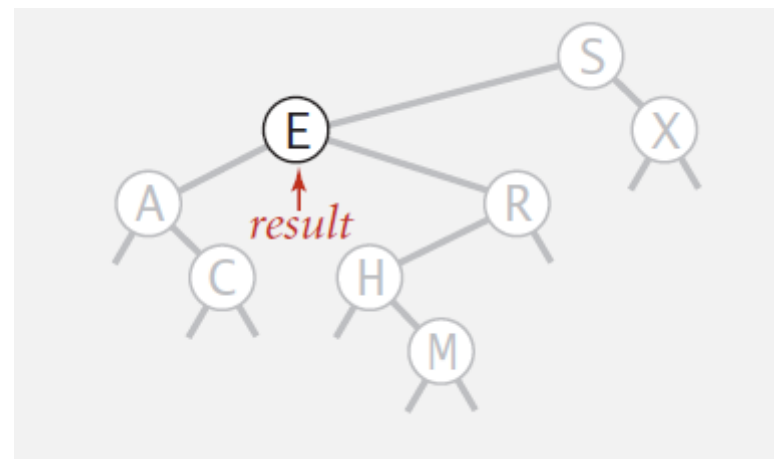
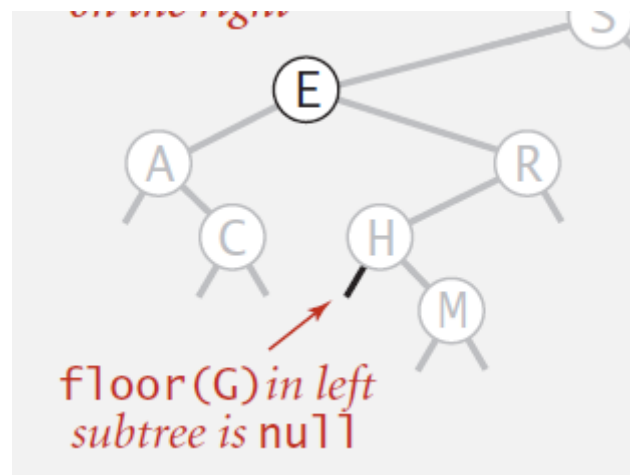
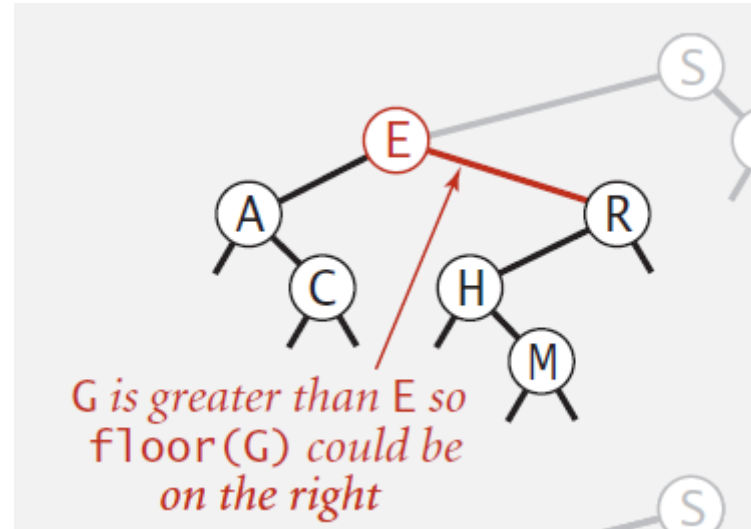
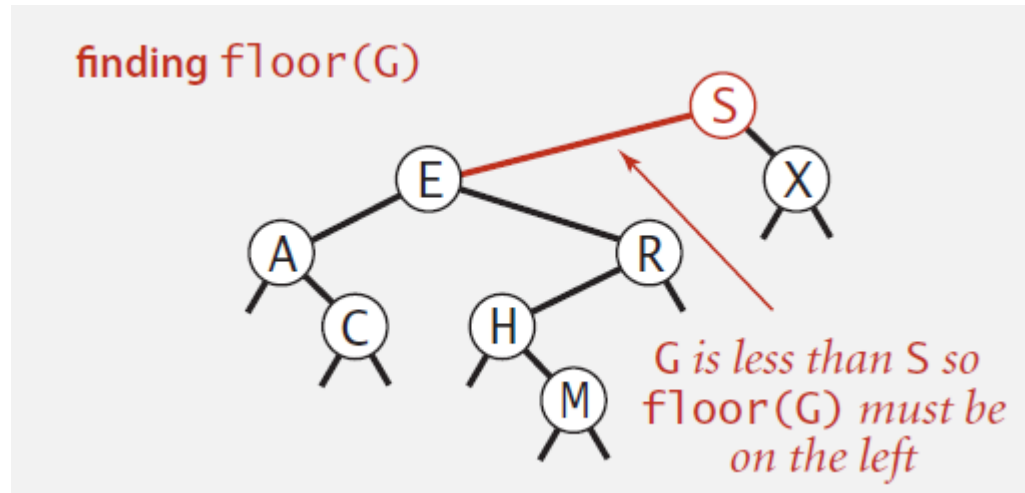
**Floor.** Largest key  $\leq$  a given key.

**Ceiling.** Smallest key  $\geq$  a given key.



[Sedgewick & Wayne]

# Floor



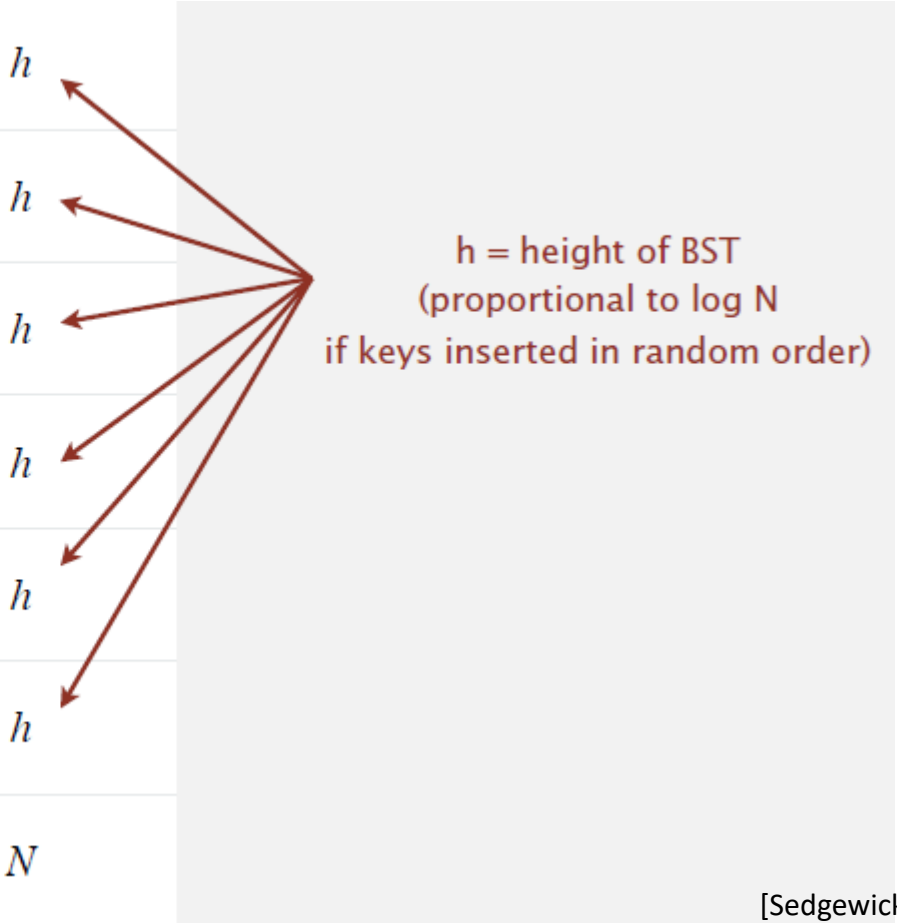
[Sedgewick & Wayne]

# Eficiência



# 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]

# 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**