

# Árvores Binárias II

08/11/2023

# Ficheiro ZIP

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

# Sumário

- Recap
- Representação de expressões algébricas
- Travessias **recursivas** em Pré-Ordem, Em-Ordem e Pós-Ordem
- Travessia iterativa, **por níveis**, usando uma **FILA / QUEUE**
- Travessia iterativa, em **pré-ordem**, usando uma **PILHA / STACK**
- O TAD **Árvore Binária de Procura** – BST : Binary Search Tree

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()**

# Determinar a **altura** de uma árvore

```
int TreeGetHeight(const Tree* root) {  
    if (root == NULL) return -1;  
  
    int heightLeftSubTree = TreeGetHeight(root->left);  
    int heightRightSubTree = TreeGetHeight(root->right);  
  
    if (heightLeftSubTree > heightRightSubTree) {  
        return 1 + heightLeftSubTree;  
    }  
  
    return 1 + heightRightSubTree;  
}
```

# Verificar se duas árvores são iguais

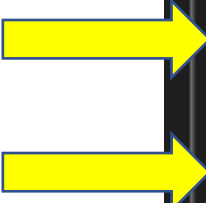
```
int TreeEquals(const Tree* root1, const Tree* root2) {  
    if (root1 == NULL && root2 == NULL) {  
        return 1;  
    }  
    if (root1 == NULL || root2 == NULL) {  
        return 0;  
    }  
    if (root1->item != root2->item) {  
        return 0;  
    }  
    return TreeEquals(root1->left, root2->left) &&  
           TreeEquals(root1->right, root2->right);  
}
```

# Um item **pertence** à árvore ? – Fizeram ?


- Desenvolver uma função recursiva



# Um item **pertence** à árvore ?



```
int TreeContains(const Tree* root, const ItemType item) {  
    if (root == NULL) return 0;  
    if (root->item == item) return 1;  
    return TreeContains(root->left, item) || TreeContains(root->right, item);  
}
```



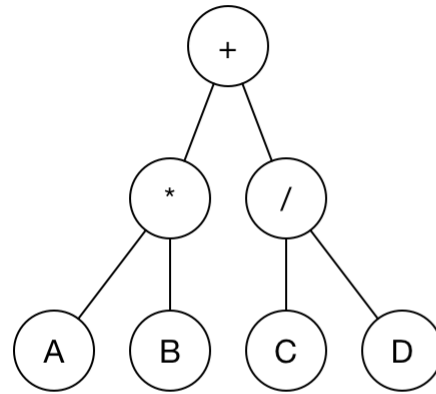
- Eficiência ?
  - Pode ser necessário visitar **todos os nós** !!
  - Como evitar ?

# Qual é o menor elemento ? – Fizeram ?

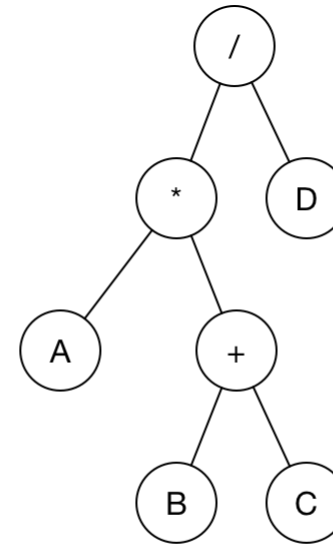
- Desenvolver uma função recursiva

# Qual é o menor elemento ? – Eficiência ?

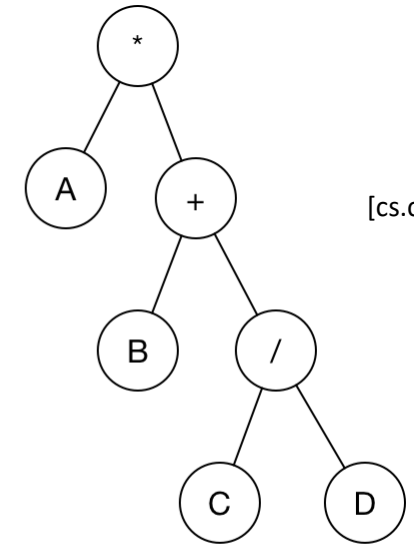
```
ItemType TreeGetMin(const Tree* root) {  
    if (root == NULL) {  
        return NO_ITEM;  
    }  
    ItemType min = root->item;  
    ItemType minLeftSubTree = TreeGetMin(root->left);  
    if (minLeftSubTree != NO_ITEM && minLeftSubTree < min) {  
        min = minLeftSubTree;  
    }  
    ItemType minRightSubTree = TreeGetMin(root->right);  
    if (minRightSubTree != NO_ITEM && minRightSubTree < min) {  
        min = minRightSubTree;  
    }  
    return min;  
}
```



$((A * B) + (C / D))$



$((A * (B + C)) / D)$



$(A * (B + (C / D)))$

[cs.colostate.edu]

# Representação de expressões

# Como representar uma expressão ?

- Notação **INFIXA** : operando **operador** operando
- Notação **PREFIXA** : **operador** operando operando
- Notação **POSFIXA** : operando operando **operador**

| PREFIX    | POSTFIX   | INFIX       |
|-----------|-----------|-------------|
| * + a b c | a b + c * | (a + b) * c |
| + a * b c | a b c * + | a + (b * c) |

# Outro exemplo

| PREFIX            | POSTFIX           | INFIX             |
|-------------------|-------------------|-------------------|
| + * * / a b c d e | a b / c * d * e + | a / b * c * d + e |

- Como ler cada string e efetuar as operações ?
- Como usar o TAD **STACK** ?

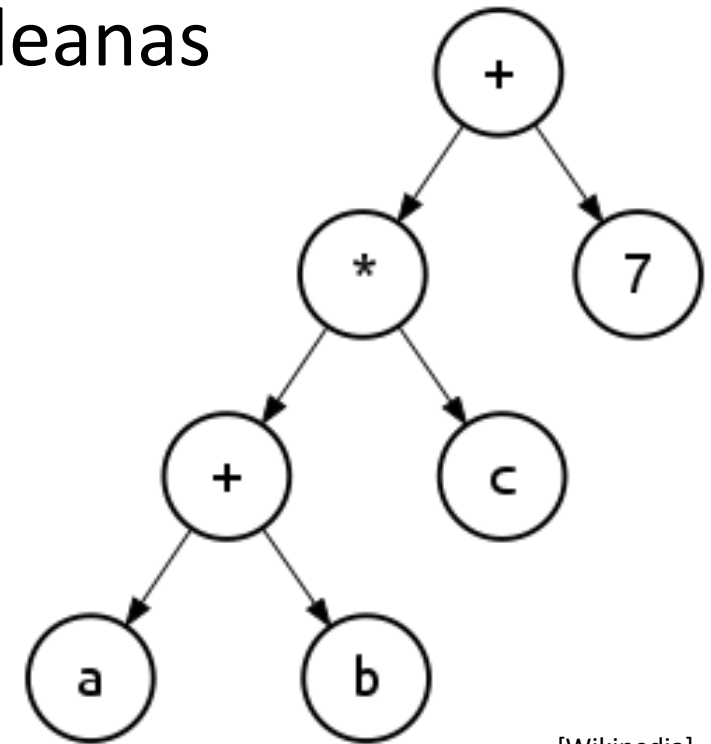
# Exemplo – POSTFIX

- $2\ 3\ +\ 8\ * = ?$
- STACK
- Ler da esquerda para a direita
- Empilhar os operandos
- Sempre que se encontra um operador :
  - retirar os dois operandos que estão no topo da STACK
  - empilhar o resultado
- Façam este exemplo !!

$$3 + 2 = 5 \times 8 = 40$$

# Representação usando uma árvore binária

- Expressões aritméticas / algébricas / booleanas
- Folha : **operando**
- Nó não terminal : **operador**
- Não são necessários parênteses !!
- Expressão ?
- Que **travessias** são possíveis ?



[Wikipedia]



# Travessias recursivas

- Travessia em **pré-ordem** (NLR)

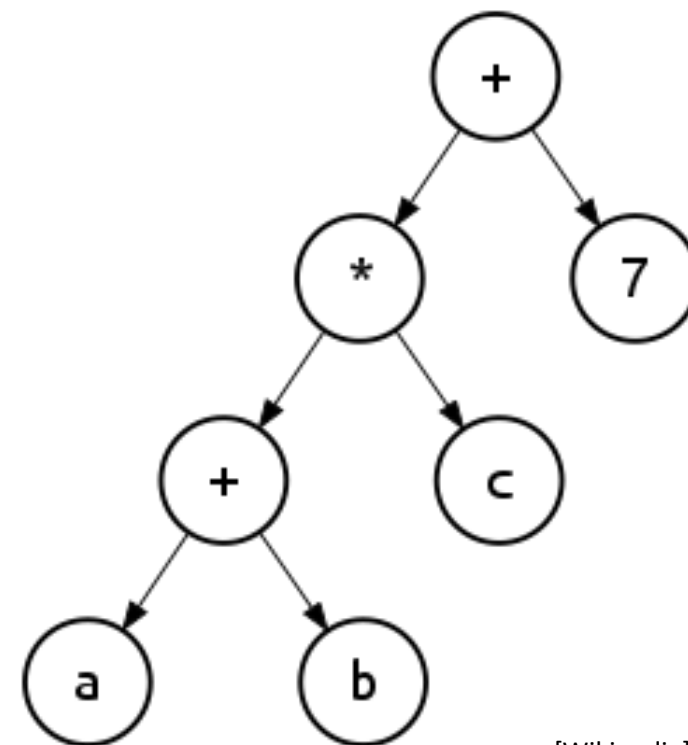
$+ * + a b c 7$

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

$a + b * c + 7$

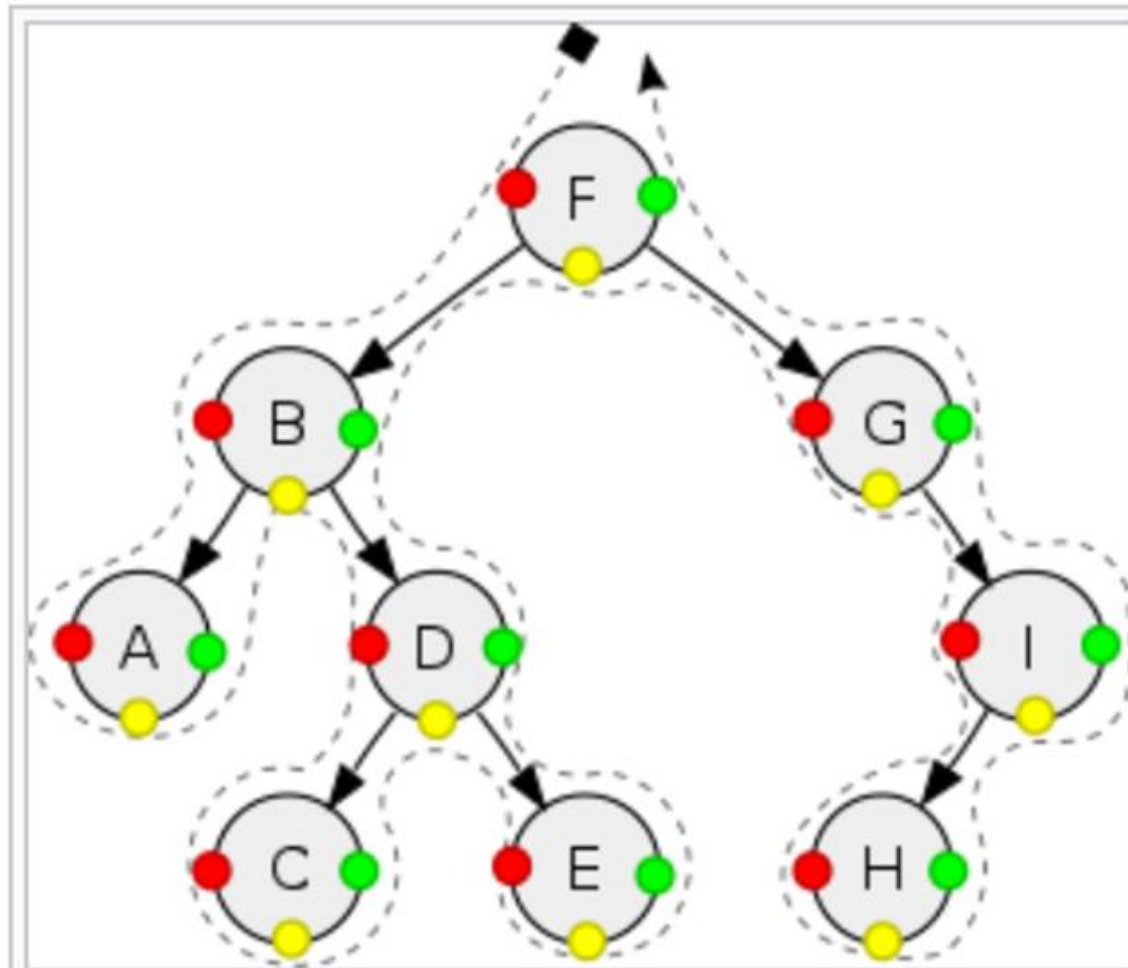
- Travessia em **pós-ordem** (LRN)

$a b + c * 7 +$



[Wikipedia]

# Travessias



Depth-first traversal of an example tree:

*pre-order (red)*: F, B, A, D, C, E, G, I, H;

*in-order (yellow)*: A, B, C, D, E, F, G, H, I;

*post-order (green)*: A, C, E, D, B, H, I, G, F.

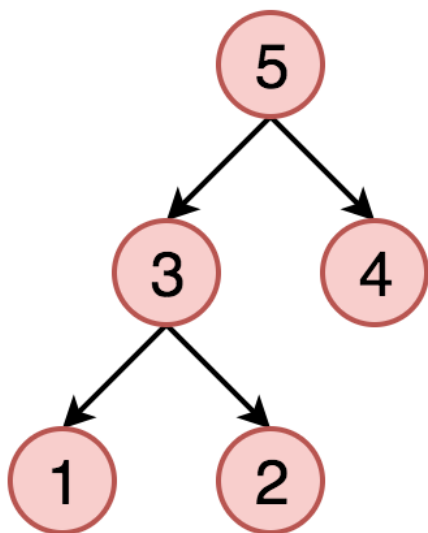


[Wikipedia]

# Ordem / Travessias em **profundidade**

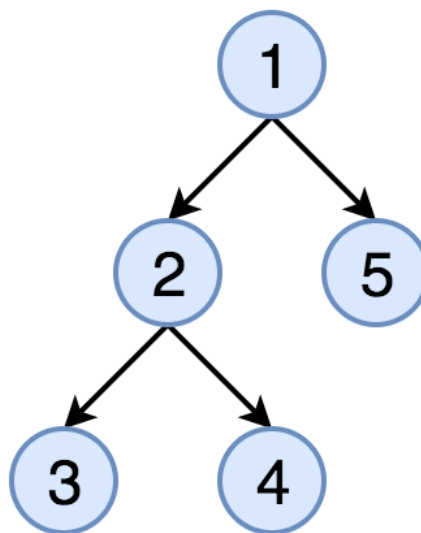
## DFS Postorder

Bottom -> Top  
Left -> Right



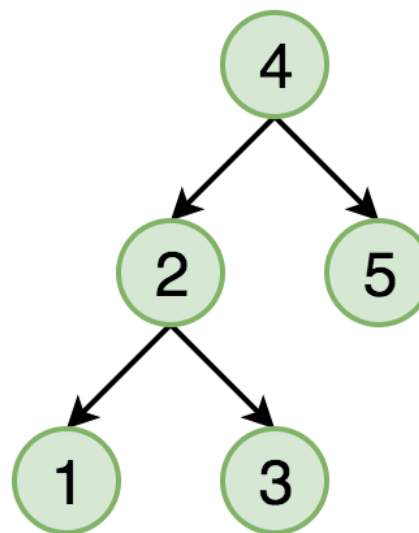
## DFS Preorder

Top -> Bottom  
Left -> Right



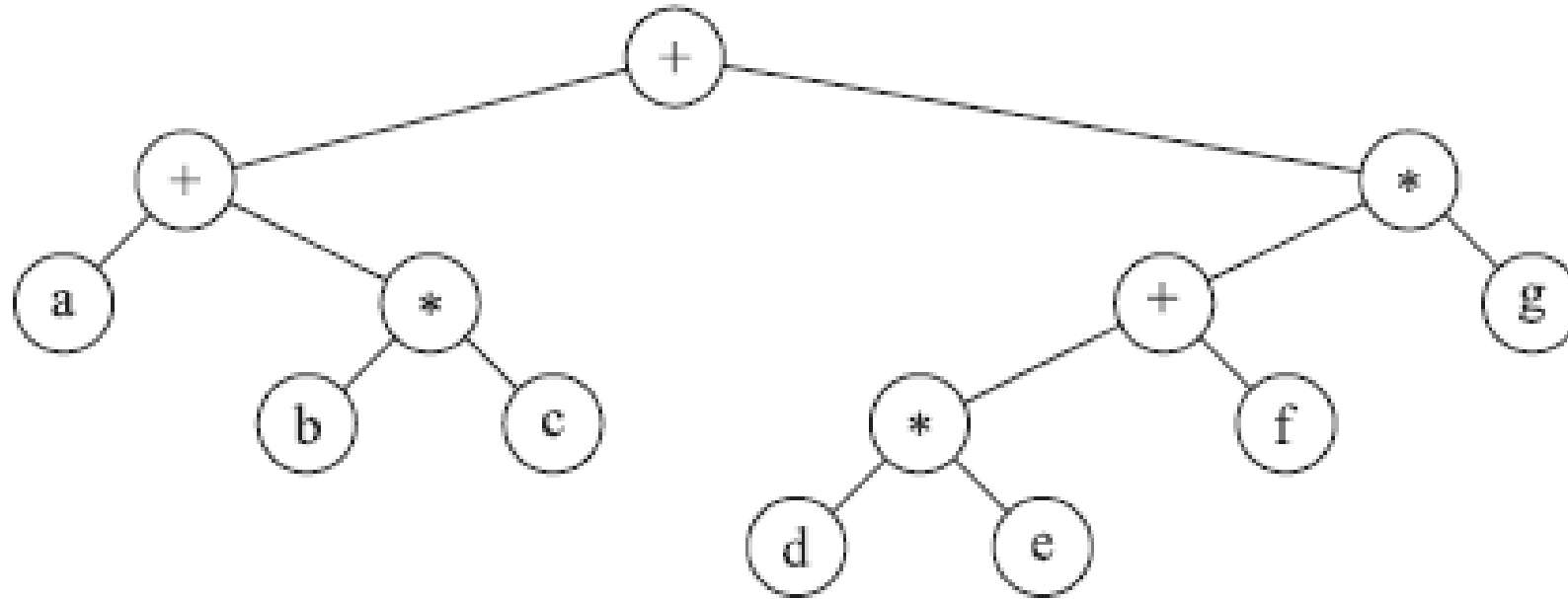
## DFS Inorder

Left -> Node -> Right



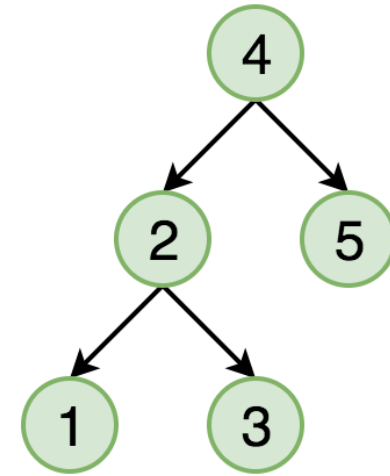
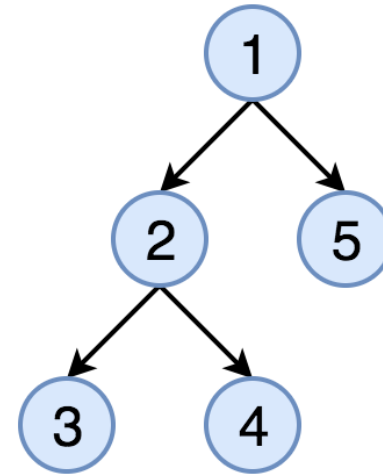
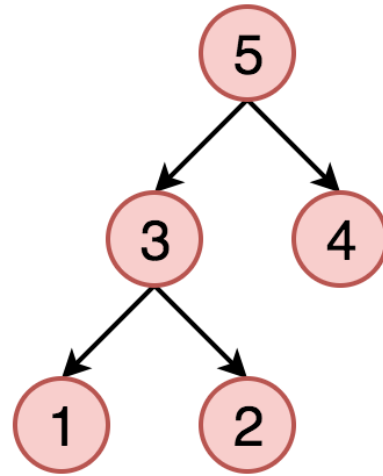
[zhang-xiao-mu.blog]

Tarefa : Escrever a expressão nas 3 notações



**Figure 4.14** Expression tree for  $(a + b * c) + ((d * e + f) * g)$

[Weiss]



[zhang-xiao-mu.blog]

# Travessias Recursivas

# Travessias

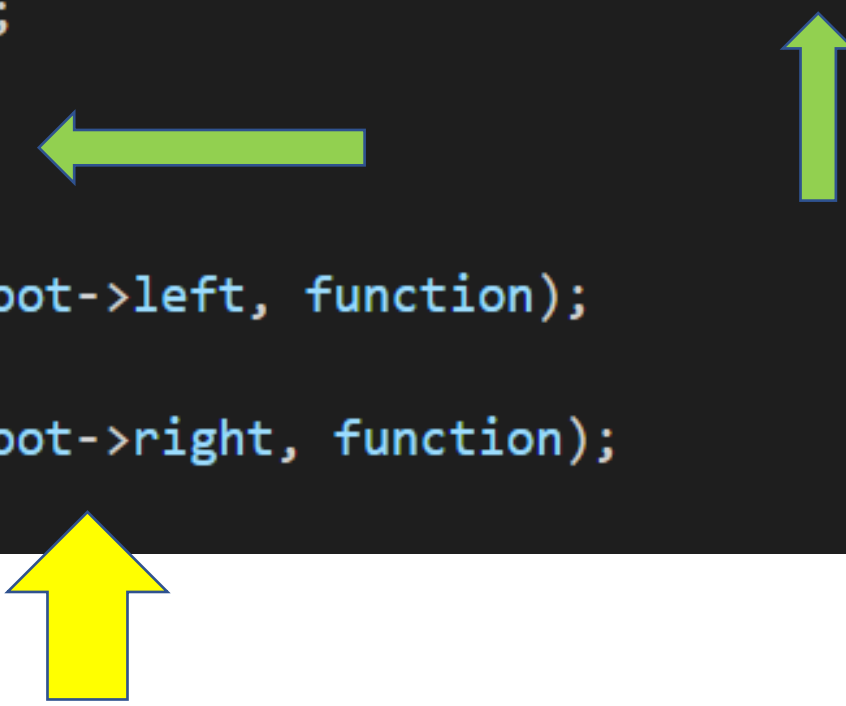
- Visitar cada **nó** exatamente **uma vez**
- E efetuar algum tipo de **processamento**
  - Imprimir
  - Alterar o valor
  - Escrever em ficheiro
  - ...
- Vários tipos / **ordens** de travessia

# Travessias recursivas

- **NLR – Pré-Ordem**  
processar o nó **raiz**  
chamada recursiva para a **subárvore esquerda**  
chamada recursiva para a **subárvore direita**
- **LNR – Em-Ordem**  
chamada recursiva para a **subárvore esquerda**  
processar o nó **raiz**  
chamada recursiva para a **subárvore direita**
- **LRN – Pós-Ordem**  
...

# 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);  
}
```





# Exemplos de utilização

```
void printInteger(int* p) { printf("%d ", *p); }  
  
void multiplyIntegerBy2(int* p) { *p *= 2; }
```

```
printf("PRE-Order traversal : ");  
  
TreeTraverseInPREOrder(tree, printInteger);
```



```
printf("Multiply each value by 2\n");  
  
TreeTraverseInPREOrder(tree, multiplyIntegerBy2);
```

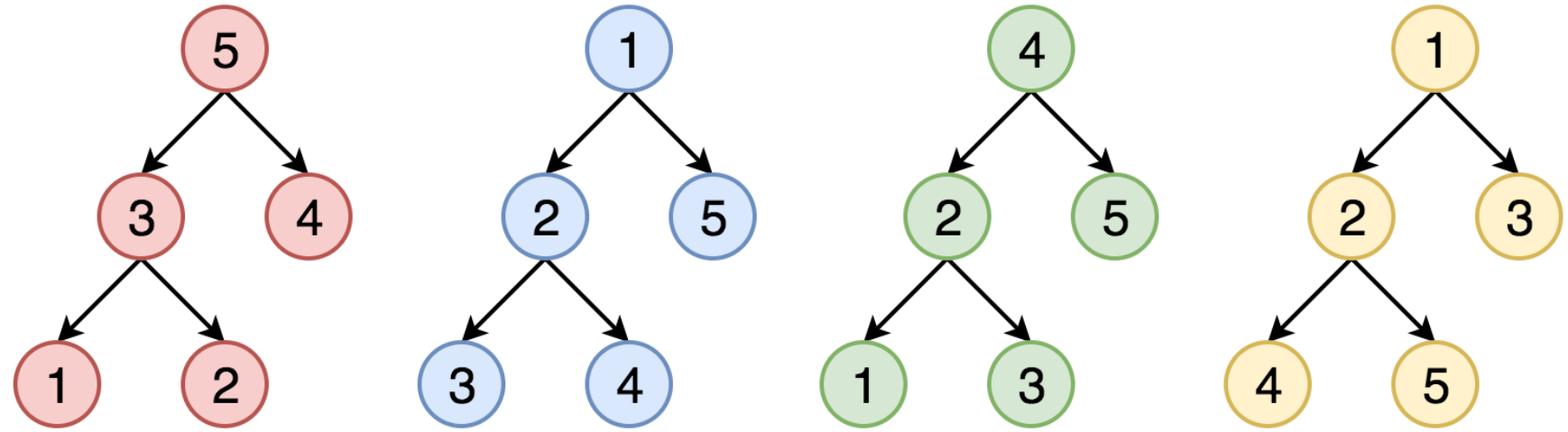


# Tarefa : Implementar as travessias recursivas

- Travessia em pré-ordem
- Travessia em-ordem
- Travessia em pós-ordem
- Atenção à ordem de visita das subárvores !!
- Listar os elementos de uma árvore e confirmar a ordem

# Exemplo de Aplicação – Analisar o código

- **Registar** a informação de uma árvore num **ficheiro**, usando uma travessia em **pré-ordem**
- **Recuperar** a informação de uma árvore a partir de um **ficheiro**, usando uma travessia em **pré-ordem**

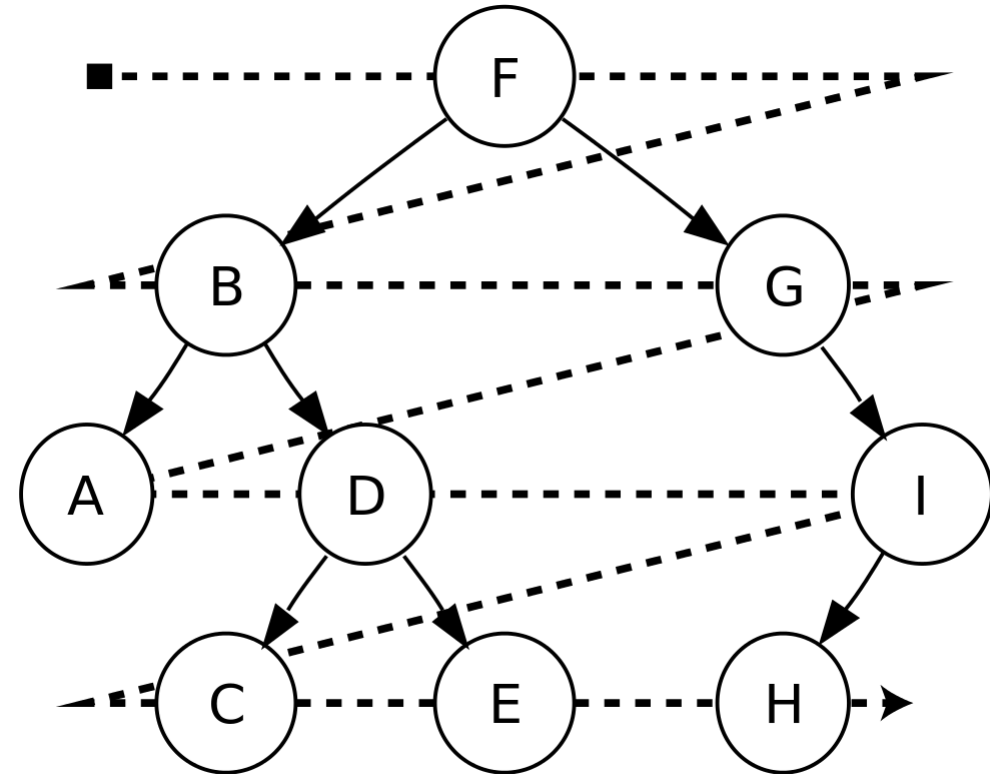


[zhang-xiao-mu.blog]

# Travessias Iterativas

# Travessia por níveis

- Mais um tipo de travessia
- **Breadth-First** traversal
- Como são visitados os nós da árvore ?
- A solução habitual usa uma **FILA / QUEUE**

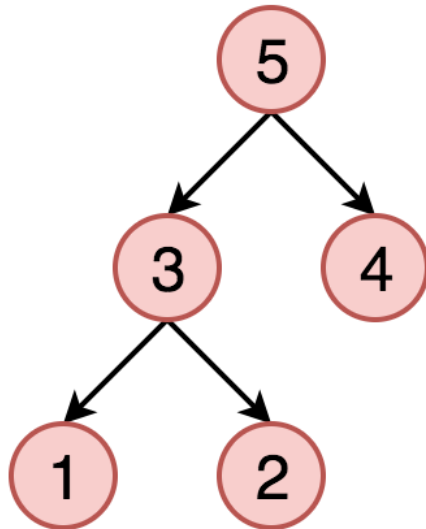


[Wikipedia]

# Ordem / Travessias

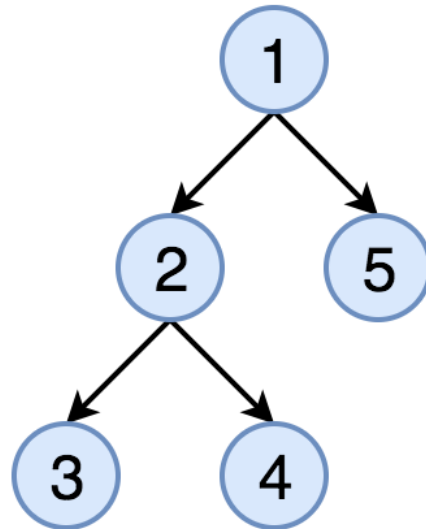
## DFS Postorder

Bottom -> Top  
Left -> Right



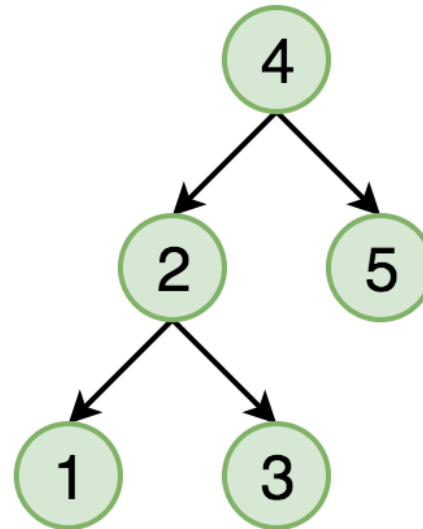
## DFS Preorder

Top -> Bottom  
Left -> Right



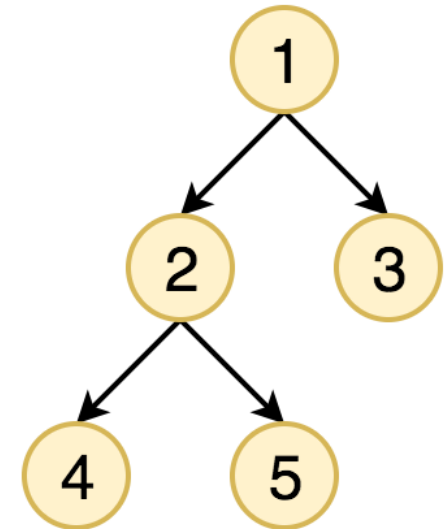
## DFS Inorder

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

# Estratégia básica

- Criar um conjunto vazio de ponteiros
- Adicionar o ponteiro para a raiz da árvore
- Enquanto o conjunto não for vazio

Retirar do conjunto o ponteiro para o próximo nó



Processar esse ponteiro / nó

Se necessário, adicionar ponteiro(s) ao conjunto



- Destruir o conjunto vazio




# 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);  
}
```

# Travessia em Pré-Ordem – STACK



```
while (StackIsEmpty(stack) == 0) {  
    Tree* p = StackPop(stack);   
  
    function(&(p->item));  
  
    // Pay attention to the push order   
    if (p->right != NULL) {   
        StackPush(stack, p->right);  
    }  
    if (p->left != NULL) {   
        StackPush(stack, p->left);  
    }  
}
```

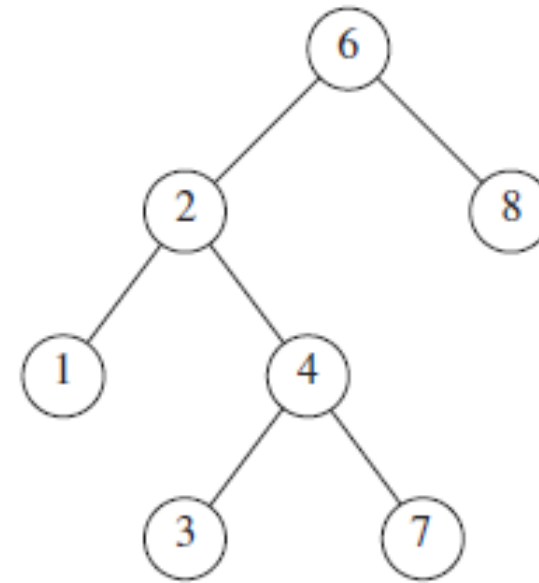
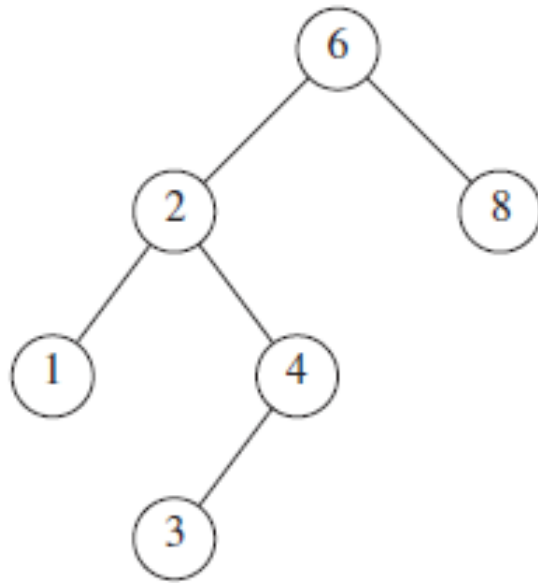
# Tarefa – Analisar o código

- Travessia iterativa EM-ORDEM, usando uma PILHA / STACK
- Travessia iterativa em PÓS-ORDEM, usando uma PILHA / STACK

# Árvores Binárias de Procura (ABP)

## – Binary Search Trees (BST)

# Critério de **ordem** ?



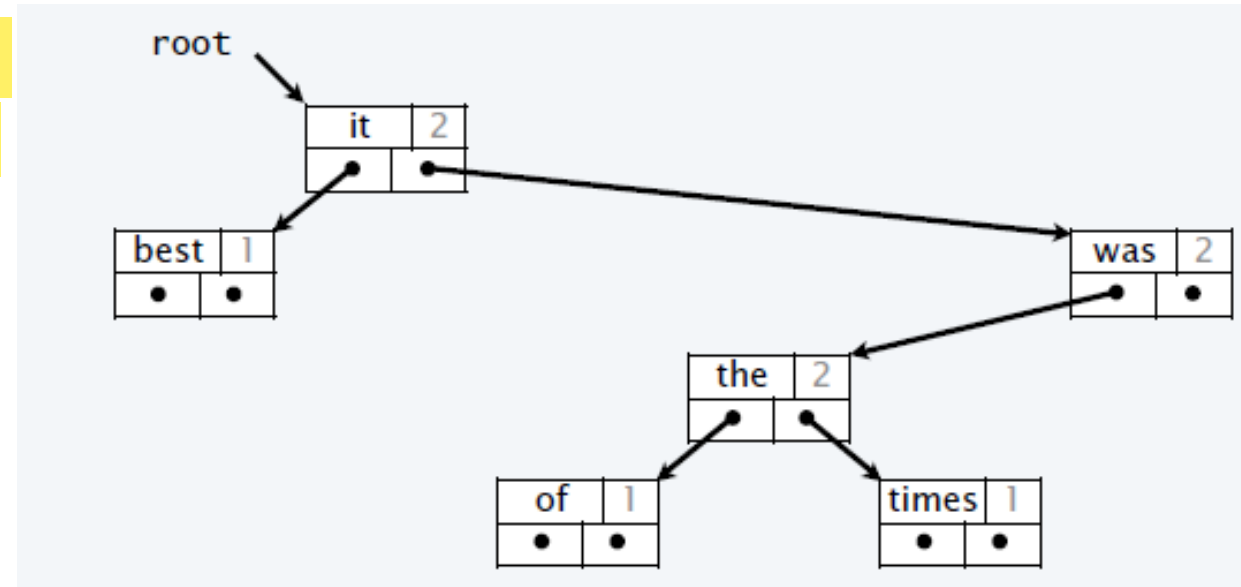
- Qual das árvores está **ordenada** ?
- Que operações são mais eficientes por existir uma ordem ?

# 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

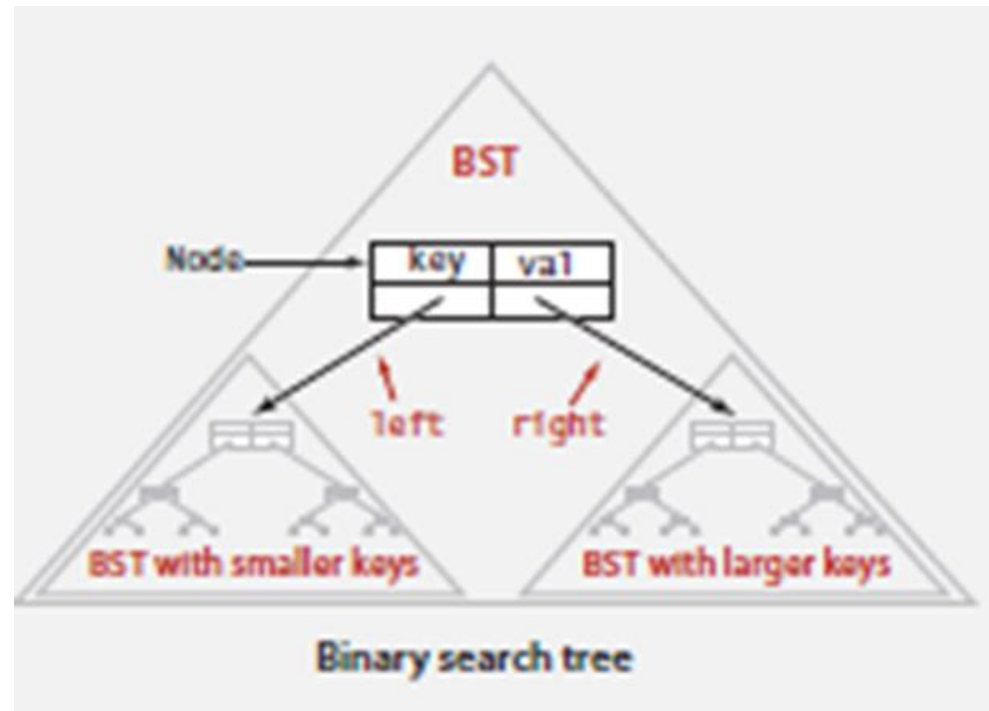
- Para cada nó, os elementos da sua **subárvore esquerda** são **inferiores** ao nó
- E os elementos da sua **subárvore direita** são **superiores** ao nó
- **Não** há elementos **repetidos** !!
- A organização da árvore depende da **sequência de inserção** dos elementos



[Sedgewick & Wayne]



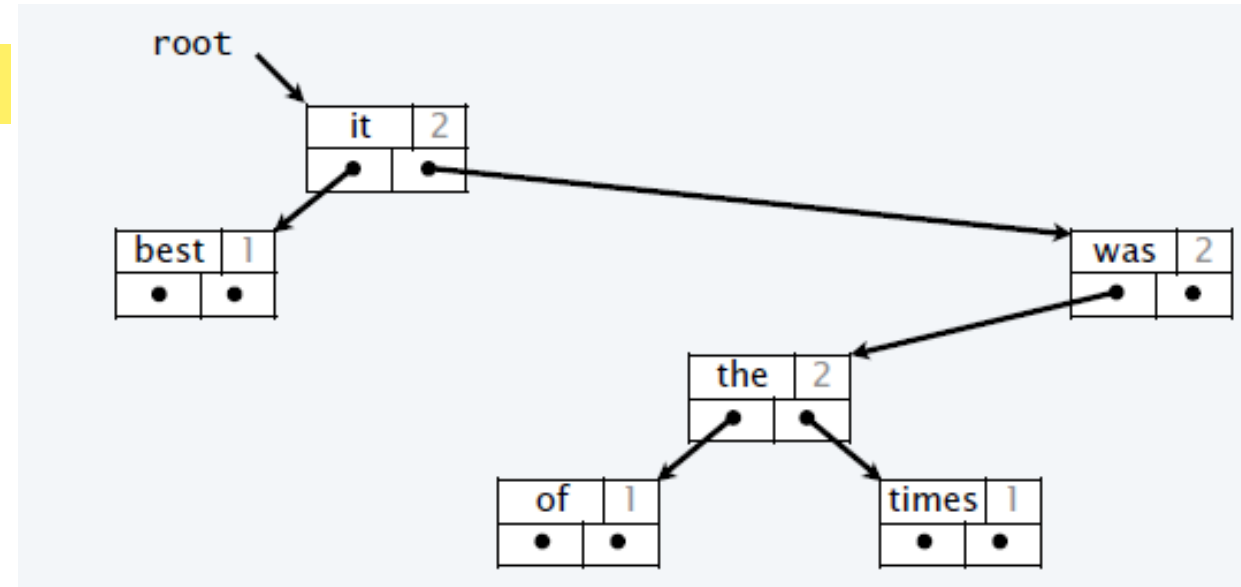
# Critério de ordem – Definição recursiva



[Sedgewick & Wayne]

# Operações habituais

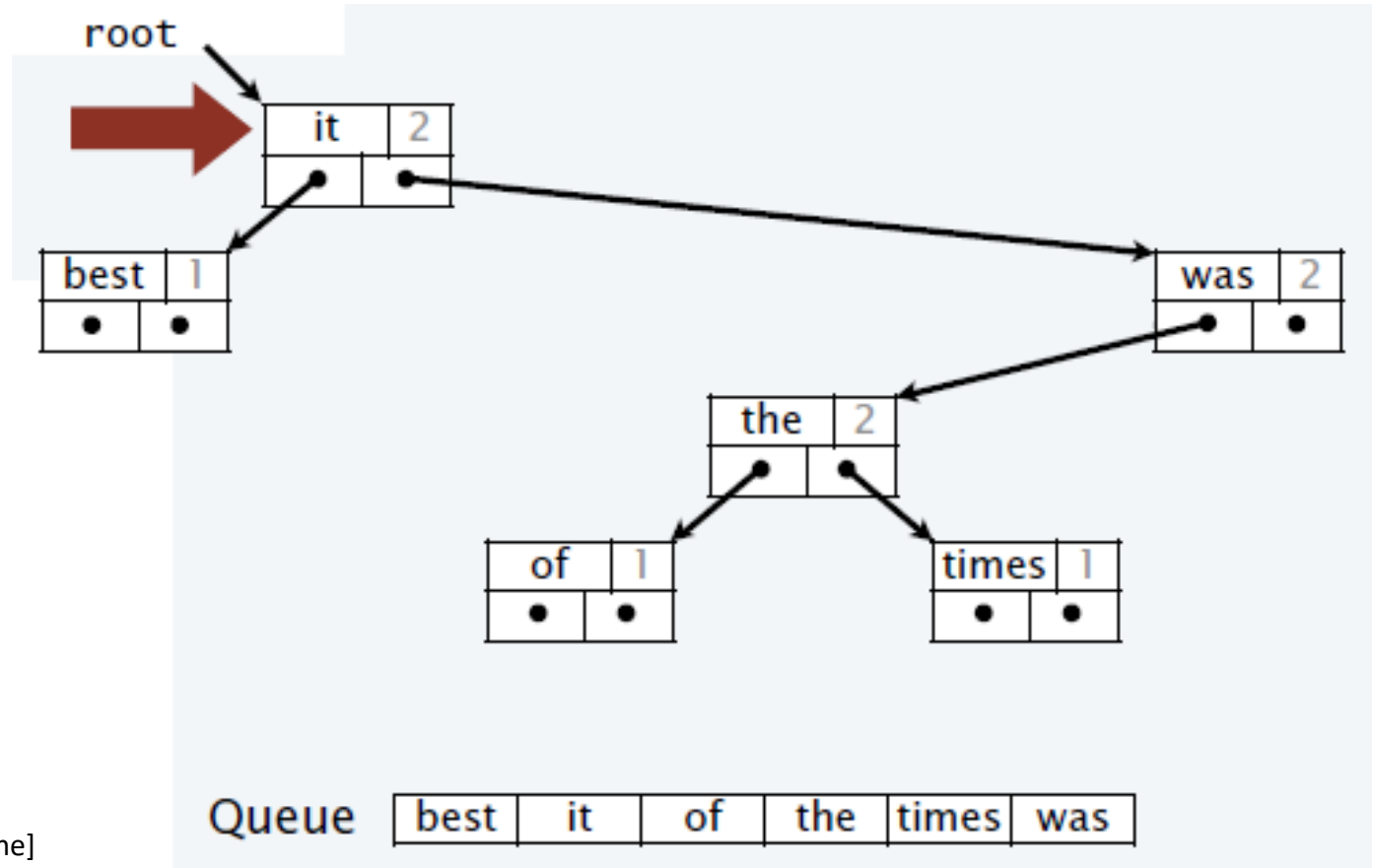
- O **item** armazenado em cada nó é, em geral, um par (**chave, valor**)
- Procurar
- Adicionar
- Alterar
- Remover
- Visitar em-ordem



[Sedgewick & Wayne]

# Travessia em-ordem

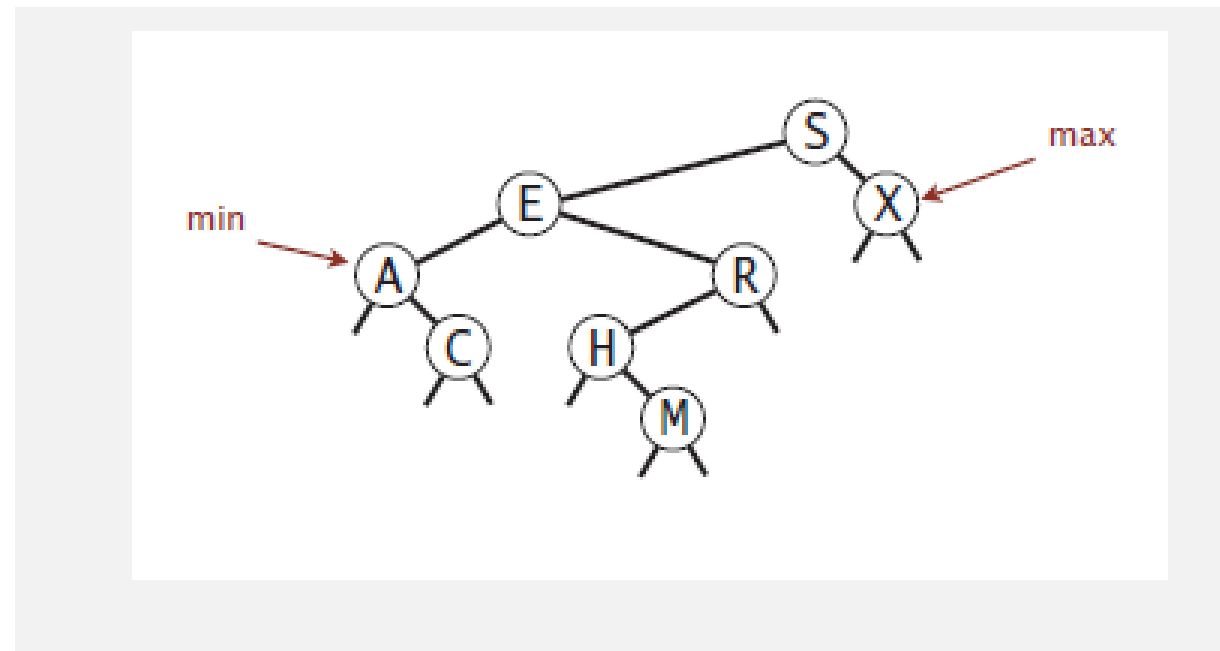
- **Exemplo** : preencher uma fila com os itens ordenados



# Algumas funções

# Menor item ? / Maior item ?

- Como fazer ?
- Eficiência ?



[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;  
}
```

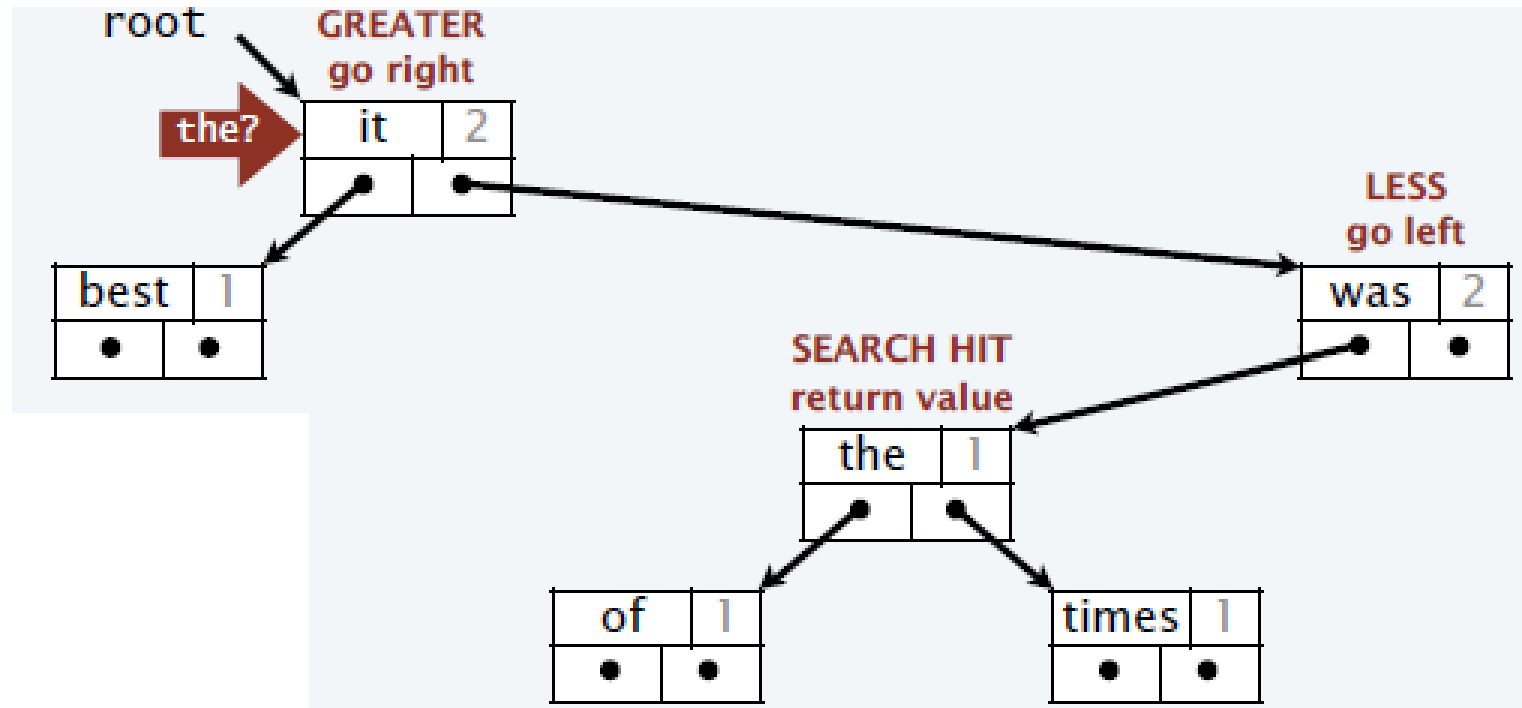




# getMax()

- Tarefa
- Versão recursiva

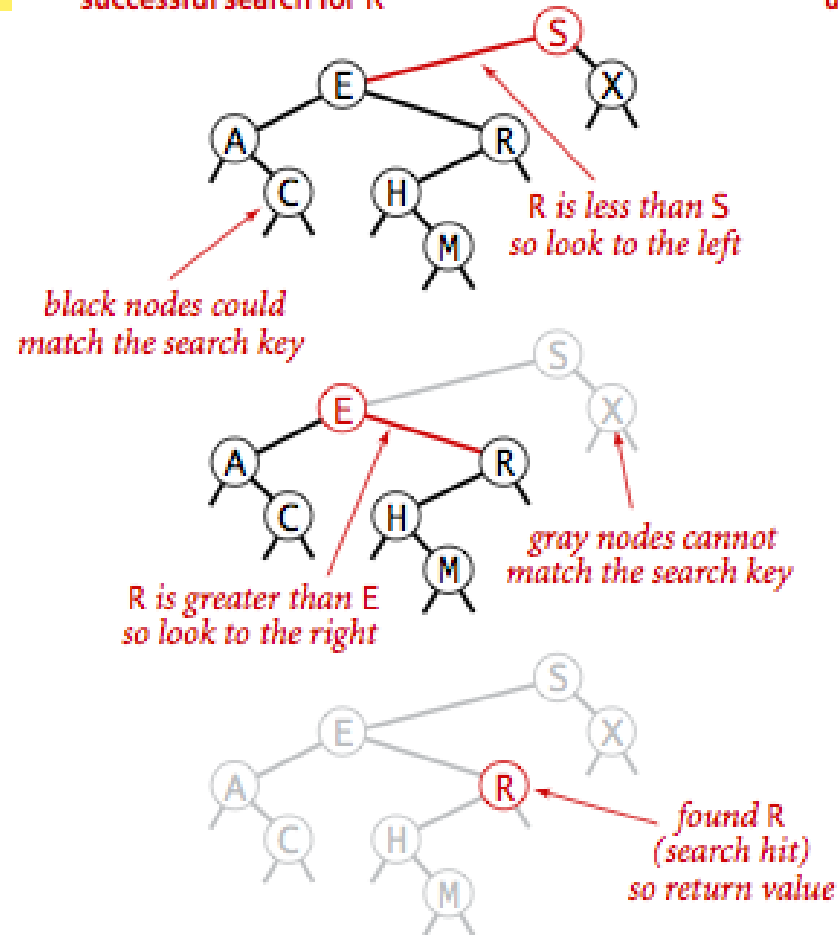
# Procurar



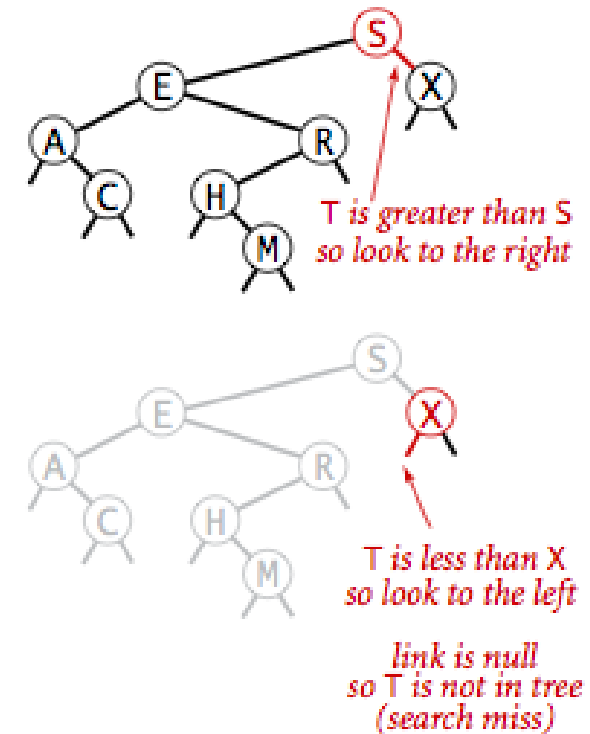
[Sedgewick & Wayne]

# Exemplos

successful search for R



unsuccessful search for T

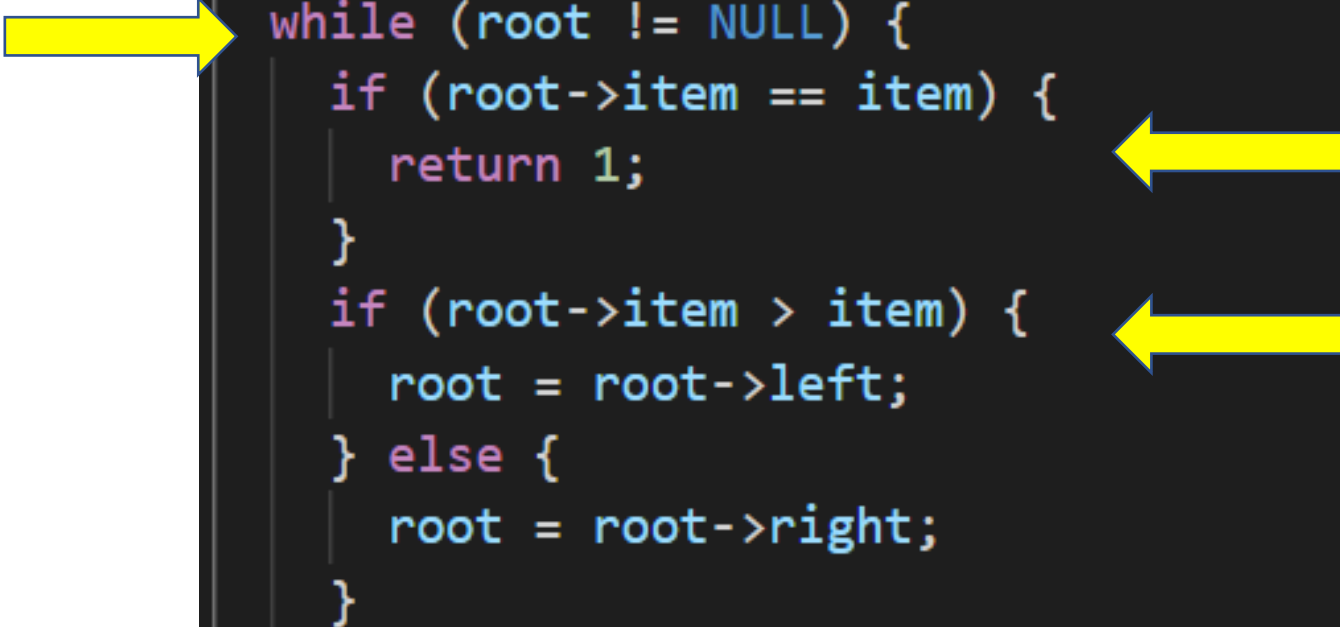


Successful (left) and unsuccessful (right) search in a BST

[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