

Árvores Binárias II

Joaquim Madeira

30/04/2020

Ficheiro ZIP

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

Sumário

- Recap
- Representação de expressões algébricas
- Travessias em Pré-Ordem, Em-Ordem e Pós-Ordem
- Travessia por níveis usando uma FILA / QUEUE
- Travessias usando uma PILHA / STACK
- Aplicação: registo e leitura de uma árvore usando um ficheiro

Let's
RECAP

Recapitulação

Tipos de árvores

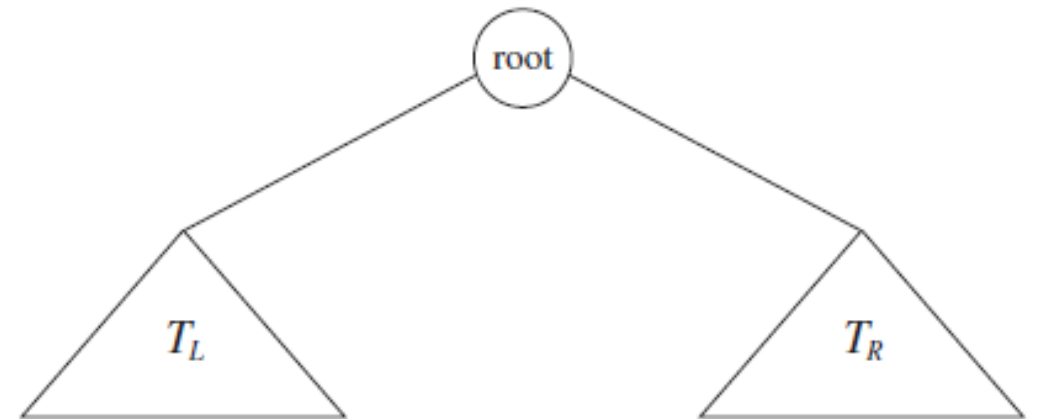
- Árvores **orientadas** vs não orientadas
- Árvores **binárias**, ternárias, quaternárias, ... , **m-árias**
- Árvores binárias **ordenadas** – Como ?
- Árvores binárias **equilibradas** – Como ?

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()**


Definição recursiva

- Uma árvore binária é formada por um conjunto finito de nós ($n \geq 0$)
- Uma árvore binária é **vazia**
- **OU** é constituída por um nó **raiz** que referencia **duas (sub-)árvores** binárias disjuntas (**SAEsq** e **SADir**)
 - Arcos orientados para a SAEsq e para a SADir





[Weiss]

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);  
}
```

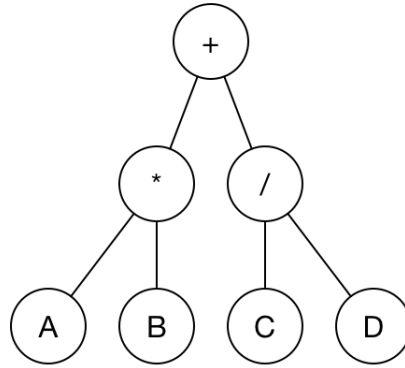


Qual é o **menor** elemento ? – Fizeram ?

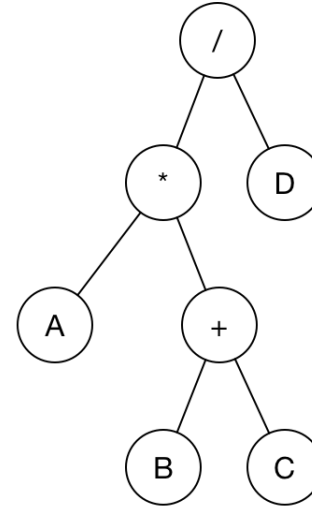
- Desenvolver uma função recursiva

Qual é o **menor** elemento ?

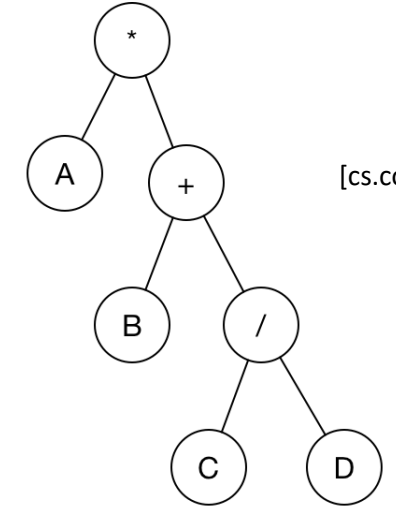
```
ItemType TreeGetMin(const Tree* root) {  
    if (root == NULL) {  
        return NO_ITEM;   
    }  
    ItemType min = root->item;  
    ItemType minLeftSubTree = TreeGetMin(root->left);  
    if (minLeftSubTree != NO_ITEM && minLeftSubTree < min) {  
        min = minLeftSubTree;    
    }  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

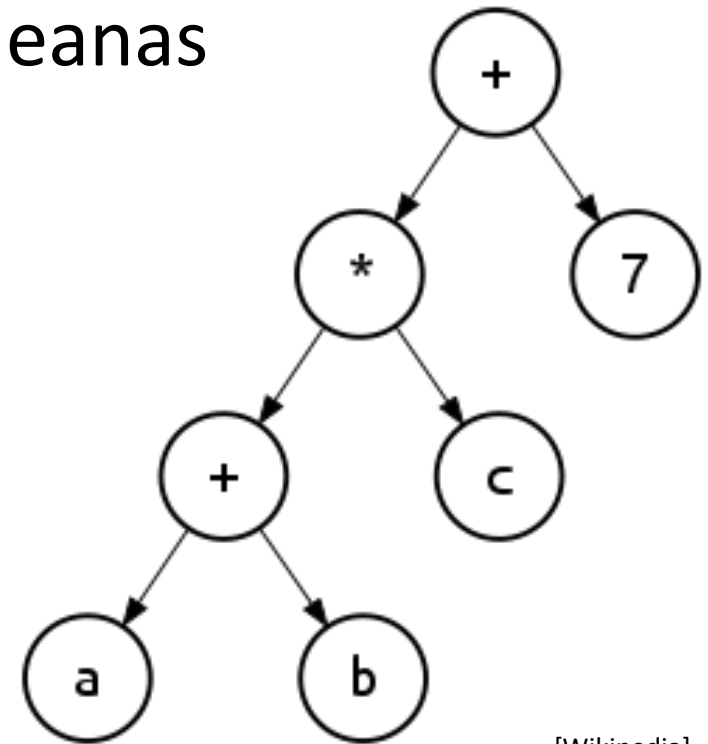
- Como ler cada string e efetuar as operações ?
- Como usar o TAD **STACK** ?

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 !!

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**)

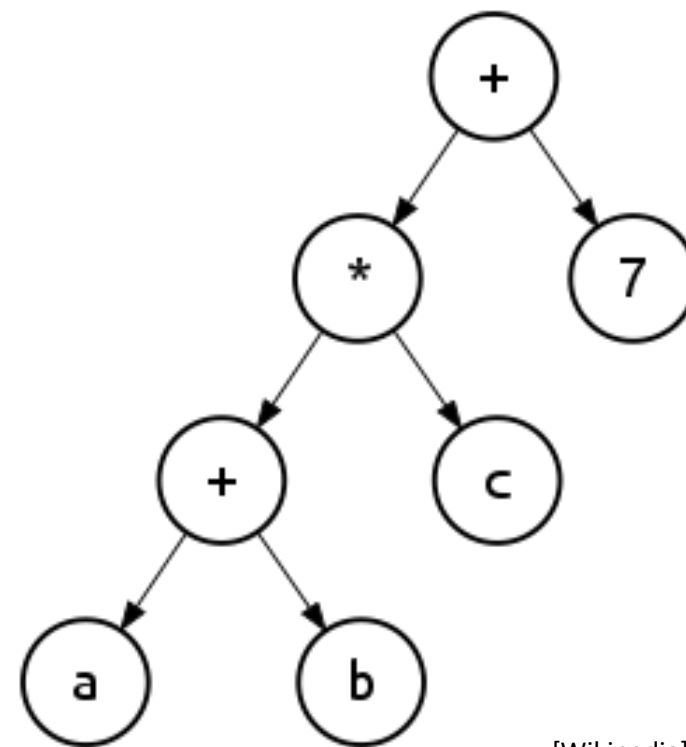
+ * + a b c 7

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

a + b * c + 7

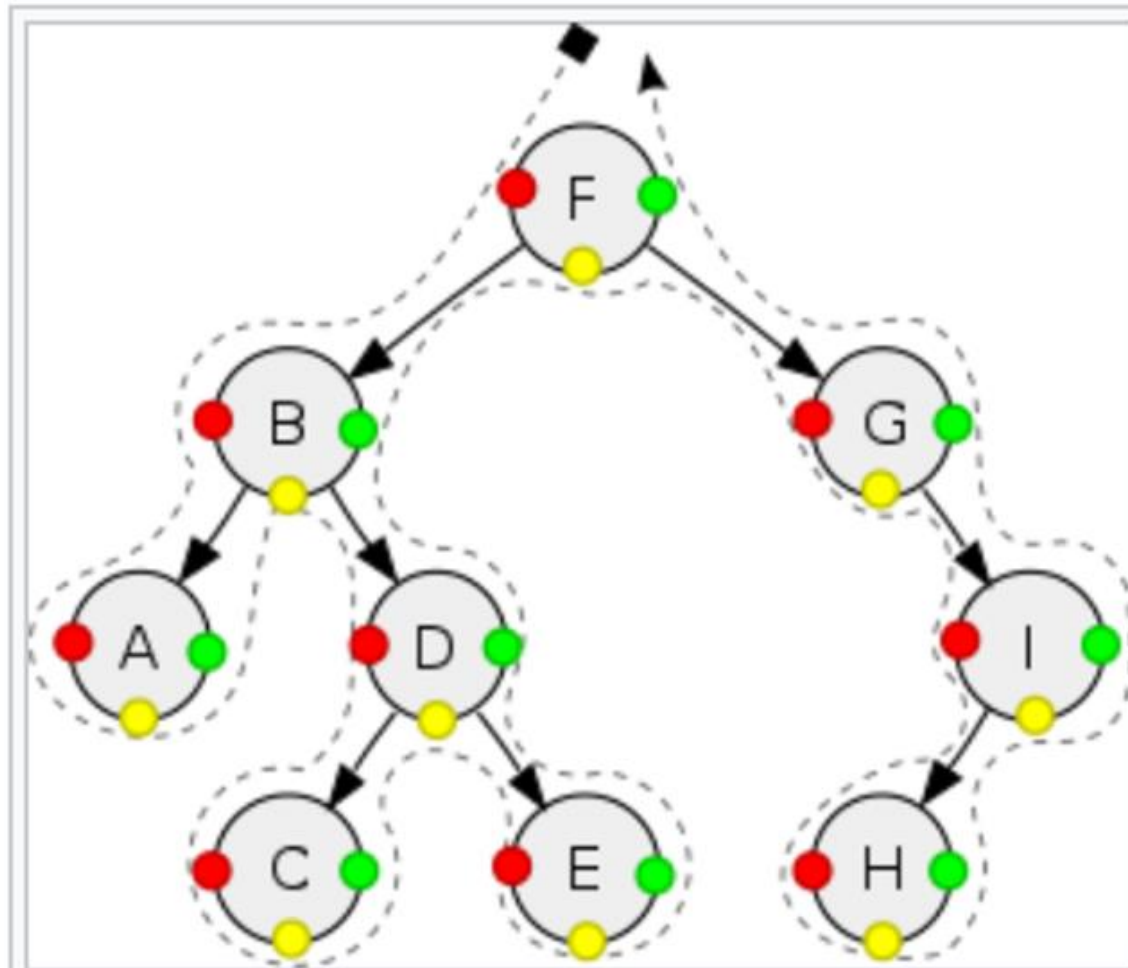
- Travessia em **pós-ordem** (**LRN**)

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.

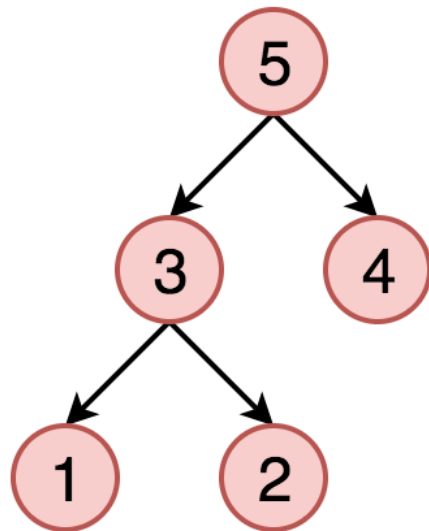


[Wikipedia]

Ordem / Travessias em **profundidade**

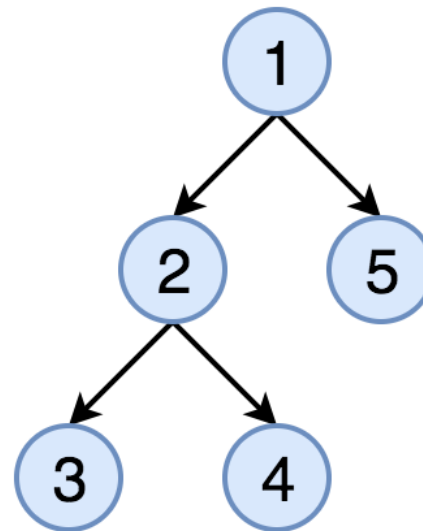
DFS Postorder

Bottom -> Top
Left -> Right



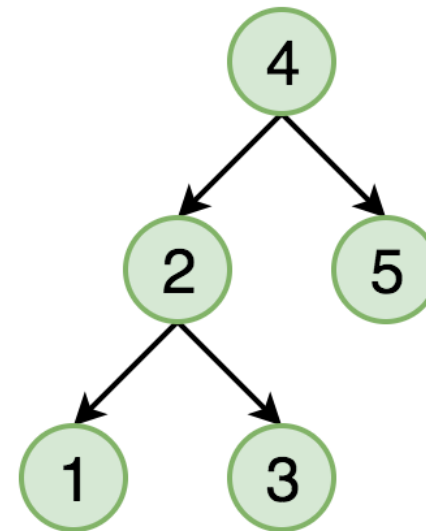
DFS Preorder

Top -> Bottom
Left -> Right



DFS Inorder

Left -> Node -> Right



[zhang-xiao-mu.blog]

Escrever a expressão nas 3 notações

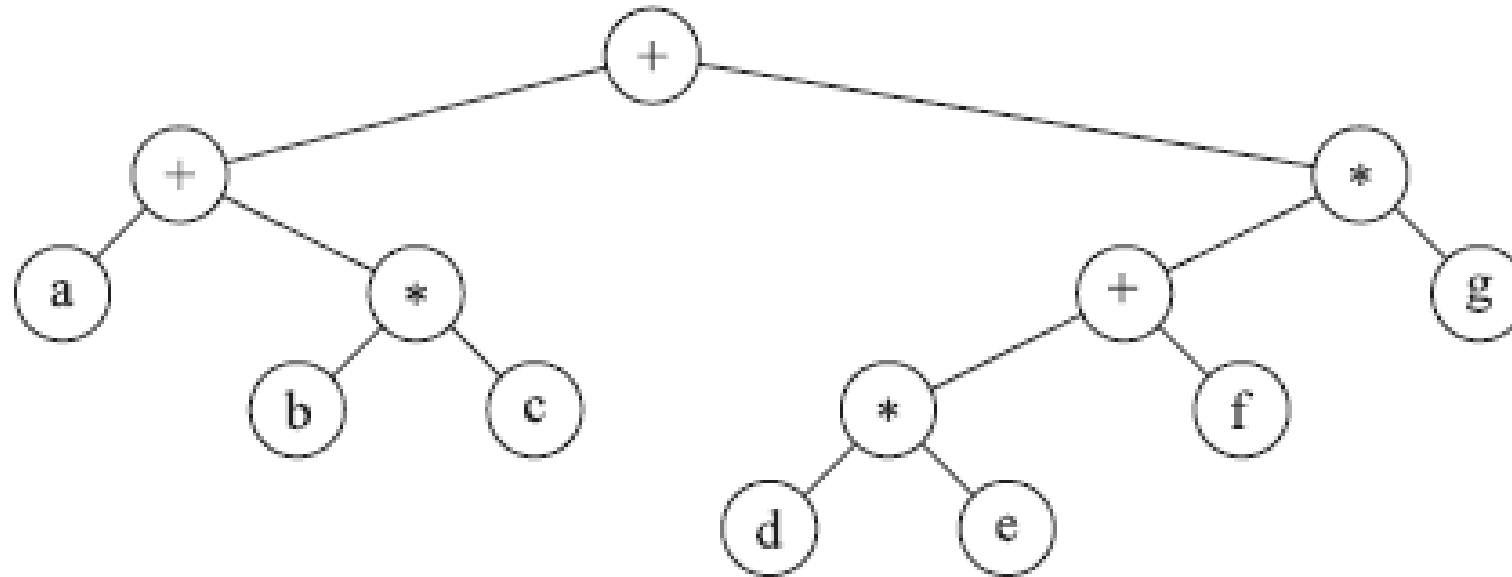
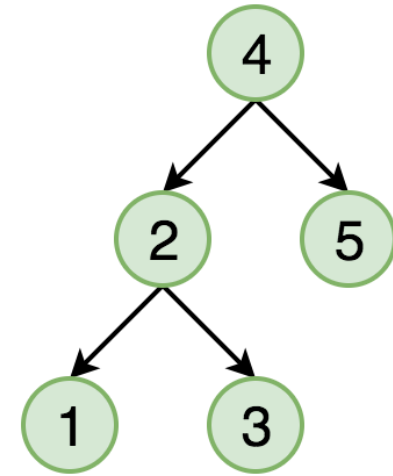
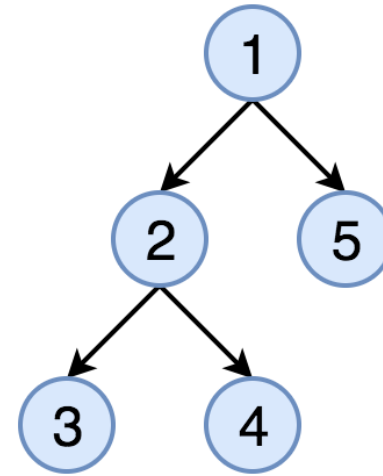
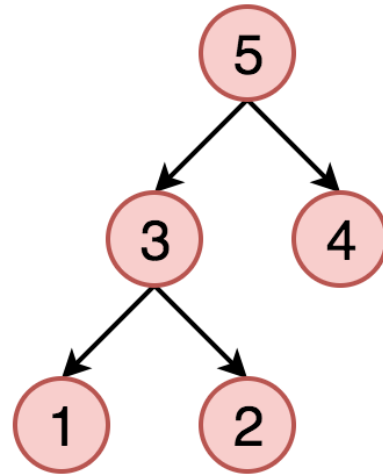


Figure 4.14 Expression tree for $(a + b * c) + ((d * e + f) * g)$

[Weiss]



[zhang-xiao-mu.blog]

Travessias Recursivas

Travessias

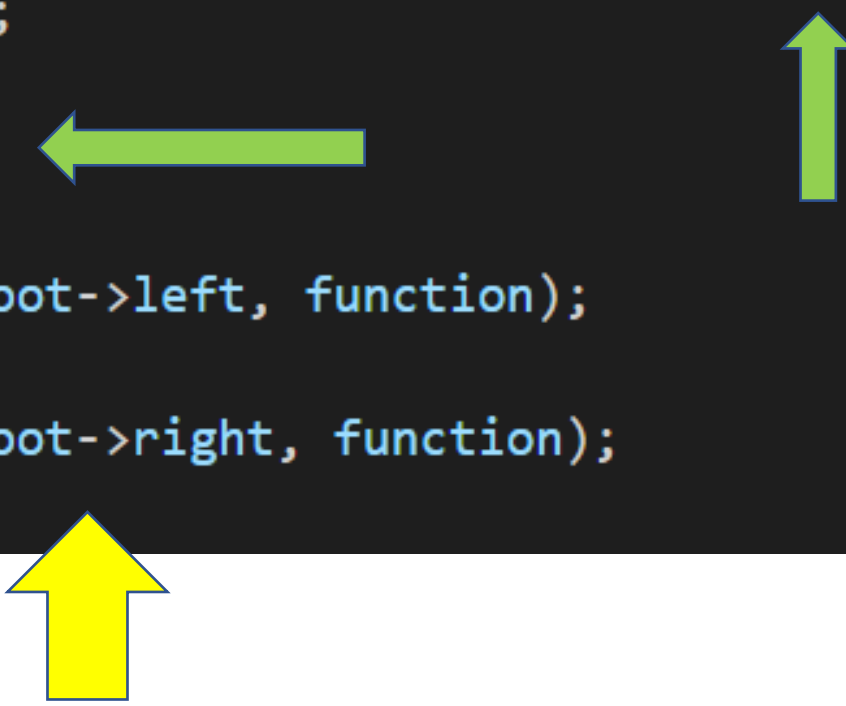
- Visitar cada **nó** exatamente **uma vez**
- E efetuar algum tipo de **processamento**
 - Imprimir
 - Alterar o valor
 - Escrever em ficheiro
 - ...
- Vários tipos / ordens de travessia

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);
```

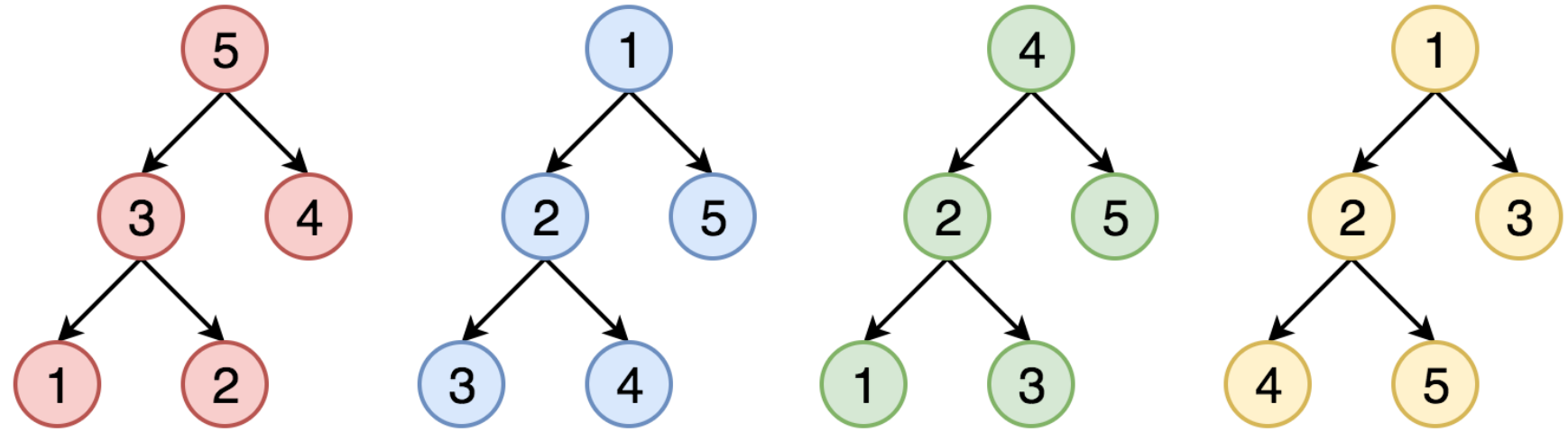


```
printf("Multiply each value by 2\n");  
  
TreeTraverseInPREOrder(tree, multiplyIntegerBy2);
```



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

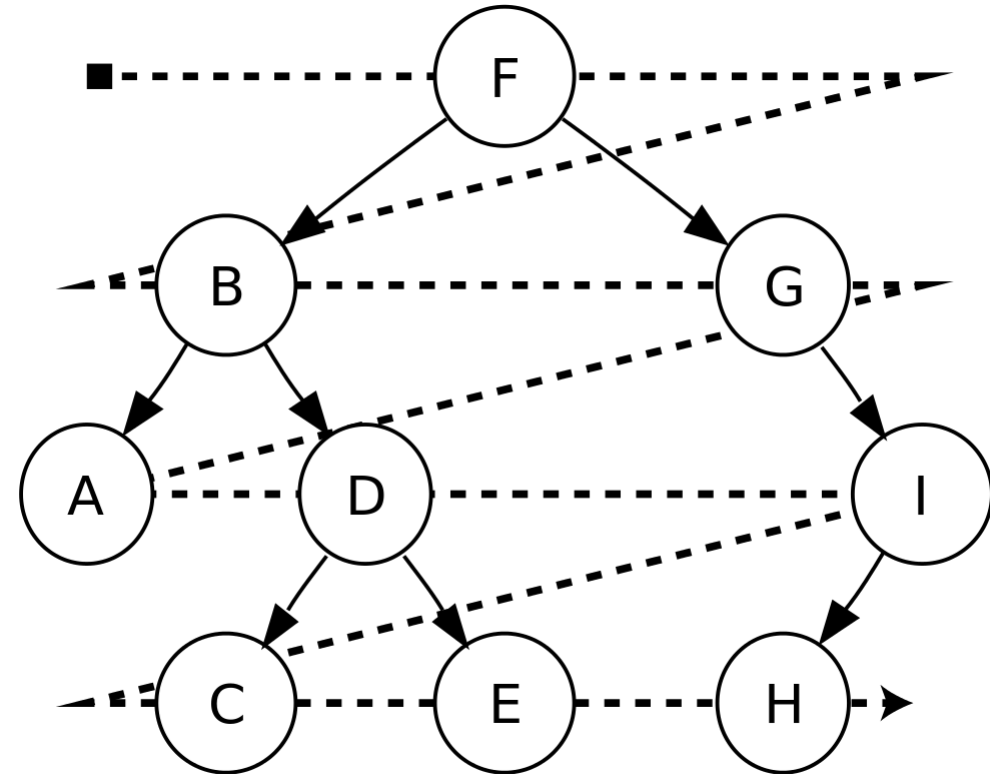


[zhang-xiao-mu.blog]

Versões 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**

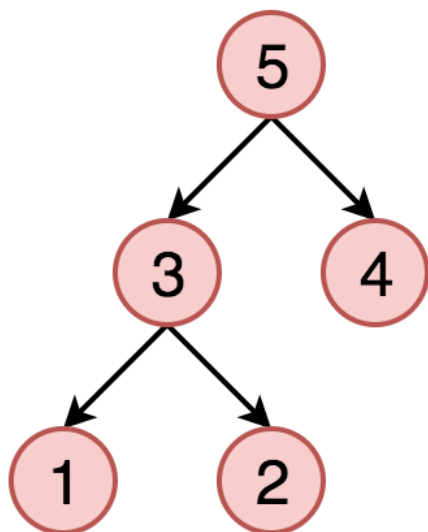


[Wikipedia]

Ordem / Travessias

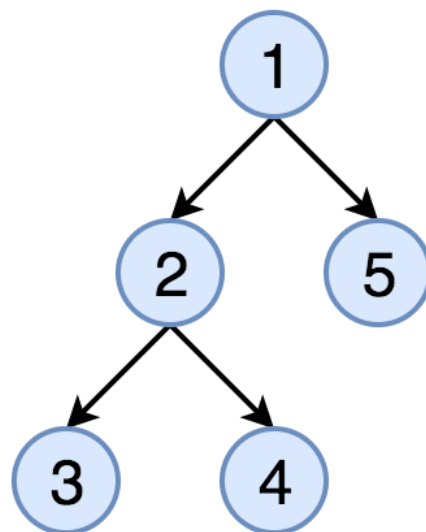
DFS Postorder

Bottom -> Top
Left -> Right



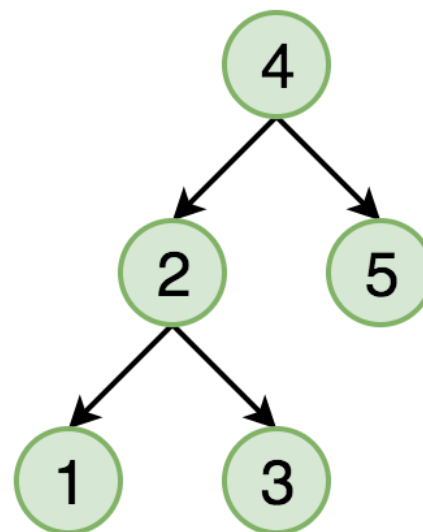
DFS Preorder

Top -> Bottom
Left -> Right



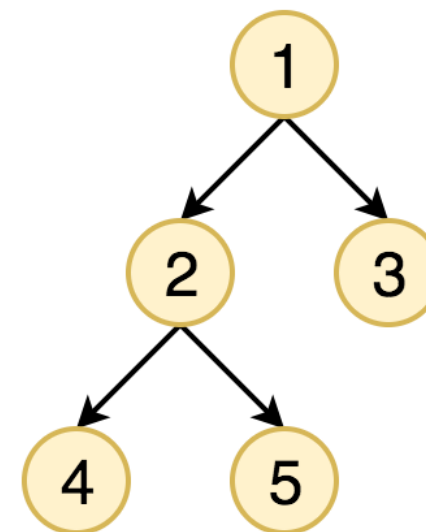
DFS Inorder

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

TAD QUEUE

```
#ifndef _POINTERS_QUEUE_
#define _POINTERS_QUEUE_

typedef struct _PointersQueue Queue;

Queue* QueueCreate(int size);

void QueueDestroy(Queue** p);

void QueueClear(Queue* q);

int QueueSize(const Queue* q);

int QueueIsFull(const Queue* q);

int QueueIsEmpty(const Queue* q);

void* QueuePeek(const Queue* q);

void QueueEnqueue(Queue* q, void* p);

void* QueueDequeue(Queue* q);

#endif // _POINTERS_QUEUE_
```

TAD STACK

```
#ifndef _POINTERS_STACK_
#define _POINTERS_STACK_

typedef struct _PointersStack Stack;

Stack* StackCreate(int size);

void StackDestroy(Stack** p);

void StackClear(Stack* s);

int StackSize(const Stack* s);

int StackIsFull(const Stack* s);

int StackIsEmpty(const Stack* s);

void* StackPeek(const Stack* s);

void StackPush(Stack* s, void* p);

void* StackPop(Stack* s);

#endif // _POINTERS_STACK_
```

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


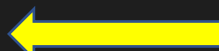
Travessias por níveis – QUEUE

```
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

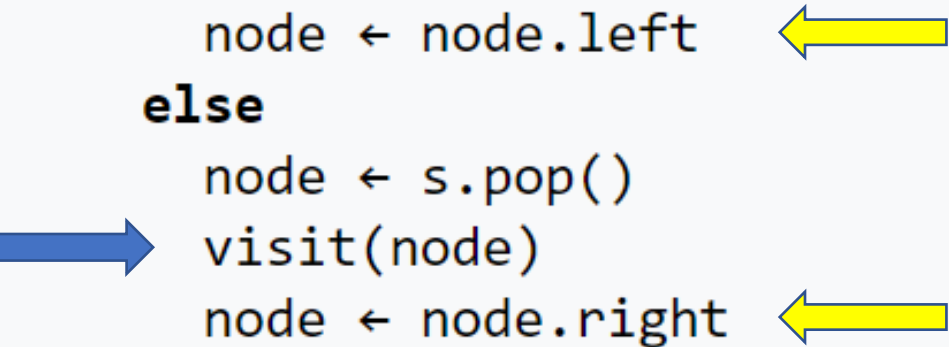
```
while (QueueIsEmpty(queue) == 0) {  
    Tree* p = QueueDequeue(queue);  
  
    function(&(p->item));  
  
    if (p->left != NULL) {  
        QueueEnqueue(queue, p->left);  
    }  
    if (p->right != NULL) {  
        QueueEnqueue(queue, p->right);  
    }  
}  
  
QueueDestroy(&queue);  
}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

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);  
    }  
}
```

Travessia em-ordem

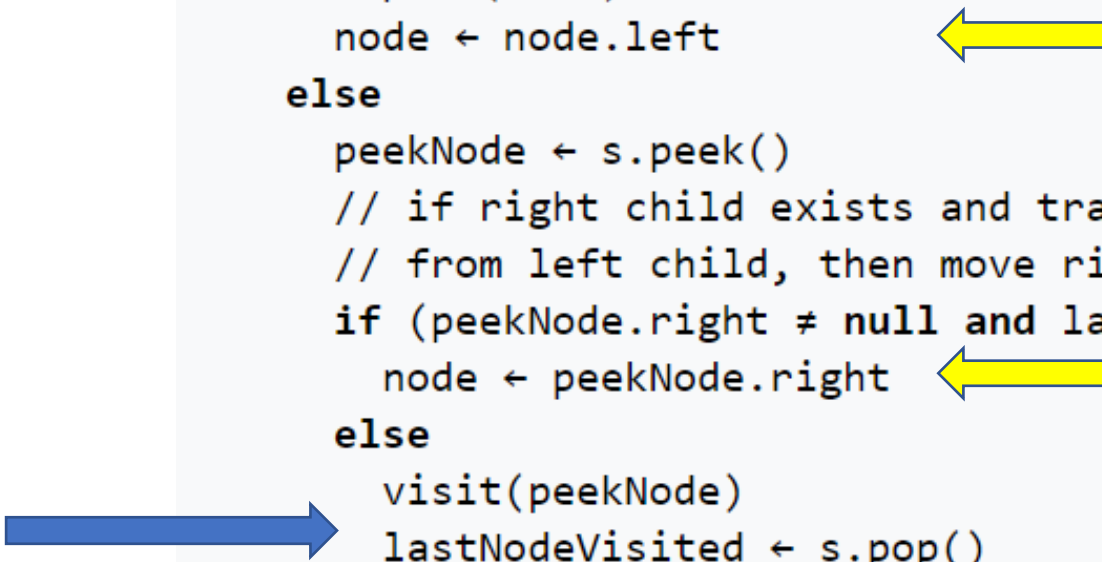
```
iterativeInorder(node)
  s ← empty stack
  while (not s.isEmpty() or node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left ←
    else
      node ← s.pop()
      visit(node)
      node ← node.right ←
```



[Wikipedia]

Travessia em pós-ordem


```
iterativePostorder(node)
  s ← empty stack
  lastNodeVisited ← null
  while (not s.isEmpty() or node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left
    else
      peekNode ← s.peek()
      // if right child exists and traversing node
      // from left child, then move right
      if (peekNode.right ≠ null and lastNodeVisited ≠ peekNode.right)
        node ← peekNode.right
      else
        visit(peekNode)
        lastNodeVisited ← s.pop()
        node ← null
```



[Wikipedia]


Escrita e Reconstrução de Ficheiro

Exemplo de aplicação – Escrita em ficheiro




```
int TreeStoreInFile(const Tree* root, char* fileName, int fileType) {  
    FILE* f = fopen(fileName, "w");  
    if (f == NULL) {  
        return 0;  
    }  
  
    _storeInFile(root, f, fileType);  
  
    fclose(f);  
    return 1;  
}
```

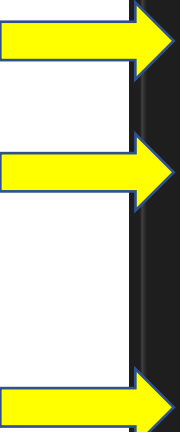
Exemplo de aplicação – Escrita em ficheiro



```
static void _storeInFile(const Tree* root, FILE* f, int fileType) {  
    if (root == NULL) {  
        return;  
    }  
  
    struct _fileNode r;  
    r.item = root->item;  
    r.emptyLeftSubTree = (root->left == NULL);  
    r.emptyRightSubTree = (root->right == NULL);
```



Travessia em pré-ordem



```
if (fileType == 1) {  
    fprintf(f, "%d %d %d ", r.item, r.emptyLeftSubTree, r.emptyRightSubTree);  
} else {  
    fwrite(&r, sizeof(struct _fileNode), 1, f);  
}  
  
_storeInFile(root->left, f, fileType);  
_storeInFile(root->right, f, fileType);  
}
```

Exemplo

```
printf("STORING in a file\n");  
TreeStoreInFile(tree, "arvore1.txt", 1);
```

```
2 0 0 4 0 0 8 0 1 16 1 1 10 1 1 6 0 0 12 1 1 14 1 1
```

- Qual é a árvore ?

Exemplo de aplicação – Reconstrução




```
Tree* TreeGetFromFile(char* fileName, int fileType) {  
    FILE* f = fopen(fileName, "r");  
    if (f == NULL) {  
        return NULL;  
    }  
  
    Tree* root;  
    _getFromFile(&root, f, fileType);  
  
    fclose(f);  
    return root;  
}
```




Exemplo de aplicação – Reconstrução

```
static void _getFromFile(Tree** pRoot, FILE* f, int fileType) {  
    struct _fileNode r;  
  
    if (fileType == 1) {  
        fscanf(f, "%d", &r.item);  
        fscanf(f, "%d", &r.emptyLeftSubTree);  
        fscanf(f, "%d", &r.emptyRightSubTree);  
    } else {  
        fread(&r, sizeof(struct _fileNode), 1, f);  
    }  
}
```



Reconstrução em pré-ordem




```
Tree* newNode = (Tree*)malloc(sizeof(struct _TreeNode));

newNode->item = r.item;

if (r.emptyLeftSubTree) {
    newNode->left = NULL;
} else {
    _getFromFile(&(newNode->left), f, fileType);
}
```


Reconstrução em pré-ordem



```
if (r.emptyRightSubTree) {  
    newNode->right = NULL;  
} else {  
    _getFromFile(&(newNode->right), f, fileType);  
}  
  
*pRoot = newNode;  
}
```

Que funcionalidades faltam ?

Em falta ?

- **Adicionar** um elemento à árvore
- **Apagar** um elemento da árvore
 - Pode não ser fácil...
- ...

[Wikipedia]

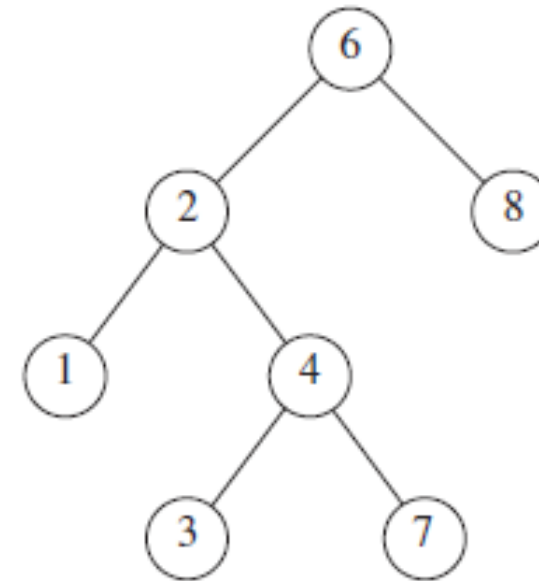
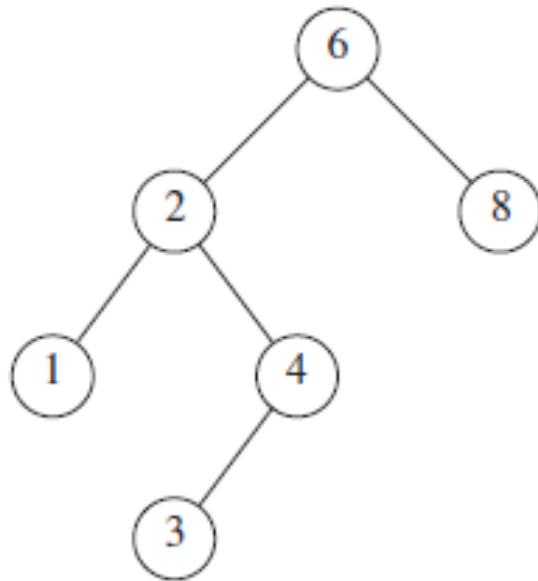
IDEIA – Adicionar – Faz sentido? – Testem...

- Se a árvore é **vazia**
o novo nó será a (nova) raiz
- Se a raiz **não** tem **filhos**
o novo nó será o seu filho esquerdo
- Se a raiz tem **um filho**
o novo nó será o outro filho
- Se a raiz tem **dois filhos**
o novo nó será a (nova) raiz
a antiga raiz será o seu filho esquerdo

Árvores Binárias de Procura

- próxima semana !

Critério de **ordem** ?



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