经常换位思考，挖掘人生精彩

第11讲 二叉树的遍历

信息学院（智能应用研究院）
欧 新 宇

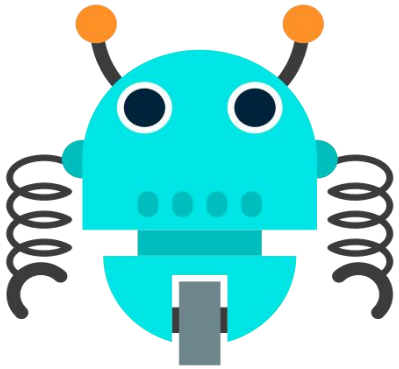
第5章 树和二叉树

考核要点

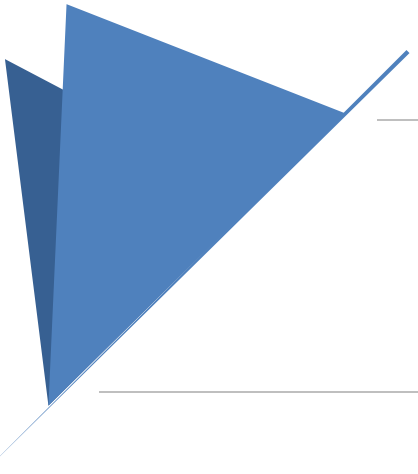
- **考核大纲**

树的基本概念及存储结构；二叉树的定义、主要特征及其存储结构，二叉树的遍历，线索二叉树的基本概念和构造；森林与二叉树的的转换；树和森林的遍历；哈夫曼树和哈夫曼编码。

- **复习要点**
  - 熟练掌握二叉树的前、中、后序遍历方法及算法
  - 掌握森林和二叉树的转换方法



- 二叉树的遍历
- 二叉树遍历的相关性质
- 二叉树遍历的应用
- 线索二叉树



二叉树的遍历

/ 先序遍历

/ 中序遍历

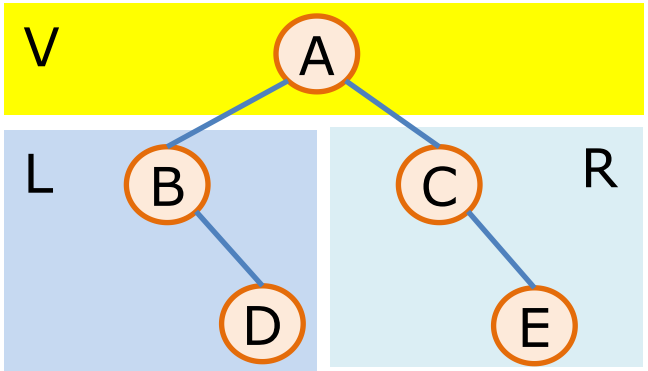
/ 后序遍历

/ 层次遍历

二叉树的遍历

什么是遍历?

遍历（traverse）是指对于一个有限元素的集合，按照某条线索路线对每个元素执行一次且只一次的访问操作。二叉树遍历是指对二叉树的每个结点，按照某条线索路线访问且只访问一次的操作。遍历是二叉树一切运算的核心。



- ✓ 根结点 → V：访问根节点
- ✓ 左子树 → L：遍历左子树
- ✓ 右子树 → R：遍历右子树

二叉树的遍历

二叉树遍历的类别

递归遍历

- ✓先序遍历(VLR)：根，左子树、右子树
- ✓中序遍历(LVR)：左子树，根，右子树
- ✓后序遍历(LRV)：左子树，右子树，根

左子树永远**先于**右子树访问

非递归遍历

- ✓先序遍历(VLR)：根，左子树、右子树
- ✓中序遍历(LVR)：左子树，根，右子树
- ✓后序遍历(LRV)：左子树，右子树，根
- ✓层次遍历

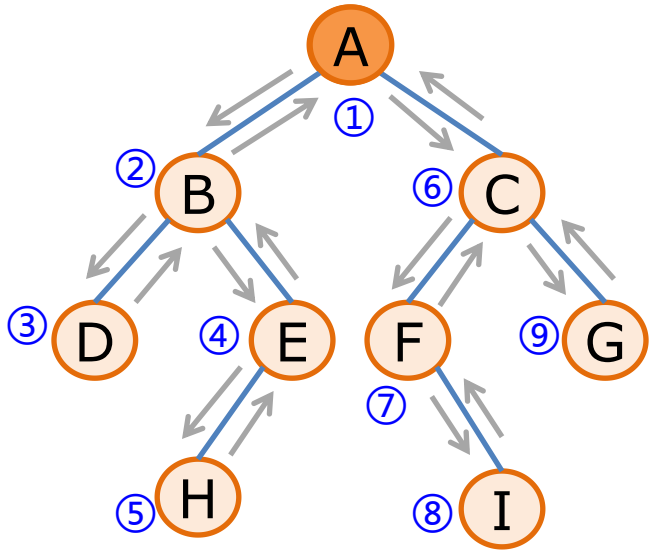
二叉树的递归遍历

先序遍历（PreOrder）

基本规则：

- 若二叉树为空，则返回；
- 否则，执行如下步骤：
  - 访问根结点；
  - 先序遍历左子树；
  - 先序遍历右子树。

递归




```
void PreOrder (BinaryTree BT) {
    if (BT != Null) {
        printf( "%d" , BT -> Data);  // 访问根节点
        PreOrder(BT -> lChild);      // 递归遍历左子树
        PreOrder(BT -> rChild);      // 递归遍历右子树
    }
}
```

先序遍历 => A (B D E H) (C F I G)

特性：最先访问的是二叉树的根以及每个子树的根

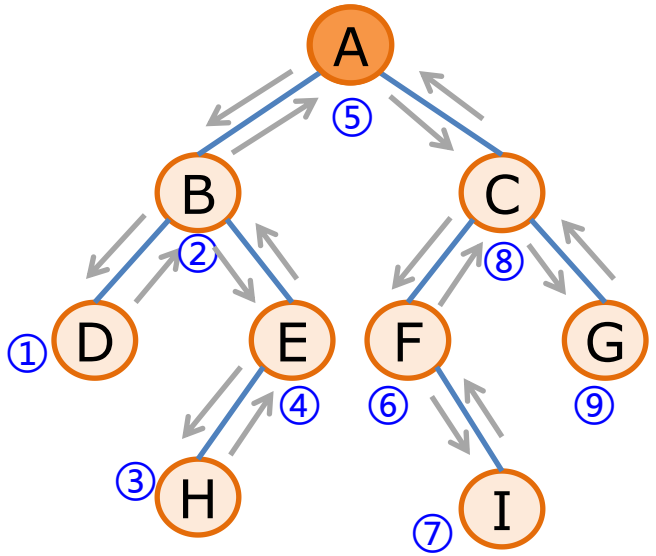
二叉树的递归遍历

中序遍历（InOrder）

基本规则：

- 若二叉树为空，则返回；
- 否则，执行如下步骤：
  - 中序遍历左子树；
  - 访问根结点；
  - 中序遍历右子树。

递归




```
void InOrder (BinaryTree BT) {
    if (BT != Null) {
        InOrder(BT -> lChild);          // 递归遍历左子树
        printf( "%d" , BT -> Data);     // 访问根节点
        InOrder(BT -> rChild);          // 递归遍历右子树
    }
}
```

中序遍历 => (D B H E) A (F I C G)

特性：左子树所有结点一定先于根被访问，右子树结点一定在根之后访问

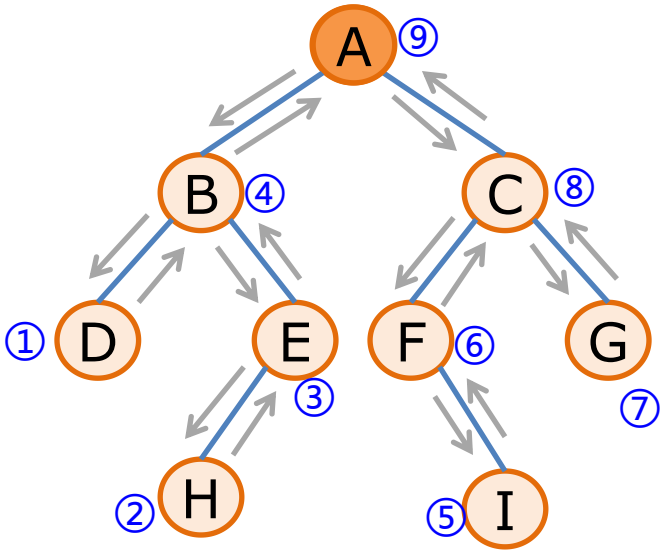
二叉树的递归遍历

后序遍历（PostOrder）

基本规则：

- 若二叉树为空，则返回；
- 否则，执行如下步骤：
  - 后序遍历左子树；
  - 后序遍历右子树。
  - 访问根结点；

递归




```
void PostOrder (BinaryTree BT) {
    if (BT != Null) {
        PostOrder(BT -> lChild);      // 递归遍历左子树
        PostOrder(BT -> rChild);      // 递归遍历右子树
        printf( "%d" , BT -> Data); // 访问根节点
    }
}
```

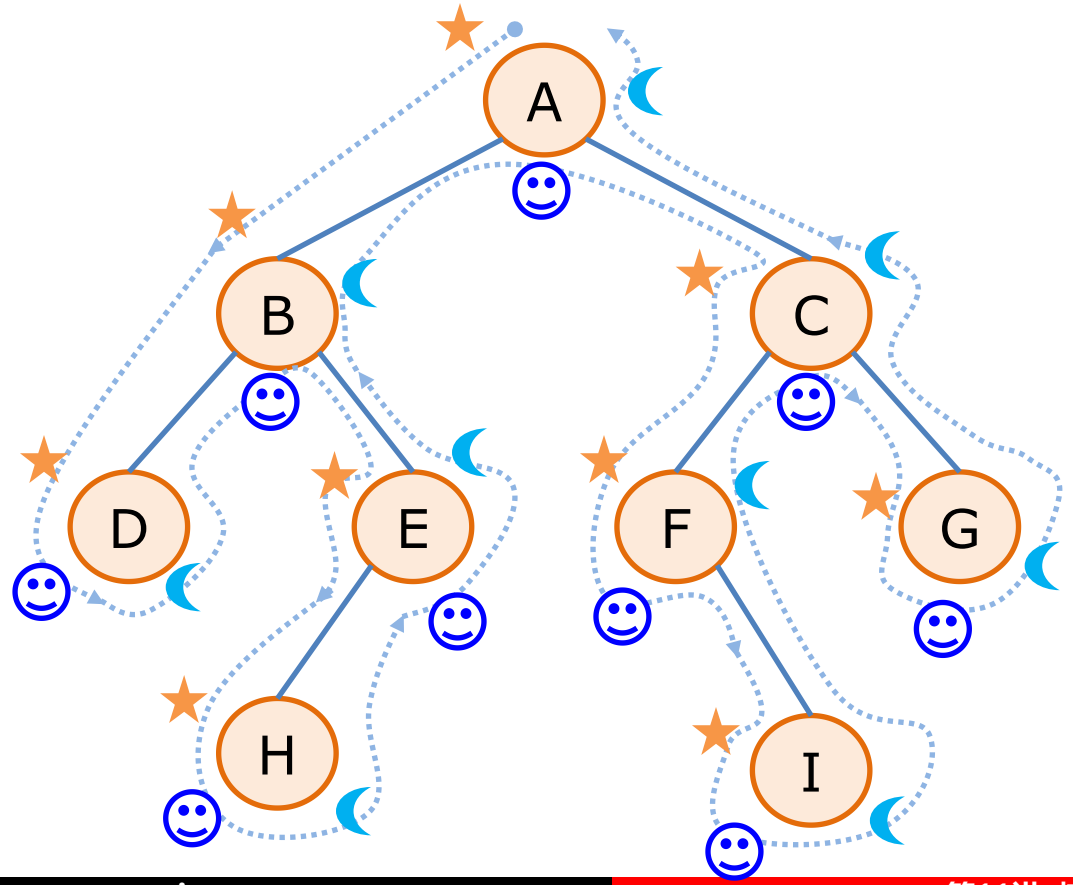
后序遍历 => (D H E B) (I F G C) A

特性：无论是整棵二叉树还是子树，最后访问输出的一定是根节点

二叉树的递归遍历

三种递归遍历算法分析

从递归的角度看，先序、中序和后序遍历的**访问路径**都是相同的，只是**访问结点的时机**不同。



从虚线的出发点到终点的路径上，每个结点经过3次，但每个结点访问且访问一次。

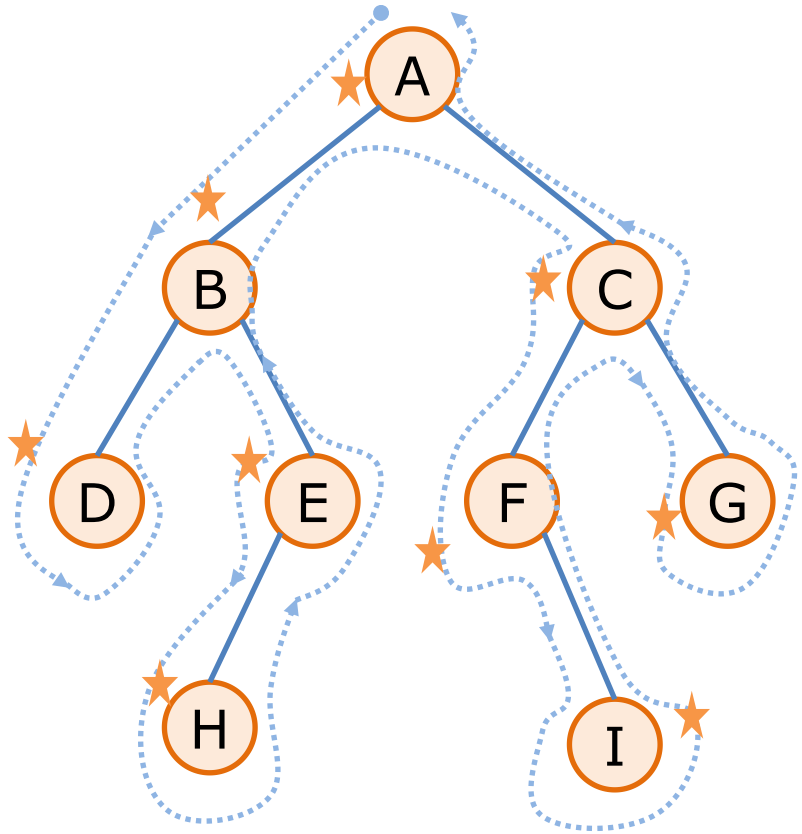
第1次经过时访问 = 先序遍历

第2次经过时访问 = 中序遍历

第3次经过时访问 = 后序遍历

时间复杂度：O(n)

二叉树的非递归遍历



先序遍历 => A (B D E H) (C F I G)

非递归先序遍历

由于二叉树的先序遍历是沿着一条固定的线路进行。因此，可以借助于**堆栈**，沿着这条线路，在第1次遇到一个结点时，就执行入栈操作及对当前结点执行输出操作。具体按以下规则依次进行遍历：

1. 将根压入堆栈，并立即执行输出；
2. 从根开始访问左孩子，并将左孩子当作新子树的根，进行入栈并执行输出，直到左孩子为空，退回到根结点；
3. 当退回到当前子树的根时，访问右孩子。若右孩子为空，则退回到当前子树的根，并将根结点出栈；
4. 继续执行**步骤3**的回退操作；若右孩子不空，则将右孩子当作新子树的根执行**步骤2**。

二叉树的非递归遍历

非递归先序遍历

使用非递归方法对二叉树BT进行先序遍历

```
void PreOrder (BinaryTree BT) {
    InitStack(S);
    BinTree *p = BT;
    while (p || !IsEmpty(S)){
        if (p) {
            Push(S, p);
            printf( “%c” , p->data);
            p = p->lchild;
        }
    else {
        p = p->rChild;
        Pop(S, p);
    }
}
```

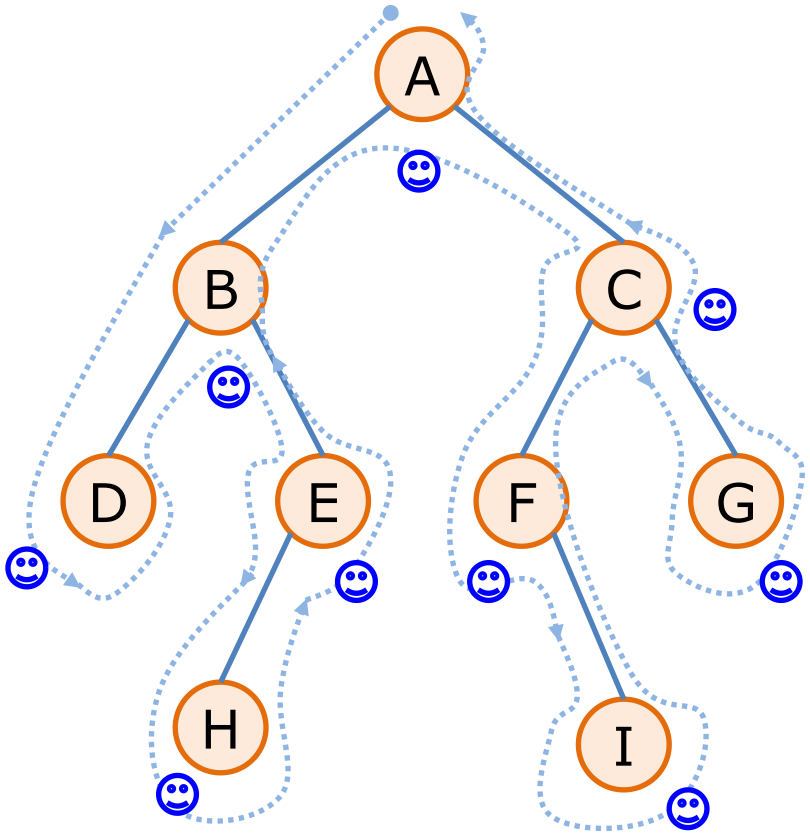
算法步骤：

1. 初始化堆栈S
2. 初始化工作指针p，指向二叉树BT
3. 若树不空，则一路向左

    3.1 将当前结点压入堆栈
    3.2 访问并输出当前结点（第1次遇到）
    3.3 左孩子不空，则一直向左走

4. 左孩子为空
    4.1 右孩子不空，则转向访问右孩子
    4.2 右孩子访问结束后，将当前子树根结点进行出栈

二叉树的非递归遍历



中序遍历 => (D B H E) A (F I C G)

非递归中序遍历

由于二叉树的中序遍历是沿着一条固定的线路进行。因此，可以借助于**堆栈**，沿着这条线路，在第1次遇到一个结点时，就执行入栈操作及对当前结点执行~~输出操作~~。具体按以下规则依次进行遍历：

1. 将根压入堆栈，~~并立即执行输出~~；
2. 从根开始访问左孩子，并将左孩子当作新子树的根，进行入栈，直到左孩子为空，退回到根结点，并输出根结点；
3. 当退回到当前子树的根时，访问右孩子。若右孩子为空，则将根结点出栈；
4. 继续执行**步骤3**，若右孩子不空，则将右孩子当作新子树的根执行**步骤2**。

二叉树的非递归遍历

非递归中序遍历

使用非递归方法对二叉树BT进行先序遍历

```
void InOrder (BinaryTree BT) {
    InitStack(S);
    BinTree *p = BT;
    while(p || !IsEmpty(S)){
        if (p) {
            Push(S, p);
            p = p->lChild;
        }
        else {
            printf( “%c” , p->data);
            p = p->rChild;
            Pop(S, p);
        }
}
```

算法步骤：

1. 初始化堆栈S
2. 初始化工作指针p，指向二叉树BT
3. 若树不空，则一路向左

    3.1 将当前结点压入堆栈

    3.2 左孩子不空，则一直向左走

4. 左孩子为空

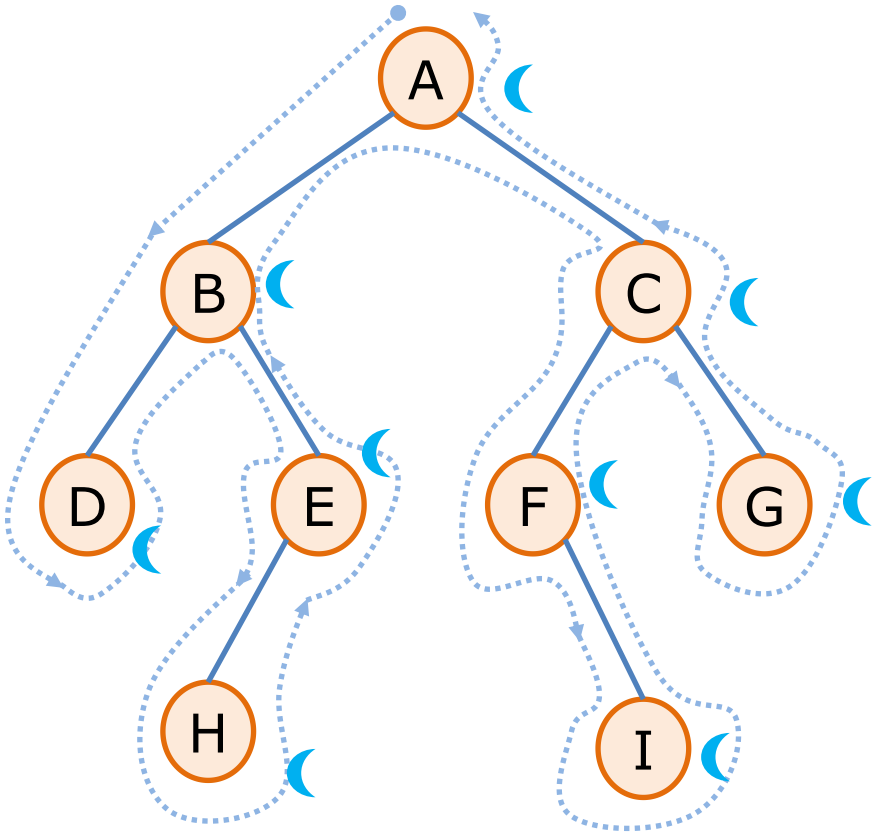
    4.1 输出当前子树的根结点元素

    4.2 右孩子访问结束后，将当前子树根结点进行出栈

    4.3 右孩子不空，则转向访问右孩子

注意，在实际应用中，为了简化代码，可以先出栈栈顶元素，再访问右子树

二叉树的非递归遍历



后序遍历 => (D H E B) (I F G C) A

非递归后序遍历

二叉树的后序遍历所使用的线路与先序、中序是一样。但应注意的是，如果要将某个结点出栈，必须保证该结点的左右孩子已经出栈。具体操作如下：

1. 将根压入堆栈，~~并立即执行输出~~；
2. 从根开始访问左孩子，并将左孩子当作新子树的根，进行入栈，直到左孩子为空，退回到根结点，~~并输出根结点~~；
3. 当退回到当前子树的根时，访问右孩子。若右孩子为空，则退回到当前子树的根，并将根结点出栈；
4. 继续执行**步骤3**，若右孩子不空，则将右孩子当作新子树的根执行**步骤2~3**。

二叉树的非递归遍历

使用非递归方法对二叉树BT进行后序遍历

```
void PostOrder (BinaryTree BT) {
    InitStack(S);
    BinaryTree *p = BT, BTNode *r =Null;
    while(p || !IsEmpty(S)){
        if (p) {
            Push(S, p);
            p = p->lChild;
        }
        else {
            GoTop(S, p);
            if (p->rchild&p->rchild!=r)
                p = p->rchild;
            else{
                printf( “%c”, p->data);
                r = p;
                Pop(S, p);
                p = Null;
            }
        }
}
```

非递归后序遍历

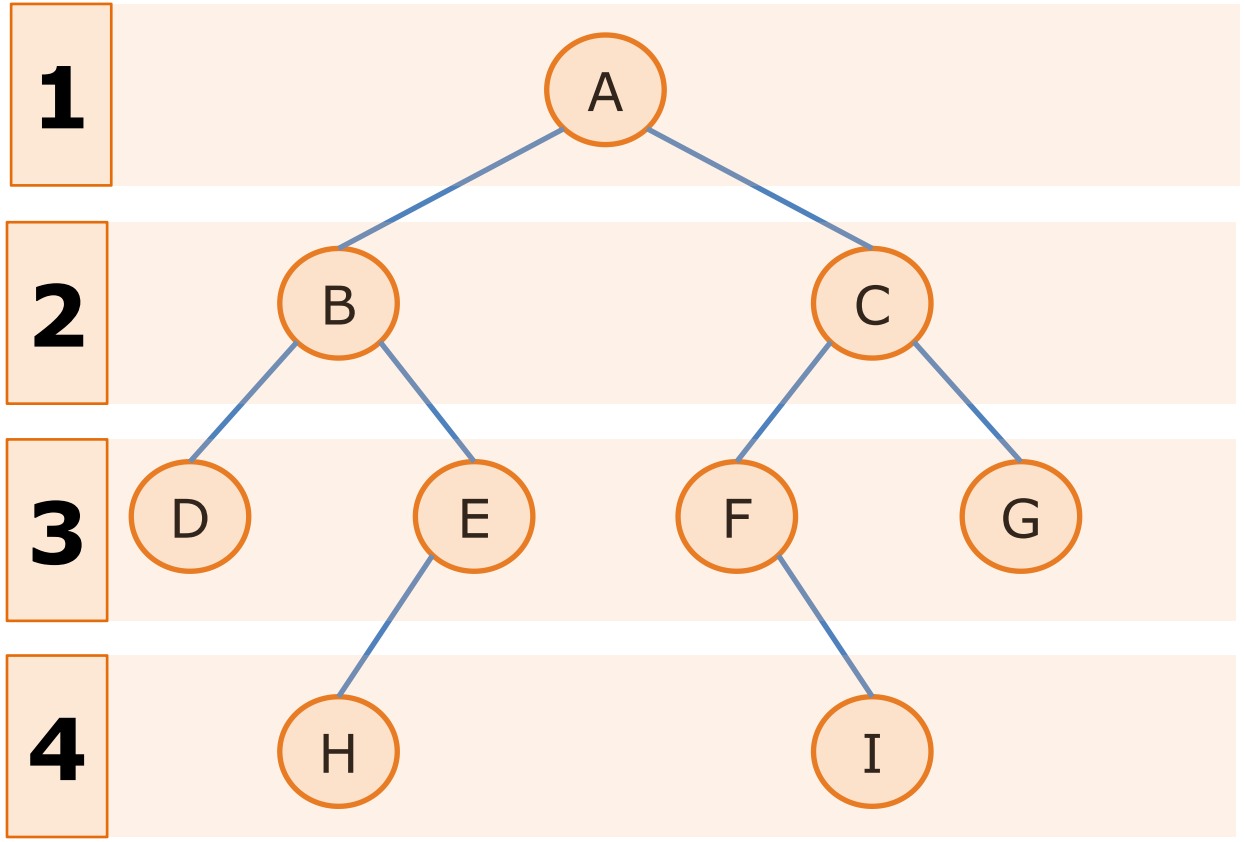
1. 初始化堆栈S
2. 初始化工作指针p，以及指示指针r，用于指向最近访问过的结点，区分是从左子树返回，还是右子树返回。
3. 后序遍历二叉树，若树不空，则一路向左
   3.1 将当前结点压入堆栈
   3.2 左孩子不空，则一直向左走
4. 左孩子为空
   4.1 读取栈顶子树根结点（暂不出栈，也不输出）
   4.2 若右子树存在，且未被访问，则转向访问右孩子
   4.3 否则，输出当前子树的根结点元素
   4.4 同时，将指针r指向最近访问过的元素
   4.5 将当前栈顶根结点元素进行出栈
   4.3 结点访问过后，重置指针p

注意：指针 r 标识的访问过的元素是指以该结点为根的子树已经全部遍历完成（包括左右子树）。

二叉树的非递归遍历

层次遍历（LevelOrder）

二叉树的**层次遍历**是指遍历过程按照从上到下、从左至右的顺序对每个结点执行访问且只一次的访问，又称为**广度优先遍历**。



层次遍历的输出顺序：

A B C D E F G H I

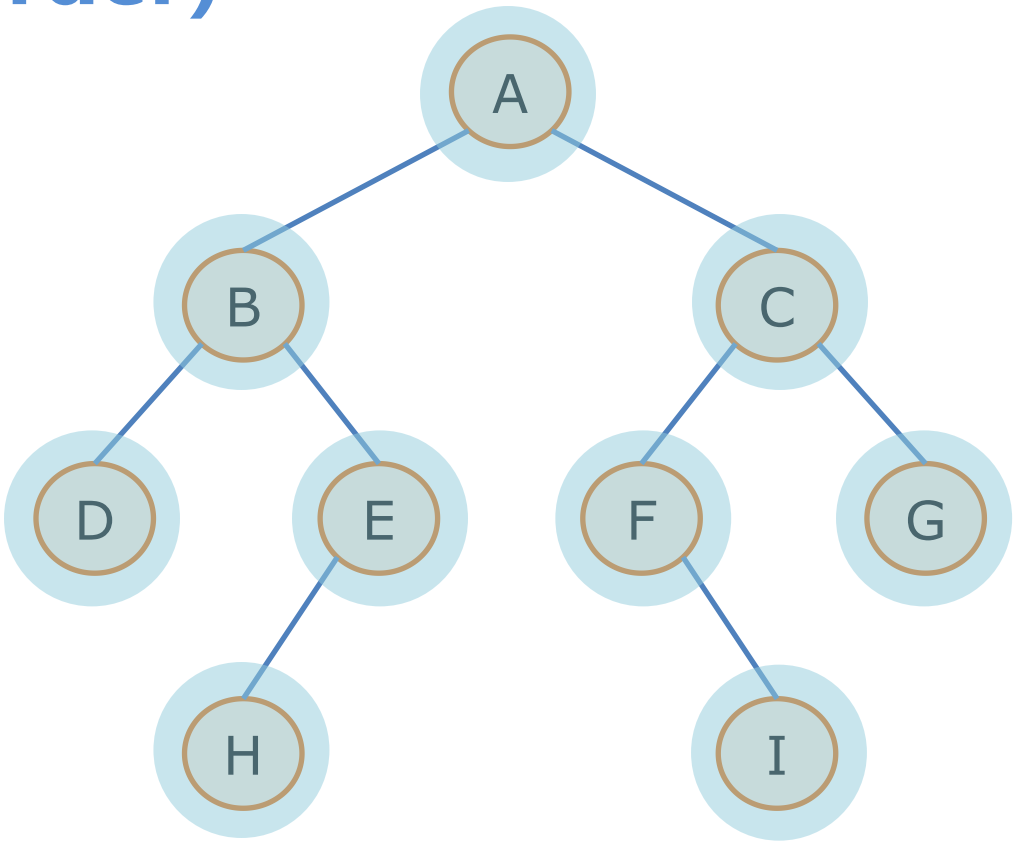
二叉树的非递归遍历

层次遍历（LevelOrder）

基本规则：

1. 若二叉树为空，则返回；否则初始化队列，并将根结点入队；
2. 判断队列是否为空，若不为空，则执行以下操作：
   - ✓ 获取队头结点p，并将队头结点出队；
   - ✓ 访问结点p中的数据；
   - ✓ 若p的左孩子结点存在，则将该左孩子结点进队；
   - ✓ 若p的右孩子结点存在，则将该右孩子结点进队；
   - ✓ 反复执行步骤2，直至所有结点都完成遍历

- 二叉树层次遍历结束。



队列输出序列：A B C D E F G H I

二叉树的非递归遍历

层次遍历（LevelOrder）

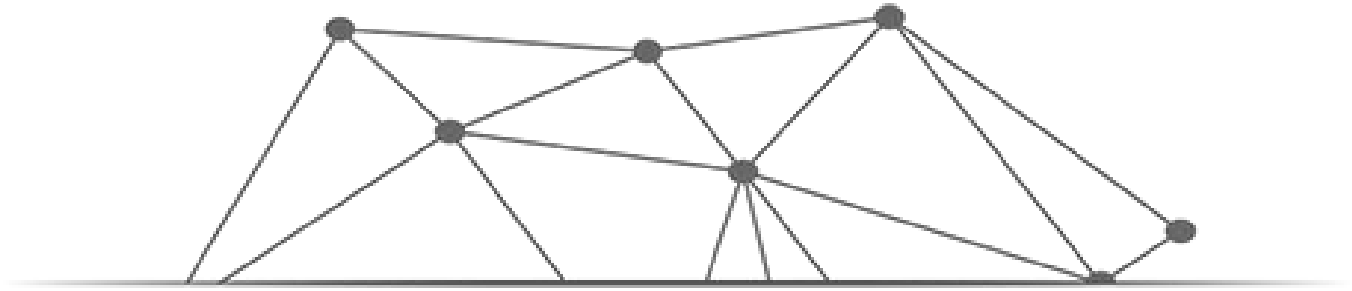
二叉树的层次遍历

```
void LevelOrder (BinaryTree BT) {
    SqQueue Q;  BTNode *p;
    if (!BT->root) return;
    InitQueue (&Q);
    EnQueue (&Q, BT);
    while (QueueEmpty(Q)) {
        DeQueue(&Q, BT->root);
        printf( '%c' , p->data);
        if (p->lChild != Null)   EnQueue (&Q, p->lChild);
        if (p->rChild != Null)   EnQueue (&Q, p->rChild);
    }
    Destroy(&Q);
} LevelOrder
```

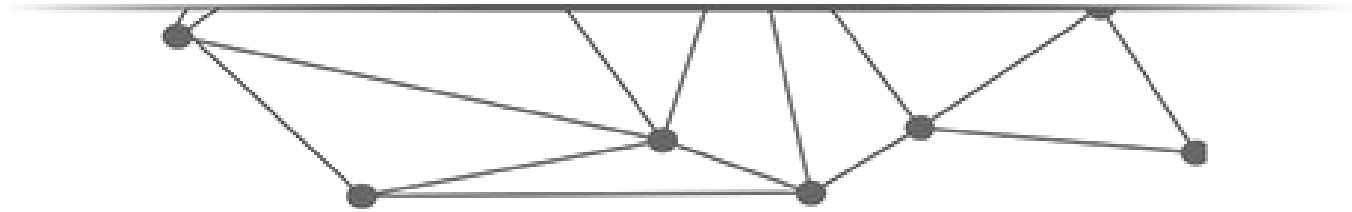
操作步骤：

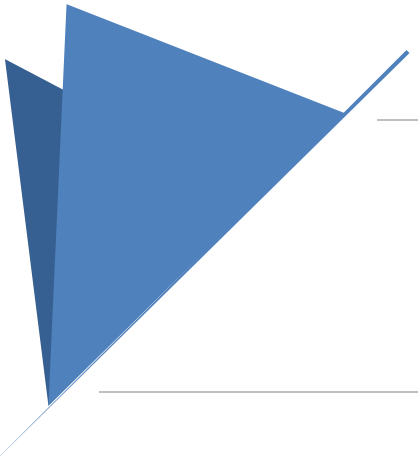
1. 创建一个队列Q，以及指针p指向二叉树
2. 若二叉树为空，则返回；
3. 若二叉树不为空，则初始化队列，
4. 并将根结点入队；
5. 判断队列是否为空，若不为空，则：
   5.1 获取队头结点p，并将队头结点出队；
   5.2 访问结点p中的数据；
   5.3 若p的左孩子存在，则将左孩子进队；
   5.4 若p的右孩子存在，则将右孩子进队；
   // 反复执行循环5，直至遍历完所有结点
6. 销毁用于层次遍历的临时队列。

时间复杂度O(n)



课堂互动 11.1





二叉树遍历的相关性质

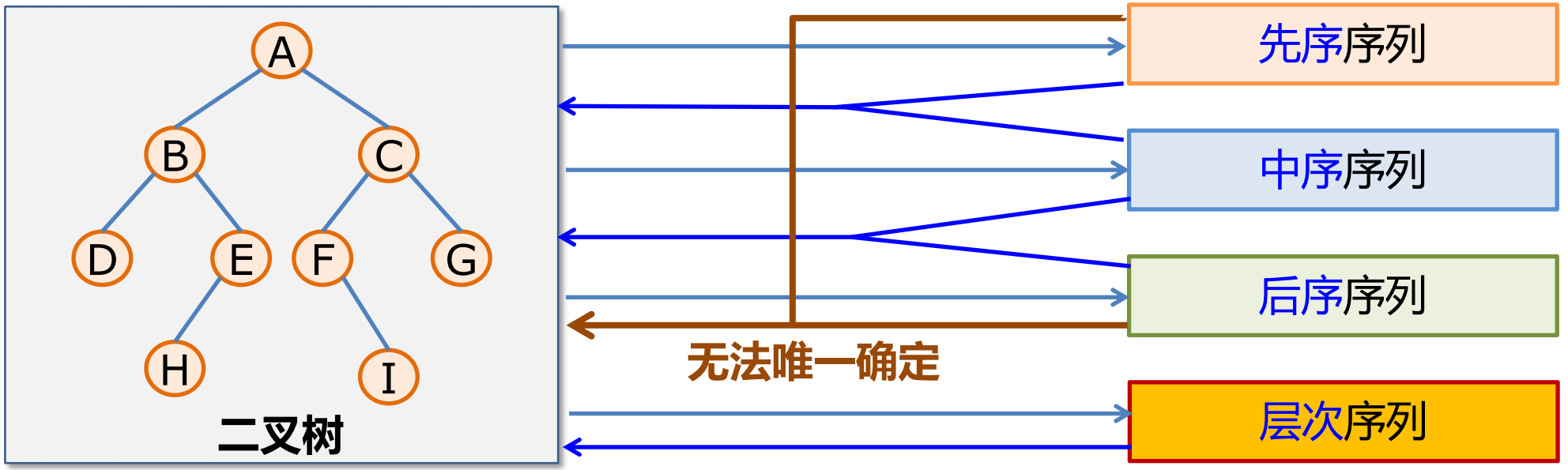
/ 二叉树的重构

选根 -> 分左右

二叉树遍历的相关性质

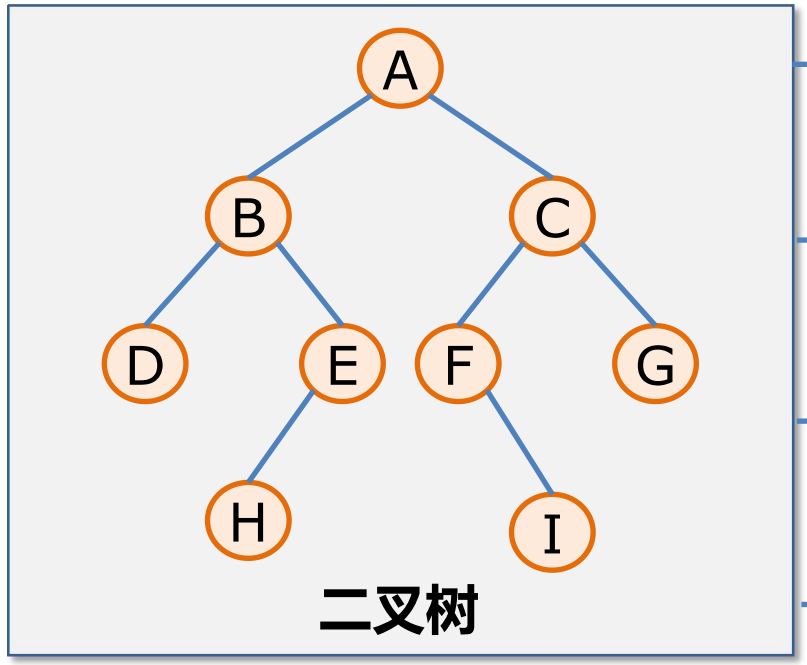
是否可以通过二叉树的遍历来实现二叉树的重构?

- 给定一棵二叉树，可以唯一确定其先序、中序、后序和层次遍历序列；
- 已知二叉树的先序遍历和中序遍历序列，可以唯一确定一棵二叉树；
- 已知二叉树的后序遍历和中序遍历序列，可以唯一确定一棵二叉树；
- 已知二叉树的先序遍历和后序遍历序列，无法唯一确定一棵二叉树；
- 已知二叉树的层次遍历和中序遍历序列，也可以唯一确定一棵二叉树。



二叉树遍历的相关性质

给定一棵二叉树，可以唯一确定其先序、中序、后序和层次遍历序列



先序序列：A (BDEH) (CFIG)

中序序列：(DBHE) A (FICG)

后序序列：(DHEB) (IFGC) A

层次序列：A B C D E F G H I

二叉树遍历的相关性质

已知二叉树的先序遍历和中序遍历序列，可以唯一确定一棵二叉树

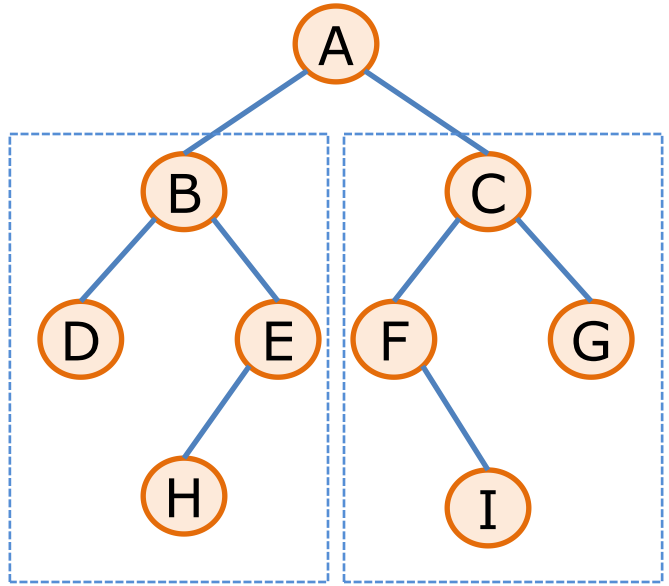
基本原则：

1. 先序遍历序列：**根结点**出现在先序遍历序列的最前面
2. 中序遍历序列：左子树中的结点在**根结点**的左侧，而右子树中的结点在**根结点**的右侧

例1：已知先序序列 A (B D E H) (C F I G) 和中序序列 (D B H E) A (F I C G)，试确定二叉树。

基本思路：

1. 根据1，确定根结点A
2. 根据2，确定DBHE为A的左子树，FICG为A的右子树
3. 根据1，确定B为A的左子树的根，C为A的右子树的根
4. 根据2，确定D为B的左子树且是根，HE为B的右子树；
   确定FI为C的左子树，G为C的右子树且是根
5. 根据1，确定E为B的右子树的根；F为C的左子树的根
6. 根据2，确定H为E的左子树；确定I为F的右子树



二叉树遍历的相关性质

已知二叉树的后序遍历和中序遍历序列，可以唯一确定一棵二叉树

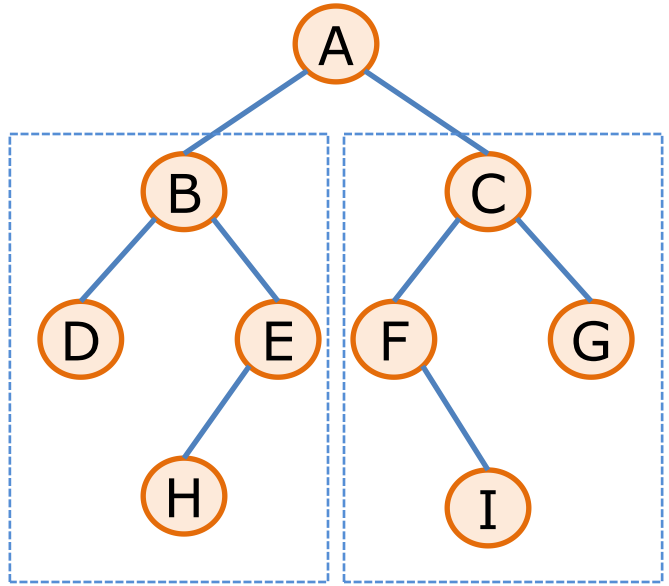
基本原则：

1. 后序遍历序列：**根结点**出现在后序遍历序列的最后面
2. 中序遍历序列：左子树中的结点在**根结点**的左侧，而右子树中的结点在**根结点**的右侧

例2：已知后序序列 (D H E B) (I F G C) A 和中序序列 (D B H E) A (F I C G)，试确定二叉树。

基本思路：

1. 根据1，确定根结点A
2. 根据2，确定DBHE为A的左子树，FICG为A的右子树
3. 根据1，确定B为A的左子树的根，C为A的右子树的根
4. 根据2，确定D为B的左子树且是根，HE为B的右子树；
   确定FI为C的左子树，G为C的右子树且是根
5. 根据1，确定E为B的右子树的根；F为C的左子树的根
6. 根据2，确定H为E的左子树；确定I为F的右子树



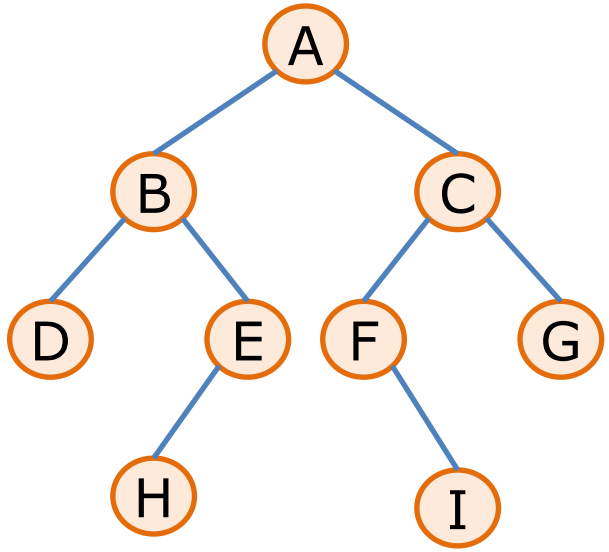
二叉树遍历的相关性质

已知二叉树的先序遍历和后序遍历序列，无法唯一确定一棵二叉树

- 根据先序遍历序列，可知**根结点**出现在先序遍历序列的最前面；
- 根据后序遍历序列，可知**根结点**出现在后序遍历序列的最后面；

因此，先序遍历和后续遍历都只能实现根节点和其他结点的区分，无法实现左右子树的区分。所以，它们虽然可以重构二叉树，但不一定是唯一的。

先序序列 A B D E H C F I G
后序序列 D H E B I F G C A



二叉树遍历的相关性质

已知二叉树的层次遍历和中序遍历序列，可以唯一确定一棵二叉树

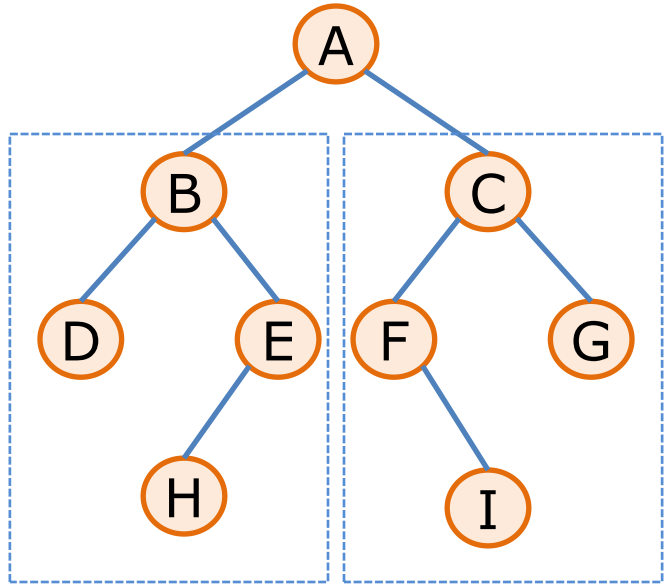
基本原则：

1. 层次遍历序列：最近遍历的结点，必为**根结点**或左、右子树的**根结点**
2. 中序遍历序列：左子树中的结点在**根结点**的左侧，而右子树中的结点在**根结点**的右侧

例3：已知层次序列 A B C D E F G H I 和中序序列 (D B H E) A (F I C G)，试确定二叉树。

基本思路：

1. 根据1，确定根结点A
2. 根据2，确定DBHE为A的左子树，FICG为A的右子树
3. 根据1，确定B为A的左子树的根，C为A的右子树的根
4. 根据2，确定D为B的左子树且是根，HE为B的右子树；
   确定FI为C的左子树，G为C的右子树且是根
5. 根据1，确定E为B的右子树的根；F为C的左子树的根
6. 根据2，确定H为E的左子树；确定I为F的右子树

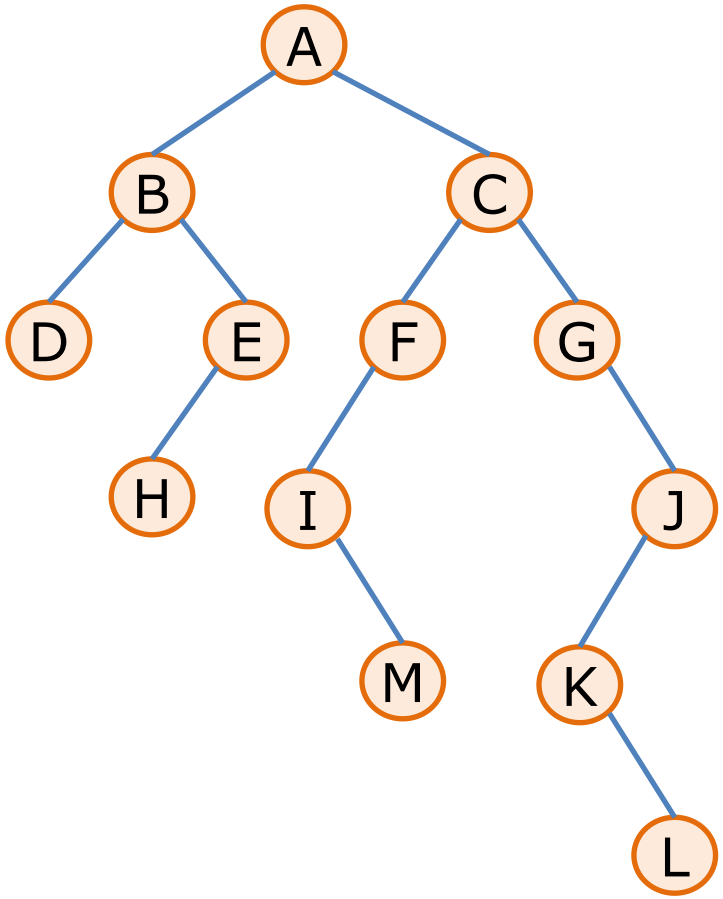


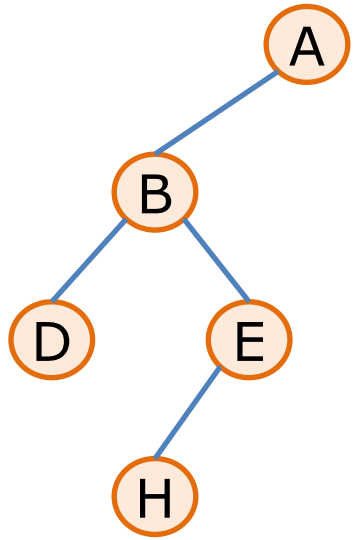
二叉树遍历的相关性质

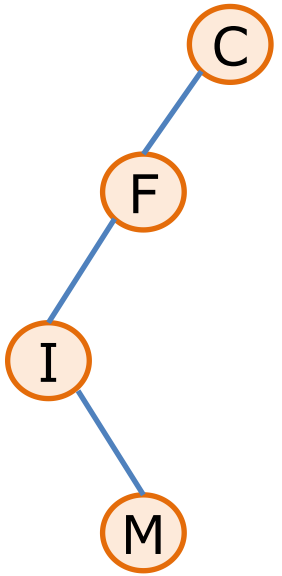
利用二叉树遍历的相关性质重构二叉树的基本方法

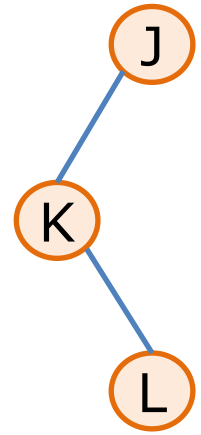
1. 利用先序遍历、后序遍历或层次遍历，先确定当前树/子树的根结点；
2. 根据中序遍历的特点，将前根结点左边的所有结点确定为当前根结点的左子树，当前根结点右边的所有结点确定为当前结点的右子树；
3. 分别针对**步骤2**中确定的左子树和右子树，使用**步骤1**中的方法确定子树的根结点；
4. 再次使用**步骤2**实现左右子树的划分；
5. ...循环执行**步骤1**和**步骤2**，直至整棵树遍历完成。

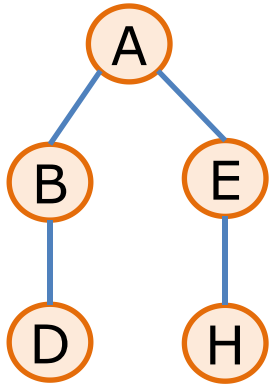
例4：已知先序序列ABDEHCFIMGJKL和中序序列DBHEAIMFCGKLJ，试确定该二叉树对应的森林。

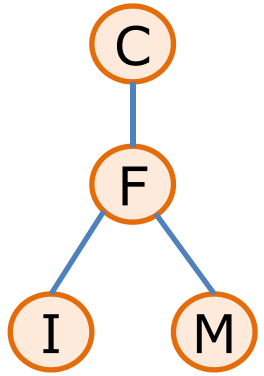


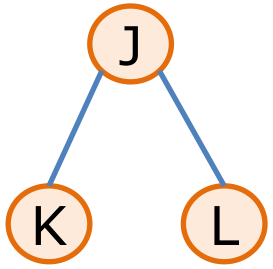


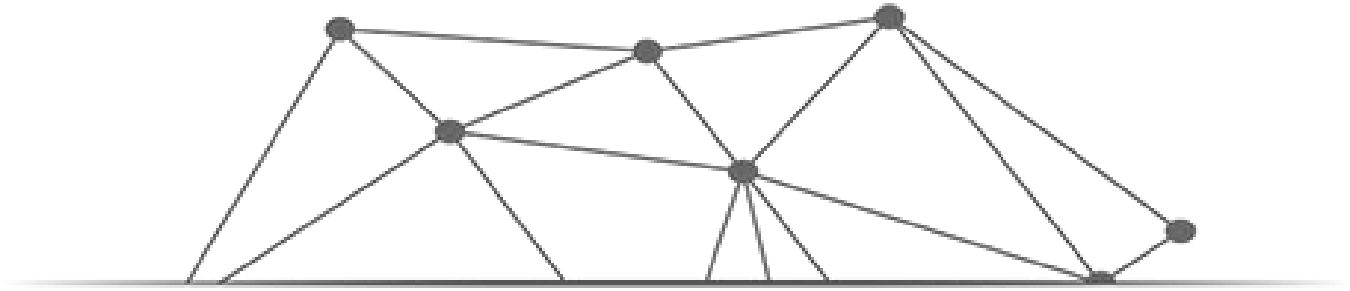




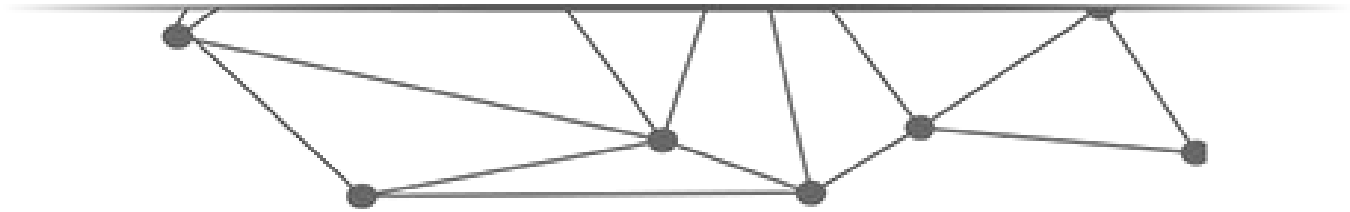


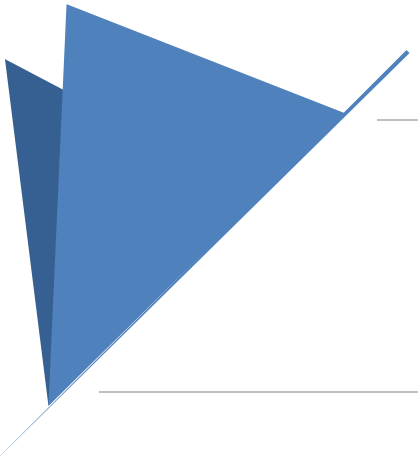






课堂互动 11.2

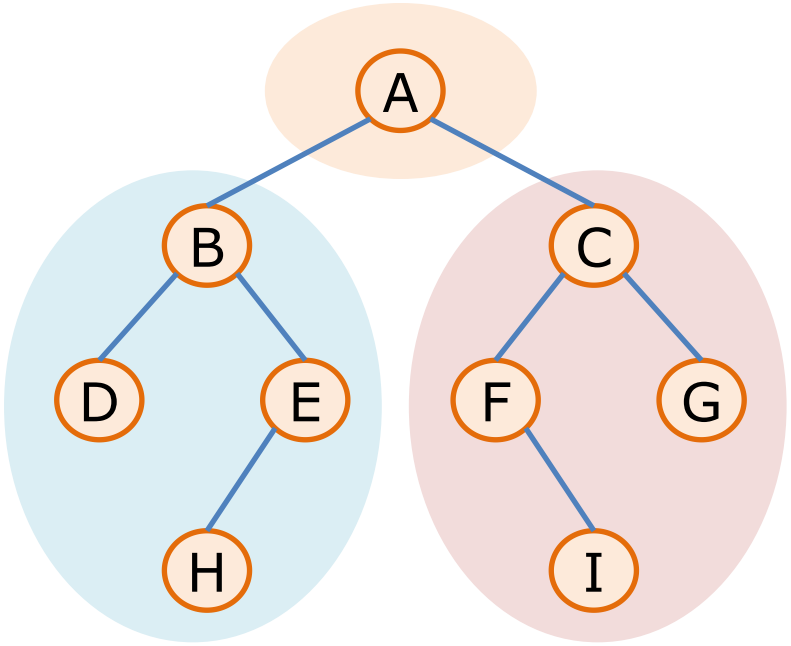




二叉树遍历的应用

二叉树遍历的应用

应用实例1：计算二叉树的结点总数量



计算二叉树结点总数量**算法思想：**

- 若二叉树为空，则返回0；
- 若二叉树非空，则分别计算根结点的左、右子树的结点总数量，并求和，再加1，然后返回。

二叉树的结点总数量 = 根结点左子树的结点总数 + 根结点右子树的结点总数 + 根结点数量(1)

二叉树遍历的应用

应用实例1：计算二叉树的结点总数量

计算二叉树的结点总数量

```
int TreeSize (BinaryTree *BT) {
    return Size(BT->root);
}

int Size(BTNode *t) {
    if(!t)
        return 0;
    else
        return Size(t->lChild) + Size(t->rChild) + 1
}
```

递归计算

计算左子树的结点总数

计算左子树的结点总数

计算二叉树结点总数量**算法思想**：

1. 定义二叉树总结点数量函数

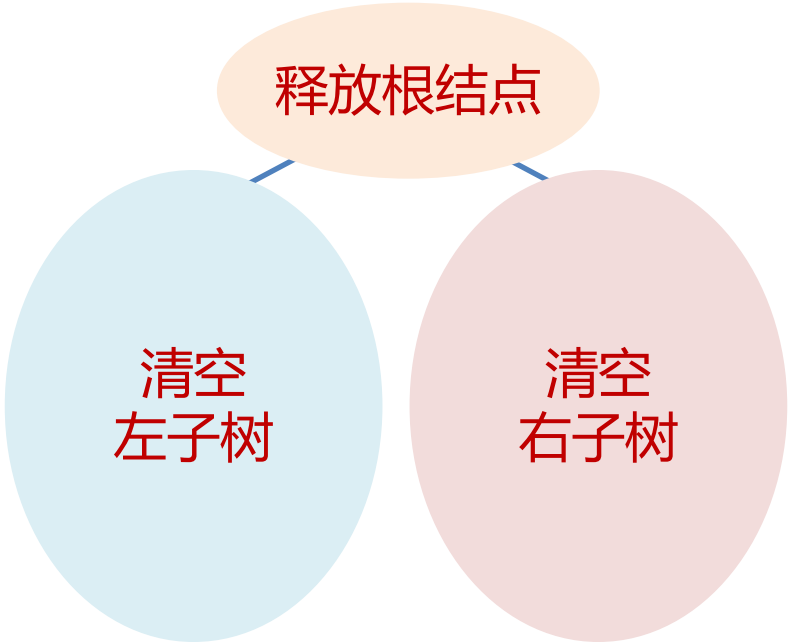
2. 计算以T为根的二叉树的结点总数量

2.1 若二叉树为空，则返回0；

2.2 若二叉树非空，则分别计算根结点的左、右子树的结点总数量，并求和，再加1，然后返回。

二叉树遍历的应用

应用实例2：清空二叉树



清空二叉树

```
void TreeClear (BinaryTree *BT) {// 1. 清空二叉树BT
      Clear (BT->root);
}
int Clear(BTNode *t) {                  // 2. 清空以T为根的二叉树
      if(!t)                              // 2.1 递归出口
            return;
      Clear(t->lChild);                   // 2.2 清空左子树
      Clear(t->rChild);                   // 2.3 清空右子树
      free(t);                            // 2.4 释放根结点
}
```

清空二叉树**算法思想：**

- 若二叉树为空，则二叉树无需清空，直接返回；
- 若二叉树非空，则先清空二叉树的左子树和右子树，最后再释放根结点

二叉树遍历的应用

应用实例3：计算二叉树叶子结点总数

算法思想：

1. 如果是空树，则叶子结点个数为0；
2. 否则，计算左子树叶子结点个数+右子树叶子结点个数。

若使用孩子兄弟法从存储，该如何统计叶子结点数?

什么是叶子节点？ 左右子树均等于0的结点

计算二叉树的结点总数量

```
int LeafCount (BinaryTree *BT) {
    if (BT == Null)
        return 0;
    if (T->lChild == Null && T->rChild == Null)
        return 1;
    else
        return LeafCount(T->lChild) + LeafCount(T->rChild);
}
```

二叉树遍历的应用

应用实例3：计算二叉树深度

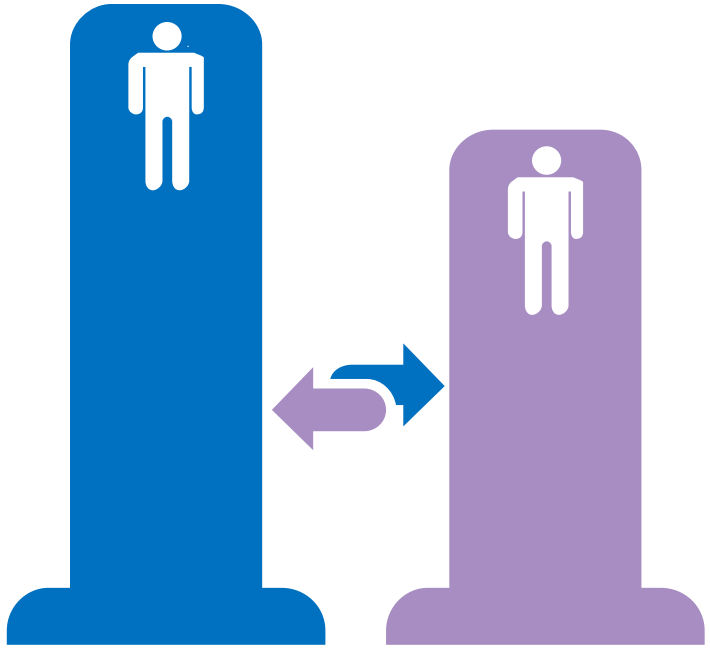
算法思想：

- 1. 如果是空树，则深度为 0；
- 2. 否则：

✓ 递归计算左子树的深度 m

✓ 递归计算右子树的深度 n

✓ 树的深度 = max(m, n) + 1



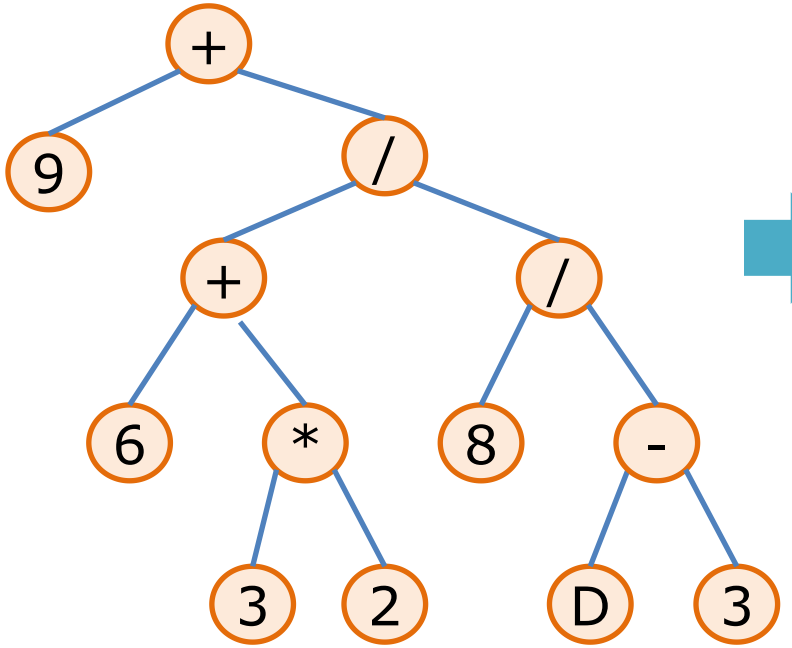
二叉树遍历的应用

应用实例4：表达式树

算术表达式：由常数、运算符和括号组成，有前缀、中缀、后缀三种表示法

9+(6+3*2)/(8/(5-3))

中缀表达式



- **前缀表示法：**

前序遍历结果：+ 9 / + 6 * 3 2 / 8 - 5 3

- **后缀表示法：**

后序遍历结果：9 6 3 2 * + 8 5 3 - / / +

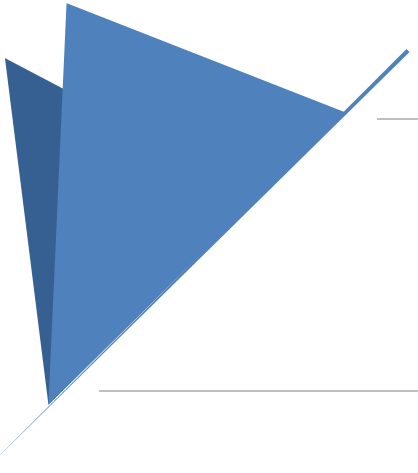
中序遍历结果：9 + 6 + 3 * 2 / 8 / 5 – 3

≠ 中缀表达法

没有括号无法正确计算

需要将括号加入到生成的表达式树中

- ✓ 叶节点对应常数
- ✓ 分支节点存放运算符
- ✓ 根节点是表达式中优先级最低的运算符



线索二叉树

/ 先序线索二叉树

/ 中序线索二叉树

/ 后序线索二叉树

/ 线索二叉树的序列化算法实现

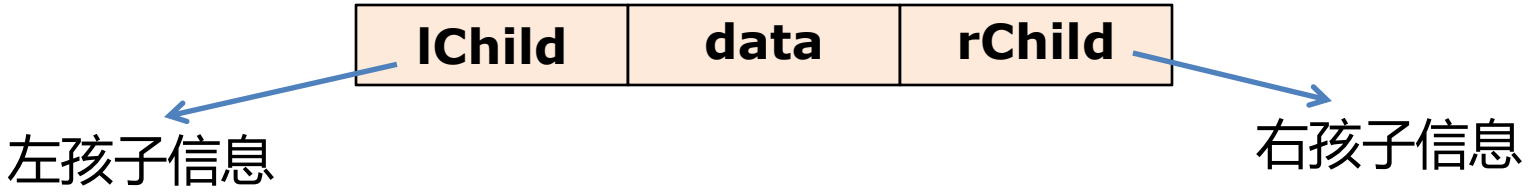
/ 线索二叉树的反序列化算法实现

列序列 -> 找空 -> 补线索

线索二叉树

什么是线索二叉树?

回顾：二叉树的存储结构，我们能从一个结点找到哪些相关信息?



- 无法从标准的二叉树结点中获取直接前驱和直接后继；
- 唯一的直接前驱和直接后继只能在遍历过程中获得。例如，二叉树先序遍历序列“A（B D E H）（C F I G）”指出了每一个结点的唯一前驱和唯一后继。

增加标志域 forward 和 backward 来标识前驱和后继信息。

线索二叉树

什么是线索二叉树?

包含线索结构的二叉链表结构称为线索链表，指向结点前驱和后继的指针称为线索；加上线索的二叉树称为**线索二叉树（Threaded Binary Tree）**。对二叉树以某种遍历为基础转变为线索二叉树的过程称为线索化或序列化。

若结点有左子树，则lTag=0，lChild域指向左孩子
若结点无左子树，则lTag=1，lChild域指向其前驱（线索）

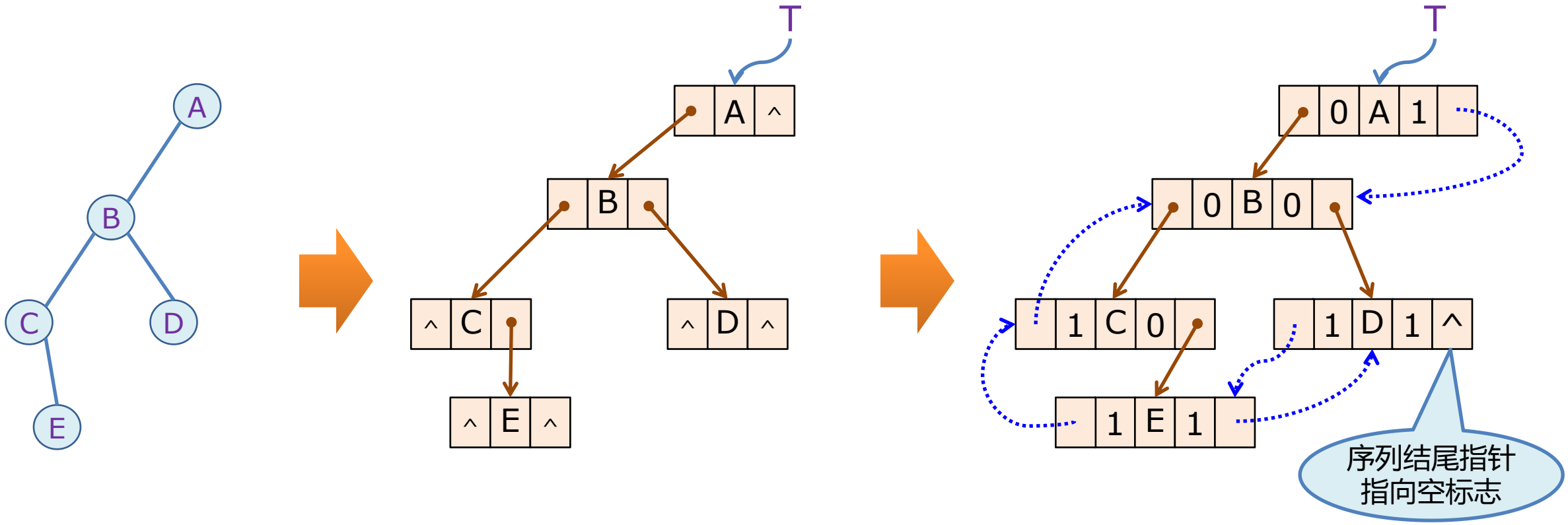
若结点有右子树，则 rTag=0，rChild域指向右孩子
若结点无右子树，则 rTag=1，rChild域指向其后继（线索）

二叉树的二叉线索存储表示

```
typedef struct BinThrNode {
    ElemType data;
    struct BinThrNode *lChild, *rChild;
    int lTag, rTag;
} BinThrNode, *BinThrTree
```

线索二叉树

先序线索二叉树



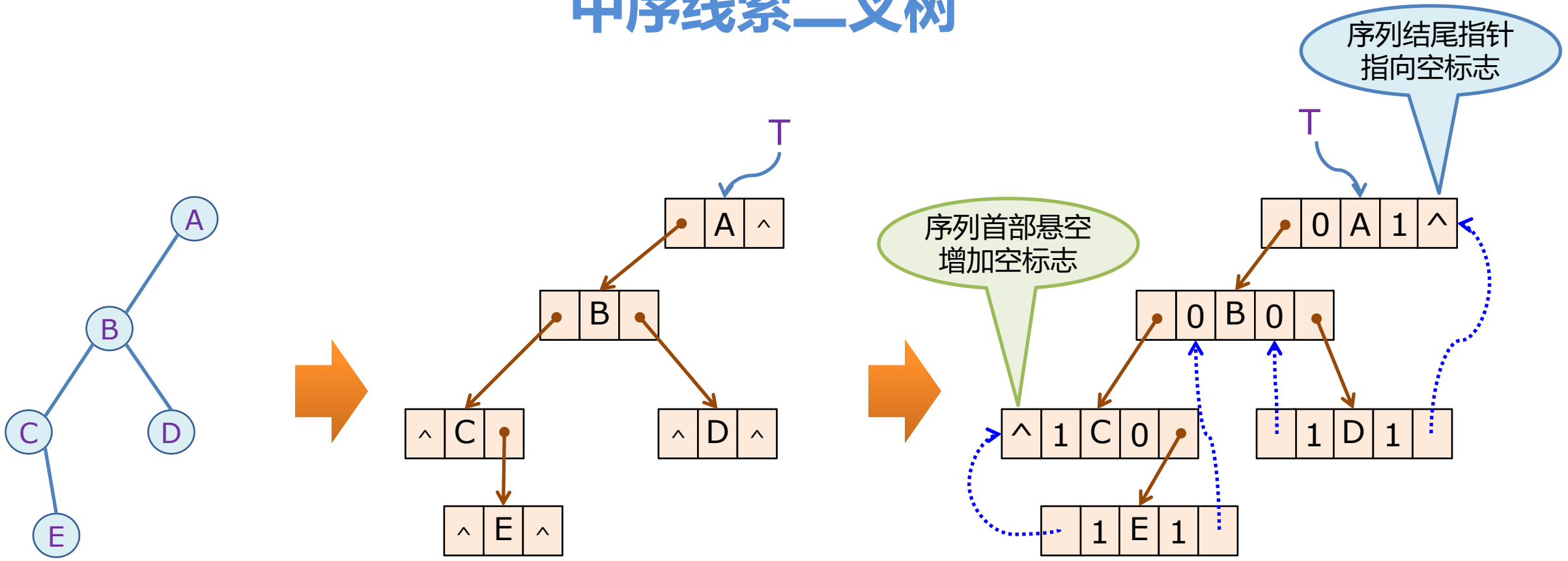
lTag=0，lChild域指向左孩子；lTag=1，lChild域指向其前驱

rTag=0，rChild域指向右孩子；rTag=1，rChild域指向其后继

先序遍历序列：A B C E D

线索二叉树

中序线索二叉树



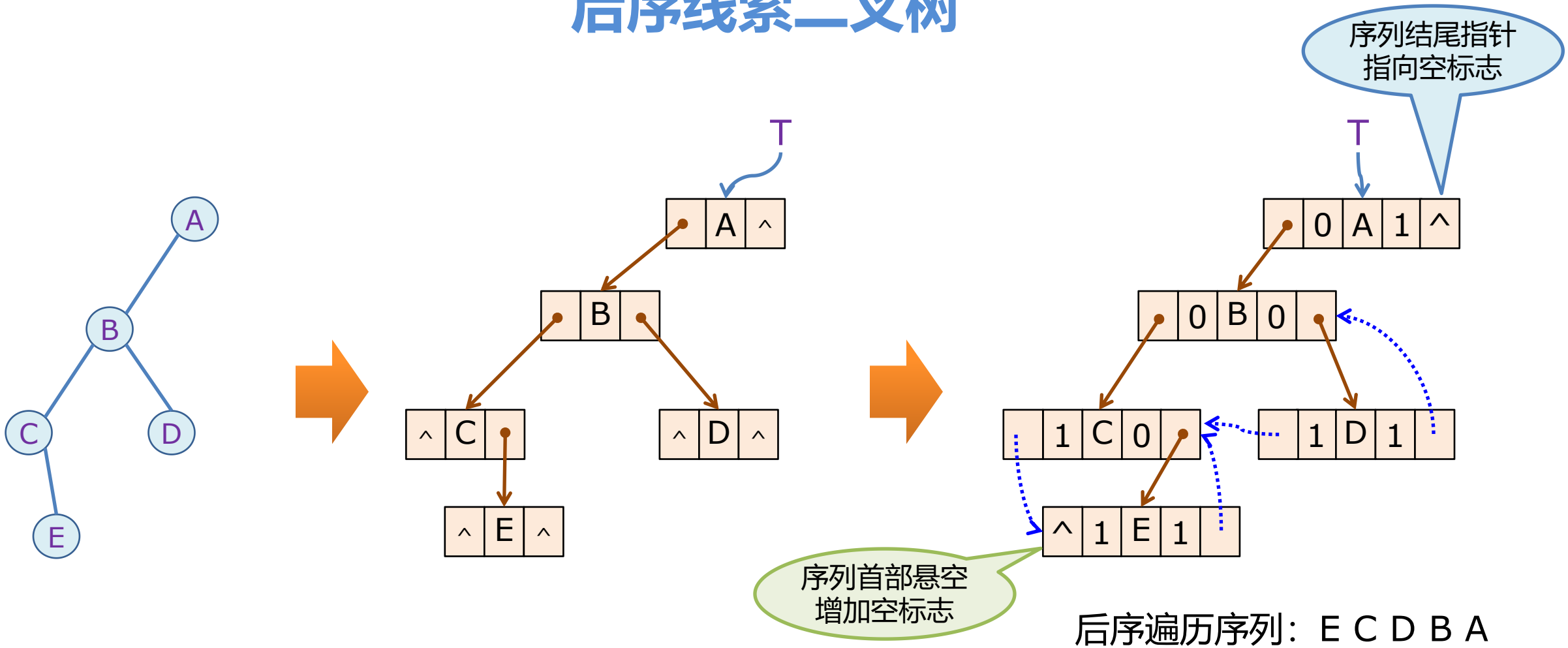
中序遍历序列：C E B D A

lTag=0，lChild域指向左孩子；lTag=1，lChild域指向其前驱

rTag=0，rChild域指向右孩子；rTag=1，rChild域指向其后继

线索二叉树

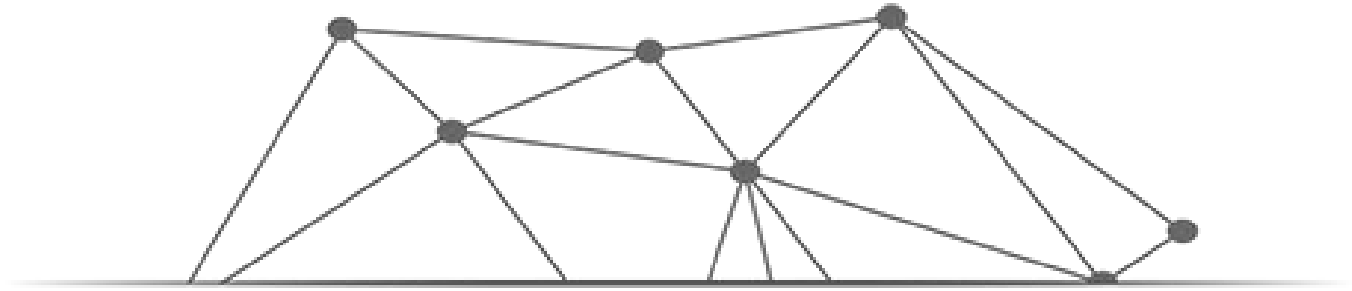
后序线索二叉树



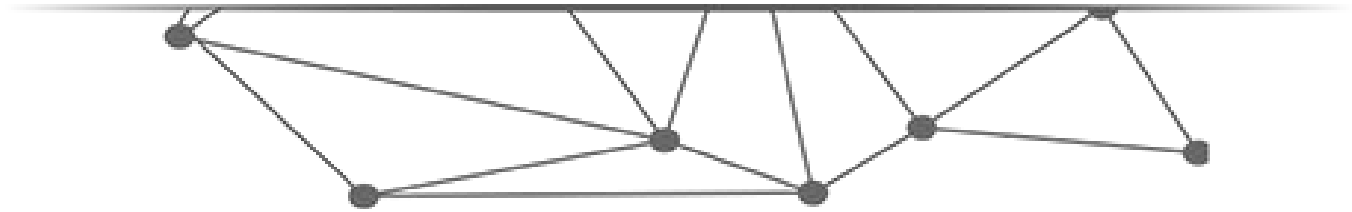
后序遍历序列：E C D B A

lTag=0，lChild域指向左孩子；lTag=1，lChild域指向其前驱

rTag=0，rChild域指向右孩子；rTag=1，rChild域指向其后继



课堂互动 11.3



第12讲 树的遍历

小　结

- 二叉树的遍历算法是其他运算的基础
- 通过遍历可以得到二叉树中结点访问的线性序列，实现非线性结构的线性化。
- 根据结点访问次序的不同，常见遍历方法包括先序遍历、中序遍历、后序遍历
- 按从上到下的顺序，可以实现树的层次遍历
- 树遍历的**时间复杂度**都是 **O(n)**
- 在**线索二叉树**中，可以利用二叉链表中的空指针域（不含左右子树指针的结点）来存放指向某种遍历次序下的前驱结点和后继结点指针，这些附加的指针称为“**线索**”。
- **线索二叉树**有利于加速查找结点前驱和后继的速度。

欧老师的联系方式

读万卷书 行万里路 只为最好的修炼

QQ：14777591 (宇宙骑士)

Email：ouxinyu@alumni.hust.edu.cn

Website: http://ouxinyu.cn

Tel：18687840023

地址：安宁校区 诚远楼201

南院 智能应用研究院A306-2