6.3 遍历二叉树

一. 遍历的含义

树是一种非线性的数据结构，在对它进行操作时，总是需要逐一对每个数据元素实施操作，这样就存在一个操作顺序问题，由此提出了树的遍历操作。

所谓遍历树就是按某种顺序访问树中的每个结点一次且仅一次的过程。这里的访问可以是输出、比较、更新、查看元素内容等等各种操作。

6.3 遍历二叉树

一. 遍历的含义

- 树的遍历就是从根结点出发，按某种次序访问树中的结点，要求每个结点访问一次且仅访问一次
- 遍历的结果：产生一个关于结点的线性序列。（非线性结构线性化）
- 对“二叉树”而言，可以有三条搜索路径：
  - 先上后下的按层次遍历；
  - 先左（子树）后右（子树）的遍历
  - 先右（子树）后左（子树）的遍历

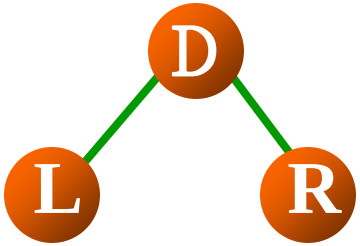
6.3 遍历二叉树

一. 遍历的含义

- 一个二叉树由根结点与左子树和右子树组成
- 设访问根结点用D表示，遍历左、右子树用L、R表示

二叉树的遍历方式：**DLR**、**DRL**、**LDR**、**LRD**、**RLD**、**RDL**

- 如果规定先左后右，则共有三种方式：
  - DLR [先序遍历]PreOrder
  - LDR [中序遍历]InOrder
  - LRD [后序遍历]PostOrder



6.3 遍历二叉树

二. 先序遍历

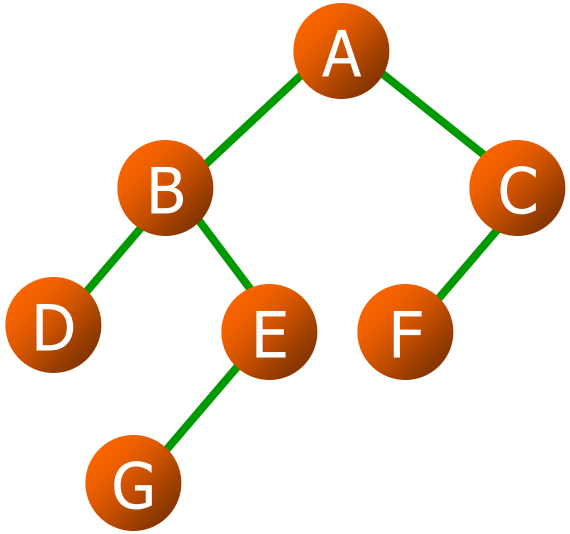
- 递归算法

若二叉树为空，则空操作返回；否则：

① 访问根结点(D)

② 先序遍历左子树(L)

③ 先序遍历右子树(R)



算法(举例)：

过程：A(AL)(AR)

过程：AB(BL)(BR)(AR)

过程：ABD(BR)(AR)

输出结果：ABDEGCF

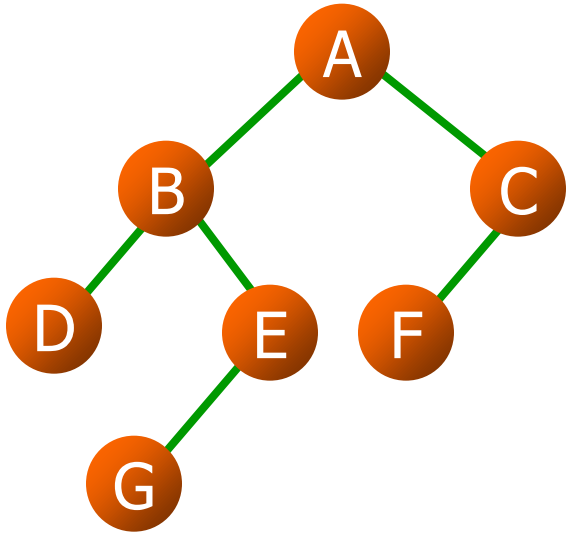
(第一个输出结点必为根结点)

6.3 遍历二叉树

二. 先序遍历

- 递归程序实现

```
void PreOrderTraverse ( BinTree T ) {
    if ( T ! = NULL)
    {
        cout << T->data;
        PreOrderTraverse ( T->lChild );
        PreOrderTraverse ( T->rChild );
    }
}
```



6.3 遍历二叉树

三. 中序遍历

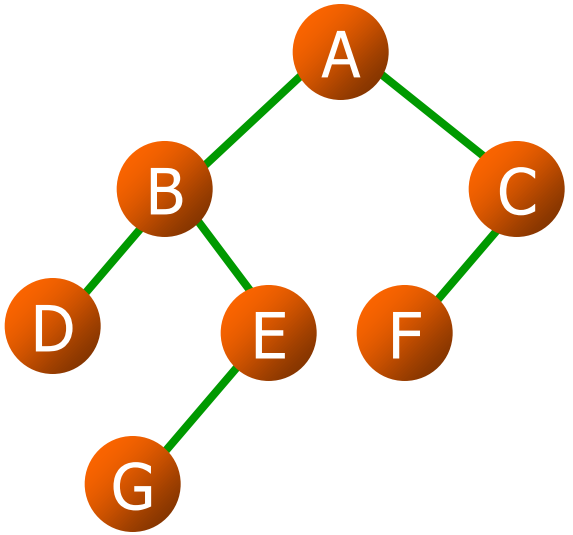
- 递归算法

若二叉树为空，则空操作返回；否则：

① 中序遍历左子树(L)

② 访问根结点(D)

③ 中序遍历右子树(R)



算法(举例)：

过程：(AL)A(AR)

过程：(BL)B(BR)A(AR)

过程：DB(EL)E(ER)A(AR)

过程：DBGEA(AR)

输出结果：DBGEAFC

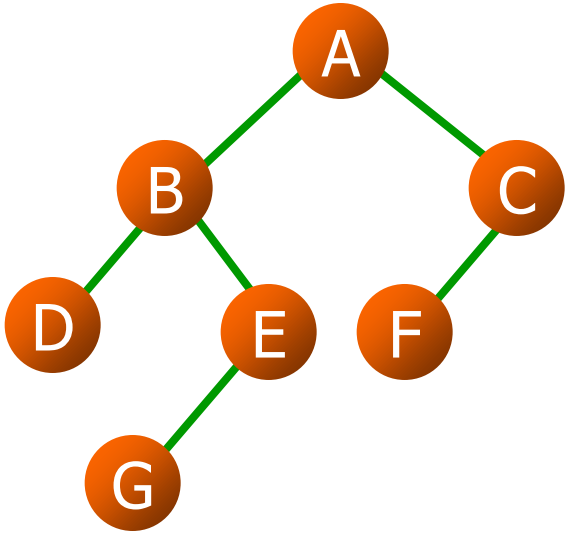
（先于根结点输出的为左子树的结点，后于根结点输出的为右子树的结点）

6.3 遍历二叉树

三. 中序遍历

- 递归程序实现

```
void InOrderTraverse ( BinTree T ) {
    if ( T ! = NULL)
    {
        InOrderTraverse ( T->lChild );
        cout << T->data;
        InOrderTraverse ( T->rChild );
    }
     else   return;
}
```



6.3 遍历二叉树

四. 后序遍历

- 算法

若二叉树为空，则空操作返回；否则：

① 后序遍历左子树(L)

② 后序遍历右子树(R)

③ 访问根结点(D)

算法(举例)：

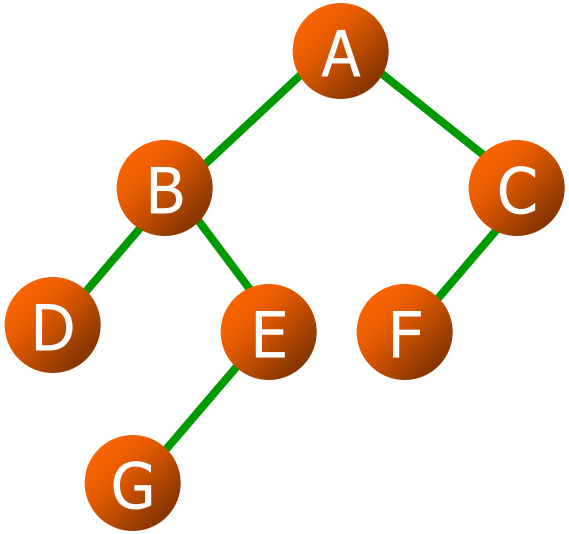
过程：(AL)(AR)A

过程：(BL)(BR)B(AR)A

过程：D(EL)(ER)EB(AR)A

过程：DGEB(AR)A

输出结果：DGEBFCA



（最后一个输出结点必为根结点）

6.3 遍历二叉树

四. 后序遍历

- 递归算法程序实现

```
void PostOrderTraverse ( BinTree T ) {
     if ( T ! = NULL)
     {
       PostOrderTraverse ( T->lChild );
       PostOrderTraverse ( T->rChild );
       cout << T->data;
     }
}
```

6.3 遍历二叉树

遍历练习:

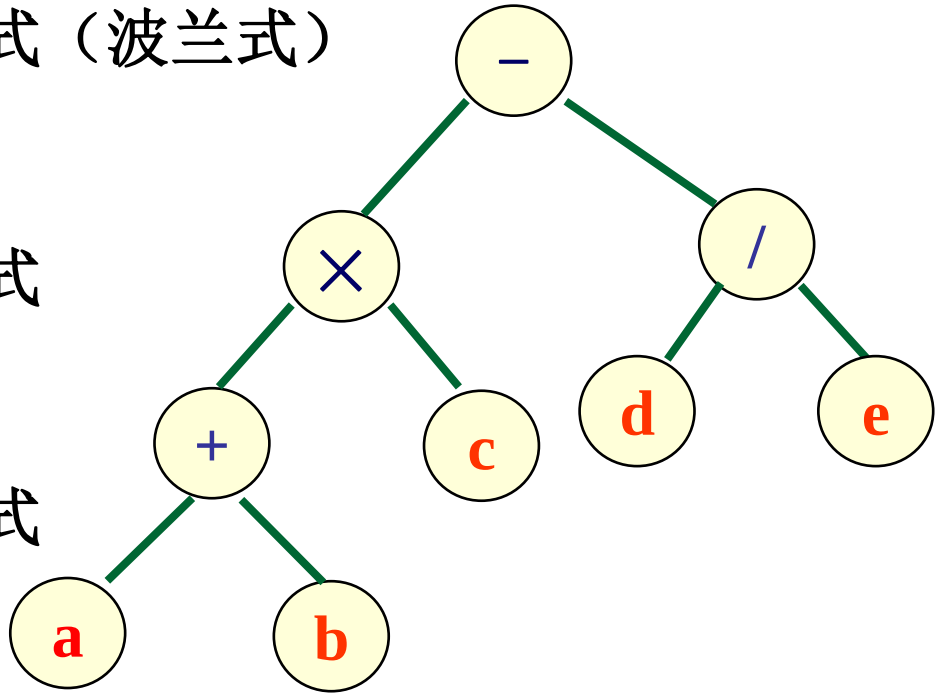
- 用二叉树来表示表达式
  - 先序遍历得到前缀表达式（波兰式）

    -×+ a b c / d e
  - 中序遍历得到中缀表达式

    (a+b)×c – d/e
  - 后续遍历得到后缀表达式

    （逆波兰式）

    ab+c×de/-



6.3 遍历二叉树

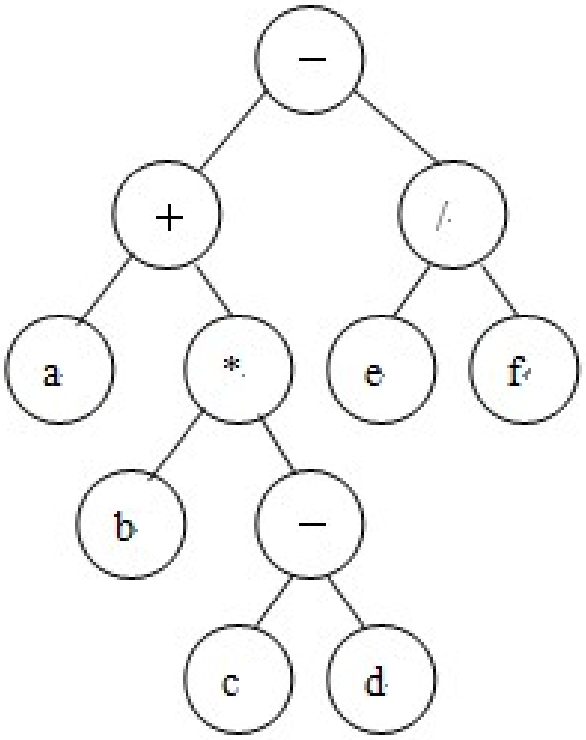
表达式与二叉树

- 当我们对此二叉树进行先序、中序和后序遍历后，便可得到表达式的前缀、中缀和后缀形式：

  前缀：-+a*b-cd/ef

  中缀：a+b*c-d-e/f

  后缀：abcd-*+ef/-



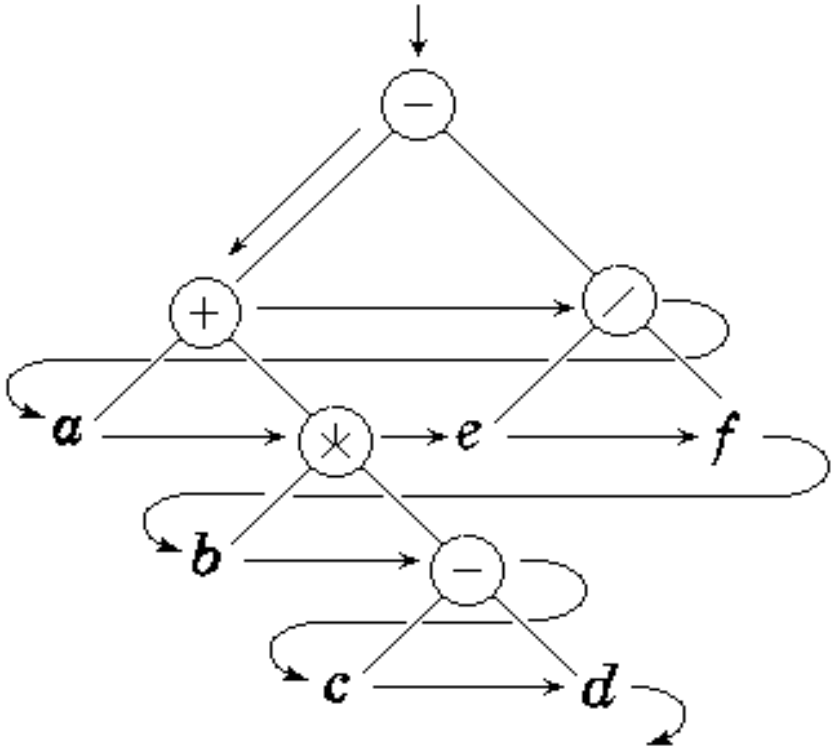
- 后缀表达式：不包含括号，运算符放在两个运算对象的后面，所有的计算按运算符出现的顺序，严格从左向右进行，不再考虑运算符的优先规则

- 中缀形式是表示算术表达式的通常形式；在计算机中，往往使用后缀表达式易于求值

6.3 遍历二叉树

五. 层次遍历

- 层次遍历二叉树，是从根结点开始遍历，按层次次序“自上而下，从左至右”访问树中的各结点。



6.3 遍历二叉树

五. 层次遍历

- 为保证是按层次遍历，从根开始逐层访问，用FIFO队列实现。

① 根结点不空，将根结点进队；

② 队不空时循环： 从队列中出队一个结点*p，访问它；

若它有左孩子结点，将左孩子结点进 队；

若它有右孩子结点，将右孩子结点进队。

6.3 遍历二叉树

五. 层次遍历

- 非递归算法：
  - 设T是指向根结点的指针变量，若二叉树为空，则返回；
  - 否则，令p=T，p入队，执行以下循环：

    (1) 队首元素出队到p；

    (2) 访问p所指向的结点；

    (3) 将p所指向的结点的左、右子结点依次入队。

    直到队空为止。

6.3 遍历二叉树

二叉树遍历的非递归算法

◆ 先序遍历

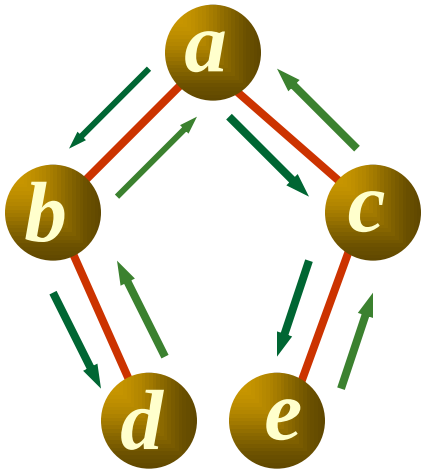
关键： 在先序遍历过某结点的整个左子树后，如何找到该结点的右孩子？

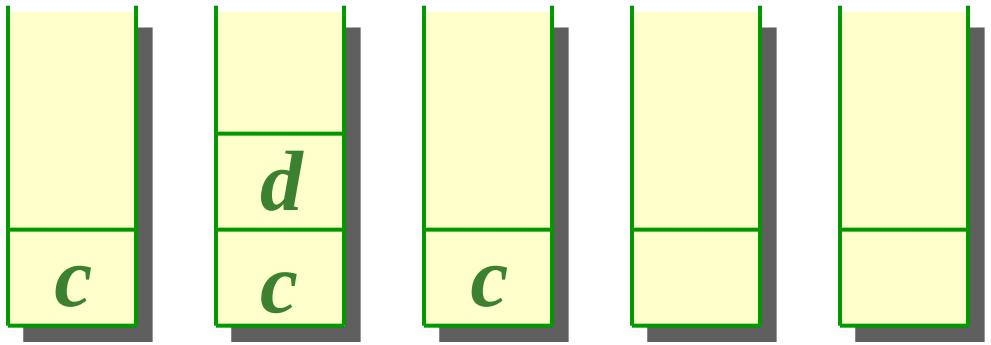
解决办法：

在访问完该结点后，将该结点的右孩子保存在栈中，以便以后能通过它找到右子树的根结点。

6.3 遍历二叉树

先序遍历非递归算法





访问	访问	退栈	退栈	访问
a	*b*	*d*	*c*	*e*
进栈	进栈	访问	访问	左进
c	*d*	*d*	*c*	*空*
左进	左进	左进	左进	栈空
b	*空*	*空*	*e*	结束

6.3 遍历二叉树

- 先序遍历非递归算法
  - 设T是根结点的指针变量，若二叉树为空，则返回；

    否则，令p=T，执行以下循环直到p为空：

⑴ 访问p所指向的结点；

⑵ q=p->Rchild ，若q不为空，则q进栈；

⑶ p=p->Lchild ，若p不为空，转(1)，否则转(4)；

⑷ p->Lchild 为空，则将栈顶结点出栈，并令p指向出栈结点，转(1)，直到栈空为止。

6.3 遍历二叉树

- 先序非递归程序实现

```
Status PreOrderTraverse(BiTree T, Status (*Visit)(ElemType)) {
  Stack S;  BiTree p;
  if( T == NULL ) return ERROR;
  InitStack(S);  p = T;
  while ( p || ! StackEmpty(S)) {
    if ( p ) {  //访问其所指结点，非空右指针进栈，继续左进
        if ( !Visit(p->data) ) return ERROR;
        q = p->rchild;
        if( q ) Push(S, q);
        p = p->lchild;
     }
     else       Pop(S, p);  // 右指针退栈
     }
  return OK;
}
```

6.3 遍历二叉树

二叉树遍历的非递归算法

◆ 中序遍历

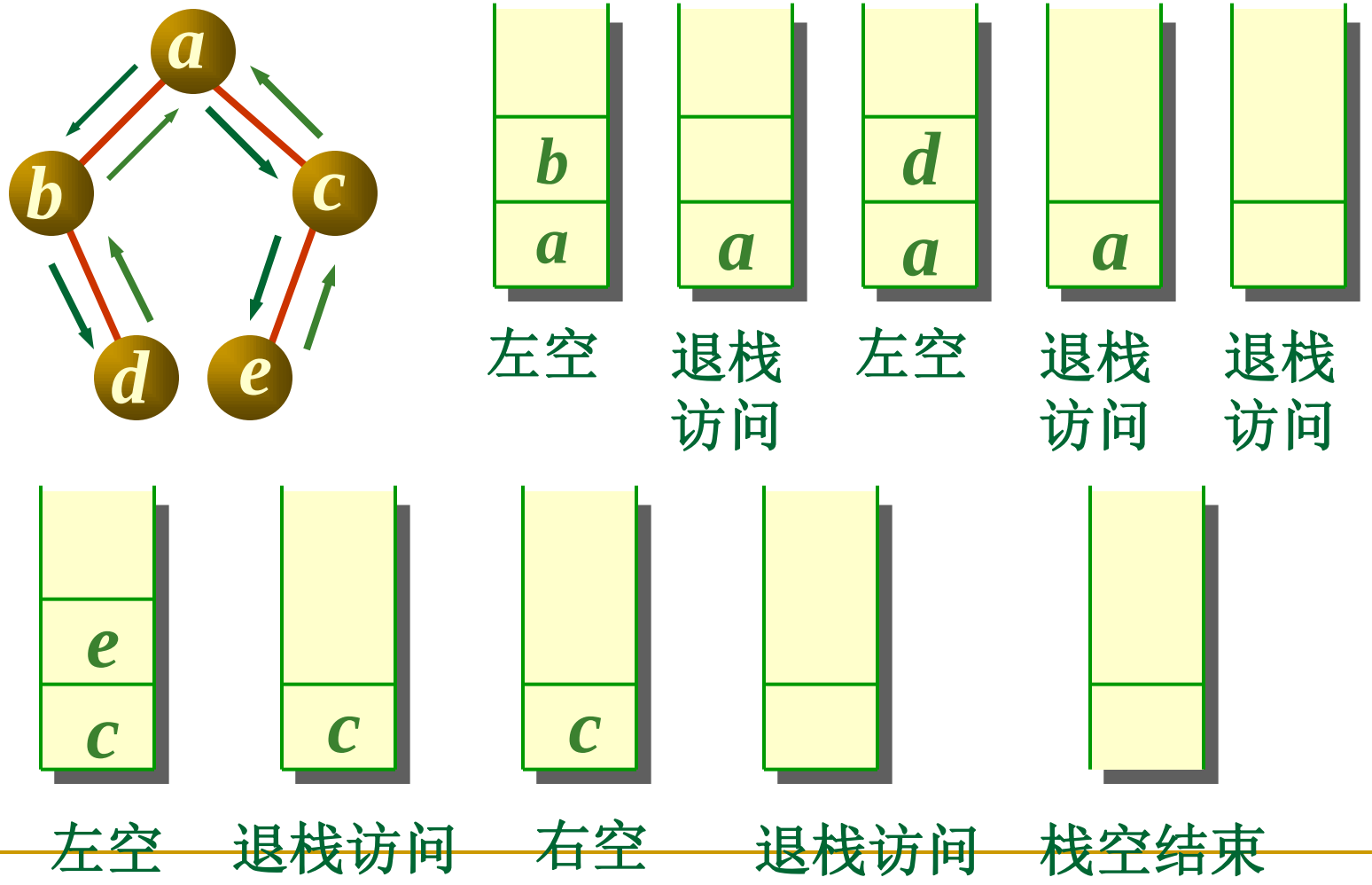
关键： 在中序遍历过某结点的整个左子树后，如何找到该结点？

解决办法：

在访问该结点的左子树前，将该结点保存在栈中，以便以后能再找到它及它的右子树。

6.3 遍历二叉树

中序遍历——非递归算法



6.3 遍历二叉树

- 中序遍历非递归算法
  - 设T是指向根结点的指针变量，若二叉树为空，则返回；

否则，令p=T，执行以下循环：

(1) 若p不为空，p进栈， p=p->Lchild ；

(2) 否则(即p为空)，退栈到p，访问p所指向的结点；

(3) p=p->Rchild ，转(1)；

直到栈空为止。

6.3 遍历二叉树

- 中序非递归程序实现

```
Status InOrderTraverse(BiTree T, Status (*Visit)(ElemType)) {
  Stack S;  BiTree p;
  InitStack(S);  p = T;
  while ( p || !StackEmpty(S)) {
    if ( p ) { Push(S, p);  p = p->lchild; }  // 非空指针进栈，继续左进
    else {        // 上层指针退栈，访问其所指结点，再向右进
      Pop(S, p);
      if (!Visit(p->data)) return ERROR;
      p = p->rchild;
    }
  }
  return OK;
} //代码关键：一个左孩子同时也是一个父亲
```

6.3 遍历二叉树

后序非递归算法

- 设立状态标志变量**tag**，**tag=0**表示结点暂不能访问； **tag=1**表示结点可以访问；
- 设两个堆栈S_1、S_2 ，S_1保存结点，S_2保存结点的状态标志变量**tag** 。S_1和S_2共用一个栈顶指针
- 设**T**是指向根结点的指针变量，若二叉树为空，则返回；

  否则，令p=T，执行以下循环：

  （采用do-while结构，循环条件是当s1栈不空则继续循环）

1. 若p不空：p进栈S1， tag赋值0进栈S2
2. p=p->Lchild，跳转1
3. 当s1栈为空，跳出循环，使用break语句
4. 若p为空，取s1栈的状态标志值tag

   ①若tag=0：修改S2栈顶元素值(0变成1)，p取S1栈顶元素的右子树

   ②若tag=1：S1出栈到p，S2出栈，输出p的数据表示结点已访问；

   然后p设为空，这样使得do-while循环中的步骤1和2不用执行，跳到步骤3

循环直到栈空为止

6.3 遍历二叉树

六. 遍历的应用

遍历二叉树是二叉树各种操作的基础，遍历算法中对每个结点的访问操作可以是多种形式及多个操作，根据遍历算法的框架，适当修改访问操作的内容，可以派生出很多关于二叉树的应用算法。

例如：

- 应用后序遍历求二叉树的结点个数；
- 应用后序遍历求二叉树的高度（或深度）；
- 应用前序遍历判断两棵二叉树是否相等。

6.3 遍历二叉树

六. 遍历的应用

遍历是二叉树各种操作的基础，可以在遍历的过程中进行各种操作，例如建立一棵二叉树。

1. 根据先序序列构建二叉树的二叉链表（物理上）

2. 根据遍历序列来确定二叉树的结构（逻辑上）

6.3 遍历二叉树

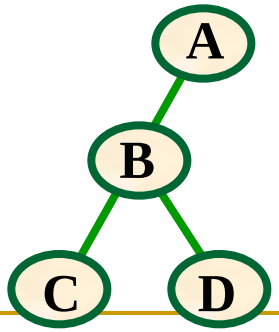
1、建立二叉树的存储结构

- 在此讨论利用先序遍历建立二叉树链表

假设二叉树中的结点均为一个字符，由键盘输入二叉树的先序遍历序列。这里约定：

空树　　以字符“#”表示

只含一个根结点的二叉树　　以字符串“A##”表示



以字符串“ABC##D###”表示

6.3 遍历二叉树

1、建立二叉树的存储结构

- 在此讨论利用二叉树的先序遍历建立二叉链表

假设T为指向二叉树根结点的指针，二叉链表的建立过程是：

- ✓ 首先建立根结点，若输入的是一个“#”字符，则表明该二叉树为空树，即T=NULL；
- ✓ 否则输入的字符应该赋给T->data,，之后依次递归建立它的左子树和右子树 。

6.3 遍历二叉树

二叉树的构建

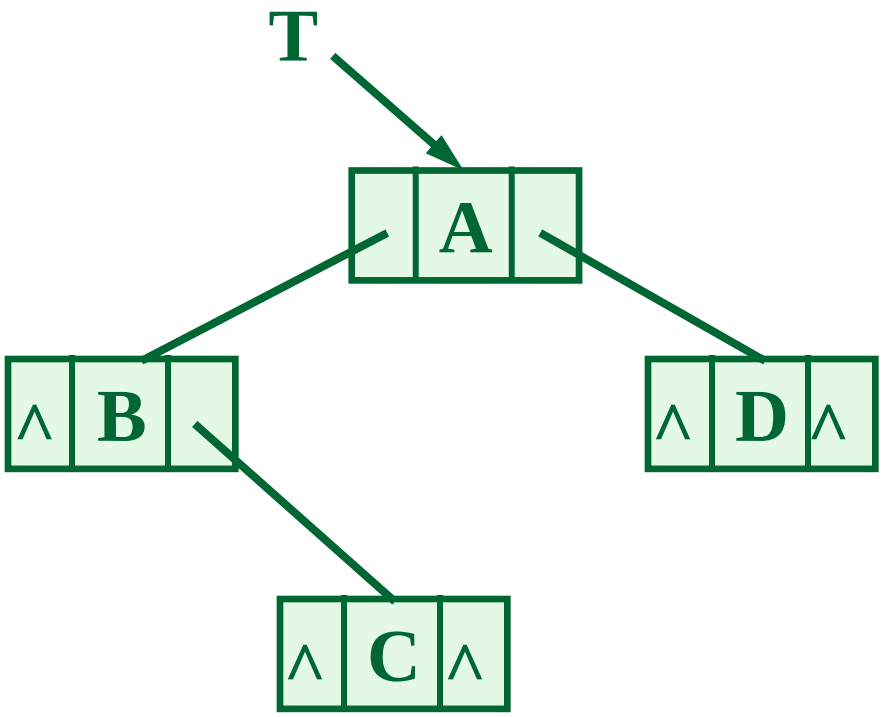
- C语言版，逐个输入字符构建链式存储结构

```
BiTree CreateBiTree(BiTree &T) {
char ch;
  scanf("%c",&ch);
  if ( ch=='#' ) T = NULL;
  else {
    if ( !(T = (BiTNode *)malloc(sizeof(BiTNode))) )  return ERROR;
    T->data = ch;              // 生成根结点
    CreateBiTree(  T->lchild  );   // 构造左子树
    CreateBiTree(  T->rchild  );   // 构造右子树
  }
  return T;
} // CreateBiTree
```

6.3 遍历二叉树

二叉树的构建

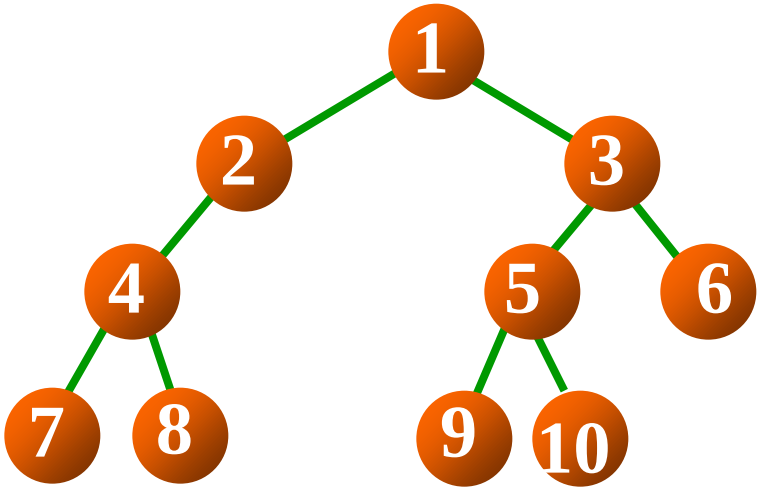
A B ■ C ■ ■ D ■ ■



6.3 遍历二叉树

二叉树的构建

- 按完全二叉树的顺序表示，结合二叉树性质5构建

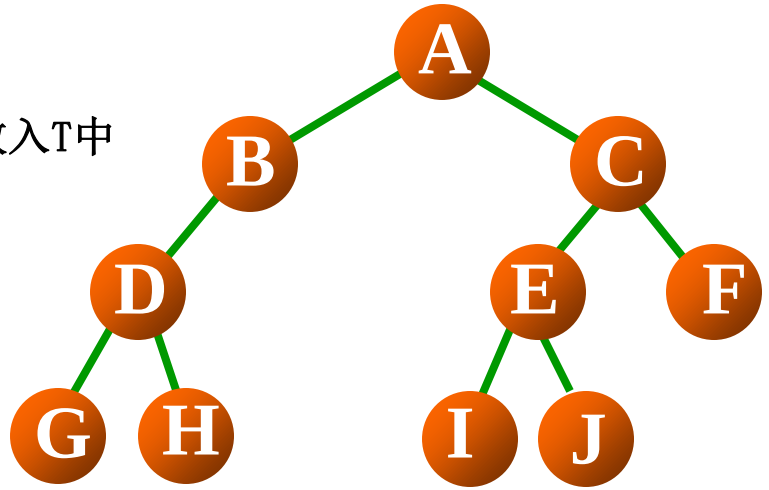


6.3 遍历二叉树

二叉树的构建

- 按完全二叉树的顺序表示，结合二叉树性质5构建
  - 根据二叉树性质5，因为数组从0编号，父结点是i，则左、右孩子的位置是2i+1和2i+2
  - 采用递归实现，设定函数参数pos，表示当前结点在数组的位置
  - 函数作用：根据pos位置的数值创建结点t，返回结点t
  - 输入：一个数组的数据，实际就是一个字符串，其中字符0表示结点为空
  - 算法流程：

1. 数组越界检查，检查pos如果超过数组长度，直接返回空
2. 获取pos位置的数值，如果为字符0，则结点t为空
3. 如果数值不为0，执行以下过程
   ① 为结点T分配空间，使用new方法，并把数值放入T中
   ② 在第2i+1位置递归创建t的左子树
   ③ 在第2i+2位置递归创建t的右子树
4. 返回结点t



6.3 遍历二叉树

六. 遍历的应用

问题：由已知遍历序列，如何求二叉树？

2. 根据遍历序列来确定二叉树的结构（逻辑上）

6.3 遍历二叉树

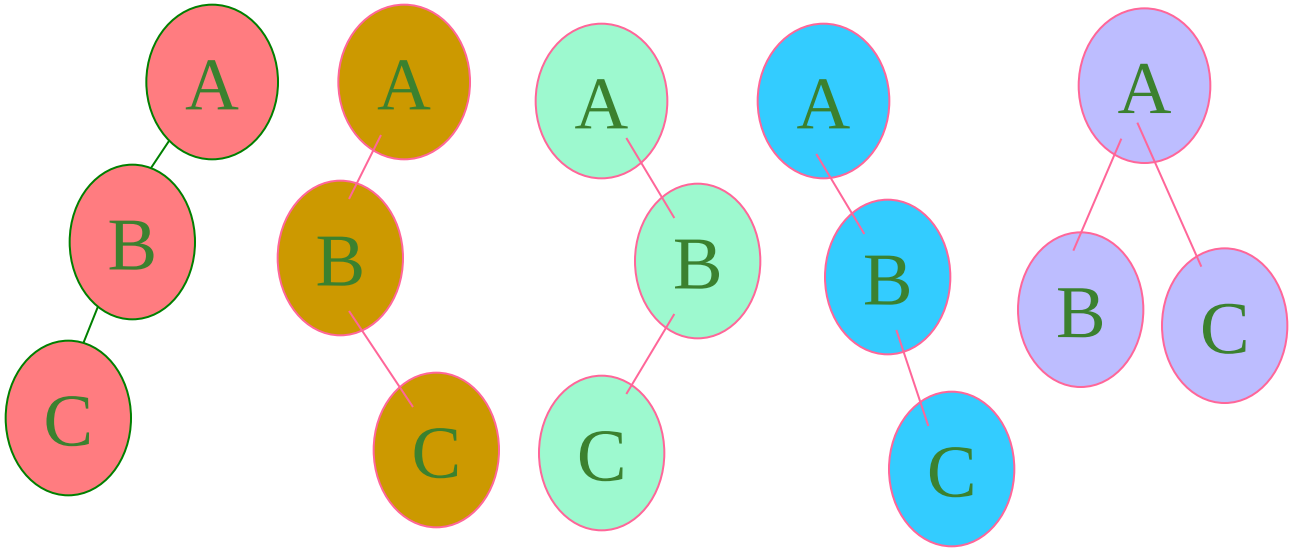
六．遍历的应用

2、确定二叉树的逻辑结构

- 已知先序序列和中序序列，可唯一确定一棵二叉树。

若ABC是二叉树的先序序列，可画出5棵不同的二叉树；

若ABCD是二叉树的先序序列，可画出14棵不同二叉树。



6.3 遍历二叉树

六．遍历的应用

2、确定二叉树的逻辑结构

- 已知先序序列和中序序列，可唯一确定一棵二叉树。

  仅知二叉树的先序序列，不能唯一确定一棵二叉树。

若已知二叉树的先序序列，并加上二叉树的中序序列，就可唯一确定一棵二叉树。

先序序列：　**D**　L　R

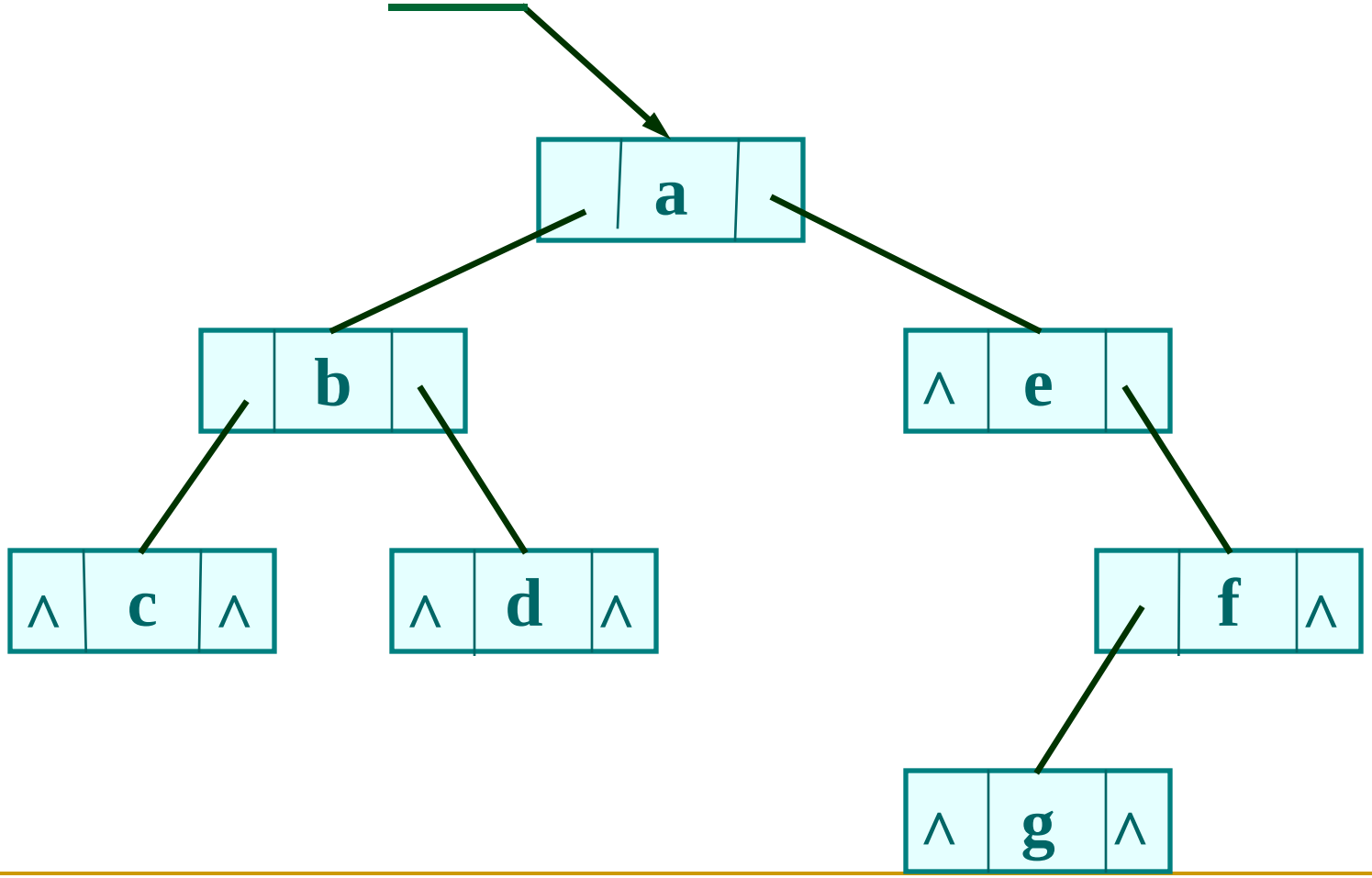
中序序列：　L　**D**　R

在先序序列中确定根，在中序序列中分左右子树

例：

a b c d e f g　先序序列

c b d a e g f　中序序列



6.3 遍历二叉树

2、确定二叉树的逻辑结构

根据遍历顺序来确定二叉树的逻辑结构。

- 根据两种遍历序列构建二叉树，根据遍历的特性已知：
  - ① 先序遍历中，第一个输出结点必为根结点
  - ② 中序遍历中，先于根结点D输出的结点为左子树的结点，后于根结点D输出的结点为右子树的结点
  - ③ 后序遍历中，最后一个输出结点必为根结点
- 构建方法：
  - 先序+中序构建
  - 中序+后序构建

6.3 遍历二叉树

2、确定二叉树的逻辑结构

- 已知先、中序遍历构建二叉树：

1. 在先序遍历序列中，第一个结点为根结点D
2. 在中序遍历序列中，根结点D左边的结点归为左子树，根结点D右边的结点归为右子树
3. 对每个子树反复使用1,2两步，直到确定二叉树。

6.3 遍历二叉树

2、确定二叉树的逻辑结构

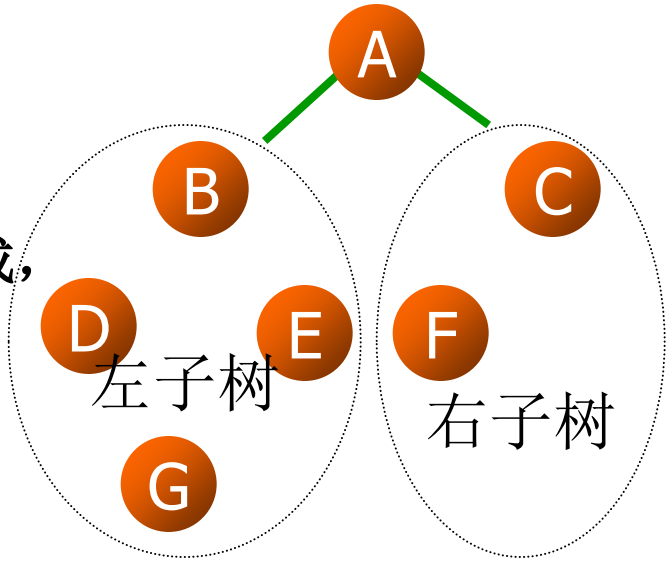
- 举例：已知先、中序遍历构建二叉树。
  - 先序遍历序列为：ABDEGCF
  - 中序遍历序列为：DBGEAFC

1、根据先序遍历序列，可知根结点为A；再根据中序遍历序列可知，左子树由DBGE组成，右子树由FC组成

2、在左子树的DBGE中，根据先序找出根结点是B，然后根据中序遍历，又分为左孙子树为D，右孙子树为GE

3、在右子树的FC中，根据先序遍历找出根结点C，然后根据中序遍历，由分为左孙子树为F，右孙子树为空

以此类推



6.3 遍历二叉树

2、确定二叉树的逻辑结构

- 已知后、中序遍历构建二叉树：

1. 在后序遍历序列中，最后一个结点为根结点D
2. 在中序遍历序列中，根结点D左边的结点归为左子树，根结点D右边的结点归为右子树
3. 对每个子树反复使用1,2两步，直到确定二叉树

6.3 遍历二叉树

2、确定二叉树的逻辑结构

- 举例：

  后序遍历序列为：DGEBFCA

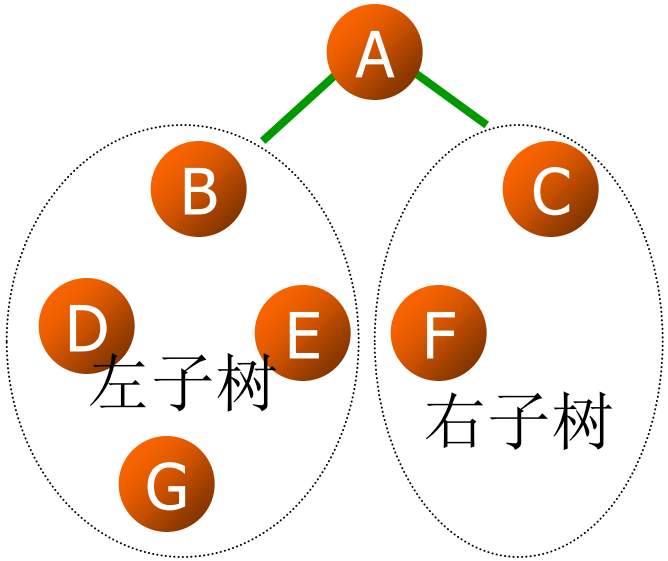
  中序遍历序列为：DBGEAFC

1、根据后序遍历序列，可知根结点为A；再根据中序遍历序列可知，左子树由DBGE组成，右子树由FC组成

2、在左子树的DBGE中，根据后序找出根结点是B，然后根据中序遍历，又分为左孙子树为D，右孙子树为GE

3、在右子树的FC中，根据后序遍历找出根结点C，然后根据中序遍历，又分为左孙子树为F，右孙子树为空

以此类推



练习

假设一颗二叉树的先序遍历序列为**ABDEHCFGI**

中序遍历序列为**DBEHAFCIG**

(1) 请画出这颗二叉树

(2) 思考：如果给出后序遍历序列和中序遍历序列，是否可以确定该二叉树的结构？