6 树和二叉树

董洪伟

http://hwdong.com

树和二叉树

- 主要内容

  一、树的类型定义

  二、二叉树的类型定义

  三、二叉树的存储结构

  四、二叉树的操作

  五、线索二叉树

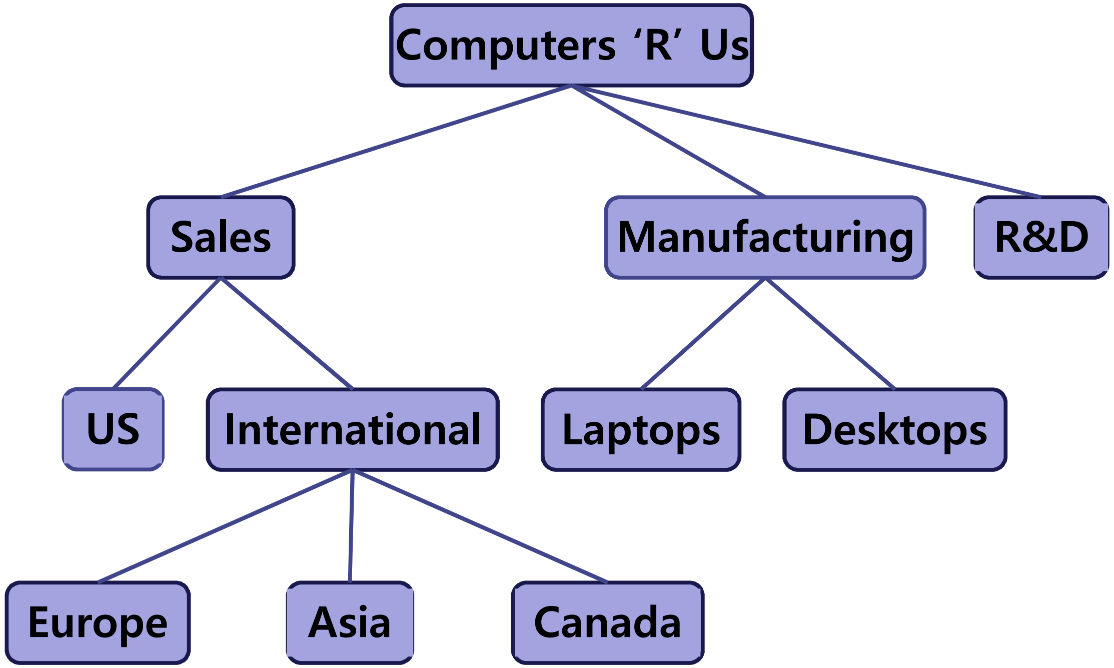
  六、树和森林

  七、赫夫曼树

  八、树的计数

树的类型定义

- 树是一个层次结构的抽象模型
- 树是由具有父子关系的结点构成的
- 应用示例：
  - –组织结构
  - –文件系统

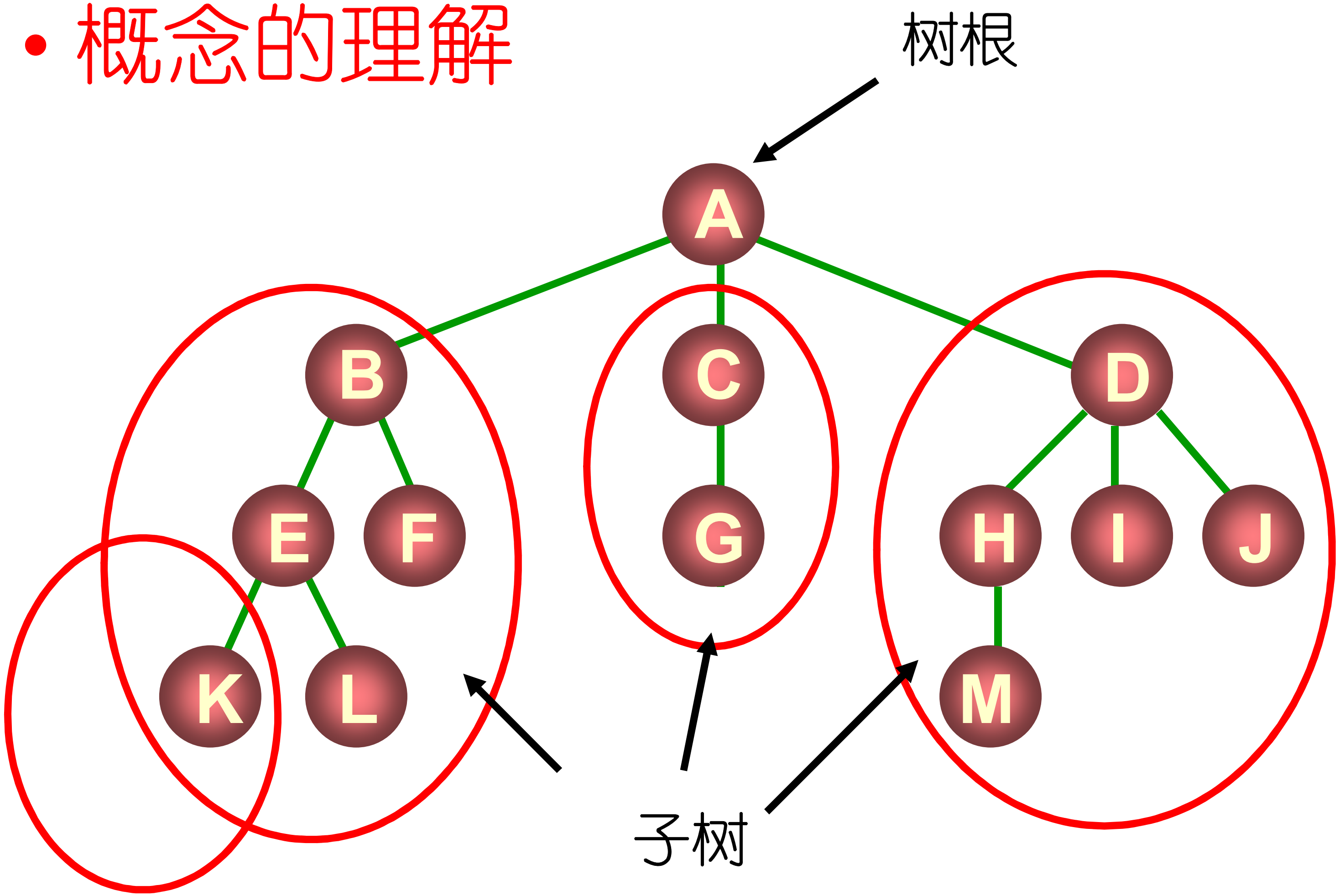


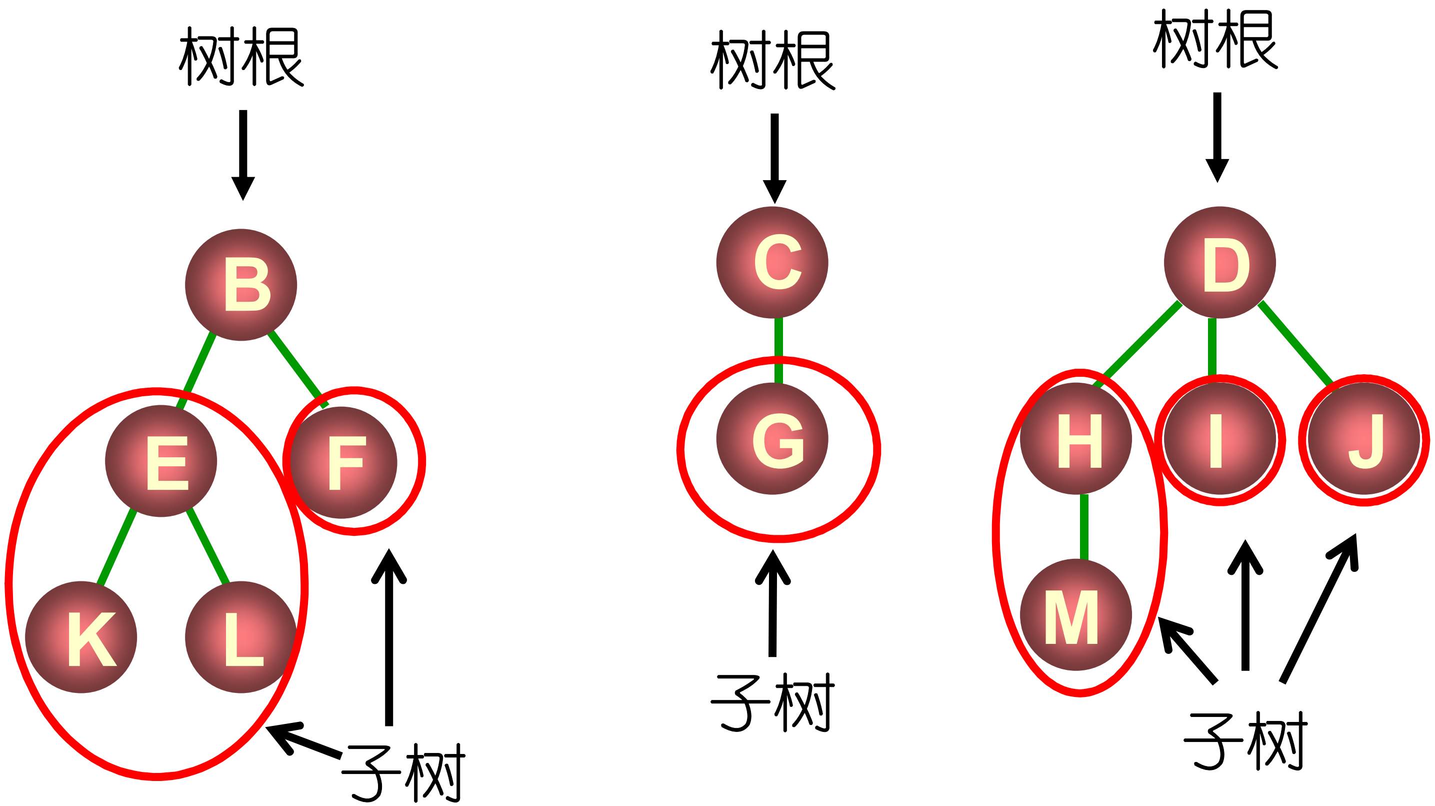
树的类型定义

- 树的定义(**Tree**)
  - 树是由**n(n>=0)**个结点组成的有限集合
  - 如果**n=0**，称为空树
  - 如果**n>0**，则
    - 有一个特定的称之为根(**root**)的结点，它只有直接后继，但没有直接前驱
    - 除根以外的其它结点划分为**m(m>=0)**个互不相交的有限集合$T_0, T_1, \ldots, T_{m-1}$，每个集合又是一棵树，并且称之为根的子树

树的类型定义

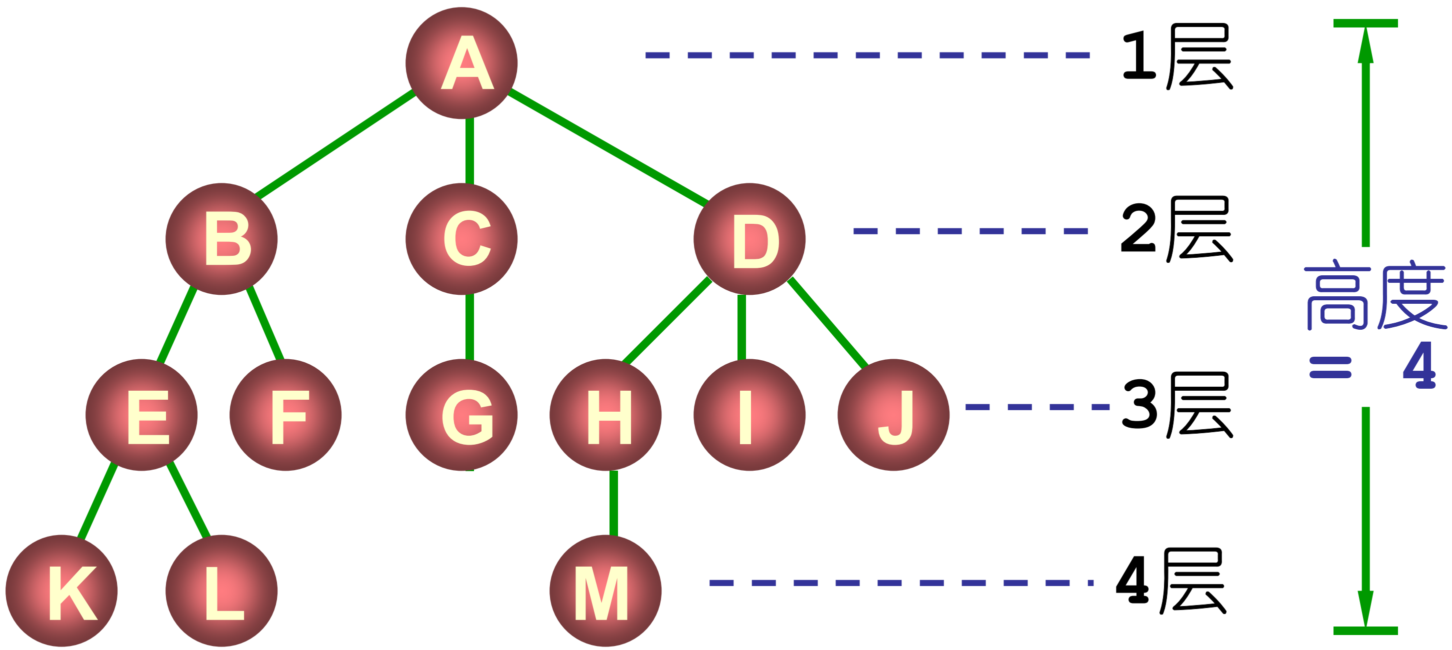
- 概念的理解




树根
树根
树根
B
C
D
E
F
G
H
I
J
K
L
M
子树
子树
子树

树的类型定义

- 树的特点
  - 每棵子树的根结点有且仅有一个直接前驱，但可以有0个或多个直接后继



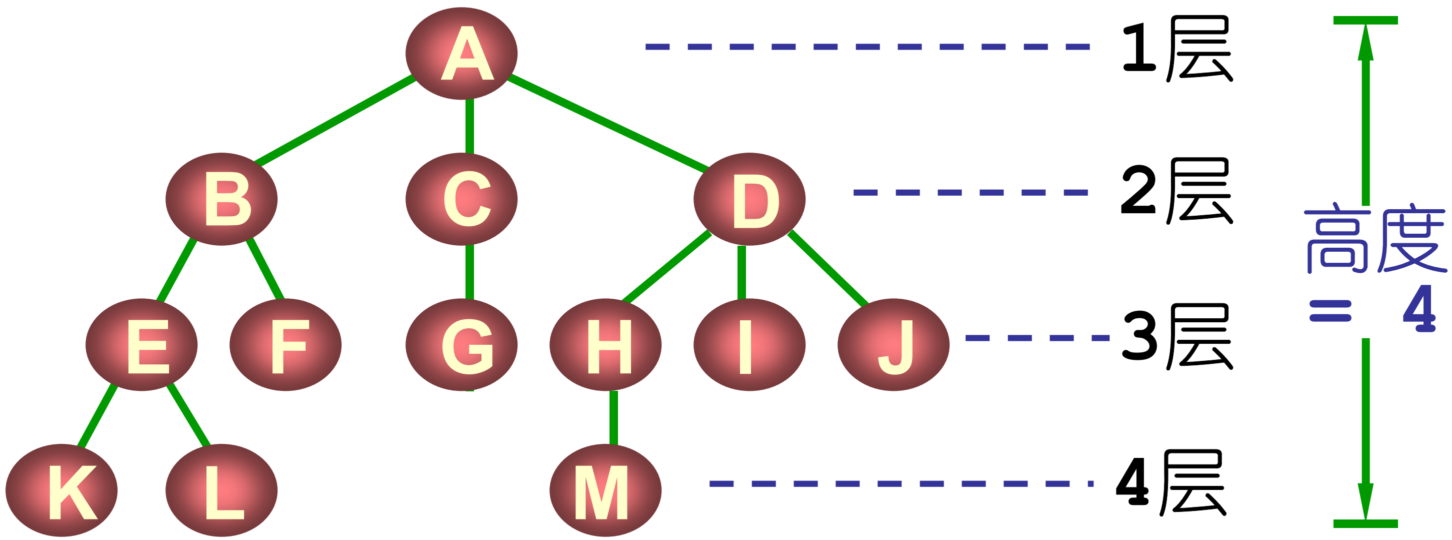
树的类型定义

- 树和线性结构的对比
  - 线性结构：一对一
  - 树结构：一对多

线性结构	树结构
第一个元素（无前驱）	根结点（无前驱）
最后一个元素（无后继）	多个叶子结点（无后继）
其它数据元素（一个前驱、一个后继）	树中的其它结点（一个前驱、多个后继）

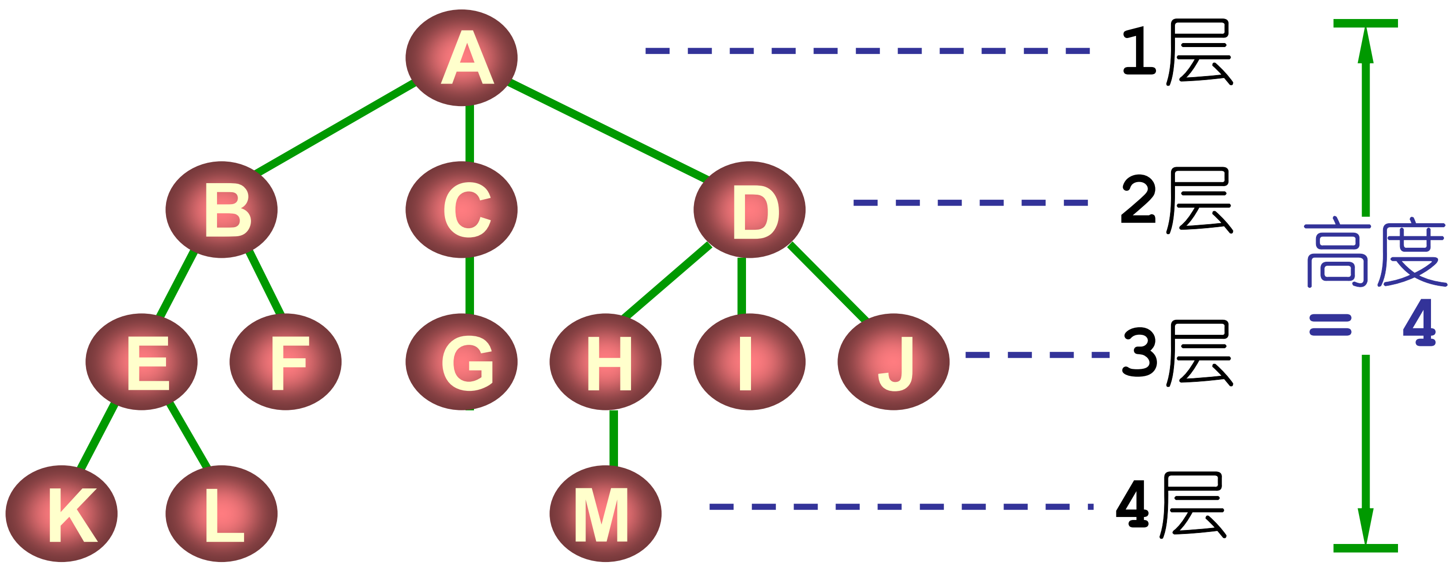
树的类型定义

- 基本概念
  - 结点的度：子树的个数
  - 树的度：结点的度的最大值
  - 分支结点：度不为0的结点
  - 叶结点：度为0的结点



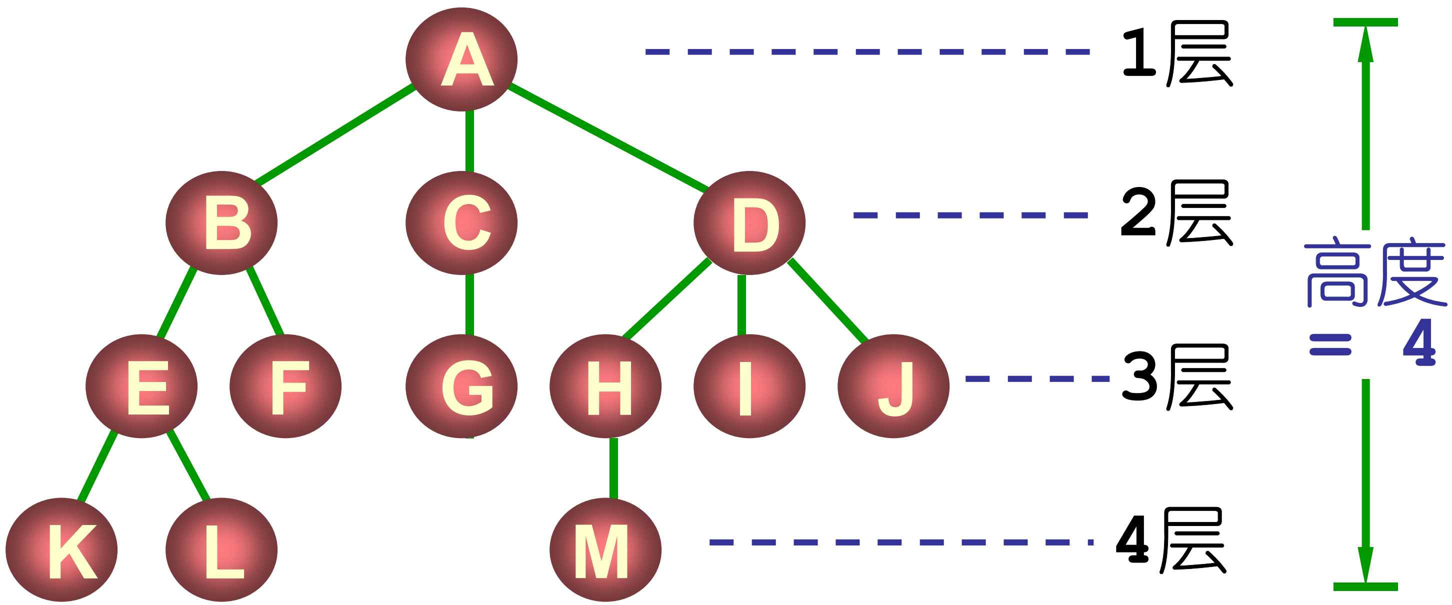
树的类型定义

- 孩子：某结点的子树的根
- 双亲：该结点称为孩子的双亲（不妨记成父亲）
- 兄弟：同一个双亲的孩子之间互为兄弟
- 祖先：从根到该结点所经分支的所有结点
- 子孙：以某结点为根的子树中的任一结点



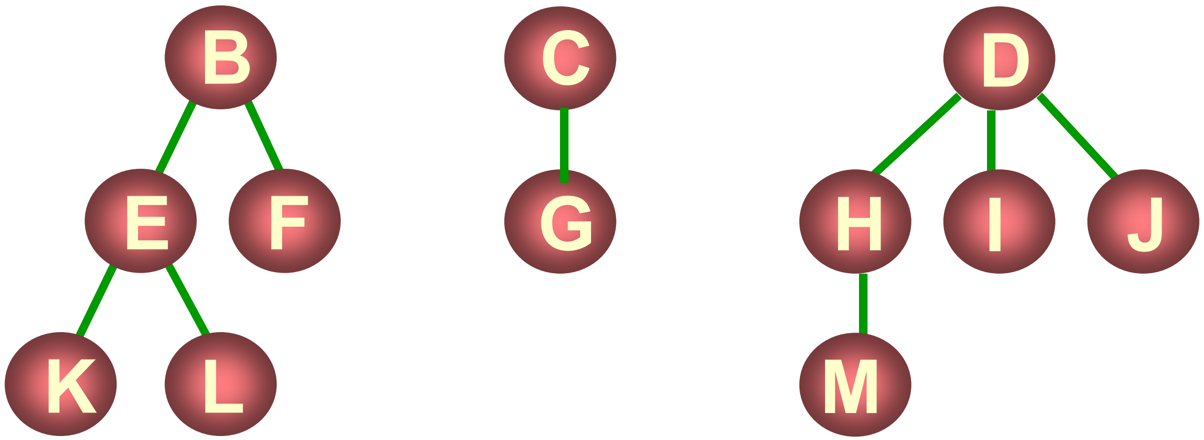
树的类型定义

- 结点的层次：根为第一层，孩子为第二层...
- 堂兄弟：双亲在同一层的结点互为堂兄弟
- 树的深度（高度）：树中结点的最大层次



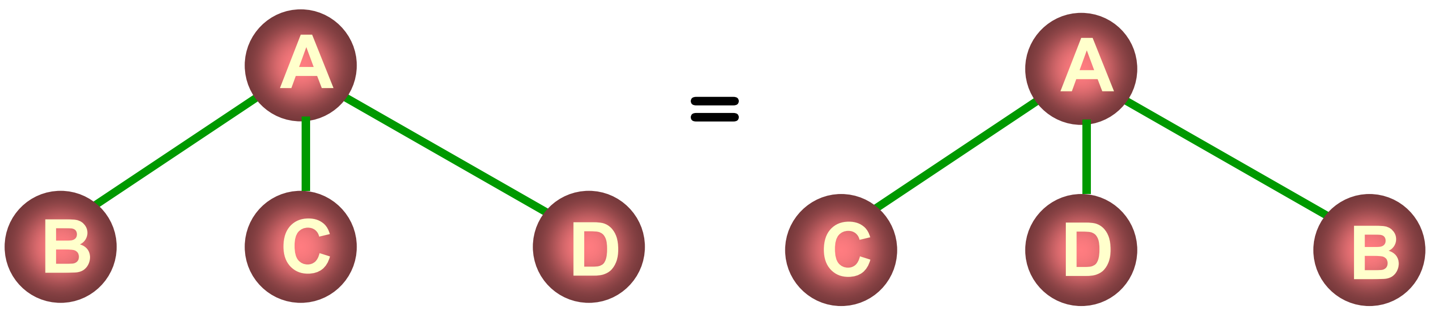
树的类型定义

– 森林： **m(m>=0)** 棵互不相交的树的集合



树的类型定义

- 有序树
  - 子树有次序之分
- 无序树
  - 子树无次序之分



树的类型定义： ADT Tree

- 数据关系： 父子关系
- 基本操作：

```
bool Init(&T) //初始化树
int size(T); //树中结点个数
bool isEmpty(T) ;//是否空树？
Node root(T) ; // 返回树根结点
int Depth(T);  //查询树的深度
bool Destory(&T); //销毁树
void Clear(&T); //清空树
void Create(&T,definition);//创建树
```

树的类型定义： ADT Tree

- 基本操作：

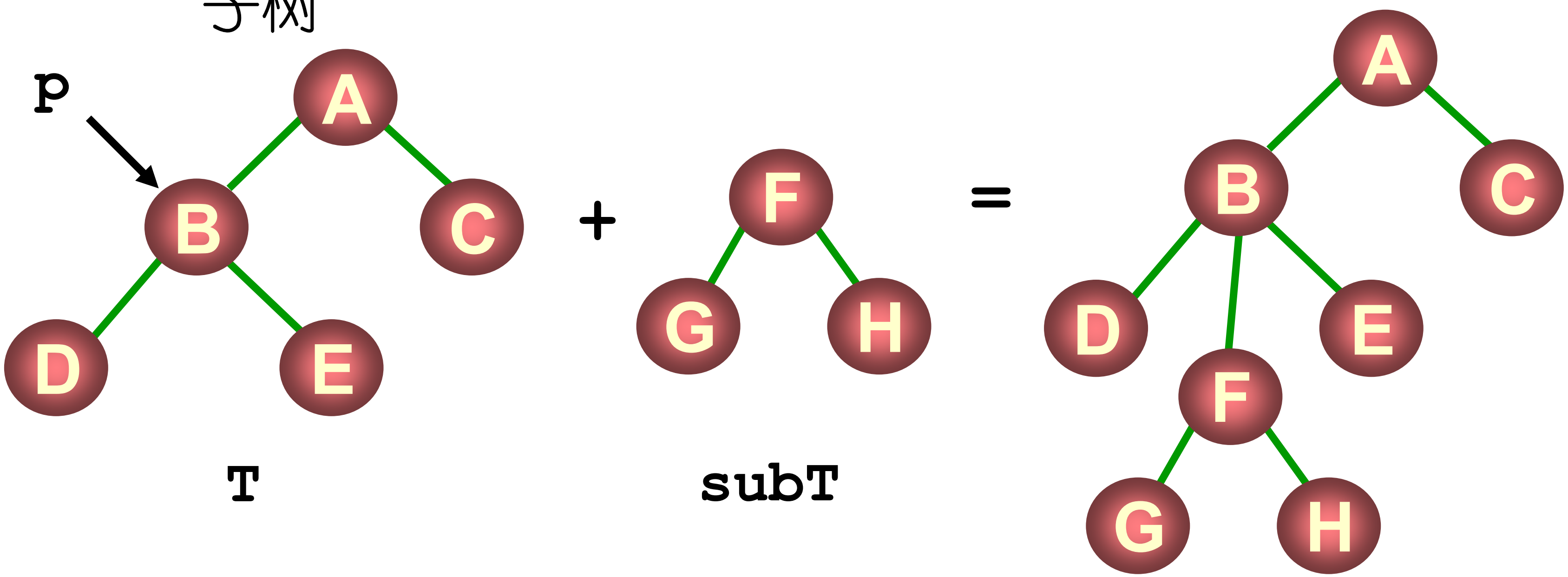
```
Node Find(T, condition); //查询
Node Parent(T, Node p); //查询双亲
Node[] Children(T, Node p); //查询子女
bool Insert(T,&p,i,subTree)   //插入子树
bool Delete(T,&p,i)   //删除p的第i个子树
void Traverse(T, visit()) //遍历
```

visit()做什么用？

1）打印 2）统计 3)…

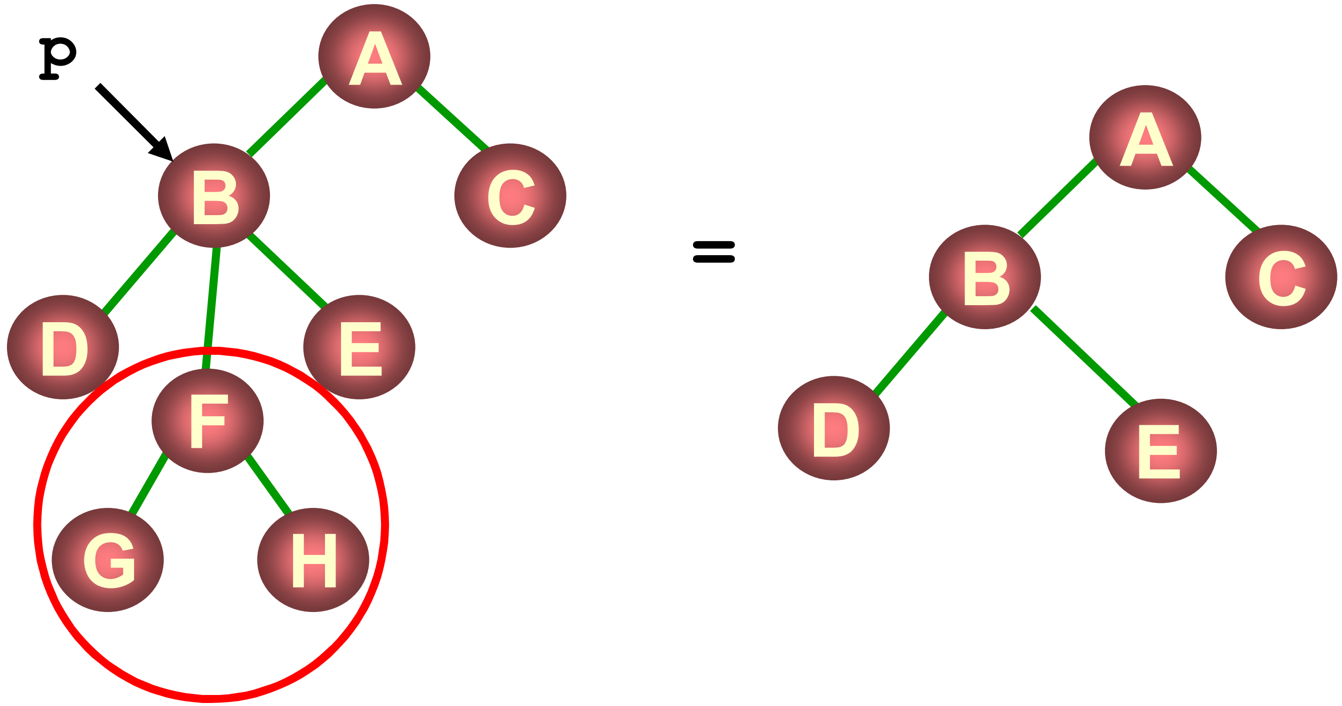
树的类型定义

– **InsertChild(&T, &p, i, subT)**
  - 初始条件：树**T**存在，**p**指向**T**中某个结点，**1≤i≤p**所指向结点的度**+1**，非空树**subT**与**T**不相交
  - 操作结果：**subT**插入**T**中，作为**p**所指结点的第**i**棵子树



树的类型定义

– **DeleteChild(&T, &p, i)**
  - 初始条件：树 **T** 存在，**p** 指向 **T** 中某个结点，**1≤i≤p**所指向结点的度
  - 操作结果：删除 **T** 中 **p** 所指结点的第 **i** 棵子树

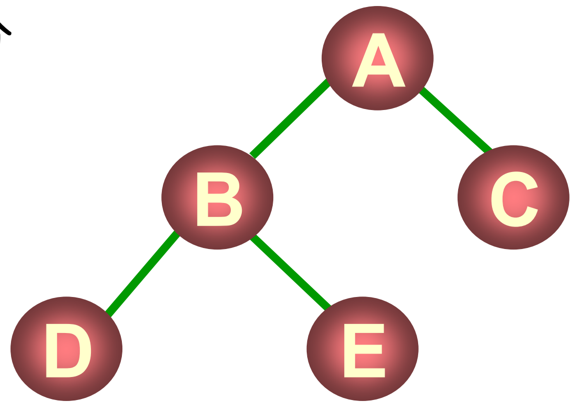


树的类型定义

- 本节小结
  - 树的定义：递归定义
  - 树的各种术语：建议和家族树类比记忆
  - 树的类型定义：理解即可

二叉树

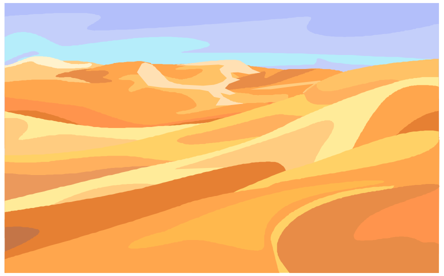
- 为什么要引入二叉树？
  - 树太一般，子树的个数无限制，表示困难
  - 事实上很多问题最多只需要**2**个子树
- 二叉树
  - 树的一种
  - 每个结点最多有**2**棵子树（即度**<=2**）
  - 子树有左右之分



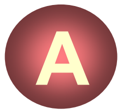
二叉树的类型定义

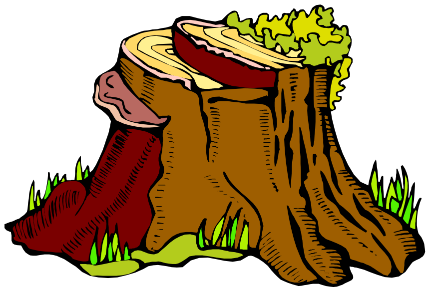
- 二叉树的五种基本形态
  - 空树

    Φ

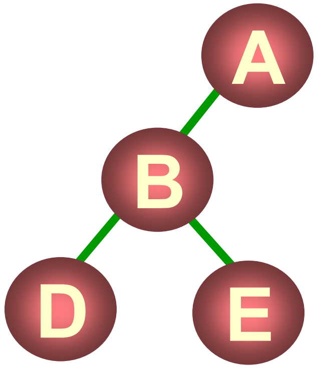


  - 只有树根

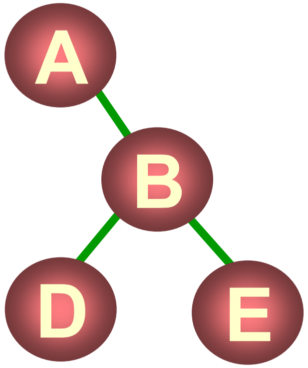


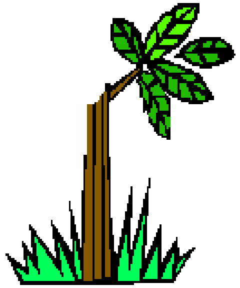


- 只有左子树

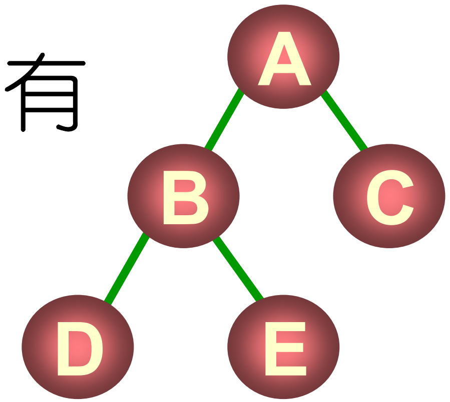


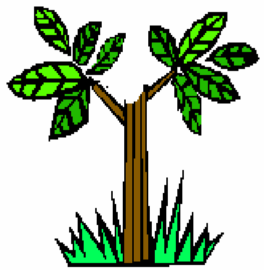
- 只有右子树





- 左右子树都有





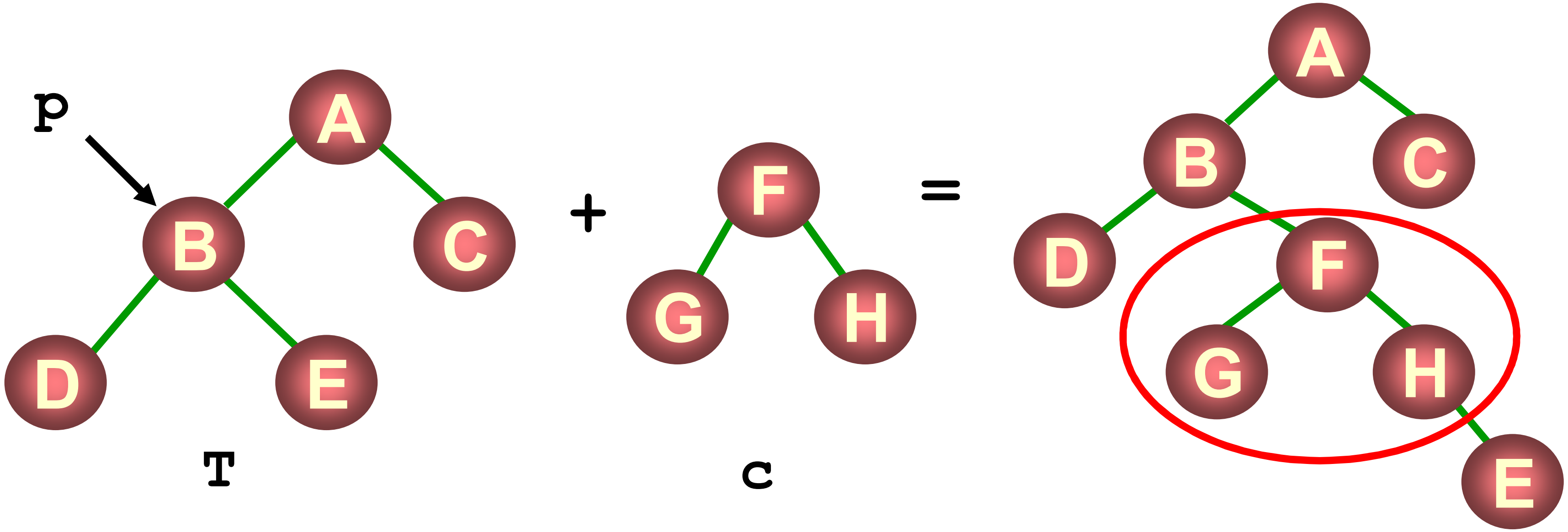
二叉树的类型定义 :ADT BiTree

- 数据关系：父子关系
  - 有一个称为“根结点”的结点没有双亲
  - 每个结点最多两个孩子结点
- 基本操作：除一般树的方法外，还有下列方法

```
Node leftChild(T,p); //返回结点p的左孩子
Node rightChild(T,p); //返回结点p的右孩子
bool hasLeft(T,p); //结点p有左孩子吗？
bool hasRight(T,p); //结点p有右孩子吗？
```

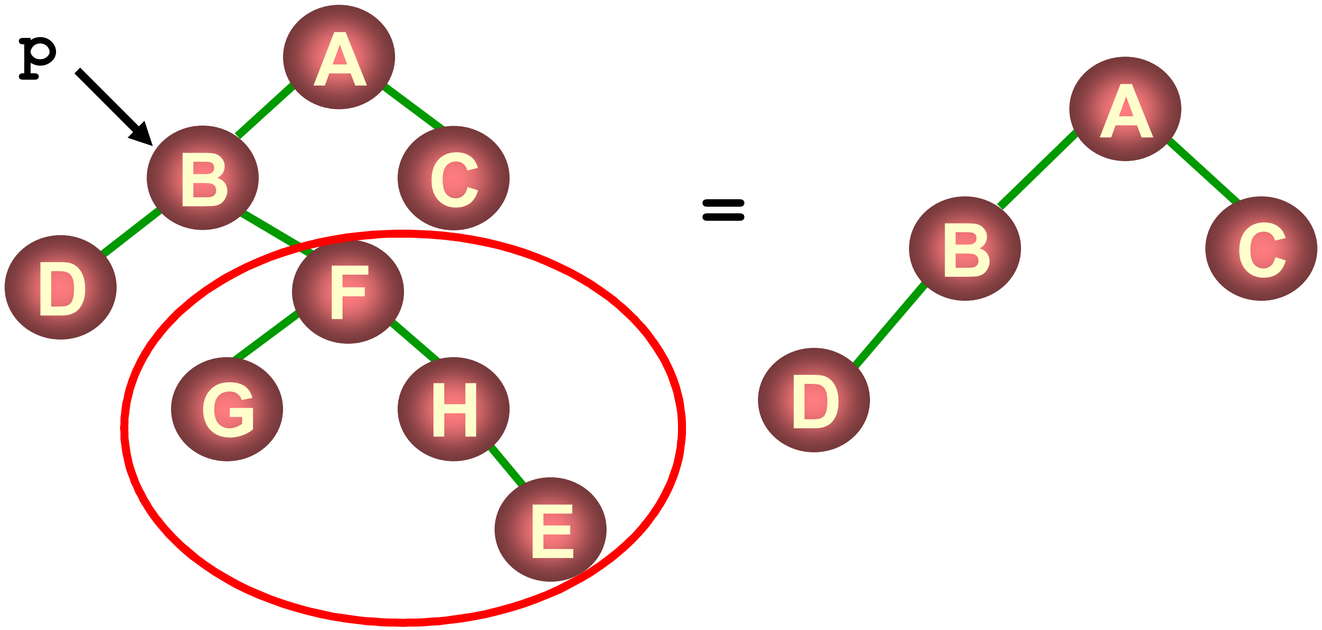
二叉树的类型定义

– **InsertChild(&T, &p, LR, subT)**
  - 若**LR=0/1**，插入**subT**为**T**中**p**所指结点的左/右子树.



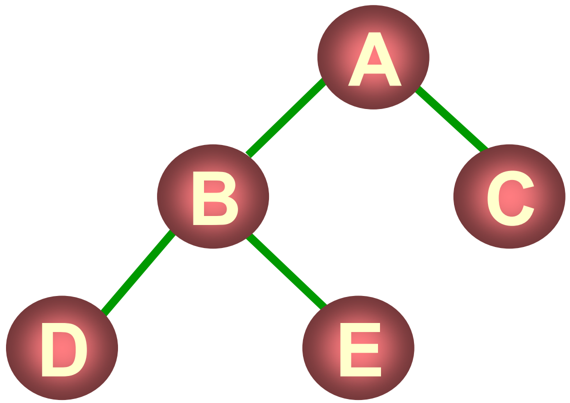
二叉树的类型定义

– **DeleteChild(&T, &p, LR)**
  - 若**LR=0/1**，则删除**T**中**p**所指结点的左/右子树



二叉树的类型定义

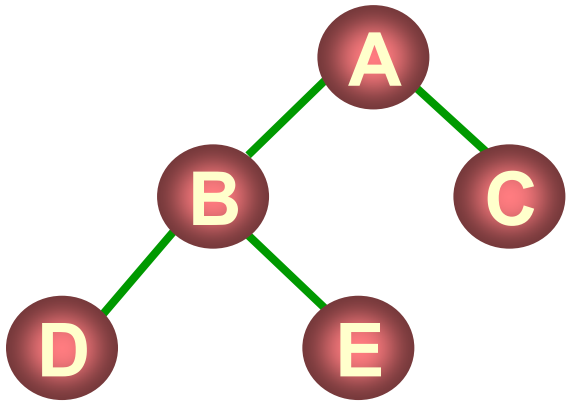
- LevelOrderTraverse(T, visit())
  - 依层序遍历次序对二叉树 **T** 的每个数据元素调用函数**visit**进行访问



A

二叉树的类型定义

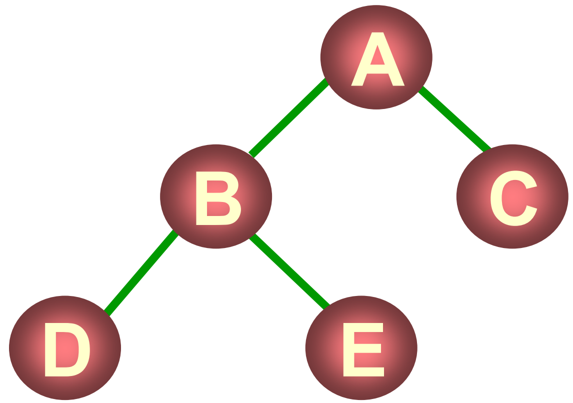
- LevelOrderTraverse(T, visit())
  - 依层序遍历次序对二叉树 **T** 的每个数据元素调用函数**visit**进行访问



A　B　C

二叉树的类型定义

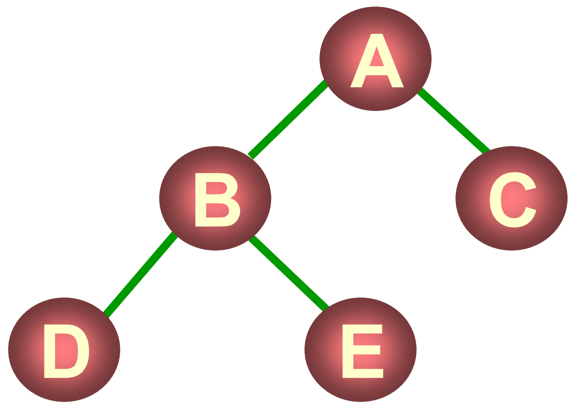
- **LevelOrderTraverse(T, visit())**
  - 依层序遍历次序对二叉树 **T** 的每个数据元素调用函数**visit**进行访问



A B C D E

二叉树的类型定义

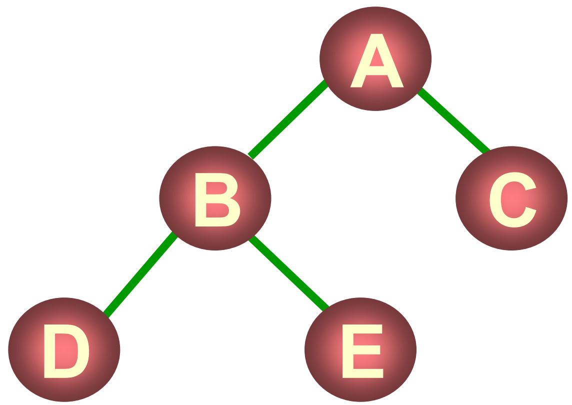
- **PreOrderTraverse(T, visit())**
  - 依先序遍历次序对二叉树**T**的每个数据元素调用函数**visit**进行访问



A B D E C

二叉树的类型定义

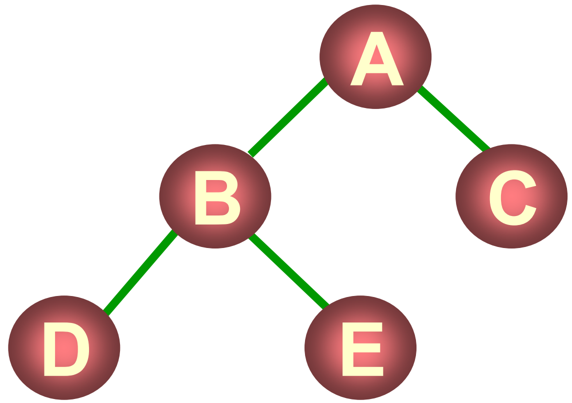
- **InOrderTraverse(T, visit())**
  - 依中序遍历次序对二叉树**T**的每个数据元素调用函数**visit**进行访问



D B E A C

二叉树的类型定义

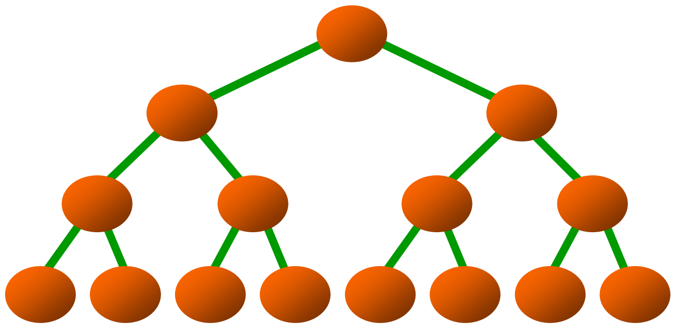
- PostOrderTraverse(T, visit())
  - 依后序遍历次序对二叉树T的每个数据元素调用函数visit进行访问



D E B C A

二叉树的性质

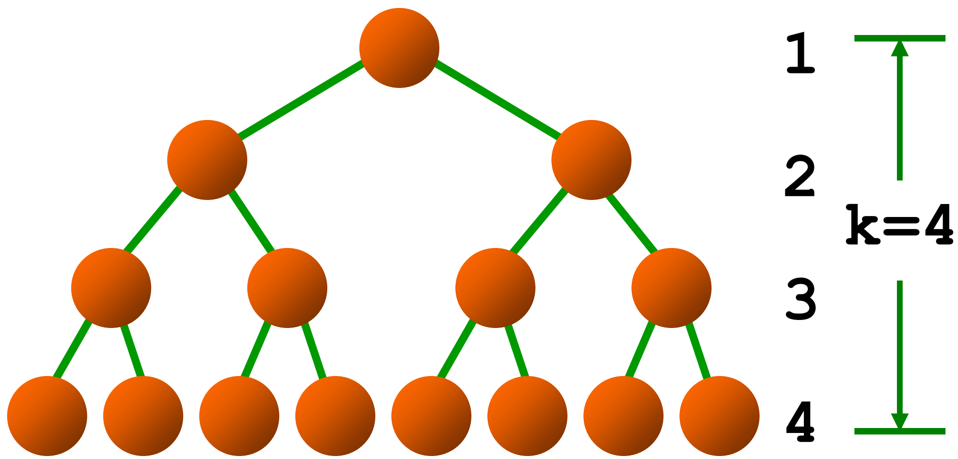
- 性质1：
  - 在二叉树的第i层最多有2^{i-1}个结点（i>=1）
  - 证明：
    - 当i=1时，显然成立
    - 假设当i=k时，也成立，即第k层最多2^{k-1}个结点
    - 当i=k+1时，由于二叉树的每个结点最多有2个孩子，所以第k+1层最多有$2*2^{k-1}=2^{(k+1)-1}$个结点
    - 所以对于任意i(i>=1)，二叉树的第i层最多有2^{i-1}个结点



二叉树的性质

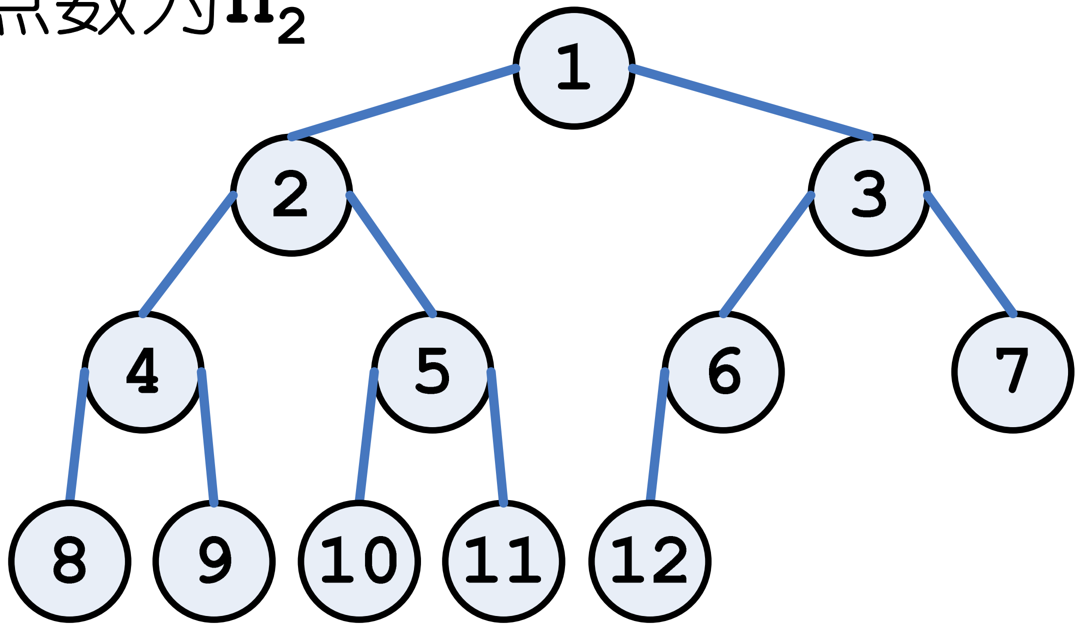
- 性质2：
  - 深度为k的二叉树最多有2^k-1个结点（k>=1）
  - 证明：
    - 由性质1可知：第i层最多有2^{i-1}个结点
    - 所以总的结点数最多为

$$\boxed{\sum_{i=1}^{k} 2^{i-1} = 2^k - 1}$$



二叉树的性质

- 性质3：
  - 对任何一棵二叉树**T**,
  - 若叶结点数(即度为0的结点数)为$\mathbf{n_0}$
  - 度为2的结点数为$\mathbf{n_2}$
  - 则$\boldsymbol{n_0=n_2+1}$



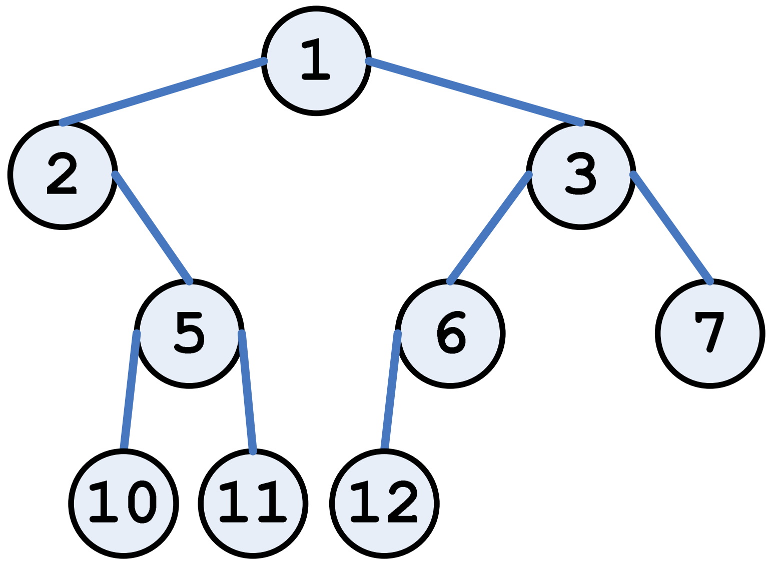
二叉树的性质

– 证明：

- 总结点数$n=n_0+n_1+n_2$
- 设分支数为B，则$n=B+1$
- 又$B=n_1+2n_2$
- 解方程组：

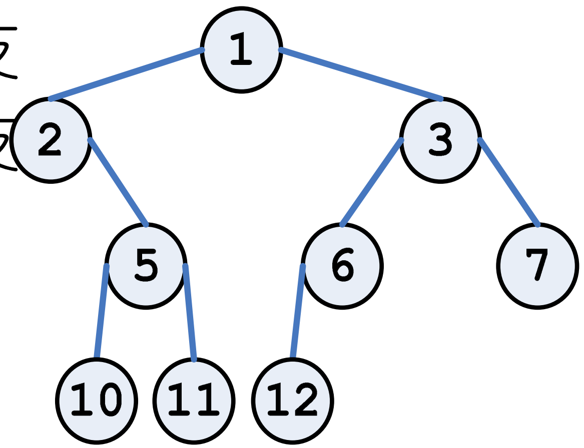
$$\begin{cases} n = n_0 + n_1 + n_2 \\ n = B + 1 \\ B = n_1 + 2n_2 \end{cases}$$

得：$n_0=n_2+1$



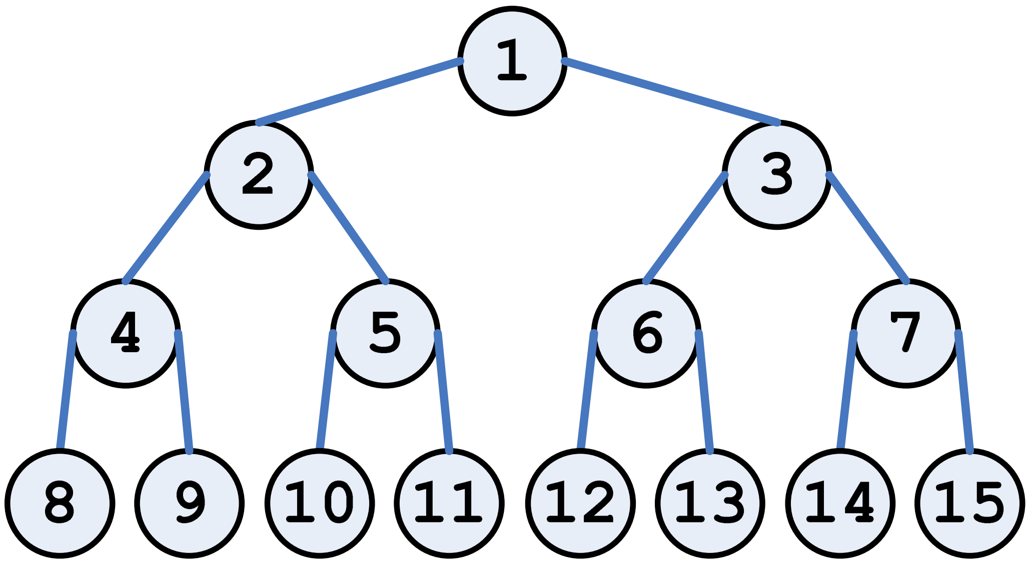
– 理解

- 方程1： $n=n_0+n_1+n_2$
  - 结点无外乎度为0、1、2三种情况
- 方程2： $n=B+1$
  - “五个手指四个叉”
  - 除了树根，其余每个结点“上方”都有一个分支
- 方程3： $B=n_1+2n_2$
  - 度为2的结点“下方”有2个分支
  - 度为1的结点“下方”有1个分支
  - 度为0的结点“下方”有0个分支



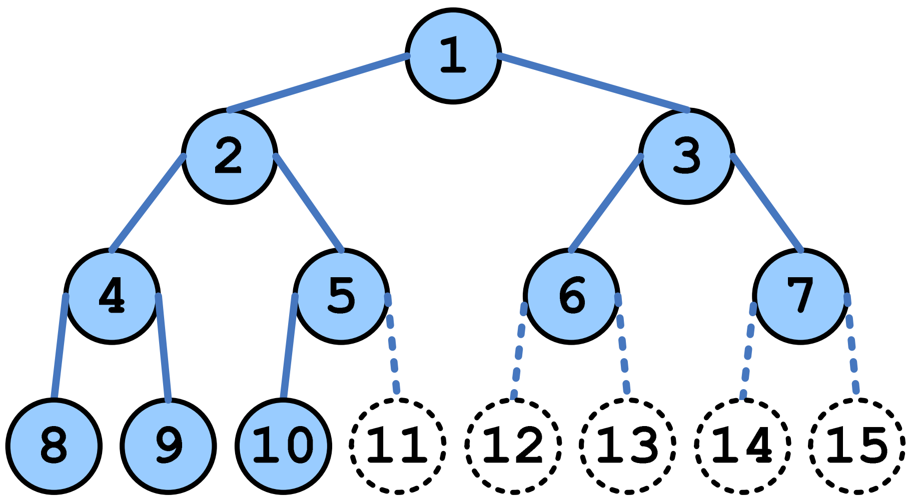
二叉树的性质

- 特殊形态的二叉树
  - 满二叉树（**Full Binary Tree**）
    - 深度为**k**，结点数为2^k-1
    - 即结点数达到最大值

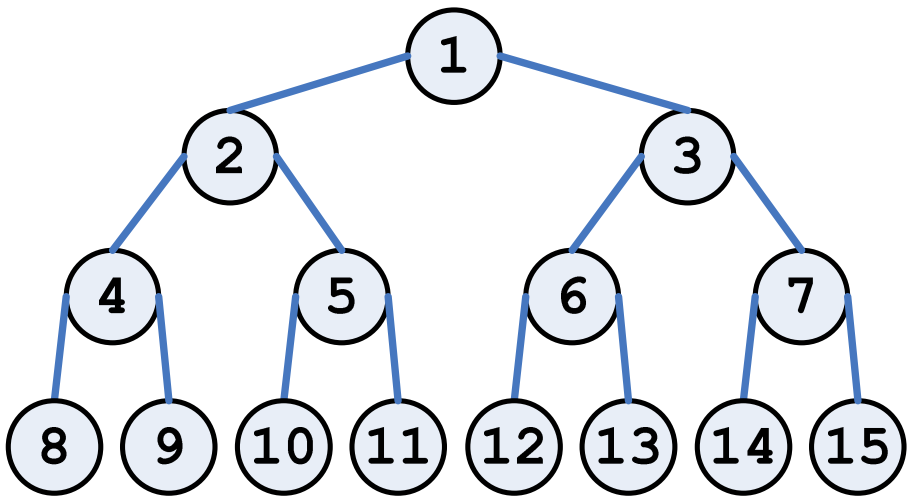


二叉树的性质

- 完全二叉树 (**Complete Binary Tree**)
  - 从上到下，从左到右对结点编号
  - 该树的每一个结点的编号都与一个同深度的满二叉树的结点一一对应



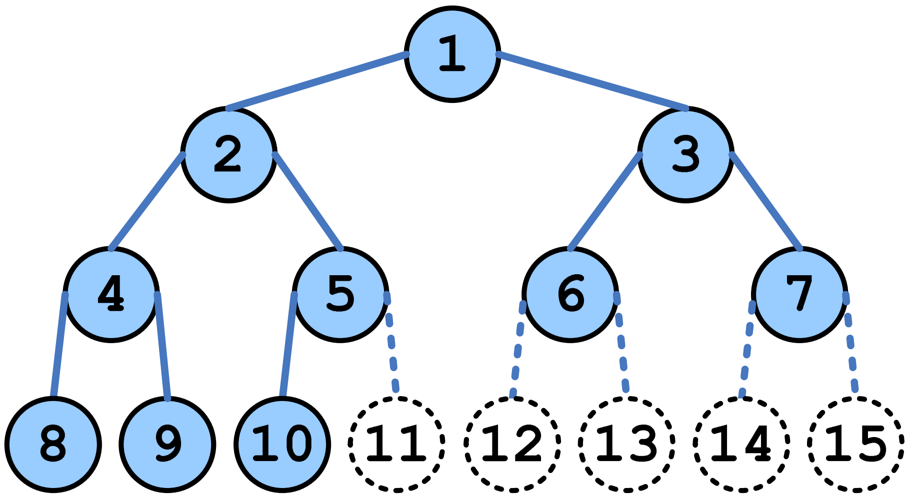
完全二叉树



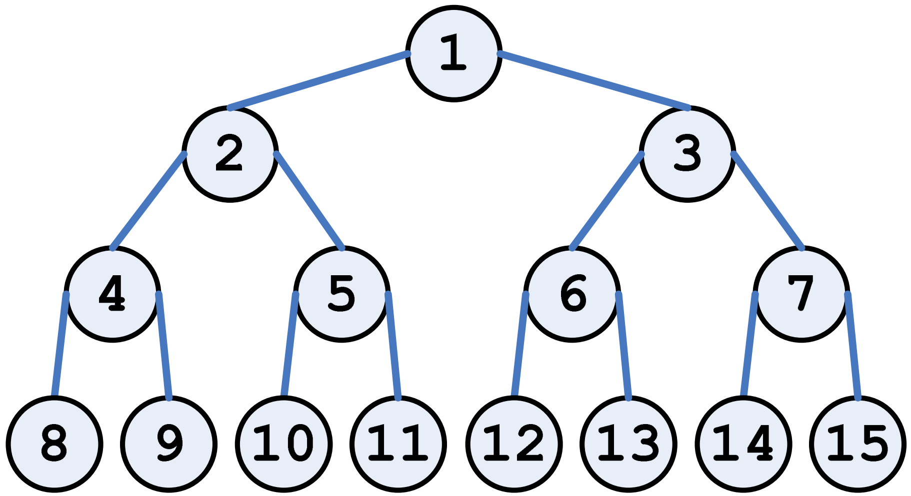
满二叉树

二叉树的性质

- 理解1：和满二叉树相比，就是最底层最右边连续缺少一些结点
- 理解2：结点是按照从上到下，从左到右的顺序一个一个地加到树上的



完全二叉树

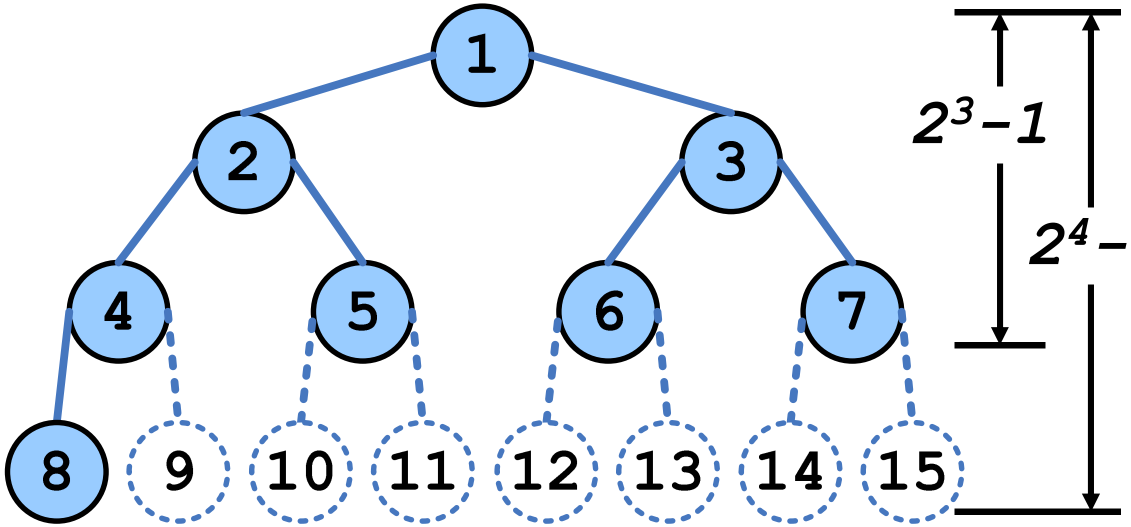


满二叉树

二叉树的性质

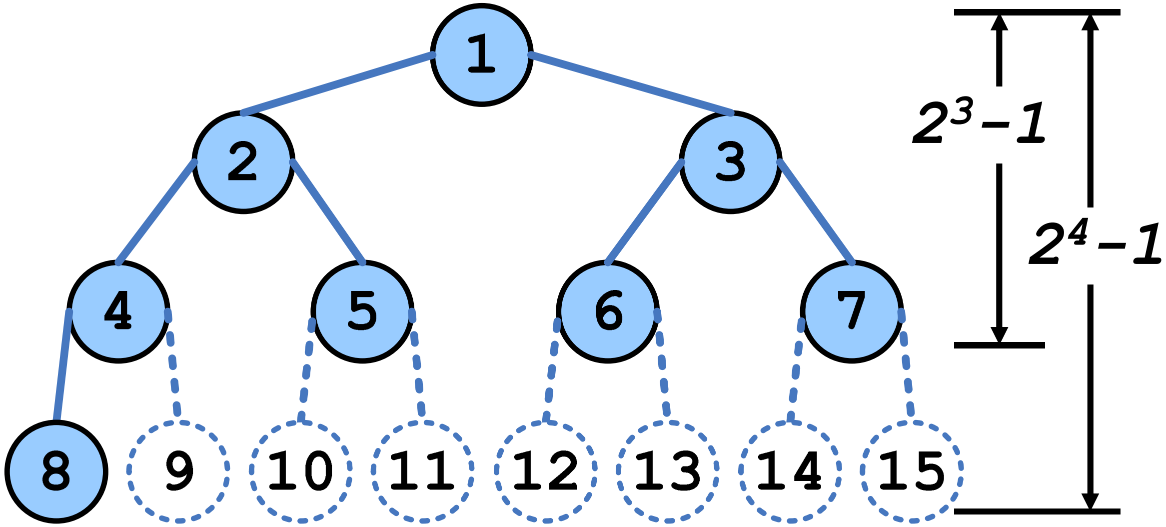
- 性质4：
  - 具有**n**个结点的完全二叉树的深度为

$$\lfloor \log_2 n \rfloor + 1$$



二叉树的性质

- 证明：
  - 设深度为**k**，则：$2^{k-1} <= n < 2^k$
  - 两边求对数：$k-1 <= \log_2 n < k$
  - 所以：$k = \lfloor \log_2 n \rfloor + 1$

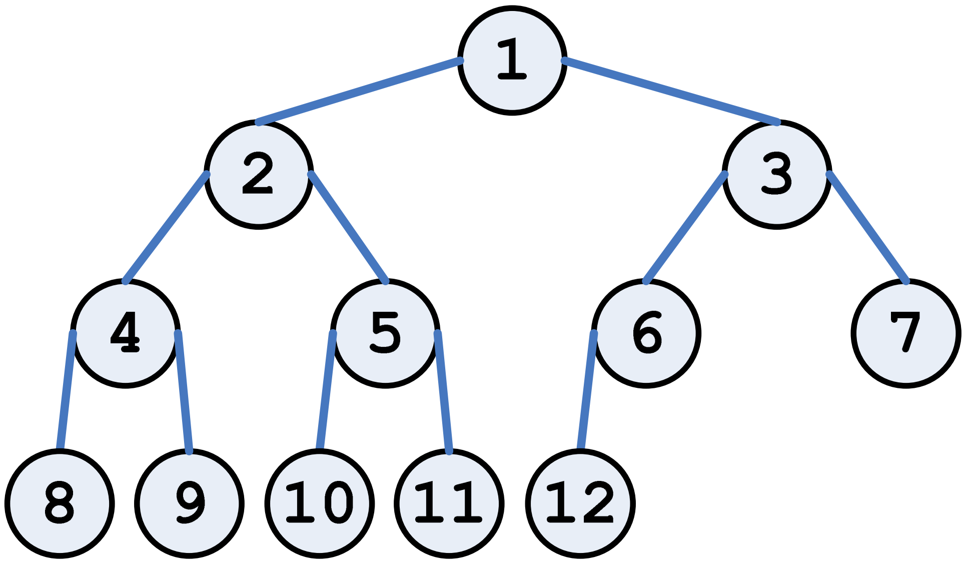


二叉树的性质

- 性质5：
  - 若将一棵有**n**个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号：

    (1) 若**i=1**，则结点**i**是树根，无双亲

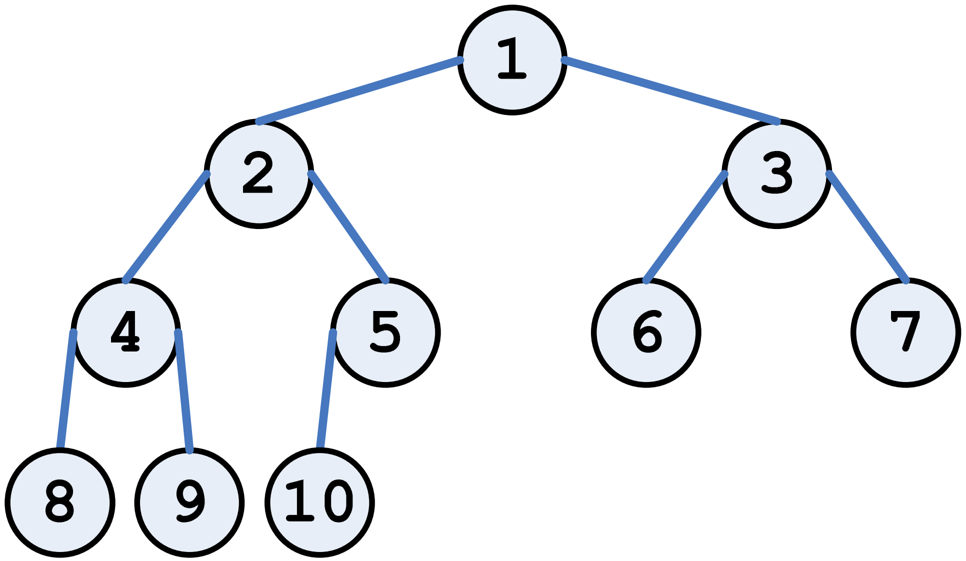
    若**i>1**，则其双亲是节点 $\lfloor i/2 \rfloor$



二叉树的性质

(2) 若2i>n，则结点i无左孩子（即i为叶结点）否则其左孩子为2i

- 理解：
  - 结点i如果有左孩子的话，其编号应该为2i
  - 如果2i>n，则左孩子不存在

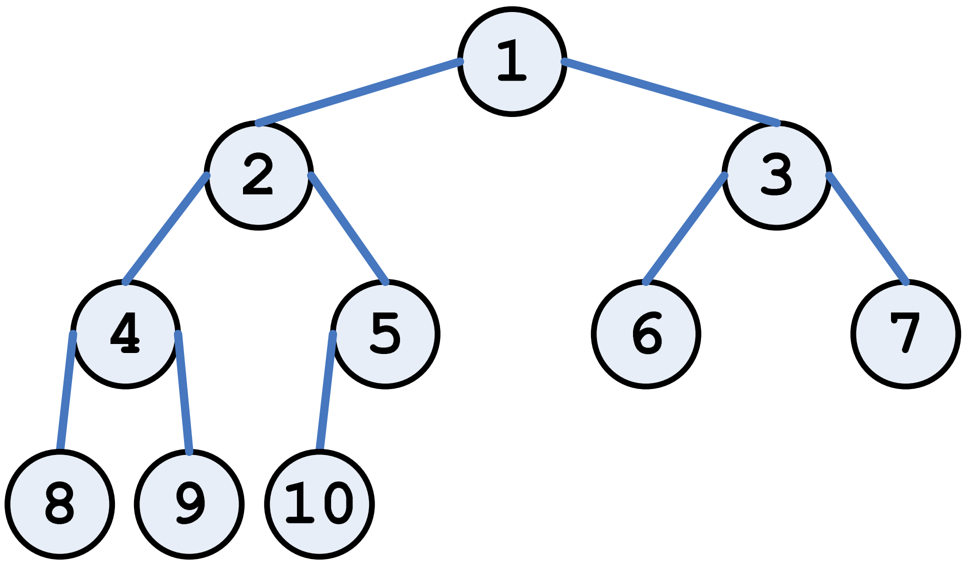


二叉树的性质

(3) 若 $2i+1>n$，则结点 i 无右孩子

否则其右孩子为 $2i+1$

由(2)(3)可以推导出(1)



二叉树

- 本节小结
  - 二叉树的概念和类型定义
    - 注意和树的类型定义的对比
  - 二叉树的性质
    - 要求自己能推导、应用、推广

二叉树的顺序表示

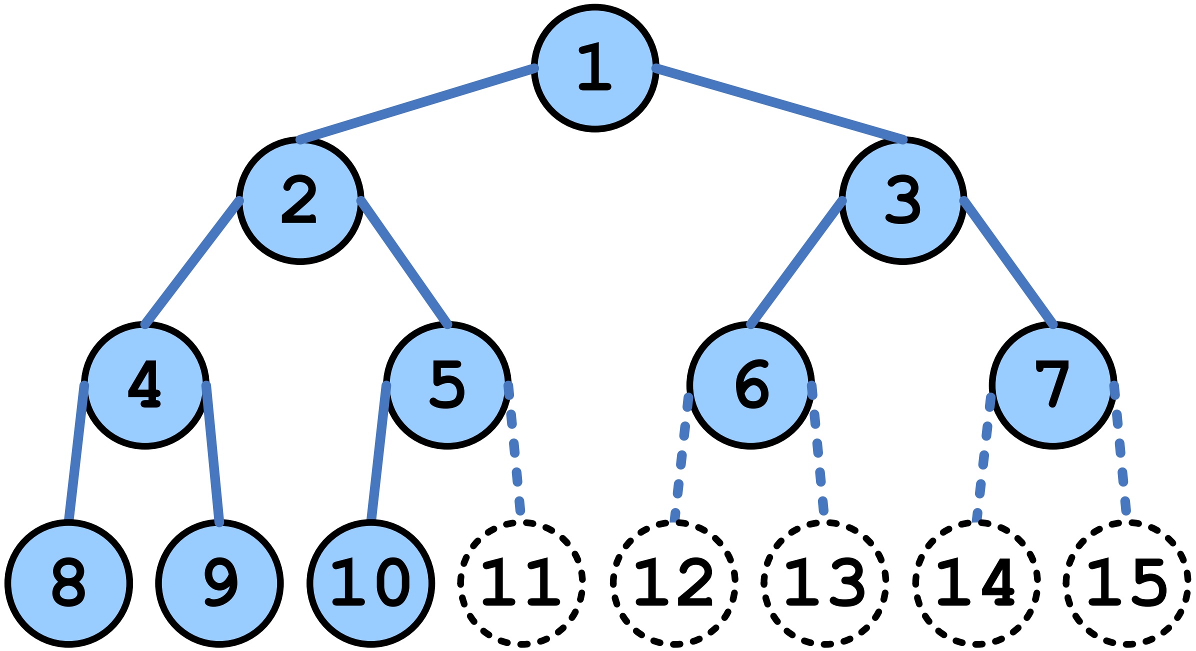
- 顺序表示
  - 用一维数组来表示

```
#define MAX_TREE_SIZE 100
typedef TElemType
            SqBiTree[MAX_TREE_SIZE];
SqBiTree bt;
```

  - 按照满二叉树的顺序存放

二叉树的顺序表示

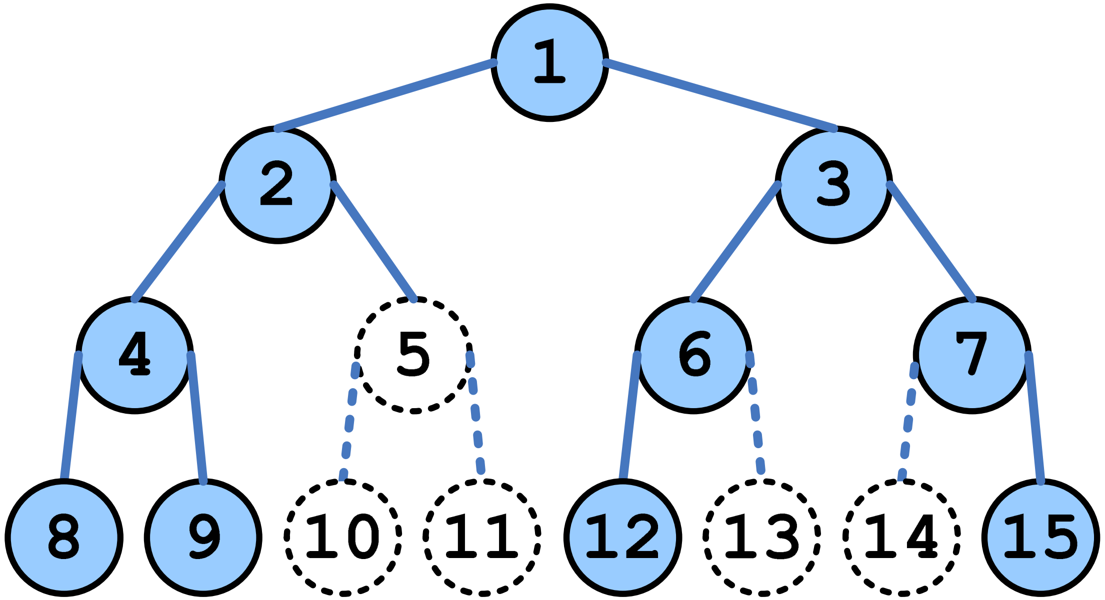
– 完全二叉树



1	2	3	4	5	6	7	8	9	10

二叉树的顺序表示

– 一般二叉树



1	2	3	4	0	6	7	8	9	0	0	12	0	0	15

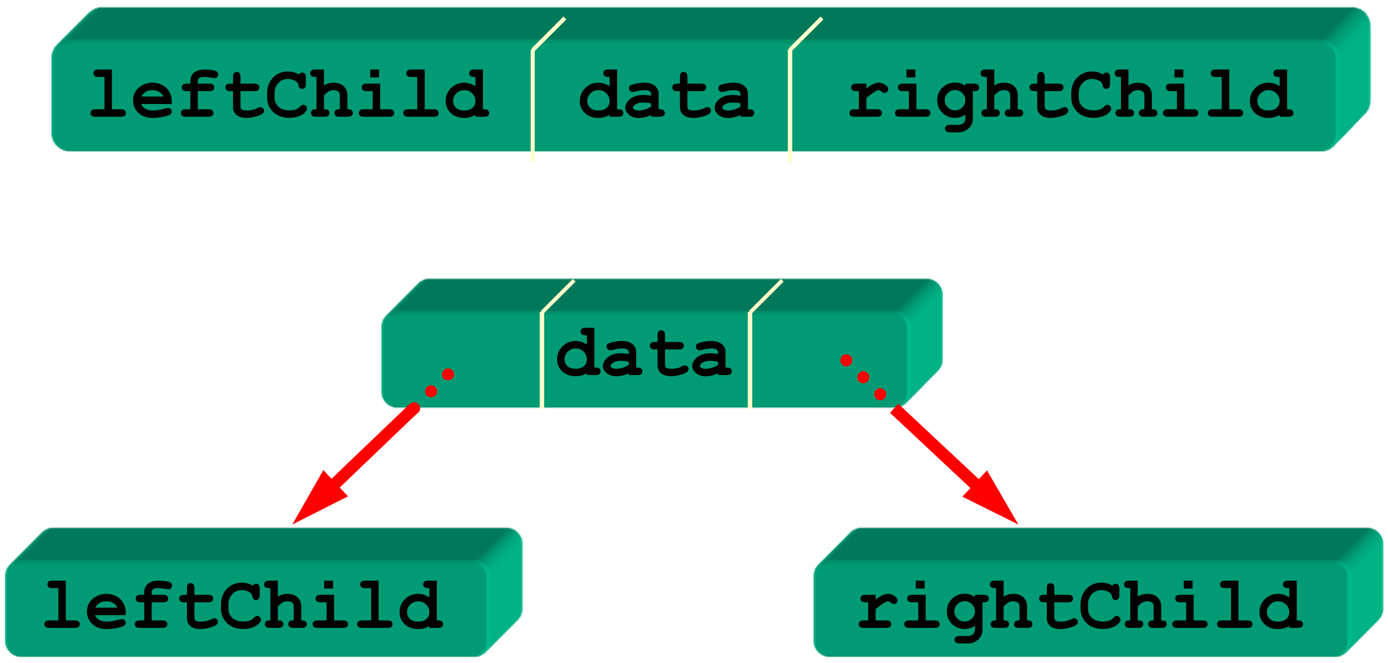
二叉树的顺序表示

– 极端情形：单支树
  - 深度为**k**的二叉树，最少只有**k**个结点
  - 却需要2^k-1个存储单元

1	0	3	0	0	0	7	0	0	0	0	0	0	0	15

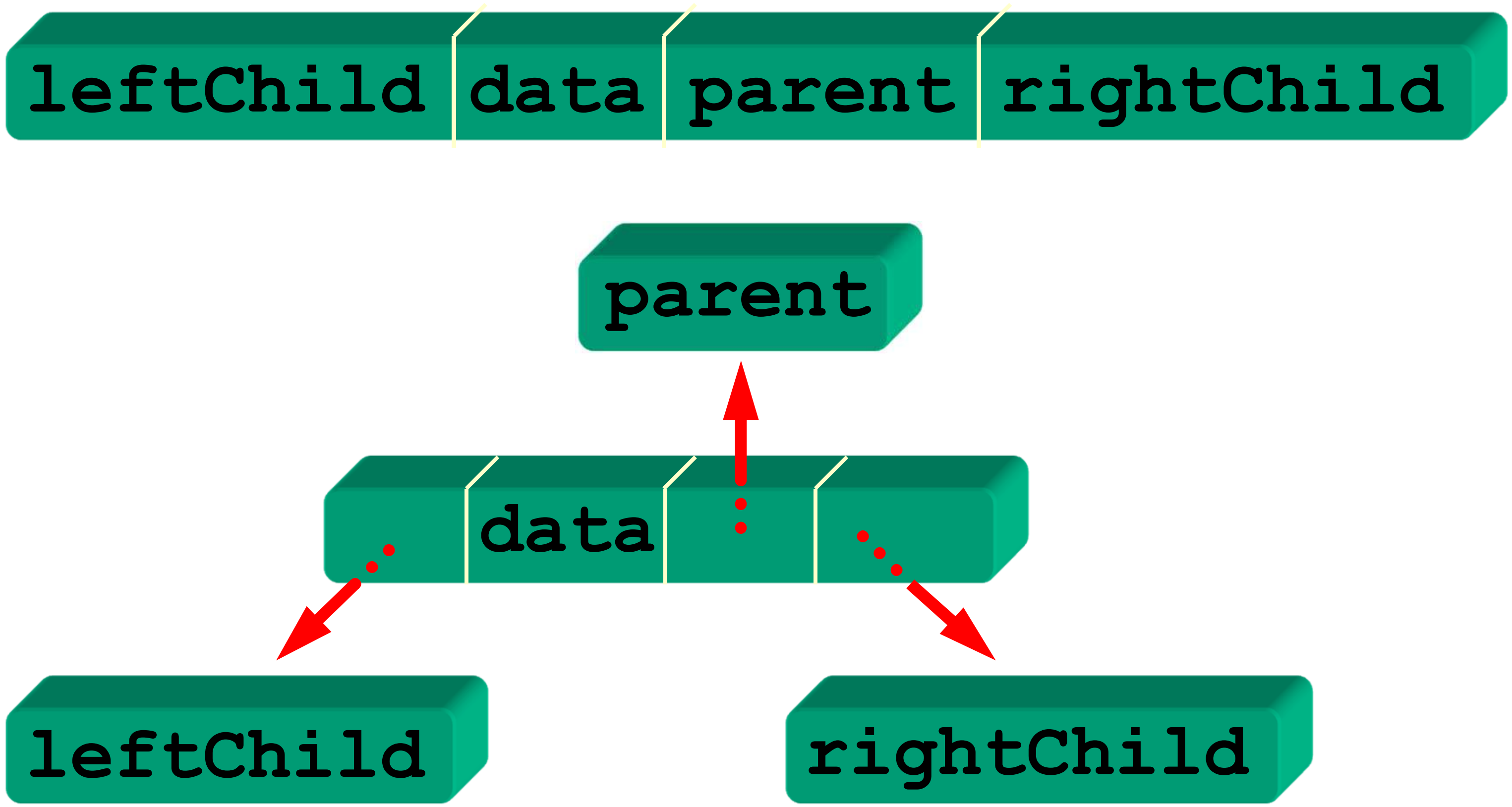
二叉树的链表表示

- 二叉链表



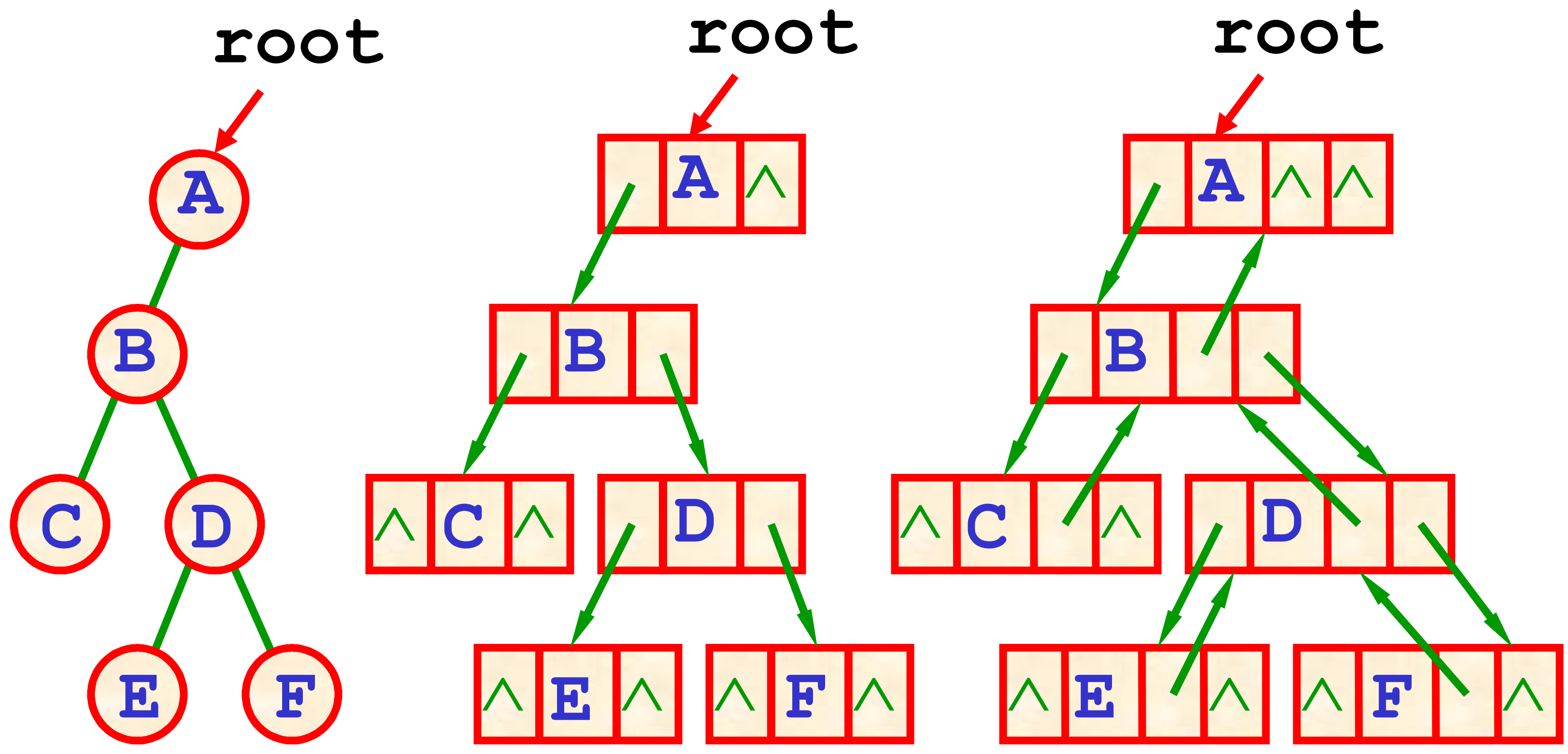
二叉树的链表表示

- 三叉链表



二叉树的链表表示

- 例如



二叉树　　二叉链表　　三叉链表

二叉树的链表表示

- 结构的定义

```
typedef struct _BiNode{
    TElemType data;
    struct _BiNode *lchild, *rchild;
}BiNode;
typedef BiNode* BiTree;
```

二叉树的存储结构

- 本节小结
  - 各种存储结构
    - 注意各自的优缺点
    - 比如顺序存储空间浪费大
    - 二叉链表不能直接找到父结点

二叉树的遍历

- 遍历
  - 按照某种搜索路径访问每个单元，且每个单元仅被访问一次
- 二叉树的遍历
  - 深度优先遍历
    - 先（前）序遍历
    - 中序遍历
    - 后序遍历
  - 广度优先遍历（层序）

二叉树的遍历：先序遍历

- 先序遍历（**Preorder Traversal**）
  - 若二叉树为空，则空操作
  - 否则
    - 先访问根结点
    - 再先序遍历左子树
    - 最后先序遍历右子树

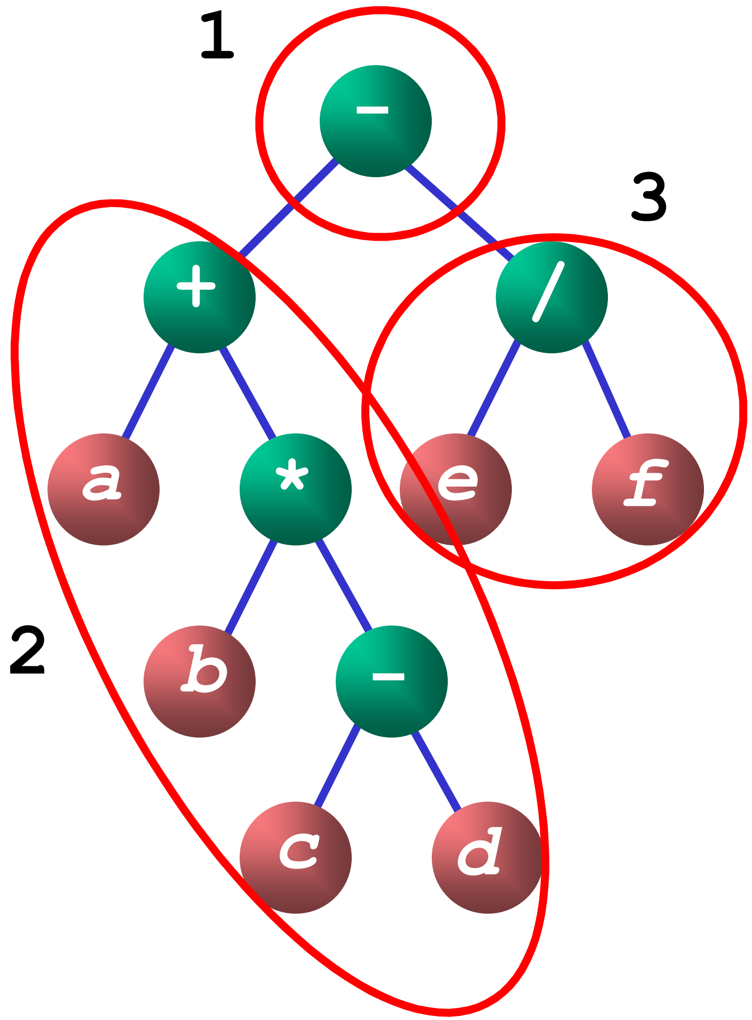
```
POT(Tree T){
   if(!T) return;
   visit(T的根);
   POT(T的左子树)
   POT(T的右子树)
}
```

二叉树的遍历：先序遍历

1 先访问树根 “-”

2 再访问 “-”的左子树

3 再访问 “-”的右子树



二叉树的遍历：先序遍历

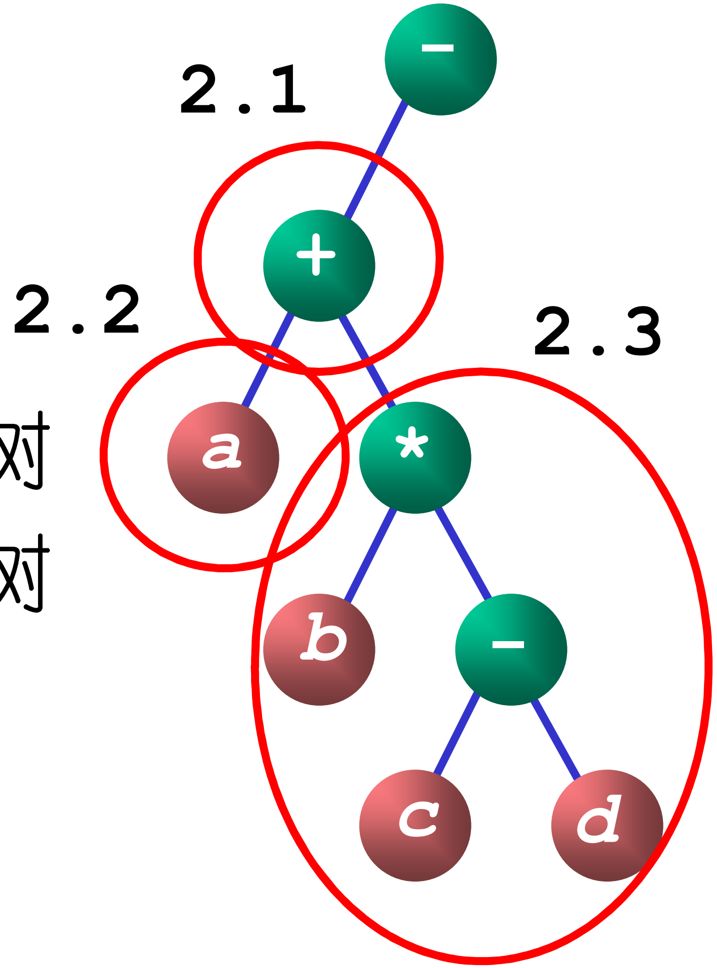
2 访问“-”的左子树

对于这棵左子树

2.1 先访问树根“+”

2.2 再访问“+”的左子树

2.3 再访问“+”的右子树



二叉树的遍历：先序遍历

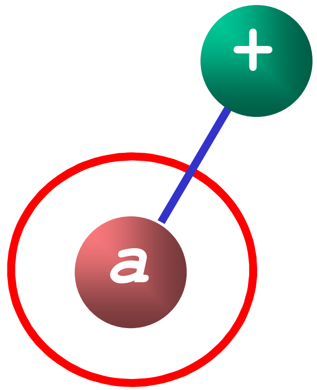
2.2 访问“**+**”的左子树

对于这棵左子树

2.2.1 先访问树根“**a**”

2.2.2 再访问“**a**”的左子树（空）

2.2.3 再访问“**a**”的右子树（空）



二叉树的遍历：先序遍历

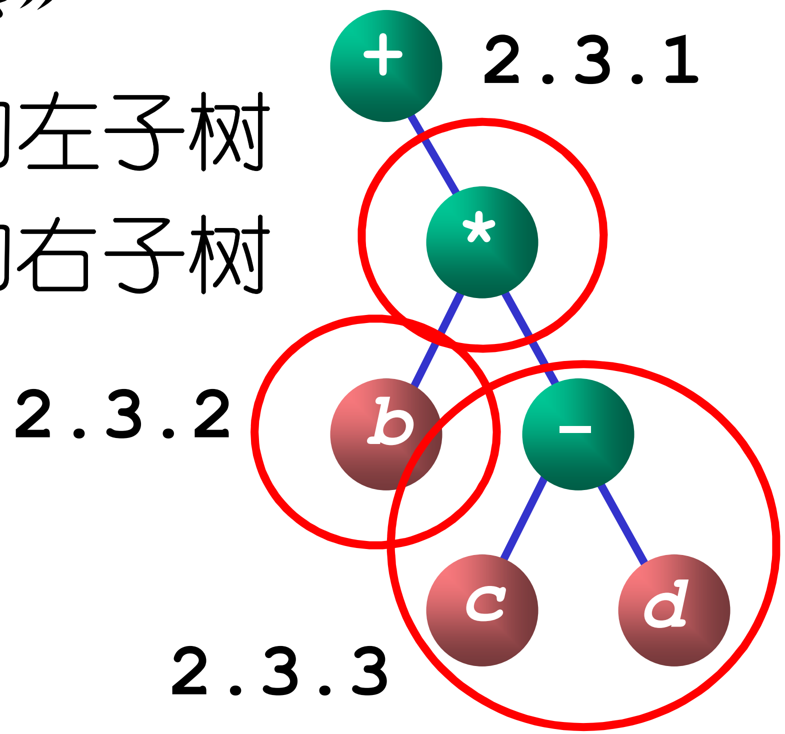
2.3 访问“+”的右子树

对于这棵右子树

2.3.1 先访问树根“*”

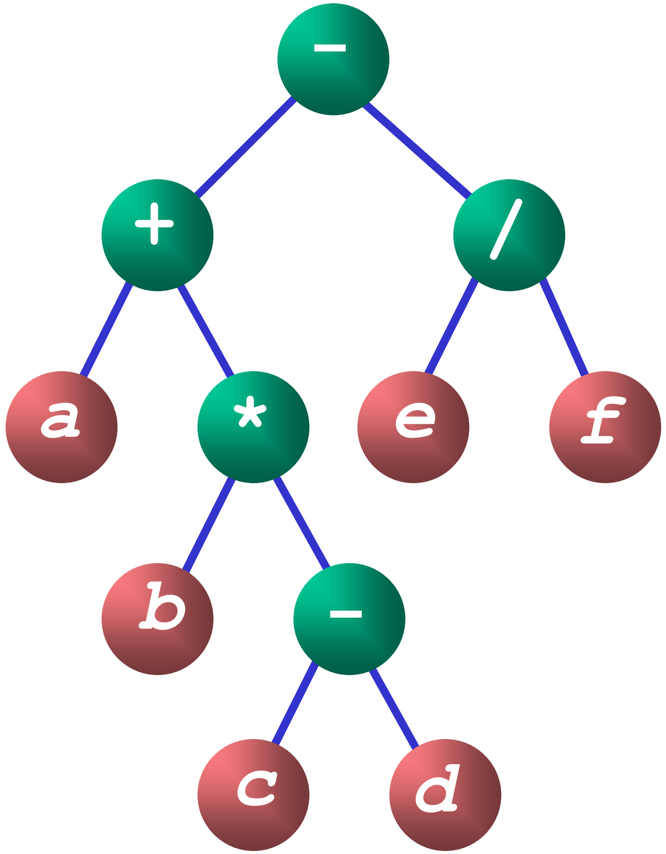
2.3.2 再访问“*”的左子树

2.3.3 再访问“*”的右子树



二叉树的遍历：先序遍历

- 理解
  - 如果是空树，直接结束
  - 如果树非空
    - 先访问树根
    - 把左子树看成跟原来地位相同的另一棵树，用同样的方法去遍历它
    - 左子树遍历完以后再同样的方法去遍历右子树
  - 右图的先序遍历结果为：
    - - + a * b - c d / e f



二叉树的遍历：先序遍历

- 理解
  - 如果是空树，直接结束
  - 如果树非空
    - 先访问树根
    - 把左子树看成跟原来地位相同的另一棵树，用同样的方法去遍历它
    - 左子树遍历完以后再同样的方法去遍历右子树

```
POT(Tree T){
   if(!T) return;
   visit(T的根);
   POT(T的左子树)
   POT(T的右子树)
}
```

二叉树的遍历：中序遍历

- 中序遍历（**Inorder Traversal**）
  - 若二叉树为空，则空操作
  - 否则
    - 先中序遍历左子树
    - 再访问根结点
    - 最后中序遍历右子树

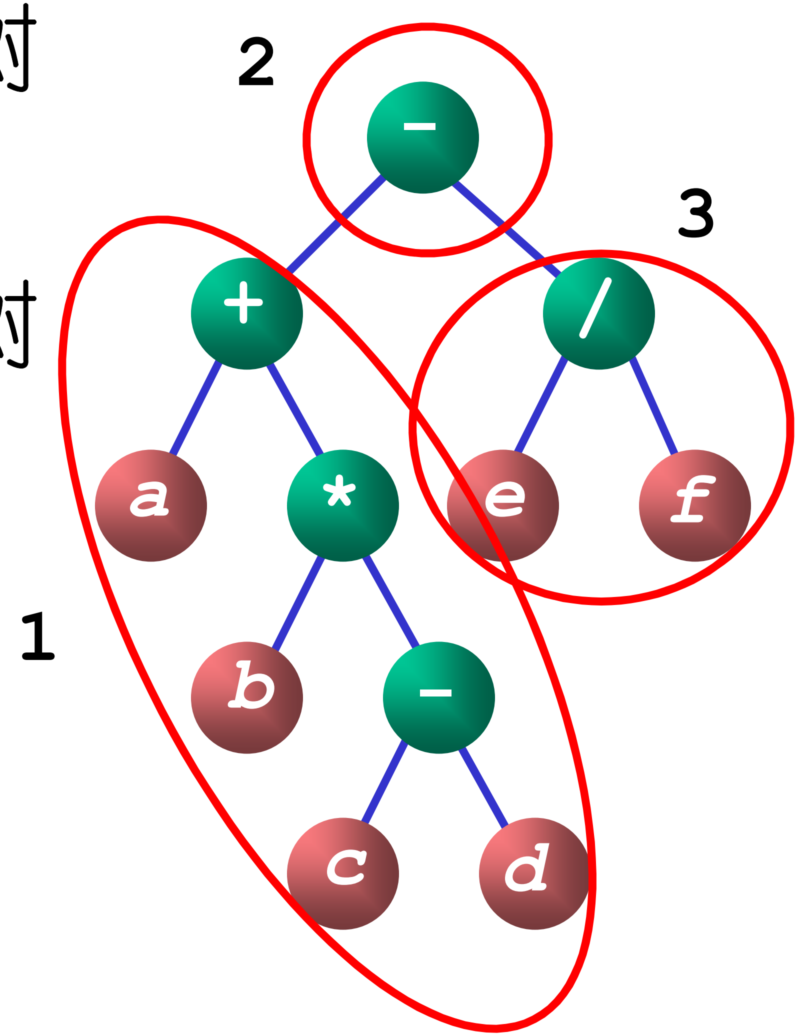
```
IOT(Tree T){
   if(!T) return;
   IOT(T的左子树)
   visit(T的根);
   IOT(T的右子树)
}
```

二叉树的遍历：中序遍历

1 先访问 “-”的左子树

2 再访问树根 “-”

3 再访问 “-”的右子树



二叉树的遍历：中序遍历

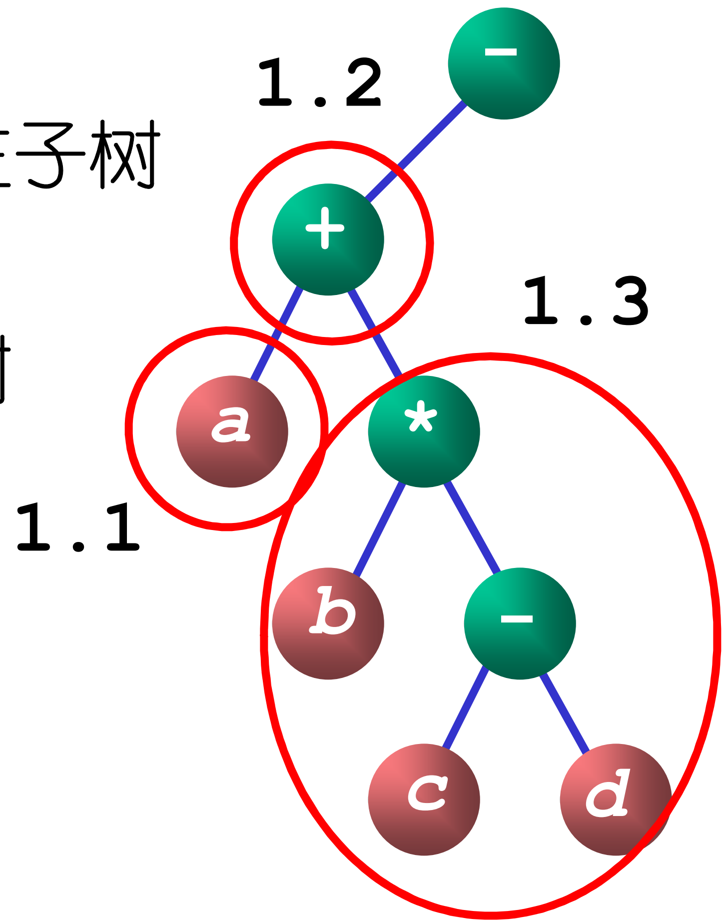
1 访问“-”的左子树

对于这棵左子树

1.1 先访问子树根“+”的左子树

1.2 再访问子树根“+”

1.3 最后访问“+”的右子树



二叉树的遍历：中序遍历

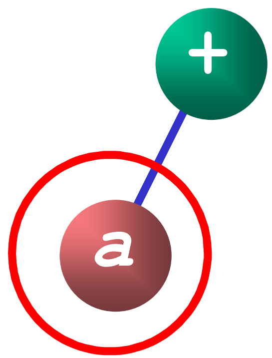
1.1 访问**“+”**的左子树

对于这棵左子树

1.1.1 先访问**“a”**的左子树（空）

1.1.2 再访问树根**“a”**

1.1.3 最后访问**“a”**的右子树（空）



二叉树的遍历：中序遍历

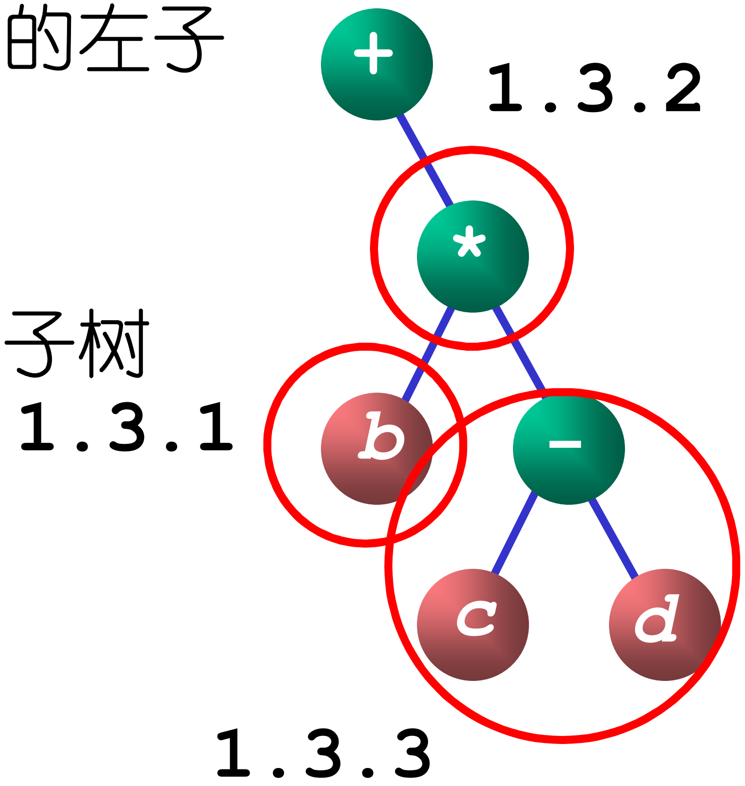
1.3 访问“+”的右子树

对于这棵右子树

1.3.1 先访问子树根“*”的左子树

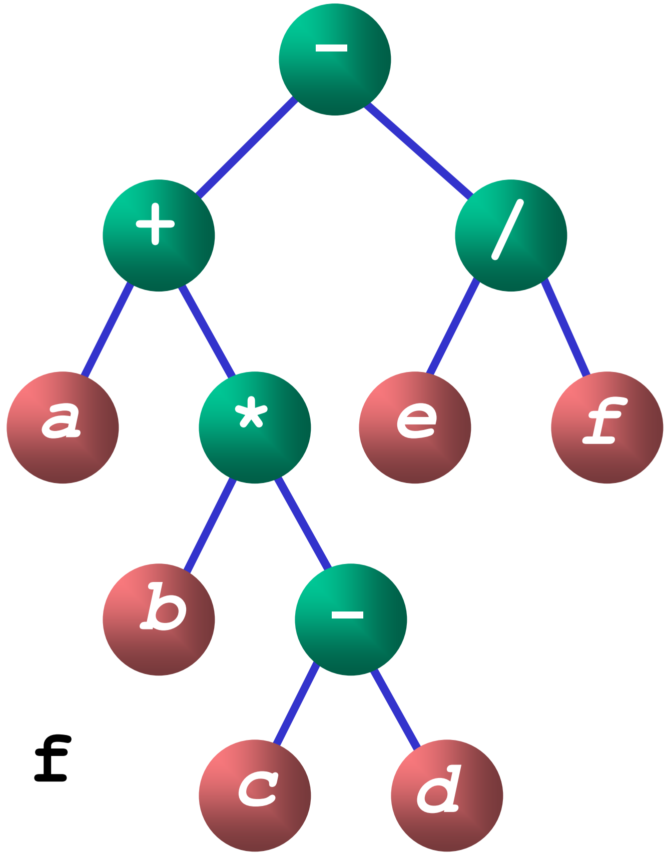
1.3.2 再访问子树根“*”

1.3.3 最后访问“*”的右子树



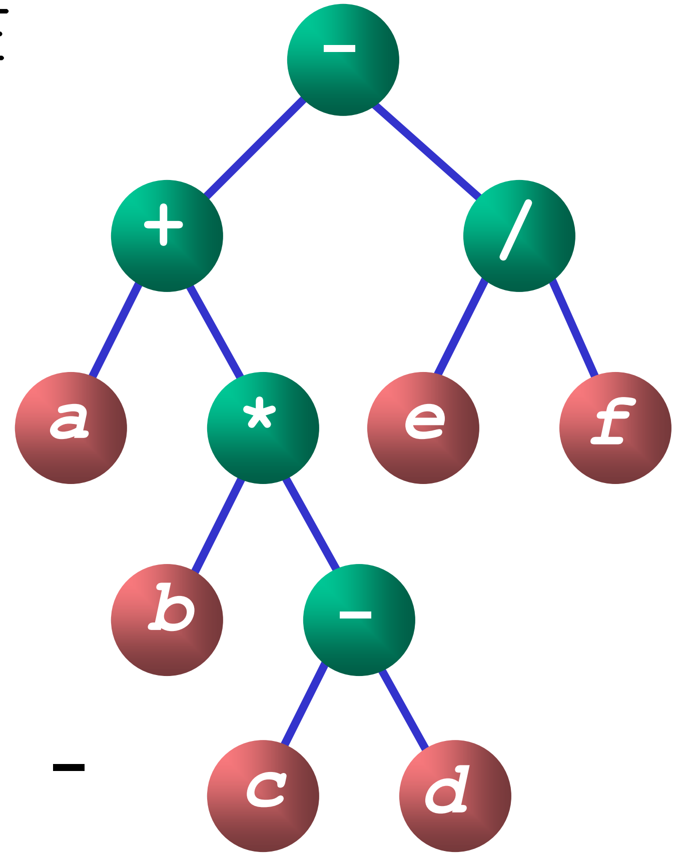
二叉树的遍历：中序遍历

- 理解
  - 原理和先序遍历相同
  - 区别在于递归的顺序：
    - 树根在左右两棵子树的中间被访问
  - 右图中序遍历结果为：
    - a + b * c - d - e / f



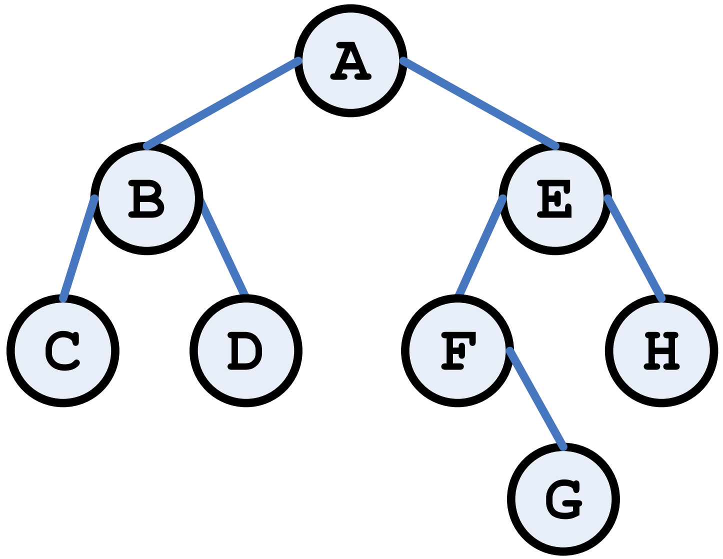
二叉树的遍历：后序遍历

- 后序遍历（**Postorder Traversal**）
  - 若二叉树为空，则空操作
  - 否则
    - 先后序遍历左子树
    - 再后序遍历右子树
    - 最后访问根结点
  - 右图后序遍历结果为：
    - **a b c d - * + e f / -**



二叉树的遍历

- 练习
  - 请写出下图先、中、后序遍历的结果
    - 先序：**ABCDEFGH**
    - 中序：**CBDAFGEH**
    - 后序：**CDBGFHEA**



二叉树的遍历：算法

- 回忆一下递归算法的适用情况
  - **(1)** 问题本身直接用递归定义的
  - **(2)** 问题的规律有递归的特点
- 二叉树及其遍历是用递归定义的
  - 用递归算法肯定可以解决
  - 如果不用递归呢？

二叉树的遍历：递归算法

- 编写递归程序的要点
  - **(1)** 把部分看成整体
    - 把整体的解决分成若干部分
    - 每个部分的解决方法和整体相同
    - 解决整体时假设部分已经解决
  - **(2)** 注意留递归出口

二叉树的遍历：递归算法

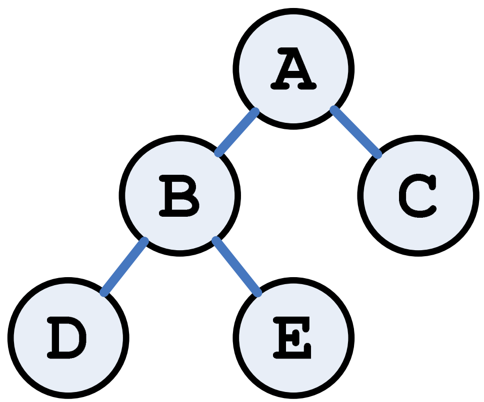
- 以中序遍历为例

```
void IOT(BiTree T, int(*Visit)(TElemType e))
{
   if(!T) return; //递归出口
   IOT(T->lchild); //中序遍历左子树
   Visit(T->data);
   IOT(T->rchild); //中序遍历右子树

}
```

IOT: InOrderTraverse

IOT(A)

- (1) IOT(B)
  - (1.1) IOT(D)
    - (1.1.1) IOT(NULL)
    - (1.1.2) Visit(D)
    - (1.1.3) IOT(NULL)
  - (1.2) Visit(B)
  - (1.3) IOT(E)
    - (1.3.1) IOT(NULL)
    - (1.3.2) Visit(E)
    - (1.3.3) IOT(NULL)
- (2) Visit(A)
- (3) IOT(C)
  - (3.1) IOT(NULL)
  - (3.2) Visit(C)
  - (3.3) IOT(NULL)



二叉树的遍历：非递归算法

- 用一个栈Stack跟踪访问的历史
- 伪代码：

```
void  dfs( root){  //输入: root node of tree
    InitStack(S);
    PushStack(S, root);  //push root to the stack S
    while( ! isEmpty(S) ){  //只要栈S不空
      node = S.pop();  //出栈一个结点
      visit(node);       //访问它
      //push node's right and left children
      if(node有右孩子rchild) PushStack(S, rchild);
      if(node有左孩子lchild) PushStack(S, lchild);
    }
}
```

二叉树的遍历：非递归算法

- 回顾遍历的过程（以中序为例）

  1 先走到最左

  2 往回访问父结点

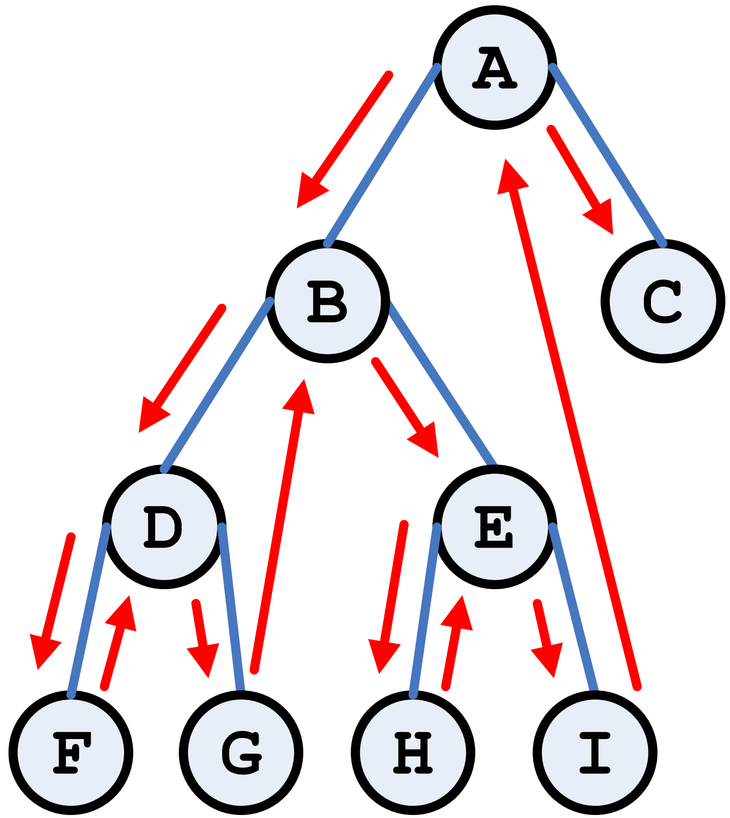
  3 往右访问右子树

    3.1 走到最左

    3.2 回访父结点

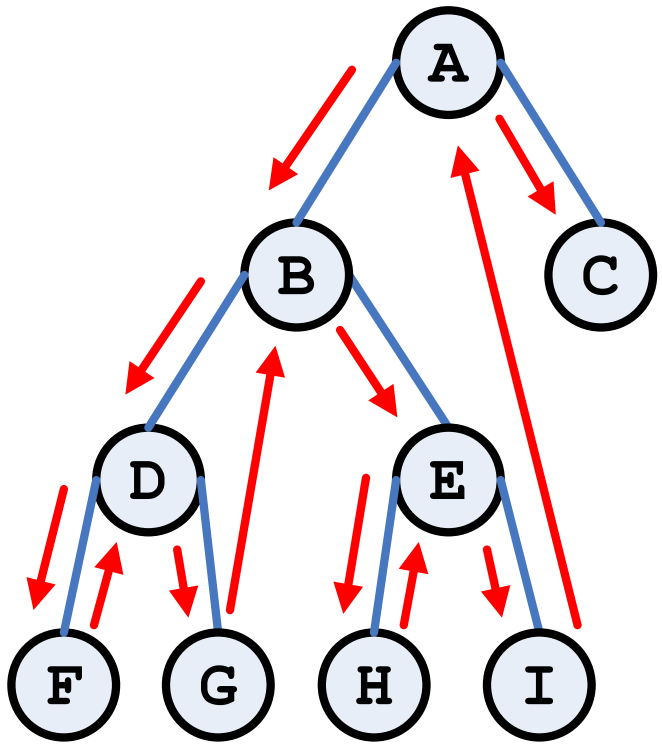
    3.3 访问右子树

    . . .



二叉树的遍历：非递归算法

- 有一个问题
  - 当向左/右走到底时怎么办？
  - 需要返回到祖先
  - 那个祖先？
  - 某个尚未访问的祖先
  - 这就需要一种机制能记录访问的“历史信息”，以便能够回溯回去
  - “历史信息”：路过却未访问的结点



二叉树的遍历：非递归算法

- 堆栈的特性及其应用
  - 特性：先进后出
  - 应用：
    - 颠倒元素的顺序（比如**P48：3.2.1**）
    - 记录历史信息
      - 人记忆历史：
        - 越近的事情回忆起来越容易
        - 越久的事情回忆起来越困难
      - 栈顶元素是最近压进去的，最先出来
      - 栈底元素是最早压进去的，最后出来

算法1：

```
Status InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);      //先把树根入栈
    while(!StackEmpty(S)){          //只要栈非空
        while(GetTop(S, p) && p)    //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                  //弹出多入栈的空结点
        if(!StackEmpty(S)){         //如果栈非空
            Pop(S, p);              //弹出一个元素
            if(!Visit(p->data)      //访问之
                return ERROR;
            Push(S, p->rchild);     //再向右走
        }
    }
    return OK;
}
```

```
Status InOrderTraverse (BiTree T)
{
    InitStack(S);  Push(S, T);
    while(!StackEmpty(S))
    {
        while(GetTop(S, p) && p)
            Push(S, p->lchild);
        Pop(S, p);
        if(!StackEmpty(S))
        {
            Pop(S, p);
            if(!Visit(p->data))
                return ERROR;
            Push(S, p->rchild);
        }
    }
    return OK;
}
```

树根入栈

只要栈非空

只要栈顶元素p非空

p的左孩子入栈

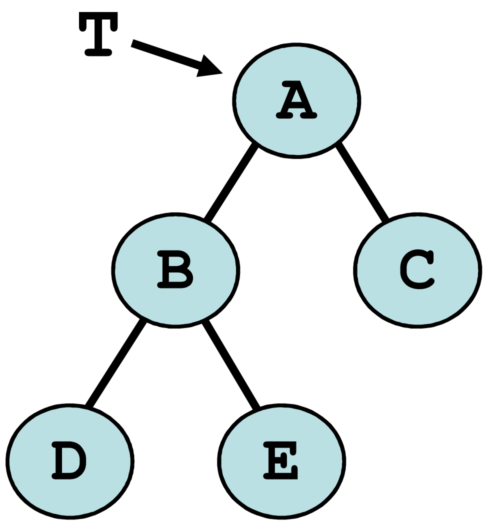
弹出多入栈的空结点

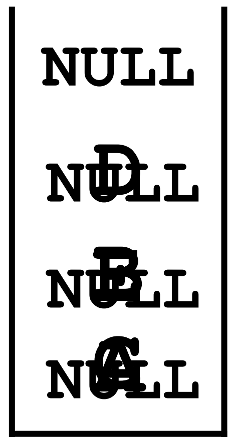
只要栈非空

弹出一个，访问之

p的右孩子入栈

程序结束





算法1:

```
Status InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);        //先把树根入栈
    while(!StackEmpty(S)){            //只要栈非空
        while(GetTop(S, p) && p)      //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                    //弹出多入栈的空结点
        if(!StackEmpty(S))
            Pop(S, p)
            if(!Vi
                re
            Push(S,
        }
    }
    return OK;
}
```

不停的“向左下走”

前面会“走过头”，所以以弹出多入栈的空指针

算法2：

```
Status InOrderTraverse (BiTree T) {
    InitStack(S);                        //新建一个堆栈
    p = T;                               //从树根开始
    while(p || !StackEmpty(S)){          //还有未访问的
        if(p) {                          //一直向左走到底
            Push(S, p);
            p = p->lchild;
        }
        else {                     //p为NULL，说明走到了底
            Pop(S, p);             //弹出一个还没访问的结点
            if(!Visit(p->data))          //访问之
                return ERROR;
            p = p->rchild;               //再向右走
        }
    }
    return OK;
}
```

```
Status InOrderTraverse (BiTree T)
{
    InitStack(S);
    p = T;
    while(p || !StackEmpty(S))
    {
      if(p)
      {
          Push(S, p);
          p = p->lchild;
      }
      else
      {
          Pop(S, p);
          if(!Visit(p->data))
              return ERROR;
          p = p->rchild;
      }
    }
    return OK;
}
```

p指向树根

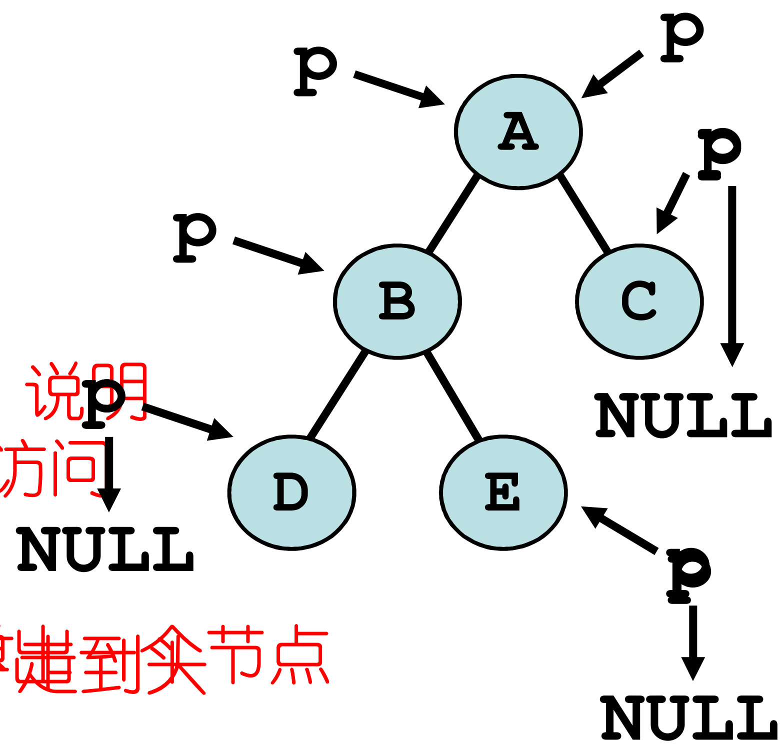
p或栈非空，说明还有节点未访问

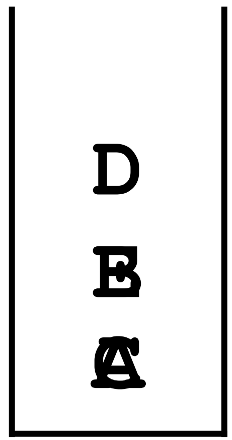
p向左走到头

从栈中弹出一个节点访问之

并让p向右走

程序结束





算法2：

```
Status InOrderTraverse (BiTree T) {
    InitStack(S);  p = T;          //从树根开始
    while(p || !StackEmpty(S)){    //还有未访问的
      if(p) {                      //一直向左走到底
          Push(S, p);
          p = p->lchild;
      }
      else {
          Pop(S, p);
          if(!Visit(p->data))     //访问之
              return ERROR;
          p = p->rchild;
      }
    }
    return OK;
}
```

p向左下走到底
并记录下沿途的节点

p走到了底，再依次弹
出刚才路过却没有访
问的节点，访问之，
然后p向右走

后序非递归遍历

```
vector<int> postorderTraversal(TreeNode *root) {
    list<int> result;
    stack<TreeNode *> S;
    TreeNode *p = root;
    while (!S.empty() || p!= nullptr) {
        if (p!= nullptr) {//访问p,并走向右子树
            S.push(p);
            result.push_front(p->val);
            p = p->right;
        }
        else {  //右子树走完了
            TreeNode *cur = S.top();
            S.pop();
            p = cur->left;
        }
    }
    return vector<int>(result.rbegin(), result.rend());
}
```

后序非递归遍历

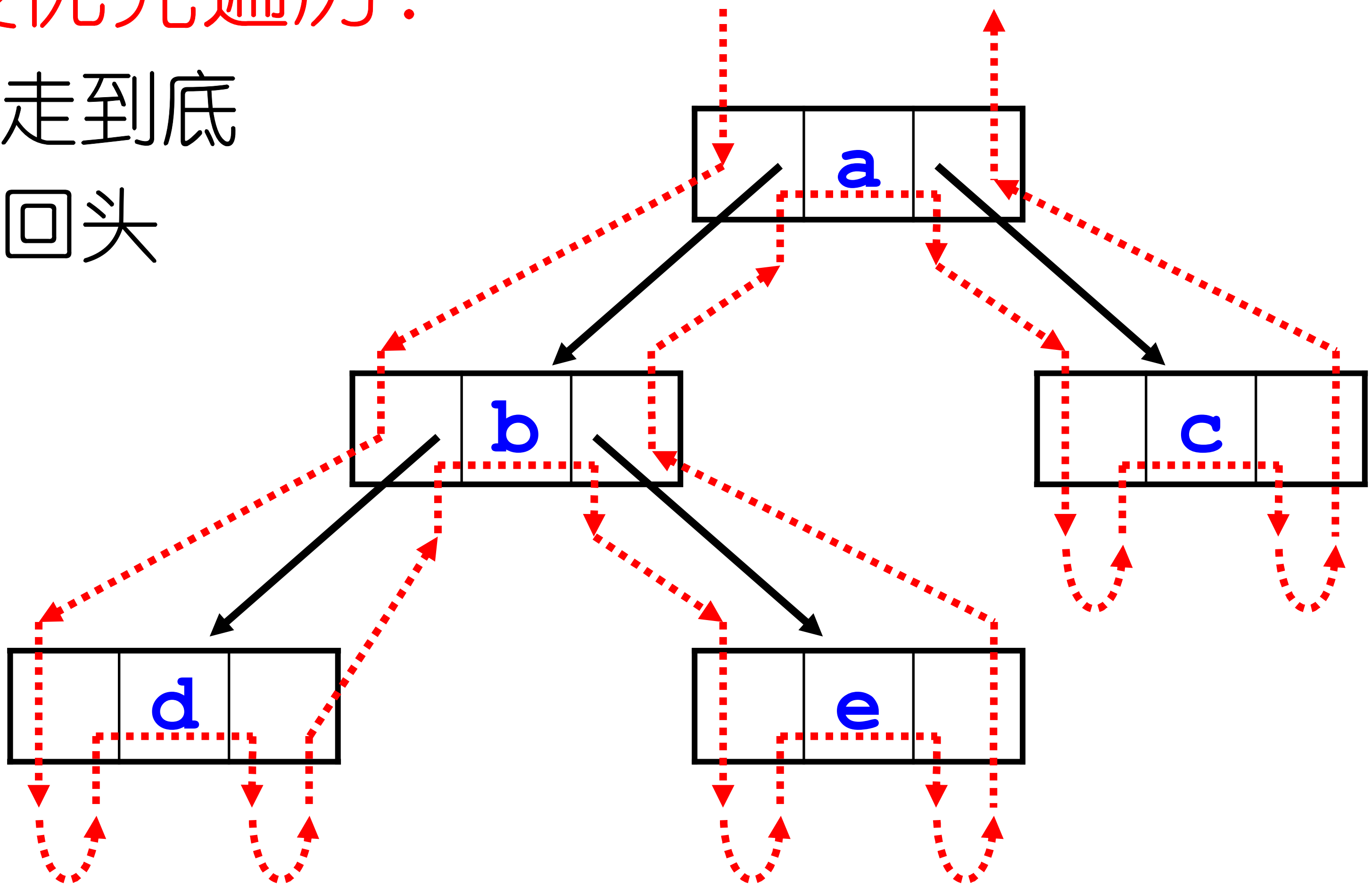
```
vector<int> postorderTraversal(TreeNode *root) {
    vector<int> result;
    stack<const TreeNode *> s;
    /* p, 正在访问的结点, q, 刚刚访问过的结点 */
    const TreeNode *p = root, *q = nullptr;

    do {
        while (p != nullptr) { /* 往左下走 */
            s.push(p);
            p = p->left;
        }
        q = nullptr;
        while (!s.empty()) {
            p = s.top();
            s.pop();
            /* 右孩子不存在或已被访问, 访问之 */
            if (p->right == q) {
                result.push_back(p->val);
                q = p; /* 保存刚访问过的结点 */
            } else {
                /* 当前结点不能访问, 需第二次进栈 */
                s.push(p);
                /* 先处理右子树 */
                p = p->right;
                break;
            }
        }
    } while (!s.empty());

    return result;
}
```

二叉树的遍历：算法效率

- 时间复杂度
  - **n**个结点，每一个都要访问一次
  - 显然时间复杂度为O(**n**)（这里**n**为结点数）
- 空间复杂度
  - 不论是递归还是非递归算法都要用到堆栈
  - 堆栈的最大深度 **=** 树的深度
  - 所有空间复杂度为O(**k**)（这里**k**为树的深度）

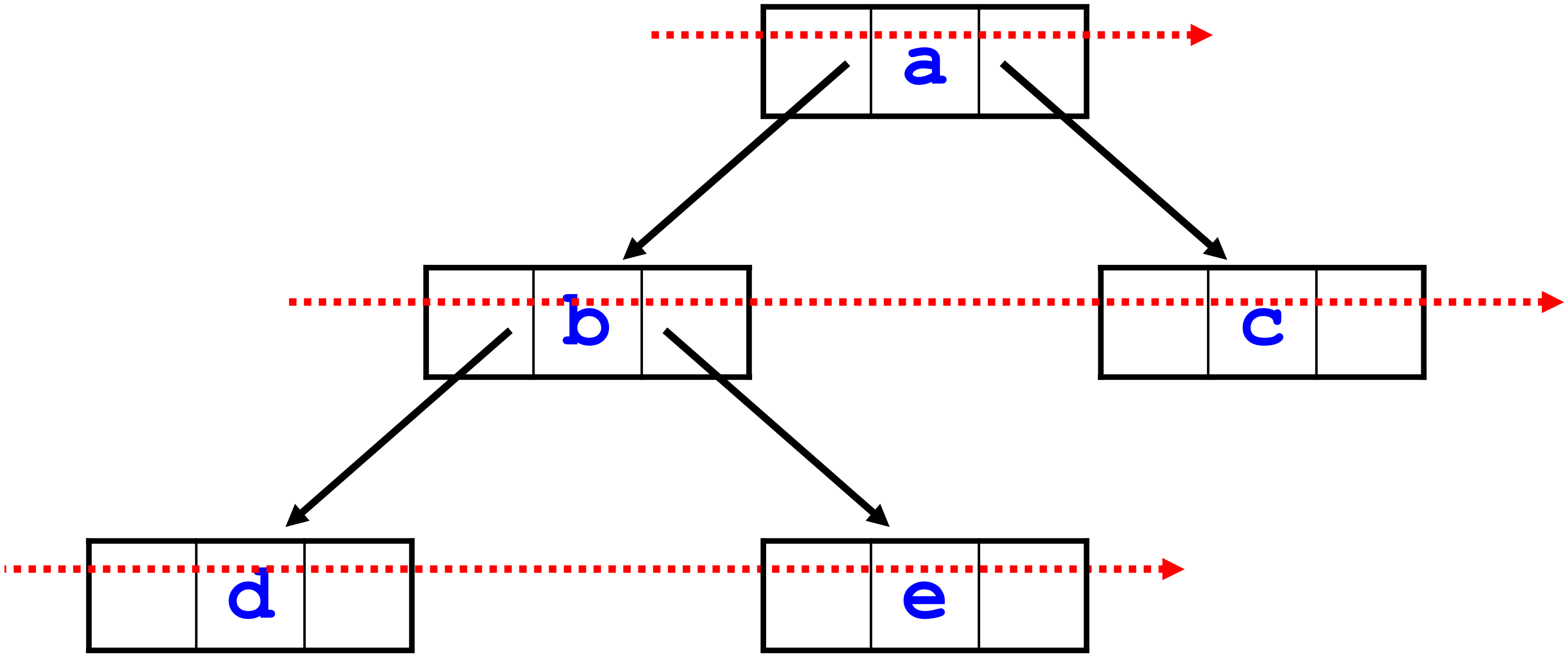
二叉树的其它操作：广度优先遍历

- 深度优先遍历：
  - 先走到底
  - 再回头



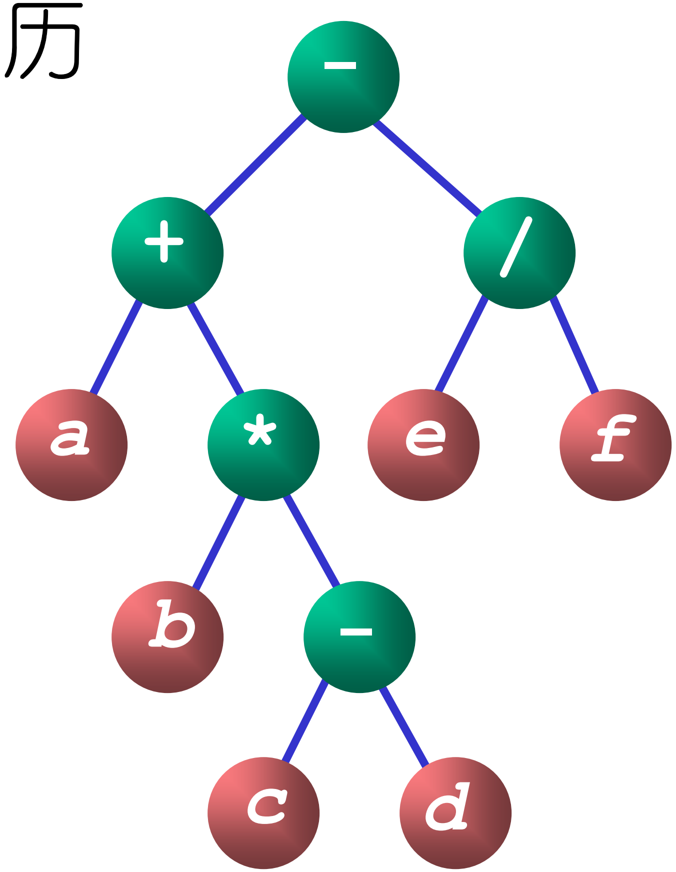
二叉树的其它操作：广度优先遍历

- 广度优先遍历
  - 也叫层序遍历



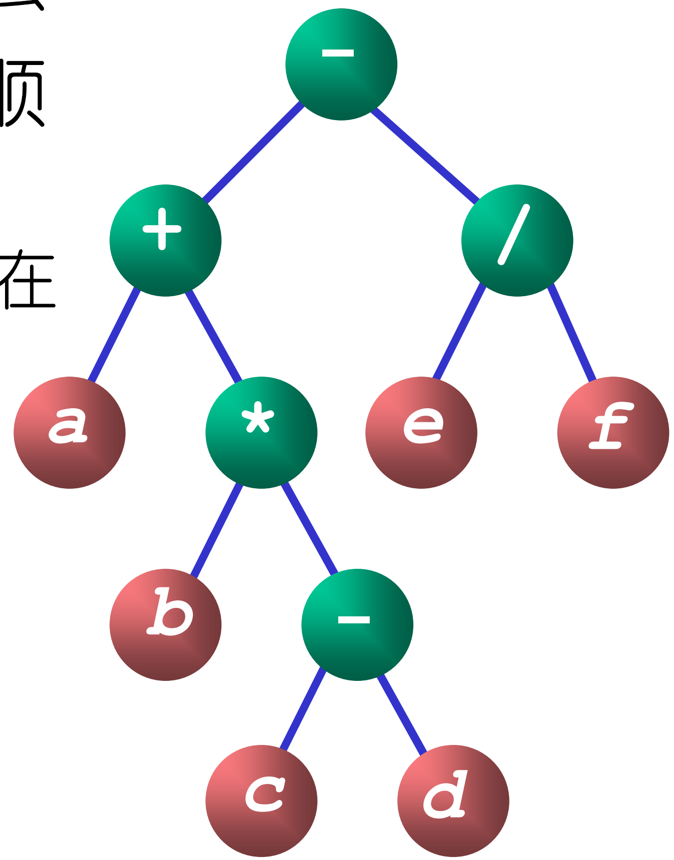
二叉树的其它操作：广度优先遍历

- 练习
  - 对右图进行广度优先遍历
  - **-+/a*efb-cd**



二叉树的其它操作：广度优先遍历

- 规律
  - 一个层次完成后才进入下一层
  - 下一层的结点按照上一层的顺序进行访问
    - “老爸在长辈中在先，则儿子在晚辈中也在先”
  - 总之，先遇上的先访问
- 对比深度优先遍历：
  - 先遇上的后访问
  - 算法使用了堆栈



算法：

```
Status LevelOrderTraverse (BiTree T)
{
  InitQueue(Q);  AddQueue(Q, T); //树根入队
  while(!QueueEmpty(Q)) {        //只要队列非空
     DeleteQueue(Q, p);          //出队一个结点
     if(!Visit(p->data))         //访问之
        return ERROR;
     if(p->lchild)               //左孩子入队
        AddQueue(Q, p->lchild);
     if(p->rchild)               //右孩子入队
        AddQueue(Q, p->rchild);
  }
  return OK;
}
```

```
Status LevelOrderTraverse(BiTree T) {
   InitQueue(Q);  AddQueue(Q, T);
   while(!QueueEmpty(Q))
   {
      DeleteQueue(Q, p);
      if(!Visit(p->data))
         return ERROR;
      if(p->lchild)
         AddQueue(Q, p->lchild);
      if(p->rchild)
         AddQueue(Q, p->rchild);
   }
   return OK;
}
```

树根入队

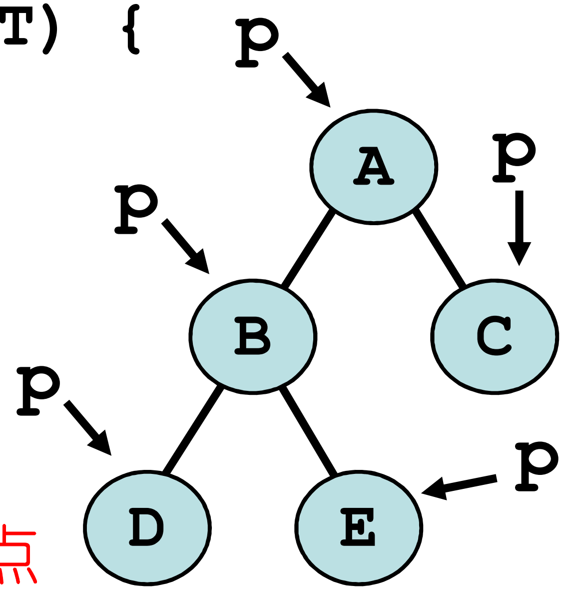
只要队列非空

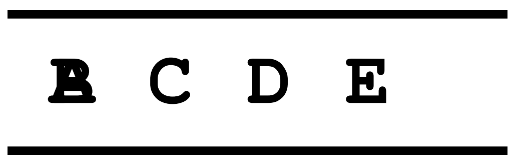
出队一个节点并访问之

如果**p**有左孩子则左孩子入队

如果**p**有右孩子则右孩子入队

程序结束





二叉树的其它操作

- 建议思考以下问题：
  - 判断两棵二叉树是否相同
  - 复制一棵二叉树
  - 交换一棵二叉树的所有左右子树
  - 计算一棵二叉树叶结点的个数
  - 计算一棵二叉树的深度、宽度（每层结点数的最大值）
  - 判断一棵二叉树是否是完全二叉树
  - . . .

二叉树的其它操作

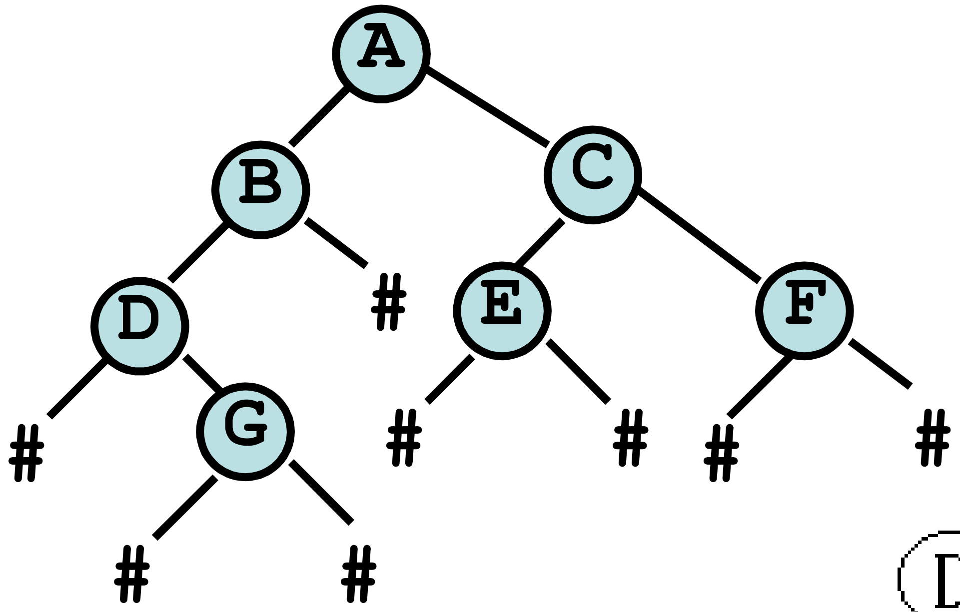
1． 求二叉树的深度(后序遍历)

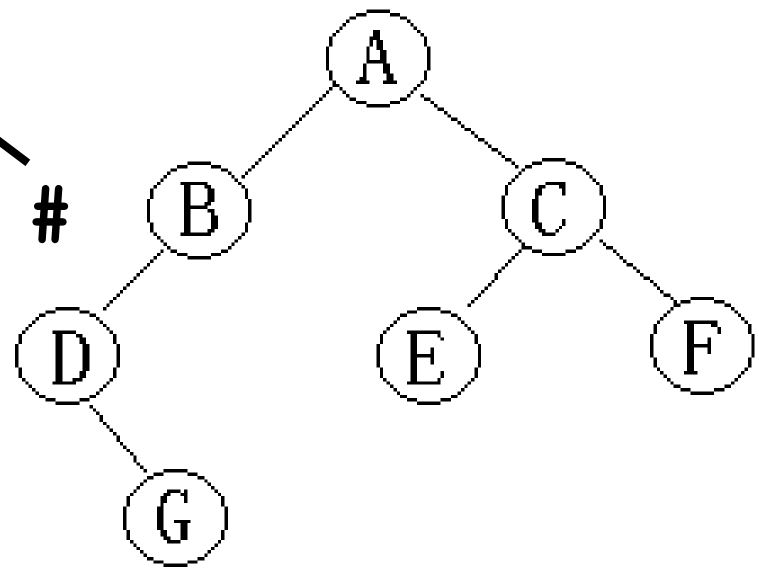
```
int Depth (BiTNode* T ){
  if (!T) return 0 ;
  int l = Depth( T->lchild );
  int r = Depth( T->rchild );
  return l>r?l+1:r+1;
}
```

二叉树的其它操作

2. 按带空子树标记(#符号)的先序序列建立二叉链表.

A B D # G # # # C E # # F # #





二叉树的其它操作

```
//ABD #G # # #CE ##F ##
void PreOrderCreatBiTree(BiTNode* &T)
{
   scanf(ch);
   if(ch== ´#´){ T = 0; return;} //递归出口

   if(!(T= new BiTNode) ){ //处理根
     throw “内存不够!”;return;
   }
   T->data = ch;

   PreOrderCreatBiTree(T->lchild);//左子树
   PreOrderCreatBiTree(T->rchild);//右子树
}
```

A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
   scanf(ch);
   if(ch== ´#´)T = NULL;
   else{
       if( !(T = new BiTNode)) )
          throw "内存不够!";
       T->data = ch;
       PreOrderCreatBiTree(T->lchild);
       PreOrderCreatBiTree(T->rchild);
  }
}
```

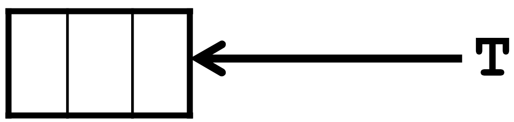
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```

T

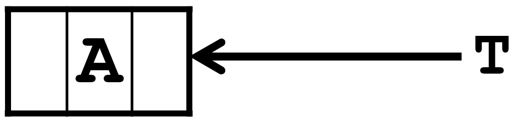
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



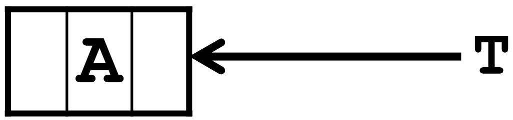
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
   scanf(ch);
   if(ch== ´#´)T = NULL;
   else{
       if( !(T = new BiTNode)) )
           throw “内存不够!”;
       T->data = ch;
       PreOrderCreatBiTree(T->lchild);
       PreOrderCreatBiTree(T->rchild);
   }
}
```



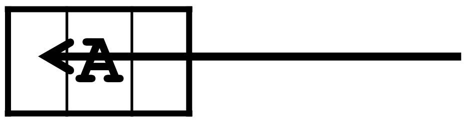
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



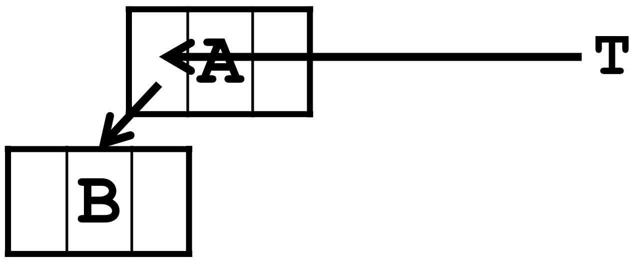
T

A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw “内存不够!”;
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```

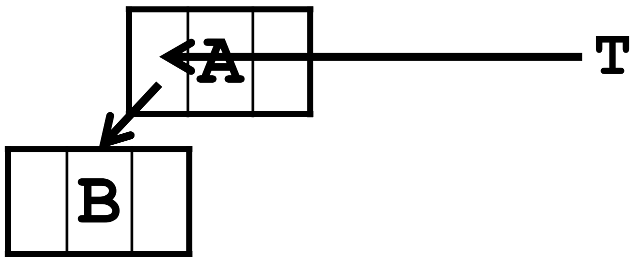
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw “内存不够!”;
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



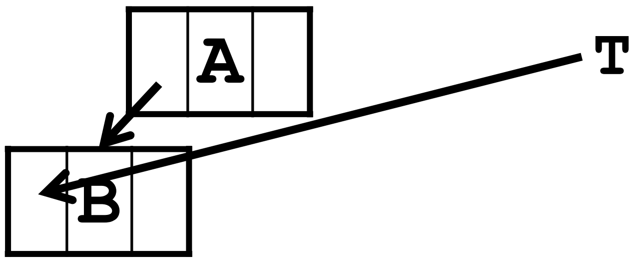
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



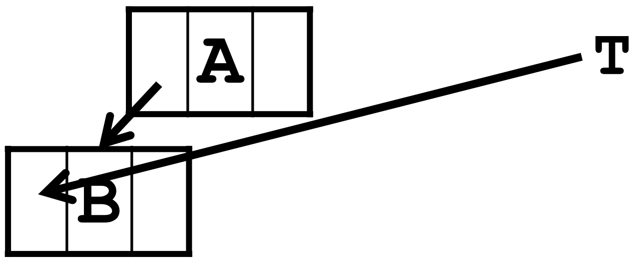
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
   scanf(ch);
   if(ch== ´#´)T = NULL;
   else{
       if( !(T = new BiTNode)) )
          throw "内存不够!";
       T->data = ch;
       PreOrderCreatBiTree(T->lchild);
       PreOrderCreatBiTree(T->rchild);
  }
}
```



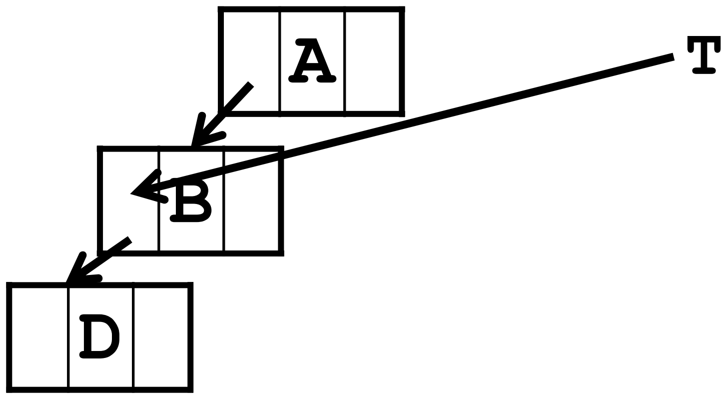
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



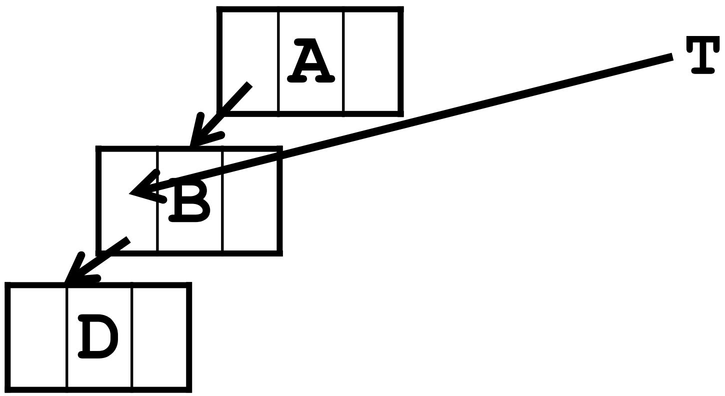
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



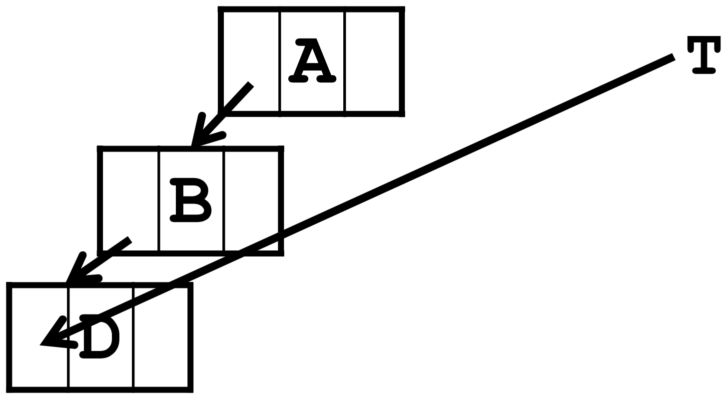
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



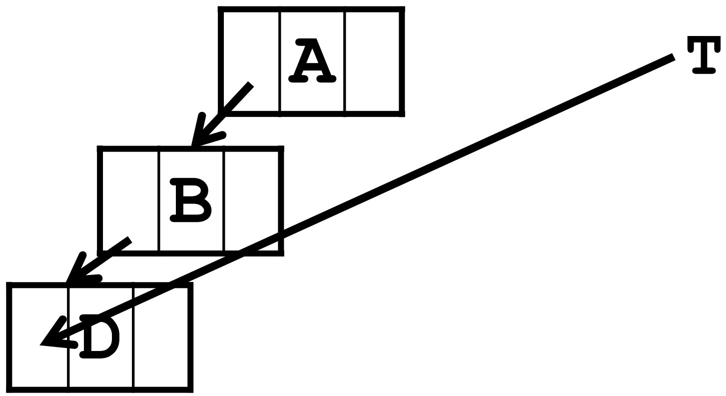
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw “内存不够!”;
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



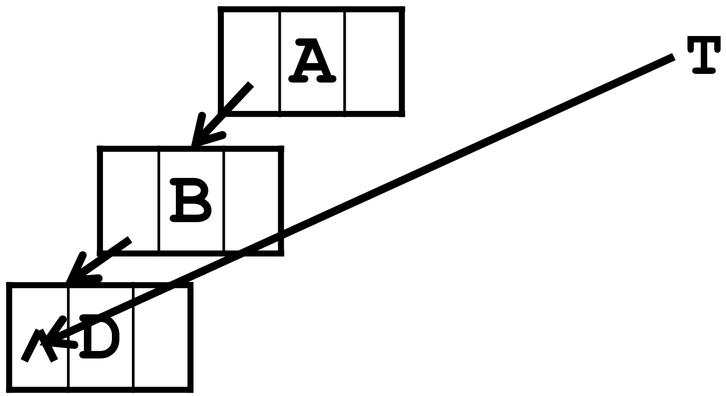
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw “内存不够!”;
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



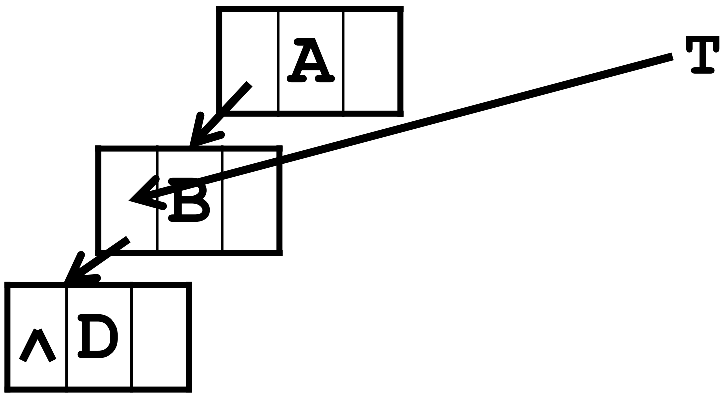
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
   scanf(ch);
   if(ch== ´#´)T = NULL;
   else{
       if( !(T = new BiTNode)) )
           throw "内存不够!";
       T->data = ch;
       PreOrderCreatBiTree(T->lchild);
       PreOrderCreatBiTree(T->rchild);
  }
}
```



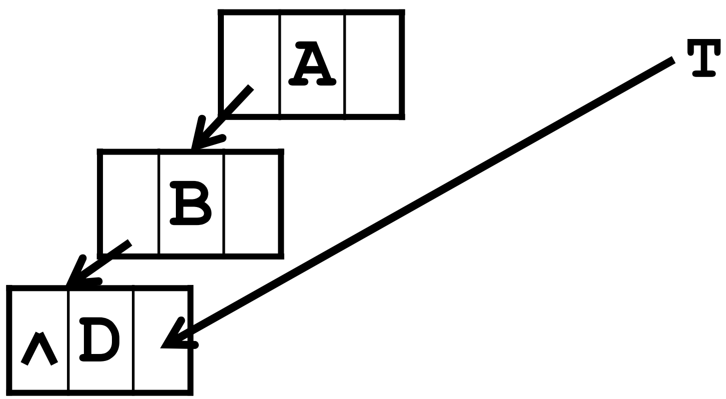
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
   scanf(ch);
   if(ch== ´#´)T = NULL;
   else{
       if( !(T = new BiTNode)) )
          throw "内存不够!";
       T->data = ch;
       PreOrderCreatBiTree(T->lchild);
       PreOrderCreatBiTree(T->rchild);
  }
}
```



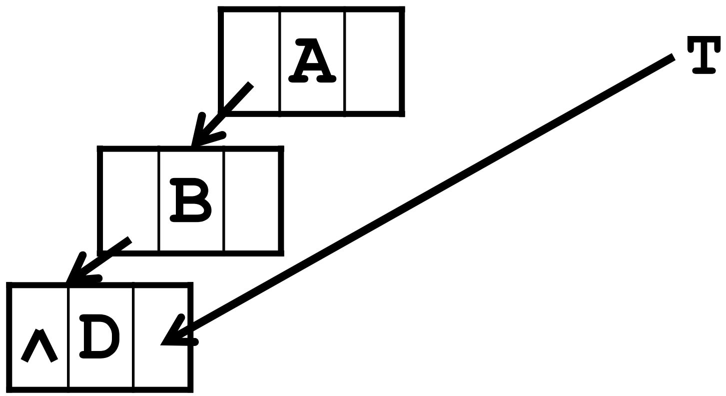
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw “内存不够!”;
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



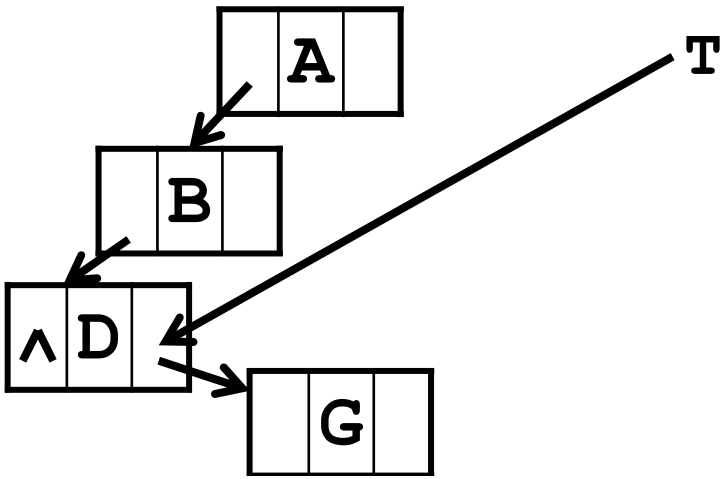
A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
   scanf(ch);
   if(ch== ´#´)T = NULL;
   else{
       if( !(T = new BiTNode)) )
          throw "内存不够!";
       T->data = ch;
       PreOrderCreatBiTree(T->lchild);
       PreOrderCreatBiTree(T->rchild);
  }
}
```



A B D # G # # # C E # # F # #

```
int PreOrderCreatBiTree(BiTNode* &T){
    scanf(ch);
    if(ch== ´#´)T = NULL;
    else{
        if( !(T = new BiTNode)) )
            throw "内存不够!";
        T->data = ch;
        PreOrderCreatBiTree(T->lchild);
        PreOrderCreatBiTree(T->rchild);
    }
}
```



二叉树的操作

- 本节小结
  - 深度优先遍历（包括先、中、后序）
    - 手工能写出先、中、后序遍历的结果
    - 要求能够写出递归和非递归的算法
  - 广度优先遍历
    - 了解其算法：为什么要用队列
  - 其它操作
    - 能把握大致方向：借助深度优先遍历、递归、广度优先遍历等等

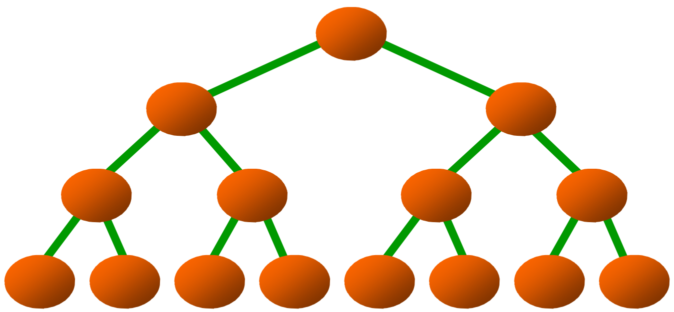
二叉树的其它操作

- 作业1
  - 习题集6.8、6.9

线索二叉树

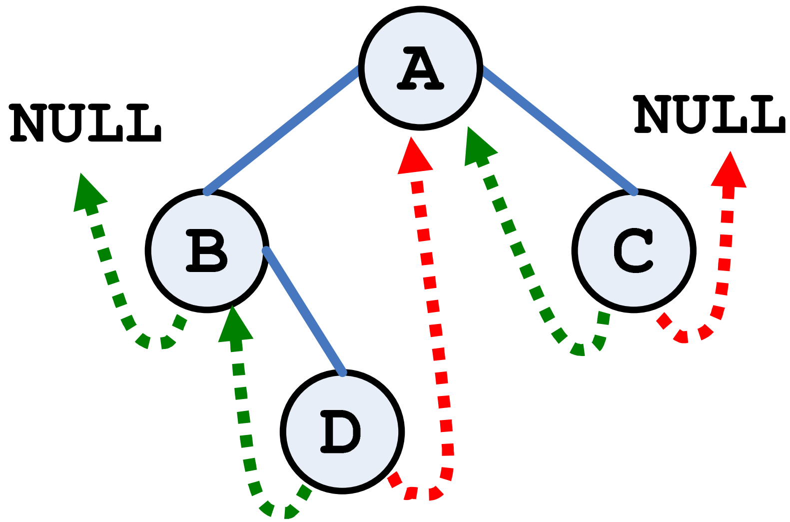
- 二叉链表的空间浪费

  - 对于叶结点，左右孩子指针无用！
    - 度为**1**的结点也闲置了一个指针
  - 而所有结点中最多有一半以上是叶结点

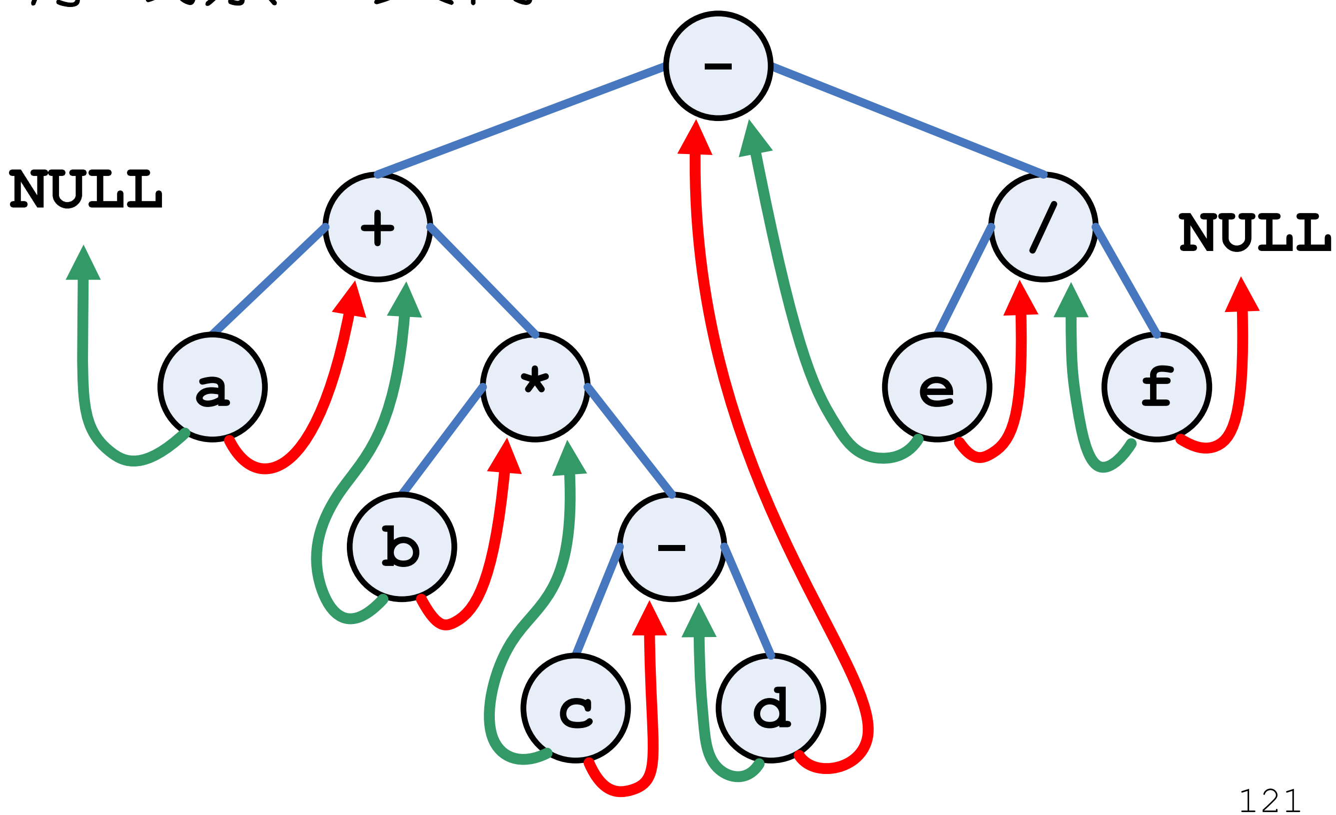


线索二叉树

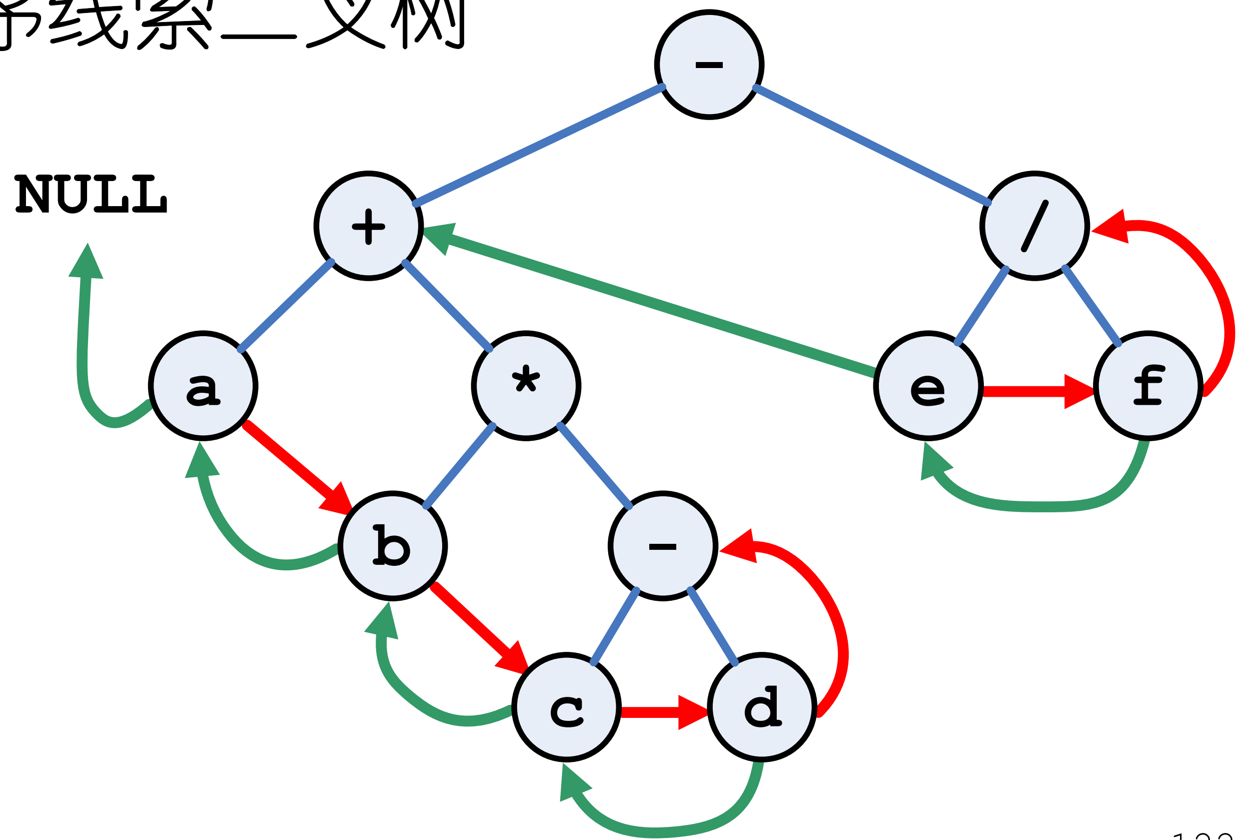
- 闲置指针的“废物利用”
  - 二叉树最常用的操作是遍历
  - 不妨让这些闲置的指针指向遍历时要访问的前驱/后继结点
  - 比如对于中序遍历
    - 结果应该是：**BDAC**



- 练习
  - 对下图按照中序遍历进行线索化
  - 中序遍历的结果为：**a+b*c-d-e/f**
  - 称作中序线索二叉树



- 练习
  - 对下图按照后序遍历进行线索化
  - 后序遍历的结果为：**abcd-*+ef/-**
  - 称作后序线索二叉树



线索二叉树

- 课后练习
  - 例题**5**

结点的结构

- 现在**lchild**有**2**个功能

  (1)指向左孩子

  (2)指向前驱结点

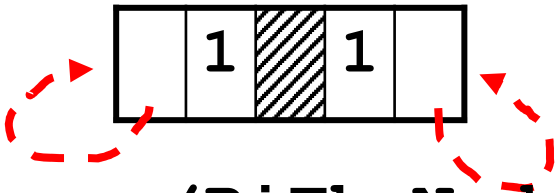
- 如何区别呢？
  - 增加一个标记说明其当前的功能
  - 当**LTag=0**时：指向左孩子
  - 当**LTag=1**时：指向前驱结点
- **rchild**类似处理

lchild	**LTag**	**Data**	**RTag**	**rchild**

结点的结构

```
enum PointerTag{Link,Thread};

 template <class TElemType>
 class BiThrNode {
 private:
     TElemType data;
     Struct BiThrNode *lchild;
     Struct BiThrNode *rchild;
     PointerTag LTag;
     PointerTag RTag;
 };
```

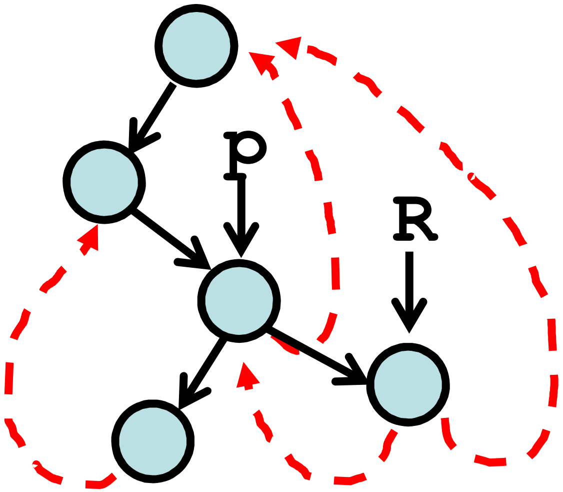
初始化一个头结点




```
bool InitThrTree (BiThrNode *& head) {
    head = (BiThrNode *)
                malloc(sizeof(BiThrNode));
    if(!head) return false;
    head->Ltag = 1;   //线索
    head->Rtag = 1;   //线索
    head->lchild = 0;
    head->rchild = 0;
    return true;
};
```

线索二叉树：插入一个右孩子

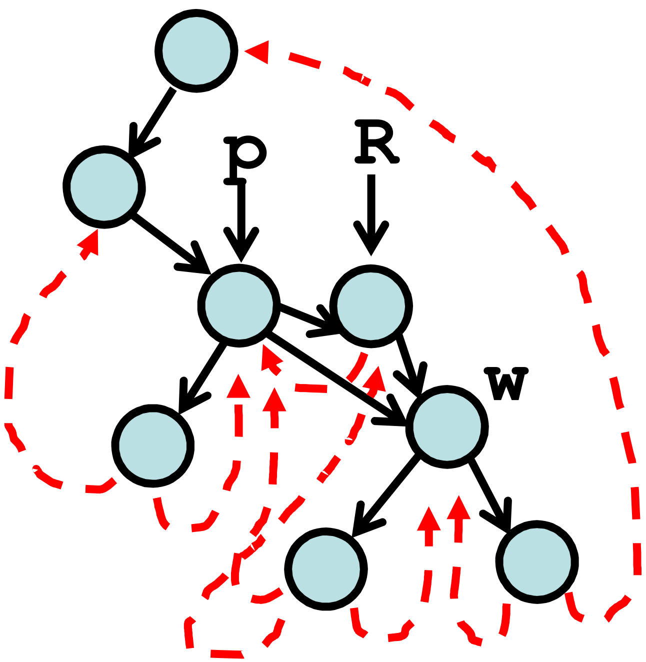
1) P没有右孩子




```
R->rchild = p->rchild
R->Rtag = p->Rtag;

R->lchild = p;
R->Ltag = 1; //线索

P->rchild = R;
P->Rtag = 0; //孩子
```

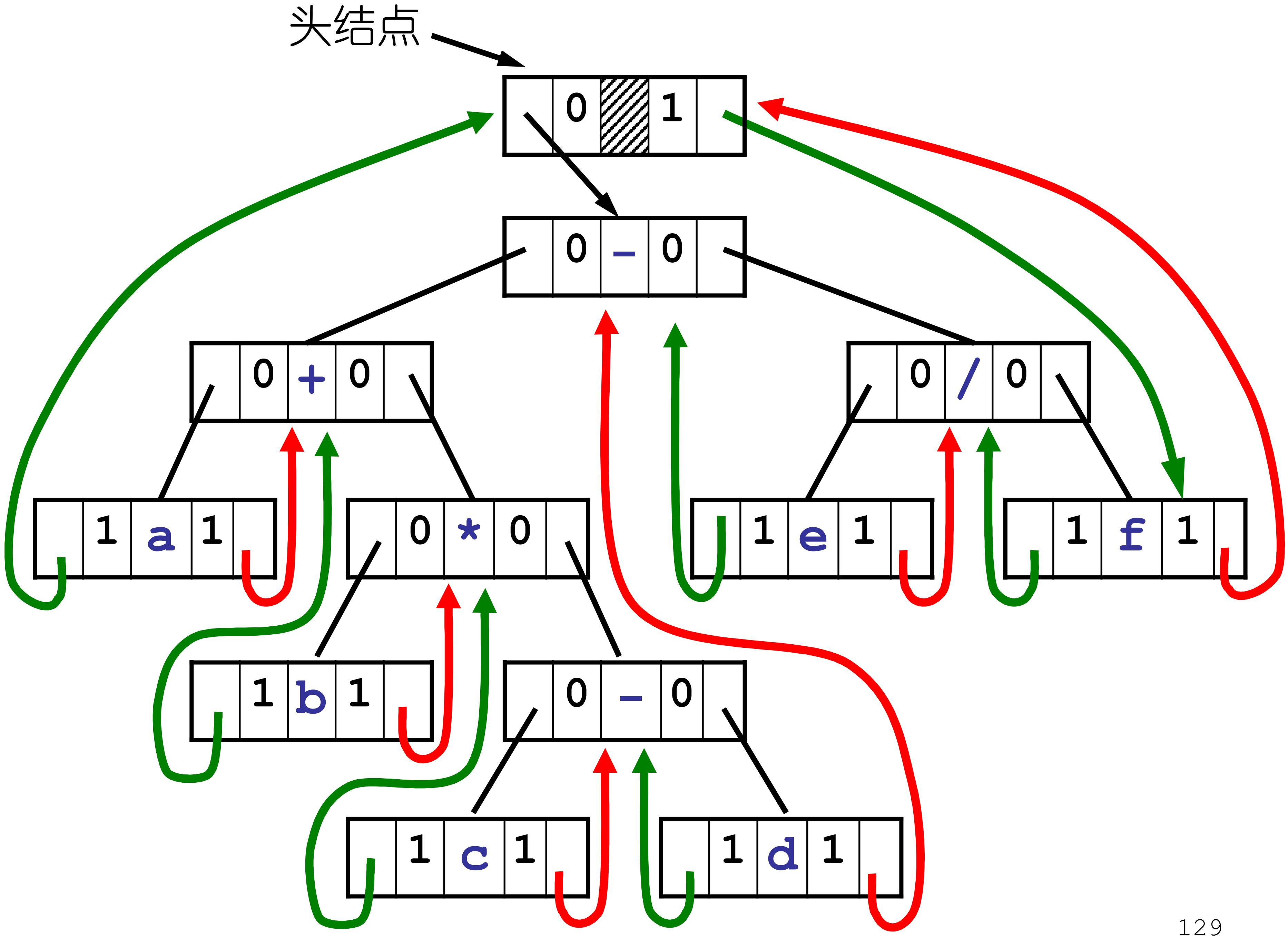
插入一个右孩子

2)P有右孩子



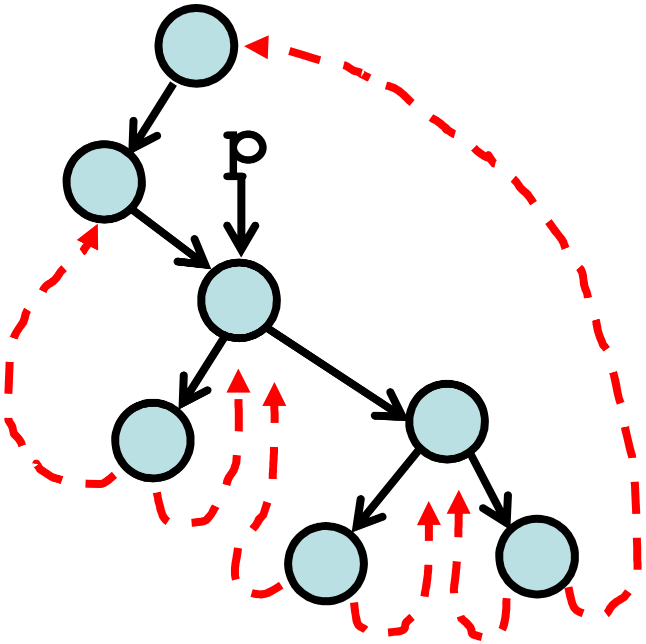
寻找R->rchild的
最左下的结点

```
R->rchild = p->rchild
R->Rtag = p->Rtag;

R->lchild = p;
R->Ltag = 1; //线索

P->rchild = R;
P->Rtag = 0; //孩子
if(R->Rtag) return;//线索
w= R->rchild;
while(w->LRag)
    w = w->lchild;
W->lchild = R;
```


头结点
0 1
0 - 0
0 + 0
0 / 0
1 a 1
0 * 0
1 e 1
1 f 1
1 b 1
0 - 0
1 c 1
1 d 1

中序线索二叉树的遍历

- 基本思想
  - 第一个访问的结点应该是最左下角的结点
  - 假设刚才访问的结点是**p**
  - 然后**P**的后继是谁？
    - 若**p->rchild**是指针，说明**P**有右子树，下一个结点应该是**P**右子树中最左下角的结点
    - 若**p->rchild**是线索，直接访问**p->rchild**
  - 如此循环往复...



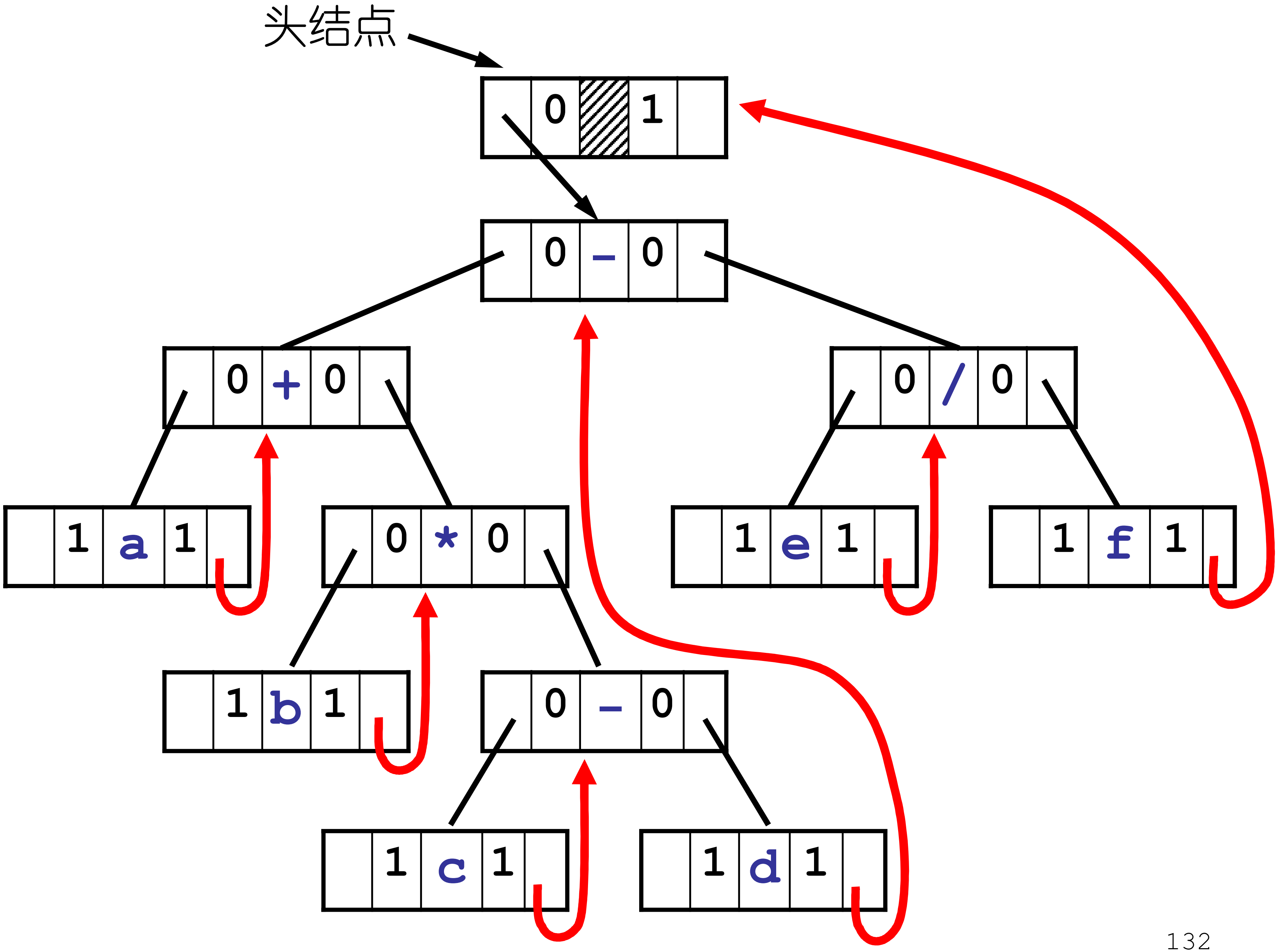
然后P的后继是谁？

1)若p->rchild是指针，说明P有右子树，下一个结点应该是P右子树中最左下角的结点

```
if(p->Rtag==Link){
    p = p->rchild;
    while(p->Ltag==Link)
      p = p->lchild;
}
```

2)若p->rchild是线索，直接访问p->rchild

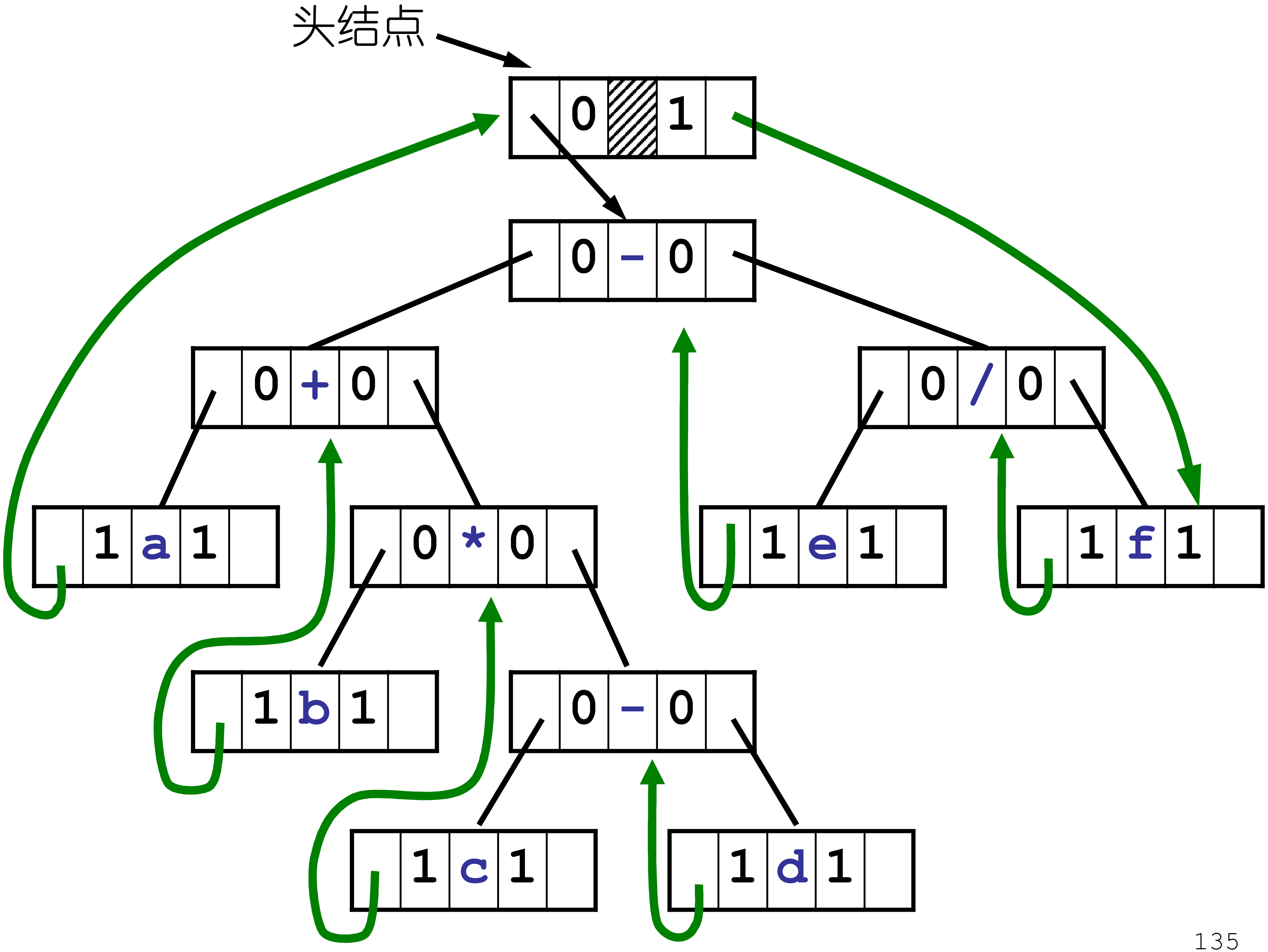
```
  p = p->rchild;
```


头结点
0
1
0 - 0
0 + 0
0 / 0
1 a 1
0 * 0
1 e 1
1 f 1
1 b 1
0 - 0
1 c 1
1 d 1

```
Status InOrderTraverse_Thr(BiTree Head){ //T为头节点
    p = Head ->lchild;          //p指向树根
    while(p->LTag == Link)      //p->lchild为指针
          p = p->lchild;          //则向左下走到底

    while(p != Head){       //p等于T则说明已经遍历完毕
        std::cout << p->data;//visit(p->data)
        if(p->Rtag==Link){
           p = p->rchild;
           while(p->Ltag==Link)
              p = p->lchild; //则向左下走到底
        }
        else p = p->rchild;
    }
    return OK;
}
```

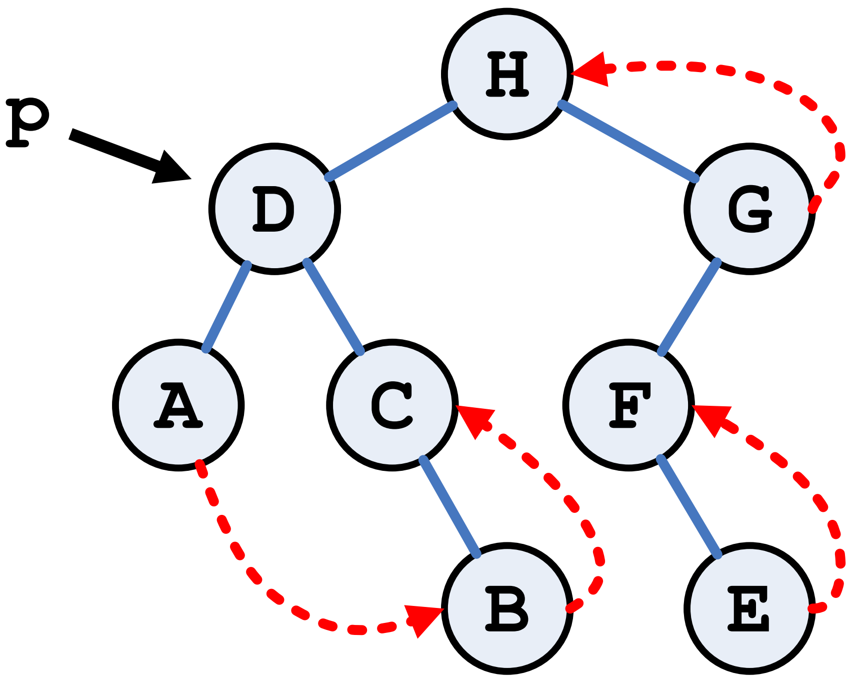
中序线索二叉树的(反向)遍历

- 思考：如果反方向进行遍历呢？
  - 第一个访问的结点应该是最右下角的结点
  - 假设刚才访问的结点是**p**
  - 然后**p**的后继是谁？
    - 若**p->lchild**是指针，说明**p**有左子树，下一个结点应该是**p**左子树中最右下角的结点
    - 若**p->lchild**是线索，直接访问**p->lchild**
  - 如此循环往复...


头结点
0 1
0 - 0
0 + 0
0 / 0
1 a 1
0 * 0
1 e 1
1 f 1
1 b 1
0 - 0
1 c 1
1 d 1

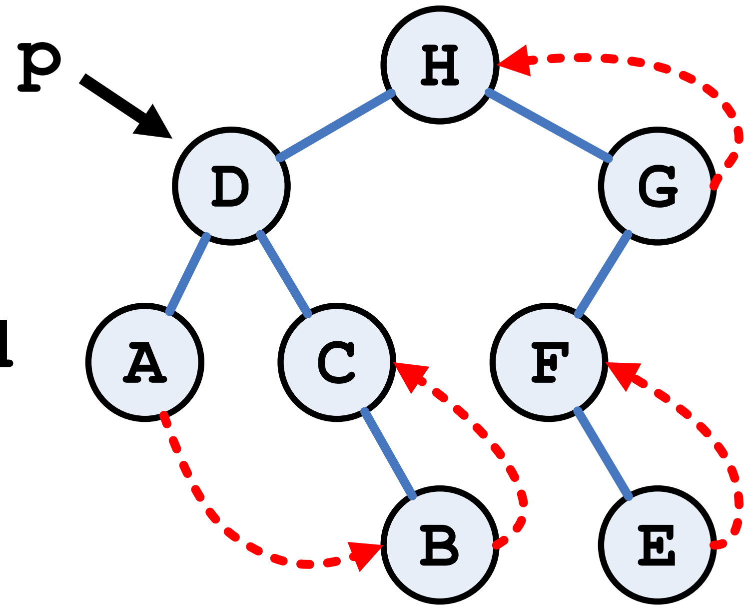
先/后序线索二叉树的遍历

- 以后序线索二叉树为例
  - 第一个访问的应该是最左下角的结点
  - 假设刚才访问的是**p**，**p**的后继是谁？



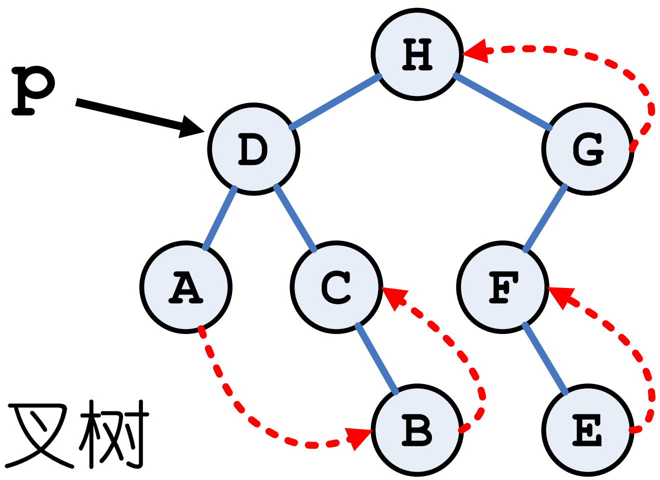
先/后序线索二叉树的遍历

- 解答
  - 若p->rchild是线索
    - 则后继就是p->rchild
  - 否则
    - 若p是树根，则无后继
    - 若p是右孩子或者是唯一的左孩子，则后继是其父结点
    - 若p是左孩子，且有右兄弟，则后继是p的父结点的右子树中第一个访问的结点



先/后序线索二叉树的遍历

- 困难
  - 对于后序线索二叉树
  - 想要找结点**p**的后继结点，可能需要知道**p**的父结点是谁
  - 可是这是很难办到的
  - 两种方法
    - 从树根开始查找
    - 改用三叉链表来表示二叉树
  - 此困难对于后序线索二叉树找前驱结点、先序线索二叉树找后继/前驱结点同样存在



普通二叉树的线索化

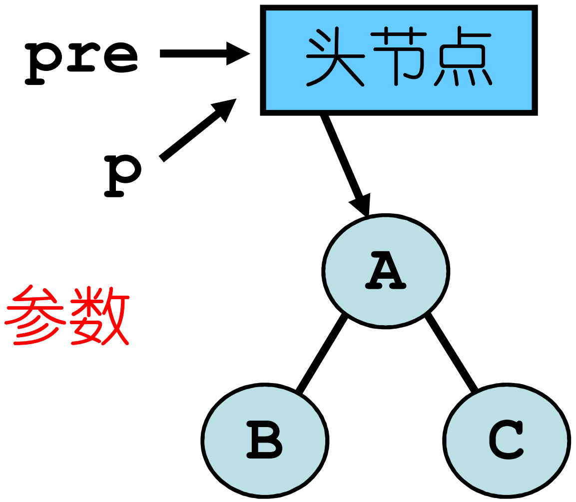
- 普通的二叉树怎么变成线索二叉树？
  - 称作线索化
- 基本思想
  - 线索其实就是按照遍历的顺序把闲置的指针链接到前驱/后继结点：
    - 遍历过程中维护两个指针：**pre**和**p**，分别指向遍历序列中一前一后的两个结点
    - 若**pre->rchild**闲置，**pre->rchild = p**
    - 若**p->lchild**闲置，**p->lchild = pre**

```
Status InOrderThreading(BiThrTree &Thrt,BiThrTree T){
    Thrt = (BiThrTree) malloc(sizeof(BiThrNode)); //头结点
    if(!Thrt)  exit(OVERFLOW);
    Thrt->LTag = Link;          //头结点的lchild是指针
    Thrt->RTag = Thread;        //头结点的rchild是线索
    if(!T) {                    //若T为空树，头结点的左右指针回指
        Thrt->lchild=Thrt;  Thrt->rchild=Thrt; }
    else {
        Thrt->lchild=T;         //头结点的lchild指向树根
        pre = Thrt;             //pre是全局变量
        InThreading(T);         //调用中序线索化函数处理二叉树T

        pre->rchild = Thrt;     //InThreading调用完以后
        pre->RTag = Thread;     //就差最后一个结点没有链接好
        Thrt->rchild = pre;     //此时，pre指向最后一个结点
    }
    return OK;
}
```

```
void InThreading(BiThrTree p) {
    if(p) {                              //p为空则返回
        InThreading(p->lchild);   //左子树线索化
        if(!p->lchild) {        //若p->lchild闲置
            p->lchild = pre;
            p->LRag   = Thread;
        }
        if(!pre->rchild) {  //若pre->rchild闲置
            pre->rchild = p;
            pre->RTag = Thread;
        }
        pre = p;             //维持pre和p一前一后的关系
        InThreading(p->rchild);  //右子树线索化
    }
}
```

```
void InThreading(BiThrTree p) {
    if(p)
    {
        InThreading(p->lchild);
        if(!p->lchild)
        {
            p->lchild = pre;
            p->LRag    = Thread;
        }
        if(!pre->rchild)
        {
            pre->rchild = p;
            pre->RTag = Thread;
        }
        pre = p;
        InThreading(p->rchild);
    }
}
```

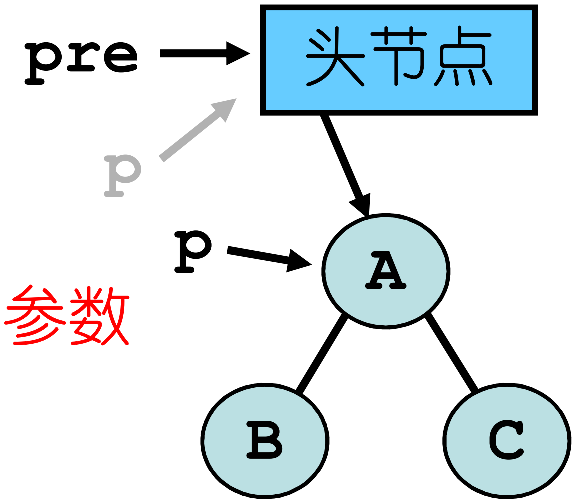
以p->lchild为参数递归调用



```
void InThreading(BiThrTree p) {
    if(p)
    {
        InThreading(p->lchild);
        if(!p->lchild)
        {
            p->lchild = pre;
            p->LRag   = Thread;
        }
        if(!pre->rchild)
        {
            pre->rchild = p;
            pre->RTag = Thread;
        }
        pre = p;
        InThreading(p->rchild);
    }
}
```

此时的p就是
原来的p->lchild

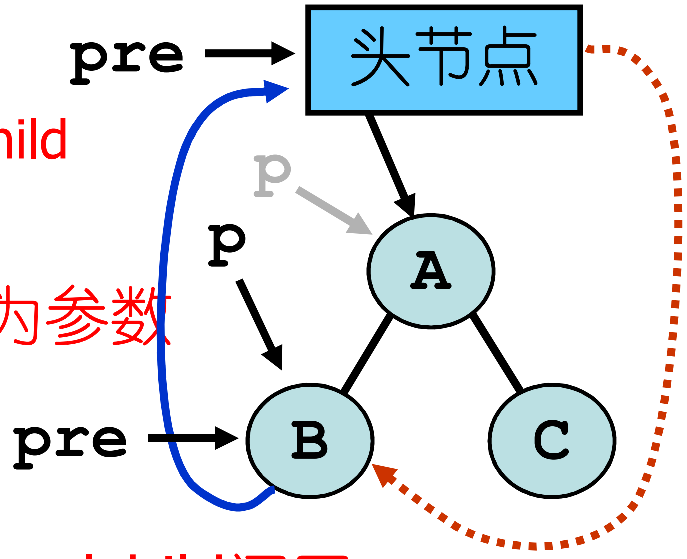
以p->lchild为参数
递归调用



```
void InThreading(BiThrTree p) {
    if(p)
    {
      InThreading(p->lchild);
      if(!p->lchild)
      {
         p->lchild = pre;
         p->LRag    = Thread;
      }
      if(!pre->rchild)
      {
         pre->rchild = p;
         pre->RTag = Thread;
      }
      pre = p;
      InThreading(p->rchild);
    }
}
```

此时的**p**就是原来的p->lchild

以p->lchild为参数递归调用



此时p->lchild闲置应作为线索，指向**pre**

此时pre->rchild闲置应作为线索，指向**p**

注意：这次操作是无效的

pre跟进

以p->rchild为参数递归调用

```
void InThreading(BiThrTree p) {
    if(p)
    {
      InThreading(p->lchild);
      if(!p->lchild)
      {
         p->lchild = pre;
         p->LRag    = Thread;
      }
      if(!pre->rchild)
      {
         pre->rchild = p;
         pre->RTag = Thread;
      }
      pre = p;
      InThreading(p->rchild);
    }
}
```

调用完毕

p->lchild不闲置

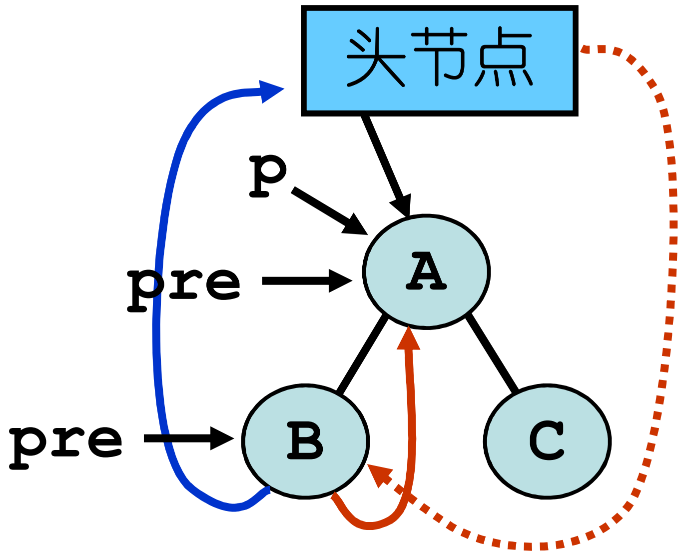
pre->rchild闲置

应作为索引，指向**p**

pre跟进

以p->rchild为参数

递归调用



```
void InThreading(BiThrTree p) {
    if(p)
    {
      InThreading(p->lchild);
      if(!p->lchild)
      {
         p->lchild = pre;
         p->LRag   = Thread;
      }
      if(!pre->rchild)
      {
         pre->rchild = p;
         pre->RTag = Thread;
      }
      pre = p;
      InThreading(p->rchild);
    }
}
```

此时的**p**就是原来的p->lchild

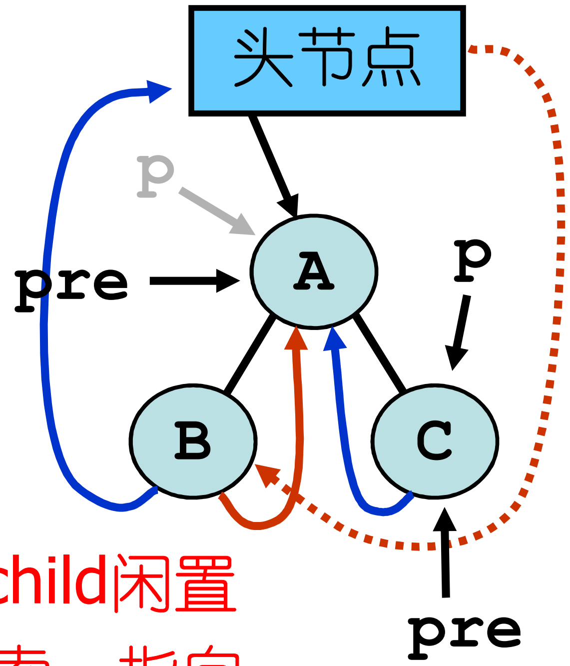
以p->lchild为参数递归调用

此时p->lchild闲置应作为线索，指向**pre**

pre->rchild不闲置

pre跟进

以p->rchild为参数递归调用



```
Status InOrderThreading(BiThrTree &Thrt,
                        BiThrTree T)
{
        ... ...
        InThreading(T);
        pre->rchild = Thrt;
        pre->RTag = Thread;
        Thrt->rchild = pre;
    }
    return OK;
}
```

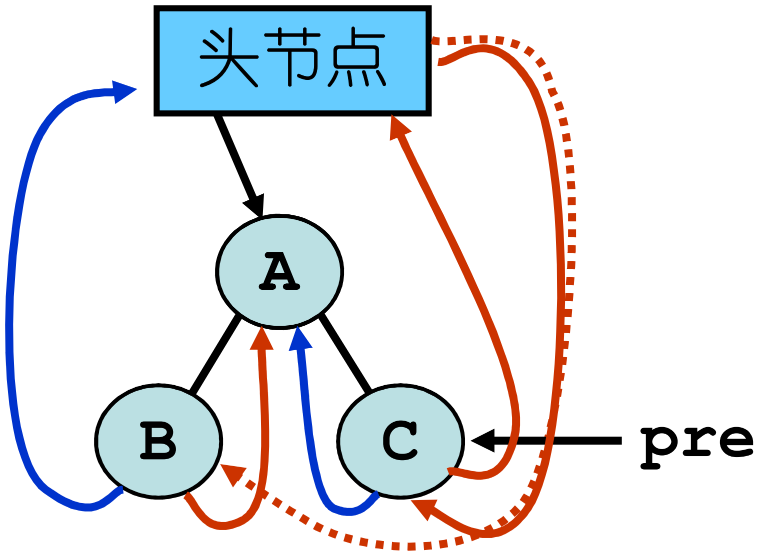
调用完毕

pre此时指向遍历序列的最后一个节点，

头节点的rchild域应指向遍历序列的最后节点

pre->rchild域应指向头节点

程序结束



线索二叉树

- 本节小结
  - 线索二叉树的原理
    - 手工能对二叉树进行线索化
    - 掌握线索二叉树的存储结构
  - 线索二叉树的遍历
    - 能够写出中序线索二叉树的遍历算法
    - 了解先/后序线索二叉树的遍历
  - 二叉树的线索化
    - 了解算法

线索二叉树

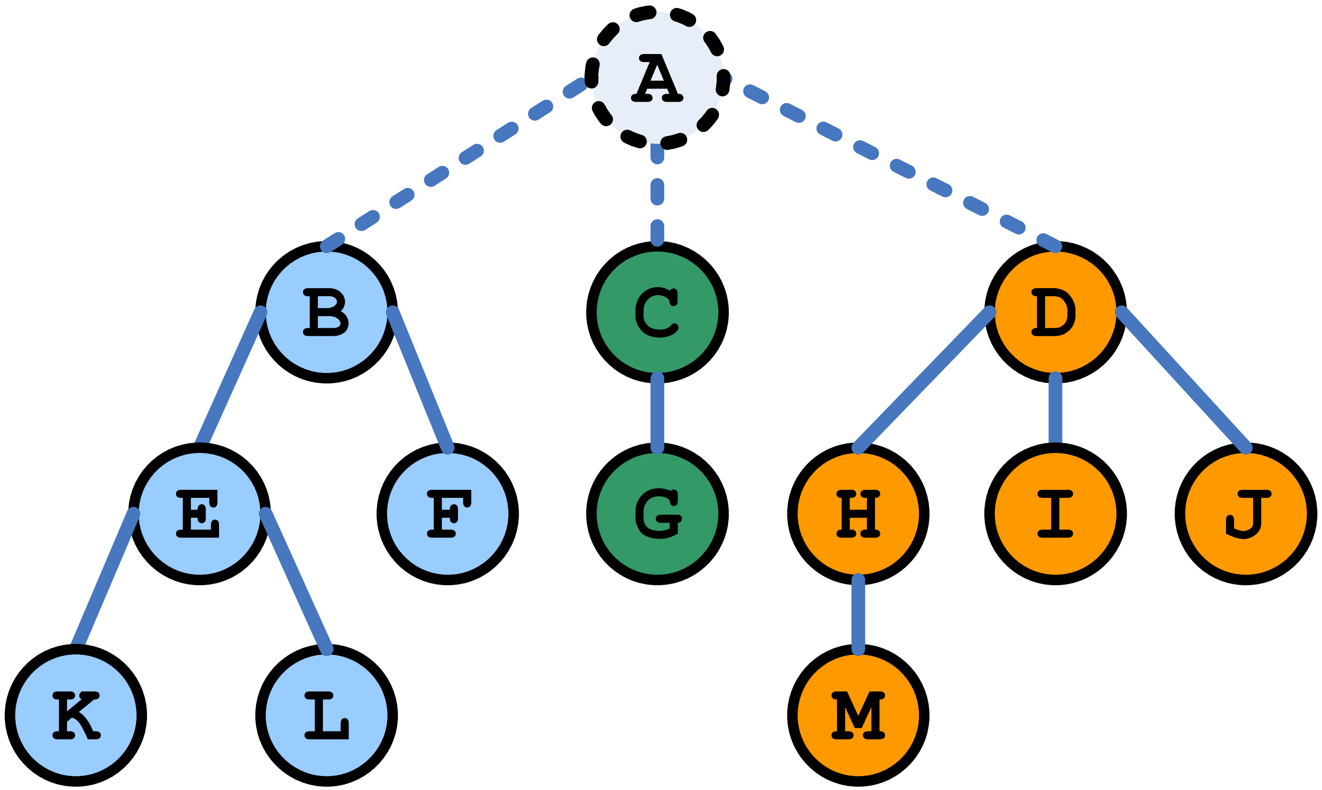
- 作业2
  - 习题集：**6.3**

树和森林：回顾

- 树：
  - n (n>=0) 个结点组成的有限集合
  - 如果 $n=0$，称为空树
  - 如果 $n>0$，则
    - 有一个特定的称之为根 (**root**) 的结点，它只有直接后继，但没有直接前驱
    - 除根以外的其它结点划分为 m (m>=0) 个互不相交的有限集合 $T_0, T_1, \ldots, T_{m-1}$，每个集合又是一棵树，并且称之为根的子树

树和森林：回顾

- 森林：
  - m(m>=0)棵互不相交的树的集合



树的存储结构：孩子表示法

- 孩子表示法
  - 每个结点可以有多个孩子
  - 方法一：

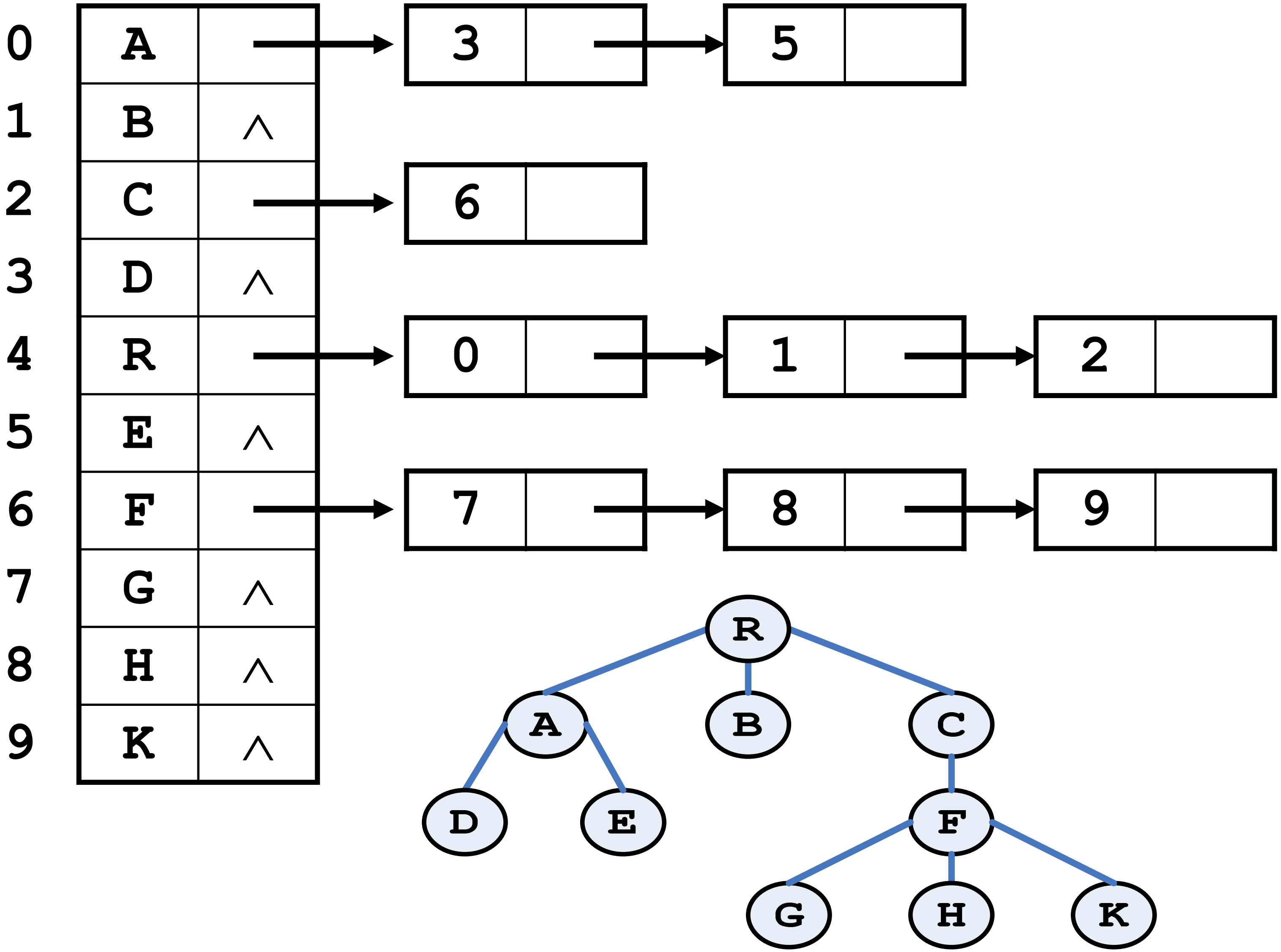
data	child1	child2	child3	...	childn

或者：

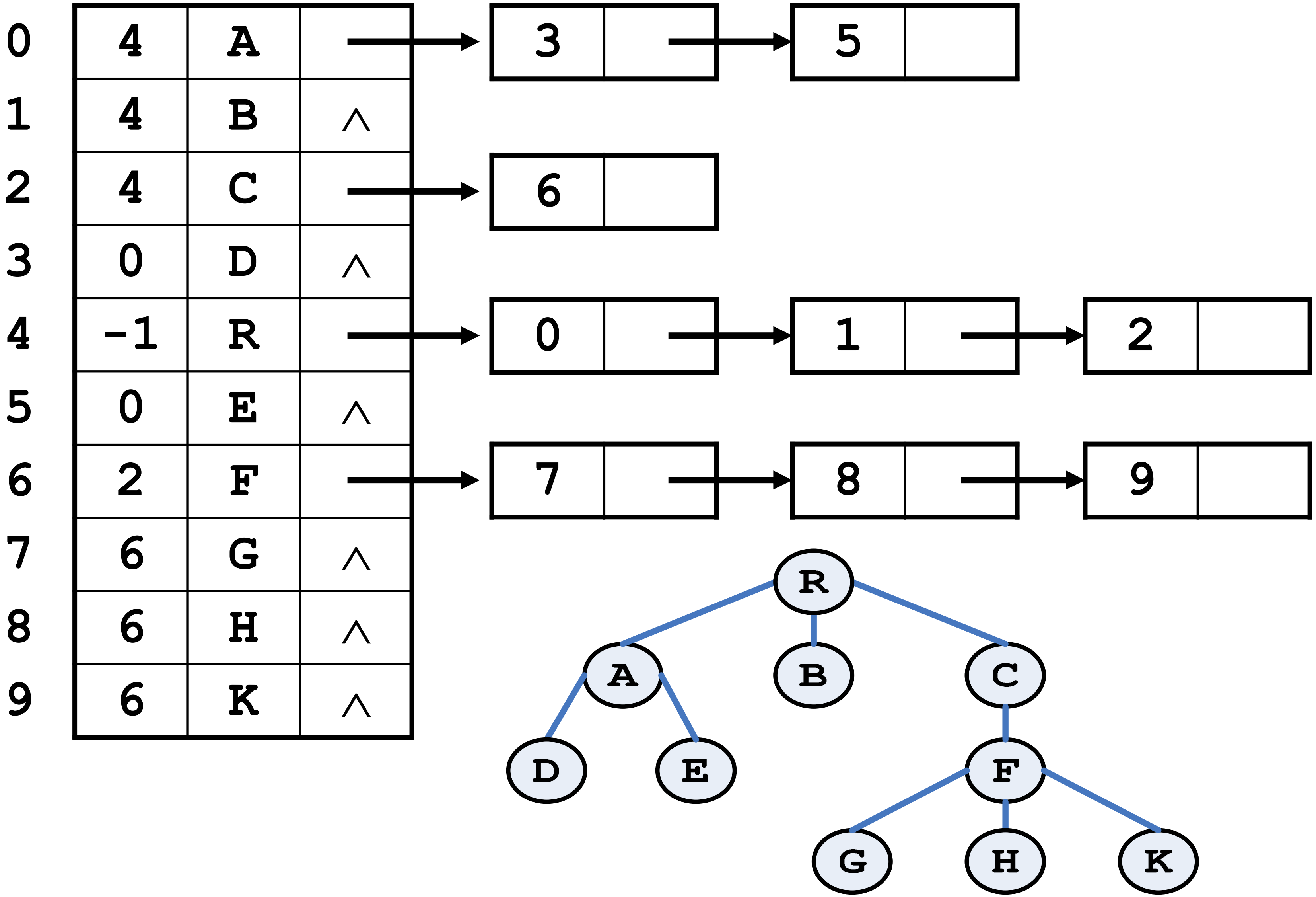
data	degree	child1	child2	...	childn

- 浪费空间！

-方法二

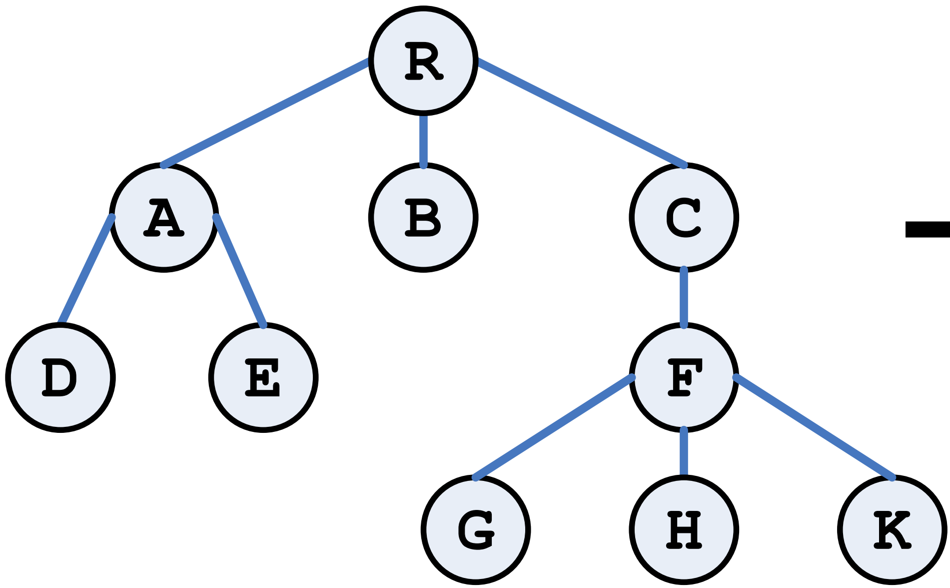


–还可以增加双亲信息 (孩子-双亲表示法)



树的存储结构：双亲表示法

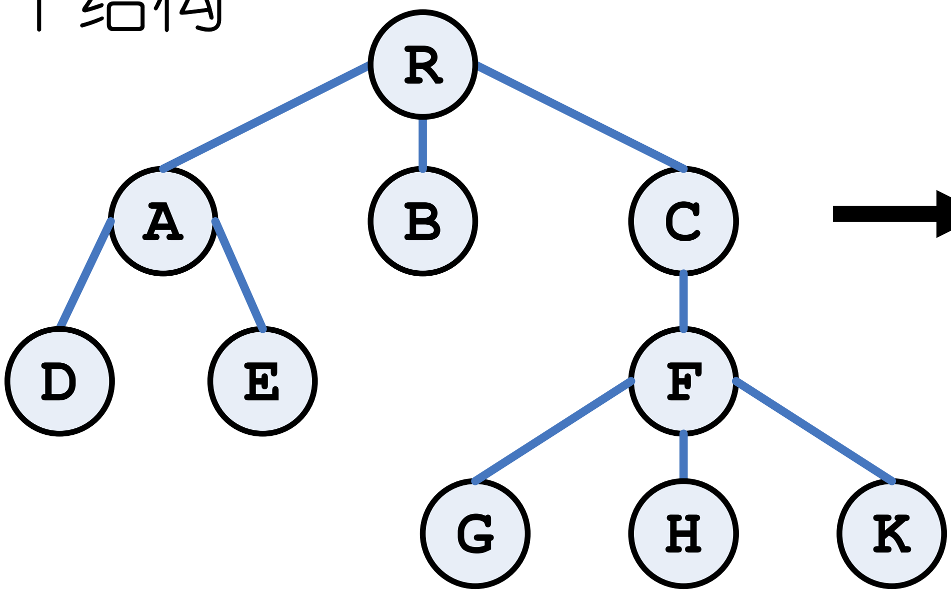
- 双亲表示法
  - 树中一个结点的孩子的数量不定
  - 但是双亲却只有一个
  - 所以保存每个结点的双亲



	结点	双亲
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

树的存储结构：双亲表示法

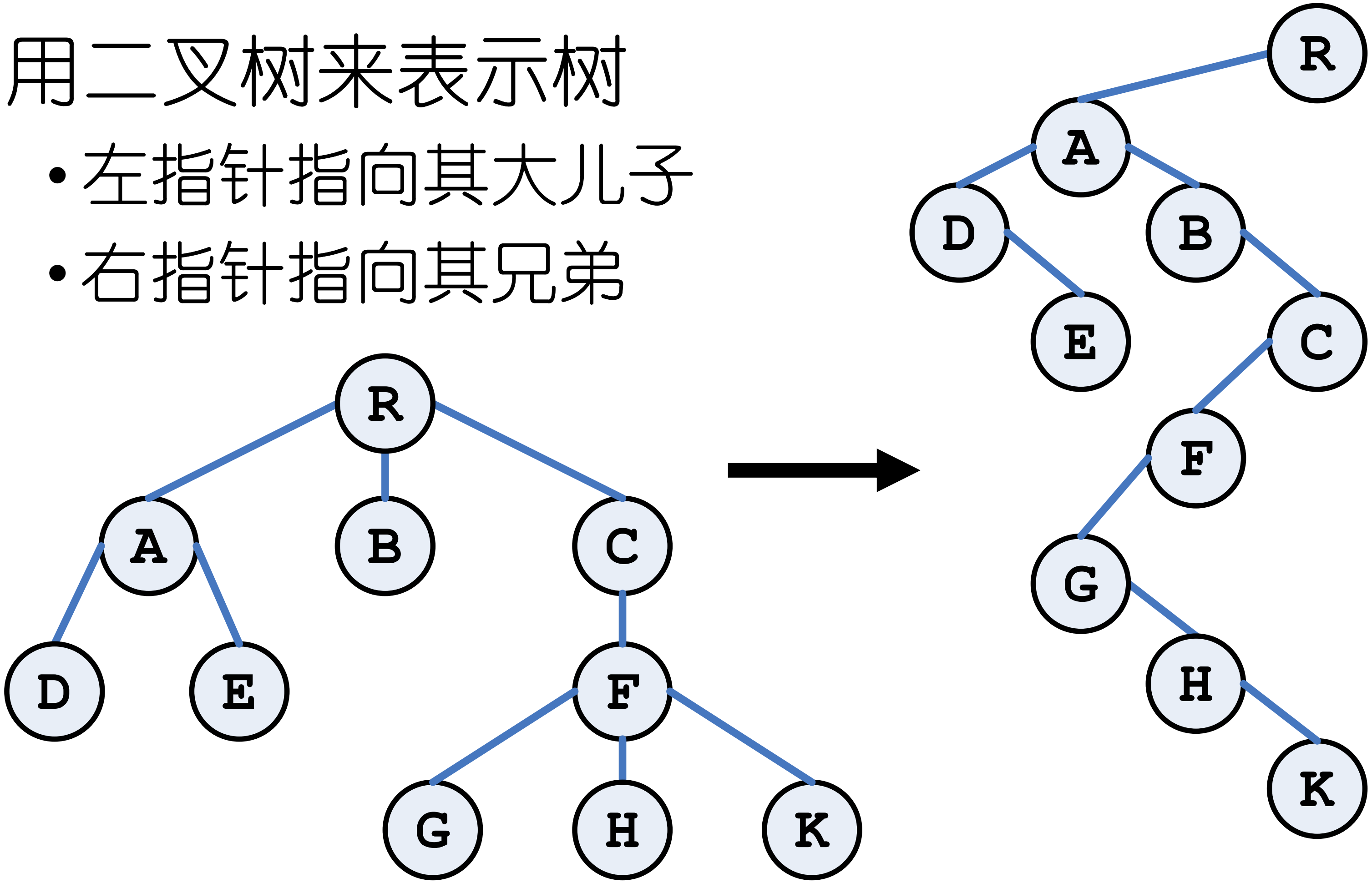
– 优点
  - 查找双亲、树根操作很快
– 缺点
  - 查找孩子操作很慢，需要遍历整个结构



	结点	双亲
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

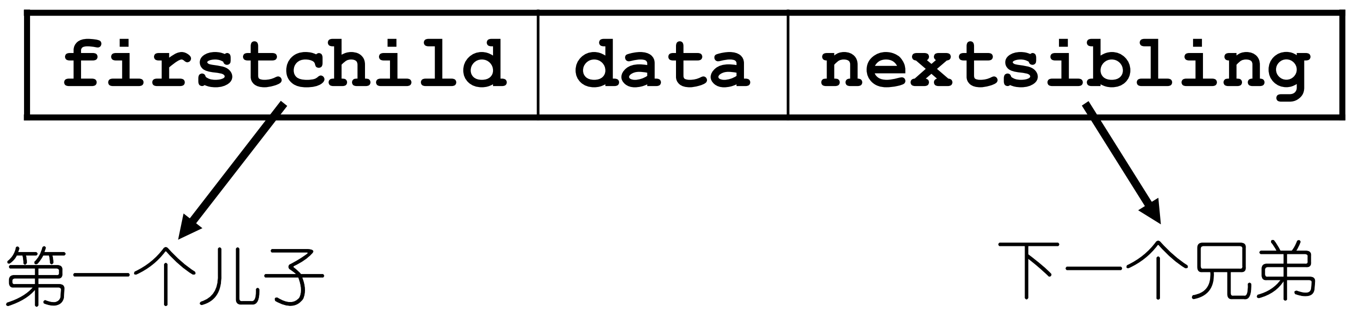
树的存储结构：孩子兄弟表示法

- 即左孩子、右兄弟表示法
  - 用二叉树来表示树
    - 左指针指向其大儿子
    - 右指针指向其兄弟

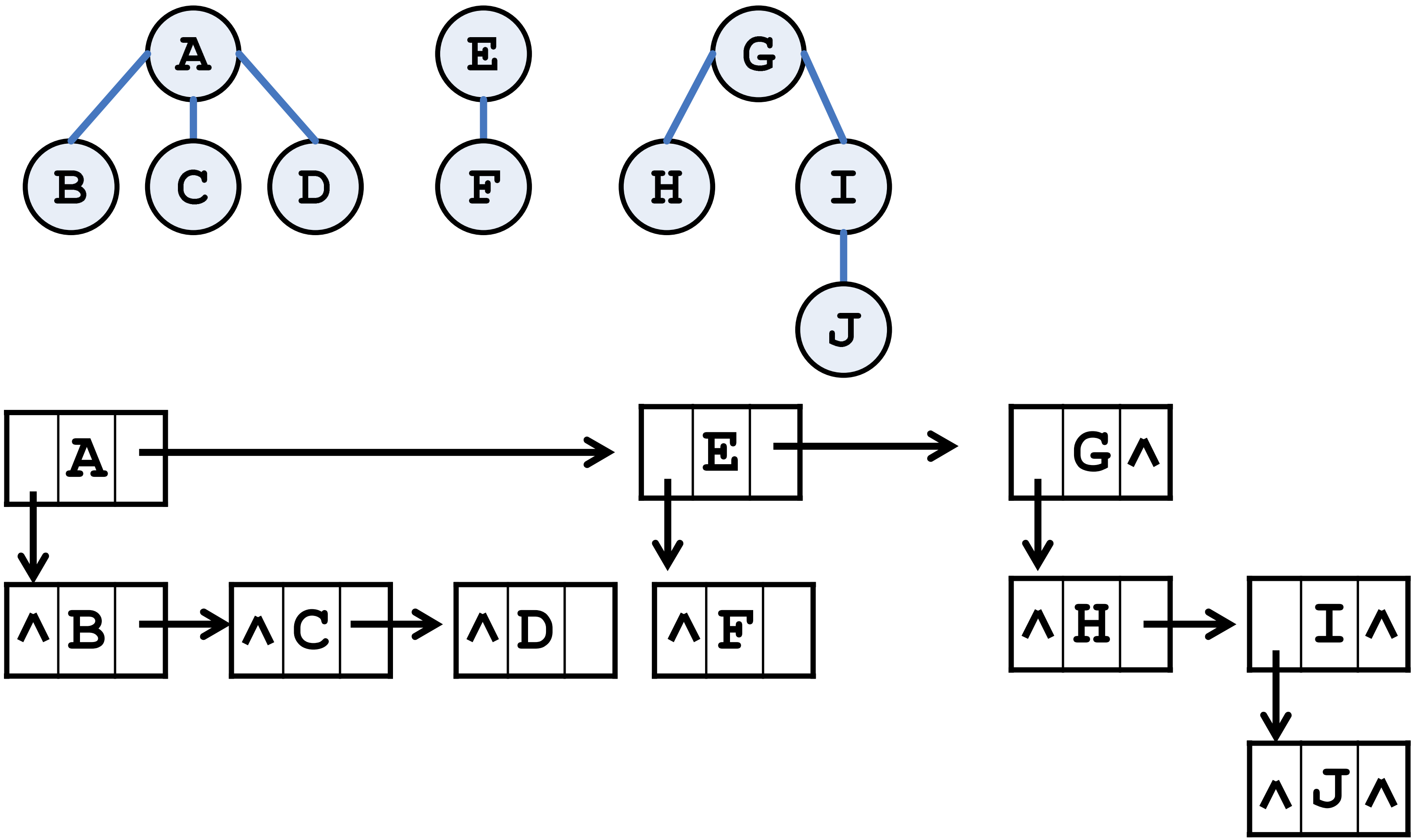


树的存储结构：孩子兄弟表示法

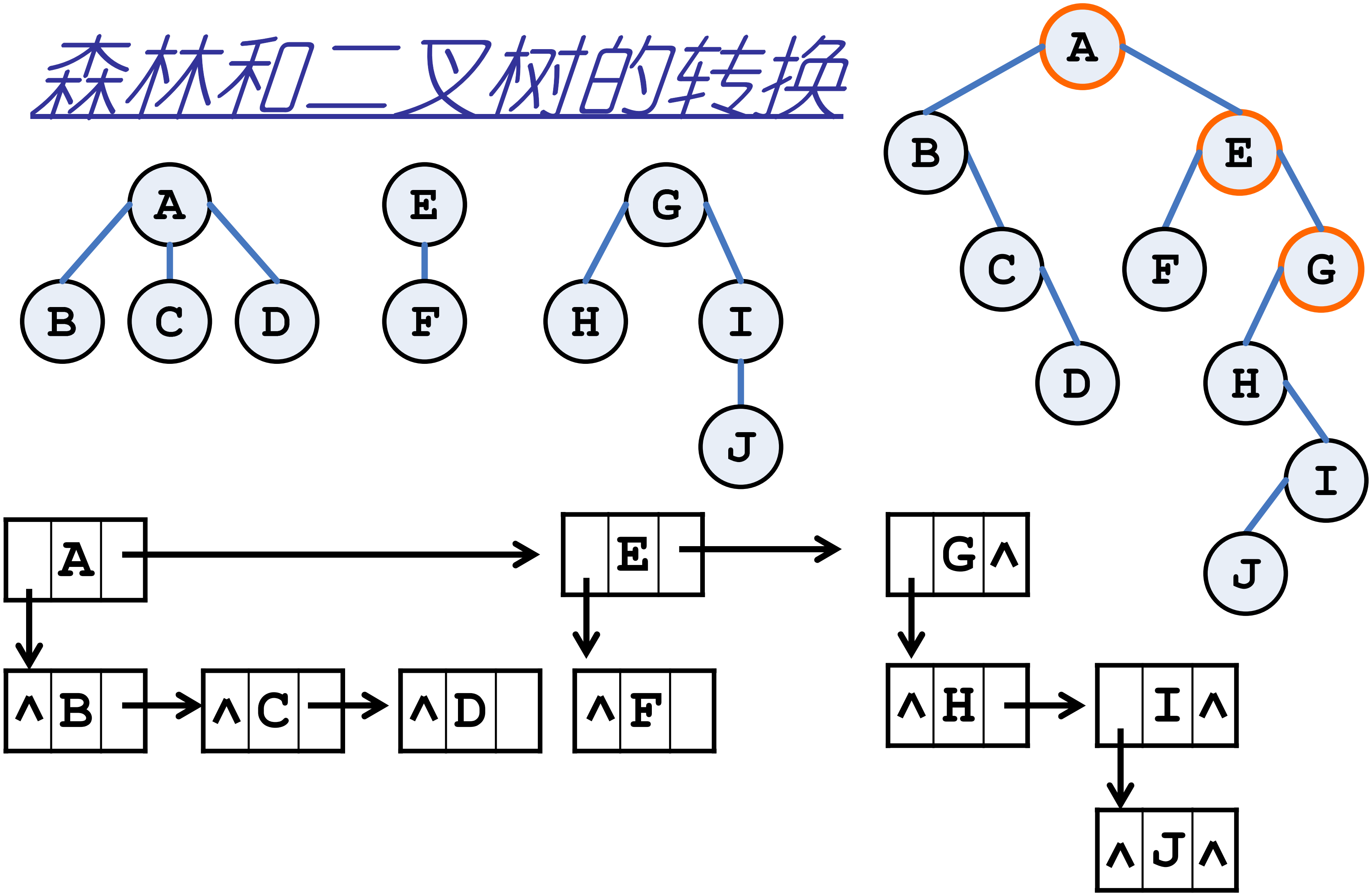
```
typedef struct _csNode {
  ElemType data;
  struct _csNode *firstchild,
                 *nextsibling;
}CSNode;
```



森林和二叉树的转换



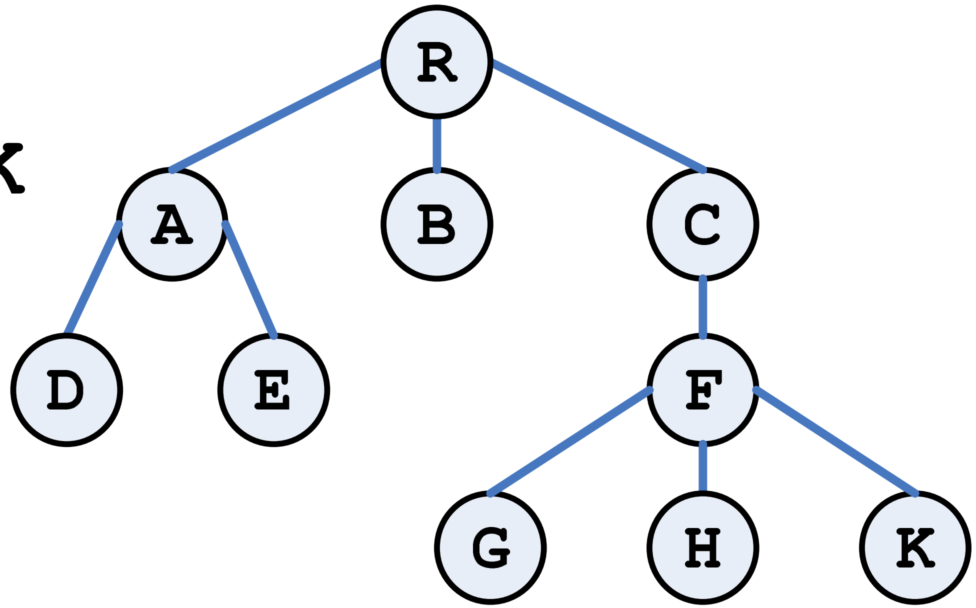
森林和二叉树的转换



树（森林）的左孩子右兄弟　　二叉树的左右孩子链表

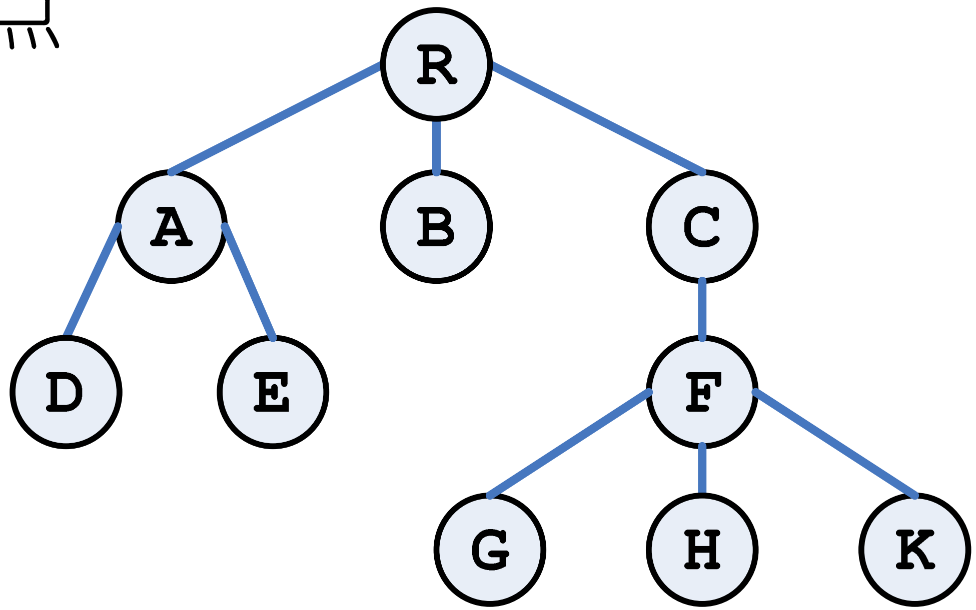
树和森林的遍历：树的遍历

- 树的先根遍历
  - 若森林非空
  - 先访问树的根结点
  - 再从左至右，先根遍历树根的每棵子树
  - 比如右图
    - **RADEBCFGHK**



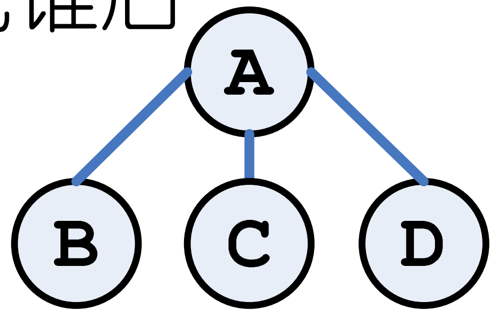
树和森林的遍历：树的遍历

- 树的后根遍历
  - 若森林非空
  - 先从左至右，后根遍历树根的每棵子树
  - 再访问树的根结点
  - 比如右图
    - **DEABGHKFCR**

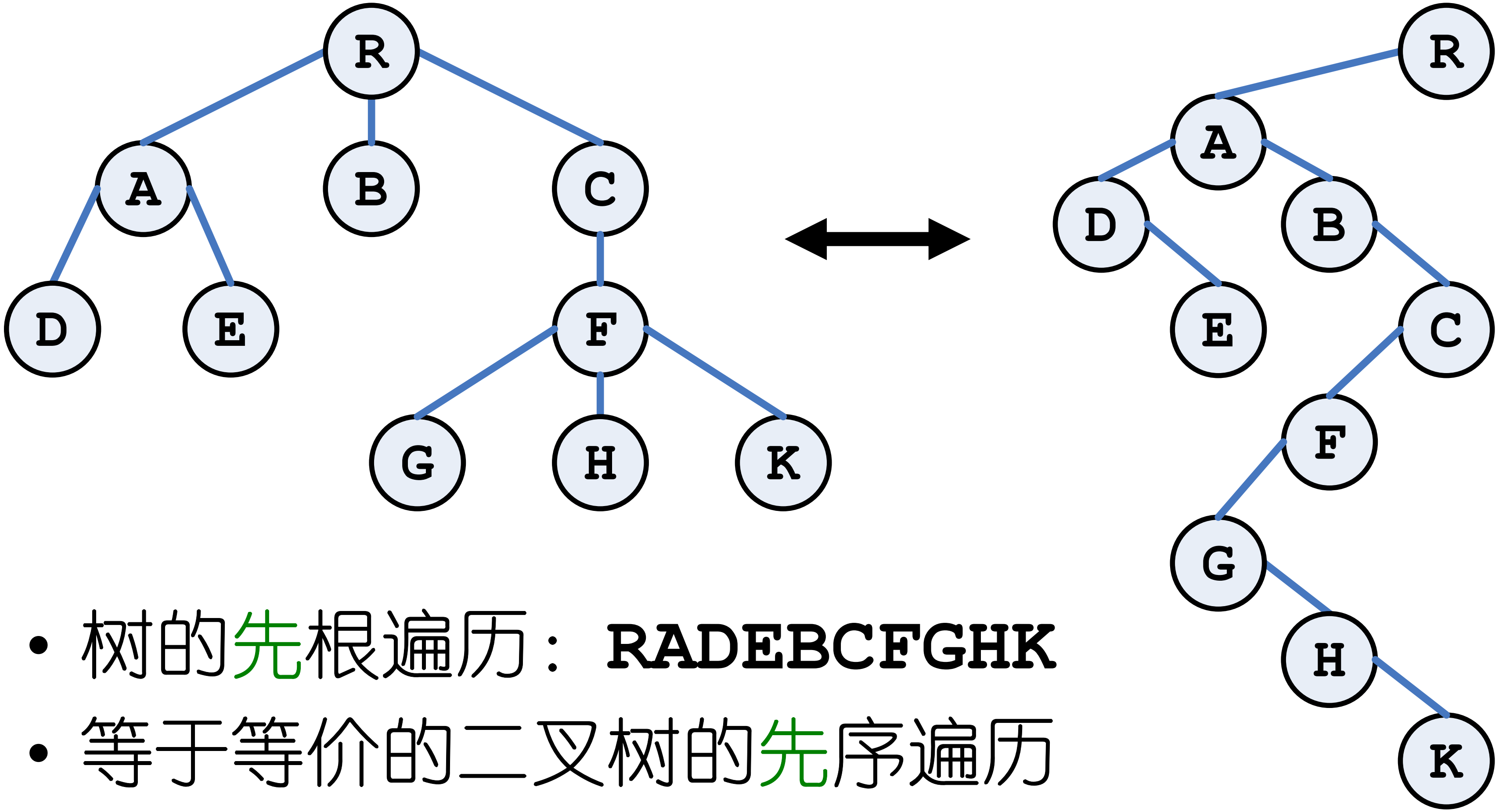


树和森林的遍历：树的遍历

- 问题
  - 树的中根遍历呢？
  - “中”在哪里？
- 注意：
  - 树的遍历中的“先根”“后根”指的是树根和它的孩子相比，谁先谁后

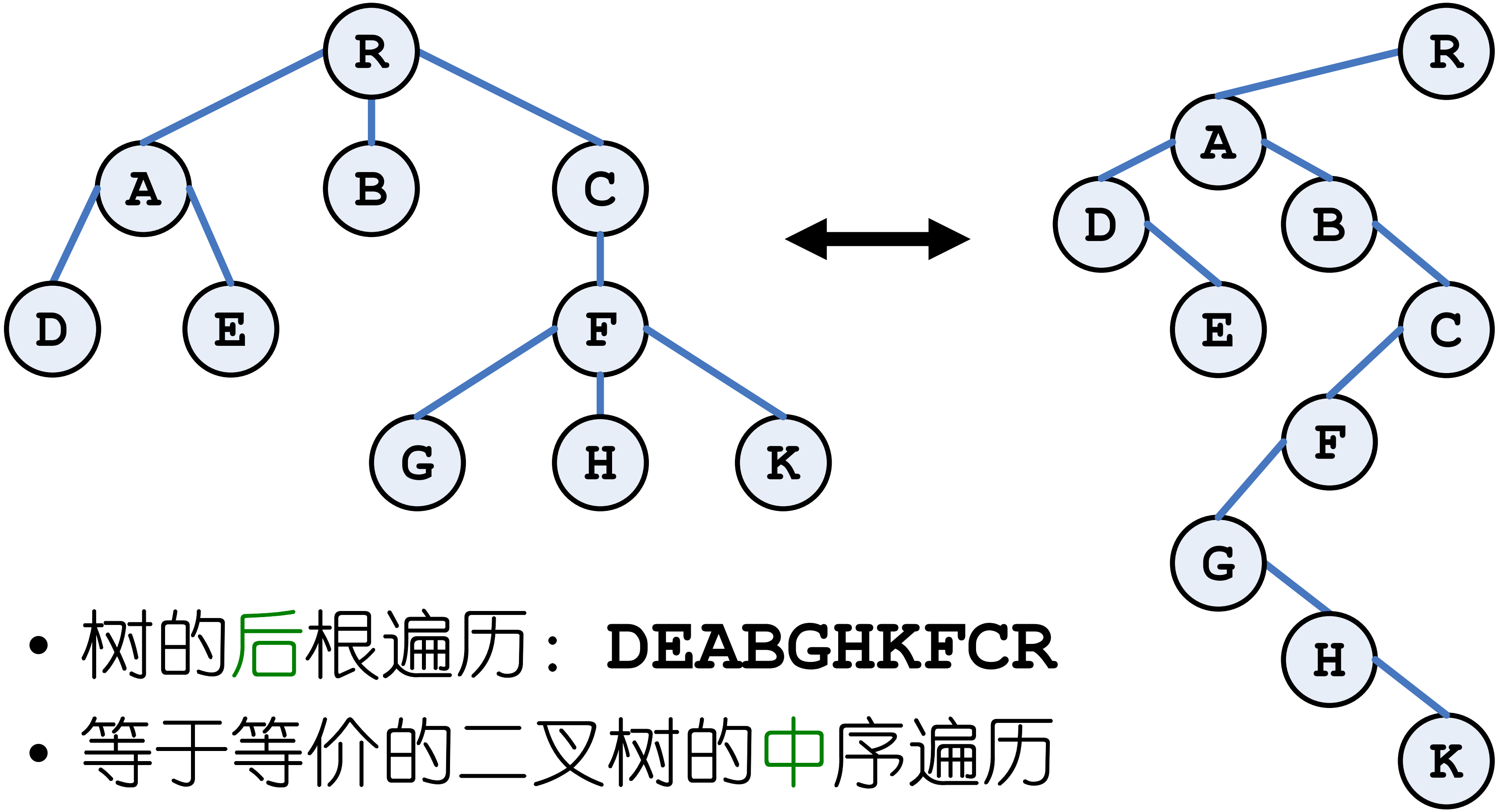


树的遍历和相应二叉树的遍历的关系



- 树的先根遍历： **RADEBCFGHK**
- 等于等价的二叉树的先序遍历

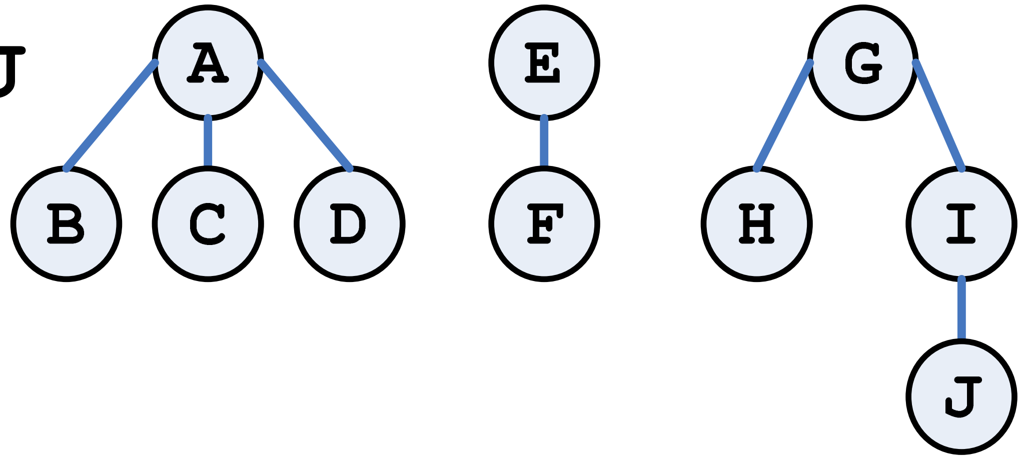
树的遍历和相应二叉树的遍历的关系



- 树的后根遍历： **DEABGHKFCR**
- 等于等价的二叉树的中序遍历

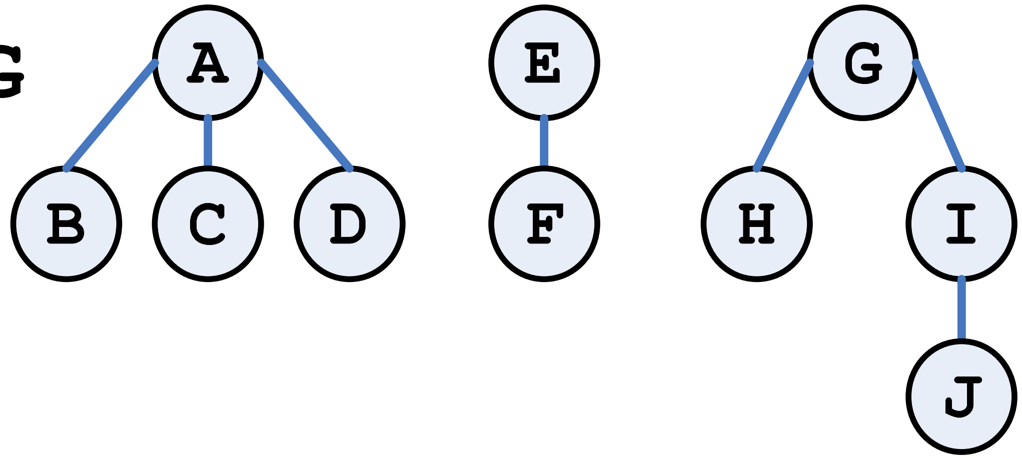
树和森林的遍历：森林的遍历

- 森林的先序遍历
  - 先访问森林中第一棵树的树根
  - 再先序遍历第一棵树中树根的子树森林
  - 最后先序遍历剩余的树构成的森林
  - 比如右图
    - **ABCDEFGHIJ**



树和森林的遍历：森林的遍历

- 森林的中序遍历
  - 先中序遍历第一棵树中树根的子树森林
  - 再访问森林中第一棵树的树根
  - 最后中序遍历剩余的树构成的森林
  - 比如右图
    - **BCDAFEHJIG**



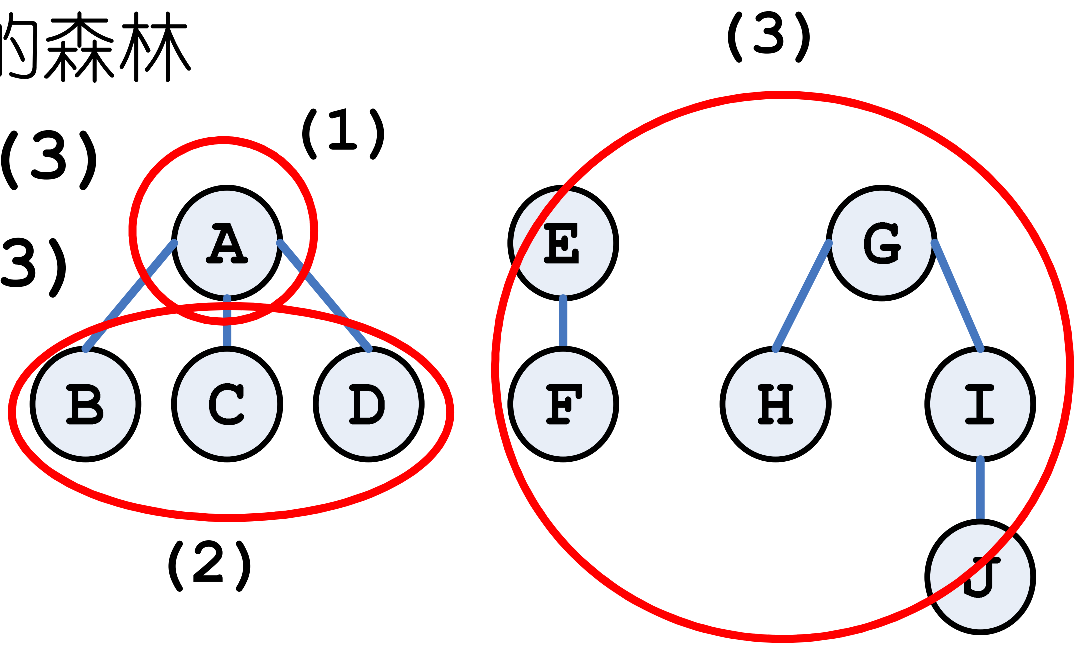
树和森林的遍历：森林的遍历

- 注意：
  - 森林的遍历中的“先”“中”指的以下三者：

  (1) 第一棵树的树根

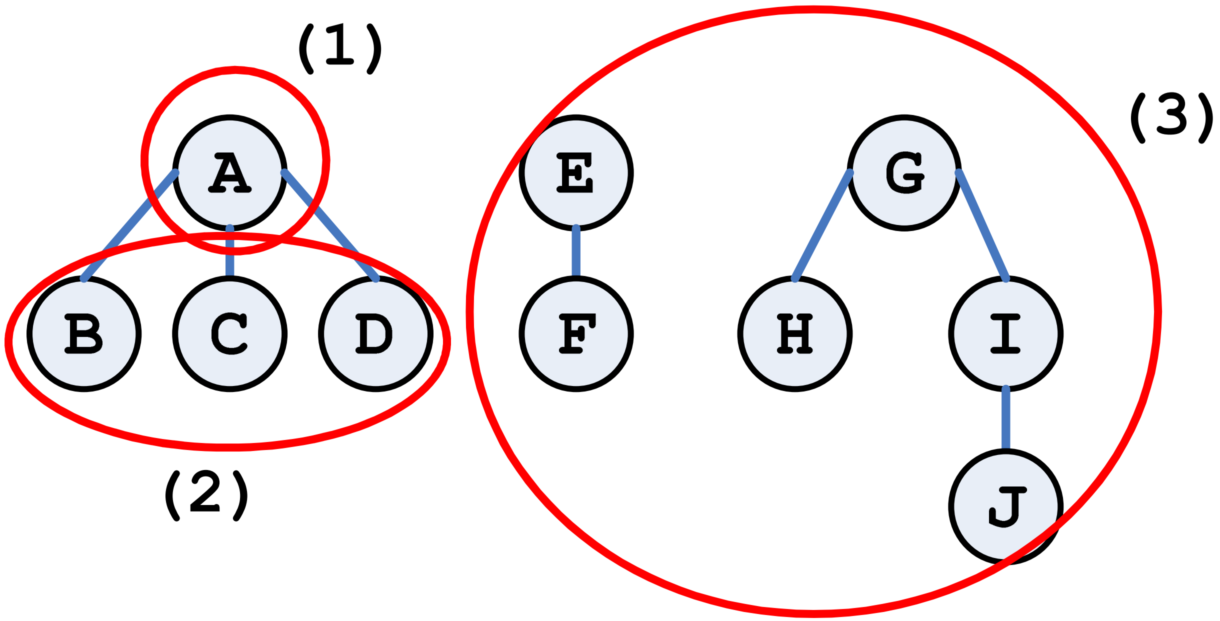
  (2) 第一棵树的树根的子树森林

  (3) 剩余的树构成的森林
  - “先”：**(1) (2) (3)**
  - "中”：**(2) (1) (3)**



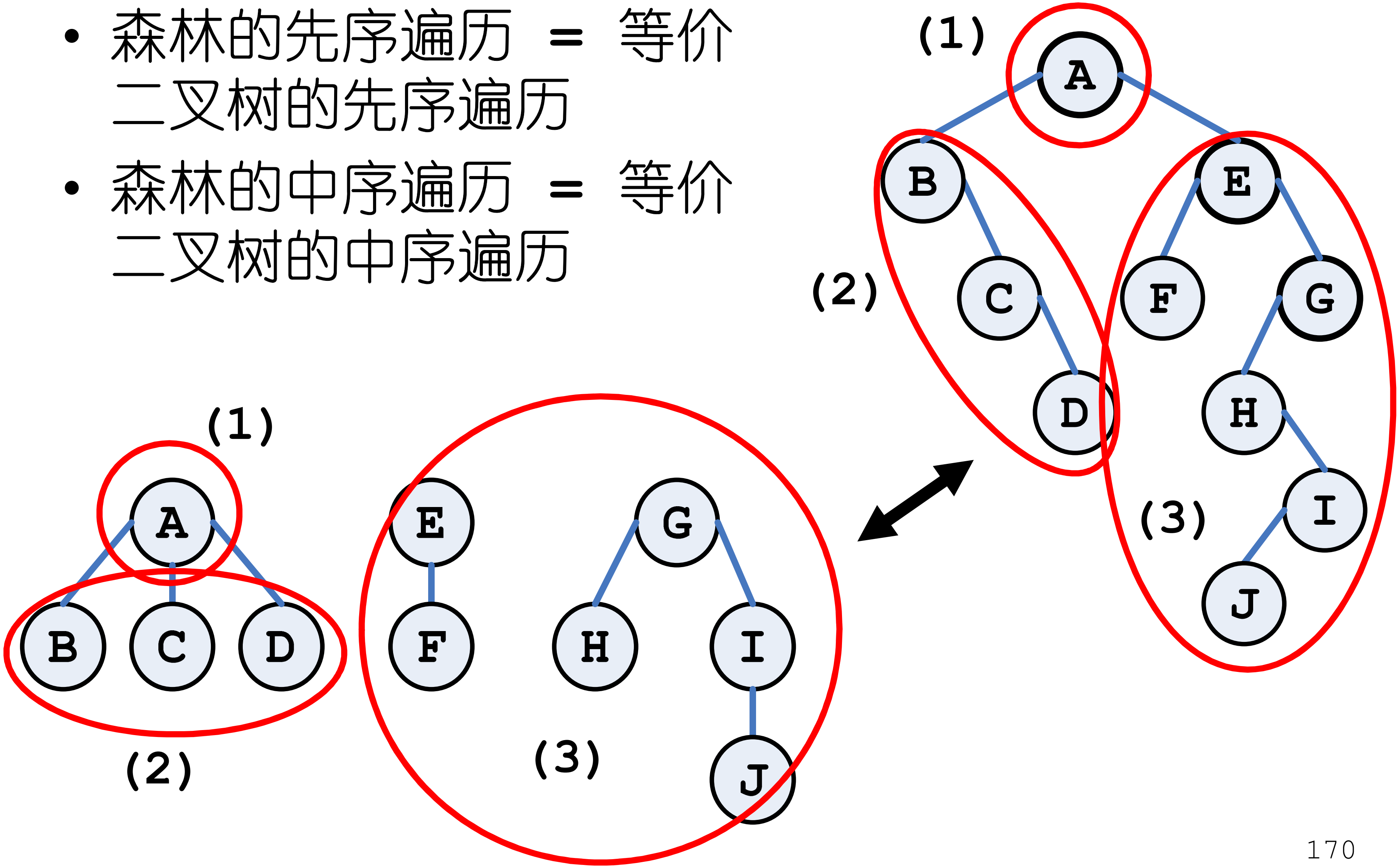
树和森林的遍历：森林的遍历

- 理解
  - 森林的先序遍历相当于对每棵树依次进行树的先根遍历
  - 森林的中序遍历相当于对每棵树依次进行树的后根遍历



森林的遍历及其等价二叉树的遍历

- 森林的先序遍历 = 等价二叉树的先序遍历
- 森林的中序遍历 = 等价二叉树的中序遍历



树和森林

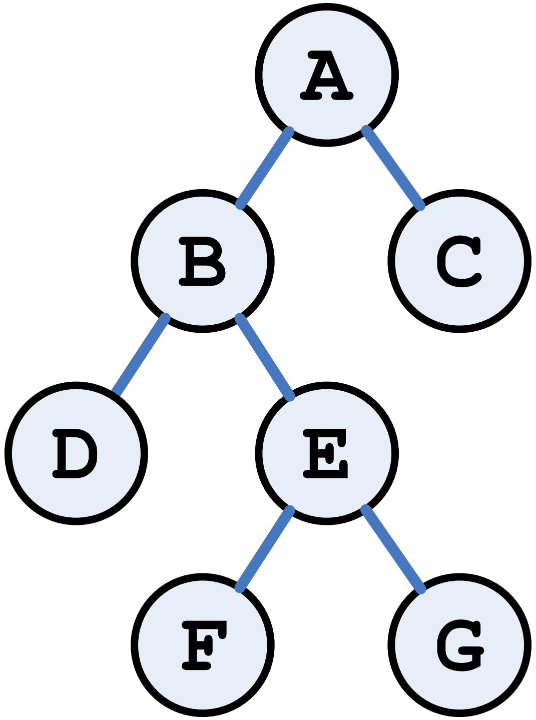
- 本节小结
  - 树和森林的概念
  - 树的表示：
    - 手工能画出示意图
  - 树、森林 **<-->** 二叉树
    - 手工完成
  - 树和森林的遍历
    - 手工写出遍历结果

树和森林

- 课后练习
  - 习题集**P40：6.19~6.22**
  - 有答案，不用交

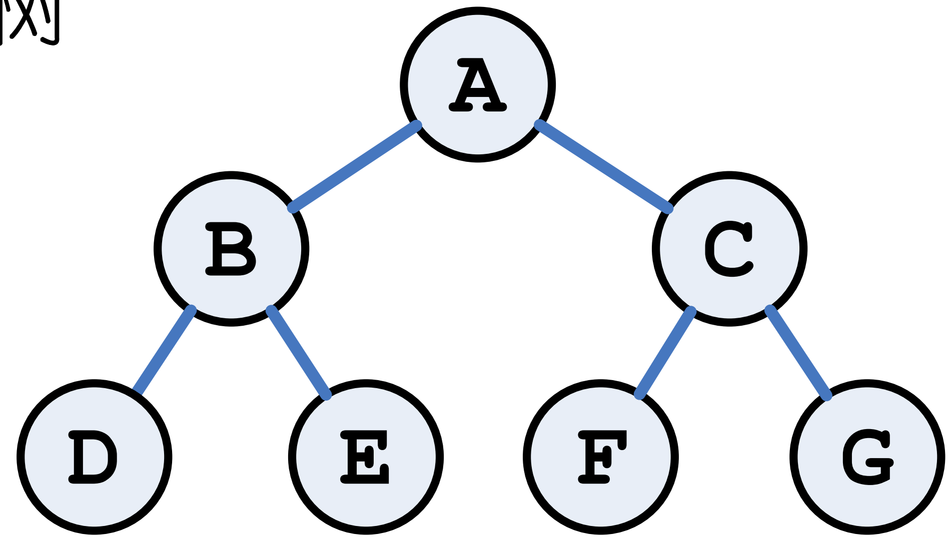
赫夫曼树及其应用

- 路径
  - 从一个结点到另一个结点的分支构成
- 路径长度
  - 路径上的分支的个数
- 树的路径长度
  - 从树根到每一个结点的路径长度之和



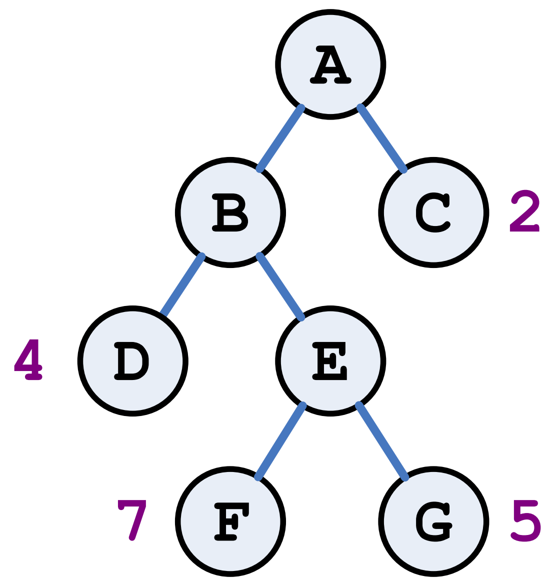
赫夫曼树及其应用

- 以下我们只讨论二叉树
- 问题：
  - 树的路径长度最短的是哪一种？
  - 完全二叉树



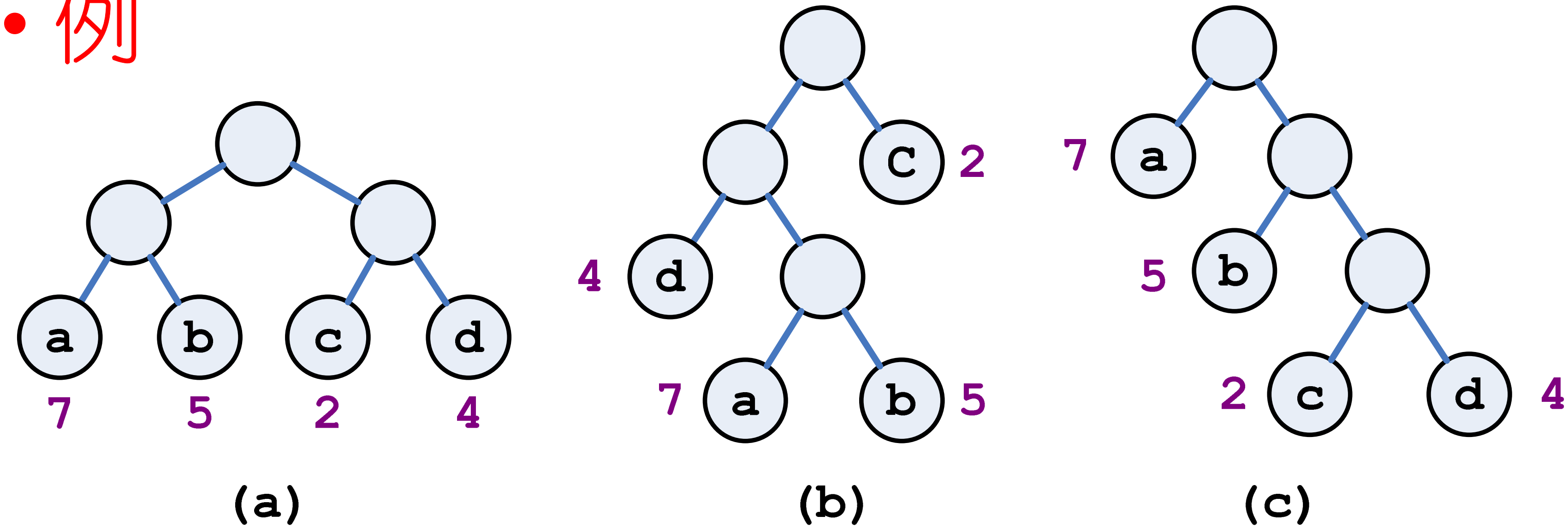
赫夫曼树及其应用

- 权(**weight**)
  - 给结点赋予一定的数值
- 结点的带权路径长度
  - 从树根到该节点的路径长度×该结点的权
- 树的带权路径长度
  - **Weighted Path Length**
  - 所有叶结点的带权路径长度之和
  - $\mathbf{WPL} = \sum_{k=1}^{n} w_k l_k$



赫夫曼树及其应用

- 例



$$WPL_a = 7\times 2 + 5\times 2 + 2\times 2 + 4\times 2 = 36$$

$$WPL_b = 7\times 3 + 5\times 3 + 2\times 1 + 4\times 2 = 46$$

$$WPL_c = 7\times 1 + 5\times 2 + 2\times 3 + 4\times 3 = 35$$

赫夫曼树及其应用

- 最优二叉树
  - 又称赫夫曼树(**Huffman**)
  - 有**n**个权值$\{w_1, w_2, ..., w_n\}$
  - 构造一棵有**n**个叶结点的二叉树
  - 每个叶结点的权值为w_i
  - 其中**WPL**最小的那一棵称作最优二叉树，即赫夫曼树
- 最优又能怎样？

赫夫曼树及其应用

- 例
  - 一个判断成绩等级的程序

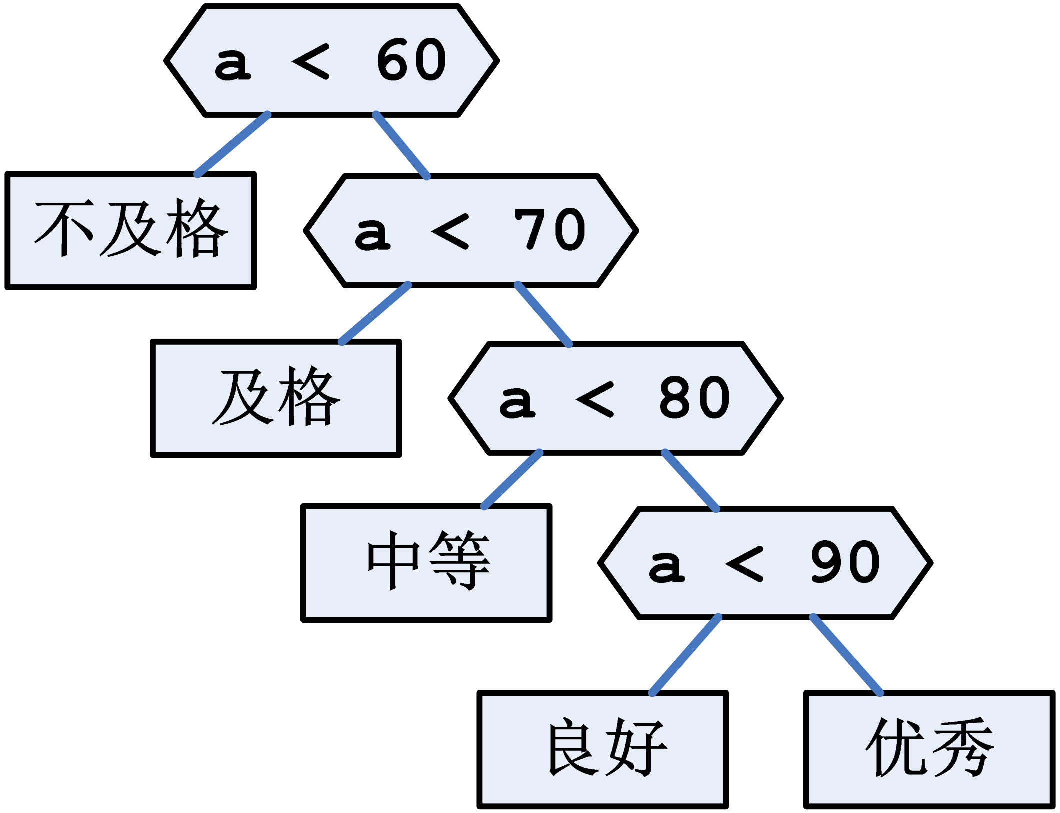
```
if(a < 60) b = "不及格"
else if(a < 70) b = "及格"
     else if(a < 80) b = "中等"
          else if(a < 90) b = "良好"
               else b = "优秀"
```

  - 一个**a**最多要经过**4**次比较才能得出**b**
  - 我们当然希望比较的总次数最小

- 假设有如下统计数据

分数	0-59	60-69	70-70	80-89	90-100
概率	0.05	0.15	0.40	0.30	0.10

  - 原判定树为：



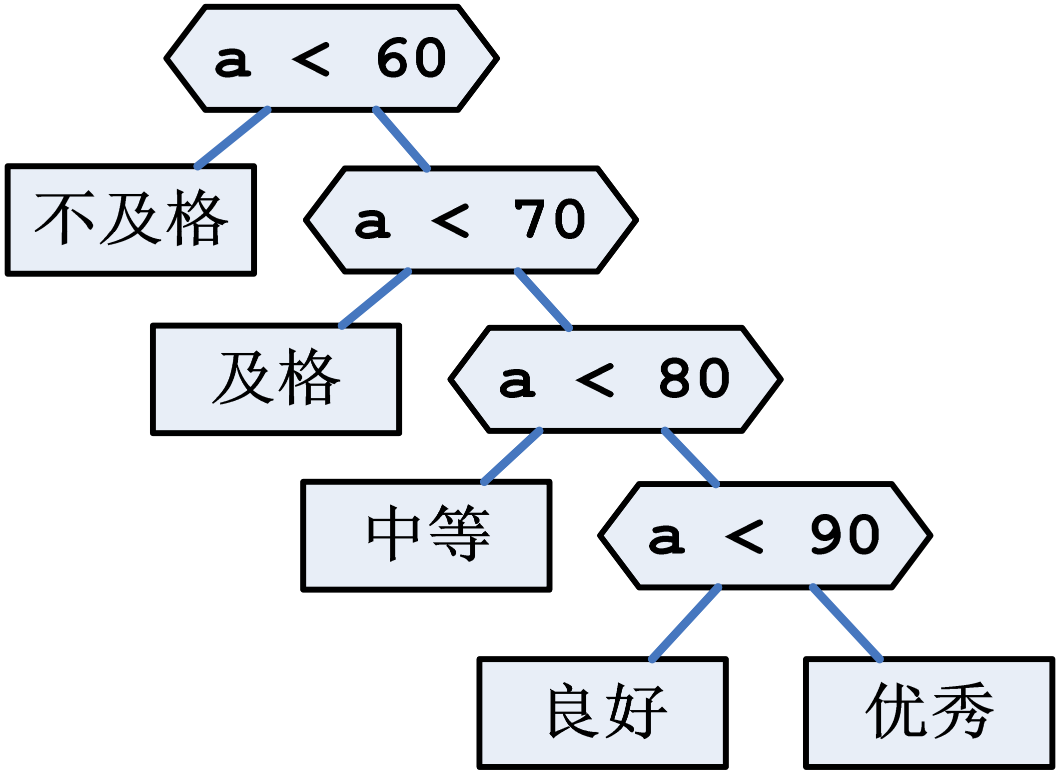
– 若一共有10000个输入数据

– 则总共的比较次数

= 500*1 + 1500*2 + 4000*3 +

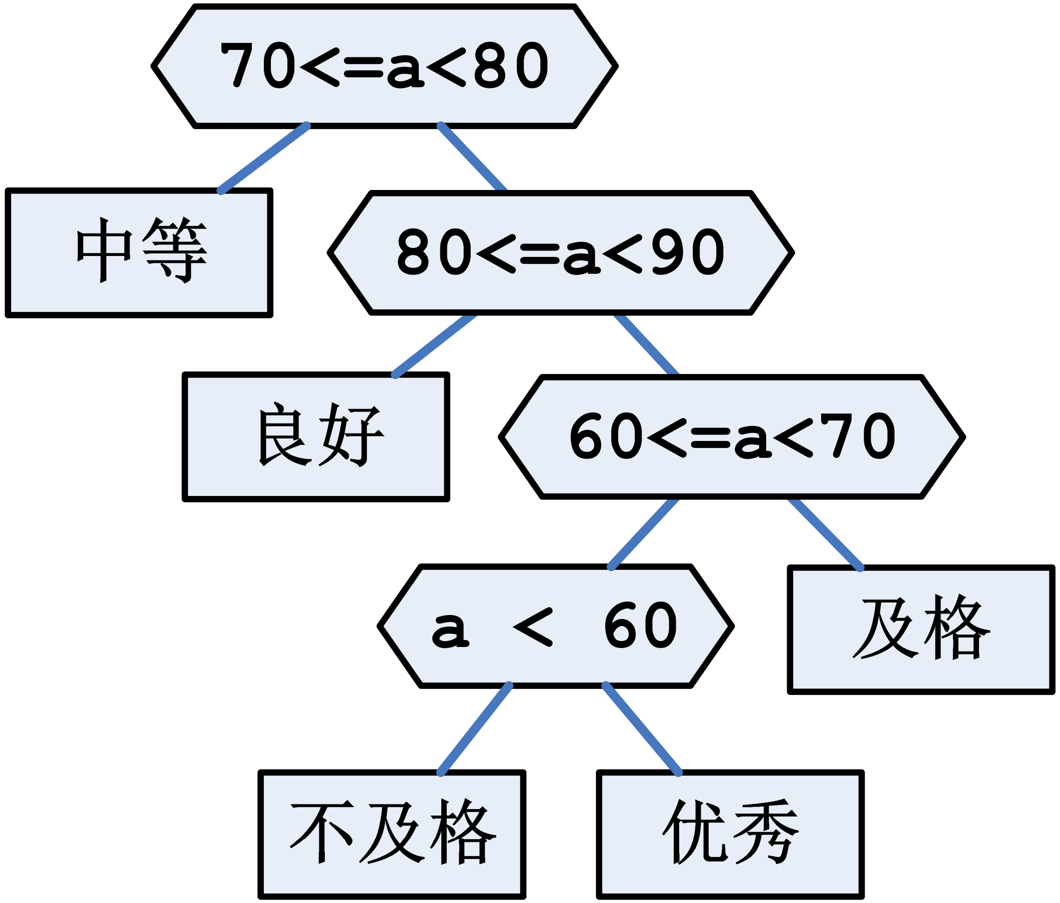
3000*4 + 1000*4

= 31500

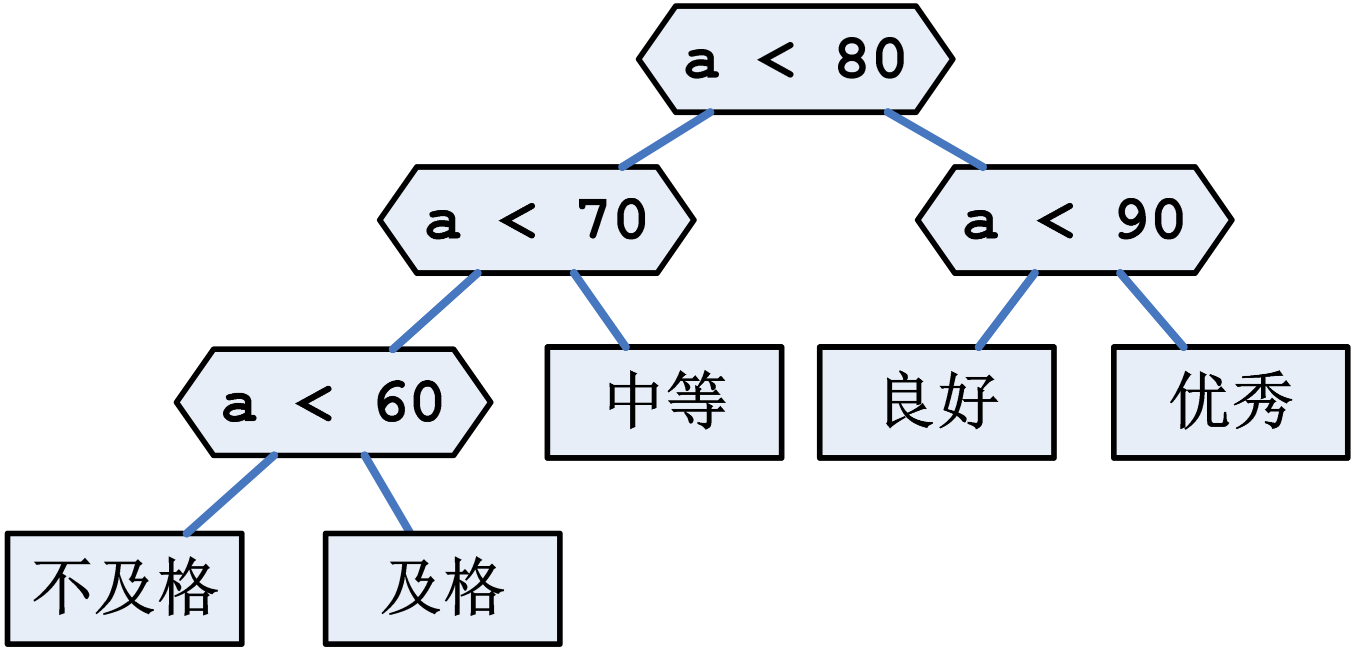


- 构造一棵赫夫曼树
  - 有**5**个叶结点，权值分别为：

0.05, 0.15, 0.4, 0.3, 0.1



- 根据这棵赫夫曼树导出新的判定树：



  - 总的比较次数

    = 500*3 + 1500*3 + 4000*2 + 3000*2 + 1000*2

    = 22000

赫夫曼树及其应用

- 赫夫曼树的构造算法

(1) 根据给定的**n**个权值{**w1, w2, …, wn**}构成**n**棵二叉树的集合**F={T1, T2, …, Tn}**，其中每棵二叉树只含一个带权的根结点，其左右子树均空

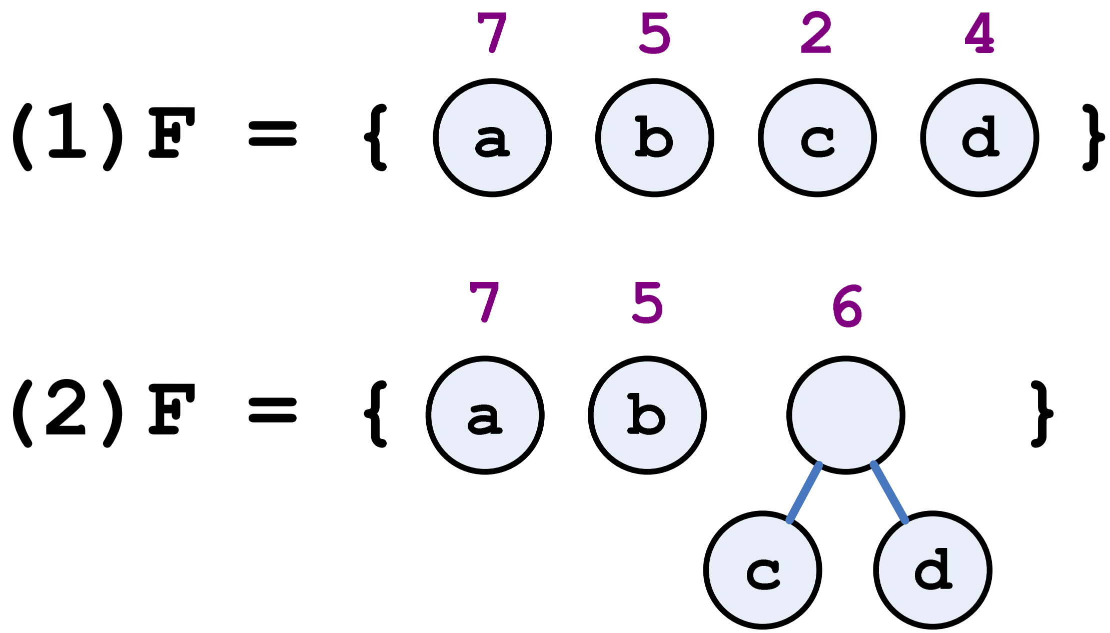
(2) 在**F**中选取两棵根结点的权值最小的树作为左右子树，构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树根结点的权值之和

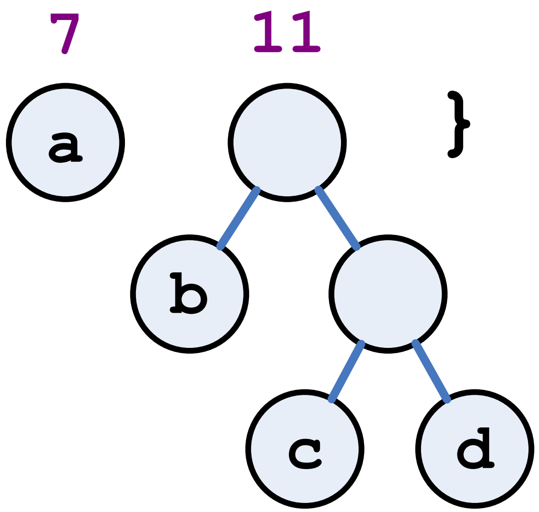
(3) 在**F**中删除这两棵二叉树，同时将新得到的二叉树加入**F**

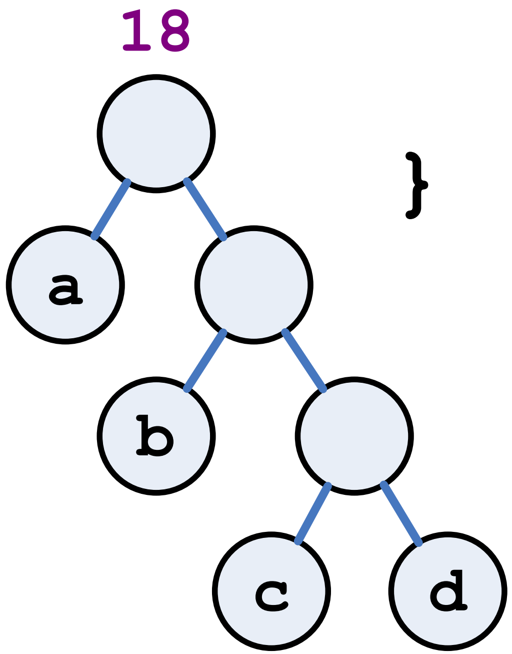
(4) 重复(2)和(3)，直到**F**只含一棵二叉树

赫夫曼树及其应用

- 例：
  - 设结点a,b,c,d的权值分别为7,5,2,4,试构造赫夫曼树

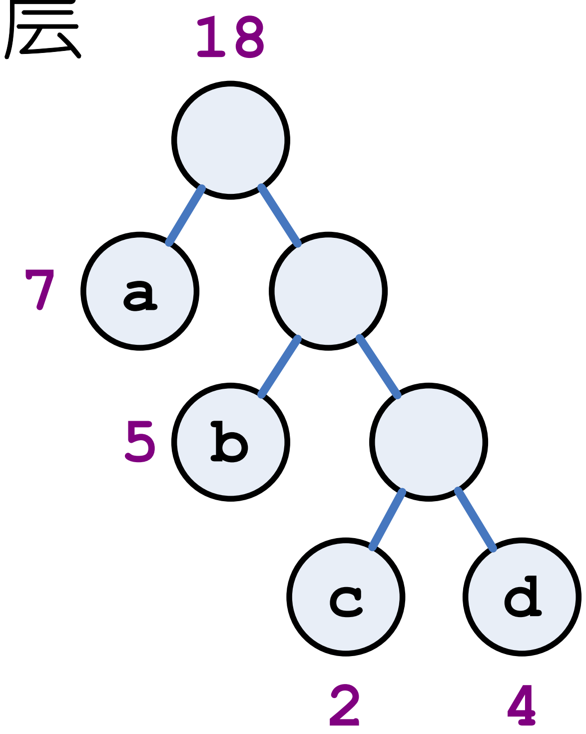


(3) F = {  **}**

(4) F = {  **}**

赫夫曼树及其应用

- 理解
  - 为什么赫夫曼树的带权路径长度最小？
  - 它把权值小的结点放在底层
  - 权值大的结点放在上层



赫夫曼树的应用——赫夫曼编码

- 例
  - 某系统在通信联络中只可能出现八种字符**(A,B,C,D,E,F,G,H)**，其使用概率分别为**0.05**、**0.29**、**0.07**、**0.08**、**0.14**、**0.23**、**0.03**、**0.11**，如何设计这些字符的二进制编码，以使通信中总码长尽可能短？
- 方案一：固定长度编码
  - **8**个字符，只需要**3**位二进制数就能表示
  - 比如**000**代表**A**，**001**代表**B**，**...111**代表**H**
  - 这样平均每个字符用**3**位二进制数表示

赫夫曼树的应用——赫夫曼编码

- 方案二：赫夫曼编码

A	B	C	D	E	F	G	H
0.05	0.29	0.07	0.08	0.14	0.23	0.03	0.11
0110	10	1110	1111	110	00	0111	010

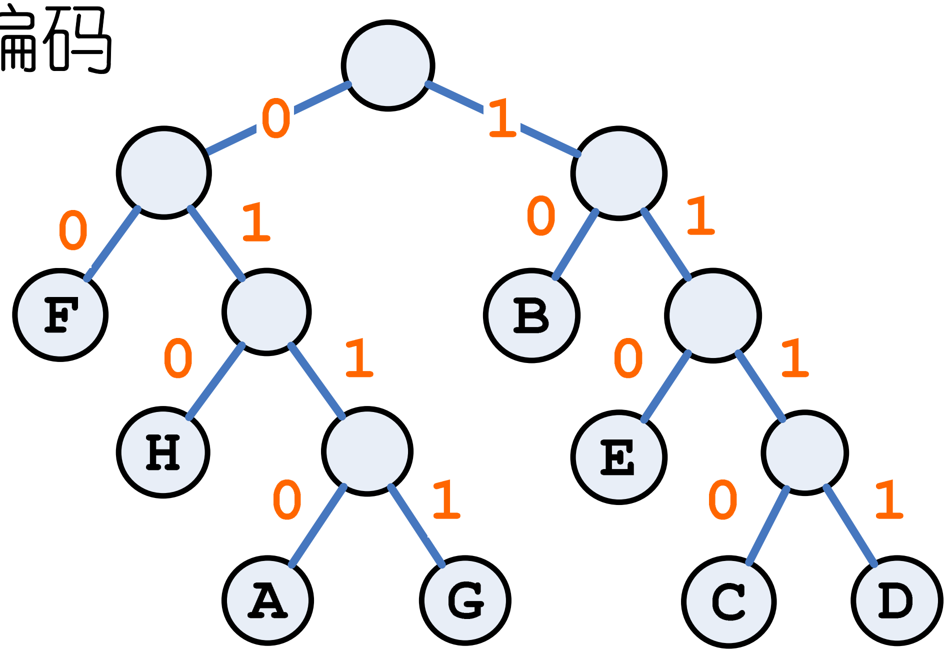
– 平均每个字符的编码长度

$$= 4*0.05 + 2*0.29 + 4*0.07 + 4*0.08 + 3*0.14 + 2*0.23 + 4*0.03 + 3*0.11$$

$$= 2.71$$

赫夫曼树的应用——赫夫曼编码

- 赫夫曼编码的方法
  - 以字符出现频率为权值，构造赫夫曼树
  - 左分支表示**0**，右分支表示**1**，把从根到叶子的路径上所有分支构成的**0,1**作为叶子的二进制编码，即为赫夫曼编码
  - 比如
    - **A**：**0110**
    - **B**：**10**
    - **C**：**1110**
    - **...**



赫夫曼树的应用——赫夫曼编码

【数据结构】

```
typedef struct{
  unsigned weight;
  unsigned p;
  unsigned lc;
  unsigned rc;
}HTNode,*HuffmanTree;

typedef char * *HuffmanCode;
```

赫夫曼树的应用——赫夫曼编码

HT

weight	p	lc	rc	
				0
5	0	0	0	1
29	0	0	0	2
7	0	0	0	3
8	0	0	0	4
14	0	0	0	5
23	0	0	0	6
3	0	0	0	7
11	0	0	0	8
				9
				10
				11
				12
				13
				14
				15

(1)为赫夫曼树的存储开辟空间，含 n 个叶子结点的赫夫曼树共2n-1 个结点；

(2)生成 n 个叶子结点，使它们的权值为给定的 n 个权值，双亲和左右孩子均为0 ；

赫夫曼树的应用——赫夫曼编码

HT

	weight	p	lc	rc
0				
1	5	9	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	9	0	0
8	11	0	0	0
9	8	0	1	7
10				
11				
12				
13				
14				
15				

(3)在双亲为 **0** 的结点中选两个权值最小的，生成以这两个结点为左、右孩子的分支结点。

(4)重复(3)，直至生成 n−1 个分支结点。

赫夫曼树的应用——赫夫曼编码

HT

weight	p	lc	rc	
				0
5	9	0	0	1
29	0	0	0	2
7	10	0	0	3
8	10	0	0	4
14	0	0	0	5
23	0	0	0	6
3	9	0	0	7
11	0	0	0	8
8	0	1	7	9
15	0	3	4	10
				11
				12
				13
				14
				15

(3)在双亲为 **0** 的结点中选两个权值最小的，生成以这两个结点为左、右孩子的分支结点。

(4)重复(3)，直至生成 n−1 个分支结点。

赫夫曼树的应用——赫夫曼编码

- 求出编码

对赫夫曼树中的每个叶子结点，求从根到它的路径：

(1) 开辟 n 个存储空间，用以保存分别指向 n 个编码串的指针变量；

(2) 开辟空间，用以暂存当前所求编码串

(3) 对每个叶子结点，从该结点开始向根逆向求编码。

HT

weight	p	lc	rc	
				0
5	9	0	0	1
29	14	0	0	2
7	10	0	0	3
8	10	0	0	4
14	12	0	0	5
23	13	0	0	6
3	9	0	0	7
11	11	0	0	8
8	11	7	1	9
15	12	3	4	10
19	13	9	8	11
29	14	5	10	12
42	15	11	6	13
58	15	2	12	14
100	0	13	14	15

cd

start 3

			0	0	0	1	\0

HC

0						
1		0	0	0	1	\0
2						
3						
4						
5						
6						
7						

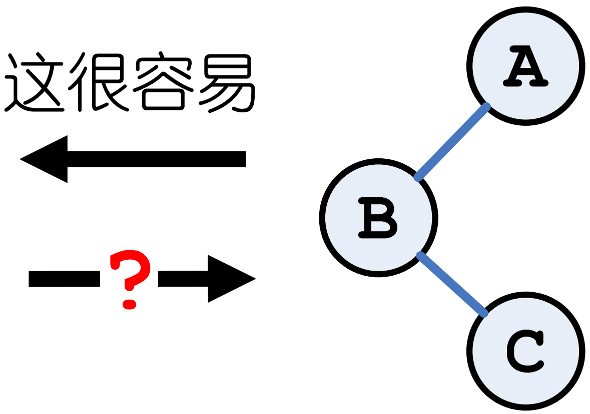
赫夫曼树及其应用

- 本节小结
  - 树的带权路径长度：掌握计算方法
  - 最优二叉树（赫夫曼树）：概念
  - 赫夫曼树的构造方法：手工
  - 赫夫曼编码：手工

树的计数

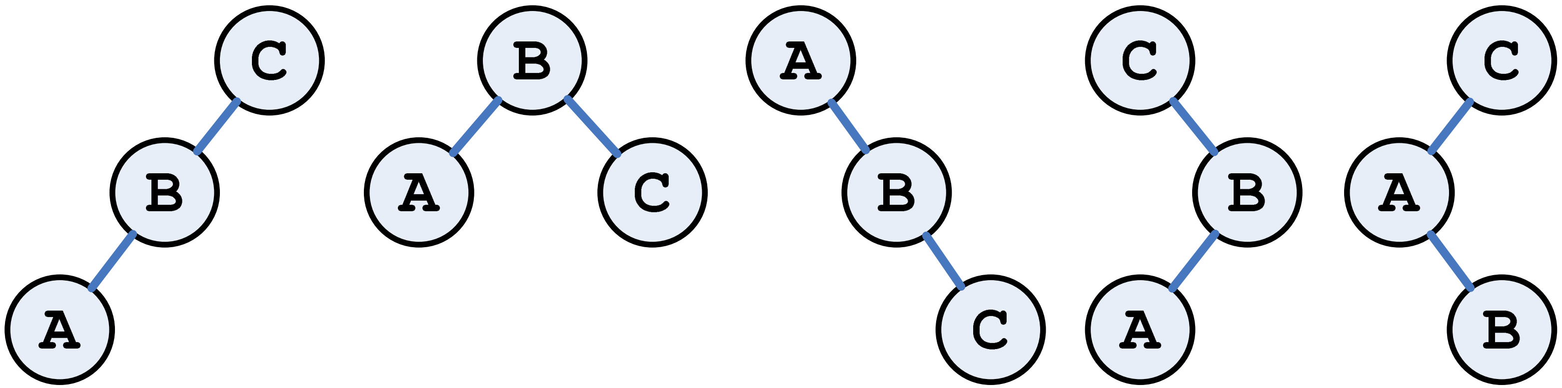
- 以下命题显然成立：
  - 给定一棵二叉树，其先序、中序、后序和层序遍历的结果是唯一的
- 反过来呢？
  - 给定一棵二叉树先序、中序、后序或层序遍历的结果，你能倒推回那棵二叉树么？

先序遍历：	**A**	**B**	**C**
中序遍历：	**B**	**C**	**A**
后序遍历：	**C**	**B**	**A**
层序遍历：	**A**	**B**	**C**



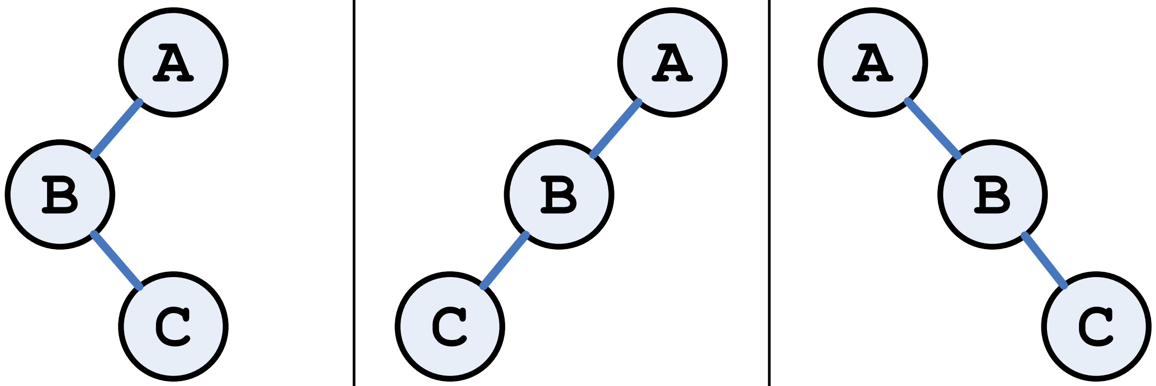
树的计数

- 首先：只有一个遍历的结果是不够的
- 比如：
  - 中序遍历结果是：**A B C**
  - 可能的二叉树有：



  - 那给定两个遍历结果呢？

树的计数



. . .

先序	A B C	A B C	A B C
后序	C B A	C B A	C B A
层序	A B C	A B C	A B C
中序	B C A	C B A	A B C

相同

树的计数

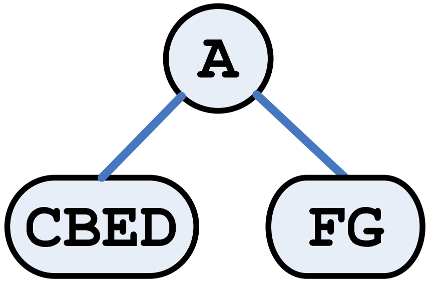
- 上面的例子告诉我们
  - 要唯一确定该二叉树，需要中序遍历的结果 + 任意一种其它遍历的结果
  - 比如中序+前序、中序+后序、中序+层序

树的计数

- 例题
  - 已知一棵二叉树
    - 先序遍历的结果为**ABCDEFG**
    - 中序遍历的结果为**CBEDAFG**
  - 请画出这个二叉树
- 思路
  - 先序序列中的第一个一定是树根
  - 中序序列...**x**...中，如果**x**是树根，则
    - **x**左边的一定是左子树的结点
    - **x**右边的一定是右子树的结点

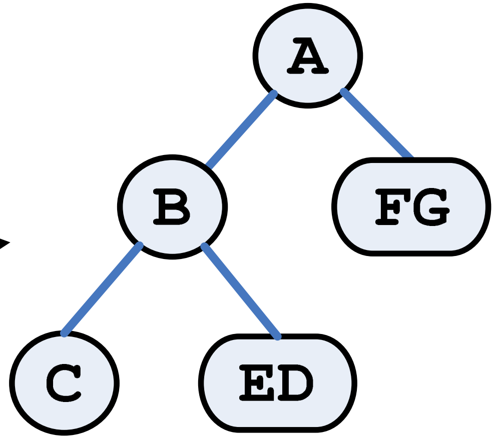
树的计数

- 解答



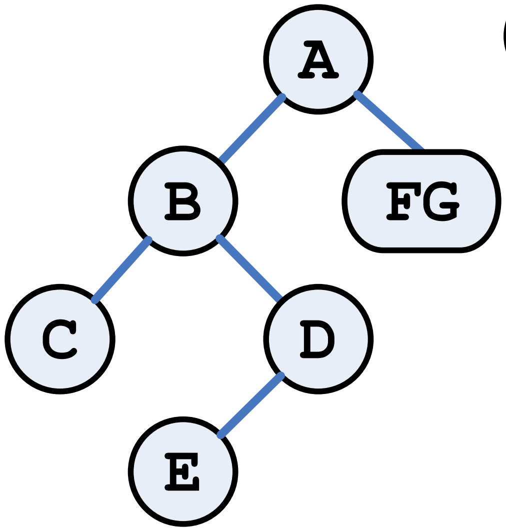
先序： A B C D E F G

中序： C B E D A F G



先序： B C D E

中序： C B E D



先序： D E

中序： E D

树的计数

- 作业3
  - 习题集6.6
- 思考
  - 证明：由一棵二叉树的先序序列和中序序列可以唯一的确定这棵二叉树
    - 提示：数学归纳法
  - 另外，你能写出算法么？

本章总结

- 一、树的类型定义
  - 树是递归定义的
    - 子树也是树
  - 相关术语可以类比家族树来记忆
- 二、二叉树的类型定义
  - 相关术语要注意跟树做对比
    - 孩子最多两个
    - 而且要分左右
  - **5**个性质要牢记并且会推导

本章总结

- 三、二叉树的存储结构
  - 顺序存放
    - 简单，但是空间浪费大
  - 二叉链表
    - 常用
  - 三叉链表
    - 找父亲很快

本章总结

- 四、二叉树的操作
  - 遍历操作
    - 遍历的定义是递归的
    - 算法当然也可以递归
    - 非递归的话要借助堆栈
    - 层序遍历要借助队列
  - 其它操作
    - 基本方法无外乎是深度、广度和递归

本章总结

- 五、线索二叉树
  - 叶子的指针的闲置
    - 只有一个孩子的结点也有一个指针闲置
  - 因此利用它来帮助遍历
  - 遍历的算法是要会写
  - 一些不足也要知道
    - 前序/后序线索二叉树难以找后继/前驱结点
    - 因为二叉链表找父亲不方便

本章总结

- 六、树和森林
  - 树的表示方法多样
    - 各种方法的各有优劣
    - 要求手工能画出来就可以了
  - 树、森林←→二叉树相互转化
  - 树和森林的遍历要注意
    - 方法和二叉树有点儿不同

本章总结

- 七、赫夫曼树
  - 赫夫曼树的相关概念
  - 手工构造要掌握
  - 应用：赫夫曼编码
- 八、树的计数
  - 想要确定一棵二叉树
    - 中序序列是必须的
    - 中序序列+其它一种序列就唯一确定该树

练习