# 第六章 树和二叉树

**树的概念和基本术语**

**二叉树**

**二叉树遍历**

**二叉树的计数**

**树与森林**

**霍夫曼树**

# 树的概念和基本术语

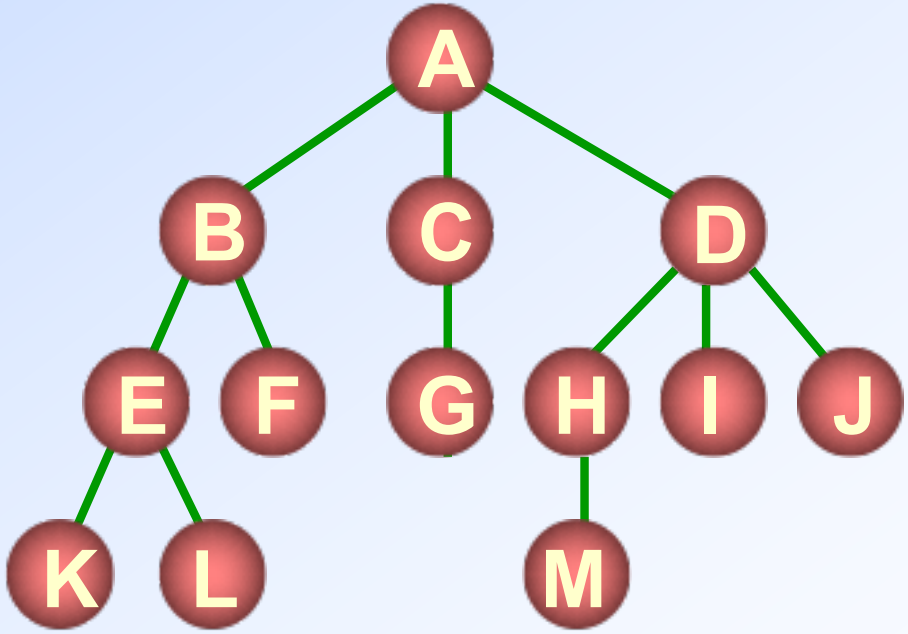
## 树的定义

树是由 $n\ (n \geq 0)$ 个结点的有限集合。如果 $n = 0$，称为空树；如果 $n > 0$，则

- 有且仅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；
- 当$n > 1$，除根以外的其它结点划分为 $m\ (m > 0)$ 个互不相交的有限集 $T_1, T_2, \ldots, T_m$，其中每个集合本身又是一棵树，并且称为根的子树(SubTree)。

# 例如


只有根结点的树



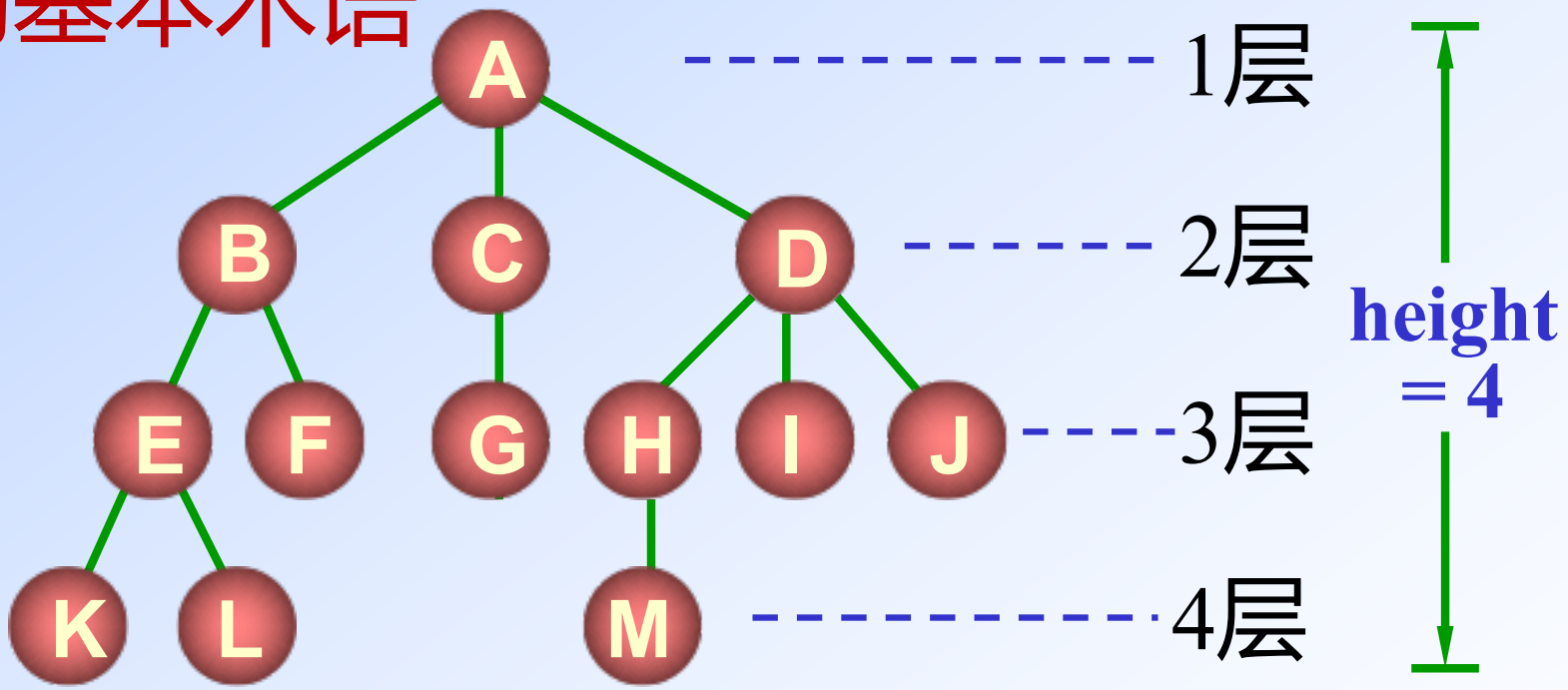

有13个结点的树

其中：A是根；其余结点分成三个互不相交的子集，

T1={B,E,F,K,L}； T2={C,G}； T3={D,H,I,J,M},

T1,T2,T3都是根A的子树，且本身也是一棵树

# 树的基本术语




结点
结点的度
叶结点
分支结点

子女
双亲
兄弟

祖先
子孙
结点层次

树的深度
树的度
有序树
无序树
森林

结点：一个数据元素及指向其子树的分支。
结点的度：结点拥有的子树个数。
叶结点：度为零的结点。
子女：结点子树的根。
兄弟：同一结点子女。
祖先：根到该结点路径上的所有结点。
子孙：某结点为根的子树上的任意结点。
结点层次：从根开始，根为第一层，根的子女为第二层，以此类推。
树的深度（高度）：树中结点的最大层次数。
有序树：树中结点的子树由左向右有序。
森林：$m(m \geq 0)$棵互不相交的树。

# 二叉树 (Binary Tree)

**定义**　一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。

**特点**　每个结点至多只有两棵子树（二叉树中不存在度大于2的结点）

**五种形态**

# 性质

**性质1** 在二叉树的第 $i$ 层上至多有 $2^{i-1}$ 个结点。($i \geq 1$) [证明用归纳法]

证明：当i=1时，只有根结点，$2^{i-1}=2^0=1$。

假设对所有j，i>j≥1，命题成立，即第j层上至多有$2^{j-1}$ 个结点。

由归纳假设第i-1 层上至多有 $2^{i-2}$个结点。

由于二叉树的每个结点的度至多为2，故在第i层上的最大结点数为第i-1层上的最大结点数的2倍，即$2* 2^{i-2}= 2^{i-1}$。

**性质2**　　深度为 k 的二叉树至多有 $2^k-1$ 个结点($k \geq 1$)。

**证明：由性质1可见，深度为k的二叉树的最大结点数为**

$$\sum_{i=1}^{k}(\text{第}i\text{层上的最大结点数})$$

$$= \sum_{i=1}^{k} 2^{i-1} = 2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$$

**性质3**　　对任何一棵二叉树T, 如果其叶结点数为 $n_0$, 度为2的结点数为 $n_2$,则$n_0 = n_2 + 1$.

证明：若度为1的结点有 $n_1$ 个，总结点个数为 n，总边数为 e，则根据二叉树的定义，

$$n = n_0 + n_1 + n_2 \qquad e = 2n_2 + n_1 = n - 1$$

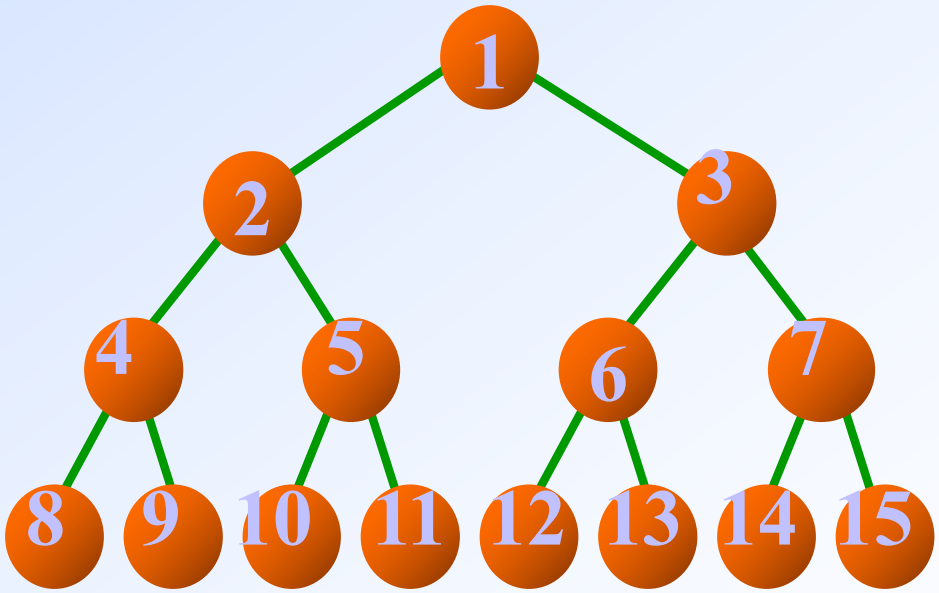
因此，有 $2n_2 + n_1 = n_0 + n_1 + n_2 - 1$

$$n_2 = n_0 - 1 \Rightarrow n_0 = n_2 + 1$$

# 两种特殊形态的二叉树

**定义1 满二叉树 (Full Binary Tree)**

一棵深度为k且有$2^k-1$个结点的二叉树称为满二叉树。

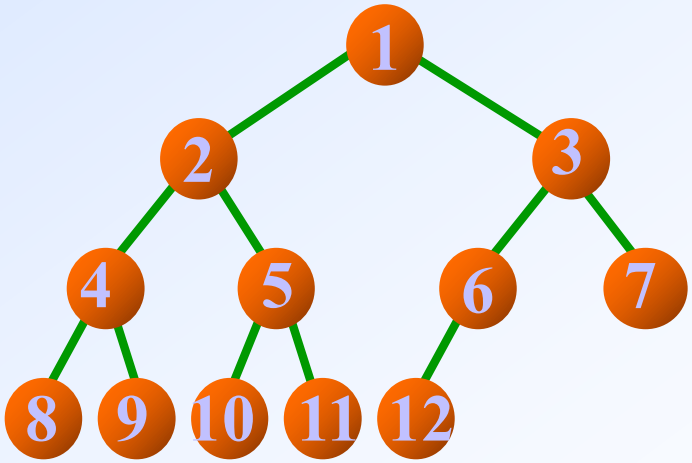


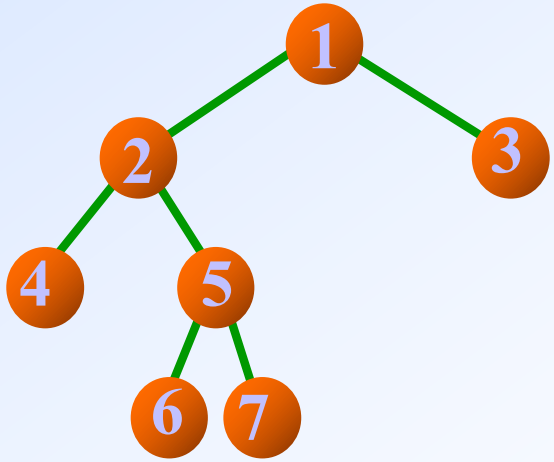

满二叉树

## 定义2 完全二叉树 (Complete Binary Tree)

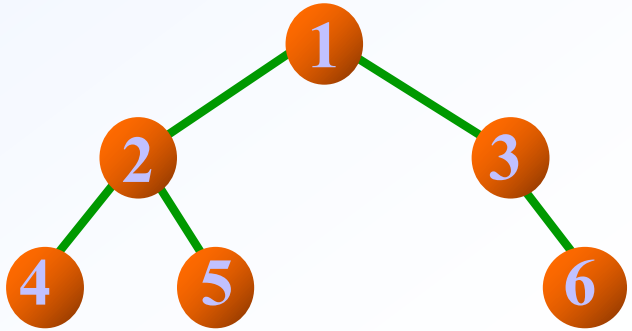
若设二叉树的高度为h，则共有h层。除第 h 层外，其它各层 (0 ~ h−1) 的结点数都达到最大个数，第 h 层从右向左连续缺若干结点，这就是完全二叉树。

完全二叉树 →




非完全二叉树



**性质4** 具有 $n\ (n \geq 0)$ 个结点的完全二叉树的深度为 $\lfloor \log_2(n) \rfloor + 1$

证明：

设完全二叉树的深度为 $h$，则根据性质2和完全二叉树的定义有

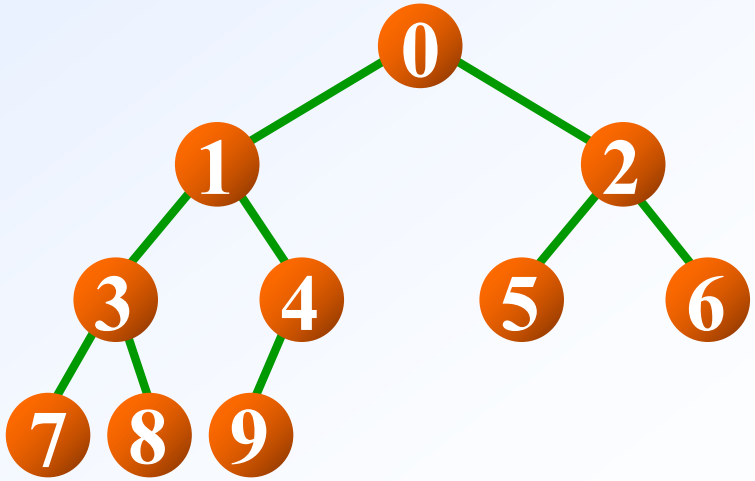
$$2^{h-1} - 1 < n \leq 2^h - 1 \text{ 或 } 2^{h-1} \leq n < 2^h$$

取对数 $h - 1 < \log_2 n \leq h$，又h是整数，

因此有 $h = \lfloor \log_2(n) \rfloor + 1$

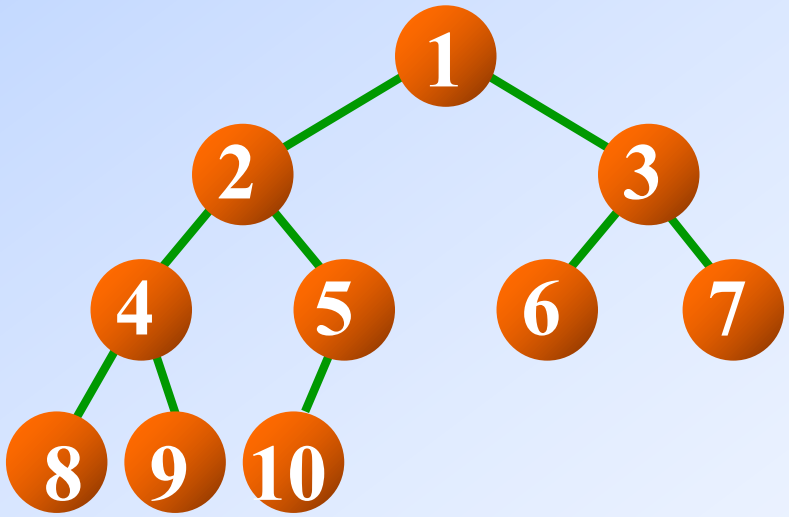
**性质5** 如将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号0, 1, 2, …, n-1，则有以下关系：

- 若i = 0, 则 i 无双亲
  若i > 0, 则 i 的双亲为⌊(i -1)/2⌋
- 若2*i+1 < n, 则 i 的左子女为 2*i+1，若2*i+2 < n, 则 i 的右子女为2*i+2
- 若结点编号i为偶数，且i!=0,则左兄弟结点i-1.
- 若结点编号i为奇数，且i!=n-1,则右兄弟结点为i+1.
- 结点i 所在层次为⌊log2(i+1) ⌋



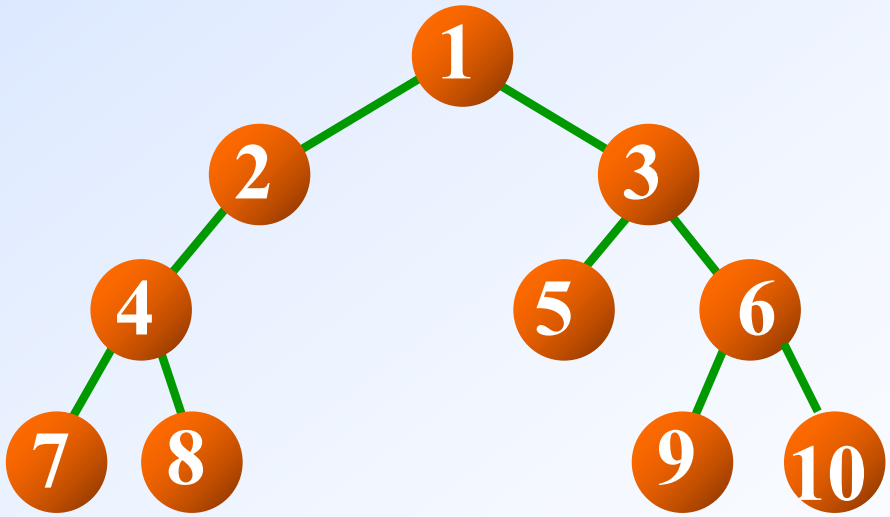
# 二叉树的存储结构

- **顺序表示**





**完全二叉树的顺序表示**

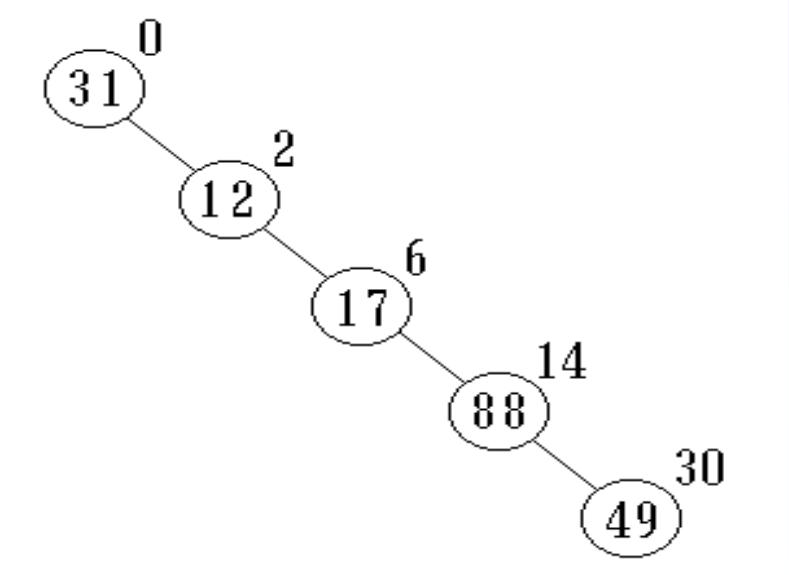




**一般二叉树的顺序表示**

由于一般二叉树必须仿照完全二叉树那样存储，可能会浪费很多存储空间，单支树就是一个极端情况。

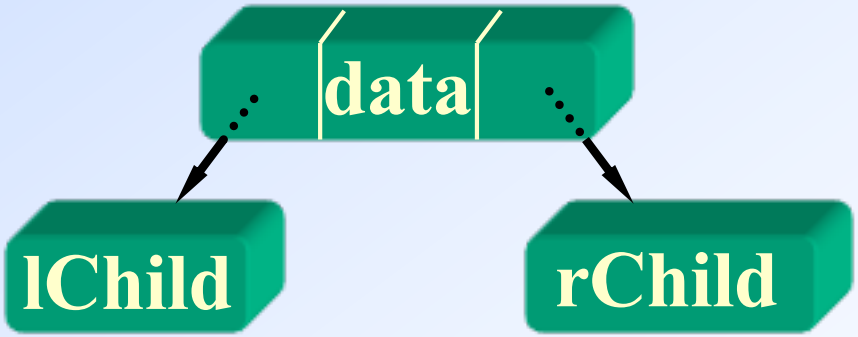
单支树



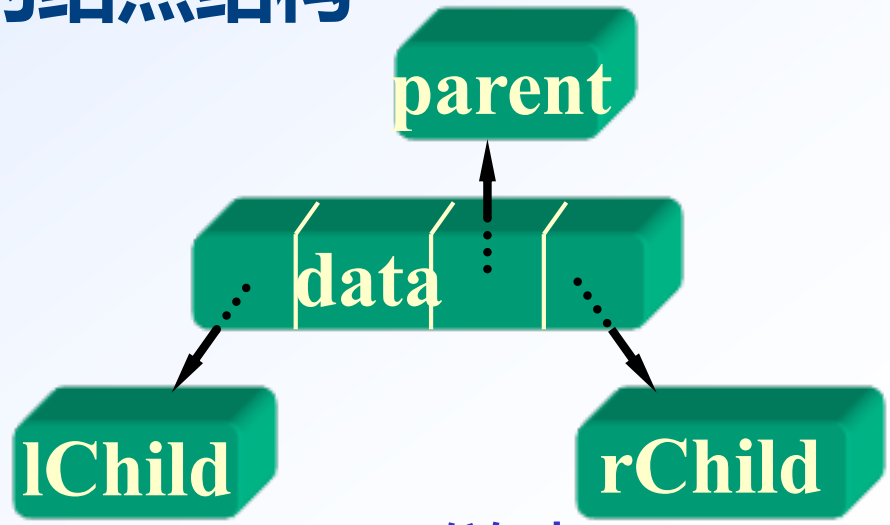
## ▪链表表示


含两个指针域的结点结构


含三个指针域的结点结构




二叉链表




三叉链表

性质：

含有n个结点的二叉链表中有n+1个空链域。

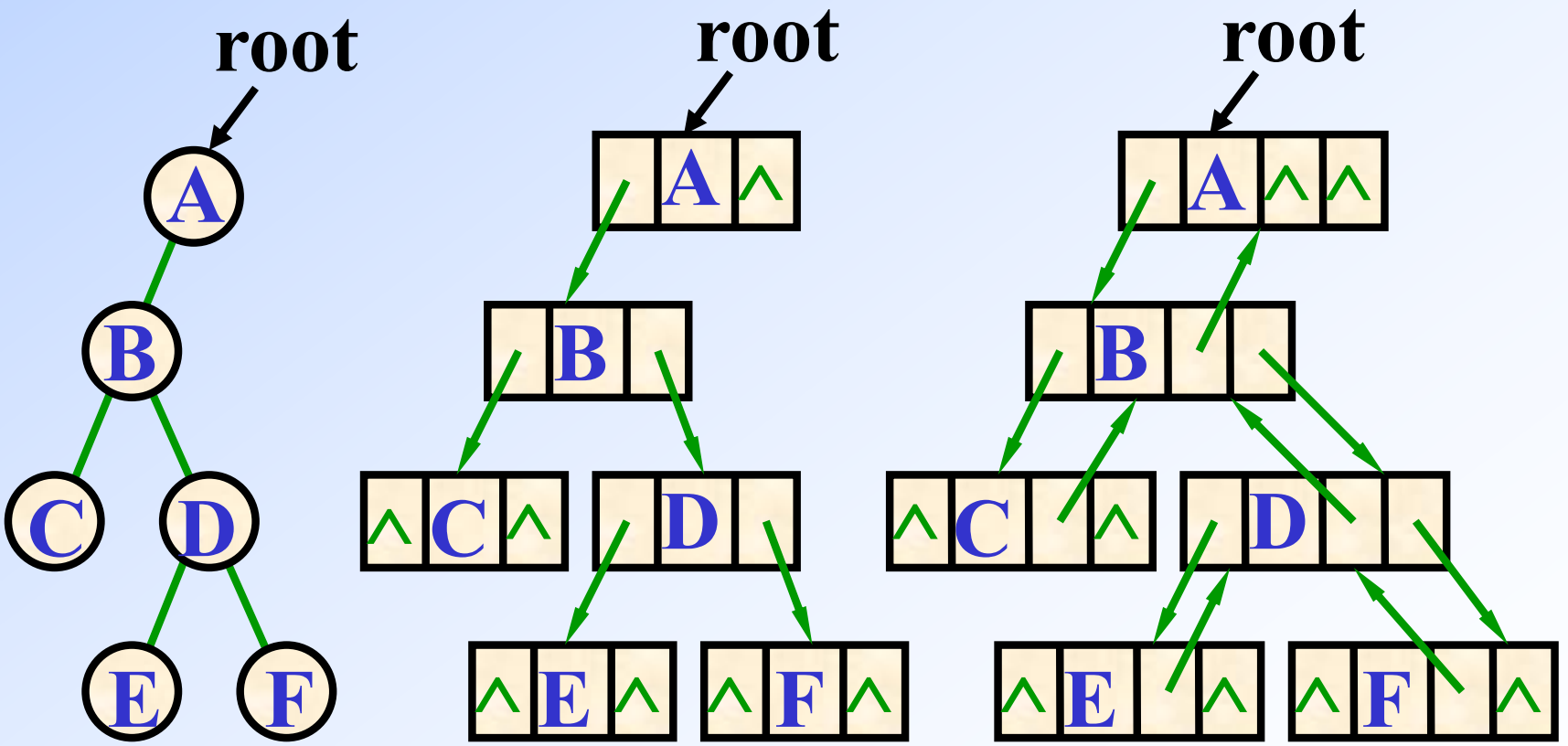
证明：对于叶结点$n_0$有两个空链域，$n_1$有1个空链域，$n_2$没有空链域。所以

空链域数 = $2n_0+n_1$

将$n_0=n_2+1$代入得(根据性质3)

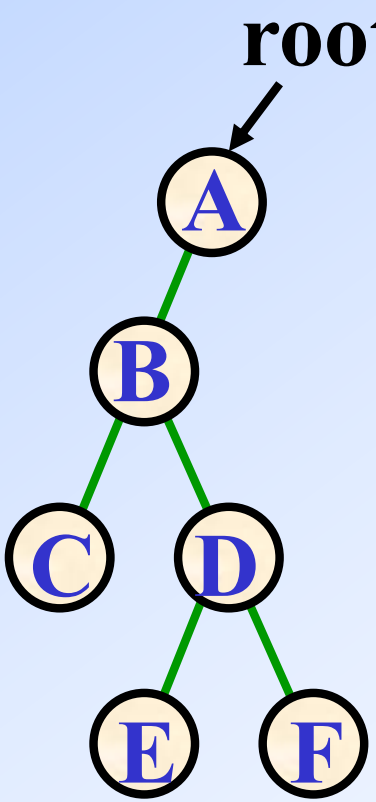
$2n_0+n_1=n_0+n_1+n_2+1=n+1$

# 二叉树链表表示的示例




二叉树　　二叉链表　　三叉链表

# 三叉链表的静态结构




| | data | parent | leftChild | rightChild |
|---|---|---|---|---|
| 0 | A | -1 | 1 | -1 |
| 1 | B | 0 | 2 | 3 |
| 2 | C | 1 | -1 | -1 |
| 3 | D | 1 | 4 | 5 |
| 4 | E | 3 | -1 | -1 |
| 5 | F | 3 | -1 | -1 |

# 二叉链表的定义

```
typedef char TreeData;  //结点数据类型

typedef struct node {      //结点定义
    TreeData data;
    struct node * leftChild, * rightchild;
} BinTreeNode;

typedef BinTreeNode * BinTree;
                           //根指针
```

# 二叉树遍历

树的遍历就是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。

设访问根结点记作 V

遍历根的左子树记作 L

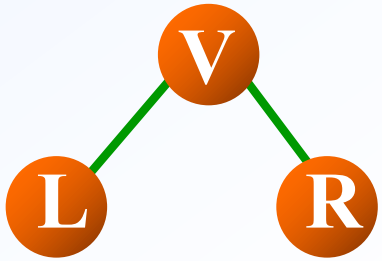
遍历根的右子树记作 R

则可能的遍历次序有

前序　VLR

中序　LVR

后序　LRV



# 中序遍历 (Inorder Traversal)

## 中序遍历二叉树算法的定义：

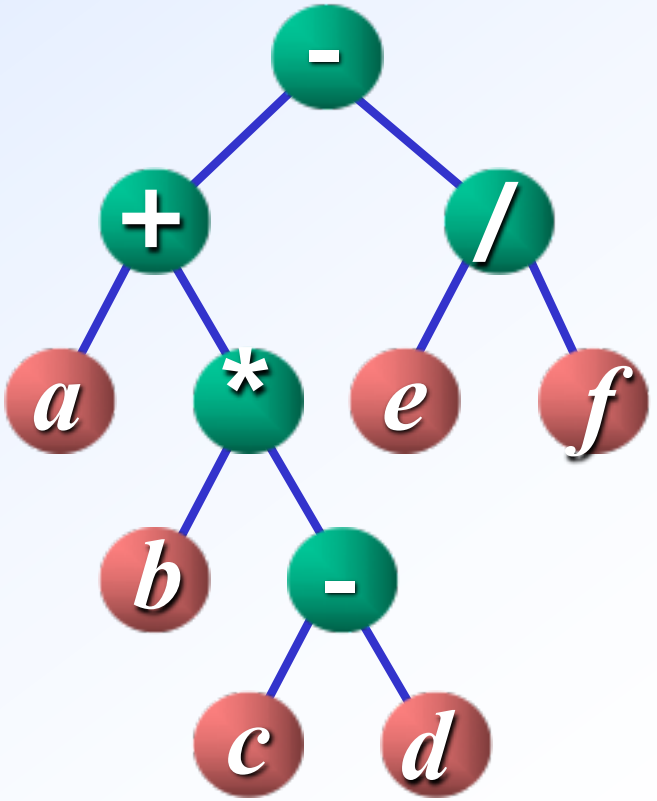
若二叉树为空，则空操作；

否则

中序遍历左子树 (L)；

访问根结点 (V)；

中序遍历右子树 (R)。

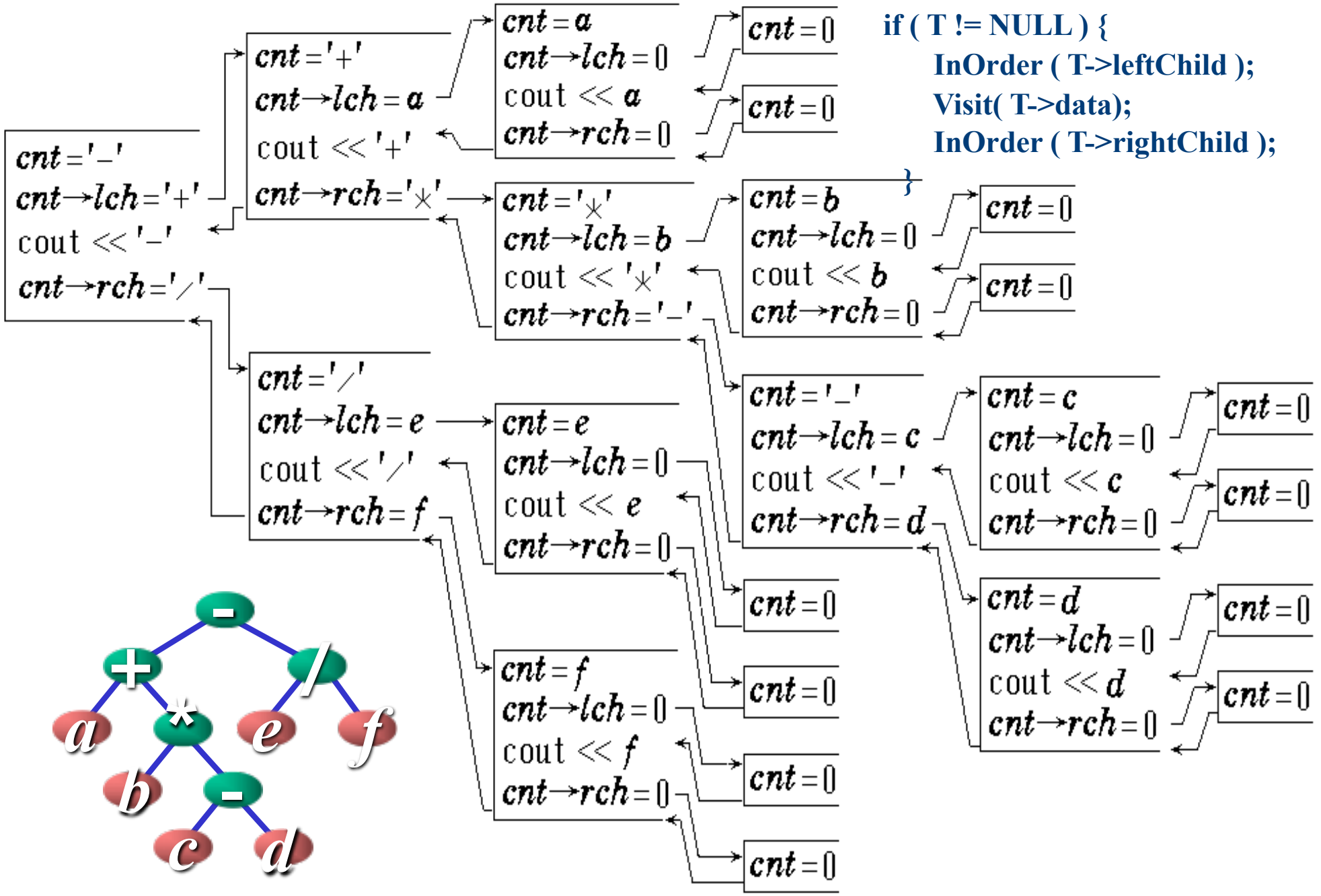



遍历结果

a + b * c - d - e / f

- **中序遍历二叉树的递归算法**

```
void InOrder ( BinTreeNode *T )
{
    if ( T != NULL ) {
        InOrder ( T->leftChild );
        Visit( T->data);
        InOrder ( T->rightChild );
    }
}
```




# 中序遍历二叉树的递归过程图解

# 前序遍历 (Preorder Traversal)

## 前序遍历二叉树算法的定义：

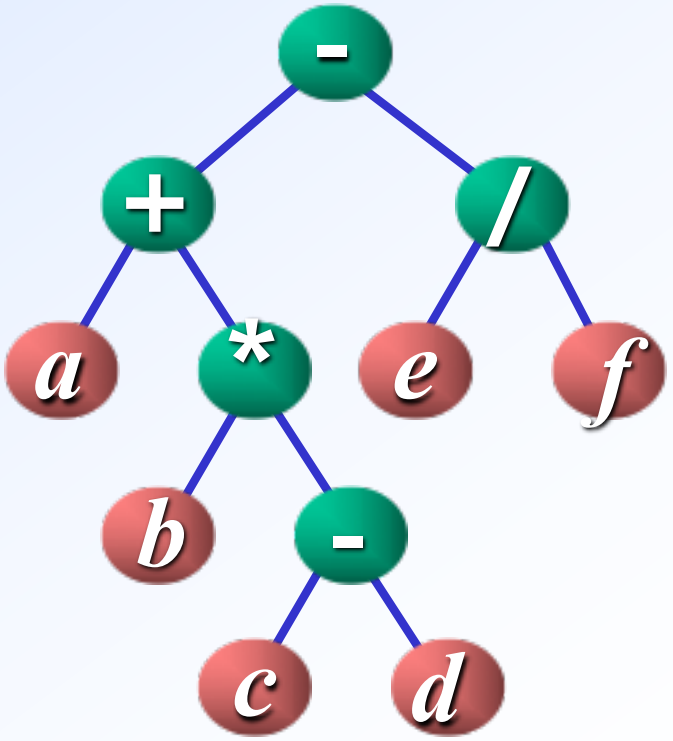
若二叉树为空，则空操作；

否则

访问根结点 (V)；

前序遍历左子树 (L)；

前序遍历右子树 (R)。




遍历结果

- + a * b - c d / e f

## ▪前序遍历二叉树的递归算法

```
void PreOrder ( BinTreeNode *T )
{
    if ( T != NULL ) {
        Visit( T->data);
        PreOrder ( T->leftChild );
        PreOrder ( T->rightChild );
    }
}
```

# 后序遍历 (Postorder Traversal)

## 后序遍历二叉树算法的定义：

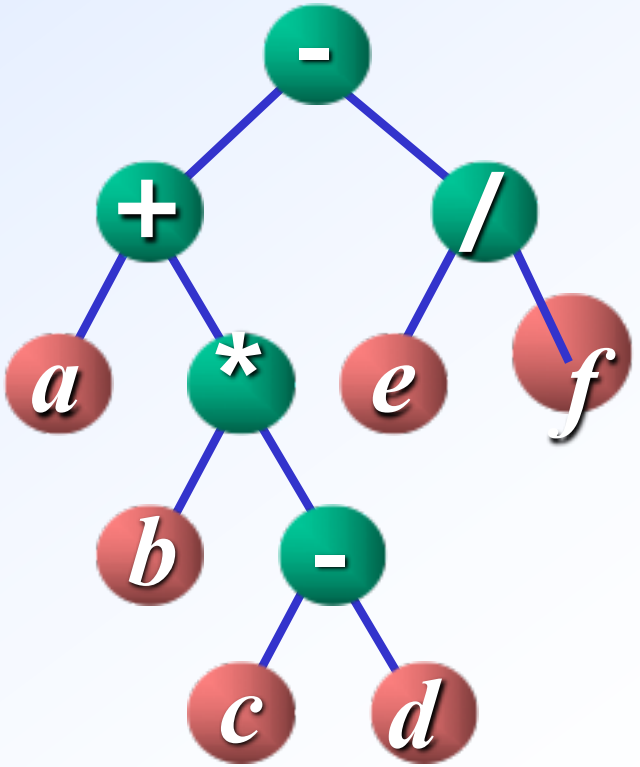
若二叉树为空，则空操作；

否则

后序遍历左子树 (L)；

后序遍历右子树 (R)；

访问根结点 (V)。




遍历结果

a b c d - * + e f / -

- **后序遍历二叉树的递归算法：**

```
void PostOrder ( BinTreeNode * T ) {
    if ( T != NULL ) {
        PostOrder ( T->leftChild );
        PostOrder ( T->rightChild );
        Visit( T->data);
    }
}
```

# 二叉树遍历应用

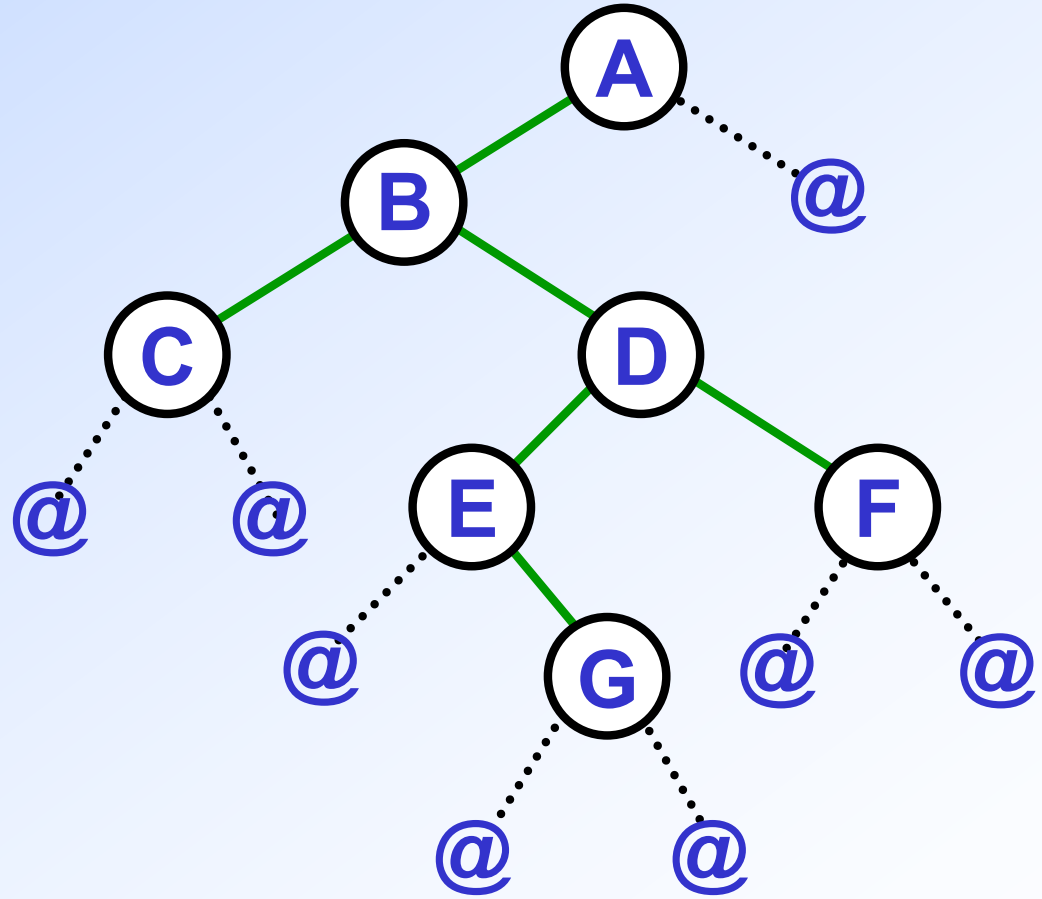
## 1. 按前序建立二叉树(递归算法)

以递归方式建立二叉树。

输入结点值的顺序必须对应二叉树结点前序遍历的顺序。并约定以输入序列中不可能出现的值作为空结点的值以结束递归, 此值在RefValue中。例如用“@”或用“-1”表示字符序列或正整数序列空结点。

# 如图所示的二叉树的前序遍历顺序为

A B C @ @ D E @ G @ @ F @ @ @



```
Status CreateBiTree ( BiTree& T ) {
    scanf(&ch);
    if ( ch==‘’ ) T=NULL; //读入根结点的值
    else{
        if ( !(T=(BiTNode *)malloc(sizeof(BiTNode))))
exit(OVERFLOW); //建立根结点
        T->data = ch;
        CreateBiTree ( T->leftChild );
        CreateBiTree ( T->rightChild );
        }
   return OK;
}//CreateBiTree
```

## 2. 计算二叉树结点个数(递归算法)

```
int Count ( BinTreeNode *T ) {
    if ( T == NULL ) return 0;
    else return 1 + Count ( T->leftChild )
                  + Count ( T->rightChild );
}
```

# 3. 求二叉树中叶子结点的个数

```
int Leaf_Count(Bitree T)
 {//求二叉树中叶子结点的数目
        if(!T) return 0; //空树没有叶子
        elseif(!T->lchild&&!T->rchild) return 1;
        //叶子结点
         else return Leaf_Count(T-lchild)+Leaf_Count(T-
rchild); //左子树的叶子数加上右子树的叶子数
}
```

# 4. 求二叉树高度(递归算法)

```
int Height ( BinTreeNode * T )
{
    if ( T == NULL ) return 0;
    else{
        int m = Height ( T->leftChild );
        int n = Height ( T->rightChild ) );
        return (m > n) ? m+1 : n+1;
        }
}
```

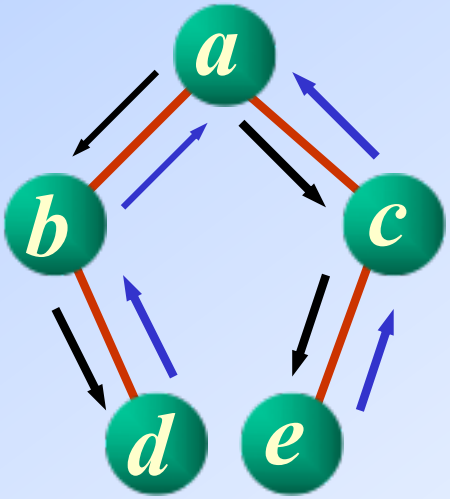
## 5. 复制二叉树(递归算法)

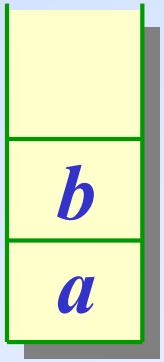
```
BiTNode* Copy( BinTreeNode * T )
{
        if ( T == NULL ) return NULL;
        if(!(Temp=(BiTNode
*)malloc(sizeof(BiTNode)))) exit(OVERFLOW);
        Temp->data=T->data;
        Temp-> leftChild = Copy( T->leftChild );
        Temp-> rightChild = Copy(T->rightChild );
        return Temp;
}
```

# 6. 判断二叉树等价(递归算法)

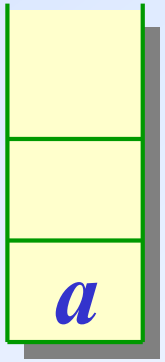
```
int Equal( BinTreeNode *a, BinTreeNode *b) {
    if ( a == NULL && b == NULL ) return 1;
    if ( a !== NULL && b !== NULL
        && a->data==b->data
        && equal( a->leftChild, b->leftChild)
        && equal( a->rightChild, b->rightChild) )
        return 1;
    return 0;//如果a和b的子树不等同，则函数返回0
}
```

# 中序遍历二叉树(非递归算法)用栈实现

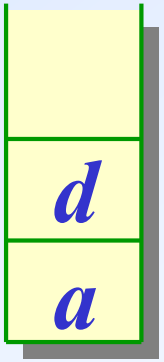


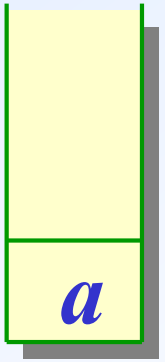

a b入栈



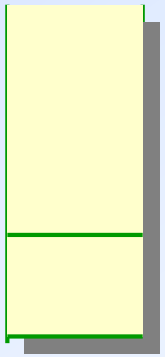

b退栈
访问



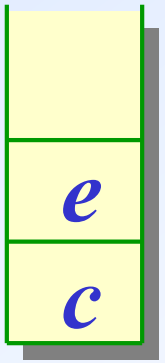

*d*入栈



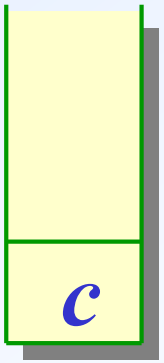

*d*退栈
访问



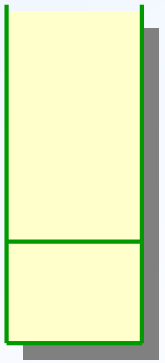
*a*退栈
访问



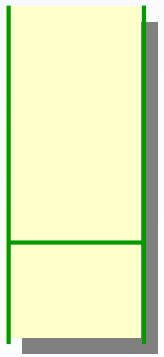

*c e* 入栈




*e* 退栈
访问

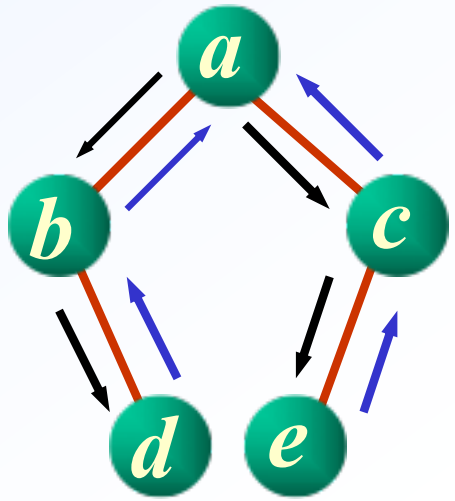


*c* 退栈
访问

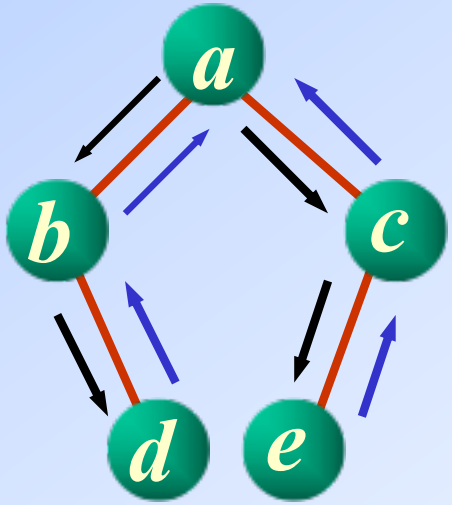


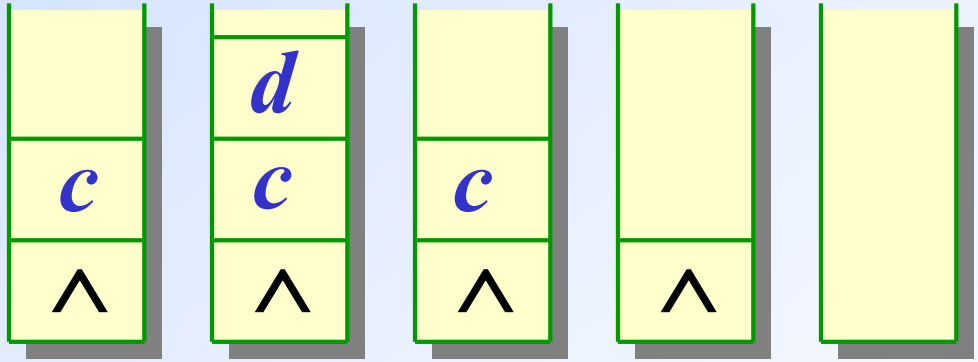
栈空

```
void InOrder ( BinTree T ) {
    stack S;   InitStack( &S );    //递归工作栈
    Push(S,T);          //初始化
    while ( !StackEmpty(S) {
        while(GetTop(S,p)&&p) Push(S, p->lchild);
        Pop(S,p);
        if ( !StackEmpty(S) ) {          //栈非空
            Pop(S, p);                   //退栈
            Visit(p->data);   //访问根
            Push(S, p->rchild);
        }//if
    }//while
 return ok;
}
```



# 前序遍历二叉树(非递归算法)用栈实现

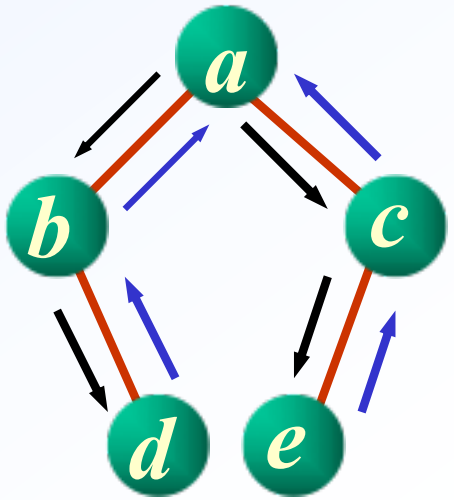



| 访问 | 访问 | 退栈 | 退栈 | 访问 |
|---|---|---|---|---|
| *a* | *b* | *d* | *c* | *e* |
| 进栈 | 进栈 | 访问 | 访问 | 左进 |
| *c* | *d* | *d* | *c* | *空* |
| 左进 | 左进 | 左进 | 左进 | 退栈 |
| *b* | *空* | *空* | *e* | ^ |

结束

```
void PreOrder( BinTree T ) {
    stack S;   InitStack(S); //递归工作栈
    BinTreeNode * p = T;  Push (S, NULL);
    while ( p != NULL ) {
        Visit(p->data);
        if ( p->rightChild != NULL )
            Push ( S, p->rightChild );
        if  ( p->leftChild != NULL )
            p = p->leftChild;//进左子树
        else Pop( S, p );
    }
}
```



# ❖后序遍历二叉树(非递归算法)用栈实现

## 后序遍历时使用的栈的结点定义

```
typedef  struct {
    BinTreeNode *ptr;      //结点指针
    enum tag{ L, R };      //该结点退栈标记
} StackNode;
```


根结点的

tag = L，表示从左子树退出, 访问右子树。

tag = R，表示从右子树退出, 访问根。

```
void PostOrder ( BinTree T ) {
  stack S;   InitStack(S);  StackNode w;
  BinTreeNode * p = T;
  do {
     while ( p != NULL ) {   //向左子树走
       w.ptr = p;   w.tag = L;   Push (S, w);
       p = p->leftChild;
                    }
     int continue = 1;          //继续循环标记
```

```
    while ( continue && !StackEmpty(S) ) {
        Pop (S, w);  p = w.ptr;
        switch ( w.tag ) {          //判断栈顶tag标记
          case L :  w.tag = R;   Push (S, w);
                    continue = 0;
                    p = p->rightChild;   break;
          case R :  Visit( p->data);
        }
      }
   } while ( p != NULL ||  !StackEmpty(S) );
}
```

# 练习:交换二叉树各结点的左、右子树(递归算法)

# 练习:交换二叉树各结点的左、右子树(递归算法)

```
void  unknown ( BinTreeNode * T ) {
   BinTreeNode *p = T, *temp;
   if ( p != NULL ) {
      temp = p->leftChild;
      p->leftChild = p->rightChild;
      p->rightChild = temp;
      unknown ( p->leftChild );
      unknown ( p->rightChild );
   }
}
```

# 不用栈消去递归算法中的第二个递归语句

```
void unknown ( BinTreeNode * T ) {
    BinTreeNode *p = T, *temp;
    while ( p != NULL ) {
        temp = p->leftChild;
        p->leftChild = p->rightChild;
        p->rightChild = temp;
        unknown ( p->leftChild );
        p = p->rightChild;
    }
}
```

# 使用栈消去递归算法中的两个递归语句

```
void unknown ( BinTreeNode * T ) {
    BinTreeNode *p, *temp;
    stack S;   InitEmpty (&S);
    if ( T != NULL ) {
        push(&S, T);
        while ( ! StackEmpty(&S) ) {
            Pop(&S, p);   //栈中退出一个结点
            temp = p->leftChild;     //交换子女
            p->leftChild = p->rightChild;
            p->rightChild = temp;
```

```
            if ( p->rightChild != NULL )
                push (&S, p->rightChild );
            if ( p->leftChild != NULL )
                push (&S, p->leftChild );
        }
    }
}
```

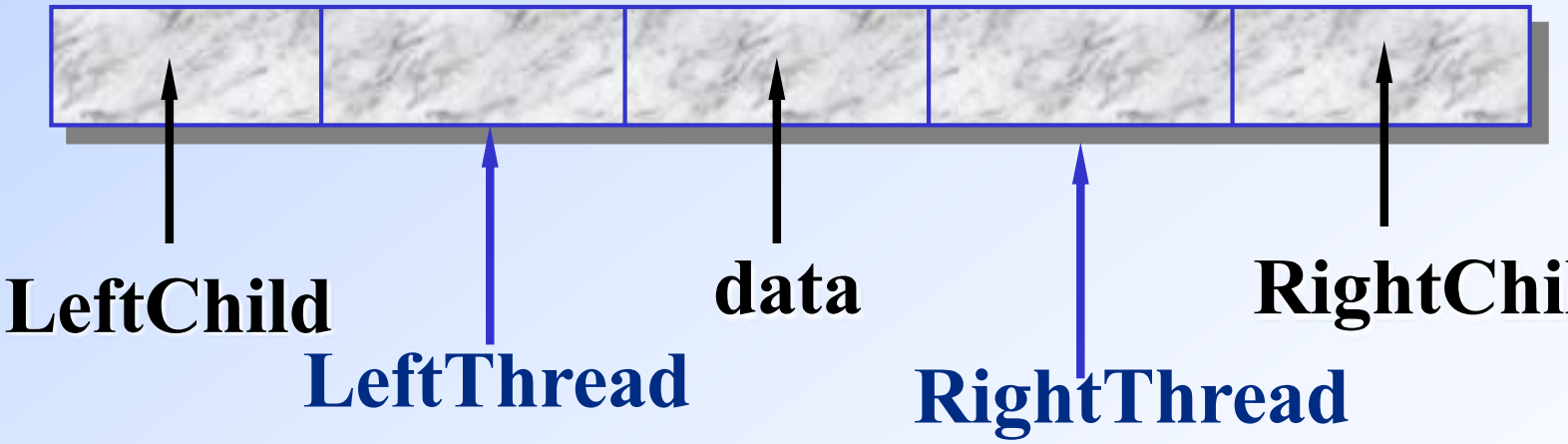
## ❖线索二叉树 (Threaded Binary Tree)

对二叉树遍历的过程实质上是对一个非线性结构进行线性化的操作.

以二叉链表为存储结构时,结点的左右孩子可以直接找到,但不能直接找到结点的前驱和后继,而结点的前驱和后继只有在遍历二叉树的过程中才能得到.

用二叉链表表示的二叉树中,n个结点的二叉树有n+1个空链域，可利用这些空链域存储结点的前驱或后继。

# 结点结构



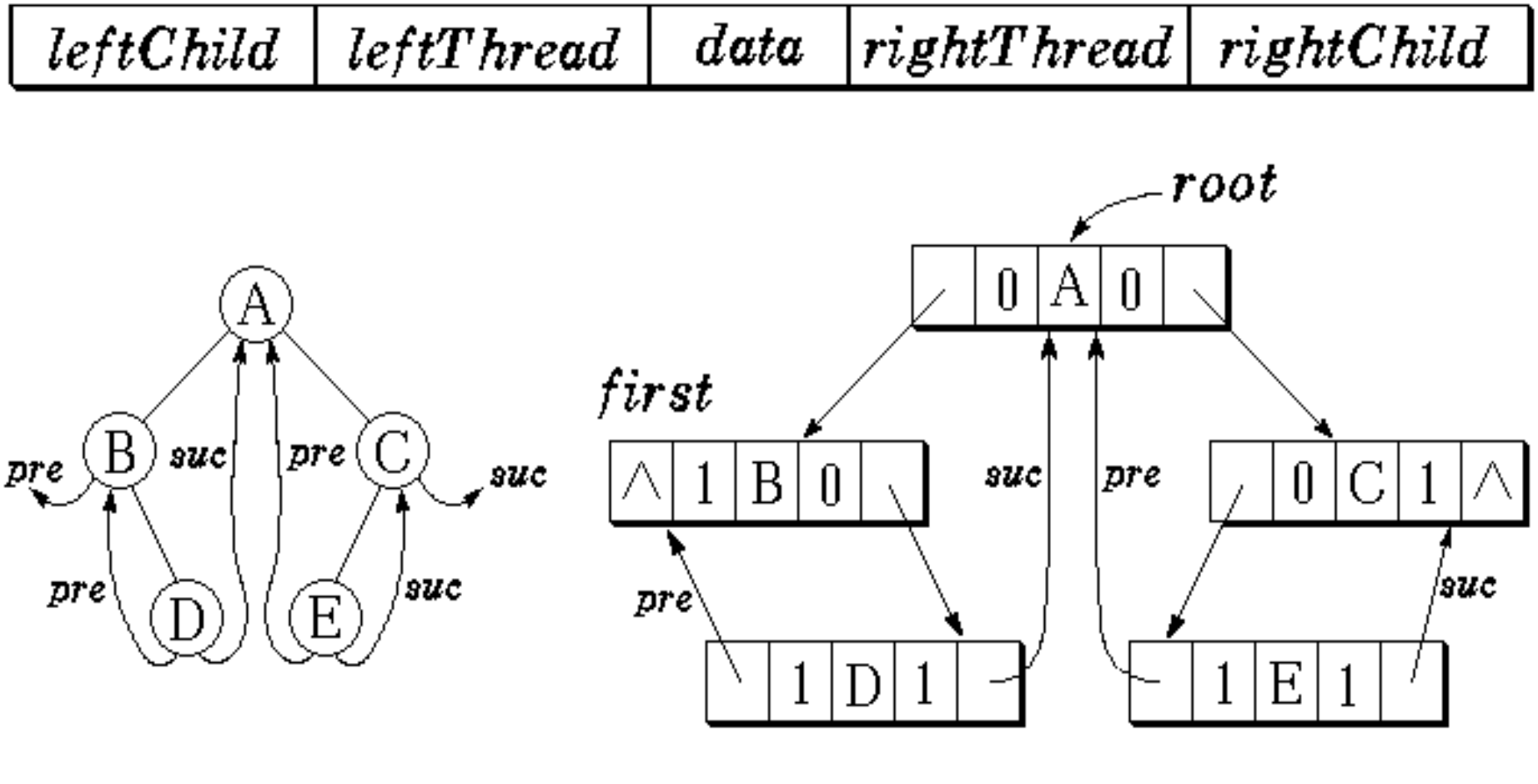

LeftThread=0, LeftChild为左子女指针

LeftThread=1, LeftChild为前驱线索

RightThread=0, RightChild为右子女指针

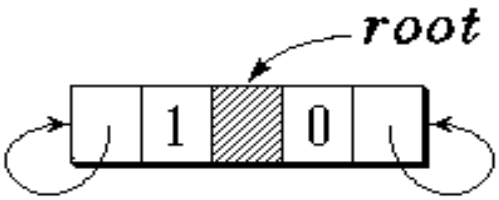
RightThread=1, RightChild为后继指针

```
typedef enum {Link,Thread} PointerTag;
//PointerTag为标志，Link=0代表指针，
Thread=1代表线索
typedef struct BiThrNode{
        TElemType data;
        struct BiThrNode *lchild, *rchild;
        PointerTag LTag,RTag;
}BiThrNode, *BiThrTree;
```



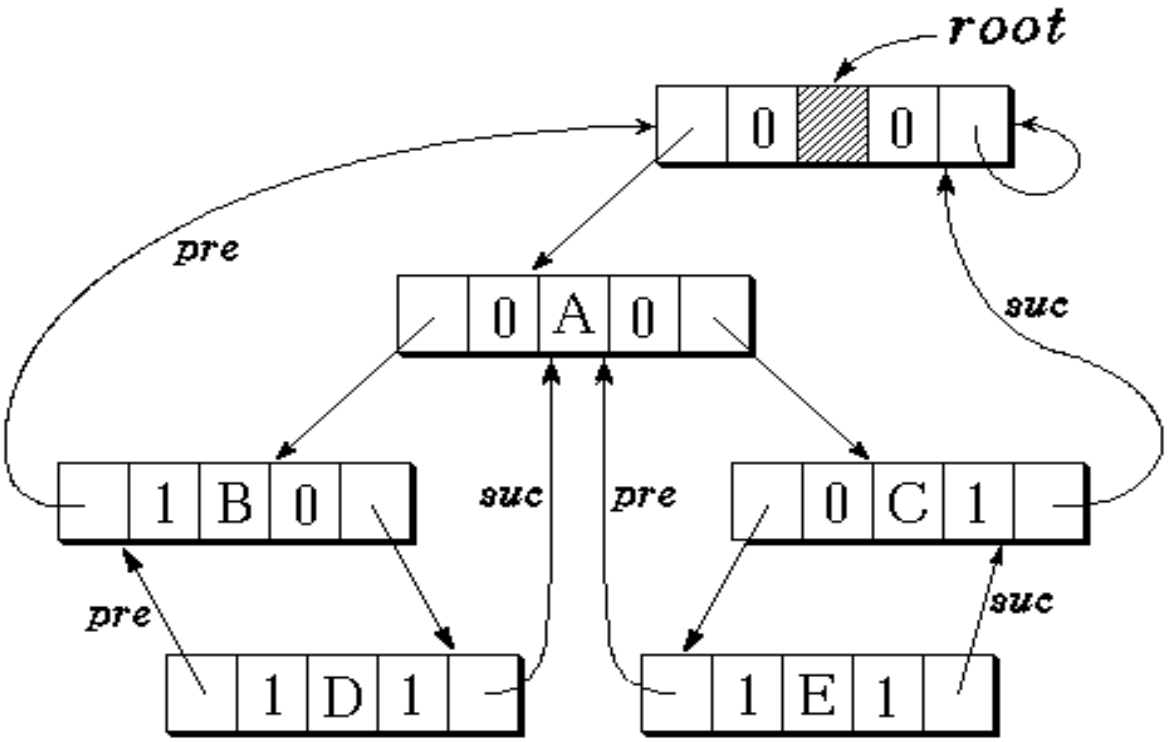

# 中序线索二叉树BDAEC

# 带表头结点的中序线索二叉树

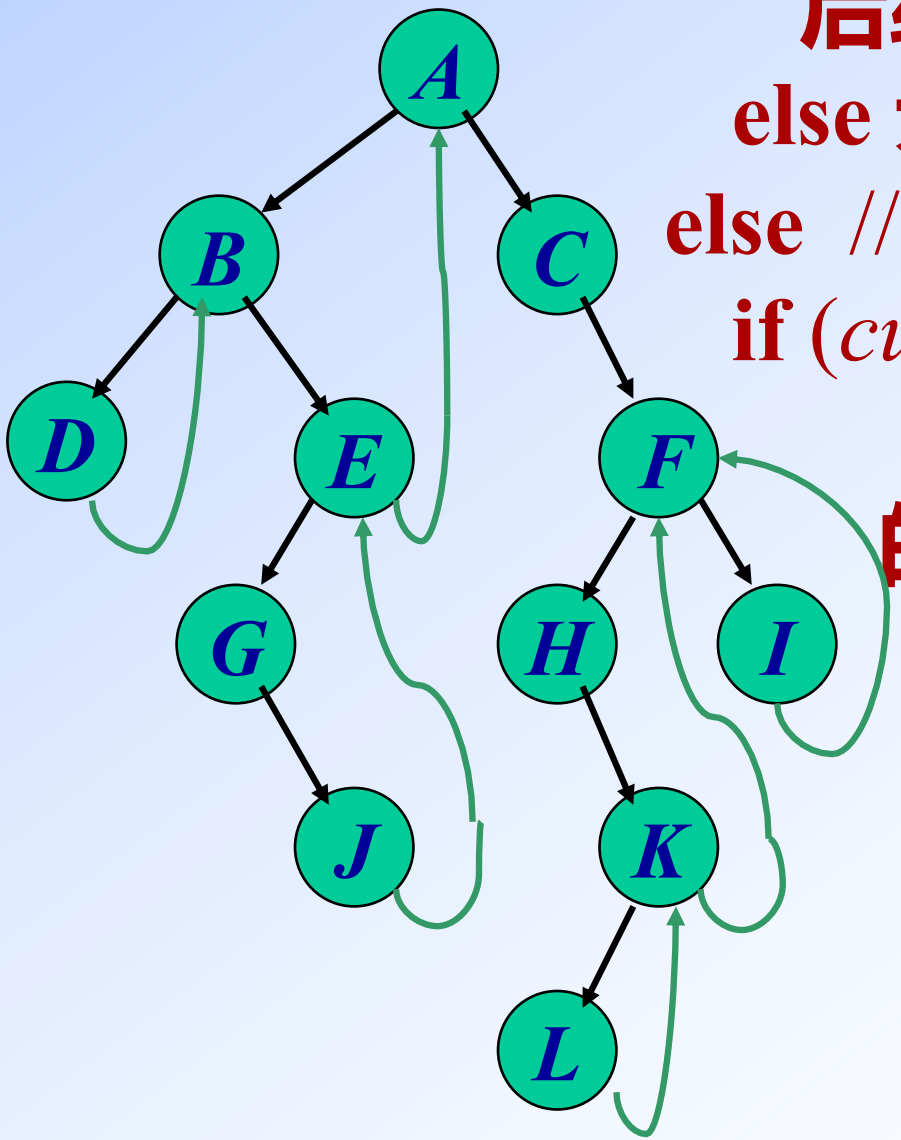



（a）空二叉链表

（b）非空二叉链表



# 寻找当前结点在中序下的后继




if (*current→rightThread* ==1)

if (*current→rightChild* !=*T.root*)

后继为 *current→rightChild*

else 无后继

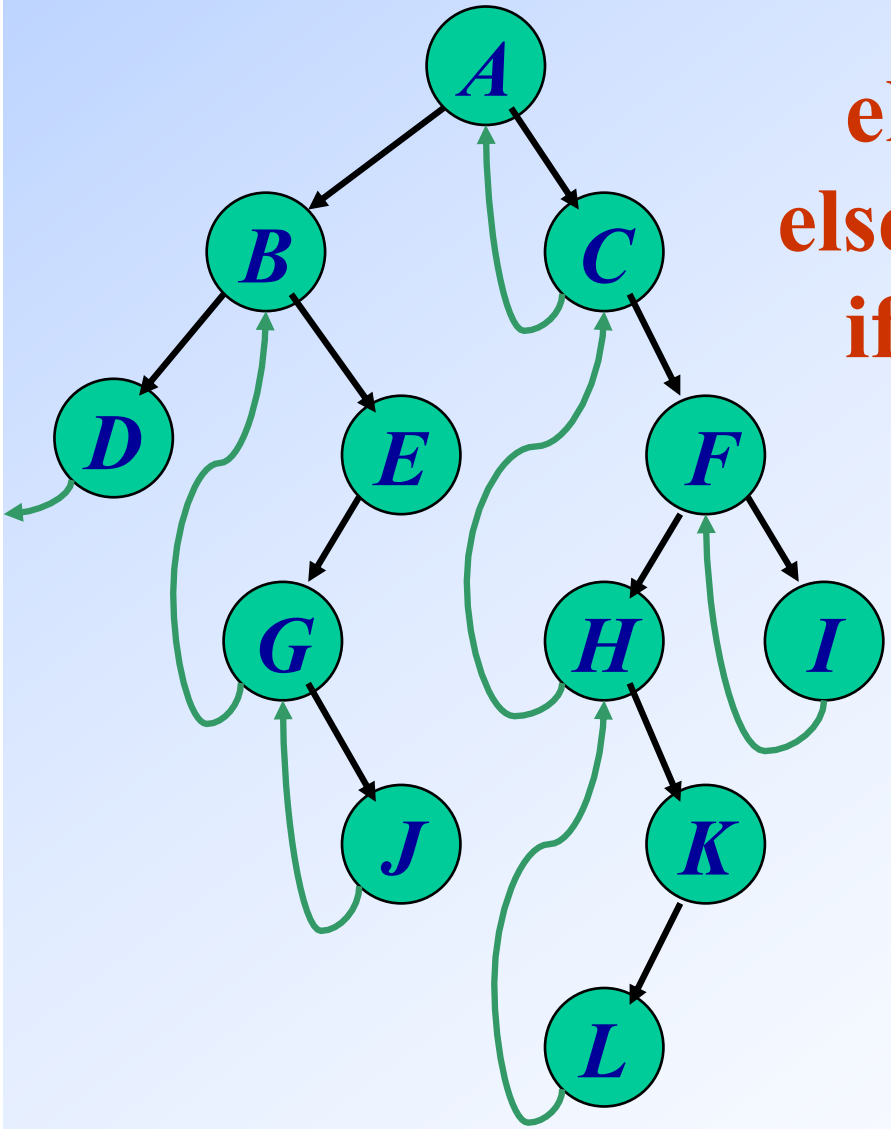
else //*current→rightThread* !=1

if (*current→rightChild* !=*T.root*)

后继为当前结点右子树的中序下的第一个结点

else 出错情况

# 寻找当前结点在中序下的前驱




if (*current*→*leftThread* ==1)

  if (*current*→*leftChild* !=*T.root*)

    前驱为*current*→*leftChild*

  else 无前驱

else //*current*→*leftThread* ==0

  if (*current*→*leftChild* !=*T.root*)

    前驱为当前结点左子树的中序下的最后一个结点

  else 出错情况

```
Status InOrderTraverse_Thr(BiThrTree T, Status
(*Visit)(TElemType e)){
        p=T->lchild;
        while(p!=T){
                while(p->LTag==Link)p=p->lchild;
                if(!Visit(p->data)) return ERROR;
                while(p->RTag==Thread && p-
>rchild!=T){
                        p=p->rchild; Visit(p->data);
                }
                p=p->rchild;
        }
}//InOrderTraverse_Thr中序遍历线索二叉树
```

## 后序线索二叉树中寻找后继

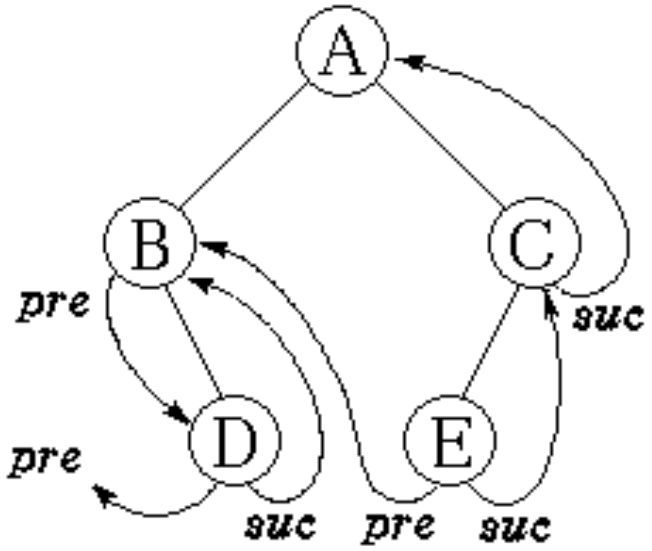
后继：(1)根的后继为空。

(2)如果结点是双亲的右孩子或是左孩子但双亲没有右子树，则后继是其双亲。

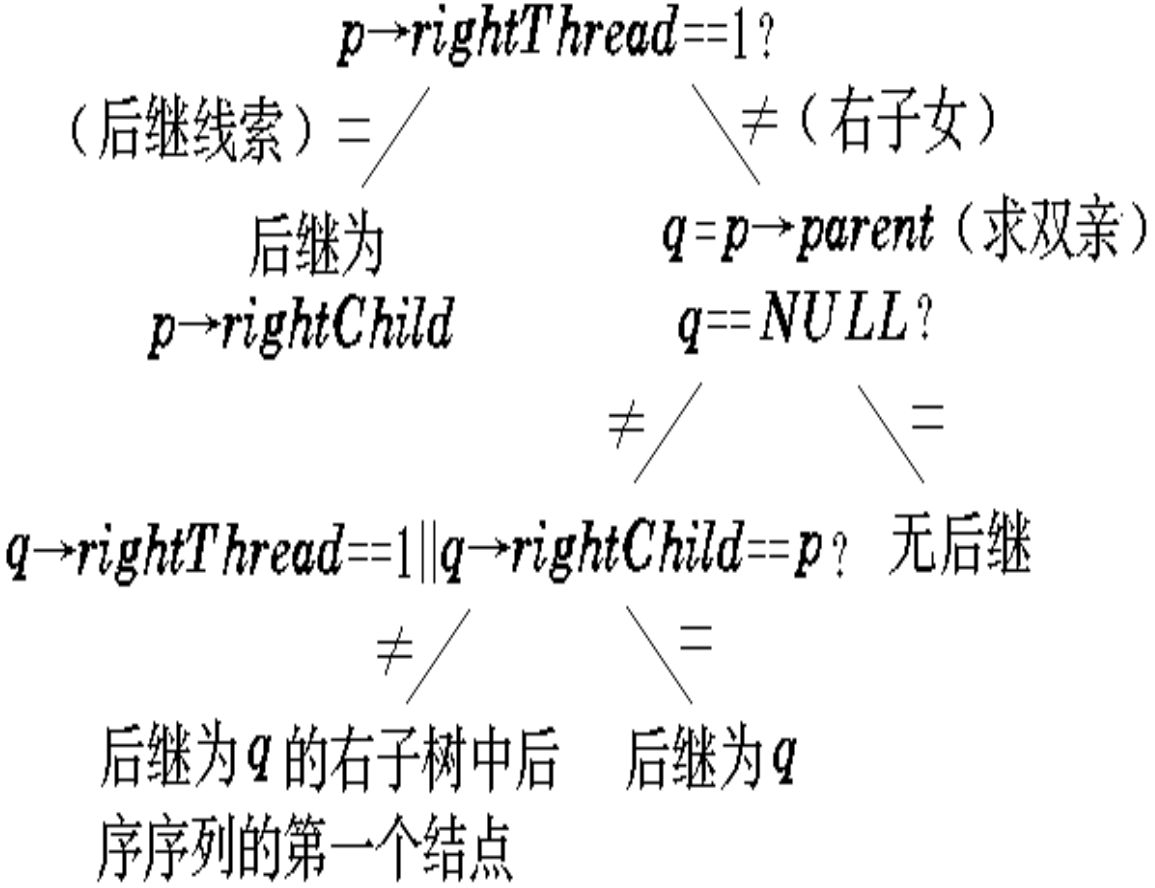
(3)如果结点是双亲的左孩子且双亲有右子树，则后继是双亲右子树上先遍历到的结点。

总之，如果二叉树的应用主要是遍历或查找结点的前驱和后继则使用线索二叉树更为合适。

# 后序线索化二叉树




（b）后序穿线二叉树



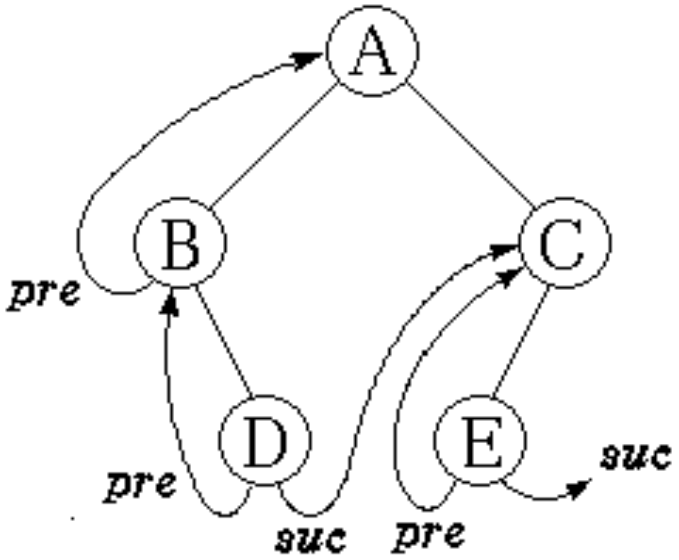

（a）求结点 $p$ 的后继

**后序序列**

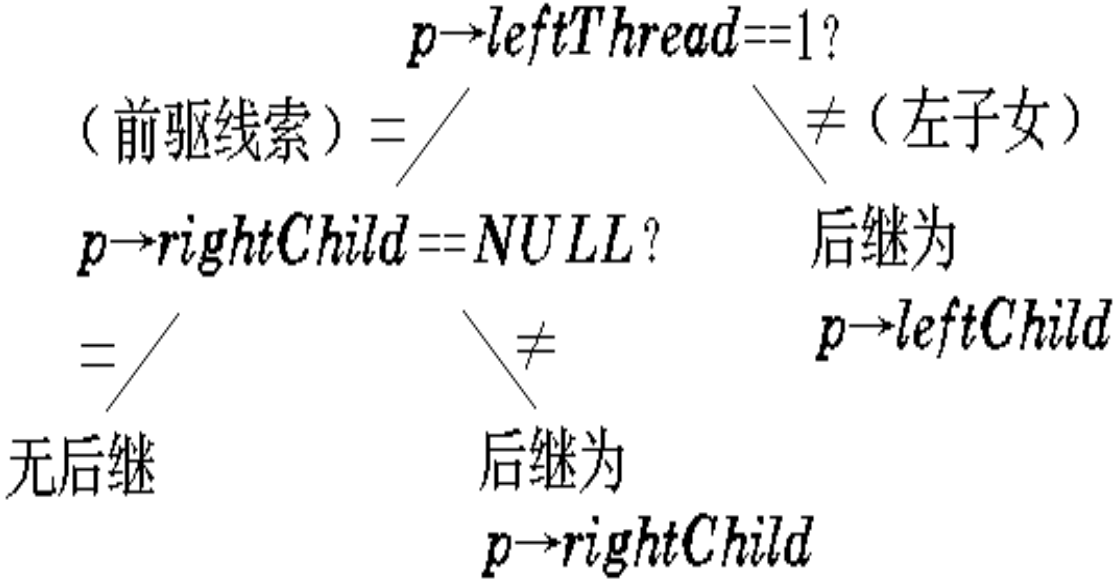
***D B E C A***

**在后序线索化二叉树中**
**寻找当前结点的后继**

# 前序线索化二叉树




（a）前序穿线二叉树




（a）求结点 p 的后继

前序序列

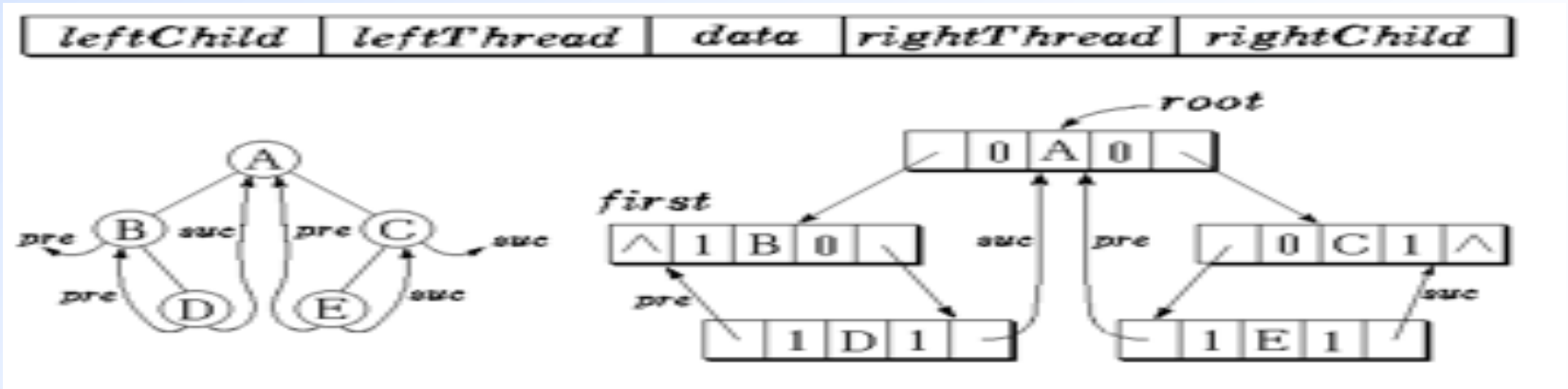
*A B D C E*

**在前序线索化二叉树中**
**寻找当前结点的后继**

# 通过中序遍历建立中序线索化二叉树

```
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T){
//中序遍历并线索化二叉树，Thrt为头结点
    if(!(Thrt=(BiThrTree)malloc(sizeof(BiThrNode))))
    exit(OVERFLOW); //建头结点
    Thrt->LTag=Link;
    Thrt->RTag=Thread;
    Thrt->rchild=Thrt; //头结点右指针回指
    if(!T)Thrt->lchild=Thrt; //如果二叉树为空，则左指针也回指
    else{
          Thrt->lchild=T; pre=Thrt;//pre总是指向最后一个访问过的结
    点，也就是其后要访问结点的前驱
          InThreading(T);//线索化的主过程
          pre->rchild=Thrt; pre->RTag=Thread;
          Thrt->rchild=pre;//将最后一个结点线索化
    }
    return OK;
}//InOrderThreading
```

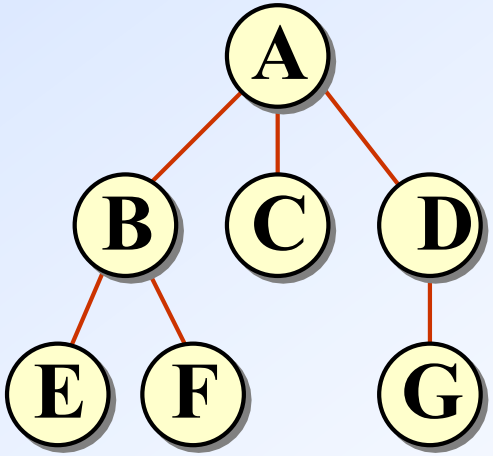
```
void InThreading(BiThrTree p){
        if(p){
                InThreading(p->lchild);//左子树线索化
                if(!p->lchild) {p->LTag=Thread; p->lchild=pre;}//建立
前驱线索
                if(!pre->rchild) {pre->RTag=Thread; pre-
>rchild=p;}//建立后继线索
                pre=p;//保持pre为下一结点的前驱
                InThreading(p->rchild); //右子树线索化
        }
}//InThreading
```



# 树与森林

## ❑树的存储结构

▪双亲表示：以一组连续空间存储树的结点，同时在结点中附设一个指针，存放双亲结点在链表中的位置。该方法利用每个结点只有一个双亲的特点，可以很方便求结点的双亲，但不方便求结点的孩子。

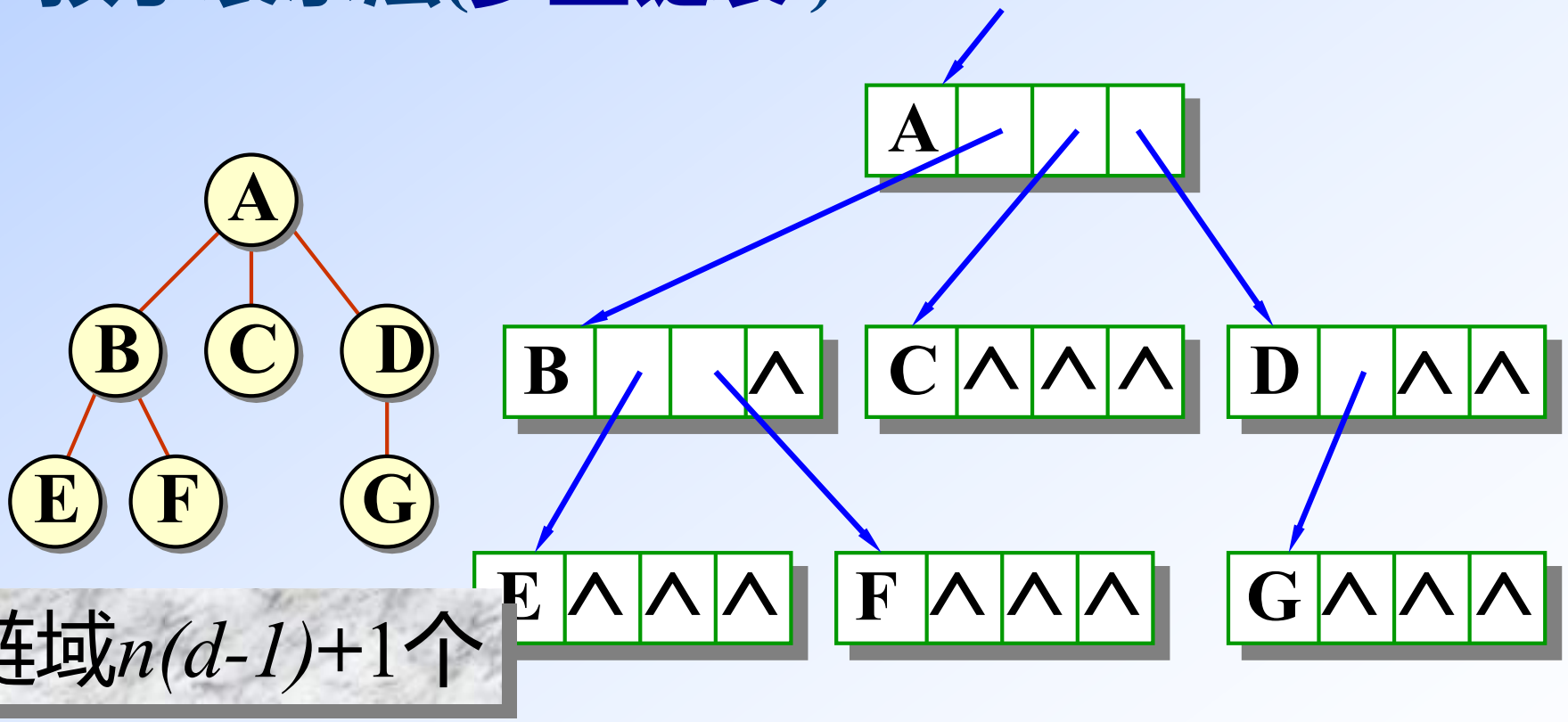



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| data | A | B | C | D | E | F | G |
| parent | -1 | 0 | 0 | 0 | 1 | 1 | 3 |

# 用双亲表示实现的树定义

```
#define MaxSize      100      //最大结点个数

typedef char TreeData;   //结点数据

typedef struct {                  //树结点定义
    TreeData data;
    int parent;
} TreeNode;

typedef TreeNode Tree[MaxSize];   //树
```

## ▪孩子表示法(多重链表 )




空链域$n(d-1)+1$个

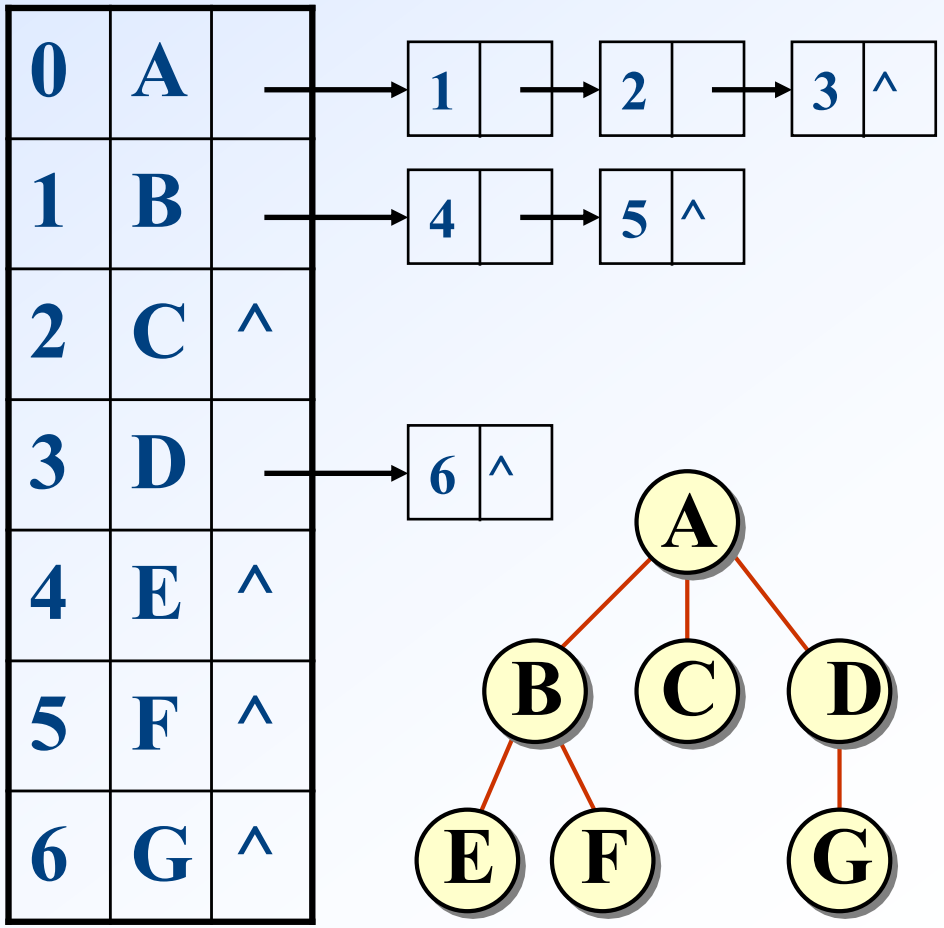
**第一种解决方案　等数量的链域 (d为树的度)**

| data | $child_1$ | $child_2$ | $child_3$ | ····· | $child_d$ |
|---|---|---|---|---|---|

# 第二种解决方案 孩子链表

将每个结点的孩子链在该结点之后组成链表，再将所有头结点组成一个线性表。

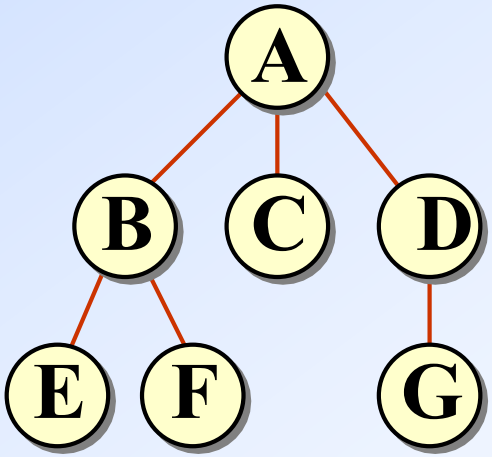
```
typedef struct CTNode {
        int child;
        struct CTNode *next;
}*ChildPtr;
typedef struct{
        TElemType data;
        ChildPtr firstchild;
}CTBox;
typedef struct{
        CTBox
nodes[MAX_TREE_SIZE];
        int n,r;//结点数和根的位置
}CTree;
```



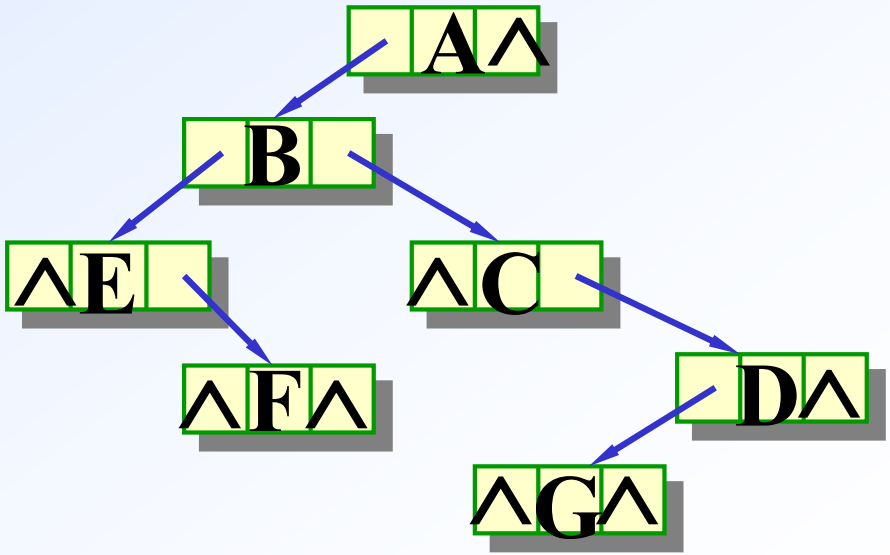
# 树的左子女-右兄弟表示

## （二叉链表表示）

结点结构

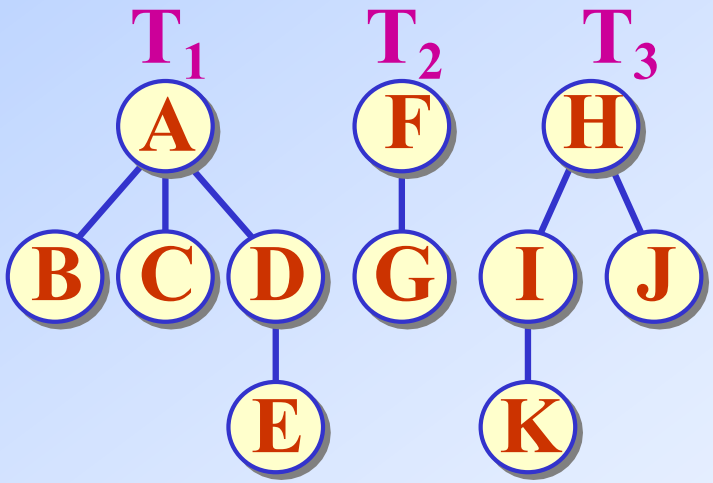

空链域$n$+1个



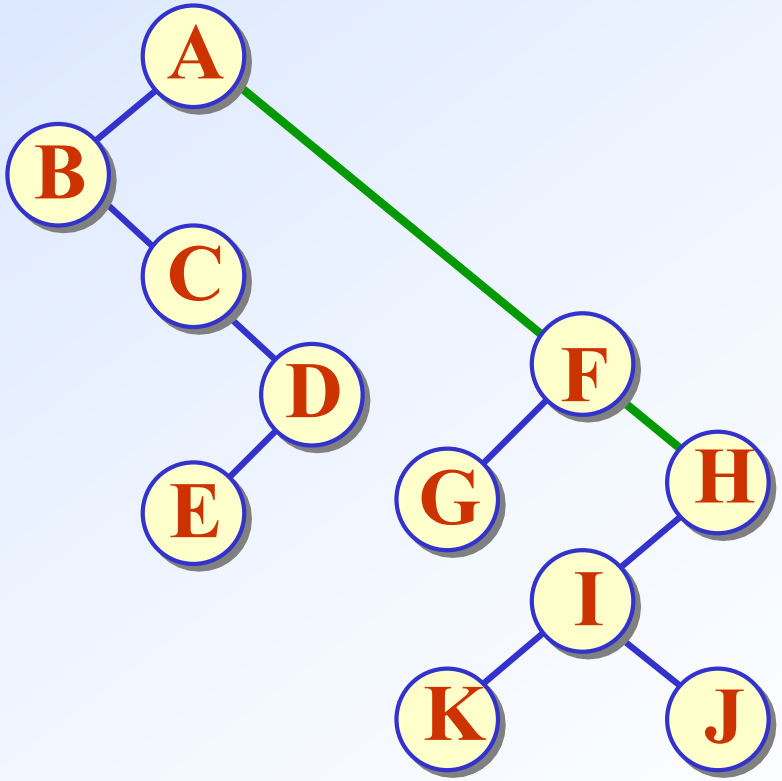
# 用左子女-右兄弟表示实现的树定义

```
typedef char TreeData;

typedef struct node {
    TreeData data;
    struct node *firstChild, *nextSibling;
} TreeNode;

typedef TreeNode * Tree;
```

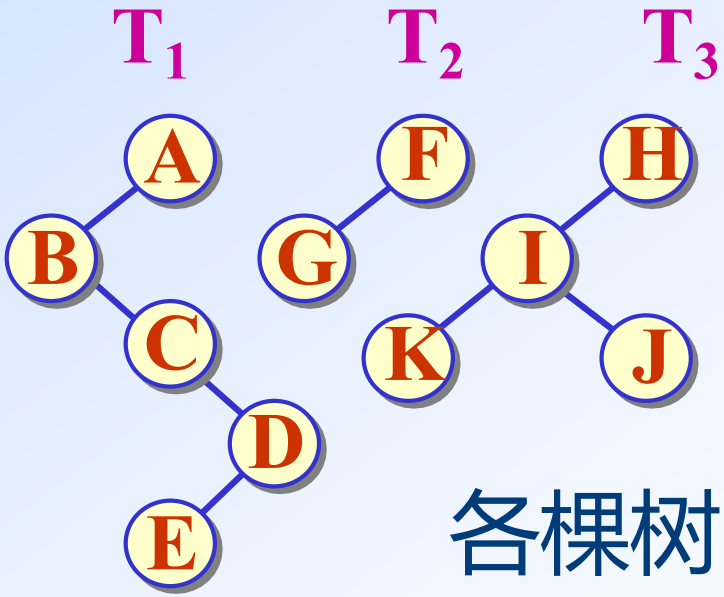
## ❑森林与二叉树的转换




3 棵树的森林

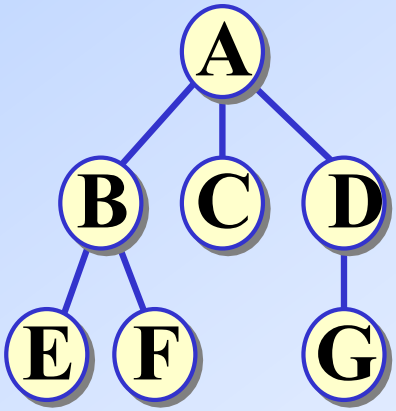


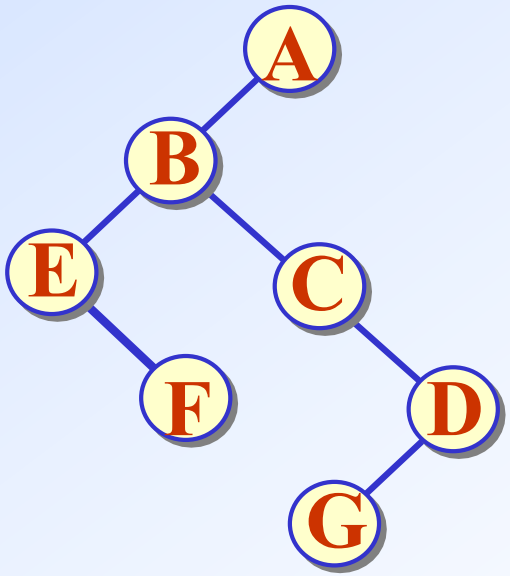

森林的二叉树表示




各棵树的二叉树表示

# 树的二叉树表示:




- 树的遍历

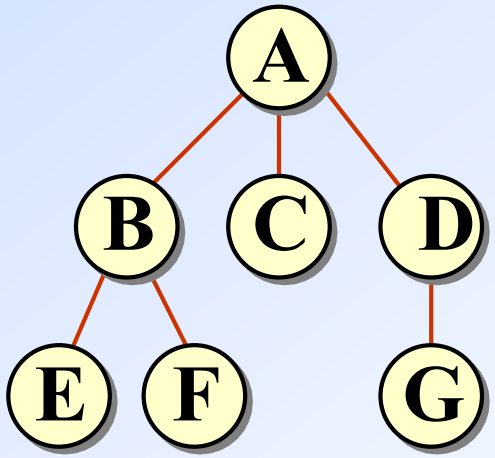
深度优先遍历

先根次序遍历

后根次序遍历

# 深度优先遍历

## ▪树的先根次序遍历




当树非空时

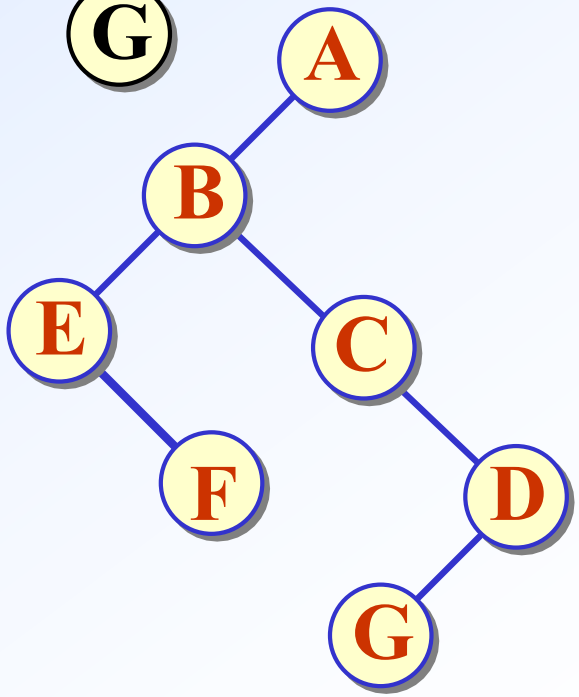
{访问根结点

依次先根遍历根的各棵子树}

树先根遍历 ABEFCDG

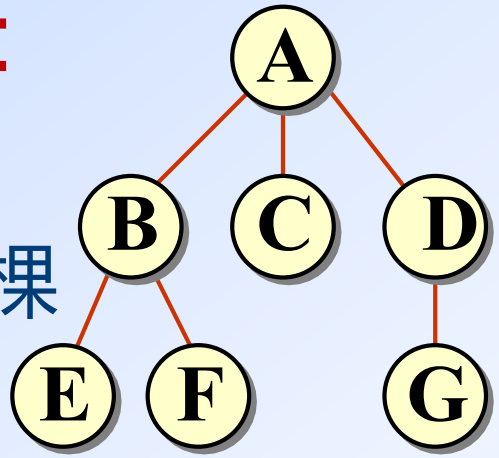
对应二叉树前序遍历 ABEFCDG

树的先根遍历结果与其对应二叉树表示的前序遍历结果相同

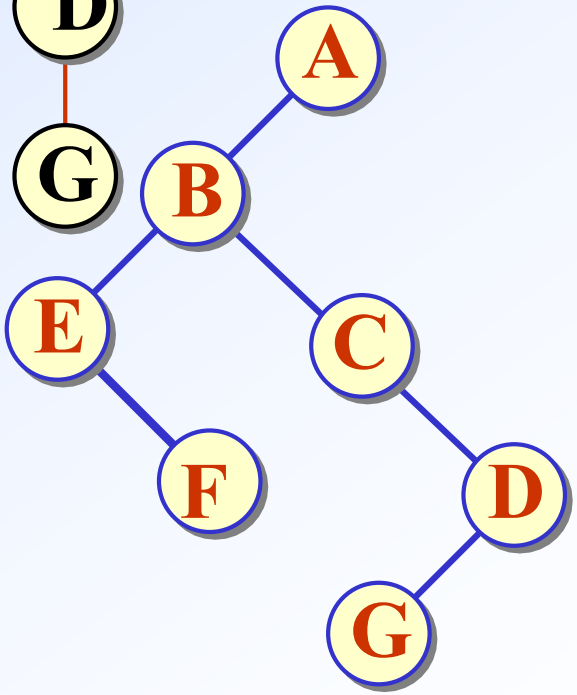



树的先根遍历可以借助对应二叉树的前序遍历算法实现

## ▪树的后根次序遍历:




当树非空时
{依次后根遍历根的各棵
子树
访问根结点}
树后根遍历 EFBCGDA
对应二叉树中序遍历 EFBCGDA
树的后根遍历结果与其对应二叉树
表示的中序遍历结果相同

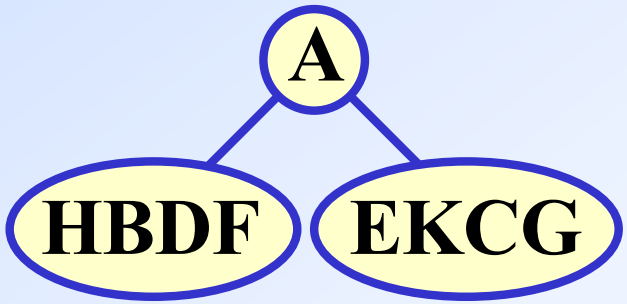


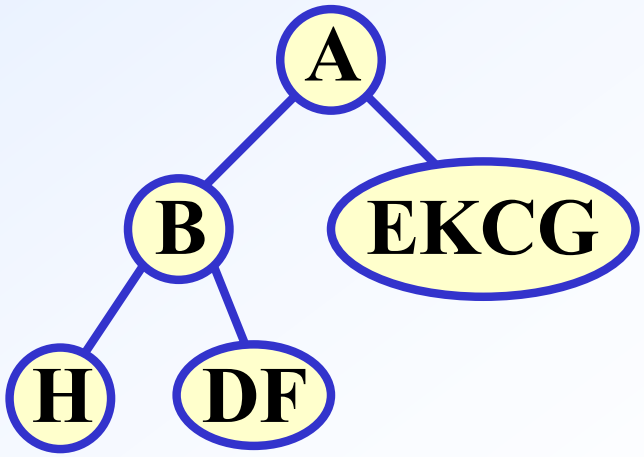

树的后根遍历可以借助对应二叉树的中序遍历算法
实现

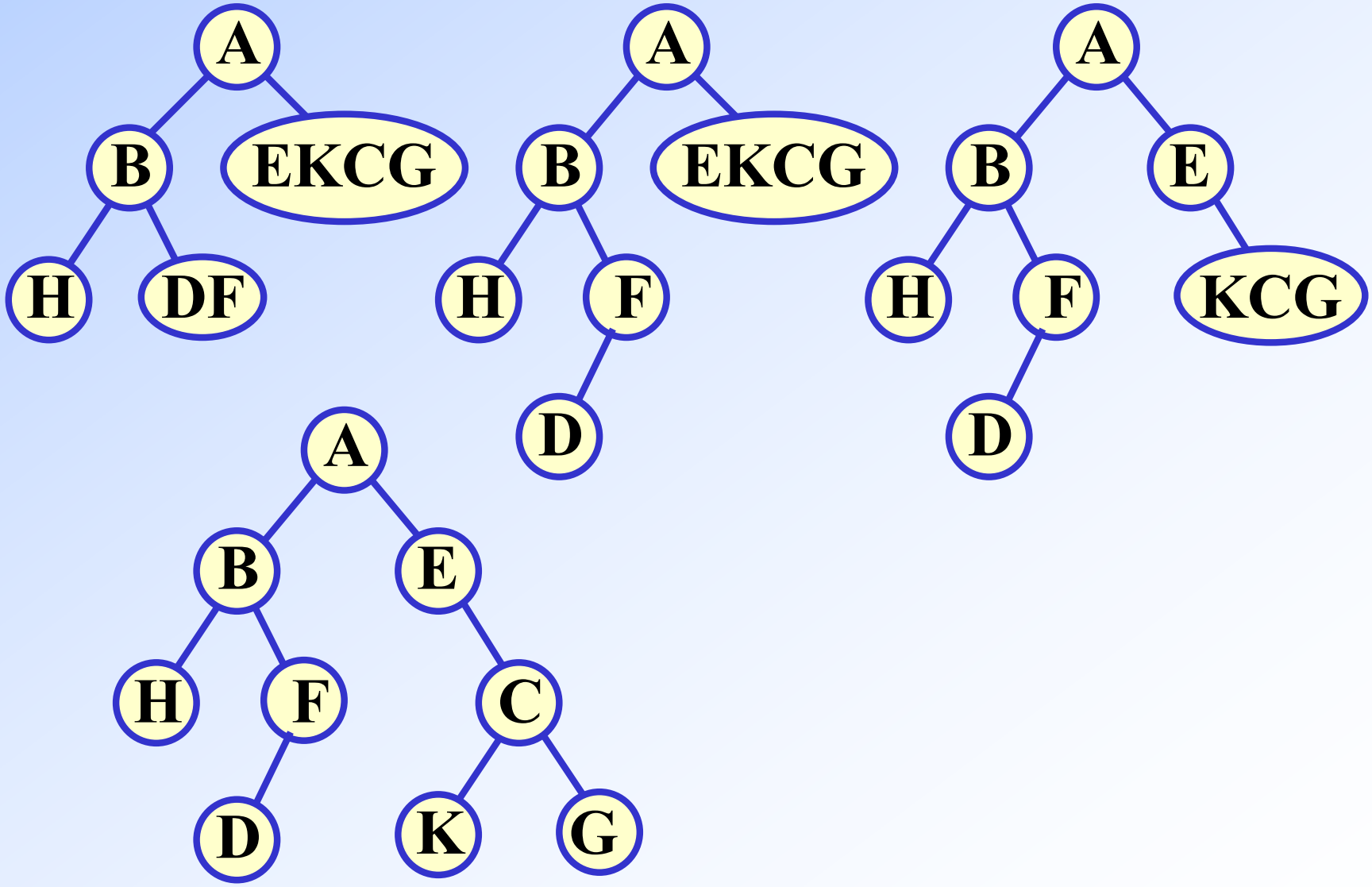
# 二叉树的计数

- 由二叉树的前序序列和中序序列可唯一地确定一棵二叉树。

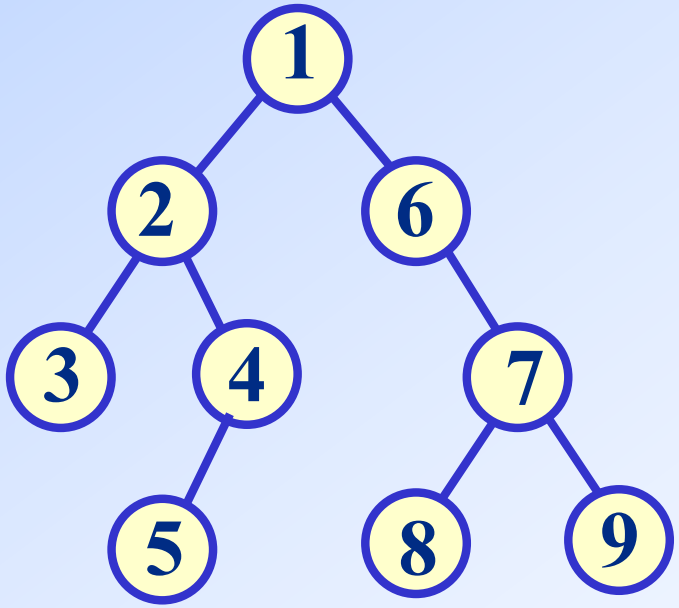
例, 前序序列 { ABHFDECKG } 和中序序列 { HBDFAEKCG }, 构造二叉树过程如下：

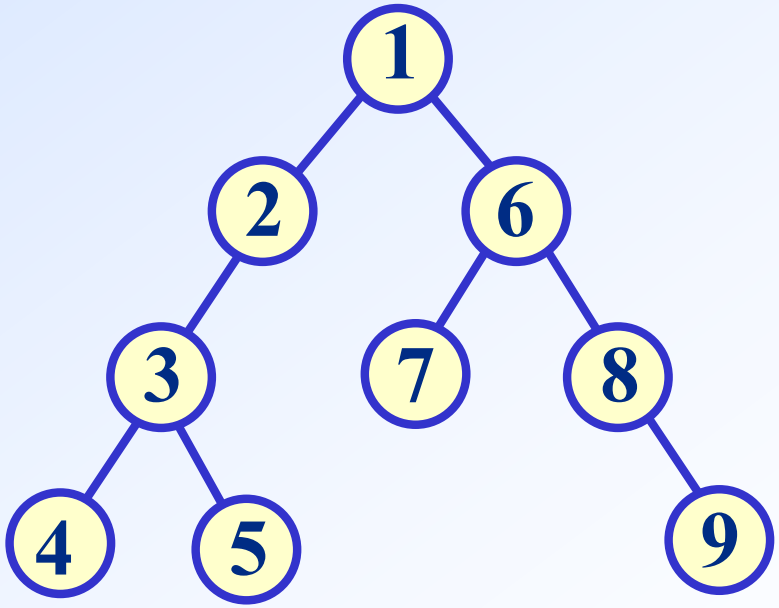



A
B
EKCG
H
DF
A
B
EKCG
H
F
D
A
B
E
H
F
KCG
D
A
B
E
H
F
C
D
K
G

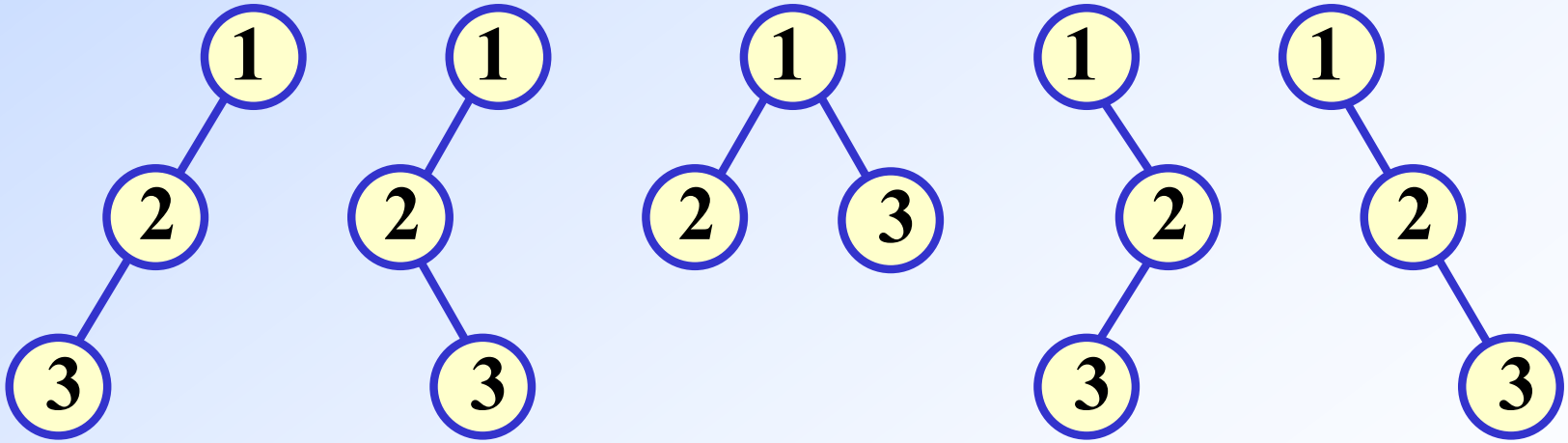
如果前序序列固定不变，给出不同的中序序列，可得到不同的二叉树。




•问题是有 $n$ 个数据值，可能构造多少种不同的二叉树？我们可以固定前序排列，选择所有可能的中序排列。

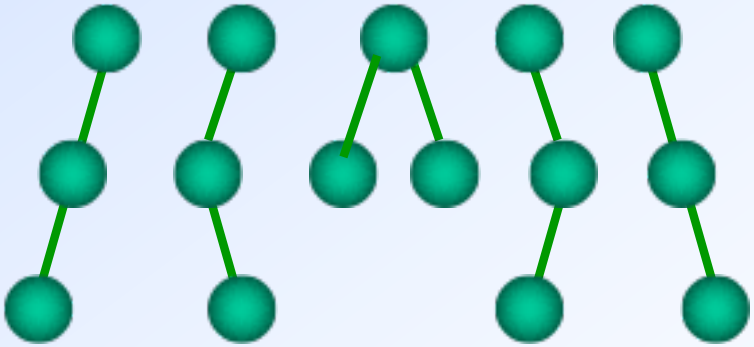
例如，有 3 个数据 { 1, 2, 3 }，可得 5 种不同的二叉树。它们的前序排列均为 123，中序序列可能是 321 ， 231， 213， 132，123.



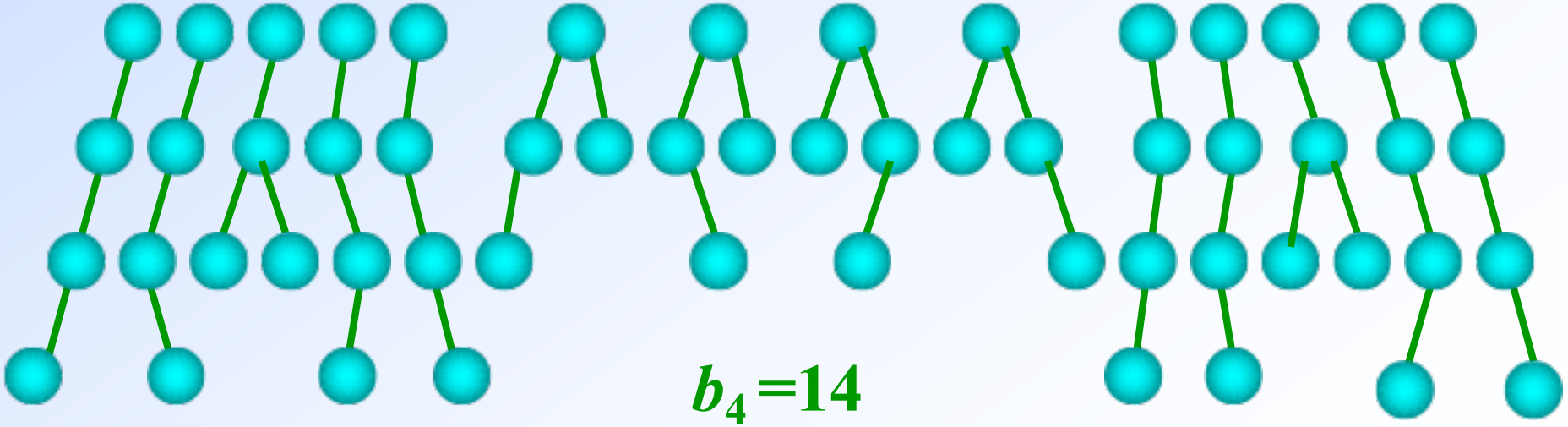

- 具有n个结点不同形态的树的数目和具有n-1个结点互不相似的二叉树的数目相同。

# 有0个, 1个, 2个, 3个结点的不同二叉树如下

$b_0=1$ $b_1=1$ $b_2=2$



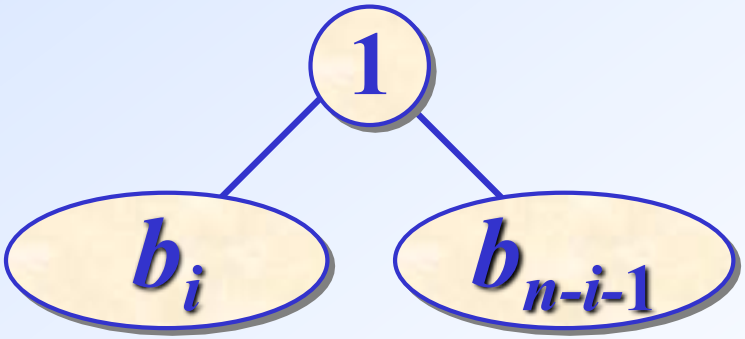
$b_3=5$



$b_4=14$

## ❖计算具有 n 个结点的不同二叉树的棵数

$$b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1}$$




**最终结果：**

$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1} \frac{(2n)!}{n! \cdot n!}$$

# 霍夫曼树 (Huffman Tree)

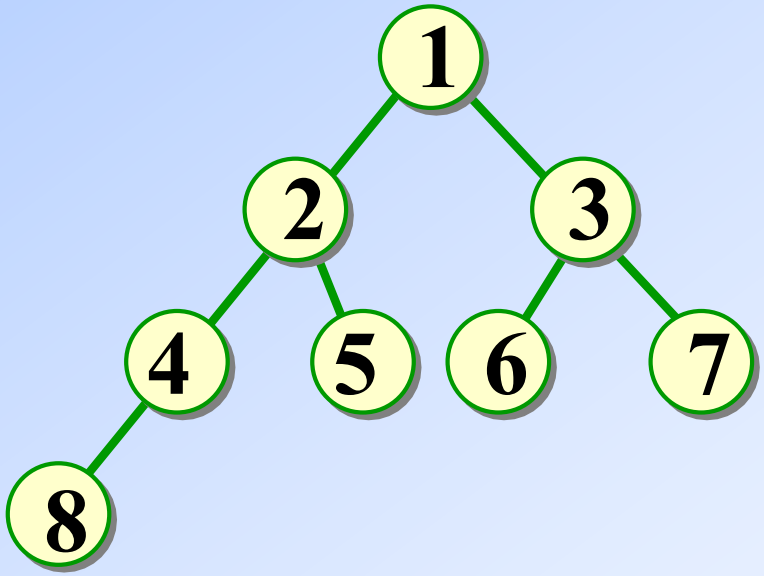
- **路径长度 (Path Length)**

**两个结点之间的路径长度 PL 是连接两结点的路径上的分支数。**

**树的外部路径长度是各叶结点(外结点)到根结点的路径长度之和 EPL。**

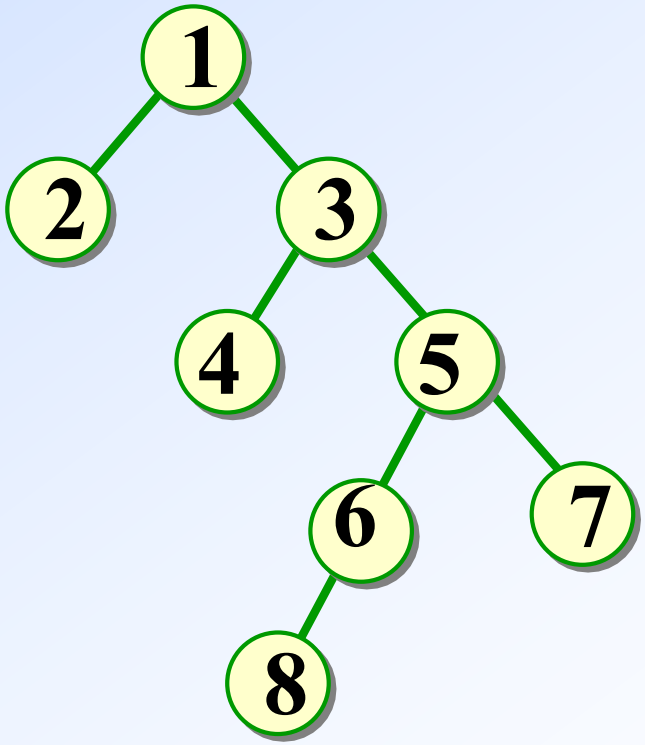
**树的内部路径长度是各非叶结点(内结点)到根结点的路径长度之和 IPL。**

**树的路径长度 PL = EPL + IPL**




**树的外部路径长度**

**EPL = 3*1+2*3 = 9**



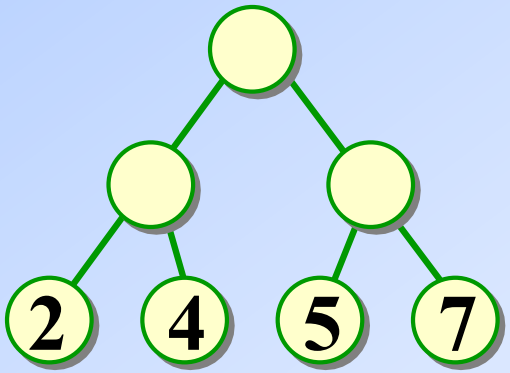

**树的外部路径长度**

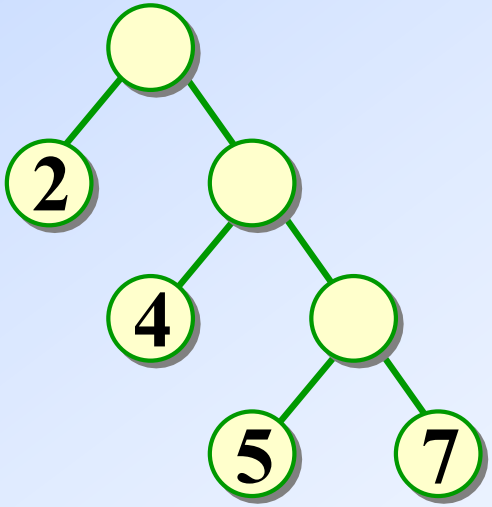
**EPL = 1*1+2*1+3*1+4*1 = 10**

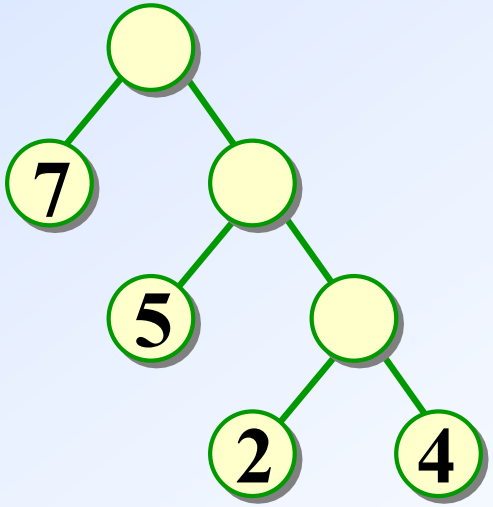
- **带权路径长度 (Weighted Path Length, WPL)**

二叉树的带权 (外部) 路径长度是树的各叶结点所带的权值 $w_i$ 与该结点到根的路径长度 $l_i$ 的乘积的和。

$$WPL = \sum_{i=0}^{n-1} w_i * l_i$$




WPL = 2*2+
4*2+5*2+
7*2 = 36

WPL = 2*1+
4*2+5*3+
7*3 = 46

WPL = 7*1+
5*2+2*3+
4*3 = 35

带权(外部)路径
长度达到最小

# 霍夫曼树

带权路径长度达到最小的二叉树即为霍夫曼树。

在霍夫曼树中，权值大的结点离根最近。

## 霍夫曼算法

**(1) 由给定的 n 个权值 $\{w_0, w_1, w_2, \cdots, w_{n-1}\}$，构造具有 n 棵扩充二叉树的森林 $F = \{T_0, T_1, T_2, \cdots, T_{n-1}\}$，其中每棵扩充二叉树 $T_i$ 只有一个带权值 $w_i$ 的根结点, 其左、右子树均为空。**

(2) 重复以下步骤, 直到 F 中仅剩下一棵树为止：

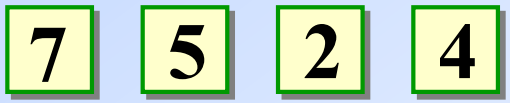
① 在 F 中选取两棵根结点的权值最小的扩充二叉树， 做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

② 在 F 中删去这两棵二叉树。

③ 把新的二叉树加入 F。

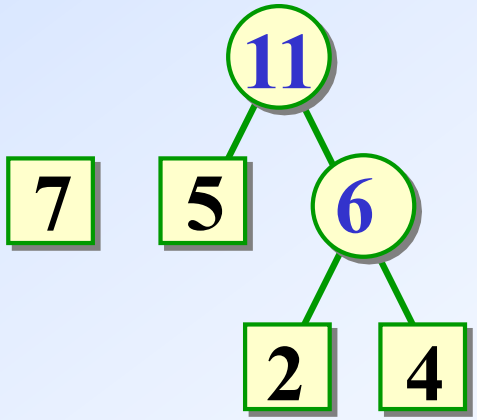
# 举例霍夫曼树的构造过程

F : {7} {5} {2} {4}



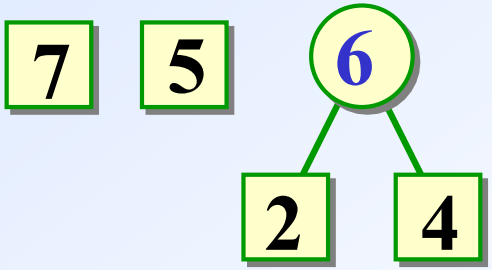

初始

F : {7} {11}



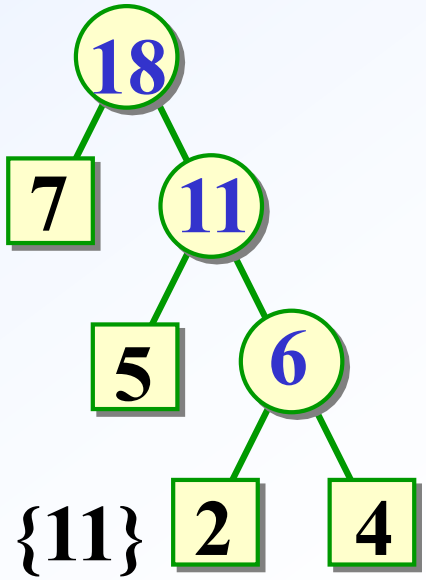

合并{5} {6}

F : {7} {5} {6}




合并{2} {4}

F : {18}

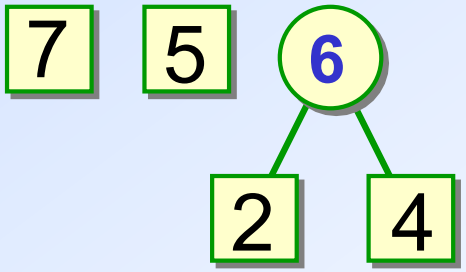



合并{7} {11}

# 上例存储结构

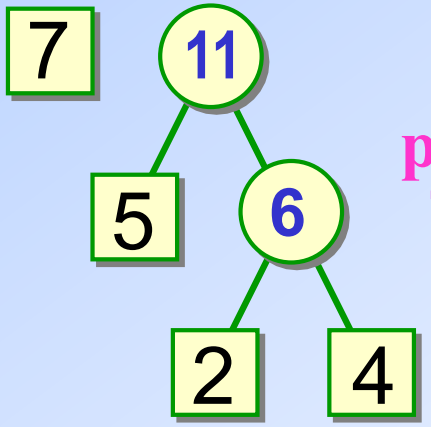

| | Weight | parent | leftChild | rightChild |
|---|---|---|---|---|
| 0 | 7 | -1 | -1 | -1 |
| 1 | 5 | -1 | -1 | -1 |
| 2 | 2 | -1 | -1 | -1 |
| 3 | 4 | -1 | -1 | -1 |
| 4 | | -1 | -1 | -1 |
| 5 | | -1 | -1 | -1 |
| 6 | | -1 | -1 | -1 |

初 态

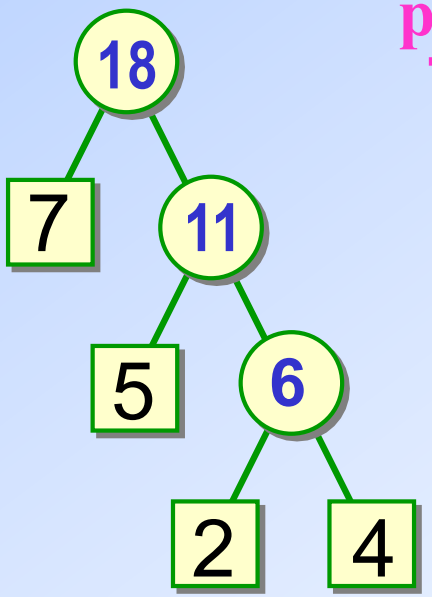



| | Weight | parent | leftChild | rightChild |
|---|---|---|---|---|
| 0 | 7 | -1 | -1 | -1 |
| 1 | 5 | -1 | -1 | -1 |
| p1 → 2 | 2 | 4 ~~-1~~ | -1 | -1 |
| p2 → 3 | 4 | 4 ~~-1~~ | -1 | -1 |
| i → 4 | 6 | -1 | 2 ~~-1~~ | 3 ~~-1~~ |
| 5 | | -1 | -1 | -1 |
| 6 | | -1 | -1 | -1 |

过 程




| | Weight | parent | leftChild | rightChild |
|---|---|---|---|---|
| 0 | 7 | -1 | -1 | -1 |
| p1 → 1 | 5 | ~~-1~~ 5 | -1 | -1 |
| 2 | 2 | 4 | -1 | -1 |
| 3 | 4 | 4 | -1 | -1 |
| p2 → 4 | 6 | ~~-1~~ 5 | 2 | 3 |
| i → 5 | 11 | -1 | ~~-1~~ 1 | ~~-1~~ 4 |
| 6 | | -1 | -1 | -1 |




| | Weight | parent | leftChild | rightChild |
|---|---|---|---|---|
| p1 → 0 | 7 | 6 ~~-1~~ | -1 | -1 |
| 1 | 5 | 5 | -1 | -1 |
| 2 | 2 | 4 | -1 | -1 |
| 3 | 4 | 4 | -1 | -1 |
| 4 | 6 | 5 | 2 | 3 |
| p2 → 5 | 11 | 6 ~~-1~~ | 1 | 4 |
| *i* → 6 | 18 | -1 | 0 ~~-1~~ | 5 ~~-1~~ |

**终 态**

# 霍夫曼树的定义

```
const  int n = 20;//叶结点数
const int m = 2*n -1;//结点数

typedef struct {
    float weight;
    int parent, leftChild, rightChild;
} HTNode;

typedef HTNode HuffmanTree[m];
```

# 建立霍夫曼树的算法

```
void CreateHuffmanTree ( HuffmanTree T,
        float fr[ ] ) {
    for ( int i = 0; i < n; i++ )
        T[i].weight = fr[i];
    for ( i = 0; i < m; i++ ) {
        T[i].parent = -1;
        T[i].leftChild = -1;
        T[i].rightChild = -1;
    }
    for ( i = n; i < m; i++ ) {
```

```
int min1 = min2 = MaxNum;
int pos1, pos2;
for ( int j = 0; j < i; j++ )
   if ( T[j].parent == -1 )
      if ( T[j].weight < min1 )
         { pos2 = pos1;  min2 = min1;
            pos1 = j;  min1 = T[j].weight;
         }
      else if ( T[j].weight < min2 )
         { pos2 = j;  min2 = T[j].weight; }
T[i].leftChild = pos1;
T[i].rightChild = pos2;
```

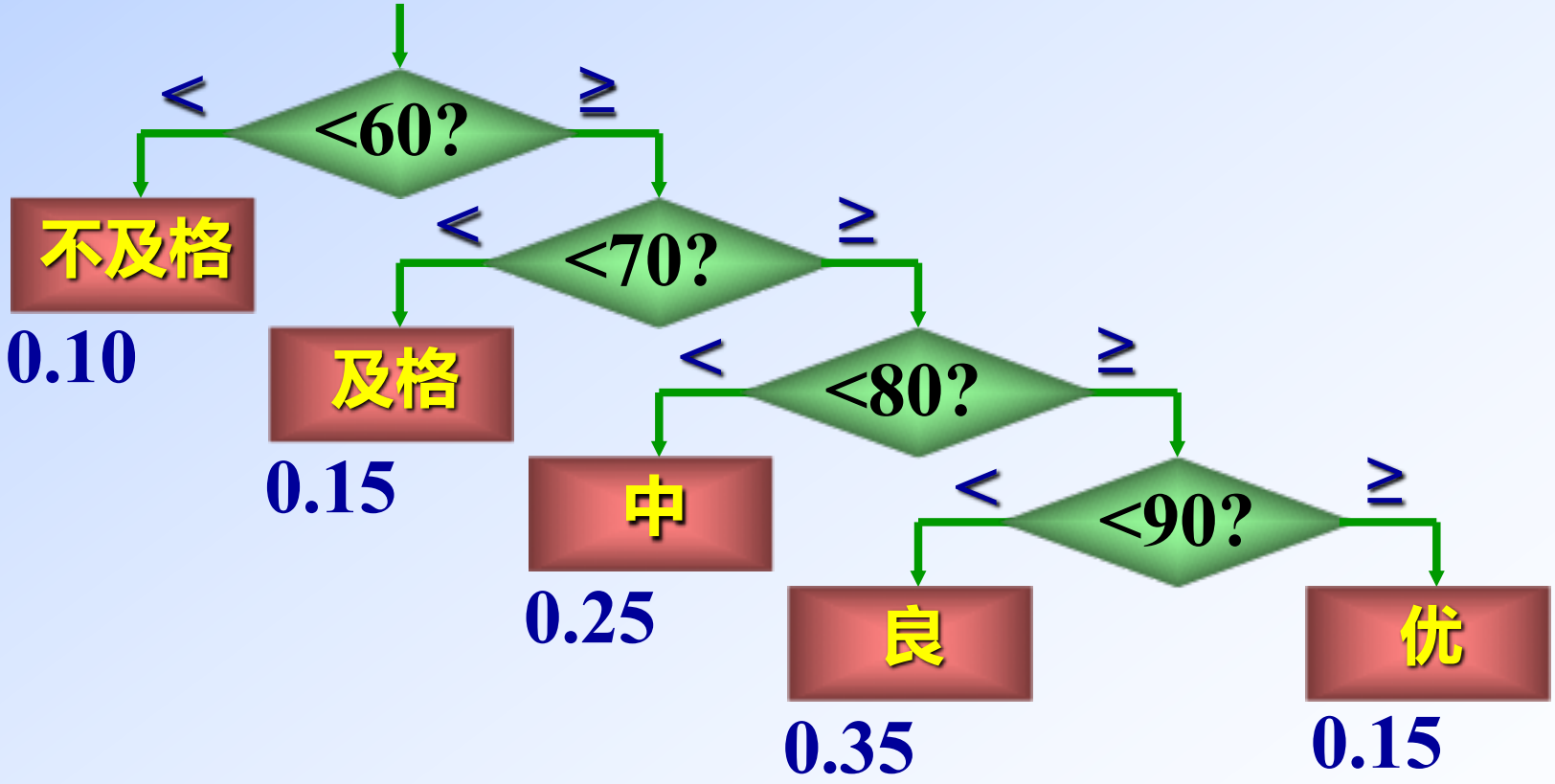
```
            T[i].weight =
                T[pos1].weight + T[pos2].weight;
            T[pos1].parent = T[pos2].parent = i;
        }
    }
```

# 最佳判定树

## 考试成绩分布表

| [0, 60 ) | [60, 70 ) | [70, 80 ) | [80, 90 ) | [90, 100 ) |
|---|---|---|---|---|
| 不及格 | 及格 | 中 | 良 | 优 |
| 0.10 | 0.15 | 0.25 | 0.35 | 0.15 |

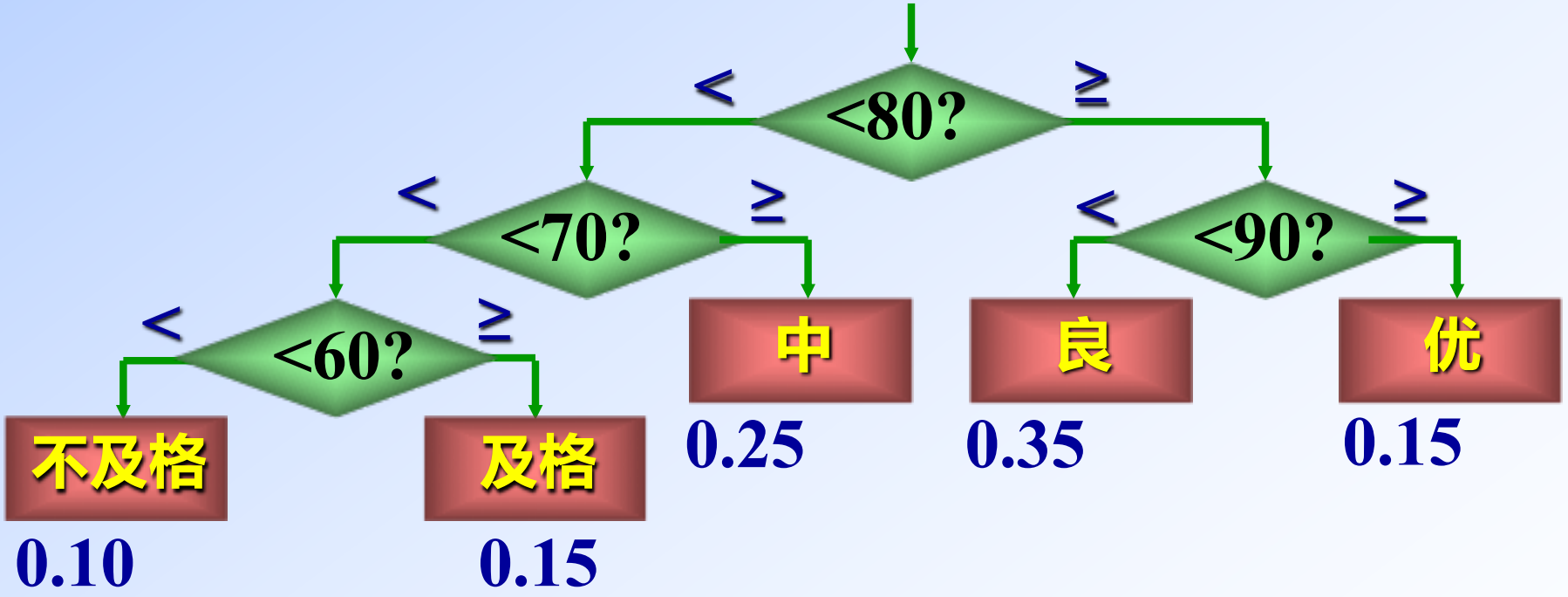
# 判定树




$$WPL = 0.10*1+0.15*2+0.25*3+0.35*4+0.15*4$$

$$= 3.15$$

# 最佳判定树




$$WPL = 0.10*3+0.15*3+0.25*2+0.35*2+0.15*2$$

$$= 0.3+0.45+0.5+0.7+0.3 = 2.25$$

# 霍夫曼编码

主要用途是实现数据压缩。

设给出一段报文：

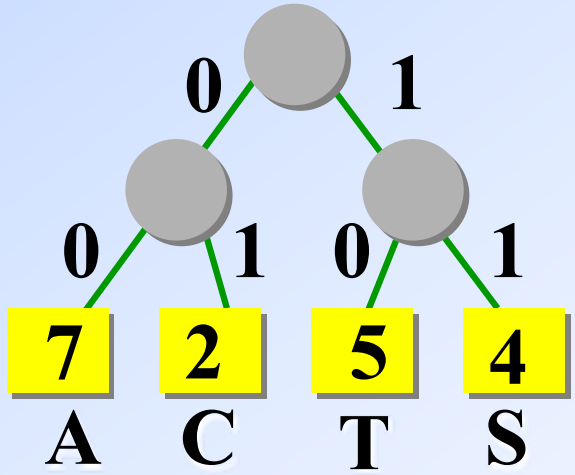
CAST CAST SAT AT A TASA

字符集合是 { C, A, S, T } ，各个字符出现的频度(次数)是 W = { 2, 7, 4, 5 }。

若给每个字符以等长编码

A : 00　T : 10　C : 01　S : 11

则总编码长度为 ( 2+7+4+5 ) * 2 = 36.




若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度。

各字符出现概率为{ C:2/18, A:7/18, S:4/18, T:5/18 },化整为 { 2, 7, 4, 5 }。以它们为各叶结点上的权值, 建立霍夫曼树。左分支赋 0，右分支赋 1，得霍夫曼编码(变长编码)。

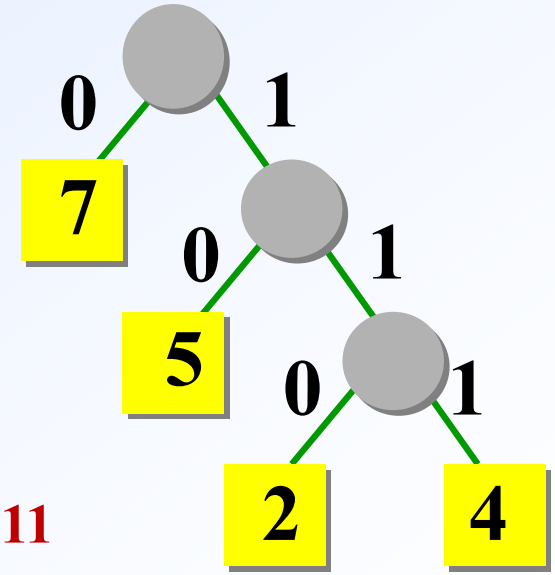
CAST CAST SAT AT A TASA

A : 0　T : 10　C : 110　S : 111

它的总编码长度：7*1+5*2+( 2+4 )*3 = 35。比等长编码的情形要短。

总编码长度正好等于霍夫曼树的带权路径长度WPL。

霍夫曼编码是一种无前缀编码(都由叶结点组成,路径不会重复)。解码时不会混淆。




霍夫曼编码树

霍夫曼编码: A : 0　T : 10　C : 110　S : 111

110011110 1100111101110100100100111O

等长编码: A : 00　T : 10　C : 01　S : 11

0100111001001110110010001000100110O