第六章 树和二叉树

本章主要内容：

- 树的概念和基本术语
- 二叉树
- 二叉树遍历
- 树与森林
- 霍夫曼树

一、树的概念和基本术语

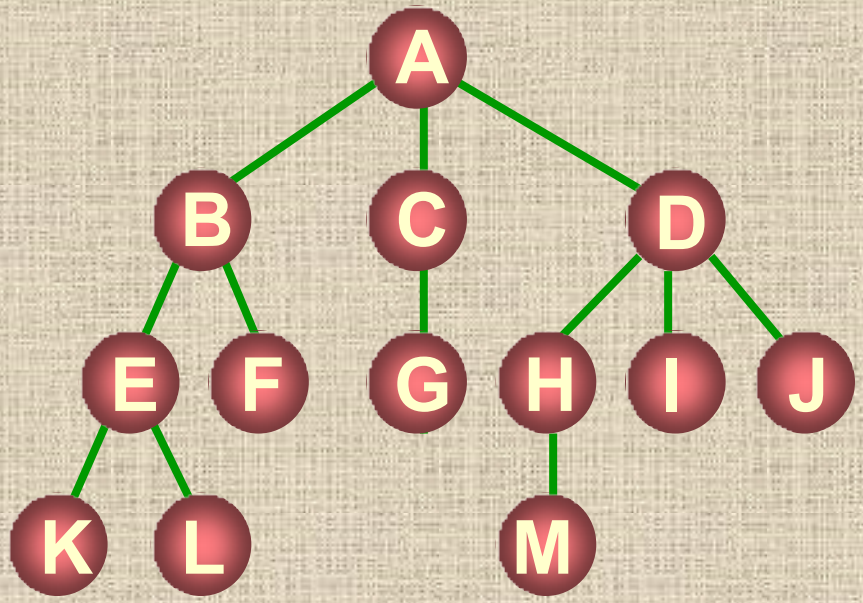
1、树的定义

树是由 n (n ≥ 0) 个结点的有限集合。假如 n = 0，称为空树；假如 n > 0，则

- 有且仅有一种特定的称之为根(Root)的结点，它只有直接后继，但没有直接前驱；
- 当n > 1，除根以外的其他结点划分为 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，而且称为根的子树(SubTree)。

例如

只有根结点的树



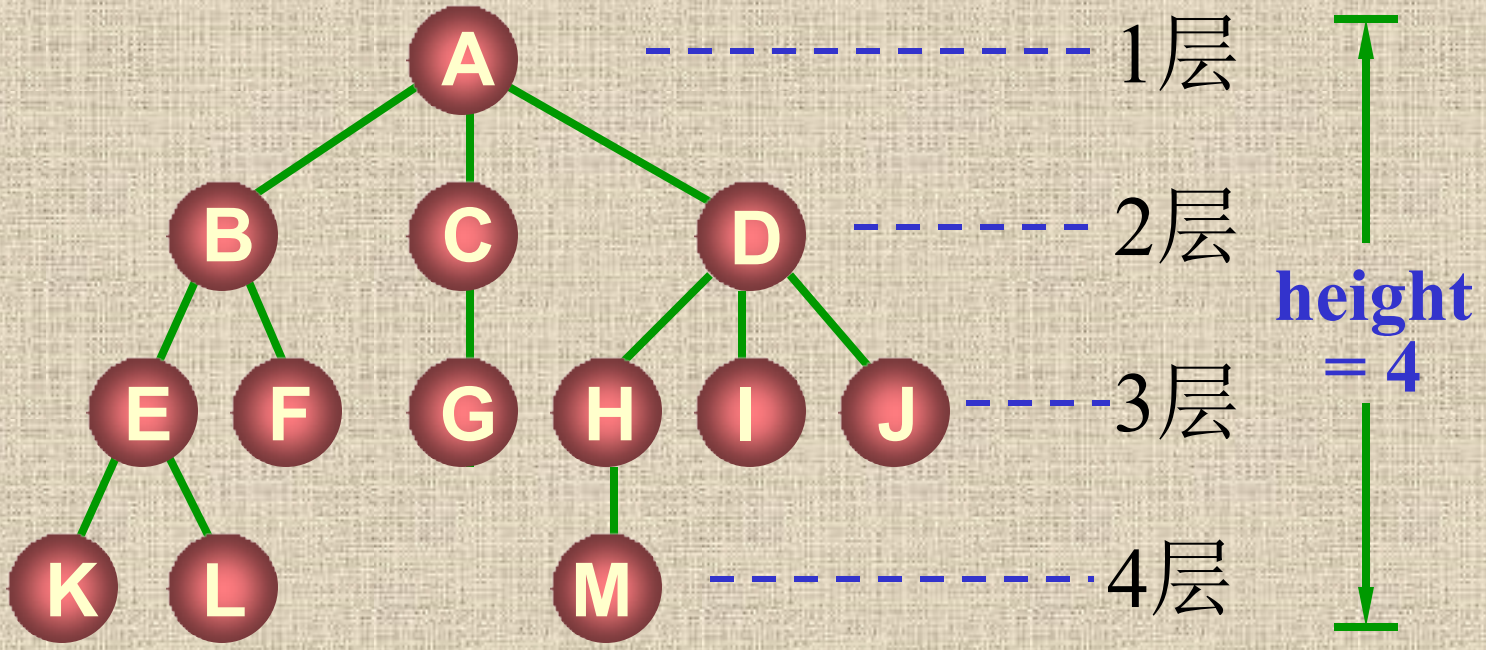
有13个结点的树

其中：A是根；其他结点提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

2、树的基本术语



- 结点
- 结点的度
- 分支结点
- 叶结点
- 树的度
- 树的深度
- 森林
- 子女
- 双亲
- 弟兄
- 祖先
- 子孙
- 结点层次

二、二叉树

1、二叉树的概念与性质

定义：二叉树是一种特殊类型的树，它是由一种根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树构成。

特点：每个结点至多只有两棵子树（二叉树中不存在度不小于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，$2^{i-1}=2^0=1$。

假设对全部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。

因为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, 假如其叶结点数为 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，

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

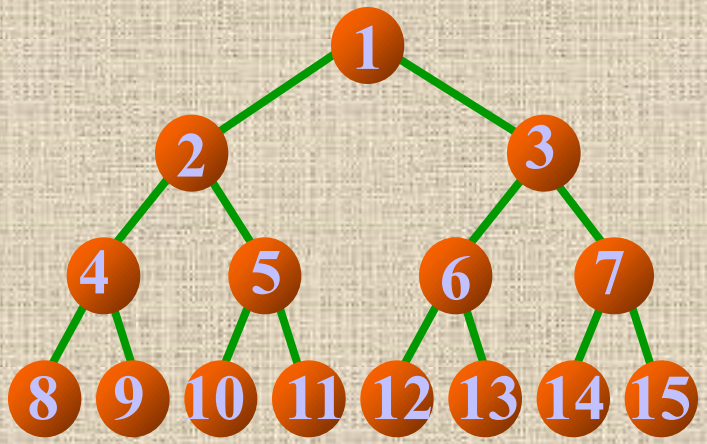
所以，有 $2n_2 + n_1 = n_0 + n_1 + n_2 - 1$

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

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。

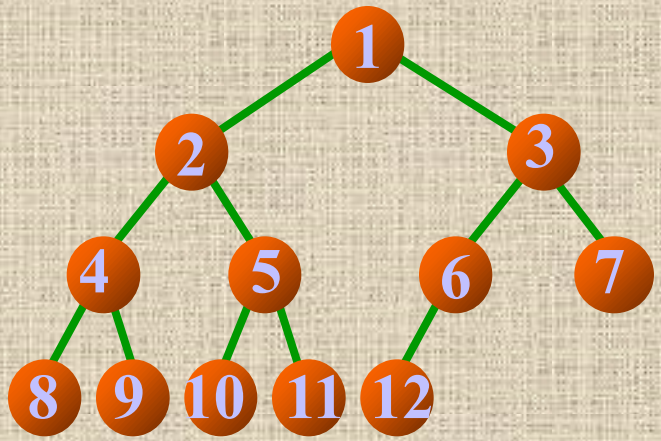


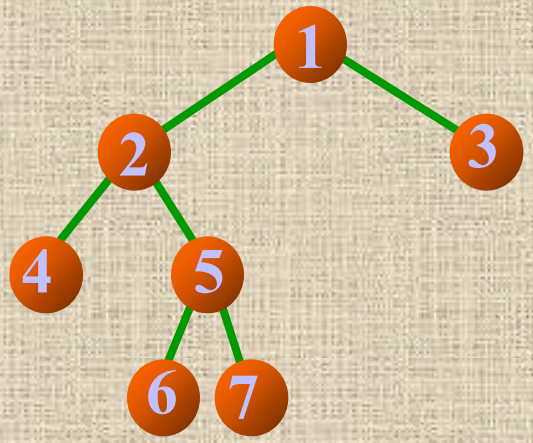
满二叉树

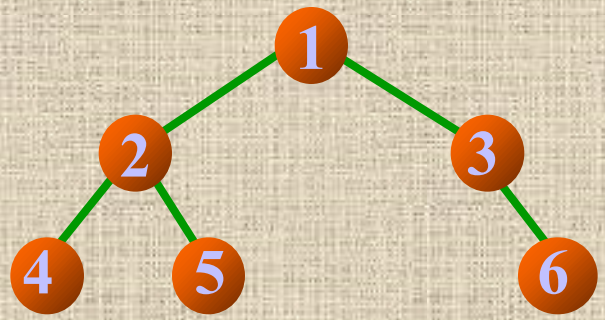
定义2 完全二叉树 (Complete Binary Tree)

若设二叉树的高度为h，则共有h层。除第 h 层外，其他各层 (0 ~ h-1) 的结点数都到达最大个数，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

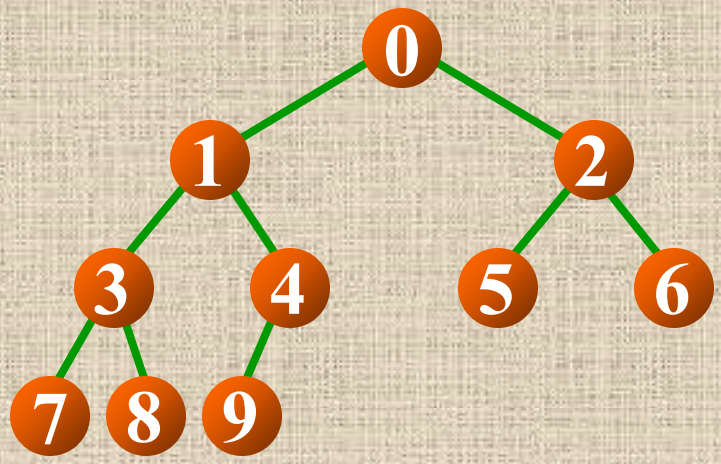
$2^{h-1} - 1 < n \leq 2^h - 1$ 或 $2^{h-1} \leq n < 2^h$

取对数 $h-1 < \log_2 n \leq h$，又h是整数，

所以有 $h = \lfloor \log_2(n) \rfloor + 1$

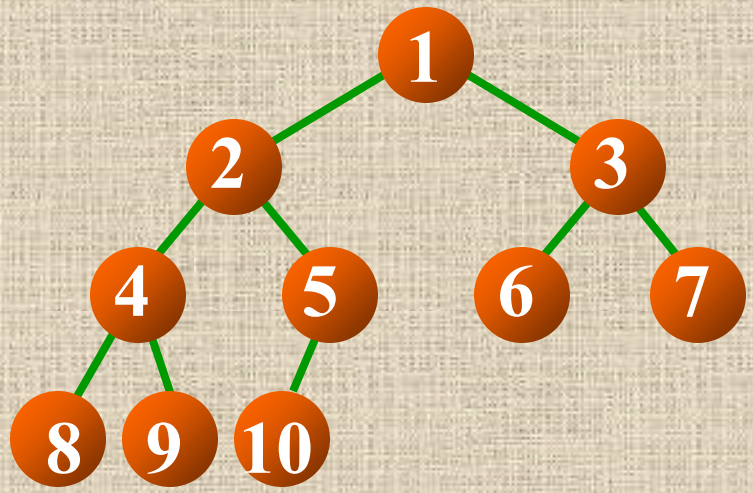
性质5 如将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号0, 1, 2, …, n−1，则有下列关系：

- 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

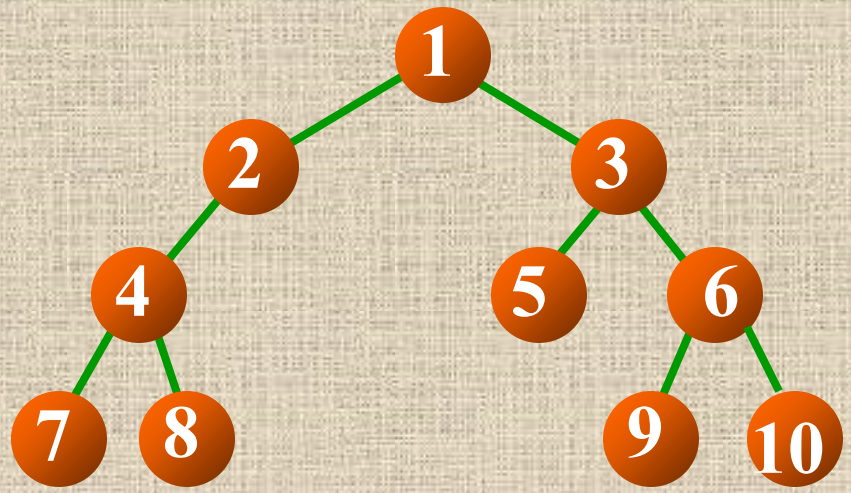


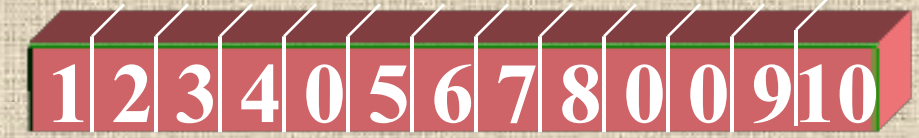
2、二叉树的存储构造

- 顺序表达



完全二叉树
的顺序表达

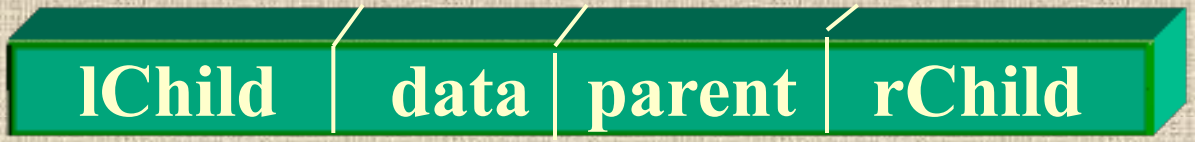




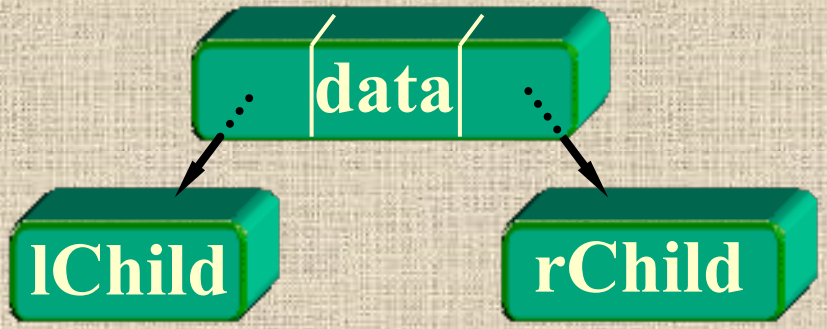
一般二叉树
的顺序表达

- 链表表达

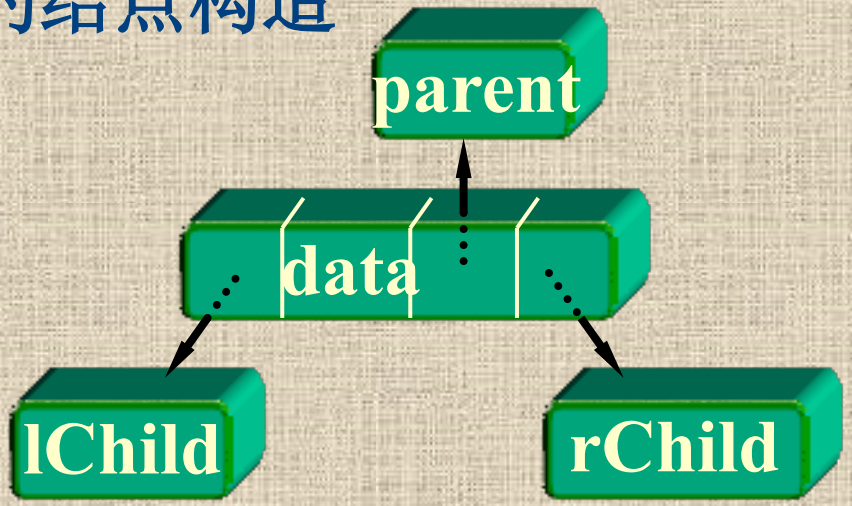
含两个指针域的结点构造



含三个指针域的结点构造

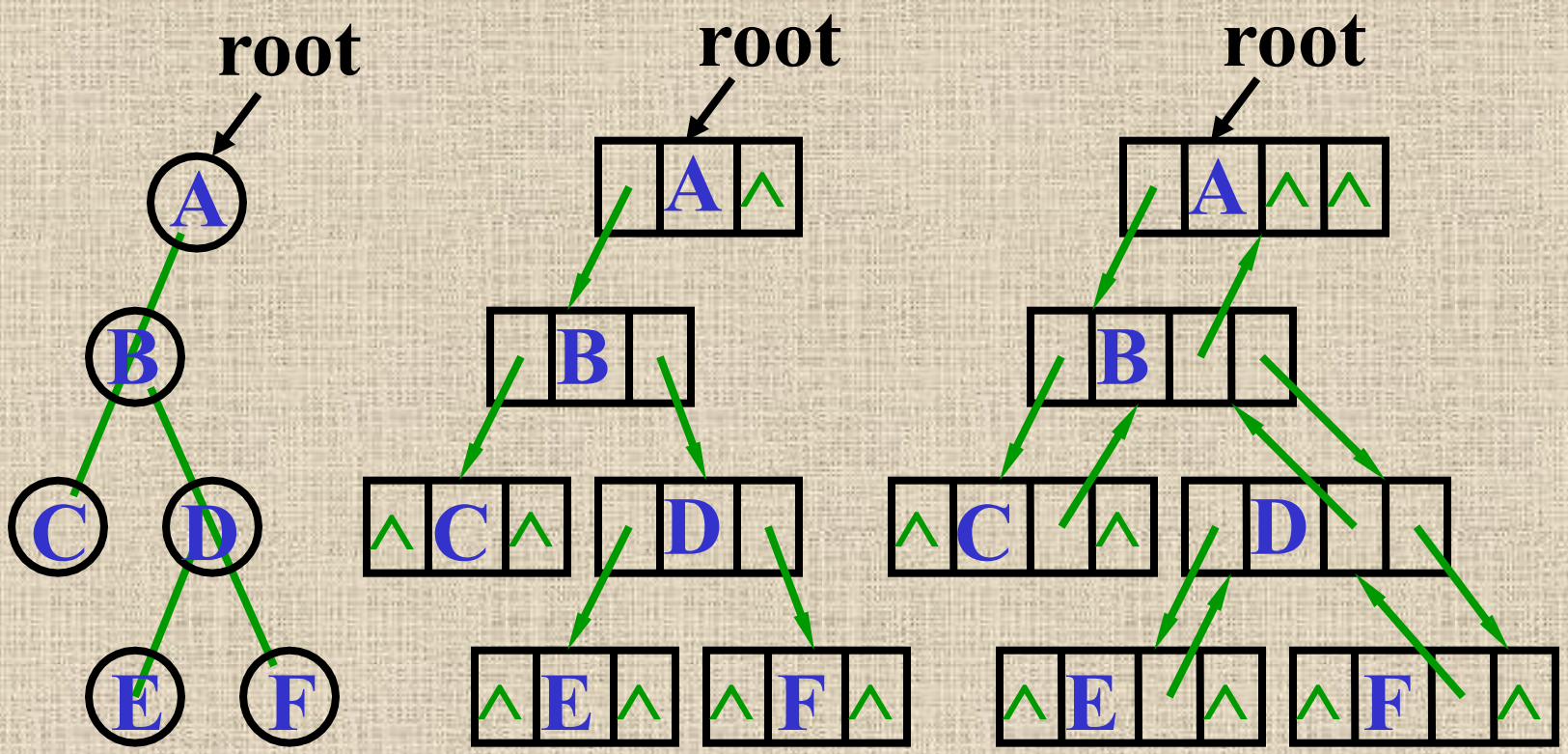


二叉链表



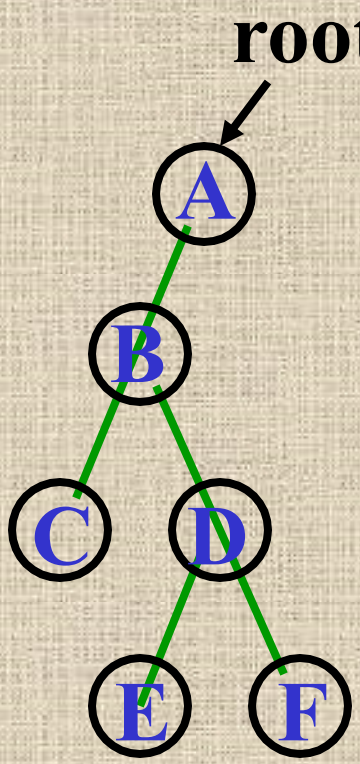
三叉链表

二叉树链表表达的示例



二叉树　　二叉链表　　三叉链表

三叉链表的静态构造



	data	parent	leftChild	rightChi
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 ElemType;//结点数据类型

typedef struct BiTNode {          //结点定义
    ElemType data;
    struct BiTNode * lchild, * rchild;
} BiTNode,*BiTree;
```

3、二叉树遍历

树的遍历就是按某种顺序访问树中的结点，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

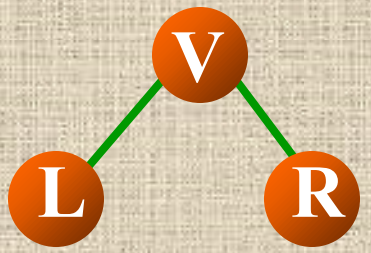
遍历根的右子树记作 R

则可能的遍历顺序有

先序　VLR

中序　LVR

后序　LRV



■中序遍历

中序遍历二叉树算法的定义：

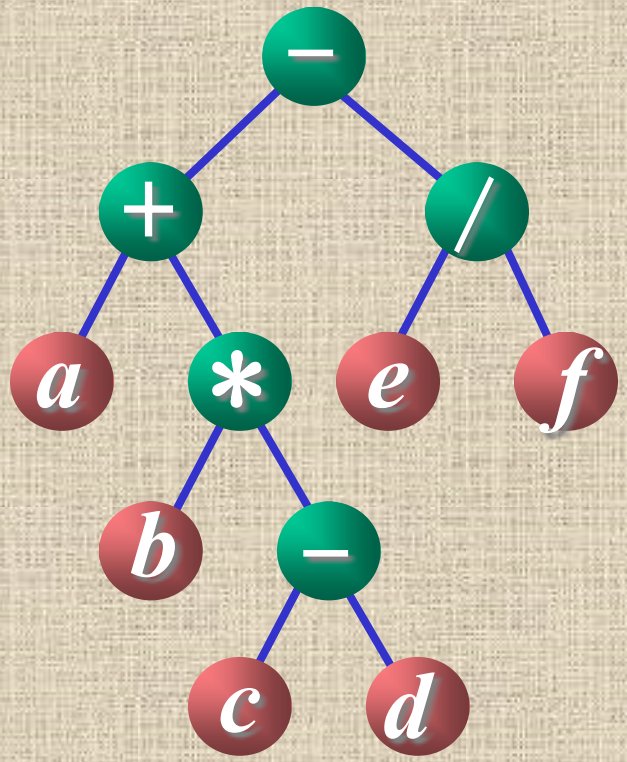
若二叉树为空，则空操作；

不然

中序遍历左子树 (L)；

访问根结点 (V)；

中序遍历右子树 (R)。



遍历成果

a + b * c − d − e / f

中序遍历二叉树的递归算法演示

```
void InOrderTraverse ( BiTree T,
                       int (*visit(ElemType data ) {
    if (T) {
        InOrderTraverse ( T->lchild );
        visit(T->data);
        InOrderTraverse ( T->rchild );
    }
}
```

■ 先序遍历 (Preorder Traversal)

前序遍历二叉树算法的定义：

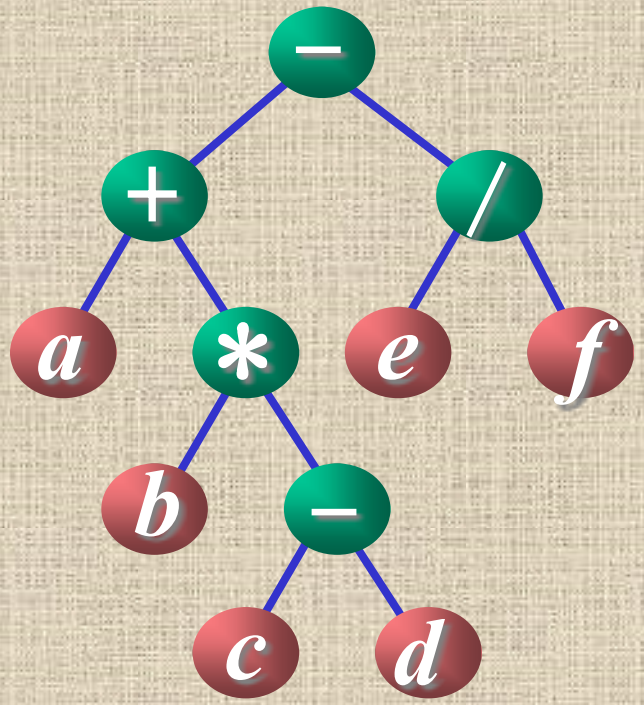
若二叉树为空，则空操作；

不然

访问根结点 (V)；

前序遍历左子树 (L)；

前序遍历右子树 (R)。



遍历成果

− + a * b − c d / e f

前序遍历二叉树的递归算法演示

```
void PreOrderTraverss ( BiTree T,
                int(*visit )(ElemType data)) {
   if ( T ) {
      visit( T->data);
      PreOrderTraverss ( T->lchild );
      PreOrderTraverss ( T->rchild );
   }
}
```

■后序遍历 (Postorder Traversal)

后序遍历二叉树算法的定义：

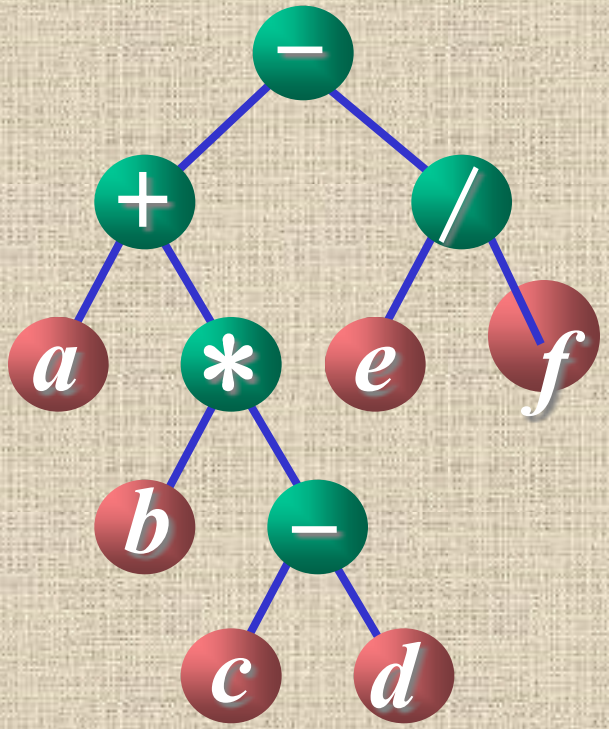
若二叉树为空，则空操作；

不然

后序遍历左子树 (L)；

后序遍历右子树 (R)；

访问根结点 (V)。



遍历成果

a b c d - * + e f / -

后序遍历二叉树的递归算法 演示

```
void PostOrderTraverss ( BiTree T,
                         int (*visit)(ElemType data) ) {
    if ( T ) {
        PostOrderTraverss ( T->lchild );
        PostOrderTraverss ( T->rchild );
        visit( T->data);
    }
}
```

4、二叉树主要操作

- 按前序建立二叉树演示

```
void CreateBiTree(BiTRee &T){
    scan(&ch);
    if(ch==“”)  T=null;
    else{
        if(!(T=(BiTree)malloc(sizeof(BiTNode))))
            exit (overflow);
        T->data=ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
  return;
}
```

- 计算二叉树结点个数(递归算法)

```
int CountBiTNods ( BinTreeNode *T ) {
    if ( !T ) return 0;
    else
        return 1 + CountBiTNods ( T->lchild
)
                 + CountBiTNods ( T->rchild
);
}
```

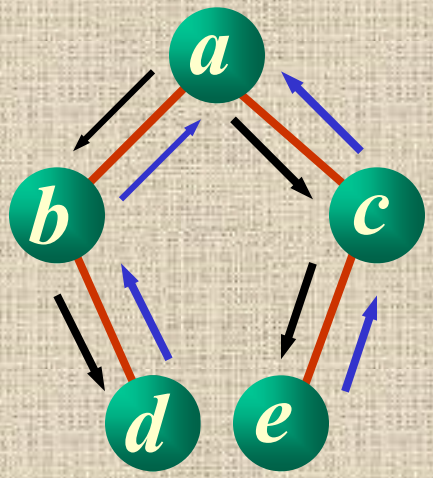
◆求二叉树中叶子结点的个数

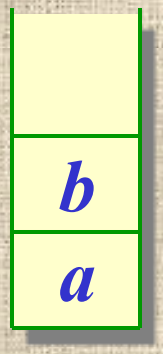
```
int CountLeafs (Bitree T) {
//求二叉树中叶子结点的数目
    if(!T) return 0; //空树没有叶子
    else
      if(!T->lchild&&!T->rchild) return 1;
                                  //叶子结点
      else
        return CountLeafs (T->lchild)
                +CountLeafs(T->rchild);
                 //左子树的叶子数加上右子树的叶子数
}//Leaf_Count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

◆求二叉树高度(递归算法)

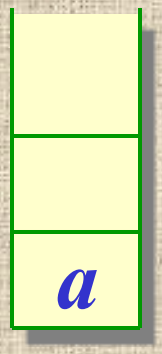
```
int CountHeight ( BiTree T ) {
    if ( !T) return 0;
    else {
        int m =CountHeight ( T->lchild );
        int n =CountHeight ( T->rchild ) );
        return (m > n) ? m+1 : n+1;
}
```

◆中序遍历二叉树(非递归算法)用栈实现

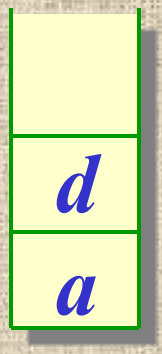




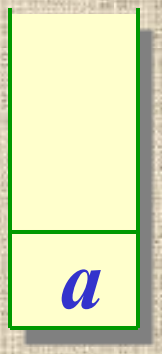
a b入栈



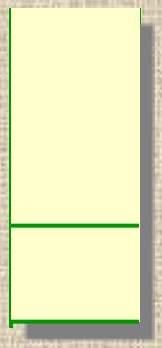
b退栈
访问



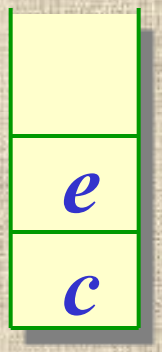
*d*入栈



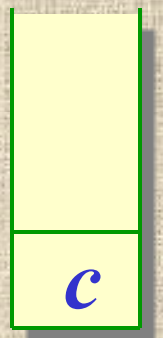
*d*退栈
访问



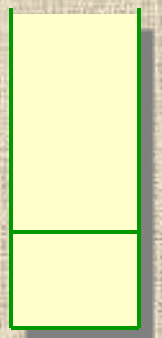
*a*退栈
访问



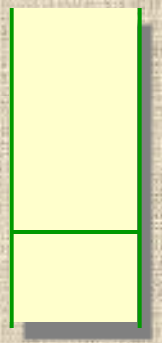
c e 入栈



e 退栈
访问



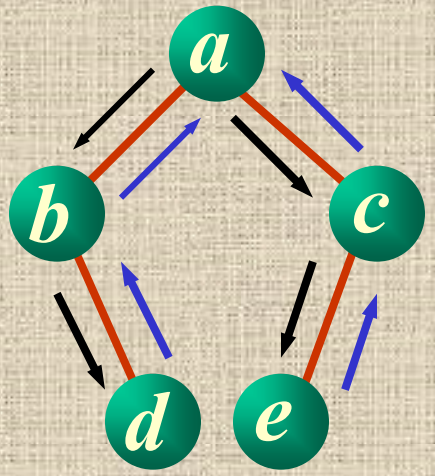
c 退栈
访问



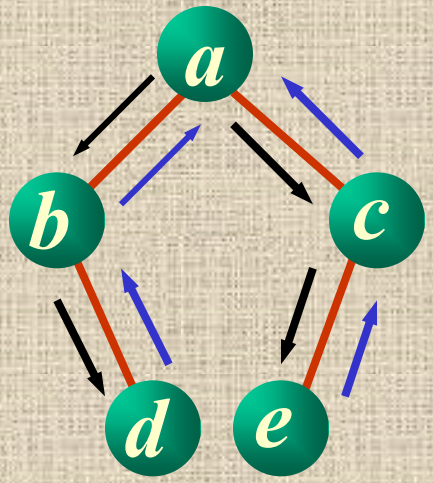
栈空

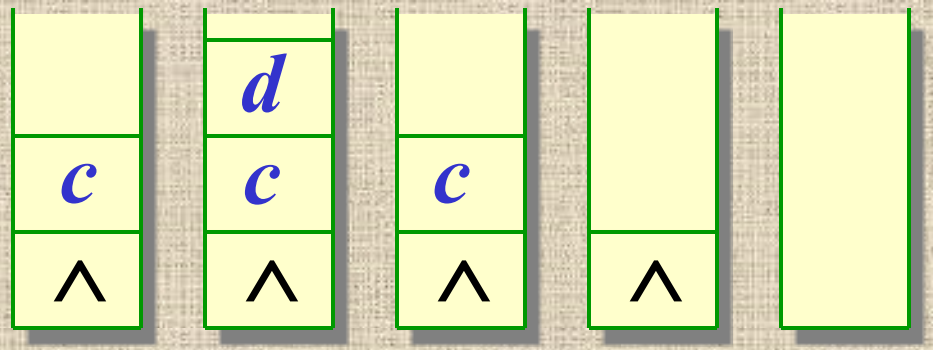
```
void InOrderTraverss ( BiTree T ) {
   stack S;   InitStack( &S );    //递归工作栈
   Push(S,T);                     //初始化
 while ( !StackEmpty(S) ) {
    while(GetTop(S,p)&&p)
         Push(S,p->lchild);
     Pop(S,p);
     if ( !StackEmpty(S) ) {          //栈非空
          Pop(S, p);                   //退栈
          if(!visit(p->data)) return error;
          Push(S, p->rchild);
       }//if
   }//while
 return ;
}
```

演示



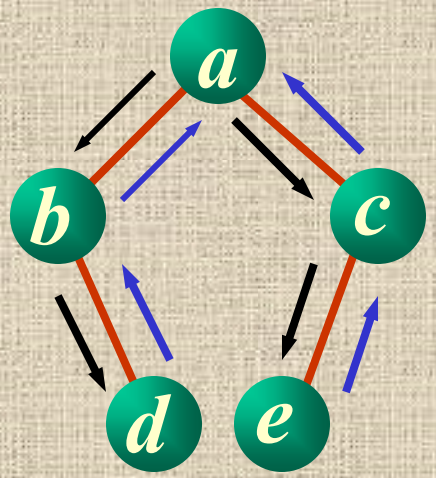
◆前序遍历二叉树(非递归算法)用栈实现





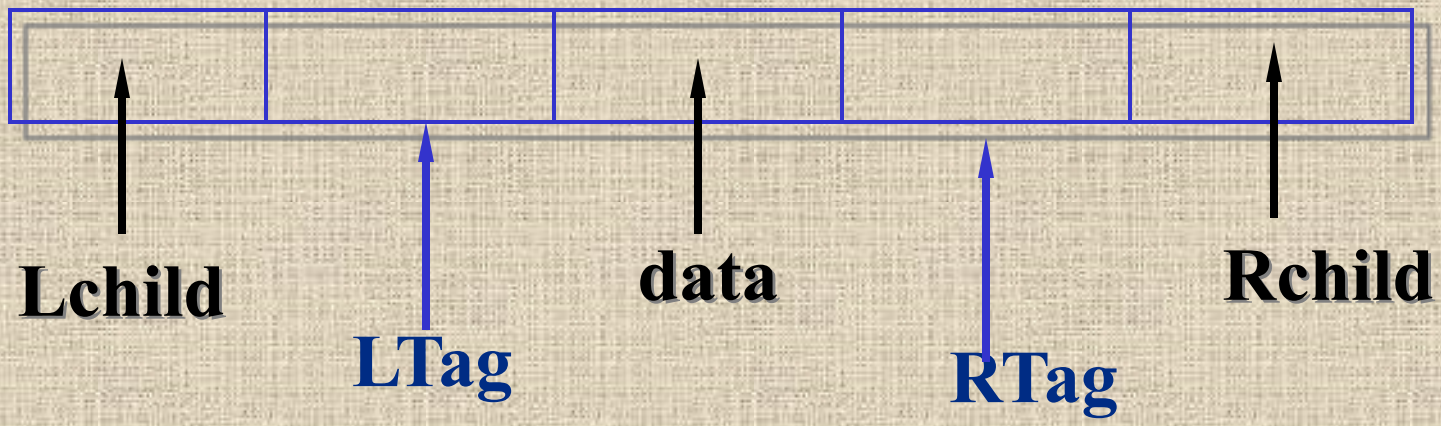
访问	访问	退栈	退栈	访问	
a	*b*	*d*	*c*	*e*	
进栈	进栈	访问	访问	左进	
c	*d*	*d*	*c*	*空*	
左进	左进	左进	左进	退栈	
b	*空*	*空*	*e*	^	结束

```
void PreOrderTraverss( BiTree T ) {
    stack S;   InitStack(S); //递归工作栈
    BiTree p = T;  Push (S, NULL);
    while ( p ) {
        visit( p->data);
        if ( p->rchild )
            Push ( S, p->rchild );
        if  ( p->lchild )
            p = p->lchild;//进左子树
        else Pop( S, p );
    }
}
```

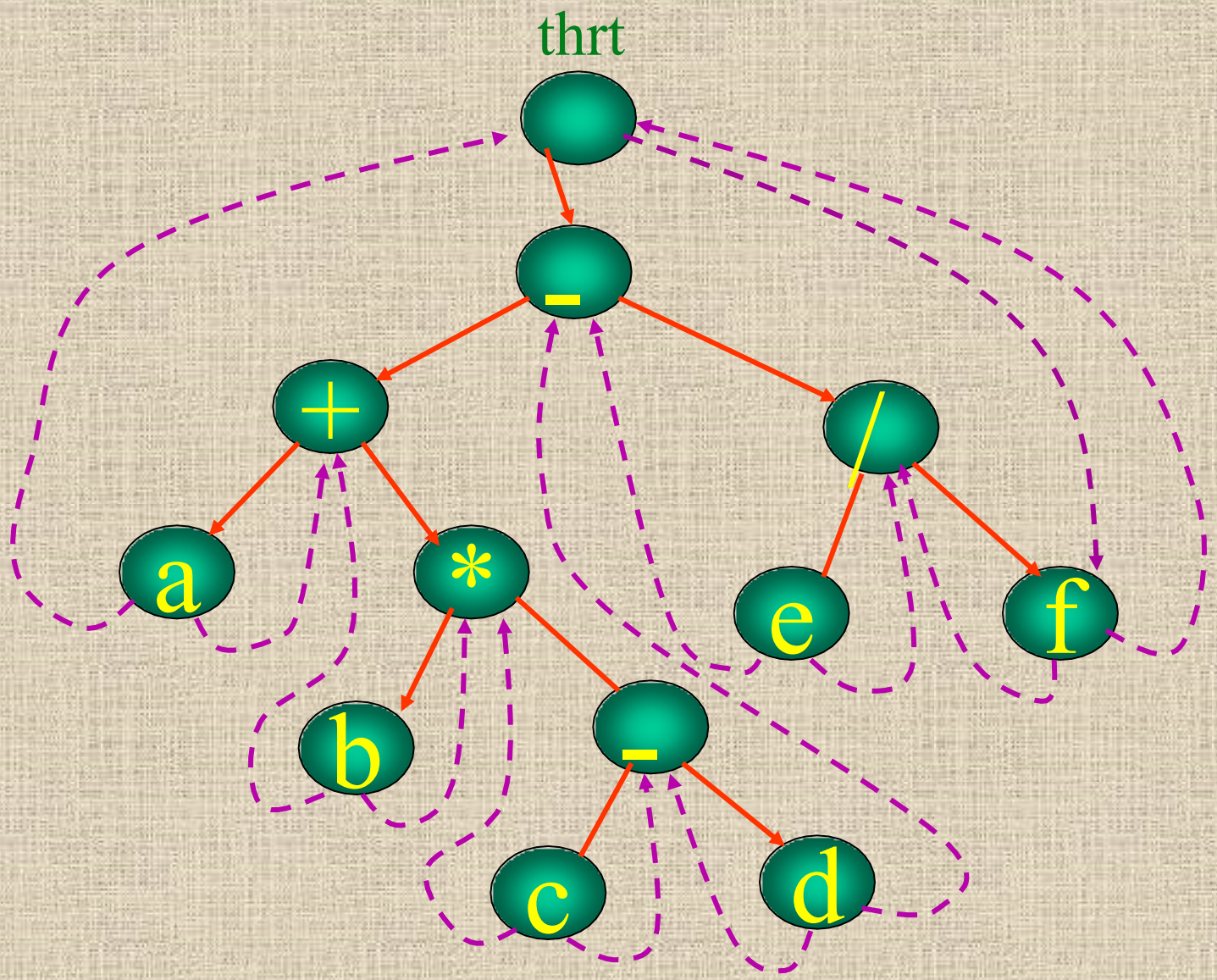


5、线索二叉树（Threaded Binary Tree）

■结点构造



LTag=0, Lchild为左孩子
LTag=1, Lchild为前驱线索
RTag=0, Rchild为右孩子
RTag=1, Rchild为后继指针



中序线索二叉树

■线索二叉树存储构造

```
typedef enum PointerTag{Link=0,Thread=1}
    //Link==0：指针，Thread==1：线索;
typedef struct BiThrNode{
   ElemType data;
    struct BithrNode *lchild,*rchild;
    PointerTag LTag,RTag;  //左右标志
}BiThrNode,*BiThrTree;
```