

第六章

树和二叉树

6.1 树的类型定义

6.2 二叉树的类型定义

6.3 二叉树的存储结构

6.4 二叉树的遍历

6.5 线索二叉树

6.6 树和森林的表示方法

6.7 树和森林的遍历

6.8 哈夫曼树与哈夫曼编码

6.1

树的类型定义



数据对象 D:

D是具有相同特性的数据元素的集合。

数据关系 R:

若D为空集，则称为空树。

否则:

- (1) 在D中存在唯一的称为根的数据元素root;
- (2) 当 $n > 1$ 时，其余结点可分为 m ($m > 0$)个互不相交的有限集 T_1, T_2, \dots, T_m ，其中每一棵子集本身又是一棵符合本定义棵树，称为根root的子树。





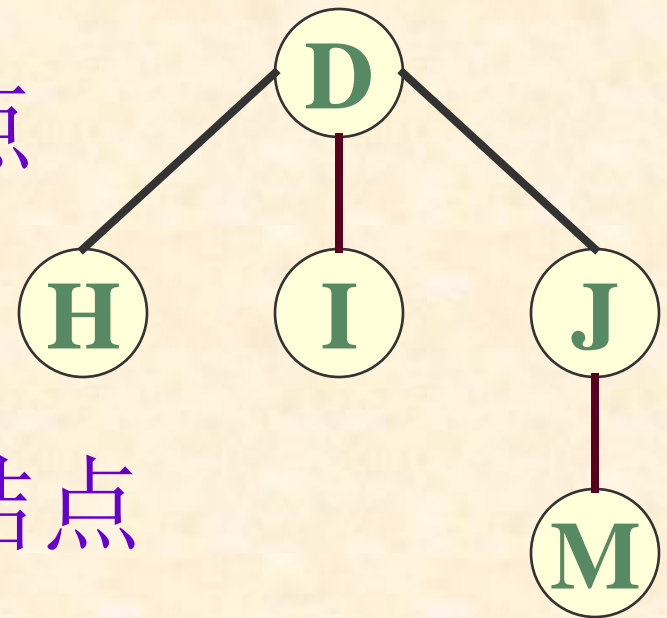
基 本 术 语

结点： 数据元素+若干指向子树的分支

结点的度： 分支的个数

树的度： 树中所有结点的度的最大值

叶子结点： 度为零的结点

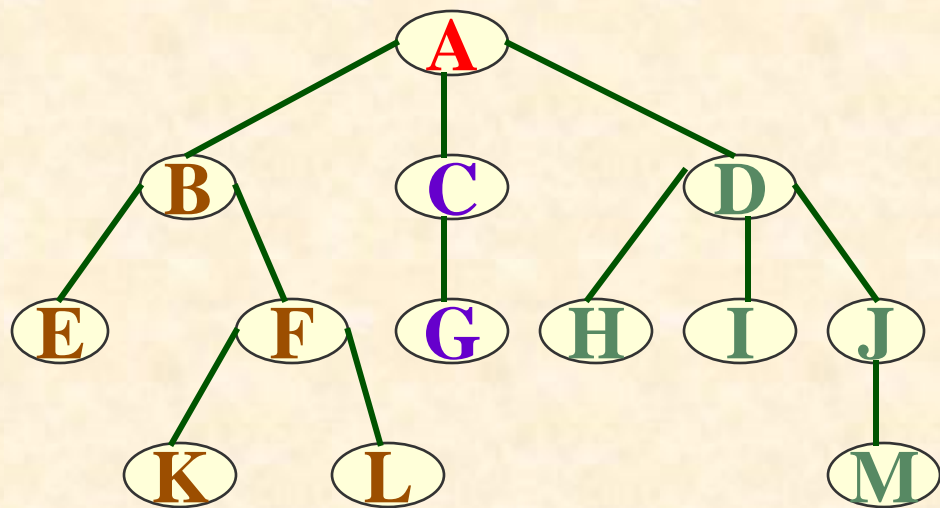


分支结点： 度大于零的结点

(从根到结点的)**路径**:

由从**根**到该结点
所经分支和结点构成

孩子结点、**双亲**结点
兄弟结点、**堂兄弟**
祖先结点、**子孙**结点

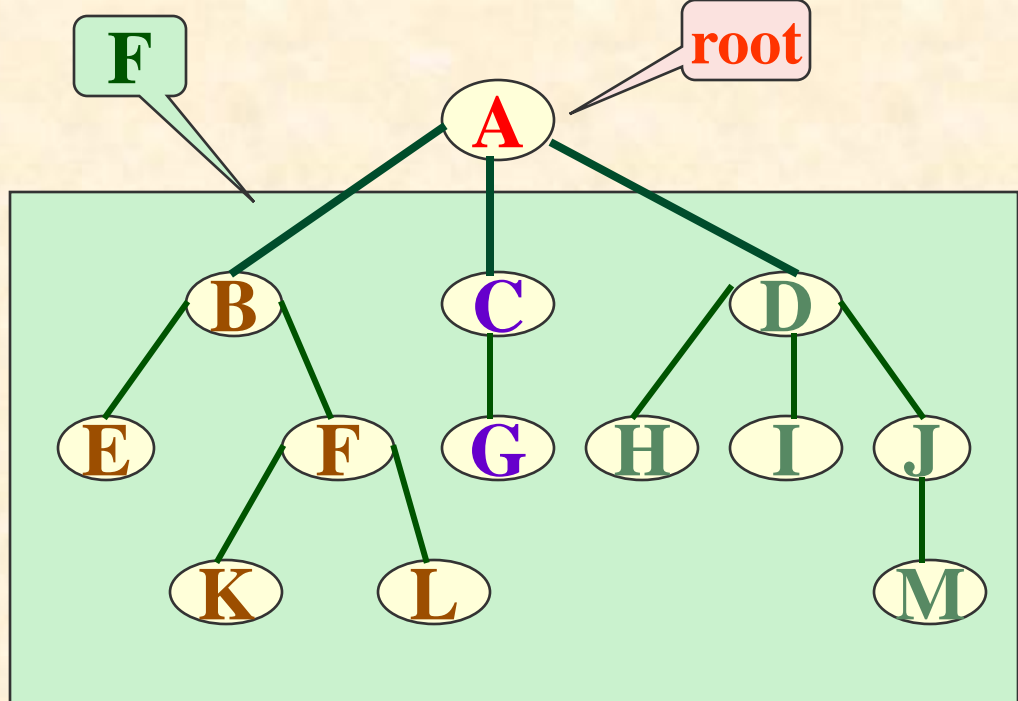


结点的层次: 假设根结点的层次为1, 第 l
层的结点的子树根结点的层
次为 $l+1$

树的深度: 树中叶子结点所在的最大层次

森林:

是 m ($m \geq 0$) 棵互不相交的树的集合



任何一棵非空树是一个二元组

$$\text{Tree} = (\text{root}, F)$$

其中: root 被称为根结点

F 被称为子树森林



基本操作：

✦ 查 找 类

✦ 插 入 类

✦ 删 除 类



查找类:

Root(T) // 求树的根结点

Value(T, cur_e) // 求当前结点的元素值

Parent(T, cur_e) // 求当前结点的双亲结点

LeftChild(T, cur_e) // 求当前结点的最左孩子

RightSibling(T, cur_e) // 求当前结点的右兄弟

TreeEmpty(T) // 判定树是否为空树

TreeDepth(T) // 求树的深度

TraverseTree(T, Visit()) // 遍历



插入类:

InitTree(&T) // 初始化置空树

CreateTree(&T, definition)

// 按定义构造树

Assign(T, cur_e, value)

// 给当前结点赋值

InsertChild(&T, &p, i, c)

// 将以c为根的树插入为结点p的第i棵子树



删除类:

ClearTree(&T) // 将树清空

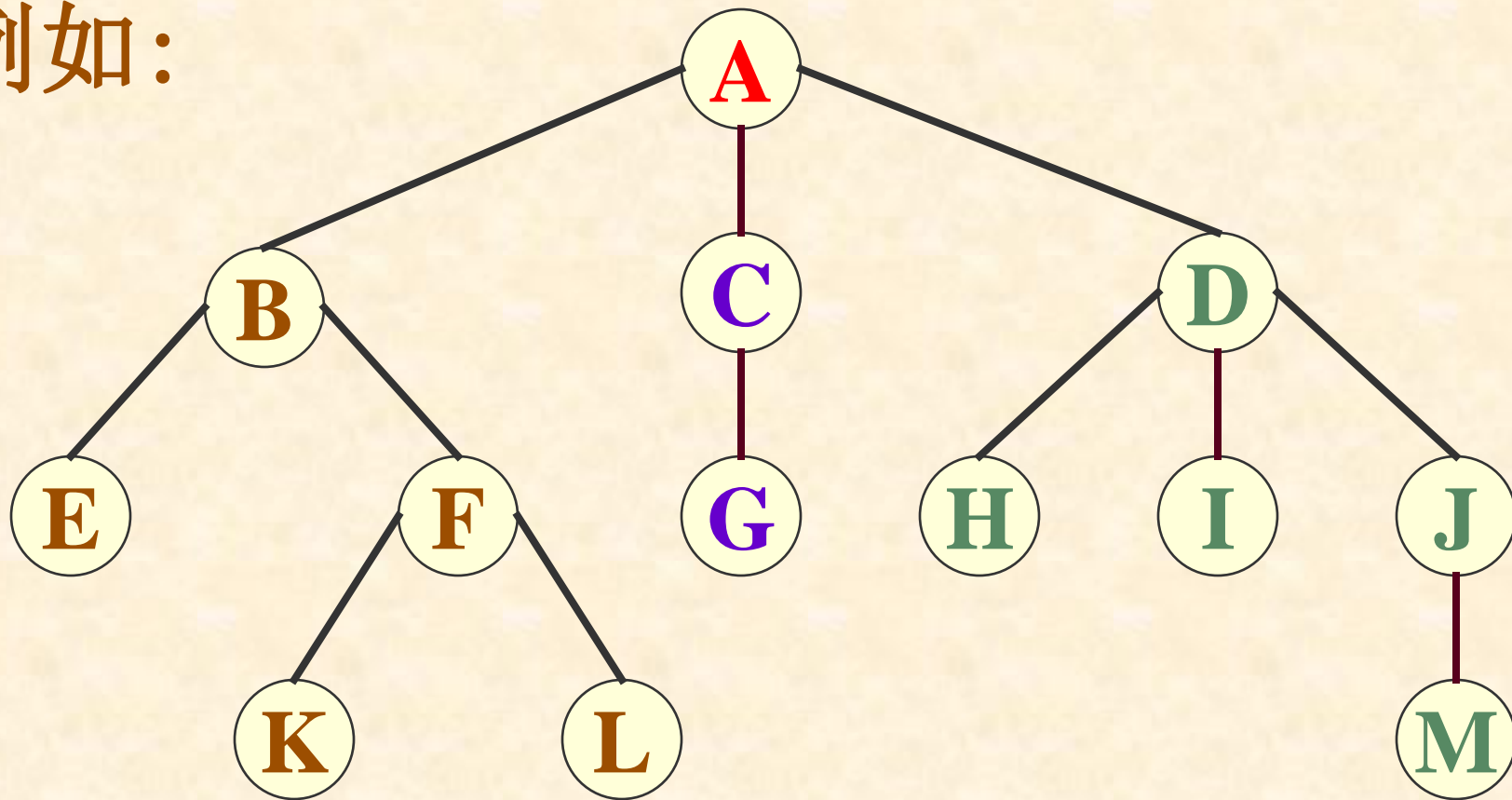
DestroyTree(&T) // 销毁树的结构

DeleteChild(&T, &p, i)

// 删除结点p的第i棵子树



例如：



A(**B**(**E**, **F**(**K**, **L**)), **C**(**G**), **D**(**H**, **I**, **J**(**M**)))

树根 T_1 T_2 T_3

有向树:

- (1) 有确定的根;
- (2) 树根和子树根之间为有向关系。

有序树:



子树之间存在确定的次序关系。

无序树:

子树之间不存在确定的次序关系。



对比树型结构和线性结构 的结构特点



线性结构

第一个数据元素
(无前驱)

最后一个数据元素
(无后继)

其它数据元素
(一个前驱、
一个后继)

树型结构

根结点
(无前驱)

多个叶子结点
(无后继)

其它数据元素
(一个前驱、
多个后继)



6.2

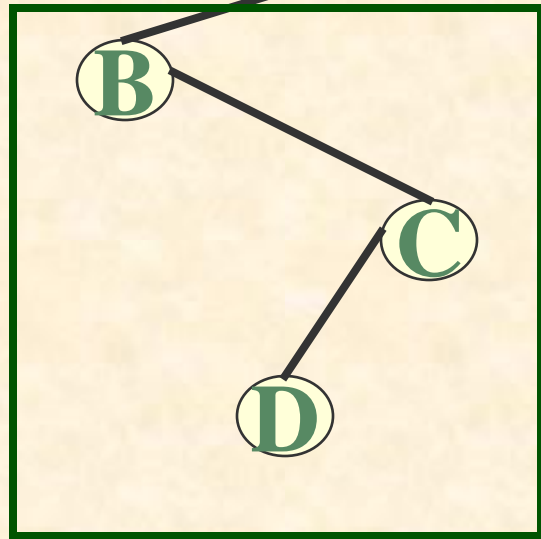
二叉树的类型定义



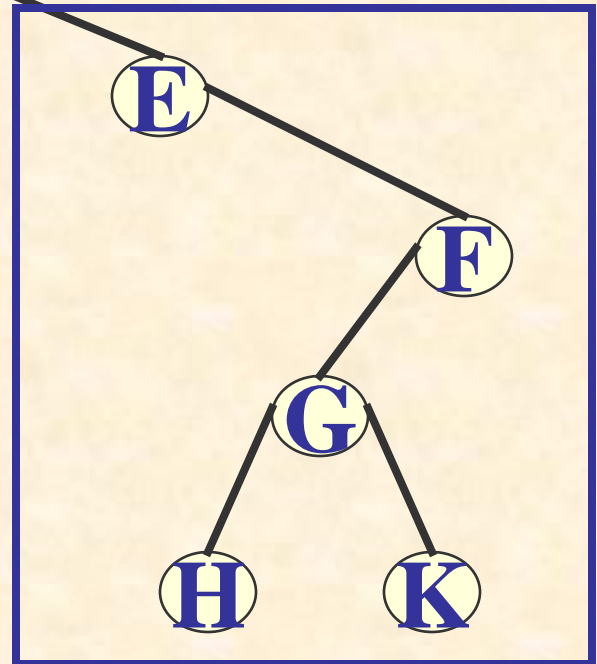
二叉树或为空树，或是由一个根结点加上两棵分别称为**左子树**和**右子树**的、**互不交的**二叉树组成。

右子树

根结点



左子树



二叉树的五种基本形态:

空树

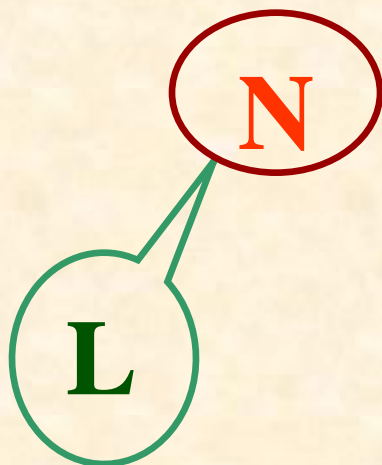


只含根结点

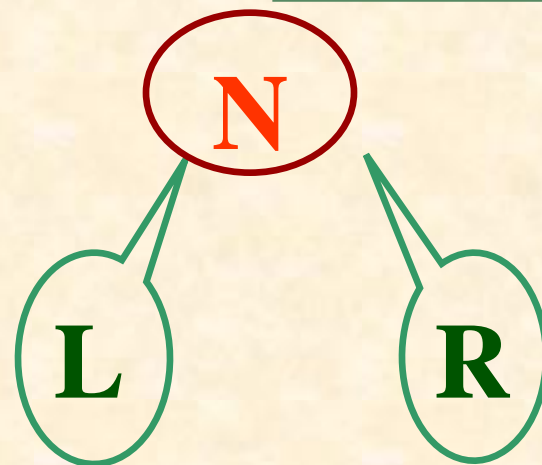
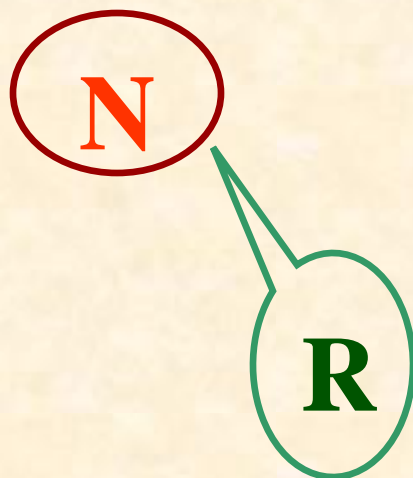


左右子
树均不
为空树

右子树为空树



左子树为空树



二叉树的主要基本操作：



查找类



插入类



删除类



Root(T); Value(T, e); Parent(T, e);

LeftChild(T, e); RightChild(T, e);

LeftSibling(T, e); RightSibling(T, e);

BiTreeEmpty(T); BiTreeDepth(T);

PreOrderTraverse(T, Visit());

InOrderTraverse(T, Visit());

PostOrderTraverse(T, Visit());

LevelOrderTraverse(T, Visit());



InitBiTree(&T);

Assign(T, &e, value);

CreateBiTree(&T, definition);

InsertChild(T, p, LR, c);



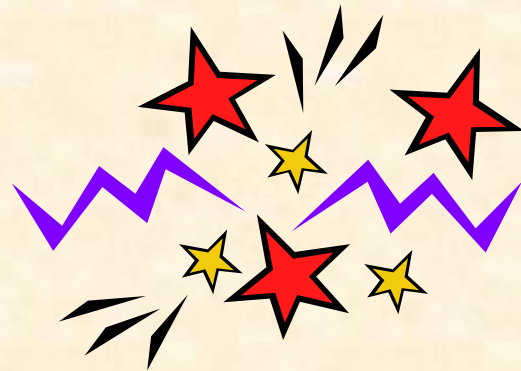
ClearBiTree(&T);

DestroyBiTree(&T);

DeleteChild(T, p, LR);



二叉树 的重要特性



§ 性质 1 :

在二叉树的第 i 层上至多有 2^{i-1} 个结点。
($i \geq 1$)

用归纳法证明:

归纳基: $i = 1$ 层时, 只有一个根结点:

$$2^{i-1} = 2^0 = 1;$$

归纳假设: 假设对所有的 j , $1 \leq j < i$, 命题成立;

归纳证明: 二叉树上每个结点至多有两棵子树,
则第 i 层的结点数 $= 2^{i-2} \times 2 = 2^{i-1}$ 。

§ 性质 2 :

深度为 k 的二叉树上至多含 2^k-1 个结点 ($k \geq 1$)。

证明:

基于上一条性质，深度为 k 的二叉树上的结点数至多为

$$2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1。$$

§ 性质 3 :

对任何一棵二叉树, 若它含有 n_0 个叶子结点、 n_2 个度为 2 的结点, 则必存在关系式: $n_0 = n_2 + 1$ 。

证明:

设 二叉树上结点总数 $n = n_0 + n_1 + n_2$

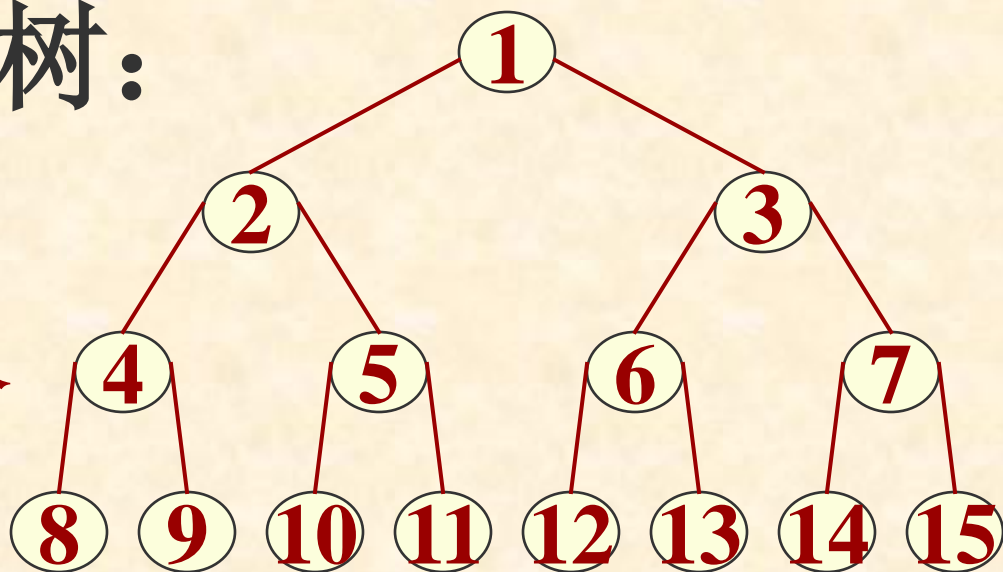
又 二叉树上分支总数 $b = n_1 + 2n_2$

而 $b = n - 1 = n_0 + n_1 + n_2 - 1$

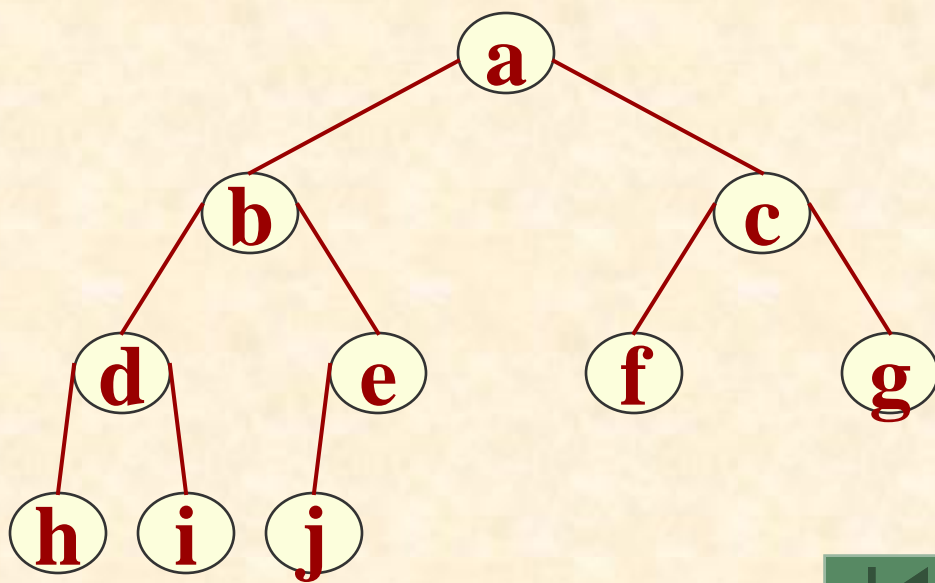
由此, $n_0 = n_2 + 1$ 。

两类特殊的二叉树：

满二叉树： 指的是深度为 k 且含有 2^k-1 个结点的二叉树。



完全二叉树： 树中所含的 n 个结点和满二叉树中编号为 1 至 n 的结点一一对应。



§ 性质 4 :

具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ 。

证明:

设完全二叉树的深度为 k

则根据第二条性质得 $2^{k-1} \leq n < 2^k$

即 $k-1 \leq \log_2 n < k$

因为 k 只能是整数, 因此, $k = \lfloor \log_2 n \rfloor + 1$ 。

§ 性质 5 :

若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号, 则对完全二叉树中任意一个编号为 i 的结点:

(1) 若 $i=1$, 则该结点是二叉树的根, 无双亲, 否则, 编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点;

(2) 若 $2i > n$, 则该结点无左孩子, 否则, 编号为 $2i$ 的结点为其左孩子结点;

(3) 若 $2i+1 > n$, 则该结点无右孩子结点, 否则, 编号为 $2i+1$ 的结点为其右孩子结点。

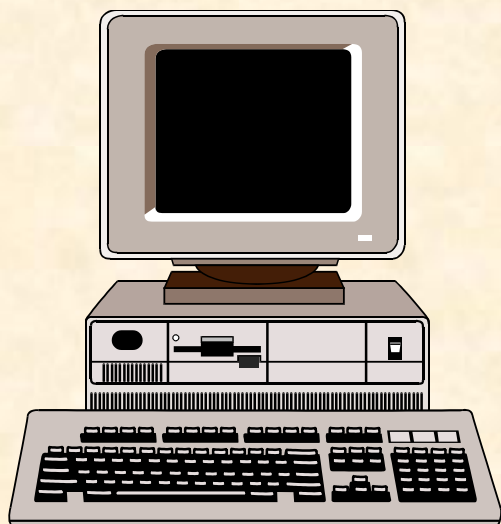


6.3

二叉树的存储结构

一、二叉树的顺序
存储表示

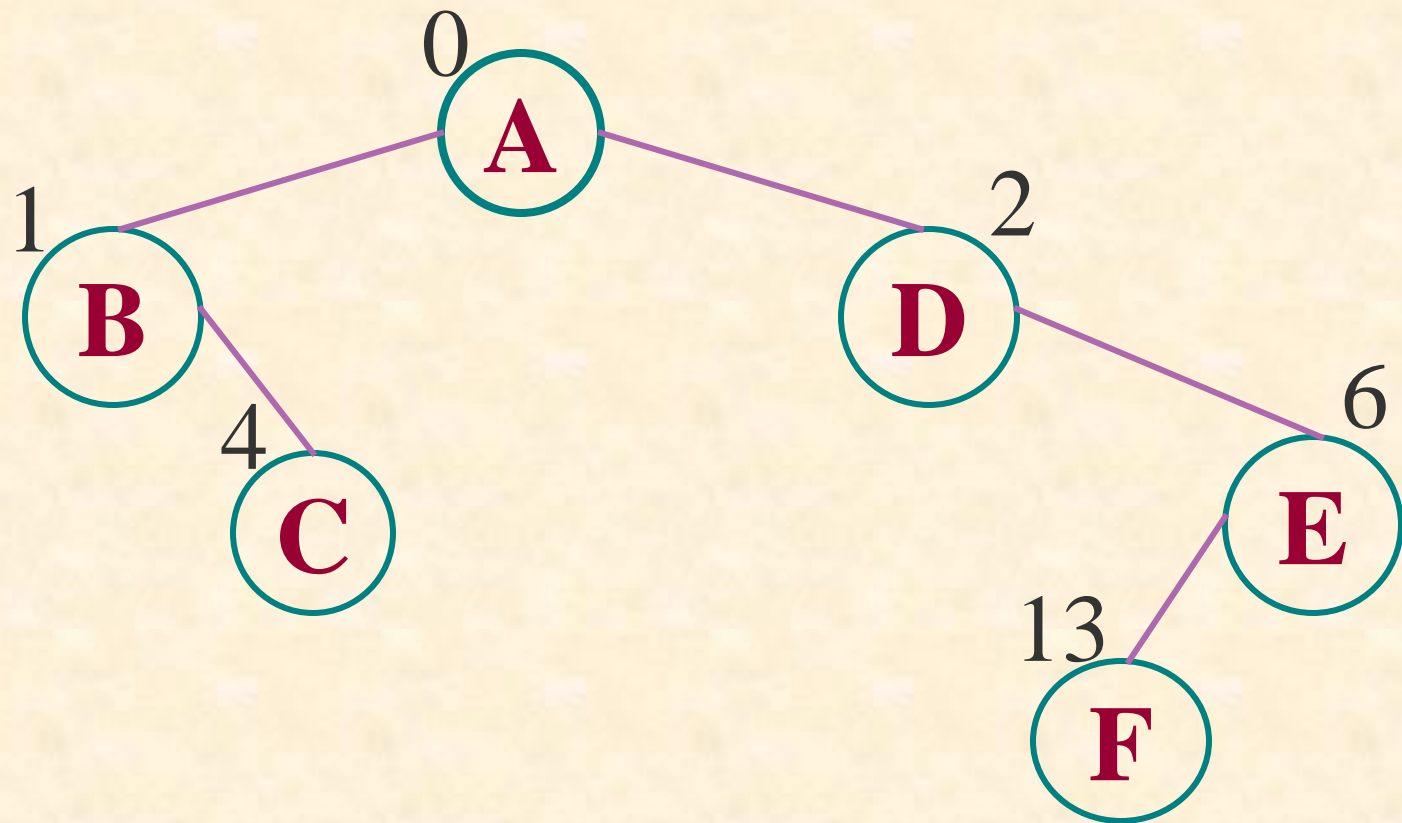
二、二叉树的链式
存储表示



一、 二叉树的顺序存储表示

```
#define MAX_TREE_SIZE 100
    // 二叉树的最大结点数
typedef TElemType SqBiTree[MAX_
    TREE_SIZE];
    // 0号单元存储根结点
SqBiTree bt;
```


例如:



0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	B	D		C		E							F



二、二叉树的链式存储表示

1. 二叉链表

3. 双亲链表

2. 三叉链表

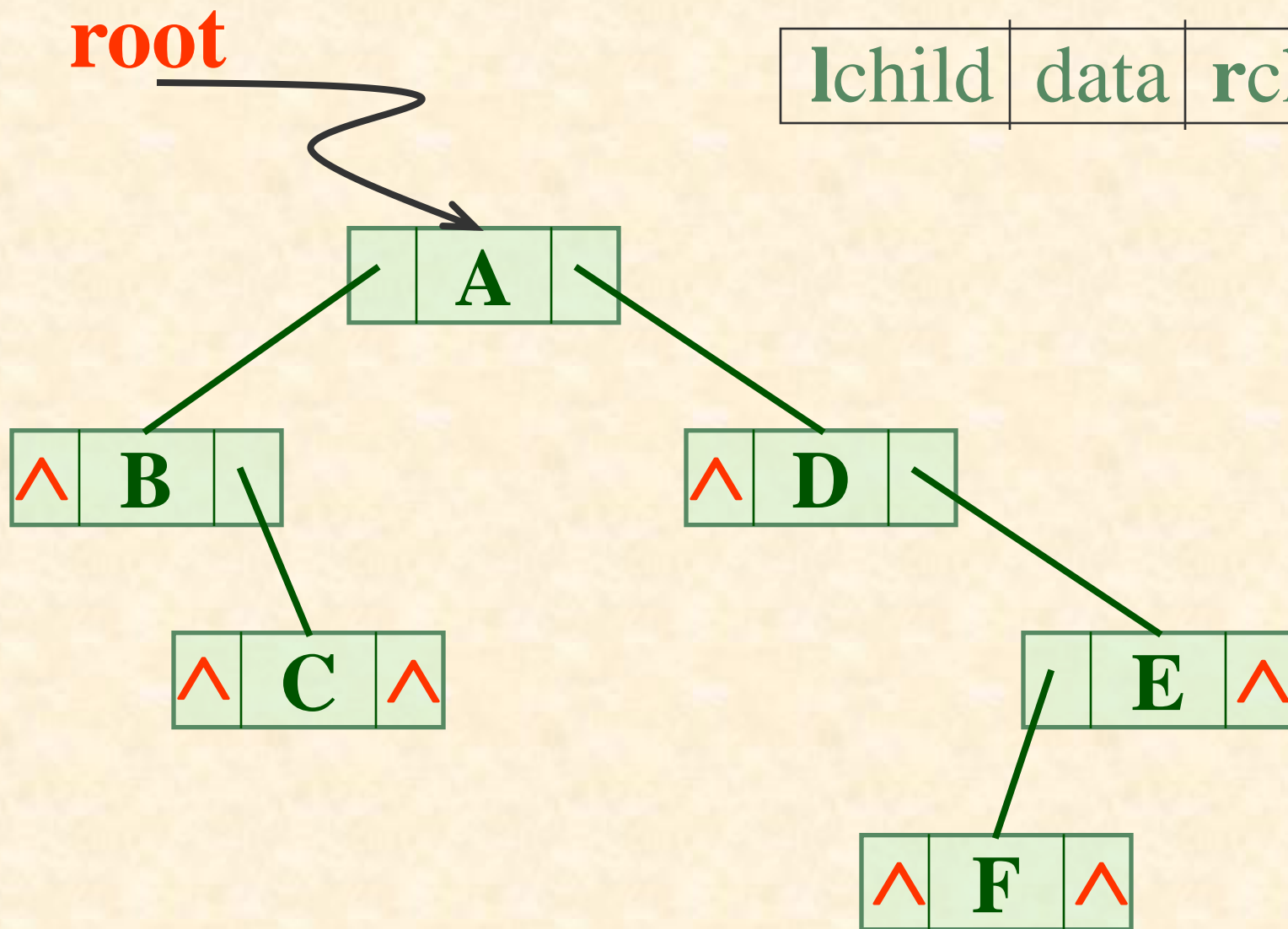
4. 线索链表



1. 二叉链表

结点结构:

lchild	data	rchild
--------	------	--------



C 语言的类型描述如下：

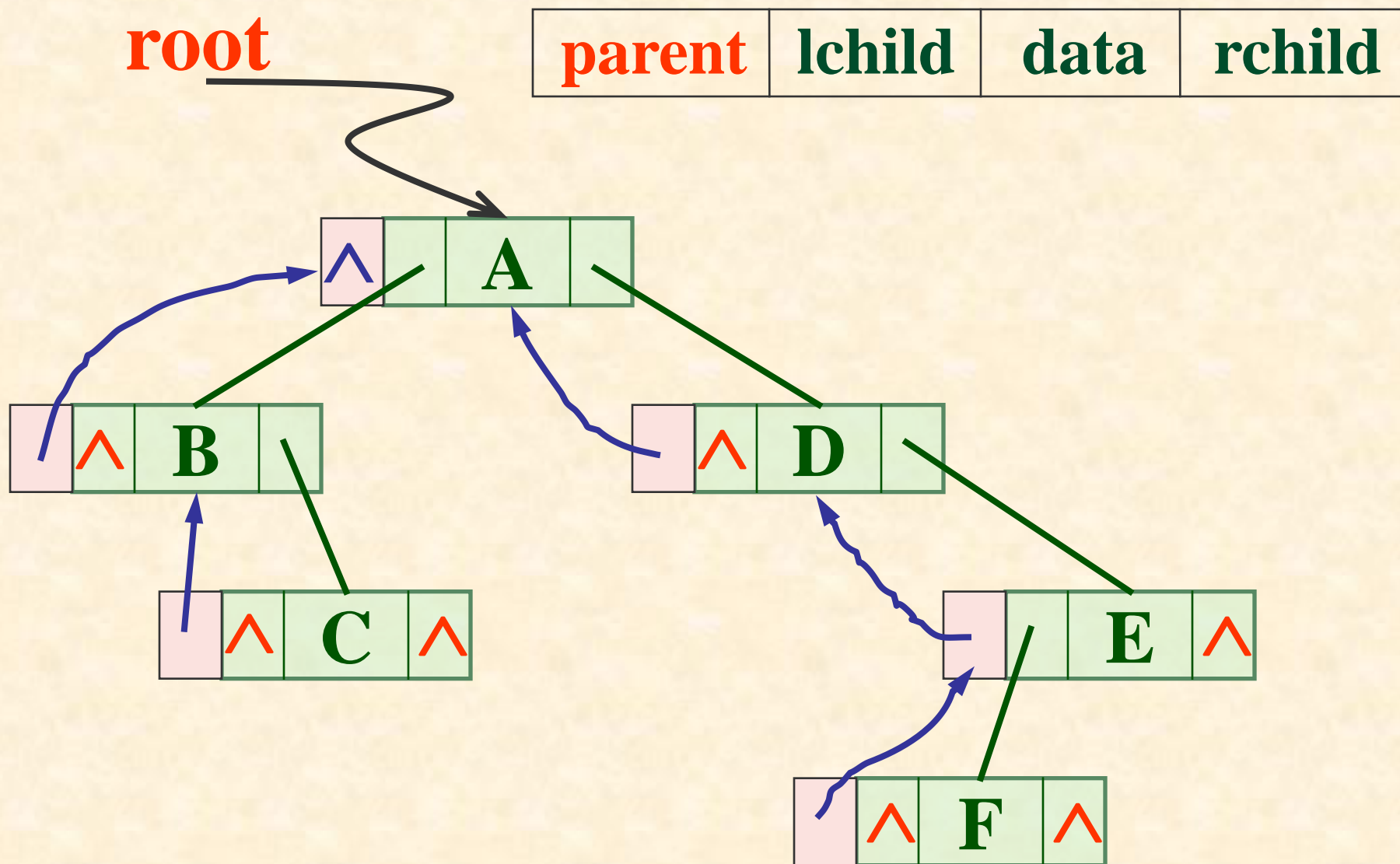
```
typedef struct BiTNode { // 结点结构  
    TElemType    data;  
    struct BiTNode *lchild, *rchild;  
                                // 左右孩子指针  
} BiTNode, *BiTree;
```

结点结构:



2. 三叉链表

结点结构:



C 语言的类型描述如下：

```
typedef struct TriTNode { // 结点结构  
    TElemType    data;  
    struct TriTNode *lchild, *rchild;  
                                // 左右孩子指针  
    struct TriTNode *parent; // 双亲指针  
} TriTNode, *TriTree;
```

结点结构：

parent	lchild	data	rchild
---------------	---------------	-------------	---------------



3. 双亲链表

结点结构:

data	parent	LRTag
------	--------	-------

0	B	2	L
1	C	0	R
2	A	-1	
3	D	2	R
4	E	3	R
5	F	4	L
6			

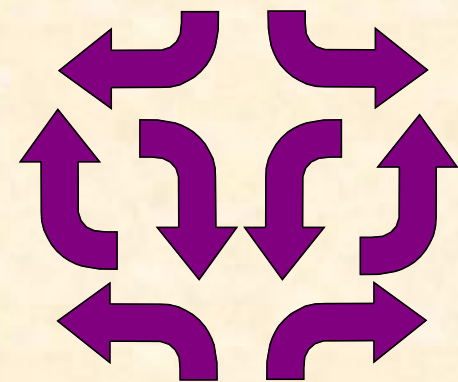
```
typedef struct BPTNode { // 结点结构  
    TElemType data;  
    int *parent;    // 指向双亲的指针  
    char LRTag;    // 左、右孩子标志域  
} BPTNode
```

```
typedef struct BPTree{ // 树结构  
    BPTNode nodes[MAX_TREE_SIZE];  
    int num_node;    // 结点数目  
    int root;        // 根结点的位置  
} BPTree
```



6.4

二叉树的遍历



一、问题的提出

二、先左后右的遍历算法

三、算法的递归描述

四、中序遍历算法的非递归描述

五、遍历算法的应用举例

一、问题的提出

顺着某一条搜索路径**巡访**二叉树中的结点，使得每个结点**均被访问一次**，而且**仅被访问一次**。

“**访问**”的含义可以很广，如：输出结点的信息等。

“**遍历**” 是任何类型均有的操作，
对线性结构而言， 只有一条搜索路
径(因为每个结点均只有一个后继)，
故不需要另加讨论。而二叉树是非
线性结构， 每个结点有两个后继，
则存在如何遍历即按什么样的**搜索**
路径遍历的问题。

对“二叉树”而言，可以有
三条搜索路径：

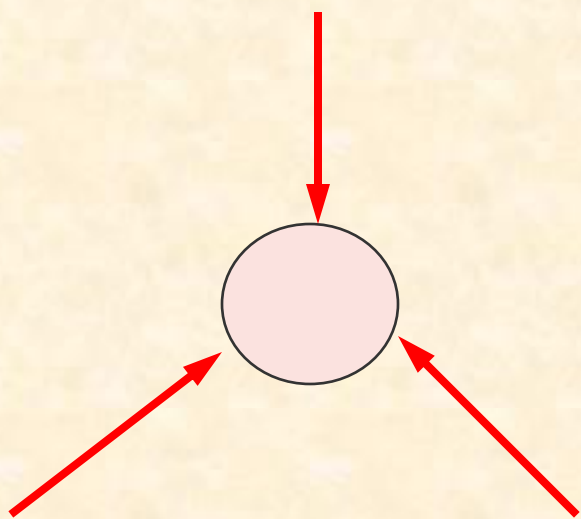
§ 1. 先上后下的按层次遍历；

§ 2. 先左（子树）后右（子树）
的遍历；

§ 3. 先右（子树）后左（子树）
的遍历。



二、先左后右的遍历算法



先（根）序的遍历算法

中（根）序的遍历算法

后（根）序的遍历算法



先（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

- （1）访问根结点；
- （2）先序遍历左子树；
- （3）先序遍历右子树。

● 中（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

（1）中序遍历左子树；

（2）访问根结点；

（3）中序遍历右子树。

● 后（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

（1）后序遍历左子树；

（2）后序遍历右子树；

（3）访问根结点。



三、算法的递归描述

```
void Preorder (BiTree T,  
                void( *visit)(TElemType& e))  
{ // 先序遍历二叉树  
  if (T) {  
    visit(T->data);           // 访问结点  
    Preorder(T->lchild, visit); // 遍历左子树  
    Preorder(T->rchild, visit); // 遍历右子树  
  }  
}
```



四、中序遍历算法的非递归描述

```
BiTNode *GoFarLeft(BiTree T, Stack *S){  
    if (!T ) return NULL;  
    while (T->lchild ) {  
        Push(S, T);  
        T = T->lchild;  
    }  
    return T;  
}
```

```
void Inorder_I(BiTree T, void (*visit)  
                (TelemType& e)){
```

```
Stack *S;
```

```
t = GoFarLeft(T, S); // 找到最左下的结点
```

```
while(t) {
```

```
    visit(t->data);
```

```
    if (t->rchild)
```

```
        t = GoFarLeft(t->rchild, S);
```

```
    else if ( !StackEmpty(S )) // 栈不空时退栈
```

```
        t = Pop(S);
```

```
        else    t = NULL; // 栈空表明遍历结束
```

```
    } // while
```

```
} // Inorder_I
```



五、遍历算法的应用举例

- 1、统计二叉树中叶子结点的个数
(先序遍历)
- 2、求二叉树的深度(后序遍历)
- 3、复制二叉树(后序遍历)
- 4、建立二叉树的存储结构

1、统计二叉树中叶子结点的个数

算法基本思想：

先序(或中序或后序)遍历二叉树，在遍历过程中查找叶子结点，并计数。

由此，需在遍历算法中增添一个“计数”的参数，并将算法中“访问结点”的操作改为：若是叶子，则计数器增1。

```
void CountLeaf (BiTree T, int& count){  
    if ( T ) {  
        if ((!T->lchild)&& (!T->rchild))  
            count++;    // 对叶子结点计数  
        CountLeaf( T->lchild, count);  
        CountLeaf( T->rchild, count);  
    } // if  
} // CountLeaf
```



2、求二叉树的深度(后序遍历)

算法基本思想：

首先分析二叉树的深度和它的左、右子树深度之间的关系。

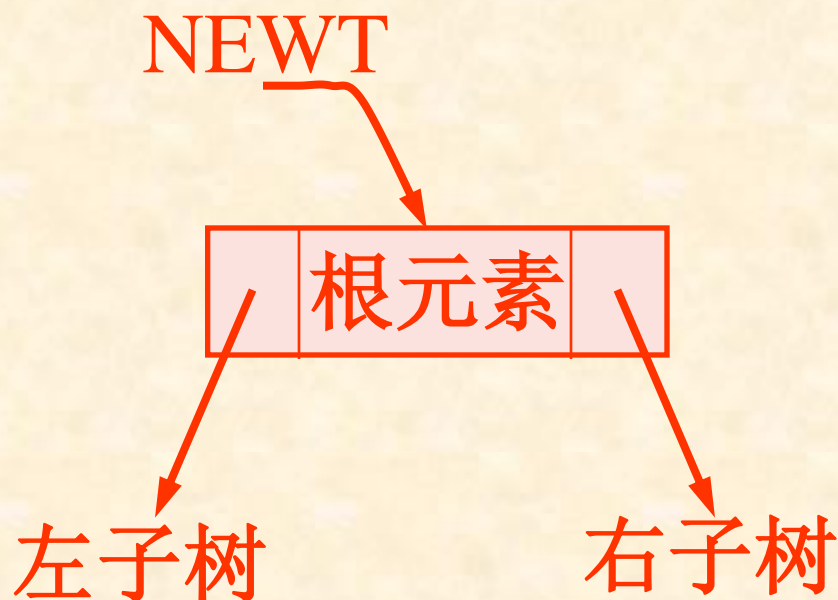
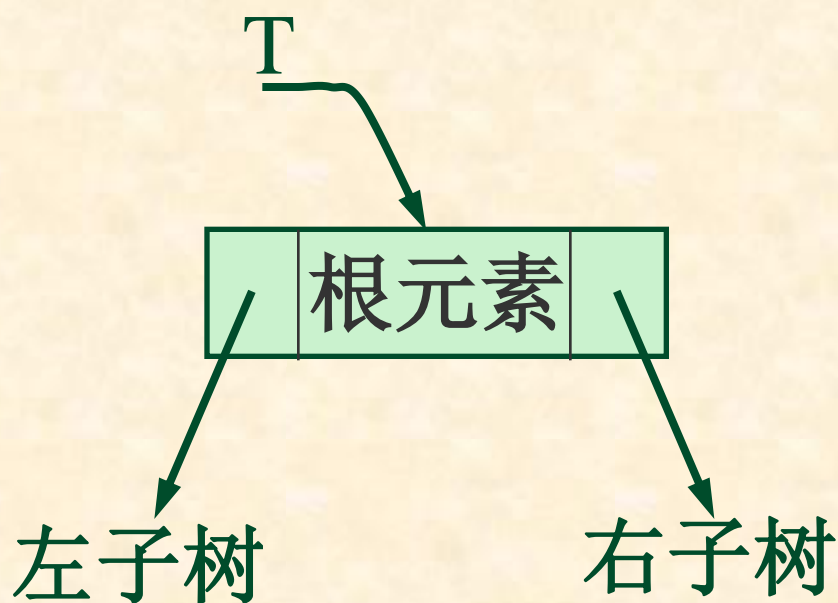
从二叉树深度的定义可知，二叉树的深度应为其左、右子树深度的最大值加1。由此，需先分别求得左、右子树的深度，算法中“访问结点”的操作为：求得左、右子树深度的最大值，然后加 1 。


```
int Depth (BiTree T ){ // 返回二叉树的深度
    if ( !T )    depthval = 0;
    else {
        depthLeft = Depth( T->lchild );
        depthRight= Depth( T->rchild );
        depthval = 1 + (depthLeft > depthRight ?
                        depthLeft : depthRight);
    }
    return depthval;
}
```



3、复制二叉树(后序遍历)

其基本操作为：生成一个结点。

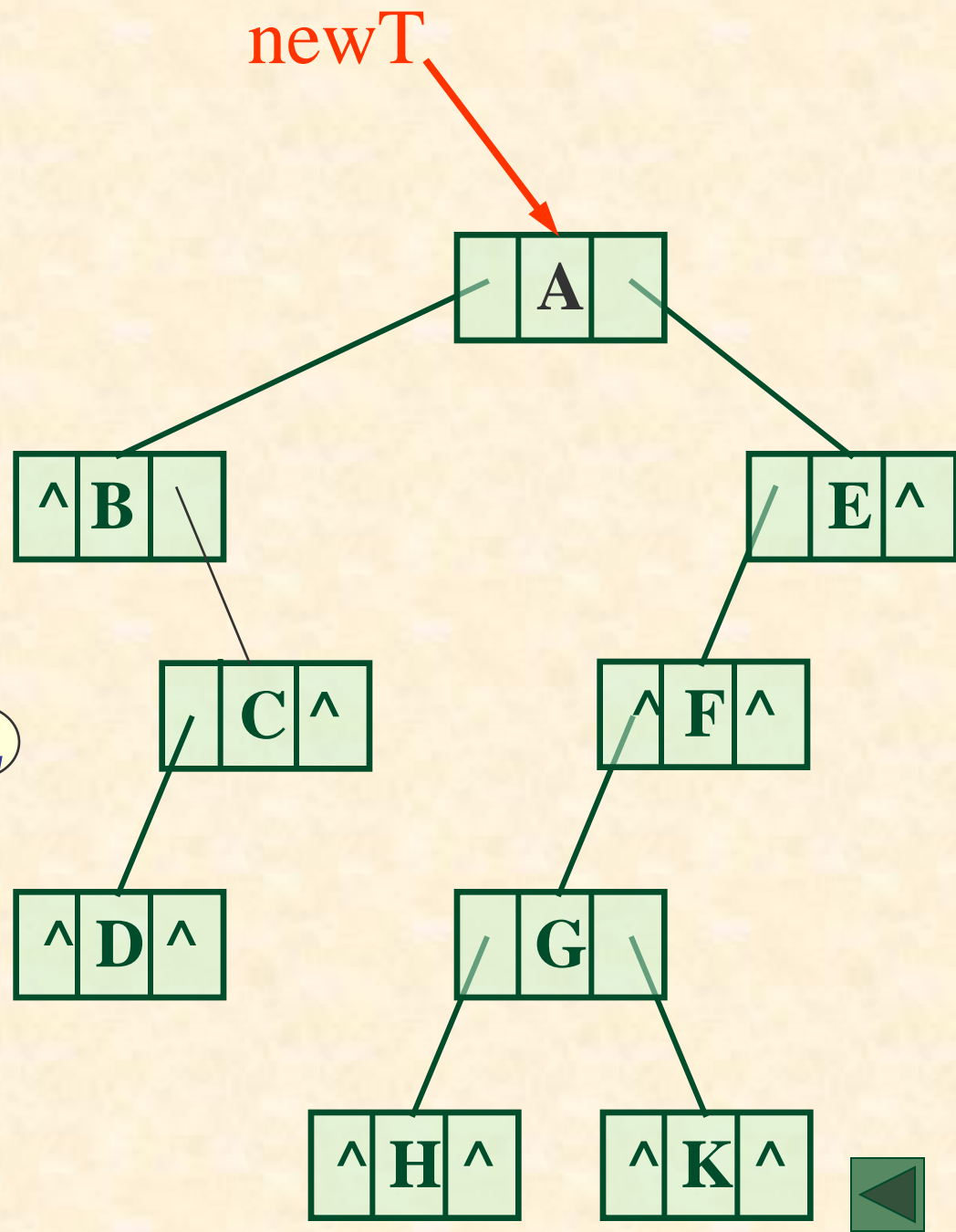
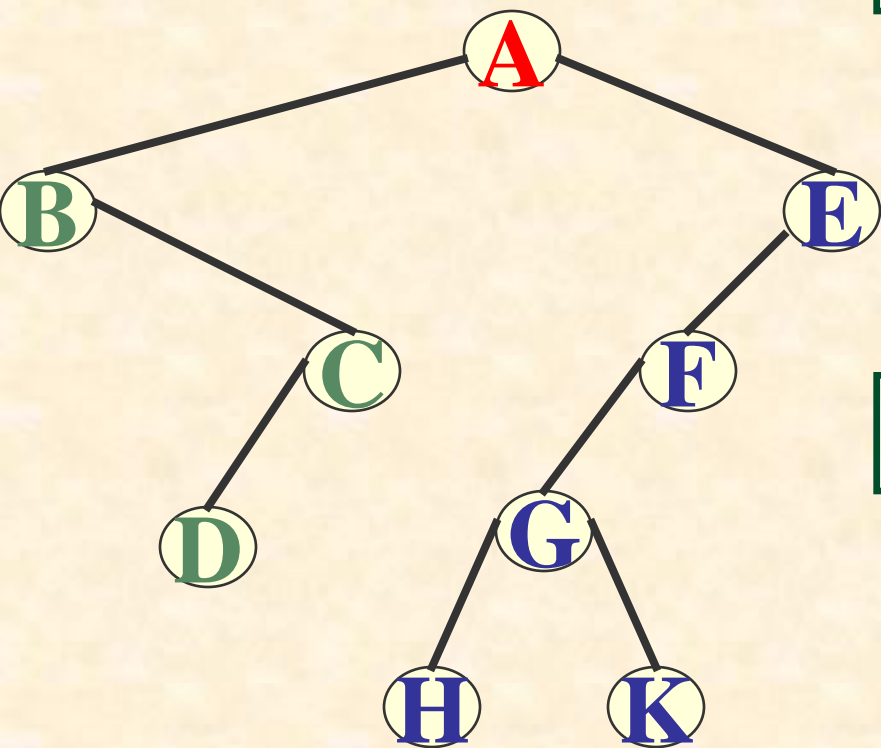


生成一个二叉树的结点
(其数据域为item,左指针域为lptr,右指针域为rptr)

```
BiTNode *GetTreeNode(TElemType item,  
    BiTNode *lptr , BiTNode *rptr ){  
    if (!(T = (BiTNode*)malloc(sizeof(BiTNode))))  
        exit(1);  
    T->data = item;  
    T->lchild = lptr;    T->rchild = rptr;  
    return T;  
}
```

```
BiTNode *CopyTree(BiTNode *T) {  
    if (!T)    return NULL;  
    if (T->lchild )  
        newlptr = CopyTree(T->lchild); //复制左子树  
    else newlptr = NULL;  
    if (T->rchild )  
        newrptr = CopyTree(T->rchild); //复制右子树  
    else newrptr = NULL;  
    newT = GetTreeNode(T->data, newlptr, newrptr);  
    return newT;  
} // CopyTree
```

例如：下列二叉树的复制过程如下：



4、建立二叉树的存储结构

不同的定义方法相应有不同的
存储结构的建立算法



● 以字符串的形式 根 左子树 右子树

定义一棵二叉树

例如：

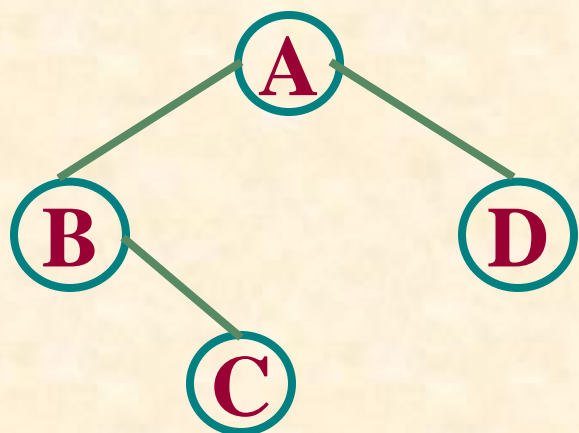
空树

以空白字符“**■**”表示

只含一个根结点的
二叉树

A

以字符串“**A ■ ■**”表示



以下列字符串表示

A(**B**(**■**,**C**(**■**,**■**)),**D**(**■**,**■**))

```
Status CreateBiTree(BiTree &T) {
```

```
    scanf(&ch);
```

```
    if (ch==' ') T = NULL;
```

```
    else {
```

```
        if (!(T = (BiTNode *)malloc(sizeof(BiTNode))))
```

```
            exit(OVERFLOW);
```

```
        T->data = ch;           // 生成根结点
```

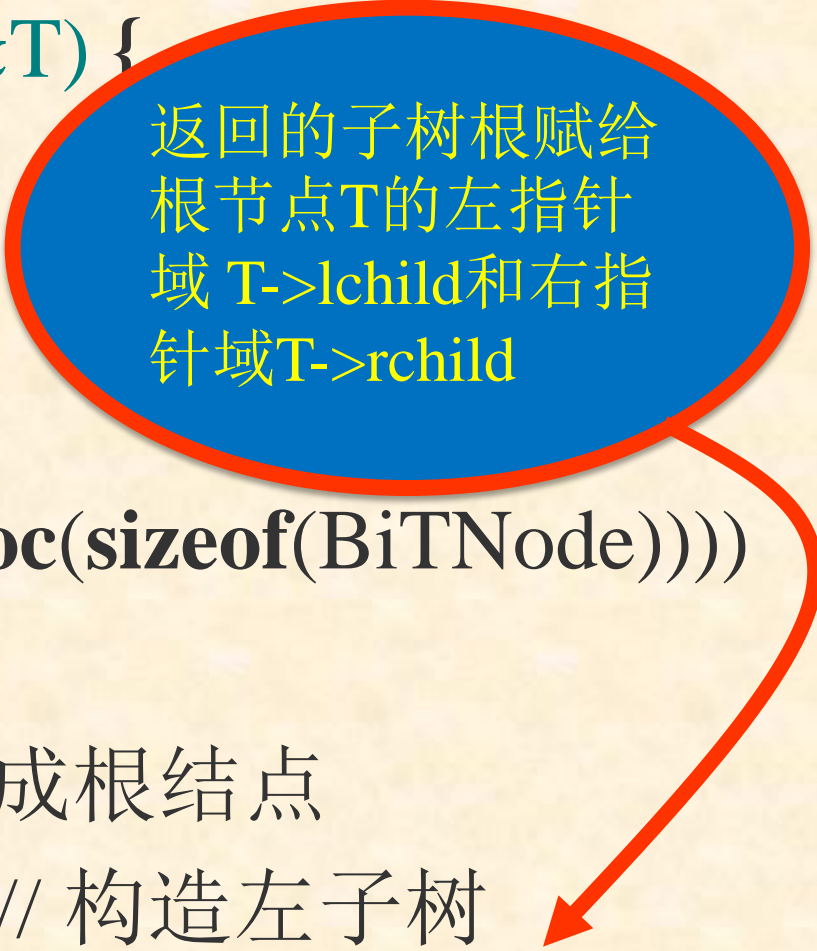
```
        CreateBiTree(T->lchild); // 构造左子树
```

```
        CreateBiTree(T->rchild); // 构造右子树
```

```
    }
```

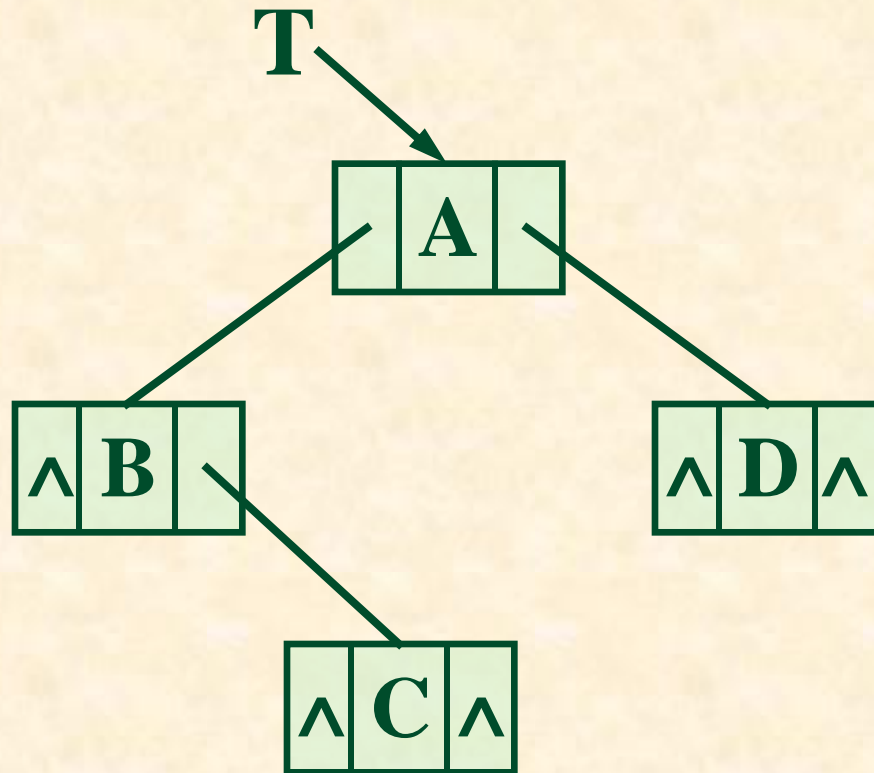
```
    return OK; } // CreateBiTree
```

返回的子树根赋给
根节点T的左指针
域 T->lchild和右指
针域T->rchild



上页算法执行过程举例如下：

A **B** ■ **C** ■ ■ **D** ■ ■



- 按给定的表达式建相应二叉树

- 由先缀表示式建树

例如：已知表达式的先缀表示式

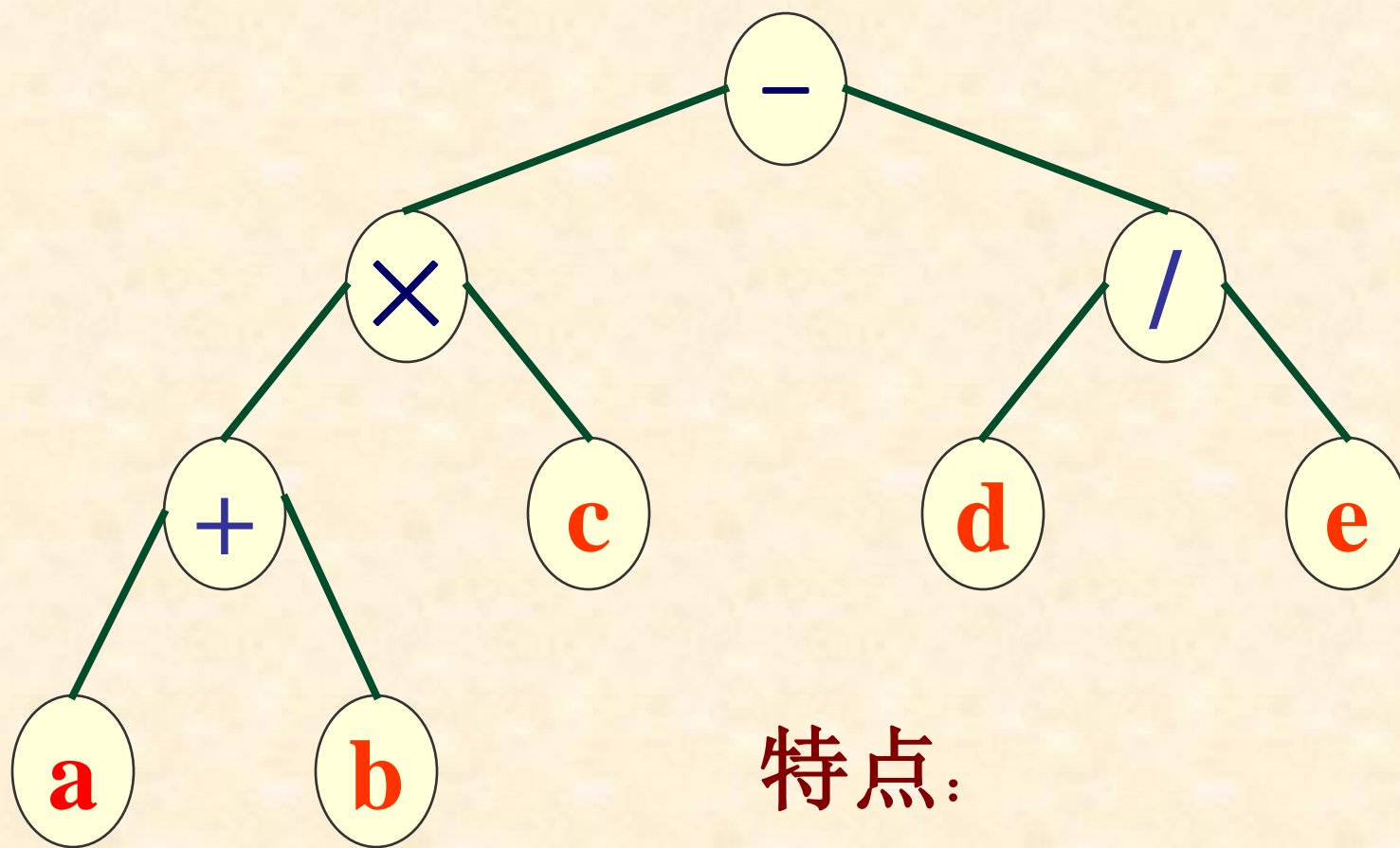
$$- \times + a b c / d e$$

- 由原表达式建树

例如：已知表达式 $(a+b) \times c - d/e$



对应先缀表达式 $- \times + a b c / d e$ 的二叉树



特点:

操作数为叶子结点

运算符为分支结点

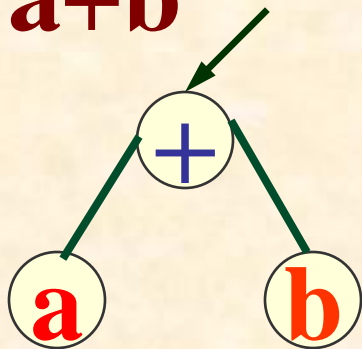
由先缀表示式建树的算法的基本操作:

```
scanf(&ch);  
if ( In(ch, 字母集 )) 建叶子结点;  
else { 建根结点;  
        递归建左子树;  
        递归建右子树;  
}
```

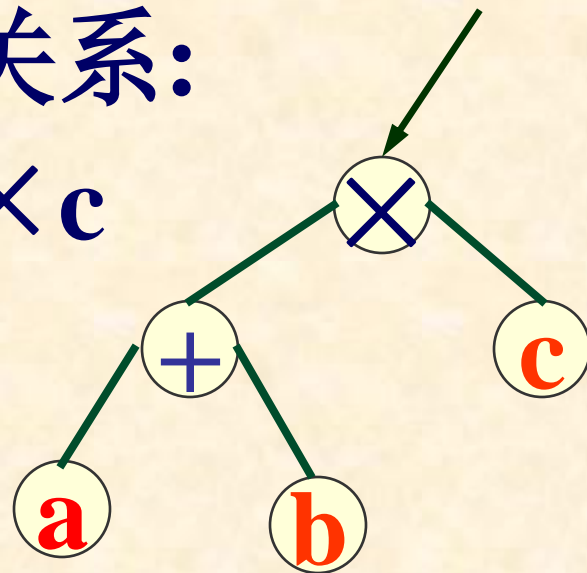


分析表达式和二叉树的关系:

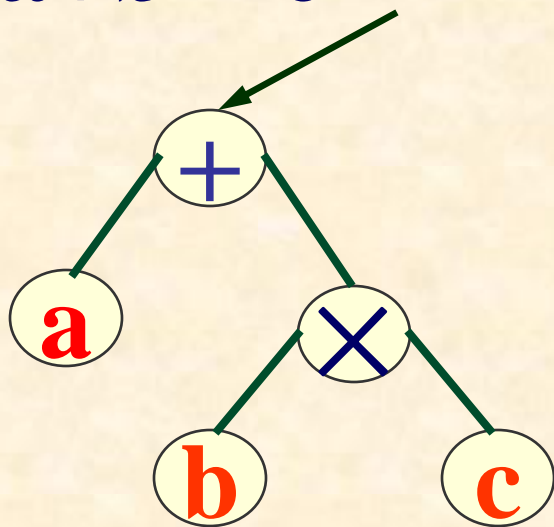
$a+b$



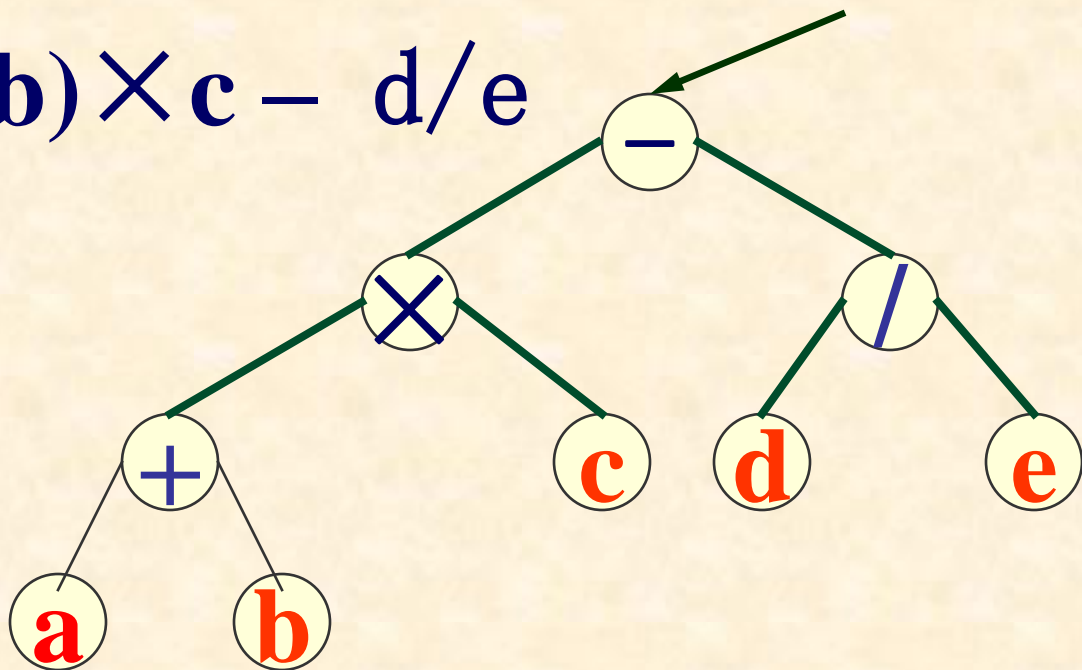
$(a+b) \times c$



$a+b \times c$



$(a+b) \times c - d/e$



基本操作:

scanf(&ch);

if (In(ch, 字母集)) { 建叶子结点; 暂存; }

else if (In(ch, 运算符集))

{ 和前一个运算符比较优先数;

若当前的优先数“高”，则暂存;

否则建子树;

}

```
void CrtExptree(BiTree &T, char exp[] ) {  
    InitStack(S); Push(S, '#'); InitStack(PTR);  
    p = exp; ch = *p;  
    while (!(GetTop(S)=='#' && ch=='#')) {  
        if (!IN(ch, OP)) CrtNode( t, ch );  
                                // 建叶子结点并入栈  
        else { ... ... }  
        if ( ch!= '#' ) { p++; ch = *p;}  
    } // while  
    Pop(PTR, T);  
} // CrtExptree
```



```
switch (ch) {  
    case '(' : Push(S, ch); break;  
    case ')' : Pop(S, c);  
        while (c!= '(' ) {  
            CrtSubtree( t, c); // 建二叉树并入栈  
            Pop(S, c)      }  
        break;  
    default :  
        ... ..  
} // switch
```




```
while(!Gettop(S, c) && ( precede(c,ch)))  
{  
    CrtSubtree( t, c);  
    Pop(S, c);  
}  
if ( ch!= '#' ) Push( S, ch);  
break;
```



建叶子结点的算法为:

```
void CrtNode(BiTree& T,char ch)
{
    T=(BiTNode*)malloc(sizeof(BiTNode));
    T->data = ch;
    T->lchild = T->rchild = NULL;
    Push( PTR, T );
}
```

建子树的算法为：

```
void CrtSubtree (Bitree& T, char c)
{
    T=(BiTNode*)malloc(sizeof(BiTNode));
    T->data = c;
    Pop(PTR, rc); T->rchild = rc;
    Pop(PTR, lc); T->lchild = lc;
    Push(PTR, T);
}
```



● 由二叉树的先序和中序序列建树

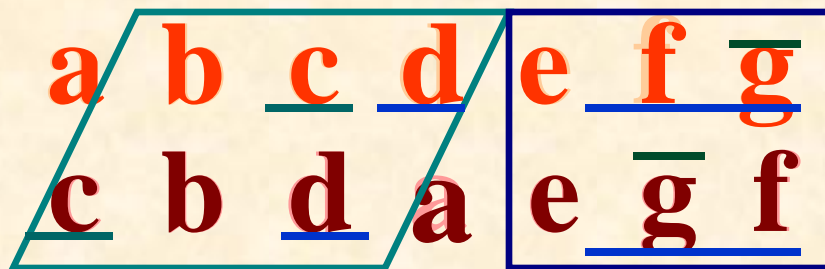
仅知二叉树的先序序列 “**abcdefg**”
不能唯一确定一棵二叉树，

如果同时已知二叉树的中序序列
“**cbdaegf**”，则会如何？

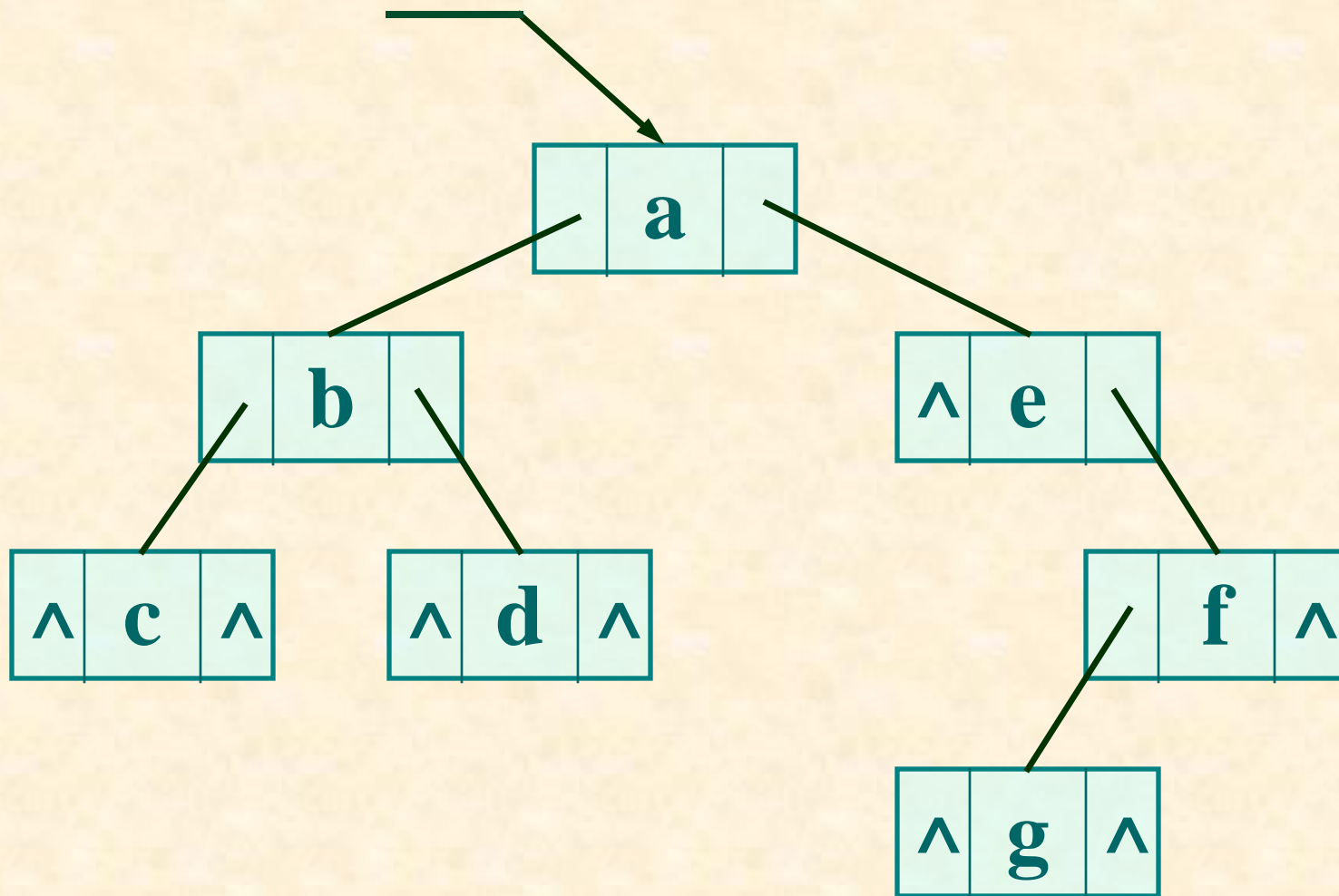
二叉树的先序序列 **根** **左子树** **右子树**

二叉树的中序序列 **左子树** **根** **右子树**

例如：



先序序列
中序序列



```
void CrtBT(BiTree& T, char pre[], char ino[],  
           int ps, int is, int n ) {  
    // 已知pre[ps..ps+n-1]为二叉树的先序序列,  
    // ino[is..is+n-1]为二叉树的中序序列, 本算  
    // 法由此两个序列构造二叉链表  
    if (n==0) T=NULL;  
    else {  
        k=Search(ino, pre[ps]); // 在中序序列中查询  
        if (k== -1) T=NULL;  
        else {      ... ..      }  
    } //  
} // CrtBT
```






```
T=(BiTNode*)malloc(sizeof(BiTNode));  
T->data = pre[ps];  
if (k==is) T->Lchild = NULL;  
else  CrtBT(T->Lchild, pre[], ino[],  
            ps+1, is, k-is );  
if (k==is+n-1) T->Rchild = NULL;  
else  CrtBT(T->Rchild, pre[], ino[],  
            ps+1+(k-is), k+1, n-(k-is)-1 );
```



6.5

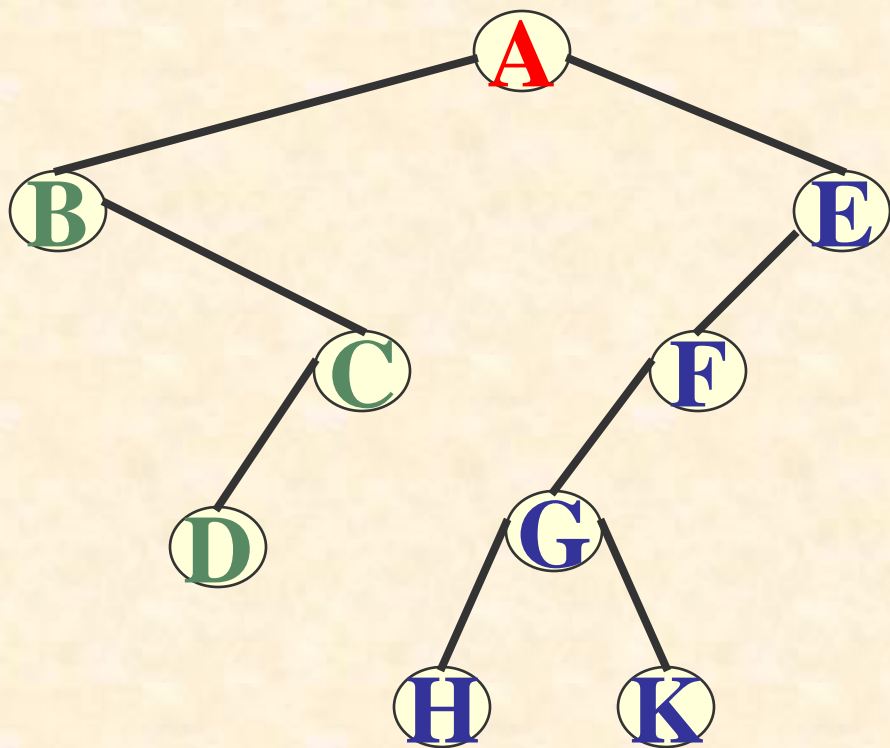
线索二叉树

- § 何谓线索二叉树？ 
- § 如何建立线索链表？ 
- § 线索链表的遍历算法 

一、何谓线索二叉树？

遍历二叉树的结果是，
求得结点的一个线性序列。

例如：



先序序列：

A B C D E F G H K

中序序列：

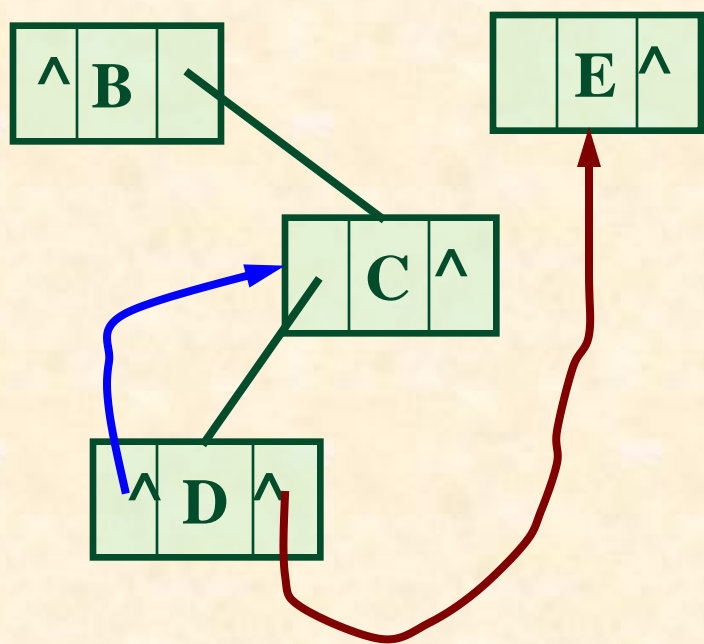
B D C A H G K F E

后序序列：

D C B H K G F E A

指向该线性序列中的“前驱”和“后继”的指针，称作“**线索**”

A B C D E F G H K



包含“**线索**”的存储结构，称作“**线索链表**”

与其相应的二叉树，称作“**线索二叉树**”

并不是每个节点都加前驱和后继信息，**线索化二叉树是在二叉树的基础上来构成的。**

那么前驱后继指针加在哪里呢？

利用空链域

前驱加在左子树为空的时候

后继加在右子树为空的时候

n 个结点的二叉树有 $n+1$ 个空链域，

用这 $n+1$ 个空链域来存线索

对线索链表中结点的约定：

在二叉链表的结点中，指针共用，增加两个标志域

lchild	LTag	data	RTag	rchild
--------	------	------	------	--------

- 若该结点的左子树不空，
则Lchild域的指针指向其左子树，
且左标志域的值为“指针 Link”；
否则，Lchild域的指针指向其“前驱”
且左标志的值为“线索 Thread”。

lchild	LTag	data	RTag	rchild
--------	------	------	------	--------

- 若该结点的右子树不空，
则rchild域的指针指向其右子树，
且右标志域的值“指针 Link”；
否则，rchild域的指针指向其“后继”，
且右标志的值为“线索 Thread”。
- 如此定义的二叉树的存储结构称作
“线索链表”。

线索链表的类型描述:

```
typedef enum { Link, Thread } PointerThr;
```

```
// Link==0:指针, Thread==1:线索
```

```
typedef struct BiThrNod {
```

```
TElemType      data;
```

```
struct BiThrNode *lchild, *rchild; // 左右指针
```

```
PointerThr      LTag, RTag; // 左右标志
```

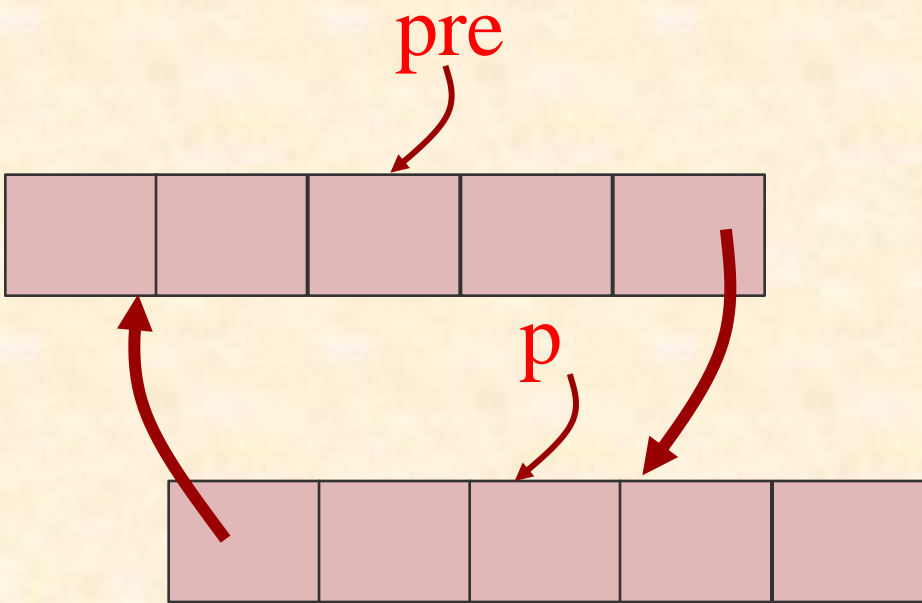
```
} BiThrNode, *BiThrTree;
```



二、如何建立线索链表？

在中序遍历过程中修改结点的左、右指针域，以保存当前访问结点的“前驱”和“后继”信息。遍历过程中，附设指针pre和p，p指向当前访问的结点，pre指向p的前驱。

通过pre和p方便获取当前结点的前驱和后继。然后修改左右子树有为空的域，不空的不改。



pre指向刚刚访问过的节点
p指向当前访问的节点

如果 $p \rightarrow lchild = \wedge$
则 $p \rightarrow lchild = pre$
即让 $p \rightarrow lchild$ 指向它的前驱

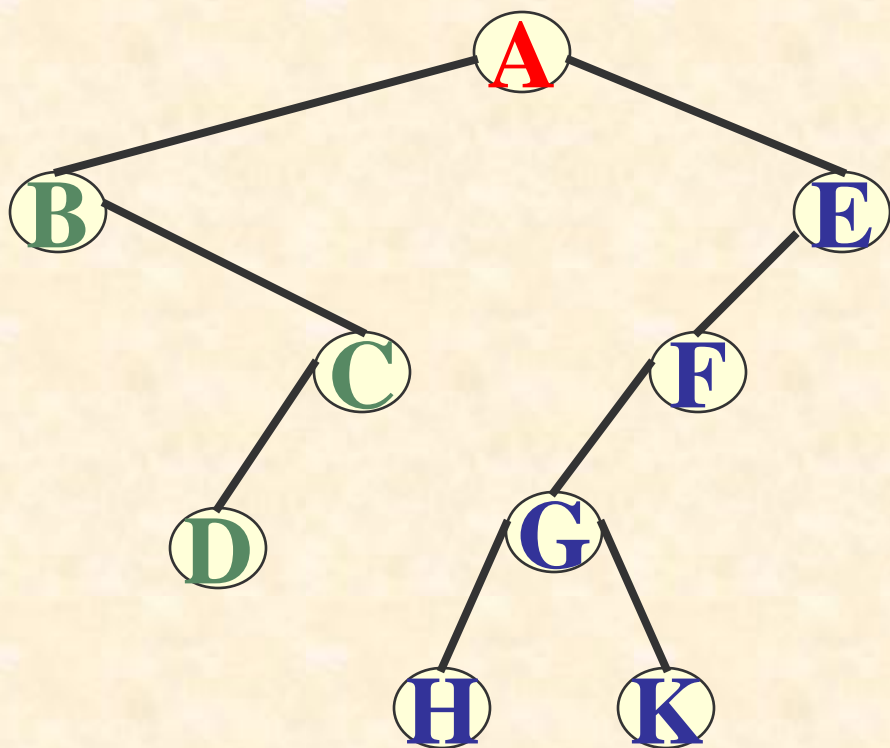
如果 $pre \rightarrow rchild = \wedge$
则 $pre \rightarrow rchild = p$
即 $pre \rightarrow rchild$ 指向它的后继

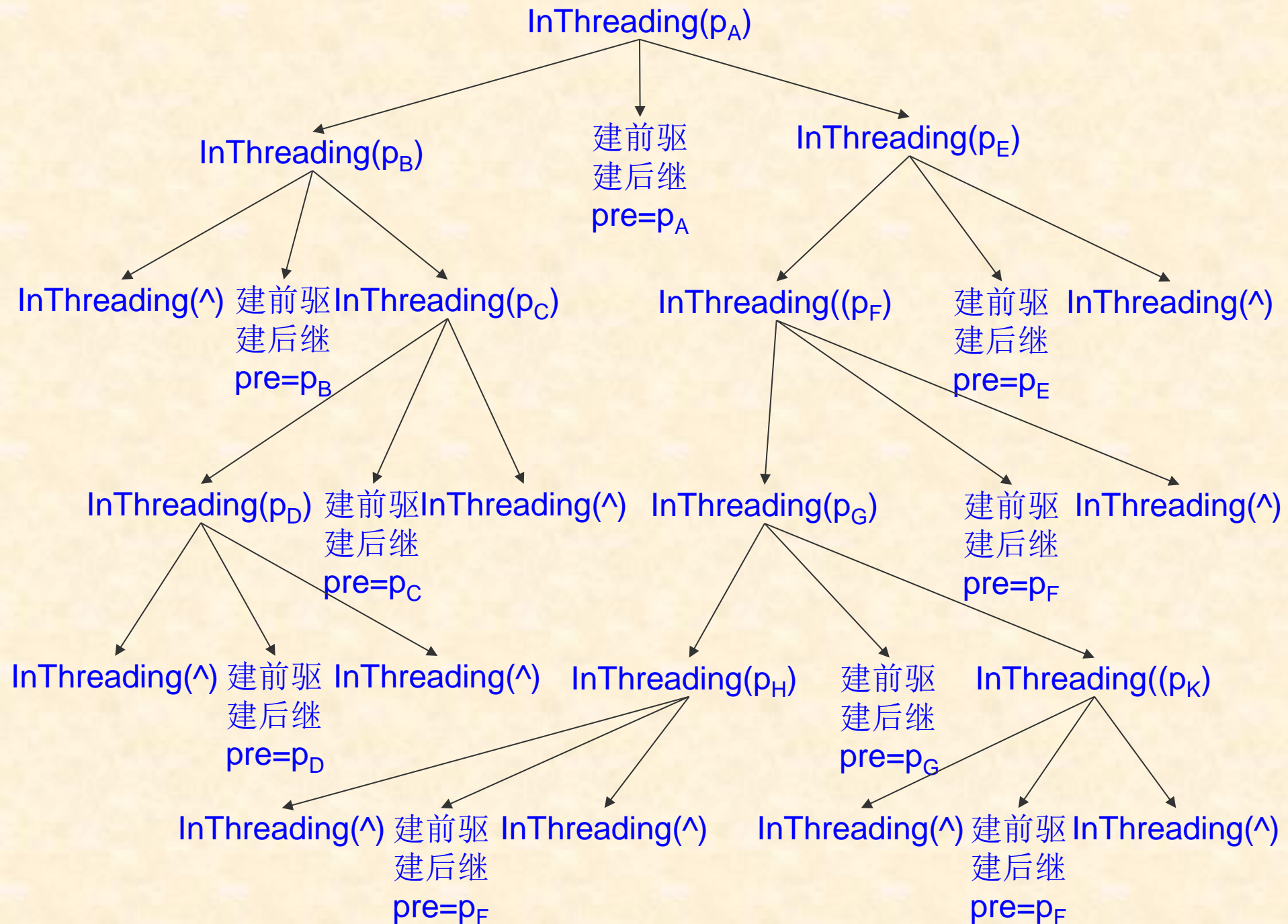

```
void InThreading(BiThrTree p) {  
    if (p) { // 对以p为根的非空二叉树进行线索化  
        InThreading(p->lchild); // 左子树线索化  
        if (!p->lchild) // 建前驱线索  
            { p->LTag = Thread; p->lchild = pre; }  
        if (!pre->rchild) // 建后继线索  
            { pre->RTag = Thread; pre->rchild = p; }  
        pre = p; // 保持 pre 指向 p 的前驱  
        InThreading(p->rchild); // 右子树线索化  
    } // if  
} // InThreading
```

中序线索化

中序序列:

B D C A H G K F E

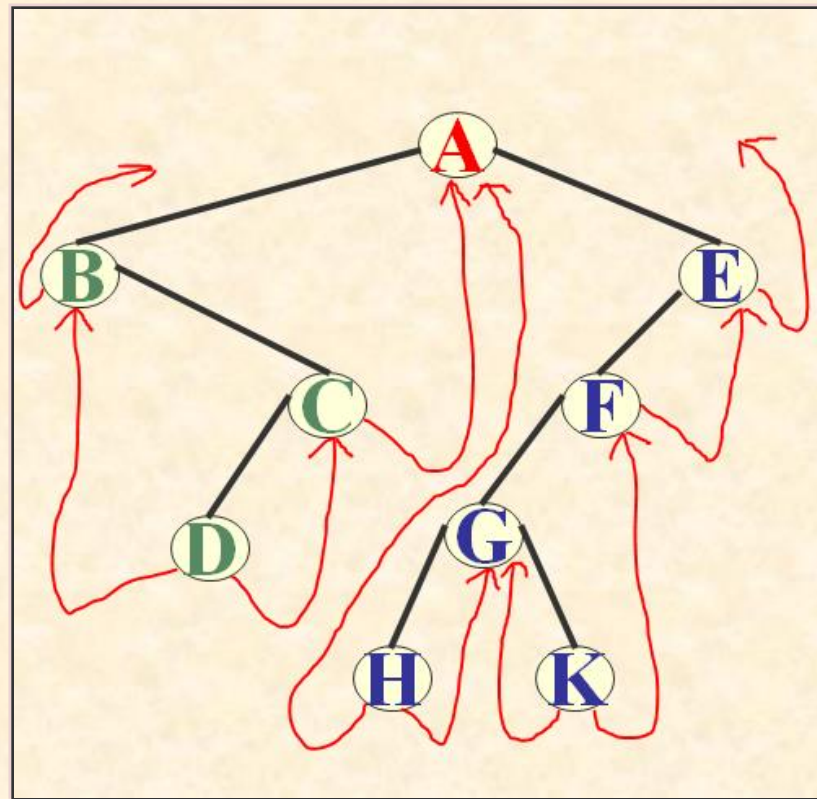
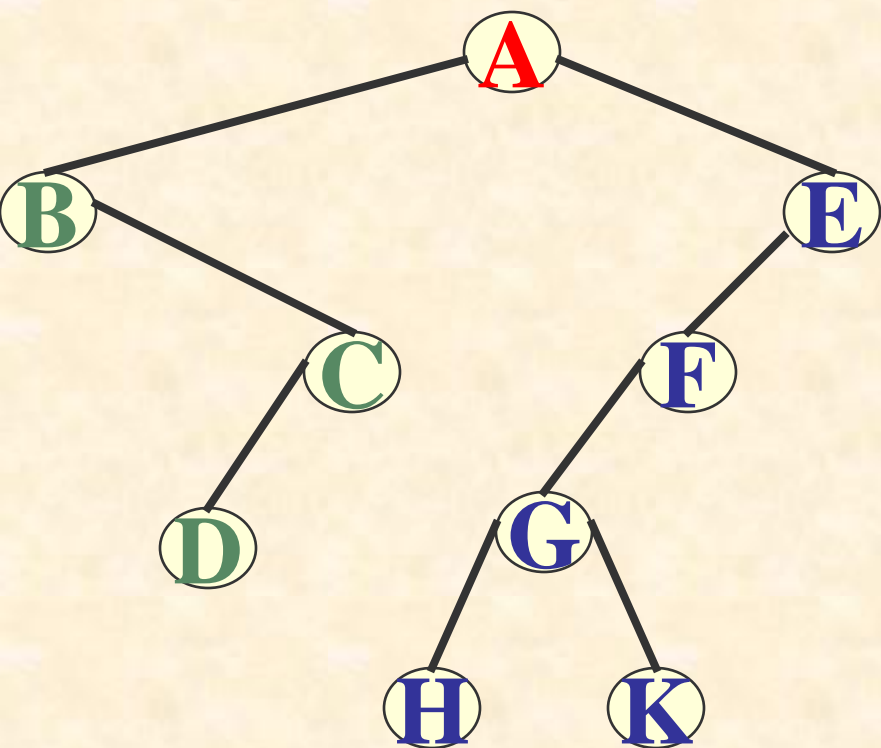




中序线索化

中序序列:

B D C A H G K F E



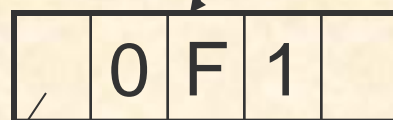
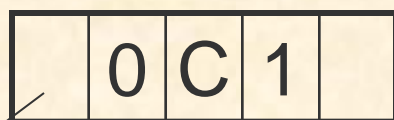
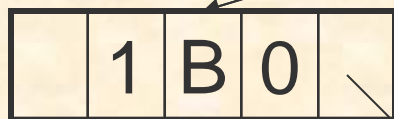
线索树无根
怎么破？

lchild	LTag	data	RTag	rchild
--------	------	------	------	--------

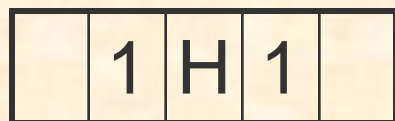
开始时Pre指向头结点 pre



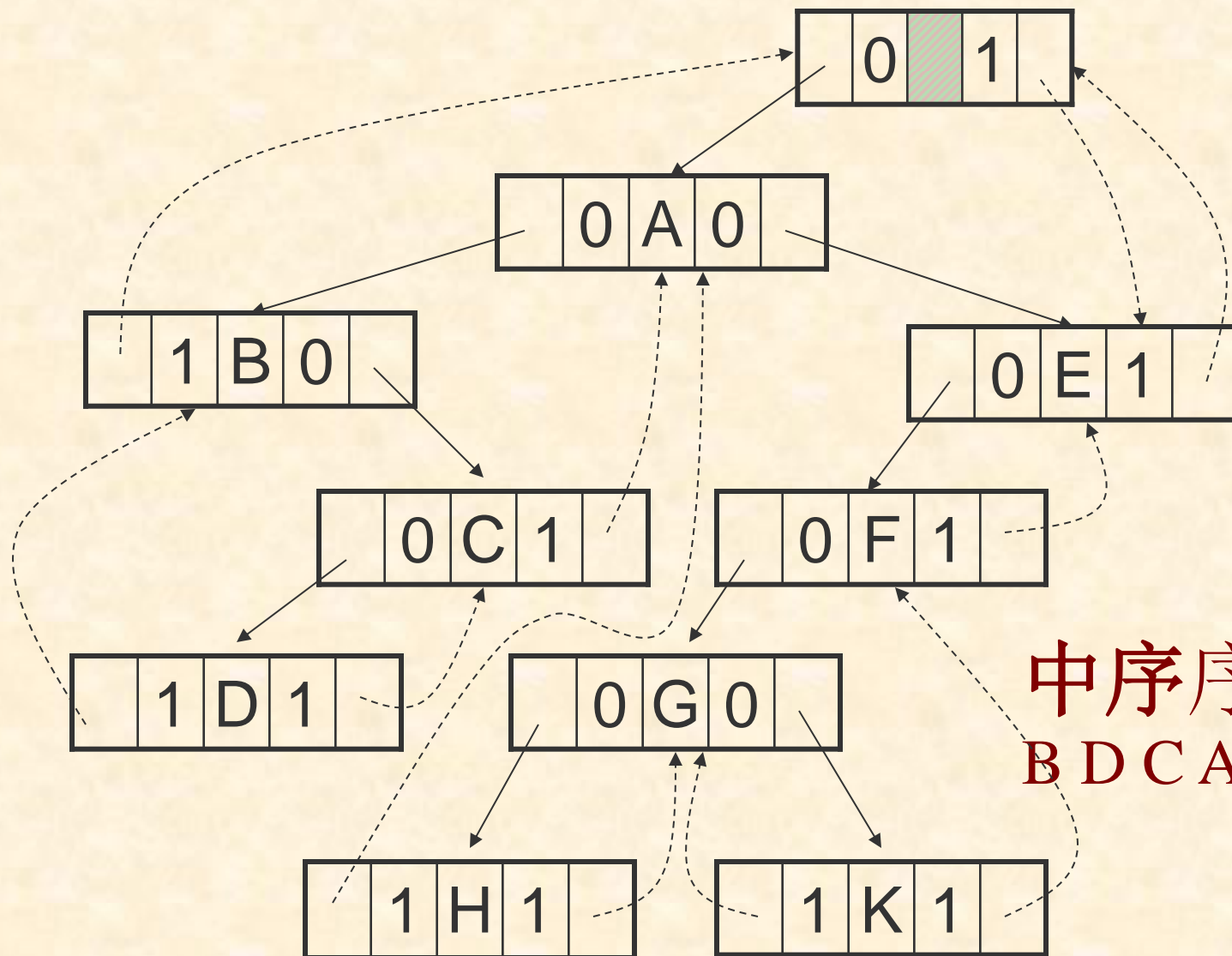
增加一个头结点



中序序列:
B D C A H G K F E



lchild	lTag	data	rTag	rchild
--------	------	------	------	--------



中序序列:
B D C A H G K F E

```
Status InOrderThreading(BiThrTree &Thrt,  
                        BiThrTree T) { // 构建中序线索链表  
    if (!(Thrt = (BiThrTree)malloc(  
                                sizeof( BiThrNode))))  
        exit (OVERFLOW);  
    Thrt->LTag = Link; Thrt->RTag = Thread;  
    Thrt->rchild = Thrt;    // 添加头结点  
    ... ..  
    return OK;  
} // InOrderThreading
```




```
if (!T) Thrt->lchild = Thrt;
```

```
else {
```

```
    Thrt->lchild = T;  pre = Thrt;
```

```
    InThreading(T);
```

```
    pre->rchild = Thrt;  // 处理最后一个结点
```

```
    pre->RTag = Thread;
```

```
    Thrt->rchild = pre;
```

```
}
```



三、线索链表的遍历算法:

由于在线索链表中添加了遍历中得到的“前驱”和“后继”的信息，省去了栈（递归栈或非递归栈），因为前后关系已经通过存储结构解决。从而简化了遍历的算法。

```
for ( p = firstNode(T); p; p = Succ(p) )
```

```
Visit (p);
```

每次访问完p都指向他的后继，然后继续访问。省略了栈！

例如：

对中序线索化链表的遍历算法

※ **中序遍历的第一个结点** ？

左子树上处于“最左下”（没有左子树）的结点。

※ **在中序线索化链表中结点的后继** ？

若无右子树，**则为**后继线索所指结点；
否则为对其右子树进行中序遍历时访问的第一个结点。

```
void InOrderTraverse_Thr(BiThrTree T,  
                           void (*Visit)(TElemType e)) {  
    p = T->lchild;    // p指向根结点  
    while (p != T) {    // 空树或遍历结束时, p==T  
        while (p->LTag==Link) p = p->lchild; // 第一个结点  
        if(!Visit(p->data)) return ERROR;//访问找到的第一个结点  
        while (p->RTag==Thread && p->rchild!=T) {  
            p = p->rchild; Visit(p->data);    // 访问后继结点  
        }  
        p = p->rchild;    // p进至其右子树根  
    }  
} // InOrderTraverse_Thr
```



在后序线索二叉树中，若某结点 p 且 $p \rightarrow rtag=1$ ，
则 p 的直接后继是 [填空1]

6.6 树和森林 的表示方法



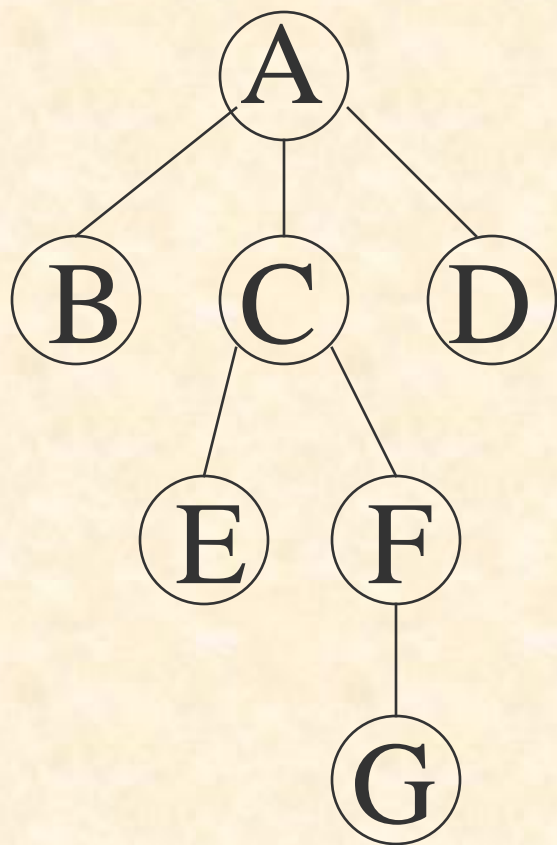
树的三种存储结构

一、双亲表示法

二、孩子链表表示法

**三、树的二叉链表(孩子-兄弟)
存储表示法**

一、双亲表示法:



data parent

0

A

-1

1

B

0

2

C

0

3

D

0

4

E

2

5

F

2

6

G

5

r=0

n=6

C语言的类型描述:

```
#define MAX_TREE_SIZE 100
```

结点结构:

data	parent
-------------	---------------

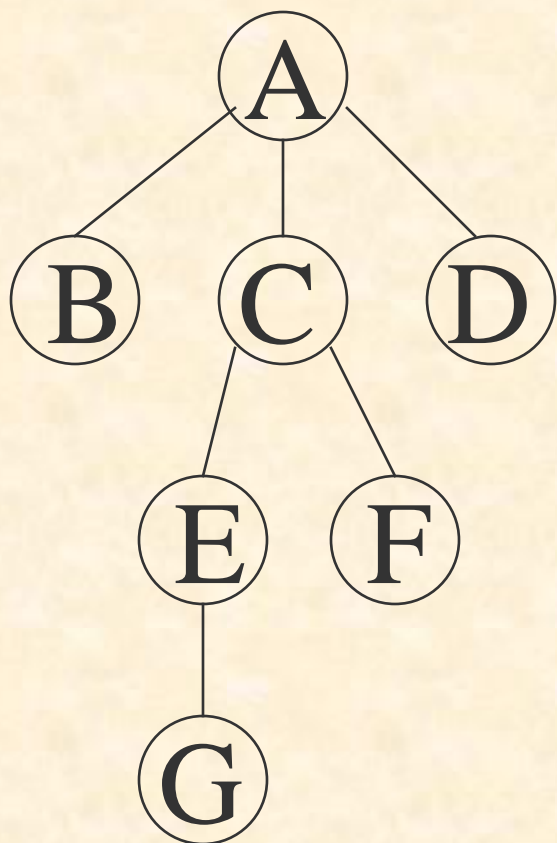
```
typedef struct PTNode {  
    Elem data;  
    int   parent; // 双亲位置域  
} PTNode;
```


树结构:

```
typedef struct {  
    PTNode nodes  
        [MAX_TREE_SIZE];  
    int    r, n;  
        // 根结点的位置和结点个数  
} PTree;
```



二、孩子链表表示法:



	data	firstchild	
0	A	-1	→ [1] → [2] → [3] Λ
1	B	0	Λ
2	C	0	→ [4] → [5] Λ
3	D	0	Λ
4	E	2	→ [6] Λ
5	F	2	Λ
6	G	4	Λ

r=0

n=6

C语言的类型描述:

孩子结点结构:

child	next
--------------	-------------

```
typedef struct CTNode {  
    int      child;  
    struct CTNode *next;  
} *ChildPtr;
```

双亲结点结构

data	firstchild
-------------	-------------------

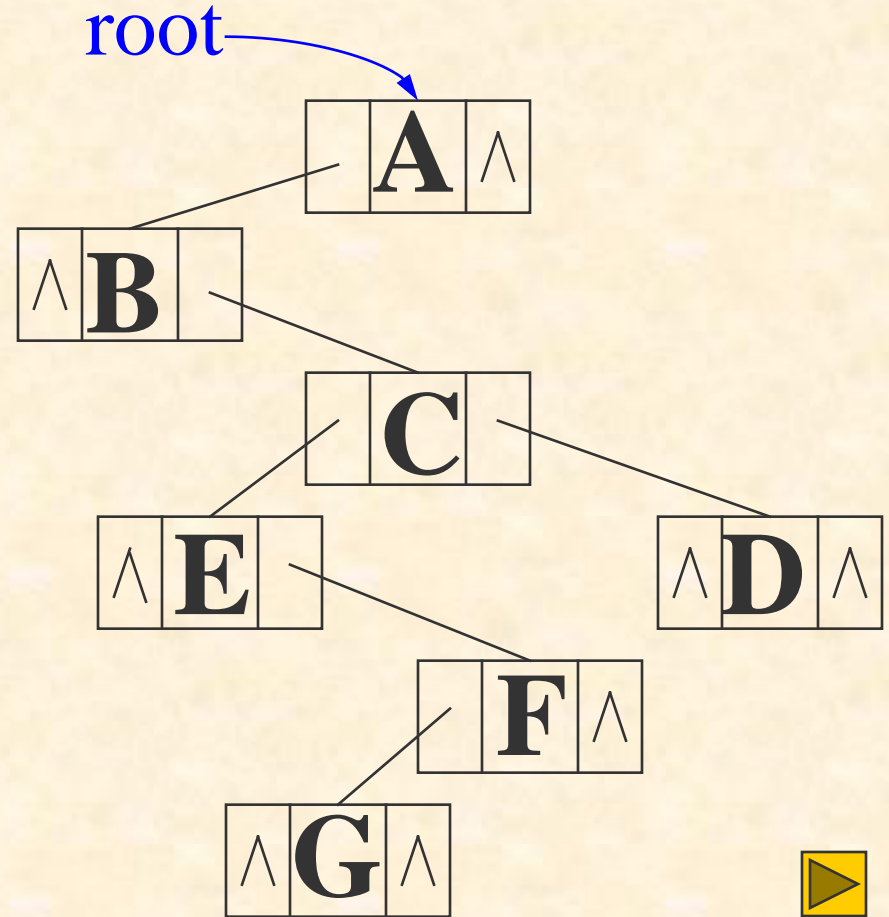
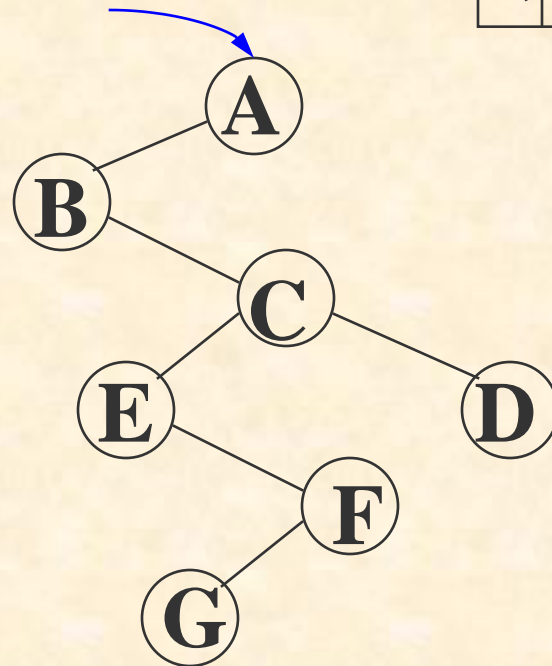
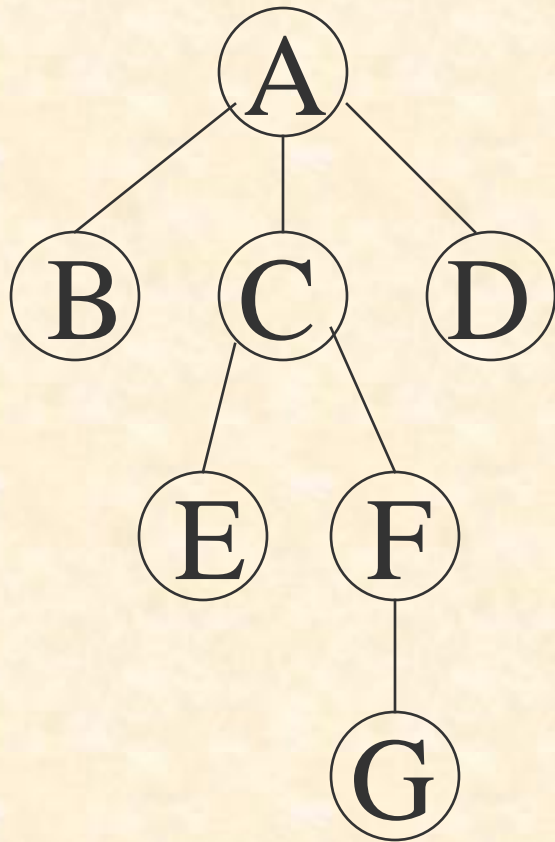
```
typedef struct {  
    Elem    data;  
    ChildPtr firstchild;  
           // 孩子链的头指针  
} CTBox;
```

树结构:

```
typedef struct {  
    CTBox  nodes[MAX_TREE_SIZE];  
  
    int    n, r;  
        // 结点数和根结点的位置  
  
} CTree;
```



三、树的二叉链表(孩子-兄弟) 存储表示法



C语言的类型描述:

结点结构:

firstchild	data	nextsibling
-------------------	-------------	--------------------

```
typedef struct CSNode{
```

```
    Elem      data;
```

```
    struct CSNode
```

```
        *firstchild, *nextsibling;
```

```
} CSNode, *CSTree;
```



森林和二叉树的对应关系

设森林

$$F = (T_1, T_2, \dots, T_n);$$

$$T_1 = (\text{root}, t_{11}, t_{12}, \dots, t_{1m});$$

二叉树

$$B = (\text{LBT}, \text{Node}(\text{root}), \text{RBT});$$

由森林转换成二叉树的转换规则为：

若 $F = \Phi$ ，则 $B = \Phi$ ；

否则，

由 $ROOT(T_1)$ 对应得到 $Node(root)$ ；

由 $(t_{11}, t_{12}, \dots, t_{1m})$ 对应得到 LBT ；

由 (T_2, T_3, \dots, T_n) 对应得到 RBT 。

由二叉树转换为森林的转换规则为：

若 $\mathbf{B} = \Phi$ ， 则 $\mathbf{F} = \Phi$ ；

否则，

由 $\mathbf{Node}(\mathbf{root})$ 对应得到 $\mathbf{ROOT}(T_1)$ ；

由 \mathbf{LBT} 对应得到 $(t_{11}, t_{12}, \dots, t_{1m})$ ；

由 \mathbf{RBT} 对应得到 (T_2, T_3, \dots, T_n) 。

由此，树的各种操作均可对应二叉树的操作来完成。

应当注意的是，和树对应的二叉树，其左、右子树的概念已改变为：**左是孩子，右是兄弟。**



6.7

树和森林的遍历



一、树的遍历

二、森林的遍历

三、树的遍历的应用



树的遍历可有三条搜索路径:

先根(次序)遍历:

若树不空, 则先访问根结点, 然后依次先根遍历各棵子树。

后根(次序)遍历:

若树不空, 则先依次后根遍历各棵子树, 然后访问根结点。

按层次遍历:

若树不空, 则自上而下自左至右访问树中每个结点。

先根遍历时顶点的访问次序:

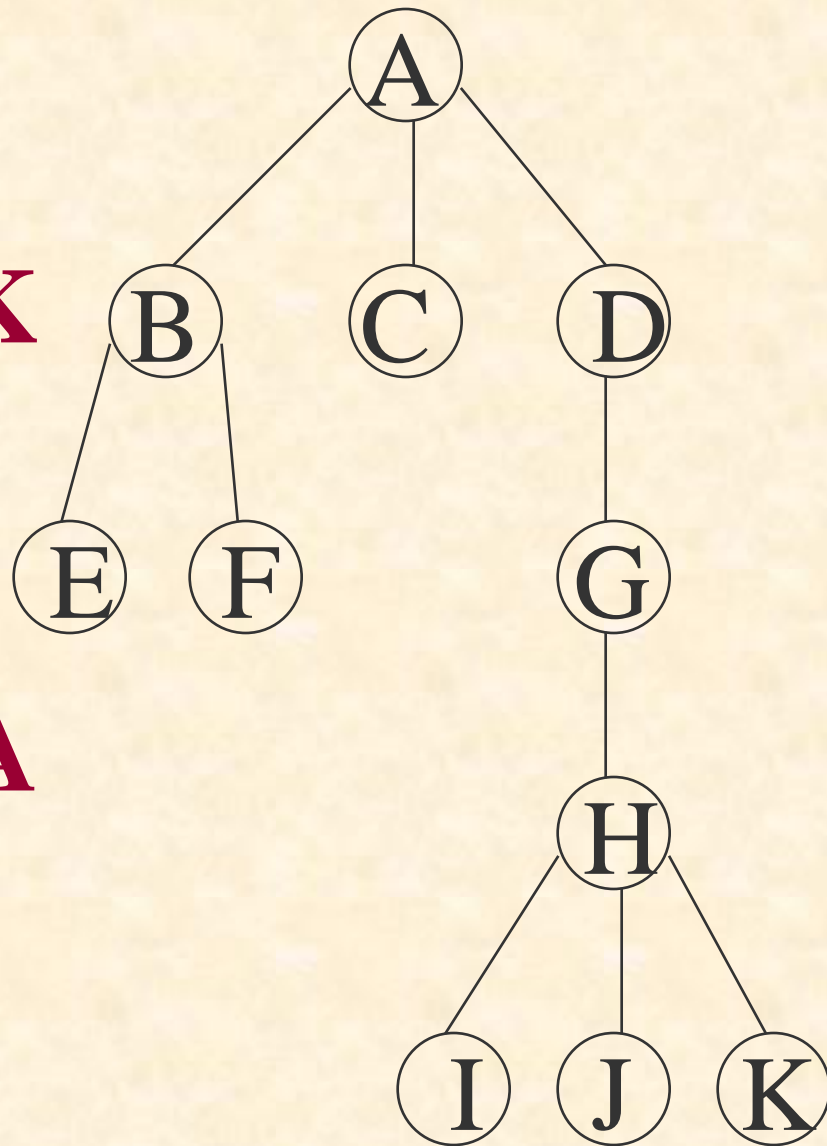
A B E F C D G H I J K

后根遍历时顶点的访问次序:

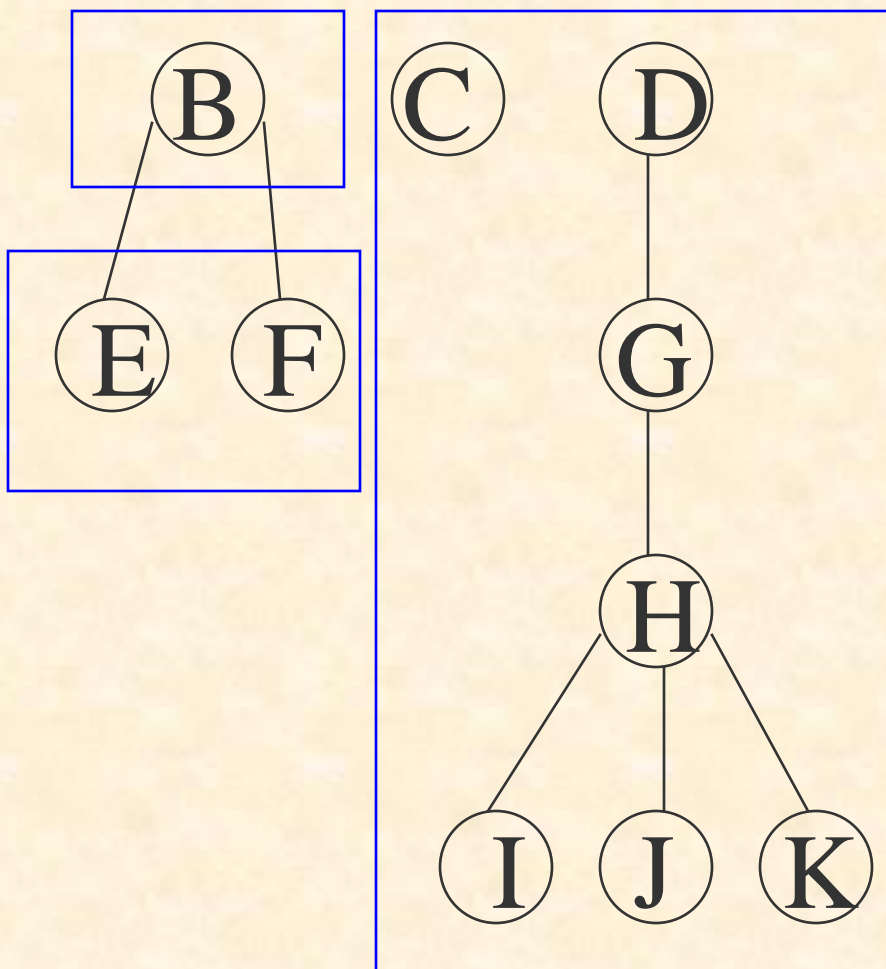
E F B C I J K H G D A

层次遍历时顶点的访问次序:

A B C D E F G H I J K



森林由三部分构成：



1. 森林中第一棵树的根结点；

2. 森林中第一棵树的子树森林；

3. 森林中其它树构成的森林。

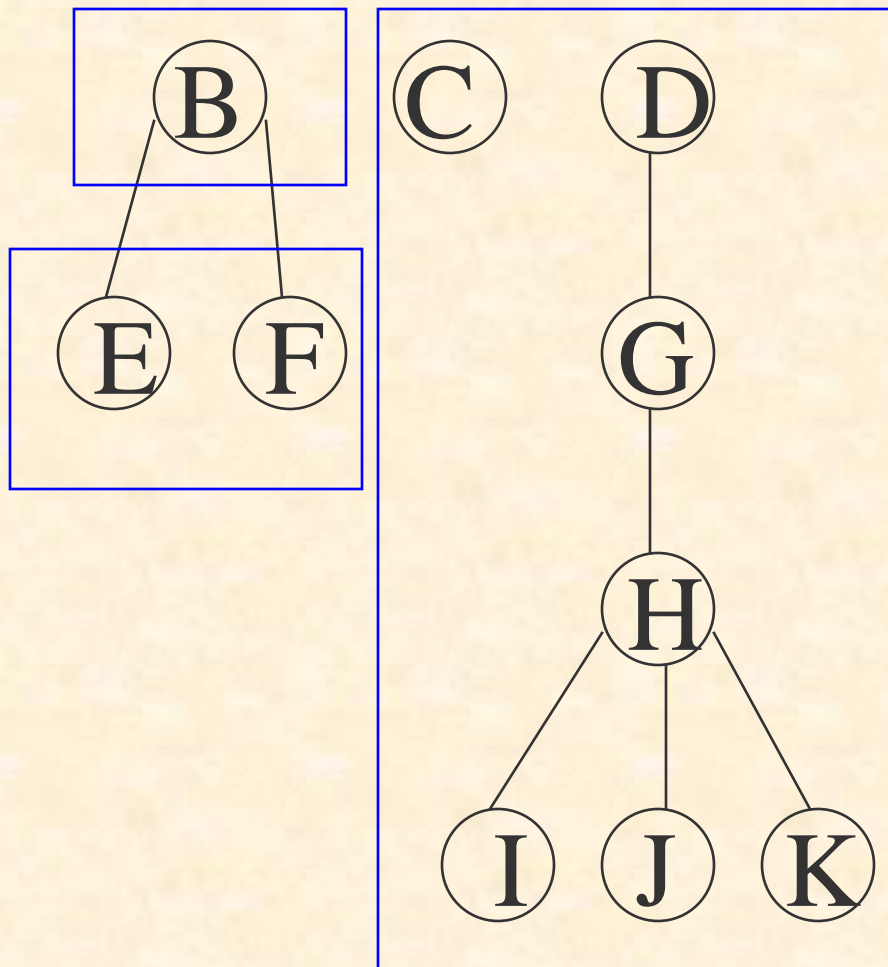


森林的遍历

1. 先序遍历 若森林不空，则
访问森林中第一棵树的根结点；
先序遍历森林中第一棵树的子树森林；
先序遍历森林中(除第一棵树之外)其
余树构成的森林。

即：依次从左至右对森林中的每一棵树进行先根遍历。





先根： BEF C DGH IJK

2. 中序遍历

若森林不空，则

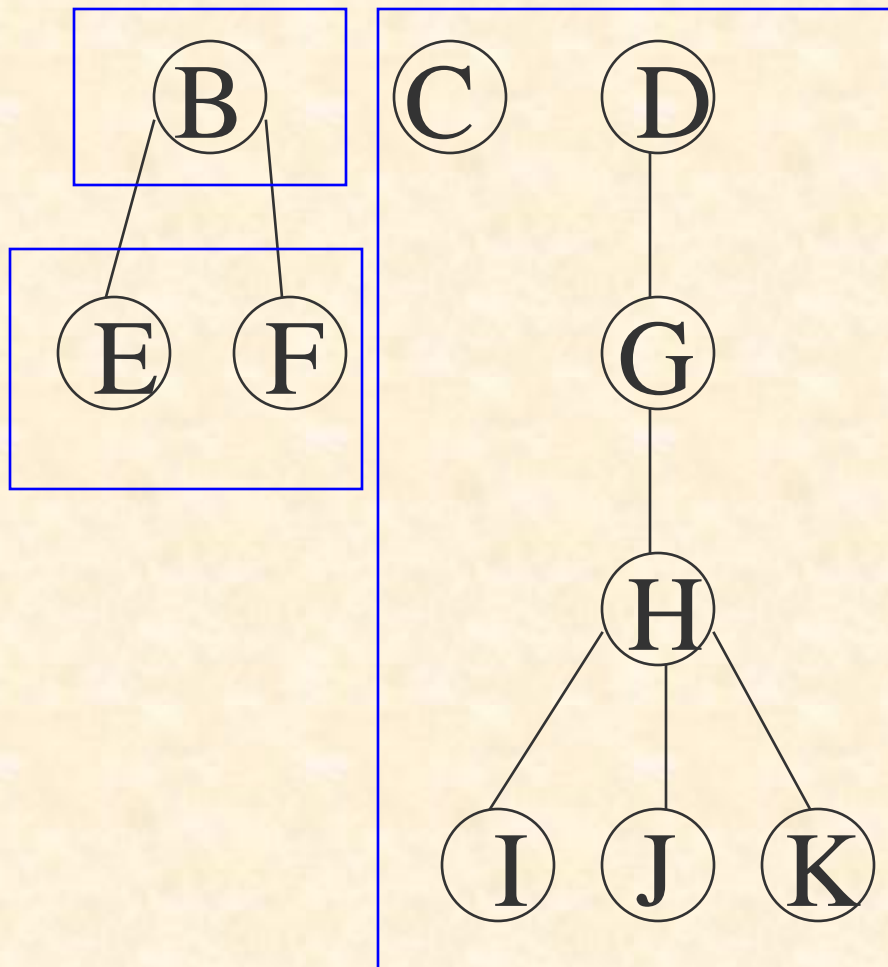
中序遍历森林中第一棵树的子树森林；

访问森林中第一棵树的根结点；

中序遍历森林中(除第一棵树之外)其余树构成的森林。

即：依次从左至右对森林中的每一棵树进行后根遍历。





中根： EFBC IJHGD

树的遍历和二叉树遍历 的对应关系？

树

森林

二叉树

先根遍历

先序遍历

先序遍历

后根遍历

中序遍历

中序遍历



设树的存储结构为孩子兄弟链表

```
typedef struct CSNode{  
    Elem      data;  
    struct CSNode *firstchild, *nextsibling;  
} CSNode, *CSTree;
```

一、求树的深度

二、输出树中所有从根到叶子的路径

三、建树的存储结构



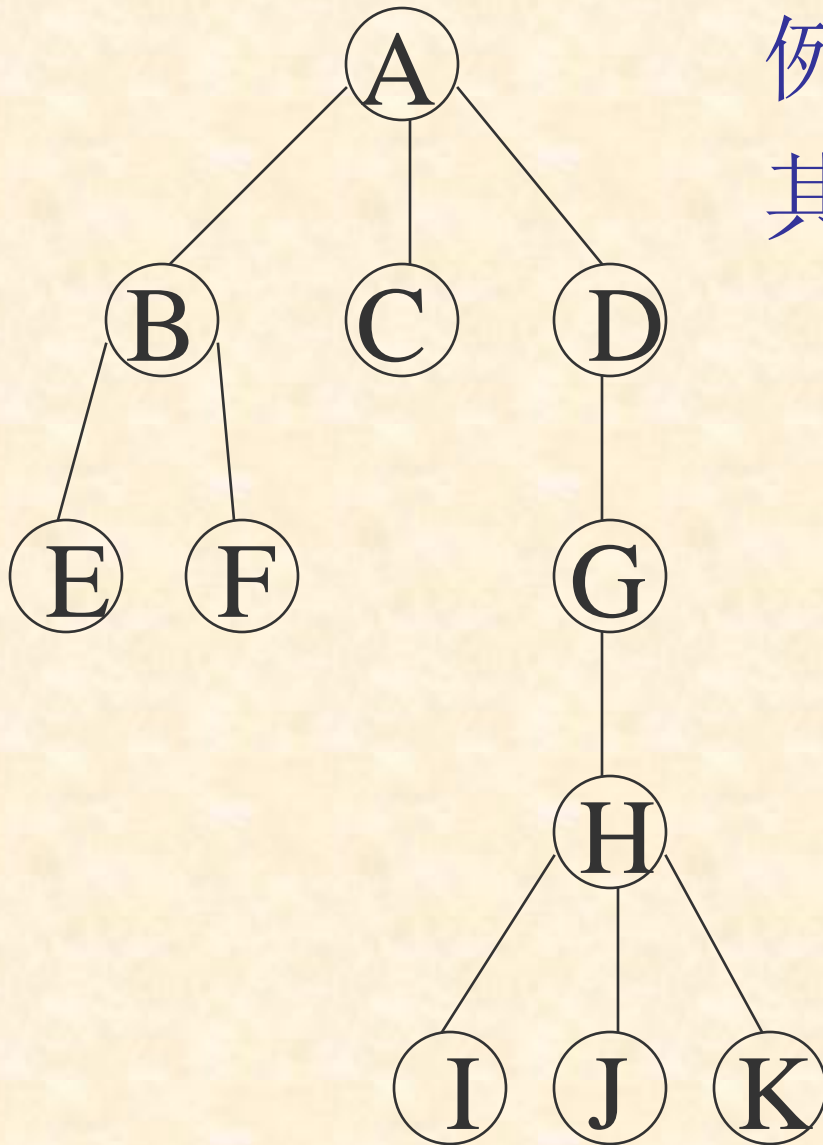
一、求树的深度的算法:

```
int TreeDepth(CSTree T) {  
    if(!T) return 0;  
    else {  
        h1 = TreeDepth( T->firstchild );  
        h2 = TreeDepth( T->nextsibling);  
        return(max(h1+1, h2));  
    }  
} // TreeDepth
```



二、输出树中所有从根到叶子的路径的算法：

例如：对左图所示的树，
其输出结果应为：



A B E

A B F

A C

A D G H I

A D G H J

A D G H K


```
void AllPath( Bitree T, Stack& S ) {
```

```
    // 输出二叉树上从根到所有叶子结点的路径
```

```
if (T) {
```

```
    Push( S, T->data );
```

```
    if (!T->Lchild && !T->Rchild ) PrintStack(S);
```

```
    else { AllPath( T->Lchild, S );
```

```
        AllPath( T->Rchild, S );
```

```
    }
```

```
    Pop(S); //if也好， else也好， 结束后都要pop
```

```
    } // if(T)
```

```
} // AllPath
```

```
void OutPath( Bitree T, Stack& S ) {
```

```
// 输出森林中所有从根到叶的路径
```

```
while ( !T ) {
```

```
    Push(S, T->data );
```

```
    if ( !T->firstchild ) Printstack(s);
```

```
    else OutPath( T->firstchild, s );
```

```
    Pop(S);
```

```
    T = T->nextsibling; // 往右走前要出栈，因为右链域是兄弟
```

```
} // while
```

```
} // OutPath
```

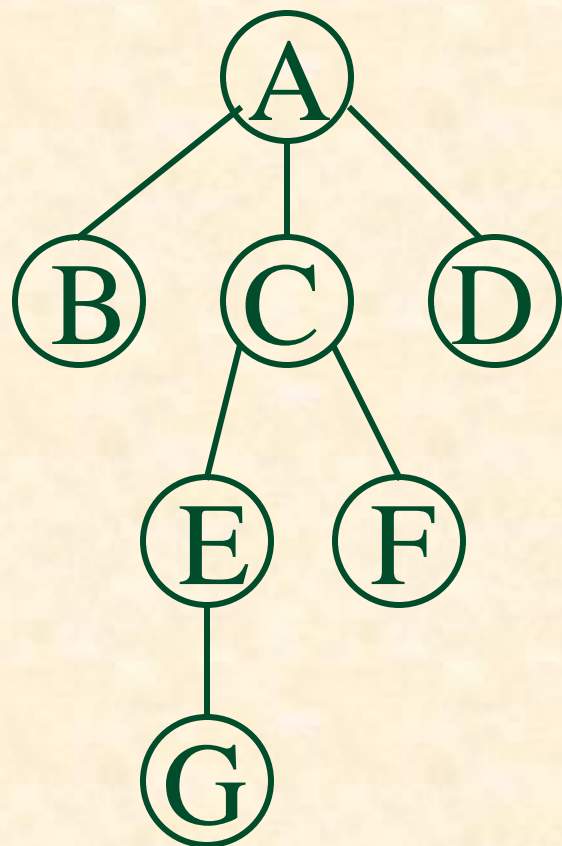


三、建树的存储结构的算法:

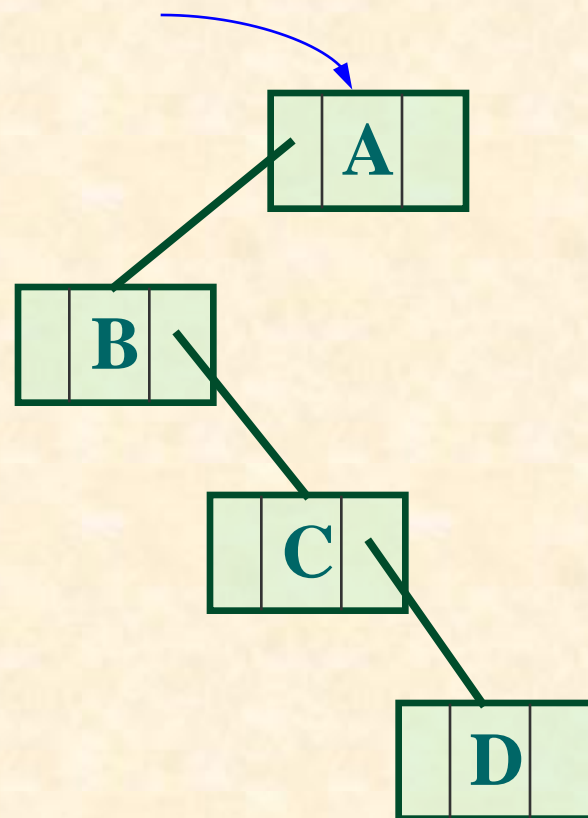
和二叉树类似，不同的定义相应有不同的算法。

假设以二元组(**F**, **C**)的形式自上而下、自左而右依次输入树的各边，建立树的**孩子-兄弟链表**。

例如： 对下列所示树的输入序列应为：



(‘#’, ‘A’)
(‘A’, ‘B’)
(‘A’, ‘C’)
(‘A’, ‘D’)
(‘C’, ‘E’)
(‘C’, ‘F’)
(‘E’, ‘G’)



可见，算法中需要一个队列保存已建好的结点的指针。

```
void CreatTree( CSTree &T ) {  
    T = NULL;  
    for( scanf(&fa, &ch); ch!='# ';  
        scanf(&fa, &ch);) {  
        p = GetTreeNode(ch); // 创建结点  
        EnQueue(Q, p);      // 指针入队列  
        if (fa == '# ') T = p; // 所建为根结点  
        else { ... .. } // 非根结点的情况  
    } // for  
} // CreateTree
```



```
GetHead(Q,s); // 取队列头元素(指针值)
while (s->data != fa ) { // 查询双亲结点
    DeQueue(Q,s); GetHead(Q,s);
}
if (!(s->firstchild))
    { s->firstchild = p; r = p; }
    // 链接第一个孩子结点
else { r->nextsibling = p; r = p; }
    // 链接其它孩子结点
```



6.8 哈夫曼树与 哈夫曼编码

§ 最优树的定义

§ 如何构造最优树

§ 前缀编码

一、最优树的定义

◆ 结点的路径长度定义为：

从根结点到该结点的路径上
分支的数目。

◆ 树的路径长度定义为：

树中每个结点的路径长度之和。

◆ 树的带权路径长度定义为：

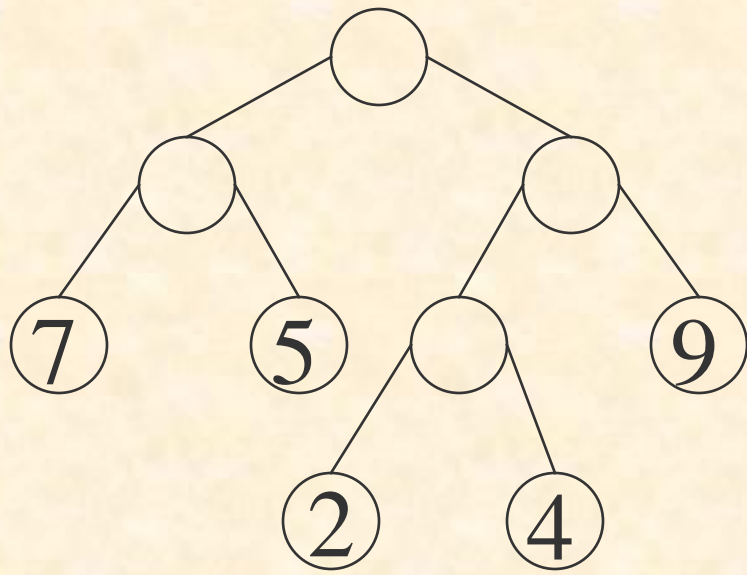
树中所有叶子结点的带权路径长度之和

$$WPL(T) = \sum w_k l_k \text{ (对所有叶子结点)}。$$

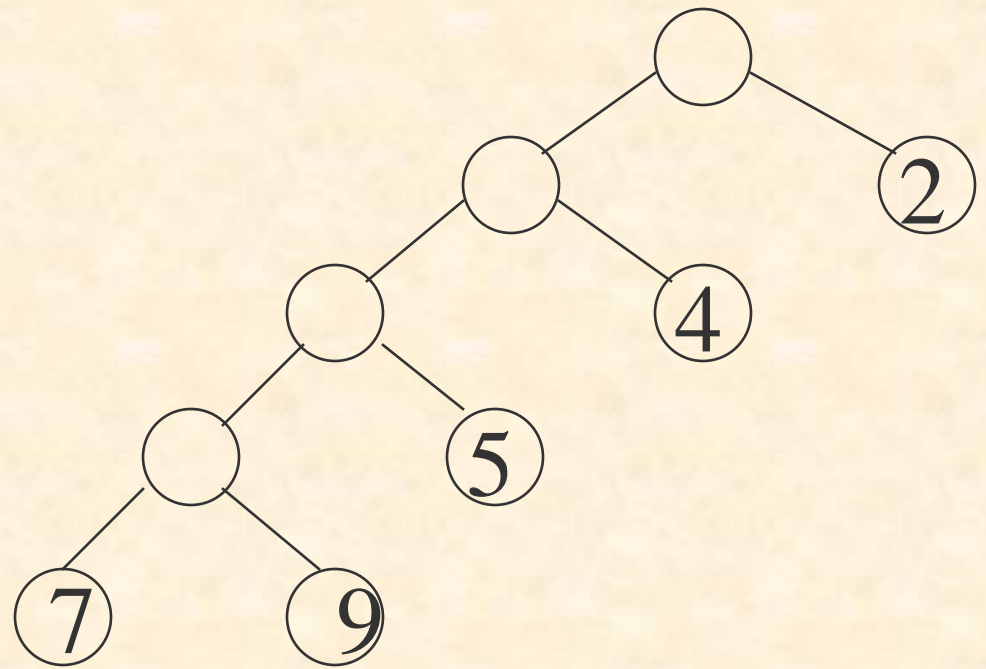
例如： 

在所有含 n 个叶子结点、并带相同权值的 m 叉树中，必存在一棵其带权路径长度取最小值的树，称为“最优树”。





$$\begin{aligned}
 \text{WPL}(T) &= \\
 &7 \times 2 + 5 \times 2 + 2 \times 3 + \\
 &4 \times 3 + 9 \times 2 \\
 &= 60
 \end{aligned}$$



$$\begin{aligned}
 \text{WPL}(T) &= \\
 &7 \times 4 + 9 \times 4 + 5 \times 3 + \\
 &4 \times 2 + 2 \times 1 \\
 &= 89
 \end{aligned}$$



二、如何构造最优树

(赫夫曼算法) 以二叉树为例:

(1) 根据给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$,
构造 n 棵二叉树的集合

$$F = \{T_1, T_2, \dots, T_n\},$$

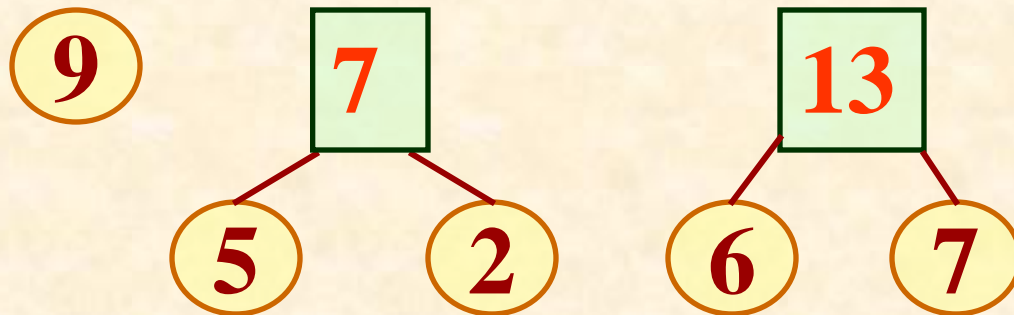
其中每棵二叉树中均只含一个带权值为 w_i 的根结点, 其左、右子树为空树;

(2) 在 F 中选取其根结点的权值为最小的两棵二叉树，分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和；

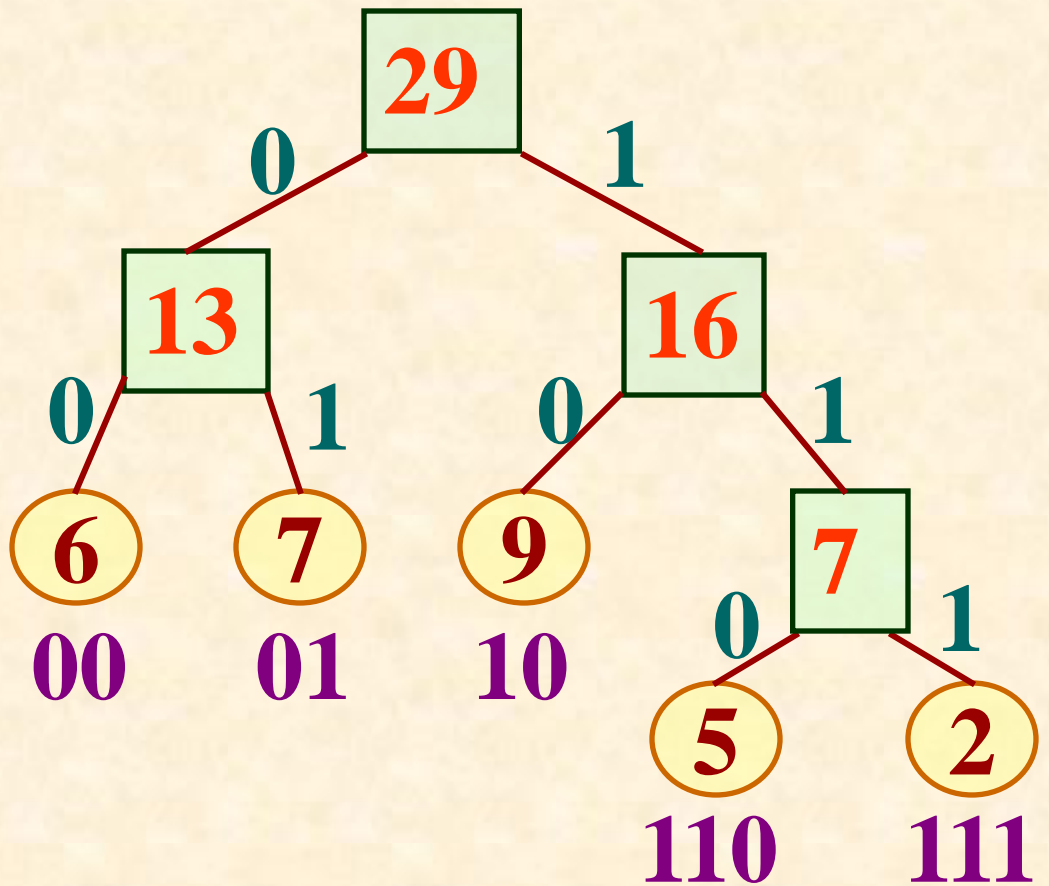
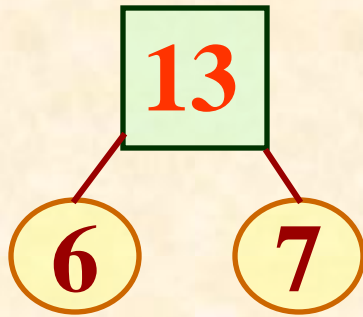
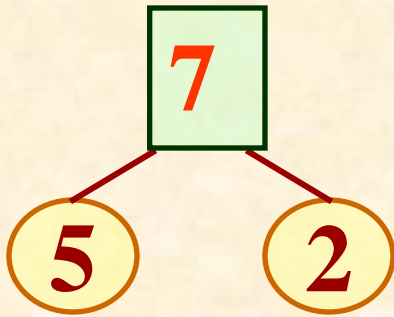
(3) 从 F 中删去这两棵树，同时加入刚生成的新树；

(4) 重复 (2) 和 (3) 两步，直至 F 中只含一棵树为止。

例如：已知权值 $W=\{ 5, 6, 2, 9, 7 \}$



9



三、前缀编码

指的是，任何一个字符的编码都不是同一字符集中另一个字符的编码的前缀。

利用赫夫曼树可以构造一种不等长的二进制编码，并且构造所得的赫夫曼编码是一种最优前缀编码，即使所传电文的总长度最短。



1. 熟练掌握二叉树的结构特性，了解相应的证明方法。

2. 熟悉二叉树的各种存储结构的特点及适用范围。

3. 遍历二叉树是二叉树各种操作的基础。实现二叉树遍历的具体算法与所采用的存储结构有关。掌握各种遍历策略的递归算法，灵活运用遍历算法实现二叉树的其它操作。层次遍历是按另一种搜索策略进行的遍历。

4. 理解二叉树**线索化**的实质是建立结点与其在相应序列中的前驱或后继之间的直接联系，熟练掌握二叉树的**线索化过程**以及在中序线索化树上找给定结点的前驱和后继的方法。二叉树的**线索化过程**是基于对二叉树进行**遍历**，而线索二叉树上的**线索**又为相应的遍历提供了方便。

5. 熟悉树的各种存储结构及其特点，掌握树和森林与二叉树的转换方法。建立存储结构是进行其它操作的前提，因此读者应掌握 1 至 2 种建立二叉树和树的存储结构的方法。

6. 学会编写实现树的各种操作的算法。

7. 了解最优树的特性，掌握建立最优树和哈夫曼编码的方法。