



# 第七章 二叉树的遍历

---



# 树的遍历

- 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**
  - 得到树中所有结点的一个线性排列
  - 将树的节点线性化（定义一个次序）

“**访问**”的含义可以很广，如：输出结点的信息等。



# 树的遍历方法

- 先序遍历

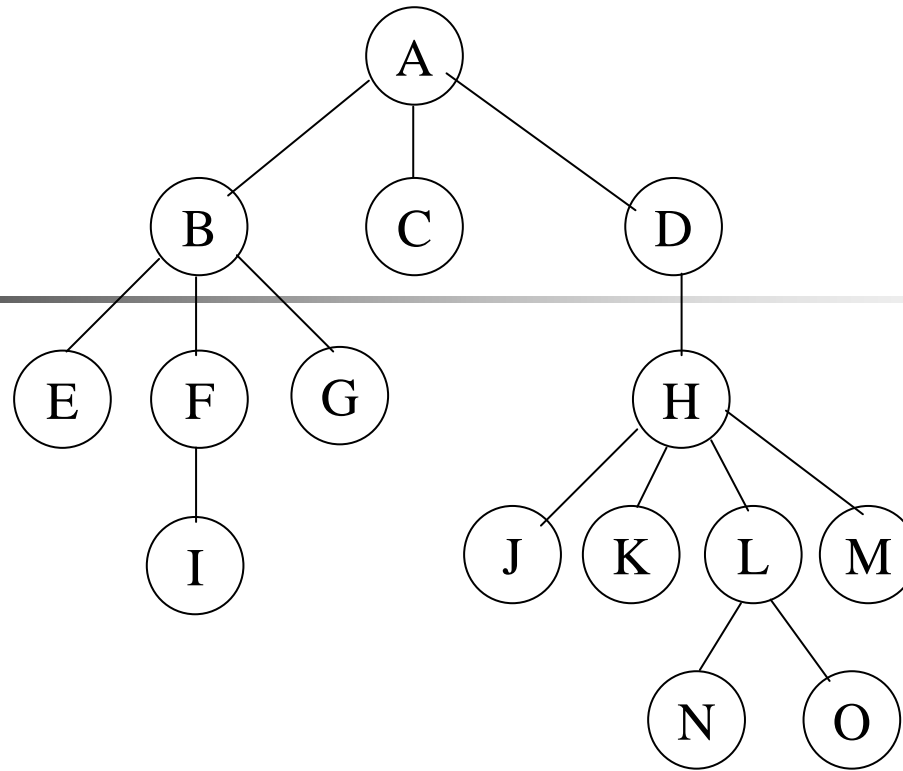
- 先访问树的根结点，然后依次先序遍历根的每棵子树

- 后序遍历

- 先依次后序遍历每棵子树，然后访问根结点

- 按层次遍历

- 先访问第一层上的结点，然后依次遍历第二层，……，第 $n$ 层的结点



先序遍历 : ABEFI GCDHJ KLNOM

后序遍历 : EIFGBCJKNOLMHDA

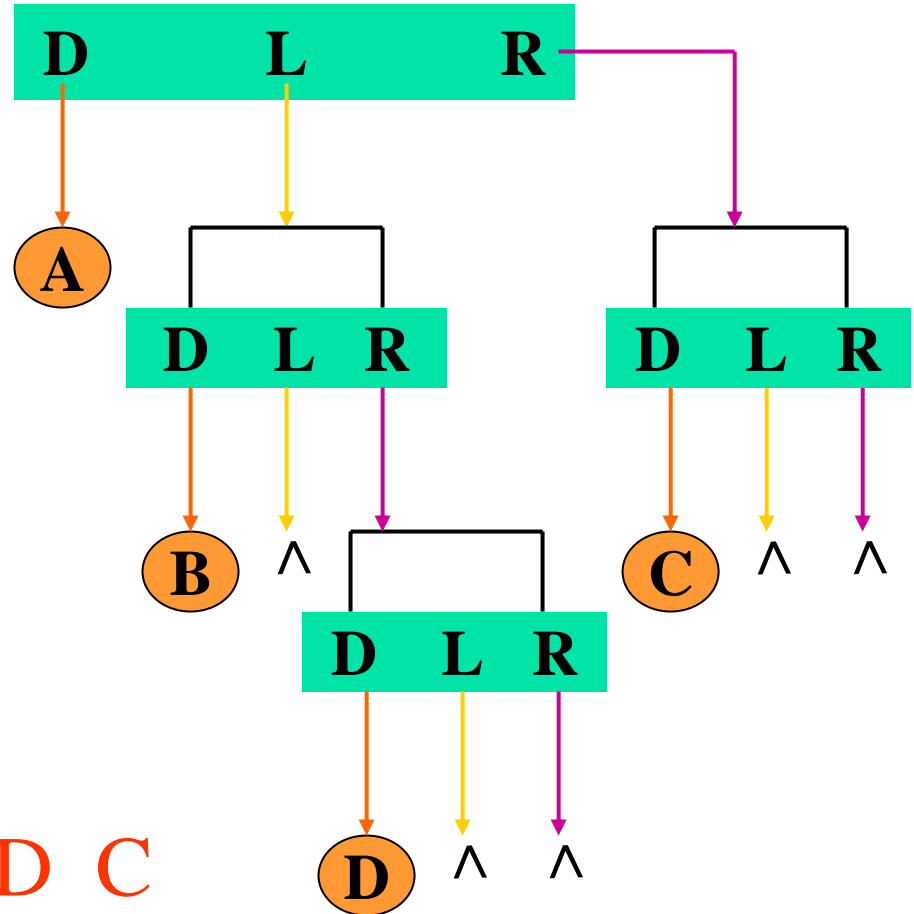
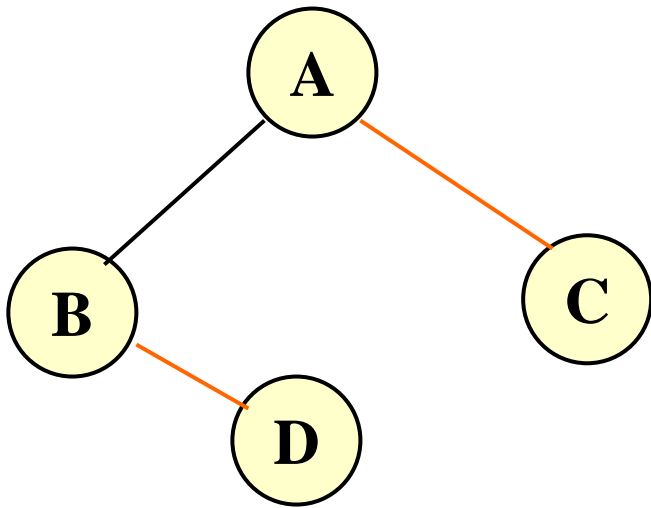
层次遍历 : ABCDEFGHIJ KLMNO



# 二叉树的遍历

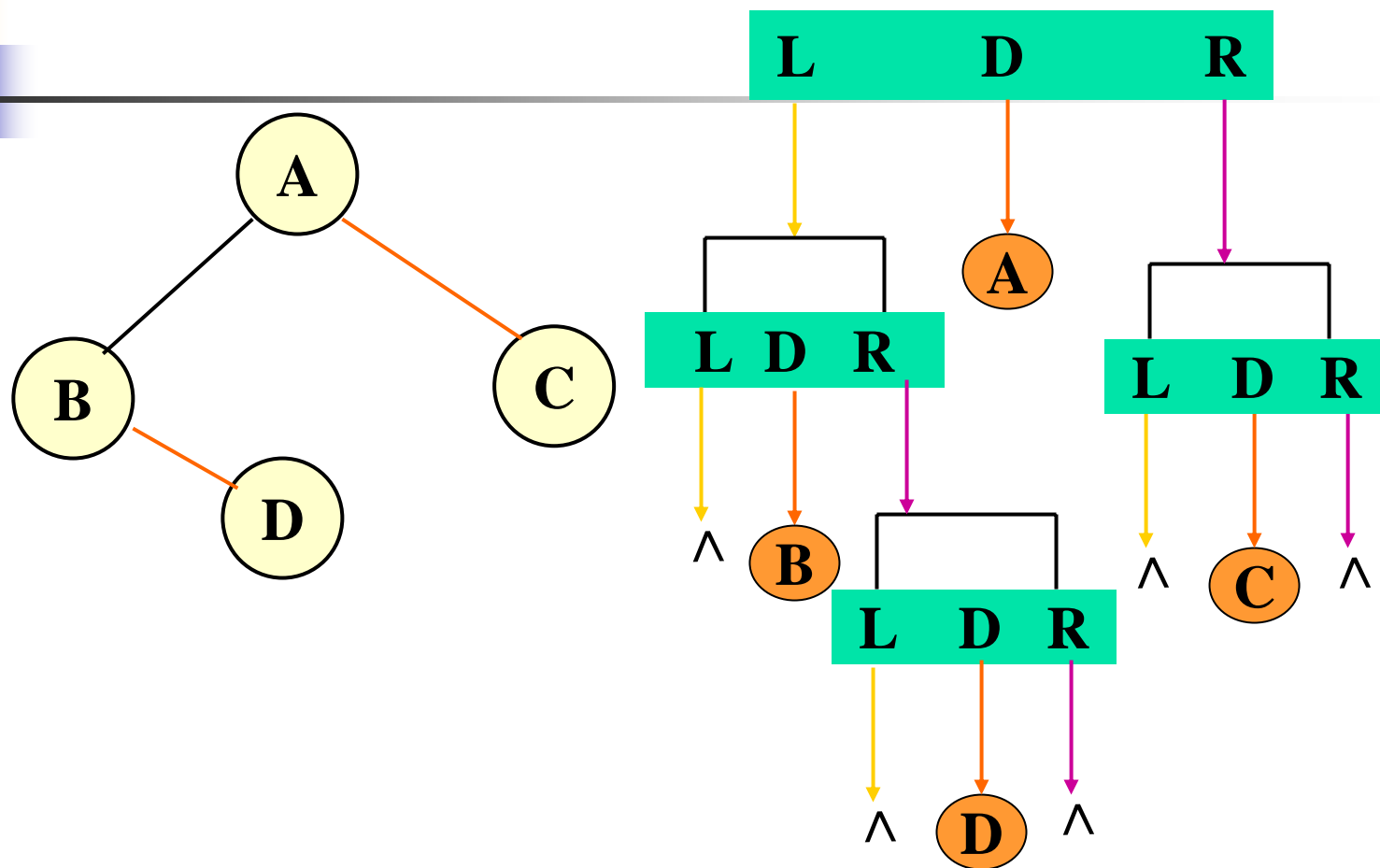
- 先序遍历
  - 先访问根结点，然后分别先序遍历左子树、右子树
- 中序遍历
  - 先中序遍历左子树，然后访问根结点，最后中序遍历右子树
- 后序遍历
  - 先后序遍历左、右子树，然后访问根结点
- 层次遍历
  - 从上到下、从左到右访问各结点

# 先序遍历



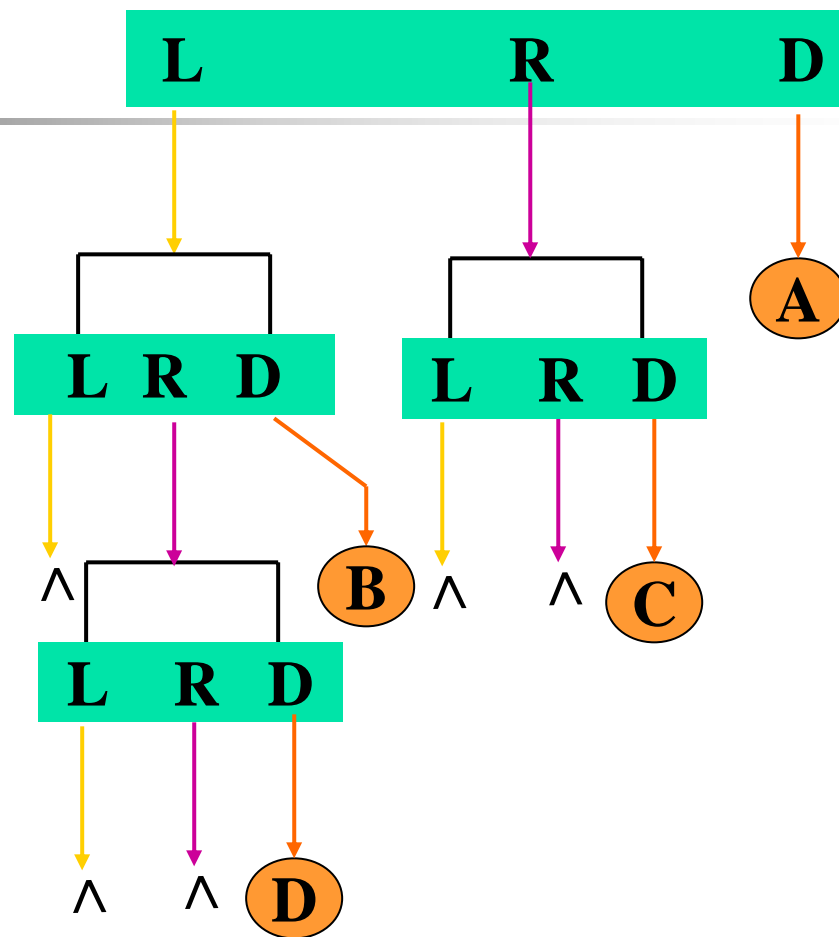
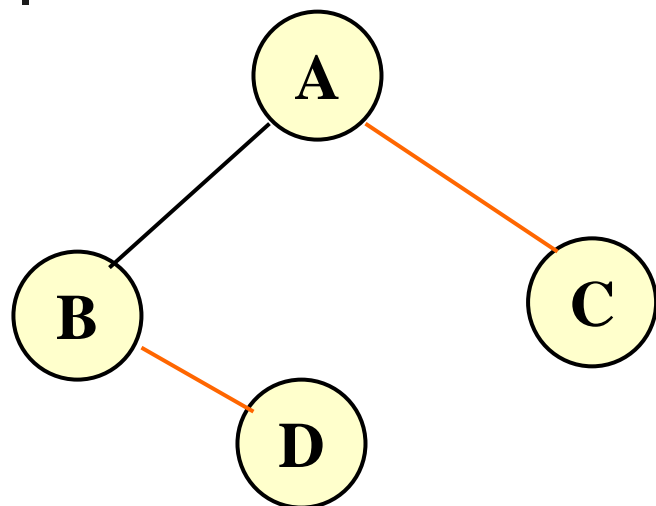
先序遍历序列：A B D C

# 中序遍历



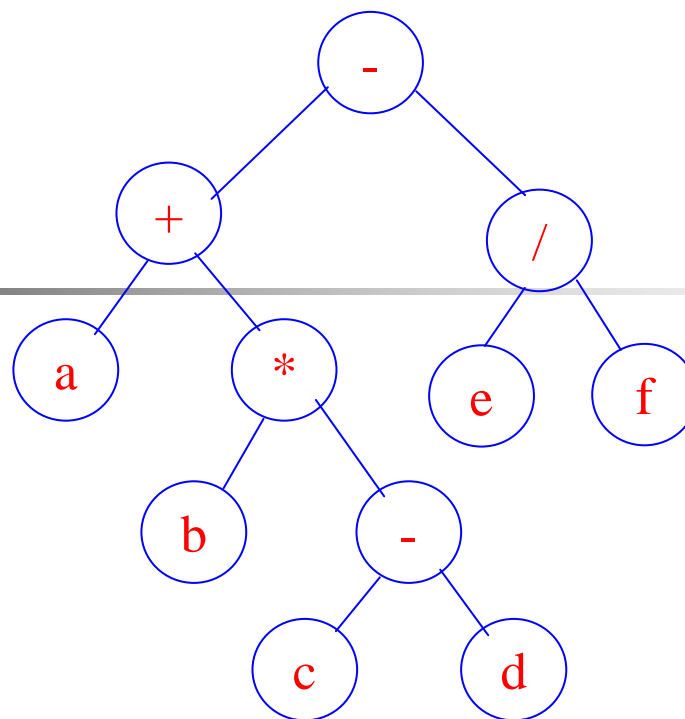
中序遍历序列：B D A C

# 后序遍历



后序遍历序列：D B C A





先序遍历： - + a \* b - c d / e f

中序遍历： a + b \* (c - d) - e / f

后序遍历： a b c d - \* + e f / -

层次遍历： - + / a \* e f b - c d



# 遍历算法

- 递归算法
- 非递归算法

先序遍历

中序遍历

后序遍历



# 先序遍历的递归算法

```
void PreOrder(CNode *pNode)
```

```
{
```

```
    if(pNode!=NULL)
```

```
    {
```

```
        printf("%lf\t",pNode->data);
```

```
        PreOrder(p->lchild);
```

```
        PreOrder(p->rchild);
```

```
    }
```

```
}
```

```
class CNode
```

```
{
```

```
    DataType    data;
```

```
    CNode       *lchild, *rchild;
```

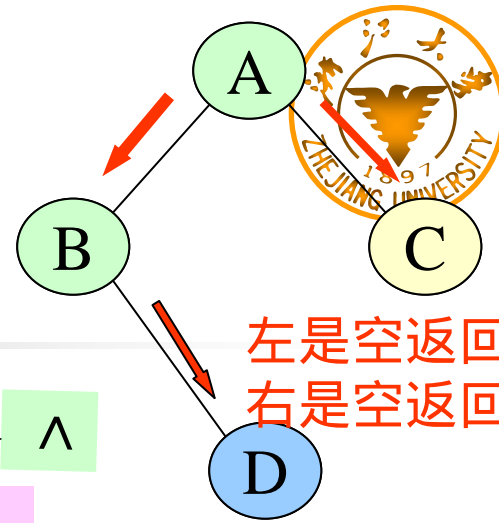
```
};
```



```
void PreOrder(CNode *pNode)
{
    if(pNode!=NULL)
    {
        printf("%lf\t",pNode->data);
        PreOrder(p->lchild);
        PreOrder(p->rchild);
    }
}
```

左是空返回

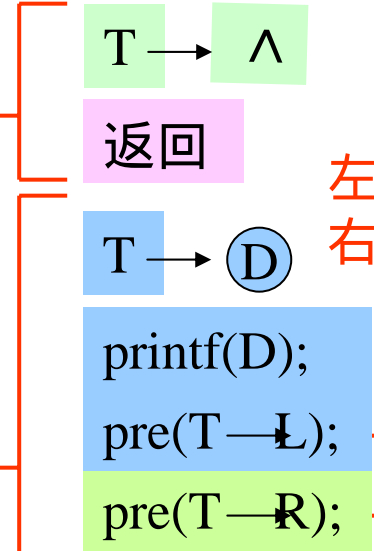
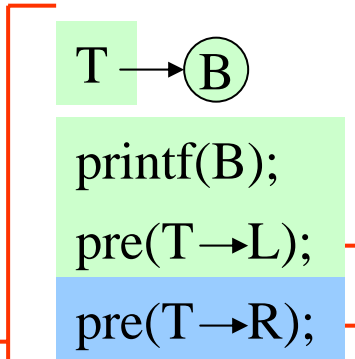
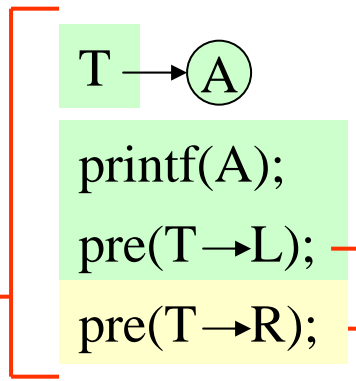
左是空返回  
右是空返回



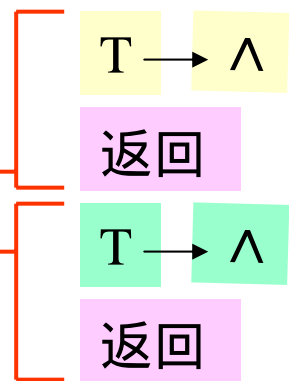
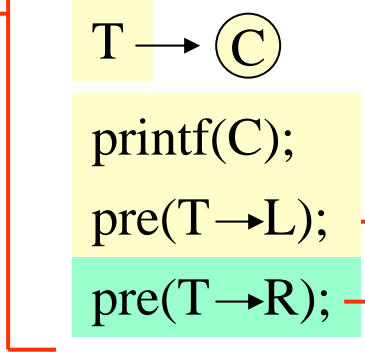
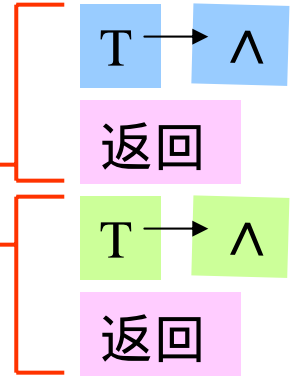
主程序

Pre( T )

先序序列 : A B D C



左是空返回  
右是空返回





## 中序遍历 算法

```
void InOrder(CNode *pNode)
{
    if(pNode!=NULL)
    {
        InOrder (pNode->lchild);
        printf("%lf\t",pNode->data);
        InOrder (pNode->rchild);
    }
}
```

## 后序遍历 算法

```
void PostOrder(CNode *pNode)
{
    if(pNode!=NULL)
    {
        PostOrder (pNode->lchild);
        PostOrder (pNode->rchild);
        printf("%lf\t",pNode->data);
    }
}
```



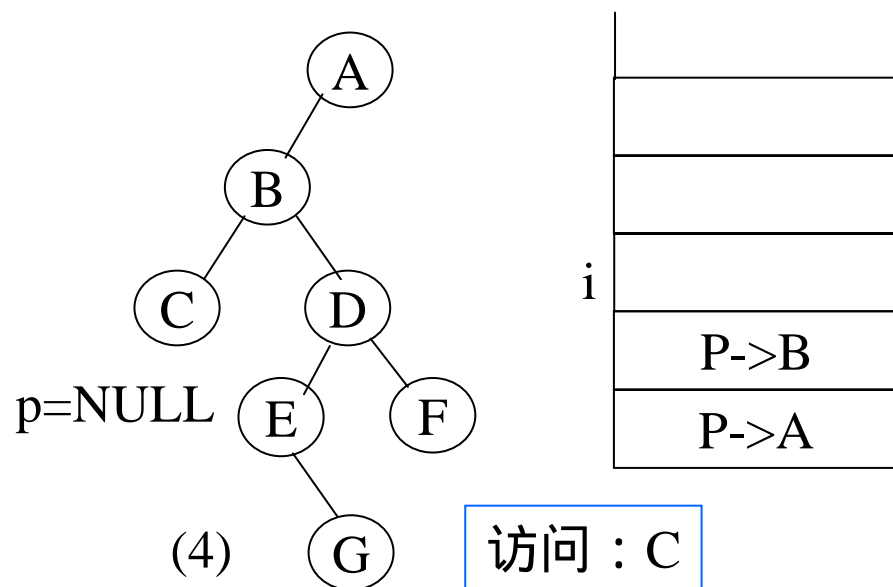
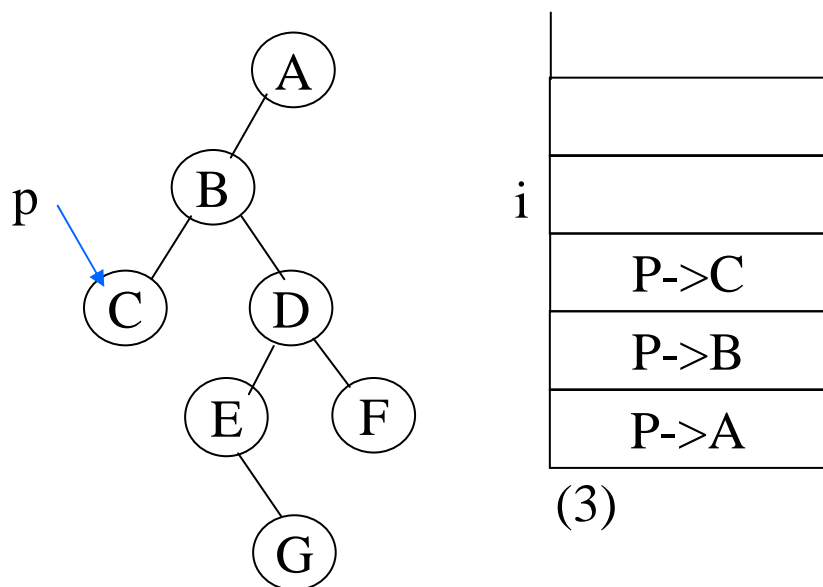
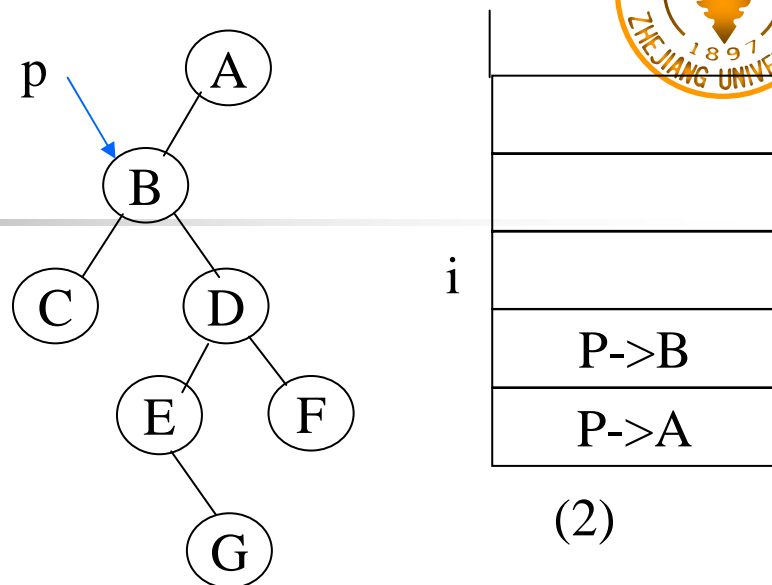
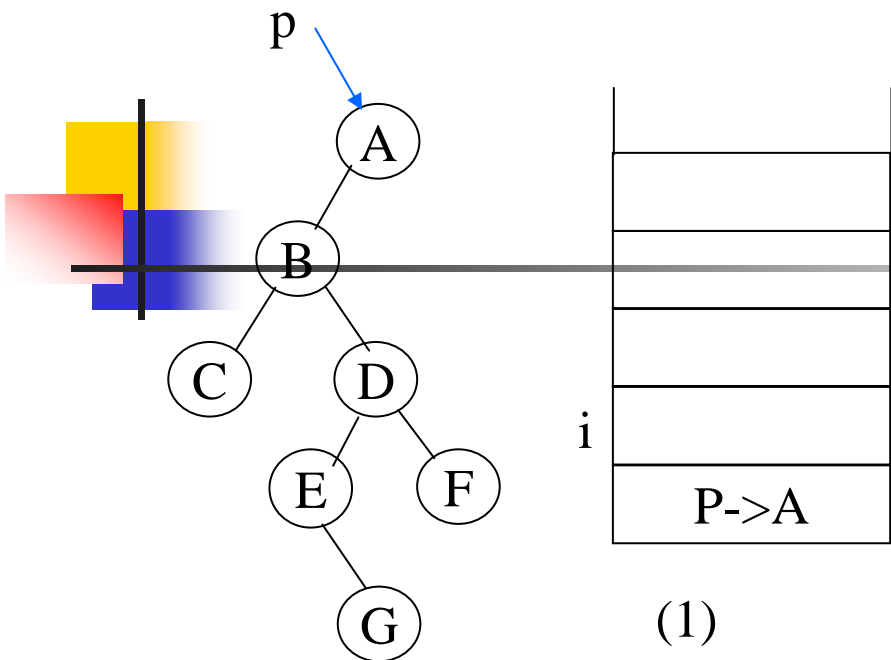
# 中序遍历的非递归算法

- 本质：利用堆栈

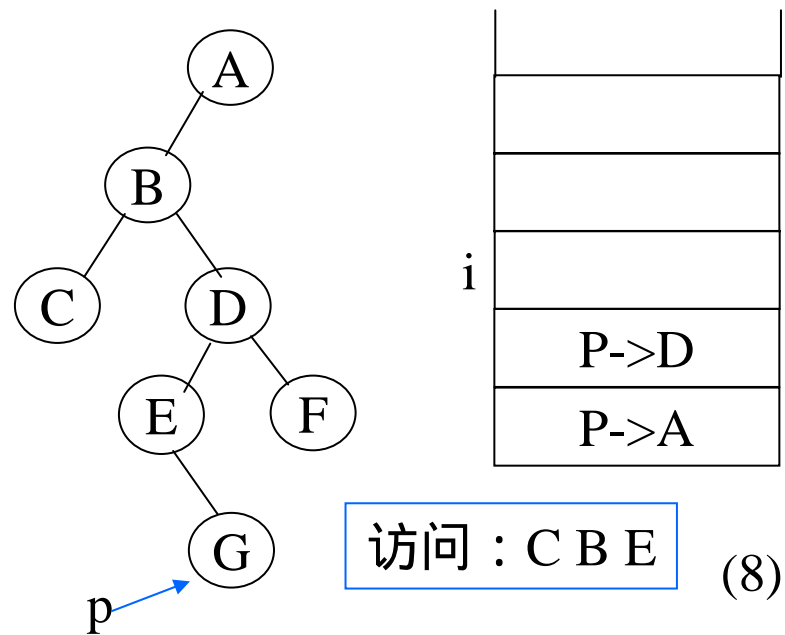
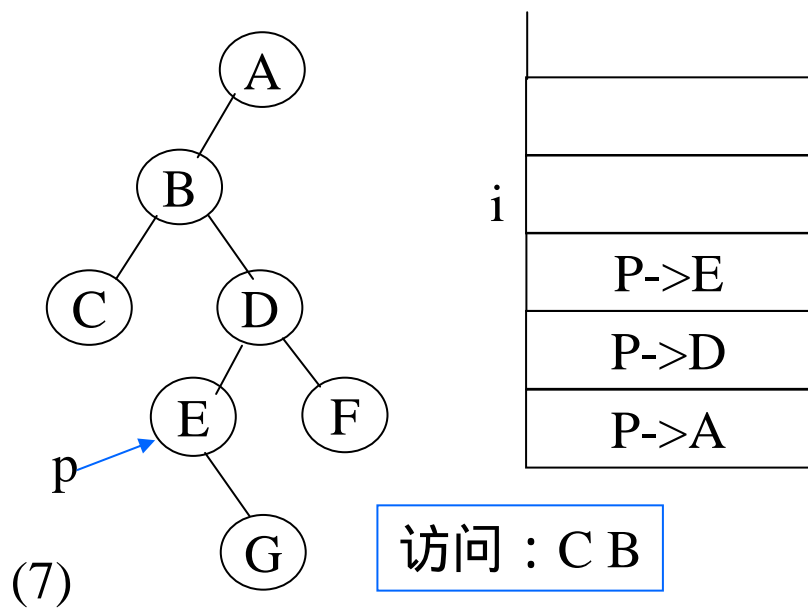
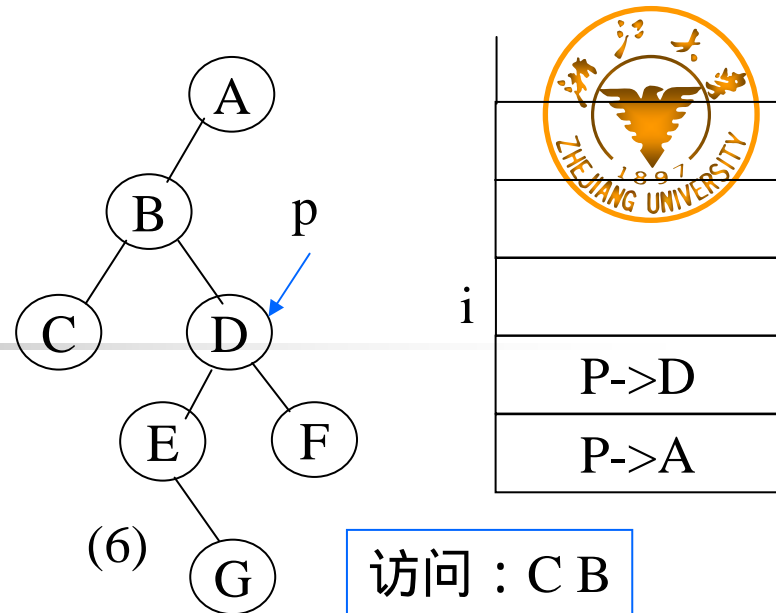
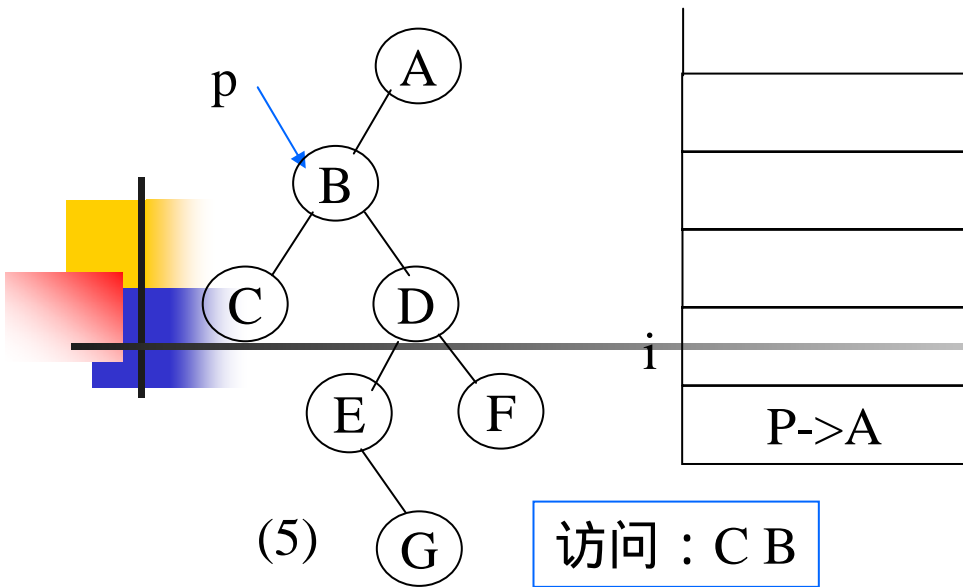
```
class CNode
{
    DataType    data;
    CNode      *lchild, *rchild;
};
```

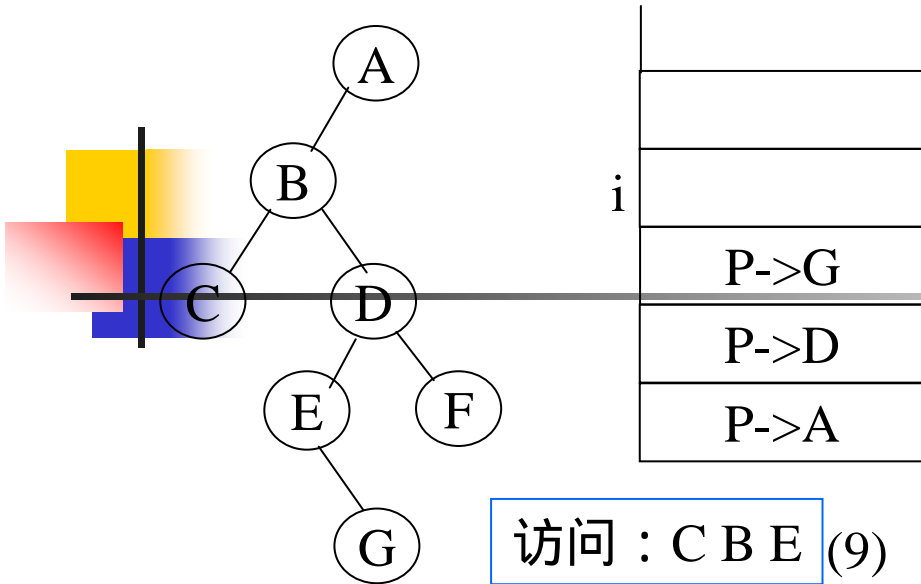


```
void InOrder(CNode *bt)
{  int i=0;
   CNode *p, *s[M];
   p=bt;
   do
   {  while(p!=NULL)
      {  s[i++]=p;      // push
         p=p->lchild;
      }
      if(i>0)
      {  p=s[--i];      // popup
         printf("%lf\t",p->data);
         p=p->rchild;
      }
   }while(i>0||p!=NULL);
}
```

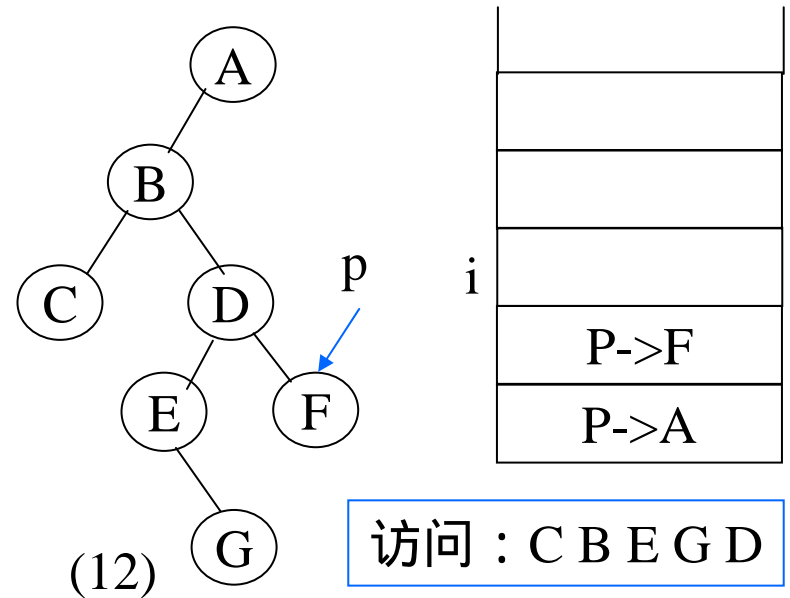
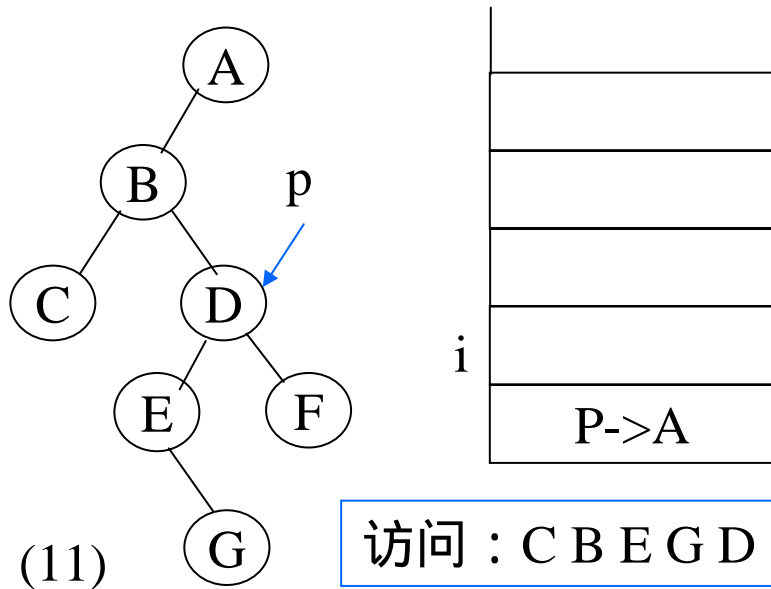
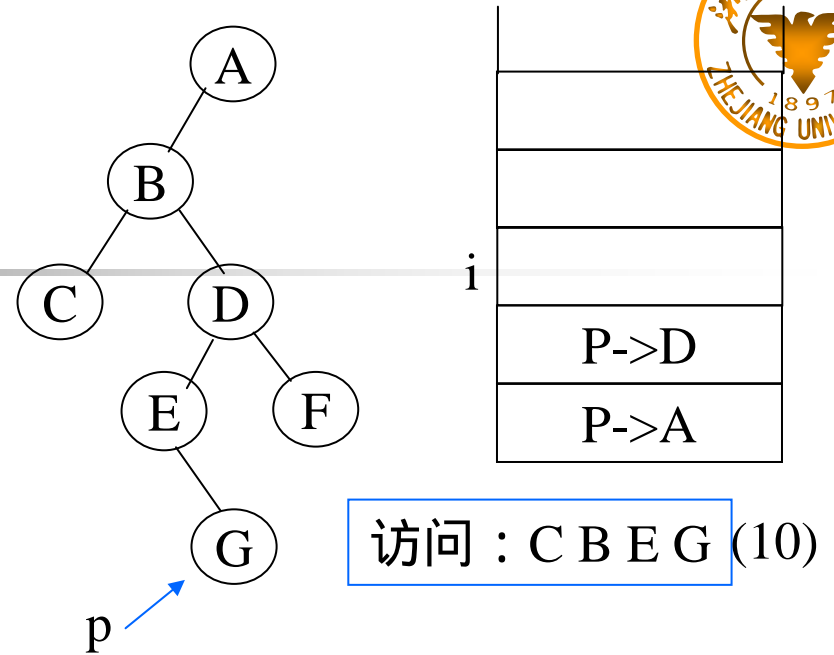


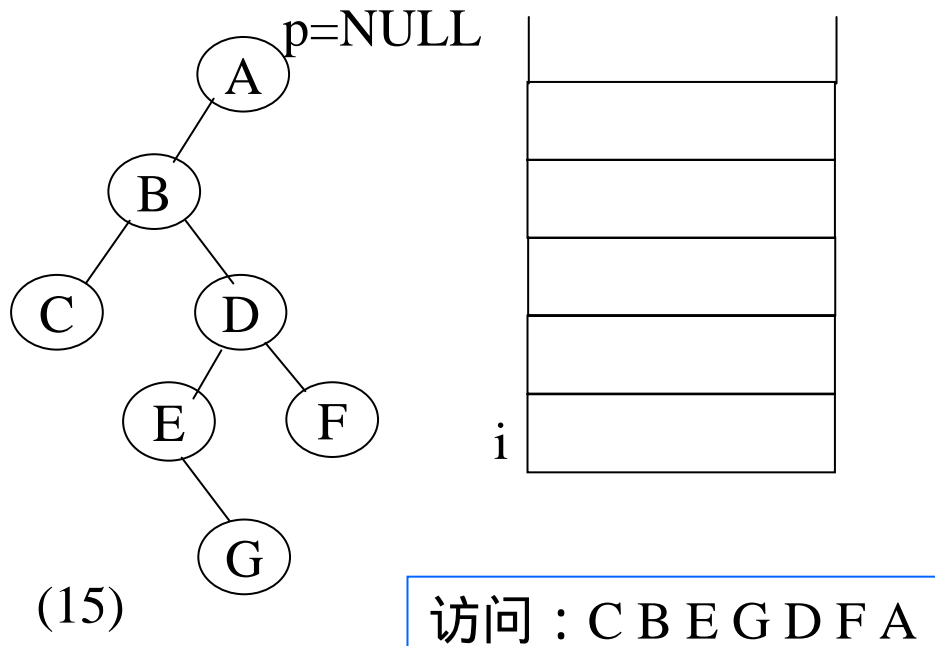
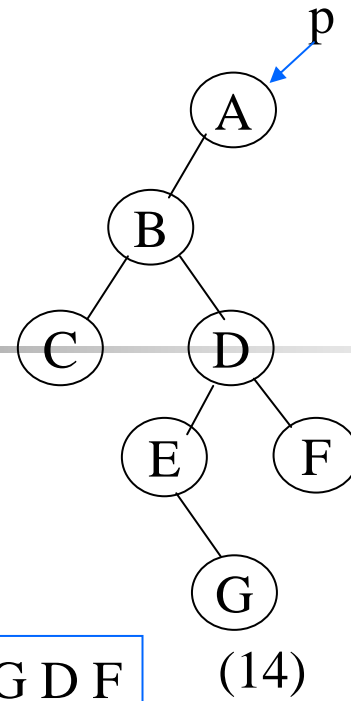
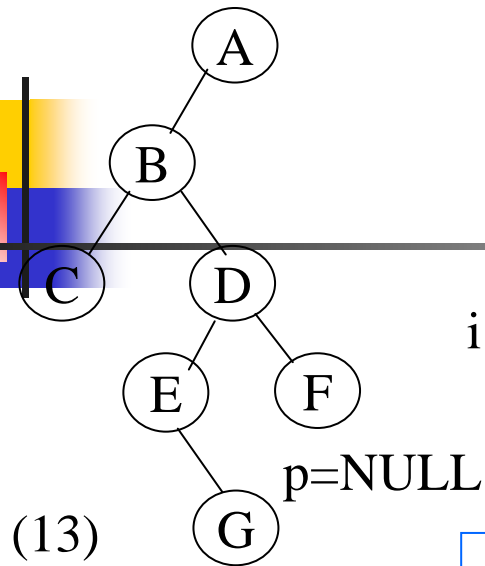






P=NULL







# 遍历算法的应用



# 遍历算法的应用举例

- 查询二叉树中某个结点
- 统计二叉树中叶子结点的个数
- 求二叉树的深度
- 复制二叉树
- 建立二叉树的存储结构



# 查询二叉树中某个结点

1. 在二叉树不空的前提下,和根结点的元素进行比较,若相等,则找到返回TRUE;
2. 否则在左子树中进行查找,若找到,则返回 TRUE;
3. 否则继续在右子树中进行查找,若找到,则返回 TRUE,否则返回 FALSE;

```
Status Preorder (BiTree T, ElemType x, BiTree &p) {  
    // 若二叉树中存在和 x 相同的元素 , 则 p 指向该结点并返回 OK,  
    // 否则返回 FALSE  
    if (T) {  
        if (T->data==x) { p = T; return TRUE;}  
        else {  
            if (Preorder(T->lchild, x, p)) return TRUE;  
            else return(Preorder(T->rchild, x, p)) ;  
        }  
    }  
    else return FALSE;  
}
```



# 统计叶子结点的个数

## ■ 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
```



```
int CountLeaf (BiTree T){
```

```
//返回指针T所指二叉树中所有叶子结点个数
```

```
if (!T ) return 0;
```

```
if (!T->lchild && !T->rchild) return 1;
```

```
else{
```

```
    m = CountLeaf( T->lchild);
```

```
    n = CountLeaf( T->rchild);
```

```
    return (m+n);
```

```
    } //else
```

```
} // CountLeaf
```



# 统计所有结点的个数

- 算法基本思想类似



```
int CountAll(BiTree T){
```

```
//返回指针T所指二叉树中所有结点个数
```

```
    if (!T ) return 0;
```

```
    if (!T->lchild && !T->rchild) return 1;
```

```
    else{
```

```
        m = CountAll ( T->lchild);
```

```
        n = CountAll ( T->rchild);
```

```
        return (m+n+1);
```

```
    } //else
```

```
} // CountAll
```





# 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;
}
```

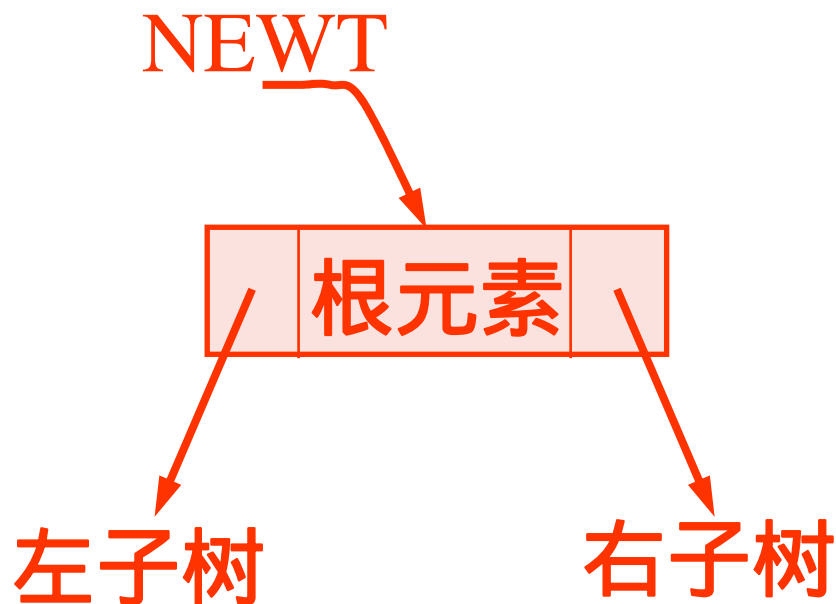
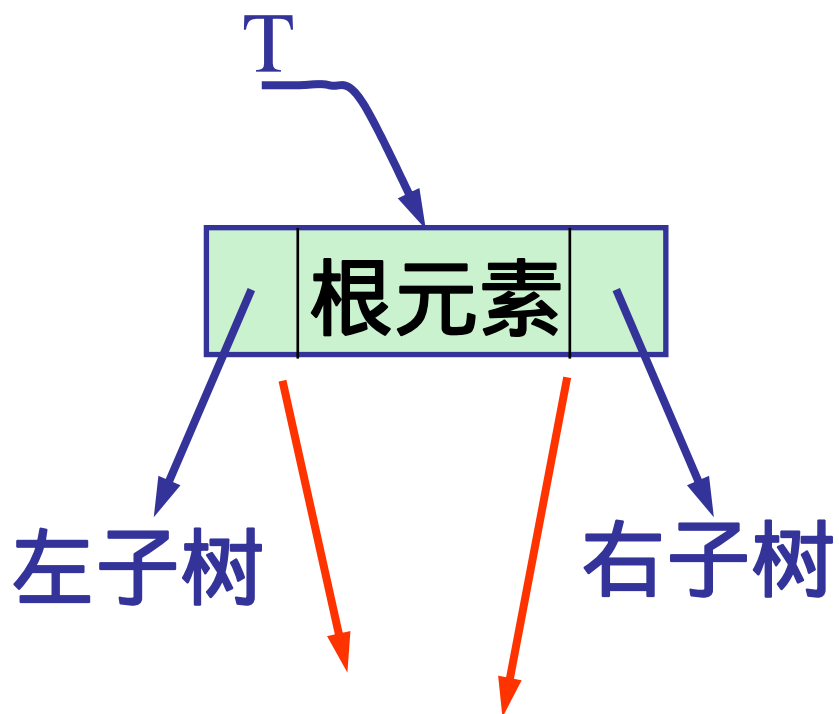
```
void Depth(BiTree T , int level, int &dval){  
    if ( T ) {  
        if (level>dval) dval = level;  
        Depth( T->lchild, level+1, dval );  
        Depth( T->rchild, level+1, dval );  
    }  
}
```

// 调用之前 level 的初值为 1。

// dval 的初值为 0.

# 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)

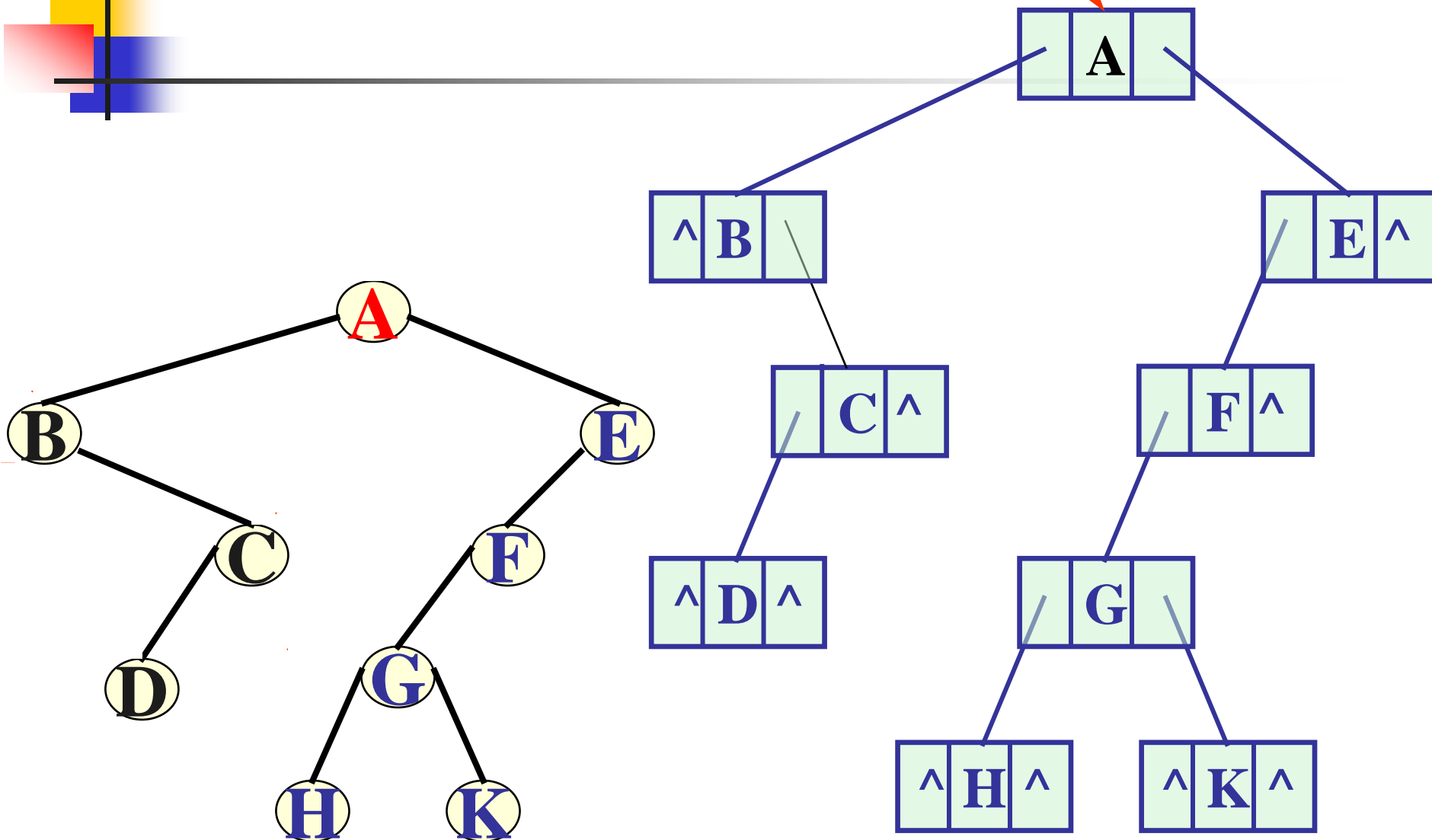
```
BiTNode *GetTreeNode(TElemType item,  
    BiTNode *lptr , BiTNode *rptr ){  
    if (!(T = new BiTNode))  
        exit(1);  
    T->data = item;  
    T->lchild = lptr;    T->rchild = rptr;  
    return T;  
}
```

```
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：



newT





# Q&A

---



# 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



# 从遍历序列恢复二叉树

## ■ 问题

- 能否根据一棵二叉树的某种遍历顺序来唯一地确定这棵二叉树？

## ■ 答案

- 一般不能！
- 一般需要2种遍历顺序



# 从遍历序列恢复二叉树

- 二叉树可被唯一确定，如果已知
  - 先序遍历序列和中序遍历序列
  - 或者：中序遍历序列和后序遍历序列
- 除了特殊情况，不能根据先序遍历序列和后序遍历序列来确定对应的二叉树

# 以字符串的形式 “根 左子树 右子树” 定义一棵二叉树

例如：

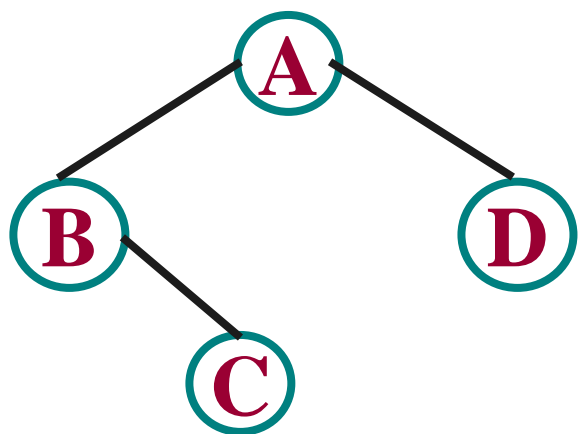
空树

以空白字符 “ $\square$ ” 表示

只含一个根结点的  
二叉树

Ⓐ

以字符串 “ $A\square\square$ ” 表示



以下列字符串表示

$A(\overline{B(\underline{\square}, \underline{C(\underline{\square}, \underline{\square})})}, \overline{D(\underline{\square}, \underline{\square})})$





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

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

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

```
else {
```

```
if (!(T = new BiTNode))
```

```
exit(OVERFLOW);
```

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

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

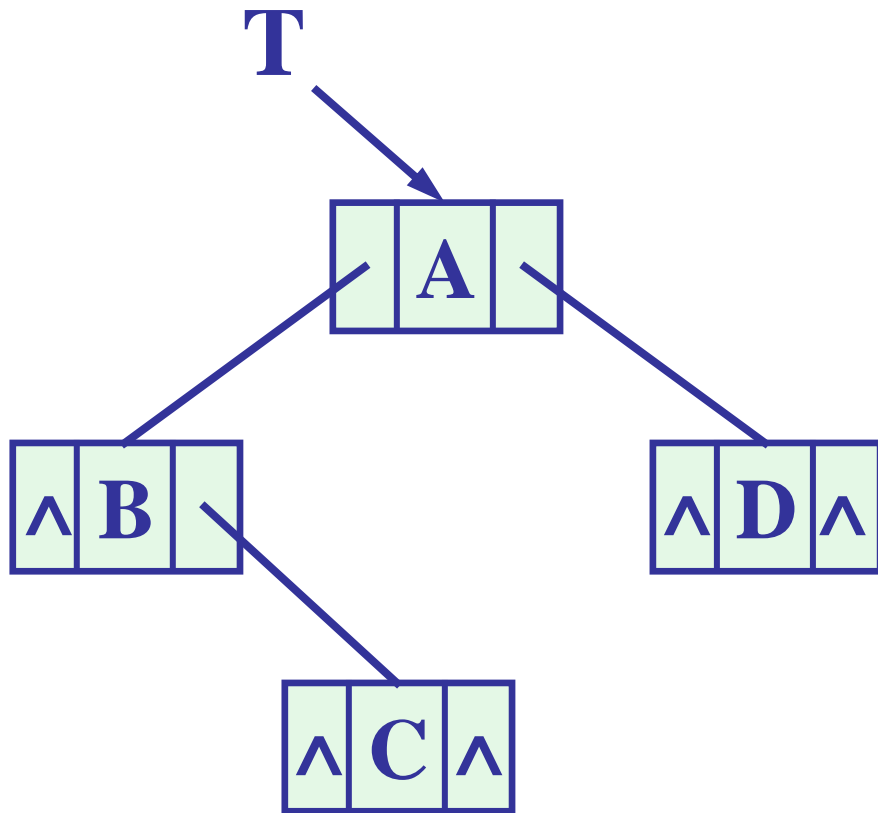
```
CreateBiTree(T->rchild); // 构造右子树
```

```
}
```

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



上页算法执行过程举例如下：



```
scanf(&ch);  
if (ch==' ') T = NULL;  
else {  
    if (!(T = new BiTNode))  
        exit(OVERFLOW);  
    T->data = ch;  
    CreateBiTree(T->lchild);  
    CreateBiTree(T->rchild);  
}
```



# 按给定的表达式建相应二叉树



- 由先缀表示式建树

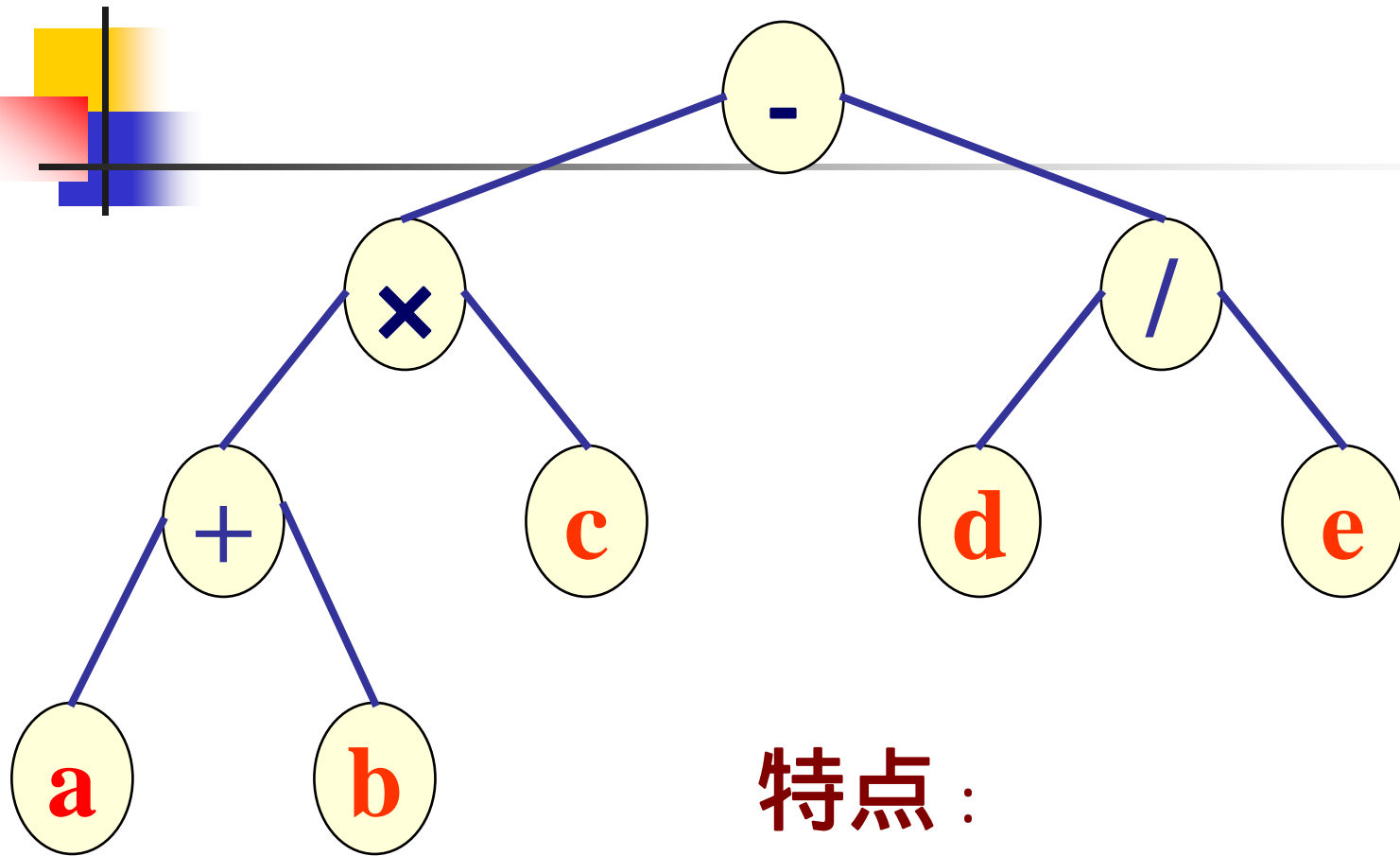
例如：已知表达式的先缀表示式

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

- 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



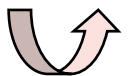
特点：

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:

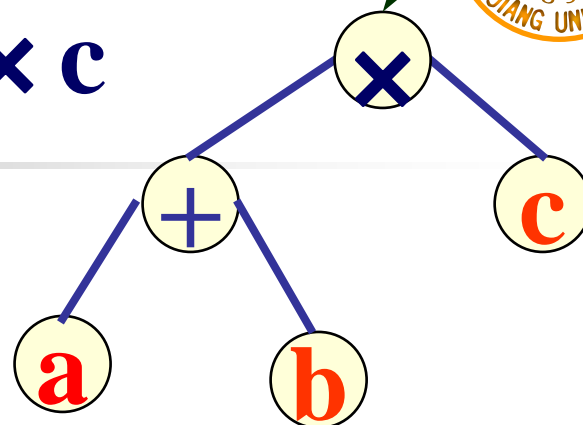
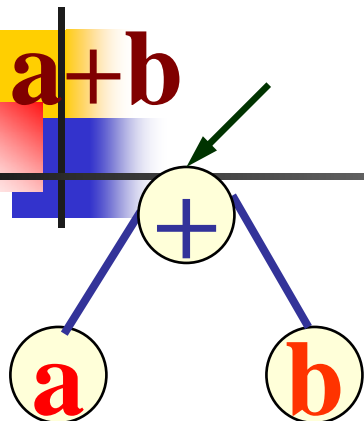
```
scanf(&ch);  
if ( In(ch, 字母集 )) 建叶子结点;  
else { 建根结点;  
        递归建左子树;  
        递归建右子树;  
}
```



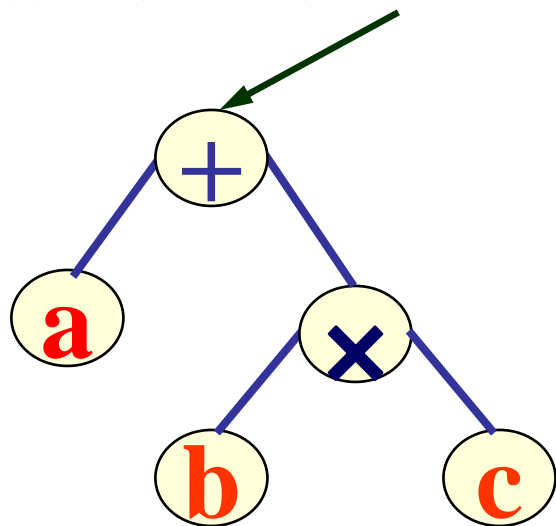
# 分析表达式和二叉树的关系:



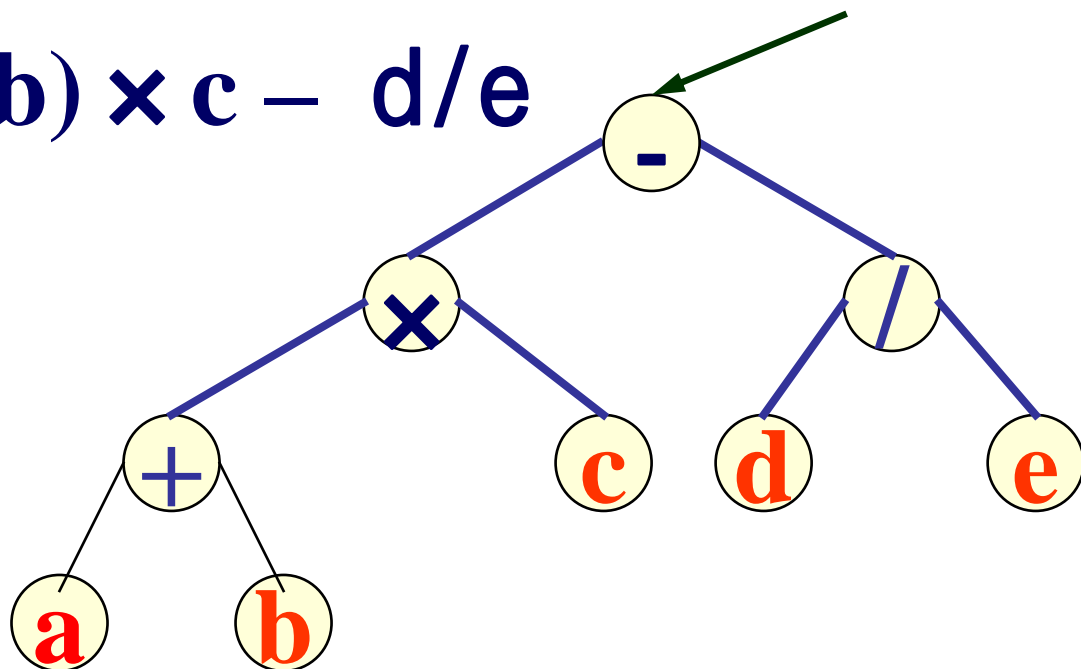
$$(a+b) \times c$$



$$a+b \times c$$



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





# 基本操作:

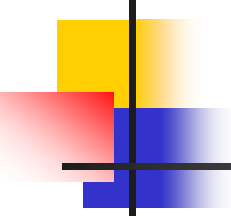
```
-scanf(&ch);  
if (In(ch, 字母集 )) { 建叶子结点; 暂存; }  
else if (In(ch, 运算符集))  
{ 和前一个运算符比较优先数;  
    若当前的优先数 “ 高 ” , 则暂存;  
    否则建子树;  
}
```

```
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
```

A decorative graphic in the top-left corner consisting of overlapping yellow, red, and blue squares with a black crosshair.

---

```
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)
```

```
{
```

```
    if (!(T= new BiTNode))
```

```
        exit(OVERFLOW);
```

```
    T->data = char;
```

```
    T->lchild = T->rchild = NULL;
```

```
    Push( PTR, T );
```

```
}
```

# 建子树的算法为：

A decorative graphic consisting of overlapping yellow, red, and blue squares with a black crosshair.

```
void CrtSubtree (Bitree& T, char c)
```

```
{
```

```
    if (!(T= new BiTNode))
```

```
        exit(OVERFLOW);
```

```
    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?

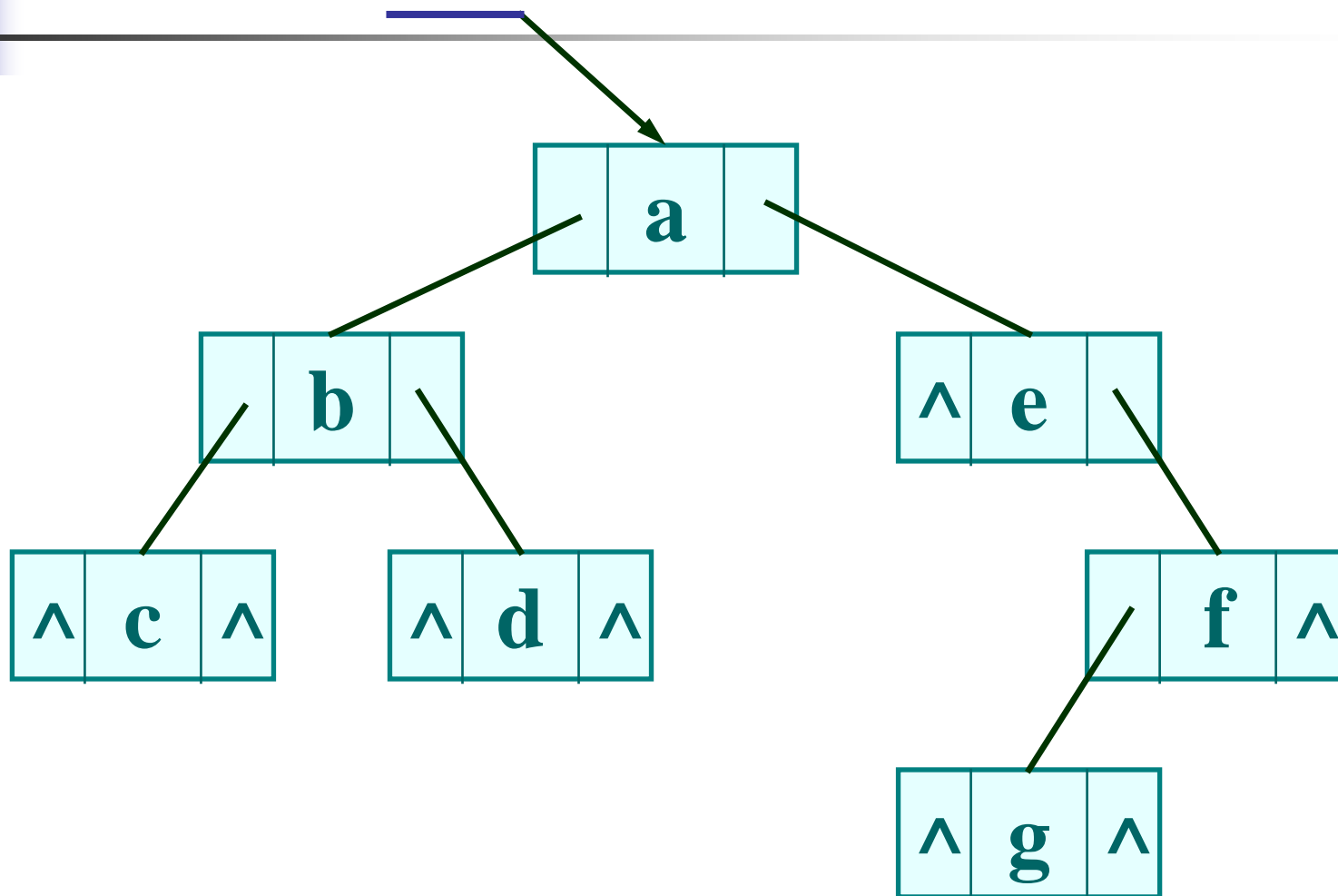
二叉树的先序序列 **根** **左子树** **右子树**

二叉树的中序序列 **左子树** **根** **右子树**

例如：

a	b	<u>c</u>	<u>d</u>	e	<u>f</u>	<u>g</u>
<u>c</u>	b	<u>d</u>	a	e	<u>g</u>	f

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 ,  
    // ins[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
```



```
if (!(T= new BiTNode)) exit(OVERFLOW);  
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 );
```





# Q&A

---



# 线索二叉树

何谓线索二叉树？

线索链表的遍历算法

如何建立线索链表？

# 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：

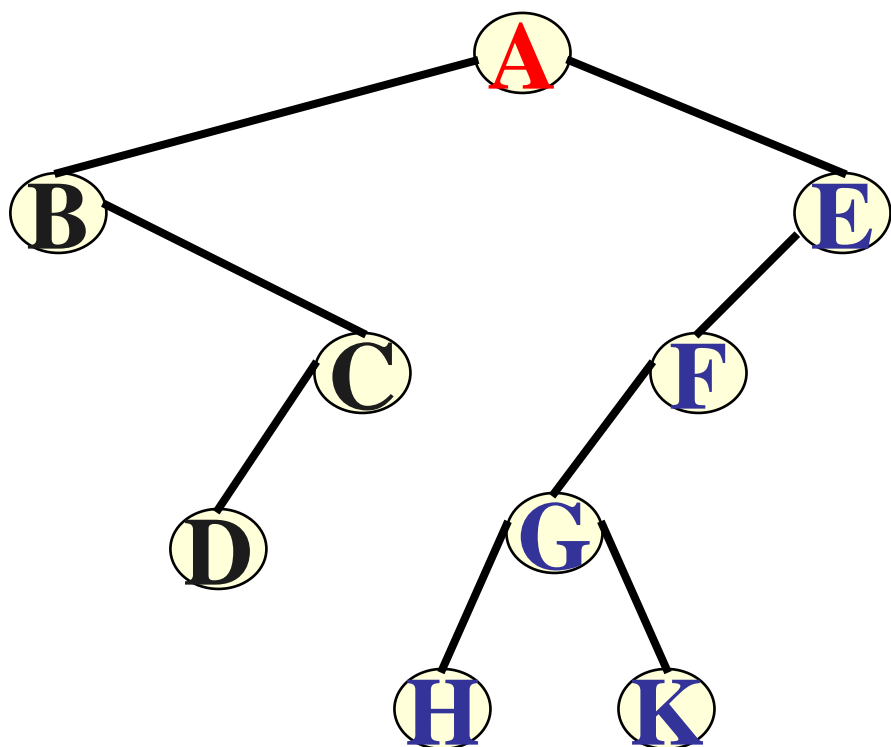
A B C D E F G H K

中序序列：

B D C A H G K F E

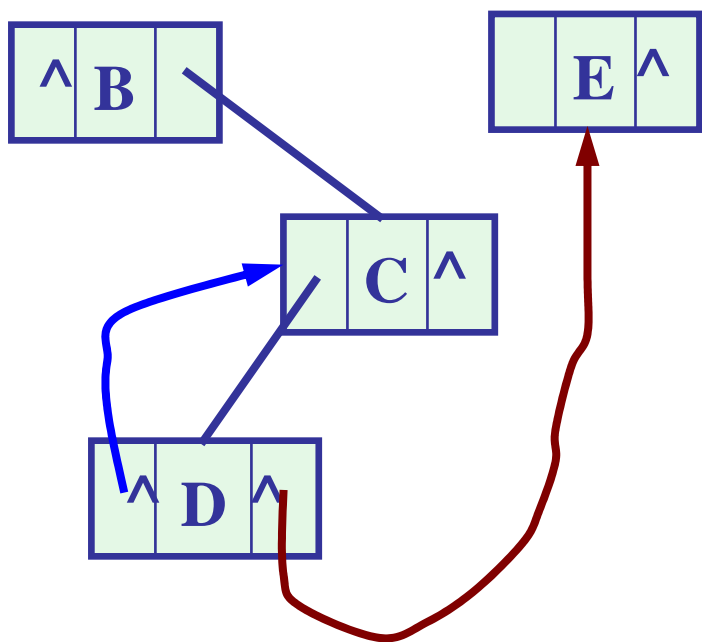
后序序列：

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

A horizontal sequence of letters A through K. A blue arrow points from D to C, and a red arrow points from D to E.

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**”



# 对线索链表中结点的约定：

在二叉链表的结点中增加两个标志域，  
并作如下规定：

- 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”。



- 若该结点的右子树不空，  
则rchild域的指针指向其右子树，  
且右标志域的值为“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
```

```
class BiThrNod {
```

```
    TElemType      data;
```

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

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

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



## 二、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，从而简化了遍历的算法。

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





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  
        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
```



### 三、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，并始终保持指针pre指向当前访问的、指针p所指结点的前驱。

```
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
```

```
Status InOrderThreading(BiThrTree &Thrt,  
                        BiThrTree T) { // 构建中序线索链表  
    if (!(Thrt = new 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;
```

```
}
```



# Q&A

---