# 数据结构与算法

## 第六章　树和图

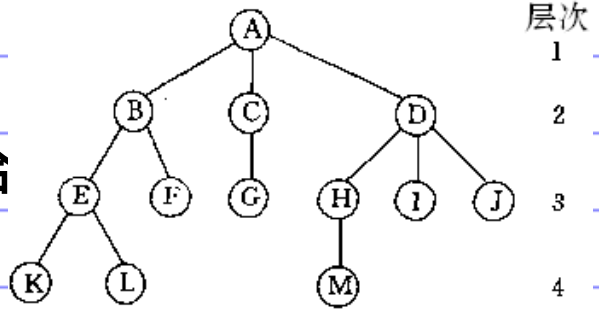
树的定义
二叉树
树的存储结构
树和二叉树的遍历
哈夫曼树

# 树和图

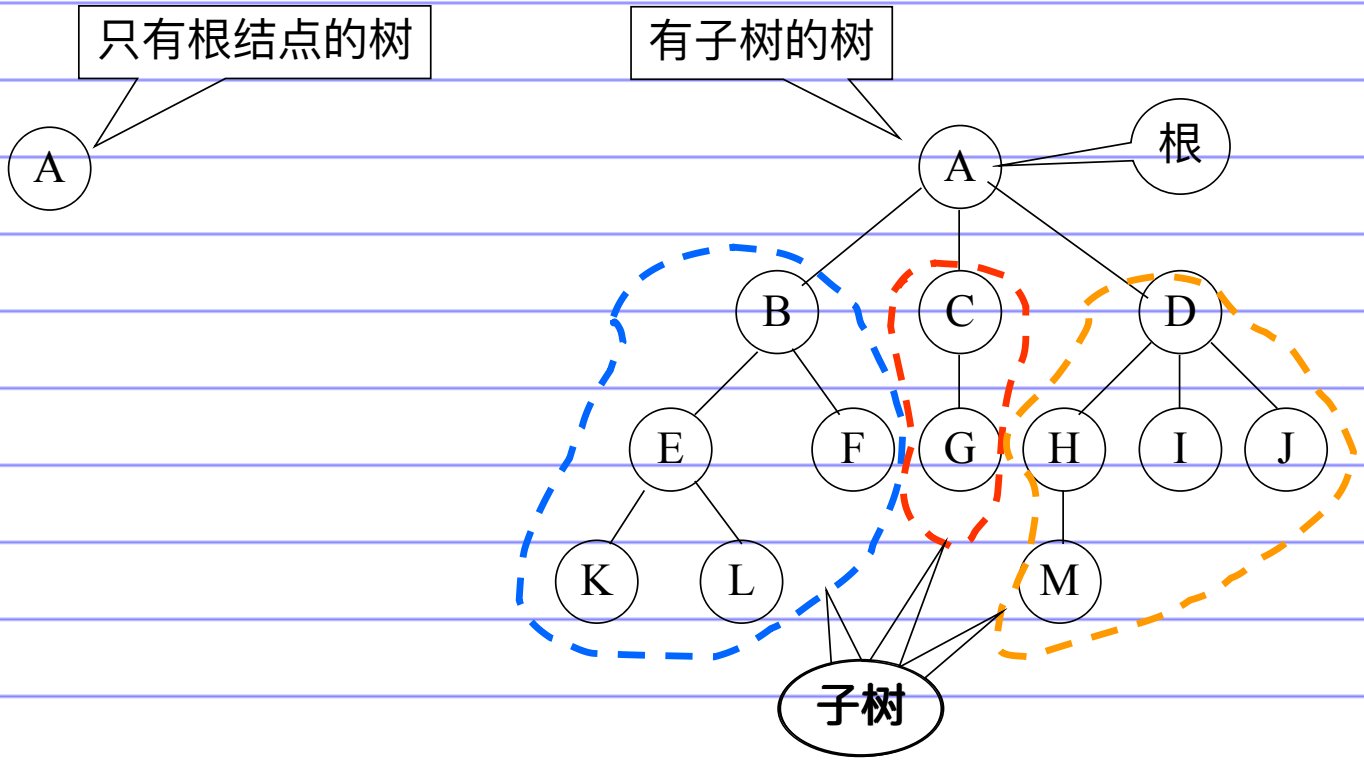
- 树和图都是非常重要的非线性数据结构。
- 树是以分支关系定义的层次结构

# 第一节 树的定义

- 定义：树(tree)是n(n>0)个结点的有限集T，其中：
  - 有且仅有一个特定的结点，称为树的根(root)
  - 当n>1时，其余结点可分为m(m>0)个互不相交的有限集$T_1,T_2,\ldots\ldots T_m$，其中每一个集合本身又是一棵树，称为根的子树(subtree)
- 特点：
  - 树中至少有一个结点——根
  - 树中各子树是互不相交的集合



# 树的例子



# 树的形式定义

树是一种数据结构　　$Tree = (D, R)$

其中：D 是具有相同特性的数据元素的集合；若 D 只含一个数据元素，则 R 为空集，否则 R 是 D 上某个二元关系 H 的集合，即 R＝{H}。H 为如下描述的二元关系：

（1）在 D 中存在唯一的称为根的数据元素 root，它在关系 H 下无前驱；

（2）若 $D-\{root\}\neq\Phi$，则存在 $D-\{root\}$ 的一个划分 $D_1, D_2, \cdots, D_m (m>0)$，对任意一对 $j\neq k (1\leqslant j,k\leqslant m)$ 有 $D_j\cap D_k=\Phi$，且对任意的 $i (1\leqslant i\leqslant m)$，唯一存在数据元素 $x_i\in D_i$，有 $\langle root, x_i\rangle\in H$；

（3）对应于 $D-\{root\}$ 的划分，$H-\{\langle root, x_1\rangle, \cdots, \langle root, x_m\rangle\}$ 有唯一的一个划分 $H_1, H_2, \cdots, H_m (m>0)$，对任意一对 $j\neq k (1\leqslant j,k\leqslant m)$ 有，$H_j\cap H_k=\Phi$，且对任意的 $i (1\leqslant i\leqslant m) H_i$ 是 $D_i$ 上的二元关系，$(D_i, \{H_i\})$ 是一棵符合本定义的树，称为根 root 的子树。

这是一个递归的定义，即在树的定义中又用到树的概念，它道出了树的固有特性。

# 树的基本术语

- 树的结点包含一个数据元素及若干指向其子树的分支。结点拥有的子树数称为结点的度(Degree)。度为0的结点称为叶子(Leaf)或终端结点。度不为0的结点称为非终端结点或分支结点。除根结点之外，分支结点也称为内部结点。树的度是树内各结点的度的最大值。
- 结点的子树的根称为该结点的孩子(child)，相应地，该结点称为孩子的双亲(Parent)。同一个双亲的孩子之间互称兄弟(sibling)。结点的祖先是从根到该结点所经分支上的所有结点。以某结点为根的子树中的任一结点都称为该结点的子孙。

# 树的基本术语

- **结点的层次**(Level)从根开始定义起，根为第一层，根的孩子为第二层。若某结点在第i层．则其子树的根就在第i+1 层。其双亲在同一层的结点互为堂兄弟。树中结点的最大层次称为**树的深度**(Depth)或高度。
- 如果将树中结点的各子树看成从左至右是有次序的(即不能互换)，则称该树为**有序树**，否则称为**无序树**。在有序树中最左边的子树的根称为第一个孩子，最右边的称为最后一个孩子。
- **森林**(Forest)是m(m>0)棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。由此，也可以森林和树相互递归的定义来描述树。

# 树的术语例

叶子：K，L，F，G，M，I，J
结点A的度：3
结点B的度：2
结点M的度：0

树的度：3

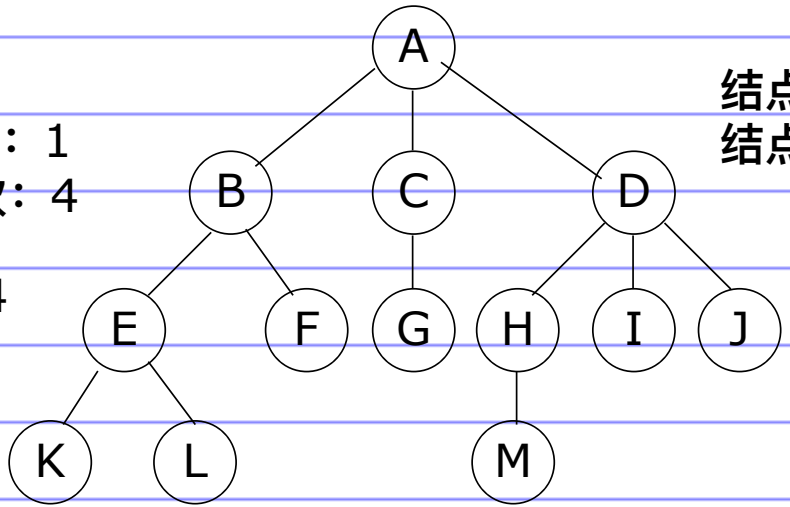
结点A的层次：1
结点M的层次：4

树的深度：4

结点A的孩子：B，C，D
结点B的孩子：E，F

结点I的双亲：D
结点L的双亲：E

结点B，C，D为兄弟
结点K，L为兄弟




结点F，G为堂兄弟
结点A是结点F，G的祖先

# 树的基本操作

InitTree(&T);
操作结果：构造空树 T。

DestroyTree(&T);
初始条件：树 T 存在。
操作结果：销毁树 T。

CreateTree(&T, definition);
初始条件：definition 给出树 T 的定义。
操作结果：按 definition 构造树 T。

ClearTree(&T);
初始条件：树 T 存在。
操作结果：将树 T 清为空树。

TreeEmpty(T);
初始条件：树 T 存在。
操作结果：若 T 为空树，则返回 TRUE，否则 FALSE。

TreeDepth(T);
初始条件：树 T 存在。
操作结果：返回 T 的深度。

Root(T);
初始条件：树 T 存在。
操作结果：返回 T 的根。

Value(T, cur_e);
初始条件：树 T 存在，cur_e 是 T 中某个结点。
操作结果：返回 cur_e 的值。

Assign(T, cur_e, value);
初始条件：树 T 存在，cur_e 是 T 中某个结点。
操作结果：结点 cur_e 赋值为 value。

Parent(T, cur_e);
初始条件：树 T 存在，cur_e 是 T 中某个结点。
操作结果：若 cur_e 是 T 的非根结点，则返回它的双亲，否则函数值为“空”。

LeftChild(T, cur_e);
初始条件：树 T 存在，cur_e 是 T 中某个结点。
操作结果：若 cur_e 是 T 的非叶子结点，则返回它的最左孩子，否则返回“空”。

RightSibling(T, cur_e);
初始条件：树 T 存在，cur_e 是 T 中某个结点。
操作结果：若 cur_e 有右兄弟，则返回它的右兄弟，否则函数值为“空”。

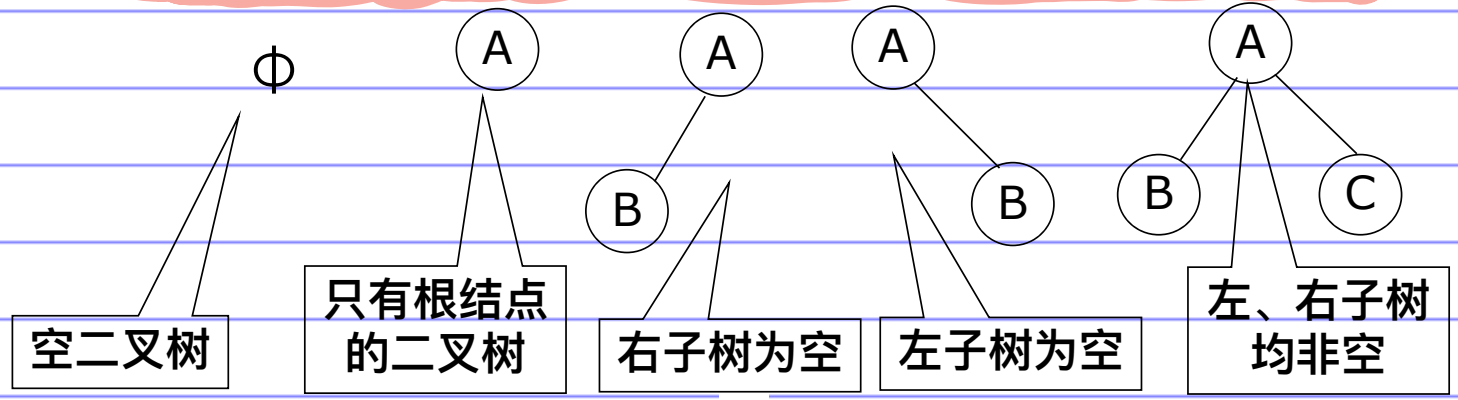
InsertChild(&T, &p, i, c);
初始条件：树 T 存在，p 指向 T 中某个结点，$1 \leqslant i \leqslant p$ 所指结点的度 + 1，非空树 c 与 T 不相交。
操作结果：插入 c 为 T 中 p 指结点的第 i 棵子树。

DeleteChild(&T, &p, i);
初始条件：树 T 存在，p 指向 T 中某个结点，$1 \leqslant i \leqslant p$ 指结点的度。
操作结果：删除 T 中 p 所指结点的第 i 棵子树。

TraverseTree(T, Visit());
初始条件：树 T 存在，Visit 是对结点操作的应用函数。
操作结果：按某种次序对 T 的每个结点调用函数 visit()一次且至多一次。一旦 visit()失败，则操作失败。

# 第二节 二叉树

- 定义：二叉树是n(n≥0)个结点的有限集，它或为空树(n=0)，或由一个根结点和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
  - 每个结点至多有二棵子树(即不存在度大于2的结点)
  - 二叉树的子树有左、右之分，且其次序不能任意颠倒




- 有三个节点的树有几种形态？

# 二叉树性质

- **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：用归纳法证明之

①i=1时，只有一个根结点，$2^{i-1} = 2^0 = 1$是对的

②假设对所有j($1 \leq j < i$)命题成立，即第j层上至多有$2^{j-1}$个结点

那么，第i-1层至多有$2^{i-2}$个结点

又二叉树每个结点的度至多为2

∴ 第i层上最大结点数是第i-1层的2倍，即$2 \cdot 2^{i-2} = 2^{i-1}$

故命题得证

- **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，可得深度为k 的二叉树最大结点数是

$$\sum_{i=1}^{k}(\text{第}i\text{层的最大结点数}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

# 二叉树性质

- **性质3：对任何一棵二叉树T，如果其终端结点数为$n_0$，度为2的结点数为$n_2$，则$n_0=n_2+1$**

证明：$n_1$为二叉树T中度为1的结点数

因为：二叉树中所有结点的度均小于或等于2

所以：其结点总数$n=n_0+n_1+n_2$

又二叉树中，除根结点外，其余结点都只有一个分支进入

设B为分支总数，则$n=B+1$

又：分支由度为1和度为2的结点射出，

$\therefore B=n_1+2n_2$

于是，$n=B+1=n_1+2n_2+1=n_0+n_1+n_2$

$\therefore n_0=n_2+1$

# 特殊形式的二叉树

- 满二叉树
  - 定义：一深度为k且有$2^k-1$个结点的二叉树
  - 特点：每一层上的结点数都是最大结点数
- 完全二叉树
  - 定义：深度为k，有n个结点的二叉树当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应时，称完全二叉树
  - 特点
    - 叶子结点只可能在层次最大的两层上出现
    - 对任一结点，若其右分支下子孙的最大层次为l，则其左分支下子孙的最大层次必为l 或l+1
  - 性质：

# 完全二叉树性质

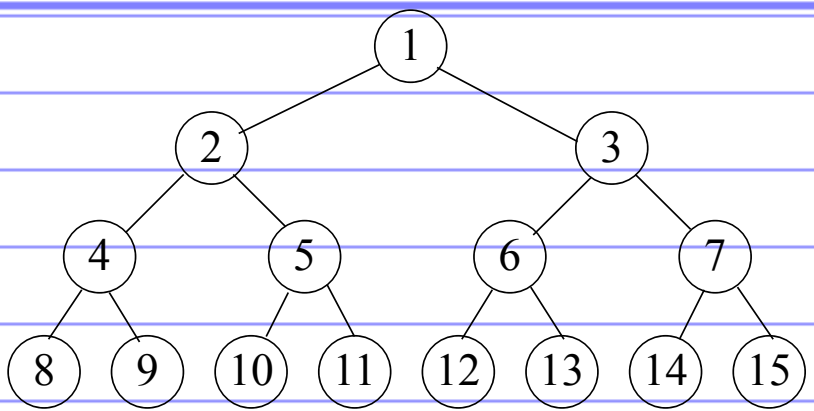
- 性质4：具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$
- 性质5：如果对一棵有n个结点的完全二叉树的结点按层序编号，则对任一结点i(1≤i≤n)，有：

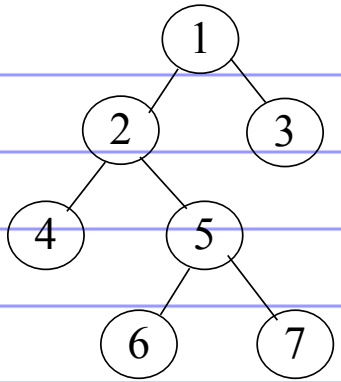
(1) 如果i=1，则结点i是二叉树的根，无双亲；如果i>1，则其双亲是⌊i/2⌋

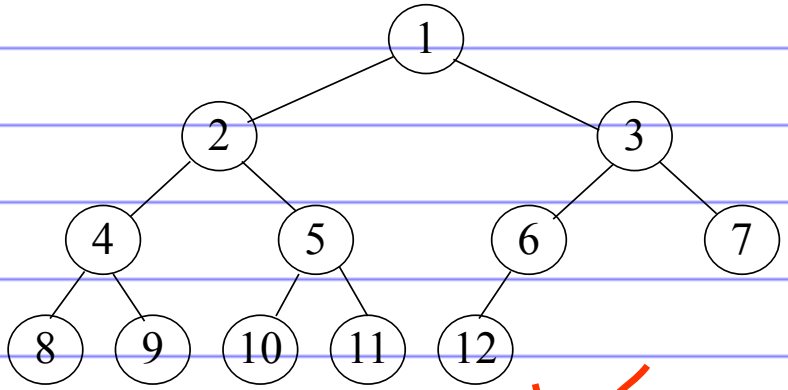
(2) 如果2i>n，则结点i无左孩子；如果2i≤n，则其左孩子是2i

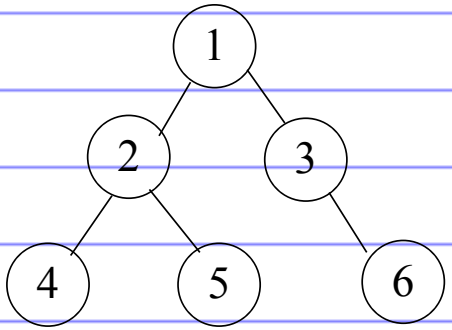
(3) 如果2i+1>n，则结点i无右孩子；如果2i+1≤n，则其右孩子是2i+1

# 二叉树例



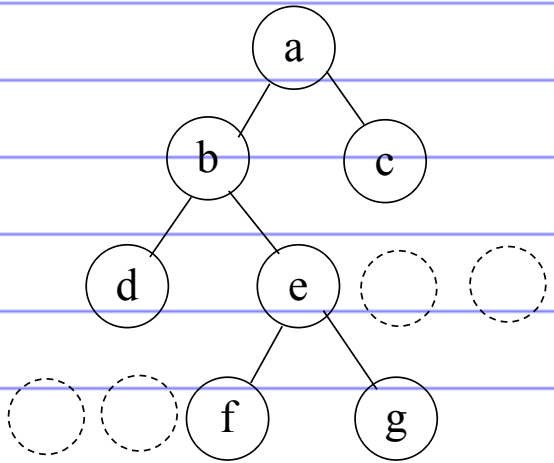



# 二叉树的存储结构

- 顺序存储结构
  - 实现：按满二叉树的结点层次编号，依次存放二叉树中的数据元素
  - 特点：
    - 结点间关系蕴含在其存储位置中
    - 浪费空间，适于存满二叉树

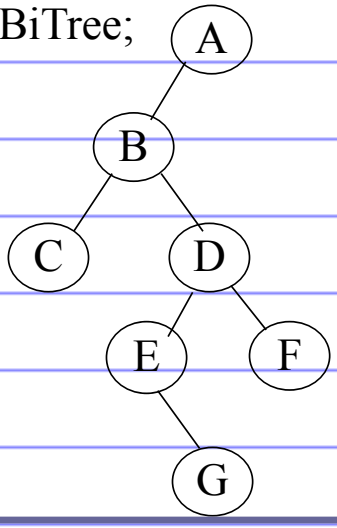


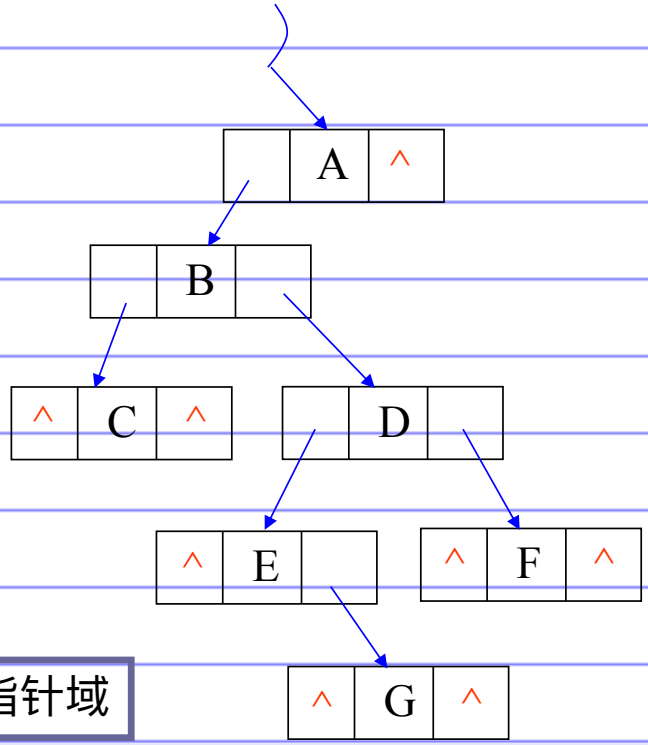

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | 0 | 0 | 0 | 0 | f | g |

# 二叉树的链式存储结构

## 二叉链表

```
typedef  struct BiTNode {
      TElemType   data;
      Struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

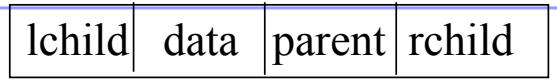

在n个结点的二叉链表中，有n+1个空指针域

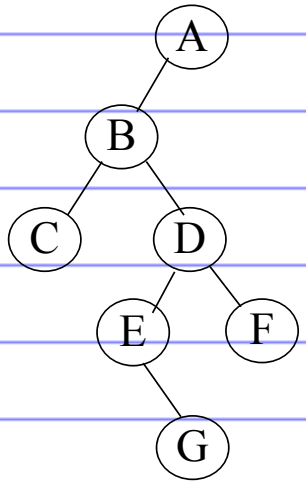
2n个指针，(n-1)个有指向，则有(n+1)个空

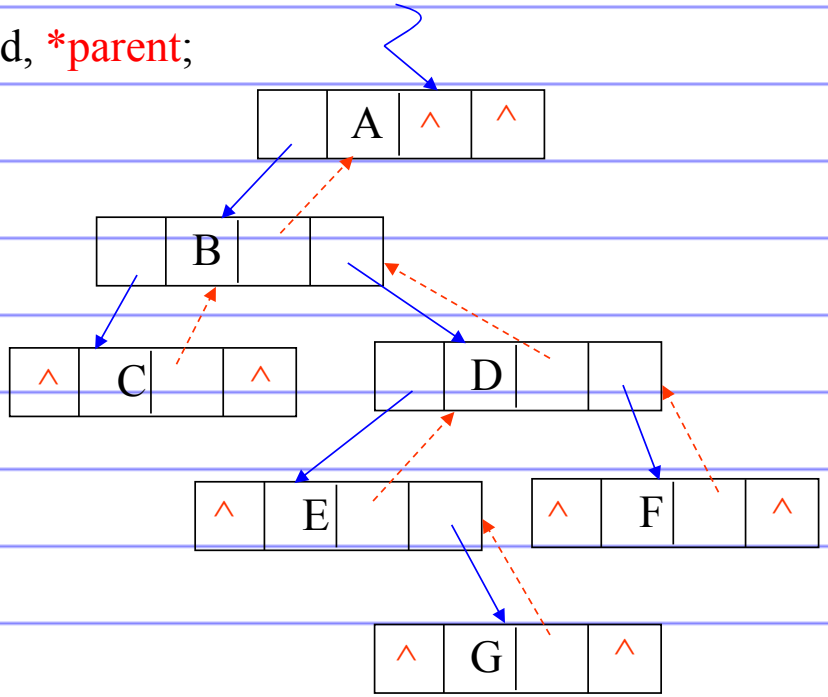
# 二叉树的链式存储结构

## 三叉链表

```
typedef  struct TrTNode {
   TElemType       data;
   Struct TrTNode*lchild, *rchild, *parent;
} TrTNode, TrTree;
```

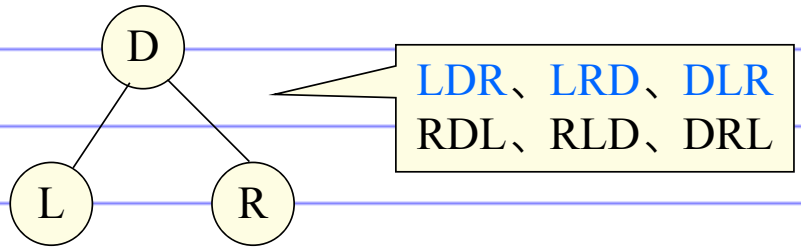


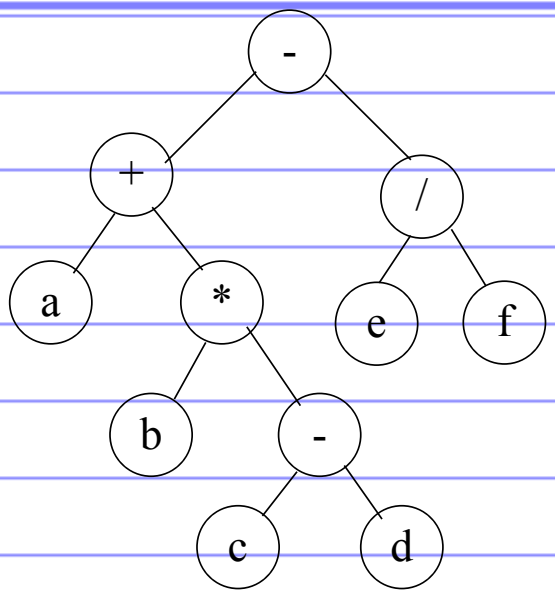



# 二叉树的遍历

- 深度优先算法
  - 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
- 广度优先算法
  - 按层次遍历：从上到下、从左到右访问各结点






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

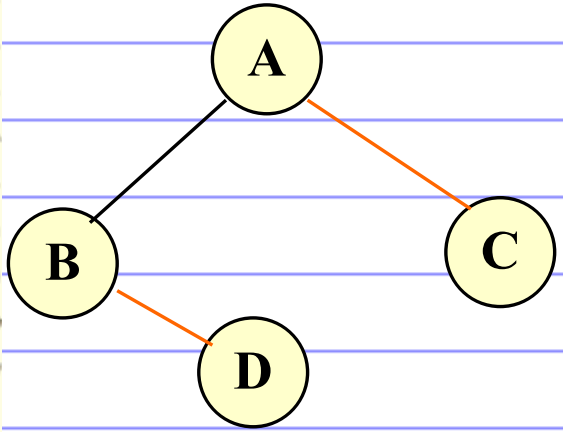
# 树的遍历算法

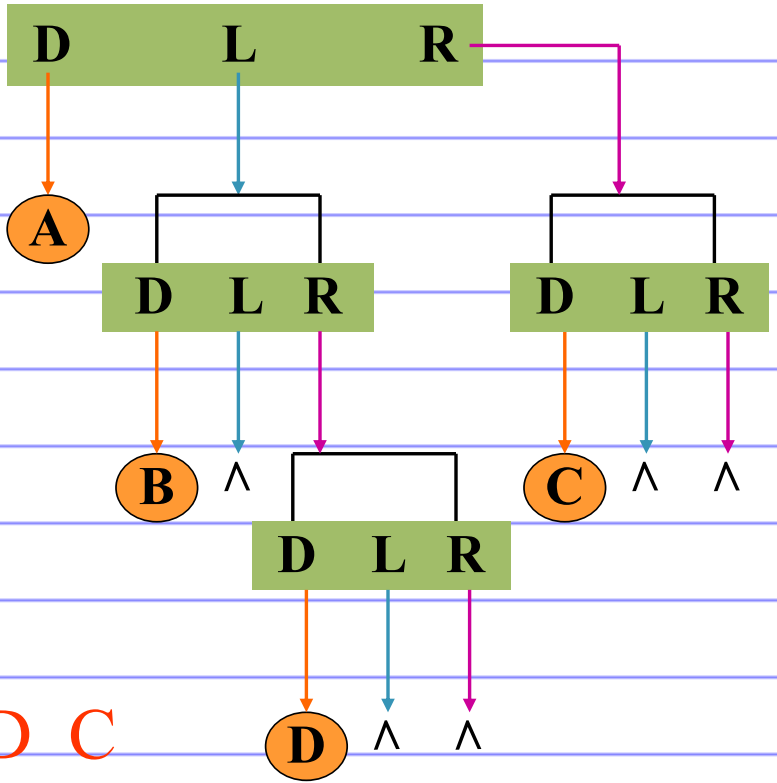
```
void preorder(TreeNode *bt)
{
    if(bt!=NULL){
            printf("%d\t",bt->data);
            preorder(bt->lchild);
            preorder(bt->rchild);
    }
}
```

```
void postorder(TreeNode *bt)
{
    if(bt!=NULL){
            postorder(bt->lchild);
            postorder(bt->rchild);
            printf("%d\t",bt->data);
    }
}
```

```
void inorder(TreeNode *bt)
{
    if(bt!=NULL) {
            inorder(bt->lchild);
            printf("%d\t",bt->data);
            inorder(bt->rchild);
    }
}
```

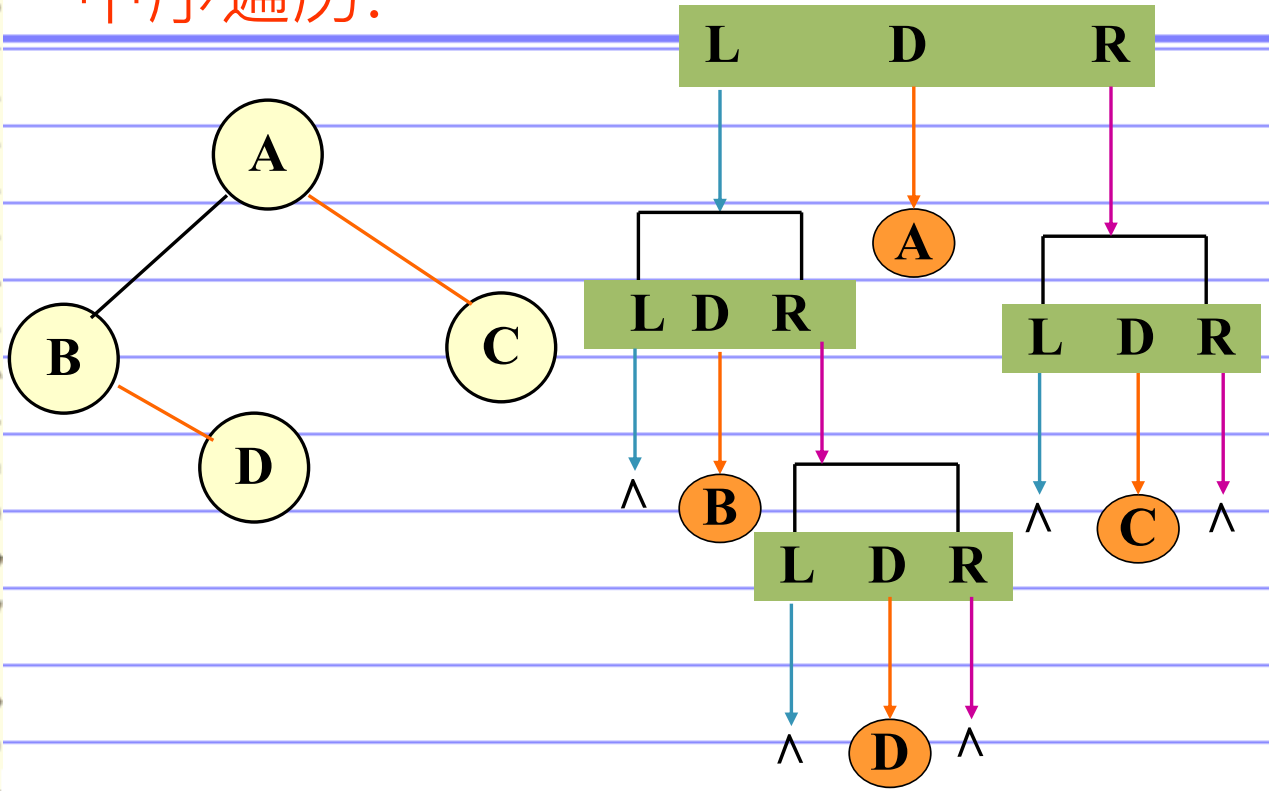
先序遍历:



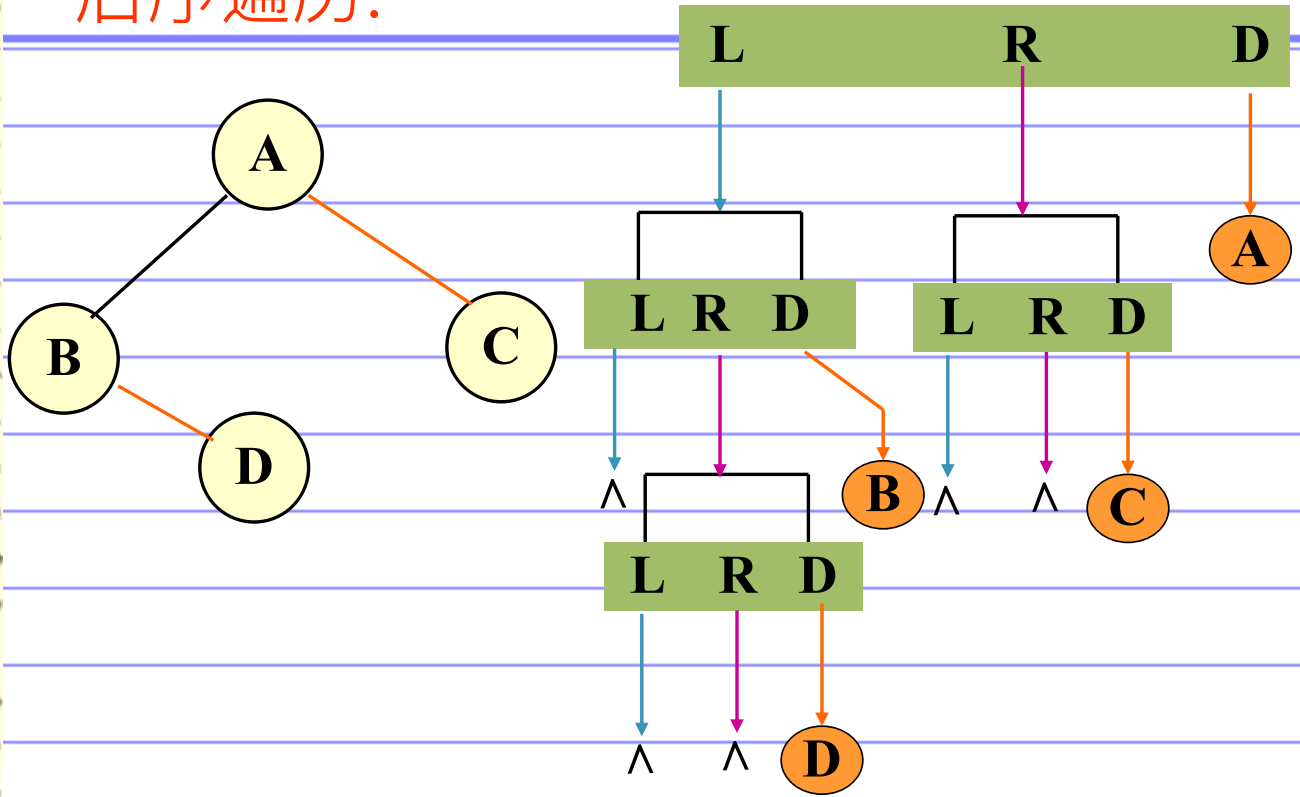

先序遍历序列：A B D C

# 中序遍历:



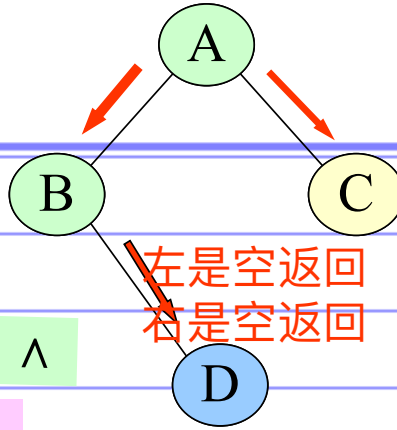

中序遍历序列：B D A C

# 后序遍历:




后序遍历序列： D B C A

```
void preorder(BiTree *bt)
{  if(bt!=NULL)
   {  printf("%d\t",bt->data);
      preorder(bt->lchild);
      preorder(bt->rchild);
   }
}
```




主程序

Pre( T )

T → A
printf(A);
pre(T→L);
pre(T→R);

T → B
printf(B);
pre(T→L);
pre(T→R);

T → C
printf(C);
pre(T→L);
pre(T→R);

T → D
printf(D);
pre(T→L);
pre(T→R);

T → ∧ 返回

T → ∧ 返回

T → ∧ 返回

T → ∧ 返回

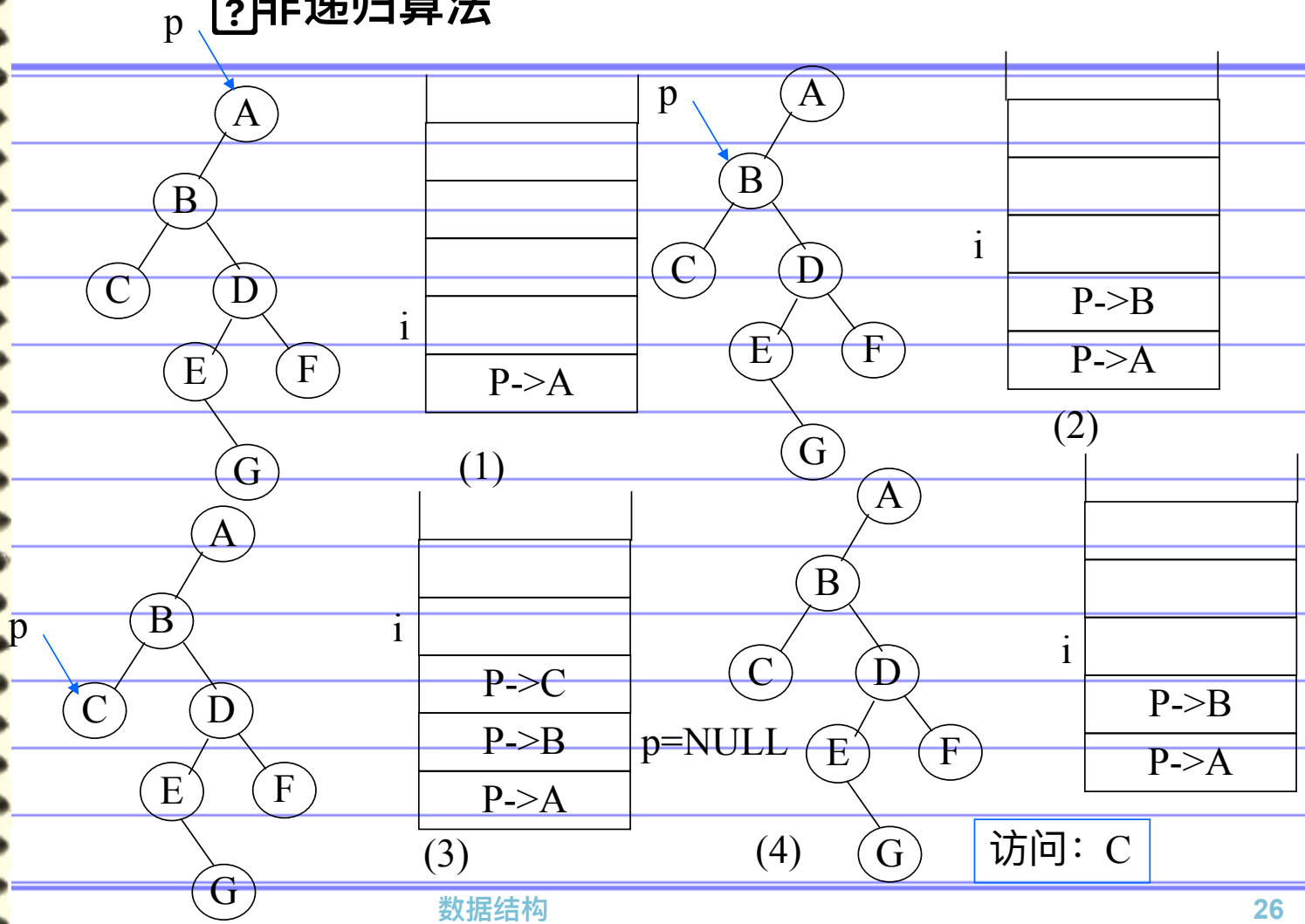
左是空返回
右是空返回

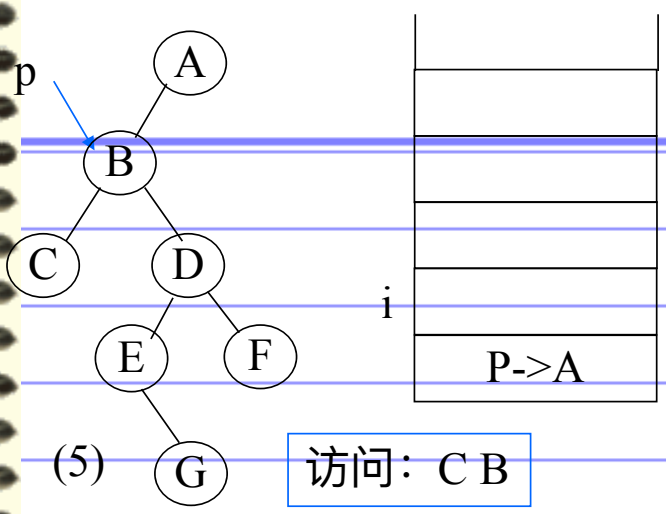
先序序列：A B D C



Back

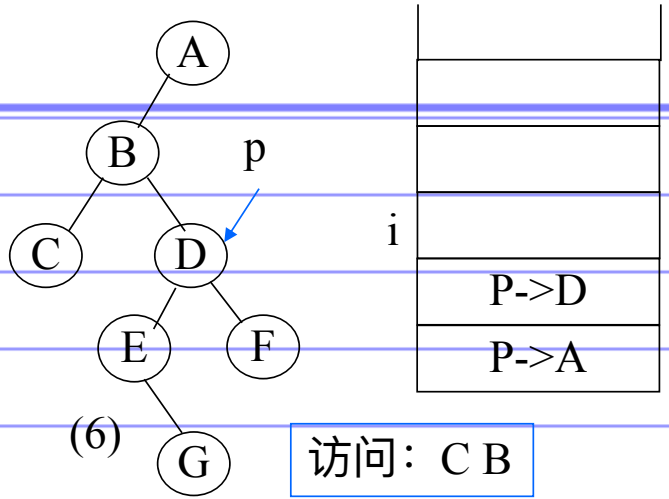
## 非递归算法


p
A
B
C
D
E
F
G
i
P->A
(1)
p
A
B
C
D
E
F
G
i
P->B
P->A
(2)
p
A
B
C
D
E
F
G
i
P->C
P->B
P->A
(3)
p=NULL
A
B
C
D
E
F
G
i
P->B
P->A
(4)
访问：C



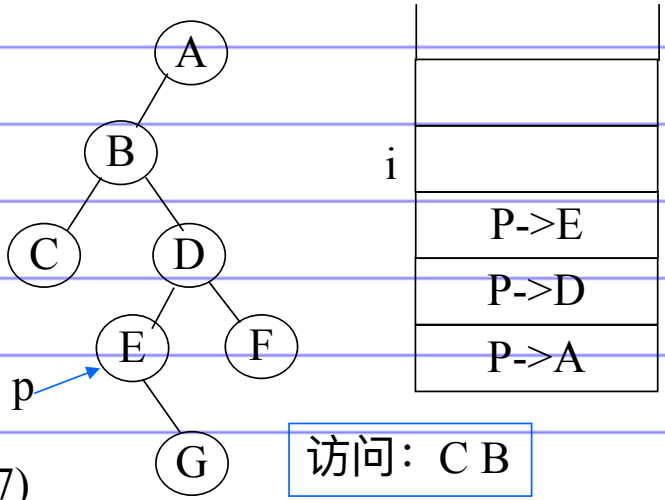

(5)

访问：C B



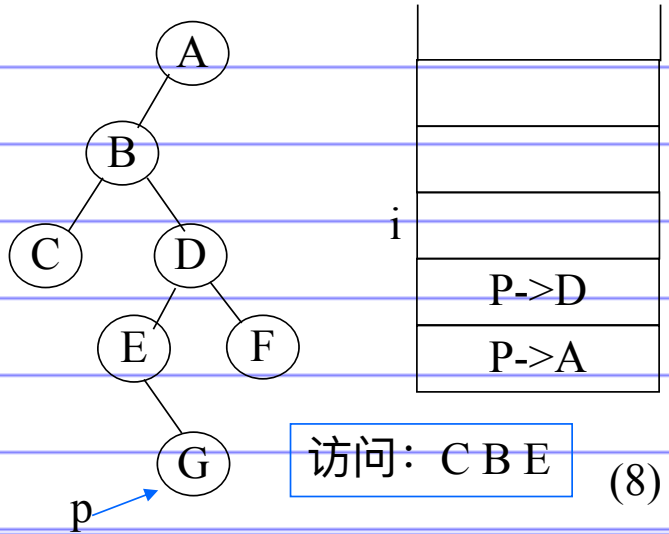

(6)

访问：C B



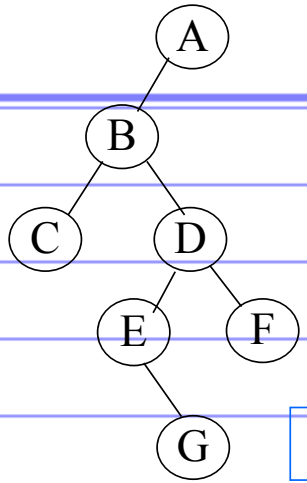

(7)

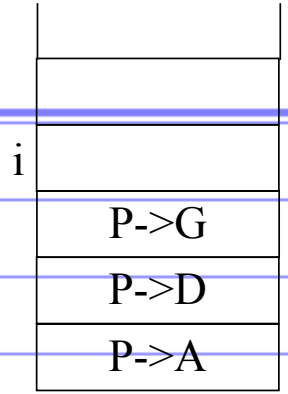
访问：C B



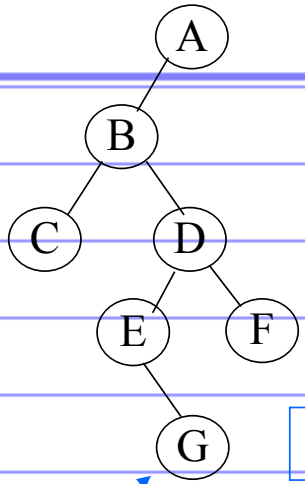

访问：C B E

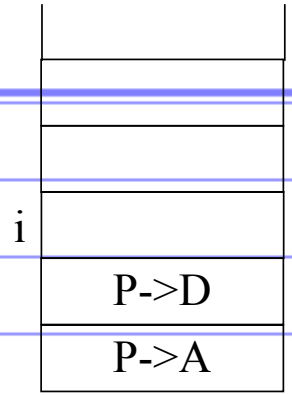
(8)




访问：C B E (9)

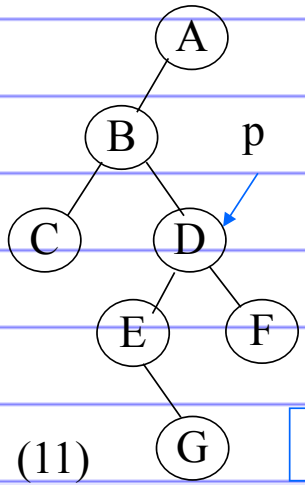


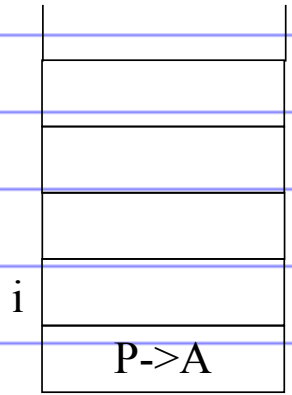




访问：C B E G (10)

P=NULL

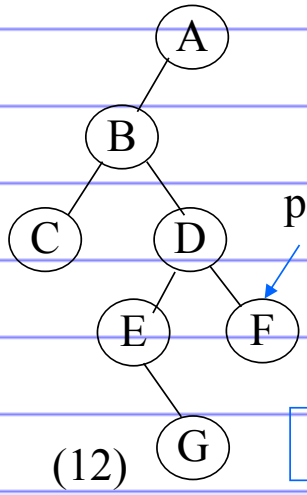
p

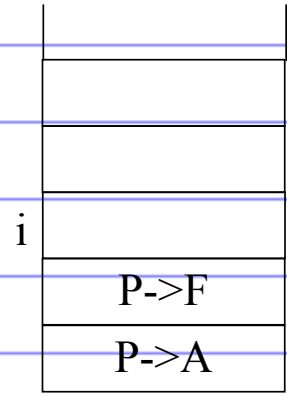



(11)

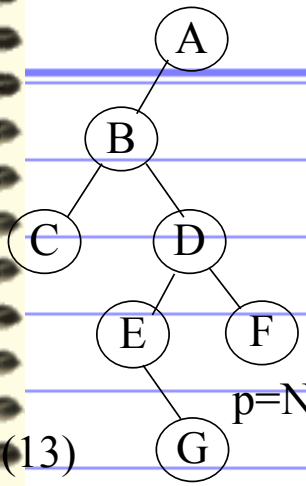
访问：C B E G D

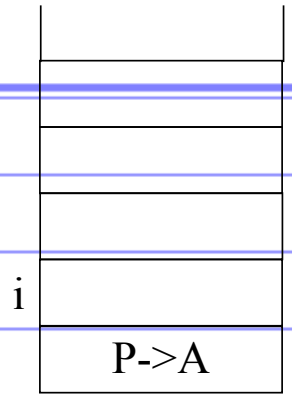



(12)

访问：C B E G D

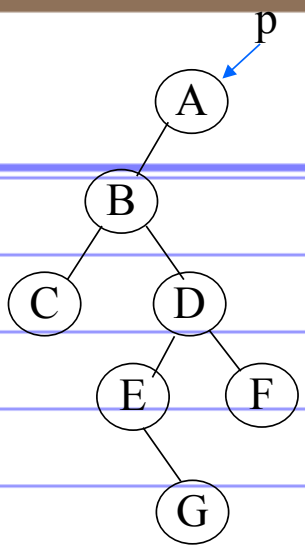


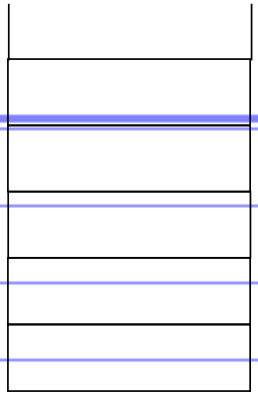

p=NULL

(13)

访问：C B E G D F

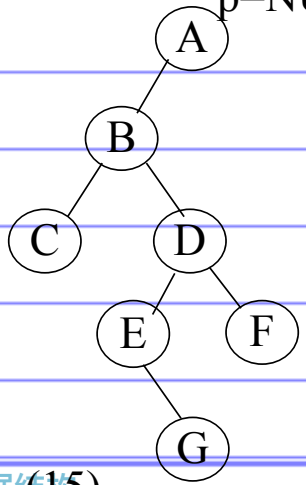



(14)




访问：C B E G D F A

p=NULL



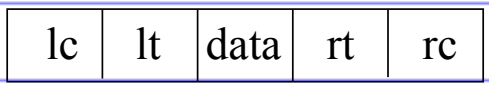


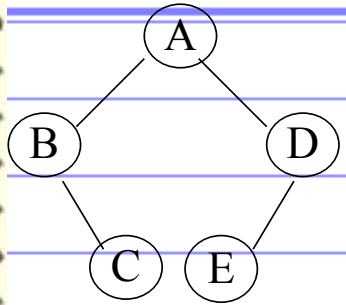
(15)

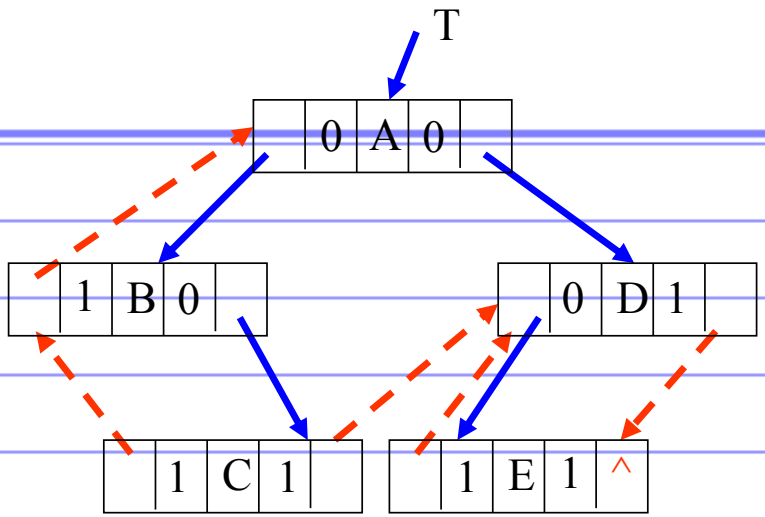
访问：C B E G D F A

# 线索二叉树

- 定义：
  - 前驱与后继：在二叉树的先序、中序或后序遍历序列中两个相邻的结点互称为~
  - 线索：指向前驱或后继结点的指针称为~
  - 线索二叉树：加上线索的二叉链表表示的二叉树叫~
  - 线索化：对二叉树按某种遍历次序使其变为线索二叉树的过程叫~
- 实现
  - 在有n个结点的二叉链表中必定有n+1个空链域
  - 在线索二叉树的结点中增加两个标志域
    - lt ：若 lt =0, lc 域指向左孩子；若 lt=1, lc域指向其前驱
    - rt ：若 rt =0, rc 域指向右孩子；若 rt=1, rc域指向其后继

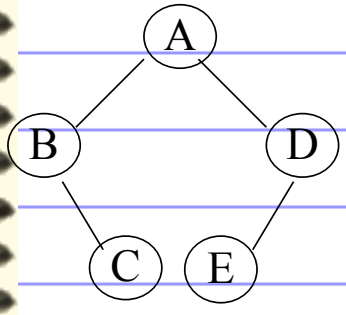


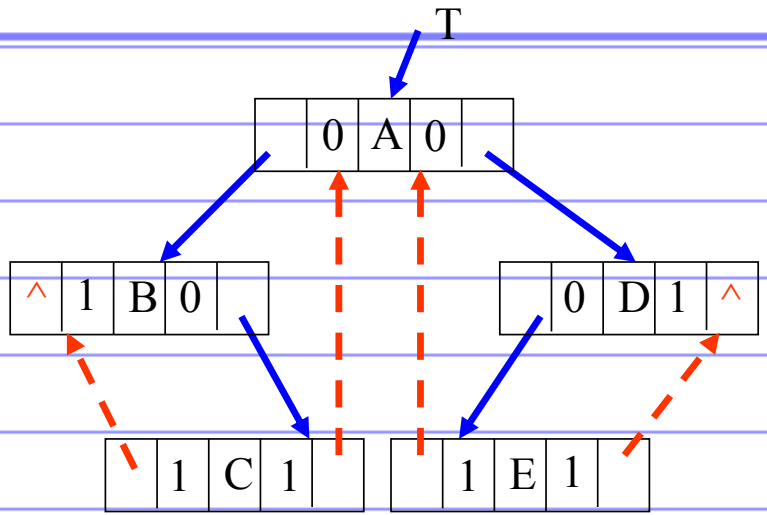



先序序列：ABCDE
先序线索二叉树

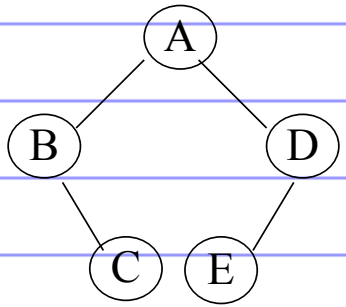
```
Typedef enum PointerTag { Link, Thread};
typedef  struct BiThrNode {
    TElemType data;
    PointerTag LTag, RTag;
    struct BiThrNode *lchild, *rchild;
} BiThrNode, *BiThrTree;
```

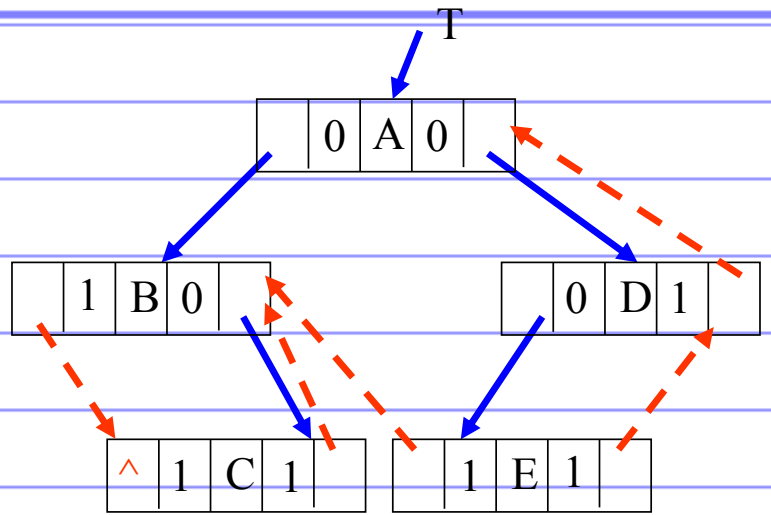



中序序列：BCAED
中序线索二叉树

```
Typedef enum PointerTag { Link, Thread};
typedef  struct BiThrNode {
    TElemType data;
    PointerTag LTag, RTag;
    struct BiThrNode *lchild, *rchild;
} BiThrNode, *BiThrTree;
```

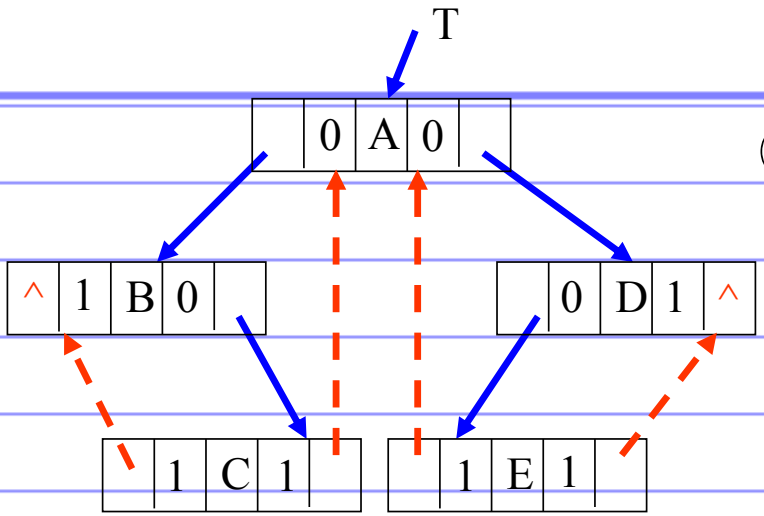



后序序列：CBEDA

后序线索二叉树

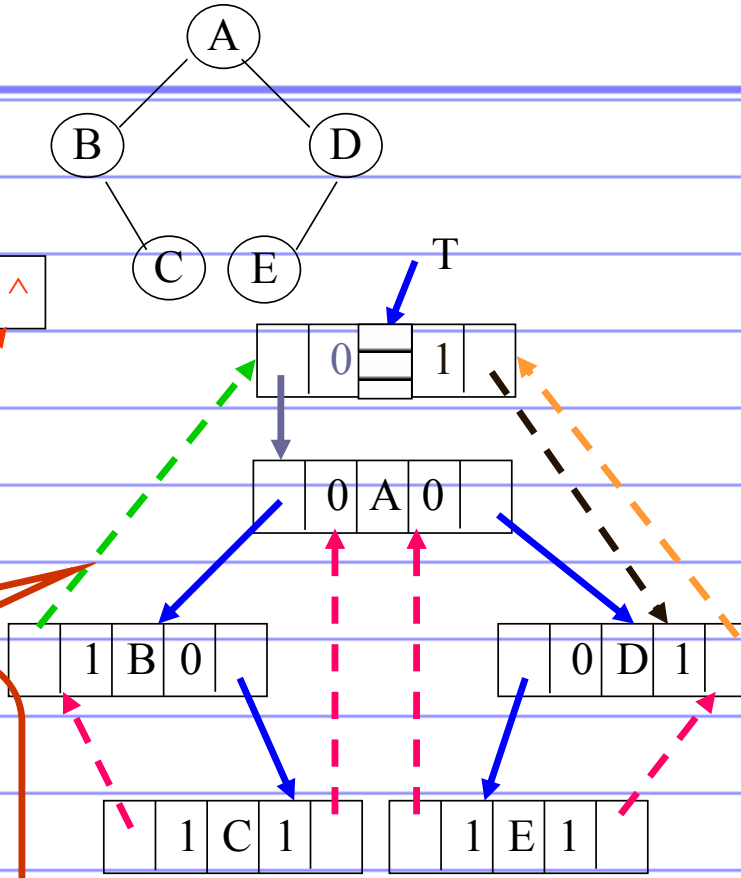
```
Typedef enum PointerTag { Link, Thread};
typedef  struct BiThrNode {
    TElemType data;
    PointerTag LTag, RTag;
    struct BiThrNode *lchild, *rchild;
} BiThrNode, *BiThrTree;
```




中序序列：BCAED
中序线索二叉树

头结点：
lt=0, lc指向根结点
rt=1, rc指向遍历序列中最后一个结点

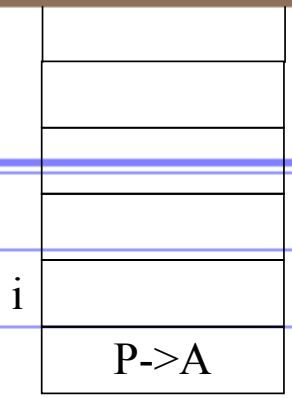
遍历序列中第一个结点的lc域和最后一个结点的rc域都指向头结点

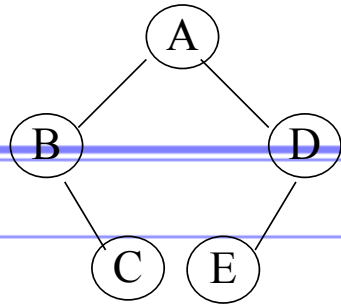


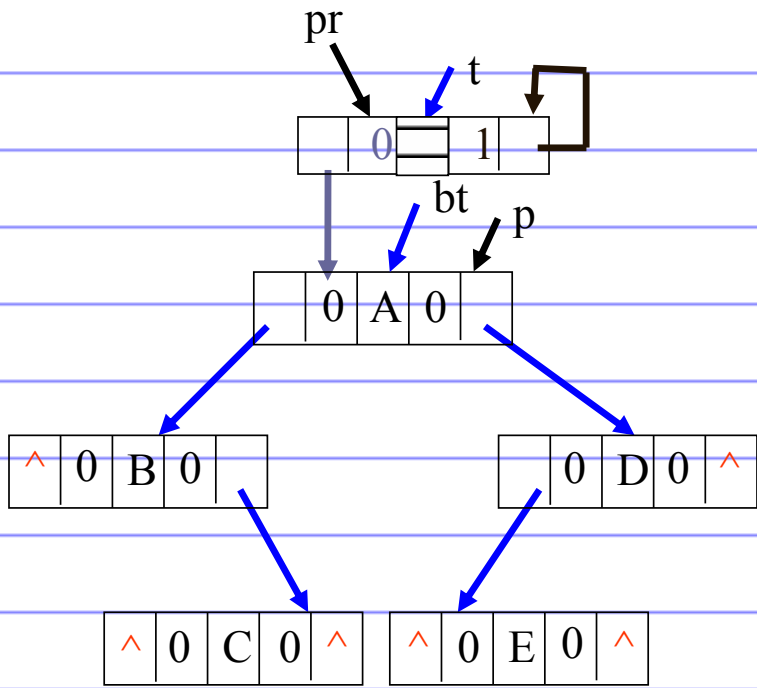

中序序列：BCAED
带头结点的中序线索二叉树

## ✧算法

### ?按中序线索化二叉树

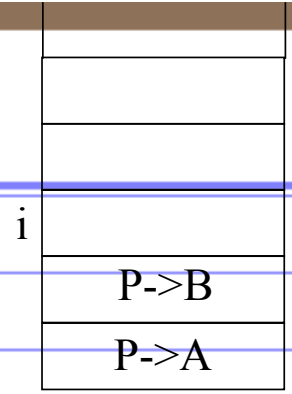


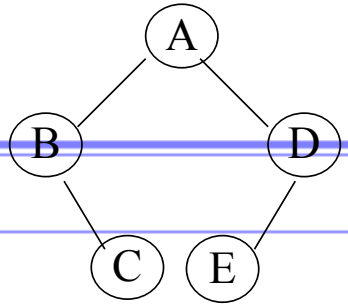



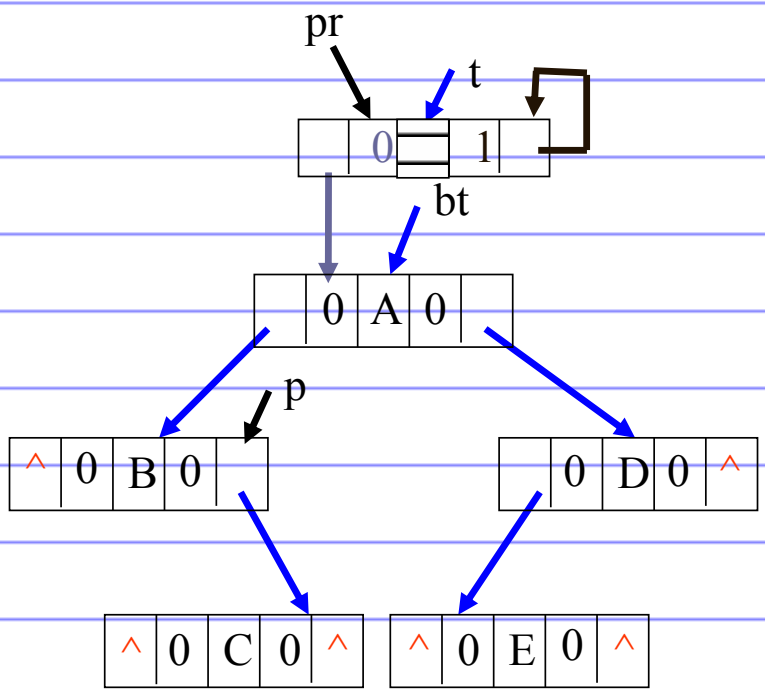



## ✧算法

### ?按中序线索化二叉树

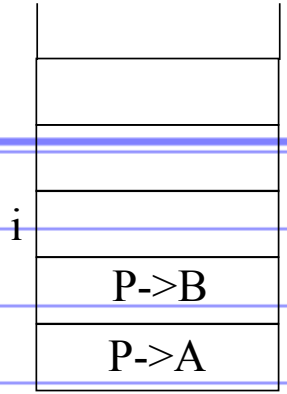


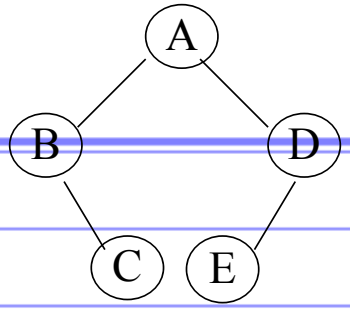



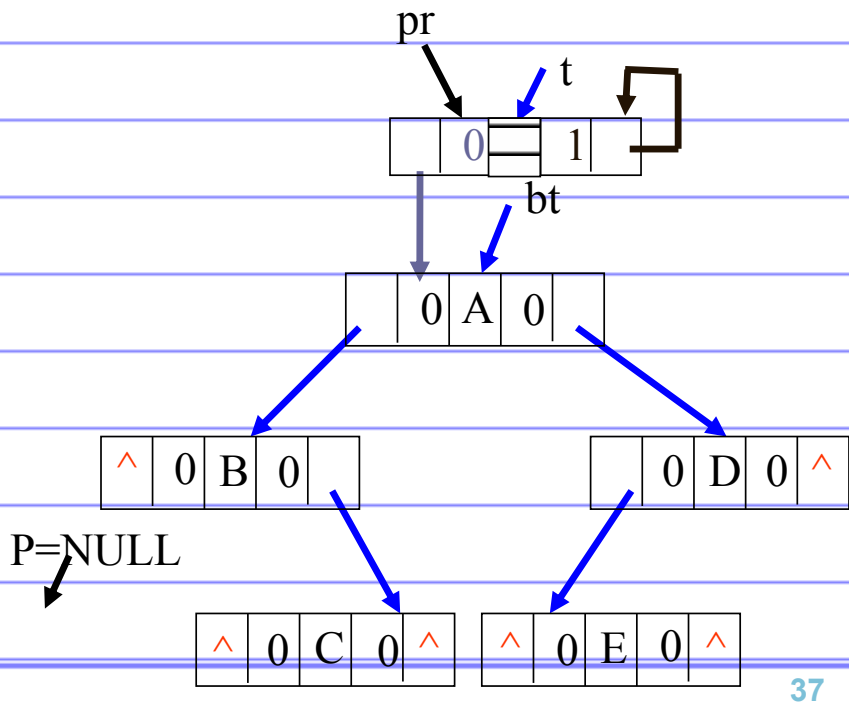



## ✧算法

### ?按中序线索化二叉树

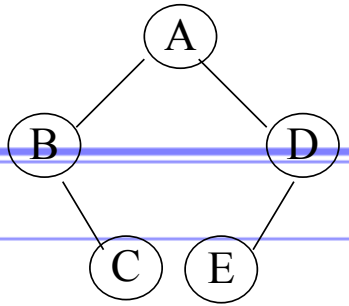


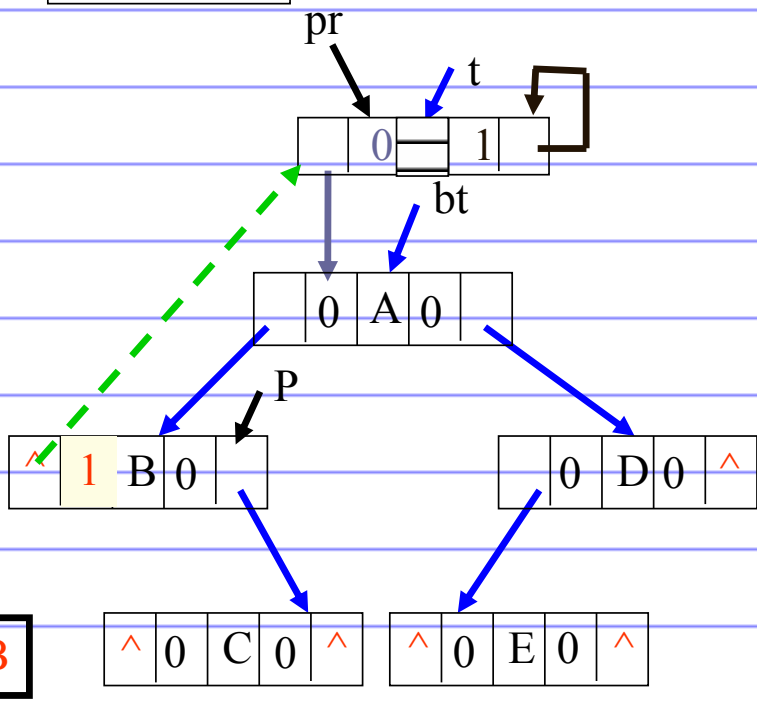



✧算法

?按中序线索化二叉树

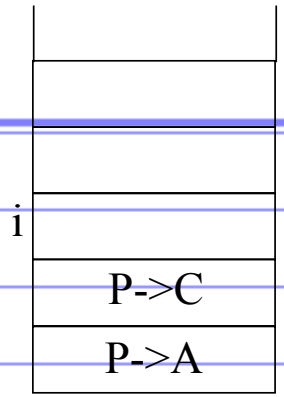



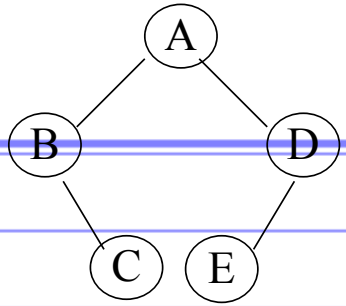



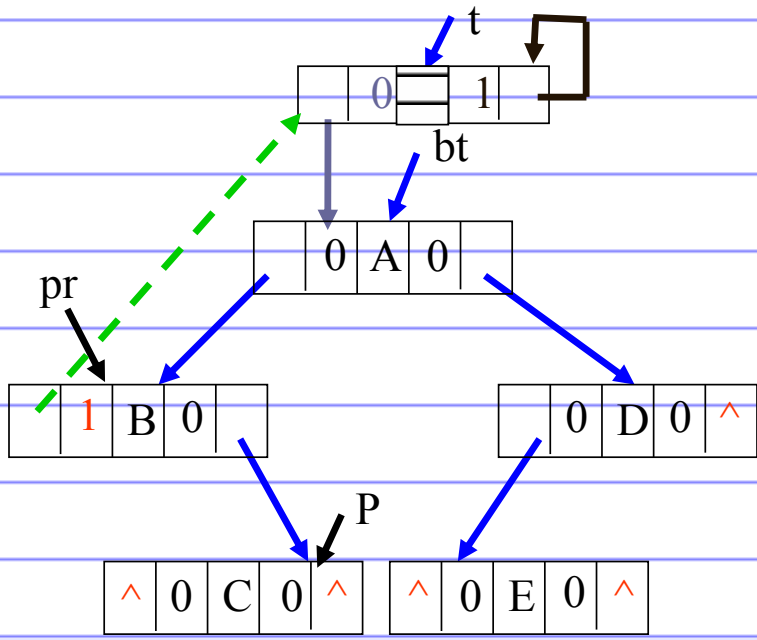

输出:B

## ✧算法

### ?按中序线索化二叉树



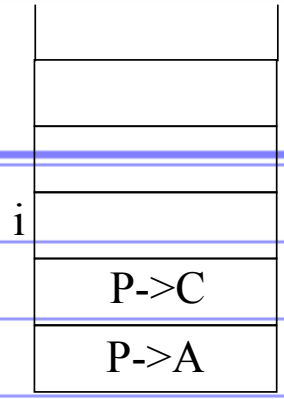



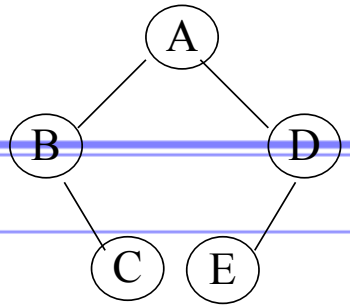



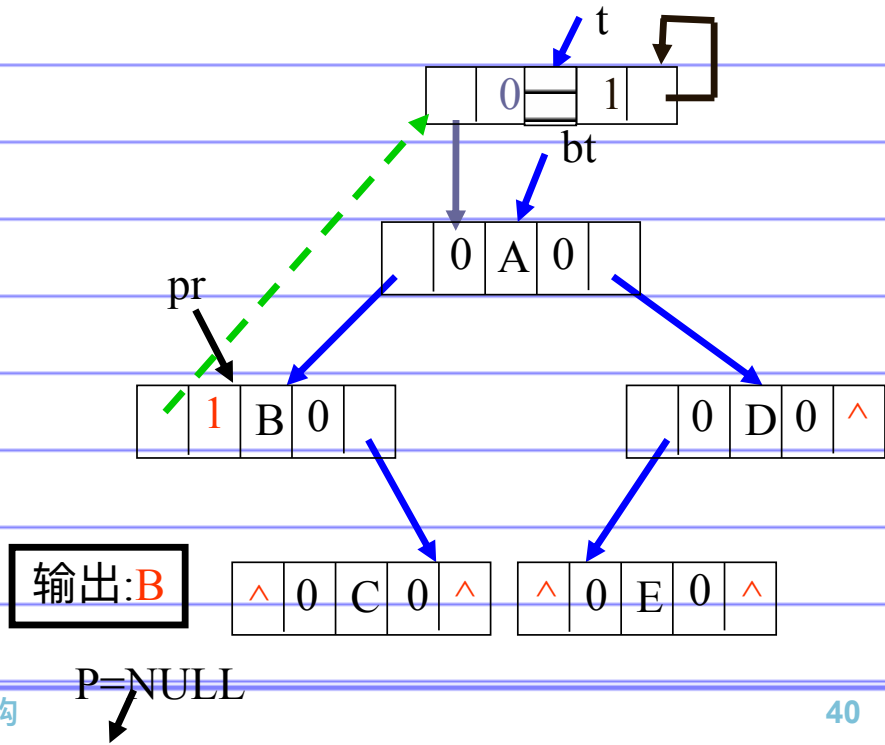

输出:B

## ✧算法

### ?按中序线索化二叉树

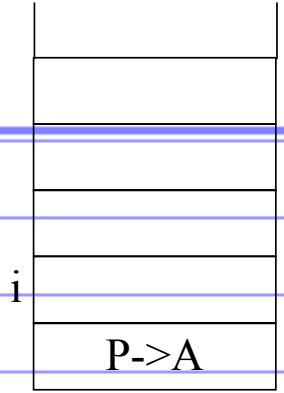


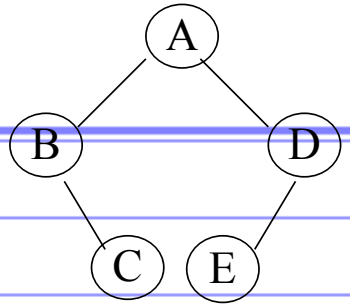



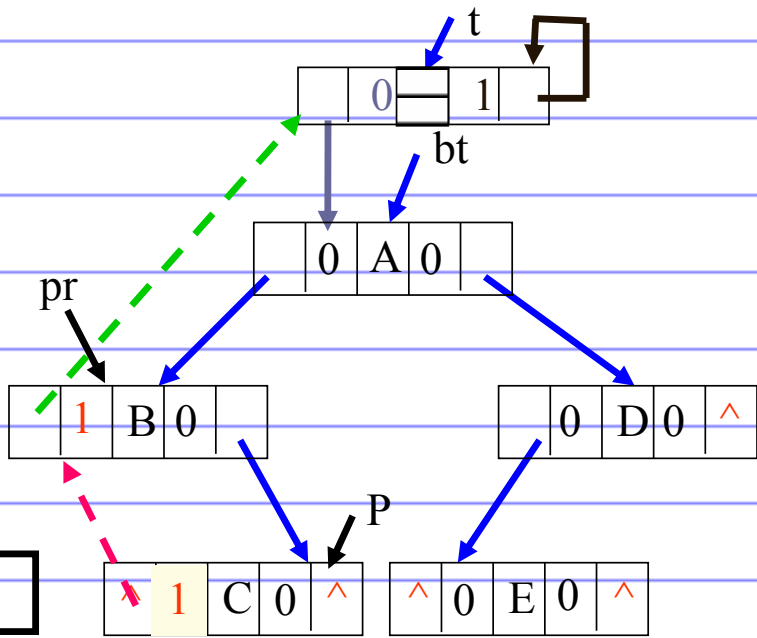



## ✧算法

### ?按中序线索化二叉树



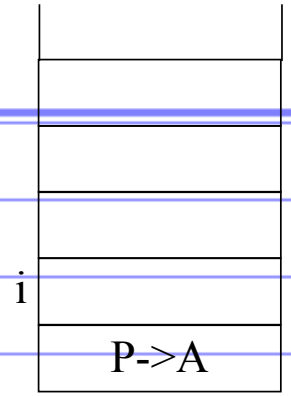



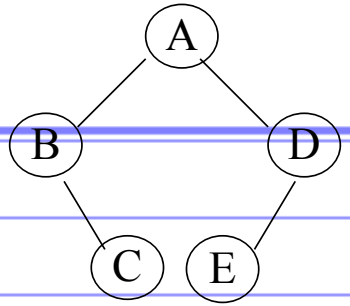



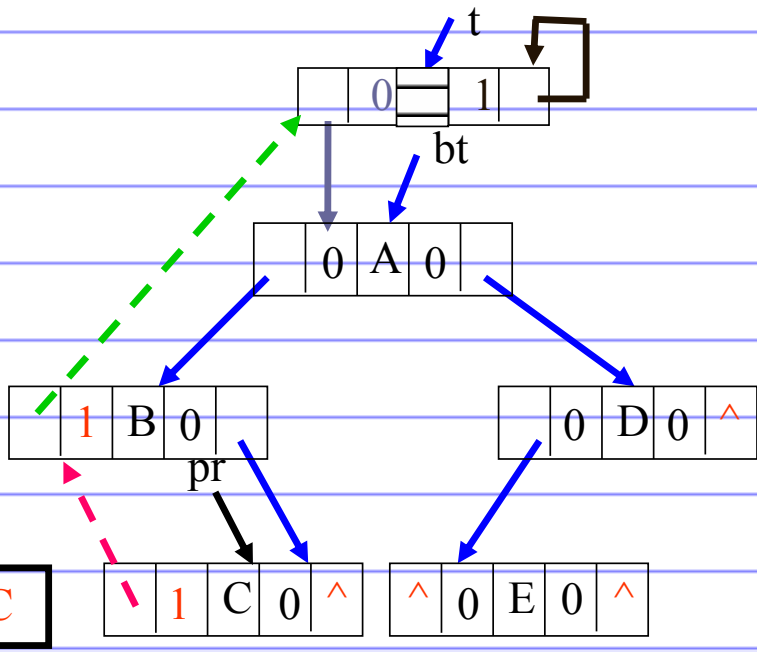

输出: B C

## ✧算法

### ?按中序线索化二叉树



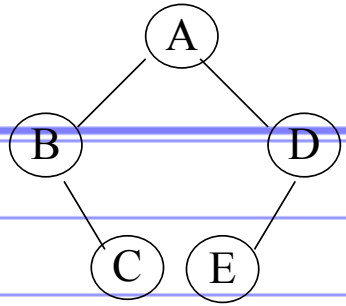



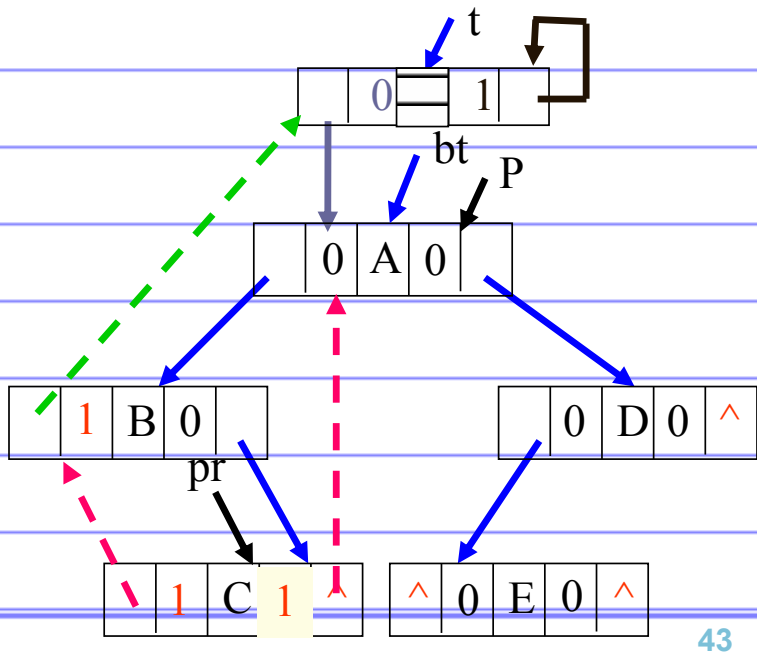




输出: B C

## ✧算法

### ?按中序线索化二叉树

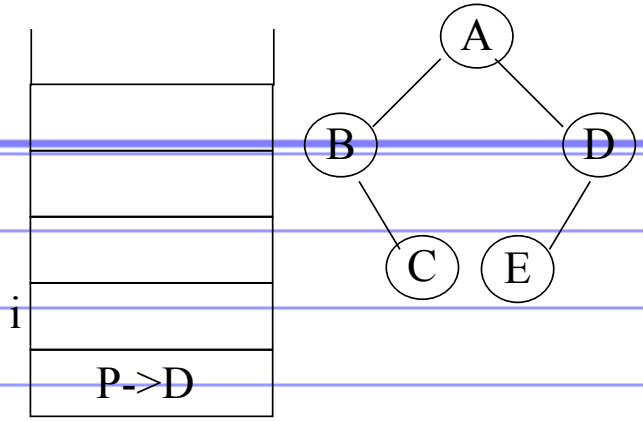



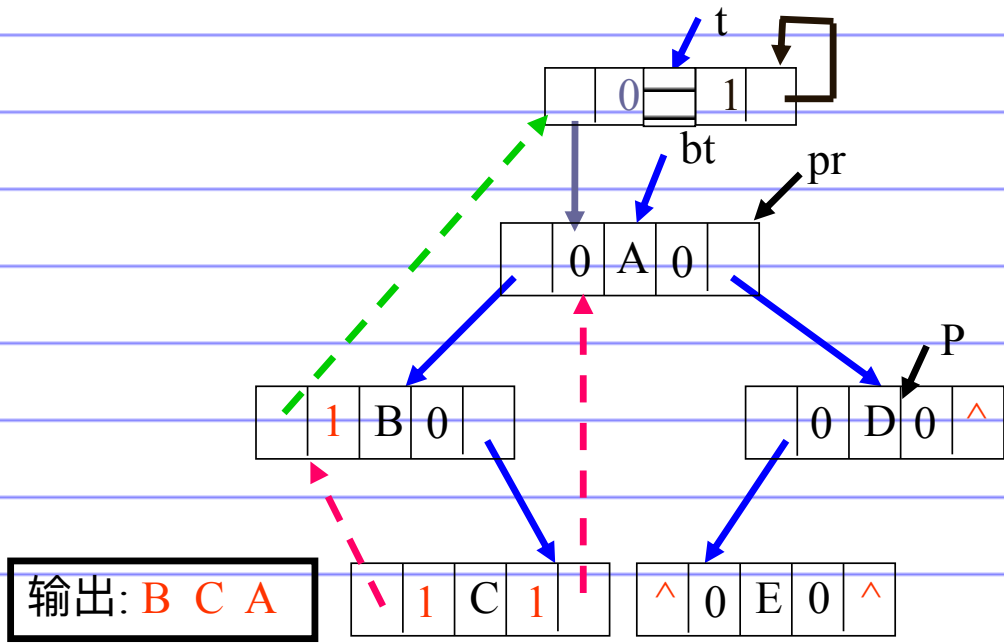




输出: B C A

## ✧算法

### ?按中序线索化二叉树

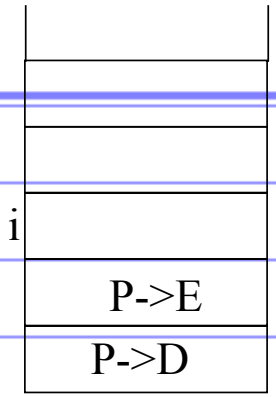


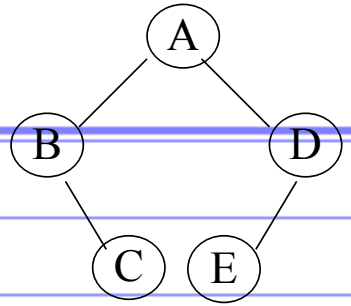



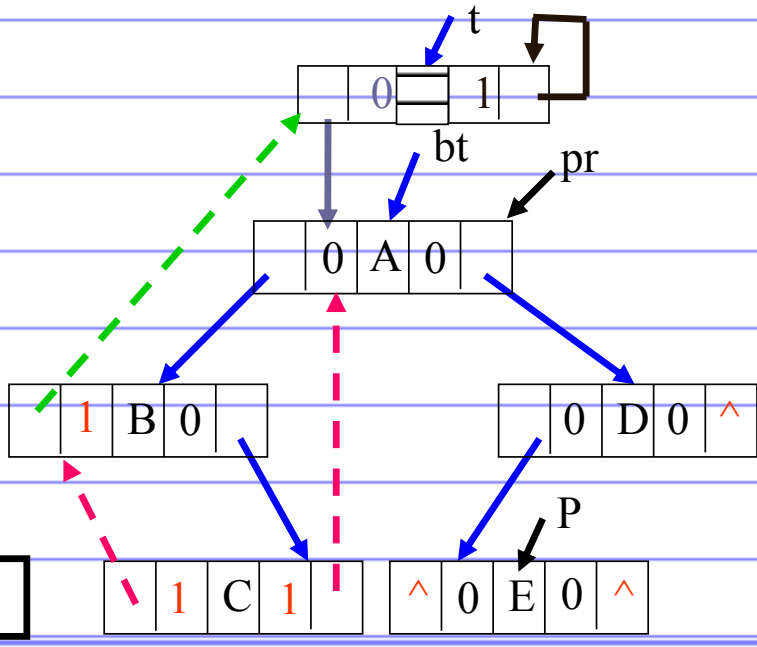

输出: B C A

## ✧算法

### ?按中序线索化二叉树



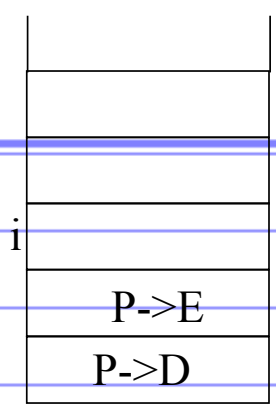



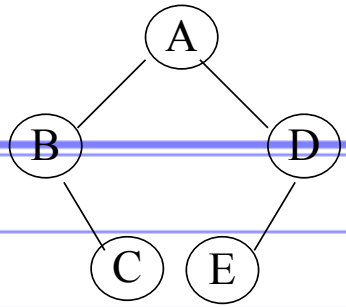



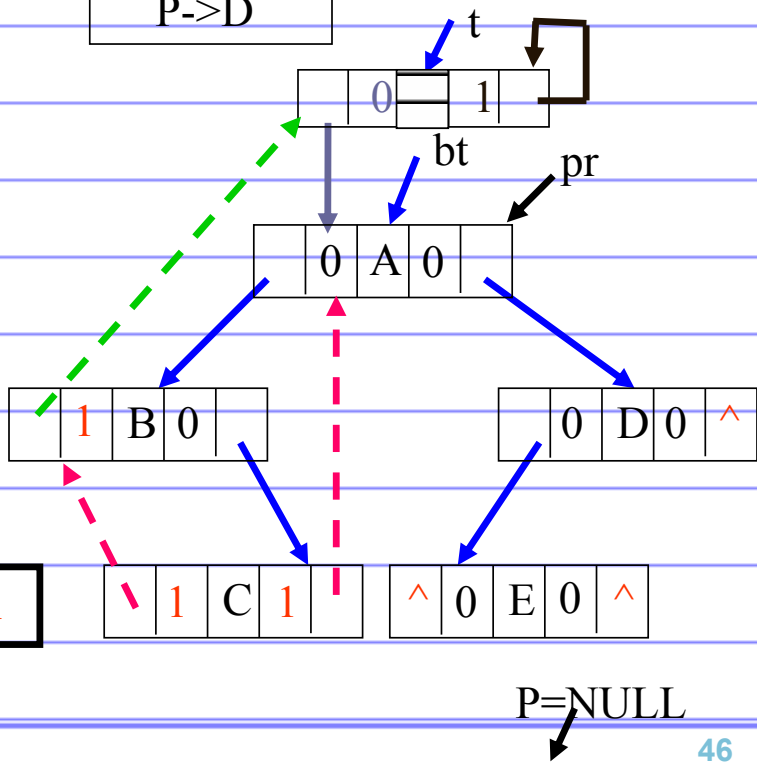

输出: B C A

## ✧算法

### ?按中序线索化二叉树



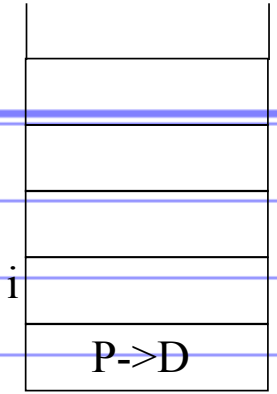



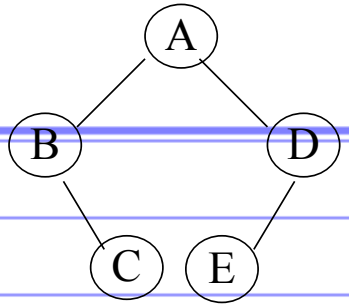



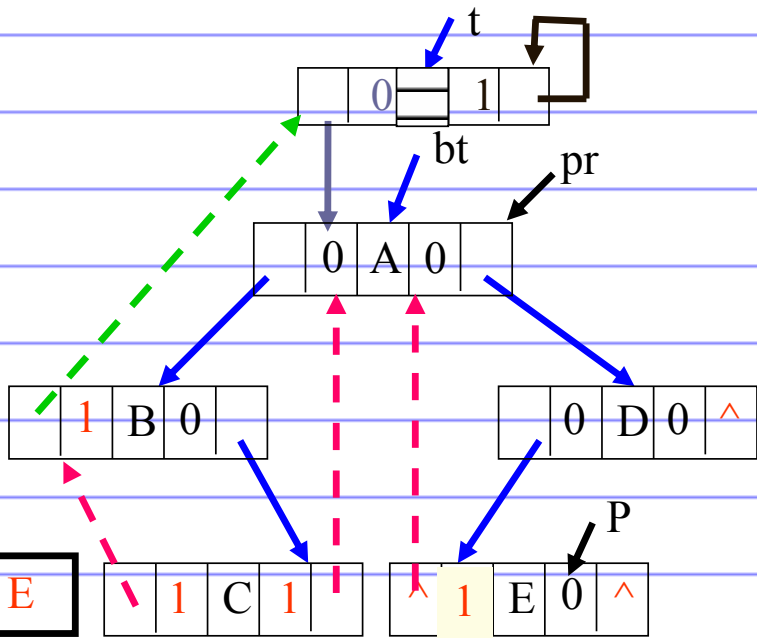

输出: B C A

## ✧算法

### ?按中序线索化二叉树



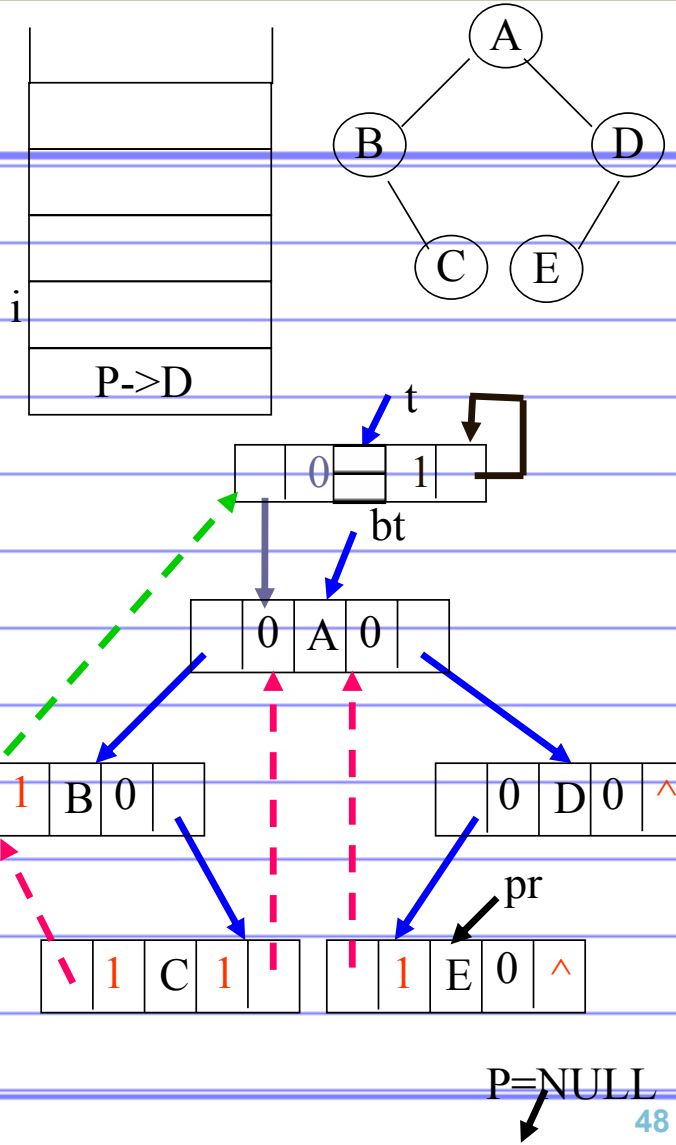




输出: B C A E

## ✧算法

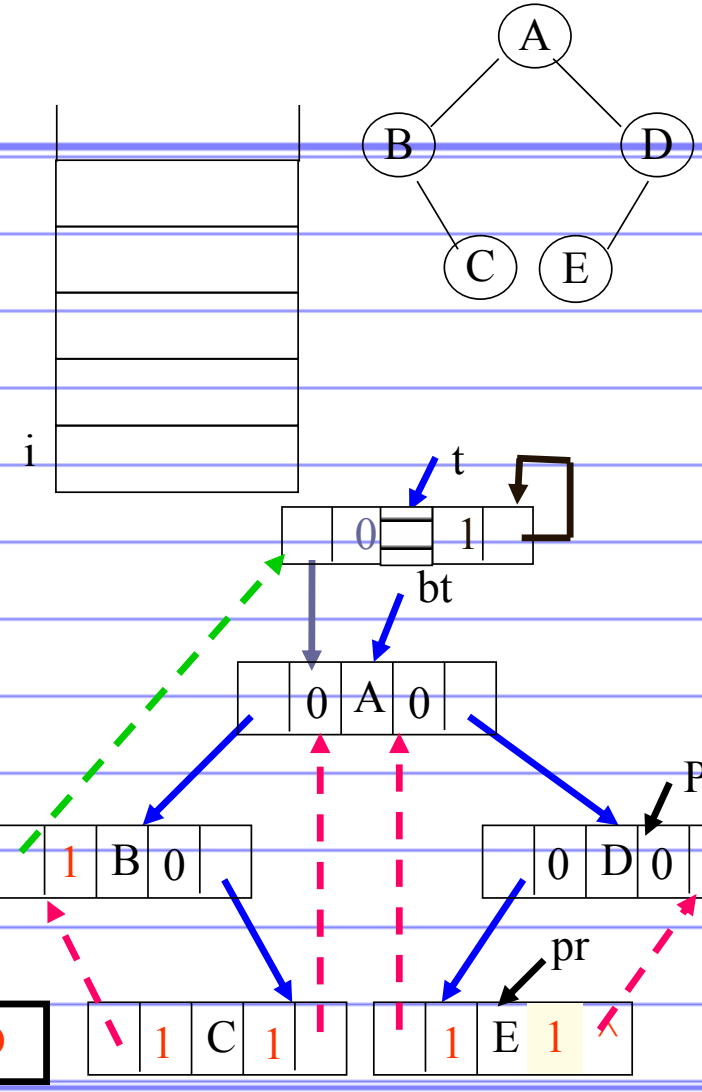
### ?按中序线索化二叉树




输出: B C A E

## ✧算法

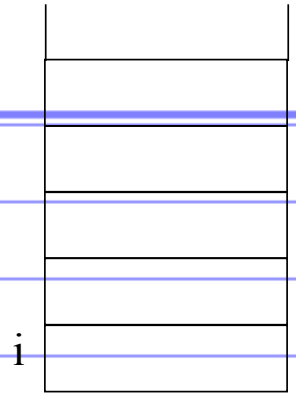
### ?按中序线索化二叉树

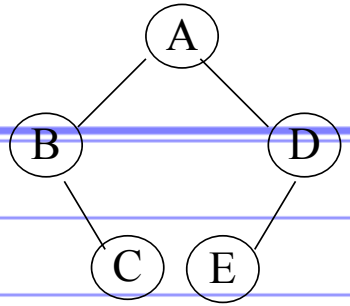


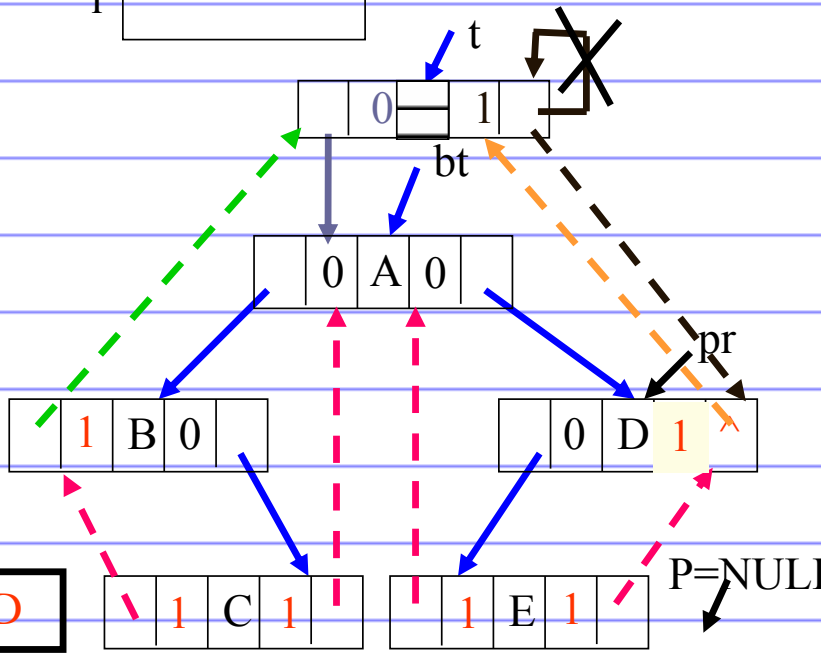

输出: B C A E D

## ✧算法

### ?按中序线索化二叉树



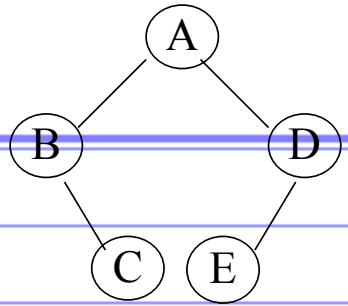



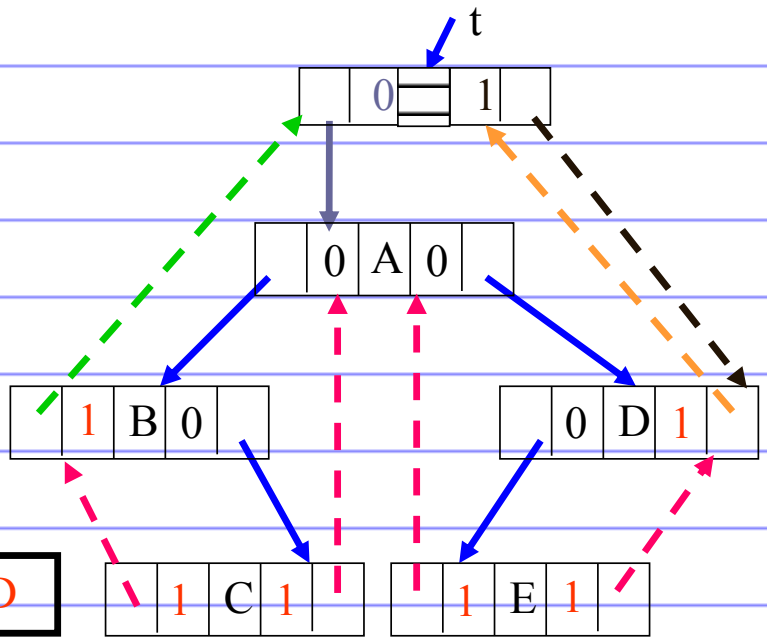




输出: B C A E D

## ✧算法

### ?按中序线索化二叉树

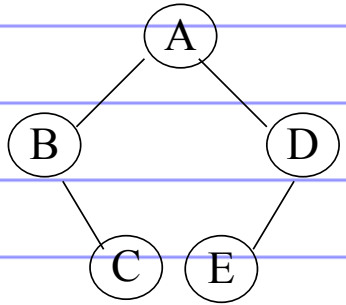


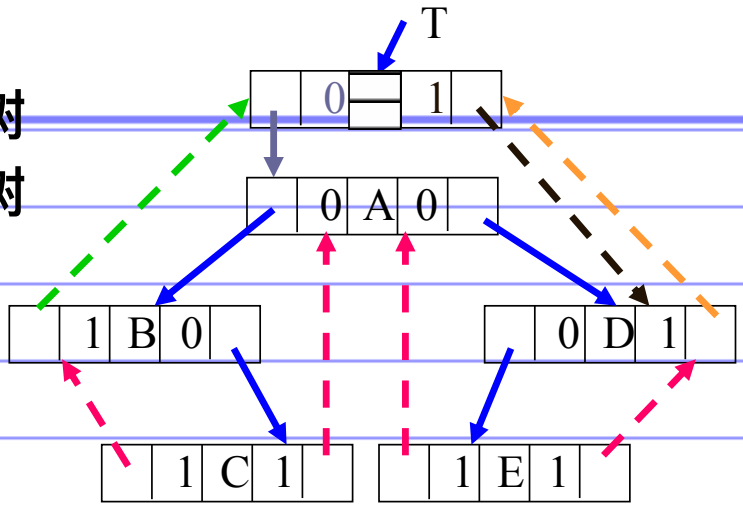




输出: B C A E D

## ✧算法

- 按中序线索化二叉树
- 遍历中序线索二叉树




中序序列：BCAED

带头结点的中序线索二叉树

在中序线索二叉树中找结点后继的方法：

（1）若rt=1, 则rc域直接指向其后继

（2）若rt=0, 则结点的后继应是其右子树的左链尾（lt=1)的结点

在中序线索二叉树中找结点前驱的方法：

（1）若lt=1, 则lc域直接指向其前驱

（2）若lt=0, 则结点的前驱应是其左子树的右链尾（rt=1)的结点

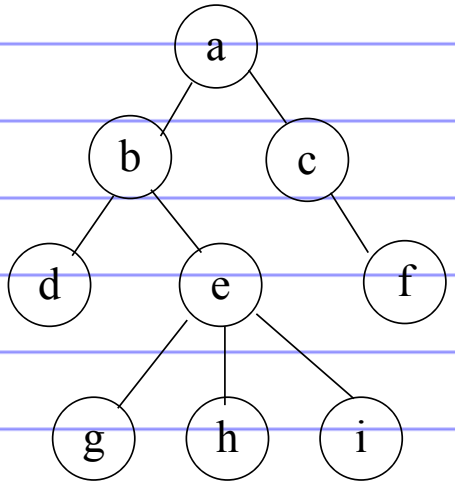
# 第三节 树的存储结构

- 双亲表示法
  - 实现:定义结构数组存放树的结点,每个结点含两个域:
    - 数据域:存放结点本身信息
    - 双亲域:指示本结点的双亲结点在数组中位置
  - 特点:找双亲容易，找孩子难
- 孩子表示法
  - 多重链表:每个结点有多个指针域,分别指向其子树的根
    - 结点同构：结点的指针个数相等，为树的度D
    - 结点不同构：结点指针个数不等，为该结点的度d
  - 孩子链表：每个结点的孩子结点用单链表存储，再用含n个元素的结构数组指向每个孩子链表
- 父子表示法

# 树的双亲表示法

```
#define MAX_TREE_SIZE 100
typedef  struct PTNode {    //节点结构
    TElemType  data;
    int              parent;    //双亲位置域
} PTNode;
typedef struct {   //树结构
    PTNode nodes[MAX_TREE_SIZE];
    int  r, n;          //根的位置和节点数
} PTree;
```

# 双亲表示法例



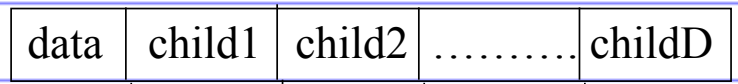

| | data | parent |
|---|---|---|
| 0 | 0 | 9 |
| 1 | a | 0 |
| 2 | b | 1 |
| 3 | c | 1 |
| 4 | d | 2 |
| 5 | e | 2 |
| 6 | f | 3 |
| 7 | g | 5 |
| 8 | h | 5 |
| 9 | i | 5 |

0号单元不用或存结点个数

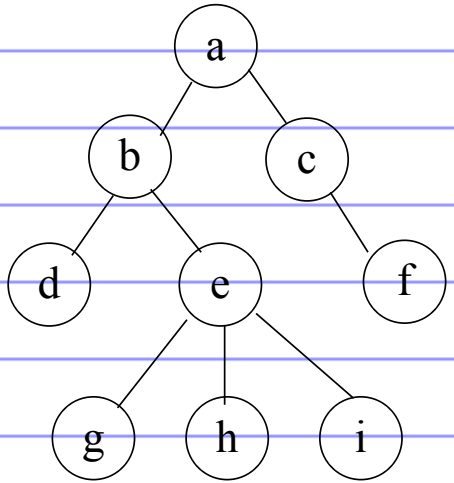
如何找孩子结点

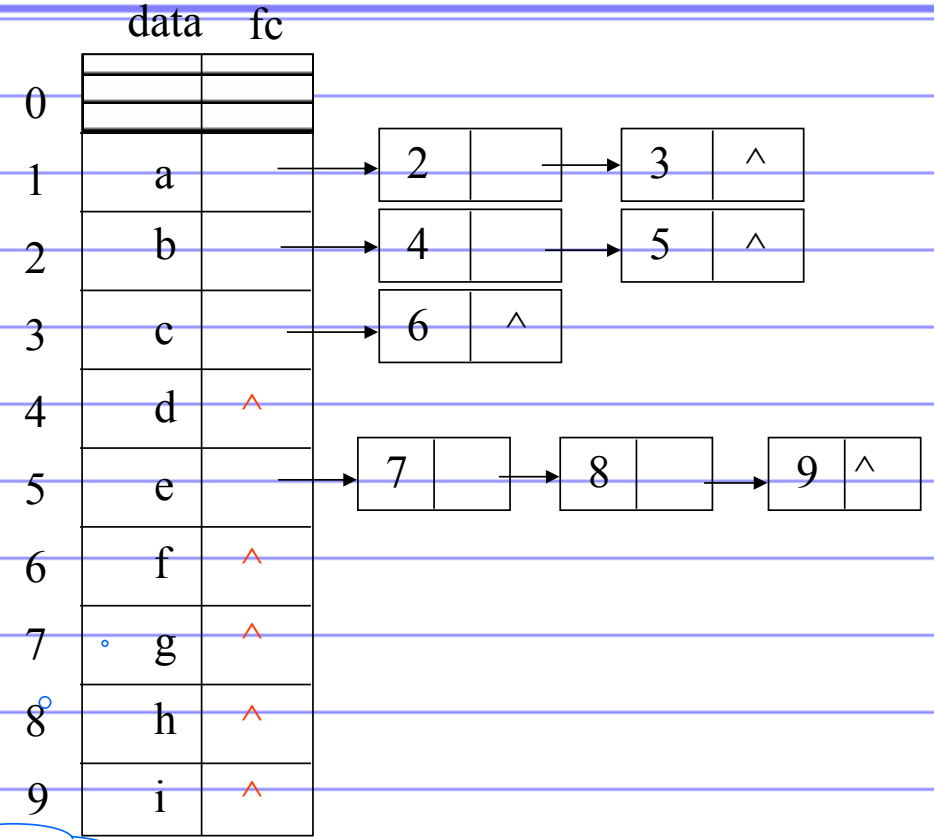
# 树的孩子表示法

```
typedef  struct CTNode {    //孩子节点
    int  child;
    struct CTNode *next;
} *ChildPtr;
typedef struct {
    TElemType  data;
    ChildPtr firstchild;    //孩子链表头指针
} CTBox;
typedef struct {
    CTBox nodes[MAX_TREE_SIZE];
    int n, r;
} CTree;
```



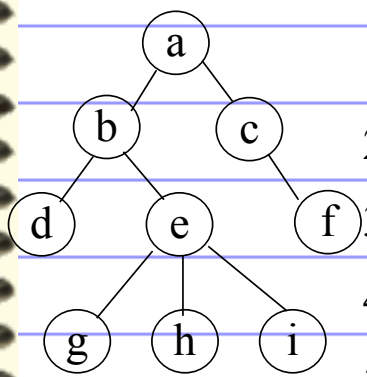
# 孩子表示法例



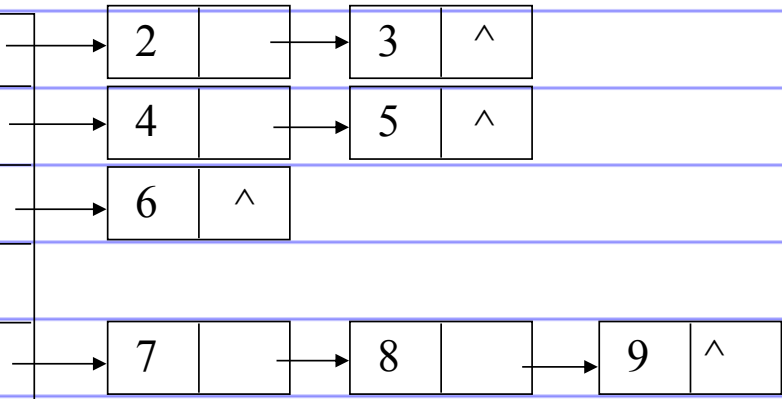




如何找双亲结点

# 带双亲的孩子链表




| | data | parent | fc |
|---|---|---|---|
| 1 | a | 0 | |
| 2 | b | 1 | |
| 3 | c | 1 | |
| 4 | d | 2 | ^ |
| 5 | e | 2 | |
| 6 | f | 3 | ^ |
| 7 | g | 5 | ^ |
| 8 | h | 5 | ^ |
| 9 | i | 5 | ^ |

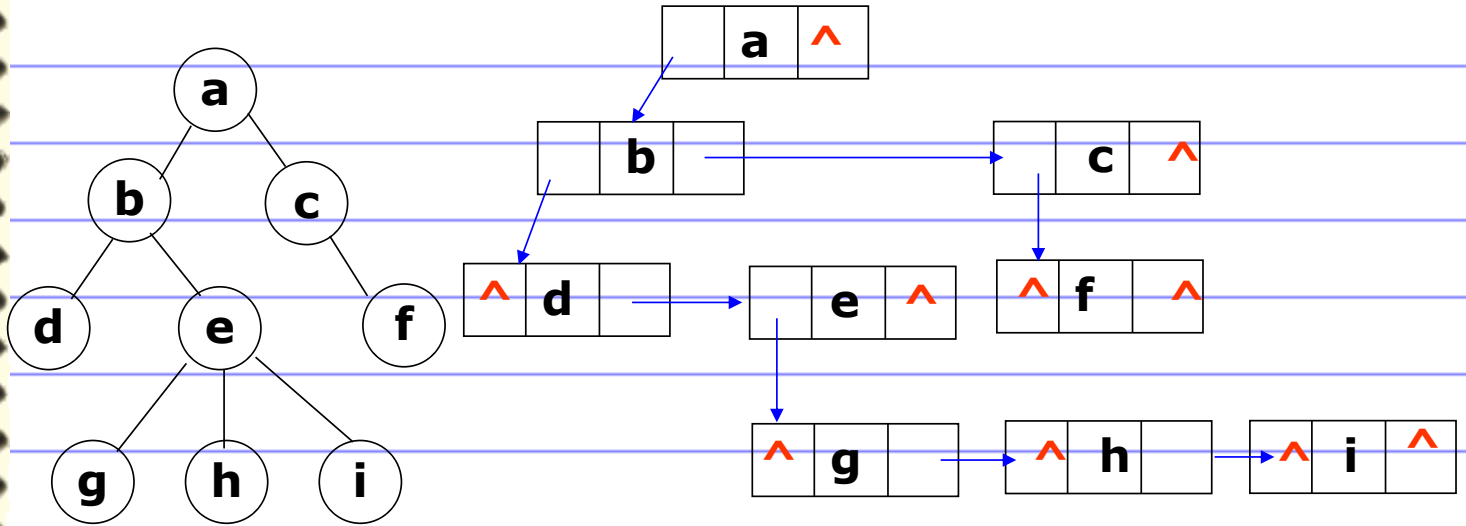


# 树的父子兄弟表示法

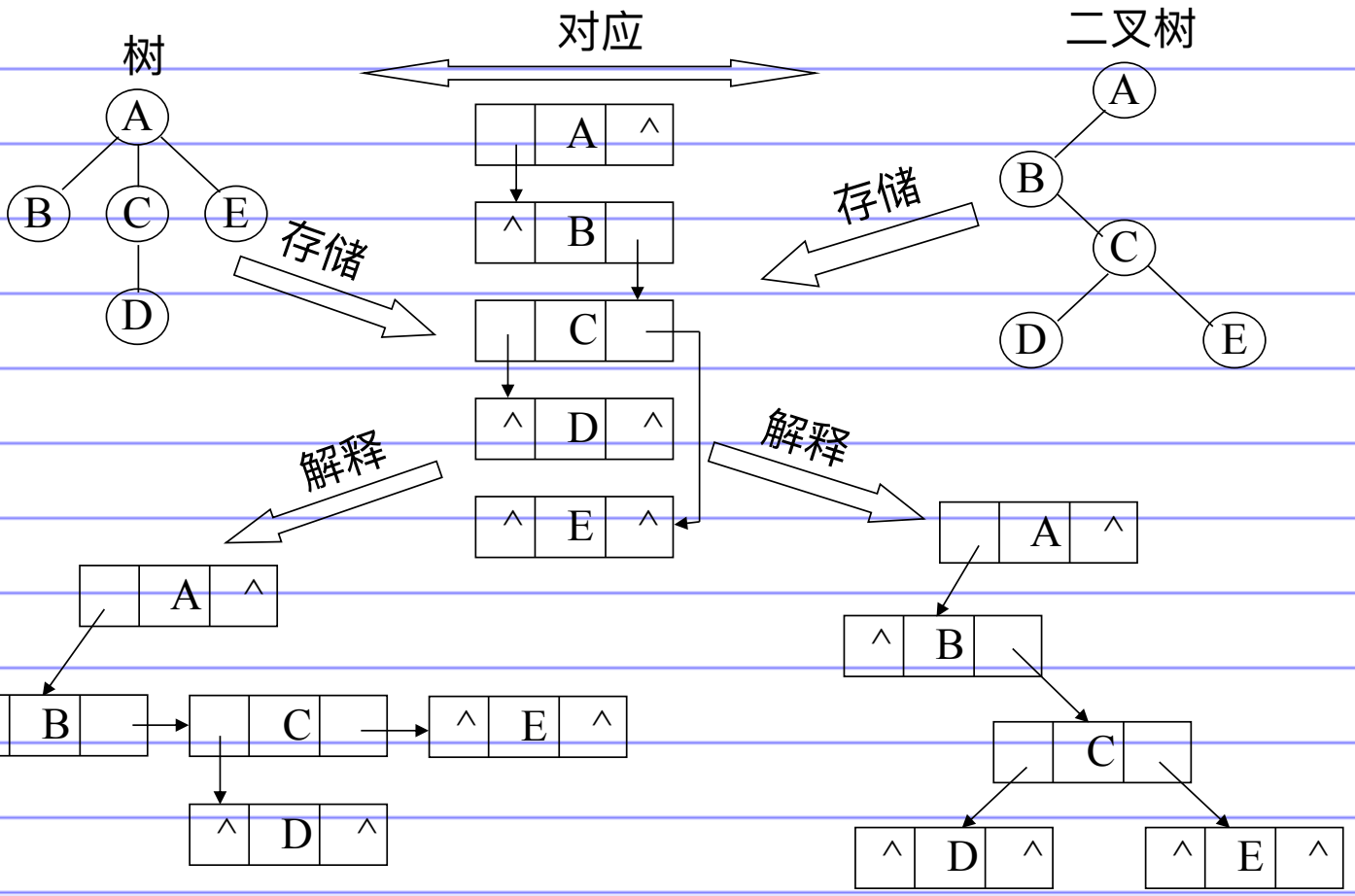
```
typedef  struct CSNode
{
    ElemType  data;
    struct CSNode *firstchild, *nextsibling;
} CSNode, *CSTree;
```

# 树的父子兄弟表示法

- 实现：用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点
- 特点
  - 操作容易
  - 破坏了树的层次

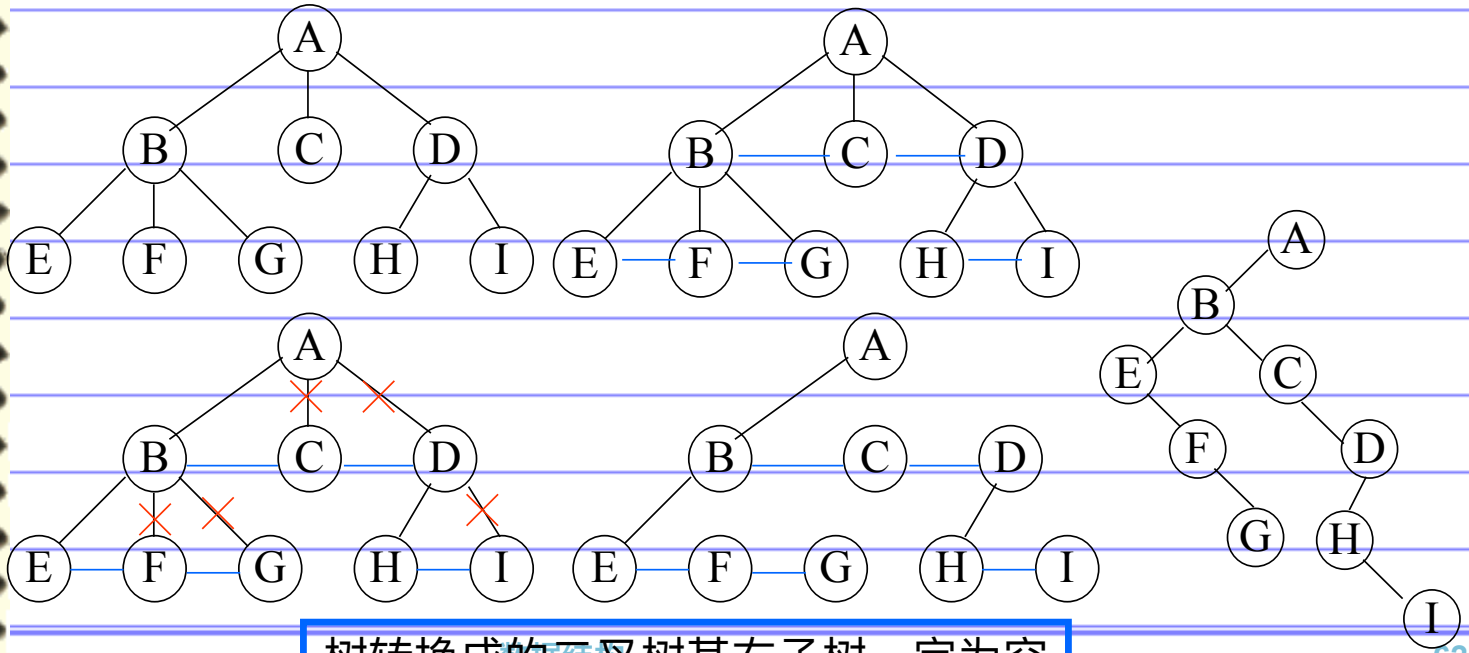


# 树与二叉树转换



# 将树转换成二叉树的步骤

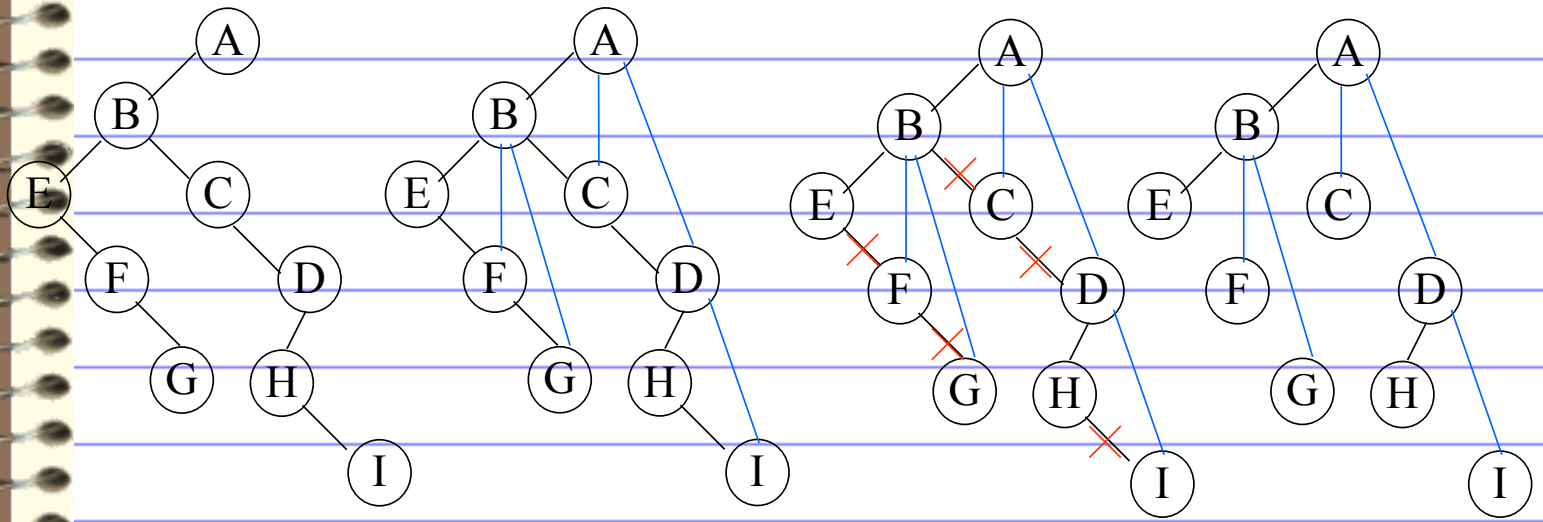
- 加线：在兄弟之间加一连线
- 抹线：对每个结点，除了其左孩子外，去除其与其余孩子之间的关系
- 旋转：以树的根结点为轴心，将整树顺时针转45°




树转换成的二叉树其右子树一定为空

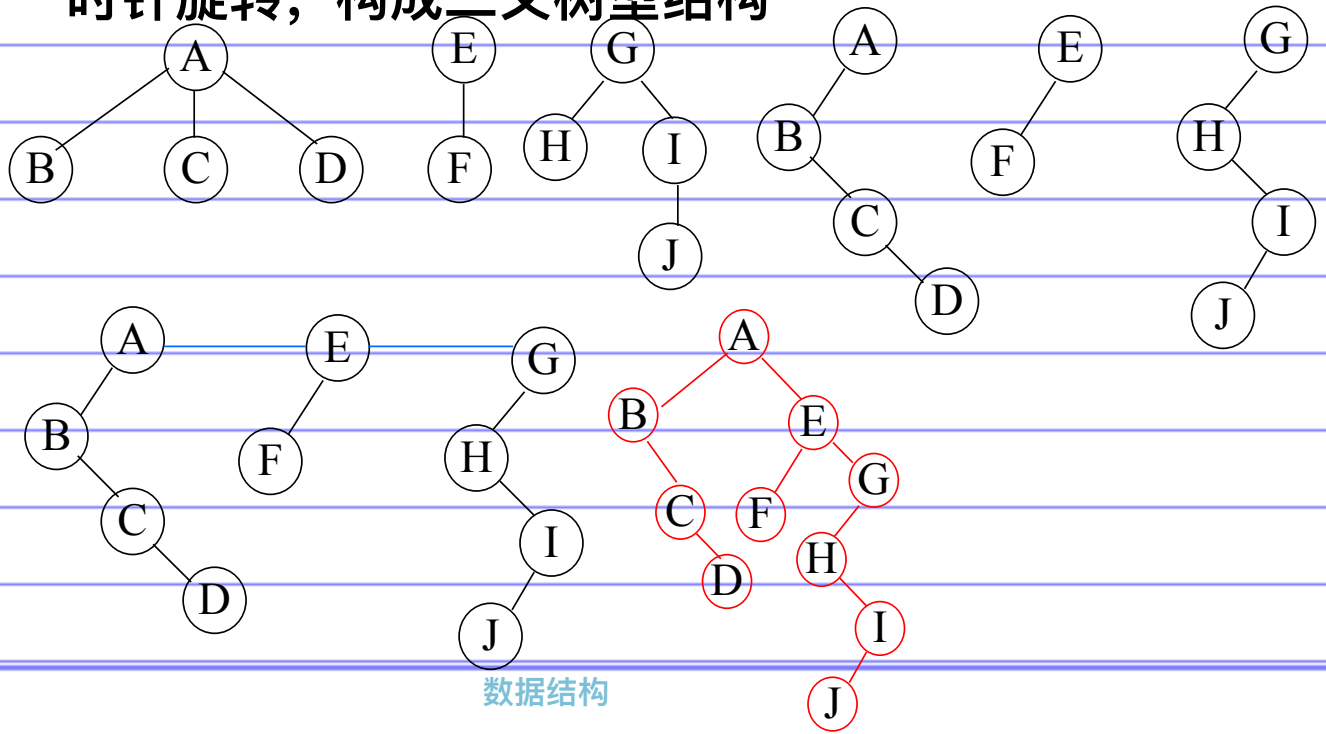
# 将二叉树转换成树的步骤

- 加线：若p结点是双亲结点的左孩子，则将p的右孩子，右孩子的右孩子，……沿分支找到的所有右孩子，都与p的双亲用线连起来
- 抹线：抹掉原二叉树中双亲与右孩子之间的连线
- 调整：将结点按层次排列，形成树结构



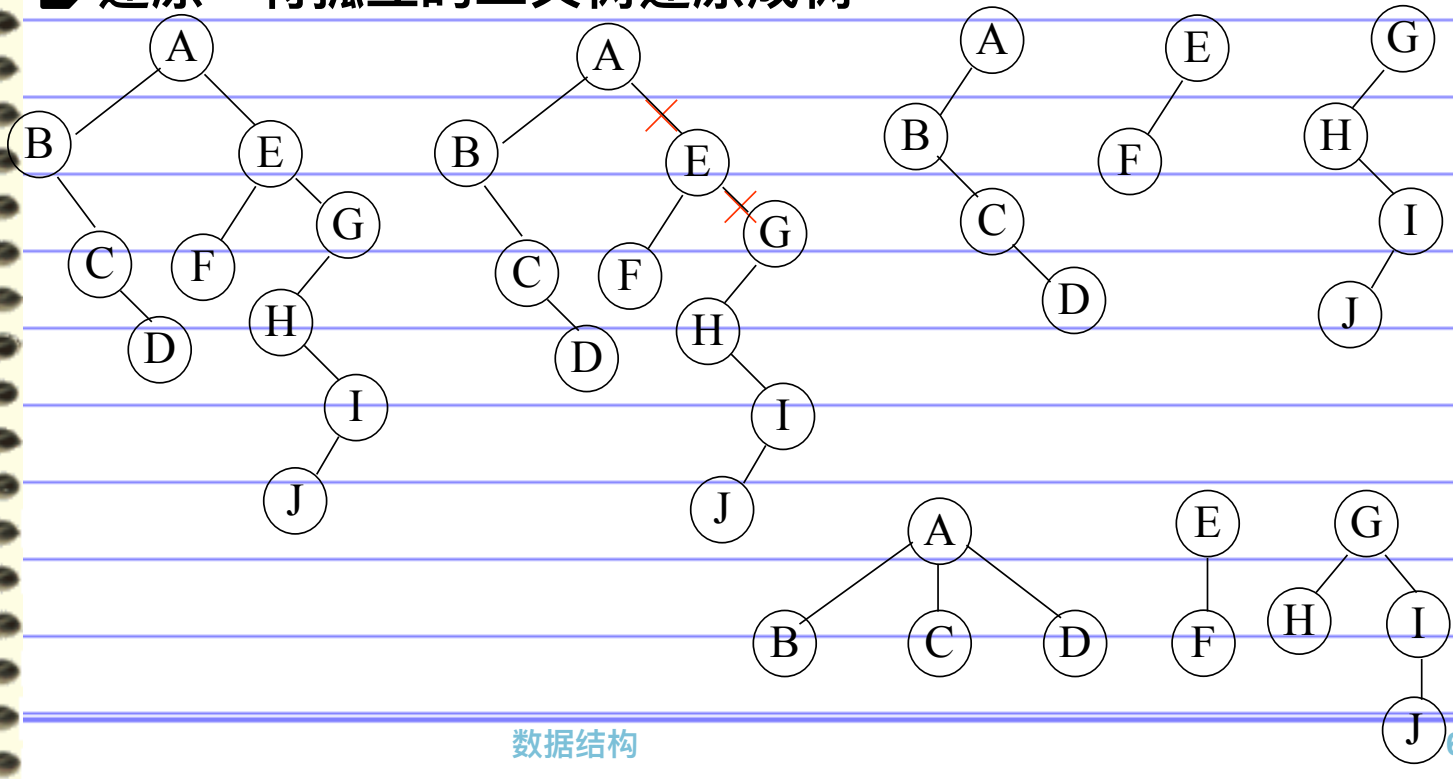
# 将森林转换成二叉树的步骤

- 将各棵树分别转换成二叉树
- 将每棵树的根结点用线相连
- 以第一棵树根结点为二叉树的根，再以根结点为轴心，顺时针旋转，构成二叉树型结构



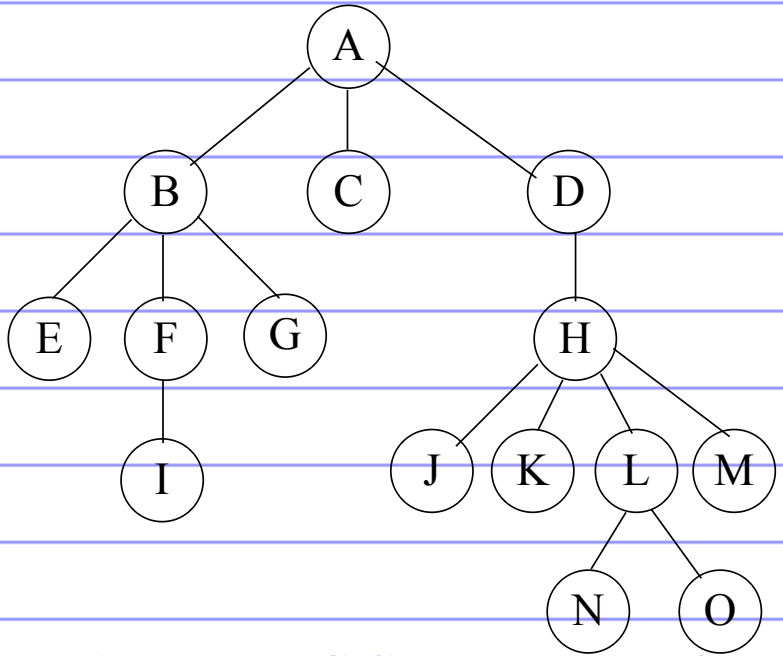
# 将二叉树转换成森林的步骤

- 抹线：将二叉树中根结点与其右孩子连线，及沿右分支搜索到的所有右孩子间连线全部抹掉,使之变成孤立的二叉树
- 还原：将孤立的二叉树还原成树



# 第四节 树的遍历

- 树的遍历——按一定规律走遍树的各个顶点，且使每一顶点仅被访问一次，即找一个完整而有规律的走法，以得到树中所有结点的一个线性排列
- 常用方法
  - 先根（序）遍历：先访问树的根结点，然后依次先根遍历根的每棵子树
  - 后根（序）遍历：先依次后根遍历每棵子树，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：A B E F I G C D H J K L N O M

后序遍历：E I F G B C J K N O L M H D A

层次遍历：A B C D E F G H I J K L M N O

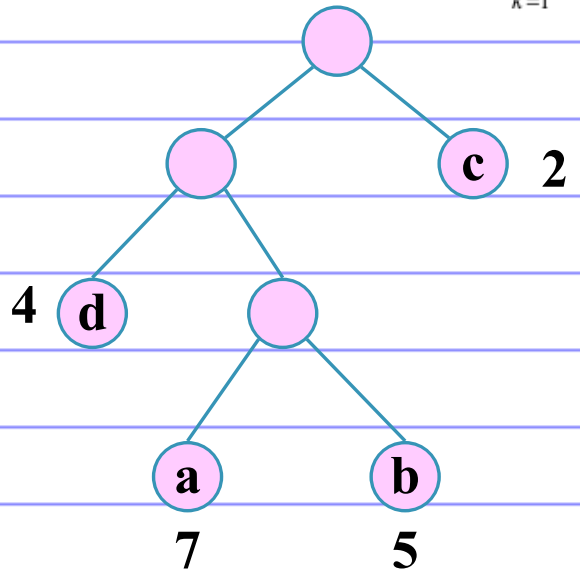
# 5.5 二叉树的应用

- 哈夫曼树(Huffman)——带权路径长度最短的树
  - 路径：从树中一个结点到另一个结点之间的分支构成这两个结点间的~
  - 路径长度：路径上的分支数
  - 树的路径长度：从树根到每一个结点的路径长度之和
  - 树的带权路径长度：树中所有带权结点的路径长度之和
  - Huffman树——设有n个权值$\{w_1,w_2,\ldots\ldots w_n\}$，构造一棵有n个叶子结点的二叉树，每个叶子的权值为$w_i$，则wpl最小的二叉树叫~

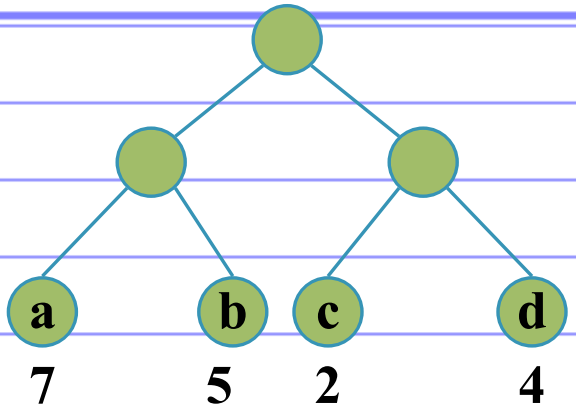
$$WPL=\sum_{k=1}^{n}W_K L_K$$

例　有4个结点，权值分别为7，5，2，4，构造有4个叶子结点的二叉树

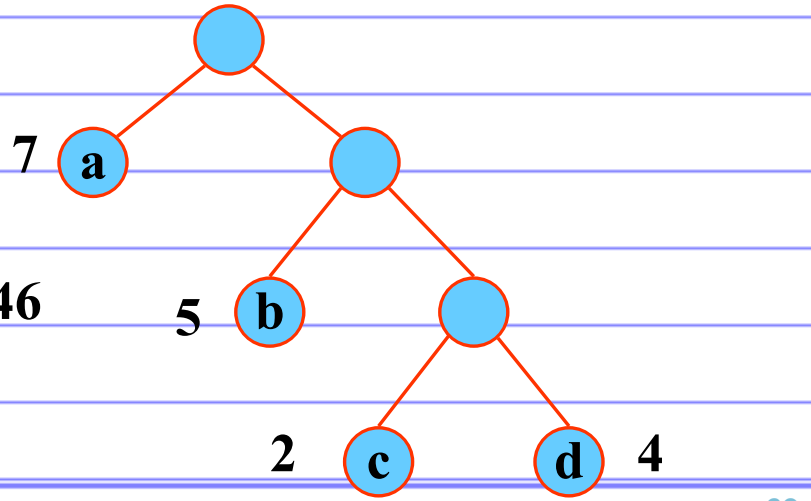
$$WPL=\sum_{k=1}^{n}W_K L_K$$




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



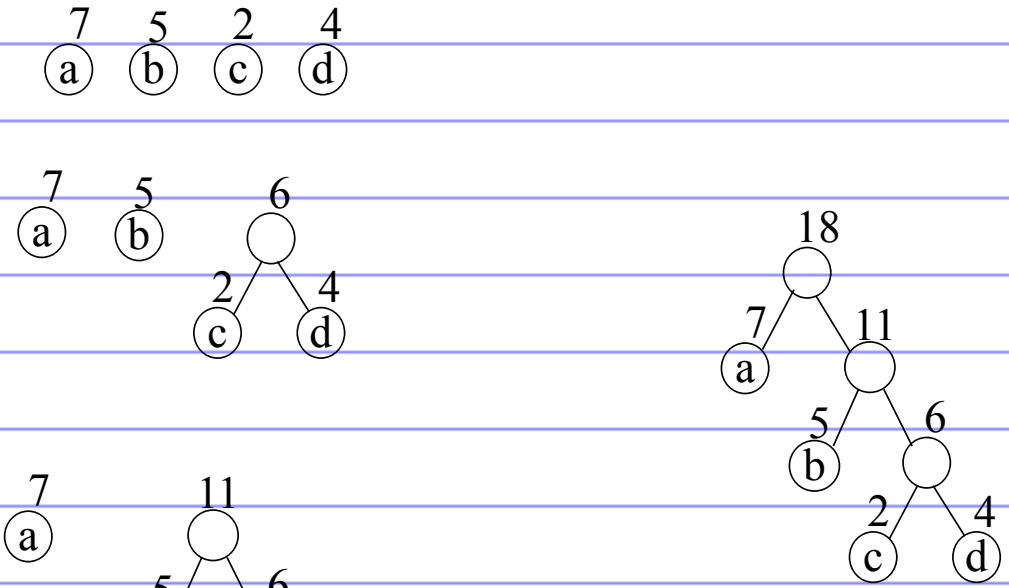

WPL=7*2+5*2+2*2+4*2=36



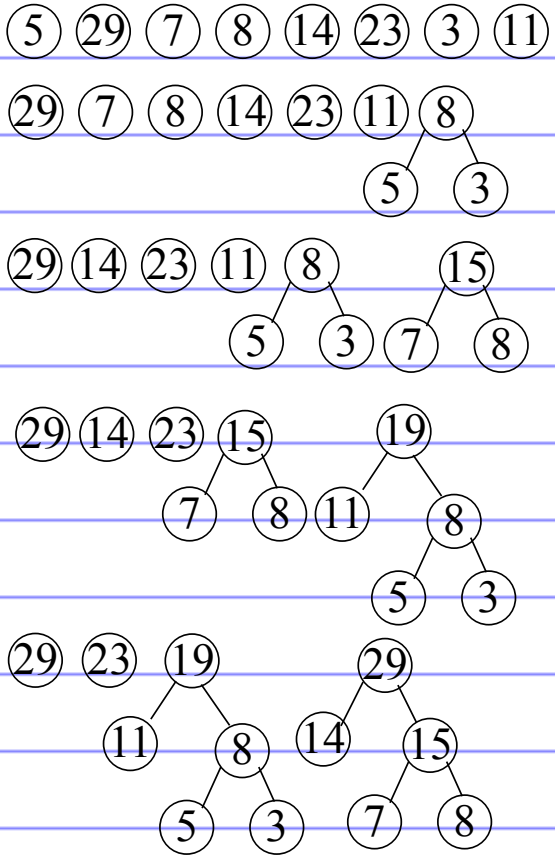

WPL=7*1+5*2+2*3+4*3=35

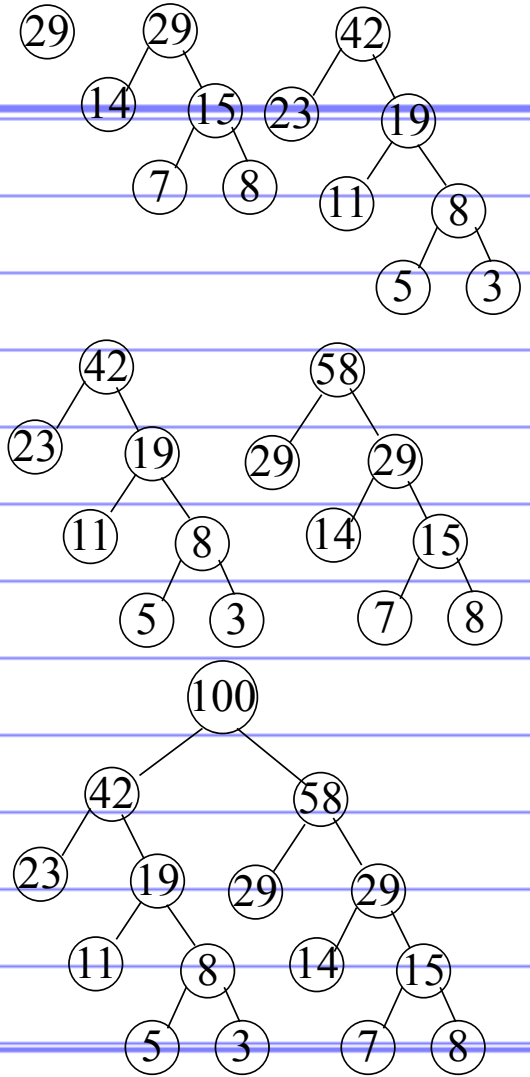
- 构造Huffman树的方法——Huffman算法
  - 构造Huffman树步骤
    - 根据给定的n个权值$\{w_1,w_2,\ldots\ldots w_n\}$，构造n棵只有根结点的二叉树，令其权值为$w_j$
    - 在森林中选取两棵根结点权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和
    - 在森林中删除这两棵树，同时将新得到的二叉树加入森林中
    - 重复上述两步，直到只含一棵树为止，这棵树即哈夫曼树

例



例 w={5, 29, 7, 8, 14, 23, 3, 11}

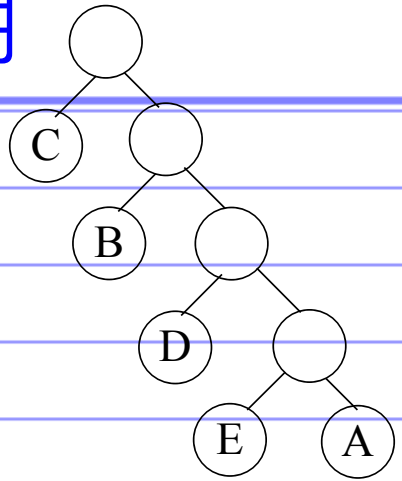


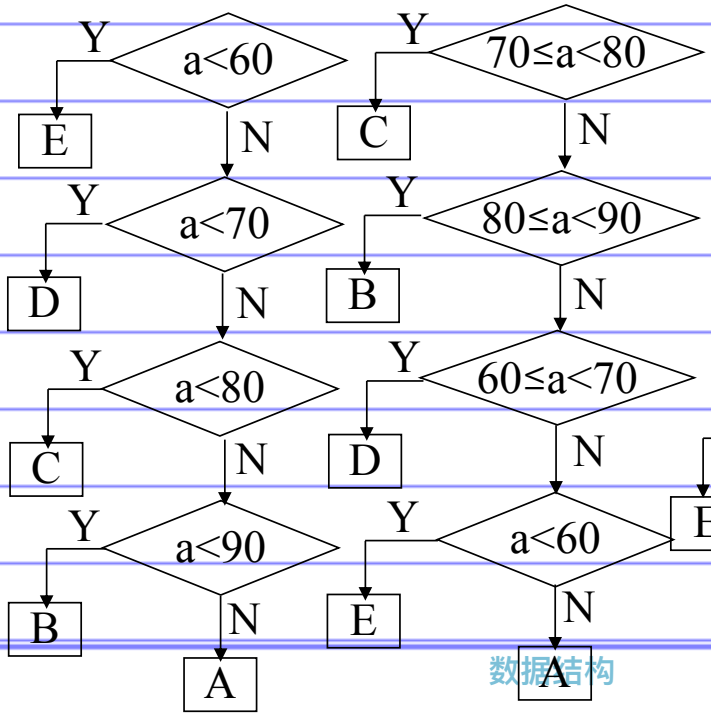



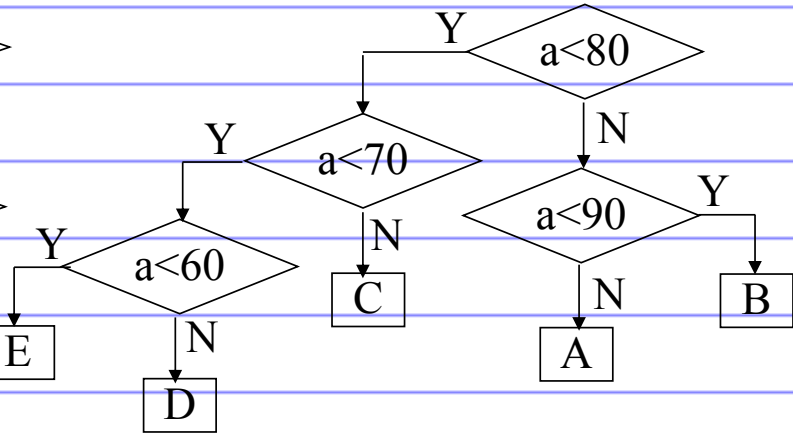
# Huffman树应用

## 最佳判定树

| 等级 | E | D | C | B | A |
|---|---|---|---|---|---|
| 分数段 | 0~59 | 60~69 | 70~79 | 80~89 | 90~100 |
| 比例 | 0.05 | 0.15 | 0.40 | 0.30 | 0.10 |



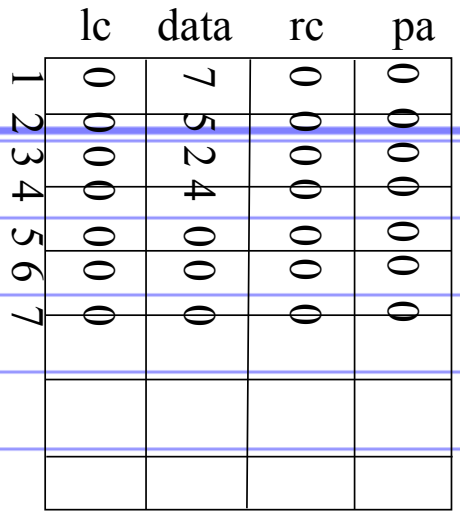



# Huffman算法实现

- 一棵有n个叶子结点的Huffman树有2n-1个结点
- 采用顺序存储结构——一维结构数组
- 结点类型定义

```
typedef  struct {
    unsigned int weight;
    unsigned int parent, lchild, rchild;
} HTNode, *HuffmanTree;
```

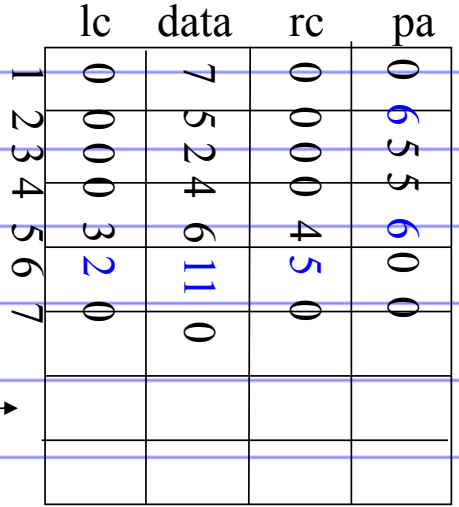



(1)


(2)

x1=3,x2=4

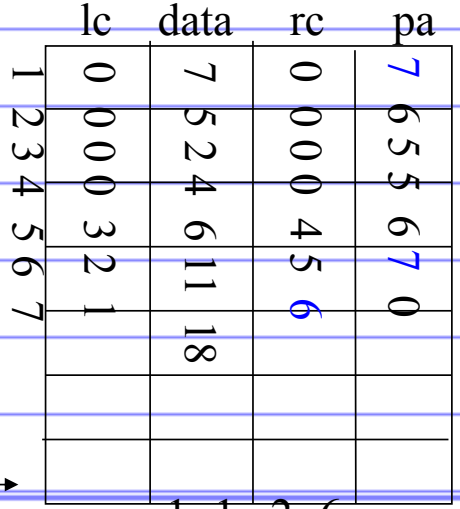
m1=2,m2=4




(3)

x1=2,x2=5

m1=5,m2=6

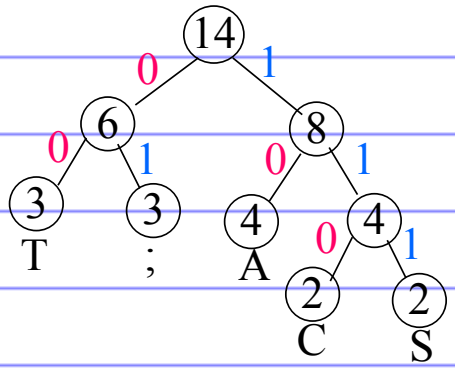



(4)

x1=1,x2=6

m1=7,m2=11

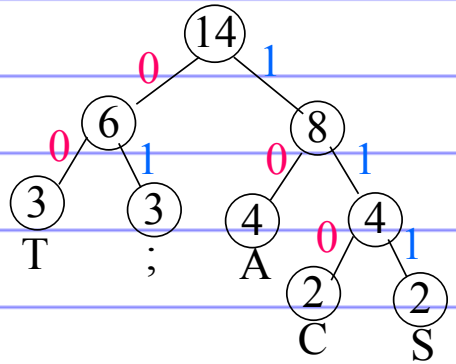
- Huffman编码：数据通信用的二进制编码
  - ✧ 思想：根据字符出现频率编码，使电文总长最短
  - ✧ 编码：根据字符出现频率构造Huffman树，然后将树中结点引向其左孩子的分支标“0”，引向其右孩子的分支标“1”；每个字符的编码即为从根到每个叶子的路径上得到的0、1序列




例 要传输的字符集 D={C,A,S,T, ; }
字符出现频率 w={2,4,2,3,3}

T : 00
; : 01
A : 10
C : 110
S : 111

✧ 译码：从Huffman树根开始，从待译码电文中逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束




T : 00
; : 01
A : 10
C : 110
S : 111

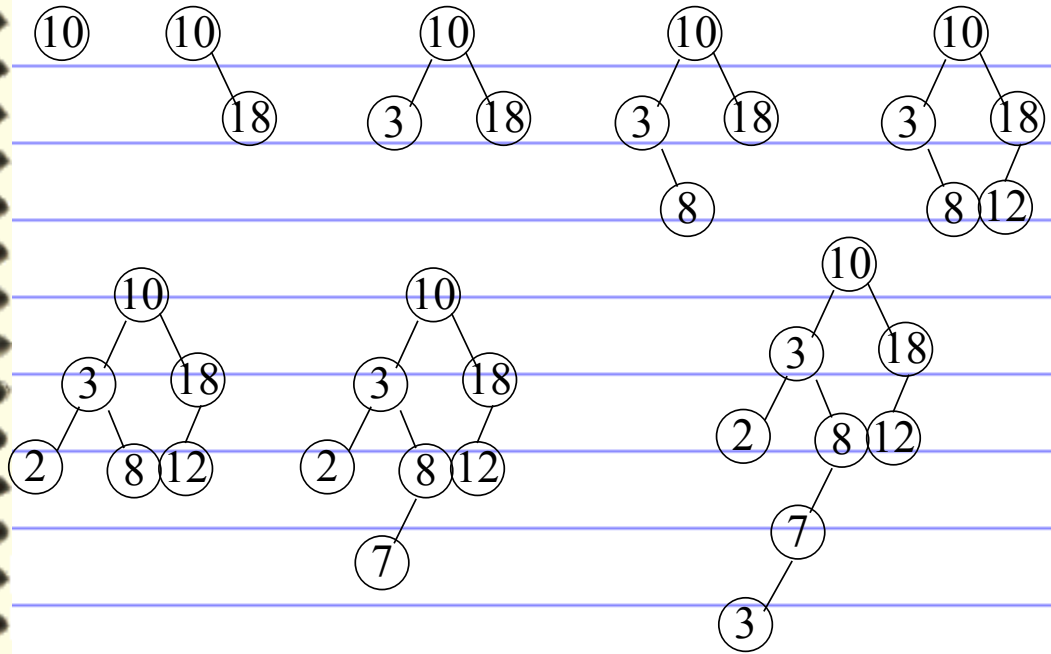
例 电文是{CAS;CAT;SAT;AT}
其编码 “1101011101110100001111100001100 0”
电文为“1101000”
译文只能是“CAT”

# 二叉排序树

- 定义：二叉排序树或是一棵空树，或是具有下列性质
  - 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值
  - 若它的右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值
  - 它的左、右子树也分别为二叉排序树
- 二叉排序树的插入
  - 插入原则：若二叉排序树为空，则插入结点应为新的根结点；否则，继续在其左、右子树上查找，直至某个叶子结点的左子树或右子树为空为止，则插入结点应为该叶子结点的左孩子或右孩子
  - 二叉排序树生成：从空树出发，经过一系列的查找、插入操作之后，可生成一棵二叉排序树

## 插入算法

例 {10, 18, 3, 8, 12, 2, 7, 3}




中序遍历二叉排序树可得到一个关键字的有序序列

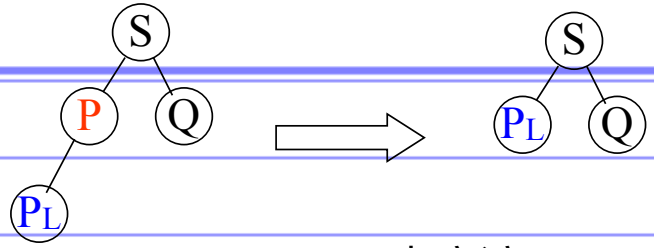
✧二叉排序树的删除

要删除二叉排序树中的p结点，分三种情况：

- p为叶子结点，只需修改p双亲f的指针

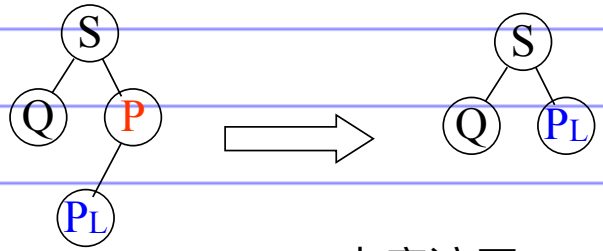
  f->lchild=NULL　　f->rchild=NULL

- p只有左子树或右子树
  - ✧p只有左子树，用p的左孩子代替p　　(1)(2)
  - ✧p只有右子树，用p的右孩子代替p　　(3)(4)
- p左、右子树均非空
  - ✧沿p左子树的根C的右子树分支找到S，S的右子树为空，将S的左子树成为S的双亲Q的右子树，用S取代p　(5)
  - ✧若C无右子树，用C取代p　　(6)



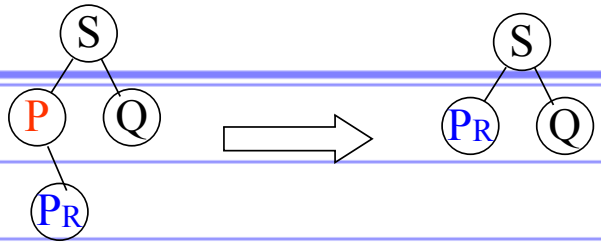

中序遍历：$P_L$ P S Q　　中序遍历：$P_L$ S Q

(1)



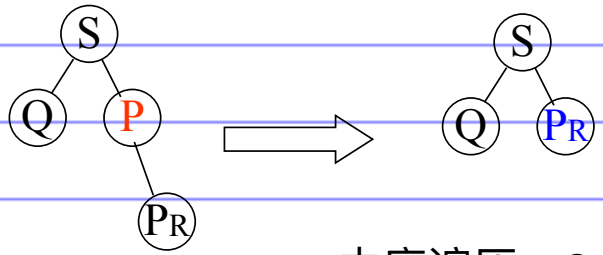

中序遍历：Q S $P_L$ P　　中序遍历：Q S $P_L$

(2)




中序遍历：P PR S Q　　中序遍历：PR S Q

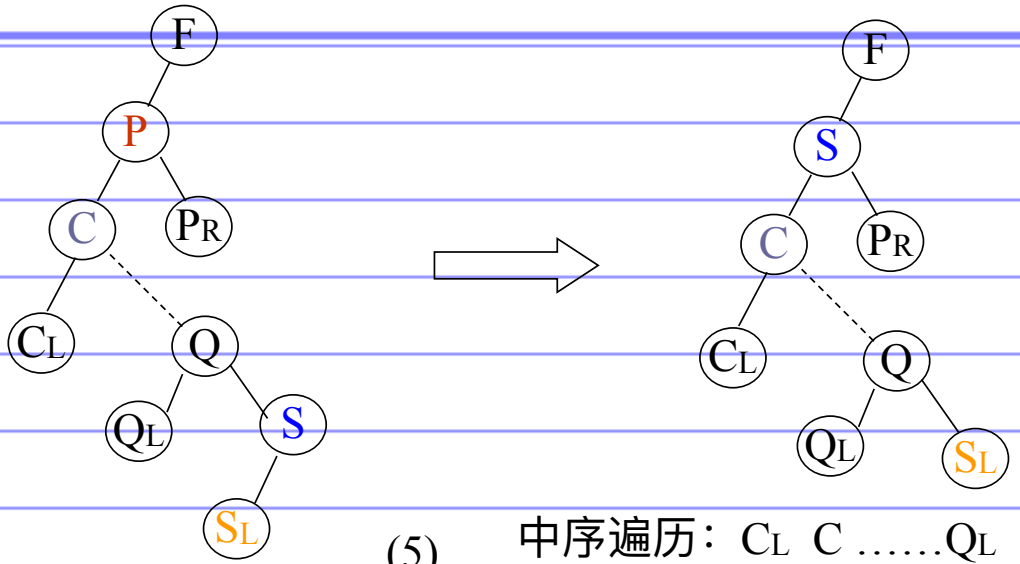
(3)




中序遍历：Q S P PR　　中序遍历：Q S PR

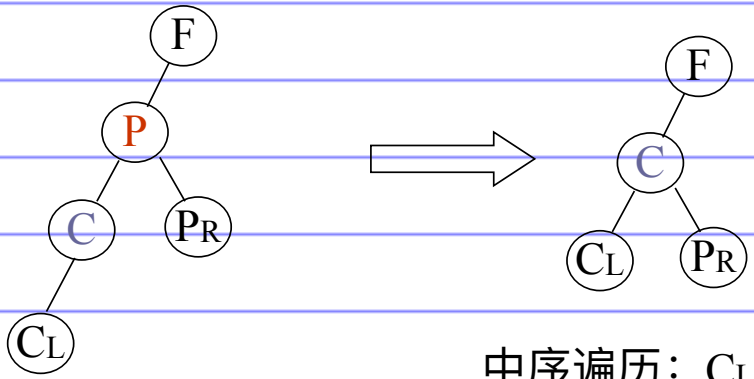
(4)

中序遍历：$C_L$ C ……$Q_L$ Q $S_L$ S P $P_R$ F




中序遍历：$C_L$ C ……$Q_L$ Q $S_L$ S $P_R$ F

(5)

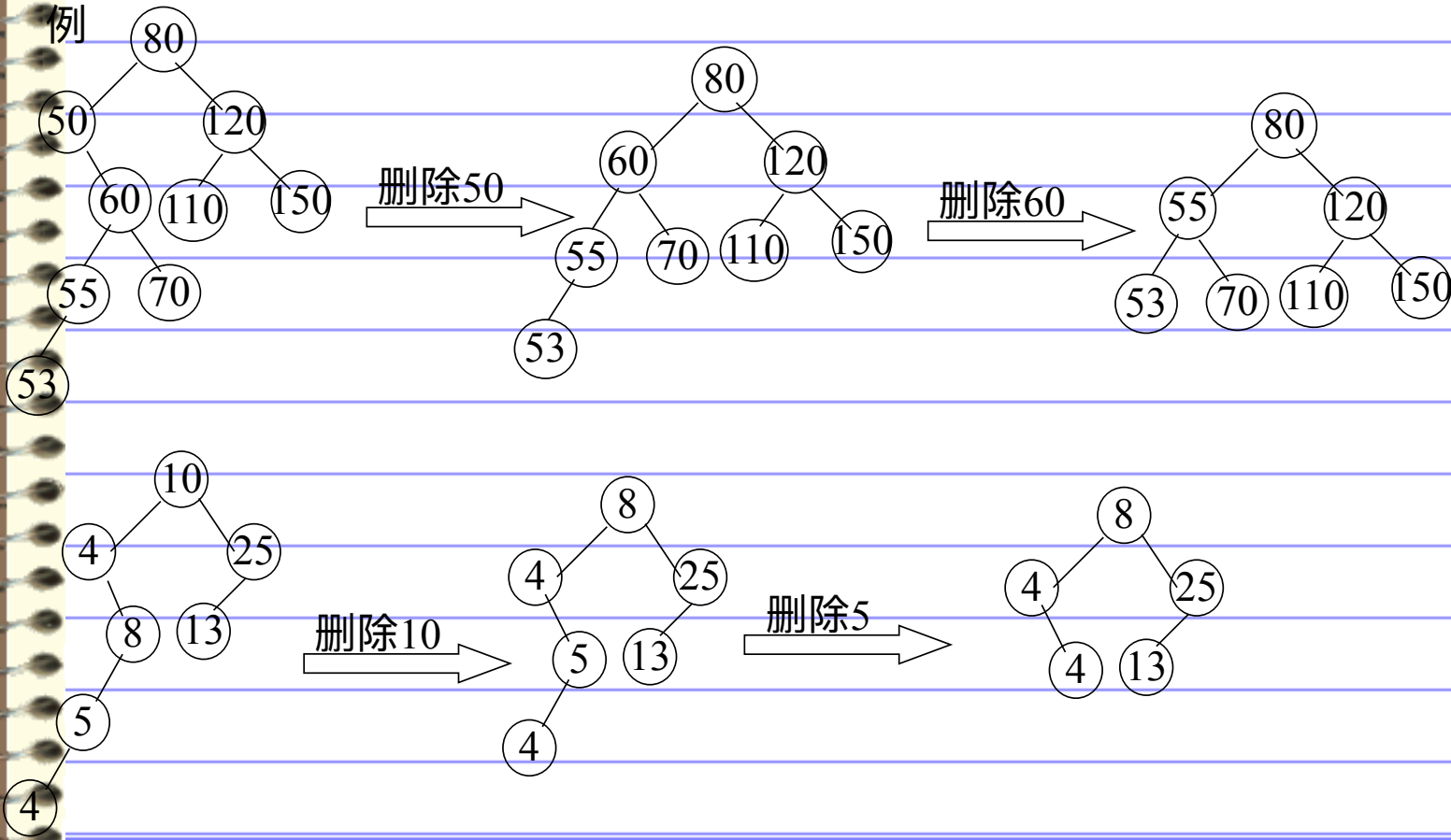



中序遍历：$C_L$ C $P_R$ F

中序遍历：$C_L$ C P $P_R$ F (6)

## 删除算法

例


80
50
120
60
110
150
55
70
53
删除50
80
60
120
55
70
110
150
53
删除60
80
55
120
53
70
110
150
10
4
25
8
13
5
4
删除10
8
4
25
5
13
4
删除5
8
4
25
4
13