第6章　树

6.1 树的基本概念

1. 什么是树（Tree）？

n（n≥0）个节点构成的一个层次结构

当n=0时，称为空树，记为Φ；当n＞0时，称为非空树。

非空树T满足下列条件：

（1）T有且只有一个根节点（root）；

（2）T的其余节点可分为互不相交的m(m≥0)个有限集

T_1、T_2、…、T_m，每个集合**T_i**($1\le i\le m$)是根的子树。

6.1 树的基本概念

树T的形式化描述：

T＝（D，R）

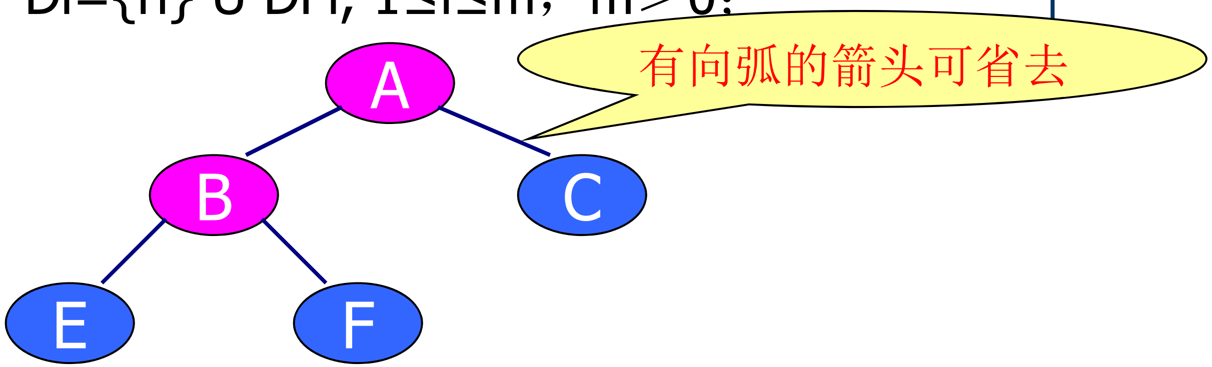
其中D，R分别为元素集和关系集，若D=φ，则树T为空树，否则有：

D＝{root}∪DF ，root∈datatype,为根元素,DF=根下各子树Ti的元素集：

$\bigcup_{i=1}^{m} D_i$ (m≥0)，且Di∩Dj =φ(不相交)，1≤i，j≤m，i≠j。

$$\mathbf{R=}\begin{cases} \mathbf{\langle root, ri\rangle,} & \text{ri 是 root的子树Ti之根,Ti=(Di, Ri)——递归；} \\ \text{Ri} & \text{Di=\{ri\} U DFi, 1≤i≤m，m＞0；} \\ \boldsymbol{\phi} & \text{m=0} \end{cases}$$

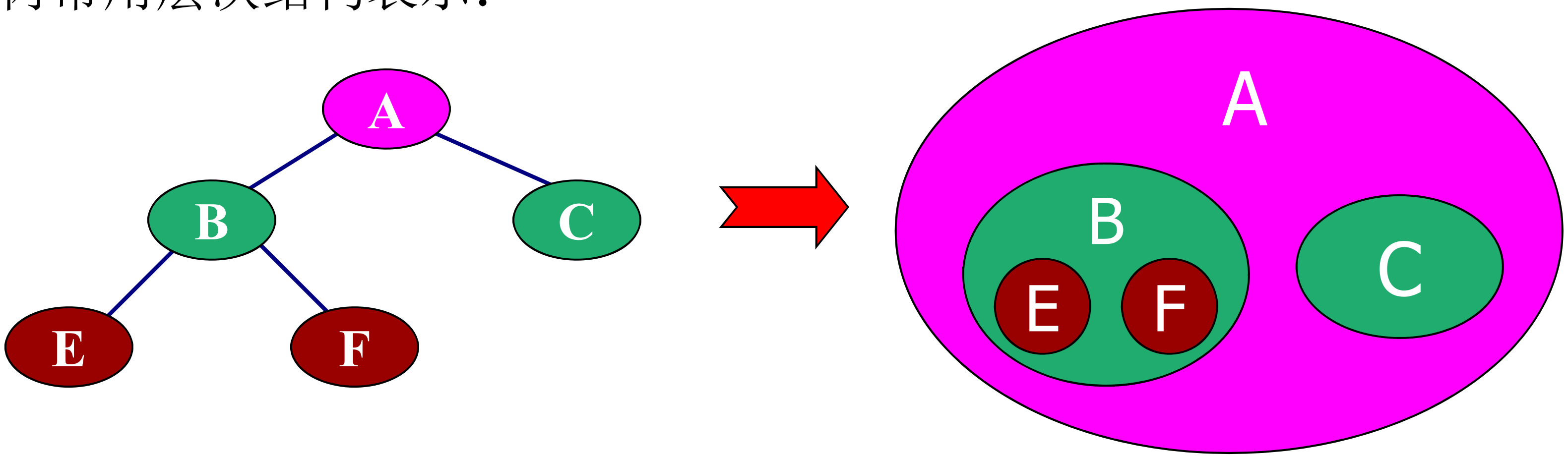
设树T：



即：D={A}∪DF，DF=D_1∪D_2，D_1={B}∪DF_1，DF_1=D_{11}∪D_{12}，D_{11}={E}，D_{12}={F}，D_2={C},故**D＝{A,B,C,E,F}**，而**R={<A,B>，<B,E>，<B,F>,<A,C>}**。

6.1 树 的 基 本 概 念

树常用层次结构表示：



还有嵌套表示法、广义表表示法。 **(A (B (E， F)， C))**

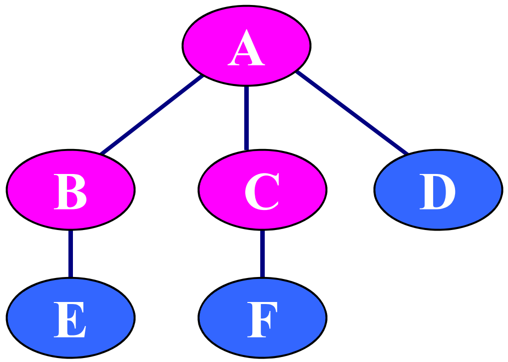
说明：

树可以用广义表表示，但广义表不一定能表示为树。

因为树中各子树不能相交，但广义表允许子表共享。

6.1 树的基本概念

2. 树的基本术语



(1) 父节点（双亲）与子节点（孩子）

如A是B、C、D的父节点， B、C、D是A的孩子

除根节点外，每个节点有且只有1个父节点

(2) 兄弟：具有同一父节点

如B、C、D是兄弟

(3) 根节点：1个，无父节点

叶节点：无子节点

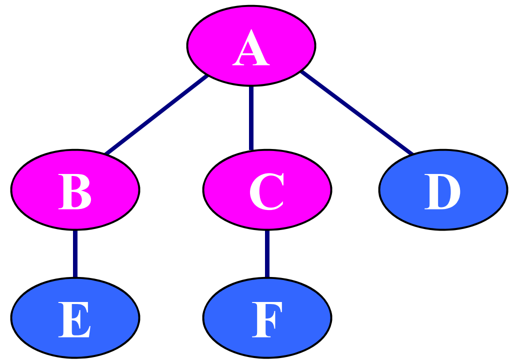
分支节点：其余

6.1 树的基本概念

(4) 节点的入度(ID)：指向节点的分支数目

节点的出度(OD)：子节点数

树的度(TD)：max(所有节点的出度)



如OD(A)=3，OD(B)=1。叶节点的出度为0。

根的入度为0，其它各节点的入度=1。

度=K的树称为K叉树。

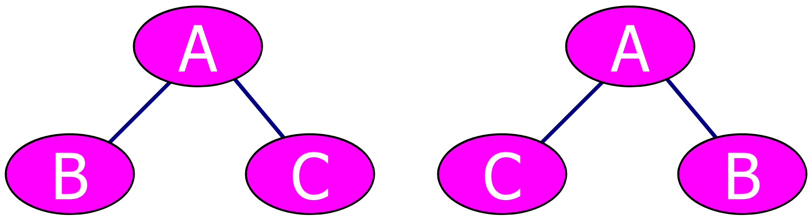
(5) 节点的层次：根为第1层。若某节点在第i层，则其孩子在第i+1层。

树的深度（或高度）：层次数的最大值为该树的深度。

6.1 树的基本概念

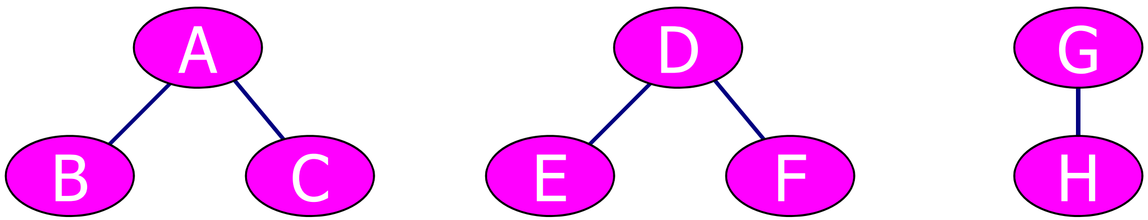
(6) 有序树： 树中任一节点的各子树从左到右有序

无序树： 否则



若T1，T2为有序树，它们是两棵不同的树；若T1，T2为无序树，它们是两棵相同的树。

(7) 森林（或树林）： m(m≥0) 棵互不相交的有序树的有序集合。



6.1 树的基本概念

3. 树的抽象数据类型

ADT Tree{

数据元素集：D（前面已介绍）；

数据关系集：R；

基本操作集：P；

TreeInit(&T)

操作结果：构造空树T。

TreeDestroy(&T)

初始条件：树T存在。操作结果：撤销树T。

TreeCreat(&T)

操作结果：依照建树规则构造一棵树T。

TreeClear(&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，返回TRUE，否则返回FLASE。

6.1 树的基本概念

TreeDepth(T)

初始条件：树T存在。

操作结果：返回T的深度。T＝Φ时，返回0。

Root(x)

初始条件：x是树或x是树中的节点。

操作结果：返回树x或x所在树的根节点。空树返回“空值”。

Parent(T,x)

初始条件：树T存在，x是T中的节点。

操作结果：返回树T中节点x的父节点。若x为根则返回“空值”。

Leftchild(T,x)

初始条件：树T存在，x是T中的节点。

操作结果：返回树 T 中节点x的最左孩子，若x为叶节点则返回“空值”。

Rightbro(T,x)

初始条件：树T存在，x是T中节点。

操作结果：返回树T中节点x的右兄弟，若x为双亲的最右子则返回“空值”。

6.1 树的基本概念

InsertChild(&T,p, i, q)

初始条件：p是树T中的节点，0≤i≤OD(p)，q是另一棵树的根节点。

操作结果：将以q节点为根的树，作为p的第i棵子树插入T中。

DeleteChild(&T,p,i)

初始条件：p是树T中的节点，0≤i≤OD(p)-1。

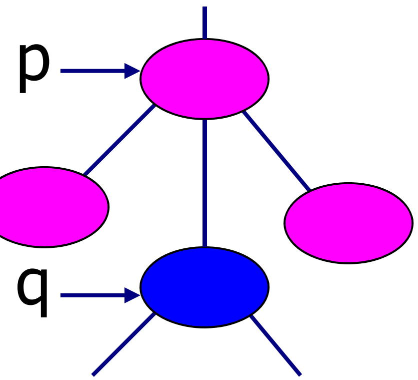
操作结果：删除树T中某p节点的第i棵子树。

TraverseTree(T)

初始条件：树T存在。

操作结果：依照某种次序（或规则）对树中的节点利用visit()函数进行访问，称为遍历（visit()是根据具体datatype和实际对数据的应用方式编写的访问函数）。

}ADT Tree；



6.1 树的基本概念

3. 树的性质

性质**1**：树中节点总数**n**（**n≥0**）等于树中各节点的出度之和加**1** 。即：

$$n = \sum_{i=1}^{n} OD(i) + 1$$

(OD(i)为第i节点的出度)

证：除根外，每节点的入度=1

所以树中总的分支数B＝n-1或**n＝B+1**。

又：一个节点发出的分支数＝该节点的出度，

所以各节点的分支数之和B＝ 树中各节点的出度之和。

代入n=B+1, 性质得证 。

6.1 树的基本概念

性质2：度＝K的树（K叉树）第i(i≥1)层至多有K^{i-1}个节点。

性质3：深度＝h(h≥1)的K（K＞1）叉树至多有$(K^h-1)/(K-1)$个节点。

如何证明？

性质2证明：采用数学归纳法。

i=1时，第1层最多1个节点，故$K^{1-1}=1$成立。

设K叉树第i-1层至多有$K^{(i-1)-1}=K^{i-2}$个节点，

因为是K叉树，所以第i-1层上每个节点最多有K个孩子节点，即

第i层至多有$K^{i-2}\bullet K=K^{i-1}$个节点，证毕。

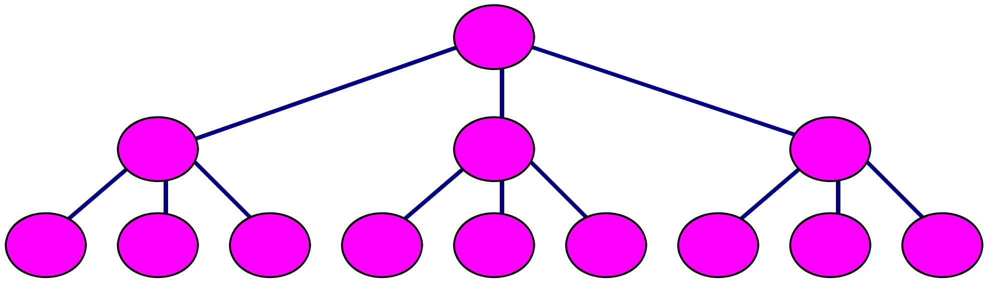
性质3证明：设深度=h的k叉树最大节点数为S，显然

$$S=\sum_{i=1}^{h}(\text{第i层最大节点数})=\sum_{i=1}^{h}K^{i-1}=\frac{K^h-1}{K-1}$$

6.1 树的基本概念

若一棵深度＝**h**的**K**叉树的节点数**=(K^h-1)/(K-1)**，则称之为**满K叉树**。

如：h=3，K=3的3层满3叉树：



节点数=（3^3-1）/（3-1）=13

6.1 树的基本概念

性质4：包含**n**（**n≥0**）个节点的**K**（**K＞1**）叉树的最小深度为：

$$\lceil \log_K (n(K-1)+1) \rceil$$

同样数量的节点，什么情况下树的深度最小？

证：设有n个节点的K叉树的深度为h，若该树第1~h-1层都是满的，即每层有最大节点数K^{i-1}（1≤i≤h-1），且其余节点都落在第h层，则该树的深度最小。

根据性质3有

$$\frac{K^{h-1}-1}{K-1} < n \le \frac{K^h-1}{K-1}$$

或：$(K^{h-1}-1) < n(K-1) \le (K^h-1)$　　即：$K^{h-1} < n(k-1)+1 \le K^h$

取对数：（h-1）$< \log_k(n(K-1)+1) \le h$

因为h是正整数，所以：h= $\lceil \log_K (n(K-1)+1) \rceil$

6.2 二叉树

1. 什么是二叉树（Binary Tree ）？

度=2的树。即每个节点最多2个孩子

二叉树

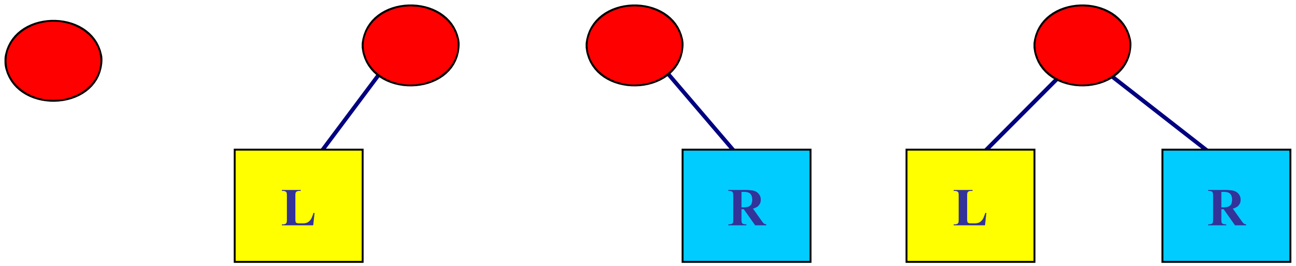
或者是空树；

或者由**1**个根节点以及左子树和右子树组成，左子树和右子树是二叉树。

(1) 5种基本形态

Φ

空树



L、R分别为二叉树根的左、右子树

6.2 二叉树

(2) 二叉树的抽象数据类型

ADT BinaryTree{　数据元素集：D；
数据关系集：R；
基本操作集：P；

BinaryTreeInit(&BT)
　　操作结果：构造空二叉树BT。
BinaryTreeDestroy(&BT)
　　初始条件：二叉树BT存在。操作结果：撤销二叉树BT。
BinaryTreeCreat (&BT)
　　操作结果：依照建树规则构造一棵二叉树BT。
BinaryTreeClear(&BT)
　　初始条件：二叉树BT存在。操作结果：二叉树BT清为空树。
BinaryTreeEmpty(BT)
　　初始条件：二叉树BT存在。操作结果：若BT为空二叉树，返回TRUE，否则返回FLASE。

6.2 二叉树

BinaryTreeDepth(BT)

初始条件：二叉树BT存在。操作结果：返回二叉树BT的深度。BT＝Φ时，返回0。

Root(x)

初始条件：x是二叉树或二叉树中的节点。操作结果：返回二叉树x或x所在二叉树的根节点。空二叉树返回“空值”。

Parent(BT,x)

初始条件：二叉树BT存在，x是BT中节点。操作结果：返回二叉树BT中节点x的父节点。若x为根则返回“空值”。

Child(BT,x,i)

初始条件：二叉树BT存在，x是BT中节点，i=0或1。

操作结果：若i=0，则返回二叉树BT中节点x的左子；若i=1，则返回二叉树BT中节点x的右子。无相应的孩子时返回“空值”。

6.2 二叉树

Brother(BT,x,i)

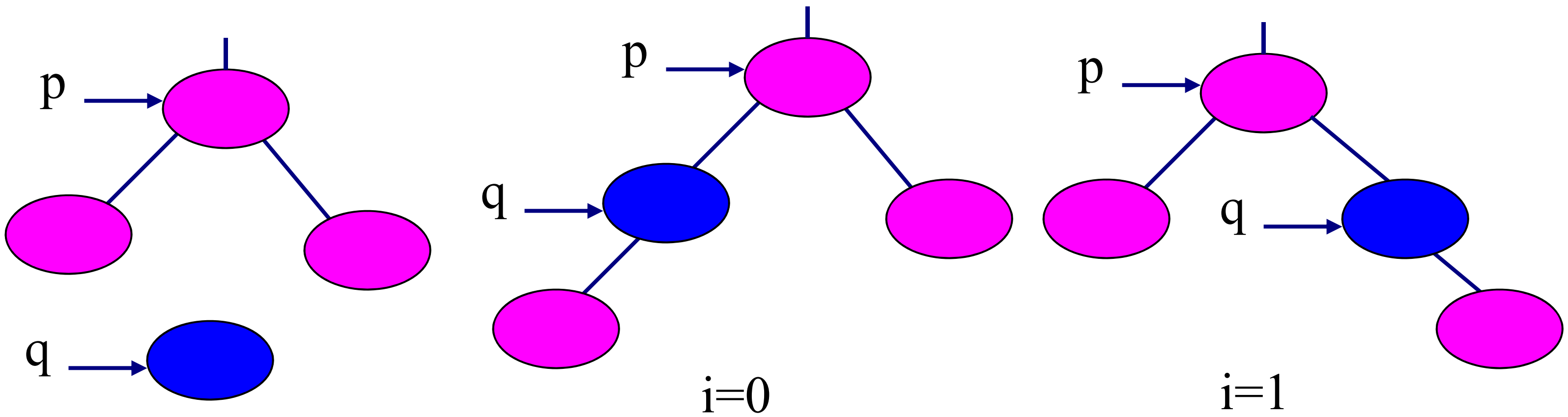
初始条件：二叉树BT存在，x是BT中节点，i=0或1。

操作结果：若i=0，则返回二叉树BT中节点x的左兄弟；若i=1，则返回二叉树BT中节点x的右兄弟。无相应的兄弟时返回“空值”。

InsertChild(&BT,p, i, q)

初始条件：p是二叉树BT中的节点，i=0或1，q是一个二叉树节点。

操作结果：i=0时，将q节点作为p的左子插入，i=1时作为p的右子插入，原p的左(右)子树改为q的左(右)子树。



6.2 二叉树

DeleteChild(&BT,p,i)

初始条件：p是二叉树BT中的节点，i=0或1。

操作结果：若i=0，则删除BT中P节点的左子树；若i=1，则删除P节点的右子树。

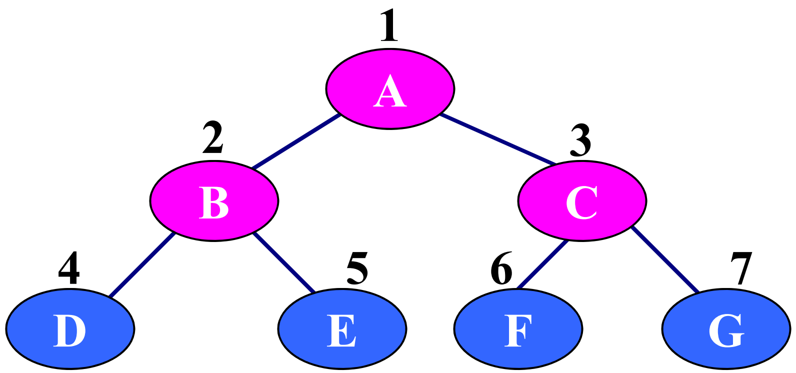
TraverseBinaryTree(BT)

初始条件：二叉树BT存在。操作结果：依照某种次序（或规则）对二叉树树中的节点利用visit()函数进行访问，称为遍历。

}ADT BinaryTree；

6.2 二叉树

(3) 满二叉树与完全二叉树



满二叉树（ **Full Binary Tree** ）：

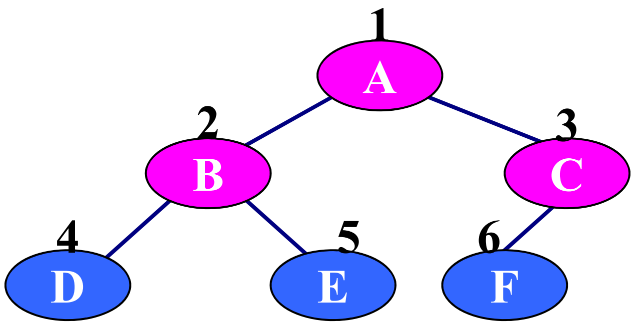
节点数=2^h-1，h是深度，即

每一层都是满的

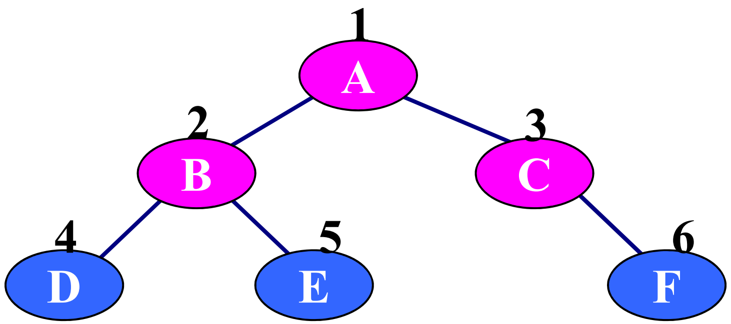
完全二叉树（**Complete Binary Tree** ）：

除最底层外，每一层都是满的，底层节点集中在左边

若将h层的满二叉树从上到下、从左到右编号，则具有n个节点的完全二叉树包含的节点是1~n，与满二叉树的1~n号节点对应。



完全二叉树



非完全二叉树

6.2 二 叉 树

2. 二叉树的性质

- **性质1**：二叉树第$i(i \geq 1)$层上至多有2^{i-1}个节点。
- **性质2**：深度为$h(h \geq 1)$的二叉树至多有2^h-1个节点。
- **性质3**：设二叉树BT中叶节点数为n_0，出度为2的节点为n_2，则有：

$$n_0=n_2+1$$

证：设BT中总节点数为n，出度＝1的节点数为n_1，有：

$$n=n_0+n_1+n_2 \qquad (1)$$

又：根据树的性质1，树中分支数B与n的关系为n=B+1。另外，n_2个出度＝2的节点共发出分支数为$2n_2$；n_1个出度＝1的节点发出分支数为$1*n_1$；而n_0个叶节点不发出分支。故$B=2n_2+n_1$，有：

$$n=2n_2+n_1+1 \qquad (2)$$

(2)式–(1)式, 得： $0=n_2-n_0+1$或$n_0= n_2+1$，证毕。

- **推论:**设K叉树中叶点数为n_0，出度=2,3, …, k的节点数分别为$n_2,n_3,\ldots,n_K$，则: $n_0=n_2+2n_3+\ldots+(k-1)n_K+1$（其证明方法可参照二叉树的性质3）

6.2 二叉树

性质4：含有**n(n ≥1)**个节点的完全二叉树的深度 $h = \lfloor \log_2 n \rfloor + 1$

证：根据二叉树性质2，有：

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

即 $$2^{h-1} \leq n < 2^h$$

取对数： $$(h-1) \leq \log_2 n < h$$

因为 h为正整数，所以有：

$$\lfloor \log_2 n \rfloor = h - 1$$

即

$$h = \lfloor \log_2 n \rfloor + 1$$

6.2 二叉树

性质**5**：设完全二叉树**BT**节点数为**n**，节点按层编号。对**BT**中第**i**节点**(1≤i≤n)**，有：

（**1**）若**i=1**，则**i**节点**(**编号为**i**的节点**)**是**BT**之根，无父节点；否则**(i＞1)**，**parent(i)=**$\lfloor i/2 \rfloor$，即**i**节点父节点的编号为$\lfloor i/2 \rfloor$；

（**2**）若**2i＞n**，则**i**节点无左子，否则**Lchild(i)=2i**，即**i**节点的左子位于第**2i**号节点；

（**3**）若**2i+1＞n**，则**i**节点无右子，否则**Rchild(i)=2i+1**，即**i**节点的右子位于第**2i+1**号节点。

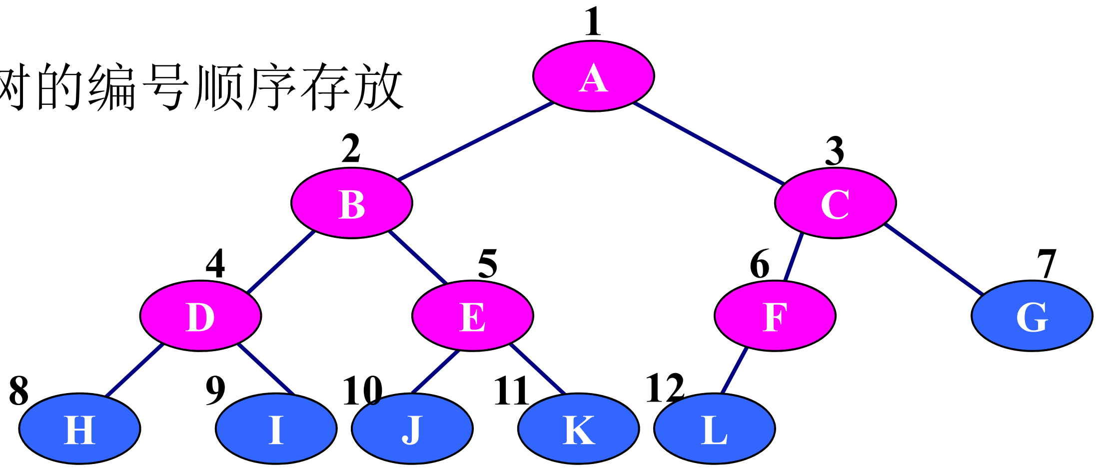
证：采用数学归纳法，先证（2）和（3）。

6.2 二叉树

3. 二叉树的存储结构

(1) 顺序结构

用一维数组，按完全二叉树的编号顺序存放



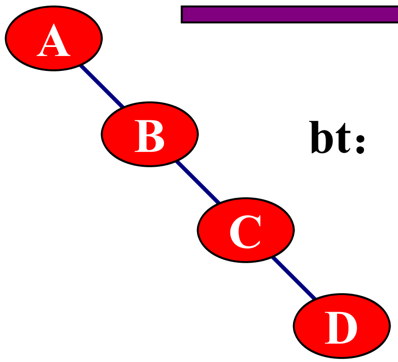
bt:	A	B	C	D	E	F	G	H	I	J	K	L	#	…
下标:	1	2	3	4	5	6	7	8	9	10	11	12	13	…

完全二叉树顺序存储于数组bt中（bt[0]未用到）

优点：适合完全二叉树。根据下标很容易计算任一节点的父子节点。
如节点5的父节点是2，左子是10，右子是11

二叉树的存储结构

缺点：非完全二叉树会浪费空间。如单斜树



bt:

A	@	B	@	@	@	C	@	@	@	@	@	@	@	D	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

（@为‘虚结点’或‘空结点’）

二叉树的顺序存储结构

按二叉树层次顺序读入树中元素（虚节点为‘@’,结束符为‘#’），建立顺序存储结构的算法：：

```
#define  maxsize 1024  //二叉树节点数最大值//
typedef  datatype  sqtree [maxsize];
void CreateBtree(sqtree bt)
{    int i=1; datatype ch=getchar();  // 读入数据 //
     while ( ch!= '#' )
           {bt [i++ ]=ch;  ch=getchar ();}
     bt [i] ='#';
}
```

二叉树的存储结构

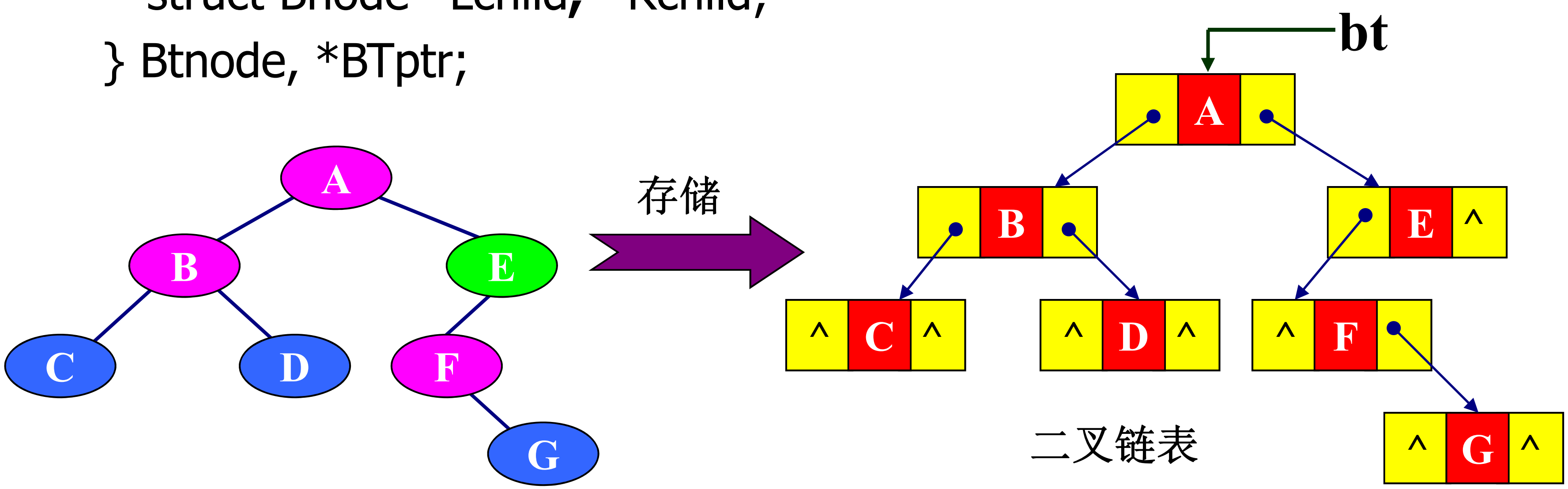
(2) 链式结构

节点的形式：

Lchild	data	Rchild
左子指针	数据	右子指针

节点类型：

```
typedef struct Bnode {
    datatype data;
    struct Bnode *Lchild, *Rchild;
} Btnode, *BTptr;
```



二叉树的存储结构

(2) 链式结构 – 建立二叉链表

算法思路：

按层次顺序依次输入节点(设为字符)，若≠虚节点（@），则建立新节点，并将其链入到它的双亲之下。为了使节点能正确链入，算法用到队列技术，保存输入节点的地址(虚节点地址=NULL)。队头(为指针)指向相应的双亲，队尾元素指向相应的孩子。若输入的节点序号为偶数，孩子作为双亲的左子插入，否则作为右子插入，而对虚节点，无须链接。

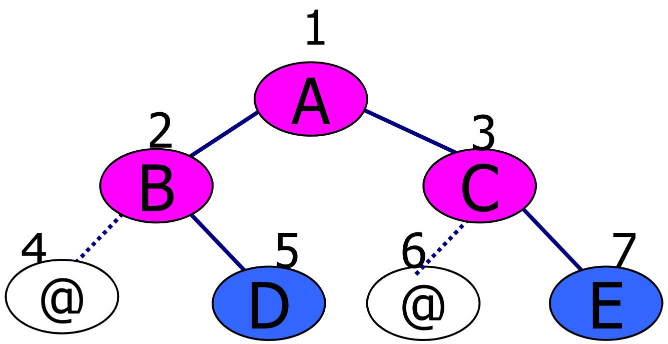
建立二叉链表

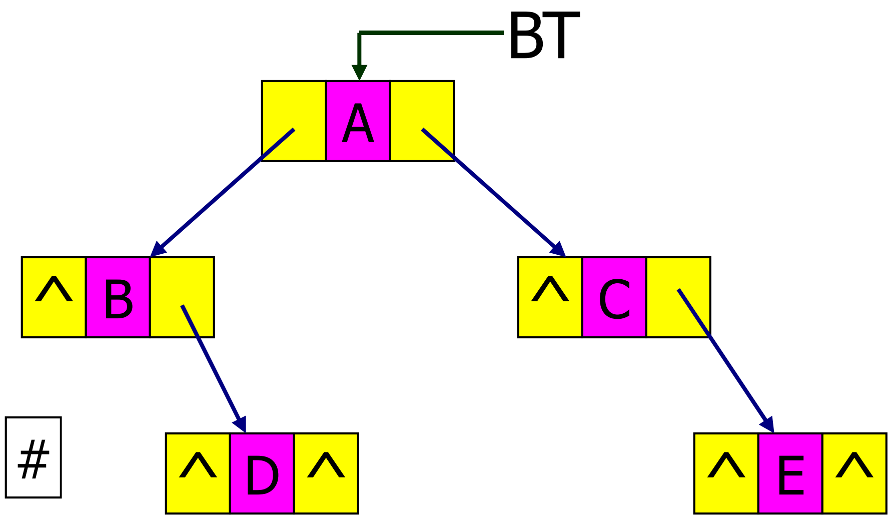
算法描述：

```
BTptr CreateLBtree (BTptr BT)  //建立以BT为根节点指针的二叉链表
{  datatype ch; int i=0; BTptr p,q; queuetype Q;
   Clearqueue(Q); BT=NULL; ch=getchar ( ); //置队Q、树为空，读入数据
   while (ch!=‘#’)
   {  p=NULL; //P为新节点地址，但空节点地址为NULL
      if (ch！ =‘@’ )  { p=(BTptr) malloc (sizeof (BTnode); //申请新节点
                           p->data=ch; p->Lchild=p->Rchild=NULL;}
      i++; Enqueue(Q, p ); //节点序号计数,新节点地址或虚地址(NULL)进队
      if (i==1) BT=p;          // 第一输入节点为根
      else {q=Getqtop(Q) ；   // 取队头元素q,为p之父节点指针
             if(q &&p)  if (i%2= =0 )  q->Lchild=p;  //i=偶数,P是双亲之左子
             else q->Rchild=p;              //i=奇数,P是双亲之右子
             if(i%2==1) Delqueue(Q,q); }  //当前双亲处理完出队
      ch=getchar(); }      //输入下一数据
   return(BT);}
```

建立二叉链表

设二叉树**BT**如下图：



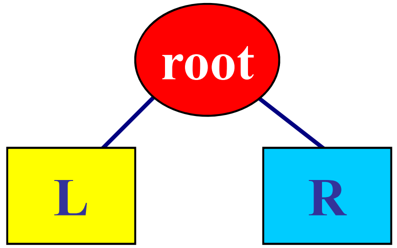


读入：	A	B	C	@	D	@	E	#
序号：	1	2	3	4	5	6	7	8
队列：				^	D↑	^	E↑	

6.3 二叉树的遍历

遍历（traversal）：按某种次序访问每个节点各1次

1. 3种常用的遍历方法



(1) 前序（或先序、先根）遍历 (简称DLR遍历)

1) 访问根节点——**D**

2) 前序遍历左子树——**L**

3) 前序遍历右子树——**R**

(2) 中序（或中根）遍历(简称LDR遍历)

1) 中序遍历左子树——L

2) 访问根节点——D

3) 中序遍历右子树——R

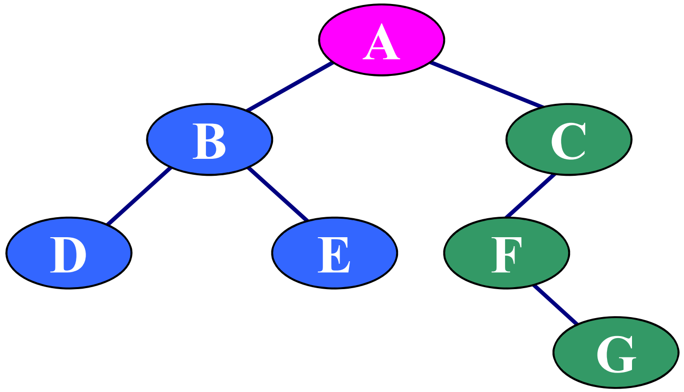
(3) 后序（或后根）遍历 (简称LRD遍历)

1) 后序遍历左子树——L

2) 后序遍历右子树——R

3) 访问根节点——D

6.3 二叉树的遍历



DLR: (A B D E C F G)

LDR: (D B E A F G C)

LRD: (D E B G F C A)

二叉树的遍历可理解为对层次模型的数据结构按一定规则线性化，得到一个类似线性表的序列。

6.3 二叉树的遍历

2. 遍历的递归算法

```
void preorder( BTptr T)  //对当前根节点指针为T的二叉树按前序遍历
{
    if (T == NULL)  return;
    visit(T);                        //访问T所指节点
    preorder(T->Lchild);  //前序遍历T之左子树
    preorder(T->Rchild);  //前序遍历T之右子树
}
```

中序和后序遍历也类似，只是visit()的位置不同而已。

6.3 二叉树的遍历

```
void  Inorder(BTptr T) //对当前根节点指针为T的二叉树按中序遍历
{
    if (T == NULL)   return;
    Inorder( T->Lchild);    //中序遍历T之左子树
    visit(T);               //访问T所指节点
    Inorder(T->Rchild);     //中序遍历T之右子树
}
void postorder( BTptr T) //对当前根节点指针为T的二叉树按后序遍历
{
     if (T == NULL)   return;
     postorder(T–>Lchild);  //后序遍历T之左子树
     postorder(T–>Rchild); //后序遍历T之右子树
     visit(T);              //访问T所指节点
}
```

6.3 二叉树的遍历

3. 遍历的非递归算法

设bt是指向根节点的指针

(1) 前序遍历

算法思路：

从根节点开始，一直向左走。每遇到1个节点，访问之，并将其右子指针进栈。

直到走到头为止。

然后，再出栈，访问右子树。

遍历的非递归算法

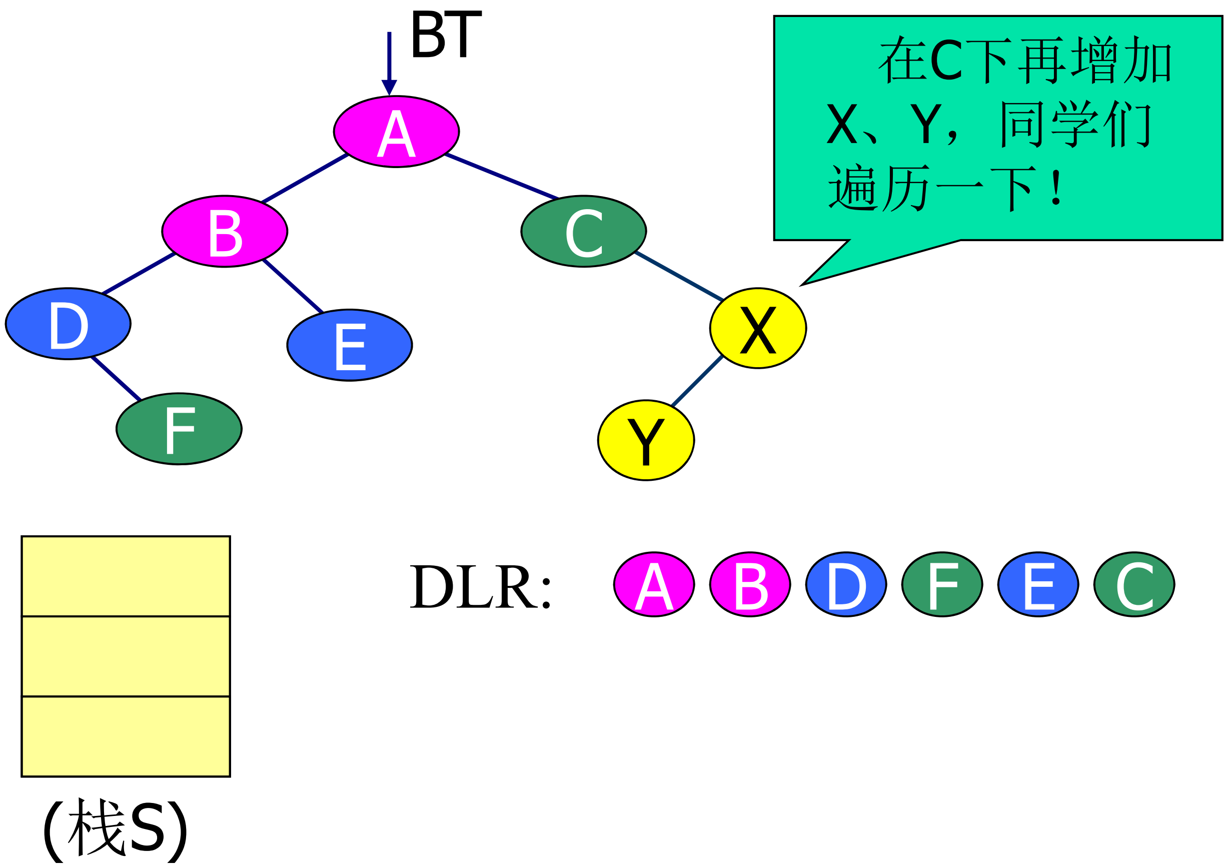
前序遍历的非递归算法：

```
void  Preoder-1(BTptr T)  //前序非递归遍历二叉树T
{  BTptr p；  stacktype s;
   Clearstack(s); push(s,T); //置栈S为空、根指针 T 进栈
   while (!Emptystack(s) )
    { p=pop(s);    //出栈,栈顶=>P
       while (p)
        { visit (p);   //访问p节点
          if (p–>Rchild)  push(s,p–>Rchild); //右子存在时,进栈
          p=p–>Lchild; } //向左走
      }
}
```

遍历的非递归算法

设二叉树BT如下图：

按中序非递归遍历时，栈S的变化状态



遍历的非递归算法

(2) 中序遍历

当首次遇到根节点**A**时如何处理?

只能记录下A的地址，先向左走!

待处理完其左子树后，再返回访问A及其右子树。

算法思路:

中序遍历与前序遍历类似: 从根节点出发一直向左走

不同的是: 每遇到1个节点，不能马上访问（因为需要先访问左子树），必须将指向该节点的指针进栈，然后向左走。向左走到头后，再出栈，访问当前节点、访问右子树。

遍历的非递归算法

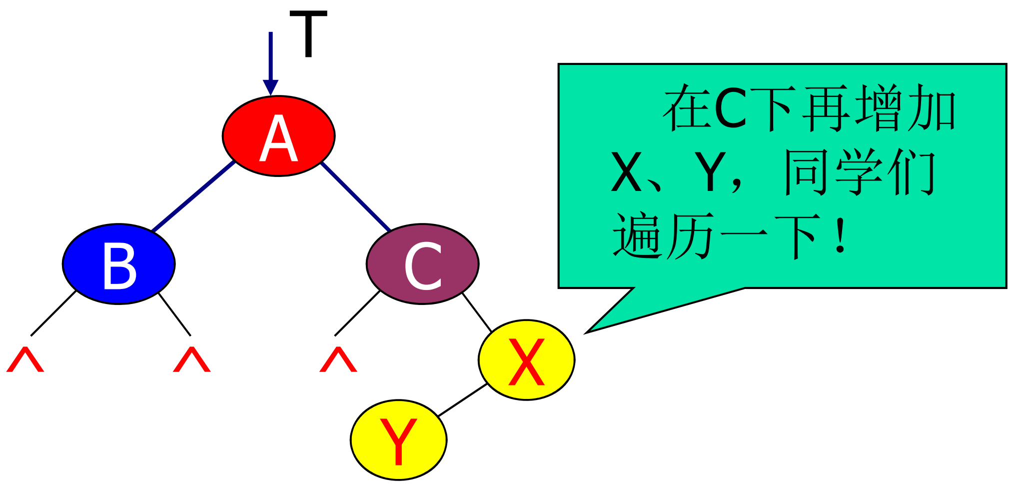
中序遍历的非递归算法：

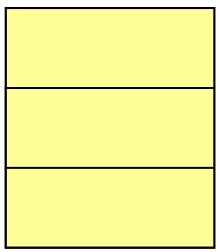
```
void Inorder-1 (BTptr T)                    // 中序非递归遍历二叉树T
{  BTptr  p; stacktype s;
   Clearstack(s);  push (s,T);              //置栈S空、根指针进栈
   while (!Emptystack(s))
   {  while ((p=Getstop (s))&& p)           // 取栈顶且栈顶不为空时
          push(s,p->lchild);                //p之左子指针进栈
      p=pop(s);                             //去掉最后的空指针
      if (!Emptystack (s))
      {  p=pop(s);                          //取当前访问节点的指针=>P
         visit(p);                          //访问P节点
         push(s,p-> Rchild); }              //遍历P之右子树
   }
}
```

遍历的非递归算法

设二叉树BT如下图：

按中序非递归遍历时，栈S的变化状态





(栈S)

LDR: B A C

遍历的非递归算法

(3) 后序遍历

算法思路：类似于中序，但有其特殊性。

当搜索指针指向某节点时，不能马上访问，而要先遍历其左子树，故需将指向该节点的指针进栈，这一点同中序。

但当遍历完左子树时，再次返回到该节点时，还不能访问，要先遍历其右子树，故需要出栈后再进栈。

为了区分同一节点地址的两次进栈，

设一标志tag，tag和节点指针一起进栈。

$$\text{tag} = \begin{cases} 0 & \text{表示该节点地址首次进栈，暂不能访问} \\ 1 & \text{表示该节点地址第2次进栈，可以访问} \end{cases}$$

定义栈元素类型：

```
typedef struct {
    BTptr q; //节点地址
    int tag;  //标志
} STYPE;
```

遍历的非递归算法

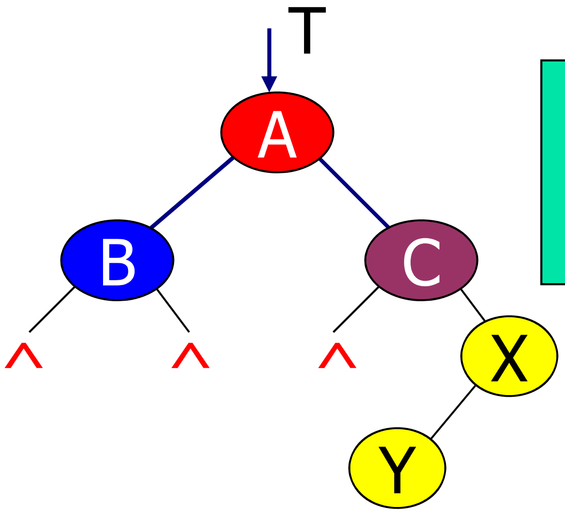
后序遍历的非递归算法：

```
STYPE sdata;   ClearStack(S);    p = bt;
do { while (p)
        { sdata.q = p; sdata.tag = 0;
          Push(S, sdata);                          // (p,0)进栈
          p = p->Lchild;    }                      //遍历p之左子树
      sdata = Pop(S);p = sdata.q; tag = sdata.tag;         //退栈，取指针、状态位
      if (tag == 0)                                //应先遍历右子树
        { sdata.tag = 1;   Push(S, sdata);         // (p,1) 进栈
          p = p->Rchild; }                         //遍历右子树
      else {   visit(p);                           //访问p节点
               p = NULL; }
 }while (!Emptystack(S));
```

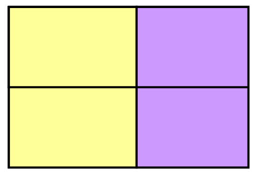
遍历的非递归算法

设二叉树BT如下图：

按后序非递归遍历时，栈S的变化状态



在C下再增加X、Y，同学们遍历一下！



(栈S)

LRD: B C A

6.3 二叉树的遍历

4. 按层次遍历二叉树

先遍历二叉树的第1层（根），然后遍历第2层，……，每层节点从左至右依次访问。

采用什么数据结构？

算法思路：

采用队列，即访问当前节点后，将该节点的左子和右子指针进队（若有）。

为了统一，首先将根节点指针进队。

处理方法：出队，访问之，将该节点左子和右子指针进队（若有）。直到队空为止。

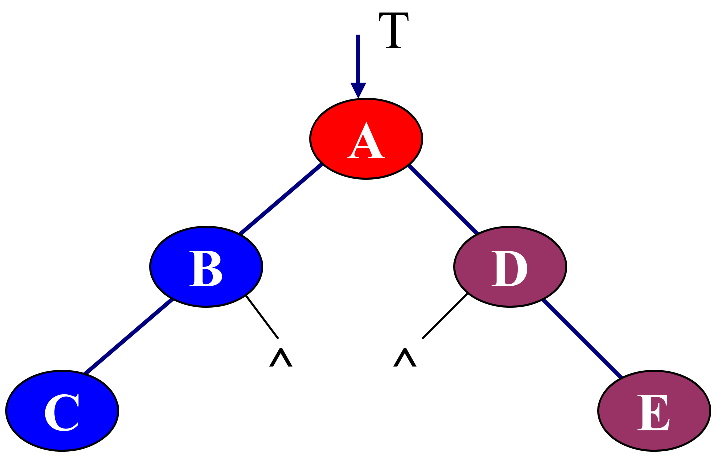
按层次遍历二叉树

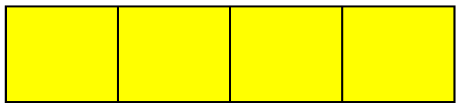
算法描述：

```
void LayerOrder(BTptr T)  //对二叉树T按层次遍历
{
    if (T == NULL) return;
    BTptr p;
    ClearQueue(Q);
    Enqueue (Q, T);   //将根节点指针进队
    while (!EmptyQueue(Q) ) {
        p = Dequeue(Q);   //出队，队头元素⇒p
        visit (p);              //访问p节点
        if (p->Lchild) Enqneue (Q, p->Lchid);   //左子指针进队
        if (p->Rchild) Enqneue (Q, p->Rchid);  //右子指针进队
    }
}
```

按层次遍历二叉树

例 设二叉树BT如下图：



队列：

遍历序列：

按层次遍历的过程：

1）根节点指针A↑进队；

2）出队访问A，因A的左、右子（B，D）存在，故B↑、D↑进队；

3）出队访问队头B，B的左子指针C↑进队；

4）出队访问D，D的右子指针E↑进队。

5）因C、E为叶节点，出队访问C、E后队为空，结束。

6.3 二叉树的遍历

5. 遍历算法的应用

- ✓ 凡是对二叉树中各节点均处理1次的问题，都可以直接利用前序/中序/后序遍历的递归/非递归算法，修改visit()。通常总体结构不变。但要注意变量的初始化。
- ✓ 根据实际问题的需要，可能需要加入其他处理语句。
- ✓ 根据需要选择前序、中序或后序遍历。

（1）**统计二叉树中出度=0、1、2的节点个数**

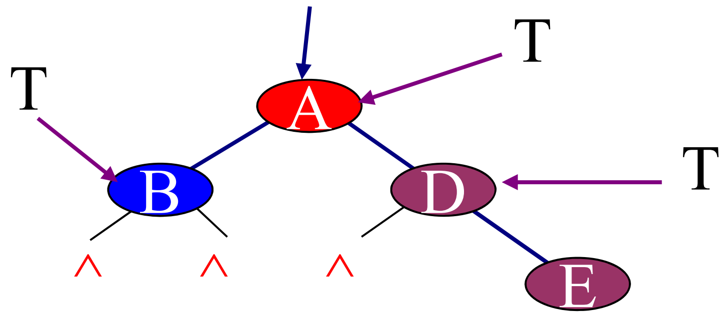
算法思路：

利用前序/中序/后序遍历的递归/非递归算法，修改visit()。

相对来说，用前序更容易理解。

遍历算法的应用

```
//统计二叉树T中出度=0、1、2的节点个数
//in:   *n0 = *n1 = *n2 = 0
//out: *n0, *n1, *n2分别为出度=0、1、2的节点个数
void PreorderCount( BTptr T, int *n0, int *n1, int *n2)
{
    if (T == NULL)  return;
    if (T->Lchild == NULL && T->Rchild == NULL ) { //叶节点
        *n0 += 1;
    } else if (T->Lchild != NULL && T->Rchild != NULL ) {
        *n2 += 1;
    } else {
        *n1 += 1;
    }
    PreorderCount(T–>Lchild, n0, n1, n2);
    PreorderCount(T–>Rchild, n0, n1, n2);
}
```



遍历算法的应用

```
//统计二叉树T中出度=0、1、2的节点个数
//in: *n0 = *n1 = *n2 = 0
//out: *n0, *n1, *n2分别为出度=0、1、2的节点个数
void PostorderCount( BTptr T, int *n0, int *n1, int *n2)
{
    if (T == NULL)  return;
    PostorderCount(T–>Lchild, n0, n1, n2);
    PostorderCount(T–>Rchild, n0, n1, n2);
    if (T->Lchild == NULL && T->Rchild == NULL ) { //叶节点
        *n0 += 1;
    } else if (T->Lchild != NULL && T->Rchild != NULL ) {
        *n2 += 1;
    } else {
        *n1 += 1;
    }
}
```

遍历算法的应用

(2) 交换二叉树中各节点左、右子树

算法思路：

利用前序/中序/后序遍历的递归/非递归算法，修改visit()。

```
void PreExchange( BTptr T)  //交换二叉树BT中各节点左、右子树
{
    if (T == NULL)  return;
    BTptr p;
    p = T->Lchild;
    T->Lchild =  T->Rchild;
    T->Rchild = p;
    PreExchange(T–>Lchild);
    PreExchange(T–>Rchild);
}
```

如果用中序、后序呢？

遍历算法的应用

(3) 求二叉树的深度

1）用递归

空树的深度=0；

非空二叉树的深度=max(左子树的深度，右子树的深度) + 1。

算法描述：

```
//返回二叉树BT的深度
int DepthofBT( BTptr BT)
{
    if (BT == NULL)  return 0;
    return max(DepthofBT(BT->Lchild),
        DepthofBT(BT->Rchild)) + 1;
}
```

```
//返回二叉树BT的深度，利用后序遍历
int DepthofBT1( BTptr T)
{
    if (T == NULL)   return 0;
    int ldep = DepthofBT1(T–>Lchild);
    int rdep = DepthofBT1(T–>Rchild);
    return max(ldep, rdep) + 1;
}
```

适合采用后序递归遍历。

遍历算法的应用

2）在非递归遍历算法的基础上修改visit()

由于求深度必然要遍历所有节点，故可在遍历算法的基础上修改visit()。但要引入其他语句，整体结构不变。

算法思路：

- ✓ 二叉树的深度是处在最底层的叶节点的层数；
- ✓ 节点所在的层数 = 父节点层数+1
- ✓ 访问节点时，当前层数为该节点父节点层数+1
- ✓ 节点地址进栈时，其层数也需要随之进栈

```
typedef struct { //栈元素类型
    BTptr pnode;
    int depth; //pnode所指节点的层数
} STDATA;
```

遍历算法的应用

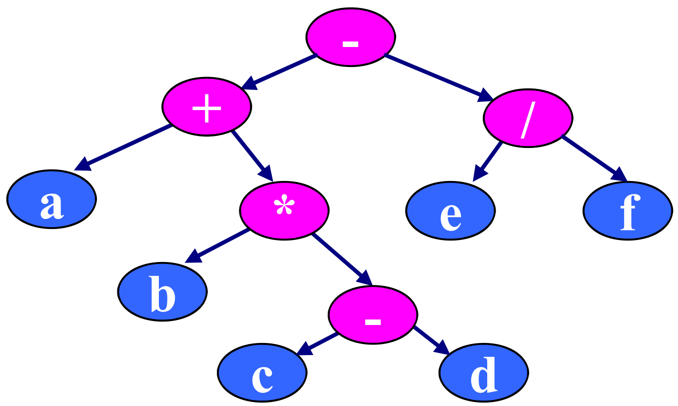
采用前序遍历非递归算法求二叉树深度：

```
int PreorderDep(BTptr BT)
{
    int curdep = 0;
    int maxdep = 0;
    STDATA sdata;
    ClearStack(S);
    BTptr p = BT;
    while (1) {
        if (p != NULL) {
            curdep++; //p所指节点层数
        } else {
            if (EmptyStack(S)) break;
            sdata = Pop(S);
            p = sdata.pnode;
            curdep = sdata.depth;
        }
        if (p->Lchild == NULL
            && p->Rchild == NULL){
        //当前节点是叶节点时，修改maxdep
            if (curdep > maxdep)
                maxdep = curdep;
        }
        if (p->Rchild != NULL) {
            sdata.pnode = p->Rchild;
            sdata.depth = curdep + 1;
            Push(S, sdata);
        }
        p = p->Lchild;
    }
    return maxdep;
}
```

遍历算法的应用

(4) 表达式求值

设表达式以二叉树表示，例如a+b*(c-d)-e/f的二叉树结构：



对该树进行前、中、后序遍历，可得到表达式的前缀、中缀和后缀表达式：

DLR:-+a*b-cd/ef

LDR:a+b*c-d-e/f

LRD:abcd-*+ef/-

适合采用后序遍历。

节点定义：

Lchild	tag	data/optr	Rchild

若tag=0，则data/optr项为data，存放操作数；

若tag=1，则data/optr项放optr，存放操作符。

遍历算法的应用

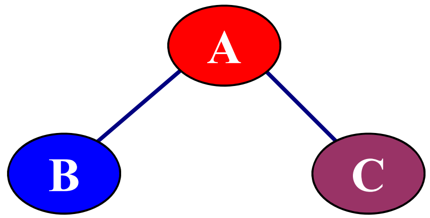
利用后序遍历递归算法对表达式求值的算法描述：

节点描述：

```
typedef struct Bnode {
    int tag;     //标志位
    union {
        float data;  char optr;
    } dtype;
    struct Bnode *Lchild,*Rchild ;
}BTnode , *BTptr;
int postorder-E(BTptr BT, float *value)      //value返回结果
{
    if (T == NULL) return -1; //表达式为空
    if (T->tag==0) { //节点为操作数，直接返回其值
        *value = T->dtype.data; return 0;
    }
    float loprd, roprd;
    postorder-E(T-> Lchild, &loprd);  //求左子树对应子表达式的值
    postorder-E(T-> Rchild, &roprd); //求右子树对应子表达式之值
    *value = opetate(loprd,T->dtype.optr, roprd); //做左右两个操作数的运算
    return 0;
}
```

6.4 线索二叉树

1. 线索二叉树的引入



对二叉树的遍历，可以得到一个线性序列，如：

按中序遍历得到：**B**，**A**，**C**，则**A**的前驱为**B**，**A**的后继为**C**。

要查找二叉树节点在某序下的前驱和后继，可采取什么方法？

- 对二叉树按某序遍历，则任一节点的前驱和后继即可得到，但这样做费时，需要遍历整棵树；
- 给每个节点增加两个指针，分别指向前驱和后继，但增加了系统开销。
- 线索二叉树。A.J.Perlis（帕利斯）和C.Thornton（桑顿）二人注意到，**n**个节点的二叉链表中，有**n+1**个指针是空的，于是提出：利用这些空的指针域来指向某序下的前驱和后继——即线索二叉树。

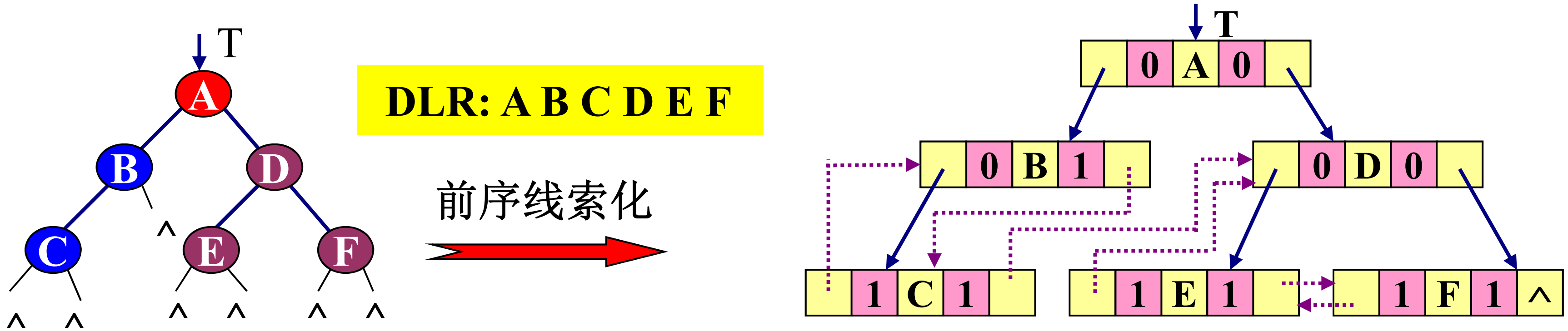
6.4 线索二叉树

线索二叉树节点定义：

Lchild	Ltag	data	Rtag	Rchild

Ltag = { 0　Lchild≠∧（指向本节点左子）
　　　　 1　Lchild=∧（利用起来，指向本节点前驱）

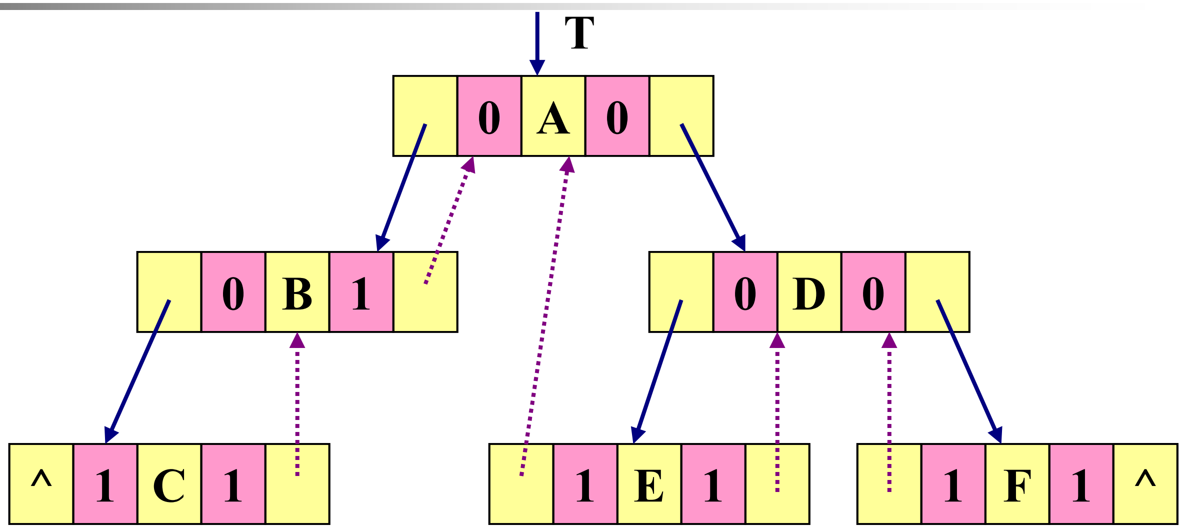
Rtag= { 0　Rchild≠∧（指向本节点右子）
　　　　1　Rchild=∧ (指向本节点后继)



6.4 线索二叉树

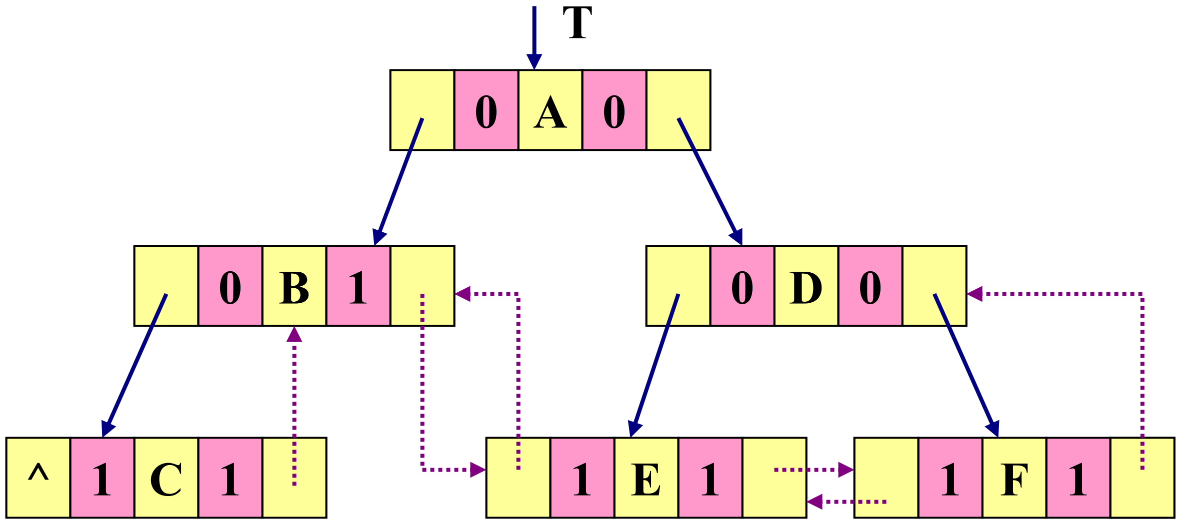
中序线索化

LDR: C B A E D F



后序线索化

LRD: C B E F D A

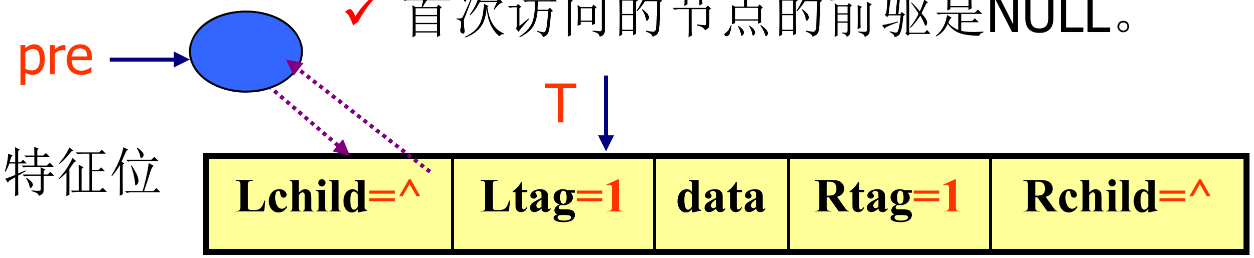


6.4 线索二叉树

2. 建立线索二叉树

以建立中序线索二叉树为例。

- ✓ 执行visit(BT)时，上次访问的节点是其前驱，下次访问的节点是其后继。
- ✓ 首次访问的节点的前驱是NULL。




```
//节点类型描述
typedef struct Bnode {
    int Ltag, Rtag;  //左右特征位
    datatype data;
    struct Bnode *Lchild, *Rchild ;
}BTnode , *BTptr;
BTptr  pre=NULL;
void  Inorder(BTptr BT) //中序遍历的递归算法
{
    if (BT == NULL)   return;
    Inorder(BT->Lchild);
    visit(BT);
    Inorder(BT->Rchild);
}
```

```
if(T->Lchild==NULL)
{T->Ltag=1;T->Lchild=pre;}
if(T->Rchild==NULL)
   T->Rtag=1;
if(pre&&pre->Rtag==1)
    pre->Rchild=T;
 pre=T;
```

6.4 线索二叉树

建立中序线索二叉树的**算法描述**：

将所有节点的Ltag和Rtag初始化为0，无孩子节点的指针域均为NULL;

```
BTptr pre = NULL; //全局变量
void  Inthreadbt (BTptr BT)   //二叉树BT的中序线索化
{
    if (BT ==  NULL) return;
    Inthreadbt(BT->Lchild); //线索化左子树
    if (BT->Lchild == NULL) {
        BT->Ltag=1; BT->Lchild=pre;
    }
    if (BT->Rchild == NULL)
        BT->Rtag=1; // 其后继目前未知，待其后继被访问时设置
    if (pre != NULL && pre->Rtag == 1) //设置当前节点前驱的后继
        pre->Rchild=BT;
    pre=BT; //修改前驱
    Inthreadbt(T->Rchild); //线索化右子树
}
```

6.4 线索二叉树

如何求中序线索二叉树中p节点的前驱?

if (p节点无左子) //p->Ltag一定等于1

    p->Lchild即是其前驱;

else p节点的前驱是p节点的左子树最右边的节点;

```
BTptr Inpre(BTptr p)  //求中序线索二叉树中p节点之前驱
{
    BTptr pre;
    if (p == NULL) return NULL;
    pre = p->Lchild;
    if (p->Ltag == 0) { //p节点有左子树
        while (pre->Rtag == 0) //获取以pre为根的子树中最右边的节点
            pre = pre->Rchild;
    }
    return pre;
}
```

6.4 线索二叉树

如何求中序线索二叉树中p节点的后继?

if (p节点无右子) //p->Rtag一定等于1

p->Rchild即是其后继;

else p节点的后继是p节点的右子树最左边的节点;

```
BTptr Insucc(BTptr p)  //求中序线索二叉树中p节点之后继
{
    BTptr succ;
    if (p == NULL) return NULL;
    succ = p->Rchild;
    if (p->Rtag == 0) { //p节点有右子树
        while (succ->Ltag == 0) //获取以succ为根的子树中最左边的节点
            succ = succ->Lchild;
    }
    return succ;
}
```

6.4 线索二叉树

3. 线索二叉树的遍历

以中序线索二叉树BT为例。

算法：

1）找到中序下的第1个节点;

```
p = BT;
while (p->Ltag == 0)
    p = p->Lchild;
```

2）访问之;

3）依次找后继访问;

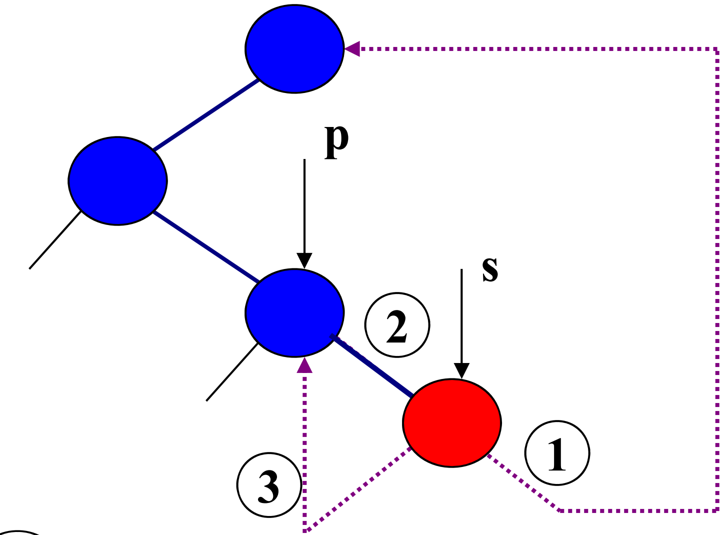
```
while (p != NULL) {
    visit(p);
    p = Insucc(p);
}
```

6.4 线索二叉树

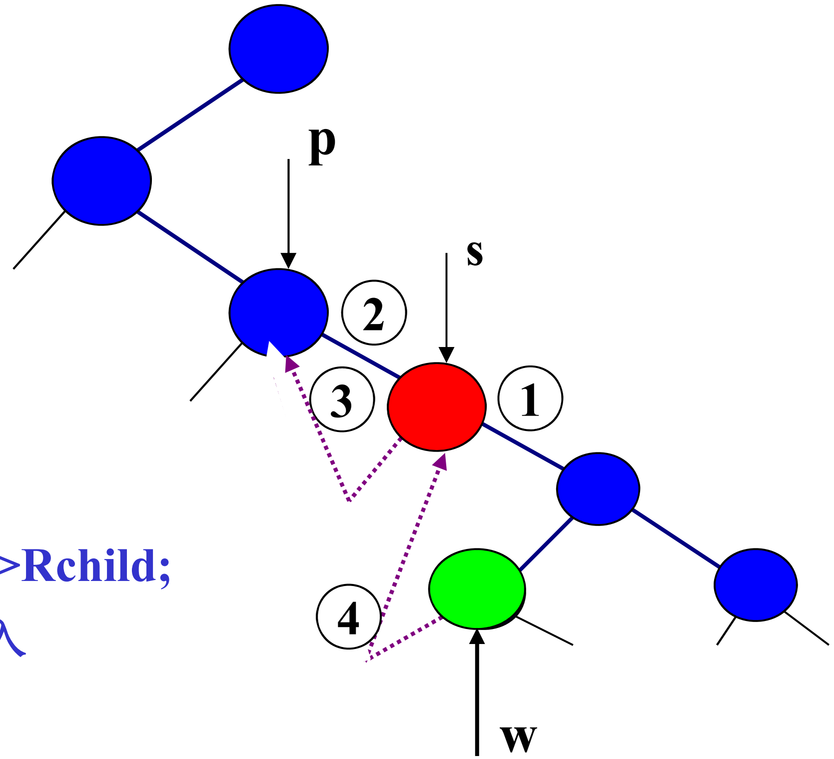
4. 线索二叉树的更新

中序线索二叉树某**p**节点之右插入**s**节点，也就是**s**作为**p**的右子（后继）插入。

(1) p节点右子=空：



(2) p节点右子存在：



1 s->Rtag = p->Rtag; s->Rchild = p->Rchild;

2 p->Rtag = 0; p->Rchild = s; // 插入

3 s->Ltag = 1; s->Lchild = p;

4 if (s->Rtag == 0) { w = Insucc(s); w->Lchild = s; }

6.5 树和森林

1. 树的存储结构

(1) 双亲表示法（顺序存储）

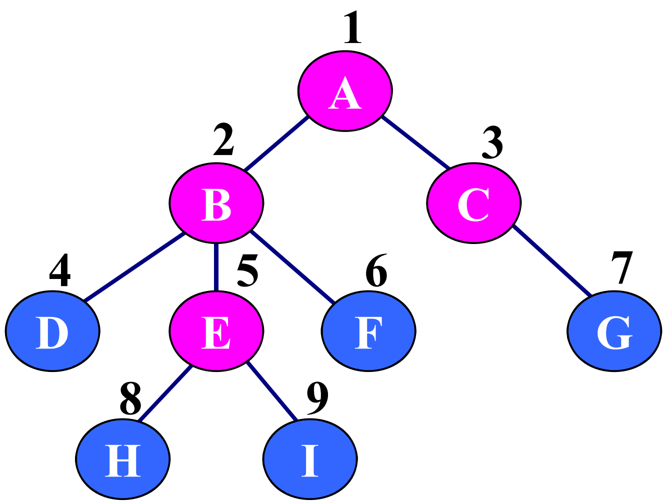
节点形式：

data	parent

data：数据值；**parent**：父节点的地址（或序号）。

```
typedef struct tnode {  //节点描述
    datatype data;
    int parent ;
} PTnode ;
typedef struct {
    PTnode nodes[MAXSIZE];  //树存储空间
    int n ;  //当前树的节点数
} Ptree ;
```

双亲表示法



Ptree pt;

	data	parent
pt.nodes[1]	A	0
2	B	1
3	C	1
4	D	2
5	E	2
6	F	2
7	G	3
8	H	5
9	I	5
10		
…		

优点：利用了每节点（除根外）只有唯一双亲的性质，故查找父节点很方便。

缺点：确定某结点的孩子节点需遍历整个树。如确定E的孩子，因E的序号（或下标）为5，扫描整个数组空间，查找出parent域为5的那些节点，即是E的孩子节点。

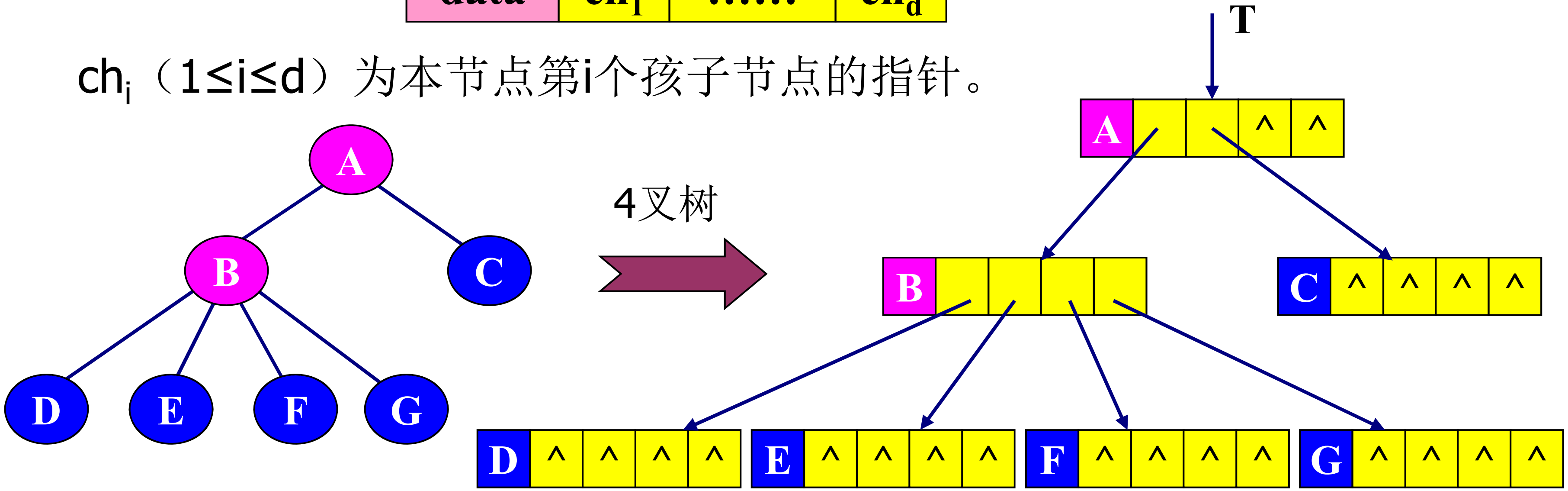
树的存储结构

(2) 孩子表示法（链式存储）

1) 固定指针数表示法：设树T的度为d（d叉树），即树中任一节点最多发出d个分支，所以节点定义为：

data	ch_1	……	ch_d

ch_i（1≤i≤d）为本节点第i个孩子节点的指针。



若树中节点数为n，非空指针数仅为n-1，而空指针数为**nd-(n-1)=n(d-1)+1**，显然当d很大时，浪费存储空间。

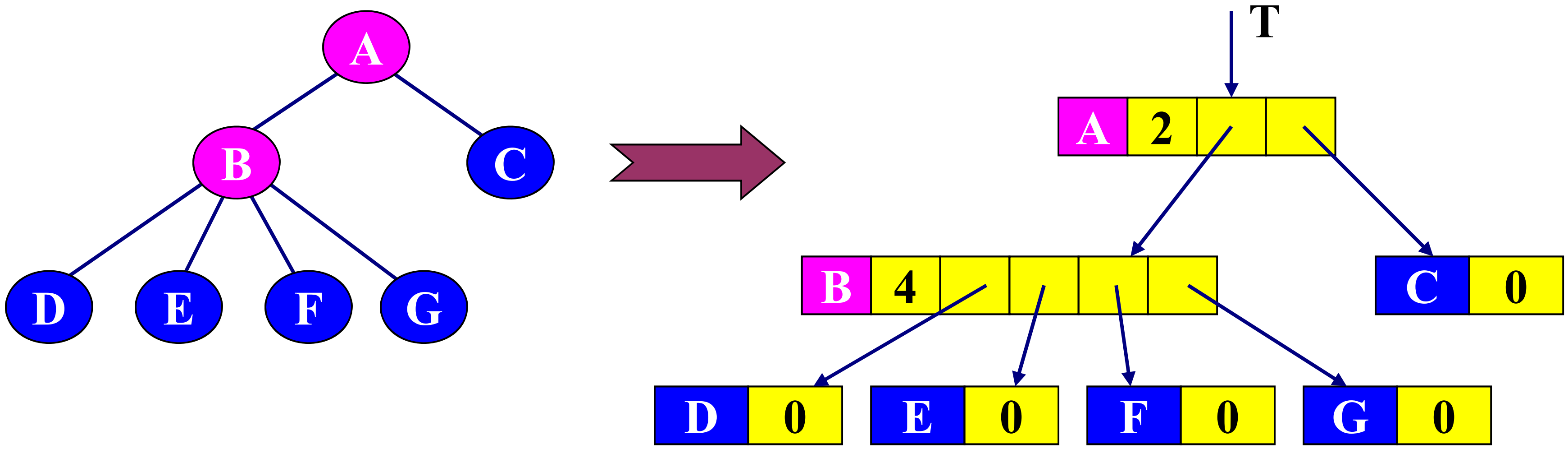
孩子表示法

2) 可变指针数表示法

节点形式：

data	**d**	**ch_1**	**……**	**ch_d**

d为本节点的出度，ch_i为第i个孩子节点的指针。节点的指针数= 其出度



此表示法其节点不规范，给节点的描述及树的操作带来不便。

树的存储结构

(3) 孩子链表示法（顺序存储+链式存储）

树中所有节点组成头节点表。头节点形式：

data	parent	fchild

data：数据值；parent：父节点的序号；**fchild：指向本节点第一个孩子的指针**

对于头节点表中的每个节点，以其为头，将该节点的所有孩子从左到右链成单链表。链表节点形式

child	next

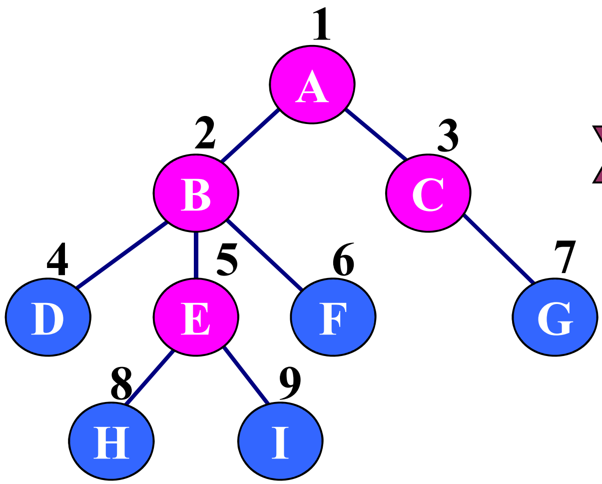
child：某孩子节点在头节点表的序号；**next：指向该孩子的右兄弟**

```
typedef struct node  {  //孩子链表节点
    int child;
    struct node * next;
} *chptr ;
typedef struct {  //头节点
    datatype data;
    int parent;
    chptr fchild;
} Tnode ;
```

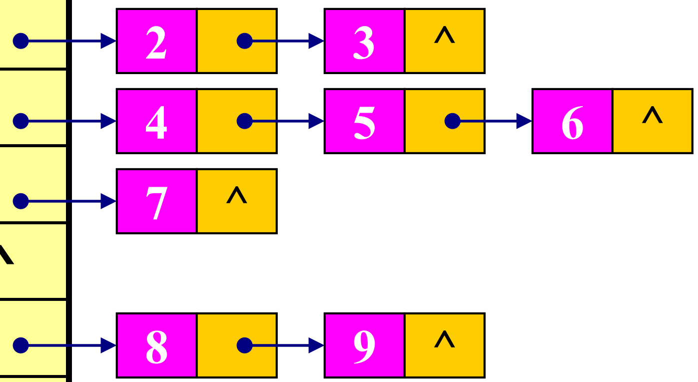
```
typedef struct  {
    Tnode nodes[MAXSIZE]; //头节点表
    int n;     //当前树中节点数
    int root; //根节点所在位置
} CHLtree;
```

孩子链表示法

CHLtree ct;



	data	parent	fchild
ct.nodes[1]	A	0	
2	B	1	
3	C	1	
4	D	2	^
5	E	2	
6	F	2	^
7	G	3	^
8	H	5	^
9	I	5	^
..	..	..	..



求父节点时，取相应节点的parent之值；找孩子时，搜索相应孩子链表。

说明：简单的孩子链表示可以不包含parent。

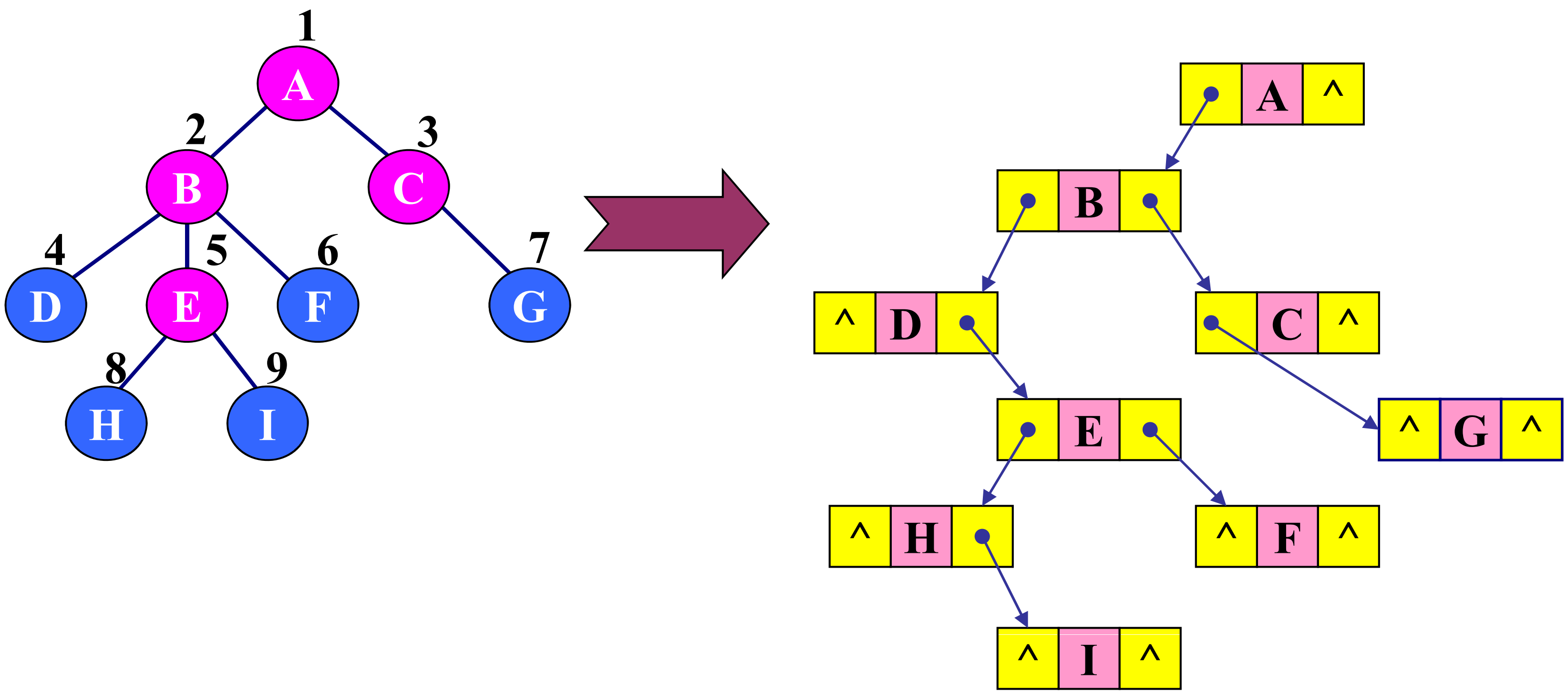
树的存储结构

(4) 孩子-兄弟表示法（或称二叉树表示法）

节点形式（同二叉树链式结构）：

fchild	**data**	**nextrb**

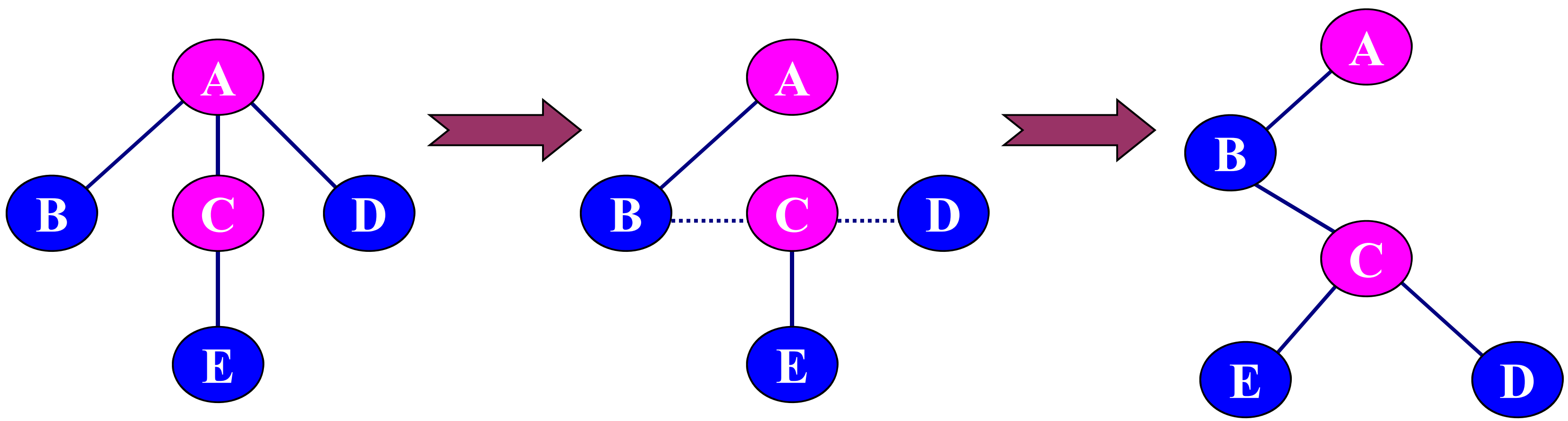
fchild：指向本节点第一孩子的指针；nextrbr：指向本节点右兄弟的指针。



2. 森林和二叉树的转换

(1) 树T转换成二叉树BT（T⇒BT）

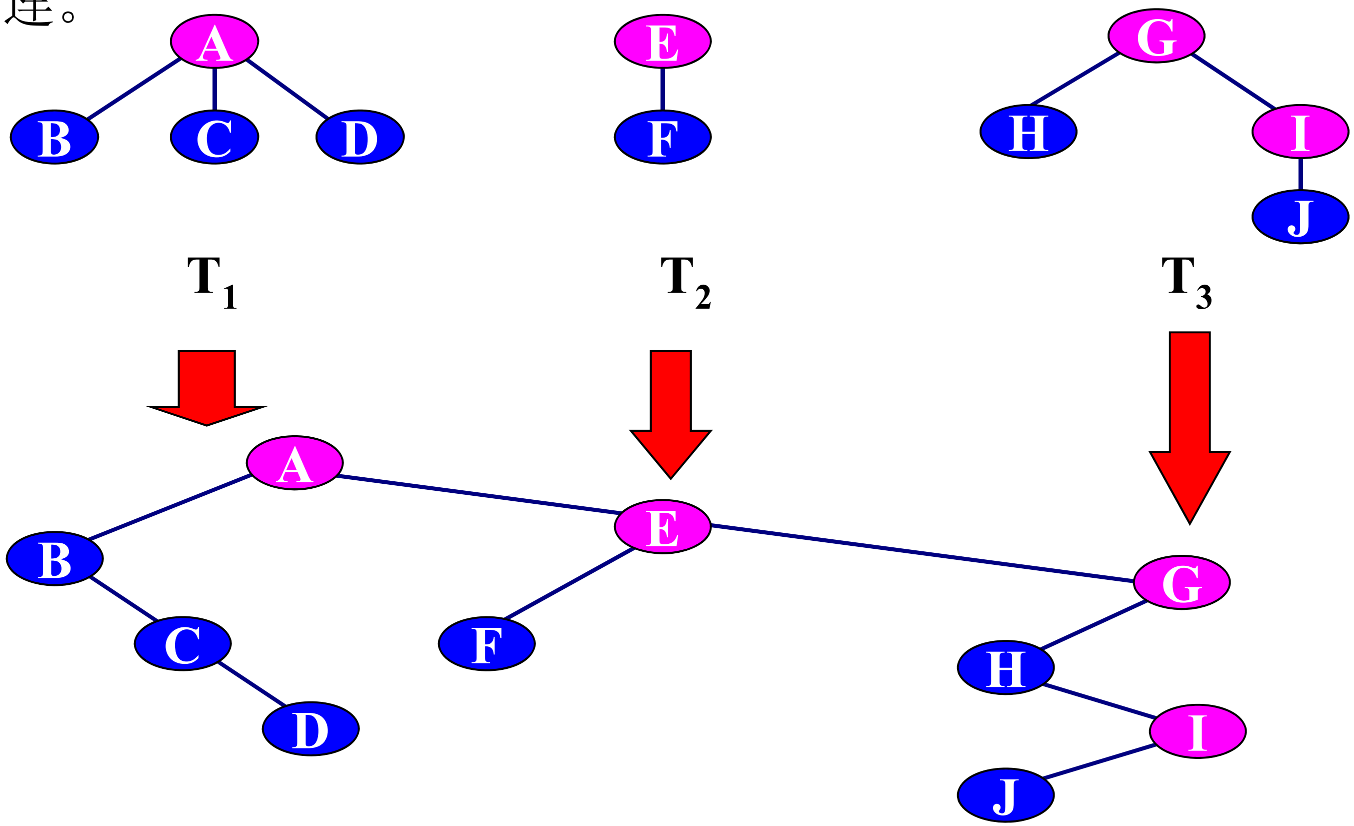
转换方法：左孩子，右兄弟。对树T中每一节点，以第一孩子作为左子，以右兄弟为其右子。转换后根节点的右子必为空。



森林和二叉树的转换

(2) 森林**F**转换成二叉树**BT(F⇒BT)**

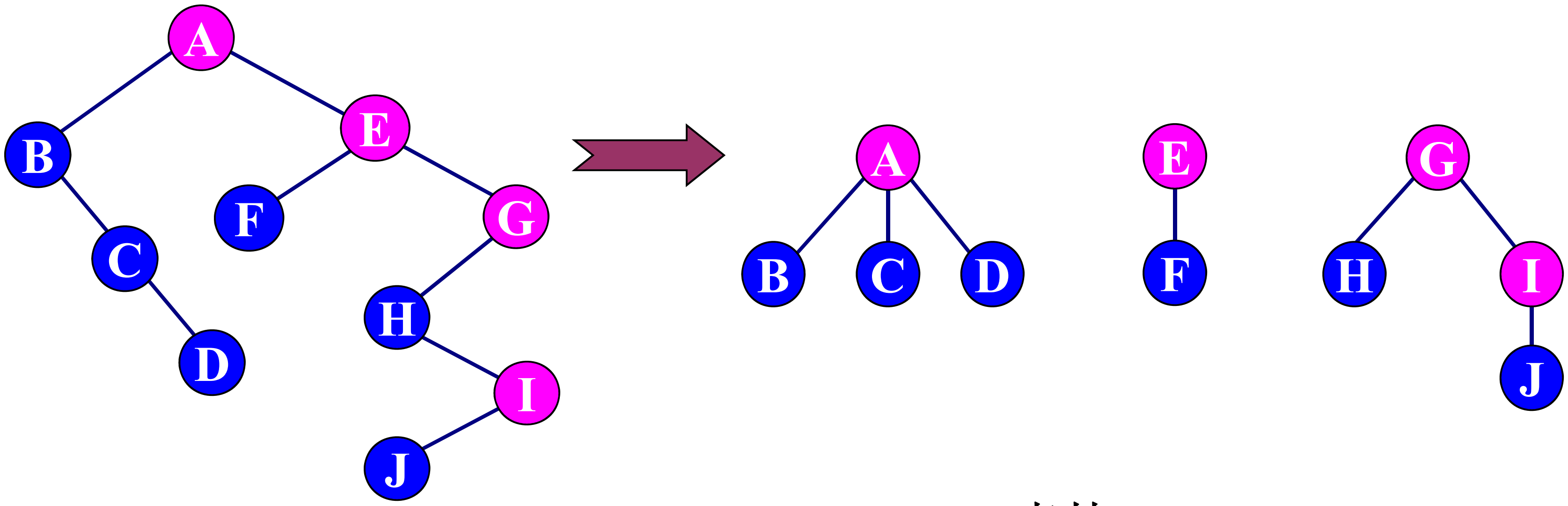
转换方法：先将F中各树转换成二叉树；然后各二叉树通过根的右指针相连。



森林和二叉树的转换

(3) 二叉树**BT**恢复成森林**F**（**BT⇒F**）

转换方法：对BT中任一节点，其Lchild所指节点仍为孩子，而Rchild所指节点为它的右兄弟，即“左孩子，右兄弟”。



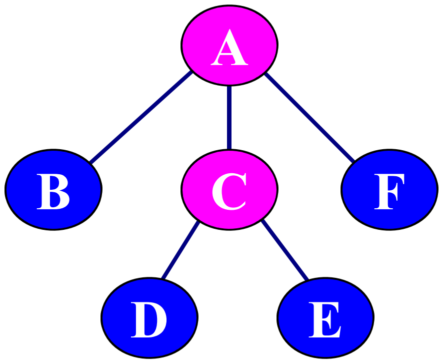
二叉树**BT**　　森林**F**

6.5 树和森林

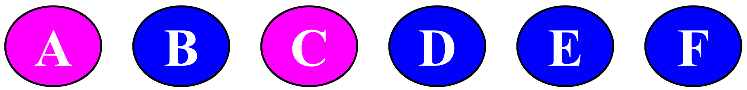
3. 树和森林的遍历

(1) 先根遍历树T

方法：若T≠∧，先访问T的根节点，然后从左至右依次先根遍历根下的各子树（递归）。



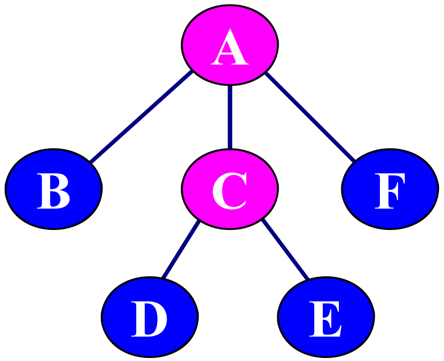
先根序列:



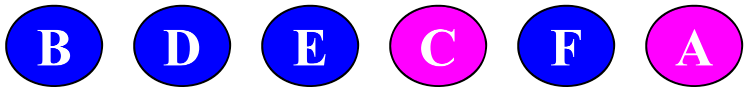
树和森林的遍历

(2) 后根遍历树**T**

方法：若T≠∧，从左至右后根遍历根下的各子树，最后访问根节点。



后根序列:



树和森林的遍历

(3) 先序遍历森林**F**

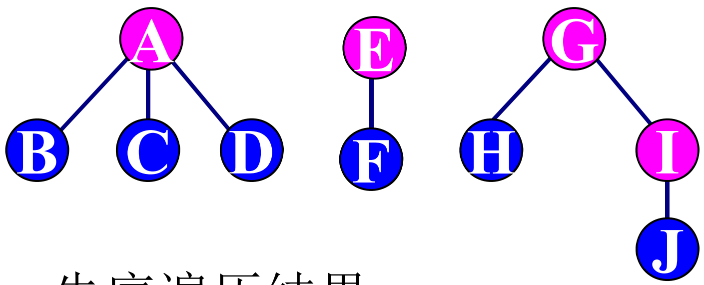
设F={ T_1, T_2, …, T_m }，其中T_i(1≤i≤m)为F中第i棵子树。

方法：若F≠Φ，则：

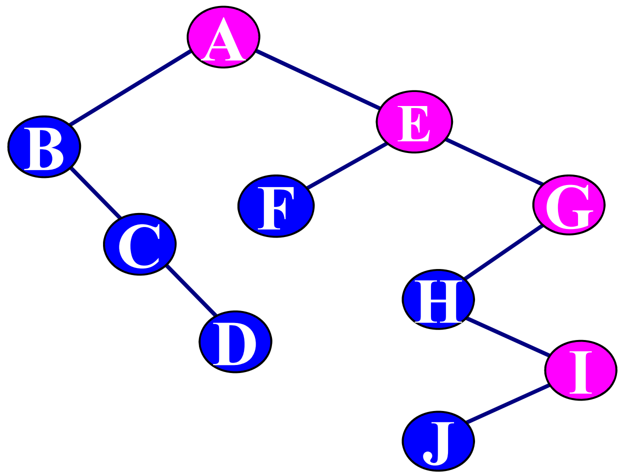
1）访问F中T_1之根；

2）先序遍历T_1之根下的各子树（子森林）；

3）先序遍历除T_1之外的森林（T_2, …, T_m）。



等价于：先将F转换成二叉树BT，然后对BT按前序（DLR）遍历



先序遍历结果：

A,B,C,D,E,F,G,H,I,J

DLR：A,B,C,D,E,F,G,H,I,J

树和森林的遍历

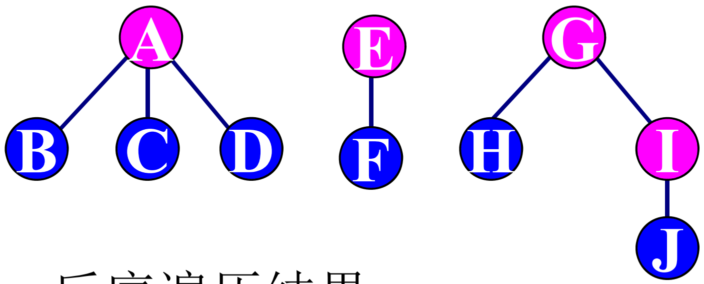
(4) 后序遍历森林**F**

若F≠Φ，则：

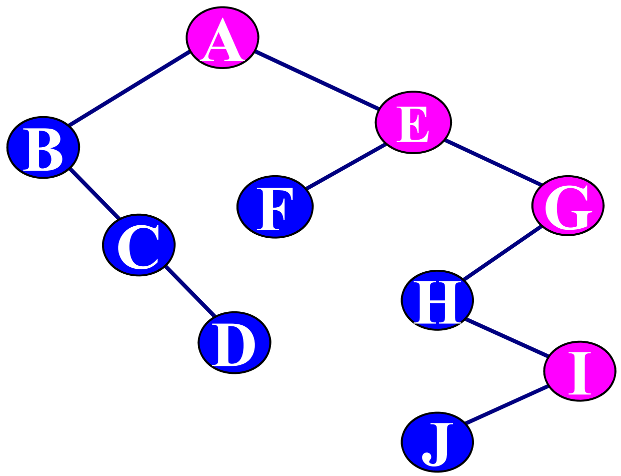
1）后序遍历F中T_1之根下的各子树（子森林）；

2）访问T_1之根；

3）后序遍历除T_1之外的森林{ T_2, …, T_m }。



等价于：先将F转换成二叉树BT，然后对BT按中序(LDR)遍历



后序遍历结果：

B,C,D,A,F,E,H,J,I,G）

LDR：B,C,D,A,F,E,H,J,I,G

6.6 二叉树应用举例

Huffman树及其编码和译码

1. Huffman树的引入

例 对学生的成绩s进行分类统计。

A：s >= 90，出现的概率是10%

B： 80 <= s < 90，出现的概率是40%

C： 70 <= s < 80，出现的概率是30%

D： 60 <= s < 70，出现的概率是15%

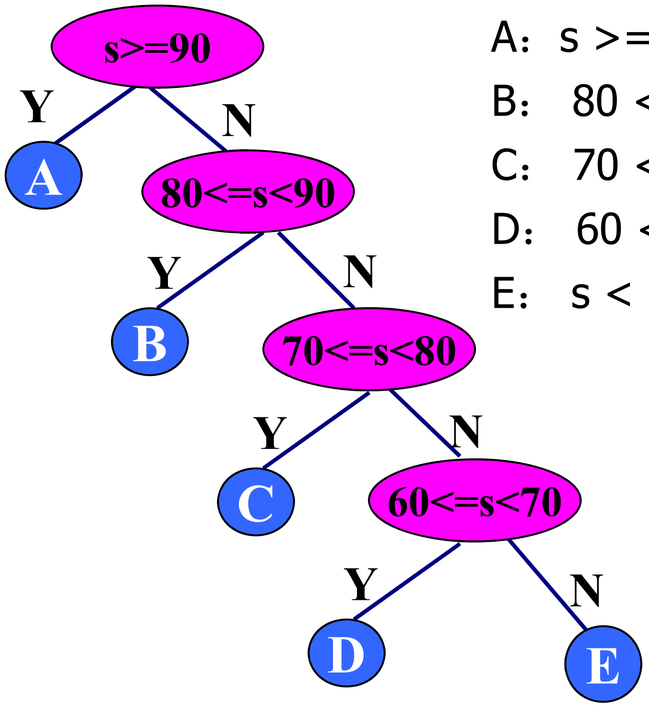
E： s < 60，出现的概率是5%

如何组织判断过程？

这里给出2种判断过程。

Huffman树的引入

1)



概率：

A：s >= 90，10%

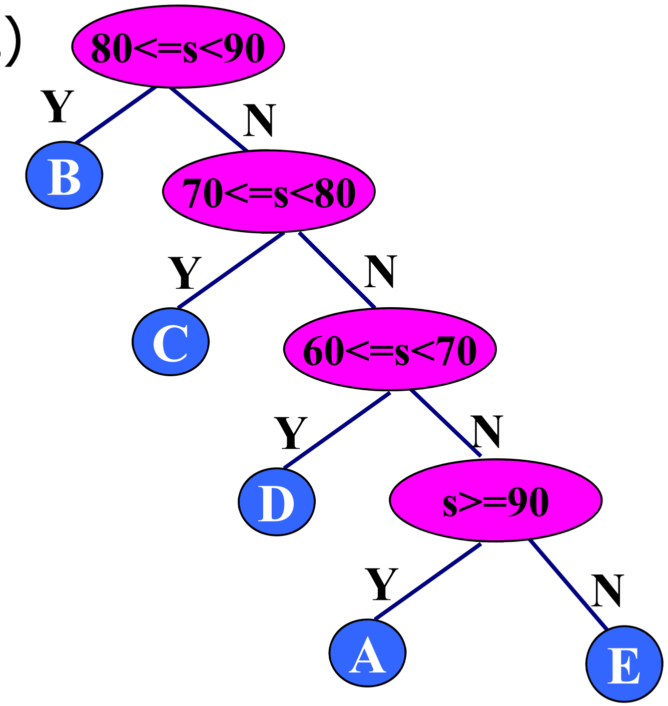
B： 80 <= s < 90，40%

C： 70 <= s < 80，30%

D： 60 <= s < 70，15%

E： s < 60，5%

2)



哪一种好？   看哪一种的平均判断次数少？

是Huffman树！

1）平均判断次数：10% * 1 + 40% * 2 + 30% * 3 + 15% * 4 + 5% * 4 = 2.6

2）平均判断次数：40% * 1 + 30% * 2 + 15% * 3 + 10% * 4 + 5% * 4 = 2.05

第 2）种好。

Huffman树

2. Huffman树及其构造算法

满足下列条件的二叉树为Huffman树 (设叶节点数为n)：简称H树

$$WPL = \sum_{k=1}^{n} L_k * W_k$$ 最小　　即加权路径长度最小

WPL(Weighted Path Length)：加权路径长度

L_k: 从根节点到叶节点k的路径长度（经过的分支数）

W_k: 叶节点k的权值（不同应用可表达不同含义）

如何构造一棵**Huffman**树 ？

思想：　“权值”越大的叶节点离根越近。

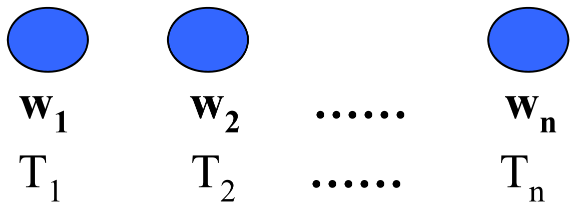
方法：从“权值”最小的2个节点开始，依次将“权值”最小的2个子树合并为1个新树。

Huffman树

Huffman树的构造算法（ Huffman算法）：

设给定权值集合$\mathbf{W=\{w_1, w_2, \ldots, w_n\}}$ ($n\geq1$)

① 初始化：先构成n棵单节点的二叉树森林$\mathbf{F=\{T_1,T_2, \ldots, T_n\}}$，其中$T_i$($1\leq i\leq n$)为只含有一个节点且带权值$w_i$的二叉树，表示为：



② 从当前F中选两棵根节点权值最小的树，作为左、右子树，构成一棵新的二叉树，新树根的权值为左、右子树根的权值之和；

③ 从F中删除选出的两棵树，同时加入新树；

④ 重复②、③，直到F中只含一棵树为止。最后的那棵二叉树即为**H**树。

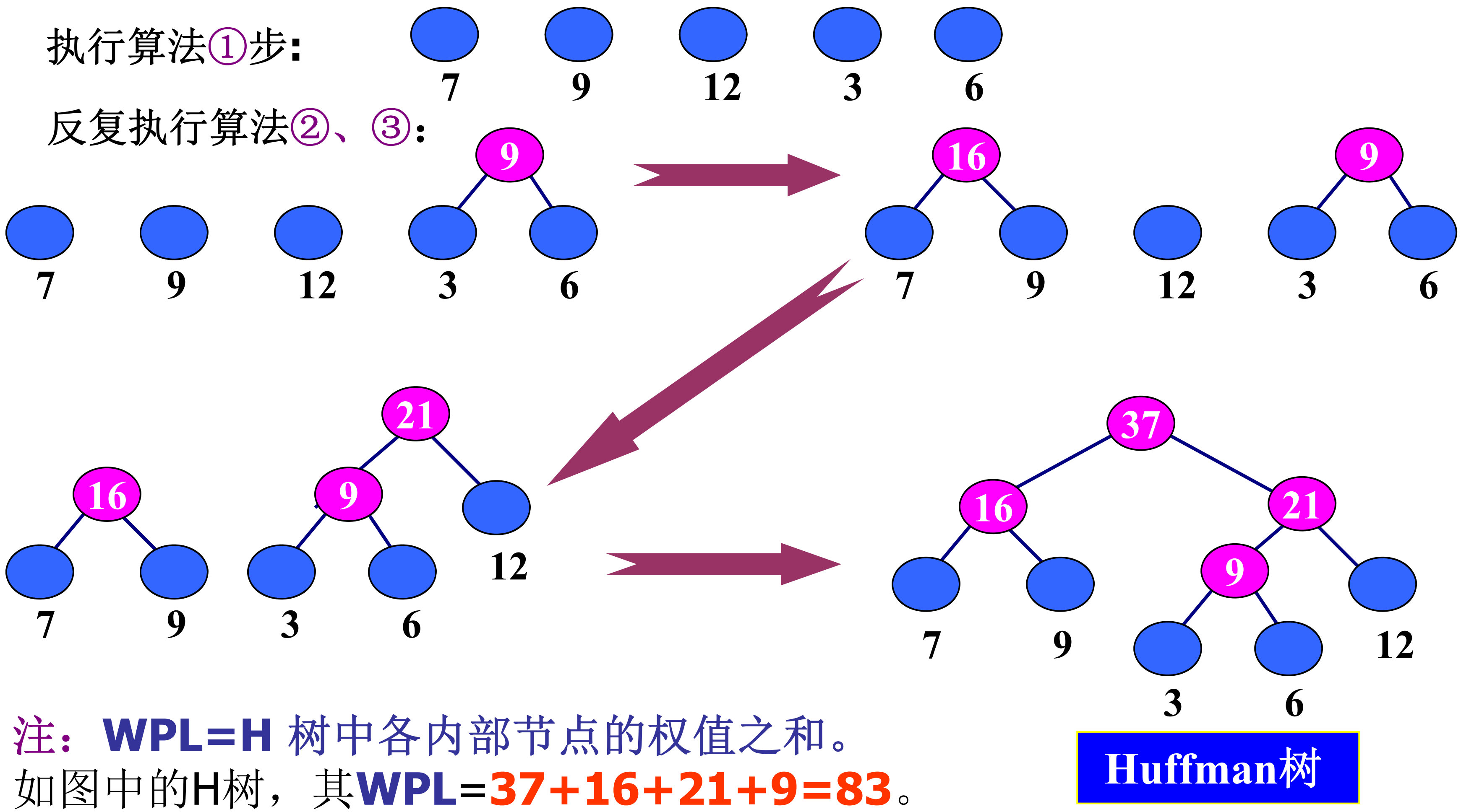
Huffman算法属于贪心算法（greedy algorithm）。

Huffman树

例 **6.27** 设**W={7，9，12，3，6}**，构造关于W的H树的过程：

执行算法①步:

反复执行算法②、③:



注：**WPL=H** 树中各内部节点的权值之和。

如图中的H树，其**WPL=37+16+21+9=83**。

Huffman树

说明：

- ✓ Huffman树不一定是完全二叉树。
- ✓ Huffman树不唯一。因为当有多个最小的“权值”时，任选2个即可。另外，谁左谁右无规定。

Huffman树的构造算法

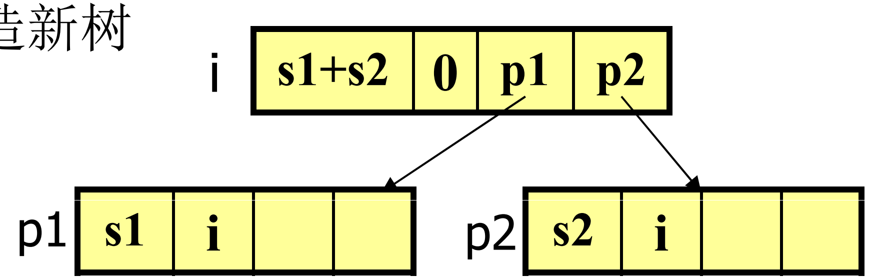
叶节点数=n（即给定的权值个数）
故H树中总的节点数m=2n-1。

```
#define n 8                    //设定权值数
#define m 2*n-1                //H树的节点数
typedef struct                 //定义节点
{ int wi;              //节点权值
   char data;       //该节点data值
   int parent ,Lchild ,Rchild;  //双亲及左、右子指针
}huffm;                        //H树节点说明符
void  HuffmTree (huffm HT[m+1] )   //构造H树的算法
{ int i, j, p1, p2;  int w, s1, s2;
   for (i=1,i<=m,i++)          //初始化
   { HT[i]. wi=0;
      HT[i].parent=0;
      HT[i].Lchild=HT[i].Rchild=0; }
```

wi	**data**	**parent**	**Lchild**	**Rchild**

Huffman树的构造算法

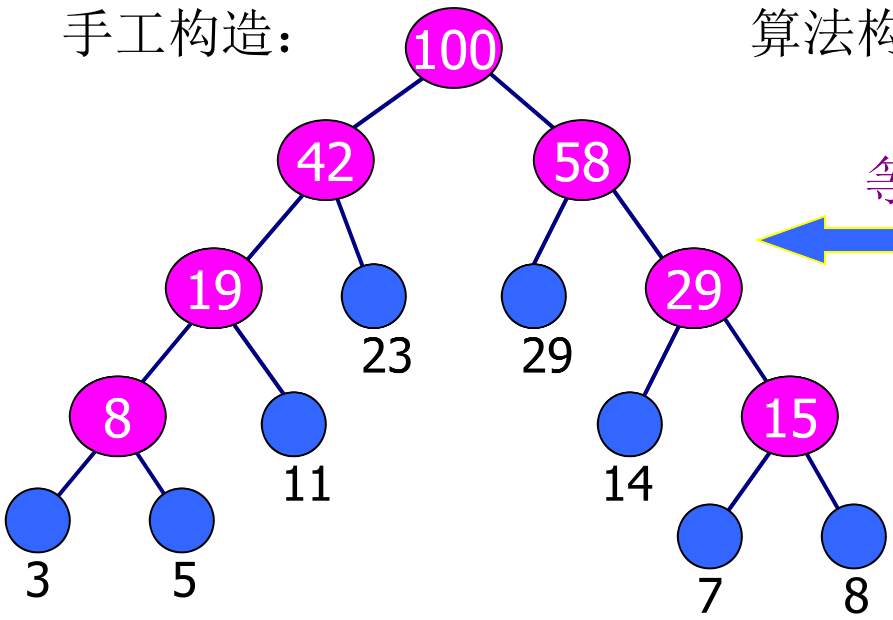
```
for (i=1; i<=n; i++)
  scanf("%d",& HT[i]. wi ] );    //读入权值
for (i=n+1; i<=m; i++)  //进行n-1次循环，产生n-1个新节点，构造H树
  {  p1=p2=0;              //p1、p2 为所选权值最小的根节点序号
     s1=s2=max;          //设max为机器能表示的最大整数
     for(j=1;j<=i-1;j++)       //从HT[1]~HT[i-1]中选两个权值最小的根节点
       if (HT[j].parent==0)
         if (HT[j]. wi<s1)
         { s2=s1; p2=p1;     //以j节点为第一个权值最小的根节点
           s1=HT[j].wi; p1=j; }
         else if (HT[j].wi<s2) {s2=HT[j].wi ; p2=j; }  //以j为第二个权值最小的根
     HT[p1].parent=HT[p2].parent=i; //构造新树
     HT[i].Lchild=p1; HT[i].Rchild=p2;
     HT[i].wi =HT[p1].wi +HT[p2].wi;
  }      //权值相加送新节点
}
```



Huffman树的构造

- 设w={5，29，7，8，14，23，3，11}
- n=8，m=15，执行算法HuffmTree(HT)

手工构造：



算法构造：

等价

	wi	pa	Lch	Rch
HT[1]	5	0	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0	0	0
8	11	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0

Huffman树及其编码和译码

3. Huffman编码及译码

问题：文件由若干字符组成。如何对字符进行编码，使得文件占用的空间少呢？

一种方法：使经常出现的字符编码尽量短，以压缩空间。

(1) **Huffman编码**：出现的频率越高，编码越短

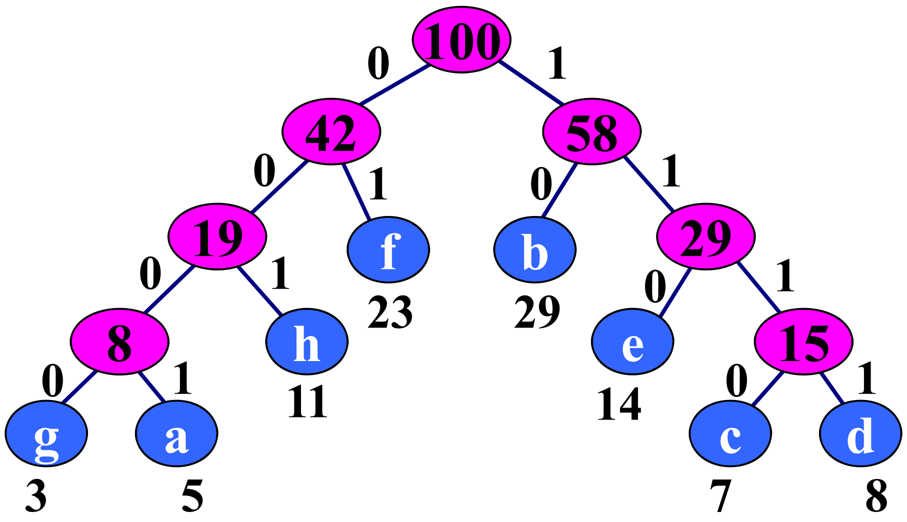
① 以待编码的所有字符作为叶节点，以出现频率作为权值，构造一棵Huffman树；

② 令所有的左分支取编码0，右分支取编码1，使每个叶节点都有唯一的从根出发的路径；

③ 每个字符的Huffman编码=从根到该字符叶节点路径的二进制代码。

Huffman编码

设待编码字符集为D＝｛a，b，c，d，e，f，g，h｝（n=8），各字符出现的概率分别为：0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11。可令W＝{5，29，7，8，14，23，3，11}。构造一棵H树：



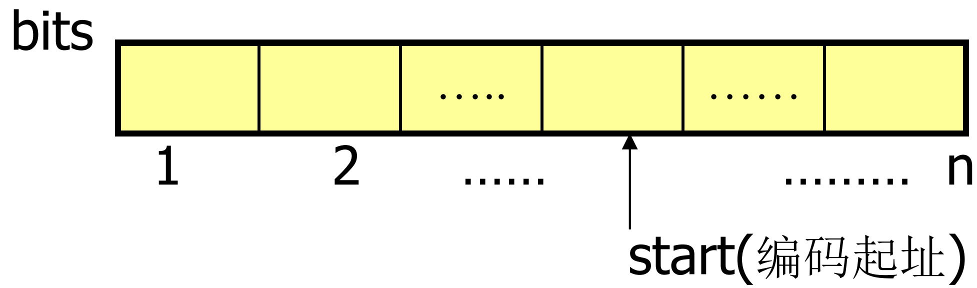
约定：从根开始，左分支取**0**，右分支取**1**

各字符的Huffman编码：

a： 0001
b： 10
c： 1110
d： 1111
e： 110
f： 01
g： 0000
h： 001

Huffman编码

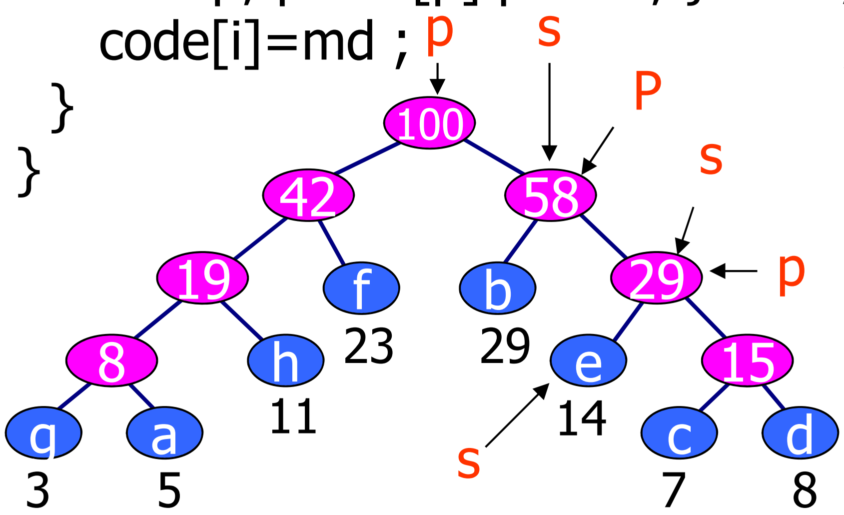
```
typedef struct        //存放一个字符的Huffman编码：
 {char bits[n+1];
   int start;
   char ch;  //待编码字符
 }ctype;
```



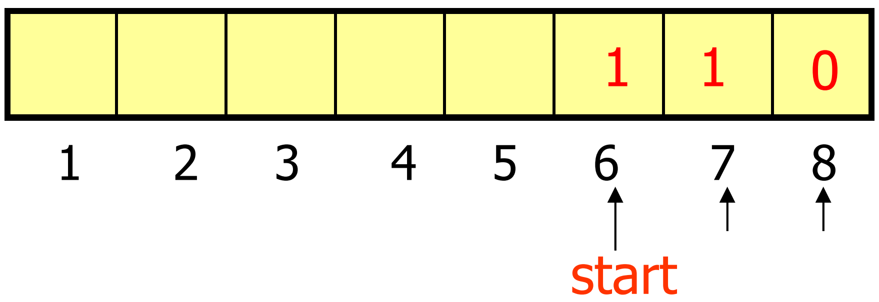

```
void Huffmcode(ctype code[n+1])      //求n个字符的Huffman编码的算法
 { int  i,  j,  p, s;  huffm  HT[m+1];    //H树存储空间
   ctype md;                            //存放当前编码的变量
   for (i=1;i<=n; i++)                   //读入待编码的字符
      HT[i].data=code[i].ch=getchar( );
   HuffmanTree(HT);                      //构造H树
```

Huffman编 码

```
for (i=1;i<=n;i++)        //求n个字符的Huffman编码
  { md.ch=code[i].ch;  md.start=n+1;
     s=i;                  //第i个字符地址（或下标）⇒s
     p=HT[i].parent;       //p为s之父节点地址
     while (p!=0)          //p存在时
      {md.start-- ;
        if (HT[p].Lchild==s)  md.bits[md.start]='0';  //左走一步为‘0’
        else md.bits[md.start]='1' ;                   //右走一步为‘1’
        s=p; p=HT[p].parent; }        //求下一位
     code[i]=md ;                     //存入第i字符的编码
  }
}
```



例：求 e 的编码 (i=s=5)



Huffman编码和译码

(2) Huffman编码的特点

- ✓ Huffman编码是一种前缀编码。

  前缀编码：任一字符的编码都不是其他字符编码的前缀（前面部分）

  前缀编码保证了译码时不会出现多种可能

- ✓ 若所编码的文件中字符出现的次数与预期相符，则Huffman编码最优（节省空间）；

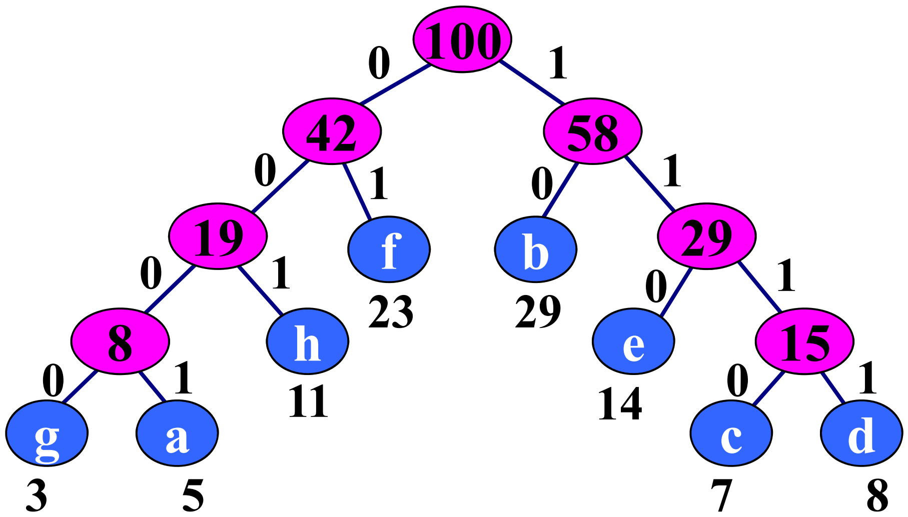
- ✓ 一般地，若字符出现次数变化范围越大，则Huffman编码越有效。

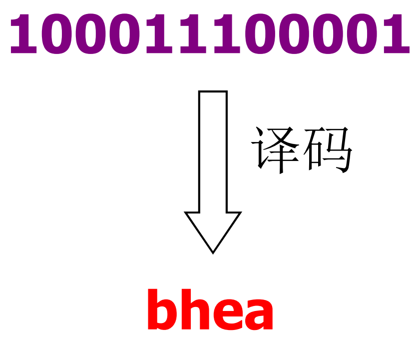
Huffman编码和译码

(3) Huffman译码

根据编码识别其对应的字符

从根出发，根据每一位的0或1选择左分支或右分支，直到到达叶节点，该叶节点对应的字符就是！重复上述译码过程，直到结束。





6.7 小结

- 树
  - 树、二叉树的定义、操作
  - 树、二叉树的性质
    - 树的4个性质
    - 二叉树的5个性质
  - 二叉树的存储结构
    - 顺序存储
    - 链式存储
  - 二叉树的遍历
    - DLR,LDR,LRD,层次
      - 递归算法
      - 非递归算法
  - 二叉树的线索化
    - 中序线索化算法; 求中序下前驱、后继; 线索树的遍历; 线索树节点的插入
  - 树的存储、树和森林与二叉树的转换、遍历
  - Huffman 树及其编码、译码

第6章 作业

1. 设m叉树中叶节点数=n，OD=2、3、……、m的节点数分别为n_2、n_3、……、n_m，证：

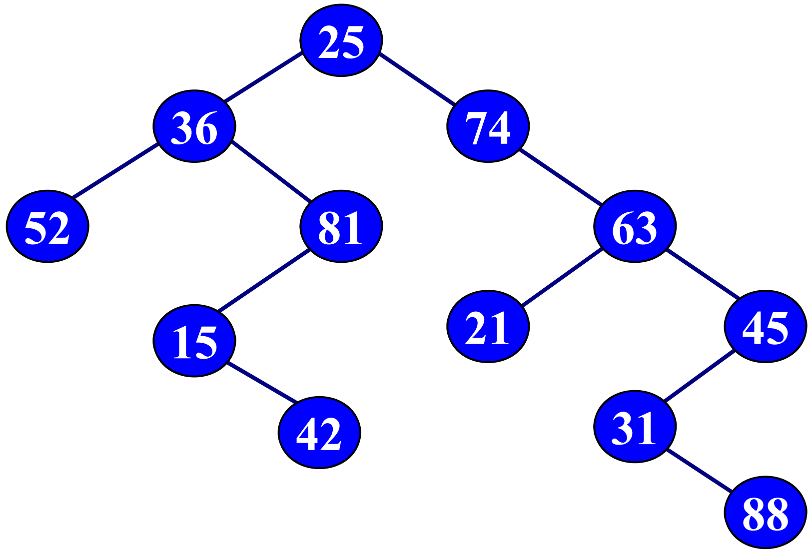
$$n=n_2+2n_3+\dots\dots+(m-1)n_m+1$$

2. 设 h(>1)层的完全二叉树中叶节点数=n，且第h层节点数≥ 2。证：

$$h=\lceil \log_2 n \rceil + 1$$

第6章 作业

3. 已知二叉树BT如下：



(1) 写出按DLR、LDR和LRD方法对BT遍历的结果序列，并计算相应满二叉树的节点个数；

(2) 画出BT的前、中、后序线索二叉树。

第6章 作业

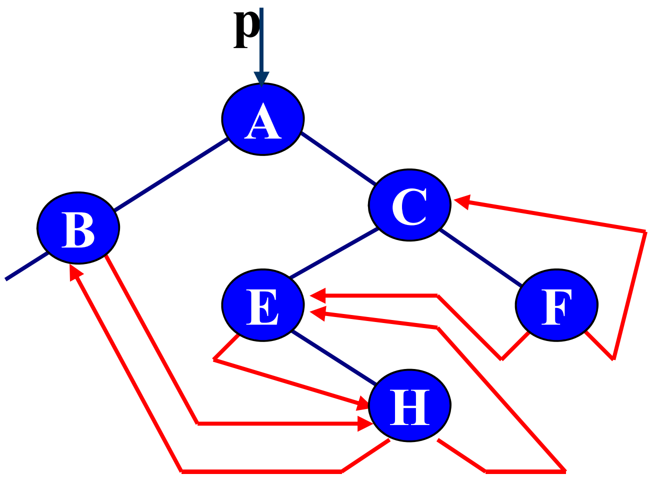
4. 设某二叉树的前、中序遍历序列为：

DLR：（A，B，C，D，E，F，G，H）

LDR：（B，D，C，A，E，G，F，H）

试画出此二叉树的逻辑结构。

5. 若后序线索二叉树中某p节点存在右子树，写出求p之右子树在后序下第一节点指针的算法(见图示, 红线为线索指针)。



第6章 作业

6. 设加权集合W={7，19，2，6，32，3，21，10}，试构造关于W的一棵Huffman树，并求该树的WPL。