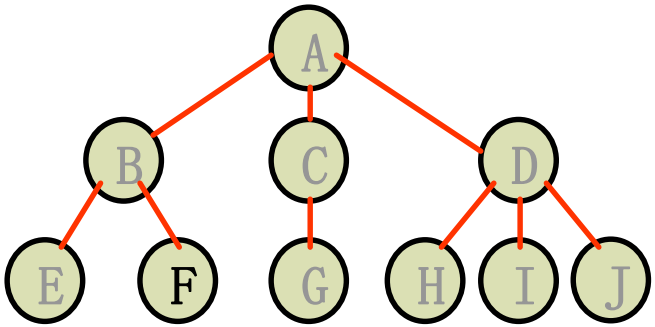
第六章　　树和二叉树

树型结构是一类重要的非线性结构。树型结构是结点之间有分支，并且具有层次关系的结构，它非常类似于自然界中的树。

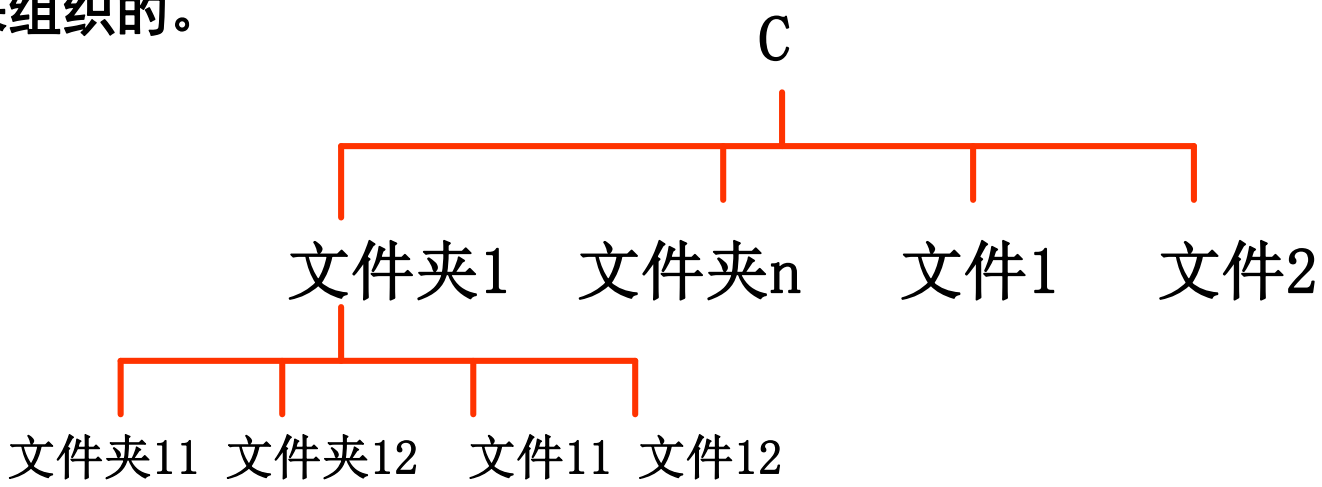
例1．家族族谱

设某家庭有10个成员A、B、C、D、E、F、G、H、I、J，他们之间的关系可用树表示：



例2　计算机的文件系统

不论是DOS文件系统还是window文件系统，所有的文件是用树的形式来组织的。



6.1 树和二叉树的定义

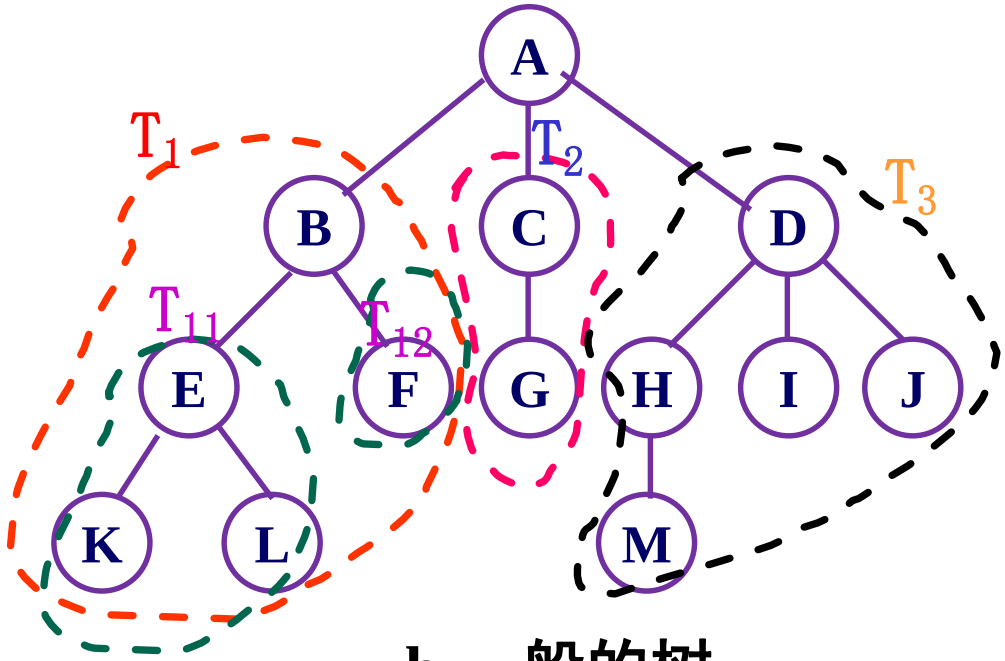
1. 树的定义和基本术语

定义：**树(Tree)**是n(n≥0)个结点的有限集T，T为空时称为空树，否则它满足如下两个条件：

（1）有且仅有一个特定的称为**根**的结点；

（2）当n>1时，其余的结点可分为m(m>0)个互不相交的子集$T_1, T_2, T_3 \ldots T_m$，其中每个子集又是一棵树，并称为根的**子树**。

a. 只有根结点的树



b.一般的树

树的基本术语：

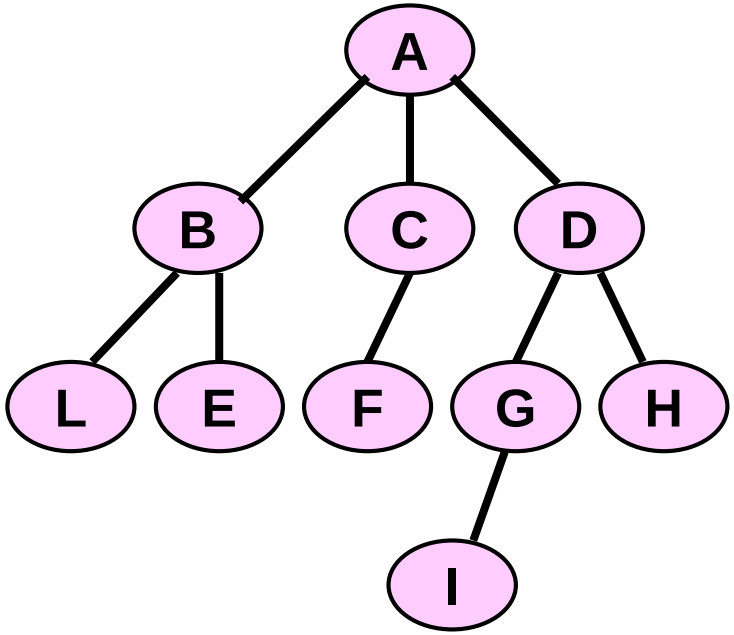
(1) 结点：树中的一个独立单元，包含一个数据元素 及若干指向其子树的分支。如图中的A、B、C、D等。

(2) 结点的度 ：结点拥有的子树数。

(3) 树的度：树中所有结点的度的最大值。

(4) 叶子结点：度为零的结点称为叶子结点或终端结点。

(5) 非终端结点：度大于零的结点。



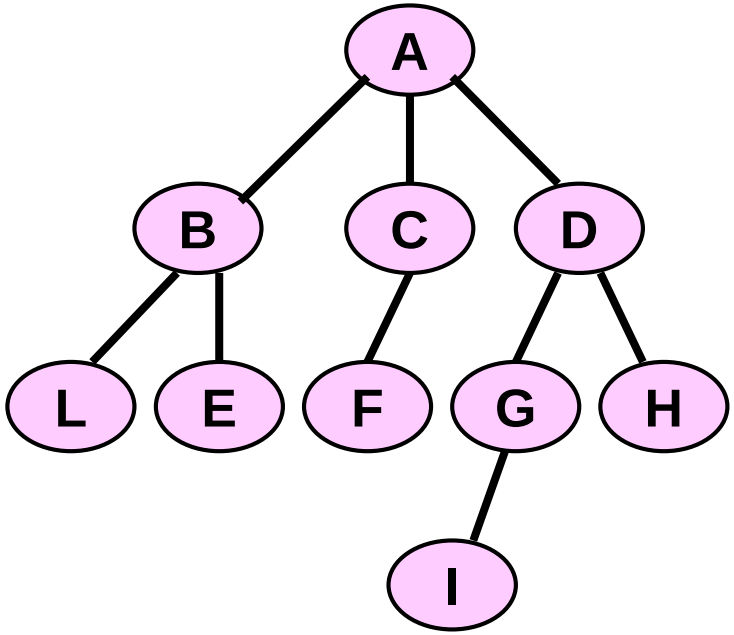
(6) 双亲和孩子： 结点的子树的根称为该结点的孩子，相应地，该结点称为孩子的双亲。

(7) 兄弟：同一个双亲的孩子之间互称兄弟。

(8) 堂兄弟：其双亲在同一层的结点互为堂兄弟。

(9) 祖先：结点的祖先是从根到该结点所经分支上的所有结点。

(10) 子孙：以某结点为根的子树中的任一结点都称为该结点的子孙。

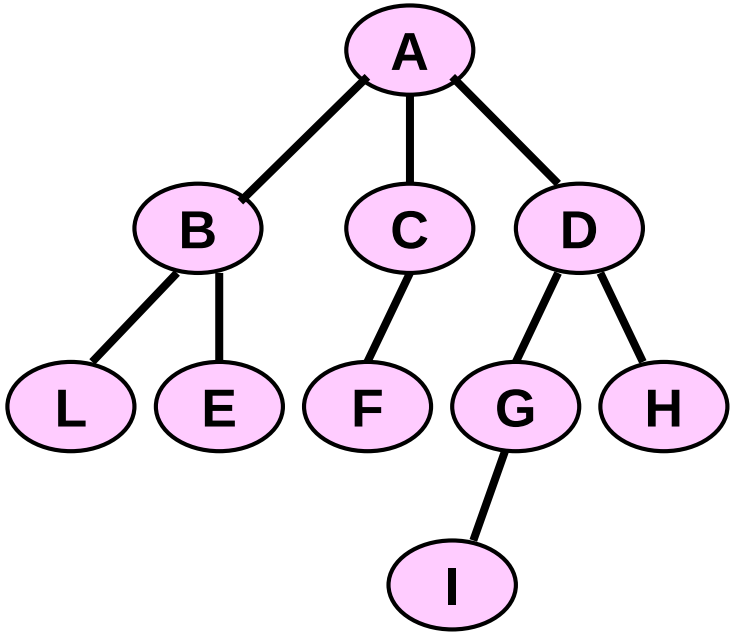


(11) 层次：结点的层次从根开始定义起，根为第一层，根的孩子为第二层。

(12) 树的深度：树中结点的最大层次称为树的深度，或高度。

(13) 有序树：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。

(14) 森林：是m（m≥0）棵互不相交的树的集合。

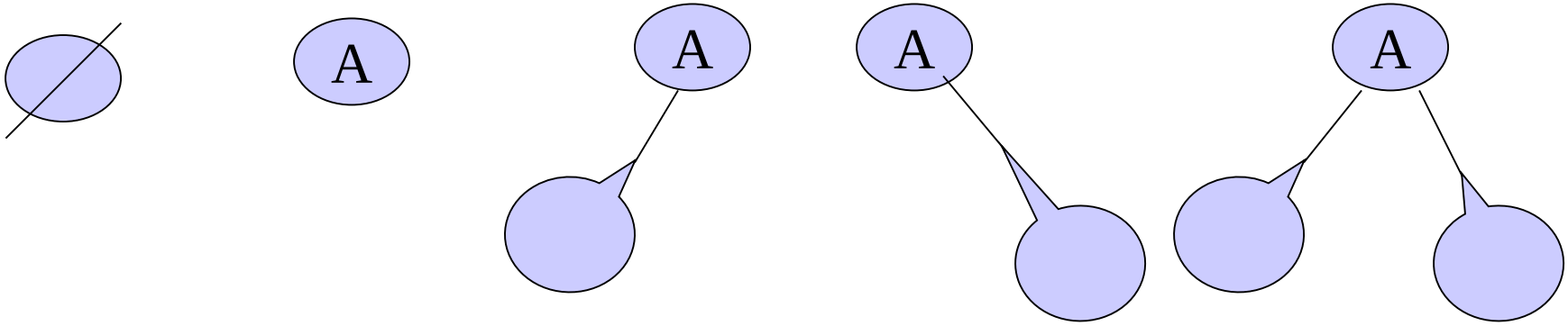


2. 二叉树的定义

二叉树是n(n≥0)个结点所构成的集合，它或为空树；或是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。

- 二叉树中每个结点最多有两棵子树，即二叉树每个结点度小于等于2;
- 二叉树的子树有左右之分，其次序不能颠倒。;

二叉树的五种基本形态：



(a)

空二叉树

(b)

仅有根结点的二叉树

(c)

右子树为空的二叉树

(d)

左子树为空的二叉树

(e)

左、右子树均非空的二叉树

5.2 二叉树的性质和存储结构

1. 二叉树的性质

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

采用归纳法证明此性质。

当i=1时，只有一个根结点，$2^{i-1}=2^0=1$，命题成立。

现在假定对所有的j($1 \leq j < i$)，命题成立，即第j层上至多有2^{j-1}个结点，那么可以证明j＝i时命题也成立。由归纳假设可知，第i-1层上至多有2^{i-2}个结点。

由于二叉树每个结点的度最大为2，故在第i层上最大结点数为第i-1层上最大结点数的二倍， 即$2\times 2^{i-2}=2^{i-1}$。

性质2：深度为k的二叉树至多有2^k-1个结点（$k \geq 1$)。

深度为k的二叉树的最大的结点为二叉树中每层上的最大结点数之和，由性质1得到每层上的最大结点数，因此：

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

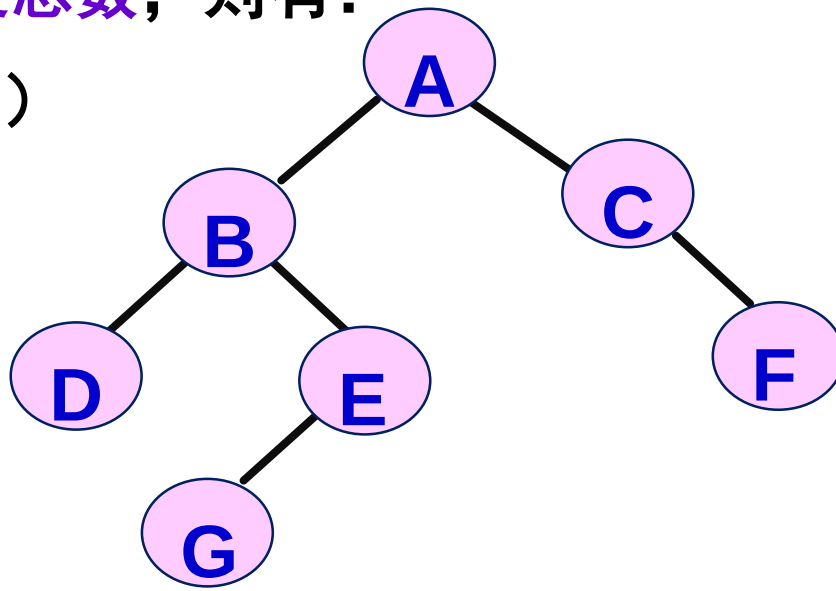
性质3： 对任何一棵二叉树，如果其终端结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。

设二叉树中度为1的结点数为n1，二叉树中总结点数为N，因为二叉树中所有结点均小于或等于2，所以有：

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

再看二叉树中的分支数，除根结点外，其余结点都有一个进入分支，设B为二叉树中的分支总数，则有：

$$N=B+1 \qquad (2)$$

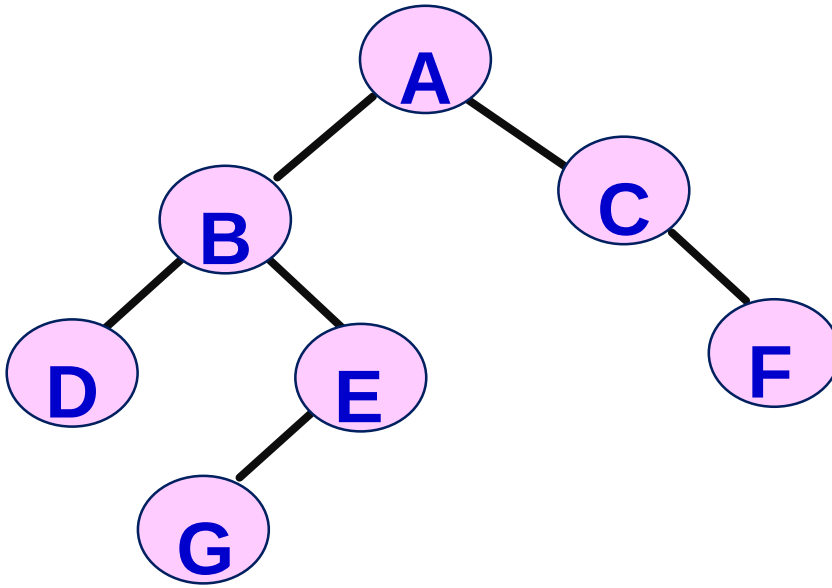


由于这些分支都是由度为1和2的结点发出的，所以有：

$$B=n1+2*n2 \qquad (3)$$

由式1、2、3得到：

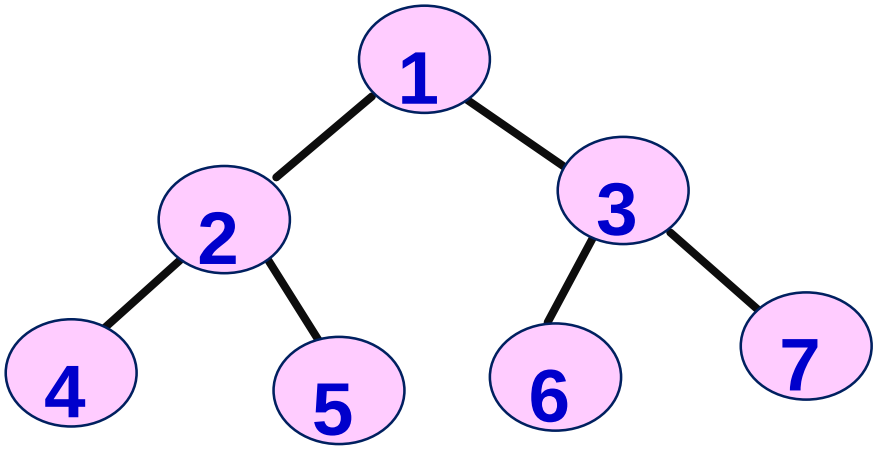
$$n0=n2+1$$



下面介绍两种特殊形态的二叉树：满二叉树和完全二叉树。

满二叉树：

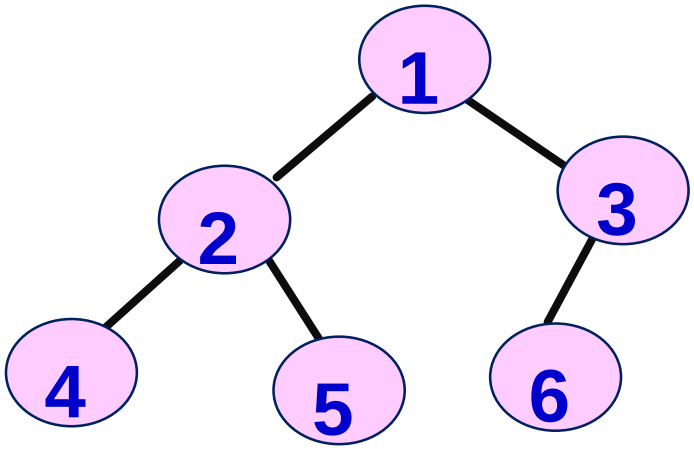
一棵深度为k且有2^k-1个结点的二叉树称为满二叉树。



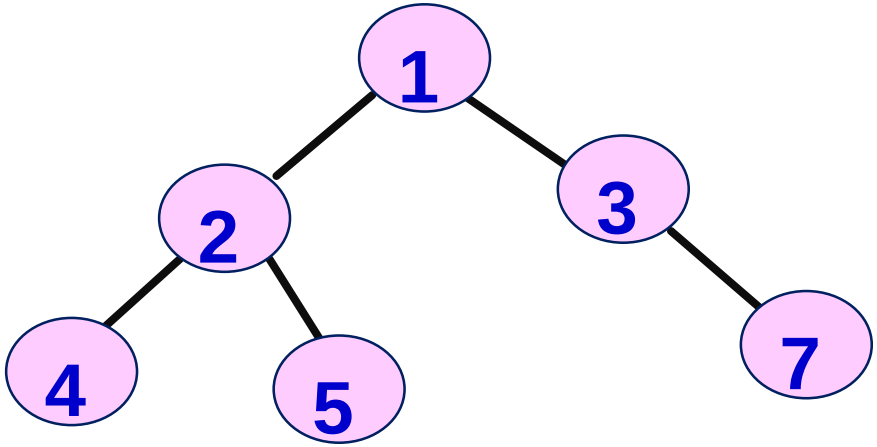
可以对满二叉树的结点进行连续编号，约定编号从根结点起，自上层至下层，每层自左至右。

完全二叉树：

深度为k的，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，称之为完全二叉树。



(a) 完全二叉树

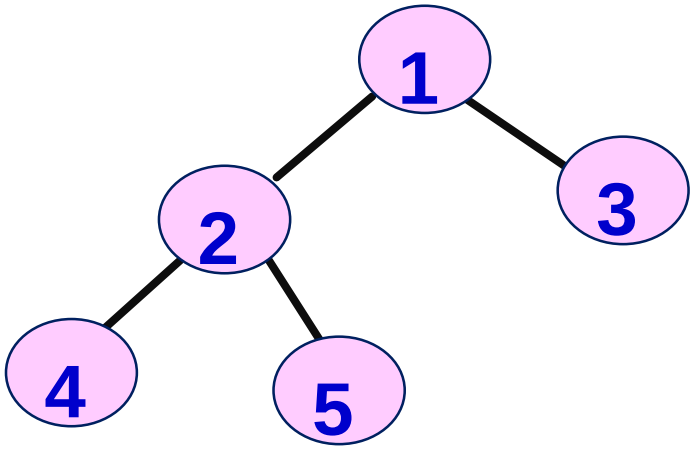


(b) 非完全二叉树

深度为K的完全二叉树的特点是：

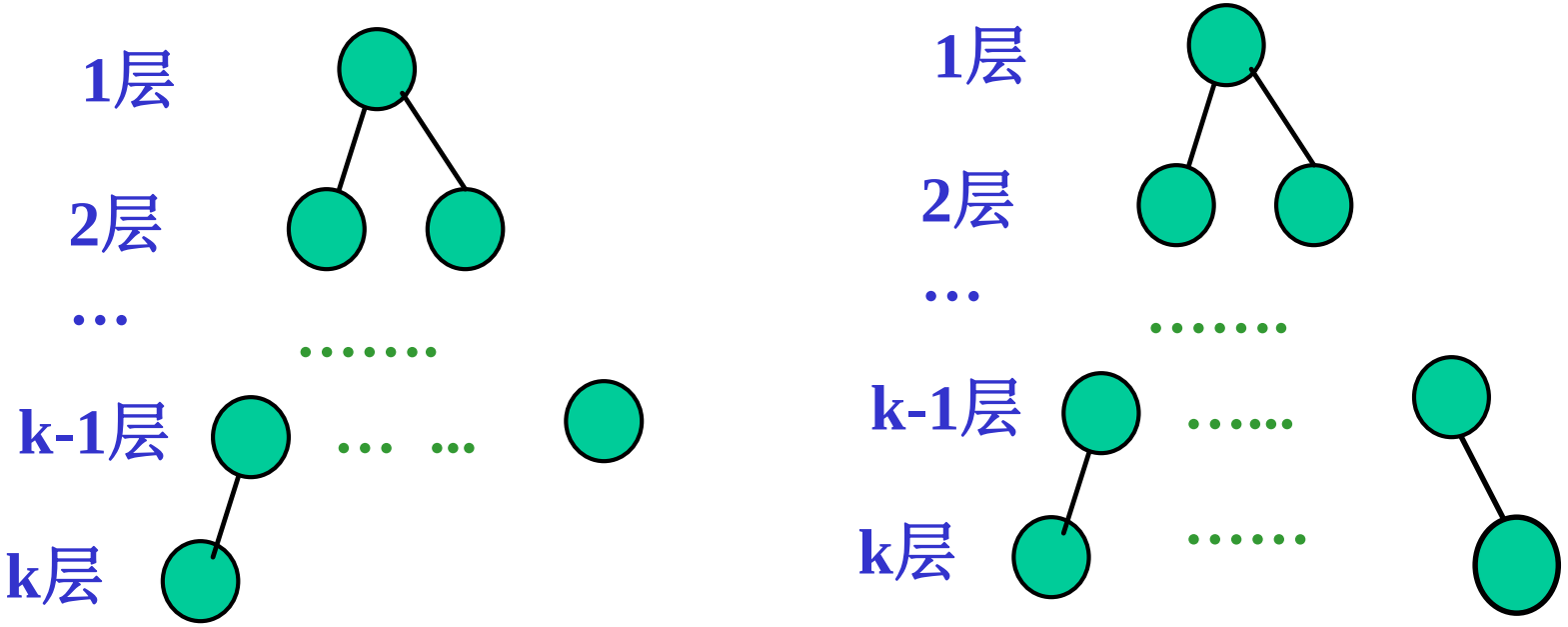
（1）叶子结点只可能出现在第k层或k-1层。

（2）任一结点，如果其右子树的最大层次为L，则其左子树的最大层次为L或L＋1。



性质4：具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$。

假设此二叉树的深度为k，则如下图所示：



根据性质2及完全二叉树的定义得到：

$2^{k-1}-1<n\leqslant 2^k-1$ 或 $2^{k-1} \leqslant n<2^k$

于是：$k-1 \leqslant \log_2 n < k$

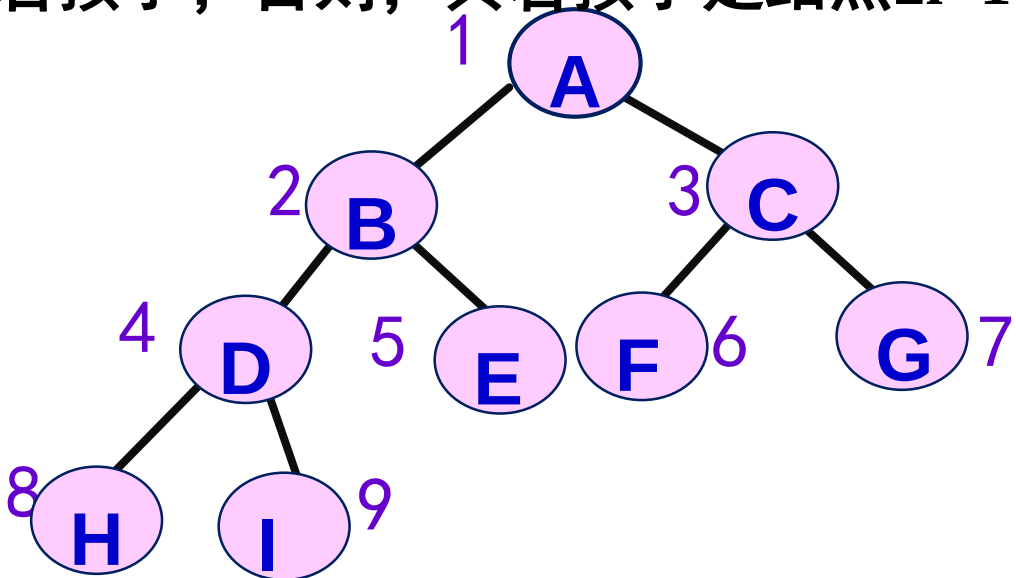
因为k是整数，所以有：$k=\lfloor \log_2 n \rfloor + 1$

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(从第1层到第 $\lfloor \log_2 n \rfloor + 1$层，每层从左到右)，则对任一结点i(1≤i≤n)，有：

(1) 如果i=1，则结点i无双亲，是二叉树的根；如果i>1，则其双亲是结点 $\lfloor i/2 \rfloor$

(2) 如果2i>n，则结点i为叶子结点，无左孩子；否则，其左孩子是结点2i。

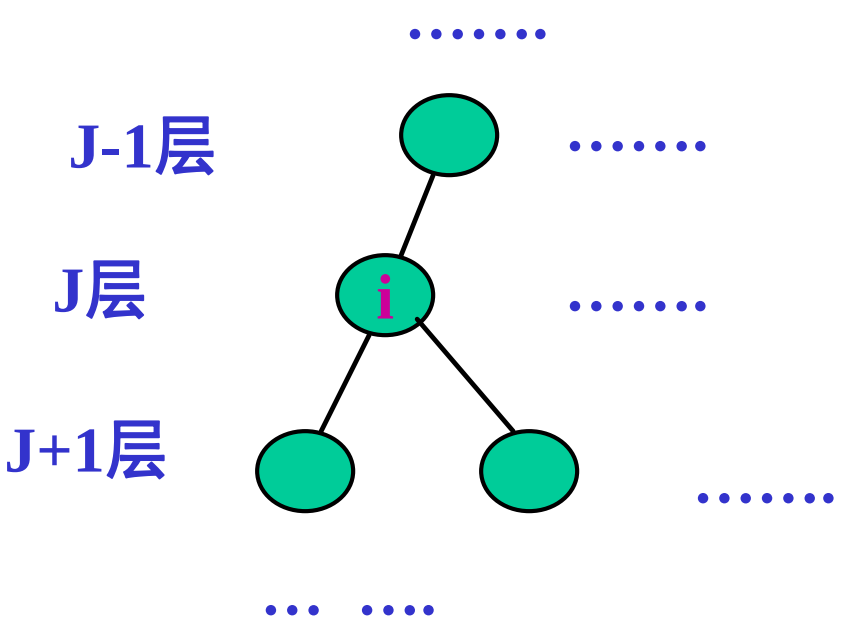
(3)如果2i+1>n，则结点i无右孩子；否则，其右孩子是结点2i+1。



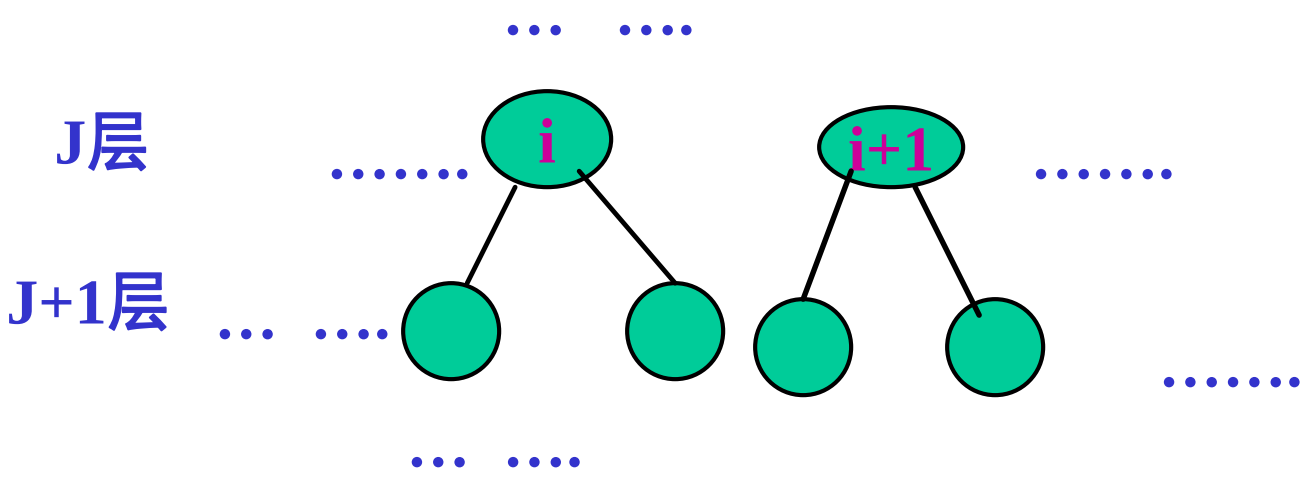
i＝1，由完全二叉树的定义，其左孩子是结点2，若2>n，即不存在结点2，此时，结点i无左孩子。结点i的右孩子也只能是结点3，若结点3不存在，即3>n，此时结点i无右孩子。

对于i>1，可分为两种情况：

（1）设第j层的第一个结点的编号为i，



（2）假设第j层上的某个结点编号为i

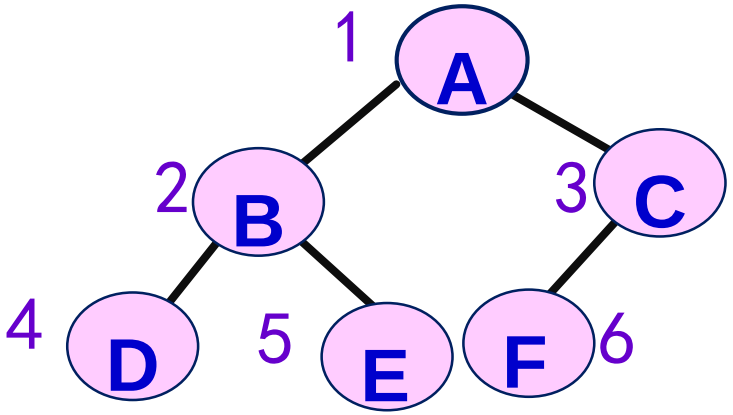


2. 二叉树的存储结构

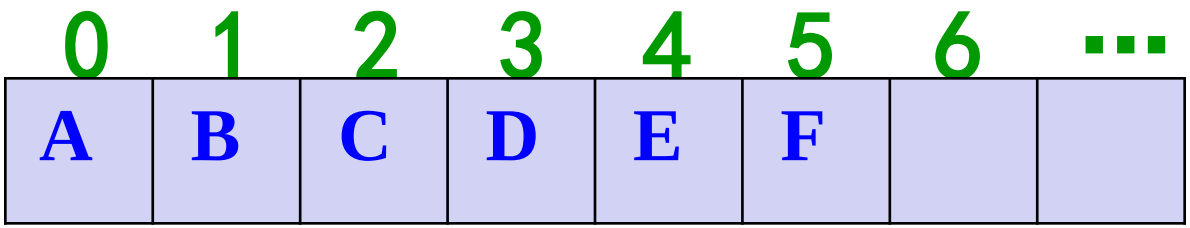
(1). 顺序存储结构

完全二叉树的顺序结构：

用一组连续的内存单元，按编号顺序依次存储完全二叉树的元素。

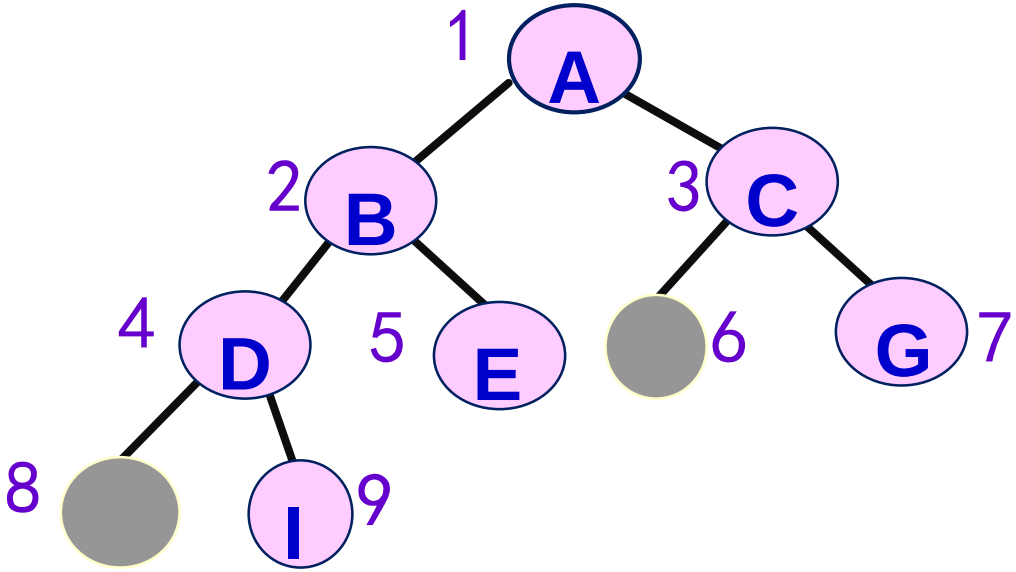


顺序结构图示：



一般二叉树的顺序结构：

将二叉树的结点与完全二叉树的结点相对照，按编号存储到内存单元“相应”的位置上。



1	2	3	4	5	6	7	8	9	10	…
A	B	C	D	E		F		G		

顺序存储结构有可能对存储空间造成极大的浪费，

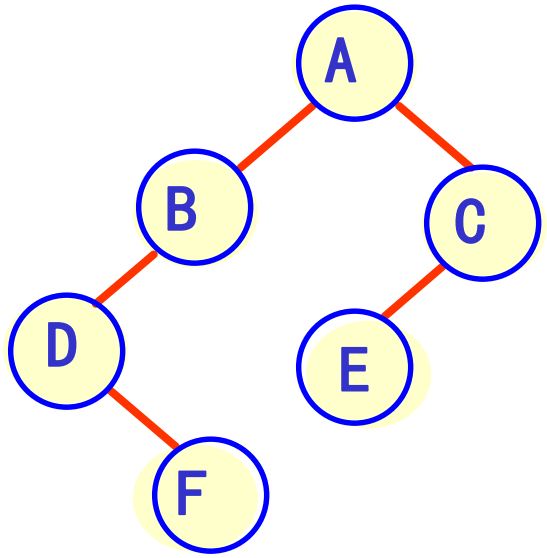
在最坏的情况下，一个深度为k且只有k个结点的单支树却需要2^k-1个结点存储空间。

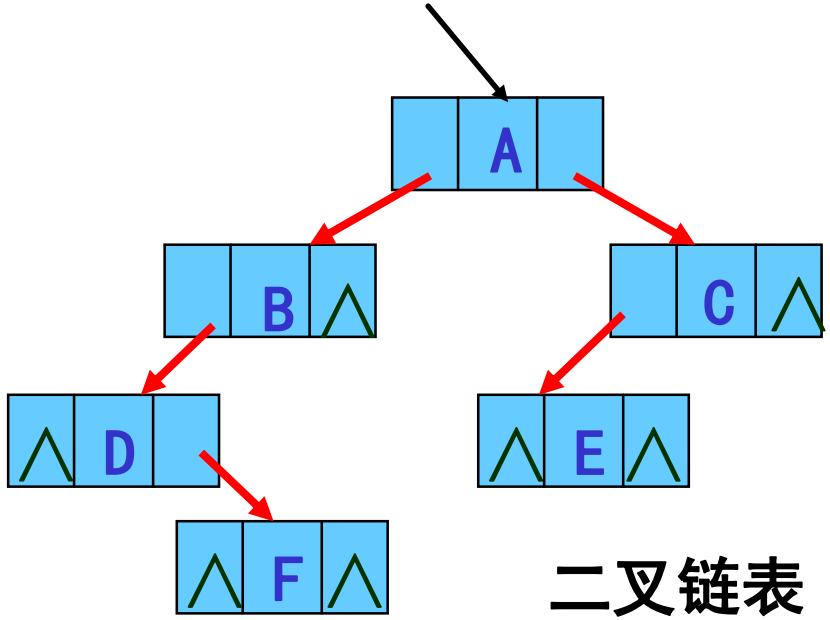
(2). 链式存储结构

二叉链表:

一个二叉树的结点由一个数据元素和分别指向左、右子树的两个分支构成。则表示二叉树的链表中的结点至少包含三个域：数据域和左、右指针域。

结点的结构：





二叉链表

二叉链表的类型定义：

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

三叉链表

三叉链表中每个结点包含四个域：数据域、双亲指针域、左指针域、右指针域。

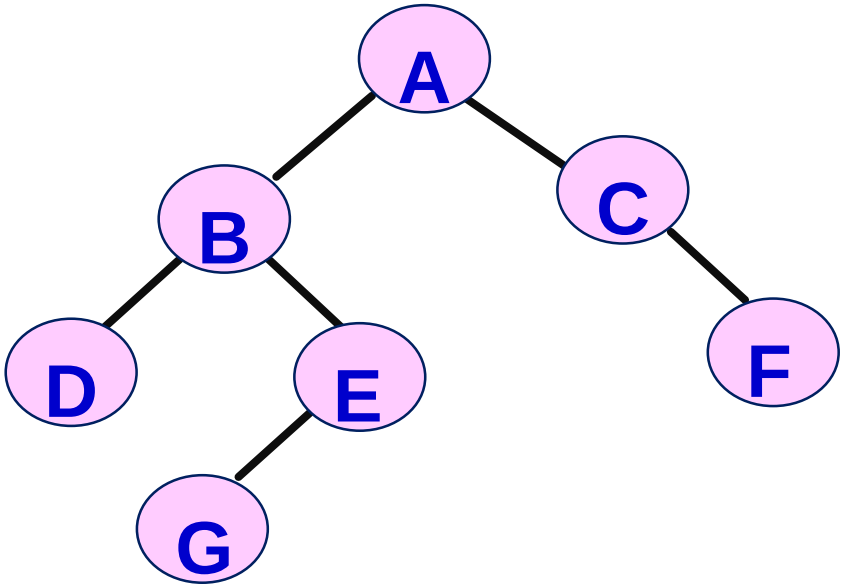
结点的结构：

6.3　遍历二叉树

1. 遍历二叉树(Traversing Binary Tree)

顺着某一条搜索路径巡访二叉树中的结点，使得每一个结点均被访问一次，而且仅被访问一次。

访问：含义很广，可以是对结点的各种处理，如修改结点数据、输出结点数据。



遍历的实质是对二叉树进行线性化的过程，即遍历的结果是将非线性结构的树中结点排成一个线性序列。二叉树是非线性结构，每个结点有两个后继，如何访问二叉树的每个结点，而且每个结点仅被访问一次？。

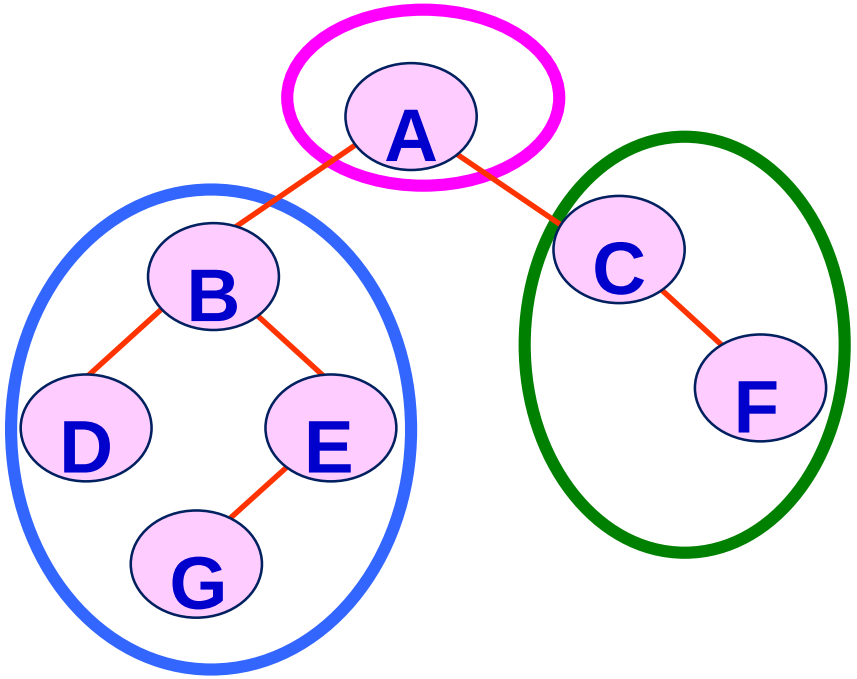
二叉树由根、左子树、右子树三部分组成

二叉树的遍历可以分解为：访问根，遍历左子树和遍历右子树

令：L：遍历左子树

D：访问根结点

R：遍历右子树



约定先左后右，有三种遍历方法： DLR、LDR、LRD ，分别称为先序遍历、中序遍历、后序遍历

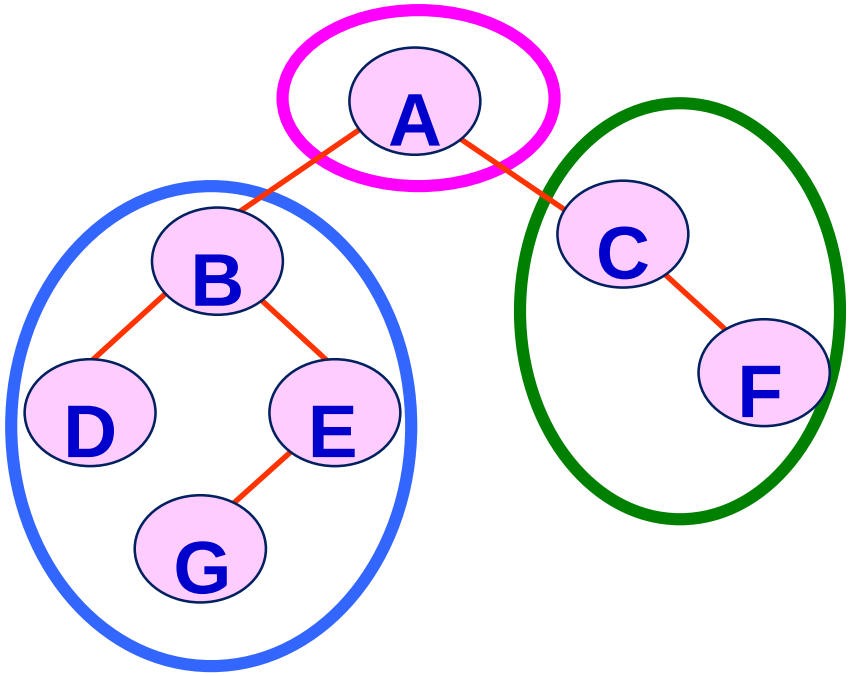
先序遍历（DLR）

若二叉树非空

（1）访问根结点；

（2）先序遍历左子树；

（3）先序遍历右子树；



先序遍历序列：A, B, D, E, G, C, F

根　左子树　右子树

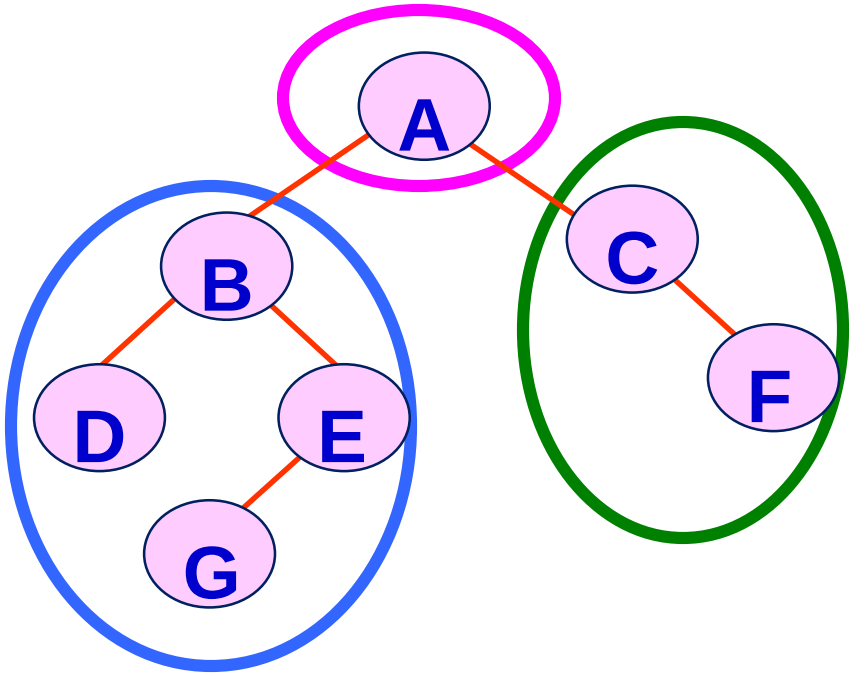
中序遍历（LDR）

若二叉树非空

（1）中序遍历左子树

（2）访问根结点

（3）中序遍历右子树



中序遍历序列： D, B, G, E, A, C, F

左子树　根　右子树

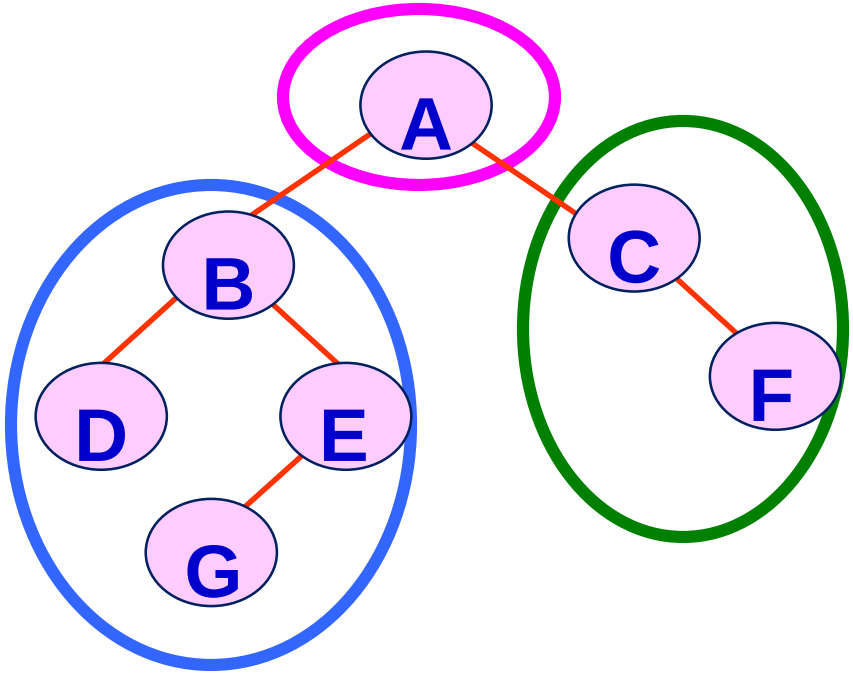
后序遍历（LRD）

若二叉树非空

（1）后序遍历左子树

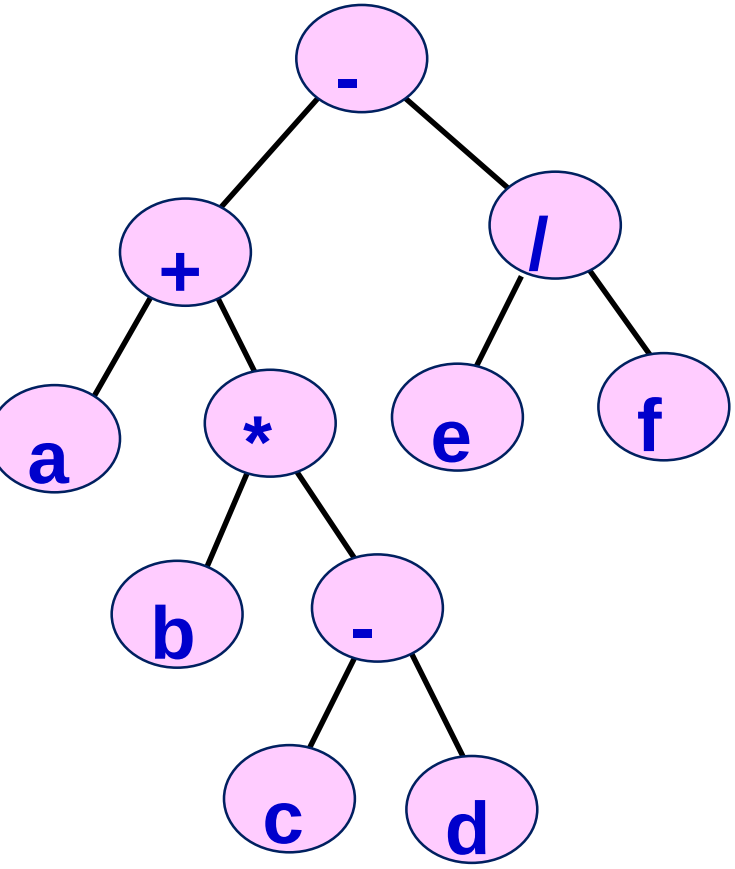
（2）后序遍历右子树

（3）访问根结点



后序遍历序列： D, G, E, B, F, C, A

左子树 右子树 根



例如左图所示的二叉树，

若**先序遍历**此二叉树，按访问结点的先后次序将结点排列起来，其先序序列为：

- + a * b – c d / e f

按**中序遍历**，其中序序列为：

a + b * c – d – e / f

按**后序遍历**，其后序序列为：

a b c d - * + e f / -

任何一个二叉树的叶子结点在先序、中序和后序遍历序列中的相对次序不发生改变。

判断与填空:

1. 在一个非空的二叉树的中序遍历序列中,根结点的右边只有右子树的部分结点.

2. 二叉树的先序遍历序列中, 任意一个结点均处在其子孙结点的前面.

3. 任何一个二叉树的叶子结点在先序.中序和后序遍历序列中的相对次序不发生改变.

4. 已知一棵二叉树的中序序列为BDCEAFHG, 后序序列为DECBHGFA, 则其先序遍历序列为_____.

5. 已知某二叉树的先序遍历序列是abdgcefh,中序遍历序列是dgbaechf, 则其后序遍历序列为_____.

由二叉树遍历的先序和中序序列或后序和中序序列可以唯一构造一棵二叉树.

例：已知一棵二叉树的中序序列为BDCEAFHG，后序序列为DECBHGFA，请画出这棵二叉树。

分析：

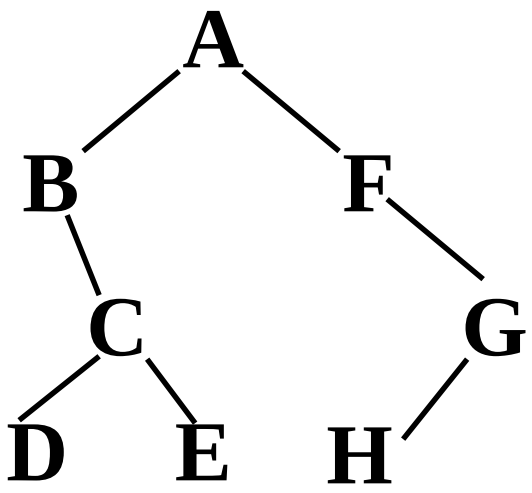
①由后序遍历特征，根结点必在后序序列尾部（即A）；

②由中序遍历特征，根结点必在其中间，而且其左部必全部是左子树的子孙（BDCE），其右部必全部是右子树的子孙（FHG）；

③继而，根据后序中的DECB子树可确定B为A的左孩子，根据HGF子串可确定F为A的右孩子；以此类推。

已知中序遍历：B D C E A F H G

后序遍历：D E C B H G F A



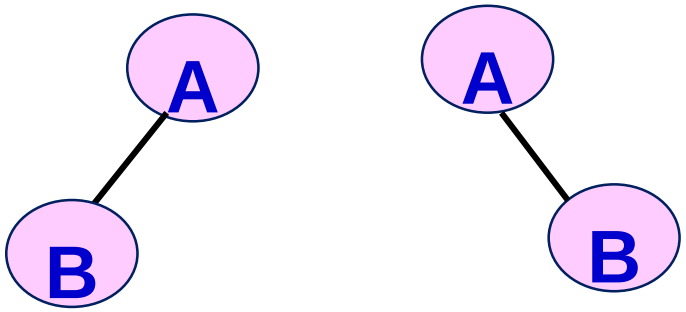
（D C E）　（F H G）

由先序和后序序列不能唯一确定一棵二叉树。因为无法确定左右子树两部分。

如某二叉树：

先序序列：a b

后序序列：b a



两棵不同的二叉树

2. 二叉树遍历的递归算法

上面介绍了三种遍历方法，显然是用递归的方式给出的三种遍历方法，以先序为例：

先序遍历（DLR）的定义：

若二叉树非空

（1）访问根结点；

（2）先序遍历左子树

（3）先序遍历右子树；

先序遍历递归定义

递归项

该定义隐含着若二叉树为空，结束

这实际上是先序遍历的递归定义，我们知道递归定义包括两个部分：

1）基本项（也叫终止项）描述递归终止时问题的求解；

2）递归项　将问题分解为与原问题性质相同，但规模较小的问题；

上面先序遍历的定义等价于：

若二叉树为空，结束　　——基本项（也叫终止项）

若二叉树非空　　——递归项

（1）访问根结点；

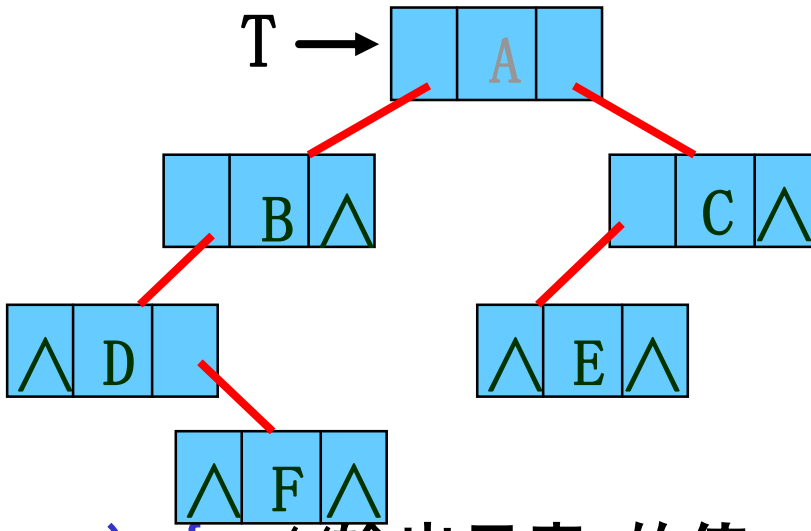
（2）先序遍历左子树

（3）先序遍历右子树；

下面给出先序、中序、后序遍历递归算法，为了增加算法的可读性，这里对书上算法作了简化，没有考虑访问结点出错的情况（即我们假设调用函数visit( )访问结点总是成功的）。

1、先序遍历递归算法

```
void PreOrder (BiTree T, Status(*Visit)(TElemType e)) {
//采用二叉链表存贮二叉树，visit( )是访问结点的函数。本算法先序
 //遍历以T为根结点指针的二叉树，对每个数据元素调用函数Visit( )
    if (T) {//若二叉树为空，结束返回
      // 若二叉树不为空，访问根结点；遍历左子树，遍历右子树
      Visit(T->data);
      PreOrder(T->lchild, Visit);
      PreOrder(T->rchild, Visit);
   }
}
```



最简单的Visit函数是：

```
    Status PrintElement(TElemType  e) {  //输出元素e的值
        printf(e);
        return OK;
   }
```

```
void PreOrder (BiTree T)
{ //先序遍历二叉树的递归算法
    if (T)   //若二叉树非空
   {
      printf(T->data);           //访问根结点
      PreOrder(T->lchild);       //先序遍历左子树
      PreOrder(T->rchild);       //先序遍历右子树
   }
}
```

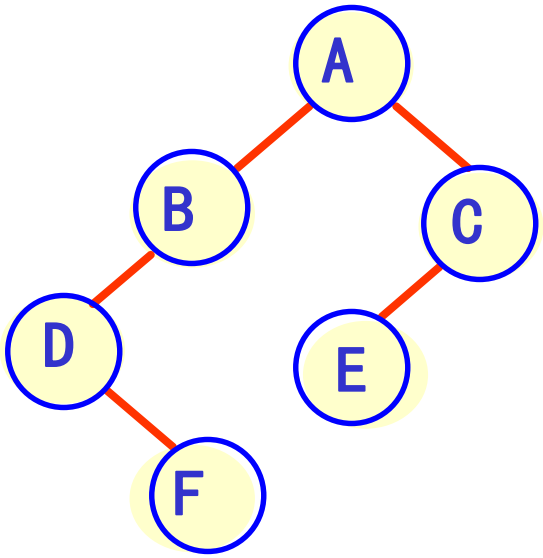
```
void InOrder (BiTree T)
{ //中序遍历二叉树的递归算法
    if (T)  //若二叉树非空
   {
      InOrder(T->lchild);     //中序遍历左子树
      printf(T->data);        //访问根结点
      InOrder(T->rchild);     //中序遍历右子树
   }
}
```

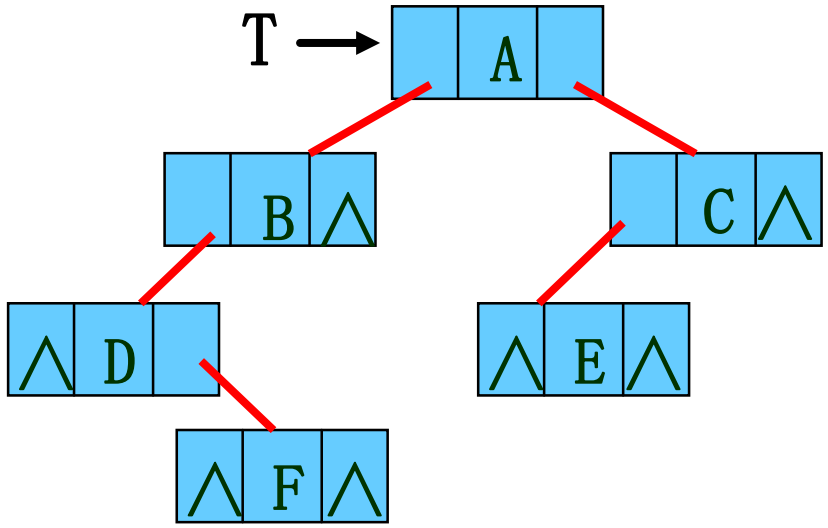
```
void PostOrder (BiTree T)
{ //后序遍历二叉树的递归算法
    if (T) //若二叉树非空
   {
      PostOrder(T->lchild); //后序遍历左子树
      PostOrder(T->rchild); //后序遍历右子树
      printf(T->data);      //访问根结点

   }
}
```

3. 二叉树遍历算法的应用

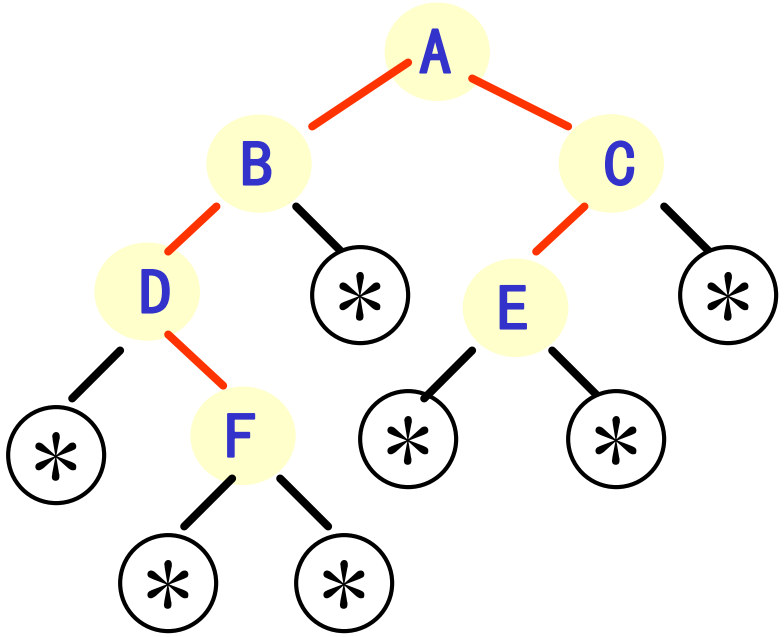
“遍历”是二叉树各种操作的基础，假设访问结点的具体操作不仅仅局限于输出结点数据域的值，而把“访问”延伸到对结点的判别、计数等其他操作，可以解决一些关于二叉树的其他实际问题。如果在遍历过程中生成结点，便可建立二叉树的存储结构。

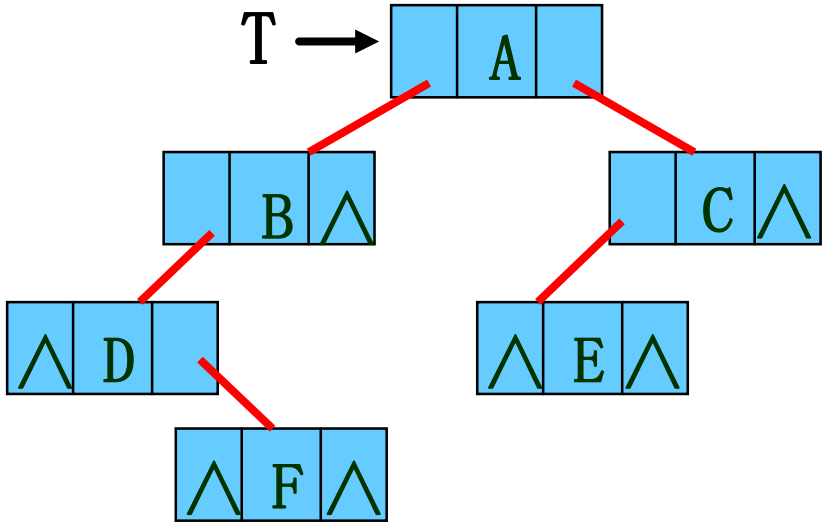
例1　创建二叉树的存储结构---二叉链表





先序序列：A B D F C E





（在空子树处添加*的二叉树的）先序序列：

A B D * F * * * C E * * *

基本思想：输入（在空子树处添加*的）二叉树的先序序列（设每个元素是一个字符），按先序遍历的顺序，建立二叉链表的所有结点并完成相应结点的链接。

```
void CreateBiTree(BiTree &T)
{ //输入二叉树的先序序列（设每个元素是一个字符），建立二叉 链表
    scanf (&ch);
    if (ch== '*' )    T=NULL;   //递归结束，建空树
    else     //  递归创建二叉树
    {
      T=(BiTNode * )malloc(sizeof(BiTNode));
      T->date = ch; //生成根结点，数据域置为ch
      CreateBiTree(T->lchild);   //递归创建左子树
      CreateBiTree(T->rchild);   //递归创建右子树

    }
}
```

例2. 求二叉树的叶子结点。

基本思想：

遍历操作访问二叉树的每个结点，而且每个结点仅被访问一次。所以可在二叉树遍历的过程中，统计叶子结点的个数。

```
void leaf(BiTree T)
{//在先序遍历二叉树的过程中，统计叶子结点的个数，n为全局变量，用于累加二叉树的叶子结点数
  if(T)
   {
      if(T->lchild==NULL && T->rchild==NULL)
                        n=n+1;  //如果T是叶子结点，n加1
      leaf(T->lchild);   //递归计算左子树中叶子结点数
      leaf(T->rchild);  //递归计算右子树中叶子结点数
    }
}
```

```
void PreOrder (BiTree T) {
    //采用二叉链表存贮二叉树， 本算法先序遍历以T为根结点指针的二叉树
   if (T) {
         visit(T->data);
         PreOrder(T->lchild);
         PreOrder(T->rchild);
    }
}
```

访问结点时
调用visit( )

法和计算叶子结点算法，有什么相同和不同？

```
void leaf(BiTree T) {
//采用二叉链表存贮二叉树，n为全局变量，用于累加二叉树的叶子结点,的个数
//本算法在先序遍历二叉树的过程中，统计叶子结点的
  if(T) {
      if (T->lchild==NULL&&T->rchild==NULL
       leaf(T->lchild);
       leaf(T->rchild);
  }
}
```

访问结点时
统计叶子结点的个数

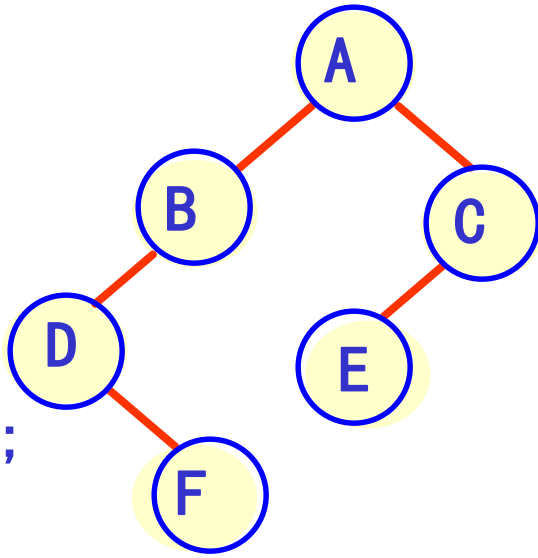
结构类似

求叶子结点第二种方法

[算法思想]采用递归算法……

$$Leaf(T)=\begin{cases}0 & \text{若T=NULL}\\ 1 & \text{T->Lchild==NULL 且T->Rchild==NULL}\\ Leaf(T->Lchild)+Leaf(T->Rchild) & \text{其它}\end{cases}$$

```
int  Leaf(BiTree T)//求叶子结点第二种方法
{
 if (T==NULL)     return 0;
 if(T->lchild==NULL && T->rchild==NULL)
     return 1;
 return  Leaf(T->lchild)+Leaf(T->rchild);
}
```



例3.　求二叉树的深度

$$Depth(T)=\begin{cases}0 & \text{当T=NULL} \\ \max(Depth(T->rchild),\ Depth(T->rchild))+1 & \text{其它}\end{cases}$$

```
int Depth (BiTree T )  // 计算二叉树T的深度
{
    if ( T==NULL )    depthval = 0;   //如果是空树,深度为0,递归结束
    else
    {
       dL= Depth( T->lchild );     //计算左子树的深度
       dR= Depth( T->rchild );   //计算右子树的深度
       depthval = 1 +(dL>dR?dL:dR);  //二叉树的深度为较大者加1
     }
     return depthval;
}
```

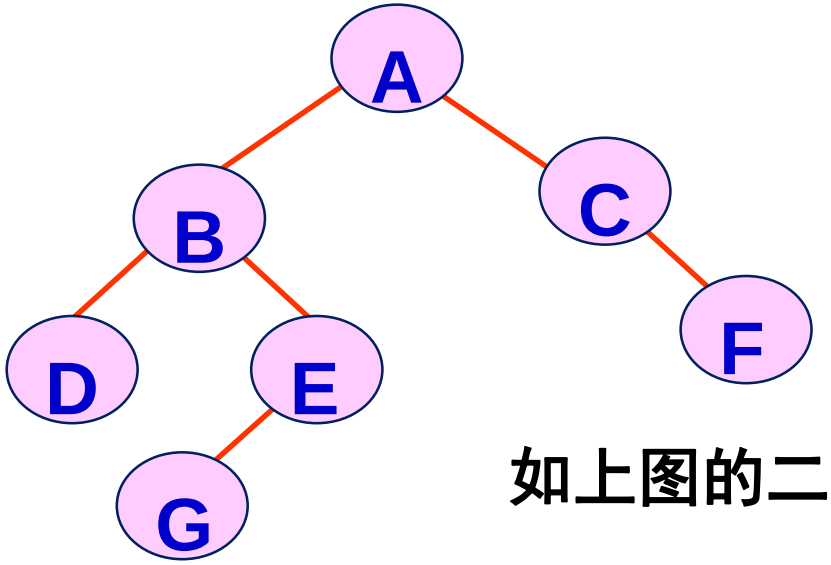
例4. 若已知两棵二叉树B1和B2皆为空，或者皆不空且B1的左、右子树和B2的左、右子树分别相似，则称二叉树B1和B2相似。试编写算法，判别给定两棵二叉树是否相似.

Tlike(B1,B2)=
- True　　若B1=NULL，且B2=NULL
- False　　B1=NULL，B2!=NULL 或 B1！=NULL，B2==NULL
- Tlike(B1->Lchild,B2->Lchild) &&Tlike(B1->Rchild,B2->Rchild)　　其它

```
int tlike(BiTree B1,BiTree B2)
{ if (B1==NULL && B2==NULL)  return 1;
  if ((B1==NULL && B2!=NULL)||(B1!=NULL && B2==NULL))
       return 0;
  if tlike(B1->lchild,B2->lchild)
       return tlike(B1->rchild,B2->rchild)
  else
     return 0;
}
```

二叉树的层次遍历

按照“从上到下、从左到右”的顺序遍历二叉树。



如上图的二叉树，层次遍历序列为:

ABCDEFG

层次遍历:

1、根结点进队列;

2、结点出队列，被访问;

3、结点的左、右孩子（非空）进队列;

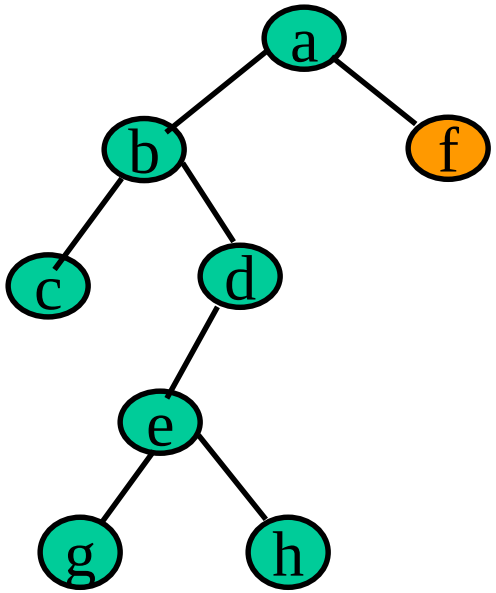
4、反复执行 2、3，至队列空为止。

```
void LevelOrderTraverse(BiTree T) // 层次遍历T(利用队列)
 {
    if(T)
    {    InitQueue(q);    // 初始化队列q
         EnQueue(q, T);   // 根指针入队
        while(!QueueEmpty(q)) // 队列不空
       {
         DeQueue(q, a);   // 出队元素(指针)，赋给a
         printf(a->data);     // 访问a所指结点
         if(a->lchild!=NULL)    // a有左孩子
              EnQueue(q, a->lchild);     // 入队a的左孩子
         if(a->rchild!=NULL)      // a有右孩子
              EnQueue(q, a->rchild);     // 入队a的右孩子
    }
  }
 }
```

遍历算法的非递归描述：

中序遍历：

采用一个栈保存返回的结点，先扫描根结点的所有左结点并入栈，出栈一个结点，访问之，然后扫描该结点的右结点并入栈，再扫描该右结点的所有左结点并入栈，如此这样，直到栈空为止。



先序的程序实现：

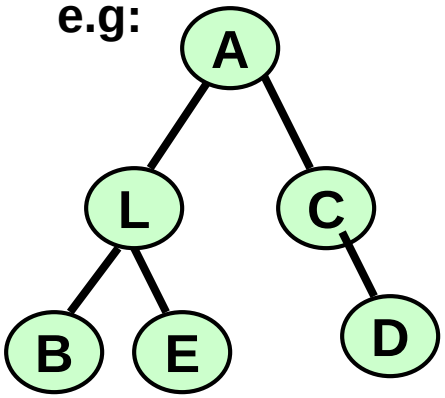
1、根结点进栈

2、结点出栈，被访问

3、结点的右、左儿子（非空）进栈

4、反复执行 2、3 ， 至栈空为止。

e.g:



先序：A、L、B、E、C、D

6.4　线索二叉树

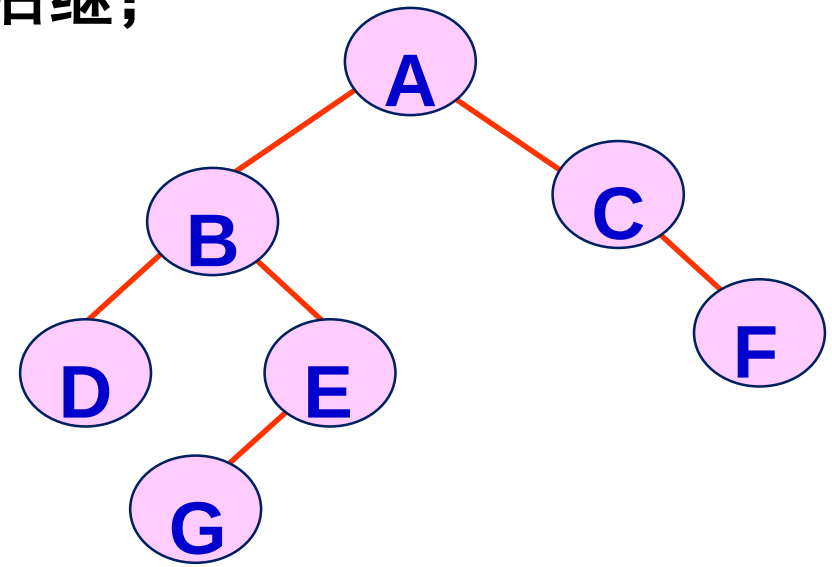
线索二叉树的概念

与线性表相比，对二叉树的遍历存在如下问题：

1）遍历算法要复杂、费时；

2）为查找二叉树中某结点在某种遍历下的后继，必须从根开始遍历，直到找到该结点及后继；

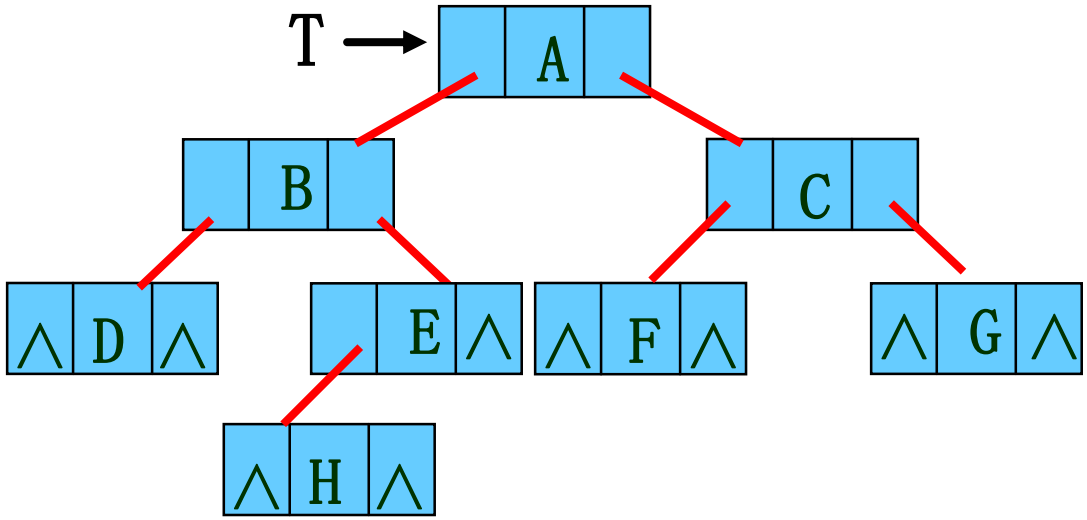
为提高二叉树遍历的效率，可将遍历序列中每个结点前趋、后继信息保存起来。



结点加上前趋、后继信息（线索）的二叉树称为线索二叉树。

线索二叉树的存储方法：

n个结点的二叉链表，有n+1个空指针域，故可利用这些的空指针域存放结点的前趋和后继指针。



规 定：

- 若结点有左子树，则lchild指向其左孩子；

  否则， lchild指向其直接前驱(即线索)；

- 若结点有右子树，则rchild指向其右孩子；

  否则，rchild指向其直接后继(即线索) 。

为区别两种不同情况，增加两个标志域：

lchild	ltag	data	rtag	rchild

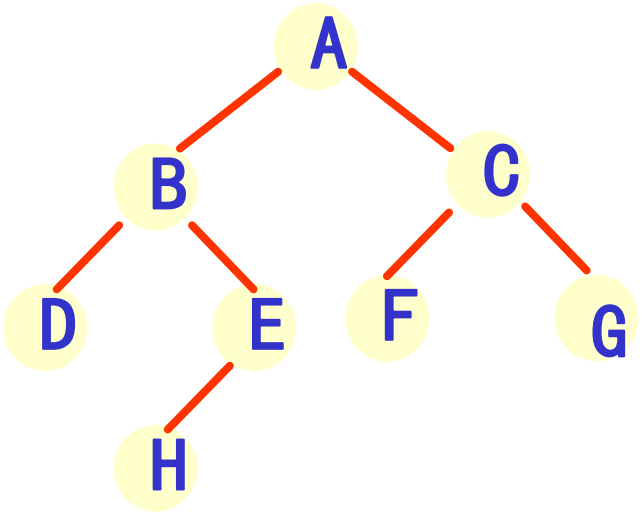
ltag= { 0　lchild 域指示结点的左孩子
　　 { 1　lchild 域指示结点的前驱

rtag= { 0　rchild 域指示结点的右孩子
　　 { 1　rchild 域指示结点的后驱

以这种结构构成的二叉链表作为二叉树的存储结构，叫做线索链表。

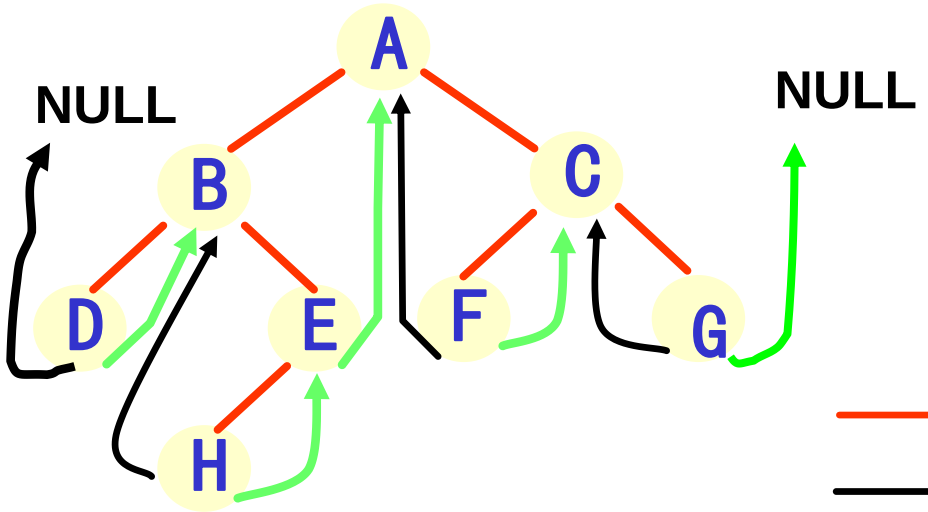
构造线索二叉树---线索化

线索化过程就是在遍历过程中修改空指针的过程。



中序遍历序列结果为：D, B, H, E, A, F, C, G

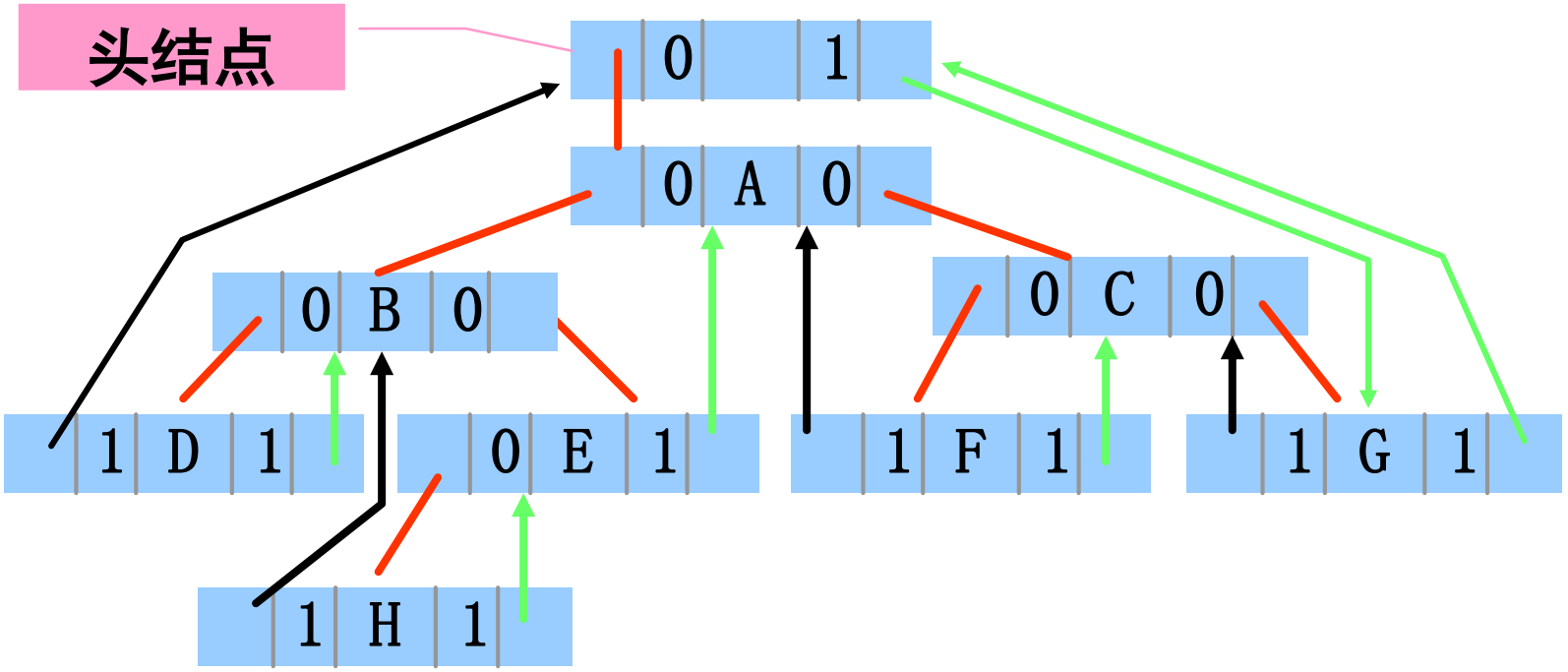
中序线索二叉树



孩子指针

前趋指针

后继指针



中序线索二叉树存储结构

为线索链表加上一头结点，

头结点的lchild域：存放线索链表的根结点指针；

头结点的rchild域：中序序列最后一个结点的指针；

中序序列第一个结点lchild域指向头结点；

中序序列最后一个结点的rchild域指向头结点；

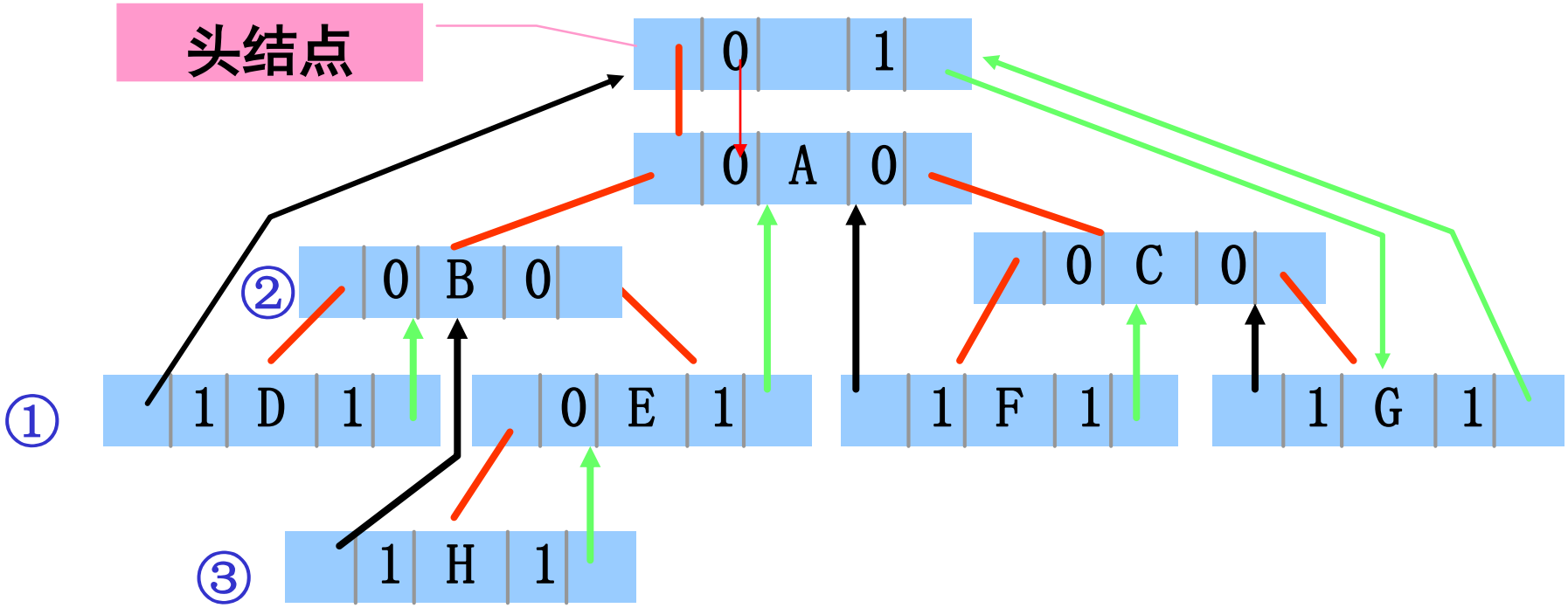
遍历线索二叉树

二叉树的线索化过程是基于对二叉树进行遍历，而线索二叉树上的线索又为相应的遍历提供了方便。

中序线索二叉树的遍历算法：

※ 中序遍历的第一个结点 ？

※ 在中序线索二叉树中结点的后继 ？



中序遍历序列：D, B, H, E, A, F, C, G

如图标出的中序二叉树结点的顺序，可看出

1）中序序列的第一结点，是二叉树的最左下结点；

2）若p所指结点的右孩子域为线索，则p的右孩子结点即为后继结点

3）若p所指结点的右孩子域为孩子指针，则p的后继结点为其右子树最左下结点；

下面是线索链表的遍历算法。

基本步骤：

1） p=T->lchild; p指向线索链表的根结点；

2）若线索链表非空，循环：

（a）循环，顺着p左孩子指针找到最左下结点；访问之；

（b）若p所指结点的右孩子域为线索， p的右孩子结点即为后继结点循环： p=p->rchild； 并访问p所指结点；（在此循环中，顺着后继线索访问二叉树中的结点）

（c） 一旦线索“中断”，p所指结点的右孩子域为右孩子指针，p=p->rchild，使 p指向右孩子结点；

3）返回OK；结束

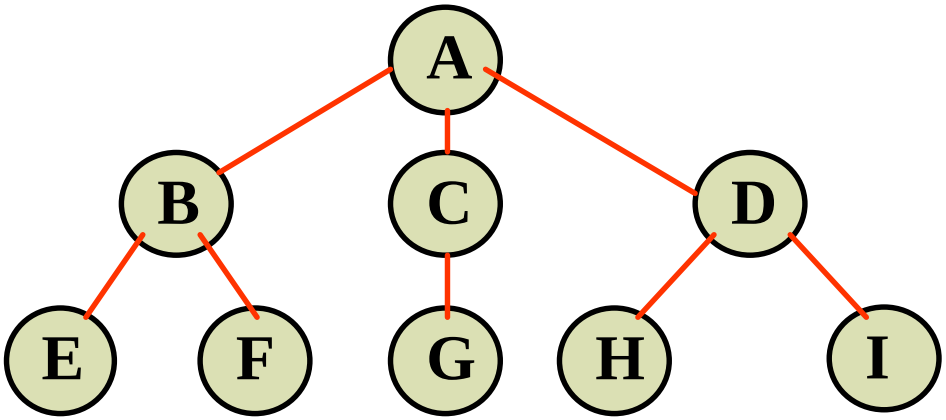
```
Status InOrderTraverse_Thr(BiThrTree T, Status (*Visit)(TElemType e)) {
// T指向头结点，头结点的左链lchild指向根结点，头结点的右链lchild
// 指向中序遍历的最后一个结点。中序遍历二叉线索链表表示的二叉树，
// 对每个数据元素调用函数Visit。
  p = T->lchild;                                  // p指向根结点
  while (p != T) {                                // 空树或遍历结束时，p==T
    while (p->LTag==Link) p = p->lchild; //找到最左下结点；访问之
    if (!Visit(p->data)) return ERROR;            // 访问其左子树为空的结点
   while (p->RTag==Thread && p->rchild!=T) {// 若p所指结点的右孩子域为
                                                  //线索 且不是最后一个结点
      p = p->rchild; Visit(p->data);              // 访问后继结点
    }
     p = p->rchild;                               // p进至其右子树根
  }
  return OK;
} // InOrderTraverse_Thr
```

6.5 树和森林

树的存贮结构

1、双亲表示法

用一组连续的存储单元存储树的结点，每个结点包含两个域：一个数据域，一个“双亲位置域”，用于指示其双亲结点的位置.

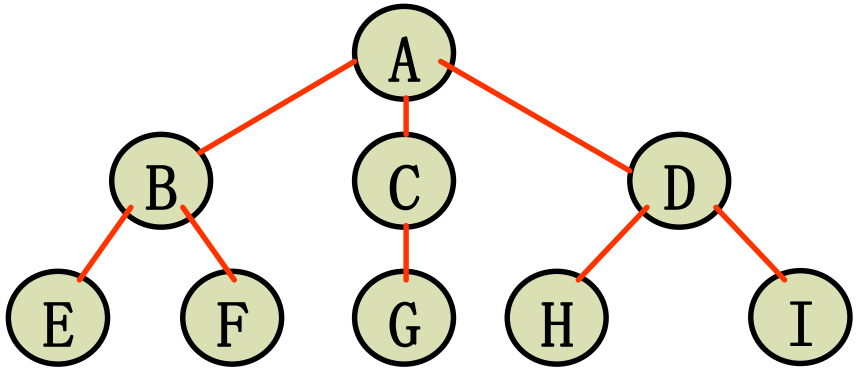


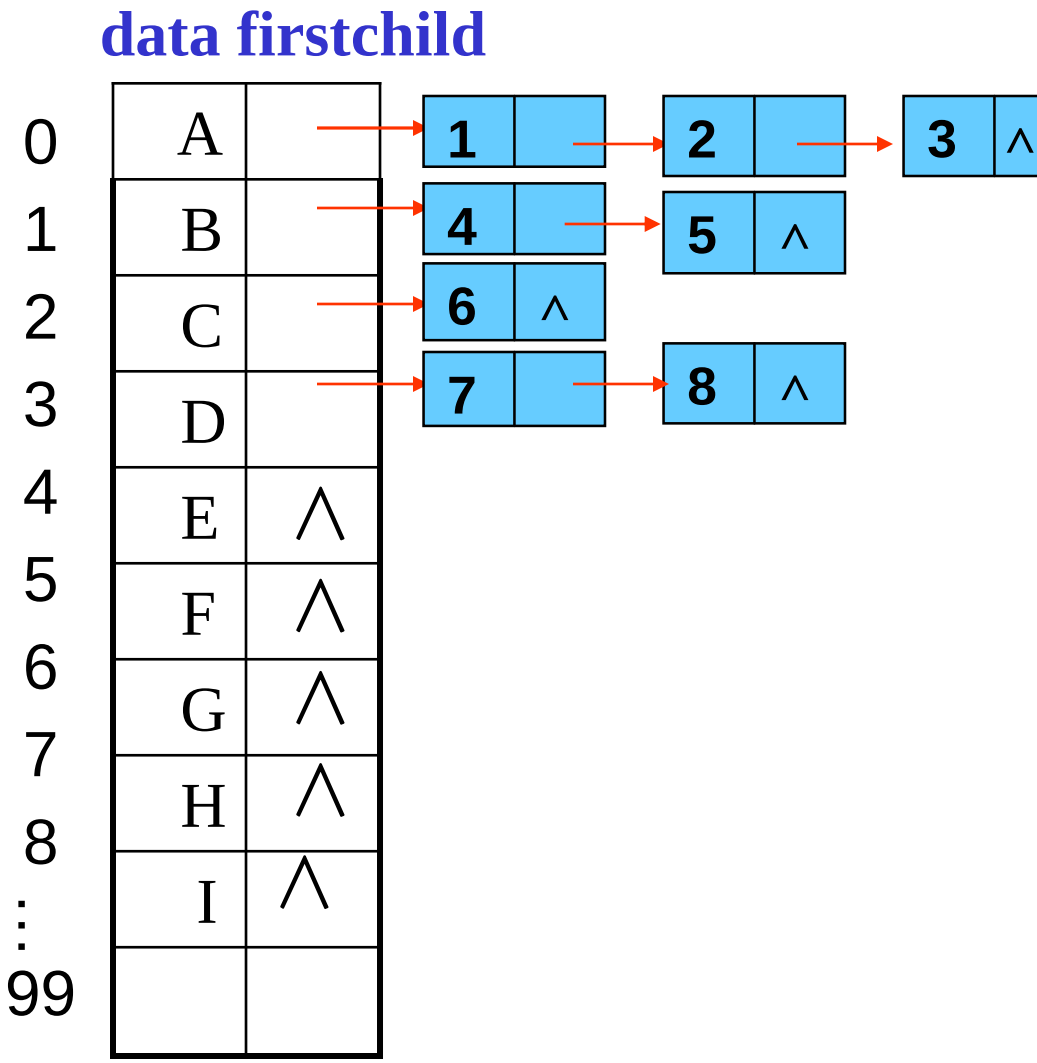
	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	2
7	H	3
8	I	3
.		

树的双亲表示法示例

2、孩子链表表示法：

把每个结点的所有孩子结点组织成一个链表，则有n个结点的树就有n个孩子链表， n个孩子链表的表头结点（双亲结点）又构成一个线性表。


A
B
C
D
E
F
G
H
I


data firstchild
0
1
2
3
4
5
6
7
8
99
A
B
C
D
E
F
G
H
I
1
2
3
4
5
6
7
8
^

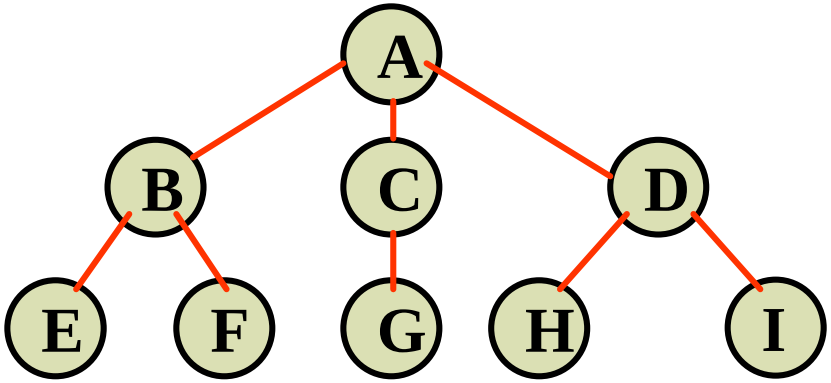
树的孩子链表图示

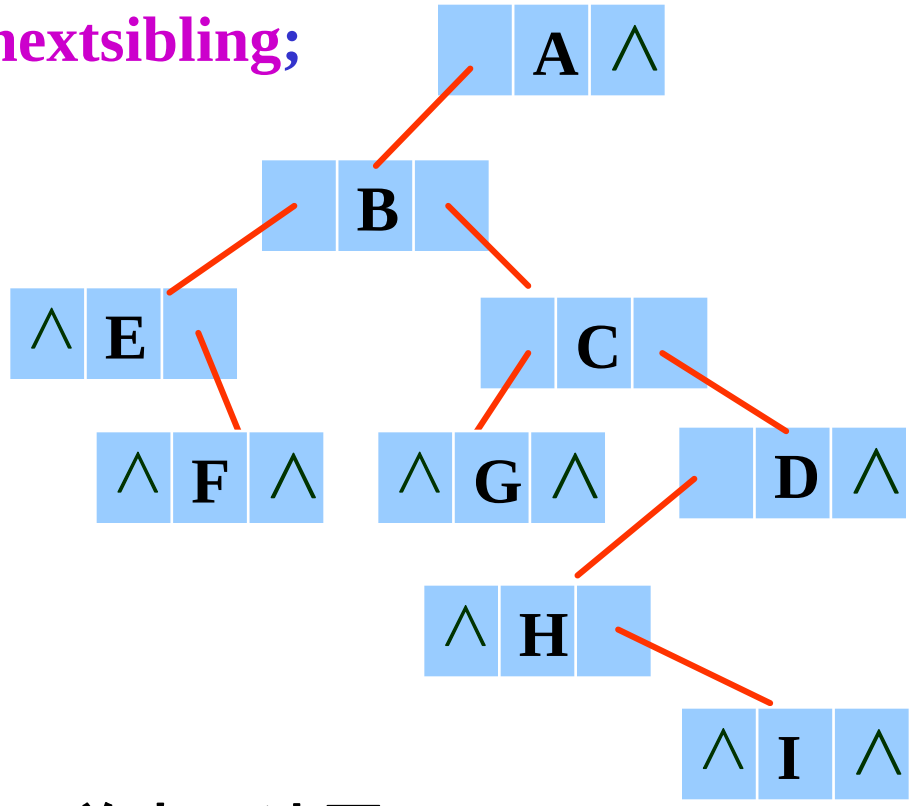
3、孩子-兄弟表示法

孩子兄弟表示法用二叉链表作为树的存贮结构。

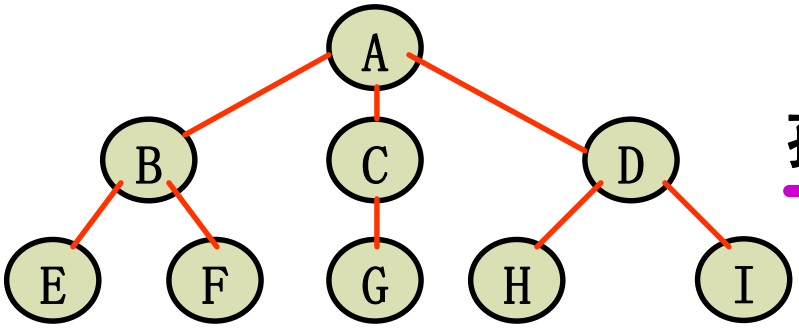
孩子兄弟表示法类型定义：

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



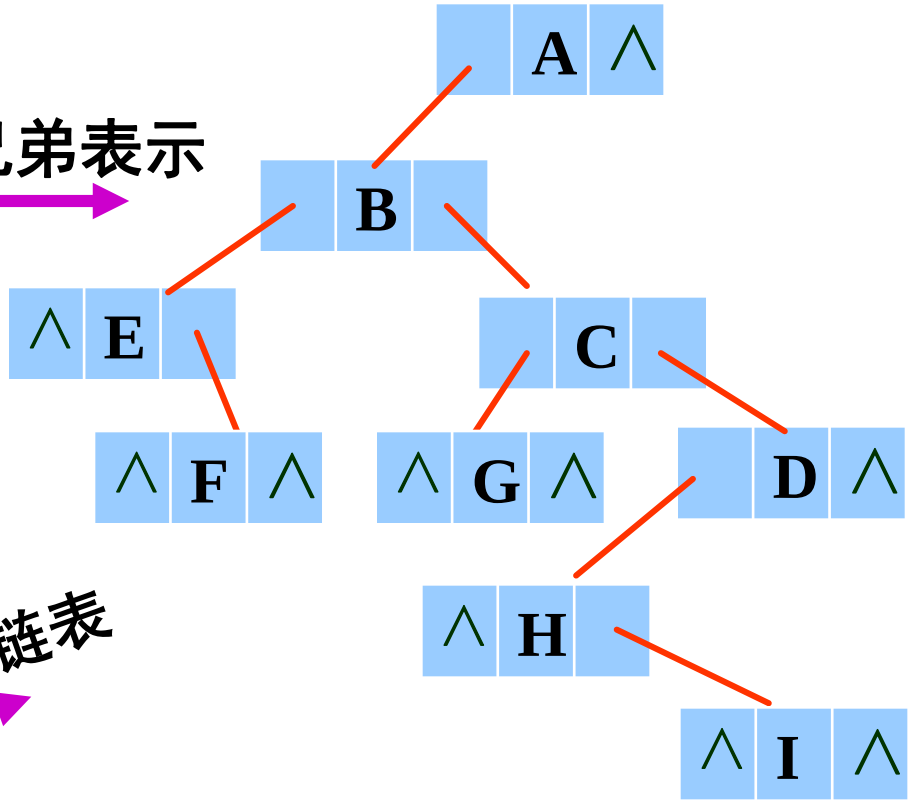


树的孩子兄弟表示法图示

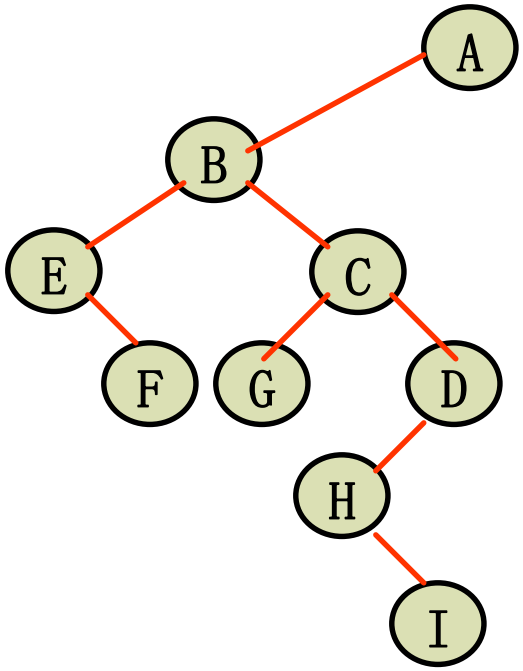


(a)

孩子兄弟表示



对应的二叉链表



(b)

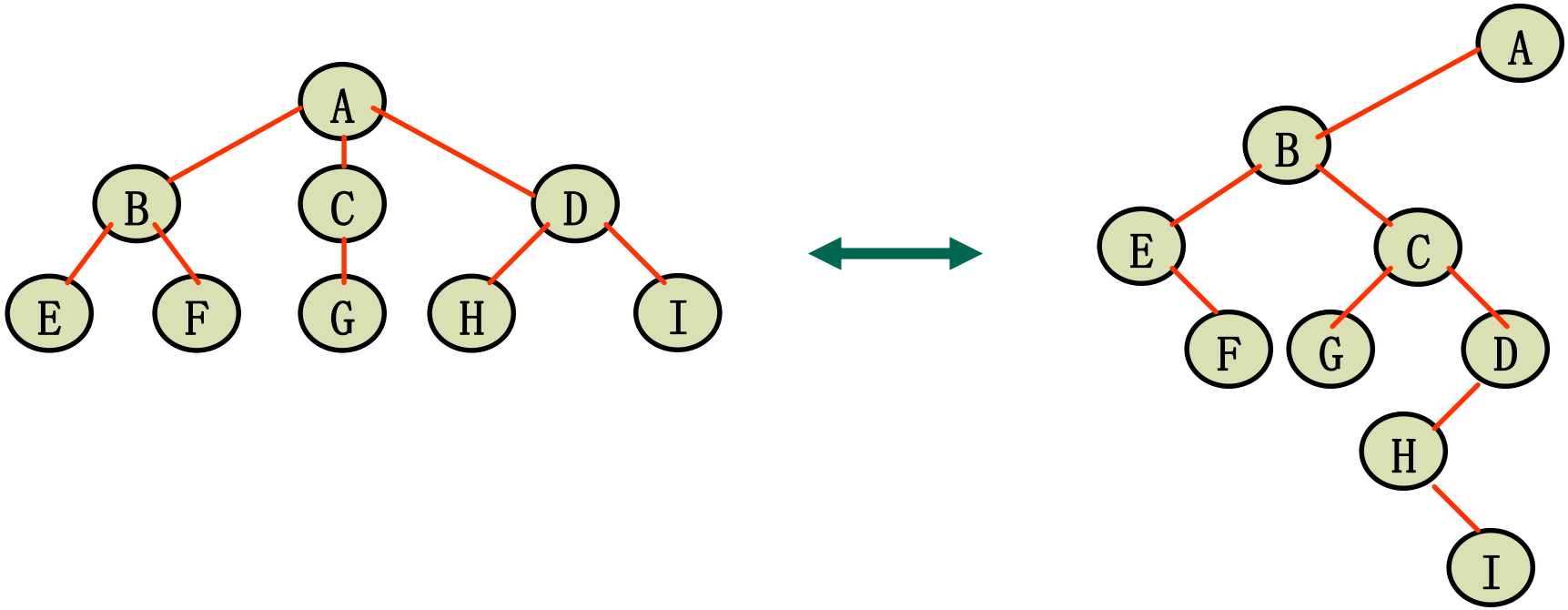
此二叉链表既是树(a)的孩子兄弟表示，又是二叉树(b)的二叉链表，由此可将树与二叉树对应起来。

树与二叉树的转换

二叉树与树都可用二叉链表存贮，以二叉链表作中介，可导出树与二叉树之间的转换。

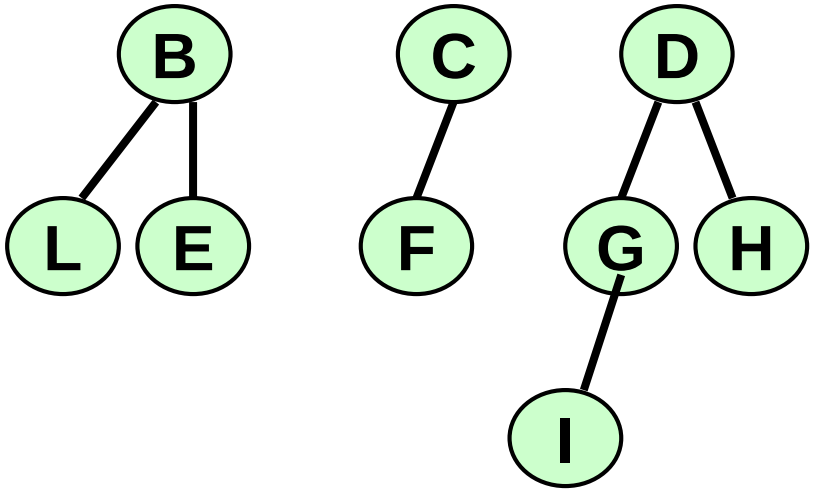
树与二叉树转换方法：

树		二叉树
根	⟷	根
结点X的第一个孩子	⟷	结点X的左孩子
结点X紧邻的右兄弟	⟷	结点X的右孩子

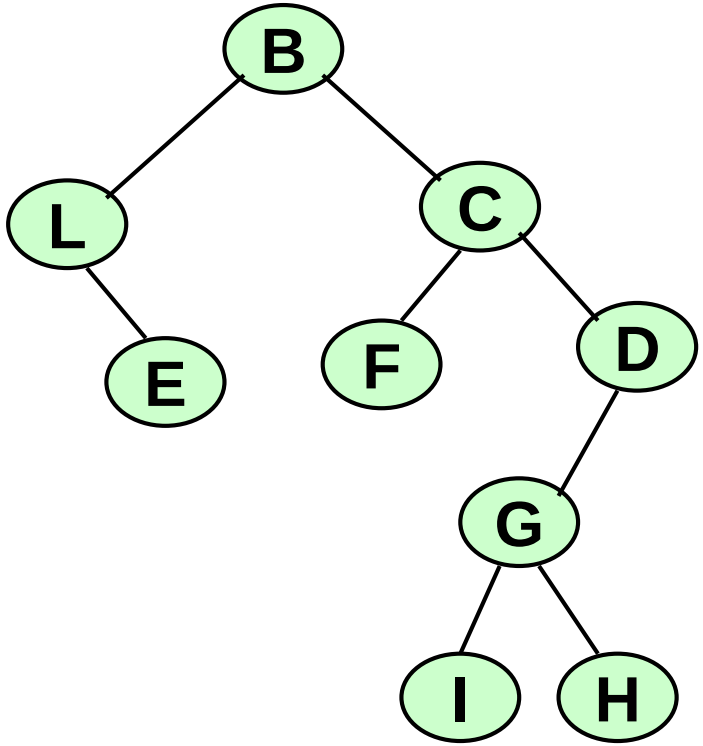


森林：树的集合。

将森林中**树的根看成兄弟**，可用树孩子兄弟表示法存储森林；用树与二叉树的转换方法，进行**森林与二叉树转换**。

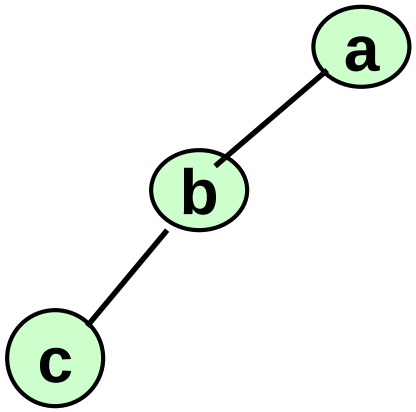


包含三棵树的森林

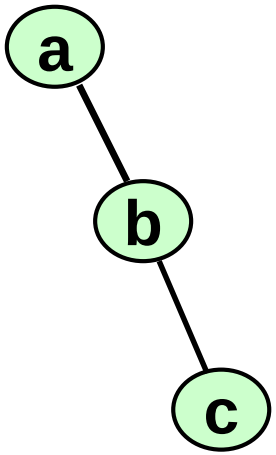


相应的二叉树

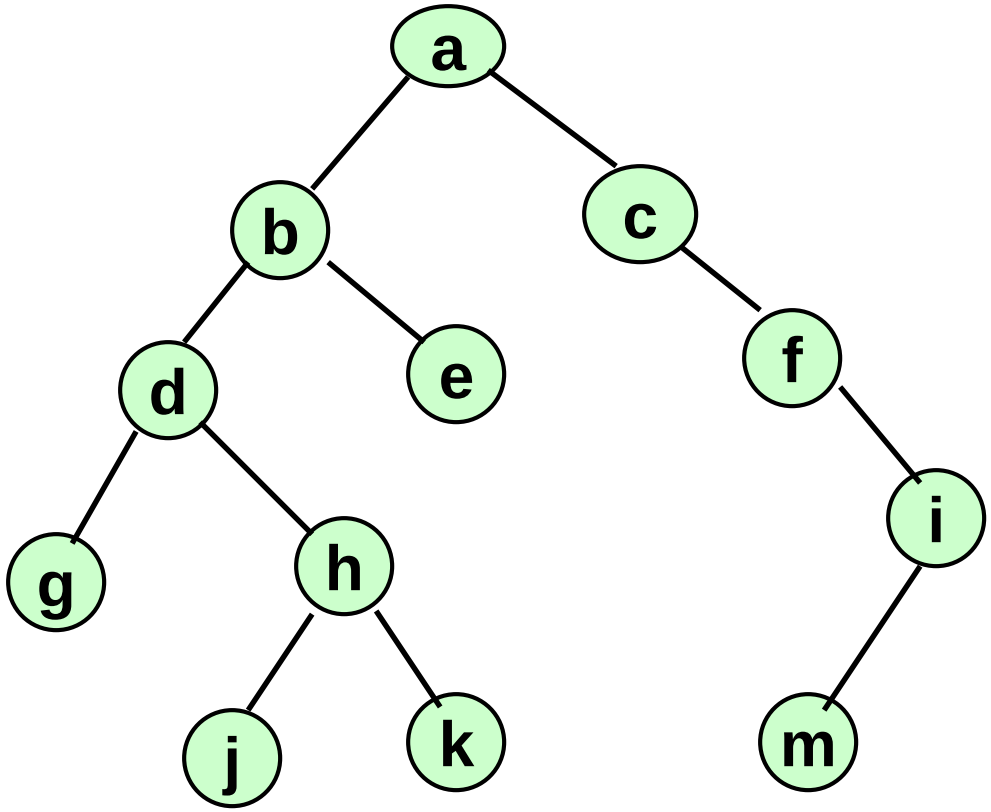
1. 试画出和下列二叉树相应的森林：



(a)



(b)



树和森林的遍历

树的遍历可有三条搜索路径:

先根(次序)遍历:

若树不空，则先访问根结点，然后依次**先根遍历**各棵子树。

例 先根遍历序列 A B E F C G D H I

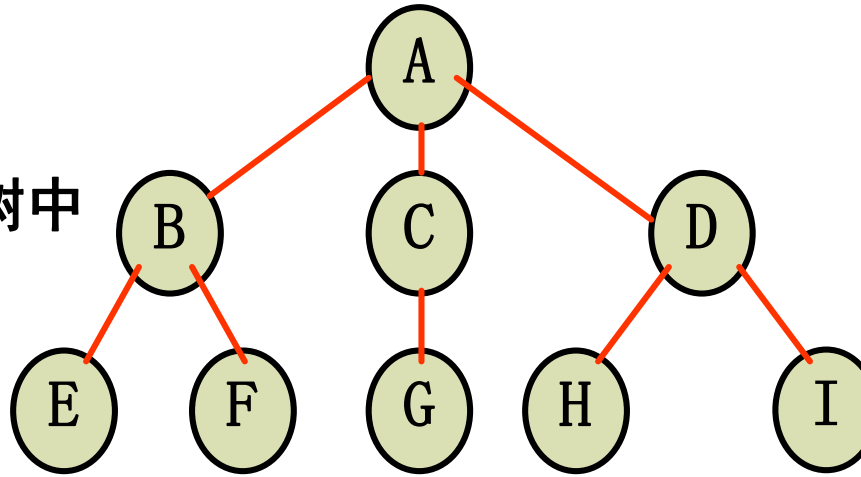
后根(次序)遍历:

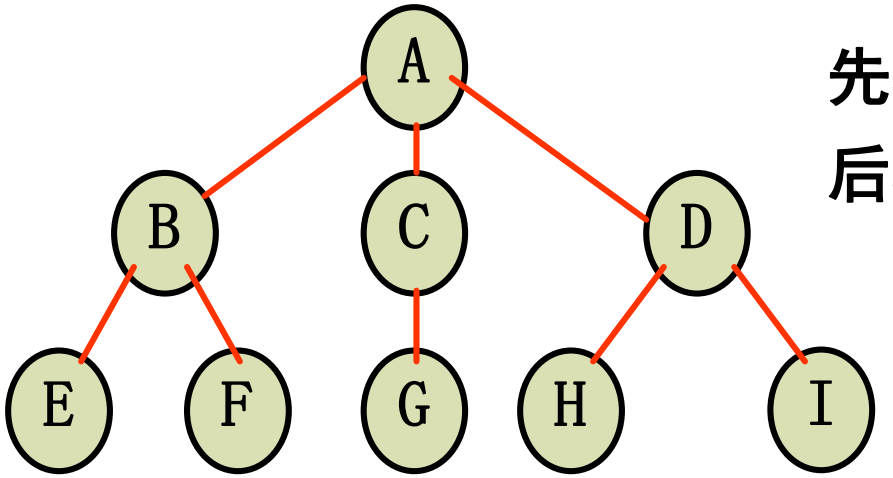
若树不空，则先依次**后根遍历**各棵子树，然后访问根结点

后根遍历序列 E F B G C H I D A

按层次遍历:

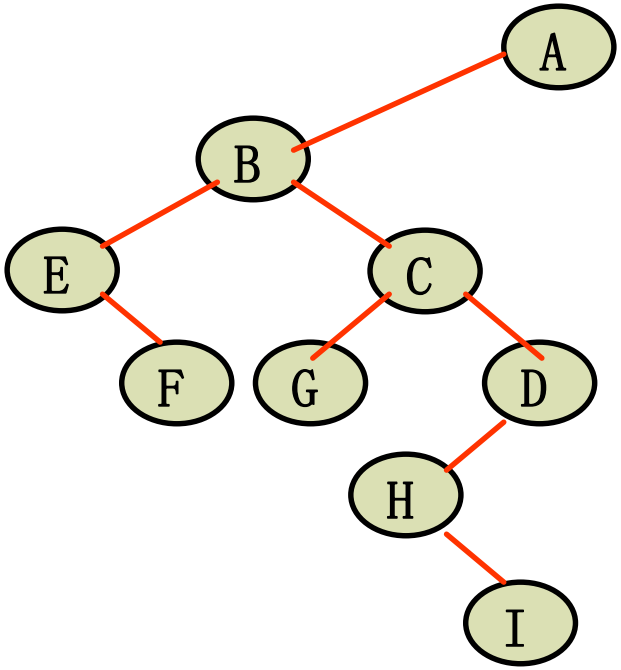
若树不空，则自上而下自左至右访问树中每个结点。





先根遍历序列 A B E F C G D H I

后根遍历序列 E F B G C H I D A



树的遍历和二叉树遍历的对应关系 ？

树的先根遍历 对应 二叉树的 ？遍历 **先序遍历**

树的后根遍历 对应 二叉树的 ？遍历 **中序遍历**

森林的遍历

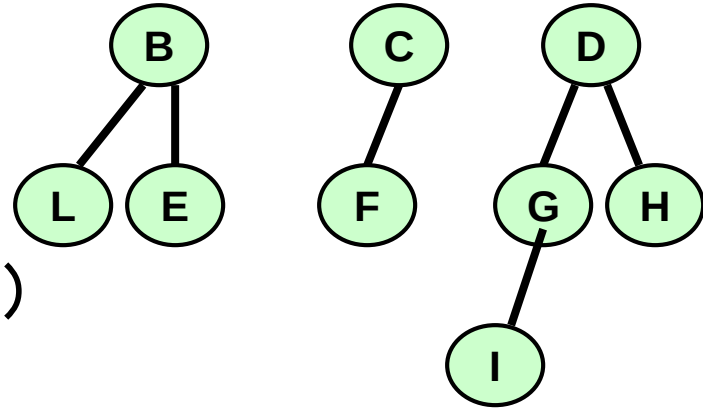
先序遍历（对森林中的每一棵树进行先根遍历）

若森林不空，则

访问森林中第一棵树的根结点；

先序遍历森林中第一棵树的子树森林；

先序遍历森林中（除第一棵树之外）其余树构成的森林。



中序遍历（对森林中的每一棵树进行后根遍历）

若森林不空，则

中序遍历森林中第一棵树的子树森林；

访问森林中第一棵树的根结点；

中序遍历森林中（除第一棵树之外）其余树构成的森林。

6.6 赫夫曼树及应用

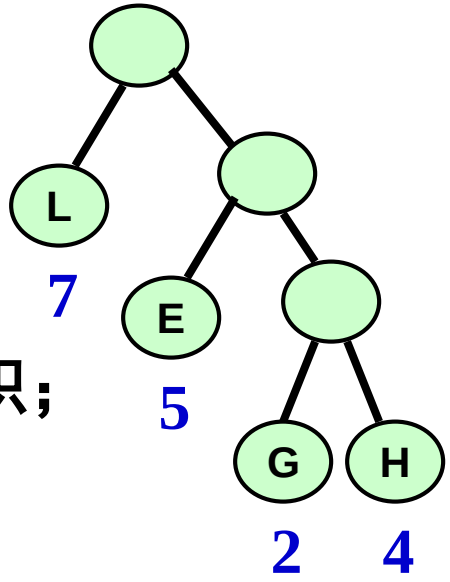
1. 最优二叉树(赫夫曼树)

结点的路径长度定义为：

从根结点到该结点的路径上分支的数目。

结点的带权路径长度：

从根到该结点的路径长度与该结点权的乘积；

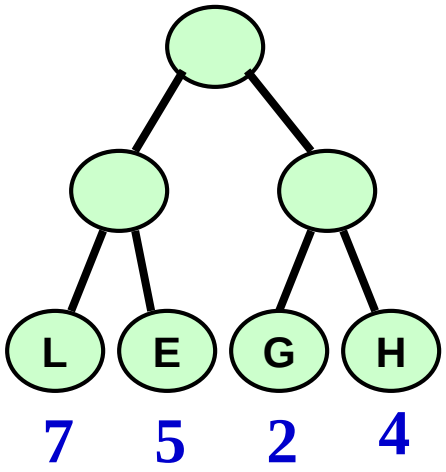


树的带权路径长度定义为：

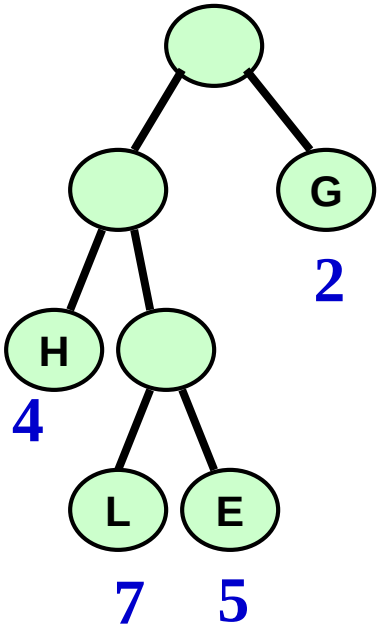
树中所有**叶子结点**的带权路径长度之和

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

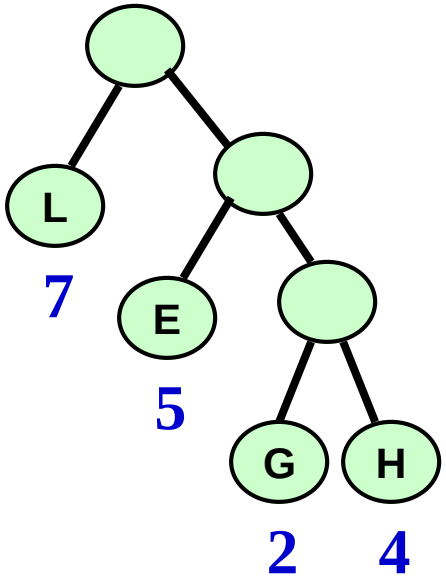
假设有n个权值$\{w_1, w_2, \ldots, w_n\}$，试构造一棵有n个叶子结点的二叉树，每个叶子结点的带权为w_n，则其中带权路径长度WPL最小的二叉树称做“**最优二叉树或赫夫曼树**”。



WPL=36



WPL=46



WPL=35

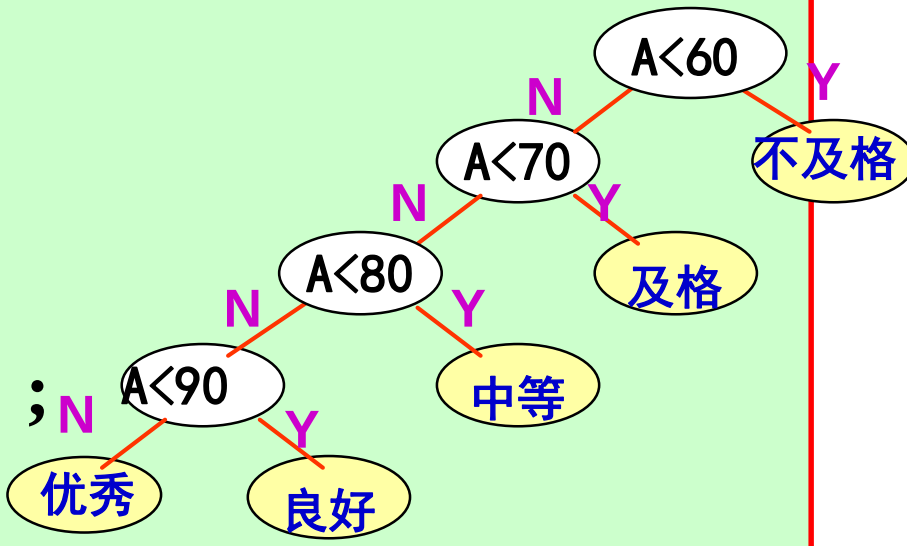
*应用举例

在求解某些判定问题时，利用哈夫曼树获得最佳判定算法。

例　　编制一个将百分制转换成五分制的程序。

最直观的方法是利用if语句来实现。可用二叉树描述判定过程。

```
if (a<60)     b="不及格";
else if (a<70)     b="及格";
   else if(a<80)     b="中等";
      else if(a<90)     b="良好";
         else b="优秀";
```



```
if (a<60)  b="不及格";                1
else if (a<70)  b="及格";             2
   else if(a<80)  b="中等";           3
      else if(a<90)  b="良好";        4
         else b="优秀";
```

如果该程序经常要使用或数据量很大。比如对北京市几十万小学生的分数进行转换，在这种情况下，要考虑转换程序的效率。

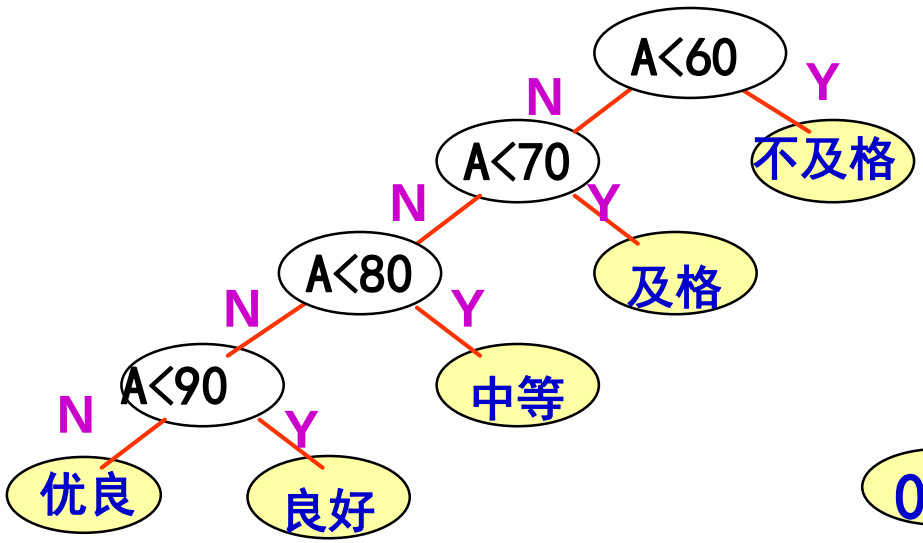
设有10000个百分制分数要转换，设学生成绩在5个等级上的分布如下：

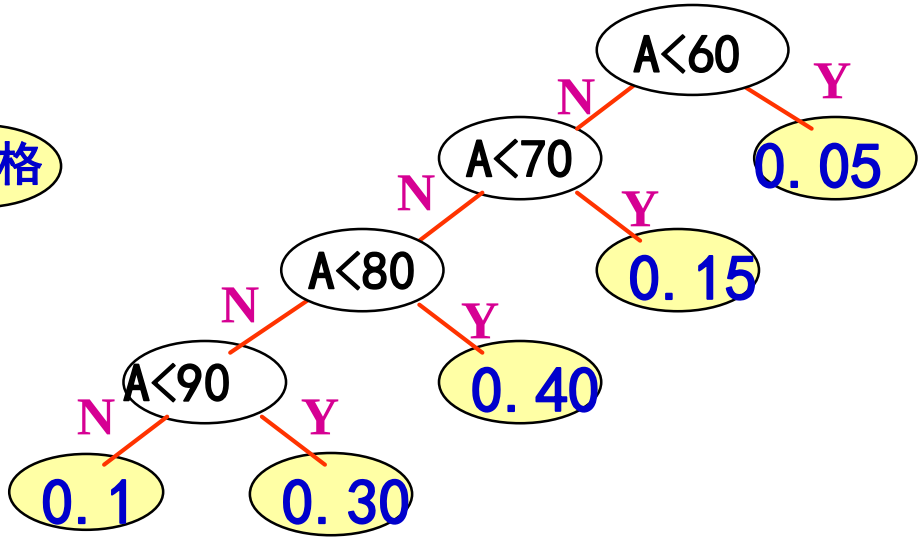
分数	0-59	60-69	70-79	80-89	90-100
比例数	0.05	0.15	0.40	0.30	0.10

转换10000个分数所需的总比较次数=

10000×（0.05 × 1+0.15 × 2+0.4 × 3+0.3 × 4+0.1 × 4）

若将学生成绩在5个等级上的分布比例看作描述判定过程二叉树叶子结点权值，（0.05 × 1+0.15 × 2 +0.4 × 3+0.3 × 4+0.1 × 4）正是该二叉树的带权路径长度。可见要想获得效率较高的转换程序，可构造以分数的分布比例为权值的哈夫曼树。





2、如何构造赫夫曼树

赫夫曼算法:

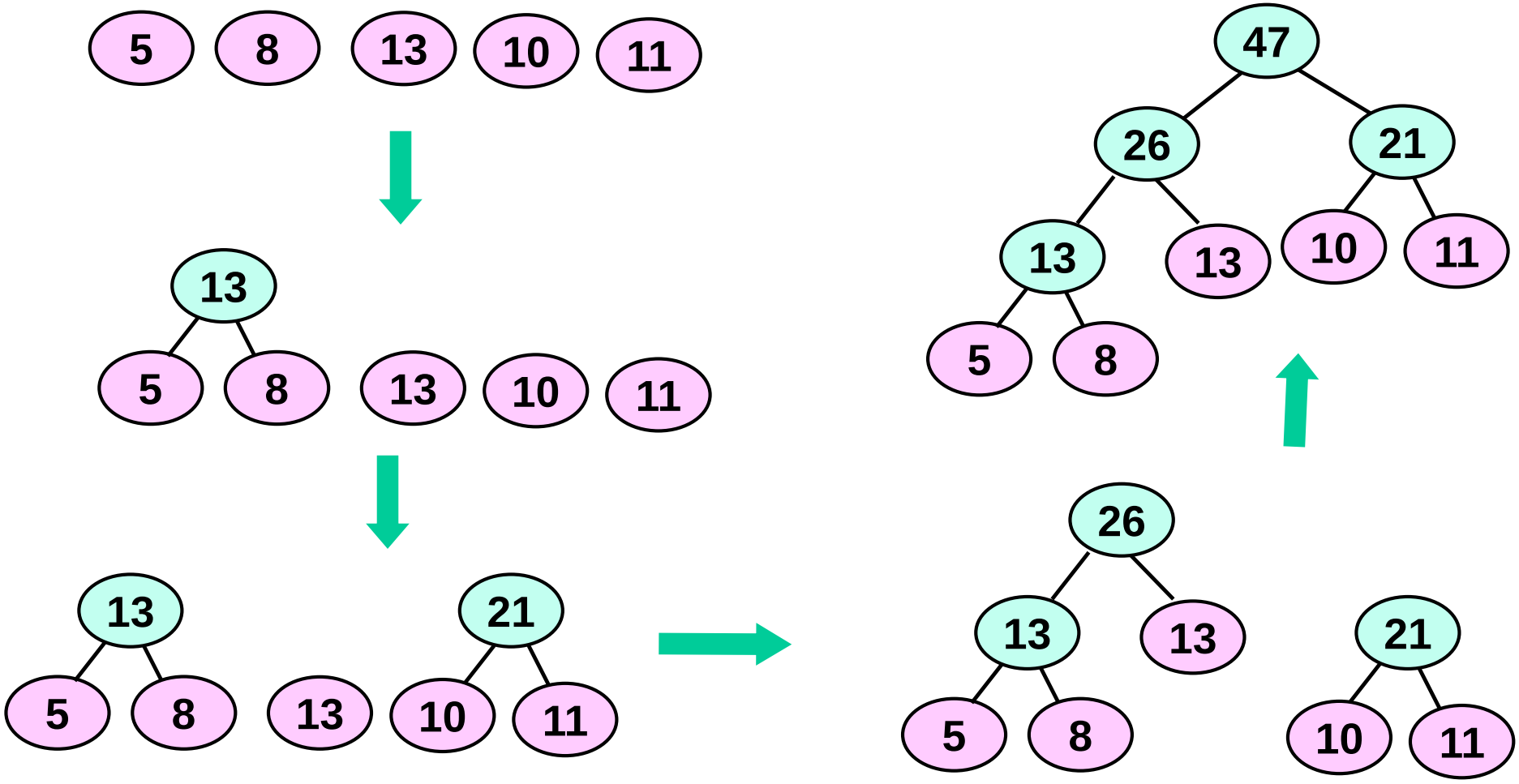
(1) 根据给定的n个权值{$w_1, w_2, \ldots, w_n$}，构造n棵二叉树的集合F = {$T_1, T_2, \ldots, T_n$}，其中每棵二叉树中均只含一个带权值为wi的根结点,其左、右子树均为空;

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

(3) 从F中删去这两棵树，同时加入刚生成的新树;

(4) 重复(2)和(3)两步，直至F中只含一棵树为止。

例：构造以W=（5，8，13，10，11）为权的赫夫曼树。



一棵有n个叶子结点的赫夫曼树共有2n-1个结点。

3. 赫夫曼编码

赫夫曼树除了能求解最优判定问题解，还用于其他一些最优问题的求解。这里介绍用赫夫曼树求解数据的二进制编码。

例　要传输的原文为ABACCDA

设ABCD的编码为

A: 00	A: 0	A: 1
B: 01	B: 00	B: 000
C：10	C: 1	C: 01
D：11	D: 01	D: 001

发送方：将ABACCDA　转换成　000011010

接收方：将 000011010 还原为

在数据传输时，为节省费用，总希望传输的二进制串尽可能短，可采用不等长编码，为出现次数较多的字符编以较短的编码。

若要设计不等长编码，为能进行正确的解码，编码要求是**前缀编码**。

前缀编码：如果在一个编码方案中，任何一个编码都不是其他任何编码的前缀，则称编码是前缀编码。

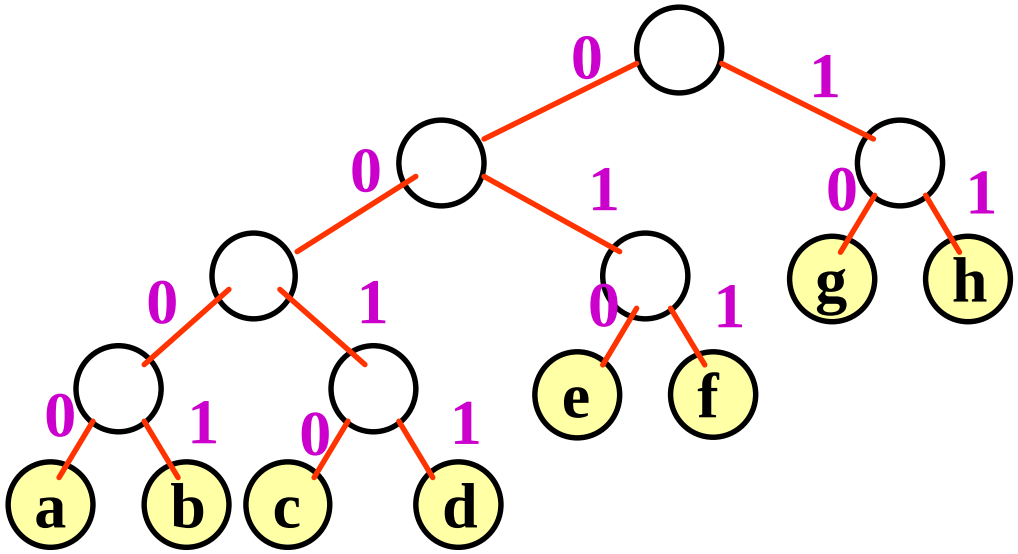
可以利用二叉树来设计前缀编码.

例：某通讯系统只使用8种字符a、b、c、d、e、f、g、h，利用二叉树设计一种不等长编码：

(1)构造以 a、b、c、d、e、f、g、h为叶子结点的二叉树；

(2)将该二叉树所有左分支标记0，所有右分支标记1；

(3)从根到叶子结点路径上标记作为叶子结点所对应字符的编码。



a: 0000
b: 0001
c: 0010
d: 0011
e: 010
f: 011
g: 10
h: 11

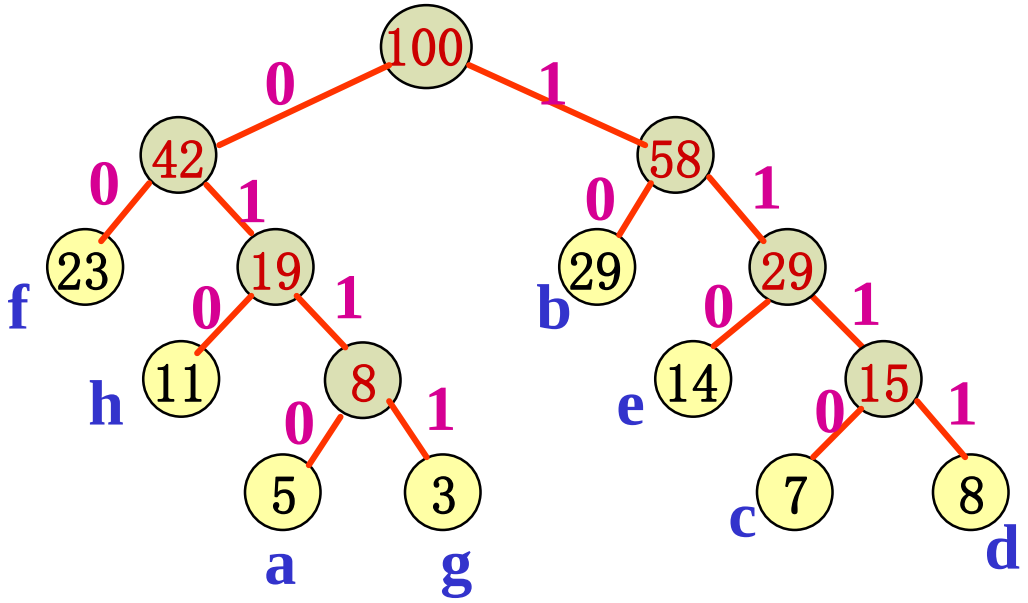
应用中每个字符的使用频率是不一样的。显然，为使传输的二进制串尽可能的短，使用频率高的字符用较短编码，使用频率低的字符用较长编码。

为设计电文总长最短编码，可通过构造以字符使用频率作为权值的哈夫曼树实现。

赫夫曼编码：设计电文总长最短的二进制前缀编码即为以n种字符出现的频率作权，设计一棵赫夫曼树的问题，由此得到的二进制前缀编码称为赫夫曼编码。

例 某通讯系统只使用8种字符a、b、c、d、e、f、g、h，其使用频率分别为0.05，0.29，0.07，0.08，0.14，0.23，0.03，0.11。构造以字符使用频率作为权值的哈夫曼树。

将权值取为整数w=(5，29，7，8，14，23，3，11)，按赫夫曼算法构造的一棵赫夫曼树如下：



对应字符的编码：

a：0110

b：10

c：1110

d：1111

e：110

f：00

g：0111

h：010

4. Huffman树的构造算法

（1） Huffman树的存储表示

- Huffman树中总的结点个数为2n-1个

当给定n个叶子结点构造Huffman树时，共需要进行n-1次合并，每次合并都要产生一个新结点，合并过程共产生n-1个新结点，因此，Huffman树中总的结点个数为2n-1个。

- 如何设计Huffman树的存储结构？

将Huffman树中的2n-1个结点可以存储在一个大小为2n-1的数组中，前n个分量中存放的是叶子结点。

■ Huffman树中结点的结构设计：

Huffman树中每个结点要包含其双亲信息和孩子结点的信息，因此，每个结点的存储结构设计如下：

weight	parent	lchild	rchild

```
//赫夫曼树的存储表示
typedef struct
{   int weight;     //结点的权值
    int parent, lchild, rchild; //结点的双亲、左孩子、右孩子的下标
}HTNode,*HuffmanTree;  //动态分配数组存储哈夫曼树
typedef  char * *HuffmanCode;
(或者typedef char*HuffmanCode[n];)
          //动态分配数组存储Huffman编码表
```

(2) 构造Huffman树HT

```
void  HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
{  //w存放n个字符的权值，构造哈夫曼树HT，并求出n个字符的编码HC
  if  (n<=1)  return;
  m=2*n-1;                          //n 个叶子的Huffman树共有2n-1个结点
  HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode));   //0单元未用

  for(p=HT+1,i=1; i<=n; ++i,++p,++w)*p={*w,0,0,0};  //初始化前n个单元
  for(；i<=m; ++i,++p)   *p ={0,0,0,0};  //对叶子之后的存储单元清零
  for(i=n+1;i<=m; ++i){     //建Huffman树
       Select(HT, i-1, s1, s2);
        //在HT[1…i-1]选择parent为0且weight最小的两个结点，其序号分别为
  S1和s2
       HT[s1].parent=i;  HT[s2].parent=i;
       HT[i].lchild=s1; HT[i].rchild=s2;  //s1、s2分别作为i的左右孩子
       HT[i].weight=HT[s1].weight+ HT[s2].weight;
  }
```

。 。 。 。

	weight	parent	lchild	rchild
1	5	9	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	9	0	0
8	11	0	0	0
9	8	0	1	7
10				
11				
12				
13				
14				
15				

n个叶子结点

n-1个非叶子结点

	weight	parent	lchild	rchild
1	**5**	**9**	**0**	**0**
2	**29**	**14**	**0**	**0**
3	**7**	**10**	**0**	**0**
4	**8**	**10**	**0**	**0**
5	**14**	**12**	**0**	**0**
6	**23**	**13**	**0**	**0**
7	**3**	**9**	**0**	**0**
8	**11**	**11**	**0**	**0**
9	8	11	**1**	**7**
10	15	12	**3**	**4**
11	19	13	**8**	**9**
12	29	14	**5**	**10**
13	42	15	**6**	**11**
14	58	15	**2**	**12**
15	100	0	**13**	**14**

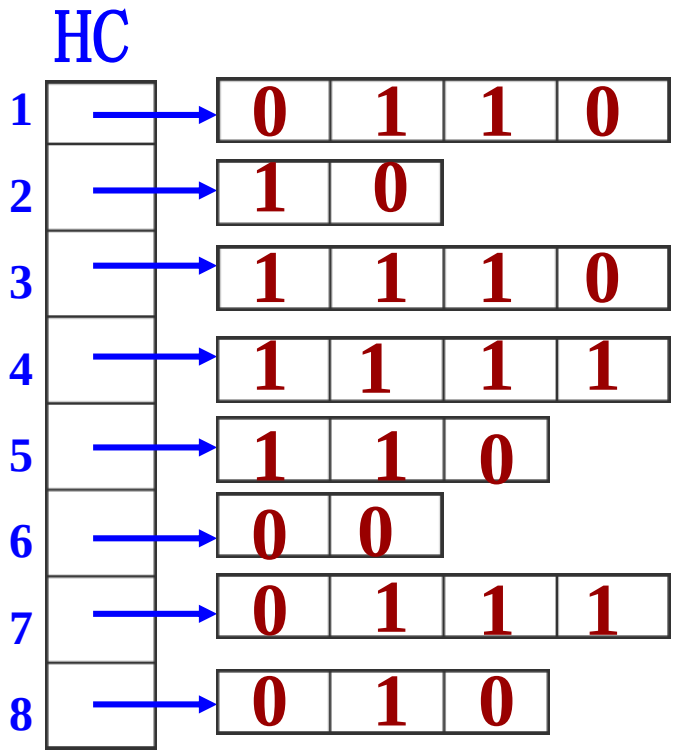
n个叶子结点

n-1个非叶子结点

(3) 求出n个字符的Huffman编码HC

```
    //从叶子到根逆向求每个字符的赫夫曼编码
    HC=(HuffmanCode)malloc((n+1)*sizeof(char*));
            //分配n个字符编码的头指针向量
    cd=(char*) malloc(n*sizeof(char)); //分配求编码的工作空间
    cd[n-1]='\0';  //编码结束符（从cd[0]~cd[n-1]为合法空间）
    for(i=1;i<=n;++i){  //逐个字符求Huffman编码
        start=n-1;   //编码结束符位置
        for(c=i,f=HT[i].parent;   f!=0;   c=f, f=HT[f].parent)
                    //从叶子到根逆向求编码
            if(HT[f].lchild==c) cd[--start]='0';
            else cd[--start]='1';
        HC[i]=(char*)malloc((n-start)*sizeof(char));
                        //为第i个字符编码分配空间
        strcpy(HC[i],&cd[start]); //从cd复制编码串到HC
    }
    free(cd);  //释放工作空间
}
```

求得各个叶子结点所表示的字符的哈夫曼编码如下：



一、假设用于通信的电文由字符集{a, b, c, d, e, f, g}中的字母构成。它们在电文中出现的频度分别为

{0. 31, 0. 16, 0. 10, 0. 08, 0. 11, 0. 20, 0. 04},

1）为这7个字母设计哈夫曼编码；

2）对这7个字母进行等长编码，至少需要几位二进制数

思考题：

下表展示了在一段文本中每个字母出现的次数。对于这段文本，使用Huffman编码比使用等长编码能够节约多少比特的空间？

a	12
e	8
i	15
o	4
u	9