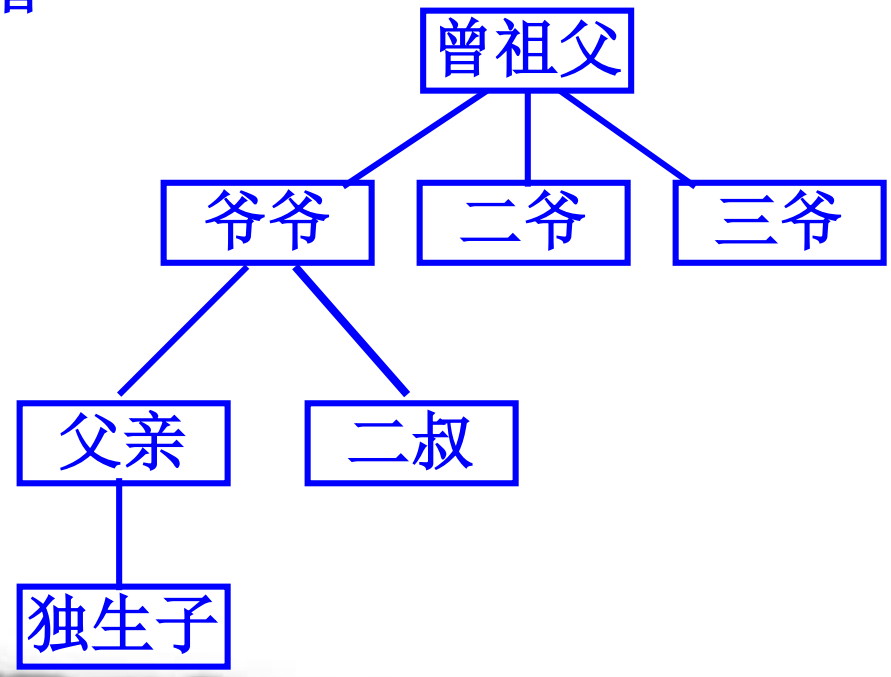
第六章　树和二叉树

族谱



树是以分支关系定义的层次结构。

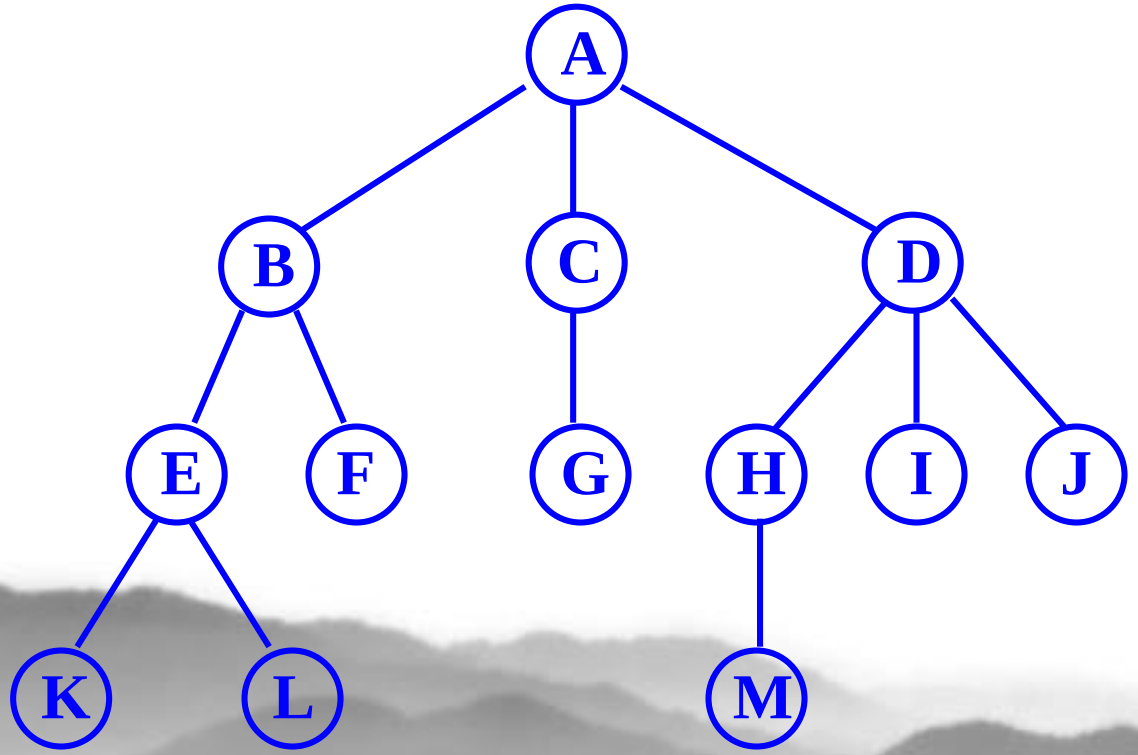
树型结构是一类重要的非线性结构。

学习重点：

- 树的基本概念
- 二叉树的基本概念、二叉树遍历与线索二叉树
- 树和森林与二叉树之间的相互转换
- 二叉树的应用
- 树的等价问题、回溯法与状态树、树的计数[选]

6.1 树的定义和基本术语

树(Tree)：是具有层次结构的 n(n≥0) 个结点的有限集。



一般树

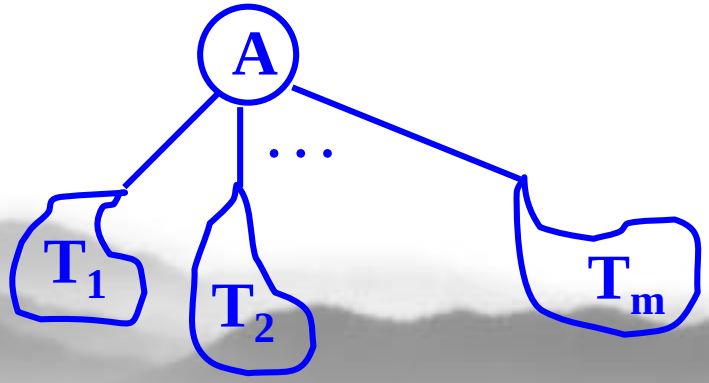
树(Tree)：是 n(n≥0) 个结点的有限集。

n=0，空树 。

n=1，有且仅有一个称为根的结点。

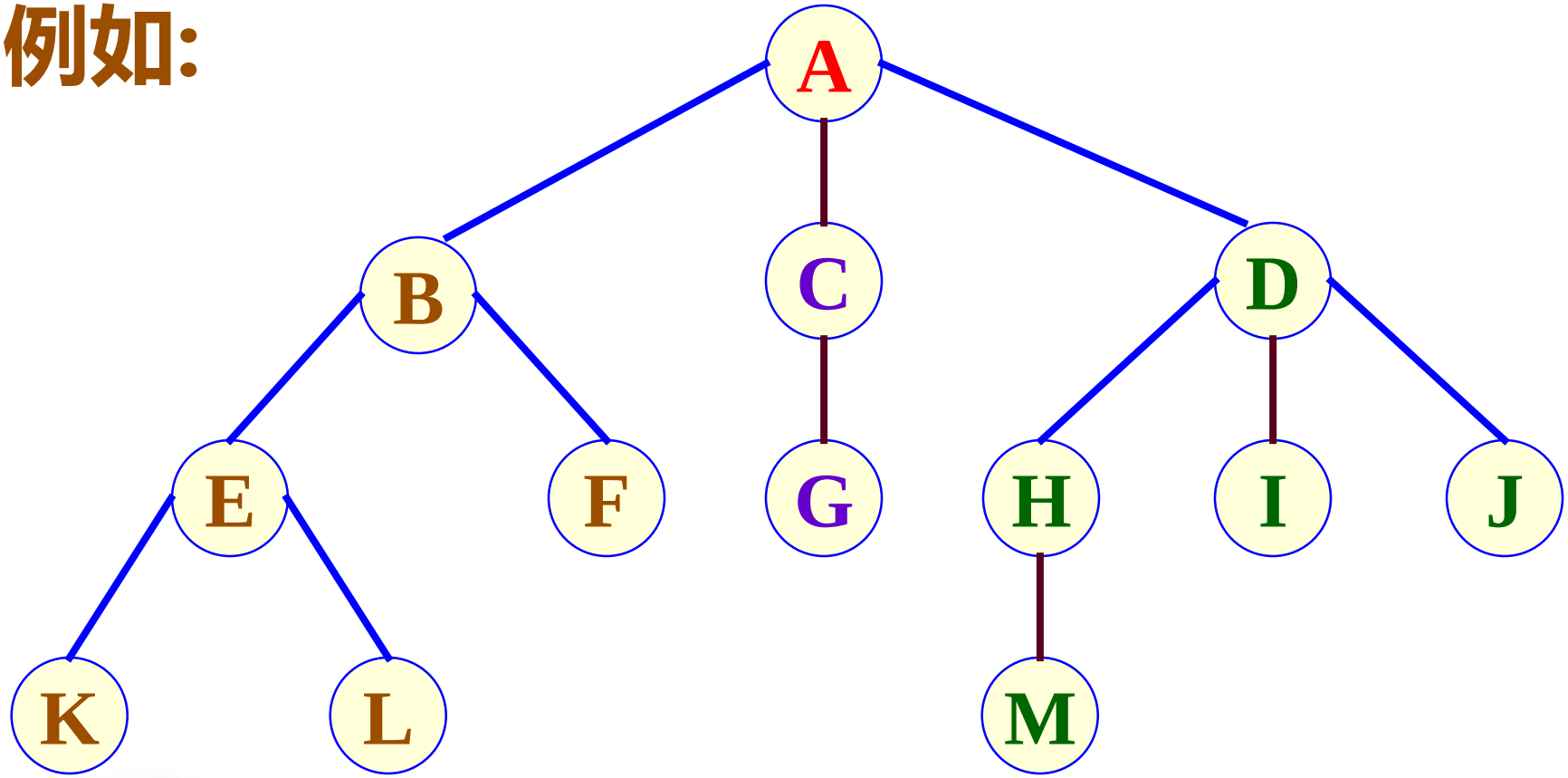
n>1，除根结点外，其余结点可分为 m(m>0) 个互不相交的有限子集 $T_1, T_2, \ldots T_m$，其中每个子集都称为根结点的子树 。

只有根结点的树



n＞1

例如:



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

树根

T_1 T_2 T_3

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。

例，A 的度为 3 ，F 的度为 0 。

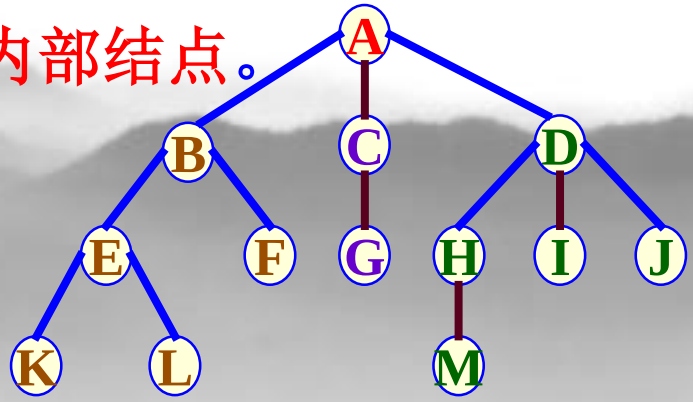
度为0的结点称为叶子或终端结点。

度不为0的结点称为分支结点或非终端结点。

例，K,L,F,G,M,I,J 为叶子；A,B,C,D,E,H 为分支结点 。

除根结点外，其余分支结点也称为内部结点。

例，B,C,D,E,H 为内部结点 。



树的度是树内各结点的度的最大值。

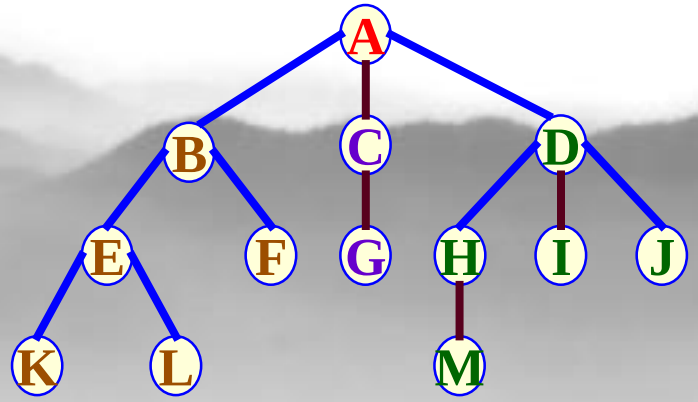
例，树的度为 3。

结点的子树的根称为该结点的儿子，结点称为孩子的父亲。

例，B,C,D 是 A 的儿子，A 是 B,C,D 的父亲。

同一个父亲的孩子之间互称兄弟。

例，B,C,D 互为兄弟。



从根到结点所经分支上的所有结点称为该结点的祖先。

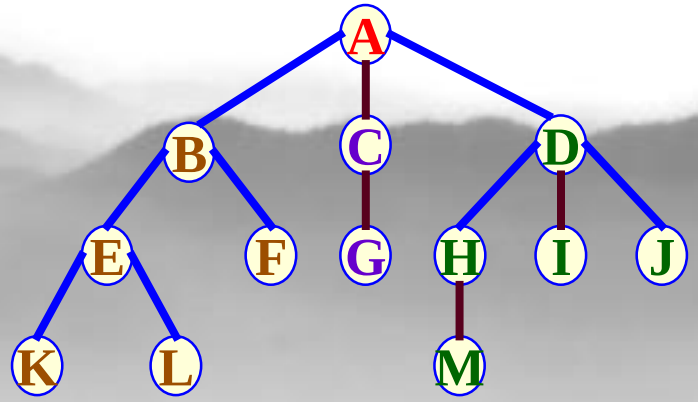
例，M 的祖先为 H,D,A。

以某结点为根的子树中的任一结点都称为该结点的子孙。

例，B 的子孙有 E,K,L,F。

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。

例，A 在第一层，B,C,D 在第二层。



其父亲在同一层的结点互为堂兄弟。

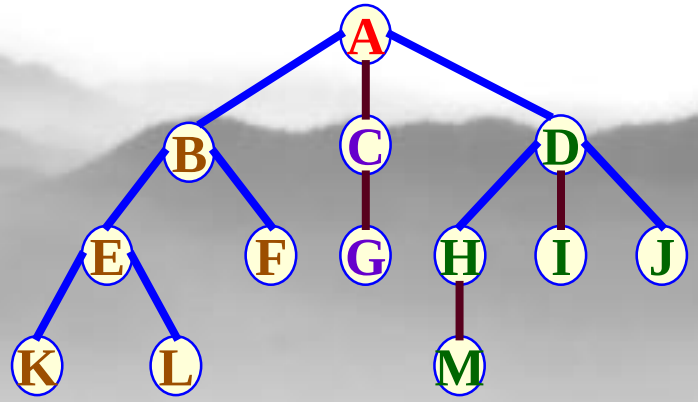
例，G 与 E,F,H,I,J 互为堂兄弟。

树中结点的最大层次称为树的深度或高度。

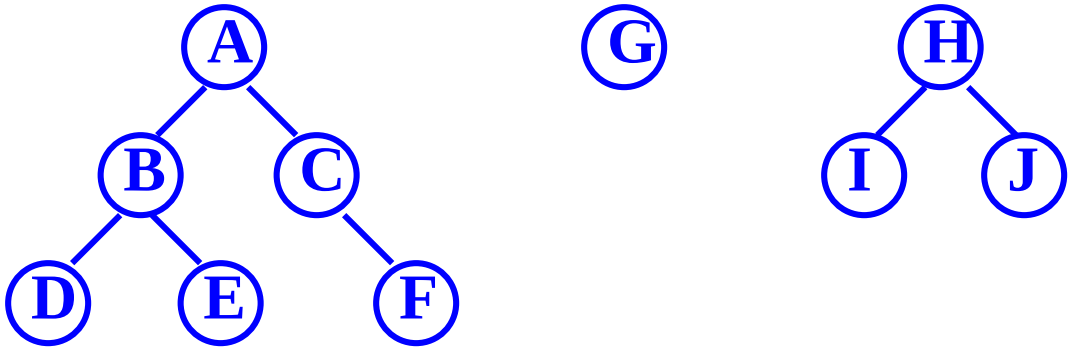
例，树的深度为 4 。

如果将树中结点的各子树看成从左到右是有序的(即不能互换)，则称该树为有序树，否则称为无序树。

例，B,C,D 分别称为 A 的第 1，2，3 个儿子 。



森林是m(m≥0)棵互不相交的树的集合。



树属于森林。

树是一个二元组 Tree = (root，F) 。

root： 根结点。

F： m(m≥0)棵树的森林，$F = (T_1, T_2, \ldots, T_m)$。

$T_i = (r_i, F_i)$ 称为 root 的第 i 棵子树。

$m \neq 0 \Rightarrow RF = \{<root, r_i> \mid i = 1, 2, ..., m, m > 0\}$

ADT Tree{

数据对象 D： $D=\{a_i \mid a_i \in ElemSet,\ i=1,2,\ldots,n,\ n \geqslant 0\}$

数据关系 R：

若D为空集，则称为**空树**。

若D仅含一个元素，则R为空集，否则R={H}，H是如下二元关系：

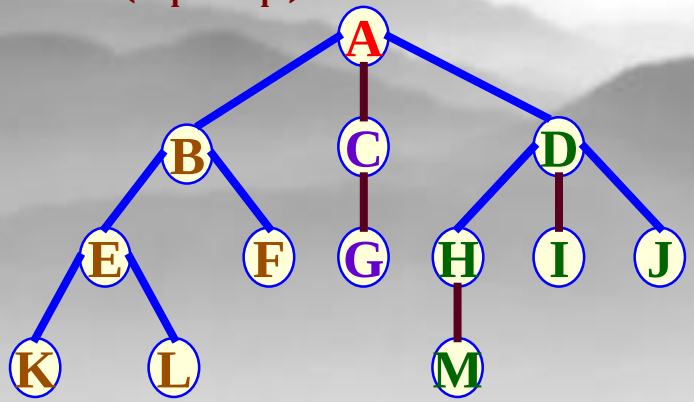
(1) 在D中存在唯一的称为**根**的数据元素**root**，它在关系H下无前驱；

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

(3) 对应于$D-\{root\}$的划分，$H-\{<root, x_1>,\ldots, <root, x_m>\}$有唯一划分$H_1, H_2, \ldots, H_m(m>0)$，对任意$j\neq k(1\leq j,k\leq m)$有$H_j\cap H_k=\Phi$，且对任意$i(1\leq i\leq m)$，$H_i$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。

// 基本操作

} ADT Tree



{
<A, B>, <A, C>, <A, D>,
<B, E>, <B, F>,
<C, G>,
<D, H>, <D, I>, <D, J>
<E, K>, <E, L>, <H, M>
}

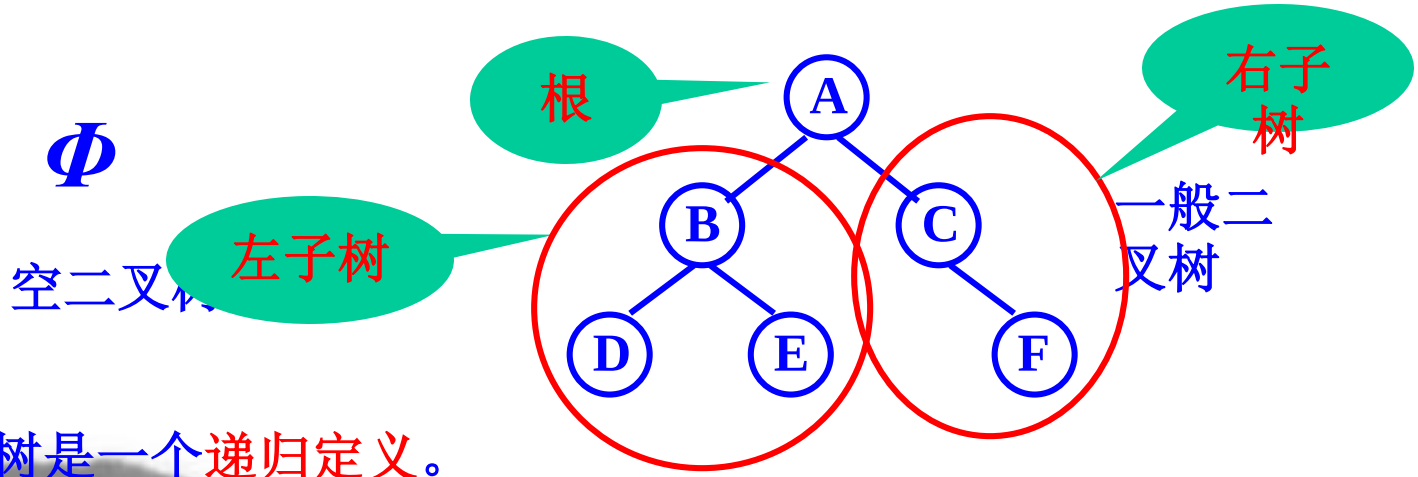
基本操作：

```
InitTree(&T)                    // 初始化置空树
CreateTree(&T, definition)      // 按定义构造树
DestroyTree(&T)                 // 销毁树的结构
ClearTree(&T)                   // 将树清空
TreeEmpty(T)                    // 判定树是否为空树
TreeDepth(T)                    // 求树的深度
Root(T)                         // 求树的根结点
Value(T, cur_e)                 // 求当前结点的元素值
Assign(T, cur_e, value)         // 给当前结点赋值
Parent(T, cur_e)                // 求当前结点的双亲结点
LeftChild(T, cur_e)             // 求当前结点的最左孩子
RightSibling(T, cur_e)          // 求当前结点的右兄弟
InsertChild(&T, &p, i, c)       // 将以c为根的树插入为结点p
                                // 的第i棵子树
DeleteChild(&T, &p, i)          // 删除结点p的第i棵子树
TraverseTree( T, Visit() )      // 遍历
```

6.2 二叉树

6.2.1 二叉树(Binary Tree)的定义

二叉树是 n(n≥0) 个结点的有限集，它或者是空集，或者是由一个根和称为左、右子树的两个互不相交的二叉树组成。



二叉树是一个递归定义。

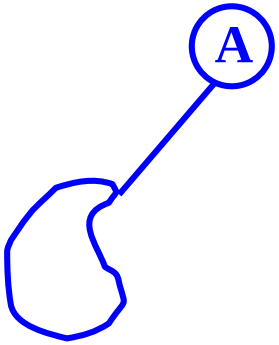
树的子树次序不作规定，二叉树的两个子树有左、右之分。

树中结点的度没有限制，二叉树中结点的度只能取 0、1、2。

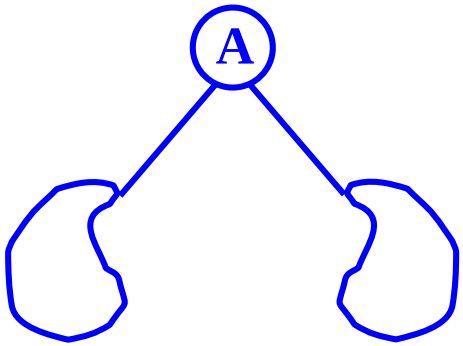
根据定义，二叉树通常具有 5 种基本形态：

空二叉树

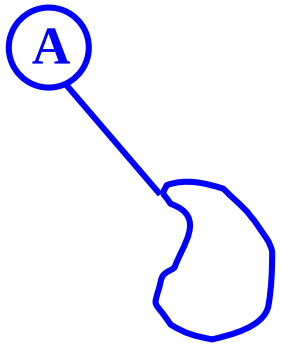
仅有根结点的二叉树



右子树为空的二叉树



左、右子树均非空的二叉树



左子树为空的二叉树

6.1 节关于树的基本术语也都适用于二叉树。

根据树的ADT练习定义二叉树的ADT

6.2.2 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($i \geq 1$)。

归纳法证明：

(1) i = 1，只有一个根结点，$2^{i-1} = 2^0 = 1$，正确；

(2) 假设 i−1 成立，即第 i−1 层上至多有 2^{i-2} 个结点；

(3) 由于二叉树的结点的度至多为 2，故在第 i 层上的最大结点数为第 i−1 层上的最大结点数的 2 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($k\geq 1$)。

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

设有n个树枝

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

证明：(1) 已知，终端结点数为 n_0，度为 2 的结点数为 n_2，

设度为 1 的结点数为 n_1，

由于二叉树中的所有结点的度只能为 0、1、2，

故二叉树的结点总数为 $n = n_0 + n_1 + n_2$；

(2) 除根结点外，其它结点都有一个分支进入，

设 B 为分支总数，故 $n = B + 1$，

由于这些分支均是由度为 1 或 2 的结点引出的，

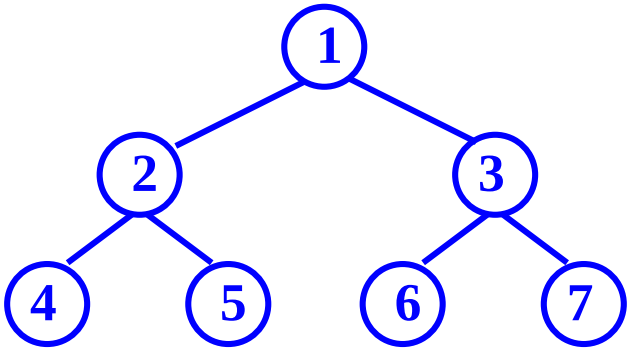
所以有 $B = n_1 + 2n_2$，故 $n = n_1 + 2n_2 + 1$，

由 (1) 和 (2)，可得 $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$，

故有 $n_0 = n_2 + 1$。

两种特殊形态二叉树：满二叉树、完全二叉树。

一棵深度为 k 且有 2^k-1 个结点的二叉树称为满二叉树。



满二叉树

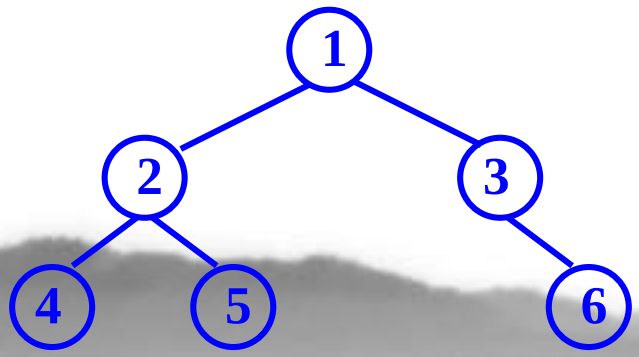
特点：(1) 每一层的结点数都达到最大结点数。

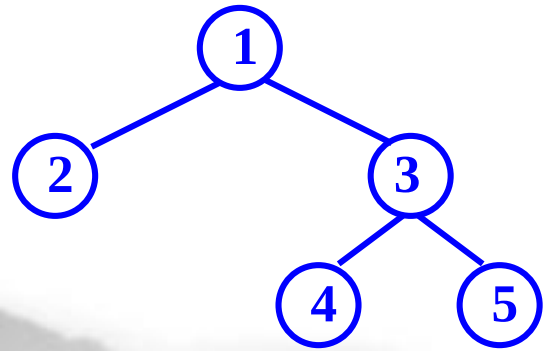
(2) 叶子结点在最大层。

(3) 任一结点，其左、右分支下的子孙的最大层次相等。

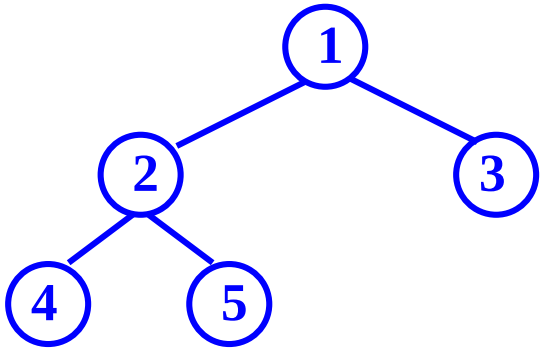
对满二叉树的结点进行连续编号，从根结点起，自上而下，自左至右，1、2、3、……、2^k-1 。

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，称为完全二叉树。

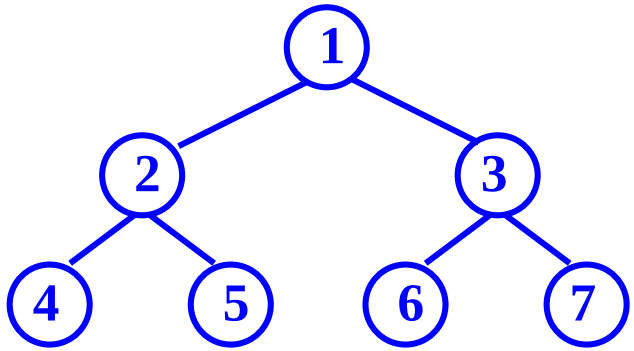




非完全二叉树



完全二叉树(1)



完全二叉树(2)

特点：

(1) 叶子结点只可能在层次最大的两层上出现。

(2) 对任一结点，若其右分支的子孙的最大层次为 *l* ，则其左分支下的子孙的最大层次必为 *l* 或 *l*+1。

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

证明： 设 n 结点完全二叉树的深度为 k ，

因为，

k−1 层满二叉树结点数	<	k 层完全二叉树结点数	≤	k 层满二叉树结点数

故 $2^{k-1}-1 < n \leq 2^k-1$

有 $2^{k-1} \leq n < 2^k$

又 $k-1 \leq \log_2 n < k$

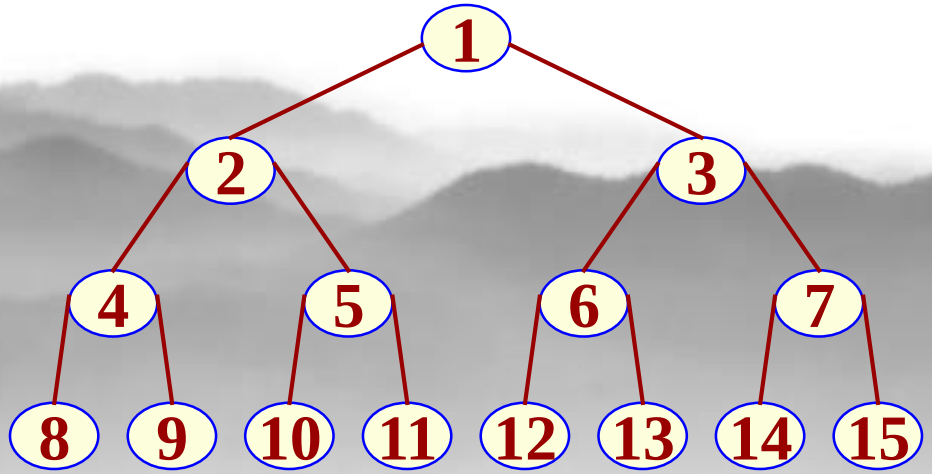
因为 k 是整数，所以 $k = \lfloor \log_2 n \rfloor + 1$ 。

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

(2) 若 *2i>n*，则该结点无左孩子(结点为叶结点)，否则其**左孩子**是结点 *2i*；

(3) 若 *2i+1>n*，则该结点无右孩子，否则其**右孩子**为结点*2i+1*。



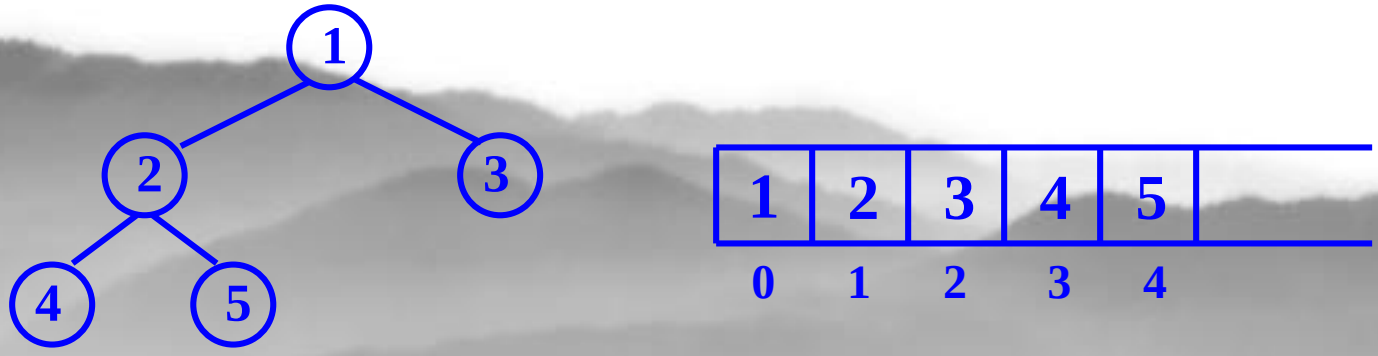
6.2.3 二叉树的存储结构 (顺序、链式)

1. 顺序存储结构

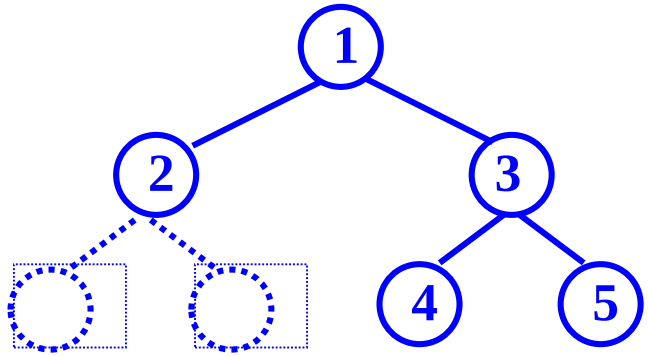
用一组地址连续的存储单元依次*自上而下、自左至右*存储二叉树上的结点元素。

```
# define   MAX_TREE_SIZE     100
typedef   TElemType   SqBiTree[MAX_TREE_SIZE]
```

完全二叉树：编号为 i 的元素存储在数组下标为 i−1 的分量中；



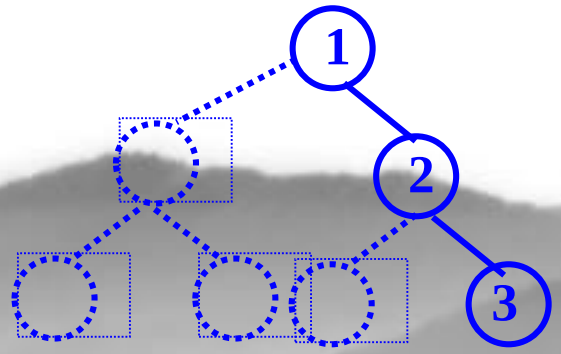
一般二叉树：对照完全二叉树，存储在数组的相应分量中；



1	2	3	0	0	4	5	
0	1	2	3	4	5	6	

0 表示不存在此结点

在最坏情况下，深度为 k 的右单支二叉树需要 2^k-1 个存储空间。



1	0	2	0	0	0	3	
0	1	2	3	4	5	6	

空间浪费

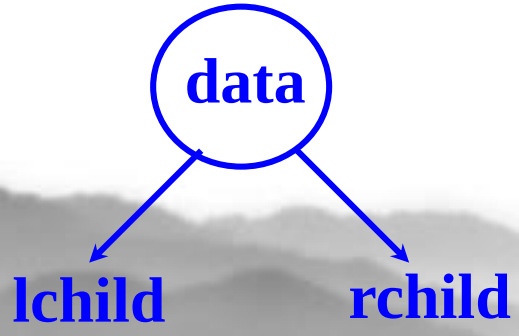
结论：顺序存储结构适用于完全二叉树。

2. 链式存储结构

$D = (root, D_L, D_R)$。

链表结点包含 3 个域： 数据域、左指针域、右指针域

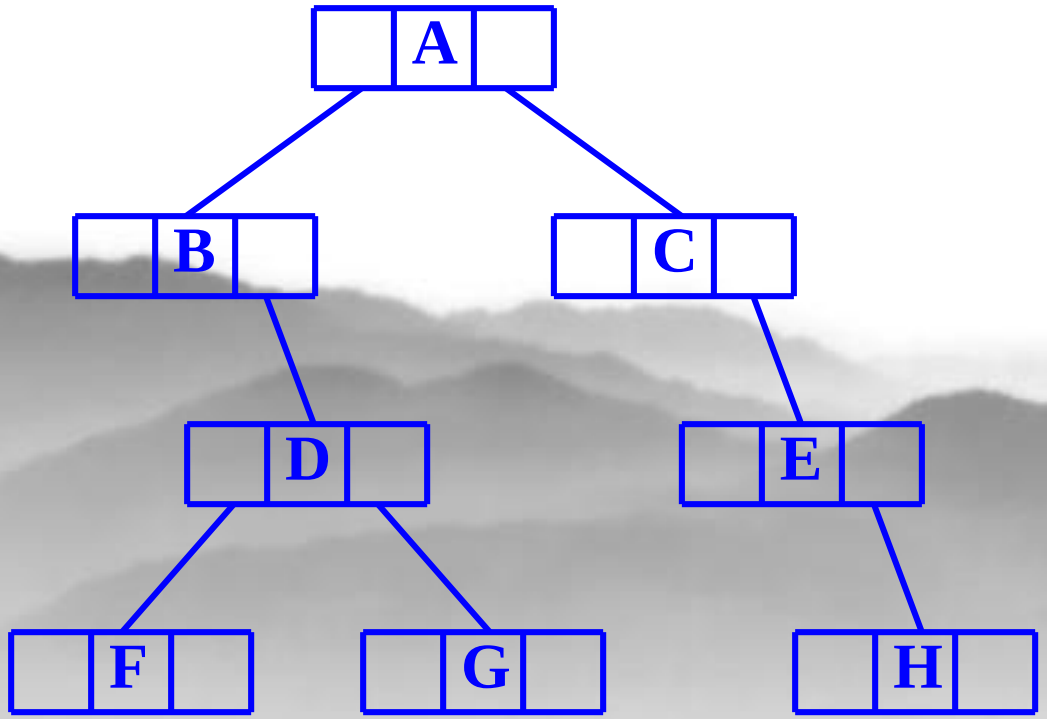
左指针域　数据域　右指针域



由这种结点结构得到的二叉树存储结构称为二叉链表。

二叉链表存储表示

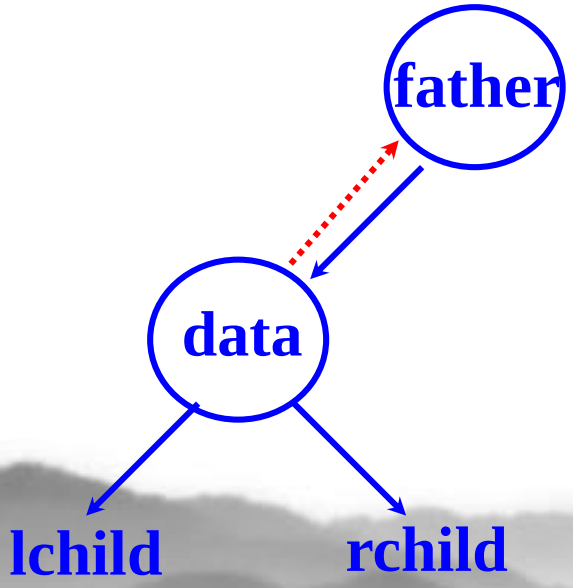
```
struct   BitNode  {
    TElemType          data ;
    struct   BitNode     * lchild ,  * rchild ;
}
```



有时还可以在结点结构中增加一个指向父亲的指针。

lchild	data	father	rchild

左指针域　数据域　父指针域 右指针域



采用何种存储结构，依赖于进行何种操作？

6.3 遍历二叉树和线索二叉树

6.3.1 遍历二叉树

遍历二叉树：如何按某条搜索路径巡访树中每个结点，使得每个结点均被访问一次，而且仅被访问一次。

线性结构的遍历；

非线性结构—二叉树的遍历。

$D = (root, D_L, D_R)$。

如果能依次遍历这三部分，就可以遍历整个二叉树；

设以 D、L、R 分别表示访问根结点、遍历左子树、遍历右子树，则可以存在 6 种遍历方案: DLR、DRL、LDR、LRD、RDL、RLD；

若限定先左后右的原则，则只有 3 种情况:

先(根)序遍历

中(根)序遍历

后(根)序遍历。

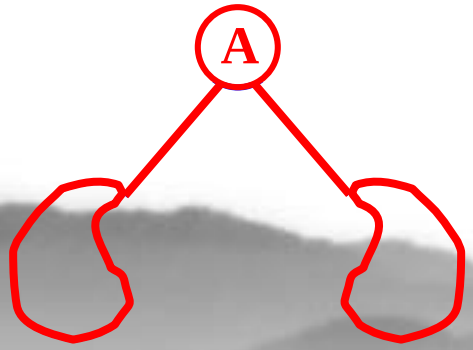
先(根)序遍历:

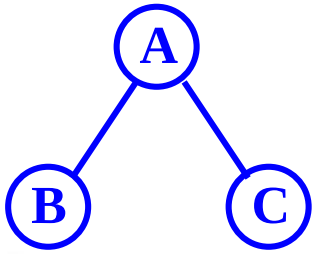
若二叉树为空，则返回；否则

(1) 访问根结点；

(2) 先(根)序遍历左子树；

(3) 先(根)序遍历右子树；



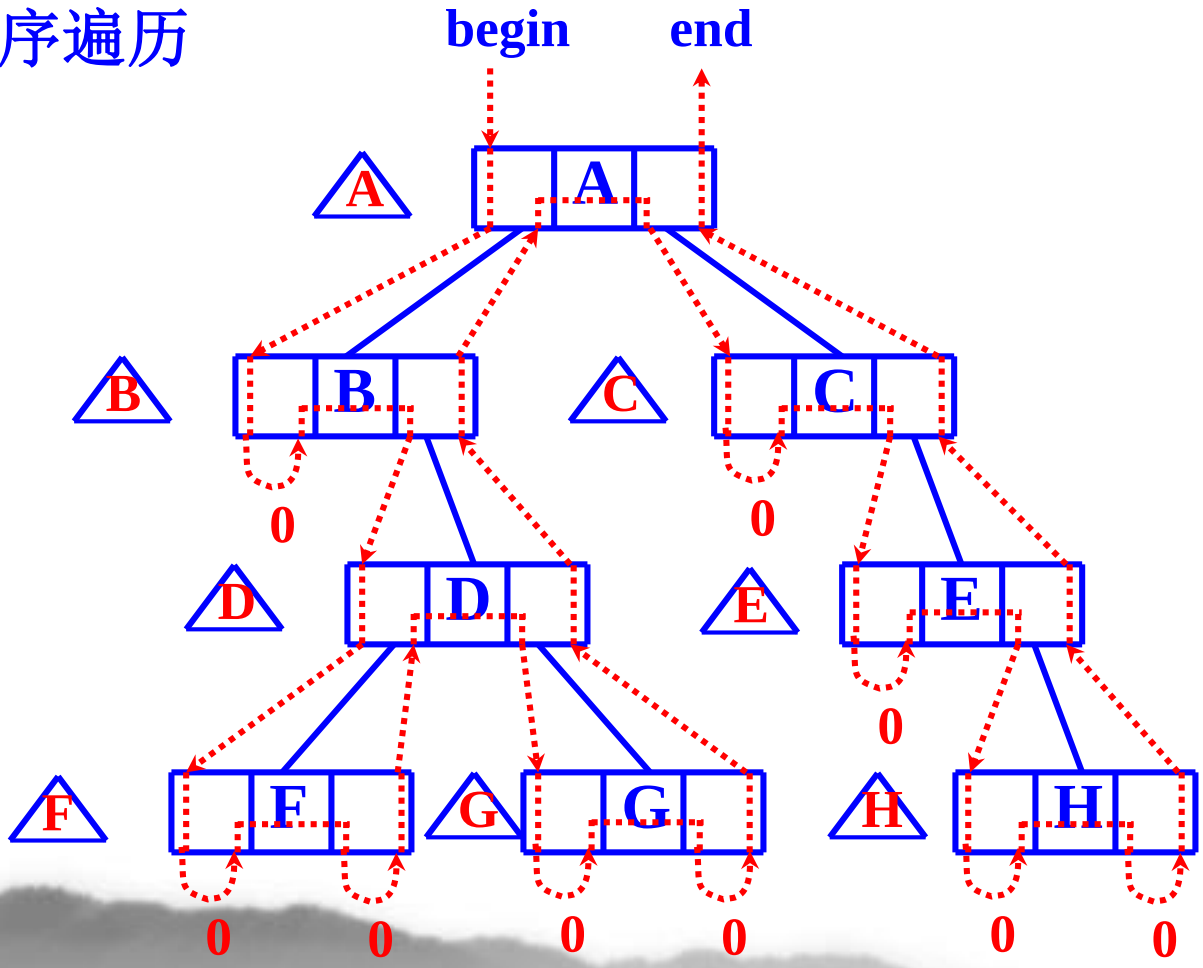


A B C

算法 6.1　先序遍历递归算法

```
Status PreOrderTraverse ( BiTree T , printf(e) )
{    // visit(e) 函数可以看作 printf (e)
     If  (T)  {
          printf(T->data) ;
          PreOrderTraverse ( T->lchild, printf ) ;
          PreOrderTraverse ( T->rchild, printf ) ;
     }
     else   return OK ;
}
```

先序遍历



先序遍历顺序： A B D F G C E H

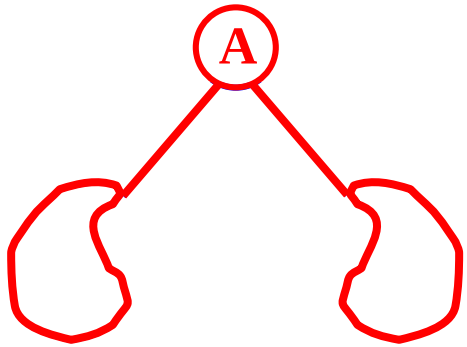
中(根)序遍历:

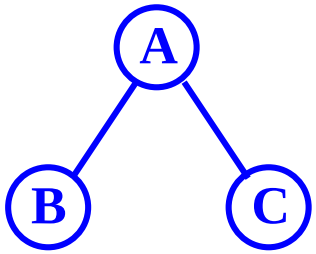
若二叉树为空，则返回；否则

(1) 中(根)序遍历左子树；

(2) 访问根结点；

(3) 中(根)序遍历右子树；



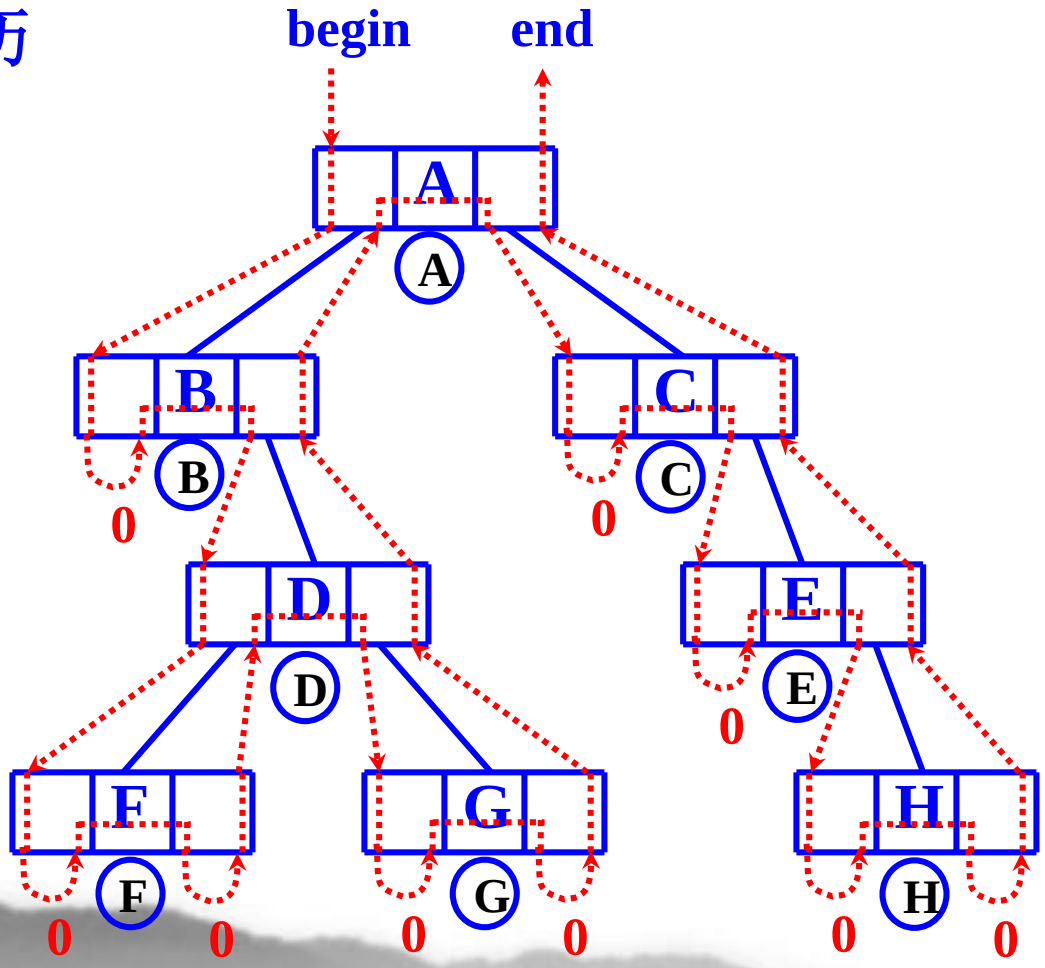


B A C

算法 6.2　中序遍历递归算法

```
Status InOrderTraverse ( BiTree T , printf(e) )
{     // visit(e) 函数可以看作 printf (e)
    If  (T)  {
        InOrderTraverse ( T->lchild, printf ) ;
        printf(T->data) ;
        InOrderTraverse ( T->rchild, printf ) ;
    } else   return OK ;
}
```

中序遍历



中序遍历顺序： B F D G A C E H

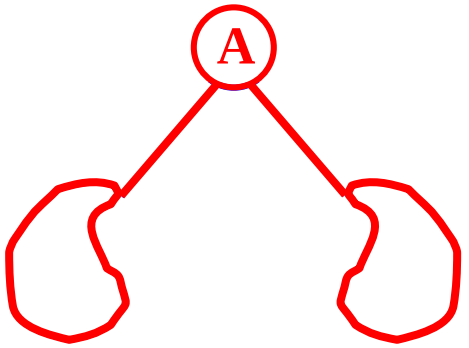
后(根)序遍历:

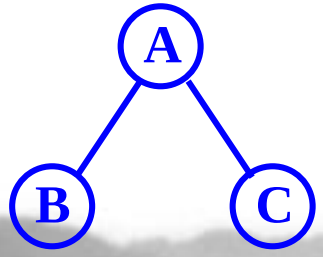
若二叉树为空，则返回；否则

(1) 后(根)序遍历左子树；

(2) 后(根)序遍历右子树；

(3) 访问根结点；



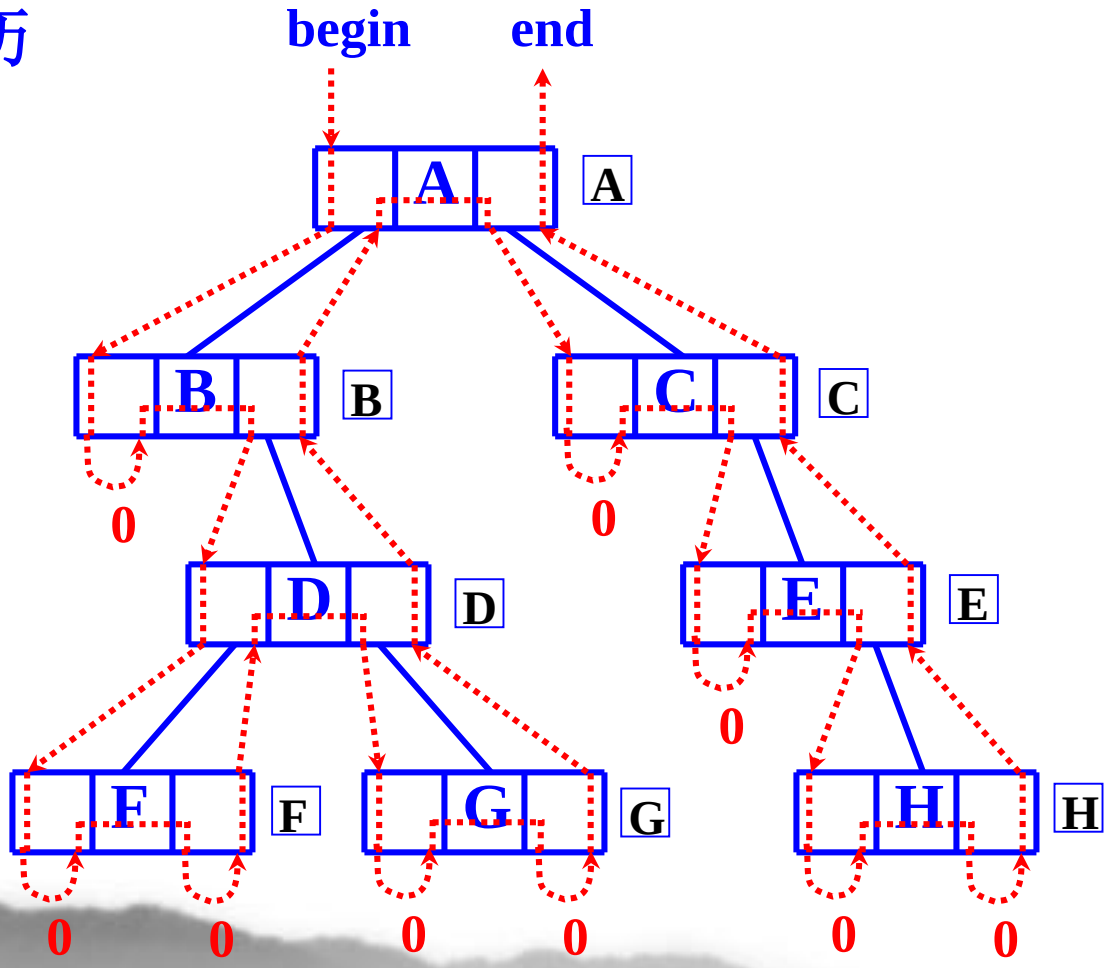


B C A

算法 6.3　后序遍历递归算法

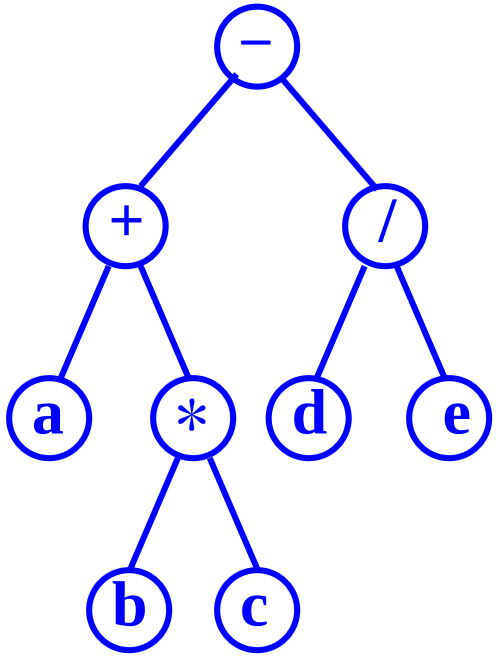
```
Status PostOrderTraverse ( BiTree T , printf(e) )
{      // visit(e) 函数可以看作 printf (e)
     If  (T)   {
          PostOrderTraverse ( T->lchild, printf ) ;
          PostOrderTraverse ( T->rchild, printf ) ;
          printf(T->data) ;
     }  else   return OK ;
}
```

后序遍历



后序遍历顺序：F G D B H E C A

例、表达式树： a + b * c – d / e



先序遍历:

– + a * b c / d e

前缀式/波兰式

中序遍历:

a + b * c – d / e

中缀式/算术表达式，适于人的思维

后序遍历:

a b c * + d e / –

后缀式/逆波兰式，适于计算机的思维

算法 6.2　非递归中序遍历算法

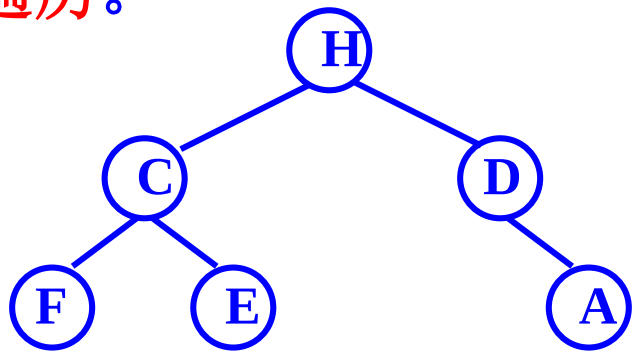
```
void Inorder_Traverse(BiTree T, void (*Visit)(TElemType& e))
{ // 采用二叉链表存储结构，Visit是对数据元素操作的应用函数。
  InitStack(S); Push(S, T);
  while(!StackEmpty(S )){
    while(GetTop(S, p) && p)    // 向左走到尽头(far left leaf)
        Push(S, p->lchild);
    Pop(S, p);   // 空指针退栈
    if ( !StackEmpty(S )){    // 访问结点，向右一步
        Pop(S, p);
        if(!Visit(p->data)) return ERROR;
        Push(S, p->rchild);
    } // if
  } // while
  return OK;
}// Inorder_Traverse
```

创建二叉树的算法见pp131
CreateBiTree()

算法 6.3　非递归中序遍历算法

```
void Inorder_Traverse(BiTree T, void (*Visit)(TElemType & e))
{ // 采用二叉链表存储结构，Visit是对数据元素操作的应用函数。
  InitStack(S); p = T;
  while(p || !StackEmpty(S )){
    if(p){   // 根指针进栈，遍历左子树
        Push(S, p);
        p = p->lchild;
    }
    else{
        Pop(S, p);   // 根指针退栈，遍历右子树
        if(!Visit(p->data)) return ERROR;
        p=p->rchild;
    } // else
  } // while
  return OK;
}// Inorder_Traverse
```

对二叉树除可以进行先序、中序、后序的遍历外，还可以进行层次遍历。



层次遍历：H，C，D，F，E，A

过程：

打印 H；

打印 H 的左儿子 C；

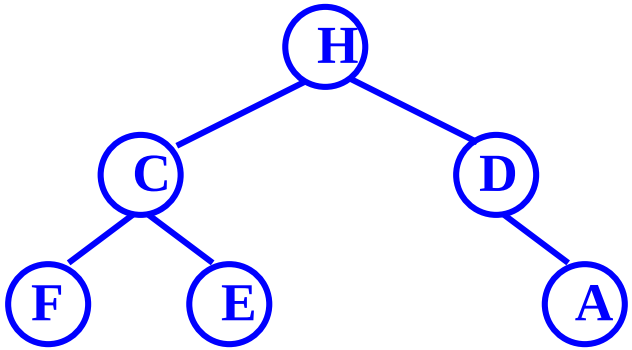
打印 H 的右儿子 D；

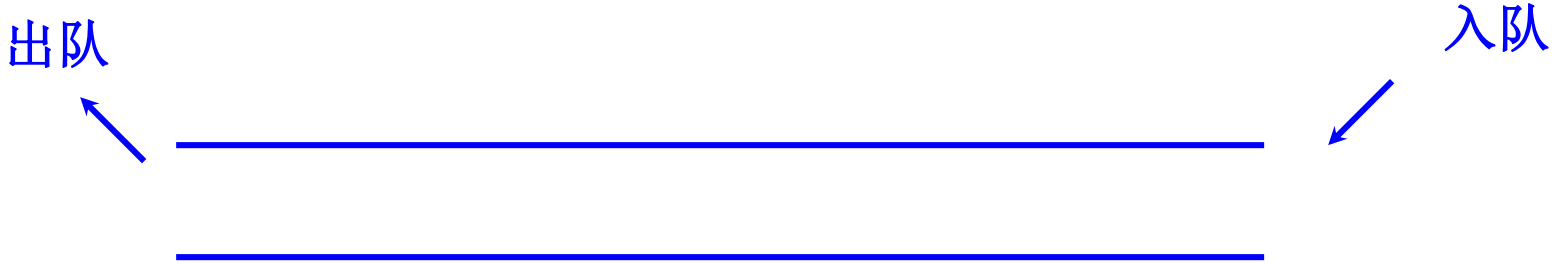
打印 C 的左、右儿子 F、E；

打印 D 的左、右儿子 A；

栈实现？

队列实现层次遍历



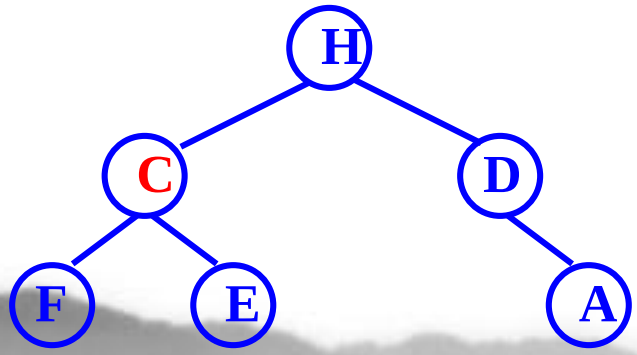


H C D F E A

6.3.2 线索二叉树

二叉树的遍历实现了对一个非线性结构进行线性化的操作，从而使每个结点在线性序列中有且仅有一个直接前驱和直接后继。 => 不完全双向链

但以二叉链表作为存储结构时，



如何直接找到任意结点的前驱和后继结点？

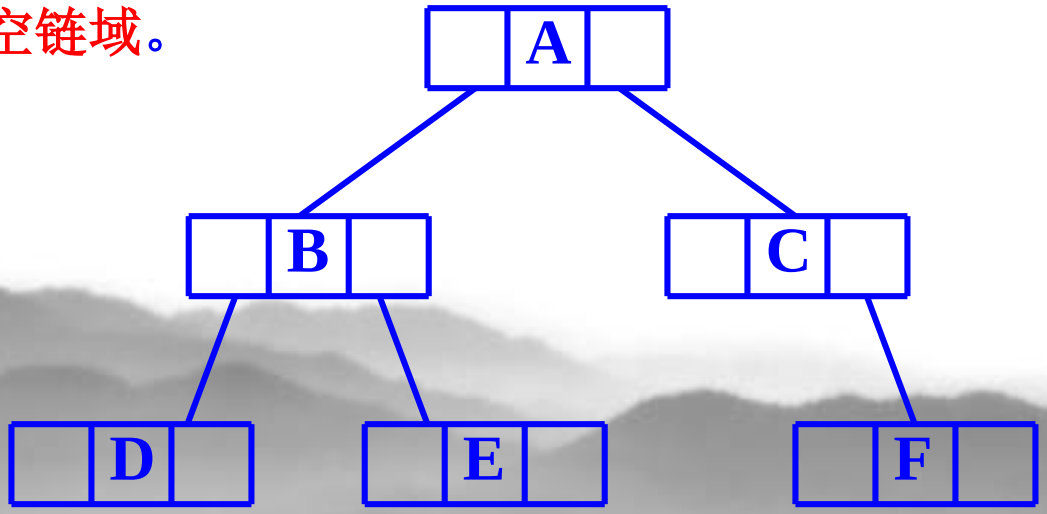
方法1：增加两个指针域 fwd 和 bkwd，分别指示其前驱和后继。

优点：实现方便、简单。

缺点：需要大量额外空间。

fwd	lchild	data	rchild	bkwd
前驱	左指针域	数据域	右指针域	后继

方法2：利用闲余的空链域。



性质：含有 n 个结点的二叉链表中有 n+1 个空链域。

证明：(1) 设，终端结点数为 n_0，

度为 1 的结点数为 n_1，

度为 2 的结点数为 n_2，

故二叉树的结点总数为 $n = n_0 + n_1 + n_2$；

(2) 空链域个数为 $2n_0 + n_1$，

已知，$n_0 = n_2 + 1$，

故，$2n_0 + n_1$

$= n_0 + n_1 + n_2 + 1$

$= n + 1$

如何利用空链域描述前驱和后继信息？

lchild	LTag	data	RTag	rchild

其中：

$$LTag = \begin{cases} 0 & \text{lchild域指示结点的左儿子} \\ 1 & \text{lchild域指示结点的前驱结点} \end{cases}$$

$$RTag = \begin{cases} 0 & \text{rchild域指示结点的右儿子} \\ 1 & \text{rchild域指示结点的后继结点} \end{cases}$$

增加线索的二叉树称之为线索二叉树。

指向结点前驱和后继的指针称为线索。

二叉树按某种次序成为线索二叉树的过程叫做线索化

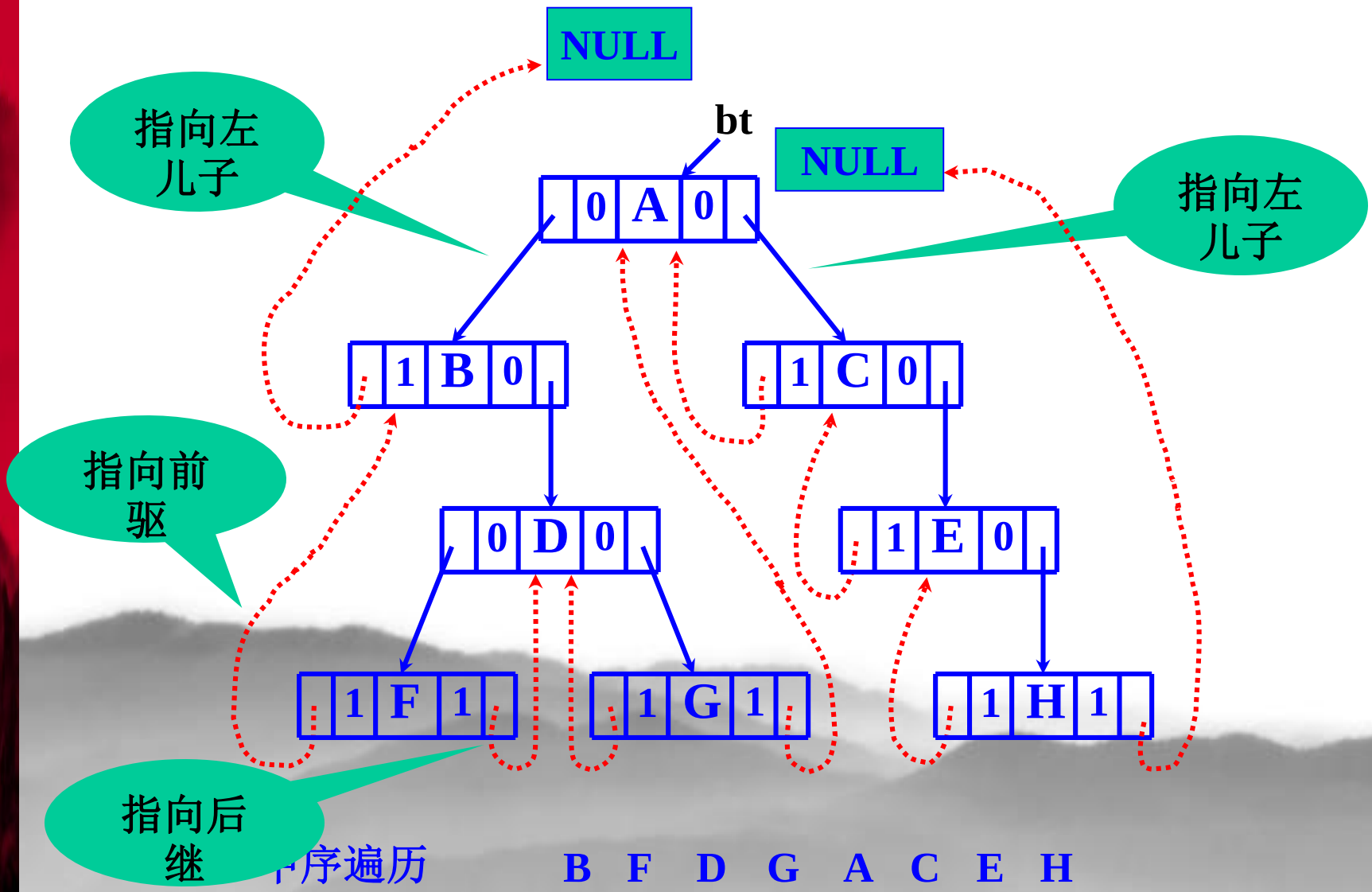
二叉线索树的存储表示

```
typedef   enum  PionterTag  ( Link ，Thread ) ;   // 0，1

typedef  struct  BiThrNode  {

    TElemType              data ;

    struct   BiThrNode     * lchild ，* rchild ;

    PointerTag             LTag ，RTag ;

}
```

！ ！ ！

线索二叉链表的建立依据二叉树的遍历顺序

例，中序线索链表


NULL
bt
NULL
指向左儿子
指向左儿子
0 A 0
1 B 0
1 C 0
指向前驱
0 D 0
1 E 0
1 F 1
1 G 1
1 H 1
指向后继
中序遍历
B F D G A C E H

线索链表的遍历算法:

由于在线索链表中添加了遍历中得到的“前驱”和“后继”的信息，从而简化了遍历的算法。

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

例如:

对中序线索化链表的遍历算法

※ 中序遍历的第一个结点？

左子树上处于“**最左下**”（没有左子树）的结点。

※ 在中序线索化链表中结点的后继？

若无右子树，**则为后继线索**所指结点；

否则为对其**右子树**进行中序**遍历**时访问的**第一个结点。--右子数中的最左下结点**

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

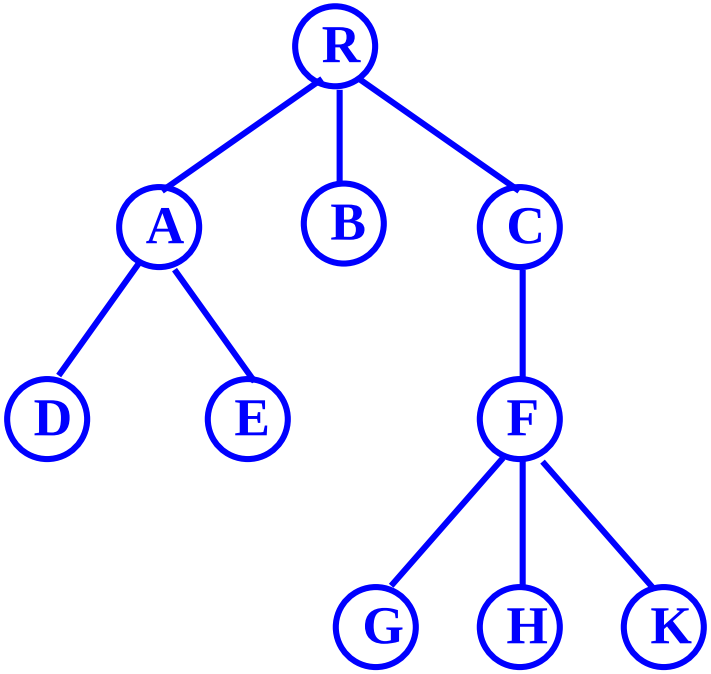
如何建立线索链表?

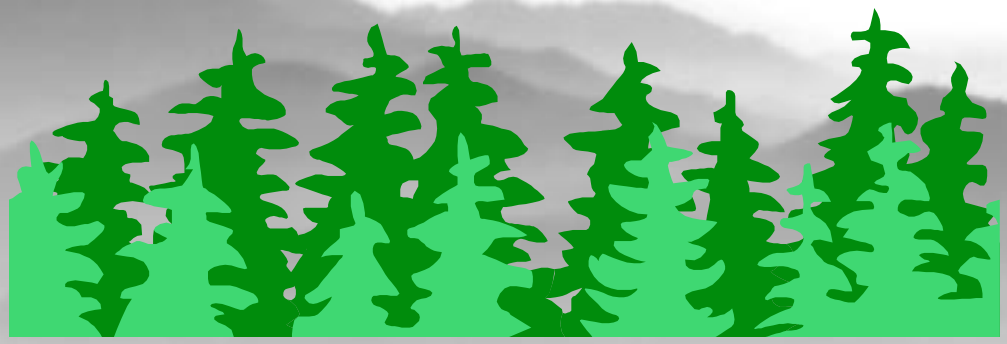
在中序遍历过程中修改结点的左、右空指针域，以保存当前访问结点的“前驱”和“后继”信息。遍历过程中，附设指针pre并始终指向当前访问的、指针p所指结点的前驱。

```
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T)
{ // 构建中序线索链表，中序遍历线索化后Thrt为头结点
  if (!(Thrt = (BiThrTree)malloc(sizeof( BiThrNode))))
     exit (OVERFLOW);
  Thrt->LTag = Link;  Thrt->RTag =Thread;   // 建头结点
  Thrt->rchild = Thrt;                      // 右指针回指
 if (!T)  Thrt->lchild = Thrt;              // 树为空，左指针回指
 else {
    Thrt->lchild = T;   pre = Thrt;
    InThreading(T);                         // 中序遍历同时线索化
    pre->rchild = Thrt;                     // 处理最后一个结点
    pre->RTag = Thread;
    Thrt->rchild = pre;
  }
  return OK;
} // InOrderThreading
```

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

6.4 树和森林

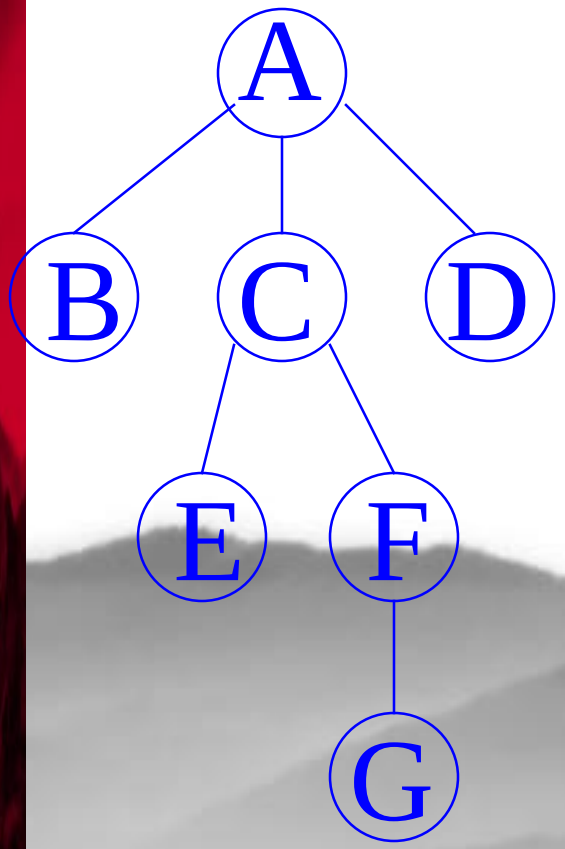




6.4.1 树的三种存储结构

一、双亲表示法
二、孩子(链表)表示法
三、孩子兄弟(二叉链表)表示法

1、双亲表示法



	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	2
5	F	2
6	G	5

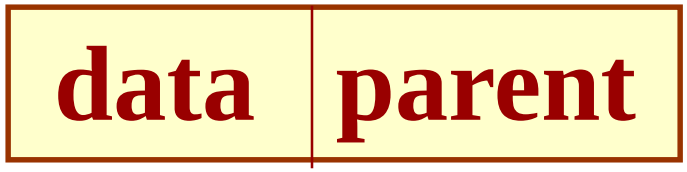
r=0

n=6

C语言的类型描述:

```
#define MAX_TREE_SIZE  100
```

结点结构:

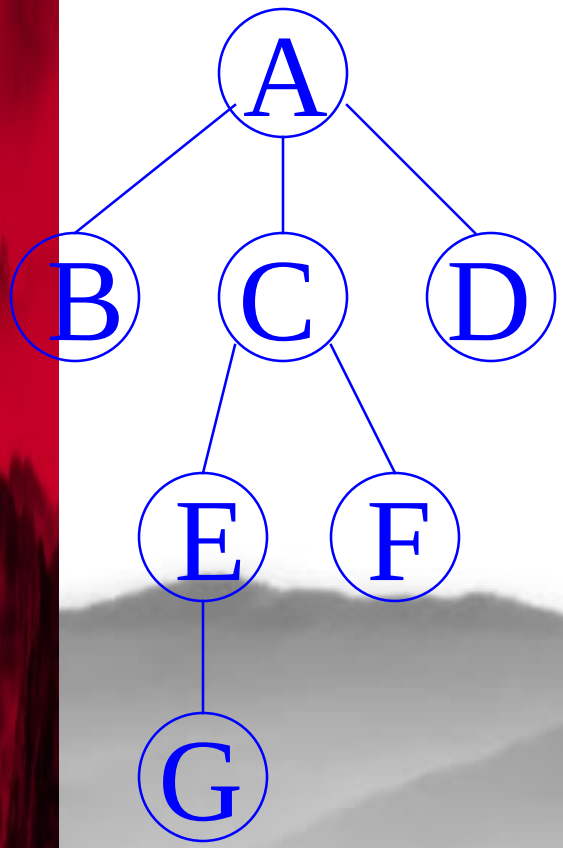



```
typedef struct PTNode {
    Elem  data;
    int    parent;   // 双亲位置域
} PTNode;
```

树结构:

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

2、孩子链表表示法



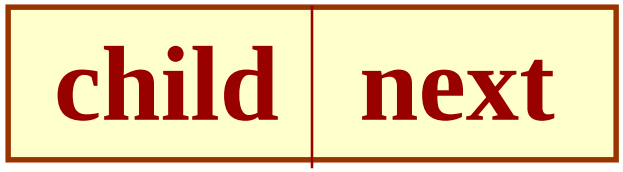
	data	firstchild	
0	A	-1	
1	B	0	∧
2	C	0	
3	D	0	∧
4	E	2	
5	F	2	∧
6	G	4	∧

r=0

n=6

C语言的类型描述:

孩子结点结构:




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

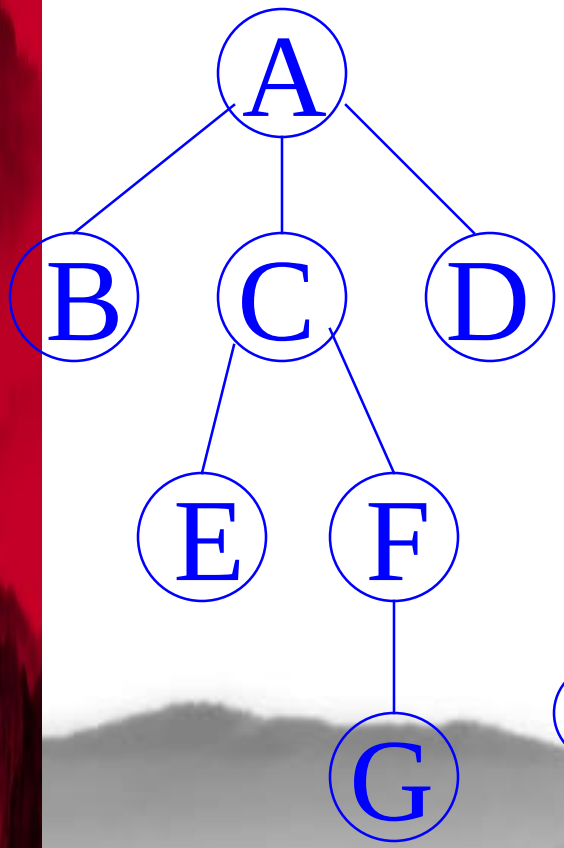
树结点结构

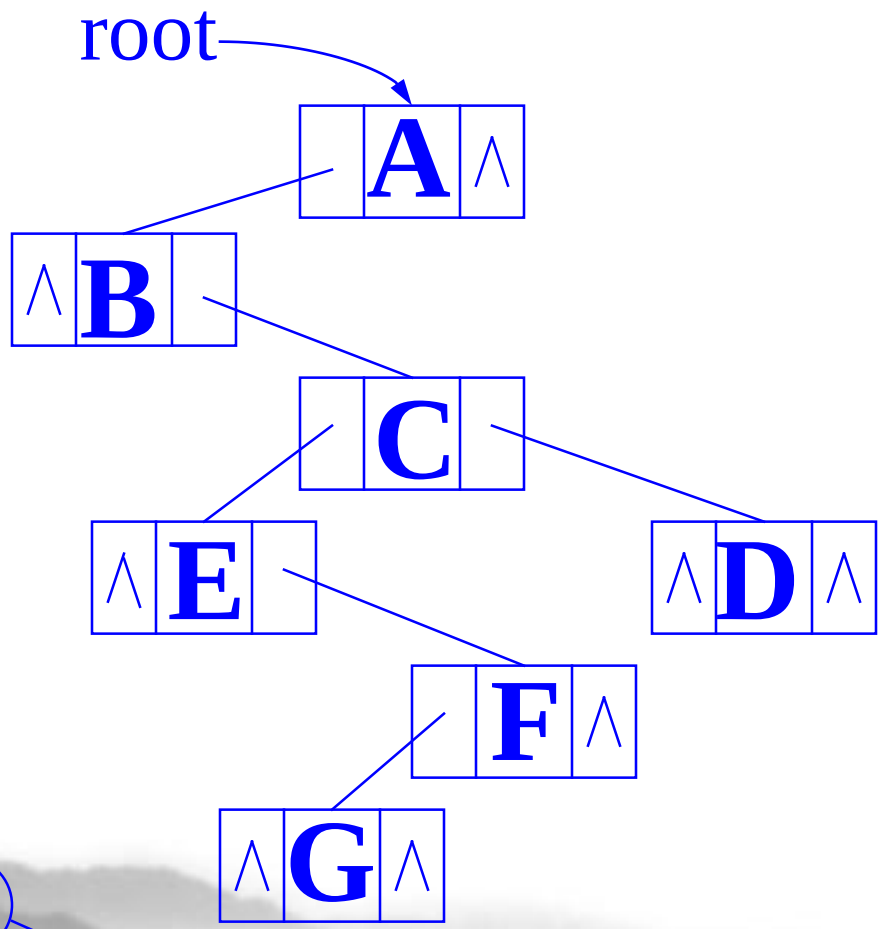

```
typedef struct {
   TElemType    data;
   ChildPtr  firstchild;
                    // 孩子链表头指针
} CTBox;
```

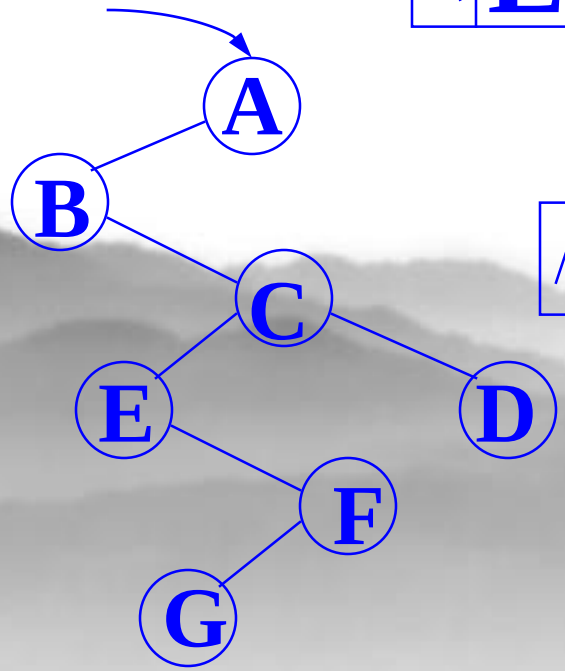
树结构:

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

3、孩子-兄弟 (二叉链表）表示法



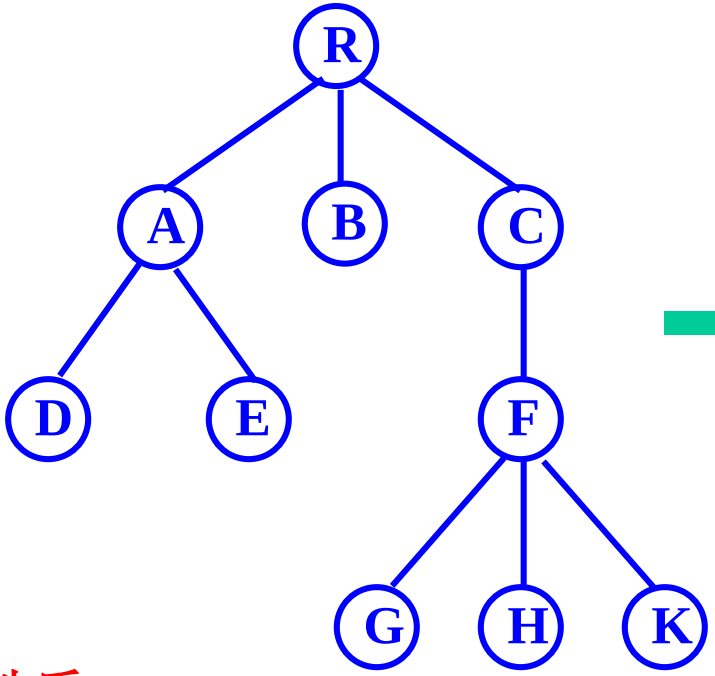


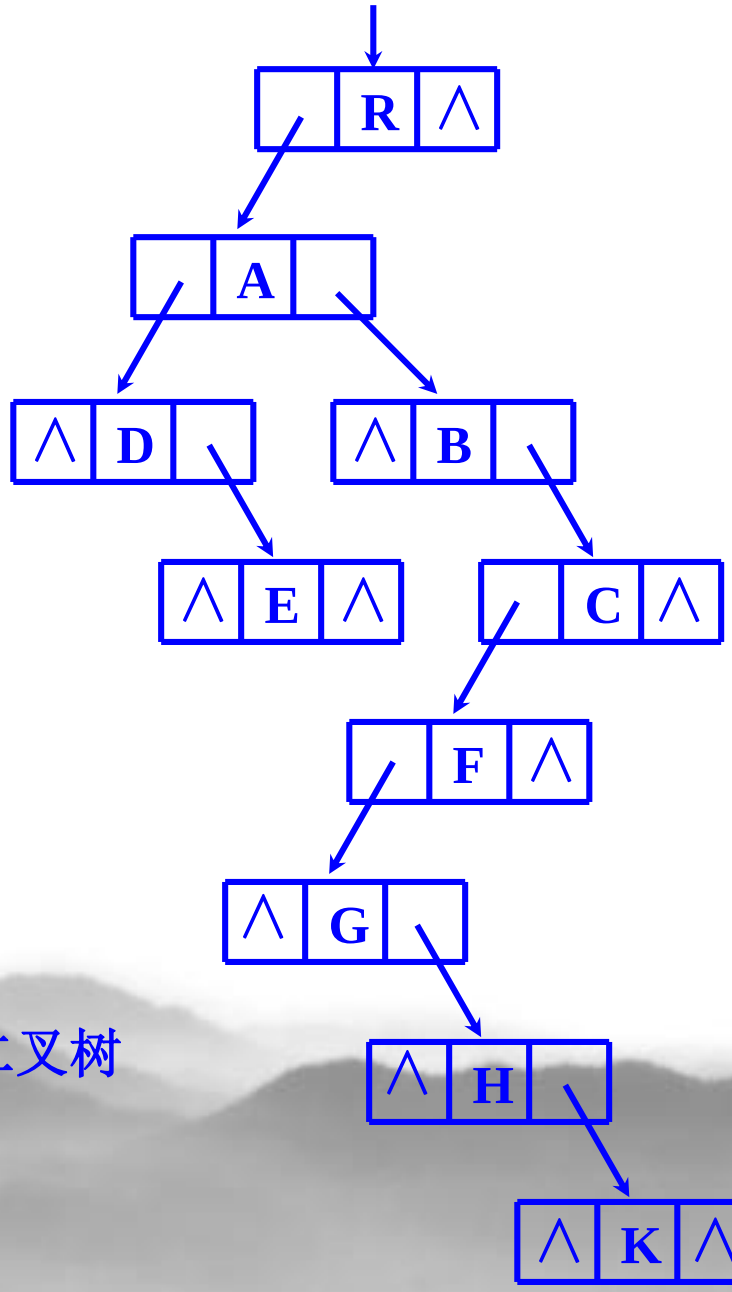


C语言的类型描述:

结点结构:


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





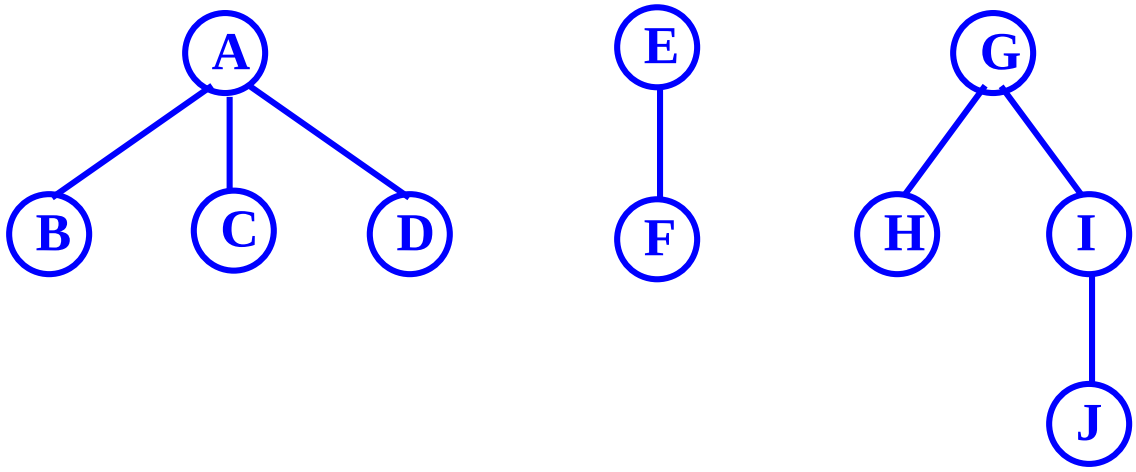
性质：

1. 树可以表示成二叉树的形式

   启示：　树与二叉树的转换

2. 树转换成一棵根只有左子树的二叉树

6.4.2 森林与二叉树的转换



(1). 任何一棵树都可以转换为一棵根没有右子树的二叉树。

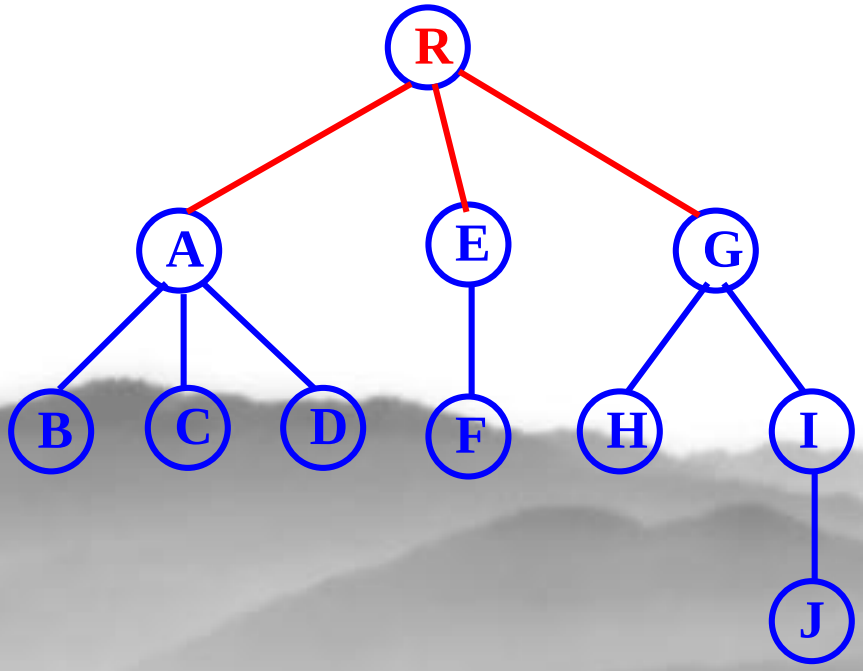
(2). 森林是由若干棵树构成的集合，若把森林中前一棵树的根结点看成是后一棵树的根结点兄弟，就可以导出森林与二叉树的转换。

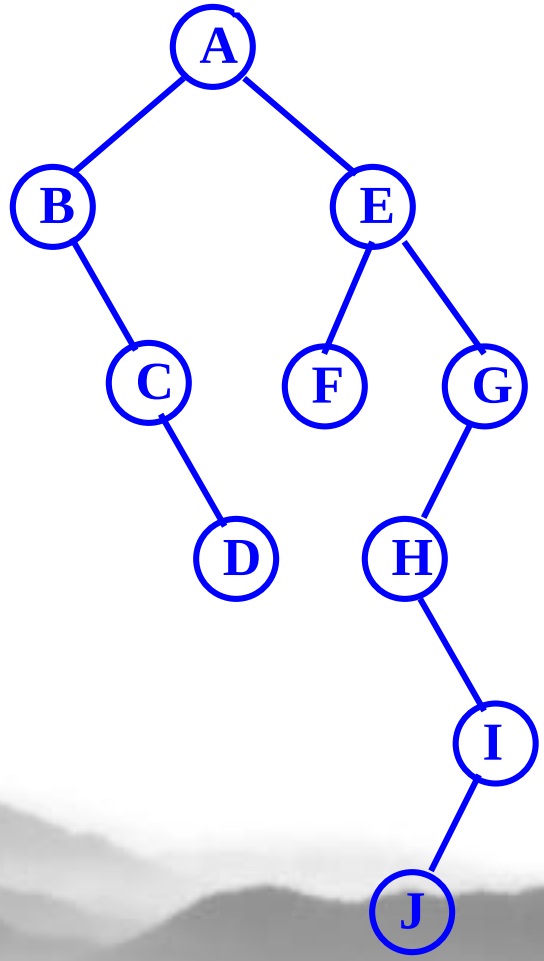
1. 森林转换成二叉树

(1) 增加一个根结点，作为原森林中各树根结点的父结点。

(2) 将新树转换成二叉树。

(3) 删除二叉树的根结点。





6.6 树的应用

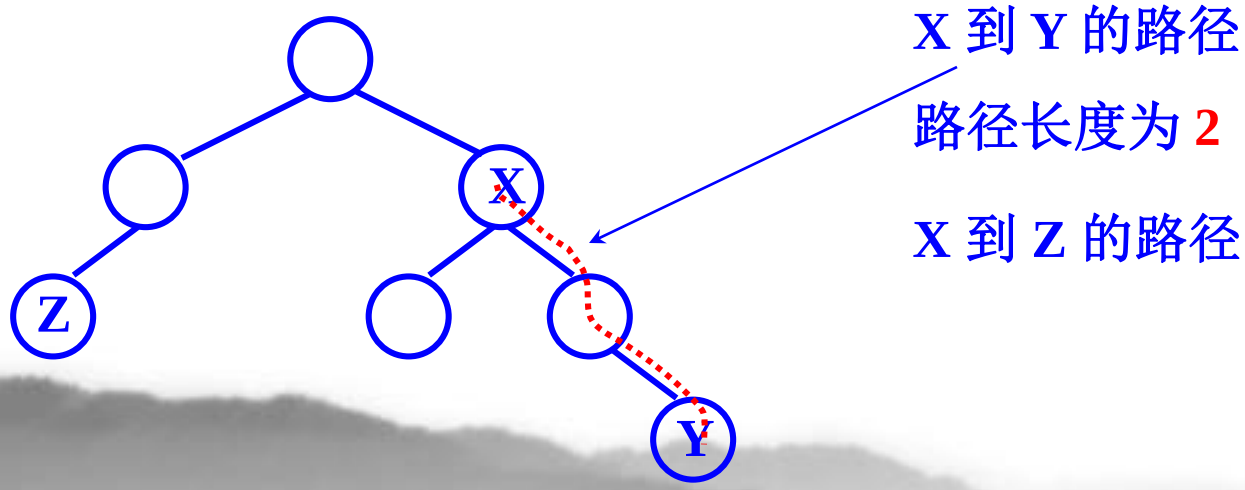
1. 赫夫曼树(最优二叉树)
2. 赫夫曼编码
3. 二叉分类树(二叉排序树)
4. 判定树

6.6.1 赫夫曼树 Huffman (最优二叉树)

基本概念:

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之间的路径。

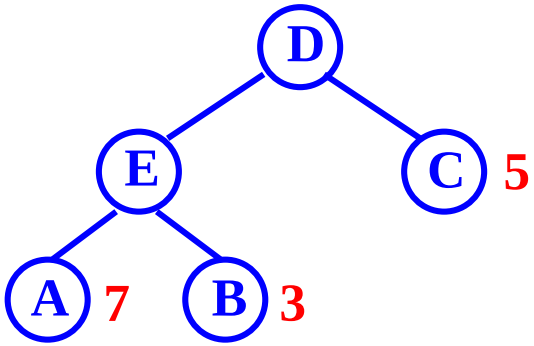
路径上的分支数目称做路径长度。



树的路径长度是从树根到每一个结点的路径长度之和。

在具有相同结点数的所有二叉树中，完全二叉树 的路径长度是最短的。

推广，为结点加权 w 。



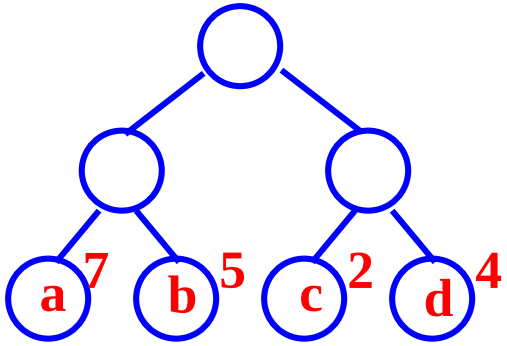
结点的带权路径长度为从根结点到该结点之间的路径长度与结点上权值的乘积。

树的带权路径长度为树中所有叶子结点的带权路径长度之和，通常记做 $WPL = \sum_{k=1}^{n} w_k \cdot l_k$

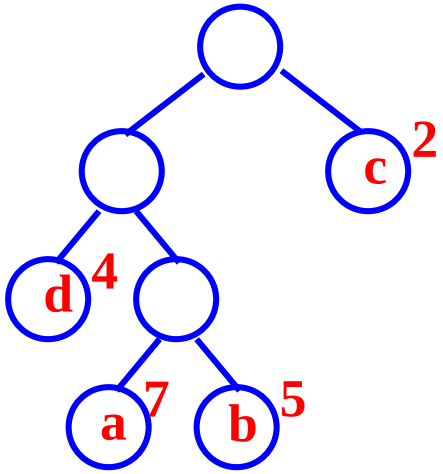
w_k 为叶子结点 v_k 的权值

l_k 为叶子结点 v_k 的路径长度

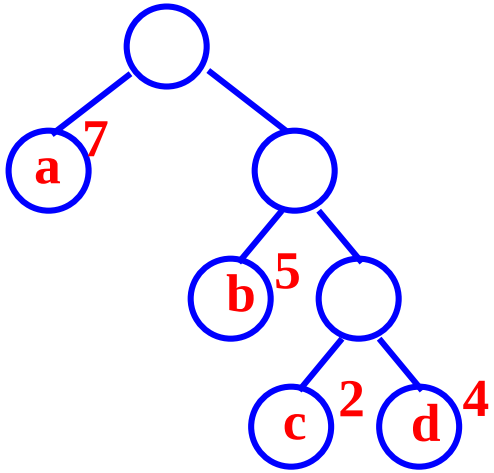
例：3 棵二叉树，都有 4 个叶子结点 a、b、c、d，分别带权7、5、2、4，求它们各自的带权路径长度。



(1)



(2)



(3)

(1) WPL = 7×2 + 5×2 + 2×2 + 4×2 = 36

(2) WPL = 7×3 + 5×3 + 2×1 + 4×2 = 46

(3) WPL = 7×1 + 5×2 + 2×3 + 4×3 = 35

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，每个叶子结点带权为 w_i，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。

如何构造赫夫曼树？

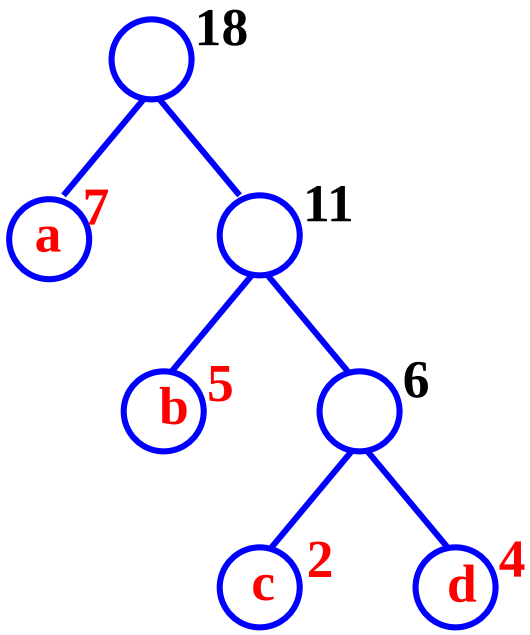
(1) 根据给定的 n 个权值 $\{w_1,w_2,\ldots w_n\}$ 构成 n 棵二叉树的集合 $F=\{T_1,T_2,\ldots T_n\}$，其中每棵二叉树 T_i 中只有一个权值为 w_i 的根结点。

(2) 在 F 中选取两棵根结点权值最小的树作为左、右子树构造一棵新的二叉树，且置新二叉树的根结点的权值为其左、右子树根结点的权值之和。

(3) 在 F 中删除这两棵树，同时将新得到的二叉树加入集合 F 中。

(4) 重复 (2) 和 (3)，直到 F 中只含一棵树为止。

例， 4 个叶子结点 a、b、c、d，分别带权7、5、2、4。



WPL = 7×1 + 5×2 + 2×3 + 4×3 = 35

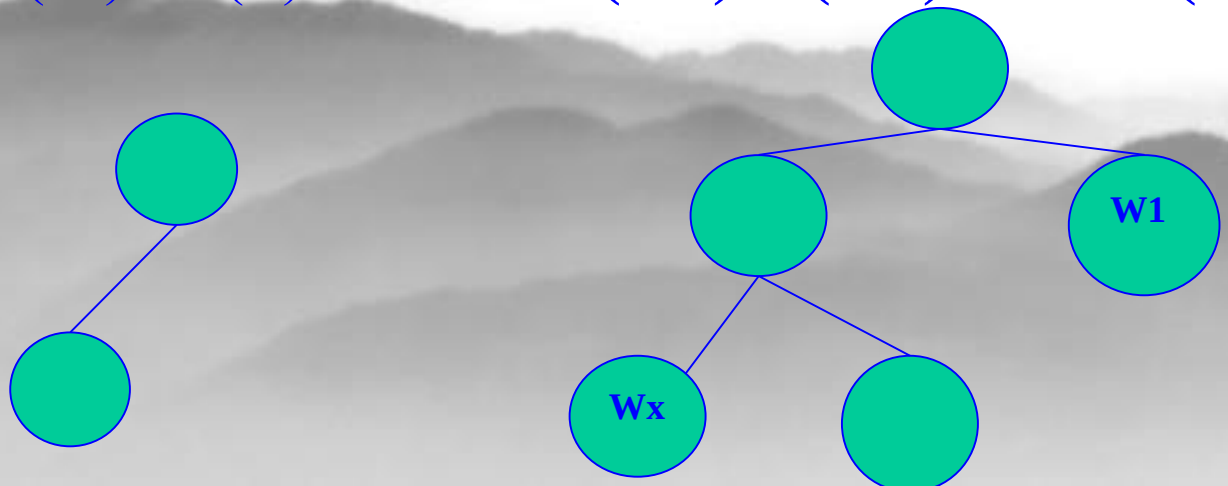
思考

- 为什么赫夫曼树是最优的？

- （1）最小的两个一定放在最下面，而且是兄弟
- （2）最优树删除最小的两个，父节点充当叶子，值为两个之和，则依旧是最优树

最小的两个一定放在最下面，而且是兄弟

- 令T：W1<=W2<=…<=Wt，设最长路径上两个叶节点分别为Wx, Wy, 则有：
- L(Wx)>=L(w1), L(Wy)>=L(w2),
- 若L(Wx)>L(W1)，将Wx和W1对调，得到T’
- W(T’)-W(T)=L(Wx)W1+L(w1)Wx-(L(w1)W1 + L(Wx)Wx)
- =(L(W1)-L(Wx))(Wx-W1)<0
- W(T’)<W(T)矛盾，则L(Wx)=L(W1)，同理L(Wy)=L(W2)

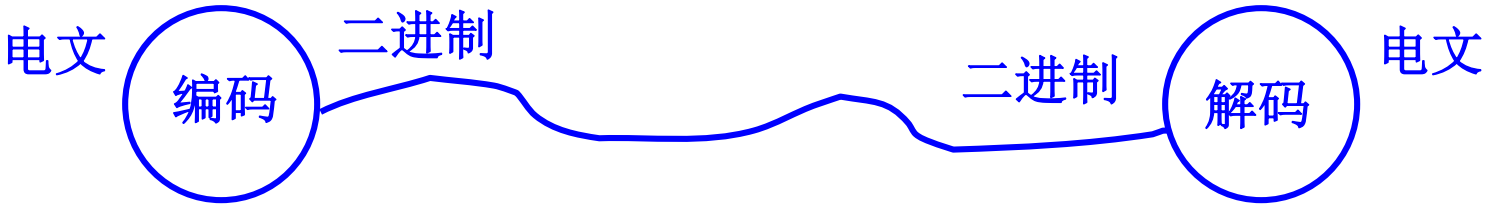


最优树删除最小的两个，父节点充当叶子，值为两个之和，则依旧是最优树

- 反证法
- 假设T是一棵最优树，W(T)是权值，最小的AB在最下层，对应的路径权值分别是Wa, Wb
- 删除AB后，父节点C成为叶子节点，此时的新树T*， W(T)=W(T*)+Wa+Wb
- 如果T*不是最优树，则必有最优树T11（包含C），W(T11)<=W(T*)
- 如果把T11中的C展开为A和B，则W(T1)=W(T11)+Wa+Wb<=W(T*) 矛盾

6.6.2 赫夫曼编码 [编码-前缀编码-赫夫曼编码]

1. 编码



例，传送 ABACCD，四种字符，可以分别编码为 00,01,10,11。

则原电文转换为 00 01 00 10 10 11。

对方接收后，采用二位一分进行译码。

当然，为电文编码时，总是希望总长越短越好，

如果对每个字符设计长度不等的编码，且让电文中出现次数较多的字符采用较短的编码，则可以减短电文的总长。

例，对 **ABACCD** 重新编码，分别编码为 **0 , 00 , 1 , 01**。

A B C D

则原电文转换为 **0 00 0 1 1 01**。 减短了。

问题： 如何译码？

前四个二进制字符就可以多种译法。

AAAA **BB**

2. 前缀编码

若设计的长短不等的编码，满足任一个编码都不是另一个编码的前缀，则这样的编码称为前缀编码。

例， A , B , C , D 前缀编码可以为 0 , 110 , 10 , 111

利用二叉树设计二进制前缀编码。

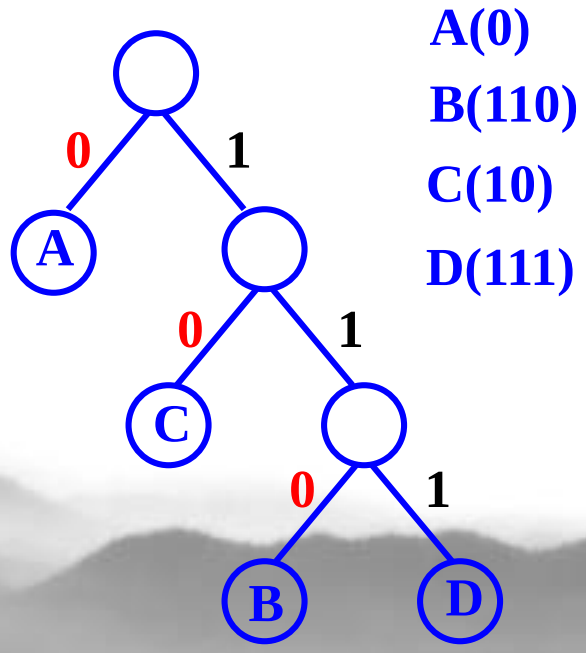
叶子结点表示 A , B , C , D 这 4 个字符

左分支表示 ‘0’，右分支表示 ‘1’

从根结点到叶子结点的路径上经过的二进制符号串作为该叶子结点字符的编码

路径长度为编码长度

证明其必为前缀编码



如何得到最短的二进制前缀编码？

3. 赫夫曼编码

设每种字符在电文中出现的概率 w_i 为，则依此 n 个字符出现的概率做权，可以设计一棵赫夫曼树，使

$$WPL = \sum_{i=1}^{n} w_i \ l_i \quad \text{最小}$$

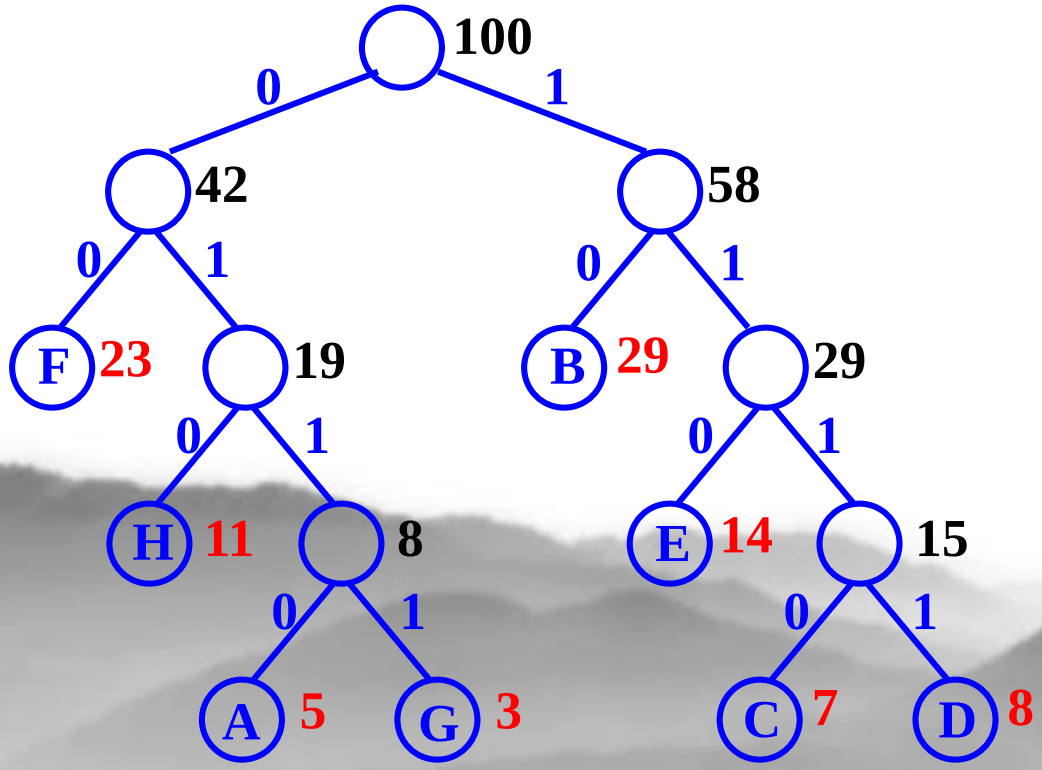
w_i 为叶子结点的出现概率 (权)

l_i 为根结点到叶子结点的路径长度

例，某通信可能出现 A B C D E F G H 8 个字符，其概率分别为 0.05 , 0.29 , 0.07 , 0.08 , 0.14 , 0.23 , 0.03 , 0.11 ， 试设计赫夫曼编码

不妨设 w = { 5 , 29 , 7 , 8 , 14 , 23 , 3 , 11 }

排序后 w = { 100 }



A (0110)

B (10)

C (1110)

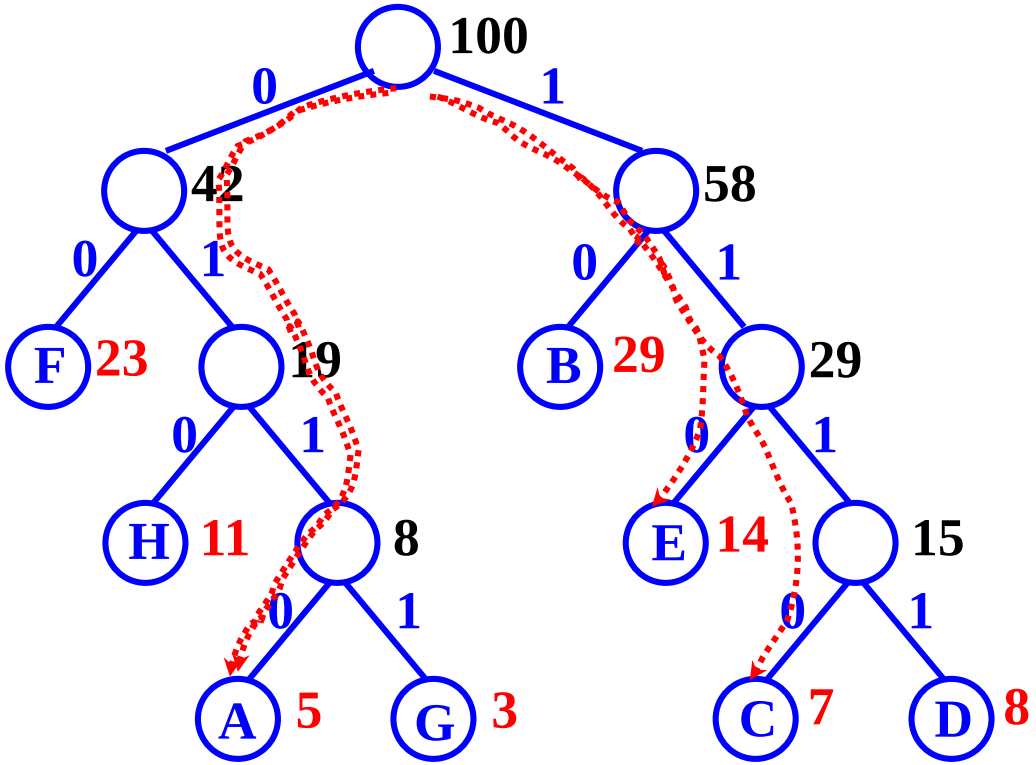
D (1111)

E (110)

F (00)

G (0111)

H (010)



A (0110)

B (10)

C (1110)

D (1111)

E (110)

F (00)

G (0111)

H (010)

ACEA 编码为 0110 1110 110 0110

A C E A

如何译码？

1. 从根结点出发，从左至右扫描编码，
2. 若为 ‘0’ 则走左分支，若为 ‘1’ 则走右分支，直至叶结点为止，
3. 取叶结点字符为译码结果，返回重复执行 1,2,3 直至全部译完为止

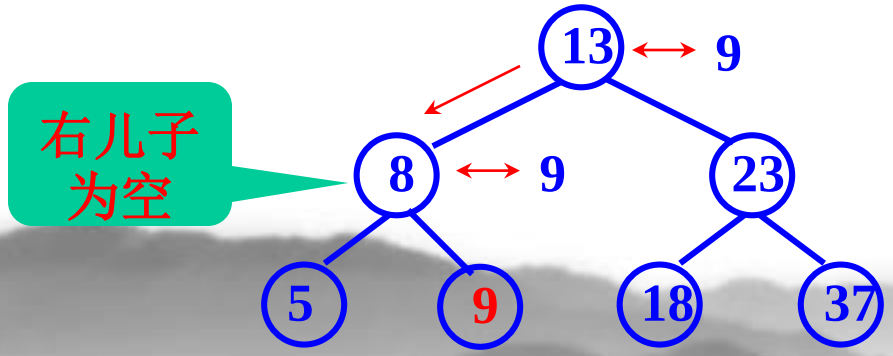
6.6.3 二叉分类树(二叉排序树)

二叉分类树或者是一棵空树；或者是具有下列性质的二叉树：

(1) 左子树上所有结点的值均小于等于它的根结点的值；

(2) 右子树上所有结点的值均大于它的根结点的值；

(3) 根结点的左、右子树也分别为二叉分类树。

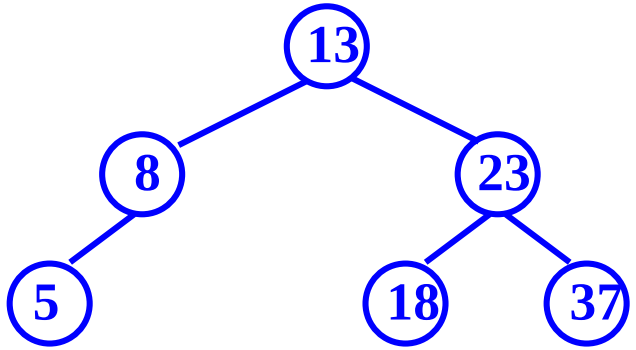


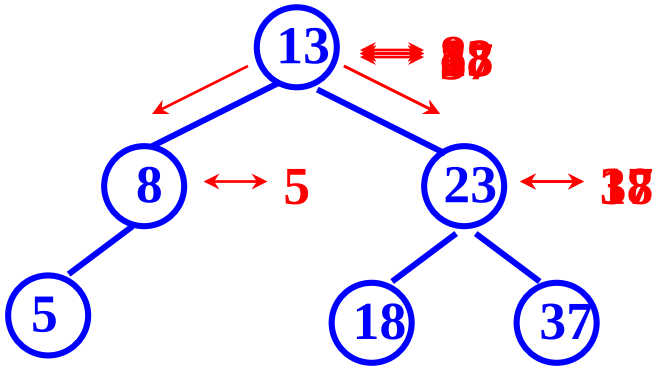
如何插入新结点 9？

利用插入操作可以构造一棵二叉分类树

首先给出结点序列:

13、8、23、5、18、37





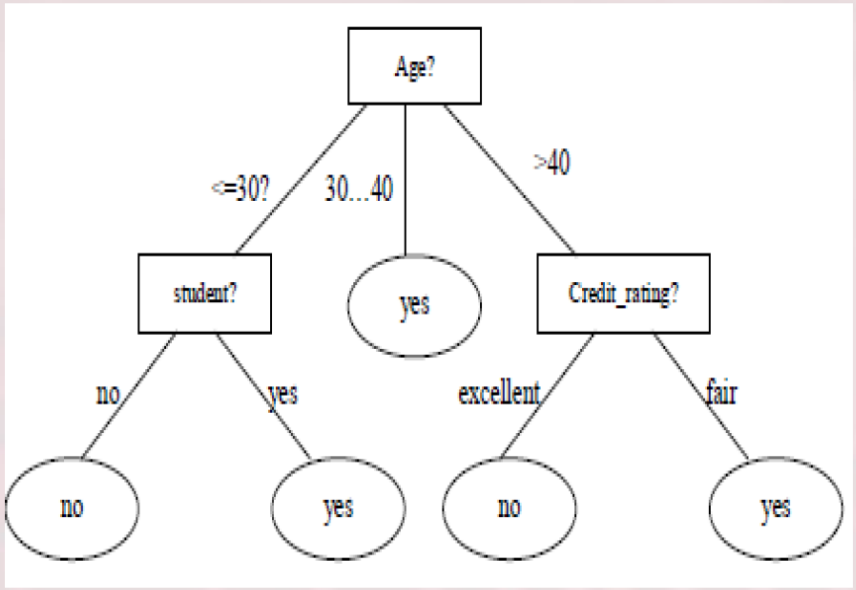
分类二叉树的应用:

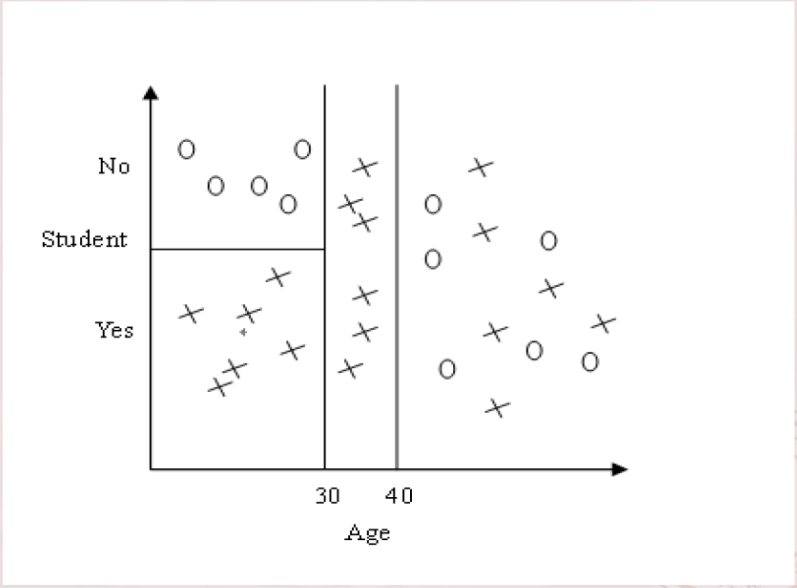
快速、方便查找某个结点

实现二叉分类树的插入算法!

决策树分类

决策树分类的主要任务是要确定各个类别的决策区域，或者说，确定不同类别之间的边界。在决策树分类模型中，不同类别之间的边界通过一个树状结构来表示





决策树算法

- 最大高度 = 决策属性的个数
- 树越矮越好
- 要把重要的好的属性放在树根
- 因此，决策树建树算法就是： 选择树根的过程

决策树算法

1 开始时，所有的训练集样本都在树根

2 属性都是可分类的属性(如果是连续值的话，先要对其进行离散化)

停止划分的条件：

1 某个节点上的所有样本都属于相同的类别

2 所有属性都用到了– 采用多数有效法对叶子节点分类

3 没有样本了

决策树分类第一步：选择树根

- 比较流行的属性选择法：信息增益
- 信息增益最大的属性被认为是最好的树根

- H：信息熵 $$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i).$$
- C：样本（分类）集合，T：属性
- IG(T)=H(C)-H(C|T)。

示例

共有5个属性

前4个属性用作预测属性，最后一个属性是类别属性

共有14个样本，或者说14条记录

age	income	student	credit_rating	buys_computer
<=30	high	no	fair	no
<=30	high	no	excellent	no
31…40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31…40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31…40	medium	no	excellent	yes
31…40	high	yes	fair	yes
>40	medium	no	excellent	no

H(C)

$$I(p,n) = I(\frac{9}{14},\frac{5}{14}) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940$$

- Class N: buys_computer = "no"
- I(p, n) = I(9, 5) =0.940
- Compute the entropy for *age*:

age	p_i	n_i	$I(p_i, n_i)$
<=30	2	3	0.971
30…40	4	0	0
>40	3	2	0.971

age	income	student	credit_rating	buys_computer
<=30	high	no	fair	no
<=30	high	no	excellent	no
31…40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31…40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31…40	medium	no	excellent	yes
31…40	high	yes	fair	yes
>40	medium	no	excellent	no

$$E(age) = \frac{5}{14}I(2,3) + \frac{4}{14}I(4,0) + \frac{5}{14}I(3,2) = 0.694$$

$\frac{5}{14}I(2,3)$ means "age <=30" has 5 out of 14 samples, with 2 yes'es and 3 no's. Hence

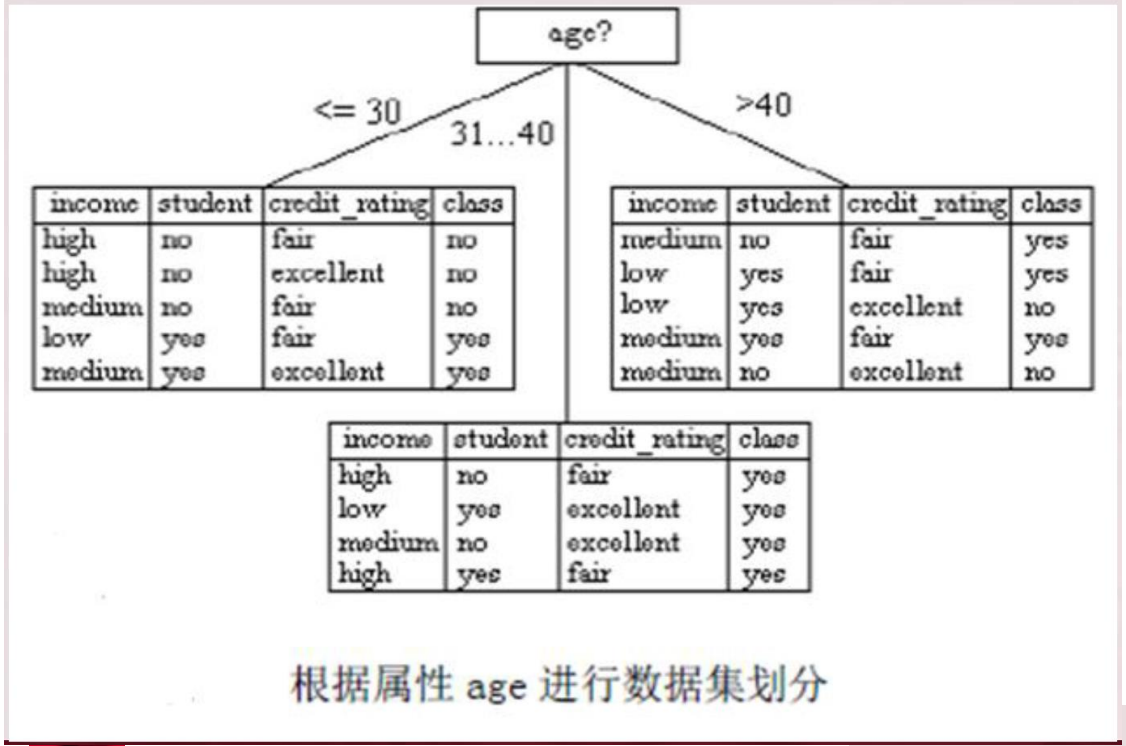
$$Gain(age) = I(p,n) - E(age) = 0.246$$

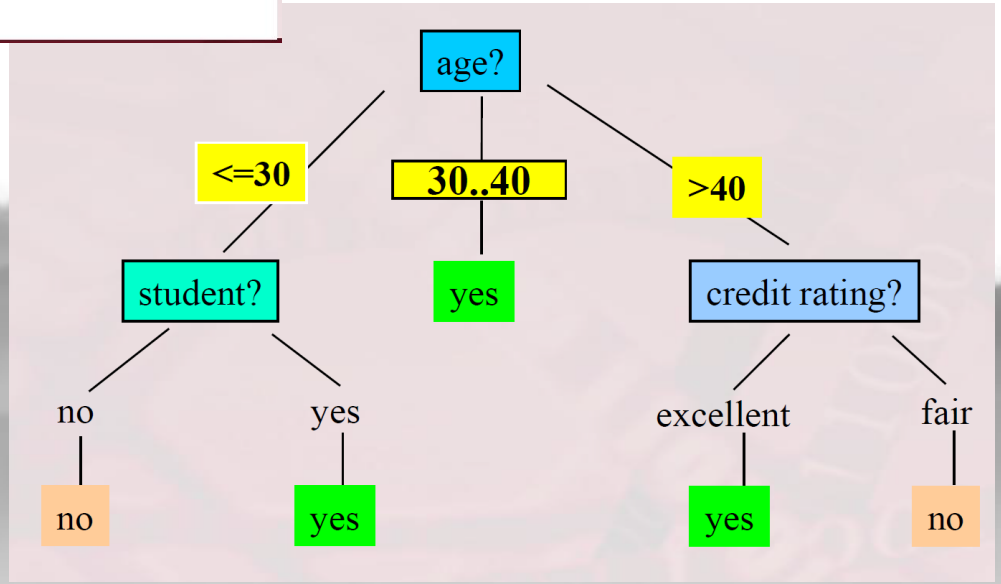
Similarly,

$$Gain(income) = 0.029$$

$$Gain(student) = 0.151$$

$$Gain(credit\_rating) = 0.048$$


age?
<= 30
31…40
>40
income student credit_rating class
high no fair no
high no excellent no
medium no fair no
low yes fair yes
medium yes excellent yes
income student credit_rating class
medium no fair yes
low yes fair yes
low yes excellent no
medium yes fair yes
medium no excellent no
income student credit_rating class
high no fair yes
low yes excellent yes
medium no excellent yes
high yes fair yes
根据属性 age 进行数据集划分


age?
<=30
30..40
>40
student?
yes
credit rating?
no
yes
excellent
fair
no
yes
yes
no

练习

- 深度为5的二叉树至多有(　　)个结点。

  A．16　　B．32

  C．31　　D．10

- 具有10个叶子结点的二叉树中有____个度为2的结点。

A．8　　B．9　　C．10　　D．11

- 将一棵有多个结点的完全二叉树从根这一层开始，每一层上从左到右依次对结点进行编号，根结点的编号为1，则编号为49的结点的左孩子编号为(　　)。

  A．98　　B．99　　C．50　　D．48

- 按照二叉树的定义，具有3个结点的二叉树有(　　)种形态。

  A．3　　B．4

  C．5　　D．6

- 某二叉树的先序序列和后序序列正好相反，则该二叉树一定是（　　）的二叉树。

  A．空或只有一个结点

  B．高度等于其结点数

  C．任一结点无左孩子

  D．任一结点无右孩子

- 假定一棵二叉树的结点个数为50，则它的最小深度为_____，最大深度为_____。
- 一棵树的后根序列与其转换的二叉树的___ 序列相同,先根序列与其转换的二叉树的_____ 序列相同。
- 具有400个结点的完全二叉树的深度为_____。
- 假定一棵二叉树的结点数为18，则它的最小深度为_____，最大深度为_____。

- 已知二叉树的后序和中序序列如下，画出该二叉树。

  后序序列：DEABFCR

  中序序列：DAERBCF

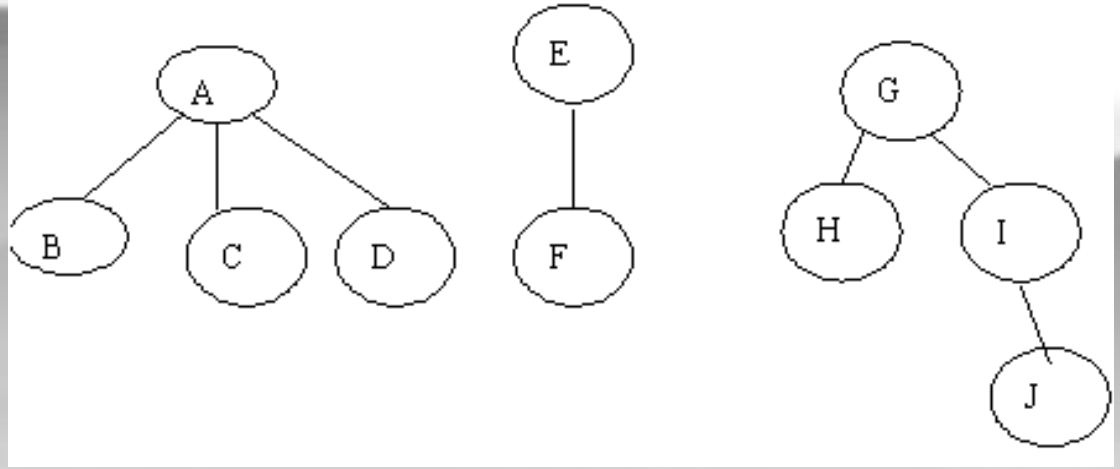
- 已知二叉树的后序和中序序列如下，画出该二叉树。

  后序序列：ABCDEFG

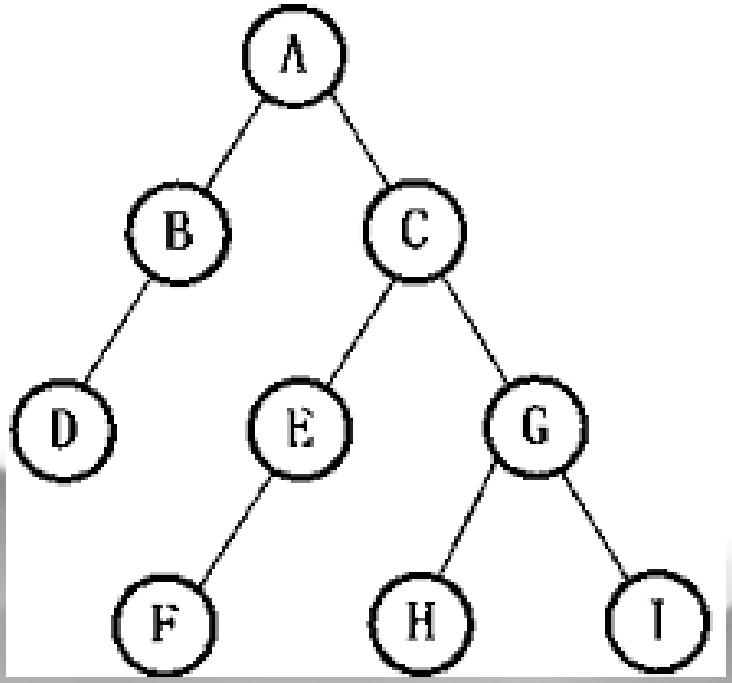
  中序序列：ACBGEDF

- 有7个带权结点，其权值分别为3，7，8，2，6，10，14，试以它们为叶子结点生成一棵哈夫曼树，画出相应的哈夫曼树(左子树根结点的权小于等于右子树根结点的权)。

- 已知如下树林，画出对应的二叉树。



⌘ 已知二叉树，画出中序的线索。



- 有一份电文中共使用五个字符：a、b、c、d、e,它们的出现频率依次为8、14、10、4、18，请构造相应的哈夫曼树(左子树根结点的权小于等于右子树根结点的权)，求出每个字符的哈夫曼编码。

- 已知二叉树以二叉链表作为存储结构，阅读下列算法，说出它的功能，k为全局变量，初值为 0，首次调用时i值为0。

```
void unknown(struct node *t,  int i)
{
      if (t!=NULL)
      {
           cout<< t->data;   /* 访问根结点 */
             i++;
             if(k<i)  k=i;
             unknown(t->lch, i);    /* 先根遍历左子树 */
             unknown(t->rch, i);     /* 先根遍历右子树 */

      }
} /* unknown  */
```

以下为中根次序线索化二叉树的遍历算法，在空缺上填写具体语句，使之成为一个完整的算法。

```
struct nodex
{
   char data;
   struct nodex *lch, *rch;
   int ltag, rtag;   /* 左、 右标志域，有孩子时标志域为0，为线索时值为1 */
 };
struct nodex    *insucc(struct nodex *q)    //计算结点q的后继
{
    if(q->rtag==1) _____(1)_____
    else
    {
          r=q->rch;
          while (r->ltag!=1)  ____(2)_____
          p=r;
     }
  return(p);
}
```

```
void inthorder(stuct nodex  *t)
{
      p=t;
      if(p! =NULL)
            while(p->lch! =NULL)
                  p=p->lch;  /* 查找二叉树的最左结点 */
      printf( p->data);
      while(p->rch!=NULL)
      {
            p=insucc(p);   /* 调用求某结点p后继的算法 */
             printf(p->data);
      }
 } /* inthorder */
```