# 树和二叉树的相关概念、术语

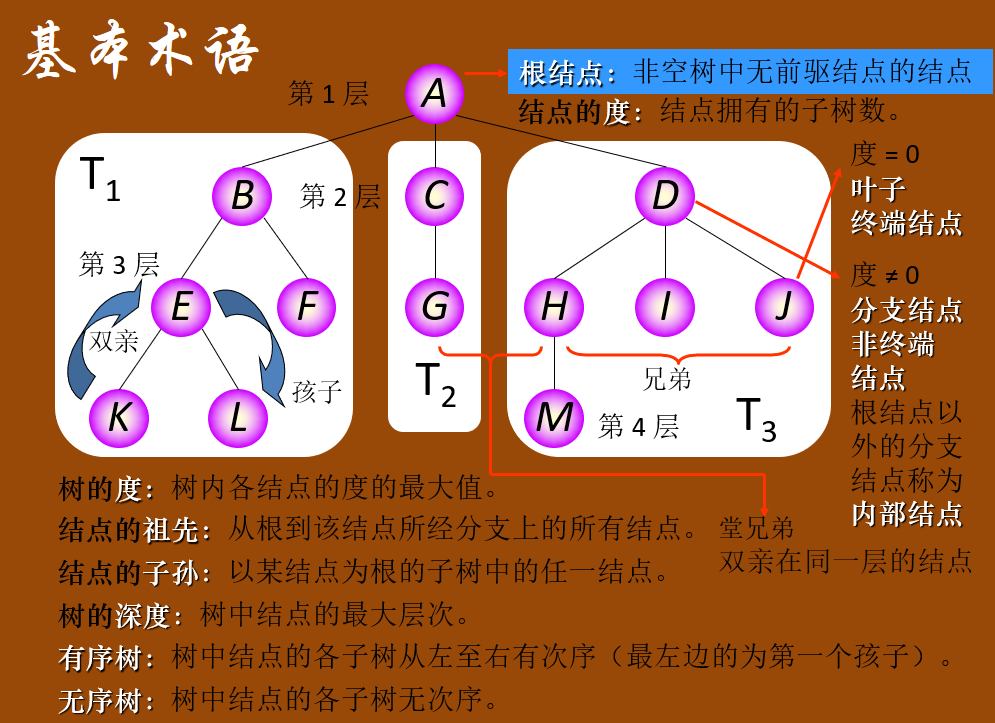
## 树的定义

树(Tree)是 *n* (*n*≥0)个结点的有限集。若 *n*=0，称为空树；若 *n* > 0，则它满足如下两个条件：

(1) 有且仅有一个特定的称为根 (Root) 的结点；

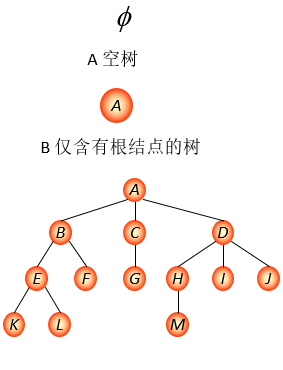
(2) 其余结点可分为 *m* (*m*≥0) 个互不相交的有限集 *T*1, *T*2, *T*3, …, *Tm*， 其中每一个集合本身又是一棵树，并称为根的子树 (SubTree)。

## 树的基本术语

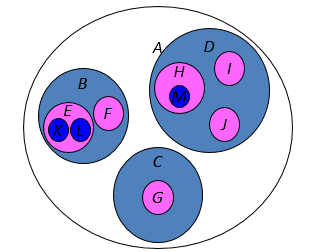


## 树的表示形式

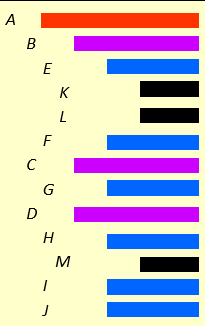
1．树形表示法



2．嵌套集合（文氏）表示法



3．凹入表示法



4．广义表表示法

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

# 二叉树

## 定义

二叉树是 *n* (*n*≥0) 个结点的有限集，它或者是**空集 (*n* = 0)**，或者由一个根结点及两棵**互不相交**的分别称作这个根的**左子树**和**右子树**的**二叉树**组成。

**特点：**

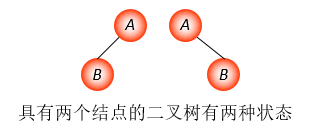
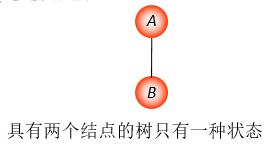
**1、每个结点最多有俩孩子 (二叉树中不存在度大于 2 的结点) 。**

**2、子树有左右之分，其次序不能颠倒。**

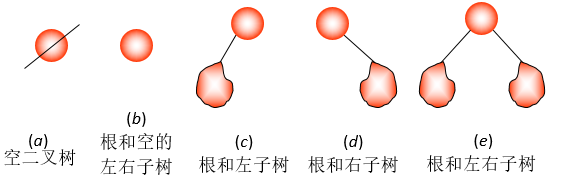
**3、二叉树可以是空集合，根可以有空的左子树或空的右子树。**

**二叉树不是树的特殊情况，它们是两个概念。**

二叉树结点的子树要区分左子树和右子树，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。树当结点只有一个孩子时，就无须区分它是左还是右。（也就是二叉树每个结点位置或者说次序都是固定的，可以是空，但是不可以说它没有位置，而树的结点位置是相对于别的结点来说的，没有别的结点时，它就无所谓左右了），因此二者是不同的。**这是二叉树与树的最主要的差别。**

## 二叉树的 5 种基本形态



注：虽然二叉树与树概念不同，但有关树的基本术语对二叉树都适用。

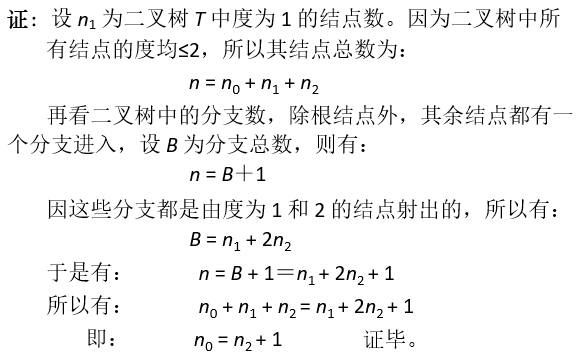
# 二叉树的五个性质

**性质 1：** 在二叉树的第 *i* 层上至多有 2*i* - 1 个结点 (*i* ≥1)。

**性质 2：**深度为 *k* 的二叉树至多有 2*k*－1 个结点（*k* ≥1)。



**性质 3：**对任何一棵二叉树 *T*，如果其叶子数为 *n*0，度为 2 的结点数为 *n*2，则 *n*0 = *n*2 + 1。

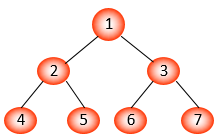


**满二叉树 (Full binary tree)**

一棵深度为 *k* 且有 2*k*- 1 个结点的二叉树称为满二叉树。

**特点：（1）**每一层上的结点数都达到最大。**（2）叶子全部在最底层**。

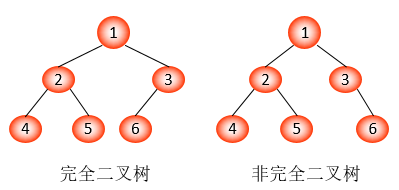
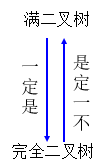
**编号规则：**从根结点开始，自上而下，自左而右。



**完全二叉树 (Complete binary tree)**

深度为 *k* 的具有 *n* 个结点的二叉树，当且仅当其每一个结点都与深度为 *k* 的满二叉树中编号为 1~ *n* 的结点一一对应时，称之为**完全二叉树**。

特点：（1）**叶子只可能分布在层次最大的两层上。** （2）**对任一结点，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 ⎣log2*n*⎦ + 1。

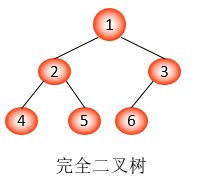
**性质 5：** 如果对一棵有 *n* 个结点的完全二叉树 (深度为⎣log2*n*⎦+1) 的结点按层序编号 (从第 1 层到第⎣log2*n*⎦+1 层，每层从左到右)，则对任一结点 *i* (1≤*i*≤*n*)，有：

(1) 如果 *i* = 1，则结点 *i* 是二叉树的根，无双亲；

如果 *i* >1，则其双亲是结点 ⎣*i* / 2⎦。

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

(3) 如果 2*i* + 1 > *n*，则结点 *i* 无右孩子；否则，其右孩子是结点 2*i* + 1。

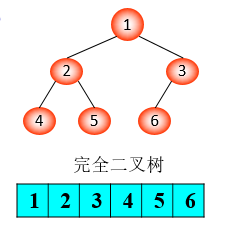


# 二叉树存储、遍历、线索化

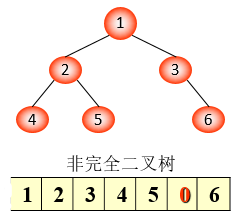
## 二叉树的存储结构

**顺序存储结构**

**完全二叉树：**用一组地址连续的存储单元依次自上而下、自左至右存储结点元素，即将编号为 *i* 的结点元素存储在一维数组中下标为 *i* –1 的分量中。

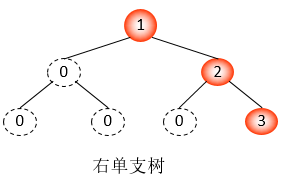


**一般二叉树：**将其每个结点与完全二叉树上的结点相对照，存储在一维数组的相应分量中。



**此顺序存储结构仅适用于完全二叉树**

**最坏情况：**深度为 *k* 的且只有 *k* 个结点的右单支树需要长度为2*k*-1 的一维数组。



//表示方式：

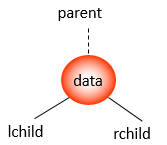
#define MAX\_TREE\_SIZE 100 // 二叉树的最大结点数

typedef TElemType SqBiTree[MAX\_TREE\_SIZE]; // 0 号单元存储根结点

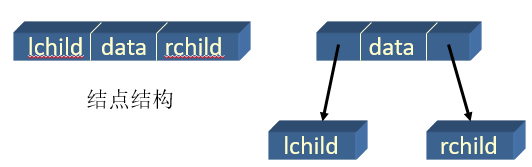
SqBiTree bt;

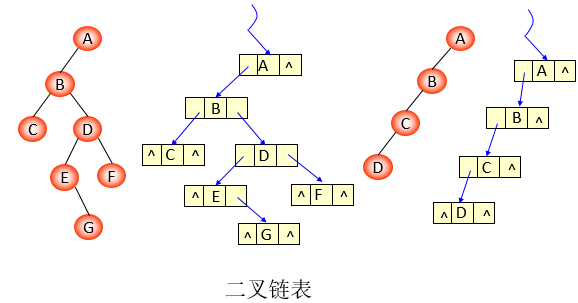
**链式存储结构**

**二叉树结点的构成**



**存储方式**





**在 *n* 个结点的二叉链表中有 *n* + 1 个空指针域。**

**C 语言的类型描述如下：**

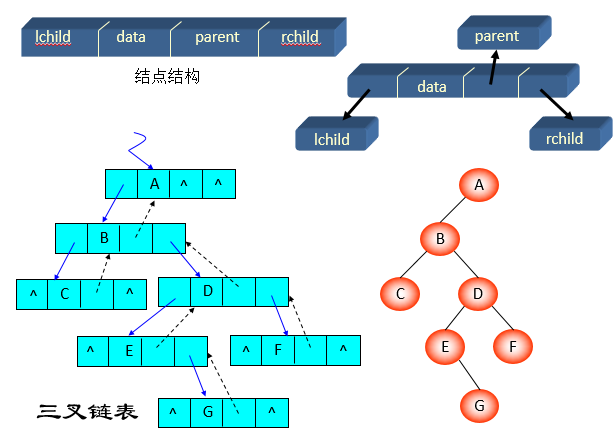
//表示方式：

typedef struct BiTNode { // 结点结构

TElemType data;

struct BiTNode \*lchild, \*rchild; // 左右孩子指针

} BiTNode, \*BiTree;



## 遍历二叉树

**遍历概念：**

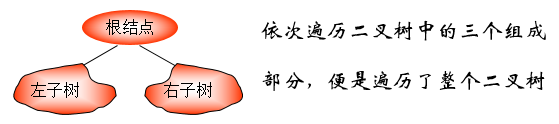
顺着某一条**搜索路径**巡访二叉树中的结点，使得每个结点均被**访问**一次，而且仅被访问一次。

“访问”的含义很广，可以是对**结点作各种处理**， 如：输出结点的信息、修改结点的数据值等，但要求这种**访问不破坏原来的数据结构**。 （因此访问是引用型的操作）

**遍历目的：**

得到树中所有结点的一个**线性**排列。

**遍历方法：**



### 先序遍历二叉树（前序遍历二叉树）

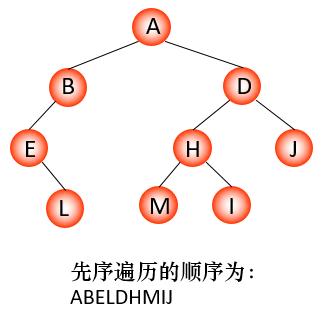
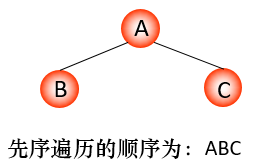
先序遍历二叉树的操作定义：

若二叉树为空，则**空操作**；否则

(1) 访问根结点；

(2) 先序遍历左子树；

(3) 先序遍历右子树。



**先序遍历二叉树基本操作的递归算法在二叉链表上的实现：**

Status PreOrderTraverse (Bitree T, Visit)

{ // 最简单的 Visit 函数是：

// Status PrintElement (TElemType e)

// { Printf (e); // 实用时加上格式串。

// return OK; }

// 调用实例：PreOrderTraverse (T, PrintElement);

if (T)

{ if (Visit (T->data))

if (PreOrderTraverse (T->lchild, Visit))

if (PreOrderTraverse (T->rchild, Visit))

return OK;

return ERROR;

}

else return OK;

} // PreOrderTraverse

### 中序遍历二叉树

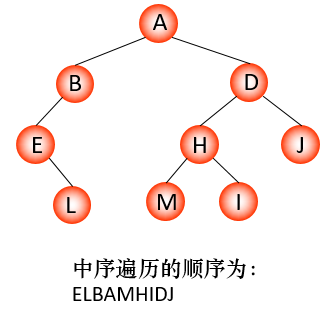
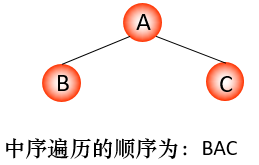
中序遍历二叉树的操作定义：

若二叉树为空，则**空操作**；否则

(1) 中序遍历左子树；

(2) 访问根结点；

(3) 中序遍历右子树。



**中序遍历二叉树基本操作的递归算法在二叉链表上的实现：**

Status InOrderTraverse (Bitree T, Visit)

{ if (T)

{ if (InOrderTraverse ( T->lchild, Visit ))

if (Visit (T->data))

if (InOrderTraverse (T->rchild, Visit))

return OK;

return ERROR;

}

else

return OK;

} // InOrderTraverse

### 后序遍历二叉树

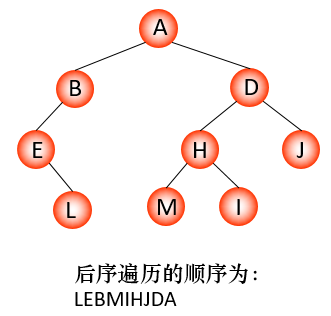
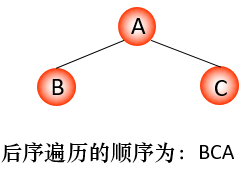
后序遍历二叉树的操作定义：

若二叉树为空，则**空操作**；否则

(1) 后序遍历左子树；

(2) 后序遍历右子树；

(3) 访问根结点。



**后序遍历二叉树基本操作的递归算法在二叉链表上的实现：**

Status PostOrderTraverse (Bitree T, Visit)

{ if (T)

{ if (PostOrderTraverse ( T->lchild, Visit ))

if (PostOrderTraverse (T->rchild, Visit))

if (Visit (T->data))

return OK;

return ERROR;

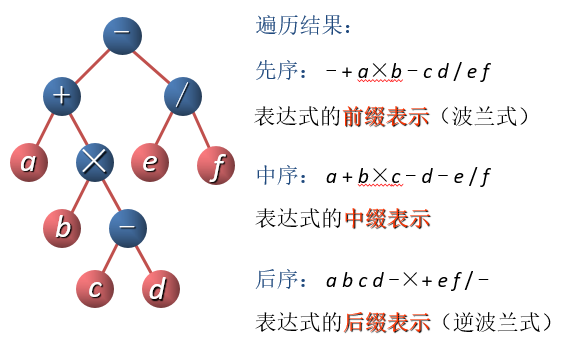
}

else

return OK;

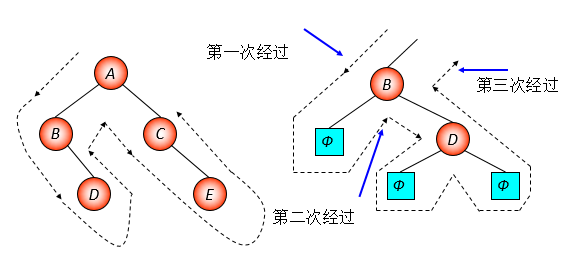
} // PostOrderTraverse

**例：请写出下图所示二叉树的先序、中序和后序遍历顺序。**



**上述式子中，中序遍历二叉树的结果是最接近原来表达式的结果的。**

下图中，第一次经过代表先序遍历二叉树，第二次经过代表中序遍历二叉树，第三次经过代表后序遍历二叉树



课堂练习

1. 二叉树的前序遍历序列中，任意一个结点均处在其子女结点的前面，

这种说法（A） （A）正确 （B）错误

1. 由于二叉树中每个结点的度最大为 2，所以二叉树是一种特殊的树，

这种说法（B） （A）正确 （B）错误

1. 已知某二叉树的后序遍历序列是 dabec。中序遍历序列是 debac，它

的前序遍历序列是（D）。 （分析法，画图法）

（A）acbed （B）decab（C）deabc （D）cedba

1. 某二叉树的前序遍历结点访问顺序是 abdgcefh，中序遍历的结点访

问顺序是 dgbaechf，则其后序遍历的结点访问顺序是（D）。

（A）bdgcefha （B）gdbecfha （C）bdgaechf （D）gdbehfca

1. 在一非空二叉树的中序遍历序列中，根右边（A）

（A）只有右子树上的所有结点 （B）只有右子树上的部分结点

（C）只有左子树上的部分结点 （D）只有左子树上的所有结点

1. 任一二叉树的叶子结点在先、中和后序遍历序列中的相对次序 (A) 。
2. 不发生改变 （B）发生改变 （C）不能确定 （D）以上都不对

### 遍历的非递归运算（中序）

**中序遍历二叉树基本操作非递归算法在二叉链表上的实现：**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*方法一\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InOrderTraverse (BiTree T, Status(\* Visit)(TElemType e))

{

InitStack(S); //初始化一个栈

Push(S, T); // 根指针进栈

while (!StackEmpty(S))

{

while (GetTop (S, p) && p)

Push (S, p->lchild); // 向左走到尽头。

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)) //将结点输出

return ERROR;

Push (S, p->rchild);

} // if

} // while

return OK;

} // InOrderTraverse

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*方法二\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InOrderTraverse(BiTree T, Status(\* Visit)(TElemType e))

{

InitStack(S);

p = T;// p指向了首元素

while (p||!StackEmpty(S))

{

if (p)

{

push(S, p);

p = p->lchild;

} // 根指针进栈，遍历左子树。

else

{

// 根指针退栈，访问根结点，遍历右子树。

Pop (S, p);

if ( !Visit(p->data))

return ERROR;

p = p->rchild;

} // else

} // while

return OK;

} // InOrderTraverse

## 二叉树其它操作算法举例

### 统计二叉树中叶子结点的个数

实现此操作只需对二叉树“遍历”一遍，并在遍历过程中对“**叶子结点计数**”即可。显然这个遍历的次序可以随意，只是为了在遍历时进行计数，需要在算法的参数中设一个“计数器”。

**问题**：能否将 count 设成函数中的局部变量，然后以count 的值作为函数值返回？ **（不能）**

**原因：**算法需要在递归执行的过程中对叶子“累加计数”

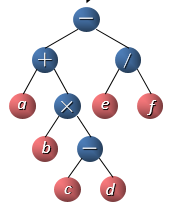
typedef struct BiTNode

{ // 结点结构

TElemType data;

struct BiTNode \*Lchild, \*Rchild; // 左右孩子指针

} BiTNode, \*BiTree;

void CountLeaf(BiTree T,int &count)

{

//先序遍历二叉树以count返回二叉树中叶子节点的数目

if(T!=NULL)

{

if((!T->Lchild)&&(!T->Rchild))

count ++;

CountLeaf(T->Lchild,count);

CountLeaf(T->Rchild,count);

}

}

### 求二叉树的深度（后序）

二叉树的深度 = MAX（左子树深度，右子树深度）+ 1 。

int height(BiTree T)

{

//后序遍历二叉树以实现求左子树的深度和右子树的深度，最后加一

int deep = 0;

int leftdeep,rightdeep;

if(T!=NULL)

{

leftdeep = height(T->Lchild);

rightdeep = height(T->Rchild);

deep = leftdeep>rightdeep?leftdeep+1:rightdeep+1;

}

return deep;

}

### 统计二叉树中所有节点的个数

int visit(BiTree T)

{

//先序遍历二叉树以实现求左子树和右子树的节点个数

if(T!=NULL)

{

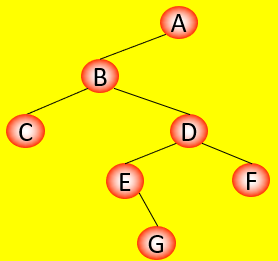
return 1+visit(T->Lchild)+visit(T->Rchild);

}

return 0;

}

### 建立二叉树的存储结构 —— 二叉链表（先序）

 若要建立一个ABCDEGF的二叉树，如图所示：

对右图所示二叉树，按下列顺序读入字符：

AB CΦΦDEΦGΦΦFΦΦΦ

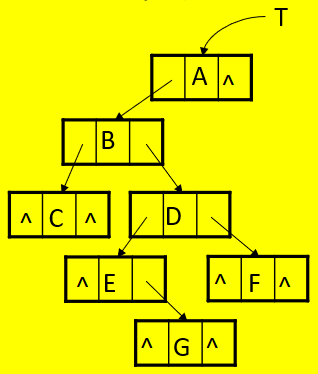
Status CreateBiTree(BiTree &T)

{

char ch;

scanf("%c",&ch);

if(ch == ' ')

 T = NULL;

else

{

if(!(T = (BiTNode \*)malloc(sizeof(BiTNode))))

exit(OVRFLOW);

T->data = ch;

CreateBiTree(T->Lchild);

CreateBiTree(T->Rchild);

}

return OK;

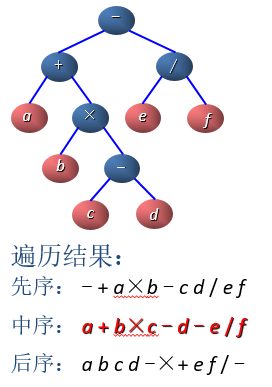
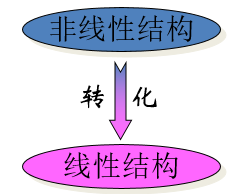
}// CreateBiTree

## 线索二叉树

问题 1：为什么要研究线索二叉树 ？

问题 2：如何保存结果以免重复遍历？

问题描述：

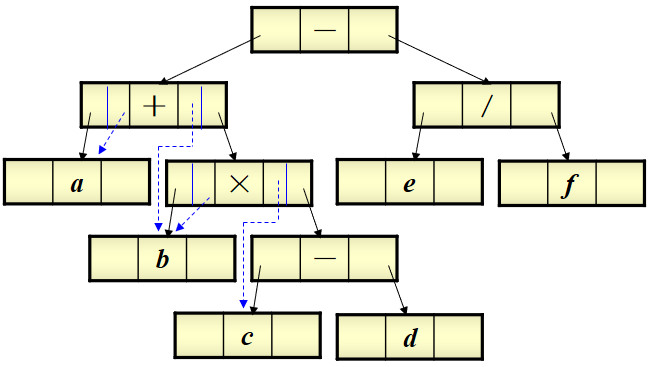
 

**如何找到遍历的前驱和后继呢？**

解决方案：

**1、**另辟空间存放遍历结果。需要付出额外的存储花销。

**2、**在原二叉链表的每个结点上增加两个指针域。



**3、**在原二叉链表的存储空间内反映遍历结果。

**线索二叉树的存储表示**

typedef enum PointerTag

{

Link, Thread

};

// Link == 0：指针，Thread == 1：线索

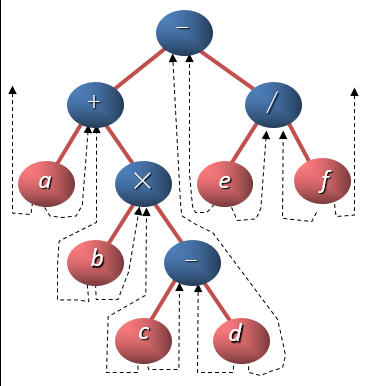
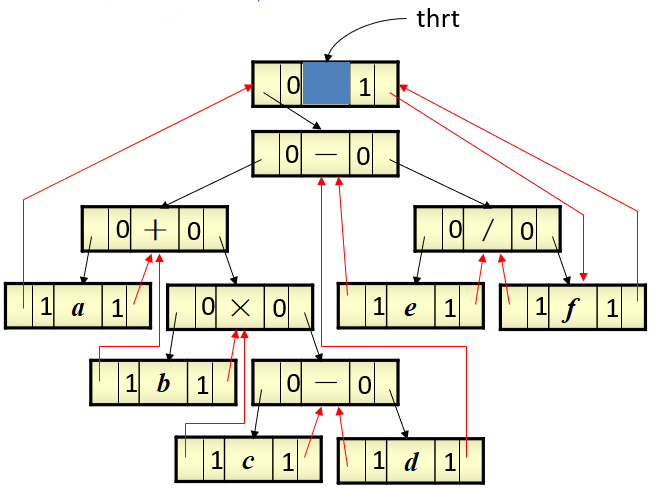
typedef struct BiThrNode {

TElemType data;

struct BiThrNode \*lchild, \*rchild; // 左右指针

PointerTag LTag, RTag; // 左右标志

} BiThrNode, \*BiThrTree;



在每一个节点中设置左右标记位，若标记为0，则代表是指针，指向的是结点，如果标记为1，则代表是线索，指示的是前驱元素或后继元素

其中为方便起见，在二叉树的线索链表上也添加了一个**头结点**，并另其lchild域的指针指向二叉树的根结点，其rchild的域的指针指向中序遍历是访问的最后一个结点；反之，令二叉树中序序列中的第一个结点的lchild域指针和最后一个rchild域指针均指向头结点。

**线索化：**对二叉树按照某种次序遍历使其变为线索二叉树的**过程**。

**二叉树线索化的目的：**利用线索化后的二叉树中的线索就可以直接找到某些结点在某种遍历序列中的**前趋和后继**结点。

**二叉树线索化的实质：**在遍历过程中**用线索取代空指针**。

**在线索树（中序）中找结点前驱的方法：**

1 、若左链是线索，则直接指示前驱；

2 、若左链是指针，则“左孩中找最右”。 即：中序前驱左孩找右。

**在线索树（中序）中找结点后继的方法：**

1 、若右链是线索，则直接指示后继；

2 、若右链是指针，则“右孩中找最左”。 即：中序后继右孩找左。

**在线索树上进行遍历的方法：**

1 、从序列中的第一个结点起，依次找后继，直至后继为空。

2 、从序列中的最后一个结点起，依次找前驱，直至前驱为空。

选择题

1. 在线索化二叉树中，t 所指结点没有左子树的充要条件是（C）

1. t -> lchild==NULL （B）t -> ltag==1

（C）t -> ltag==1且t -> lchild==NULL （D）以上都不对

2. 二叉树按某种顺序线索化后，任一结点均有指向其前趋和后继的线索，这种说法（B） （A）正确 （B）错误

# 树和森林

## 树的存储结构

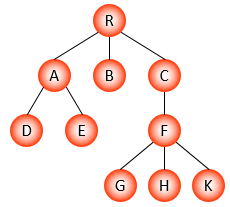
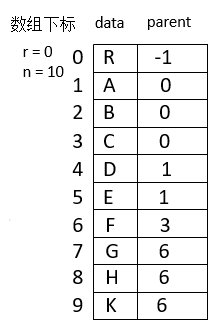
### 双亲表示法

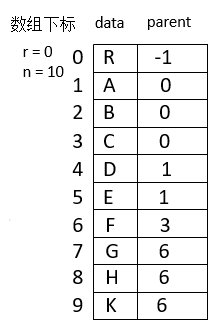
**实现：**定义结构数组存放树的结点，每个结点含两个域：

**数据域：**存放结点本身信息。

**双亲域：**指示本结点的双亲结点在数组中的位置。

**特点：找双亲容易，找孩子困难！**

**/\*\*\*约定根节点的双亲域是-1\*\*\*/**

/\*\*\*\*\*\*\*\*\*\*\***双亲表示法的存储结构**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct PTNode{

//定义每一个结点的结构--> 数据域、双亲域

TElemType data;

int parent; // 双亲位置域

}PTNode;

#define MAX\_TREE\_SIZE 100

typedef struct {

//利用双亲表示法定义数的结构

PTNode nodes[MAX\_TREE\_SIZE]; //结构数组

int r, n; // 根结点的位置和结点个数

} PTree;

### 孩子表示法（树的链式存储结构）

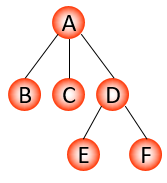
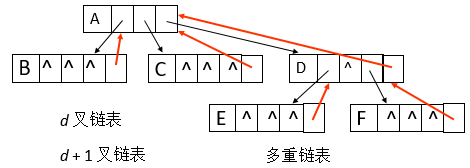
**1、多重链表**

即每一个节点有多个指针域，其中每个指针指向一棵字数的根节点。

**（1）结点同构的多重链表**

结点同构： 结点的指针个数相等，为树的度 *d* 。



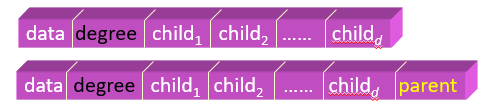
 

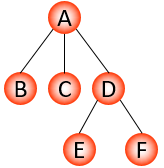
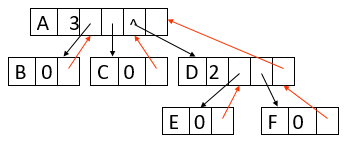
在有 *n* 个结点、度为*d* 的树的 *d* 叉链表中，有***n*×(*d*－1)＋1** 个空链域。

**特点：**由于树中很多结点的度小于d，所以链表中有很多空链域，空间较浪费。

**（2）结点不同构的多重链表**

结点不同构：结点的指针个数不相等，为该结点的度 **degree**。

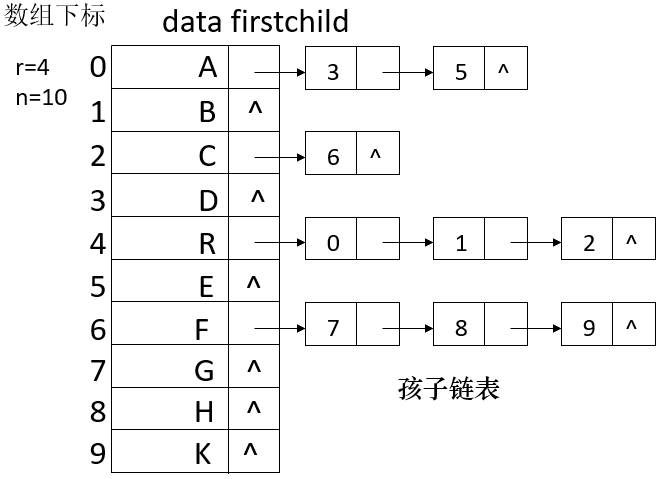
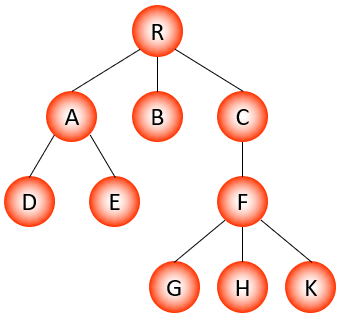


**特点：**节约存储空间，但操作不方便

**2、孩子链表**

**孩子链表：**把每个结点的孩子结点排列起来，看成是一个线性表，用单链表存储，则 *n* 个结点有 ***n*** 个孩子链表**（叶子的孩子链表为空表）**。而 *n* 个头指针又组成一个线性表，用顺序表（含 *n* 个元素的结构数组）存储。



**特点：找孩子容易，找双亲难。**

/\*\*\*\*\*\*\*\*\*\*孩子链表存储结构\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define MAX\_TREE\_SIZE 100

typedef struct CTNode{

//孩子结点的结构

int child; //表示的是孩子的下标

struct CTNode \*next;

}\*ChildPtr;

typedef struct{

//双亲结点结构

TElemType data;

ChildPtr firstchild; // 孩子链表头指针

}CTBox;

typedef struct{

//树结构

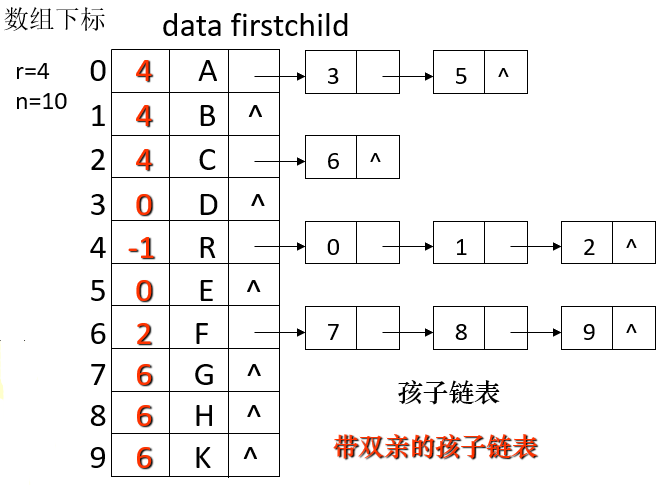
CTBox nodes[MAX\_TREE\_SIZE]; //结构数组

int n, r; // 结点数和根结点的位置

}CTree;

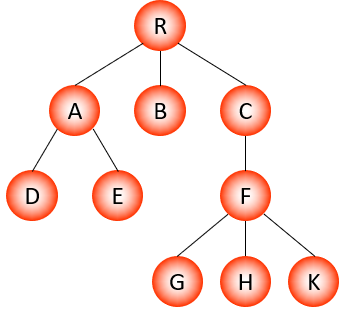
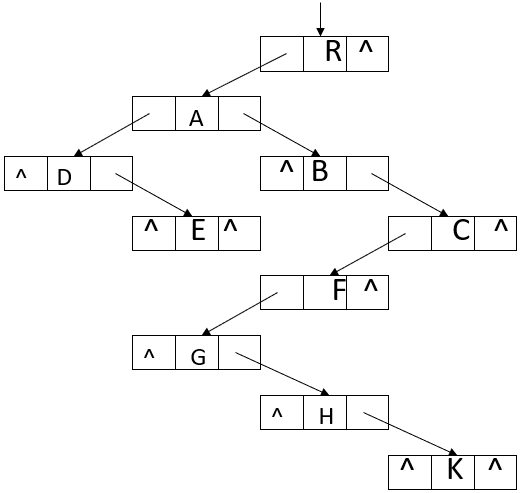
**3、双亲表示法**

通过上面两种方法，我们可以将双亲表示法和孩子表示法结合起来，即将双亲表示和孩子链表合在一起：



### 孩子兄弟表示法（二叉树表示法，二叉链表表示法）

实现：用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其**第一个孩子结点和下一个兄弟结点**

**注意：**孩子兄弟链表的结构形式与二叉链表完全相同，但存储结点中指针的含义不同：**二叉链表中结点的左右指针分别指向该结点的左右孩子**；**而孩子兄弟链表结点的左右指针分别指向它的“长子”和“大弟”。**

/\*\*\*\*\*\*\*\*\*\*孩子-兄弟链表存储结构\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct CSNode{

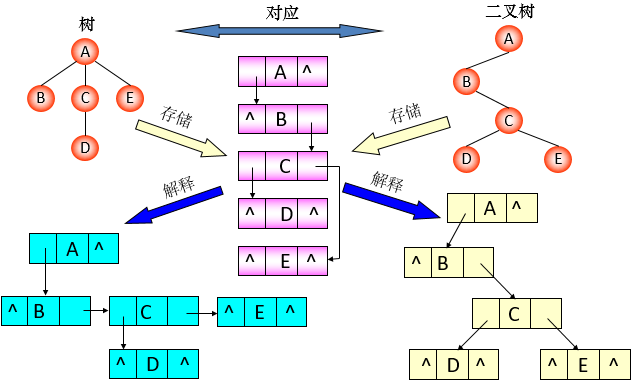
TElemType data;

struct CSNode \*firstchild, \*nextsibling;

} CSNode, \*CSTree;

## 森林和二叉树的转换

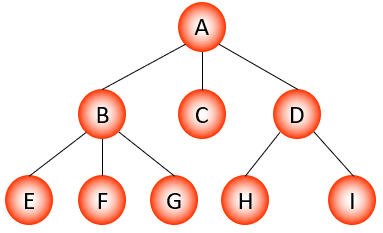
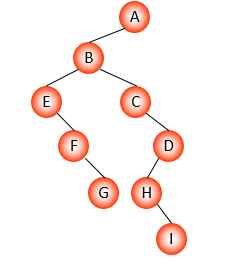
### 树与二叉树的转换



**1、将树转换成二叉树**

**树变二叉树：兄弟相连留长子。**

* **加线**：在兄弟之间加一连线
* **抹线**：对每个结点去除其与孩子之间的关系（第一孩子除外）
* **旋转**：以树的根结点为轴心，顺时针转45°。

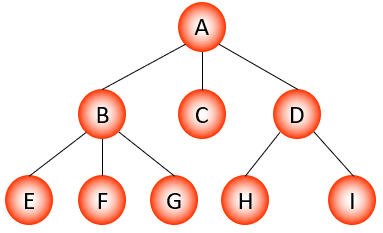
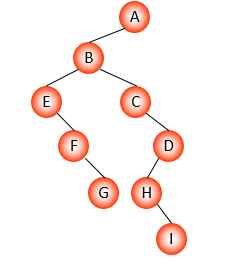
 

**树转换成的二叉树其右子树一定为空。**

**2、将二叉树转换成树**

**二叉树变树：左孩右右连双亲，去掉原来右孩线。**

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

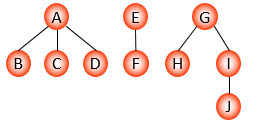
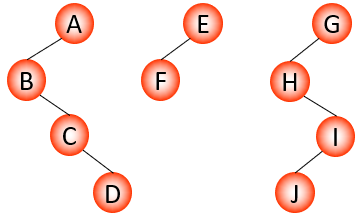
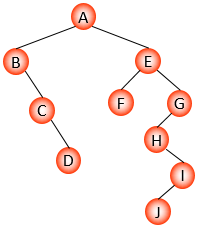


### 森林与二叉树转换

**1、将森林转换成二叉树**

**森林变二叉树：树变二叉根相连。**

* 将各棵树分别转换成二叉树。
* 将每棵二叉树的根结点用线相连。
* 以第一棵二叉树根结点为二叉树的根，再以根结点为轴心，顺时针旋转，构成二叉树型结构。

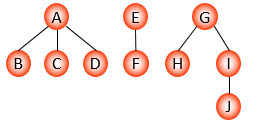
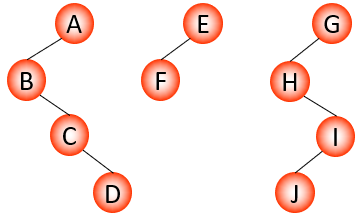
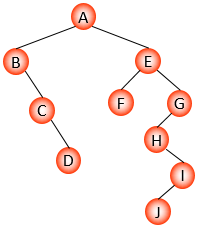


**2、将二叉树转换成森林**

**二叉树变森林：去掉全部右孩线，孤立二叉再还原。**

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

**还原：**将孤立的二叉树还原成树。



## 树与森林的遍历

### 树的遍历（先，后，层）

**1、先根（次序）遍历:**

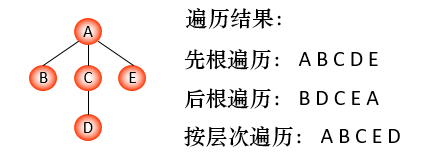
若树不空，则先访问根结点，然后依次先根遍历各棵子树。

**2、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。

**3、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。



### 森林的遍历（先序（树先），中序（树后））

森林由三部分构成：

1、森林中第一棵树的根结点；

2．森林中第一棵树的子树森林；

3．森林中其它树构成的森林。

* **先序遍历**：**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。

**即：依次从左至右对森林中的每一棵树进行先根遍历。**

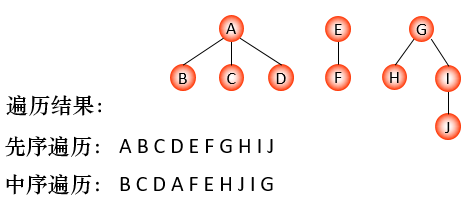
* **中序遍历**：若森林不空，则

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。

**即：依次从左至右对森林中的每一棵树进行后根遍历。**

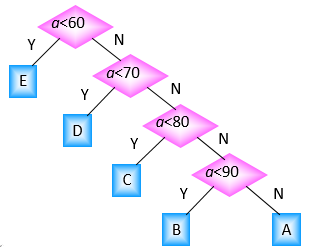


# 哈夫曼（Huffman）树及其应用

## 问题描述

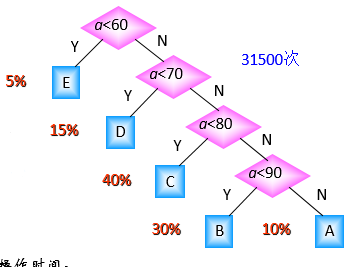
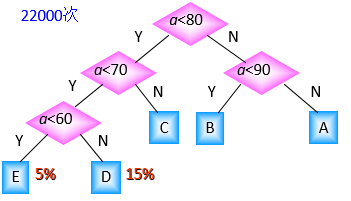
例：将学生的百分制成绩转换为五分制成绩：≥90 分: A，80～89分: B，70～79分: C，60～69分: D，＜60分: E。

转换为程序语言：



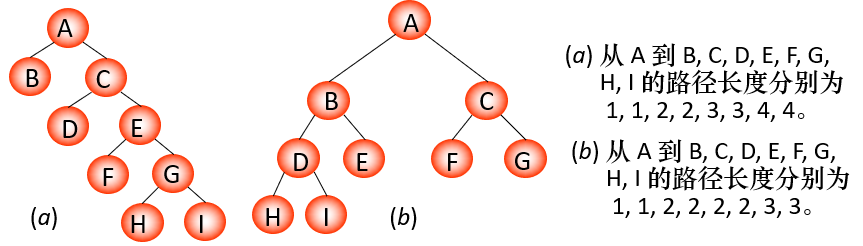
若学生的成绩数据共10000个：

则5％的数据需 1 次比较，15％的数据需 2 次比较，40％的数据需 3 次比较，40％的数据需 4 次比较，因此 10000 个数据比较的次数为： 10000 (5％＋2×15％＋3×40％＋4×40％)＝31500次

## 基本概念

**路径**：从树中一个结点到另一个结点之间的**分支**构成这两个结点间的路径。**结点的路径长度**：两结点间路径上的分支数。 （主要考虑分支的数量）



**树的路径长度**：从树根到每一个结点的路径长度之和。记作：TL

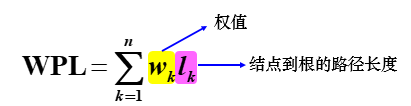
TL（*a*）＝0＋1＋1＋2＋2＋3＋3＋4＋4＝20   
TL（*b*）＝0＋1＋1＋2＋2＋2＋2＋3＋3＝16

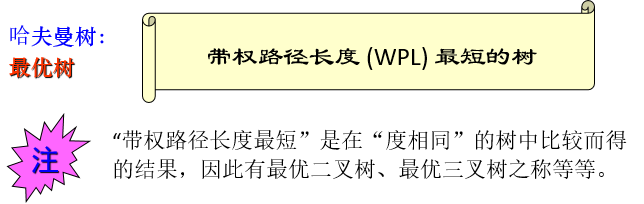


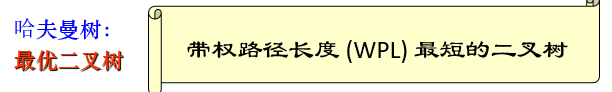
**权**：将**树中结点赋给一个有着某种含义的数值**，则这个数值称为该结点的权。

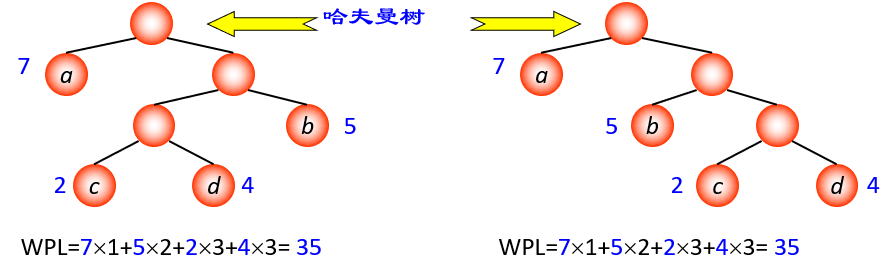
**结点的带权路径长度**：从根结点到该结点之间的路径长度与该结点的权的乘积。

**树的带权路径长度**：树中所有叶子结点的带权路径长度之和。 记作：









注意：

* **满二叉树不一定是哈夫曼树**
* **哈夫曼树中权越大的叶子离根越近**
* **具有相同带权结点的哈夫曼树不惟一**

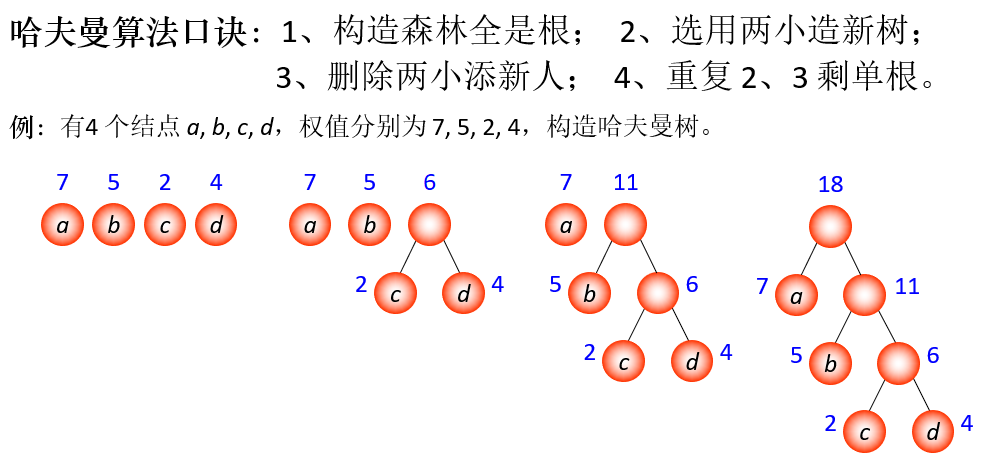
## 哈夫曼算法（构造哈夫曼树的方法）

1、根据 *n* 个给定的权值 {*w*1, *w*2, …, *wn*} 构成 *n* 棵二叉树的森林*F*={*T*1, *T*2, …, *Tn*}，其中 *Ti* 只有一个带权为 *wi* 的根结点。

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

3、在 *F* 中删除这两棵树，同时将新得到的二叉树加入森林中。

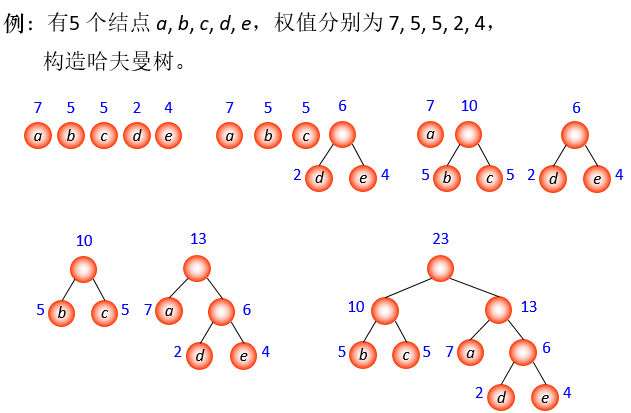
4、重复 (2) 和 (3)，直到森林中只有一棵树为止，这棵树即为哈夫曼树。



注意：

**（1）哈夫曼树的结点的度数为 0 或 2， 没有度为 1 的结点。**

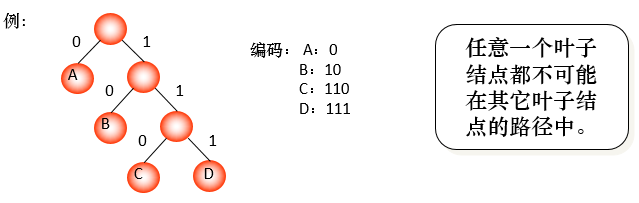
**（2）包含 *n* 个叶子结点的哈夫曼树中共有 2*n* – 1 个结点。**



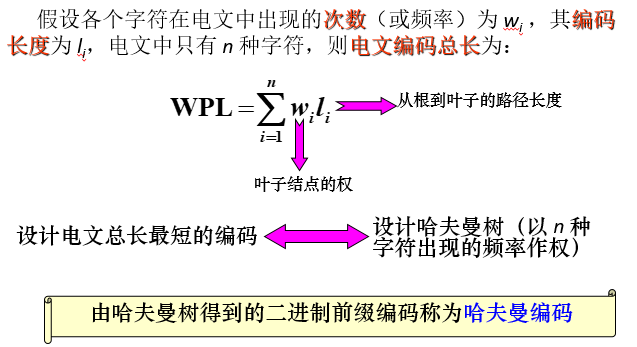
## 哈夫曼编码

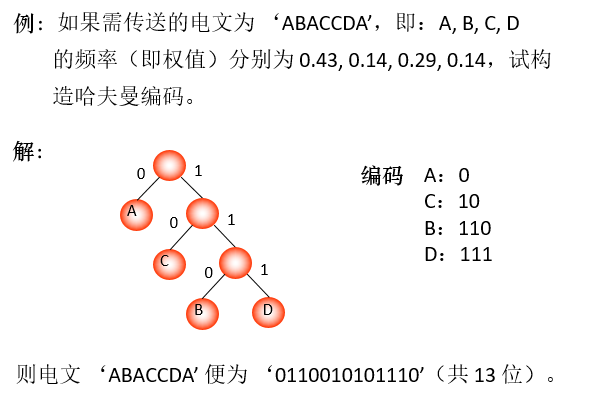
用二叉树设计二进制前缀编码

以电文中的字符作为叶子结点构造二叉树。然后将二叉树中结点引向其左孩子的分支标 ‘0’，引向其右孩子的分支标 ‘1’； 每个字符的编码即为从根到每个叶子的路径上得到的 0, 1 序列。如此得到的即为**二进制前缀编码**。



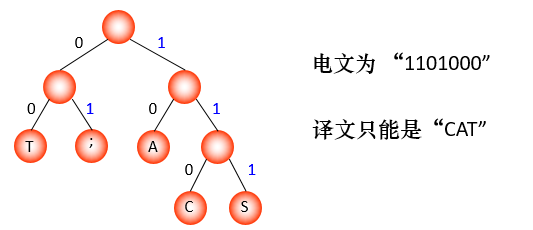
用哈夫曼树设计总长最短的二进制前缀编码





## 译码

从哈夫曼树根开始，对待译码电文逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束。



1、一棵哈夫曼树有 19 个结点，则其叶子结点的个数是( )。

2、有七个带权结点，其权值分别为 3, 7, 8, 2, 6, 10, 14，试以它们为叶结点构造 一棵哈夫曼树（请按照每个结点的左子树根结点的权小于等于右子树根结点的权的次序构造），并计算出带权路径长度WPL及该树的结点总数。  
3、有一电文共使用五种字符 a, b, c, d, e，其出现频率依次为 4, 7, 5, 2, 9。  
 (1)、试画出对应的编码哈夫曼树（要求左子树根结点的权小于等于右子树根结点的权）。   
 (2)、求出每个字符的哈夫曼编码。

(3)、求出传送电文的总长度。  
 (4)、并译出编码系列11000111000101011的相应电文。

4、对于给定的一组权值 W＝{1, 3, 7, 8, 14, 20, 28} 建立哈夫曼树，并计算带权路径长度。

5、假定有 7 个字符 a, b, c, d, e, f, g 出现的概率分别为 0.07, 0.09, 0.14, 0.23, 0.44, 0.58, 0.77，求这 7 个字符的哈夫曼编码。