# 树的定义和基本术语

定义：树是一个n(n>=0)个结点的有序合集，当n=0的时候 树为空。

树的性质：

1. 根节点只有一个，子树的个数没有限制，但是不会相交！
2. 结点的度degree 结点的度为结点孩子的个数，二叉树的度要么是0，要么是1或者2，树不分入度和出度，树的度为树中所有结点中最大的度数，度为0的结点称为叶子节点，也成为终端节点，其余的结点称为非终端结点。
3. 内部节点是，除了根节点之外的非终端结点，也就是根节点和叶子节点中间的结点
4. 结点：双亲&孩子&兄弟（相同的双亲）&祖先（根到该节点经过的所有结点）
5. 结点的层次，根节点算第一层，树的深度Depth是最大的层数
6. 森林是m棵互不连通的树

# 树的存储结构

孩子表示法

设置一个数组，每个slot里面都存储树中的结点元素内容，并且将所有结点的数据都存在其中。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| childNode[0] | childNode[1] | childNode[2] | childNode[3] | childNode[4] | childNode[5] | childNode[6] | childNode[7] | childNode[8] | childNode[9] | childNode[10] | childNode[11] | childNode[12] |

这种方式要定义一个数据结构

Typedef struct ChildNode{

ElemType data; //用来记录孩子的数据

int parents; //用来记录孩子的双亲

}ChildNode;

按照这种数据结构建立数组Array，每个孩子都有指向自己双亲的“指针“，这样如果要找到某个孩子双亲，直接访问其parents指向的元素即可，时间复杂度为O(1),但是这种如果先要访问某个双亲的孩子，就必须遍历整个数组，看看哪个元素的双亲是要找的（即看看哪个孩子的双亲是要找的）时间复杂度就为O（n）

双亲表示法

为了解决孩子表示法找到双亲的孩子很麻烦的问题，引入双亲表示法，双亲表示法也是设计一个数组，这里面存放了所有的元素，并且定义一个数据类型

Typedef struct Parents{

ElemType data;

Parents\* next;

}Parents;

每个结点都有一个指向自己孩子的指针，这种查找孩子的时候很方便但是如果找孩子的双亲比较困难，由此又引入了一个双亲数组，也激就是双亲孩子表示法

**双亲孩子表示法**

定义数据结构

Typedef struct ChildNode{

int data;

ChildNode \*next;

}ChildNode,\*ChildNodePtr;

//定义一个孩子结点

Typedef struct CTBox{

int data;

ChildNodePtr \*child;

}CTBox,\*CTBoxPtr

//定义一个双亲BOX

CTBox ctBOX[ARRAY\_LENGTH];

//定义双亲BOX类型的数组

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ctBox[0] | ctBox[1] | ctBox[2] | ctBox[3] | ctBox[4] | ctBox[5] | ctBox[6] | ctBox[7] |

ChildNodN

ChildNodN

ChildNodN

ChildNodN

ChildNodN

NULL

用这样的方式既可以通过双亲找到双亲的孩子，也可以通过孩子找到与之对应的双亲，包含了前两种方式的优点。

# 二叉树

每个结点的子树为0-1-2个，即每个结点最多有两个子树，不存在degree>2的情况

二叉树是有序树，区分左右子树。

一般二叉树的性质：

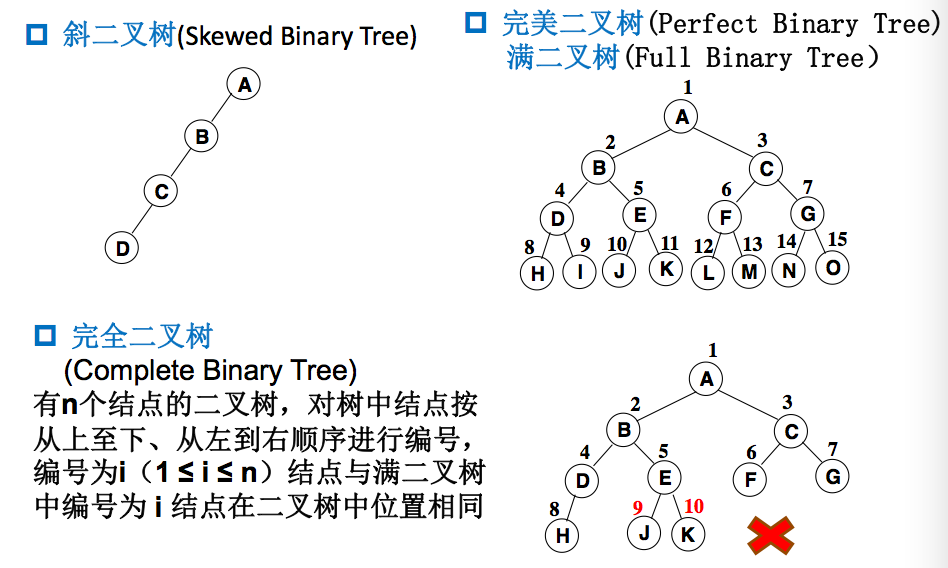
1. 一棵二叉树所有度为0的结点为n0 度为2的结点为n1 则有关系 n0=n1+1
2. 二叉树上第i层最大的结点个数为 2^（i-1） （满二叉树）
3. 深度为k的二叉树最多结点个数n=2^k-1 即满二叉树的情况

A A

A

B B

这两种是不同的树，称为斜二叉树



编号为i的结点与对应深度的满二叉树位置相同就是完全二叉树，满二叉树是完全二叉树的特例，是完全二叉树派生出来，所以满二叉树一定是完全二叉树，完全二叉树不一定是满二叉树。 满二叉树 能推出 完全二叉树 完全二叉树推不出满二叉树

（完全二叉树只能从满二叉树上删除最右面的结点，不能从中间删除，这样就不能保证顺序的一致性）所以说叶子只能出现在最下面两层，并且如果某个结点度数为1，那么只有左孩子

完全二叉树重要性质：

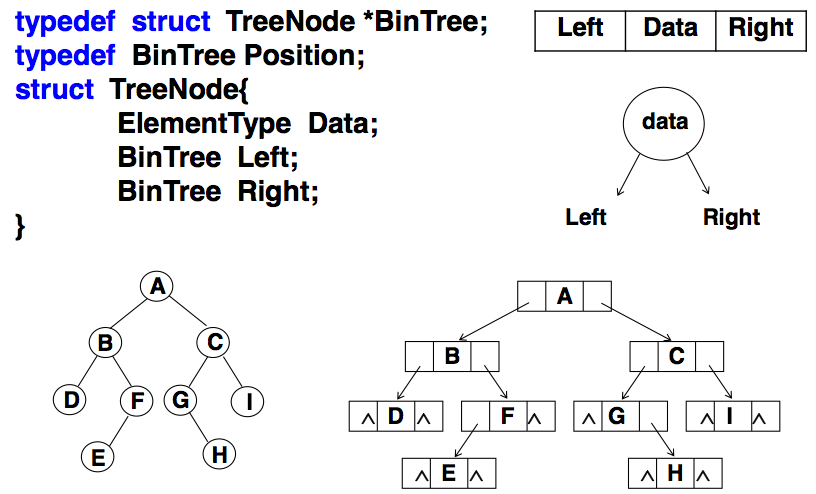
（1）如果存在数组里面 index从1开始，那么任意根节点的index\*2为左节点 index\*2+1为右节点

（2）完全二叉树的深度 depth = log2n(取下限)+1

（3）对于一棵完全二叉树 如果i=1 那么就是根结点，如果i>1那么根节点为i/2取下限

对于结点i 如果2i<n 那么左孩子为2i，如果2i+1<n那么右孩子为2i+1

# 二叉树的链式存储结构

定义数据结构

Typedef struct tree{

ElemType data；

Tree\* lchild;

Tree\*rchild;

}tree,\*treePtr;

# 二叉树的生成

使用链式存储二叉树的时候，需要按照 先序 中序 后序的方式输入数据来生成二叉树，对于内些没有指向的结点指针，可以用一个符号入”#”代替，这样完成了以下的初始化二叉树的代码，采用先序遍历输入数据，并且递归的创建二叉树。

**int CreateBiTree(PTNode &T)**

**{**

**char ch;**

**scanf\_s("%c", &ch); //每次从缓冲区中读数据**

**if (ch == '#')**

**T = NULL;**

**else**

**{**

**T = (TNode\*)malloc(sizeof(TNode));**

**if (!T) exit(EOVERFLOW);**

**T->Data = ch;**

**T->LTag = link;**

**T->RTag =link;**

**CreateBiTree(T->Left); //生成左节点的子树**

**CreateBiTree(T->Right); //生成右节点的子树}**

**return 1;}**

# ****二叉树的层数&叶子节点数****

使用递归

int maxDepth(treePtr&tree)

{

If(tree==NULL)

Return 0;

//如果T本来就为空，那么就返回0，说明没有节点了/已经到叶子节点了

else

{

int maxLeft=maxDepth(tree->lchild);

int maxRight=maxDepth(tree->rchild);

//递归的求得该节点左右节点的最大值

if(maxLeft>maxRight)

return maxLeft+1;

else if(maxLeft<maxRight)

return maxRight+1;

}

}

使用递归求叶子节点

int NumOfLeaf(TreePtr&tree)

{

if(tree==NULL)

return 0; //结点不存在

else

{

if(tree->lchild==NULL&&tree->rchild==NULL)//结点为叶子节点

return 1;

else

return tree->lchild+tree->rchild;

}

}

# (重点)二叉树的遍历

先序遍历 先根节点，再左右节点

int PreOrderTraverse(PTNode&T)

{

if (T == NULL)

return -1;

else {

printf("先序遍历|节点信息为:%c\n", T->Data);

PreOrderTraverse(T->Left);

PreOrderTraverse(T->Right);

}

return 1;

}

中序遍历，先左子树 再该节点 再右子树 利用递归

int InOrderTraverse(PTNode&T)

{

if (T == NULL)

return -1;

else {

//后续的顺序不同

InOrderTraverse(T->Left);

printf("中序遍历|节点信息为%c\n", T->Data);

InOrderTraverse(T->Right);

}

return 1;

}

后序遍历，先左子节点 再右子节点 再根节点

int PostOrderTraverse(PTNode&T)

{

if (T == NULL)

return -1;

else {

//后续的顺序不同

PostOrderTraverse(T->Left);

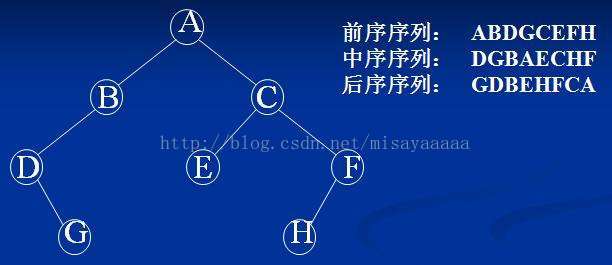
PostOrderTraverse(T->Right);

printf("后序遍历|节点信息为%c\n", T->Data);

}

return 1;

}



# 线索二叉树

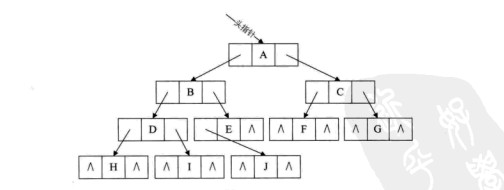
使用中序遍历，设置前驱后继

完全二叉树，中序遍历之后每隔一个有一个浪费空间的位置

设置ltag rtag

Ltag为0的时候，结点为左孩子，为1的时候是前驱

同理的 Rtag为0的时候，结点为右孩子，为1是后继



http://blog.chinaunix.net/attachment/201301/17/26548237_13584034551s55.jpg

enum PointerTag {link,Thread};

typedef struct TreeNode

{

char Data;

TreeNode \*Left;

TreeNode \*Right;

PointerTag LTag, RTag; //左右标志 0 表示存储前驱后继 1表示线索

}TNode,\*PTNode;

//设置全局指针

PTNode ch;

int counter = 0;

int BiThrTree(PTNode&T) //线索二叉树

{

if (T) {

//中序

BiThrTree(T->Left);

//ABC##D##EFG##H###

if (T->Left == NULL) {

T->LTag = Thread;

T->Left = ch;

printf("线索化结点:%c的左孩子\n", T->Data);

}

if (ch->Right == NULL&&counter!=0)//判断ch的lchild

{

ch->RTag = Thread;

ch->Right = T;

printf("线索化结点:%c的右边孩子\n", ch->Data);

}

ch = T;

counter++;

BiThrTree(T->Right);

}

return 1;

}

void InOrderThreading(PTNode&P,PTNode&T)

{

P = (PTNode)malloc(sizeof(TNode));

P->LTag =link;

P->RTag = Thread;

P->Right = P;

P->Left = T;

ch = P->Left;

printf("初始化完成...\n");

BiThrTree(T);

ch->RTag = link;

ch->Right = P;

P->Right = ch;

}

这里面是设计一个头节点，并且让头节点的左指针指向树的根节点，从根节点开始，设置ch记录当前结点的前驱结点，最左边的结点与根节点链接，然后根节点可以与该节点链接，这个时候开始中序遍历，ch记录前驱结点，如果当前结点的lchild为空即ltag==thread (1)那么就让这个左指着指向ch，并且如果ch的右指针是空，则让其指向该结点，由于中序遍历的过程中，我们只能得到当前结点左面的数据，所以不能直接让当前节点链接其后继，所以这里面引入ch，既可以为当前节点链接前驱，又可以为前驱结点链接后继 最后需要将遍历最后的一个结点（ch）与头节点相互连接 这样就将整个遍历串起来了。

void inorder(bintree \*head){

bintree \*t = head->leftchild;//通过头结点进入根节点

while(t!=head){//表示已遍历完成，指针t已经回到了头结点

while(t->lefttag==0)//循环找到中序遍历的第一个节点

t = t->leftchild;

cout<<t->data<<" ";

while(t->righttag==1&&t->rightchild!=head){//不断的输出后继，直到某个节点rightchild所指的不是后继，即righttag!=1

t = t->rightchild;

cout<<t->data<<" ";

}

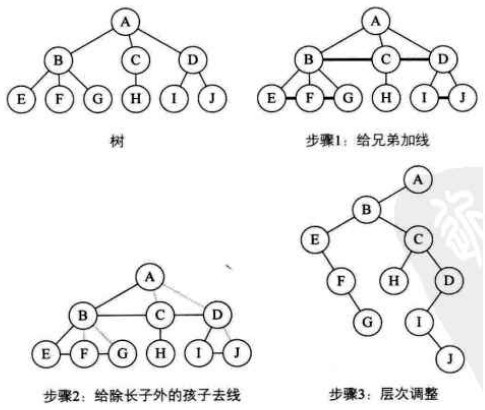
t = t->rightchild;//进入右子树

}

}

# 树的转换

树到二叉树的转换



1.在树种所有兄弟节点之间加1连线

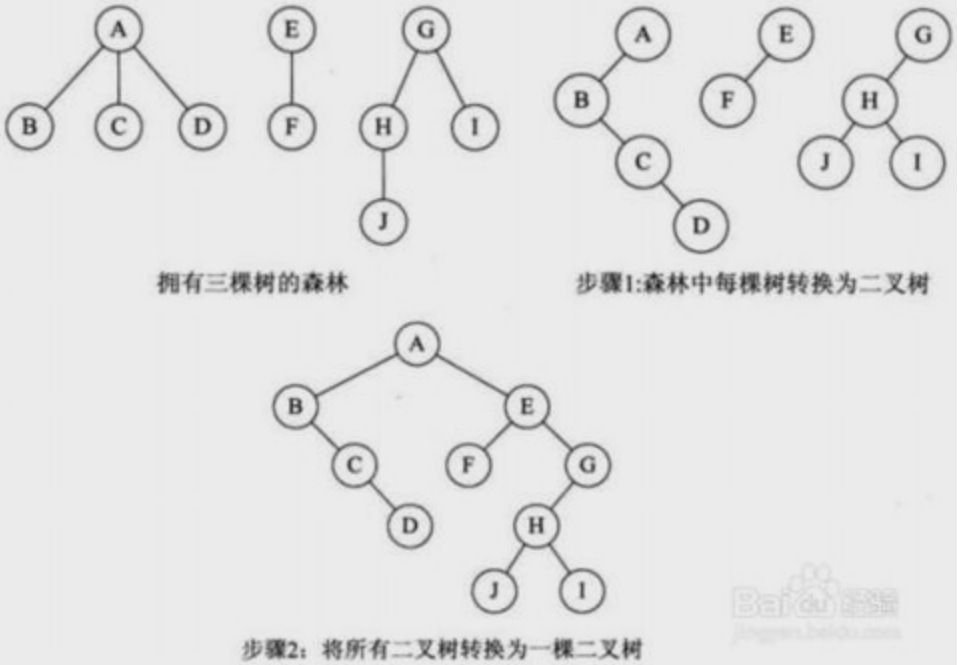
2.对每个结点，除了保留长子连线外，去掉其他孩子的连线

3.调整

由于仅仅保留了最左子树，所以根节点没有右子树

森林转化二叉树

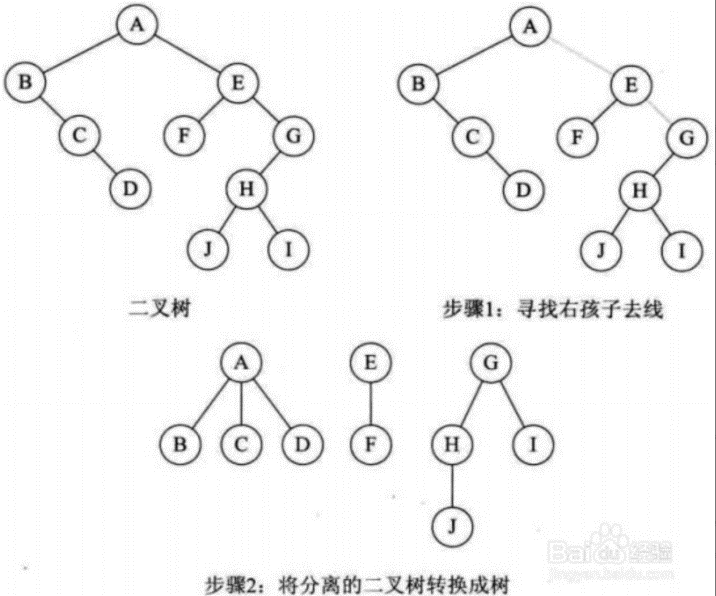
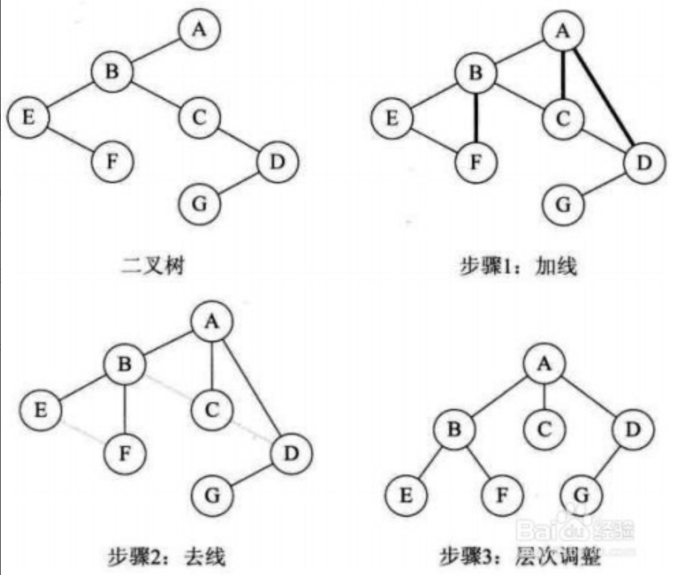
1. 将森林的每一棵树变成二叉树
2. 将每一棵二叉树的根结点视为兄弟，从左到右连接
3. 调整



二叉树到树和森林的转换

1. 若结点x是其双亲y的左孩子，则把x的右孩子，右孩子的右孩子，用y连接起来
2. 去掉所有双亲结点到右孩子的连线

判断二叉树是否能转换成树还是森林，那么就是判断二叉树根节点有没有右孩子，没有右孩子则转换为树，有右孩子就是森林

 二叉树到树 二叉树到森林

# 树与森林的遍历

先根遍历 后根遍历

森立就是按照先根遍历 后根遍历 一次访问每一棵树

树 森林的前序遍历与对应二叉树的前序遍历相同，后序遍历与对应二叉树的中序遍历相同

# 赫夫曼树

**1．树的路径长度**

树的路径长度是从树根到树中每一结点的路径长度之和。在结点数目相同的二叉树中，完全二叉树的路径长度最短。

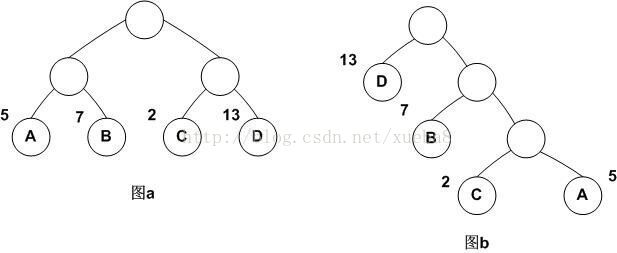
**2．树的带权路径长度（**Weighted Path Length of Tree，简记为WPL）

　　结点的权：在一些应用中，赋予树中结点的一个有某种意义的实数。

　　结点的带权路径长度：结点到树根之间的路径长度与该结点上权的乘积。

　　树的带权路径长度（Weighted Path Length of Tree）：定义为树中所有叶结点的带权路径长度之和

哈夫曼树是一种带权路径长度最短的二叉树，也称为最优二叉树。下面用一幅图来说明。



它们的带权路径长度分别为：

图a： WPL=5\*2+7\*2+2\*2+13\*2=54

图b： WPL=5\*3+2\*3+7\*2+13\*1=48

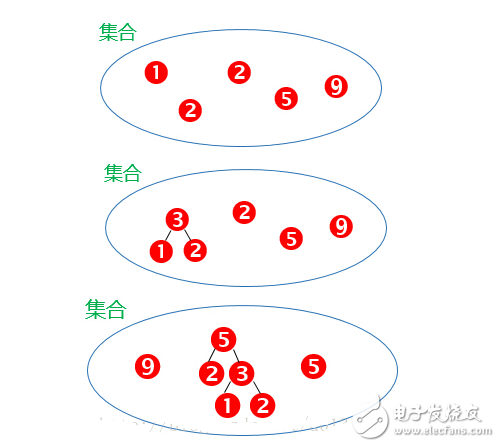
构造哈夫曼树的算法描述。

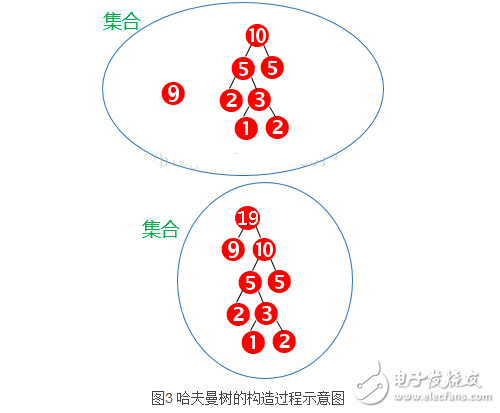
　　1、根据给定的n个权值{w［1］，w［2］，…，w［n］}构成n棵二叉树的集合F={T［1］，T［2］，…T［n］}， 其中每棵二叉树T［i］;中只有一个带权为w［i］的根结点，其左右子树均为空。

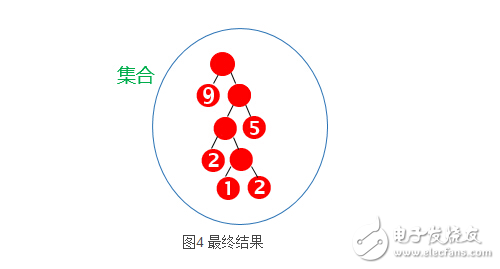
　　2、在F中选取两棵根结点的权值最小的树作为左右子树构造一棵新的。二叉树，且置新的二叉树的根结点的权值为其左右子树上根结点的权值之和，

　　3、在F中删除这两棵树，同时将新得到的二义树加入F中。

　　4重复2和3步骤，直到F只含一棵树为止。这棵树便是哈夫曼树。







WPL = 1\*9 + 2\*5 + 3\*2 + 4\*1 + 4\*2 =37

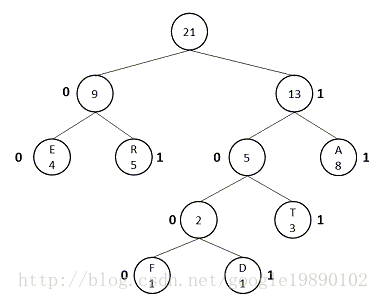
赫夫曼编码

赫夫曼编码是一种前缀码，即任何一个编码不是另一个编码的前缀

这样可以接收编码的位数，使得每位不是等位数的也可以正确无误的表达意思

* 将权值小的最为左节点，权值大的作为右节点
* 左孩子编码为0，右孩子编码为1

比如26个英文字母’A’~’Z’只要设置每个字母不同的权值，就会生成不同的赫夫曼编码树，这样我们可以规定左子树和右子树的编码(0，1)这样找到一个字母就可以有不同的0 1 组合



就比如说，找到E 可以经过 00 找到 D可以经过 1001

# 练习题

**若二叉树用二叉链表作存贮结构，则在n个结点的二叉树链表中只有n—1个非空指针域。**

T

因为二叉树的每个结点都有一个指针指向，所以N个节点一共有N-1个指针非空，根节点除外，所以N个节点的二叉树有2N个子树 所以有N+1个空指针

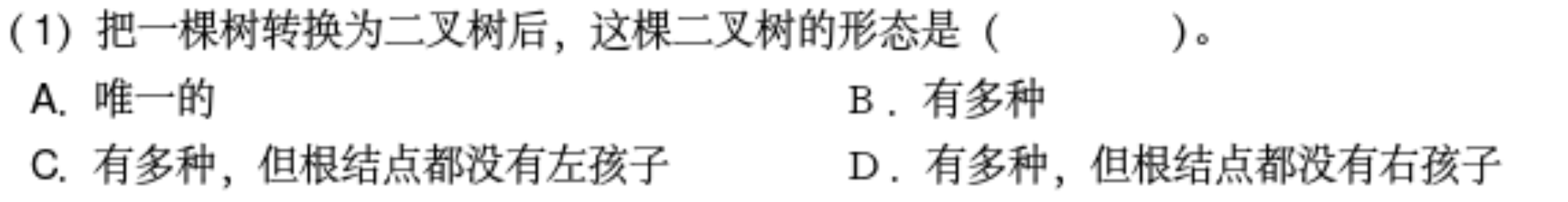
**（ A ）4．把一棵树转换为二叉树后，这棵二叉树的形态是 。**

**（Ａ）唯一的 （Ｂ）有多种**

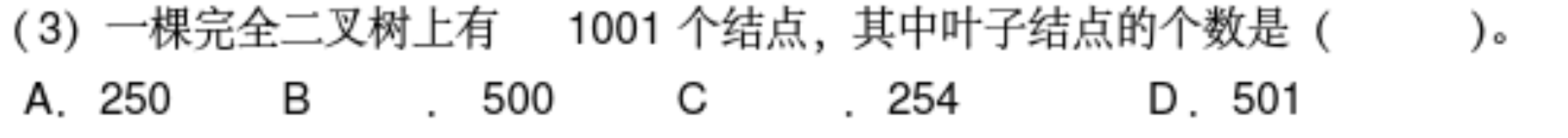
**（Ｃ）有多种，但根结点都没有左孩子 （Ｄ）有多种，但根结点都没有右孩子**

前序中序确定二叉树 后续中序确定二叉树 前后不能确定

假设某二叉树的先序遍历序列是abdgcefh，中序遍历序列是dgbaechf

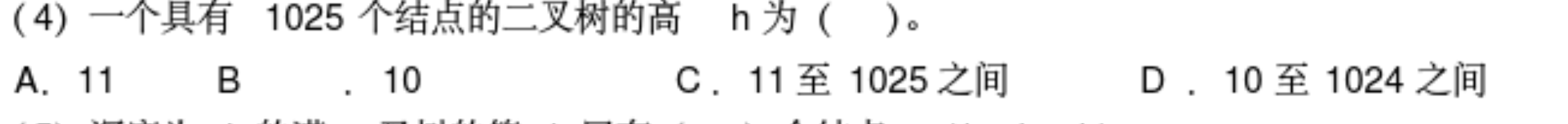


A

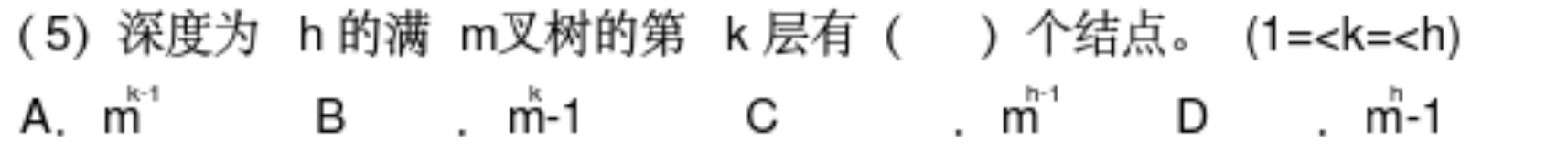


由完全二叉树得出 1001是右孩子 那么其双亲为500

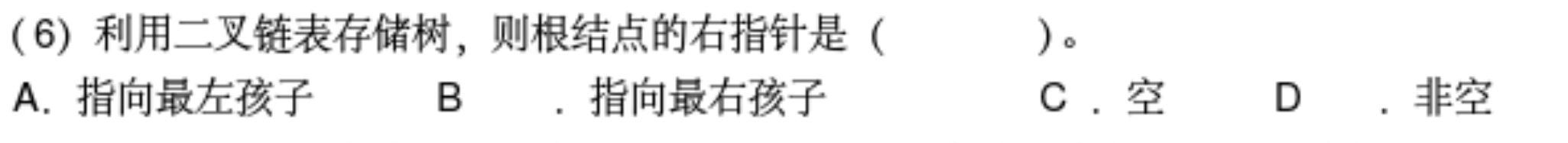
由于是最后一个结点 所以n1=0 故 n0+n2=1001 n0=n2+1 n0=501 选D



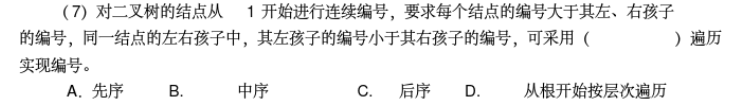
如果是满二叉树 那么根据公式为11 斜树为1025



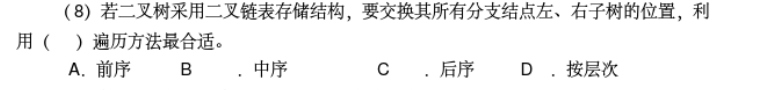
A类比满二叉树



C二叉链表右孩子为NULL



由于其每个结点的编号大于其左右孩子的编号，所以先遍历该结点的孩子，再遍历该结点。在一结点的左右孩子中，由于其左孩子的编号小于其右孩子的编号，所以先遍历左孩子再遍历右孩子。由此可知，遍历的顺序为：左孩子→右孩子→根结点。可采用后序遍历

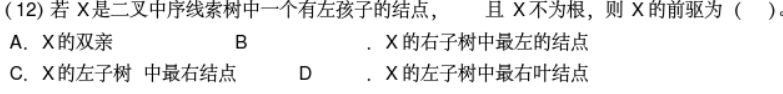


显然后序遍历比较合理。正常的逻辑应该就是：做好当前结点子树内部的交换，然后交换当前结点的左右子树。刚好符合后序遍历的算法逻辑。  
1. 交换好左子树  
2. 交换好右子树  
3. 交换左子树与右子树  
其他算法如先序和按层次其逻辑都差不多，即访问当前结点时交换其左右子树。从逻辑上来看稍显别扭一点点。因此说最合适应该是后序遍历，但是从实现上来说先序和按层次都是可以的。

判断下列说法是否正确：若一棵非空二叉树的先序遍历和后序遍历具有相同的结点访问顺序，则它一定是一棵只有根结点的二又树。（）

* 正确
* 错误

A



C不一定叶结点



[物理结构](https://www.baidu.com/s?wd=%E7%89%A9%E7%90%86%E7%BB%93%E6%9E%84&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)  
逻辑结构：集合、线性、树和图  
[物理结构](https://www.baidu.com/s?wd=%E7%89%A9%E7%90%86%E7%BB%93%E6%9E%84&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)：线性存储和非线性存储  
其中，线性[存储结构](https://www.baidu.com/s?wd=%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)有顺序（sequential）、链接（linked）、索引（indexed）和散列（hashing）4种结构  
非线性[存储结构](https://www.baidu.com/s?wd=%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)有：树形[存储结构](https://www.baidu.com/s?wd=%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)、图形存储结构。

# 设F是一个森林，B是由F转换得到的二叉树，F中有n个非终端结点，B中右指针域为空的结点有?

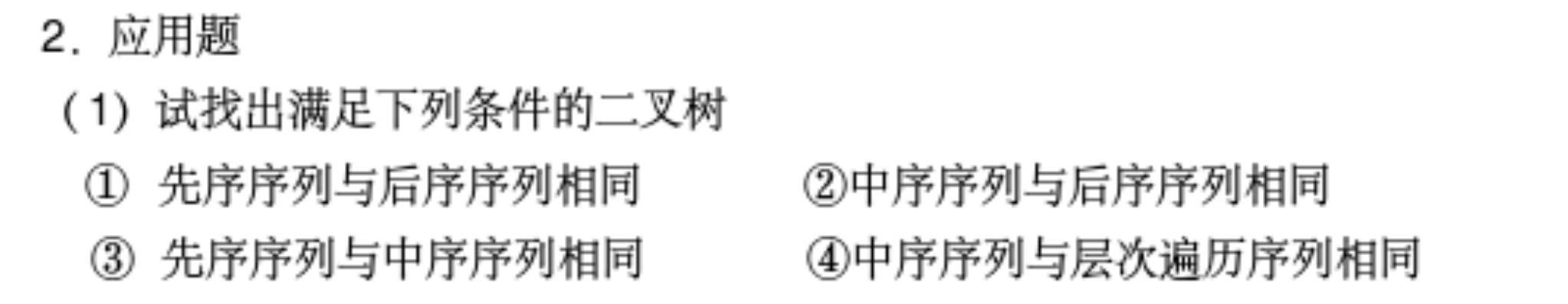
设终端结点数为f，总结点数 f + n。每个结点有两个指针，总指针数 2（f + n）

二叉树B除去根结点，都是某个结点的孩子， 也就是其余每个结点都有指针指向， 占用指针数为 f + n - 1；

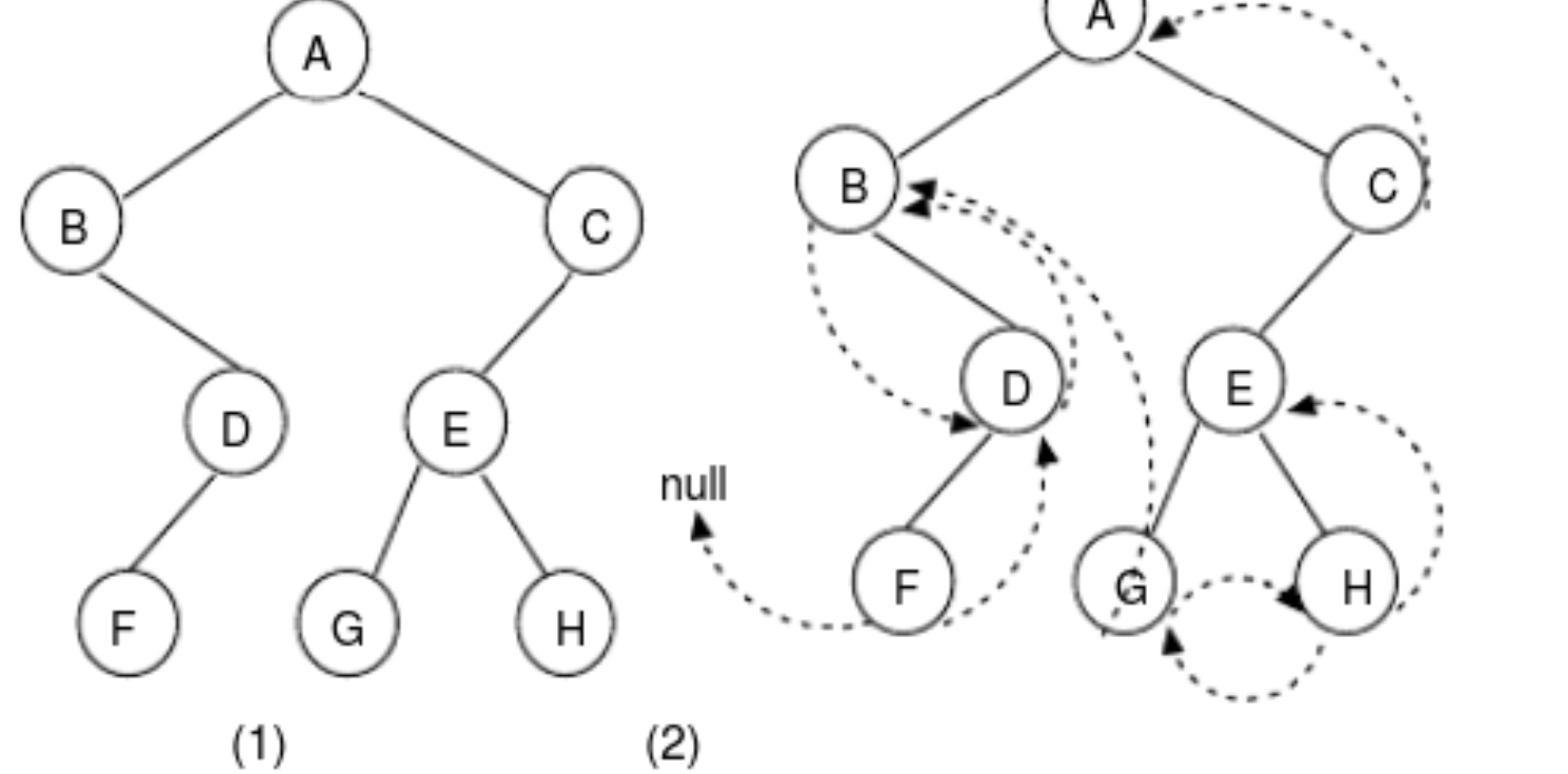
剩余空指针数为 2（f + n）- （f + n - 1）= f + n + 1 个

f 个终端结点没有孩子， 所以空的左指针域数为f 个；

二叉树B 中右指针域为空的结点有 ( f + n + 1 ) - f = n + 1；



1. 先序遍历与后序遍历同 那么只能是空树
2. 中序遍历与后序遍历同 那么可能是空树 也可能只有左子树
3. 中序遍历与先序遍历同 可能空树 可能只有右子树
4. 中序遍历与层序同 空树或者 只有右子树

 ,

