jyang
@cs.ecnu.edu.cn

第五章
树

1 树的基本概念
2 树的存储结构
3 用树表示集合
4 树的遍历
5 树的线性表示
6 二叉树
7 二叉树的遍历
8 二叉树的顺序存储
9 穿线树和穿线排序
10 计算二叉树的数目

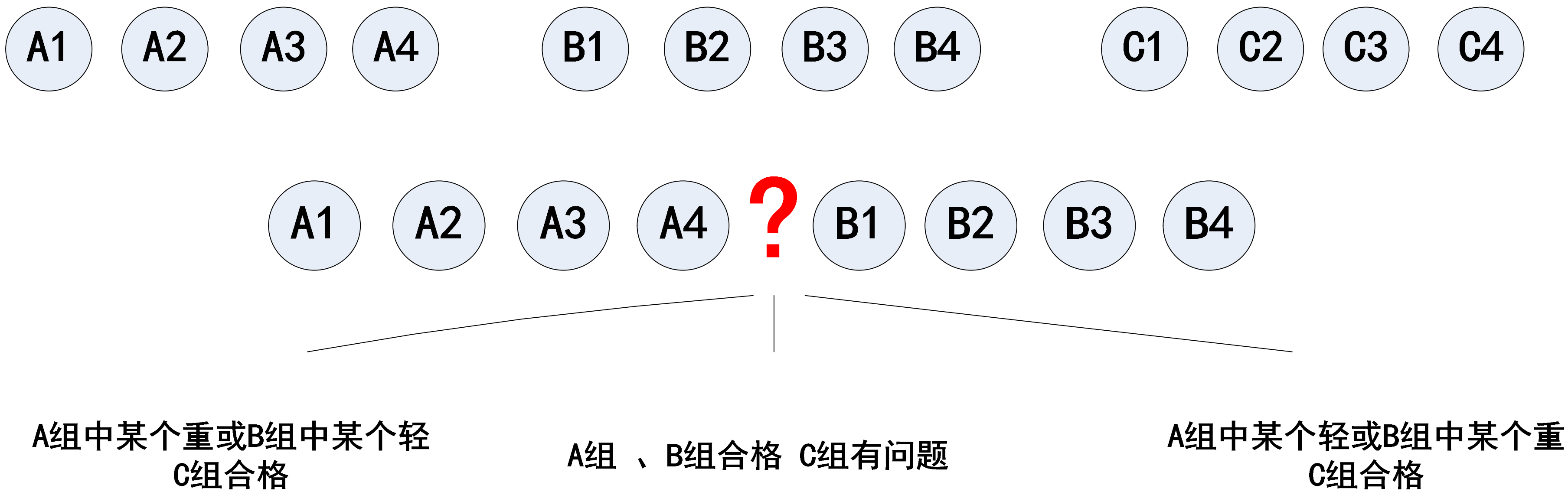
非线性结构

分支关系
层次特性

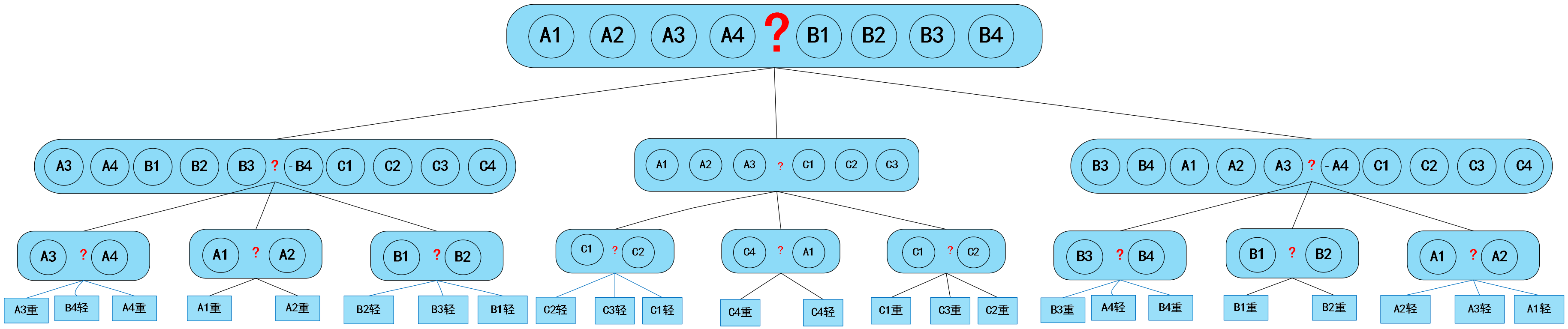
表示分支：

例：有12个小球，外观一致，在这12个小球里有一个是坏球，坏球和其它11个好球重量不同，但不知道是重还是轻，用一架天平只称三次，如何找出这个坏球并判断它轻或重？

例：有12个小球，外观一致，在这12个小球里有一个是坏球，和其它11个好球重量不同，但不知道是重还是轻，用一架天平只称三次，如何找出这个坏球并判断轻或重?



1 树的基本概念


A1 A2 A3 A4 ? B1 B2 B3 B4
A3 A4 B1 B2 B3 ? B4 C1 C2 C3 C4
A1 A2 A3 ? C1 C2 C3
B3 B4 A1 A2 A3 ? A4 C1 C2 C3 C4
A3 ? A4
A1 ? A2
B1 ? B2
C1 ? C2
C4 ? A1
C1 ? C2
B3 ? B4
B1 ? B2
A1 ? A2
A3重 B4轻 A4重
A1重 A2重
B2轻 B3轻 B1轻
C2轻 C3轻 C1轻
C4重 C4轻
C1重 C3重 C2重
B3重 A4轻 B4重
B1重 B2重
A2轻 A3轻 A1轻

jyang
@cs.ecnu.edu.cn

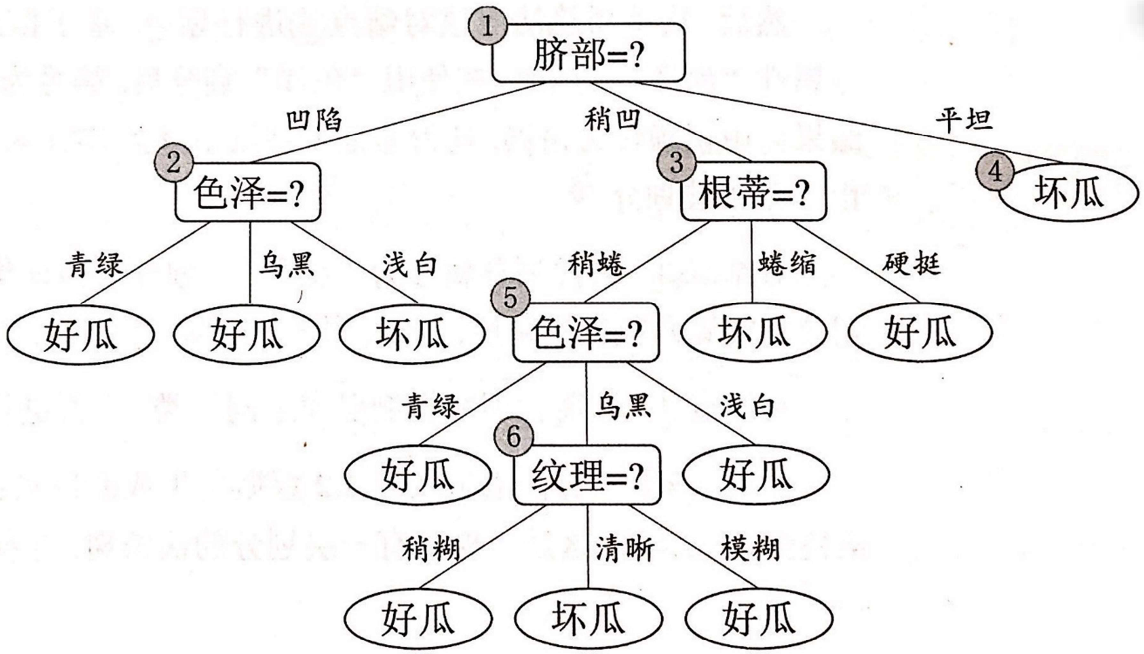
第五章
树

编号	色泽	根蒂	敲声	纹理	脐部	触感	好瓜
1	青绿	蜷缩	浊响	清晰	凹陷	硬滑	是
2	乌黑	蜷缩	沉闷	清晰	凹陷	硬滑	是
3	乌黑	蜷缩	浊响	清晰	凹陷	硬滑	是
4	青绿	蜷缩	沉闷	清晰	凹陷	硬滑	是
5	浅白	蜷缩	浊响	清晰	凹陷	硬滑	是
6	青绿	稍蜷	浊响	清晰	稍凹	软粘	是
7	乌黑	稍蜷	浊响	稍糊	稍凹	软粘	是
8	乌黑	稍蜷	浊响	清晰	稍凹	硬滑	是
9	乌黑	稍蜷	沉闷	稍糊	稍凹	硬滑	否
10	青绿	硬挺	清脆	清晰	平坦	软粘	否

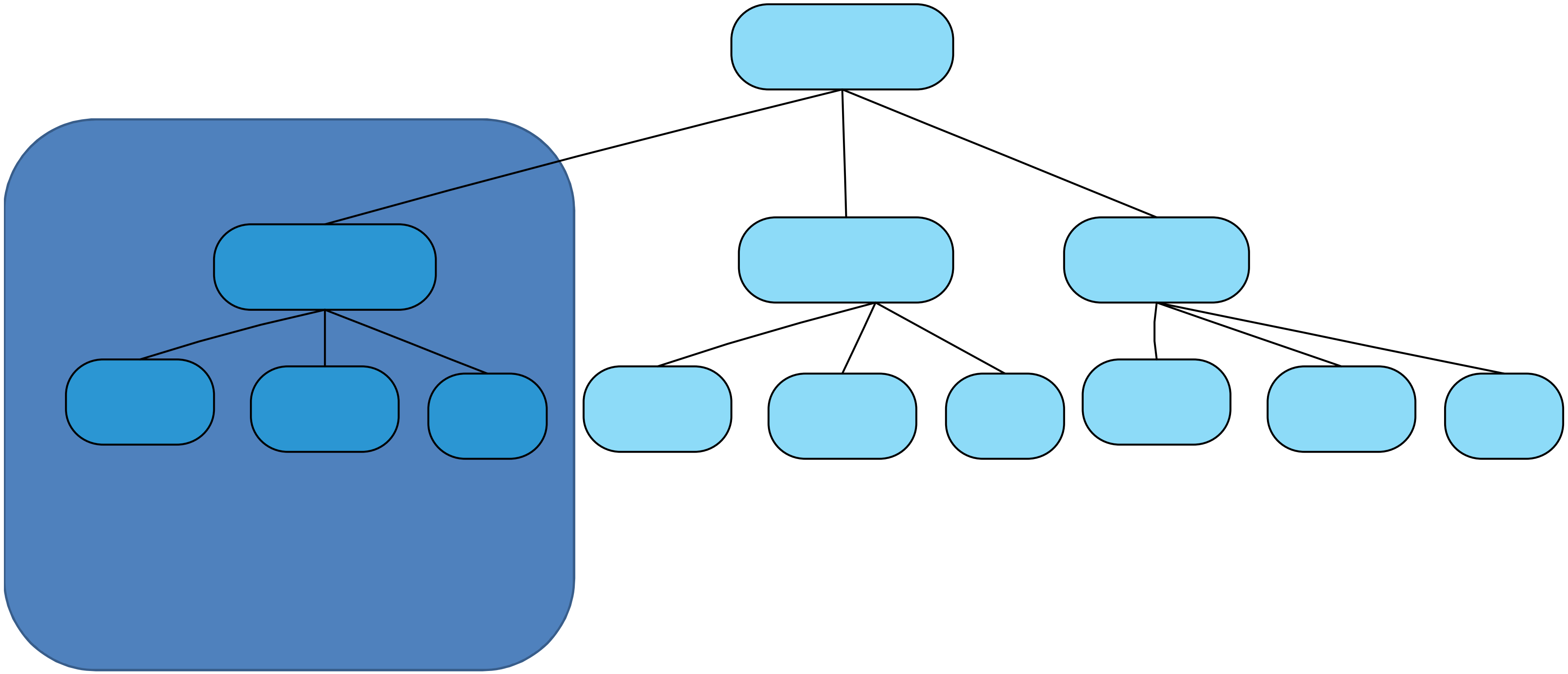
....

jyang
@cs.ecnu.edu.cn

第五章
树



决策树示例



1 树的基本概念

树的定义：

树是由一个结点或多个结点组成的有限集T，它满足下面两个条件：

（1）有一个特定的结点，称之为根节点（root）；

（2）其余的结点分成m（m>=0）个互不相交的有限集T_0，T_1,……T_2，T_{m-1}，其中每个集合都是一棵树，称为根结点的子树（subtree）。

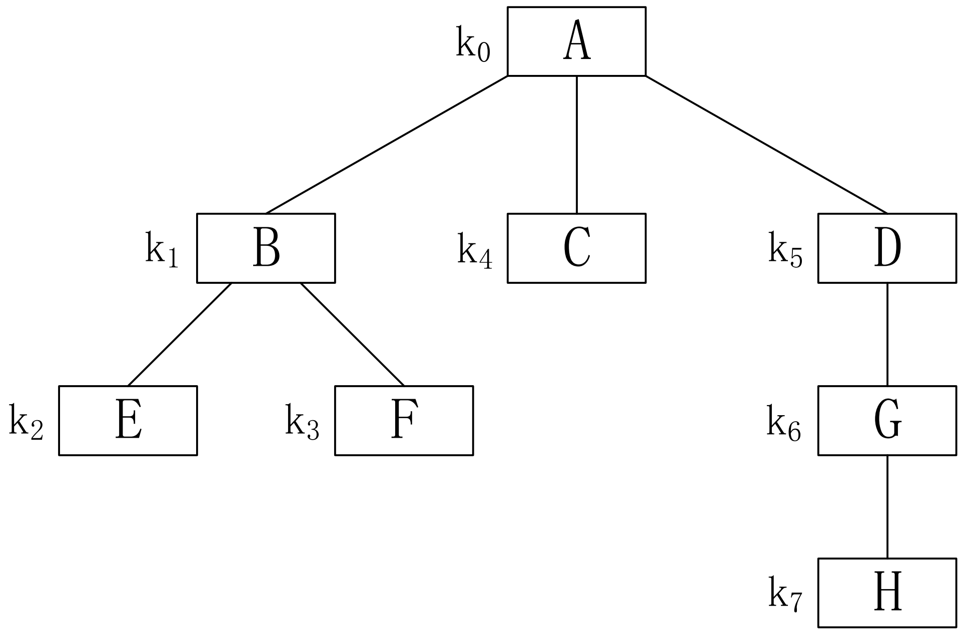
上面的定义是递归的。

一棵树至少有一个结点（根）。（二叉树没有这个要求）

例：由结点集合构成的树。结点集合T={k_0, k_1, k_2, ……, k_7}

k_0, 这棵树的根结点。
T中其余结点分成三个互不相交的有限集合，分别是：
T_0={k_1, k_2, k_3}
T_1={k_4}
T_2={k_5, k_6, k_7}

T_0，T_1，T_2本身又都是一棵树，也都是根结点k_0的子树。



其他相关定义：

结点的次数（也称为结点的度Degree）：一个结点的子树的个数为该结点的次数。

叶子结点（leaf）：次数为0的结点。即叶子结点没有子树。从树的定义可知，树中每一个非叶子结点至少有一棵子树。

树的次数（树的度）：树中各结点的次数的最大值。

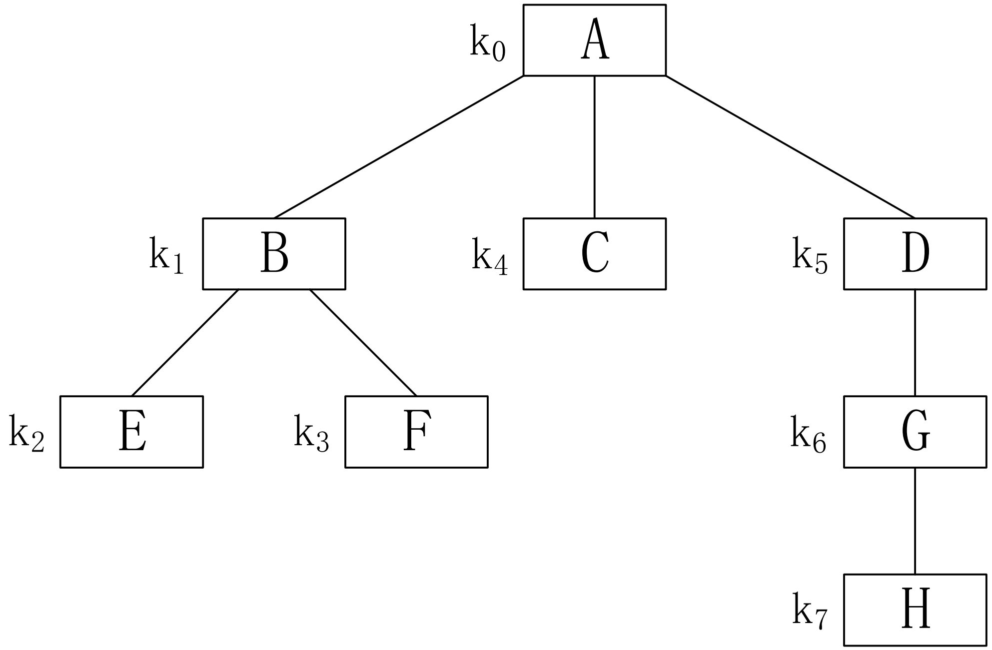
m次完全树：假设树T是一棵m次树，如果T中非叶子结点的次数都为m，那么称树T为一棵m次完全树。

其他相关定义

例：结点k_0 的次数是3
结点k1 的次数是2
结点k_5和k_6的次数都是1

叶子结点k_2, k_3, k_4, k_7的次数都是0

树的次数为3，不是完全树。



其他相关定义：

在用图形表示的树中，我们用线段连接两个相关联的结点。

结点的子树的根称为该节点的**子结点（或称为孩子结点）**。
相应的，该结点称为孩子的**父结点（或双亲结点，Parent）**。

如果结点k有两个或两个以上的子结点，那么称结点k的这些子结点为**兄弟结点**。

其他相关定义：

对于树中的任意两个不同的结点k_i和k_j，
如果从k_i出发能够“自上而下地”通过树中的结点到达结点k_j，
那么称k_i到k_j存在一条**路径**。

我们用路径经过的结点序列表示这条路径。

路径中所包含的边的数量称为路径长度（路径的长度等于这条路径上的结点个数减1）。

其他相关定义：

一棵树的根结点到树中的其余结点一定存在着路径。

如果从结点k_{i0}到结点k_{in}有路径（k_{i0}，k_{i1}，……k_{in}），则称结点k_{i0}，k_{i1}，……k_{in-1}都是**结点k_{in}的祖先（aucestor）**。这里约定，结点k_i的祖先不包括结点k_i本身。

结点k_{i1}，k_{i2}，……k_{in}都是结点k_{i0}的后代（descendant）。这里约定，结点k_i的后代不包括结点k_i本身。

其他相关定义：

结点的层次从根开始定义。一棵树的根结点所在的层次为0，而其他结点所在的层次等于它的父结点所在的层次加1。

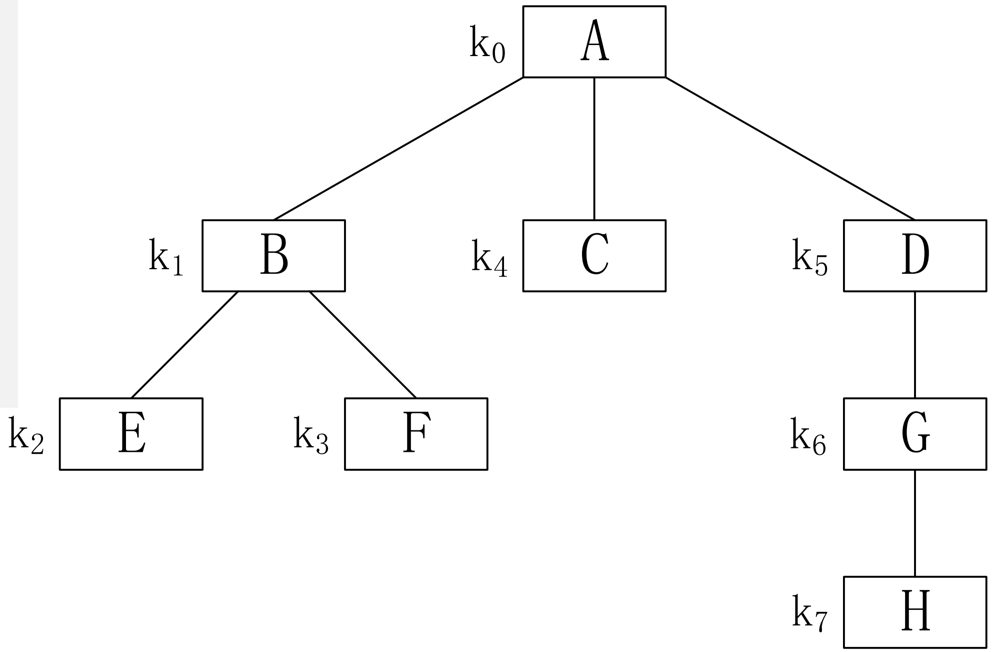
结点的层次：从根结点到树上某一结点k的路径所经过的结点的个数（包括根结点，但不包括k）。

树中层次最大的结点的层次称为**树的深度（Depth）或高度（Height）**。

其他相关定义

结点k_0 的层次是0
结点k_1 的层次是1
……

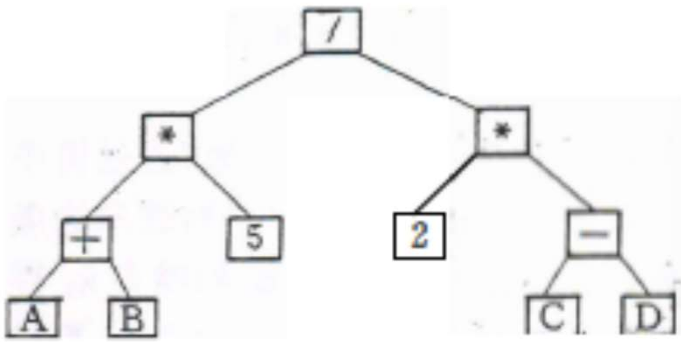
树的高度数为3



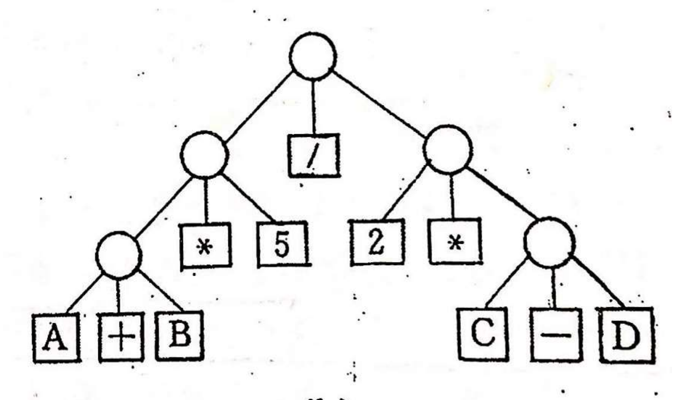
其他相关定义：

有序树： 如果在给定的m次树中，给树中的每个结点的每棵子树规定好它们的序号，那么称此树为有序树（ordered tree）。（在后面讨论的有序树中，我们通常是从左到右用整数0，1，2，……，m-1 给结点的各棵子树规定序号。

我们可以用有序树表示算术表达式。
（A+B）*5/(2*(C-D))



(a)



(b)

2 树的存储结构

树是非线性结构，必须把树中各结点之间存在的关系反映在存储结构之中。

- ☐ 标准形式
- ☐ 逆形式
- ☐ 扩充标准形式

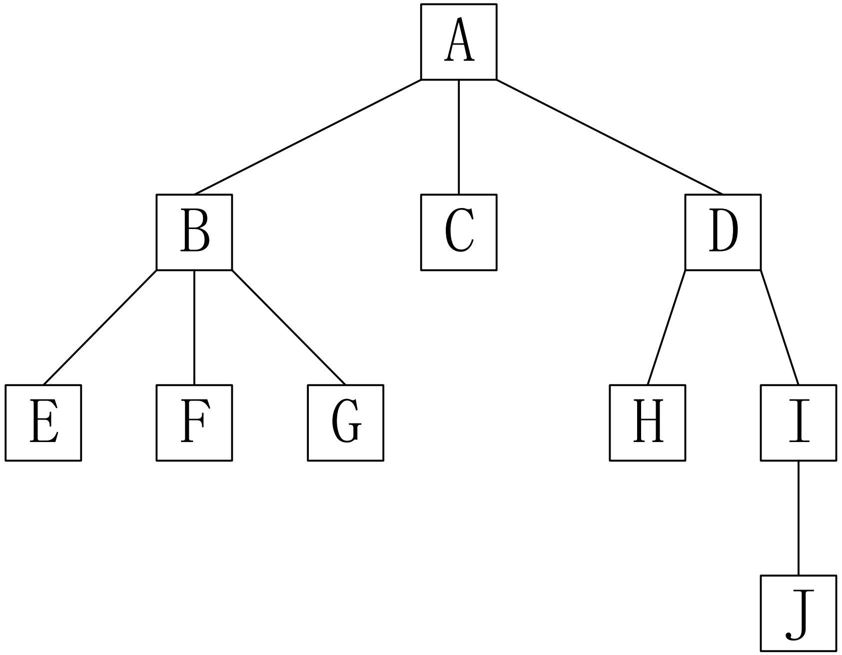
结点的一般形式为：

data	pointers

data：结点的值。通常包含一个关键字（或称为键）。

pointers：结点的指针部分，可以由若干个指针所组成。

一棵三次树



树的标准形式存储结构

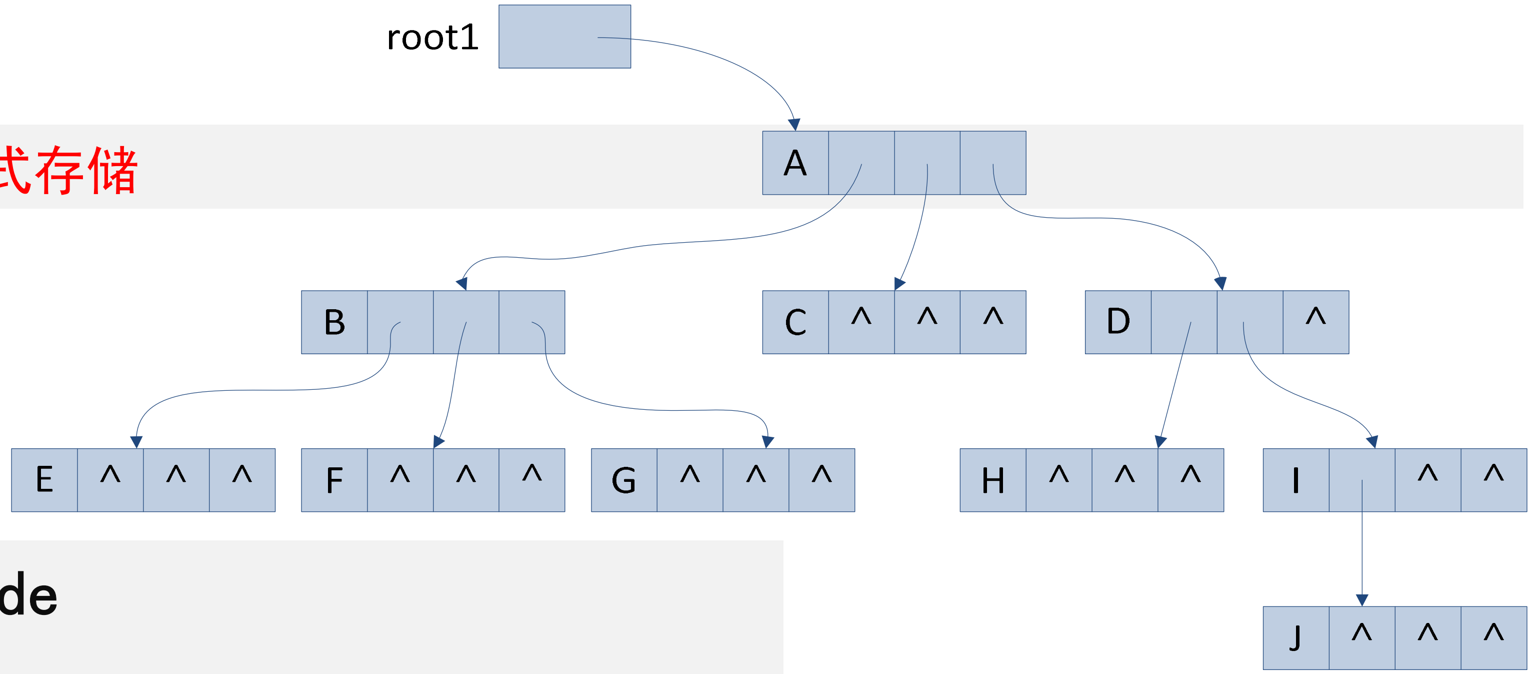
data	pointer

pointer：指针，用它指向子结点；pointer有m个字段，依次存放子结点（最多有m个）的所在地址。

更一般的形式为：

data	$Child_0$ ，$child_1$ ，……，$child_{m-1}$

树的标准形式存储




```
struct node
{
      char data;//数据部分
      struct node* child[MAXN];
};
typedef struct node NODE;

NODE* root1;
```

树的逆形式存储结构

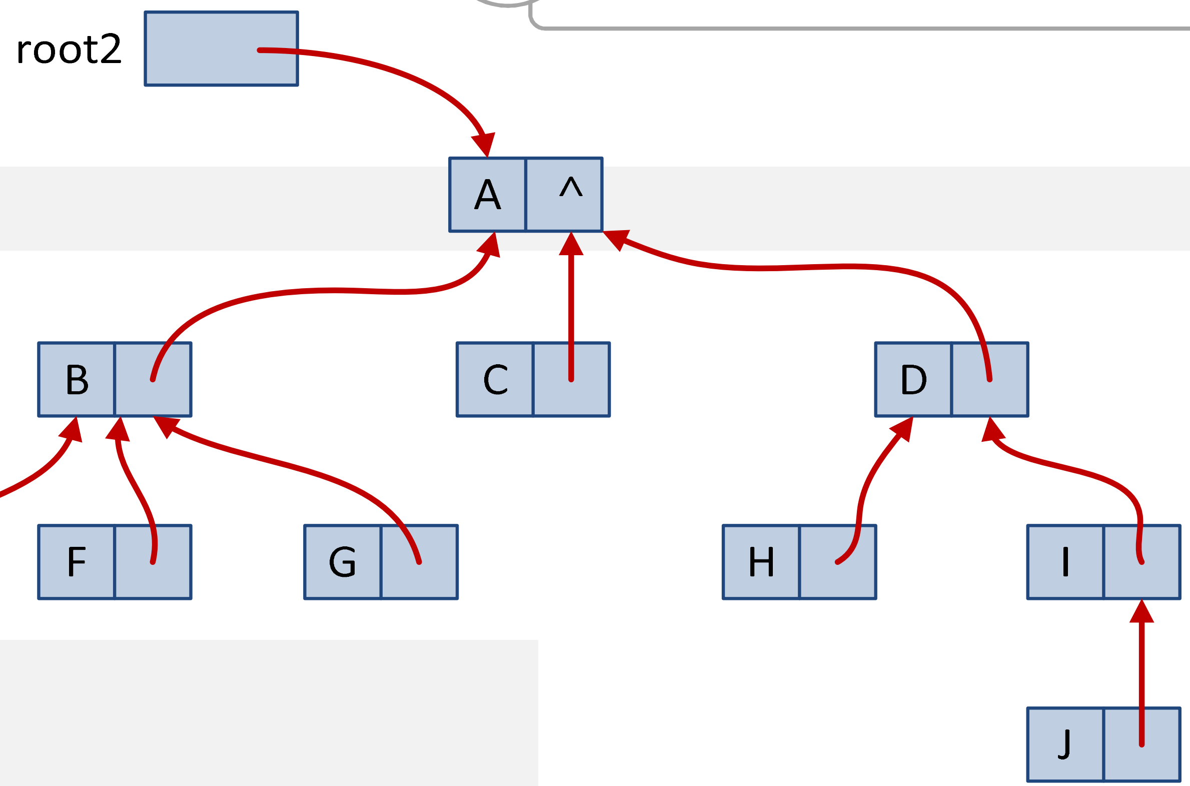
data	Parent

parent：指针，用它指向父节点；因树中结点最多有一个父节点，所以parent指针只需要一个字段，用它存放父结点所在地址。

注：根结点没有父结点，可在根结点的parent字段上填“空”。

2 树的存储结构

树的逆形式存储



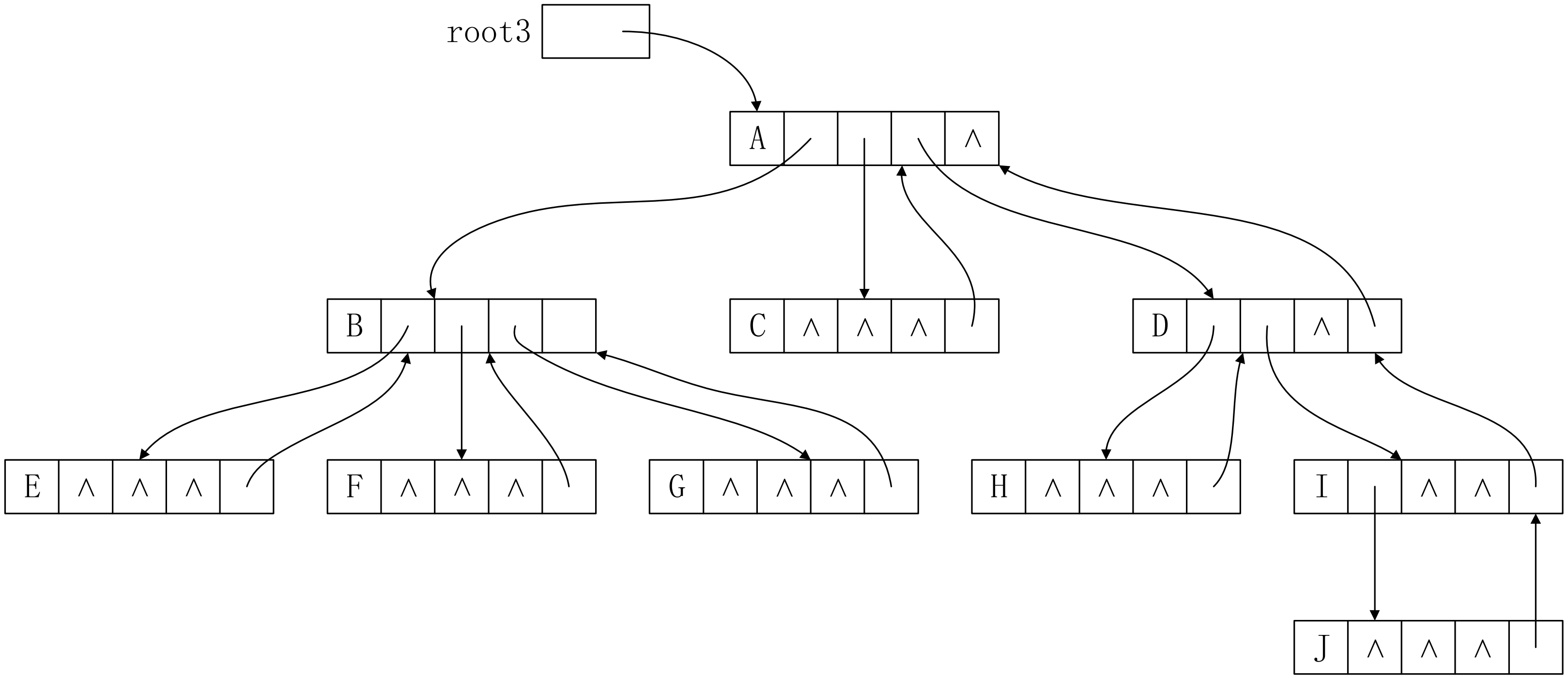

```
struct r_node
{
        char data; ;//数据部分
        struct r_node* parent;
};
typedef struct r_node R_NODE;
R_NODE* root2;
```

jyang
@cs.ecnu.edu.cn

树的扩充标准形式存储结构

```
struct e_node
{
        char data;//数据部分
        struct e_node* child[3];
        struct e_node* parent;
};
typedef struct e_node E_NODE;
E_NODE* root3;
```

data	$child_0$, $child_1$, …… , $child_{m-1}$	parent



问题：

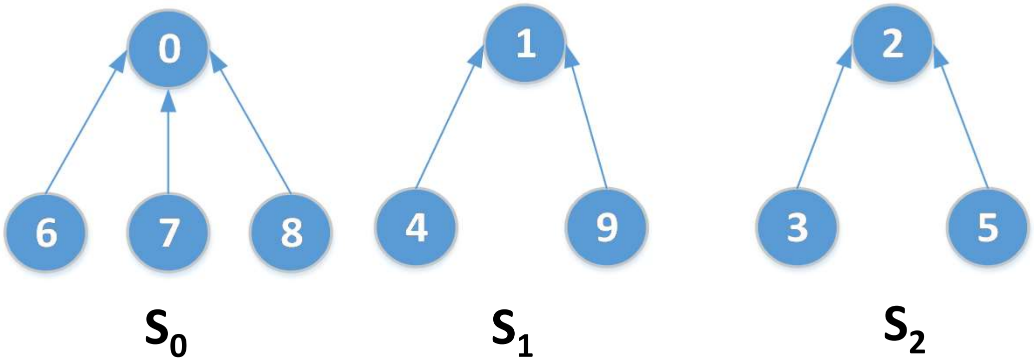
初始情况：3个集合，每个集合里的元素互不相交。（没有重复的元素）
{0, 6, 7, 8}, {1, 4, 9}, {2, 3，5}

假设需要完成的操作只有两类：
1）把两个不相交的集合合并。——UNION，并
2）给出一个元素，判断它属于哪个集合？——FIND ，查

假设：集合不相交

树结构在集合表示法中的应用——UNION和FIND

集合用树表示，如图

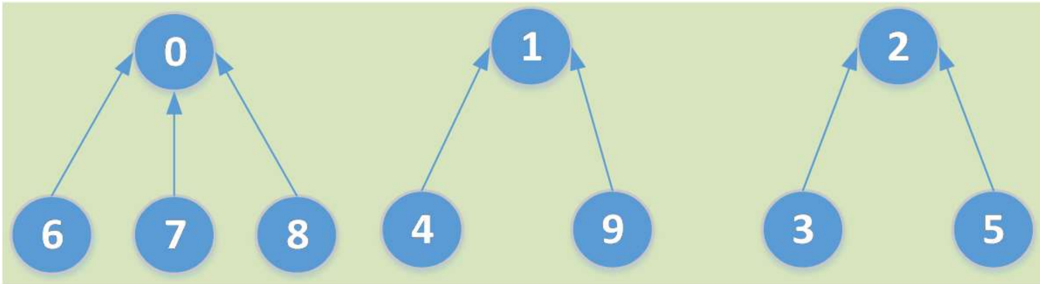


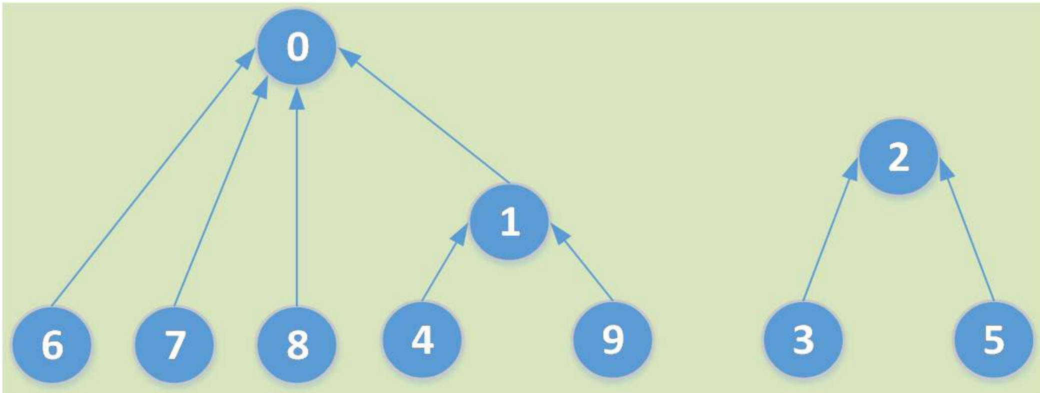
假定：所表示的集合都是不相交的，也没有重复元素。

假设：集合不相交

UNION（S_0， S_1）

UNION：首先用树表示两个集合
然后将其中的一棵树作为另一棵树的子树，用新获得的树表示这两个集合的并集。





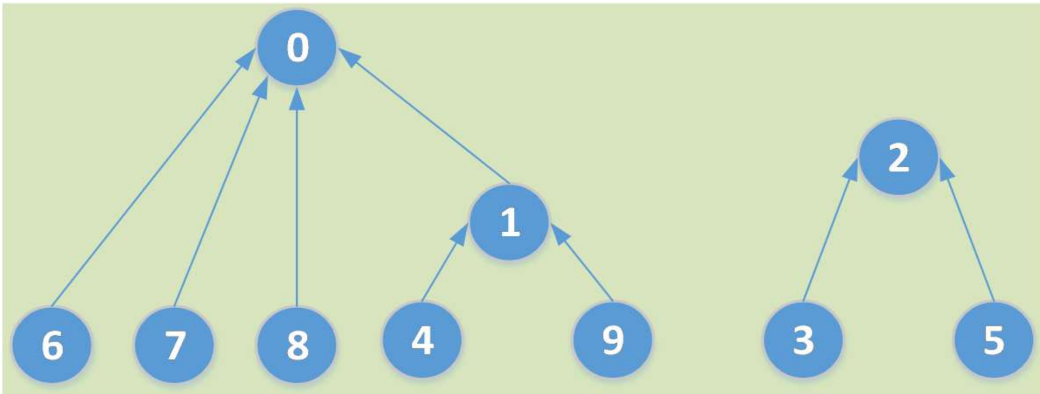
2 用树表示集合

假设：集合不相交

FIND：查找某个元素所在的集合
FIND（5），找到元素5 所在的树的根结点

判断元素i和元素j是否在同一个集合?

If （FIND（i）==FIND(j))



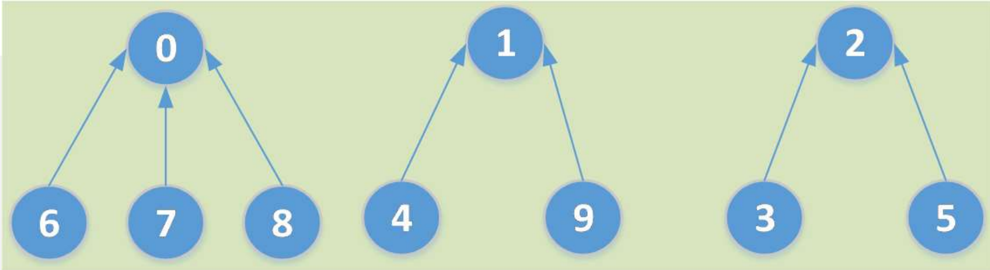
用树表示集合的简单实现

数据结构：树。用表示集合的树的根结点来标识集合。

存储形式：按逆形式进行存储。
每个结点（data部分，parent指针），Parent指针指向该结点的父结点。根结点没有parent，此字段可设NULL。

简化：用0, 1, ……, n-1对结点进行编号，取结点的编号作为结点值。用整数数组来存放所有的结点，使每个结点简化为只需要parent字段，省去data字段（数组下标就是data）。

```
int parent[MAXN]; //MAXN是元素个数的上限
```



用树表示集合的简单实现

```
int parent[MAXN];
............
//集合已表示好
int findSet(int parent[],int i) //i是任意结点
{
    while (parent[i]>=0)  i=parent[i];
    return(i);
}
```

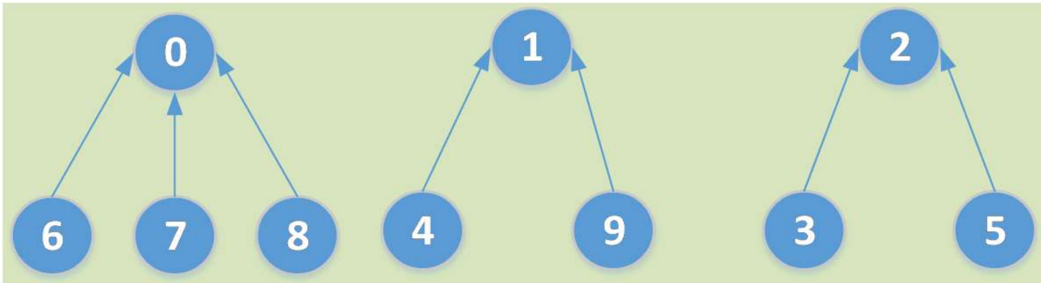
示例： if (findSet(1) == findSet(4))

用树表示集合的简单实现

```
int parent[MAXN]
int unionSet(int parent[],int i,int j)  //i必须是“根”
{   //if  (parent[i]!= -1)  return ERR
        parent[i]=j;
        return (j);
}
```

//如何判断i是否是根？？

//j是否需要也必须是“根”？



改进

```
int parent[MAXN]
int unionSet(int parent[], int i, int j)
{
        int root1, root2;
        root1= findSet(parent, i);
        root2= findSet(parent, j);
        parent[root1]=root2;
        return (root2);
}
```

用树表示集合的简单实现——效率分析

有可能会产生一棵“退化”的树——树退化为链表时，效率下降。

为了避免产生退化树的情况，可以对UNION（i，j）进行加权：
如果在树i中的节点数少于树j中的节点数，则把j作为i的双亲；
否则，就把i作为j的双亲。（结点多的做父结点）

为了使用加权规则，需要知道任意一棵树有多少个结点。
解决方案：在每棵树的根节点上，增加一个count字段

程序样例 P112

用树表示集合的简单实现——优化FIND

在find（i）时进行优化，如果j是从i到根结点的路径上的一个结点且s[i].parent 不等于i的根结点，则将i的根结点写入j的parent字段。

OJ1079-并查集.pdf

存储形式进一步节省：省去count字段，只用parent字段。如果某个集合的元素个数为t，那么置该集合的根结点的parent的值为（-t）。

树的遍历： 按某种次序获得树中的所有结点。

几种方法：

（1）树的前序遍历： 首先访问根结点，然后按前序遍历根结点的各棵子树。

（2）树的后序遍历： 首先按后续遍历根节点的各棵子树，然后访问根结点。

（3）树的层次遍历： 首先访问处于第0层上的根结点，然后访问处于第一层上的结点，再访问处于第二层上的结点，再依次访问以下各层上的结点。

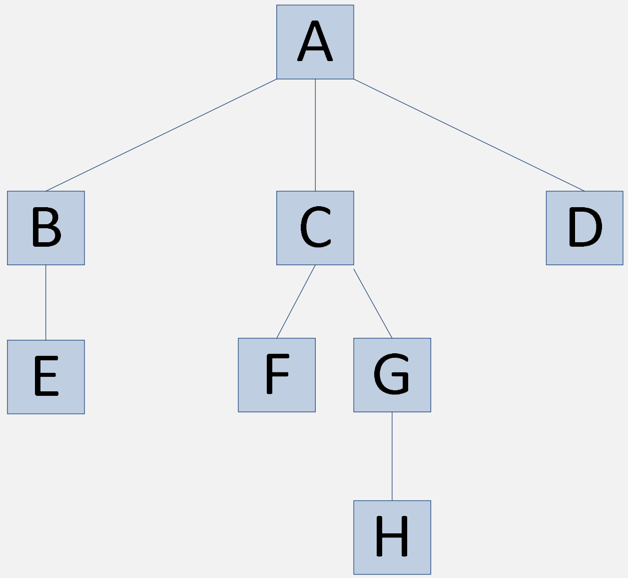
jyang
@cs.ecnu.edu.cn

4 树的遍历

树的遍历： 按某种次序获得树中的所有结点。如果约定访问树中的结点时，总是从左到右，则给定树，结点序列唯一。

例子：

前序遍历的结点序列：ABECFGHD
后序遍历的结点序列：EBFHGCDA
层次遍历的结点序列：ABCDEFGH



问题：给定结点序列，树是否唯一？

例：前序序列：ABC ？
前序序列：ABC + 层次序列：ABC ？
前序序列：ABC + 后序序列：CBA ？

用递归程序实现树的前序遍历

```
//m次树 标准存储形式中结点的常用结构
struct node
{
    char data; //结点的标识
    struct node * child[MAXN];
};
Typedef struct node NODE;
void r_preorder(NODE *t, int m) //递归前序遍历, t:根, m: 次数
{
        int i;
        If (t!=NULL)
        {       printf( “%c” ,t->data);
                for (i=0;i<m;i++)    r_preorder(t->child[i],m);
        }
}
```

非递归程序实现树的前序遍历

```
void s_preorder((NODE *t, int m)
{
  NODE * s[MAXN];//栈：用于保存尚未遍历树的根节点的地址
  int top,i;
  if (t==NULL) return;    //t为空则停止
  S[0]=t;  //入栈
  top=1;  //top: 下一次入栈的位置
  while (top>0)//栈不空
      {     t=s[--top];  //栈顶结点出栈,
            printf( "%c",t->data); //访问当前"根"节点
            for (i=m-1;i>=0;i--)  //从右向左依次入栈
                  if(t ->child[i] != NULL)
                        s[top ++] = t ->child[i];
      }
}
```

实现按层次遍历使用一个顺序存储的队列存放还没有处理的子树的根结点的地址

```
void levorder(NODE *t, int m)
{  NODE *q[100],*p ;
   int head,tail,i;
   if (t==NULL)        return;
   q[0]=t;  head=0;  tail=1;  //队列
   while (head<tail)
       {      p=q[head++];  //出队
              printf( “%c” ,p->data);
              for (i=0;i<m;i++)   //从左到右连续入队
                     if(p ->child[i] != NULL)
                            s[tail++] = p ->child[i];
       }
}
```

```
void post_order(NODE   *root) //m次树的非递归后序遍历（选学）
{
            Node *stack[MAXN],*p;
            int mark[MAXM], top = -1,   j;
            stack[++top] = root;
            mark[top] = 0;
            while(top>=0)
            {
                        p = stack[top];
                        if(mark[top]==0&&p->child[0]!=NULL)
                        {
                           mark[top] = 1;
                           for(j=MAXM-1;j>=0;j--)
                           {          if(p->child[j]!=NULL)
                                         {   stack[++top] = p->child[j];
                                              mark[top] = 0;
                                         }
                           }
                        }
                        if(stack[top]->child[0]==NULL||mark[top]==1)
                               printf（“%c”，stack[top--]->data)；
            }
}
```

5 **树的线性表示**

应用实例：树的结点序列输入计算机中，从而建立树的结构。

两种树的线性表示：

- 树的层号表示
- **树的括号表示**
- 树的双亲表示法

树的线性表示——层号表示

树中结点的层号：如果k是树中的一个结点，那么我们为结点规定一个整数lev（k），它满足下面两个条件：
（1）如果k’是k的子结点，那么lev（k‘）>lev(k)；
（2）如果k’和k’’同是结点k的子结点，那么lev（k‘）=lev(k‘’)；
称lev（k）是结点k的层号。

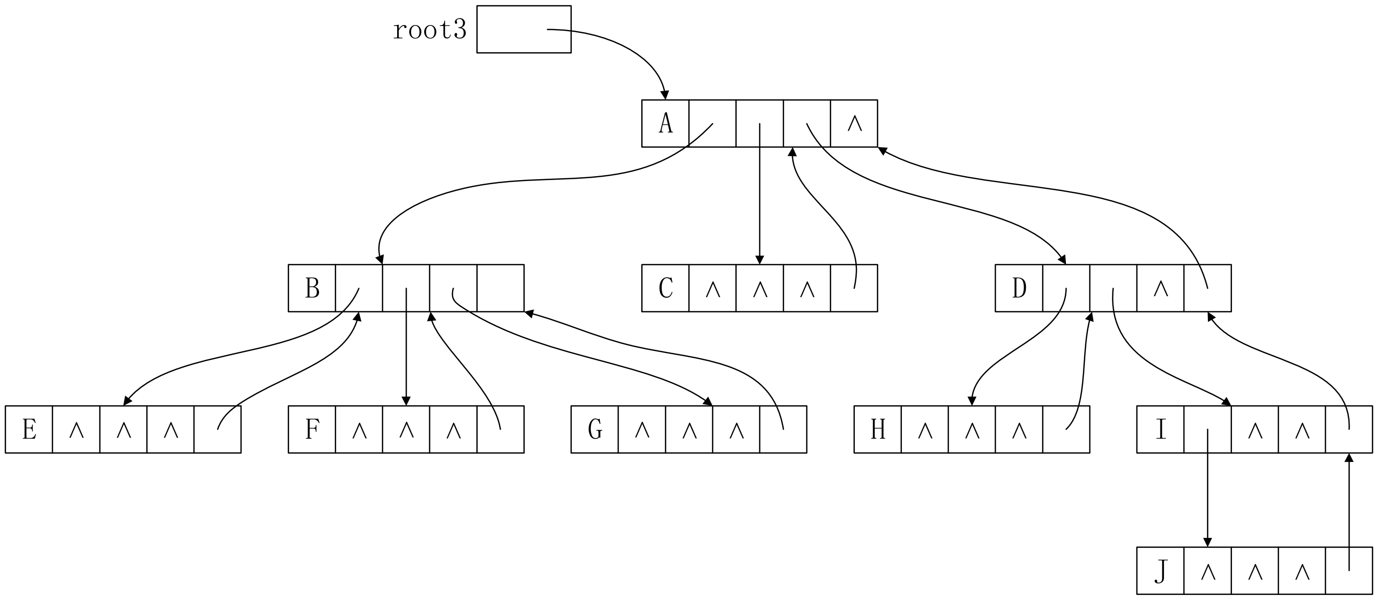
“结点的层号”与“结点所在的层次”

结点的层次：结点所在的层次是唯一的。
结点的层号：不唯一，满足层号的两个条件的整数都可以作为结点的层号。

树的层号表示：按前序写出树中全部结点，并在结点之前带上结点的层号。

树的线性表示——层号表示

给定树的层号表示，如何建立一棵按扩充标准形式存储的m次树？



树的层号表示：前序，并在结点之前带上结点的层号。

树的线性表示——层号表示

给定树的层号表示，如何建立一棵按扩充标准形式存储的m次树？

```
typedef struct node {
        int lev;
        char data;
        struct node *child[MAXM];  //m次树
        struct node * parent;
}NODE;
```

树的线性表示——层号表示

给定树的层号表示，如何建立一棵按扩充标准形式存储的m次树？
关键：正确处理子结点、父结点的指针。

```
//读入第一个结点，是树根。(前序序列中，第一个一定是树根)
//依次读入剩下的结点
//     构造结点，为child[i]填入初值NULL
//     找到当前结点的父结点，处理当前结点的parent指针，
//     处理父结点的child指针
```

jyang
@cs.ecnu.edu.cn

树的线性表示——层号表示

```
NODE * lev_tree(输入数组intputTree, m, n)  //m次树，共有n个结点
{
        ............//判断输入，如果不合法，返回NULL
        //处理根结点
        NODE * root, *p, *q;
        int i, j;
        root = (NODE*)malloc(sizeof(NODE));
        root->data = inputTree[0].data;
        root->lev = inputTree[0].lev;
        for (j = 0 ; j< m ; j ++)  root->child[j] = NULL;
         root->parent=NULL;
        ............
```

树的线性表示——层号表示

```
//根结点之后处理剩余结点。关键：正确处理子结点、父结点的指针。
p=root;
for(i=1;i<n;i++) //依次读入剩下的结点
{
      q=……;   // 构造结点
      for (j=0;j<m;j++)  q->child[j]=NULL; //为child[i]填入初值NULL
//    找到当前结点的父结点，处理当前结点的parent指针，//
      处理父结点的child指针
}
```

树的线性表示——层号表示

```
//读入第一个结点，是树根。
for(i=1;i<n;i++) //依次读入剩下的结点
{
        q=……;  //构造结点，为child[i]填入初值NULL
        for (j=0;j<m;j++)  q->child[j]=NULL;
        while (q->lev <= p->lev) p=p->parent; //找到当前结点的父结
点,while循环结束后，p指向q的parent
//  q:当前结点，p: 上一次处理过的结点。
//(1)大于：按照前序的定义，q是p的子结点。此时while循环不执行；
//(2)等于：如果两者lev相同，说明他们是兄弟结点，则p指向他自己的父
节点之后，q是p的子结点；
//(3)小于：“p=p->parent;”，目的：向“上”。直到(2)的情况满足
//      处理q的parent指针，
//      处理父结点的child指针
}
```

树的线性表示——层号表示

```
//读入第一个结点，是树根。
for(i=1;i<n;i++) //依次读入剩下的结点
{
        q=……;  //    构造结点，为child[i]填入初值NULL
        for (j=0;j<m;j++)  q->child[j]=NULL;
        while (q->lev <= p->lev) p=p->parent; //找到当前结点的父结点
        q->parent=p;//处理当前结点的parent指针，
        j=-1;    //    处理父结点的child指针
        while(p->child[++j]!==NULL) ;
        p->child[j]=q;
        p=q;
}
Return (root);
```

树的线性表示——括号表示

树T的括号表示的规则：

（1）如果树T只有一个结点，则此结点就是它的括号表示。

（2）如果树T是由根节点A和子树T0，T1，……，Tm-1组成，则树T的括号表示是：

A（T_0的括号表示，T_1的括号表示，…，T_{m-1}的括号表示）

δT =A（δB， δC)

=A（B （δD，δE，δF)，C(δF ））

=A（B （D， E（H，I），F（J）)，C(G(K，L) ））

树的线性表示——括号表示

给定树的括号表示，如何建立一棵按扩充标准形式存储的m次树？

A（B （D, E（H, I）, F（J）), C(G(K, L)))

//读入第一个结点，是树根。
//依次读入剩下的结点
// 构造结点，为child[i]填入初值NULL
// 找到当前结点的父结点，处理当前结点的parent指针，
// 处理父结点的child指针

树的线性表示——括号表示

给定树的括号表示，如何建立一棵按标准形式存储的m次树？

```
Ch=a[0];
While ( ch!= '\0' )
{      if (isalpha(ch)) { //构造结点p，处理p的child指针初值}
       else
       switch (ch)
       { case' (' : //括号前面读入的“p” 是接下来结点的父结点
         case  ',' ://前面的“p” 是当前子树上的一个结点
         case  ')' ://前面的“p” 是当前子树的最后一个节点
       }
}
```

树的线性表示——括号表示

给定树的括号表示，如何建立一棵按扩充标准形式存储的m次树？

```
Ch=a[0];
While ( ch!= '\0' )
{       if (isalpha(ch)) { //构造结点p，处理p的child指针初值}
        else
        switch (ch)
        {       case' (' : stack[top++]=p; break;
                                        //前面读入的p是接下来结点的父结点，接
                                              下来要处理的是p的若干子树
                ......
        }
}
```

树的线性表示——括号表示

```
    switch（ch）
    {     ……
          case  ‘,’ : //p是当前子树上的一个结点,它的父结点当前在栈顶
                q=stack[top-1];//注意不出栈!
                q指向p的父结点，处理child指针 ，
                break;
}
```

树的线性表示——括号表示?

```
…………………
        switch（ch）
        {       ……
                case  ‘)’ :q=stack[--top];
                            处理q的child ，指向p
                            p=q;
```

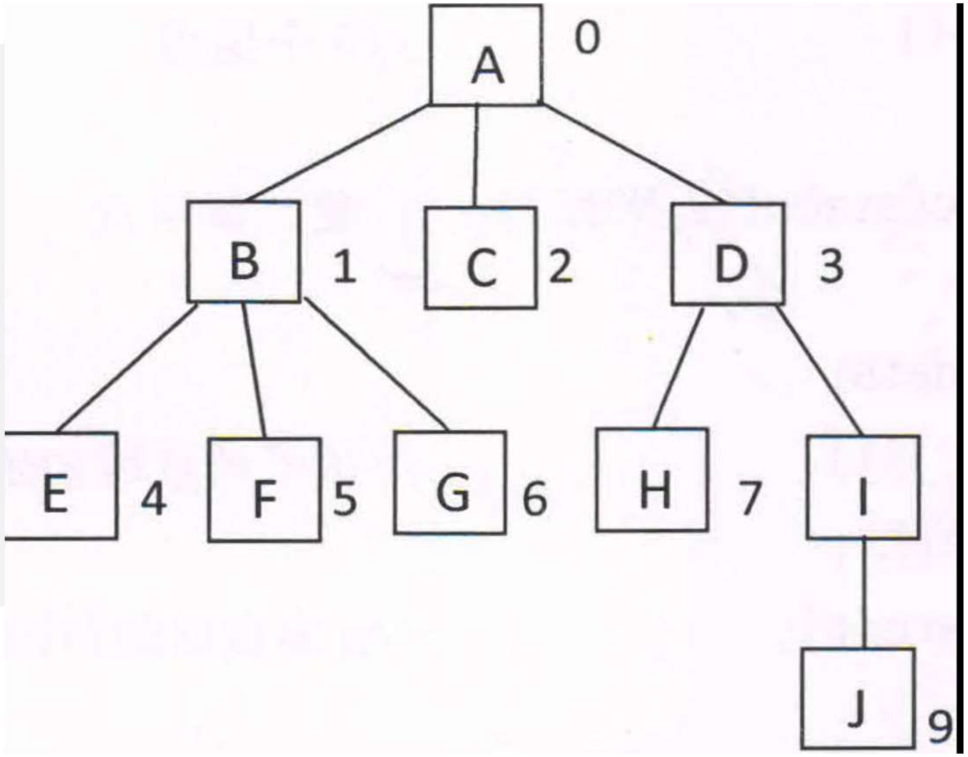
//与“）”对应的“（”前的结点是正在处理的子树的根，现在在栈顶。当前的p是当前子树的最后一个节点,所以，找到它的父结点q，出栈，q的child指向p。以q为根的子树全部处理完成，因此让p=q；p总是指向等待处理的结点

```
        }
}
```

树的线性表示——双亲表示

树中最重要的逻辑关系：
双亲结点与孩子结点

给出结点序列，
同时给出每个节点的parent



parent	-1	0	0	0	1	1	1	3	3	8
data	A	B	C	D	E	F	G	H	I	J
	0	1	2	3	4	5	6	7	8	9

树的最重要的逻辑关系——parent

//每个结点，找到parent，修改parent的child的指针指向自己

```
typedef struct node{
    char data;
    int parent;
}NODE;

typedef struct link_node {
char data;
struct link_node *child[MAXM];
}LINK_NODE;
```

树的最重要的逻辑关系——parent

```
LINK_NODE *creat_tree_fromParent(NODE inputTree[], int m, int n)
{
    int i, j;
    LINK_NODE *root, *p, *q;
    if (n < 1) return NULL;

//生成根结点，根结点赋值，孩子结点域初始化为空
    root = (LINK_NODE*)malloc(sizeof(LINK_NODE));
    root->data = inputTree[0].data;
    for (i = 0 ; i < m ; i ++) root->child[i] = NULL;

//用一个数组addr_NODE来记录每个结点的指针
        addr_NODE[0]= root;
……
}
```

树的最重要的逻辑关系——parent

```
LINK_NODE *creat_tree_fromParent(NODE inputTree[],int m,int n)
{……
//用一个数组addr_NODE来记录每个结点的指针
addr_NODE[0]= root;
 for (i = 1 ; i < n ; i ++) //最多n个结点
    {
        q =(LINK_NODE*)malloc(sizeof(LINK_NODE)); //生成新结点，q
        q->data = inputTree[i].data;
        for (j = 0 ; j < m ; j ++) q->child[j] = NULL;
        addr_NODE[i]= q;
       //找到q的父亲p，将p的第一个空的child字段指向q
        p = addr_NODE[inputTree[i].parent];
        j=-1;
        while (p->child[++j]!= NULL);
        p->child[j]=q;
    }
    return root;}
```

树的最重要的逻辑关系——parent

```
LINK_NODE *creat_tree_fromParent1(NODE inputTree[],int m,int n)
{     int i,j,last=-1;
      for (i = 0 ; i < n ; i ++)
      {
          addr_NODE[i]=(LINK_NODE*)malloc(sizeof(LINK_NODE));
          addr_NODE[i]->data = inputTree[i].data;
          for (j = 0 ; j < m ; j ++) addr_NODE[i]->child[j] = NULL;
      }
      for(i = 1 ; i < n ; i ++)
      {
          j=-1;
          while (addr_NODE[inputTree[i].parent]->child[++j]!= NULL);
          addr_NODE[inputTree[i].parent]->child[j]=addr_NODE[i];
      }
      return addr_NODE[0];
}
```

树的最重要的逻辑关系——parent

```
LINK_NODE *creat_tree_fromParent1(NODE inputTree[],int m,int n)
{     int i,j,last=-1;
      for (i = 0 ; i < n ; i ++)
      {
          addr_NODE[i]=(LINK_NODE*)malloc(sizeof(LINK_NODE));
          addr_NODE[i]->data = inputTree[i].data;
          for (j = 0 ; j < m ; j ++) addr_NODE[i]->child[j] = NULL;
      }

      for(i = 1 ; i < n ; i ++)
       {  if (last!=inputTree[i].parent)    j=0;
         else    j ++;
         addr_NODE[inputTree[i].parent]->child[j]=addr_NODE[i];
         last = inputTree[i].parent;
      }
return addr_NODE[0];
}
```

树的最重要的逻辑关系——parent
（上机练习）
//用链式结构来存储子结点的信息（由于树的“次数”未知）

```
typedef struct linkchild{
    int treeIndex;//程序用一个数组tree来存放所有结点的指针，treeIndex表示本结点在tree数组中的下标,
    struct linkchild* next;
}LINKCHILD;

typedef struct treenode{
    int data;  //结点的值
    CHILDLINK* childhead;
}TREE_NODE;

TREE_NODE* tree[100000];
int parent[100000];
```

树的最重要的逻辑关系——parent

//用链式结构来存储子结点的信息（由于树的“次数”未知）

```
void createTree( )  //输入保存在parent数组中，数组下标就是结点的值。
{    int i, j;
     for (i = 0; i < n; i++)
         { //将每个结点构造好（结点地址写入tree数组），childhead 初始化
          tree[i] = (TREE_NODE*)malloc(sizeof(TREE_NODE));
          tree[i] -> data = i;
          tree[i] -> childhead = NULL;
         }
     //根据题目提示，根结点一定是第一个。接下来处理从第二个结点开始的结点的“父子”关系
     for (i = 1; i < n; i++) {
          LINKCHILD * p = malloc(sizeof(LINKCHILD));
          p -> treeIndex = i; //当前结点的index值是“i”，当前结点的父节点是
tree[parent[i]]
          p -> next = (tree[parent[i]]) -> childhead; //链表插入
          (tree[parent[i]]) -> childhead = p; //链表插入
     }
```

树的最重要的逻辑关系——parent

//用链式结构来存储子结点的信息（由于树的“次数”未知）

```
void postorder(TREE_NODE *t)
{
        CHILDLINK *p;
        if (t!= NULL)
        {
                p=t->childhead;
                while (p!=NULL)
                {
                        postorder( tree[p->treeIndex] );
                        p = p->next;
                }
                printf("%d ",t->data);
    }
}
```

6 二叉树

二叉树：一个有限的结点集合，这个集合或者为空；或者由一个根结点及表示根结点的左、右子树的两个互不相交的结点集合所组成，而根结点的左、右子树也都是二叉树。

我们称用空集表示的二叉树为空的二叉树。空的二叉树不含有结点。

二叉树是有序树，把第一个和第二个子结点（或子树）分别称为左子结点和右子结点（或子树）。严格区分左右子树。

把任意次树转换成二叉树

把具有m个子结点k_0，k_1，……，k_{m-1}的结点k转换成以k_0作为结点k的左子结点，并且k_{i+1}作为k_i（i=0, 1, …，m-2）的右子结点。

更一般的转换方法：
若T=（T_0，T_1, ……, T_{m-1}），是m（M>0）棵树的序列，得到与T相对应的二叉树β（T）的方法如下：
（1）如果m=0，那么β（T）为空的二叉树；
（2）如果m>0，那么β（T）的根结点就是T_0的根结点， β（T）的根结点的左子树是β（A_0，A_1, ……，A_{r-1}），其中A_0，A_1, ……，A_{r-1} 是T_0的根节点的子树； β（T）的根结点的右子树是 β（ T1, ……, Tm-1 ）。

把任意次树转换成二叉树

例：P123

左——子结点
右——兄弟结点

二叉树的遍历：层次遍历方法与一般有序树的完全相同。除此之外常见的方法：

（1）按前序遍历二叉树：

首先访问根结点，
然后按前序遍历根节点的左子树，
最后按前序遍历根结点的右子树。

（2）按中序遍历二叉树：

首先按中序遍历根节点的左子树；
访问根结点，
最后按中序遍历根结点的右子树。

（2）按后序遍历二叉树：

首先按后序遍历根节点的左子树；
然后按后序遍历根结点的右子树
最后访问根结点。

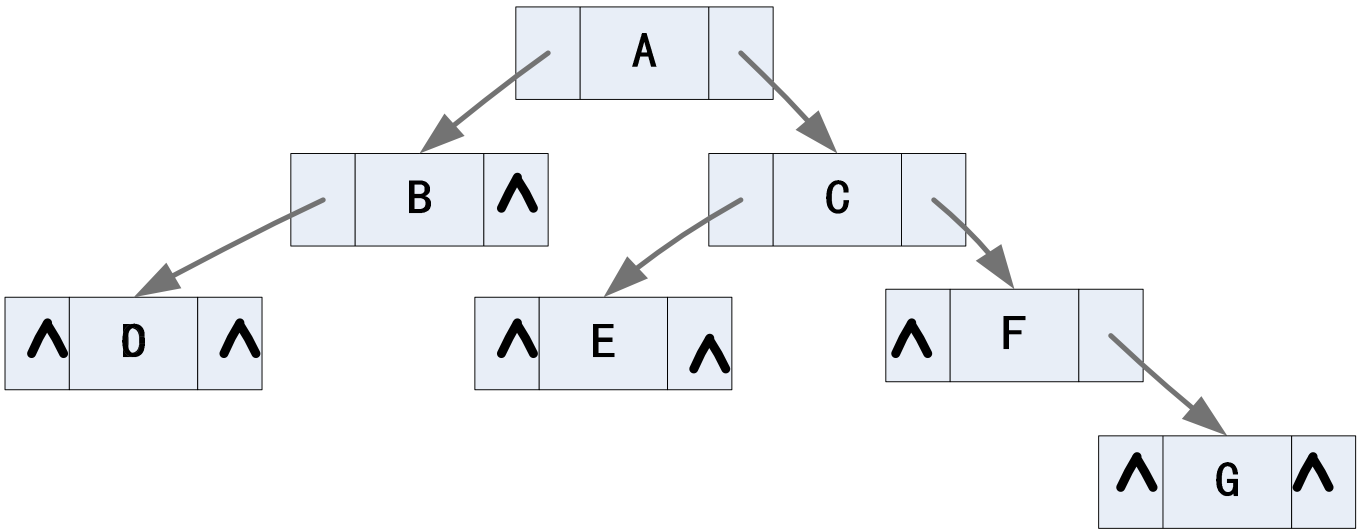
二叉树遍历算法的实现

前序、中序、后序 都可以用递归程序方便的实现

```
typedef struct Binode{  char data;
                        struct Binode  *lchild;
                        struct Binode  *rchild;
}NODE;
void r_midorder(NODE *t)  //中序遍历
{
    if(t != NULL){
        r_midorder(t ->lchild);  //中序遍历左子树
        printf("%c", t ->data);  //访问结点
        r_midorder(t ->rchild);  //中序遍历右子树
    }
}
```

二叉树遍历算法的实现

非递归实现中序



二叉树遍历算法的实现——非递归实现中序

中序的第一个结点：树“最左边” 的一个结点(“投影法”)。
方法：
(如果根结点不为空，则根结点进栈，访问左子树；
每个子树的根结点都入栈，一直到左儿子为空；
栈顶元素出栈。访问出栈的结点。
)
按上述方法访问出栈结点的右子树。

栈空或没有新的子树，则结束

```
typedef struct Binode{  char data;
                        struct Binode  *lchild;
                        struct Binode  *rchild;
}NODE;  //二叉树的结点

typedef struct snode{   NODE *addr;
                        struct snode *link;
}SNODE; //用于实现链接存储的栈，栈中存放二叉树上某个结点的地址
```

```
//初始化
SNODE *p=NULL,*top=NULL;
```

```
//入栈
p=(SNODE*)malloc(sizeof(SNODE));
p->addr = ……;    //NODE *……;
P->link = top;
top=p;
```

```
//出栈
……=top->addr;
p=top;
top= top->link;
free(p);
```

```
typedef struct Binode{  char data;
                        struct Binode  *lchild;
                        struct Binode  *rchild;
}NODE;  //二叉树的结点

typedef struct snode{   NODE *addr;
                        struct snode *link;
}SNODE; //用于实现链接存储的存队列
```

```
//初始化
SNODE *head=NULL,*tail=NULL;
SNODE *p;
```

```
//head端  队首出队
if (head==NULL) 队空，出队失败;
……=head->addr;
p=head;
Head = head->link;
free(p);
```

```
// 队尾进队

p=(SNODE*)malloc(sizeof(SNODE));
p->addr = ……;    //NODE *……;
P->link = NULL;
if (head==NULL) head=p;
        else tail->link=p;
tail=p;
```

7 二叉树的遍历

```
void s_midorder(NODE *t)
{ SNODE *top=NULL,*p;
   while (t!=NULL || top!=NULL )
  {   while (t!=NULL) //子树根连续入栈，沿左孩子向下。
      {
          p=……; p->addr=t;      // “t” 入栈
          p->link=top; top=p;
          t=t->lchild;
      }
      if (top!=NULL)
      {   t=top->addr;  //工作指针t，指向栈顶元素指向的结点
          printf(……);//访问结点输出 t指向的结点

          p=top; top=top->link; free(p); //出栈

          t=t->rchild;//接下来处理出栈结点的右子树
      }
  }
}
```

二叉树遍历算法的实现

非递归实现前序：前序遍历的第一个被访问的结点是根结点，然后访问左子树，最后访问右子树。

根入栈。
重复：出栈，输出出栈结点；先把该结点右子树根入栈，再把左子树根入栈。

栈空时，说明所有的子树都已被遍历，函数结束

二叉树遍历算法的实现

非递归实现前序

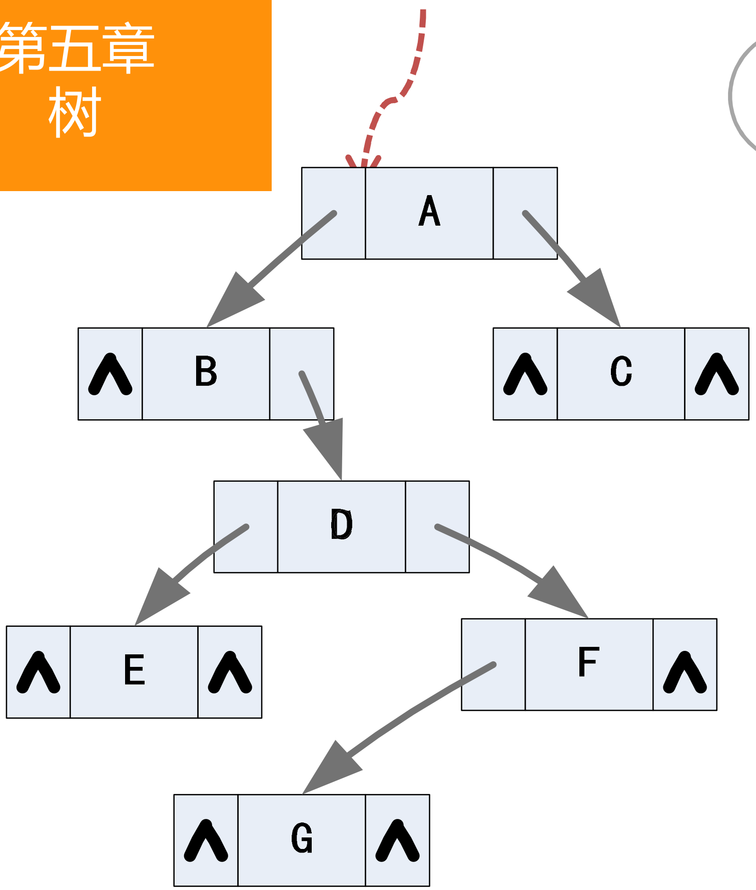
```
s.push(t);
While (s!=NULL)
{
        p=s.top();
        s.pop;
        输出 p;
        if (p->right!=NULL )  s.push(p->rchild);
        if (p->light!=NULL )  s.push(p->lchild);
}
```

二叉树遍历算法的实现

非递归实现后序（可以借鉴m次树的后序非递归实现）
留做练习

jyang
@cs.ecnu.edu.cn

7 **二叉树的遍历**



一个结点会被经过三次：
到达（到达之后去向左子树），
左子树处理完毕从左子树回来（之后去向右子树），
右子树处理完毕从右子树回来（之后“向上”）

沿着虚线箭头，得到序列：

A B B D E E E D F G G G F F D B A C C C A

二叉树遍历算法的实现——逆转链接指针

//第一次到达一个结点（输出），去访问左子树

//左子树处理完，回到左子树的parent（输出），将去访问右子树

//右子树处理完毕，从右子树回来（输出），

//向上回溯

二叉树遍历算法的实现——逆转链接指针（不使用栈或递归）

当沿着结点k准备向结点k的左（或右）子树“向下”作进一步遍历时，就改变结点k的左（或右）指针，使之指向结点k的双亲结点，为以后“向上”回溯提供路径。

在遍历了结点k的左（或右）子树后，必须“向上”回溯，同时还要恢复结点k的左（或右）指针。

有两个问题：其一，需要找到父节点，其二，需要告知父节点，本节点是左子节点还是右子节点。

二叉树遍历算法的实现——逆转链接指针

```
typedef struct binarytree{
    char data;
    struct binarytree *lchild;
    struct binarytree *rchild;
    int tag; //初始为0，当要沿着rchild向下移动时，置tag=1
}TNODE;
```

二叉树遍历算法的实现——逆转链接指针

```
//  q: 当前结点,
//p:当前结点的parent,
//r: 用于“探索”或临时使用

q=t;
p=NULL;
//第一次到达一个结点。（输出），访问左子树
输出q;
r=q->lchild; //沿左指针向下，可能出现的情况：r空或r不空
        //r空：可以看做左子树处理完毕，从空的左子树回到当前结点
        //r不空：继续向下，向下之前，为“回”做好准备
```

jyang
@cs.ecnu.edu.cn

二叉树遍历算法的实现——逆转链接指针

```
q=t;
p=NULL;
label1:  if(i==1) printf (…q…) //第一次到达时输出，实现前序
        r=q->lchild;//沿左指针向下，如果空，则“回到”q
        if (r!= NULL )//不空
                {       q->lchild=p;//逆转指针
                        p=q; //指针向下一层移动
                        q=r; //指针向下一层移动，q指向新到达的结点
                        goto label1;
                }
//从左子树回来
```

二叉树遍历算法的实现——逆转链接指针

```
Q=t;
P=NULL;
label1:  if(i==1) printf (q->data) //若第一次到达时输出, 实现前序
         r=q->lchild;//沿左指针向下, 如果空, 则“回到”q
         if (r!= NULL )//不空
                  {       q->lchild=p;//逆转指针
                          p=q; //指针向下一层移动
                          q=r; //指针向下一层移动, q指向新到达的结点
                          goto label1;
                  }
//从左子树回来。回到q, 若在这里输出, 实现中序。
//去向右子树, 即沿右子树指针向下。右指针空或不空
```

二叉树遍历算法的实现——逆转链接

```
label1:  if(i==1) printf ( q->data ) //第一次到达时输出，实现前序
……
Label2: if(i==2) printf ( q->data ) //q的左指针空或从左子树回来，第二次到达q，如果在这里输出，则实现中序
        r=q->rchild; //去向右子树
        if  (r!=NULL )
                {       q->tag=1; //从右指针向下，标志位置1
                        q->rchild=p;//逆转指针
                        p=q;
                        q=r;
                        goto babel1;//q为从右指针到达的新结点
                }
```

二叉树遍历算法的实现——逆转链接

```
label1:  if(i==1) printf (…q…) //第一次到达时输出，实现前序
……
Label2: if(i==2) printf (…q… ) //q的左指针空或从左子树回来，第二次到达q，如果在这里输出，实现中序
       r=q->rchild;
       if  (r!=NULL )
              {      q->tag=1; //从右指针向下，标志位置1
                     q->rchild=p;//逆转指针
                     p=q;
                     q=r;
                     goto babel1;//q为从右指针到达的新结点
              }
//if r=NULL，则从右子树回来 :
Label3: if(i==3) printf (q)  //q的右指针空或从右子树回来，第三次到达q，在这里输出，则实现后序遍历。
```

```
Label3: if(i==3) printf (q) //q的右指针空或从右子树回来，第三次到达q
//向上回溯
        if (p!=NULL )     //  P 是当前结点q的父亲，
                if (p->tag==0)  //q是从p的左指针进入的
                {
                        r=p->lchild;//恢复指针，P的左指针指向q
                        p->lchild=q;
                        q=p;//p和q都向上一层
                        p=r;
                        goto label2;//指针恢复完后，从左子树回到q
                }
                else//q是从p的右指针进入的，恢复p的指针和tag,
                {       p->tag=0;
                        r=p->rchild;
                        p->rchild=q;
                        q=p;
                        p=r;
                        goto label3; //指针回复完后，从右子树回到q
        }
```

二叉树遍历算法——应用

树的定义是递归的，用递归程序描述树的操作较为方便。基于树的遍历递归程序，可以实现一些树的基本操作。

- 求结点的数量
- 求叶子结点的数量
- 求树的高度
- 复制二叉树
- 判断两棵给定的二叉树是否等价

二叉树遍历算法的实现——应用

求树的高度：

- 空树的高度为-1；
- 只有一个根结点，高度为0
- 若不空，它的高度等于：max{左子树深度，右子树深度}+1

二叉树遍历算法的实现——应用

求树的高度：

```
int BiTreeDepth(NODE *t)
{
      if (!t) return (-1);
      else
      {
       if ((!t->lchild)&&(!t->rchild)) return (0);
       else
            hL= BiTreeDepth(t->lchild);
            rL= BiTreeDepth(t->rchild);
            return （max{hL,rL}+1）;
      }
}
```

二叉树遍历算法的实现——应用

求结点的数量

- 空树：0
- 非空树：左子树结点数量+右子树结点数量 + 1

二叉树遍历算法的实现——应用

求结点的数量

```
int count (NODE  *root)
{
        if (root==NULL)  return (0);
        lcount=count(root->lchild);
        rcount=count(root->rchild);
        return (lcount+rcount+1);
}
```

二叉树遍历算法的实现——应用

求叶子结点的个数

- 空树：0
- 只有一个根节点：1
- 有子树：左子树叶子结点个数 + 右子树叶子结点个数

二叉树遍历算法的实现——应用

求叶子结点的个数

```
void CountLeaf(NODE *t, int &count)
{
        if (t)  //递归程序，先考虑好结束条件
        {
                if ((!t->lchild)&&(!t->rchild)) //t是叶子结点
                        count++;
                CountLeaf(t->lchild, count);
                CountLeaf(t->rchild, count);
        }
}
```

二叉树遍历算法的实现——应用

复制二叉树：复制根，复制左子树，复制右子树

```
NODE *BiTreeCopy(NODE *t) t;
{      NODE *p;
       if (t==NULL) return (NULL);
       else {      p=(NODE *) malloc(sizeof(NODE));
                   p->data=t->data;
                   p->lchild=BiTreeCopy (t->lchild);
                   p->rchild=BiTreeCopy (t->rchild);
            }
}
```

二叉树遍历算法的实现——应用

判断两棵给定的二叉树是否等价：假设t1和t2是两棵二叉树，如果t1和t2都是空的二叉树；

或t1和t2的根节点的值相同，并且t1和t2的根结点的左、右子树分别是等价的，那么我们称二叉树t1和t2是等价的。

二叉树遍历算法的实现——应用

```
int equaltree(NODE *t1,*t2)
{      if t1空并且t2空 return (1);
       if  (t1不空且t2不空)
         if  (t1和t2的数据字段相等)
            if  ( equaltree(t1->lchild,t2->lchild) )
               return(equaltree(t1->rchild,t2->rchild);

       teturn (0);
}
```

二叉树的性质：

（1）如果从0开始计数二叉树的层次，则在第i层最多有 2^i 个结点。（i ≥0）；

（2）高度为h的二叉树，最多有$2^{h+1}-1$个结点（ h≥0 ）；

（3）任意一棵二叉树，如果其叶结点有n_0个，次数为2的非叶结点有n_2个，则有$n_0=n_2+1$；

（4）具有n（n>0 ）个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor$

注：二叉树的高度：沿用树的定义，只有一个根节点的时候，高度为0，空二叉树高度为-1

二叉树的性质：

（3）任意一棵二叉树，如果其叶结点有n_0个，次数为2的非叶结点有n_2个，则有$n_0=n_2+1$；

设次数为1的结点有n_1个，总结点个数为n，总边数为e，则：

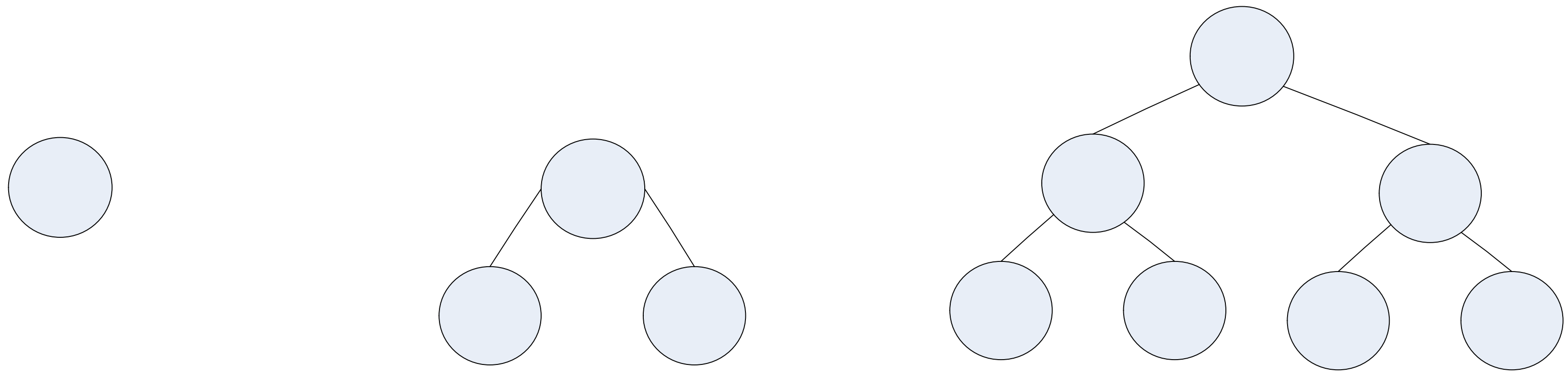
$$n=n_0+n_1+n_2,$$
$$e=2*n_2+n_1=n-1$$

所以， $n_0=n_2+1$

6 二叉树

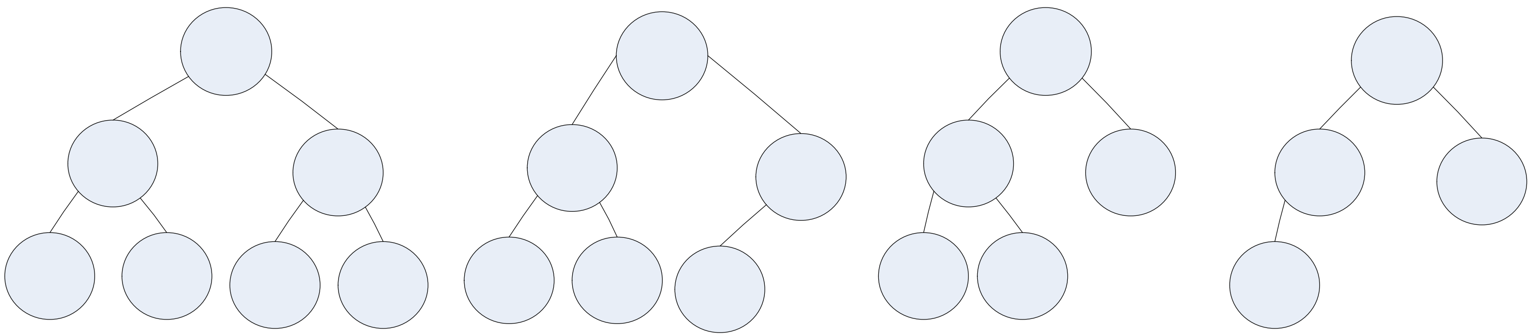
满二叉树：一类特殊的二叉树，在满二叉树中，每一层结点数目都达到了最大。

高度为k的满二叉树有$2^{k+1}-1$ 个结点。

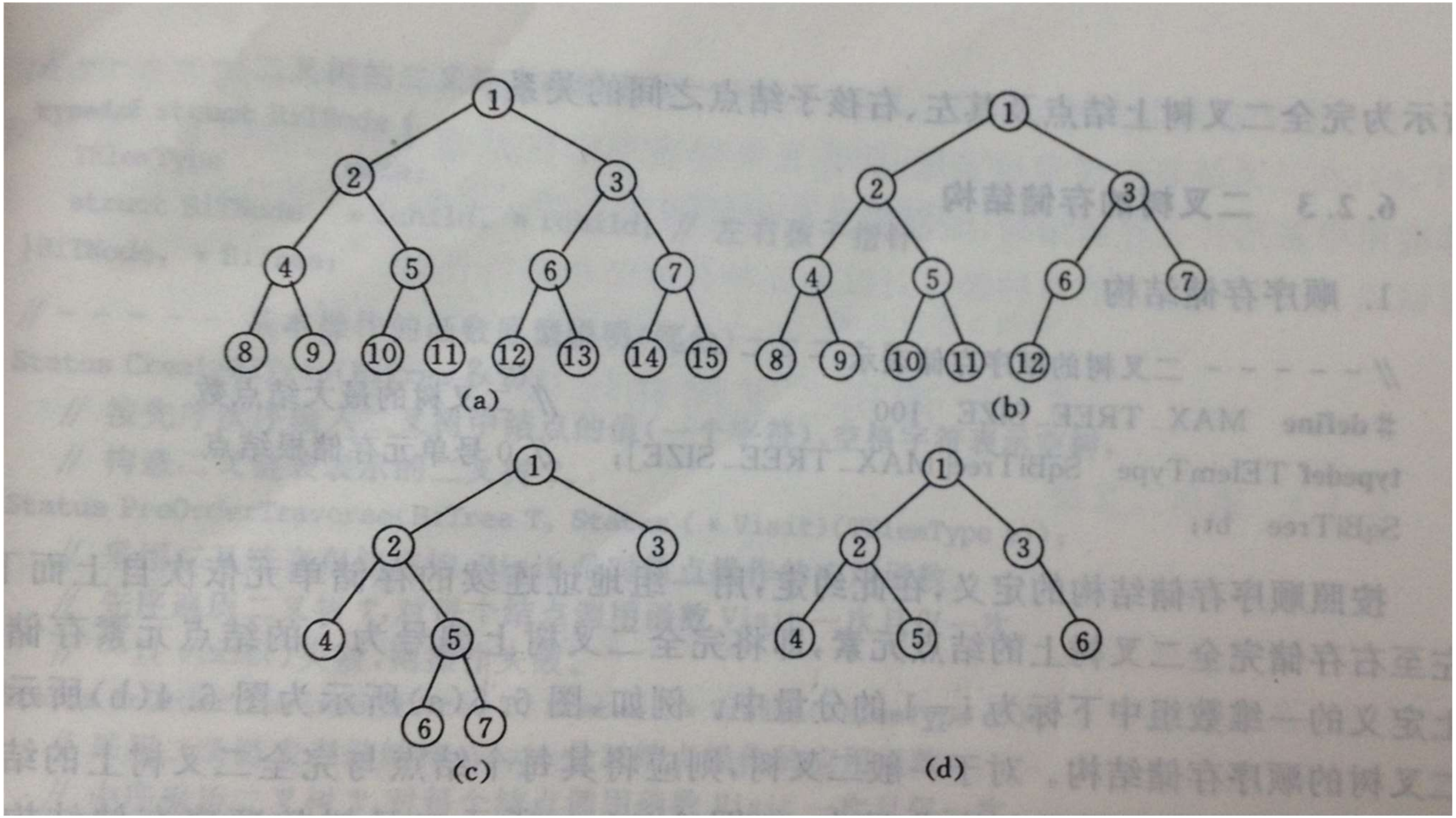


完全二叉树：对于一棵二叉树，除最后一层外，其他各层的结点个数都达到最大，最后一层则从右向左**连续缺**若干个结点。

完全二叉树具有如下性质：具有n（n>0）个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor$



6 二叉树



完全二叉树具有如下性质：具有n（n>0）个结点的完全二叉树的深度为

$$\lfloor \log_2 n \rfloor$$

证明：

设深度为k，$2^k-1<n<=2^{(k+1)}-1$

$2^k<=n<2^{(k+1)}$，即 $k<=\log_2 n<k+1$

k是整数，所以$k=\lfloor \log_2 n \rfloor$

如果一棵**有n个结点的完全二叉树**中的结点按照层次自顶向下，层内自左相右的顺序连续编号为 0，1，2，3，……，n-1，

则这些编号之间有以下关系：

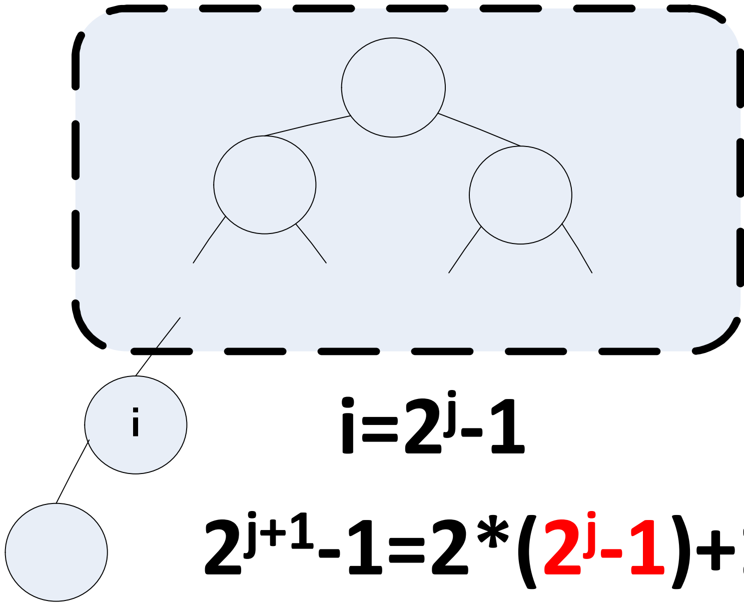
如果i=0，则结点i为根结点。
如果i>0，则结点i的父节点的编号为 $\lfloor (i-1)/2 \rfloor$

如果2i+1<n,则结点i的左孩子为2i+1，如果2i+2<n,则结点i的右孩子为2i+2。

如果i为偶数，且i不等于0，则结点i的左兄弟为i-1,如果i为奇数，且i不等于n-1，则结点i的右兄弟为i+1,

如果i=0，则结点i为根结点。
如果i>0，分两种情况讨论。
（1）第j层的第一个结点的编号为i，则 $i=2^j-1$

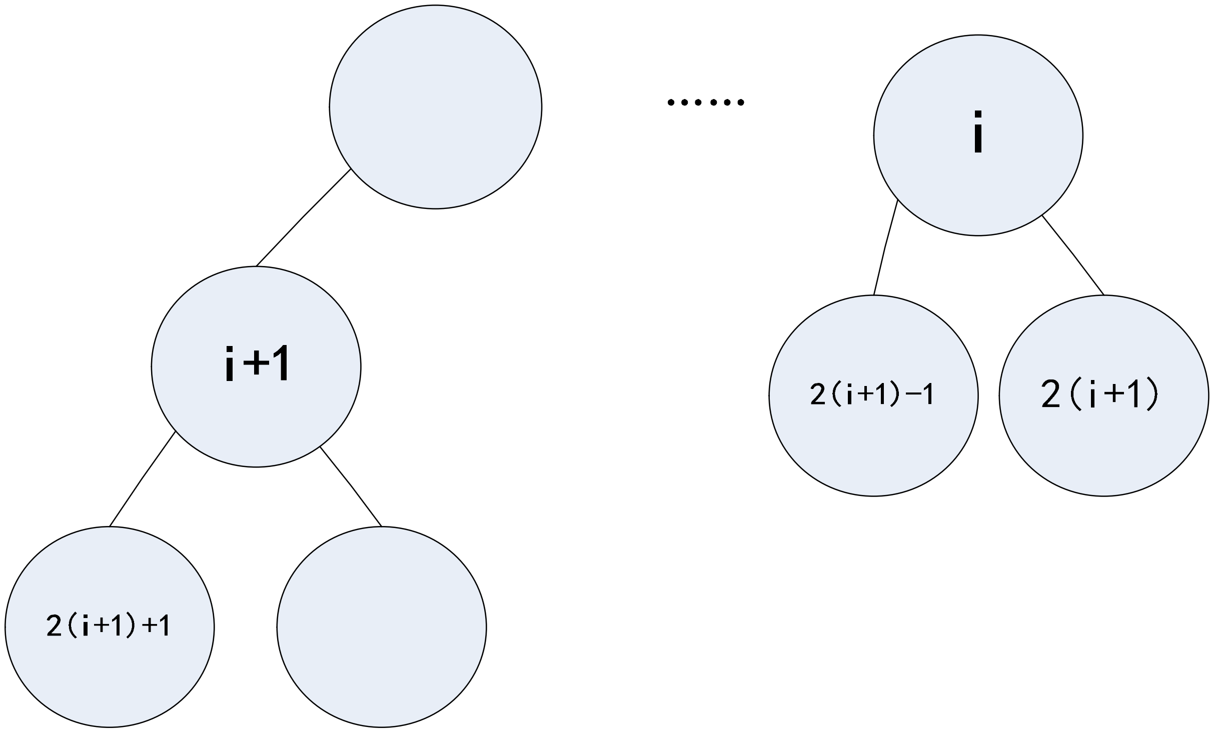
最右结点的编号	本层满后结点个数	层号
0	1	0
2	3	1
		……
2^j-2	2^j-1	j-1
$2^{j+1}-2$	$2^{j+1}-1$	J
		J+1



$$i=2^j-1$$

$$2^{j+1}-1=2*(2^j-1)+1=2i+1$$

如果i=0，则结点i为根结点。
如果i>0，分两种情况讨论。
（1）第j层的第一个结点的编号为i
（2）第j层的最后一个结点的编号为i



8 二叉树的顺序存储

树最常用的标准形式：使用指针。优点：插入和删除较为方便。

顺序存储：当不需要经常进行插入和删除时，可以使用“适当的次序”依次存放。

- 按层次序的存储形式
- 按前序的存储形式

8 二叉树的顺序存储

按层次序的存储形式——适用于存储完全二叉树

按照完全二叉树的方法从上到下、从左到右放满。

n个结点，（具有n（n>0 ）个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor$），构造完成：

当0≤i ≤$\lfloor (n-2)/2 \rfloor$ 时，k_i有左子结点k_{2i+1}
当0≤i ≤$\lfloor (n-3)/2 \rfloor$ 时，k_i有左子结点k_{2i+2}
当1≤i ≤$n-1$时，k_i有父结点为$\lfloor (i-1)/2 \rfloor$
当0≤i ≤$\lfloor (n-3)/2 \rfloor$ 时，k_{2i+1}和k_{2i+2}有相同的父结点

8 二叉树的顺序存储

按层次序的存储形式——适用于存储完全二叉树

对于一般的二叉树，可以将其每个节点与完全二叉树上的结点相对照，存储在顺序存储结构中，以某种特殊的值表示不存在的结点。

按前序的存储形式

如果仅把前序中的结点依次存放在一个一维数组中，无法完全反映树中结点之间的关系。需要设置附加信息：

两种方法：

（一）附加左标志位和右指针

（二）附加左标志位和右标志位

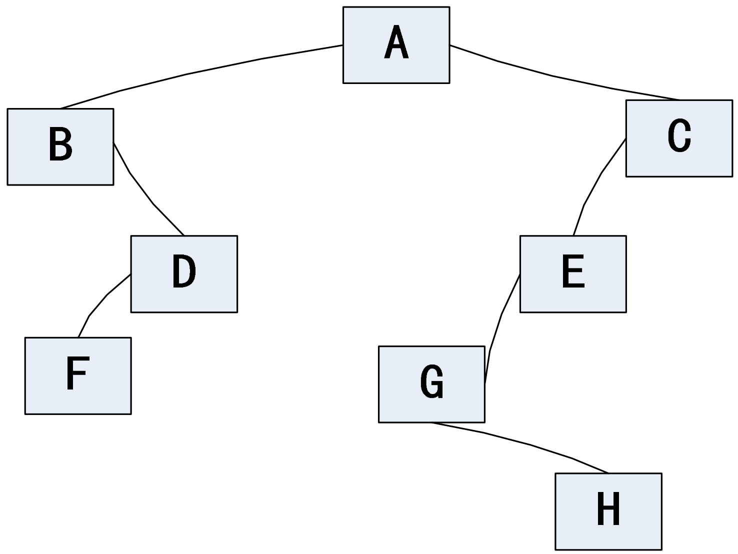
按前序的存储形式——（一）附加左标志位和右指针

（ltag，data,rchild）

ltag=0,结点k后面的结点是k的左子结点；
ltag=1,结点k无左子结点；
rchild：数组下标，记录其右子结点的信息。（用-1表示空的指针）

按前序的存储形式——（一）附加左标志位和右指针

ltag：ltag=0,结点k后面的结点是k的左子结点； ltag=1,结点k无左子结点；
rchild：数组下标，记录其右子结点的信息（数组下标）。
（用-1表示空的指针）



下标	ltag	data	rchild
0	0	A	4
1	1	B	2
2	0	D	-1
3	1	F	-1
4	0	C	-1
5	0	E	-1
6	1	G	7
7	1	H	-1

按前序的存储形式——（一）附加左标志位和右指针

- 查找结点k的左子结点
- 查找结点k的右子节点
- 查找结点k的按前序的前面结点
- 查找结点k的按前序的后面结点
- 查找结点k的父结点

按前序的存储形式——（一）附加左标志位和右指针

查找结点k的父结点可使用下面的语句：

```
if (p-1<0)
      结点k没有父结点
else  if   (a[p-1].ltag==0)
            printf( “%c” ,a[p-1].data);
      else {
                  for (q=p-1;a[q].rchild!=p;q--) ;
                        printf( “%c” ,a[p-1].data);
            }
```

按前序的存储形式——（二）附加左标志位和右标志位

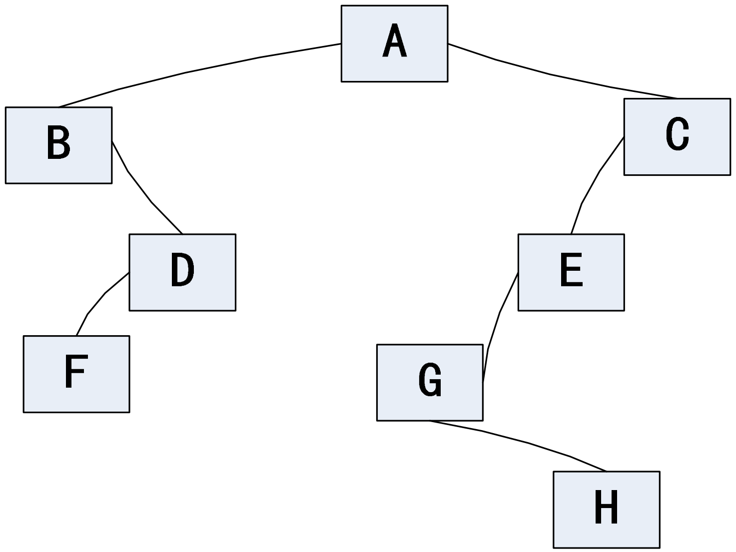
（ltag， data, rtag）

ltag=0, 结点k后面的结点是k的左子结点；
ltag=1, 结点k无左子结点；
rtag=0： 有右子结点；
Rtag=1 :无右子结点；

8 二叉树的顺序存储

按前序的存储形式——（一）附加左标志位和右标志位

ltag=0, 结点k后面的结点是k的左子结点；
ltag=1, 结点k无左子结点；
rtag=0： 有右子结点；
rtag=1:无右子结点；



下标	ltag	data	rtag
0	0	A	0
1	1	B	0
2	0	D	1
3	1	F	1
4	0	C	1
5	0	E	1
6	1	G	0
7	1	H	1

按前序的存储形式——（二）附加左标志位和右标志位

- 查找结点k的左子结点
- 查找结点k的按前序的前面结点和后面结点
- 查找结点k的右子结点

如何查找结点k的右子结点?
K没有左子结点：后续就是k的右子结点；
K有左子结点：？？

按前序的存储形式——（二）附加左标志位和右标志位

如何查找结点k的右子结点？
K有左子结点：找到k的左子树的最后一个叶子节点。

方法（使用一个栈，存放rtag=0（有右子结点）、但尚未找到右子结点的那些结点的地址）：
在查找的过程中，遇到rtag=0的结点，进栈；
如果当前结点ltag=1（无左子结点），那么此结点的后一个结点一定是栈顶结点的右子结点，则栈顶结点出栈；

按前序的存储形式——（二）附加左标志位和右标志位

```
// 定义struct
//………………………………NODE;
struct lrnode
{       char char;    //结点的值
        char ltag,rtag;   //结点的标志位
}
typedef struct lrnode LRNODE;
//ltag=0,结点k后面的结点是k的左子结点;
//ltag=1,结点k无左子结点;
//rtag=0: 有右子结点;
//rtag=1:无右子结点;

//LRNODE tree[MAXN] 中顺序存储二叉树，转换成按标准形式
```

按前序的存储形式——（二）附加左标志位和右标志位

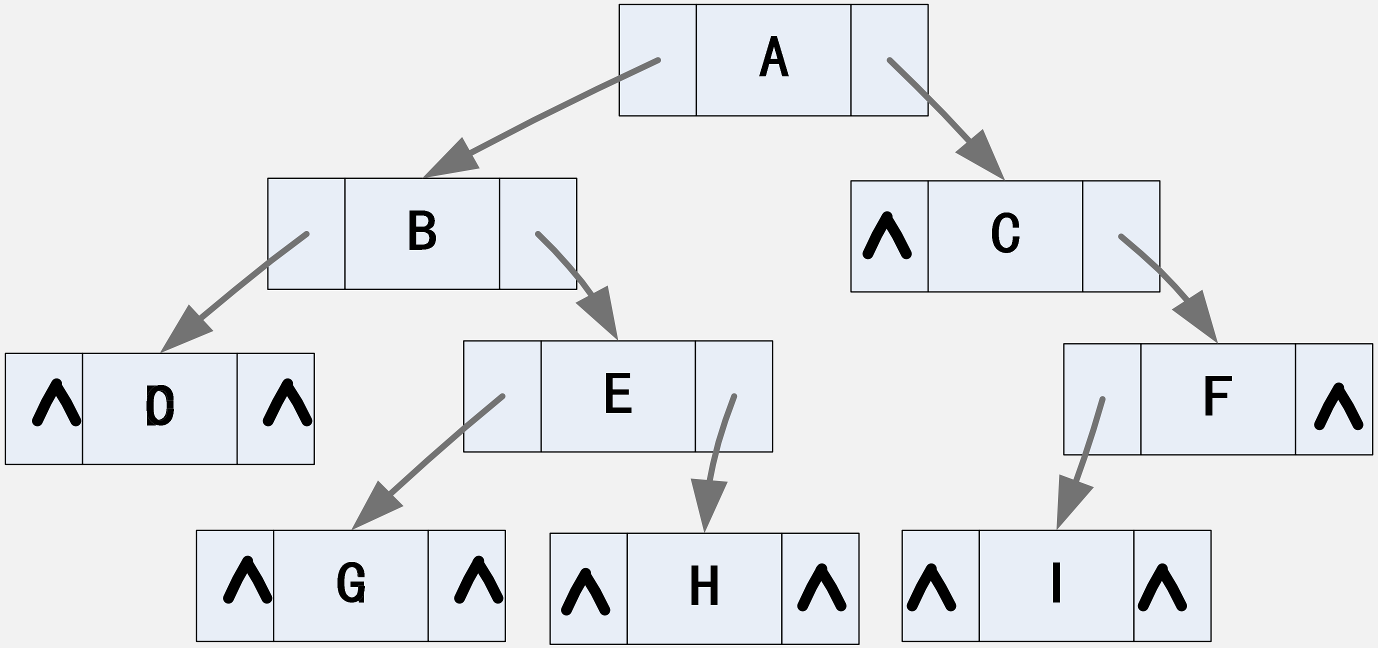
```
NODE *root,*p,*q,*stack[MAXN];
//malloc……构造根结点，等待填入data, lchild, rchild
p=root;      //p 指向当前正在构造的结点
top=0;
for ( i=0;i<n-1;i++) // 顺序存储的结点依次读入
{       p->data=tree[i].data;
        if (tree[i].rtag== '0' ) stack[top++]=p;//有右孩子，先入栈
            else p->rchild =NULL;  //没有右孩子
        q=(NODE*)malloc(sizeof(NODE));
        if (tree[i].ltag== '0' )  p->lchild =q;  / /p有左子结点
        else
        {       p->lchild=NULL;
                p=stack[--top];
                p->rchild=q;
        }
        p=q;
}//......
```

```
……//构造根结点，等待填入data，lchild，rchild
p=root;//p 指向当前正在构造的结点
top=0;
for ( i=0;i<n-1;i++) // 顺序存储的结点依次读入
{       p->data=tree[i].data;
        if (tree[i].rtag== '0' ) stack[top++]=p;//有右孩子，先入栈
        else p->rchild =NULL;  //没有右孩子
        q=(*)malloc(sizeof(NODE));
        if (tree[i].ltag== '0' )   p->lchild =q;   / /p有左子结点，在tree[i+1]
        else
        {     p->lchild=NULL;  //p没有左子节点
              p=stack[--top];  //后续结点tree[i+1]是栈顶结点的右子结点
              p->rchild=q;
        }
        p=q;
}
P->data =tree[n-1].data;
P->lchild=NULL;
P->rchild=NULL;
Return (root);
```

如果二叉树的结点的值唯一，且是可比较大小的。
如何判断一棵二叉树是否左子树上的结点的值都小于根，右子树上的结点的值都大于根？

n个结点的二叉树，如果按标准形式来存储，会有多少个指针字段？
其中多少个是空的？

当我们用标准形式存储一棵二叉树时，树中有一半以上的指针是空的。如何利用这些指针？



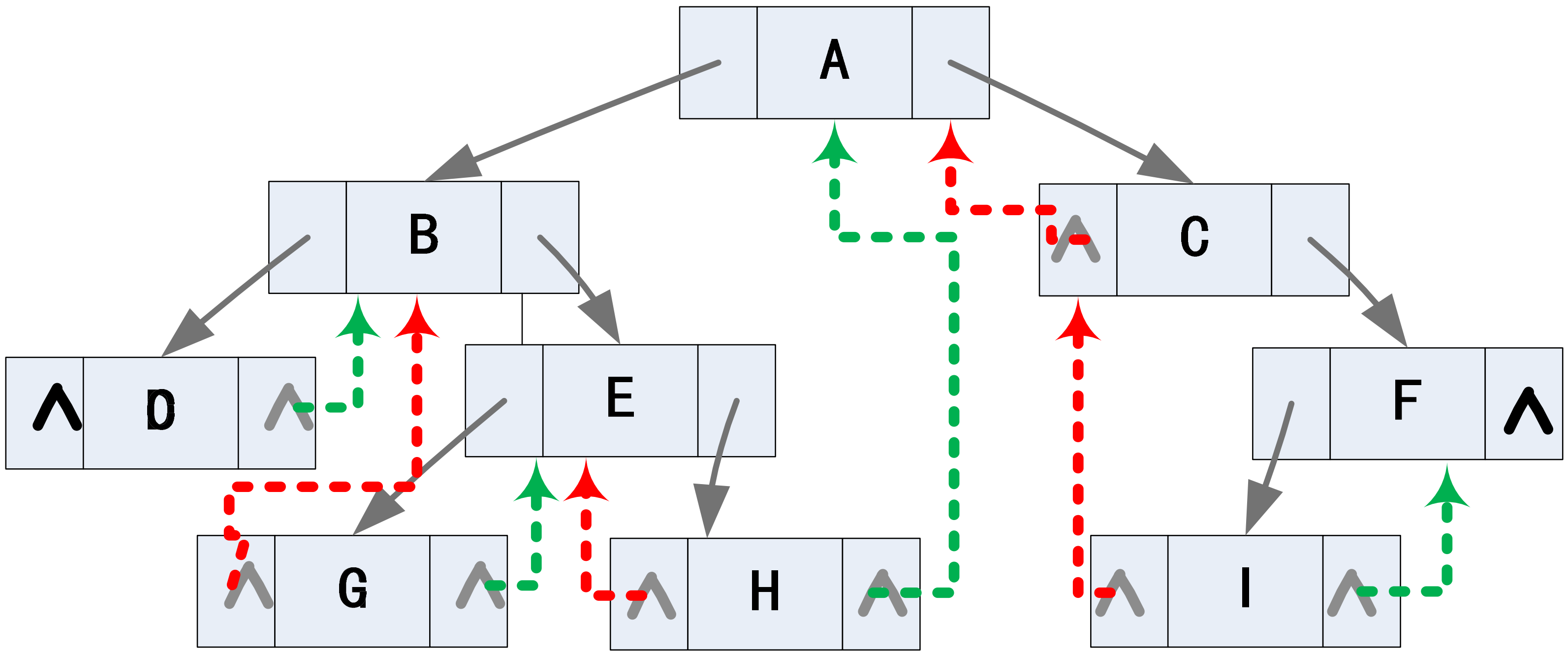
做法：用来穿线（也叫做线索化二叉树）

9 穿线树和穿线排序

穿线树（也叫做线索化二叉树，Threaded Binary Tree）

中序穿线树：设T是一棵二叉树，我们采用标准形式存储这棵二叉树。对于T中的每个结点k，如果它没有左（或右）子结点，而k’是k的按中序的前面（或后面）结点，那么置结点k的左（或右）指针为k’的地址。

9 穿线树和穿线排序



9 穿线树和穿线排序

为了区分k的lchild和rchild字段表达的是“序”还是k的真正子结点，我们在结点上增加两个字段，ltag和rtag。

ltag=1，lchild用来存放“线”，指向的是该结点的按中序的前面结点。
ltag=0，lchild指向真正的左子结点。
rtag=1，rchild用来存放“线”，指向中序的后继
rtag=0，rchild指向真正的右子结点

只有当k是二叉树T按中序的最前面一个结点时，k的左指针才为空，同时k的ltag取值为0；
通过这样处理的二叉树T为中序穿线树。
用一个指针root指向根结点，还用一个指针head指向按中序的最前面一个结点。

结点结构：

lchild	ltag	data	rtag	rchild

```
struct node
{
        char data;
        struct node *lchild,*rchild;
        int ltag,rtag;
};
```

中序穿线树中的结点t:
无左子结点：t->ltag=1 或 t->lchild =NULL （中序的第一个）

无右子结点：t->rtag=1 或 t-rchild =NULL （中序的最后一个）

9 穿线树和穿线排序

在给定的中序穿线树中进行的操作：

- 找出指针t所指结点的按中序的前面结点和后面结点
- 按中序输出树中的全部结点
- 向树中插入结点

在给定的穿线树中，找出指针t所指结点的按中序的前面结点

```
NODE * pred(NODE *t)
{
      if ( t->ltag==1  || t->lchild ==NULL )  //无左子树
            return (t->lchild);
      //有左子树，则寻找左子树中“最右”的结点
      t=t->lchild;
      while (t->rtag==0)   t=t->rchild;
      return (t);
}
```

在给定的穿线树中，找出指针t所指结点的按中序的后面结点

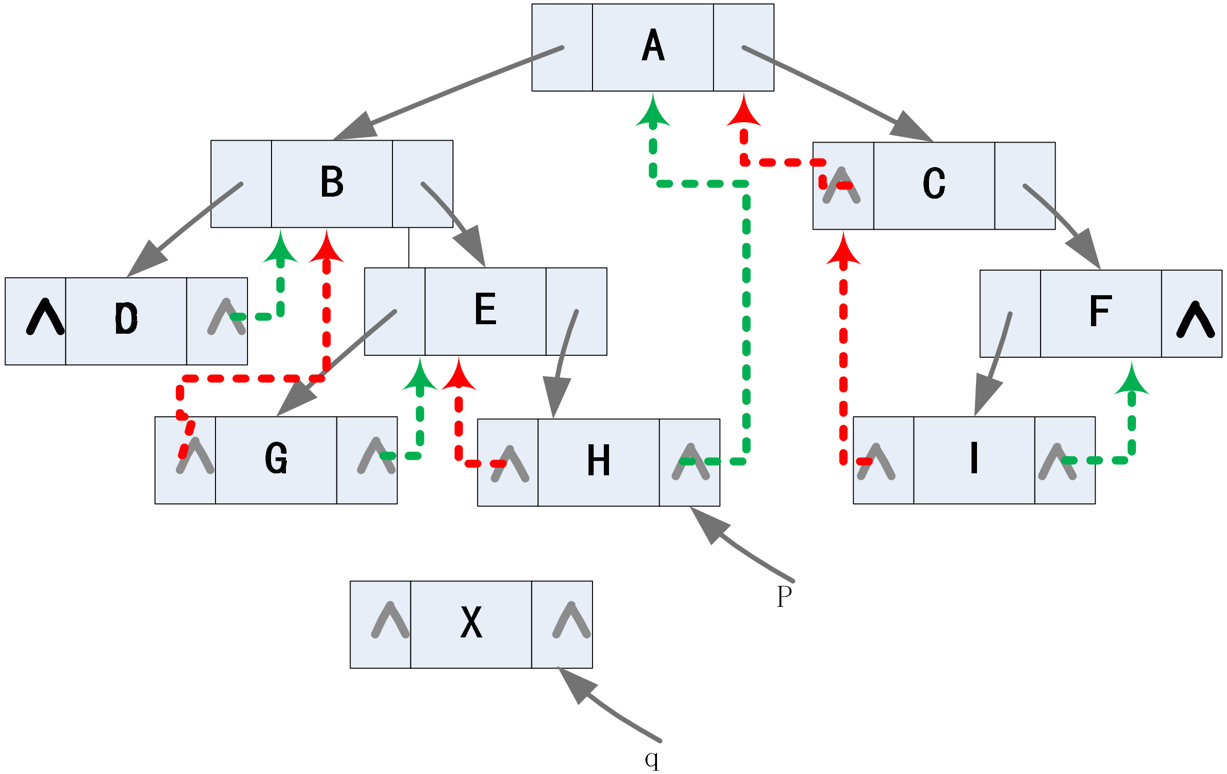
```
NODE * succ(NODE  *t)
{
if  ( t->rtag==1 | |t->rchild ==NULL )  //无右子树
            return (t->rchild);
      //有右子树，则中序的后面的结点是右子树中“最左”的结点
      t=t->rchild;
      while (t->ltag==0)  t=t->lchild;
      return (t);
}
```

head指向穿线树按中序的最前面的结点，输出中序遍历的序列

```
void midorder(NODE  *head)
    {  while (head!=NULL)
        {
                printf( “%c” , head->data);
                head=succ(head);
        }
}
```

在穿线树中插入一个结点：

void left_insert (p, q, p_head) //q插在p所指结点按中序的前面



在穿线树中插入一个结点：//q插在p所指结点按中序的前面

```
void left_insert (NODE *p, NODE *q,NODE **p_head)
{     NODE *r;
      if (p->ltag==1  || p->lchild ==NULL )  //无左
      {     q->lchild=p->lchild;
            q->ltag=p->ltag;
            q->rchild=p;
            q->rtag =1;
            p->lchild = q;
            p->ltag=0;
            if (q->lchild==NULL ) *p_head =q; //特殊情况：新插入的结点成为中序的第一个结点，所以修改head
      }     else …… //
```

在穿线树中插入一个结点：

```
void left_insert (NODE *p, NODE *q,NODE **p_head)
{     NODE *r;
      if (p->ltag==1  || p->lchild ==NULL )  //无左
      {………… }
      else  //p有左子结点，则q作为p的左子树中序遍的最后一个结点r（也是P的中序的前驱）的右孩子，r：p的左子树的“最右”的结点
      {}
```

在穿线树中插入一个结点：

```
{     if (p->ltag==1  || p->lchild ==NULL )  //无左
      {………… }
      else
      {     r=pred(p);
            q->rchild = r->rchild;
            q->rtag=r->rtag;
            q->lchild = r;
            q->ltag=1;
            r->rchild = q;
            r->rtag=0;
      }
}
```

jyang
@cs.ecnu.edu.cn

在穿线树中插入一个结点：

```
//q插在p所指结点按中序的后面
void right_insert (NODE *p, NODE* q)
{     NODE *r;
      if (p->rtag==1  || p->rchild ==NULL ) //p无右子结点
      {
            q->rchild = p->rchild;
            q->rtag=p->rtag;
            q->lchild = p;
            q->ltag=1;
            r->rchild = q;
            r->rtag=0;
      }
      else
```

在穿线树中插入一个结点:

```
//q插在p所指结点按中序的后面
void right_insert (NODE *p,NODE* q)
{     NODE *r;
      if (p->rtag==1  || p->rchild ==NULL ) //p无右子结点
      {…… }
      else //p有右子结点
      {
      r= succ (p) ;
      ……
      }
}
```

用穿线树进行排序

首先，我们用给定的n个结点的序列建造一棵穿线树，使得树中每个结点的都值大于该结点的非空左子树中所有结点的值，且都小于该结点的非空右子树中所有结点的值，我们称这样的树为穿线排序树。

按中序遍历穿线排序树，这时得到的n个结点的新序列，是由小到大排好序的。这种排序方法就称为穿线排序。

如何建立穿线排序树?

用穿线树进行排序

```
NODE * thread_sort_tree(char a[], int n)
{
……//构造第一个结点，作为初始的root
head=root;
for (i=1;i<n;i++)
{
 //找到合适的位置插入结点，保证插入后仍然是中序穿线排序

}
return (head);
}
```

用穿线树进行排序

```
for (i=1;i<n;i++)
{
        r=(NODE *)malloc(sizeof(NODE));
        r->data =a[i];
        p=root; //从根开始进行比较，寻找r插入的位置
        while (1)
        { if (r->data <= p->data)
                //r应该在p的左子树中
          else
                //r应该在p的右子树中
        }
//插入
}
return (head);
}
```

用穿线树进行排序

```
for (i=1;i<n;i++)
{
        r=(NODE *)malloc(sizeof(NODE));
        r->data =a[i];
        p=root;
        while (1)  //寻找r插入的位置
        { if (r->data <= p->data) //r应该插入到p的左子树中
                if (p->ltag==0 && p->lchild != NULL)
                        p=p->lchild;
                else break;
          else if (p->ltag==0 && p->lchild != NULL)
                        p=p->lchild; //r应该插入在p的右子树中
                else break;
        }
//找到p后，插入，有两种情况，r作为p的左子结点或作为右子结点
}
return (head;)
}
```

用穿线树进行排序

```
for (i=1;i<n;i++)
{
        ……
        while (1)  //寻找r插入的位置p
        {……   }
//找到p后，插入，插入，有两种情况，r作为p的左子结点或作为右子结点
        if (r->data < p->data) //r作为p的左子结点
        {
                r->lchild = p->lchild;
                r->ltag=p->ltag;
                r->rchild=p;
                r->rtag=1;
                p->lchild = r;
                p->ltag=0;
                if (r->lchild==NULL) head=r;
        }
        else ……//r作为p的右子结点
}
return (head);
}
```

用穿线树进行排序

```
void thread_sort(char a[], int n)
{
        NODE  *head;
        head = thread_sort_tree(a,n);
        printf( “Output mid_order:  ” );
        midorder(head);

}
```

10 计算二叉树的数目

如果有n个结点，共有多少棵不同的二叉树？

$$b_n = \frac{1}{n+1} C_{2n}^{n}$$

10 计算二叉树的数目

n=0 或 n=1 ，只有一棵。

n=2 ？

n=3 ？

10 计算二叉树的数目

设有n个结点的不同二叉树数目为b_n，那么：

挑选n个结点中的一个作为根结点，

i （0<=i<=n-1） 是根结点的左子树中结点的个数，

剩下的(n-i-1)个结点在根的右子树中，

此时二叉树的数目 $b_i b_{n-i-1}$

$b_0=1$

$b_1=1$

$$b_n= b_0b_{n-1}+b_1b_{n-2}+ \quad +b_{n-1}b_0$$

$$b_n=\frac{1}{n+1}C_{2n}^{n}$$