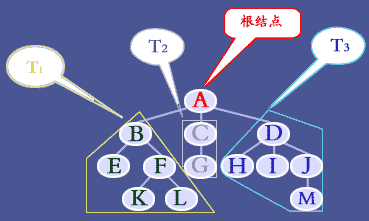
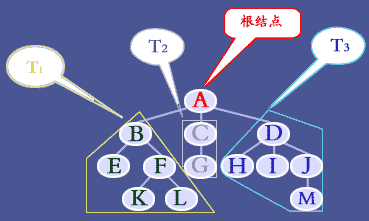
**5.1 树的类型定义**

* A为“根”
* T1、T2和T3都是一棵树，称为A的子树。
* 称根和子根之间的连线为“分支”
* 结点分支的个数定义为“结点的度”，如结点B的度为2，D的度为3。
* 树中所有结点度的最大值定义为“树的度”。
* 称度为零的结点为“叶子”或“终端结点”
* 所有度不为零的结点被称作“分支结点”



**5.1.1基本定义**

* 森林为m(m>=0)棵互不相交的树的集合。
* 树的深度定义为树中叶子结点所在最大层次数。
* 称根结点为子树根的“双亲”。
* 称子树根为根结点的“孩子”。
* 根的所有子树根互为“兄弟”。
* 有序树、无序树 如果树中每棵子树从左向右的排列拥有一定的顺序，不得互换，则称为有序树，否则称为无序树。



**5.1.2树的抽象数据类型**

ADT Tree {

数据对象：D是具有相同特性的数据元素的集合。  
 数据关系：  
 若D为空集，则称为空树；  
 若D中仅含一个数据元素，则关系R为空集；  
 否则R={H}，  
 (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)当n>1时，其余数据元素可分为m(m>0)个互不相交的(非空)有限集T1,T2,…,Tm,其中每一个子集本身又是一棵符合本定义的树，称为根root的子树，每一棵子树的根xi都是根root的后继，即<root,xi>H,i=1,2,…,m。

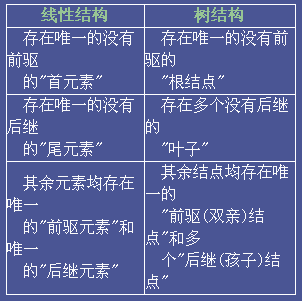
基本操作：  
  
 {结构初始化}  
 InitTree(&T);  
 操作结果：构造空树T。  
  
 CreateTree(&T,definition);  
 初始条件：definition给出树T的定义。  
 操作结果：按definition构造树T。  
  
 {销毁结构}  
 DestroyTree(&T);  
 初始条件：树T存在。  
 操作结果：销毁树T。

{引用型操作}  
 TreeEmpty(T);  
 初始条件：树T存在。  
 操作结果：若T为空树，则返回TRUE，否则返回FALSE。  
  
 TreeDepth(T);  
 初始条件：树T存在。  
 操作结果：返回T的深度。  
  
 Root(T);  
 初始条件：树T存在。  
 操作结果：返回T的根。  
  
 Value(T,cur\_e);  
 初始条件：树T存在，cur\_e是T中某个结点。  
 操作结果：返回cur\_e的值。

Parent(T, cur\_e);  
　　初始条件：树 T 存在，cur\_e 是 T 中某个结点。  
　　操作结果：若 cur\_e 是T的非根结点，则返回它的双亲，否则返回"空"。  
  
 LeftChild(T,cur\_e);  
 初始条件：树T存在，cur\_e是T中某个结点。  
 操作结果：若cur\_e是T的非叶子结点，则返回它的最左孩子，否则返回"空"。  
  
 RightSibling(T,cur\_e);  
 初始条件：树T存在，cur\_e是T中某个结点。  
 操作结果：若cur\_e有右兄弟，则返回它的右兄弟，否则返回"空"。  
  
 TraverseTree(T,visit());  
 初始条件：树T存在，visit是对结点操作的应用函数。  
 操作结果：按某种次序对T的每个结点调用函数visit()一次且至多一次。一旦visit()失败，则操作失败。

{加工型操作}  
 Assign(T,cur\_e,value);  
 初始条件：树T存在，cur\_e是T中某个结点。  
 操作结果：结点cur\_e赋值为value。  
  
 ClearTree(&T);  
 初始条件：树T存在。  
 操作结果：将树T清为空树。  
  
 InsertChild(&T,&p,i,c);  
 初始条件：树T存在，p指向T中某个结点，1≤i≤p所指结点的度＋1，非空树c与T不相交。  
 操作结果：插入c为T中p所指结点的第i棵子树。  
  
 DeleteChild(&T,&p,i);  
 初始条件：树T存在，p指向T中某个结点，1≤i≤p指结点的度。  
 操作结果：删除T中p所指结点的第i棵子树。  
} ADT Tree

**5.1.3树和线性结构对照**



**5.2 二叉树类型**

定义：二叉树是另一种树形结构。它与树形结构的区别是：

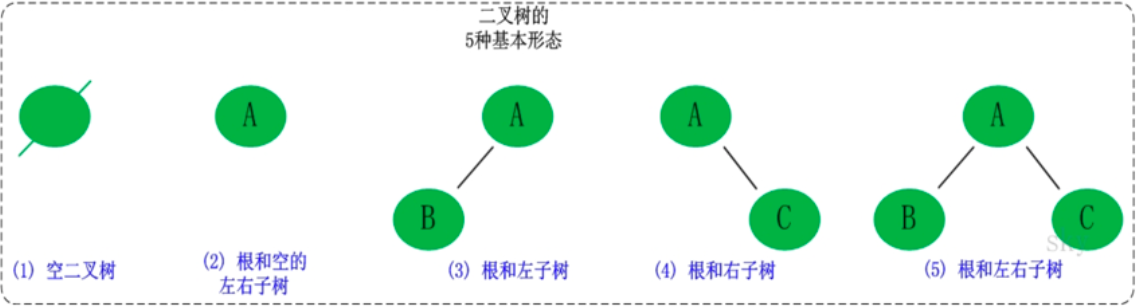
1. **每个结点最多有两颗子树**；
2. **子树有左右之分**。

二叉树也可以用递归的形式定义。

即：二叉树是n（n>=0）个结点的有限集合。

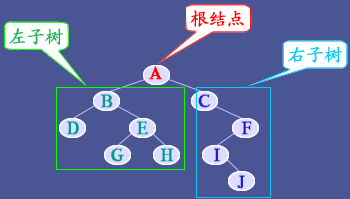
当n=0时，称为空二叉树；当n>0时，有且仅有一个结点为二叉树的根，其余结点被分成两个互不相交的子集，一个作为左子集，另一个作为右子集，每个子集又是一个二叉树。

**二叉树的5种形态：**



**5.2.1二叉树的类型定义**

ADT BinaryTree{  
 数据对象：D是具有相同特性的数据元素的集合。  
 数据关系：  
 若D为空集，称BinaryTree为空二叉树；  
 否则关系R={H}：  
 (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)D中其余元素必可分为两个互不相交的子集L和R，每一个子集都是一棵符合本定义的二叉树，并分别为root的左子树和右子树。如果左子树L不空，则必存在一个根结点，它是root的"左后继"(<root,>∈H)，如果右子树R不空，则必存在一个根结点为root的"右后继"(<root,>∈H)。



基本操作P：  
{结构初始化}

InitBiTree(&T);  
 操作结果：构造空二叉树T。  
 CreateBiTree(&T,definition);  
 初始条件：definition给出二叉树T的定义。  
 操作结果：按definition构造二叉树T。  
  
{销毁结构}  
 DestroyBiTree(&T);  
 初始条件：二叉树T存在。  
 操作结果：销毁二叉树T。

{引用型操作}  
 BiTreeEmpty(T);  
 初始条件：二叉树T存在。  
 操作结果：若T为空二叉树，则返回TRUE，否则返回FALSE。  
  
 BiTreeDepth(T);  
 初始条件：二叉树T存在。  
 操作结果：返回T的深度。  
  
 Root(T);  
 初始条件：二叉树T存在。  
 操作结果：返回T的根。  
  
 Value(T,e);  
 初始条件：二叉树T存在，e是T中某个结点。  
 操作结果：返回e的值。

Parent(T,e);  
 初始条件：二叉树T存在，e是T中某个结点。  
 操作结果：若e是T的非根结点，则返回它的双亲，否则返回"空"。  
  
 LeftChild(T,e);  
 初始条件：二叉树T存在，e是T中某个结点。  
 操作结果：返回e的左孩子。若e无左孩子，则返回"空"。  
  
 RightChild(T,e);  
 初始条件：二叉树T存在，e是T中某个结点。  
 操作结果：返回e的右孩子。若e无右孩子，则返回"空"。  
  
 LeftSibling(T,e);  
 初始条件：二叉树T存在，e是T中某个结点。  
 操作结果：返回e的左兄弟。若e是其双亲的左孩子或无左兄弟，则返回"空"。

RightSibling(T,e);  
 初始条件：二叉树T存在，e是T的结点。  
 操作结果：返回e的右兄弟。若e是其双亲的右孩子或无右兄弟，则返回"空"。  
  
 PreOrderTraverse(T,visit());  
 初始条件：二叉树T存在,visit是对结点操作的应用函数。  
 操作结果：先序遍历T，对每个结点调用函数visit一次且仅一次。一旦visit()失败，则操作失败。  
  
 InOrderTraverse(T,vsit());  
 初始条件：二叉树T存在，visit是对结点操作的应用函数。  
 操作结果：中序遍历T，对每个结点调用函数Visit一次且仅一次。一旦visit()失败，则操作失败。  
  
 PostOrderTraverse(T,visit());  
 初始条件：二叉树T存在，visit是对结点操作的应用函数。  
 操作结果：后序遍历T，对每个结点调用函数visit一次且仅一次。一旦visit()失败，则操作失败。

LevelOrderTraverse(T,visit());  
 初始条件：二叉树T存在，visit是对结点操作的应用函数。  
 操作结果：层序遍历T，对每个结点调用函数visit一次且仅一次。一旦visit()失败，则操作失败。  
  
{加工型操作}  
 ClearBiTree(&T);  
 初始条件：二叉树T存在。  
 操作结果：将二叉树T清为空树。  
  
 Assign(&T,&e,value);  
 初始条件：二叉树T存在，e是T中某个结点。  
 操作结果：结点e赋值为value。

InsertChild(&T,p,LR,c);  
 初始条件：二叉树T存在，p指向T中某个结点，LR为0或1，非空二叉树c与T不相交且右子树为空。  
 操作结果：根据LR为0或1，插入c为T中p所指结点的左或右子树。p所指结点原有左或右子树成为c的右子树。  
  
 DeleteChild(&T,p,LR);  
 初始条件：二叉树T存在，p指向T中某个结点，LR为0或1。  
 操作结果：根据LR为0或1，删除T中p所指结点的左或右子树。  
  
} ADT BinaryTree

**5.2.2二叉树的几个特性**

性质1：二叉树第i层上的结点数目最多为2i-1(i>=1)；

性质2：深度为k的二叉树至多有2k-1个结点（k>=1）；

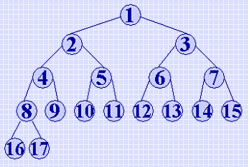
性质3：包含n个结点的完全二叉树的深度为(log2n)+1；

性质4：在任意一棵二叉树T中，若终端结点的个数为n0，度为2的结点数为n2，则n0=n2+1；

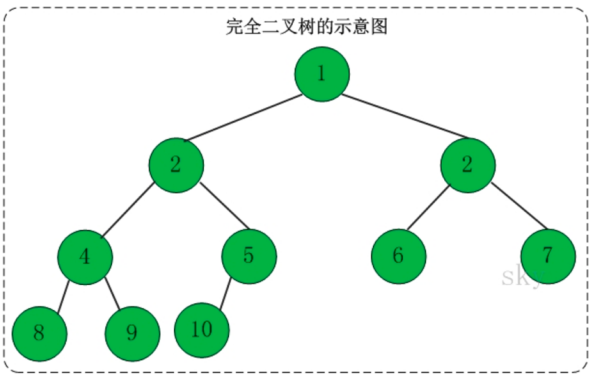
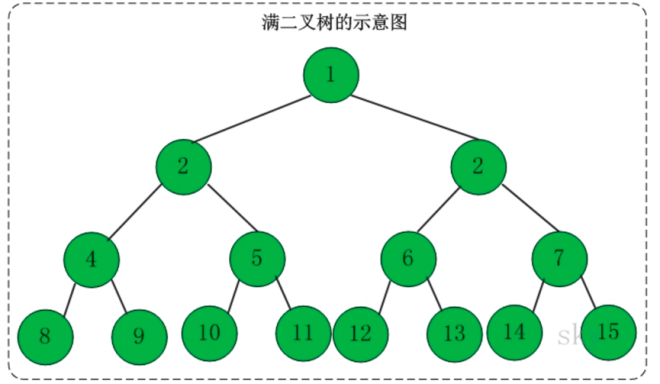
性质5：如果有一颗有n个节点的完全二叉树的节点按层次序编号，对任一层的节点i；（1<=i<=n）有：

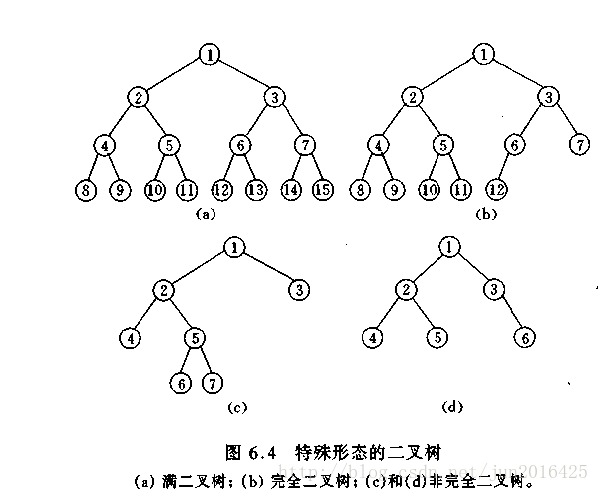
1. 如果i=1，则节点是二叉树的根，无双亲，如果i>1，则其双亲节点为[i/2]。
2. 如果2i>n那么节点i没有左孩子，否则其左孩子为2i。
3. 如果2i+1>n那么节点没有右孩子，否则右孩子为2i+1。

可以通过画图验证以上性质



* 若二叉树中所有的分支结点的度数都为2，且叶子结点都在同一层次上，则称这类二叉树为满二叉树。
* 一棵二叉树中，只有最下面两层结点的度可以小于2，并且最下层的叶结点集中在靠左的若干位置上，这样的二叉树称为完全二叉树。（树中所含的n的结点和满二叉树中编号为1至n的结点一一对应。）



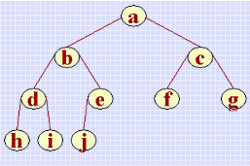


**5.3 二叉树的存储表示**

**5.3.1顺序存储结构**

二叉树的顺序存储结构的定义如下：

const MAXSIZE=100; //暂定二叉树中结点数的最大值为100  
typedef struct{  
 ElemType\*data; //存储空间基址(初始化时分配空间)  
 int nodeNum; //二叉树中结点数  
} SqBiTree; //二叉树的顺序存储结构

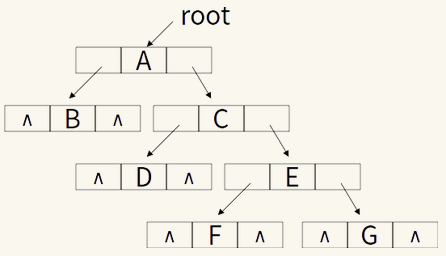
 

**5.3.2链式存储结构**

**（1）二叉树的二叉链表存储表示如下：**

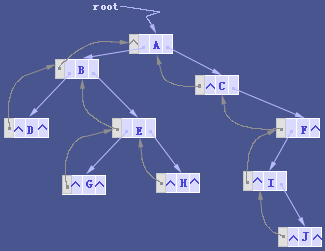
typedef struct BiTNode{  
 ElemType data;  
 struct BiTNode \*Lchild, \*Rchild; //左、右孩子指针  
} \*BiTree;





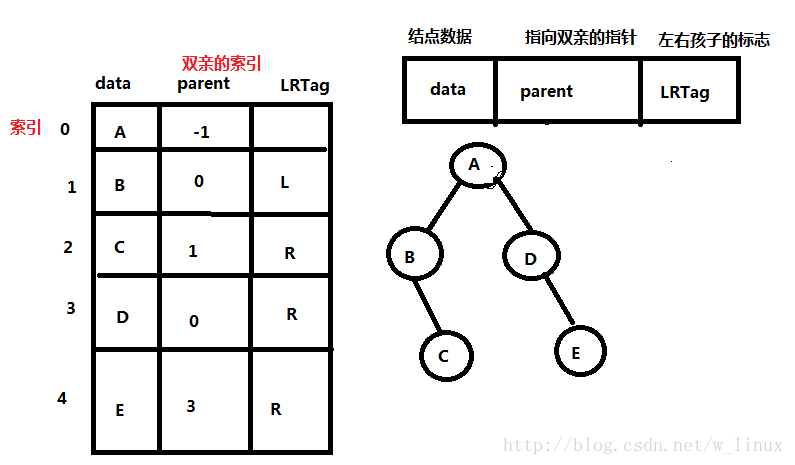
**（2）二叉树的三叉链表存储表示如下：**

typedef struct TriTNode{  
 ElemType data;  
 struct BiTNode \*Lchild, \*Rchild; //左、右孩子指针  
 struct BiTNode \*parent; //双亲指针  
} \*TriTree;



**（3）二叉树的双亲链表存储表示如下：**

typedef struct BPTNode{ //结点结构  
 ElemType data;  
 int \*parent; //指向双亲的指针  
 char LRTag; //左、右孩子标志域  
} BPTNode

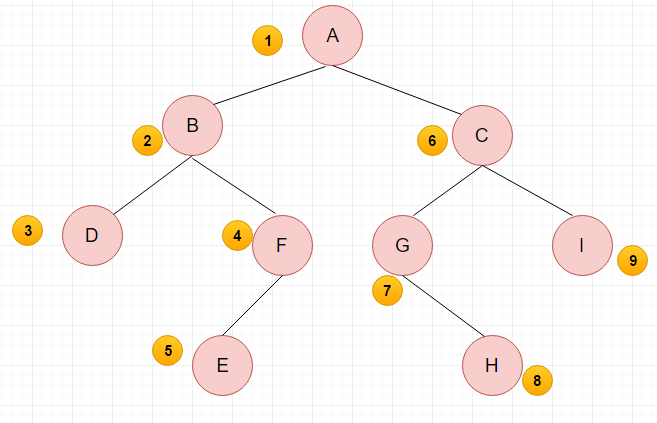


**5.4 二叉树的遍历**

* “遍历”的含义是对结构中的每个数据元素都访问一次且仅仅访问一次。
* 由于二叉树中每个结点都有两个后继，则可以有三条搜索路径：

**5.4.1先序遍历**

先序遍历：按照根结点->左子树->右子树的顺序访问二叉树



先序遍历：

（1）访问根结点；（2）采用先序递归遍历左子树；（3）采用先序递归遍历右子树；

（注：每个节点的分支都遵循上述的访问顺序，体现“递归调用”）

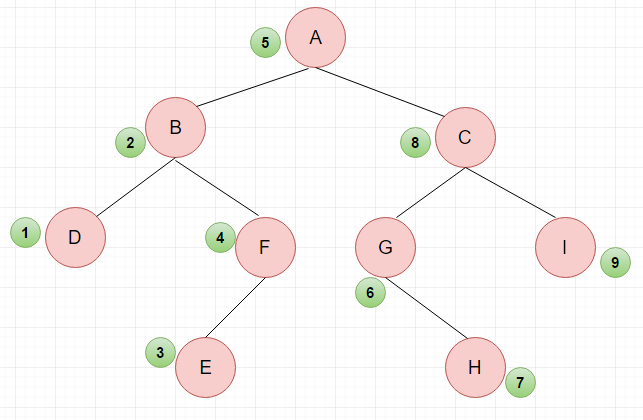
先序遍历结果：A BDFE CGHI

思维过程：

1. 先访问根结点A；
2. A分为左右两个子树，因为是递归调用，所以左子树也遵循“先根结点-再左-再右”的顺序，所以访问B节点；
3. 然后访问D节点；
4. 访问F节点的时候有分支，同样遵循“先根节点-再左-再右”的顺序；
5. 访问E节点，此时左边的大的子树已经访问完毕；
6. 然后遵循最后访问右子树的顺序，访问右边大的子树，右边大子树同样先访问根结点C；
7. 访问左子树G；
8. 因为G的左子树没有，所以接下来访问G的右子树H;
9. 最后访问C的右子树。

**5.4.2中序遍历**

中序遍历：按照左子树->根节点->右子树的顺序访问



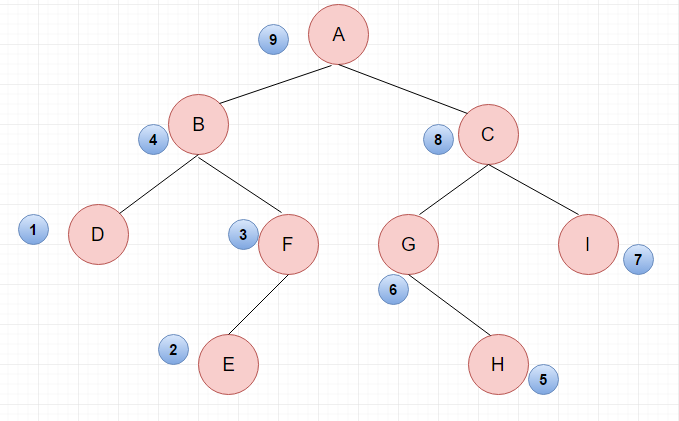
中序遍历：

（1）采用中序遍历左子树；（2）访问根节点；（3）采用中序遍历右子树

中序遍历结果：DBEF A GHCI

**5.4.2后序遍历**

先后遍历：按照左子树->右子树->根结点的顺序访问



后序遍历：

1. 采用后续递归遍历左子树；（2）采用后续递归遍历右子树；（3）访问根结点

后序遍历的结果：DEFB HGIC A

小节：三种方法遍历过程中经过节点的路线一样；只是访问各个节点的时机不同。

递归算法主要使用堆栈来实现。

**5.4.3先左后右的遍历**

**（1）先序遍历递归算法**

void PreOrder(BTree BT)

{

if(BT) {

Visit(BT);

PreOrder(BT->Lchild);

PreOrder(BT->Rchild);}

}

**（2）中序遍历递归算法**

void InOrder(BTree BT)

{

if(BT){

InOrder(BT->Lchild);

Visit(BT);

InOrder(BT->Rchild);

}

}

**（3）后序遍历递归算法**

void PostOrder(BTree BT)

{

if(BT){

PostOrder(BT->Lchild);

PostOrder(BT->Rchild);

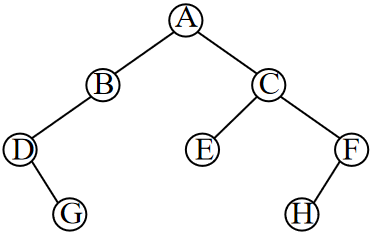
Visit(BT);

}

}

**按层次遍历二叉树**

按层次遍历该二叉树的序列为：ABCDEFGH



* 二叉树用链式存储结构表示时，按层遍历的算法实现

1. 访问根节点，并将根节点入队；
2. 当队列不空时，重复下列操作：

* 从队列退出一个结点；
* 若其有左孩子，则访问左孩子，并将其左孩子入队；
* 若其有右孩子，则访问右孩子，并将其右孩子入队。

void LevelOrder(BTree\*BT)

{

if(!BT) exit;

InitQueue(Q); p=BT; //初始化

Visite(p); EnQueue(&Q,p);

//访问根结点，并将根结点入队

while(!QueueEmpty(Q))

{ //当队非空时重复执行下列操作

DeQueue(&Q,&p); //出队

if(!p->Lchild)

{Visite(p->Lchild);EnQueue(&Q,p->Lchild);} //处理左孩子

if(!p->Rchild)

{Visite(p->Rchild);EnQueue(&Q,p->Rchild);} //处理右孩子

}

}

**建立二叉树**

void CreateBiTree(BiTree&T)  
{  
 //在先序遍历二叉树的过程中输入二叉树的"先序字符串"，  
 //建立根指针为T的二叉链表存储结构。在先序字符串中，  
 //字符'#'表示空树，其它字母字符为结点的数据元素  
 cin>>ch；  
 if(ch=='#')T=NULL; //建空树  
 else{  
 T=new BiTNode; //"访问"操作为生成根结点  
 T->data=ch;  
 CreateBiTree(T->Lchild); //递归建(遍历)左子树  
 CreateBiTree(T->Rchild); //递归建(遍历)右子树  
 }  
}

**统计二叉树中叶子结点的个数**

void CountLeaf(BiTree T,int&count)  
{  
 //先序遍历二叉树，以count返回二叉树中叶子结点的数目  
 if(T){  
 if((!T->Lchild)&&(!T->Rchild))  
 count++; //对叶子结点计数  
 CountLeaf(T->Lchild,count);  
 CountLeaf(T->Rchild,count);  
 } //if  
}

**求二叉树的深度**

void BiTreeDepth(BiTree T,int level,int&depth)  
{ //T指向二叉树的根，level为T所指结点所在层次，其初值为1，depth为当前求得的最大层次,其初值为0  
 if(T){  
 if(level>depth)depth=level;  
 BiTreeDepth(T->Lchild,level+1,depth);  
 BiTreeDepth(T->Rchild,level+1,depth);  
 }  
}

**复制二叉树**

生成一个二叉树的结点的算法：

BiTNode\*GetTreeNode(TElemType item,BiTNode\*lptr,BiTNode\*rptr)  
{ //生成一个其元素值为item，左指针为lptr，右指针为rptr的结点  
 T=new BiTNode; T->data=item;  
 T-> Lchild=lptr; T-> Rchild=rptr;  
 return T;  
}

后序遍历复制二叉树的操作为:

BiTNode\*CopyTree(BiTNode\*T)  
{  
 //已知二叉树的根指针为T，本算法返回它的复制品的根指针  
 if(!T)  
 return NULL; //复制一棵空树

if(T->Lchild)  
 newlptr=CopyTree(T->Lchild); //复制(遍历)左子树  
 else newlptr=NULL;

if(T->Rchild)  
 newrptr=CopyTree(T->Rchild); //复制(遍历)右子树  
 else newrptr=NULL;

newnode=GetTreeNode(T->data,newlptr,newrptr); //生成根结点  
 return newnode;  
}

**在二叉树上查询某个结点**

void locate(BiTree T,ElemType x,BiTree&p)  
{  
 //若二叉树T中存在和x相同的元素，则p指向该结点，否则p的值不变，p的初值为NULL  
 if(T)  
 {if(T->data==x)p=T;  
 locate(T->lchild,x,p);  
 locate(T->rchild,x,p);  
 }  
}

**5.4.4中序遍历（非递归）**

InitStack(S);

Push(S,T); //根入栈

while(!StackEmpty(S))

{

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

Push(S,p->lchild); //向左到尽头

Pop(S,p); //空指针出栈

if(!StackEmpty(S)) //访问，向右

{

Pop(S,p);

if(!Visit(p->data))return ERROR;

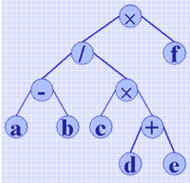
Push(S,p->rchild);

}

}

return OK;

**表达式的二叉树**

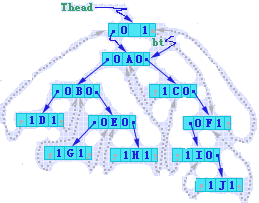
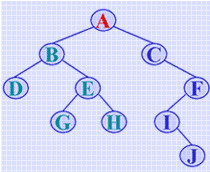


**二叉树的线索链表**

将在二叉树中每个结点（除第一个和最后一个外）的直接前驱和直接后继保存起来。这种信息是在遍历的动态过程中产生的，如果将这些信息在第一次遍历时就保存起来，则在以后再次需要对二叉树进行“遍历”时就可以将二叉树视作线性结构进行访问操作了。

typedef enum PointerType{Link=0,Thread=1};  
 //定义指针类型，以Link表示指针，Thread表示线索  
 typedef struct BiThrNode{  
 ElemType data;  
 struct BiThrNode \*Lchild, \*Rchild; //左右指针  
 PointerType LTag, RTag; //左右指针类型标志  
} \*BiThrTree;

在线索链表中添加了一个"头结点"，头结点的左指针指向二叉树的根结点，其右线索指向中序序列中的最后一个结点。



**以中序线索链表为存储结构的中序遍历**

void InOrderTraverse\_Thr(BiThrTree T,void(\*Visit)(ElemType e))  
{ //T指向中序线索链表中的头结点，头结点的左指针Lchild指向二叉树的根结点，头结点的右线索Rchild指向中序遍历访问的最后一个结点。本算法对此二叉树进行中序遍历，对树中每个元素调用函数Visit进行访问操作  
 p=T->Lchild; //p指向二叉树的根结点  
 while(p!=T)

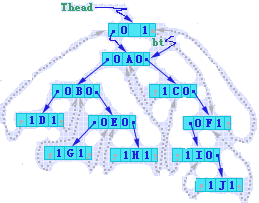
{ //空树或遍历结束时，p==Thead  
 while(p->LTag==Link)p=p->Lchild;  
 Visit(p->data); //访问其左子树为空的结点  
 while(p->RTag==Thread&&p->Rchild!=T)

{  
 p=p->rchild;Visit(p->data); //访问“右线索”所指后继结点  
 }  
 p=p->Rchild; //p进至其右子树根  
 }  
}

p=T->Lchild; //p指向二叉树的根结点  
while(p!=T)

{ //空树或遍历结束时，p==Thead  
 while(p->LTag==Link)p=p->Lchild;  
 Visit(p->data); //访问其左子树为空的结点  
 while(p->RTag==Thread&&p->Rchild!=T)

{  
 p=p->rchild;Visit(p->data); //访问“右线索”所指后继结点  
 }  
 p=p->Rchild; //p进至其右子树根  
}



**线索链表的生成**

void InThreading(BiThrTree p，BiThrTree&pre)  
{  
 //对p指向根结点的二叉树进行中序遍历，遍历过程中进行“中序线索化”。若p所指结点的左指针为空，则将它改为“左线索”，若pre所指结点的右指针为空，则将它改为“右线索”。指针pre在遍历过程中紧随其后，即始终指向p所指结点在中序序列中的前驱。  
 if(p) {  
 InThreading(p->Lchild,pre); //对左子树进行线索化  
 if(!p->Lchild) {

p->LTag=Thread;p->Lchild=pre;

} //建前驱线索  
 if(!pre->Rchild) {

pre->RTag=Thread;pre->Rchild=p;

} //建后继线索  
 pre=p; //保持pre指向p的前驱  
 InThreading(p->Rchild,pre); //对右子树进行线索化  
}

}

void InOrderThreading(BiThrTree&Th,BiThrTree BT)  
{ //BT为指向二叉树根结点的指针，由此二叉链表建立二叉树的中序线索链表，Thead指向线索链表中的头结点。  
 BiThrTree pre;  
 if(!(Th=new BiThrNode)) exit(1); //存储分配失败  
 Th->LTag=Link; Th->RTag=Thread; //建头结点  
 Th->Rchild=Th; //右指针回指  
 if(!BT)Th->Lchild=Th; //若二叉树空，则左指针回指  
 else{  
 Th->Lchild=BT;pre=Thead;  
 InThreading(BT,pre); //中序遍历进行中序线索化  
 pre->Rchild=Th;pre->RTag=Thead; //对中序序列中最后一个结点进行线索化  
 Th->Rchild=pre; //建非空树的头结点的"右线索"  
 }  
}

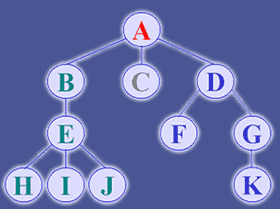
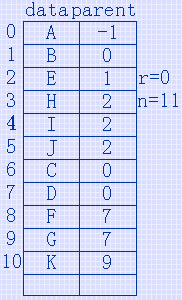
**5.4.5树和深林的存储表示**

**1、树的双亲表示法**

const MAX\_TREE\_SIZE=100;

typedef struct{//结点结构  
 ElemType data;  
 int parent;//双亲位置域  
} PTNode;

typedef struct{//树结构  
 PTNode nodes[MAX\_TREE\_SIZE];  
 int r,n;//根的位置和结点数  
} PTree;

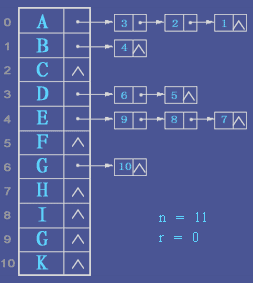
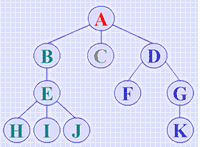
 

**2、树的孩子表示法**

树的孩子链表存储表示  
typedef struct CTNode{ //孩子结点  
 int child;  
 struct CTNode\*next;  
} \*ChildPtr;

typedef struct{  
 ElemType data; //结点的数据元素  
 ChildPtr firstchild; //孩子链表头指针  
} CTBox;

typedef struct{  
 CTBox nodes[MAX\_TREE\_SIZE];  
 int n,r; //结点数和根结点的位置  
} CTree;

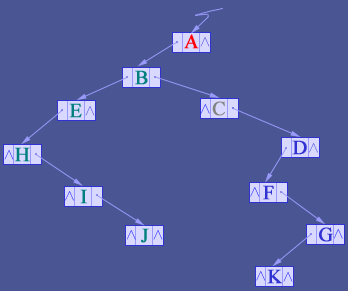
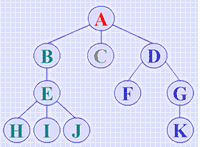


**3、树和森林的孩子兄弟表示法**

存储表示

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

firstchild指向该结点的"第一个"子树根结点，nextsibling指向它的"下一个"兄弟结点。

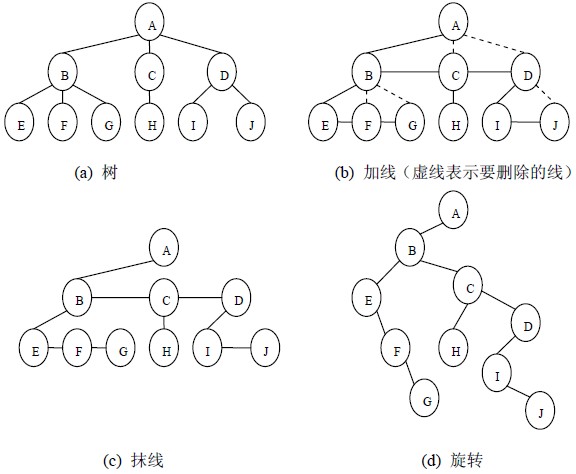


**4、树、森林和二叉树的相互转换**

**（1）树->二叉树**

#### 由于二叉树是有序的，为了避免混淆，对于无序树，我们约定树中的每个结点的孩子结点按从左到右的顺序进行编号。

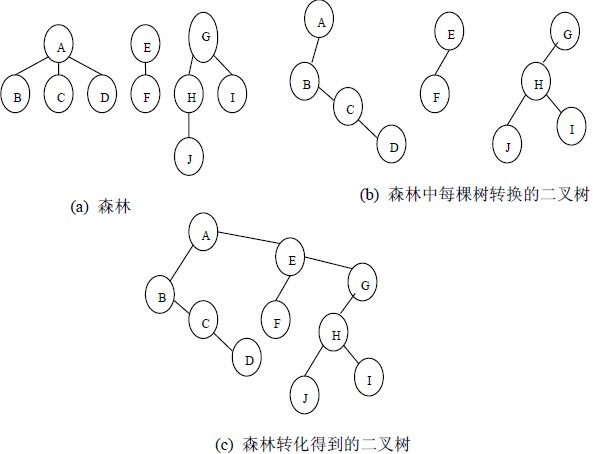
#### 将树转换成二叉树的步骤是： （1）**加线**。就是在所有兄弟结点之间加一条连线； （2）**抹线**。就是对树中的每个结点，只保留他与第一个孩子结点之间的连线，删除它与其它孩子结点之间的连线； （3）**旋转**。就是以树的根结点为轴心，将整棵树顺时针旋转一定角度，使之结构层次分明。



**（2）森林->二叉树**

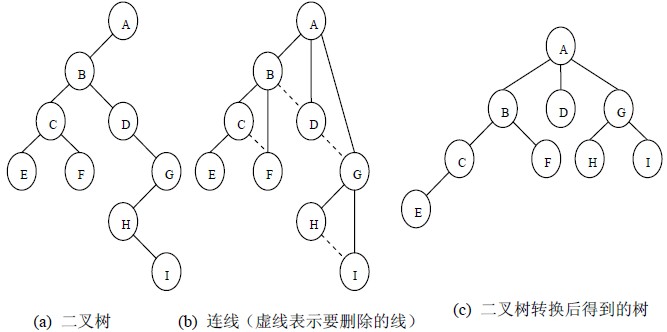
森林是由若干棵树组成，可以将森林中的每棵树的根结点看作是兄弟，由于每棵树都可以转换为二叉树，所以森林也可以转换为二叉树。

将森林转换为二叉树的步骤是：  
（1）先把每棵树转换为二叉树；  
（2）第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子结点，用线连接起来。当所有的二叉树连接起来后得到的二叉树就是由森林转换得到的二叉树。



**（3）二叉树转换为树**

二叉树转换为树是树转换为二叉树的逆过程，其步骤是：  
（1）若某结点的左孩子结点存在，将左孩子结点的右孩子结点、右孩子结点的右孩子结点……都作为该结点的孩子结点，将该结点与这些右孩子结点用线连接起来；  
（2）删除原二叉树中所有结点与其右孩子结点的连线；  
（3）整理（1）和（2）两步得到的树，使之结构层次分明。



（4）二叉树转换为森林

#### 二叉树转换为森林比较简单，其步骤如下： （1）先把每个结点与右孩子结点的连线删除，得到分离的二叉树； （2）把分离后的每棵二叉树转换为树； （3）整理第（2）步得到的树，使之规范，这样得到森林。

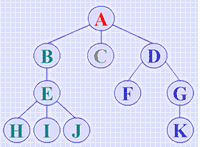
根据树与二叉树的转换关系以及二叉树的遍历定义可以推知，树的先序遍历与其转换的相应的二叉树的先序遍历的结果序列相同；树的后序遍历与其转换的二叉树的中序遍历的结果序列相同；树的层序遍历与其转换的二叉树的后序遍历的结果序列相同。由森林与二叉树的转换关系以及森林与二叉树的遍历定义可知，森林的先序遍历和中序遍历与所转换得到的二叉树的先序遍历和中序遍历的结果序列相同。

**5、树的遍历**

（1）先根(次序)遍历树

若树不空，则先访问根结点，然后依次从左到右先根遍历根的各科子树；

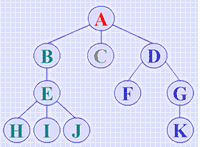
ABEHIJCDFGK



（2）后跟(次序)遍历树

若树不空，则先依次从左到右后根遍历根的各棵子树，然后访问根结点；

HIJEBCFKGDA



1. **森林的遍历**

（1）先序遍历森林

若森林不空，则可依下列次序进行遍历

1. 访问森林中第一棵树的根结点；
2. 先序遍历第一棵树中的子树森林；
3. 先序遍历除去第一棵树之后剩余的树构成的森林。

**7、求森林的深度**

森林的深度=Max{每一棵树的深度}  
树的深度=其子树森林的深度+1

int Depth\_Tree(CSTree T)  
{  
 //T是以孩子-兄弟链表存储的森林的头指针，返回该森林的深度  
 if(!T)return 0;  
 else{  
 dep=0; //初始化森林的深度为0  
 p=T; //指针p指向第一棵树的根  
 while(p){  
 d=Depth\_Tree(p->firstchild); //返回\*p的子树森林的深度  
 if(d+1>dep)dep=d+1; //求各棵树的深度的最大值  
 p=p->nextsibling; //指针p移向下一棵树的根  
 }  
 return dep;  
 }  
}