第五章　树与二叉树

基本内容：

- 树的基本概念
- 树和树林的存储表示
- 树与树林的周游（遍历）
- 二叉树
- 二叉树的存储表示
- 二叉树的周游和线索二叉树
- 树、树林与二叉树的转换
- 哈夫曼树和哈夫曼编码

线性结构：　唯一前驱、后继，一对一关系。
非线性结构：存在两个或两个以上前驱或后继。
　　一对多，或多对多的关系。

层次关系的问题可以用树、二叉树表示

重点：

- 树、二叉树的存储结构
- 二叉树的操作及其运算的实现
- 树、树林与二叉树的转换
- Haffman树与Haffman编码

5.1 树的基本概念

1. 树的定义
2. 基本术语（概念）
3. 树林
4. 树的基本运算

1. 树的定义

- 树的逻辑表示：

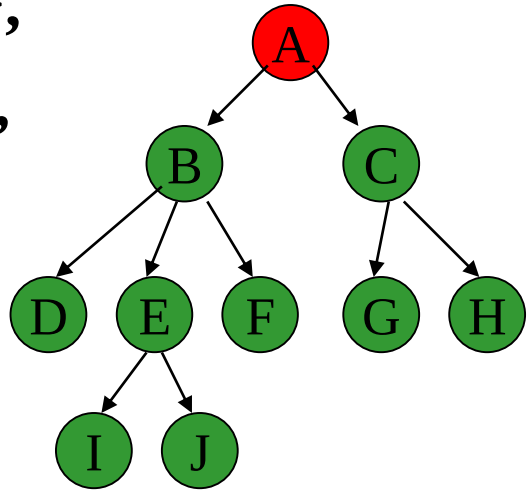
T = (N, R)

N = { A, B, C, D, E, F, G, H, I, J}

R = { <A, B>, <A, C>, <B, D>, <B, E>,
<B, F>, <C, G>, <C, H>, <E, I>,
<E, J> }

A, B, …, J → 结点

<A, B>…, → 树支（关系）

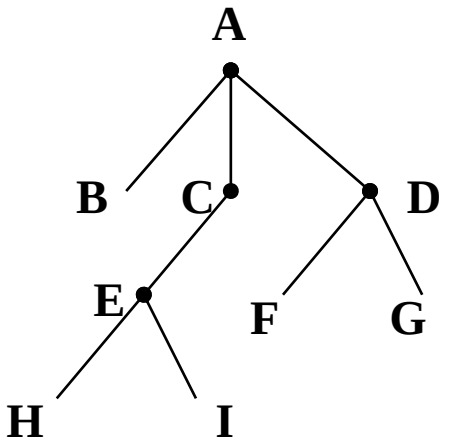


一对多的关系，允许多个后继。

空树

- 树的递归定义：n(n≥0)个结点的有穷集合T，当T非空时满足：
  - 有且仅有一个特别标志的称为根的结点
  - 除根外，其余结点分为m≥0个互不相交的非空集合T_1、T_2、…、T_m，而且每个非空集合T_i又是一颗树，称为T的子树。
- 树的特点：
  - 根无前驱，其它结点有唯一前驱
  - 除树叶（无子树的结点）结点外，其它结点有一个或多个后继

2. 基本术语（概念）



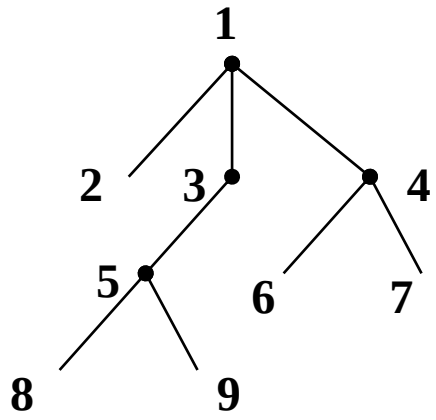
- 父结点、子结点、边

  若y为x的一颗子树的根，x为y的父结点
  y为x的子结点，
  有向对<x, y>为从x到y的边

- 兄弟：同一父母的结点互为兄弟

- 祖先、子孙

  若y在x的一颗子树中（y≠x），
  x为y的祖先，y为x的子孙



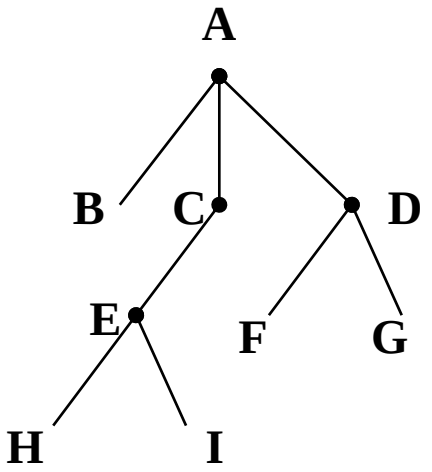
- 路径、路径长度

  从祖先x到子孙y的边的序列为x到y的路径，
  边的数目为路径长度

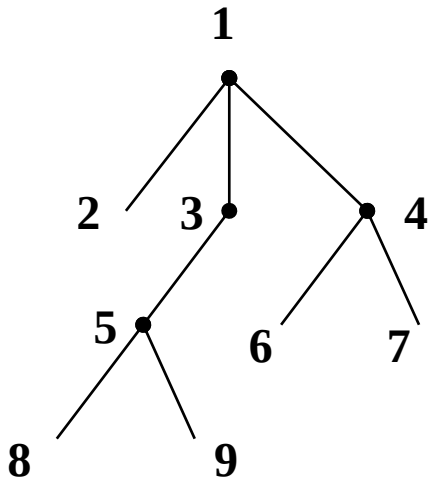
- 结点的层、树的层数

  根的层为0，其余结点的层为父结点层+1
  树的层数：结点的最大层数【空树: 0, 其它:≥1】

- **树的深度和高度**：结点的最大层数
  【空树: 0, 其它: ⩾1】
- **结点的度和树的度**
  结点的子树个数为结点的度，
  最大结点的度为树的度
- **树叶和分支结点**
  度数为0的结点为树叶（终端结点），
  其余结点为分支结点



- **无序树、有序树**：
  子树无次序的树为无序树，否则为有序树
- **结点的次序**
  在有序树中从左往右规定结点的次序，
  以右结点为根的子树中所有结点皆在左结点的右边。

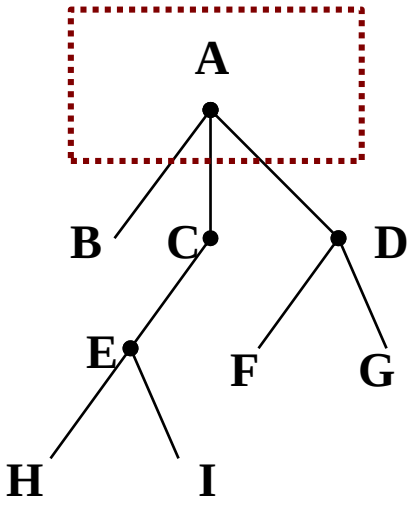


3. 树林

零颗或多颗互不相交的树构成树林（森林），树根互为兄弟。对树中每个结点而言，其子树的集合即为树林。

就逻辑结构而言，任何一棵树是一个二元组Tree=(root,F)，其中root称为树的根结点；F是m（m>=0）棵子树构成的树林，$F=(T_1, T_2,\ldots,T_m)$, 其中$T_i=(r_i,F_i)$称作根root的第i棵子树；当m≠0时，在树根和其子树林之间存在下列关系:

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



4. 树的基本运算

- 创建一颗空树
- 判断是否为空树
- 求根结点
- 求父结点
- 求第一个子结点
- 求右兄弟
- 遍历树（树的周游）

5.2 树和树林的存储表示

1. 树的存储表示
2. 树林的存储表示

选择存储表示方法原则：结点本身+结点之间的关系

1. 树的存储表示（三种）

- 父结点表示法　　[双亲表示法，上层（前驱）关系]
- 子表表示法　　[孩子表示法，下层（后继）关系]
- 长子-兄弟表示法 [孩子兄弟表示法，下层（长子）和同层（兄弟）关系]

（1）父结点表示法

用一组连续空间存储树的结点，并附设一个指示器指示其双亲结点的位置。结构类型如下：

```
typedef struct  ParTreeNode  /* 树中结点结构 */
{
    DataType      info;          /* 结点信息 */
    int           parent;        /* 父结点位置 */
} ParTreeNode
```

```
typedef  struct ParTree
{   ParTreeNode nodelist[MAXNUM];
    int         n;
}ParTree, *PParTree;
```

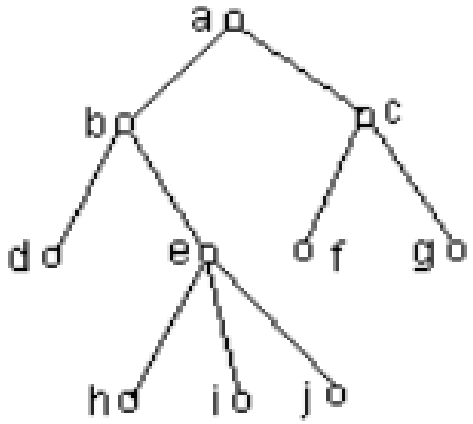
结点顺序存放，结点的位置由结点在数组中的下标给出。根结点无父结点，因此其parent为-1。

优点：a) 容易求根、找父结点及其所有的祖先；

b) 能找到结点的子女和兄弟；

缺点：a) 没有表示出结点之间的左右次序（无法求最左右兄弟）；

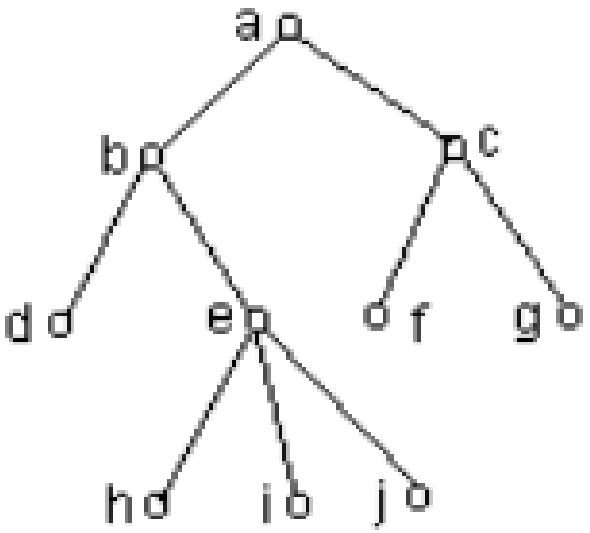
b) 找结点的子女和兄弟比较费事；



Info	parent
a	-1
b	0
c	0
d	1
e	1
f	2
g	2
h	4
i	4
j	4

改进方法：按一种周游次序在数组中存放结点。

常见的一种方法是依次存放树的**先根序列**，如下图：



Info	parent
a	-1
b	0
c	0
d	1
e	1
f	2
g	2
h	4
i	4
j	4

普通

	info	parent
0	a	-1
1	b	0
2	d	1
3	e	1
4	h	3
5	i	3
6	j	3
7	c	0
8	f	7
9	g	7

改进

改进后，任何结点的所有子孙都在该结点后连续存放。

找长子运算：如果存在长子，必定在结点的下一个位置

```
if (t->nodelist[p+1].parent == p)
        return p+1;
else    return -1;
```

找右兄弟运算：与结点具有相同父结点的后面结点

```
parent = t->nodelist[p].parent;
for (i = p+1; i < t->n; i++)
{       if (t->nodelist[i].parent == parent)
                return i;
}
return –1;
```

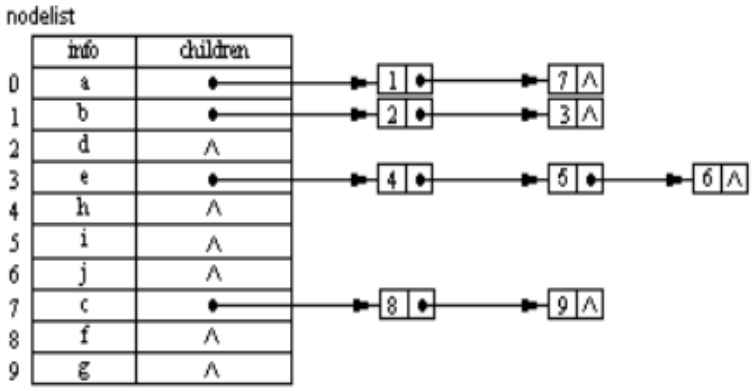
优点：存储密度高，求双亲和最左子女方便
缺点：求右兄弟等运算麻烦。

	info	parent
0	a	-1
1	b	0
2	d	1
3	e	1
4	h	3
5	i	3
6	j	3
7	c	0
8	f	7
9	g	7

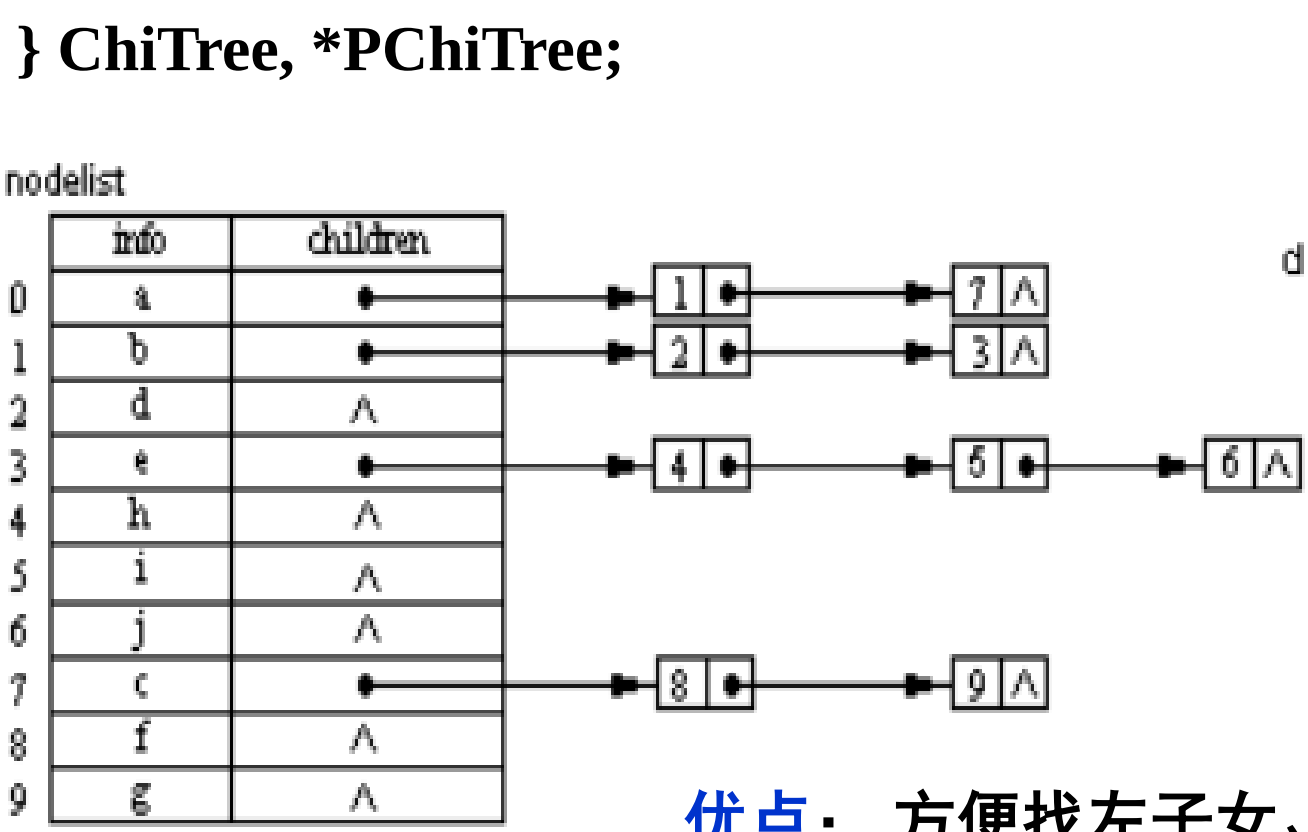
（2）子表表示法

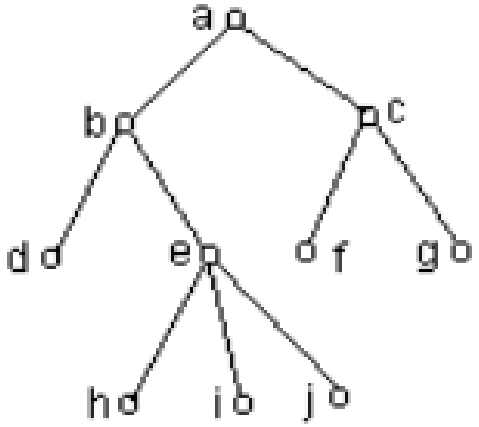
结点表中包含所有结点，其每一元素除了包含结点信息外还包含一个子表，存放该结点的所有子结点。结点表顺序存放，子表用链接表示(按照从左往右次序存放子结点）。

```
typedef struct EdgeNode                    /* 子表中结点的结构 */
{       int       node_position;
        struct  EdgeNode *next;
} EdgeNode;
typedef  struct ChiTreeNode                /* 结点表中结点的结构 */
{       DataType   info;
        EdgeNode *children;
} ChiTreeNode;
```



```
typedef  struct ChiTree                    /* 树结构 */
{        ChiTreeNode node_list[MAXNUM];
         int                root;          /* 根结点的位置 */
         int                n;             /* 结点的个数 */
} ChiTree, *PChiTree;
```





优点： 方便找左子女、找所有子女等

缺点： 找父结点、找右兄弟麻烦

找右兄弟：

找到包含p的子表。在子表中，右边的结点即为右兄弟

```
for (m=0; m < t->n; m++)
{  v = t->nodelist[m].children;
   while (v)
   {    if (v->node_position == p)
        {   if (v->next) return v->next->node_position;
            else          return –1;
        }
        v = v->next;
   }
 }
```

找双亲：

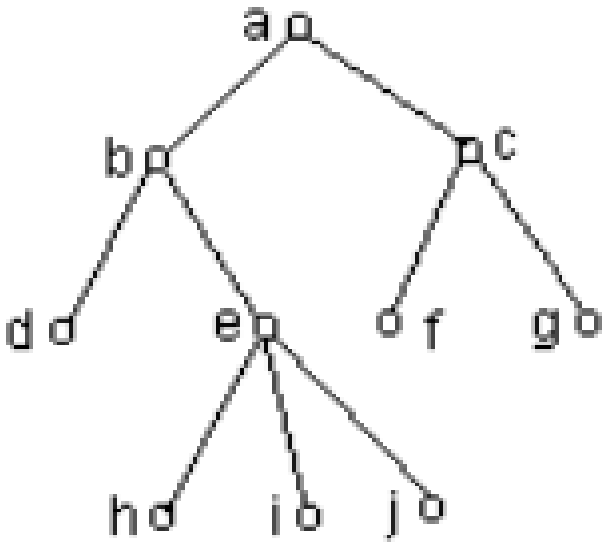
找到包含p的子表。该子表对应的结点即为双亲

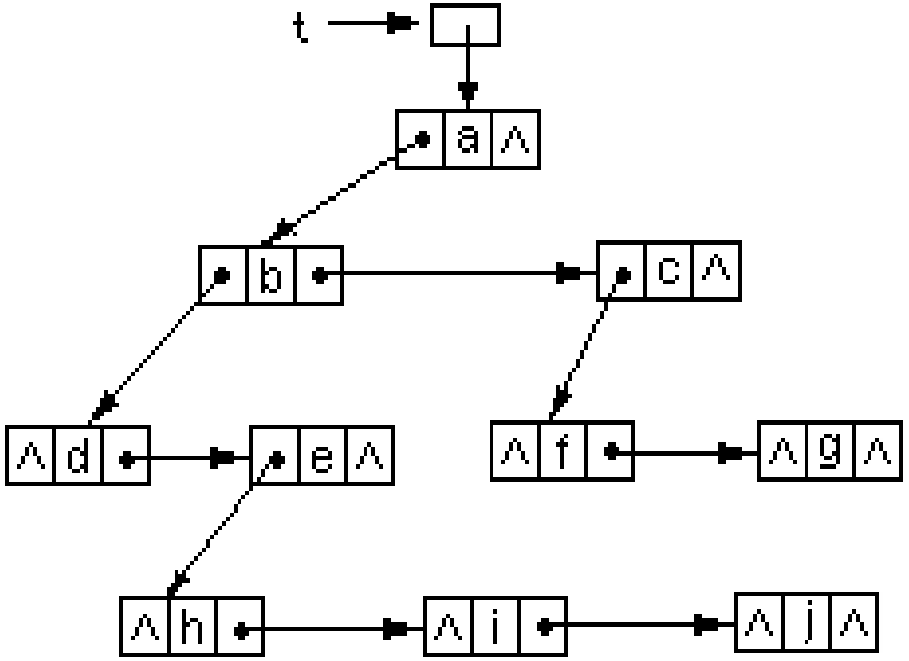
```
for (m = 0; m < t->n; m++)
{   v = t->nodelist[m].children;
    while(v)
    {       if (v->nodeposition == p)    return(m);
            v = v->next;
    }
    return –1;
}
```

如果没有root，如何找根？

（3）长子-兄弟表示法

```
typedef  struct CSNode                  /* 结点结构 */
{        DataType        info;          /* 结点信息 */
         struct CSNode  *lchild;        /* 最左子女 */
         struct CSNode  *rsibling;      /* 右兄弟   */
} CSTree, *PCSTree;
```





优点：方便找子女、找兄弟等运算
缺点：找父结点麻烦

注意： 通过长子-兄弟表示法，将一颗树转换为一颗二叉树。并且在此二叉树中，根结点的右兄弟指针一定为空（根无兄弟），该二叉树根结点无右子树，在树林存储表示中将用到此指针。

找子女：　找到长子，再沿着找右兄弟的指针找所有子女
找父结点：首先找到长子，然后再找父结点。

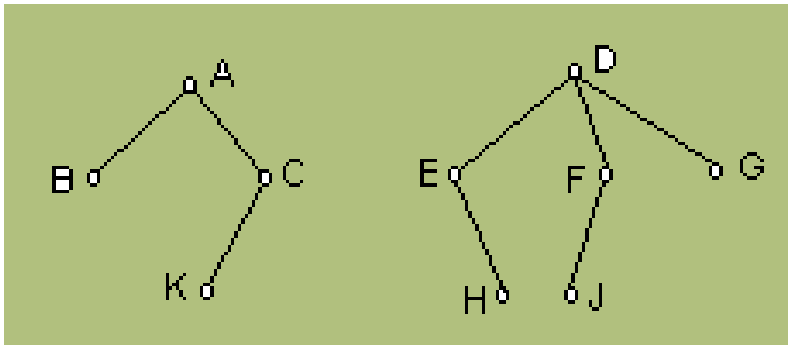
思考：如何修改长子-兄弟存储结构，更方便找双亲？

```
typedef  struct CSNode                  /* 结点结构 */
{          DataType   info;             /* 结点信息 */
             struct CSNode  *parent;    /* 父结点 */
           struct CSNode  *lchild;      /* 最左子女 */
           struct CSNode  *rsibling;    /* 右兄弟   */
} CSTree, *PCSTree;
```

2. 树林的存储表示（与树对应的三种）

- 父结点表示法　　[双亲表示法]
- 子表表示法　　[孩子表示法]
- 长子-兄弟表示法 [孩子兄弟表示法]

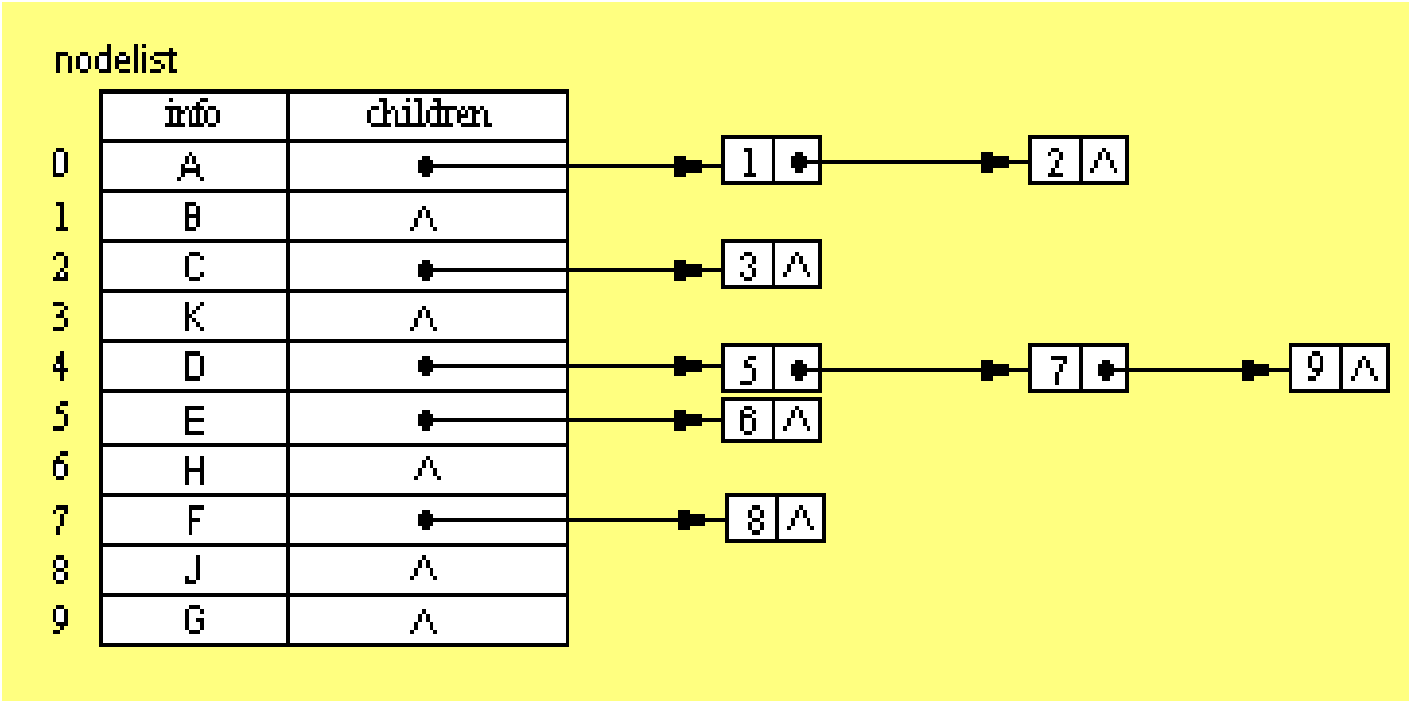
（1）父结点表示法



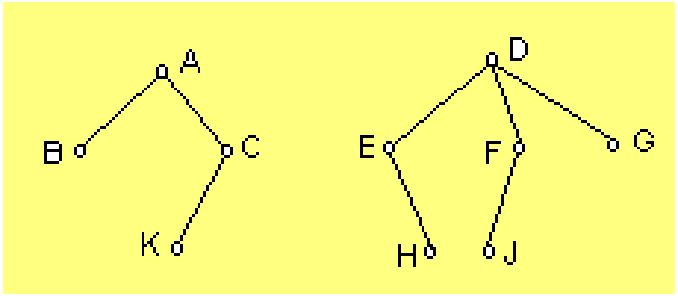
	info	parent
0	A	-1
1	B	0
2	C	0
3	K	2
4	D	-1
5	E	4
6	H	5
7	F	4
8	J	7
9	G	4

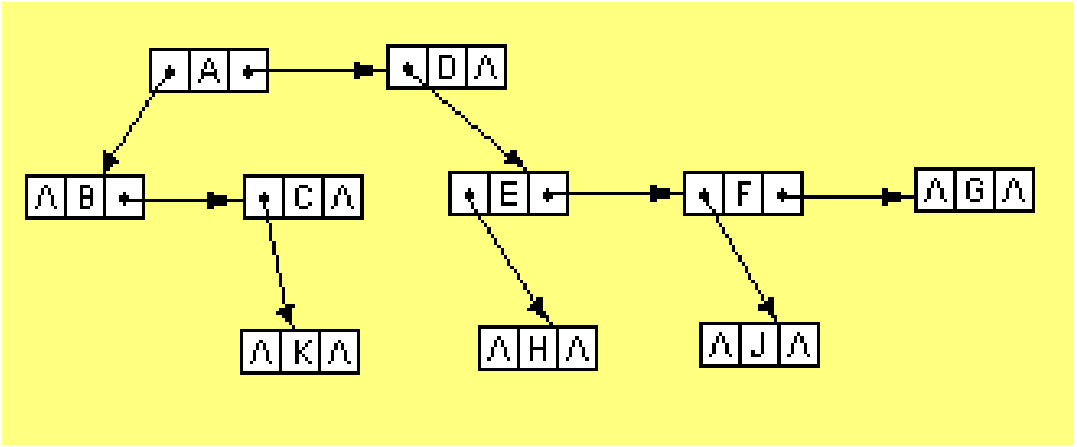
(2) 子表表示法

思考：
如何求森林中包含几棵树？



(3) 长子-兄弟表示法





5.3 树和树林的周游（遍历）

1. 树的周游
2. 树林的周游
3. 与周游结合的存储

周游的定义：按照某种**规则（次序）**访问树或树林中的所有结点，并且每个结点只能访问一次。

本质：将非线性结构转换为元素线性序列。

1. 树的周游

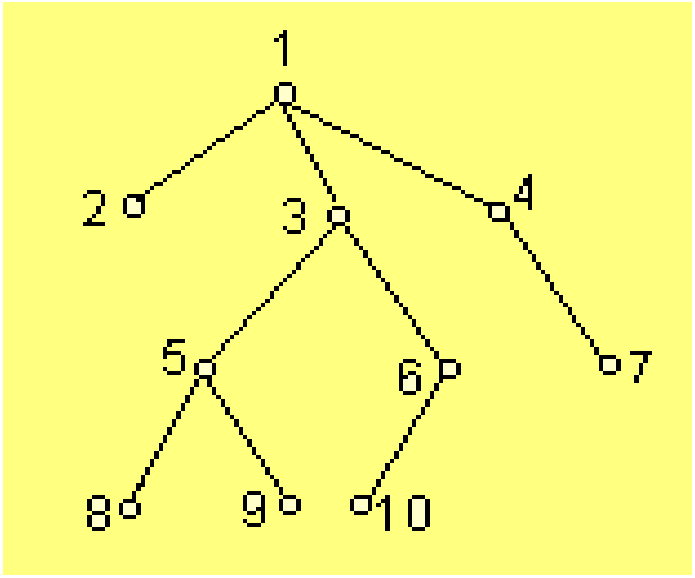
- 按深度方向周游[纵向遍历]

  先根遍历

  中根遍历

  后根遍历

- 按宽度方向周游[横向遍历]



（1）深度方向

先根遍历 – 先访问根，再按照从左往右次序先根遍历根的每颗子树。

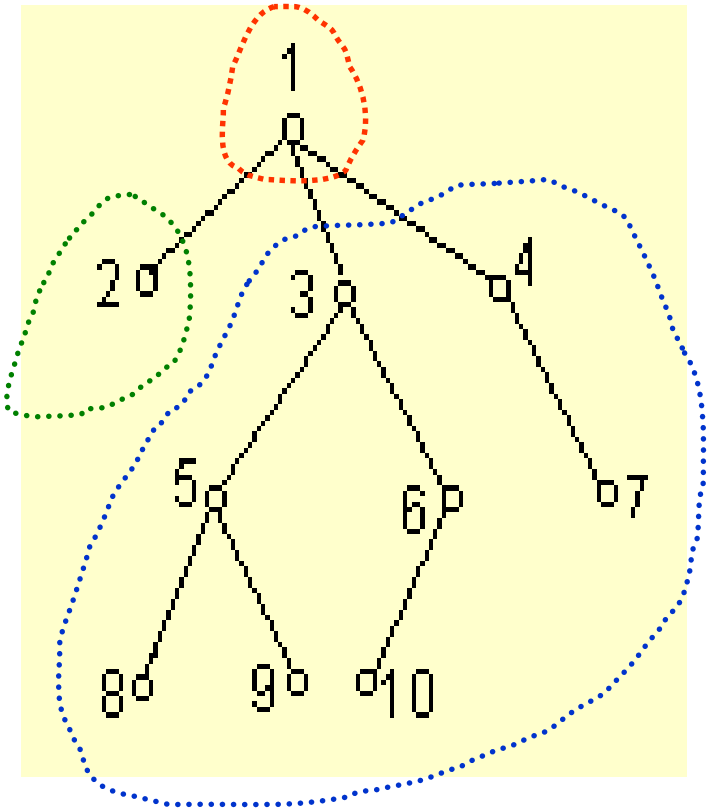
先根序列(1,2,3,5,8,9,6,10,4,7)

中根遍历 – 先中根遍历最左子树，再访问根，再按照从左往右次序中根遍历根的其它子树。

中根次序(2,1,8,5,9,3,10,6,7,4)

后根遍历 – 按照从左往右次序后根遍历根的每颗子树，再访问根

后根次序(2,8,9,5,10,6,3,7,4,1)



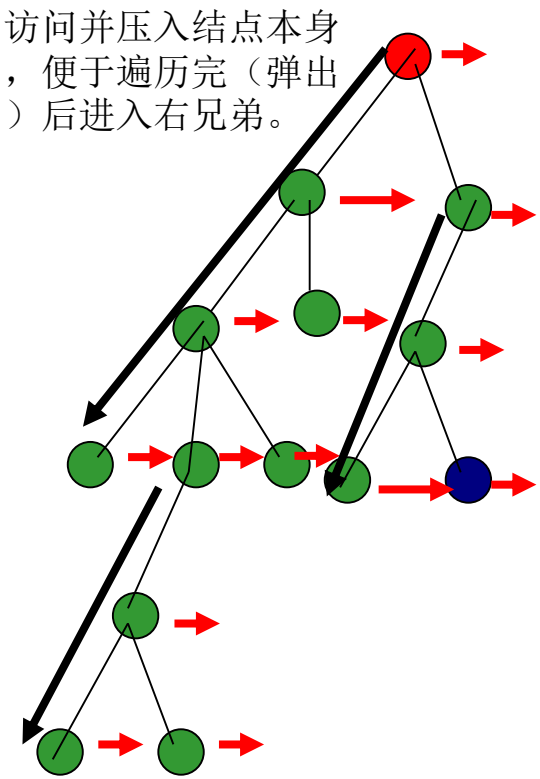
特点： a) 在先、中和后根遍历序列中，树结点的左右次序不变;

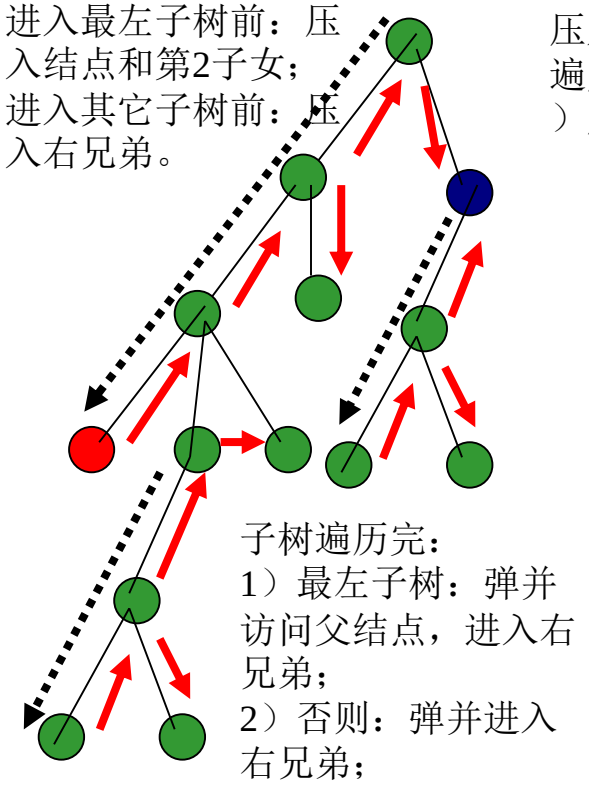
b) 在先根遍历序列中，
结点的所有子孙都紧密排列在该结点的右边;
假定post(n)表示结点n在先根序列中的位置，
desc(n)表示结点n的子孙个数，
则结点x是结点n的子孙的充分必要条件为：

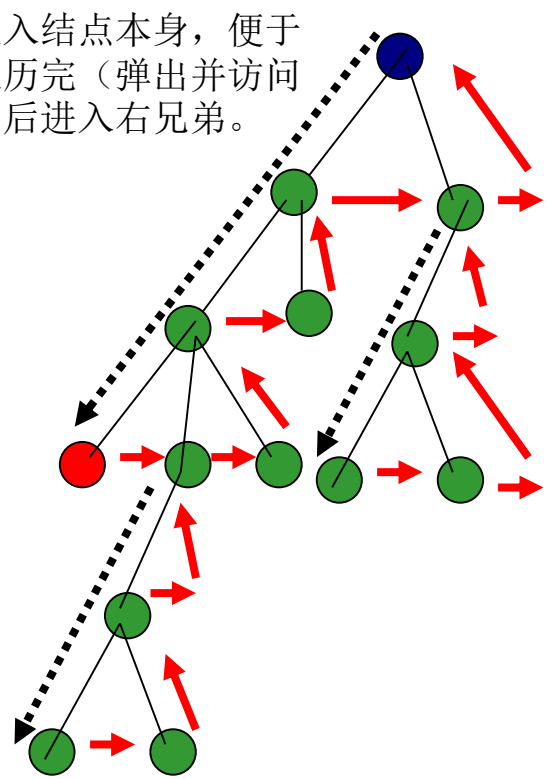
$$post(n)+desc(n) \geqslant post(x) > post(n)$$

c) 在后根遍历序列中，
结点的所有子孙都紧密排列在该结点的左边;
假定post(n)表示结点n在后根序列中的位置，
desc(n)表示结点n的子孙个数，
则结点x是结点n的子孙的充分必要条件为：

$$post(n)-desc(n) \leqslant post(x) < post(n)$$







先根遍历

任何结点，先根遍历完后：
如有右兄弟，进入右兄弟；否则，进入上一层（父结点）的右兄弟。

中根遍历

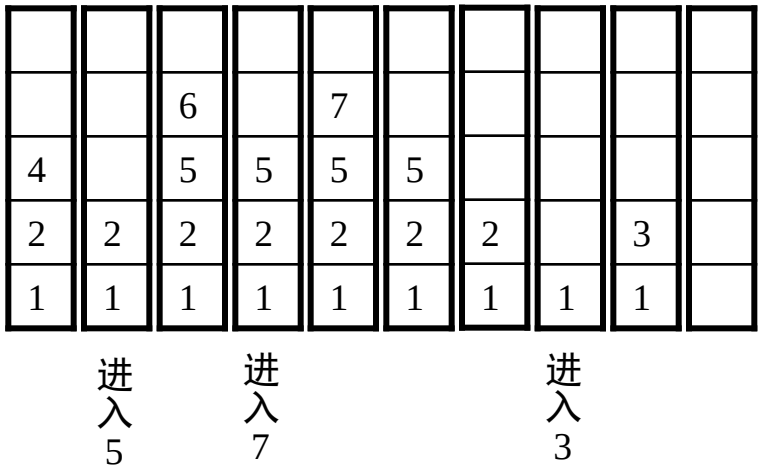
任何结点，中根遍历完（长子子树也已经遍历完）后：
1）如有第2颗子树，进入。
2）否则（也无其它子树），父结点。
第2颗和2后的子树，进入前需要保存右兄弟，以便子树遍历完后，进入右兄弟。

后根遍历

任何结点，后根遍历完后：
如有右兄弟，进入右兄弟；否则，上一层（父结点）

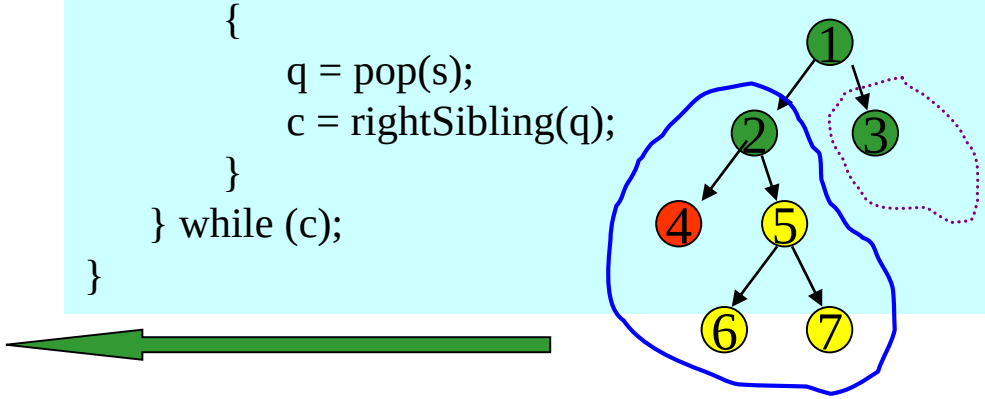
先根遍历（递归）：

```
void PreOrder(Node p)
{
    Node c;
    visit(p);              //先访问根
    c = leftChild(p);   //获取长子
    //按照从左往右次序先根遍历子女
    while (c)
    {
        PreOrder( c);          //先根遍历
        c = rightSibling(c);   //右兄弟
    }
}
```



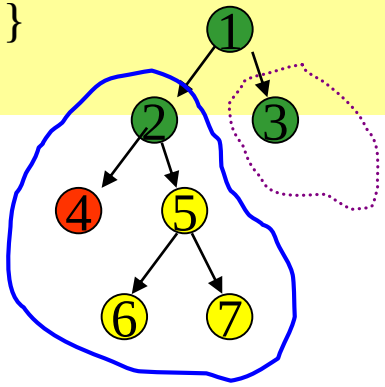
先根遍历（非递归）:结点被弹栈，说明以该结点为根的子树已遍历完。如存在右兄弟，进入右兄弟子树；否则，继续弹栈（弹出双亲）。

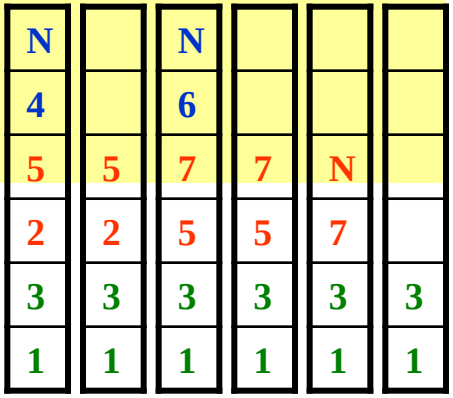
```
void PreOrder(Node p)
{
    Node c, q;
    Stack s = CreateEmptyStack();  //创建空栈
    c = root(p);
    do
    {
        //顺长子链一直下去，边走边访问
        //并压入栈，便于今后右兄弟的访问
        while (c)
        {
            visit(c);              //访问该结点
            push(s, c);            //将该结点压入栈
            c = leftChild(c);   //获取该结点的长子
        }
        while (!c && !isEmptyStack(s))
        {
            q = pop(s);
            c = rightSibling(q);
        }
    } while (c);
}
```



中根遍历（递归）：

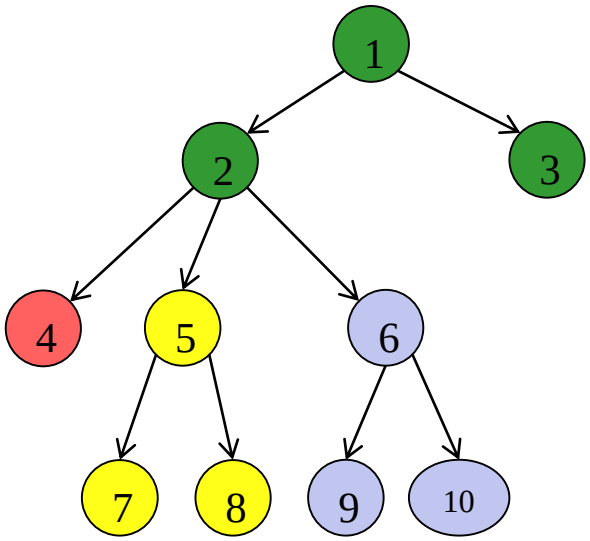
```
void inOrder(Node p)
{
    Node c;
    c = leftChild(p);  //获取该结点的长子
    if (!c)   visit(p);  //不存在，访问该结点
    else
    {
        inOrder(c);     //中根遍历长子
        visit(p);       //访问根
        //中根遍历其它子女
        c = rightSibling(c);
        while (c)
        {
            inOrder(c);
            c = rightSibling(c);
        }
    }
}
```



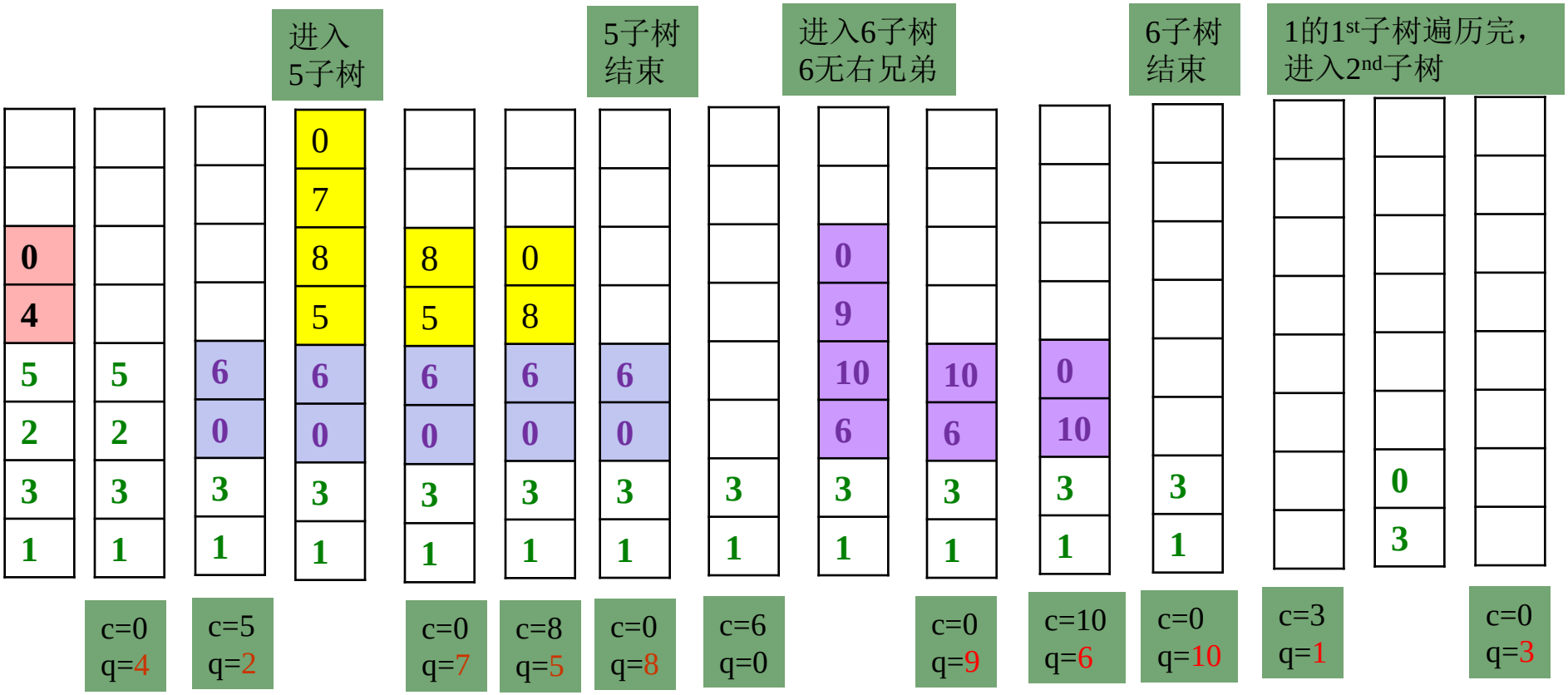


中根遍历（非递归）：结点被弹栈，说明其长子子树已遍历完。此时，访问结点，并进入第2颗子树继续遍历。（如2nd不存在，继续弹栈）。2nd颗和其它子树进入前，如右兄弟存在需要压栈右兄弟，以便子树遍历完后进入右兄弟。

```
void InOrder(Node p)
{   Node c, q;
    Stack s = CreateEmptyStack();  //创建空栈
    c = root(p);
    do
    {
        while (c)                  //顺长子链, 边走边压栈
        {                          //【结点+第2子女】
            push(s, c);            //结点入栈
            c = leftChild(c);      //长子
            q = rightSibling(c);   //第2子女
            push(s, q);            //第2子女入栈
        }
        if (!IsEmptyStack(s))
        {
            c = pop(s);            //第2子女(或下一子女)
            q = pop(s);            //结点（或空）
            if (q) visit(q);       //访问结点
            if (c && (q = rightSibling(c)))
            {                      //下一子女入栈
                push(s, NULL);     //区别第2子女
                push(s, q);        //将来进入[只进入，不访问]
            }
        }
    } while (!IsEmptyStack(s));
}
```

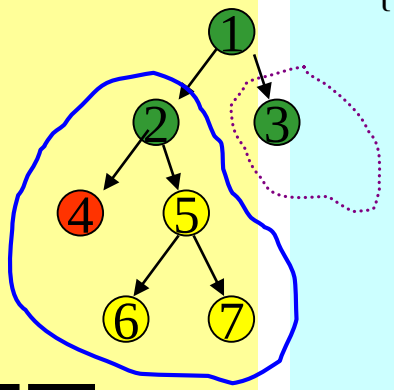


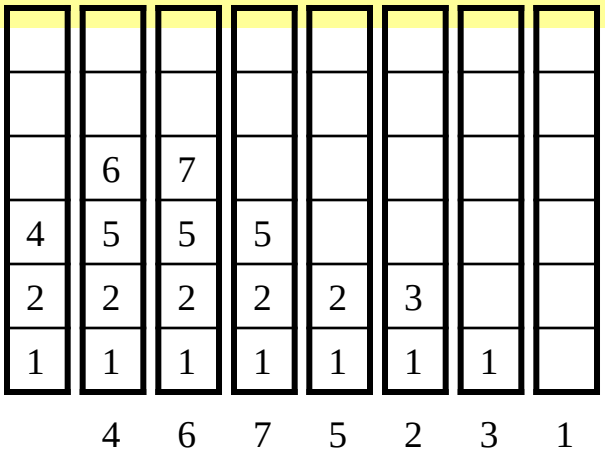
中根遍历（非递归）：结点被弹栈，说明其长子子树已遍历完。此时，访问结点，并进入第2颗子树继续遍历。（如2nd不存在，继续弹栈）。2nd颗（含2后的其它子树）进入前，如右兄弟存在需要压栈右兄弟，以便子树遍历完后进入右兄弟继续遍历。【注：如是二叉树，2nd无右兄弟】



后根遍历（递归）：

```
void postOrder(Node p)
{
    Node c;
     c = leftChild(q);  //获取该结点的长子，
    //然后按照从左往右的次序
    //后根遍历该结点的所有子女
    while (c)
    {
        //后根遍历
        postOrder( c);
        //后根遍历右兄弟
        c = rightSibling(c);
     }
     visit(p);  //最后访问根
}
```





后根遍历（非递归）：站点被弹栈，说明其所有子树已经遍历完。此时，需要访问结点，并继续右兄弟子树遍历（如右兄弟不存在，弹栈父结点）。

```
void postOrder(Node p)
{
    Node c, q;
    Stack s = CreateEmptyStack();   //创建空栈
    c = root(p);
    do
    {
        while (c)
        {
            push(s, c);              //结点入栈
            c = leftChild(c);        //顺长子链
        }
        if (!IsEmptyStack(s))
        {
            q = pop(s);   visit(q);
            c = rightSibling(q); //右兄弟存在, 进入
            if (!c)                  //右兄弟不存在：上一层
            {
                q = pop(s);  visit(q);
                c = rightSibling(q);
            }
        }
    } while (!IsEmptyStack(s));
}
```

(2) 宽度方向

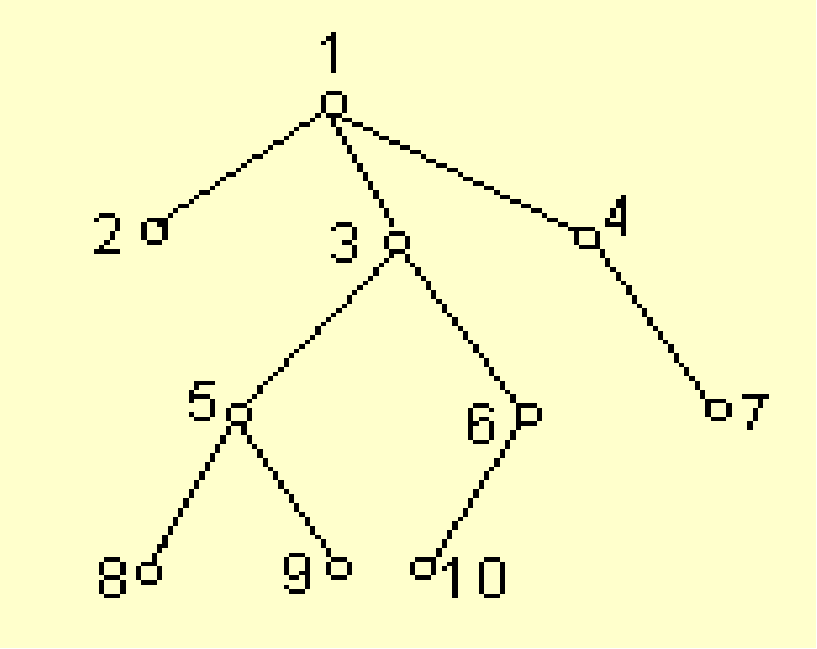
按照层序进行访问：按照从左往右规则，首先访问根，再访问层为1的结点，再访问层为2的结点，…，直到所有结点访问为止。

访问序列：(1,2,3,4,5,6,7,8,9,10)

得到层序序列；

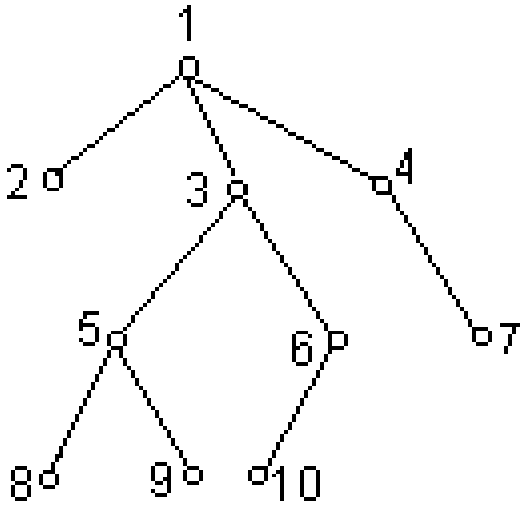
实现方法（采用队列实现）：

- 首先将根送入队列；
- 其后每当从队列取出一个结点访问之后，马上把它的子女按照从左往右的次序送入队列的尾部；
- 重复该过程直到队列为空。



宽度优先遍历算法

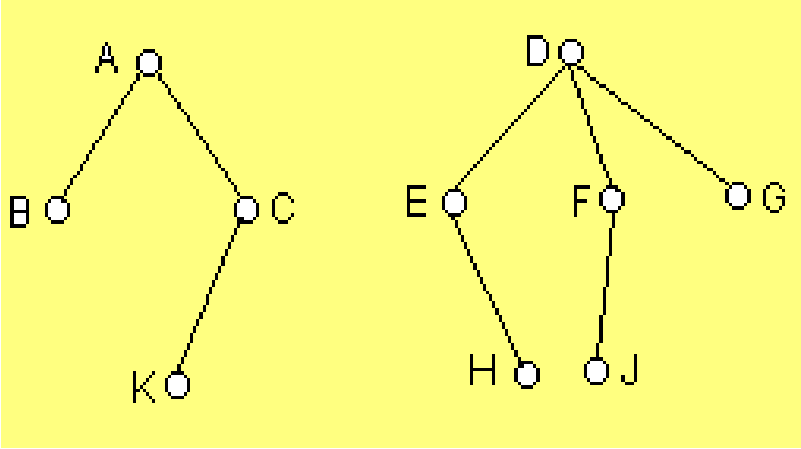
```
//队列控制下的层次遍历
void LevelOrder(Tree t)
{   Node c;
    Queue q;                          //队列元素的类型为Node
    q = CreateEmptyQueue();   //建立空队列
    c = root(t);
    if (!c)   return;
    enQueue(q, c);                    //根入队列
    while (!isEmptyQueue(q))  //队列非空
    {   c = frontQueue(q);        //访问并删除结点
        deQueue(q);
        visit(c);
        //获取该结点的长子，然后将该结点的
        //子女按照从左往右的次序加入队列
        c = leftChild(c);
        while (c)
        {   enQueue(q, c);        //入队列
            c = rightSibling(c); //右兄弟
        }
    }
}
```



	1							
1	2	3	4					
2	3	4						
3	4	5	6					
4	5	6	7					
5	6	7	8	9				
6	7	8	9	10				
7	8	9	10					

……

2. 树林的周游



- **先根**：

  访问第一颗树的根结点；

  先根遍历第一颗树的根结点的子树树林；

  先根遍历除去第一颗树后的树林

  先根遍历序列：（A, B, C, K, D, E, H, F, J, G)

先根遍历第一棵树
先根遍历其它树

- **后根**：

  后根遍历第一颗树的根结点的子树树林；

  访问第一颗树的根结点；

  后根遍历除去第一颗树后的树林

  后根遍历序列：(B, K, C, A, H, E, J, F, G, D)

后根遍历第一棵树
后根遍历其它树

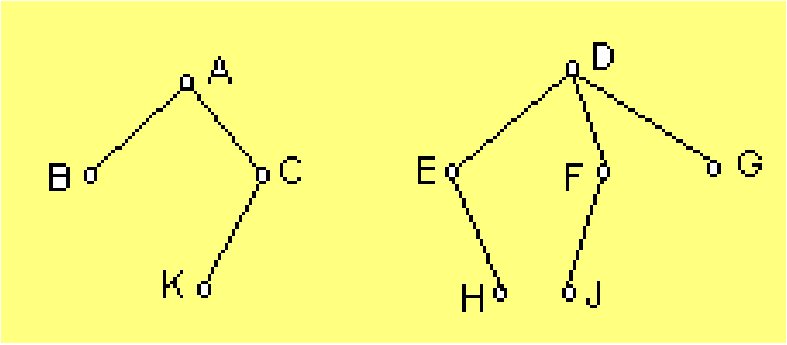
3. 树、树林的其它存储表示[与周游方法结合]

先得到周游结果，再根据结点结构求其它信息。

带右兄弟指针的先根次序表示：

```
struct Node
{  DataType info;
   int ltag;
   Pnode rsibling;
}
```

$$ltag = \begin{cases} 0 & \text{结点无子女} \\ 1 & \text{结点有子女} \end{cases}$$



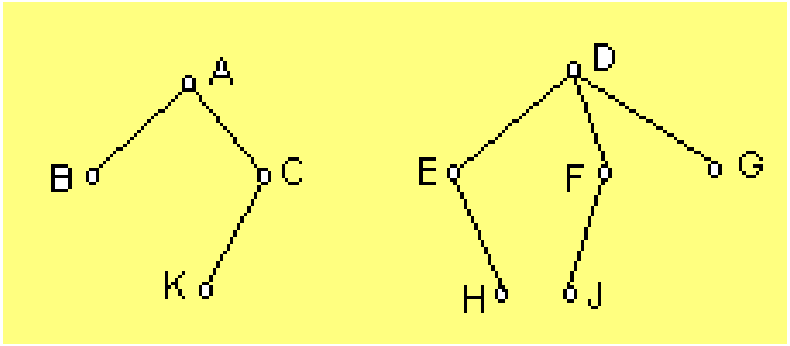
info	A	B	C	K	D	E	H	F	J	G
rsibling		↑			↑			↑		↑
ltag	1	0	1	0	1	1	0	1	0	0

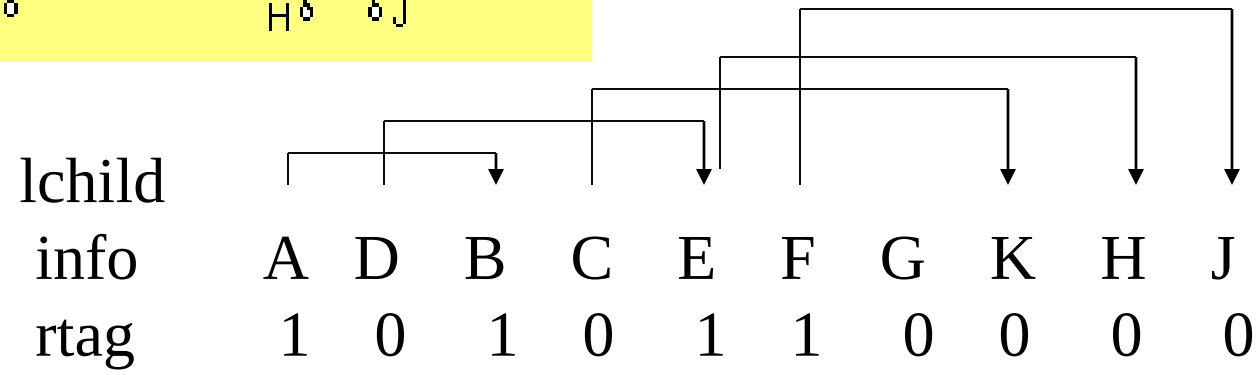
找子女：？
找双亲：？

带双标记的先根次序表示(课后)：

带长子指针的层次次序表示：

```
struct Node
{    DataType info;
     Pnode lchild;
     int rtag;  (0:无右兄弟,1:有)
}
```





找兄弟：rtag = 0(无); rtag = 1: 右边连续rtag=1，最后一个rtag=0

找长兄：上一结点rtag=0，无长兄；
否则找前面连续rtag=1的第一个结点为长兄

找孩子：lchild后，找长子的兄弟

找双亲：先找长兄，再找lchild指向长兄的结点。

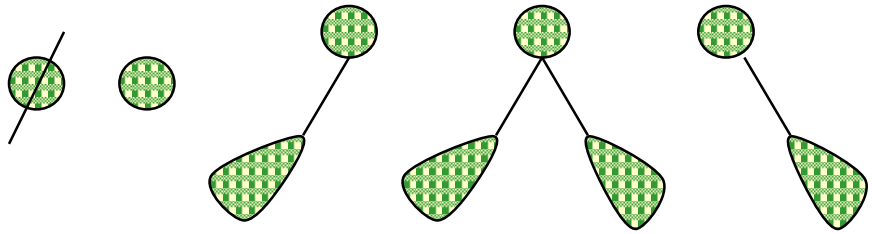
带度数的后根次序表示(课后)：

5.4 二叉树

1. 基本概念
2. 重要性质
3. 基本运算

二叉树的定义：结点的有限集合，该集合或者为空集，或者由一个称为根的结点和两颗互不相交的分别称为根的“左子树”和“右子树”的二叉树组成。

1. 基本概念



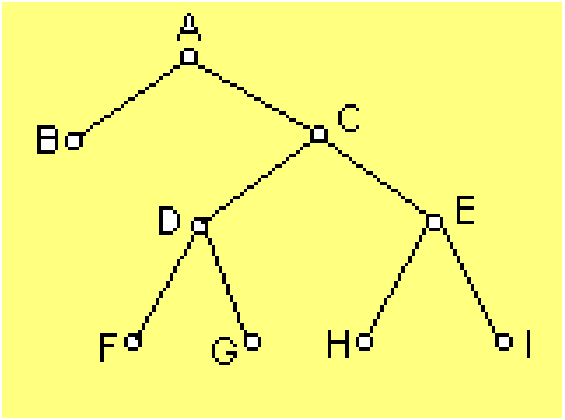
a) 五种基本形态：

b) 二叉树不是树的特殊情形，它们是两个概念。树和二叉树之间最主要的差别是：二叉树中结点的子树要区分为“左子树”和“右子树”，即使在结点只有一棵子树的情况下也要明确指出该子树是左子树还是右子树。

与其它教材的定义有区别

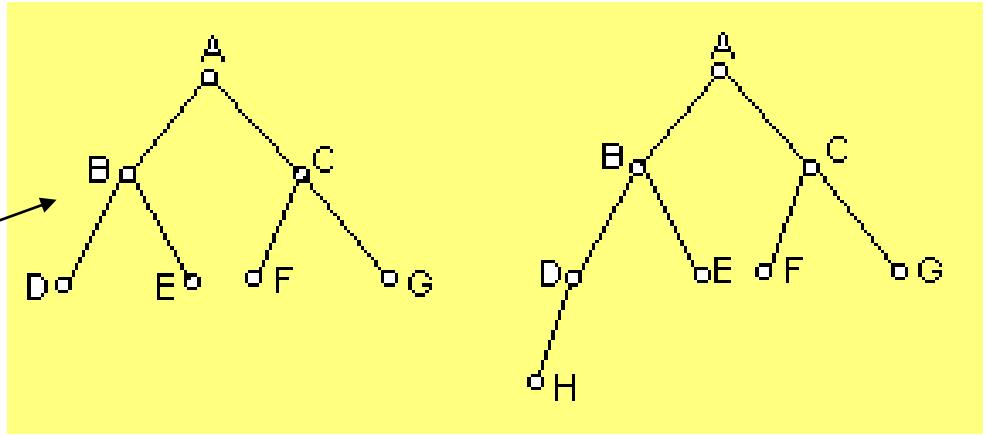
c) 满二叉树：如果一棵二叉树的任何结点或者是树叶，或者有两棵非空子树，则此二叉树称作“满二叉树”。

满二叉树



d) 完全二叉树：如果一棵二叉树至多只有最下面的两层结点度数可以小于2，并且最下面一层的结点都集中在该层最左边的若干位置上，则此二叉树称为“完全二叉树”。完全二叉树不一定是其它教材定义的“满二叉树”。

完全二叉树



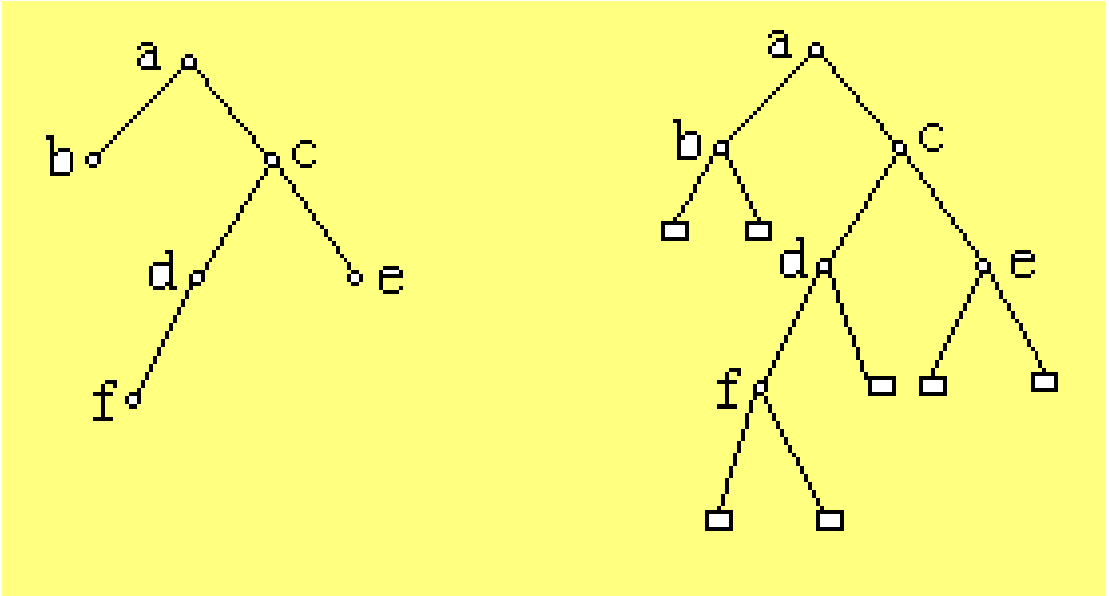
其它教材的满二叉树形态
结点数: $n = 2^k-1$，k为层数

e) 扩充二叉树：把原二叉树的结点都变为度数为2的分支结点，也就是说，不改变度数为2的结点；对于度数为1的结点，则增加一个分支；对于度数为0（树叶）增加两个分支。

在扩充的二叉树里，新增加的“**外部结点**”的个数比原来的“**内部结点**”个数多1。

假定内部结点数为n，则扩充二叉树中有2n条边，2n+1个结点，因此外部结点数目为：2n+1-n=n+1。

2n+1中包含一个根结点



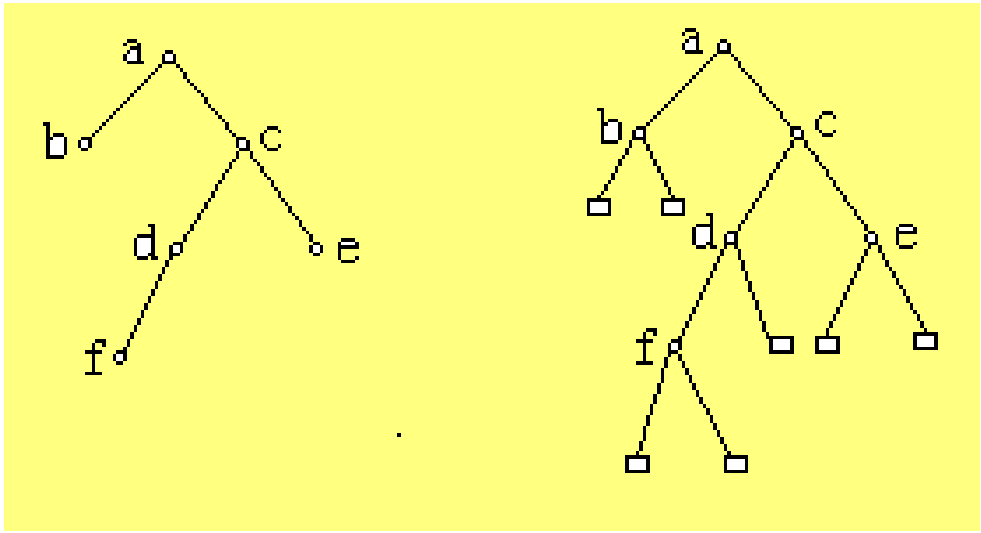
“外部路径长度”E：

在扩充的二叉树里从根到每个外部结点的路径长度之和。

“内部路径长度”I：

在扩充的二叉树里从根到每个内部结点的路径长度之和。

$$E = I + 2n$$



E = 2+2+4+4+3+3+3=21

I = 1+3+2+1+2=9

E = 9+2*6 = 9+12 = 21

2. 二叉树的重要性质

（1）在二叉树的i层上最多有2^i个结点 （$i \geq 0$）

第i层： 2^{i-1}个结点（$i \geqslant 1$）

（2）深度为k的二叉树最多有2^k-1个结点，（$k \geq 0$）

【其它教材中，深度k，有2^k-1个结点：满二叉树】

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

证明：假定度数为1的结点数为n_1，则有： $n = n_0+n_1+n_2$

除根外，其它结点都有一条分支进入，则分支数为：

$B = n-1$

又所有的分支都有度数为2和度数为1的结点发出，

因此 $B = 2n_2+n_1$

综合上述三式得：$n_0=n_2+1$

（4）具有n个结点的完全二叉树的深度k为 $\lceil \log_2(n+1) \rceil$

（5）如果对一棵有n个结点的完全二叉树按从上到下、从左往右的顺序对树中所有结点从1开始编号，则对任一结点i（1≤i≤n)，有：

【a】i=1，结点i是根；i>1, 其双亲结点是 $\lfloor i/2 \rfloor$

【b】2i>n，结点i无左孩子；否则，其左孩子是结点2i

【c】2i+1>n，结点i无右孩子；否则，其右孩子是结点2i+1

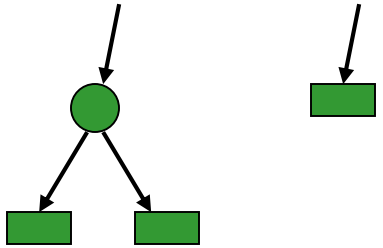
（6） 在扩充二叉树中，外部结点的个数比内部结点个数多1

（7） 对于任意扩充二叉树，外部路径长度E和内部路径长度I满足如下关系：E=I+2n，其中n为内部结点个数。

证明：n=0， E=0， I=0，结论成立

n=1， E=2， I=0，结论成立

假定对于任意n，有：$E_n=I_n+2n$



现在考虑n+1个内部结点的扩充二叉树。

在n+1个内部结点的扩充二叉树中删除一个原来二叉树作为树叶的、路径长度为K的内部结点，使之成为包含n个内部结点的扩充二叉树。由于删除了一个路径长度为K的内部结点，**内部路径长度**的变化为：

$$I_n=I_{n+1}-K$$

删除了一个路径长度为K的内部结点导致删除了两个外部路径长度为K+1的外部结点，增加了一个路径长度为K的外部结点，**外部路径长度**的变化为：

$$E_n= E_{n+1} -2(K+1)+K = E_{n+1}-K-2$$

即， $E_{n+1} = E_n+K+2= \underline{I_n}\underline{+2n}+K+2= \underline{I_n}\underline{+K}+2(n+1)= I_{n+1}+2(n+1)$

因此，n+1时结论成立。

返回二叉链表

3. 二叉树的基本运算

- 创建一棵空二叉树；
- 判断某棵二叉树是否为空；
- 求二叉树的根结点，若为空，则返回一特殊值；
- 求二叉树中某个指定结点的父结点，当指定结点为根时，返回一特殊值；
- 求二叉树中某个指定结点的左子女结点，当指定结点没有左子女时，返回一特殊值；
- 求二叉树中某个指定结点的右子女结点，当指定结点没有右子女时，返回一特殊值；
- 二叉树的周游，即按某种方式访问二叉树中的所有结点，并使每个结点恰好被访问一次。

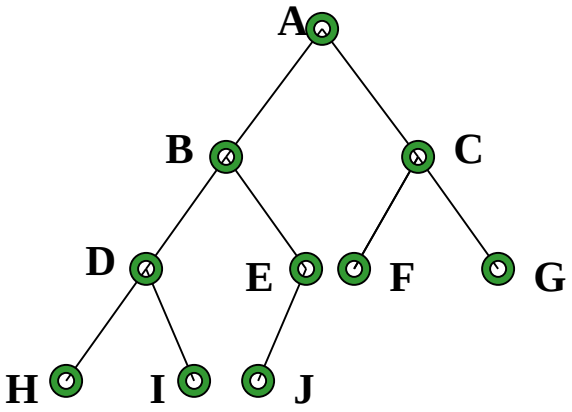
5.5 二叉树的存储表示

1. 顺序表示
2. 链接表示

1. 顺序表示

（1）完全二叉树

根据性质5，结点序号可以反映结点的逻辑关系，因此可以用一维数组按照从上到下、从左往右的顺序存储树中所有结点，通过数组元素下标反映完全二叉树中结点的逻辑关系。



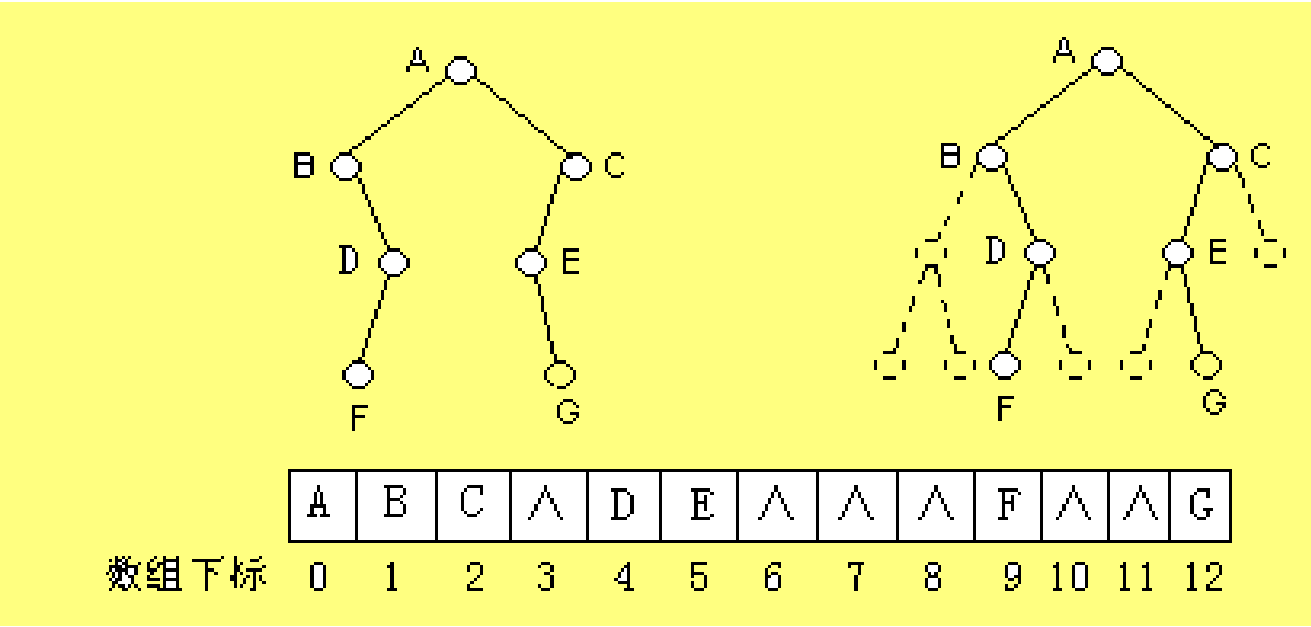
i的左右子女
(2i+1, 2i+2)
i的双亲 (i-1)/2

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

数组下标：

（2）一般二叉树

如果仍然采用上述完全二叉树方式存储，则数组下标不能反映结点之间的逻辑关系。为此，可以首先对一般的二叉树进行改造，增加一些不存在的空结点，使之成为完全二叉树，然后再用一维数组顺序存储，增加的空结点用空表示。



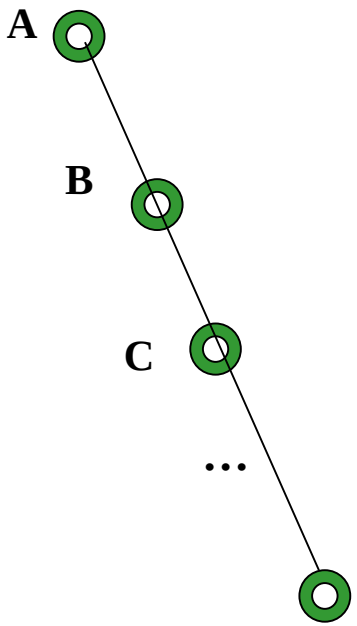
父结点：(i-1)/2，左子女：2i+1，右子女：2i+2

与性质5有差别（从0开始编号，性质5从1开始）

问题：对于完全或近似完全的二叉树，宜采用上述顺序结构存储。但对于一般二叉树，如果增加的空结点非常多，则容易造成空间浪费，此时不宜采用上述方法存储。

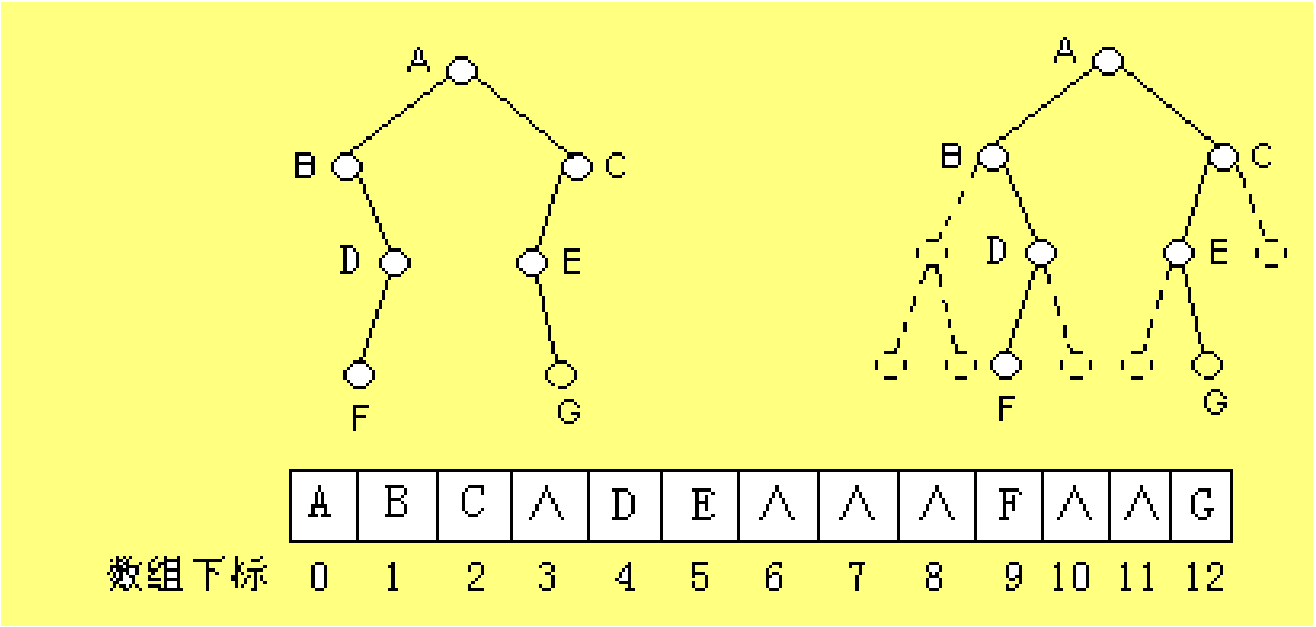
（3）特殊二叉树

对于一种特殊情况，如果二叉树为右单支树[深度为k]，
则需要一个长度为2^k-1的一维数组，
造成（2^k-1）-k个结点浪费。
如果k=6，则有57个结点空间浪费。



顺序存储表示如下：

```
typedef struct SeqBTree
{       DataType    nodelist[MAXNODE];
        int   n;          /* 改造成完全二叉树后 结点的个数 */
}SeqBTree, *PSeqBTree;
```



2. 链接表示

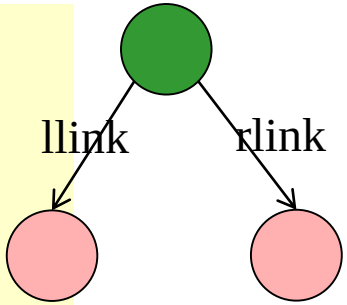
（1）二叉链表

每个结点结构为：

llink	info	rlink

其中：info项存储结点信息，llink指向左孩子，rlink指向右孩子。

```
typedef struct BinTreeNode
{           DataType            info;
            struct BinTreeNode  *llink;
            struct BinTreeNode  *rlink;
}BinTreeNode, *PBinTreeNode, BinTree, *PBinTree
```



基本运算实现

n个结点的二叉链表中有n+1个空指针。通过扩充二叉树中外部结点个数可以证明

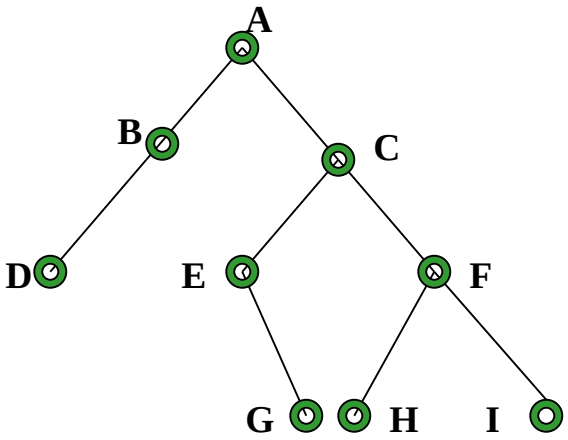
（2）三叉链表

每个结点结构为：

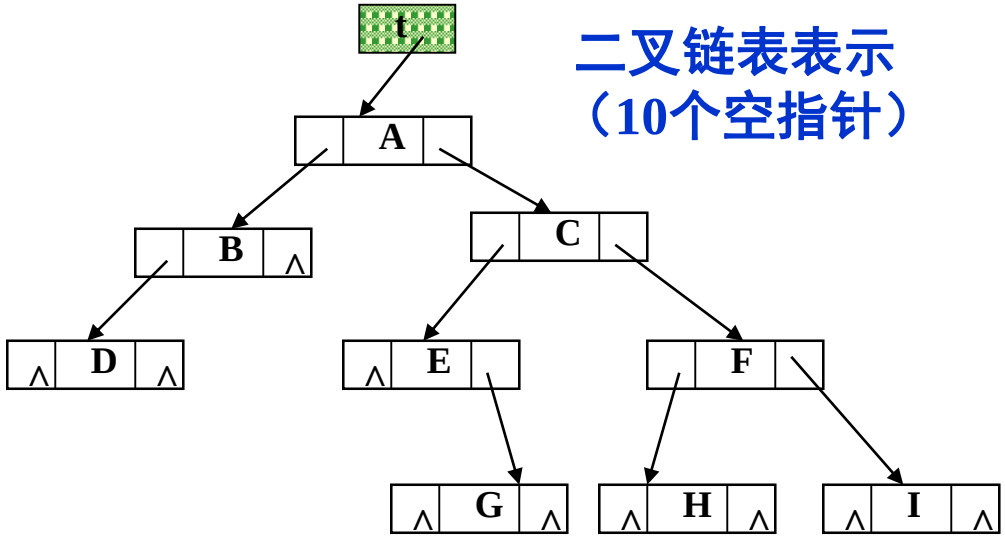
二叉链表每个结点增加指向父结点的指针plink，便得到三叉链表存储表示。

三叉链表对于找父结点等操作非常方便，但增加了空间要求。

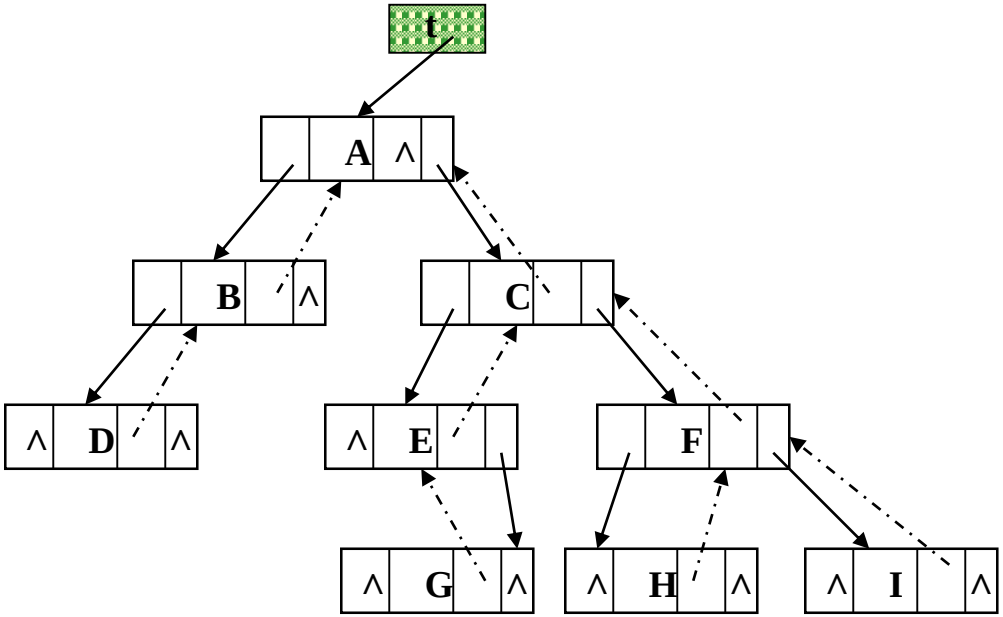
```
typedef struct BinTreeNode
 {              DataType              info;
                struct BinTreeNode   *llink;
                struct BinTreeNode   *rlink;
                struct BinTreeNode   *plink;
 }BinTreeNode, *PBinTreeNode, BinTree, *PBinTree
```



二叉树



二叉链表表示
（10个空指针）



三叉链表表示
（空指针数？）

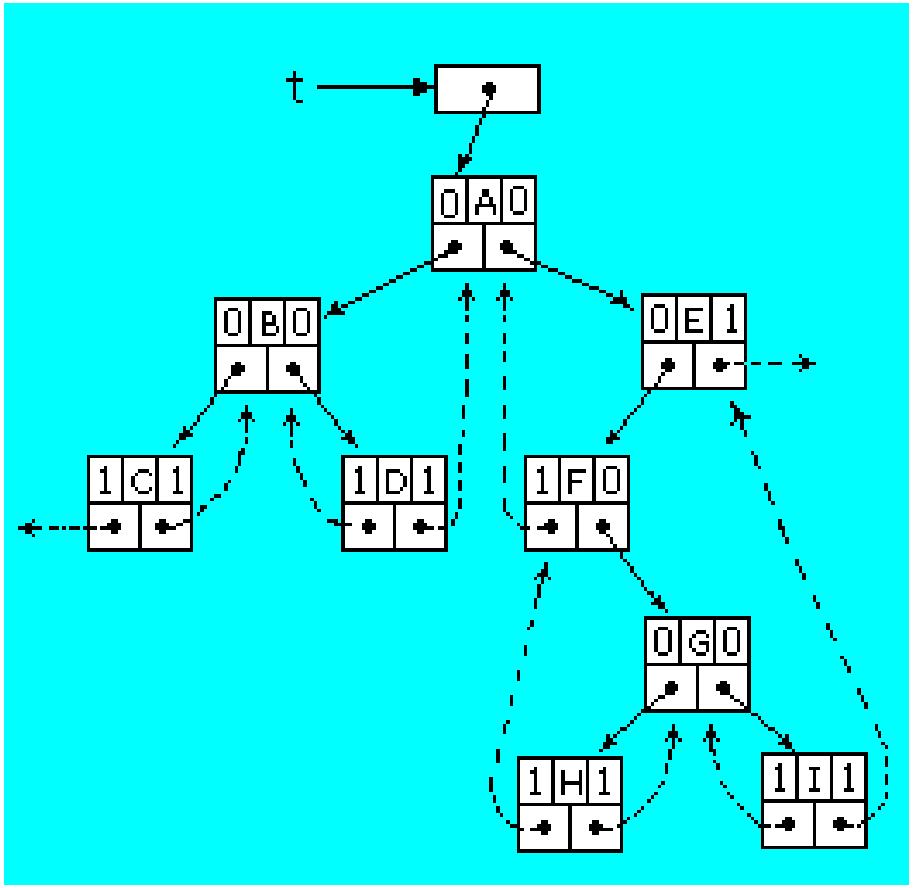
（3）线索链表：修改二叉链表中的空指针，指向其直接前驱或后继的二叉链表。[增加了线索的二叉链表。与遍历方法有关，后面详细讨论]

llink	info	rlink
ltag		rtag

ltag = 0　llink为指针，指向左孩子
1　llink为线索，指向结点的前驱

rtag = 0　rlink为指针，指向右孩子
1　rlink为线索，指向结点的后继

中根遍历线索二叉树



5.6 二叉树的遍历和线索二叉树

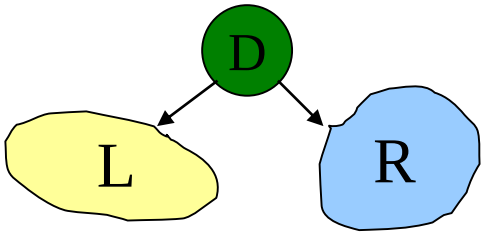
1. 遍历定义
2. 遍历算法
3. 二叉树的建立
4. 线索二叉树

1. 二叉树的遍历

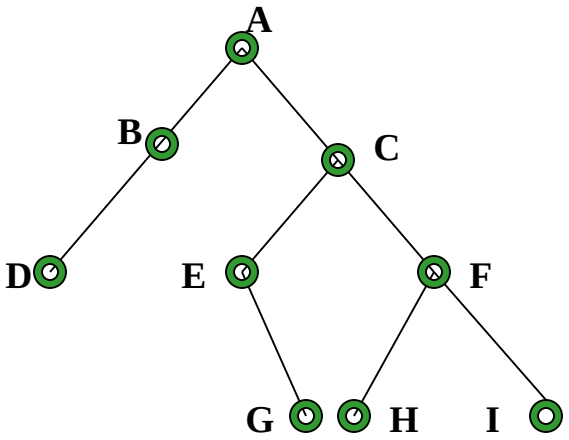
(a) 定义： 按某条搜索路径（规则）访问树中每一个结点，使得每个结点被访问一次且仅被访问一次。通过一次完整的遍历，可以将二叉树结点信息由非线性序列变为线性序列。因此，二叉树的遍历过程实际上是非线性结构线性化的过程。

(b) 遍历方法： 由二叉树定义可知，二叉树有三部分组成：根、左子树和右子树。因此，遍历整个二叉树实质上就是按照某种次序依次遍历这三部分。

假定D、L、R分别表示遍历根、遍历左子树、遍历右子树，按照排列组合，有6种遍历方法：DLR, LDR, LRD, DRL, RDL, RLD。



如果限定左右子树的访问按照先左后右，则上面6种只有3种符合，即：DLR, LDR, LRD，分别称为：先根遍历、中根遍历、后根遍历。



(c) 先根遍历

根—>先根遍历左子树—>先根遍历右子树

遍历结果：ABDCEGFHI

(d) 中根遍历

中根遍历左子树—>根—>中根遍历右子树

遍历结果：DBAEGCHFI

(e) 后根遍历

后根遍历左子树—>后根遍历右子树—>根

遍历结果：DBGEHIFCA

表达式：(a-b)÷(c+d)的二叉树形式如右图：

则其DLR遍历为：÷-ab+cd [前缀表达，波兰式]

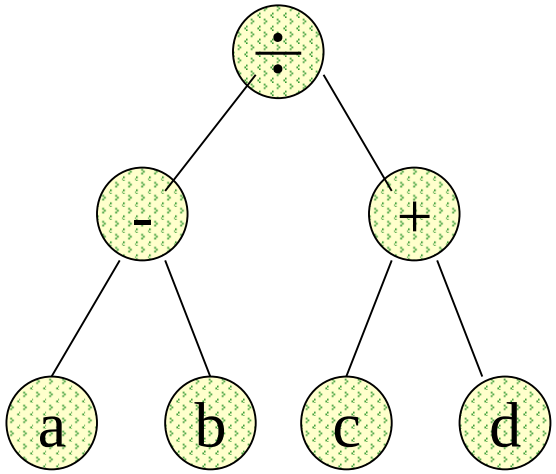
LDR遍历为：a-b÷c+d [中缀表达]

LRD遍历为：ab-cd+÷ [后缀表达，逆波兰式]

任何一个算术表达式，都可以给出其对应的二叉树形式，其遍历结果可以得到表达式的各种表达形式。

终端结点：操作数

分支结点：运算符



2. 二叉树的遍历算法

[1] 递归算法

a)先根　　b)中根　　c)后根

[2] 非递归算法

利用栈实现遍历非递归算法。栈保存遍历过程中结点或结点的左右孩子。数组S[M]作为栈，Top为栈顶指针，假定栈空间足够大，不会产生溢出问题。

与树类似，但有差别！
二叉树基本运算中，无找右兄弟运算。
如有：某些遍历更简单！但需要选择合适的存储表示。

```
Void PreOrder(BinTree t)
{
    if (!t) return;
    visit(t);
    PreOrder(t->llink);
    PreOrder(t->rlink);
}
```

```
Void InOrder(BinTree t)
{
    if (!t) return;
    InOrder(t->llink);
    visit(t);
    InOrder(t->rlink);
}
```

```
Void PostOrder(BinTree t)
{
    if (!t) return;
    PostOrder(t->llink);
    PostOrder(t->rlink);
    visit(t);
}
```

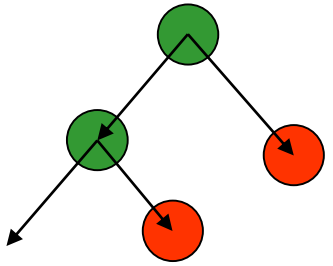
a)先根

思想：从根开始，沿左子树一直走到末端为止，在走的过程中访问所遇结点，并依次将所遇**结点的非空右孩子进栈**。当左子树结点全处理完后，从栈顶退出某结点的右孩子，此时该结点的左子树已经遍历完，再按照上述过程遍历结点的右子树，如此重复直到栈空为止。

先根非递归算法

也可压结点本身，弹栈时（根+左子树已经遍历完）

通过右孩子进入右子树。【类似树的先根遍历】



b) 中根

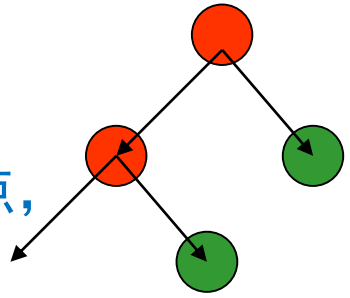
思想：与先根遍历基本类同，只是在沿左分支（左子树）向前搜索过程中将遇到的**结点进栈**，待遍历完左子树后，从栈顶退出结点并访问，然后再遍历右子树。

中根非递归算法

由于只有左、右子女，与树相比，简单。

也可以同时压右子女，弹栈时（左子树已经遍历完）访问结点，

通过右孩子进入右子树。【类似树的中根遍历】



c) 后根

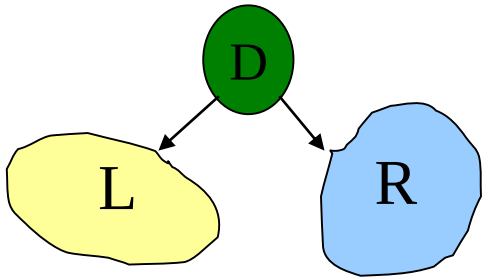
思想：使用栈实现后根遍历要比先、中根遍历复杂。在后根遍历中，当搜索指针指向某个根结点时，不能马上进行访问，而先要遍历左子树，所以要求根结点进栈保存。当遍历完左子树后，再次搜索到该结点时，还不能进行访问，还要遍历其右子树。所以，**需要再次将该结点进栈**保存。为了**区别同一结点的两次入栈**，需要一个特别的标志：1表示该结点首次进栈[遍历左子树前入栈]，2表示第二次进栈[遍历右子树前入栈]。为此，有两种办法实现：

** 设立两个栈，一个栈s1[M]用于存放结点，一个s2[M]栈用于存放结点进栈标志。

** 修改原来的栈，增加标志项（教材中的方法）

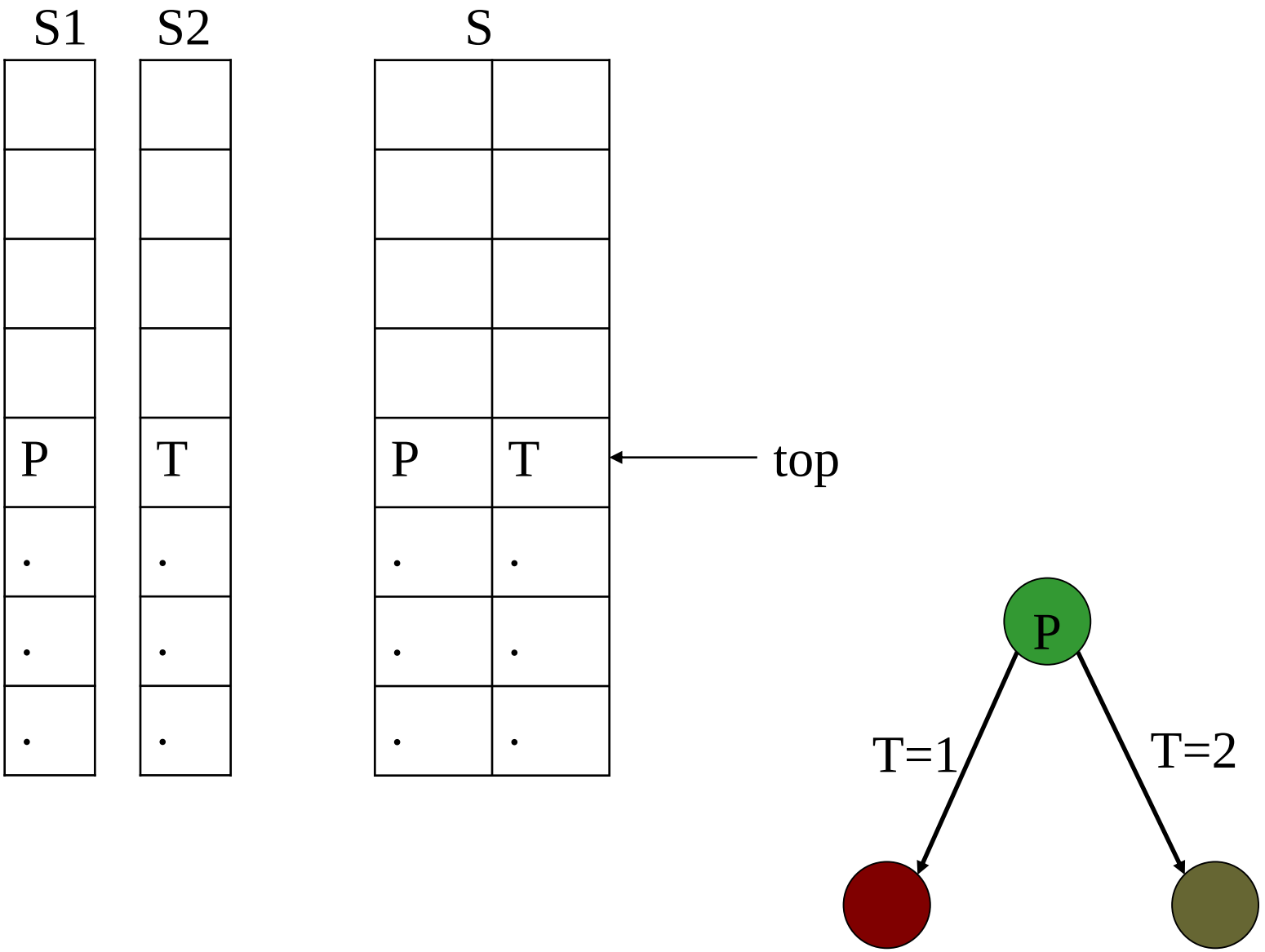
后根非递归算法

教材中还给出了一种牺牲资源的后根遍历算法。



是否可参考树的后根遍历实现？

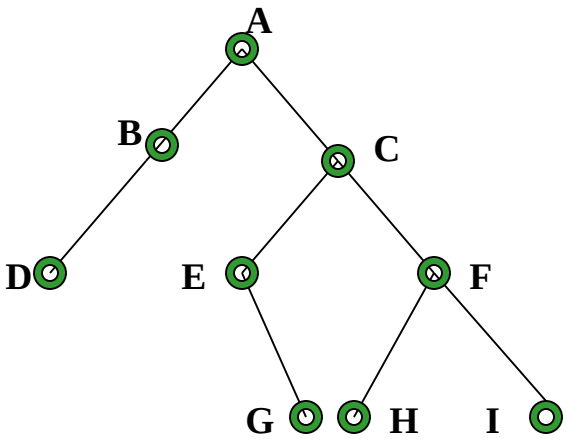
左子树遍历完后，如何进入右子女。二叉树无右兄弟基本运算！


S1
S2
S
P
T
P
T
top
.
.
.
.
.
.
.
.
.
P
T=1
T=2

3. 二叉树的建立

“遍历”是二叉树各种操作的基础，可以在遍历过程中进行各种操作，如可以在遍历过程中生成结点，从而建立一棵二叉树。

顺序存储时：按照顺序要求，把二叉树中的结点加以扩充，转换成一个完全二叉树形式，然后按层次周游顺序把各个结点的信息输入到nodelist数组中即可。



二叉链表存储：按先根遍历次序生成二叉树：

例如要生成右面的二叉树，只要按照下列顺序输入字符，即可建立相应的二叉链表。

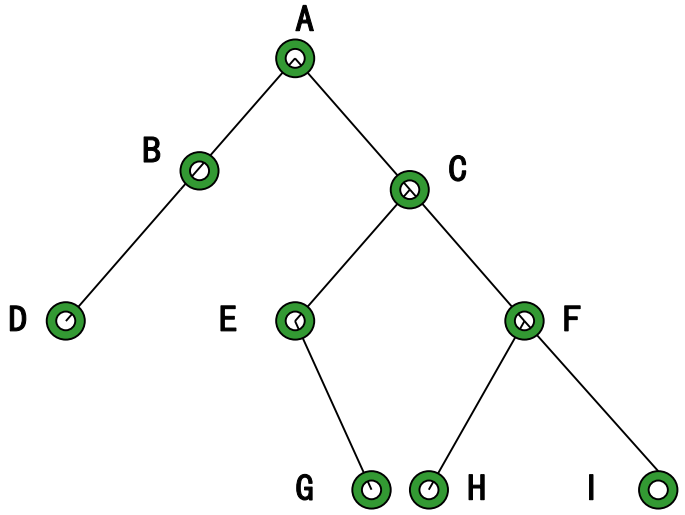
ABD@@@CE@G@@FH@@I@@

其中，@为空结点(对应于扩充二叉树中的外部结点)。

```
BinTree CreateBinTree()
{
    BinTree t;
    char ch;

    scanf("%c", &ch);
    if (c == '@')  t = NULL;
    else
    {
        t = (BinTree)malloc(sizeof(BinNode));
        t->data = ch;                    //根
        t->llink = CreateBinTree();  //左子树
        t->rlink = CreateBinTree(); //右子树
    }
    return t;
}
```

对于每个结点，需要给出其左右子女结点信息。



4. 线索二叉树

（1） 问题的提出

遍历可以将二叉树中结点排列为一个线性表，因此结点的前驱、后继可以在此线性表中得到。但是，二叉树的存储结构中并没有反映这种逻辑上的关系，只能通过对二叉树进行遍历得到。

问题：如果通过一次遍历，保存主要信息，后面重复遍历时利用这些信息，则可大大加快后面的遍历速度。如何在遍历过程中保存结点的前驱、后继结点信息？

解决办法1：每个结点增加指向前驱、后继结点的指针，遍历过程中完成指针的设置。缺点：存储空间增加，存储要求提高，降低空间效率来提高时间效率。

解决办法2：前面证明过：在n个结点二叉树的二叉链表存储结构中，有n+1个空指针。修改这些空指针让其指向前驱或后继结点，空llink指向前驱，空rlink指向后继。

线索：　　指向前驱或后继的指针称为线索。

线索树：　加进线索的二叉树

线索化：　修改空指针为线索的过程为线索化，即通过线索化将二叉树变为线索二叉树。

线索化与二叉树的遍历规则相互对应，本节主要讨论中序遍历下的线索化。

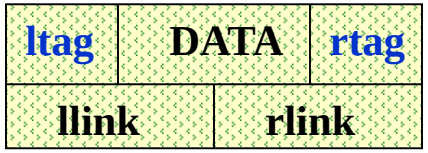
在线索二叉树中，指针有两类： 指向其左右孩子的指针和指向前驱、后继的线索。

如何区分它们？

每个结点增加两个**标志项**，分别标志左右指针的类别：

ltag = 0　llink为指针，指向左孩子

　　1　llink为线索，指向结点的前驱

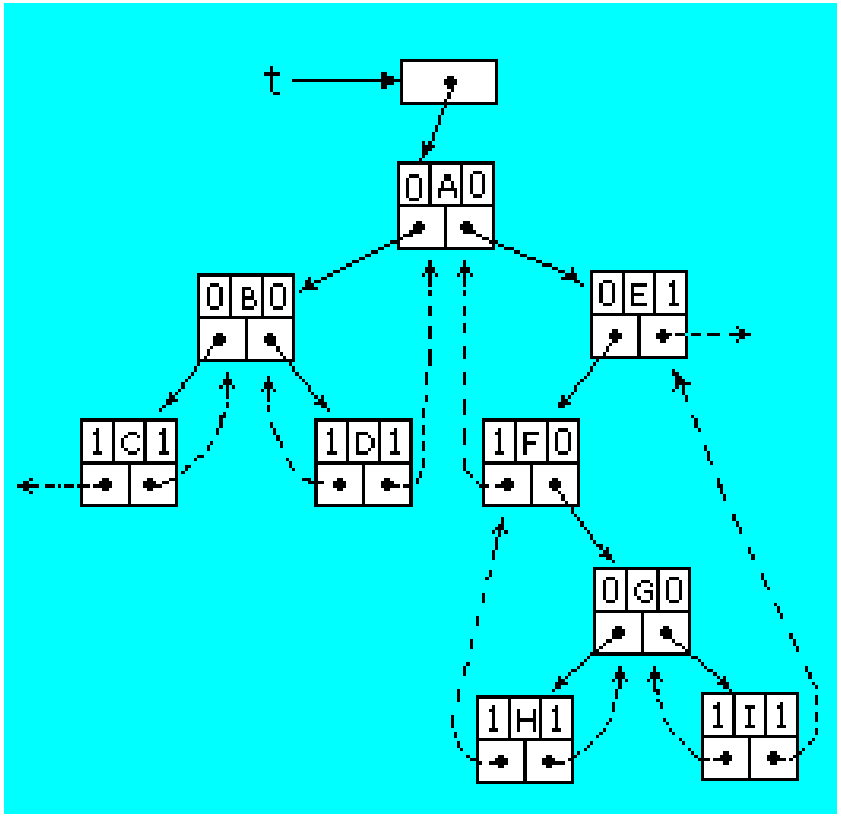


rtag = 0　rlink为指针，指向右孩子

　　1　rlink为线索，指向结点的后继

（2）线索树结构描述：

```
typedef struct ThrTreeNode
{ DataType      info;
  struct ThrTreeNode  *llink, *rlink;
  int           ltag, rtag;
}ThrTree, *PThrTree, *PThrTreeNode;
```



(3) 中序线索化算法

给定一棵树，要将它按照中序线索化，其思想就是按照中序遍历原则遍历该二叉树，在遍历过程中用线索代替空指针。

在未线索化之前，所有的llink和rlink皆为指针，因此所有的ltag和rtag为0。

假定当前正在访问的结点为p，pr指向其中序遍历前驱结点[上次刚访问的结点]，修改线索包括两步：

[a] 如果pr的右指针为空，
修改pr的右指针指向p [p为pr的后继]；

[b] 如果p的左指针为空，
修改p的左指针指向pr[pr为p的前驱]。

要实现非递归中序遍历，需要一个附加的控制栈，
栈结构如下：

```
#define MAXNUM  1000
struct SeqStack
{   PThrTreeNode s[MAXNUM];          //顺序栈
    int  t;                          //栈顶
}
type struct SeqStack *PSeqStack;
```

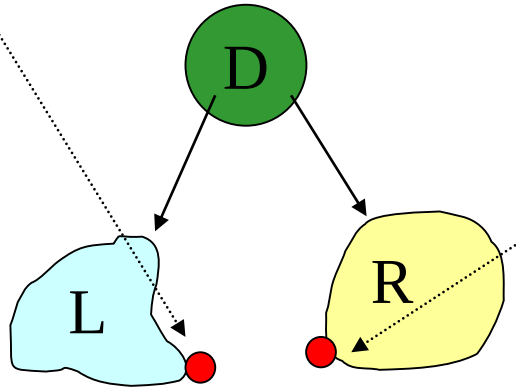
[中序线索化算法]

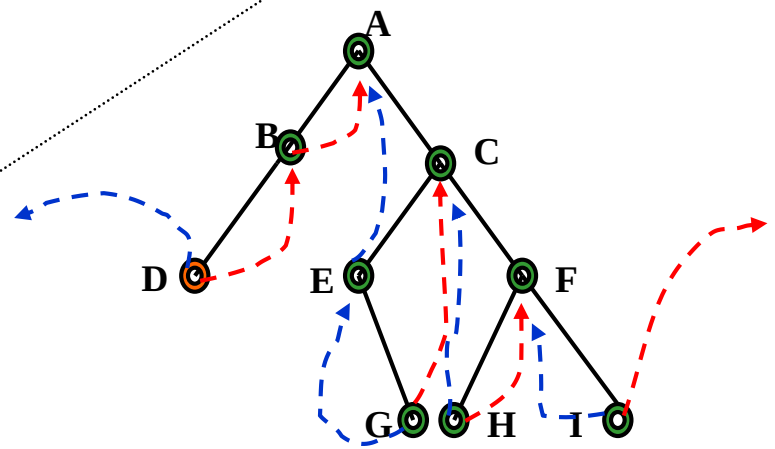
(4) 线索二叉树遍历

线索树由于线索的存在，使得二叉树的某些操作变得非常容易。例如：在某种次序遍历序列中找结点的前驱、后继不需要遍历整个二叉树，可直接根据线索或指针找到。

找后继：在中序线索二叉树中，如果结点p的右指针为线索，则直接指向后继；否则，其后继为其右子树最左下角的结点[中序遍历右子树第一个被访问的结点]。

找前驱：在中序线索二叉树中，如果结点p的左指针为线索，则直接指向前驱；否则，其前驱为其左子树最右下角的结点[中序遍历左子树最后一个被访问的结点]。





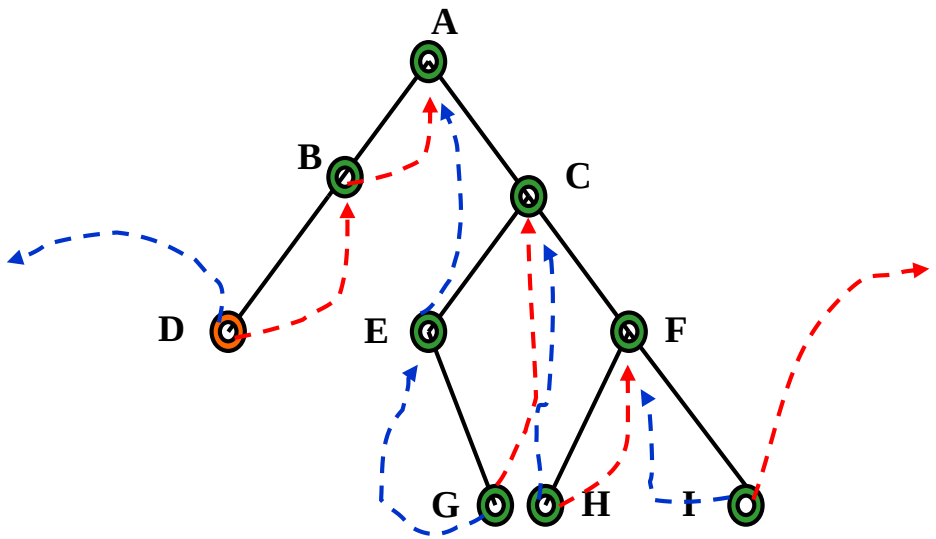
这样，线索二叉树的遍历就不需要另设栈，算法简单直接：

[a] 找第一个被访问的结点；

[b] 沿着找后继的办法，找结点的后继；

[c] 重复[b]，直到结点的后继为空为止。

找第一个被访问的结点也非常容易。对于中序线索二叉树，第一个被访问的结点是：从根结点出发沿左指针不断往下走，左指针为空的结点[二叉树最左下结点]。



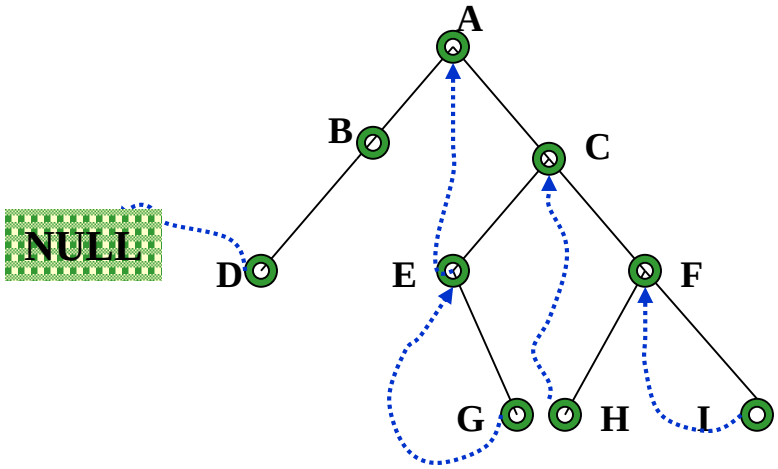
（5）单边线索二叉树

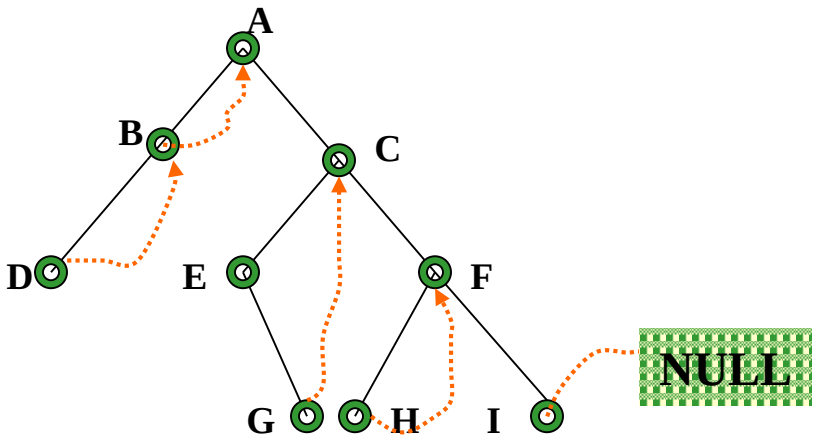
从上面的讨论可以看出：线索二叉树中既有指向前驱的左线索，又有指向后继的右线索。如果只保留一边[左或右]线索，便得到单边线索二叉树。

左线索二叉树：只将llink为空的指针修改为指向其前驱的线索，rlink不改动；

右线索二叉树：只将rlink为空的指针修改为指向其后继的线索，llink不改动；

单边线索二叉树应用也非常广泛。





5.7 树、树林与二叉树的转换

1. 树转换为二叉树
2. 二叉树转换为树

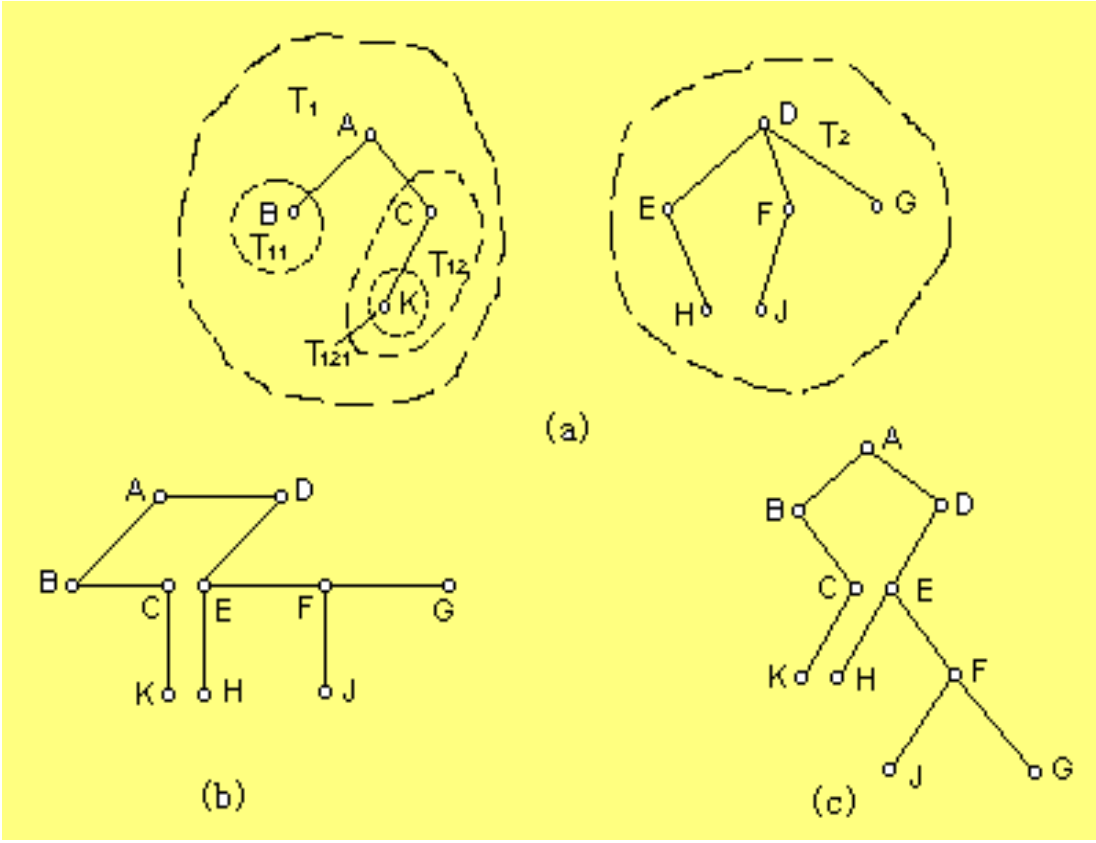
树、树林和二叉树皆可以通过二叉链表作为存储结构，因此借助二叉链表可以实现它们之间的转换。

1. 树、树林转换为二叉树

通过树、树林的孩子-兄弟表示法得到的二叉链表作为转换工具。

树对应的二叉树中，右子树一定为空。

树林对应的二叉树中，最后一棵树的右子树必为空。

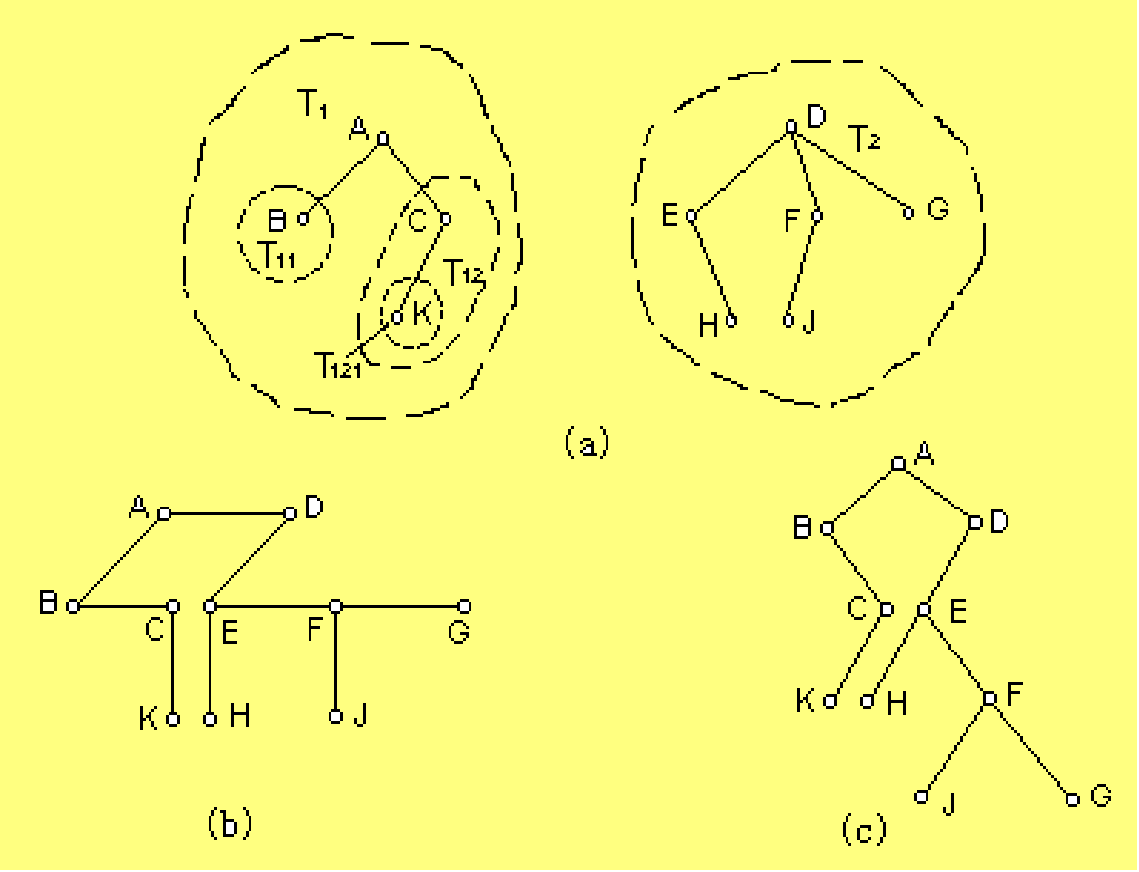


转换步骤：

① 在所有相邻的兄弟结点之间加一条线；

② 对每个非终端结点，只保留它与最左子女的连线，删除它与其它子女的连线；

③ 以根为轴心，将整颗树顺时针旋转45度，使其结构分明。

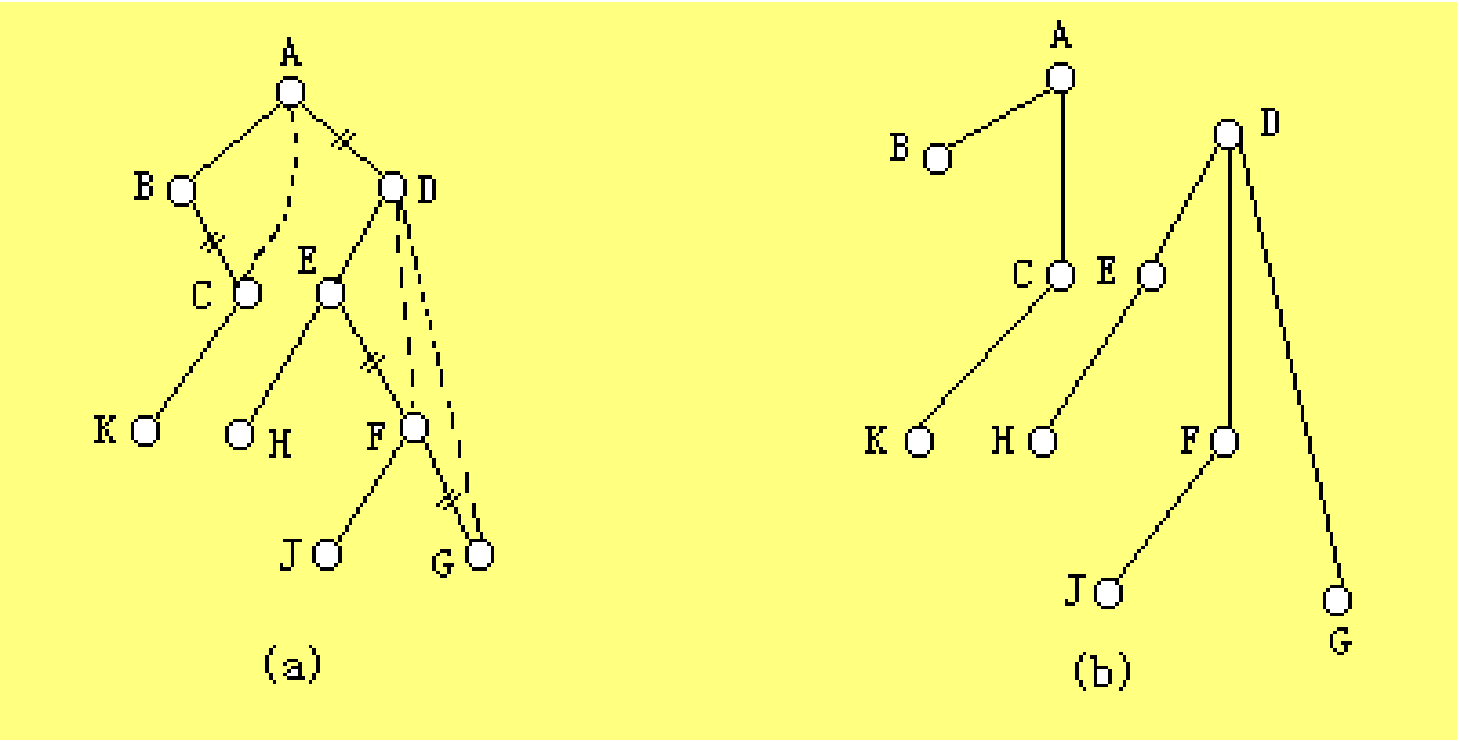


2. 二叉树转换为树、树林

① 如某结点为其父母的左子女，则把该结点的右子女，右子女的右子女，……，都与该结点的父母用虚线连接起来；【它们原来与左子女是兄弟】

② 去掉原二叉树中所有父母到右子女的连线；

③ 整理上面得到的树或树林，并将虚线改为实线。



5.8 哈夫曼树及其应用

（霍夫曼）

1. HuffMan树的定义
2. HuffMan算法构造最优二叉树
3. HuffMan编码

1. HuffMan树的定义

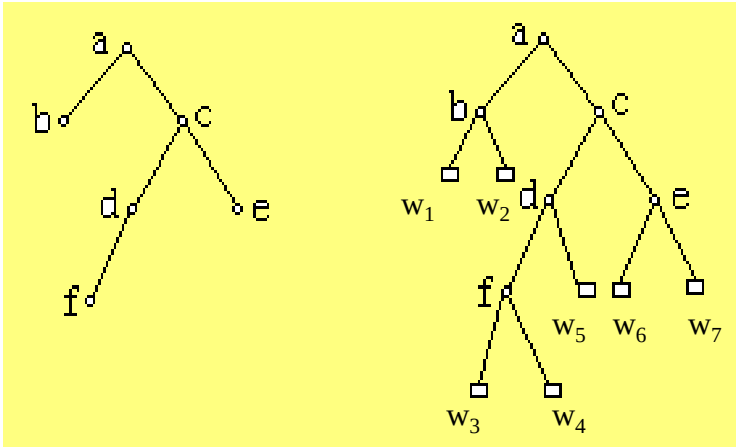
扩充二叉树外部路径长度：$E = \sum_{i=1}^{m} l_i$

其中，L_i是从根到第i个外部结点的路径长度，
m为外部结点个数

如果所有的外部结点都有一定的加权值，则外部路径长度可以推广为“带权的外部路径长度”：

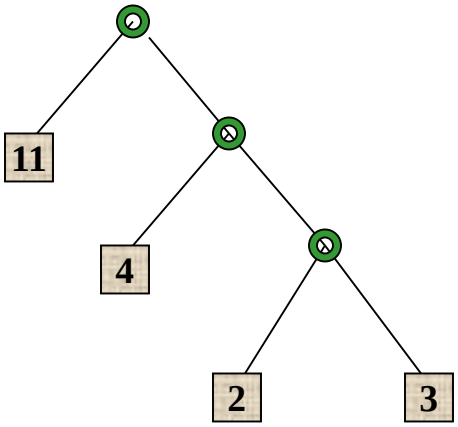
$$WPL = \sum_{i=1}^{m} w_i l_i$$

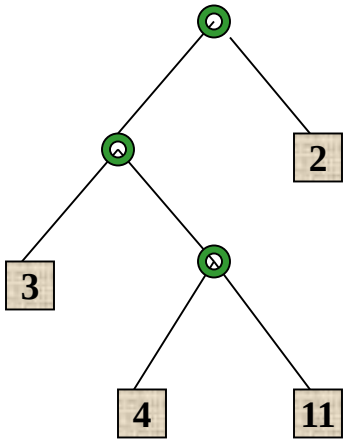
其中，w_i是第i个外部结点的加权值，
其它同上。

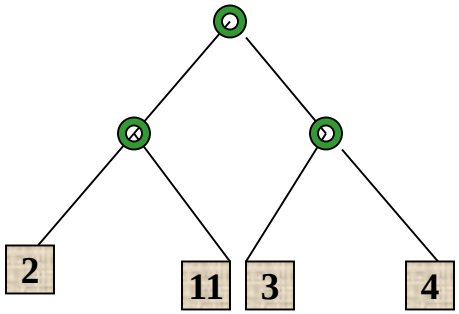


假定有一组实数{w_1, w_2, …, w_m}，现在要构造一棵有m个外部结点的扩充二叉树，外部结点的加权值序列为{w_1, w_2, …, w_m}，这样的扩充二叉树有许多，**带权外部长度WPL最小所对应的扩充二叉树称为“HuffMan树”或“最优二叉树”**。

例如：给定权值序列{2, 3, 4, 11}，可以构造出不同的扩充二叉树，并且可以计算出它们的WPL值，其中WPL最小值所对应的扩充二叉树为HuffMan树。







WPL=1x11+2x4+3x(2+3)=34　　WPL=2x3+3x(4+11)+1x2=53　　WPL=2x(2+11+3+4)=40

对于扩充二叉树，外部结点个数=内部结点个数+1。
另外，从上面图可以看出，在不同的扩充二叉树中，为了使得WPL值尽可能小，必须是：加权值较大的外部结点应尽可能地靠近根（路径长度短），这样才能使得总的WPL值降低。

给定一组加权值，有没有一个通用的方法来寻找WPL最小的HuffMan树？

2. HuffMan算法构造最优二叉树

HuffMan最早给出了一种构造最优二叉树的方法，因此称为HuffMan算法：

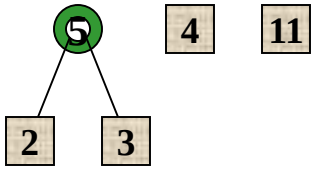
(1) 根据给定的n个权值$\{w_1,w_2,\ldots,w_n\}$，构成n棵二叉树的集合$F=\{T_1,T_2,\ldots,T_n\}$，其中每一棵二叉树T_i中只有一个带权为w_i的根结点，其左右子树为空。

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

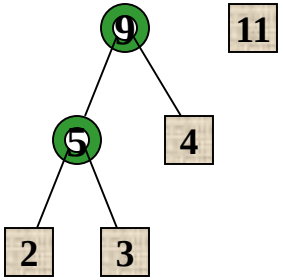
(3) 在F中删除这两棵树，同时将新得到的二叉树加入F中。

(4) 重复(2)和(3)，直到F中只含一棵树为止。

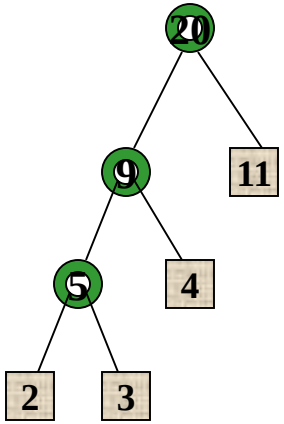
4棵只有根的二叉树



2、3合并得到3棵二叉树



5、4合并得到2棵二叉树



9、11合并得到1棵二叉树

*** 左右选择不同，以及可能存在相同的加权值，导致得到的HuffMan树形态可能不同，但WPL是相同的（最小）。

HuffMan树构造的HuffMan算法实现：

存储结构可以有多种，如二叉链表、三叉链表等。下面给出一种顺序结构(一维数组），结点定义：

ww	parent	llink	rlink

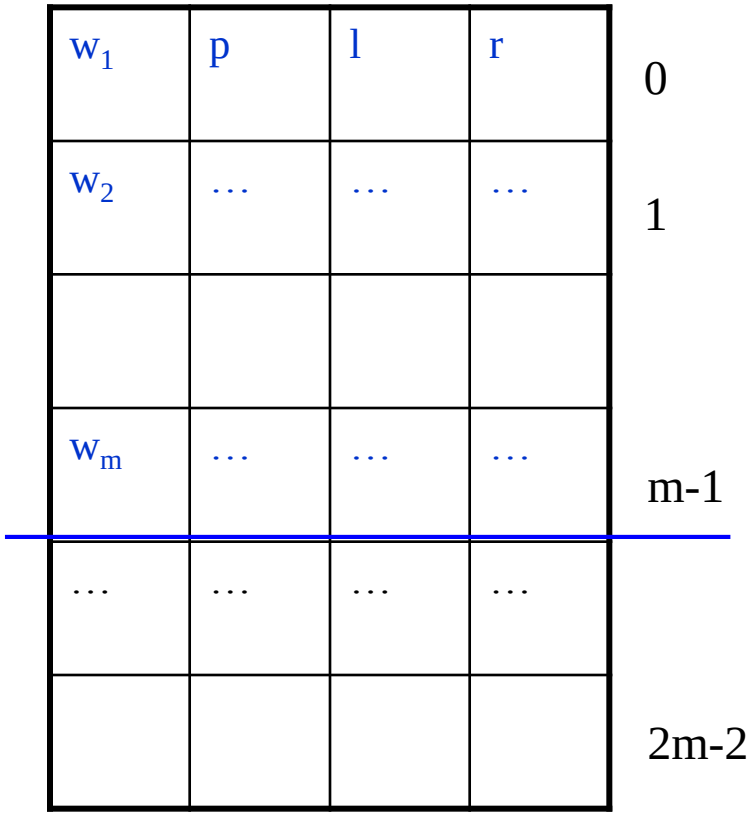
ww: 以该结点为根的子树中所有外部结点的加权和。
parent: 父结点在数组中的存储位置（下标），
根无父，设为-1。
llink: 左孩子存储位置，对于外部结点，无孩子，设为-1。
rlink: 右孩子存储位置，对于外部结点，无孩子，设为-1。

存储结构：三叉静态链表

w_1	p	l	r
w_2	…	…	…

假定外部结点个数为m，则内部结点个数必为m-1，最后得到的HuffMan树必定有2m-1个结点。因此，用2m-1个元素的一维数组就可以存储该HuffMan树。每个元素表示一个结点，前m个存储外部结点，后m-1个用于内部结点（需要构造的结点）。

```
struct HtNode
{  int ww;
   int parent, llink, rlink;
};
struct HtTree
{  struct HtNode ht[MAXNUM];
    int root;
};
tydef struct HtTree *PHtTree;
```



算法实现

3. HuffMan编码

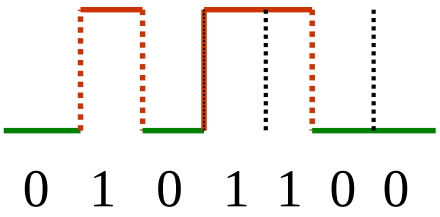
HuffMan树有许多应用，本节主要讨论它在信息编码中的应用。

（1）信息编码：

定长编码 – 所有的字符都具有相同的编码长度

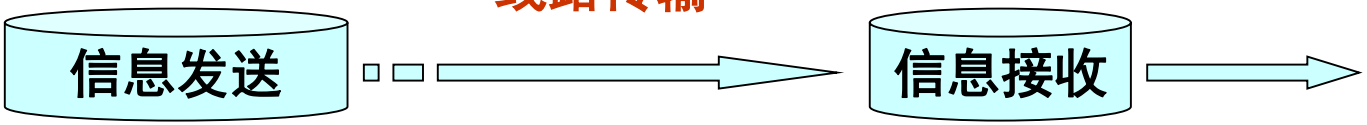
如26个字符：A, B, C, D, …, Z

A: 00000
B: 00001
C: 00010
D: 00011
……
Z: 11001



高低脉冲

线路传输



ABB…

编码：00000 00001 00001
A B B

译码：00000 00001 00001
A B B

定长编码的好处：编码简单，译码更简单，用于字符等概率出现时。

为了减少线路信息传输的长度提高信息传输速度，需要对上面的信息编码进行研究。对于各个字符出现频率不等概率情况(实际情况）下，定长编码不是最好的，如果能够使得经常出现的字符编码长度短，不经常出现的可以长一些，那么整个信息串的编码长度会减少。

不定长编码：假定字符出现频率分布为$\{p_1, p_2, \ldots, p_m\}$，根据减少信息串编码长度的要求，频率最大的字符其编码位数应该最小。假定各个字符得到的编码位数为$\{L_1, L_2, \ldots, L_m\}$，则总的信息串的长度为：

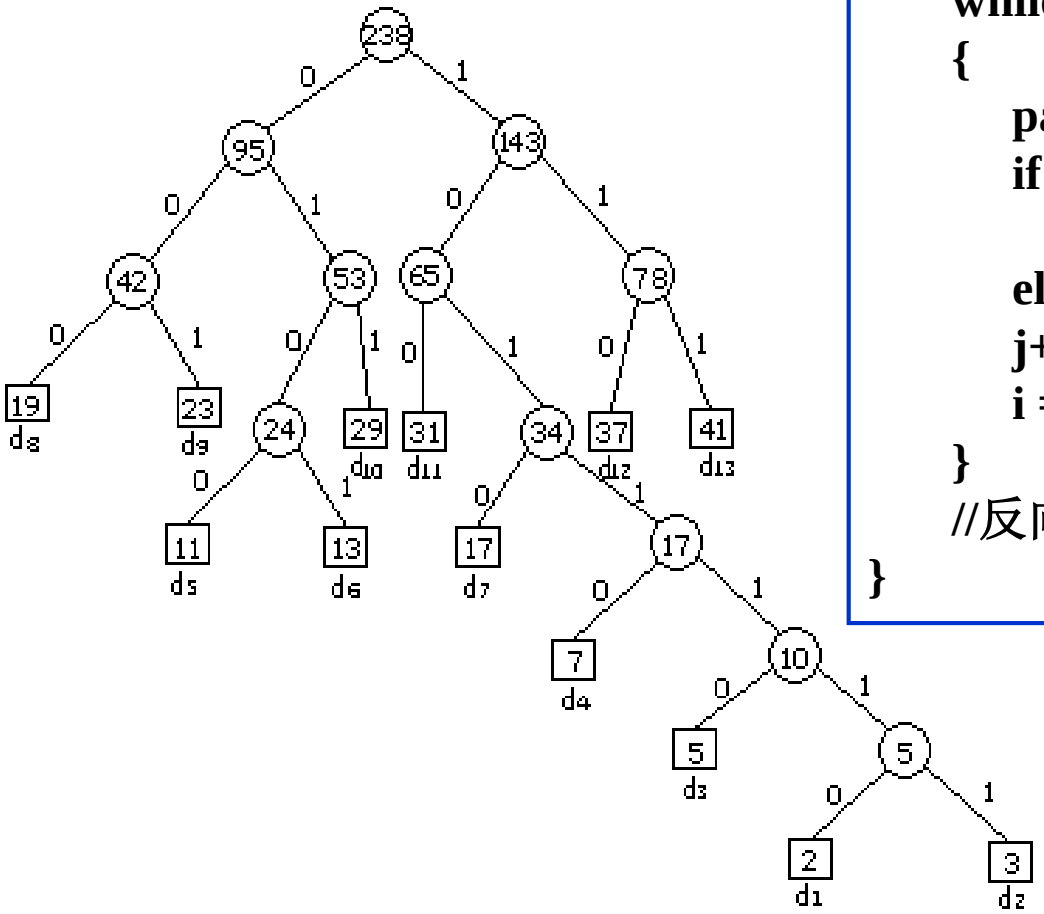
$$AL = \sum_{i=1}^{m} P_i L_i$$

可以看到，该长度与前面的WPL的定义非常相似。假定字符出现的概率为加权值，寻找AL最小的编码问题实质上就是建立一棵包含m个外部结点的HuffMan树过程。

其中L_i就是概率为P_i的字符的编码长度，既HuffMan树中外部结点的路径长度。
另外，在不等长编码中，为了译码时不出现二义问题，必须保证任何字符的编码都不是其它字符编码的前缀。
如A: 0，B: 10，C: 010，如果接收到 01010 序列，如何译码？是ABB还是CB？ 无法确定，因此必须保证不出现上述二义问题。

根据字符出现的概率，建立HuffMan树，并且约定左路径为0，右路径为 1，这样从根到任何终端结点的路径便是该字符的编码。可以证明，由此而得到的任何字符的编码都不是其它字符的前缀，避免了译码中的二义问题。

w = {2，3，5，7，11，13，
17，19，23，29，31，
37，41}




```
//已经构造好的哈夫曼存储在hfm_tree结构中
char codes[32];
for (k = 0; k < m; k++)
{
    //获取第k+1个外部结点（字符）的编码
    i = k;
    j = 0;
    while (hfm_tree.nodes[i].parent != -1)
    {
        parent = hfm_tree.nodes[i].parent;
        if (hfm_tree.nodes[parent ].llink == i)
            code[j] = 0;
        else codes[j] = 1;
        j++;
        i = parent;
    }
    //反向输出code
}
```

由外部结点（字符）开始往上找，一直到根结点（无父结点）编码：左子女0，右子女1

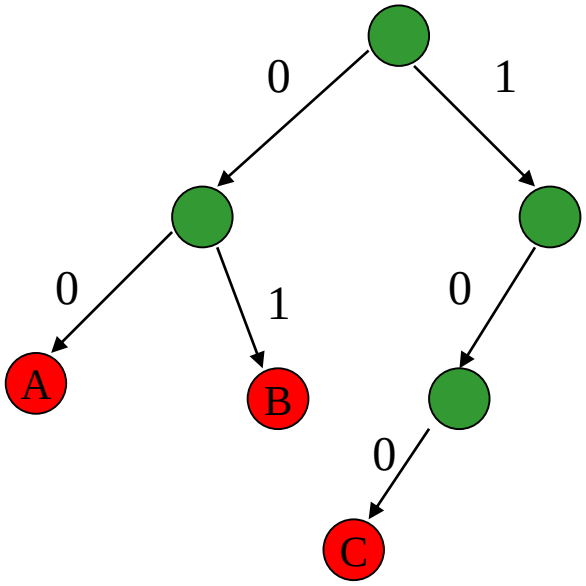
译码：HuffMan编码得到的字符串的二进制序列，如何译码？

假定三个字符的编码为　A – 00　B – 01　C – 100

二进制序列为：　010010000。

从二进制序列的第一个字符开始（HFM树的根结点），如果为0，沿左分支走，如果为1，沿右分支走，直到找到一个终端结点为止，该终端结点对应的字符为其译码；

同样，从根开始，继续二进制序列的下一个字符，…。




```
//bits数组存01码（二进制序列），hfm_tree是对应的哈夫曼树
while (k < num_bits)
{
    parent = hfm_tree.root;
    for ( ;  ;)
    {
       if (bits[k] == 0)  i = hfm_tree.nodes[parent].llink;
       else                    i = hfm_tree.nodes[parent].rlink;
       if (i < 0)
      { printf("%c", hfm_tree.nodes[parent].data);  break; }
       else       {  parent = i;    k++;  }
    }
}
```

本章小结：

两种数据结构：　树（森林）和二叉树

树：三种存储表示：父、孩子、孩子一兄弟

树的遍历（深度优先、宽度优先），

深度优先（先根、中根、后根）

二叉树：7个重要性质

存储方式：二叉链表、三叉链表、线索链表

二叉树的三种遍历方法: DLR、LDR、LRD的递归、

非递归实现，线索链表的构造和遍历

HuffMan树：　HuffMan树的构造

HuffMan编码与译码

多个算法：主要是非递归算法[遍历、线索、HuffMan树构造等]

书面作业：

p168:　8, 9,12,13

p169:　1, 4

上机作业：通讯模拟问题（HuffMan编码与译码）

要求：

1. 给定m【27个，‘a’~’z’+空格】个字符的出现频率，得到这m个字符的HuffMan编码。

2. 任意一个字符序列，得到一个二进制编码序列。
对该编码序列进行译码，得到原来的字符序列。

如：5个字符（A, B, C, D, E)的出现频率为｛20, 5, 30, 30, 15), 可能得到的编码为： 00, 010, 10, 11, 011

字符序列： ABBDEEDCC
二进制序列： 00 010 010 11 010 010 11 10 10
译码： A B B D E E D C C