

# Tree

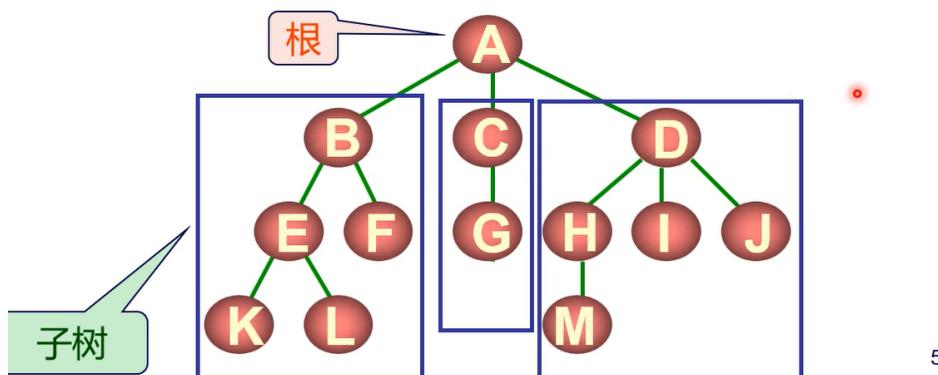
## 6.1 树的定义

- 由n个结点组成的有限集合T，如果n>0且满足
  - 有且仅有一个结点称为根(root)
  - 当n>1时，其余的结点分别为几个互不相交的有限集合，集合本身又是棵树，

### 6.1.2 树的定义

**树是由n个( $n \geq 0$ )结点组成的有限集合T，如果n>0，并且满足两个条件：**

- 1) 有且仅有一个结点称为根 (root) ，
- 2) 当n>1时，其余的结点分为几个互不相交的有限集合，每个集合本身又是棵树，被称作这个根的子树。



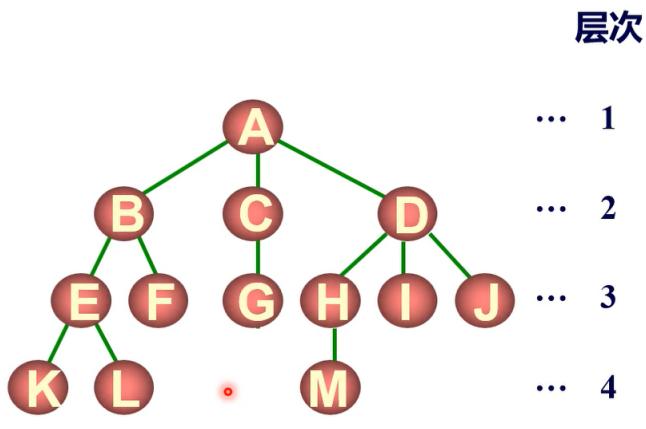
5

## 术语

- \* 结点的度：孩子个数；
- \* 叶子：度为0
- \* 双亲、孩子：E是B的孩子，B是F、E的双亲
- \* 兄弟：E、F是兄弟，F、G是堂兄
- \* 祖先：H、D、A都是M的祖先
- \* 树的度：最大的结点的度数：3
- \* 结点的层次：A为第一层
- \* 树的深度：最大的结点层数次：4
- \* 有序树：兄弟的左右次序不可互换
- \* 森林：多个树，根在同一层次

# 树的若干术语

结点的度  
叶子  
双亲与孩子  
兄弟  
祖先  
树的度  
结点的层次  
树的深度  
有序树与无序树  
森林



## 抽样数据类型定义

树的抽象数据类型定义

ADT Tree{

数据对象D: 数据元素的集合。

数据关系R: 数据元素之间关系的集合: 二元关系集

基本操作 P:

运算函数: 如求树深, 求某结点的双亲

}ADT Tree

(见教材P118-119)

## 树的逻辑结构

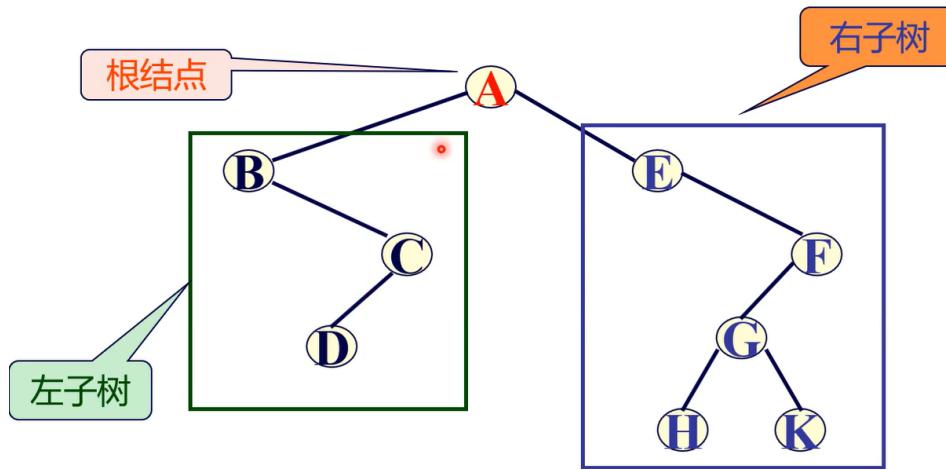
- 一对多 (1:n)
  - 多个直接后继
  - 一个直接前驱 (除根以外)

## 树的存储结构

- 顺序存储
  - 记录自己的父节点
  - 对于完全二叉树, 还行
- 链式存储
  - 两个指针
  - 三个指针

## 6.2 二叉树

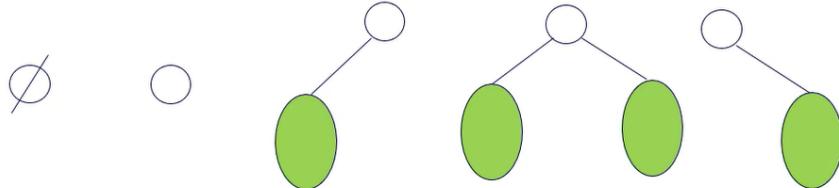
- 定义：（递归定义）
  - 二叉树 或 空树 或是 由一个 根节点 加上两颗分别称为 左子树 和 右子树 的、互不相交的二叉树



- 五种基本形态

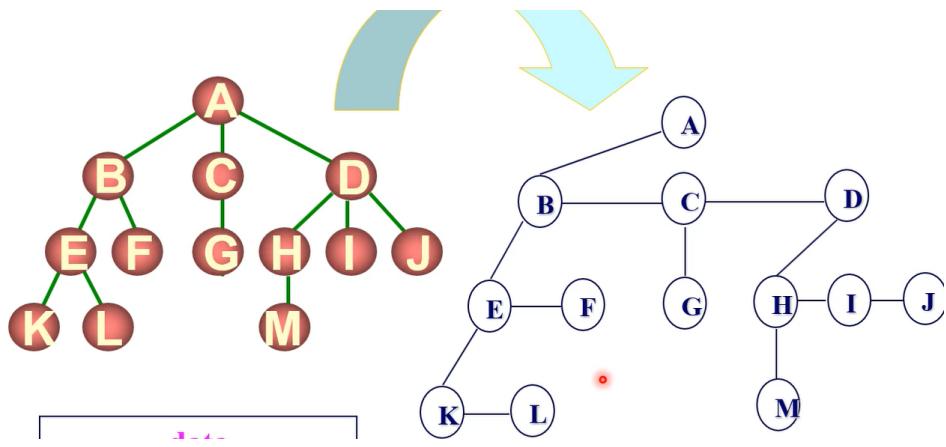
### 二叉树的基本形态

5种：



### 树转二叉树





- 方法：加线（兄弟相连） —— 抹线（长兄为父） —— 旋转（孩子靠左）
- 左：长子
- 右：下一个兄弟

## 二叉树还原为树

- 要点：逆操作：所有右孩子变为兄弟

## 二叉树的性质

- 低  $i$  层上，至多共有  $2^{(i-1)}$  个节点
- 深度为  $k$  的二叉树，至多共有  $2^{(k)} - 1$  个
- 对任意二叉树，若终端节点（叶子）为  $n_0$ ，度为 2 的节点为  $n_2$ ，则  $n_0 = n_2 + 1$ ；

证明：

节点总数  $n = n_0 + n_1 + n_2$ ; // 二叉树仅有 0 (叶子)、1、2 个入度可能性

分支总数  $e = n_1 + 2 * n_2$  // 两个入度的两个分支，一个入度的一个分支

又因为  $e = n - 1$  // 每个结点都有唯一前驱（除了根）

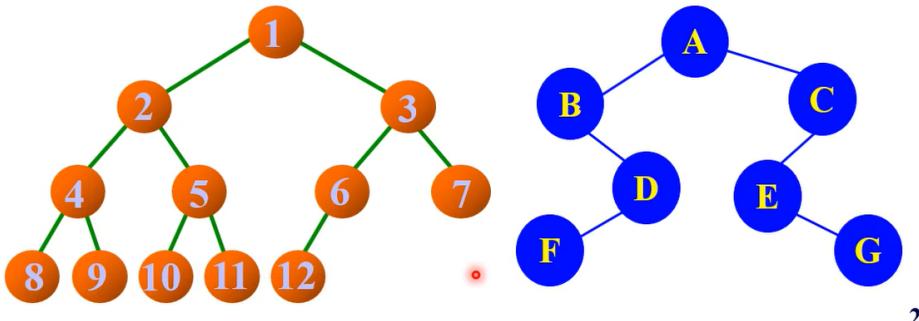
## 满二叉树

- 深度为  $k$  且有  $2^{(k)} - 1$  个结点

## 完全二叉树

- 深度为  $h$
- 除  $h$  层外，其他各层结点数达到最大值
- 且第  $h$  层从右向左连续缺若干结点（或不缺）

## 完全二叉树



## 非完全二叉树

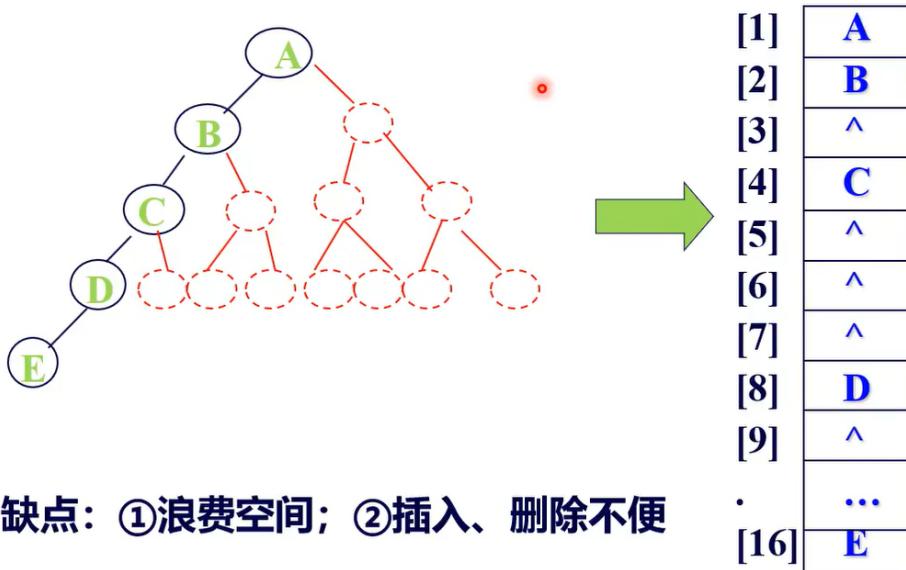
### 性质

- 度为1的结点数:  $n_1 = 1$  ( $n$ 为奇数) 或 0 ( $n$ 为偶数)
- 叶子节点数:  $n_0 = n/2$  ( $n$ 为偶数) 或  $(n+1)/2$  ( $n$ 为奇数)
- $n$ 个完全二叉树的深度:  $\text{floor}(\log_2(n))+1$  //  $\text{floor}()$ 为对float类型向下取整
- 自上到下、自左到右编号, 对于第*i*个节点 (可以顺序存储)
  - 左孩子:  $2i$
  - 右孩子:  $2i+1$
  - 双亲:  $i/2$  (向下取整)

## 不是完全二叉树怎么顺序存储?

一律转为完全二叉树!

方法很简单, 将空缺处补上“虚结点”, 其内容为空。



缺点: ①浪费空间; ②插入、删除不便

## 二叉树的存储结构

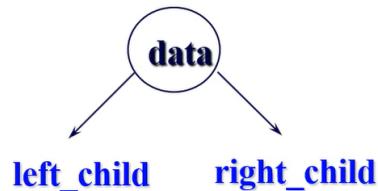
- 顺序存储: 适合于 完全二叉树
- 链式存储

## 链式存储结构

```
typedef struct node{  
    int data;  
    node *left_child, *right_child;  
}node;  
typedef node *tree_pointer;
```



二叉链表结点结构

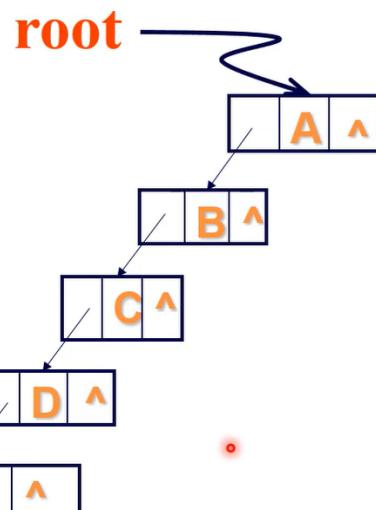
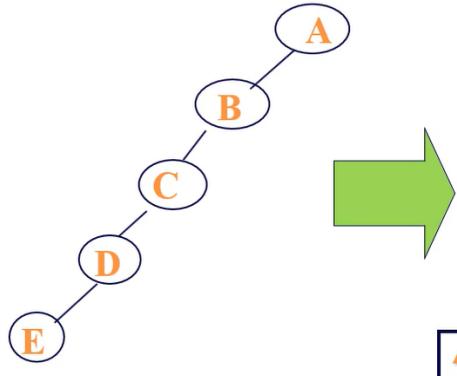


二叉树结点数据类型定义：

```
typedef struct node{  
    int data;  
    node *left_child, *right_child;  
} node;  
typedef node *tree_pointer;
```

- 二叉链表

二叉树链式存储举例：



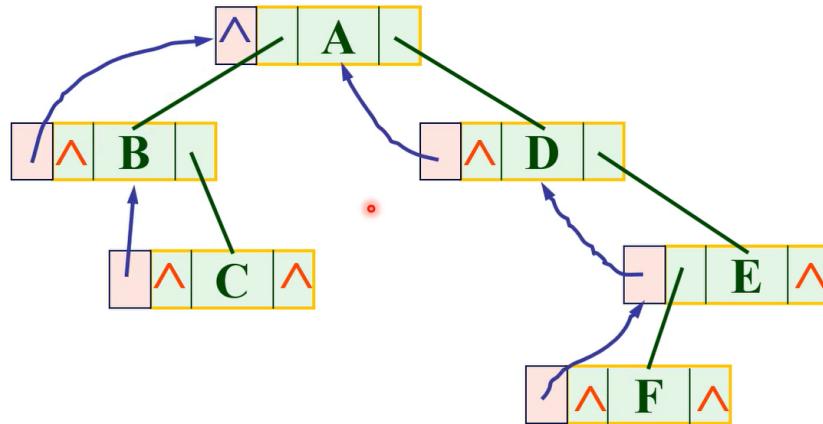
优点：①不浪费空间；②插入、删除方便

- 三叉链表

# 三叉链表

## 结点结构

parent	lchild	data	rchild
--------	--------	------	--------

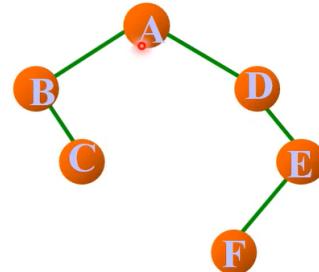


31

## 创建链式二叉树

## 创建链式二叉树

```
tree_pointer buildtree()
{ char c; node *p;
  c=getchar();
  if(c=='0') return(0); //递归出口
  p=new(node);
  p->data=c;
  p-> left_child =buildtree(); //递归
  p-> right_child =buildtree();
  return(p);
}
```



创建上面二叉树要输入什么序列?

```
tree_pointer buildtree()
{
    char c;
    node *p;
    c = getchar();
    if(c=='0') return (0);
    p = new(node);
    p->data=c;
    p->left_child = buildtree();
    p->right_child = buildtree();
    return (p);
}
```

序列:

## 6.3 遍历二叉树和线索二叉树

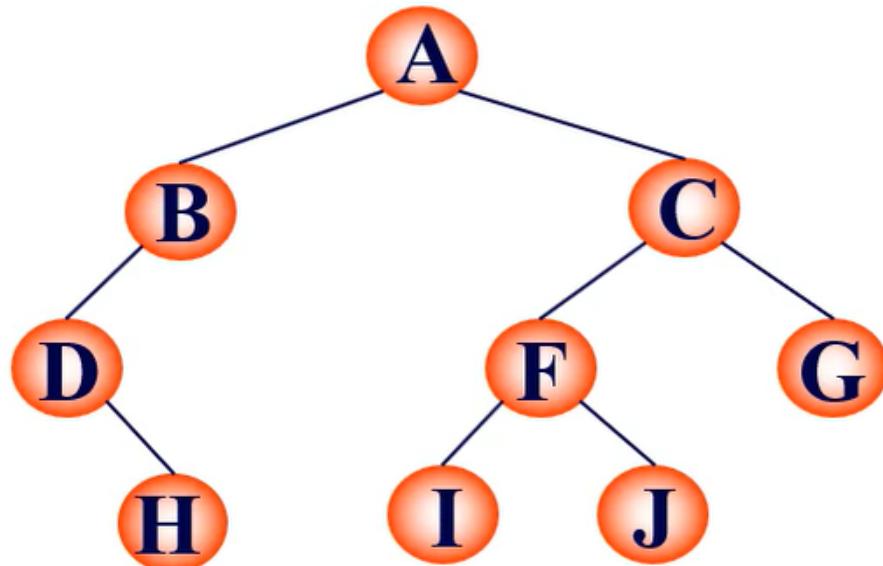
### 6.3.1 遍历二叉树

- 遍历：顺着某一条搜索路径，巡防二叉树中的结点，使每个结点均被访问一次，且仅被访问一次
- 访问：对结点左各种处理
- 遍历目的：得到树中所有结点的一个线性序列
- 两种思想：
  - 层次遍历（队列）
  - 递归算法（栈）

### 层次遍历

- 从上到下、从左到右依次访问结点
- 顺序存储上的实现：
  1. 补全为完全二叉树
  2. 按从小到大遍历、且忽略空结点
  3. 时间复杂度：稀疏二叉树，补全，导致时间浪费
- 链式存储上的实现：
  1. 队列思想
  2. 根入队
  3. 出队时，访问并将子节点从左到右入队

### 递归算法



- 先序遍历：根->左->右

```

DLR(node *root)
{
    if(root==NULL) return;
    cout << root->data;
    DLR(root->lchild);
    DLR(root->rchild);
}
cout:
A B D H C F I J G

```

- 中序遍历：左->根->右

```

LDR(node *root)
{
    if(root==NULL) return;
    LDR(root->lchild);
    cout << root->data;
    LDR(root->rchild);
}
cout:
D H B A I F J C G

```

- 后序遍历：左->右->根

```

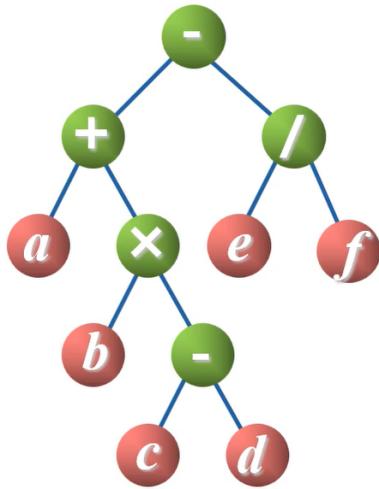
LRD(node *root)
{
    if(root==NULL) return;
    RDL(root->lchild);
    RDL(root->rchild);
    cout << root;
}
cout:
H D B I J F G C A

```

- 备注：先左后右，根的位置确定先、中、后续遍历

## 例题1

例：写出下图二叉树的先序中序和后序遍历顺序。



先序遍历: - + a x b - c d / e f // 前缀表示

中序遍历: a + b x c - d - e / f // 中缀表示

后序遍历: a b c d - x + e f / - // 后缀表示

层次遍历: - + / a x e f b - c d

## 算法分析

- 访问路径相同、时机不同：
  1. 第1次经过，就访问，先序遍历
  2. 第2次经过，访问，中序遍历
  3. 第3次经过，才访问，后序遍历

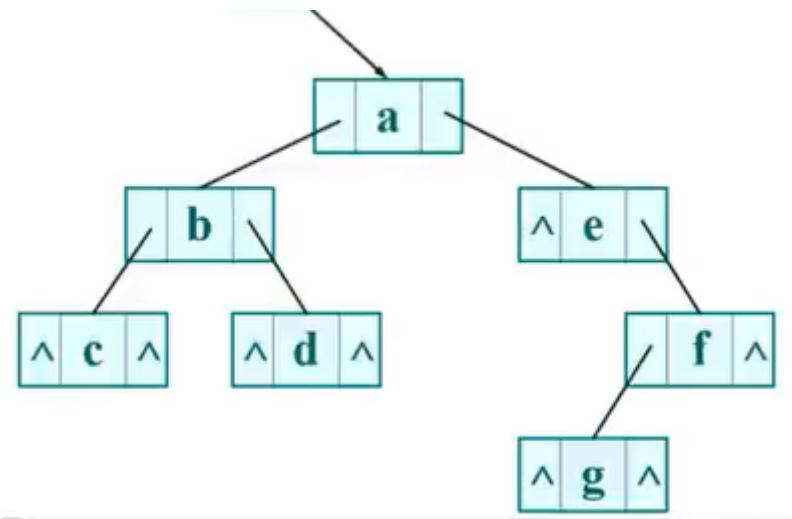
## 时间复杂度

- $O(n)$ : 每个结点只访问一次

## 空间复杂度

- 所需辅助栈空间的最大容量为  $k+1$ ,  $k$  为深度
- 最好：完全二叉树， $k = \text{floor}(\log_2(n))+1$ ,  $O(\log_2(n))$
- 最坏：单支树  $n$ , 空间复杂度  $O(n)$
- 说明：此处的最好和最坏是相对而言的。 $n$  为固定值，结点的总数。

## 例题2



先序遍历: a b c d e f g

中序遍历: c b d a e g f

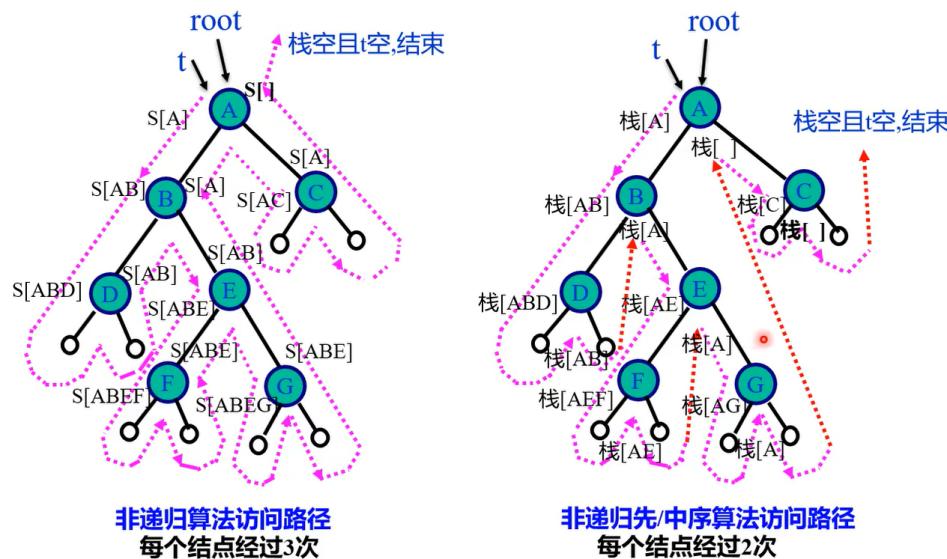
后序遍历: c d b g f e a

## 非递归算法

- 目标: 不用递归算法, 得到三种遍历序列
- 用栈: 保存地址信息
- 用栈来模拟函数递归的过程
  - 三种访问顺序中, 压栈顺序相同: 先左孩子, 后右孩子、
  - 每个结点经过3次, 哪次进行访问, 即对应哪种递归遍历
- 改进的非递归先/中序算法:
  - 仅考虑先/中序算法时: 每个节点经过2次 (第二次经过访问后, 直接pop掉当前节点, 然后push进当前节点的孩子)

## 改进的非递归先/中序算法

经过第二次时出栈!

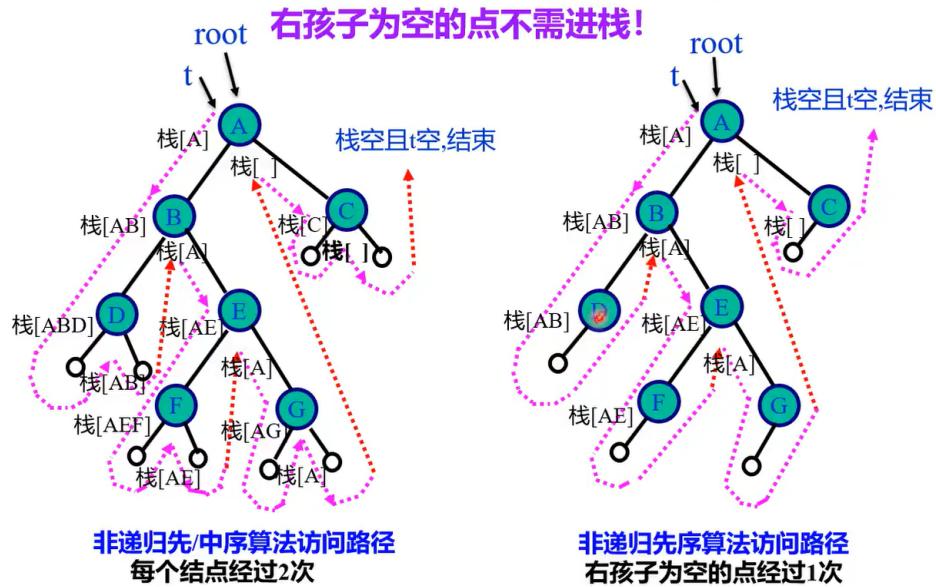


- 改进的复杂度分析:

- 时间复杂度:  $O(n)$

- 空间复杂度：
  - 最好情况：右单支： $O(1)$
  - 最坏情况：左单支： $O(n)$
- 再次改进的非递归线序算法

## 非递归先序算法的进一步改进



- 复杂度分析：
  - 时间复杂度： $O(n)$
  - 空间复杂度：最好的情况：右单支、左单支均为 $O(1)$

## 二叉树递归算法的应用举例

- 求二叉树的深度
  - 统计叶子节点个数
  - 求第k层节点的个数
- 后序遍历

### 求深度

- $\text{depth}[i] = \text{结点 } i \text{ 的左子树、右子树最大深度} + 1$

### 求叶子总数

- 空树：0
- 根是叶子：1
- 左右子树的叶子之和

根：访问的当前节点

### 求第k层结点的个数

### 3、求二叉树中第k层结点的个数

```
int KthLayerNum(BiTree root, int k)
{
    int a, b;
    if(!root) return(0);
    if (k == 1) return(1);
    a= KthLayerNum(root->lchild, k - 1);
    b= KthLayerNum(root->rchild, k - 1);
    return(a+b); . .
}
```

### 二叉树的重建

由二叉树的前序序列和中序序列可以唯一地确定一棵二叉树

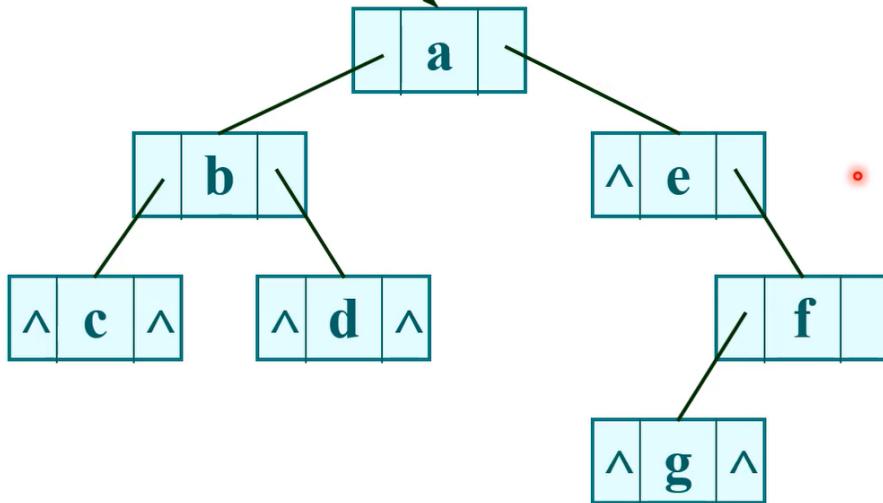
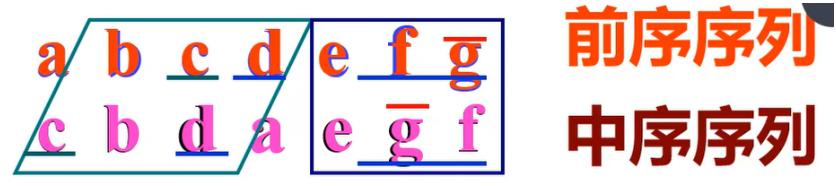
由二叉树的后序序列和中序序列可以唯一地确定一棵二叉树

由二叉树的前序序列和后序序列不可唯一地确定一棵二叉树

### 例题：前序、中序建树

前序： abcdefg

中序： cbdaegf



```

bitree BuTrPM(sqlist s1, sqlist s2)
{ int j,l1,l2,h1,h2; char c; node *p;
  l1=s1.low;l2=s2.low;h1=s1.high;h2=s2.high;
  if(l1 > h1 || l2 > h2) return(0);           //递归出口
  c=s1.ch[s1.low];      //由先序遍历序列确定根
  p=new(node); p->data=c;                      //生成根结点
  for(j=s2.low;j<=s2.high;j++)
    if(c==s2.ch[j]) break; //在中序序列找根的位置j
  s1.low=l1+1; s1.high=l1+j-l2; s2.low=l2;s2.high=j-1;
  p->lchild= BuTrPM(s1,s2);        //递归构建左子树
  s1.low=l1+j-l2+1; s1.high=h1; s2.low=j+1; s2.high=h2;
  p->rchild= BuTrPM(s1,s2);        //递归构建右子树
  return(p);
}

```

```

typedef struct {
  char ch[maxsize];
  int low,high;
}sqlist;

```

## 关于建树：找竞赛书看看

- 必须知道中序（得到“根左” - “根” - “根右”此三者的唯一的正确的顺序，然后可以进行递归建树）

## 6.3.2 线索二叉树

- 引入：

- 共n个结点时：有2n个指针，其中只用了(n-1)个（每个结点指向其直接前驱）
- 于是可以得出：空指针有：n+1个，比较多，可以用起来

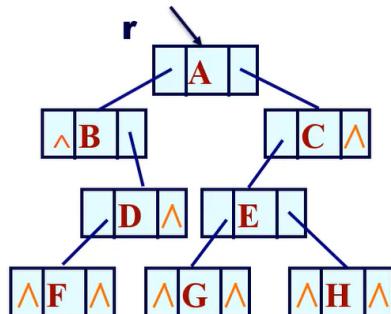
**n+1个空指针，可以用它来干什么？**

**根据不同遍历序列，存放当前结点直接前驱和后继的地址  
(线索)**

**作用：能快速（不用栈）得到原遍历序列（原遍历序列不用另外外存储），并可以按原遍历序列进行快速查找。**

**如何存放线索？**

**左孩子地址/前驱地址复用，右孩子地址/后继地址复用**

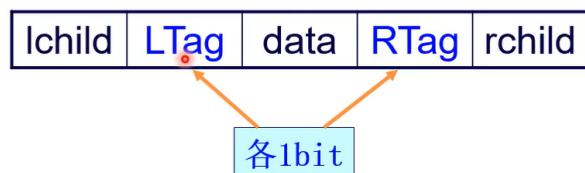


- 此处的 前驱 和 后继 指的是在 遍历顺序 中的 “前驱” 和 “后继”

## 线索二叉树的生成

- 先进行结构体的改进

**为识别两种不同信息，结点结构增加两个标志域：**



**约定：** 当 Tag 域为 0 时, 表示正常情况;  
当 Tag 域为 1 时, 表示 线索 情况.

**左(右)孩子**

**前驱(后继)**

\* 对空指针的利用：

\* 空左：指前驱

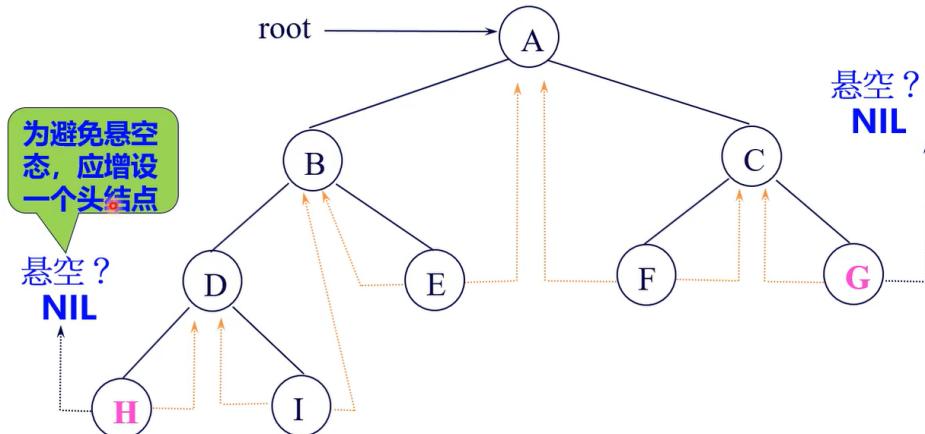
\* 空右：指后继

## 2. 线索二叉树的生成——线索化

线索化过程就是<sub>00:07:49</sub>  
过程中修改空指针的过程

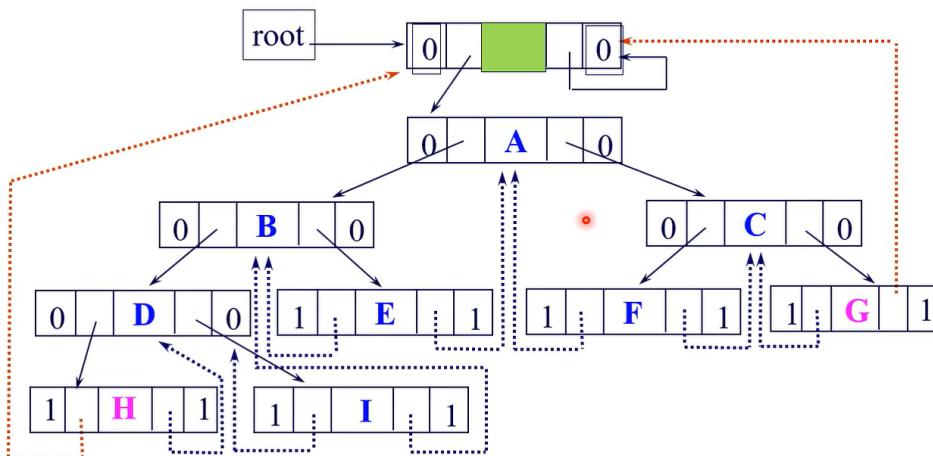
例1：画出以下二叉树对应的中序线索二叉树。

解：对该二叉树中序遍历的结果为：**H, D, I, B, E, A, F, C, G**  
所以添加线索应当按如下路径进行：



对应的中序线索二叉树存储结构如图所示：

注：此图中序遍历结果为：**H, D, I, B, E, A, F, C, G**



- 图中叶子的右指针有点小奇怪

\* 设计技巧：每次只修改前驱结点的右指针（后继）和本结点的左指针（前驱）

\* 设置两个指针：p：当前结点；pre：前趋结点

\* 遍历二叉树时，对于每个结点p：

```cpp

```
if(p->lchild==NULL) p->Ltag=1,p->lchild=pre; // p的前驱线索应在p结点的左边
if(pre->rchild==NULL) pre->Rtag=1,pre->rchild=p; //pre的后继线索在右边
```

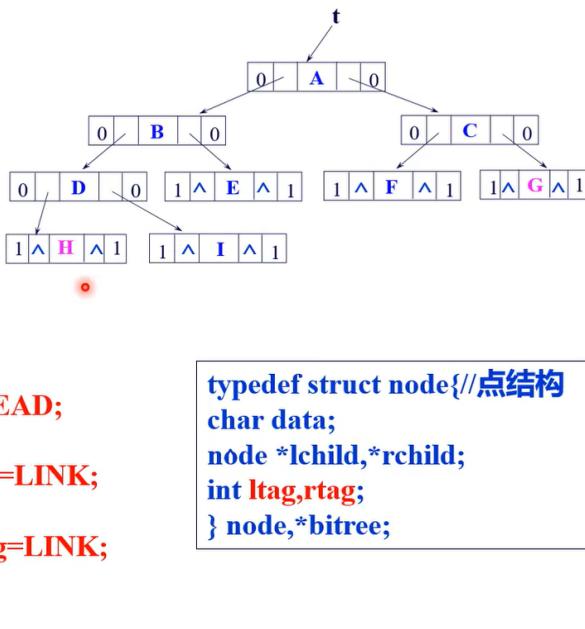
```

## 生成算法

## 线索二叉树的生成算法

分两步：

```
#define LINK 0
#define THREAD 1
bitree buildtree()
{ char c;
  node *p;
  c=getchar();
  if(c=='0')return(0);
  p=new(node);
  p->data=c;
  p->ltag=p->rtag=THREAD;
  p->lchild=buildtree();
  if(p->lchild!=0) p->ltag=LINK;
  p->rchild=buildtree();
  if(p->rchild!=0) p->rtag=LINK;
  return(p);
}
```

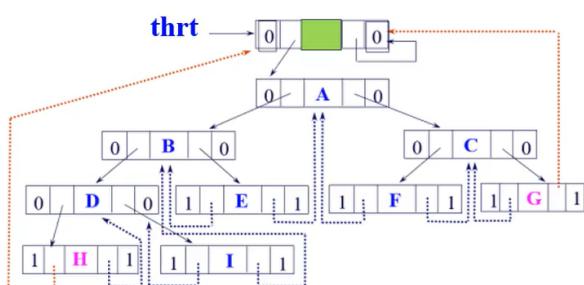


- 先建立一个普通的树，并把均设置为LINK

## 线索二叉树的递归生成算法

```
bitree pre; // pre全局变量
void buildTHtr(bitree &thrt, bitree t)
{ thrt=new(node);
  thrt->ltag=thrt->rtag=LINK;
  thrt->rchild=thrt;
  if(!t)thrt->lchild=thrt;
  else{
    thrt->lchild=t;
    pre=thrt;
    InThreading(t);
    pre->rchild=thrt;
  }
}
```

```
void InThreading(bitree p)
{
  if(!p) return;
  InThreading(p->lchild);
  if(!p->lchild) p->lchild=pre;
  if(pre->rchild) pre->rchild=p;
  pre=p;
  InThreading(p->rchild);
}
```



- 然后，先导入一个thrt作为整棵树的头头
- 再按照中序进行遍历和设置空结点

## 中序线索二叉树遍历步骤

00:28:30

有后继找后继，  
无后继找右子树  
的最左子孙

- 1) 设置一个搜索指针p;
- 2) 先寻找中序遍历之首结点 (即最左下角结点) ;
- 3) 接着进入该结点的右子树, 检查RTag 和p->rchild ;
- 4) 若该结点的RTag=1(线索), 则p->rchild指向其后  
继结点, 访问, 直到后继结点的RTag=0;
- 5) 当RTag=0时 (表示有右孩子) , 此时该结点的右  
子树左下角结点为其后继结点; 即重复2)

```
void Middle_Travel(bitree T)
{
    bitree p = T->lchild;
    while(p!=T)
    {
        // 当 有 左孩子时, 往死里找
        while(p->LTag==LINK) p=p->lchild;
        cout << p->data;
        // 当 无 右孩子时, rchild中存其后继
        while(p->RTag==THREAD && p->rchild!=T)
        {
            p = p->rchild;
            cout << p->data;
        }
        p = p->rchild;
    }
}
```

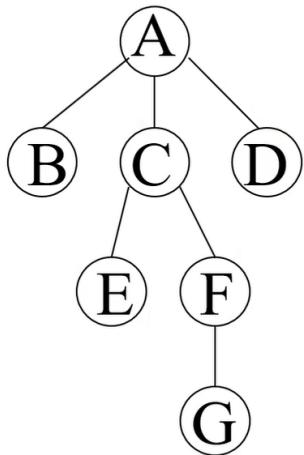
## 关于线索树的看法

- 在已知访问顺序: 前、中、后序的遍历顺序下, 通过一定顺序的遍历, 定义出他们空节点  
指向的地址
- 可以减少空间复杂度 (不用栈进行递归)

## 树、森林与二叉树的转换

### 树的表示

## 双亲表示



data parent

0	A	-1
1	B	0
2	C	0
3	D	0
4	E	2
5	F	2
6	G	5

r=0  
n=7

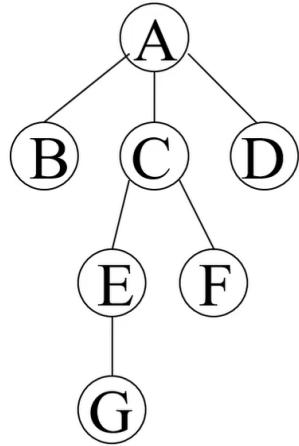
- 数据域、指针域
- 子节点 记录他们 双亲 的所在的“脚标”

```
// 节点定义
typedef struct PTNode{
    ElemtType data;
    int parent; //双亲位置域
} PTNode;

// 树定义
typedef struct{
    PTNode nodes[MAX_TREE_SIZE];
    int r,n; //r:root, 根节点; n:节点总个数
} PTree
```

## 孩子链表表示（好）

- 每个结点加一个链表，该链表的数据域是其孩子的“数组脚标”



data parent firstchild

	data	parent	firstchild	
0	A	-1		→ 1 → 2 → 3 ↴
1	B	0 ↴		
2	C	0	→ 4 → 5 ↴	
3	D	0 ↴		
4	E	2	→ 6 ↴	
5	F	2 ↴		
6	G	4 ↴		

r=0  
n=7  
.

```

// 每个结点找孩子的链表节点
typedef struct CTNode{
    int child;
    CTNode *next;
}CTNode, *ChildPtr;

// 树节点定义
typedef struct{
    Elemtyppe data;
    int parent;
    ChildPtr firstchild;
}CTBox;

//树定义
typedef struct{
    CTBox nodes[MAX_TREE_SIZE];
    int r,n;
}CTree;

```

## 树的二叉链表（孩子-兄弟）存储表示法

- 树 --> 二叉树：左孩子为节点的长子（左数第一个），右孩子为节点兄弟
- 其结点的结构和普通二叉树的区别在于其命名

```

typedef struct CSNode{
    Elemtyppe data;
    //“左孩子”装“长子”，“右孩子”装“兄弟”
    CSNode *firstChild, *nextSibling;
}CSNode, *CSTree;

```

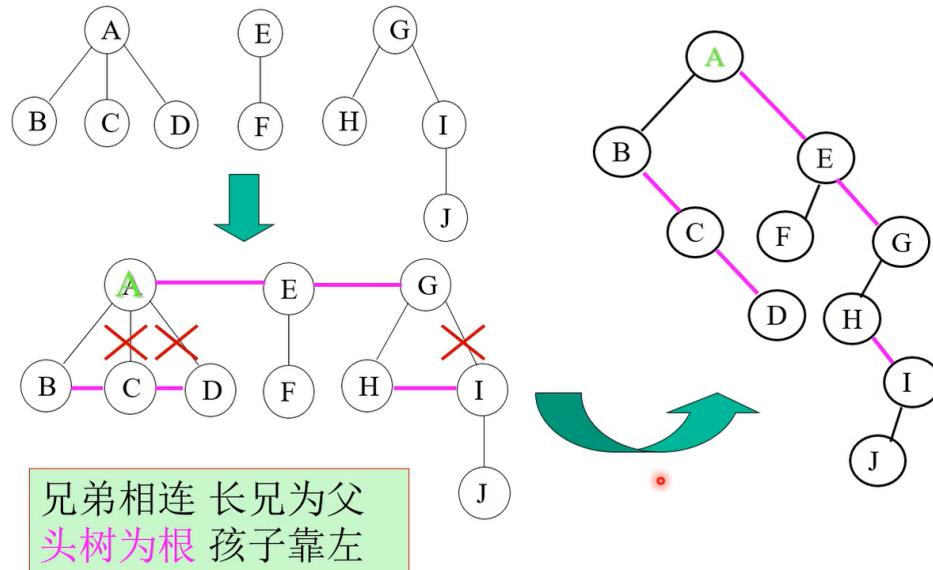
## 6.4.2 森林转换二叉树

- 回顾：树->二叉树、二叉树->树

- 森林->二叉树

- 各自转、依次连接到一个上
- 森林直变兄弟，在转二叉树

### 森林转二叉树举例： (用法二，森林直接变兄弟，再转为二叉树)

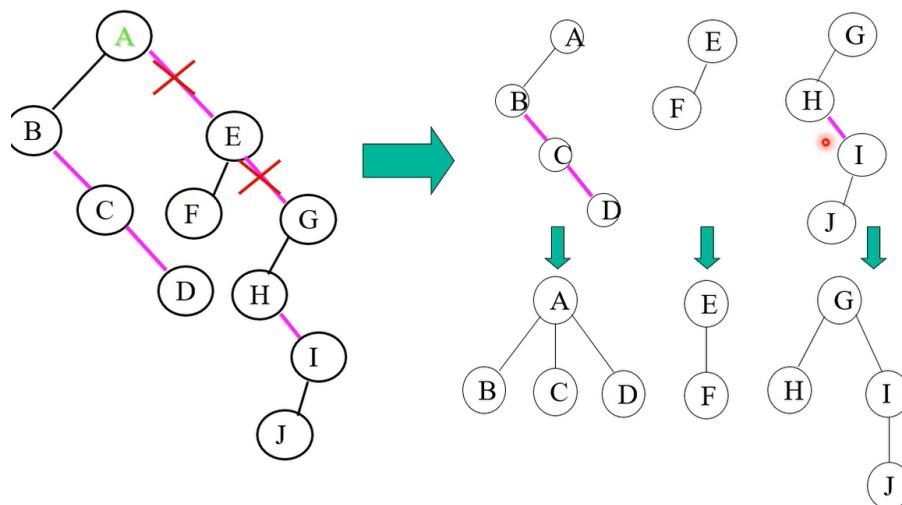


二叉树->森林

### 讨论2：二叉树如何还原为森林？

即  $B = \{\text{root}, LB, RB\} \longleftrightarrow F = \{T_1, T_2, \dots, T_m\}$

要点：把最右边的子树变为森林，其余右子树变为兄弟

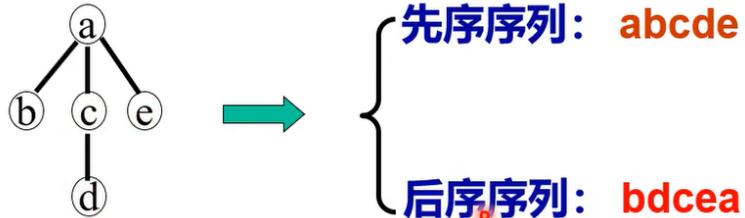


### 6.4.3 树和森林的遍历

#### 1. 树的遍历

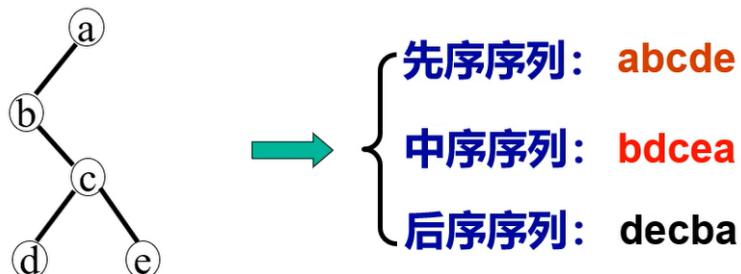
- a. 先序遍历
  - 先访问根节点；
  - 依次先序遍历根节点的每一棵树：可以理解为朴素的深度搜索
- b. 后序遍历
  - 依次后序遍历每一棵子树：最后访问根节点，每个子树亦然
  - 最后访问根节点；

例如：



- 利用二叉树进行树的遍历

**树的遍历也可以借用其对应的二叉树来实现：**



由以上例子可以看出：

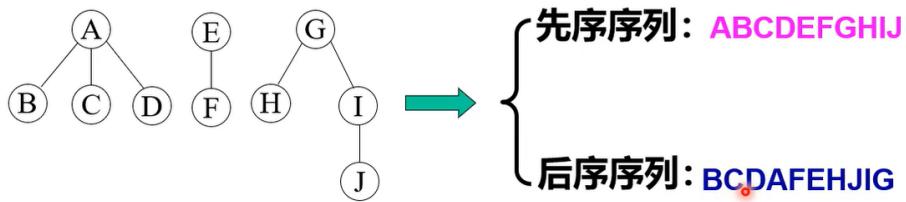
**树的先序遍历序列 = 对应二叉树的先序序列**

**树的后序遍历序列 = 对应二叉树的中序序列**

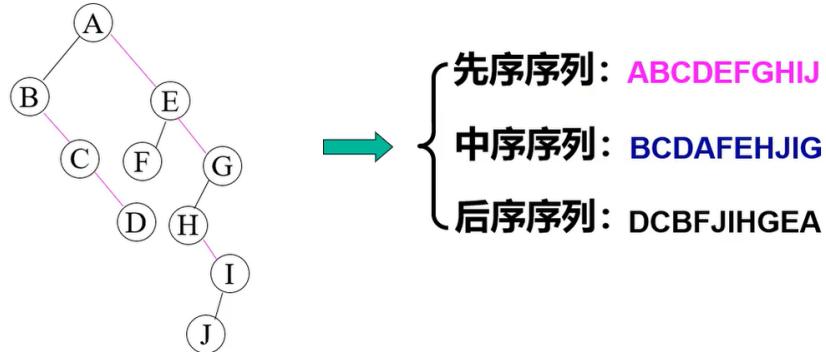
#### 2. 森林的遍历

- a. 先序遍历
  - 森林为空，返回
  - 依次对每棵树进行先序遍历
- b. 后序遍历
  - 森林为空，返回
  - 依次对每棵树进行后序遍历

例如：



森林的遍历也可以借用其对应的二叉树来实现：

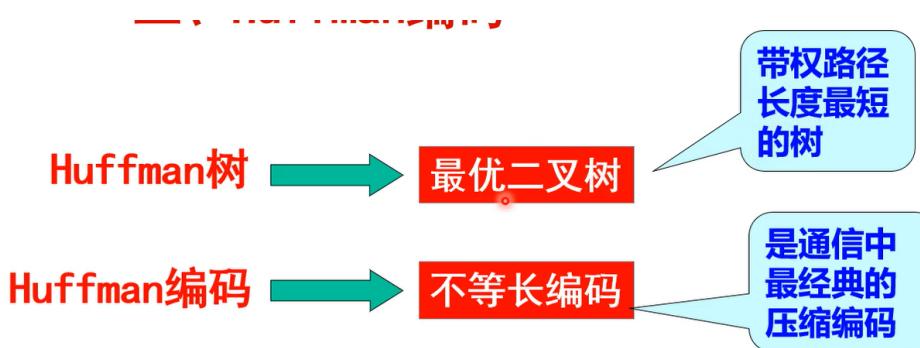


3. 结论：

- a. 树/森林 先序 遍历 = 对应二叉树的 先序 遍历
- b. 树/森林 后序 遍历 = 对应二叉树的 中序 遍历

## 6.6 哈夫曼树与哈夫曼编码

- Huffman树
- Huffman编码



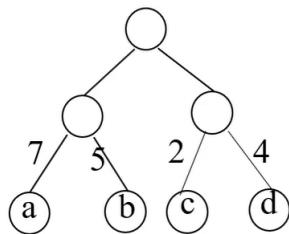
引入

树的带权路径长度如何计算？

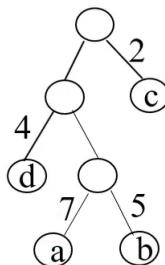
$$WPL = \sum_{k=1}^n w_k l_k$$

树中所有叶子结点的带权路径长度之和

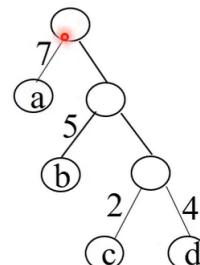
经典之例：



(a)



(b)



(c)

$WPL = 36$

$WPL = 46$

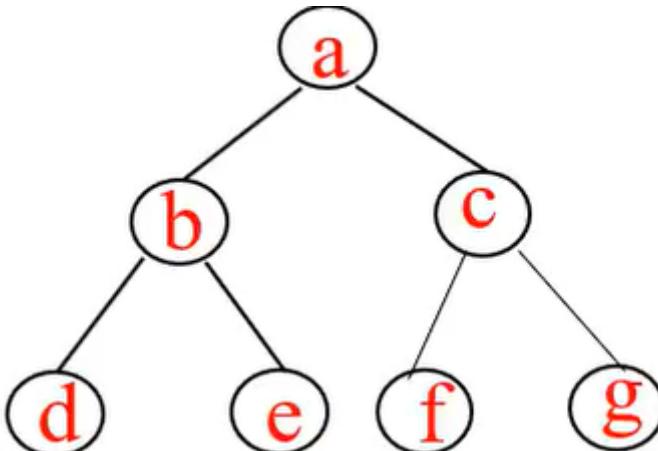
$WPL = 35$

Huffman树是  $WPL$  最小的树

\* 说明：

1.  $w_k$ : 出现概率, 图中  $w_k[a]=7$ ,  $w_k[b]=5$ ;
2.  $l_k$ : 叶子到根的距离, 图(a)中  $w_k[a]=2$ ,  $w_k[b]=2$

## 术语



- \* 路径：由一节点到零一节点的分支所构成
- \* 路径长度：路径上的分支数目，例如：从  $a \rightarrow e$  的路径长度：2,  $a \rightarrow f$  路径长度：4
- \* 树的路径长度：从 \*\*数根\*\* 到每个节点的路径长度之和
- \* 带权路径长度：\*\*结点\*\* 到 \*\*根\*\* 的路径长度乘上 结点上的权 ( $WPL$ )  
$$// \text{Weighted Path Length}$$
- \* 树的带权路径长度：树中所有叶子结点的带权路径长度之和
- \* Huffman树：带权路径长度最小的树

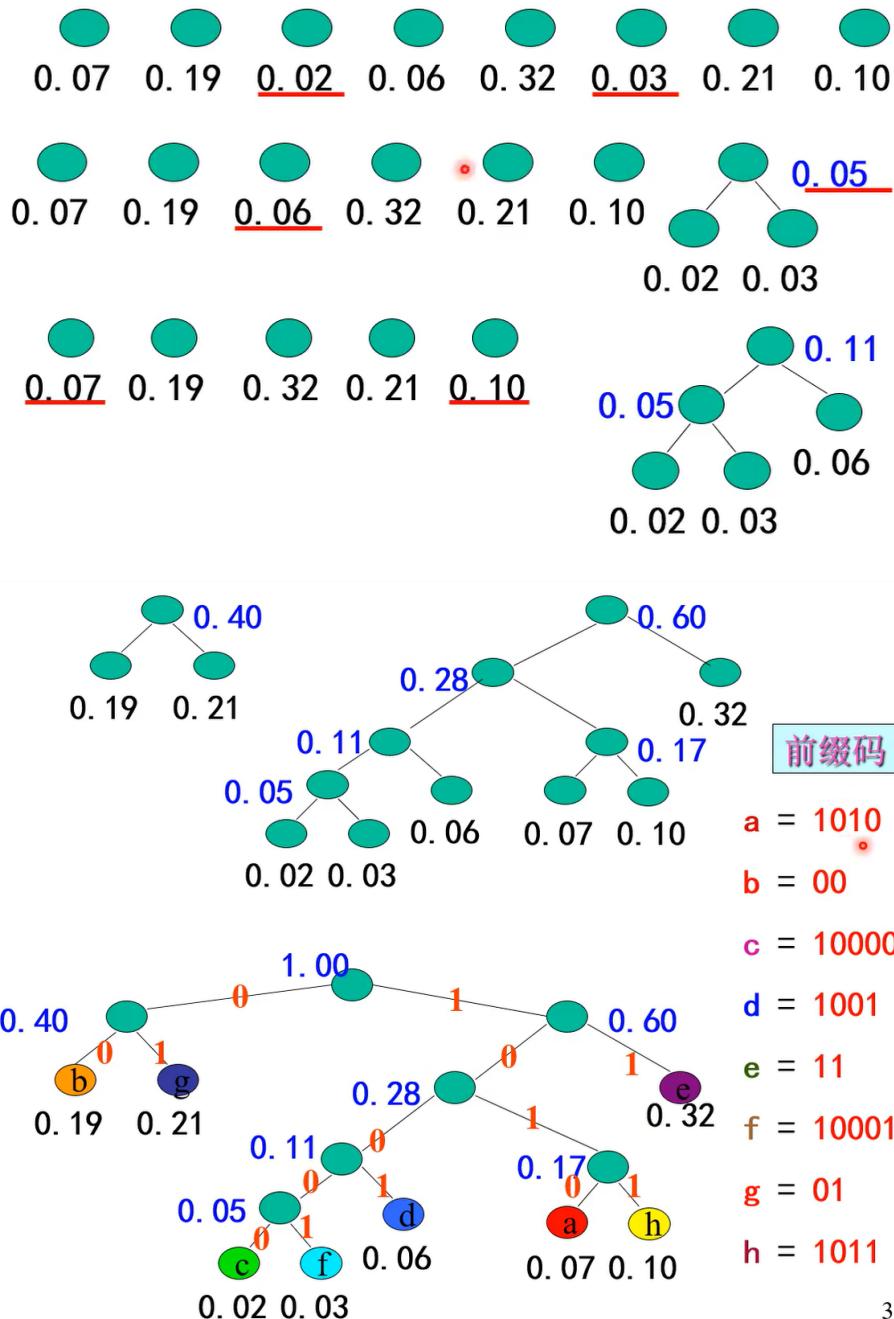
# 基本思想

- 权值大的结点，短路经；权值小的结点，长路径。

## Huffman算法

- 给定n个权值，构成n棵只有根结点的“树”的“森林”，定为集合F
  - 在F中取出两棵根节点权值最小的树作为左右子树，构建一棵新的二叉树，并将根的权值设定为这两者之和，将产生的根放入集合F中
  - 重复2的过程直到F只含一棵树为止
- 生成的树，只有入度为0和2的结点
  - 不成文的规矩：合并时，较小者放在左边

### 例题：求Huffman编码



37

- 上例中WPL = 2.61

## Huffman编码

- 可以编码压缩，广泛用于多媒体编码中

以前面通信电文为例说明什么是哈夫曼编码

- 1) 如果按等长编码对{a,b,c,d,e,f,g,h}8个字符进行编码，则需要3个比特；
- 2) 但如果按照前面这8个字符出现的概率进行编码，则需要的比特数为：

$$4*0.07 + 2*0.19 + 5*0.02 + 4*0.06 + 2*0.32 + 5*0.03 + 2*0.21 + 4*0.10 = 2.61$$

哈夫曼编码的基本思想是：概率大的字符用短码，概率小用长码。由于哈夫曼树的  $WPL$  最小这样，编码所需要的比特数最小，从而达到编码压缩的目的。它广泛应用于多媒体编码中。

## 译码

- 0: 左，1: 右
- 到叶子结点：译出字符，并返回根继续

## 唯一性

- 前缀编码
- 要求任一字符的编码都不能是另一字符编码的前缀！
- 这种编码称为前缀编码（其实是非前缀码）。

编码两个问题：

1. 数据的最小冗余
2. 译码的唯一性

## Huffman树的实现

1. 结点结构定义：

```
|weight|parent|lchild|rchild|
```

2. 存储结构：

顺序存储：结点存在 `HT[]`，对应的编码存在 `HC[]`

建立树：

- \* Huffman树没有度为1的点
- \* 一棵有 $n_0$ 个叶子结点的Huffman树共有：  
n个结点:  $n=n_0+n_2$ ,  
又因为:  $n-1 = 2 * n_2 + 0 * n_0$ ;  
所以:  $n = 2 * n_0 + 1$ ;
- \* 存储在大小为  $n = 2 * n_0 - 1$  的一维数组中

```
typedef struct{
    unsigned int weight;
    unsigned int parent, lchild, rchild;
} HTNode, * HuffmanTree; // 动态分配数组, 存储哈夫曼树
typedef char ** HuffmanCode;
```

```
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
{//w存放n个字符的权值, 构造赫夫曼树HT, 并求出n个字符的赫夫曼编码HC

    if (n<=1) return;      // n为字符数目,
    m=2*n-1;                // m为结点数目

    HT = (HuffmanTree)malloc((m+1)*sizeof(HTNode));
    //HT存放Huffman树结构, 0号单元未用, 其余前n个单元

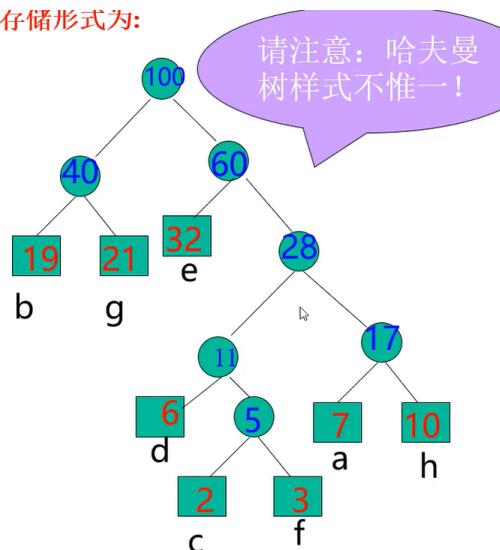
    //存放树的叶子结点, n-1个单元存放内部结点
    for (p=HT, i=1; i<=n; ++i, ++p, ++w)
    { p->weight = *w; p->parent=0;
      p->lchild=0;     p->rchild=0;   }
    // *p={*w, 0, 0, 0}; 初始化HT中的叶结点循环退出时i=n+1;

    for (; i<=m; ++i, ++p)
    { p->weight = 0;   p->parent=0;
      p->lchild=0;     p->rchild=0;   }

    // *p={0, 0, 0, 0}; 初始化HT中的内部结点
    for (i=n+1; i<=m; ++i)
        // 建赫夫曼树, (建立HT静态链表中的链)
    { Select(HT, i-1, s1, s2); //在HT[1~i-1]中选择parent
      // 为0且weight最小的两个结点, 序号分别为s1和s2
      HT[s1].parent=i;  HT[s2].parent = i;
      HT[i].lchild = s1; HT[i].rchild = s2;
      HT[i].weight = HT[s1].weight + HT[s2].weight;
    }
```

w={ 7, 19, 2, 6, 32, 3, 21, 10 }在机内存储形式为:

	w	p	l	r
1	7 ✓	C 11	0	0
2	19 ✓	C 13	0	0
3	2 ✓	0 9	0	0
4	6 ✓	0 10	0	0
5	32 ✓	0 14	0	0
6	3 ✓	0 9	0	0
7	21 ✓	C 13	0	0
8	10 ✓	C 11	0	0
9	5 ✓	0 10	0 3	0 6
10	11 ✓	C 12	0 4	0 9
11	17 ✓	C 12	0 1	0 8
12	28 ✓	C 14	0 10	0 11
13	40 ✓	C 15	0 2	0 7
14	60 ✓	C 15	0 5	0 12
15	100	0	0 13	0 14



w={ 7, 19, 2, 6, 32, 3, 21, 10 }

- 找某个叶子的编码：从子往上找，判断其是其父节点的“左”孩子，还是“右”孩子，确定这个部分是1还是0。

//从叶子到根逆向求赫夫曼编码

```

1.  HC= (HuffmanCode)malloc((n+1)*sizeof(char *));
2.  cd = (char *)malloc( n * sizeof(char) );
3.  cd[n-1] = '\0';      //编码结束符
4.  for (i=1;i<=n;++i) //逐个字符求赫夫曼编码
5.  {
6.    start = n-1; //编码结束符位置
7.    for (c=i,f=HT[c].parent; f!=0; c=f,f=HT[f].parent) //从叶子到根逆向求编码
8.      if (HT[f].lchild == c) cd[--start]='0';
9.      else cd[--start]='1';
10.     HC[i]=(char *)malloc((n-start)*sizeof(char));
11.     strcpy(HC[i],&cd[start]); //从cd复制编码串到HC
12.   }
13.   free(cd); //释放工作空间
} //HuffmanCoding
  
```

## 小结

- 2叉链表和3叉链表区别：3叉多记录了一个父节点
- 遍历
  - 森林的先序 = 对应二叉树先序
  - 森林的后续 = 对应二叉树中序

# 本章小结

