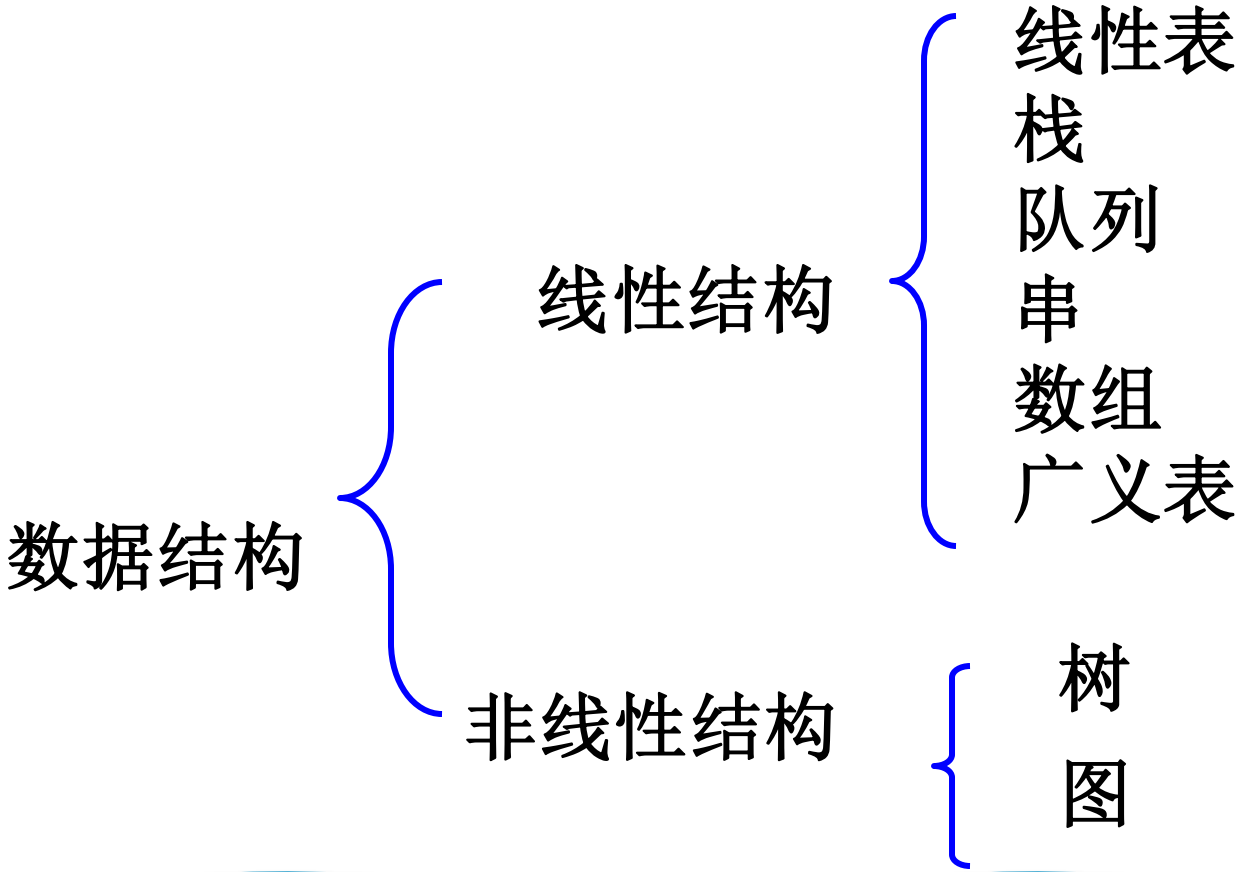
数据结构与算法

第五章 树

裴文杰

计算机科学与技术学院 教授

数据结构类型



本章是课程的重点

本章学习要点


树和二叉树
树的ADT
逻辑结构
存储结构
二叉树
逻辑结构
存储结构
基本性质
遍历二叉树
线索二叉树
树和森林
树的存储结构
树的遍历
树和森林与二叉树的转换
树的应用
Huffman树
判定过程

本章内容

5.1 树的有关概念

5.2 二叉树

5.3 二叉树的遍历

5.4 遍历的应用

5.5 线索二叉树

5.6 树和森林

5.7 树及应用

本章内容

5.1 树的有关概念

5.2 二叉树

5.3 二叉树的遍历

5.4 遍历的应用

5.5 线索二叉树

5.6 树和森林

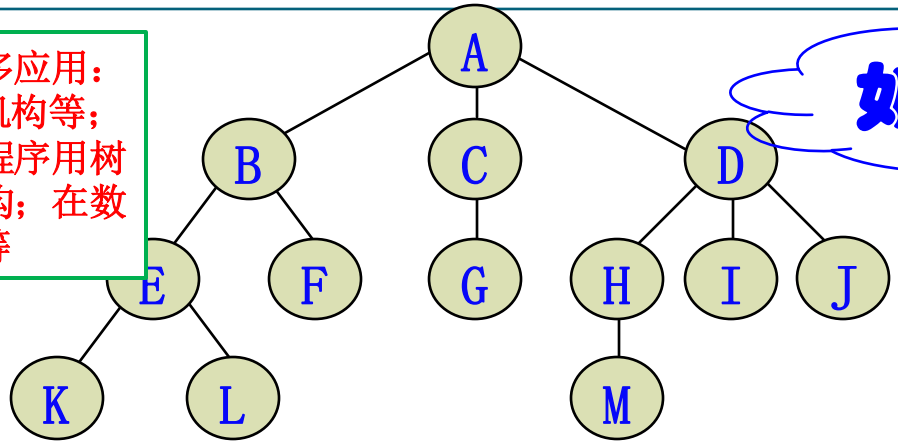
5.7 树及应用

5.1 树的有关概念

- 树的定义：

树形结构是一种重要的非线性结构。是以分支关系定义的层次结构，树是n个结点的有限集合，在任一棵非空树中：
（1）有且仅有一个称为根的结点。
（2）其余结点可分为m个互不相交的集合，而且这些集合中的每一集合都本身又是一棵树，称为根的子树。

树在现实世界中有很多应用：例如族谱，社会组织机构等；在计算机领域：编译程序用树表示源程序的语法结构；在数据库中信息组织结构等



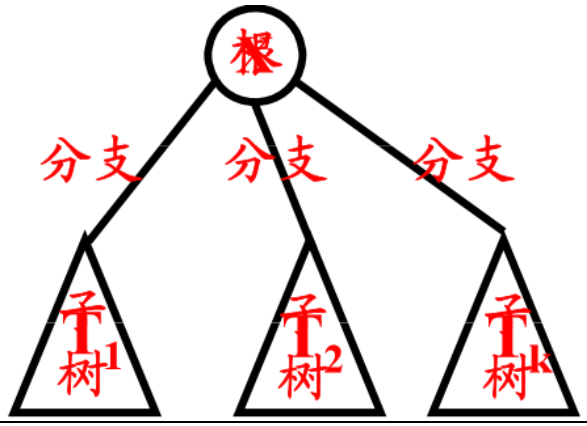
如何定义树？

说明：树是递归结构，在树的定义中又用到了树的概念

离散数学中树的定义：联通无回路的图

5.1 树的有关概念

- 树的构造性递归定义：
  - 一个结点X 组成的集合{ X }是一棵树，这个结点X 称为这棵树的根（root）。
  - 假设X 是一个结点，T_1，T_2，…，T_k是k棵互不相交的树，可以构造一棵新树：令X 为根，并有k 条边由X 指向树T_1，T_2，…，T_k。这些边也叫做分支，T_1，T_2，…，T_k称作根为X的树之子树（SubTree）。



说明：

✓ 递归定义，但不会产生循环定义；

✓ 构造性定义便于树型结构的建立；

✓ 一株树的每个结点都是这株树的某株子树的根。

5.1 树的有关概念

- 树的概念：

ADT Tree{

数据对象 D：**D是具有相同特性的数据元素的集合。**

数据关系 R：

若D为空集，则称为空树 。

否则：

(1) 在D中存在唯一的称为根的数据元素root；

(2) 当n>1时，其余结点可分为m(m>0)个互不相交的有限集T_1，T_2，…，T_m，其中每一棵子树本身又是一棵符合本定义的树，称为根root的子树。

5.1 树的有关概念

- 树的概念：

ADT Tree{

数据对象 D：

数据关系 R：

基本操作 P：

查 找 类

插 入 类

删 除 类

5.1 树的有关概念

- 树的概念：

查找类：

Root(T) // 求树的根结点

Value(T, cur_e) // 求当前结点的元素值

Parent(T, cur_e) // 求当前结点的双亲结点(就是父节点)

LeftChild(T, cur_e) // 求当前结点的最左孩子

RightSibling(T, cur_e) // 求当前结点的右兄弟

TreeEmpty(T) // 判定树是否为空树

TreeDepth(T) // 求树的深度

TraverseTree(T, Visit()) // 遍历，按某种次序对T的每个结点调用函数Visit()一次且至多一次

5.1 树的有关概念

- 树的概念：

插入类：

```
InitTree(&T)    // 初始化置空树

CreateTree(&T, definition)    // 按定义构造树

Assign(T, cur_e, value)    // 给结点cur_e赋值

InsertChild(&T, &p, i, c)
// p指向T中某结点，非空树c与T不相交，将以c为根的树插入为结点p的第i棵子树
```

5.1 树的有关概念

- 树的概念：

删除类：

```
ClearTree(&T)      // 将树清空

DestroyTree(&T)    // 销毁树的结构

DeleteChild(&T, &p, i)
                   // 删除结点p的第i棵子树
```

5.1 树的有关概念

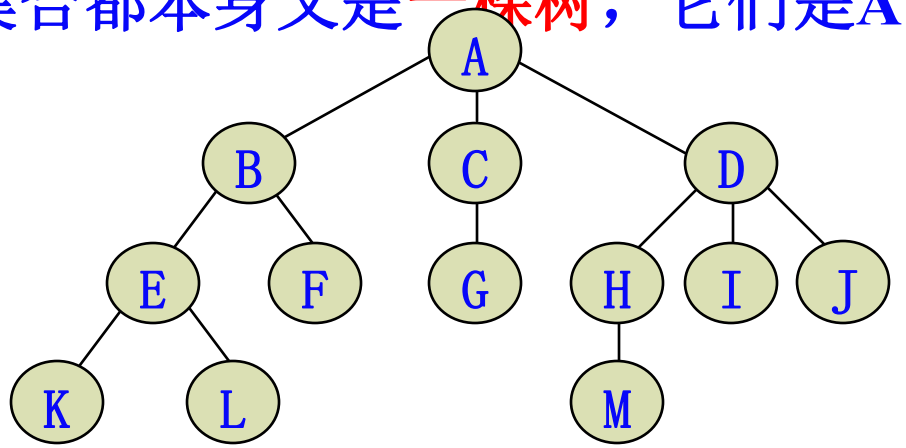
- 树的概念：

例如：集合 T={A, B, C, D, E, F, G, H, I, J, K, L, M}

A是根，其余结点可以划分为3个互不相交的集合：

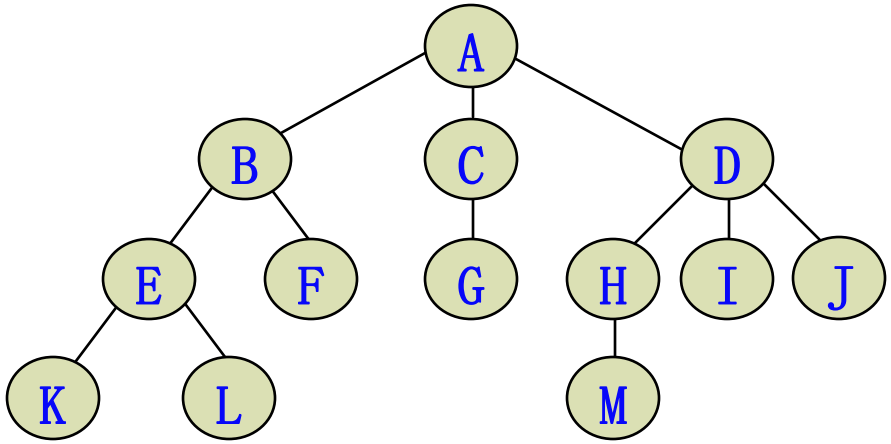
T1={B, E, F, K, L} , T2={C, G} , T3={D, H, I, J, M}

这些集合中的每一集合都本身又是一棵树，它们是A的子树。



5.1 树的有关概念

- 树的逻辑结构特点：
  - 树是一种分枝结构，树中只有根结点没有前驱；其余结点有零个或多个后继；
  - 除根外，其余结点都有且仅一个前驱；都存在唯一一条从根到该结点的路径。



5.1 树的有关概念

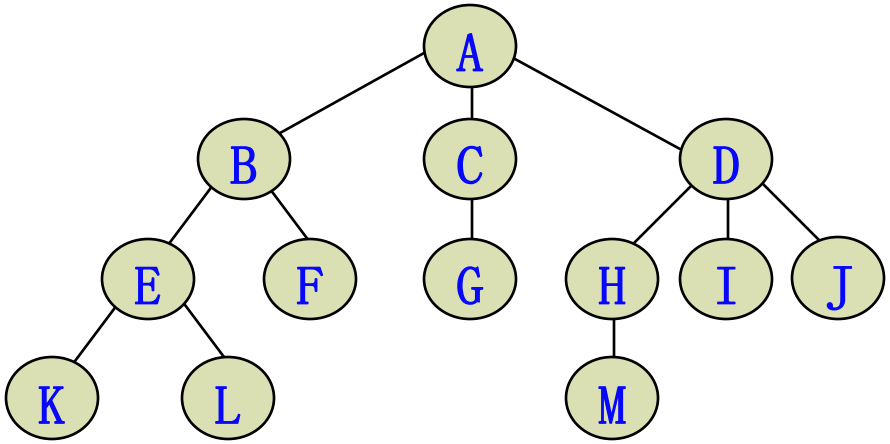
线性结构	非线形结构——树
第一个数据元素 （无前驱）	根结点 （无前驱）
最后一个数据元素 （无后继）	多个叶子结点 （无后继）
其它数据元素 （一个前驱、一个后继）	其它数据元素 （一个前驱、多个后继）

5.1 树的有关概念

- 树的应用：

  1）树可表示具有分枝结构关系的对象

例 单位行政机构的组织关系



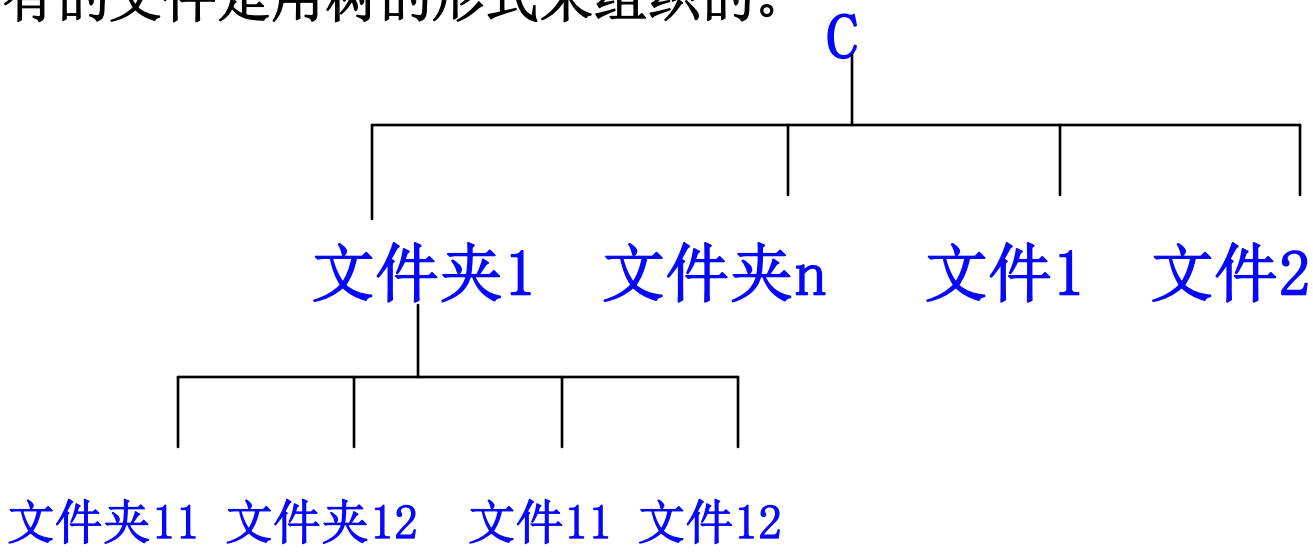
5.1 树的有关概念

- 树的应用：

  2）树是常用的数据组织形式

  - 有些应用中数据元素之间并不存在分支结构关系，但是为了便于管理和使用数据，将它们用树的形式来组织。

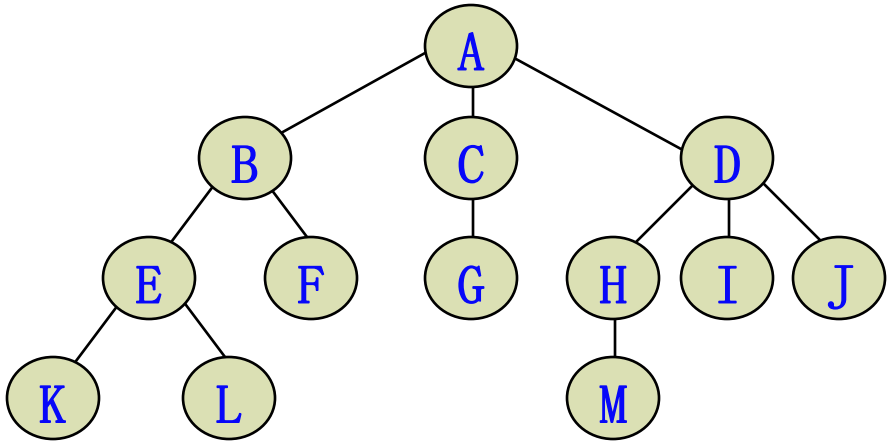
例 计算机的文件系统： 不论是DOS文件系统还是window文件系统，所有的文件是用树的形式来组织的。



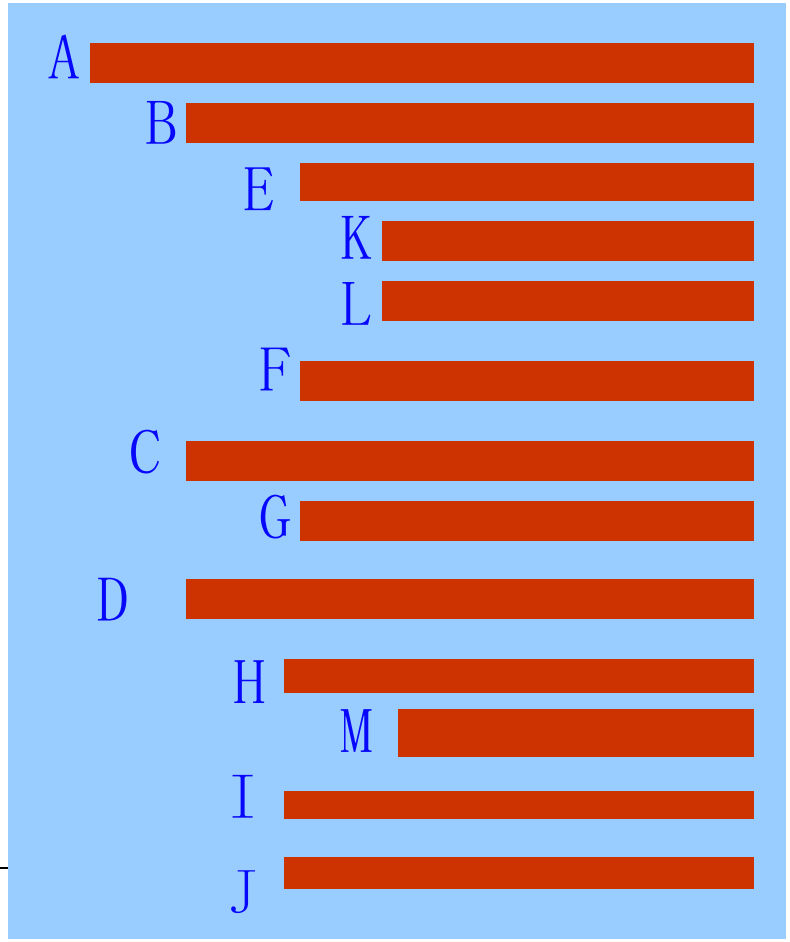
5.1 树的有关概念

- 树的表示：

（1）树形表示法



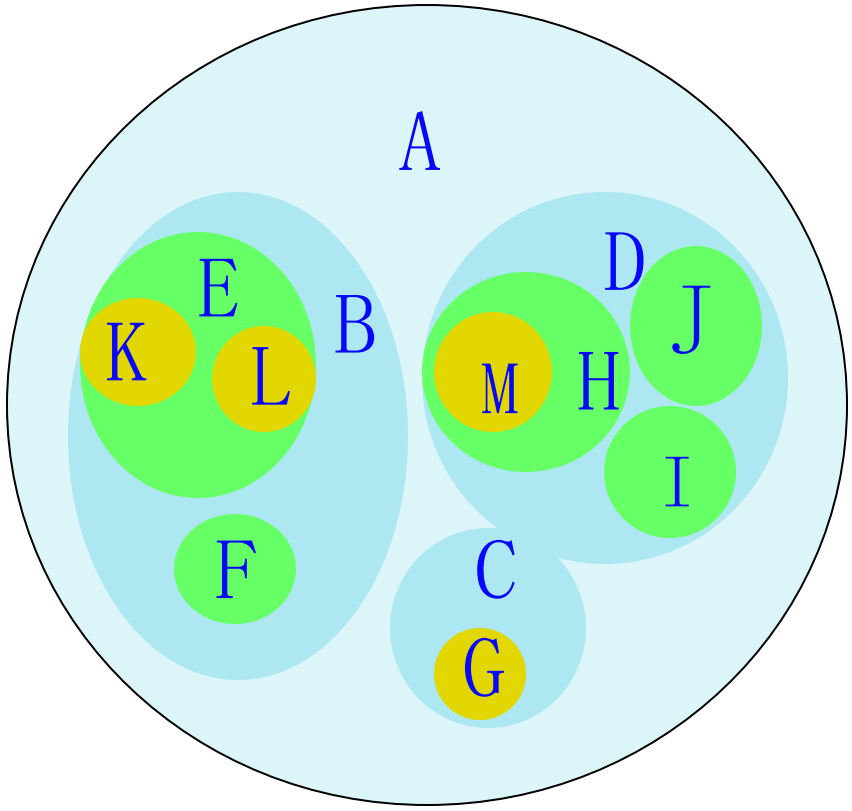
（2）凹入表示法

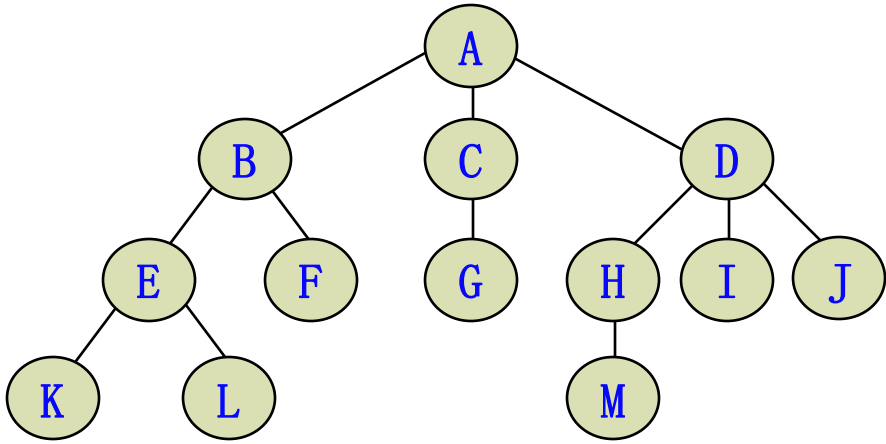


5.1 树的有关概念

- 树的表示：

（3）嵌套集合表示法 （文氏图）



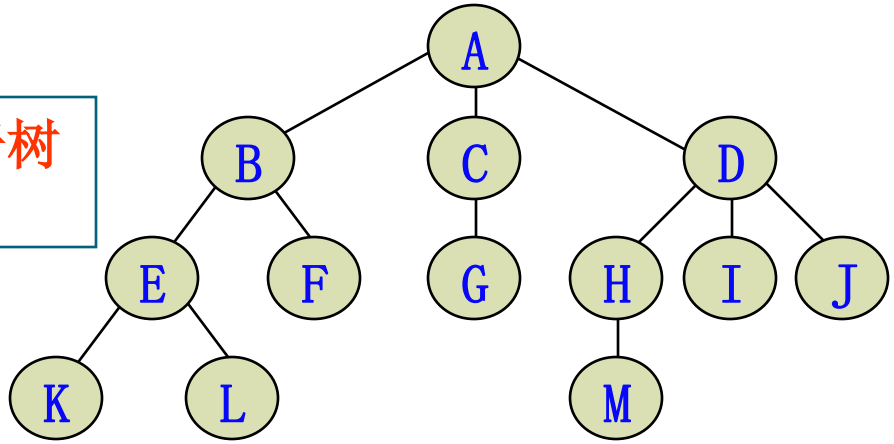


5.1 树的有关概念

- 树的表示：

（4）广义表表示法

每棵子树的根放在左边，括号内为其子树森林构成的表



(A) 第一层

(A(B(E,F), C(G), D(H,I,J))) 第三层

(A(B(E(K,L),F), C(G), D(H(M),I,J))) 第四层

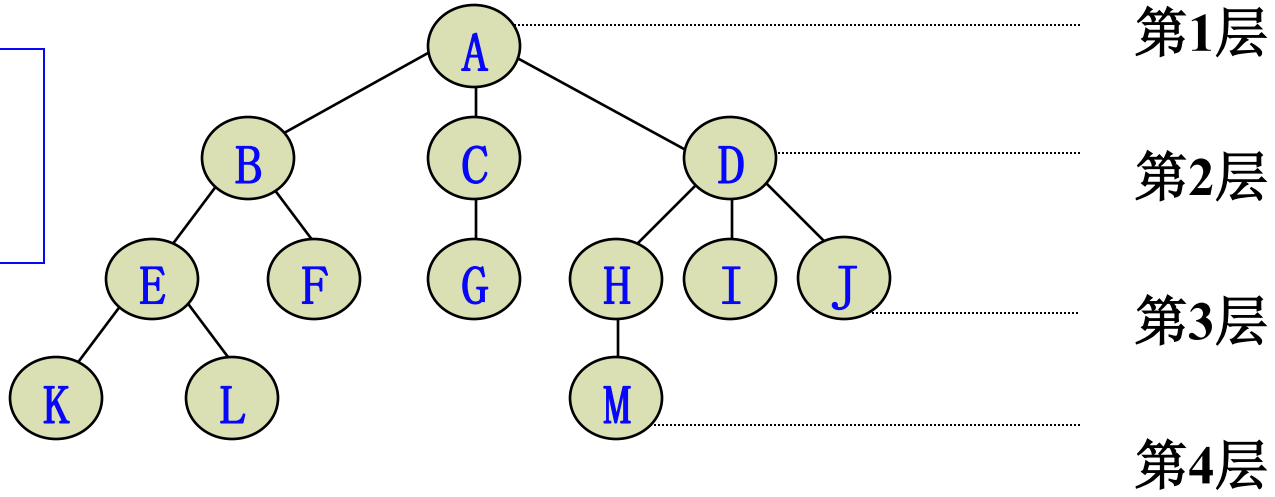
表示方法的多样化说明树结构应用的重要性

5.1 树的有关概念

- 树的有关术语：
  - 结点层：根结点的层定义为1，其它依此类推；
  - 树的深度：树中最大的结点层；
  - 结点的度：结点子树的个数；
  - 树的度：树中最大的结点度；
  - 叶子结点：也叫终端结点，是度为 0 的结点；

树的度为3

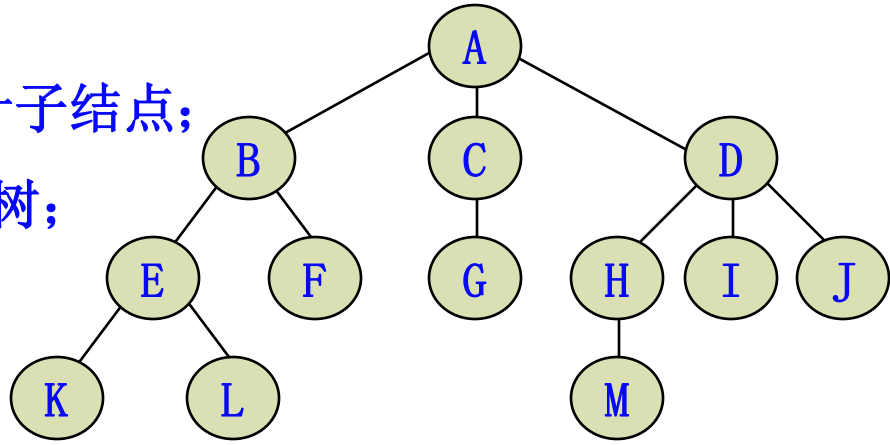
树的深度为4



5.1 树的有关概念

- 树的有关术语：

  - 分枝结点：度不为0的结点，即非叶子结点；
  - 有序树：子树有序的树，如：家族树；
  - 无序树：不考虑子树的顺序；
  - 森林：互不相交的树集合。

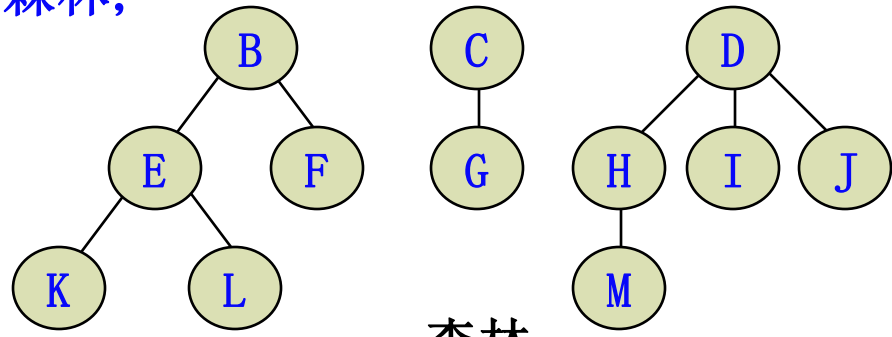


树

- 树和森林的关系：

（1）一棵树去掉根 ，其子树构成一个森林；

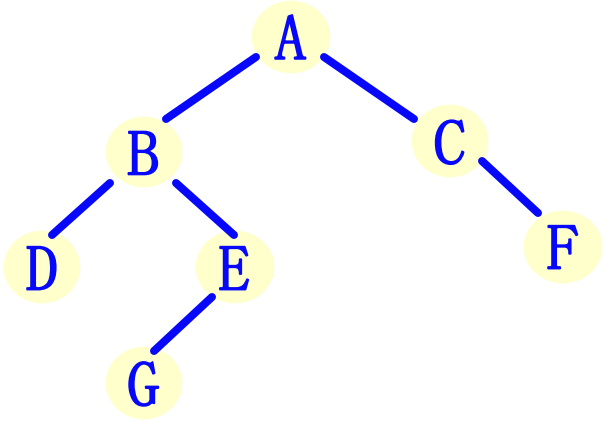
（2）一个森林增加一个根结点成为树。



森林

5.2 二叉树

树是一种分枝结构的对象，在树的概念中，对每一个结点孩子的个数没有限制，因此树的形态多种多样，本节我们主要讨论另一种树型结构——二叉树。



5.2 二叉树

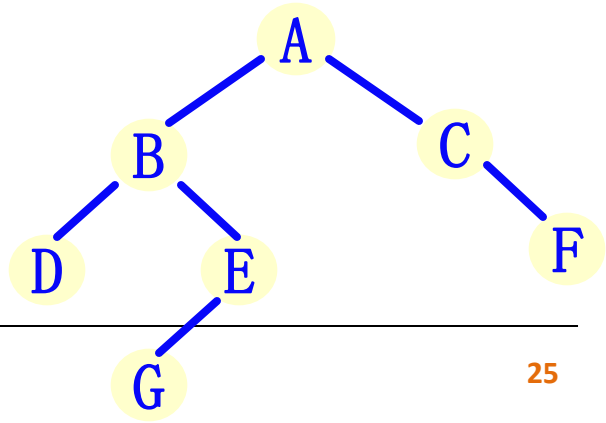
5.2.1 二叉树的概念

5.2.2 二叉树的性质

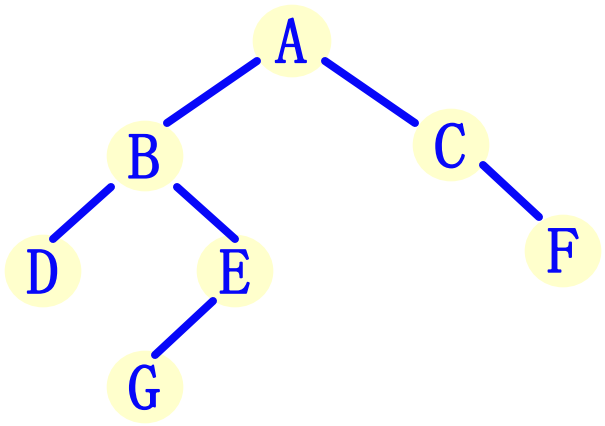
5.2.3 二叉树的存储结构

5.2.1 二叉树的概念

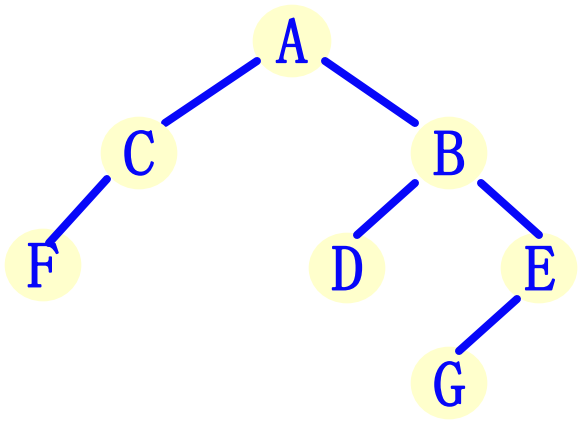
- 概念：二叉树或为空树，或由根及两颗不相交的左、右子树构成，并且左、右子树本身也是二叉树。
- 特点：
  - 二叉树中每个结点最多有两棵子树；即二叉树每个结点度小于等于2;
  - 左、右子树不能颠倒——有序树;
  - 二叉树是递归结构，在二叉树的定义中又用到了二叉树的概念;



5.2.1 二叉树的概念



(a)

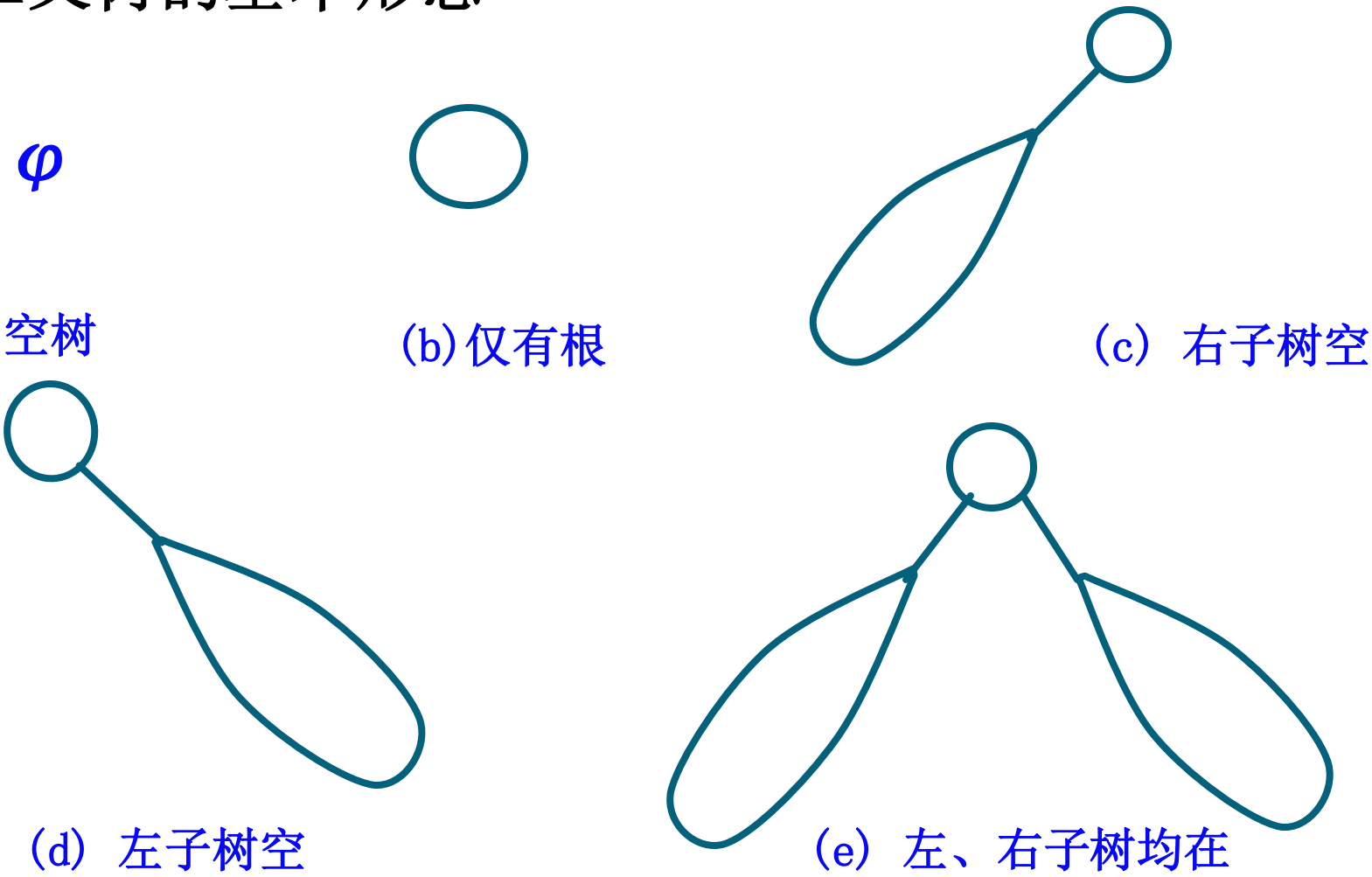


(b)

二叉树是有左右之分的

5.2.1 二叉树的概念

- 二叉树的基本形态



5.2.2 二叉树的性质

性质1 在二叉树的第i(i≥1)层上**至多**有2^{i-1}个结点。

证明：用数学归纳法就可以证明。

1）验证i=1时，只有一个根结点，$2^{i-1} = 2^0 = 1$,是对的；

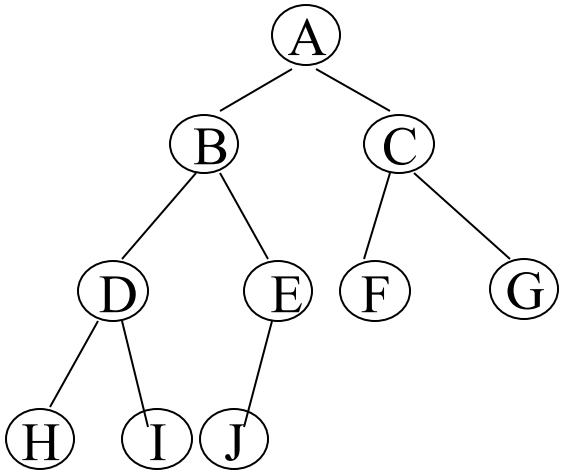
2）假设对所有的j，j<i，命题成立，即第j层上至多有2^{j-1}个结点；

3）由归纳假设，第i-1层至多有2^{i-2}个结点，由于二叉树每个结点度至多为2，故第i层的最大结点数为i-1层的两倍，即2^{i-1}。

性质2 深度为k的二叉树**最多**有2^k-1个结点。

证明：各层的最大结点个数相加，即

$$1+2+4+\cdots+2^{k-1}=2^k-1$$



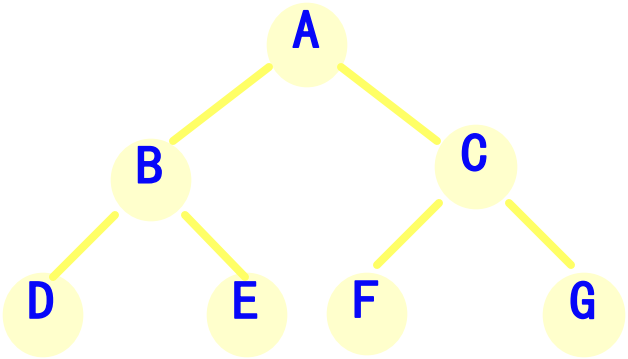
k层二叉树

5.2.2 二叉树的性质

- 两种特殊的二叉树

满二叉树：如果深度为k的二叉树，有2^k-1个结点则称为满二叉树，即每一层的结点数为最大结点数。

满二叉树的连续编号：约定编号从根结点起，自上而下，自左至右。



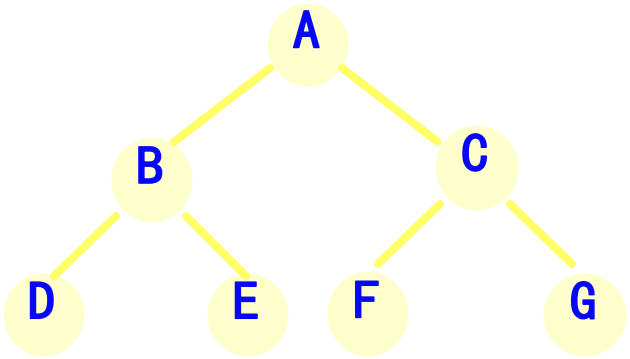
(a)

K=3的满二叉树

5.2.2 二叉树的性质

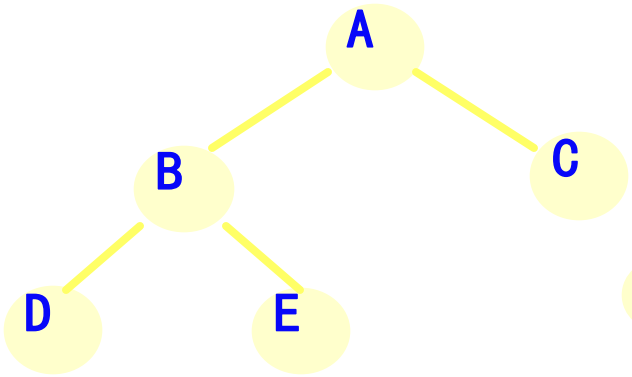
- 两种特殊的二叉树

完全二叉树：二叉树中所含的 n 个结点和满二叉树中编号为1至n的结点一一对应。



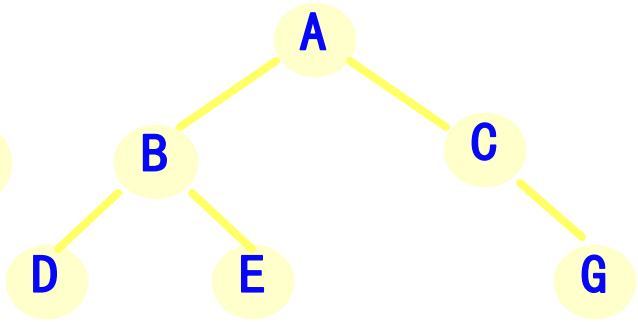
(a)

K=3的满二叉树



(b)

(b)完全二叉树



(c)

(c) 不是完全二叉树

结论：满二叉树一定是完全二叉树，反之不一定

5.2.2 二叉树的性质

性质3 具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$

证明：设所求完全二叉树的深度为k

由性质2和完全二叉树的定义知：

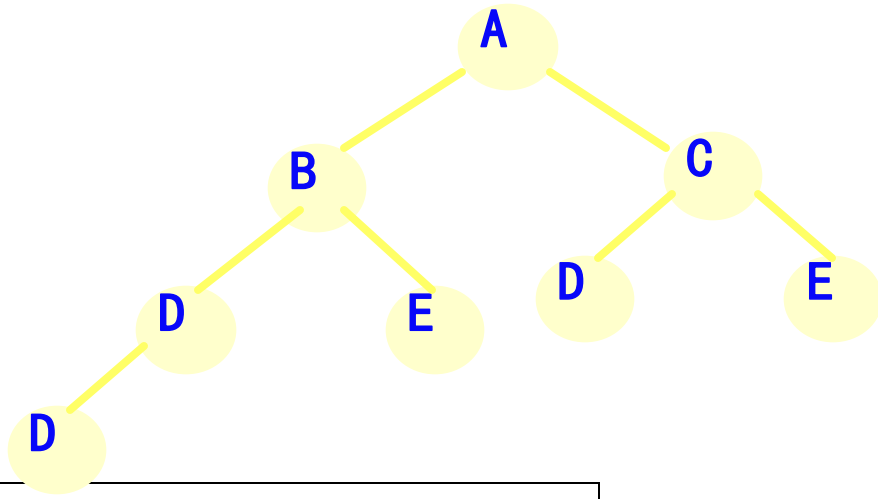
k-1层二叉树的最多结点数 ← $2^{k-1}-1 < n \le 2^k-1$ → k层的最多结点数

由此可以推出：$2^{k-1} \le n < 2^k$

取对数得：$k-1 \le \log_2 n < k$

由于k为整数，故有$k-1=\lfloor \log_2 n \rfloor$

即：$k=\lfloor \log_2 n \rfloor + 1$



性质2:深度为k的二叉树最多有2^k-1个结点

5.2.2 二叉树的性质

性质4　对任意二叉树T，如果度数为0的结点数为n_0，度数为1的结点数为n_1，度数为2的结点数为n_2，则$n_0=n_2+1$。

证明：二叉树T的结点总数　$n=n_0+n_1+n_2$　　（1）

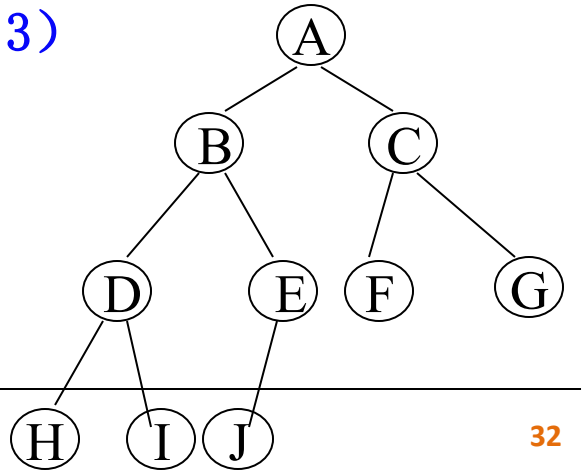
二叉树的分支总数，即除去根节点的结点总数（除去根节点，其他每个结点都有一个分支进入）　$b=n-1$　　（2）

注意：n_1生成n_1*1个节点，n_2生成n_2*2个节点，根结点不由任何结点产生

由于分支是由度为1和度为2的结点生成的

即分支结点总数　$b=n_1+2*n_2$　　（3）

由(1)(2)(3)得　求得：$n_0=n_2+1$



5.2.2 二叉树的性质

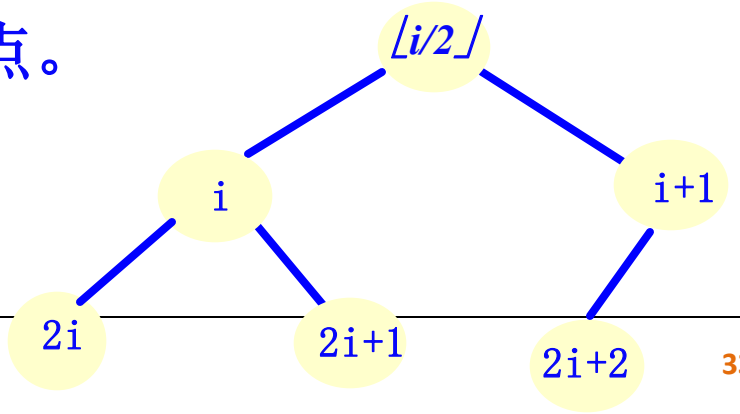
性质5:若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点：

(1) 若 $i=1$，则该结点是二叉树的根，无双亲；如果i>1，编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点；

(2) 若 2i>n，则该结点无左孩子，否则(2i ≤ n)，编号为2i的结点为其左孩子结点；

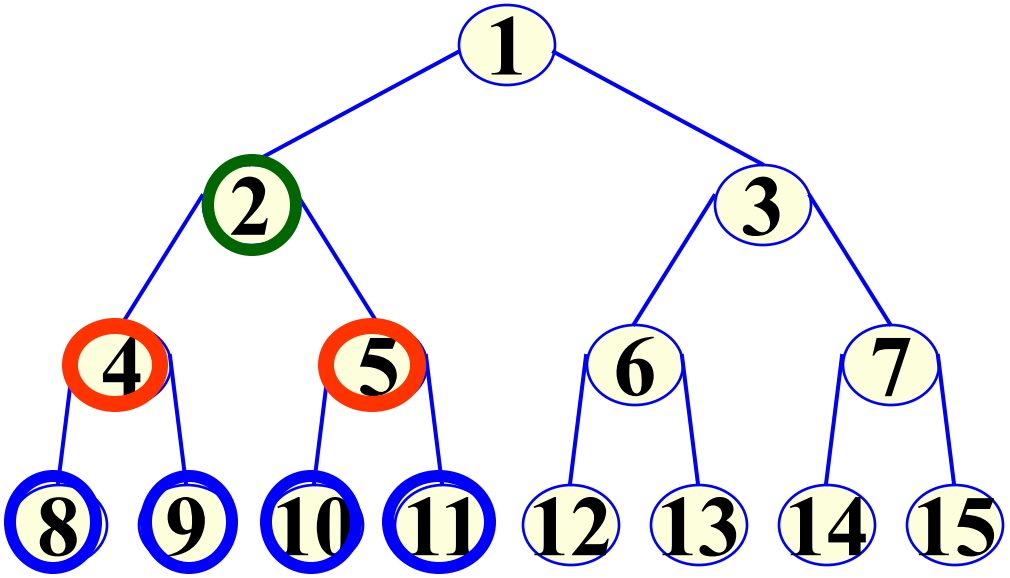
(3) 若 2i+1>n，则该结点无右孩子结点，否则(2i+1≤n)，编号为2i+1 的结点为其右孩子结点。

连续编号的特点（优点）



5.2.2 二叉树的性质

i=1 只有根结点



编号i=4
双亲为⌊i/2⌋=2
左子树为2i=8
右子树为2i+1=9

编号i=5
双亲为⌊i/2⌋=2
左子树为2i=10
右子树为2i+1=11

i=8, n=15
2i>n

无左子树

通过性质5把非线性结构转化成了线性结构

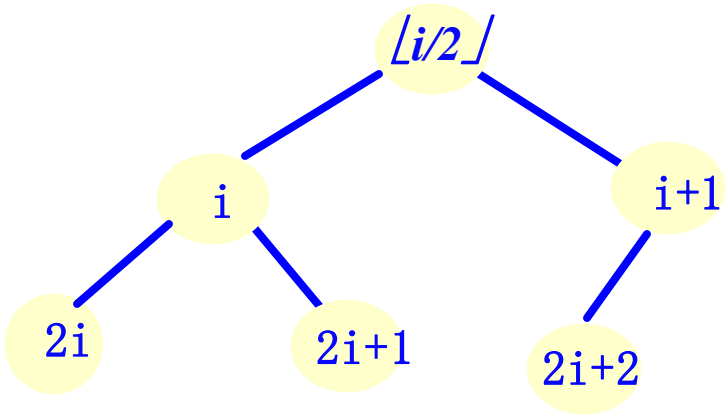
5.2.3 二叉树的存储结构

- 二叉树的顺序存储表示
- 二叉树的链式存储表示

5.2.3 二叉树的存储结构

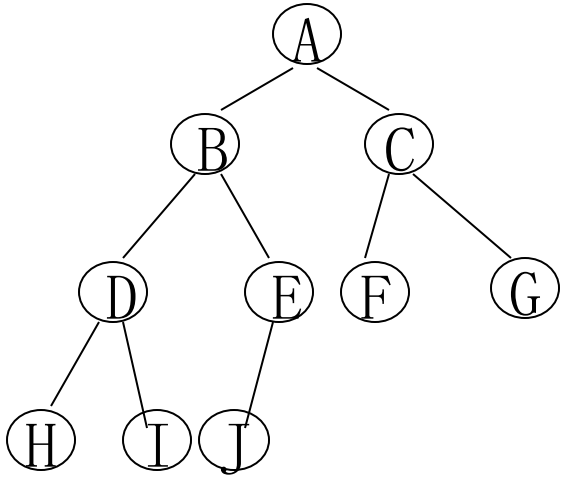
- 二叉树的顺序存储表示

(1) 完全（或满）二叉树



采用一维数组，按层序顺序依次存储二叉树的每一个结点。
如下图所示：

A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10



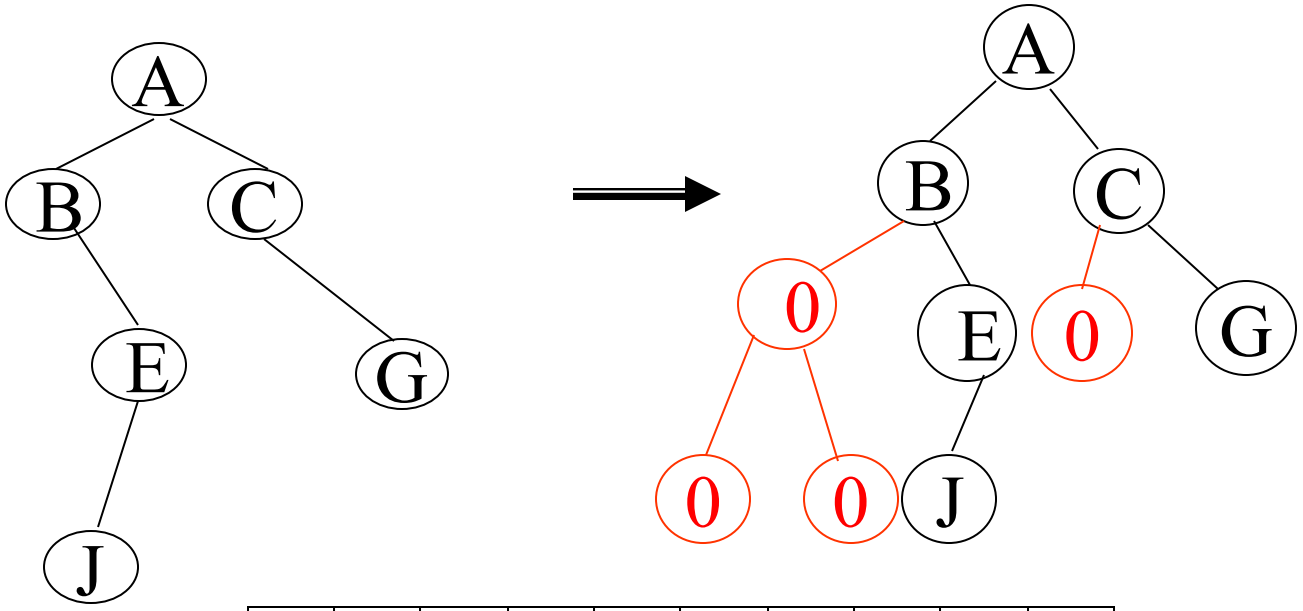
利用性质5实现线性结构和非线性结构的灵活转换。

5.2.3 二叉树的存储结构

- 二叉树的顺序存储表示

(2) 一般二叉树

通过虚设部分结点，使其变成相应的完全二叉树。

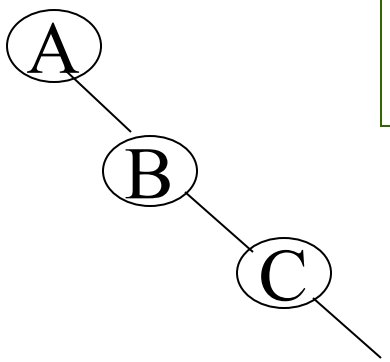


A	B	C	0	E	0	G	0	0	J
1	2	3	4	5	6	7	8	9	10

5.2.3 二叉树的存储结构

- 二叉树的顺序存储表示

(3)特殊的二叉树



对于一个深度为k且只有k个结点的单支树，却需要长度为2^k-1的一维数组。

说明：顺序存储方式对于畸形二叉树，浪费较大空间

5.2.3 二叉树的存储结构

- 二叉树的链式存储表示

二叉链表存储：

二叉链表中每个结点包含三个域：数据域、左指针域、右指针域

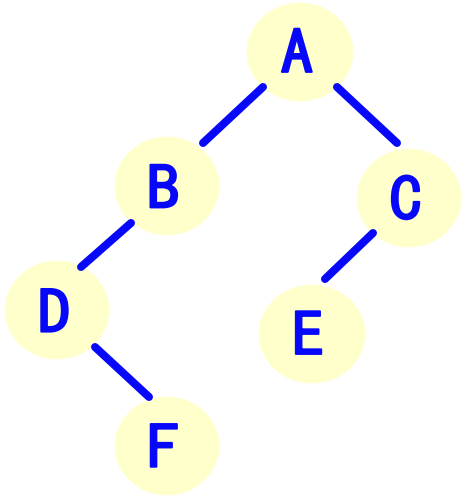
lch	data	rch

C 语言的类型描述如下:

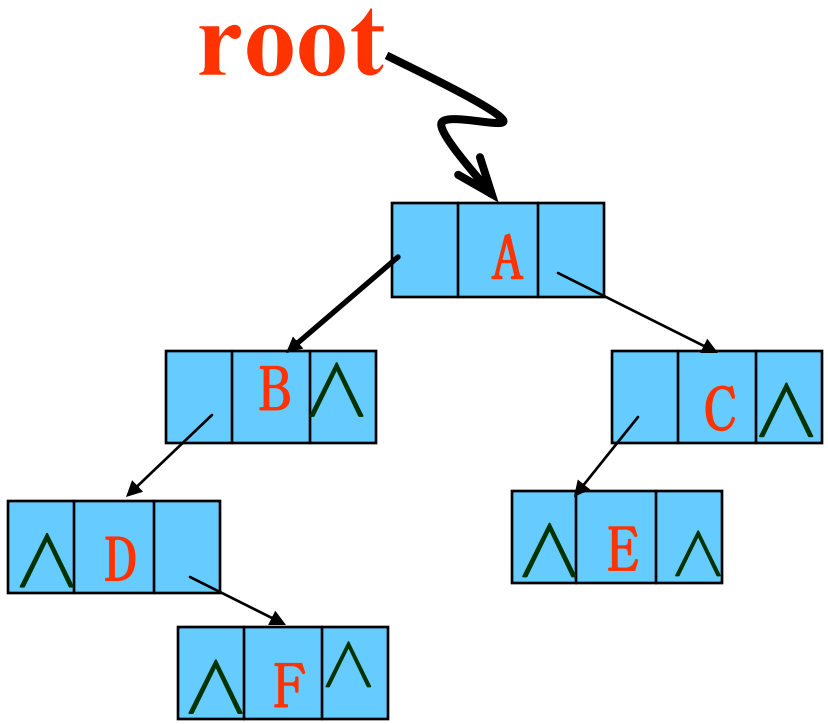
```
typedef Struct node
    { DataType  data;
      Struct node *lch,*rch;
    }BinTNode;
```

5.2.3 二叉树的存储结构

- 二叉树的链式存储表示



二叉树



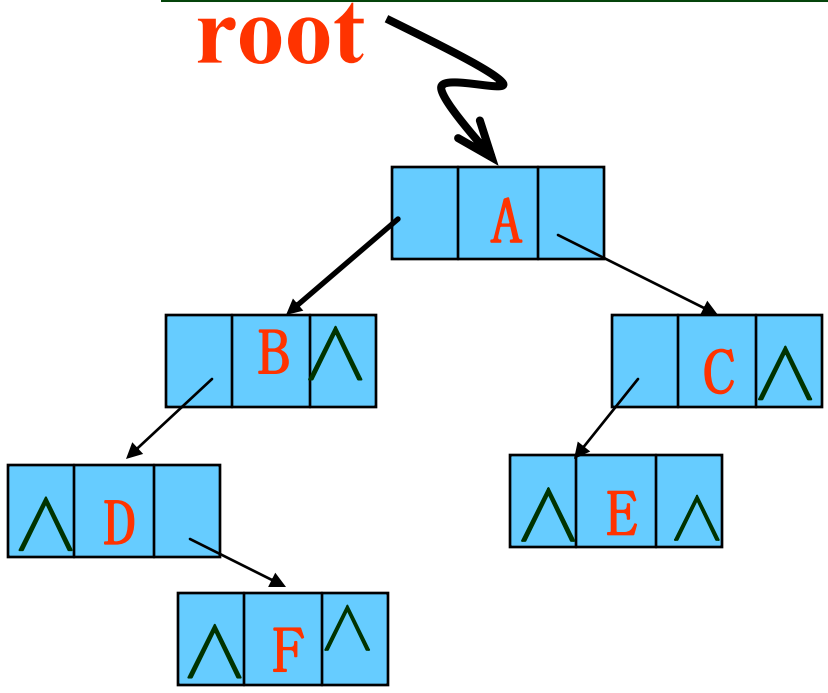
二叉链表图示

n 个结点的二叉树中，
有多少个空链接域？

5.2.3 二叉树的存储结构

- 二叉树的链式存储表示

性质6：n 个结点的二叉树中，共有 n+1 个空指针域。



二叉链表图示

证：n个结点总的指针域数2n；

除了根结点外，其余n-1个结点都是由指针域指出的结点；

所以，剩余的结点数即空指针域个数为：

2n-（n-1）=n+1

二叉链表的缺点是很难找到结点的双亲

5.2.3 二叉树的存储结构

- 二叉树的链式存储表示---三叉链表

三叉链表（带双亲指针的二叉链表）：三叉链表中每个结点包含四个域：数据域、左指针域、右指针域、双亲指针域

结点结构:

lch	data	rch	parent

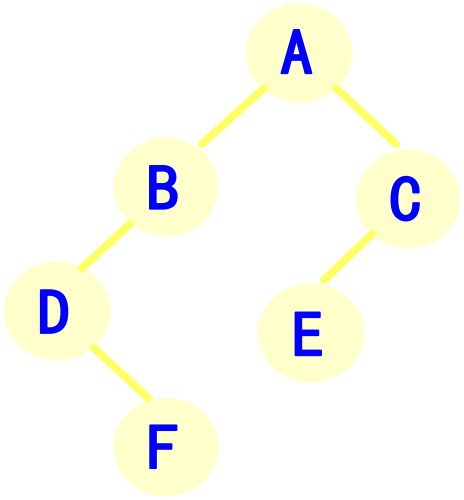
C 语言的类型描述如下:

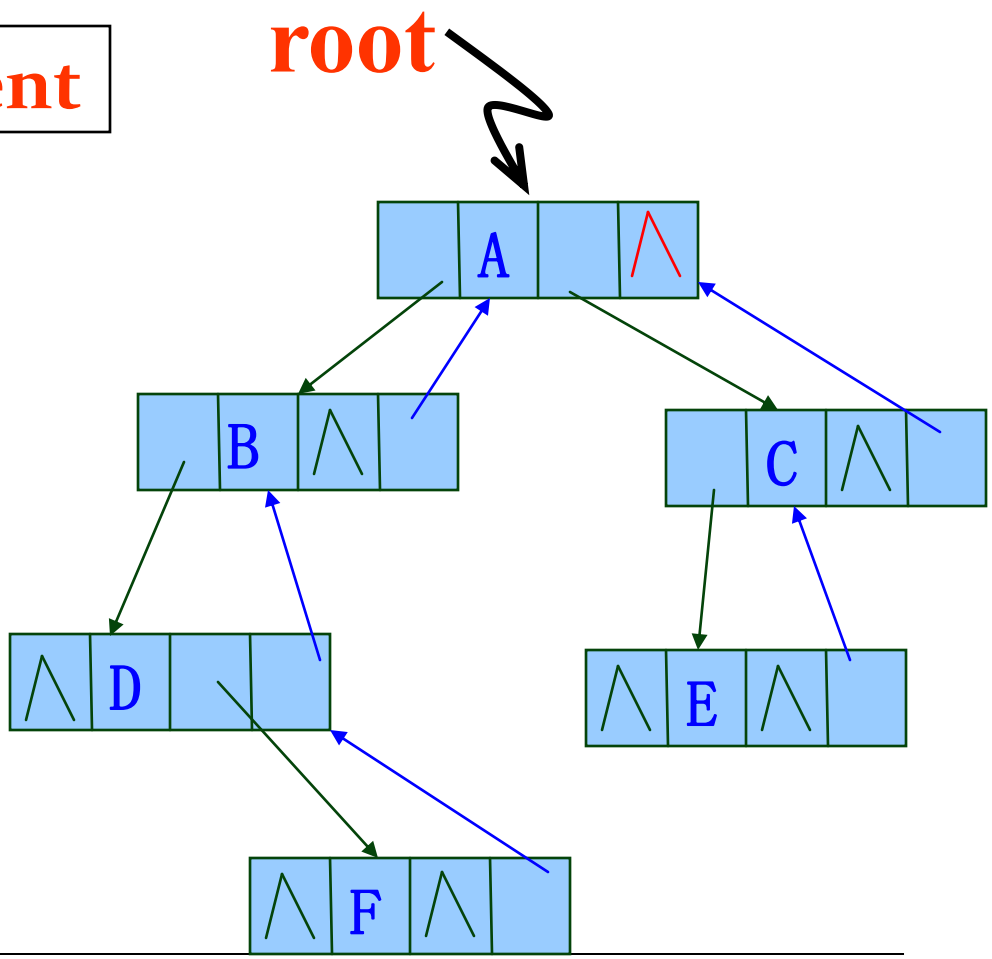
```
typedef  Struct node
{
    DataType data;
    Struct node *lch,*rch,*parent;
};
```

5.2.3 二叉树的存储结构

- 二叉树的链式存储表示---三叉链表

lch	data	rch	parent





5.3 二叉树的遍历

5.3.1 二叉树的遍历方法

5.3.2 遍历的递归算法

5.3.3 遍历的非递归算法

5.3.1 二叉树的遍历方法

- 遍历：
  - 按某种搜索路径访问二叉树的每个结点，而且每个结点仅被访问一次。
- 访问：
  - 访问是指对结点进行各种操作的简称，包括输出、查找、修改等等操作。

遍历是各种数据结构最基本的操作，许多其它的操作可以在遍历基础上实现。

5.3.1 二叉树的遍历方法

- “遍历”是任何类型均有的操作：
  - 线性结构的遍历：只有一条搜索路径(因为每个结点均只有一个后继)；
  - 非线性结构的遍历：二叉树是非线性结构，则存在如何遍历；即按什么样的搜索路径遍历的问题。

如何访问二叉树的每个结点，
而且每个结点仅被访问一次？

需要找到一种方法，使得二叉树的所有结点都排列成一个线性序列

5.3.1 二叉树的遍历方法

- 对“二叉树”而言，可以有三条搜索路径：
  - 先上后下的按层次遍历；
  - 先左（子树）后右（子树）的遍历；
  - 先右（子树）后左（子树）的遍历。

5.3.1 二叉树的遍历方法

二叉树由根、左子树、右子树三部分组成

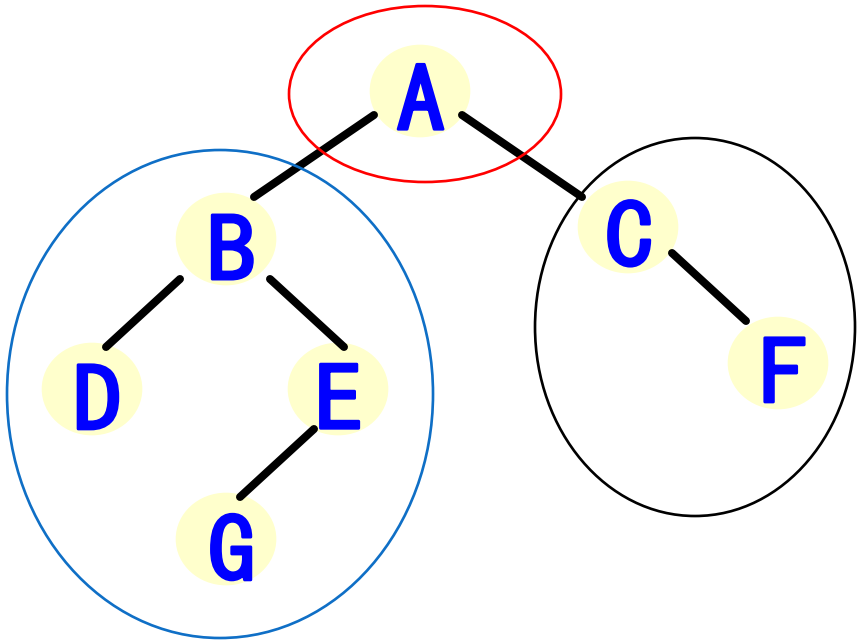
二叉树的遍历可以分解为：访问根，遍历左子树和遍历右子树

令：L：遍历左子树
T：访问根结点
R：遍历右子树

有六种遍历方法：

T L R，L T R，L R T，

T R L，R T L，R L T

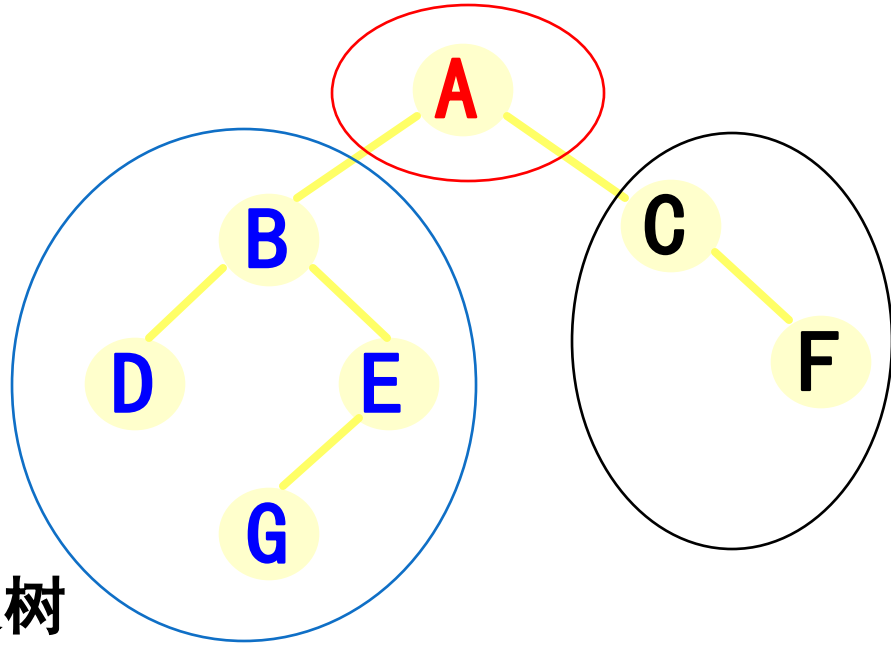


约定先左后右，有三种遍历方法： T L R、L T R、L R T ，分别称为
先序（先根）遍历、中序（中根）遍历、后序（后根）遍历

5.3.1 二叉树的遍历方法

- 中序遍历（ L T R ）

若二叉树非空
（1）中序遍历左子树
（2）访问根结点
（3）中序遍历右子树

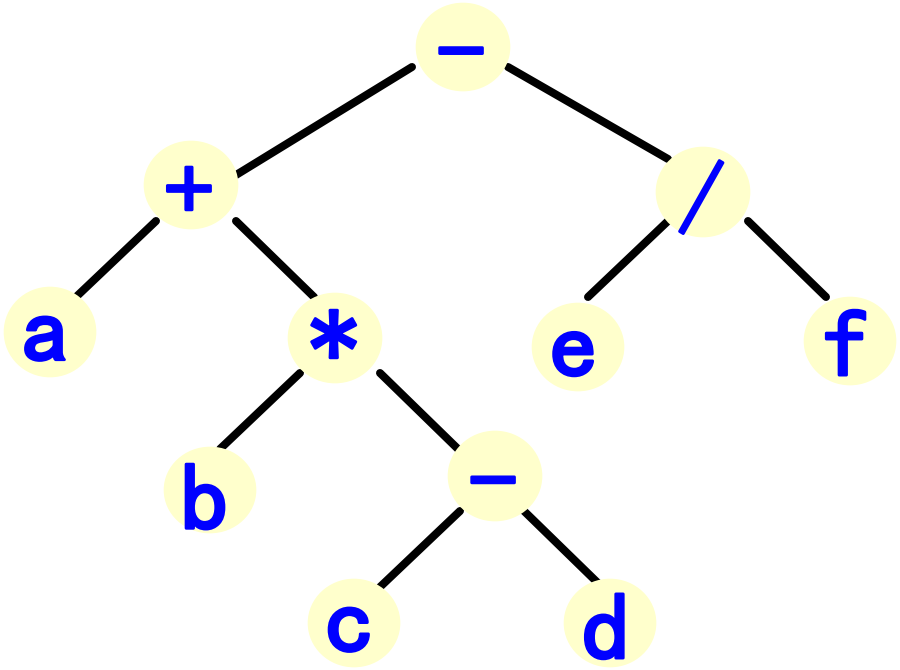


例 中序遍历右图所示的二叉树

中序遍历序列： D ,B ,G ,E ,A,C ,F

5.3.1 二叉树的遍历方法

练习 表达式 a+b*(c-d)-e/f 用二叉树表示如下：

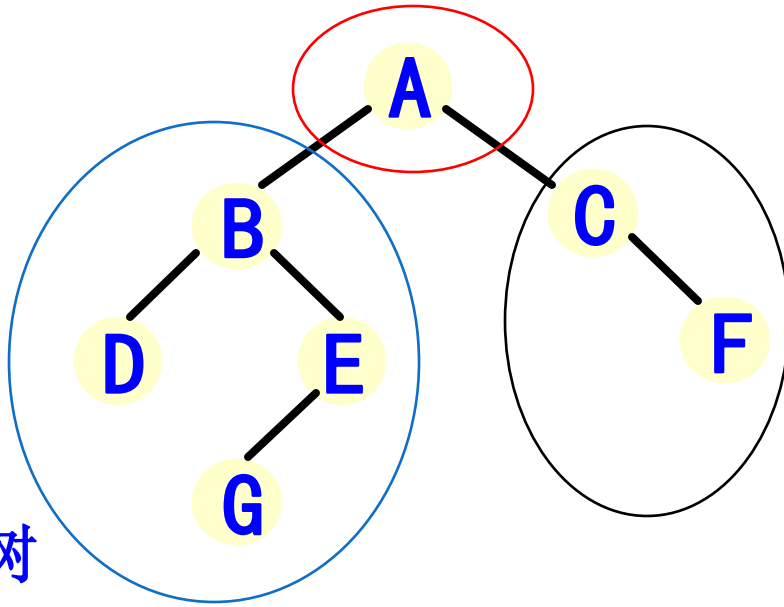


中序 a+b*c-d-e/f ---- 中缀表示

5.3.1 二叉树的遍历方法

- 先序遍历（T L R）

若二叉树非空
（1）访问根结点；
（2）先序遍历左子树；
（3）先序遍历右子树；



例 先序遍历右图所示的二叉树

（1）访问根结点A

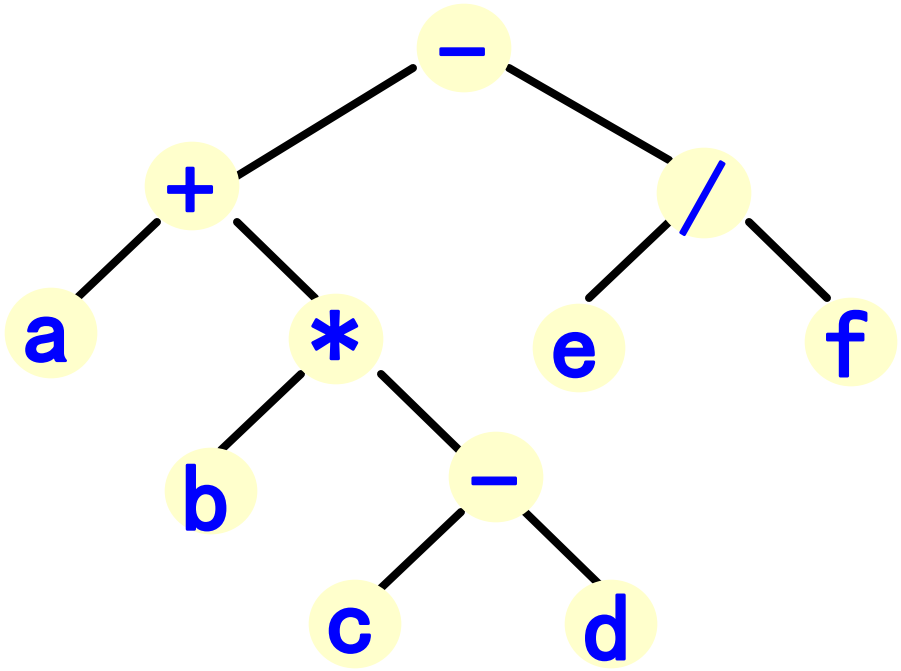
（2）先序遍历左子树：即按 T L R 的顺序遍历左子树

（3）先序遍历右子树：即按 T L R 的顺序遍历右子树

先序遍历序列：A, B, D, E, G, C, F

5.3.1 二叉树的遍历方法

练习 先序遍历下图所示的二叉树



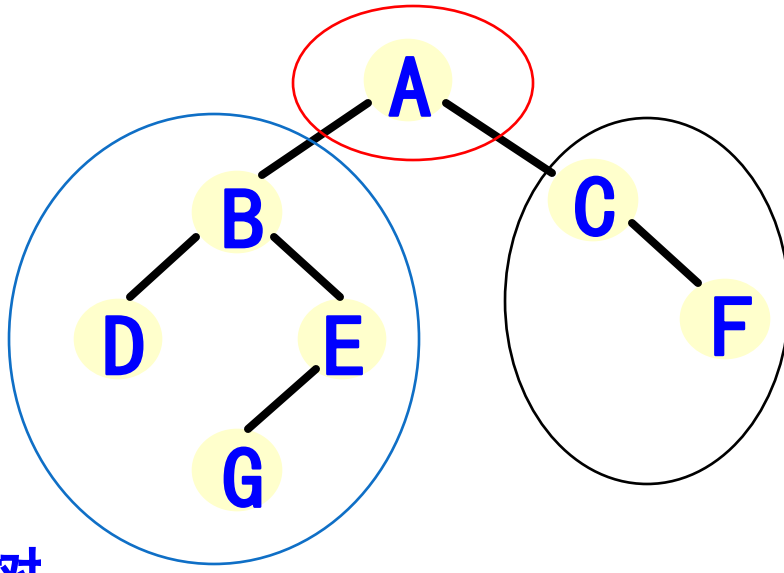
先序 - + a * b - c d / e f

前缀表示（波兰式）

5.3.1 二叉树的遍历方法

- 后序遍历（L R T）

若二叉树非空
（1）后序遍历左子树
（2）后序遍历右子树
（3）访问根结点



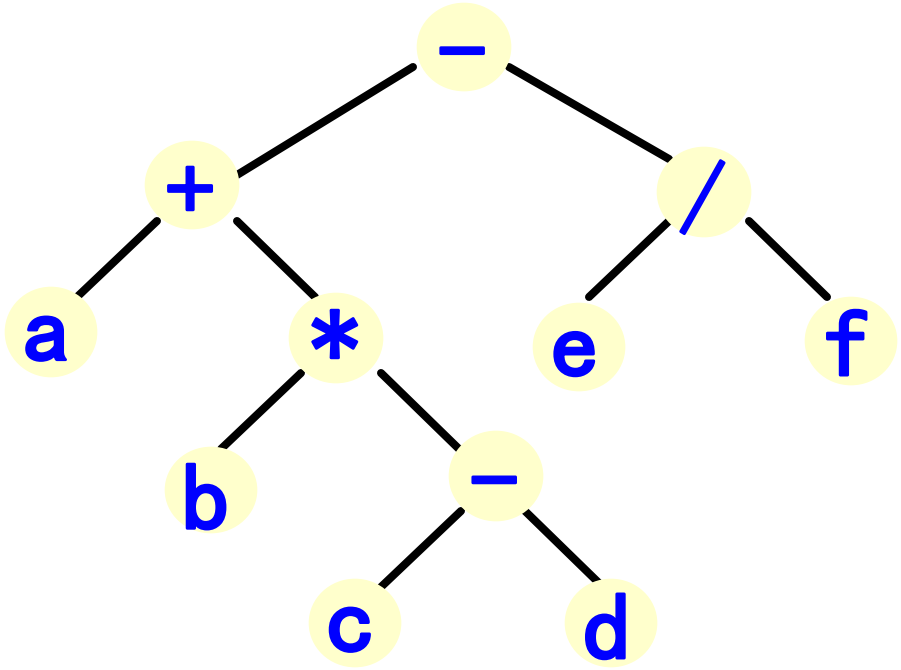
例 后序遍历右图所示的二叉树

（1）后序遍历左子树：即按 L R T 的顺序遍历左子树
（2）后序遍历右子树：即按 L R T 的顺序遍历右子树
（3）访问根结点A

后序遍历序列： D, G, E, B, F, C, A

5.3.1 二叉树的遍历方法

练习 后序遍历下图所示的二叉树



后序 a b c d - * + e f/ -

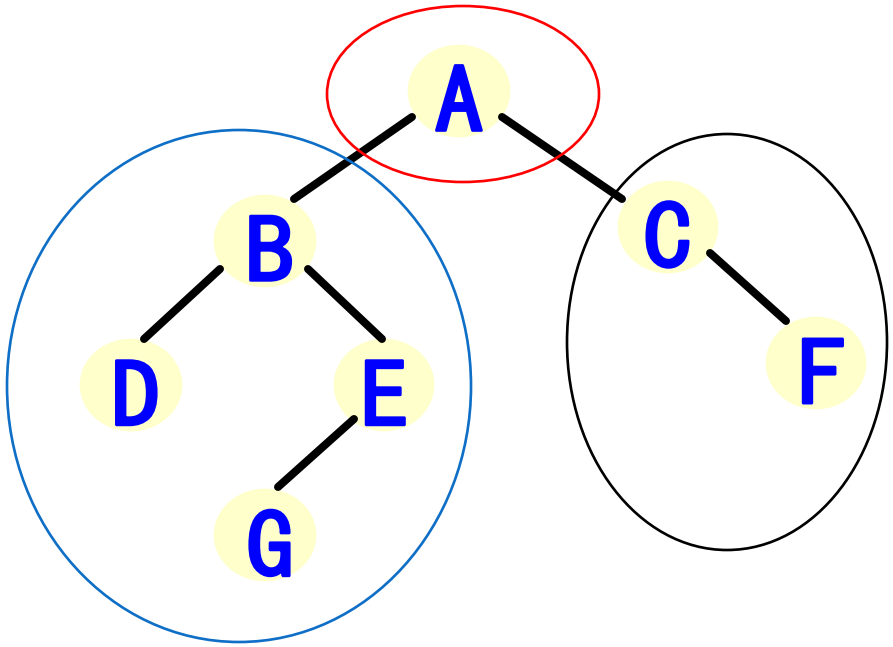
后缀表示（逆波兰式）

5.3.1 二叉树的遍历方法

- 按层遍历

层次遍历序列：

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



5.3.1 二叉树的遍历方法

- 按层遍历

按层遍历引入了队列作为辅助工具。

算法思想为：

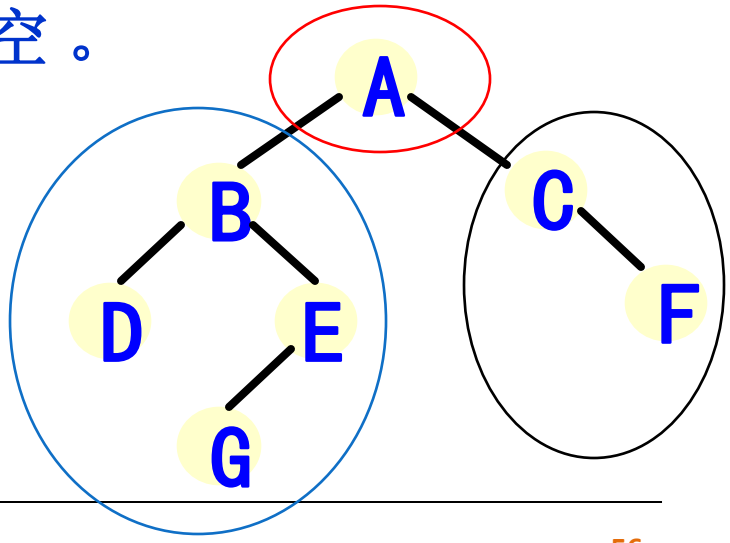
（1）将二叉树根入队列；

（2）将队头元素出队列，并判断此元素是否有左右孩子，若有，则将它的左右孩子入列，否则转（3）；

（3）重复步骤（2），直到队列为空 。

队列的厉害之处

课后思考：按层遍历算法



5.3 二叉树的遍历

5.3.1 二叉树的遍历方法

5.3.2 遍历的递归算法

5.3.3 遍历的非递归算法

5.3.2 遍历的递归算法

- 先序遍历（TLR）的定义

若二叉树非空

（1）访问根结点；

（2）先序遍历左子树

（3）先序遍历右子树；

上面先序遍历的定义等价于：

若二叉树为空，结束　　——基本项（也叫终止项）

若二叉树非空　　——递归项

（1）访问根结点；

（2）先序遍历左子树；

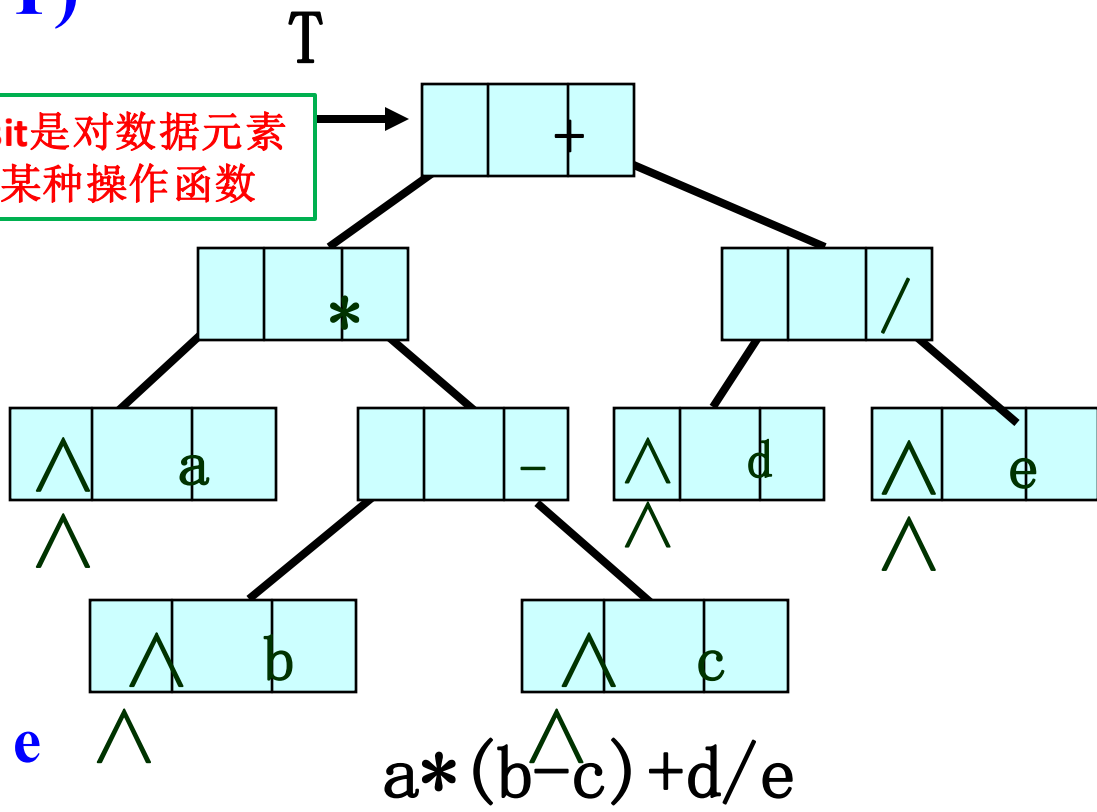
（3）先序遍历右子树；

5.3.2 遍历的递归算法

- 先序遍历递归算法

```
void Prev (BinTree   T)
 { if (T)
   {  visit(T->data);
      Prev(T->lch);
      Prev(T->rch);
   }
 }
```

visit是对数据元素的某种操作函数

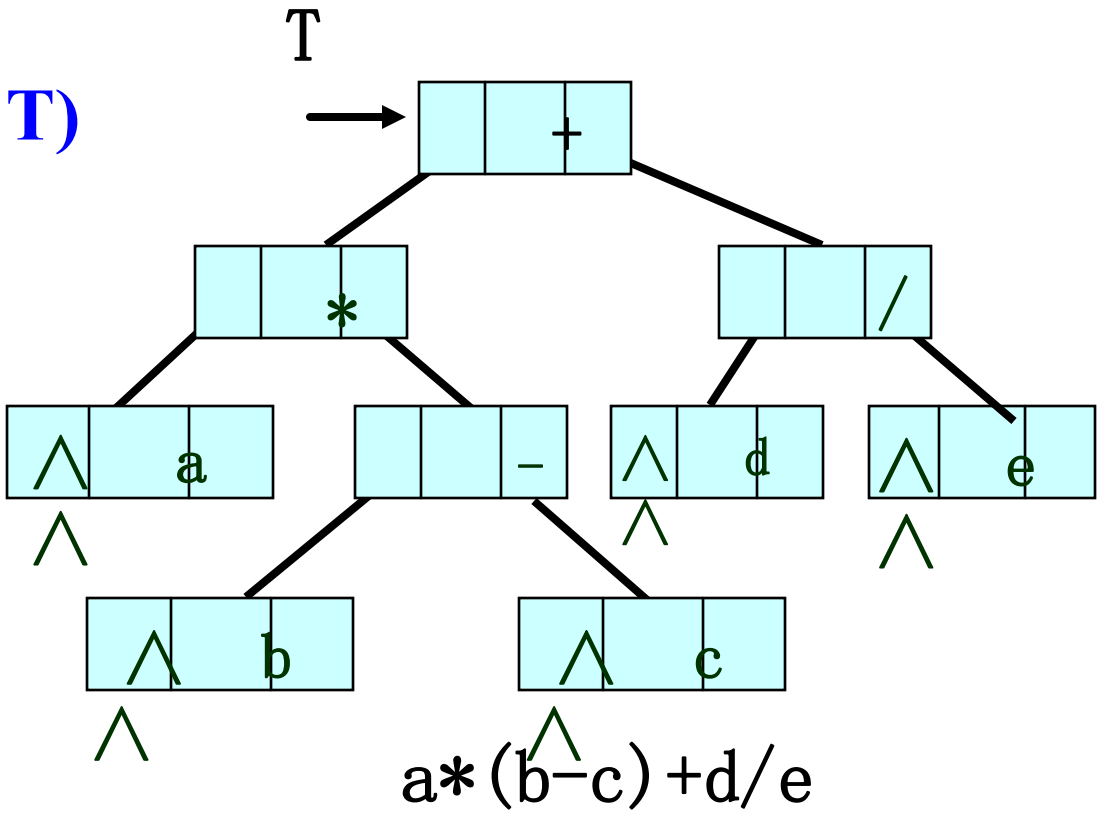


先序序列为 + * a – b c / d e
称为前缀表达式

5.3.2 遍历的递归算法

- 中序遍历递归算法

```
void  Mid (BinTree   T)
{ if (T)
  { Mid(T->lch);
    visit( T->data);
    Mid(T->rch);
  }
}
```

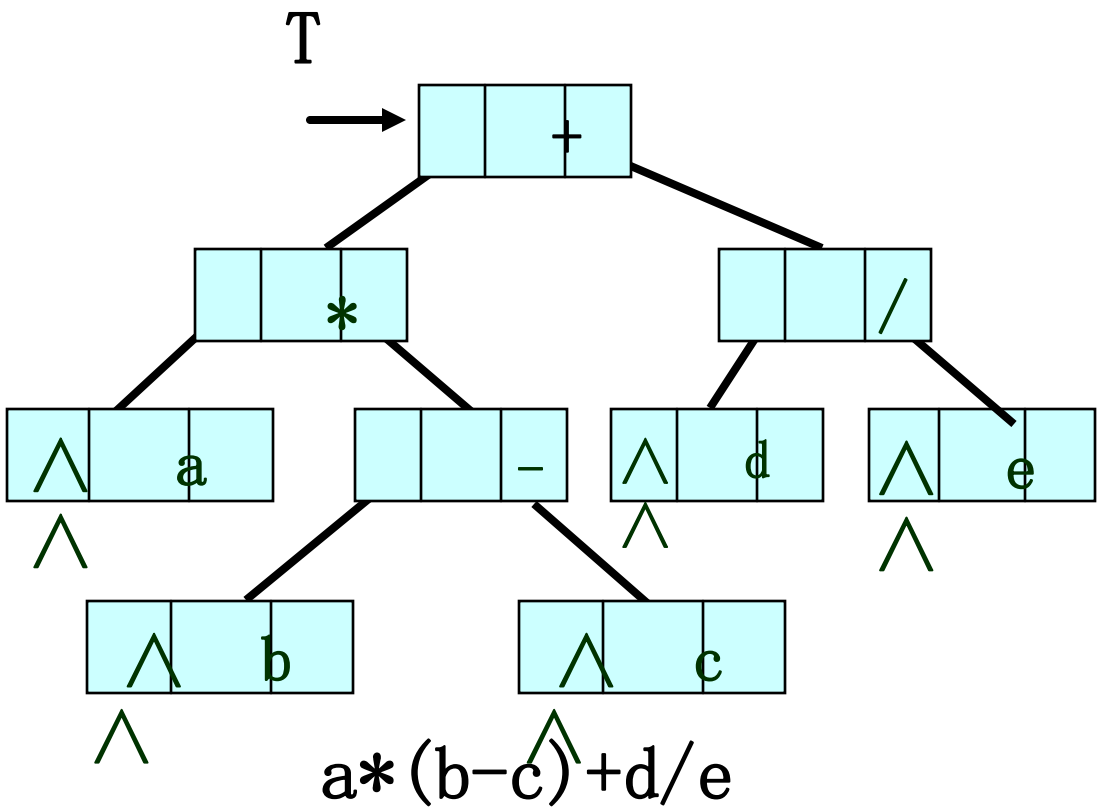


中序序列为 a * b – c+ d / e
称为中缀表达式

5.3.2 遍历的递归算法

- 后序遍历递归算法

```
void Post(BinTree   T)
 { if (T)
   {  Post(T->lch);
      Post(T->rch);
      visit( T->data);
   }
 }
```



后序序列为　a b c – * d e / +
称为后缀表达式

5.3 二叉树的遍历

5.3.1 二叉树的遍历方法

5.3.2 遍历的递归算法

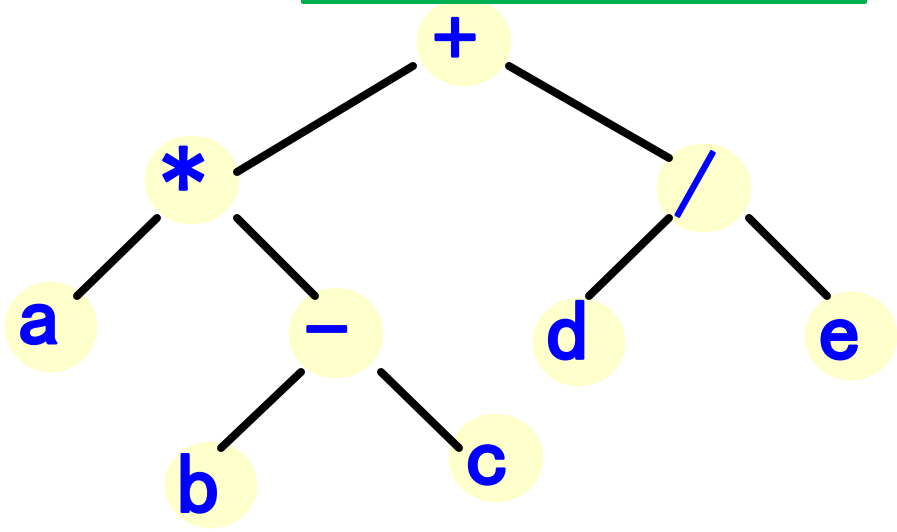
5.3.3 遍历的非递归算法

5.3.3 遍历的非递归算法

- 中序遍历的非递归算法

```
BiTNode *GoFarLeft(BiTree T, Stack *S)
{//找到左子树的最左下的结点
  if (!T )  return NULL;
  while (T->lch ){
    Push(S, T);
    T = T->lch;
  }
  return T;
}
```

中序遍历：先遍历左子树，所以当经过一个结点时，如果其有左子树，先访问其左子树，此时将其入栈，以便从左子树返回时，通过栈找到回到之前经过但未访问的结点



中序序列为：　a * b – c+ d / e

5.3.3 遍历的非递归算法

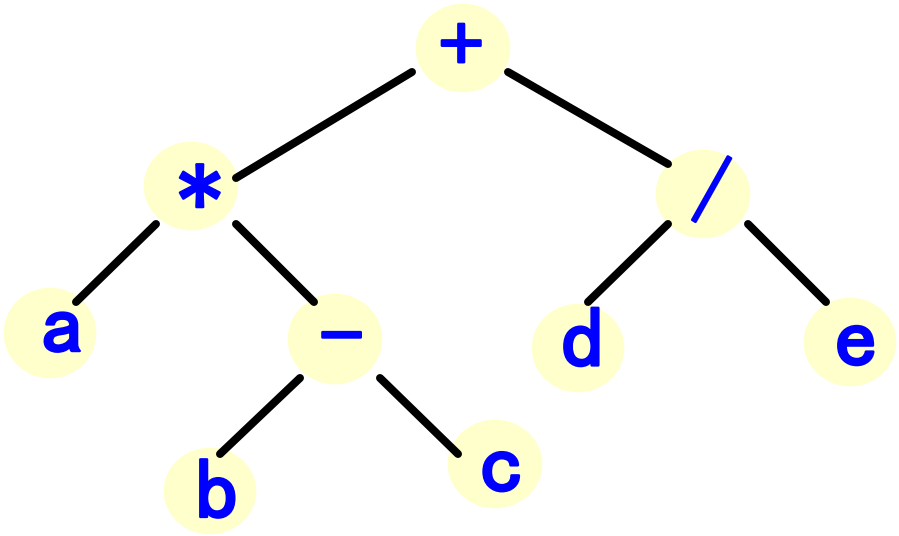
- 中序遍历的非递归算法

```
void Inorder_I(BiTree T)
{   Stack *S;
   t = GoFarLeft(T, S);  // 找到最左下的结点
   while(t){
      visit(t->data);
      if (t->rch)
         t = GoFarLeft(t->rch, S);//从右子树根结点出发，继续往左走
      else if ( !StackEmpty(S ))    // 栈不空时退栈
         t = Pop(S);
      else
         t = NULL; // 栈空表明遍历结束
    } // while
}// Inorder_I
```

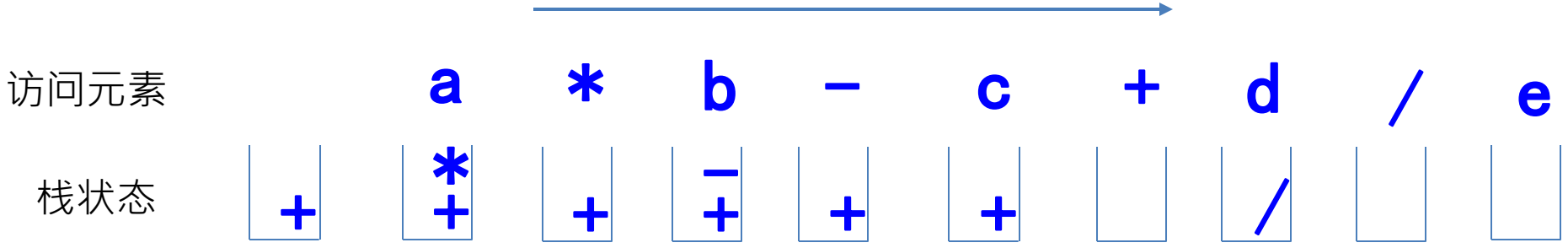
先找到最左下子树，然后：
对于当前结点，或者没有左子树，或者左子树已经访问，所以接下来的整个思路：
1）先访问当前结点，
2）如果有右子树，则去访问右子树，如果没有右子树，则通过栈返回到之前经过但未访问的结点
通过栈记录了相对的层次关系，起到了递归的作用

5.3.3 遍历的非递归算法

- 中序遍历的非递归算法



先找到最左下子树，然后：
对于当前结点，或者没有左子树，或者左子树已经访问，所以接下来的整个思路：
1）先访问当前结点，
2）如果有右子树，则去访问右子树，如果没有右子树，则通过栈返回到之前经过但未访问的结点



5.4 遍历的应用

遍历是二叉树各种操作的基础，可以在遍历过程中对结点进行各种操作：

（1）求结点的双亲、孩子结点、结点的层次；

（2）遍历过程中生成结点，建立二叉树；

…… ……

遍历二叉树的过程实质是把二叉树的非线性结构的结点进行线性排列的过程。

5.4 遍历的应用

5.4.1 遍历的基本应用

5.4.2 二叉树的遍历与存储结构的应用

5.4.3 二叉树的相似与等价

5.4.1 遍历的基本应用

- 二叉树的生成

递归建立二叉树

我们按先序递归遍历的思想来建立二叉树。

其建立思想如下：

（1）建立二叉树的根结点；

（2）先序建立二叉树的左子树；

（3）先序建立二叉树的右子树。

5.4.1 遍历的基本应用

二叉树的生成

二叉树的生成的递归算法

```
bitree *creat()
{
   bitree *t;
   t=(bitree*)malloc(sizeof(bitree));
   t->data=x;
   t->lch=creat();
   t->rch=creat();
   return t;
}
```

5.4.1 遍历的基本应用

求二叉树的叶子数

算法思想：采用任何遍历方法，遍历时判断访问的结点是不是叶子，若是则叶子数加1。

```
int countleaf(bitree t ,int num)
{ if(t!=NULL)
  {if((t->lch==NULL) &&(t- >rch)==NULL))
       num++;
   num=countleaf(t->lch,num);
   num=countleaf(t->rch,num);
  }
return num;
}
```

5.4.1 遍历的基本应用

求二叉树的深度

算法思想：从第一层的根结点开始往下递推可得到。

求二叉树深度的递归算法(后序遍历)

```
Int treedepth(bitree *t)
 {int h,lh,rh;
  if(t==NULL)    h=0; //递归停止条件
  else {
            lh=treedepth(t->lch);
            rh=treedepth(t->rch);
            if(lh>=rh)  h=lh+1;
            else   h=rh+1;
        }
   return h;
 }
```

体会递归方程的魅力

5.4 遍历的应用

5.4.1 遍历的基本应用

5.4.2 二叉树的遍历与存储结构的应用

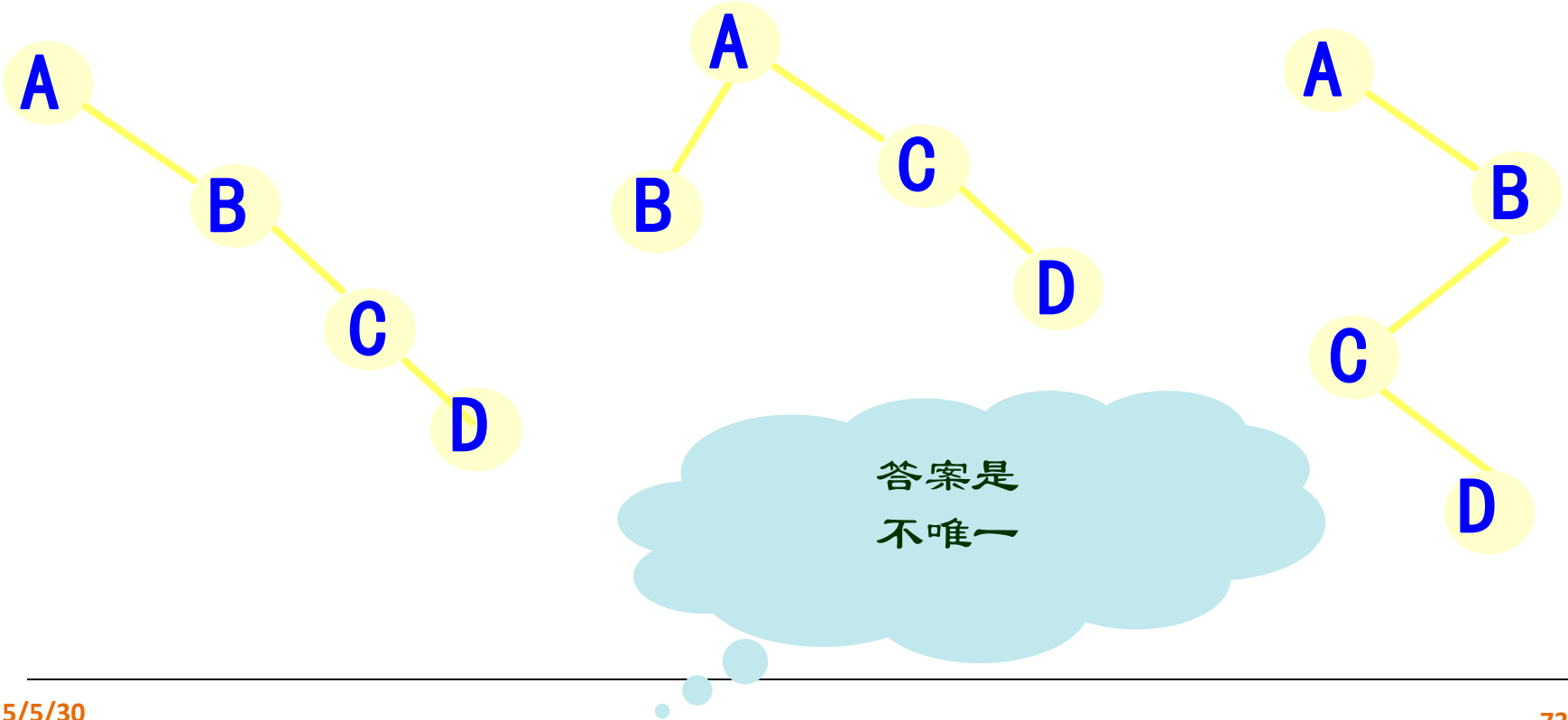
5.4.3 二叉树的相似与等价

5.4.2 二叉树的遍历与存储结构的应用

□二叉树的遍历与存储结构之间的转化

问题：给定一个遍历序列，能否唯一确定一棵二叉树？

例如：先序序列为ABCD,其二叉树的结构是什么？



5.4.2 二叉树的遍历与存储结构的应用

□ 构造二叉树

关键 **(1) 确定二叉树的根结点；**

(2) 结点的左右次序。

给定某两种遍历序列能否唯一确定一棵二叉树?

给定中序和后序 ⟶ 唯一确定一棵二叉树

给定中序和先序 ⟶ 唯一确定一棵二叉树

给定先序和后序不能唯一确定一棵二叉树

5.4.2 二叉树的遍历与存储结构的应用

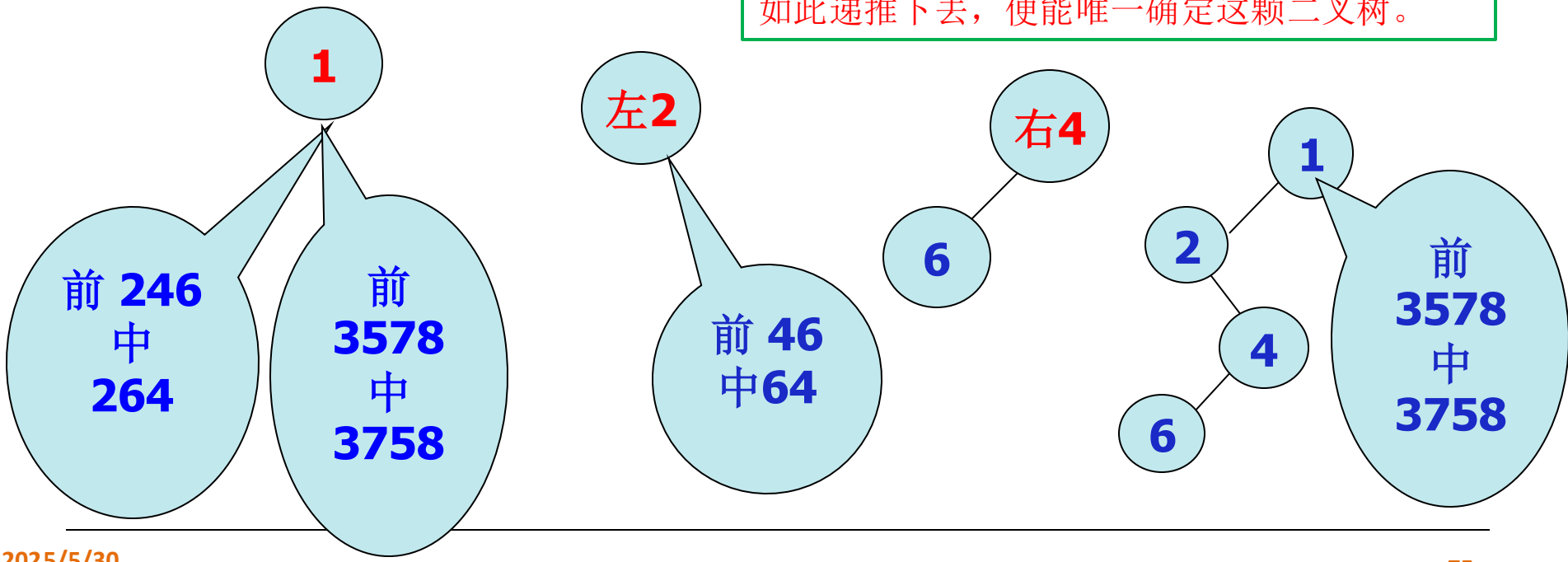
□ 构造二叉树

例 **给定二叉树先序和中序遍历序列，如何构造二叉树?**

先序: 1 2 4 6 3 5 7 8

中序: 2 6 4 1 3 7 5 8

在先序遍历序列中，第一个结点是二叉树的根结点，而在中序遍历中，根结点将中序序列分割成两个子序列：左子树和右子树的中序序列，由此可以确定根节点和左右两颗子树。
如此递推下去，便能唯一确定这颗二叉树。



5.4.2 二叉树的遍历与存储结构的应用

结论：

- “遍历”是二叉树各种操作的基础；
- 可以在遍历过程中对结点进行各种操作，
  - 对于一棵已知树可求结点的双亲；
  - 求结点的孩子结点；
  - 判定结点所在层次；
  - 树的深度；
  - 生成二叉树
  - ……

5.4 遍历的应用

5.4.1 遍历的基本应用

5.4.2 二叉树的遍历与存储结构的应用

5.4.3 二叉树的相似与等价

5.4.2 二叉树的相似与等价

- 二叉树的相似与等价的含义

  两株二叉树具有相同结构指：

  （1）它们都是空的；

  （2）它们都是非空的，且左右子树分别具有相同结构。

“形状”相同

- 定义具有相同结构的二叉树为**相似**二叉树。
- 相似且相应结点包含相同信息的二叉树称为**等价**二叉树。

5.4.2 二叉树的相似与等价

- 判断两株二叉树是否等价

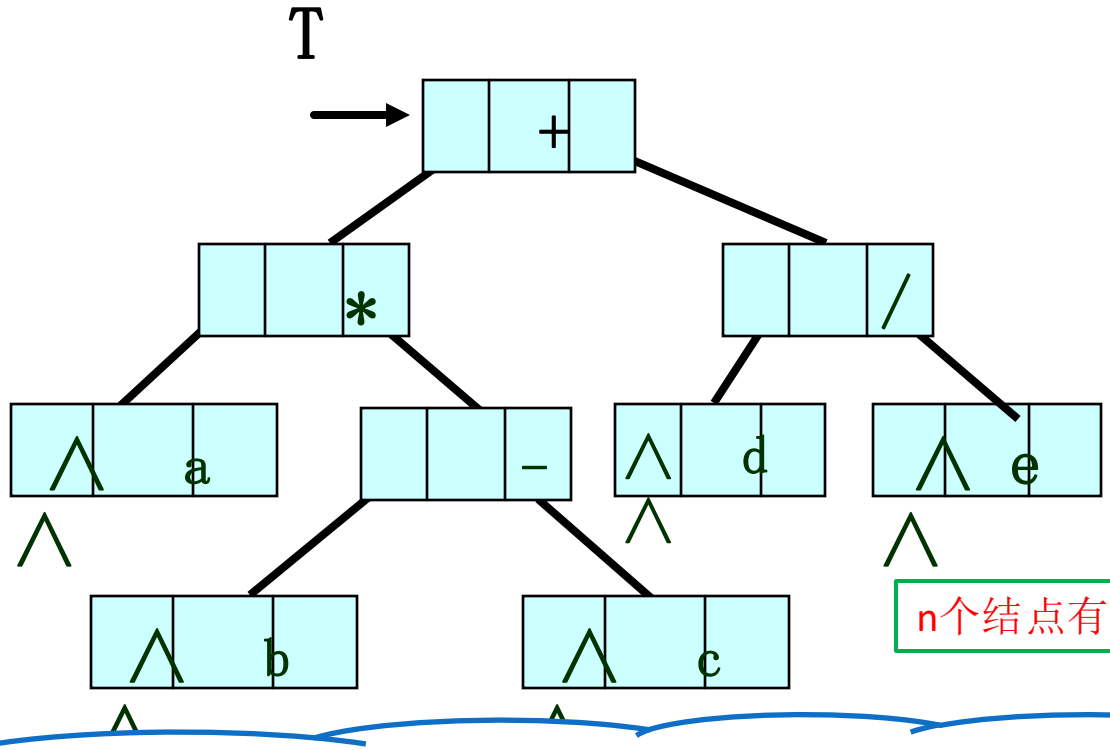
```
int   EQUAL( t1 ,  t2 )
BTREE  t1 , t2 ;
{  int  x ;
   x = 0 ;
   if ( ISEMPTY(t1) && ISEMPTY(t2) )//二叉树空
      x = 1 ;
   else if ( !ISEMPTY( t1 ) && ! ISEMPTY( t2) ) //二叉树不空
      if (  DATA( t1 ) == DATA( t2 )  )
         if ( EQUAL( LCHILD( t1 ) , LCHILD( t2 ) ) )
            x= EQUAL( RCHILD( t1) , RCHILD( t2) )
   return( x ) ;
} /* EQUAL */
```

体会递归方程的魅力

5.4.2 二叉树的相似与等价

- 二叉树的复制

```
BTREE   COPY( BTREE   oldtree )
{
   BTREE   temp ;
   if ( oldtree != NULL )
     {
        temp = new  Node ;
        temp -> lch = COPY( oldtree->lch ) ;
        temp -> rch = COPY( oldtree->rch ) ;
        temp -> data = oldtree->data ;
        return ( temp );
     }
   return ( NULL ) ;
}
```



n个结点有n+1个空指针域。

如何利用二叉链表的空指针域？

线性链表:从单向链表→双向链表

知识回顾

- 正像线性链表可以从单向链表发展到双向链表一样，二叉链表也可以采用双向链表。
- 二叉链表是一种单向链接结构，从一个结点出发，沿着指针走向只能到达其子孙结点，却无法返回其祖先结点。
- 按某种遍历方式对二叉树进行遍历，本质是将二叉树中的非线性结构的结点排列成一个线性序列。但是，二叉树的存储结构不能反映结点的前驱和后继关系，只能在对二叉树遍历的动态过程中得到这些信息。

线索二叉树

5.5 线索二叉树

5.5.1 线索二叉树的表示

5.5.2 二叉树的线索化

5.5.3 线索二叉树的遍历

5.5.1 线索二叉树的表示

- 线索二叉树的概念

➢考虑利用这些空链域来存放遍历后结点的前驱和后继信息，这就是线索二叉树构成的思想。

➢采用既可以指示其前驱又可以指示后继的双向链接结构的二叉树被称为线索二叉树。

对于每个结点，直接加两个指针域，指向前驱和后继结点呢？
存储效率太低。
为何不利用空指针域呢？

利用空链域存储信息

↓

链式存储结构——线索链表

5.5.1 线索二叉树的表示

- 线索链表的结点结构

lch	ltag	data	rtag	rch

标志位比指针域占存储空间更小

```
typedef struct node
 {datatype data;
  struct node *lch, *rch;
  int ltag, rtag;
 }threadbithptr;
```

其中：ltag，rtag为两个标志域

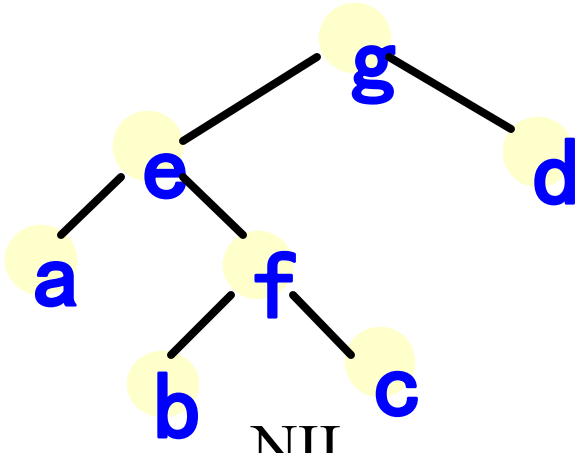
ltag= { 0 lch域指示结点的左孩子
        1 lch域指示结点的前驱

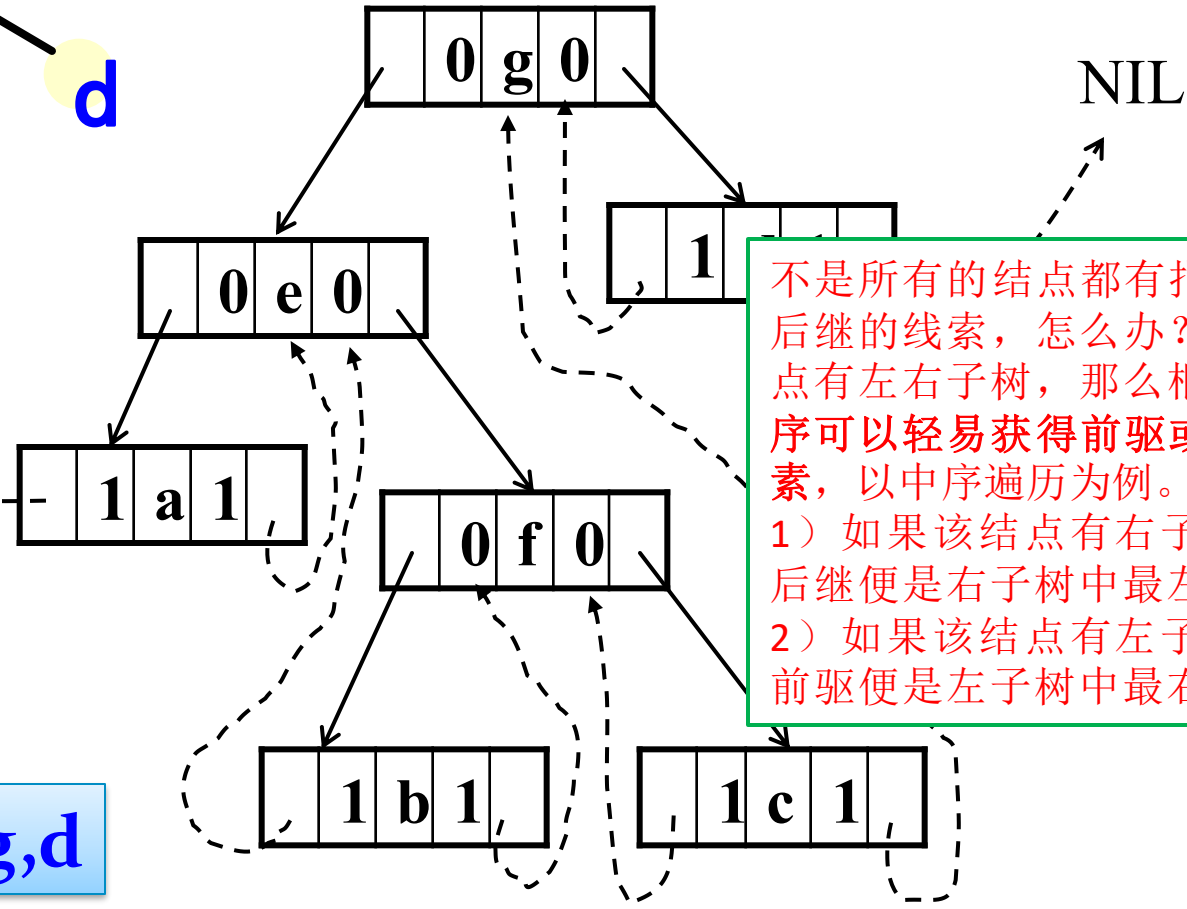
rtag= { 0 rch域指示结点的右孩子
        1 rch域指示结点的后继

5.5.1 线索二叉树的表示

- 二叉树存储结构--线索链表

遍历指向该线性序列中的“前驱”和“后继”的指针，称作“线索”。





不是所有的结点都有指向前驱和后继的线索，怎么办？如果该结点有左右子树，那么根据**遍历次序可以轻易获得前驱或者后继元素**，以中序遍历为例。

1）如果该结点有右子树，那么后继便是右子树中最左下的结点；

2）如果该结点有左子树，那么前驱便是左子树中最右下的结点。

中序 a,e,b,f,c,g,d

线索链表

5.5.1 线索二叉树的表示

- 线索二叉树的相关概念

➢线索链表：以上述结点结构构成的二叉链表作为二叉树的存储结构，叫线索链表。

➢线索：指向前驱和后继的指针。

➢线索二叉树：采用双向链接结构表示的二叉树。

➢线索化：对二叉树以某种次序遍历使其变为线索二叉树的过程。

5.5 线索二叉树

5.5.1 线索二叉树的表示

5.5.2 二叉树的线索化

5.5.3 线索二叉树的遍历

5.5.2 二叉树的线索化

- 关于线索二叉链表的头结点的设定方法

线性链表:从单向链表→双向链表

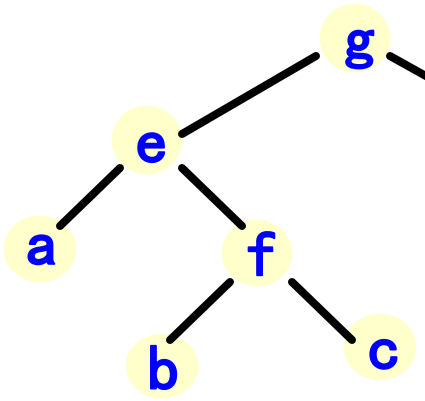
二叉树：二叉链表→线索二叉链表

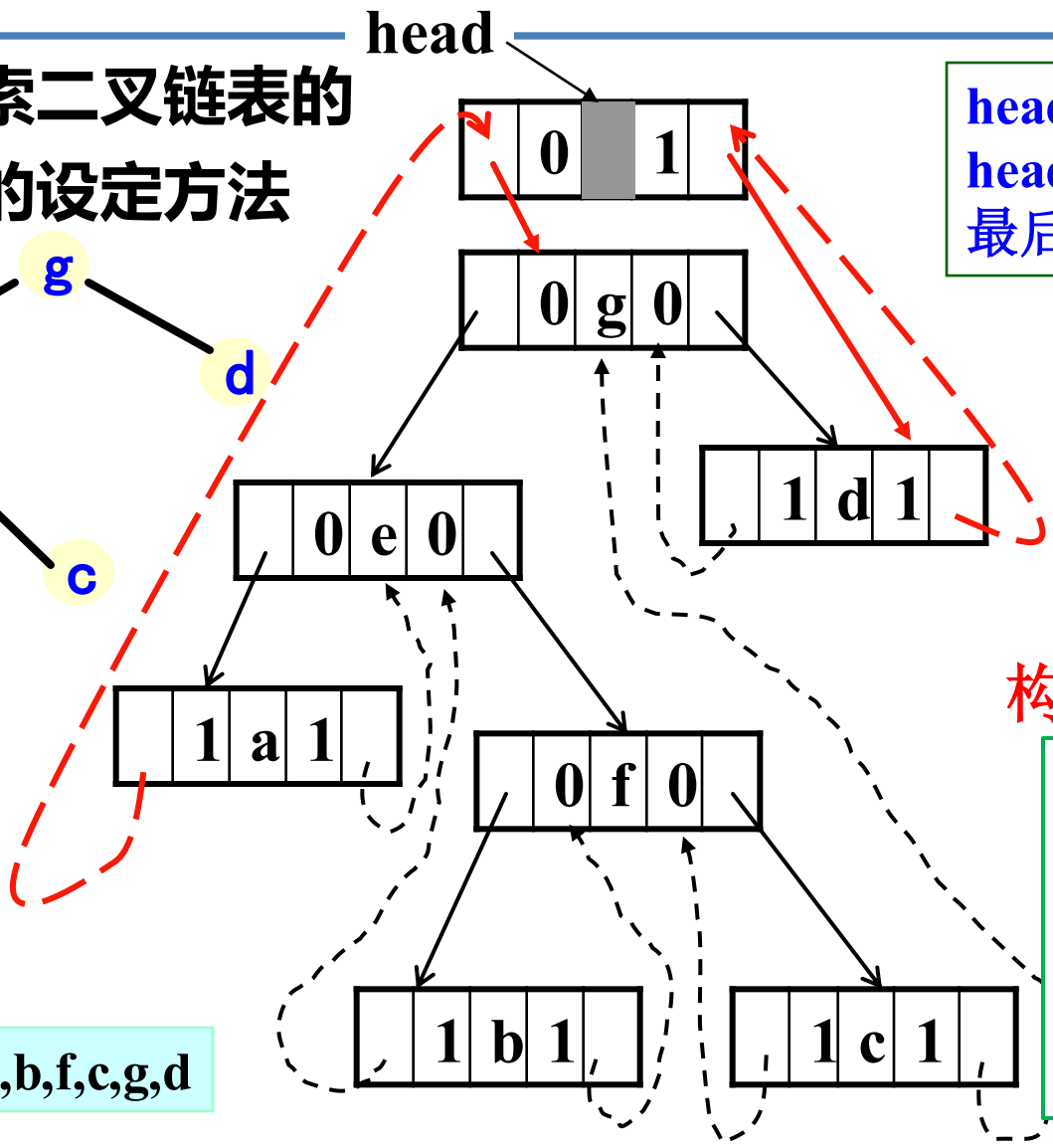
如何设定头结点？

线索二叉树的头结点：在某种遍历下的第一个结点的前驱线索和最后一个结点的后继线索均指向头结点。

5.5.2 二叉树的线索化

- 关于线索二叉链表的头结点的设定方法





head ->lch指向根结点
head ->rch指向中序最后一个结点

中序遍历的第一个结点和最后一个结点都指向头结点

构成双向线索链表

既可以从第一个结点顺着后继结点进行遍历，也可以从最后一个结点顺着前驱结点进行遍历，而通过头结点可以方便的找到第一个结点和最后一个结点

中序序列 a,e,b,f,c,g,d

5.5.2 二叉树的线索化

- 查找---在线索树中找结点---中序后继

在中序线索树中查找中序后继的方法

（1）当结点没有右子树时，即：

当p->rtag =1 时，p->rch 既为所求后继结点（线索）。

（2）当结点有右子树时，即：

当p->rtag = 0 时，p的后继结点为p的右子树的最左下结点。

如何在中序线索树找指定结点的后继？

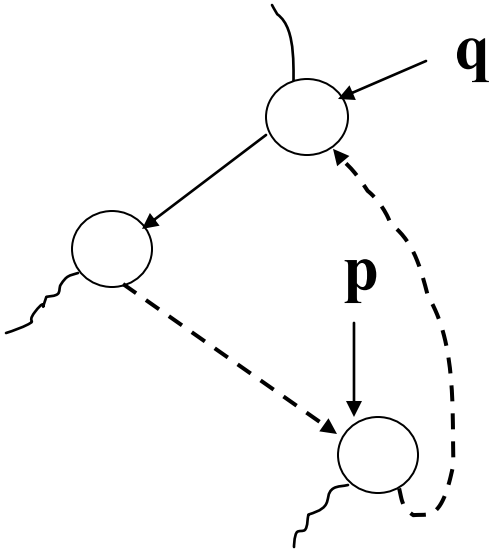
5.5.2 二叉树的线索化

- 在中序线索树中找结点*p的中序后继

```
threadbithptr * Inordernext(bithptr *p)
{
  threadbithptr *q;
  if (p->rtag==1)  // 右子树空
     return(p->rch);
  else     // 右子树非空

    ?

}
```



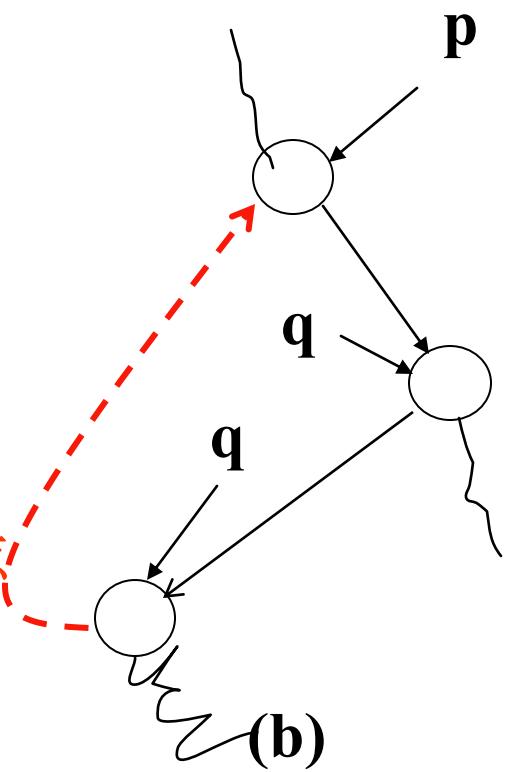
(a)

p的右子树空

5.5.2 二叉树的线索化

- 在中序线索树中找结点*p的中序后继

```
threadbithptr * Inordernext(bithptr *p)
{
  threadbithptr *q;
  if (p->rtag==1)  // 右子树空
      return(p->rch);
  else              // *p右子树非空
     {
        q=p->rch;  //从*p的右子树开始找
        while(q->ltag==0) // 找右子树的“最左下”结点
            q=q->lch;
        return(q);
     }
}
```



p的右子树不空

5.5.2 二叉树的线索化

- 查找--在线索二叉树树中找结点--中序前驱

在线索二叉树中查找中序前驱的方法

（1）当结点没有左子树时，即：

当p->ltag =1 时， p->lch 既为所求前驱结点（线索）。

（2）当结点有左子树时，即：

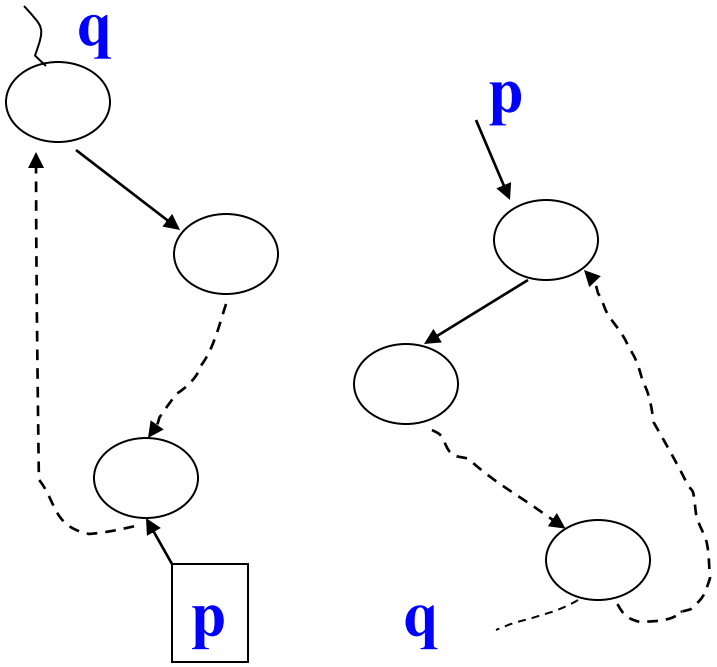
当p->ltag = 0 时， p的前驱结点为p的左子树的最右下结点。

5.5.2 二叉树的线索化

- 在中序线索树中找结点*p的中序前驱

分析：(1) 当p->ltag = 1时，p->lch 既为所求（线索）。

(2) 当p->ltag = 0时，q为p的左子树的最右下结点。



p的左孩子为空　p的左孩子不空

```
threadbithptr inpre(threadbithptr p)
threadbithptr p ;
{   threadbithptr q ;
    q=p->lch ;
    if (p->ltag = = 0 )
          while(q->rtag = = 0)
            q = q->rch ;
    return q;
}
```

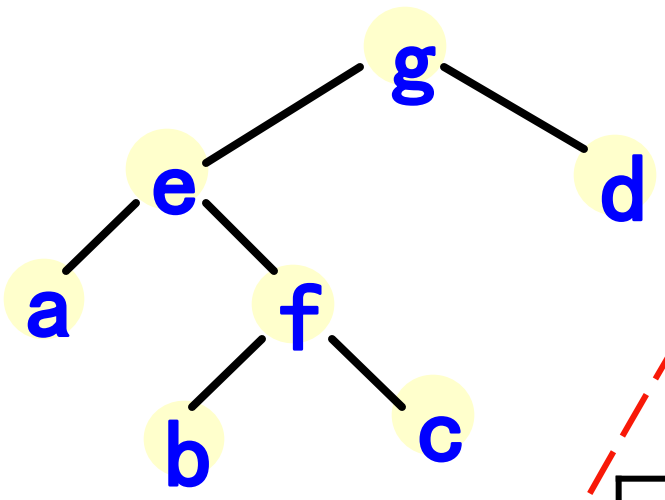
5.5 线索二叉树

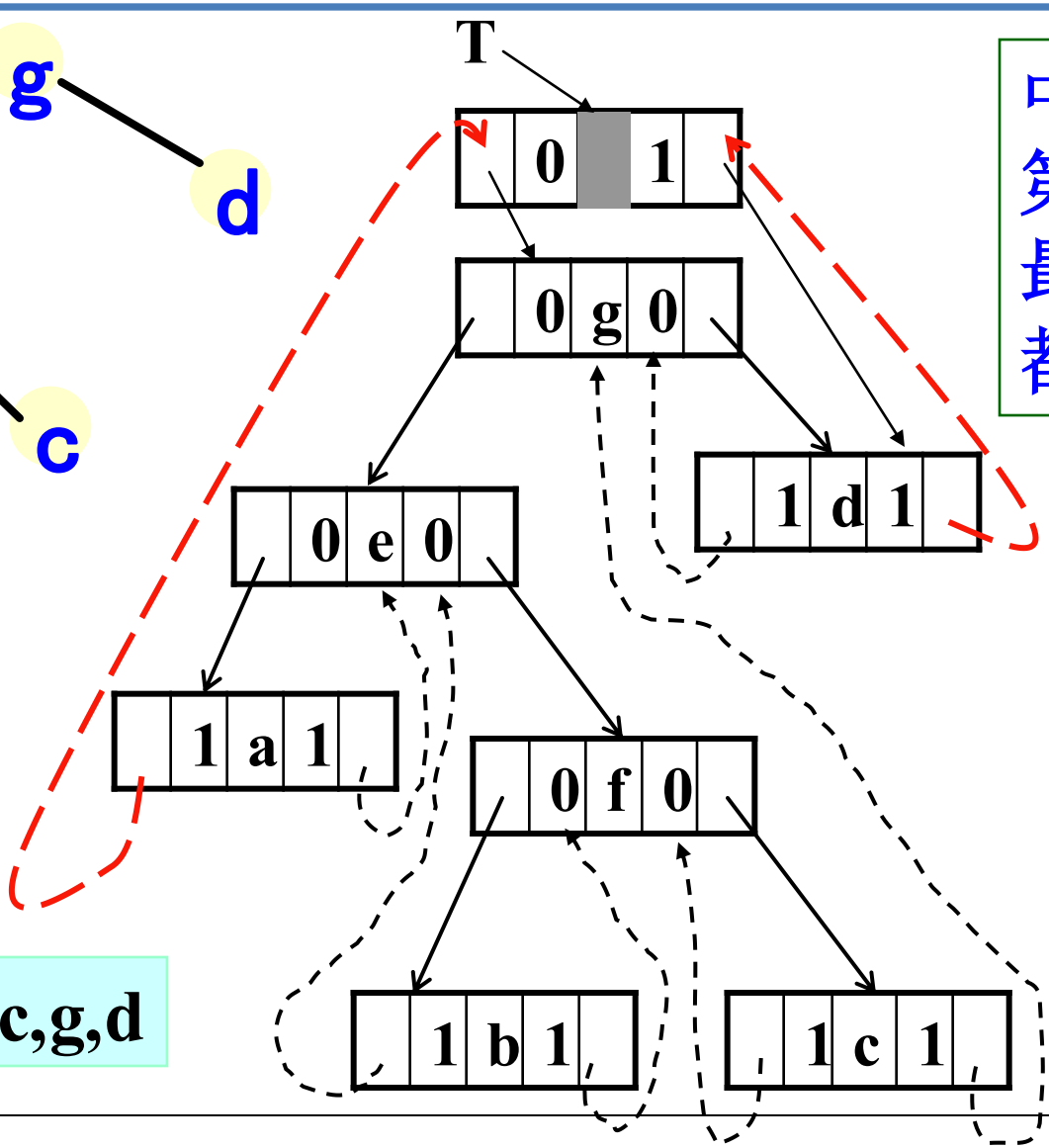
5.5.1 线索二叉树的表示

5.5.2 二叉树的线索化

5.5.3 线索二叉树的遍历

5.5.3 线索二叉树的遍历





中序遍历的第一个结点和最后一个结点都指向根结点

中序 a,e,b,f,c,g,d

线索链表

5.5.3 线索二叉树的遍历

- 遍历中序线索二叉树的解法一：基于中序后继查找算法

```
TraverseInthread(threadbithptr *p)
{
    if (p!=Null)
    {
        while (p->ltag==0)//找中序序列的开始结点
            p=p->lch;
        do
        {   visit(p->data)
            p=Inordernext(p);//找*p的中序后继结点
        } while (p!=Null);
    }
}
```

5.5.3 线索二叉树的遍历

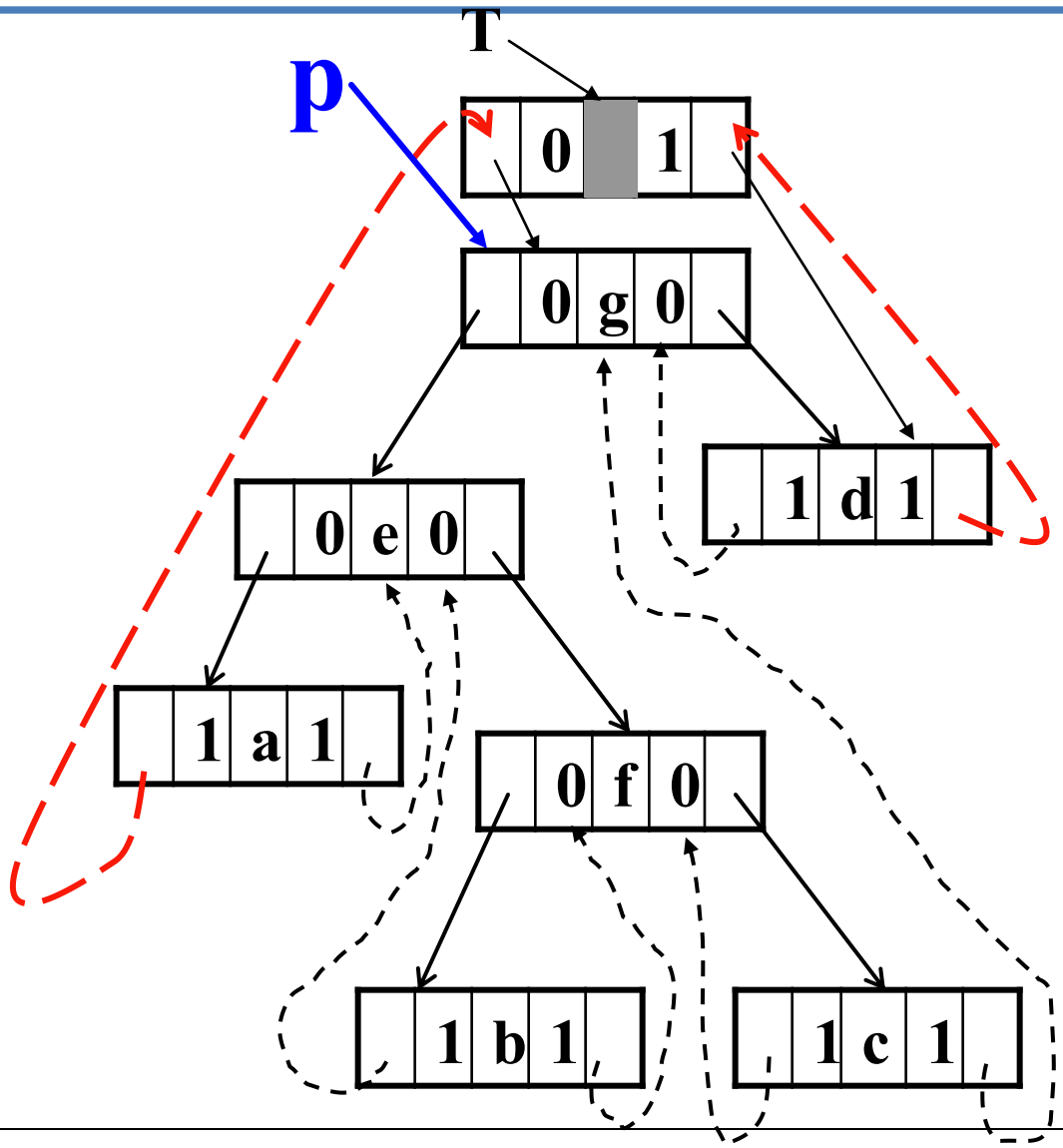
- 遍历中序线索二叉树解法二

```
Status InthreadTraverse (BiThrTree T，Status(*visit)){
//中序遍历线索二叉树T的非递归算法
T指向线索二叉树的头结点，头结点的lch指向根结点，中序的最后一个结点
   指向头结点。每次判定结点的ltag和rtag
  p=T->lch;                          //p指向二叉树真正的根结点
  while(p!=T){                       //空树或遍历结束时，p=T




                                     // 重复以上过程，直到有线索指向头结点为止
   }   return(ok);
  }
```

5.5.3 线索二叉树的遍历



线索链表

5.5.3 线索二叉树的遍历

```
Status InthreadTraverse (BiThrTree T，Status(*visit)){
//中序遍历线索二叉树T解法二
T指向线索二叉树的头结点，头结点的lch指向根结点，中序的最后一个结点指向头结点。每次判定结点的ltag和rtag
    p=T->lch;                                  //p指向二叉树真正的根结点
    while(p!=T) //空树或遍历结束时，p=T
    {
        while (p->ltag==0）     p=p->lch;   //找中序序列的开始结点
        visit(p->data);                        //访问图中的结点a
        while (p->rtag==1&&p->rch!=T ）//p不是中序遍历的最后一个结点
        {
            p=p->rch;                          //右线索指向后继结点
            visit(p->data);
        }
        p=p->rch;                              //p存在右子树
    }                                          // 重复以上过程，直到有线索指向头结点为止
    return(ok);
}
```

5.5.3 线索二叉树的遍历

基于线索二叉树进行遍历，时间复杂性与普通二叉树都是O(n)，关键不需要设栈，所以适用于经常需要遍历，或者需要获得遍历序列中的前驱和后继元素。

5.5.3 线索二叉树的遍历

- 中序线索化二叉树

线索化的实质：

是将二叉链表中的空指针改为指向前驱或后继的线索，而前驱或后继信息只有在遍历时才能得到，因此线索化的过程即为在遍历过程中修改空指针的过程。

5.5.3 线索二叉树的遍历

- 中序线索化二叉树

如何建立线索链表？

1. 在中序遍历过程中修改结点的左、右指针域
2. 保存当前访问结点的“前驱”和“后继”信息。
3. 遍历过程中，附设指针pre，pre指向刚访问过的结点。P指向当前访问结点。即pre是p的前驱

类似线性表的两个相邻的结点，修改相应的pre域和next域

5.5.3 线索二叉树的遍历

- 构建中序线索化二叉树//有头结点

```
Status InOrderThreading(BiThrTree &head, BiThrTree T) {
  // 中序遍历二叉树，并将其线索化，head指向头结点
  if (!(head = (BiThrTree)malloc(sizeof( BiThrNode))))
        exit(OVERFLOW);
  head->ltag = 0;
  head->rtag =1;
  head->rchild = head;       // 添加头结点，初始化指向自身
```

5.5.3 线索二叉树的遍历

- 中序线索化二叉树

```
if (!T)   head->lchild = head; //二叉树空，指向自身
else {  // 修改头结点和遍历的最后一个结点
        head->lchild = T;   pre = head; //初始pre指向头结点head
        InThreading(T);                 //对二叉树进行中序线索化
        pre->rtag = 1;
        pre->rchild = head;
        head->rchild = pre;     //修改头结点和最后一个结点
                    //头结点head与最后一个结点pre 互链接
    }
return OK;
} // InOrderThreading
```

InThreading()怎么定义？

5.5.3 线索二叉树的遍历

- 中序线索化二叉树

```
void InThreading(BiThrTree p) {
  if (p) {          // 对非空二叉树进行中序线索化
    InThreading(p->lchild);              // 左子树线索化
    if (!p->lchild)                      // p结点无左子树，建前驱线索
    { p->ltag = 1;   p->lchild = pre; }
    if (!pre->rchild)                    //pre结点无右子树， 建后继线索
    { pre->rtag = 1;  pre->rchild = p; }
    pre = p;                             // 保持 pre 指向 p 的前驱
    InThreading(p->rchild);              // 右子树线索化
  } // if
} // InThreading
```

注意：每次对p（当前指针）建立前驱线索，而对pre建立后继线索。因为此时p的后继线索还未遍历到，还未知。

5.5.3 线索二叉树的遍历

- 在二叉树中一般不讨论结点的插入与删除，原因是其插入与删除的操作与线性表相同，所不同的是需要详细说明操作的具体要求。
- 而在线索树中结点的插入与删除则不同，因为要同时考虑修正线索的操作。

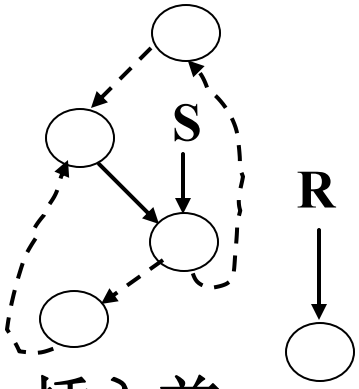
线索树的缺点：

在插入和删除时，除了修改指针外，还要相应地修改线索。

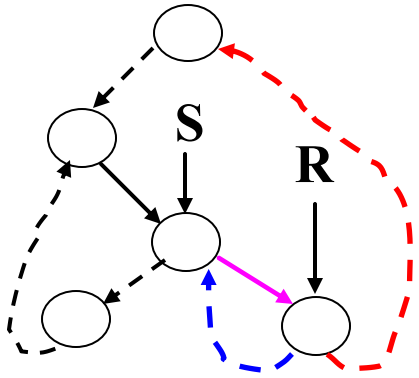
5.5.3 线索二叉树的遍历

例

将结点 R 插入作为结点 S 的右孩子结点：
(1)若S的右子树为空，则直接插入;如图所示



插入前
（S的右子树为空）



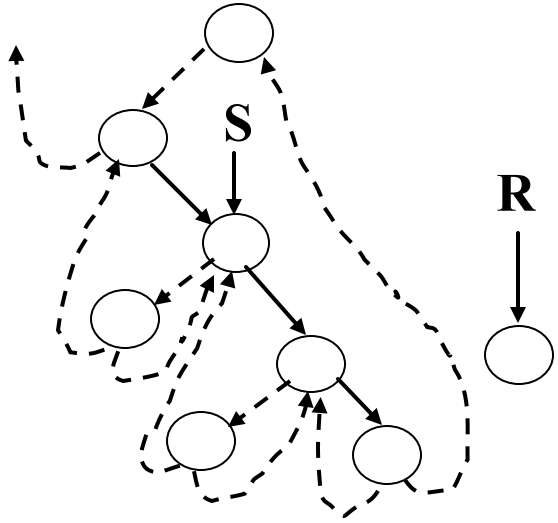
插入后

(a)

5.5.3 线索二叉树的遍历

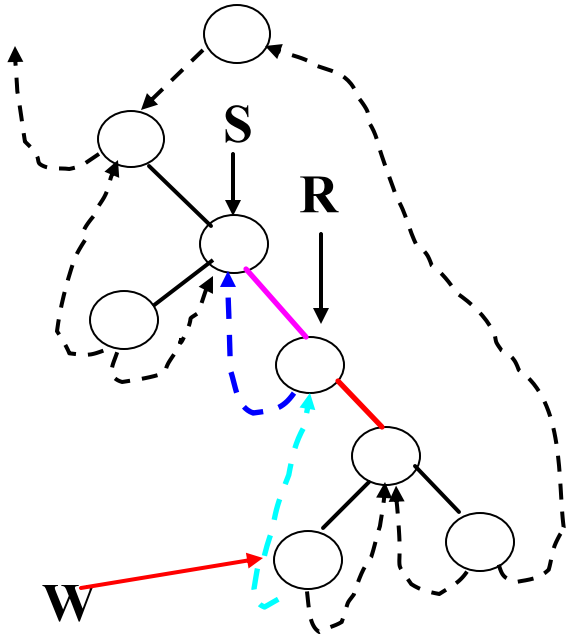
例

(2)若S的右子树非空，则R插入后原来S的右子树作为R的右子树。



插入前
（右子树非空）

(b)



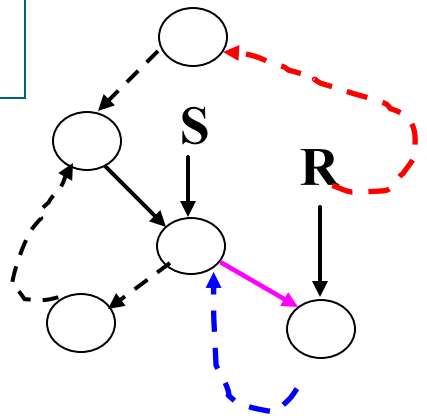
插入后

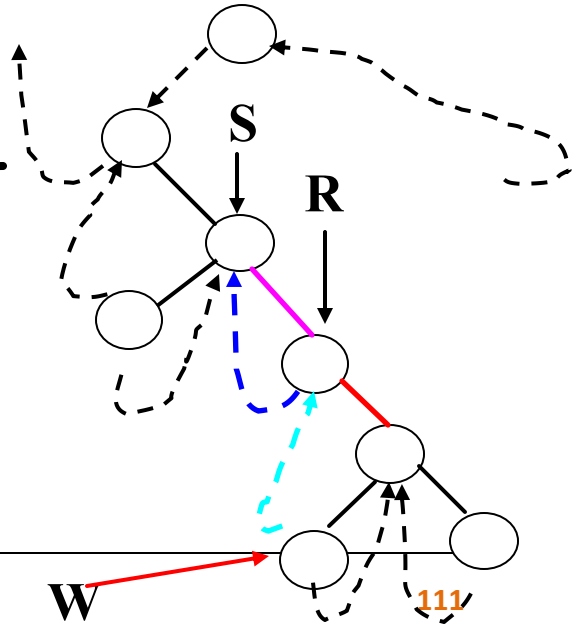
5.5.3 线索二叉树的遍历

例

中序线索二叉树的右插入算法：
将结点 R 插入作为结点 S 的右孩子

```
Void   RINSERT (threadbithptr   S , R )
{  if ( S->rtag = 1)          // 右子树空
   {   R->ltag = 1;   R->rtag = 1 ;
       R->rch = S->rch ; //R的中序后继是原来S的后继
       R->lch = S;    //R的中序前驱是S
       S->rtag = 0; //修改s的右标识
       S->rch = R;  //S的右孩子是R
   }
   else                          //右子树非空
   {  ..............
       w = Inordernext ( S ) ; //找S的中序后继
       w->lch=R  ;       //w的中序前驱是R
   }
}
```





5.5.3 线索二叉树的遍历

思考题：
线索二叉树的左插入算法:将结点 R 插入作为结点 S 的左孩子

5.5.3 线索二叉树的遍历

应用举例

按给定前缀的算式表达式建立相应的二叉树

已知算式表达式的前缀表示式　-×+ a b c / d e

用二叉树表示四则运算表达式：运算符都在内部分支结点上，而运算数在叶子结点上，被除数和被减数放在左儿子结点。

前缀式的运算规则为：

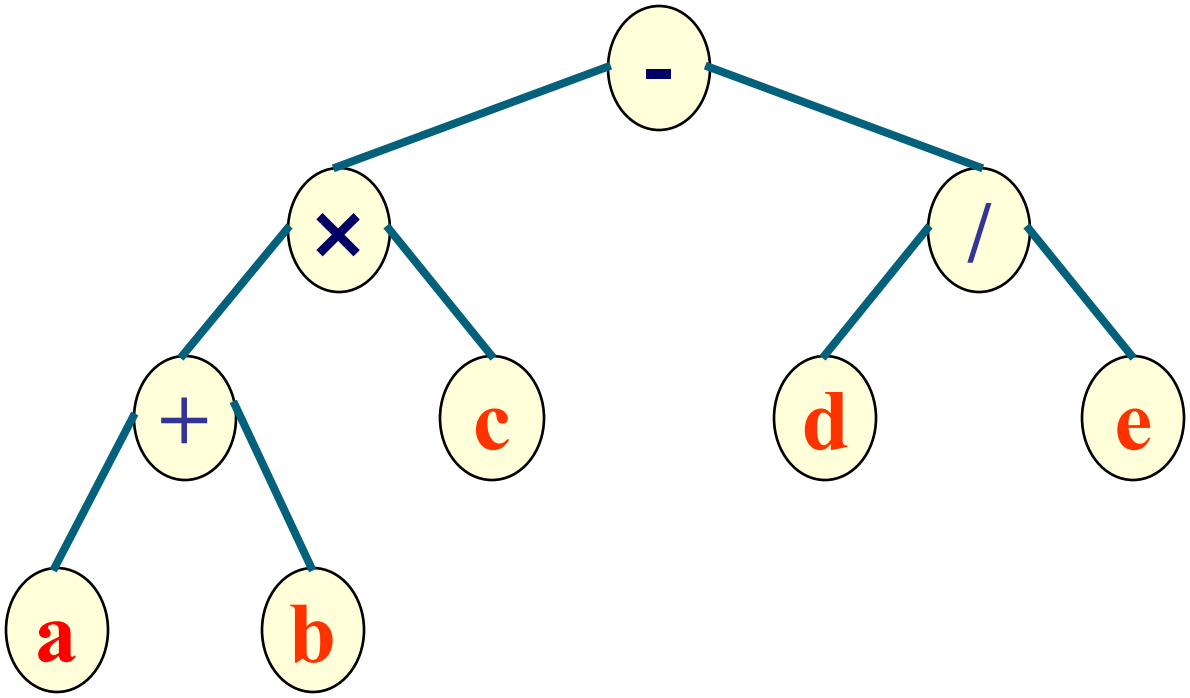
- 连续出现的两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式；
- 前缀式唯一确定了运算顺序；

5.5.3 线索二叉树的遍历

应用举例

按给定前缀的算式表达式建立相应的二叉树

已知表达式的前缀表示式　- × + a b c / d e



对于二元运算符：

- **左右子树不空；**
- **操作数为叶子结点；**
- **运算符为分支结点；**

表达式=（第一操作数）（运算符）（第二操作数）

5.5.3 线索二叉树的遍历

应用举例

按给定后缀的表达式建立相应的二叉树

后缀式的运算规则为

- 运算符在式中出现的顺序恰为表达式的运算顺序；
- 每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式。
- 先找运算符，再找操作数。
- 操作数的顺序不变。

5.5.3 线索二叉树的遍历

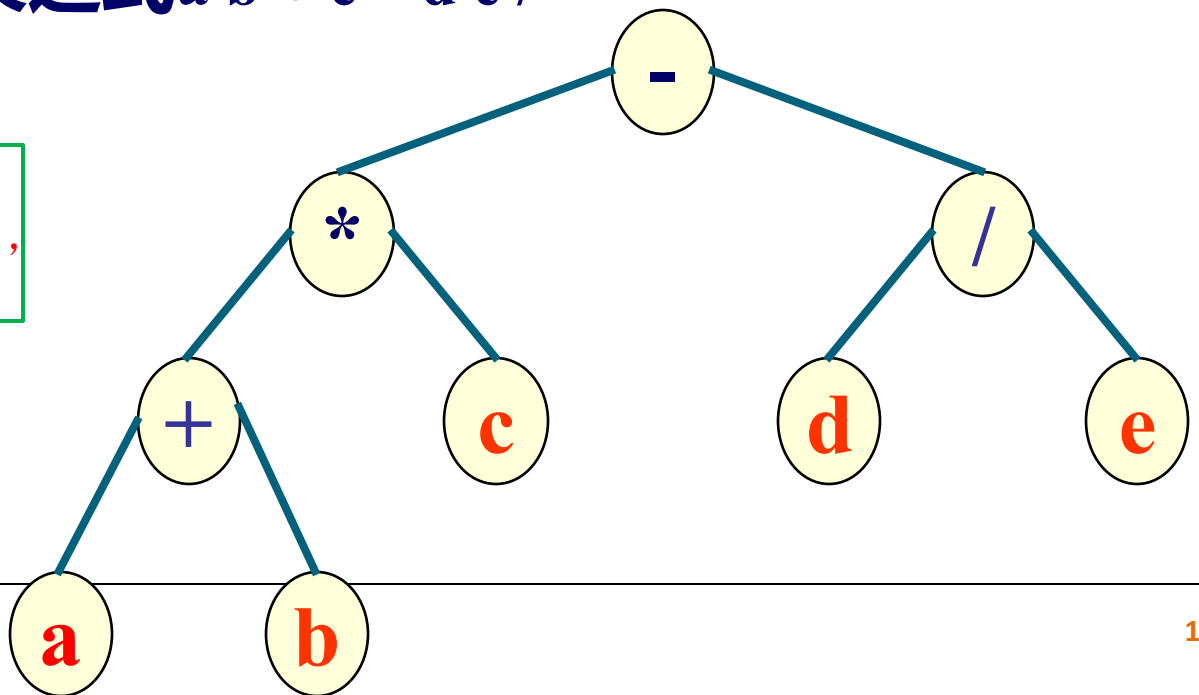
应用举例

按给定后缀的表达式建立相应的二叉树

方法：从左到右扫描后缀表达式，遇到操作符则对前面的操作数建立二叉树，以此类推。

例如：后缀表达式a b + c * d e / -

注意：中序表示法必须用括号表示运算的先后关系，否则有歧义。



研究树的目的：

使树的操作都是通过二叉树来完成的

5.6 树和森林

5.5.1 树的存储结构

5.5.2 树、森林与二叉树的转换

5.5.3 树和森林的遍历

5.5.4 huffman树及其应用

5.6 树和森林

5.5.1 树的存储结构

5.5.2 树、森林与二叉树的转换

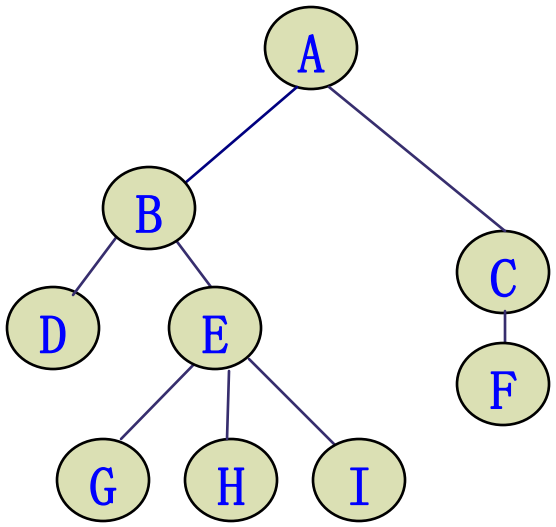
5.5.3 树和森林的遍历

5.5.4 huffman树及其应用

5.5.1 树的存储结构

- 双亲表示法

采用一组连续空间存储树的结点，通过保存每个结点的双亲结点的位置，表示树中结点之间的结构关系。



结点数

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

5.5.1 树的存储结构

- 双亲表示法

树结构:

data	parent

双亲表示类型定义

```
#define MAX 100
Typedef struct PTnode{//结点结构
    Elem data;
    int  parent;   //双亲位置域
 } PTnode
```

5.5.1 树的存储结构

- 双亲表示法

树结构:

```
typedef struct {
    PTNode  nodes[MAX_TREE_SIZE];
    int    r, n;  //  根结点的位置和结点个数
} PTree;
```

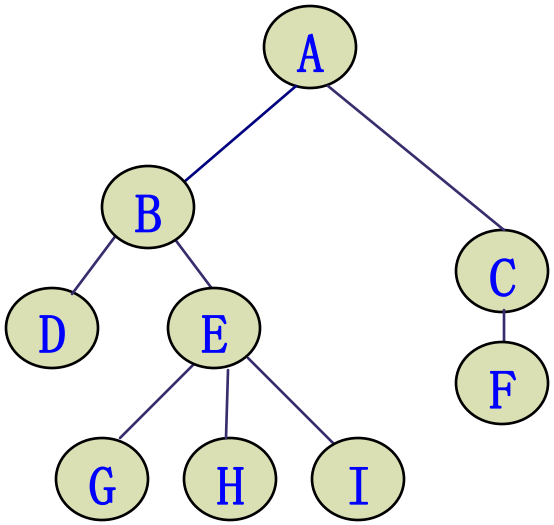
双亲表示法可以在常数时间找到双亲结点，反复调用，可以找到树根结点。

该方法求结点的孩子结点需要遍历整个数据结构。

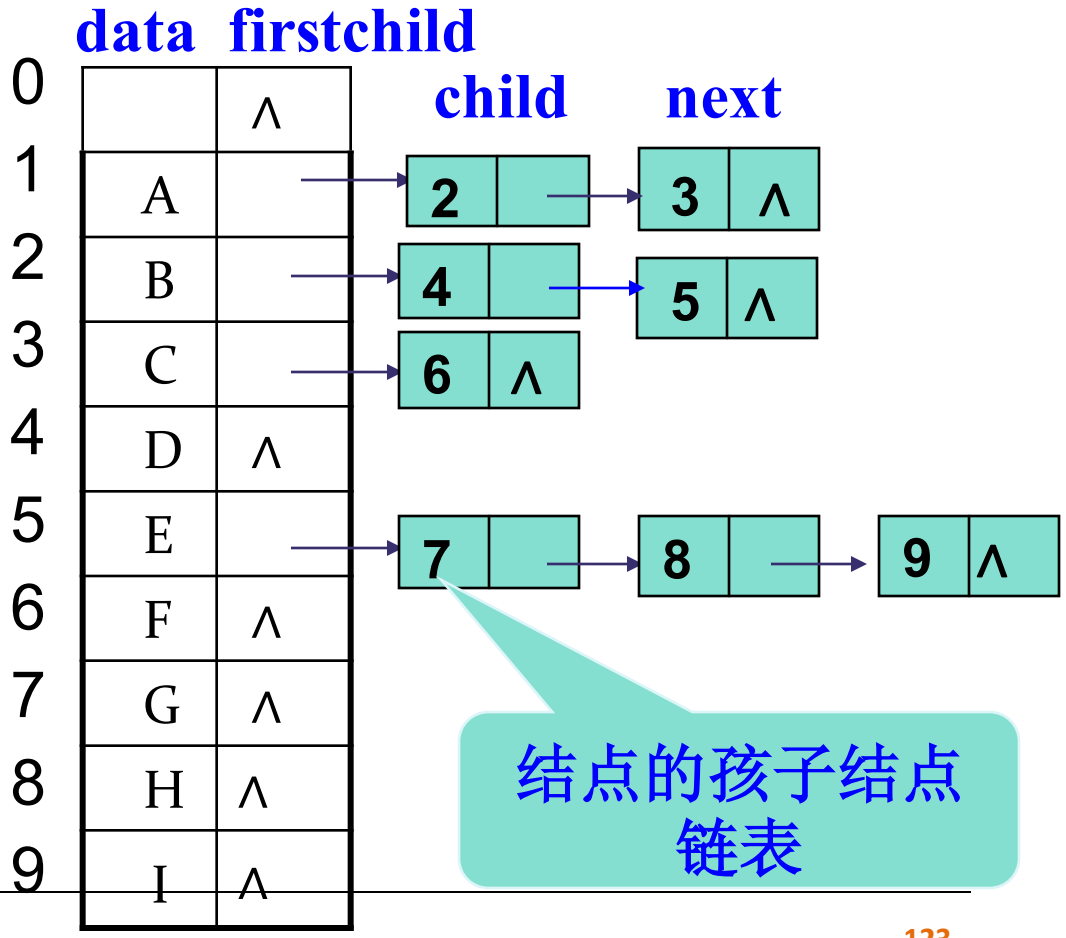
5.5.1 树的存储结构

- 孩子链表表示法

对树的每个结点用线性链表存贮它的孩子结点



树的孩子链表图示



孩子链表表示法示意图

5.5.1 树的存储结构

- 孩子链表表示法

对树的每个结点用线性链表存贮它的孩子结点

孩子结点结构:

child	next

双亲结点结构:

data	firstchild

```
typedef struct CTNode {
    int     child;
    struct CTNode *next;
  } *ChildPtr;
```

```
typedef struct {
  Elem    data;
  ChildPtr  firstchild;
          // 孩子链的头指针
} CTBox;
```

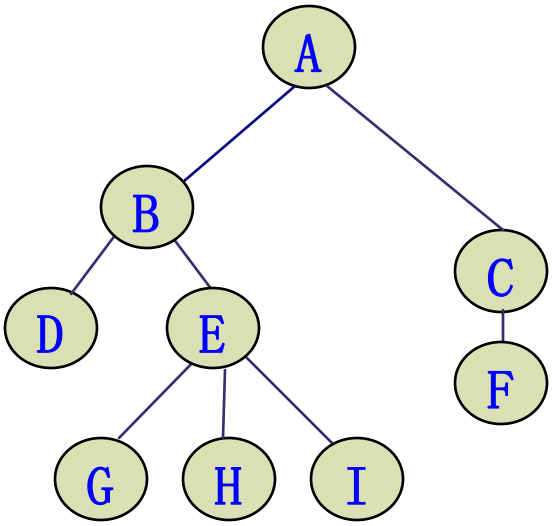
```
typedef struct {
   CTBox node[MAX_TREE_SIZE];
   int n, r; // 结点数和根的位置
} CTree;
```

找一个结点的孩子十分方便，但要找一个结点的双亲则要遍历整个结构

5.5.1 树的存储结构

- 双亲孩子表示法

结合双亲表示法和孩子表示法



r=0
n=9

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

结点的孩子结点链表

带双亲孩子链表示意图

5.5.1 树的存储结构

- 树的二叉链表---孩子兄弟表示法

如何把树与二叉树联系起来？

5.5.1 树的存储结构

- 树的二叉链表---孩子兄弟表示法

用二叉链表作为树的存贮结构。链表的两个指针域分别指向该结点的第一个孩子结点和右边第一个兄弟结点。

结点结构：

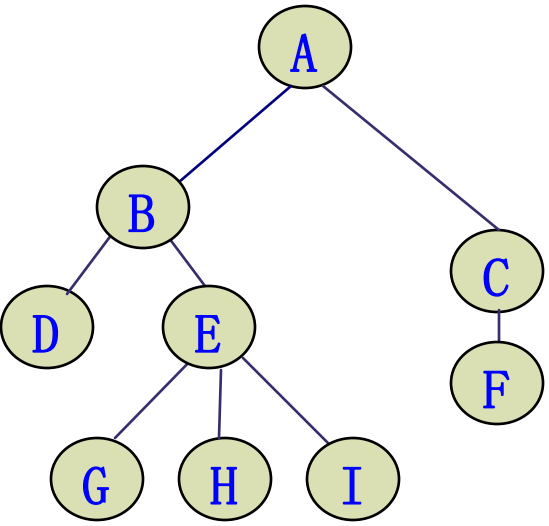

```
typedef struct CSNode{
    Elem  data;
    struct CSNode  *firstchild, *nextsibling;
} CSNode, *CSTree;
```

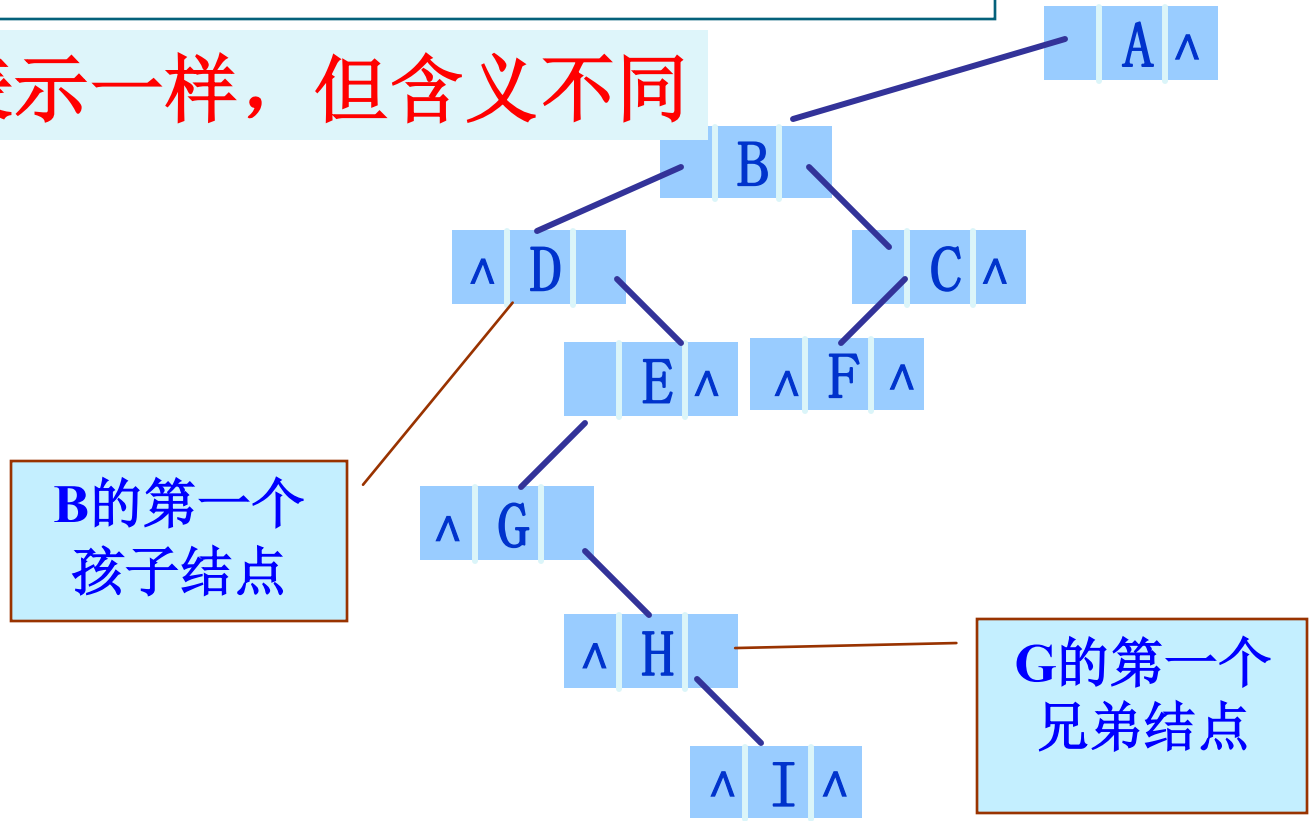
5.5.1 树的存储结构

- 树的二叉链表---孩子兄弟表示法

用二叉链表作为树的存贮结构。链表的两个指针域分别指向该结点的第一个孩子结点和右边第一个兄弟结点。

与二叉树的存储表示一样，但含义不同

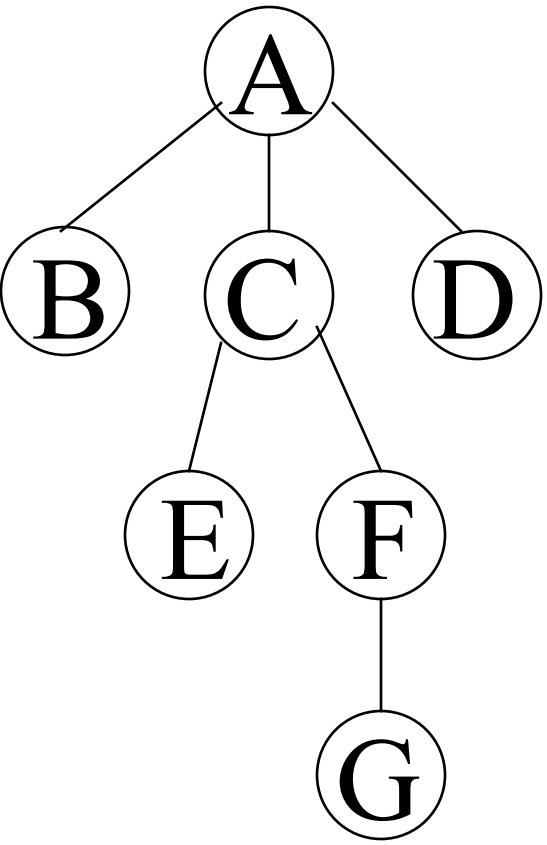


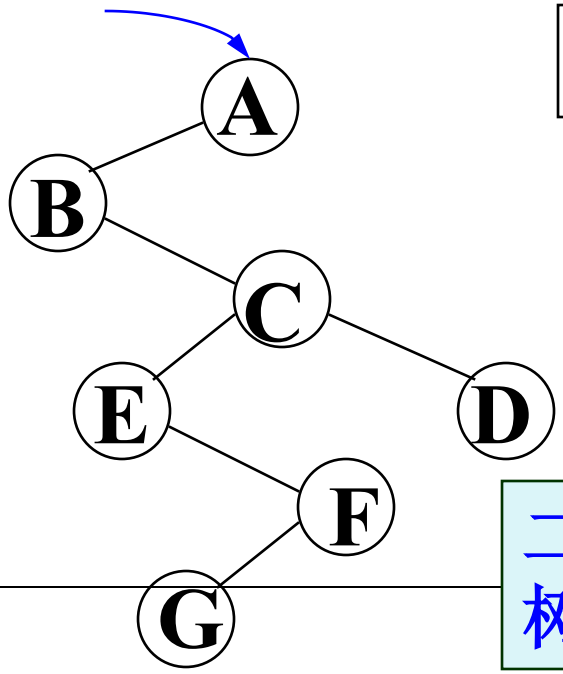


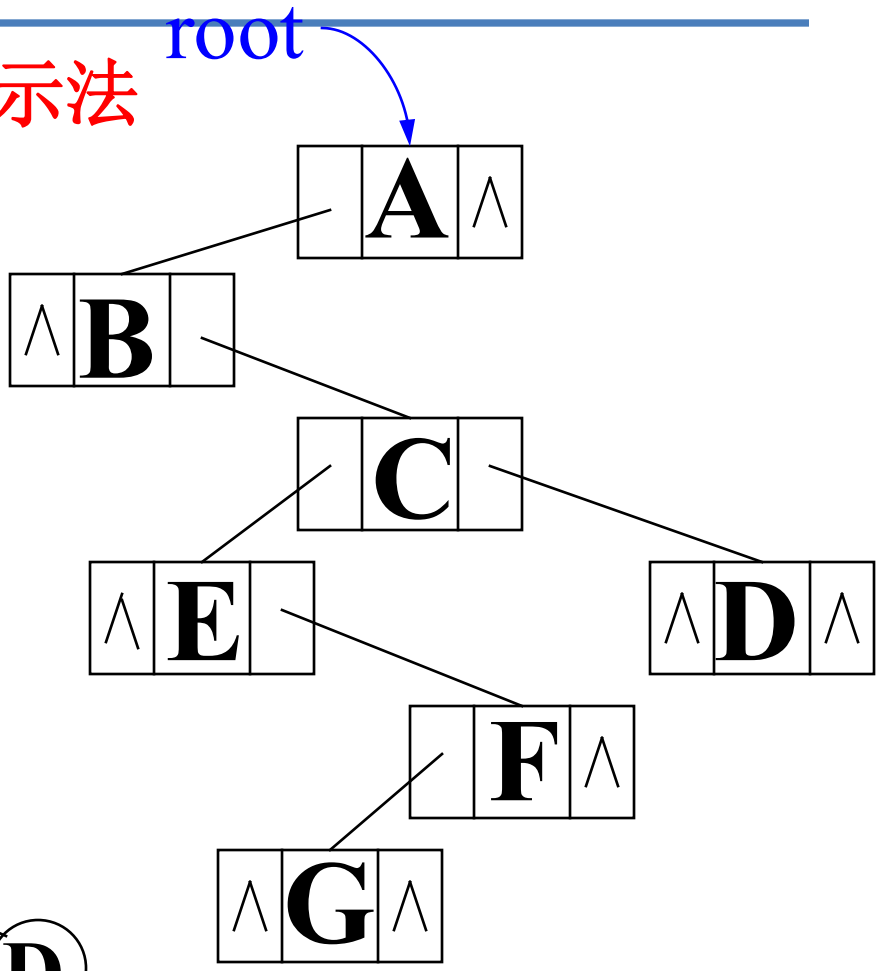
树的孩子兄弟表示法图示

5.5.1 树的存储结构

- 树的二叉链表---孩子兄弟表示法







二叉树：左、右子树
树： 左是孩子，右是兄弟

5.5.1 树的存储结构

- 树的二叉链表---孩子兄弟表示法

树和二叉树的存储表示方式是一样的，只是左右孩子表达的逻辑关系不同：

- 二叉树：左右孩子；
- 树的二叉链表：第一个孩子结点和右边第一个兄弟结点。

把树和二叉树通过二叉链表对应起来

如何把树转化成二叉树？

5.6 树和森林

5.5.1 树的存储结构

5.5.2 树、森林与二叉树的转换

5.5.3 树和森林的遍历

5.5.4 huffman树及其应用

5.5.2 树、森林与二叉树的转换

- 树转换为二叉树的方法

树转换为二叉树的方法

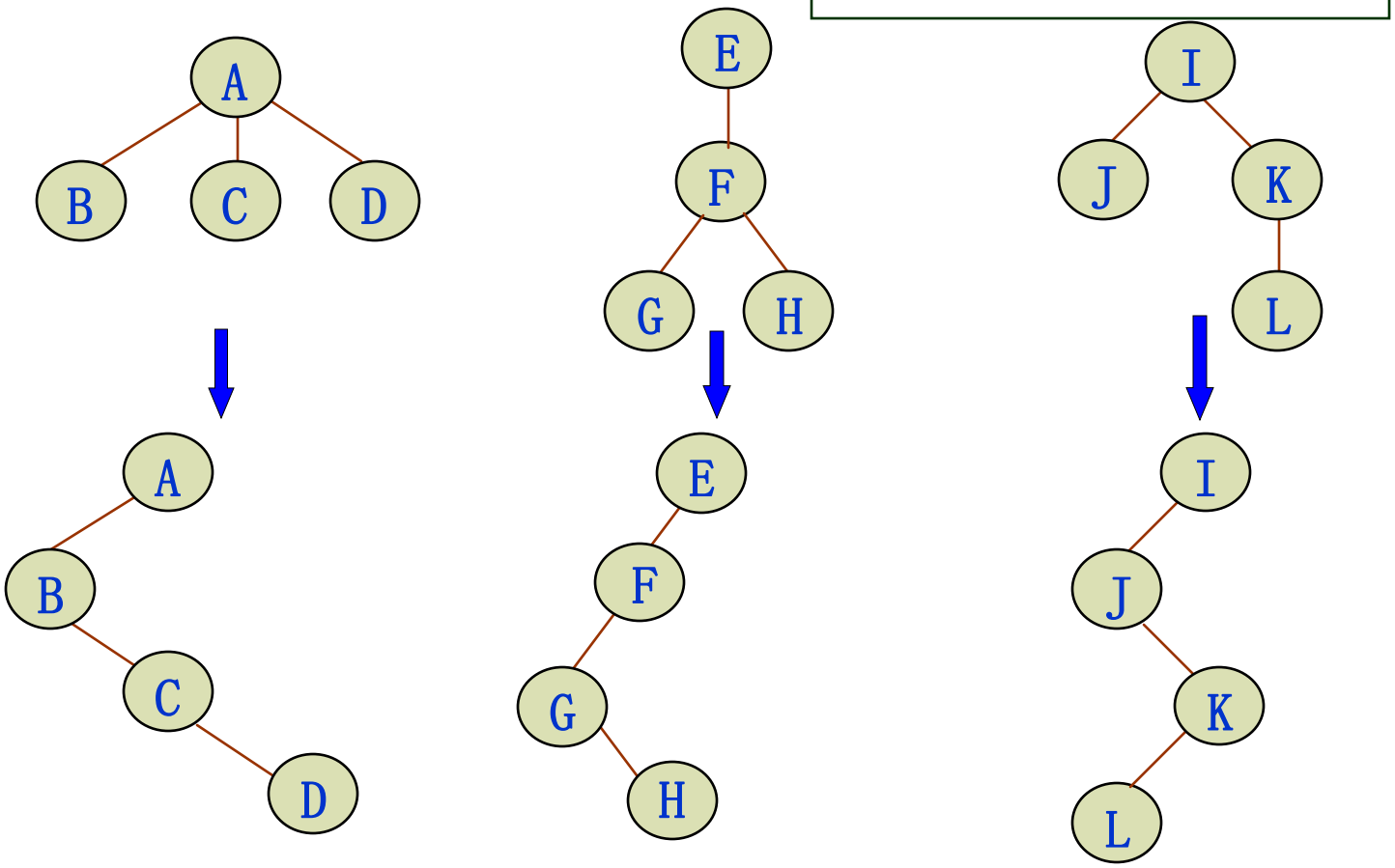
（1）在所有兄弟结点之间加一条连线；

（2）对每个结点，除了保留与其长子的连线外，去掉该结点与其它孩子的连线；

5.5.2 树、森林与二叉树的转换

例 将树转换为二叉树

把树和二叉树通过二叉链表对应起来



每棵树对应的二叉树

5.5.2 树、森林与二叉树的转换

- 森林转换为二叉树的方法

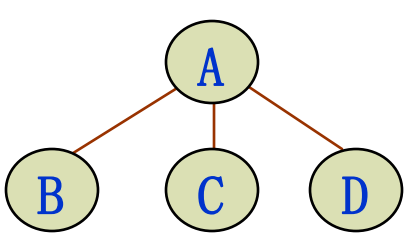
（1）将森林中的每一树转换二叉树；

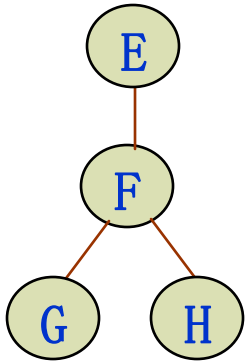
（2）将各二叉树的根结点视为兄弟连在一起。

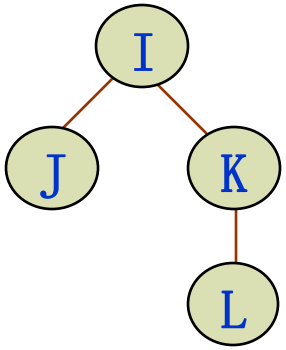
5.5.2 树、森林与二叉树的转换

- 森林转换为二叉树的方法

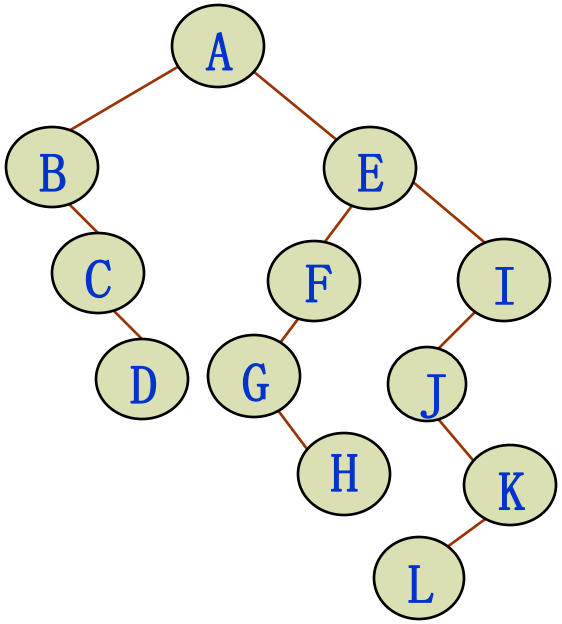
例 森林转换为二叉树



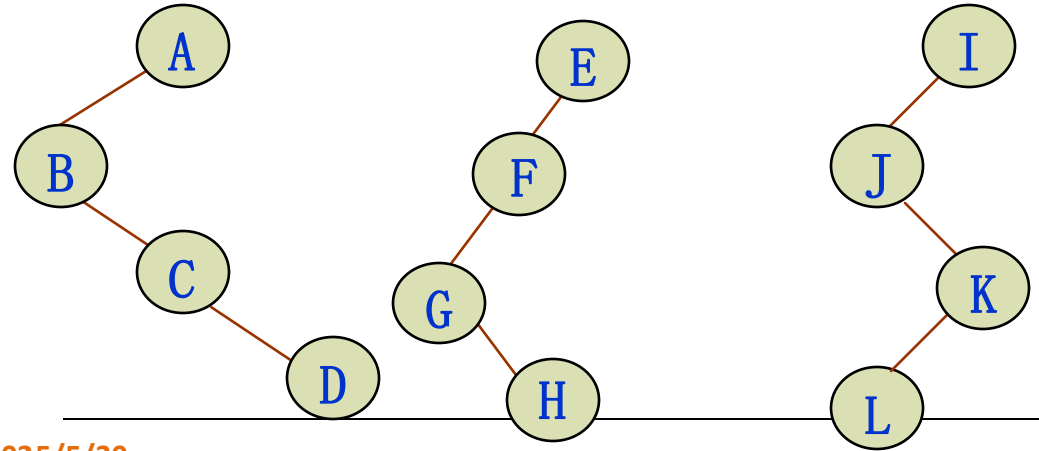




包含3棵树的森林



森林对应的二叉树



每棵树对应的二叉树

5.5.2 树、森林与二叉树的转换

- 森林和二叉树的对应关系

设森林 $F = (T_1, T_2, \ldots, T_n)$;

$T_1 = (root, t_{11}, t_{12}, \ldots, t_{1m})$;

二叉树 $B = (Node(root), LBT, RBT)$;

由森林转换成二叉树的转换规则为:

若 $F = \Phi$，则 $B = \Phi$;

否则，

由 $ROOT(T_1)$ 对应得到 $Node(root)$;

由 $(t_{11}, t_{12}, \ldots, t_{1m})$ 对应得到 LBT;

由 $(T_2, T_3, \ldots, T_n)$ 对应得到 RBT。

5.5.2 树、森林与二叉树的转换

- 森林和二叉树的对应关系

由二叉树转换为森林的转换规则为：

若 $B = \Phi$，则 $F = \Phi$；

否则，

由 Node(root) 对应得到 $ROOT(T_1)$；

由LBT 对应得到 $(t_{11}, t_{12}, \cdots, t_{1m})$；

由RBT 对应得到 $(T_2, T_3, \cdots, T_n)$。

5.5.2 树、森林与二叉树的转换

- 二叉树到树、森林的转换

（1）如果结点X是其双亲Y的左孩子，则把X的右孩子，右孩子的右孩子，…，都与Y用连线连起来；

（2）去掉所有双亲到右孩子的连线

根据上述规则可以写出相互转化的递归算法

5.5.2 树、森林与二叉树的转换

- 树或森林与二叉树之间有一个自然的一一对应的关系。
- 任何一个森林或树可以唯一地对应到一棵二叉树；
- 任何一棵二叉树可以唯一地对应到一个森林或一棵树

任何一棵与树对应的二叉树，其右子树必为空

5.6 树和森林

5.5.1 树的存储结构

5.5.2 树、森林与二叉树的转换

5.5.3 树和森林的遍历

5.5.4 huffman树及其应用

5.5.3 树和森林的遍历

- 与二叉树的遍历类似，树的遍历也是从根结点出发，对树中各个结点访问一次且仅进行一次。
- 由于一个结点可以有两棵以上的子树，因此一般不讨论中根（序）遍历。
- 对树进行遍历可有两条搜索路径：
- （1）从左到右
- （2）按层次从上到下。
- 树的按层次遍历类似于二叉树的按层次遍历；

5.5.3 树和森林的遍历

- 树的先根（序）遍历

先根顺序

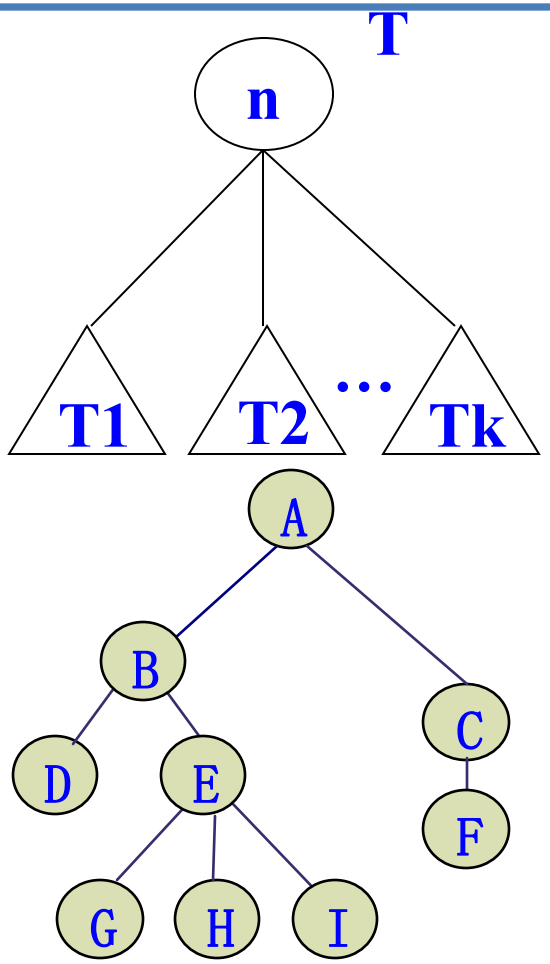
访问根结点；

先根顺序遍历T_1；

先根顺序遍历T_2；

…

先根顺序遍历T_k；



先根遍历次序:
A, B, D, E, G, H, I, C, F

5.5.3 树和森林的遍历

- 树的先根遍历

树的先根遍历递归该算法如下：

```
void preordertre(CSnode  * root) // root 一般树的根结点
   { if(root!=NULL)
        {  visit(root->data); /* 访问根结点 */
           preordertre( root-> firstchild);
           preordertre( root-> nextsibling);
        }
   }
```

5.5.3 树和森林的遍历

- 树的后根遍历

后根顺序

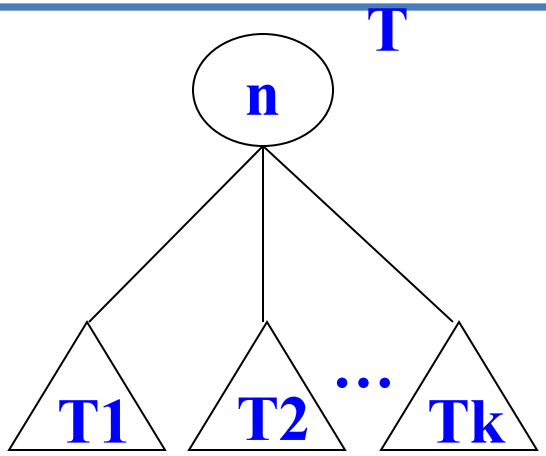
后根顺序遍历T_1；

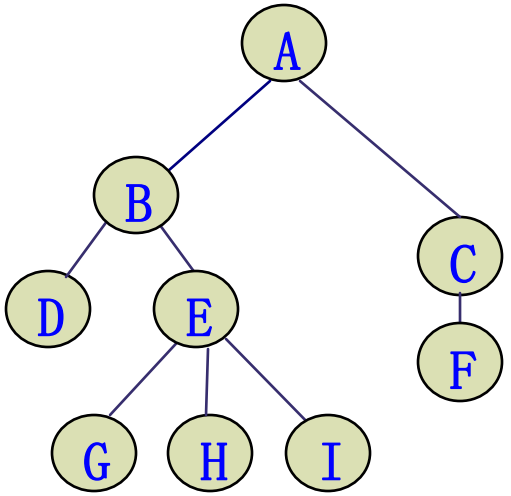
后根顺序遍历T_2；

…

后根顺序遍历T_k；

访问根结点；

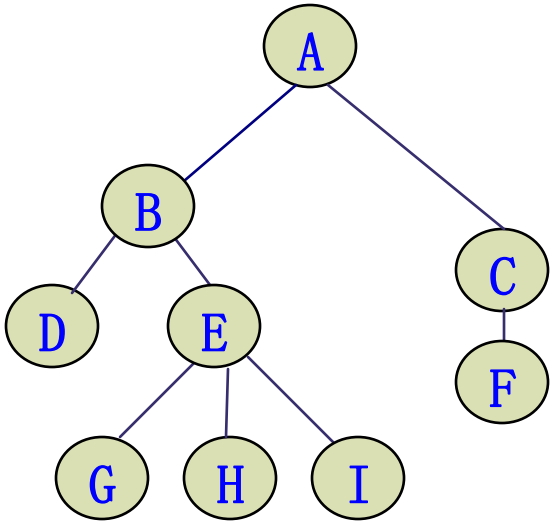


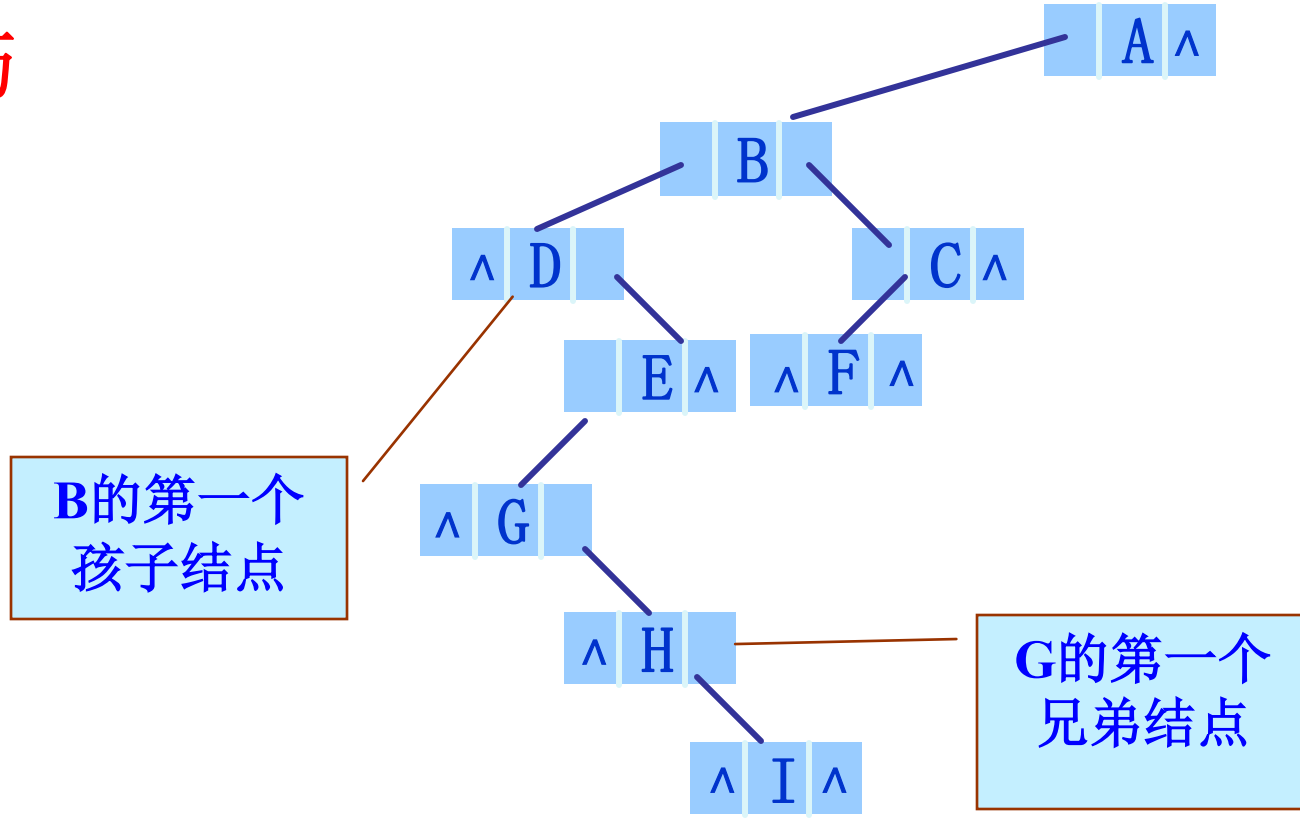


后根遍历次序:
D, G, H, I, E, B, F, C, A

5.5.1 树的存储结构

- 树的后根遍历





树的孩子兄弟表示法图示

后根遍历次序:
D, G, H, I, E, B, F, C, A

<=>

由二叉链表对应的二叉树的中序遍历次序:
D, G, H, I, E, B, F, C, A

5.5.3 树和森林的遍历

- 树的后根遍历

树的后根遍历递归算法：

```
void postordertre(CSnode * root) // root 一般树的根结点
{  if (root!=NULL )
    {
        postordertre( root-> firstchild);
        visit(root->data);
        postordertre( root-> nextsibling);
}
 }
```

大家注意：这里是用基于二叉链表的孩子兄弟表示法，所以树的后根遍历对应的是二叉链表所对应的二叉树的中序遍历。

5.5.3 树和森林的遍历

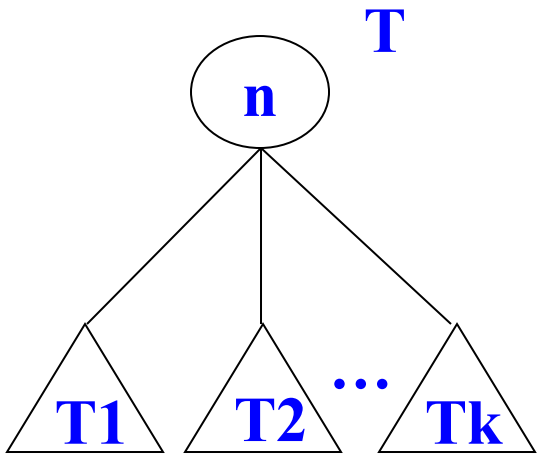
- 树的按层次遍历

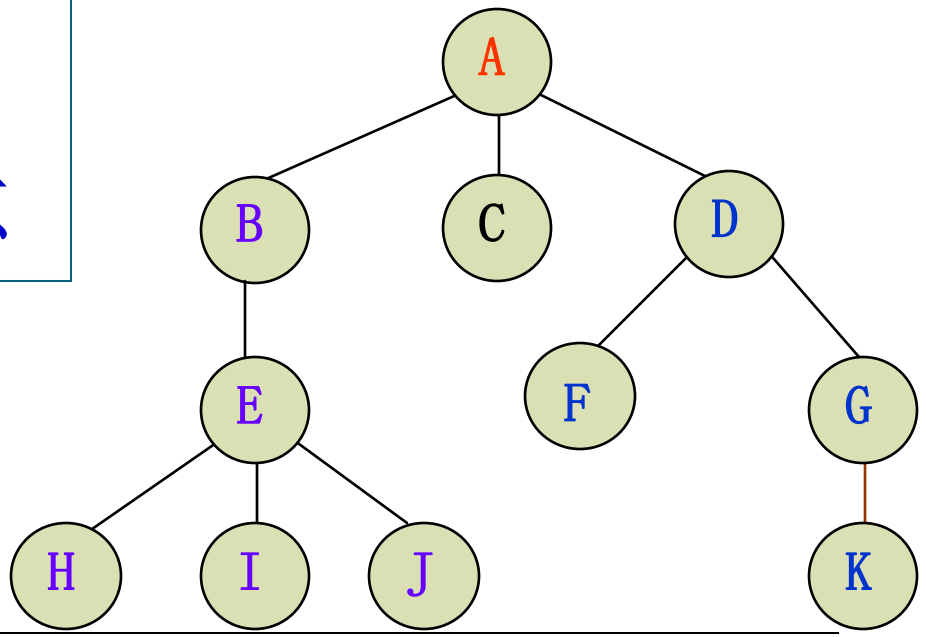
层次遍历

自上而下

自左到右

依次访问树中的每个结点





层次遍历：

A, B, C, D, E, F, G, H, I, J, K

5.5.3 树和森林的遍历

- 树的按层次遍历

算法思想：

算法运用队列做辅助存储结构。其步骤为：

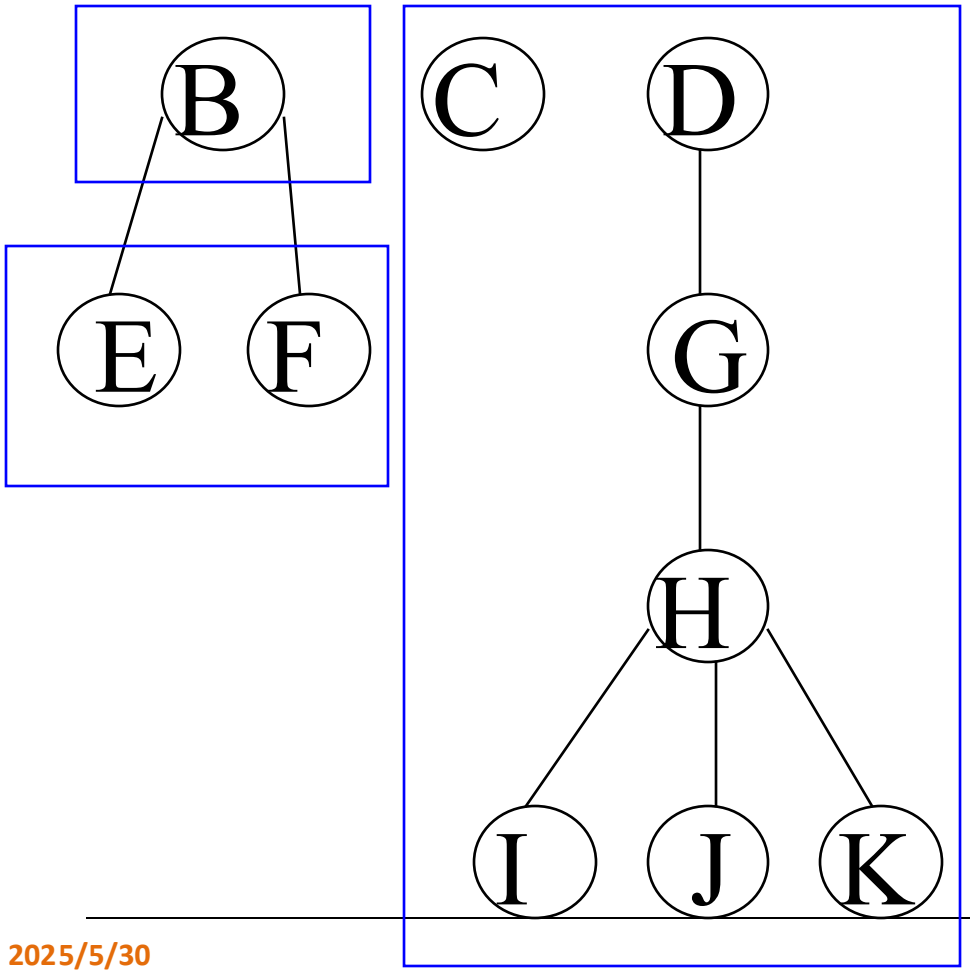
（1）首先将树根入队列；

（2）出队一个结点便立即访问之，然后将它的非空的第一个孩子结点进队，同时将它的所有非空兄弟结点逐一进队；

（3）重复（2），这样便实现了树按层遍历。

5.5.3 树和森林的遍历

- 森林的组成部分



森林由三部分构成：

1．森林中第一棵树的根结点；

2．森林中第一棵树根结点的子树森林；

3．森林中其它树构成的森林。

5.5.3 树和森林的遍历

- 森林的先序遍历

□ 先序遍历

若森林不空，则

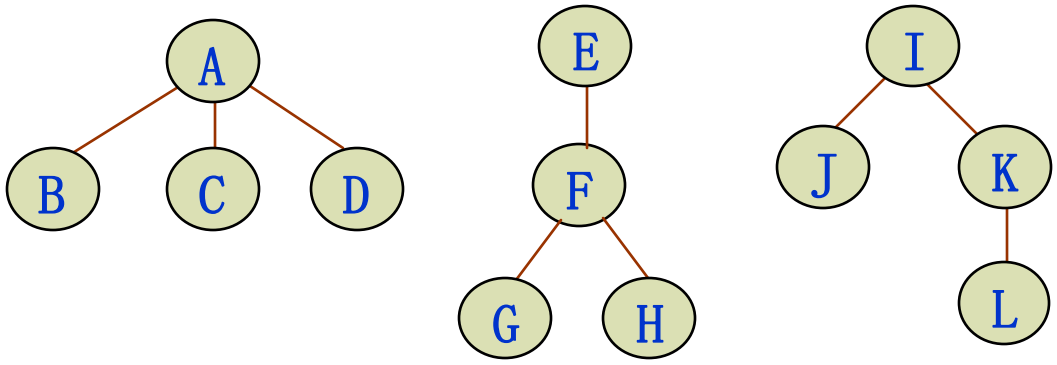
- 访问森林中第一棵树的根结点；
- 先序遍历森林中第一棵树的子树森林；
- 先序遍历森林中(除第一棵树之外)其余树构成的森林。

即：依次从左至右对森林中的每一棵树进行先根遍历。

5.5.3 树和森林的遍历

- 森林的先序遍历

例



A B C D E F G H I J K L

5.5.3 树和森林的遍历

- 森林的中序（后序）遍历

不同教材的说法不一：中序或后序

□ （中）后序遍历

若森林不空，则

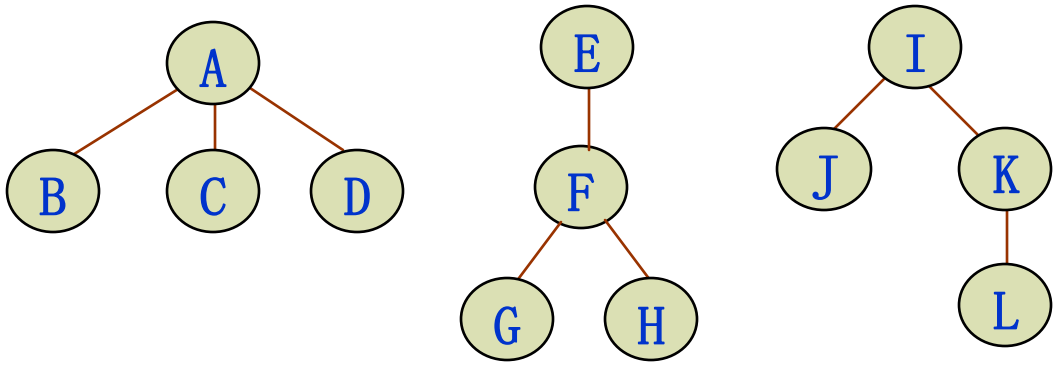
- 后序遍历森林中第一棵树的子树森林；
- 访问森林中第一棵树的根结点；
- 后序遍历森林中(除第一棵树之外)其余树构成的森林。

即依次从左至右对森林中的每一棵树进行后根遍历。

5.5.3 树和森林的遍历

- 森林的中（后）序遍历

例 森林的遍历



B C D A G H F E J L K I

5.5.3 树和森林的遍历

- 森林和二叉树的对应关系

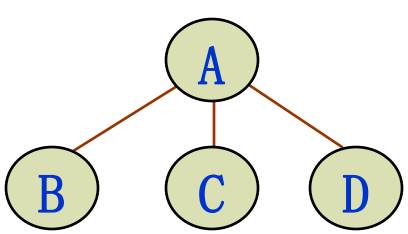
在森林转换成二叉树过程中：

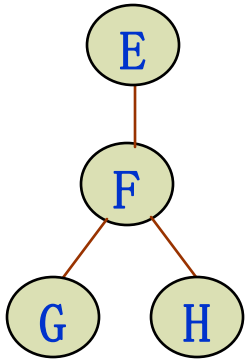
森林		二叉树
第一棵树的根节点	→	二叉树的根
第一棵树的子树森林	→	左子树
剩余树森林	→	右子树

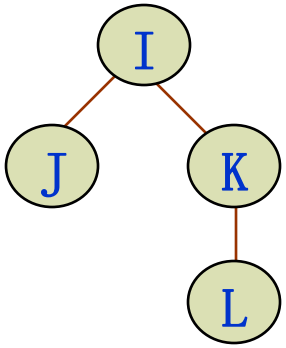
5.5.3 树和森林的遍历

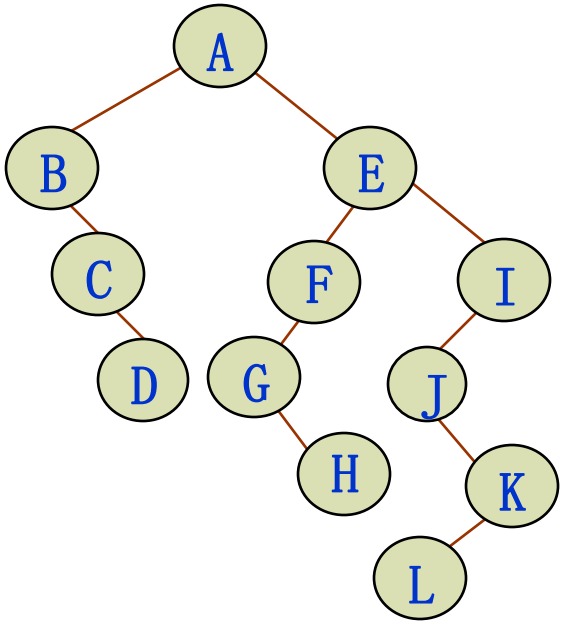
- 森林的中（后）序遍历对应二叉树的中序遍历

例 森林的遍历









森林对应的二叉树

森林的后续遍历：　　B C D A G H F E J L K I

对应的二叉树的中序遍历：B C D A G H F E J L K I

5.5.3 树和森林的遍历

树的遍历和二叉树遍历的对应关系 ？

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中（后）序遍历	中序遍历

当森林转换成二叉树时，森林的先序和中序遍历即为其对应的二叉树的先序和中序遍历。
对树和森林的遍历可以调用二叉树对应的遍历算法。

5.5.3 树和森林的遍历

- 求森林的深度的算法：

```
int TreeDepth(CSTree T) {//T是森林
  if(!T)  return 0;
  else {
     h1 = TreeDepth( T->firstchild );
      //森林中第一棵树的子树森林的深度
     h2 = TreeDepth( T->firstbrother);
      //森林中其他子树构成的森林的深度，与T在同一层
     return(max(h1+1, h2));
  }
} // TreeDepth
```

5.6 树和森林

5.5.1 树的存储结构

5.5.2 树、森林与二叉树的转换

5.5.3 树和森林的遍历

5.5.4 Huffman树及其应用

5.5.4 Huffman树及其应用（连接第8章贪心算法）

- 应用：在电报通信中，电文是以二进制按照一定的编码反射传送。
- 发送方：按照预先规定的方法将要传送的字符换成0和1组成的序列----编码；
- 接收方：由0和1组成的序列换成对应的字符----解码

如何编码能获得较高的传送效率？

Huffman成功解决了该问题！

5.5.4 Huffman树及其应用

- **Huffman教授简介**
  - David Huffman教授，美国人，1999年逝世。
  - 在他的一生中，他对于有限状态自动机、开关电路、异步过程和信号设计有杰出的贡献。
  - 他发明的Huffman编码能够使我们通常的数据传输数量减少到最小。
  - 1950年，Huffman在MIT(麻省理工)的信息理论与编码研究生班学习。Robert Fano教授让学生们自己决定是参加期末考试还是做一个大作业。而Huffman选择了后者，原因很简单，因为解决一个大作业可能比期末考试更容易通过。这个大作业促使了Huffman算法的诞生。

5.5.4 Huffman树及其应用

Huffman树是一类带权路径长度最短的树

---哈夫曼树

---赫夫曼树

---最优二叉树

应用

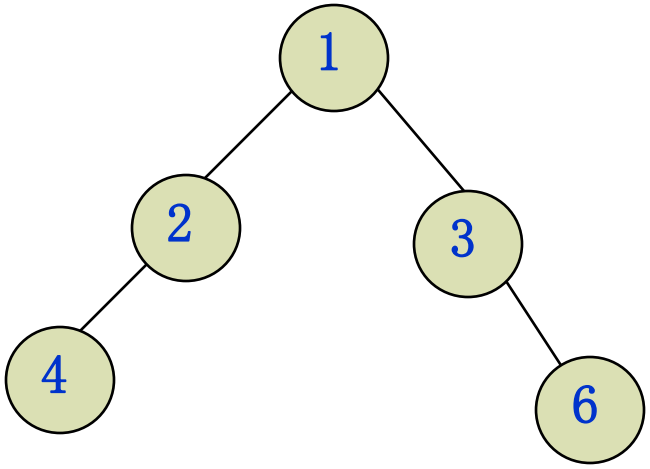
- 哈夫曼编码；
- 通信与数据传送的二进制编码
- 堆结构
- 树形选择排序；
- 折半查找的判定树；
- 动态查找的二叉排序树；
- ……

5.5.4 Huffman树及其应用

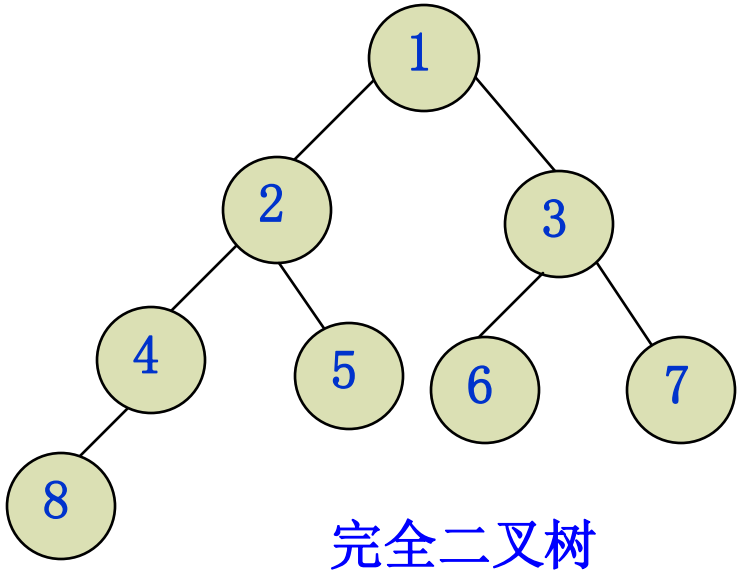
- Huffman树的相关概念

□ 结点的路径长度：
从根结点到该结点的路径上分支的数目

□ 树的路径长度：
树中每个结点的路径长度之和。



非完全二叉树



完全二叉树

5.5.4 Huffman树及其应用

- Huffman树的相关概念

□ 结点的路径长度：
从根结点到该结点的路径上分支的数目

□ 树的路径长度：
树中每个结点的路径长度之和。

在结点数相同的条件下，
完全二叉树是路径最短的二叉树

5.5.4 Huffman树及其应用

- Huffman树的相关概念

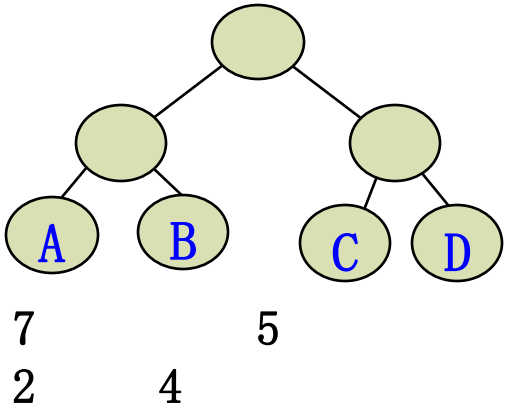
□ 树的带权路径长度：
树中所有叶子结点的带权路径长度之和：
$WPL(T) = \Sigma w_k l_k$ (对所有叶子结点，l为路径长度，w为权重)。

□“最优树”或Huffman树。
在所有含 n 个叶子结点、并带相同权值的 m 叉树中，必存在一棵其带权路径长度取最小值的树，称为“最优树”或Huffman树。

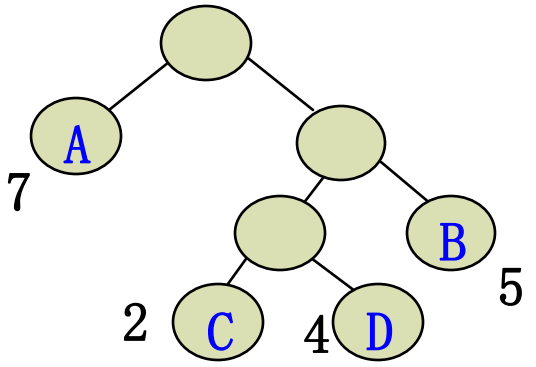
□最优二叉树或哈夫曼树（Huffman）：带权路径长度 WPL 最小的二叉树，称该二叉树为最优二叉树或哈夫曼树。

5.5.4 Huffman树及其应用

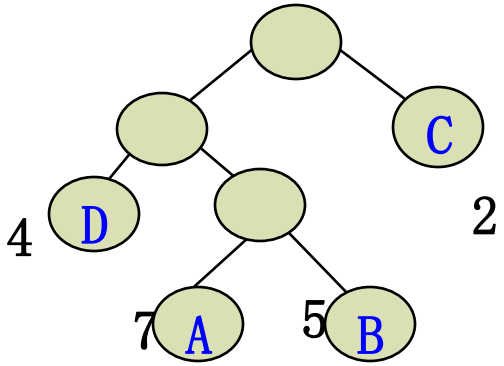
- 带权路径长度



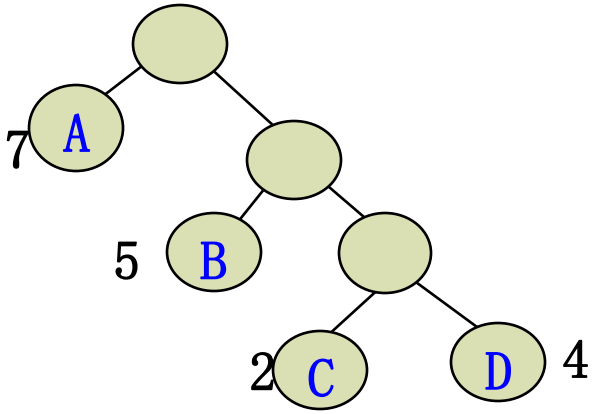
WPL=7*2+5*2+2*2+4*2=36



WPL=7*1+5*2+2*3+4*3=35



WPL=7*3+5*3+2*1+4*2=46

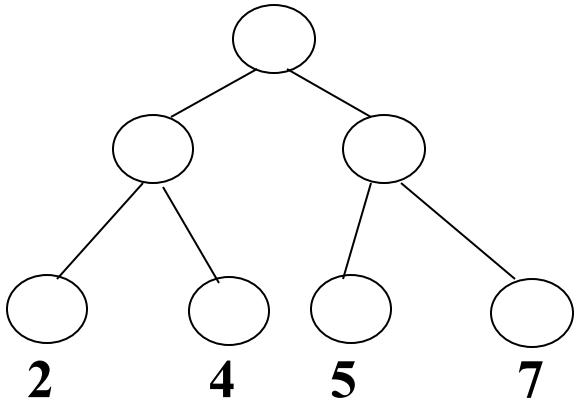


WPL=7*1+5*2+2*3+4*3=35

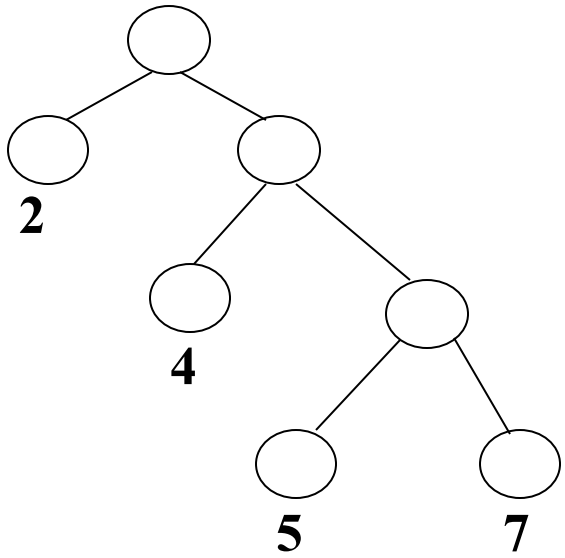
5.5.4 Huffman树及其应用

- Huffman树的相关概念

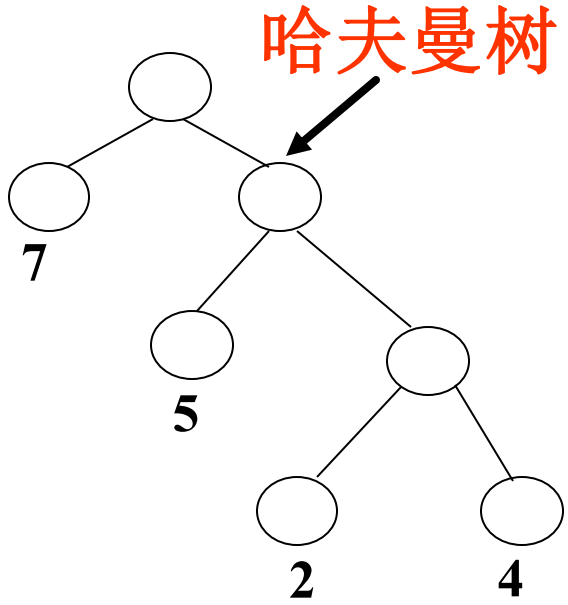
在哈夫曼树中，权值大的结点离根最近。



(a) WPL=36



(b) WPL=46



(c) WPL=35

5.5.4 Huffman树及其应用

- **Huffman树的构造**

为什么这么设计？

要使二叉树WPL小,就须在构造树时,将权值大的结点靠近根。

构造Huffman树的步骤：

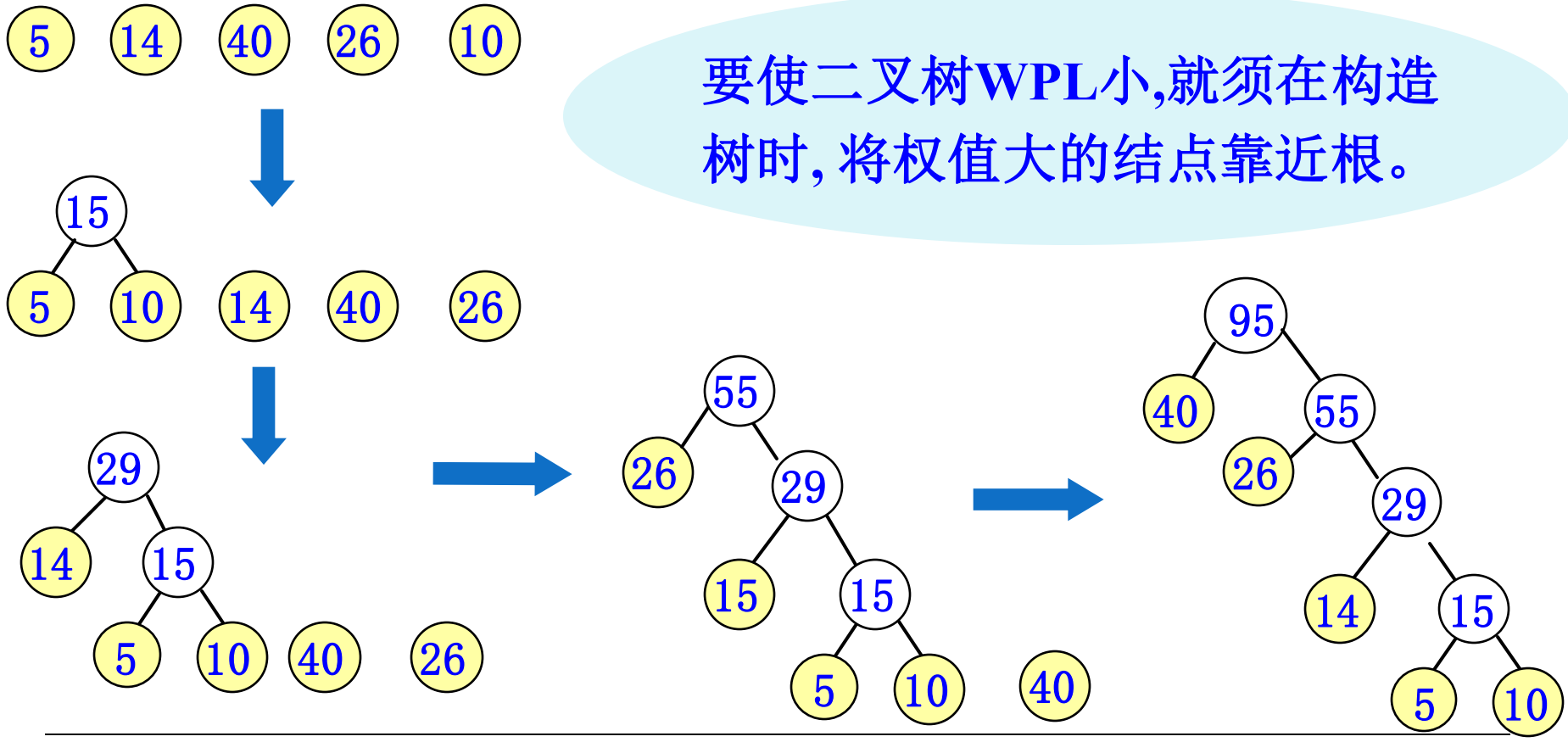
1. 根据给定的n个权值 ，构造n棵只有一个根结点的二叉树，n个权值分别是这些二叉树根结点的权。
2. 设F是由这n棵二叉树构成的集合，在F中选取两棵根结点权值最小的树作为左、右子树，构造一颗新的二叉树，置新二叉树根的权值为左、右子树根结点权值之和；
3. 从F中删除这两颗（权值）树，并将新树加入F；
4. 重复 2、3，直到F中只含一颗树为止；这棵树便是Huffman树

贪心思想：

循环地选择具有最低频率的两个结点，生成一棵子树，直至形成树

5.5.4 Huffman树及其应用

例 构造以W=（5，14，40，26，10）为权的哈夫曼树。



要使二叉树WPL小,就须在构造树时,将权值大的结点靠近根。

5.5.4 Huffman树及其应用

- **Huffman编码**

Huffman树在通讯编码中的一个应用

利用哈夫曼树构造一组最优前缀编码。主要用途是实现数据压缩。在某些通讯场合，需将传送的文字转换成由二进制字符组成的字符串。

方法：

利用哈夫曼树构造一种不等长的二进制编码，并且构造所得的哈夫曼编码是一种最优前缀编码，使所传电文的总长度最短。

5.5.4 Huffman树及其应用

- Huffman编码

□等长编码

这类编码的二进制串的长度取决于电文中不同的字符个数，假设需传送的电文中只有四种字符，只需两位字符的串便可分辨。(01, 10, 11, 00)

例 要传输的原文为ABACCDA

等长编码 A：00 B：01 C：10 D：11

发送方：将ABACCDA 转换成 000100101011100

接收方：将 000100101011100 还原为 ABACCDA

5.5.4 Huffman树及其应用

- Huffman编码

□不等长编码

即各个字符的编码长度不等。(0, 10, 110, 011,)

出现次数较多的字符采用尽可能短的编码，编码总长度减小

例

不等长编码　A：0　B：00　C：1　D：01

发送方：将ABACCDA　转换成　000011010

接收方：000011010　转换成？

解码不唯一，怎么办？前缀编码。

5.5.4 Huffman树及其应用

- Huffman编码

□前缀编码

任何一个字符的编码都不是同一字符集中另一个字符的编码的前缀。

例 前缀编码 A：0 B：110 C：10 D：111

发送方：将ABACCDA 转换成 0110010101110 发出

接收方： 0110010101110 。所得的译码是唯一的。

5.5.4 Huffman树及其应用

- **Huffman编码**

例 假定我们希望压缩一个10万字符的数据文件，其中a出现了45000次。

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

使用定长编码：需要3位二进制来表示每个字符，那么总共需要300,000个二进制位来编码文件。

5.5.4 Huffman树及其应用

- **Huffman编码**

例 假定我们希望压缩一个10万字符的数据文件，其中a出现了45000次。

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

使用变长编码：高频字符用短码，低频字符用长码，如图，总共需要：

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

与定长编码相比，节省了25%的空间。这也是此文件的最优字符编码。

5.5.4 Huffman树及其应用

- Huffman编码

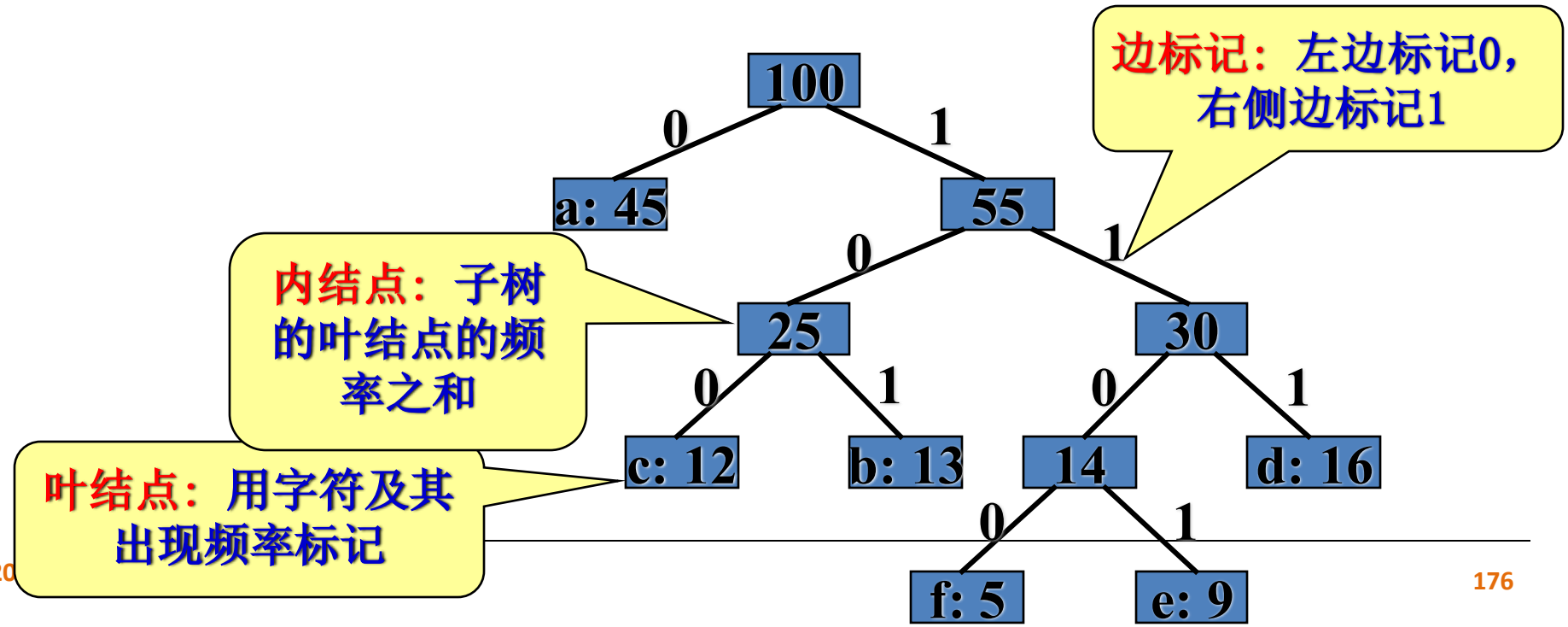
□不等长编码的好处

可以使传送电文的字符串的总长度尽可能地短。对出现频率高的字符采用尽可能短的编码，则传送电文的总长便可减少。

怎么基于Huffman得到最短的前缀编码？

5.5.4 Huffman树及其应用

- **Huffman编码**
  - 如何表示前缀码？以便于解码。
  - 编码树是前缀码的一种表示形式，是一种二叉树：叶节点为给定的字符以及频率，字符的二进制编码（码字）为从根节点到该字符叶节点的简单路径表示，其中0为转向左子树，1为转向右子树。内部节点标记了其子树中叶节点的频率之和。



5.5.4 Huffman树及其应用

- 利用二叉树设计前缀编码：

例 某通讯系统只使用8种字符a、b、c、d、e、f、g、h，其使用频率分别为0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11，利用二叉树设计一种不等长编码：

1）构造以 a、b、c、d、e、f、g、h为叶子结点的二叉树；

2）约定：将该二叉树所有左分枝标记0，所有右分枝标记1；

3）从根到叶子结点路径上的标记字符串作为叶子结点所对应字符的编码；

电文总的编码长度为该二叉树的带权路径长度：
$WPL(T) = \Sigma w_k l_k$，其中w_k为第k个叶子的权重，l_k为路径长度，由此：
设计电文总长度最短的二进制前缀编码问题转化为求Huffman树问题

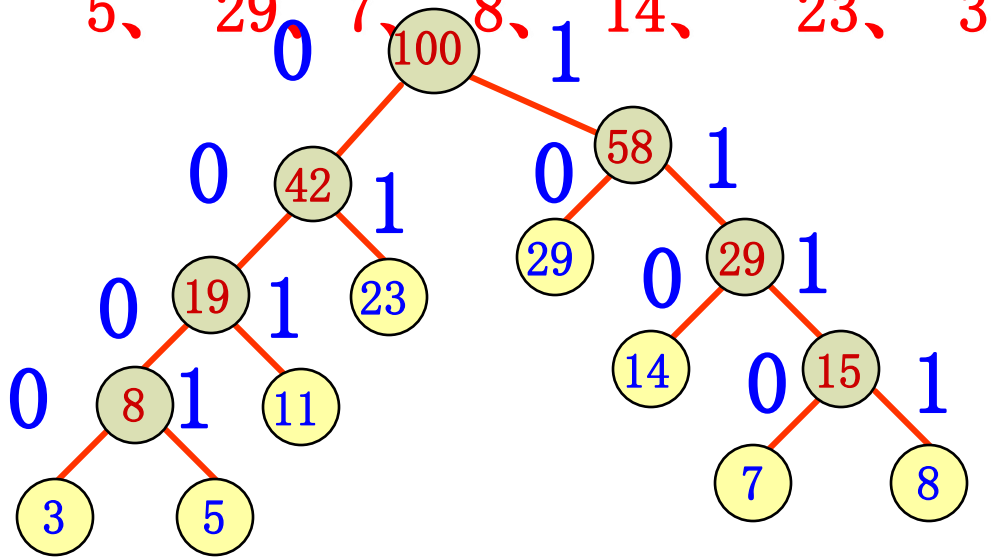
5.5.4 Huffman树及其应用

- 利用二叉树设计前缀编码：

构造以字符使用频率作为权值的*Huffman*树

a、 b、 c、 d、 e、 f、 g、 h

5、 29、 7、 8、 14、 23、 3、 11



方式一：两棵子树中，小的值放在左边

a：0001
b：10
c：1110
d：1111
e：110
f：01
g：0000
h：001

- 总编码长度等于Huffman树的带权路径长度WPL。
- Huffman编码是一种前缀编码。解码时不会混淆。

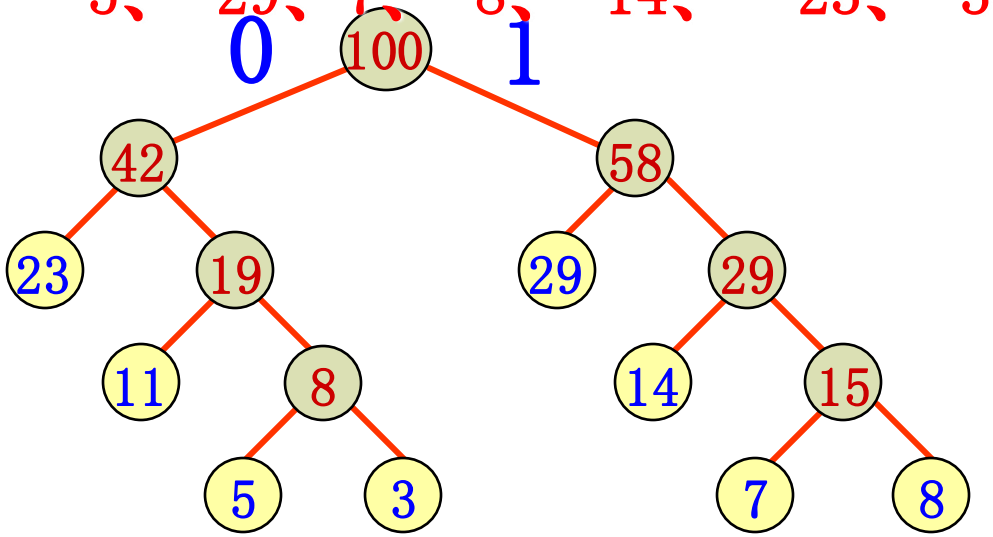
5.5.4 Huffman树及其应用

- 利用二叉树设计前缀编码：

构造以字符使用频率作为权值的*Huffman*树

a、 b、 c、 d、 e、 f、 g、 h

5、 29、 7、 8、 14、 23、 3、 11



方式二：两棵子树中，小的值放在左边

a: 0110
b: 10
c: 1110
d: 1111
e: 110
f: 00
g: 0111

方式一vs方式二，最佳前缀码并不唯一：由于每一步选择两个最小的权的选法可能不唯一（如果有多个并列的最小权值），而且两个权对应的顶点所放的左右位置也可以不同，所以画出的最优树可能不同。但是最优树的带权路径长度之和WPL一定相等。

5.5.4 Huffman树及其应用

显然字符使用频率越小权值越小，权值越小叶子就越靠下，于是频率小编码长，频率高编码短，这样就保证了此树的最小带权路径长度，效果上就是传送报文的最短长度。

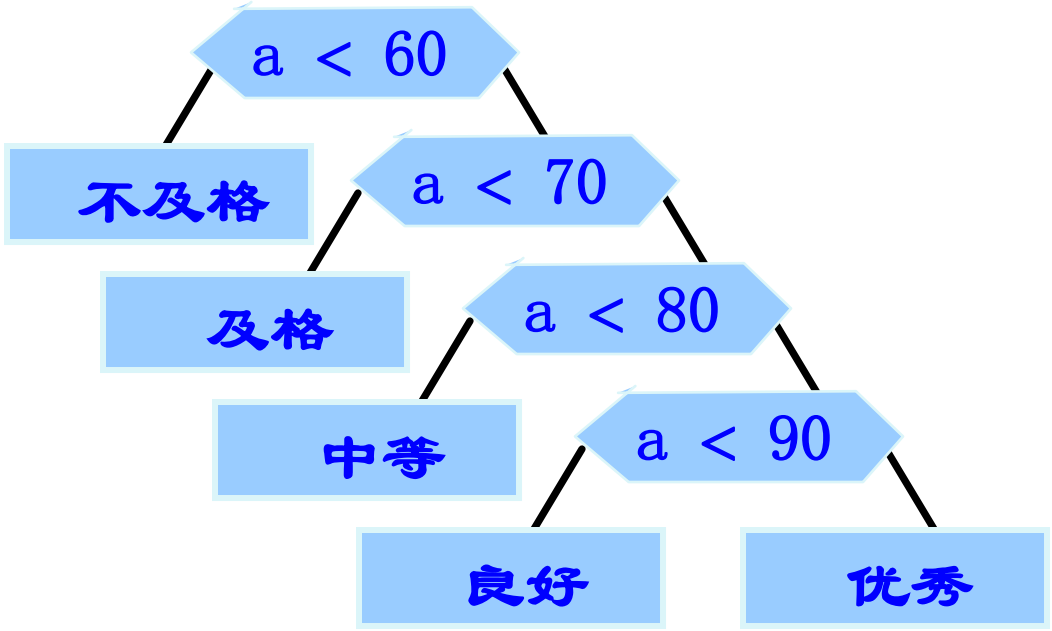
将求传送报文的最短长度问题转化为求由字符集中的所有字符作为叶子结点，由字符出现频率作为其权值所产生的哈夫曼树的问题。利用哈夫曼树来设计二进制的前缀编码，既满足前缀编码的条件，又保证报文编码总长最短。

为什么Huffman算法产生的二叉树为最优二叉树？如何证明？

严谨证明详见后续贪心算法课件！

5.5.4 Huffman树及其应用

例 编制一个将百分制转换成五分制的程序。可用二叉树描述判定过程。



在求得某些判定问题时，利用Huffman树获得最佳判定算法。

5.5.4 Huffman树及其应用

例 设有10000个百分制分数要转换，设学生成绩在5个等级以上的分布如下：

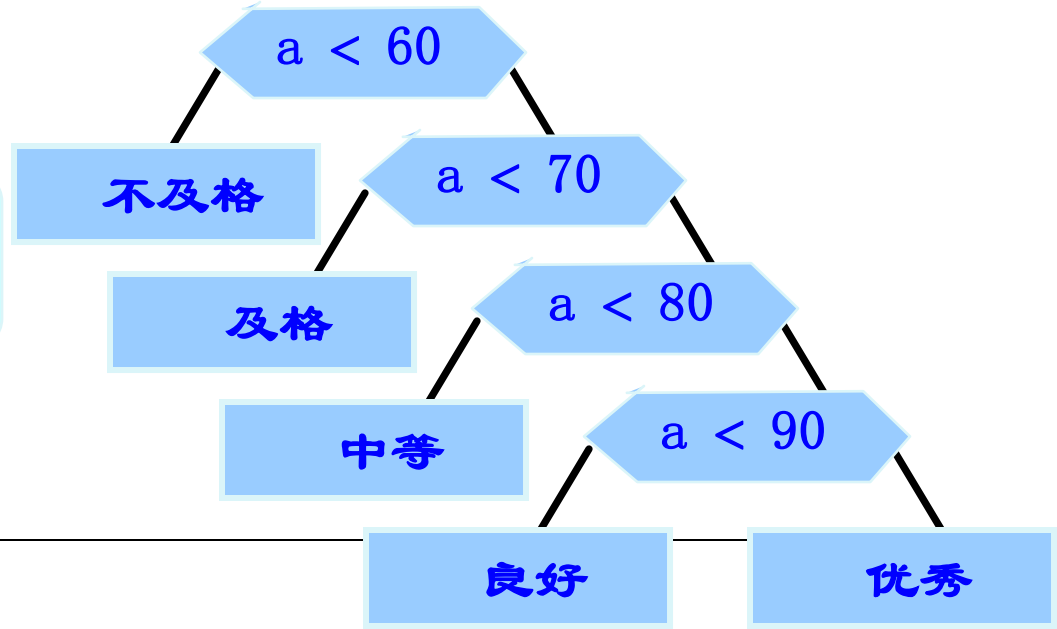
分数	0-59	60-69	70-79	80-89	90-100
比例数	0.05	0.15	0.40	0.30	0.10

其判定过程：

转换一个分数所需的比较次数= 从根到对应结点的路径长度

转换10000个分数所需的总比较次数= $10000 \times (0.05 \times 1+0.15 \times 2+0.4 \times 3+0.3 \times 4+0.1 \times 4)$

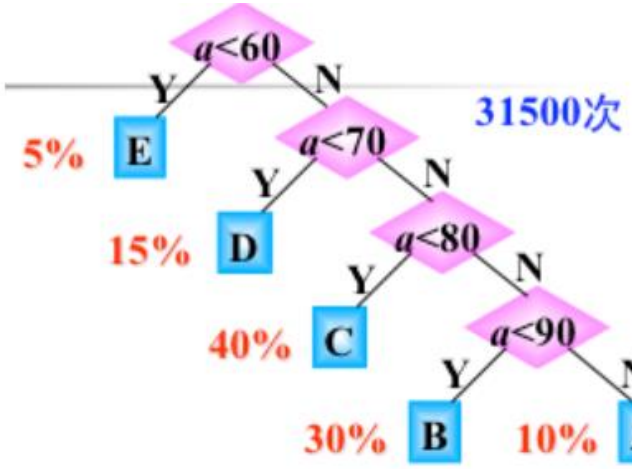
二叉树的
带权路径长度

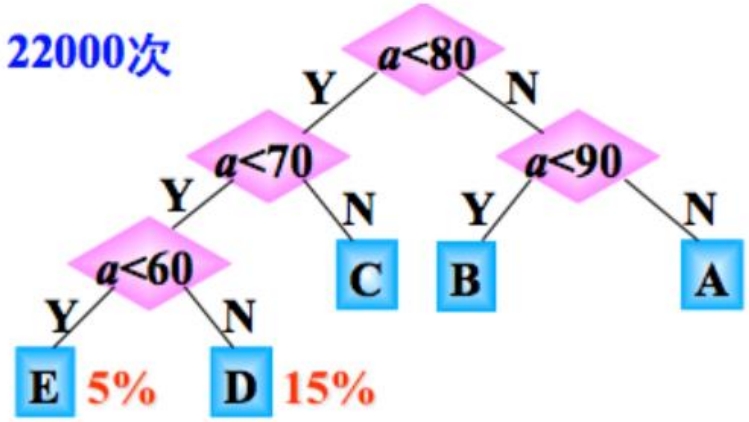


5.5.4 Huffman树及其应用

例 设有10000个百分制分数要转换，设学生成绩在5个等级以上的分布如下：

分数	0-59	60-69	70-79	80-89	90-100
比例数	0.05	0.15	0.40	0.30	0.10





次数为： 10000 (5%＋2×15%＋3×40%＋4×40%)＝31500次

此种形状的二叉树，需要的比较次数是：10000 (3×20%＋2×80%)＝22000次，显然：两种判别树的效率是不一样的。

必须临近的结点进行合并，不能跳跃，因为有连续性判定的问题

5.5.4 Huffman树及其应用

如何用数据结构实现Huffman树？

5.5.4 Huffman树及其应用

- Huffman树的构造

Huffman树中没有度为1的结点，这类树又称严格的（strict）或正则的二叉树

n个叶子结点构造的Huffman树最终有多少个结点？

Huffman树的构造过程说明：

n个结点需要进行n-1次合并，每次合并都产生一个新的结点，所以最终的Huffman树共有2n-1个结点。

5.5.4 Huffman树及其应用

- Huffman算法

```
Huffman 树结点结构
typedef struct
{
    float weight;
    int lch, rch, parent;
}Hufmtree;
Hufmtree tree[m]; // 用一维数组存储m=2n-1个结点，其中下标0到n-1个元素对应n个叶子结点，下标n到m-1对应新建的内部节点
```

```
CreatHuffmanTree (tree)
{
    int m = 2*n-1;
    int i, j, p1, p2;
    float small1, small2;
    for (i=0; i<m; i++) //初始化
    {
        tree[i].weight=0.0;
        tree[i].lch=-1;
        tree[i].rch=-1;
        tree[i].parent=-1;
    }
}
```

5.5.4 Huffman树及其应用

- Huffman算法

```
int Huffmantree(HuffmanTree T)
{
   //步骤1：初始化权值
   float w; int i;
   for (i=0; i<n;i++)
   {
        scanf("%f",&w);
        tree[i].weight=w;
    }
   //步骤2：建Huffman树，把合并后的结点放入向tree[n]~tree[m]
   for(i=n; i<m; i++)
   {
   //步骤2.1：选择parent为-1且weight最小的两个结点，其序号赋值为p1和p2
        p1=0; p2=0;          // 两个根结点在向量tree中的下标
        small1=maxval; // small1和small2分别记录当前最小的权重
        small2=maxval; //maxval 是float类型的最大值
```

5.5.4 Huffman树及其应用

- Huffman算法

```
for (j=0;j<i-1;j++)
{
   if(tree[j].parent==-1) {
        if(tree[j].weight<small1)
        {
           small2=small1;//改变最小权，次小权及其位置
           small1=tree[j].weight;//找出最小的权值
           p2=p1; p1=j;
        }
        else if(tree[j].weight<small2)
        {
           small2=tree[j].weight;//改变次小权及位置
           p2=j;
        }}
}//步骤2.1选择结束
```

5.5.4 Huffman树及其应用

- Huffman算法

```
    // 步骤2.2：新建新结点，作为p1和p2的父结点，存放在tree[i]
    tree[p1].parent=i;
    tree[p2].parent=i;
    tree[i].lch = p1;
    tree[i].rch = p2;
    tree[i].weight= tree[p1]. weight + tree[p2]. weight;
    }// for(i=n; i<m; i++) 步骤2结束
}//Huffmantree
```

建好树后，可以从叶子到根逆向求每个字符的Huffman编码，参照清华《数据结构》p147页，算法6.12

5.5.4 Huffman树及其应用

- Huffman算法

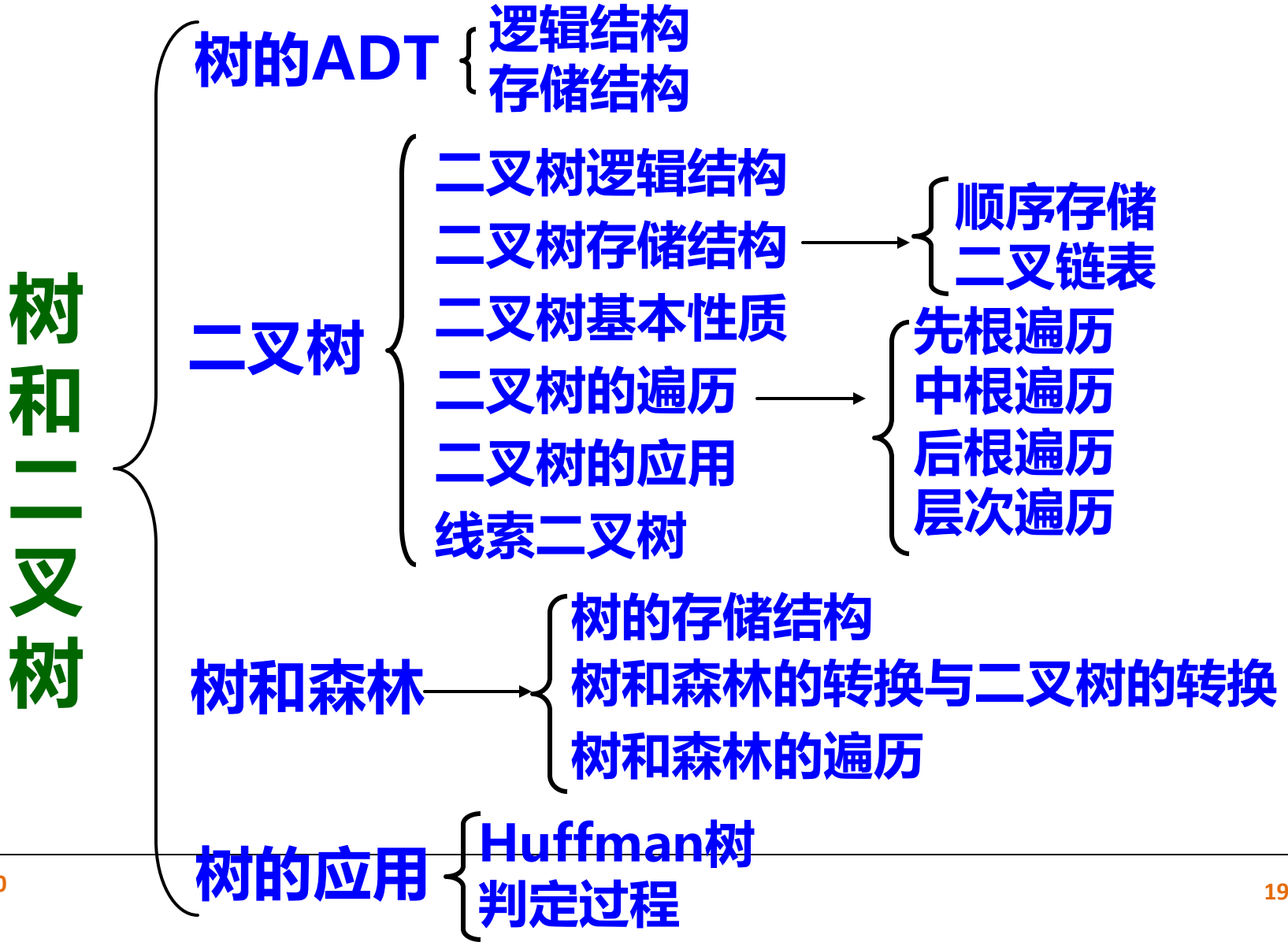
另一种数据结构方法：用tag标志位

```
Huffman 树结点结构
typedef struct
{
    float weight;
    int lch,rch;
     int tag;
}huffree;
hufftree tree[m];
```

	tag	lch	weight	rch
1	0	0	5	0
2	0	0	6	0
3	0	0	2	0
4	0	0	9	0
5	0	0	7	0
6				
7				
8				
9				

tag=0 结点独立
Tag=1 结点已并入树中

本章小结


树和二叉树
树的ADT
逻辑结构
存储结构
二叉树
二叉树逻辑结构
二叉树存储结构
顺序存储
二叉链表
二叉树基本性质
二叉树的遍历
先根遍历
中根遍历
后根遍历
层次遍历
二叉树的应用
线索二叉树
树和森林
树的存储结构
树和森林的转换与二叉树的转换
树和森林的遍历
树的应用
Huffman树
判定过程

本章小结

- ✓ 树和二叉树两种数据类型的定义
- ✓ 二叉树的性质
- ✓ 二叉树的遍历
- ✓ 线索二叉树
- ✓ 二叉树的应用
- ✓ 树存储和遍历
- ✓ 树、森林与二叉树之间的转换
- ✓ 树的应用（重点是Huffman 树）

本章小结

- ✓ 熟练掌握二叉树的结构特性，了解相应的证明方法。（完全二叉树、满二叉树、哈夫曼树）
- ✓ 熟悉二叉树的各种存储结构的特点及适用范围。
- ✓ 遍历二叉树是二叉树各种操作的基础。实现二叉树遍历的具体算法与所采用的存储结构有关。掌握各种遍历策略的递归算法，灵活运用遍历算法实现二叉树的其它操作。层次遍历是按另一种搜索策略进行的遍历。
- ✓ 理解二叉树线索化的实质是建立结点与其在相应序列中的前驱或后继之间的直接联系；
- ✓ 熟练掌握二叉树的线索化过程以及在中序线索化树上找给定结点的前驱和后继的方法；
- ✓ 二叉树的线索化过程是基于对二叉树进行遍历，而线索二叉树上的线索又为相应的遍历提供了方便。

本章小结

- ✓ 熟悉树的各种存储结构及其特点，掌握树和森林与二叉树的转换方法。
- ✓ 建立存储结构是进行其它操作的前提，因此应掌握 1 至 2 种建立二叉树和树的存储结构的方法。
- ✓ 学会编写实现树的各种操作的算法。
- ✓ 了解最优树的特性，掌握建立最优树和哈夫曼编码的方法。