现实问题 → 树

研究树的目的：

使树的操作都是通过二叉树来完成的

6.6 树和森林

6.6.1 树的存储结构

6.6.2 树、森林与二叉树的转换

6.6.3 树和森林的遍历

6.6.4 huffman树及其应用

6.6 树和森林

6.6.1 树的存储结构

6.6.2 树、森林与二叉树的转换

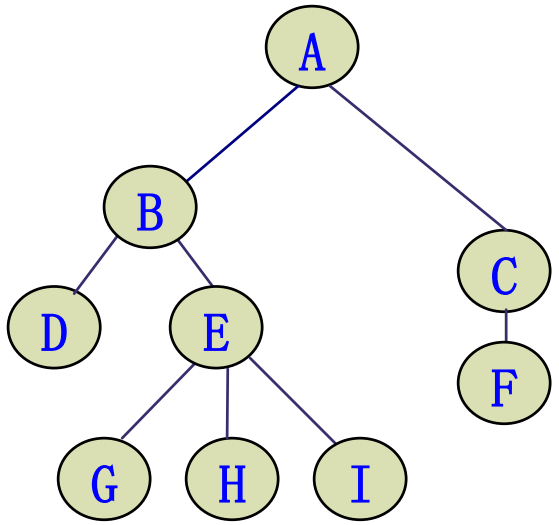
6.6.3 树和森林的遍历

6.6.4 huffman树及其应用

6.6.1 树的存储结构

- 双亲表示法（顺序存储）

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。



0	data	parent
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

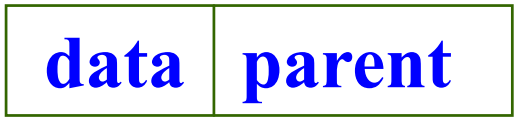
6.6.1 树的存储结构

- 双亲表示法

双亲表示类型定义

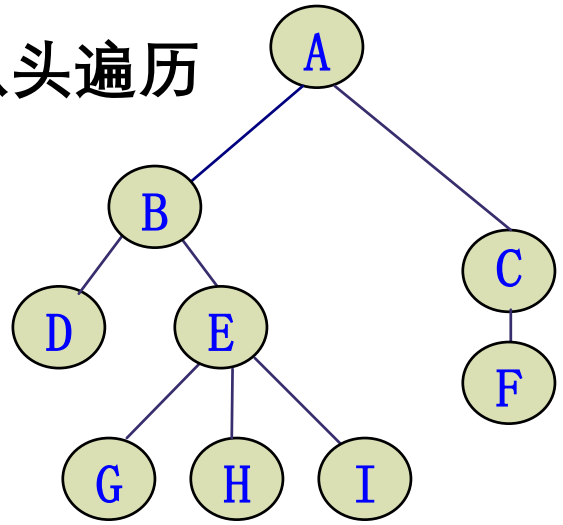
```
   #define MAX_TREE_SIZE 100
Typedef struct PTnode{    //结点结构
   Elem data;
   int   parent;   //双亲位置域
} PTnode;

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



6.6.1 树的存储结构

- 双亲表示法特点
  - 增加一个元素无需按逻辑顺序存储
  - 删除叶子结点容易，非叶子结点困难
  - 查找指定结点的双亲方便
  - 查找指定结点的孩子只能从头遍历

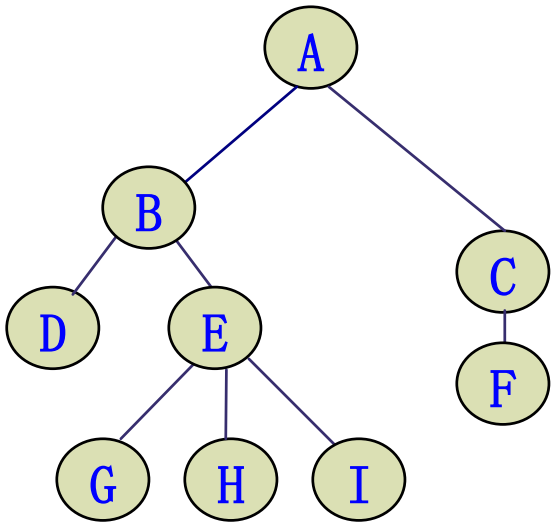


0	data	parent
1	A	0
2	B	1
3	C	1
4	D	2
5	E	2
6	F	3
7	G	5
8	H	5
9	I	5

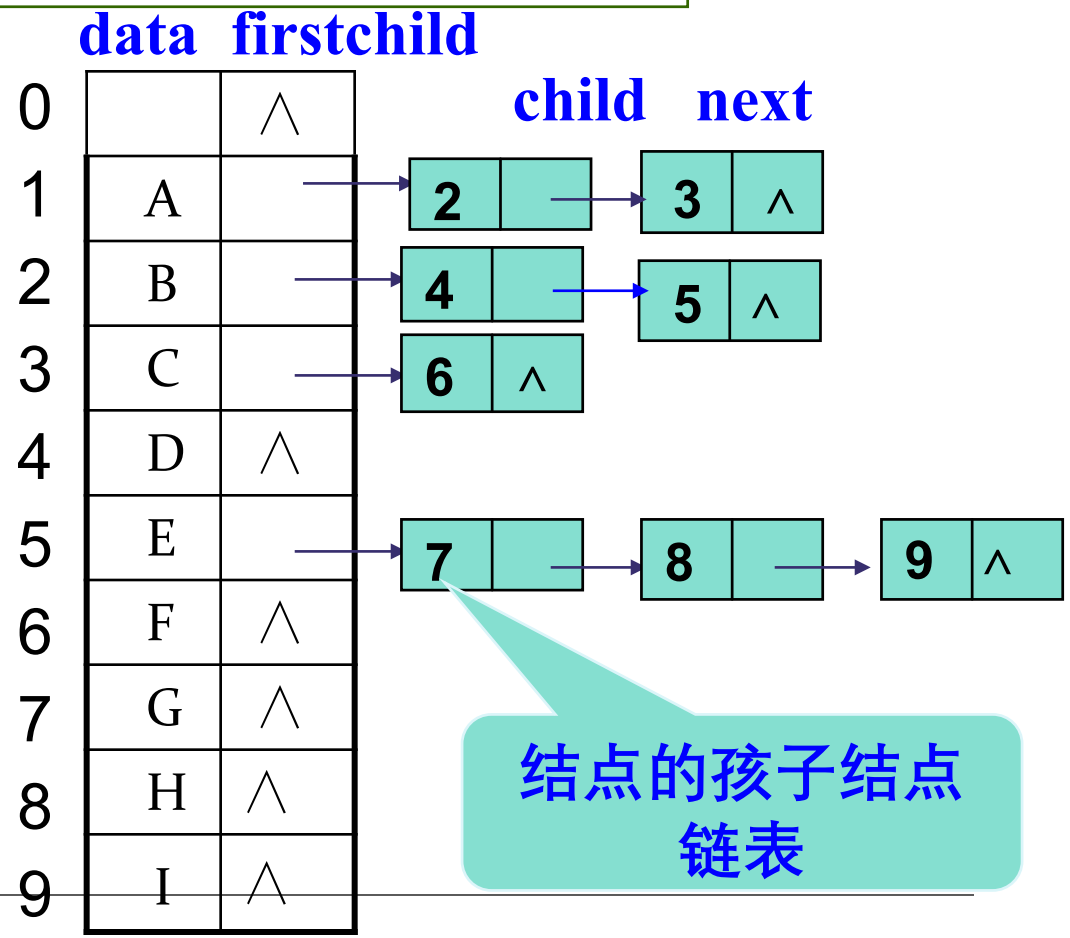
6.6.1 树的存储结构

- 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

6.6.1 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

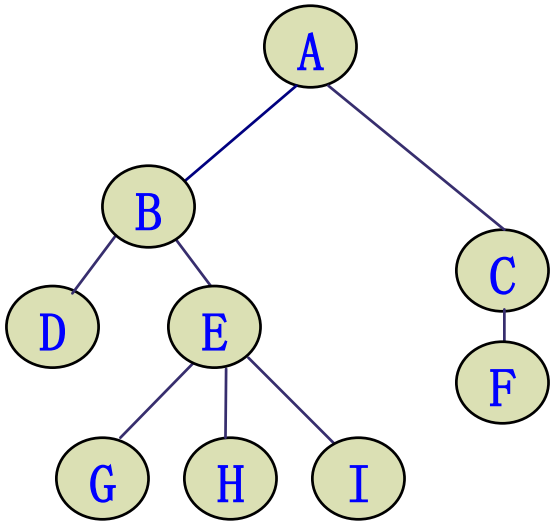
```
typedef  struct  CTNode  {
  int    child ;
  struct  CTNode  *next ; }   *ChildPtr ;
typedef  struct  {
  Telementtype  data ;
   ChildPtr  firstchild ;      }     CTBox ;
typedef  struct  {
   CTBox  Nodes[MAX_TREE_SIZE] ;  int   n , r ; }  Ctree ;
```

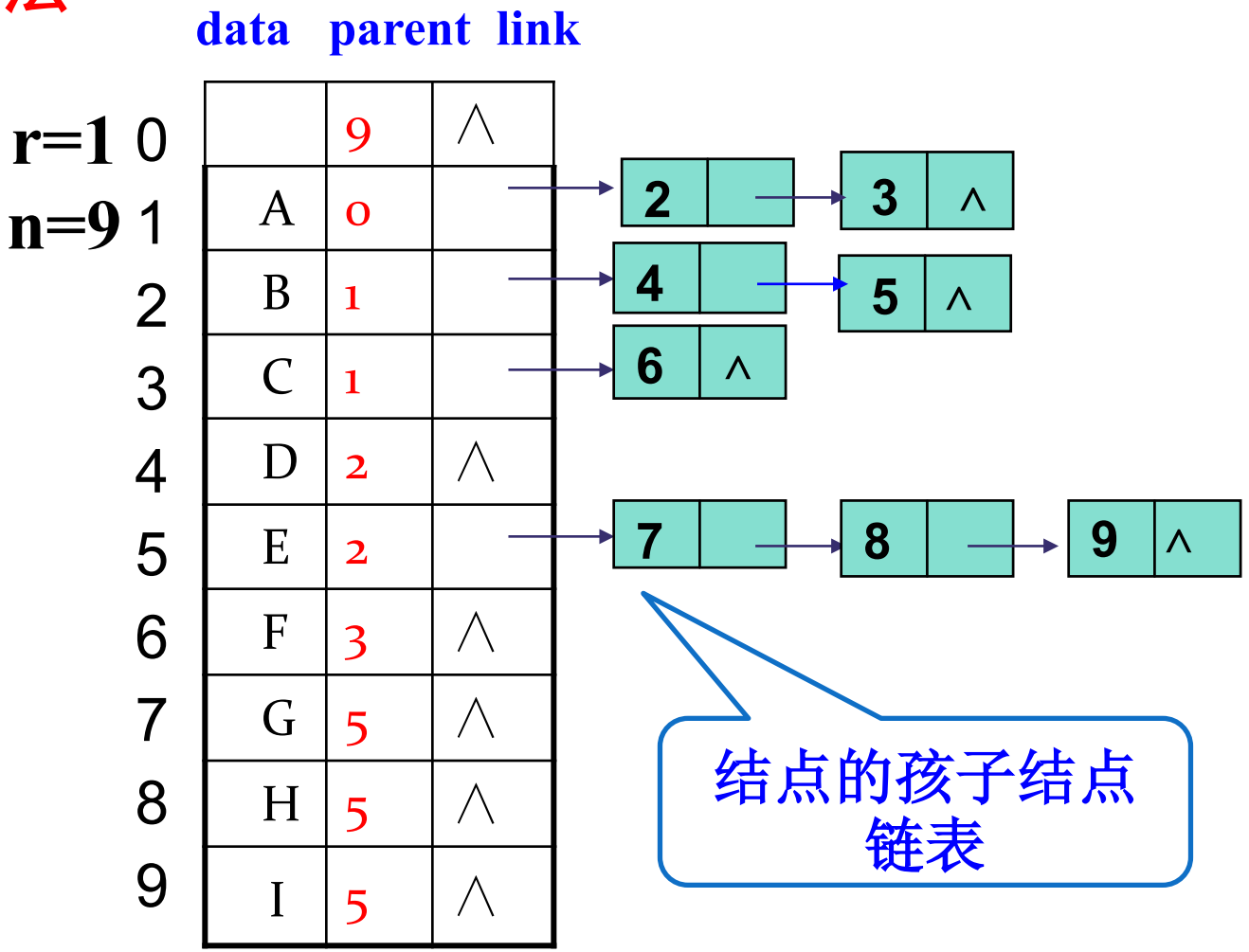
找一个结点的孩子十分方便，但要找一个结点的双亲则要遍历整个结构

6.6.1 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





带双亲孩子链表示意图

6.6.1 树的存储结构

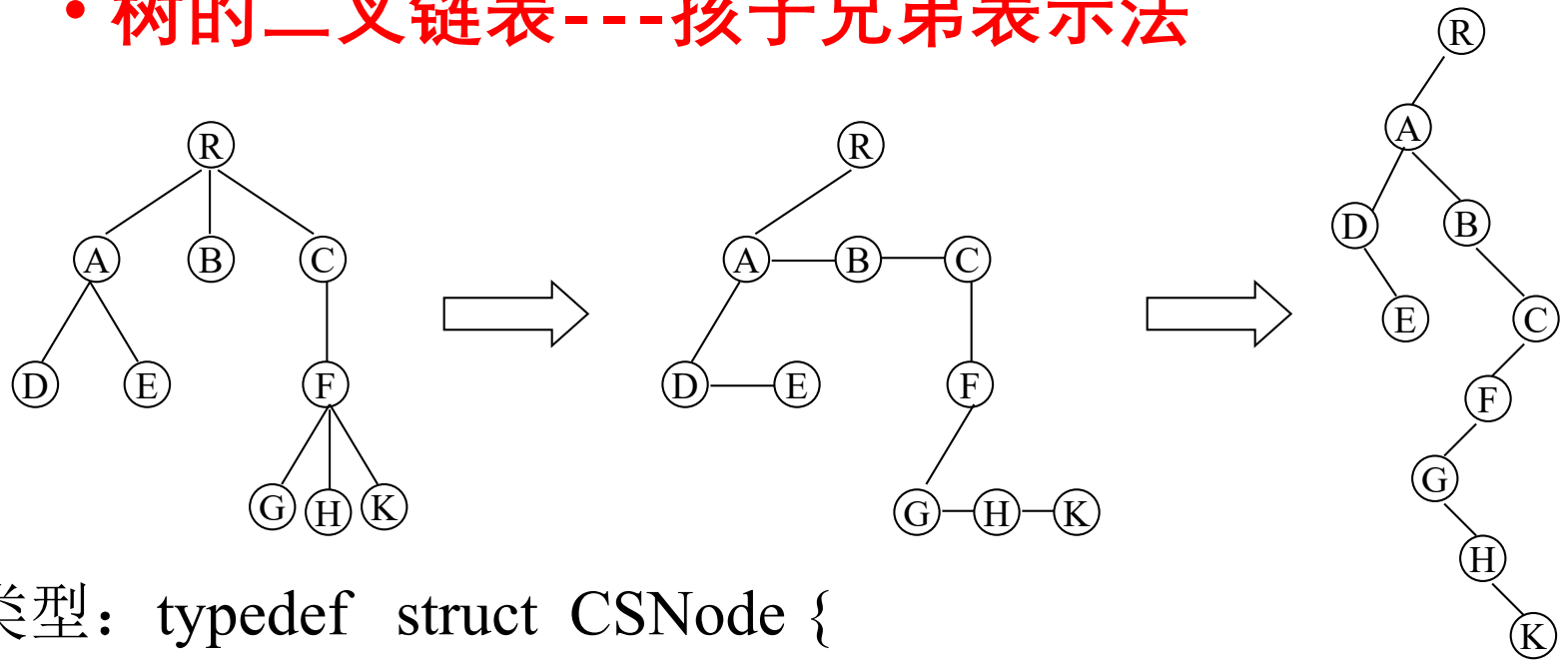
- 树的二叉链表---孩子兄弟表示法

如何把树与二叉树联系起来？

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下一个兄弟结点。

6.6.1 树的存储结构

- 树的二叉链表---孩子兄弟表示法



类型：typedef struct CSNode {
ElemType data;
struct CSNode *firstchild , *nextsibling ; } ;

与二叉树的存储表示（结点类型）一样，但含义不同

6.6.1 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。

把树和二叉树对应起来

如何把树转化成二叉树？

6.6 树和森林

6.6.1 树的存储结构

6.6.2 树、森林与二叉树的转换

6.6.3 树和森林的遍历

6.6.4 huffman树及其应用

6.6.2 树、森林与二叉树的转换

- 树转换为二叉树的方法

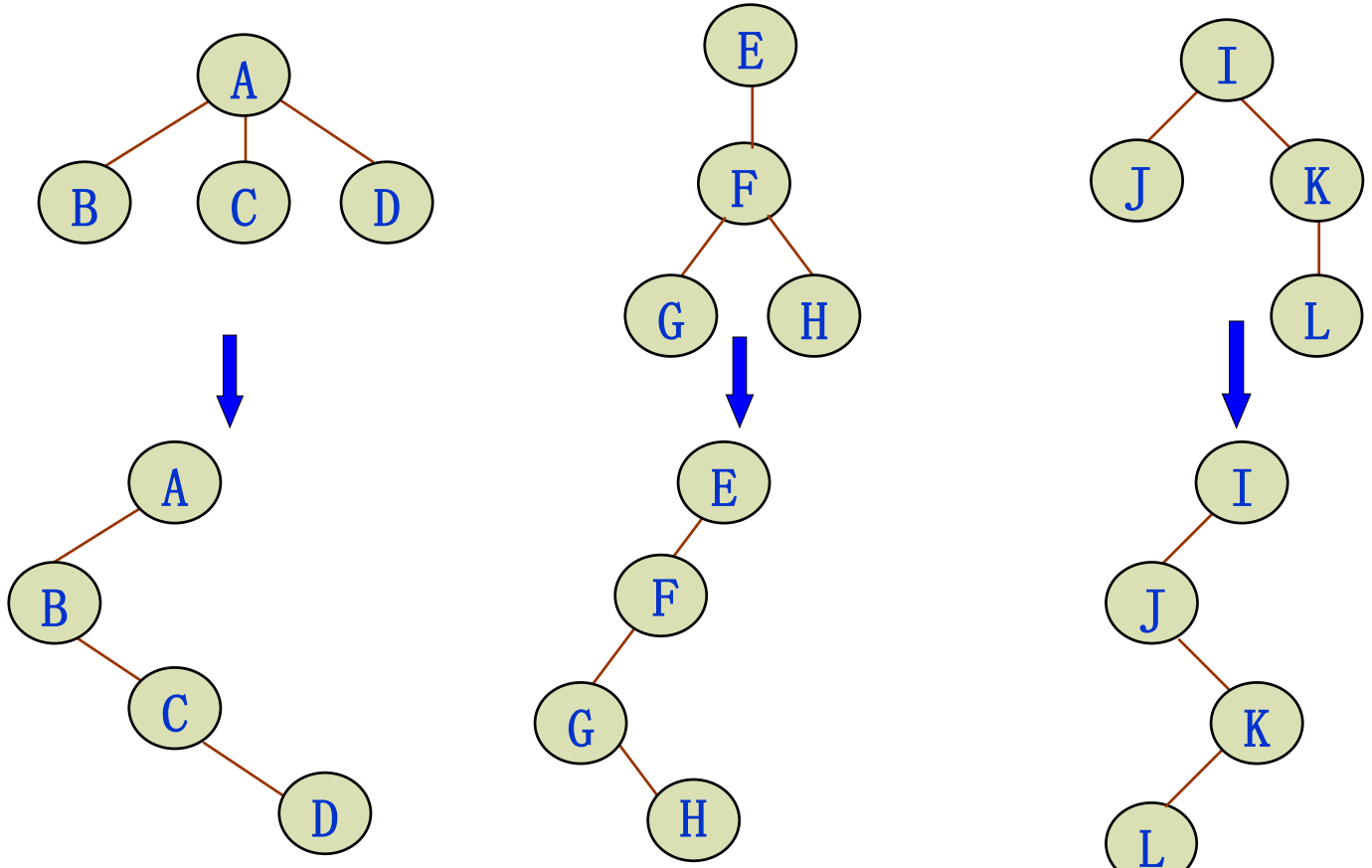
（1）在所有兄弟结点之间加一条连线；

（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；

图示

6.6.2 树、森林与二叉树的转换

例 将树转换为二叉树



每棵树对应的二叉树

6.6.2 树、森林与二叉树的转换

- 森林转换为二叉树的方法

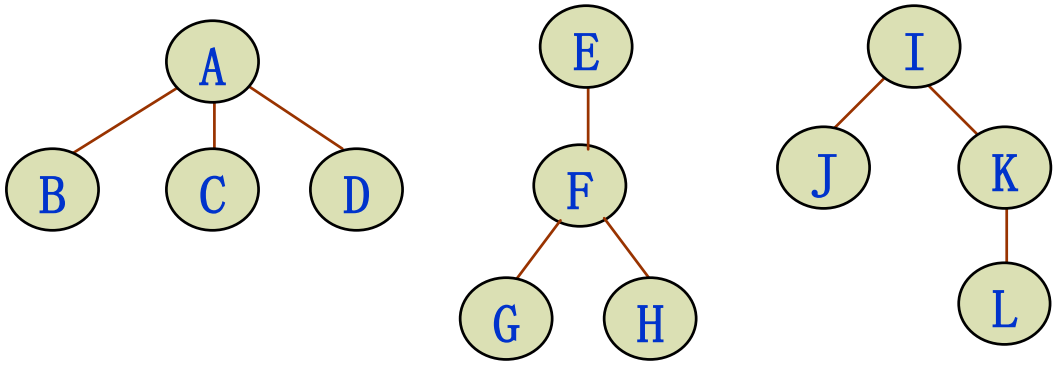
（1）将森林中的每一树转换二叉树；

（2）将各二叉树的根结点视为兄弟连在一起。

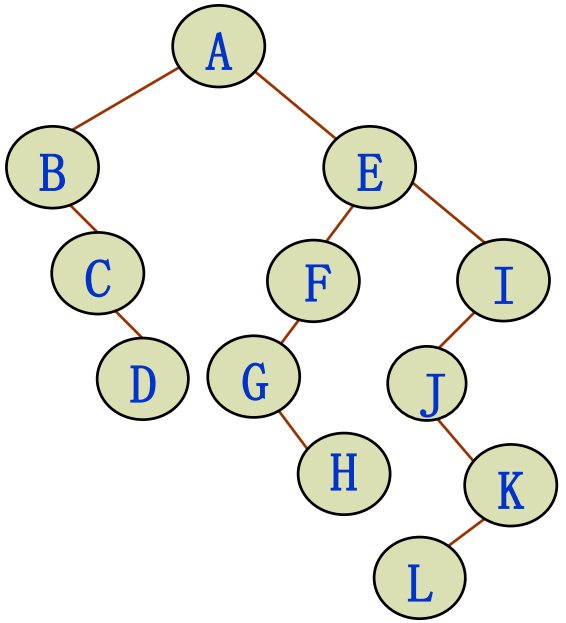
图示

- 森林转换为二叉树的方法

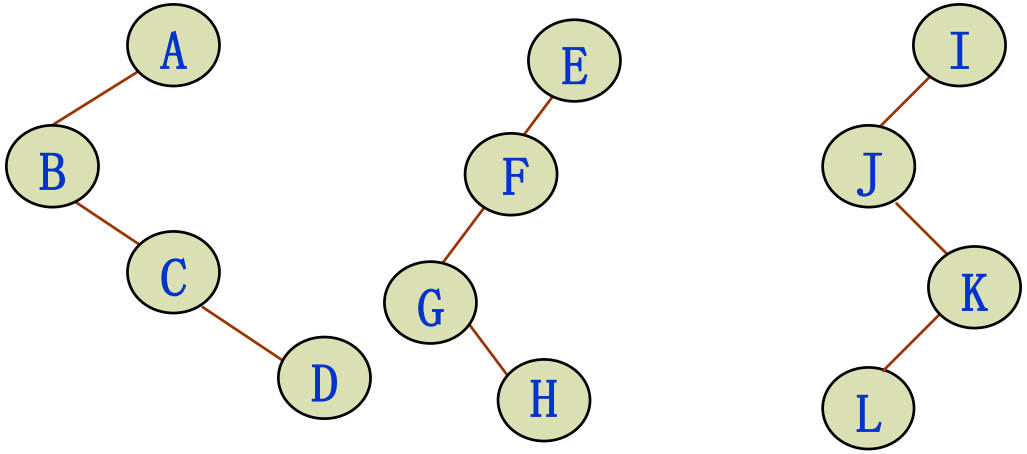
例 森林转换为二叉树



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

6.6.2 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

图示

6.6.2 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

6.6 树和森林

6.6.1 树的存储结构

6.6.2 树、森林与二叉树的转换

6.6.3 树和森林的遍历

6.6.4 huffman树及其应用

6.6.3 树和森林的遍历

树的遍历

- 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。
- 对树进行遍历可有两条搜索路径：
- （1）按层次从左到右，从上到下。
  - 树的按层次遍历类似于二叉树的按层次遍历；
- （2）按树的组成（根+子树），进行先根遍历和后根遍历
  - 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遍历。

6.6.3 树和森林的遍历

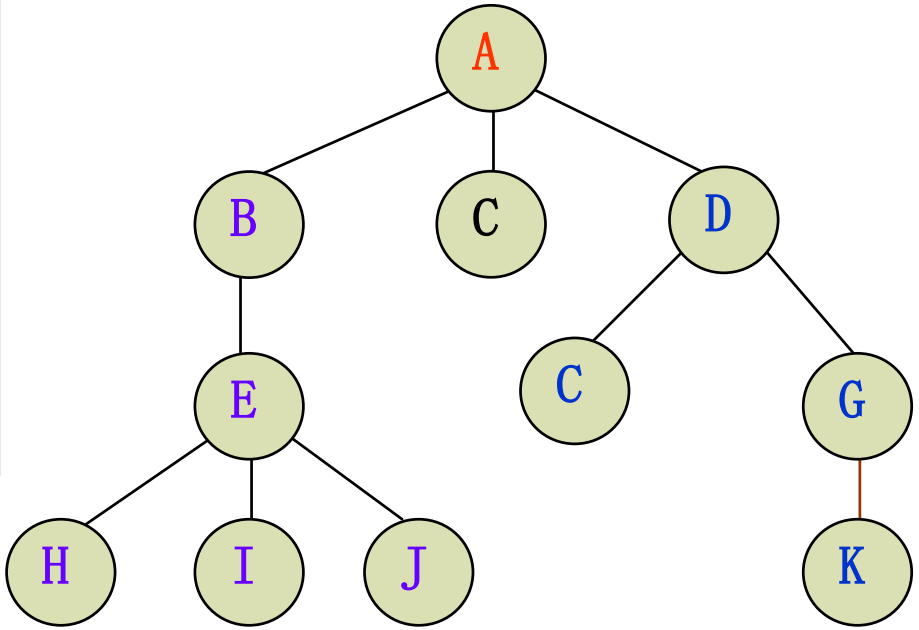
- 树的按层次遍历

层次遍历

自上而下

自左到右

依次访问树中的每个结点



6.6.3 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。

6.6.3 树和森林的遍历

• 树的先根遍历

□先根顺序

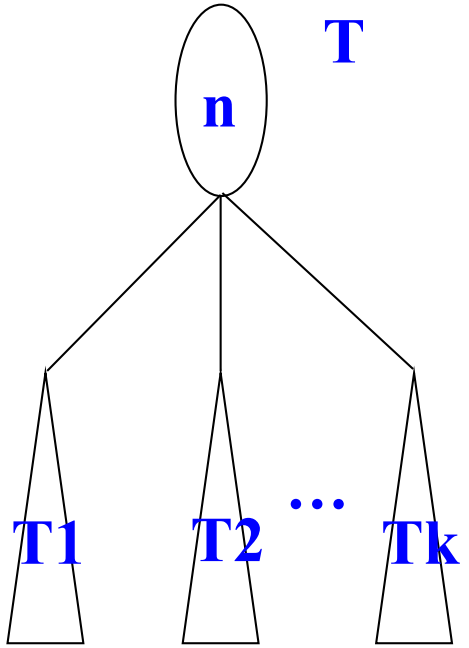
访问根结点；

先根顺序遍历T_1；

先根顺序遍历T_2；

…

先根顺序遍历T_k；



先根图示

6.6.3 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);
           }
}
```

6.6.3 树和森林的遍历

- 树的后根遍历

□后根顺序

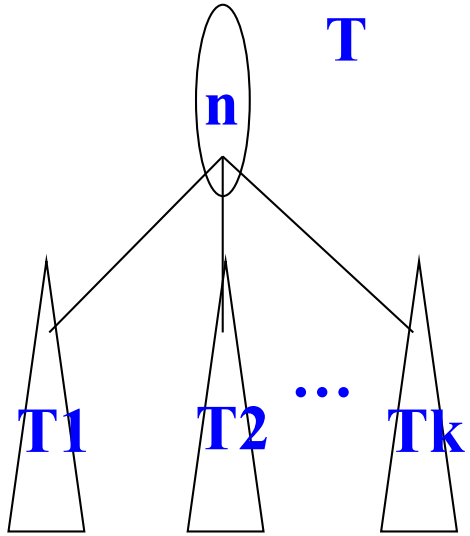
后根顺序遍历T_1；

后根顺序遍历T_2；

…

后根顺序遍历T_k；

访问根结点；



后根图示

6.6.3 树和森林的遍历

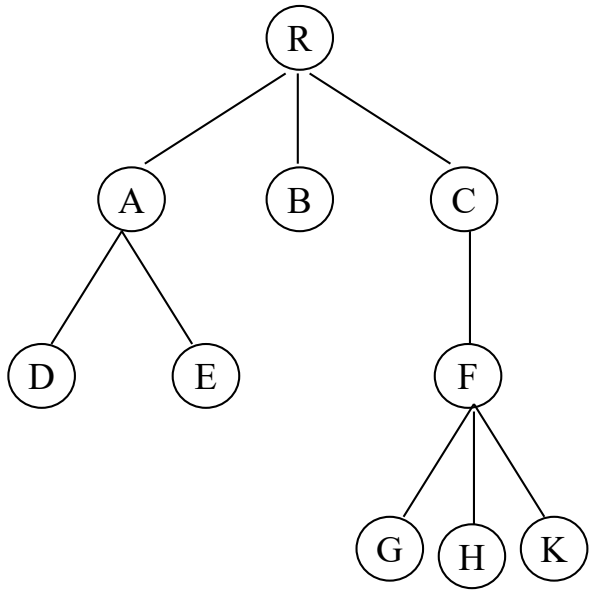
- 树的后根遍历

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);
        postordertre( root-> nextsilbing);
        visit(root->data);
     }
 }
```

6.6.3 树和森林的遍历

- 例：写出如下树的先根、后根序列

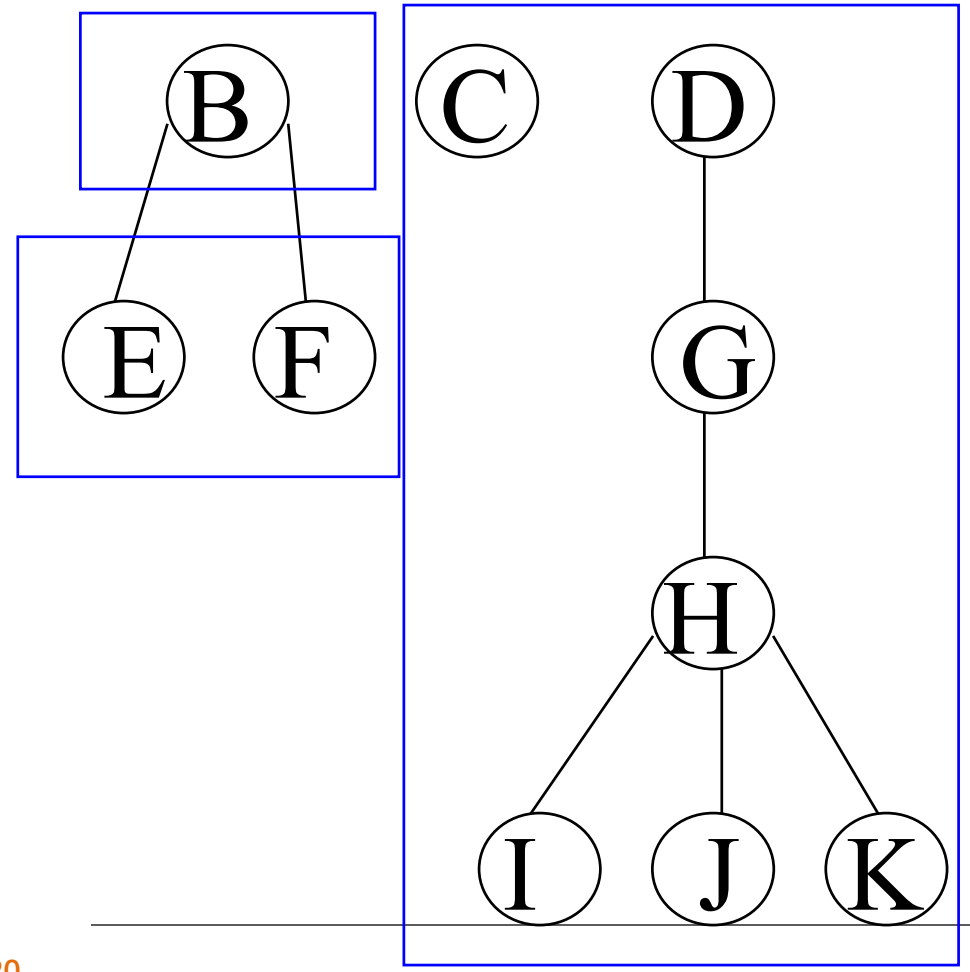


先根：RADEBCFGHK

后根：DEABGHKFCR

6.6.3 树和森林的遍历

- 森林的遍历



森林由三部分构成：

1．森林中第一棵树的根结点；

2．森林中第一棵树的子树森林；

3．森林中其它树构成的森林。

6.6.3 树和森林的遍历

- 森林的先序遍历

□ 先序遍历

若森林不空，则

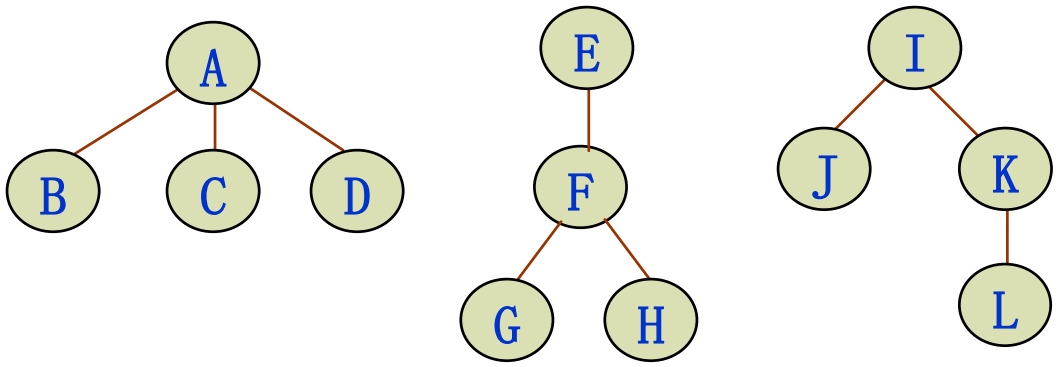
- 访问森林中第一棵树的根结点；
- 先序遍历森林中第一棵树的子树森林；
- 先序遍历森林中（除第一棵树之外）其余树构成的森林。

先序遍历森林

6.6.3 树和森林的遍历

- 森林的先序遍历

例



即：依次从左至右对森林中的每一棵树进行先根遍历。

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

6.6.3 树和森林的遍历

- 森林的后序（中序）遍历

不同教材的说法不一：中序或后序

□ 后序（中序）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。

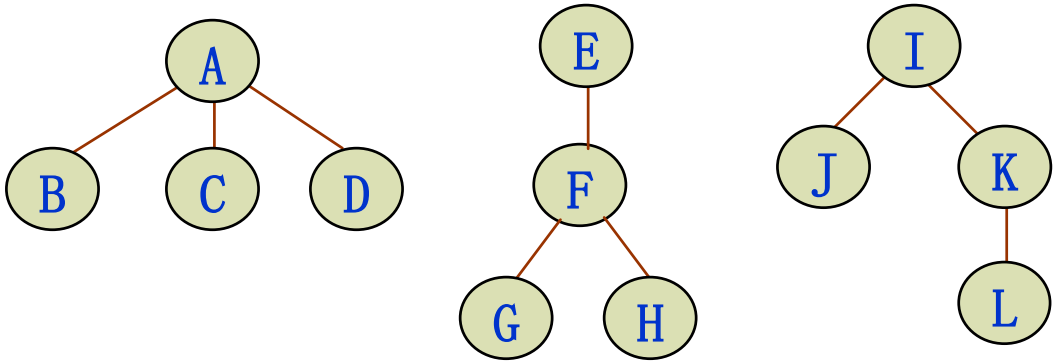
即依次从左至右对森林中的每一棵树进行后根遍历。

中（后）序遍历森林

6.6.3 树和森林的遍历

- 森林的后序遍历

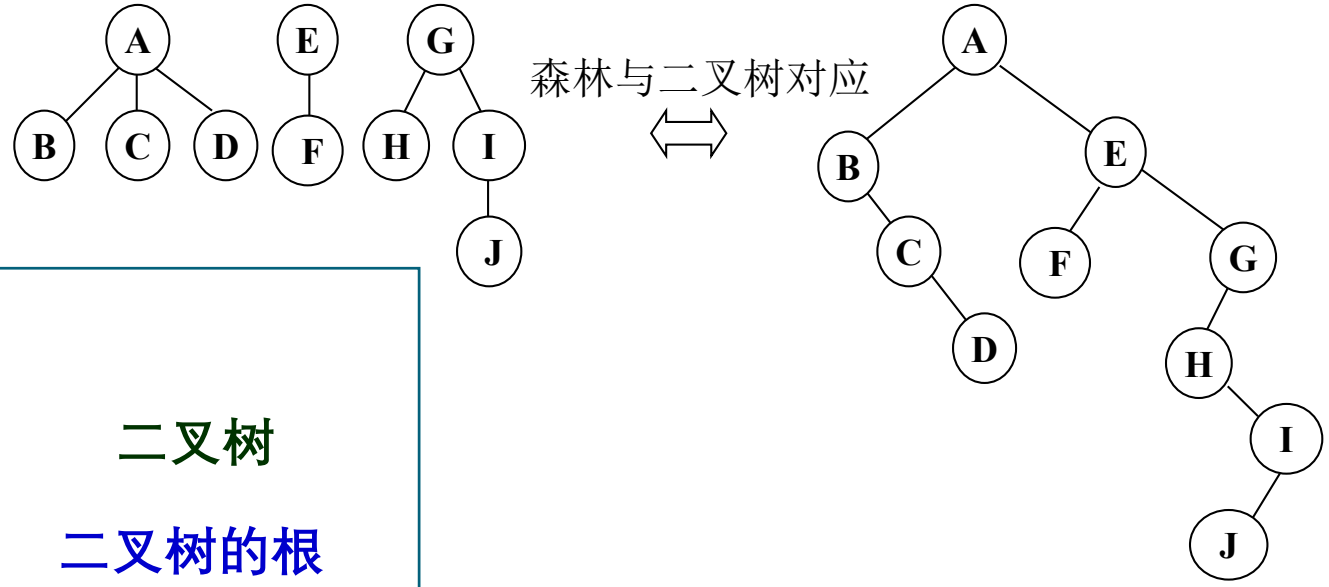
例 森林的遍历



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

6.6.3 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（树）		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；

遍历森林与遍历二叉树的对应关系

遍历森林		遍历相应的二叉树	
先根	访问第一棵树的根； 先根顺序遍历第一棵树的全部子树； 先根顺序遍历其余全部树。	先根	访问树根； 先根顺序遍历左子树； 先根顺序遍历右子树。
后根	后根顺序遍历第一棵树的全部子树 访问第一棵树的根； 后根顺序遍历其余的树。	中根	中根顺序遍历左子树； 访问树根； 中根顺序遍历右子树。

6.6.3 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	后序遍历	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

6.6.3 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
```

6.6 树和森林

6.6.1 树的存储结构

6.6.2 树、森林与二叉树的转换

6.6.3 树和森林的遍历

6.6.4 huffman树及其应用

6.6.4 huffman树及其应用

- **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！

6.6.4 huffman树及其应用

- Huffman教授简介
  - David Huffman教授，美国人，1925-1999
  - 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未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未考试更容易通过。这个大作业促使了Huffman以后算法的诞生。

6.6.4 huffman树及其应用

Huffman树是一类带权路径长度最短的二叉树

---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；
- ……

6.6.4 huffman树及其应用

- Huffman树的相关概念

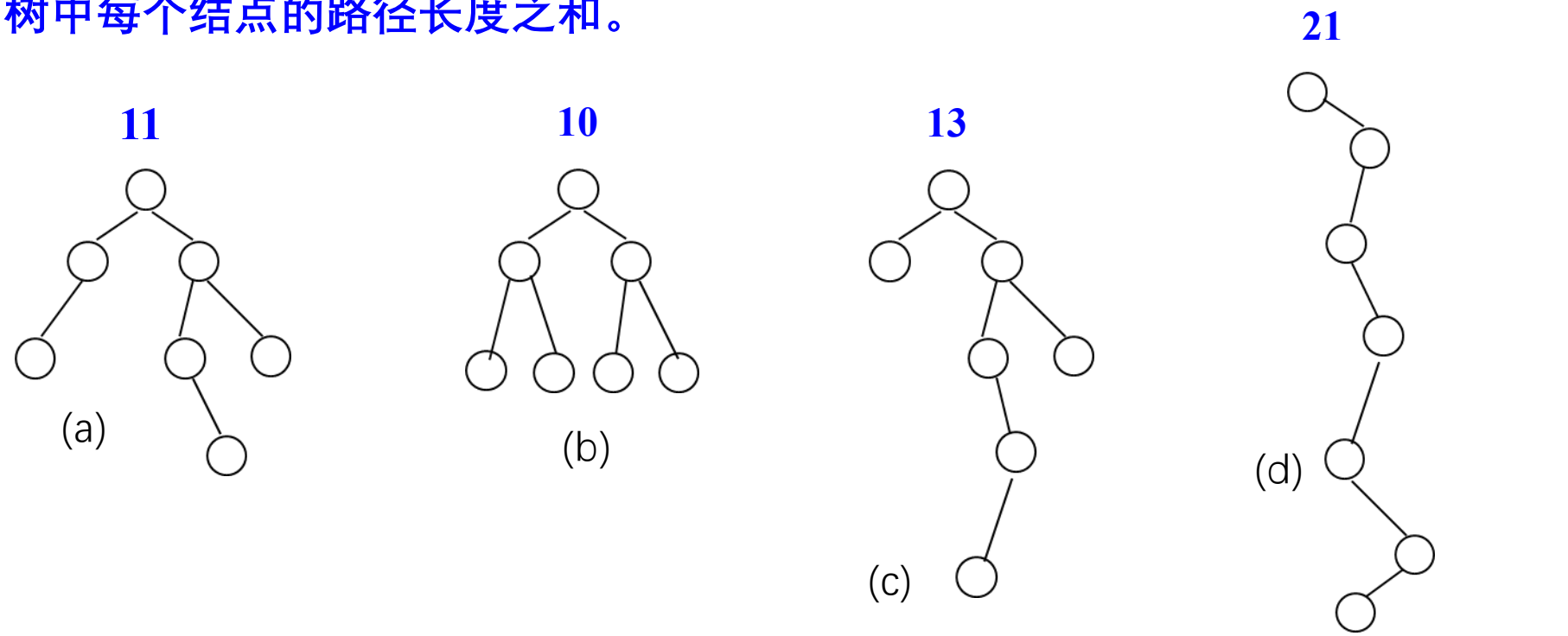
结点的路径长度：

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，二叉树路径长度以与满或完全二叉树相同的高度形态为最小。

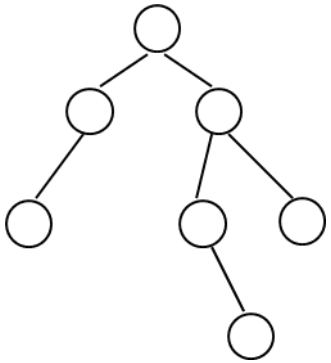


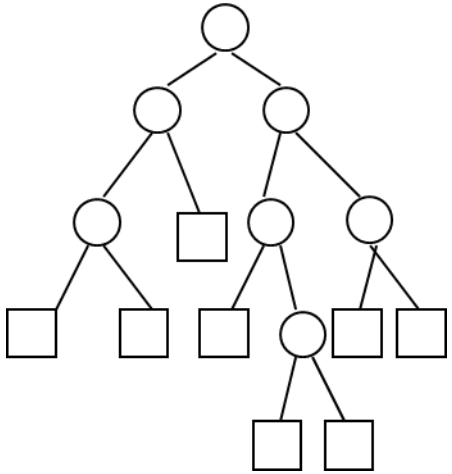
6.6.4 huffman树及其应用

- Huffman树的相关概念

将非满二叉树中，所有度不满2的节点扩充为2，便得到了扩充二叉树。扩充的节点称为外部结点，其余原来的节点称为内部结点。

增长树 { 内结点 ○
外结点 □





如内结点数为 n，则外结点？ n + 1

n0=n2+1

6.6.4 huffman树及其应用

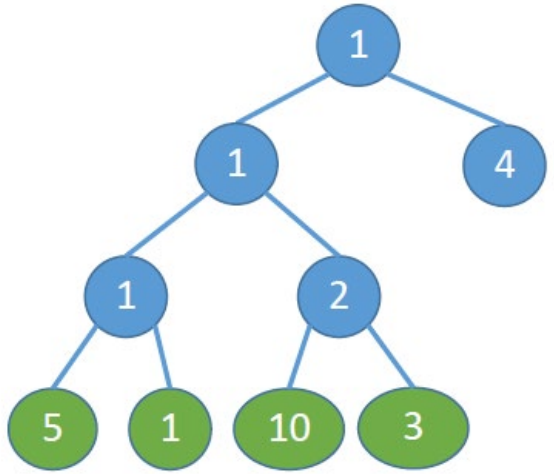
- Huffman树的相关概念

□结点的权：
有某种现实含义的数值（如：表示结点的重要性等）

□节点的带权路径长度：
从根结点到该结点的路径长度与节点上权的乘积

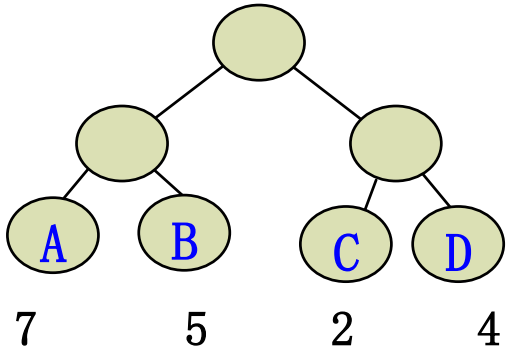
□树的带权路径长度：
树中所有叶子结点的带权路径长度之和

$$WPL(T) = \Sigma w_k l_k \text{（对所有叶子结点）}$$

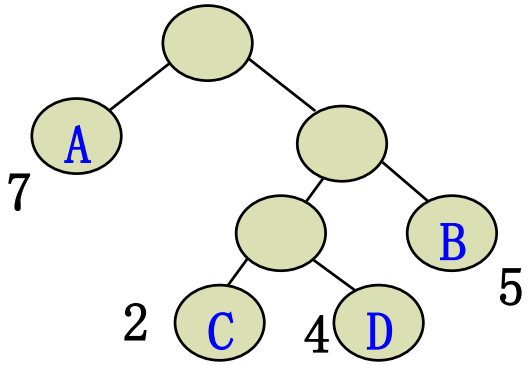


6.6.4 huffman树及其应用

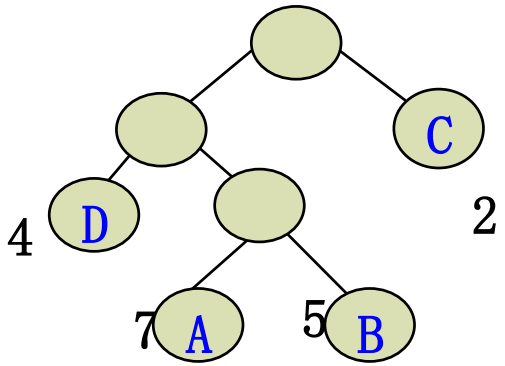
- 带权路径长度



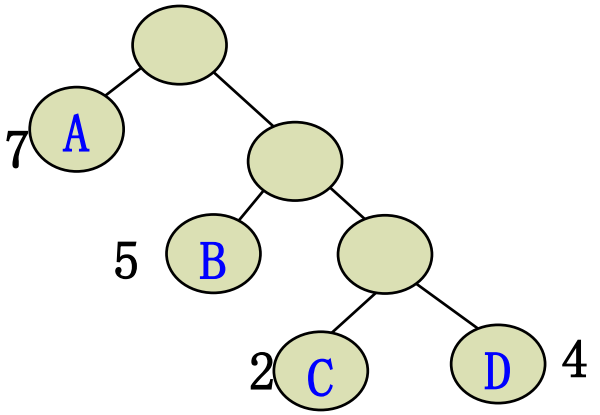
WPL=7*2+5*2+2*2+4*2=36



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



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



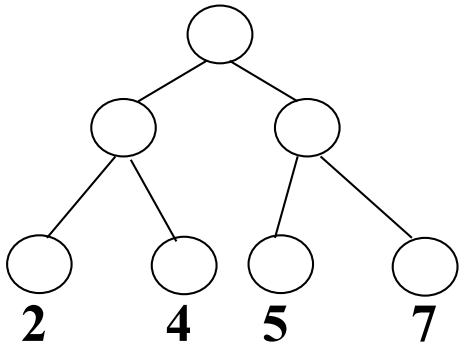
WPL=7*1+5*2+2*3+4*3=35

6.6.4 huffman树及其应用

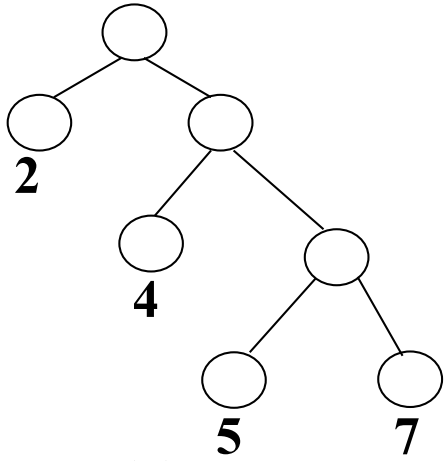
- Huffman树的相关概念

□最优二叉树或哈夫曼树（Huffman）：

含有*n*个带权叶结点的二叉树中，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。



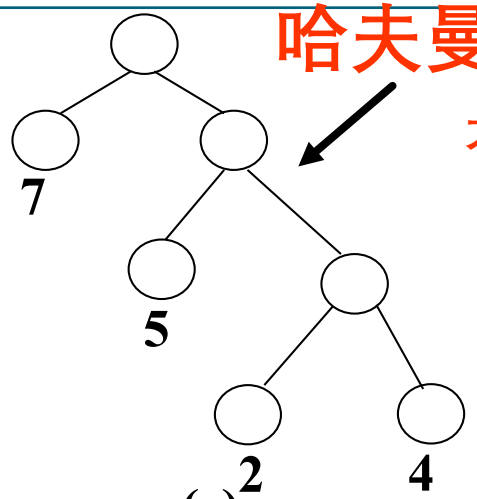
(a)
WPL=36



(b)
WPL=46

哈夫曼树

不止一棵



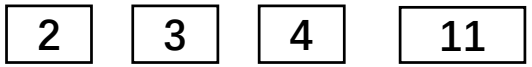
(c)
WPL=35

6.6.4 huffman树及其应用

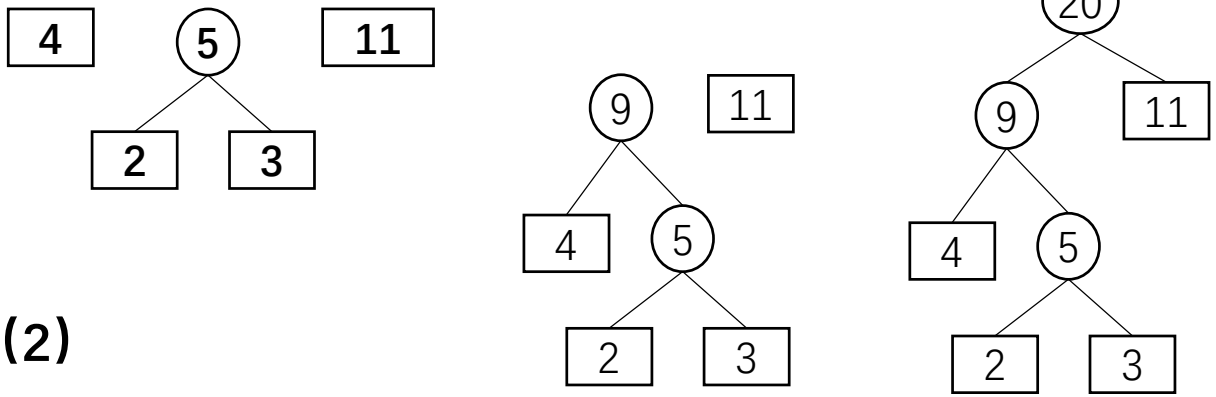
- Huffman树的构造

$w_i = \{2, 3, 4, 11\}$，构造哈夫曼树

（1）将每一个w_i作为一个外结点，并按从小到大顺序排列；
外结点：



（2）选取最小的两个外结点，增加一个内加点，形成一个增长树。
外结点权之和写入内结点，与其他外结点/增长树一起再次排序



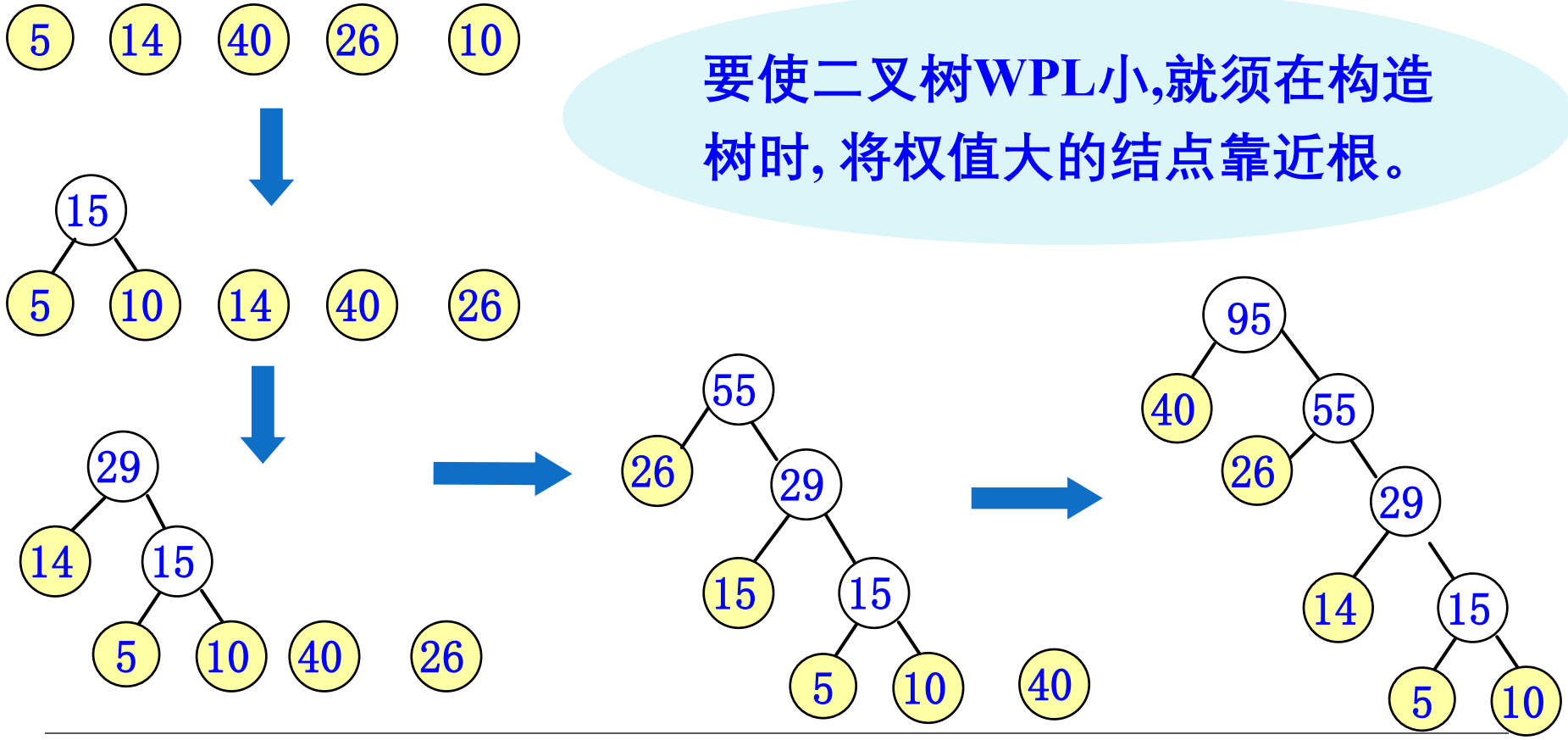
（3）重复步骤（2）

（4）直到最后形成一棵增长树，为哈夫曼树。

$\sum w_i \cdot l_i = 34$

6.6.4 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。

6.6.4 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？

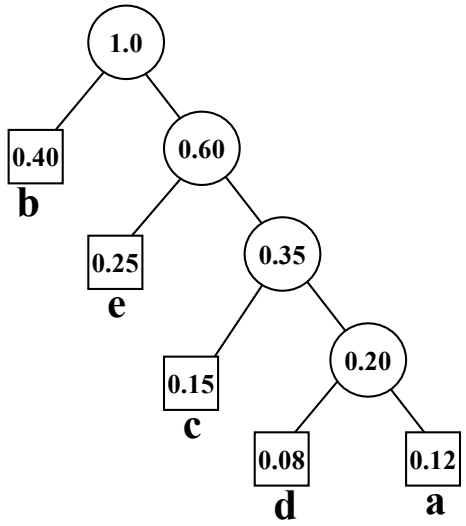
Huffman树的构造过程说明：

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。

6.6.4 huffman树及其应用

- Huffman算法

字符	概率
a	0.12
b	0.40
c	0.15
d	0.08
e	0.25




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

6.6.4 huffman树及其应用

- Huffman算法

字符	概率
a	0.12
b	0.40
c	0.15
d	0.08
e	0.25

	weight	parent	lchild	rchild
0	0.12	-1	-1	-1
1	0.40	-1	-1	-1
2	0.15	-1	-1	-1
3	0.08	-1	-1	-1
4	0.25	-1	-1	-1
5	0.0	-1	-1	-1
6	0.0	-1	-1	-1
7	0.0	-1	-1	-1
8	0.0	-1	-1	-1

```
CreatHuffmanTree(hufmtree tree[])
    {
        int i,j,p1,p2;
        float small1,small2,f;
        for (i=0; i<m; i++) //初始化m=2n-1
        {  tree[i].weight=0.0;
           tree[i].lch=-1;
           tree[i].rch=-1;
           tree[i].parent=-1;
        }
        float w;  //输入权值
        for (i=0; i<n;i++)
           {      scanf("%f",&w);
                  tree[i].weight=w;}
           }
    }
```

6.6.4 huffman树及其应用

- Huffman算法 (续)

	weight	parent	lchild	rchild
0	0.12	-1	-1	-1
1	0.40	-1	-1	-1
2	0.15	-1	-1	-1
3	0.08	-1	-1	-1
4	0.25	-1	-1	-1
5	0.0	-1	-1	-1
6	0.0	-1	-1	-1
7	0.0	-1	-1	-1
8	0.0	-1	-1	-1

```
for(i=n; i<m; i++){  //把合并后的结点放入向量tree[n]~tree[m]
   p1=0;p2=0;              // 两个根结点在向量tree中的下标
   small1=maxval;small2=mavval;   //maxval 是float类型的最大值
   for (j=0;j<i-1;j++) {        //选择权值最小和次小的结点
      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;}
   }//选择结束
```

6.6.4 huffman树及其应用

- Huffman算法（续）

	weight	parent	lchild	rchild
0	0.12	5	-1	-1
1	0.40	8	-1	-1
2	0.15	6	-1	-1
3	0.08	5	-1	-1
4	0.25	7	-1	-1
5	0.20	6	3	0
6	0.35	7	2	5
7	0.60	8	4	6
8	1.00	-1	1	7

```
        tree[p1].parent=i+1;
        tree[p2].parent=i+1;
                    //新生成的结点放在向量tree[i+1]
         tree[i].lch=p1;
        tree[i].rch=p2;
        tree[i].weight= tree[p1]. weight + tree[p1]. weight;
        }
}//huffmantree
```