



燕山大学
YANSHAN UNIVERSITY

Data Structure

数据结构——

第6章 树和二叉树



第6章 树和二叉树

一对多的非线性结构

6.1



树的定义和基本术语

6.2



二叉树

6.3



遍历二叉树和线索二叉树

6.4



树和森林

6.6



赫夫曼树及其应用



■ 本章学习要点

- 熟练掌握二叉树的结构特性，了解相应的证明方法；
- 熟悉二叉树的各种存储结构的特点和适用范围；
- 掌握二叉树遍历的各种算法和操作；
- 理解线索二叉树的实质和具体过程；
- 熟悉树的各种存储结构及特点，掌握树和森林与二叉树的转换方法；
- 掌握建立最优二叉树和设计哈夫曼编码的方法。

第

6.1

部分

树的定义和基本术语



6.1 树的定义和基本术语

树的定义：树是以**分支关系**定义的**层次结构**。

从**分支关系**和**层次结构**角度给出关系描述

层次

1

A

2

B

C

D

3

E

F

G

H

I

J

4

K

L

M

数据对象D

树(Tree)是 $n(n \geq 0)$ 个结点的**有限集**。

数据关系R

在任意一棵非空树中：

递归式定义

- 有且仅有一个特定的称为根(Root)的结点；
- 当 $n > 1$ 时，除根以外的其余结点可分为 $m(m > 0)$ 个**互不相交的有限集** T_1, T_2, \dots, T_m ，其中每一个集合本身又是一棵树，并且称为根的**子树(SubTree)**。



6.1 树的定义和基本术语

树的抽象数据类型定义

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;

若D仅含一个元素, 则R为空集, 否则 $R=\{H\}$, **H是如下二元关系**:

(1) 在D中存在惟一的称为根的数据元素root, **它在关系H下无前驱**;

(2) 若 $D-\{\text{root}\} \neq \emptyset$, 则存在 $D-\{\text{root}\}$ 的一个划分 $D_1, D_2, \dots, D_m (m>0)$, 对任意的 $j \neq k (1 \leq j, k \leq m)$ 有 $D_j \cap D_k = \emptyset$, 且对任意的 $i (1 \leq i \leq m)$, 存在惟一的元素, 有 $\langle \text{root}, x_i \rangle \in H$;

(3) 对应于 $D-\{\text{root}\}$ 的划分, $H-\{\langle \text{root}, x_1 \rangle, \dots, \langle \text{root}, x_m \rangle\}$ 有惟一的一个划分 $H_1, H_2, \dots, H_m (m>0)$, $j \neq k (1 \leq j, k \leq m)$ 有 $H_j \cap H_k = \emptyset$, 且对任意的 $i (1 \leq i \leq m)$, **H_i 是 D_i 上的二元关系, $(D_i, \{H_i\})$ 是一棵符合本定义**的树, 称为root的子树。

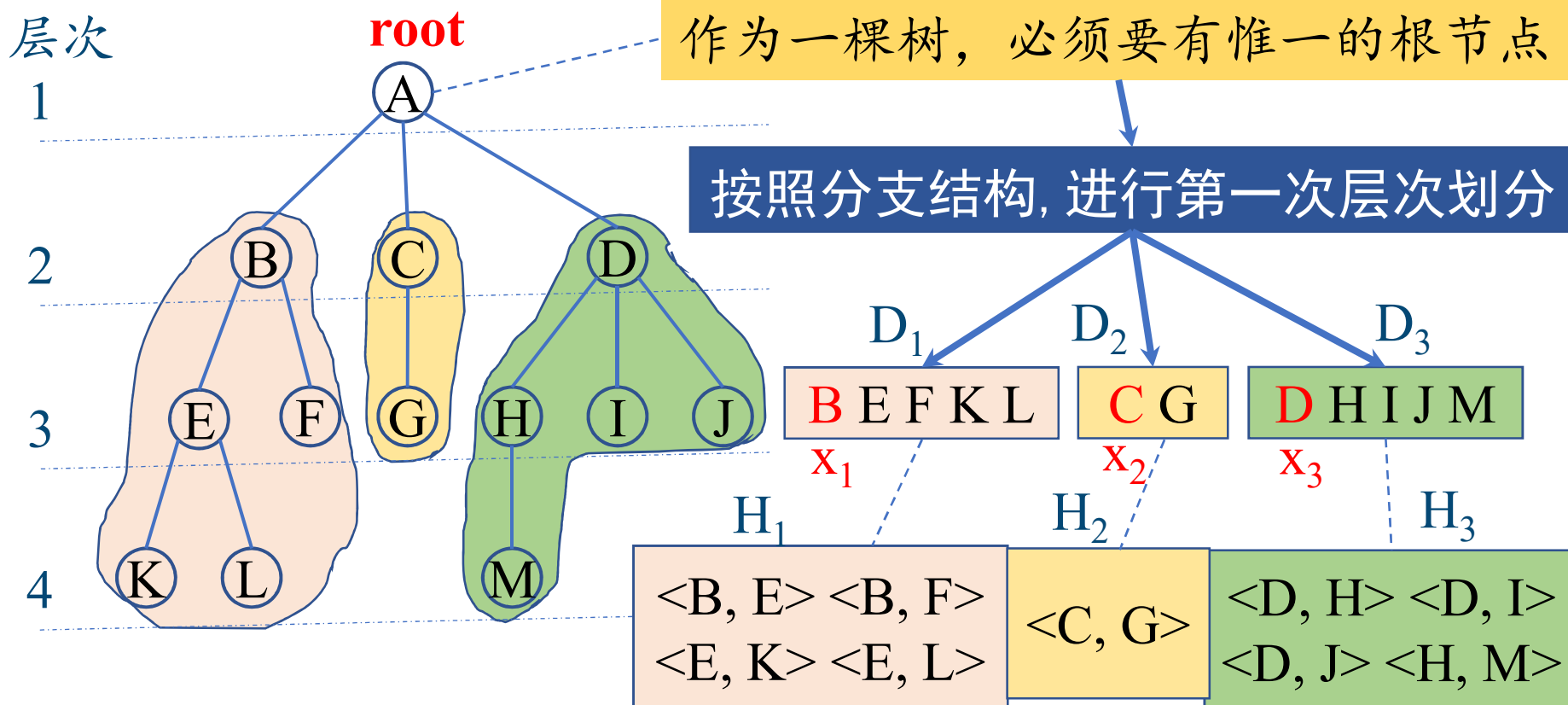
}ADT Tree



6.1 树的定义和基本术语

树的抽象数据类型定义

从分支关系和层次结构角度
给出关系描述



$$D = \{\text{root}\} \cup D_1 \cup D_2 \cup D_3$$

$$H = \{\langle \text{root}, x_1 \rangle, \langle \text{root}, x_2 \rangle, \langle \text{root}, x_3 \rangle\} \cup H_1 \cup H_2 \cup H_3$$



6.1 树的定义和基本术语

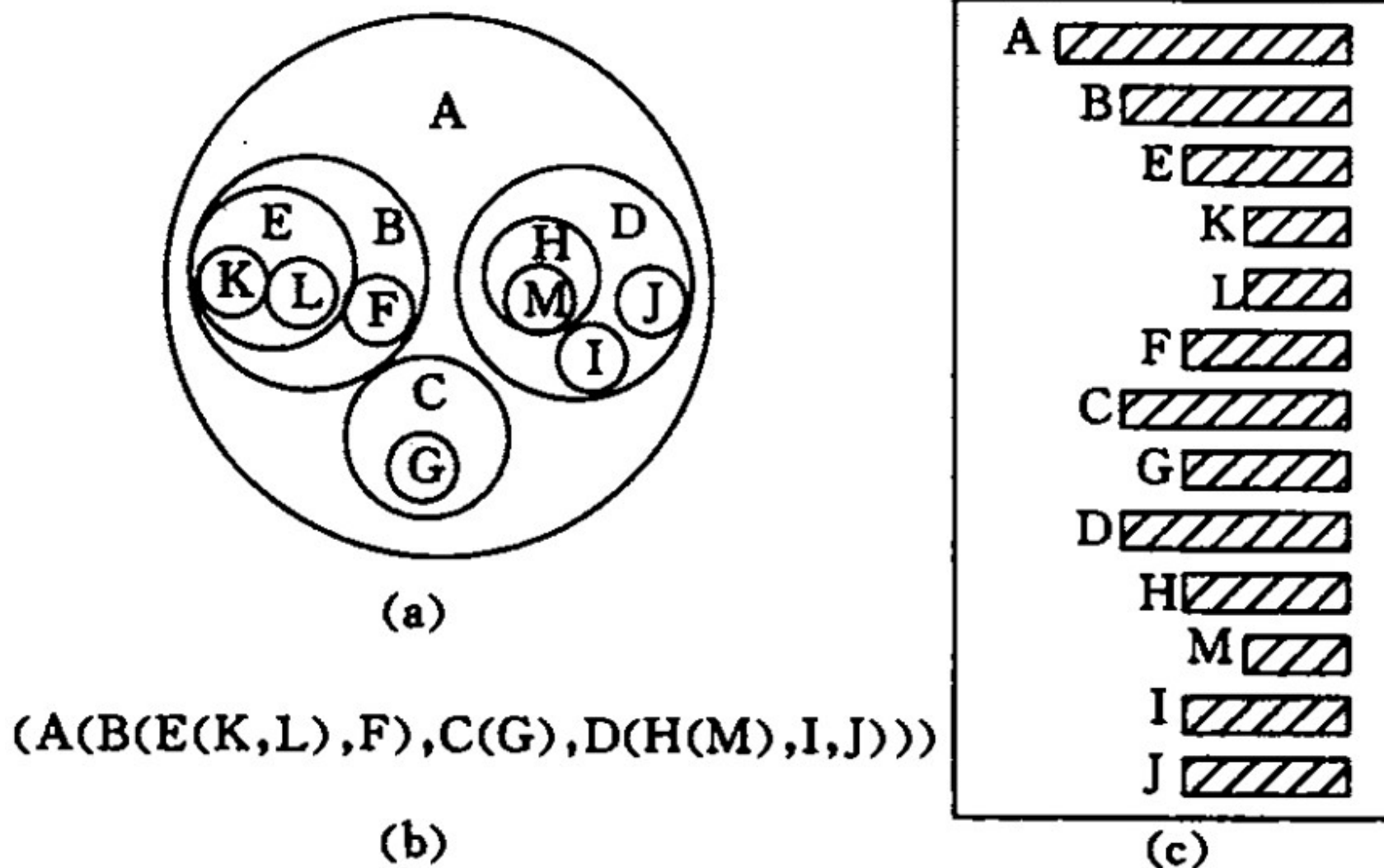
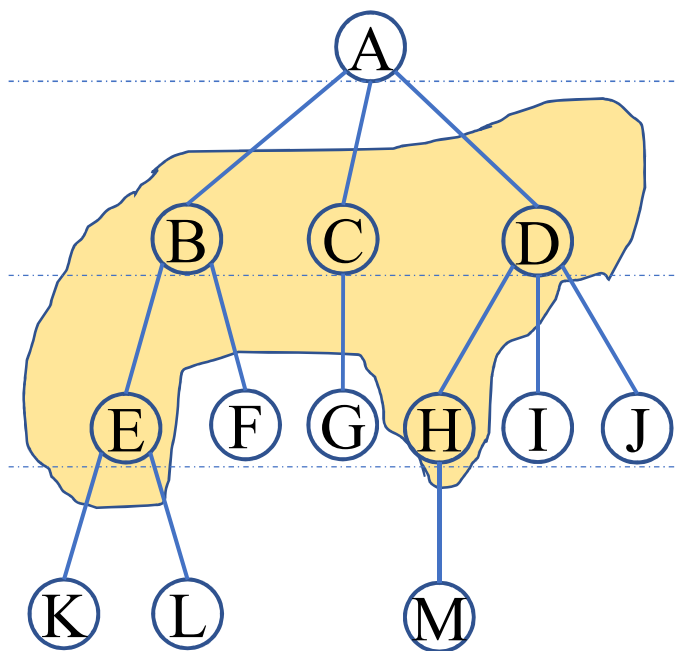


图 6.2 树的其他 3 种表示法

6.1 树的定义和基本术语

基本术语



结点

结点的度

叶子结点

终端结点

分支结点

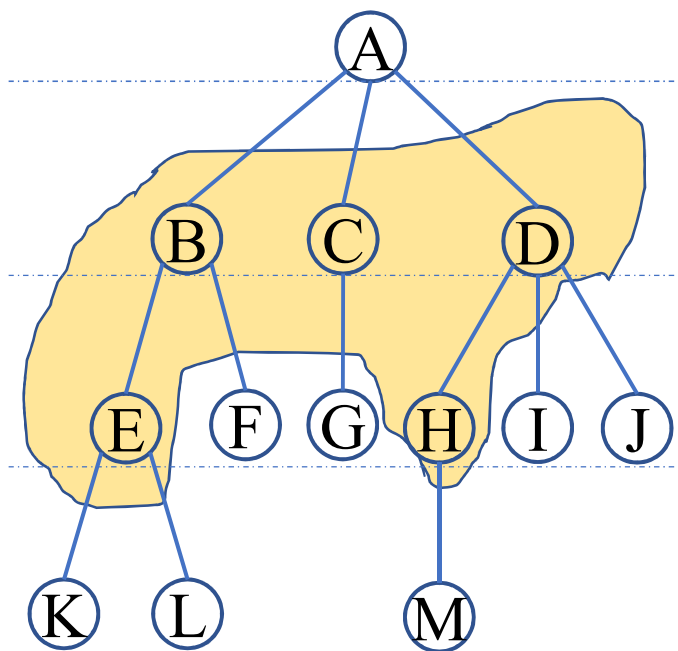
非终端结点

内部结点

树的度

6.1 树的定义和基本术语

基本术语



孩子

双亲

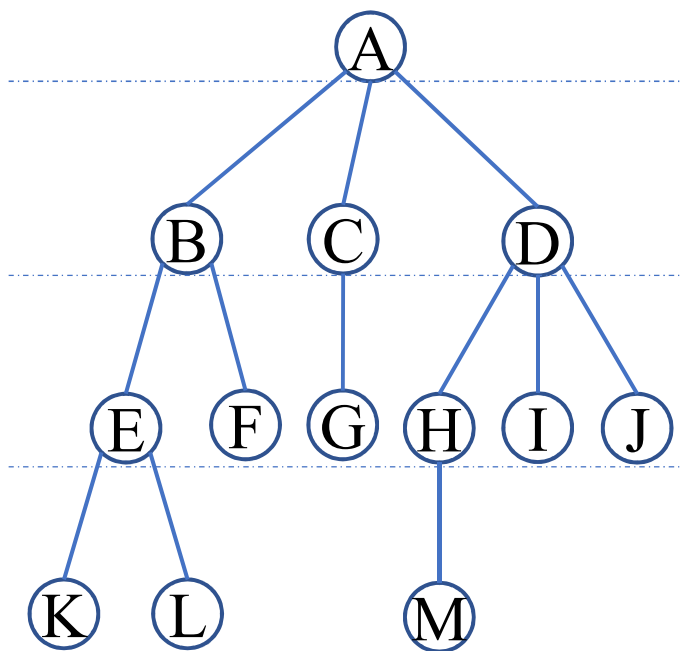
兄弟

祖先

子孙

6.1 树的定义和基本术语

基本术语



结点层次

堂兄弟

树的深度

有序树

无序树

森林

6.1 树的定义和基本术语

树的抽象数据类型定义

ADT Tree{

 基本操作P:

InitTree(&T)

 操作结果: 构造空树T

DestroyTree(&T)

 初始条件: 树T存在

 操作结果: 销毁树T

CreateTree(&T, definition)

 初始条件: definition给出树T的定义

 操作结果: 按definition构造树T

ClearTree (&T)

 初始条件: 树T存在

 操作结果: 将树T清为空树

TreeEmpty (T)

 初始条件: 树T存在

 操作结果: 若T为空树, 则返回TRUE, 否则FALSE

}ADT Tree



6.1 树的定义和基本术语

树的抽象数据类型定义

ADT Tree{

 基本操作P:

TreeDepth(T)

 初始条件: 树T存在

 操作结果: 返回树T的深度

Root(T)

 初始条件: 树T存在

 操作结果: 返回树T的根

Value(T, cur_e)

 初始条件: 树T存在, cur_e是T中某个结点

 操作结果: 返回cur_e的值

Assign(T, cur_e, value)

 初始条件: 树T存在, cur_e是T中某个结点

 操作结果: 结点cur_e赋值为value

}ADT Tree



6.1 树的定义和基本术语

ADT Tree{

Parent(T, cur_e)

初始条件：树T存在，cur_e是T中某个结点

操作结果：若cur_e是T的非根结点，则返回它的双亲，否则函数值为“空”

LeftChild(T, cur_e)

初始条件：树T存在，cur_e是T中某个结点

操作结果：若cur_e是T的非叶子节点，则返回它的最左孩子，否则返回“空”

RightSibling(T, cur_e)

初始条件：树T存在，cur_e是T中某个结点

操作结果：若cur_e有右兄弟，则返回它的右兄弟，否则函数值为“空”

}ADT Tree



6.1 树的定义和基本术语

ADT Tree{

InsertChild(&T, &p, i, c)

初始条件：树T存在，p指向T中某个结点， $1 \leq i \leq p$ 所指结点的度+1，非空树c与T不相交

操作结果：插入c为T中p所指结点的第i棵子树

DeleteChild(&T, &p, i)

初始条件：树T存在，p指向T中某个结点， $1 \leq i \leq p$ 所指结点的度

操作结果：删除T中p所指结点的第i棵子树

TraverseTree(T, visit())

初始条件：树T存在，visit()是对结点操作的应用函数

操作结果：按照某种次序对T中每个结点调用visit()一次且至多一次。一旦visit()失败，则操作失败

}ADT Tree



第 6.2 部分

二叉树

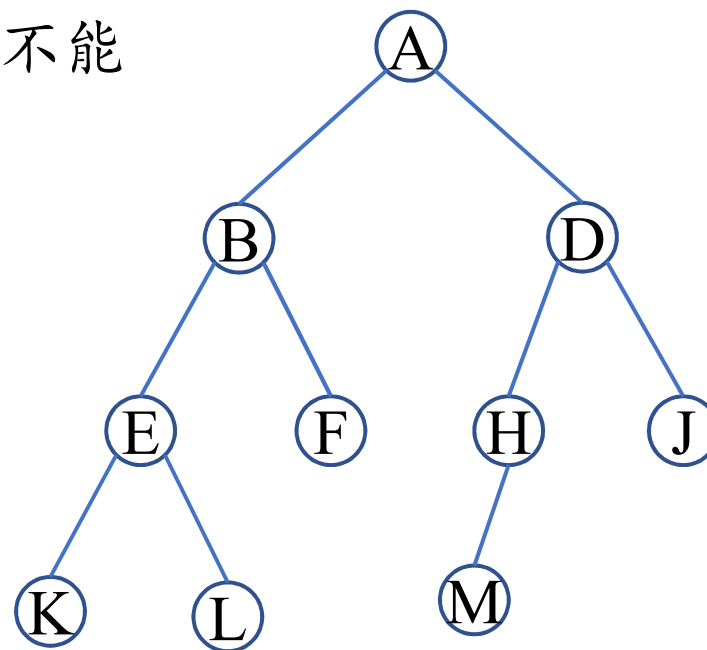
6.2 二叉树

二叉树的定义

二叉树(Binary Tree) 是**另一种**树形结构

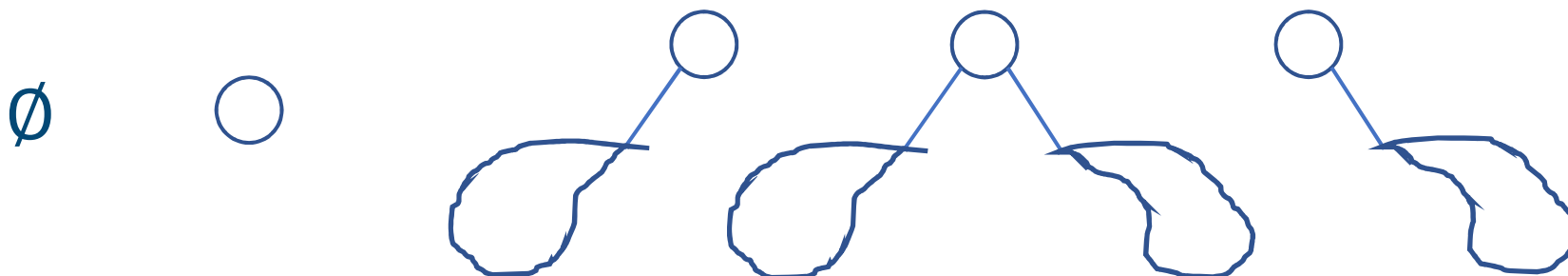
特点是每个结点**最多只有两棵**子树(即二叉树中不存在度 >2 的结点)

二叉树的子树有**左右之分**，其次序不能任意颠倒



6.2 二叉树

二叉树的5种基本形态



ADT BinaryTree{

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 $D=\emptyset$ ，则 $R=\emptyset$ ，则称为空二叉树；

若 $D\neq\emptyset$ ，否则 $R=\{H\}$ ，H是如下二元关系集：

(1)在D中存在惟一的称为根的数据元素root，它在关系H下无前驱；

(2)若 $D-\{\text{root}\} \neq \emptyset$ ，则存在 $D-\{\text{root}\}=\{D_l, D_r\}$ ，且 $D_l \cap D_r = \emptyset$ ；

(3)若 $D_l \neq \emptyset$ ，则 D_l 中存在惟一的元素 x_l ， $\langle \text{root}, x_l \rangle \in H$ ，且存在 D_l 上的关系 $H_l \subset H$ ；

若 $D_r \neq \emptyset$ ，则 D_r 中存在惟一的元素 x_r ， $\langle \text{root}, x_r \rangle \in H$ ，且存在 D_r 上的关系 $H_r \subset H$ ； $H=\{\langle \text{root}, x_l \rangle, \langle \text{root}, x_r \rangle, H_l, H_r\}$ ；

(4)($D_l, \{H_l\}$)是一棵符合本定义的二叉树，称为根的左子树，

($D_r, \{H_r\}$)是一棵符合本定义的二叉树，称为根的右子树。

}ADT BinaryTree



6.2 二叉树

ADT BinaryTree{

 基本操作P:

LeftChild(T, e)

 初始条件: 二叉树T存在, e是T中某个结点

 操作结果: 返回e的左孩子, 若e无左孩子则返回“空”

RightChild(T, e)

 初始条件: 二叉树T存在, e是T中某个结点

 操作结果: 返回e的右孩子, 若e无右孩子则返回“空”

LeftSibling(T, e)

 初始条件: 二叉树T存在, e是T中某个结点

 操作结果: 返回e的左兄弟, 若e是T的左孩子或无左兄弟, 则返回“空”

RightSibling(T, e)

 初始条件: 二叉树T存在, e是T中某个结点

 操作结果: 返回e的右兄弟, 若e是T的右孩子或无右兄弟, 则返回“空”

 }ADT BinaryTree



6.2 二叉树

ADT BinaryTree{

 基本操作P:

InsertChild(T, p, LR, c)

 初始条件: 二叉树T存在, p指向T中某个结点, LR为0或1, **非空二叉树c与T不相交且右子树为空**

 操作结果: 根据LR为0或1, 插入c为T中p所指结点的左或右子树, p所指结点的**原有左或右子树则成为c的右子树**

DeleteChild(T, p, LR)

 初始条件: 二叉树T存在, p指向T中某个结点, LR为0或1

 操作结果: 根据LR为0或1, 删除T中p所指结点的左或右子树

}ADT BinaryTree



6.2 二叉树

ADT BinaryTree{

PreOrderTraverse(T, visit())

 初始条件：二叉树T存在，visit()是对结点操作的应用函数

 操作结果：**先序遍历**T，对T中每个结点调用函数visit()一次且仅一次，一旦visit()失败，则操作失败

InOrderTraverse(T, visit())

 初始条件：二叉树T存在，visit()是对结点操作的应用函数

 操作结果：**中序遍历**T，对T中每个结点调用函数visit()一次且仅一次，一旦visit()失败，则操作失败

}ADT BinaryTree



6.2 二叉树

ADT BinaryTree{

PostOrderTraverse(T, visit())

 初始条件：二叉树T存在，visit()是对结点操作的应用函数

 操作结果：**后序遍历**T，对T中每个结点调用函数visit()一次且仅一次，一旦visit()失败，则操作失败

LevelOrderTraverse(T, visit())

 初始条件：二叉树T存在，visit()是对结点操作的应用函数

 操作结果：**层序遍历**T，对T中每个结点调用函数visit()一次且仅一次，一旦visit()失败，则操作失败

}ADT BinaryTree



6.2 二叉树

二叉树的性质

性质1 在二叉树的第 i 层上至多有 2^{i-1} 个结点($i \geq 1$)

利用归纳法证明:

当 $i=1$ 时, 则 $2^{1-1} = 2^0 = 1$, 成立;

假设对所有的第 j 层($1 \leq j < i$), 命题成立;

由归纳假设: 第 $i-1$ 层上至多有 2^{i-2} 个结点;

因此, 在第 i 层上最多有 $2 \times 2^{i-2} = 2^{i-1}$ 个结点



6.2 二叉树

二叉树的性质

性质2 深度为k的二叉树至多有 2^k-1 个结点($k \geq 1$)

由性质1可知，二叉树的第i层最多有 2^{i-1} 个结点，

从而，深度为k的二叉树的最大结点数为：

$$\sum_{i=1}^k 2^{i-1} = 2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$$



6.2 二叉树

二叉树的性质

性质3

对任一棵二叉树T，如果其终端结点数为 n_0 ，度为2的结点数为 n_2 ，则 $n_0 = n_2 + 1$

$$n = n_0 + n_1 + n_2 \quad (1)$$

假设T中总分支数量为B，则

$$B = n - 1 \quad (2)$$

由于分支都是由度为1和2的结点射出的，则

$$B = n_1 + 2n_2 \quad (3)$$

结合(2)和(3)则有

$$n = n_1 + 2n_2 + 1 \quad (4)$$

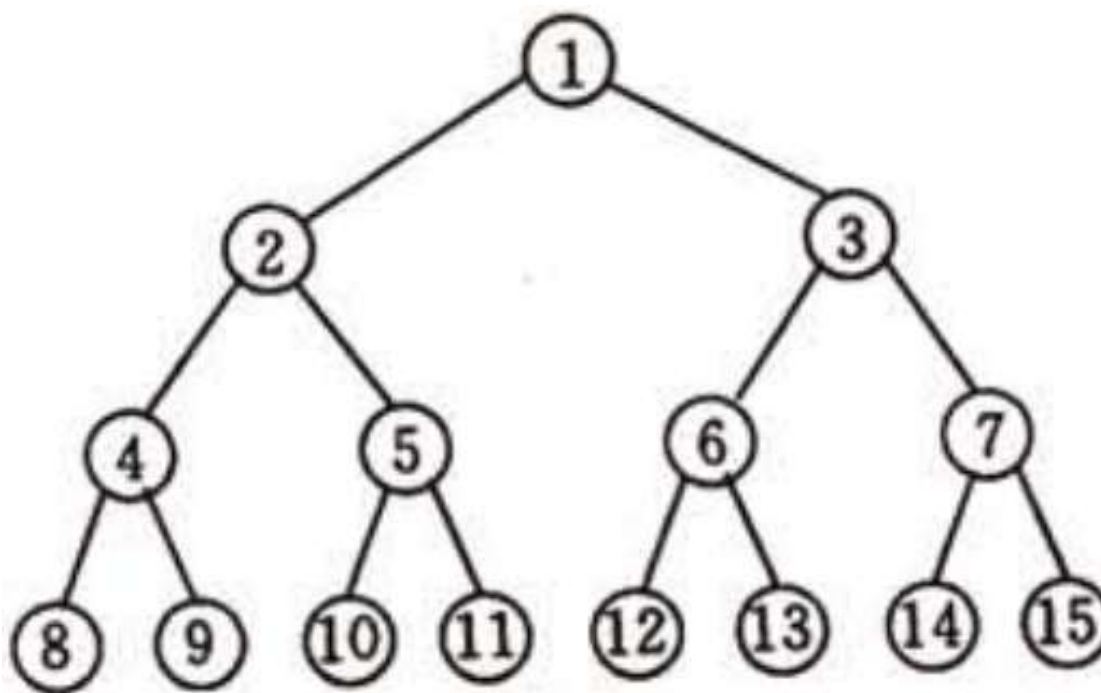
结合(1)和(4)可得 $n_0 = n_2 + 1$



6.2 二叉树

满二叉树和完全二叉树

满二叉树 一棵深度为 k 且有 2^k-1 个结点的二叉树



6.2 二叉树

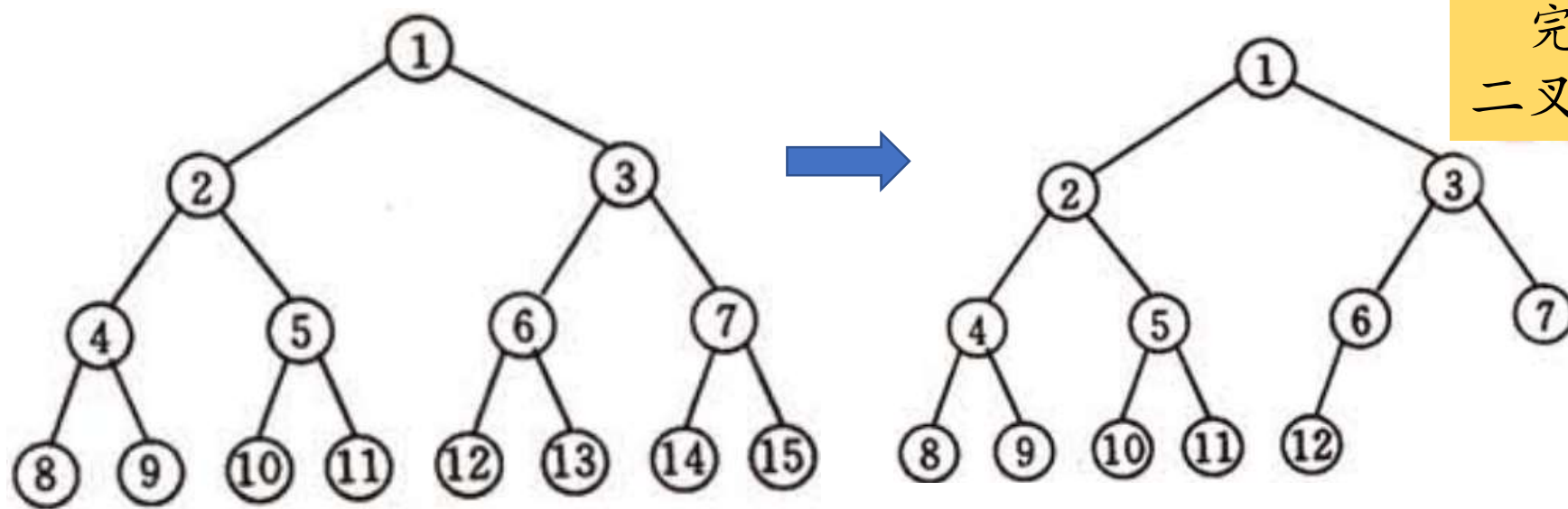
满二叉树和完全二叉树

完全二叉树

深度为 k ，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从1至 n 的结点一一对应时

叶子结点只可能在层次最大的两层上出现

对任一结点，若其右分支下的子孙最大层次为 l ，则其左分支下的子孙的最大层次必为 l 或 $l+1$



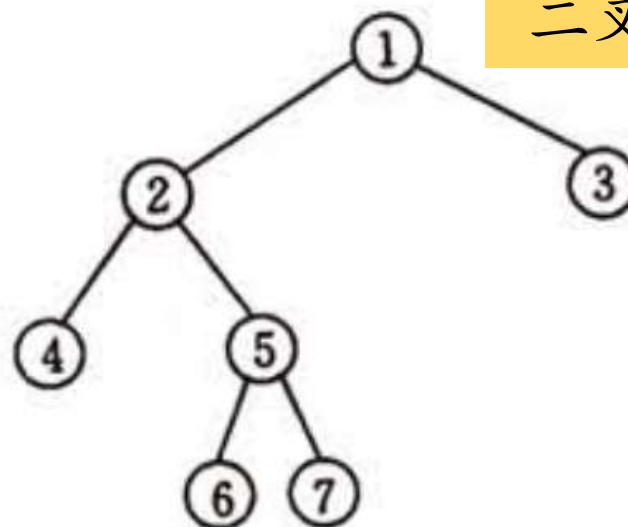
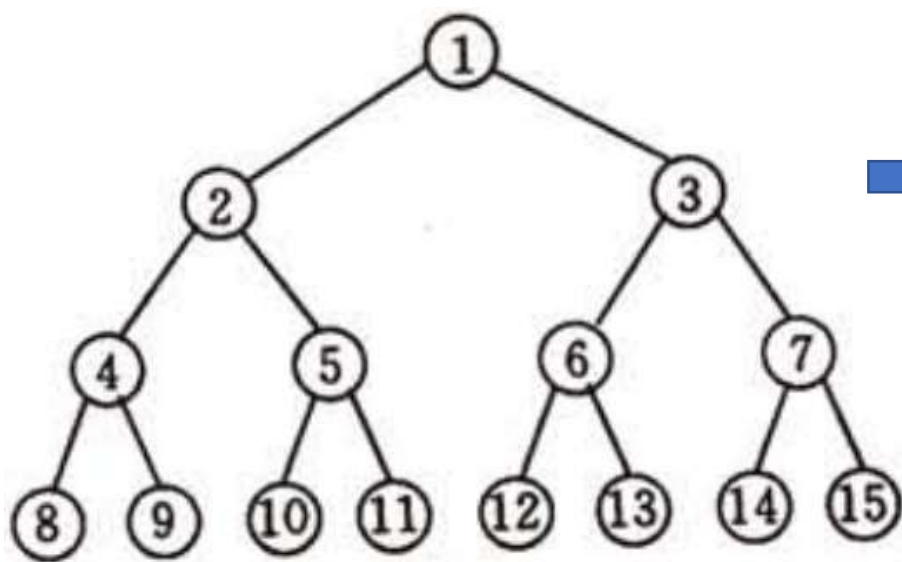
完全
二叉树

6.2 二叉树

满二叉树和完全二叉树

完全二叉树

深度为 k ，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从1至 n 的结点一一对应时



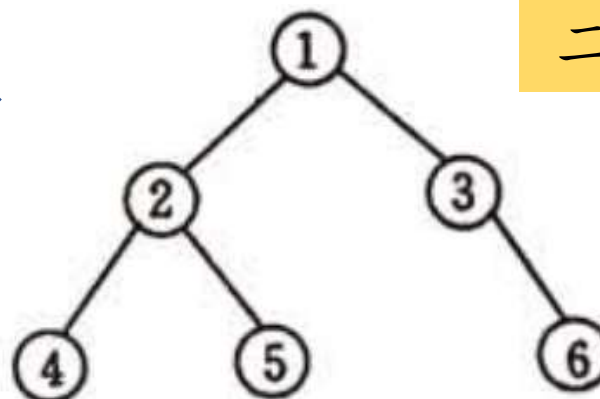
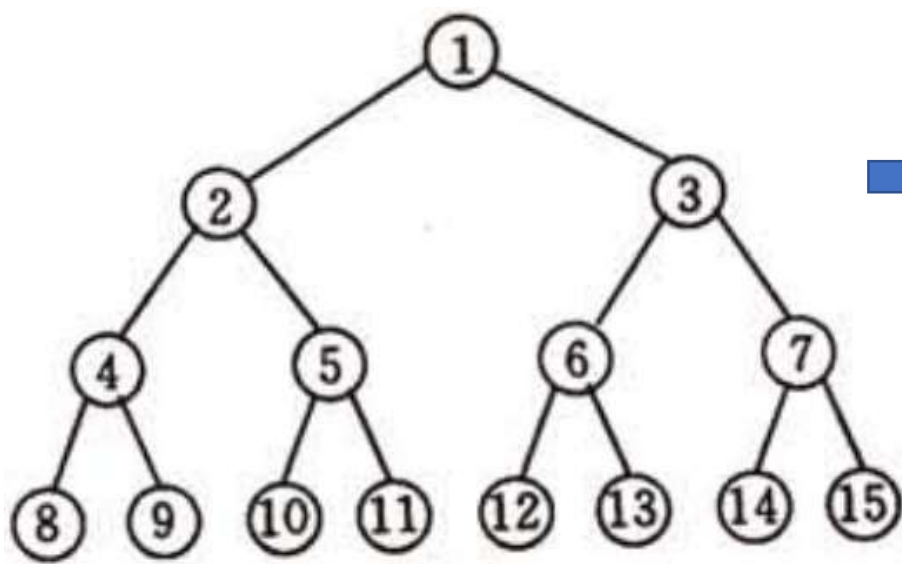
非完全
二叉树

6.2 二叉树

满二叉树和完全二叉树

完全二叉树

深度为 k ，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从1至 n 的结点一一对应时



非完全
二叉树

6.2 二叉树

二叉树的性质

性质4 具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$

假设完全二叉树的深度为 k

可得 $2^{k-1}-1 < n \leq 2^k-1$

$$2^{k-1} \leq n < 2^k$$

$$k-1 \leq \log_2 n < k$$

$$\text{即 } k-1 = \lfloor \log_2 n \rfloor$$

$$\text{所以 } k = \lfloor \log_2 n \rfloor + 1$$



6.2 二叉树

二叉树的性质

性质5

如果对一棵有 n 个结点的完全二叉树(其深度为 $\lfloor \log_2 n \rfloor + 1$)的结点按层序编号(从第1层到第 $\lfloor \log_2 n \rfloor + 1$ 层, 每层从左到右), 则对任一结点 i ($1 \leq i \leq n$), 有

- (1) 如果 $i=1$, 则结点 i 是二叉树的根, 无双亲; 如果 $i>1$, 则其双亲 $\text{PARENT}(i)$ 是结点 $\lfloor i/2 \rfloor$;
- (2) 如果 $2i>n$, 则结点 i 无左孩子(结点 i 为叶子结点), 否则其左孩子 $\text{LCHILD}(i)$ 是结点 $2i$;
- (3) 如果 $2i+1>n$, 则结点 i 无右孩子, 否则其右孩子 $\text{RCHILD}(i)$ 是结点 $2i+1$;



6.2 二叉树

二叉树的性质

性质5 (2) 结点 i 左孩子是结点 $2i$;
(3) 结点 i 右孩子是结点 $2i+1$;

归纳法:

$i=1$, 1的左孩子为 $2=2*1$, 右孩子为 $3=2*1+1$

(1)证明每层的第一个结点,

设第 j 层的第一个结点为 $i=2^{j-1}$,则 i 的左孩子为第 $j+1$ 层的第一个结点为 $2^j=2*2^{j-1}=2*i$

i 的右孩子为第 $j+1$ 层的第二个结点为 $2*i+1$

(2)对于层中任意结点, 假设 i 的左孩子为 $2*i$, 右孩子为 $2*i+1$

则第 $i+1$ 个的结点的左孩子为 $2*i+1+1=2*(i+1)$, 右孩子为 $2*i+3=2*(i+1)+1$



6.2 二叉树

二叉树的性质

性质5 (1) 结点 i 的双亲是结点 $\lfloor i/2 \rfloor$

对于结点 i , 设 i 的双亲为 j

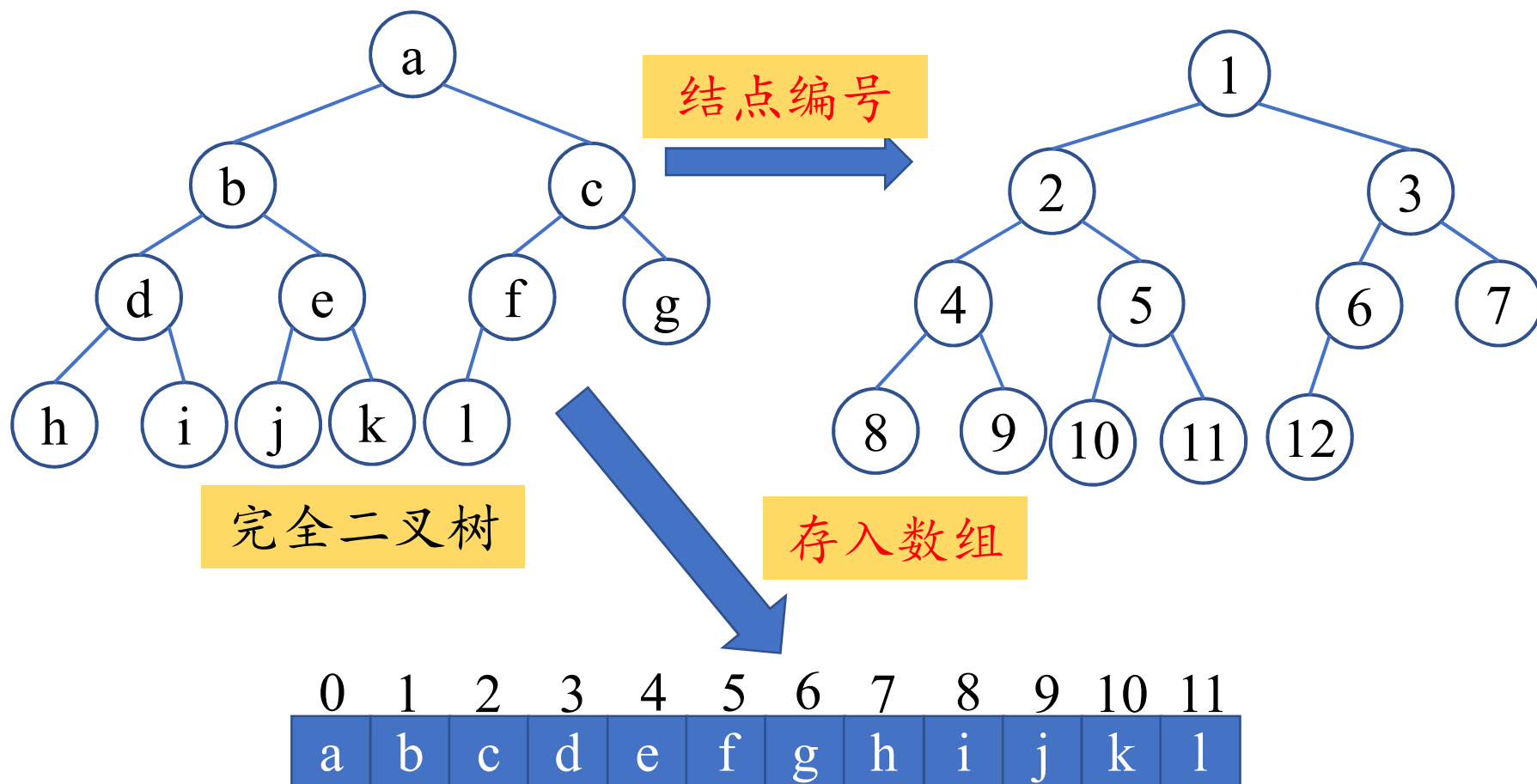
若 i 为 j 的左孩子, 则 $i=2*j$, 因此, $j= \lfloor i/2 \rfloor$

若 i 为 j 的右孩子, 则 $i=2*j+1$, 因此, $j= \lfloor i/2 \rfloor$

6.2 二叉树

二叉树的存储——顺序存储结构

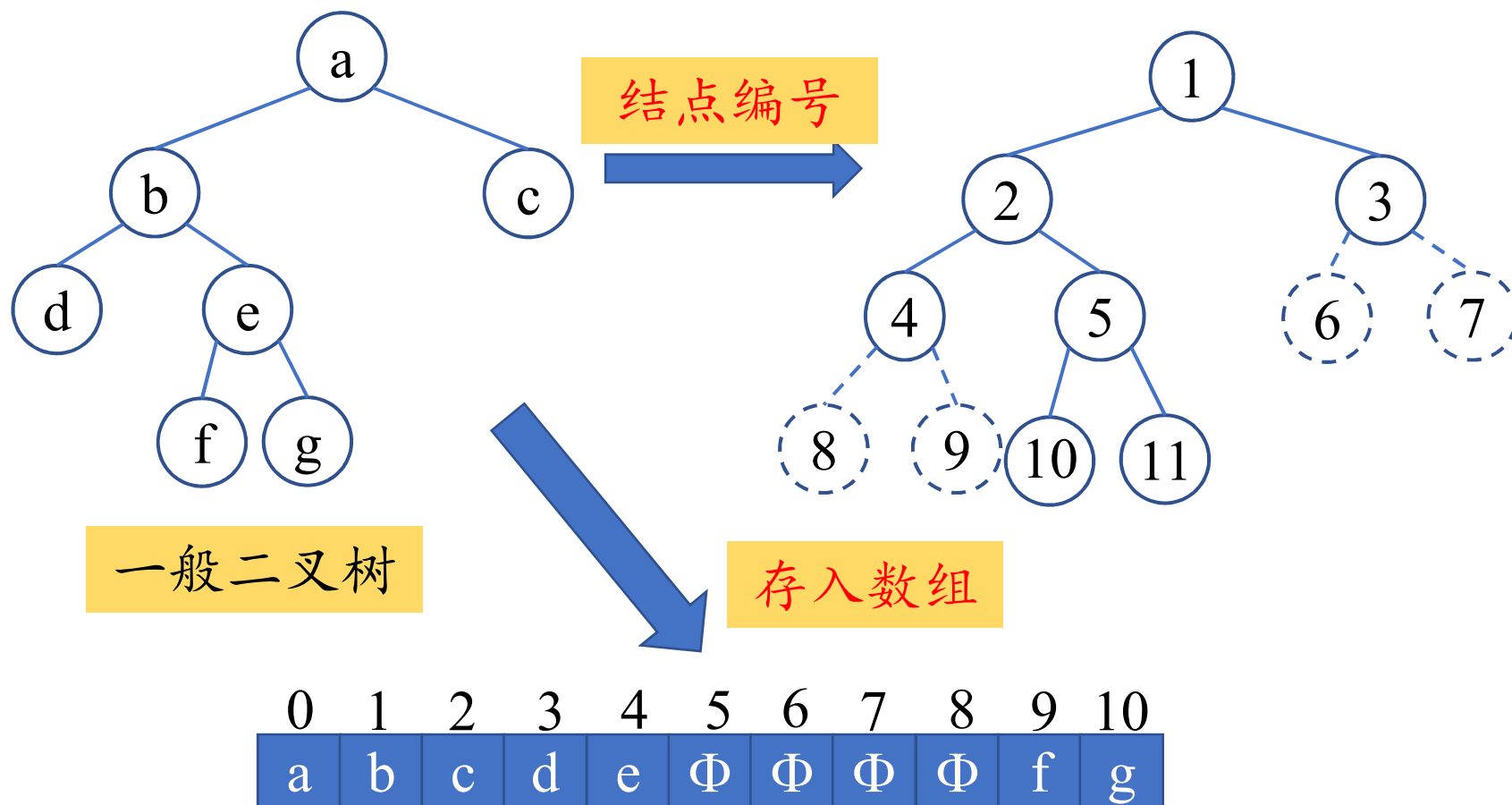
编号为 i 的结点存入下标为 $i-1$ 的数组分量



6.2 二叉树

二叉树的存储——顺序存储结构

编号为 i 的结点存入下标为 $i-1$ 的数组分量



6.2 二叉树

二叉树的存储——顺序存储结构

```
#define MAX_TREE_SIZE 100 // 二叉树的最大结点数  
  
typedef TElemType SqBiTree[MAX_TREE_SIZE];  
  
// 0号单元存储根结点  
  
SqBiTree bt;
```

深度为 k 的二叉树最大元素个数为 2^k-1 ,
即需要长度为 2^k-1 的数组!



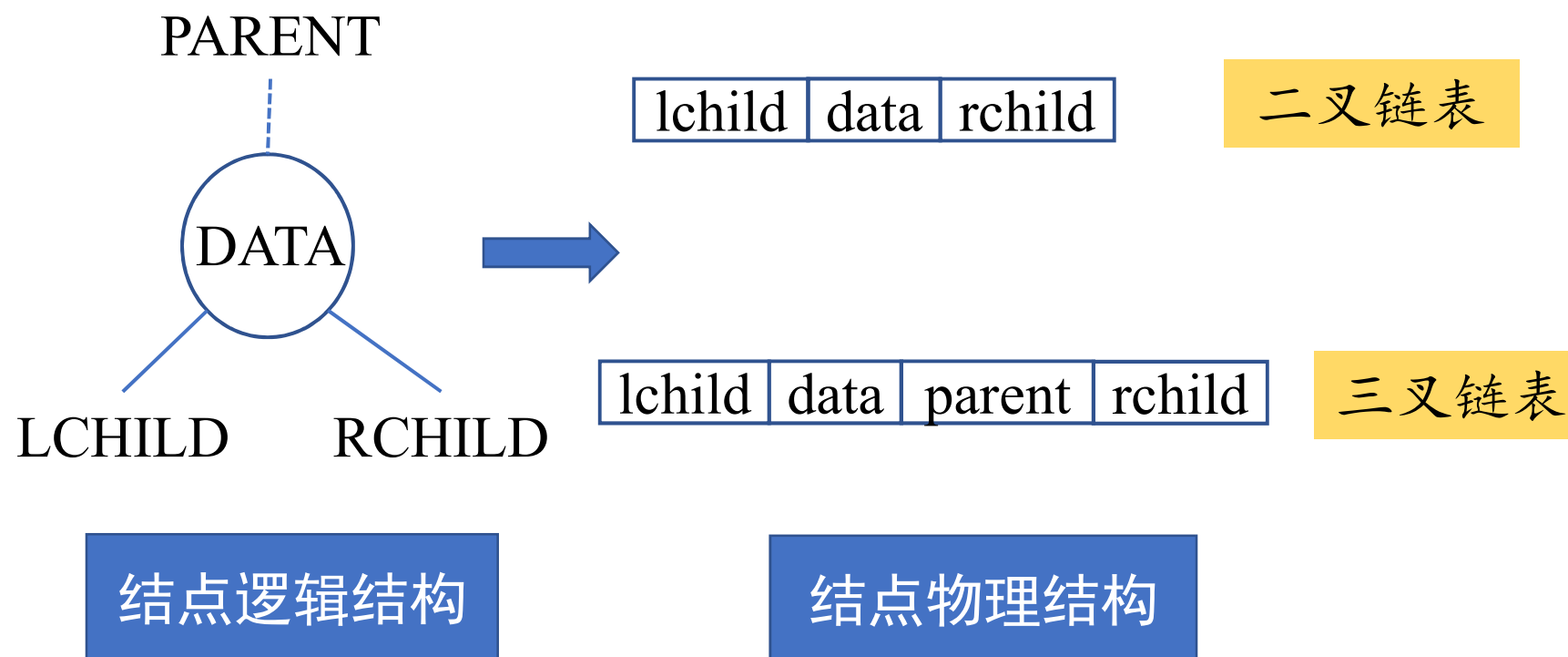
6.2 二叉树

二叉树的存储——链式存储结构

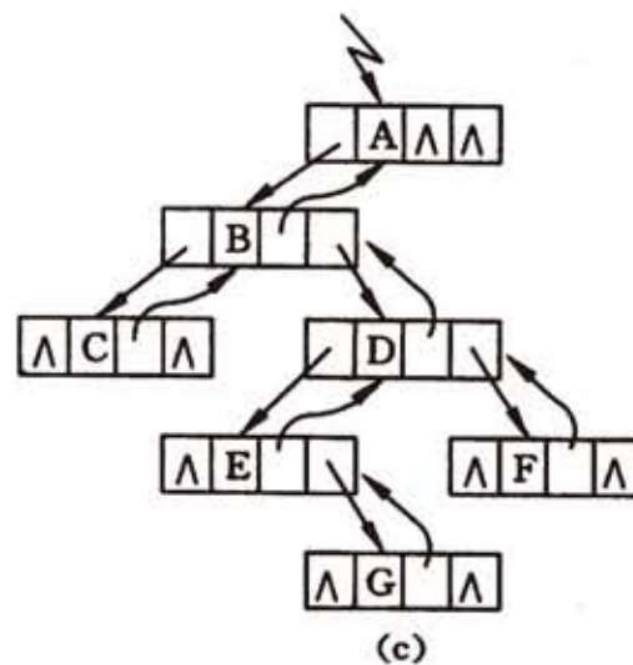
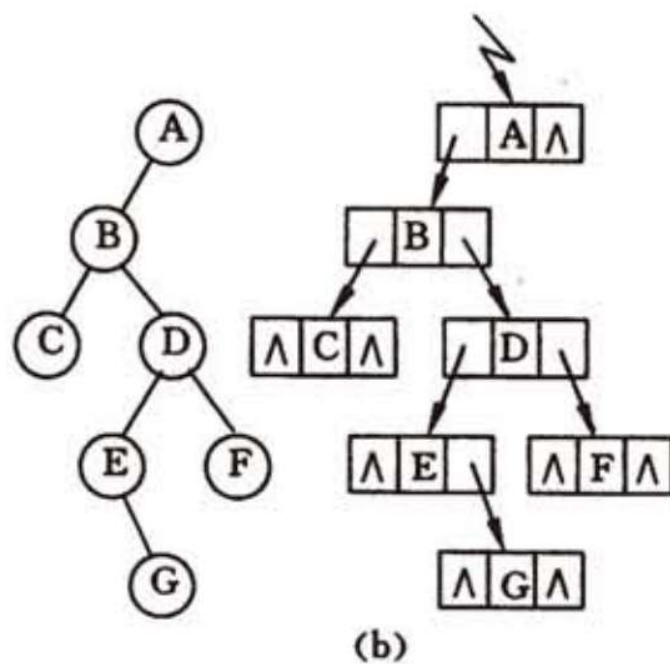
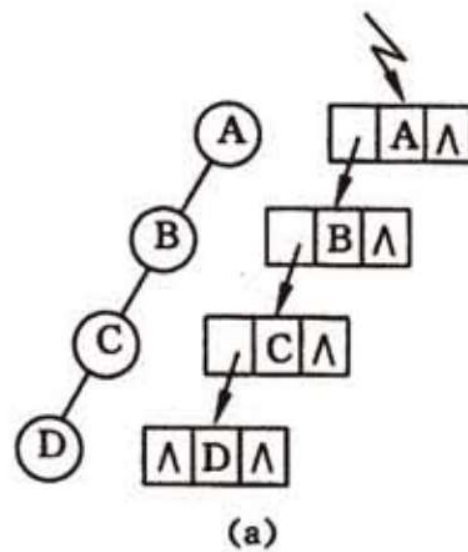
```
typedef struct BiTNode{  
    TElemType      data;  
    struct BiTNode *lchild, *rchild;  
}BiTNode, *BiTree;
```

6.2 二叉树

二叉树的存储——链式存储结构



6.2 二叉树



6.2 二叉树

二叉树的存储——链式存储结构

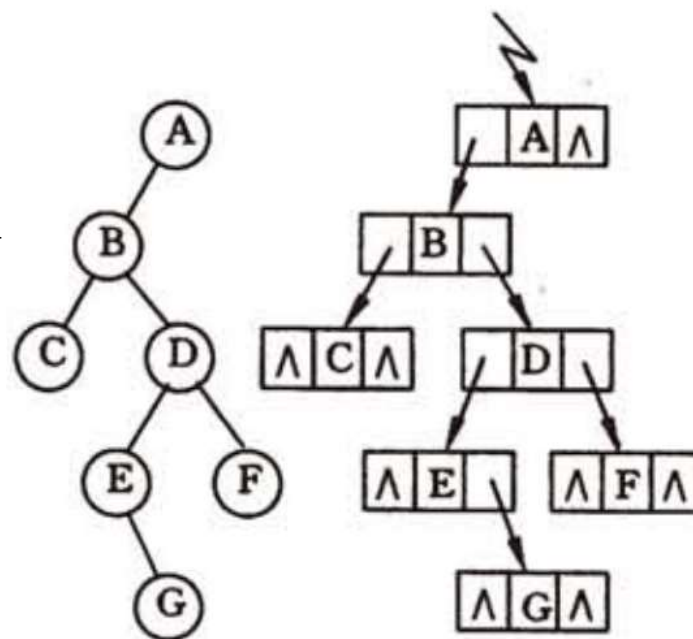
知识点 在含有 n 个结点的二叉链表中有 $n+1$ 个空链域

因为 n 个结点有 $2n$ 个指针；

且 n 个结点中有 $n-1$ 条边；

除了头结点没有边，其余结点都有一个父结点，相当于都有1条边，共 $n-1$ 条；

剩下的空链域就是 $2n-(n-1)=n+1$ ，即 $n+1$ 个空指针。



第 6.3 部分

遍历二叉树和线索二叉树



6.3.1 遍历二叉树

提出问题

在二叉树的一些应用中，常要求在树中查找具有某种特征的结点，或者对树中全部结点逐一进行某种处理。这就提出了遍历二叉树的问题。

6.3.1 遍历二叉树

遍历是任何类型均有的操作

- 对**线性结构**而言，顺序遍历；
- **二叉树**是非线性结构，每个结点有两个后继，则存在如何遍历，即**按什么样的搜索路径遍历**的问题。

6.3.1 遍历二叉树

遍历二叉树

对二叉树而言，是由三个基本单元组成：

- 根结点
- 左子树
- 右子树

因此，若能遍历这三部分，便是遍历了整个二叉树。



考虑：一共有多少种遍历
二叉树的方案？

6.3.1 遍历二叉树

遍历二叉树

假如以 L、D、R 分别表示遍历二叉树的左子树、访问根结点、遍历右子树，

先序 中序 后序

则有 DLR、LDR、LRD

~~DLR、RDL、RLD~~

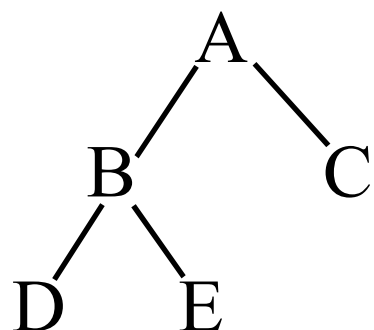
6.3.1 遍历二叉树

先左后右的遍历算法

- ① 先序遍历算法 DLR
- ② 中序遍历算法 LDR
- ③ 后序遍历算法 LRD

6.3.1 遍历二叉树

例1：二叉树遍历示例



先序遍历的结果是： **A B D E C**

中序遍历的结果是： **D B E A C**

后序遍历的结果是： **D E B C A**

口诀：

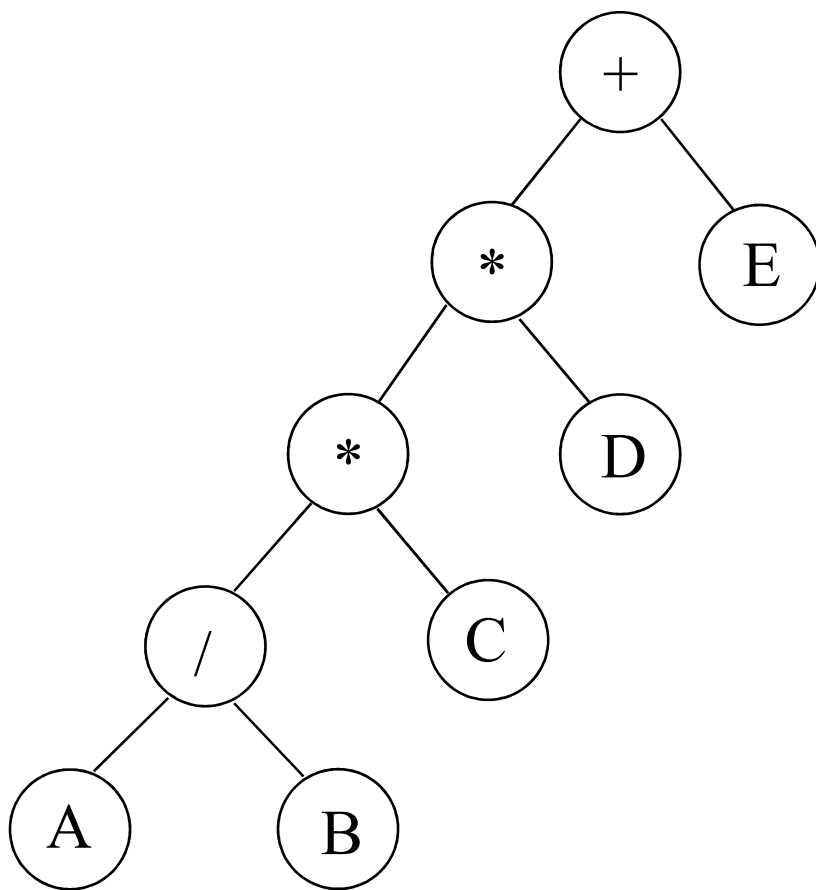
DLR—先序遍历，即**根**左右

L**D**R—中序遍历，即左**根**右

LR**D**—后序遍历，即左右**根**

6.3.1 遍历二叉树

例2：用二叉树表示算术表达式



先序遍历

+ * * / A B C D E

前缀表示

中序遍历

A / B * C * D + E

中缀表示

后序遍历

A B / C * D * E +

后缀表示

层序遍历

+ * E * D / C A B



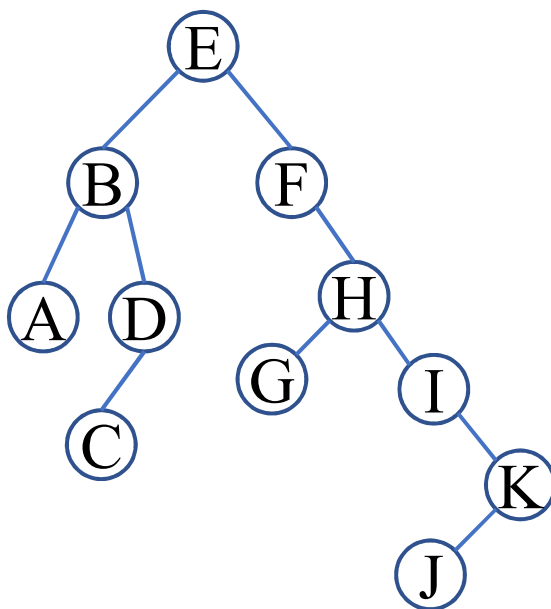
6.3 遍历二叉树和线索二叉树

遍历二叉树

先序序列为 E、B、A、D、C、F、H、G、I、K、J

中序序列为 A、B、C、D、E、F、G、H、I、J、K。

画出该二叉树。



6.3.1 遍历二叉树

先序遍历算法

若二叉树为空树，则空操作；否则

- (1) 访问根结点；
- (2) 先序遍历左子树；
- (3) 先序遍历右子树。

6.3.1 遍历二叉树

先序遍历算法

```
void PreOrderTraverse(BiTree T){  
1   if (T){  
2       printf(T->data);  
3       PreOrderTraverse(T->lchild);  
4       PreOrderTraverse(T->rchild);  
5   }  
} //PreOrderTraverse
```

6.3.1 遍历二叉树

中序遍历算法

```
void InOrderTraverse(BiTree T){  
1   if (T){  
2       InOrderTraverse(T->lchild);  
3       printf(T->data);  
4       InOrderTraverse(T->rchild);  
5   }  
} //InOrderTraverse
```

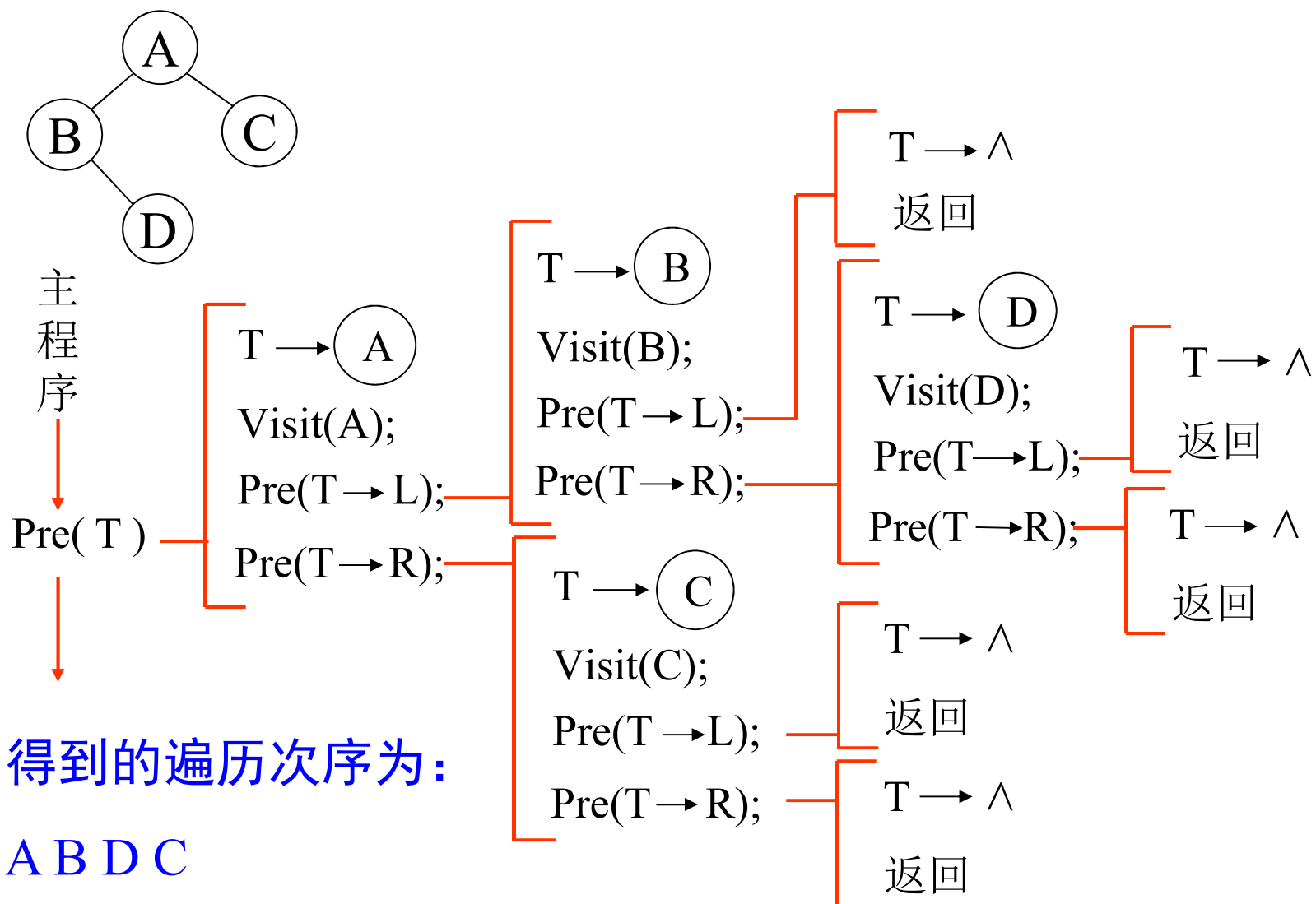
6.3.1 遍历二叉树

后序遍历算法

```
void PostOrderTraverse(BiTree T){  
1   if (T){  
2       PostOrderTraverse(T->lchild);  
3       PostOrderTraverse(T->rchild);  
4       printf(T->data);  
5   }  
} //InOrderTraverse
```

6.3.1 遍历二叉树

先序遍历算法



6.3.1 遍历二叉树

先序遍历算法

```
Status PreOrderTraverse(BiTree T, Status(*Visit)(TElemType e)){  
1   if (T){  
2       if (Visit(T->data))  
3           if (PreOrderTraverse(T->lchild, Visit))  
4               if(PreOrderTraverse(T->rchild, Visit))  
5                   return OK;  
6       return ERROR;  
7   }else return OK;  
} //PreOrderTraverse
```

```
Status PrintElement(TElemType e){  
    printf(e);  
    return OK;  
}
```

//调用实例: **PreOrderTraverse(T, PrintElement);**



6.3.1 遍历二叉树

中序遍历算法

```
Status InOrderTraverse(BiTree T, Status(*Visit)(TElemType e){
1   if (T){
2       if (InOrderTraverse(T->lchild, Visit))
3           if (Visit(T->data))
4               if (InOrderTraverse(T->rchild, Visit))
5                   return OK;
6       return ERROR;
7   }else return OK;
} // InOrderTraverse
```

6.3.1 遍历二叉树

后序遍历算法

```
Status PostOrderTraverse(BiTree T, Status(*Visit)(TElemType e){  
1   if (T){  
2       if (PostOrderTraverse(T->lchild, Visit))  
3           if (PostOrderTraverse(T->rchild, Visit))  
4               if (Visit(T->data))  
5                   return OK;  
6       return ERROR;  
7   }else return OK;  
} // PostOrderTraverse
```



6.3 遍历二叉树和线索二叉树

遍历的分析

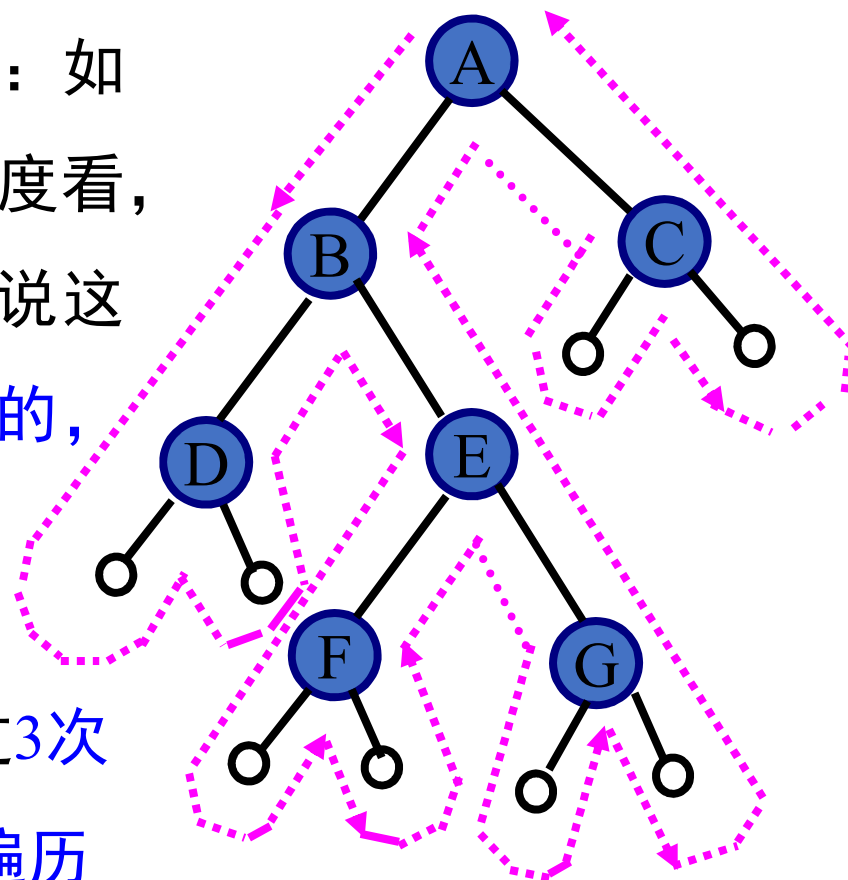
1. 从前面的三种遍历算法可以知道：如果将Visit()语句抹去，从递归的角度看，这三种算法是完全相同的，或者说这三种遍历算法的访问路径是相同的，只是访问结点的时机不同。

从虚线的起点到终点，每个结点经过3次

在第1次经过时访问Visit = 先序遍历

在第2次经过时访问Visit = 中序遍历

在第3次经过时访问Visit = 后序遍历



6.3 遍历二叉树和线索二叉树

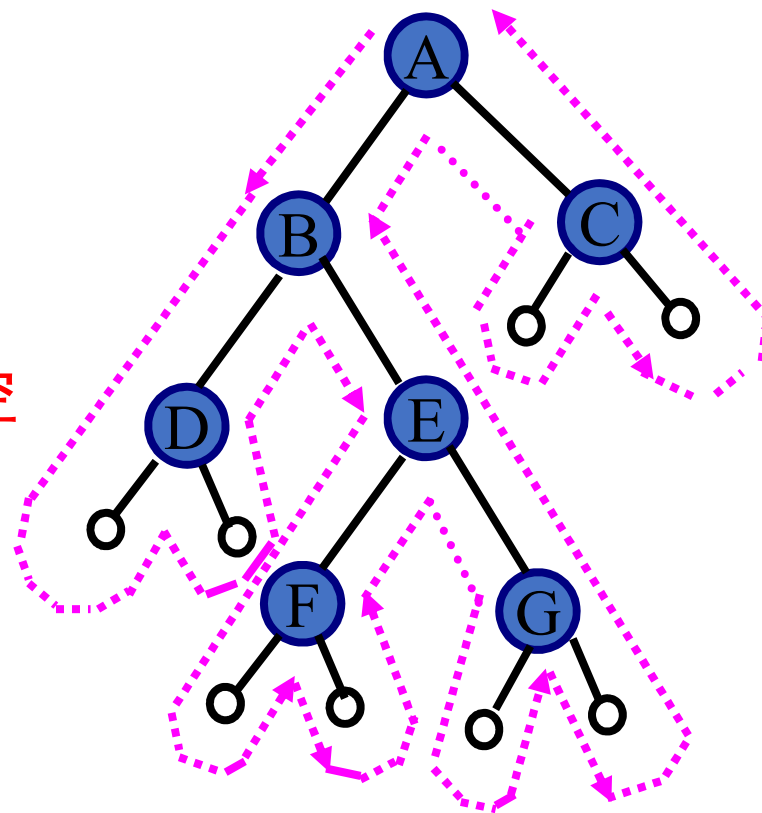
遍历的分析

2. 二叉树遍历的时间效率和空间效率

基本操作：访问结点Visit()

时间效率： $O(n)$ // 每个结点只访问一次

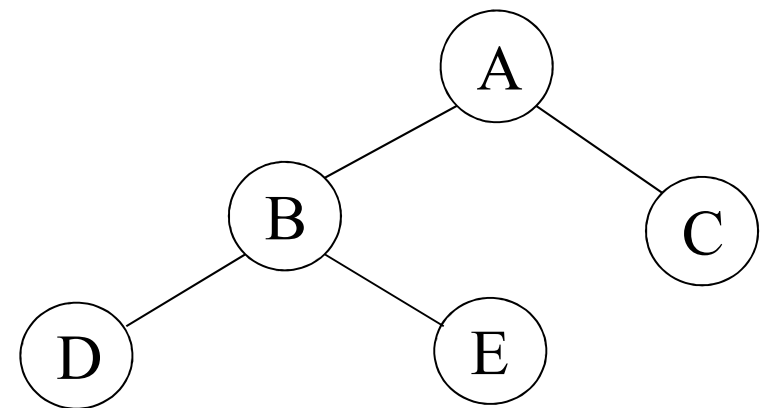
空间效率： $O(n)$ // 栈占用的最大辅助空间



6.3 遍历二叉树和线索二叉树

中序遍历二叉树的非递归算法一

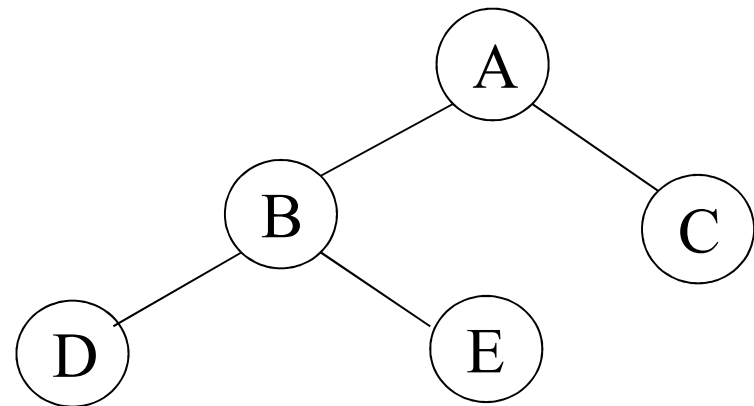
```
Status InOrderTraverse(BiTree T, Status(* Visit)(TElemType e)){  
1  InitStack(S); Push(S,T);  
2  while (!StackEmpty(S)){  
3      while ((GetTop(S,p) && p) Push(S, p->lchild);  
4      Pop(S,p);  
5      if (!StackEmpty(S)){  
6          Pop(S, p); if (!Visit(p->data)) return ERROR;  
7          Push(S, p->rchild);  
8      } //if  
9  } //while  
10 return OK;  
} //InOrderTraverse
```



6.3 遍历二叉树和线索二叉树

中序遍历二叉树的非递归算法二

```
Status InOrderTraverse (BiTree T, Status (*Visit)(TElemType e)){  
1  InitStack(S); p=T;  
2  while (p || !StackEmpty(S)){  
3      if (p){ Push(S, p); p = p->lchild; }  
4      else {  
5          Pop(S, p); if (!Visit(p->data)) return ERROR;  
6          p = p->rchild;  
7      } //else  
8  } //while  
9  return OK;  
} //InOrderTraverse
```



6.3 遍历二叉树和线索二叉树

遍历算法的应用举例

“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，如：对于一棵已知树求结点的双亲，求结点的孩子结点，判定结点所在的层次等。

6.3 遍历二叉树和线索二叉树

例1：统计二叉树中叶子结点的个数

```
Status CountLeaf (BiTree T, int &count) {  
    if ( T ) {  
        if ( (T->lchild == NULL) && (T->rchild == NULL)) {  
            count++; return OK;  
        }  
        // 统计左子树中叶子结点个数  
        CountLeaf(T->lchild, count);  
        // 统计右子树中叶子结点个数  
        CountLeaf( T->rchild, count);  
    } else return OK;  
}
```


6.3 遍历二叉树和线索二叉树

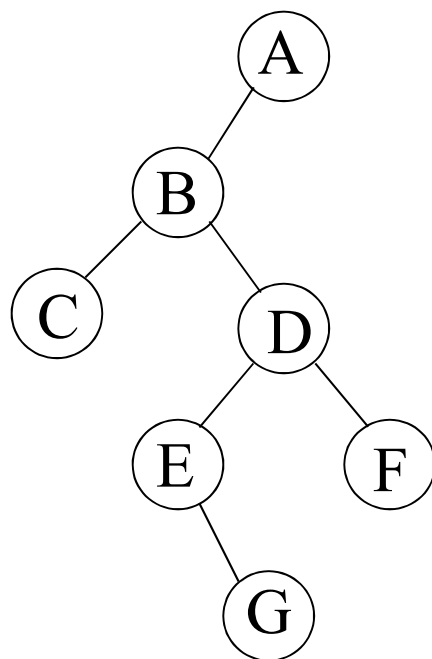
例1：求二叉树的深度

```
int Depth (BiTree T) {  
    if ( !T ) depthval = 0;  
    else {  
        depthLeft = Depth( T->lchild );  
        depthRight = Depth( T->rchild );  
        depthval = 1 + max(depthLeft, depthRight);  
    }  
    return depthval;  
}
```

6.3 遍历二叉树和线索二叉树

例3：按先序序列建立二叉树的二叉链表

已知先序序列：A B C 0 0 D E 0 G 0 0 F 0 0 0 (其中：0表示空格字符，空指针)建立相应的二叉链表



6.3 遍历二叉树和线索二叉树

例3：按先序序列建立二叉树的二叉链表

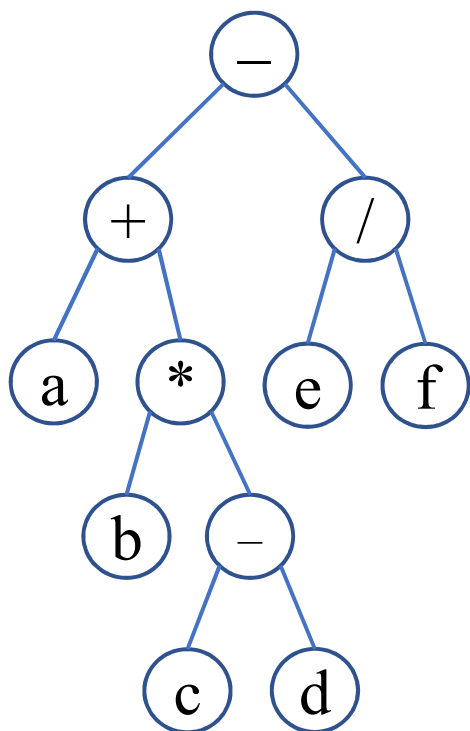
```
Status CreateBiTree (BiTree &T) {  
1    scanf(&ch);  
2    if (ch == ' ') T=NULL;  
3    else {  
4        if ( !(T=(BiTNode*)malloc(sizeof(BiTNode)) ) )  
5            exit(OVERFLOW);  
6        T->data = ch;  
7        CreateBiTree(T->lchild);  
8        CreateBiTree(T->rchild);  
9    }  
10   return OK;  
} // CreateBiTree
```



6.3 遍历二叉树和线索二叉树

线索二叉树

遍历二叉树是以一定的规则将二叉树中结点排成一个线性序列，得到二叉树中结点的先序/中序/后序序列。

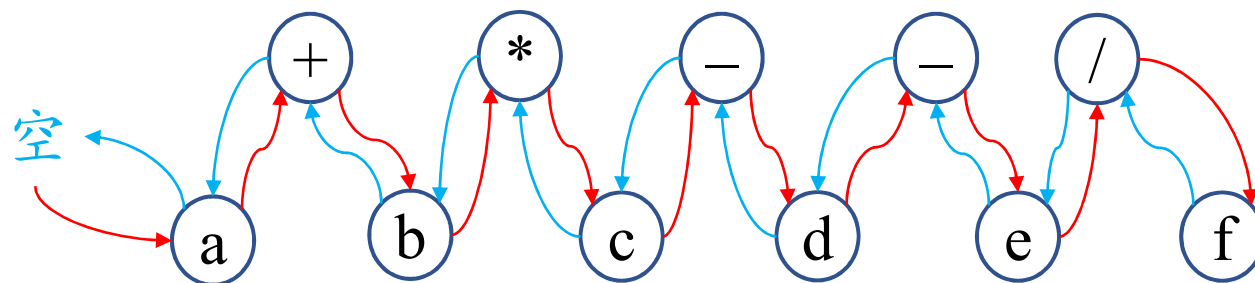


中序遍历序列

$a + b * c - d - e / f$

遍历过程实质上是对(非)线性结构进行线性化操作——按某种遍历规则。

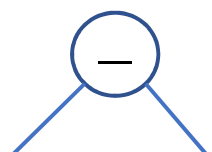
存在前驱和后继关系



6.3 遍历二叉树和线索二叉树

线索二叉树

遍历二叉树是以一定的规则将二叉树中结点排成一个线性序列，得到二叉树中结点的先序/中序/后序序列。

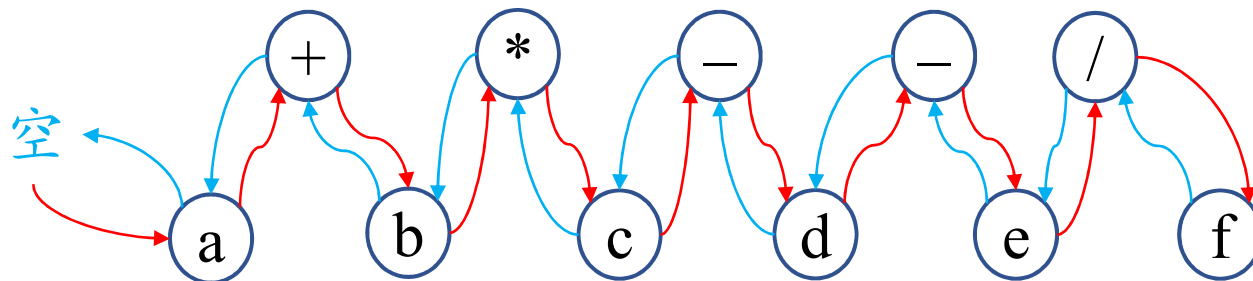
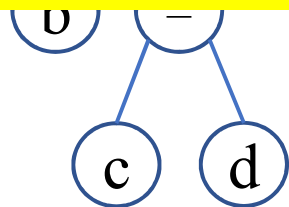


中序遍历序列

$a + b * c - d - e / f$

普通二叉树只能找到结点的左右孩子信息，而该结点的直接前驱和直接后继只能在遍历过程中获得。

若将遍历后对应的有关前驱和后继预存起来，则从第一个结点开始就能很快“顺藤摸瓜”而遍历整个树了。



6.3 遍历二叉树和线索二叉树

以中序遍历序列为例

线索二叉树

首先，如何存储前驱和后继信息。采用链式存储，分析如下：



采用左图所示存储结构：

- 增加存储指针空间
- 数据密度降低



如何简化存储，增加数据密度？



在有 n 个结点的二叉链表中，
有 $n+1$ 个空链域！



假设可以，以中序遍历序列
为例，具体分析。

问题1：如何表示前驱和后继关系？

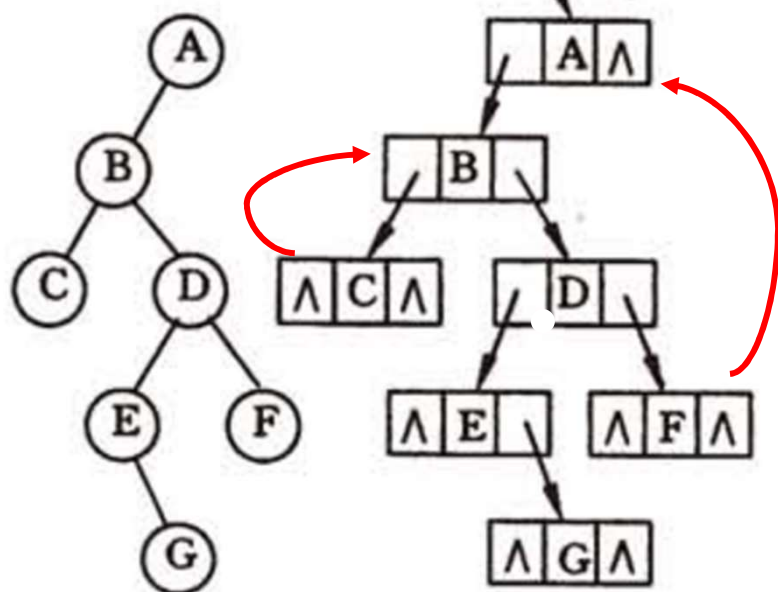
6.3 遍历二叉树和线索二叉树

以中序遍历序列为例

线索二叉树

问题2：并不是所有结点都是两个指针域为空，怎么处理？

增加两个标志位，LTag 和 RTag，用来指示存储的是左右孩子还是前驱/后继。



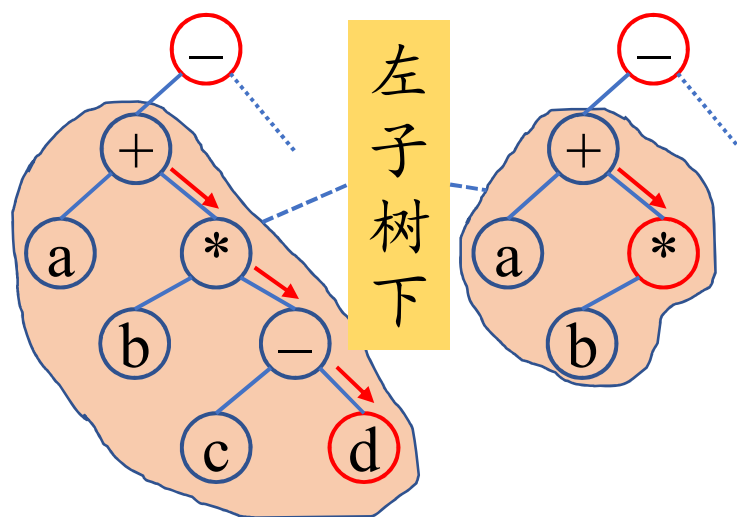
不能区分是
孩子还是前
驱/后继！

6.3 遍历二叉树和线索二叉树

以中序遍历序列为例

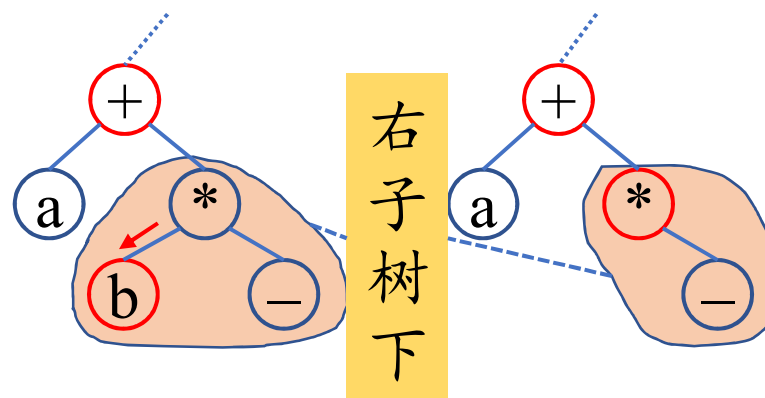
线索二叉树

问题3：具体分析能否采用这种方式找到直接前驱和后继结点



LTa_g = 0时查找前驱

一直向右下找，找到第一个无右子树结点(RTa_g = 1)



RTa_g = 0时查找后继

一直向左下找，找到第一个无左子树结点(LTa_g = 1)

6.3 遍历二叉树和线索二叉树

线索二叉树

重要概念，P132页，倒数第3段：

- 以这种结点结构构成的二叉链表作为二叉树的存储结构，叫做**线索链表**；
- 指向结点前驱和后继的指针，叫做**线索**；
- 加上线索的二叉树称为**线索二叉树**；
- 对二叉树以某种次序进行遍历，并将其变为线索二叉树的过程为**线索化**。



6.3 遍历二叉树和线索二叉树

二叉树的二叉线索表示



```
typedef enum PointerTag {Link, Thread};
```

```
typedef struct BiThrNode{
```

```
    TElemType      data;
```

```
    struct BiThrNode *lchild, *rchild;
```

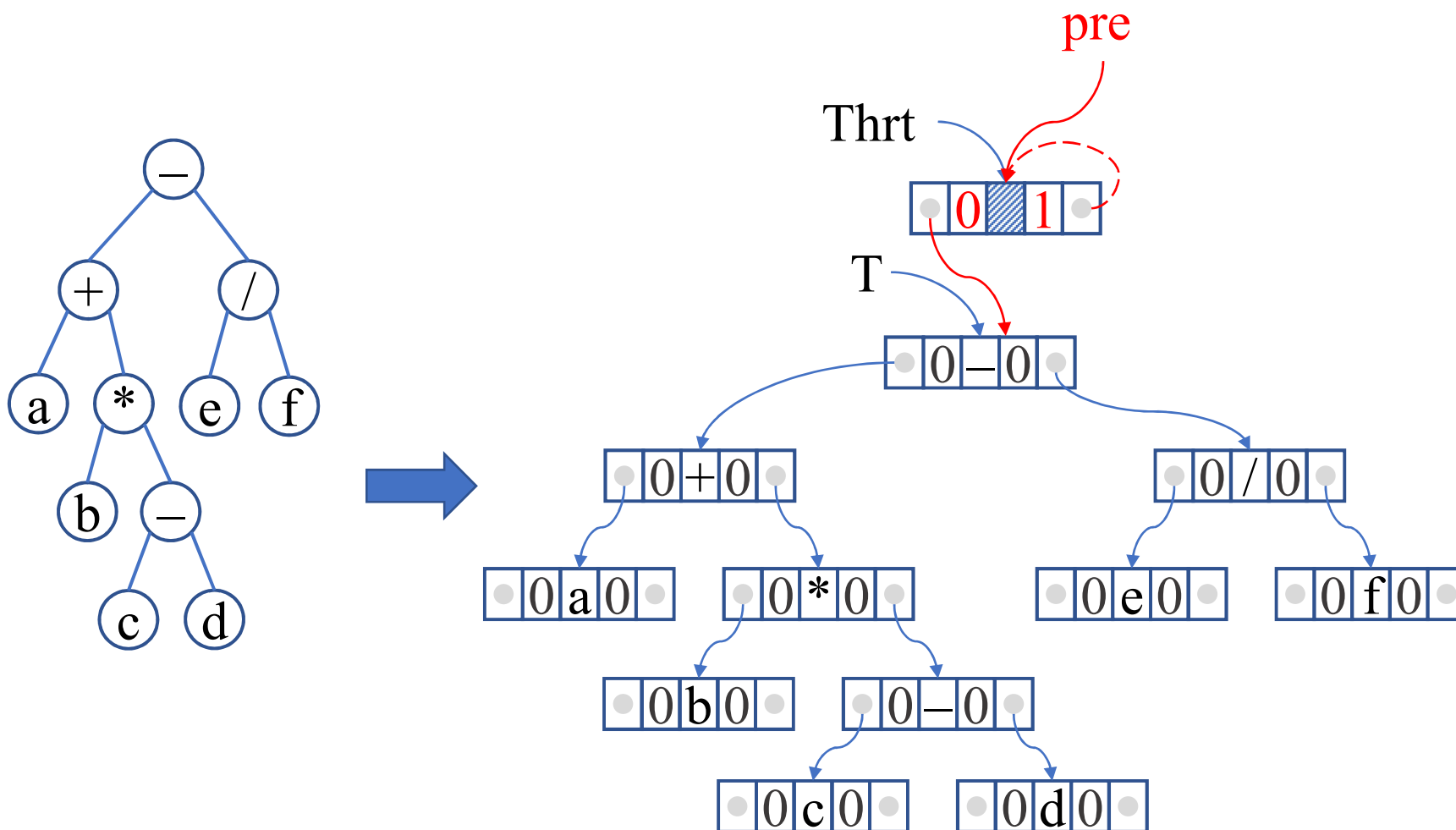
```
    PointerTag      LTag, RTag;
```

```
}BiThrNode, *BiThrTree;
```

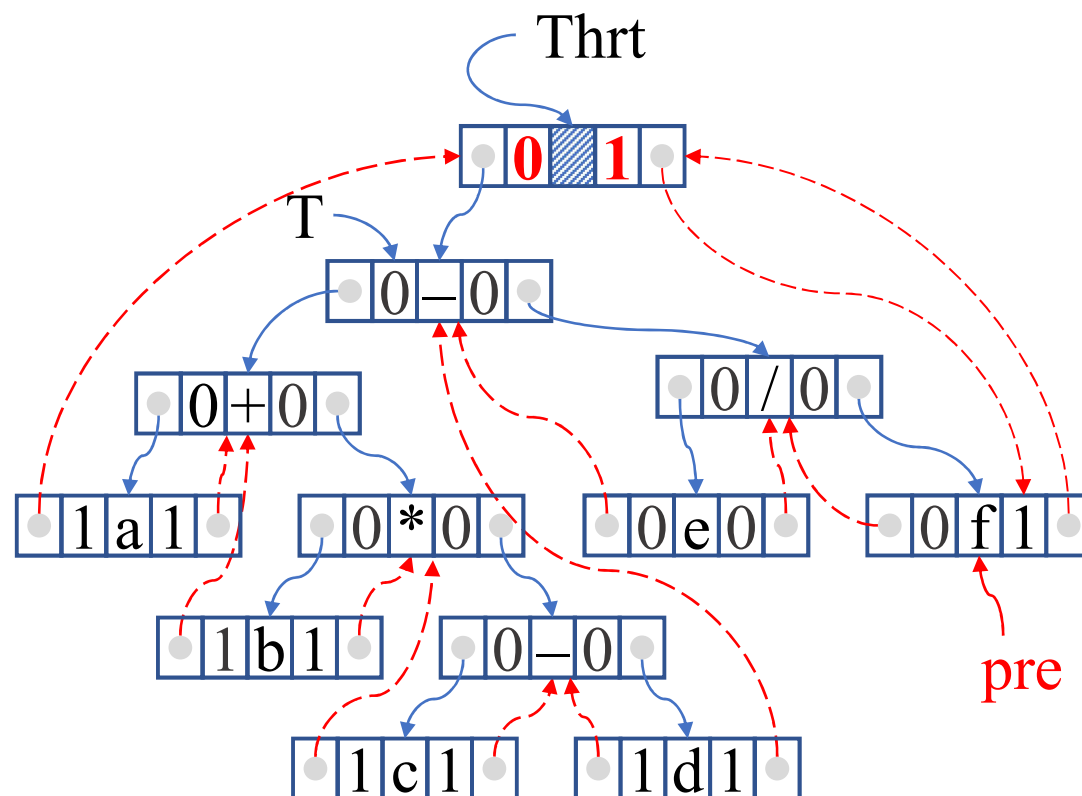
6.3 遍历二叉树和线索二叉树

线索化过程——中序线索化算法示例

二叉树中序线索化算法

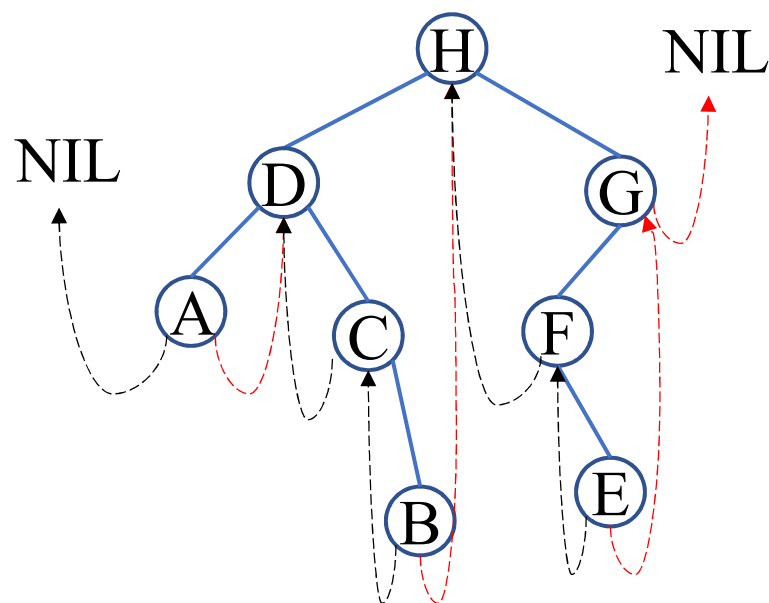


6.3 遍历二叉树和线索二叉树



6.3 遍历二叉树和线索二叉树

画出二叉树的中序线索二叉树

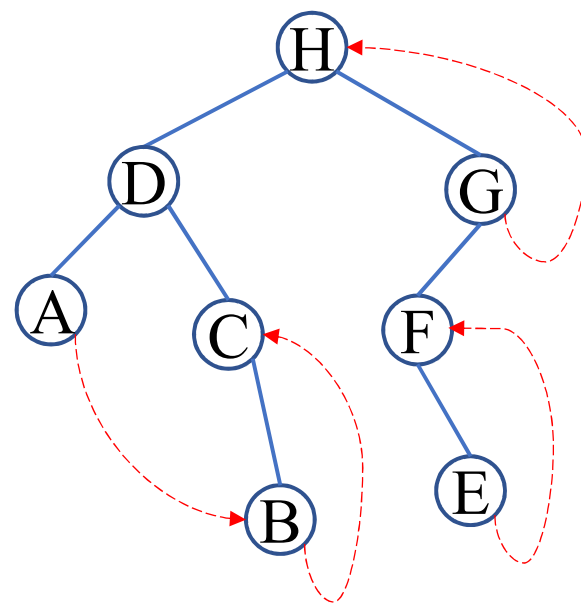


6.3 遍历二叉树和线索二叉树

画出二叉树的**后序后继线索二叉树**

在后序线索二叉树中找结点后继：

- (1) 若结点 x 是二叉树的根，则其后继为空；
- (2) 若结点 x 是其双亲的右孩子或是其双亲的左孩子且其双亲没有右子树，则其后继即为双亲结点；
- (3) 若结点 x 是其双亲的左孩子，且其双亲有右子树，则其后继为双亲的右子树上按后序遍历列出的第一个结点。



第 6.4 部分

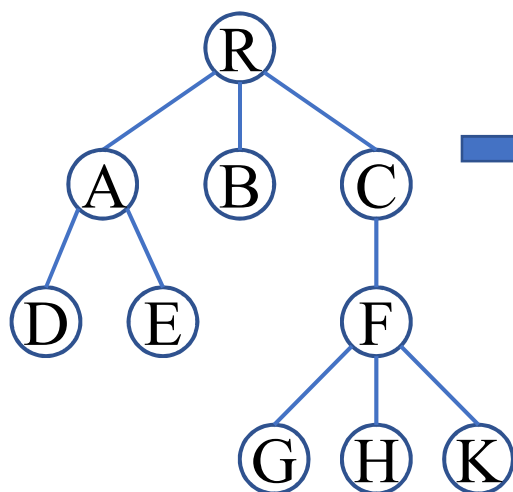
树和森林



6.4 树和森林

树的存储结构——双亲表示法

利用每个结点只有一个
双亲的性质。



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

```
#define MAX_TREE_SIZE 100
```

```
typedef struct PTNode{
```

```
TElemType data;
```

```
int parent;
```

```
}PTNode;
```

```
typedef struct{
```

```
PTNode nodes[MAX_TREE_SIZE];
```

```
int r, n;
```

```
}PTree;
```


6.4 树和森林

采用多重链表，即每个结点设置多个指针域，每个指针域指向一棵子树的根结点。

data	child1	child2	...	child \mathbf{d}
------	--------	--------	-----	--------------------

同构： \mathbf{d} 为树的度

一棵有 n 个结点且度为 k 的树中，必由 $n(k-1) + 1$ 个空链域。

data	degree	child1	child2	...	child $\overline{\mathbf{d}}$
------	--------	--------	--------	-----	-------------------------------

不同构： $\overline{\mathbf{d}}$ 为结点的度

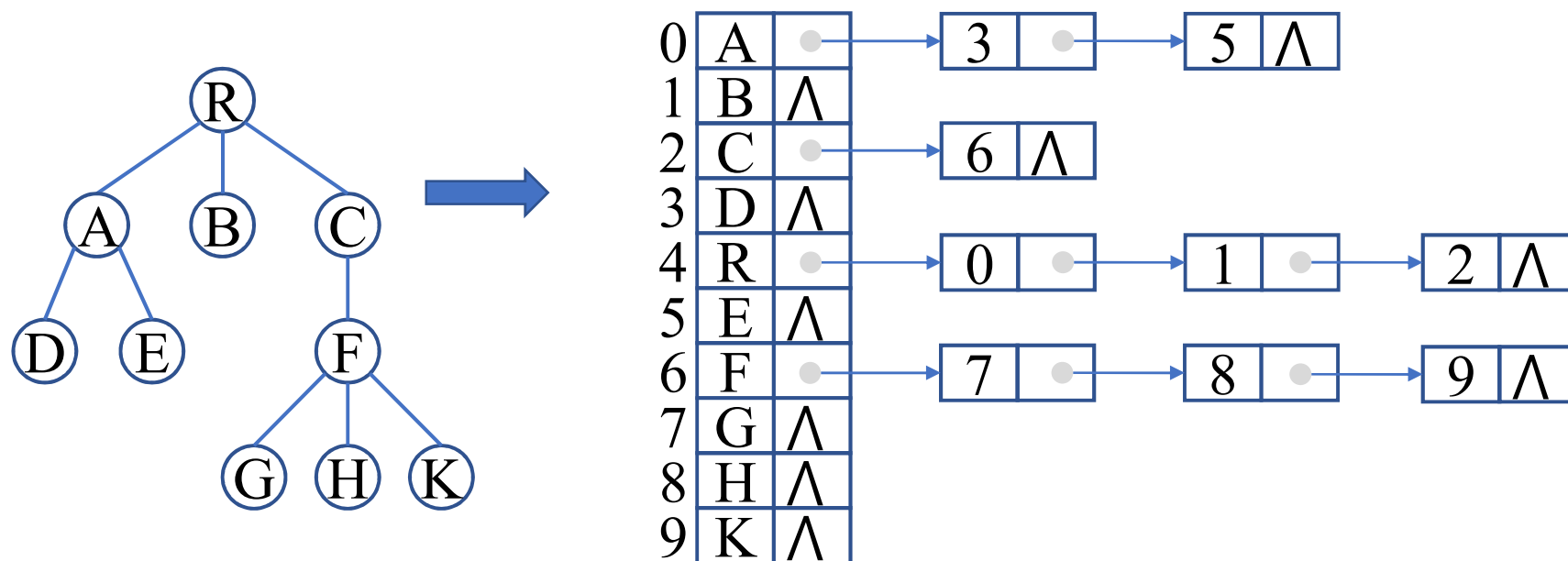
节约存储空间，但操作不方便。



6.4 树和森林

树的存储结构——孩子表示法

把每个结点的孩子排列起来，看成是一个线性表，且用单链表作存储结构，则n个结点有n个孩子链表。而n个头指针又组成一个线性表，可采用顺序结构存储。



6.4 树和森林

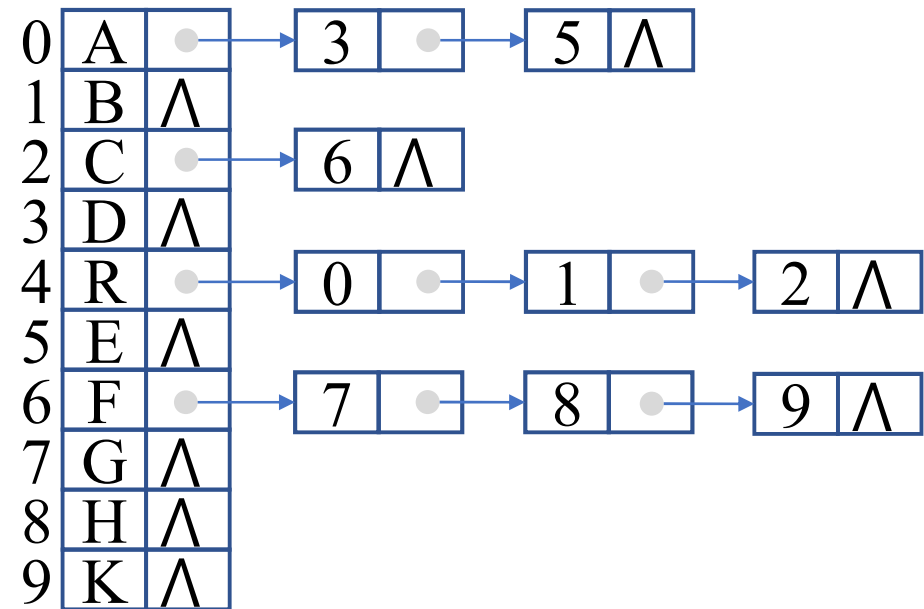
```
#define MAX_TREE_SIZE 100
```

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

```
typedef struct {  
    TElemType  data;  
    ChildPtr   firstChild;  
}CTBox;
```

```
typedef struct {  
    CTBox nodes[MAX_TREE_SIZE];  
    int      n, r;  
}CTree;
```

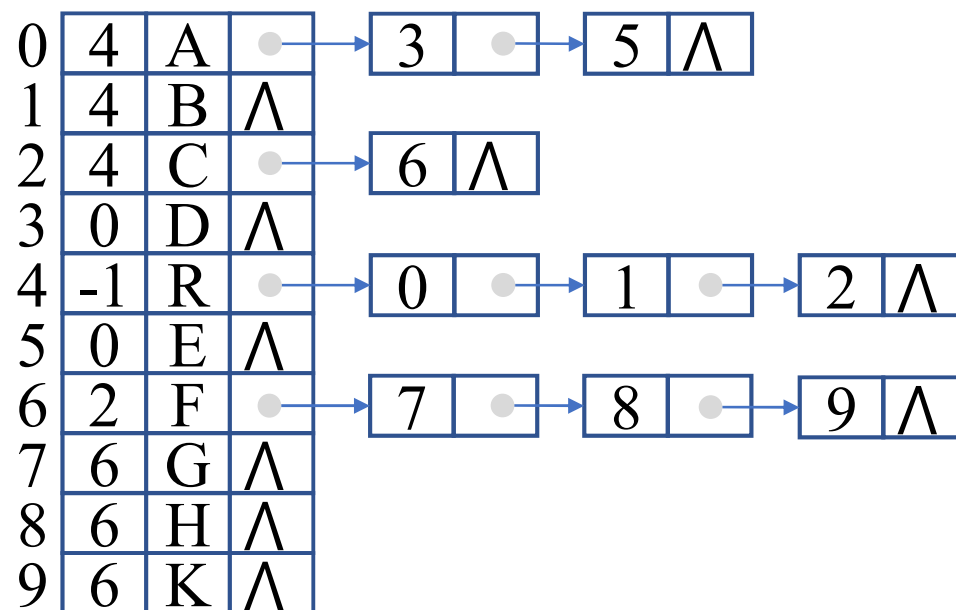
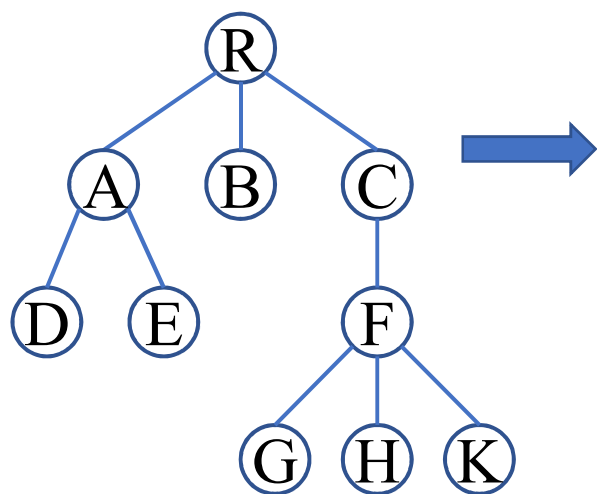
树的存储结构——孩子表示法



6.4 树和森林

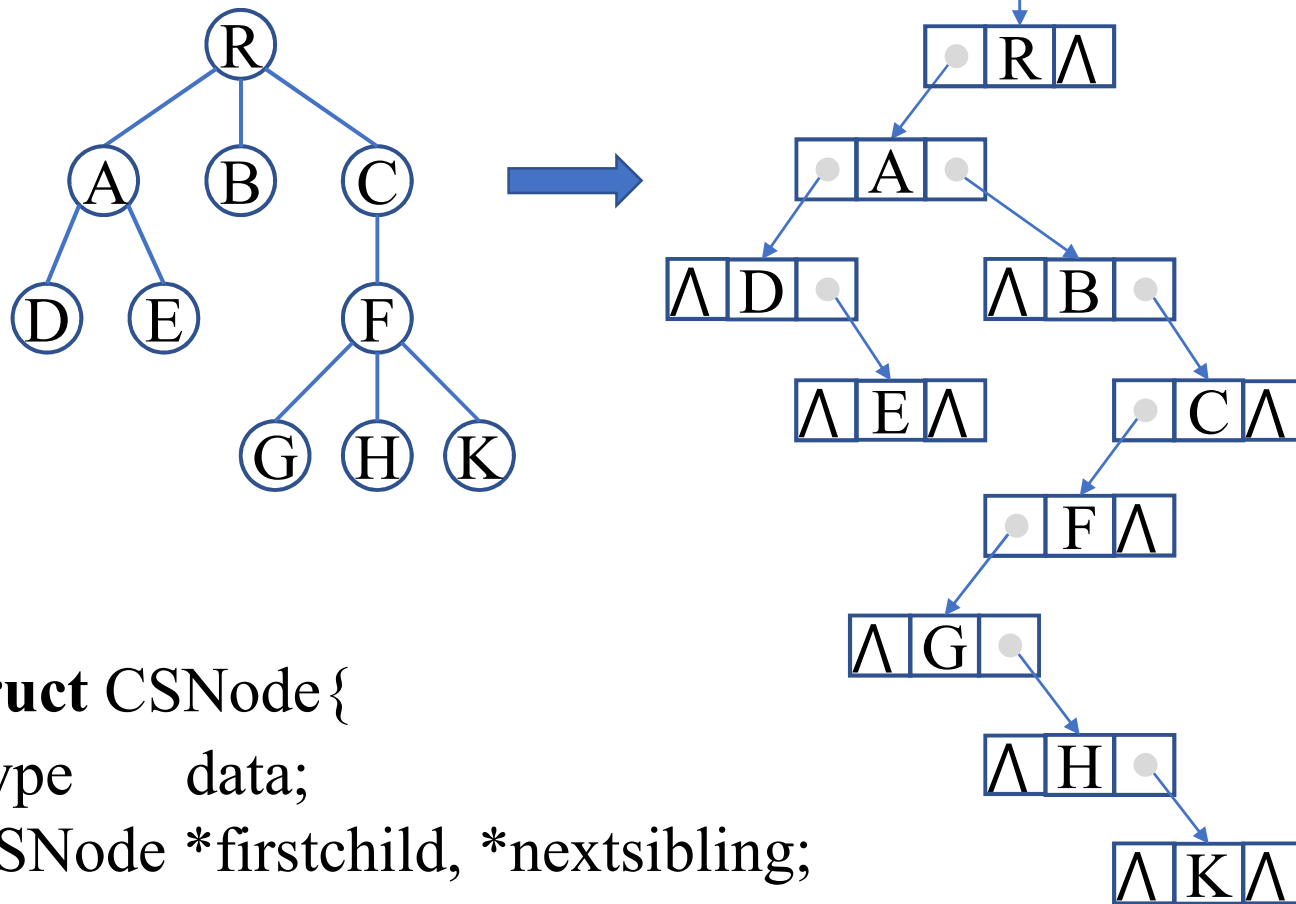
树的存储结构——孩子表示法

把双亲表示法和孩子表示法结合起来！



6.4 树和森林

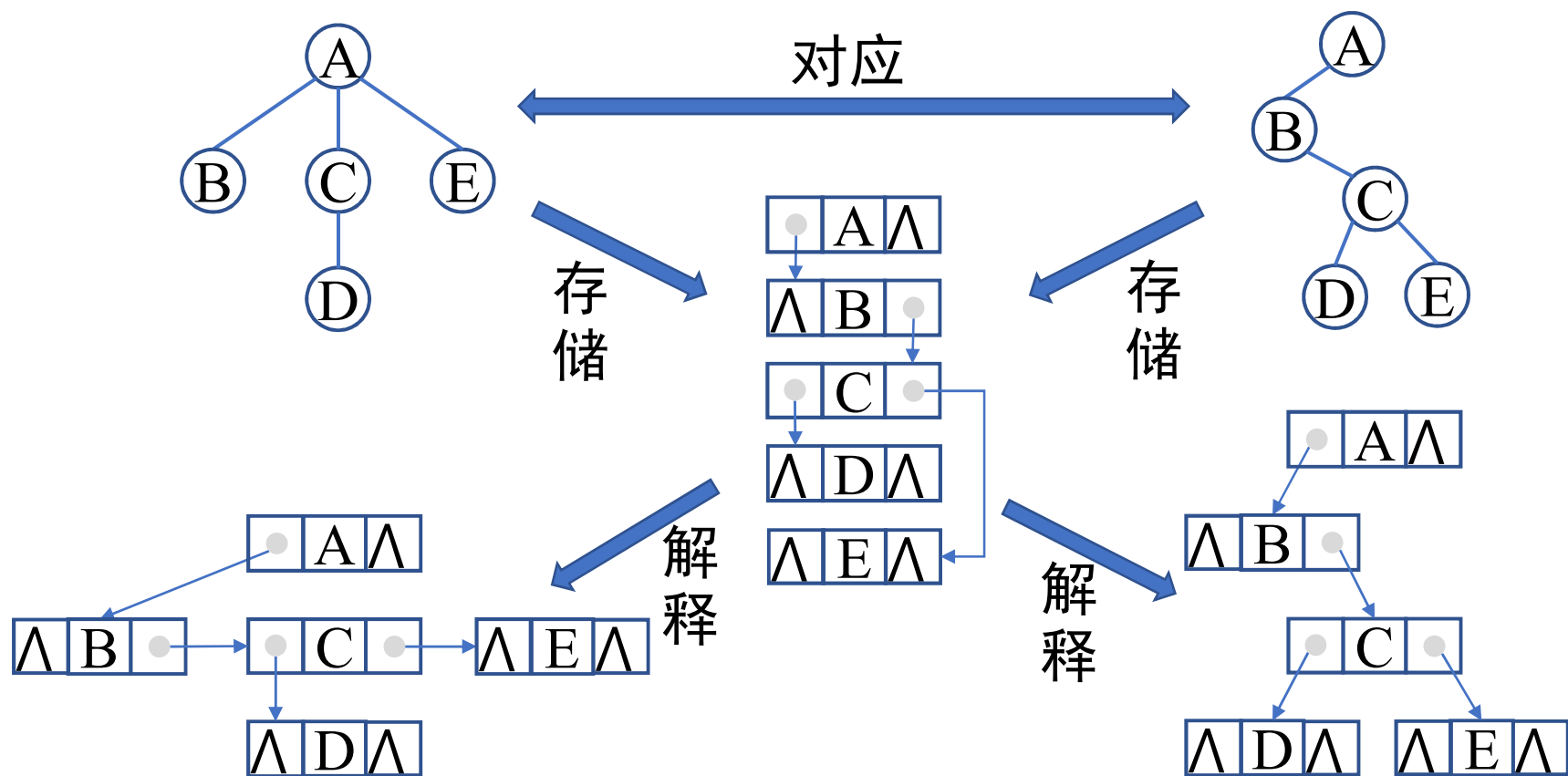
树的存储结构——孩子兄弟表示法 / 二叉树表示法 / 二叉链表表表示法



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

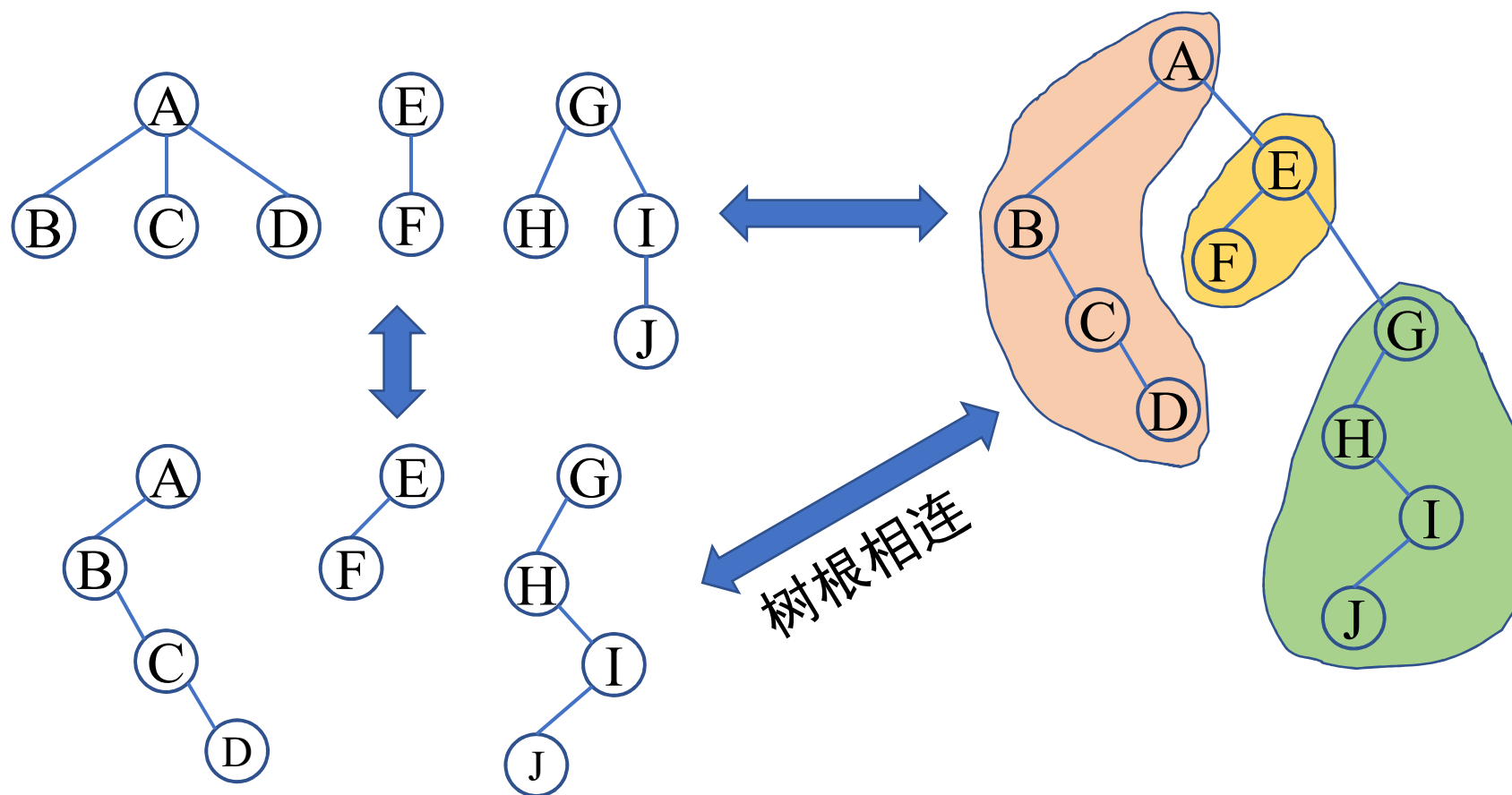
6.4 树和森林

树与二叉树的转换



6.4 树和森林

森林与二叉树的转换

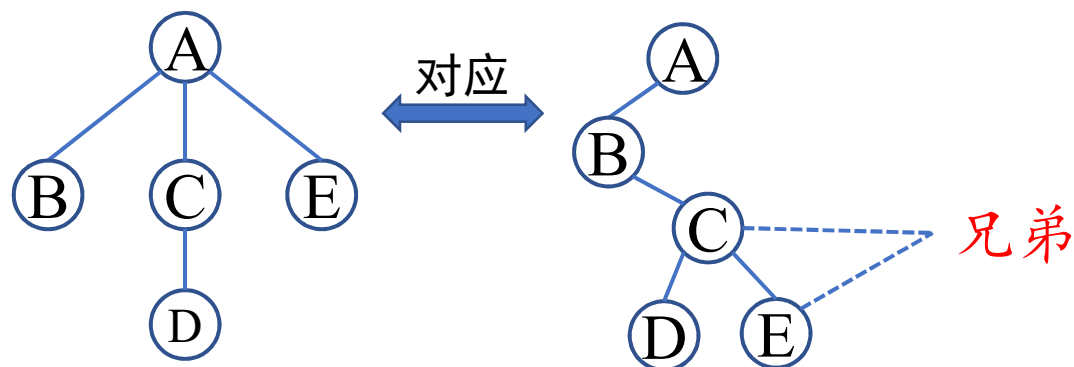


6.4 树和森林

树的遍历

■ **先根遍历树**：先访问树的根结点，然后依次先根遍历根的每棵子树；

■ **后根遍历树**：先依次后根遍历每棵子树，然后访问根结点；



先根序列： A B C D E

后根序列： B D C E A

6.4 树和森林

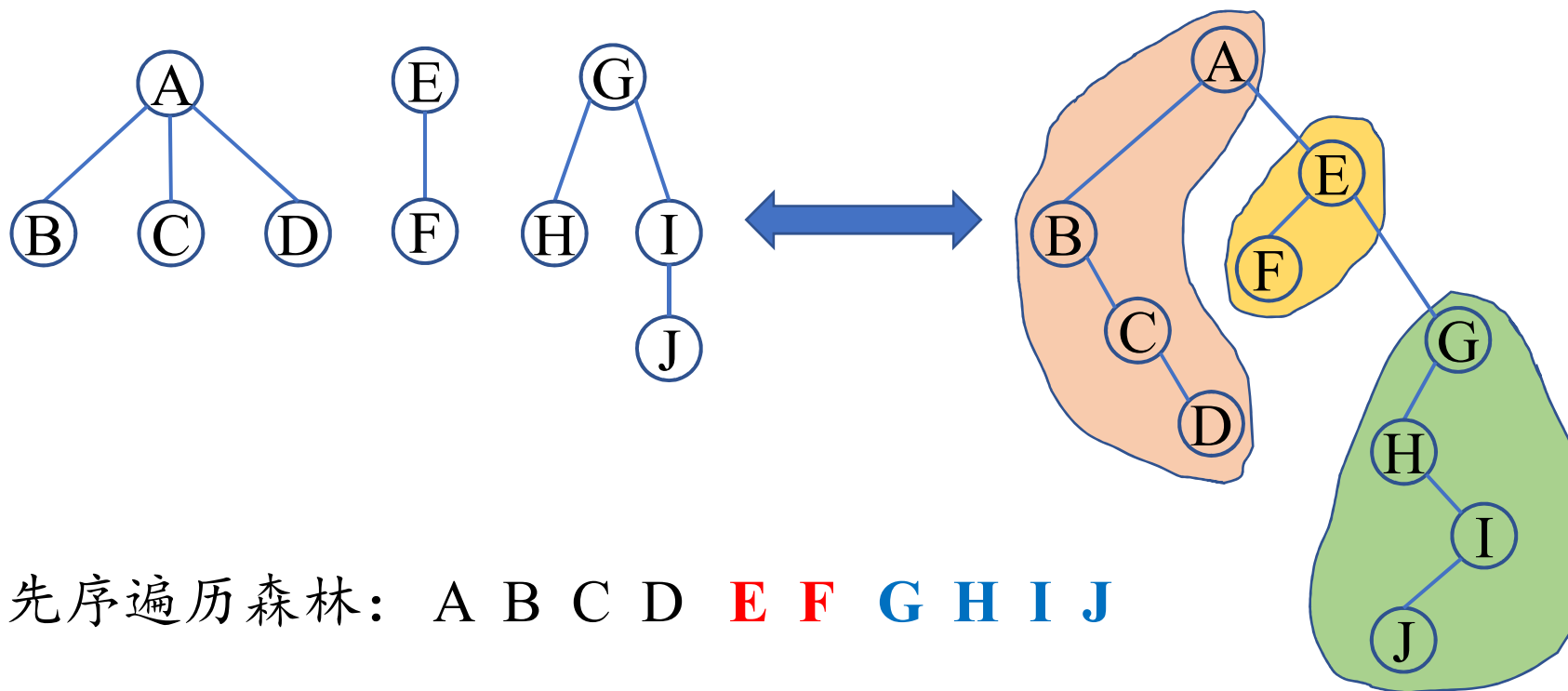
森林的遍历——先序遍历森林

若森林非空，则可按照下述规则遍历：

- (1)访问森林中**第一棵树的根结点**；
- (2)先序遍历第一棵树中根结点的**子树森林**；
- (3)先序遍历除去第一棵树之后**剩余的树构成的森林**

6.4 树和森林

森林的遍历——先序遍历森林



6.4 树和森林

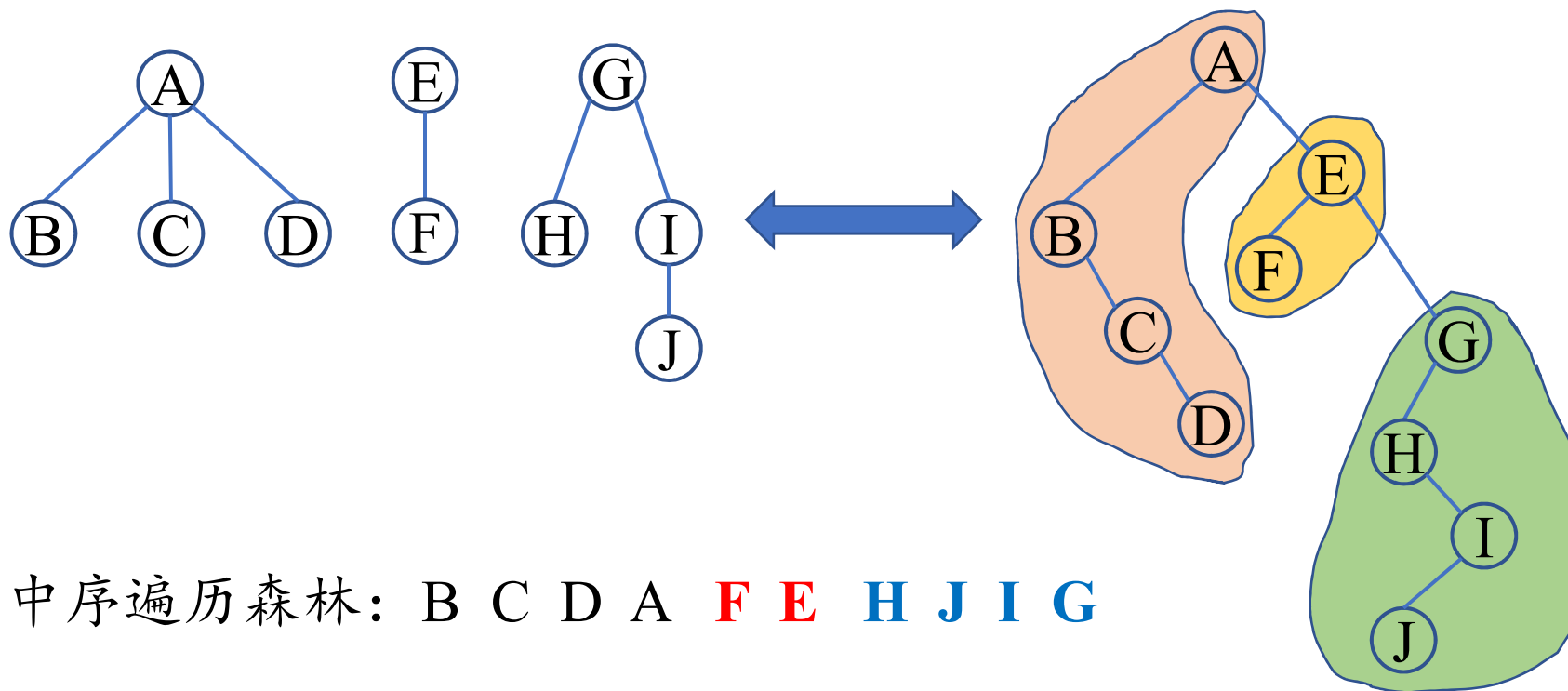
森林的遍历——中序遍历森林

若森林非空，则可按照下述规则遍历：

- (1) 中序遍历第一棵树中根节点的子树森林；
- (2) 访问森林中第一棵树的根结点；
- (3) 中序遍历除去第一棵树之后剩余的树构成的森林

6.4 树和森林

森林的遍历——中序遍历森林



第 6.6 部分

赫夫曼树及其应用



6.6 赫夫曼树及其应用

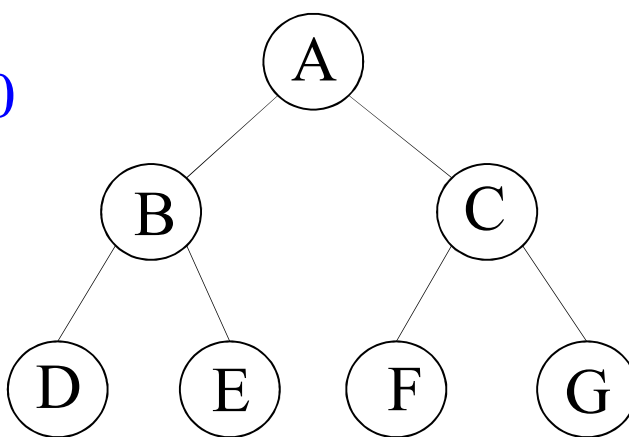
基本术语

路径：由一结点到另一结点间的**分支**所构成

路径长度：路径上的**分支数目** $A \rightarrow E$ 的路径长度 = 2

树的路径长度：从树根到**每一结点**的路径长度之和

树路径长度 = 10



6.6 赫夫曼树及其应用

基本术语

结点带权路径长度：结点 to 根的路径长度与结点上权的乘积

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

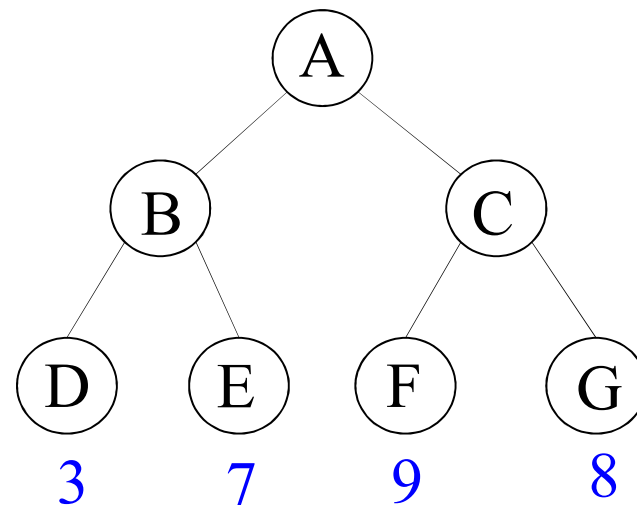
结点F的路径长度为2，其WPL=2×9=18

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

n ：叶子结点的数目

w_i ：叶子结点 k_i 的权值

l_i ：根到 k_i 之间的路径长度

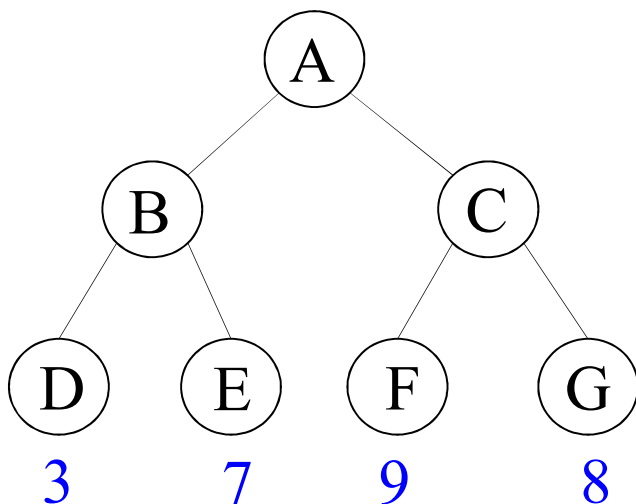


6.6 赫夫曼树及其应用

基本术语

赫夫曼树：带权路径长度最小的树，又称**最优二叉树**

树的带权路径长度如何计算？ $WPL = \sum_{i=1}^n w_i l_i$

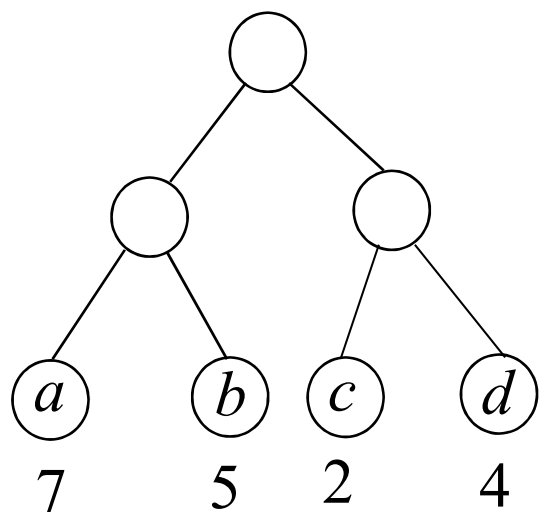


Weighted Path Length

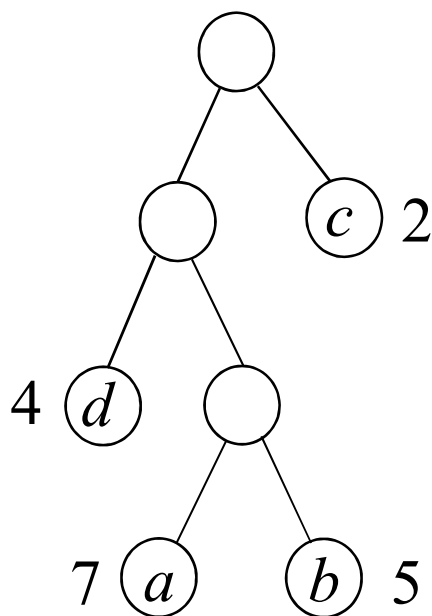
6.6 赫夫曼树及其应用

赫夫曼树

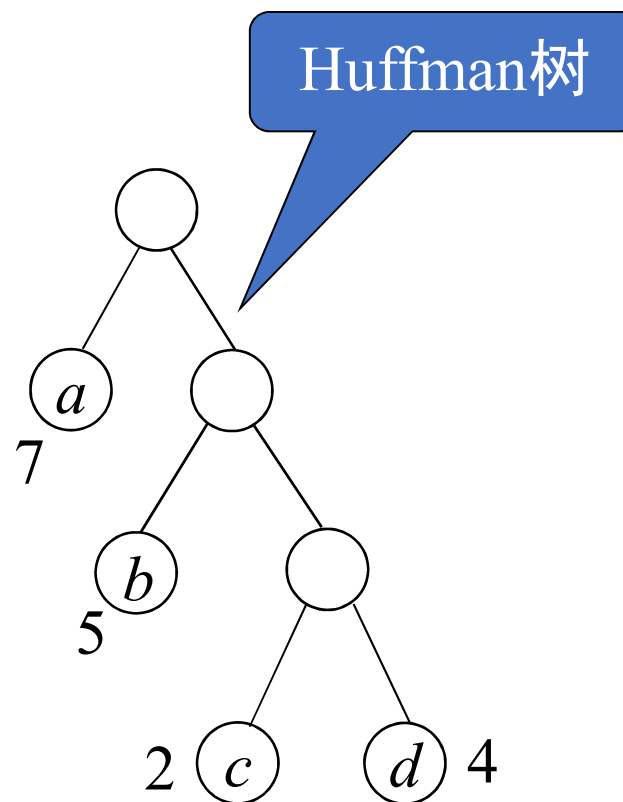
例：有四个叶子结点 a 、 b 、 c 、 d ，分别带权为7、5、2、4，由它们构成三棵不同的二叉树。



$$WPL=36$$



$$WPL=46$$



$$WPL=35$$

6.6 赫夫曼树及其应用

构造最优树(赫夫曼算法)

(1) 由给定的 n 个权值 $\{w_0, w_1, w_2, \dots, w_{n-1}\}$, 构造具有 n 棵二叉树的森林 $F = \{T_0, T_1, T_2, \dots, T_{n-1}\}$, 其中每一棵二叉树 T_i 只有一个带有权值 w_i 的根结点, 其左、右子树均为空。

(2) 重复以下步骤, 直到 F 中仅剩下一棵树为止:

① 在 F 中选取两棵根结点的权值最小的二叉树, 作为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

② 在 F 中删去这两棵二叉树。

③ 把新的二叉树加入 F 。



6.6 赫夫曼树及其应用

构造最优树(赫夫曼算法)

第一步: $9(a) \quad 4(b) \quad 5(c) \quad 2(d)$

第二步:

$9(a)$ $5(c)$

6 $4(b)$ $2(d)$

第三步:

第四步:

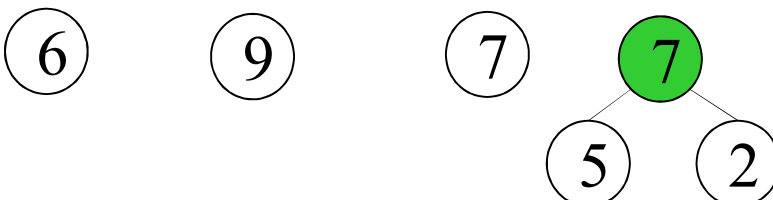
第四步:

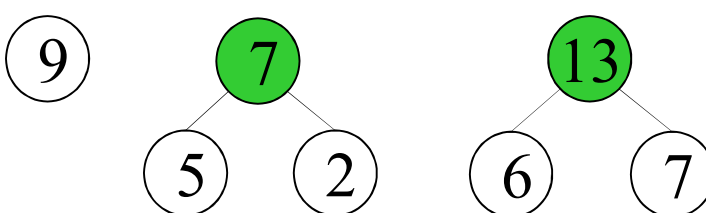
```
graph TD; Root(( )) ---|20| Node1(( )); Root ---|11| Node2(( )); Node1 ---|9| a((a)); Node2 ---|5| c((c)); Node2 ---|6| Node3(( )); Node3 ---|4| b1((b)); Node3 ---|2| d((d)); Root ---|4| b2((b));
```

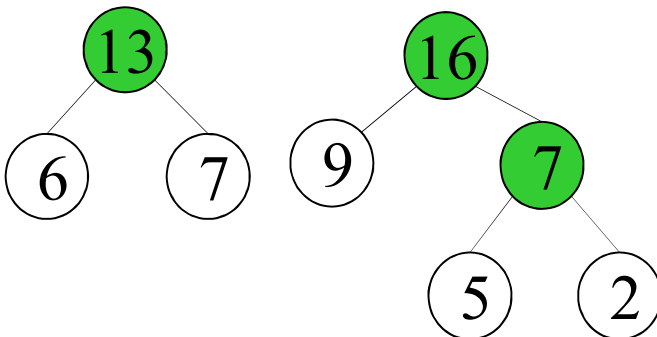
6.6 赫夫曼树及其应用

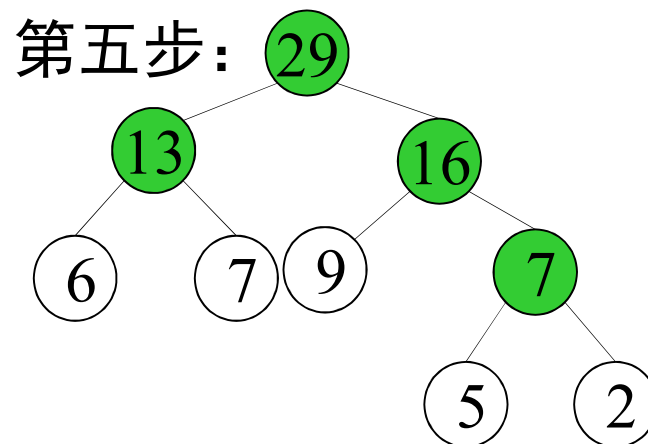
构造最优树(赫夫曼算法)

第一步: 

第二步: 

第三步: 

第四步: 

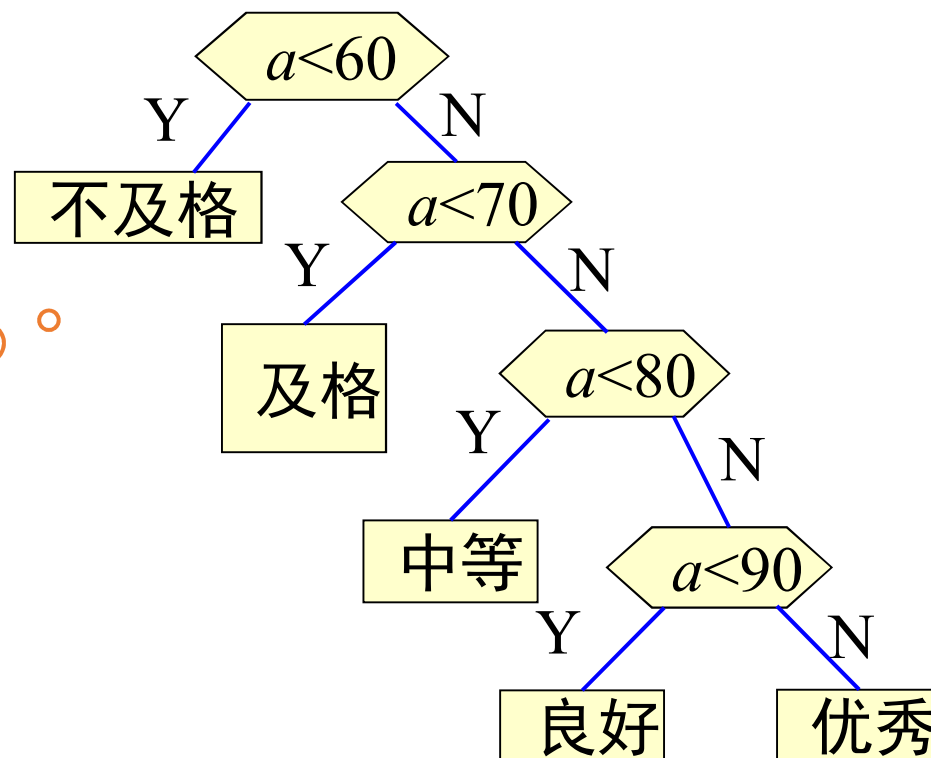
第五步: 

6.6 赫夫曼树及其应用

判定树 (赫夫曼树的应用之一)

分数	0—59	60—69	70—79	80—89	90—99
比例	0.05	0.15	0.4	0.3	0.1

比较次数: 1 2 3 4 4



输入10000个数据，则需进行31500次比较。

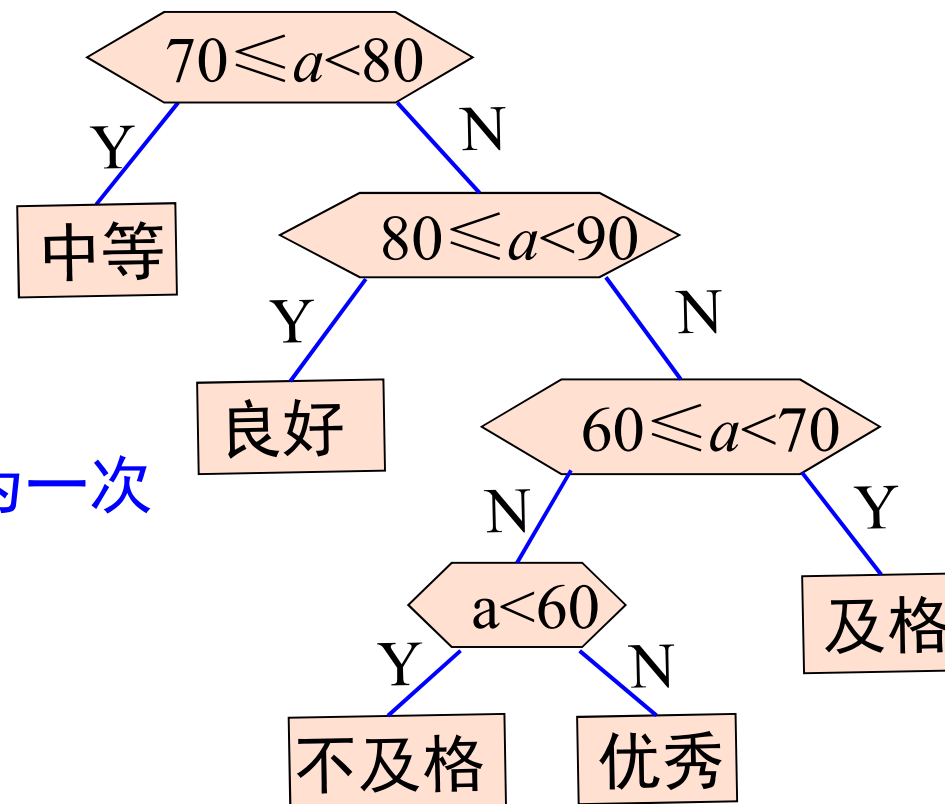
6.6 赫夫曼树及其应用

判定树 (赫夫曼树的应用之一)

分数	0—59	60—69	70—79	80—89	90—99
比例	0.05	0.15	0.4	0.3	0.1

以比例数为权构造一棵哈夫曼树，如(b)判断树所示。

再将每一比较框的两次比较改为一次

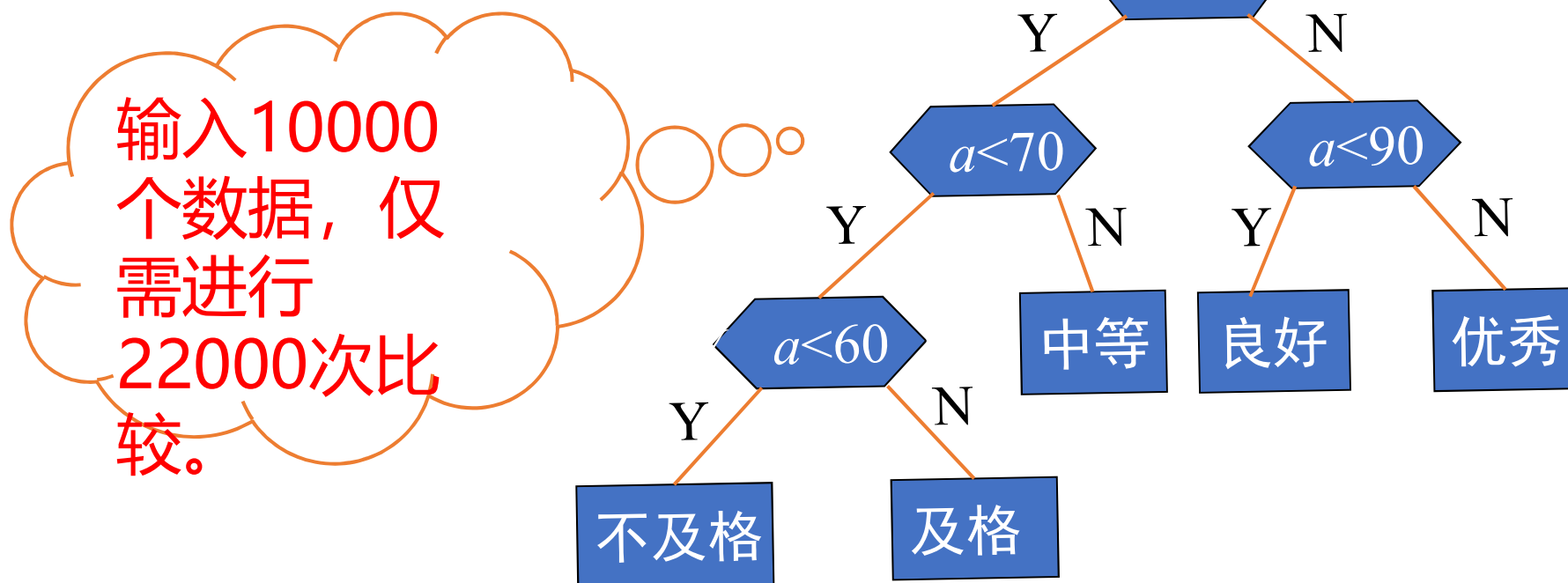


6.6 赫夫曼树及其应用

判定树 (赫夫曼树的应用之一)

分数	0—59	60—69	70—79	80—89	90—99
比例	0.05	0.15	0.4	0.3	0.1

比较次数: 3 3 2 2 2



6.6 赫夫曼树及其应用

赫夫曼编码(赫夫曼树的应用之二)

1) 二进制编码

通信中，可以采用0、1的不同排列来表示不同的字符，称为**二进制编码**。

发送端需要将电文中的字符序列转换成二进制的0、1序列，即**编码**；

接受端需要把接受的0、1序列转换成对应的字符序列，即**译码**。



6.6 赫夫曼树及其应用

赫夫曼编码(赫夫曼树的应用之二)

等长编码

A	B	C	D
00	01	10	11

发送: A B A C C C C D A B
00 01 00 10 10 10 10 11 00 01

不等长编码

让出现频率高的字符具有较短的编码, 让出现频率低的字符具有较长的编码, 缩短传送电文的总长度

A	B	C	D
1	10	0	01

发送: A B A C C C C D A B
1 10 1 0 0 0 0 01 1 10

?

无法译码! 为此引入前缀编码的概念



6.6 赫夫曼树及其应用

赫夫曼编码(赫夫曼树的应用之二)

2) 前缀编码

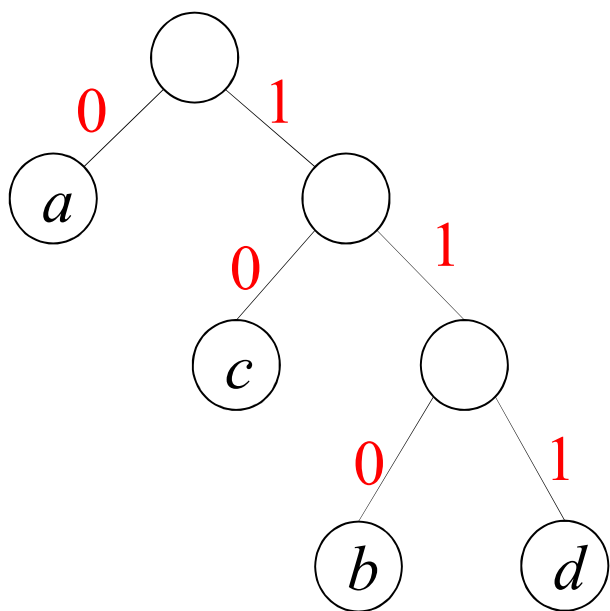
若对某一字符集进行不等长编码，则要求字符集中任一字符的编码都**不能是其他字符编码的前缀**。符合此要求的编码叫做**前缀编码**。



6.6 赫夫曼树及其应用

赫夫曼编码(赫夫曼树的应用之二)

利用二叉树设计二进制的前缀编码



假设有一棵如左图所示的二叉树，四个叶结点分别表示A、B、C、D四个字符，且约定左分支表示字符‘0’，右分支表示字符‘1’，则可以从根结点到叶子结点的路径上以分支字符组成的字符串作为该叶子结点的编码。可以证明，如此得到的必为二进制前缀编码。

6.6 赫夫曼树及其应用

赫夫曼编码(赫夫曼树的应用之二)

设计电文总长最短的二进制前缀编码即：

以 n 种字符出现的频率作权，设计一棵赫夫曼树的问题，
由此得到的二进制前缀编码称赫夫曼编码。



6.6 赫夫曼树及其应用

赫夫曼编码(赫夫曼树的应用之二)

例：设通信用的电文由字符集 $\{a, b, c, d, e, f, g, h\}$ 中的字母构成，这8个字母在电文中出现的概率分别为 $\{0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10\}$ ，试为这8个字母设计哈夫曼编码。



6.6 赫夫曼树及其应用

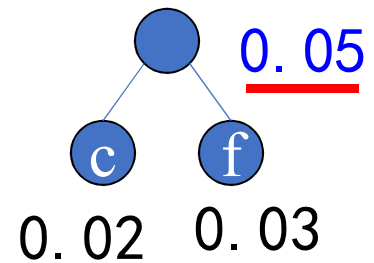
a	b	c	d	e	f	g	h
0.07	0.19	<u>0.02</u>	0.06	0.32	<u>0.03</u>	0.21	0.10

6.6 赫夫曼树及其应用

a b
0.07 0.19

d e
0.06 0.32

g h
0.21 0.10

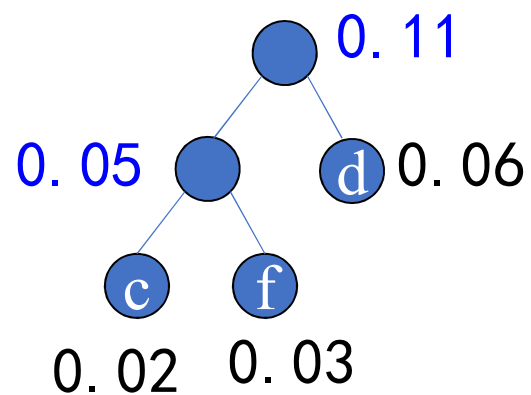


6.6 赫夫曼树及其应用

a 0.07 b 0.19

e 0.32

g 0.21 h 0.10

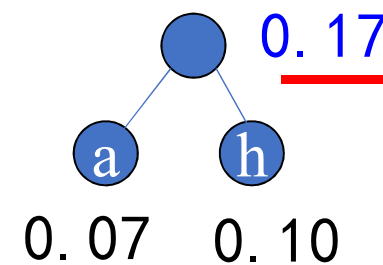
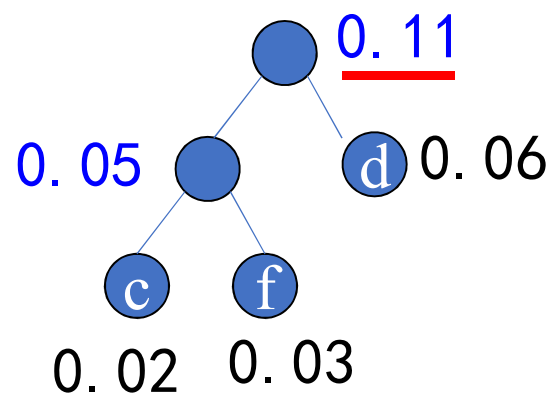


6.6 赫夫曼树及其应用

b
0.19

e
0.32

g
0.21

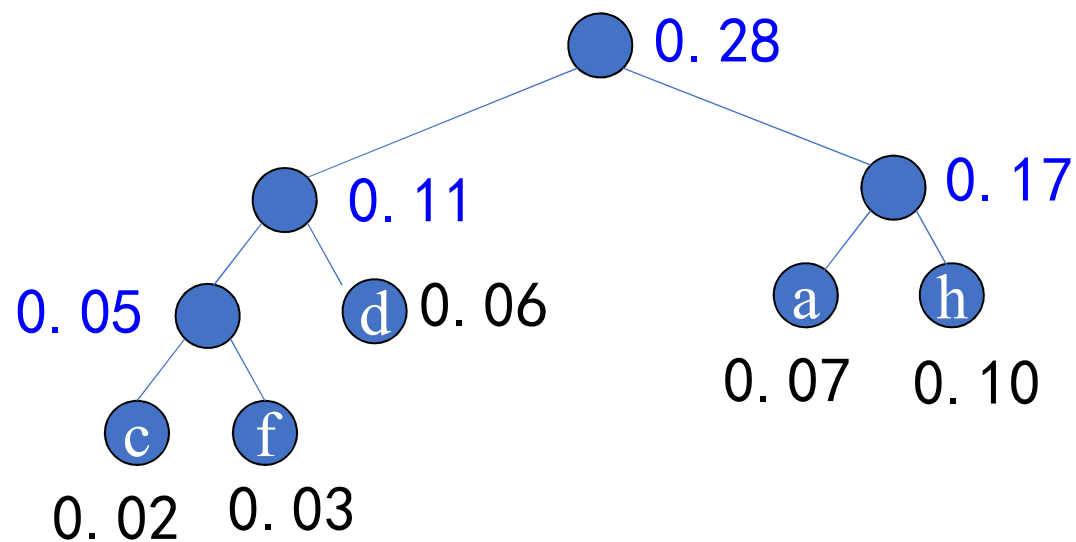


6.6 赫夫曼树及其应用

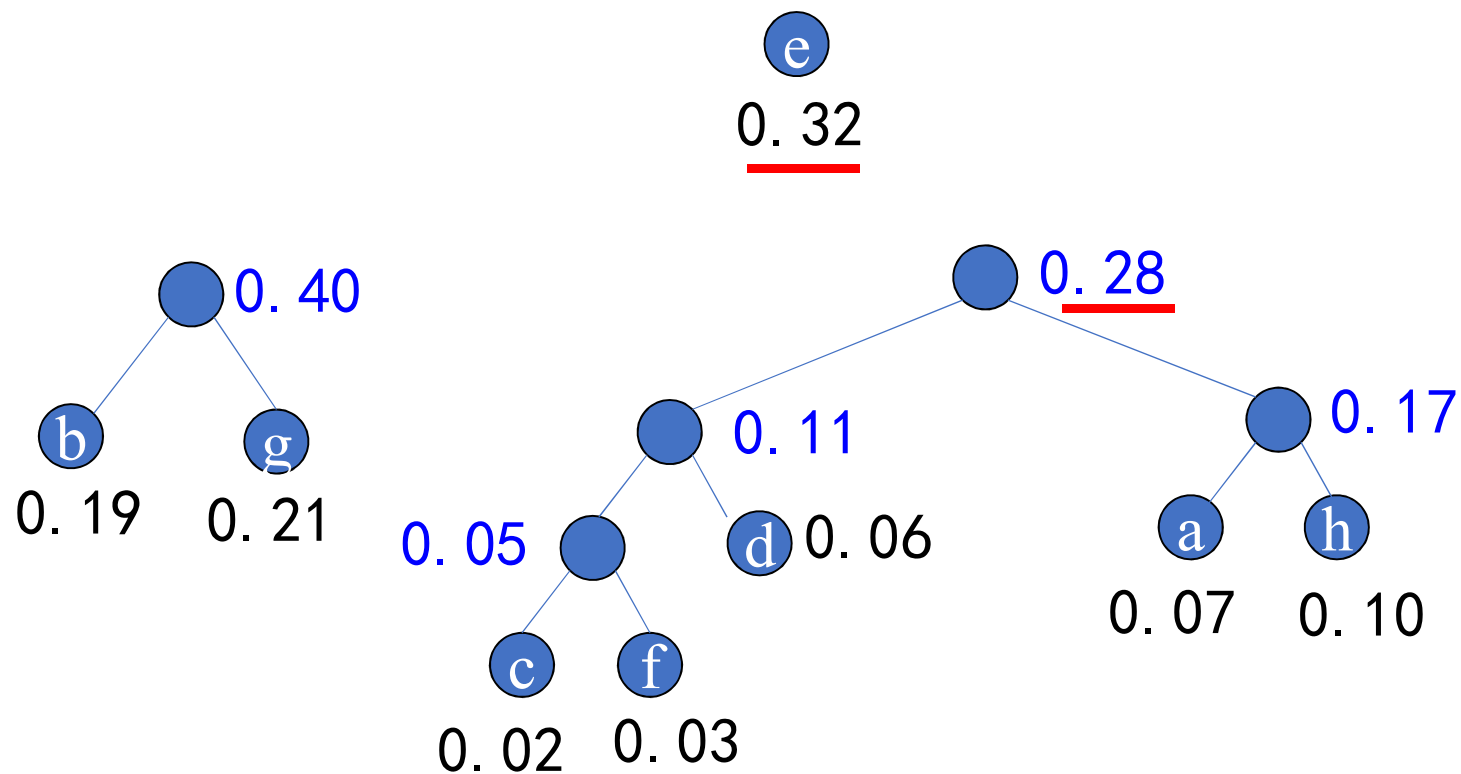
b
0.19

e
0.32

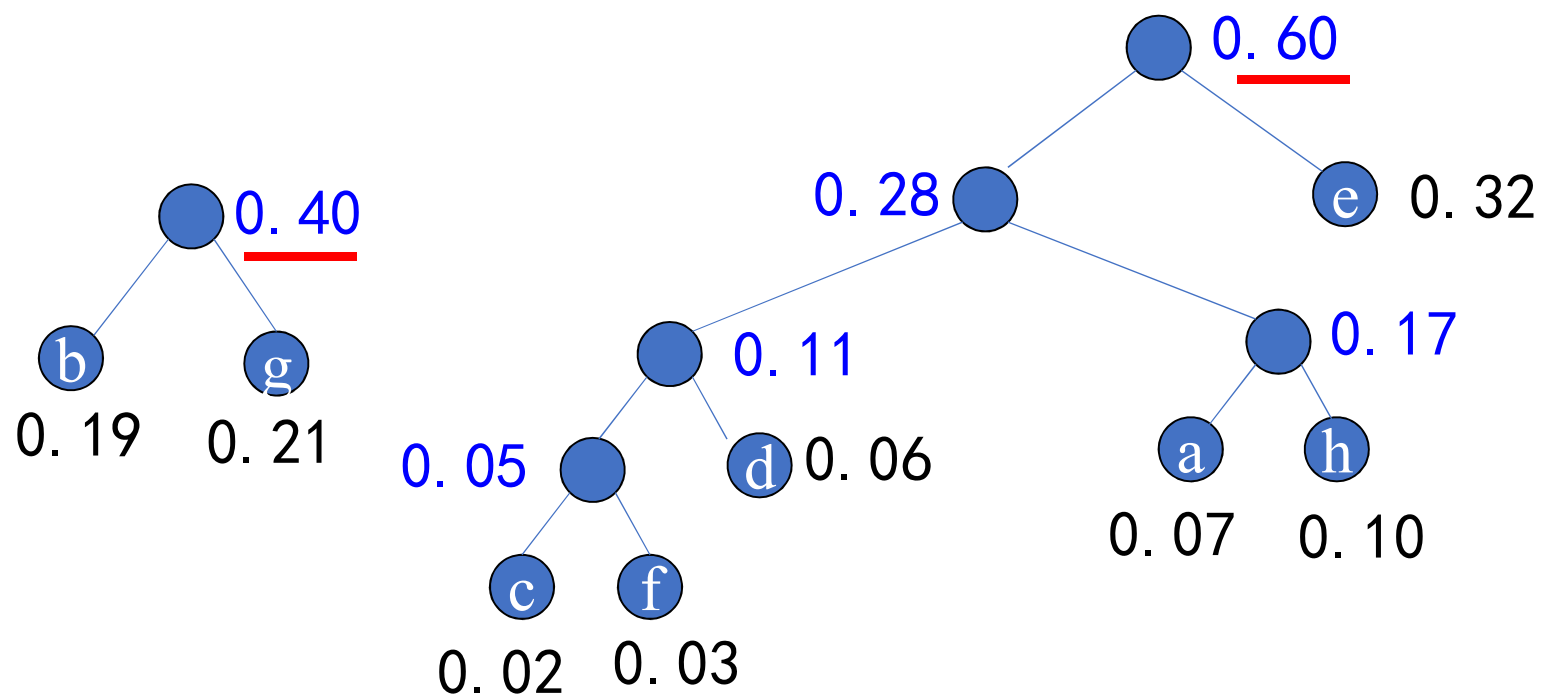
g
0.21



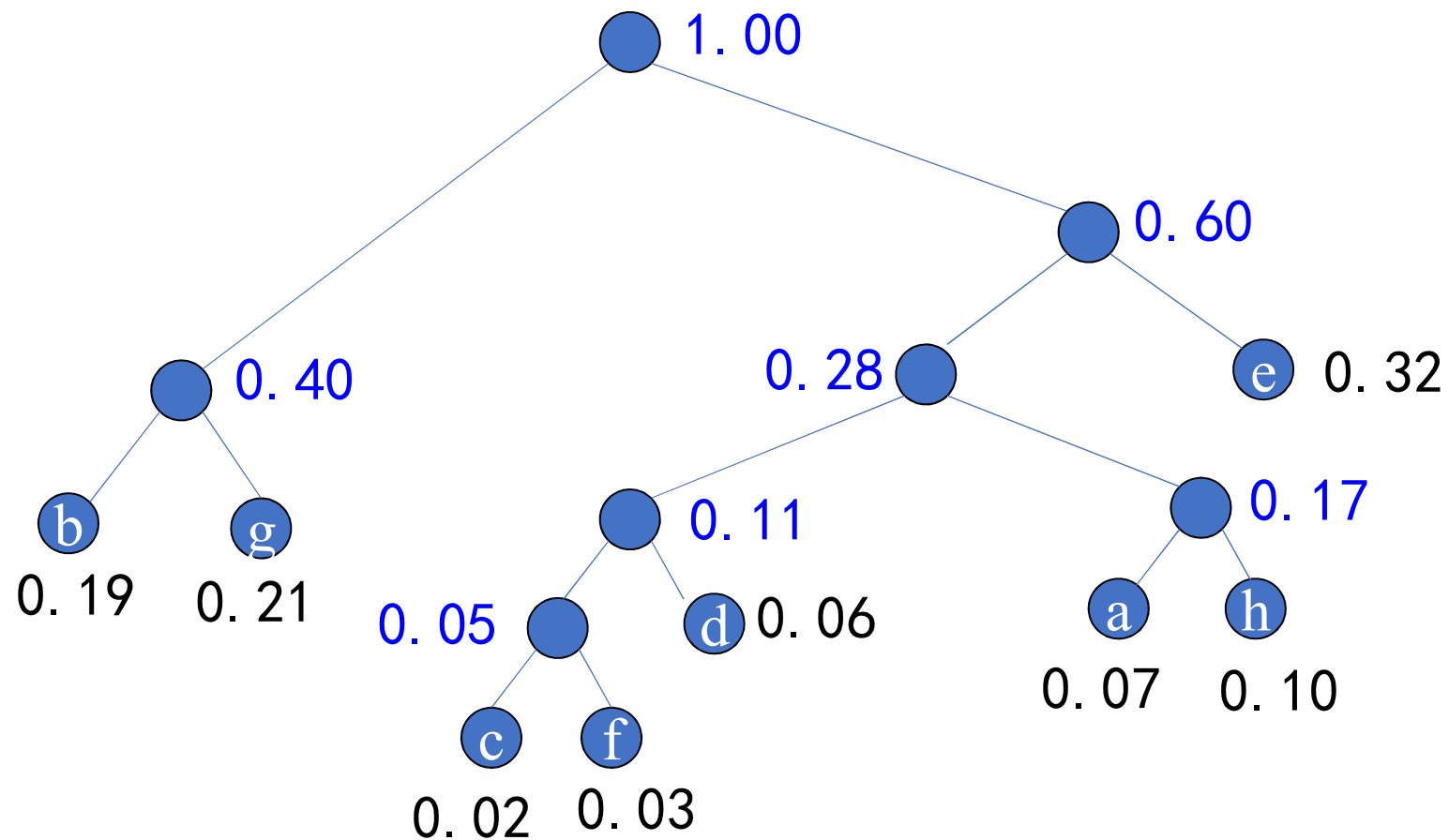
6.6 赫夫曼树及其应用



6.6 赫夫曼树及其应用



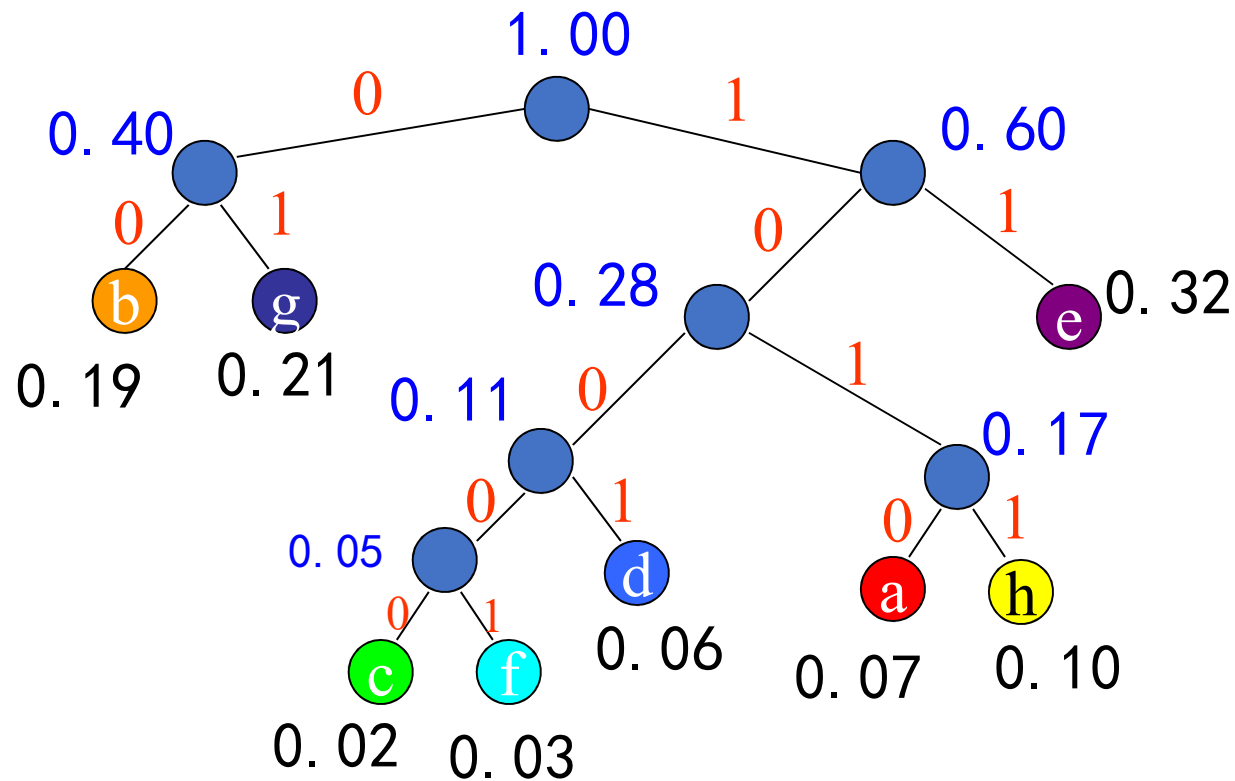
6.6 赫夫曼树及其应用



6.6 赫夫曼树及其应用

$\{a, b, c, d, e, f, g, h\}$

$\{0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10\}$



a = 1010

b = 00

c = 10000

d = 1001

e = 11

f = 10001

g = 01

h = 1011

6.6 赫夫曼树及其应用

对应的哈夫曼
编码（左0右1）

字符	编码	频率
a	1010	0.07
b	00	0.19
c	10000	0.02
d	1001	0.06
e	11	0.32
f	10001	0.03
g	01	0.21
h	1011	0.10

字符	编码	频率
a	000	0.07
b	001	0.19
c	010	0.02
d	011	0.06
e	100	0.32
f	101	0.03
g	110	0.21
h	111	0.10

Huffman码的WPL = $2 \times (0.19 + 0.32 + 0.21) + 4 \times (0.07 + 0.06 + 0.10) + 5 \times (0.02 + 0.03) = 1.44 + 0.92 + 0.25 = 2.61$

二进制码 WPL = $3 \times (0.19 + 0.32 + 0.21 + 0.07 + 0.06 + 0.10 + 0.02 + 0.03) = 3$



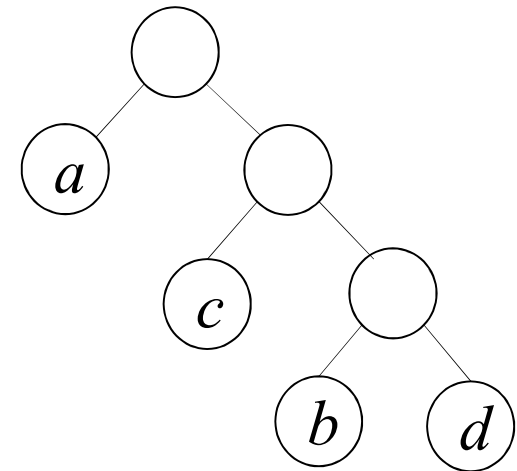
6.6 赫夫曼树及其应用

```
typedef struct{  
    unsigned int  weight;  
    unsigned int  parent, lchild, rchild;  
}HTNode, *HuffmanTree;  
  
typedef char ** HuffmanCode;
```



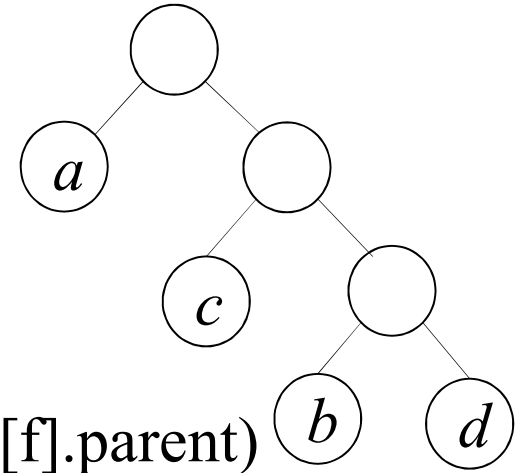
6.6 赫夫曼树及其应用

```
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n ) {  
1    if (n<=1) return;  
2    m = 2*n - 1;  
3    HT = (HuffmanTree)malloc((m+1)*sizeof(HTNode));  
4    for (p=HT+1, i=1; i<=n; ++i, ++p, ++w) *p={*w, 0, 0, 0};  
5    for ( ; i<=m; ++i, ++p) *p={0, 0, 0, 0};  
6    for (i=n+1; i<=m; ++i){  
7        Select(HT, i-1, s1, s2);  
8        HT[s1].parent = i;  HT[s2].parent = i;  
9        HT[i].lchild = s1;  HT[i].rchild = s2;  
10       HT[i].weight = HT[s1].weight + HT[s2].weight;  
11    }
```



6.6 赫夫曼树及其应用

```
12  HC = (HuffmanCode)malloc((n+1)*sizeof(char *));
13  cd = (char *)malloc(n*sizeof(char));
14  cd[n-1] = '\0';
15  for (i=1; i<=n; ++i) {
16      start = n-1;
17      for (c=i, f=HT[i].parent; f!=0; c=f, f=HT[f].parent)
18          if (HT[f].lchild==c) cd[--start]='0';
19          else cd[--start]='1';
20      HC[i]= (char *)malloc((n-start)*sizeof(char));
21      strcpy(HC[i], &cd[start]);
22  }
23  free(cd);
24} // HuffmanCoding
```



■ 本章学习要点

- 熟练掌握二叉树的结构特性，了解相应的证明方法；
- 熟悉二叉树的各种存储结构的特点和适用范围；
- 掌握二叉树遍历的各种算法和操作；
- 理解线索二叉树的实质和具体过程；
- 熟悉树的各种存储结构及特点，掌握树和森林与二叉树的转换方法；
- 掌握建立最优树和哈夫曼编码的方法。





燕山大学
YANSHAN UNIVERSITY

Thank you!

谢谢大家！ Q&A

