

数据结构与算法

树和二叉树

树和二叉树

6. 1 树的基本概念

 6. 1. 1 树的定义

 6. 1. 2 基本术语

6. 2 二叉树

 6. 2. 1 二叉树的定义

 6. 2. 2 二叉树的性质

 6. 2. 3 二叉树的存储结构

6. 3 二叉树遍历

 6. 3. 1 遍历的定义

 6. 3. 2 遍历算法

 6. 3. 3 二叉树遍历应用举例

6. 4 线索二叉树

树和二叉树

6.5 树和森林

6.5.1 树的存储表示

6.5.2 森林的存储表示

6.5.3 树和森林的遍历

6.5.4 树和森林与二叉树的转换

6.6 哈夫曼树与哈夫曼编码

6.6.1 哈夫曼树的基本概念

6.6.2 哈夫曼树构造算法

6.6.3 哈夫曼编码

6.7 树的计数

6.1 树的基本概念

- 树型结构是一类重要的非线性结构。
- 树型结构是结点之间有分支，并且具有层次关系的结构，它非常类似于自然界中的树。
- 树结构在客观世界是大量存在的，例如家谱、行政组织机构都可用树形象地表示。

6.1 树的基本概念

- 树在计算机领域中也有着广泛的应用，例如在编译程序中，用树来表示源程序的语法结构；在数据库系统中，可用树来组织信息；在分析算法的行为时，可用树来描述其执行过程。等等。

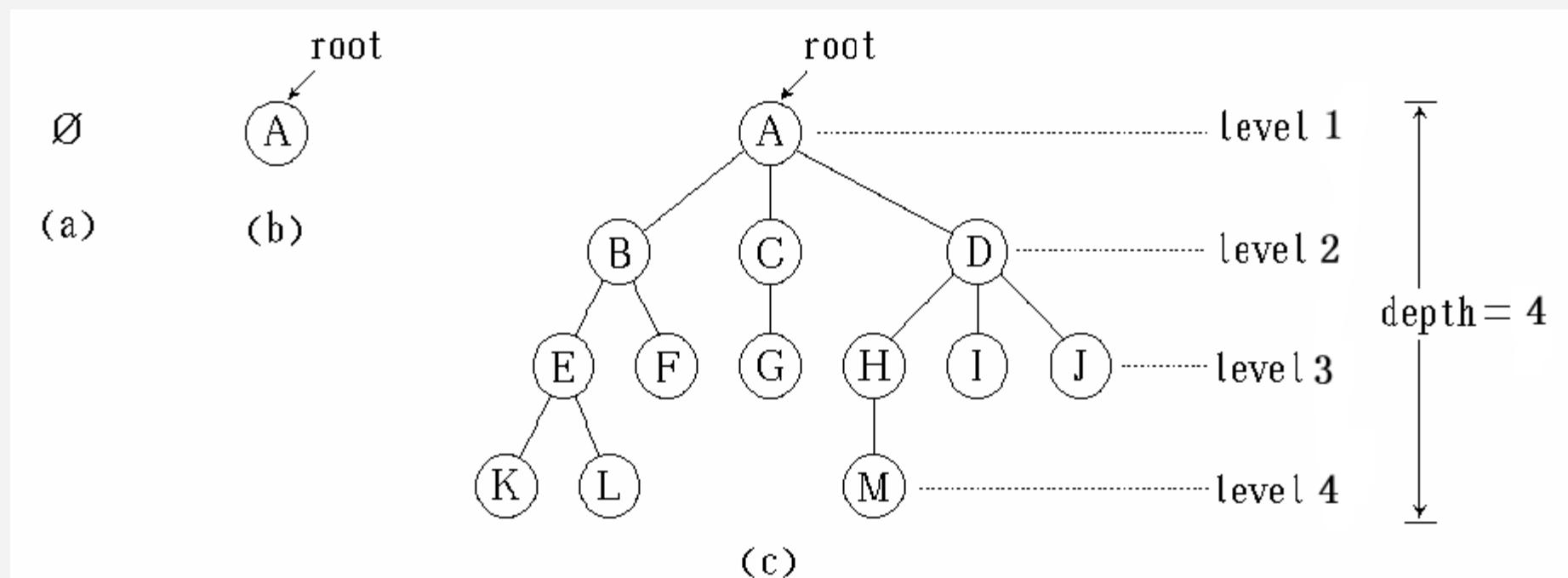
6.1.1 树的定义

树的定义：树是 n ($n \geq 0$) 个结点的有限集合 T 。
当 $n=0$ 时，称为空树；当 $n>0$ 时，该集合满足如下条件：

- 有且仅有一个称为根 (root) 的特定结点，它没有直接前驱，但有零个或多个直接后继。
- 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, \dots, T_m ，其中每一子集本身又是符合本定义的树，称为根root的子树。每棵子树的根结点有且仅有一个直接前驱，但有零个或多个直接后继。

6.1.1 树的定义

树的定义是一个递归的定义，即在树的定义中又用到树的概念，它表明了树的固有特性。树还有其他的表示形式，比如嵌套集合的表示形式，广义表的表示形式，凹入表示法等等。



6.1.2 基本术语

- 结点 (node)
- 结点的度 (degree)
- 分支 (branch) 结点
- 叶子 (leaf)
- 孩子 (child) 结点
- 双亲 (parent) 结点
- 兄弟 (sibling) 结点
- 祖先 (ancestor) 结点
- 子孙 (descendant) 结点
- 结点所处层次 (level)
- 树的深度 (depth)
- 树的度 (degree)
- 有序树
- 无序树
- 森林
- 路径

6.1.2 基本术语

- 结点：数据元素+若干指向子树的分支。
- 结点的度：一个结点的子树个数。
- 树的度：树中所有结点的度的最大值。
- 叶子节点：度为0的结点，即无后继的结点，也称为终端结点。
- 分支结点：度不为0的结点，也称为非终端结点。

6.1.2 基本术语

- 结点的层次：从根结点开始定义，根结点的层次为1，根的直接后继的层次为2，依此类推。
- 树的深度：树中所有结点的层次的最大值。树的深度也称为高度。
- 路径：从树的一个结点到另一个结点的分支构成这两个结点之间的路径。

6.1.2 基本术语

- 孩子结点：一个结点的子树的根称为该结点的孩子结点。
- 双亲结点：一个结点的直接前驱称为该结点的双亲结点。
- 兄弟结点：同一双亲结点的孩子结点之间互称兄弟。
- 祖先结点：一个结点的祖先是指从根结点到该结点的路径上的所有结点。
- 子孙结点：以某结点为根的子树中的任一结点都称为该结点的子孙。
- 堂兄弟结点：其双亲在同一层的结点互为堂兄弟。

6.1.2 基本术语

- 有序树：如果将树中结点的各子树看成从左至右是有序的（即不能互换），则称该树为有序树，否则称为无序树。
- 无序树：子树之间不存在确定的次序关系。

6.1.2 基本术语

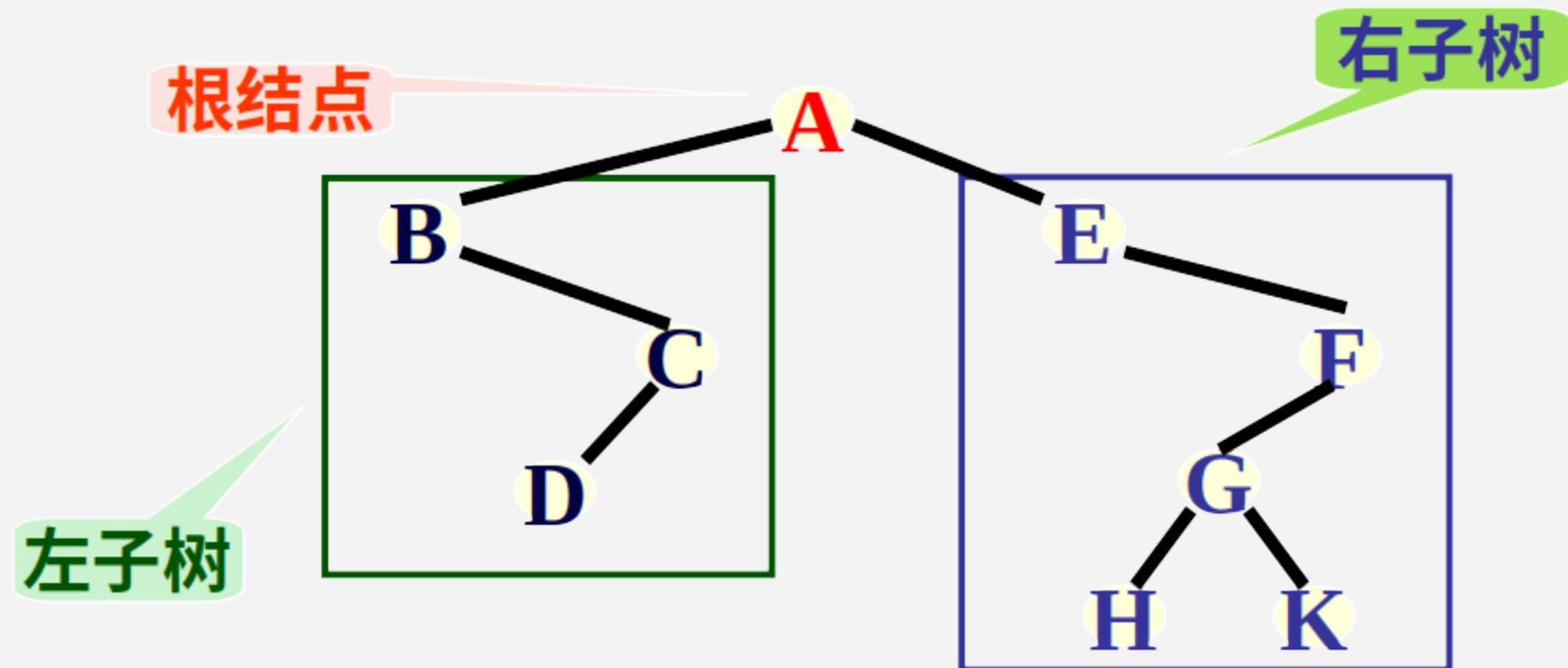
- 森林：是 $m(m \geq 0)$ 棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。由此，也可以用森林和树互相递归的定义来描述树。任何一棵非空树是一个二元组： $\text{Tree} = (\text{root}, F)$ ，其中 root 被称为根结点， F 被称为子树森林。

6.2 二叉树

- 二叉树在树结构的应用中起着非常重要的作用，二叉树是一种特殊的树，比较适合计算机处理，任何树和森林都可以转化为二叉树，这样就解决了树的存储结构及其运算中存在的复杂性。

6.2.1 二叉树的定义

二叉树或为空树，或是由一个根结点加上两棵分别称为左子树和右子树的、互不交叉的二叉树组成。



6.2.1 二叉树的定义

- 二叉树定义：我们把满足以下两个条件的树形结构叫做二叉树（Binary Tree）：
 - (1) 每个结点的度都不大于2；
 - (2) 每个结点的孩子结点次序不能任意颠倒。
- 这也是一个递归定义。二叉树可以是空集合，根可以有空的左子树或空的右子树。

6.2.1 二叉树的定义

- 二叉树结点的子树要区分为左子树和右子树，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。这是二叉树与树的最主要的差别。

6.2.1 二叉树的定义

二叉树的5种基本形态



(a) 空二叉树



(b) 只有根结点
的二叉树



(c) 只有左子树
的二叉树



(d) 左右子树均非
空的二叉树



(e) 只有右子树的
二叉树

二叉树的主要基本操作

(1) `BinTreeNode<ElemType> *GetRoot() const`

初始条件：二叉树已存在。

操作结果：返回二叉树的根。

(2) `bool Empty() const`

初始条件：二叉树已存在。

操作结果：如二叉树为空，则返回true，否则返回false。

(3) `StatusCode GetElem(TreeNode<ElemType> *cur, ElemType &e) const`

初始条件：二叉树已存在，cur为二叉树的一个结点。

操作结果：用e返回结点cur元素值，如果不存在结点cur，返回NOT_PRESENT，否则返回ENTRY_FOUND。

(4) `StatusCode SetElem(TreeNode<ElemType> *cur, const ElemType &e)`

初始条件：二叉树已存在，cur为二叉树的一个结点。

操作结果：如果存在结点cur，则返回FAIL，否则返回SUCCESS，并将结点cur的值设置为e。

二叉树的主要基本操作

(5) void InOrder(void (*visit)(const ElemType &)) const

初始条件：二叉树已存在。

操作结果：中序遍历二对树，对每个结点调用函数(*visit)。

(6) void PreOrder(void (*visit)(const ElemType &)) const

初始条件：二叉树已存在。

操作结果：先序遍历二对树，对每个结点调用函数(*visit)。

(7) void PostOrder(void (*visit)(const ElemType &)) const

初始条件：二叉树已存在。

操作结果：后序遍历二对树，对每个结点调用函数(*visit)。

(8) void LevelOrder(void (*visit)(const ElemType &)) const

初始条件：二叉树已存在。

操作结果：层次遍历二对树，对每个结点调用函数(*visit)。

(9) int NodeCount() const

初始条件：二叉树已存在。

操作结果：返回二叉树的结点个数。

二叉树的主要基本操作

(10) `BinTreeNode<ElemType> *LeftChild(const BinTreeNode<ElemType> *cur) const;`

初始条件：二叉树已存在， cur是二叉树的一个结点。

操作结果：返回二叉树结点cur的左孩子。

(11) `BinTreeNode<ElemType> *RightChild(const BinTreeNode<ElemType> *cur) const`

初始条件：二叉树已存在， cur是二叉树的一个结点。

操作结果：返回二叉树结点cur的右孩子。

(12) `BinTreeNode<ElemType> *Parent(const BinTreeNode<ElemType> *cur) const`

初始条件：二叉树已存在， cur是二叉树的一个结点。

操作结果：返回二叉树结点cur的双亲结点。

二叉树的主要基本操作

(13) void InsertLeftChild(BinTreeNode<ElemType> *cur, const ElemType &e)
初始条件：二叉树已存在， cur是二叉树的一个结点， e为一个数据元素，并且cur
非空。

操作结果：插入e为cur的左孩子，如果cur的左孩子非空，则cur原有左子树成为e
的左子树。

(14) void InsertRightChild(BinTreeNode<ElemType> *cur, const ElemType &e)
初始条件：二叉树已存在， cur是二叉树的一个结点， e为一个数据元素，并且cur
非空。

操作结果：插入e为cur的右孩子，如果cur的右孩子非空，则cur原有右子树成为e
的右子树。

(15) void DeleteLeftChild(BinTreeNode<ElemType> *cur)

初始条件：二叉树已存在， cur是二叉树的一个结点。

操作结果：删除二叉树结点cur的左子树。

二叉树的主要基本操作

(16) void DeleteRightChild(BinTreeNode<ElemType> *cur)

初始条件：二叉树已存在， cur是二叉树的一个结点。

操作结果：删除二叉树结点cur的右子树。

(17) int Height() const

初始条件：二叉树已存在。

操作结果：返回二叉树的高。

6.2.2 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($i \geq 1$)。

证明：用数学归纳法。 ?

- 归纳基础：当*i*=1时，整个二叉树只有一根结点，此时 $2^{i-1}=2^0=1$ ，结论成立。
- 归纳假设：假设*i*=*k*时结论成立，即第*k*层上结点总数最多为 2^{k-1} 个。
- 现证明当*i*=*k*+1时，结论成立： ?
- 因为二叉树中每个结点的度最大为2，则第*k*+1层的结点总数最多为第*k*层上结点最大数的2倍，即 $2 \times 2^{k-1}=2^{(k+1)-1}$ ，故结论成立。

6.2.2 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
 $(k \geq 1)$ 。

证明：基于上一条性质，深度为 k 的二叉树上的结点数至多为：

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

6.2.2 二叉树的性质

性质3：对任意一棵二叉树，若它含有 n_0 个叶子结点， n_2 个度为2的结点，则 $n_0 = n_2 + 1$

证明：设度为1的结点数为 n_1

二叉树中，除根节点外，每个节点都有且仅有一条父节点指向它的分支，固：

$$\text{分支 } B = n_0 + n_1 + n_2 - 1$$

又，根据节点度计算：

$$\text{分支 } B = 0 \times n_0 + 1 \times n_1 + 2 \times n_2$$

$$\text{所以 } n_0 + n_1 + n_2 - 1 = 0 \times n_0 + 1 \times n_1 + 2 \times n_2$$

$$\text{由此可得: } n_0 = n_2 + 1.$$

6.2.2 二叉树的性质

满二叉树

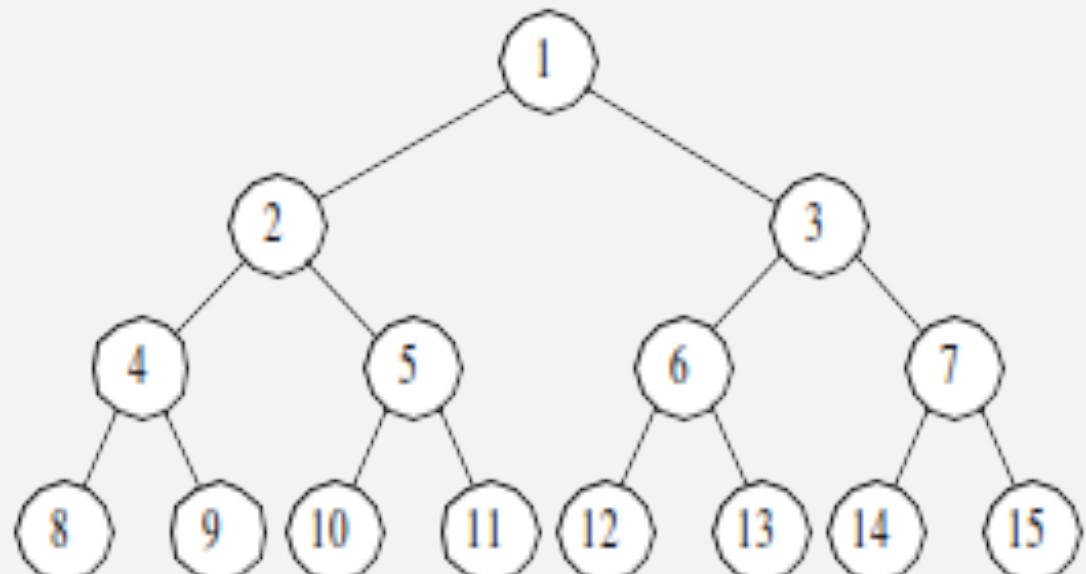
- 深度为 k 且有 $2^k - 1$ 个结点的二叉树。在满二叉树中，每层结点都是满的，即每层结点都具有最大结点数。后图所示的二叉树，即为一棵满二叉树。
- 满二叉树的顺序表示，即从二叉树的根开始，层间从上到下，层内从左到右，逐层进行编号（1, 2, …, n）。例如图所示的满二叉树的顺序表示为(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)。

6.2.2 二叉树的性质

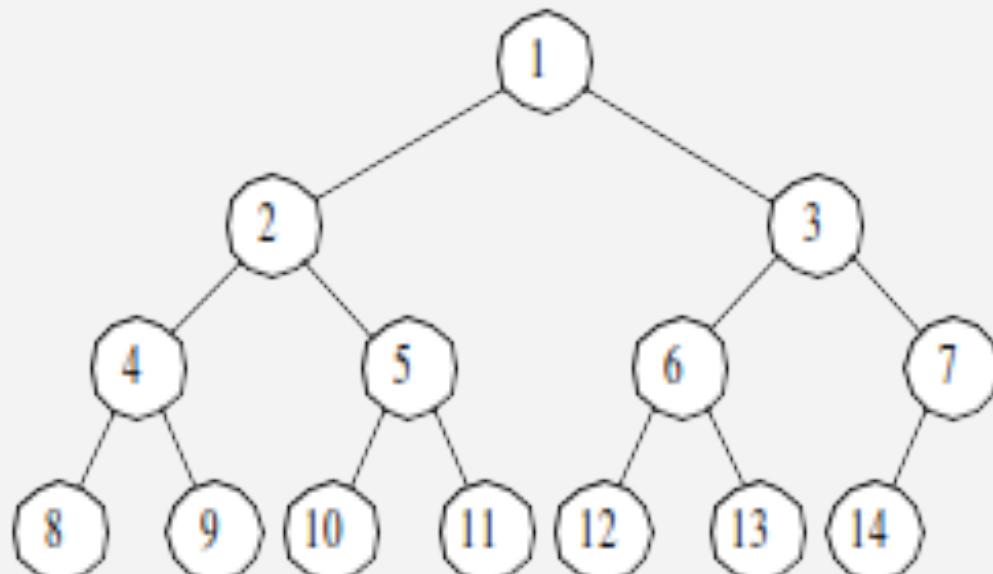
完全二叉树

- 深度为 k , 结点数为 n 的二叉树, 如果其结点 $1 \sim n$ 的位置序号分别与满二叉树的结点 $1 \sim n$ 的位置序号一一对应, 则为完全二叉树, 如图所示。
- 满二叉树必为完全二叉树, 而完全二叉树不一定满二叉树。
- 完全二叉树的特点是:
 1. 所有的叶结点都出现在第 k 层或 $k-1$ 层。
 2. 任一结点, 如果其右子树的最大层次为 L , 则其左子树的最大层次为 L 或 $L+1$ 。

6.2.2 二叉树的性质



(a) 满二叉树



(b) 完全二叉树

6.2.2 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 $\lfloor \log_2 n \rfloor + 1$ 。

证明：假设n个结点的完全二叉树的深度为k，根据性质2可知，
k-1层满二叉树的结点总数为：

$$n_1 = 2^{k-1} - 1$$

k层满二叉树的结点总数为：

$$n_2 = 2^k - 1$$

显然有 $n_1 < n \leq n_2$ ，进一步可以推出 $n_1 + 1 \leq n < n_2 + 1$ 。

将 $n_1 = 2^{k-1} - 1$ 和 $n_2 = 2^k - 1$ 代入上式，可得 $2^{k-1} \leq n < 2^k$ ，即
 $k-1 \leq \log_2 n < k$ 。 ?

因为k是整数，所以 $k-1 = \lfloor \log_2 n \rfloor$ ， $k = \lfloor \log_2 n \rfloor + 1$ ，故结论成立。

6.2.2 二叉树的性质

性质5：对于具有n个结点的完全二叉树，如果按照从上到下和从左到右的顺序对二叉树中的所有结点从1开始顺序编号，则对于任意的序号为i的结点有：

- 如 $i=1$ ，则序号为i的结点是根结点，无双亲结点；如 $i>1$ ，则序号为i的结点的双亲结点序号为 $\lfloor i/2 \rfloor$ 。
- 如 $2 \times i > n$ ，则序号为i的结点无左孩子；如 $2 \times i \leq n$ ，则序号为i的结点的左孩子结点的序号为 $2 \times i$ 。
- 如 $2 \times i + 1 > n$ ，则序号为i的结点无右孩子；如 $2 \times i + 1 \leq n$ ，则序号为i的结点的右孩子结点的序号为 $2 \times i + 1$ 。

练习

已知一棵完全二叉树的第6层（设根为第1层）有8个叶结点，则完全二叉树的结点个数最多是（ C ）

- A. 39
- B. 52
- C. 111
- D. 119

（注意：并没有说这个二叉树有几层，自己构造二叉树的形态）

练习

在一棵度为4的树T中，若有20个度为4的结点，10个度为3的结点，1个度为2的结点，10个度为1的结点，则树T的叶节点个数是 (B)

- A. 41 B. 82 C. 113 D. 122

$$(N=20+10+10+1+n_0;$$

$$20*4+10*3+2+10=N-1)$$

练习

若一棵完全二叉树有 768 个结点，则该二叉树中叶结点的个数是

- A. 257
- B. 258
- C. 384
- D. 385

参考答案： C

$$(N_0=m, N_2=m-1, N_1=1 \text{ or } 0)$$

6.2.3 二叉树的储存结构

- 二叉树的结构是非线性的，每一结点最多可有两个后继。
- 二叉树的存储结构有两种：顺序存储结构和链式存储结构。
- 二叉树的存储结构应能体现二叉树的逻辑关系。

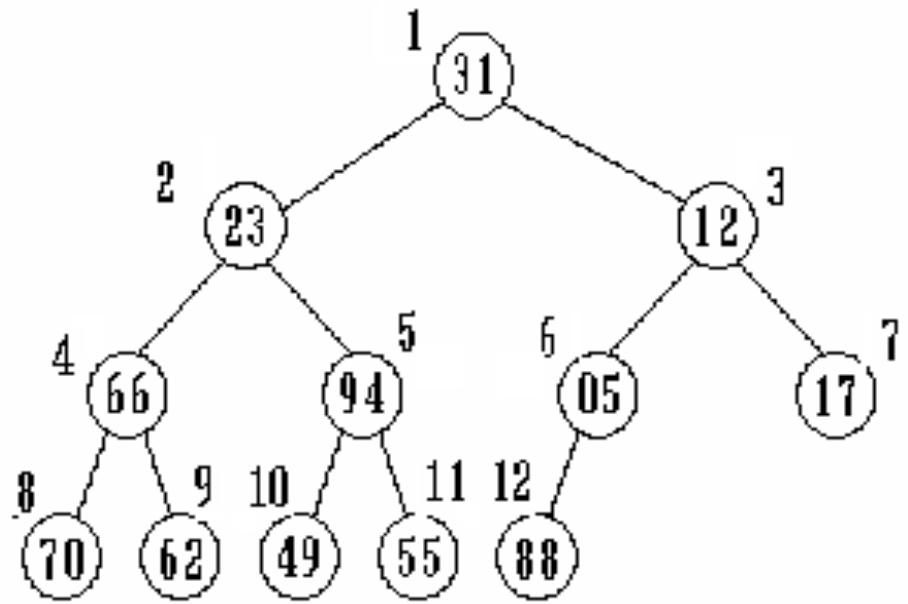
6.2.3 二叉树的储存结构

- 顺序存储结构：用一组连续的存储单元存储二叉树的数据元素。
- 因此，必须把二叉树的所有结点安排成为一个恰当的序列，结点在这个序列中的相互位置能反映出结点之间的逻辑关系，用互编址的方法，从树根起，自上层至下层，每层自左至右的给所有结点编号。

6.2.3 二叉树的储存结构

- 对于完全二叉树，若已知结点的编号，可以推算它双亲和孩子结点的编号。
- 缺点是有可能对存储空间造成极大的浪费，在最坏的情况下，一个深度为H且只有H个结点的右单支树需要 $2^h - 1$ 个结点存储空间。而且，若经常需要插入与删除树中结点时，顺序存储方式不是很好。

6.2.3 二叉树的储存结构

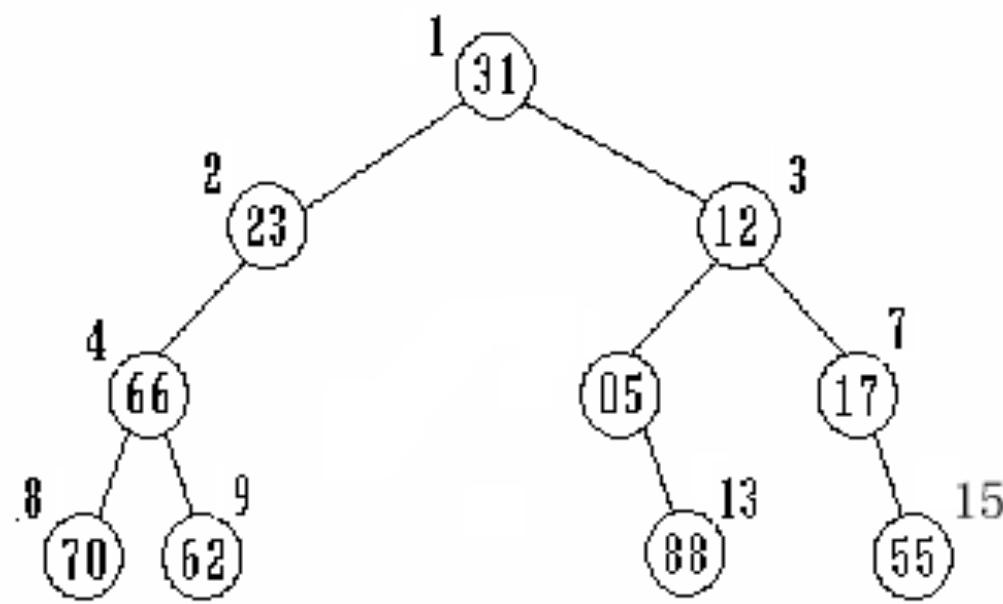


(a)

| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 31 | 23 | 12 | 66 | 94 | 05 | 17 | 70 | 62 | 49 | 55 | 88 |

(b)

完全二叉树的数组表示



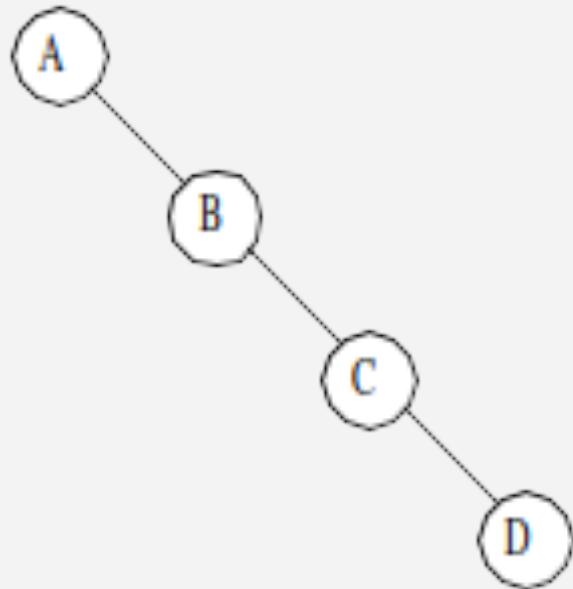
(a)

| | | | | | | | | | | | | | | |
|----|----|----|----|---|----|----|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 31 | 23 | 12 | 66 | | 05 | 17 | 70 | 62 | | | | 88 | | 55 |

(b)

一般二叉树的数组表示

6.2.3 二叉树的储存结构



(a) 单支二叉树



(b) 顺序存储结构

单支二叉树与其顺序存储结构

6.2.3 二叉树的储存结构

链式存储结构

- 设计不同的结点结构可以构成不同形式的链式存储结构。
- 二叉树的结点由一个数据元素和分别指向其左、右子树的两个分支构成，则表示二叉树的链表中的结点至少包含3个域：数据域和左、右指针域。
- 利用这种结点结构所得到的二叉树存储结构称为二叉链表。最常使用二叉链表作二叉树的存储结构。

6.2.3 二叉树的储存结构

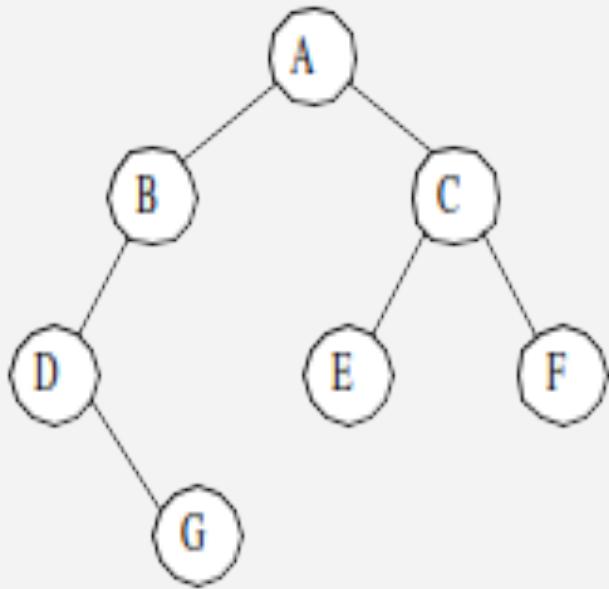
对于任意的二叉树来说，每个结点只有两个孩子，一个双亲结点。我们可以设计每个结点至少包括三个域：数据域、左孩子域和右孩子：

| | | |
|--------|------|--------|
| LChild | Data | RChild |
|--------|------|--------|

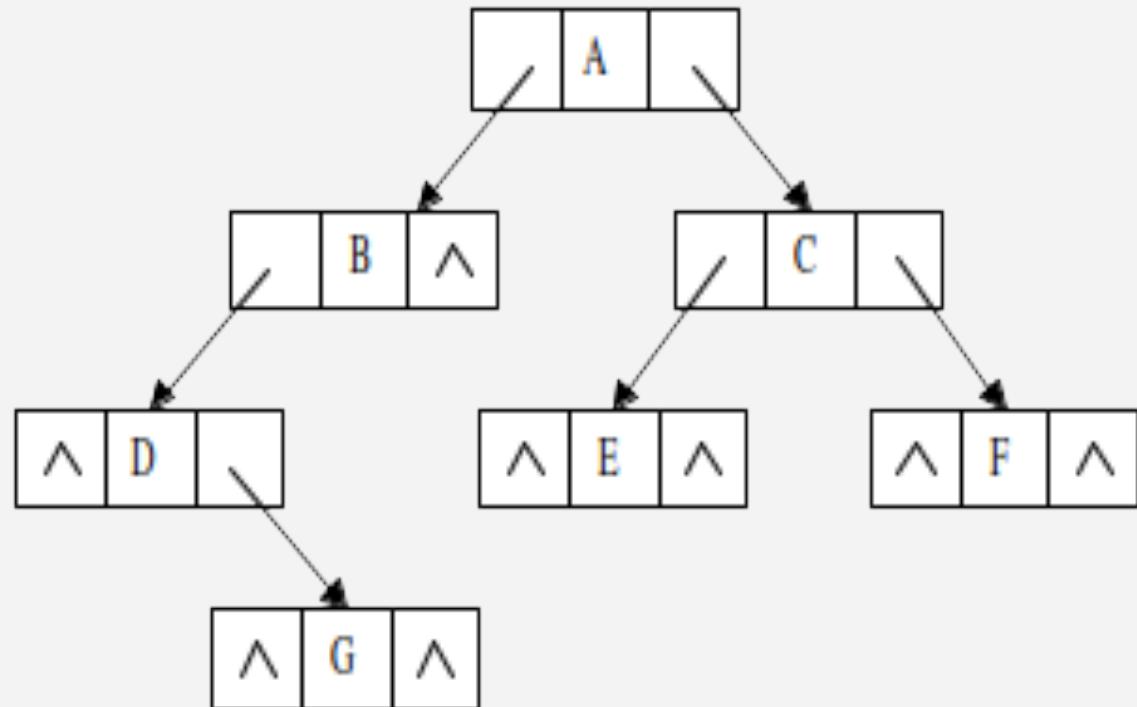
其中，LChild域指向该结点的左孩子，Data域记录该结点的信息，RChild域指向该结点的右孩子。

用C语言可以这样声明二叉树的二叉链表结点的结构

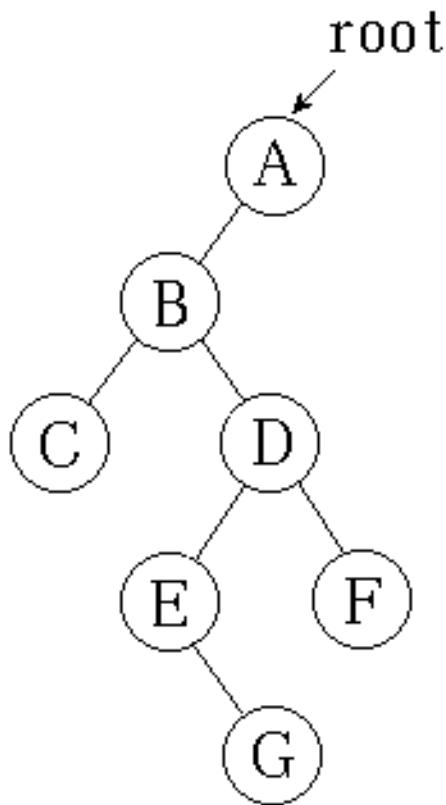
```
typedef struct Node  
{  
    DataType data;  
    struct Node *Lchild;  
    struct Node *Rchild;  
} BiTNode, *BiTree;
```



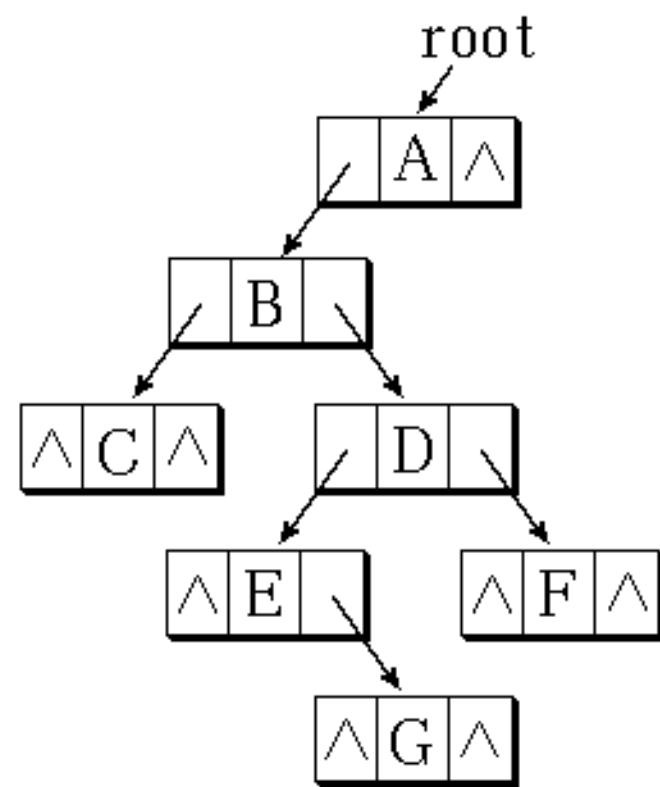
(a) 二叉树T



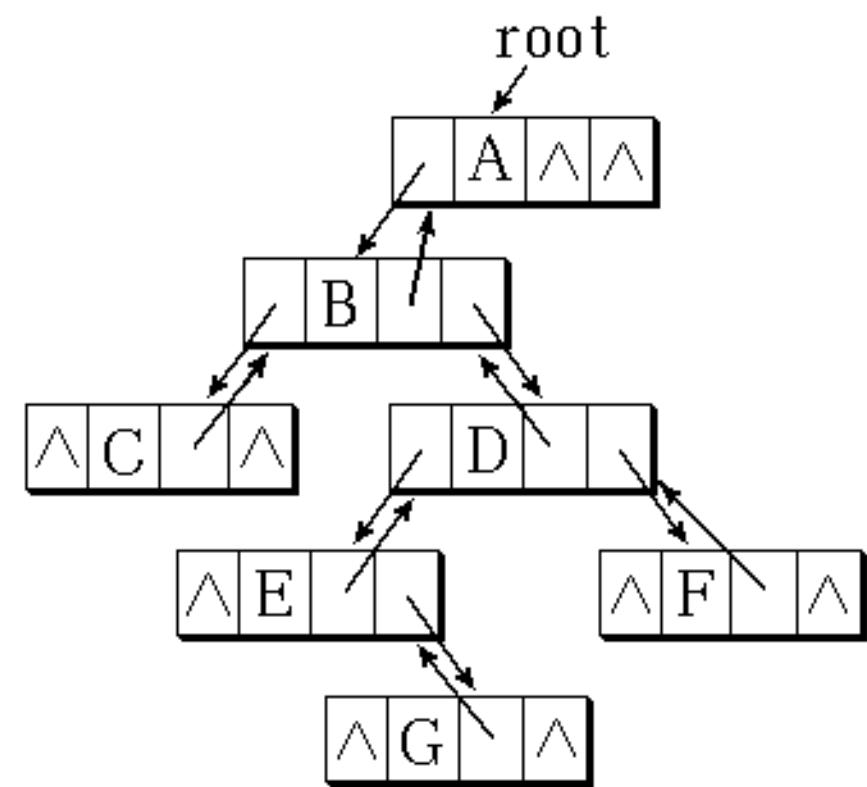
(b) 二叉树T的表达式



(a) 二叉树



(b) 二叉链表



(c) 三叉链表

静态二叉链表和静态三叉链表

data parent leftChild rightChild

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

预先开辟空间，用数组表示
leftChild, rightChild——数组元素的下标

```
// 二叉树结点类模板
template <class ElemtType>
struct BinTreeNode
{
    // 数据成员:
    ElemtType data;                                // 数据域
    BinTreeNode<ElemtType> *leftChild;             // 左孩子
    BinTreeNode<ElemtType> *rightChild;            // 右孩子
    // 构造函数模板:
    BinTreeNode();                                  // 无参构造函数模板
    BinTreeNode(const ElemtType &val,
                BinTreeNode<ElemtType> *lChild = NULL,
                BinTreeNode<ElemtType> *rChild = NULL); // 已知元素值, 指向左右孩子构造结点
};
```

```
// 二叉树类模板
template <class ElemType>
class BinaryTree
{
protected:
// 二叉树的数据成员:
BinTreeNode<ELEMType> *root;
// 辅助函数模板:
BinTreeNode<ELEMType> *CopyTreeHelp(BinTreeNode<ELEMType> *r);
// 复制二叉树
void DestroyHelp(BinTreeNode<ELEMType> * &r);
// 销毁以r为根二叉树
void PreOrderHelp(BinTreeNode<ELEMType> *r, void (*visit)(const
    ELEMType &)) const;
// 先序遍历
```

```
void InOrderHelp(BinTreeNode<ElemType> *r, void (*visit)(const  
ElemType &)) const;  
// 中序遍历  
void PostOrderHelp(BinTreeNode<ElemType> *r, void  
(*visit)(const ElemType &)) const;  
// 后序遍历  
int HeightHelp(const BinTreeNode<ElemType> *r) const;  
// 返回二叉树的高  
int NodeCountHelp(const BinTreeNode<ElemType> *r) const;  
// 返回二叉树的结点个数  
BinTreeNode<ElemType> *ParentHelp(BinTreeNode<ElemType> *r,  
        const BinTreeNode<ElemType> *cur) const;  
// 返回cur的双亲
```

```
public:  
    // 二叉树方法声明及重载编译系统默认方法声明:  
    BinaryTree();           // 无参数的构造函数模板  
    virtual ~BinaryTree(); // 析构函数模板  
    BinTreeNode<ElemType> *GetRoot() const; // 返回二叉树的根  
    bool Empty() const; // 判断二叉树是否为空  
    StatusCode GetElem(BinTreeNode<ElemType> *cur, ElemType &e) const;  
    // 用e返回结点元素值  
    StatusCode SetElem(BinTreeNode<ElemType> *cur, const ElemType &e);  
    // 将结点cur的值置为e  
    void InOrder(void (*visit)(const ElemType &)) const;  
    // 二叉树的中序遍历  
    void PreOrder(void (*visit)(const ElemType &)) const;  
    // 二叉树的先序遍历  
    void PostOrder(void (*visit)(const ElemType &)) const;  
    // 二叉树的后序遍历
```

```
void LevelOrder(void (*visit)(const ElemtType &)) const;
// 二叉树的层次遍历
int NodeCount() const; // 求二叉树的结点个数
BinTreeNode<ElemtType> *LeftChild(const BinTreeNode<ElemtType> *cur)
const;
// 返回二叉树结点cur的左孩子
BinTreeNode<ElemtType> *RightChild(const BinTreeNode<ElemtType> *cur)
const;
// 返回二叉树结点cur的右孩子
BinTreeNode<ElemtType> *Parent(const BinTreeNode<ElemtType> *cur) const;
// 返回二叉树结点cur的双亲
void InsertLeftChild(BinTreeNode<ElemtType> *cur, const ElemtType &e);
// 插入左孩子
```

```
void InsertRightChild(BinTreeNode<ElemType>*&cur, const ElemType &e);
// 插入右孩子
void DeleteLeftChild(BinTreeNode<ElemType> *cur);
// 删除左子树
void DeleteRightChild(BinTreeNode<ElemType> *cur);
// 删除右子树
int Height() const;           // 求二叉树的高
BinaryTree(const ElemType &e);
// 建立以e为根的二叉树
BinaryTree(const BinaryTree<ElemType> &copy);
// 复制构造函数模板
BinaryTree(BinTreeNode<ElemType> *r);
// 建立以r为根的二叉树
BinaryTree<ElemType> &operator=(const BinaryTree<ElemType>& copy);
// 重载赋值运算符
};
```

6.3 二叉树遍历

- “遍历”是任何类型均有的操作，对线性结构而言，只有一条搜索路径（因为每个结点均只有一个后继），故不需要另加讨论。而二叉树则存在线性结构，每个结点最多有两条路径。孩子遍历的问题。查找在全部结点逐如何遍历即按什么样的搜索常对树中历二叉树的一些应用中，或者在二叉树的一些应用中，或者引入了遍历二叉树的问题。
- 在具有某种特征的结点，或进行某种处理。这就引入了遍历二叉树的问题。

6.3.1 遍历的定义

- 遍历二叉树：遵从某种次序，顺着某一条搜索路径访问二叉树中的各个结点，使得每一个结点均被访问一次，而且仅被访问一次。
- 访问的含义可以很广，如：输出结点的信息等

6.3.1 遍历的定义

- 遍历对于线性结构是容易解决的，而二叉树是非线性结构，因而需要寻找一种规律，以便使二叉树上的结点能排列在一个线性队列上从而便于遍历。

6.3.1 遍历的定义

对“二叉树”而言，可以有三类搜索路径

- 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

6.3.1 遍历的定义

我们用L、D、R分别表示遍历左子树、访问根结点、遍历右子树，那么对二叉树的遍历顺序就可以有以下六种方式以及层次遍历。

- (1) 访问根，遍历左子树，遍历右子树（记做DLR，前序）。
- (2) 访问根，遍历右子树，遍历左子树（记做DRL）。
- (3) 遍历左子树，访问根，遍历右子树（记做LDR，中序）。
- (4) 遍历左子树，遍历右子树，访问根（记做LRD，后序）。
- (5) 遍历右子树，访问根，遍历左子树（记做RDL）。
- (6) 遍历右子树，遍历左子树，访问根（记做RLD）。

6.3.1 遍历的定义

- 如果规定先左后右，则只剩下4种遍历方式：DLR，LDR，LRD以及层次遍历。
- 根据根结点被遍历的次序，通常称DLR，LDR，LRD这3种方式为前序遍历，中序遍历和后序遍历。

6.3.1 遍历的定义

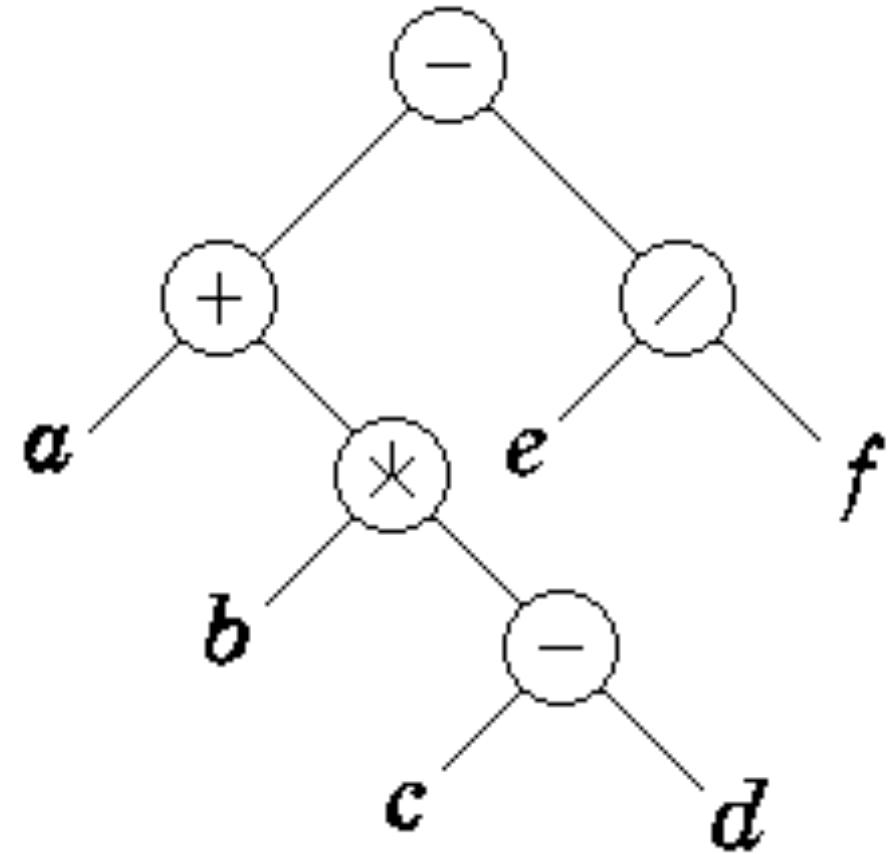
- 前序遍历也称为先序遍历。
- 先序、中序、后序遍历是递归定义的，即在其子树中亦按上述规律进行遍历。

6.3.1 遍历的定义

先序遍历 (DLR) 操作过程

若二叉树为空，则空操作，
否则依次执行如下3个操作：

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



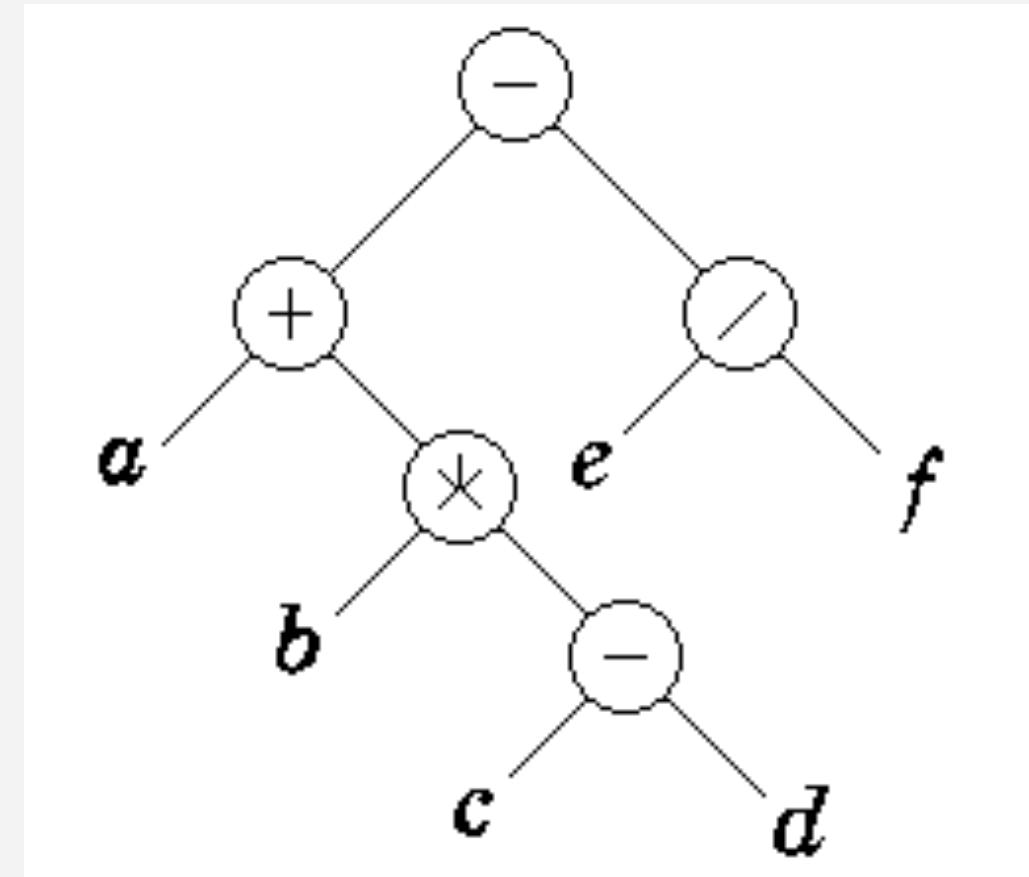
- + a * b - c d / e f

```
template <class ElemtType>
void BinaryTree<ElemtType>::PreOrderHelp(BinTreeNode<ElemtType> *r,
    void (*visit)(const ElemtType &)) const
{
    if (r != NULL) {
        (*visit)(r->data); // 访问根结点
        PreOrderHelp(r->leftChild, visit); // 遍历左子树
        PreOrderHelp(r->rightChild, visit); // 遍历右子树
    }
}
template <class ElemtType>
void BinaryTree<ElemtType>::PreOrder(void (*visit)(ElemtType &))
// 操作结果：先序遍历二叉树
{
    PreOrderHelp(root, visit);
}
```

6.3.1 遍历的定义

中序遍历 (LDR) 操作过程：
若二叉树为空，则空操作，
否则依次执行如下3个操作：

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

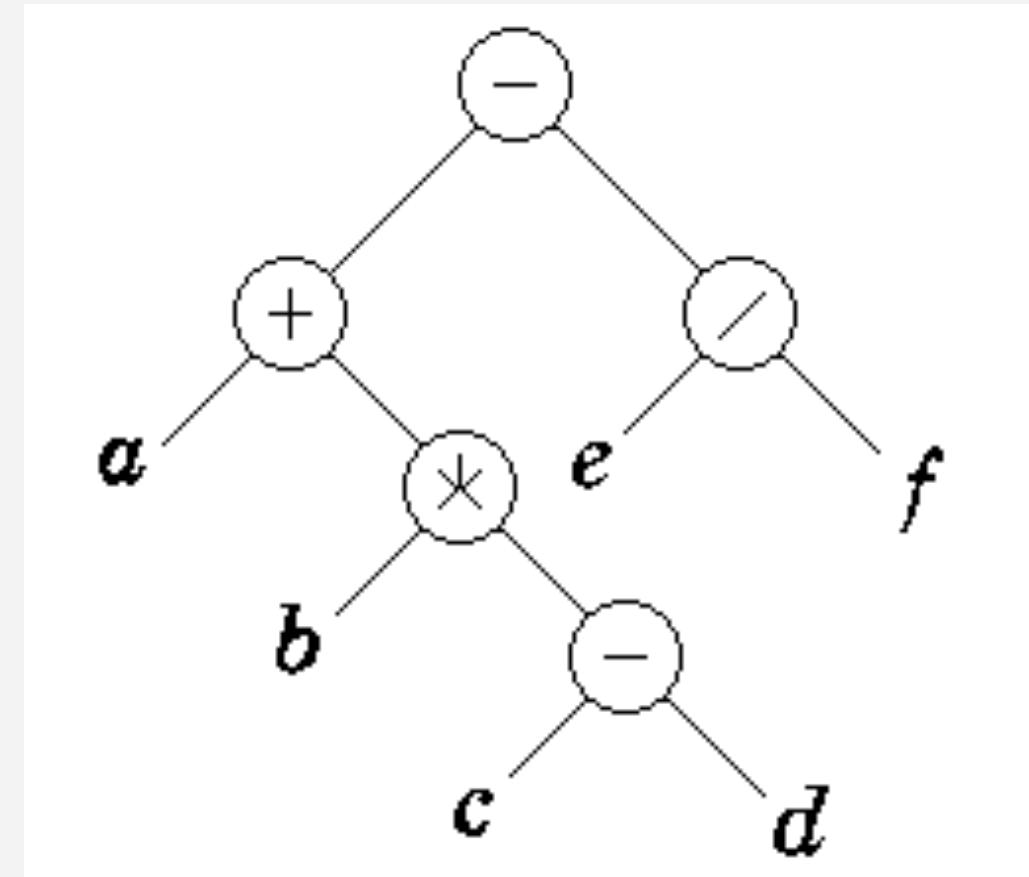


$$a + b * c - d - e / f$$

6.3.1 遍历的定义

后序遍历 (LRD) 操作过程：
若二叉树为空，则空操作，
否则依次执行如下3个操作：

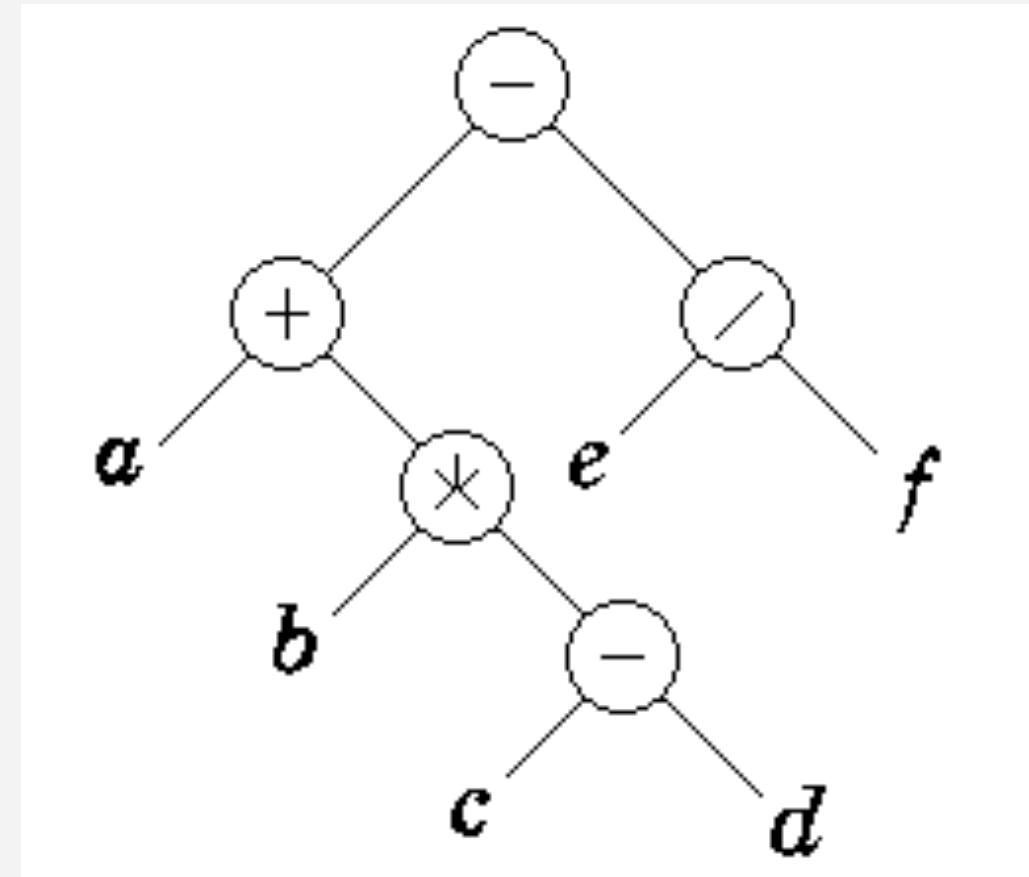
- (1) 按后序遍历左子树；
- (2) 按后序遍历右子树；
- (3) 访问根结点。



a b c d - * + e f / -

6.3.1 遍历的定义

层次遍历
从上到下，从左到右



- + / a * e f b - c d

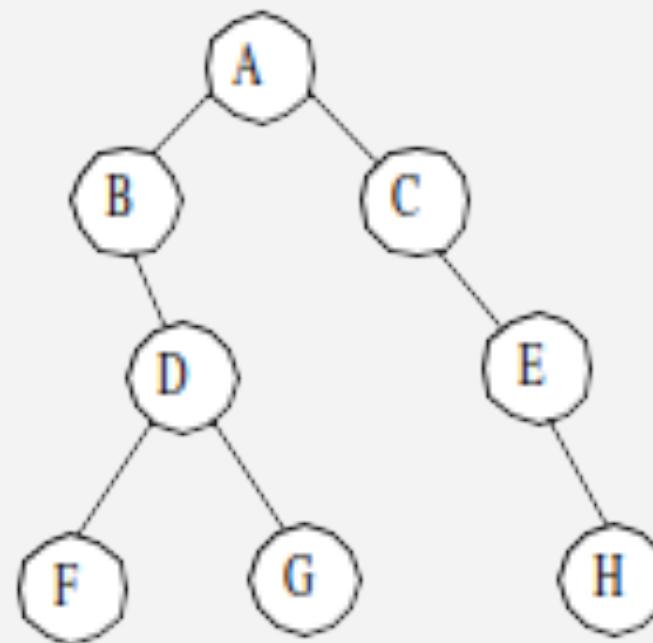
练习

对于如图所示的二叉树，其先序、中序、后序遍历的序列如下：

先序遍历： A、 B、 D、 F、 G、 C、 E、 H 。

中序遍历： B、 F、 D、 G、 A、 C、 E、 H 。

后序遍历： F、 G、 D、 B、 H、 E、 C、 A 。



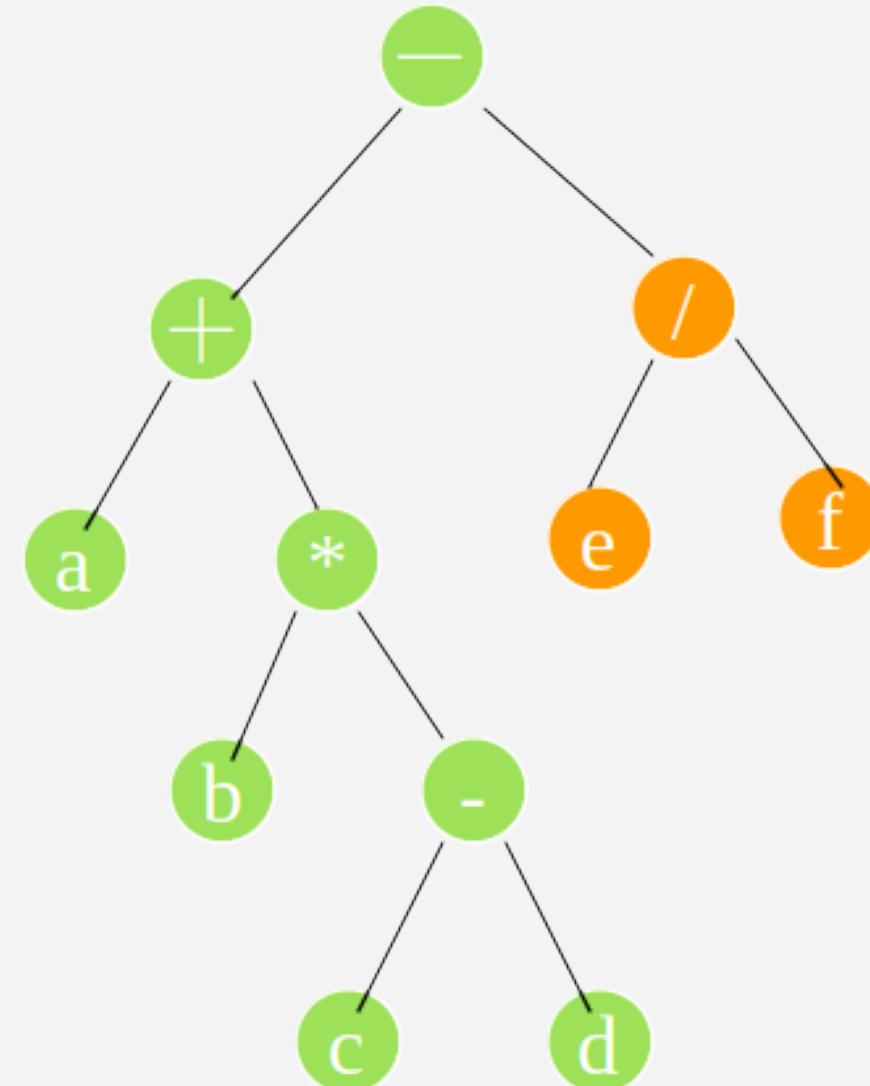
练习

如图所示的二叉树表达式
 $(a+b*(c-d)-e/f)$

按先序遍历，其先序序列为：
-+a*b-cd/ef

按中序遍历，其中序序列为：
a+b*c-d-e/f

按后序遍历，其后序序列为：
abcd-*+ef/-



6.3.2 遍历算法

- **递归算法**

二叉树的先序、中序和后序遍历采用递归方式进行定义，算法最简单直接的方式就是递归方式。

- **非递归算法**

递归算法的效率一般比非递归算法的效率低，二叉树的非递归算法更复杂。可以用数据结构栈来实现。

6.3.2 遍历算法

层次遍历是先访问层次小的结点，同一层次从左到右，可以用数据结构队列来实现。

6.3.3 二叉树遍历应用举例

统计二叉树中结点的个数

二叉树的结点个数等于左子树的结点数加上右子树的结点数再加上根结点数1，因此求二叉树的结点数的问题可以分解为计算其左右子树的结点数目问题。

```
template <class ElemType>
int BinaryTree<ELEMType>::NodeCountHelp(
    const BinTreeNode<ELEMType> *r) const
{
    if (r == NULL) return 0; // 空二叉数结点个数为0
    else
        return NodeCountHelp(r->leftChild) +
            NodeCountHelp(r->rightChild) + 1;
        // 结点个数为左右子树的结点数和再加1
}
template <class ElemType>
int BinaryTree<ELEMType>::NodeCount() const
{
    // 操作结果：返回二叉树的结点个数
    return NodeCountHelp(root);
}
```

6.3.3 二叉树遍历应用举例

求二叉树的高

- 从二叉树高的定义可知，二叉树的高应为其左、右子树高的最大值加1。
- 由此，需先分别求得左、右子树的高，算法中“访问结点”的操作为：求得左、右子树高的最大值，然后加1。

```
template <class ElemtType>
int BinaryTree<ElemtType>::HeightHelp(const BinTreeNode<ElemtType> *r) const
{
    // 操作结果：返回以r为根的二叉树的高
    if (r == NULL) {
        return 0; // 空二叉树高为0
    }
    else { // 非空二叉树高为左右子树高的最大值再加1
        int lHeight, rHeight;
        lHeight = HeightHelp(r->leftChild); // 左子树高
        rHeight=HeightHelp(r->rightChild); // 右子树高
        return (lHeight > rHeight ? lHeight : rHeight) + 1;
        // 高为左右子树高的最大值加1
    }
}
template <class ElemtType>
int BinaryTree<ElemtType>::Height() const
{
    // 操作结果：返回二叉树的高
    return HeightHelp(root);
}
```

6.3.3 二叉树遍历应用举例

由二叉树的先序和中序序列建立二叉树

- 仅知二叉树的先序序列“abcdefg”不能唯一确定一棵二叉树
- 已知二叉树的先序序列和中序序列，可以确定一颗二叉树
- 已知后序序列和中序序列，可以确定一颗二叉树

6.3.3 二叉树遍历应用举例

二叉树的先序序列

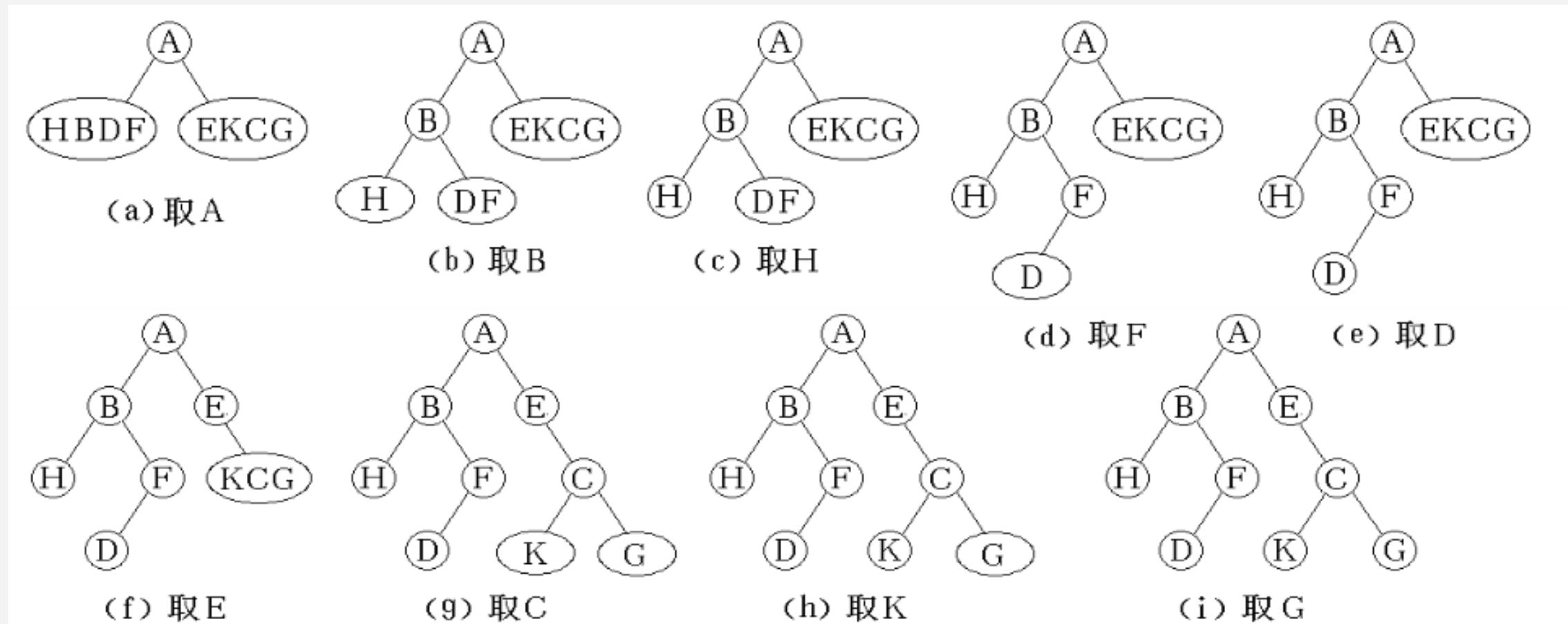


二叉树的中序序列



6.3.3 二叉树遍历应用举例

先序序列 { ABHFDECKG } 和中序序列 { HBDFAEKCG }, 构造二叉树过程如下:



练习

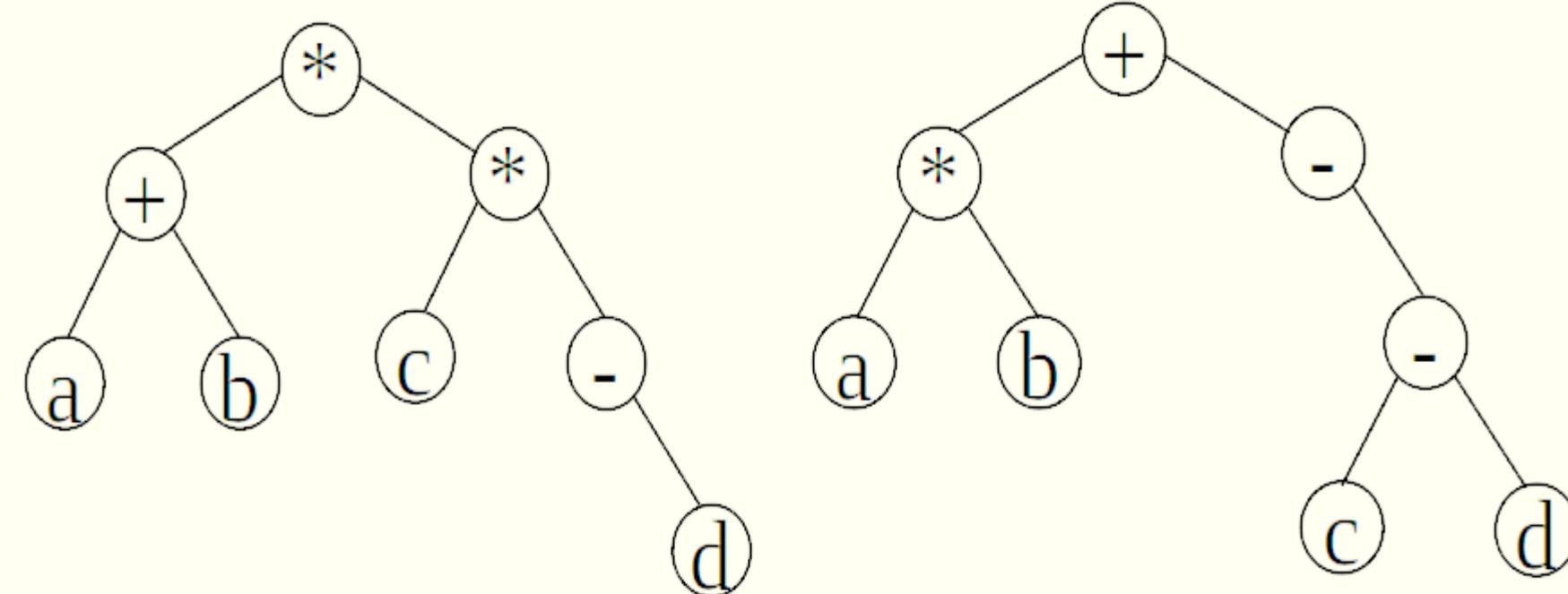
若一棵二叉树的前序遍历序列为 a, e, b, d, c， 后序遍历序列为 b, c, d, e, a，则根结点的孩子结点

- A. 只有 e
- B. 有 e、 b
- C. 有 e、 c
- D. 无法确定

参考答案： A

练习

画出中缀表达式对应的二叉树 $(a+b)*\left(c*(-d)\right)$ 和 $\left(a*b\right)+\left(-\left(c-d\right)\right)$



解答：操作数作为叶结点；二叉树无括号；中序遍历加上括号就是中缀表达式也就是算术表达式一般形式；后序遍历就是后缀表达式。

6.4 线索二叉树

- 二叉树的遍历运算是将二叉树中结点按一定规律线性化的过程。
- 当以二叉链表作为存储结构时，只能找到结点的左、右孩子信息，而不能直接得到结点在遍历序列中的前驱和后继信息。

6.4 线索二叉树

要得到这些信息可采用以下两种方法：

- 第一种方法是将二叉树遍历一遍，在遍历过程中便可得到结点的前驱和后继，但这种动态访问浪费时间；
- 第二种方法是充分利用二叉链表中的空链域，将遍历过程中结点的前驱、后继信息保存下来。

6.4 线索二叉树

- 在有 n 个结点的二叉链表中共有 $2n$ 个链域，但只有 $n-1$ 个有用的非空链域，其余 $n+1$ 个链域是空的。
- 我们可以利用剩下的 $n+1$ 个空链域来存放遍历过程中结点的前驱和后继信息。

| | | | | |
|------------------|----------------|-------------|-----------------|-------------------|
| leftChild | leftTag | data | rightTag | rightChild |
|------------------|----------------|-------------|-----------------|-------------------|

6.4 线索二叉树

- 作如下规定：若结点有左子树，则其LeftChild域指向其左孩子，否则LeftChild域指向其前驱结点；若结点有右子树，则其RightChild域指向其右孩子，否则RightChild域指向其后继结点。
- 为了区分孩子结点和前驱、后继结点，为结点结构增设两个标志域。

6.4 线索二叉树

- 指向线性序列中前驱和后继结点的指针叫做**线索**。
- 包含线索的存储结构，叫做**线索链表**。
- 对**二叉树**以某种次序进行遍历并且加上线索的过程叫做**线索化**。
- 加上了线索的二叉树称为**线索二叉树**。

6.4 线索二叉树

在二叉链表的结点中增加两个标志域leftTag和rightTag，并作如下规定

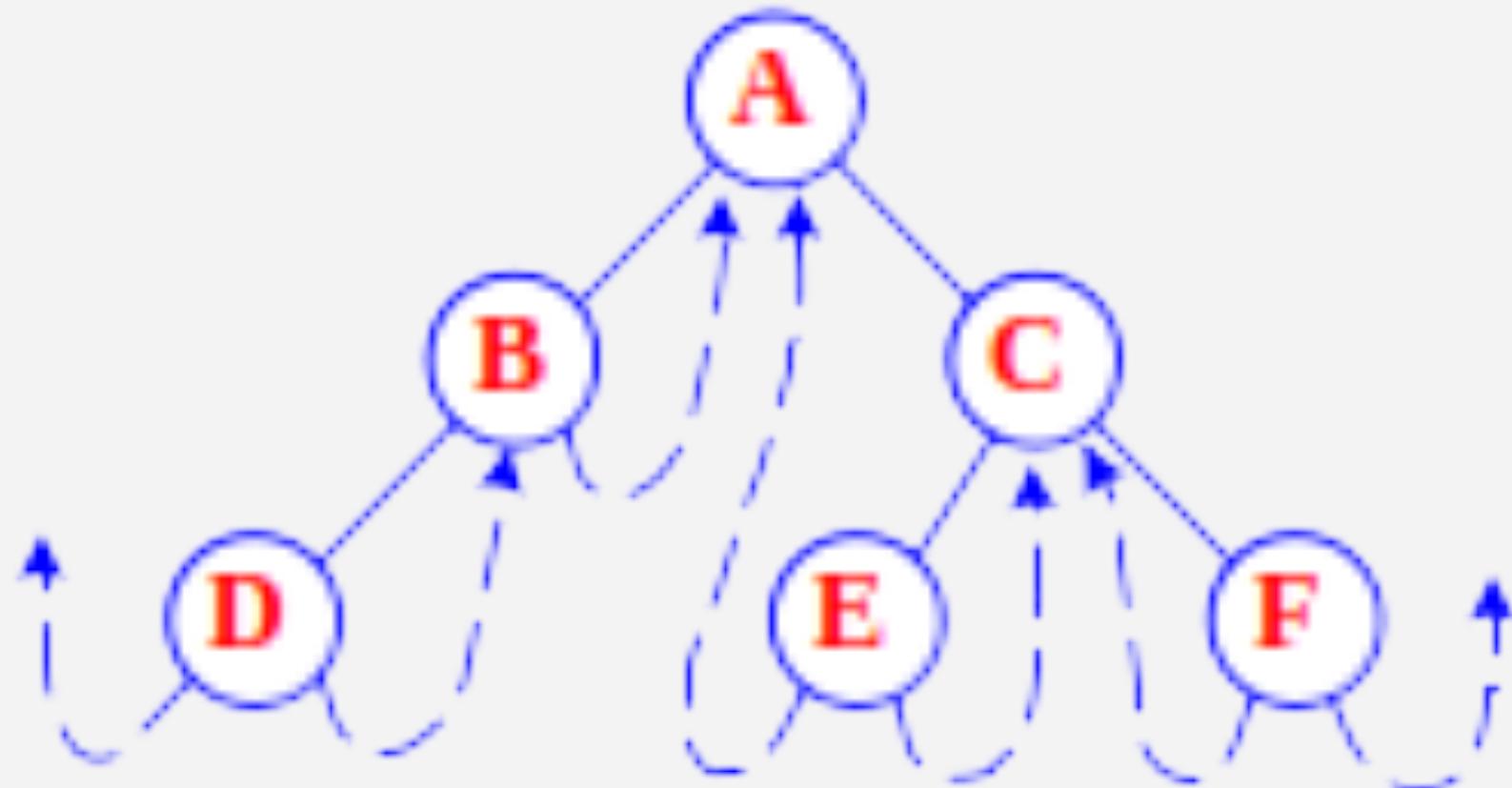
- 若该结点的左子树不空，leftChild域的指针指向其左孩子，且左标志leftTag的值为0
- 否则，leftChild域的指针指向其“前驱”，且左标志leftTag的值为1

6.4 线索二叉树

在二叉链表的结点中增加两个标志域leftTag和rightTag，并作如下规定

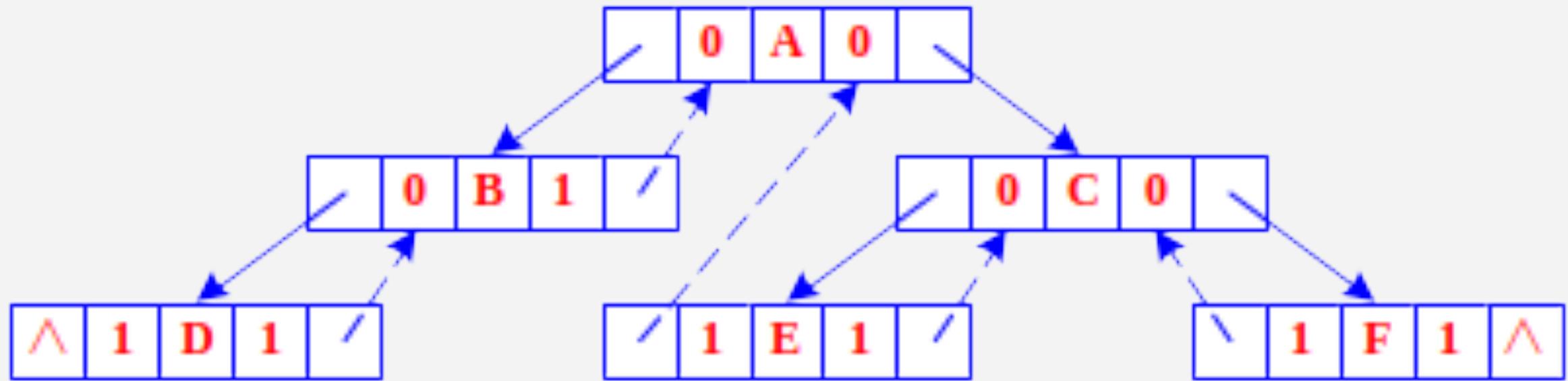
- 若该结点的右子树不空则rightChild域的指针指向其右孩子，且右标志rightTag的值为0
- 否则，rightChild域的指针指向其“后继”，且右标志rightTag的值为1

中序线索二叉树



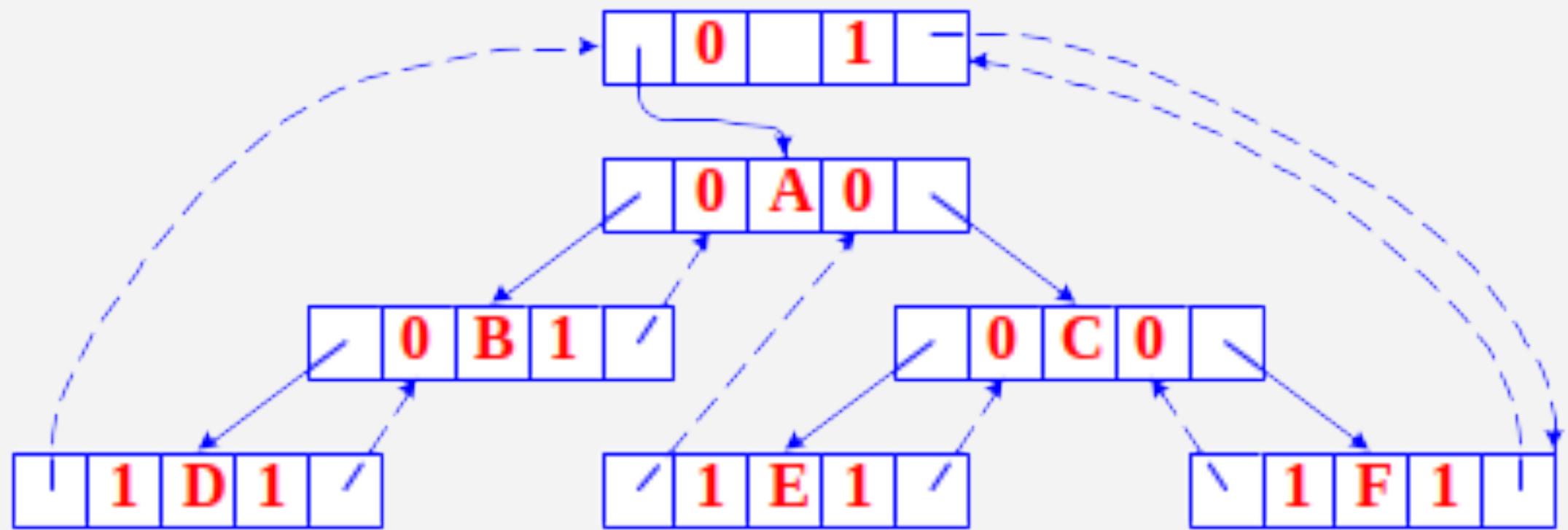
中序序列： D B A E C F

不带表头结点的中序线索二叉树示意图



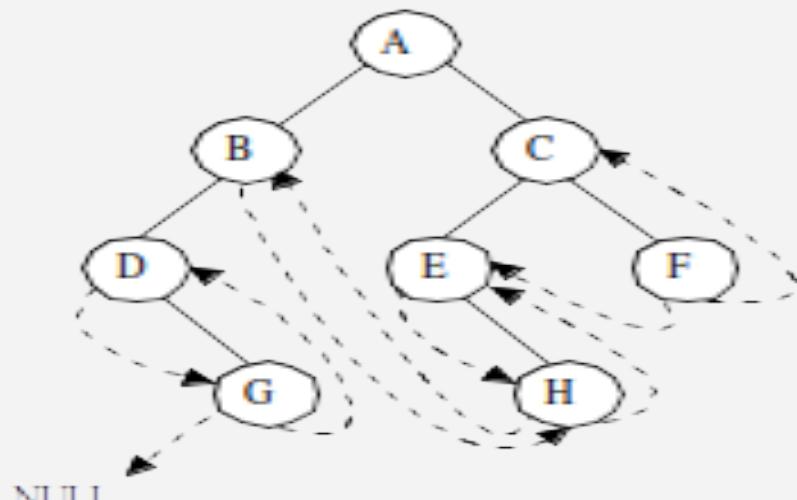
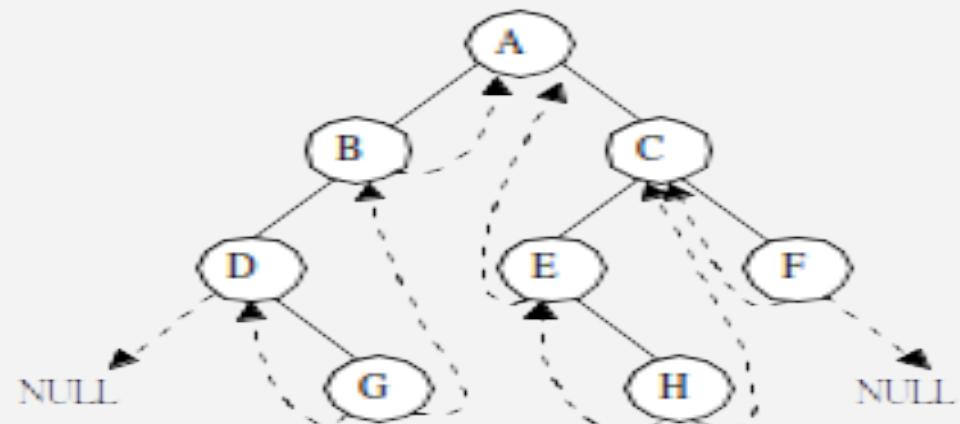
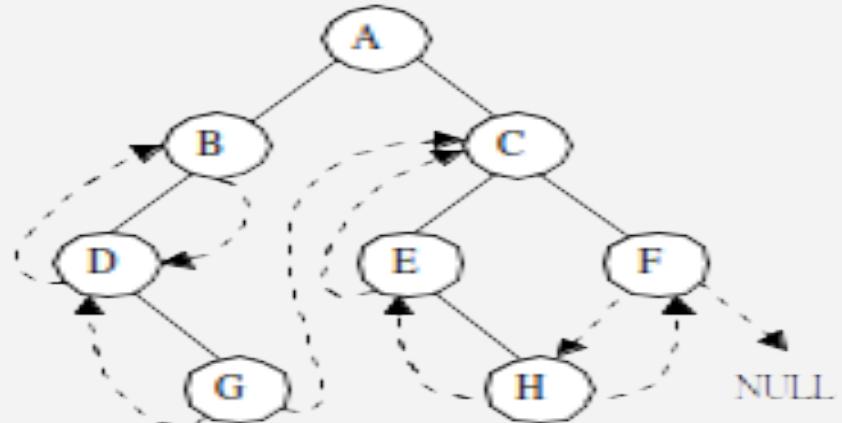
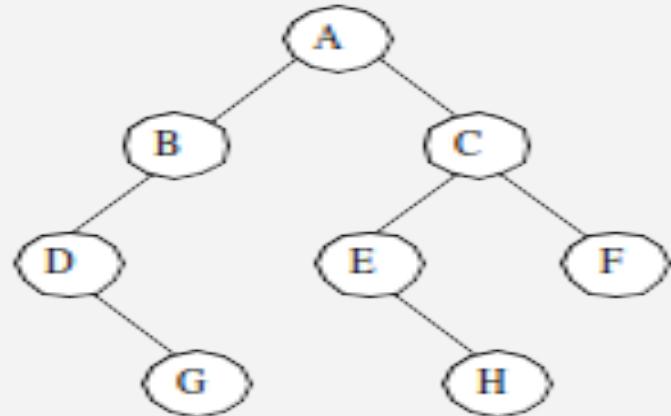
中序序列： D B A E C F

带表头结点的中序线索二叉树示意图



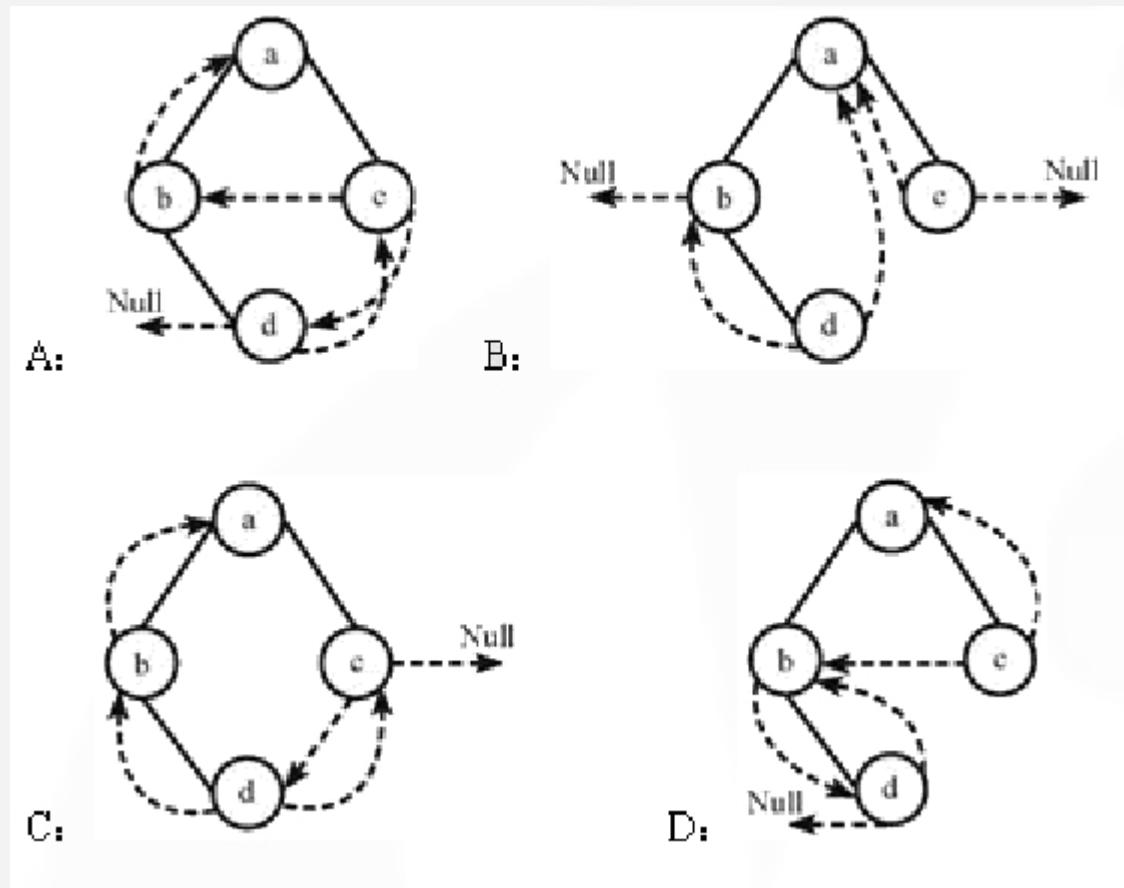
中序序列： D B A E C F

6.4 线索二叉树



练习

下列线索二叉树中（用虚线表示线索），符合后序线索树定义的是（ D ）



6.5 树和森林

树和森林的存储表示和遍历操作
树和森林与二叉树的对应关系

6.5.1 树的存储表示

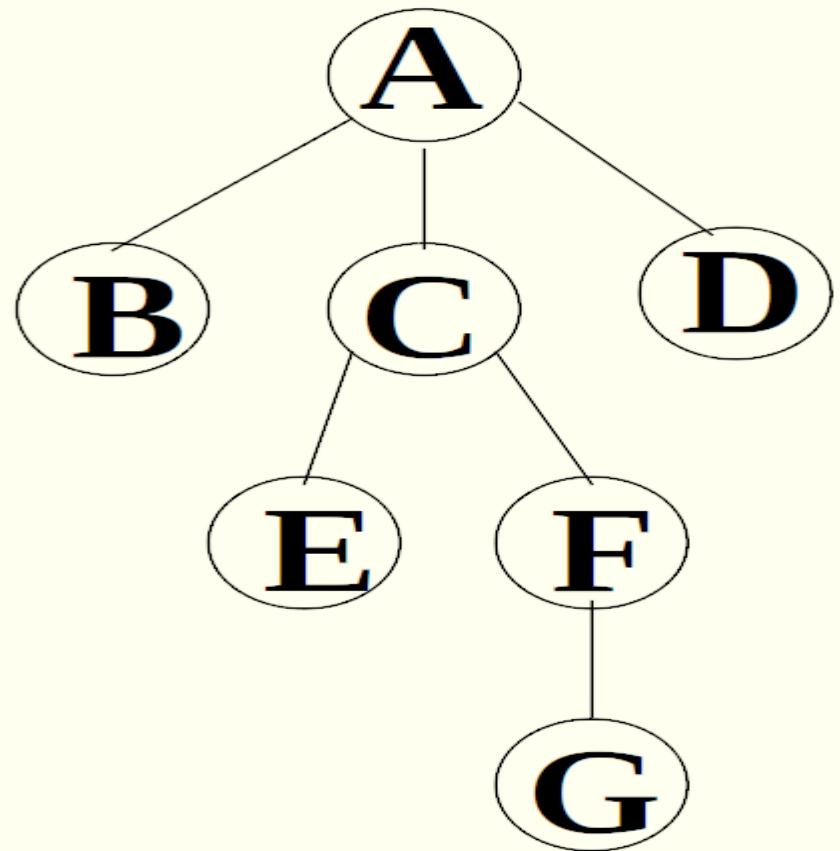
树的主要存储方法有以下三种

- 双亲表示法
- 孩子双亲表示法
- 孩子兄弟表示法

6.5.1 树的存储表示

双亲表示法：这种方法用一组连续的空间来存储树中的结点，在保存每个结点的同时附设一个指示器来指示其双亲结点在表中的位置，其结点的结构如下。

| | |
|------|--------|
| Data | Parent |
|------|--------|



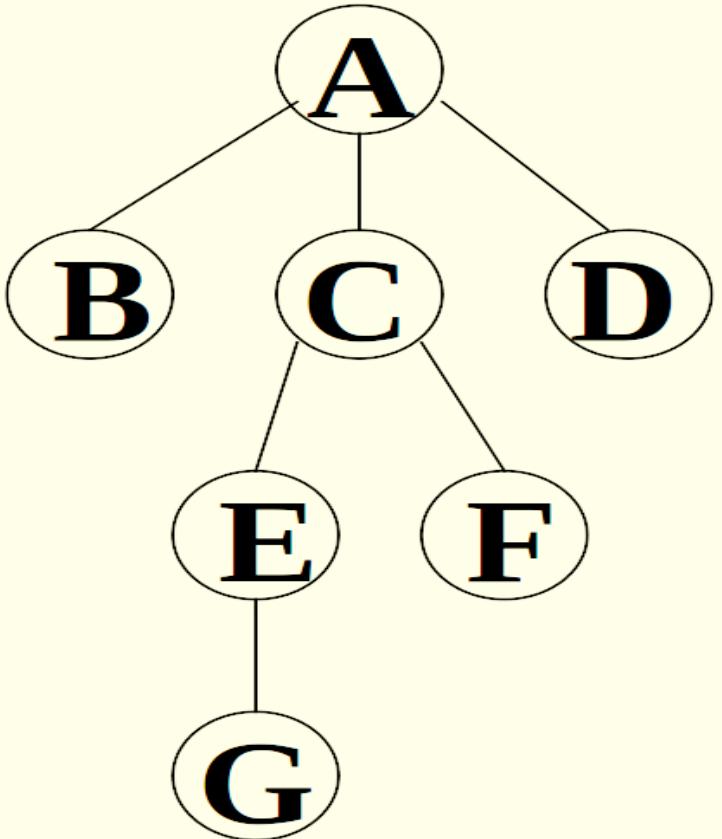
| data | parent |
|------|--------|
| A | -1 |
| B | 0 |
| C | 0 |
| D | 0 |
| E | 2 |
| F | 2 |
| G | 5 |

6.5.1 树的存储表示

- 双亲表示法这种存储结构利用了每个结点（除根以外）只有唯一的双亲的性质。 $PARENT(T, x)$ 操作可以在常量时间内实现。反复调用 $PARENT$ 操作，直到遇见无双亲的结点时，便找到了树的根。
- 双亲表示法求结点的孩子时需要遍历整个结构。

6.5.1 树的存储表示

- 孩子双亲表示法：把每个结点的孩子结点排列起来，看成是一个线性表，而且以单链表作存储结构，则n个结点有n个孩子链表（叶子的孩子链表为空表）。而n个头指针又组成一个线性表，存储在一个数组中，并在数组中同时存储数据元素的值以及双亲位置。
- 不但适合查找某个结点的孩子，也适合于查找双亲结点。



data firstchild

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

6.5.1 树的存储表示

- 一般的树中，每个结点的孩子数目是不同的。如果根据度为每个结点分配相同的指针域，会带来较大的浪费；如果采用变长结构，也会对存储和操作带来麻烦。
- 每个结点的首孩子和右兄弟是唯一的，因此可以采用二叉链表来表示——孩子兄弟表示法。

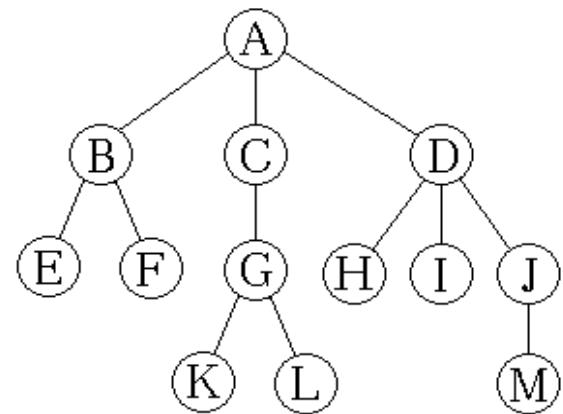
6.5.1 树的存储表示

- 孩子兄弟表示法：又称二叉树表示法，或二叉链表表示法。即以二叉链表作树的存储结构。链表中结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点。
- 利用这种存储结构便于实现各种树的操作。特别是可以实现森林与二叉树的转换。

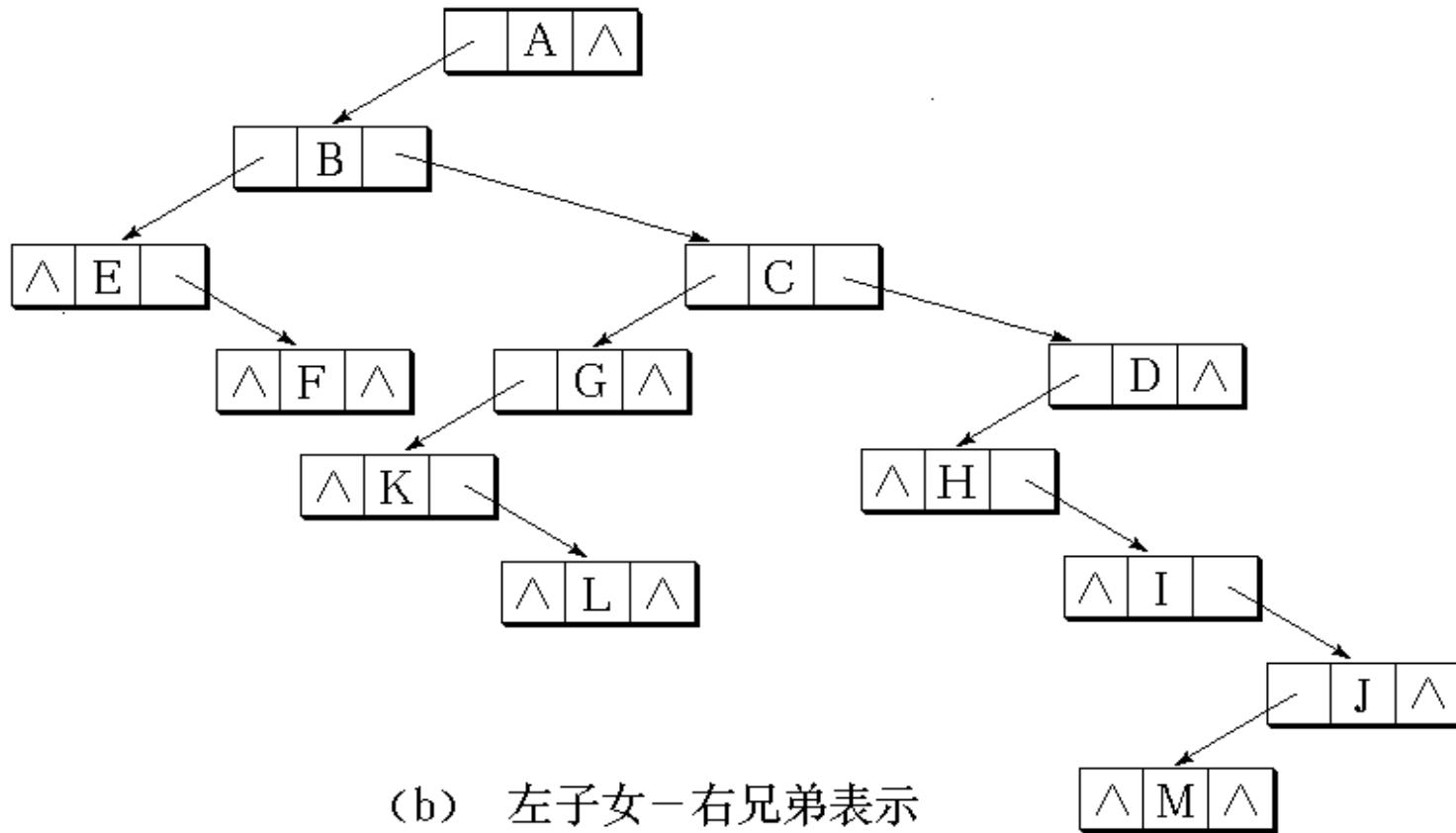
firstChild

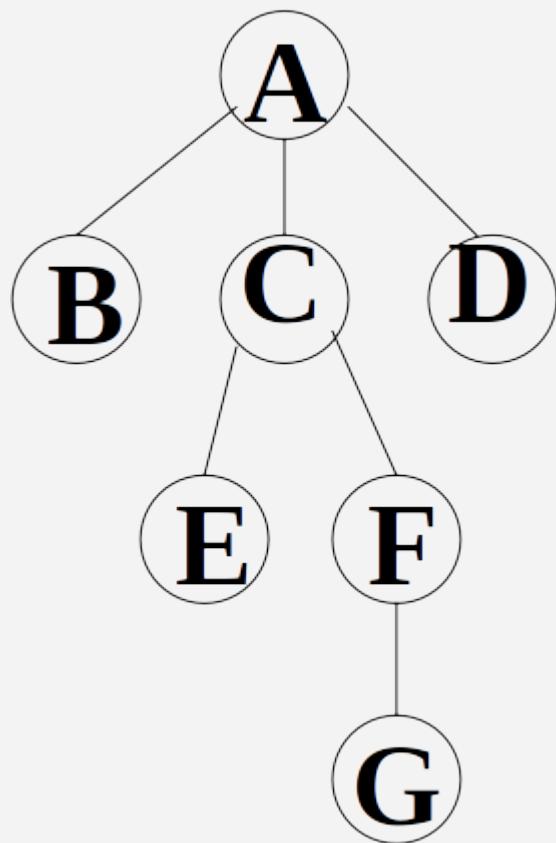
data

nextSibling

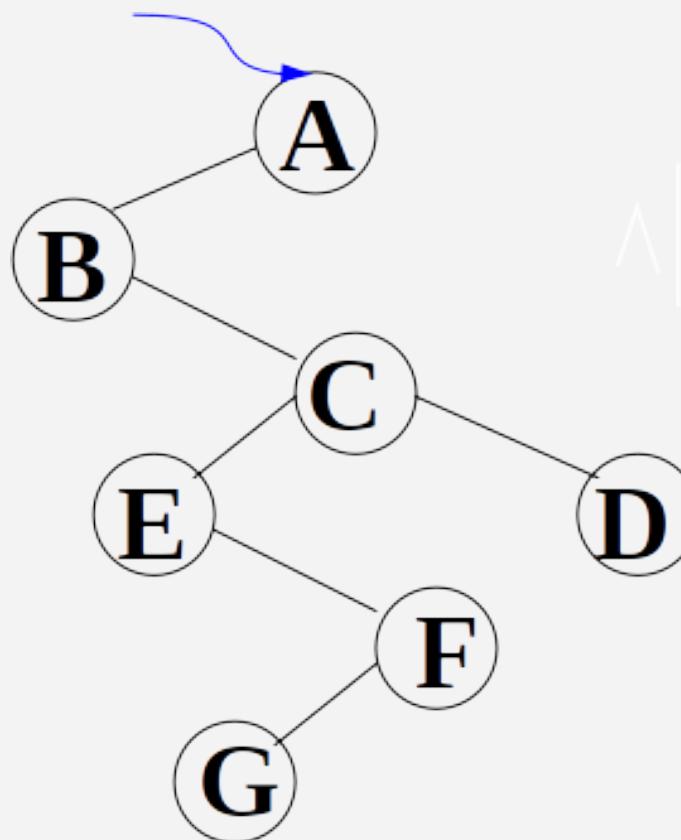


(b) 左子女-右兄弟表示

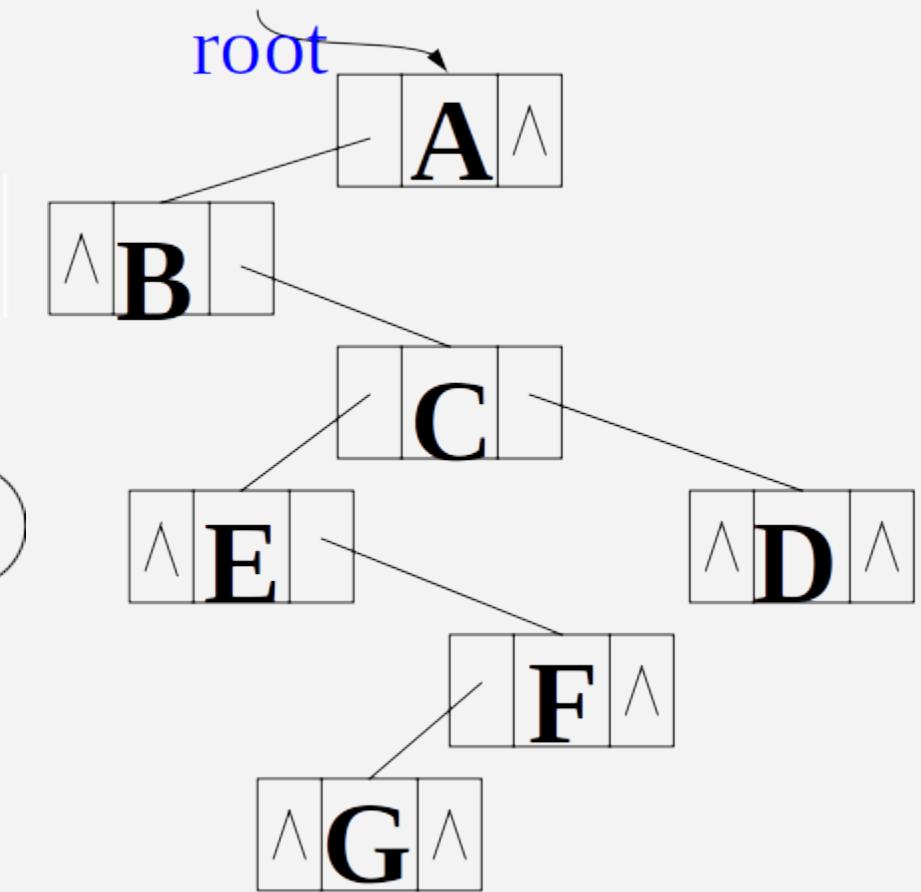




一般的树



二叉树



一般树的孩子兄弟表示
或二叉树的二叉链表表示

6.5.1 树的存储表示

- 对于孩子兄弟表示法，树的各种操作均可对应二叉树的操作来完成。应当注意的是，和树对应的二叉树，其左、右孩子的概念已改变为：左是树中第一个孩子，右是树中下一个兄弟。
- 找某一结点的所有孩子，需要先通过第一个孩子指针找到第一个孩子，然后根据兄弟指针一直找到右指针为空即可。

6.5.3 森林的存储表示

可以采用与树相同的存储方式来表示森林，森林通常也有3种表示方式。通常采用孩子兄弟表示法作为存储结构。

- 双亲表示法
- 孩子双亲表示法
- 孩子兄弟表示法

6.5.3 森林的存储表示

树的遍历方法主要有以下两种

- 先根遍历，若树非空，则遍历方法为：

✓ 访问根结点。

✓ 从左到右，依次先根遍历根结点的每一棵子树。

- 后根遍历，若树非空，则遍历方法为：

✓ 从左到右，依次后根遍历根结点的每一棵子树。

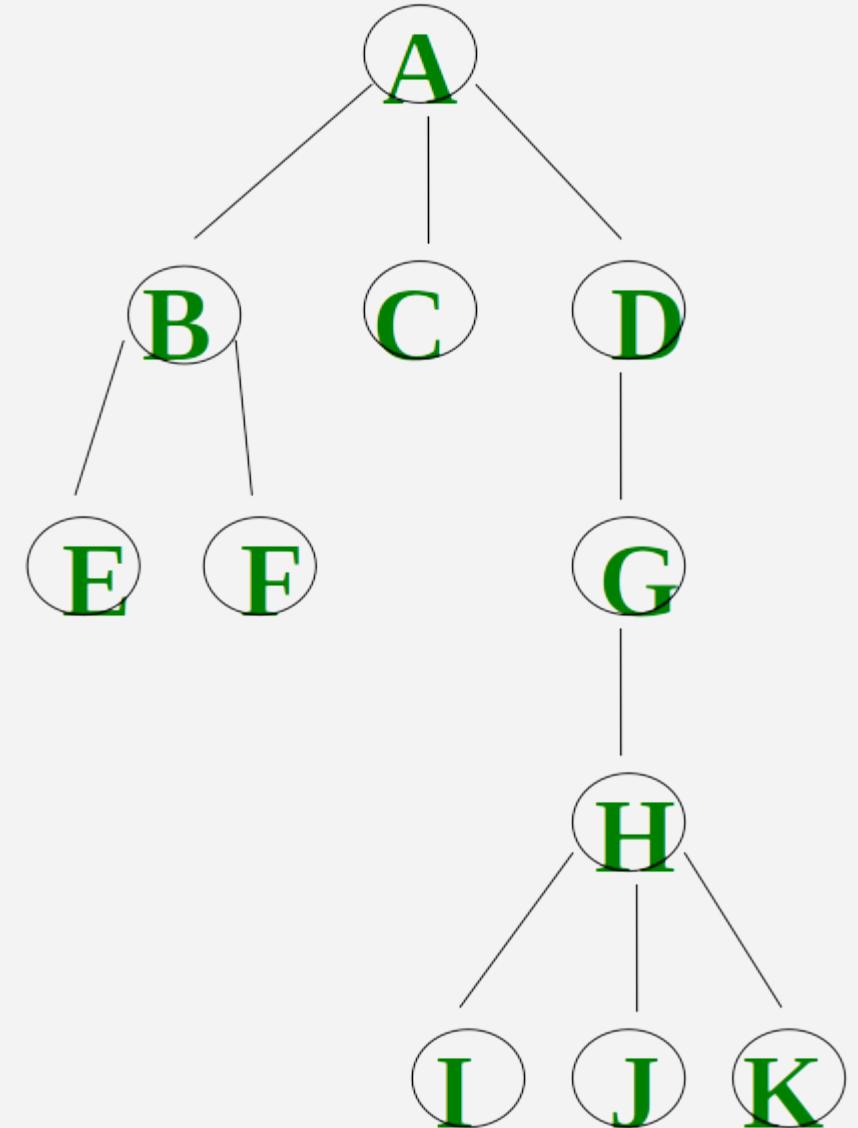
✓ 访问根结点。

先根次序遍历

A B E F C D G H I J K

后根次序遍历

E F B C I J K H G D A



6.5.4 树和森林的遍历

森林的遍历方法主要有以下两种

•先序遍历，若森林非空，则遍历方法为：

1. 访问森林中第一棵树的根结点。
2. 先序遍历第一棵树的根结点的子树森林。
3. 先序遍历除去第一棵树之后剩余的树构成的森林。

•中序遍历，若森林非空，则遍历方法为：

1. 中序遍历森林中第一棵树的根结点的子树森林。
2. 访问第一棵树的根结点。
3. 中序遍历除去第一棵树之后剩余的树构成的森林。

6.5.5 树和森林与二叉树的转换

1. 由于二叉树和树都可以用二叉链表作为存储结构，则以二叉链表作为媒介可以导出树与二叉树之间的一个对应关系。
2. 也就是说，给定一棵树，可以找到唯一的一棵二叉树与之对应。

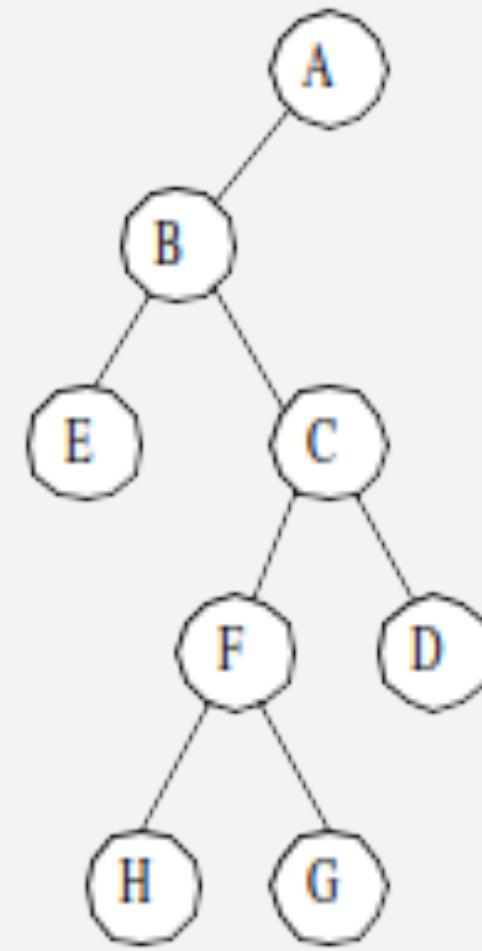
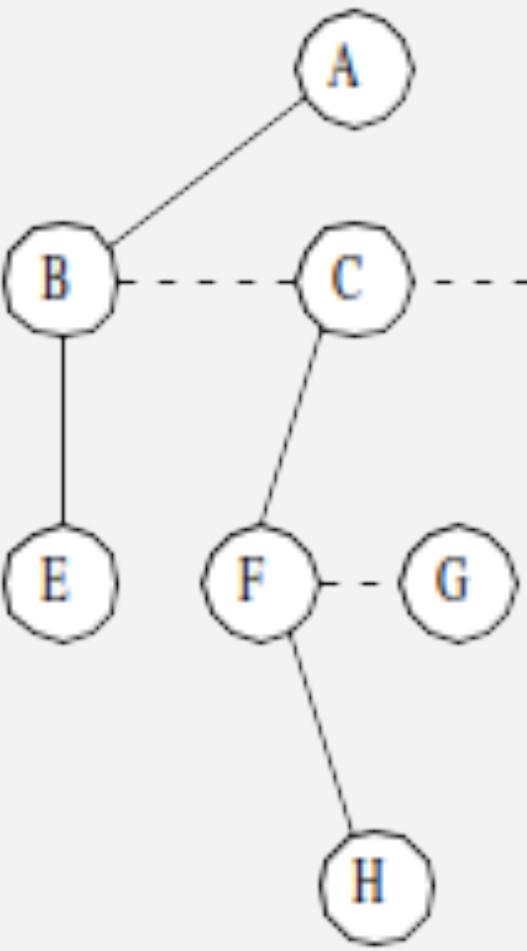
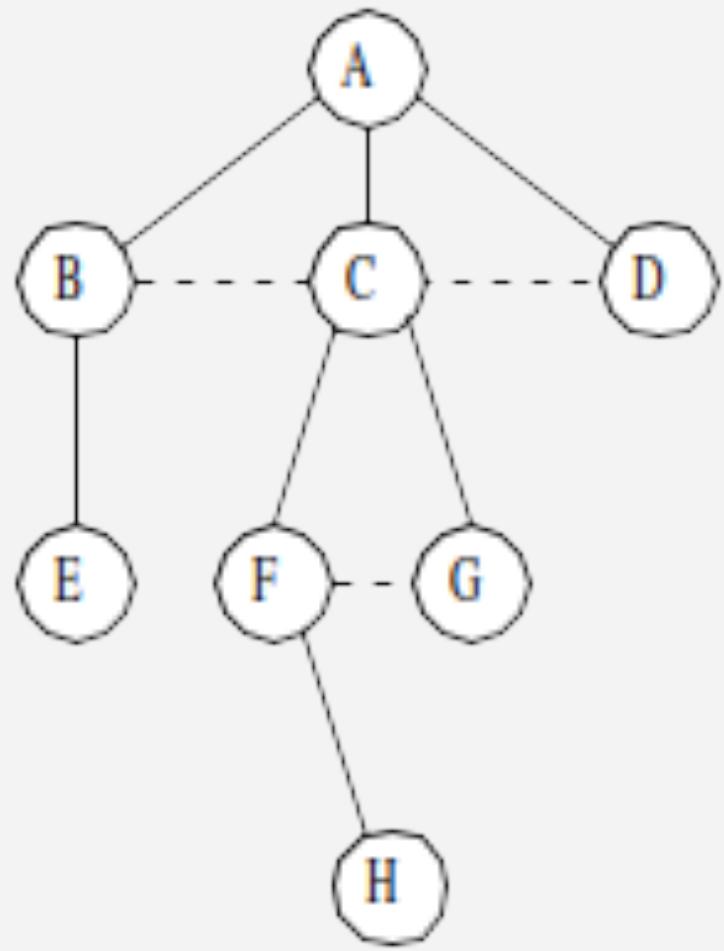
6.5.5 树和森林与二叉树的转换

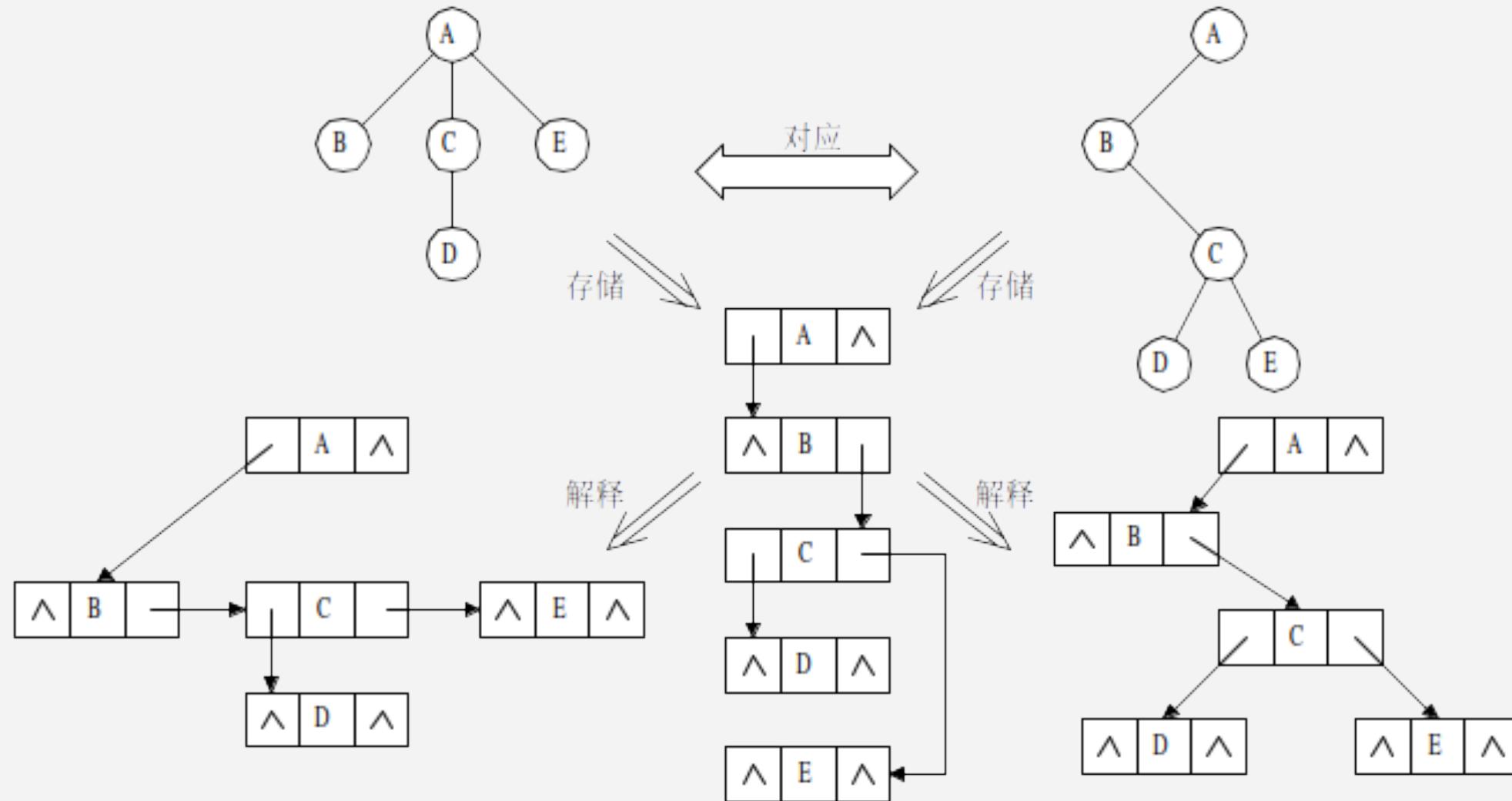
1. 给定一棵树，有唯一的一棵二叉树与之对应。从物理结构上看，它们的二叉链表是相同的，只是解释不同。
2. 任何一棵和树对应的二叉树，其右子树必为空。

6.5.5 树和森林与二叉树的转换

将一棵树转换为二叉树的方法是用孩子兄弟存储方式来表示。

- (1) 树中所有相邻兄弟之间加一条连线。
- (2) 对树中的每个结点，只保留其与第一个孩子结点之间的连线，删去其与其它孩子结点之间的连线。
- (3) 以树的根结点为轴心，将整棵树顺时针旋转一定的角度，使之结构层次分明。





树与二叉树的对应

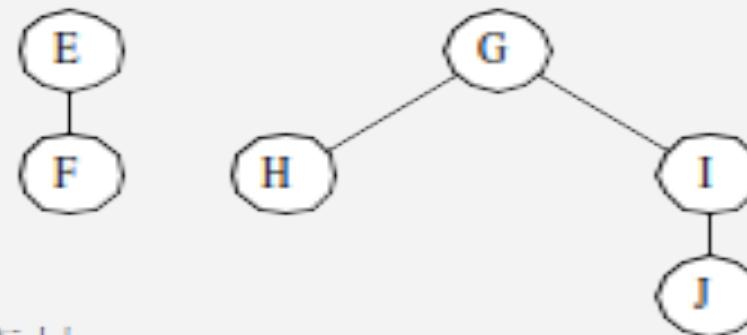
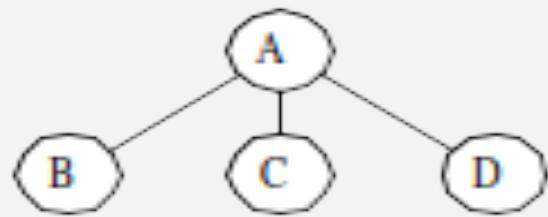
6.5.5 树和森林与二叉树的转换

1. 森林转换为二叉树：森林是若干棵树的集合。树可以转换为二叉树，森林同样也可以转换为二叉树。
2. 因此，森林也可以方便地用孩子兄弟链表表示。

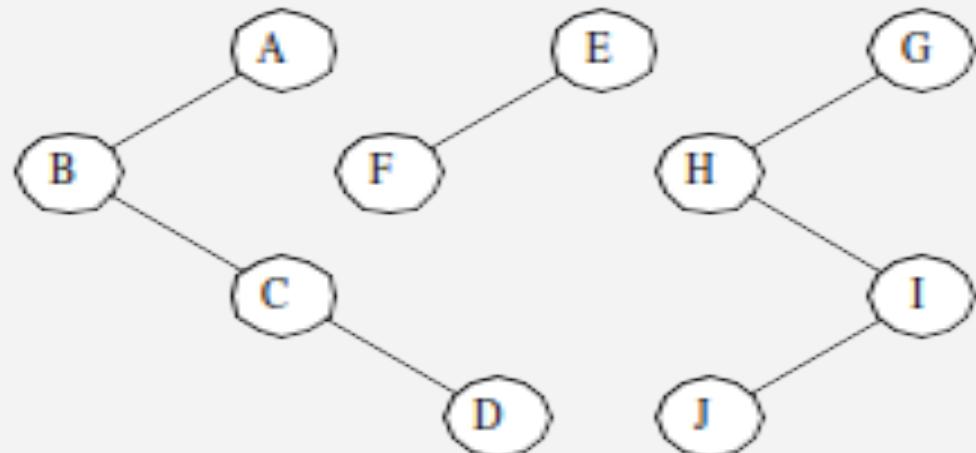
6.5.5 树和森林与二叉树的转换

森林转换为二叉树的方法如下：

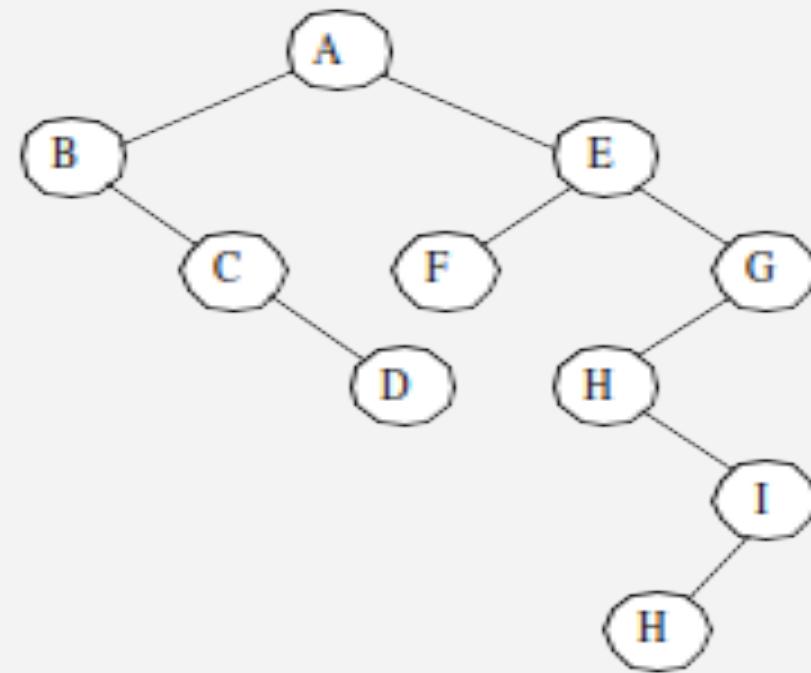
- (1) 将森林中的每棵树转换成相应的二叉树。
- (2) 第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，当所有二叉树连在一起后，所得到的二叉树就是由森林转换得到的二叉树。



(a) 森林



(b) 森林中每棵树对应的二叉树



(c) 森林对应的二叉树

6.5.5 树和森林与二叉树的转换

将森林F看作树的有序集 $F=\{T_1, T_2, \dots, T_N\}$ ，它对应的二叉树为 $B(F)$

(1) 若 $N=0$ ，则 $B(F)$ 为空。

(2) 若 $N>0$ ，二叉树 $B(F)$ 的根为森林中第一棵树 T_1 的根；
 $B(F)$ 的左子树为 $B(\{T_{11}, \dots, T_{1m}\})$ ，其中 $\{T_{11}, \dots, T_{1m}\}$ 是 T_1 的子树森林； $B(F)$ 的右子树是 $B(\{T_2, \dots, T_N\})$ 。

根据这个递归的定义，我们可以很容易地写出递归的转换算法。

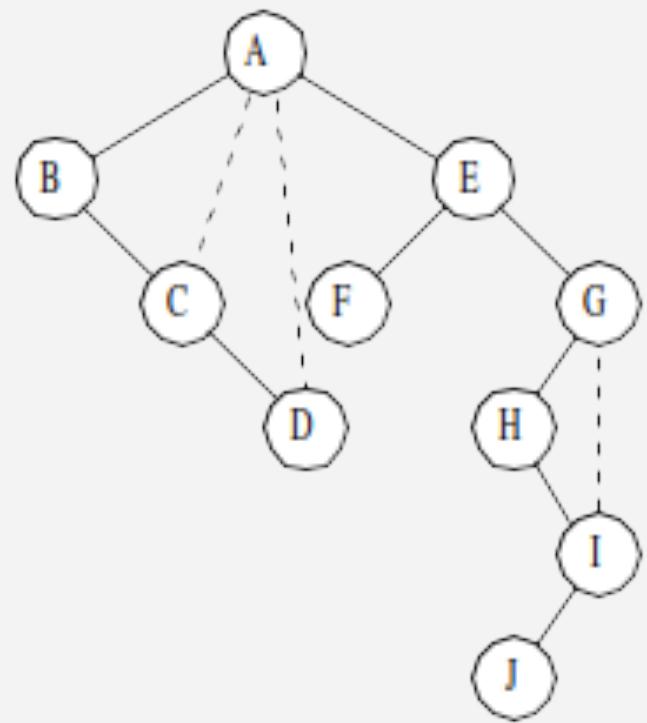
6.5.5 树和森林与二叉树的转换

1. 二叉树还原为树或森林：树和森林都可以转换为二叉树。
2. 二者不同是：树转换成的二叉树，其根结点必然无右孩子，而森林转换后的二叉树，其根结点有右孩子。

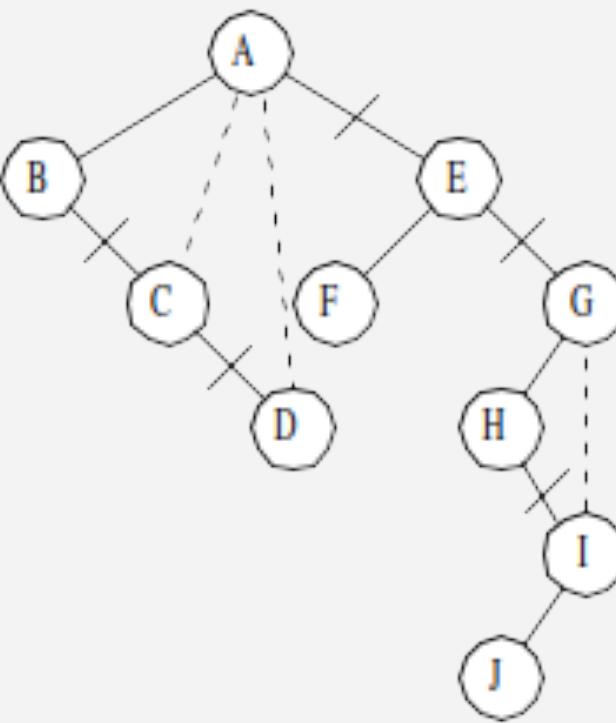
6.5.5 树和森林与二叉树的转换

将一棵二叉树还原为树或森林，具体方法如下：

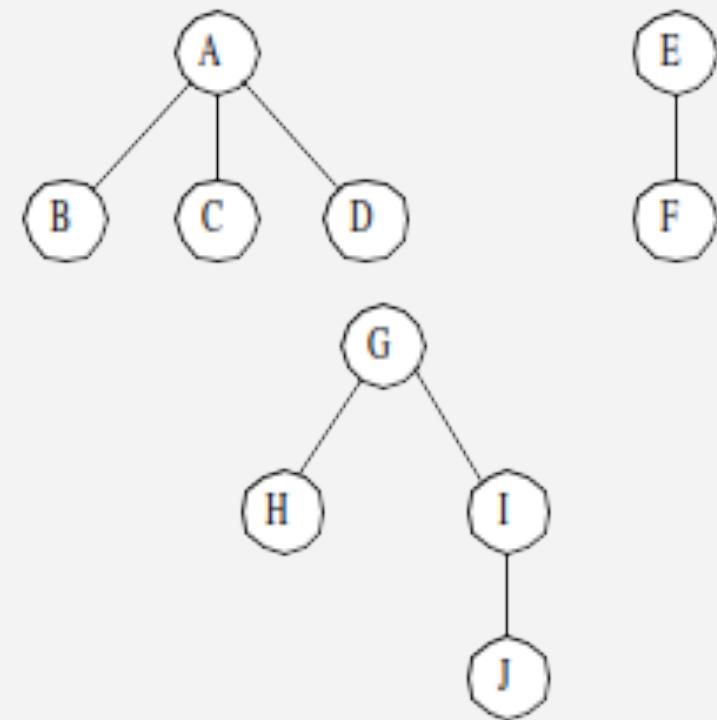
- (1) 若某结点是其双亲的左孩子，则把该结点的右孩子、右孩子的右孩子……都与该结点的双亲结点用线连起来。
- (2) 删掉原二叉树中所有双亲结点与右孩子结点的连线。
- (3) 整理由(1)、(2)两步所得到的树或森林，使之结构层次分明。



(a) 添加连线



(b) 删除右孩子结点的连线



(c) 整理

6.5.5 树和森林与二叉树的转换

用递归的方法描述上述转换过程。若B是一棵二叉树，T是B的根结点，L是B的左子树，R为B的右子树，且B对应的森林F(B)中含有n棵树为 T_1, T_2, \dots, T_n ，有

- (1) B为空，则F(B)为空的森林($n=0$)。
- (2) B非空，则F(B)中第一棵树 T_1 的根为二叉树B的根T； T_1 中根结点的子树森林由B的左子树L转换而成，即 $F(L) = \{T_{11}, \dots, T_{1m}\}$ ；B的右子树R转换为F(B)中其余树组成的森林，即 $F(R) = \{T_2, T_3, \dots, T_n\}$ 。

根据这个递归的定义，我们同样可以写出递归的转换算法。

练习

已知一棵有 2011 个结点的树，其叶结点个数为 116，该树对应的二叉树中无右孩子的结点个数是

- A. 115
- B. 116
- C. 1895
- D. 1896

参考答案：D

分析：每个分支结点（个数为 $2011 - 116$ ）最右的孩子在二叉树中就无右孩子，根本身在二叉树中无右孩子。

6.6 哈夫曼树与哈夫曼编码

1. 哈夫曼树又称最优树，是一类带权路径长度最短的树。
2. 哈夫曼树也可以称为最优二叉树。

6.6.1 哈夫曼树的基本概念

1. 路径：从树的一个结点到另一个结点的分支构成这两个结点之间的路径。对于哈夫曼树特指从根结点到某结点的路径。
2. 路径长度：路径上的分支数目。
3. 树的路径长度：从树根到每一个结点的路径长度之和。
4. 权：赋予某个事物的一个量，是对事物的某个或某些属性的数值化描述。

6.6.1 哈夫曼树的基本概念

1. 结点的带权路径长度：从树根到结点之间的路径长度与结点上权的乘积。
2. 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

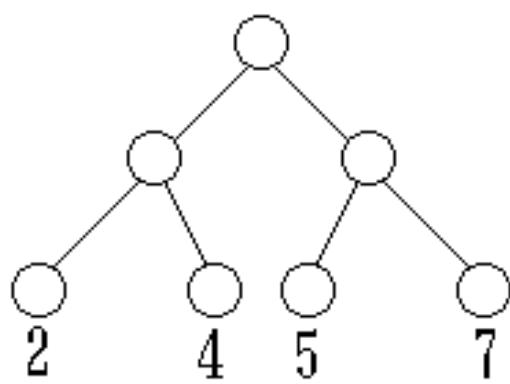
$$WPL = \sum_{i=1}^n w_i * l_i$$

图中三棵二叉树的带权路径长度分别为：

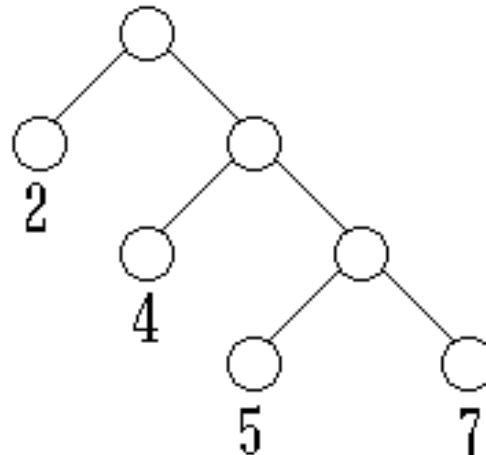
$$WPL(a) = 7 \times 2 + 5 \times 2 + 2 \times 2 + 4 \times 2 = 36$$

$$WPL(b) = 4 \times 2 + 7 \times 3 + 5 \times 3 + 2 \times 1 = 46$$

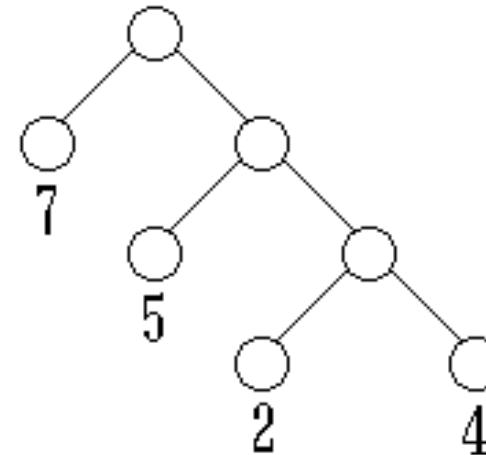
$$WPL(c) = 7 \times 1 + 5 \times 2 + 2 \times 3 + 4 \times 3 = 35$$



(a) $WPL = 36$



(b) $WPL = 46$



(c) $WPL = 35$

6.6.1 哈夫曼树的基本概念

1. 在所有含n个叶子结点、并带相同权值的二叉树中，必存在一棵其带权路径长度取最小值的树，称为“最优树”，或“哈夫曼树”（Huffman Tree）。
2. 带权路径长度达到最小的二叉树即为哈夫曼树（最优二叉树）。
3. 在哈夫曼树中，权值越大的结点离根越近。

6.6.2 哈夫曼树构造算法

哈夫曼树的构造：哈夫曼最早给出了一个带有一般规律的算法，称为哈夫曼算法。

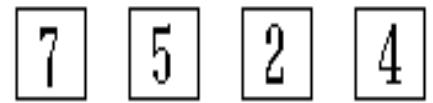
- 1、用给定的n个权值 $\{w_1, w_2, \dots, w_n\}$ 对应的n个结点构成n棵二叉树的森林 $F=\{T_1, T_2, \dots, T_n\}$ ，其中每一棵二叉树 $T_i (1 \leq i \leq n)$ 都只有一个权值为 w_i 的根结点，其左、右子树为空。
- 2、在森林F中选择两棵根结点权值最小的二叉树，作为一棵新二叉树的左、右子树，标记新二叉树的根结点权值为其左右子树的根结点权值之和。 ?

6.6.2 哈夫曼树构造算法

哈夫曼树的构造：哈夫曼最早给出了一个带有一般规律的算法，称为哈夫曼算法。

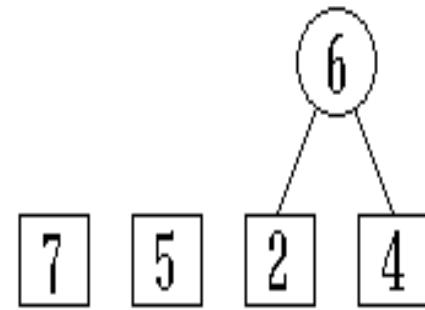
- 3、从F中删除被选中的那两棵二叉树，同时把新构成的二叉树加入到森林F中。
- 4、重复（2）、（3）操作，直到森林中只含有一棵二叉树为止，此时得到的这棵二叉树就是哈夫曼树。

$F : \{7\} \{5\} \{2\} \{4\}$



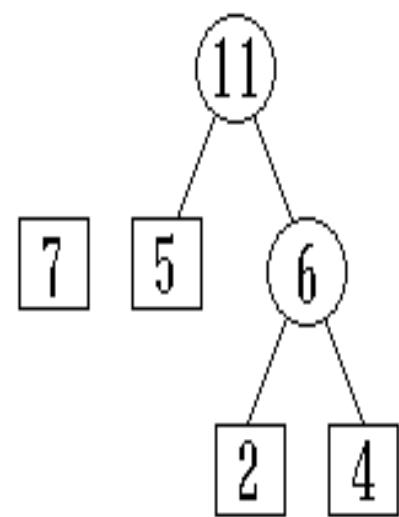
(a) 初始

$F : \{7\} \{5\} \{6\}$



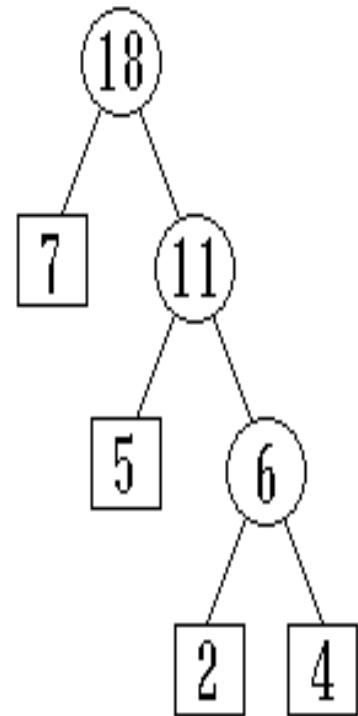
(b) 合并 {2} {4}

$F : \{7\} \{11\}$



(c) 合并 {5} {6}

$F : \{18\}$



(d) 合并 {7} {11}

6.6.2 哈夫曼树构造算法

假设有一组权值 $\{5, 29, 7, 8, 14, 23, 3, 11\}$ ，下面我们将利用这组权值演示构造哈夫曼树的过程。

第一步：以这8个权值作为根结点的权值构造具有8棵树的森林

5

29

7

8

14

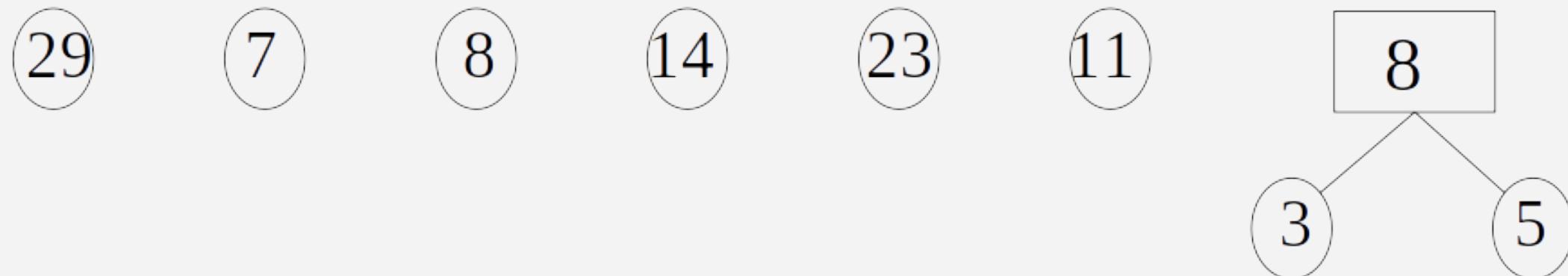
23

3

11

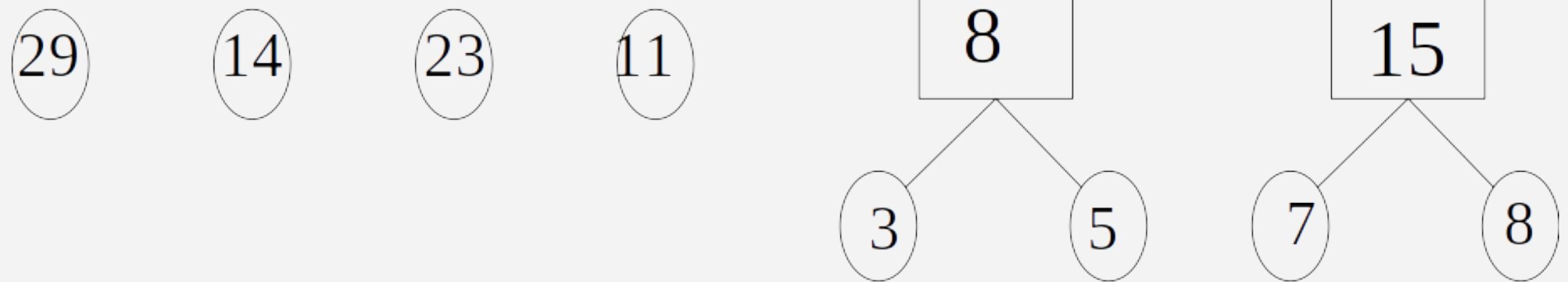
6.6.2 哈夫曼树构造算法

第二步：从中选择两个根的权值最小的树3，5作为左右子树构造一棵新树，并将这两棵树从森林中删除，并将新树添加进去。

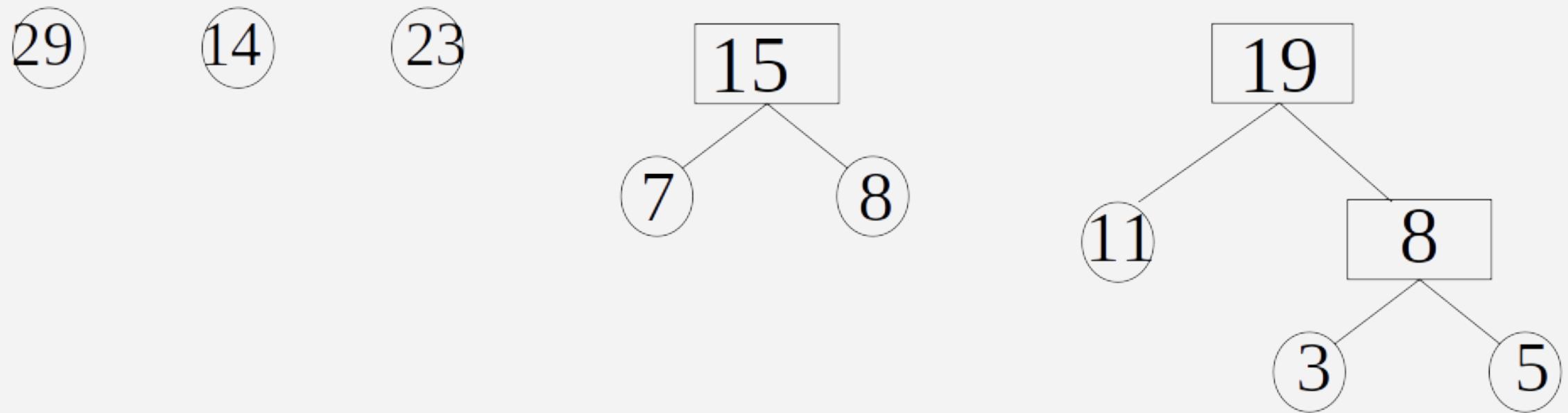


6.6.2 哈夫曼树构造算法

第三步：重复第二步过程，直到森林中只有一棵树为止；选择
7, 8



6.6.2 哈夫曼树构造算法



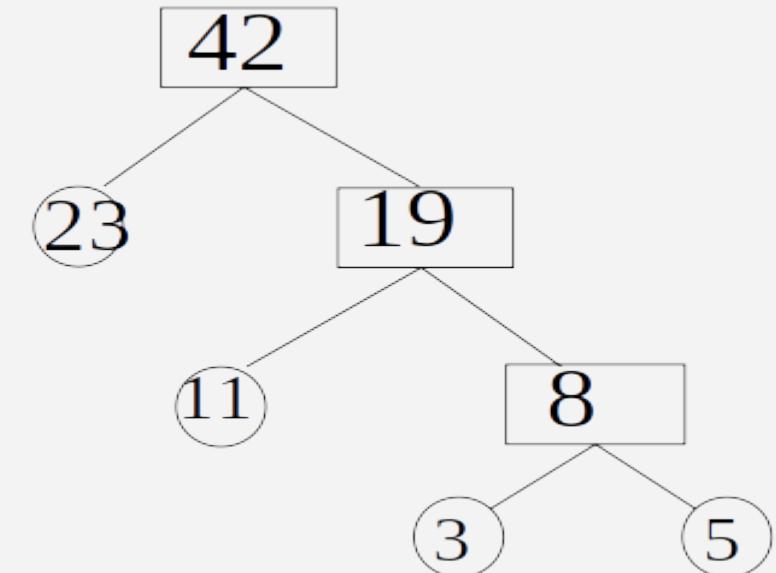
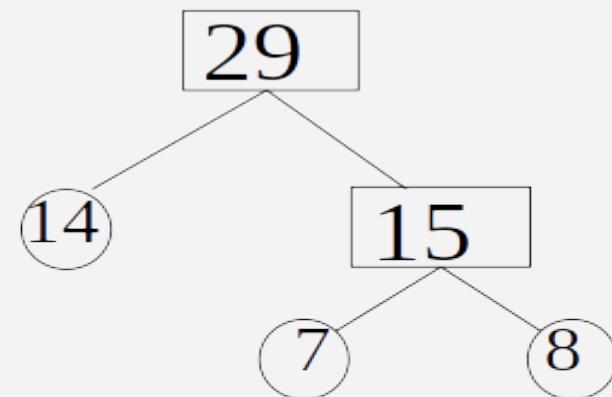
6.6.2 哈夫曼树构造算法

选择 8 , 11

选择 14 , 15

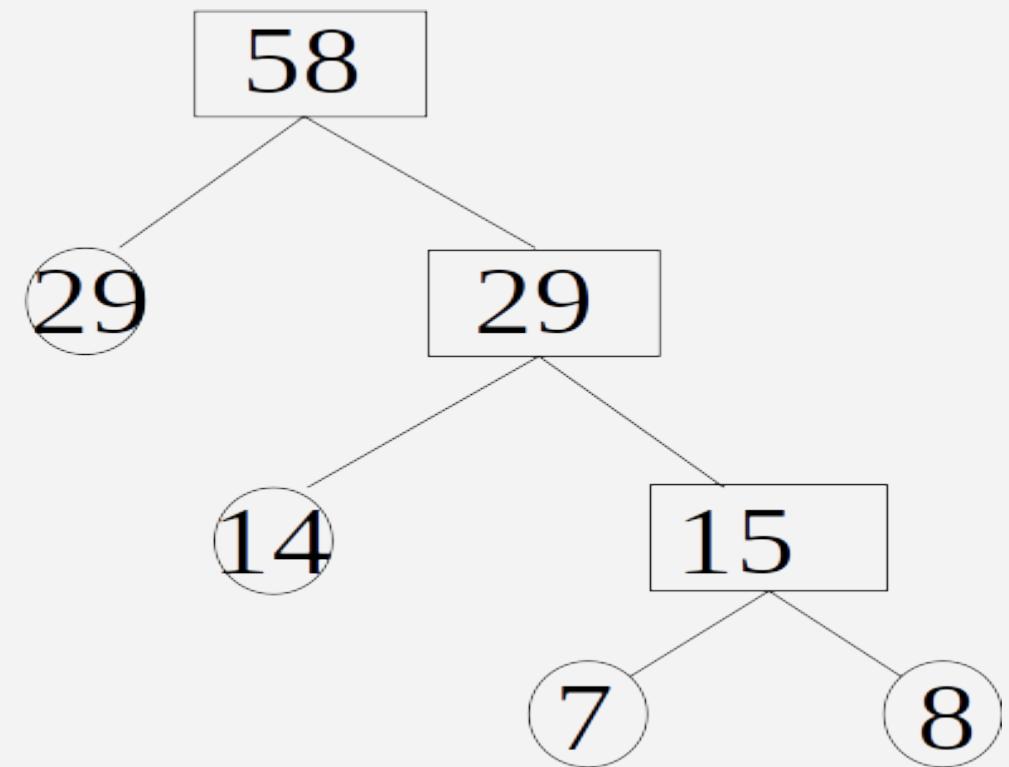
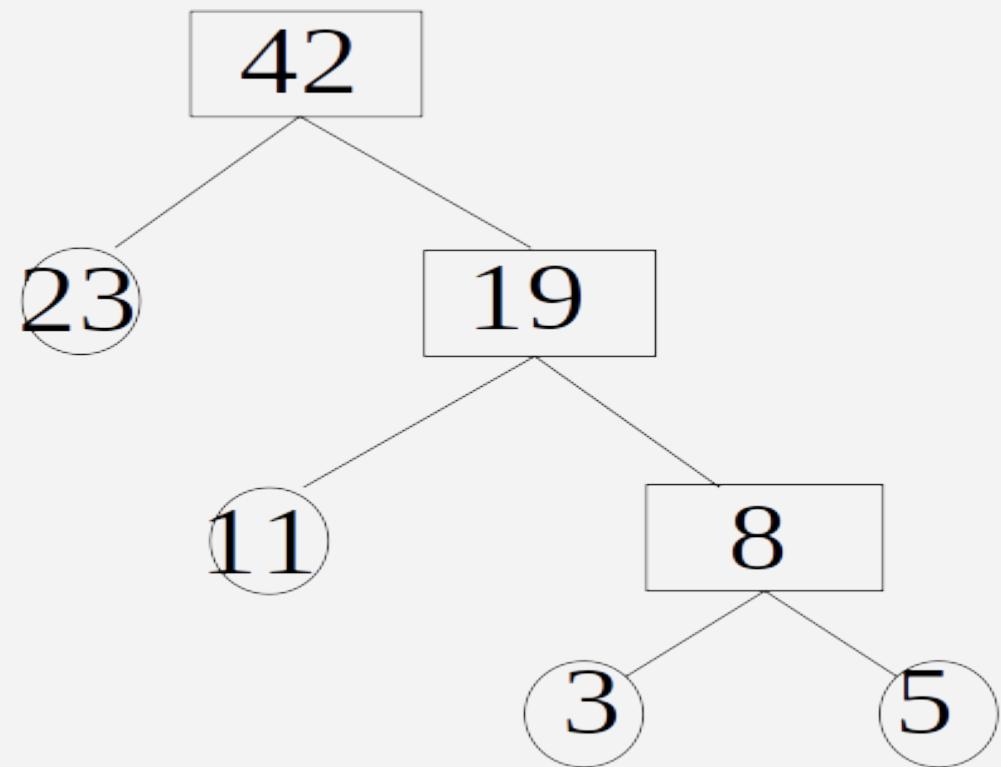
选择 19 , 23

29



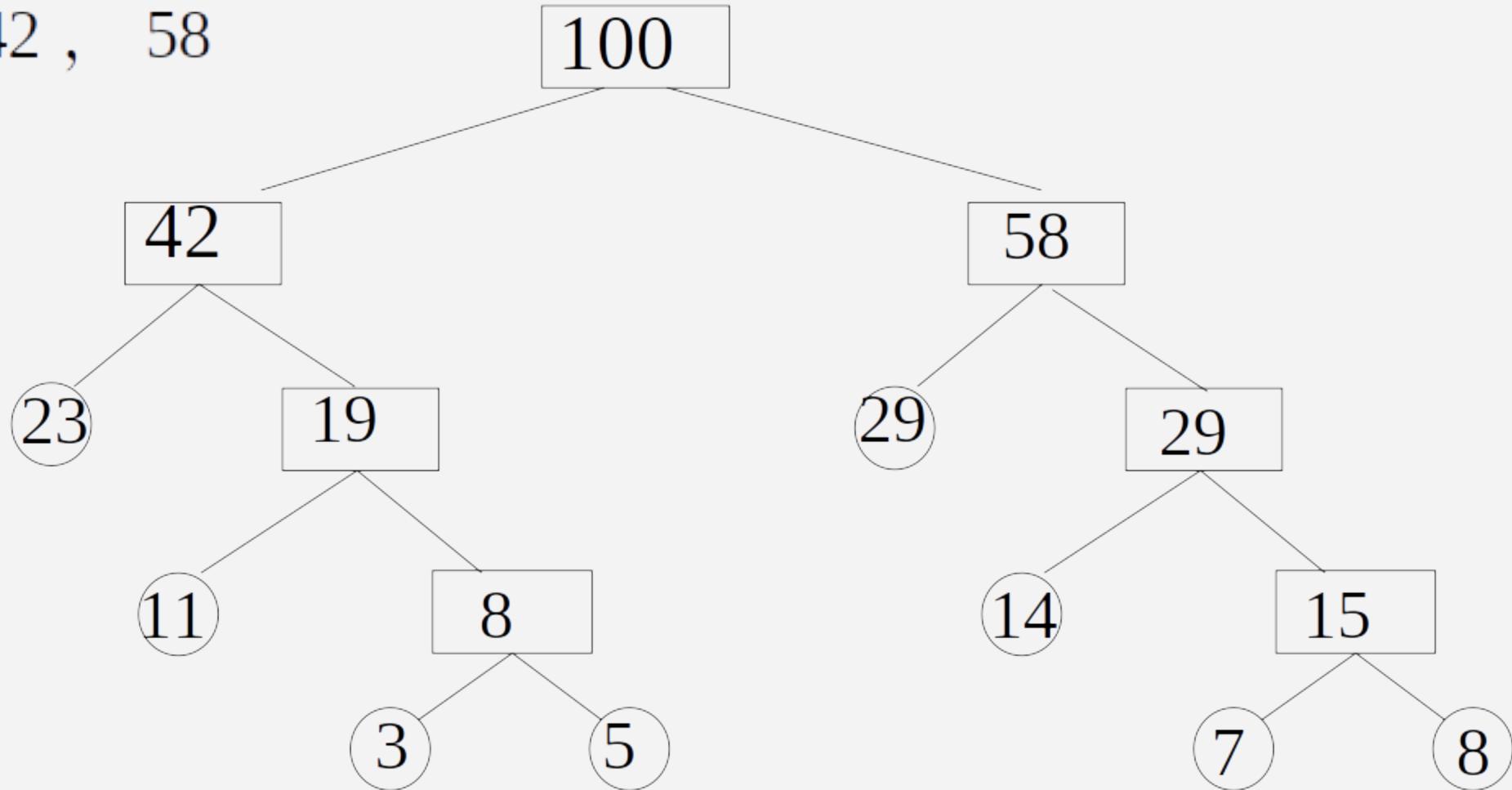
6.6.2 哈夫曼树构造算法

选择 29 , 29



6.6.2 哈夫曼树构造算法

选择 42 , 58



6.6.3 哈夫曼编码

1. 目前进行快速远距离通信的主要手段是电报，即将需要传送的文字转换成由二进制的字符组成的字符串。
2. 在传送电文时，希望总长尽可能地短。如果对每个字符设计长度不等的编码，而且让电文中出现次数较多的字符采用尽可能短的编码，则传送电文的总长就可以减少。

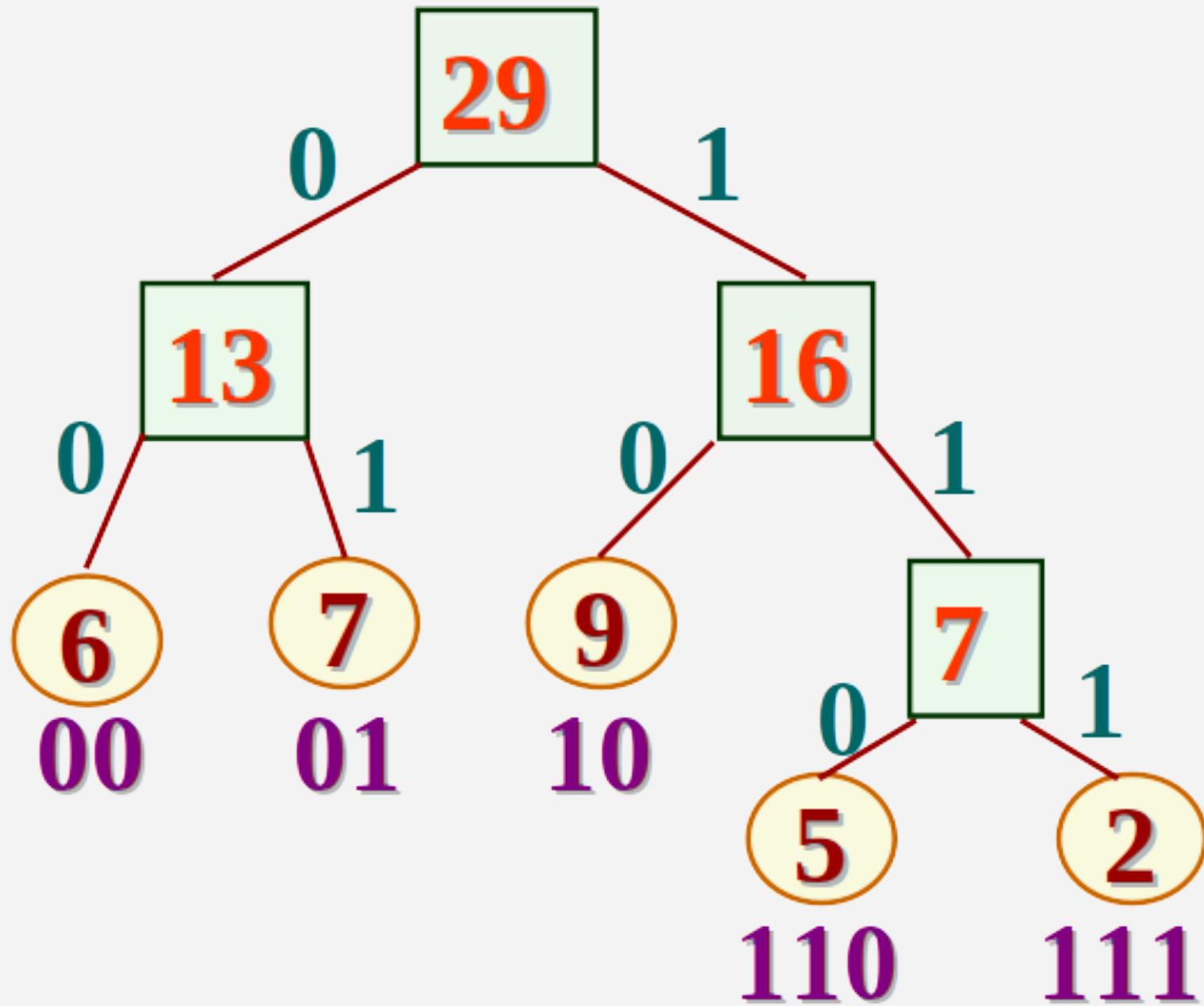
6.6.3 哈夫曼编码

1. 如果要设计长短不等的编码，则必须是任一个字符的编码都不是另外一个字符编码的前缀。
2. 如果在一个编码系统中，任一个编码都不是其他任何编码的前缀，则称此编码系统中的编码是前缀码。
3. 利用哈夫曼树可以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，使得编码的总长度最短。

6.6.3 哈夫曼编码

二进制的前缀编码可以利用哈夫曼树来设计实现。

1. 约定哈夫曼树的左分支表示0，右分支表示1
2. 从根结点到叶子结点的路径上分支字符组成的字符串作为该叶子结点字符的编码。



6.6.3 哈夫曼编码

设计电文总长最短的二进制前缀编码，即为以n种字符出现的频率作权，设计一棵哈夫曼树的问题，由此得到的二进制前缀编码便称为哈夫曼编码。

6.6.3 哈夫曼编码

假设每种字符在电文中出现的次数为 w_i ，其编码长度为 l_i ，电文中只有n种字符，则电文总长度为

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

对应到二叉树上，如果置 w_i 为叶子结点的权， l_i 为从根到叶子的路径长度，则WPL正好为二叉树上带权路径长度。

6.6.3 哈夫曼编码

例：设给出一段报文：

CAST CAST SAT AT A TASA

字符集合是 $\{ C, A, S, T \}$ ，各个字符出现的频度（次数）是 $W = \{ 2, 7, 4, 5 \}$ 。

若给每个字符以等长编码

A : 00 T : 10 C : 01 S : 11

则总编码长度为 $(2+7+4+5) * 2 = 36$ 。

若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度。

因各字符出现的概率为 $\{ 2/18, 7/18, 4/18, 5/18 \}$ 。

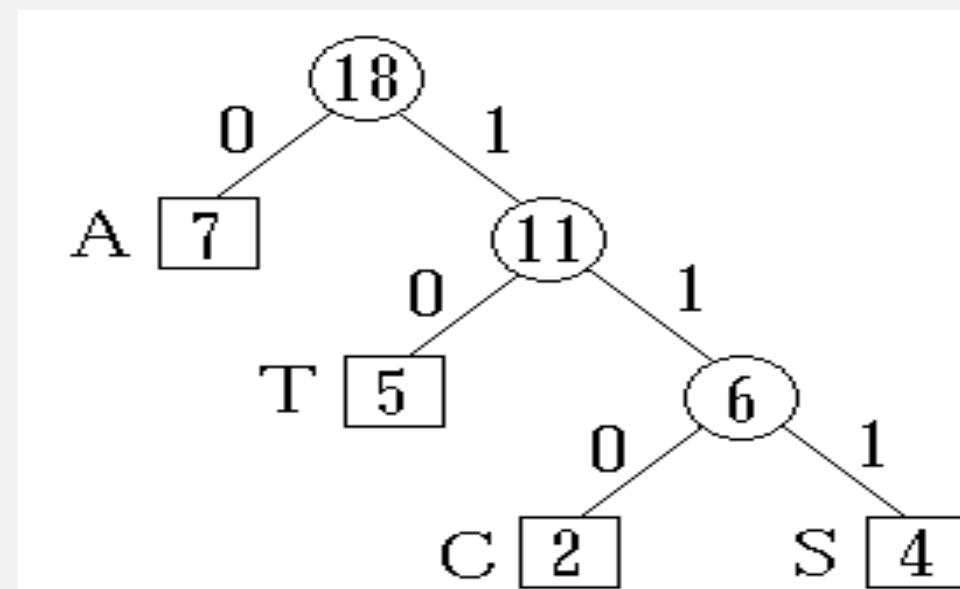
6.6.3 哈夫曼编码

化整为 { 2, 7, 4, 5 }，以它们为各叶结点上的权值，建立哈夫曼树。左分支赋 0，右分支赋 1，得哈夫曼编码(变长编码)。

A : 0 T : 10 C : 110 S : 111
它的总编码长度： $7*1+5*2+(2+4)*3 = 35$ 。比等长编码的情形要短。

总编码长度正好等于哈夫曼树的带权路径长度WPL。

哈夫曼编码是一种前缀编码。
解码时不会混淆。



6.7 树的计数

二叉树的计数：问题是如果有n个数据值，可能构造多少种不同的二叉树？这就是二叉树的计数问题。
有0个，1个，2个，3个结点的不同二叉树如下。

Φ



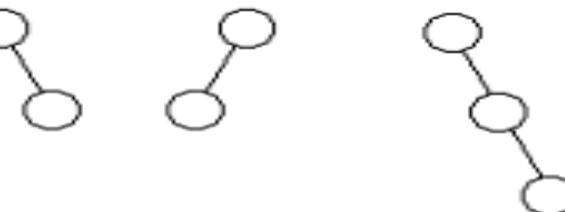
(a) $b_0 = 1$



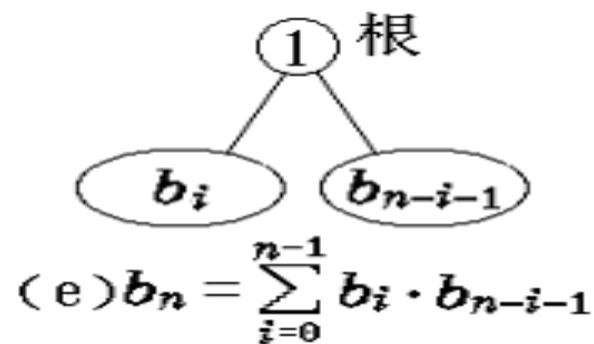
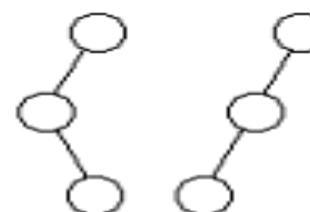
(b) $b_1 = 1$



(c) $b_2 = 2$



(d) $b_3 = 5$

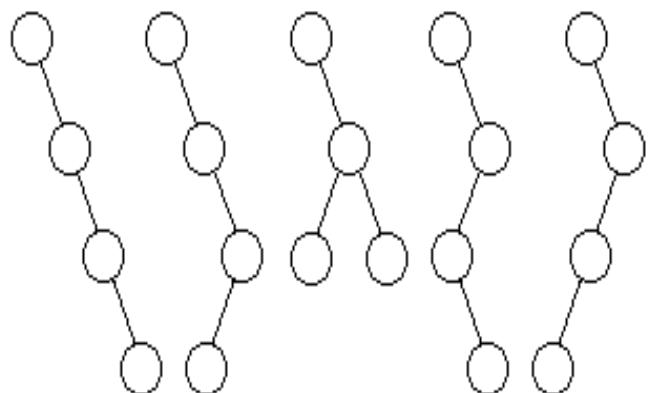


6.7 树的计数

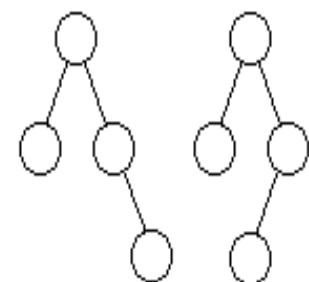
计算具有n个结点的不同二叉树的棵数

$$b_n = \frac{1}{n+1} C_{2n}^n = \frac{1}{n+1} \frac{(2n)!}{n! n!}$$

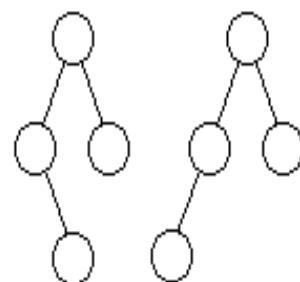
具有4个结点的不同二叉树



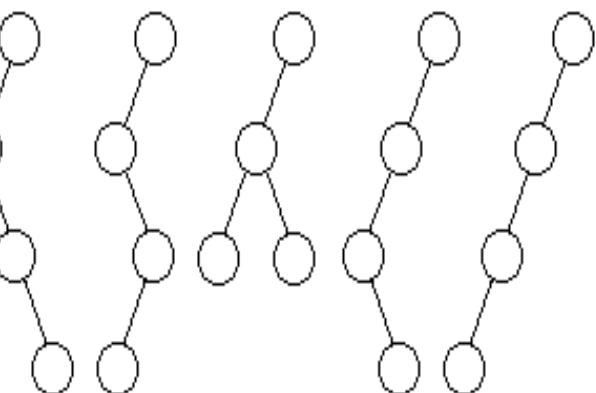
(a) $i = 0$



(b) $i = 1$



(c) $i = 2$



(d) $i = 3$

6.7 树的计数

1. 根据树和森林与二叉树的转换，由二叉树的计数可推得树的计数。
2. 具有 n 个结点的不同形态的树的数目，应该和具有 $n-1$ 个结点的不同二叉树的数目相同。
3. 具有 n 个结点的不同形态的森林的数目，应该和具有 n 个结点的不同二叉树的数目相同。

练习

- 1 一棵完全二叉树上有1001个结点，其中叶子结点的个数是（ ）
A. 250 B. 500 C. 254 D. 505 E. 以上答案都不对
 - 2 有n个叶子的哈夫曼树的结点总数为（ ）。
A. 不确定 B. $2n$ C. $2n+1$ D. $2n-1$
 - 3 具有10个叶结点的二叉树中有（ ）个度为2的结点，
A. 8 B. 9 C. 10 D. 11
 - 4 下述编码中哪一个不是前缀码（ ）。
A. $(00, 01, 10, 11)$ B. $(0, 1, 00, 11)$
C. $(0, 10, 110, 111)$ D. $(1, 01, 000, 001)$
- (E, D, B, B)

练习

5 n 个结点的线索二叉树上含有的线索数为（ ）

- A. $2n$
- B. $n - 1$
- C. $n + 1$
- D. n

6 一棵左右子树均不空的二叉树在先序线索化后，其中空的链域的个数是：（ ）。

- A. 0
- B. 1
- C. 2
- D. 不确定

7 一棵左子树为空的二叉树在先序线索化后，其中空的链域的个数是：
（ ）

- A. 不确定
- B. 0
- C. 1
- D. 2

(C, B, D)