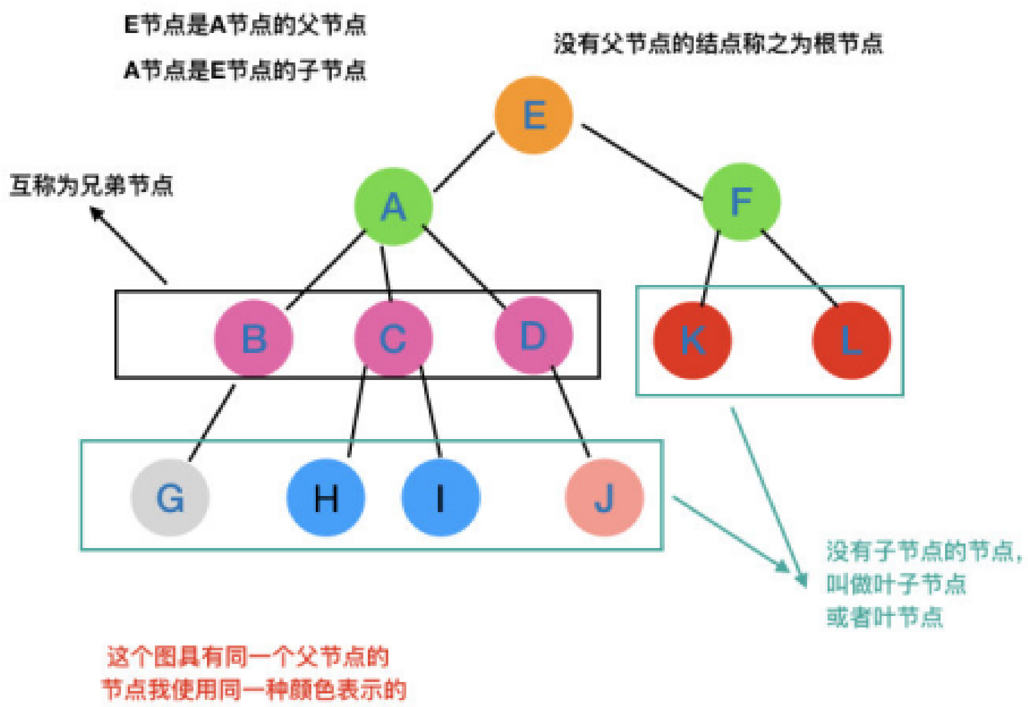
树与二叉树

🌳 的基本概念



✏ 基本术语

节点的度 结点拥有的子树的数目
叶子 度为零的结点
分支结点 度不为零的结点
树的度 树中结点的最大的度
层次 根结点的层次为1，其余结点的层次等于该结点的双亲结点的层次加1
树的高度 树中结点的最大层次
无序树 树中结点的各子树之间的次序是不重要的，可以交换位置
有序树 树中结点的各子树之间的次序是重要的，不可以交换位置
森林 0个或多个不相交的树组
节点的高度 节点到叶子节点的最长路径
节点的深度 根节点到这个节点所经历的边的个数
节点的层数 节点的深度 + 1
树的高度 根节点的高度
树的路径长度 树到每个节点长度的总和

树的性质

- 树的节点数n等于所有的节点度数之和加一

- 度为m的树中第i层最多有 m^{i-1}个节点
- 高度为h的m叉树至多有 $\frac{m^h-1}{(m-1)}$ 个节点
- 度为m，具有n个节点的树的最小高度h为 $log_m^{n(m-1)+1}$ (向上取整)
- 度为m,具有n个节点的树的最大高度h为n-m+1

二叉🌳

二叉树与度为二的树的区别

二叉树可以为空

二叉树的实现

```
typedef struct TreeNode *BiTree;
struct TreeNode
{
        int data;
        BinTree lchild;
        BinTree rchild;
};
```

几种特殊二叉树

- 满二叉树
- 完全二叉树

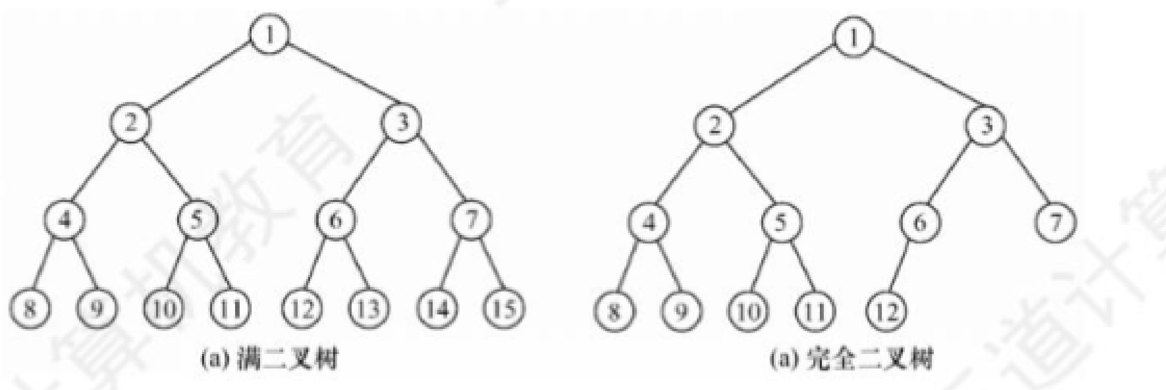
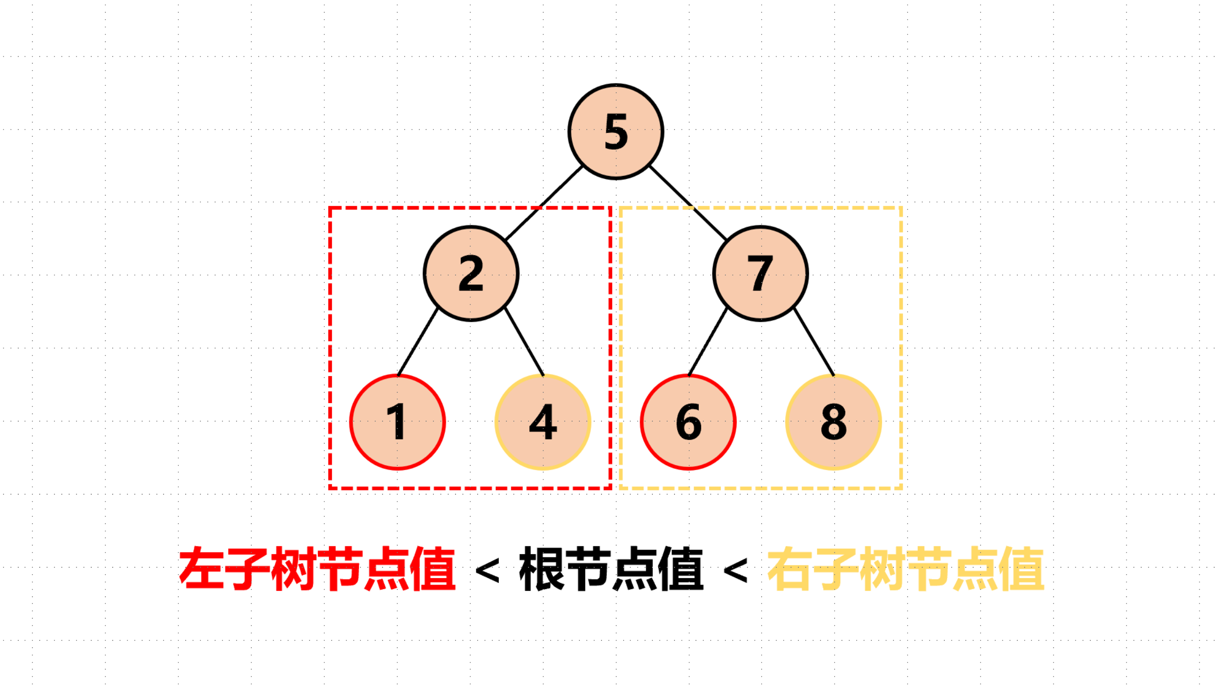
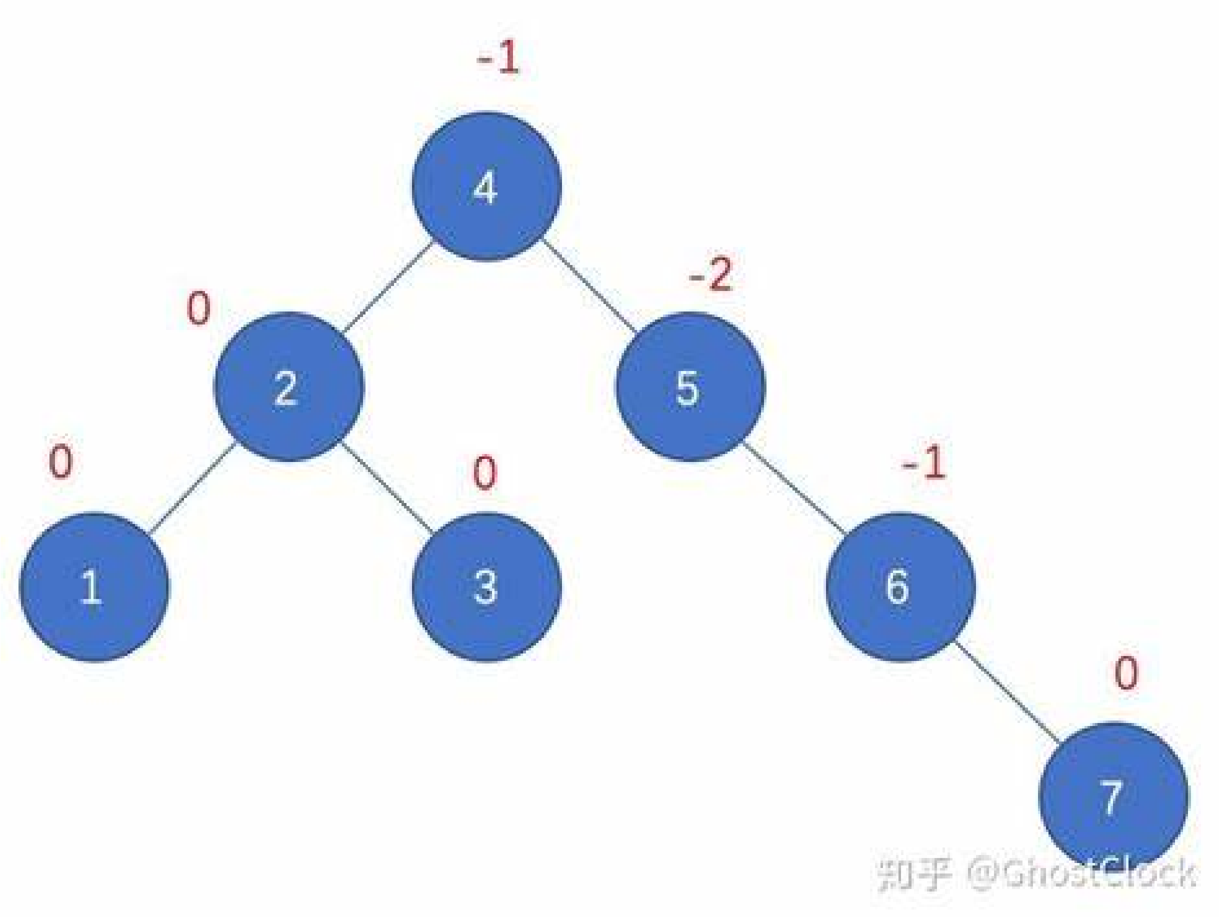


图 5.3　两种特殊形态的二叉树①

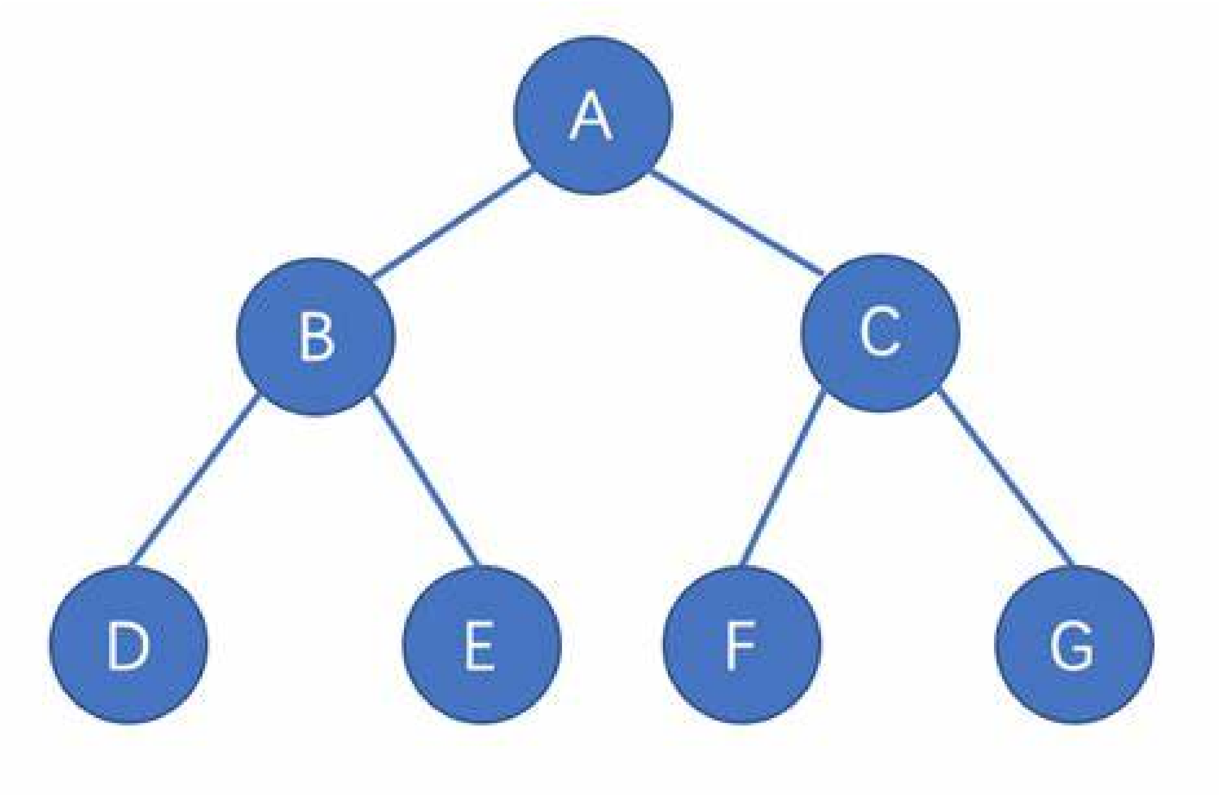
- 二叉排序树



- 平衡二叉树



- 正则二叉树
  - 正则二叉树。树中每个分支结点都有2个孩子，即树中只有度为0或2的结点。



二叉树的性质

- 非空二叉树上的叶结点数等于度为2的结点数加1，即 $n_0 = n_2 + 1$
- 非空二叉树的第k层最多有 2^{k-1}个节点
- 高度为h的二叉树至多有 $2^h - 1$个节点
- 具有n个节点的完全二叉树高度为 $[log_2^n] + 1$（向下取整）

二叉树的存储结构

顺序存储

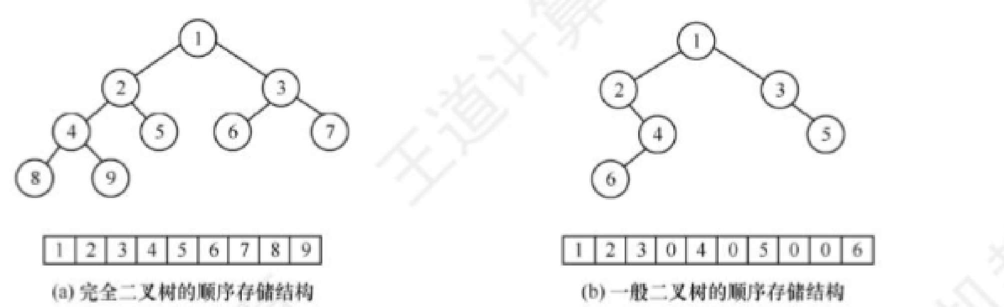


图 5.4　二叉树的顺序存储结构

满足任意性：存储所有节点哪怕只有一个节点也存储这一层所有节点

链式存储

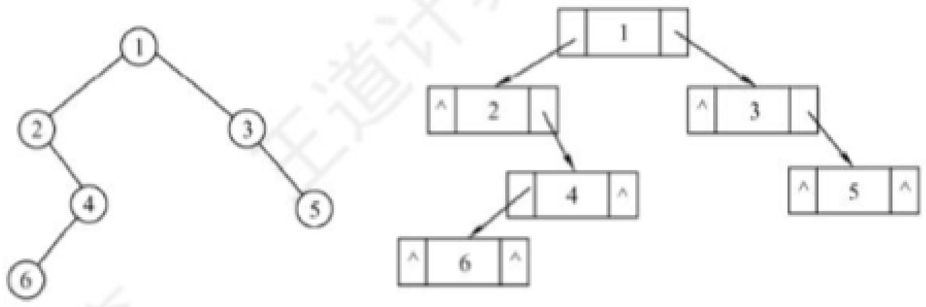


图 5.6 二叉链表的存储结构

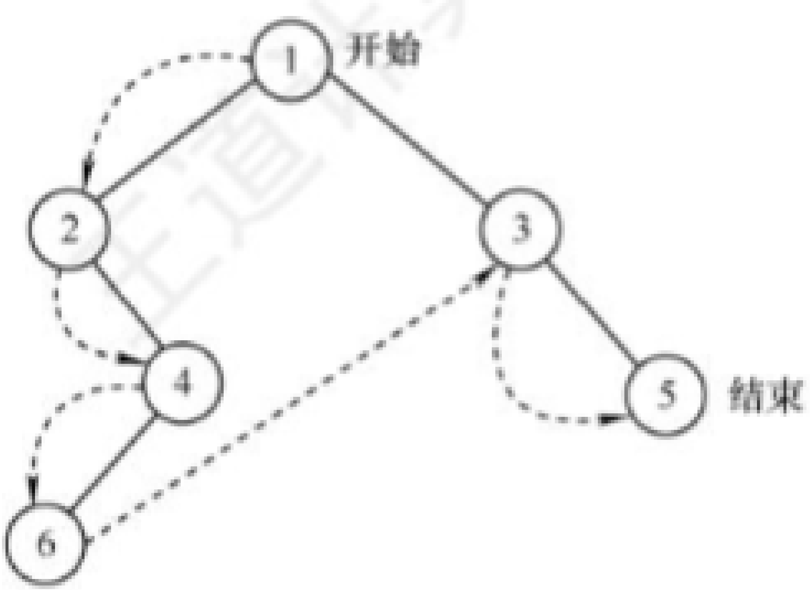
```
typedef struct BiTnode{
	ElemType data;
	struct BiTNode *lchild,*rchild;
}BiTNode,*BitTree;
```

n个节点n+1个空域

> **Abstract**
>
> 解题方法
>
> 1. 带值
> 2. 判断根节点
> 3. 注意任意性和唯一性

二叉树的遍历

先序遍历



代码

递归代码

```
void PreOrder (BiTree T)
{
	if(T!=NULL)
	{
		visit(T);
		PreOrder(T->lchild);
		PreOrder(T->rlchild);
	}
}
```

非递归代码

```
void PreOrder2(BiTree T)
{
	InitStack(S);
	BiTree p=T;
	while(p||IsEmpty(s))
	{
		if(p){
			vist(p);
			push(S,p);
			p=p->lchild;
		}
		{
			Pop(S,p);
			p=p->rchild;
		}
	}
}
```

中序遍历

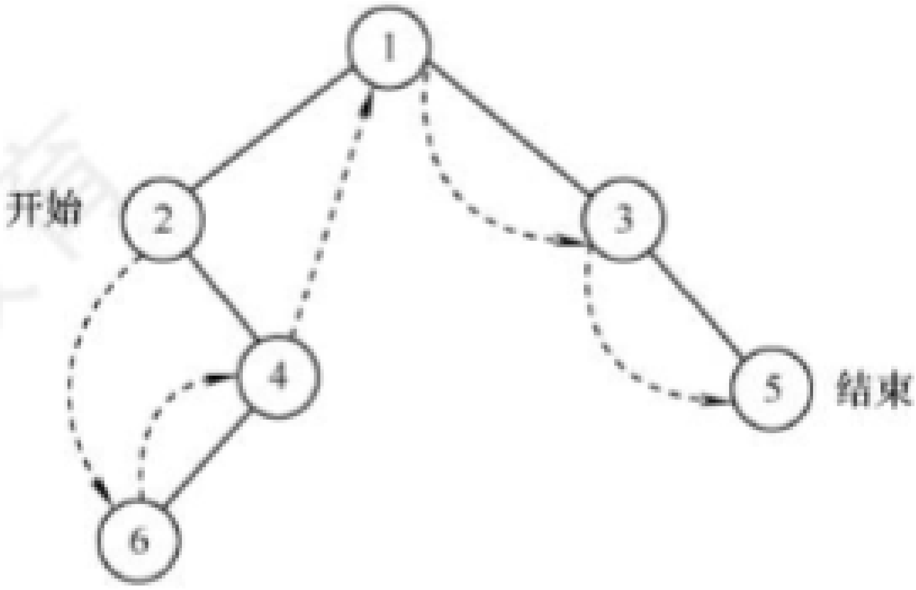


图 5.8　二叉树的中序遍历

代码

递归

```
void InOrder(BiTree T)
{
        if(T!=NULL)
        {
                InOrder(T->lchild);
                vist(T);
                InOrder(T->rchild);
        }
}
```

非递归

```
void InOrder2(BiTree T)
{
         InitStack(S);
         BiTree p=T;
         while(p||IsEmpty(S))
         {
                 if(p)
                 {
                         push(S,p);
                         p=p->lchild;
                 }
                 else
```

```
                {
                        Push(S,p);
                        visit(p);
                        p=p->rchild;
                }
        }
}
```

后序遍历

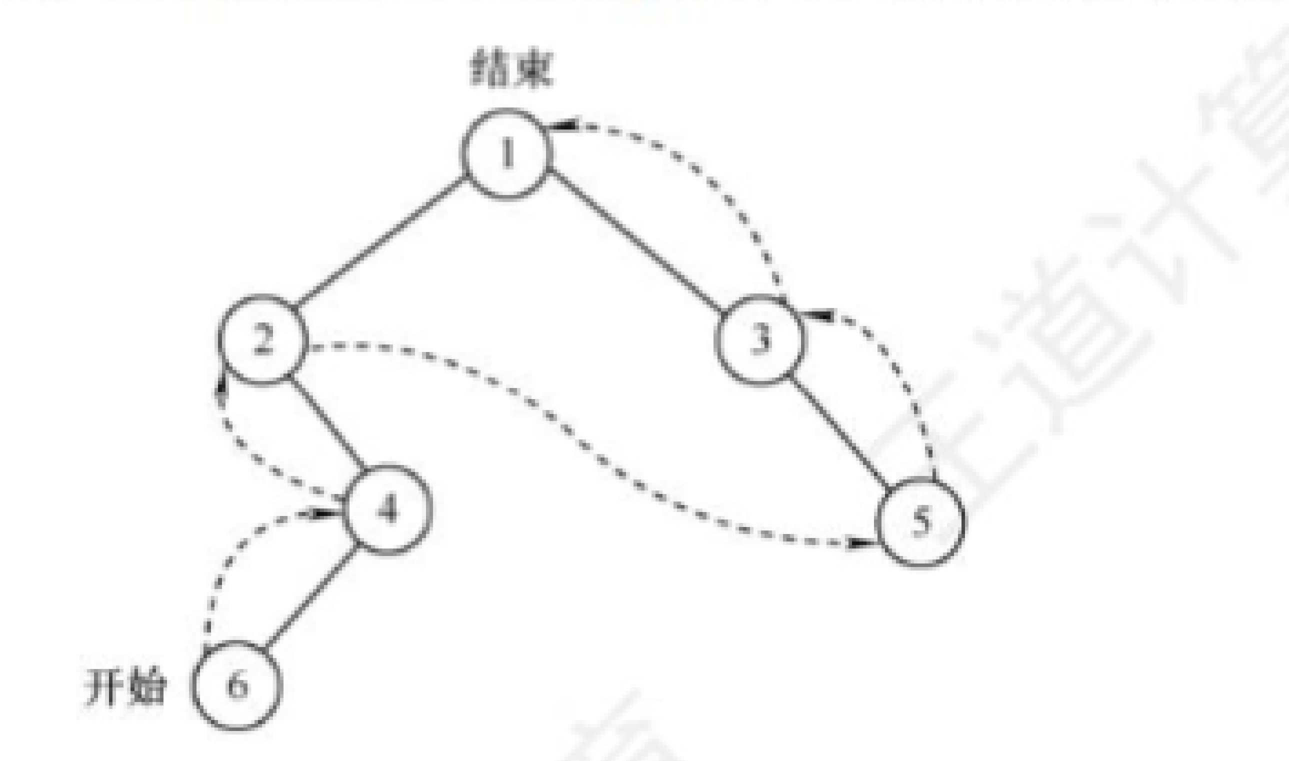


图 5.9　二叉树的后序遍历

代码

递归

```
void PostOrder(BiTree T)
{
        if(T!=NULL)
        {
                PostOrder(T->lchild);
                PostOrder(T-<rchild);
                visit(T);
        }
}
```

非递归

```
void PostOrder2(BiTree T)
{
	InitStack(S);
	BiTNode *p=T;
	BitNode *r=NULL;
	while(p||IsEmpty(S))
	{
		if(p)
		{
			push(S,p);
			p=p->lchild
		}
		else
		{
			GetTop(S,p);
			if(p->rchild&&p->rchild!=r);
				p=p->rchild;
			else
			{
				Pop(S,p);
				visit(p->data);
				r=p;
				p=NULL;
			}
		}
	}
}
```

层次遍历

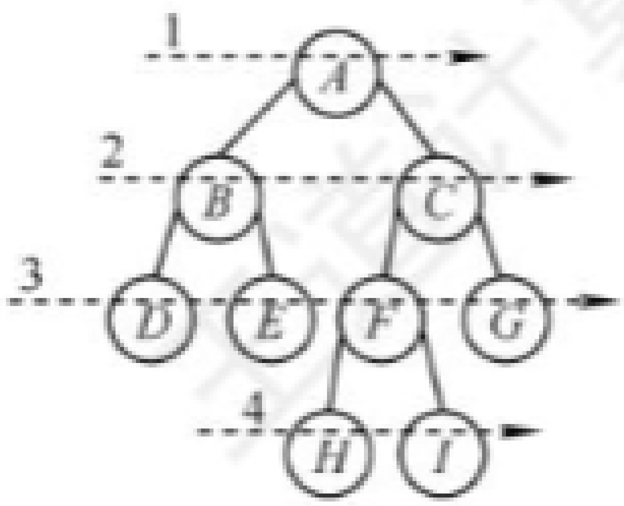


图 5.11　二叉树的层次遍历

代码

```
void Leve1Order(BiTree T)
{
        InitQueue(Q);
        BitTree p;
        EnQueue(Q,T);
        while(IsEmpty(Q))
        {
                Dequeue(Q,p);
                visit(p);
                if(p->lchild!=NULL)
                {
                        Enqueue(Q,p->lchild);
                }
                if(p->rchild!=NULL)
                {
                        Enqueue(Q,p->rchild);
                }
        }
}
```

Abstract

- 不能唯一确定一颗二叉树的是：先序序列和后序序列
- 后序遍历可以找到m到n直接的路径（其中m是n的祖先）

- 前序序列和中序序列的关系相当于以前序序列为入栈次序，以中序序列为出栈顺序

线索二叉树

- 后续线索二叉树不能有效解决求后续后继的问题，后续线索树的遍历仍需要栈的支持

节点结构

ltag=0，表示指向节点的左孩子	ltag=1，则表示lchild为线索，指向节点的直接前驱
rtag=0，表示指向节点的右孩子	rtag=1，则表示rchild为线索，指向节点的直接后继

中序线索化过程（前序，后序同）

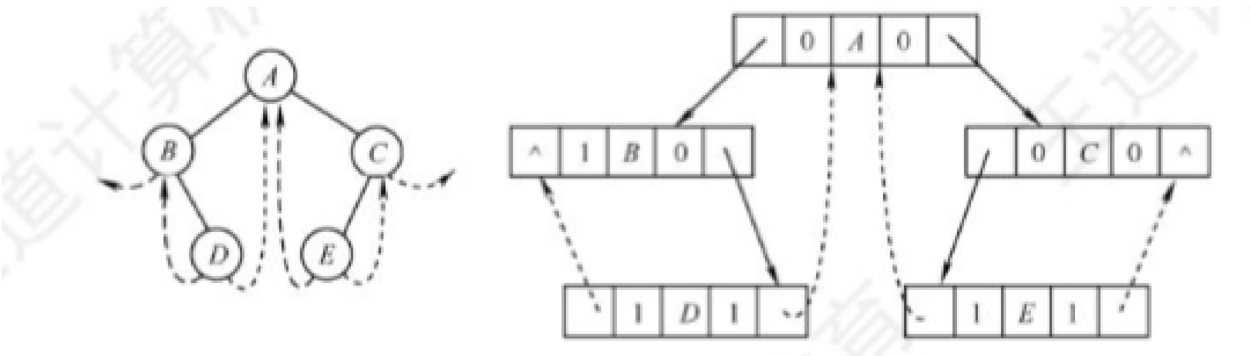


图 5.18 中序线索二叉树及其二叉链表示

易错题

线索二叉树是一种（ ）结构。

A. 逻辑　　B. 逻辑和存储　　C. 物理　　D. 线性

某二叉树的先序序列和后序序列正好相反，则该二叉树一定是（ ）。

A. 空或只有一个结点　　B. 高度等于其结点数

C. 任意一个结点无左孩子　　D. 任意一个结点无右孩子

🌳和森林

树的存储结构

双亲表示法

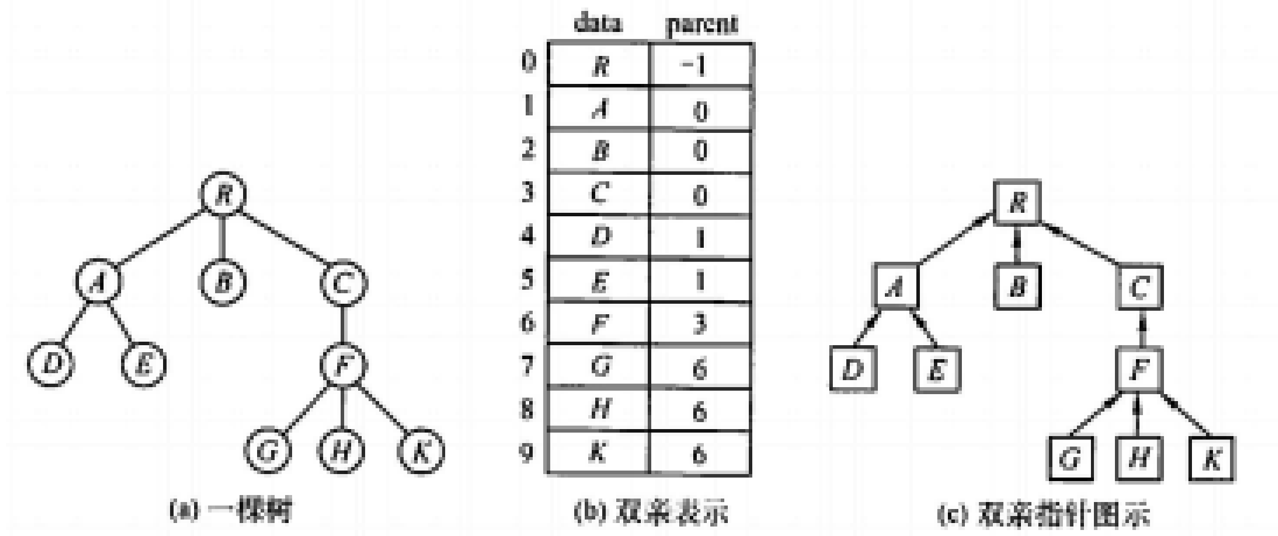
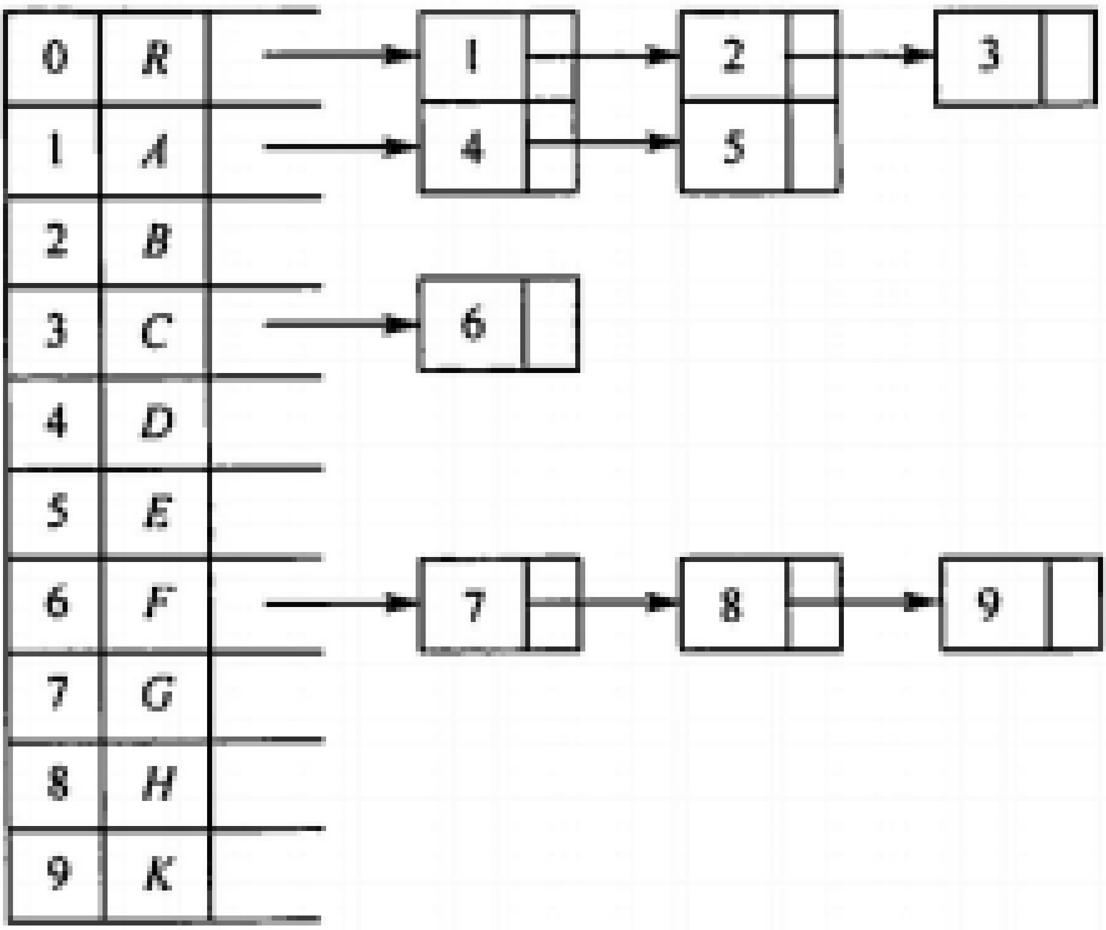


图 5.14 树的双亲表示法

- 采用一组连续空间来存储每个节点
- 在每个节点中设置一个伪指针
- 伪指针指示其双亲节点在数组中的位置

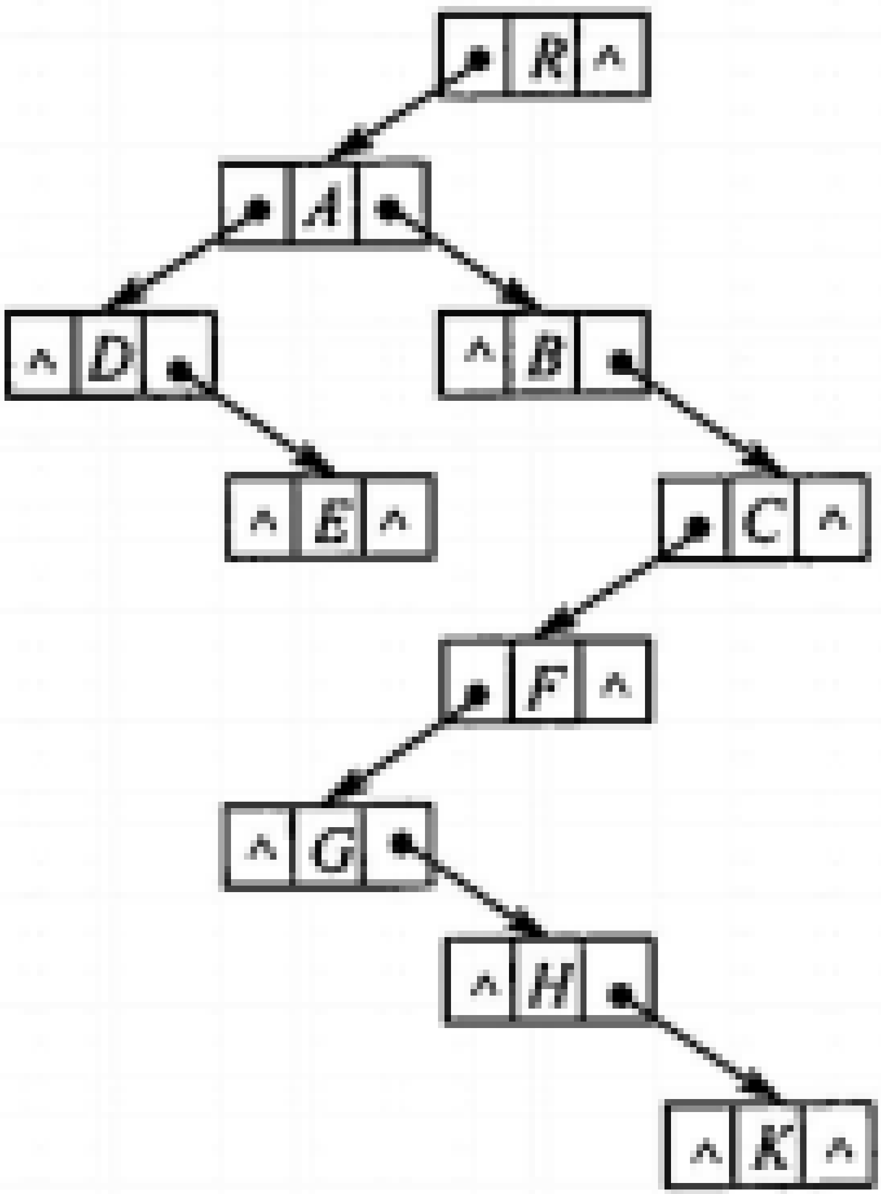
孩子表示法



(a) 孩子表示法

- 将每个节点的孩子节点用单链表连接

孩子兄弟表示法



(b) 孩子兄弟表示法

- 又叫二叉树表示法
- 以二叉链表作为树的存储结构
- 节点内容包含3个部分 【孩子节点】 【数据】 【兄弟节点】

树，森林，二叉树的转换

树与二叉树

1. 在兄弟节点之间加一连线
2. 对每个节点，只保留它与第一个孩子的连线

3. 以树根为轴心，顺时针旋转45度

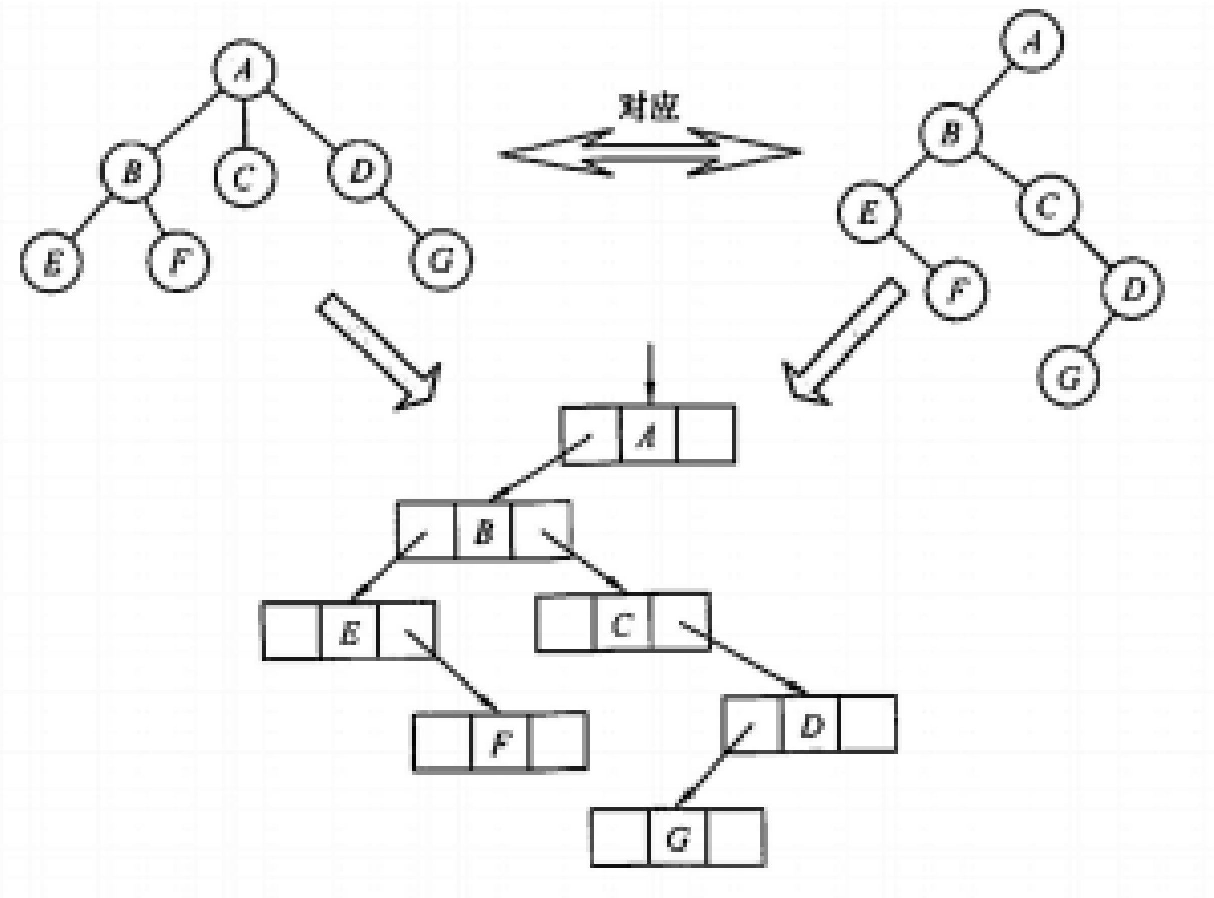


图 5.16　树与二叉树的对应关系

树，森林与二叉树

1. 将森林中的每棵树转换成相应的二叉树
2. 每棵树的根也可以视为兄弟关系，在每棵树的之间加一根连线
3. 以第一棵树的根为轴心旋转45°

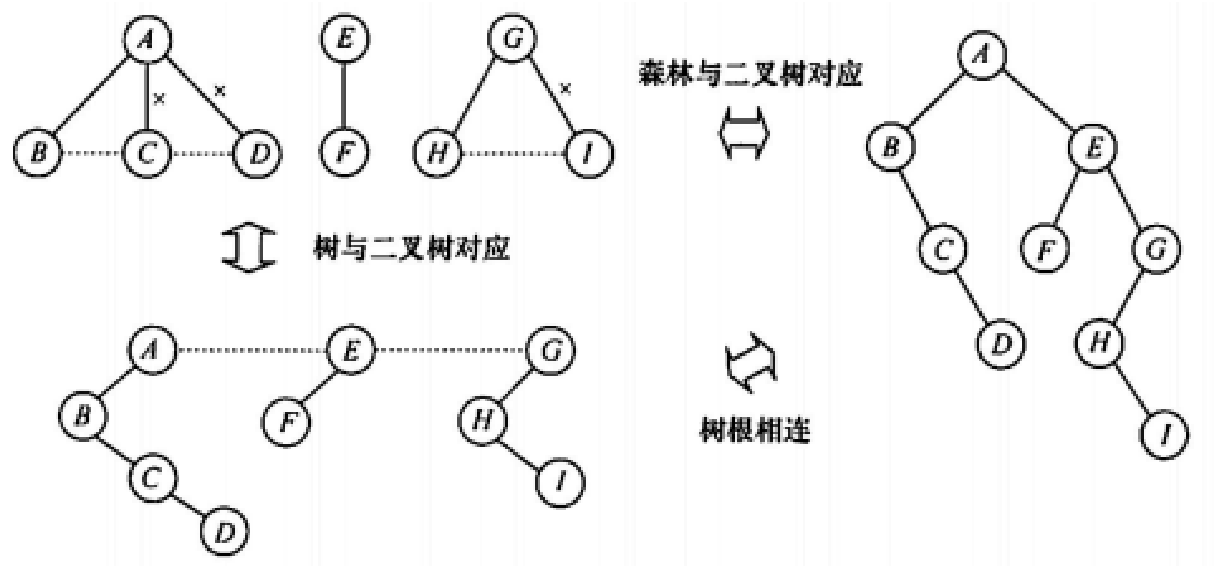


图 5.17　森林与二叉树的对应关系

树和森林的遍历

表 5.1　树和森林的遍历与二叉树遍历的对应关系

树	森　林	二　叉　树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

树与二叉树的应用

哈夫曼树和哈夫曼编码

哈夫曼树

树的带权路径长度最小的二叉树
每次把队列中值最小的合并，合并后的值放入队列中再继续比较

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

特点

- 没有度为 1 的结点
- n个叶结点的哈夫曼树共有 2n-1 个结点
- 哈夫曼树的任意非叶结点的左右子树交换后仍是哈夫曼树
- 对同一组权值，可能存在不同构的多棵哈夫曼树
- 哈夫曼树不一定是完全二叉树

哈夫曼编码

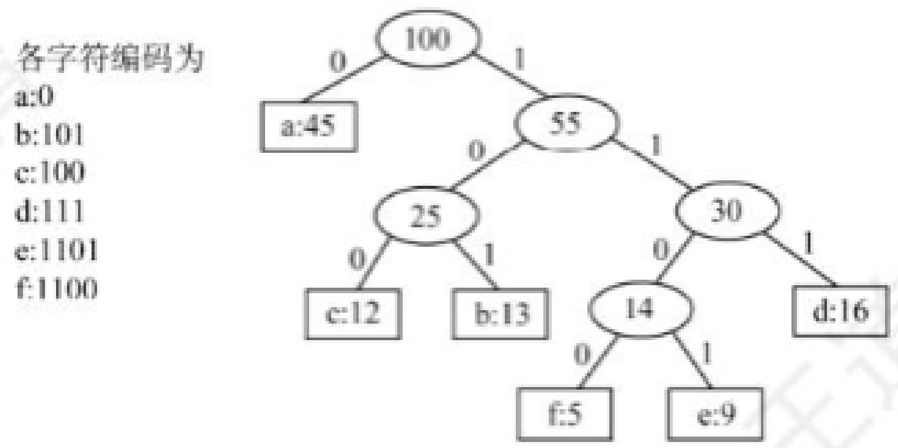


图 5.28 由哈夫曼树构造哈夫曼编码

并查集

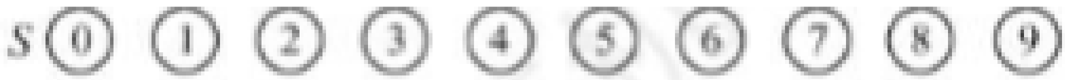
- 并查集是一种简单的集合表示，支持3种操作
- 并查集的存储结构是双亲表示法存储的树，主要是为了方便两个主要的操作

```
#define SIZE 100
int UFSets[Size];
```

主要操作

(1)初始化

```
void Inital(int S[ ])
{
	for(int i=0;i<SIZE;i++)
	{
		S[i]=-1;
	}
}
```



(a) 全集合S初始化时形成一个森林

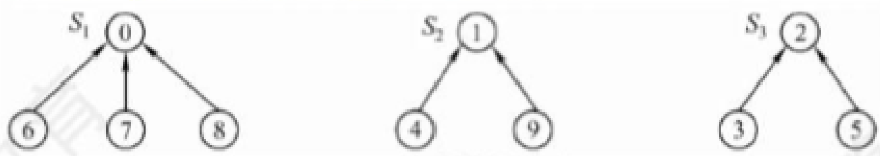
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

(b) 初始化时形成的(森林)双亲表示

图 5.29 并查集的初始化

(2)并查集的Union操作

```
void Union(int S[ ],int root1,int root2)
{
        if(root1==root2)
                return;
        else
                S[root2]=root1;
}
```

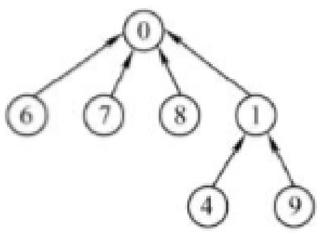


(a) 集合的树形表示

0	1	2	3	4	5	6	7	8	9
-4	-3	-3	2	1	2	0	0	0	1

(b) 集合S_1、S_2和S_3的(森林)双亲表示

图 5.30 用树表示并查集



0	1	2	3	4	5	6	7	8	9
-7	0	-3	2	1	2	0	0	0	1

图 5.31 $S_1 \cup S_2$可能的表示方法

(3)查找元素并返回根

```
int Find(int S[ ],int x)
{
        while(S[x]>=0)
                x=S[x];
```

```
        return x;
}
```

并查集优化

- 在Union之前，先判断成员数量，少->多，降低深度到 $[log_2^n]+1$
- 压缩路径，x不在第二层，则将根到x路径上的所有元素都变成跟的孩子。 降低深度不超过 $O(\alpha(n))$

易错题

. 并查集的结构是一种（ ）。

A. 二叉链表存储的二叉树　　B. 双亲表示法存储的树

C. 顺序存储的二叉树　　D. 孩子表示法存储的树