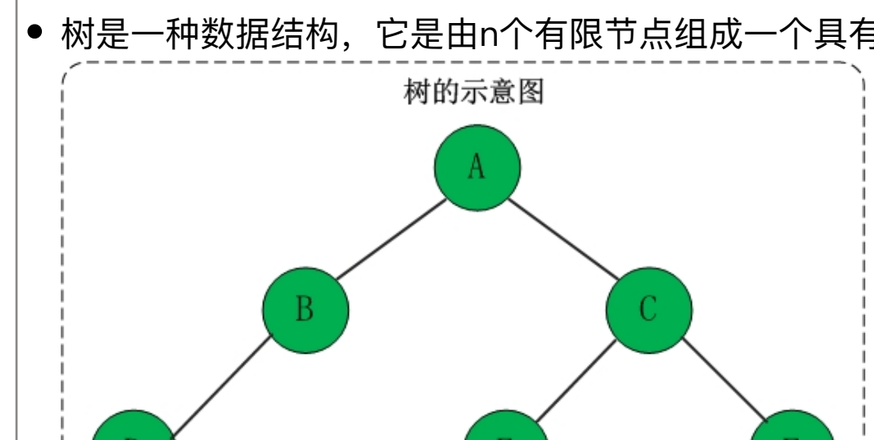
第五章 树与二叉树

2022年10月2日 星期日 19:44

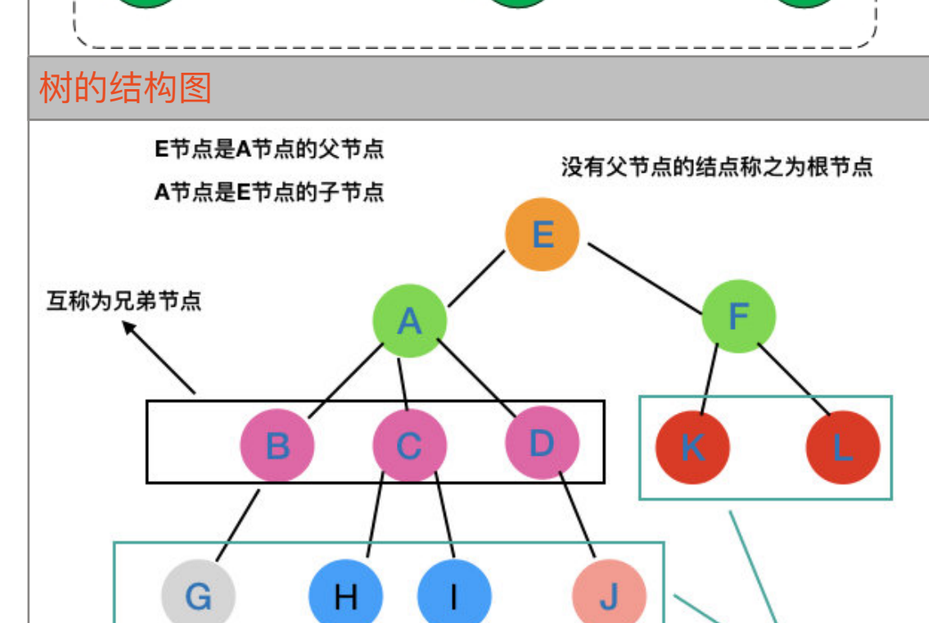
树

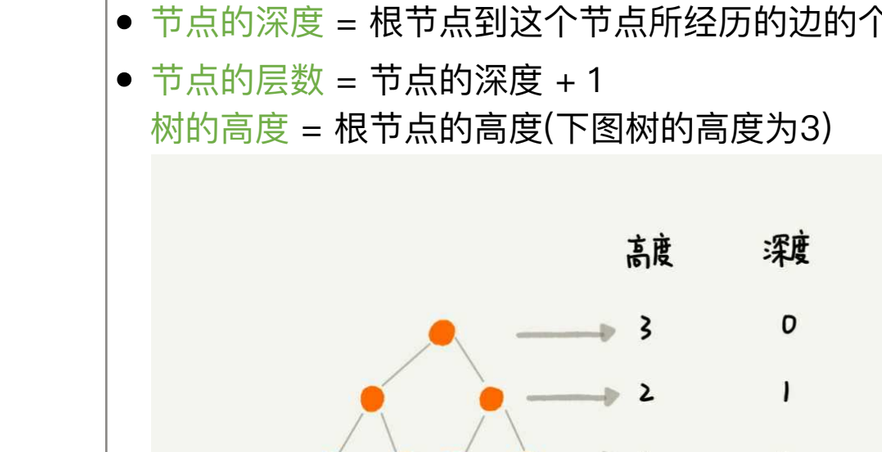
树的基本概念

树的定义	树的特点
• 树是一种数据结构，它是由n个有限节点组成一个具有层次关系的集合	• 每个节点有零个或多个子节点 • 没有父节点的节点称为根节点 • 每一个非根节点有且只有一个父节点 • 除了根节点外，每个子节点可以分为多个不相交的子树



树的结构图	树的高度，深度，层
（见下图）	• 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(下图树的高度为3)





树的基本术语		树的性质
节点的度	• 结点拥有的子树的数目	• 树的节点数 = 所有节点的度数之和 + 1
叶子	• 度为零的结点	• 度为m的树中第i层上最多有 m^{i-1} 个结点
分支结点	• 度不为零的结点	• 已知度m和高度h
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成	

求树的最少节点数	让1~h−1层节点数都为1，最后一层节点数为m
求树的最多节点数	让树成为满m叉树

树的存储结构/树的表示

双亲表示法	孩子表示法	孩子兄弟表示法
• 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	• 将每个节点的孩子节点用单链表连接	• 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】

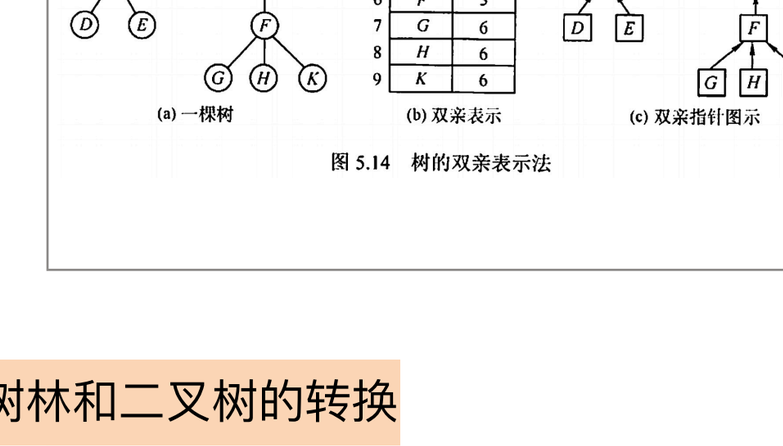
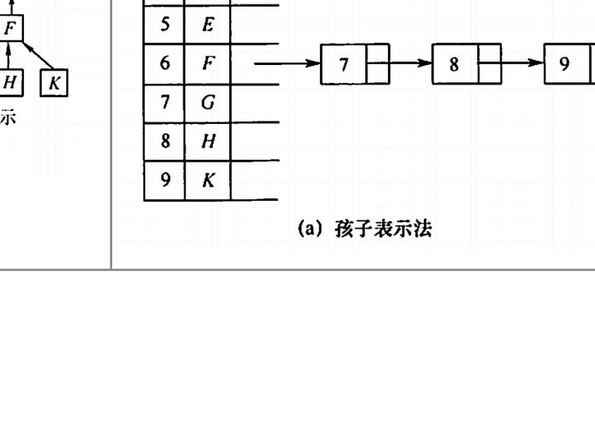
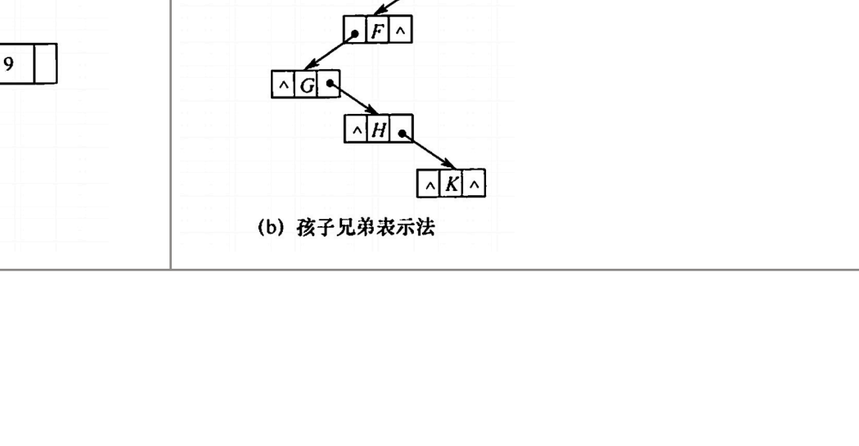


图 5.14 树的双亲表示法


(a) 孩子表示法


(b) 孩子兄弟表示法

树林和二叉树的转换

树与二叉树	树,树林与二叉树
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	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

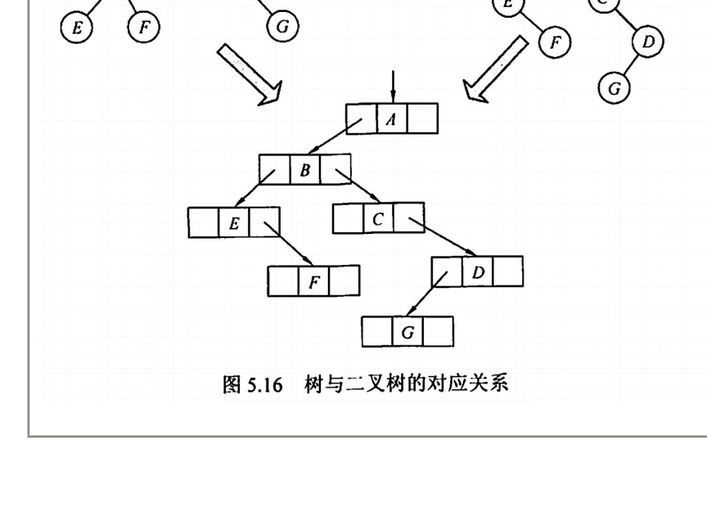


图 5.16 树与二叉树的对应关系

图 5.17 森林与二叉树的对应关系

树和森林的遍历对应关系

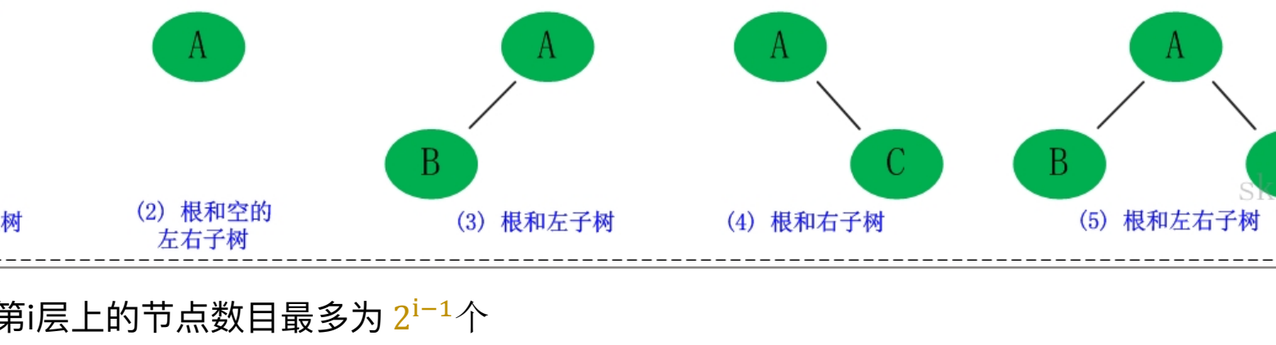
树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

二叉树

二叉树的基本概念

二叉树的定义

- 二叉树是每个节点最多有两个子树的树结构
- 它有五种基本形态【二叉树可以是空集】【根可以有空的左子树或右子树】【左、右子树皆为空】

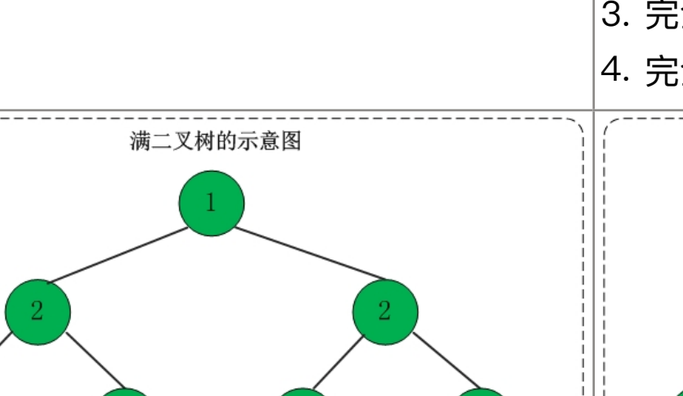


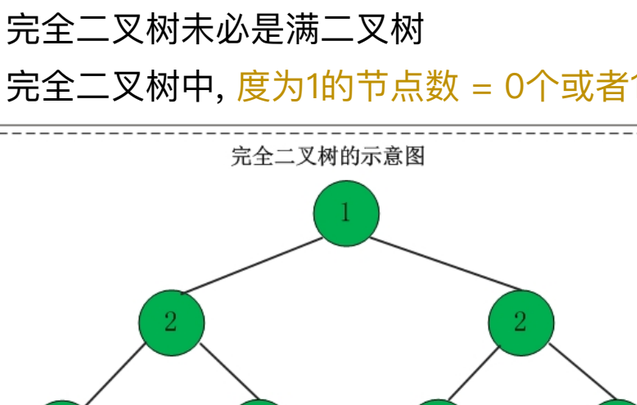
二叉树的性质

- 二叉树第i层上的节点数目最多为 2^{i-1} 个
- 深度为k的二叉树最多有 $2^k - 1$ 个节点（满二叉树）
- 包含n个节点的二叉树的高度至少为 $\log_2(n+1)$
- 树的节点数 = 所有节点的度数之和 + 1
- 在任意一颗二叉树中，若终端节点的个数为 n_0, 度为2的节点数为 n_2, 则 $n_0 = n_2 + 1$

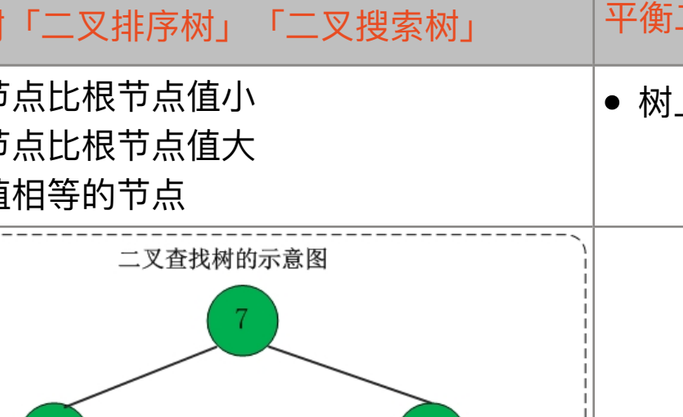
特殊的二叉树

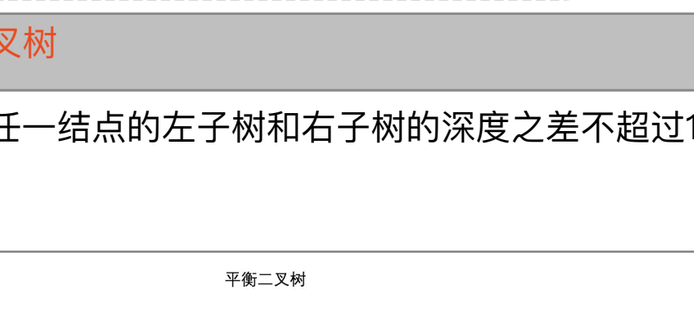
满二叉树	完全二叉树
• 高度为h，并且由 $2^h - 1$ 个结点的二叉树	1. 叶子结点只能出现在最下层和次下层, 最下层的叶子结点集中在树的左部 2. 一棵满二叉树必定是一棵完全二叉树 3. 完全二叉树未必是满二叉树 4. 完全二叉树中, 度为1的节点数 = 0个或者1个【计算时可以用这个快速计算, 配合 $n_0 = n_2 + 1$】



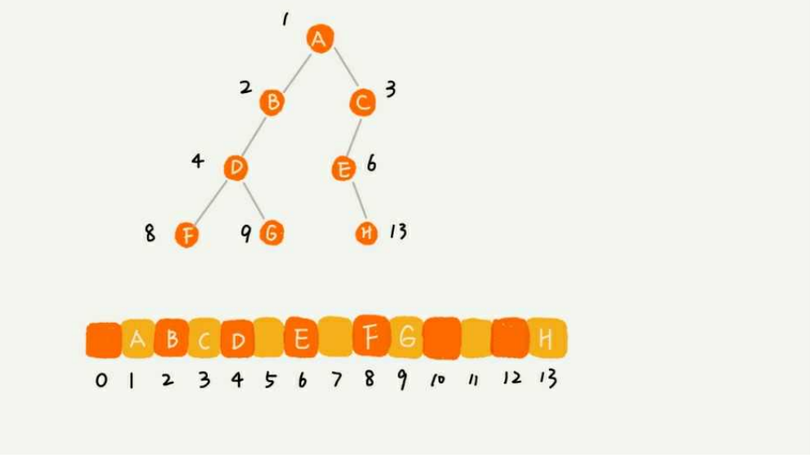
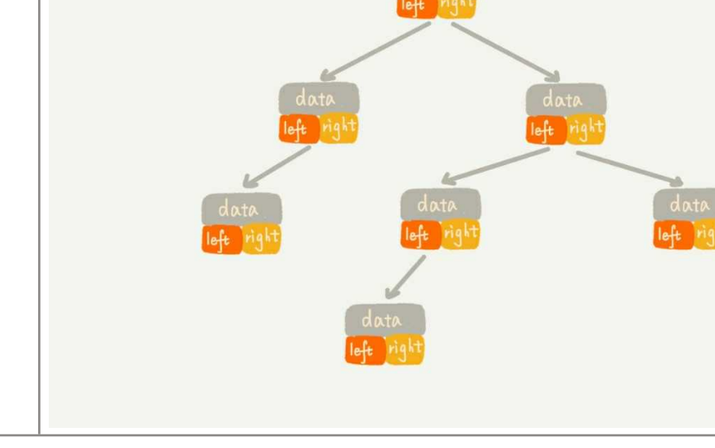


二叉查找树「二叉排序树」「二叉搜索树」	平衡二叉树
1. 左子树节点比根节点值小 2. 右子树节点比根节点值大 3. 没有键值相等的节点	• 树上任一结点的左子树和右子树的深度之差不超过1





二叉树的存储结构

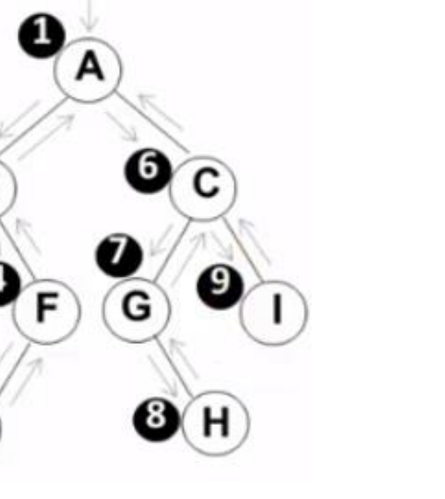
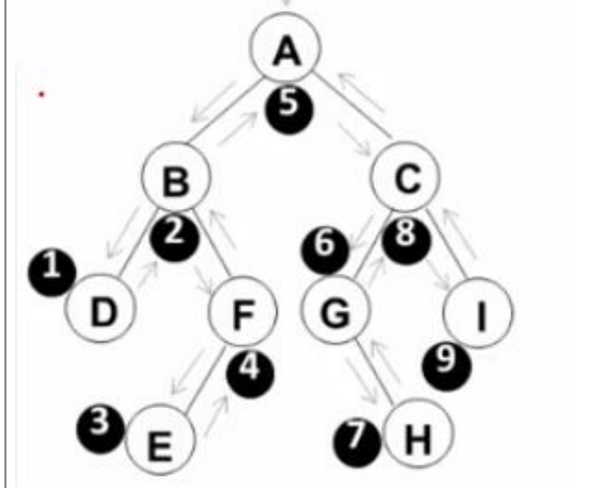
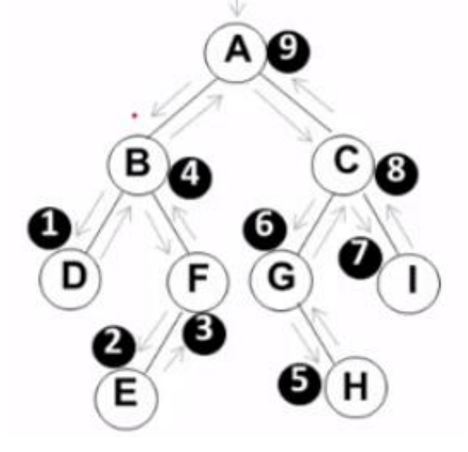
	顺序存储结构	链式存储结构
定义	• 一般用数组存二叉树的节点 • 只要知道根节点的存储位置，就可以通过下标计算，把整棵树串起来	• 只要知道根节点，就可以通过左右子节点的指针把整棵二叉树串起来
特点	• 节点x存储在下标为i的位置 • 该节点的左节点存储在下标为2i的位置 • 该节点的右节点存储在下标为2i+1的位置 • 下标i/2的位置，存储的就是该节点的父节点 • 空间利用率不高，容易造成空间浪费	• 二叉链表中至少包含3个域 • 「数据域data」「左指针域lchild」「右指针域rchild」
适用于	1. 完全二叉树 2. 满二叉树	1. 二叉树
结构图		

二叉树的实现

二叉树的代码表示

```c
typedef struct TreeNode *BinTree;
struct TreeNode
{
    int Data;       // 存值
    BinTree Left;   // 左儿子结点
    BinTree Right;  // 右儿子结点
};
```

二叉树的三种遍历方法

	前序	中序	后序	层序遍历
过程	根节点--->左节点---->右节点	左节点--->根节点---->右节点	左节点--->右节点---->根节点	从上至下，从左至右访问所有结点
解释图				基于队列实现过程 1.根结点入队 2.从队列中取出一个元素 3.访问该元素所指结点 4.若该元素所指结点的左孩子结点非空，左孩子结点入队 5.若该元素所指结点的右孩子结点非空，右孩子结点入队 6.循环2-4, 直到队列中为空

递归代码

前序：

```c
void PreOrderTraversal(BinTree BT)
{
    if (BT)
    {
        printf("%d", BT->Data);        // 打印根
        PreOrderTraversal(BT->Left);   // 进入左子树
        PreOrderTraversal(BT->Right);  // 进入右子树
    }
}
```

中序：

```c
void InOrderTraversal(BinTree BT)
{
    if (BT)
    {
        InOrderTraversal(BT->Left);   // 进入左子树
        printf("%d", BT->Data);       // 打印根
        InOrderTraversal(BT->Right);  // 进入右子树
    }
}
```

后序：

```c
void PostOrderTraversal(BinTree BT)
{
    if (BT)
    {
        PostOrderTraversal(BT->Left);   // 进入左子树
        PostOrderTraversal(BT->Right);  // 进入右子树
        printf("%d", BT->Data);         // 打印根
    }
}
```

层序遍历：無

非递归代码

前序：

```c
void PreOrderTraversal(BinTree BT)
{
    BinTree T = BT;
    Stack S = CreateStack(); // 创建并初始化堆栈 S
    while (T || !IsEmpty(S))
    { // 当树不为空或堆栈不空
        while (T)
        {
            Push(S, T);              // 压栈，第一次遇到该结点
            printf("%d", T->Data);   // 访问结点
            T = T->Left;             // 遍历左子树
        }
        if (!IsEmpty(S))
        {                  // 当堆栈不空
            T = Pop(S);    // 出栈，第二次遇到该结点
            T = T->Right;  // 访问右结点
        }
    }
}
```

中序：

```c
void InOrderTraversal(BinTree BT)
{
    BinTree T = BT;
    Stack S = CreateStack(); // 创建并初始化堆栈S
    while (T || !IsEmpty(S))
    { // 当树不为空或堆栈不空
        while (T)
        {
            Push(S, T);    // 压栈
            T = T->Left;   // 遍历左子树
        }
        if (!IsEmpty(S))
        {                            // 当堆栈不空
            T = Pop(S);              // 出栈
            printf("%d", T->Data);   // 访问结点
            T = T->Right;            // 访问右结点
        }
    }
}
```

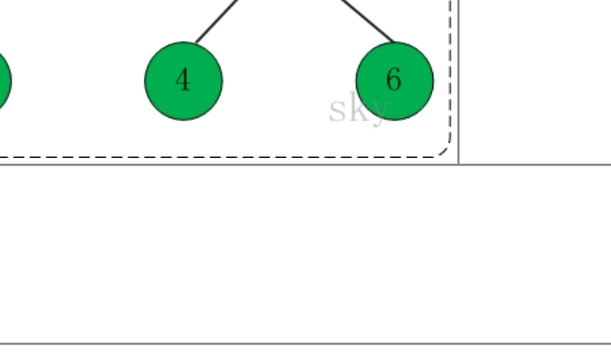
后序：

```c
void PostOrderTraversal(BinTree BT)
{
    BinTree T = BT;
    Stack S = CreateStack(); // 创建并初始化堆栈 S
    vector<BinTree> v;       // 创建存储树结点的动态数组
    Push(S, T);
    while (!IsEmpty(S))
    { // 当树不为空或堆栈不空
        T = Pop(S);
        v.push_back(T); // 出栈元素进数组
        if (T->Left)
            Push(S, T->Left);
        if (T->Right)
            Push(S, T->Right);
    }
    reverse(v.begin(), v.end());        // 逆转
    for (int i = 0; i < v.size(); i++) // 输出数组元素
        printf("%d", v[i]->Data);
}
```

层序遍历：

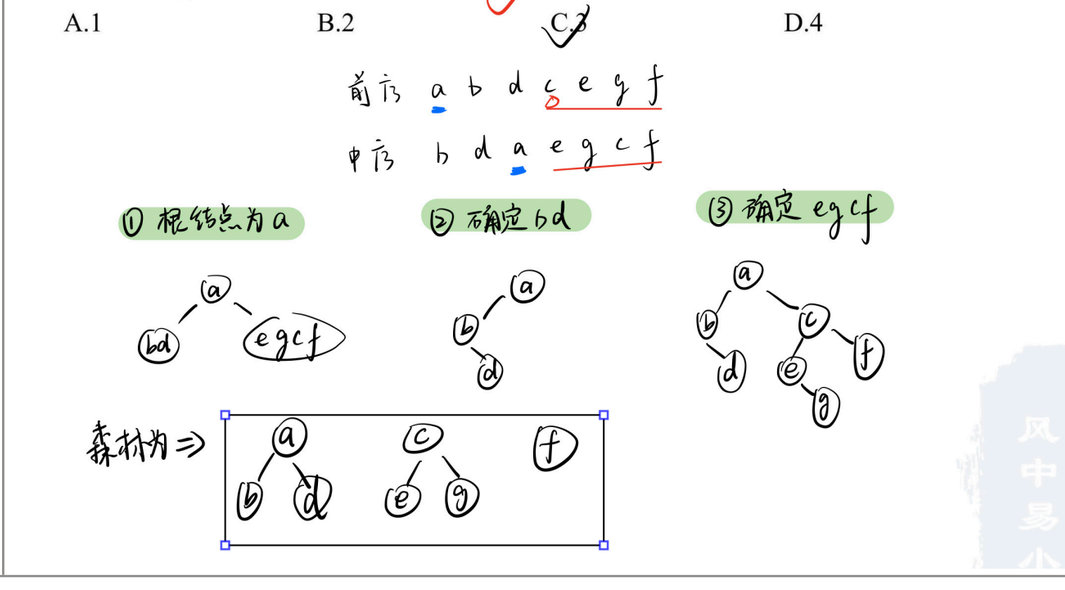
```c
void LevelOrderTraversal(BinTree BT)
{
    queue<BinTree> q; // 创建队列
    BinTree T;
    if (!BT)
        return;
    q.push(BT); // BT 入队
    while (!q.empty())
    {
        T = q.front(); // 访问队首元素
        q.pop();       // 出队
        printf("%d", T->Data);
        if (T->Left)           // 如果存在左儿子结点
            q.push(T->Left); // 入队
        if (T->Right)
            q.push(T->Right);
    }
}
```

三种遍历实例



(01) 前序遍历结果： 3 1 2 5 4 6
(02) 中序遍历结果： 1 2 3 4 5 6
(03) 后序遍历结果： 2 1 4 6 5 3

常考的结论

1、不能唯一确定一颗二叉树的是：先序序列和后序序列	
2、先序遍历第一个节点为根节点；后序遍历最后一个节点为根节点	
3、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	
4、前序序列与后序序列刚好相反的时候，二叉树的高度 = 节点数（即每层只有一个节点）	
5、后序遍历可以找到m到n直接的路径（其中m是n的祖先）	
6、根据两个序列确定二叉树的方法	

线索二叉树

基本概念和参考点

- 对一棵二叉树中所有节点的空指针域按照某种遍历方式加线索的过程叫作线索化
- 线索二叉树是一种物理结构
- 引入线索的目的是加快对二叉树的遍历
- n个节点的线索二叉树上含有线索数量为n+1个
- 线索二叉树就是利用二叉树的n+1个空指针来存放节点的前驱和后继信息的
- 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

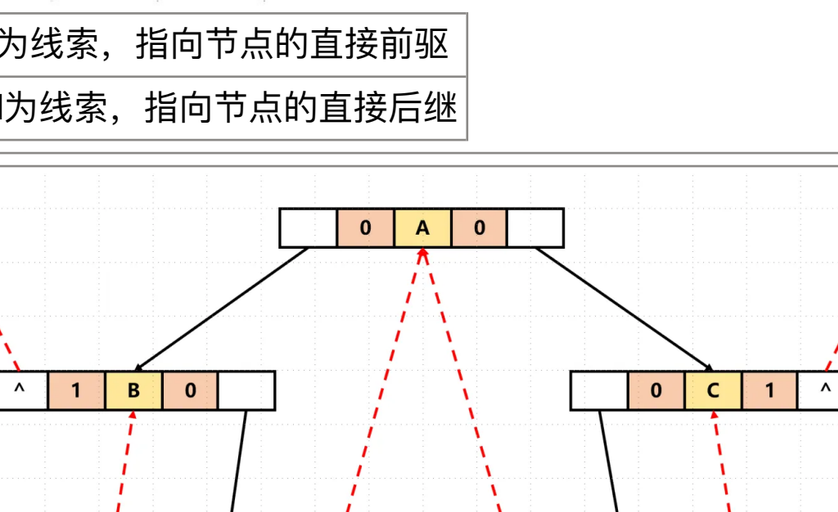
结点结构

lchild	ltag	data	rtag	rchild
指针域	标识域	数据域	标识域	指针域

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

中序线索化的过程

1. 对二叉树进行中序遍历
2. 节点右子节点为空的指针域指向它的后继节点
3. 节点左子节点为空的指针域指向它的前驱节点

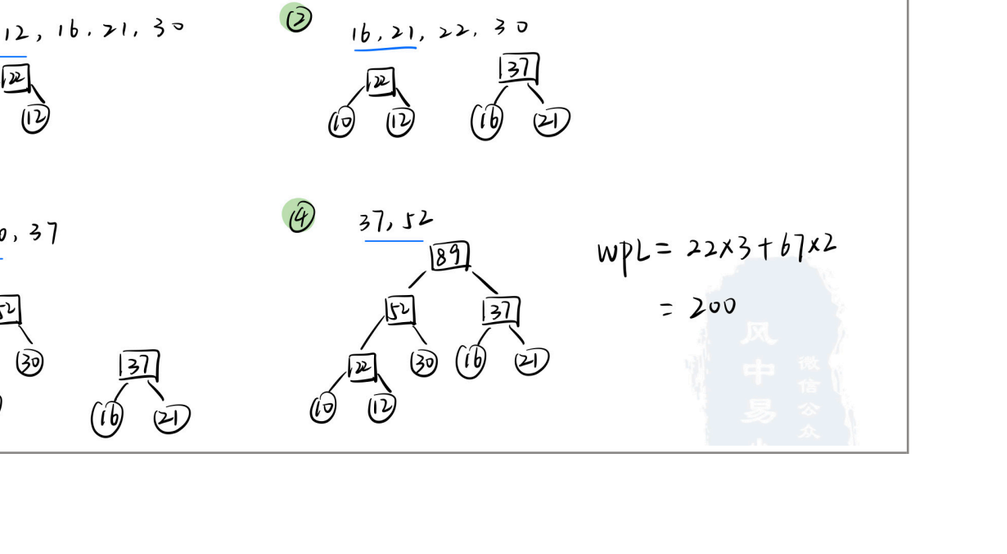


前序和后序线索化只需要把遍历方法改成前序和后序即可

树和二叉树的应用

哈夫曼树/最优二叉树

定义	特点
• 树的带权路径长度最小的二叉树 • WPL=路径长度 * 结点权值	• 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

构造	例题
• 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	

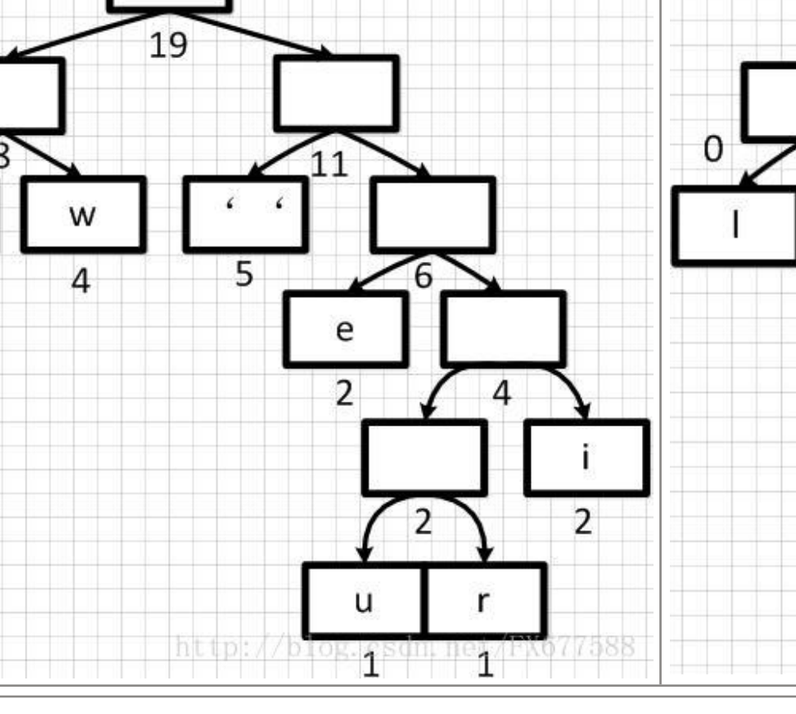
哈夫曼编码

相关概念

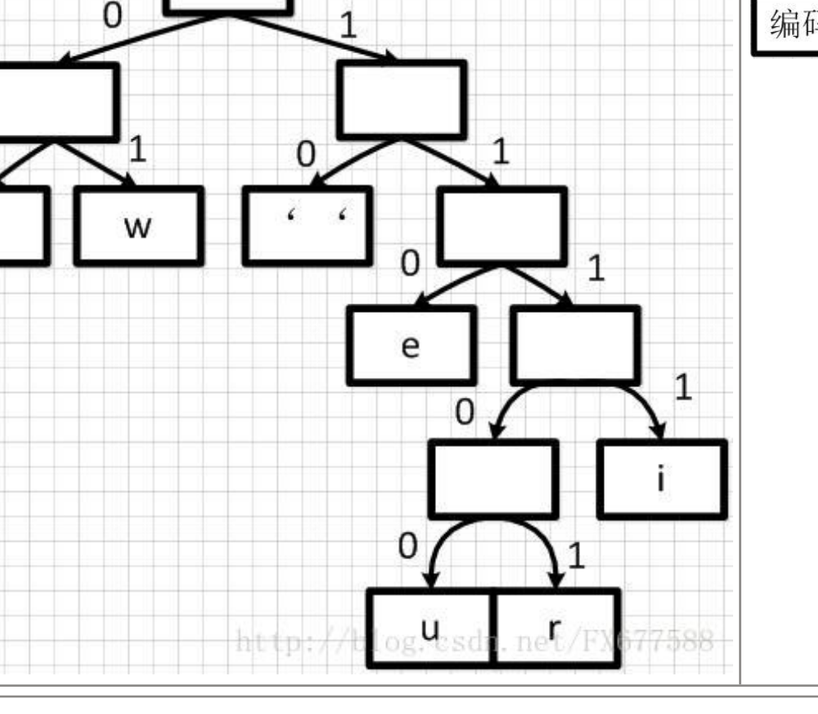
- 前缀编码：没有一个编码是另一个编码的前缀的编码
- 如右2图'l'的编码为00，'u'的编码为11100等等
- 出现频率越高的字符越会在上层，这样它的编码越短
- 出现频率越低的字符越会在下层，编码越短
- 经过这样的设计，最终整个文本存储空间才会最大化的缩减
- 有了右3图的编码表之后
- "we will we will r u"这句重新进行编码就可以得到很大的压缩
- 编码表示为
- 01 110 10 01 1111 00 00 10 01 110 10 01 1111 00 00 10 11101 10 1110
- 最终我们只需50位内存，比原ASCII码表示节约了2/3空间

由哈夫曼树构造哈夫曼编码的过程【左0右1】

哈夫曼树



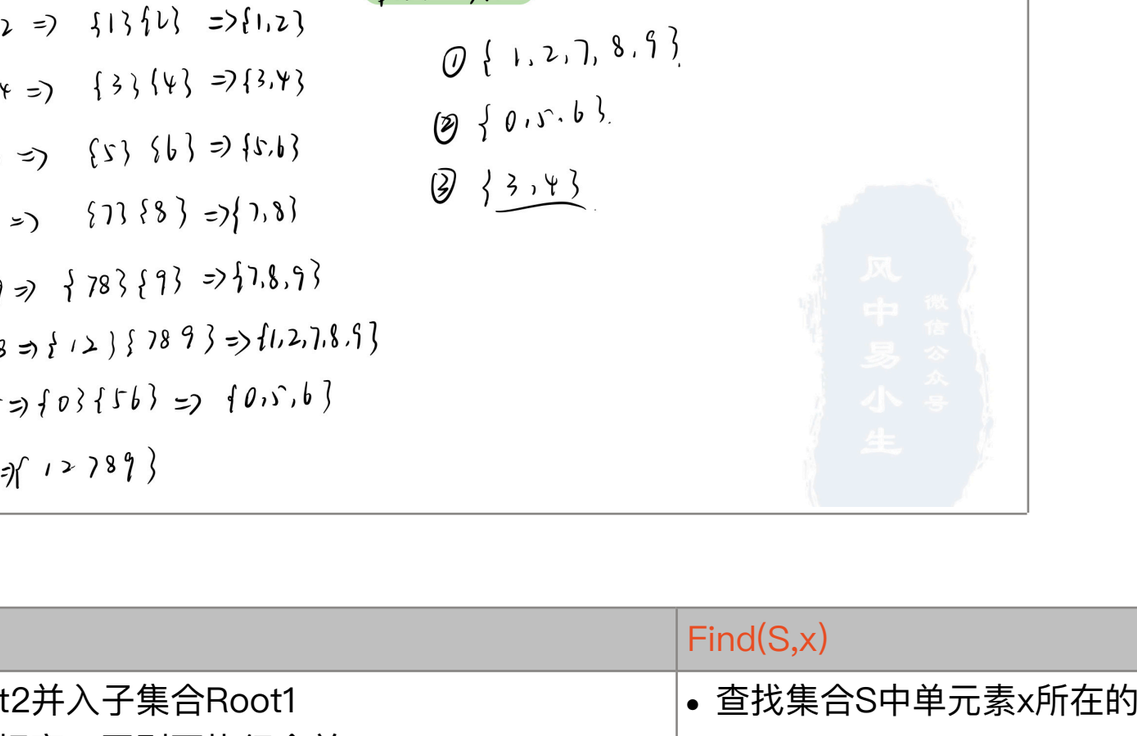
对应的哈夫曼编码



最终的编码表

字符	l	w	' '	e	i	u	r
编码	00	01	10	110	1111	11100	11101

并查集

定义	例题
• 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	

Initial(S)	Union(S,Root1,Root2)	Find(S,x)
• 将集合S中的每个元素都初始化为只有一个单元素的子集合	• 把集合S中的子集合Root2并入子集合Root1 • 要求Root1和Root2互不相交，否则不执行合并	• 查找集合S中单元素x所在的子集合，并返回该子集合的根结点

```c
void Initial(int S[])
{
    for (int i = 0; i < size; i++)
        S[i] = -1;
}
```

```c
void Union(int S[], int Root1, int Root2)
{
    S[Root2] = Root1;
}
```

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

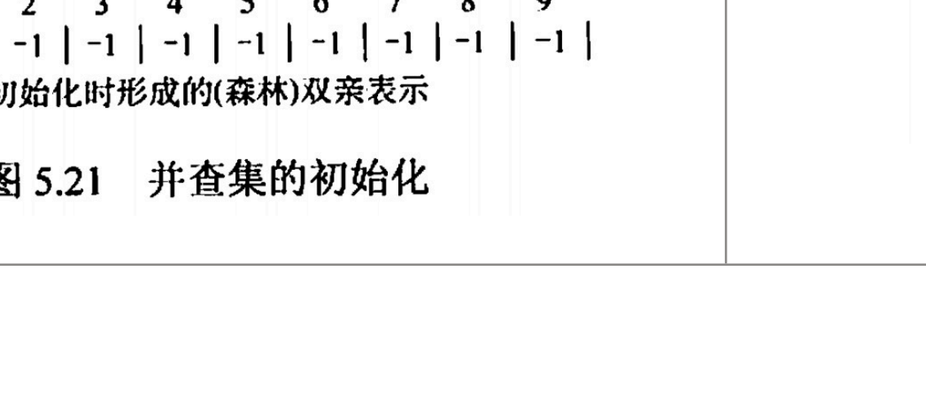


图 5.21 并查集的初始化

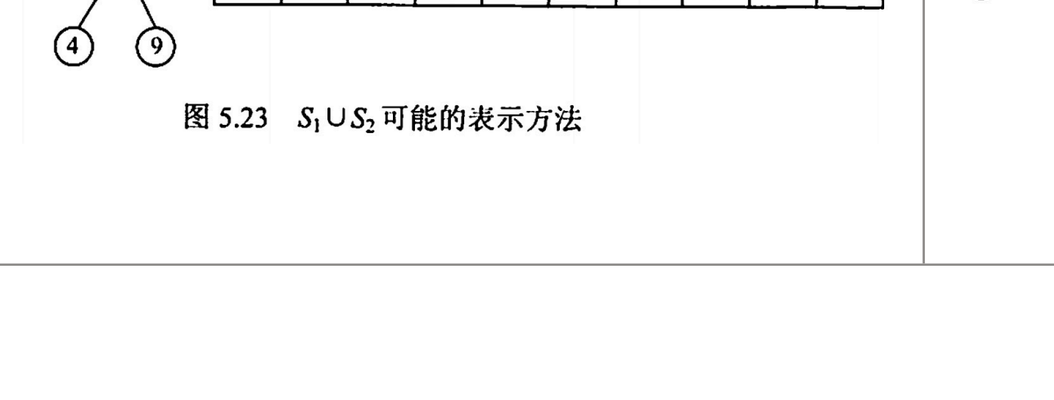


图 5.23 $S_1 \cup S_2$ 可能的表示方法

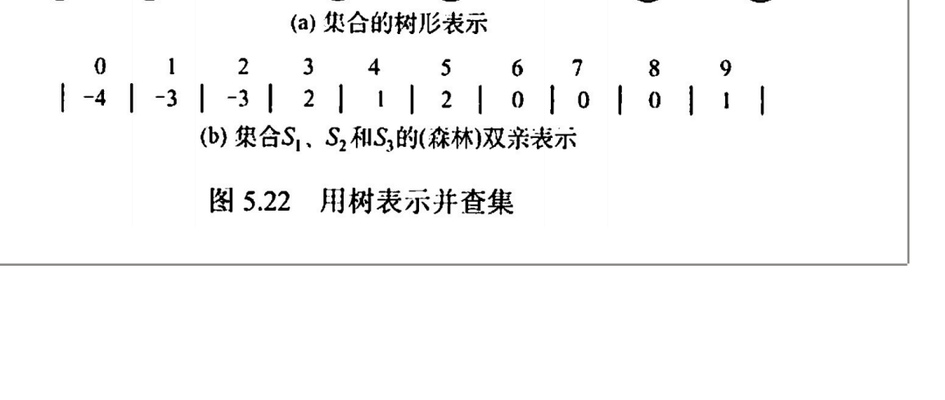


图 5.22 用树表示并查集