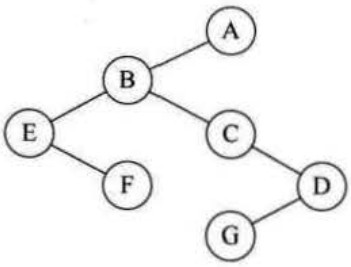
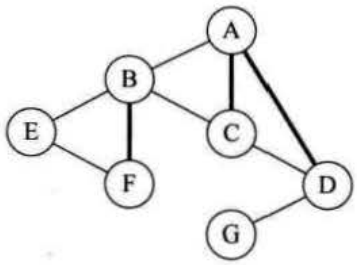
右孩子结点都作为此结点的孩子。将该结点与这些右孩子结点用线连接起来。

2．去线。删除原二叉树中所有结点与其右孩子结点的连线。

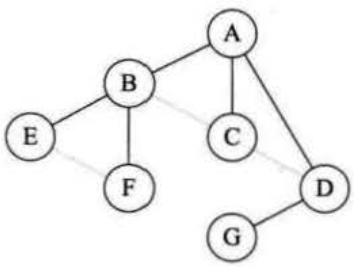
3．层次调整。使之结构层次分明。



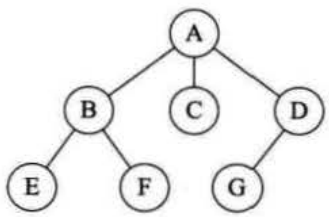
二叉树



步骤1：加线



步骤2：去线



步骤3：层次调整

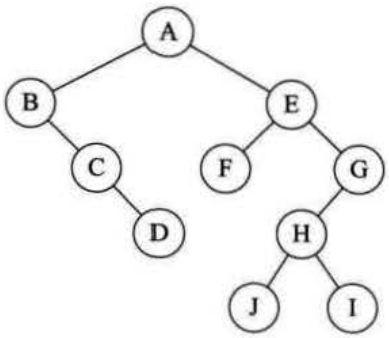
图 6-11-4

6.11.4 二叉树转换为森林

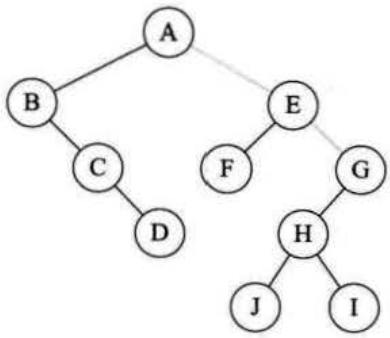
判断一棵二叉树能够转换成一棵树还是森林，标准很简单，那就是只要看这棵二叉树的根结点有没有右孩子，有就是森林，没有就是一棵树。那么如果是转换成森林，步骤如下：

1．从根结点开始，若右孩子存在，则把与右孩子结点的连线删除，再查看分离后的二叉树，若右孩子存在，则连线删除……，直到所有右孩子连线都删除为止，得到分离的二叉树。

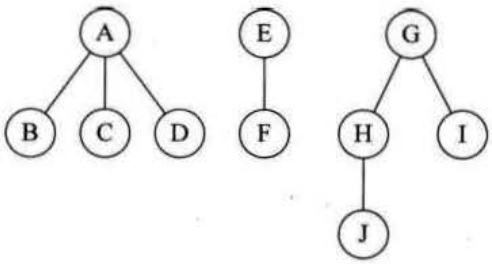
2．再将每棵分离后的二叉树转换为树即可。



二叉树



步骤1：寻找右孩子去线



步骤2：将分离的二叉树转换成树

图 6-11-5

6.11.5 树与森林的遍历

最后我们再谈一谈关于树和森林的遍历问题。

树的遍历分为两种方式。

1．一种是先根遍历树，即先访问树的根结点，然后依次先根遍历根的每棵子树。

2．另一种是后根遍历，即先依次后根遍历每棵子树，然后再访问根结点。比如图 6-11-4 中最右侧的树，它的先根遍历序列为 ABEFCDG，后根遍历序列为 EFBCGDA。

森林的遍历也分为两种方式：

1．**前序遍历**：先访问森林中第一棵树的根结点，然后再依次先根遍历根的每棵子树，再依次用同样方式遍历除去第一棵树的剩余树构成的森林。比如图 6-11-5 右侧三棵树的森林，前序遍历序列的结果就是 ABCDEFGHJI。

2．**后序遍历**：是先访问森林中第一棵树，后根遍历的方式遍历每棵子树，然后再访问根结点，再依次同样方式遍历除去第一棵树的剩余树构成的森林。比如图

6-11-5 右侧三棵树的森林，后序遍历序列的结果就是 BCDAFEJHIG。

可如果我们对图 6-11-4 的左侧二叉树进行分析就会发现，森林的前序遍历和二叉树的前序遍历结果相同，森林的后序遍历和二叉树的中序遍历结果相同。

这也就告诉我们，当以二叉链表作树的存储结构时，树的先根遍历和后根遍历完全可以借用二叉树的前序遍历和中序遍历的算法来实现。这其实也就证实，我们找到了对树和森林这种复杂问题的简单解决办法。

6.12 赫夫曼树及其应用

6.12.1 赫夫曼树

“喂，兄弟，最近无聊透顶了，有没有什么书可看？”

“我这有《三国演义》的电子书，你要不要？”

“‘既生瑜，何生亮。’《三国演义》好呀，你邮件发给我！”

“OK！文件 1M 多大小，好像大了点。我打个包，稍等……哈哈，少了一半，压缩效果不错呀。”

“太棒了，快点传给我吧。”

图 6-12-1

这是我们生活中常见的对白。现在我们都是讲究效率的社会，什么都要求速度，在不能出错的情况下，做任何事情都讲究越快越好。在计算机和互联网技术中，文本压缩就是一个非常重要的技术。玩电脑的人几乎都会应用压缩和解压缩软件来处理文档。因为它除了可以减少文档在磁盘上的空间外，还有重要的一点，就是我们可以在网络上以压缩的形式传输大量数据，使得保存和传递都更加高效。

那么压缩而不出错是如何做到的呢？简单说，就是把我们要压缩的文本进行重新编码，以减少不必要的空间。尽管现在最新技术在编码上已经很好很强大，但这一切都来自于曾经的技术积累，我们今天就来介绍一下最基本的压缩编码方法——赫夫曼编码。

在介绍赫夫曼编码前，我们必须得介绍赫夫曼树，而介绍赫夫曼树，我们不得不提这样一个人，美国数学家赫夫曼（David Huffman），也有的翻译为哈夫曼。他在1952年发明了赫夫曼编码，为了纪念他的成就，于是就把他在编码中用到的特殊的二叉树称之为赫夫曼树，他的编码方法称为赫夫曼编码。也就是说，我们现在介绍的知识全都来自于近60年前这位伟大科学家的研究成果，而我们平时所用的压缩和解压缩技术也都是基于赫夫曼的研究之上发展而来，我们应该要记住他。

什么叫做赫夫曼树呢？我们先来看一个例子。

过去我们小学、中学一般考试都是用百分制来表示学科成绩的。这带来了一个弊端，就是很容易让学生、家长，甚至老师自己都以分取人，让分数代表了一切。有时想想也对，90分和95分也许就只是一道题目对错的差距，但却让两个孩子可能受到完全不同的待遇，这并不公平。于是在如今提倡素质教育的背景下，我们很多的学科，特别是小学的学科成绩都改作了优秀、良好、中等、及格和不及格这样模糊的词语，不再通报具体的分数。

不过对于老师来讲，他在对试卷评分的时候，显然不能凭感觉给优良或及格不及格等成绩，因此一般都还是按照百分制算出每个学生的成绩后，再根据统一的标准换算得出五级分制的成绩。比如下面的代码就实现了这样的转换。

```
if (a<60)
    b="不及格";
else if (a<70)
    b="及格";
else if (a<80)
    b="中等";
else if (a<90)
    b="良好";
else
    b="优秀";
```

图6-12-2 粗略看没什么问题，可是通常都认为，一张好的考卷应该是让学生成绩大部分处于中等或良好的范围，优秀和不及格都应该较少才对。而上面这样的程序，就使得所有的成绩都需要先判断是否及格，再逐级而上得到结果。输入量很大的时候，其实算法是有效率问题的。

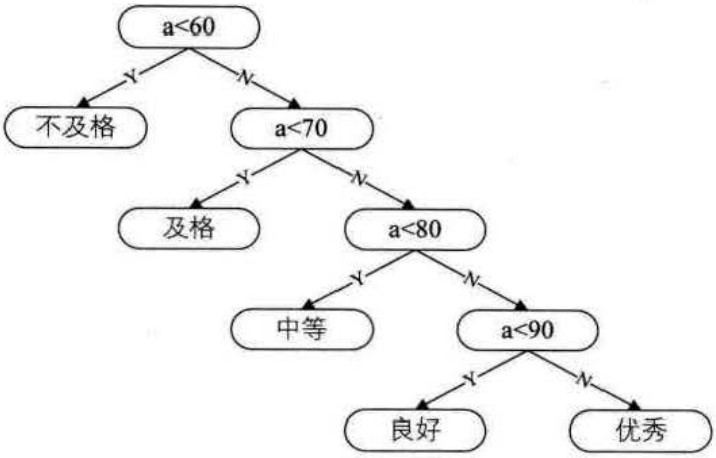


图 6-12-2

如果在实际的学习生活中，学生的成绩在 5 个等级上的分布规律如表 6-12-1 所示。

表 6-12-1

分数	0～59	60～69	70～79	80～89	90～100
所占比例	5%	15%	40%	30%	10%

那么 70 分以上大约占总数 80%的成绩都需要经过 3 次以上的判断才可以得到结果，这显然不合理。

有没有好一些的办法，仔细观察发现，中等成绩（70～79 分之间）比例最高，其次是良好成绩，不及格的所占比例最少。我们把图 6-12-2 这棵二叉树重新进行分配。改成如图 6-12-3 的做法试试看。

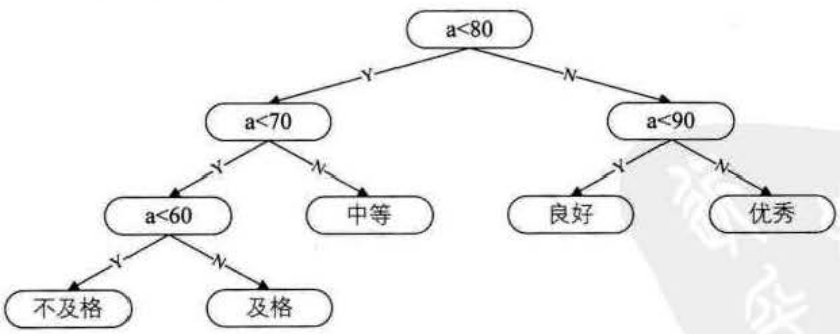
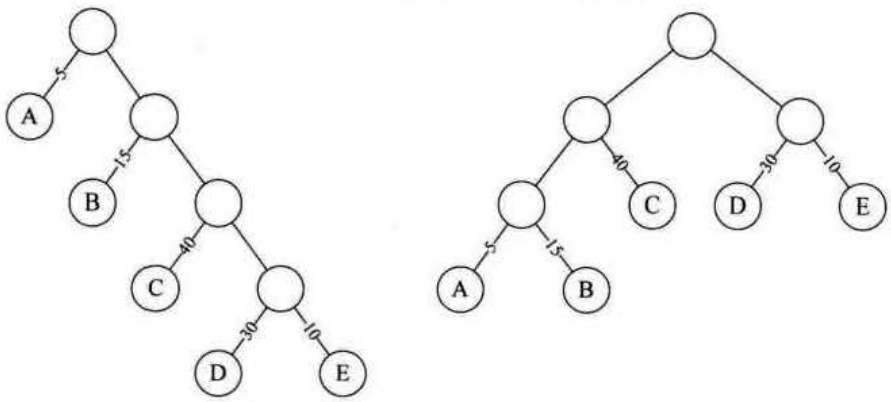


图 6-12-3

从图中感觉，应该效率要高一些了，到底高多少呢。这样的二叉树又是如何设计出来的呢？我们来看看赫夫曼大叔是如何说的吧。

6.12.2 赫夫曼树定义与原理

我们先把这两棵二叉树简化成叶子结点带权的二叉树，如图 6-12-4 所示。其中 A 表示不及格、B 表示及格、C 表示中等、D 表示良好、E 表示优秀。每个叶子的分支线上的数字就是刚才我们提到的五级分制的成绩所占比例数。



二叉树a　　二叉树b

图 6-12-4

赫夫曼大叔说，**从树中一个结点到另一个结点之间的分支构成两个结点之间的路径，路径上的分支数目称做路径长度。**图 6-12-4 的二叉树 a 中，根结点到结点 D 的路径长度就为 4，二叉树 b 中根结点到结点 D 的路径长度为 2。**树的路径长度就是从树根到每一结点的路径长度之和。**二叉树 a 的树路径长度就为 1+1+2+2+3+3+4+4=20。二叉树 b 的树路径长度就为 1+2+3+3+2+1+2+2=16。

如果考虑到带权的结点，结点的带权的路径长度为从该结点到树根之间的路径长度与结点上权的乘积。树的带权路径长度为树中所有叶子结点的带权路径长度之和。假设有 n 个权值$\{w_1,w_2,\cdots,w_n\}$，构造一棵有 n 个叶子结点的二叉树，每个叶子结点带权 w_k，每个叶子的路径长度为 lk，我们通常记作，则其中**带权路径长度 WPL 最小的二叉树称做赫夫曼树。**也有不少书中也称为最优二叉树，我个人觉得为了纪念做出巨大贡献的科学家，既然用他们的名字命名，就应该要坚持用他们的名字称呼，哪怕“最优”更能体现这棵树的品质也应该只作为别名。

有了赫夫曼对带权路径长度的定义，我们来计算一下图 6-12-4 这两棵树的 WPL 值。

二叉树 a 的 WPL=5×1+15×2+40×3+30×4+10×4=315

注意：这里 5 是 A 结点的权，1 是 A 结点的路径长度，其他同理。

二叉树 b 的 WPL=5×3+15×3+40×2+30×2+10×2=220

这样的结果意味着什么呢？如果我们现在有 10000 个学生的百分制成绩需要计算五级分制成绩，用二叉树 a 的判断方法，需要做 31500 次比较，而二叉树 b 的判断方法，只需要 22000 次比较，差不多少了三分之一量，在性能上提高不是一点点。

那么现在的问题就是，图 6-12-4 的二叉树 b 这样的树是如何构造出来的，这样的二叉树是不是就是最优的赫夫曼树呢？别急，赫夫曼大叔给了我们解决的办法。

1. 先把有权值的叶子结点按照从小到大的顺序排列成一个有序序列，即：A5，E10，B15，D30，C40。
2. 取头两个最小权值的结点作为一个新节点 N_1 的两个子结点，注意相对较小的是左孩子，这里就是 A 为 N_1 的左孩子，E 为 N_1 的右孩子，如图 6-12-5 所示。新结点的权值为两个叶子权值的和 5+10=15。
3. 将 N_1 替换 A 与 E，插入有序序列中，保持从小到大排列。即：$N_1$15，B15，D30，C40。
4. 重复步骤 2。将 N_1 与 B 作为一个新节点 N_2 的两个子结点。如图 6-12-6 所示。N_2 的权值=15+15=30。

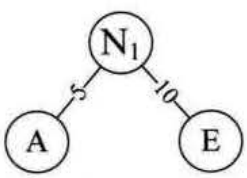


图 6-12-5

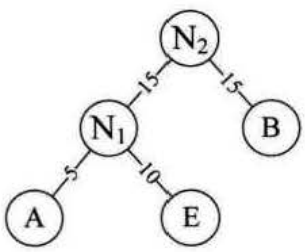


图 6-12-6

5. 将 N_2 替换 N_1 与 B，插入有序序列中，保持从小到大排列。即：$N_2$30，D30，C40。
6. 重复步骤 2。将 N_2 与 D 作为一个新节点 N_3 的两个子结点。如图 6-12-7 所示。N_3 的权值=30+30=60。
7. 将 N_3 替换 N_2 与 D，插入有序序列中，保持从小到大排列。即：C40，$N_3$60。
8. 重复步骤 2。将 C 与 N_3 作为一个新节点 T 的两个子结点，如图 6-12-8 所示。由于 T 即是根结点，完成赫夫曼树的构造。

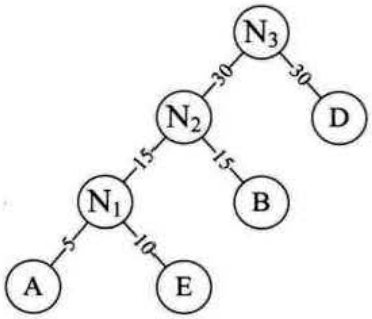


图 6-12-7

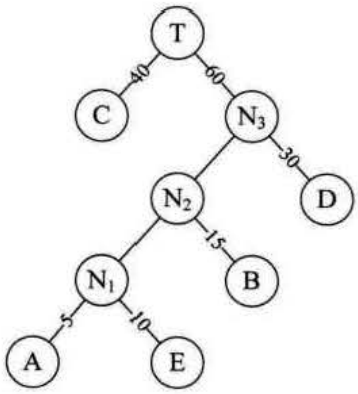


图 6-12-8

此时的图 6-12-8 二叉树的带权路径长度 WPL=40×1+30×2+15×3+10×4+5×4=205。与图 6-12-4 的二叉树 b 的 WPL 值 220 相比，还少了 15。显然此时构造出来的二叉树才是最优的赫夫曼树。

不过现实总是比理想要复杂得多，图 6-12-8 虽然是赫夫曼树，但由于每次判断都要两次比较（如根结点就是 a<80 && a>=70，两次比较才能得到 y 或 n 的结果），所以总体性能上，反而不如图 6-12-3 的二叉树性能高。当然这并不是我们要讨论的重点了。

通过刚才的步骤，我们可以得出构造赫夫曼树的赫夫曼算法描述。

1. 根据给定的 n 个权值{ $w_1,w_2,\cdots,w_n$ }构成 n 棵二叉树的集合 F={ $T_1,T_2,\cdots,T_n$ }，其中每棵二叉树 T_i 中只有一个带权为 w_i 根结点，其左右子树均为空。
2. 在 F 中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，且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。
3. 在 F 中删除这两棵树，同时将新得到的二叉树加入 F 中。
4. 重复 2 和 3 步骤，直到 F 只含一棵树为止。这棵树便是赫夫曼树。

6.12.3 赫夫曼编码

当然，赫夫曼研究这种最优树的目的不是为了我们可以转化一下成绩。他的更大目的是为了解决当年远距离通信（主要是电报）的数据传输的最优化问题。

比如我们有一段文字内容为“BADCADFEED”要网络传输给别人，显然用二进制的数字（0 和 1）来表示是很自然的想法。我们现在这段文字只有六个字母 ABCDEF，

那么我们可以用相应的二进制数据表示，如表 6-12-2 所示。

表 6-12-2

字母	A	B	C	D	E	F
二进制字符	000	001	010	011	100	101

这样真正传输的数据就是编码后的“001000011010000011101100100011”，对方接收时可以按照 3 位一分来译码。如果一篇文章很长，这样的二进制串也将非常的可怕。而且事实上，不管是英文、中文或是其他语言，字母或汉字的出现频率是不相同的，比如英语中的几个元音字母“a e i o u”，中文中的“的 了 有 在”等汉字都是频率极高。

假设六个字母的频率为 A 27，B 8，C 15，D 15，E 30，F 5，合起来正好是 100%。那就意味着，我们完全可以重新按照赫夫曼树来规划它们。

图 6-12-9 左图为构造赫夫曼树的过程的权值显示。右图为将权值左分支改为 0，右分支改为 1 后的赫夫曼树。

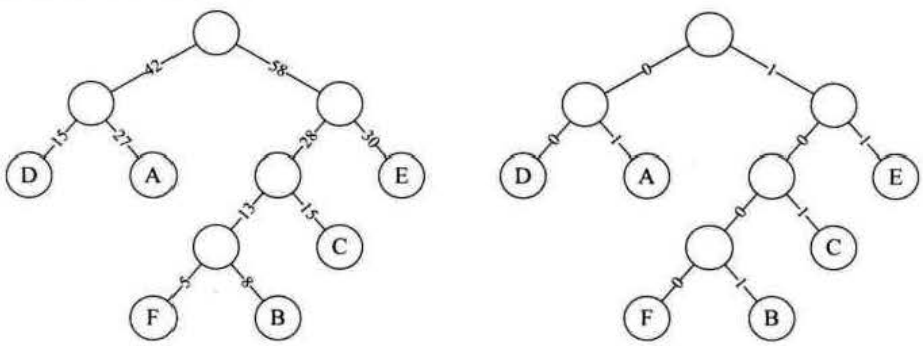


图 6-12-9

此时，我们对这六个字母用其从树根到叶子所经过路径的 0 或 1 来编码，可以得到如表 6-12-3 所示这样的定义。

表 6-12-3

字母	A	B	C	D	E	F
二进制字符	01	1001	101	00	11	1000

我们将文字内容为“BADCADFEED”再次编码，对比可以看到结果串变小了。

- 原编码二进制串：001000011010000011101100100011 （共 30 个字符）
- 新编码二进制串：100101001010100100011100 （共 25 个字符）

也就是说，我们的数据被压缩了，节约了大约 17%的存储或传输成本。随着字符

的增加和多字符权重的不同，这种压缩会更加显出其优势。

当我们接收到 1001010010101001000111100 这样压缩过的新编码时，我们应该如何把它解码出来呢?

编码中非 0 即 1，长短不等的话其实是很容易混淆的，所以**若要设计长短不等的编码，则必须是任一字符的编码都不是另一个字符的编码的前缀，这种编码称做前缀编码。**

你仔细观察就会发现，表 6-12-3 中的编码就不存在容易与 1001、1000 混淆的“10”和“100”编码。

可仅仅是这样不足以让我们去方便地解码的，因此在解码时，还是要用到赫夫曼树，即发送方和接收方必须要约定好同样的赫夫曼编码规则。

当我们接收到 1001010010101001000111100 时，由约定好的赫夫曼树可知，1001 得到第一个字母是 B，接下来 01 意味着第二个字符是 A，如图 6-12-10 所示，其余的也相应的可以得到，从而成功解码。

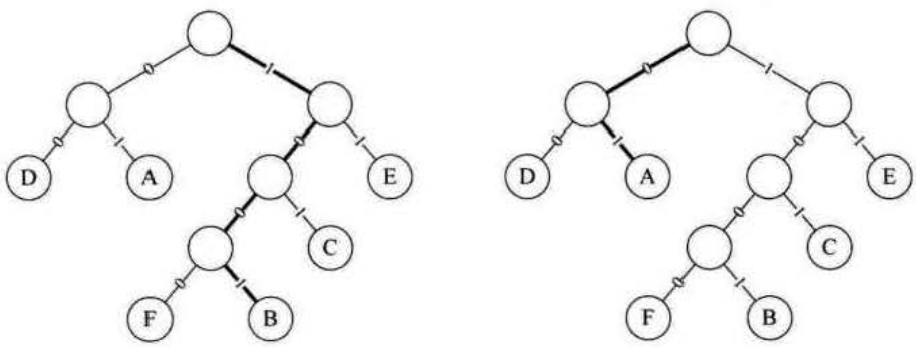


图 6-12-10

一般地，设需要编码的字符集为{ $d_1,d_2,\cdots,d_n$ }，各个字符在电文中出现的次数或频率集合为{ $w_1,w_2,\cdots,w_n$ }，以 $d_1,d_2,\cdots,d_n$ 作为叶子结点，以 $w_1,w_2,\cdots,w_n$ 作为相应叶子结点的权值来构造一棵赫夫曼树。规定赫夫曼树的左分支代表 0，右分支代表 1，则从根结点到叶子结点所经过的路径分支组成的 0 和 1 的序列便为该结点对应字符的编码，这就是赫夫曼编码。[6]

注[6]：关于赫夫曼编码详细信息，请参考《算法导论》第 16 章的 16.3 节赫夫曼编码。

6.13 总结回顾

终于到了总结的时间，这一章与前面章节相比，显得过于庞大了些，原因也就在于树的复杂性和变化丰富度是前面的线性表所不可比拟的。即使在本章之后，我们还要讲解关于树这一数据结构的相关知识，可见它的重要性。

开头我们提到了树的定义，讲到了递归在树定义中的应用。提到了如子树、结点、度、叶子、分支结点、双亲、孩子、层次、深度、森林等诸多概念，这些都是需要在理解的基础上去记忆的。

我们谈到了树的存储结构时，讲了双亲表示法、孩子表示法、孩子兄弟表示法等不同的存储结构。

并由孩子兄弟表示法引出了我们这章中最重要一种树，二叉树。

二叉树每个结点最多两棵子树，有左右之分。提到了斜树，满二叉树、完全二叉树等特殊二叉树的概念。

我们接着谈到它的各种性质，这些性质给我们研究二叉树带来了方便。

二叉树的存储结构由于其特殊性使得既可以用顺序存储结构又可以用链式存储结构表示。

遍历是二叉树最重要的一门学问，前序、中序、后序以及层序遍历都是需要熟练掌握的知识。要让自己要学会用计算机的运行思维去模拟递归的实现，可以加深我们对递归的理解。不过，并非二叉树遍历就一定要用到递归，只不过递归的实现比较优雅而已。这点需要明确。

二叉树的建立自然也是可以通过递归来实现。

研究中也发现，二叉链表有很多浪费的空指针可以利用，查找某个结点的前驱和后继为什么非要每次遍历才可以得到，这就引出了如何构造一棵线索二叉树的问题。线索二叉树给二叉树的结点查找和遍历带来了高效率。

树、森林看似复杂，其实它们都可以转化为简单的二叉树来处理，我们提供了树、森林与二叉树的互相转换的办法，这样就使得面对树和森林的数据结构时，编码实现成为了可能。

最后，我们提到了关于二叉树的一个应用，赫夫曼树和赫夫曼编码，对于带权路径的二叉树做了详尽地讲述，让你初步理解数据压缩的原理，并明白其是如何做到无

损编码和无错解码的。

6.14 结尾语

在我们这章开头，我们提到了《阿凡达》这部电影，电影中有一个情节就是人类用先进的航空武器和导弹硬是将那棵纳威人赖以生存的苍天大树给放倒了，让人很是唏嘘感慨，如图 6-14-1 所示。这尽管讲的只是一个虚构的故事，但在现实社会中，人类为了某种很短期的利益，乱砍滥伐，毁灭森林，破坏植被几乎天天都在我们居住的地球上演。

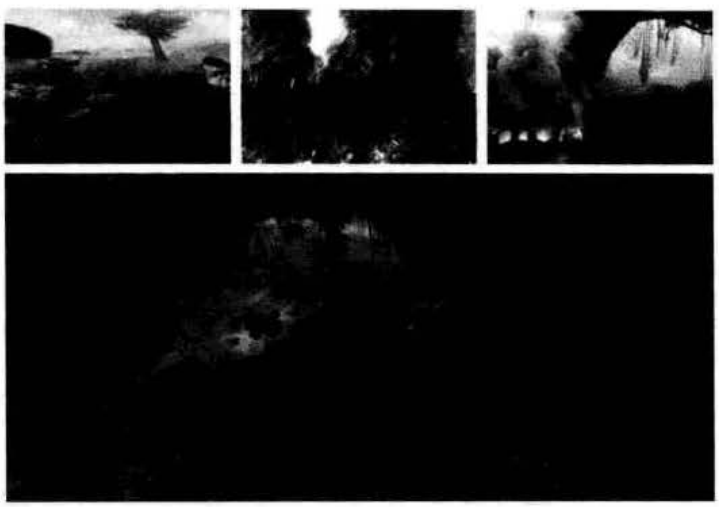


图 6-14-1

这样造成的结果就是冬天深寒、夏天酷热、超强台风、百年洪水、滚滚泥流、无尽干旱。我们地球上人类的生存环境岌岌可危。

是的，这只是一堂计算机课，讲的是无生命的数据结构——树。但在这一章的最后，我还是想呼吁一下大家。

人受伤时还会流下泪水，树受伤时，老天都不会哭泣。希望我们的未来不要仅仅有钢筋水泥建造的高楼和大厦，也要有郁郁葱葱的森林和草地，我们人类才可能与自然和谐共处。爱护树木、保护森林，让我们为生存的家园能够更加自然与美好，尽一份自己的力量。

好了，今天课就到这，下课。

第7章 图

启示

图：

图（Graph）是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：G（V,E），其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合。

7.1 开场白

旅游几乎是每个年轻人的爱好，但没有钱或没时间也是困惑年轻人不能圆梦的直接原因。如果可以用最少的资金和最少的时间周游中国甚至是世界一定是非常棒的。假设你已经有了一笔不算很丰裕的闲钱，也有了约半年的时间。此时打算全国性的旅游，你将会如何安排这次行程呢？

我们假设旅游就是逐个省市进行，省市内的风景区不去细分，例如北京玩 7 天，天津玩 3 天，四川玩 20 天这样子。你现在需要做的就是制订一个规划方案，如何才能用最少的成本将图 7-1-1 中的所有省市都玩遍，这里所谓最少的成本是指交通成本与时间成本。

如果你不善于规划，很有可能就会出现如玩好新疆后到海南，然后再冲向黑龙江这样的荒唐决策。但是即使是紧挨着省市游玩的方案也会存在很复杂的选择问题，比如游完湖北，周边有安徽、江西、湖南、重庆、陕西、河南等省市，你下一步怎么走最划算呢？

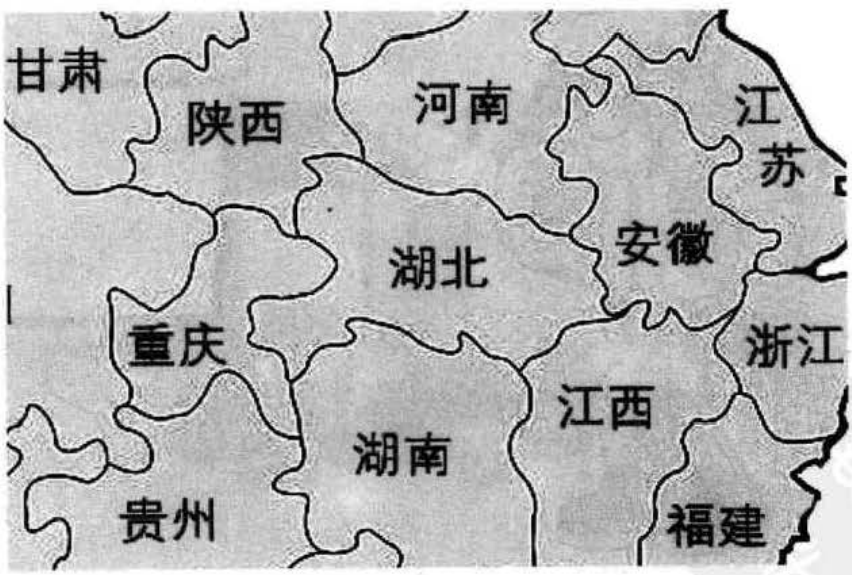


图 7-1-1

你一时解答不了这些问题是很正常的，计算的工作本来就非人脑而应该是电脑去做的事情。我们今天要开始学习最有意思的一种数据结构——图。在图的应用中，就有相应的算法来解决这样的问题。学完这一章，即便不能马上获得最终的答案，你也大概知道应该如何去做了。

7.2 图的定义

在线性表中，数据元素之间是被串起来的，仅有线性关系，每个数据元素只有一个直接前驱和一个直接后继。在树形结构中，数据元素之间有着明显的层次关系，并且每一层上的数据元素可能和下一层中多个元素相关，但只能和上一层中一个元素相关。这和一对父母可以有多个孩子，但每个孩子却只能有一对父母是一个道理。可现实中，人与人之间关系就非常复杂，比如我认识的朋友，可能他们之间也互相认识，这就不是简单的一对一、一对多，研究人际关系很自然会考虑多对多的情况。那就是我们今天要研究的主题——图。图是一种较线性表和树更加复杂的数据结构。在图形结构中，结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。

前面同学可能觉得树的术语好多，可来到了图，你就知道，什么才叫做真正的术语多。不过术语再多也是有规律可遵的，让我们开始“图”世界的旅程。如图 7-2-1 所示，先来看定义。

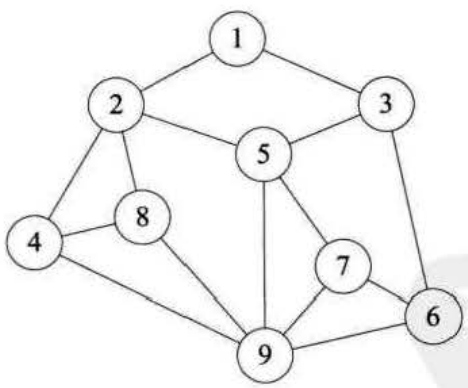


图 7-2-1

图（Graph）是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：G（V,E），其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合。

对于图的定义，我们需要明确几个注意的地方。

- 线性表中我们把数据元素叫元素，树中将数据元素叫结点，**在图中数据元素，我们则称之为顶点（Vertex）**。[7]
- 线性表中可以没有数据元素，称为空表。树中可以没有结点，叫做空树。那么对于图呢？我记得有一个笑话说一个小朋友拿着一张空白纸给别人却说这是他画的一幅“牛吃草”的画，“那草呢？”“草被牛吃光了。”“那牛呢？”“牛吃完草就走了呀。”之所以好笑是因为我们根本不认为一张空白纸算作画的。同样，在图结构中，不允许没有顶点。在定义中，若 V 是顶点的集合，则强调了顶点集合 V 有穷非空。[8]
- 线性表中，相邻的数据元素之间具有线性关系，树结构中，相邻两层的结点具有层次关系，而**图中，任意两个顶点之间都可能有关系，顶点之间的逻辑关系用边来表示**，边集可以是空的。

7.2.1 各种图定义

无向边：若顶点 v_i 到 v_j 之间的边没有方向，则称这条边为无向边（Edge），用无序偶对（v_i,v_j）来表示。如果图中任意两个顶点之间的边都是无向边，则称该图为无向图（Undirected graphs）。图 7-2-2 就是一个无向图，由于是无方向的，连接顶点 A 与 D 的边，可以表示成无序对（A,D），也可以写成（D,A）。

对于图 7-2-2 中的无向图 G_1 来说，G_1=（V_1,{E_1}），其中顶点集合 V_1={A,B,C,D}；边集合 E_1={（A,B），（B,C），（C,D），（D,A），（A,C）}

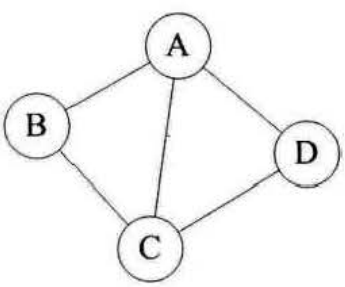


图 7-2-2

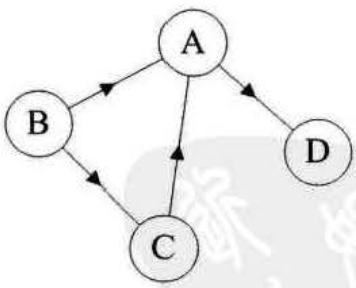


图 7-2-3

有向边：若从顶点 v_i 到 v_j 的边有方向，则称这条边为有向边，也称为弧（Arc）。

注[7]：有些书中也称图的顶点为 Node，在这里统一用 Vertex。

注[8]：此处定义有争议。国内部分教材中强调点集非空，但在 http://en.wikipedia.org/wiki/Null_graph 提出点集可为空。

用有序偶<v_i, v_j>来表示，v_i 称为弧尾（Tail），v_j 称为弧头（Head）。如果图中任意两个顶点之间的边都是有向边，则称该图为**有向图**（Directed graphs）。图 7-2-3 就是一个有向图。**连接顶点 A 到 D 的有向边就是弧，A 是弧尾，D 是弧头，<A，D>表示弧，注意不能写成<D，A>。**

对于图 7-2-3 中的有向图 G_2 来说，$G_2=(V_2,\{E_2\})$，其中顶点集合 $V_2=\{A,B,C,D\}$；弧集合 $E_2=\{<A,D>,<B,A>,<C,A>,<B,C>\}$。

看清楚了，**无向边用小括号“()”表示，而有向边则是用尖括号“<>”表示。**

在图中，若不存在顶点到其自身的边，且同一条边不重复出现，则称这样的图为简单图。我们课程里要讨论的都是简单图。显然图 7-2-4 中的两个图就不属于我们要讨论的范围。

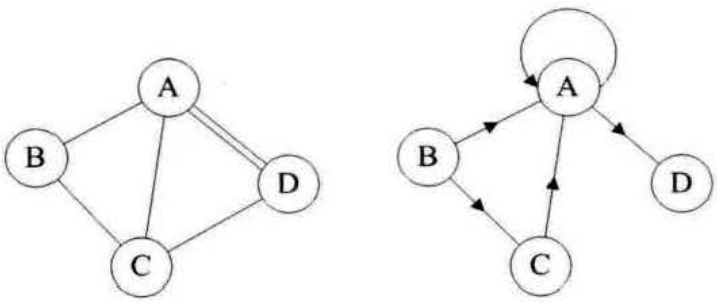


图 7-2-4

在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图。含有 n 个顶点的无向完全图有 $\frac{n\times(n-1)}{2}$ 条边。比如图 7-2-5 就是无向完全图，因为每个顶点都要与除它以外的顶点连线，顶点 A 与 BCD 三个顶点连线，共有四个顶点，自然是 4×3，但由于顶点 A 与顶点 B 连线后，计算 B 与 A 连线就是重复，因此要整体除以 2，共有 6 条边。

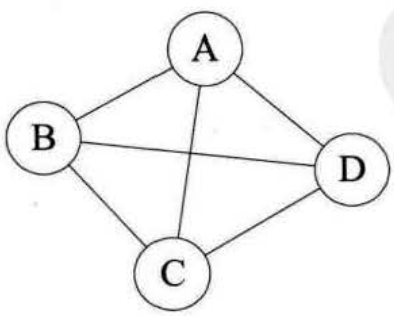


图 7-2-5

在有向图中，如果任意两个顶点之间都存在方向互为相反的两条弧，则称该图为有向完全图。含有 n 个顶点的有向完全图有 n×（n−1）条边，如图 7-2-6 所示。

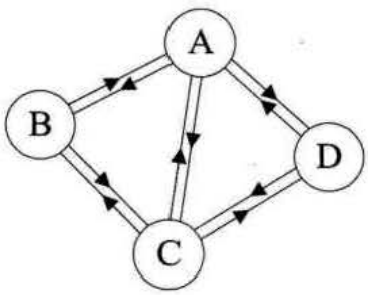


图 7-2-6

从这里也可以得到结论，对于具有 n 个顶点和 e 条边数的图，无向图 0≤e≤n（n−1）/2，有向图 0≤e≤n（n−1）。

有很少条边或弧的图称为稀疏图，反之称为稠密图。这里稀疏和稠密是模糊的概念，都是相对而言的。比如我去上海世博会那天，参观的人数差不多 50 万人，我个人感觉人数实在是太多，可以用稠密来形容。可后来听说，世博园里人数最多的一天达到了 103 万人，啊，50 万人是多么的稀疏呀。

有些图的边或弧具有与它相关的数字，这种**与图的边或弧相关的数叫做权（Weight）。**这些权可以表示从一个顶点到另一个顶点的距离或耗费。**这种带权的图通常称为网（Network）。**图 7-2-7 就是一张带权的图，即标识中国四大城市的直线距离的网，此图中的权就是两地的距离。

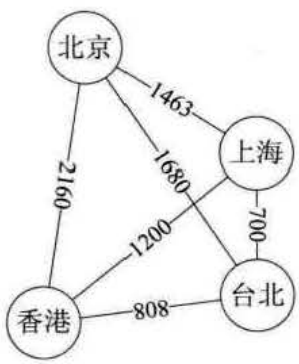


图 7-2-7

假设有两个图 G=（V,{E}）和 G'=（V',{E'}），如果 V'⊆V 且 E'⊆E，则称 G'为 G 的

子图（Subgraph）。例如图 7-2-8 带底纹的图均为左侧无向图与有向图的子图。

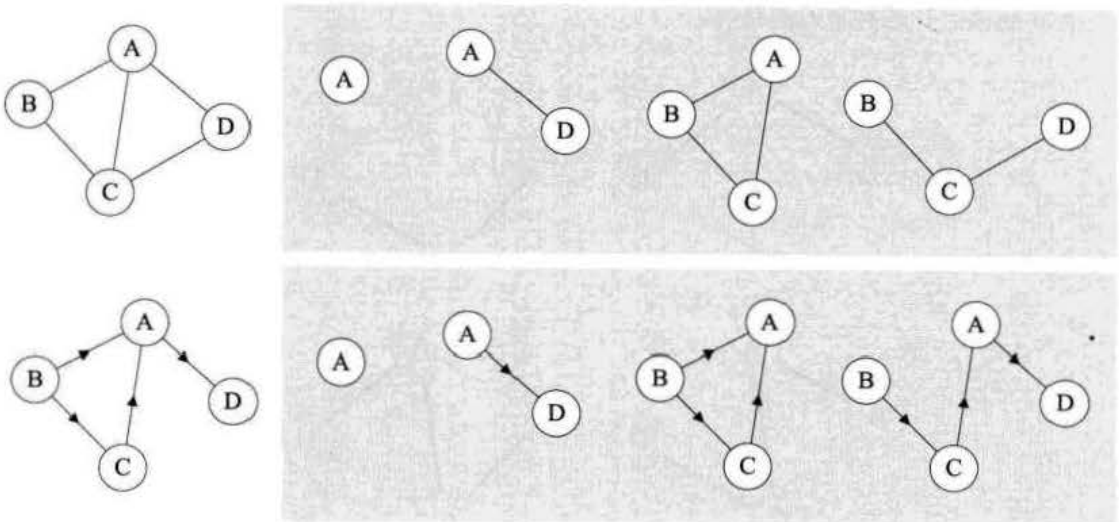


图 7-2-8

7.2.2 图的顶点与边间关系

对于无向图 **G=（V,{E}）**，如果边（**v,v'**）∈**E**，则称顶点 **v** 和 **v'**互为邻接点（**Adjacent**），即 **v** 和 **v'**相邻接。边（**v,v'**）依附（**incident**）于顶点 **v** 和 **v'**，或者说（**v,v'**）与顶点 **v** 和 **v'**相关联。顶点 **v** 的度（**Degree**）是和 **v** 相关联的边的数目，记为 **TD（v）**。例如图 7-2-8 左侧上方的无向图，顶点 A 与 B 互为邻接点，边（A,B）依附于顶点 A 与 B 上，顶点 A 的度为 3。而此图的边数是 5，各个顶点度的和=3+2+3+2=10，推敲后发现，边数其实就是各顶点度数和的一半，多出的一半是因为重复两次记数。简记之，$e=\frac{1}{2}\sum_{i=1}^{n}TD(v_i)$。

对于有向图 **G=（V,{E}）**，如果弧**<v,v'>**∈**E**，则称顶点 **v** 邻接到顶点 **v'**，顶点 **v'**邻接自顶点 **v**。弧**<v,v'>**和顶点 **v**，**v'**相关联。以顶点 **v** 为头的弧的数目称为 **v** 的入度（**InDegree**），记为 **ID（v）**；以 **v** 为尾的弧的数目称为 **v** 的出度（**OutDegree**），记为 **OD（v）**；顶点 **v** 的度为 **TD（v）=ID（v）+OD（v）**。例如图 7-2-8 左侧下方的有向图，顶点 A 的入度是 2（从 B 到 A 的弧，从 C 到 A 的弧），出度是 1（从 A 到 D 的弧），所以顶点 A 的度为 2+1=3。此有向图的弧有 4 条，而各顶点的出度和=1+2+1+0=4，各顶点的入度和=2+0+1+1=4。所以得到 $e=\sum_{i=1}^{n}ID(v_i)=\sum_{i=1}^{n}OD(v_i)$。

无向图 **G=（V,{E}）**中从顶点 **v** 到顶点 **v'**的路径（**Path**）是一个顶点序列

$(v=v_{i,0},v_{i,1},\cdots,v_{i,m}=v')$，其中 $(v_{i,j-1},v_{i,j}) \in E$，$1 \leqslant j \leqslant m$。例如图 7-2-9 中就列举了顶点 B 到顶点 D 四种不同的路径。

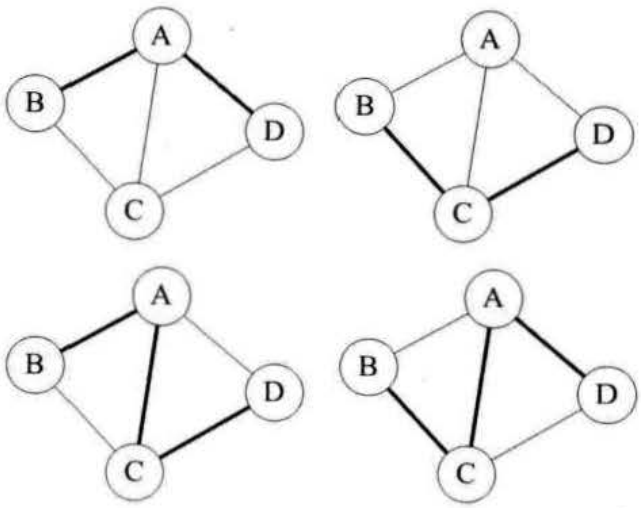


图 7-2-9

如果 G 是有向图，则路径也是有向的，顶点序列应满足$<v_{i,j-1},v_{i,j}> \in E$，$1 \leqslant j \leqslant m$。例如图 7-2-10，顶点 B 到 D 有两种路径。而顶点 A 到 B，就不存在路径。

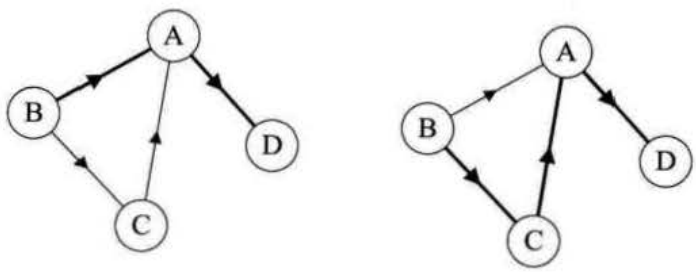


图 7-2-10

树中根结点到任意结点的路径是唯一的，但是图中顶点与顶点之间的路径却是不唯一的。

路径的长度是路径上的边或弧的数目。图 7-2-9 中的左侧两条路径长度为 2，右侧两条路径长度为 3。图 7-2-10 左侧路径长为 2，右侧路径长度为 3。

第一个顶点到最后一个顶点相同的路径称为回路或环（Cycle）。序列中顶点不重复出现的路径称为简单路径。除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路，称为简单回路或简单环。图 7-2-11 中两个图的粗线都构成环，左侧的环

因第一个顶点和最后一个顶点都是 B，且 C、D、A 没有重复出现，因此是一个简单环。而右侧的环，由于顶点 C 的重复，它就不是简单环了。

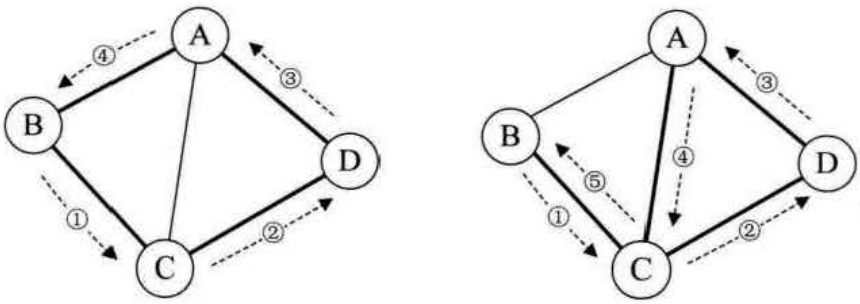


图 7-2-11

7.2.3 连通图相关术语

在无向图 **G** 中，如果从顶点 **v** 到顶点 **v'**有路径，则称 **v** 和 **v'**是连通的。如果对于图中任意两个顶点 v_i、$v_j \in E$，v_i 和 v_j 都是连通的，则称 **G** 是连通图（**Connected Graph**）。图 7-2-12 的图 1，它的顶点 A 到顶点 B、C、D 都是连通的，但显然顶点 A 与顶点 E 或 F 就无路径，因此不能算是连通图。而图 7-2-12 的图 2，顶点 A、B、C、D 相互都是连通的，所以它本身是连通图。

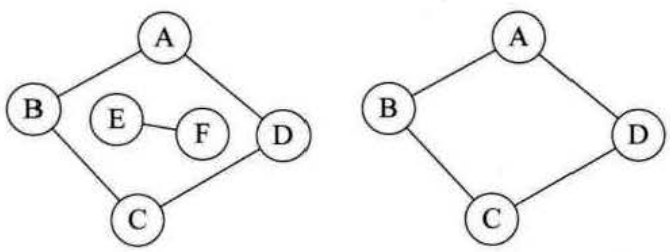


图1　　图2

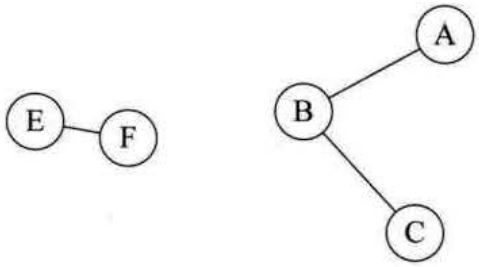


图3　　图4

图 7-2-12

无向图中的极大连通子图称为连通分量。注意连通分量的概念，它强调：

- 要是子图；
- 子图要是连通的；
- 连通子图含有极大顶点数；
- 具有极大顶点数的连通子图包含依附于这些顶点的所有边。

图 7-2-12 的图 1 是一个无向非连通图。但是它有两个连通分量，即图 2 和图 3。而图 4，尽管是图 1 的子图，但是它却不满足连通子图的极大顶点数（图 2 满足）。因此它不是图 1 的无向图的连通分量。

在有向图 G 中，如果对于每一对 v_i、$v_j \in V$、$v_i \neq v_j$，从 v_i 到 v_j 和从 v_j 到 v_i 都存在路径，则称 G 是强连通图。有向图中的极大强连通子图称做有向图的强连通分量。例如图 7-2-13，图 1 并不是强连通图，因为顶点 A 到顶点 D 存在路径，而 D 到 A 就在存在。图 2 就是强连通图，而且显然图 2 是图 1 的极大强连通子图，即是它的强连通分量。

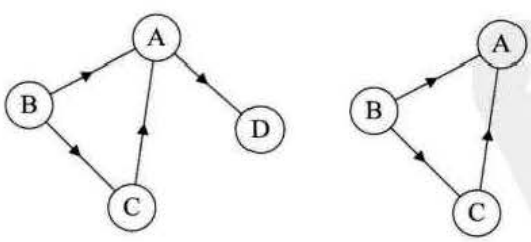


图1　　图2

图 7-2-13

现在我们再来看连通图的生成树定义。

所谓的**一个连通图的生成树是一个极小的连通子图，它含有图中全部的 n 个顶点，但只有足以构成一棵树的 n－1 条边**。比如图 7-2-14 的图 1 是一普通图，但显然它不是生成树，当去掉两条构成环的边后，比如图 2 或图 3，就满足 n 个顶点 n－1 条边且连通的定义了。它们都是一棵生成树。从这里也可知道，如果一个图有 n 个顶点和小于 n－1 条边，则是非连通图，如果它多于 n－1 边条，必定构成一个环，因为这条边使得它依附的那两个顶点之间有了第二条路径。比如图 2 和图 3，随便加哪两顶点的边都将构成环。不过有 n－1 条边并不一定是生成树，比如图 4。

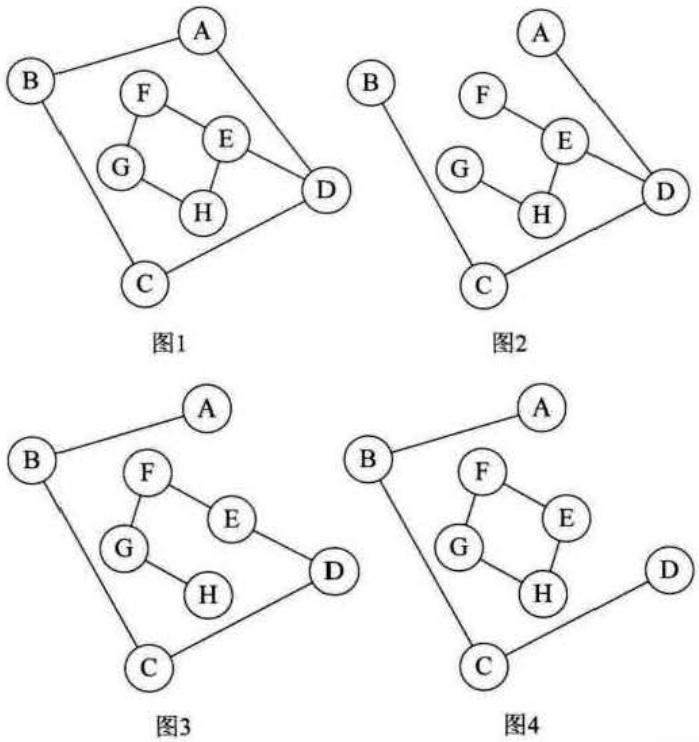


图 7-2-14

如果一个有向图恰有一个顶点的入度为 0，其余顶点的入度均为 1，则是一棵有向树。对有向树的理解比较容易，所谓入度为 0 其实就相当于树中的根结点，其余顶点入度为 1 就是说树的非根结点的双亲只有一个。**一个有向图的生成森林由若干棵有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的弧**。如图 7-2-15 的图 1 是一棵有向图。去掉一些弧后，它可以分解为两棵有向树，如图 2 和图 3，这两棵就是图 1 有向图的生成森林。

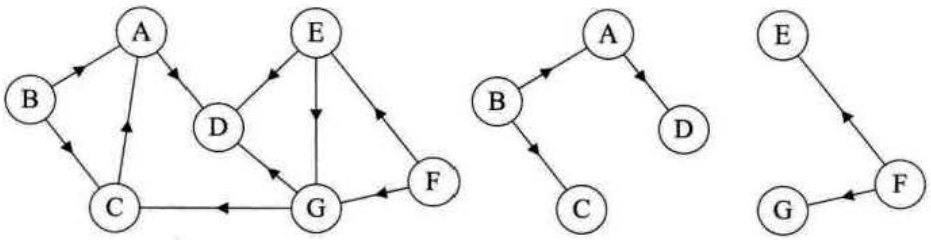


图1　　图2　　图3

图 7-2-15

7.2.4 图的定义与术语总结

术语终于介绍得差不多了，可能有不少同学有些头晕，我们再来整理一下。

图按照有无方向分为**无向图**和**有向图**。无向图由**顶点**和**边**构成，有向图由顶点和**弧**构成。弧有**弧尾**和**弧头**之分。

图按照边或弧的多少分**稀疏图**和**稠密图**。如果任意两个顶点之间都存在边叫**完全图**，有向的叫**有向完全图**。若无重复的边或顶点到自身的边则叫**简单图**。

图中顶点之间有**邻接点**、**依附**的概念。无向图顶点的边数叫做**度**，有向图顶点分为**入度**和**出度**。

图上的边或弧上带**权**则称为**网**。

图中顶点间存在**路径**，两顶点存在路径则说明是**连通**的，如果路径最终回到起始点则称为**环**，当中不重复叫**简单路径**。若任意两顶点都是连通的，则图就是**连通图**，有向则称**强连通图**。图中有子图，若子图极大连通则就是**连通分量**，有向的则称**强连通分量**。

无向图中连通且 n 个顶点 n－1 条边叫**生成树**。有向图中一顶点入度为 0 其余顶点入度为 1 的叫**有向树**。一个有向图由若干棵有向树构成**生成森林**。

7.3 图的抽象数据类型

图作为一种数据结构，它的抽象数据类型带有自己特点，正因为它的复杂，运用广泛，使得不同的应用需要不同的运算集合，构成不同的抽象数据操作。我们这里就来看看图的基本操作。

```
ADT 图(Graph)
Data
    顶点的有穷非空集合和边的集合。
Operation
    CreateGraph(*G,V,VR):按照顶点集V和边弧集VR的定义构造图G。
    DestroyGraph(*G):图G存在则销毁。
    LocateVex(G,u):若图G中存在顶点u,则返回图中的位置。
    GetVex(G,v):返回图G中顶点v的值。
    PutVex(G,v,value):将图G中顶点v赋值value。
    FirstAdjVex(G,*v):返回顶点v的一个邻接顶点,若顶点在G中无邻接顶点返回空。
    NextAdjVex(G,v,*w):返回顶点v相对于顶点w的下一个邻接顶点,若w是v的最后
                        一个邻接点则返回"空"。
    InsertVex(*G,v):在图G中增添新顶点v。
    DeleteVex(*G,v):删除图G中顶点v及其相关的弧。
    InsertArc(*G,v,w):在图G中增添弧<v,w>,若G是无向图,还需要增添对称弧
                       <w,v>。
    DeleteArc(*G,v,w):在图G中删除弧<v,w>,若G是无向图,则还删除对称弧
                       <w,v>。
    DFSTraverse(G):对图G中进行深度优先遍历,在遍历过程对每个顶点调用。
    HFSTraverse(G):对图G中进行广度优先遍历,在遍历过程对每个顶点调用。
endADT
```

7.4 图的存储结构

图的存储结构相较线性表与树来说就更加复杂了。首先，我们口头上说的“顶点的位置”或“邻接点的位置”只是一个相对的概念。其实从图的逻辑结构定义来看，图上任何一个顶点都可被看成是第一个顶点，任一顶点的邻接点之间也不存在次序关系。比如图 7-4-1 中的四张图，仔细观察发现，它们其实是同一个图，只不过顶点的位置不同，就造成了表象上不太一样的感觉。

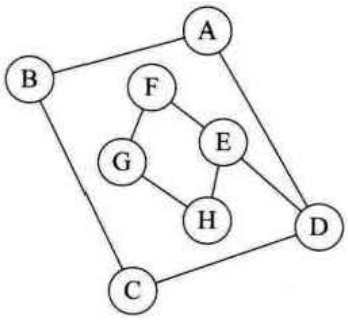


图1

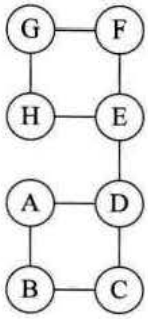


图2

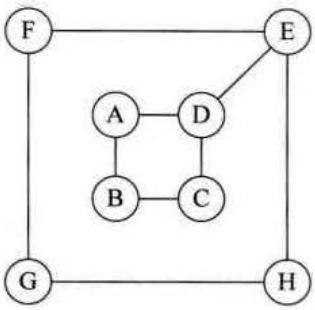


图3

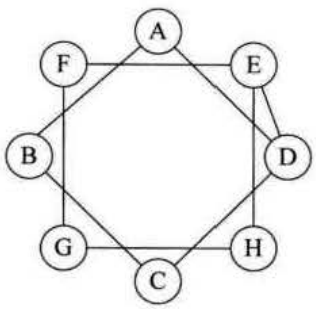


图4

图 7-4-1

也正由于图的结构比较复杂，任意两个顶点之间都可能存在联系，因此无法以数据元素在内存中的物理位置来表示元素之间的关系，也就是说，图不可能用简单的顺序存储结构来表示。而多重链表的方式，即以一个数据域和多个指针域组成的结点表示图中的一个顶点，尽管可以实现图结构，但其实在树中，我们也已经讨论过，这是有问题的。如果各个顶点的度数相差很大，按度数最大的顶点设计结点结构会造成很多存储单元的浪费，而若按每个顶点自己的度数设计不同的顶点结构，又带来操作的不便。因此，对于图来说，如何对它实现物理存储是个难题，不过我们的前辈们已经解决了，现在我们来看前辈们提供的五种不同的存储结构。

7.4.1 邻接矩阵

考虑到图是由顶点和边或弧两部分组成。合在一起比较困难，那就很自然地考虑到分两个结构来分别存储。顶点不分大小、主次，所以用一个一维数组来存储是很不错的选择。而边或弧由于是顶点与顶点之间的关系，一维搞不定，那就考虑用一个二维数组来存储。于是我们的邻接矩阵的方案就诞生了。

图的邻接矩阵（**Adjacency Matrix**）存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组（称为邻接矩阵）存储图中的边或弧的信息。

设图 G 有 n 个顶点，则邻接矩阵是一个 n×n 的方阵，定义为：

$$arc[i][j] = \begin{cases} 1, 若(v_i,v_j) \in E 或 < v_i,v_j > \in E \\ 0, 反之 \end{cases}$$

我们来看一个实例，图 7-4-2 的左图就是一个无向图。

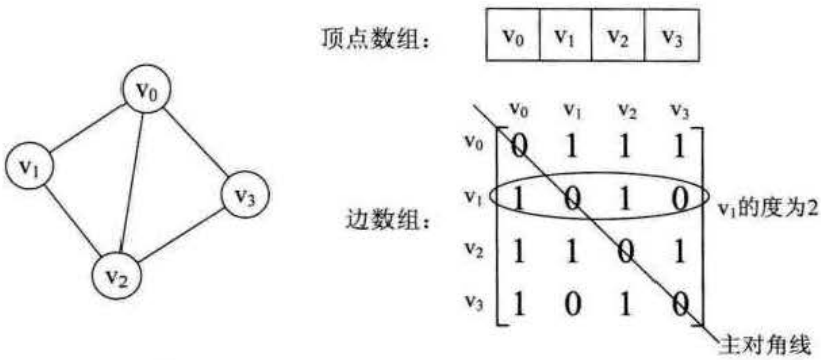


图 7-4-2

我们可以设置两个数组，顶点数组为 vertex[4]={ v_0, v_1, v_2, v_3}，边数组 arc[4][4]为图 7-4-2 右图这样的一个矩阵。简单解释一下，对于矩阵的主对角线的值，即 arc[0][0]、arc[1][1]、arc[2][2]、arc[3][3]，全为 0 是因为不存在顶点到自身的边，比如 v_0 到 v_0。arc[0][1]=1 是因为 v_0 到 v_1 的边存在，而 arc[1][3]=0 是因为 v_1 到 v_3 的边不存在。并且由于是无向图，v_1 到 v_3 的边不存在，意味着 v_3 到 v_1 的边也不存在。所以无向图的边数组是一个对称矩阵。

嗯？对称矩阵是什么？忘记了不要紧，复习一下。所谓对称矩阵就是 n 阶矩阵的元满足 $a_{ij}=a_{ji}$，($0 \leqslant i,j \leqslant n$)。即从矩阵的左上角到右下角的主对角线为轴，右上角的元与左下角相对应的元全都是相等的。

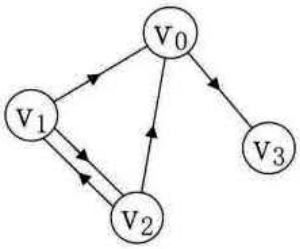
有了这个矩阵，我们就可以很容易地知道图中的信息。

1．我们要判定任意两顶点是否有边无边就非常容易了。

2．我们要知道某个顶点的度，其实就是这个顶点 v_i 在邻接矩阵中第 i 行（或第 i 列）的元素之和。比如顶点 v_1 的度就是 1+0+1+0=2。

3．求顶点 v_i 的所有邻接点就是将矩阵中第 i 行元素扫描一遍，arc[i][j]为 1 就是邻接点。

我们再来看一个有向图样例，如图 7-4-3 所示的左图。



顶点数组：

v_0	v_1	v_2	v_3

边数组：

	v_0	v_1	v_2	v_3
v_0	0	0	0	1
v_1	1	0	1	0
v_2	1	1	0	0
v_3	0	0	0	0

v_1的出度为2

v_1的入度为1

图 7-4-3

顶点数组为 vertex[4]={ v_0, v_1, v_2, v_3}，弧数组 arc[4][4]为图 7-4-3 右图这样的一个矩阵。主对角线上数值依然为 0。但因为是有向图，所以此矩阵并不对称，比如由 v_1 到 v_0 有弧，得到 arc[1][0]=1，而 v_0 到 v_1 没有弧，因此 arc[0][1]=0。

有向图讲究入度与出度，顶点 v_1 的入度为 1，正好是第 v_1 列各数之和。顶点 v_1 的出度为 2，即第 v_1 行的各数之和。

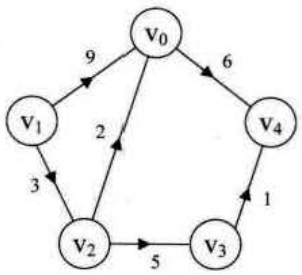
与无向图同样的办法，判断顶点 v_i 到 v_j 是否存在弧，只需要查找矩阵中 arc[i][j] 是否为 1 即可。要求 v_i 的所有邻接点就是将矩阵第 i 行元素扫描一遍，查找 arc[i][j] 为 1 的顶点。

在图的术语中，我们提到了网的概念，也就是每条边上带有权的图叫做网。那么这些权值就需要存下来，如何处理这个矩阵来适应这个需求呢？我们有办法。

设图 G 是网图，有 n 个顶点，则邻接矩阵是一个 n×n 的方阵，定义为：

$$arc[i][j]=\begin{cases} W_{ij}, 若(v_i,v_j)\in E 或 <v_i,v_j>\in E \\ 0, \quad 若 i=j \\ \infty, \quad 反之 \end{cases}$$

这里 w_{ij} 表示（v_i,v_j）或<v_i,v_j>上的权值。∞表示一个计算机允许的、大于所有边上权值的值，也就是一个不可能的极限值。有同学会问，为什么不是 0 呢？原因在于权值 w_{ij} 大多数情况下是正值，但个别时候可能就是 0，甚至有可能是负值。因此必须要用一个不可能的值来代表不存在。如图 7-4-4 左图就是一个有向网图，右图就是它的邻接矩阵。



顶点数组：

v_0	v_1	v_2	v_3	v_4

边数组：

	v_0	v_1	v_2	v_3	v_4
v_0	0	∞	∞	∞	6
v_1	9	0	3	∞	∞
v_2	2	∞	0	5	∞
v_3	∞	∞	∞	0	1
v_4	∞	∞	∞	∞	0

图 7-4-4

那么邻接矩阵是如何实现图的创建的呢？我们先来看看图的邻接矩阵存储的结构，代码如下。

```
typedef char VertexType;                /* 顶点类型应由用户定义 */
typedef int EdgeType;                   /* 边上的权值类型应由用户定义 */
#define MAXVEX 100                      /* 最大顶点数，应由用户定义 */
#define INFINITY 65535                  /* 用 65535 来代表∞ */
typedef struct
{
    VertexType vexs[MAXVEX];            /* 顶点表 */
    EdgeType arc[MAXVEX][MAXVEX];       /* 邻接矩阵，可看作边表 */
    int numVertexes, numEdges;          /* 图中当前的顶点数和边数 */
}MGraph;
```

有了这个结构定义，我们构造一个图，其实就是给顶点表和边表输入数据的过程。我们来看看无向网图的创建代码。

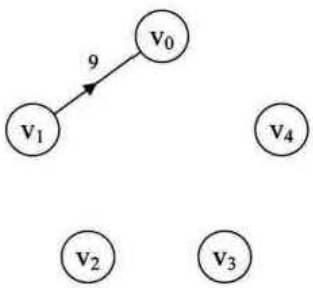
```
/* 建立无向网图的邻接矩阵表示 */
void CreateMGraph(MGraph *G)
{
    int i,j,k,w;
    printf("输入顶点数和边数:\n");
    scanf("%d,%d",&G->numVertexes,&G->numEdges);/* 输入顶点数和边数 */
    for(i = 0;i <G->numVertexes;i++)   /* 读入顶点信息，建立顶点表 */
        scanf(&G->vexs[i]);
    for(i = 0;i <G->numVertexes;i++)
        for(j = 0;j <G->numVertexes;j++)
```

```
            G->arc[i][j]= INFINITY;    /* 邻接矩阵初始化 */
    for (k = 0;k <G->numEdges;k++)  /* 读入 numEdges 条边，建立邻接矩阵 */
    {
        printf("输入边（vi,vj）上的下标 i，下标 j 和权 w:\n");
        scanf("%d,%d,%d",&i,&j,&w); /* 输入边（vi,vj）上的权 w */
        G->arc[i][j]=w;
        G->arc[j][i]= G->arc[i][j]; /* 因为是无向图，矩阵对称 */
    }
}
```

从代码中也可以得到，n 个顶点和 e 条边的无向网图的创建，时间复杂度为 $O(n+n^2+e)$，其中对邻接矩阵 Garc 的初始化耗费了 $O(n^2)$的时间。

7.4.2 邻接表

邻接矩阵是不错的一种图存储结构，但是我们也发现，对于边数相对顶点较少的图，这种结构是存在对存储空间的极大浪费的。比如说，如果我们要处理图 7-4-5 这样的稀疏有向图，邻接矩阵中除了 arc[1][0]有权值外，没有其他弧，其实这些存储空间都浪费掉了。



顶点数组：

v0	v1	v2	v3	v4

边数组：

	v0	v1	v2	v3	v4
v0	0	∞	∞	∞	∞
v1	9	0	∞	∞	∞
v2	∞	∞	0	∞	∞
v3	∞	∞	∞	0	∞
v4	∞	∞	∞	∞	0

图 7-4-5

因此我们考虑另外一种存储结构方式。回忆我们在线性表时谈到，顺序存储结构就存在预先分配内存可能造成存储空间浪费的问题，于是引出了链式存储的结构。同样的，我们也可以考虑对边或弧使用链式存储的方式来避免空间浪费的问题。

再回忆我们在树中谈存储结构时，讲到了一种孩子表示法，将结点存入数组，并对结点的孩子进行链式存储，不管有多少孩子，也不会存在空间浪费问题。这个思路