第12章　图

恭喜！你已经成功穿越了“树”的森林，下面要学习图这种数据结构。令人惊叹的是，图可以用来描述成千上万的实际问题，不过，我们仅研究其中的一小部分。本章的主要内容如下：

- 图的若干术语：顶点，边，邻接，关联，度，回路，路径，连通构件，生成树。
- 图的三种类型：无向图，有向图和加权的图。
- 图的常用表示方法：邻接矩阵，邻接链表和邻接压缩表。
- 图的标准搜索方法：宽度优先搜索和深度优先搜索。
- 在图中寻找路径，在无向图中寻找连通构件以及在无向连通图中寻找生成树的算法。
- 如何把抽象数据类型表示成一个抽象类。

本章所使用的新的C++特征是：抽象类，虚函数和虚基类。

12.1　基本概念

简单地说，图（graph）是一个用线或边连接在一起的顶点或节点的集合。正式一点的说法是，图$G=(V,E)$是一个V和E的有限集合，元素V称为顶点（vertice，也叫作节点或点），元素E称为边（edge，也叫作弧或连线），E中的每一条边连接V中两个不同的顶点。可以用（i，j）来表示一条边，其中i和j是E所连接的两个顶点。

一般来说，图是由回路和边组成，如图12-1所示。在图12-1中有些边是带方向的（带箭头），而有些边是不带方向的。带方向的边叫有向边（ directed edge），而不带方向的边叫无向边（undirected edge）。对无向边来说，(i, j)和(j, i)是一样的；而对有向边来说，它们是不同的。前者的方向是从i到j，后者是从j到i。

当且仅当(i, j)是图中的边时，顶点i和j是邻接的（adjacent）。边(i, j)关联（incident）于顶点i和j。图12-1a中的顶点1和2是邻接的，顶点1和3，1和4，2和3，3和4也是邻接的，除此之外，这个图中没有其他邻接的顶点。边（1，2）关联于顶点1和2，（2，3）关联于顶点2和3。

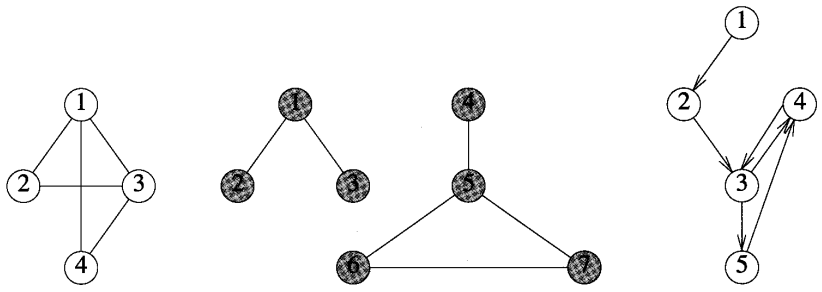


图12-1　图

⊖ 有些书中用$\{i, j\}$表示无向边，而用（i, j）表示有向边。还有一些书用（i, j）表示无向边，用$<i, j>$表示有向边。本书对两种边使用同一符号（i, j），边有向与否可从上下文中看出。

在有向图中，有时候对邻接和关联的概念作更精确的定义非常有用。有向边 (i, j) 是关联至（incident to）顶点j 而关联于（incident from）顶点i。顶点i 邻接至（adjacent to）顶点j，顶点j 邻接于（adjacent from）顶点i。在图12-1c 的图中，顶点2邻接于顶点1，而1邻接至顶点2。边（1，2）关联于顶点1而关联至顶点2。顶点4邻接至顶点3且邻接于顶点3。边（3，4）是关联于顶点3而关联至顶点4。对于无向图来说，“至”和“于”的含义是相同的。

如果使用集合的表示方法，图 12-1中的几个图可以用如下方法表示： $G1$=（$V1$，$E1$）；$G2$=（$V2$，$E2$）和$G3$=（$V3$，$E3$），其中：

V_1={1,2,3,4}； E_1={(1,2),(1,3),(2,3),(1,4),(3,4)}

V_2={1,2,3,4,5,6,7}； E_2={(1,2),(1,3),(4,5),(5,6),(5,7),(6,7)}

V_3={1,2,3,4,5}； E_3={(1,2),(2,3),(3,4),(4,3),(3,5),(5,4)}

如果图中所有的边都是无向边，那么该图叫作无向图（undirected graph），图12-1a 和b 都是无向图。如果所有的边都是有向的，那么该图叫作有向图（directed graph），图12-1c 是一个有向图。

由定义知道，一个图中不可能包括同一条边的多个副本，因此，在无向图中的任意两个顶点之间，最多只能有一条边。在有向图中的任意两个顶点之间，最多只能有一条边从顶点 i 到顶点j 或从j 到i。并且一个图中不可能包含自连边（self-edge），即(i,i) 类型的边，自连边也叫作环（loop）。

通常把无向图简称为图，有向图仍称为有向图（ digraph）。在一些图和有向图的应用中，我们会为每条边赋予一个权或耗费，这种情况下，用术语加权有向图（ weighted graph）和加权无向图（weighted digraph）来描述所得到的数据对象。术语网络（ network）在这里是指一个加权有向图或加权无向图。实际上，这里定义的所有图的变化都可以看作网络的一种特殊情况——一个无向（有向）图可以被看作是一个所有边具有相同权的无向（有向）网络。

12.2 应用

无向图，有向图和网络常常用于电子网络的分析、化合物（特别是碳氢化合物）的分子结构研究、空中航线和通信网络的描述、项目策划、遗传研究、统计、社会科学及它各种领域。这一节将用图来阐述一些实际问题。

例12-1 [路径问题] 城市中有许多街道，每一个十字路口都可以看作图中一个顶点，邻接两个十字路口之间的每一段街道既可以看作一条，也可以看作两条有向边。如果街道是双向的，就用两条有向边。如果街道是单向的，就用一条有向边。图12-2给出了假想的街道和相应的有向图。图中有三条街道：街道1，街道2和街道3以及两条大街：大街1和大街2。十字路口用数字1到6进行编号，相应的有向图（如图12-2b所示）的顶点标号与图12-2a 给出的十字路口的标号相同。

当且仅当对于每一个j（$1 \le j \le k$），边（i_j，i_{j+1}）都在E中时，顶点序列$P=i_1, i_2, \cdots, i_k$ 是图或有向图$G=(V, E)$中一条从i_1 到i_k 的路径。当且仅当相应的有向图中顶点i 到顶点j 有一条路径时，十字路口i 到j 之间存在一条路径。在图 12-2b 的有向图中，5，2，1是从5到1的一条路径，在这个有向图中，从5到4之间没有路径。

简单路径是这样一条路径：除第一个和最后一个顶点以外，路径中其他所有顶点均不同。路径5,2,1 是简单路径，而2,5,2,1则不是。

对于图或有向图的每一条边，均可以给出一个长度。路径的长度是路径上所有边的长度之和。从十字路口i 到j 的最短路径是相应网络中顶点i 到j 的最短路径。

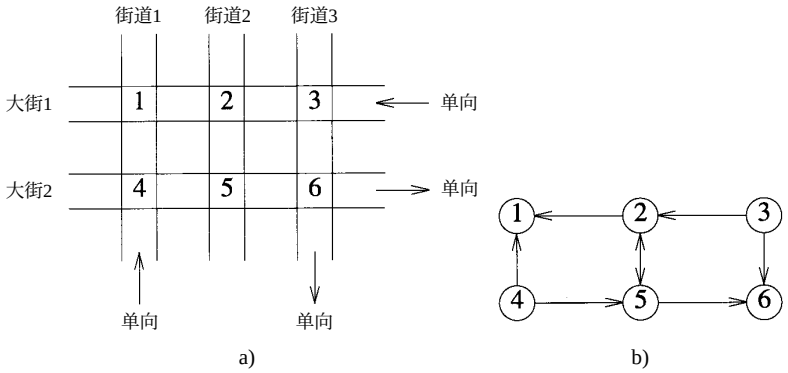


图12-2 街道及其相应的有向图

a) 街道地图 b) 有向图

例12-2 [生成树] 设$G=(V,E)$是一个无向图，当且仅当G中每一对顶点之间有一条路径时，可认为G是连通的（connected）。图12-1a 中的无向图是连通的，而b 中的无向图不是。假定G是一个通信网络，V是城市的集合，E是通信链路的集合。当且仅当G是连通的时候，V中的每一对城市之间可以通信。图12-1a 的通信网络中，城市2和4之间可以通过链路2，3，4进行通信，而图12-1b 的网络中，城市2和4不能通信。

假设G是连通的，G中的有些边可能不是必需的，因此即使将它从G中去掉，G仍然可以保持连通。在图12-1a 中，即使将边(2,3)和（1,4）去掉，整个图仍可以保持连通。

图H是图G的子图（subgraph）的充要条件是，H的顶点和边的集合是G的顶点和边的集合的子集。环路（cycle）的起始节点与结束节点是同一节点。例如，图 12-1a 中，1,2,3,1是一个环路。没有环路的无向连通图是一棵树。一棵包含 G中所有顶点并且是G的子图的树是G的生成树（spanning tree）。图12-1a 的生成树如图12-3所示。

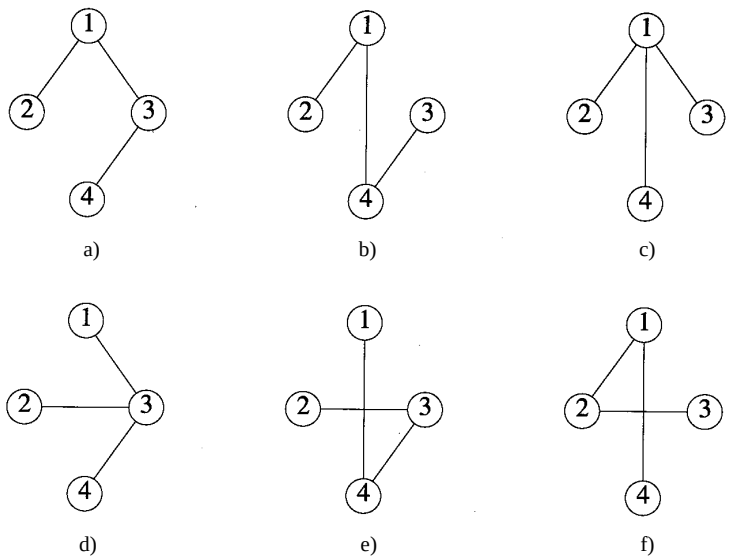


图12-3 图12-1a 的生成树

一个n节点的连通图必须至少有n-1条边。因此当通信网络的每条链路具有相同的建造费用时，在任意一棵生成树上建设所有的链路可以将网络建设费用减至最小，并且能保证每两个城市之间存在一条通信路径。如果链路具有不同的耗费，那么需要在一棵最小耗费生成树（生成树的耗费是所有边的耗费之和）上建立链路。图 12-4 给出了一个图和它的生成树，图 12-4b 的生成树是一棵最小耗费生成树。

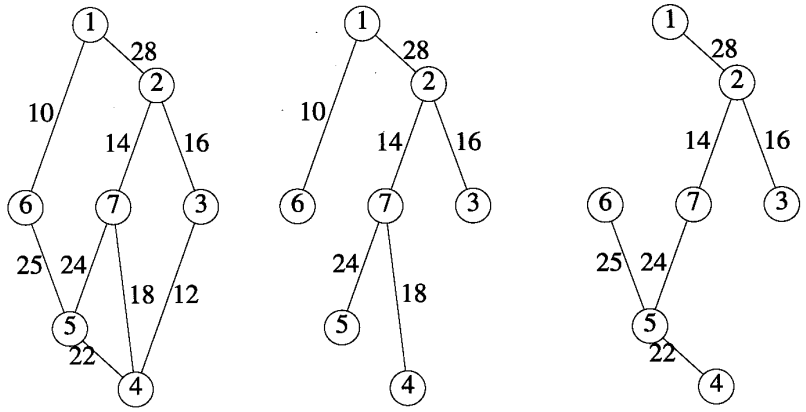


图12-4 连通图和它的两棵生成树

a) 图 b) 耗费为100的生成树 c) 耗费为129的生成树

例12-3 [**翻译人员**] 假设你正在策划一次国际性会议，此次大会上的所有发言人都只会说英语，而参加会议的其他人说的语言是{L1, L2, …, Ln}之一。翻译小组能够将英语与其他语言互译。现在你的任务是如何使翻译小组的人数最少。

可以将这个任务转化为一个图的问题。在这个问题中有两组顶点，一组是相应的翻译人员，一组是语言 (如图12-5所示)。在翻译人员i 与语言Lj 之间存在一条边的充要条件是翻译人员i 能够将英语和Lj 互译。当且仅当一条边连接翻译人员和语言时，翻译人员i 覆盖语言Li。我们需要找到能够覆盖所有语言顶点的最小翻译人员顶点子集。

图12-5有一个有趣的特征：可以将顶点集合分成两个子集 A（翻译人员顶点）和 B（语言顶点），这样每条边在A中有一个端点，在B中有一个端点，具有这种特征的图叫作二分图（bipartite graph）。

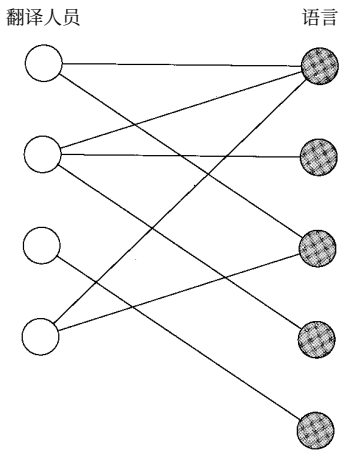


图12-5 翻译人员和语言

12.3 特性

设G是一个无向图，顶点 i 的度（degree）d_i 是与顶点i 相连的边的个数。对于图 12-1a，d_1=3, d_2=2, d_3=3, d_4=2。

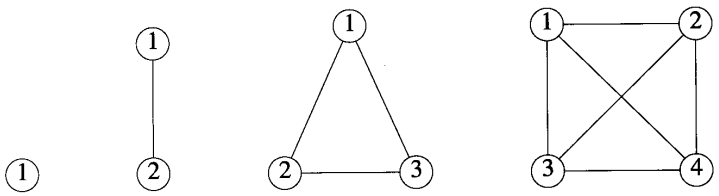
特性1 设$G=(V, E)$是一个无向图，令$|V|=n$, $|E|=e$, d_i为顶点i的度，则

1) $\sum_{i=1}^{n} d_i = 2e$

2) $0 \leq e \leq n(n-1)/2$

证明 要证明1)，注意到无向图中的每一条边与两个顶点相连，因此顶点的度之和等于边的数量的2倍。对于2)，一个顶点的度是在0到n-1之间，因此度的和在0到$n(n-1)$之间,从1) 可知，e是在0到$n(n-1)/2$之间。

一个具有n个顶点，$n(n-1)/2$条边的图是一个完全图（complete graph）。图12-6给出了n=1,2,3和4时的完全图。k_n代表n顶点的完全图。



a) b) c) d)

图12-6 完全图

a) K_1 b) K_2 c) K_3 d) K_4

设G是一个有向图，顶点i的入度（in-degree）d_i^{in}是指关联至顶点i的边的数量。顶点i的出度（out-degree）d_i^{out}是指关联于该顶点的边的数量。对于图12-1c的有向图，$d_1^{in}=0$，$d_1^{out}=0$，$d_2^{in}=1$，$d_2^{out}=1$，$d_3^{in}=2$，$d_3^{out}=2$。

特性2 设$G=(V,E)$是一个有向图，n和e的定义与特性1相同，则

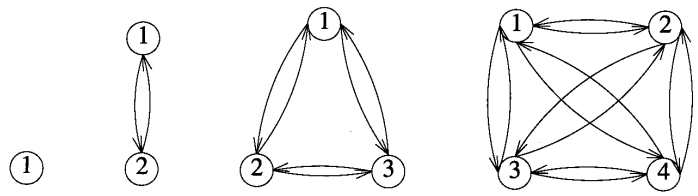
1) $0 \leq e \leq n(n-1)$

2) $\sum_{i=1}^{n} d_i^{in} = \sum_{i=1}^{n} d_i^{out} = e$

证明 在练习2中，将要求完成这个特性的证明。

一个n顶点的完全有向图（complete digraph）包含$n(n-1)$条有向边，图12-7给出了n=1,2,3和4时的完全有向图。

入度和出度在无向图中可以作为度的同义词。本节提供的定义可以直接扩充到网络中。



a) b) c) d)

图12-7 完全有向图

a) K_1 b) K_2 c) K_3 d) K_4

练习

1. 对于图12-8的每一个有向图，确定下列各项：

1) 每个顶点的入度。
2) 每个顶点的出度。
3) 邻接于顶点2的顶点集合。
4) 邻接至顶点1的顶点集合。
5) 关联于顶点3的边的集合。
6) 关联至顶点4的边的集合。
7) 所有的有向环路和它们的长度。

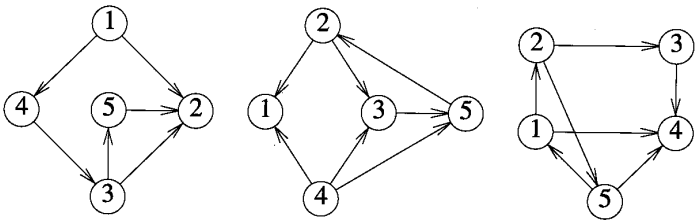


图12-8 有向图

2. 证明特性2。
3. 设G是任意无向图，证明有偶数个度数为奇数的顶点。
4. 设$G=(V,E)$是$|V|>1$的连通图，证明G中包含一个度数为1的顶点或一个环路（或两者都有）。
5. 设$G=(V,E)$是至少包含一个环路的连通图，边(i, j)至少出现在一个环路中。证明图$h=(V,E-\{(i, j)\})$也是连通的。
6. 证明：
1) 对于每一个$n(n\geq 1)$，都存在一个包含n-1条边的无向连通图。
2) 每一个n顶点的无向连通图至少有n-1条边。可以使用练习4, 5的结论。
7. 一个有向图是强连通（strongly connected）的充要条件是：对于每一对不同顶点i和j，从i到j和从j到i都有一个有向路径。
1) 证明对于每一个n（$n\geq 2$），都存在一个包含n条边的强连通有向图。
2) 证明每一个n（$n\geq 2$）顶点的强连通有向图至少包含n条边。
3) 写一个过程确定有向图G是否是强连通的。
4) 当G是一个邻接矩阵或链接相邻表时，分析程序的时间复杂性。

12.4 抽象数据类型Graph 和Digraph

抽象数据类型*Graph*专指无向图而抽象数据类型*Digraph*专指有向图。ADT 12-1和12-2的抽象数据类型描述只列出了图操作中的一小部分。在后面的讲述过程中，将不断地增加相应的操作。

抽象数据类型*WeightedGraph* 和*WeightedDigraph* 相似，只有*Add*操作的描述需要改变以反映与新添加边相关的权值。

ADT 12-1 **无向图的抽象数据类型描述**

抽象数据类型 *Graph* {

实例

顶点集合*V*和边集合*E*

操作

Create (*n*)：创建一个具有*n* 个顶点、没有边的无向图

Exist (*i*, *j*)：如果存在边（*i*, *j*）则返回true，否则返回false

Edges ()：返回图中边的数目

Vertices ()：返回图中顶点的数目

Add (*i*, *j*)：向图中添加边（*i*, *j*）

Delete (*i*, *j*)：删除边（*i*, *j*）

Degree (*i*)：返回顶点*i* 的度

InDegree (*i*)：返回顶点*i* 的度

OutDegree (*i*)：返回顶点*i* 的度

ADT 12-2 **有向图的抽象数据类型描述**

抽象数据类型 *Graph* {

实例

顶点集合*V*和边集合*E*

操作

Create(*n*)：创建一个具有*n*个顶点、没有边的有向图

Exist(*i*, *j*)：如果存在边（*i*, *j*）则返回true，否则返回false

Edges()：返回图中边的数目

Vertices()：返回图中顶点的数目

Add(*i*, *j*)：向图中添加边（*i*, *j*）

Delete(*i*, *j*)：删除边（*i*, *j*）

Degree(*i*)：返回顶点*i* 的度

InDegree(*i*)：返回顶点*i*的入度

OutDegree(*i*)：返回顶点*i*的出度

练习

8. 请给出加权无向图*WeightedGraph* 的ADT 描述。
9. 请给出加权有向图*WeightedDigraph* 的ADT 描述。

12.5 无向图和有向图的描述

无向图和有向图最常用的描述方法都是基于邻接的方式：邻接矩阵，邻接压缩表和邻接链表。

12.5.1 邻接矩阵

一个*n*顶点的图*G*=(*V*,*E*)的邻接矩阵（adjacency matrix）是一个*n*×*n* 矩阵*A*，*A*中的每一个元素是0或1。假设*V*={1, 2, ..., *n*}。如果*G*是一个无向图，那么*A*中的元素定义如下：

$$A(i,j)=\begin{cases}1 & \text{如果 } (i,j)\in E \text{ 或 } (j,i)\in E\\ 0 & \text{其它}\end{cases} \qquad (12\text{-}1)$$

如果G是有向图，那么A中的元素定义如下：

$$A(i,j)=\begin{cases}1 & \text{如果 } (i,j)\in E\\ 0 & \text{其它}\end{cases} \qquad (12\text{-}2)$$

图12-1的邻接矩阵如图12-9所示。

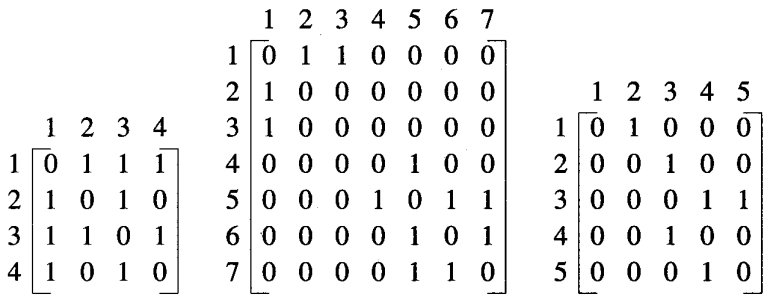


图12-9 图12-1对应的邻接矩阵

从（12-1）和（12-2）中可以得到如下结论：

1) 对于n 顶点的无向图，有$A(i,i)=0$，$1\le i\le n$。

2) 无向图的邻接矩阵是对称的，即$A(i,j)=A(j,i)$，$1\le i\le n$，$1\le j\le n$。

3) 对于n 顶点的无向图，有 $\sum_{j=1}^{n}A(i,j)=\sum_{j=1}^{n}A(j,i)=d_i$（$d_i$ 是顶点i 的度）。

4) 对于n 顶点的有向图，有 $\sum_{j=1}^{n}A(i,j)=d_i^{out}$ $\sum_{j=1}^{n}A(j,i)=d_i^{in}$，$1\le i\le n$。

1. 将邻接矩阵映射到数组

使用映射$A(i,j)=a[i][j]$可以将$n\times n$的邻接矩阵映射到一个$(n+1)\times(n+1)$ 的整型数组a 中。如果*sizeof* (*int*) 等于2个字节，映射的结果需要$2(n+1)^2$字节的存储空间。另一种方法是，采用 $n\times n$ 数组$a[n][n]$ 和映射$A(i,j)=a[i-1][j-1]$。这种映射需要$2n^2$字节，比前一种减少了$4n+2$个字节。

注意到所有对角线元素都是零而不需要储存，所以还可以进一步减少 $2n$字节的存储空间。当把对角线元素去掉后，可得到一个上（或下）三角矩阵（见 4.3.3节）。这些矩阵可以被压缩到一个$(n-1)\times n$的矩阵中，如图12-10所示。图中的阴影部分是原邻接矩阵的下三角部分。

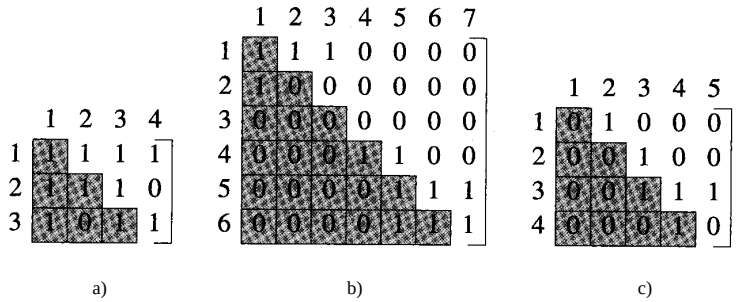


图12-10 图12-9去掉对角线元素后的邻接矩阵

注意到每个邻接矩阵元素只需要 1位的存储空间，而每个数组元素需要 16位，所以还可以

进一步缩减存储空间。通过使用 unsigned int 类型的数组a，可以将A中的16个元素压缩到a 中的1个元素中，因此，对存储空间的需求将变为$n(n-1)/8$字节。在空间减少的同时，存储和检索邻接矩阵中元素所需要的时间将增加。

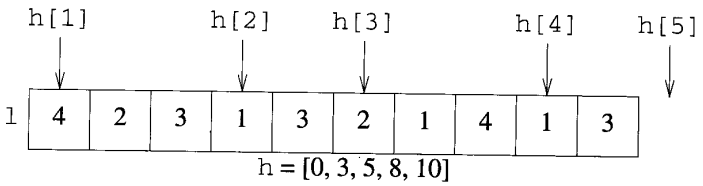
对于无向图，邻接矩阵是对称的（见4.3.5），因此只需要存储上三角（或下三角）的元素，所需空间仅为（n^2-n）/ 2位。

2. 时间需求

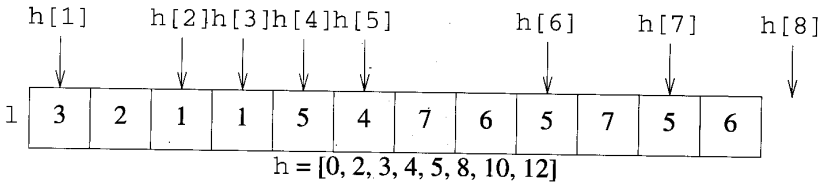
使用邻接矩阵时，需要用$\Theta(n)$时间来确定邻接至或邻接于一个给定节点的集合。寻找图中的边数也需要$\Theta(n)$的时间。另外，增加或删除一条边需要$\Theta(1)$时间。

12.5.2 邻接压缩表

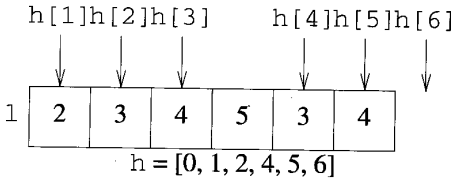
在G（$G=(V,E)$，$|V|=n$，$|e|=e$）的邻接压缩表（packed-edjacency-list）的定义中，使用了两个一维数组$h[0:n+1]$和$l[0:x]$，如果G是有向图，则$x=e-1$；如果G是无向图，则$x=2e-1$。首先将所有邻接于顶点1的顶点加入到l 中，然后将所有邻接于顶点2的顶点加入到l 中，再将邻接于顶点3的顶点加入到l 中，这样一直进行下去。（如果i 和j 是无向图的邻接顶点，那么i 邻接于j，j 邻接于i）。下面构造h，使得当$h[i]<h[i+1]$时，邻接于顶点i 的所有顶点的位置是$l[h[i]]$，$l[h[i]+1]$，...，$l[h[i+1]-1]$；当$h[i]\geq h[i+1]$时，没有邻接于i 的顶点。则$l[h[i]]$，$l[h[i]+1]$，...，$l[h[i+1]-1]$是顶点i 的邻接压缩表。顶点在表中的次序并不重要。图 12-11给出了图12-1所对应的邻接压缩表。



a)



b)



c)

图12-11 图12-1对应的邻接压缩表

对于无向图，h 的取值范围是0～2e。因为这个范围只能描述2e+1 个互不相同的值，所以

每一个$h[i]$最多需要$[\log(2e+1)]$位。数组l 的范围为1～n，因此l 的每个元素最多需$\lceil \log n \rceil$位。所以，每一个 n顶点e条边的无向图的邻接压缩表所需要的总的存储空间最多为 $(n+1)\lceil \log(2e+1) \rceil + 2e\lceil \log n \rceil = O((n+e)\log n)$。

时间需求

当e 远远小于n^2 时，邻接压缩表需要的空间远远小于邻接矩阵需要的空间。如果 G为无向图，顶点i 的度是$h[i+1]-h[i]$，G中边的数目是$h[n+1]/2$。使用邻接表可以比使用邻接矩阵更容易确定这些数量。增加或删除一条边需要$O(n+e)$ 的时间。

12.5.3 邻接链表

在邻接链表（linked-edjacency-list）中，邻接表是作为链表保存的，可以用类 Chain<int>（见程序3-8）来实现。另外，可使用一个 Chain<int> 类型的头节点数组h 来跟踪这些邻接表。$h[i]$.first 指向顶点i 的邻接表中的第一个节点。如果x 指向链表$h[i]$ 中的一个节点，那么$(i, x\rightarrow$ data) 是图中的一条边。图12-12给出了一些邻接链表。

假设每个指针和整数均为2字节长，则一个n 顶点图的邻接链表所需要的空间为2(n+m+1)，其中对于无向图，$m=2e$；而对于有向图，$m=e$。l 可从大小为$n+2$的数组h 中得到，如果用$h[i-1]$指向顶点i 的链表，可以不需要l。

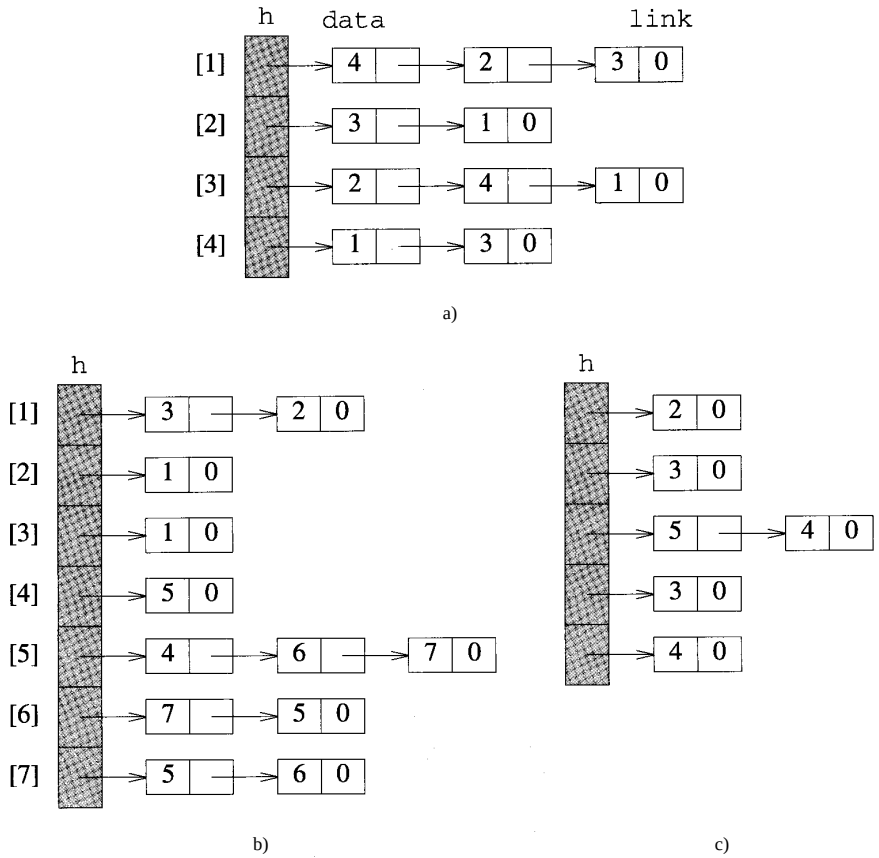


图12-12 图12-1对应的邻接链表

时间需求

邻接链表便于进行边的插入和删除操作。确定邻接表中顶点的数目所需要的时间与表中顶点的数目成正比。

练习

10. 请为图12-5和12-8a 提供下列描述：

1) 邻接矩阵。

2) 邻接压缩表。

3) 邻接链表。

11. a 是一个$(n\text{-}1)\times n$ 的数组，用来描述一个n 顶点图的邻接矩阵A。a 中没有描述矩阵的对角线（如图12-10所示）。编写两个函数Store 和Retrieve 分别存储和搜索$A(i, j)$ 的值，每个函数的复杂性应为$\Theta(1)$。

12. 用全邻接矩阵（见图12-9）完成练习11，矩阵的16个元素可压缩成数组a 的一个元素，其中a 是unsigned int类型的一维数组。

13. 用无向图完成练习11，仅在一维数组a 中存储无向图的下三角矩阵。假设a 中的每个元素非0即1。

14. 用无向图完成练习11，无向图的下三角矩阵存储在一个unsigned int类型的一维数组a中，假设a 中的每一个元素描述下三角矩阵中的16个元素。

15. 假设用一个$n\times n$ 的数组a 来描述一个有向图的$n\times n$ 邻接矩阵

1) 编写一个函数确定一个顶点的出度，函数的复杂性应为$\Theta(n)$。

2) 编写一个函数确定一个顶点的入度，函数的复杂性应为$\Theta(n)$。

3) 编写一个函数确定图中边的数目，函数的复杂性应为$\Theta(n^2)$。

16. 假设用邻接压缩表描述一个无向图

1) 编写一个函数删除边(i, j)。代码的复杂性是多少？

2) 编写一个函数增加边(i, j)。代码的复杂性是多少？

17. 对有向图完成练习16。

18. 用邻接链表完成练习15。

19. 用邻接链表完成练习16。

20. G是一个n 顶点，e 条边的无向图。e 至少是多少时，G的邻接矩阵所占用的空间才会比邻接压缩表所占用的空间少？

21. 对有向图G完成练习20。

12.6 网络描述

将图和有向图的描述进行简单扩充就可得到网络的描述，无论它是加权的无向图还是有向图。类似于邻接矩阵的描述，可用一个矩阵 C来描述耗费邻接矩阵（cost-adjacency-matrix）。如果$A(i, j)$ 是1，那么$C(i, j)$ 是相应边的耗费（或权）；如果$A(i, j)$ 是0，那么相应的边不存在，$C(i, j)$ 等于某些预置的值 NoEdge。选择NoEdge是为了便于区分边是否存在。一般来说，NoEdge的值被设为无穷大。图12-13给出了图12-1的耗费邻接矩阵。符号∞代表NoEdge的值。

用（顶点, 权）替换l 中的每一个入口，可以从相应的无权图或无权有向图中得到网络的邻接压缩表。例如，相应于图12-1c 的数组l 如图12-11c 所示，每条边的权值由图12-13c 中的耗

费邻接矩阵给出。相应的加权有向图的数组 *l* 为[(2,8)，(3,3)，(4,2)，(5,7)，(3,6)，(4,5)]。数组 *h* 不变。

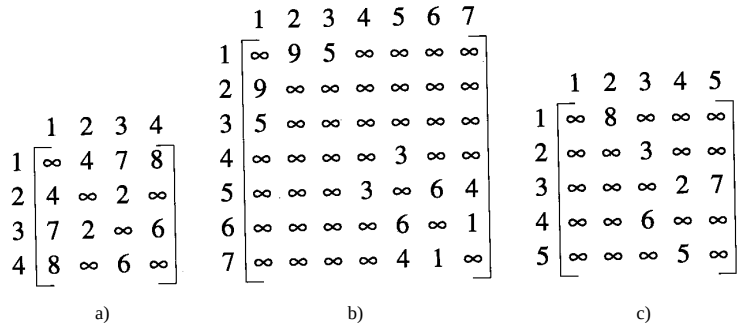


图12-13 图12-1对应的可能的耗费邻接矩阵

使用Chain<GraphNode>类型的链表，可以从相应的图的邻接表描述中得到网络的邻接表描述，其中GraphNode 包括两个部分：vertex和weight。图12-14给出了与图 12-13a 的耗费邻接矩阵相对应的网络描述。图中每个节点的第一部分是顶点，第二部分是权。

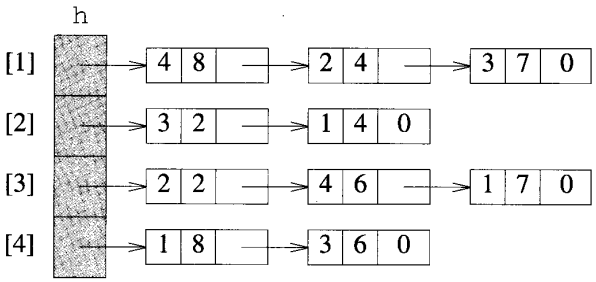


图12-14 图12-13a对应的网络的邻接链表

练习

22. 给出相应于图12-1a 和b 中耗费邻接矩阵的网络邻接压缩表。
23. 给出相应于图12-1a 和b 中耗费邻接矩阵的网络邻接链表。

12.7 类定义

12.7.1 不同的类

无权有向图和无向图可以看作每条边的权是 1的加权有向图和无向图。因此 12.4节中定义的抽象数据类型(*Graph*，*Digraph*，*WeightedGraph*，*WeightedDigraph*) 是一个更普通的抽象数据类型*Network*的子类。

对于12.4节的四种抽象数据类型中的每一种，考虑 12.5和12.6节所讨论的三种描述方法。用C++类把抽象数类型和描述方法联系起来，可以得到 12个类。本节只给出其中的八个类，其余四个对应于压缩描述的类留作练习（练习33至36）。

本节要讨论的八个类是 AdjacencyGraph，AdjacencyWGraph(矩阵描述的加权图)，AdjacencyDigraph，AdjacencyWDigraph，LinkedGraph，LinkedWGraph，LinkedDigraph和LinkedWDigraph。

4种抽象数据类型中的若干对类型之间存在IsA关系，例如，无向图可以看作边(*i*, *j*) 和边(*j*, *i*) 都存在的有向图；也可以看作所有边的权均为1 的加权图；或者看作所有边的权为1，若边(*i*,

j) 存在，则边(*j*, *i*) 也存在的加权有向图。类似地，有向图也可以看作所有边的权均为 1 的加权有向图。

利用这些关系可以很容易地设计这八个类，因为可以从其中的一个类派生出另一个类。虽然存在很多IsA关系，但只能利用其中少数几个关系。很自然地，可以从一个邻接矩阵类派生另一个邻接矩阵类，从一个链接类派生另一个链接类。图 12-15中的有向无环图给出了各个类之间的派生层次。例如， AdjacencyGraph类可以由AdjacencyWGraph派生而来。对于链接类，引入另外一个类LinkedBase来描述链表数组。利用类LinkedBase可以避免这四个链接类中公用函数的重复。而对于邻接类，不需要额外定义这样的类，因为邻接类有一个共同的根类——AdjacencyWDigraph，在这个根类中定义了所有类的公用函数。

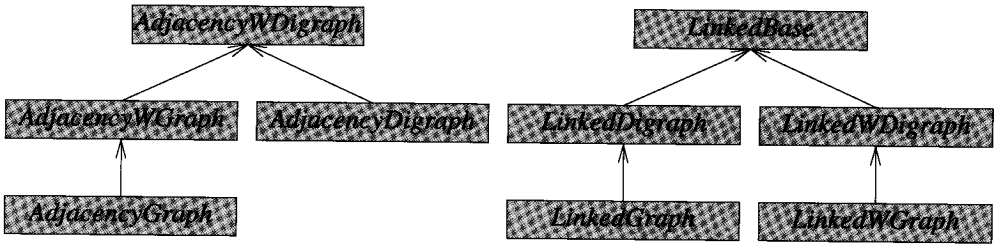


图12-15 类的派生层次

12.7.2 邻接矩阵类

邻接矩阵类的根是AdjacencyWDigraph，因此从这个类开始。程序12-1给出了类的描述。程序中，先用程序1-13中函数Make2DArray 为二组数组a 分配空间，然后对数组a 初始化，以描述一个n 顶点、没有边的图的邻接矩阵，其复杂性为$\Theta(n^2)$。该代码没有捕获可能由Make2DArray引发的异常。在析构函数中调用了程序1-14中的二维数组释放函数Delete2DArray。

程序12-1 加权有向图的耗费邻接矩阵

```
template<class T>
class AdjacencyWDigraph {
  friend AdjacencyWGraph<T>;
  public:
    AdjacencyWDigraph (int Vertices = 10, T noEdge = 0);
    ~AdjacencyWDigraph() {Delete2DArray(a,n+1);}
    bool Exist(int i, int j) const;
    int Edges()  const {return e;}
    int Vertices() const {return n;}
    AdjacencyWDigraph<T>& Add (int i, int j, const T& w);
    AdjacencyWDigraph<T>& Delete(int i, int j);
    int OutDegree(int i) const;
    int InDegree(int i) const;
  private:
    T NoEdge;  // 用于没有边存在的情形
    int n;     // 顶点数目
    int e;     // 边数
```

```
    T **a;     // 二维数组
};

template<class T>
AdjacencyWDigraph<T>::AdjacencyWDigraph(int Vertices, T noEdge)
{// 构造函数
  n = Vertices;
  e = 0;
  NoEdge = noEdge;
  Make2DArray(a, n+1, n+1);
  //初始化为没有边的图
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      a[i][j] = NoEdge;
}

template<class T>
bool AdjacencyWDigraph<T>::Exist(int i, int j) const
{// 边(i, j)存在吗?
  if (i < 1 || j < 1 || i > n || j > n || a[i][j] == NoEdge) return false;
  return true;
}

template<class T>
AdjacencyWDigraph<T>& AdjacencyWDigraph<T> ::Add(int i, int j, const T& w)
{// 如果边 (i,j) 不存在，则将该边加入有向图中
  if (i < 1 || j < 1 || i > n ||
      j > n || i == j || a[i][j] != NoEdge)
      throw BadInput();
  a[i][j] = w;
  e++;
  return *this;
}

template<class T>
AdjacencyWDigraph<T>& AdjacencyWDigraph<T> ::Delete(int i, int j)
{//删除边(i,j).
  if (i < 1 || j < 1 || i > n || j > n || a[i][j] == NoEdge)
      throw BadInput();
  a[i][j] = NoEdge;
  e--;
  return *this;
}

template<class T>
int AdjacencyWDigraph<T>::OutDegree(int i) const
{// 返回顶点 i的出度
```

```
  if (i < 1 || i > n) throw BadInput();
  // 计算顶点 i的出度
  int sum = 0;
  for (int j = 1; j <= n; j++)
    if (a[i][j] != NoEdge) sum++;
  return sum;
}

template<class T>
int AdjacencyWDigraph<T>::InDegree(int i) const
{// 返回顶点 i的入度
  if (i < 1 || i > n) throw BadInput();
  // 计算顶点 i的入度
  int sum = 0;
  for (int j = 1; j <= n; j++)
    if (a[j][i] != NoEdge) sum++;
  return sum;
}
```

函数Exist 的代码不能区分下面两种情况：1) i 和 / 或j 是否为有效顶点；2) 边(i, j) 是否存在。可以对代码进行修改，使前一种情况引发一个异常 OutOfBounds。对于Add和Delete函数也可以如法炮制。所有代码简单易懂，所以将不再作进一步说明。 Exist，Edges，Add和Delete的复杂性均为$\Theta(1)$，而OutDegree和InDegree的复杂性为$\Theta(n)$。

剩下的三种邻接矩阵类在程序12-2到程序12-4中给出。

程序12-2 加权图的耗费邻接矩阵

```
template<class T>
class AdjacencyWGraph : public AdjacencyWDigraph<T> {
  public:
    AdjacencyWGraph(int Vertices = 10, T noEdge = 0) : AdjacencyWDigraph<T>(Vertices, noEdge) {}
    AdjacencyWGraph<T>& Add(int i, int j, const T& w)
      {AdjacencyWDigraph<T>::Add(i,j,w);
       a[j][i] = w;
       return *this;}
    AdjacencyWGraph<T>& Delete(int i, int j)
      {AdjacencyWDigraph<T>::Delete(i,j);
       a[j][i] = NoEdge;
       return *this;}
    int Degree(int i) const {return OutDegree(i);}
};
```

程序12-3 有向图的邻接矩阵

```
class AdjacencyDigraph : public AdjacencyWDigraph<int> {
  public:
    AdjacencyDigraph(int Vertices = 10) : AdjacencyWDigraph<int>(Vertices, 0) {}
```

```
    AdjacencyDigraph& Add(int i, int j)
       {AdjacencyWDigraph<int>::Add(i,j,1);
        return *this;}
    AdjacencyDigraph& Delete(int i, int j)
       {AdjacencyWDigraph<int>::Delete(i,j);
        return *this;}
};
```

程序12-4 图的邻接矩阵

```
class AdjacencyGraph : public AdjacencyWGraph<int>
{
  public:
    AdjacencyGraph(int Vertices = 10) : AdjacencyWGraph<int>(Vertices, 0) {}
    AdjacencyGraph& Add(int i, int j)
       {AdjacencyWGraph<int>::Add(i,j,1);
        return *this;}
    AdjacencyGraph& Delete(int i, int j)
       {AdjacencyWGraph<int>::Delete(i,j);
        return *this;}
};
```

12.7.3 扩充Chain类

在链表描述中，对象被描述为一个链表数组，且每个链表的类型为 Chain类（见程序3-8）。我们所需要的一种链表操作目前尚未定义，因此下面增加该操作。

新的共享成员函数（见程序12-5）删除一个具有指定关键值的元素。程序在链表中搜索与x的关键值相同的元素（假设操作符 !=已被重载用于比较两个元素的关键值）。如果找到了匹配的元素，将它从链表中删除，并返回到x 中。

程序12-5 从链表中删除元素

```
template<class T>
Chain<T>& Chain<T>::Delete(T& x)
{// 删除与 x匹配的元素
 // 如果不存在相匹配的元素，则引发异常 BadInput
  ChainNode<T> *current = first,
               *trail = 0; // 指向current之后的节点

  //搜索匹配元素
  while (current && current->data != x) {
    trail = current;
    current = current->link;}
  if (!current) throw BadInput(); // 不存在匹配元素

  //在节点current中找到匹配元素
```

```
    x = current->data; // 保存匹配元素

    // 从链表中删除current 节点
    if (trail) trail->link = current->link;
    else first = current->link;

    delete current;  // 释放节点
    return *this;
  }
```

12.7.4 类LinkedBase

如图12-15所示，无权和加权图的派生路径之所以不同，其原因在于加权有向图和无向图的链表节点中有一个权值域，而无权有向图和无向图中则没有。对于后者，使用 int 类型的链节点就足够了；而对于前者，链节点必须包含一个权值域和一个顶点域。尽管节点结构存在这种差别，但某些基本函数的代码仍然是一样的。因此，引入一个新类LinkedBase（见程序12-6），它包含了构造函数、析构函数、Edges和OutDegree函数。

构造函数为链表数组分配空间。h[i] 是顶点i 的链表，1≤i≤n。析构函数释放这些空间。构造函数、析构函数及Edges的复杂性均为$\Theta(1)$，OutDegree(i)的复杂性为$\Theta(d_i^{out})$。

程序12-6 邻接链描述的基类

```
template<class T>
class LinkedBase {
  friend class LinkedDigraph;
  friend class LinkedGraph;
  friend LinkedWDigraph<int>;
  friend LinkedWGraph<int>;
  public:
    LinkedBase(int Vertices = 10)
      {n = Vertices;
       e = 0;
       h = new Chain<T> [n+1];}
    ~LinkedBase() {delete [] h;}
    int Edges() const {return e;}
    int Vertices() const {return n;}
    int OutDegree(int i) const
      {if (i < 1 || i > n) throw OutOfBounds();
       return h[i].Length();}
  private:
    int n;      //顶点数
    int e;      // 边数
    Chain<T> *h; // 邻接矩阵
};
```

12.7.5 链接类

前面所定义的四种链接类是LinkedBase 类的友员。程序12-7给出了LinkedDigraph 类。程序中增加了一个保护成员函数 AddNoCheck，在添加一条边时该函数不作任何检查。之所以增加这个函数是因为在使用邻接表时，有效性检查的开销很大，既然能够知道所增加的边是有效的，所以省略了这种检查。 Exist(i, j) 和Add(i, j) 的复杂性为$\Theta(d_i^{out})$，而AddNoCheck 的复杂性为$\Theta(1)$，Delete(i, j) 的复杂性为$\Theta(d_i^{out}+d_j^{out})$，InDegree 的复杂性为$\Theta(n+e)$。

程序12-7 有向图的邻接链表

```
class LinkedDigraph : public LinkedBase<int> {
  public:
    LinkedDigraph(int Vertices = 10) : LinkedBase<int> (Vertices) {}
    bool Exist(int i, int j) const;
    LinkedDigraph& Add(int i, int j);
    LinkedDigraph& Delete(int i, int j);
    int InDegree(int i) const;
  protected:
    LinkedDigraph& AddNoCheck(int i, int j);
};

bool LinkedDigraph::Exist(int i, int j) const
{// 边 (i,j)存在吗?
  if (i < 1 || i > n) throw OutOfBounds();
  return (h[i].Search(j)) ? true : false;
}

LinkedDigraph& LinkedDigraph::Add(int i, int j)
{// 把边 (i,j) 加入到图中
  if (i < 1 || j < 1 || i > n || j > n || i == j || Exist(i, j)) throw BadInput();
  return AddNoCheck(i, j);
}

LinkedDigraph& LinkedDigraph::AddNoCheck(int i, int j)
{// 增加边但不检查可能出现的错误
  h[i].Insert(0,j); // 把 j 添加到顶点 i 的表中
  e++;
  return *this;
}

LinkedDigraph& LinkedDigraph::Delete(int i, int j)
{// 删除边(i,j)
  if (i < 1 || i > n) throw OutOfBounds();
  h[i].Delete(j);
  e--;
  return *this;
}
```

```
int LinkedDigraph::InDegree(int i) const
{// 返回顶点 i的入度
  if (i < 1 || i > n) throw OutOfBounds();
  // 计算到达顶点 i的边
  int sum = 0;
  for (int j = 1; j <= n; j++)
    if (h[j].Search(i)) sum++;
  return sum;
}
```

程序12-8给出了类LinkedGraph，它是从类LinkedDigraph派生而来的。除函数InDegree外，其他所有函数的复杂性均与LinkedDigraph类中对应函数的复杂性相同。

程序12-8 LinkedGraph类

```
class LinkedGraph : public LinkedDigraph {
  public:
    LinkedGraph(int Vertices = 10) : LinkedDigraph (Vertices) {}
    LinkedGraph& Add(int i, int j);
    LinkedGraph& Delete(int i, int j);
    int Degree(int i) const {return InDegree(i);}
    int OutDegree(int i) const {return InDegree(i);}
  protected:
    LinkedGraph& AddNoCheck(int i, int j);
};

LinkedGraph& LinkedGraph::Add(int i, int j)
{// 向图中添加边 (i,j)
  if (i < 1 || j < 1 || i > n || j > n || i ==j || Exist(i, j)) throw BadInput();
      return AddNoCheck(i, j);
}

LinkedGraph& LinkedGraph::AddNoCheck(int i, int j)
{// 添加边(i,j), 不检查可能出现的错误
  h[i].Insert(0,j);
  try {h[j].Insert(0,i);}
  // 若出现异常, 则取消第一次插入，并引发同样的异常
  catch (...) {h[i].Delete(j); throw;}
  e++;
  return *this;
}

LinkedGraph& LinkedGraph::Delete(int i, int j)
{//删除边(i,j)
  LinkedDigraph::Delete(i,j);
  e++; // 补偿
  LinkedDigraph::Delete(j,i);
```

```
  return *this;
}
```

程序12-9给出了加权有向图的类定义。GraphNode类如程序12-10所示。InDegree的代码与LinkedDigraph类的代码相同，在程序12-9中不再给出。所有函数的复杂性与LinkedDigraph类对应函数的复杂性相同。现在，从LinkedWDigraph类可以派生出LinkedWGraph类(见练习32)。

程序12-9 加权有向图的邻接链表

```
template<class T>
class LinkedWDigraph : public LinkedBase<GraphNode<T> > {
  public:
    LinkedWDigraph(int Vertices = 10) : LinkedBase<GraphNode<T> > (Vertices) {}
    bool Exist(int i, int j) const;
    LinkedWDigraph<T>& Add(int i, int j, const T& w);
    LinkedWDigraph<T>& Delete(int i, int j);
    int InDegree(int i) const;
  protected:
    LinkedWDigraph<T>&
      AddNoCheck(int i, int j, const T& w);
};

template<class T>
bool LinkedWDigraph<T>::Exist(int i, int j) const
{// 存在边(i,j) 吗?
  if (i < 1 || i > n) throw OutOfBounds();
  GraphNode<T> x;
  x.vertex = j;
  return h[i].Search(x);
}

template<class T>
LinkedWDigraph<T>& LinkedWDigraph<T> ::Add(int i, int j, const T& w)
{// 添加边(i,j)
  if (i < 1 || j < 1 || i > n || j > n || i == j
      || Exist(i, j)) throw BadInput();
  return AddNoCheck(i, j, w);
}

template<class T>
LinkedWDigraph<T>& LinkedWDigraph<T> ::AddNoCheck(int i, int j, const T& w)
{// 添加边(i,j)，不检查可能出现的错误
  GraphNode<T> x;
  x.vertex = j; x.weight = w;
  h[i].Insert(0,x);
  e++;
  return *this;
```

```
}

template<class T>
LinkedWDigraph<T>& LinkedWDigraph<T> ::Delete(int i, int j)
{// 删除边(i,j)
  if (i < 1 || i > n) throw OutOfBounds();
  GraphNode<T> x;
  x.vertex = j;
  h[i].Delete(x);
  e--;
  return *this;
}

template<class T>
int LinkedWDigraph<T>::InDegree(int i) const
{// 返回顶点i的入度
  if (i < 1 || i > n) throw OutOfBounds();
  int sum = 0;
  GraphNode<T> x;
  x.vertex = i;
  // 检查所有的(j,i)
  for (int j = 1; j <= n; j++)
    if (h[j].Search(x)) sum++;
  return sum;
}
```

程序12-10 GraphNode类

```
template <class T>
class GraphNode {
  friend LinkedWDigraph<T>;
  friend LinkedWGraph<T>;
  friend Chain<T>;
  public:
    int operator !=(GraphNode<T> y) const
    {return (vertex != y.vertex);}
    void Output(ostream& out) const
    {out << vertex << " " << weight << " ";}
  private:
    int vertex;  // 边的第二个顶点
    T weight;    // 边的权重
};

template <class T>
ostream& operator<<(ostream& out, GraphNode<T> x)
  {x.Output(out); return out;}
```

练习

24. 编写一个输入无向图的函数 AdjacencyGraph::Input和一个输出函数 Output。假设输入内容包括顶点、边的数量以及边的集合。每条边由一对顶点给出。重载操作符 <<以便于输入无向图。

25. 编写一个输入有向图的函数 AdjacencyDigraph::Input和一个输出函数Output，重载操作符<<以便于输入有向图。

26. 编写一个输入无向网络的函数 AdjacencyWGraph::Input和一个输出函数Output，重载操作符<<和>>。

27. 编写一个输入有向网络的函数 AdjacencyWDigraph::Input和一个输出函数Output，重载操作符<<和>>。

28. 编写一个输入无向图的函数 LinkedGraph::Input和一个输出函数Output，重载操作符<<和>>。

29. 编写一个输入有向图的函数 LinkedDigraph::Input和一个输出函数Output，重载操作符<<和>>。

30. 编写一个输入无向网络的函数 LinkedWGraph::Input和一个输出函数Output，重载操作符<<和>>。

31. 编写一个输入有向网络的函数 LinkedWDigraph::Input和一个输出函数Output，重载操作符<<和>>。

32. 设计一个 C++类LinkedWGraph，用邻接链表描述加权无向图，从 LinkedWDigraph 类(见程序12-9)中派生此类。

33. 设计一个 C++类PackedAdjGraph，用邻接压缩表描述无向图，从 LinearList类(见程序3-1)中派生此类。

34. 设计一个 C++类PackedAdjWGraph，用邻接压缩表描述加权无向图，从LinearList类(见程序 3-1)中派生此类。

35. 设计一个 C++类PackedAdjDigraph，用邻接压缩表描述有向图，从LinearList类(见程序3-1) 中派生此类。

36. 设计一个C++类PackedAdjWDigraph，用邻接压缩表描述加权有向图，从 LinearList类(见程序3-1) 中派生此类。

12.8 图的遍历

12.8.1 基本概念

不论采用哪一种图类编写应用程序，都需要沿着矩阵的一行或多行向下移动，或者沿着一个或多个链表向下移动，实现这种移动的函数称为遍历器（ iterator)。对于类Chain，定义了一个附加类ChainIterator (见程序3-18)，它提供了从链表的一个元素移到下一个元素的函数。可以在图类中引入相同策略并定义一些新的、能够提供遍历函数的图类。

不过在图类中，遍历器被嵌入到了类中。在练习 37，38和39中要求设计新的遍历器类。下面将给出遍历函数及其相应说明，这些函数只使用一个游标来跟踪矩阵的每一行或每个链表，因此，它们不支持需要多个游标的应用。

- Begin(i) 对于邻接表，返回顶点i 所对应表中的第一个顶点；对于邻接矩阵，返回邻接

于顶点i 的最小（即第一个）顶点。在两种情况中，如果没有邻接顶点，都将返回零值。

• NextVertex (i) 返回顶点i 对应邻接表中的下一个顶点或返回邻接于顶点i的下一个最小顶点。同样，当没有下一个顶点时函数返回零。

• InitializePos( ) 初始化用来跟踪每一个邻接表或(耗费)邻接矩阵每一行中当前位置的存储配置。

• DeactivatePos( ) 取消InitializePos( )所产生的存储配置。

12.8.2 邻接矩阵的遍历函数

邻接矩阵的遍历函数可以作为 AdjacencyWDigraph类的一个共享函数来加以实现。由于其他3种邻接类都是从这个类派生出来的，因此它们可以从 AdjacencyDigraph类中继承该函数。

由于可能处在邻接矩阵不同行的不同位置，因此用一个数组 pos来记录每一行中的位置，这个变量是AdjacencyWDigraph的私有成员，定义如下：

```
int *pos;
```

遍历函数的代码见程序12-11。

程序12-11 邻接矩阵的遍历函数

```
void InitializePos() {pos = new int [n+1];}

void DeactivatePos() {delete [] pos;}

template<class T>
int AdjacencyWDigraph<T>::Begin(int i)
{//返回第一个与顶点 i邻接的顶点
if (i < 1 || i > n) throw OutOfBounds();

  // 查找第一个邻接顶点
  for (int j = 1; j <= n; j++)
    if (a[i][j] != NoEdge) {// j 是第一个
      pos[i] = j;
      return j;}

  pos[i] = n + 1; // 没有邻接顶点
  return 0;
}

template<class T>
int AdjacencyWDigraph<T>::NextVertex(int i)
{// 返回下一个与顶点 i邻接的顶点
  if (i < 1 || i > n) throw OutOfBounds();

  // 寻找下一个邻接顶点
  for (int j = pos[i] + 1; j <= n; j++)
    if (a[i][j] != NoEdge) {// j 是下一个顶点
      pos[i] = j; return j;}

  pos[i] = n + 1; // 不存在下一个顶点
```

```
    return 0;
}
```

12.8.3 邻接链表的遍历函数

对于用邻接链表描述的图和网络，需要将程序 12-12中定义的共享成员函数 Initialize和DeactivatePos加入到LinkedBase类中，并且，还需要定义一个私有变量pos：

```
ChainIterator<T> *pos；
```

此外，还需要将余下的两个遍历函数加入到 LinkedDigraph和LinkedWDigraph类中，代码见程序12-13和12-14。

程序12-12 加入到LinkedBase类中

```
    void InitializePos()
      {pos = new ChainIterator<T> [n+1];}
    void DeactivatePos() {delete [] pos;}
```

程序12-13 邻接链表的遍历函数

```
int LinkedDigraph::Begin(int i)
{// 返回第一个与顶点 i邻接的顶点
  if (i < 1 || i > n) throw OutOfBounds();
  int *x = pos[i].Initialize(h[i]);
  return (x) ? *x : 0;
}

int LinkedDigraph::NextVertex(int i)
{// 返回下一个与顶点 i邻接的顶点
  if (i < 1 || i > n) throw OutOfBounds();
  int *x = pos[i].Next();
  return (x) ? *x : 0;
}
```

程序12-14 链接加权有向图的遍历函数

```
template<class T>
int LinkedWDigraph<T>::Begin(int i)
{// 返回第一个与顶点 i邻接的顶点
  if (i < 1 || i > n) throw OutOfBounds();
  GraphNode<T> *x = pos[i].Initialize(h[i]);
  return (x) ? x->vertex : 0;
}

template<class T>
int LinkedWDigraph<T>::NextVertex(int i)
{// 返回下一个与顶点 i邻接的顶点
  if (i < 1 || i > n) throw OutOfBounds();
  GraphNode<T> *x = pos[i].Next();
```

```
    return (x) ? x->vertex : 0;
}
```

练习

37. 为AdjacencyWDigraph 设计一个遍历器类，并加以测试。类应提供本节中所介绍的各种遍历功能。

38. 用LinkedDigraph 完成练习37。

39. 用LinkedWDigraph 完成练习37。

12.9 语言特性

12.9.1 虚函数和多态性

考察LinkedGraph::Add（见程序12-8）和LinkedDigraph::Add（见程序12-7）的代码，两段程序是一样的！由于LinkedGraph是从LinkedDigraph 派生而来，尝试一下将LinkedGraph::Add删除并从 LinkedDigraph中继承 Add成员函数。假设继承了这个成员，如果 G的类型是LinkedGraph，那么表达式G.Add(i,j)将调用所继承的函数 LinkedDigraph::Add，该函数又将调用函数LinkedDigraph::AddNoCheck，并将边加入到链表i 而不是链表j 中。LinkedDigraph::Add的行为是单态的（unimorphic），也就是说，不管LinkedDigraph::Add是作用于LinkedDigraph类的对象还是作用于LinkedDigraph派生类的对象，LinkedDigraph::Add的行为都是一样的，所调用的函数也完全相同。

由于LinkedDigraph有它自己的AddNoCheck函数，因此，当 LinkedDigraph::Add作用于LinkedGraph类的对象上时，希望它调用LinkedGraph::AddNoCheck；当它作用于 LinkedDigraph类的对象上时，希望它调用 LinkedDigraph::AddNoCheck。即，希望LinkedDigraph::Add的行为是多态的（polymorphic），被调用的函数取决于函数所作用的对象类型。通过在程序 12-7的第10行：

```
LinkedDigraph& AddNoCheck( int i , int j ) ;
```

之前加上关键字virtual，就能把LinkedDigraph::AddNoCheck变成一个虚函数（virtual function）。此外不需要其他任何修改，特别不需要在以下函数之前加入关键字 virtual：

```
LinkedDigraph& LinkedDigraph::AddNoCheck(int i, int j )
```

虚函数可用一种特殊的方式来处理。首先，考虑单一继承的情况，在此情况中，类既可以是基类，也可是另一个类的派生类。假设 A是B的一个派生类且A和B都至少包含一个虚函数。可为A构造一个虚函数表，对于A和B的每一个虚函数F，表中都有一个对应的指针，用来指向调用A.F时实际执行的函数F。

考察图12-16的派生结构，这是一个单一继承的例子，因为每一个类最多是从一个类派生而来。这里有4个类A,B,C,和D。类A从类B派生而来，类B从类C派生而来，而类C又从类D派生而来。方框中列出了类的成员函数。例如，类D包含虚函数f 和g（在图中，vf 是virtal f 的缩写，vg 是virtual g 的缩写）和非虚函数h。函数D::f和D::g仅输出字符D，而A::f 和A::g输出字符A 。虽然g 不是B中显式声明的虚函数，它仍然是一个虚函数，因为在 D中它是虚函数。一旦一个函数被声明为虚函数，它在所有的派生类中仍然是虚函数。

图12-16中方框的右边给出了类的虚函数表。表中包含一个指针，指向每个虚函数的实际

执行函数。由于D不是派生类，它的表中只包含D中所定义的虚函数。其他每个类的虚函数表可从其父类的虚函数表中构造出来。例如，C的虚函数表中包含C中新定义的虚函数以及对D中的虚函数进行修改后的函数。可采用同样的方法来构造A和B的虚函数表。

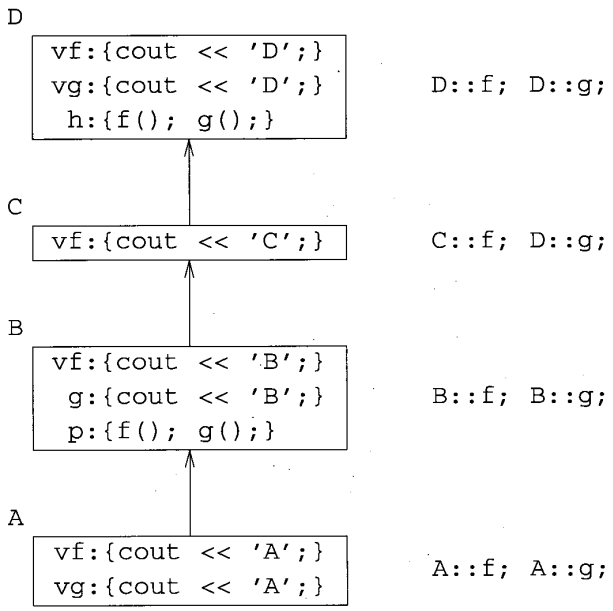


图12-16 虚函数

设a,b,c,和d 分别是A,B,C和D的对象，调用d.h()时需使用D的虚函数表来决定执行哪一个f和g，其输出为 DD。在执行c.h（）时，根据C的虚函数表来决定执行哪一个f 和g，其输出为CD。b.h ()、b.p( )、a.h( )和a.p( )所产生的输出分别是BB、BB、AA和AA。

下面考察多重继承的情况，在这种情况中，A是从两个或两个以上的类中派生而来。例如，考察图12-17a 中的派生有向图，图中A是从B和E中派生而来。现在A有两个虚函数表，第一个对应于派生路径ABCD，第二个对应于路径AEFG。当沿路径ABCD作用于A的对象时使用第一个表；当沿路径AEFG作用于A的对象时使用第二个表。采用与单一继承相同的过程从 B的虚函数表中构造第一条路径的虚函数表，同样，也可从E中构造第二条路径的虚函数表。

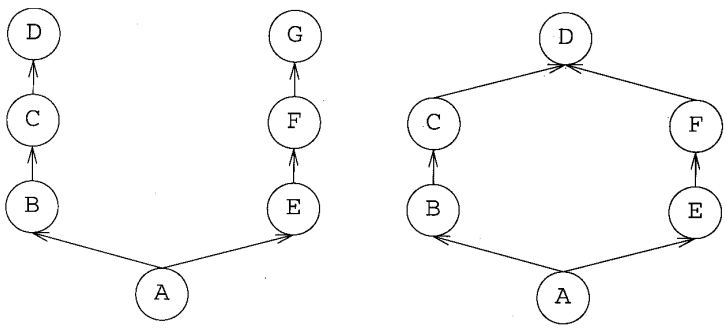


图12-17 派生层次

基于这种构造虚函数表的方法，当从A派生其他的类（比如X）时，需要从A的两个虚函数表中指定一个虚函数表来实现派生。按习惯，指定第一个基类的虚函数表。因此，如果 A采用如下语句定义：

```
class A：public B，public E
```

那么无论何时，当构造 A的其他派生类的虚函数表时，都将使用路径 ABCD的虚函数表作为A的虚函数表。

12.9.2 纯虚函数和抽象类

如下所示，如果虚函数被初始化为0：

```
virtual int f (int x,int y ) =0；
```

则称该虚函数为纯虚函数（pure virtual function）。在它的类说明语句中没有给出实现代码。包含一个纯虚函数的类称为抽象类（ abstract class）。如果A是一个抽象类，那么将不会有 A类型的对象，因为无法执行A.f()，但是可以有指向A类型对象的指针。

在12.7节中曾提到有向图，无向图，加权有向图和加权无向图都可以被看作一个网络。虽然没有定义相应的网络类，但我们为这四种特殊网络中的每一个网络都定义了两个类，因为必须在描述层对这四种网络进行区分。现在来定义一个抽象类 Network (见程序12-15)，这个类目前只包含纯虚函数，在后面的小节中将在其中加入一些非虚函数。

程序12-15 抽象类Network

```
class Network {
  public:
    virtual int Begin(int i) = 0;
    virtual int NextVertex(int i) = 0;
    virtual void InitializePos() = 0;
    virtual void DeactivatePos() = 0;
};
```

Network 中的函数可施加于所有四种特殊网络。直接使用遍历函数 Begin，NextVertex，InitializePos 和Deactivatepos 可能会失败，因为这些遍历函数是作为虚函数定义的，它们的具体实现取决于对象的类型。

12.9.3 虚基类

考察图12-17b 的派生结构，类B、C、D与图12-16中的类相同，将图12-16类A中g 的重定义省略掉就得到图12-17的A。类E和F分别与类B和C不同，区别仅在于E和F只输出字符E和F，而B和C只输出字符B和C。C和F都是从D中派生而来的。如果a 是A的对象，那么a.h ( ) 和a.p ( )含义不明。对于第1种情况，不知道应该使用 A的两个虚函数表中的哪一个(ABCD和AEFD)；对于第2种情况，不知道是调用a.B::p( )还是调用a.E::p( )。a.B::h( )调用D::h( )并使用A的ABCD路径虚函数表，输出结果为 AB；a.E::h( )产生输出AE。因此，调用的路径决定了使用 D的哪一个虚拟函数。

在许多应用中，当对 A的对象进行操作时，我们希望不管调用路径是什么，最终执行的是同样的虚函数。通过使 D成为C和F的共同虚基类可以做到这一点。不过，在图 12-17的例子中，简单地使D成为C和F的虚基类还不够，因为 D的虚函数f 和g 在路径ABCD和AEFG上

都被重新定义，因此不知道使用哪一个定义。要消除这种歧义，基类 D的每一个虚函数最多只能在一条路径上重新定义。例如，假定在类 B中重新定义f 来输出B且在A,B或C中没有重新定义g，在类E中重新定义了 g 来输出E且在A、E或F中没有重新定义 f，则在路径ABCD和AEFD的两个虚函数表中f 和g 都是相同的，调用a.B::h( )，a.E::h( )，a.E::p( )将产生相同的输出BE。

使D成为C和F的虚基类的另一个结果是A的对象仅含D的数据成员的一个拷贝。不管D是否是一个虚基类，C和F的对象都包含D的数据成员。同样，B的对象包含C和D的数据成员，而E的对象包含F和D的数据成员。如果D不是一个虚基类，A的对象将包含路过B的B,C和D的数据成员和路过E的E,F,D的数据成员。因此，在每一个A的对象中包含 2份D的数据成员。当D是一个虚基类时，每个A的对象将只包含D中的1份数据成员。

为了避免出现基类数据成员的多个副本，应将基类声明为虚基类。为使 Network成为AdjacencyWDigraph和LinkedBase的一个虚基类，将类的标题修改如下：

```
class AdjacencyWDigraph：virtual public Network
class  LinkedBase：virtual public Network
```

这些标题说明了AdjacencyWDigraph和LinkedBase是从Network中派生出来的并且Network是它们的虚基类。由于其他的类都是从 AdjacencyWDigraph和LinkedBase中派生出来的，因此这些类的成员也可以访问Network的成员。

在图的应用中，必须把 Network定义为AdjacencyWDigraph和 LinkedBase的一个虚基类，因为还需要定义另外一个类 Undirected，它也是从Network派生而来的。类Undirected中包含专用于无向图和网络的函数，因此，只能从无向图和网络类中访问这些函数。图 12-18给出了新的派生结构。

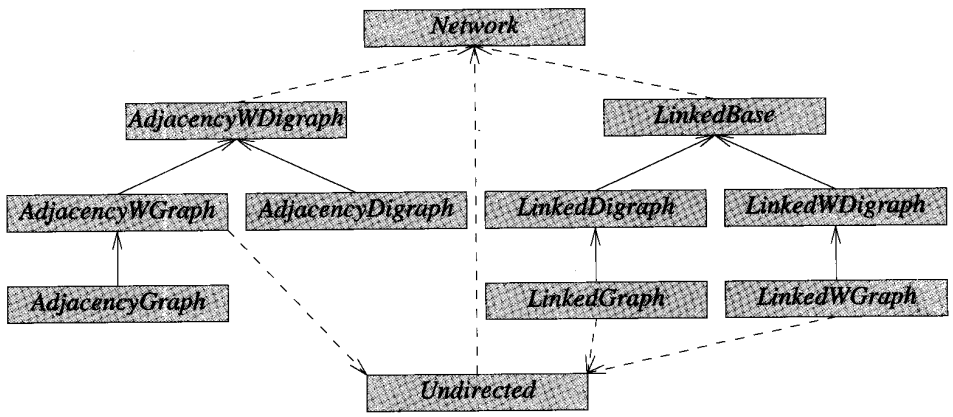


图12-18 包括Undirected的类派生层次

对于链接类，在 LinkedBase 中定义了纯虚函数 InitializePos，DeactivatePos，Begin 和Network 的NextVertex。假设G是LinkedGraph 类型并且 f 是Network 的一个成员，假设LinkedBase 和Undirected 是从Network 中以非虚拟方式派生出来的，则执行G.f ( ) 时将出现歧义，因为不知道使用LinkedGraph的哪一个虚函数表。如果执行G.g ( )，其中g是Undirected的一个成员，则g 将调用f，f 是Network 的一个成员，因此所使用的路径为LinkedGraph->Undirected->Network，但Network并未定义纯虚遍历函数。通过把 Network作为LinkedBase和Undirected的

一个虚基类，可以解决这种问题。

12.9.4 抽象类和抽象数据类型

可以用抽象类来说明抽象数据类型。到目前为止，抽象数据类型的描述都是以自然语言的形式给出的。考察抽象数据类型 *LinearList*(ADT2-1)，它是线性表数据结构的非正式描述。该描述给出了线性表必须支持的所有操作，但是无法强制某个具体的线性表实现必须满足这种要求。通过将*LinearList* 定义成一个抽象类并且要求所有的线性表都从这个抽象类派生而来，就可以使任一个线性表与相应的抽象数据类型保持一致。

对于线性表数据结构，可以使用程序12-16的抽象类定义。

程序12-16 抽象类AbstractList

```
template<class T>
class AbstractList {
public:
    virtual bool IsEmpty() const = 0;
    virtual int Length() const = 0;
    virtual bool Find(int k, T& x) const = 0; //查找第k个元素，并送入x
    virtual int Search(const T& x) const = 0; //返回x的位置
    virtual AbstractList<T>& Delete(int k, T& x) = 0; //删除第k元素，并将其放入x
    virtual AbstractList<T>& Insert(int k, const T& x) = 0; //紧靠第k个元素之后插入x
    virtual void Output(ostream& out) const = 0;
};
```

AbstractList 的所有成员函数都是虚函数。由于每一种线性表都必须从 AbstractList派生而来，因此在每种线性表中都必须实现所有的纯虚函数，否则，这种线性表将不可以拥有实例。

为了满足所有线性表都必须从相应抽象类中派生而来的要求，需要将 LinearList(见程序3-1)和Chain(见程序3-8)的类标题改为：

```
class LinearList：AbstractList<T>{
class Chain：AbstractList<T>{
```

此外，还需改变Insert 和Delete 函数的返回类型。对于LinearList 和Chain 类而言，这两个函数的返回类型与AbstractList 有关，因此，必须将语句：

```
LinearList<T>& Delete ( int k, T & x)；
LinearList<T>& Insert (int k, const T& x)；
```

和

```
Chain<T>& Delete (int k, T& x)；
Chain<T>& Insert (int k, const T& x)；
```

替换为：

```
AbstractList<T>& Delete (int k, T& x)；
AbstractList<T>& Insert (int k, const T& x)；
```

并且将语句：

```
LinearList<T> & LinearList<T>::Delete (int k, T& x)
LinearList<T> & LinearList<T>::Insert (int k, const T& x)
```

替换为：

AbstractList<T> & LinearList<T>::Delete (int k, T& x)
AbstractList<T> & LinearList<T>::Insert (int k, const T& x)

将语句：

Chain<T>& Chain<T>::Delete (int k, T& x)
Chain<T>& Chain<T>::Insert (int k, cont T& x)

替换为：

AbstractList<T> & Chain<T>::Delete(int k, T& x)
AbstractList<T> & Chain<T>::Insert (int k, const T& x)

除上述修改以外，其他地方不必做变动。

除了能强制与抽象数据类型描述保持一致以外，使用抽象类还允许编写一些共享函数作为抽象类的成员，而不必为每个派生类分别实现相应的函数。在后面几节中将看到相应的实例。

练习

40. 给出一个堆栈 (ADT5-1) 的抽象类定义 AbstractStack。修改 Stack(见程序 5-2)和 LinkedStack（见程序5-4），把它们作为AbstractStack 的派生类。试测试代码的正确性。

41. 给出一个队列(ADT6-1)的抽象类定义 AbstractQueue。修改 Queue(见程序 6-1)和 LinkedQueue（见程序6-4），把它们作为AbstractQueue 的派生类。试测试代码的正确性。

12.10 图的搜索算法

有关图、有向图和网络的函数实在太多，我们无法在这里一一列出。前面已经讨论了其中的一些函数（如寻找路径，寻找生成树，判断无向图是否连通），在后面的章节中还将讨论一些其他的函数。许多函数都要求从一个给定的顶点开始，访问能够到达的所有顶点。（当且仅当存在一条从v 到u 的路径时，顶点v 可到达顶点u。）搜索这些顶点的两种标准方法是宽度优先搜索和深度优先搜索。虽然这两种方法都很流行，但比较而言深度优先搜索使用频率更高一些（相应的效率也高一些）。

12.10.1 宽度优先搜索

考察图12-19a 中的有向图。判断从顶点1出发可到达的所有顶点的一种方法是首先确定邻接于顶点1的顶点集合，这个集合是{2,3,4}。然后确定邻接于{2,3,4}的新的顶点集合，这个集合是{5,6,7}。邻接于{5,6,7}的顶点集合为{8,9}，而不存在邻接于{8,9}的顶点。因此，从顶点1出发可到达的顶点集合为{1,2,3,4,5,6,7,8,9}。

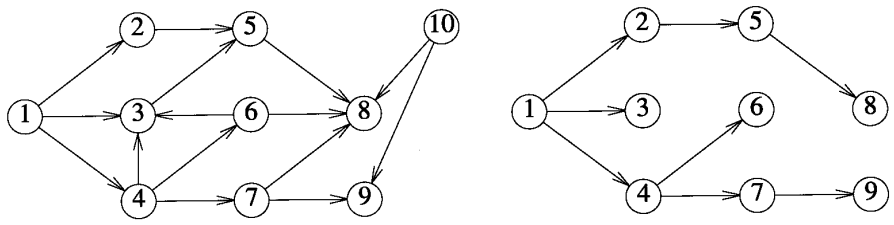


图12-19 宽度优先搜索

这种从一个顶点开始，识别所有可到达顶点的方法叫作宽度优先搜索（ Breadth-First Search, BFS）。这种搜索可使用队列来实现，图12-20给出了实现的伪代码。

```
//从顶点v 开始的宽度优先搜索
把顶点v标记为已到达顶点；
初始化队列Q，其中仅包含一个元素v;
while (Q不空) {
  从队列中删除顶点w;
  令u 为邻接于w 的顶点;
  while (u) {
    if ( u 尚未被标记) {
      把u 加入队列；
      把u 标记为已到达顶点；}
    u = 邻接于w 的下一个顶点；
   }
}
```

图12-20 BFS的伪代码

如果将图12-20的伪代码用于图12-19a 中，v=1。因此在第一个while循环中，顶点2,3,4都将被加入到队列中（假设是按此次序加入的）。在接下来的循环中，2被从队列中去掉，加入顶点5；然后删除3，之后再删除4，加入6和7；删除5并加入8；删除6后不增加；删除7后加入9，最后将8和9删除，队列成为空队列。过程终止时，顶点 1到9被加上已到达标记。图 12-19b 给出了访问过程中所经历的顶点和边构成的子图。

定理12-1 设N是一个任意的图、有向图或网络，v 是N 中的任意顶点。图12-20的伪代码能够标记从v 出发可以到达的所有顶点（包括顶点v）。

证明 这个定理的证明留作练习42。

12.10.2 类Network

根据图12-20的伪代码，在一个合适的高度，BFS的执行方式与是否正在处理一个图、有向图、加权图或加权有向图无关，也与所使用的描述方法无关。但是，为了实现下面的语句：

```
u = 邻接于w 的下一个顶点；
```

必须知道当前正在使用的图类。通过把 BFS函数作为Network类（见程序12-15）的一个成员函数以及利用图遍历器从一个邻接顶点到达下一个顶点，可以避免为每种图类分别编写不同的代码。

12.10.3 BFS的实现

BFS的代码（见程序12-17）与图12-20的伪代码非常相似。程序12-17假设初始时对于所有顶点有 reach[i]=0并且label≠0。算法终止时所有可到达顶点把对应的reach[i]设置为label。

程序12-17 BFS代码

```
void Network::BFS(int v, int reach[], int label)
```

```
{// 宽度优先搜索
  LinkedQueue<int> Q;
  InitializePos(); //初始化图遍历器数组
  reach[v] = label;
  Q.Add(v);
  while (!Q.IsEmpty()) {
    int w;
    Q.Delete(w);  // 获取一个已标记的顶点
    int u = Begin(w);
    while (u) {// 访问 w的邻接顶点
      if (!reach[u]) {// 一个未曾到达的顶点
        Q.Add(u);
        reach[u] = label;} // 标记已到达该顶点
      u = NextVertex(w); // 下一个与 w邻接的顶点
      }
    }
  DeactivatePos(); // 释放遍历器数组
}
```

12.10.4 BFS的复杂性分析

从顶点v出发，可到达的每一个顶点都被加上标记，且每个顶点只加入到队列中一次，也只从队列中删除一次，而且它的邻接矩阵中的行或它的邻接链表也只遍历一次。如果有 s 个顶点被标记，那么当使用邻接矩阵时，这些操作所需要的时间为 $\Theta(sn)$，而使用邻接链表时，所需时间为$\Theta(\sum_i d_i^{out})$。在后一种情况中，要对所有被标记的顶点 i 的出度求和。对于无向图／网络来说，顶点的出度就等于它的度。

现在，我们想知道，与为每一种描述都定制一个搜索函数相比，统一的 BFS函数的复杂性是多少。邻接矩阵和邻接链表的搜索函数分别见程序12-18和12-19。

程序12-18 邻接矩阵描述中BFS的直接实现

```
template<class T>
void AdjacencyWDigraph<T>::BFS (int v, int reach[], int label)
{// 宽度优先搜索
  LinkedQueue<int> Q;
  reach[v] = label;
  Q.Add(v);
  while (!Q.IsEmpty()) {
    int w;
    Q.Delete(w);  // 获取一个已标记的顶点
    // 对尚未标记的、邻接自w的顶点进行标记
    for (int u = 1; u <= n; u++)
      if (a[w][u] != NoEdge && !reach[u]) {
        Q.Add(u); // u 未被标记
        reach[u] = label;}
    }
}
```

程序12-19 链接图和有向图中BFS的直接实现

```
void LinkedDigraph::BFS(int v, int reach[], int label)
{// 宽度优先搜索
  LinkedQueue<int> Q;
  reach[v] = label;
  Q.Add(v);
  while (!Q.IsEmpty()) {
    int w;
    Q.Delete(w);  // 获取一个已标记的顶点
    // 使用指针p沿着邻接表进行搜索
    ChainNode<int> *p;
    for (p = h[w].First(); p; p = p->link) {
      int u = p->data;
      if (!reach[u]) {// 一个尚未到达的顶点
        Q.Add(u);
        reach[u] = label;}
      }
    }
}
```

对于用邻接矩阵描述的含有 50个顶点的无向完全图， Network::BFS的执行时间是AdjacencyWDigraph::BFS 的时间的2.6倍。对于链接描述，统一程序的执行时间是定制程序的4.5倍。

通过删除遍历函数Begin和NextVertex中一些不必要的有效性检查，可以减少统一程序和定制程序之间的差别。在BFS和以后可能定义的Network的其他一些成员中，仅当顶点参数有效时才会调用Begin和NextVertex，因此，可以改进函数Begin和NextVertex，使它们不用执行有效性检查。这样，执行因子4.5将变成3.6（对于50个顶点的完全图）。

如上所述，当使用 Network::BFS代替定制程序时，会有一个潜在的巨大代价。但是，Network::BFS也存在不少优点。例如，若使用Network::BFS，则这一份代码即可满足所有的图类描述，但若使用定制程序则必须提供多份不同的代码（每个图类对应一份）。因此，如果要设计新的图类描述并需要实现遍历函数，那么可以不加修改地使用已有的 Network成员。

12.10.5 深度优先搜索

深度优先搜索（Depth-First Search, DFS）是另一种搜索方法。从顶点v 出发，DFS按如下过程进行：首先将v 标记为已到达顶点，然后选择一个与v 邻接的尚未到达的顶点u，如果这样的u 不存在，搜索中止。假设这样的u 存在，那么从u 又开始一个新的DFS。当从u 开始的搜索结束时，再选择另外一个与v 邻接的尚未到达的顶点，如果这样的顶点不存在，那么搜索终止。而如果存在这样的顶点，又从这个顶点开始DFS，如此循环下去。

程序12-20给出了Network 类的共享成员DFS和私有成员dfs。在DFS的实现过程中，让u遍历v 的所有邻接顶点将更容易。

程序12-20 DFS代码

```
void Network::DFS(int v, int reach[], int label)
{// 深度优先搜索
```

```
    InitializePos(); // 初始化图遍历器数组
    dfs(v, reach, label); // 执行dfs
    DeactivatePos(); // 释放图遍历器数组
}

void Network::dfs(int v, int reach[], int label)
{// 实际执行深度优先搜索的代码
    reach[v] = label;
    int u = Begin(v);
    while (u) {// u邻接至 v
      if (!reach[u]) dfs(u, reach, label);
      u = NextVertex(v);}
}
```

用图12-19a 的有向图来测试DFS。如果v=1，那么顶点2,3和4成为u的候选。假设赋给u的第一个值是2，到达2的边是(1,2)，那么从顶点2 开始一次DFS，将顶点2标记为已到达顶点。这时u 的候选只有顶点5，到达5的边是(2,5)。下面又从5开始进行DFS，将顶点5标记为已到达顶点，根据边(5.8) 可知顶点8也是可到达顶点，将顶点8加上标记。从8开始没有可到达的邻接顶点，因此又返回到顶点5，顶点5也没有新的u，因此返回到顶点2，再返回到顶点1。

这时还有两个候选顶点：3和4。假设选中4，边(1,4)存在，从顶点 4开始DFS，将顶点 4标记为已到达顶点。现在顶点 3,6和7成为候选的u，假设选中6，当u=6时，顶点3是唯一的候选，到达3的边是(6,3)，从3开始DFS，并将3标记为已到达顶点。由于没有与 3邻接的新顶点，因此返回到顶点4，从4开始一个u=7的DFS，然后到达顶点9，没有与9邻接的其他顶点，这时回到1，没有与1邻接的其他顶点，算法终止。

对于DFS，可以给出一个类似于定理12-1的定理；DFS能够标记出顶点v 和所有可从v 点到达的顶点。

定理12-2　设*N*是一个任意的图、有向图或网， v 是*N*中任意顶点。对于所有可从顶点v 到达的顶点（包括v），调用DFS（v,reach,label）后，可得reached[i]=label。

证明　这个定理的证明留作练习43。

可以验证DFS与BFS有相同的时间和空间复杂性。不过，使DFS占用最大空间(递归栈空间)的图却是使BFS占用最小空间(队列空间)的图，而使BFS占用最大空间的图则是使DFS占用最小空间的图。图12-21中给出了能使DFS和BFS产生最好和最坏性能的图例。

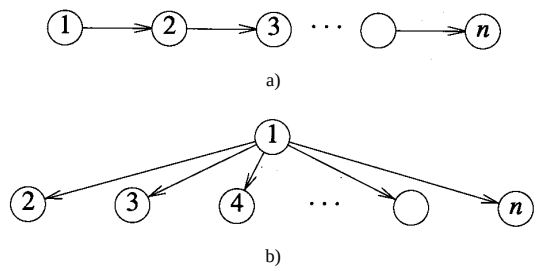


图12-21　产生最好和最坏空间复杂性的图例

a) DepthFirstSearch(1) 的最坏情况； BreadthFirst Search(1) 的最好情况

b) DepthFirstSearch(1) 的最好情况； BreadthFirst Search(1) 的最坏情况

练习

42. 证明定理12-1。
43. 证明定理12-2。
44. 编写PackedAdjGraph类的遍历函数。

12.11 应用

12.11.1 寻找路径

若从顶点v 开始搜索（宽度或深度优先）且到达顶点 w 时终止搜索，则可以找到一条从顶点v 到达顶点w 的路径（例12-1）。要实际构造这条路径，需要记住从一个顶点到下一个顶点的边。对于路径问题，所需要的边的集合已隐含在深度优先的递归过程中，因此可以很容易地利用深度优先策略设计一个寻找路径程序。完成顶点 w 的标记之后展开递归，可以反向建立起从w 到v 的路径。FindPath 的代码如程序12-21所示。这个程序要求把Vertices( )定义为Network 的一个虚拟成员。

FindPath的输入参数是路径的开始顶点 (v)和目标顶点 (w)。如果没有从 v到w的路径，FindPath返回false；否则返回true。当找到路径时，用参数length 返回路径长度（路径中边的条数），用顶点数组p[0：length] 返回路径，其中p[0]=v 且p[length]=w。

FindPath 首先检验v=w 情况。在这种情况下，返回一个长度为0的路径。如果v≠w，则调用图的遍历函数InitializePos，然后FindPath 产生并初始化一个数组reach，路径的DFS实际上是由Network的私有成员findpath完成的，当且仅当没有路径时，findpath返回false。函数findpath是一个修改过的DFS，它对标准的DFS作了如下两点修改：1) 一旦到达了目标顶点w，findpath将不再继续搜索可到达顶点；2) findpath 将开始顶点v 到当前顶点u 路径中的顶点记录到数组path 中。

Findpath与DFS具有相同的复杂性。

程序12-21 在图中寻找一个路径

```
bool Network::FindPath (int v, int w, int &length, int path[])
{// 寻找一条从 v 到 w的路径, 返回路径的长度，并将路径存入数组 path[0:length]
  //如果不存在路径，则返回 false

  //路径中的第一个顶点总是 v
  path[0] = v;
  length = 0;  // 当前路径的长度
  if (v == w) return true;

  //为路径的递归搜索进行初始化
  int  n = Vertices();
  InitializePos();  // 遍历器
  int *reach = new int [n+1];
  for (int i = 1; i <= n; i++)
    reach[i] = 0;

  // 搜索路径
```

```
    bool x = findPath(v, w, length, path, reach);

    DeactivatePos();
    delete [] reach;
    return x;
}

bool Network::findPath(int v, int w, int &length, int path[], int reach[])
{// 实际搜索v到w的路径，其中 v != w.
 // 按深度优先方式搜索一条到达w的路径
    reach[v] = 1;
    int u = Begin(v);
    while (u) {
      if (!reach[u]) {
        length++;
        path[length] = u; // 将u 加入path
        if (u == w) return true;
        if (findPath(u, w, length, path, reach))
          return true;
        // 不存在从 u 到 w的路径
        length--; //删除u
        }
      u = NextVertex(v);}
    return false;
}
```

12.11.2 连通图及其构件

通过从任意顶点开始执行DFS或BFS，并且检验所有顶点是否被标记为已到达顶点，可以判断一个无向图G是否连通。虽然这个算法只是直接检验 BFS中从开始顶点到其他每一个顶点之间是否存在一条路径，但对于判断两个顶点之间是否存在一条路径来说，这种检验已经足够了。假设i 是搜索的开始顶点并且搜索到达了图中的所有顶点，利用 i 到u 的反向路径及i 到v的路径，可以构造任意两个顶点u 和v 之间的路径。如果图不连通，则函数Connected（见程序12-22）返回false，否则返回true。由于连通的概念仅针对无向图和网络而言，因此，可以定义一个新类Undirected，函数Connected是Undirected的一个成员。图12-18给出了Undirected的类定义。

程序12-22 确定无向图是否连通

```
class Undirected : virtual public Network {
  public:
    bool Connected();
};

bool Undirected::Connected()
{// 当且仅当图是连通的，则返回true

  int n = Vertices();
```

```
    // 置所有顶点为未到达顶点
    int *reach = new int [n+1];
    for (int i = 1; i <= n; i++)
      reach[i] = 0;

    // 对从顶点1出发可到达的顶点进行标记
    DFS(1, reach, 1);

    // 检查是否所有顶点都已经被标记
    for (int   i = 1; i <= n; i++)
      if (!reach[i]) return false;
    return true;
  }
```

从顶点i 可到达的顶点的集合C与连接C中顶点的边称为连通构件（connected component）。图12-1b 的图中有2个连通构件，一个由顶点{1,2,3}和边{(1,2,), (1,3)}组成，另一个由其他顶点和边组成。在构件标识问题（component-labeling problem）中，对图中的顶点进行标识，当且仅当2个顶点属于同一构件时，分配给它们相同的标号。在图 12-1b 的例子中，顶点1和2标识为标号1，而剩下的顶点标识为标号2。

可以通过反复调用DFS或BFS算法来标识构件。从每一个尚未标识的顶点开始进行搜索，并用新的标号标识新到达的顶点。函数LabelComponents（见程序12-23）解决了构件标识问题。该函数返回图中构件的数目，并将构件标号返回至数组 L中。在程序12-23中，如果用DFS来取代BFS，也能得到相同结果。当用邻接矩阵来描述图时，程序 12-23的复杂性是$\Theta(n^2)$；而用邻接链表时，复杂性为$\Theta(n+e)$。

程序12-23 构件标识

```
int Undirected::LabelComponents(int L[])
{// 构件标识
 // 返回构件的数目， 并用 L[1:n]表示构件标号

  int n = Vertices();

  // 初始时，所有顶点都不属于任何构件
  for (int i = 1; i <= n; i++)
    L[i] = 0;

  int label = 0;  // 最后一个构件的ID
  // 识别构件
  for (int   i = 1; i <= n; i++)
    if (!L[i]) {// 未到达的顶点
      // 顶点 i 属于一个新的构件
      label++;
      BFS(i, L, label);} // 标记新构件

  return label;
}
```

12.11.3 生成树

在一个n 顶点的连通无向图中，如果从任一个顶点开始进行 BFS，那么从定理12-1可知，所有顶点都将被加上标记，并且在Network::BFS (见程序12-17) 的内层while 循环中正好有n-1个顶点是可到达的。在该循环中，若到达一个新顶点 u，则相应的边为(w,u)，这样边的数目正

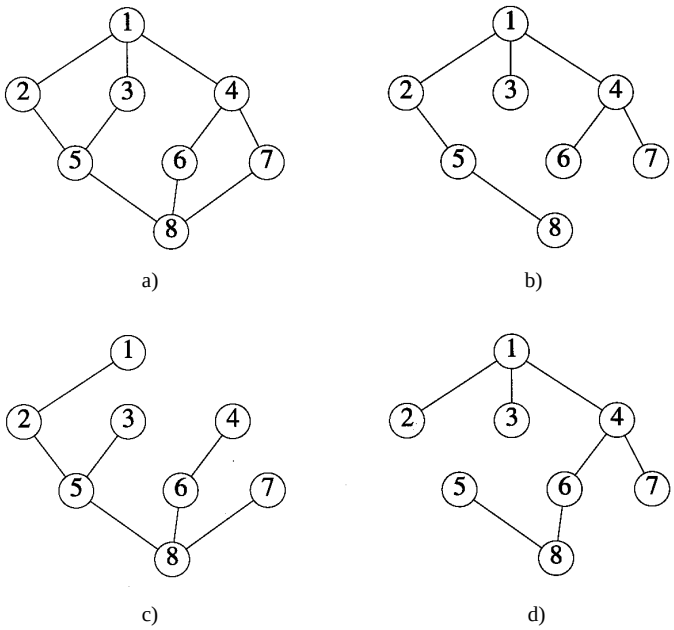


图12-22 图及其宽度优先生成树

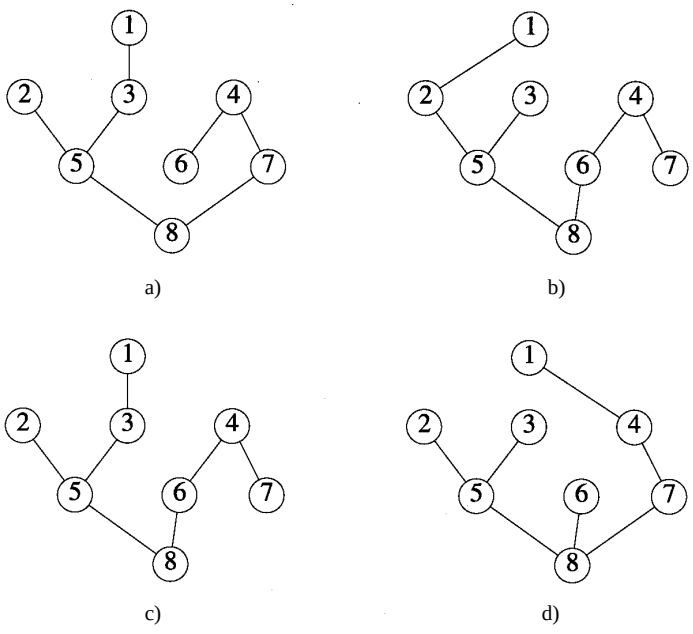


图12-23 图12-22a 的一些深度优先生成树

好是n-1。由于所得到的边的集合中包含一条从v 到图中其他每个顶点的路径，因此它构成了一个连通子图，该子图即为G的生成树。

考察图12-22a 中的图，如果从顶点1开始进行BFS，那么用来到达以前未到达顶点的边是{(1,2,)，(1,3)，(1,4)，(2,5)，(4,6)，(4,7)，(5,8)}，这个边集合即对应于图12-22b 中的生成树。

宽度优先生成树（breadth-first spanning tree）是按BFS所得到的生成树。可以验证图 12-22b, c 和d 中的生成树都是图12-22a 的宽度优先生成树。（图12-22c 和d 分别是从顶点8和6开始搜索而得到的。）

当在一个无向连通图或网络中执行BFS时，到达新顶点的边正好有n-1条，这些边组成的子图也是一个生成树, 用这种方法所得到的生成树叫作深度优先生成树（depth-first spanning tree）。图12-23给出了图12-22a 的一些深度优先生成树。

练习

45. 根据图12-1a，完成以下练习：

1) 从顶点1开始产生一个宽度优先生成树。

2) 从顶点3开始产生一个宽度优先生成树。

3) 从顶点1开始产生一个深度优先生成树。

4) 从顶点3开始产生一个深度优先生成树。

46. 编写共享成员Undirected::BSpanningTree(i,BT)，该函数从一个连通无向图或网络中的顶点i 开始寻找一个宽度优先生成树。若因内存问题导致搜索失败，程序应能引发异常；若因没有生成树（图是非连通的）而导致失败，则返回 false；否则返回true。找到生成树时，将边返回到数组BT中。定义BT的数据类型。

47. 针对Undirected::DSpanningTree(i,BT)完成练习46，该函数从顶点i开始寻找一个深度优先生成树。

48. 编写共享成员Network::Cycle( )，用于确定网络中是否存在一个（有向）环路。可基于DFS或BFS来实现。

1) 证明代码的正确性。

2) 指出程序的时间和空间复杂性。

49. 设G是一个无向连通图或网络。编写函数Undirected::Bipartite(L)，如果G不是一个二分图（见例子12-3)，则函数返回false，否则返回true。当G是二分图时，函数还得在L中返回一个标号，如对于一个子集中的顶点，有 L[i]=1，而对于另一个子集中的顶点， L[i]=2。如果G有n个顶点且用矩阵描述，那么程序的复杂性应为 $\Theta(n^2)$。而如果用链表来描述 G，则复杂性应为Θ(n+e)。(提示：执行多次BFS，每次均从目前未到达的顶点开始，将这个顶点分配到集合1；与该顶点邻接的顶点分配至集合2；与集合2中顶点邻接的顶点再放入集合1, 如此进行下去。其间需检查分配冲突)。

50. G是一个无向图或网络，它的传递闭包（transitive closure）是一个0/1数组TC，当且仅当G中存在一条边数大于 1的从i 到j 的路径时， TC[i][j]=1。编写一个函数 Undirected::TransitiveClosure(TC)，计算G的传递闭包矩阵。函数的复杂性应为 $\Theta(n^2)$, 其中n 是G的顶点数目。（提示：采用构件标识策略)。

51. 若G是有向图，针对Network::TransitiveClosure(TC) 完成练习50。函数的复杂性是多少？