(　　)个结点。

A. $2^{h-1}-1$　　B. 2^{h-1}　　C. $2^{h-1}+1$　　D. 2^h-1

(6) 高度为 7 的 AVL 树最少有(①)个结点，最多有(②)结点。

① A. 12　　B. 21　　C. 33　　D. 54

② A. 63　　B. 64　　C. 65　　D. 127

(7) 在一棵高度为 h 的 AVL 树中，离根最远的叶结点在第(①)层，离根最近的叶结点在第(②)层。

① A. h−1　　B. h　　C. h+1　　D. 2^h-1

② A. $\lfloor \log_2 n \rfloor$　　B. $\lceil \log_2 (n+1) \rceil$　　C. $\lfloor h/2 \rfloor$　　D. $\lfloor h/2 \rfloor+1$

(8) 在如图 7-29 所示的 AVL 树中插入关键码 48，得到了一棵新的 AVL 树，在这棵新的 AVL 树中，关键码 37 所在结点的左、右子女结点中保存的关键码分别是(　　)。

图　7-29

A. 13，48　　B. 24，48

C. 24，53　　D. 24，90

(9) 最优二叉树(Huffman 树)和最优二叉搜索树均为平均搜索路径长度为 $\sum_{i=1}^{n} w_i h_i$ 最小的树，其中对最优二叉树，n 表示(①)，对最优二叉搜索树，n 表示(②)，构造这两种树均(③)。

①② A. 结点数　　B. 叶结点数

C. 非叶结点数　　D. 度为 2 的结点数

③ A. 需要一张 n 个关键码的表　　B. 需要对 n 个关键码进行动态插入

C. 需要 n 个关键码的搜索概率　　D. 不需要任何前提

7-34　判断题

(1) 用数组和单链表表示的有序表均可使用折半搜索方法来提高搜索速度。

(2) 有 n 个整数存放在一维数组 A[n]中，在进行顺序搜索时，无论这 n 个整数的排列是否有序，其平均搜索长度都相同。

(3) 在二叉搜索树中，任一结点所具有的关键码值都大于它的左子女(如果存在)的关键码值，同时小于其右子女(如果存在)的关键码值。

(4) 在一棵二叉树中，任一结点的关键码值都大于它的(如果存在)左子女结点的关键码值，且小于它的右子女结点(如果存在)的关键码值，则此二叉树一定是二叉搜索树。

(5) 对二叉搜索树的搜索都是从根结点开始的，如果搜索失败，则搜索指针一定落在叶结点上。

(6) 任一棵二叉搜索树的平均搜索时间都小于用顺序搜索法搜索同样结点的顺序表的平均搜索时间。

(7) 对于同一组待输入的关键码集合，虽然各关键码的输入次序不同，但得到的二叉搜索树都是相同的。

(8) 对于两棵具有相同关键码集合而形状不同的二叉搜索树，按中序遍历它们得到的序列的各元素的顺序是一样的。

(9) 在二叉搜索树上插入新的结点时，不必移动其他结点，仅需改动某个结点的指针，

使它由空变为非空即可。

(10) 在二叉搜索树上删除一个结点时，不必移动其他结点，只要将该结点的父结点的相应的指针域置为空即可。

(11) 最优二叉搜索树的任何子树都是最优二叉搜索树。

(12) 在所有结点的权值都相等的情况下，只有最下面两层结点的度数可以小于2，其他结点的度数必须等于2的二叉搜索树才是最优二叉搜索树。

(13) 在所有结点的权值都相等的情况下，具有平衡特性的二叉搜索树一定是最优二叉搜索树。

(14) 最优二叉搜索树一定是平衡的二叉搜索树。

(15) AVL树是一棵二叉搜索树，该树上任一结点的平衡因子的绝对值不大于1。

7-35 简答题

已知一个有序顺序表A[0..8N−1]的表长为8N，并且表中没有关键码值相同的数据元素。假设按如下所述的方法查找一个关键码值等于给定值X的数据元素：先在A[7]，A[15]，A[23]，…，A[8K−1]，…，A[8N−1]中进行顺序搜索，若搜索成功，则算法报告成功位置并返回；若不成功，即X>A[8K−1]的关键码，同时X<A[8(K+1)−1]的关键码，则可确定一个缩小的搜索范围A[8K]～A[8(K+1)−2]，然后可以在这个范围内执行折半搜索；特殊情况：若X>A[8N−1]的关键码，则搜索失败。

(1) 画出描述上述查找过程的判定树。

(2) 计算等搜索概率下搜索成功的平均搜索长度。

(3) 计算等搜索概率下搜索不成功的平均搜索长度。

7-36 简答题

(1) 在有N个结点的AVL树中，为结点增加一个存放结点高度的数据成员，那么每一个结点需要增加多少位(bit)？

(2) 若每一个结点中的高度计数器有8bit，那么这样的AVL树可以有多少层？最少有多少个关键码？

7-37 线性表中各结点的搜索概率不等，则可用如下策略提高顺序搜索的效率。若找到与给定值相匹配的元素，则将该元素与其直接前驱元素(若存在)交换，使得经常被搜索的元素尽量位于表的前端。试设计算法，在线性表的顺序存储表示和链接存储表示的基础上实现顺序搜索。

7-38 设在有序顺序表中搜索x的过程为：首先用x与表中的第4i(i=0，1，…)个元素做比较，如果相等，则搜索成功；否则确定下一步搜索的区间为4(i−1)+1到4i−1。然后在此区间内与第4i−2个元素作比较，若相等则搜索成功，否则继续与第4i−3或4i−1个元素进行比较，直到搜索成功。

(1) 给出实现算法。

(2) 试画出当表长n=16时的判定树，并推导此搜索方法的平均搜索长度(考虑搜索元素等概率和n%4=0的情况)。

7-39 给出利用分块搜索对搜索区间进行等分，而不建立索引表的顺序搜索算法。

7-40 已知一组递增有序的关键码k[n]：k[0]≤k[1]≤…≤k[n−1]，在相等搜索概率的情况下，若要生成一棵二叉搜索树。以哪个关键码值为根结点，按什么方式生成二叉搜

索树平衡性最好且方法又简单？阐明算法思路，写出相应的算法。如果 k[11]为：7，12，13，15，21，33，38，41，49，55，58。按上面算法画出这棵二叉搜索树。

7-41 设计一个算法；求指定结点在二叉搜索树中的层次。

7-42 编写一个算法，判定给定的关键码值序列（假定关键码值互不相同）是否是二叉搜索树的搜索序列。若是则函数返回 1，否则返回 0。

7-43 编写一个递归算法，在一棵有 n 个结点的随机建立起来的二叉搜索树上搜索第 $k(1\leqslant k\leqslant n)$ 小的元素，并返回指向该结点的指针。要求算法的平均时间复杂度为 $O(\log_2 n)$。二叉搜索树的每个结点中除 data、leftChild、rightChild 等数据成员外、增加一个 count 成员，保存以该结点为根的子树上的结点个数。

7-44 利用二叉树遍历的思想给出一个判断二叉树是否为平衡二叉树的算法。

7-45 在平衡二叉树的每个结点中增设一个数据成员 lsize，存储以该结点为根的左子树中的结点个数加一的值。编写一个算法，确定树中第 $k(k\geqslant 1)$ 个结点的位置。

7-46 编写一个算法，将二叉搜索树中所有 data 数据成员中值小于等于给定值 x 的结点全部删除掉。

7-47 编写一个递归算法，从大到小输出二叉搜索树中所有值不小于 x 的关键码。要求算法的时间复杂度为 $O(\log_2 n+m)$，n 为树中结点数，m 为输出的关键码个数。

7.5 补充练习参考答案

7-32 单项选择题

（1）B。顺序搜索是建立在线性表上的搜索方法，线性表的存储结构有顺序存储表示和链接存储表示两种。散列存储是直接存取组织，索引存储是分块存取组织，压缩存储是一种复杂结构的映像组织。

（2）C。在有 n 个元素的线性表中搜索到第 i 个元素的数据比较次数为 i，在相等搜索概率的情况下搜索成功的平均搜索长度为：

$$ASL_{unce}=\frac{1}{n}\sum_{i=1}^{n}i=\frac{1}{n}(1+2+\cdots+n)=\frac{1}{n}\frac{n(n+1)}{2}=\frac{n+1}{2}$$

（3）B。根据（2）小题的结果，顺序搜索的搜索成功的平均搜索长度为 $(30+1)/2=15.5$。

（4）A。在不相等搜索概率的情况下，搜索成功的平均搜索长度为：

$$ASL_{unce}=\sum_{i=1}^{3}p_i c_i=(1/2)\times 1+(1/3)\times 2+(1/6)\times 3=5/3$$

（5）C。折半搜索只能在有序顺序表上执行（跳表情况例外），应选 C。

（6）B。在有序顺序表中采用折半搜索，其平均搜索长度为：

$$ASL_{succ}=\frac{n+1}{n}\log_2 n-1\approx\log_2 n-1\quad(\text{当 } n>50)$$

（7）D。折半搜索的平均搜索长度可用二叉判定树分析。二叉判定树是一个理想平衡树，其高度与完全二叉树相同，根据完全二叉树的性质，有 $h=\lceil\log_2(n+1)\rceil$。

（8）A。从（6）和（7）小题可得答案。

(9) D。采用折半搜索的搜索序列为 50，24，13，18。注意，每次在一个搜索区间找中点时，如果区间中元素个数为偶数，例如 n=4，一定取中间偏前的元素，即⌊n/2⌋=2。

(10) D。采用斐波那契搜索的判定树如图 7-30 所示，每次从根开始搜索与给定值匹配的元素，设 n=F(7)−1=12，则根在 F(6)−1=7，其左子树的根在 F(5)−1=4，该子树的左子树的根在 F(4)−1=2，依此可推出 18 在 F(3)−1=1，搜索 18 需要比较 4 次。

(11) B。设区间的左端点为 low=0，右端点为 high=12，存储表元素的数组为 A[ ]，则插值求中点的公式为

$$\text{mid} = \text{low} + \frac{\text{x} - \text{A[low]}}{\text{A[high]} - \text{A[low]}}(\text{high} - \text{low})$$

如果用二叉判定树描述搜索过程，整个树的根在 0+(82−A[0])/(A[12]−A[0])×(12−0)=9，因为 x>A[9]，到右子区间搜索，该子区间边界是 low=10，high=12，中点是 10+(82−A[10])/(A[12]−A[10])×(12−10)=10，由于 x=A[10]，正好就是要找的元素，搜索成功，比较了 2 次，如图 7-31 所示。

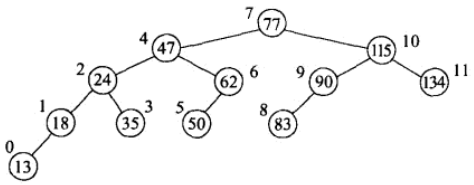


图 7-30

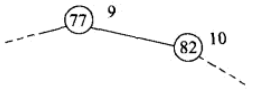


图 7-31

(12) B。分块搜索要建立索引表。因为线性表有 n=625 个元素，假设每个块可以存放 k 个元素，那么块数为 s=⌈n/k⌉，采用顺序搜索在索引表中确定与给定值匹配的元素到底在哪一个块(或子表)，平均搜索长度 $ASL = ASL_{\text{查索引}} + ASL_{\text{块搜索}} = (s+1)/2 + (k+1)/2$，对此式求关于 k 的偏导数，并令其为 0：

$$((s+1)/2 + (k+1)/2)_{k'} = ((\lceil n/k \rceil + 1)/2 + (k+1)/2)_{k'} = -n/(k^2 \times 2) + 1/2 = 0$$

得 $k = \sqrt{n} = \sqrt{625} = 25$。

7-33 单项选择题

(1) B。折半搜索的搜索性能分析可以用二叉判定树衡量，其平均搜索长度和最大搜索长度都是 $O(\log_2 n)$；二叉搜索树的搜索性能与数据的输入顺序有关，最好情况下的平均搜索长度与折半搜索相同，但最坏情况，即形成单支树的场合，其搜索长度可达 O(n)。

(2) B。理由同第(1)小题。随机输入数据构造二叉搜索树，有可能避开数据有序输入的情况，不至于形成单支树，搜索性能可接近折半搜索，时间复杂度的量级相同。

(3) B。根据输入序列构造的二叉搜索树如图 7-32 所示。由图 7-32 可知，从根到结点 20 的路径上有 5 个结点，找到元素 20 需要进行 5 次元素值的比较。

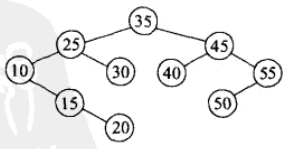


图 7-32

(4) 顺次选 C 和 A。搜索成功的平均搜索长度为 $ASL_{succ} = (1\times1 + 2\times2 + 3\times2 + 4\times1)/6 = 15/6$。为了计算搜索不成功的平均搜索长度，引进外结点，形成二叉判

定树，如图 7-28(b)所示。每次搜索不成功，一定是从根结点走到了一个外部结点，比较次数等于外部路径长度，搜索不成功的平均搜索长度等于到达各外结点的外部路径长度之和的平均值：$ASL_{unsucc}=(2\times2+3\times3+4\times2)/7=21/7$。

(5) D。如果一棵 AVL 树每个非叶结点的平衡因子都是 0，说明它每个非叶结点的左、右子树都一样高，是一棵满二叉树。高度为 h 的满二叉树的结点个数等于 2^h-1。

(6) ① C，② D。AVL 树最少有多少结点，与树的高度 h 有关。设 N_h 是高度为 h 的 AVL 树的最少结点数，则有 $N_0=0$，$N_1=1$，$N_h=N_{h-1}+N_{h-2}+1(h\geqslant2)$。如此可得 $N_2=N_1+N_0+1=2$，$N_3=N_2+N_1+1=4$，$N_4=N_3+N_2+1=7$，$N_5=N_4+N_3+1=12$，$N_6=20$，$N_7=33$。另一方面，高度为 h 的 AVL 树的最多结点数为 2^h-1，满二叉树情形。当 h=7 则有 $2^7-1=127$。

(7) ① B，② D。离根最远的叶结点当然在第 h 层了，树的高度就是从它开始计算的。而计算 AVL 树离根最近的叶结点所在层次的方法与推导高度为 h 的 AVL 树最少结点个数的过程类似。设高度为 h 的平衡二叉树的离根最近的叶结点所在层次为 L_h，则有

$$L_1=1,\quad L_2=2,\quad L_h=\min\{L_{h-1},L_{h-2}\}+1=L_{h-2}+1,\quad h>2$$

也可直接计算：$L_h=\lfloor h/2\rfloor+1$。

(8) C。48 应作为叶结点插在 24 的右子树上，53 的左子树上，37 的右子树上。插入后的 AVL 树如图 7-33 所示。它经历了一次先右后左的双旋转。关键码 37 处于根的位置，它所在结点的左、右子女结点中保存的关键码分别是 24 和 53。

(9) ① B，② D，③ C。在 Huffman 树中，n 是叶结点个数；在最优二叉搜索树中，n 是内结点个数，而内结点的度都为 2。构造这两种树都需要用到结点的权值，作为搜索树，这种权值就是关键码的搜索概率。

7-34 判断题

(1) 错。折半搜索只适合于可直接存取的有序表，即有序顺序表，链表不能使用。

(2) 错。在有序顺序表上和无序顺序表上执行顺序搜索，搜索成功的平均搜索长度相同，但搜索不成功的平均搜索长度不同。

(3) 对。二叉搜索树中任一结点的关键码值都大于它的左子女(如果存在)的关键码值，同时小于其右子女(如果存在)的关键码值。

(4) 错。一个反例如图 7-34 所示。它满足每个结点的值都大于左子女的值，小于右子女的值，但它不是二叉搜索树。题目的这种提法与二叉搜索树有出入。

图 7-33

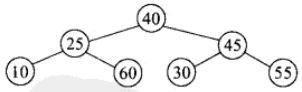


图 7-34

(5) 错。从根开始搜索不错，但搜索失败，搜索指针一定走到失败结点，它可能是从某个结点(不一定是叶结点)的空子女指针走出去的。

(6) 错。如果二叉搜索树是单支树，则其平均搜索时间不比顺序搜索快。

(7) 错。不同的输入序列，得到的二叉搜索树不同。

(8) 对。对于同一关键码值集合，虽然构造的二叉搜索树不同，但中序遍历结果相同。

都是按照关键码的值进行升序排列。

(9) 对。新结点插入时,它作为叶结点,链接到某个结点下面即可,只需改一个指针。

(10) 错。从二叉搜索树中删除某个叶结点,可以不移动其他结点,仅改变其父结点的一个指针即可,但对具有两个子女的非叶结点,删除它需要移动其他结点。

(11) 对。如果最优二叉搜索树的子树不是二叉搜索树,将会导致整个二叉搜索树的平均搜索长度不能达到最小,这与最优二叉搜索树的定义矛盾。

(12) 对。在所有结点的权值都相等的情况下,所有结点到根的路径长度满足 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3,3, 3, 3, 4, …的前 n 项,使得总的路径长度 $\sum_{i=1}^{n} w_i = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$ 达到最小,才能满足最优二叉搜索树的要求。完全二叉树或理想平衡树能达到要求。

(13) 错。具有平衡特性的二叉搜索树不一定是最优二叉搜索树,如 AVL 树就不是最优二叉搜索树。

(14) 错。最优二叉搜索树是平均搜索长度最小的二叉搜索树,是根据各个内结点和外结点的搜索概率来构造的。

(15) 对。AVL 是一棵高度平衡的二叉搜索树,该树上任一结点的平衡因子的绝对值不大于 1。

7-35 简答题

(1) 相应的判定树如图 7-35 所示。其中,每一个关键码值下的数字为其搜索成功时的关键码比较次数。

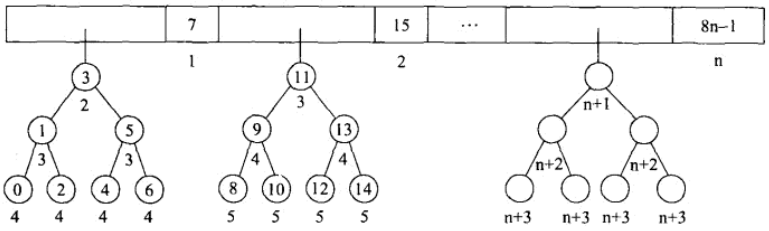


图 7-35

(2) 搜索成功的平均搜索长度为所有 8n 个内结点的比较次数的平均值:

$$ASL_{succ} = \frac{1}{8n}\sum_{i=0}^{8n-1} C_i = \frac{1}{8n}\left(\sum_{i=1}^{n} i + \sum_{i=2}^{n+1} i + 2\sum_{i=3}^{n+2} i + 4\sum_{i=4}^{n+3} i\right)$$

$$= \frac{1}{8n}\left(\sum_{i=1}^{n} (i + (i+1) + 2(i+2) + 4(i+3))\right)$$

$$= \frac{1}{8n}\sum_{i=1}^{n} (8i+17) = \frac{1}{n}\sum_{i=1}^{n} i + \frac{17}{8} = \frac{n+1}{2} + \frac{17}{8}$$

(3) 搜索不成功的平均搜索长度为所有 8n+1 个外结点的比较次数的平均值:

$$ASL_{unsucc} = \frac{1}{8n+1}\left(\left(8\sum_{i=4}^{n+3} i\right) + n\right)$$

$$= \frac{1}{8n+1}\left(8 \times \frac{(n+7) \times n}{2} + n\right) = \frac{4n^2 + 29n}{8n+1}$$

7-36 简答题

(1) 有 N 个结点的 AVL 树的高度不超过：

$$k \leqslant 1.44 \times \log_2(N+1)$$

假设为表示 k 需要 m 位，最大取值可达 2^m-1，则有：

$$2^m-1 \leqslant k \rightarrow m \leqslant \log_2(k+1)$$

假设 $N=1023$，$k=1.44\times\log_2(1023+1)=1.44\times10=14.4$。$m\leqslant\log_2(14.4+1)<4$。因此，为了存储结点的高度，可增加 4bit。

(2) 若每一个结点中的高度计数器有 $m=8$，有 $8\leqslant\log_2(k+1)$，可推出 $k\geqslant2^8-1$，AVL 树可以有 2^8-1 层。又由于 $k\leqslant1.44\times\log_2(N+1)$，则：

$$N\geqslant2^{(k/1.44)}-1>2^{177}-1$$

最少可存储 $2^{177}-1$ 个关键码。

7-37 算法设计题

(1) 采用顺序表来存储线性表，实现顺序搜索算法。如果搜索成功，先交换再返回交换后的搜索到元素所在结点的地址。算法的描述如下：

```
#include "SeqList.h"
template <class T>
int SeqSearch (SeqList<T>& L, T x) {
//在顺序表 L 中从表的前端开始搜索与给定值 x 匹配的元素，找到后与它前一个元素
//(若有)交换位置，再返回新的位置。假定本函数是 SeqList 类的友元函数
    int i=1, n=L.last+1;  T temp;
    while (i <=n && L.data[i] !=x) i++;
    if (i>n) return 0;                              //搜索不成功
    else {                                          //搜索成功
        if (i>1) {temp=L.data[i];  L.data[i]=L.data[i-1];  L.data[i-1]=temp;}
        return i-1;
    }
};
```

(2) 采用带附加头结点的线性链表存储线性表，实现顺序搜索算法。为了实现交换，需要为检测指针 p 设置一个直接前驱指针 pre。算法描述如下：

```
#include "LinkedList.h"
template <class T>
LinkNode<T> * LinkSearch (List<T>& L, T x) {
    LinkNode<T> * p=L.First()->link, * pre=L.First();  T temp;
    while (p !=NULL && p->data !=x) {pre=p;  p=p->link;}
    if (p !=NULL && pre !=L.First()) {              //搜索成功，交换结点数据
        temp=p->data;  p->data=pre->data;  pre->data=temp;
        p=pre;                                      //搜索到的记录已往前移动一个位置
    }
    return p;
};
```

7-38　算法设计题

(1) 实现算法

本题实际上就是 7-35 题的简化版。首先按一个等于 4 的间隔跨步，确定一个小的只有 3 个元素的范围再进行搜索。实现算法的描述如下，假定它是顺序表类的友元函数。

```
#include "SeqList.h"
template <class T>
int Gap_Search (SeqList<T>& L, T x) {
//数组下标从 0 开始，返回位置从 1 开始，故返回地址要在数组下标基础上加一
    int i, k=L.last/4;
    for (i=0; i <=k; i++) {                          //跨区间搜索
        if (L.data[4 * i]==x) return 4 * i+1;        //搜索成功，返回位置
        if (L.data[4 * i]>x) {                       //确定搜索区间 4i-3~4i-1
            if (!i) return 0;                        //x 值比 0 号元素还小，失败
            if (L.data[4 * i-2]==x) return 4 * i-1;  //位置等于数组下标加一
            if (L.data[4 * i-2]>x)                   //到左边搜索
                if (L.data[4 * i-3]==x) return 4 * i-2;
                else return 0;
            else if (L.data[4 * i-1]==x) return 4 * i;  //到右边搜索
                else return 0;
        }
    }
    int low=4 * k+1, high=L.last;                    //处理最后剩余部分
    i=(low+high)/2;
    if (L.data[i]==x) return i+1;                    //搜索成功
    if (i>low && L.data[i-1]==x) return i;           //处理左边
    if (i<high && L.data[i+1]==x) return i+2;        //处理右边
    return 0;                                        //搜索失败
};
```

(2) 表长 n=16 时的判定树(设表元素为 L[i])如图 7-36 所示。

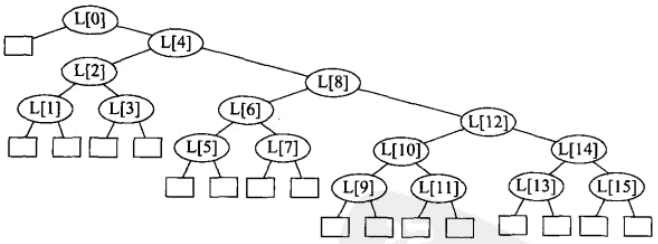


图　7-36

在此判定树中，比较 1 次能找到的有 L[0]，比较 2 次能找到的有 L[4]，比较 3 次能找到的有 L[2]、L[8]，比较 4 次能找到的有 L[1]、L[3]、L[6]、L[12]，比较 5 次能找到的有 L[5]、L[7]、L[10]、L[14]，比较 6 次能找到的有 L[9]、L[11]、L[13]、L[15]。

等概率下搜索成功的平均搜索长度为 $ASL_{succ}=(1+2+3\times2+4\times4+5\times4+6\times4)/16=$

69/16。搜索不成功的平均搜索长度为 $ASL_{unsucc}=(1+4\times4+5\times4+6\times8)/17=85/17=5$。

7-39 算法设计题

分块搜索首先将搜索区间划分为若干长度为 s 的子区间(块),最后一个子区间(块)的长度可以小于 s。搜索过程分两步:

第一步,确定搜索哪个子区间。为此,顺序搜索各个子区间,用给定值 x 与区间的右端点比较,若 x 大于右端点的关键码值,则搜索下一个子区间;否则进入第二步。

第二步,在子区间内顺序搜索。

具体算法描述如下:

```
#include "SeqList.h"
template <class T>
int BlockSearch (SeqList<T>& L, int s, T x) {
//在顺序表 L 内做分块搜索,s 是块大小,x 是给定值。假设本函数是 SeqList 类的
//友元函数。若搜索成功,函数返回与给定值匹配元素的位置(从 1 开始)
    int low=0, high=L.last, temp;
    if (s>1 && s<L.last) {                              //划分区间,跨步
        temp=((L.last+1)/s) * s-1;                      //最后完整块右端点
        high=s-1;
        while (high <=temp && x>L.data[high])           //寻找待搜索子区间
            high=high+s;
        if (high>temp) {low=temp+1;  high=L.last;}      //在最后一块
        else low=high-s+1;                              //不在最后一块
    }
    while (low <=high && L.data[low] !=x) low++;        //在块内顺序搜索
    if (low <=high) return low+1;                       //搜索成功
    else return 0;                                      //搜索不成功
};
```

7-40 算法设计题

以中间的关键码值为根结点,按折半搜索的二叉判定树的生成方法构造二叉搜索树,既有平衡性又简单。算法的描述如下:

```
#include "BST.h"
tamplate<class T>
void Create_BestBST (BSTNode<T> * & t, int low, int high, T k[]) {
//low 和 high 分别为关键码序列的下界和上界,初始值分别为 0 和 n-1,t 是创建
//子树的根。由于新根要回传给实参,所以它是引用型指针参数
    if (low>high) t=NULL;
    else {
        int m=(low+high)/2;
        t=new BSTNode<T>;  t->data=k[m];
        Create_BestBST (t->leftChild, low, m-1, k);
        Create_BestBST (t->rightChiid, m+1, high, k);
    }
```

这是一个递归算法，通过引用型参数 t 把新创建起来的子树根结点自动链接到父结点的某个子女指针。主程序的主调用语句为 Create_BestBST (root, 0, n－1, k)。

对所给关键码序列，根据上述算法可得二叉搜索树如图 7-37 所示。

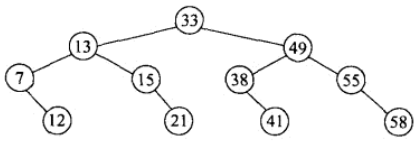


图 7-37

7-41 算法设计题

求指定结点在二叉搜索树中的层次，可以按第 5 章所介绍用层次遍历方法解决，也可以利用二叉搜索树的特性采用非递归算法来实现。算法描述如下：

```
#include "BST.h"
template <class T>
int level_Count (BST<T>& BT, BSTNode<T> * p) {
//在二叉搜索树 BT 中求指定结点 p 所在层次，如果找到结点，函数返回层次，否则
//返回 0。约定树的层次从 1 开始
    int k=0;  BSTNode<T> * t=BT.getRoot();
    if (t !=NULL) {
        k++;
        while (t !=NULL && t->data !=p->data) {
            if (t->data<p->data) t=t->rightChiid;
            else t=t->leftChiid;
            k++;
        }
        if (t==NULL) k=0;                                    //结点 p 不在二叉树 BT 中
    }
    return k;
};
```

7-42 算法设计题

根据二叉搜索树的特点，搜索路径只可能沿某一结点的左或右分支逐层向下搜索，而不可能在两个分支之间横向跳跃或往上回溯。搜索范围应在给定关键码值的上下波动，不断接近给定关键码值，且在结点左子树上的关键码值不会大于结点的关键码值，右子树上的关键码值不会小于结点的关键码值。

例如，给定一个给定值 363，在二叉搜索树上寻找关键码值为 363 的结点时，访问的关键码值序列 L＝{2, 252, 401, 398, 330, 344, 397, 363}。若将 L 分为两个子序列，L1 所包含的都是小于或等于给定值 363 的数据，即 L1＝{2, 252, 330, 344, 363}，L2 所包含的都是大于给定值 363 的数据，即 L2＝{401, 398, 397}。如果从 L 所生成的 L1 单调递增，L2 单调递减，且除待查元素外，L1 中每个数据都小于给定值，L2 中每个数据大于给定值，则 L 是一个搜索序列，否则不是搜索序列。算法描述如下：

```
#define maxSize 20
typedef struct {                                          //待认定的搜索序列结构定义
    int elem[maxsize];                                    //搜索序列数据存储数组
    int len;                                              //序列实际数据个数
} Sequence;
int issearch (Sequence& L, int x) {
//若 L 是搜索序列则函数返回 1,否则返回 0。x 是待搜索关键码值
    reduce (L, L1, L2);
    return judge (L1, L2, x);
};
void reduce (Sequence& L, Sequence& L1, Sequence& L2) {
//将序列 L 压缩并分解为 L1,L2
    int i=0, j=0, k=0;
    if (L.elem[i]<L.elem[i+1]) {
        L1.elem[j++]=L.elem[i];
        while (i+1<L.len && L.elem[i]<L.elem[i+1]) i++;      //单调上升压缩
    }
    L2.elem[k++]=L.elern[i];
    while (i<L.len) {
        while (i+1<L.len && L.elem[i]>L.elem[i+1]) i++;      //单调下降压缩
        L1.elem[j++]=L.elem[i];
        if (i<L.len) {
            while (i+1<L.len && L.elem[i]<L.elem[i+1]) i++; //单调上升压缩
            L2.elem[k++]=L.elem[i];
        }
    }
    L1.len=j;  L2.len=k;
};
int judge (Sequence& L1, Sequence& L2, int x) {
//判断 L1 是否单调递减,L2 是否单调递增,且 L1 的元素值不比 x 大,L2 的元素值
//不比 x 小
    int i=0, minlen, flag=1;
    minlen= (L1.len<L2.len)? L1.len:L2.len;
    while (flag && i+1<minlen) {
        if (L1.elem[i]>L1.elem[i+1]||L1.elem[i]>x) flag=0;
        if (L2.elem[i]<L2.elem[i+1]||L2.elem[i]<x) flag=0;
        i++;
    }
    if (L1.elem[i]>x||L2.elem[i]<x) flag=0;                  //处理 minlen 位置
    if (L1.len>L2.len) {                                     //L1 尚有一个元素未处理
        if (L1.elem[i]>L1.elem[i+1]||L1.elem[i+1]>x) flag=0;
    }
    else if (L1.len<L2.len) {                                //L2 尚有一个元素未扫描
        if (L2.elem[i]<L2.elem[i+1]||L2.elem[i+1]<x) flag=0;
    }
    return flag;
}
```

7-43 算法设计题

设二叉搜索树的根结点为 t，根据结点存储的信息，有 4 种不同搜索情况：

- ✧ 若 t->leftChild 非空且 t->leftChild->count=k-1，则结点 t 即为第 k 小的元素，搜索成功。
- ✧ 若 t->leftChild 非空且 t->leftChild->count>k-1，则第 k 小的元素必在结点 t 的左子树，继续到结点 t 的左子树中搜索。
- ✧ 若 t->leftChild 为空，则第 k 小的元素必在结点 t 的右子树，若 t->rightChild 非空且 t->rightChild->count=k-1，则结点 t 即为第 k 小的元素，搜索成功。
- ✧ 若 t->leftChild 非空，且 t->leftChild->count<k-1，则第 k 小元素必在右子树，继续搜索右子树，寻找第 t->count-(t->leftChild->count+1)小的元素。

对左、右子树的搜索采用同样的规则，递归实现的算法描述如下：

```
#include "BST.h"
template <class T>
BSTNode<T> * Search_Small (BSTNode<T> * t, int k) {
//在以 t 为根的子树上寻找第 k 小的元素，返回其所在结点地址。k 从 1 开始计算
//在树结点中增加一个 count 数据成员，存储以该结点为根子树的结点个数
    if (k<1||k>t->count) return NULL;       //k 的范围无效，返回空指针
    if (t->leftChild !=NULL && t->leftChild->count==k-1)
        return t;
    if (t->leftChild !=NULL && t->leftChild->count>k-1)
        return Search_Small (t->leftChild, k);
    if (t->leftChild==NULL)
        return Search_Small (t->rightChild, k-1);
    if (t->leftChild !=NULL && t->leftChild->count<k-1)
        return Search_Small (t->rightChild, t->count-t->leftChild->count-1);
};
```

最大搜索长度取决于树的高度。由于二叉搜索树是随机生成的，其高度应是$O(\log_2 n)$，算法时间复杂度为 $O(\log_2 n)$。

7-44 算法设计题

一棵平衡二叉树是高度平衡的二叉搜索树，又称 AVL 树。它要求根结点的平衡因子 bf 的绝对值不能超过 1，且根的左、右子树都是平衡二叉树。可以采用后序遍历二叉树的方式进行判断。首先对根结点的平衡因子进行判断，再递归地对其左、右子树进行判断。

算法的描述如下：

```
#include <stdlib.h>
#include "AVLTree.h"
template <class E, class K>
bool isBalanceTree (AVLNode<E, K> * t, int& h) {
//算法递归地判断以 t 为根的子树是否 AVL 树，是则返回 true，不是则返回 false
//引用型参数 h 返回子树的高度
    int hl, hr;
    if (t==NULL)                                          //空树的高度为 0
```

```
        {h=0;  return true;}                                    //空树的高度平衡
    if (t->leftChild==NULL && t->rightChild==NULL)
        {h=1;  return true;}                                    //叶结点的高度为 1
    if (!isBalanceTree (t->leftChild, hl)) return false;        //判断左子树是否平衡
    if (!isBalanceTree (t->rightChild, hr) return false;        //判断右子树是否平衡
    h=(hl>hr)? hl+1:hr+1;
    return abs(hl-hr)<2;
};
```

7-45 算法设计题

此算法与 7-43 题有点类似，但还是不同。实际上，增设的这个 lsize 数据成员里面记入的就是以该结点为根的子树中该结点的次序。算法的描述如下：

```
#include "AVLTree.h"
template <class E, class K>
AVLNode<E, K> * Search_Small (AVLNode<E, K> * t, int k) {
//在 AVL 树的结点中增设一个数据成员 lsize,给出以该结点为根的左子树中的结点
//个数加一。k 从 1 开始计算
    if (t==NULL||k<1) return NULL;                              //空树或 k 小于 1
    if (t->lsize==k) return t;                                  //找到第 k 小的元素结点
    if (t->lsize>k) return Search_Small (t->leftChild, k);
    else return Search_Small (t->rightChild, k-t->lsize);
};
```

在右子树中搜索第 k 小的元素结点时，要注意减去左子树及根的结点个数。

7-46 算法设计题

根据二叉搜索树的特点可知，若根结点 bt 的 data 数据成员的值小于等于 x，则其左子树的所有结点的 data 数据成员的值均小于等于 x；若根结点 bt 的 data 数据成员的值大于 x，则其右子树的所有结点的 data 数据成员的值均大于 x。算法基本思想如下：

① 若 bt－>data<＝x，则将 bt 指向 bt 的右子女，并删除根结点 bt 及 bt 的左子树的全部结点，重复这一步，直到 bt－>data>x 为止。

② 沿 bt 的左分支搜索，直到左分支结点的 data 数据成员的值小于等于 x，回到①继续执行删除。

重复执行①②，直到左子树空为止。

实现算法描述如下：

```
#include "BST.h"
template <class T>
void del_eqorlsix (BSTNode<T> * bt, T x) {
    BSTNode<T> * p, * pr=NULL;
    while (bt !=NULL) {
        while (bt !=NULL && bt->data <=x) {
            p=bt;  bt=bt->rightChild;
            delSubTree (p->leftChild);                          //删除 p 的左子树
            delete p;                                           //删除结点 p
```

```
            if (pr!=NULL) pr->leftChild=bt;
        }
        while (bt!=NULL && bt->data>x)
            {pr=bt;  bt=bt->leftChild;}
    }
};
void delSubTree (BSTNode<T> * p) {
//删除以 p 为根子树的全部结点
    if (p!=NULL) {
        delSubTree (p->leftChild);
        delSubTree (p->rightChild);
        delete p;
    }
};
```

7-47　算法设计题

中序镜像遍历二叉搜索树，可按从大到小的顺序输出各结点的关键码值，直到某结点的关键码值刚小于 x 为止。算法的描述如下：

```
#include <iostream.h>
#include "BST.h"
template <class T>
void Output (BSTNode<T> * bt, T x) {
//按从大到小的顺序输出二叉搜索树各结点的值，直到结点的值小于 x 为止
//假设当前进入函数时根结点 bt 的 data 值大于或等于 x
    if (bt!=NULL) {
        if (bt->rightChild !=NULL) Output (bt->rightChild, x);
        cout<<bt->data<<endl;
        if (bt->leftChild !=NULL && bt->leftChild->data >=x)
            Output (bt->leftChild, x);
```

第8章　图

图是一种重要的非线性结构。它的特点是每一个顶点都可以与其他顶点相关联，与树不同，图中各个顶点的地位都是平等的，对顶点的编号都是人为的。通常，定义图由两个集合构成：一个是顶点的非空有穷集合，一个是顶点与顶点之间关系（边）的有穷集合。对图的处理要区分有向图与无向图。它的存储表示可以使用邻接矩阵，可以使用邻接表，前者属顺序表示，后者属链接表示。在本章着重讨论了图的深度优先搜索和广度优先搜索算法，附带引入了生成树与生成森林的概念。对于带权图，给出了最小生成树的两种方法：Prim 算法和 Kruskal 算法，后者使用了最小堆和并查集作为它的辅助求解手段。在解决最短路径问题时，采用了逐步求解的策略。最后讨论了工程计划常用的活动网络。涉及的主要概念是拓扑排序和关键路径，在解决应用问题时它们十分有用。

8.1　复习要点

本章复习的要点是：

（1）图的基本概念。

① 图的定义、对顶点集合与边集合的要求。

② 图中各顶点的度及度的度量。

③ 无向图的连通性、连通分量、最小生成树的概念。

④ 有向图的强连通性、强连通分量。

⑤ 图的路径和路径长度、回路。

⑥ 无向连通图的最大边数和最小边数。

⑦ 有向强连通图的最大边数与最小边数。

（2）图的存储表示。

① 邻接矩阵、邻接表和邻接多重表的结构定义。

② 在这些存储结构中顶点、边的表示及其个数的计算。

③ 在这些存储表示上的典型操作：

✧ 找第一个邻接顶点；

✧ 找下一个邻接顶点；

✧ 求顶点的度；

✧ 特别地，求有向图顶点的出度和入度。

④ 建立无向带权图的邻接表的算法，要求输入边的数目随机而定。

（3）图的遍历。

① 图的深度优先搜索的递归算法（回溯法）。

② 使用队列的图的广度优先搜索算法。

③ 在图的遍历算法中辅助数组 visited 的作用。

④ 用图的深度优先搜索和广度优先搜索算法建立图的生成树或生成森林的方法。

⑤ 利用图的遍历算法求解连通性问题的方法。

⑥ 重连通图的概念和关节点的判定。

(4) 最小生成树。

① 最小生成树的概念。

② 构造最小生成树的 Prim 算法和 Kruskal 方法(要求构造步骤,不要求算法)。

③ 求解最小生成树的 Prim 算法以及算法的复杂性分析。

④ 求解最小生成树的 Kruskal 算法以及算法的复杂性分析。

⑤ 在求解最小生成树算法中最小堆和并查集的使用。

(5) 图的最短路径。

① 求解最短路径的 Dijkstra 算法的设计思想和对边上权值的限制。

② 求解最短路径的 Dijkstra 算法以及算法复杂性分析,注意 dist 和 path 数组的变化。

③ 求解最短路径的 Floyd 算法的设计思想和复杂性估计。

(6) 图的活动网络。

① 活动网络的拓扑排序概念。

② 有向图中求解拓扑排序的算法以及算法的复杂性分析。

③ 在拓扑排序算法中入度为 0 的顶点栈的作用。

④ 用邻接表作为图的存储表示,注意拓扑排序执行过程中入度为 0 的顶点栈的变化。

⑤ 利用图的深度优先搜索进行拓扑排序的递归算法。

⑥ 关键路径的概念及其工程背景。

⑦ 求解关键路径的方法。

⑧ 明确某关键活动加速不一定能使整个工程进度提前,但某关键活动延误一定导致整个工程延期。

8.2 难点和重点

(1) 图的定义。

① 是否有“空图”的概念?

② 有 n 个顶点的无向图最多有多少条边,最少有多少条边?无向连通图的情形呢?

③ 有 n 个顶点的有向图最多有多少条边,最少有多少条边?强连通图的情形呢?

④ 在无向图中顶点的度与边有何关系?在有向图中顶点的出度、入度与边有何关系?

⑤ 图中任意两顶点间的路径是用顶点序列标识的,还是用边序列标识的?

⑥ 图中各个顶点的序号是规定的,还是人为可改变的?

(2) 图的存储表示。

① 有 n 个顶点、e 条边的无向图的邻接矩阵有多少矩阵元素?其中有多少零元素?有向图的情形呢?

② 为什么在有向图的邻接矩阵中,统计某行 1 的个数,得到顶点的出度;统计某列 1 的个数,得到某顶点的入度?

③ 有一个存储 n 个顶点 e 条边的邻接表,某个算法要求检查每个顶点,并扫描每个顶

点的边链表，那么这样的算法的时间复杂度是 O(n×e)，还是 O(n+e)？

④ 设每个顶点数据占 4 个字节，顶点号码占 2 字节，每条边的权值占 4 个字节，每个指针占 2 个字节。若一个无向图有 n 个顶点 e 条边，使用邻接矩阵经济还是使用邻接表经济？

⑤ 有向图的邻接表与逆邻接表组合起来形成十字链表，它适合于什么场合？

⑥ 如何在无向图的邻接多重表中寻找与给定顶点相关联的边？

(3) 图的遍历。

① 图的遍历针对顶点，要求按一定顺序访问图中所有顶点，且每个顶点仅访问一次。可否针对边也使用遍历算法？

② 图的深度优先搜索类似于树的先根次序遍历，可归属于哪一类算法？

③ 图的遍历对无向图和有向图都适用吗？

④ 图的深度优先遍历如何体现“回溯”？

⑤ 图的广度优先遍历类似于树的层次序遍历，需要使用何种辅助结构？

⑥ 图的广度优先生成树是否比深度优先生成树的深度低？

⑦ 图的深度优先搜索是个递归的过程，而广度优先搜索为何不是非递归的过程？

⑧ 图的深度优先搜索遍历一个连通分量上的所有顶点如何得到生成树？

⑨ 对无向图进行遍历，在什么条件下可以建立一棵生成树？在什么条件下得到一个生成森林，其中每个生成树对应图的什么部分？

⑩ 对有向图进行遍历，在什么条件下可以建立一棵生成树？在什么条件下得到一个生成森林？

⑪ 如何判断一个无向图中的关节点？如何以最少的边构成重连通图？

(4) 最小生成树。

① 图的最小生成树必须满足什么要求？

② 构造图的最小生成树有多种算法，遇到在一组边集合中选出权值最小的边的问题时，为什么使用堆结构辅助最好？

③ 把所有的边按照其权值加入到最小堆中，相应算法的时间复杂度是多少？

④ Kruskal 算法中为了判断一条边的两个端点是否在同一连通分量上，为何采用并查集作为辅助结构？

⑤ Prim 算法每次选择一个端点在生成树顶点集合 U，另一个端点不在生成树顶点集合 U 的边中权值最小的边，可否采用最小堆作为辅助结构？

(5) 最短路径。

① 用 Dijkstra 算法求最短路径，为何要求所有边上的权值必须大于 0？

② Dijkstra 算法是求解单源最短路径问题的算法，可否用它解决单目标最短路径问题？

③ 用 Floyd 算法求最短路径，允许图中有带负权值的边，但为何不许有包含带负权值的边组成的回路？

(6) 活动网络。

① 什么是拓扑排序？它是针对何种结构的？

② 可以对一个有向图的所有顶点重新编号，把所有表示边的非零元素集中到邻接矩阵的上三角部分。根据什么顺序进行顶点的编号？

③ 拓扑排序的一个重要应用是判断有向图中是否有环。如何判断？

④ 如果调用深度优先搜索算法，在每次递归结束并退出时输出顶点，就可得到一个逆拓扑有序的序列。此方法有效性的前提是什么？

⑤ 为什么拓扑排序的结果不唯一？

⑥ 关键路径法的应用背景是什么？

⑦ 为有效地进行关键路径计算，应采用何种结构来存储 AOE 网络？

⑧ 为什么说加速某一关键活动不一定能缩短整个工程的工期？

⑨ 为什么说某一关键活动不能按期完成就会导致整个工程的工期延误？

⑩ 在某些 AOE 网络中各事件的最早开始时间和最迟允许开始时间都相等，是否所有活动都是关键活动？

⑪ 若一个无向图有 n 个顶点，每个顶点的度都大于或等于 2，该图是否存在环？为什么？

8.3 教材习题解析

一、思考题

8-1 在如图 8-1 所示的有向图中：

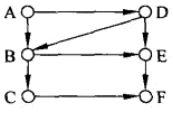


图 8-1

(1) 该图是强连通的吗？若不是，给出其强连通分量。

(2) 请给出该图的所有简单路径及有向环。

(3) 请给出每个顶点的入度和出度。

(4) 请给出该图的邻接矩阵、邻接表、逆邻接表和十字链表。

【解答】

(1) 判断一个有向图是否强连通，要看从任一顶点出发是否能够回到该顶点。图 8-1 所示的有向图做不到这一点，它不是强连通。各个顶点自成强连通分量。

(2) 所谓简单路径是指该路径上没有重复的顶点。

从顶点 A 出发，到其他各个顶点的简单路径有 A→B，A→D，A→B→C，A→B→E，A→D→E，A→D→B，A→B→C→F，A→B→E→F，A→D→B→C，A→D→B→E，A→D→E→F，A→D→B→E→F，A→D→B→C→F。

从顶点 B 出发，到其他各个顶点的简单路径有 B→C，B→E，B→C→F，B→E→F。

从顶点 C 出发，到其他各个顶点的简单路径有 C→F。

从顶点 D 出发，到其他各个顶点的简单路径有 D→B，D→E，D→B→C，D→B→E，D→E→F，D→B→C→F，D→B→E→F。

从顶点 E 出发，到其他各个顶点的简单路径有 E→F。

从顶点 F 出发，没有到其他各个顶点的简单路径。

(3) 顶点 A 的入度等于 0，出度等于 2。顶点 B 的入度等于 2，出度等于 2。顶点 C 的入度等于 1，出度等于 1。顶点 D 的入度等于 1，出度等于 2。顶点 E 的入度等于 2，出度等于 1。顶点 F 的入度等于 2，出度等于 0。

(4) 图的存储结构如图 8-2 所示。

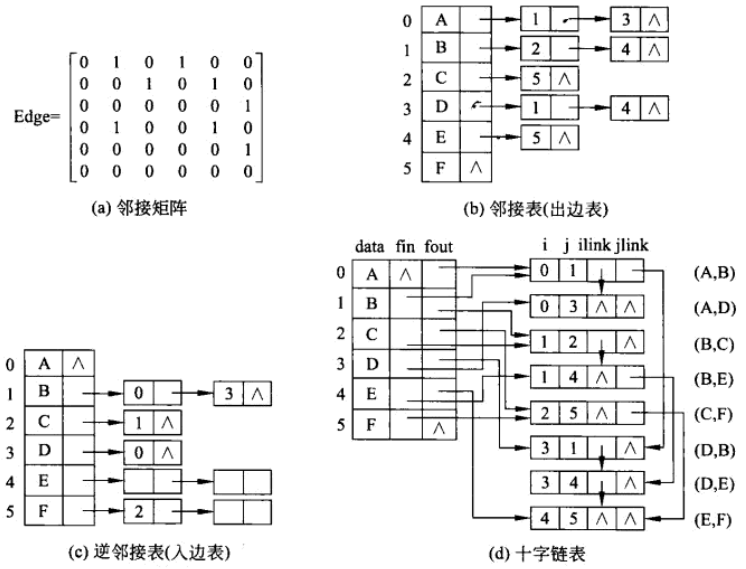


图 8-2

8-2 对 n 个顶点的无向图和有向图，采用邻接矩阵和邻接表表示时，如何判别下列有关问题：

(1) 图中有多少条边？

(2) 任意两个顶点 i 和 j 是否有边相连？

(3) 任意一个顶点的度是多少？

【解答】

(1) 对于有 n 个顶点的无向图，采用邻接矩阵存储，基于其对称性，只要统计矩阵对角线以上部分或对角线以下部分 1 的个数，就可知道图中有多少条边；若采用邻接表存储，只要统计邻接表各顶点边链表中(边)结点个数，再除以 2，就可知道图中有多少条边。对于有 n 个顶点的有向图，若采用邻接矩阵存储，需统计矩阵中所有 1 的总数，就可知道图中有多少条边；若采用邻接表存储，只要统计邻接表各顶点边链表中结点个数，就可知道图中有多少条边。

(2) 对于用邻接矩阵 A[][]存储的图，如果 A[i][j]=1，则可断定存在从顶点 i 到顶点 j 的边；对于无向图，由于对称性，也可以查看 A[j][i]是否等于 1，是则可断定存在从顶点 i 到顶点 j 的边。对于用邻接表存储的图，应在顶点 i 的边链表中查找各边结点，如果存在一个边结点，它的目标顶点的顶点号是 j，则可断定存在从顶点 i 到顶点 j 的边；对于无向图，还可以到第 j 个顶点的边链表中去查找。

(3) 对于用邻接矩阵存储的无向图，统计第 i 行或第 i 列中 1 的个数，可得顶点 i 的度；对于用邻接表存储的无向图，统计顶点 i 的边链表中结点的个数，也可得顶点 i 的度。对于

用邻接矩阵存储的有向图，统计第 i 行 1 的个数，可得顶点 i 的出度，统计第 i 列 1 的个数，可得顶点 i 的入度。对于用邻接表存储的有向图，统计邻接表顶点 i 的边链表中结点的个数，可得该顶点的出度，统计对应逆邻接表顶点 i 的边链表中结点的个数，可得该结点的入度。顶点 i 的度等于其出度加上入度。

8-3 n 个顶点的连通图至少有多少条边？强连通图呢？

【解答】

n 个顶点的连通图至少有 n－1 条边。n(n＞1)个顶点的强连通图至少有 n 条边，只有一个顶点的强连通图至少有 0 条边，如图 8-3 所示。

图 8-3

8-4 用邻接矩阵表示图时，矩阵元素的个数与顶点个数是否相关？与边的条数是否相关？

【解答】

用邻接矩阵表示图，矩阵元素的个数是顶点个数的平方，与边的条数无关。如果是非带权图，矩阵中非零元素的个数与边的条数有关；如果是带权图，矩阵中所有小于∞且大于 0 的元素个数与边的条数有关。

8-5 用邻接矩阵表示图时，若图中有 1000 个顶点，1000 条边，则形成的邻接矩阵有多少矩阵元素？有多少非零元素？是否稀疏矩阵？

【解答】

一个图中有 1000 个顶点，其邻接矩阵中的矩阵元素有 $1000^2=1000000$ 个。在有向图的情形，其邻接矩阵中有 1000 个非零元素，999000 个零元素，且这些非零元素的分布没有规律，因此是稀疏矩阵。在无向图的情形，其邻接矩阵中有 2000 个非零元素，998000 个零元素，因为矩阵是对称的，因此有人把它归于对称矩阵。其实稀疏矩阵与对称矩阵不是各自孤立的，邻接矩阵既可以是对称矩阵也可以是稀疏矩阵。故应是稀疏矩阵。

8-6 画出 1 个顶点、2 个顶点、3 个顶点、4 个顶点和 5 个顶点的无向完全图。试证明在 n 个顶点的无向完全图中，边的条数为 n(n－1)/2。

【解答】

所求图如图 8-4 所示。

【证明】

在有 n 个顶点的无向完全图中，每一个顶点都有一条边与其他某一顶点相连，所以每一个顶点有 n－1 条边与其他 n－1 个顶点相连，总计 n 个顶点有 n(n－1)条边。但在无向图中，顶点 i 到顶点 j 与顶点 j 到顶点 i 是同一条边，所以总共有 n(n－1)/2 条边。

8-7 用邻接表表示图时，顶点个数设为 n，边的条数设为 e，在邻接表上执行有关图的

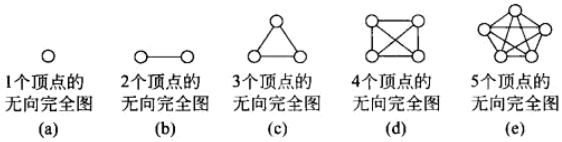


图 8-4

遍历操作时，时间代价是 O(n×e)？还是 O(n+e)？或者是 O(max(n,e))？

【解答】

在邻接表上执行有关图的遍历操作时要遍访图中所有顶点，在访问每一个顶点时要检测与该顶点相关联的所有边，看这些边的另一端的邻接顶点是否访问过。设图中有 n 个顶点，e 条边，则访问所有顶点的时间代价是 O(n)，检测各顶点相关联的边的时间代价是与所有顶点相关联的边的总和，为 O(e)，所以总的时间代价是 O(n+e)。

因为不是每访问一个顶点都要检测所有的边，所以其时间代价不是 O(n×e)；又因为访问顶点和检测边不是各自独立的操作，所以其时间代价也不是 O(max(n, e))。

8-8 DFS 和 BFS 遍历各采用什么样的数据结构来暂存顶点？当要求连通图的生成树的高度最小，应采用何种遍历？

【解答】

深度优先搜索(DFS)一般采用递归算法，需要递归工作栈辅助递归的实现。如果采用非递归算法，需要显式地设置一个栈，暂存遍历时走过的路径以备将来回溯。广度优先搜索(BFS)需要设置一个队列，让所有顶点分层进入队列，以实现层次序的访问。

当要求连通图的生成树的高度最小，应采用广度优先遍历。因为广度优先生成树的高度跟访问的层次数有关，而深度优先生成树的高度与按某条路径能走过的最远路程有关，一般深度优先生成树的高度比广度优先生成树的高度要大。

8-9 采用邻接表存储的图的深度优先遍历算法类似于二叉树的哪种遍历？广度优先遍历算法又类似于二叉树的哪种遍历？

【解答】

采用邻接表存储的图的深度优先遍历算法是递归算法，每次进入算法时先访问顶点并对该顶点做访问标志；再检测它的所有邻接顶点，对所有未访问过的邻接顶点使用深度优先搜索算法递归地进行访问。因此，这种遍历方式类似于二叉树的前序遍历。

图的广度优先遍历算法类似于二叉树的层次序遍历，都需要使用队列执行分层的访问。区别在于，二叉树有明显的层次，而图是按与起始顶点距离的远近，以路径长度递增的次序分层的。

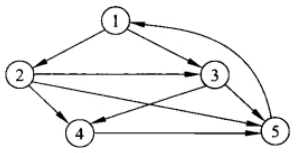


图 8-5

8-10 对于如图 8-5 所示的有向图，试写出：

(1) 从顶点①出发进行深度优先搜索所得到的深度优先生成树；

(2) 从顶点②出发进行广度优先搜索所得到的广度优先生成树。

【解答】

(1) 以顶点①为根的 DFS 生成树不唯一(顶点旁附的数字为访问顺序),如图 8-6 所示。

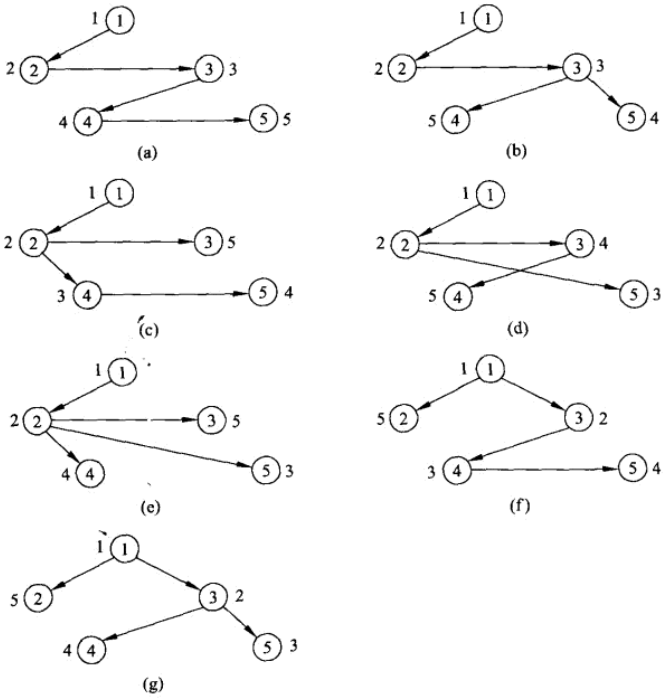


图 8-6

(2) 以顶点②为根的 BFS 生成树如图 8-7 所示。

8-11 什么样的图其最小生成树是唯一的? 用 Prim 算法和 Kruskal 算法求最小生成树的时间各为多少? 它们分别适合于哪类图?

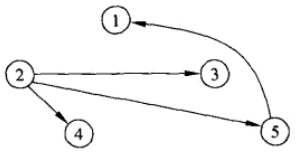


图 8-7

【解答】

对于所有边上的权值都不相同的连通网络,其最小生成树是唯一的。

用 Prim 算法求最小生成树,它是从某个顶点出发,先把该顶点加入生成树顶点集合 S,然后检测所有一个端点在 S,另一个端点不在 S 的边,选出权值最小的边加入生成树,并把其不在 S 的顶点加入 S,然后重复选边和加入的操作,直到所有顶点都加入 S 则算法结束。若使用邻接矩阵作为其存储结构,算法的时间复杂性为 $O(n^2)$,若使用邻接表作为其存储结构,算法的时间复杂性可提高到 $O(e\log_2 e)$。Prim 算法适合于稠密图的情形。

用 Kruskal 算法求最小生成树,它首先把所有边按其权值大小从小到大排列,然后顺序检测各边,如果该边的两个端点不在同一连通分量上,把它加入生成树,同时连通它的两个

端点；如果该边的两个端点在同一连通分量上，则跳过它不加入生成树。重复以上操作直到生成树的边数达到 n－1 为止(n 是图中顶点个数)。此算法中最花时间的是对所有边按其权值大小排序，设图中有 n 个顶点 e 条边，排序时间至少为 $O(elog_2e)$，每次检查一条边，查连通的时间为 $O(log_2e)$。如果采用邻接表存储图，算法的时间复杂性为 $O(elog_2e)$；如果采用邻接矩阵存储图，选边还需要检测矩阵的上三角或下三角部分，时间为 $n(n-1)/2$，算法的时间复杂性达到 $O(n^2+elog_2e)$。Kruskal 算法适合于稀疏图的情形。

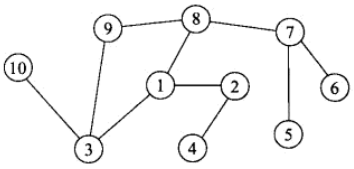


图 8-8

8-12 图 8-8 是一个连通图，请画出：

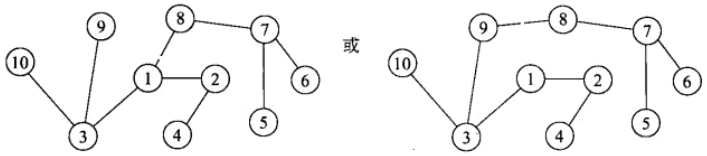
(1) 以顶点①为根的 DFS 树。

(2) 如果有关节点，请找出所有的关节点。

(3) 如果想把该连通图变成重连通图，至少在图中加几条边？如何加？

【解答】

(1) 以顶点①为根的深度优先生成树如图 8-9 所示。



(a) (b)

图 8-9

(2) 关节点为 ①，②，③，⑦，⑧。

(3) 重连通图要求每个顶点至少有两条边与外部相连，另外对于关节点，需要子女有通路通向它的祖先。因此需为度为 1 的顶点加边，并对关节点检查增加子女通向祖先的边即可得到重连通图。在图 8-8 的例子中，至少加三条边。一条边是(1，10)，消除关节点③；一条边(4，5)，消除关节点①，②，⑧；一条边(5，6)，继续消除关节点⑦。(解答不唯一，图 8-10 给出一种解答)

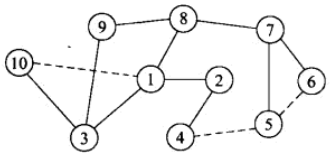


图 8-10

8-13 如何利用拓扑排序将一个有向无环图的邻接矩阵中的非零元素集中到对角线以上？

【解答】

可对一个有向无环图先做拓扑排序，把所有顶点排到一个拓扑有序序列中，然后对序列中所有顶点按次序重新编号，得到的图的邻接矩阵可把非零元素都集中到对角线以上。

例如图 8-11(a)是 5 个顶点的有向无环图，它的邻接矩阵如图 8-11(b)所示，经过拓扑排序得到的一个拓扑有序序列是 1，5，3，4，2，按序列次序重新对各顶点编号为 1→1，5→2，3→3，4→4，2→5，经过重新编号的图如图 8-11(c)所示，其邻接矩阵如图 8-11(d)所示，所

有非零元素都集中到矩阵对角线以上的部分了。

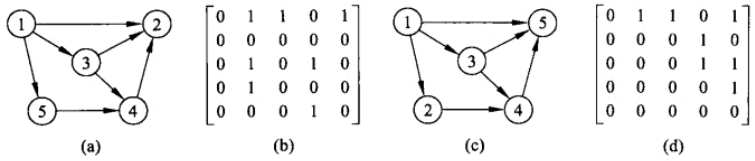


图 8-11

8-14 什么样的有向无环图的拓扑序列是唯一的？对于一个有向图，不用拓扑排序，如何判断图中是否存在环？

【解答】

如果一个有向无环图是一个全序图，即所有顶点之间都有优先(领先)关系，其拓扑排序的结果是唯一的。如图 8-12 所示，其拓扑序列为 1，2，3，4，5。

除了拓扑排序之外，如果对一个有向图做深度优先遍历，如果从某个顶点出发，经过某些边又回到这个顶点(用 visited[]判断)，则可确定图中存在环。

8-15 试证明：对于一个无向图 G=(V，E)，若 G 中各顶点的度均大于或等于 2，则 G 中必有回路。

【证明】

对于一个无向图 G=(V，E)，若 G 中各顶点的度均大于或等于 2，则每个顶点都至少有一条通路通过它。如果从某一个顶点出发，沿这条通路就会走到该顶点的一个邻接顶点，再沿那个顶点的通路又会走到下一个顶点，……，当走到开始顶点的另一邻接顶点时，沿该顶点的通路可能回到最初的开始顶点，所以这样的无向图必定存在回路。

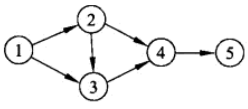


图 8-12

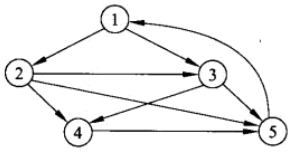


图 8-13

8-16 图 8-13 是基于元素 1 到 5 的一些优先关系的集合，试问它们是否定义了一个偏序关系？为什么？

【解答】

不是。首先复习一下偏序关系的定义。

定义：设 R 是集合 A 上的二元关系，若 R 是自反的，反对称的和传递的，则称 R 是 A 上的偏序关系。记为≤(注意，此符号在这里并不意味着小于或等于)。所谓“自反”是指对于任意 a∈A，有 aRa。所谓“反对称”是指对任意 a，b∈A，如果 aRb 且 bRa，必有 a=b。就是说当 a≠b 时，若有(a，b)∈R，则(b，a)∉R。所谓“传递”是指对于任意 a，b，c∈A，如果 aRb 且 bRc，必有 aRc 。在图 8-13 中，根据边<1，3>和<3，5>可知 1≤3，3≤5，通过传递应有 1≤5，然而通过<5，1>则有 5≤1，违反了偏序关系中的反对称的原则，所以

图 8-13 不是偏序关系。

8-17 对图 8-14 所示的 AOE 网络，回答下列问题：

（1）这个工程最早可能在什么时间结束。

（2）求每个事件的最早开始时间 Ve[i]和最迟开始时间 Vl[i]。

（3）求每个活动的最早开始时间 Ae[k]和最迟开始时间 Al[k]。

（4）确定哪些活动是关键活动。画出由所有关键活动构成的图，指出哪些活动加速可使整个工程提前完成。

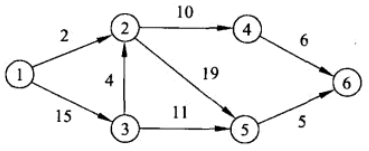


图 8-14

【解答】

按拓扑有序的顺序计算各个顶点的最早可能开始时间 Ve 和最迟允许开始时间 Vl。然后再计算各个活动的最早可能开始时间 Ae 和最迟允许开始时间 Al，根据 Al-Ae 是否等于 0 来确定关键活动，从而确定关键路径。

各个事件的最早可能开始时间 Ve 和最迟允许开始时间 Vl 的计算结果如表 8-1 所示。

表 8-1

	1	2	3	4	5	6
Ve	0	19	15	29	38	43
Vl	0	19	15	37	38	43

各个活动的最早可能开始时间 Ae 和最迟允许开始时间 Al 的计算结果如表 8-2 所示。

表 8-2

	<1, 2>	<1, 3>	<3, 2>	<2, 4>	<2, 5>	<3, 5>	<4, 6>	<5, 6>
Ae	0	0	15	19	19	15	29	38
Al	17	0	15	27	19	27	37	38
Al-Ae	17	0	0	8	0	12	8	0

此工程最早完成时间为 43。关键路径为<1，3><3，2><2，5><5，6>，如图 8-15 所示。

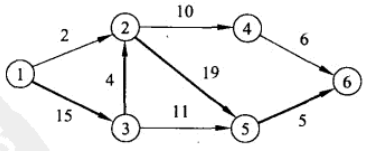


图 8-15

二、练习题

8-18 已知一个有向图的邻接表，试编写一个算法，计算各顶点的入度。

【解答】

通常统计有向图邻接表各个顶点的边链表中结点个数，可得到顶点的出度。为了统计各个顶点的入度，必须检测各个顶点的边链表，通过每个边结点内存储的边的终顶点的顶点号，计算顶点的入度。算法参数表中有一个数组 inDegree[]，通过它返回各个顶点的入度值。

```
template <class T, class E>
```

```
void calc_inDegree (Graphlnk<T,E>& G, int inDegree[]) {
    int i, w, n=G.NumberOfVertices();                    //取图中顶点个数
    for (i=0; i<n; i++) inDegree [i]=0;                  //数组 inDegree 初始化
        for (i=0; i<n; i++) {                            //逐个顶点检测
            w=G.getFirstNeighbor(i);                     //找顶点 i 的第一个邻接顶点 w
            while (w !=-1) {                             //若邻接顶点 w 存在
                inDegree[w]=inDegree[w]+1;
                G.getNextNeighbor(i, w);                 //取 i 排在 w 后的下一邻接顶点
            }
        }
};
```

8-19 试扩充深度优先搜索算法，在遍历图的过程中建立生成森林的子女-兄弟链表。算法的首部为

```
void DFS (Graph<T,E>& G, T v, bool visited[], TreeNode<T,E> * & t)
```

其中，指针 t 指向生成森林上具有图顶点 v 信息的根结点。（提示：在继续按深度方向从根 v 的某一未访问过的邻接顶点 w 向下遍历之前，建立子女结点。但需要判断是作为根的第一个子女还是作为其子女的右兄弟链入生成树）

【解答】

为建立生成森林，需要先给出建立生成树的算法，然后再在遍历图的过程中，通过一次次地调用这个算法，以建立生成森林。由于使用了树的子女—兄弟链表和图的操作，在实现程序中要加入 #include "CSTree.h" 和 #include "Graphlnk.h"。

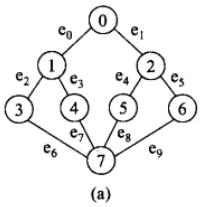
```
template<class T, class E>
void DFS_Tree (Graph<T, E>& G, int v, int visited [ ], CSTreeNode<T> * & t) {
//从图的顶点 v 出发，深度优先遍历图，建立以 t (已在上层算法中建立)为根的生成树
    visited[v]=1;  int first=1; CSTreeNode<T> * p, * q;
    int w=G.getFirstNeighbor (v);                        //取第一个邻接顶点
    while (w !=-1) {                                     //若邻接顶点存在
        if (visited[w]==0) {                             //且该邻接结点未访问过
                p=new CSTreeNode<T>;                     //建立新的生成树结点
                p->data=G.getValue (w);
                p->firstChild=p->nextSibling=NULL;
                if (first==1)                            //若根 * t 还未链入任一子女
                    {t->firstChild=p;  first=0;}         //结点 p 成为根 t 的第一个子女
                else q->nextSibling=p;                   //结点 p 成为 q 的下一个兄弟
                q=p;                                     //指针 q 指示兄弟链最后结点
                DFS_Tree (G, w, visited, q);             //从 q 向下建立子树
            }
            w=G.getNextNeighbor (v, w);                  //取顶点 v 下一邻接顶点
    }
};
```

下一个算法用于建立以子女-兄弟链表为存储表示的生成森林。

```
template<class T, class E>
```

```
void DFS_Forest (Graph<T, E>& G, CSTree<T>& RT) {
//从图的顶点 v 出发，深度优先遍历图，建立以子女-兄弟链表表示的生成森林 RT
    RT.root=NULL;  int n =G.NumberOfVertices ();        //顶点个数
    CSTreeNode<T> * p, * q;  int v;
    int * visited =new int [n];                         //建立访问标记数组
    for (v=0; v<n; v++) visited[v]=0;
    for (v=0; v<n; v++) {                               //逐个顶点检测
        if (visited[v]==0) {                            //若尚未访问过
            p =new CSTreeNode<T>;                       //建立新结点 p
            p->data=G.getValue (v);
            p->firstChild=p->nexrSibling=NULL;
            if (RT.root==NULL) RT.root=p;               //空生成森林，新结点成为根
            else q->nextSibling=p;                      //否则结点 p 成为 q 的下一兄弟
            q=p;
            DFS_Tree (G, v, visited, p);                //建立以 p 为根的生成树
        }
    }
};
```

8-20 表示图的另一种方法是使用关联矩阵 INC[n][e]。其中，一行对应于一个顶点，一列对应于一条边，n 是图中顶点数，e 是边数。因此，如果边 j 依附于顶点 i，则 INC[i][j]＝1。图 8-16(b) 就是图 8-16(a) 所示图的关联矩阵。注意，在使用关联矩阵时应把图 8-16(a)中所有的边从上到下、从左到右顺序编号。



(a)

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

(b)

图 8-16

(1) 如果 ADJ 是图 G＝(V, E) 的邻接矩阵，INC 是关联矩阵，试说明在什么条件下将有 ADJ＝INC×INC^{T}－I，其中，INC^{T}是矩阵 INC 的转置矩阵，I 是单位矩阵。两个 n×n 的矩阵的乘积 C＝A×B 定义为

$$c_{ij}=\bigcup_{k=0}^{n-1} a_{ik} \cap b_{kj}$$

公式中的“∪”定义为按位加，“∩”定义为按位乘。

(2) 设用邻接矩阵表示的图的定义如下。

```
const int MaxEdges=50;                                  //最大边数
const int MaxVertices=10;                               //最大顶点数
typedef int DataType;
```

```
typedef struct {                                        //图的结构定义
    DataType VerticesList [MaxVertices];                //顶点表
    int ADJ[MaxVertices][MaxVertices];                  //邻接矩阵
    int CurrentEdges, CurrentVertices;                  //当前边数和当前顶点数
} Graph;
```

试仿照上述定义，建立用关联矩阵表示的图的结构。

(3) 以关联矩阵为存储结构，实现图的 DFS 的递归算法。

【解答】

(1) 当图中的顶点个数等于边的条数时，$ADJ=INC\times INC^{T}-I$ 成立。

(2) 设用关联矩阵表示的图的定义如下。

```
const int MaxEdges=50;                                  //最大边数
const int MaxVertices=10;                               //最大顶点数
typedef int DataType;
typedef struct {                                        //图的结构定义
    DataType VerticesList [MaxVertices];                //顶点表
    int associatMatrix[MaxVertices][MaxEdges];          //关联矩阵
    int CurrentEdges, CurrentVertices;                  //当前边数和当前顶点数
} GraphAMT;
```

(3) 与使用邻接矩阵执行 DFS 遍历算法比较，使用关联矩阵执行 DFS 要解决寻找某指定顶点的全部邻接顶点问题。在关联矩阵中，第 i 行表示编号为 i 的一个顶点，第 j 列表示编号为 j 的一条边，设关联矩阵为 A，则 A[i][j]表示顶点 i 是否与边 j 关联：若 A[i][j]=1 标志着顶点 i 与边 j 关联，此时可在第 j 列找另一个 A[i′][j]=1 的元素，顶点 i′应是顶点 i 的邻接顶点。为此，可以定义两个操作寻找顶点 v 的第一个邻接顶点和下一个邻接顶点：

```
int getFirstNeighbor (GraphAMT& G, int v) {
//取图 G 中顶点 v 的第一个邻接顶点，找到则返回邻接顶点号，未找到则返回-1
    int i, j;
    for (j=0; j<G.CurrentEdges; j++)                    //在 v 行寻找与之关联的边 j
        if (G.associatMatrix[v][j]==1)
            for (i=0; i<G.CurrentVertices; i++)         //在 j 列寻找邻接顶点 i
                if (i !=v && G.associatMatrix[i][j]==1)
                    return i;                           //找到，返回邻接顶点号 i
    return -1;
};
int getNextNeighbor (GraphAMT& G, int v, int w) {
//取图 G 中邻接顶点 w 的下一邻接顶点，函数返回找到的下一邻接顶点号
    int i, j, k;
    for (j=0; j<G.CurrentEdges; j++)                    //寻找 w 与 v 所在的边 j
        if (G.associatMatrix[w][j]==1 && G.associatMatrix[v][j]==1) break;
    for (k=j+1; k<G.CurrentVertices; k++)               //向后继续找与 v 关联的边
        if (G.associatMatrix[v][k]==1)
            for (i=0; i<G.CurrentVertices; i++)         //在 k 列寻找邻接顶点 i
                if (i !=v && G.associatMatrix[i][k]==1)
```

```
                return i;                                    //找到,返回邻接顶点号 i
    return -1;
};
```

只要有了这两个操作,就可以使用常规的 DFS 算法遍历图了。算法描述如下:

```
#include <iostream.h>
void DFS (GraphAMT& G, int v, bool visited[]) {
//从顶点位置 v 出发, 以深度优先的次序访问所有可读入的尚未访问过的顶点
//算法中用到一个辅助数组 visited, 对已访问过的顶点作访问标记
    cout<<G.VerticesList[v]<<' ';  visited[v]=true;    //访问顶点 v
    int w=getFirstNeighbor(G, v);                      //找顶点 v 第一个邻接顶点
    while (w !=-1) {                                   //若邻接顶点 w 存在
        if (!visited[w]) DFS (G, w, visited);          //递归访问顶点 w
        w=getNextNeighbor (G, v, w);                   //取下一个邻接顶点
    }
};
void Components (GraphAMT& G) {
//通过 DFS,找出无向图的所有连通分量
int i;
bool * visited=new bool[MaxVetices];                   //visited 记录顶点是否访问过
    for (i=0; i<G.CurrentVertices; i++) visited[i]=false;
    for (i=0; i<n; i++)                                //顺序扫描所有顶点
        if (!visited[i]) {                             //若没有访问过, 则访问
            DFS(G, i, visited);
        OutputNewComponent();                          //输出这个连通分量
        }
delete [] visited;
};
```

8-21 编写一个完整的程序,首先定义堆和并查集的结构类型和相关操作,再定义 Kruskal 求连通网络的最小生成树算法的实现。并以图 8-17 为例,写出求解过程中堆、并查集和最小生成树的变化。

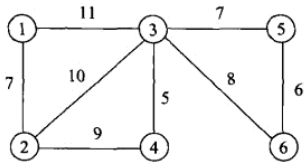


图 8-17

【解答】

求解过程的第一步是对所有的边按其权值建堆,其过程如图 8-18 所示。

求解过程中并查集与堆的变化如图 8-19 所示。

最后得到的生成树如图 8-20 所示。

完整的程序参看教材 8.4.1 节。

8-22 在有向图中的一个欧拉回路(Euler circuit)是这样的一个环:其上的每一条边被访问一次且仅被访问一次。

(1) 试证明一个有向图存在欧拉回路的充要条件是该图必须是强连通的且每一个顶点有相同的入度与出度;

(2) 设图中的顶点数为 n,试描述有向图的数据结构并编写一个时间复杂性为 O(n)的

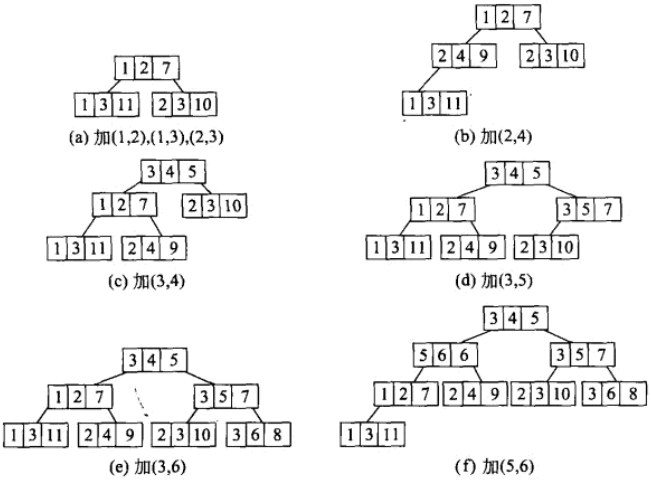


图 8-18

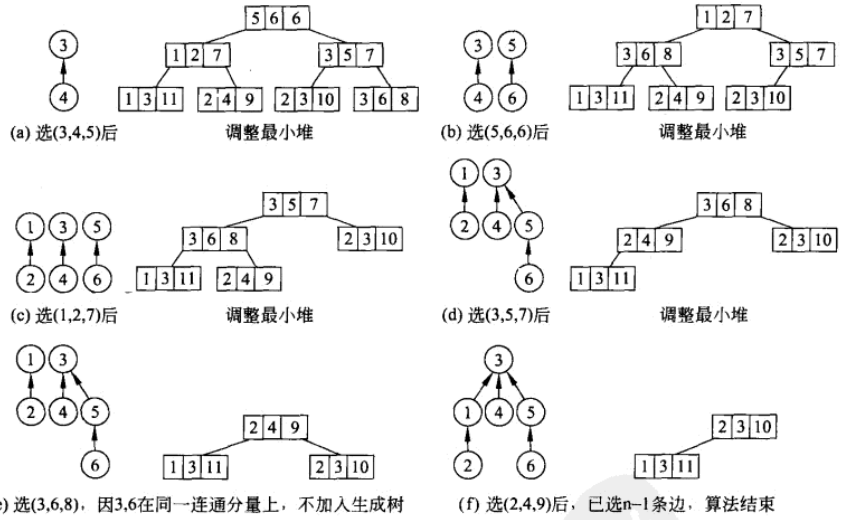


图 8-19

算法，在有向图中查找一条欧拉回路(如果它存在)。

【解答】

(1) 证明。

必要性：如果一个有向图存在欧拉回路，意味着这个图是强连通图，且在这个回路上每一条边被访问一次且仅被访问一次。就是说，从这条边进入其头顶点仅一次，并从此顶点进

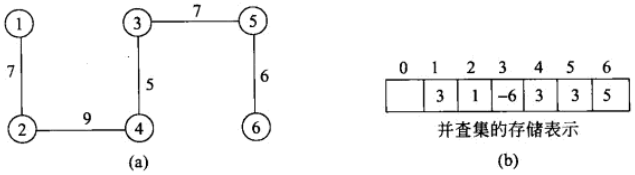


图 8-20

入回路中的下一条边仅一次，下一次可能从其他边又进入此顶点，但绝不会从此顶点进入上次走过的边。对于每个顶点，一进就有一出，顶点的入度等于出度。

充分性：如果一个图是强连通图，则一定存在回路，使得从某一顶点出发，可以到达图中其他顶点；还可以从其他顶点到达此顶点。又因每个顶点的入度等于出度，每次从不同方向进入该顶点的边一定会导致从不同方向走离开这个顶点的边，因此可以得到结论，图中每条边可以仅访问一次且仅访问一次，在图中遍历可以不走重复的边，这就是欧拉回路。

(2) 在有向图中寻找欧拉回路的算法。

例如，图 8-21(a)是一个具有欧拉回路的有向图，有 4 个顶点，6 条边。它的邻接表存储结构如图 8-21(b)所示，每个边链表中的结点有 3 个域：头顶点编号(vertex)、边编号(edgeno)和边链表的链接指针(link)，如图 8-21(c)所示。

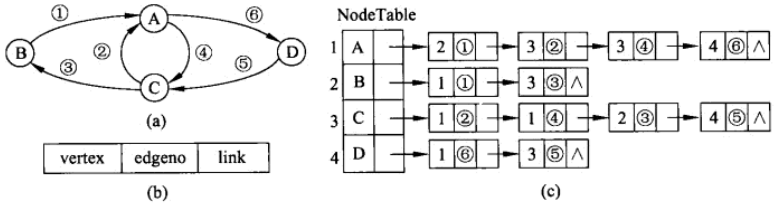


图 8-21

利用 DFS 算法解决寻找欧拉回路问题。算法设置一个边访问标志数组 visited[]，当 visited[i]=0 表示第 i 条边未访问过，一旦访问了第 i 条边，即将 visited[i]改为 1。

算法的思路是：从某个顶点 start 出发，沿某一条边前进并输出边的编号及记下该边已访问。当走到下一顶点时，看有无其他没有访问过的边。若有，从这个顶点出发继续这种访问；否则回溯，退到前一顶点，寻找还有没有未访问过的边。如果 visited 数组中所有边都已访问，则算法结束。

```
#include <iostream.h>
const int MaxEdges=50;                          //最大边数
const int MaxVertices=10;                       //最大顶点数
typedef char DataType;
typedef struct node1 {                          //边结点的结构定义
    int vertex;                                 //邻接顶点号
    int edgeno;                                 //相关边号
    struct node1 * link;                        //链接指针
} EdgeNode;
```

```
typedef struct node2 {                                  //顶点的结构定义
    DataType data;                                      //顶点数据
    struct node2 * adj;                                 //出边表指针
} VertexNode;
typedef struct {                                        //图的结构定义
    VertexNode NodeTable [MaxVertices];                 //顶点表
    int CurrentEdges, CurrentVertices;                  //当前边数和当前顶点数
} Graphlnk;
```

下面给出按深度优先搜索方法求解欧拉问题的递归算法。

```
void Euler_dfs (Graphlink& G, int start, int visited[ ]) {
    cout<<G.NodeTable[start].data<<"->";                //输出顶点数据
    EdgeNode * p=G.NodeTable[start].adj;                //找第一条关联的边
    while (p !=NULL) {                                  //出边表非空
        int j=p->edgeno;                                //边的编号
        if (!visited[j]) {                              //该边未走过
            visited[j]=1;                               //作边访问标记
            Euler_dfs (G, p->vertex, visited);          //按邻接顶点递归下去
        }
        p=p->link;                                      //查下一条关联的边
    }
};
```

主程序如下。

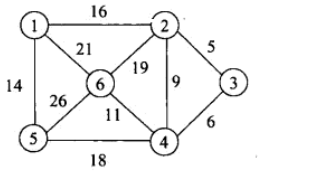
```
void main() {
    int count, i, j, k;
    Graphlnk G;
    cout<<"输入顶点数:";  cin>>G.currentVertices;
    G.NodeTable=new vertexNode[MaxVertices];            //创建邻接表数组
    for (i=0; i<G.currentVertices; i++) {               //输入顶点数据，指针置空
        cin>>G.NodeTable[i].data;
        G.NodeTable[i].adj=NULL;
        }
    cout<<"输入各条边!" <<endl;                         //输入各边建立邻接表
    G.CurrentEdges=0;
    cin>>i>>j>>k;                                      //输入边,i是边号码,j, k是顶点
    while (i !=0) {                                     //i为0，表示输入结束
        EdgeNode * p=new EdgeNode;                      //链入第j号链表
        p->edgeno=i;  p->vertex=k;
        p->link=G.NodeTable[j].adj;  G.NodeTable[j].adj=p;
        G.CurrentEdges++;                               //链入第k号链表
        cin>>i>>j>>k >>;
    }
    int * visited=new int[G.CurrentEdges];              //创建访问标志数组
    for (i=0; i<G.CurrentEdges; i++) visited[i]=0;
    Euler_dfs (G, 0, visited);                          //深度优先搜索
```

```
        count=0;
        for (i=0; i<G.CurrentEdges; i++)
            if (visited[i]) count++;
        if (count<G.CurrentEdges) {cout<<"图中不存在欧拉回路!\n";}
        else {cout<<"图中存在欧拉回路!\n";}
    delete [ ] visited;
};
```

8-23 计算连通网的最小生成树的 Dijkstra 算法可简述如下：将连通网所有的边以方便的次序逐条加入到初始为空的生成树的边集合 S 中。每次选择并加入一条边时，需要判断它是否会与先前加入 S 中的边构成回路。如果构成了回路，则从这个回路中将权值（花费）最大的边退选。试设计一个求最小生成树的算法。要求以邻接矩阵作为连通网的存储结构，并允许在运算后改变邻接矩阵的结构。

【解答】

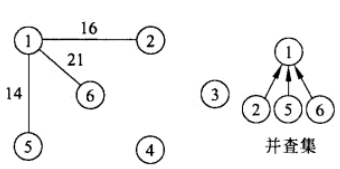
下面以图 8-22 所示的邻接矩阵作为连通网络的存储表示，并以并查集作为判断是否出现回路的工具，分析算法的执行过程如图 8-23。



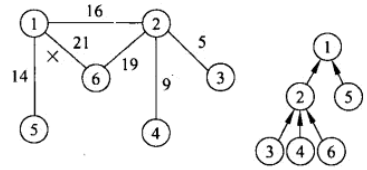
Edge=

	①	②	③	④	⑤	⑥	
	0	16	∞	∞	14	21	①
	–	0	5	9	∞	19	②
	–	–	0	6	∞	∞	③
	–	–	–	0	18	11	④
	–	–	–	–	0	26	⑤
	–	–	–	–	–	0	⑥

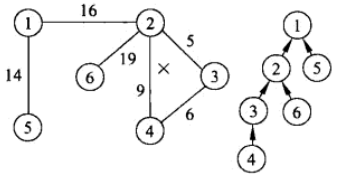
图 8-22



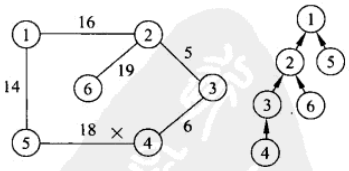
(a) 取矩阵第一行的边得并查集，表明4个结点在同一连通分量上。加入各边未出现环



(b) 取矩阵第二行的边，19加入后形成环，把最大的21删去，并查集中删去<6,1>增加<6,2>

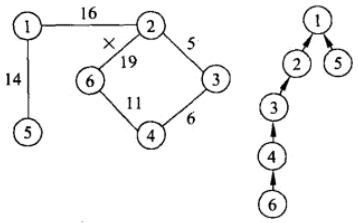


(c) 取矩阵第三行的边，6加入后形成环，把最大的9删去，并查集中删去<4,2>增加<4,3>

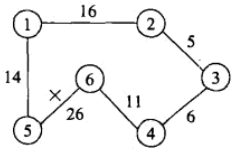


(d) 取矩阵第四行的边，18加入后形成环，它就是最大的，删去，并查集不变

图 8-23



(e) 取矩阵第四行的边，11加入后形成环，把最大的19删去，并查集中删去<6,2>增加<6,4>



(f) 取矩阵第五行的边，26加入后形成环，它就是最大的，删去，并查集不变

图 8-23（续）

最终得到的最小生成树如图 8-24 所示。

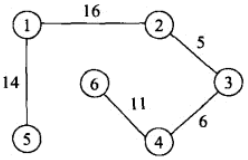


图 8-24

算法的思路如下：

（1）并查集初始化：将所有顶点置为只有一个顶点的连通分量；

（2）检查所有的边：

① 若边的两个端点 i 与 j 不在同一连通分量上（即 i 与 j 在并查集中不同根），则连通之（合并并查集）；

② 若边的两个端点 i 与 j 在同一连通分量上（即 i 与 j 在并查集中同根），则：

✧ 在并查集中寻找离 i 与 j 最近的共同祖先结点；

✧ 分别从 i 与 j 向上检测具有最大权值的边；

✧ 在并查集上删除具有最大权值的边，加入新的边。

下面给出实现算法：

```
#define maxValue 10 000                                    //问题中不可能出现的大数
#define DefaultEdges 20                                    //生成树默认边数
#define DefaultVertices 20                                 //图默认边数
struct MSTEdgeNode {                                       //最小生成树边结点的类声明
    int tail, head;                                        //两顶点位置
    float cost;                                            //边上的权值
    MSTEdgeNode():tail(-1), head(-1), cost(0) {}
};
class MinSpanTree {                                        //最小生成树的类定义
protected:
    MSTEdgeNode * edgevalue;                               //用边值数组表示树
    int maxSize, n;                                        //数组的最大元素个数和当前个数
public:
    MinSpanTree (int sz=DefaultEdges):maxSize (sz), n (0) {
        edgevalue=new MSTEdgeNode[sz];
    }
    bool Insert (MSTEdgeNode item) {
        if (n<maxSize) {adgevalue[n++]=item;  return true;}
```

```
        return false;
    };
};
class disjoint {                                        //并查集类定义
    int parent[DefultVertices];                         //父指针数组
public:
    disjoint () {                                       //初始化,每棵树 1 个结点
        for (int i=0; i<DefaultVertices; i++) parent[i]=-1;
    };
    int Find (int i) {                                  //查找结点 i 所在树的根
        while (parent[i] >=0) i=parent[i];
        return i;
    };
    void Unions (int i, int j) {                        //合并以 i 为根和以 j 为根的树
        parent[j]=i;
    };
    int Father (int i) {
        if (i >=0) return parent[i];
        else return i;
    }
    int commonAncestors (int i, int j) {                //查找 i 与 j 的共同祖先
        int p=0, q=0, k;
        for (k=i; parent[k] >=0; k=parent[k]) p++;      //i 到根的路径长度 p
        for (k=j; parent[k] >=0; k=parent[k]) q++;      //j 到根的路径长度 q
        while (parent[i] !=parent [j]) {
            if (p<q) {j=parent[j];  q--;}
            else if (p>q) {i=parent[i];  p--;}
                else {i=parent[i];  j=parent[j];}
        }
        return i;
    };
    int Remove (int i) {                                //结点 i 移出它所在的树
        int k=0;
        while (k<DefaultVertices && parent[k] !=i) k++;
        if (k<DefaultVertices)                          //链接摘下 i 后的双亲链
            {parent[k]=parent[i];  parent[i]=-1;  return k;}
        else return -1;
    }
};
void Dijkstra (Graph<int, float>& G, MinSpanTree& ST ) {
    MSTEdgeNode nd;
    int n=G.NumberOfVertices();;
    int i, j, p, q, k, s1, s2, t1, t2;  float w, max1, max2;
    disJoint D;                                         //建立并查集并初始化
    for (i=0; i<n-1; i++)                               //检查所有的边
        for (j=i+1; j<n; j++) {
            w=G.getWeight(i, j);
```

```
            if (w<maxValue) {                          //边存在
                p=D.Find(i);  q=D.Find(j);             //判 i 与 j 是否在同一连通分量上
                if (p !=q) {
                    D.Unions (i, j);                   //i 与 j 不在同一连通分量上连通之
                    nd.tail=j;  nd.head=i;  nd.cost=w;  ST.Insert (nd);
                }
            }
            else {                                     //i 与 j 在同一连通分量上
                k=D.commonAncestors(i, j);             //寻找离 i 与 j 最近的祖先结点
                p=i;  q=D.Father(p);                   //从 i 到 k 找权值最大的边(s1, s2)
                max1=-maxValue;
                while (q <=k) {
                    if (G.getWeight(p, q)>max1)
                        {max1=G.getWeight(p, q); s1=p; s2=q;}
                    p=q;  q=D.Father(p);
                }
                p=j;  q=D.Father(p);                   //从 j 到 k 找权值最大的边(t1, t2)
                max2=-maxValue;
                while (q <=k) {
                    if (G.getWeight (p, q)>max2)
                        {max2=G.getWeight (p, q); t1=p; t2=q;}
                    p =q;  q=D.Father(p);
                }
                if (max1 <=max2) {s1=t1;  s2=t2;  max1=max2;}
                if (D.Remove(s1) >=0) {                //从并查集删去权值大的边
                    D.Union(s1, s2);                   //插入新边
                    nd.tail=s2;  nd.head=s1;  nd.cost=max1;  ST.Insert(nd);
                }
            }
        }
};
```

8-24 以图 8-25 为例，按 Dijkstra 算法计算得到的从顶点 A 到其他各个顶点的最短路径和最短路径长度。

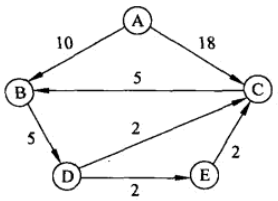


图 8-25

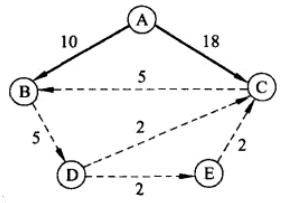
【解答】

求解最短路径问题的结果如表 8-3 所示。

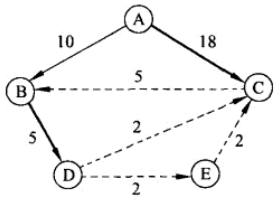
表 8-3

源点	终点	最短路径				最短路径长度			
A	B	<A,B>	<A,B>	<A,B>	<A,B>	[10]			
	C	<A,C>	<A,C>	<A,C><A,B,D,C>	<A,C><A,B,D,C>	18	18	[17]	
	D	—	<A,B,D>	<A,B,D>	<A,B,D>	∞	[15]		
	E	—	—	<A,B,D,E>	<A,B,D,E>	∞	∞	17	[17]

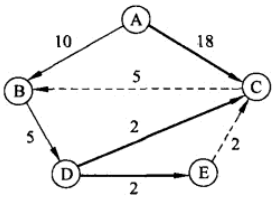
则实现过程见图 8-26。



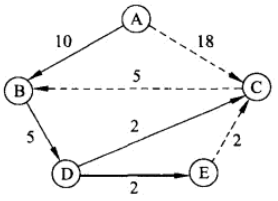
(a) 选中<A,B,10>,到B最短路径为10



(b) 选中<A,B,D,15>,到D最短路径为15



(c) 选中<A,B,D,C,17>,到C最短路径为17



(d) 选中<A,B,D,E,17>,到E最短路径为17

图 8-26

8-25 在以下假设下,重写 Dijkstra 算法:

(1) 用邻接表表示有向带权图 G,其中每个边结点有 3 个域:邻接顶点 vertex,边上的权值 length 和边链表的链接指针 link。

(2) 用集合 T=V(G)-S 代替 S(已找到最短路径的顶点集合),利用链表来表示集合 T。

试比较新算法与原来的算法,计算时间是快了还是慢了,给出定量的比较。

【解答】

(1) 用邻接表表示的带权有向图的类定义。

```
#define maxValue 100 000                          //默认最大权值
#define DefaultSize 20                            //默认顶点个数
struct EdgeNode {                                 //边结点的定义
    int vertex;                                   //边的另一顶点位置
    float length;                                 //边上的权值
    EdgeNode * link;                              //下一条边链指针
    EdgeNode () {}                                //构造函数
    EdgeNode (int num, float cost) {              //构造函数
        vertex=num;  length=cost;  link=NULL;
    }
};
struct VertexNode {                               //顶点的定义
    char data;                                    //顶点的名字
    EdgeNode * adj;                               //边链表的头指针
};
struct Graphlnk {                                 //图的类定义
    VertexNode * NodeTable;                       //顶点表 (各边链表的头结点)
```

```
    int numVertices;                                    //当前顶点个数
    int numEdges;                                       //当前边数
    Graphlnk (int sz=DefaultVertices);                  //构造函数
    ~Graph();                                           //析构函数
};
```

(2) 用集合 T=V(G)－S 代替 S（已找到最短路径的顶点集合），利用链表来表示集合 T。

集合 T 用有序链表表示，数据域为顶点序号，链表 T 中的顶点都是未找到最短路径的顶点。另外在参数表中设置两个数组 pre[]和 len[]，分别记录并返回已找到的顶点 0 到其他各顶点的最短路径及最短路径长度。算法的主要思路是：

① 对数组 pre、len 及链表 T 初始化，记录顶点 0 到各个顶点的初始最短路径及其长度。

② 扫描链表 T，寻找链表 T 中各顶点到顶点 0 的当前最短路径中长度最小者，记为 u。

③ 在邻接表中扫描第 u 个顶点的出边表，确定每一边的邻接顶点号 k。若顶点 k 的最短路径没有选中过，比较绕过顶点 u 到顶点 k 的路径长度和原来顶点 0 到顶点 k 的最短路径长度，取其小者作为从顶点 0 到顶点 k 的新的最短路径。

④ 重复执行②、③步，直到图中所有顶点的最短路径长度都已选定为止。

算法的实现如下：

```
#include "SortedChain.h"
void ShortestPath (Graphlnk& G, int pre[ ], float len[ ]) {
//对于图 G,使用 Dijkstra 算法求从顶点 0 到其他各顶点的最短路径,数组 pre 返回
//每个顶点求到的最短路径(路径上前一顶点序号),len 返回相应最短路径长度
    SortedChain<int, int>T;                             //未选定最短路径顶点链表
    int i, k, u;  ChainNode<int, int> * q;
    for (i=1; i<G.numVertices; i++) {
        pre[i]=0;  len[i]=maxValue;                     //辅助数组初始化
        T.Insert (i);                                   //形成有序链表 T
    }
    EdgeNode * p=G.NodeTable[0].adj;
    while (p !=NULL)
        {len[p->vertex]=p->length;  p=p->link;}
    while (1) {                                         //循环检测链表 T
        q=T.Begin();
        if (q->link==NULL) break;                       //链表仅剩一个顶点，跳出循环
        float min=maxValue;  u=0;
        while (q !=NULL) {                              //链表还有剩余顶点未确定最短路径
            i=q->data;                                  //取剩余顶点号
            if (len[i]<min) {min=len[i];  u=i;}         //寻找最短路径长度结点 u
                q=q->link;
          }
          p=G.NodeTable[u].adj;
          while (p !=NULL) {                            //比较绕过 u 到其他顶点的路径长度
            k=p->vertex;                                //顶点 k 是 u 的邻接顶点
            if (T.Search(k) !=NULL && len[u]+p->length<len[k])
```

```
                {len[k]=len[u]+p->length;  pre[k]=u;}
            p=p->link;
        }
        T.Remove(u, u);                          //在链表 T 中删除顶点 u, 返回 u
    }
}
```

因为是在邻接表上实现 DijkStra 算法，时间复杂性为 O(n+e)，其中 n 是图中顶点个数，e 是图中边数。而原来使用邻接矩阵作为图的存储结构，算法的时间复杂性为 $O(n^2)$。

8-26 若用邻接表表示图 G，试重写 Bellman-Ford 算法。在邻接表的边结点中增加一个记录边上的权值的域 length。并以图 8-27 为例，验证新算法的正确性。

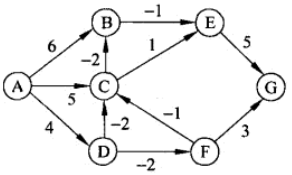


图 8-27

【解答】

Dijkstra 算法要求带权有向图各边上的权值非负，而 Bellman-Ford 算法没有此限制。教材上的 Bellman-Ford 算法是基于邻接矩阵实现的。它的思路是构造一个最短路径长度数组序列 $dist^1[u]$，$dist^2[u]$，……，$dist^{n-1}[u]$。其中，$dist^1[u]$是从源点 v 到终点 u 的只经过一条边的最短路径的长度，并有 $dist^1[u]$=Edge[v][u]；而 $dist^2[u]$是从源点 v 最多经过两条边到达终点 u 的最短路径的长度，$dist^3[u]$是从源点 v 出发最多经过不构成带负长度边回路的三条边到达终点 u 的最短路径的长度，……，$dist^{n-1}[u]$是从源点 v 出发最多经过不构成带负长度边回路的 n−1 条边到达终点 u 的最短路径的长度。算法的最终目的是计算出 $dist^{n-1}[u]$。

如果用邻接表实现算法，主要是把有关邻接矩阵的操作改为邻接表操作。邻接表使用 8-25 题给出的类定义。算法采用广度优先搜索方法分层求最短路径，层次等于经过顶点数。为控制层次，在算法中直接定义队列，并使用一个辅助变量 last 控制每层的结尾。下面给出基于邻接表实现的计算有向带权图的最短路径长度的 Bellman-Ford 算法。算法中使用了同一个 dist[u]数组来存放一系列的 $dist^k[u]$，算法结束时 dist[u]中存放的是 $dist^{n-1}[u]$。

```
void Bellman-Ford (Graphlnk& G, int v, float dist[], int path[], int& level) {
//在有向带权图中有的边具有负的权值。从顶点 v 找到所有其他顶点的最短路径
    int i, k, u;  float w;
    int Q[DefaultVertices];  int front=0, rear=0, last;
    for (i=0; i<G.numVertices; i++) {dist[i]=0;  path[i]=-1;}
    EdgeNode * p=G.NodeTable[v].adj;             //dist 和 path 数组初始化
    while (p !=NULL) {
        dist[p->vertex]=p->length;  path[p->vertex]=v;
        raer=(rear+1)% DefaultVertices;
        Q[rear]=p->vertex;                       //第一层邻接顶点进队
        p=p->link;
    }
    last=rear; level=1;                          //第一层结点在队列中的尾
    while (front !=rear) {                       //队列不空，循环
        front=(front+1)% DefaultVertices;
        u=Q[front];                              //出队
```