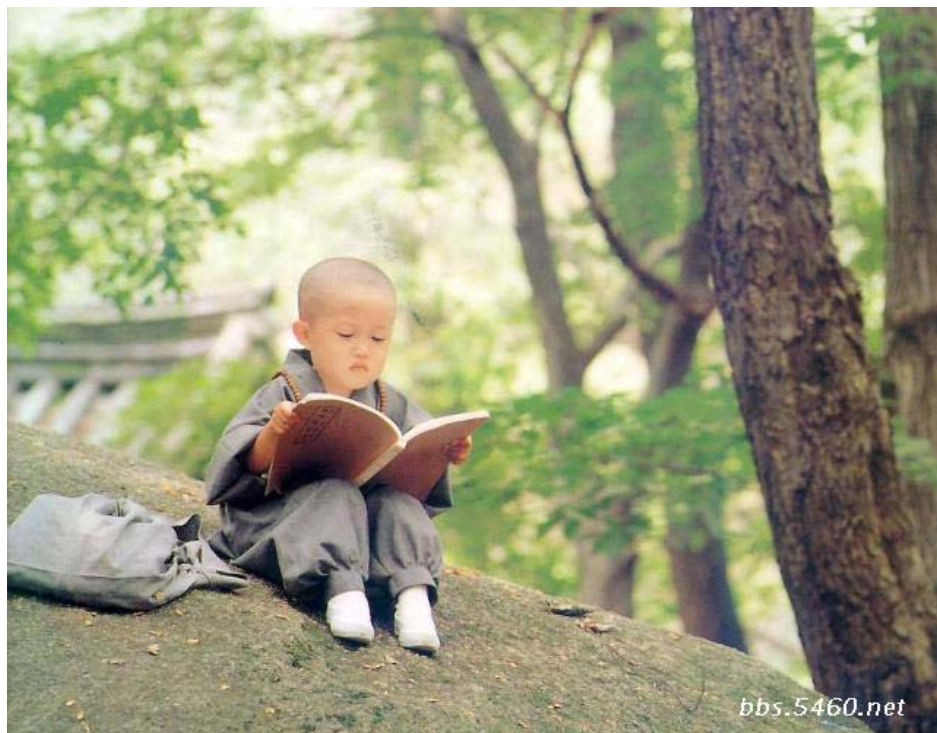


数据结构 1800 题



(答案)

向我开炮出品



2005.9.23

第1章 绪论

一、选择题

1.B	2.C	3.1C	3.2B	4.B	5.D	6.C	7.C	8.D	9.D	10.A	11.C
12.D	13.D	14.A	15.C	16.A	17.C						

二、判断题

1. ×	2. ×	3. ×	4. ×	5. √	6. ×	7. ×	8. √	9. ×	10. ×	11. ×	12. √
13. ×											

三、填空题

- 数据元素 数据元素间关系
- 集合 线性结构 树形结构 图状结构或网状结构。
- 数据的组织形式，即数据元素之间逻辑关系的总体。而逻辑关系是指数据元素之间的关联方式或称“邻接关系”。
- 表示（又称映像）。
- (1) 逻辑特性 (2) 在计算机内部如何表示和实现 (3) 数学特性。
- 算法的时间复杂度和空间复杂度。
- (1) 逻辑结构 (2) 物理结构 (3) 操作（运算） (4) 算法。
- (1) 有穷性 (2) 确定性 (3) 可行性。
- (1) $n+1$ (2) n (3) $n(n+3)/2$ (4) $n(n+1)/2$ 。
- $1+(1+2+\dots+(1+2+3)+\dots+(1+2+\dots+n))=n(n+1)(n+2)/6$ $O(n^3)$
- $\log_2 n$
- $n \log_2 n$
- $\log_2 n^2$
- $(n+3)(n-2)/2$
- $O(n)$
- ① (1) 1 (2) 1 (3) $f(m, n-1)$ (4) n ② 9
- $n(n-1)/2$

四、应用题

- 数据结构是一门研究在非数值计算的程序设计问题中，计算机的操作对象及对象间的关系和施加于对象的操作等的学科。
- 四种表示方法
 - 顺序存储方式。数据元素顺序存放，每个存储结点只含一个元素。存储位置反映数据元素间的逻辑关系。存储密度大，但有些操作（如插入、删除）效率较差。
 - 链式存储方式。每个存储结点除包含数据元素信息外还包含一组（至少一个）指针。指针反映数据元素间的逻辑关系。这种方式不要求存储空间连续，便于动态操作（如插入、删除等），但存储空间开销大（用于指针），另外不能折半查找等。
 - 索引存储方式。除数据元素存储在一地址连续的内存空间外，尚需建立一个索引表，索引表中索引指示存储结点的存储位置（下标）或存储区间端点（下标），兼有静态和动态特性。
 - 散列存储方式。通过散列函数和解决冲突的方法，将关键字散列在连续的有限的地址空间内，并将散列函数的值解释成关键字所在元素的存储地址，这种存储方式称为散列存储。其特点是存取速度快，只能按关键字随机存取，不能顺序存取，也不能折半存取。
- 数据类型是程序设计语言中的一个概念，它是一个值的集合和操作的集合。如C语言中的整型、实型、字符型等。整型值的范围（对具体机器都应有整数范围），其操作有加、减、乘、除、求余等。实际上数据类型是厂家提供给用户的已实现了的数据结构。“抽象数据类型（ADT）”指一个数学模型及定义在该模型上的一组操作。“抽象”的意义在于数据类型的数学抽象特性。抽象数据类型的定义仅取决于它的逻辑特性，而与其在计算机内部如何表示和实现无关。无论其内部结构如何变化，只要它的数学特性不变就不影响它的外部使用。抽象数据类型和数据类型实质上是一个概念。此外，抽象数据类型的范围更广，它已不再局限于机器已定义和实现的数据类型，还包括用户在设计软件系统时自行定义的数据类型。使用抽象数据类型定义的软件模块含定义、表示和实现三部分，封装在一起，对用户透明（提供接口），而不必了解实现细节。抽象数据类型的出现使程序设计不再是“艺术”，而是向“科学”迈进了一步。
- (1) 数据的逻辑结构反映数据元素之间的逻辑关系（即数据元素之间的关联方式或“邻接关系”），数据的存储结构是数据结构在计算机中的表示，包括数据元素的表示及其关系的表示。数据的运算是对其数据定义的一组操作，运算是定义在逻辑结构上的，和存储结构无关，而运算的实现则是依赖于存储结构。
- 逻辑结构相同但存储不同，可以是不同的数据结构。例如，线性表的逻辑结构属于线性结构，采用顺序存储结构为顺序表，而采用链式存储结构称为线性链表。
- 栈和队列的逻辑结构相同，其存储表示也可相同（顺序存储和链式存储），但由于其运算集合不同而成为不同的数据结构。
- 数据结构的评价非常复杂，可以考虑两个方面，一是所选数据结构是否准确、完整的刻划了问题的基本特征；二是是否容易实现（如对数据分解是否恰当；逻辑结构的选择是否适合于运算的功能，是否有利于运算的实现；基本运算的选择是否恰当。）
- 评价好的算法有四个方面。一是算法的正确性；二是算法的易读性；三是算法的健壮性；四是算法的时空效率（运行）。
- (1) 见上面题3 (2) 见上面题4 (3) 见上面题3
- 算法的时间复杂性是算法输入规模的函数。算法的输入规模或问题的规模是作为该算法输入的数据所含数据元素的数目，或与此数目有关的其它参数。有时考虑算法在最坏情况下的时间复杂度或平均时间复杂度。
- 算法是对特定问题求解步骤的描述，是指令的有限序列，其中每一条指令表示一个或多个操作。算法具有五个重要特性：有穷性、确定性、可行性、输入和输出。
- 频度。在分析算法时间复杂度时，有时需要估算基本操作的原操作，它是执行次数最多的一个操作，该

操作重复执行的次数称为频度。

7. 集合、线性结构、树形结构、图形或网状结构。 8. 逻辑结构、存储结构、操作（运算）。

9. 通常考虑算法所需要的存储空间量和算法所需要的时间量。后者又涉及到四方面：程序运行时所需输入的数据总量，对源程序进行编译所需时间，计算机执行每条指令所需时间和程序中指令重复执行的次数。

10. D是数据元素的有限集合，S是D上数据元素之间关系的有限集合。

11. “数据结构”这一术语有两种含义，一是作为一门课程的名称；二是作为一个科学的概念。作为科学概念，目前尚无公认定义，一般认为，讨论数据结构要包括三个方面，一是数据的逻辑结构，二是数据的存储结构，三是对数据进行的操作（运算）。而数据类型是值的集合和操作的集合，可以看作是已实现了的数据结构，后者是前者的一种简化情况。

12. 见上面题2。

13. 将学号、姓名、平均成绩看成一个记录（元素，含三个数据项），将100个这样的记录存于数组中。因一般无增删操作，故宜采用顺序存储。

```
typedef struct
{
    int num;//学号
    char name[8];//姓名
    float score;//平均成绩
}node;
node student[100];
```

14. 见上面题4（3）。

15. 应从两方面进行讨论：如通讯录较少变动（如城市私人电话号码），主要用于查询，以顺序存储较方便，既能顺序查找也可随机查找；若通讯录经常有增删操作，用链式存储结构较为合适，将每个人的情况作为一个元素（即一个结点存放一个人），设姓名作关键字，链表安排成有序表，这样可提高查询速度。

16. 线性表中的插入、删除操作，在顺序存储方式下平均移动近一半的元素，时间复杂度为 $O(n)$ ；而在链式存储方式下，插入和删除时间复杂度都是 $O(1)$ 。

17. 对算法A1和A2的时间复杂度T1和T2取对数，得 $n\log^2$ 和 $2\log^n$ 。显然，算法A2好于A1。

18. **struct** node

```
{int year,month,day;};
```

```
typedef struct
```

```
{int num;//帐号
```

```
char name[8];//姓名
```

```
struct node date;//开户年月日
```

```
int tag;//储蓄类型，如：0- 零存，1- 一年定期……
```

```
float put;//存入累加数；
```

```
float interest;//利息
```

```
float total;//帐面总数
```

```
}count;
```

19. (1) n (2) $n+1$ (3) n (4) $(n+4)(n-1)/2$ (5) $(n+2)(n-1)/2$ (6) $n-1$

这是一个递归调用，因k的初值为1，由语句（6）知，每次调用k增1，故第(1)语句执行 n 次。（2）是FOR循环语句，在满足(1)的条件下执行，该语句进入循环体(3) n 次，加上最后一次判断出界，故执行了 $n+1$ 次。（4）也是循环语句，当 $k=1$ 时判断 $n+1$ 次（进入循环体(5) n 次）， $k=2$ 时判断 n 次，最后一次 $k=n-1$ 时判断3次，故执行次数是 $(n+1)+n+\dots+3=(n+4)(n-1)/2$ 次。语句(5)是(4)的循环体，每次比(4)少一次判断，故执行次数是 $n+(n-1)+\dots+2=(n+2)(n-1)/2$ 次。注意分析时，不要把(2)分析成 n 次，更不是1次。

20. 4（这时 $i=4$ ， $s=100$ ） REPEAT语句先执行循环体，后判断条件，直到条件为真时退出循环。

21. 算法在最好情况下，即二进制数的最后一位为零时，只作一次判断，未执行循环体，赋值语句A[i]执行了一次；最坏情况出现在二进制数各位均为1（最高位为零，因题目假设无溢出），这时循环体执行了 $n-1$ 次，时间复杂度是 $O(n)$ ，循环体平均执行 $n/2$ 次，时间复杂度仍是 $O(n)$ 。

22. 该算法功能是将原单循环链表分解成两个单循环链表：其一包括结点h到结点g的前驱结点；另一个包括结点g到结点h的前驱结点。时间复杂度是 $O(n)$ 。

23. 第一层FOR循环判断 $n+1$ 次，往下执行 n 次，第二层FOR执行次数为 $(n+(n-1)+(n-2)+\dots+1)$ ，第三层循环体受第一层循环和第二层循环的控制，其执行次数如下表：

i=	1	2	3	...	n
j=n	n	n	n	...	n
j=n-1	n-1	n-1	n-1	...	
...		
j=3	3	3			
j=2	2	2			
j=1	1				

执行次数为 $(1+2+\dots+n)+(2+3+\dots+n)+\dots+n=n*(n+1)/2-n(n^2-1)/6$ 。在 $n=5$ 时， $f(5)=55$ ，执行过程中，输出结果为：sum=15, sum=29, sum=41, sum=50, sum=55（每个sum= 占一行，为节省篇幅，这里省去换行）。

$$\sum_{i=1}^{n/2} (n-2i+1) = \frac{n^2}{4}$$

24. $O(n^2)$, m 的值等于赋值语句 $m:=m+1$ 的运行次数, 其计算式为

25. (1) $O(1)$ (2) $O(n^2)$ (3) $O(n^3)$

26. (1) $O(n)$ (2) $O(n^2)$

27. (1) 由斐波那契数列的定义可得:

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &= 2F_{n-2} + F_{n-3} \\ &= 3F_{n-3} + 2F_{n-4} \\ &= 5F_{n-4} + 3F_{n-5} \\ &= 8F_{n-5} + 5F_{n-6} \\ &\dots\dots \\ &= pF_1 + qF_0 \end{aligned}$$

设 F_m 的执行次数为 B_m ($m=0, 1, 2, \dots, n-1$), 由以上等式可知, F_{n-1} 被执行一次, 即 $B_{n-1}=1$; F_{n-2} 被执行两次, 即 $B_{n-2}=2$; 直至 F_1 被执行 p 次、 F_0 被执行 q 次, 即 $B_1=p$, $B_0=q$ 。 B_m 的执行次数为前两等式第一因式系数之和, 即 $B_m=B_{m-1}+B_{m-2}$, 再有 $B_{n-1}=1$ 和 $B_{n-2}=2$, 这也是一个斐波那契数列。可以解得:

$$B_m = \frac{\sqrt{5}}{5} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n-m+2} - \left(\frac{1-\sqrt{5}}{2} \right)^{n-m+2} \right] \quad (m=0, 1, 2, \dots, n-1)$$

(2) 时间复杂度为 $O(n)$

28. 从小到大排列为: $\log n$, $n^{1/2} + \log n$, n , $n \log n$, $n^2 + \log n$, n^3 , $n - n^3 + 7n^5$, $2^{n/2}$, $(3/2)^n$, $n!$, $\binom{2n}{n}$

第2章 线性表

一. 选择题

1. A	2. B	3. C	4. A	5. D	6. D	7. D	8. C	9. B	10. B, C	11. 1I	11. 2I	11. 3E
11. 4B	11. 5C	12. B	13. C	14. C	15. C		16. A	17. A	18. A	19. D	20. C	21. B
22. D	23. C	24. B	25. B	26. A	27. D							

二. 判断题

1. ×	2. √	3. √	4. ×	5. ×	6. ×	7. ×	8. ×	9. ×	10. ×	11. ×	12. ×
13. ×	14. √	15. ×	16. √								

部分答案解析如下。

1. 头结点并不“仅起”标识作用，并且使操作统一。另外，头结点数据域可写入链表长度，或作监视哨。
4. 两种存储结构各有优缺点，应根据实际情况选用，不能笼统说哪个好。
7. 集合中元素无逻辑关系。
9. 非空线性表第一个元素无前驱，最后一个元素无后继。
13. 线性表是逻辑结构，可以顺序存储，也可链式存储。

三. 填空题

1. 顺序
2. $(n-1)/2$
3. $py \rightarrow next = px \rightarrow next; px \rightarrow next = py$
4. $n-i+1$
5. 主要是使插入和删除等操作统一，在第一个元素之前插入元素和删除第一个结点不必另作判断。另外，不论链表是否为空，链表指针不变。
6. $O(1)$, $O(n)$
7. 单链表, 多重链表, (动态) 链表, 静态链表
8. $f \rightarrow next = p \rightarrow next; f \rightarrow prior = p; p \rightarrow next \rightarrow prior = f; p \rightarrow next = f;$
9. $p \rightarrow prior$ $s \rightarrow prior \rightarrow next$
10. 指针
11. 物理上相邻
- 指针
12. 4 2
13. 从任一结点出发都可访问到链表中每一个元素。
14. $u = p \rightarrow next; p \rightarrow next = u \rightarrow next; free(u);$
15. $L \rightarrow next \rightarrow next == L$
16. $p \rightarrow next != null$
17. $L \rightarrow next == L \ \&\& \ L \rightarrow prior == L$
18. $s \rightarrow next = p \rightarrow next; p \rightarrow next = s;$
19. (1) **IF** $pa = NIL$ **THEN return**(true);
(2) $pb \neq NIL$ **AND** $pa \rightarrow data = pb \rightarrow data$
(3) **return**(inclusion(pa, pb));
(4) $pb := pb \rightarrow next;$
(5) **return**(false);
非递归算法:
(1) $pre := pb;$ (2) $pa \neq NIL$ **AND** $pb \neq NIL$ **AND** $pb \rightarrow data = pa \rightarrow data$ (3) $pa := pa \rightarrow next; pb := pb \rightarrow next;$
(4) $pb := pre \rightarrow next; pre := pb; pa := pa \rightarrow next;$ (5) **IF** $pa = NIL$ **THEN return**(true) **ELSE return**(false);
[注]: 本题是在链表上求模式匹配问题。非递归算法中用指针pre指向主串中开始结点(初始时为第一元素结点)。若主串与子串对应数据相等, 两串工作指针pa和pb后移; 否则, 主串工作指针从pre的下一结点开始(这时pre又指向新的开始结点), 子串工作指针从子串第一元素开始, 比较一直继续到循环条件失败。若pa为空, 则匹配成功, 返回true, 否则, 返回false。
20. A. **VAR** head:ptr B. new(p) C. $p \rightarrow data := k$ D. $q \rightarrow next := p$ E. $q := p$ (带头结点)
21. (1) new(h); //生成头结点, 以便于操作。
(2) $r \rightarrow next := p;$ (3) $r \rightarrow next := q;$ (4) **IF** ($q = NIL$) **THEN** $r \rightarrow next := p;$
22. A: $r \rightarrow link \rightarrow data \neq \max$ **AND** $q \rightarrow link \rightarrow data \neq \max$
B: $r := r \rightarrow link$ C: $q \rightarrow link$ D: $q \rightarrow link$ E: $r \rightarrow link$ F: $r \rightarrow link$
G: $r := s$ (或 $r := r \rightarrow link$) H: $r := r \rightarrow link$ I: $q \rightarrow link := s \rightarrow link$
23. (1) la (2) 0 (3) $j < i-1$ (4) $p \uparrow \rightarrow next$ (5) $i < 1$
24. (1) $head \rightarrow left := s$ //head的前驱指针指向插入结点
(2) $j := 1;$
(3) $p := p \rightarrow right$ //工作指针后移
(4) $s \rightarrow left := p$
(5) $p \rightarrow right \rightarrow left := s;$ //p后继的前驱是s
(6) $s \rightarrow left := p;$
25. (1) $i \leq L \rightarrow last$ //L.last 为元素个数
(2) $j := j+1$ //有值不相等的元素
(3) $L \rightarrow elem[j] := L \rightarrow elem[i]$ //元素前移
(4) $L \rightarrow last := j$ //元素个数
26. (A) $p \rightarrow link := q;$ //拉上链, 前驱指向后继
(B) $p := q;$ //新的前驱
(C) $p \rightarrow link := head;$ //形成循环链表
(D) $j := 0;$ //计数器, 记被删结点
(E) $q := p \rightarrow link$ //记下被删结点

- (F)p[^].link=q[^].link //删除结点
27. (1)p:=r; //r指向工作指针s的前驱, p指向最小值的前驱。
 (2)q:=s; //q指向最小值结点, s是工作指针
 (3)s:=s[^].link //工作指针后移
 (4)head:=head[^].next; //第一个结点值最小;
 (5)p[^].link=q[^].link; //跨过被删结点(即删除一结点)
28. (1) l[^].key:=x; //头结点l这时起监视哨作用
 (2) l[^].freq:=p[^].freq //头结点起监视哨作用
 (3) q->pre->next=q->next; q->next->pre=q->pre; //先将q结点从链表上摘下
 q[^].next:=p; q[^].pre:=p[^].pre; p[^].pre->next:=q; p[^].pre:=q; //结点q插入结点p前
 (4) q[^].freq=0 //链表中无值为x的结点, 将新建结点插入到链表最后(头结点前)。
29. (1)a[^].key:=' @' //a的头结点用作监视哨, 取不同于a链表中其它数据域的值
 (2)b[^].key:=p[^].key //b的头结点起监视哨作用
 (3)p:=p[^].next //找到a, b表中共同字母, a表指针后移
 (4) 0(m*n)
30. C 部分: (1)p!=null //链表未到尾就一直作
 (2)q //将当前结点作为头结点后的第一元素结点插入
31. (1)L=L->next; //暂存后继
 (2)q=L; //待逆置结点
 (3)L=p; //头指针仍为L
32. (1) p[^].next<>p0 (2)r:= p[^].next (3) p[^].next:= q0;
 (4) q0:= p; (5) p:=r
33. (1)r (2)NIL (3)x<head[^].data (4)p[^].data<x
 (5)p:=p[^].next (6)p[^].data>x; (7)r (8)p
 (9)r (10)NIL (11)NIL
34. (1)pa!=ha //或pa->exp!=-1
 (2)pa->exp==0 //若指数为0, 即本项为常数项
 (3)q->next=pa->next //删常数项
 (4)q->next //取下一元素
 (5)=pa->coef*pa->exp
 (6)— //指数项减1
 (7)pa //前驱后移, 或q->next
 (8)pa->next //取下一元素
35. (1)q:=p; //q是工作指针p的前驱
 (2)p[^].data>m //p是工作指针
 (3)r:=q; //r 记最大值的前驱,
 (4)q:=p; //或q:=q[^].next;
 (5)r[^].next:=q[^].next; //或r[^].next:=r[^].next[^].next 删最大值结点
36. (1)L->next=null //置空链表, 然后将原链表结点逐个插入到有序表中
 (2)p!=null //当链表尚未到尾, p为工作指针
 (3)q!=null //查p结点在链表中的插入位置, 这时q是工作指针。
 (4)p->next=r->next //将p结点链入链表中
 (5)r->next=p //r是q的前驱, u是下个待插入结点的指针。
37. 程序(a) PASCAL部分(编者略)
 程序(b) C部分
 (1) (A!=null && B!=null) //两均未空时循环
 (2) A->element==B->element //两表中相等元素不作结果元素
 (3) B=B->link //向后移动B表指针
 (4) A!=null //将A 表剩余部分放入结果表中
 (5) last->link=null //置链表尾

四、应用题

1. (1) 选链式存储结构。它可动态申请内存空间, 不受表长度(即表中元素个数)的影响, 插入、删除时间复杂度为O(1)。

(2) 选顺序存储结构。顺序表可以随机存取, 时间复杂度为O(1)。

2. 链式存储结构一般说克服了顺序存储结构的三个弱点。首先, 插入、删除不需移动元素, 只修改指针, 时间复杂度为O(1); 其次, 不需要预先分配空间, 可根据需要动态申请空间; 其三, 表容量只受可用内存空间的限制。其缺点是因为指针增加了空间开销, 当空间不允许时, 就不能克服顺序存储的缺点。

3. 采用链式存储结构, 它根据实际需要申请内存空间, 而当不需要时又可将不用结点空间返还给系统。在链式存储结构中插入和删除操作不需要移动元素。

4. 线性表 栈 队列 串 顺序存储结构和链式存储结构。

顺序存储结构的定义是：

CONST maxlen=线性表可能达到的最大长度；

TYPE sqliстtp=RECORD

elem:ARRAY[1..maxlen] OF ElemType;

last:0..maxlen;

END;

链式存储结构的定义是：

TYPE pointer=↑nodetype;

nodetype=RECORD

data:ElemType;

next:pointer;

END;

linklisttp=pointer;

5. 顺序映射时， a_i 与 a_{i+1} 的物理位置相邻；链表表示时 a_i 与 a_{i+1} 的物理位置不要求相邻。

6. 在线性表的链式存储结构中，头指针指链表的指针，若链表有头结点则是链表的头结点的指针，头指针具有标识作用，故常用头指针冠以链表的名字。头结点是为了操作的统一、方便而设立的，放在第一元素结点之前，其数据域一般无意义（当然有些情况下也可存放链表的长度、**用做监视哨**等等），有头结点后，对在第一元素结点前插入结点和删除第一结点，其操作与对其它结点的操作统一了。而且无论链表是否为空，头指针均不为空。首元结点也就是第一元素结点，它是头结点后边的第一个结点。

7. 见上题6。

8. (1) 将next域变为两个域：pre和next，其值域均为0..maxsize。初始化时，头结点（下标为0的元素）其next域值为1，其pre域值为n（设n是元素个数，且 $n < \text{maxsize}$ ）

(2) stalist[stalist[p].pre].pre;

(3) stalist[p].next;

9. 在单链表中不能从当前结点（若当前结点不是第一结点）出发访问到任何一个结点，链表只能从头指针开始，访问到链表中每个结点。在双链表中求前驱和后继都容易，从当前结点向前到第一结点，向后到最后结点，可以访问到任何一个结点。

10. 本题是链表的逆置问题。设该链表带头结点，将头结点摘下，并将其指针域置空。然后从第一元素结点开始，直到最后一个结点为止，依次前插入头结点的后面，则实现了链表的逆置。

11. 该算法的功能是判断链表L是否是非递减有序，若是则返回“true”；否则返回“false”。pre指向当前结点，p指向pre的后继。

12. $q=p \rightarrow \text{next}$; $p \rightarrow \text{next}=q \rightarrow \text{next}$; free(q);

13. 设单链表的头结点的头指针为head, 且 $\text{pre}=\text{head}$;

while($\text{pre} \rightarrow \text{next} \neq p$) $\text{pre}=\text{pre} \rightarrow \text{next}$;

$s \rightarrow \text{next}=p$; $\text{pre} \rightarrow \text{next}=s$;

14. 设单链表带头结点，工作指针p初始化为 $p=\text{H} \rightarrow \text{next}$;

(1) while($p \neq \text{null}$ && $p \rightarrow \text{data} \neq X$) $p=p \rightarrow \text{next}$;

if($p = \text{null}$) return(null); // 查找失败

else return(p); // 查找成功

(2) while($p \neq \text{null}$ && $p \rightarrow \text{data} < X$) $p=p \rightarrow \text{next}$;

if($p = \text{null}$ || $p \rightarrow \text{data} > X$) return(null); // 查找失败

else return(p);

(3) while($p \neq \text{null}$ && $p \rightarrow \text{data} > X$) $p=p \rightarrow \text{next}$;

if($p = \text{null}$ || $p \rightarrow \text{data} < X$) return(null); // 查找失败

else return(p); // 查找成功

15. 本程序段功能是将pa和pb链表中的值相同的结点保留在pa链表中（pa中与pb中不同结点删除），pa是结果链表的头指针。链表中结点值与从前逆序。S1记结果链表中结点个数（即pa与pb中相等的元素个数）。S2记原pa链表中删除的结点个数。

16. 设 $q:=p \wedge \text{llink}$; 则

$q \wedge \text{rlink}:=p \wedge \text{rlink}$; $p \wedge \text{rlink} \wedge \text{llink}:=q$; $p \wedge \text{llink}:=q \wedge \text{llink}$;

$q \wedge \text{llink} \wedge \text{rlink}:=p$; $p \wedge \text{rlink}:=q$; $q \wedge \text{llink}:=p$

17. (1) 前两个语句改为：

$p \wedge \text{llink} \wedge \text{rlink} <- p \wedge \text{rlink}$;

$p \wedge \text{rlink} \wedge \text{llink} <- p \wedge \text{llink}$;

(2) 后三个语句序列应改为：

$q \wedge \text{rlink} <- p \wedge \text{rlink}$; // 以下三句的顺序不能变

$p \wedge \text{rlink} \wedge \text{llink} <- q$;

$p \wedge \text{rlink} <- q$;

18. mp是一个过程，其内嵌套有过程subp。

subp(s, q)的作用是构造从s到q的循环链表。

subp(pa, pb)调用结果是将pa到pb的前驱构造为循环链表。

subp(pb, pa)调用结果是将pb到pa的前驱（指在L链表中，并非刚构造的pa循环链表中）构造为循环链

表。

总之，两次调用将L循环链表分解为两个。第一个循环链表包含从pa到pb的前驱，L中除刚构造的pa到pb前驱外的结点形成第二个循环链表。

19. 在指针p所指结点前插入结点s的语句如下：

`s->pre=p->pre; s->next=p; p->pre->next=s; p->pre=s;`

20. (A) `f1<>NIL`并且`f2<>NIL`

(B) `f1↑.data < f2↑.data`

(C) `f2↑.data<f1↑.data`

(D) `f3↑.data<f1↑.data`

(E) `f1<- f1↑.link` 或 `f2=f2↑.link`;

21. 1) 本算法功能是将双向循环链表结点的数据域按值从小到大排序，成为非递减（可能包括数据域值相等的结点）有序双向循环链表。

2) (1) `r->prior=q->prior;` //将q结点摘下，以便插入到适当位置。

(2) `p->next->prior=q;` // (2) (3) 将q结点插入

(3) `p->next=q;`

(4) `r=r->next;` 或 `r=q->next;` // 后移指针，再将新结点插入到适当位置。

五、 算法设计题

1. [题目分析]因为两链表已按元素值递增次序排列，将其合并时，均从第一个结点起进行比较，将小的链入链表中，同时后移链表工作指针。该问题要求结果链表按元素值递减次序排列。故在合并的同时，将链表结点逆置。

`LinkedList Union(LinkedList la, lb)`

// la, lb分别是带头结点的两个单链表的头指针，链表中的元素值按递增序排列，本算法将两链表合并成一个按元素值递减次序排列的单链表。

{ `pa=la->next; pb=lb->next;` // pa, pb分别是链表la和lb的工作指针

`la->next=null;` // la作结果链表的头指针，先将结果链表初始化为空。

while(`pa!=null && pb!=null`) // 当两链表均不为空时作

if(`pa->data<=pb->data`)

{ `r=pa->next;` // 将pa 的后继结点暂存于r。

`pa->next=la->next;` // 将pa结点链于结果表中，同时逆置。

`la->next=pa;`

`pa=r;`

// 恢复pa为当前待比较结点。

}

else

{`r=pb->next;` // 将pb 的后继结点暂存于r。

`pb->next=la->next;` // 将pb结点链于结果表中，同时逆置。

`la->next=pb;`

`pb=r;` // 恢复pb为当前待比较结点。

}

while(`pa!=null`) // 将la表的剩余部分链入结果表，并逆置。

{`r=pa->next; pa->next=la->next; la->next=pa; pa=r;` }

while(`pb!=null`)

{`r=pb->next; pb->next=la->next; la->next=pb; pb=r;` }

} // 算法Union结束。

[算法讨论] 上面两链表均不为空的表达式也可简写为**while**(`pa&&pb`)，两递增有序表合并成递减有序表时，上述算法是边合并边逆置。也可先合并完，再作链表逆置。后者不如前者优化。算法中最后两个**while**语句，不可能执行两个，只能二者取一，即哪个表尚未到尾，就将其逆置到结果表中，即将剩余结点依次前插入到结果表的头结点后面。

与本题类似的其它题解答如下：

(1) [问题分析] 与上题类似，不同之处在于：一是链表无头结点，为处理方便，给加上头结点，处理结束再删除之；二是数据相同的结点，不合并到结果链表中；三是hb链表不能被破坏，即将hb的结点合并到结果链表时，要生成新结点。

`LinkedList Union(LinkedList ha, hb)`

// ha和hb是两个无头结点的数据域值递增有序的单链表，本算法将hb中并不出现在ha中的数据合并到ha中，合并中不能破坏hb链表。

{`LinkedList la;`

`la=(LinkedList)malloc(sizeof(LNode));`

`la->next=ha;` // 申请头结点，以便操作。

`pa=ha;` // pa是ha链表的工作指针

`pb=hb;` // pb是hb链表的工作指针

`pre=la;` // pre指向当前待合并结点的前驱。

while(`pa&&pb`)

```

if(pa->data<pb->data) //处理ha中数据
{pre->next=pa;pre=pa;pa=pa->next;}
else if(pa->data>pb->data) //处理hb中数据。
{r=(LinkedList)malloc(sizeof(LNode)); //申请空间
r->data=pb->data; pre->next=r;
pre=r; //将新结点链入结果链表。
pb=pb->next; //hb链表中工作指针后移。
}
else //处理pa->data=pb->data;
{pre->next=pa; pre=pa;
pa=pa->next; //两结点数据相等时, 只将ha的数据链入。
pb=pb->next; //不要hb的相等数据
}
if(pa!=null)pre->next=pa; //将两链表中剩余部分链入结果链表。
else pre->next=pb;
free(la); //释放头结点. ha, hb指针未被破坏。
} //算法nion结束。

```

(2) 本题与上面两题类似, 要求结果指针为lc, 其核心语句段如下:

```

pa=la->next;pb=hb->next;
lc=(LinkedList )malloc(sizeof(LNode));
pc=lc; //pc是结果链表中当前结点的前驱
while(pa&&pb)
if(pa->data<pb->data)
{pc->next=pa;pc=pa;pa=pa->next;}
else {pc->next=pb;pc=pb;pb=pb->next;}
if(pa)pc->next=pa; else pc->next=pb;
free(la);free(lb); //释放原来两链表的头结点。

```

算法时间复杂度为 $O(m+n)$, 其中m和n分别为链表la和lb的长度。

2. [题目分析]本组题有6个, 本质上都是链表的合并操作, 合并中有各种条件。与前组题不同的是, 叙述上是用线性表代表集合, 而操作则是求集合的并、交、差($A \cup B$, $A \cap B$, $A - B$)等。

本题与上面1. (2) 基本相同, 不同之处1. (2) 中链表是“非递减有序”, (可能包含相等元素), 本题是元素“递增有序”(不准有相同元素)。因此两表中合并时, 如有元素值相等元素, 则应删掉一个。

LinkedList Union(LinkedList ha, hb)

// 线性表A和B代表两个集合, 以链式存储结构存储, 元素递增有序。ha和hb分别是其链表的头指针。本算法求A和B的并集 $A \cup B$, 仍用线性表表示, 结果链表元素也是递增有序。

```

{ pa=ha->next;pb=hb->next; //设工作指针pa和pb。
pc=ha; //pc为结果链表当前结点的前驱指针。
while(pa&&pb)
if(pa->data<pb->data)
{pc->next=pa;pc=pa;pa=pa->next;}
else if(pa->data>pb->data)
{pc->next=pb;pc=pb;pb=pb->next;}
else //处理pa->data=pb->data.
{pc->next=pa;pc=pa;pa=pa->next;
u=pb;pb=pb->next;free(u);}
if(pa) pc->next=pa; //若ha表未空, 则链入结果表。
else pc->next=pb; //若hb表未空, 则链入结果表。
free(hb); //释放hb头结点
return(ha);
} //算法Union结束。

```

与本题类似的其它几个题解答如下:

(1) 解答完全同上2。

(2) 本题是求交集, 即只有同时出现在两集合中的元素才出现在结果表中。其核心语句段如下:

```

pa=la->next;pb=lb->next; //设工作指针pa和pb;
pc=la; //结果表中当前合并结点的前驱的指针。
while(pa&&pb)
if(pa->data==pb->data) //交集并入结果表中。
{ pc->next=pa;pc=pa;pa=pa->next;
u=pb;pb=pb->next;free(u);}
else if(pa->data<pb->data) {u=pa;pa=pa->next;free(u);}
else {u=pb;pb=pb->next;free(u);}
while(pa) { u=pa; pa=pa->next; free(u);} //释放结点空间

```

```
while(pb) {u=pb; pb=pb->next; free(u);} //释放结点空间
```

```
pc->next=null; //置链表尾标记。
```

```
free(lb); //注： 本算法中也可对B表不作释放空间的处理
```

(3) 本题基本与(2)相同,但要求无重复元素,故在算法中,待合并结点数据要与其前驱比较,只有在与前驱数据不同时才并入链表。其核心语句段如下。

```
pa=L1->next;pb=L2->next; // pa、pb是两链表的工作指针。
```

```
pc=L1; //L1作结果链表的头指针。
```

```
while(pa&&pb)
```

```
if(pa->data<pb->data) {u=pa;pa=pa->next;free(u);} //删除L1表多余元素
```

```
else if (pa->data>pb->data) pb=pb->next; //pb指针后移
```

```
else // 处理交集元素
```

```
{if(pc==L1) {pc->next=pa;pc=pa;pa=pa->next;} //处理第一个相等的元素。
```

```
else if(pc->data==pa->data) { u=pa;pa=pa->next;free(u);} //重复元素不进入L1表。
```

```
else{ pc->next=pa;pc=pa;pa=pa->next;} //交集元素并入结果表。
```

```
} //while
```

```
while(pa) {u=pa;pa=pa->next;free(u);} // 删L1表剩余元素
```

```
pc->next=null; //置结果链表尾。
```

注： 本算法中对L2表未作释放空间的处理。

(4) 本题与上面(3)算法相同,只是结果表要另辟空间。

(5) [题目分析]本题首先求B和C的交集,即求B和C中共有元素,再与A求并集,同时删除重复元素,以保持结果A递增。

```
LinkedList union(LinkedList A,B,C)
```

//A,B和C均是带头结点的递增有序的单链表,本算法实现 $A = A \cup (B \cap C)$,使求解结构保持递增有序。

```
{pa=A->next;pb=B->next;pc=C->next; //设置三个工作指针。
```

```
pre=A; //pre指向结果链表中当前待合并结点的前驱。
```

```
if(pa->data<pb->data||pa->data<pc->data) //A中第一个元素为结果表的第一元素。
```

```
{pre->next=pa;pre=pa;pa=pa->next;}
```

```
else{while(pb&&pc) //找B表和C表中第一个公共元素。
```

```
if(pb->data<pc->data)pb=pb->next;
```

```
else if(pb->data>pc->data)pc=pc->next;
```

```
else break; //找到B表和C表的共同元素就退出while循环。
```

```
if(pb&&pc) //因共同元素而非B表或C表空而退出上面while循环。
```

```
if(pa->data>pb->data) //A表当前元素值大于B表和C表的公共元素,先将B表元素链入。
```

```
{pre->next=pb;pre=pb;pb=pb->next;pc=pc->next;} // B,C公共元素为结果表第一元素。
```

```
} //结束了结果表中第一元素的确定
```

```
while(pa&&pb&&pc)
```

```
{while(pb&&pc)
```

```
if(pb->data<pc->data) pb=pb->next;
```

```
else if(pb->data>pc->data) pc=pc->next;
```

```
else break; //B表和C表有公共元素。
```

```
if(pb&&pc)
```

```
{while(pa&&pa->data<pb->data) //先将A中小于B,C公共元素部分链入。
```

```
{pre->next=pa;pre=pa;pa=pa->next;}
```

```
if(pre->data!=pb->data){pre->next=pb;pre=pb;pb=pb->next;pc=pc->next;}
```

```
else{pb=pb->next;pc=pc->next;} //若A中已有B,C公共元素,则不再存入结果表。
```

```
}
```

```
} // while(pa&&pb&&pc)
```

```
if(pa) pre->next=pa; //当B,C无公共元素(即一个表已空),将A中剩余链入。
```

```
} //算法Union结束
```

[算法讨论]本算法先找结果链表的第一个元素,这是因为题目要求结果表要递增有序(即删除重复元素)。这就要求当前待合并到结果表的元素要与其前驱比较。由于初始 $pre=A$ (头结点的头指针),这时的data域无意义,不能与后继比较元素大小,因此就需要确定第一个元素。当然,不要这样作,而直接进入下面循环也可以,但在链入结点时,必须先判断 pre 是否等于 A ,这占用了过多的时间。因此先将第一结点链入是可取的。

算法中的第二个问题是要求时间复杂度为 $O(|A|+|B|+|C|)$ 。这就要求各个表的工作指针只能后移(即不能每次都从头指针开始查找)。本算法满足这一要求。

最后一个问题是,当B,C有一表为空(即B和C已无公共元素时),要将A的剩余部分链入结果表。

3. [题目分析]循环单链表L1和L2数据结点个数分别为 m 和 n ,将二者合成一个循环单链表时,需要将一个循环链表的结点(从第一元素结点到最后一个结点)插入到另一循环链表的第一元素结点前即可。题目要求“用最快速度将两表合并”,因此应找结点个数少的链表查其尾结点。

```
LinkedList Union(LinkedList L1,L2;int m,n)
```

//L1和L2分别是两循环单链表的头结点的指针,m和n分别是L1和L2的长度。

```

// 本算法用最快速度将L1和L2合并成一个循环单链表。
{if(m<0||n<0) {printf("表长输入错误\n"); exit(0);}
if(m<n) // 若m<n, 则查L1循环单链表的最后一个结点。
{if(m==0)return(L2); // L1为空表。
else{p=L1;
while(p->next!=L1) p=p->next; // 查最后一个元素结点。
p->next=L2->next; // 将L1循环单链表的元素结点插入到L2的第一元素结点前。
L2->next=L1->next;
free(L1); // 释放无用头结点。
}
} // 处理完m<n情况
else // 下面处理L2长度小于等于L1的情况
{if(n==0)return(L1); // L2为空表。
else{p=L2;
while(p->next!=L2) p=p->next; // 查最后元素结点。
p->next=L1->next; // 将L2的元素结点插入到L1循环单链表的第一元素结点前。
L1->next=L2->next;
free(L2); // 释放无用头结点。
}
} // 算法结束。

```

类似本题叙述的其它题解答如下:

(1) [题目分析] 本题将线性表la和lb连接, 要求时间复杂度为 $O(1)$, 且占用辅助空间尽量小。应该使用只设尾指针的单循环链表。

```

LinkedList Union(LinkedList la, lb)
// la和lb是两个无头结点的循环单链表的尾指针, 本算法将lb接在la后, 成为一个单循环链表。
{ q=la->next; // q指向la的第一个元素结点。
la->next=lb->next; // 将lb的最后元素结点接到lb的第一元素。
lb->next=q; // 将lb指向la的第一元素结点, 实现了lb接在la后。
return(lb); // 返回结果单循环链表的尾指针lb。
} // 算法结束。

```

[算法讨论] 若循环单链表带有头结点, 则相应算法片段如下:

```

q=lb->next; // q指向lb的头结点;
lb->next=la->next; // lb的后继结点为la的头结点。
la->next=q->next; // la的后继结点为lb的第一元素结点。
free(q); // 释放lb的头结点
return(lb); // 返回结果单循环链表的尾指针lb。

```

(2) [题目分析] 本题要求将单向链表ha和单向循环链表hb合并成一个单向链表, 要求算法所需时间与链表长度无关, 只有使用带尾指针的循环单链表, 这样最容易找到链表的首、尾结点, 将该结点序列插入到单向链表第一元素之前即可。

其核心算法片段如下 (设两链表均有头结点)

```

q=hb->next; // 单向循环链表的表头指针
hb->next=ha->next; // 将循环单链表最后元素结点接在ha第一元素前。
ha->next=q->next; // 将指向原单链表第一元素的指针指向循环单链表第一结点
free(q); // 释放循环链表头结点。

```

若两链表均不带头结点, 则算法片段如下:

```

q=hb->next; // q指向hb首元结点。
hb->next=ha; // hb尾结点的后继是ha第一元素结点。
ha=q; // 头指针指向hb的首元结点。

```

4. [题目分析] 顺序存储结构的线性表的插入, 其时间复杂度为 $O(n)$, 平均移动近一半的元素。线性表LA和LB合并时, 若从第一个元素开始, 一定会造成元素后移, 这不符合本题“高效算法”的要求。另外, 题中叙述“线性表空间足够大”也暗示出另外合并方式, 即应从线性表的最后一个元素开始比较, 大者放到最终位置上。设两线性表的长度各为m和n, 则结果表的最后一个元素应在m+n位置上。这样从后向前, 直到第一个元素为止。

```

PROC Union(VAR LA:SeqList;LB:SeqList)

```

// LA和LB是顺序存储的非递减有序线性表, 本算法将LB合并到LA中, 元素仍非递减有序。

```

m:=LA.last;n:=LB.last; // m, n分别为线性表LA和LB的长度。
k:=m+n; // k为结果线性表的工作指针 (下标)。
i:=m;j:=n; // i, j分别为线性表LA和LB的工作指针 (下标)。
WHILE(i>0)AND(j>0)DO
IF LA.elem[i]>LB.elem[j]
THEN[LA.elem[k]:=LA.elem[i];k:=k-1;i:=i-1;]
ELSE[LA.elem[k]:=LB.elem[j];k:=k-1;j:=j-1;]

```

```

    WHILE(j>0) DO [LA.elem[k]:=LB.elem[j];k:=k-1;j:=j-1;]
    LA.last:=m+n;

```

ENDP;

[算法讨论]算法中数据移动是主要操作。在最佳情况下(LB的最小元素大于LA的最大元素), 仅将LB的n个元素移(拷贝)到LA中, 时间复杂度为O(n), 最差情况, LA的所有元素都要移动, 时间复杂度为O(m+n)。因数据合并到LA中, 所以在退出第一个WHILE循环后, 只需要一个WHILE循环, 处理LB中剩余元素。第二个循环只有在LB有剩余元素时才执行, 而在LA有剩余元素时不执行。本算法利用了题目中“线性表空间足够大”的条件, “最大限度的避免移动元素”, 是“一种高效算法”。

5. [题目分析]本题实质上是一个排序问题, 要求“不得使用除该链表结点以外的任何链结点空间”。链表上的排序采用直接插入排序比较方便, 即首先假定第一个结点有序, 然后, 从第二个结点开始, 依次插入到前面有序链表中, 最终达到整个链表有序。

```

LinkedList LinkListSort(LinkedList list)

```

//list是不带头结点的线性链表, 链表结点构造为data和link两个域, data是数据域, link是指针域。本算法将该链表按结点数据域的值的大小, 从小到大重新链接。

```

    {p=list->link;    //p是工作指针, 指向待排序的当前元素。
    list->link=null; //假定第一个元素有序, 即链表中现只有一个结点。
    while(p!=null)
    {r=p->link;    //r是p的后继。
    q=list;
    if(q->data>p->data) //处理待排序结点p比第一个元素结点小的情况。
    {p->link=list;
    list=p; //链表指针指向最小元素。
    }
    else //查找元素值最小的结点。
    {while(q->link!=null&& q->link->data<p->data) q=q->link;
    p->link=q->link; //将当前排序结点链入有序链表中。
    q->link=p;
    }
    p=r; //p指向下个待排序结点。
    }
}

```

[算法讨论]算法时间复杂度的分析与用顺序存储结构时的情况相同。但顺序存储结构将第i(i>1)个元素插入到前面第1至第i-1个元素的有序表时, 是将第i个元素先与第i-1个元素比较。而在链表最佳情况均是和第一个元素比较。两种存储结构下最佳和最差情况的比较次数相同, 在链表情况下, 不移动元素, 而是修改结点指针。

另一说明是, 本题中线性链表list不带头结点, 而且要求“不得使用除该链表以外的任何链结点空间”, 所以处理复杂, 需要考虑当前结点元素值比有序链表第一结点的元素值还小的情况, 这时要修改链表指针list。如果list是头结点的指针, 则相应处理要简单些, 其算法片段如下:

```

p=list->link; //p指向第一元素结点。
list->link=null; //有序链表初始化为空
while(p!=null)
{r=p->link; //保存后继
q=list;
while(q->link!=null && q->link->data<p->data) q=q->link;
p->link=q->link;
q->link=p;
q=r;
}

```

6. [题目分析]本题明确指出单链表带头结点, 其结点数据是正整数且不相同, 要求利用直接插入原则把链表整理成递增有序链表。这就要求从第二结点开释, 将各结点依次插入到有序链表中。

```

LinkedList LinkListInsertSort(LinkedList la)

```

//la是带头结点的单链表, 其数据域是正整数。本算法利用直接插入原则将链表整理成递增的有序链表。

{if(la->next!=null) //链表不为空表。

{p=la->next->next; //p指向第一结点的后继。

la->next->next=null; //直接插入原则认为第一元素有序, 然后从第二元素起依次插入。

while(p!=null)

{r=p->next; //暂存p的后继。

q=la;

while(q->next!=null&& q->next->data<p->data) q=q->next; //查找插入位置。

p->next=q->next; //将p结点链入链表。

q->next=p;

p=r;

}

与本题有类似叙述的题的解答：

(1) 本题也是链表排序问题，虽没象上题那样明确要求“利用直接插入的原则”来排序，仍可用上述算法求解，这里不再赘述。

7. [题目分析] 本题要求将一个链表分解成两个链表，两个链表都要有序，两链表建立过程中不得使用NEW过程申请空间，这就是要利用原链表空间，随着原链表的分解，新建链表随之排序。

```
PROC discreat (VAR listhead, P, Q: linkedlist)
```

// listhead 是单链表的头指针，链表中每个结点由一个整数域 DATA 和指针域 NEXT 组成。本算法将链表 listhead 分解成奇数链表和偶数链表，分解由 P 和 Q 指向，且 P 和 Q 链表是有序的。

P:=NIL; Q:=NIL; // P 和 Q 链表初始化为空表。

s:=listhead;

WHILE (s<>NIL) DO

[r:=s^.NEXT; // 暂存 s 的后继。

IF s^.DATA DIV 2=0 // 处理偶数。

THEN IF P=NIL THEN [P:=s; P^.NEXT:=NIL;] // 第一个偶数链结点。

ELSE [pre:=P;

IF pre^.DATA>s^.DATA THEN [s^.NEXT:=pre; P:=s; // 插入当前最小值结点修改头指针]

ELSE [WHILE pre^.NEXT<>NIL DO

IF pre^.NEXT^.DATA<s^.DATA THEN pre:=pre^.NEXT; // 查找插入位置。

s^.NEXT:=pre^.NEXT; // 链入此结点。

pre^.NEXT:=s;

]

]

ELSE // 处理奇数链。

IF Q=NIL THEN [Q:=s; Q^.NEXT:=NIL;] // 第一奇数链结点。

ELSE [pre:=Q;

IF pre^.DATA>s^.DATA THEN [s^.NEXT:=pre; Q:=s;] // 修改头指针。

ELSE [WHILE pre^.NEXT<>NIL DO // 查找插入位置。

IF pre^.NEXT^.DATA<s^.DATA THEN pre:=pre^.NEXT;

s^.NEXT:=pre^.NEXT; // 链入此结点。

pre^.NEXT:=s;

]

] // 结束奇数链结点

s:=r; // s 指向新的待排序结点。

] // 结束 “WHILE (s<>NIL) DO”

ENDP; // 结束整个算法。

[算法讨论] 由于算法要求“不得使用NEW过程申请空间，也没明确指出链表具有头结点，所以上述算法复杂些，它可能需要在第一个结点前插入新结点，即链表的头指针会发生变化。如有头结点，算法不必单独处理在第一个结点前插入结点情况，算法会规范统一，下面的(1)是处理带头结点的例子。算法中偶数链上结点是靠数据整除2等于0 (DATA DIV 2=0) 判断的。

类似本题的其它题解答如下：

(1) [题目分析] 本题基本类似于上面第7题，不同之处有二。一是带头结点，二是分解后的两个链表，一个是数据值小于0，另一个是数据值大于0。由于没明确要求用类PASCAL书写算法，故用C书写如下。

```
void DisCreat1 (LinkedList A)
```

// A 是带头结点的单链表，链表中结点的数据类型为整型。本算法将 A 分解成两个单链表 B 和 C，B 中结点的数据小于零，C 中结点的数据大于零。

```
{B=A;
```

```
C=(LinkedList )malloc(sizeof(LNode)); // 为C申请结点空间。
```

```
C->next=null // C初始化为空表。
```

```
p=A->next; // p为工作指针。
```

```
B->next=null; // B表初始化。
```

```
while(p!=null)
```

```
{r=p->next; // 暂存p的后继。
```

```
if (p->data<0) // 小于0的放入B表。
```

```
{p->next=B->next; B->next=p; } // 将小于0的结点链入B表。
```

```
else {p->next=C->next; C->next=p; }
```

```
p=r; // p指向新的待处理结点。
```

```
}
```

```
} // 算法结束。
```

[算法讨论] 因为本题并未要求链表中结点的数据值有序，所以算法中采取最简单方式：将新结点前插到头结点后面（即第一元素之前）。

(2) 本题同上面第7题，除个别叙述不同外，本质上完全相同，故不再另作解答。

(3) [题目分析] 本题中的链表有头结点，分解成表 A 和表 B，均带头结点。分解后的 A 表含有原表中序号为奇数

的元素, B表含有原A表中序号为偶数的元素。由于要求分解后两表中元素结点的相对顺序不变, 故采用在链表尾插入比较方便, 这使用一指向表尾的指针即可方便实现。

```
void DisCreat3(LinkedList A)
//A是带头结点的单链表, 本算法将其分解成两个带头结点的单链表, A表中含原表中序号为奇数
//的结点, B表中含原表中序号为偶数的结点。链表中结点的相对顺序同原链表。
{i=0; // i记链表中结点的序号。
 B=(LinkedList)malloc(sizeof(LNode)); // 创建B表表头。
 B->next=null; // B表的初始化。
 LinkedList ra, rb; // ra和rb将分别指向将创建的A表和B表的尾结点。
 ra=A; rb=B;
 p=A->next; // p为链表工作指针, 指向待分解的结点。
 A->next=null; // 置空新的A表
 while(p!=null)
 {r=p->next; // 暂存p的后继。
  i++;
  if(i%2==0) // 处理原序号为偶数的链表结点。
  {p->next=rb->next; // 在B表尾插入新结点;
   rb->next=p; rb=p; // rb指向新的尾结点;
  }
  else // 处理原序号为奇数的结点。
  {p->next=ra->next; ra->next=p; ra=p; }
  p=r; // 将p恢复为指向新的待处理结点。
 } // 算法结束
```

8. [题目分析] 题目要求重排 n 个元素且以顺序存储结构存储的线性表, 使得所有值为负数的元素移到正数元素的前面。这可采用快速排序的思想来实现, 只是提出暂存的第一个元素(枢轴)并不作为以后的比较标准, 比较的标准是元素是否为负数。

```
int Rearrange (SeqList a; int n)
//a是具有n个元素的线性表, 以顺序存储结构存储, 线性表的元素是整数。本算法重排线性表a,
//使所有值为负数的元素移到所有值为正数的数的前面。
{i=0; j=n-1; // i, j为工作指针(下标), 初始指向线性表a的第1个和第n个元素。
 t=a[0]; // 暂存枢轴元素。
 while(i<j)
 {while(i<j && a[j]>=0) j--; // 若当前元素为大于等于零, 则指针前移。
  if(i<j) {a[i]=a[j]; i++;} // 将负数前移。
  while(i<j && a[i]<0) i++; // 当前元素为负数时指针后移。
  if(i<j) a[j--]=a[i]; // 正数后移。
 }
 a[i]=t; // 将原第一元素放到最终位置。
 }
```

[算法讨论] 本算法时间复杂度为 $O(n)$ 。算法只是按题目要求把正负数分开, 如要求统计负数和大于等于零的个数, 则最后以 t 来定。如 t 为负数, 则 0 至 i 共 $i+1$ 个负数, $n-1-i$ 个正数(包括零)。另外, 题目并未提及零的问题, 笔者将零放到正数一边。对此问题的扩充是若元素包含正数、负数和零, 并要求按负数、零、正数的顺序重排线性表, 统计负数、零、正数的个数。请读者利用上面解题思想自行解答。

类似本题的选了5个题, 其解答如下:

(1) 与上面第8题不同的是, 这里要求以 a_n 为参考元素, 将线性表分成左右两部分。左半部分的元素都小于等于 a_n , 右半部分的元素都大于 a_n , a_n 位于分界位置上。其算法主要片段语句如下:

```
i=1; j=n;
t=a[n]; // 暂存参考元素。
while(i<j)
{while(i<j && a[i]<=t) i++; // 当前元素不大于参考元素时, 指针i后移。
 if(i<j) a[j--]=a[i]; // 将大于参考元素的元素后移。
 while(i<j && a[j]>t) j--; // 当前元素大于参考元素时指针前移。
 if(i<j) a[i++]=a[j]; // 将小于参考元素的当前元素前移。
 }
 a[i]=t; // 参考元素置于分界位置。
```

(2) [题目分析] 本题要求将线性表A分成B和C两个表, 表B和表C不另占空间, 而是利用表A的空间, 其算法与第8题相同。这里仅把表B和表C另设空间的算法解答如下:

```
void Rearrange2(int A[], B[], C[])
//线性表A有n个整型元素, 顺序存储。本算法将A拆成B和C 两个表, B中存放大于
//等于零的元素, C中存放小于零的元素。
{i=0; // i, j, k是工作指针, 分别指向A、B和C表的当前元素。
 j=k=-1; // j, k初始化为-1。
```

```

while(i<n)
    {if(A[i]<0) C[++k]=A[i++]; //将小于零的元素放入C表。
      else B[++j]=A[i++];      //将大于零的元素放入B表。
    }

```

[算法讨论] 本题用一维数组存储线性表，结果线性表B和C中分别有j+1和k+1个元素。若采用教材中的线性表，则元素的表示作相应改变，例如A.elem[i]，而最后B和C表应置上表的长度，如B.length=j和C.length=k。

(3) 本题与第8题本质上相同，第8题要求分开正数和负数，这里要求分开奇数和偶数，判别方式是a[i]%2==0，满足时为偶数，反之为奇数。

(4) 本题与第8题相同，只是叙述不同。

(5) 本题与第8题基本相同，不同之处在于这里的分界元素是整数19（链表中并不要求一定有19）。本题要求用标准pascal描述算法，如下所示。

```

TYPE arr=ARRAY[1..1000] OF integer;
VAR a: arr;
PROCEDURE Rearrange5 (VAR a: arr);
//a是n（设n=1000）个整数组成的线性表，用一维数组存储。本算法将n个元素中所有大于等于19的整数放在所有小于19的整数之后。
VAR i, j, t: integer;
BEGIN
    i:=1; j:=n; t:=a[1]; //i, j指示顺序表的首尾元素的下标，t暂存分界元素
    WHILE (i<j) DO
        BEGIN
            WHILE (i<j) AND (a[j]>=19) DO j:=j-1;
            IF (i<j) THEN BEGIN A[i]:=A[j]; i:=i+1 END;
            WHILE (i<j) AND (a[i]<19) DO i:=i+1;
            IF (i<j) THEN BEGIN A[j]:=A[i]; j:=j-1 END;
        END;
        a[i]:=t;
    END;

```

[算法讨论] 分界元素t放入a[i]，而不论它的值如何。算法中只用了一个t中间变量，符合空间复杂度O(1)的要求。算法也满足时间复杂度O(n)的要求。

9. [题目分析] 本题要求在单链表中删除最小值结点。单链表中删除结点，为使结点删除后不出现“断链”，应知道被删结点的前驱。而“最小值结点”是在遍历整个链表后才能知道。所以算法应首先遍历链表，求得最小值结点及其前驱。遍历结束后再执行删除操作。

LinkedList Delete (LinkedList L)

//L是带头结点的单链表，本算法删除其最小值结点。

{p=L->next; //p为工作指针。指向待处理的结点。假定链表非空。

pre=L; //pre指向最小值结点的前驱。

q=p; //q指向最小值结点，初始假定第一元素结点是最小值结点。

while (p->next!=null)

{if (p->next->data<q->data) {pre=p; q=p->next; } //查最小值结点

p=p->next; //指针后移。

}

pre->next=q->next; //从链表上删除最小值结点

free (q); //释放最小值结点空间

} //结束算法delete。

[算法讨论] 算法中函数头是按本教材类C描述语言书写的。原题中void delete(linklist &L)，是按C++的“引用”来写的，目的是实现变量的“传址”，克服了C语言函数传递只是“值传递”的缺点。

10. [题目分析] 本题要求将链表中数据域值最小的结点移到链表的最前面。首先要查找最小值结点。将其移到链表最前面，实质上是将该结点从链表上摘下（不是删除并回收空间），再插入到链表的最前面。

LinkedList delinsert (LinkedList list)

//list是非空线性链表，链结点结构是(data, link)，data是数据域，link是链域。

//本算法将链表中数据域值最小的那个结点移到链表的最前面。

{p=list->link; //p是链表的工作指针

pre=list; //pre指向链表中数据域最小值结点的前驱。

q=p; //q指向数据域最小值结点，初始假定是第一结点

while (p->link!=null)

{if (p->link->data<q->data) {pre=p; q=p->link; } //找到新的最小值结点;

p=p->link;

}

if (q!=list->link) //若最小值是第一元素结点，则不需再操作

{pre->link=q->link; //将最小值结点从链表上摘下;

```

    q->link= list->link; // 将q结点插到链表最前面。
    list->link=q;
}
} // 算法结束

```

[算法讨论] 算法中假定list带有头结点，否则，插入操作变为q->link=list; list=q。

11. [题目分析] 知道双向循环链表中的一个结点，与前驱交换涉及到四个结点（p结点，前驱结点，前驱的前驱结点，后继结点）六条链。

```
void Exchange (LinkedList p)
```

//p是双向循环链表中的一个结点，本算法将p所指结点与其前驱结点交换。

```

{q=p->llink;
 q->llink->rlink=p;    //p的前驱的前驱之后继为p
 p->llink=q->llink;    //p的前驱指向其前驱的前驱。
 q->rlink=p->rlink;    //p的前驱的后继为p的后继。
 q->llink=p;          //p与其前驱交换
 p->rlink->llink=q;    //p的后继的前驱指向原p的前驱
 p->rlink=q;          //p的后继指向其原来的前驱
} // 算法exchange结束。

```

12. [题目分析] 顺序存储的线性表递增有序，可以顺序查找，也可折半查找。题目要求“用最少的时间在表中查找数值为x的元素”，这里应使用折半查找方法。

```
void SearchExchangeInsert (ElemType a[]; ElemType x)
```

//a是具有n个元素的递增有序线性表，顺序存储。本算法在表中查找数值为x的元素，如查到则与其后继交换位置；如查不到，则插入表中，且使表仍递增有序。

```

{ low=0; high=n-1;                //low和high指向线性表下界和上界的下标
 while (low<=high)
 {mid= (low+high) /2;             //找中间位置
  if (a[mid]==x) break;           //找到x，退出while循环。
  else if (a[mid]<x) low=mid+1;    //到中点mid的右半去查。
  else high=mid-1;               //到中点mid的左部去查。
 }
 if (a[mid]==x && mid!=n) // 若最后一个元素与x相等，则不存在与其后继交换的操作。
 {t=a[mid]; a[mid]=a[mid+1]; a[mid+1]=t; } // 数值x与其后继元素位置交换。
 if (low>high) // 查找失败，插入数据元素x
 {for (i=n-1; i>high; i--) a[i+1]=a[i]; // 后移元素。
  a[i+1]=x; // 插入x。
 } // 结束插入
} // 结束本算法。

```

[算法讨论] 首先是线性表的描述。算法中使用一维数组a表示线性表，未使用包含数据元素的一维数组和指示线性表长度的结构体。若使用结构体，对元素的引用应使用a.elem[i]。另外元素类型就假定是ElemType，未指明具体类型。其次，C中一维数组下标从0开始，若说有n个元素的一维数组，其最后一个元素的下标应是n-1。第三，本算法可以写成三个函数，查找函数，交换后继函数与插入函数。写成三个函数显得逻辑清晰，易读。

13. [题目分析] 判断链表中数据是否中心对称，通常使用栈。将链表的前一半元素依次进栈。在处理链表的后一半元素时，当访问到链表的一个元素后，就从栈中弹出一个元素，两元素比较，若相等，则将链表中下一元素与栈中再弹出元素比较，直至链表到尾。这时若栈是空栈，则得出链表中心对称的结论；否则，当链表中一元素与栈中弹出元素不等时，结论为链表非中心对称，结束算法的执行。

```
int dc (LinkedList h, int n)
```

// h是带头结点的n个元素单链表，链表中结点的数据域是字符。本算法判断链表是否是中心对称。

```

{char s[]; int i=1; // i记结点个数， s字符栈
 p=h->next; // p是链表的工作指针，指向待处理的当前元素。
 for (i=1; i<=n/2; i++) // 链表前半一半元素进栈。
 {s[i]=p->data; p=p->next; }
 i--; // 恢复最后的i值
 if (n%2==1) p=p->next; } // 若n是奇数，后移过中心结点。
 while (p!=null && s[i]==p->data) {i--; p=p->next; } // 测试是否中心对称。
 if (p==null) return (1); // 链表中心对称
 else return (0); // 链表不中心对称
} // 算法结束。

```

[算法讨论] 算法中先将“链表的前一半”元素（字符）进栈。当n为偶数时，前半一半和后一半的个数相同；当n为奇数时，链表中心结点字符不必比较，移动链表指针到下一字符开始比较。比较过程中遇到不相等时，立即退出while循环，不再进行比较。

14. [题目分析] 在单链表中删除自第i个元素起的共len个元素，应从第1个元素起开始计数，记到第i个时

开始数len个,然后将第i-1个元素的后继指针指向第i+len个结点,实现了在A链表中删除自第i个起的len个结点。这时应继续查到A的尾结点,得到删除元素后的A链表。再查B链表的第j个元素,将A链表插入之。插入和删除中应注意前驱后继关系,不能使链表“断链”。另外,算法中应判断i, len和j的合法性。

```
LinkedList DelInsert (LinkedList heada, headb, int i, j, len)
```

//heada和headb均是带头结点的单链表。本算法删除heada链表中自第i个元素起的共len个元素,然后将单链表heada插入到headb的第j个元素之前。

```
{if (i<1 || len<1 || j<1) {printf ( “参数错误\n” ); exit (0); } // 参数错, 退出算法。
p=heada; //p为链表A的工作指针, 初始化为A的头指针, 查到第i个元素时, p指向第i-1个元素
k=0; // 计数
while (p!=null && k<i-1) // 查找第i个结点。
    {k++; p=p->next; }
if (p==null) {printf ( “给的%d太大\n”, i); exit (0); } // i太大, 退出算法
q=p->next; // q为工作指针, 初始指向A链表第一个被删结点。
k=0;
while (q!=null && k<len) {k++; u=q, q=q->next; free (u); } // 删除结点, 后移指针。
if (k<len) {printf ( “给的%d太大\n”, len); exit (0); }
p->next=q; // A链表删除了len个元素。
if (heada->next!=null) // heada->next=null说明链表中结点均已删除, 无需往B表插入
    {while (p->next!=null) p= p->next; // 找A的尾结点。
    q=headb; // q为链表B的工作指针。
    k=0; // 计数
    while (q!=null && k<j-1) // 查找第j个结点。
        {k++; q= q->next; } // 查找成功时, q指向第j-1个结点
    if (q==null) {printf ( “给的%d太大\n”, j); exit (0); }
    p->next=q->next; // 将A链表链入
    q->next=heada->next; // A的第一元素结点链在B的第j-1个结点之后
    } // if
free (heada); // 释放A表头结点。
} // 算法结束。
```

与本题类似的题的解答如下:

(1) 本题与第14题基本相同,不同之处仅在于插入B链表第j个元素之前的,不是删除了len个元素的A链表,而是被删除的len个元素。按照上题,这len个元素结点中第一个结点的指针p->next,查找从第i个结点开始的第len个结点的算法修改为:

```
k=1; q=p->next; // q指向第一个被删除结点
while (q!=null && k<len) // 查找成功时, q指向自i起的第len个结点。
    {k++; q= q->next; }
if (k<len) {printf ( “给的%d太大\n”, len); exit (0); }
```

15. [题目分析] 在递增有序的顺序表中插入一个元素x,首先应查找待插入元素的位置。因顺序表元素递增有序,采用折半查找法比顺序查找效率要高。查到插入位置后,从此位置直到线性表尾依次向后移动一个元素位置,之后将元素x插入即可。

```
void Insert (ElemType A[], int size, ElemType x)
```

// A是有size个元素空间目前仅有num (num<size)个元素的线性表。本算法将元素x插入到线性表中,并保持线性表的有序性。

```
{low=1; high=num; // 题目要求下标从1开始
while (low<=high) // 对分查找元素x的插入位置。
    {mid= (low+high) /2;
    if (A[mid]==x) {low=mid+1; break; }
    else if (A[mid]>x) high=mid-1; else low=mid+1;
    }
for (i=num; i>=low; i--) A[i+1]=A[i]; // 元素后移。
A[i+1]=x; // 将元素x插入。
} // 算法结束。
```

[算法讨论] 算法中当查找失败(即线性表中无元素x)时,变量low在变量high的右面(low=high+1)。移动元素从low开始,直到num为止。特别注意不能写成for (i=low; i<=num; i++) A[i+1]=A[i],这是一些学生容易犯的错误。另外,题中未说明若表中已有值为x的元素时不再插入,故安排在A[mid]=x时,用low (=mid+1)记住位置,以便后面统一处理。查找算法时间复杂度为O(logn),而插入时的移动操作时间复杂度为O

(n),若用顺序查找,则查找的时间复杂度亦为O(n)。

类似本题的其它题的解答:

(1) [题目分析] 本题与上面15题类似,不同之处是给出具体元素值,且让编写turbo pascal程序,程序如下:

```
PROGRAM example (input, output);
```

```

TYPE  pointer=^node;
      node=RECORD
          data: integer;
          next: pointer;
      END;
VAR  head, q: pointer;
PROCEDURE  create (VAR la: pointer);
    VAR  x: integer;
          p,q: pointer;
    BEGIN
        new (la); la^.next:=NIL; {建立头结点。}
        read (x); q:=la; {q用以指向表尾。}
        WHILE NOT EOF DO {建立链表}
            BEGIN
                new (p); p^.data:=x; p^.next:=q^.next; q^.next:=p; q:=p; read(x);
            END;
        END;
PROCEDURE  insert (VAR la: pointer; s: pointer);
VAR  p,q: pointer; found: boolean;
BEGIN
    p:= la^.next; {p为工作指针。}
    q:=la; {q为p的前驱指针。}
    found:=false;
    WHILE (p<>NIL) AND NOT found
        IF (p^.data<x) THEN BEGIN  q:=p; p:= p^.next;  END
        ELSE found:=true;
    s^.next:=p; {将s结点插入链表}
    q^.next:=s;
END;
BEGIN {main}
    writeln ( “请按顺序输入数据, 建立链表” )
    create (head);
    writeln ( “请输入插入数据” )
    new (q);
    readln (q^.data);
    insert (head, q);
END. {程序结束}

```

[程序讨论] 在建立链表时, 输入数据依次为12, 13, 21, 24, 28, 30, 42, 键入CTRL-Z, 输入结束。“插入数据”输26即可。本题编写的是完整的pascal程序。

16. [题目分析] 将具有两个链域的单循环链表, 改造成双向循环链表, 关键是控制给每个结点均置上指向前驱的指针, 而且每个结点的前驱指针置且仅置一次。

```
void  StoDouble (LinkedList la)
```

//la是结点含有pre, data, link三个域的单循环链表。其中data为数据域, pre为空指针域, link是指向后继的指针域。本算法将其改造成双向循环链表。

```

{while (la->link->pre==null)
    {la->link->pre=la;    //将结点la后继的pre指针指向la。
    la=la->link;          //la指针后移。
    }
} //算法结束。

```

[算法讨论] 算法中没有设置变量记住单循环链表的起始结点, 至少省去了一个指针变量。当算法结束时, la恢复到指向刚开始操作的结点, 这是本算法的优点所在。

17. [题目分析] 求两个集合A和B的差集A-B, 即在A中删除A和B中共有的元素。由于集合用单链表存储, 问题变成删除链表中的结点问题。因此, 要记住被删除结点的前驱, 以便顺利删除被删结点。两链表均从第一元素结点开始, 直到其中一个链表到尾为止。

```
void  Difference (LinkedList A, B, *n)
```

//A和B均是带头结点的递增有序的单链表, 分别存储了一个集合, 本算法求两集合的差集, 存储于单链表A中, *n是结果集中元素个数, 调用时为0

```

{p=A->next;          //p和q分别是链表A和B的工作指针。
q=B->next;  pre=A;    //pre为A中p所指结点的前驱结点的指针。
while (p!=null && q!=null)
    if (p->data<q->data) {pre=p; p=p->next; *n++; } // A链表中当前结点指针后移。

```

```

else if (p->data>q->data) q=q->next; //B链表中当前结点指针后移。
else {pre->next=p->next; //处理A, B中元素值相同的结点, 应删除。
    u=p; p=p->next; free(u); } //删除结点

```

18. [题目分析] 本题要求对单链表结点的元素值进行运算, 判断元素值是否等于其序号的平方减去其前驱的值。这里主要技术问题是结点的序号和前驱及后继指针的正确指向。

```

int Judge (LinkedList la)
//la是结点的元素为整数的单链表。本算法判断从第二结点开始, 每个元素值是否等于其序号的平方减去其前驱的值, 如是返回true; 否则, 返回false。
{p=la->next->next; //p是工作指针, 初始指向链表的第二项。
pre=la->next; //pre是p所指结点的前驱指针。
i=2; //i是la链表中结点的序号, 初始值为2。
while (p!=null)
    if (p->data==i*i-pre->data) {i++; pre=p; p=p->next; } //结点值间的关系符合题目要求
    else break; //当前结点的值不等于其序号的平方减去前驱的值。
if (p!=null) return (false); //未查到表尾就结束了。
else return (true); //成功返回。
} //算法结束。

```

[算法讨论] 本题不设头结点也无影响。另外, 算法中还可节省前驱指针pre, 其算法片段如下:

```

p=la; //假设无头结点, 初始p指向第一元素结点。
i=2;
while (p->next!=null) //初始p->next指向第二项。
    if (p->next->data==i*i-p->data)
        {i++; p=p->next; }
    if (p->next!=null) return (false); //失败
    else return (true); //成功

```

19. [题目分析] 本题实质上是一个模式匹配问题, 这里匹配的元素是整数而不是字符。因两整数序列已存入两个链表中, 操作从两链表的第一个结点开始, 若对应数据相等, 则后移指针; 若对应数据不等, 则A链表从上次开始比较结点的后继开始, B链表仍从第一结点开始比较, 直到B链表到尾表示匹配成功。A链表到尾B链表未到尾表示失败。操作中应记住A链表每次的开始结点, 以便下趟匹配时好从其后继开始。

```

int Pattern (LinkedList A, B)
//A和B分别是数据域为整数的单链表, 本算法判断B是否是A的子序列。如是, 返回1; 否则, 返回0表示失败。
{p=A; //p为A链表的工作指针, 本题假定A和B均无头结点。
pre=p; //pre记住每趟比较中A链表的开始结点。
q=B; //q是B链表的工作指针。
while (p && q)
    if (p->data==q->data) {p=p->next; q=q->next; }
    else {pre=pre->next; p=pre; //A链表新的开始比较结点。
        q=B; } //q从B链表第一结点开始。
if (q==null) return (1); //B是A的子序列。
else return (0); //B不是A的子序列。
} //算法结束。

```

20. [题目分析] 本题也是模式匹配问题, 应先找出链表L2在链表L1中的出现, 然后将L1中的L2倒置过来。设L2在L1中出现时第一个字母结点的前驱的指针为p, 最后一个字母结点在L1中为q所指结点的前驱, 则在保存p后继结点指针(s)的情况下, 执行p->next=q。之后将s到q结点的前驱依次插入到p结点之后, 实现了L2在L1中的倒置。

```

LinkedList PatternInvert (LinkedList L1, L2)
//L1和L2均是带头结点的单链表, 数据结点的数据域均为一个字符。本算法将L1中与L2中数据域相同的连续结点的顺序完全倒置过来。
{p=L1; //p是每趟匹配时L1中的起始结点前驱的指针。
q=L1->next; //q是L1中的工作指针。
s=L2->next; //s是L2中的工作指针。
while (p!=null && s!=null)
    if (q->data==s->data) {q=q->next; s=s->next;} //对应字母相等, 指针后移。
    else {p=p->next; q=p->next; s=L2->next; } //失配时, L1起始结点后移, L2从首结点开始。
if (s==null) //匹配成功, 这时p为L1中与L2中首字母结点相同数据域结点的前驱, q为L1中与L2最后一个结点相同数据域结点的后继。
    {r=p->next; //r为L1的工作指针, 初始指向匹配的首字母结点。
    p->next=q; //将p与q结点的链接。
    while (r!=q); //逐结点倒置。
    {s=r->next; //暂存r的后继。

```

```

        r->next=p->next; //将r所指结点倒置。
        p->next=r;
        r=s;             //恢复r为当前结点。
    }
}
else printf(“L2并未在L1中出现”);
} //算法结束。

```

【算法讨论】本算法只讨论了L2在L1至多出现一次（可能没出现），没考虑在L1中多次出现的情况。若考虑多次出现，可在上面算法找到第一次出现后的q结点作L1中下次比较的第一字母结点，读者可自行完善之。

类似本题的另外叙述题的解答：

(1) [题目分析] 本题应先查找第i个结点，记下第i个结点的指针。然后从第i+1个结点起，直至第m ($1 < i < m$) 个结点止，依次插入到第i-1个结点之后。最后将暂存的第i个结点的指针指向第m结点，形成新的循环链表，结束了倒置算法。

```

LinkedList PatternInvert1(LinkedList L, int i, m)

```

//L是有m个结点的链表的头结点的指针。表中从第i ($1 < i < m$) 个结点到第m个结点构成循环部分链表，本算法将这部分循环链表倒置。

```

{if (i<1 || i>m || m<4) {printf(“%d,%d参数错误\n”,i,m); exit(0); }
p=L->next->next;           //p是工作指针，初始指向第二结点（已假定i>1）。
pre=L->next;               //pre是前驱结点指针，最终指向第i-1个结点。
j=1;                       //计数器
while (j<i-1)              //查找第i个结点。
    {j++; pre=p; p=p->next; } //查找结束，p指向第i个结点。
q=p;                       //暂存第i个结点的指针。
p=p->next;                  //p指向第i+1个结点，准备逆置。
j+=2;                       //上面while循环结束时，j=i-1，现从第i+1结点开始逆置。
while (j<=m)
    {r=p->next;              //暂存p的后继结点。
    p->next=pre->next;        //逆置p结点。
    pre->next=p;
    p=r;                    //p恢复为当前待逆置结点。
    j++;                    //计数器增1。
    }
q->next=pre->next; //将原第i个结点的后继指针指向原第m个结点。

```

【算法讨论】算法中未深入讨论i, m, j的合法性，因题目的条件是 $m > 3$ 且 $1 < i < m$ 。因此控制循环并未用指针判断（如一般情况下的 $p != null$ ），结束循环也未用指针判断。注意最后一句 $q->next=pre->next$ ，实现了从原第i个结点到原第m个结点的循环。最后 $pre->next$ 正是指向原第m个结点，不可用 $p->next$ 代替 $pre->next$ 。

21. [题目分析] 顺序存储结构的线性表的逆置，只需一个变量辅助空间。算法核心是选择循环控制变量的初值和终值。

```

void SeqInvert (ElemType a[ ], int n)

```

//a是具有n个元素用一维数组存储的线性表，本算法将其逆置。

```

{for (i=0; i<= (n-1) /2; i++)
    {t=a[i]; a[i]= a[n-1-i]; a[n-1-i]=t; }
} //算法结束

```

【算法讨论】算法中循环控制变量的初值和终值是关键。C中数组从下标0开始，第n个元素的下标是n-1。因为首尾对称交换，所以控制变量的终值是线性表长度的一半。当n为偶数，“一半”恰好是线性表长度的二分之一；若n是奇数，“一半”是小于n/2的最大整数，这时取大于1/2的最小整数的位置上的元素，恰是线性表中间位置的元素，不需要逆置。另外，由于pascal数组通常从下标1开始，所以，上下界处理上略有不同。这点请读者注意。

类似本题的其它题的解答：

这一组又选了6个题，都是单链表（包括单循环链表）的逆置。链表逆置的通常作法是：将工作指针指向第一个元素结点，将头结点的指针域置空。然后将链表各结点从第一结点开始直至最后一个结点，依次前插至头结点后，使最后插入的结点成为链表的第一结点，第一个插入的结点成为链表的最后结点。

(1) 要求编程实现带头结点的单链表的逆置。首先建立一单链表，然后逆置。

```

typedef struct node
{int data; //假定结点数据域为整型。
struct node *next;
}node,*LinkedList;
LinkedList creat ( )
{LinkedList head, p
int x;
head= (LinkedList) malloc (sizeof (node));
head->next=null; /*设置头结点*/

```

```

scanf ( "%d" , &x ) ;
while (x!=9999) /*约定输入9999时退出本函数*/
{p= (LinkedList) malloc (sizeof (node) ) ;
p->data=x;
p->next=head->next; /* 将新结点链入链表*/
head->next=p;
scanf ( "%d" , &x ) ;
}
return (head) ;
} //结束creat函数。
LinkedList invert1 (LinkedList head)
/*逆置单链表*/
{LinkedList p=head->next; /*p为工作指针*/
head->next=null;
while (p!=null)
{r=p->next; /*暂存p的后继*/
p->next=head->next;
head->next=p;
p=r;
}
return (head) ;
} /*结束invert1函数*/

```

```
main ( )
```

```

{LinkedList la;
la=creat ( ) ; /*生成单链表*/
la=invert1 (la) ; /*逆置单链表*/
}

```

(2) 本题要求将数据项递减有序的单链表重新排序, 使数据项递增有序, 要求算法复杂度为 $O(n)$ 。虽没说要求将链表逆置, 这只是叙述不同, 本质上是将单链表逆置, 现编写如下:

```

LinkedList invert2 (LinkedList la)
//la是带头结点且数据项递减有序的单链表, 本算法将其排列成数据项递增有序的单链表。
{p=la->next; /*p为工作指针*/
la->next=null;
while (p!=null)
{r=p->next; /*暂存p的后继。*/
p->next=la->next; /*将p结点前插入头结点后。*/
la->next=p; p=r;
}
} //结束算法

```

(3) 本题要求倒排循环链表, 与上面倒排单链表处理不同之处有二: 一是初始化成循环链表而不是空链表; 二是判断链表尾不用空指针而用是否是链表头指针。算法中语句片段如下:

```

p=la->next; //p为工作指针。
la->next=la; //初始化成空循环链表。
while (p!=la) //当p=la时循环结束。
{r=p->next; //暂存p的后继结点
p->next=la->next; //逆置
la->next=p; p=r;
}

```

(4) 不带头结点的单链表逆置比较复杂, 解决方法可以给加上头结点:

```

la= (LinkedList) malloc (sizeof (node) ) ;
la->next=L;

```

之后进行如上面 (2) 那样的逆置, 最后再删去头结点:

```
L=la->next; //L是不带头结点的链表的指针。
```

```
free (la) ; //释放头结点。
```

若不增加头结点, 可用如下语句片段:

```
p=L->next; //p为工作指针。
```

```
L->next=null; //第一结点成为尾结点。
```

```
while (p!=null)
```

```
{r=p->next;
```

```
p->next=L; //将p结点插到L结点前面。
```

```
L=p; //L指向新的链表“第一”元素结点。
```

```
p=r;
}
```

(5) 同(4)，只是叙述有异。

(6) 同(2)，差别仅在于叙述不同。

22. [题目分析] 在无序的单链表上，查找最小值结点，要查遍整个链表，初始假定第一结点是最小值结点。当找到最小值结点后，判断数据域的值是否是奇数，若是，则“与其后继结点的值相交换”即仅仅交换数据域的值，用三个赋值语句即可交换。若与后继结点交换位置，则需交换指针，这时应知道最小值结点的前驱。至于删除后继结点，则通过修改最小值结点的指针域即可。

[算法设计]

```
void MiniValue (LinkedList la)
```

//la是数据域为正整数且无序的单链表，本算法查找最小值结点且打印。若最小值结点的数值是奇数，则与后继结点值交换；否则，就删除其直接后继结点。

```
{p=la->next;    // 设la是头结点的头指针，p为工作指针。
pre=p;          // pre指向最小值结点，初始假定首元结点值最小。
while (p->next!=null) // p->next是待比较的当前结点。
{if (p->next->data<pre->data) pre=p->next;
 p=p->next;    // 后移指针
}
printf (“最小值=%d\n”, pre->data);
if (pre->data%2!=0)    // 处理奇数
if (pre->next!=null) // 若该结点没有后继，则不必交换
{t= pre->data; pre->data=pre->next->data; pre->next->data=t; } // 交换完毕
else // 处理偶数情况
if (pre->next!=null) // 若最小值结点是最后一个结点，则无后继
{u=pre->next; pre->next=u->next; free (u); } // 释放后继结点空间
```

23. [题目分析] 将一个结点数据域为字符的单链表，分解成含有字母字符、数字字符和其它字符的三个循环链表，首先要构造分别含有这三类字符的表头结点。然后从原链表第一个结点开始，根据结点数据域是字母字符、数字字符和其它字符而分别插入到三个链表之一的链表。注意不要因结点插入新建链表而使原链表断链。另外，题目并未要求链表有序，插入采用“前插法”，每次插入的结点均成为所插入链表的第一元素的结点即可。

```
void OneToThree (LinkedList L, la, ld, lo)
```

//L是无头结点的单链表第一个结点的指针，链表中的数据域存放字符。本算法将链表L分解成含有英文字母字符、数字字符和其它字符的带头结点的三个循环链表。

```
{la= (LinkedList) malloc (sizeof (LNode)); // 建立三个链表的头结点
ld= (LinkedList) malloc (sizeof (LNode));
lo= (LinkedList) malloc (sizeof (LNode));
la->next=la; ld->next=ld; lo->next=lo; // 置三个循环链表为空表
while (L!=null) // 分解原链表。
{r=L; L=L->next; // L指向待处理结点的后继
if (r->data>= 'a' && r->data<= 'z' || r->data>= 'A' && r->data<= 'Z' )
{r->next=la->next; la->next=r; } // 处理字母字符。
else if (r->data>= '0' && r->data<= '9' )
{r->next=ld->next; ld->next=r; } // 处理数字字符
else {r->next=lo->next; lo->next=r; } // 处理其它符号。
} // 结束while (L!=null)。
```

// 算法结束

[算法讨论] 算法中对L链表中每个结点只处理一次，时间复杂度 $O(n)$ ，只增加了必须的三个表头结点，符合题目“用最少的时间和最少的空间”的要求。

24. [题目分析] 在递增有序的线性表中，删除数值相同的元素，要知道被删除元素结点的前驱结点。

```
LinkedList DelSame (LinkedList la)
```

//la是递增有序的单链表，本算法去掉数值相同的元素，使表中不再有重复的元素。

```
{pre=la->next; // pre是p所指向的前驱结点的指针。
p=pre->next; // p是工作指针。设链表中至少有一个结点。
while (p!=null)
if (p->data==pre->data) // 处理相同元素值的结点
{u=p; p=p->next; free (u); } // 释放相同元素值的结点
else {pre->next=p; pre=p; p=p->next; } // 处理前驱，后继元素值不同
pre->next=p; // 置链表尾。
} // DelSame
```

[算法讨论] 算法中假设链表至少有一个结点，即初始时pre不为空，否则p->next无意义。算法中最后pre->next=p是必须的，因为可能链表最后有数据域值相同的结点，这些结点均被删除，指针后移使p=null而退出while循环，所以应有pre->next=p使链表有尾。若链表尾部没数据域相同的结点，pre和p为前驱和后继，

pre->next=p也是对的。

顺便提及，题目应叙述为非递减有序，因为“递增”是说明各结点数据域不同，一个值比一个值大，不会存在相同值元素。

25. [题目分析] 建立递增有序的顺序表，对每个输入数据，应首先查找该数据在顺序表中的位置，若表中没有该元素则插入之，如已有该元素，则不再插入，为此采用折半查找方法。

```
FUNC BinSearch (VAR a: sqliстtp; x: integer): integer;
```

// 在顺序表a中查找值为x的元素，如查找成功，返回0值，如x不在a中，则返回查找失败时的较大下标值。

```
low:=1; high:=a.last; found:=false;
```

```
WHILE (low<=high) AND NOT found DO
```

```
    [mid:=(low+high) DIV 2;
```

```
    IF a.elem[mid]=x THEN found:=true
```

```
    ELSE IF a.elem[mid]>x THEN high:=mid-1 ELSE low:=mid+1;
```

```
    ]
```

```
IF found=true THEN return (0)
```

```
ELSE return (low); // 当查找失败时，low=high+1。
```

```
ENDF; // 结束对分查找函数。
```

```
PROC create (VAR L: sqliстtp)
```

```
    // 本过程生成顺序表L。
```

```
L.last:=0; // 顺序表L初始化。
```

```
read (x);
```

```
WHILE x<>9999 DO // 设x=9999时退出输入
```

```
    [k:=binsearch (L, x); // 去查找x元素。
```

```
    IF k<>0 // 不同元素才插入
```

```
        THEN [FOR i:=L.last DOWNT0 k DO L.elem[i+1]:=L.elem[i];
```

```
            L.elem[k]=x; L.last:= L.last+1; // 插入元素x，线性表长度增1
```

```
        ]
```

```
    read (x);
```

```
    ]
```

```
ENDP; // 结束过程creat
```

26. [题目分析] 在由正整数序列组成的有序单链表中，数据递增有序，允许相等整数存在。确定比正整数x大的数有几个属于计数问题，相同数只计一次，要求记住前驱，前驱和后继值不同时移动前驱指针，进行计数。将比正整数x小的数按递减排序，属于单链表的逆置问题。比正整数x大的偶数从表中删除，属于单链表中结点的删除，必须记住其前驱，以使链表不断链。算法结束时，链表中结点的排列是：小于x的数按递减排列，接着是x（若有的话），最后是大于x的奇数。

```
void exam (LinkedList la, int x)
```

// la是递增有序单链表，数据域为正整数。本算法确定比x大的数有几个；将比x小的数按递减排序，并将比x大的偶数从链表中删除。）

```
{p=la->next; q=p; // p为工作指针 q指向最小值元素，其可能的后继将是>=x的第一个元素。
```

```
pre=la; // pre为p的前驱结点指针。
```

```
k=0; // 计数（比x大的数）。
```

```
la->next=null; // 置空单链表表头结点。
```

```
while (p && p->data<x) // 先解决比x小的数按递减次序排列
```

```
    {r=p->next; // 暂存后继
```

```
    p->next=la->next; // 逆置
```

```
    la->next=p;
```

```
    p=r; // 恢复当前指针。退出循环时，r指向值>=x的结点。
```

```
    }
```

```
q->next=p; pre=q; // pre指向结点的前驱结点
```

```
while (p->data==x) {pre=p; p=p->next;} // 从小于x到大于x可能经过等于x
```

```
while (p) // 以下结点的数据域的值均大于x
```

```
    {k++; x=p->data; // 下面仍用x表示数据域的值，计数
```

```
    if (x % 2==0) // 删偶数
```

```
        {while (p->data==x)
```

```
            {u=p; p=p->next; free(u); }
```

```
        pre->next=p; // 拉上链
```

```
        }
```

```
    else // 处理奇数
```

```
        while (p->data==x) // 相同数只记一次
```

```
            {pre->next=p; pre=p; p=p->next; }
```

```
    } // while(p)
```

```
printf ( “比值%d大的数有%d个\n” , x, k );
```

```
    } // 算法exam结束
```

[算法讨论] 本题“要求用最少的时间和最小的空间”。本算法中“最少的时间”体现在链表指针不回溯，最小空间是利用了几个变量。在查比x大的数时，必须找到第一个比x大的数所在结点（因等于x的数可能有，也可能多个，也可能没有）。之后，计数据的第一次出现，同时删去偶数。

顺便指出，题目设有“按递增次序”的“有序单链表”，所给例子序列与题目的论述并不一致。

27. [题目分析] 单链表中查找任何结点，都必须从头指针开始。本题要求将指针p所指结点与其后继结点交换，这不仅要求知道p结点，还应知道p的前驱结点。这样才能在p与其后继结点交换后，由原p结点的前驱来指向原p结点的后继结点。

另外，若无特别说明，为了处理的方便统一，单链表均设头结点，链表的指针就是头结点的指针。并且由于链表指针具有标记链表的作用，也常用指针名冠以链表名称。如“链表head”既指的是链表的名字是head，也指出链表的头指针是head。

```
LinkedList Exchange (LinkedList HEAD, p)
```

```
// HEAD是单链表头结点的指针，p是链表中的一个结点。本算法将p所指结点与其后继结点交换。
```

```
{q=head->next; // q是工作指针，指向链表中当前待处理结点。
```

```
pre=head; // pre是前驱结点指针，指向q的前驱。
```

```
while (q!=null && q!=p) {pre=q; q=q->next; } // 未找到p结点，后移指针。
```

```
if (p->next==null) printf ( “p无后继结点\n” ); // p是链表中最后一个结点，无后继。
```

```
else // 处理p和后继结点交换
```

```
{q=p->next; // 暂存p的后继。
```

```
pre->next=q; // p前驱结点的后继指向p的后继。
```

```
p->next=q->next; // p的后继指向原p后继的后继。
```

```
q->next=p ; // 原p后继的后继指针指向p。
```

```
}
```

```
} // 算法结束。
```

类似本题的其它题目的解答：

(1) 与上面第27题基本相同，只是明确说明“p指向的不是链表最后那个结点。”

(2) 与上面第27题基本相同，仅叙述不同，故不再作解答。

28. [题目分析] 本题链表结点的数据域存放英文单词，可用字符数组表示，单词重复出现时，链表中只保留一个，单词是否相等的判断使用strcmp函数，结点中增设计数域，统计单词重复出现的次数。

```
typedef struct node
```

```
{int freg; // 频度域，记单词出现的次数。
```

```
char word[maxsize]; // maxsize是单词中可能含有的最多字母个数。
```

```
struct node *next;
```

```
}node, *LinkedList;
```

```
(1)LinkedList creat ()
```

```
// 建立有n (n>0) 个单词的单向链表，若单词重复出现，则只在链表中保留一个。
```

```
{LinkedList la;
```

```
la= (LinkedList) malloc (sizeof (node) ); // 申请头结点。
```

```
la->next=null; // 链表初始化。
```

```
for (i=1; i<=n; i++) // 建立n个结点的链表
```

```
{scanf ( “%s” , a ); // a是与链表中结点数据域同等长度的字符数组。
```

```
p=la->next; pre=p; // p是工作指针,pre是前驱指针。
```

```
while (p!=null)
```

```
if (strcmp(p->data, a) ==0) {p->freg++; break; } // 单词重复出现,频度增1。
```

```
else {pre=p; p=p->next; } // 指针后移。
```

```
if (p==null) // 该单词没出现过，应插入。
```

```
{p= (LinkedList) malloc (sizeof (node) );
```

```
strcpy (p->data, a) ; p->freg=1; p->next=null; pre->next=p;
```

```
} // 将新结点插入到链表最后。
```

```
} // 结束for循环。
```

```
return (la) ;
```

```
} // 结束creat算法。
```

```
(2) void CreatOut ( )
```

```
// 建立有n个单词的单向链表，重复单词只在链表中保留一个，最后输出频度最高的k个单词。
```

```
{LinkedList la;
```

```
la= (LinkedList) malloc (sizeof (node) ); // 申请头结点。
```

```
la->next=null; // 链表初始化。
```

```
for (i=1; i<=n; i++) // 建立n个结点的链表
```

```
{scanf ( “%s” , a ); // a是与链表中结点数据域同等长度的字符数组。
```

```
p=la->next; pre=p; // p是工作指针,pre是前驱指针。
```

```
while (p!=null)
```

```

    if (strcmp(p->data, a)==0)
    {p->freg++;          // 单词重复出现, 频度增1。
    pre->next=p->next; // 先将p结点从链表上摘下, 再按频度域值插入到合适位置
    pre=la; q=la->next;
    while(q->freg>p->freg) (pre=q; q=q->next; )
    pre->next=p; p->next=q; // 将p结点插入到合适位置
    }
    else {pre=p; p=p->next; }      // 指针后移。
if (p==null)          // 该单词没出现过, 应插入到链表最后。
{p= (LinkedList) malloc (sizeof (node) );
  strcpy (p->data, a) ; p->freg=1; p->next=null; pre->next=p;
} // if 新结点插入。
} // 结束for循环建表。
int k, i=0;
scanf( "输入要输出单词的个数%d" , &k);
p=la->next;
while (p && i<k) // 输出频度最高的k个单词
{printf ( "第%d个单词%s出现%d次\n" , ++i, p->data, p->freg) ;
  p=p->next;
}
if (!p)
  printf( "给出的%d值太大\n" , k);
} // 结束算法

```

29. [题目分析] 双向循环链表自第二结点至表尾递增有序, 要求将第一结点插入到链表中, 使整个链表递增有序。由于已给条件 ($a_1 < x < a_n$), 故应先将第一结点从链表上摘下来, 再将其插入到链表中相应位置。由于是双向链表, 不必象单链表那样必须知道插入结点的前驱。

```

void DInsert (DLinkedList dl)
// dl是无头结点的双向循环链表, 自第二结点起递增有序。本算法将第一结点 ( $a_1 < x < a_n$ ) 插入到链表中,
// 使整个链表递增有序。
{ s=la;          // s暂存第一结点的指针。
  p=la->next; p->prior=la->prior; p->prior->next=p; // 将第一结点从链表上摘下。
  while (p->data<x) p=p->next;    // 查插入位置
  s->next=p; s->prior=p->prior; p->prior->next=s; p->prior=s; // 插入原第一结点s
} // 算法结束。

```

[算法讨论] 由于题目已给 $a_1 < x < a_n$, 所以在查找第一结点插入位置时用的循环条件是 $p->data < x$, 即在 a_1 和 a_n 间肯定能找到第一结点的插入位置。若无此条件, 应先看第一结点数据域值 x 是否小于等于 a_1 , 如是, 则不作任何操作。否则, 查找其插入位置, 循环控制要至多查找完 a_1 到 a_n 结点。

```

if (p->data<x) p=p->next; else break;

```

30. [题目分析] 在顺序存储的线性表上删除元素, 通常要涉及到一系列元素的移动 (删第 i 个元素, 第 $i+1$ 至第 n 个元素要依次前移)。本题要求删除线性表中所有值为 $item$ 的数据元素, 并未要求元素间的相对位置不变。因此可以考虑设头尾两个指针 ($i=1, j=n$), 从两端向中间移动, 凡遇到值 $item$ 的数据元素时, 直接将右端元素左移至值为 $item$ 的数据元素位置。

```

void Delete (ElemType A[ ], int n)
// A是有n个元素的一维数组, 本算法删除A中所有值为item的元素。
{ i=1; j=n; // 设置数组低、高端指针 (下标)。
  while (i<j)
  { while (i<j && A[i]!=item) i++;          // 若值不为item, 左移指针。
    if (i<j) while (i<j && A[j]==item) j--; // 若右端元素值为item, 指针左移
    if (i<j) A[i++]=A[j--];
  }
}

```

[算法讨论] 因元素只扫描一趟, 算法时间复杂度为 $O(n)$ 。删除元素未使用其它辅助空间, 最后线性表中的元素个数是 j 。若题目要求元素间相对顺序不变, 请参见本章三、填空题25的算法。

31. [题目分析] 本题所用数据结构是静态双向链表, 其结构定义为:

```

typedef struct node
{char data[maxsize]; // 用户姓名, maxsize是可能达到的用户名的最大长度。
  int Llink, Rlink; // 前向、后向链, 其值为乘客数组下标值。
}unode;
unode user[max]; // max是可能达到的最多客户数。

```

设 av 是可用数组空间的最小下标, 当有客户要订票时, 将其姓名写入该单元的 $data$ 域, 然后在静态链表中查找其插入位置。将该乘客姓名与链表中第一个乘客姓名比较, 根据大于或小于第一个乘客姓名, 而决定沿第一个乘客的右链或左链去继续查找, 直到找到合适位置插入之。

```

void Insert (unode user[max], int av)

```

//user是静态双向链表，表示飞机票订票系统，元素包含data、Llink和Rlink三个域，结点按来客姓名排序。本算法处理任一乘客订票申请。

```
{scanf ( "%s" , s ) ;           //s是字符数组，存放乘客姓名。
strcpy (user[av].data, s) ;
p=1;           //p为工作指针（下标）
if(strcmp (user[p].data, s) <0) //沿右链查找
{while (p!=0 && strcmp (user[p].data, s) <0) {pre=p; p=user[p].Rlink; }
user[av].Rlink=p; user[av].Llink=pre; //将新乘客链入表中
user[pre].Rlink=av; user[p].Llink=av;
}
else //沿左右链查找
{while (p!=0 && strcmp (user[p].data, s) >0) {pre=p; p=user[p].Llink; }
user[av].Rlink=pre; user[av].Llink=p; //将新乘客链入表中
user[pre].Llink=av; user[p].Rlink=av;
}
} //算法结束
```

[算法讨论] 本算法只讨论了乘客订票情况，未考虑乘客退票。也未考虑从空开始建立链表。增加乘客时也未考虑姓名相同者（实际系统姓名不能做主关键字）。完整系统应有（1）初始化，把整个数组空间初始化成双向静态链表，全部空间均是可利用空间。（2）申请空间。当有乘客购票时，要申请空间，直到无空间可用为止。（3）释放空间。当乘客退票时，将其空间收回。由于空间使用无优先级，故可将退票释放的空间作为下个可利用空间，链入可利用空间表中。

32. [题目分析] 首先在双向链表中查找数据值为x的结点，查到后，将结点从链表上摘下，然后再顺结点的前驱链查找该结点的位置。

```
DLinkedList locate(DLinkedList L, ElemType x)
// L是带头结点的按访问频度递减的双向链表，本算法先查找数据x，查找成功时结点的访问频度域增1，最后将该结点按频度递减插入链表中适当位置。
{ DLinkedList p=L->next, q; //p为L表的工作指针，q为p的前驱，用于查找插入位置。
while (p && p->data !=x) p=p->next; // 查找值为x的结点。
if (!p) {printf( "不存在值为x的结点\n" ); exit(0);}
else { p->freq++; // 令元素值为x的结点的freq域加1 。
p->next->pred=p->pred; // 将p结点从链表上摘下。
p->pred->next=p->next;
q=p->pred; // 以下查找p结点的插入位置
while (q !=L && q->freq < p->freq) q=q->pred;
p->next=q->next; q->next->pred=p; // 将p结点插入
p->pred=q; q->next=p;
}
return(p); // 返回值为x的结点的指针
} // 算法结束
```

33. [题目分析] 题目要求按递增次序输出单链表中各结点的数据元素，并释放结点所占存储空间。应对链表进行遍历，在每趟遍历中查找出整个链表的最小值元素，输出并释放结点所占空间；再查次最小值元素，输出并释放空间，如此下去，直至链表为空，最后释放头结点所占存储空间。当然，删除结点一定要记住该结点的前驱结点的指针。

void MiniDelete (LinkedList head)
//head是带头结点的单链表的头指针，本算法按递增顺序输出单链表中各结点的数据元素，并释放结点所占的存储空间。

```
{while (head->next!=null) //循环到仅剩头结点。
{pre=head; //pre为元素最小值结点的前驱结点的指针。
p=pre->next; //p为工作指针
while (p->next!=null)
{if (p->next->data < pre->next->data) pre=p; //记住当前最小值结点的前驱
p=p->next;
}
printf (pre->next->data) ; //输出元素最小值结点的数据。
u=pre->next; pre->next=u->next; free (u) ; //删除元素值最小的结点，释放结点空间
} // while (head->next!=null)
free (head) ; } //释放头结点。
```

[算法讨论] 算法中使用的指针变量只有pre，p和u三个，请读者细心体会。要注意没特别记最小值结点，而是记其前驱。

34. [题目分析] 留下三个链表中公共数据，首先查找两表A和B中公共数据，再去C中找有无该数据。要消除重复元素，应记住前驱，要求时间复杂度 $O(m+n+p)$ ，在查找每个链表时，指针不能回溯。

```
LinkedList Common (LinkedList A, B, C)
```

//A, B和C是三个带头结点且结点元素值非递减排列的有序表。本算法使A表仅留下三个表均包含的结点, 且结点值不重复, 释放所有结点。

```
{pa=A->next; pb=B->next; pc=C->next; // pa, pb和pc分别是A, B和C三个表的工作指针。
pre=A; //pre是A表中当前结点的前驱结点的指针。
while (pa && pb && pc) // 当A, B和C表均不空时, 查找三表共同元素
{ while (pa && pb)
  if (pa->data<pb->data) {u=pa; pa=pa->next; free (u) ; } // 结点元素值小时, 后移指针。
  else if (pa->data>pb->data) pb=pb->next;
  else if (pa && pb) // 处理A和B表元素值相等的结点
    {while (pc && pc->data<pa->data) pc=pc->next;
     if(pc)
       {if (pc->data>pa->data) // pc当前结点值与pa当前结点值不等, pa后移指针。
        {u=pa; pa=pa->next; free (u) ; }
        else // pc, pa和pb对应结点元素值相等。
          {if(pre==A) { pre->next=pa; pre=pa; pa=pa->next} // 结果表中第一个结点。
           else if(pre->data==pa->data) // (处理) 重复结点不链入A表
            {u=pa; pa=pa->next; free (u) ; }
           else {pre->next=pa; pre=pa; pa=pa->next; } // 将新结点链入A表。
          pb=pb->next; pc=pc->next; // 链表的工作指针后移。
         } } // else pc, pa和pb对应结点元素值相等
    if (pa==null) pre->next=null; // 原A表已到尾, 置新A表表尾
    else // 处理原A表未到尾而B或C到尾的情况
      {pre->next=null; // 置A表表尾标记
       while (pa!=null) // 删除原A表剩余元素。
        {u=pa; pa=pa->next; free (u) ; }
      }
```

[算法讨论] 算法中A表、B表和C表均从头到尾(严格说B、C中最多一个到尾)遍历一遍, 算法时间复杂度符合 $O(m+n+p)$ 。算法主要由while (pa && pb && pc)控制。三表有一个到尾则结束循环。算法中查到A表与B表和C表的公共元素后, 又分三种情况处理: 一是三表中第一个公共元素值相等的结点; 第二种情况是, 尽管不是第一结点, 但与前驱结点元素值相同, 不能成为结果表中的结点; 第三种情况是新结点与前驱结点元素值不同, 应链入结果表中, 前驱指针也移至当前结点, 以便与以后元素值相同的公共结点进行比较。算法最后要给新A表置结尾标记, 同时若原A表没到尾, 还应释放剩余结点所占的存储空间。

第三章 栈和队列

一、选择题

1. B	2. 1B	2. 2A	2. 3B	2. 4D	2. 5. C	3. B	4. D	5. D	6. C	7. D	8. B
9. D	10. D	11. D	12. C	13. B	14. C	15. B	16. D	17. B	18. B	19. B	20. D
21. D	22. D	23. D	24. C	25. A	26. A	27. D	28. B	29. BD	30. C	31. B	32. C
33. 1B	33. 2A	33. 3C	33. 4C	33. 5F	34. C	35. C	36. A	37. AD			

二、判断题

1. \checkmark	2. \checkmark	3. \checkmark	4. \checkmark	5. \times	6. \checkmark	7. \checkmark	8. \checkmark	9. \checkmark	10. \times	11. \checkmark	12. \times
13. \times	14. \times	15. \checkmark	16. \times	17. \checkmark	18. \times	19. \checkmark	20. \checkmark				

部分答案解释如下。

1、尾递归的消除就不需用栈

2、这个数是前序序列为1,2,3,...,n, 所能得到的不相似的二叉树的数目。

三、填空题

- 操作受限（或限定仅在表尾进行插入和删除操作） 后进先出
- 栈 3、3 1 2 4、23 100CH 5、0 n+1 top[1]+1=top[2]
- 两栈顶指针值相减的绝对值为1（或两栈顶指针相邻）。
- (1)满 (2)空 (3)n (4)栈底 (5)两栈顶指针相邻（即值之差的绝对值为1）
- 链式存储结构 9、 $S \times SS \times S \times \times$ 10、data[++top]=x;
23. 12. 3*2-4/34. 5*7/++108. 9/+（注：表达式中的点(.)表示将数隔开，如23. 12. 3是三个数）
- 假溢出时大量移动数据元素。
- (M+1) MOD N (M+1)% N; 14、队列 15、先进先出 16、先进先出
- s=(LinkedList)malloc(sizeof(LNode)); s->data=x;s->next=r->next; r->next=s; r=s;
- 牺牲一个存储单元 设标记
- (TAIL+1) MOD M=FRONT（数组下标0到M-1，若一定使用1到M，则取模为0者，值改取M
- sq. front=(sq. front+1)%(M+1); return(sq. data(sq. front)); (sq. rear+1)%(M+1)==sq. front;
- 栈 22、(rear-front+m) % m; 23、(R-P+N) % N;
- (1) a[i]或a[1] (2) a[i] (3) pop(s) 或s[1];
- (1) PUSH(OPTR, w) (2) POP(OPTR) (3) PUSH(OPND, operate(a, theta, b))
- (1) T>0 (2) i<n (3) T>0 (4) **top<n** (5) top+1 (6) true (7) i-1 (8) top-1 (9) T+w[i] (10) false

四、应用题

1、栈是只准在一端进行插入和删除操作的线性表，允许插入和删除的一端叫栈顶，另一端叫栈底。最后插入的元素最先删除，故栈也称后进先出（**LIFO**）表。

2、队列是允许在一端插入而在另一端删除的线性表，允许插入的一端叫队尾，允许删除的一端叫队头。最先插入队的元素最先离开（删除），故队列也常称先进先出（**FIFO**）表。

3、用常规意义下顺序存储结构的一维数组表示队列，由于队列的性质（队尾插入和队头删除），容易造成“假溢出”现象，即队尾已到达一维数组的高下标，不能再插入，然而队中元素个数小于队列的长度（容量）。循环队列是解决“假溢出”的一种方法。通常把一维数组看成首尾相接。在循环队列下，通常采用“牺牲一个存储单元”或“作标记”的方法解决“队满”和“队空”的判定问题。

4、(1) 通常有两条规则。第一是给定序列中S的个数和X的个数相等；第二是从给定序列的开始，到给定序列中的任一位置，S的个数要大于或等于X的个数。

(2) 可以得到相同的输出元素序列。例如，输入元素为A, B, C, 则两个输入的合法序列ABC和BAC均可得到输出元素序列ABC。对于合法序列ABC，我们使用本题约定的 $S \times S \times S \times$ 操作序列；对于合法序列BAC，我们使用 $SS \times \times S \times$ 操作序列。

5、三个：CDEBA, CDBEA, CDBAE

6、输入序列为123456，不能得出435612，其理由是，输出序列最后两元素是12，前面4个元素（4356）得到后，栈中元素剩12，且2在栈顶，不可能栈底元素1在栈顶元素2之前出栈。

得到135426的过程如下：1入栈并出栈，得到部分输出序列1；然后2和3入栈，3出栈，部分输出序列变为：13；接着4和5入栈，5，4和2依次出栈，部分输出序列变为13542；最后6入栈并退栈，得最终结果135426。

7、能得到出栈序列B、C、A、E、D，不能得到出栈序列D、B、A、C、E。其理由为：若出栈序列以D开头，说明在D之前的入栈元素是A、B和C，三个元素中C是栈顶元素，B和A不可能早于C出栈，故不可能得到D、B、A、C、E出栈序列。

8、借助栈结构，n个入栈元素可得到 $1/(n+1)((2n)!/(n!*n!))$ 种出栈序列。本题4个元素，可有14种出栈序列，abcd和dcba就是其中两种。但dabc和adbc是不可能得到的两种。

9、不能得到序列2, 5, 3, 4, 6。栈可以用单链表实现，这就是链栈。由于栈只在栈顶操作，所以链栈通常不设头结点。

10、如果 $i < j$ ，则对于 $p_i < p_j$ 情况，说明 p_i 在 p_j 入栈前先出栈。而对于 $p_i > p_j$ 的情况，则说明要将 p_j 压到 p_i 之上，也就是在 p_j 出栈之后 p_i 才能出栈。这就说明，对于 $i < j < k$ ，不可能出现 $p_j < p_k < p_i$ 的输出序列。换句话说，对于输入序列1, 2, 3, 不可能出现3, 1, 2的输出序列。

11、(1) 能得到325641。在123依次进栈后，3和2出栈，得部分输出序列32；然后4, 5入栈，5出栈，得部

分出栈序列325；6入栈并出栈，得部分输出序列3256；最后退栈，直到栈空。得输出序列325641。其操作序列为AAADDAADADD。

(2) 不能得到输出顺序为154623的序列。部分合法操作序列为ADAAAADDAD，得到部分输出序列1546后，栈中元素为23，3在栈顶，故不可能2先出栈，得不到输出序列154623。

12、(1) 一个函数在结束本函数之前，直接或间接调用函数自身，称为递归。例如，函数f在执行中，又调用函数f自身，这称为直接递归；若函数f在执行中，调用函数g，而g在执行中，又调用函数f，这称为间接递归。在实际应用中，多为直接递归，也常简称为递归。

(2) 递归程序的优点是程序结构简单、清晰，易证明其正确性。缺点是执行中占内存空间较多，运行效率低。

(3) 递归程序执行中需借助栈这种数据结构来实现。

(4) 递归程序的入口语句和出口语句一般用条件判断语句来实现。递归程序由基本项和归纳项组成。基本项是递归程序出口，即不再递归即可求出结果的部分；归纳项是将原来问题化成简单的且与原来形式一样的问题，即向着“基本项”发展，最终“到达”基本项。

13、函数调用结束时vol=14。执行过程图示如下：

14、过程p递归调用自身时，过程p由内部定义的局部变量在p的2次调用期间，不占同一数据区。每次调用都保留其数据区，这是递归定义所决定，用“递归工作栈”来实现。

15、设 H_n 为n个盘子的Hanoi塔的移动次数。(假定n个盘子从钢针X移到钢针Z，可借助钢针Y)

则 $H_n = 2H_{n-1} + 1$ //先将n-1个盘子从X移到Y，第n个盘子移到Z，再将那n-1个移到Z

$$= 2(2H_{n-2} + 1) + 1$$

$$= 2^2 H_{n-2} + 2 + 1$$

$$= 2^2 (2H_{n-3} + 1) + 2 + 1$$

$$= 2^3 H_{n-3} + 2^2 + 2 + 1$$

$$\begin{aligned} & \cdot \\ & \cdot \\ & \cdot \\ & = 2^k H_{n-k} + 2^{k-1} + 2^{k-2} + \dots + 2^1 + 2^0 \\ & = 2^{n-1} H_1 + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0 \end{aligned}$$

因为 $H_1=1$ ，所以原式 $H_n = 2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0 = 2^n - 1$

故总盘数为n的Hanoi塔的移动次数是 $2^n - 1$ 。

16、运行结果为：1 2 1 3 1 2 1 (注：运行结果是每行一个数，为节省篇幅，放到一行。)

17、两栈共享一向量空间（一维数组），栈底设在数组的两端，两栈顶相邻时为栈满。设共享数组为S[MAX]，则一个栈顶指针为-1，另一个栈顶指针为 MAX 时，栈为空。

用C写的入栈操作push(i, x)如下：

const MAX=共享栈可能达到的最大容量

typedef struct node

{elemtype s[MAX];

int top[2];

} anode;

anode ds;

int push(int i, elemtype x)

//ds为容量有MAX个类型为elemtype的元素的一维数组，由两个栈共享其空间。i的值为0或1，x为类型为elemtype的元素。本算法将x压入栈中。如压栈成功，返回1；否则，返回0。

{if (ds.top[1]-ds.top[0]==1) {printf(“栈满\n”); return (0); }

switch (i)

{case 0: ds.s[++ds.top[i]]=x; break;

case 1: ds.s[--ds.top[i]]=x;

return (1); } //入栈成功。

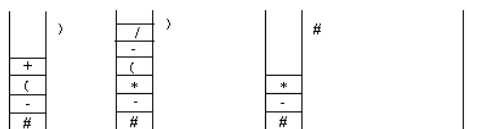
}

18、本程序段查找栈S中是否有整数为k的元素，如有，则删除。采用的办法使用另一个栈T。在S栈元素退栈时，若退栈元素不是整数k，则压入T栈。遇整数k，k不入T栈，然后将T栈元素全部退栈，并依次压入栈S中，实现了在S中删除整数k的目的。若S中无整数k，则在S退成空栈后，再将T栈元素退栈，并依次压入S栈。直至T栈空。这后一种情况下S栈内容操作前后不变。

19、中缀表达式 $8-(3+5)*(5-6/2)$ 的后缀表达式是：8 3 5 + 5 6 2 / - * -

栈的变化过程图略（请参见22题），表达式生成过程为：

(1) 8 3 5 (2) 8 3 5 + 5 6 2 (3) 8 3 5 + 5 6 2 / - (4) 8 3 5 + 5 6 2 / - * -



中缀表达式exp1转为后缀表达式exp2的规则如下：

设操作符栈s，初始为空栈后，压入优先级最低的操作符‘#’。对中缀表达式从左向右扫描，遇操作数，直接写入exp2；若是操作符（记为w），分如下情况处理，直至表达式exp1扫描完毕。

(1) w为一般操作符（‘+’，‘-’，‘*’，‘/’等），要与栈顶操作符比较优先级，若w优先级高于栈顶操作符，则入栈；否则，栈顶运算符退栈到exp2，w再与新栈顶操作符作上述比较处理，直至w入栈。

(2) w为左括号（‘（’），w入栈。

(3) w为右括号（‘）’），操作符栈退栈并进入exp2，直到碰到左括号为止，左括号退栈（不能进入exp2），右括号也丢掉，达到exp2中消除括号的目的。

(4) w为‘#’，表示中缀表达式exp1结束，操作符栈退栈到exp2，直至碰到‘#’，退栈，整个操作结束。

这里，再介绍一种简单方法。中缀表达式转为后缀表达式有三步：首先，将中缀表达式中所有的子表达式按计算规则用嵌套括号括起来；接着，顺序将每对括号中的运算符移到相应括号的后面；最后，删除所有括号。

例如，将中缀表达式 $8-(3+5)*(5-6/2)$ 转为后缀表达式。按如上步骤：

执行完上面第一步后为： $8-((3+5)*(5-(6/2))))$ ；

执行完上面第二步后为： $8((35)+(5(62)/)-)*-$ ；

执行完上面第三步后为： $8\ 3\ 5\ +\ 5\ 6\ 2\ /\ -\ *\ -$ 。

可用类似方法将中缀表达式转为前缀表达式。

20、中缀表达式转为后缀表达式的规则基本上与上面19题相同，不同之处是对运算符**优先级的规定。在算术运算中，先乘除后加减，先括号内后括号外，相同级别的运算符按从左到右的规则运算。而对**运算符，其优先级同常规理解，即高于加减乘除而小于左括号。为了适应本题中“从右到左计算”的要求，规定栈顶运算符**的级别小于正从表达式中读出的运算符**，即刚读出的运算符**级别高于栈顶运算符**，因此也入栈。

下面以 $A**B**C$ 为例说明实现过程。

读入A，不是操作符，直接写入结果表达式。再读入*，这里规定在读入*后，不能立即当乘号处理，要看下一个符号，若下个符号不是*，则前个*是乘号。这里因为下一个待读的符号也是*，故认为**是一个运算符，与运算符栈顶比较（运算符栈顶初始化后，首先压入‘#’作为开始标志），其级别高于‘#’，入栈。再读入B，直接进入结果表达式。接着读入**，与栈顶比较，均为**，我们规定，后读入的**级别高于栈顶的**，因此**入栈。接着读入C，直接到结果表达式。现在的结果（后缀）表达式是ABC。最后读入‘#’，表示输入表达式结束，这时运算符栈中从栈顶到栈底有两个**和一个‘#’。两个运算符**退栈至结果表达式，结果表达式变为ABC****。运算符栈中只剩‘#’，退栈，运算结束。

21、(1) $\text{sum}=21$ 。当x为局部变量时，每次递归调用，都要给局部变量分配存储单元，故x数值4，9，6和2均保留，其递归过程示意图如下：

(2) $\text{sum}=8$ ，当x为全局变量时，在程序的整个执行期间，x只占一个存储单元，先后读入的4个数(4,9,6,2),仅最后一个起作用。当递归调用结束，逐层返回时 $\text{sum}:=\text{sum}(n-1)+x$ 表达式中，x就是2，所以结果为 $\text{sum}=8$ 。

22、设操作数栈是opnd，操作符栈是optr，对算术表达式 $A-B*C/D-E \uparrow F$ 求值，过程如下：

步骤	opnd栈	optr栈	输入字符	主要操作
初始		#	$A-B*C/D-E \uparrow F\#$	PUSH(OPTR, ' #')
1	A	#	$A-B*C/D-E \uparrow F\#$	PUSH(OPND, A)
2	A	# -	$-B*C/D-E \uparrow F\#$	PUSH(OPTR, ' -')
3	AB	# -	$B*C/D-E \uparrow F\#$	PUSH(OPND, B)
4	AB	# - *	$*C/D-E \uparrow F\#$	PUSH(OPTR, ' *')
5	ABC	# - *	$C/D-E \uparrow F\#$	PUSH(OPND, C)
6	AT (T=B*C)	# - /	$/D-E \uparrow F\#$	PUSH(OPND, POP(OPND)*POP(OPND)) PUSH(OPTR, ' /')
7	ATD	# - /	$D-E \uparrow F\#$	PUSH(OPND, D)
8	AT (T=T/D) T (T=A-T)	# - # -	$-E \uparrow F\#$	$x=\text{POP(OPND)}; y=\text{POP(OPND)}$ PUSH(OPND, y/x) ; $x=\text{POP(OPND)}; y=\text{POP(OPND)}$; PUSH(OPND, $y-x$) PUSH(OPTR, ' -')
9	TE	# -	$E \uparrow F\#$	PUSH(OPND, E)
10	TE	# - ↑	$\uparrow F\#$	PUSH(OPTR, ' ↑')
11	TEF	# - ↑	$F\#$	PUSH(OPND, F)

12	TE TS(S=E ↑ F) R(R=T-S)	#- #	#	X=POP(OPND) Y=POP(OPND) POP(OPTR) PUSH(OPND, y ↑ x) x=POP(OPND) y=POP(OPND) POP(OPTR) PUSH(OPND, y-x)
----	-----------------------------------	-------------	---	---

23、

步骤	栈S1	栈S2	输入的算术表达式（按字符读入）
初始	—	®	A-B*C/D+E/F®
<u>1</u>	A	®	<u>A</u> -B*C/D+E/F®
<u>2</u>	A	®-	<u>-</u> B*C/D+E/F®
<u>3</u>	AB	®-	<u>B</u> *C/D+E/F®
<u>4</u>	AB	®-*	<u>*</u> C/D+E/F®
<u>5</u>	ABC	®-*	<u>C</u> /D+E/F®
<u>6</u>	AT ₁ （注：T ₁ =B*C）	®-/	<u>/</u> D+E/F®
<u>7</u>	AT ₁ D	®-/	<u>D</u> +E/F®
<u>8</u>	AT ₂ （注：T ₂ =T ₁ /D） T ₃ （注：T ₃ =A-T ₂ ）	®- ®+	<u>+</u> E/F®
<u>9</u>	T ₃ E	®+	<u>E</u> /F®
<u>10</u>	T ₃ E	®+/	<u>/</u> F®
<u>11</u>	T ₃ EF	®+/	<u>F</u> ®
<u>12</u>	T ₃ T ₄ （注：T ₄ =E/F） T ₅ （注：T ₅ =T ₃ +T ₄ ）	®+ ®	<u>®</u>

24、XSXXXSSSXXSXXSXXSSSS

25、S1和S2共享内存中一片连续空间（地址1到m），可以将S1和S2的栈底设在两端，两栈顶向共享空间的中心延伸，仅当两栈顶指针相邻（两栈顶指针值之差的绝对值等于1）时，判断为栈满，当一个栈顶指针为0，另一个栈顶指针m+1时为两栈均空。

26、设栈S1和栈S2共享向量V[1..m]，初始时，栈S1的栈顶指针top[0]=0，栈S2的栈顶指针top[1]=m+1，当top[0]=0为左栈空，top[1]=m+1为右栈空；当top[0]=0并且top[1]=m+1时为全栈空。当top[1]-top[0]=1时为栈满。

27、（1）每个栈仅用一个顺序存储空间时，操作简便，但分配存储空间小了，容易产生溢出，分配空间大了，容易造成浪费，各栈不能共享空间。

（2）多个栈共享一个顺序存储空间，充分利用了存储空间，只有在整个存储空间都用完时才能产生溢出，其缺点是当一个栈满时要向左、右栈查询有无空闲单元。如果有，则要移动元素和修改相关的栈底和栈顶指针。当接近栈满时，查询空闲单元、移动元素和修改栈底栈顶指针的操作频繁，计算复杂并且耗费时间。

（3）多个链栈一般不考虑栈的溢出（仅受用户内存空间限制），缺点是栈中元素要以指针相链接，比顺序存储多占用了存储空间。

28、设top1和top2分别为栈1和2的栈顶指针

（1）入栈主要语句

```
if(top2-top1==1) {printf(“栈满\n”); exit(0);}
case1:top1++; SPACE[top1]=x;    //设x为入栈元素。
case2:top2--; SPACE[top2]=x;
```

出栈主要语句

```
case1: if (top1==-1) {printf(“栈空\n”); exit(0);}
      top1--; return (SPACE[top1+1]);    //返回出栈元素。
case2: if (top2==N) {printf(“栈空\n”); exit(0);}
      top2++; return (SPACE[top2-1]);    //返回出栈元素。
```

（2）栈满条件: top2-top1=1

栈空条件: top1=**-1**并且top2=**N** //top1=**-1**为左栈空，top2=**N**为右栈空

29、设顺序存储队列用一维数组q[m]表示，其中m为队列中元素个数，队列中元素在向量中的下标从0到m-1。设队头指针为front，队尾指针是rear，约定front指向队头元素的前一位置，rear指向队尾元素。当front等于-1时队空，rear等于m-1时为队满。由于队列的性质（“删除”在队头而“插入”在队尾），所以当队尾指针rear等于m-1时，若front不等于-1，则队列中仍有空闲单元，所以队列并不是真满。这时若再有入队操作，会造成假“溢出”。其解决办法有二，一是将队列元素向前“平移”（占用0至rear-front-1）；二是将队列看成首尾相连，即循环队列（0..m-1）。在循环队列下，仍定义front=rear时为队空，而判断队满则用两种办法，一是用“牺牲一个单元”，即rear+1=front（准确记是（rear+1）%m=front，m是队列容量）时为队满。另一种解法是“设标记”方法，如设标记tag，tag等于0情况下，若删除时导致front=rear为队空；tag=1情况下，若因插入导致front=rear则为队满。

30、见上题29的解答。 31、参见上面29题。

32、typedef struct node

```
{elemtype elemcq[m]; //m为队列最大可能的容量。
```

```

    int front, rear;    //front和rear分别为队头和队尾指针。
}cqnode;
cqnode cq;
(1) 初始状态
    cq.front=cq.rear=0;
(2) 队列空
    cq.front==cq.rear;
(3) 队列满
    (cq.rear+1)%m==cq.front;

```

33、栈的特点是后进先出，队列的特点是先进先出。初始时设栈s1和栈s2均为空。

(1) 用栈s1和s2模拟一个队列的输入：设s1和s2容量相等。分以下三种情况讨论：若s1未满，则元素入s1栈；若s1满，s2空，则将s1全部元素退栈，再压栈入s2，之后元素入s1栈；若s1满，s2不空（已有出队列元素），则不能入队。

(2) 用栈s1和s2模拟队列出队（删除）：若栈s2不空，退栈，即是队列的出队；若s2为空且s1不空，则将s1栈中全部元素退栈，并依次压入s2中，s2栈顶元素退栈，这就是相当于队列的出队。若栈s1为空并且s2也为空，队列空，不能出队。

(3) 判队空 若栈s1为空并且s2也为空，才是队列空。

讨论：s1和s2容量之和是队列的最大容量。其操作是，s1栈满后，全部退栈并压栈入s2（设s1和s2容量相等）。再入栈s1直至s1满。这相当队列元素“入队”完毕。出队时，s2退栈完毕后，s1栈中元素依次退栈到s2，s2退栈完毕，相当于队列中全部元素出队。

在栈s2不空情况下，若要求入队操作，只要s1不满，就可压入s1中。若s1满和s2不空状态下要求队列的入队时，按出错处理。

```

34、(1) 队空 s.front=s.rear;    //设s是sequetp类型变量
(2) 队满: (s.rear+1) MOD MAXSIZE=s.front //数组下标为0.. MAXSIZE-1

```

具体参见本章应用题第29题

35、**typedef struct**

```

{elemtp q[m];
    int front, count; //front是队首指针，count是队列中元素个数。
}cqnode;    //定义类型标识符。
(1) 判空: int Empty(cqnode cq)    //cq是cqnode类型的变量
    {if(cq.count==0) return(1); else return(0); //空队列}
    入队: int EnQueue(cqnode cq, elemtp x)
    {if(count==m){printf("队满\n"); exit(0); }
    cq.q[(cq.front+count)%m]=x; //x入队
    count++; return(1); //队列中元素个数增加1, 入队成功。
    }
    出队: int DelQueue(cqnode cq)
    {if (count==0){printf("队空\n"); return(0);}
    printf("出队元素", cq.q[cq.front]);
    x=cq.q[cq.front];
    cq.front=(cq.front+1)%m; //计算新的队头指针。
    return(x)
    }

```

(2) 队列中能容纳的元素的个数为m。队头指针front指向队头元素。

36、循环队列中元素个数为 (REAR-FRONT+N)%N。其中FRONT是队首指针，指向队首元素的前一位置；REAR是队尾指针，指向队尾元素；N是队列最大长度。

37、循环队列解决了用向量表示队列所出现的“假溢出”问题，但同时又出现了如何判断队列的满与空问题。例如：在队列长10的循环队列中，若假定队头指针front指向队头元素的前一位置，而队尾指针指向队尾元素，则front=3, rear=7的情况下，连续出队4个元素，则front==rear为队空；如果连续入队6个元素，则front==rear为队满。如何判断这种情况下的队满与队空，一般采取牺牲一个单元的做法或设标记法。即假设front==rear为队空，而 (rear+1)%表长==front为队满，或通过设标记tag。若tag=0, front==rear则为队空；若tag=1, 因入队而使front==rear，则为队满。

本题中队尾指针rear，指向队尾元素的下一位置，listarray[rear]表示下一个入队的元素。在这种情况下，我们可规定，队头指针front指向队首元素。当front==rear时为队空，当 (rear+1)%n=front时为队满。出队操作（在队列不空情况下）队头指针是front=(front+1)%n，

38、既不能由输入受限的双端队列得到，也不能由输出受限的双端队列得到的输出序列是dbca。

39、(1) 4132 (2) 4213 (3) 4231

40、(1) 队空的初始条件: f=r=0;

(2) 执行操作 A^3 后, r=3; // A^3 表示三次入队操作

执行操作 D^1 后, f=1; // D^1 表示一次出队操作

执行操作 A^5 后, r=0;

执行操作 D^2 后, $f=3$;

执行操作 A^1 后, $r=1$;

执行操作 D^2 后, $f=5$;

执行操作 A^4 后, 按溢出处理。因为执行 A^3 后, $r=4$, 这时队满, 若再执行A操作, 则出错。

41. 一般说, 高级语言的变量名是以字母开头的字母数字序列。故本题答案是:

AP321, PA321, P3A21, P32A1, P321A。

五、算法设计题

1、[题目分析]两栈共享向量空间, 将两栈栈底设在向量两端, 初始时, s_1 栈顶指针为-1, s_2 栈顶为maxsize。两栈顶指针相邻时为栈满。两栈顶相向, 迎面增长, 栈顶指针指向栈顶元素。

#define maxsize 两栈共享顺序存储空间所能达到的最多元素数

#define elemtp int //假设元素类型为整型

typedef struct

{elemtp stack[maxsize]; //栈空间

int top[2]; //top为两个栈顶指针

}stk;

stk s; //s是如上定义的结构类型变量, 为全局变量。

(1) 入栈操作:

int push(int i, int x)

//入栈操作。i为栈号, $i=0$ 表示左边的栈 s_1 , $i=1$ 表示右边的栈 s_2 , x是入栈元素。入栈成功返回1, 否则返回0。

{if($i<0 || i>1$) {printf(“栈号输入不对”); exit(0);}}

if(s.top[1]-s.top[0]==1) {printf(“栈已满\n”); return(0);}

switch(i)

{case 0: s.stack[++s.top[0]]=x; return(1); break;

case 1: s.stack[--s.top[1]]=x; return(1);

}

}//push

(2) 退栈操作

elemtp pop(int i)

//退栈算法。i代表栈号, $i=0$ 时为 s_1 栈, $i=1$ 时为 s_2 栈。退栈成功返回退栈元素, 否则返回-1。

{if($i<0 || i>1$) {printf(“栈号输入错误\n”); exit(0);}}

switch(i)

{case 0: if(s.top[0]==-1) {printf(“栈空\n”); return(-1); }

else return(s.stack[s.top[0]--]);

case 1: if(s.top[1]==maxsize {printf(“栈空\n”); return(-1);}

else return(s.stack[s.top[1]++]);

}

}//算法结束

[算法讨论] 请注意算法中两栈入栈和退栈时的栈顶指针的计算。两栈共享空间示意图略, s_1 栈是通常意义下的栈, 而 s_2 栈入栈操作时, 其栈顶指针左移(减1), 退栈时, 栈顶指针右移(加1)。

2、#define maxsize 栈空间容量

void InOutS(int s[maxsize])

//s是元素为整数的栈, 本算法进行入栈和退栈操作。

{int top=0; //top为栈顶指针, 定义top=0时为栈空。

for($i=1$; $i \leq n$; $i++$) //n个整数序列作处理。

{scanf(“%d”, &x); //从键盘读入整数序列。

if($x \neq -1$) //读入的整数不等于-1时入栈。

if($top == maxsize - 1$) {printf(“栈满\n”); exit(0);} else s[++top]=x; //x入栈。

else //读入的整数等于-1时退栈。

{if($top == 0$) {printf(“栈空\n”); exit(0);} else printf(“出栈元素是%d\n”, s[top--]); }}

}//算法结束。

3、[题目分析]判断表达式中括号是否匹配, 可通过栈, 简单说是左括号时进栈, 右括号时退栈。退栈时, 若栈顶元素是左括号, 则新读入的右括号与栈顶左括号就可消去。如此下去, 输入表达式结束时, 栈为空则正确, 否则括号不匹配。

int EXYX(char E[], int n)

//E[]是有n字符的字符数组, 存放字符串表达式, 以‘#’结束。本算法判断表达式中圆括号是否匹配。

{char s[30]; //s是一维数组, 容量足够大, 用作存放括号的栈。

int top=0; //top用作栈顶指针。

s[top]=‘#’; //‘#’先入栈, 用于和表达式结束符号‘#’匹配。

int i=0; //字符数组E的工作指针。

```

while(E[i] != '#') //逐字符处理字符表达式的数组。
switch(E[i])
{case '(': s[++top] = '('; i++; break;
case ')': if(s[top] == '(') {top--; i++; break;}
           else {printf("括号不配对"); exit(0);}
case '#': if(s[top] == '#') {printf("括号配对\n"); return (1);}
           else {printf(" 括号不配对\n"); return (0);} //括号不配对
default: i++; //读入其它字符, 不作处理。
}
} //算法结束。

```

[算法讨论]本题是用栈判断括号匹配的特例：只检查圆括号的配对。一般情况是检查花括号（‘{’，‘}’）、方括号（‘[’，‘]’）和圆括号（‘（’，‘）’）的配对问题。编写算法中如遇左括号（‘{’，‘[’，或‘（’）就压入栈中，如遇右括号（‘}’，‘]’，或‘）’），则与栈顶元素比较，如是与其配对的括号（左花括号，左方括号或左圆括号），则弹出栈顶元素；否则，就结论括号不配对。在读入表达式结束符‘#’时，栈中若应只剩‘#’，表示括号全部配对成功；否则表示括号不匹配。

另外，由于本题只是检查括号是否匹配，故对从表达式中读入的不是括号的那些字符，一律未作处理。再有，假设栈容量足够大，因此入栈时未判断溢出。

4、[题目分析]逆波兰表达式(即后缀表达式)求值规则如下：设立运算数栈OPND, 对表达式从左到右扫描(读入)，当表达式中扫描到数时，压入OPND栈。当扫描到运算符时，从OPND退出两个数，进行相应运算，结果再压入OPND栈。这个过程一直进行到读出表达式结束符\$，这时OPND栈中只有一个数，就是结果。

```

float expr()
//从键盘输入逆波兰表达式，以‘$’表示输入结束，本算法求逆波兰式表达式的值。
{float OPND[30]; // OPND是操作数栈。
init(OPND); //两栈初始化。
float num=0.0; //数字初始化。
scanf("%c",&x); //x是字符型变量。
while(x!='$')
{switch
{case '0' <= x <= '9': while((x>='0' && x<='9') || x=='.' ) //拼数
if(x!='.') //处理整数
{num=num*10+(ord(x)-ord('0')); scanf("%c",&x);}
else //处理小数部分。
{scale=10.0; scanf("%c",&x);
while(x>='0' && x<='9')
{num=num+(ord(x)-ord('0')/scale;
scale=scale*10; scanf("%c",&x);}
} //else
push(OPND,num); num=0.0; //数压入栈，下个数字初始化

case x=' ': break; //遇空格，继续读下一个字符。
case x='+': push(OPND, pop(OPND)+pop(OPND)); break;
case x='-': x1=pop(OPND); x2=pop(OPND); push(OPND, x2-x1); break;
case x='*': push(OPND, pop(OPND)*pop(OPND)); break;
case x='/': x1=pop(OPND); x2=pop(OPND); push(OPND, x2/x1); break;
default: //其它符号不作处理。
} //结束switch
scanf("%c",&x); //读入表达式中下一个字符。
} //结束while (x != '$')
printf("后缀表达式的值为%f", pop(OPND));
} //算法结束。

```

[算法讨论]假设输入的后缀表达式是正确的，未作错误检查。算法中拼数部分是核心。若遇到大于等于‘0’且小于等于‘9’的字符，认为是数。这种字符的序号减去字符‘0’的序号得出数。对于整数，每读入一个数字字符，前面得到的部分数要乘上10再加新读入的数得到新的部分数。当读到小数点，认为数的整数部分已完，要接着处理小数部分。小数部分的数要除以10（或10的幂数）变成十分位，百分位，千分位数等等，与前面部分数相加。在拼数过程中，若遇非数字字符，表示数已拼完，将数压入栈中，并且将变量num恢复为0，准备下一个数。这时对新读入的字符进入‘+’、‘-’、‘*’、‘/’及空格的判断，因此在结束处理数字字符的case后，不能加入break语句。

5、（1）A和D是合法序列，B和C 是非法序列。

（2）设被判定的操作序列已存入一维数组A中。

```

int Judge(char A[])
//判断字符数组A中的输入输出序列是否是合法序列。如是，返回true，否则返回false。
{i=0; //i为下标。
j=k=0; //j和k分别为I和字母O的个数。

```

```

while(A[i]!='\0') //当未到字符数组尾就作。
{switch(A[i])
{case 'I' : j++; break; //入栈次数增1。
case 'O' : k++; if(k>j){printf("序列非法\n"); exit(0);}
}
i++; //不论A[i]是'I'或'O',指针i均后移。}
if(j!=k){printf("序列非法\n"); return(false);}
else {printf("序列合法\n"); return(true);}
} //算法结束。

```

[算法讨论]在入栈出栈序列(即由'I'和'O'组成的字符串)的任一位置,入栈次数('I'的个数)都必须大于等于出栈次数(即'O'的个数),否则视作非法序列,立即给出信息,退出算法。整个序列(即读到字符数组中字符串的结束标记'\0'),入栈次数必须等于出栈次数(题目中要求栈的初态和终态都为空),否则视为非法序列。

6、[题目分析]表达式中的括号有以下三对: '('、')'、 '['、']'、 '{'、'}',使用栈,当为左括号时入栈,右括号时,若栈顶是其对应的左括号,则退栈,若不是其对应的左括号,则结论为括号不配对。当表达式结束,若栈为空,则结论表达式括号配对,否则,结论表达式括号不配对。

```

int Match(LinkedList la)
//算术表达式存储在以la为头结点的单循环链表中,本算法判断括号是否正确配对
{char s[]; //s为字符栈,容量足够大
p=la->link; //p为工作指针,指向待处理结点
StackInit(s); //初始化栈s
while (p!=la) //循环到头结点为止
{switch (p->ch)
{case '(' :push(s,p->ch); break;
case ')' :if(StackEmpty(s)||StackGetTop(s)!='(')
{printf("括号不配对\n"); return(0);} else pop(s);break;
case '[' :push(s,p->ch); break;
case ']' : if(StackEmpty(s)||StackGetTop(s)!='[')
{printf("括号不配对\n"); return(0);} else pop(s);break;
case '{' :push(s,p->ch); break;
case '}' : if(StackEmpty(s)||StackGetTop(s)!='{')
{printf("括号不配对\n"); return(0);} else pop(s);break;
} p=p->link; 后移指针
} //while
if (StackEmpty(s)) {printf("括号配对\n"); return(1);}
else {printf("括号不配对\n"); return(0);}
} //算法match结束

```

[算法讨论]算法中对非括号的字符未加讨论。遇到右括号时,若栈空或栈顶元素不是其对应的左圆(方、花)括号,则结论括号不配对,退出运行。最后,若栈不空,仍结论括号不配对。

7、[题目分析]栈的特点是后进先出,队列的特点是先进先出。所以,用两个栈s1和s2模拟一个队列时,s1作输入栈,逐个元素压栈,以此模拟队列元素的入队。当需要出队时,将栈s1退栈并逐个压入栈s2中,s1中最先入栈的元素,在s2中处于栈顶。s2退栈,相当于队列的出队,实现了先进先出。显然,只有栈s2为空且s1也为空,才算是队列空。

```

(1) int enqueue(stack s1,elemtp x)
//s1是容量为n的栈,栈中元素类型是elemtp。本算法将x入栈,若入栈成功返回1,否则返回0。
{if(top1==n && !Semtpy(s2)) //top1是栈s1的栈顶指针,是全局变量。
{printf("栈满");return(0);} //s1满s2非空,这时s1不能再入栈。
if(top1==n && Semtpy(s2)) //若s2为空,先将s1退栈,元素再压栈到s2。
{while(!Semtpy(s1)) {POP(s1,x);PUSH(s2,x);}
PUSH(s1,x); return(1); //x入栈,实现了队列元素的入队。
}
}
(2) void dequeue(stack s2,s1)
//s2是输出栈,本算法将s2栈顶元素退栈,实现队列元素的出队。
{if(!Semtpy(s2)) //栈s2不空,则直接出队。
{POP(s2,x); printf("出队元素为",x); }
else //处理s2空栈。
if(Semtpy(s1)) {printf("队列空");exit(0);} //若输入栈也为空,则判定队空。
else //先将栈s1倒入s2中,再作出队操作。
{while(!Semtpy(s1)) {POP(s1,x);PUSH(s2,x);}
POP(s2,x); //s2退栈相当队列出队。
printf("出队元素",x);
}
}

```

```

    } //结束算法dequeue。
(3) int queue_empty()
    //本算法判用栈s1和s2模拟的队列是否为空。
    {if (Sempty(s1)&&Sempty(s2)) return(1); //队列空。
      else return(0); //队列不空。
    }

```

[算法讨论] 算法中假定栈s1和栈s2容量相同。出队从栈s2出，当s2为空时，若s1不空，则将s1倒入s2再出栈。入队在s1，当s1满后，若s2空，则将s1倒入s2，之后再入队。因此队列的容量为两栈容量之和。元素从栈s1倒入s2，必须在s2空的情况下才能进行，即在要求出队操作时，若s2空，则不论s1元素多少（只要不空），就要全部倒入s2中。

类似本题叙述的其它题的解答：

(1) 该题同上面题本质相同，只有叙述不同，请参考上题答案。

8、[题目分析] 本题要求用链接结构实现一个队列，我们可用链表结构来实现。一般说，由于队列的先进先出性质，所以队列常设队头指针和队尾指针。但题目中仅给出一个“全局指针p”，且要求入队和出队操作的时间复杂性是O(1)，因此我们用只设尾指针的循环链表来实现队列。

```

(1) PROC addq (VAR p:linkedList, x:elemtp);
    //p是数据域为data、链域为link的用循环链表表示的队列的尾指针，本算法是入队操作。
    new(s); //申请新结点。假设有内存空间，否则系统给出出错信息。
    s↑.data:=x; s↑.link:=p↑.link; //将s结点入队。
    p↑.link:=s; p:=s; //尾指针p移至新的队尾。
    ENDP;
(2) PROC deleq (VAR p:linkedList, VAR x:elemtp);
    // p是数据域为data、链域为link的用循环链表表示的队列的尾指针，本算法实现队列元素的出队，若
    出队成功，返回出队元素，否则给出失败信息。
    IF (p↑.link=p) THEN [writeln(“空队列”); return(0);] //带头结点的循环队列。
    ELSE [s:=p↑.link↑.link; //找到队头元素。
          p↑.link↑.link:=s↑.link; //删队头元素。
          x:=s↑.data; //返回出队元素。
          IF (p=s) THEN p:=p↑.link; //队列中只有一个结点，出队后成为空队列。
          dispose(s); //回收出队元素所占存储空间。
        ]
    ENDP;

```

[算法讨论] 上述入队算法中，因链表结构，一般不必考虑空间溢出问题，算法简单。在出队算法中，首先要判断队列是否为空，另外，对出队元素，要判断是否因出队而成为空队列。否则，可能导致因删除出队结点而将尾指针删掉成为“悬挂变量”。

9、本题与上题本质上相同，现用类C语言编写入队和出队算法。

```

(1) void EnQueue (LinkedList rear, ElemType x)
    // rear是带头结点的循环链队列的尾指针，本算法将元素x插入到队尾。
    { s = (LinkedList) malloc (sizeof(LNode)); //申请结点空间
      s->data=x; s->next=rear->next; //将s结点链入队尾
      rear->next=s; rear=s; //rear指向新队尾
    }
(2) void DeQueue (LinkedList rear)
    // rear是带头结点的循环链队列的尾指针，本算法执行出队操作，操作成功输出队头元素；否则给出出错
    信息。
    { if (rear->next==rear) { printf(“队空\n”); exit(0); }
      s=rear->next->next; //s指向队头元素，
      rear->next->next=s->next; //队头元素出队。
      printf(“出队元素是”，s->data);
      if (s==rear) rear=rear->next; //空队列
      free(s);
    }

```

10、[题目分析] 用一维数组 v[0..M-1]实现循环队列，其中M是队列长度。设队头指针 front和队尾指针rear，约定front指向队头元素的前一位置，rear指向队尾元素。定义front=rear时为队空，(rear+1)%m=front 为队满。约定队头端入队向下标小的方向发展，队尾端入队向下标大的方向发展。

(1) #define M 队列可能达到的最大长度

```

typedef struct
{ elemtp data[M];
  int front, rear;
} cycqueue;

```

(2) elemtp delqueue (cycqueue Q)

//Q是如上定义的循环队列，本算法实现从队尾删除，若删除成功，返回被删除元素，否则给出出错信息。

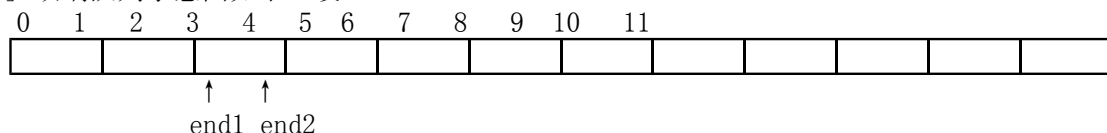
```

{ if (Q.front==Q.rear) {printf(“队列空”); exit(0);}
  Q.rear=(Q.rear-1+M)%M; //修改队尾指针。
  return(Q.data[(Q.rear+1+M)%M]); //返回出队元素。
} //从队尾删除算法结束
void enqueue (cycqueue Q, elemtp x)
// Q是顺序存储的循环队列，本算法实现“从队头插入”元素x。
{if (Q.rear==(Q.front-1+M)%M) {printf(“队满”); exit(0);}
  Q.data[Q.front]=x; //x入队列
  Q.front=(Q.front-1+M)%M; //修改队头指针。
} //结束从队头插入算法。

```

11、参见9。

12、[题目分析] 双端队列示意图如下（设maxsize =12）



用上述一维数组作存储结构，把它看作首尾相接的循环队列。可以在任一端（end1或end2）进行插入或删除。初始状态end1+1=end2被认为是队空状态；end1=end2被认为是队满状态。（左端队列）end1指向队尾元素的前一位置。end2指向（右端队列）队尾元素的后一位置。入队时判队满，出队（删除）时判队空。删除一个元素时，首先查找该元素，然后，从队尾将该元素前的元素依次向后或向前（视end1端或end2端而异）移动。

FUNC add (Qu:deque; var x:datatype;tag 0..1):integer;
 //在双端队列Qu中插入元素x，若插入成功，返回插入元素在Qu中的下标；插入失败返回-1。tag=0表示在end1端插入；tag=1表示在end2端插入。

```

IF Qu.end1=Qu.end2 THEN [writeln(“队满”);return(-1);]
CASE tag OF
  0: //在end1端插入
    [Qu.end1:=x; //插入x
      Qu.end1:=(Qu.end1-1) MOD maxsize; //修改end1
      RETURN(Qu.end1+1) MOD maxsize); //返回插入元素的下标。
  1: //在end2端插入
    [Qu.end2:=x;
      Qu.end2:=(Qu.end2+1) MOD maxsize;
      RETURN(Qu.end2-1) MOD maxsize);
]
ENDC; //结束CASE语句

```

ENDF; //结束算法add

FUNC delete (Qu: deque; VAR x:datatype; tag:0..1):integer;
 //本算法在双端队列Qu中删除元素x，tag=0时从end1端删除，tag=1时从end2端删除。删除成功返回1，否则返回0。

```

IF (Qu.end1+1) MOD maxsize=Qu.end2 THEN [writeln(“队空”);return(0);]
CASE tag OF
  0: //从end1端删除
    [i:=(Qu.end1+1) MOD maxsize; //i是end1端最后插入的元素下标。
      WHILE(i<>Qu.end2) AND (Qu.elem[i]<>x) DO
        i=(i+1) MOD maxsize; //查找被删除元素x的位置
      IF (Qu.elem[i]=x) AND (i<>Qu.end2) THEN
        [ j:=i;
          WHILE((j-1+maxsize) MOD maxsize <>Qu.end1) DO
            [Qu.elem[j]:=Qu.elem[(j-1+maxsize) MOD maxsize];
              j:=(j-1+maxsize) MOD maxsize;
            ] //移动元素，覆盖达到删除
          Qu.end1:=(Qu.end1+1) MOD maxsize; //修改end1指针
          RETURN(1);
        ]
      ELSE RETURN(0);
    ] //结束从end1端删除。
  1: //从end2端删除
    [i:=(Qu.end2-1+maxsize) MOD maxsize; //i是end2端最后插入的元素下标。
      WHILE(i<>Qu.end1) AND (Qu.elem[i]<>x) DO
        i=(i-1+maxsize) MOD maxsize; //查找被删除元素x的下标
      IF (Qu.elem[i]=x) AND (i<>Qu.end1) THEN //被删除元素找到
        [ j:=i;

```

```

    WHILE((j+1) MOD maxsize <> Qu.end2) DO
        [Qu.elem[j]:=Qu.elem[(j+1) MOD maxsize];
        j:=(j+1) MOD maxsize;
        ]//移动元素, 覆盖达到删除
    Qu.end2:=(Qu.end2-1+maxsize) MOD maxsize; //修改end2指针
    RETURN(1); //返回删除成功的信息
]
ELSE RETURN(0); //删除失败
]//结束在end2端删除。
ENDC; //结束CASE语句
ENDF; //结束delete

```

[算法讨论] 请注意下标运算。 $(i+1) \text{ MOD } \text{maxsize}$ 容易理解, 考虑到 $i-1$ 可能为负的情况, 所以求下个 i 时用了 $(i-1+\text{maxsize}) \text{ MOD } \text{maxsize}$ 。

13、[题目分析] 本题与上面12题基本相同, 现用类C语言给出该双端队列的定义。

```

#define maxsize 32
typedef struct
{
    datatype elem[maxsize];
    int end1, end2; //end1和end2取值范围是0..maxsize-1
} deque;

```

14、[题目分析] 根据队列先进先出和栈后进先出的性质, 先将非空队列中的元素出队, 并压入初始为空的栈中。这时栈顶元素是队列中最后出队的元素。然后将栈中元素出栈, 依次插入到初始为空的队列中。栈中第一个退栈的元素成为队列中第一个元素, 最后退栈的元素(出队时第一个元素)成了最后入队的元素, 从而实现了原队列的逆置。

```

void Invert(queue Q)
//Q是一个非空队列, 本算法利用空栈S和已给的几个栈和队列的ADT函数, 将队列Q中的元素逆置。
{
    makempty(S); //置空栈
    while (!isEmpty(Q)) // 队列Q中元素出队
    {
        value=deQueue(Q); push(S, value); } // 将出队元素压入栈中
    while (!isEmpty(S)) //栈中元素退栈
    {
        value=pop(S); enQueue(Q, value); } //将出栈元素入队列 Q
} //算法invert 结束

```

15、为运算方便, 设数组下标从0开始, 即数组 $v[0..m-1]$ 。设每个循环队列长度(容量)为 L , 则循环队列的个数为 $n = \lceil m/L \rceil$ 。为了指示每个循环队列的队头和队尾, 设如下结构类型

```

typedef struct
{
    int f, r;
} scq;
scq q[n];
(1) 初始化的核心语句
for(i=1; i<=n; i++) q[i].f=q[i].r=(i-1)*L; //q[i]是全局变量

```

(2) 入队 `int addq(int i; elemtp x)`
 // n 个循环队列共享数组 $v[0..m-1]$ 和保存各循环队列首尾指针的 $q[n]$ 已经定义为全局变量, 数组元素为 `elemtp` 类型, 本过程将元素插入到第 i 个循环队列中。若入队成功, 返回1, 否则返回队满标记0(入队失败)。

```

{
    if (i<1 || i>n) {printf("队列号错误"); exit(0);}
    if (q[i].r+1)%L+(i-1)*L==q[i].f {printf("队满\n"); exit(0);}
    q[i].r=(q[i].r+1)%L+(i-1)*L; // 计算入队位置
    v[q[i].r]=x; return(1); //元素x入队
}

```

(3) 出队 `int deleteq (int i)`
 // n 个循环队列共享数组 $v[0..m-1]$ 和保存各循环队列首尾指针的 $q[n]$ 已经定义为全局变量, 数组元素为 `elemtp` 类型, 本过程将第 i 个循环队列出队。若出队成功, 打印出队列元素, 并返回1表示成功; 若该循环队列为空, 返回0表示出队失败。

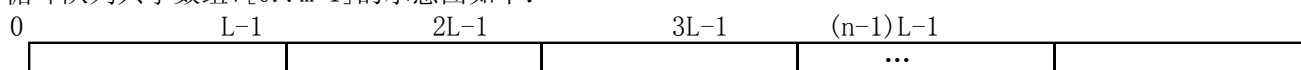
```

{
    if (<1 || >n) {printf("队列号错误\n"); exit(0);}
    if (q[i].r==q[i].f) {printf("队空\n"); return(0);}
    q[i].f=(q[i].f+1)%L+(i-1)*L;
    printf("出队元素", q[i].f); return(1);
}

```

(4) 讨论, 上述算法假定最后一个循环队列的长度也是 L , 否则要对最后一个循环队列作特殊处理。另外, 未讨论一个循环队列满而相邻循环队列不满时, 需修改个循环队列首尾指针的情况(即各循环队列长度不等)。

n 个循环队列共享数组 $v[0..m-1]$ 的示意图如下:



第*i*个循环队列从下标 $(i-1)L$ 开始, 到 $iL-1$ 为止。设每个循环队列均用牺牲一个单元的办法来判断队满, 即为 $(q[i].r+1) \% L + (i-1) * L = q[i].f$ 时, 判定为队满。

16、**int** MaxValue (**int** a[], **int** n)

// 设整数组列存于数组a中, 共有n个, 本算法求解其最大值。

```
{if (n==1) max=a[1];
  else if a[n]>MaxValue(a,n-1) max=a[n];
    else max=MaxValue(a,n-1);
  return(max);
}
```

17、本题与上题类似, 只是这里是同时求n个数中的最大值和最小值的递归算法。

int MinMaxValue(**int** A[], **int** n, **int** *max, **int** *min)

// 一维数组A中存放有n个整型数, 本算法递归的求出其中的最小数。

```
{if (n>0)
  {if(*max<A[n]) *max=A[n];
    if(*min>A[n]) *min=A[n];
    MinMaxValue(A,n-1,max,min);
  } // 算法结束
```

[算法讨论] 调用本算法的格式是 MinMaxValue(arr, n, &max, &min); 其中, arr 是具有n个整数的一维数组, max=-32768 是最大数的初值, min=32767 是最小数的初值。

18、[题目分析] 求两个正整数m和n的最大公因子, 本题叙述的运算方法叫辗转相除法, 也称欧几里德定理。其函数定义为:

$$\text{gcd}(m, n) = \begin{cases} m & \text{若 } n = 0 \\ \text{gcd}(n, m \% n) & \text{若 } n > 0 \end{cases}$$

int gcd (**int** m, **int** n)

// 求正整数m和n的最大公因子的递归算法

```
{if (m<n) return(gcd(n, m)); // 若m<n, 则m和n互换
  if (n==0) return(m); else return(gcd(n, m%n));
} // 算法结束
```

使用栈, 消除递归的非递归算法如下:

int gcd(**int** m, **int** n)

{**int** s[max][2]; // s是栈, 容量max足够大

top=1; s[top][0]=m; s[top][1]=n;

while (s[top][1]!=0)

if (s[top][0]<s[top][1]) // 若m<n, 则交换两数

{t=s[top][0]; s[top][0]=s[top][1]; s[top][1]=t;}

else {t=s[top][0]%s[top][1]; top++; s[top][0]=s[top-1][1]; s[top][1]=t; }

return(s[top][0]);

} // 算法结束

由于是尾递归, 可以不使用栈, 其非递归算法如下

int gcd (**int** m, **int** n)

// 求正整数m和n的最大公因子

{if (m<n) {t=m; m=n; n=t;}} // 若m<n, 则m和n互换

while (n!=0) {t=m; m=n; n=t%n;}

return(m);

} // 算法结束

19、[题目分析] 这是以读入数据的顺序为相反顺序进行累乘问题, 可将读入数据放入栈中, 到输入结束, 将栈中数据退出进行累乘。累乘的初值为1。

PROC test;

CONST maxsize=32;

VAR s:ARRAY[1..maxsize] OF integer, top, sum, a:integer;

[top:=0; sum:=1; //

read(a);

WHILE a<>0 **DO**

[top:=top+1; s[top]:=a; read(a);]

write(sum:5);

WHILE top>0 **DO**

[sum:=sum*s[top]; top:=top-1; write(sum:5);]

ENDP;

20、[题目分析] 本题与第19题基本相同, 不同之处就是求和, 另外用C描述。

int test;

{**int** x, sum=0, top=0, s[];

scanf("%d" , &x)

while (x<>0)

```

        {s[++top]:=a; scanf( "%d" ,&x); }
    printf(sum:5);
    while (top)
        {sum+=s[top--]; printf(sum:5); }
};

```

```

21、int Ack(int m,n)
{if (m==0) return(n+1);
 else if(m!=0&& n==0) return(Ack(m-1,1));
  else return(Ack(m-1,Ack(m,m-1)));
} //算法结束

```

(1) Ack(2,1)的计算过程

```

Ack(2,1)=Ack(1,Ack(2,0))           //因 $m < 0$ ,  $n < 0$ 而得
      =Ack(1,Ack(1,1))             //因 $m < 0$ ,  $n = 0$ 而得
      =Ack(1,Ack(0,Ack(1,0)))       //因 $m < 0$ ,  $n < 0$ 而得
      =Ack(1,Ack(0,Ack(0,1)))       //因 $m < 0$ ,  $n = 0$ 而得
      =Ack(1,Ack(0,2))              //因 $m = 0$ 而得
      =Ack(1,3)                    //因 $m = 0$ 而得
      =Ack(0,Ack(1,2))              //因 $m < 0$ ,  $n < 0$ 而得
      =Ack(0,Ack(0,Ack(1,1)))       //因 $m < 0$ ,  $n < 0$ 而得
      =Ack(0,Ack(0,Ack(0,Ack(1,0)))) //因 $m < 0$ ,  $n < 0$ 而得
      =Ack(0,Ack(0,Ack(0,Ack(0,1)))) //因 $m < 0$ ,  $n = 0$ 而得
      =Ack(0,Ack(0,Ack(0,2)))       //因 $m = 0$ 而得
      =Ack(0,Ack(0,3))              //因 $m = 0$ 而得
      =Ack(0,4)                    //因 $n = 0$ 而得
      =5                           //因 $n = 0$ 而得

```

```

(2) int Ackerman( int m, int n)
{int akm[M][N];int i,j;
 for(j=0;j<N;j++) akm[0][j]=j+1;
 for(i=1;i<m;i++)
 {akm[i][0]=akm[i-1][1];
  for(j=1;j<N;j++)
   akm[i][j]=akm[i-1][akm[i][j-1]];
 }
 return(akm[m][n]);
} //算法结束

```

22、[题目分析]从集合 $(1..n)$ 中选出 k (本题中 $k=2$)个元素，为了避免重复和漏选，可分别求出包括1和不包括1的所有组合。即包括1时，求出集合 $(2..n)$ 中取出 $k-1$ 个元素的所有组合；不包括1时，求出集合 $(2..n)$ 中取出 k 个元素的所有组合。，将这两种情况合到一起，就是题目的解。

```

int A[],n; //设集合已存于数组A中。
void comb(int P[],int i,int k)
//从集合 $(1..n)$ 中选取 $k$  ( $k \leq n$ ) 个元素的所有组合
{if (k==0) printf(P);
 else if(k<=n) {P[i]=A[i]; comb(P,i+1,k-1); comb(P,i+1,k); }
} //算法结束

```

第四章 串

一、选择题

1. B	2. E	3. C	4. A	5. C	6. A	7. 1D	7. 2F	8. B注	9. D	10. B	
------	------	------	------	------	------	-------	-------	-------	------	-------	--

注：子串的定义是：串中任意个连续的字符组成的子序列，并规定空串是任意串的子串，任意串是其自身的子串。若字符串长度为 n ($n>0$)，长为 n 的子串有1个，长为 $n-1$ 的子串有2个，长为 $n-2$ 的子串有3个，……，长为1的子串有 n 个。由于空串是任何串的子串，所以本题的答案为： $8*(8+1)/2+1=37$ 。故选B。但某些教科书上认为“空串是任意串的子串”无意义，所以认为选C。为避免考试中的二意性，编者认为第9题出得好。

二、判断题

1. \checkmark	2. \checkmark	3. \checkmark									
-----------------	-----------------	-----------------	--	--	--	--	--	--	--	--	--

三、填空题

- (1) 由空格字符 (ASCII值32) 所组成的字符串 (2) 空格个数 2. 字符
- 任意个连续的字符组成的子序列 4. 5 5. $0(m+n)$
- 01122312 7. 01010421 8. (1) 模式匹配 (2) 模式串
- (1) 其数据元素都是字符 (2) 顺序存储 (3) 和链式存储 (4) 串的长度相等且两串中对应位置的字符也相等
- 两串的长度相等且两串中对应位置的字符也相等。
- 'xyxyxywyy' 12. $*s++=*t++$ 或 $(*s++=*t++) != '\0'$
- (1) `char s[]` (2) `j++` (3) `i >= j`
- [题目分析] 本题算法采用顺序存储结构求串s和串t的最大公共子串。串s用i指针 ($1 \leq i \leq s.\text{len}$)。t串用j指针 ($1 \leq j \leq t.\text{len}$)。算法思想是对每个i ($1 \leq i \leq s.\text{len}$ ，即程序中第一个WHILE循环)，来求从i开始的连续字符串与从j ($1 \leq j \leq t.\text{len}$ ，即程序中第二个WHILE循环) 开始的连续字符串的最大匹配。程序中第三个 (即最内层) 的WHILE循环，是当s中某字符 (`s[i]`) 与t中某字符 (`t[j]`) 相等时，求出局部公共子串。若该子串长度大于已求出的最长公共子串 (初始为0)，则最长公共子串的长度要修改。

程序 (a)： (1) ($i+k \leq s.\text{len}$) AND ($j+k \leq t.\text{len}$) AND (`s[i+k]=t[j+k]`)
//如果在s和t的长度内，对应字符相等，则指针k 后移 (加1)。

(2) `con:=false` //s和t对应字符不等时置标记退出
(3) `j:=j+k` //在t串中，从第j+k字符再与s[i]比较
(4) `j:=j+1` //t串取下一字符
(5) `i:=i+1` //s串指针i后移 (加1)。

程序 (b)： (1) $i+k \leq s.\text{len}$ && $j+k \leq t.\text{len}$ && `s[i+k]==t[j+k]` //所有注释同上 (a)
(2) `con=0` (3) `j+=k` (4) `j++` (5) `i++`

15. (1) 0 (2) `next[k]`

16. (1) `i:=i+1` (2) `j:=j+1` (3) `i:=i-j+2` (4) `j:=1`; (5) `i=mt` (或 `i:=i-j+1`) (6) 0

17. 程序中递归调用

(1) `ch1<>midch` //当读入不是分隔符&和输入结束符\$时，继续读入字符

(2) `ch1=ch2` //读入分隔符&后，判ch1是否等于ch2，得出真假结论。

(3) `answer:=true`

(4) `answer:=false`

(5) `read(ch)`

(6) `ch=endch`

18. (1) `initstack(s)` // 栈s初始化为空栈。

(2) `setnull(exp)` //串exp初始化为空串。

(3) `ch in opset` //判取出字符是否是操作符。

(4) `push(s,ch)` //如ch是运算符，则入运算符栈s。

(5) `empty(s)` //判栈s是否为空。

(6) `succ:=false` //若读出ch是操作数且栈为空，则按出错处理。

(7) `exp` (8) `ch` //若ch是操作数且栈非空，则形成部分中缀表达式。

(9) `exp` (10) `gettop(s)` //取栈顶操作符。

(11) `pop(s)` //操作符取出后，退栈。

(12) `empty(s)` //将pre的最后一个字符 (操作数) 加入到中缀式exp的最后。

四、应用题

1. 串是零个至多个字符组成的有限序列。从数据结构角度讲，串属于线性结构。与线性表的特殊性在于串的元素是字符。

2. 空格是一个字符，其ASCII码值是32。空格串是由空格组成的串，其长度等于空格的个数。空串是不含任何字符的串，即空串的长度是零。

3. 最优的 $T(m, n)$ 是 $O(n)$ 。串S2是串S1的子串，且在S1中的位置是1。开始求出最大公共子串的长度恰是串S2的长度，一般情况下， $T(m, n) = O(m*n)$ 。

4. 朴素的模式匹配 (Brute-Force) 时间复杂度是 $O(m*n)$ ，KMP算法有一定改进，时间复杂度达到 O

($m+n$)。本题也可采用从后面匹配的方法,即从右向左扫描,比较6次成功。另一种匹配方式是从左往右扫描,但是先比较模式串的最后一个字符,若不等,则模式串后移;若相等,再比较模式串的第一个字符,若第一个字符也相等,则从模式串的第二个字符开始,向右比较,直至相等或失败。若失败,模式串后移,再重复以上过程。按这种方法,本题比较18次成功。

5. KMP算法主要优点是主串指针不回溯。当主串很大不能一次读入内存且经常发生部分匹配时,KMP算法的优点更为突出。

6. 模式串的next函数定义如下:

$$\text{next}[j] = \begin{cases} 0 & \text{当 } j = 1 \text{ 时} \\ \max\{k \mid 1 < k < j \text{ 且 } 'p_1 \dots p_{k-1}' = 'p_{j-k+1} \dots p_{j-1}'\} & \text{当此集合不空时} \\ 1 & \text{其它情况} \end{cases}$$

根据此定义,可求解模式串t的next和nextval值如下:

j	1	2	3	4	5	6	7	8	9	10	11	12
t串	a	b	c	a	a	b	b	a	b	c	a	b
next[j]	0	1	1	1	2	2	3	1	2	3	4	5
nextval[j]	0	1	1	0	2	1	3	0	1	1	0	5

7. 解法同上题6,其next和nextval值分别为0112123422和0102010422。

8. 解法同题6,t串的next和nextval函数值分别为0111232和0110132。

9. 解法同题6,其next和nextval值分别为011123121231和011013020131。

10. p1的next和nextval值分别为:0112234和0102102;p2的next和nextval值分别为:0121123和0021002。

11. next数组值为011234567改进后的next数组信息值为010101017。

12. 011122312。

13. next定义见题上面6和下面题20。串p的next函数值为:01212345634。

14. (1) S的next与nextval值分别为012123456789和002002002009,p的next与nextval值分别为012123和002003。

(2) 利用BF算法的匹配过程:

第一趟匹配: aabaabaabaac

aabaac(i=6,j=6)

第二趟匹配: aabaabaabaac

aa(i=3,j=2)

第三趟匹配: aabaabaabaac

a(i=3,j=1)

第四趟匹配: aabaabaabaac

aabaac(i=9,j=6)

第五趟匹配: aabaabaabaac

aa(i=6,j=2)

第六趟匹配: aabaabaabaac

a(i=6,j=1)

第七趟匹配: aabaabaabaac

(成功)

aabaac(i=13,j=7)

利用KMP算法的匹配过程:

第一趟匹配: aabaabaabaac

aabaac(i=6,j=6)

第二趟匹配: aabaabaabaac

(aa)baac

第三趟匹配: aabaabaabaac

(成功)(aa)baac

15. (1) p的nextval函数值为0110132。(p的next函数值为0111232)。

(2) 利用KMP(改进的nextval)算法,每趟匹配过程如下:

第一趟匹配: abcaabbabcabaacbacba

abcab(i=5,j=5)

第二趟匹配: abcaabbabcabaacbacba

abc(i=7,j=3)

第三趟匹配: abcaabbabcabaacbacba

a(i=7,j=1)

第四趟匹配: abcaabbabcabaac bacba

(成功)

abcabaa(i=15,j=8)

16. KMP算法的时间复杂度是 $O(m+n)$ 。

p的next和nextval值分别为01112212321和01102201320。

17. (1) p的nextval函数值为01010。(next函数值为01123)

(2) 利用所得nextval数值,手工模拟对s的匹配过程,与上面16题类似,为节省篇幅,故略去。

18. 模式串T的next和nextval值分别为0121123和0021002。

19. 第4行的 $p[J]=p[K]$ 语句是测试模式串的第J个字符是否等于第K个字符,如是,则指针J和K均增加1,继续比较。第6行的 $p[J]=p[K]$ 语句的意义是,当第J个字符在模式匹配中失配时,若第K个字符和第J个字符不等,则下个与主串匹配的字符是第K个字符;否则,若第K个字符和第J个字符相等,则下个与主串匹配的字符是第K个字符失配时的下一个(即NEXTVAL[K])。

该算法在最坏情况下的时间复杂度 $O(m^2)$ 。

20. (1) 当模式串中第一个字符与主串中某字符比较不等(失配)时, $\text{next}[1]=0$ 表示模式串中已没有字符可与主串中当前字符 $s[i]$ 比较, 主串当前指针应后移至下一字符, 再和模式串中第一字符进行比较。

(2) 当主串第 i 个字符与模式串中第 j 个字符失配时, 若主串 i 不回溯, 则假定模式串第 k 个字符与主串第 i 个字符比较, k 值应满足条件 $1 < k < j$ 并且 ' $p_1 \cdots p_{k-1}$ ' = ' $p_{j-k+1} \cdots p_{j-1}$ ', 即 k 为模式串向后移动的距离, k 值有多个, 为了不使向右移动丢失可能的匹配, k 要取大, 由于 $\max\{k\}$ 表示移动的最大距离, 所以取 $\max\{k\}$, k 的最大值为 $j-1$ 。

(3) 在上面两种情况外, 发生失配时, 主串指针 i 不回溯, 在最坏情况下, 模式串从第1个字符开始与主串第 i 个字符比较, 以便不致丢失可能的匹配。

21. 这里失败函数 f , 即是通常讲的模式串的 next 函数, 其定义见本章应用题的第6题。

进行模式匹配时, 若主串第 i 个字符与模式串第 j 个字符发生失配, 主串指针 i 不回溯, 和主串第 i 个字符进行比较的是模式串的第 $\text{next}[j]$ 个字符。模式串的 next 函数值, 只依赖于模式串, 和主串无关, 可以预先求出。

该算法的技术特点是主串指针 i 不回溯。在经常发生“部分匹配”和主串很大不能一次调入内存时, 优点特别突出。

22. 失败函数(即 next)的值只取决于模式串自身, 若第 j 个字符与主串第 i 个字符失配时, 假定主串不回溯, 模式串用第 k (即 $\text{next}[j]$)个字符与第 i 个相比, 有 ' $p_1 \cdots p_{k-1}$ ' = ' $p_{j-k+1} \cdots p_{j-1}$ ', 为了不因模式串右移与主串第 i 个字符比较而丢失可能的匹配, 对于上式中存在的多个 k 值, 应取其中最大的一个。这样, 因 $j-k$ 最小, 即模式串向右滑动的位数最小, 避免因右移造成的可能匹配的丢失。

23. 仅从两串含有相等的字符, 不能判定两串是否相等, 两串相等的充分必要条件是两串长度相等且对应位置上的字符相同(即两串串值相等)。

24. (1) s_1 和 s_2 均为空串; (2) 两串之一为空串; (3) 两串串值相等(即两串长度相等且对应位置上的字符相同)。(4) 两串中一个串长是另一个串长(包括串长为1仅有一个字符的情况)的数倍, 而且长串就好像是由数个短串经过连接操作得到的。

25、题中所给操作的含义如下:

//: 连接函数, 将两个串连接成一个串

$\text{substr}(s, i, j)$: 取子串函数, 从串 s 的第 i 个字符开始, 取连续 j 个字符形成子串

$\text{replace}(s_1, i, j, s_2)$: 置换函数, 用 s_2 串替换 s_1 串中从第 i 个字符开始的连续 j 个字符

本题有多种解法, 下面是其中的一种:

- (1) $s_1 = \text{substr}(s, 3, 1)$ //取出字符: 'y'
- (2) $s_2 = \text{substr}(s, 6, 1)$ //取出字符: '+'
- (3) $s_3 = \text{substr}(s, 1, 5)$ //取出子串: '(xyz)'
- (4) $s_4 = \text{substr}(s, 7, 1)$ //取出字符: '*'
- (5) $s_5 = \text{replace}(s_3, 3, 1, s_2)$ //形成部分串: '(x+z)'
- (6) $s = s_5 // s_4 // s_1$ //形成串 t 即 '(x+z)*y'

五、算法设计

1、[题目分析]判断字符串 t 是否是字符串 s 的子串, 称为串的模式匹配, 其基本思想是对串 s 和 t 各设一个指针 i 和 j , i 的值域是 $0..m-n$, j 的值域是 $0..n-1$ 。初始值 i 和 j 均为0。模式匹配从 s_0 和 t_0 开始, 若 $s_0=t_0$, 则 i 和 j 指针增加1, 若在某个位置 $s_i \neq t_j$, 则主串指针 i 回溯到 $i=i-j+1$, j 仍从0开始, 进行下一轮的比较, 直到匹配成功($j>n-1$), 返回子串在主串的位置($i-j$)。否则, 当 $i>m-n$ 则为匹配失败。

```
int index(char s[], t[], int m, n)
```

//字符串 s 和 t 用一维数组存储, 其长度分别为 m 和 n 。本算法求字符串 t 在字符串 s 中的第一次出现, 如是, 输出子串在 s 中的位置, 否则输出0。

```
{int i=0, j=0;
while (i<=m-n && j<=n-1)
    if (s[i]==t[j]) {i++;j++;} //对应字符相等, 指针后移。
    else {i=i-j+1;j=0;} //对应字符不相等, I回溯, j仍为0。
    if (i<=m-n && j==n) {printf("t在s串中位置是%d", i-n+1);return(i-n+1);} //匹配成功
    else return(0); //匹配失败
} //算法index结束
```

```
main () //主函数
```

```
{char s[], t[]; int m, n, i;
scanf("%d%d", &m, &n); //输入两字符串的长度
scanf("%s", s); //输入主串
scanf("%s", t); //输入子串
i=index(s, t, m, n);
} //程序结束
```

[程序讨论]因用C语言实现, 一维数组的下标从0开始, $m-1$ 是主串最后一个字符的下标, $n-1$ 是 t 串的最后字符的下标。若匹配成功, 最佳情况是 s 串的第0到第 $n-1$ 个字符与 t 匹配, 时间复杂度为 $O(n)$; 匹配成功的最差情况是, 每次均在 t 的最后一个字符才失败, 直到 s 串的第 $m-n$ 个字符成功, 其时间复杂度为 $O((m-n)*n)$, 即 $O(m*n)$ 。失败的情况是 s 串的第 $m-n$ 个字符比 t 串某字符比较失败, 时间复杂度为 $O(m*n)$ 。之所以串 s 的指针 i 最大到 $m-n$, 是因为在 $m-n$ 之后, 所剩子串长度已经小于子串长度 n , 故不必再去比较。算法中未讨论输

入错误 (如s串长小于t串长)。

另外, 根据子串的定义, 返回值 $i-n+1$ 是子串在主串中的位置, 子串在主串中的下标是 $i-n$ 。

2. [问题分析] 在一个字符串内, 统计含多少整数的问题, 核心是如何将数从字符串中分离出来。从左到右扫描字符串, 初次碰到数字字符时, 作为一个整数的开始。然后进行拼数, 即将连续出现的数字字符拼成一个整数, 直到碰到非数字字符为止, 一个整数拼完, 存入数组, 再准备下一整数, 如此下去, 直至整个字符串扫描到结束。

```
int CountInt ()
```

```
// 从键盘输入字符串, 连续的数字字符算作一个整数, 统计其中整数的个数。
```

```
{int i=0, a[]; // 整数存储到数组a, i记整数个数
```

```
scanf ( "%c" , &ch); // 从左到右读入字符串
```

```
while (ch!= '#' ) // '#' 是字符串结束标记
```

```
if (isdigit (ch)) // 是数字字符
```

```
{num=0; // 数初始化
```

```
while (isdigit (ch) && ch!= '#' ) // 拼数
```

```
{num=num*10+ 'ch' - '0' ;
```

```
scanf ( "%c" , &ch);
```

```
}
```

```
a[i]=num; i++;
```

```
if (ch!= '#' ) scanf ( "%c" , &ch); // 若拼数中输入了 '#' , 则不再输入
```

```
} // 结束while (ch!= '#' )
```

```
printf ( "共有%d个整数, 它们是: " i);
```

```
for (j=0; j<i; j++)
```

```
{printf ( "%6d" , a[j]);
```

```
if ( (j+1) %10==0) printf ( "\n" ); } // 每10个数输出在一行上
```

```
} // 算法结束
```

[算法讨论] 假定字符串中的数均不超过32767, 否则, 需用长整型数组及变量。

3. [题目分析] 设以字符数组s表示串, 重复子串的含义是由一个或多个连续相等的字符组成的子串, 其长度用max表示, 初始长度为0, 将每个局部重复子串的长度与max相比, 若比max大, 则需要更新max, 并用index记住其开始位置。

```
int LongestString(char s[], int n)
```

```
//串用一维数组s存储, 长度为n, 本算法求最长重复子串, 返回其长度。
```

```
{int index=0, max=0; //index记最长的串在s串中的开始位置, max记其长度
```

```
int length=1, i=0, start=0; //length记局部重复子串长度, i为字符数组下标
```

```
while(i<n-1)
```

```
if(s[i]==s[i+1]) {i++; length++;}
```

```
else //上一个重复子串结束
```

```
{if(max<length) {max=length; index=start; } //当前重复子串长度大, 则更新max
```

```
i++;start=i;length=1; //初始化下一重复子串的起始位置和长度
```

```
}
```

```
printf( "最长重复子串的长度为%d, 在串中的位置%d\n" , max, index);
```

```
return(max);
```

```
} //算法结束
```

[算法讨论] 算法中用 $i < n-1$ 来控制循环次数, 因C数组下标从0 开始, 故长度为n的串, 其最后一个字符下标是 $n-1$, 当 i 最大为 $n-2$ 时, 条件语句中 $s[i+1]$ 正好是 $s[n-1]$, 即最后一个字符。子串长度的初值数为1, 表示一个字符自然等于其身。

算法的时间复杂度为 $O(n)$, 每个字符与其后继比较一次。

4. [题目分析] 教材中介绍的串置换有两种形式: 第一种形式是 $\text{replace}(s, i, j, t)$, 含义是将s串中从第 i 个字符开始的 j 个字符用t串替换, 第二种形式是 $\text{replace}(s, t, v)$, 含义是将s串中所有非重叠的t串用v代替。我们先讨论第一种形式的替换。因为已经给定顺序存储结构, 我们可将s串从第 $(i+j-1)$ 到串尾 (即 $s.\text{curlen}$) 移动 $t.\text{curlen}-j$ 绝对值个位置 (以便将t串插入): 若 $j > t.\text{curlen}$, 则向左移; 若 $j < t.\text{curlen}$, 则向右移动; 若 $j = t.\text{curlen}$, 则不必移动。最后将t串复制到s串的合适位置上。当然, 应考虑置换后的溢出问题。

```
int replace(strtp s, t, int i, j)
```

```
//s和t是用一维数组存储的串, 本算法将s串从第i个字符开始的连续j个字符用t串置换, 操作成功返回1, 否则返回0表示失败。
```

```
{if(i<1 || j<0 || t.curlen+s.curlen-j>maxlen)
```

```
{printf ( "参数错误\n" );exit(0);} //检查参数及置换后的长度的合法性。
```

```
if(j<t.curlen) //若s串被替换的子串长度小于t串长度, 则s串部分右移,
```

```
for(k=s.curlen-1;k>=i+j-1;k--) s.ch[k+t.curlen-j]=s.ch[k];
```

```
else if (j>t.curlen) //s串中被替换子串的长度小于t串的长度。
```

```
for(k=i-1+j;k<=s.curlen-1;k++) s.ch[k-(j-t.curlen)]=s.ch[k];
```

```
for(k=0;k<t.curlen;k++) s.ch[i-1+k]=t.ch[k]; //将t串复制到s串的适当位置
```

```
if(j>t.curlen) s.curlen=s.curlen-(j-t.curlen);else s.curlen=s.curlen+(t.curlen-j);
```

}//算法结束

[算法讨论]若允许使用另一数组,在检查合法性后,可将s的第i个(不包括i)之前的子串复制到另一子串如s1中,再将t串接到s1串后面,然后将s的第i+j直到尾的部分加到s1之后。最后将s1串复制到s。主要语句有:

```
for(k=0;k<i;k++) s1.ch[k]=s.ch[k]; //将s1第i个字符前的子串复制到s1,这时k=i-1
for(k=0;k<t.curlen;k++) s1.ch[i+k]=t.ch[k]//将t串接到s1的尾部
l=s.curlen+t.curlen-j-1;
for(k=s.curlen-1;k>i-1+j;k--); //将子串第i+j-1个字符以后的子串复制到s1
s1.ch[l--]=s.ch[k]
for(k=0;k<s.curlen+t.curlen-j;k++) s.ch[k]=s1.ch[k]; //将结果串放入s。
```

下面讨论replace(s,t,v)的算法。该操作的意义是用串v替换所有在串s中出现的和非空串t相等的不重叠的子串。本算法不指定存储结构,只使用串的基本运算。

```
void replace(string s,t,v)
//本算法是串的置换操作,将串s中所有非空串t相等且不重复的子串用v代替。
{i=index(s,t); //判断s是否有和t相等的子串
if(i!=0) //串s中包含和t相等的子串
{creat(temp,""); //creat操作是将串常量(此处为空串)赋值给temp。
m=length(t);n=length(s); //求串t和s的长度
while(i!=0)
{assign(temp,concat(temp,substr(s,l,i-1),v)); //用串v替换t形成部分结果
assign(s,substr(s,i+m,n-i-m+1)); //将串s中串后的部分形成新的s串
n=n-(i-1)-m; //求串s的长度
i=index(s,t); //在新串s中再找串t的位置
}
assign(s,contact(temp,s)); //将串temp和剩余的串s连接后再赋值给s
} //if结束
} //算法结束
```

5、[题目分析]本题是字符串的插入问题,要求在字符串s的pos位置,插入字符串t。首先应查找字符串s的pos位置,将第pos个字符到字符串s尾的子串向后移动字符串t的长度,然后将字符串t复制到字符串s的第pos位置后。

对插入位置pos要验证其合法性,小于1或大于串s的长度均为非法,因题目假设给字符串s的空间足够大,故对插入不必判溢出。

```
void insert(char *s,char *t,int pos)
//将字符串t插入字符串s的第pos个位置。
{int i=1,x=0; char *p=s,*q=t; //p,q分别为字符串s和t的工作指针
if(pos<1) {printf("pos参数位置非法\n");exit(0);}
while(*p!='\0' && i<pos) {p++;i++;} //查pos位置
//若pos小于串s长度,则查到pos位置时,i=pos。
if(*p=='\0') {printf("%d位置大于字符串s的长度",pos);exit(0);}
else //查找字符串的尾
while(*p!='\0') {p++;i++;} //查到尾时,i为字符'\0'的下标,p也指向'\0'。
while(*q!='\0') {q++;x++;} //查找字符串t的长度x,循环结束时q指向'\0'。
for(j=i;j>=pos;j--) {*(p+x)=*p; p--;} //串s的pos后的子串右移,空出串t的位置。
q--; //指针q回退到串t的最后一个字符
for(j=1;j<=x;j++) *p--=*q--; //将t串插入到s的pos位置上
```

[算法讨论]串s的结束标记('\0')也后移了,而串t的结尾标记不应插入到s中。

6、[题目分析]本题属于查找,待查找元素是字符串(长4),将查找元素存放在一维数组中。二分检索(即折半查找或对分查找),是首先用一维数组的“中间”元素与被检索元素比较,若相等,则检索成功,否则,根据被检索元素大于或小于中间元素,而在中间元素的右方或左方继续查找,直到检索成功或失败(被检索区间的低端指针大于高端指针)。下面给出类C语言的解法

```
typedef struct node
{char data[4]; //字符串长4
}node;
非递归过程如下:
int binsearch(node string [],int n;char name[4])
//在有n个字符串的数组string中,二分检索字符串name。若检索成功,返回name在string中的下标,否则返回-1。
{int low = 0,high = n - 1; //low和high分别是检索区间的下界和上界
while(low <= high)
{mid = (low + high) / 2; //取中间位置
if(strcmp(string[mid],name) == 0) return (mid); //检索成功
else if(strcmp(string[mid],name) < 0) low=mid+1; //到右半部分检索
else high=mid-1; //到左半部分检索
```

```

    }
    return 0; //检索失败
} //算法结束

```

最大检索长度为 $\log_2 n$ 。

7. [题目分析] 设字符串存于字符数组X中，若转换后的数是负数，字符串的第一个字符必为‘-’，取出的数字字符，通过减去字符零（‘0’）的ASCII值，变成数，先前取出的数乘上10加上本次转换的数形成部分数，直到字符串结束，得到结果。

```

long atoi(char X[])
//一数字字符串存于字符数组X中，本算法将其转换成数
{
    long num=0;
    int i=1; //i 为数组下标
    while (X[i]!='\0') num=10*num+(X[i++]-'0'); //当字符串未到尾，进行数的转换
    if(X[0]=='-') return (-num); //返回负数
    else return ((X[0]-'0')*10+num); //返回正数，第一位若不是负号，则是数字
} //算法atoi结束

```

[算法讨论] 如是负数，其符号位必在前面，即字符数组的x[0]，所以在作转换成数时下标i从1 开始，数字字符转换成数使用X[i]-‘0’，即字符与‘0’的ASCII值相减。请注意对返回正整数的处理。

8. [题目分析] 本题要求字符串s1拆分成字符串s2和字符串s3，要求字符串s2 “按给定长度n格式化成两端对齐的字符串”，即长度为n且首尾字符不得为空格字符。算法从左到右扫描字符串s1，找到第一个非空格字符，计数到n，第n个拷入字符串s2的字符不得为空格，然后将余下字符复制到字符串s3中。

```

void format (char *s1,*s2,*s3)
//将字符串s1拆分成字符串s2和字符串s3，要求字符串s2是长n且两端对齐
{
    char *p=s1, *q=s2;
    int i=0;
    while(*p!='\0' && *p==' ') p++; //滤掉s1左端空格
    if(*p=='\0') {printf("字符串s1为空串或空格串\n"); exit(0); }
    while( *p!='\0' && i<n) { *q=*p; q++; p++; i++; } //字符串s1向字符串s2中复制
    if(*p=='\0') { printf("字符串s1没有%d个有效字符\n",n); exit(0); }
    if(*(--q)==' ') //若最后一个字符为空格，则需向后找到第一个非空格字符
    {
        p--; //p指针也后退
        while(*p==' ' && *p!='\0') p++; //往后查找一个非空格字符作串s2的尾字符
        if(*p=='\0') { printf("s1串没有%d个两端对齐的字符串\n",n); exit(0); }
        *q=*p; //字符串s2最后一个非空字符
        *(++q)='\0'; //置s2字符串结束标记
    }
    *q=s3;p++; //将s1串其余部分送字符串s3。
    while (*p!='\0') { *q=*p; q++; p++; }
    *q='\0'; //置串s3结束标记
}

```

9. [题目分析] 两个串的相等，其定义为两个串的值相等，即串长相等，且对应字符相等是两个串相等的充分必要条件。因此，首先比较串长，在串长相等的前提下，再比较对应字符是否相等。

```

int equal(strtp s, strtp t)
//本算法判断字符串s和字符串t是否相等，如相等返回1，否则返回0
{
    if (s.curlen!=t.curlen) return (0);
    for (i=0; i<s.curlen;i++) //在类C中，一维数组下标从零开始
        if (s.ch[i]!= t.ch[i]) return (0);
    return (1); //两串相等
} //算法结束

```

10. [问题分析] 由于字母共26个，加上数字符号10个共36个，所以设一长36的整型数组，前10个分量存放数字字符出现的次数，余下存放字母出现的次数。从字符串中读出数字字符时，字符的ASCII代码值减去数字字符‘0’的ASCII代码值，得出其数值(0..9)，字母的ASCII代码值减去字符‘A’的ASCII代码值加上10，存入其数组的对应下标分量中。遇其它符号不作处理，直至输入字符串结束。

```

void Count ()
//统计输入字符串中数字字符和字母字符的个数。
{
    int i, num[36];
    char ch;
    for (i=0; i<36; i++) num[i]=0; // 初始化
    while ( (ch=getchar()) != '#' ) // '#' 表示输入字符串结束。
    {
        if ( '0' <=ch<= '9' ) { i=ch-48; num[i]++; } // 数字字符
        else if ( 'A' <=ch<= 'Z' ) { i=ch-65+10; num[i]++; } // 字母字符
    }

    for (i=0; i<10; i++) // 输出数字字符的个数

```

```

    printf ( “数字%d的个数=%d\n” , i, num[i] );
    for ( i=10; i<36; i++) // 求出字母字符的个数
        printf ( “字母字符%c的个数=%d\n” , i+55, num[i] );
} // 算法结束。

```

11. [题目分析]实现字符串的逆置并不难,但本题“要求不另设串存储空间”来实现字符串逆序存储,即第一个输入的字符最后存储,最后输入的字符先存储,使用递归可容易做到。

```

void InvertStore(char A[])
//字符串逆序存储的递归算法。
{ char ch;
  static int i = 0; //需要使用静态变量
  scanf ("%c",&ch);
  if (ch!= '.') //规定'.'是字符串输入结束标志
  { InvertStore(A);
    A[i++] = ch; //字符串逆序存储
  }
  A[i] = '\0'; //字符串结尾标记
} //结束算法InvertStore。

```

12. 串s'''可以看作由以下两部分组成:'caabcbca...a'和'ca...a',设这两部分分别叫串s1和串s2,要设法从s,s'和s''中得到这两部分,然后使用联接操作联接s1和s2得到s'''。

```

i=index(s,s'); //利用串s'求串s1在串s中的起始位置
s1=substr(s,i,length(s) - i + 1); //取出串s1
j=index(s,s''); //求串s''在串s中的起始位置,s串中'bc'b'后是'ca...a')
s2=substr(s,j+3,length(s) - j - 2); //形成串s2
s3=concat(s1,s2);

```

13. [题目分析]对读入的字符串的第奇数个字符,直接放在数组前面,对第偶数个字符,先入栈,到读字符串结束,再将栈中字符出栈,送入数组中。限于篇幅,这里编写算法,未编程序。

```

void RearrangeString()
//对字符串改造,将第偶数个字符放在串的后半部分,第奇数个字符前半部分。
{ char ch,s[],stk[]; //s和stk是字符数组(表示字符串)和字符栈
  int i=1,j; //i和j字符串和字符栈指针
  while((ch=getchar())!='#') // '#'是字符串结束标志
  { s[i++]=ch; //读入字符串
    s[i]='\0'; //字符数组中字符串结束标志
    i=1;j=1;
    while(s[i]) //改造字符串
    { if(i%2==0) stk[i/2]=s[i]; else s[j++]=s[i];
      i++; } //while
    i--; i=i/2; //i先从'\0'后退,是第偶数字符的个数
    while(i>0) s[j++]=stk[i--] //将第偶数个字符逆序填入原字符数组
  }
}

```

14. [题目分析]本题是对字符串表达式的处理问题,首先定义4种数据结构:符号的类码,符号的TOKEN表示,变量名表NAMEL和常量表CONSL。这四种数据结构均定义成结构体形式,数据部分用一维数组存储,同时用指针指出数据的个数。算法思想是从左到右扫描表达式,对读出的字符,先查出其符号类码:若是变量或常量,就到变量名表和常量表中去查是否已有,若无,则在相应表中增加之,并返回该字符在变量名表或常量表中的下标;若是操作符,则去查其符号类码。对读出的每个符号,均填写其TOKEN表。如此下去,直到表达式处理完毕。先定义各数据结构如下。

```

struct // 定义符号类别数据结构
{ char data[7]; //符号
  char code[7]; //符号类码
} TYPL;
typedef struct //定义TOKEN的元素
{ int typ; //符号码
  int addr; //变量、常量在名字表中的地址
} cmp;
struct { cmp data[50]; //定义TOKEN表长度<50
  int last; //表达式元素个数
} TOKEN;
struct { char data[15]; //设变量个数小于15个
  int last; //名字表变量个数
} NAMEL;
struct { char data[15]; //设常量个数小于15个
  int last; //常量个数
} CONSL;

```

```

    }CONSL;
int operator (char cr)
//查符号在类码表中的序号
{for (i=3; i<=6; i++)
    if (TYPL.data[i]==cr) return (i);
}
void PROCeString ()
//从键盘读入字符串表达式 (以 ‘#’ 结束), 输出其TOKEN表示。
{NAMEL.last=CONSL.last=0; //各表元素个数初始化为0
TYPL.data[3]= ‘*’ ; TYPL.data[4]= ‘+’ ; TYPL.data[5]= ‘(’ ;
TYPL.data[6]= ‘)’ ; //将操作符存入数组
TYPL.code[3]= ‘3’ ; TYPL.code[4]= ‘4’ ; TYPL.code[5]= ‘5’ ;
TYPL.code[6]= ‘6’ ; //将符号的类码存入数组
scanf ( “%c”, &ch); //从左到右扫描 (读入) 表达式。
while (ch!= ‘#’) // ‘#’ 是表达式结束符
{switch (ch) of
    {case ‘A’ : case ‘B’ : case ‘C’ : //ch是变量
        TY=0; //变量类码为0
        for (i=1; i<=NAMEL.last; i++)
            if (NAMEL.data[i]==ch) break; //已有该变量, i记住其位置
            if (i>NAMEL.last) {NAMEL.data[i]=ch; NAMEL.last++; } //变量加入
        case ‘0’ : case ‘1’ : case ‘2’ : case ‘3’ : case ‘4’ : case ‘5’ : //处理常量
        case ‘6’ : case ‘7’ : case ‘8’ : case ‘9’ : TY=1; //常量类码为1
        for (i=1; i<=CONSL.last; i++)
            if (CONSL.data[i]==ch) break; ////已有该常量, i记住其位置
            if (i>CONSL.last) {CONSL.data[i]=ch; CONSL.last++; } //将新常量加入
        default: //处理运算符
            TY=operator (ch); //类码序号
            i= ‘\0’ ; //填入TOKEN的addr域 (期望输出空白)
        } //结束switch, 下面将ch填入TOKEN表
        TOKEN.data [++TOKEN.last].typ=TY; TOKEN.data [TOKEN.last].addr=i;
        scanf ( “%c”, &ch); //读入表达式的下一符号。
    } //while
} //算法结束

```

[程序讨论]为便于讨论, 各一维数组下标均以1开始, 在字符为变量或常量的情况下, 将其类码用TY记下, 用i记下其NAMEL表或CONSL表中的位置, 以便在填TOKEN表时用。在运算符 (‘+’, ‘*’, ‘(’, ‘)’) 填入TOKEN表时, TOKEN表的addr域没意义, 为了程序统一, 这里填入了 ‘\0’。本题是表达式处理的简化情况 (只有3个单字母变量, 常量只有0..9, 操作符只4个), 若是真实情况, 所用数据结构要相应变化。

第五章 数组和广义表

一、选择题

1. B	2. 1L	2. 2J	2. 3C	2. 4I	2. 5C	3. B	4. B	5. A	6. 1H	6. 2C	6. 3E
6. 4A	6. 5F	7. B	8. 1E	8. 2A	8. 3B	9. B	10. B	11. B	12. B	13. A	14. B
15. B	16. A	17. C	18. D	19. C	20. D	21. F	22. C	23. D	24. C	25. A	26. C
27. A											

二、判断题

1. ×	2. √	3. √	4. ×	5. ×	6. ×	7. √	8. ×	9. ×	10. ×	11. ×	12. √
13. √	14. √										

部分答案解释如下。

- 错误。对于完全二叉树，用一维数组作存储结构是效率高的（存储密度大）。
- 错误。数组是具有相同性质的数据元素的集合，数据元素不仅有值，还有下标。因此，可以说数组是元素值和下标构成的偶对的有穷集合。
- 错误。数组在维数和界偶确定后，其元素个数已经确定，不能进行插入和删除运算。
- 错误。稀疏矩阵转置后，除行列下标及行列数互换外，还必须确定该元素转置后在新三元组中的位置。
- 错误。广义表的取表尾运算，是非空广义表除去表头元素，剩余元素组成的表，不可能是原子。
- 错误。广义表的表头就是广义表的第一个元素。只有非空广义表才能取表头。
- 错误。广义表中元素可以是原子，也可以是表（包括空表和非空表）。
- 错误。广义表的表尾，指去掉表头元素后，剩余元素所组成的表。

三、填空题

- 顺序存储结构
- (1) 9572 (2) 1228
- (1) 9174 (2) 8788
- 1100
- 1164 公式： $LOC(a_{ijk}) = LOC(a_{000}) + [v_2 * v_3 * (i - c_1) + v_3 * (j - c_2) + (k - c_3)] * 1$ (1为每个元素所占单元数)
- 232
- 1340
- 1196
- 第1行第3列
- (1) 270 (2) 27 (3) 2204
- $i(i-1)/2 + j$ ($1 \leq i, j \leq n$)
- (1) $n(n+1)/2$ (2) $i(i+1)/2$ (或 $j(j+1)/2$) (3) $i(i-1)/2 + j$ (4) $j(j-1)/2 + i$ ($1 \leq i, j \leq n$)
- 1038 三对角矩阵按行存储： $k = 2(i-1) + j$ ($1 \leq i, j \leq n$)
- 33 ($k = i(i-1)/2 + j$) ($1 \leq i, j \leq n$)
- 非零元很少 ($t \ll m * n$) 且分布没有规律
- 节省存储空间。
- 上三角矩阵中，主对角线上第 r ($1 \leq r \leq n$) 行有 $n-r+1$ 个元素， a_{ij} 所在行的元素数是 $j-i+1$ 。所以，元素在一维数组的下标 k 和二维数组下标关系： $k = ((i-1) * (2n-i+2)) / 2 + (j-i+1) = (i-1)(2n-i)/2 + j$ ($1 \leq j$)
- 93
- $i(i-1)/2 + j$
- 线性表
- 其余元素组成的表
- (1) 原子（单元素）是结构上不可再分的，可以是一个数或一个结构；而表带结构，本质就是广义表，因作为广义表的元素故称为子表。
- (2) 大写字母 (3) 小写字母 (4) 表中元素的个数 (5) 表展开后所含括号的层数
- 深度
- (1) () (2) (()) (3) 2 (4) 2
- head(head(tail(tail(head(tail(tail(A)))))))
- 表展开后所含括号的层数
- (1) 5 (2) 3
- head(head(tail(LS)))
- head(tail(tail(head(tail(head(A))))))
- head(tail(head(tail(H))))
- (b)
- (x, y, z)
- (d, e)
- GetHead(GetHead(GetTail(L)))
- 本算法中，首先数组 b 中元素以逆置顺序放入 d 数组中，然后数组 a 和数组 d 的元素比较，将大者拷贝到数组 c 。第一个 WHILE 循环到数组 a 或数组 d 结尾，第二个和第三个 WHILE 语句只能执行其中的一个。
- (1) $b[m-i+1]$ (2) $x := a[i]$ (3) $i := i+1$ (4) $x := d[j]$ (5) $j := j+1$ (6) $k := k+1$ (7) $i \leq 1$ (8) $j \leq m$
- (1) ($i == k$) **return** (2) $i+1$ (3) $i-1$ (4) $i! = k$

本算法利用快速排序思想，找到第 k 个元素的位置（下标 $k-1$ 因而开初有 $k--$ ）。内层 do 循环以 $t(a[low])$ 为“枢轴”找到其应在 i 位置。这时若 i 等于 k ，则算法结束。（即第一个空格处 $if(i == k)$ **return**）。否则，若 $i < k$ ，就在 $i+1$ 至 $high$ 中间去查；若 $i > k$ ，则在 low 到 $i-1$ 间去找，直到找到 $i = k$ 为止。

37. 逆置广义表的递归模型如下

$$f(LS) = \begin{cases} \text{null} & \text{若 } LS \text{ 为空} \\ LS & \text{若 } LS \text{ 为原子, 且 } tail(LS) \text{ 为空} \\ \text{append}(f(tail(LS)), head(LS)) & \text{若 } LS \rightarrow tag = 0, \text{ 且 } LS \rightarrow val.ptr.tp \neq \text{null} \\ \text{append}(f(tail(LS)), f(head(LS))) & \text{若 } LS \rightarrow tag = 1 \end{cases}$$

上面 **append(a, b)** 功能是将广义表 a 和 b 作为元素的广义表连接起来。

- (1) $p \rightarrow tag == 0$ // 处理原子
- (2) $h = reverse(p \rightarrow val.ptr.hp)$ // 处理表头
- (3) $p \rightarrow val.ptr.tp$ // 产生表尾的逆置广义表
- (4) $s \rightarrow val.ptr.tp = t$ // 连接

(5) `q->val.ptr.hp=h` //头结点指向广义表

38. 本题要求将1, 2, ..., $n \times n$ 个自然数, 按蛇型方式存放在二维数组 $A[n][n]$ 中。“蛇型”方式, 即是按“副对角线”平行的各对角线, 从左下到右上, 再从右上到左下, 存放 n^2 个整数。对角线共 $2n-1$ 条, 在副对角线上方的对角线, 题目中用 k 表示第 k 条对角线(最左上角 $k=1$), 数组元素 x 和 y 方向坐标之和为 $k+1$ (即题目中的 $i+j=k+1$)。副对角线下方第 k 条对角线与第 $2n-k$ 条对角线对称, 其元素的下标等于其对称元素的相应坐标各加 $(k-n)$ 。

(1) $k \leq 2 \times n - 1$ //共填 $2 \times n - 1$ 条对角线

(2) $q = 2 \times n - k$ //副对角线以下的各条对角线上的元素数

(3) $k \% 2 \neq 0$ // k 为偶数时从右上到左下, 否则从左下向右上填数。(本处计算下标 i 和 j)

(4) $k > n$ //修改副对角线下方的下标 i 和 j 。

(5) $m++$; 或 $m=m+1$ //为填下个数据作准备, m 变化范围 $1..n \times n$ 。

本题解法的另一种思路见本章算法设计题第9题。

39. 本题难点有二: 一是如何求下一出圈人的位置, 二是某人出圈后对该人的位置如何处理。

按题中要求, 从第 s 个人开始报数, 报到第 m 个人, 此人出圈。 n 个人围成一圈, 可看作环状, 则下个出圈人, 其位置是 $(s+m-1) \% n$ 。 n 是人数, 是个变量, 出圈一人减1, 算法中用 i 表示。对第二个问题, 算法中用出圈人后面人的位置依次前移, 并把出圈人的位置(下标)存放到当时最后一个人的位置(下标)。算法最后打印出圈人的顺序。

(1) $(s+m-1) \% i$ //计算出圈人 $s1$

(2) $s1:=i$ //若 $s1=0$, 说明是第 i 个人出圈($i \% i=0$)

(3) $s1 \text{ TO } i-1$ //从 $s1$ 到 i 依次前移, 使人数减1, 并将出圈人放到当前最后一个位置 $A[i]=w$ 。

40. 若第 n 件物品能放入背包, 则问题变为能否再从 $n-1$ 件物品中选出若干件放入背包(这时背包可放入物品的重量变为 $s-w[n]$)。若第 n 件物品不能放入背包, 则考虑从 $n-1$ 件物品选若干件放入背包(这时背包可放入物品仍为 s)。若最终 $s=0$, 则有一解; 否则, 若 $s < 0$ 或虽然 $s > 0$ 但物品数 $n < 1$, 则无解。

(1) $s-w[n], n-1$ //Knap($s-w[n], n-1$)=true

(2) $s, n-1$ // Knap \leftarrow Knap($s, n-1$)

四、应用题

1、958 三维数组以行为主序存储, 其元素地址公式为:

$$LOC(A_{ijk}) = LOC(A_{c1c2c3}) + [(i-c1)V_2V_3 + (j-c2)V_3 + (k-c3)] * L + 1$$

其中 c_i, d_i 是各维的下界和上界, $V_i = d_i - c_i + 1$ 是各维元素个数, L 是一个元素所占的存储单元数。

2. b 对角矩阵的 b 条对角线, 在主对角线上方和下方各有 $b/2$ 条对角线(为叙述方便, 下面设 $a = b/2$)。从第1行至第 a 行, 每行上的元素数依次是 $a+1, a+2, \dots, b-2, b-1$, 最后的 a 行上的元素个数是 $b-1, b-2, \dots, a+1$ 。中间的 $(n-2a)$ 行, 每行元素个数都是 b 。故 b 条对角线上元素个数为 $(n-2a)b + a \times (a+b)$ 。存放在一维数组 $V[1..nb-a(b-a)]$ 中, 其下标 k 与元素在二维数组中下标 i 和 j 的关系为:

$$k = \begin{cases} \frac{(i-1)(i+2a)}{2} + j & \text{当 } 1 \leq i \leq a+1 \\ a(2i-a) + j & \text{当 } a+1 < i \leq n-a+1 \\ a(2i-a) - \frac{(i-n+a)(i-n+a-1)}{2} + j & \text{当 } n-a+1 < i \leq n \end{cases}$$

3. 每个元素32个二进制位, 主存字长16位, 故每个元素占2个字长, 行下标可平移至1到11。

(1) 242 (2) 22 (3) $s+182$ (4) $s+142$

4. 1784 (公式: $Loc(A_{ijkl}) = 100(\text{基地址}) + [(i-c1)v_2v_3v_4 + (j-c2)v_3v_4 + (k-c3)v_4 + (l-c4)] * 4$)

5. $1210+108L$ ($LOC(A[1, 3, -2]) = 1210 + [(k-c3)v_2v_1 + (j-c2)v_1 + (i-c1)] * L$ (设每个元素占 L 个存储单元))

6. 数组占的存储字节数 $= 10 \times 9 \times 7 \times 4 = 2520$; $A[5, 0, 7]$ 的存储地址 $= 100 + [4 \times 9 \times 7 + 2 \times 7 + 5] \times 4 = 1184$

7. 五对角矩阵按行存储, 元素在一维数组中下标(从1开始) k 与 i, j 的关系如下:

$$k = \begin{cases} 4(i-1) + j & (\text{当 } i = 1 \text{ 时}) \\ 4(i-1) + j - 1 & (\text{当 } 1 < i < n \text{ 时}) \\ 4(i-1) + j - 2 & (\text{当 } i = n \text{ 时}) \end{cases}$$

$A[15, 16]$ 是第71个元素, 在向量 $[-10:m]$ 中的存储位置是60。

8. (1) 540 (2) 108 (3) $i=3, j=10$, 即 $A[3, 10]$ 9. $k = i(i-1)/2 + j$

10. 稀疏矩阵 A 有 t 个非零元素, 加上行数 mu 、列数 nu 和非零元素个数 tu (也算一个三元组), 共占用三元组表LTMA的 $3(t+1)$ 个存储单元, 用二维数组存储时占用 $m \times n$ 个单元, 只有当 $3(t+1) < m \times n$ 时, 用LTMA表示 A 才有意义。

解不等式得 $t < m \times n / 3 - 1$ 。

11. 参见10。

12. 题中矩阵非零元素用三元组表存储, 查找某非零元素时, 按常规要从第一个元素开始查找, 属于顺序查找, 时间复杂度为 $O(n)$ 。若使查找时间得到改善, 可以建立索引, 将各行行号及各行第一个非零元素在数组 B 中的位置(下标)偶对放入一向量 C 中。若查找非零元素, 可先在数组 C 中用折半查找到该非零元素的行号, 并取

出该行第一个非零元素在B中的位置，再到B中顺序（或折半）查找该元素，这时时间复杂度为 $O(\log n)$ 。

13. (1) 176 (2) 76和108 (3) 28和116。

14. (1) $k = 3(i-1)$ (主对角线左下角，即 $i=j+1$)

$k = 3(i-1)+1$ (主对角线上，即 $i=j$)

$k = 3(i-1)+2$ (主对角线上，即 $i=j-1$)

由以上三式，得 $k=2(i-1)+j$ ($1 \leq i, j \leq n$; $1 \leq k \leq 3n-2$)

(2) $10^3 * 10^3 - (3 * 10^3 - 2)$

15. 稀疏矩阵A采用二维数组存储时，需要 $n*n$ 个存储单元，完成求 $\sum a_{ii}$ ($1 \leq i \leq n$)时，由于 $a[i][i]$ 随机存取，速度快。但采用三元组表时，若非零元素个数为 t ，需 $3(t+1)$ 个存储单元（第一个分量中存稀疏矩阵A的行数，列数和非零元素个数，以后 t 个分量存各非零元素的行值、列值、元素值），比二维数组节省存储单元；但在求 $\sum a_{ii}$ ($1 \leq i \leq n$)时，要扫描整个三元组表，以便找到行列值相等的非零元素求和，其时间性能比采用二维数组时差。

16. 特殊矩阵指值相同的元素或零元素在矩阵中的分布有一定规律，因此可以对非零元素分配单元（对值相同元素只分配一个单元），将非零元素存储在向量中，元素的下标 i 和 j 和该元素在向量中的下标有一定规律，可以用简单公式表示，仍具有随机存取功能。而稀疏矩阵是指非零元素和矩阵容量相比很小 ($t \ll m*n$)，且分布没有规律。用十字链表作存储结构自然失去了随机存取的功能。即使用三元组表的顺序存储结构，存取下标为 i 和 j 的元素时，要扫描三元组表，下标不同的元素，存取时间也不同，最好情况下存取时间为 $O(1)$ ，最差情况下是 $O(n)$ ，因此也失去了随机存取的功能。

17. 一维数组属于特殊的顺序表，和有序表的差别主要在于有序表中元素按值排序（非递增或非递减），而一维数组中元素没有按元素值排列顺序的要求。

18. $n(n+1)/2$ (压缩存储) 或 n^2 (不采用压缩存储)

19. $LOC(A[i, j]) = LOC(A[3, 2]) + [(i-3)*5 + (j-2)] \times 2$ (按行存放)

$LOC(A[i, j]) = LOC(A[3, 2]) + [(j-2)*6 + (i-3)] \times 2$ (按列存放)

20. n 阶下三角矩阵元素 $A[i][j]$ ($1 \leq i, j \leq n, i \geq j$)。第1列有 n 个元素，第 j 列有 $n-j+1$ 个元素，第1列到第 $j-1$ 列是等腰梯形，元素数为 $(n+(n-j+2))(j-1)/2$ ，而 a_{ij} 在第 j 列上的位置是为 $i-j+1$ 。所以 n 阶下三角矩阵A按列存储，其元素 a_{ij} 在一维数组B中的存储位置 k 与 i 和 j 的关系为：

$k = (n+(n-(j-1)+1))(j-1)/2 + (i-j+1) = (2n-j)(j-1)/2 + i$

21. 三对角矩阵第一行和最后一行各有两个非零元素，其余每行均有三个非零元素，所以共有 $3n-2$ 个元素。

(1) 主对角线左下对角线上的元素下标间有 $i=j+1$ 关系， k 与 i 和 j 的关系为 $k=3(i-1)$ ；主对角线上元素下标间有关系 $i=j$ ， k 与 i 和 j 的关系为 $k=3(i-1)+1$ ；主对角线右上那条对角线上元素下标间有关系 $i=j-1$ ， k 与 i 和 j 的关系为 $k=3(i-1)+2$ 。综合以上三式，有 $k=2(i-1)+j$ ($1 \leq i, j \leq n, |i-j| \leq 1$)

(2) $i=k/3+1$; ($1 \leq k \leq 3n-2$) // $k/3$ 取小于 $k/3$ 的最大整数。下同

$j=k-2(i-1)=k-2(k/3)=k\%3+k/3$

22. 这是一个递归调用问题，运行结果为：DBHEAIFJCKGL

23. (1) FOR循环中，每次执行PerfectShuffle(A, N)和CompareExchange(A, N)的结果：

第1次：A[1..8]=[90, 30, 85, 65, 50, 80, 10, 100]

A[1..8]=[30, 90, 65, 85, 50, 80, 10, 100]

第2次：A[1..8]=[30, 50, 90, 80, 65, 10, 85, 100]

A[1..8]=[30, 50, 80, 90, 10, 65, 85, 100]

第3次：A[1..8]=[30, 10, 50, 65, 80, 85, 90, 100]

A[1..8]=[10, 30, 50, 65, 80, 85, 90, 100]

(2) Demo的功能是将数组A中元素按递增顺序排序。

(3) PerfectShuffle 中WHILE循环内是赋值语句，共 $2N$ 次，WHILE外成组赋值语句，相当 $2N$ 个简单赋值语句；CompareExchange中WHILE循环内是交换语句，最好情况下不发生交换，最差情况下发生 N 次交换，相当于 $3N$ 个赋值语句；Demo中FOR循环循环次数 $\log_2 2N$ ，故按赋值语句次数计算Demo的时间复杂度为：最好情况：

$O(4N * \log_2 2N) \approx O(N \log(2 * N))$ ；最差情况： $O((4N+3N) * \log_2 2N) \approx O(N \log(2 * N))$ 。

24. 这是一个排序程序。运行后B数组存放A数组各数在排序后的位置。结果是：

A={121, 22, 323, 212, 636, 939, 828, 424, 55, 262}

B={3, 1, 6, 4, 8, 10, 9, 7, 2, 5}

C={22, 55, 121, 212, 262, 323, 424, 639, 828, 939}

25. (1) $c = \begin{bmatrix} 3 & 3 & 1 \\ 1 & 1 & 1 \\ 3 & 3 & 2 \end{bmatrix}$ (2) $a = \begin{bmatrix} 3 & 3 & 1 \\ 1 & 1 & 3 \\ 3 & 3 & 3 \end{bmatrix}$

26. (1) 同上面26题 (1)

(2) 对c数组的赋值同所选择的下标 i 和 j 的次序（指外层循环用 j 内层用 i ）没有关系

(3) 同上题26 (2)

(4) 对 i, j 下标取反序后，重复执行第(3)步，A数组所有元素均变为2。（在机器上验证，反复循环3次后，所有元素均变为2）

27. 错误有以下几处：

- (1) 过程参数没有类型说明; (2) 出错条件判断: 缺少OR (i+k>last+1);
 (3) 删除元素时FOR循环应正向, 不应应用反向DOWNTO; (4) count没定义;
 低效体现在两处:

- (1) 删除k个元素时, 不必一个一个元素前移, 而应一次前移k个位置;
 (2) last指针不应一次减1, 而应最后一次减k。

正确的高效算法如下:

```
const m=64;
```

```
TYPE ARR=ARRAY[1..m] OF integer;
```

```
PROCEDURE delk (VAR A:ARR; VAR last:integer;i,k: integer);
```

```
{从数组A[1..last]中删除第i个元素起的k个元素, m为A的上限}
```

```
VAR count: integer;
```

```
BEGIN
```

```
IF (i<0) OR (i>last) OR (k<0) OR (last>m) OR (i+k>last+1)
```

```
THEN write (' error' )
```

```
ELSE[FOR count:= i+k TO last DO A[count-k]:=A[count];
```

```
last:=last-k; ]
```

```
END;
```

28. 这是计数排序程序。

- (a) $c[i]$ ($1 \leq i \leq n$) 中存放A数组中值为i的元素个数。
 (b) $c[i]$ ($1 \leq i \leq n$) 中存放A数组中小于等于i的个数。
 (c) B中存放排序结果, B[1..n]已经有序。
 (d) 算法中有4个并列for循环语句, 算法的时间复杂度为 $O(n)$ 。
 29. 上三角矩阵第一行有n个元素, 第i-1行有 $n - (i-1) + 1$ 个元素, 第一行到第i-1行是等腰梯形, 而第i行上第j个元素 (即 a_{ij}) 是第i行上第j-i+1个元素, 故元素 a_{ij} 在一维数组中的存储位置 (下标k) 为:
 $k = (n + (n - (i-1) + 1)) (i-1) / 2 + (j - i + 1) = (2n - i + 2) (i-1) / 2 + j - i + 1$
 30. 将上面29题的等式进一步整理为:

$$k = (n+1/2) i - i^2/2 + j - n,$$

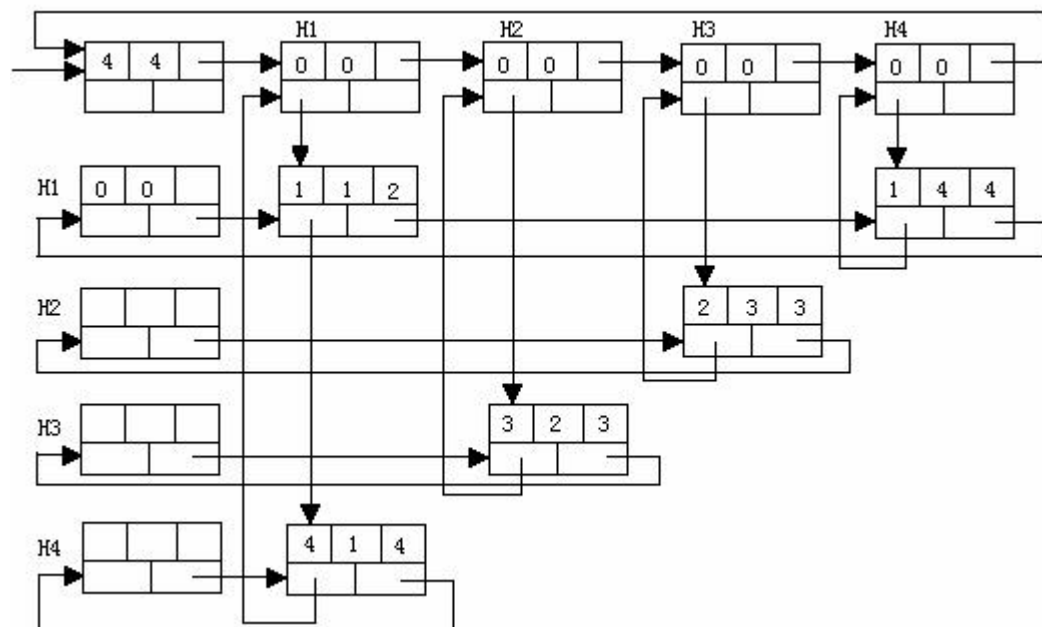
则得 $f_1(i) = (n+1/2) i - i^2/2$, $f_2(j) = j$, $c = -n$ 。

31. (1) 将对称矩阵对角线及以下元素按行序存入一维数组中, 结果如下:

2	0	0	0	3	0	4	0	0	0
---	---	---	---	---	---	---	---	---	---

下标 1 2 3 4 5 6 7 8 9 10

(2) 因行列表头的“行列域”值用了0和0, 下面十字链表中行和列下标均从1开始。



注: 上侧列表头 H_i 和左侧行表头 H_i 是一个 (即 H_1 、 H_2 、 H_3 和 H_4) , 为了清楚, 画成了两个。

32. (1) $k = (2n - j + 2) (j-1) / 2 + i - j + 1$ (当 $i \geq j$ 时, 本题 $n=4$)
 $k = (2n - i + 2) (i-1) / 2 + j - i + 1$ (当 $i < j$ 时, 本题 $n=4$)
 (2) 稀疏矩阵的三元组表为: $s = ((4, 4, 6), (1, 1, 1), (1, 4, 2), (2, 2, 3), (3, 4, 5), (4, 1, 2), (4, 3, 5))$ 。其中第一个三元组是稀疏矩阵行数、列数和非零元素个数。其它三元组均为非零元素行值、列值和元素值。
 33. (1) $k = 2(i-1) + j$ ($1 \leq i, j \leq n, |i-j| \leq 1$)
 $i = \text{floor}(k/3) + 1$ // floor(a) 是取小于等于a的最大整数
 $j = k - 2(i-1)$

推导过程见上面第25题。

(2) 行逻辑链接顺序表是稀疏矩阵压缩存储的一种形式。为了随机存取任意一行的非零元，需要知道每一行第一个非零元在三元组表中的位置。为此，除非零元的三元组表外，还需要一个向量，其元素值是每行第一个非零元在三元组表中的位置。其类型定义如下：

```
typedef struct
{ int mu, nu, tu;          //稀疏矩阵的行数、列数和非零元素个数
  int rpos[maxrow+1];     //行向量，其元素值是每行第一个非零元在三元组表中的位置。
```

```
    Triple data[maxsize];
```

```
} SparsMatrix;
```

因篇幅所限，不再画出行逻辑链接顺序表。

34. 各维的元素数为 $d_i - c_i + 1$ ，则 $a[i_1, i_2, i_3]$ 的地址为：

$a_0 + [(i_1 - c_1)(d_3 - c_3 + 1)(d_2 - c_2 + 1) + (i_2 - c_2)(d_2 - c_2 + 1) + (i_3 - c_3)] * L$

35. 主对角线上元素的坐标是 $i=j$ ，副对角线上元素的坐标 i 和 j 有 $i+j=n+1$ 的关系

(1) $i=j$ 或 $i=n+1-j$ ($1 \leq i, j \leq n$)

(2) 非零元素分布在两条主、副对角线上，除对角线相交处一个元素（下称“中心元素”）外，其余每行都有两个元素。主对角线上的元素，在向量B中存储的下标是 $k=2i-1$ ($i=j, 1 \leq i, j \leq n, 1 \leq k \leq 2n-1$)。

副对角线上的元素，在中心元素前，在向量B中存储的下标是 $k=2i$ ($i < j, 1 \leq i, j \leq n/2$)；在中心元素后，其下标是 $k=2(i-1)$ ($i < j, n/2+1 \leq i, j \leq n, 1 \leq k \leq 2n-1$)。

$$k = \begin{cases} A_0 + 2(i-1) & (i=j, 1 \leq i, j \leq n) \\ A_0 + 2(i-1) + 1 & (i < j, 1 \leq i, j \leq n/2) \\ A_0 + 2(i-1) - 1 & (i < j, n/2+1 \leq i, j \leq n) \end{cases}$$

(3) a_{ij} 在B中的位置 $k=$

36. 由于对称矩阵采用压缩存储，上三角矩阵第一列一个元素，第二列两个元素，第 j 列 j 个元素。上三角矩阵共有 $n(n+1)/2$ 个元素。我们将这些元素存储到一个向量B[n(n+1)/2+1]中。可以看到B[k]和矩阵中的元素 a_{ij} 之间存在着——对应关系：

$$k = \begin{cases} \frac{j(j-1)}{2} + i & \text{当 } i \leq j \\ \frac{i(i-1)}{2} + j & \text{当 } i > j \end{cases}$$

$$\frac{\text{MAX}(i, j) * (\text{MAX}(i, j) - 1)}{2} + \text{MIN}(i, j)$$

则其对应关系可表示为： $k = \frac{\text{MAX}(i, j) * (\text{MAX}(i, j) - 1)}{2} + \text{MIN}(i, j)$ ($1 \leq i, j \leq n, 1 \leq k \leq n(n+1)/2$)

```
int MAX(int x, int y)
{ return (x > y ? x : y);
}
int MIN(int x, int y)
{ return (x < y ? x : y);
}
```

37. 设用 μ, ν 和 t 表示稀疏矩阵行数，列数和非零元素个数，则转置矩阵的行数，列数和非零元素的个数分别是 ν, μ 和 t 。转置可按转置矩阵的三元组表中的元素顺序进行，即按稀疏矩阵的列序，从第1列到第 ν 列，每列中按行值递增顺序，找出非零元素，逐个放入转置矩阵的三元组表中，转时行列值互换，元素值复制。按这种方法，第1列到第1个非零元素一定是转置后矩阵的三元组表中的第1个元素，第1列非零元素在第2列非零元素的前面。这种方法时间复杂度是 $O(n * P)$ ，其中 p 是非零元素个数，当 p 和 $m * n$ 同量级时，时间复杂度为 $O(n^3)$ 。

另一种转置方法称作快速转置，使时间复杂度降为 $O(m * n)$ 。它是按稀疏矩阵三元组表中元素的顺序进行。按顺序取出一个元素，放到转置矩阵三元组表的相应位置。这就要求出每列非零元素个数和每列第一个非零元素在转置矩阵三元组表中的位置，设置了两个附加向量。

38. 广义表中的元素，可以是原子，也可以是子表，即广义表是原子或子表的有限序列，满足线性结构的特性：在非空线性结构中，只有一个称为“第一个”的元素，只有一个成为“最后一个”的元素，第一元素有后继而没有前驱，最后一个元素有前驱而没有后继，其余每个元素有唯一前驱和唯一后继。从这个意义上说，广义表属于线性结构。

39. 数组是具有相同性质的数据元素的集合，同时每个元素又有唯一下标限定，可以说数组是值和下标偶对的有限集合。 n 维数组中的每个元素，处于 n 个关系之中，每个关系都是线性的，且 n 维数组可以看作其元素是 $n-1$ 维数组的一个线性表。而广义表与线性表的关系，见上面38题的解释。

40. 线性表中的元素可以是各种各样的，但必须具有相同性质，属于同一数据对象。广义表中的元素可以是原子，也可以是子表。其它请参见38

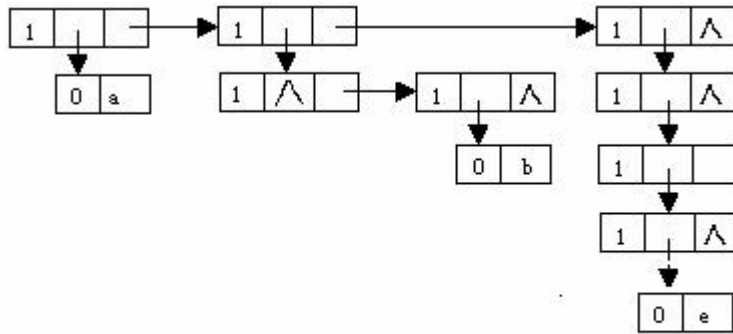
41. (1) (c, d) (2) (b) (3) b (4) (f) (5) ()

42. Head (Tail (Head (Head (L1)))))

Head (Head (Head (Tail (Head (Tail (L2)))))))

类似本题的另外叙述的几个题解答如下：

(1) head (head (tail (tail (L))))) , 设 $L = (a, (c), b), ((e))$



(2) head (head (head (head (tail (tail (L)))))))

(3) head (tail (head (tail (A)))))

(4) H (H (T (H (T (H (T (L))))))))

(5) tail (L) = ((c, d), (e, f))

head (tail (L)) = (c, d)

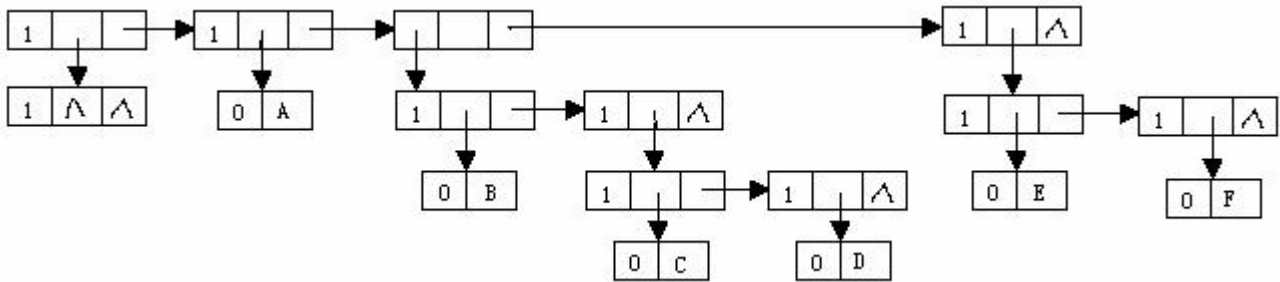
head (head (tail (L))) = (c, d)

tail (head (head (tail (L)))) = (d)

head (tail (head (head (tail (L))))) = d

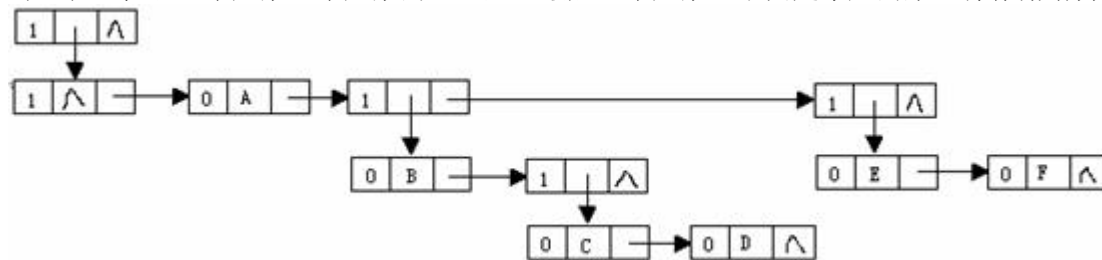
(6) head (tail (head (head (tail (tail (A)))))))

43. 广义表的第一种存储结构的理论基础是, 非空广义表可唯一分解成表头和表尾两部分, 而由表头和表尾可唯一构成一个广义表。这种存储结构中, 原子和表采用不同的结点结构 (“异构”, 即结点域个数不同)。



原子结点两个域: 标志域tag=0表示原子结点, 域DATA表示原子的值; 子表结点三个域: tag=1表示子表, hp和tp分别是指向表头和表尾的指针。在画存储结构时, 对非空广义表不断进行表头和表尾的分解, 表头可以是原子, 也可以是子表, 而表尾一定是表 (包括空表)。上面是本题的第一种存储结构图。

广义表的第二种存储结构的理论基础是, 非空广义表最高层元素间具有逻辑关系: 第一个元素无前驱有后继, 最后一个元素无后继有前驱, 其余元素有唯一前驱和唯一后继。有人将这种结构看作扩充线性结构。这种存储结构中, 原子和表均采用三个域的结点结构 (“同构”)。结点中都有一个指针域指向后继结点。原子结点中还包括标志域tag=0和原子值域DATA; 子表结点还包括标志域tag=1和指向子表的指针hp。在画存储结构时, 从左往右一个元素一个元素的画, 直至最后一个元素。下面是本题的第二种存储结构图。



由于存储结构图占篇幅较大, 下面这类题均不再解答。

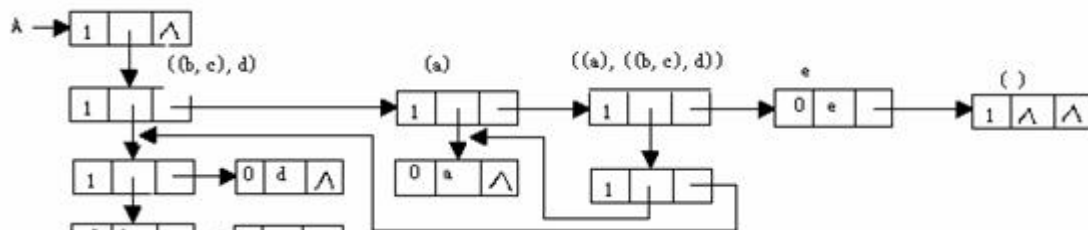
44. 深度为5, 长度为2

45. (1) 略

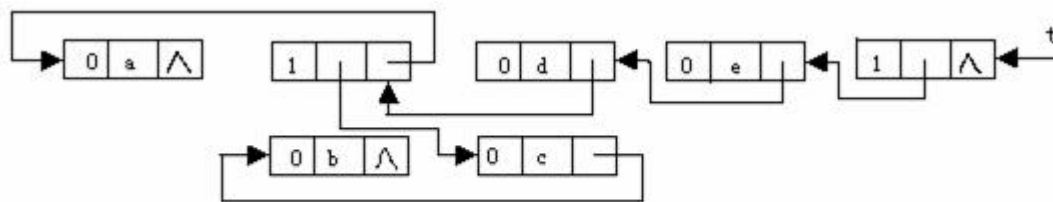
(2) 表的长度为5, 深度为4

(3) head (tail (head (head (head (tail (tail (tail (tail (A)))))))))))

46. 共享结构广义表 $A = ((b, c), d), (a), ((a), (b, c), d), e, ()$ 的存储表示:



47. (1) 算法A的功能是逆置广义表p (即广义表由p指针所指)。逆置后的广义表由t指向。
(2) 逆置后的广义表由t指向, 这时p=nil。



51. $p(x_1, x_2, x_3) = 2x_1^5x_2^2x_3^4 + 5x_1^5x_2^3x_3^3 + 3x_1x_2^4x_3^2 + (x_1^5x_2^3 + x_2)x_3 + 6$

52. (1) $H(A(a_1, a_2), B(b_1), C(c_1, c_2), x)$
 $HEAD(TAIL(HEAD(H))) = a_2$
 (2) 略

五. 算法设计题

1. [题目分析] 本题是在向量D内插入元素问题。首先要查找插入位置，数据x插入到第i个数据组的末尾，即是第i+1个数据组的开始，而第i ($1 \leq i \leq n$) 个数据组的首地址由数组s (即数组元素s[i]) 给出。其次，数据x插入后，还要维护数组s，以保持空间区D和数组s的正确的相互关系。

```
void Insert (int s[], datatype D[], x, int i,m)
//在m个元素的D数据区的第i个数据组末尾，插入新数据x，第i个数据组的首址由数组s给出。
{if (i<1 || i>n) {printf (“参数错误”)； exit (0)；}
  if (i==n) D[m]=x;    // 在第n个数据组末尾插入元素。
  else {for (j=m-1;j>=s[i+1];j--) D[j+1]=D[j]; // 第i+1 个数据组及以后元素后移
        D[s[i+1]]=x; // 将新数据x插入
        for(j=i+1;j<=n;j++) s[j]++; // 维护空间区D和数组s的关系。
      } //结束元素插入
  m++; //空间区D的数据元素个数增1。
} // 算法Insert结束
```

【算法讨论】数据在空间区从下标0开始，最后一个元素的下标是m-1。设空间区容量足够大，未考虑空间溢出问题。数组s随机存数，而向量D数据插入，引起数组元素移动，时间复杂度是 $O(n)$ 。

2. [题目分析] 设稀疏矩阵的非零元素的三元组以行序为主存储在三元组表中。矩阵的相加是对应元素的相加。对两非零元素相加, 若行号不等, 则行号大者是结果矩阵中的非零元素。若行号相同, 则列号大者是结果中一非零元素; 若行号列号相同, 若对应元素值之和为零, 不予存储, 否则, 作为新三元组存到三元组表中。题目中要求时间复杂度为 $O(m+n)$ 。因此需从两个三元组表的最后一个元素开始相加。第一个非零元素放在A矩阵三元组表的第 $m+n$ 位置上。结果的三元组至多是 $m+n$ 个非零元素。最后若发生对应元素相加和为零的情况, 对三元组表中元素要进行整理, 以便使第一个三元组存放在下标1的位置上。

```
CONST maxnum=大于非零元素数的某个常量
TYPE tuple=RECORD
    i,j: integer;  v: elemtyp;
END;
sparmattyp=RECORD
    mu, nu, tu: integer;
    data: ARRAY[1..maxnum] OF tuple;
END;
```

```
PROC AddMatrix (VAR A: sparmatt; B: sparmatt);
// 稀疏矩阵A和B各有m和n个非零元素，以三元组表存储。A的空间足够大，本算法实现两个稀疏矩阵相加，结果放到A中。
  L:=m; p:=n; k:=m+n; // L, p为A, B三元组表指针，k为结果三元组表指针（下标）。
  A.tu:=m+n; // 暂存结果矩阵非零元素个数
  WHILE (L≥1) AND (p≥1) DO
```

[CASE // 行号不等时, 行号大者的三元组为结果三元组表中一项。

A.data[L].i>B.data[p].i: A.data[k]:=A.data[L]; L:=L-1; // A中当前项为结果项

A.data[L].i<B.data[p].i: A.data[k]:=B.data[p]; p:=p-1; //B中当前项为结果当前项

A.data[L].i=B.data[p].i:

CASE //行号相等时, 比较列号

A.data[L].j>B.data[p].j: A.data[k]:=A.data[L]; L:=L-1;

A.data[L].j<B.data[p].j: A.data[k]:=B.data[p]; p:=p-1;

A.data[L].j=B.data[p].j: IF A.data[L].v+B.data[p].v≠0 THEN

[A.data[L].v=A.data[L].v+ B.data[p].v;

A.data[k]:= A.data[L];]

L:=L-1; p:=p-1;

ENDC; //结束行号相等时的处理

ENDC; //结束行号比较处理。

k:=k-1; //结果三元组表的指针前移(减1)

]//结束WHILE循环。

WHILE p>0 DO[A.data[k]:=B.data[p]; k:=k-1; p:=p-1;] //处理B的剩余部分。

WHILE L>1 DO[A.data[k]:=A.data[L]; k:=k-1; L:=L-1;] //处理A的剩余部分。

IF k>1 THEN //稀疏矩阵相应元素相加时, 有和为零的元素, 因而元素总数<m+n。

[FOR p:=k TO m+n DO A[p-k+1]:=A[p]; // 三元组前移, 使第一个三元组的下标为1。

A.tu=m+n-k+1;] // 修改结果三元组表中非零元素个数。

ENDP; // 结束addmatrix

[算法讨论]算法中三元组的赋值是“成组赋值”, 可用行值、列值和元素值的三个赋值句代替。A和B的三元组表的当前元素的指针L和p, 在每种情况处理后均有修改, 而结果三元组表的指针k在CASE语句后统一处理(k:=k-1)。算法在B的第一个元素“大于”A的最后一个元素时, 时间复杂度最佳为O(n), 最差情况是每个元素都移动(赋值)了一次, 且出现了和为零的元素, 致使最后(m+n-k+1)个元素向前平移一次, 时间复杂度最差为O(m+n)。

3. [题目分析]从n个数中, 取出所有k个数所有组合。设数已存于数组A[1..n]中。为使结果唯一, 可以分别求出包括A[n]和不包括A[n]的所有组合。即包括A[n]时, 求出从A[1..n-1]中取出k-1个元素的所有组合, 不包括A[n]时, 求出从A[1..n-1]中取出k个元素的所有组合。

CONST n=10; k=3;

TYPE ARR=ARRAY[1..n] OF integer;

VAR A, B: ARR; // A中存放n个自然数, B中存放输出结果。

PROC outresult; //输出结果

FOR j:=1 TO k DO write (B[j]); writeln;

ENDP;

PROC nkcombination (i, j, k: integer);

//从i个数中连续取出k个数的所有组合, i个数已存入数组A中, j为结果数组B中的下标。

IF k=0 THEN outresult

ELSE IF (i-k≥0) THEN [B[j]:=A[i]; j:=j+1;

nkcombination (i-1, k-1, j);

nkcombination (i-1, k, j-1);]

ENDP;

[算法讨论]本算法调用时, i是数的个数(题目中的n), $k \leq i$, j是结果数组的下标。按题中例子, 用nkcombination(5, 1, 3)调用。若想按正序输出, 如123, 124, ..., 可将条件表达式 $i-k \geq 0$ 改为 $i+k-1 \leq n$, 其中n是数的个数, i初始调用时为1, 两个调用语句中的i-1均改为i+1。

4. [题目分析]题目中要求矩阵两行元素的平均值按递增顺序排序, 由于每行元素个数相等, 按平均值排列与按每行元素之和排列是一个意思。所以应先求出各行元素之和, 放入一维数组中, 然后选择一种排序方法, 对该数组进行排序, 注意在排序时若有元素移动, 则与之相应的行中各元素也必须做相应变动。

void Translation(float *matrix, int n)

//本算法对n×n的矩阵matrix, 通过行变换, 使其各行元素的平均值按递增排列。

{int i, j, k, l;

float sum, min; //sum暂存各行元素之和

float *p, *pi, *pk;

for(i=0; i<n; i++)

{sum=0.0; pk=matrix+i*n; //pk指向矩阵各行第1个元素。

for (j=0; j<n; j++) {sum+=*(pk); pk++;} //求一行元素之和。

*(p+i)=sum; //将一行元素之和存入一维数组。

}//for i

for(i=0; i<n-1; i++) //用选择法对数组p进行排序

{min=*(p+i); k=i; //初始设第i行元素之和最小。

for(j=i+1; j<n; j++) if(p[j]<min) {k=j; min=p[j];} //记新的最小值及行号。

```

    if(i!=k)                //若最小行不是当前行,要进行交换(行元素及行元素之和)
    {pk=matrix+n*k;         //pk指向第k行第1个元素.
      pi=matrix+n*i;        //pi指向第i行第1个元素.
      for(j=0;j<n;j++)      //交换两行中对应元素.
      {sum=*(pk+j); *(pk+j)=*(pi+j); *(pi+j)=sum;}
      sum=p[i]; p[i]=p[k]; p[k]=sum; //交换一维数组中元素之和.
    } //if
  } //for i
  free(p); //释放p数组.
} // Translation

```

[算法分析] 算法中使用选择法排序, 比较次数较多, 但数据交换(移动)较少. 若用其它排序方法, 虽可减少比较次数, 但数据移动会增多. 算法时间复杂度为 $O(n^2)$.

5. [题目分析] 因为数组中存放的是从1到N的自然数, 原程序运行后, 数组元素A[i] ($1 \leq i \leq N$) 中存放的是A[1]到A[i-1]中比原A[i]小的数据元素的个数. 易见A[N]+1就是原A[N]的值(假定是j, $1 \leq j \leq N$). 设一元素值为1的辅助数组flag, 采用累加, 确定一个值后, flag中相应元素置零. 下面程序段将A还原成原来的A:

```

VAR flag:ARRAY[1..N] OF integer;
FOR i:=1 TO N DO flag[i]:=1; //赋初值
FOR i:=N DOWNTO 1 DO
  BEGIN sum:=0; j:=1; found:=false;
    WHILE j<=N AND NOT found DO
      BEGIN sum:=sum+flag[j];
        IF sum=A[i]+1 THEN BEGIN flag[j]:=0; found:=true; END;
      END;
    A[i]:=j;
  END;
END;

```

6. [题目分析] 寻找马鞍点最直接的方法, 是在一行中找出一个最小值元素, 然后检查该元素是否是元素所在列的最大元素, 如是, 则输出一个马鞍点, 时间复杂度是 $O(m*(m+n))$. 本算法使用两个辅助数组max和min, 存放每列中最大值元素的行号和每行中最小值元素的列号, 时间复杂度为 $O(m*n+m)$, 但比较次数比前种算法会增加, 也多使用向量空间.

```

int m=10, n=10;
void Saddle(int A[m][n])
//A是m*n的矩阵, 本算法求矩阵A中的马鞍点.
{int max[n]={0}, //max数组存放各列最大值元素的行号, 初始化为行号0;
  min[m]={0}, //min数组存放各行最小值元素的列号, 初始化为列号0;
  i, j;
  for(i=0; i<m; i++) //选各行最小值元素和各列最大值元素.
    for(j=0; j<n; j++)
      {if(A[max[j]][j]<A[i][j]) max[j]=i; //修改第j列最大元素的行号
        if(A[i][min[i]]>A[i][j]) min[i]=j; //修改第i行最小元素的列号.
      }
  for (i=0; i<m; i++)
    {j=min[i]; //第i行最小元素的列号
      if(i==max[j]) printf("A[%d][%d]是马鞍点, 元素值是%d", i, j, A[i][j]); //是马鞍点
    }
} // Saddle

```

[算法讨论] 以上算法假定每行(列)最多只有一个可能的马鞍点, 若有多个马鞍点, 因为一行(或一列)中可能的马鞍点数值是相同的, 则可用二维数组min2, 第一维是行向量, 是各行行号, 第二维是列向量, 存放一行中最大值的列号. 对最大值也同样处理, 使用另一二维数组max2, 第一维是列向量, 是各列列号, 第二维存放该列最大值元素的行号. 最后用类似上面方法, 找出每行(i)最小值元素的每个列号(j), 再到max2数组中找该列是否有最大值元素的行号(i), 若有, 则是马鞍点.

7. [题目分析] 我们用l代表最长平台的长度, 用k指示最长平台在数组b中的起始位置(下标)。用j记住局部平台的起始位置, 用i指示扫描b数组的下标, i从0开始, 依次和后续元素比较, 若局部平台长度(i-j)大于1时, 则修改最长平台的长度k (l=i-j) 和其在b中的起始位置 (k=j), 直到b数组结束, l即为所求。

```

void Platform (int b[ ], int N)
//求具有N个元素的整型数组b中最长平台的长度。
{l=1; k=0; j=0; i=0;
  while(i<n-1)
  {while(i<n-1 && b[i]==b[i+1]) i++;
    if(i-j+1>l) {l=i-j+1; k=j;} //局部最长平台
    i++; j=i; } //新平台起点
  printf("最长平台长度%d, 在b数组中起始下标为%d", l, k);
} // Platform

```

8. [题目分析]矩阵中元素按行和按列都已排序,要求查找时间复杂度为 $O(m+n)$,因此不能采用常规的二层循环的查找。可以先从右上角($i=a, j=d$)元素与 x 比较,只有三种情况:一是 $A[i, j]>x$, 这情况下向 j 小的方向继续查找;二是 $A[i, j]<x$, 下步应向 i 大的方向查找;三是 $A[i, j]=x$, 查找成功。否则,若下标已超出范围,则查找失败。

```
void search(datatype A[ ][ ], int a,b,c,d, datatype x)
//n*m矩阵A,行下标从a到b,列下标从c到d,本算法查找x是否在矩阵A中.
{i=a; j=d; flag=0; //flag是成功查到x的标志
while(i<=b && j>=c)
    if(A[i][j]==x) {flag=1;break;}
    else if (A[i][j]>x) j--; else i++;
if(flag) printf("A[%d][%d]=%d", i, j, x); //假定x为整型.
else printf("矩阵A中无%d 元素", x);
}算法search结束。
```

[算法讨论]算法中查找 x 的路线从右上角开始,向下(当 $x>A[i, j]$)或向左(当 $x<A[i, j]$)。向下最多是 m ,向左最多是 n 。最佳情况是在右上角比较一次成功,最差是在左下角($A[b, c]$),比较 $m+n$ 次,故算法最差时间复杂度是 $O(m+n)$ 。

9. [题目分析]本题的一种算法前面已讨论(请参见本章三、填空题38)。这里给出另一中解法。分析数的填法,是按“从右上到左下”的“蛇形”,沿平行于副对角线的各条对角线上,将自然数从小到大填写。当从右上到左下时,坐标 i 增加,坐标 j 减小,当 j 减到小于0时结束,然后 j 从0开始增加,而 i 从当前值开始减少,到 $i<0$ 时结束。然后继续如此循环。当过副对角线后,在 $i>n-1$ 时, $j=j+2$,开始从左下向右上填数;而当 $j>n-1$ 时 $i=i+2$,开始从右上向左下的填数,直到 $n*n$ 个数填完为止。

```
void Snake_Number(int A[n][n],int n)
//将自然数1..n*n,按“蛇形”填入n阶方阵A中.
{i=0; j=0; k=1; //i, j是矩阵元素的下标, k是要填入的自然数.
while(i<n && j<n)
    {while(i<n && j>=0) //从右上向左下填数,
        {A[i][j]=k++; i++; j--;}
    if((j<0)&&(i<n)) j=0; //副对角线及以上部分的新i, j坐标.
    else {j=j+2; i=n-1;} // 副对角线以下的新的i, j坐标.
    while(i>=0 && j<n) //从左下向右上
        {A[i][j]=k++; i--; j++;}
    if(i<0 && j<n) i=0;
    else {i=i+2; j=n-1;}
    }//最外层while
} //Snake_Number
```

10. [题目分析]判断二维数组中元素是否互不相同,只有逐个比较,找到一对相等的元素,就可结论为不是互不相同。如何达到每个元素同其它元素比较一次且只一次?在当前行,每个元素要同本行后面的元素比较一次(下面第一个循环控制变量 p 的for循环),然后同第 $i+1$ 行及以后各行元素比较一次,这就是循环控制变量 k 和 p 的二层for循环。

```
int JudgeEqual(int a[m][n],int m,n)
//判断二维数组中所有元素是否互不相同,如是,返回1;否则,返回0.
{for(i=0; i<m; i++)
    for(j=0; j<n-1; j++)
        { for(p=j+1; p<n; p++) //和同行其它元素比较
            if(a[i][j]==a[i][p]) {printf("no"); return(0); }
            //只要有一个相同的,就结论不是互不相同
        for(k=i+1; k<m; k++) //和第i+1行及以后元素比较
            for(p=0; p<n; p++)
                if(a[i][j]==a[k][p]) {printf("no"); return(0); }
        } // for(j=0; j<n-1; j++)
    printf("yes"); return(1); //元素互不相同
} //算法JudgeEqual结束
```

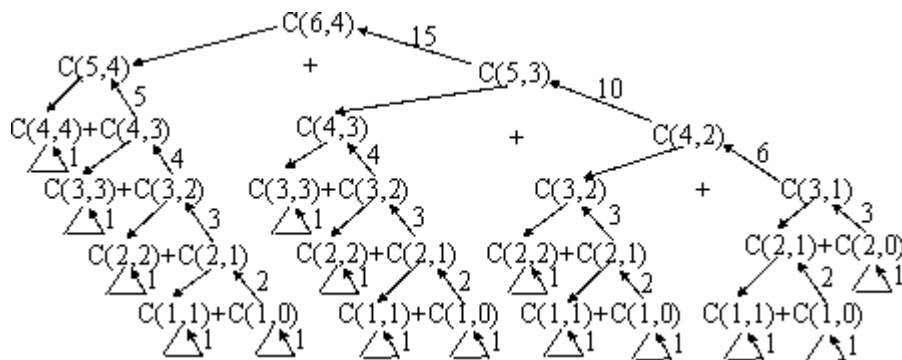
(2) 二维数组中的每一个元素同其它元素都比较一次,数组中共 $m*n$ 个元素,第1个元素同其它 $m*n-1$ 个元素比较,第2个元素同其它 $m*n-2$ 个元素比较,……,第 $m*n-1$ 个元素同最后一个元素($m*n$)比较一次,所以在元素互不相等时总的比较次数为 $(m*n-1)+(m*n-2)+\cdots+2+1=(m*n)(m*n-1)/2$ 。在有相同元素时,可能第一次比较就相同,也可能最后一次比较时相同,设在 $(m*n-1)$ 个位置上均可能相同,这时的平均比较次数约为 $(m*n)$

$(m*n-1)/4$, 总的时间复杂度是 $O(n^4)$ 。

11. 二项式 $(a+b)^n$ 展开式的系数的递归定义为:

$$C(n, k) = \begin{cases} 1 & \text{当 } k=0 \text{ 或 } k=n (n \geq 0) \\ C(n-1, k) + C(n-1, k-1) & \text{当 } (0 < k < n) \end{cases} \quad C(n, k) = C_n^k = \frac{n(n-1)\cdots(n-k+1)}{1*2*\cdots*(k-1)*k}$$

(1) `int BiForm(int n, k)` //二项式展开式的系数的递归算法
`{if(n<0 || k<0 || k>=n) {printf(“参数错误\n ”);exit(0);}`
`if(k==0 || k==n) return(1);`
`else return(BiForm(n-1, k)+BiForm(n-1, k-1);`
`}`
 (2) $C(6, 4)$ 的递归树



(3) 计算 $C(n, k)$ ($0 \leq k \leq n$) 的非递归算法

```
int cnk(int n, int k)
{int i; long x=1, y=1;
for (i=1; i<=k; i++) x*=i;
for (i=n-k+1; i<=n; i++) y*=i;
return(y/x)
} //cnk
```

12. [题目分析] 本题属于排序问题，只是排出正负，不排出大小。可在数组首尾设两个指针 i 和 j ， i 自小至大搜索到负数停止， j 自大至小搜索到正数停止。然后 i 和 j 所指数据交换，继续以上过程，直到 $i=j$ 为止。

```
void Arrange(int A[], int n)
//n个整数存于数组A中，本算法将数组中所有正数排在所有负数的前面
{int i=0, j=n-1, x; //用类C编写，数组下标从0开始
while(i<j)
{while(i<j && A[i]>0) i++;
while(i<j && A[j]<0) j--;
if(i<j) {x=A[i]; A[i]=A[j]; A[j]=x; } //交换A[i] 与A[j]
}
} //算法Arrange结束.
```

[算法讨论] 对数组中元素各比较一次，比较次数为 n 。最佳情况(已排好，正数在前，负数在后)不发生交换，最差情况(负数均在正数前面)发生 $n/2$ 次交换。用类C编写，数组界偶是 $0..n-1$ 。空间复杂度为 $O(1)$ 。

类似本题的其它题的解答：

(1) 与上面12题同，因要求空间复杂度也是 $O(n)$ ，可另设一数组 C ，对 A 数组从左到右扫描，小于零的数在 C 中从左(低下标)到右(高下标)存，大于等于零的数在 C 中从右到左存。

(2) 将12题中判定正数 ($A[i]>0$) 改为判偶数 ($A[i]\%2==0$)，将判负数 ($A[j]<0$) 改为 ($A[j]\%2!=0$)。

(3) 同 (2)，只是要求奇数排在偶数之前。

(4) 利用快速排序思想，进行一趟划分。

```
int Partition(int A[], int n)
//将n个元素的数组A调整为左右两部分，且左边所有元素小于右边所有元素，返回分界位置。
{int i=0, j=n-1, rp=A[0]; //设数组元素为整型
while(i<j)
{while(i<j && A[j]>=rp) j--;
while(i<j && A[i]<=rp) i++;
if(i<j) { x=A[i]; A[i]=A[j]; A[j]=x; }
}
A[i]=rp; return(i); //分界元素
} // Partition
```

13. [题目分析] 设 n 个元素存放在数组 $A[1..n]$ 中。设 S 初始为空集，可依次将数组 A 的每一个元素并入 S ，产生了含一个元素的若干集合，再以含一个元素的集合为初始集合，依次并入 A 的第二个(异于 S 的那个元素)元素并入 S ，形成了含两个元素的若干集合，……，如此下去，直至 $A[i]$ 的全部元素并入。

```
CONST n=10;
TYPE datatype=char;
VAR A: array[1..n] OF datatype;
PROC powerset(s:set OF datatype)
```

```
[outset(s); //输出集合S
FOR i:=1 TO n DO powerset(S+A[i]);
]
```

ENDP;

调用本过程时，参数S为空集[]。

- 14、[题目分析] 设稀疏矩阵是Amxn, Hm是总表头指针。设rch是行列表头指针，则rch→right=rch时该行无非零元素，用i记行号，用一维数组元素A[i]记第i行非零元个数。（为方便输出，设元素是整数。）

```
int MatrixNum(Olink Hm)
//输出由Hm指向的十字链表中每一行的非零元素个数
{Olink rch=Hm->uval.next, p;
int A[]; i=1;//数组A记各行非零元个数, i记行号
while(rch!=Hm)//循环完各行列表头
{p=rch->right; num=0; //p是稀疏矩阵行内工作指针, num记该行非零个数
while(p!=rch)//完成行内非零元的查找
{printf( "M[%d][%d]=%d", p->row, p->col, p->uval.e);
num++;p=p->right; printf( "\n"); //指针后移 }
A[i++]=num; //存该行非零元个数
rch=rch->uval.next; //移到下一行列表头
}
num=0
for(j=1; j<=i; j++)//输出各行非零元个数
{num+=A[j]; printf( "第%d行非零元个数为%d\n", j, A[j]); }
return(num); //稀疏矩阵非零元个数
} //算法结束
```

- 15、[题目分析] 广义表的元素有原子和表。在读入广义表“表达式”时，遇到左括号‘（’就递归的构造子表，否则若是原子，就建立原子结点；若读入逗号‘，’，就递归构造后续子表；若n=0，则构造含空格字符的空表，直到碰到输入结束符号（‘#’）。设广义表的形式定义如下：

```
typedef struct node
{int tag; //tag=0为原子, tag=1为子表
struct node *link; //指向后继结点的指针
union {struct node *slink; //指向子表的指针
char data; //原子
}element;
}Glist;
Glist *creat ()
//建立广义表的存储结构
{char ch; Glist *gh;
scanf( "%c", &ch);
if(ch==' ') gh=null;
else {gh=(Glist*)malloc(sizeof(Glist));
if(ch=='(') {gh->tag=1; //子表
gh->element.slink=creat(); } //递归构造子表
else {gh->tag=0; gh->element.data=ch;} //原子结点
}
scanf( "%c", &ch);
if(gh!=null) if(ch==' , ') gh->link=creat(); //递归构造后续广义表
else gh->link=null;
return(gh);
}
} //算法结束
```

- 16、(1)略

(2)求广义表原子个数的递归模型如下

$$f(p) = \begin{cases} 1 + f(p^{\wedge}link) & \text{如果 } p^{\wedge}tag = 0 \\ f(p^{\wedge}sublist) + f(p^{\wedge}link) & \text{如果 } p^{\wedge}tag = 1 \end{cases}$$

```
PROC Number(p:glist; VAR n: integer)
VAR m:integer;
n:=0;
IF p<>NIL THEN
[IF p^tag=0 THEN n:=1 ELSE Number(p^.sublist, m)
n:=n+m; Number(p^.link, m); n:=n+m; ]
ENDP;
```

17. `int Count(glist *gl)`

//求广义表原子结点数据域之和, 原子结点数据域定义为整型

```
{if(gl==null) return(0);
  else if (gl->tag==0) return((p->data)+count(gl->link));
    else return(count(gl->sublist)+count(gl->link)); }
} // Count
```

18. (1) 在 n 个正整数中, 选出 k ($k \leq m$) 个最大的数, 应使用堆排序方法。对深度为 h 的堆, 筛选算法中关键字的比较次数至多为 $2(h-1)$ 次。建堆总共进行的关键字比较次数不超过 $4n$, 堆排序在最坏情况下的时间复杂度是 $O(n \log n)$ 。

`int r[1000];` // `r[1000]`是整型数组

(2) `void sift(int r[], int k, m, tag)`

//已知`r[k+1..m]`是堆, 本算法将`r[k..m]`调整成堆, `tag=1`建立大根堆, `tag=2`建立小根堆

```
{i=k; j=2*i; x=r[k];
  while (j<=m)
  {if (tag==2) //建立小根堆
    {if (j<m && r[j]>r[j+1]) j++; //沿关键字小的方向筛选
      if(r[j]<x) {r[i]=r[j]; i=j; j=2*i;}
      else break;}
    else //建立大根堆
    {if (j<m && r[j]<r[j+1]) j++; //沿关键字小的方向筛选
      if(r[j]>x) {r[i]=r[j]; i=j; j=2*i;}
      else break;}
  }
```

`r[i]=x;`

`} //sift`

`main(int argc, char *argv[])`

//根据命令行中的输入, 从1000个数中选取 n 个最大数或 n 个最小数

`{int m=1000, i, j;`

`n=argv[2];` //从命令行输入的第二个参数是需要输出的数的个数

`if(n>m) {printf("参数错误\n"); exit(0);}`

`for(i=0; i<m; i++) scanf("%d", &r[i]);` //输入1000个大小不同的正整数

`if (argv[1]== 'a')` //输出 n 个最大数, 要求建立大根堆

`{for(i=m/2; i>0; i--) sift(r, i, m, 1)`

`printf("%d个最大数依次为\n", n);`

`for(i=m; i>m-n+1; i--)` //输出 n 个最大数

`{printf("%5d", r[i]); j++; if((j+1)%5==0) printf("\n");` //一行打印5个数

`sift(r, 1, i-1, 1); }` //调堆

`}`

`else // (argv[1]== 'i')` //输出 n 个最小数, 要求建立小根堆

`{for(i=m/2; i>0; i--) sift(r, i, m, 2)`

`printf("%d个最小数依次为\n", n);`

`for(i=m; i>m-n+1; i--)` //输出 n 个最小数

`{printf("%5d", r[i]); j++; if((j+1)%5==0) printf("\n");` //一行打印5个数

`sift(r, 1, i-1, 2); }` //调堆

`}`

`} //main`

[算法讨论] 算法讨论了建堆, 并输出 n (n 小于等于 m) 个最大(小)数的情况, 由于要求输出 n 个最大数或最小数, 必须建立极大化堆和极小化堆。注意输出时的`for`循环控制到变量 i 从 m 变化到 $m-n+1$, 这是堆的性质决定的, 只有堆顶元素才是最大(小)的。要避免使 i 从1到 n 来输出 n 个最大(小)数的错误。

19. [题目分析] 题目要求调整后第一数组(A)中所有数均不大于第二个数组(B)中所有数。因两数组分别有序, 这里实际是要求第一数组的最后一个数 $A[m-1]$ 不大于第二个数组的第一个数 $B[0]$ 。由于要求将第二个数组的数插入到第一个数组中。因此比较 $A[m-1]$ 和 $B[0]$, 如 $A[m-1]>B[0]$, 则交换。交换后仍保持A和B有序。重复以上步骤, 直到 $A[m-1] \leq B[0]$ 为止。

`void Rearranger (int A[], B[], m, n)`

//A和B是各有 m 个和 n 个整数的非降数组, 本算法将B数组元素逐个插入到A中, 使A中各元素均不大于B中各元素, 且两数组仍保持非降序排列。

`{ while (A[m-1]>B[0])`

`{x=A[m-1]; A[m-1]=B[0];` //交换 $A[m-1]$ 和 $B[0]$

`j=1;`

`while(j<n && B[j]<x) B[j-1]=B[j++];` //寻找 $A[m-1]$ 的插入位置

`B[j-1]=x;`

`x=A[m-1]; i=m-2;`

```

    while(i>=0 && A[i]>x) A[i+1]=A[i--]; //寻找B[0]的插入位置
    A[i+1]=x;
}
}算法结束

```

20、[题目分析]本题中数组A的相邻两段分别有序，要求将两段合并为一段有序。由于要求附加空间为 $O(1)$ ，所以将前段最后一个元素与后段第一个元素比较，若正序，则算法结束；若逆序则交换，并将前段的最后一个元素插入到后段中，使后段有序。重复以上过程直到正序为止。

```

void adjust(int A[],int n)
//数组A[n-2k+1..n-k]和[n-k+1..n]中元素分别升序，算法使A[n-2k+1..n]升序
{
    i=n-k;j=n-k+1;
    while(A[i]>A[j])
    {
        x=A[i]; A[i]=A[j]; //值小者左移，值大者暂存于x
        k=j+1;
        while (k<n && x>A[k]) A[k-1]=A[k++]; //调整后段有序
        A[k-1]=x;
        i--; j--; //修改前段最后元素和后段第一元素的指针
    }
}
}算法结束

```

[算法讨论]最佳情况出现在数组第二段[n-k+1..n]中值最小元素A[n-k+1]大于等于第一段值最大元素A[n-k]，只比较一次无须交换。最差情况出现在第一段的最小值大于第二段的最大值，两段数据间发生了k次交换，而且每次段交换都在段内发生了平均 $(k-1)$ 次交换，时间复杂度为 $O(n^2)$ 。

21、[题目分析]题目要求按B数组内容调整A数组中记录的次序，可以从i=1开始，检查是否B[i]=i。如是，则A[i]恰为正确位置，不需再调；否则，B[i]=k≠i，则将A[i]和A[k]对调，B[i]和B[k]对调，直到B[i]=i为止。

```

void CountSort (rectype A[],int B[])
//A是100个记录的数组，B是整型数组，本算法利用数组B对A进行计数排序
{
    int i,j,n=100;
    i=1;
    while(i<n)
    {
        if(B[i]!=i) //若B[i]=i则A[i]正好在自己的位置上，则不需要调整
        {
            j=i;
            while (B[j]!=i)
            {
                k=B[j]; B[j]=B[k]; B[k]=k; // B[j]和B[k]交换
                r0=A[j]; A[j]=A[k]; A[k]=r0; } //r0是数组A的元素类型, A[j]和A[k]交换
            i++; } //完成了一个小循环，第i个已经安排好
    }
}
}算法结束

```

22、[题目分析]数组A和B的元素分别有序，欲将两数组合并到C数组，使C仍有序，应将A和B拷贝到C，只要注意A和B数组指针的使用，以及正确处理一数组读完数据后将另一数组余下元素复制到C中即可。

```

void union(int A[], B[], C[], m, n)
//整型数组A和B各有m和n个元素，前者递增有序，后者递减有序，本算法将A和B归并为递增有序的数组C。
{
    i=0; j=n-1; k=0; // i, j, k分别是数组A, B和C的下标，因用C描述，下标从0开始
    while(i<m && j>=0)
    {
        if(a[i]<b[j]) c[k++]=a[i++]; else c[k++]=b[j--];
        while(i<m) c[k++]=a[i++];
        while(j>=0) c[k++]=b[j--];
    }
}
}算法结束

```

[算法讨论]若不允许另辟空间，而是利用A数组（空间足够大），则初始k=m+n-1，请参见第2章算法设计第4题。

23、[题目分析]本题要求建立有序的循环链表。从头到尾扫描数组A，取出A[i] ($0 \leq i < n$)，然后到链表中去查找值为A[i]的结点，若查找失败，则插入。

```

LinkedList creat(ElemType A[],int n)
//由含n个数据的数组A生成循环链表，要求链表有序并且无值重复结点
{
    LinkedList h;
    h=(LinkedList)malloc(sizeof(LNode)); //申请结点
    h->next=h; //形成空循环链表
    for(i=0;i<n;i++)
    {
        pre=h;
        p=h->next;
        while(p!=h && p->data<A[i])
        {
            pre=p; p=p->next; } //查找A[i]的插入位置
        if(p==h || p->data!=A[i]) //重复数据不再输入
        {
            s=(LinkedList)malloc(sizeof(LNode));
            s->data=A[i]; pre->next=s; s->next=p; //将结点s链入链表中
        }
    }
}

```

```
    }  
  } //for  
  return(h);  
} 算法结束
```


四. 应用题

1. (1) G_1 最多 $n(n-1)/2$ 条边，最少 $n-1$ 条边 (2) G_2 最多 $n(n-1)$ 条边，最少 n 条边
(3) G_3 最多 $n(n-1)$ 条边，最少 $n-1$ 条边 (注：弱连通有向图指把有向图看作无向图时，仍是连通的)
2. $n-1, n$ 3. 分块对称矩阵
4. 证明：具有 n 个顶点 $n-1$ 条边的无向连通图是自由树，即没有确定根结点的树，每个结点均可当根。若边数多于 $n-1$ 条，因一条边要连接两个结点，则必因加上这一条边而使两个结点多了一条通路，即形成回路。形成回路的连通图不再是树（在图论中树定义为无回路的连通图）。
5. 证明：该有向图顶点编号的规律是让弧尾顶点的编号大于弧头顶点的编号。由于不允许从某顶点发出并回到自身顶点的弧，所以邻接矩阵主对角元素均为0。先证明该命题的充分条件。由于弧尾顶点的编号均大于弧头顶点的编号，在邻接矩阵中，非零元素 $(A[i][j]=1)$ 自然是落到下三角矩阵中；命题的必要条件是要使上三角为0，则不允许出现弧头顶点编号大于弧尾顶点编号的弧，否则，就必然存在环路。（对该类有向无环图顶点编号，应按顶点出度顺序编号。）
6. 设图的顶点个数为 $n(n \geq 0)$ ，则邻接矩阵元素个数为 n^2 ，即顶点个数的平方，与图的边数无关。
7. (1) $n(n-1), n$
(2) 10^6 ，不一定是稀疏矩阵（稀疏矩阵的定义是非零个数远小于该矩阵元素个数，且分布无规律）
(3) 使用深度优先遍历，按退出dfs过程的先后顺序记录下的顶点是逆向**拓扑**有序序列。若在执行dfs(v)未退出前，出现顶点u到v的回边，则说明存在包含顶点v和顶点u的环。
8. (1)
(2) 开始结点：（入度为0） K_1, K_2 ，终端结点（出度为0） K_6, K_7 。
(3) 拓扑序列 $K_1, K_2, K_3, K_4, K_5, K_6, K_8, K_9, K_7$
 $K_2, K_1, K_3, K_4, K_5, K_6, K_8, K_9, K_7$
规则：开始结点为 K_1 或 K_2 ，之后，若遇多个入度为0的顶点，按顶点编号顺序选择。

(4)

8 (4) 邻接表和逆邻接表

9. (1) 注：邻接矩阵下标按字母升序: abcdefghi
(2) 强连通分量：(a), (d), (h),
(b, e, i, f, c, g)
(3) 顶点a到顶点i的简单路径：
(a@b@e@i), (a@c@g@i),
(a@c@b@e@i)

10. 图G的具体存储结构略。

邻接矩阵表示法，有 n 个顶点的图占用 n^2 个元素的存储单元，与边的个数无关，当边数较少时，存储效率较低。这种结构下，对查找结点的度、第一邻接点和下一邻接点、两结点间是否有边的操作有利，对插入和删除顶点的操作不利。

邻接表表示法是顶点的向量结构与顶点的邻接点的链式存储结构相结合的结构，顶点的向量结构含有 n ($n \geq 0$) 个顶点和指向各顶点第一邻接点的指针，其顶点的邻接点的链式存储结构是根据顶点的邻接点的实际设计的。这种结构适合查找顶点及邻接点的信息，查顶点的度，增加或删除顶点和边（弧）也很方便，但因指针多占用了存储空间，另外，某两顶点间是否有边（弧）也不如邻接矩阵那么清楚。对有向图的邻接表，查顶点出度容易，而查顶点入度却困难，要遍历整个邻接表。要想查入度象查出度那样容易，就要建立逆邻接表。无向图邻接表中边结点是边数的二倍也增加了存储量。

十字链表是有向图的另一种存储结构，将邻接表和逆邻接表结合到一起，弧结点也增加了信息（至少弧尾，弧头顶点在向量中的下标及从弧尾顶点发出及再入到弧头顶点的下一条弧的四个信息）。查询顶点的出度、入度、邻接点等信息非常方便。

邻接多重表是无向图的另一种存储结构，边结点至少包括5个域：连接边的两个顶点在顶点向量中的下标，指向与该边相连接的两顶点的下一条边的指针，以及该边的标记信息（如该边是否被访问）。边结

点的个数与边的个数相同，这是邻接多重表比邻接表优越之处。

11.

已知顶点*i*，找与*i*相邻的顶点*j*的规则如下：在顶点向量中，找到顶点*i*，顺其指针找到第一个边结点（若其指针为空，则顶点*i*无邻接点）。在边结点中，取出两顶点信息，若其中有*j*，则找到顶点*j*；否则，沿从*i*发出的另一条边的指针（*ilink*）找*i*的下一邻接点。在这种查找过程中，若边结点中有*j*，则查找成功；若最后*ilink*为空，，则顶点*i*无邻接点*j*。

12. 按各顶点的出度进行排序。*n*个顶点的有向图，其顶点最大出度是*n*-1，最小出度为0。这样排序后，出度最大的顶点编号为1，出度最小的顶点编号为*n*。之后，进行调整，即若存在弧<*i*, *j*>，而顶点*j*的出度大于顶点*i*的出度，则把*j*编号在顶点*i*的编号之前。本题算法见下面算法设计第28题。

13. 采用深度优先遍历算法，在执行dfs(*v*)时，若在退出dfs(*v*)前，碰到某顶点*u*，其邻接点是已经访问的顶点*v*，则说明*v*的子孙*u*有到*v*的回边，即说明有环，否则，无环。（详见下面算法题13）

14. 深度优先遍历序列：125967384

宽度优先遍历序列：123456789

注：（1）邻接表不唯一，这里顶点的邻接点按升序排列

（2）在邻接表确定后，深度优先和宽度优先遍历序列唯一

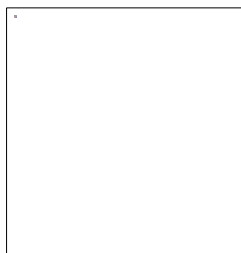
（3）这里的遍历，均从顶点1开始



15. (1) V₁V₂V₄V₃V₅V₆

(2) V₁V₂V₃V₄V₅V₆

16. (1)



16题 (3) 宽度优先生成树



(2) 深度优先生成树

为节省篇幅，生成树横画，下同。

17. 设从顶点1开始遍历，则深度优先生成树(1)和宽度优先生成树(2)为：

图17(1)

图17(2)

18. 遍历不唯一的因素有：开始遍历的顶点不同；存储结构不同；在邻接表情况下邻接点的顺序不同。

19. (1) 邻接矩阵：(6*6个元素)*2字节/元素=72字节

邻接表：表头向量6*(4+2)+边结点9*(2+2+4)*2=180字节

邻接多重表: 表头向量 $6 \times (4+2) +$ 边结点 $9 \times (2+2+2+4+4) = 162$ 字节

邻接表占用空间较多, 因为边较多, 边结点又是边数的2倍, 一般来说, 邻接矩阵所占空间与边个数无关 (不考虑压缩存储), 适合存储稠密图, 而邻接表适合存储稀疏图。邻接多重表边结点个数等于边数, 但结点中增加了一个顶点下标域和一个指针域。

(2) 因未确定存储结构, 从顶点1开始的DFS树

不唯一, 现列出两个:

20. 未确定存储结构, 其DFS树不唯一, 其中之一 (按邻接点逆序排列) 是

(2) 关节点有3, 1, 8, 7, 2

21. (1)



(2) AFEDBC

22. 设邻接表 (略) 中顶点的邻接点按顶点编号升序排列 (V1编号为1)

(1) 广度优先搜索序列: V1V2V3V4V5V6V7V8

(2) 深度优先搜索序列: V1V2V4V8V5V3V6V7

23. (1) 略 (2) V1V2V5V3V4V6 (4) V1V2V3V4V5V6

(3)

(6) 见本章五算法设计第6题

24. 设该图用邻接表存储结构存储, 顶点的邻接点按顶点编号升序排列

(1) ABGFDEC

(2) EACFBDG

(3)

25. 在有相同权值边时生成不同的MST, 在这种情况下, 用Prim或Kruskal也会生成不同的MST。

26. 无向连通图的生成树包含图中全部n个顶点, 以及足以使图连通的n-1条边。而最小生成树则是各边权值之和最小的生成树。从算法中WHILE (所剩边数 \geq 顶点数) 来看, 循环到边数比顶点数少1 (即n-1) 停止, 这符合n个顶点的连通图的生成树有n-1条边的定义; 由于边是按权值从大到小排序, 删去的边是权值大的边, 结果的生成树必是最小生成树; 算法中“若图不再连通, 则恢复ei”, 含义是必须保留使图连通的边, 这就保证了是生成树, 否则或者是有回路, 或者成了连通分量, 均不再是生成树。

27. Prim算法构造最小生成树的步骤如24题所示, 为节省篇幅, 这里仅用Kruskal算法, 构造最小生成树过程如下: (下图也可选(2, 4)代替(3, 4), (5, 6)代替(1, 5))



28. (1) 最小生成树的定义见上面26题

(2) 最小生成树有两棵。

(限于篇幅, 下面的生成树只给出顶点集合和边集合, 边以三元组 (V_i, V_j, W) 形式), 其中W代表权值。

$V(G) = \{1, 2, 3, 4, 5\}$ $E_1(G) = \{(4, 5, 2), (2, 5, 4), (2, 3, 5), (1, 2, 7)\};$

$E_2(G) = \{(4, 5, 2), (2, 4, 4), (2, 3, 5), (1, 2, 7)\}$

29. $V(G) = \{1, 2, 3, 4, 5, 6, 7\}$

$E(G) = \{(1, 6, 4), (1, 7, 6), (2, 3, 5), (2, 4, 8), (2, 5, 12), (1, 2, 18)\}$

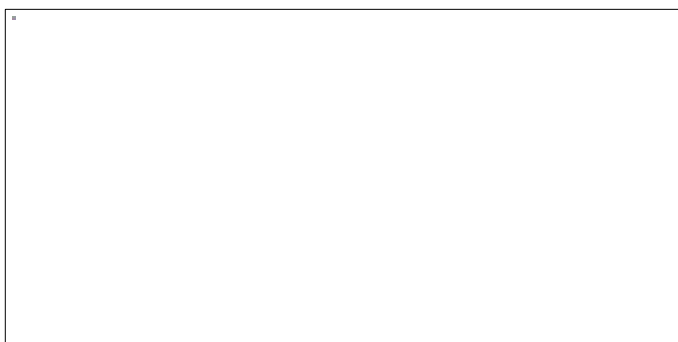
30. $V(G) = \{1, 2, 3, 4, 5, 6, 7, 8\}$
 $E(G) = \{(3, 8, 2), (4, 7, 2), (3, 4, 3), (5, 8, 3), (2, 5, 4), (6, 7, 4), (1, 2, 5)\}$
 注：(或将 $(3, 4, 3)$ 换成 $(7, 8, 3)$)
31. 设顶点集合为 $\{1, 2, 3, 4, 5, 6\}$ ，
 由右边的逻辑图可以看出，在 $\{1, 2, 3\}$ 和 $\{4, 5, 6\}$ 回路中，
 各任选两条边，加上 $(2, 4)$ ，则可构成9棵不同的最小生成树。
32. (1) 邻接矩阵略
 (2)

Y Closedge	2	3	4	5	6	7	8	U	V-U
Vex Lowcost	① 2	① 3						{1}	{2, 3, 4, 5, 6, 7, 8}
Vex Lowcost		① 3	② 2					{1, 2}	{3, 4, 5, 6, 7, 8}
Vex Lowcost		④ 1		④ 2	④ 4			{1, 2, 4}	{3, 5, 6, 7, 8}
Vex Lowcost				④ 2	④ 4			{1, 2, 4, 3}	{5, 6, 7, 8}
Vex Lowcost					⑤ 1	⑤ 2		{1, 2, 4, 3, 5}	{6, 7, 8}
Vex Lowcost						⑤ 2	⑥ 4	{1, 2, 4, 3, 5, 6}	{7, 8}
Vex Lowcost							⑦ 3	{1, 2, 4, 3, 5, 6, 7}	{8}
Vex Lowcost								{1, 2, 4, 3, 5, 6, 7, 8}	{}

33. (1) (2)

(3) 最小生成树6个顶点5条边： $V(G) = \{Pe, N, Pa, L, T, M\}$
 $E(G) = \{(L, Pa, 3), (Pe, T, 21), (M, N, 32), (L, N, 55), (L, Pe, 81)\}$

34. (1)



TYPE edgeptr = ^enode;
 enode = RECORD ivex, jvex: vtxptr; w: integer; jlink, ilink: edgeptr; END;
 vexnode = RECORD data: vertertype; firstedge: edgeptr; END;
 adjmulist = ARRAY[vexptr] OF vexnode;

- (2) 深度优先遍历序列：1, 4, 6, 5, 3, 2;
 深度优先生成树的边集合： $\{(1, 4), (4, 6), (6, 5), (5, 3), (3, 2)\}$




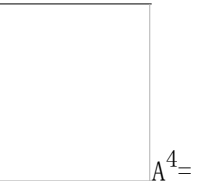


- (3) 宽度优先遍历序列：143265;
 宽度优先生成树的边集合： $\{(1, 4), (1, 3), (1, 2), (4, 6), (4, 5)\}$;

(4) 按Prim构造过程只画出最小生成树的边和权植：
 $E(G) = \{(1, 4, 3), (4, 3, 4), (3, 5, 1), (3, 2, 2), (3, 6, 10)\}$

35. $E_1(G) = \{(1, 2, 16), (2, 3, 5), (2, 6, 6), (2, 4, 11), (6, 5, 18)\}$,

$V(G) = \{1, 2, 3, 4, 5, 6\}$
 $E(G) = \{(1, 2, 16), (2, 3, 5), (3, 6, 6), (2, 4, 11), (6, 5, 18)\}$
 36.

36 (1) 邻接矩阵和邻接表
 36 (2) 深度优先搜索序列: abcdehf; 36 (3) 最小生成树
 宽度优先搜索序列: abfcdef

37. $A^0 =$  $A^1 =$  $A^2 =$  $A^3 =$  $A^4 =$ 
 38. (1)  (2) $V_1, V_2, V_4, V_6, V_3, V_5$

(3) 顶点集合 $V(G) = \{V_1, V_2, V_3, V_4, V_5, V_6\}$
 边的集合 $E(G) = \{(V_1, V_2), (V_2, V_3), (V_1, V_4), (V_4, V_5), (V_5, V_6)\}$
 (4) V_1 到 V_3 最短路径为67: $(V_1 - V_4 - V_3)$


所选 顶点	S (已确定最短路径的 点集合)	T (尚未确定最短路径的顶 点集合)	DIST										39.
			b	c	d	e	f	g	h	i	j	z	
初态	{a}	{b, c, d, e, f, g, h, i, j, z}	3	2	1	¥	¥	¥	¥	¥	¥	¥	
d	{a, d}	{b, c, e, f, g, h, i, j, z}	3	2		¥	¥	4	¥	¥	¥	¥	
c	{a, d, c}	{b, e, f, g, h, i, j, z}	3			¥	6	4	¥	¥	¥	¥	
b	{a, d, c, b}	{e, f, g, h, i, j, z}				9	6	4	¥	¥	¥	¥	
g	{a, d, c, b, g}	{e, f, h, i, j, z}				9	5		¥	10	13	¥	
f	{a, d, c, b, g, f}	{e, h, i, j, z}				9			14	7	13	¥	
i	{a, d, c, b, g, f, i}	{e, h, j, z}				9			14		11	16	
e	{a, d, c, b, g, f, i, e}	{h, j, z}							14		11	16	
j	{a, d, c, b, g, f, i, e, j}	{h, z}							14			13	
z	{a, d, c, b, g, f, i, e, j, z}	{h}							14				
h	{a, d, c, b, g, f, i, e, j, z, h}	{}											

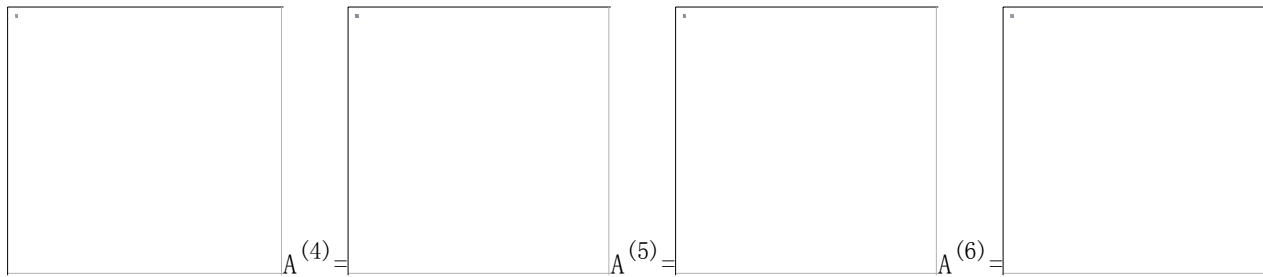
注: (1). 本表中DIST中各列最下方的数字是顶点a到顶点的最短通路, a到z为13。

(2). DIST数组的常量表达式 (即下标) 应为符号常量, 即 ‘b’, ‘c’ 等, 为节省篇幅, 用了b, c等。

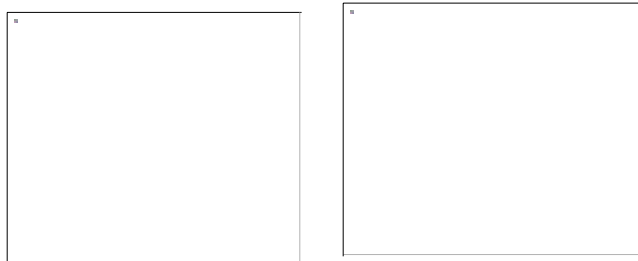
(3). 最短路径问题属于有向图, 本题给出无向图, 只能按有向完全图理解。

40. 下面用FLOYD算法求出任意两顶点的最短路径 (如图A⁽⁶⁾所示)。题目要求娱乐中心 “距其它各结点的最长往返路程最短”, 结点 V_1, V_3, V_5 和 V_6 最长往返路径最短都是9。按着 “相同条件下总的往返路径越短越好”, 选点 V_5 , 总的往返路径是34。

$A^{(0)} =$  $A^{(1)} =$  $A^{(2)} =$  $A^{(3)} =$ 



41. 顶点1到其它顶点的最短路径依次是20, 31, 28, 10, 17, 25, 35。按Dijkstra算法所选顶点依次是5, 6, 2, 7, 4, 3, 8, 其步骤原理请参见第39题, 因篇幅所限, 不再画出各步状态。
42. 顶点a到顶点b, c, d, e, f, g间的最短路径分别是15, 2, 11, 10, 6, 13。
43. 顶点A到顶点B, C, D, E的最短路径依次是3, 18, 38, 43,
按Dijkstra所选顶点过程是B, C, D, E。支撑树的边集合为{<A, B>, <B, C>, <C, D>, <B, E>}



44. (1) (2) (3) V1, V2, V5, V3, V4, V6, V8, V7, V9, V10
设一栈, 将入度为零的顶点放入栈中

45. (1) 略
(2)

- (3) 关键路径有3条, 长17。(各事件允许发生的最早时间和最晚时间略)
V1→V2→V4→V6→V8, V1→V3→V5→V7→V8, V1→V2→V4→V6→V5→V7→V8
- (4) V1结点到其它各结点的最短距离为: 2, 3, 6, 12, 10, 15, 16。

46. (1) 对有向图, 求拓扑序列步骤为:
1) 在有向图中选一个没有前驱(即入度为零)的顶点并输出。
2) 在图中删除该顶点及所有以它为尾的弧。
3) 重复1)和2), 直至全部顶点输出, 这时拓扑排序完成; 否则, 图中存在环, 拓扑排序失败。
- (2) 这里使用形式化描述方法, 当有多个顶点可以输出时, 将其按序从上往下排列, 这样不会丢掉一种拓扑序列。这里只画出从顶点1开始的所有可能的拓扑序列, 从顶点3开始的拓扑序列可类似画出。

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

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

47. 图的深度优先遍历可用于拓扑排序。带环的有向图不能拓扑排序。深度优先遍历如何发现环呢？若从有向图某顶点V出发进行遍历，在dfs(v)结束之前出现从顶点W到顶点V的回边，由于W在生成树上是V的子孙，则有向图中必定存在包含顶点V和W的环。其算法实现见下面算法题第41题。

48. (1) V1, V2, V3, V8, V5, V7, V4, V6 (2)

V1, V2, V4, V6, V3, V5, V7, V8

(3) V1到V8最短路径56，路径为

V1----V2----V5----V7----V8

(4)

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

V1到V8的关键路径是V1----V6----V5----V3----V8，长97。

49. (1)



(2) 略

(3) 深度优先遍历序列：ABCDE 广度优先遍历序列：ABCD (4) 关键路径A--B (长100)

50. AOE网的始点是入度为零的顶点，该顶点所代表的事件无任何先决条件，可首先开始。终点是出度为零的顶点，表示一项工程的结束。正常的AOE网只有一个始点和一个终点。

51. (1) AOE网如右图

图中虚线表示在时间上前后工序之间仅是接续顺序关系不存在依赖关系。顶点代表事件，弧代表活动，弧上的权代表活动持续时间。题中顶点1代表工程开始事件，顶点11代表工程结束事件。

(2) 各事件发生的最早和最晚时间如下表

事 件	1	2	3	4	5	6	7	8	9	10	11	12
最早发生时间	0	15	10	65	50	80	200	380	395	425	445	420
最晚发生时间	0	15	57	65	380	80	335	380	395	425	445	425

(3) 关键路径为顶点序列：1->2->4->6->8->9->10->11；

事件序列：A->C->E->G->H->L->M，完成工程所需的最短时间为445。

52

顶点	α	A	B	C	D	E	F	G	H	W
$Ve(i)$	0	1	6	3	4	24	13	39	22	52
$Vl(i)$	0	29	24	3	7	31	13	39	22	52

活动	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12	a13	a14	a15	a16	a17
$e(i)$	0	0	0	0	1	6	6	3	3	4	24	13	13	13	39	22	22
$l(i)$	28	18	0	3	29	24	31	34	3	7	31	20	36	13	39	22	40

关键路径是：

活动与顶点的对照表： $a1 < \alpha, A>$ $a2 < \alpha, B>$ $a3 < \alpha, C>$ $a4 < \alpha, D>$ $a5 < A, E>$ $a6 < B, E>$ $a7 < B, W>$ $a8 < C, G>$
 $a9 < C, F>$ $a10 < D, F>$ $a11 < E, G>$ $a12 < F, E>$ $a13 < F, W>$ $a14 < F, H>$ $a15 < G, W>$ $a16 < H, G>$ $a17 < H, W>$

53. A. 0—2—6—9—11 B. 20 C. 1, 5, 7 D. 各2 天 E. 0

五. 算法设计题

1. void CreatGraph (AdjList g)

//建立有n个顶点和m 条边的无向图的邻接表存储结构

{int n,m;

scanf ("%d%d",&n,&m);

for (i =1, i<=n; i++)//输入顶点信息，建立顶点向量

{scanf (&g[i].vertex); g[i].firstarc=null;}

for (k=1;k<=m;k++)//输入边信息

{scanf (&v1,&v2); //输入两个顶点

i=GraphLocateVertex (g,v1); j=GraphLocateVertex (g,v2); //顶点定位

p=(ArcNode *)malloc(sizeof(ArcNode)); //申请边结点

p->adjvex=j; p->next=g[i].firstarc; g[i].firstarc=p; //将边结点链入

p=(ArcNode *)malloc(sizeof(ArcNode));

p->adjvex=i; p->next=g[j].firstarc; g[j].firstarc=p;

}

//算法CreatGraph结束

2. void CreatAdjList(AdjList g)

//建立有向图的邻接表存储结构

{int n;

scanf ("%d",&n);

for (i=1; i<=n; i++)

{scanf (&g[i].vertex); g[i].firstarc=null; } //输入顶点信息

scanf (&v1, &v2);

while (v1 && v2) //题目要求两顶点之一为0表示结束

{i=GraphLocateVertex (g,v1);

p=(ArcNode*)malloc(sizeof(ArcNode));

p->adjvex=j; p->next=g[i].firstarc; g[i].firstarc=p;

scanf (&v1,&v2);

}

3. void CreatMGraph(AdjMList g)

//建立有n个顶点e条边的无向图的邻接多重表的存储结构

{int n,e;

scanf ("%d%d",&n,&e);

for(i=1, i<=n; i++) //建立顶点向量

{ scanf (&g[i].vertex); g[i].firstedge=null; }

for(k=1;k<=e;k++) //建立边结点

{scanf (&v1,&v2);

i=GraphLocateVertex(g,v1); j=GraphLocateVertex(g,v2);

p=(ENode *)malloc(sizeof(ENode));

p->ivex=i; p->jvex=j; p->ilink=g[i].firstedge; p->jlink=g[j].firstedge;

g[i].firstedge=p; g[j].firstedge=p;

}

//算法结束

4. void CreatOrthList(OrthList g)

//建立有向图的十字链表存储结构

{int i, j, v; //假定权值为整型

scanf ("%d",&n);

for(i=1, i<=n; i++) //建立顶点向量

{ scanf (&g[i].vertex); g[i].firstin=null; g[i].firstout=null; }

scanf ("%d%d%d",&i,&j,&v);

while (i && j && v) //当输入i, j, v之一为0时，结束算法运行

{p=(OrthNode *)malloc(sizeof(OrthNode)); //申请结点

p->headvex=j; p->tailvex=i; p->weight=v; //弧结点中权值域

p->headlink=g[j].firstin; g[j].firstin=p;

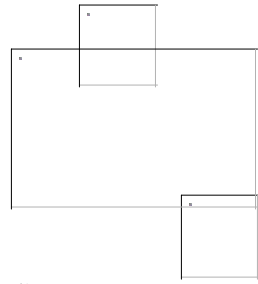
p->tailink=g[i].firstout; g[i].firstout=p;

scanf ("%d%d%d",&i,&j,&v);

} } 算法结束

[算法讨论] 本题已假定输入的i和j是顶点号，否则，顶点的信息要输入，且用顶点定位函数求出顶点在顶点向量中的下标。图建立时，若已知边数（如上面1和2题），可以用for循环；若不知边数，可用while循环（如本题），规定输入特殊数（如本题的零值）时结束运行。本题中数值设为整型，否则应以和数值类型相容的方式输入。

5. void InvertAdjList(AdjList gin, gout)



```

//将有向图的出度邻接表改为按入度建立的逆邻接表
{for (i=1;i<=n;i++)//设有向图有n个顶点, 建立逆邻接表的顶点向量。
{gin[i].vertex=gout[i].vertex; gin.firstarc=null; }
for (i=1;i<=n;i++) //邻接表转为逆邻接表。
{p=gout[i].firstarc;//取指向邻接表的指针。
while (p!=null)
{ j=p->adjvex;
s=(ArcNode *)malloc(sizeof(ArcNode));//申请结点空间。
s->adjvex=i; s->next=gin[j].firstarc; gin[j].firstarc=s;
p=p->next;//下一个邻接点。
} //while
} //for
}

6. void AdjListToAdjMatrix(AdjList gl, AdjMatrix gm)
//将图的邻接表表示转换为邻接矩阵表示。
{for (i=1;i<=n;i++) //设图有n个顶点, 邻接矩阵初始化。
for (j=1;j<=n;j++) gm[i][j]=0;
for (i=1;i<=n;i++)
{p=gl[i].firstarc; //取第一个邻接点。
while (p!=null) {gm[i][p->adjvex]=1;p=p->next; } //下一个邻接点
} //for } //算法结束

7. void AdjMatrixToAdjList( AdjMatrix gm, AdjList gl )
//将图的邻接矩阵表示法转换为邻接表表示法。
{for (i=1;i<=n;i++) //邻接表表头向量初始化。
{scanf(&gl[i].vertex); gl[i].firstarc=null;}
for (i=1;i<=n;i++)
for (j=1;j<=n;j++)
if (gm[i][j]==1)
{p=(ArcNode *)malloc(sizeof(ArcNode)) ;//申请结点空间。
p->adjvex=j;//顶点i的邻接点是j
p->next=gl[i].firstarc; gl[i].firstarc=p; //链入顶点i的邻接点链表中
}
} //end

```

[算法讨论] 算法中邻接表中顶点在向量表中的下标与其在邻接矩阵中的行号相同。

8. [题目分析] 在有向图中, 判断顶点 V_i 和顶点 V_j 间是否有路径, 可采用遍历的方法, 从顶点 V_i 出发, 不论是深度优先遍历 (dfs) 还是宽度优先遍历 (bfs), 在未退出dfs或bfs前, 若访问到 V_j , 则说明有通路, 否则无通路。设一全程变量flag。初始化为0, 若有通路, 则flag=1。

算法1: **int** visited[]=0; //全局变量, 访问数组初始化

```

int dfs(AdjList g, vi)
//以邻接表为存储结构的有向图g, 判断顶点 $V_i$ 到 $V_j$ 是否有通路, 返回1或0表示有或无
{ visited[vi]=1; //visited是访问数组, 设顶点的信息就是顶点编号。
p=g[vi].firstarc; //第一个邻接点。
while ( p!=null)
{ j=p->adjvex;
if (vj==j) { flag=1; return (1) ;} //vi 和 vj 有通路。
if (visited[j]==0) dfs(g, j);
p=p->next; } //while
if (!flag) return(0);
} //结束

```

[算法讨论] 若顶点 v_i 和 v_j 不是编号, 必须先用顶点定位函数, 查出其在邻接表顶点向量中的下标i和j。

下面算法2输出 v_i 到 v_j 的路径, 其思想是用一个栈存放遍历的顶点, 遇到顶点 v_j 时输出路径。

算法2: **void** dfs(AdjList g, **int** i)

//有向图g的顶点 v_i (编号i) 和顶点 v_j (编号j) 间是否有路径, 如有, 则输出。

```

{int top=0, stack[]; //stack是存放顶点编号的栈
visited[i]=1; //visited 数组在进入dfs前已初始化。
stack[++top]=i;
p=g[i].firstarc; //求第一个邻接点。
while (p)
{if (p->adjvex==j)
{stack[++top]=j; printf( "顶点 vi 和 vj 的路径为: \n");
for (i=1; i<=top; i++) printf( "%4d", stack[i]); exit(0);
} //if
else if (visited[p->adjvex]==0) {dfs(g, p->adjvex); top--; p=p->next;} //else if
}
}

```

```

    }//while
} //结束算法2
算法3: 本题用非递归算法求解。
int Connectij (AdjList g , vertype vi , vj )
//判断n个顶点以邻接表表示的有向图g中, 顶点 Vi 各Vj 是否有路径, 有则返回1, 否则返回0。
{ for (i=1;i<=n;i++) visited[i]=0; //访问标记数组初始化。
  i=GraphLocateVertex(g,vi); //顶点定位, 不考虑 vi或 vj不在图中的情况。
  j=GraphLocateVertex(g,vj);
  int stack[],top=0;stack[++top]=i;
  while(top>0)
  {k=stack[top--]; p=g[k].firstarc;
   while(p!=null && visited[p->adjvex]==1) p=p->next; //查第k个链表中第一个未访问的弧结点。
   if(p==null) top--;
   else {i=p->adjvex;
         if(i==j) return(1); //顶点vi和vj 间有路径。
         else {visited[i]=1; stack[++top]=i;} //else
        } //else
   } while
  return(0); } //顶点vi和vj 间无通路。

```

```

9. void DeletEdge(AdjList g,int i,j)
//在用邻接表方式存储的无向图g中, 删除边(i,j)
{p=g[i].firstarc; pre=null; //删顶点i 的边结点(i,j),pre是前驱指针
  while (p)
  {if (p->adjvex==j)
    {if(pre==null)g[i].firstarc=p->next;else pre->next=p->next;free(p);} //释放结点空间。
    else {pre=p; p=p->next;} //沿链表继续查找
    p=g[j].firstarc; pre=null; //删顶点j 的边结点(j,i)
    while (p)
    {if (p->adjvex==i)
      {if(pre==null)g[j].firstarc=p->next;else pre->next=p->next;free(p);} //释放结点空间。
      else {pre=p; p=p->next;} //沿链表继续查找
    } // DeletEdge
}

```

[算法讨论] 算法中假定给的i, j 均存在, 否则应检查其合法性。若未给顶点编号, 而给出顶点信息, 则先用顶点定位函数求出其在邻接表顶点向量中的下标i和j。

```

10. void DeleteArc (AdjList g,vertype vi,vj)
//删除以邻接表存储的有向图g的一条弧<vi,vj>, 假定顶点vi和vj存在
{ i=GraphLocateVertex(g,vi); j=GraphLocateVertex(g,vj); //顶点定位
  p=g[i].firstarc; pre=null;
  while (p)
  {if (p->adjvex==j)
    {if(pre==null) g[i].firstarc=p->next;else pre->next=p->next;free(p);} //释放结点空间。
    else { pre=p; p=p->next;}
  } //结束
}

```

```

11. void InsertArc ( OrthList g ,vertype vi,vj)
//在以十字链表示的有向图g中插入弧<vi,vj>
{ i=GraphLocateVertex(g,vi); //顶点定位。
  j=GraphLocateVertex(g,vj);
  p=(OrArcNode *)malloc(sizeof(OrArcNode));
  p->headvex=j; p->tailvex=i; //填写弧结点信息并插入十字链表。
  p->headlink=g[j].firstin; g[j].firstin=p;
  p->taillink=g[i].firstout; g[i].firstout=p;
} //算法结束

```

12. [题目分析]在有向图的邻接表中, 求顶点的出度容易, 只要简单在该顶点的邻接点链表中查结点个数即可。而求顶点的入度, 则要遍历整个邻接表。

```

int count (AdjList g , int k )
//在n个顶点以邻接表表示的有向图g中, 求指定顶点k(1<=k<=n)的入度。
{ int count =0;
  for (i=1;i<=n;i++) //求顶点k的入度要遍历整个邻接表。
  { if(i!=k) //顶点k的邻接链表不必计算
    {p=g[i].firstarc; //取顶点 i 的邻接表。
     while (p)
     {if (p->adjvex==k) count++;

```

```

        p=p->next;
    }//while
} //if
return(count); //顶点k的入度.
}

```

13. [题目分析]有向图判断回路要比无向图复杂。利用深度优先遍历, 将顶点分成三类: 未访问; 已访问但其邻接点未访问完; 已访问且其邻接点已访问完。下面用0, 1, 2表示这三种状态。前面已提到, 若dfs(v)结束前出现顶点u到v的回边, 则图中必有包含顶点v和u的回路。对应程序中v的状态为1, 而u是正访问的顶点, 若我们找出u的下一邻接点的状态为1, 就可以输出回路了。

```

void Print(int v,int start) //输出从顶点start开始的回路。
{
    for(i=1;i<=n;i++)
        if(g[v][i]!=0 && visited[i]==1) //若存在边(v,i), 且顶点i的状态为1。
            {printf("%d",v); if(i==start) printf("\n"); else Print(i,start);break;} //if
} //Print
void dfs(int v)
{
    visited[v]=1;
    for(j=1;j<=n;j++)
        if (g[v][j]!=0) //存在边(v,j)
            if (visited[j]!=1) {if (!visited[j]) dfs(j); } //if
            else {cycle=1; Print(j,j);}
    visited[v]=2;
} //dfs
void find_cycle() //判断是否有回路, 有则输出邻接矩阵。visited数组为全局变量。
{
    for (i=1;i<=n;i++) visited[i]=0;
    for (i=1;i<=n;i++) if (!visited[i]) dfs(i);
} //find_cycle

```

14. [题目分析]有几种方法判断有向图是否存在环路, 这里使用拓扑排序法。对有向图的顶点进行拓扑排序, 若拓扑排序成功, 则无环路; 否则, 存在环路。题目已假定有向图用十字链表存储, 为方便运算, 在顶点结点中, 再增加一个入度域indegree, 存放顶点的入度。入度为零的顶点先输出。为节省空间, 入度域还起栈的作用。值得注意的是, 在邻接表中, 顶点的邻接点非常清楚, 顶点的单链表中的邻接点域都是顶点的邻接点。由于十字链表边(弧)结点个数与边(弧)个数相同(不象无向图边结点个数是边的二倍), 因此, 对某顶点v, 要判断其邻接点是headvex还是tailvex。

```

int Topsor(OrthList g)
//判断以十字链表为存储结构的有向图g是否存在环路, 如是, 返回1, 否则, 返回0。
{
    int top=0; //用作栈顶指针
    for (i=1;i<=n;i++) //求各顶点的入度。设有向图g有n个顶点, 初始时入度域均为0
        {p=g[i].firstin; //设顶点信息就是顶点编号, 否则, 要进行顶点定位
         while(p)
             {g[i].indegree++; //入度域增1
              if (p->headvex==i) p=p->headlink; else p=p->taillink; //找顶点i的邻接点
             } //while(p) } //for
    for (i=1;i<=n;i++) //建立入度为0的顶点的栈
        if (g[i].indegree==0) {g[i].indegree=top; top=i; }
    m=0; //m为计数器, 记输出顶点个数
    while (top<>0)
        {i=top; top=g[top].indegree; m++; //top指向下一入度为0的顶点
         p=g[i].firstout;
         while (p) //处理顶点i的各邻接点的入度
             {if (p->tailvex==i) k=p->headvex; else k=p->tailvex; } //找顶点i的邻接点
              g[k].indegree--; //邻接点入度减1
              if (g[k].indegree==0) {g[k].indegree=top; top=k; } //入度为0的顶点再入栈
              if (p->headvex==i) p=p->headlink; else p=p->taillink; //找顶点i的下一邻接点
             } //while (p)
        } // while (top<>0)
    if (m<n) return(1); //有向图存在环路
    else return(0); //有向图无环路
} //算法结束

```

15. int FirstAdj(AdjMuList g, vertype v)
 //在邻接多重表g中, 求v的第一邻接点, 若存在, 返回第一邻接点, 否则, 返回0。
 {i=GraphLocateVertex(g,v); //确定顶点v在邻接多重表向量中的下标, 不考虑不存在v的情况。
 p=g[i].firstedge; //取第一个边结点。
 if (p==null) return (0);

```

    else {if (ivex==i) return (jvex); else return (ivex);}
    //返回第一邻接点，ivex和jvex中必有一个等于i
} // FirstAdj

```

16. [题目分析] 本题应使用深度优先遍历，从主调函数进入dfs(v)时，开始记数，若退出dfs()前，已访问完有向图的全部顶点（设为n个），则有向图有根，v为根结点。将n个顶点从1到n编号，各调用一次dfs()过程，就可以求出全部的根结点。题中有向图的邻接表存储结构、记顶点个数的变量、以及访问标记数组等均设计为全局变量。建立有向图g的邻接表存储结构参见上面第2题，这里只给出判断有向图是否有根的算法。

```

int num=0, visited[]=0 //num记访问顶点个数，访问数组visited初始化。

```

```

const n=用户定义的顶点数;

```

```

AdjList g; //用邻接表作存储结构的有向图g。

```

```

void dfs(v)

```

```

{visited[v]=1; num++; //访问的顶点数+1

```

```

if (num==n) {printf(“%d是有向图的根。\\n”,v); num=0;} //if

```

```

p=g[v].firstarc;

```

```

while (p)

```

```

{if (visited[p->adjvex]==0) dfs (p->adjvex);

```

```

p=p->next;} //while

```

```

visited[v]=0; num--; //恢复顶点v

```

```

} //dfs

```

```

void JudgeRoot()

```

```

//判断有向图是否有根，有根则输出之。

```

```

{static int i;

```

```

for (i=1;i<n;i++) //从每个顶点出发，调用dfs()各一次。

```

```

{num=0; visited[1..n]=0; dfs(i); }

```

```

} // JudgeRoot

```

算法中打印根时，输出顶点在邻接表中的序号（下标），若要输出顶点信息，可使用g[i].vertex。

17. [题目分析] 使用图的遍历可以求出图的连通分量。进入dfs或bfs一次，就可以访问到图的一个连通分量的所有顶点。

```

void dfs ()

```

```

{visited[v]=1; printf ( “%3d”,v); //输出连通分量的顶点。

```

```

p=g[v].firstarc;

```

```

while (p!=null)

```

```

{if (visited[p->adjvex]==0) dfs(p->adjvex);

```

```

p=p->next;

```

```

} //while

```

```

} // dfs

```

```

void Count()

```

```

//求图中连通分量的个数

```

```

{int k=0; static AdjList g; //设无向图g有n个结点

```

```

for (i=1;i<n;i++)

```

```

if (visited[i]==0) { printf (“\\n第%d个连通分量:\\n”,++k); dfs(i);} //if

```

```

} //Count

```

算法中visited[]数组是全程变量，每个连通分量的顶点集按遍历顺序输出。这里设顶点信息就是顶点编号，否则应取其g[i].vertex分量输出。

18. void bfs(AdjList GL, vertype v)

```

//从v发广度优先遍历以邻接表为存储结构的无向图GL。

```

```

{visited[v]=1;

```

```

printf( “%3d”,v); //输出第一个遍历的顶点。

```

```

QueueInit(Q); QueueIn(Q,v); //先置空队列，然后第一个顶点v入队列，设队列容量足够大

```

```

while (!QueueEmpty(Q))

```

```

{v=QueueOut(Q); p=GL[v].firstarc; //GL是全局变量，v入队列。

```

```

while (p!=null)

```

```

{if(visited[p->adjvex]==0)

```

```

{printf(“%3d”,p->adjvex); visited[p->adjvex]=1; QueueIn(Q,p->adjvex);} //if

```

```

p=p->next;

```

```

} //while

```

```

} // while (!QueueEmpty(Q))

```

```

} //bfs

```

```

void BFSCOM()

```

```

//广度优先搜索，求无向图G的连通分量。

```

```

{ int count=0; //记连通分量个数。

```

```

for (i=1;i<n;i++) visited[i]=0;

```

```

    for (i=1;i<=n;i++)
        if (visited[i]==0) {printf("\n第%d个连通分量:\n", ++count); bfs(i);} //if
    } //BFSCOM

```

19. 请参见上题18。HEAD, MARK, VER, LINK相当于上题GL, visited, adjvex, next。

20. void Traver(AdjList g, vertype v)
//图g以邻接表为存储结构, 算法从顶点v开始实现非递归深度优先遍历。

```

{struct arc *stack[];
 visited[v]=1; printf(v); //输出顶点v
 top=0; p=g[v].firstarc; stack[++top]=p;
 while(top>0 || p!=null)
 {while (p)
     if (p && visited[p->adjvex]) p=p->next;
     else {printf(p->adjvex); visited[p->adjvex]=1;
           stack[++top]=p; p=g[p->adjvex].firstarc;
           } //else
     if (top>0) {p=stack[top--]; p=p->next; }
 } //while } //算法结束。

```

[算法讨论] 以上算法适合连通图, 若是非连通图, 则再增加一个主调算法, 其核心语句是

```
for (vi=1;vi<=n;vi++) if (!visited[vi]) Traver(g, vi);
```

21. (1) 限于篇幅, 邻接表略。

(2) 在邻接点按升序排列的前提下, 其dfs和bfs序列分别为BADCEF和BACEDF。

(3) void dfs(v)

```

{i=GraphLocateVertex(g, v); //定位顶点
 visited[i]=1; printf(v); //输出顶点信息
 p=g[i].firstarc;
 while (p)
 { if (visited[p->adjvex]==0) dfs(g[p->adjvex].vertex);
   p=p->next;
 } //while
} //dfs

```

void traver()

//深度优先搜索的递归程序; 以邻接表表示的图g是全局变量。

```

{ for (i=1;i<=n;i++) visited[i]=0; //访问数组是全局变量初始化。
  for (vi=v1;vi<=vn;vi++)
      if (visited[GraphLocateVertex(g, vi)]==0) dfs(vi);
} //算法结束。

```

22. [题目分析] 强连通图指从任一顶点出发, 均可访问到其它所有顶点, 故这里只给出dfs()过程。

PROC dfs(g:AdjList, v0:vtxptr)

//从v0出发, 深度优先遍历以邻接表为存储结构的强连通图g。

TYPE stack=ARRAY[1..n] OF arcptr; //栈元素为弧结点指针, n为顶点个数。

s:stack; top:integer; top:=0

visited[v0]:=1;

write(v0:4); //设顶点信息就是顶点编号; 否则要顶点定位。

p:=g[v0].firstarc;

WHILE (top>0 || p!=NIL) DO

BEGIN WHILE (p!= NIL) DO

IF (visited[p^.adjvex]=1) THEN p:=p^.next; //查未访问的邻接点。

ELSE BEGIN w:=p^.adjvex; visited[w]:=1; top:=top+1; s[top]:=p;
 p:=g[w].firstarc ;

END; //深度优先遍历。

IF (top>0) THEN BEGIN p:=s[top]; top:=top-1; p:=p^.next END;

END;

ENDP;

23. [题目分析] 本题是对无向图G的深度优先遍历, dfs算法上面已有几处(见20-22)。这里仅给出将连通分量的顶点用括号括起来的算法。为了得出题中的输出结果, 各顶点的邻接点要按升序排列。

void Traver()

```
{for (i=1;i<=nodes(g);i++) visited[i]=0; //visited是全局变量, 初始化。
```

```
for (i=1;i<=nodes(g);i++)
```

```
    if (visited[i]==0) {printf("("); dfs(i); printf(")");} //if
```

```
} //Traver
```

24. void visit(vertype v) //访问图的顶点v。

void initqueue (vertype Q[]) //图的顶点队列Q初始化。

```

void enqueue (vertype Q[] ,v) //顶点v入队列Q。
vertype delqueue (vertype Q[]) //出队操作。
int empty (Q) //判断队列是否为空的函数，若空返回1，否则返回0。
vertype firstadj(graph g ,vertype v)//图g中v的第一个邻接点。
vertype nextadj(graph g ,vertype v ,vertype w)//图g中顶点v的邻接点中在w后的邻接点
void bfs (graph g ,vertype v0)
//利用上面定义的图的抽象数据类型，从顶点v0出发广度优先遍历图g。
{visit(v0);
visited[v0]=1; //设顶点信息就是编号，visited是全局变量。
initqueue(Q); enqueue(Q,v0); //v0入队列。
while (!empty(Q))
{v=delqueue(Q); //队头元素出队列。
w=firstadj(g ,v); //求顶点v的第一邻接点
while (w!=0) //w!=0表示w存在。
{if (visited[w]==0) //若邻接点未访问。
{visit(w); visited[w]=1; enqueue(Q,w);} //if
w=nextadj(g,v,w); //求下一个邻接点。
} //while } //while
} //bfs
void Traver()
//对图g进行宽度优先遍历，图g是全局变量。
{for (i=1;i<=n;i++) visited[i]=0;
for (i=1;i<=n;i++)
if (visited[i]==0) bfs(i);
} //Traver

```

25. [题目分析] 本题应使用深度优先遍历。设图的顶点信息就是顶点编号，用num记录访问顶点个数，当num等于图的顶点个数（题中的NODES(G)），输出所访问的顶点序列，顶点序列存在path中，path和visited数组，顶点计数器num，均是全局变量，都已初始化。

```

void SPathdfs(v0)
//判断无向图G中是否存在以v0为起点，包含图中全部顶点的简单路径。
{visited[v0]=1; path[++num]=v0;
if (num==nodes(G) //有一条简单路径，输出之。
{for (i=1;i<=num;i++) printf( "%3d",path[i]); printf( "\n"); exit(0);} //if
p=g[v0].firstarc;
while (p)
{if (visited[p->adjvex]==0) SPathdfs(p->adjvex); //深度优先遍历。
p=p->next; //下一个邻接点。
} //while
visited[v0]=0; num--; //取消访问标记，使该顶点可重新使用。
} //SPathdfs

```

26. 与上题类似，这里给出非递归算法，顶点信息仍是编号。

```

void AllSPdfs(AdjList g,vertype u,vertype v)
//求有向图g中顶点u到顶点v的所有简单路径，初始调用形式：AllSPdfs(g,u,v)
{ int top=0,s[];
s[++top]=u; visited[u]=1;
while (top>0 || p)
{p=g[s[top]].firstarc; //第一个邻接点。
while (p!=null && visited[p->adjvex]==1) p=p->next; //下一个访问邻接点表。
if (p==null) top--; //退栈。
else { i=p->adjvex; //取邻接点（编号）。
if (i==v) //找到从u到v的一条简单路径，输出。
{for (k=1;k<=top;k++) printf( "%3d",s[k]); printf( "%3d\n",v);} //if
else { visited[i]=1; s[++top]=i; } //else深度优先遍历。
} //else } //while
} // AllSPdfs

```

类似本题的另外叙述的第（2）题：u到v有三条简单路径：uabfv, ucdv, ucdefv。

27. （1）[题目分析] D_搜索类似BFS，只是用栈代替队列，入出队列改为入出栈。查某顶点的邻接点时，若其邻接点尚未遍历，则遍历之，并将其压入栈中。当一个顶点的所有邻接点被搜索后，栈顶顶点是下一个搜索出发点。

```

void D_BFS(AdjList g ,vertype v0)
// 从v0顶点开始，对以邻接表为存储结构的图g进行D_搜索。
{ int s[], top=0; //栈，栈中元素为顶点，仍假定顶点用编号表示。

```

```

for (i=1,i<=n;i++) visited[i]=0; //图有n个顶点，visited数组为全局变量。
for (i=1,i<=n;i++) //对n个顶点的图g进行D_搜索。
    if (visited[i]==0)
        {s[++top]=i; visited[i]=1; printf( "%3d",i);
         while (top>0)
             {i=s[top--]; //退栈
              p=g[i].firstarc; //取第一个邻接点
              while (p!=null) //处理顶点的所有邻接点
                  {j=p->adjvex;
                   if (visited[j]==0) //未访问的邻接点访问并入栈。
                       {visited[j]=1; printf( "%3d",i);s[++top]=j;}
                   p=p->next;
                  } //下一个邻接点
              }//while(top>0)
            } //if
        } //D_BFS

```

(2) D_搜索序列：1234675，生成树如图：

28, [题目分析] 本题的含义是对有向无环图(DAG)的顶点，以整数适当编号后，可使其邻接矩阵中对角线以下元素全部为零。根据这一要求，可以按各顶点出度大小排序，使出度最大的顶点编号为1，出度次大者编号为2，出度为零的顶点编号为n。这样编号后，可能出现顶点编号*i*<*j*，但却有一条从顶点到*j*到*i*的弧。这时应进行调整，即检查每一条弧，若有<*i*, *j*>，且*i*>*j*，则使顶点*j*的编号在顶点*i*的编号之前。

```
void Adjust(AdjMatrix g1 ,AdjMatrix g2)
```

//对以邻接矩阵存储的DAG图g1重新编号，使若有<*i*, *j*>，则编号*i*<*j*，重新编号后的图以邻接矩阵g2存储。

```

typedef struct { int vertex ,out ,count }node ; //结点结构： 顶点， 出度， 计数。
node v[]; //顶点元素数组。
int c[]; //中间变量
for (i=1;i<=n;i++) //顶点信息数组初始化，设图有n个顶点。
    {v[i].vertex=i; v[i].out=0; v[i].count=1; c[i]=1;} //count=1为最小
for (i=1;i<=n;i++) //计算每个顶点的出度。
    for (j=1;j<=n;j++) v[i].out+=g1[i][j];
for (i=n;i>=2;i--) //对v的出度域进行计数排序，出度大者，count域中值小。
    for (j=i-1;j>=1;j--)
        if (v[i].count<=v[j].count) v[i].count++; else v[j].count++;
for (i=1;i<=n;i++) //第二次调整编号。若<i, j>且i>j，则顶点j的编号在顶点i的编号之前
    for (j=i;j<=n;j++)
        if (g1[i][j]==1 && v[i].count>v[j].count) {v[i].count=v[j].count;v[j].count++;}
for (i=n;i>=2;i--) //对v的计数域v[i].count排序，按count域从小到大顺序存到数组c中。
    for (j=i-1;j>=1;j--)
        if (v[i].count<v[j].count) c[j]++; else c[i]++;
for (i=1;i<=n;i++) v[i].count=c[i]; //将最终编号存入count 域中。
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        g2[v[i].count][v[j].count]=g1[v[i].vertex][v[j].vertex];
} //算法结束

```

29.



[题目分析] 将顶点放在两个集合V1和V2。对每个顶点，检查其和邻接点是否在同一个集合中，如是，则为非二部图。为此，用整数1和2表示两个集合。再用一队列结构存放图中访问的顶点。

```
int BPGraph (AdjMatrix g)
```

//判断以邻接矩阵表示的图g是否是二部图。

```

{int s[]; //顶点向量，元素值表示其属于那个集合（值1和2表示两个集合）
 int Q[]; //Q为队列，元素为图的顶点，这里设顶点信息就是顶点编号。
 int f=0,r,visited[]; //f和r分别是队列的头尾指针，visited[]是访问数组
 for (i=1;i<=n;i++) {visited[i]=0;s[i]=0;} //初始化，各顶点未确定属于那个集合
 Q[1]=1; r=1; s[1]=1; //顶点1放入集合S1
 while(f<r)
     {v=Q[++f]; if (s[v]==1) jh=2; else jh=1; //准备v的邻接点的集合号
     if (!visited[v])
         {visited[v]=1; //确保对每一个顶点，都要检查与其邻接点不应在一个集合中
          for (j=1,j<=n;j++)

```

```

        if (g[v][j]==1){if (!s[j]) {s[j]=jh; Q[++r]=j;} //邻接点入队列
        else if (s[j]==s[v]) return(0);} //非二部图
    } //if (!visited[v])
} //while
return(1); } //是二部图

```

[算法讨论] 题目给的是连通无向图, 若非连通, 则算法要修改。

30. [题目分析] 连通图的生成树包括图中的全部n个顶点和足以使图连通的n-1条边, 最小生成树是边上权值之和最小的生成树。故可按权值从大到小对边进行排序, 然后从大到小将边删除。每删除一条当前权值最大的边后, 就去测试图是否仍连通, 若不再连通, 则将该边恢复。若仍连通, 继续向下删; 直到剩n-1条边为止。

```

void SpnTree (AdjList g)
//用“破圈法”求解带权连通无向图的一棵最小代价生成树。
{typedef struct {int i, j, w} node; //设顶点信息就是顶点编号, 权是整型数
node edge[];
scanf( "%d%d", &e, &n) ; //输入边数和顶点数。
for (i=1; i<=e; i++) //输入e条边: 顶点, 权值。
    scanf("%d%d%d", &edge[i].i, &edge[i].j, &edge[i].w);
for (i=2; i<=e; i++) //按边上的权值大小, 对边进行逆序排序。
    {edge[0]=edge[i]; j=i-1;
    while (edge[j].w<edge[0].w) edge[j+1]=edge[j--];
    edge[j+1]=edge[0]; } //for
k=1; eg=e;
while (eg>n) //破圈, 直到边数e=n-1.
    {if (connect(k)) //删除第k条边若仍连通。
        {edge[k].w=0; eg--; } //测试下一条边edge[k], 权值置0表示该边被删除
        k++; //下条边
    } //while
} //算法结束。

```

connect() 是测试图是否连通的函数, 可用图的遍历实现, 若是连通图, 一次进入dfs或bfs就可遍历全部结点, 否则, 因为删除该边而使原连通图成为两个连通分量时, 该边不应删除。前面第17, 18题就是测连通分量个数的算法, 请参考。“破圈”结束后, 可输出edge中w不为0的n-1条边。限于篇幅, 这些算法不再写出。

31. [题目分析] 求单源点最短路径问题, 存储结构用邻接表表示, 这里先给出所用的邻接表中的边结点的定义: struct node {int adjvex, weight; struct node *next;}p;

```

void Shortest_Dijkstra(AdjList cost, vtype v0)
//在带权邻接表cost中, 用Dijkstra方法求从顶点v0到其它顶点的最短路径。
{int dist[], s[]; //dist数组存放最短路径, s数组存顶点是否找到最短路径的信息。
for (i=1; i<=n; i++) {dist[i]=INFINITY; s[i]=0; } //初始化, INFINITY是机器中最大的数
s[v0]=1;
p=g[v0].firstarc;
while(p) //顶点的最短路径赋初值
    {dist[p->adjvex]=p->weight; p=p->next;}
for (i=1; i<n; i++) //在尚未确定最短路径的顶点集中选有最短路径的顶点u。
    {mindis=INFINITY; //INFINITY是机器中最大的数, 代表无穷大
    for (j=1; j<=n; j++)
        if (s[j]==0 && dist[j]<mindis) {u=j; mindis=dist[j];} //if
    s[u]=1; //顶点u已找到最短路径。
    p=g[u].firstarc;
    while(p) //修改从v0到其它顶点的最短路径
        {j=p->adjvex;
        if (s[j]==0 && dist[j]>dist[u]+p->weight) dist[j]=dist[u]+p->weight;
        p=p->next;
        }
    } // for (i=1; i<n; i++)
} //Shortest_Dijkstra

```

32. 本题用FLOYD算法直接求解如下:

```

void ShortPath_FLOYD(AdjMatrix g)
//求具有n个顶点的有向图每对顶点间的最短路径
{AdjMatrix length; //length[i][j]存放顶点vi到vj的最短路径长度。
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++) length[i][j]=g[i][j]; //初始化。
for (k=1; k<=n; k++)
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)

```

```

        if (length[i][k]+length[k][j]<length[i][j])
            length[i][j]=length[i][k]+length[k][j];
    } //算法结束

```

33. [题目分析] 该题可用求每对顶点间最短路径的FLOYD算法求解。求出每一顶点（村庄）到其它顶点（村庄）的最短路径。在每个顶点到其它顶点的最短路径中，选出最长的一条。因为有n个顶点，所以有n条，在这n条最长路径中找出最短一条，它的出发点（村庄）就是医院应建立的村庄。

```

void Hospital(AdjMatrix w,int n)
//在以邻接带权矩阵表示的n个村庄中, 求医院建在何处, 使离医院最远的村庄到医院的路径最短。
{for (k=1;k<=n;k++) //求任意两顶点间的最短路径
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (w[i][k]+w[k][j]<w[i][j]) w[i][j]=w[i][k]+w[k][j];
m=MAXINT; //设定m为机器内最大整数。
for (i=1;i<=n;i++) //求最长路径中最短的一条。
    {s=0;
    for (j=1;j<=n;j++) //求从某村庄i (1<=i<=n) 到其它村庄的最长路径。
        if (w[i][j]>s) s=w[i][j];
    if (s<=m) {m=s; k=i;} //在最长路径中, 取最短的一条。m记最长路径, k记出发顶点的下标。
    Printf(“医院应建在%d村庄, 到医院距离为%d\n”, i, m);
    } //for
} //算法结束

```

对以上实例模拟的过程略。在 $A^{(6)}$ 矩阵中（见下图），各行中最大数依次是9, 9, 6, 7, 9, 9。这几个最大数中最小者为6，故医院应建在第三个村庄中，离医院最远的村庄到医院的距离是6。

34. (1) 该有向图的强连通分量有两个：一个是顶点a, 另一个由b, c, e, d四个顶点组成。
 (2) 因篇幅有限从略，请参见本章四？
 (3) 用FLOYD算法求每对顶点间最短距离，其5阶方阵的初态和终态如下：



由于要求各村离医院较近（不是上题中“离医院最远的村庄到医院的路径最短”），因此算法中求出矩阵终态后，将各列值相加，取最小的一个，该列所在村庄即为所求（本题答案是医院建在b村，各村到医院距离和是11），下面是求出各顶点间最短的路径后，求医院所在位置的语句段：

```

min=MAXINT ; //设定机器最大数作村庄间距离之和的初值。
k=1; //k设医院位置。
for (j=1;j<=n;j++)
    {m=0 ;
    for (i=1;i<=n;i++) m=m+w[i][j];
    if (min>m) { min=m ;k=j;} //取顶点间的距离之和的最小值。
    } //for

```

35. [题目分析] 本题应用宽度优先遍历求解。若以v0作生成树的根为第1层，则距顶点v0最短路径长度为K的顶点均在第K+1层。可用队列存放顶点，将遍历访问顶点的操作改为入队操作。队列中设头尾指针f和r，用level表示层数。

```

void bfs_K ( graph g ,int v0 ,K)
//输出无向连通图g（未指定存储结构）中距顶点v0最短路径长度为K的顶点。
{int Q[]; //Q为顶点队列, 容量足够大。
int f=0,r=0,t=0; //f和r分别为队头和队尾指针, t指向当前层最后顶点。
int level=0,flag=0; //层数和访问成功标记。
visited[v0]=1; //设v0为根。
Q[++r]=v0; t=r; level=1; //v0入队。
while (f<r && level<=K+1)
    {v=Q[++f];
    w=GraphFirstAdj(g,v);
    while (w!=0) //w!=0 表示邻接点存在。
        {if (visited[w]==0)
            {Q[++r]=w; visited[w]=1; //邻接点入队列。
            if (level==K+1) { printf("距顶点v0最短路径为k的顶点%d ",w); flag=1;} //if

```

```

    }//if
    w=GraphNextAdj(g , v , w);
    }//while(w!=0)
    if (f==t) {level++; t=r; }//当前层处理完, 修改层数, t指向下一层最后一个顶点
    }//while(f<r && level<=K+1)
    if (flag==0) printf( "图中无距v0顶点最短路径为%d的顶点。\\n",K);
    }//算法结束。

```

[设法讨论]本题亦可采取另一个算法。由于在生成树中结点的层数等于其双亲层次数加1, 故可设顶点和层次数 2 个队列, 其入队和出队操作同步, 其核心语句段如下:

```

QueueInit(Q1) ; QueueInit(Q2); //Q1和Q2是顶点和顶点所在层次数的队列。
visited[v0]=1; //访问数组初始化, 置v0被访问标记。
level=1; flag=0; //是否有层次为K的顶点的标志
QueueIn(Q1,v0); QueueIn(Q2,level); //顶点和层数入队列。
while (!empty(Q1) && level<=K+1)
{v=QueueOut(Q1); level=QueueOut(Q2); //顶点和层数出队。
w=GraphFirstAdj(g,v0);
while (w!=0) //邻接点存在。
{if (visited[w]==0)
if (level==K+1)
{printf("距离顶点v0最短路径长度为K的顶点是%d\\n",w);
visited[w]=1; flag=1; QueueIn(Q1 ,w); QueueIn(Q2,level+1); }//if
w=GraphNextAdj(g , v , w);
} //while(w!=0)
} //while(!empty(Q1) && level<K+1)
if (flag==0) printf( "图中无距v0顶点最短路径为%d的顶点。\\n",K);

```

36. 对于无环连通图, 顶点间均有路径, 树的直径是生成树上距根结点最远的两个叶子间的距离, 利用深度优先遍历可求出图的直径。

```

int dfs(Graph g , vertype parent , vertype child , int len)
//深度优先遍历, 返回从根到结点child所在的子树的叶结点的最大距离。
{current_len=len; maxlen=len;
v=GraphFirstAdj(g , child);
while (v!=0) //邻接点存在。
if (v!=parent)
{len=len+length(g , child , c); dfs(g , child , v , len);
if (len>maxlen) maxlen=len;
v=GraphNextAdj(g , child , v); len=current_len; } //if
len=maxlen;
return(len);
} //结束dfs。
int Find_Diameter (Graph g)
//求无向连通图的直径, 图的顶点信息为图的编号。
{maxlen1=0; maxlen2=0; //存放目前找到的根到叶结点路径的最大值和次大值。
len=0; ///深度优先生成树根到某叶结点间的距离
w=GraphFirstAdj(g,1); //顶点 1 为生成树的根。
while (w!=0) //邻接点存在。
{len=length(g , 1 , w);
if (len>maxlen1) {maxlen2=maxlen1; maxlen1=len;}
else if (len>maxlen2) maxlen2=len;
w=GraphNextAdj(g , 1 , w); //找顶点1的下一邻接点。
} //while
printf( "无向连通图g的最大直径是%d\\n" , maxlen1+maxlen2);
return(maxlen1+maxlen2);
} //结束find_diameter

```

算法主要过程是对图进行深度优先遍历。若以邻接表为存储结构, 则时间复杂度为 $O(n+e)$ 。

37. [题目分析] 利用FLOYD算法求出每对顶点间的最短路径矩阵w, 然后对矩阵w, 求出每列j的最大值, 得到顶点j的偏心度。然后在所有偏心度中, 取最小偏心度的顶点v就是有向图的中心点。

```

void FLOYD_PXD(AdjMatrix g)
//对以带权邻接矩阵表示的有向图g, 求其中心点。
{AdjMatrix w=g ; //将带权邻接矩阵复制到w。
for (k=1;k<=n;k++) //求每对顶点间的最短路径。
for (i=1;i<=n;i++)
for (j=1;j<=n;j++)

```

```

        if ( w[i][j]>w[i][k]+w[k][j]) w[i][j]=w[i][k]+w[k][j];
v=1;    dist=MAXINT;    //中心点初值顶点v为1，偏心率初值为机器最大数。
for (j=1;j<=n;j++)
{
    s=0;
    for (i=1;i<=n;i++) if (w[i][j]>s) s=w[i][j]; //求每列中的最大值为该顶点的偏心率。
    if (s<dist) {dist=s; v=j;} //各偏心率中最小值为中心点。
} //for
printf( "有向图g的中心点是顶点%d, 偏心率%d\n", v, dist);
} //算法结束

```

38. 上面第35题是求无向连通图中距顶点 v_0 的最短路径长度为 k 的所有结点，本题是求有向无环图中距每个顶点最长路径的长度，由于每条弧的长度是1，也需用宽度优先遍历实现，当队列为空前，最后一个结点的层次(减1)可作为从某顶点开始宽度优先遍历的最长路径长度。

```

PROC bfs_Longest ( g: Hnodes)
//求以邻接表表示的有向无环图g的各个顶点的最长路径长度。有关变量已在高层定义。
visited: ARRAY[1..n] OF integer; //访问数组。
Q: ARRAY[1..n] OF integer; //顶点队列。
f, r, t, level: integer; //f, r是队头队尾指针，t记为当前队尾，level层数。
FOR i:=1 TO n DO //循环，求从每个顶点出发的最长路径的长度
[visited[1..n]:=0; f:=0; level:=0; //初始化。
 visited[i]:=1; r:=r+1; Q[r]:=i; t:=r; level:=1; //从顶点i开始宽度优先遍历。
 WHILE (f<r) DO
 [f:=f+1; v:=Q[f]; //队头元素出队。
  p:=g[v].firstarc;
  WHILE (p<>NIL) DO
 [j:=p^adjvex; //邻接点。
  IF visited[j]=0 THEN [visited[j]:=1; r:=r+1; Q[r]:=j;]
  p:=p^nextarc;
 ] //WHILE (p<>NIL)
  IF f=t THEN [level:=level+1; t:=r; ] //t指向下层最后一个顶点。
 ] //WHILE (f<r)
 g[i].mpl:=level-1; //将顶点i的最长路径长度存入mpl域
 ] //FOR
EDNP;

```

39. [题目分析] 按Dijkstra路径增序求出源点和各顶点间的最短路径，上面31题已有解答。本题是求出最小生成树，即以源点为根，其路径权值之和最小的生成树。在确定顶点的最短路径时，总是知道其(弧出自的)前驱(双亲)顶点，我们可用向量 $p[1..n]$ 记录各顶点的双亲信息，源点为根，无双亲，向量中元素值用-1表示。

```

void Shortest_PTree ( AdjMatrix G, vtype v0)
//利用从源点v0到其余各点的最短路径的思想，产生以邻接矩阵表示的图G的最小生成树
{int d[] , s[] ; //d数组存放各顶点最短路径，s数组存放顶点是否找到最短路径。
 int p[] ; //p数组存放顶点在生成树中双亲结点的信息。
 for (i=1;i<=n;i++)
 {d[i]=G[v0][i]; s[i]=0;
  if (d[i]<MAXINT) p[i]=v0; //MAXINT是机器最大数，v0是i的前驱(双亲)。
  else p[i]=-1; } //for //i目前无前驱，p数组各量初始化为-1。
 s[v0]=1; d[v0]=0; p[v0]=-1; //从v0开始，求其最小生成树。
 for (i=1;i<n;i++)
 {mindis=MAXINT;
  for (j=1;j<=n;j++)
  if (s[j]==0 && d[j]<mindis) { u=j; mindis=d[j];}
 s[u]=1; //顶点u已找到最短路径。
  for (j=1;j<=n;j++) //修改j的最短路径及双亲。
  if (s[j]==0 && d[j]>d[u]+G[u][j]) {d[j]=d[u]+G[u][j]; p[j]=u;}
 } //for (i=1;i<=n;i++)
 for (i=1;i<=n;i++) //输出最短路径及其长度，路径是逆序输出。
 if (i!=v0)
 {pre=p[i]; printf( "\n最短路径长度=%d, 路径是: %d, ", d[i], i);
  while (pre!=-1) { printf( ", %d", pre); pre=p[pre]; } //一直回溯到根结点。
 } //if
} //算法结束

```

40. [题目分析] 由关节点定义及特性可知，若生成树的根有多于或等于两棵子树，则根结点是关节点；若生成树某非叶子顶点 v ，其子树中的结点没有指向 v 的祖先的回边，则 v 是关节点。因此，对图 $G=(v, \{Edge\})$ ，将访问函数 $visited[v]$ 定义为深度优先遍历连通图时访问顶点 v 的次序号，并定义一个 $low()$ 函数：

```
low(v)=Min(visited[v],low[w],visited[k])
```

其中 $(v,w) \in \text{Edge}$, $(v,k) \in \text{Edge}$, w 是 v 在深度优先生成树上的孩子结点, k 是 v 在深度优先生成树上的祖先结点。在如上定义下, 若 $\text{low}[w] \geq \text{visited}[v]$, 则 v 是根结点, 因 w 及其子孙无指向 v 的祖先的回边。由此, 一次深度优先遍历连通图, 就可求得所有关节点。

```
int visited[], low[]; //访问函数visited和low函数为全局变量。
int count; //记访问顶点个数。
AdjList g; //图g以邻接表方式存储
void dfs_articul(vertype v0)
//深度优先遍历求关节点。
{count++; visited[v0]=count; //访问顶点顺序号放入visited
min=visited[v0]; //初始化访问初值。
p=g[v0].firstarc; //求顶点v的邻接点。
while (p!=null)
{w=p->adjvex; //顶点v的邻接点。
if (visited[w]==0) //w未曾访问, w是v0的孩子。
{dfs_articul(g, w);
if (low[w]<min) min =low[w];
if (low[w]>=visited[v0]) printf(g[v0].vertex); //v0是关节点。
} //if
else //w已被访问, 是v的祖先。
if (visited[w]<min) min=visited[w];
p=p->next;
} //while
low[v0]=min;
} //结束dfs_articul
void get_articul()
//求以邻接表为存储结构的无向连通图g的关节点。
```

```
{for (vi=1;vi<=n;vi++) visited[vi]=0; //图有n个顶点, 访问数组初始化。
count=1; visited[1]=1; //设邻接表上第一个顶点是生成树的根。
p=g[1].firstarc; v=p->adjvex;
dfs_articul(v);
if (count<n) //生成树的根有两棵以上子树。
{printf(g[1].vertex); //根是关节点
while(p->next!=null)
{p=p->next; v=p->adjvex; if(visited[v]==0) dfs-articul(v);
} //while
} //if } //结束get-articul
```

41. [题目分析] 对有向图进行深度优先遍历可以判定图中是否有回路。若从有向图某个顶点 v 出发遍历, 在 $\text{dfs}(v)$ 结束之前, 出现从顶点 u 到顶点 v 的回边, 图中必存在环。这里设定 visited 访问数组和 finished 数组为全局变量, 若 $\text{finished}[i]=1$, 表示顶点 i 的邻接点已搜索完毕。由于 dfs 产生的是逆拓扑排序, 故设一类型是指向邻接表的边结点的全局指针变量 final , 在 dfs 函数退出时, 把顶点 v 插入到 final 所指的链表中, 链表中的结点就是一个正常的拓扑序列。邻接表的定义与本书相同, 这里只写出拓扑排序算法。

```
int visited[]=0; finished[]=0; flag=1; //flag测试拓扑排序是否成功
ArcNode *final=null; //final是指向顶点链表的指针, 初始化为0
void dfs(AdjList g, vertype v)
//以顶点v开始深度优先遍历有向图g, 顶点信息就是顶点编号。
{ArcNode *t; //指向边结点的临时变量
printf("%d", v); visited[v]=1; p=g[v].firstarc;
while(p!=null)
{j=p->adjvex;
if (visited[j]==1 && finished[j]==0) flag=0 //dfs结束前出现回边
else if(visited[j]==0) {dfs(g, j); finished[j]=1;} //if
p=p->next;
} //while
t=(ArcNode *)malloc(sizeof(ArcNode)); //申请边结点
t->adjvex=v; t->next=final; final=t; //将该点插入链表
} //dfs结束
int dfs-Topsort(Adjlist g)
//对以邻接表为存储结构的有向图进行拓扑排序, 拓扑排序成功返回1, 否则返回0
{i=1;
while (flag && i <=n)
if (visited[i]==0) {dfs(g, i); finished[i]=1; } //if
```

```

return(flag);
} // dfs-Topsort

```

42. [题目分析] 地图涂色问题可以用“四染色”定理。将地图上的国家编号(1到n)，从编号1开始逐一涂色，对每个区域用1色，2色，3色，4色(下称“色数”)依次试探，若当前所取颜色与周围已涂色区域不重色，则将该区域颜色进栈；否则，用下一颜色。若1至4色均与相邻某区域重色，则需退栈回溯，修改栈顶区域的颜色。用邻接矩阵数据结构C[n][n]描述地图上国家间的关系。n个国家用n阶方阵表示，若第i个国家与第j个国家相邻，则 $C_{ij}=1$ ，否则 $C_{ij}=0$ 。用栈s记录染色结果，栈的下标值为区域号，元素值是色数。

```

void MapColor(AdjMatrix C)
//以邻接矩阵C表示的n个国家的地区涂色
{int s[]; //栈的下标是国家编号，内容是色数
s[1]=1; //编号01的国家涂1色
i=2;j=1; //i为国家号，j为涂色号
while (i<=n)
{while (j<=4 && i<=n)
{k=1; //k指已涂色区域号
while (k<i && s[k]*C[i][k]!=j) k++; //判相邻区是否已涂色
if (k<i) j=j+1; //用j+1色继续试探
else {s[i]=j;i++;j=1;} //与相邻区不重色，涂色结果进栈，继续对下一区涂色进行试探
}
} //while (j<=4 && i<=n)
if (j>4) {i--; j=s[i]+1;} //变更栈顶区域的颜色。
} //while } //结束MapColor

```

第10章 排序（参考答案）

一、选择题

1. D	2. D	3. D	4. B	5. B	6. B	7. C, E	8. A	9. C	10. C, D, F		11. 1D, C 11. 2A, D, F			
11. 3B	11. 4(A, C, F) (B, D, E)				12. C, D	13. A	14. B, D	15. D	16. D	17. C	18. A	19. A	20. C	21. C
22. B	23. C	24. C	25. A	26. C	27. D	28. C	29. B	30. C, B	31. D	32. D	33. A	34. D	35. A	
36. A	37. A	38. C	39. B	40. C	41. C	42. B	43. A	44. B	45. A	46. C	47. B, D	48. D	49. D	
50. D	51. C	52. E, G	53. B	54. C	55. C	56. B	57. B	58. A	59. 1C	59. 2A	59. 3D	59. 4B	59. 5G	
60. 1B	60. 2C	60. 3A	61. 1B	61. 2D	61. 3B	61. 4C	61. 5F	62. A	63. A	64. B	65. A	66. A		

部分答案解析如下：

18. 对于后三种排序方法两趟排序后，序列的首部或尾部的两个元素应是有序的两个极值，而给定的序列并不满足。

20. 本题为步长为3的一趟希尔排序。

24. 枢轴是73。

49. 小根堆中，关键字最大的记录只能在叶结点上，故不可能在小于等于 $\lfloor n/2 \rfloor$ 的结点上。

64. 因组与组之间已有序，故将 n/k 个组分别排序即可，基于比较的排序方法每组的时间下界为 $O(k \log_2 k)$ ，全部时间下界为 $O(n \log_2 k)$ 。

二、判断题

1. \checkmark	2. \times	3. \times	4. \times	5. \times	6. \times	7. \times	8. \times	9. \times	10. \times	11. \times	12. \times	13. \times
14. \checkmark	15. \checkmark	16. \times	17. \times	18. \times	19. \times	20. \times	21. \times	22. \times	23. \times	24. \times	25. \checkmark	26. \times
27. \checkmark	28. \times	29. \times	30. \times	31. \checkmark								

部分答案解析如下：

5. 错误。例如冒泡排序是稳定排序，将4, 3, 2, 1按冒泡排序排成升序序列，第一趟变成3, 2, 1, 4，此时3就朝向最终位置的相反方向移动。

12. 错误。堆是 n 个元素的序列，可以看作是完全二叉树，但相对于根并无左小右大的要求，故其既不是二叉排序树，更不会是平衡二叉树。

22. 错误。待排序序列为正序时，简单插入排序比归并排序快。

三、填空题

1. 比较, 移动 2. 生成有序归并段（顺串），归并 3. 希尔排序、简单选择排序、快速排序、堆排序等

4. 冒泡, 快速 5. (1)简单选择排序 (2)直接插入排序（最小的元素在最后时）

6. 免去查找过程中每一步都要检测整个表是否查找完毕，提高了查找效率。 7. $n(n-1)/2$

8. 题中 p 指向无序区第一个记录， q 指向最小值结点，一趟排序结束， p 和 q 所指结点值交换，同时向后移 p 指针。

(1) $!=\text{null}$ (2) $p \rightarrow \text{next}$ (3) $r \neq \text{null}$ (4) $r \rightarrow \text{data} < q \rightarrow \text{data}$ (5) $r \rightarrow \text{next}$ (6) $p \rightarrow \text{next}$

9. 题中为操作方便，先增加头结点（最后删除）， p 指向无序区的前一记录， r 指向最小值结点的前驱，一趟排序结束，无序区第一个记录与 r 所指结点的后继交换指针。

(1) $q \rightarrow \text{link} \neq \text{NULL}$ (2) $r \neq p$ (3) $p \rightarrow \text{link}$ (4) $p \rightarrow \text{link} = s$ (5) $p = p \rightarrow \text{link}$

10. (1) $i < n-i+1$ (2) $j < n-i+1$ (3) $r[j].\text{key} < r[\min].\text{key}$ (4) $\min \neq i$ (5) $\max = i$ (6) $r[\max] \leftrightarrow r[n-i+1]$

11. (1) N (2) 0 (3) $N-1$ (4) 1 (5) $R[P].\text{KEY} < R[I].\text{KEY}$ (6) $R[P].\text{LINK}$ (7) $(N+2)(N-1)/2$

(8) $N-1$ (9) 0 (10) $0(1)$ (每个记录增加一个字段) (11)稳定 (请注意 I 的步长为-1)

12. 3, (10, 7, -9, 0, 47, 23, 1, 8, 98, 36) 13. 快速 14. (4, 1, 3, 2, 6, 5, 7)

15. 最好每次划分能得到两个长度相等的子文件。设文件长度 $n=2^k-1$ ，第一遍划分得到两个长度 $n/2$ 的子文件，第二遍划分得到4个长度 $n/4$ 的子文件，以此类推，总共进行 $k=\log_2(n+1)$ 遍划分，各子文件长度均为1，排序结束。

16. $O(n^2)$ 17. $O(n \log_2 n)$ 18. (1) $2*i$ (2) $r[j].\text{key} > r[j+1].\text{key}$ (3)true (4) $r[j]$ (5) $2*i$

19. (1) $2*i$ (2) $j < r$ (3) $j \leftarrow j+1$ (4) $x.\text{key} > \text{heap}[j].\text{key}$ (5) $i \leftarrow j$ (6) $j \leftarrow 2*i$ (7) x

20. (1) $j:=2*i$ (2)finished:=false (3) $(r[j].\text{key} > r[j+1].\text{key})$ (4) $r[i]:=r[j]$ (5) $i:=j$

(6) $j:=2*i$ (7) $r[i]:=t$; (8)sift(r, i, n) (9) $r[1]:=r[i]$ (10)sift($r, 1, i-1$)

21. ④是堆 (1)选择 (2)筛选法 (3) $O(n \log_2 n)$ (4) $O(1)$

22. (1)选择 (2)完全二叉树 (3) $O(N \log_2 N)$ (4) $O(1)$ (5)满足堆的性质

23. (1)finish:=false (2) $h[i]:=h[j]; i:=j; j:=2*j$; (3) $h[i]:=x$ (4) h, k, n (5)sift($h, 1, r-1$)

24. {D, Q, F, X, A, P, B, N, M, Y, C, W}

25. (1) $p[k]:=j$ (2) $i:=i+1$ (3) $k=0$ (4) $m:=n$ (5) $m < n$ (6) $a[i]:=a[m]$ (7) $a[m]:=t$

26. 程序(a) (1)true (2) $a[i]:=t$ (3)2 TO n step 2 (4)true (5)NOT flag

程序(b) (1)1 (2) $a[i]=t$ (3) $(i=2; i \leq n; i+=2)$ (4)1 (5)flag

27. (Q, A, C, S, Q, D, F, X, R, H, M, Y), (F, H, C, D, Q, A, M, Q, R, S, Y, X) 28. 初始归并段(顺串)

29. 初始归并段, 初始归并段, 减少外存信息读写次数（即减少归并趟数），增加归并路数和减少初始归并段个数。 30. $\epsilon n / m$

31. (1) $m, j-1$ (2) $m:=j+1$ (3) $j+1, n$ (4) $n:=j-1$ 最大栈空间用量为 $O(\log n)$ 。

四、应用题

1. 假设含 n 个记录的序列为 $\{ R_1, R_2, \dots, R_n \}$ ，其相应的关键字序列为 $\{ K_1, K_2, \dots, K_n \}$ ，这些关键字相互

之间可以进行比较，即在它们之间存在着这样一个关系 $Ks_1 \leq Ks_2 \leq \dots \leq Ks_n$ ，按此固有关系将 n 个记录序列重新排列为 $\{Rs_1, Rs_2, \dots, Rs_n\}$ 。若整个排序过程都在内存中完成，则称此类排序问题为内部排序。

2.

排序方法	平均时间	最坏情况	辅助空间	稳定性	不稳定排序举例
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定	
折半插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定	
二路插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	稳定	
表插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定	
起泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定	
直接选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	2, 2', 1
希尔排序	$O(n^{1.3})$	$O(n^{1.3})$	$O(1)$	不稳定	3, 2, 2', 1 ($d=2, d=1$)
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定	2, 2', 1
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定	2, 1, 1' (极大堆)
2-路归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定	
基数排序	$O(d \cdot (rd+n))$	$O(d \cdot (rd+n))$	$O(rd)$	稳定	

3. 这种说法不对。因为排序的不稳定性是指两个关键字值相同的元素的相对次序在排序前、后发生了变化，而题中叙述和排序中稳定性的定义无关，所以此说法不对。对4, 3, 2, 1起泡排序就可否定本题结论。

4. 可以做到。取 a 与 b 进行比较， c 与 d 进行比较。设 $a > b, c > d$ ($a < b$ 和 $c < d$ 情况类似)，此时需2次比较，取 b 和 d 比较，若 $b > d$ ，则有序 $a > b > d$ ；若 $b < d$ 时则有序 $c > d > b$ ，此时已进行了3次比较。再把另外两个元素按折半插入排序方法，插入到上述某个序列中共需4次比较，从而共需7次比较。

5. 本题答案之一请参见第9章的“四、应用题”第70题，这里用分治法求解再给出另一参考答案。

对于两个数 x 和 y ，经一次比较可得到最大值和最小值；对于三个数 x, y, z ，最多经3次比较可得最大值和最小值；对于 n ($n > 3$)个数，将分成长度为 $n-2$ 和2的前后两部分 A 和 B ，分别找出最大者和最小者： $Max_A, Min_A, Max_B, Min_B$ ，最后 $Max = \{Max_A, Max_B\}$ 和 $Min = \{Min_A, Min_B\}$ 。对 A 使用同样的方法求出最大值和最小值，直到元素个数不超过3。设 $C(n)$ 是所需的最多比较次数，根据上述原则，当 $n > 3$ 时有如下关系式：

$$C(n) = \begin{cases} 1 & n=2 \\ 3 & n=3 \\ C(n-2)+3 & n>3 \end{cases}$$

通过逐步递推，可以得到： $C(n) = \epsilon 3n/2 - 2$ 。显然，当 $n \geq 3$ 时， $2n-3 > 3n/2-2$ 。事实上， $\epsilon 3n/2 - 2$ 是解决这一问题的比较次数的下限。

6. 假定待排序的记录有 n 个。由于含 n 个记录的序列可能出现的状态有 $n!$ 个，则描述 n 个记录排序过程的判定树必须有 $n!$ 个叶子结点。因为若少一个叶子，则说明尚有两种状态没有分辨出来。我们知道，若二叉树高度是 h ，则叶子结点个数最多为 2^{h-1} ；反之，若有 u 个叶子结点，则二叉树的高度至少为 $\epsilon \log_2 u + 1$ 。这就是说，描述 n 个记录排序的判定树必定存在一条长度为 $\epsilon \log_2 (n!) - 1$ 的路径。即任何一个借助“比较”进行排序的算法，在最坏情况下所需进行的比较次数至少是 $\epsilon \log_2 (n!) - 1$ 。根据斯特林公式，有 $\epsilon \log_2 (n!) - 1 = O(n \log_2 n)$ 。即借助于“比较”进行排序的算法在最坏情况下能达到的最好时间复杂度为 $O(n \log_2 n)$ 。证毕。

7. 答：拓扑排序，是有向图的顶点依照弧的走向，找出一个全序集的过程，主要是根据与顶点连接的弧来确定顶点序列；冒泡排序是借助交换思想通过比较相邻结点关键字大小进行排序的算法。

8. 直接插入排序的基本思想是基于插入，开始假定第一个记录有序，然后从第二个记录开始，依次插入到前面有序的子文件中。即将记录 $R[i]$ ($2 \leq i \leq n$)插入到有序子序列 $R[1..i-1]$ 中，使记录的有序序列从 $R[1..i-1]$ 变为 $R[1..i]$ ，最终使整个文件有序。共进行 $n-1$ 趟插入。最坏时间复杂度是 $O(n^2)$ ，平均时间复杂度是 $O(n^2)$ ，空间复杂度是 $O(1)$ ，是稳定排序。

简单选择排序的基本思想是基于选择，开始有序序列长度为零，第 i ($1 \leq i \leq n$)趟简单选择排序是，从无序序列 $R[i..n]$ 的 $n-i+1$ 记录中选出关键字最小的记录，和第 i 个记录交换，使有序序列逐步扩大，最后整个文件有序。共进行 $n-1$ 趟选择。最坏时间复杂度是 $O(n^2)$ ，平均时间复杂度是 $O(n^2)$ ，空间复杂度是 $O(1)$ ，是不稳定排序。

二路并归排序的基本思想是基于归并，开始将具有 n 个待排序记录的序列看成是 n 个长度为1的有序序列，然后进行两两归并，得到 $\epsilon n/2$ 个长度为2的有序序列，再进行两两归并，得到 $\epsilon n/4$ 个长度为4的有序序列。如此重复，经过 $\epsilon \log_2 n$ 趟归并，最终得到一个长度为 n 的有序序列。最坏时间复杂度和平均时间复杂度都是 $O(n \log_2 n)$ ，空间复杂度是 $O(n)$ ，是稳定排序。

9. 错误。快速排序，堆排序和希尔排序是时间性能较好的排序方法，但都是不稳定的排序方法。

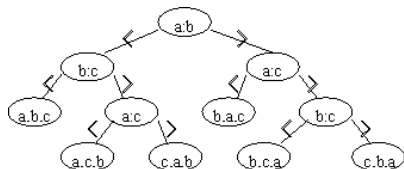
10. 等概率（后插），插入位置 $0..n$ ，则平均移动个数为 $n/2$ 。

若不等概率, 则平均移动个数为 $\sum_{i=0}^{n-1} (n-i)/(n*(n+1)/2)*(n-i) = \frac{2n+1}{3}$

11. 从节省存储空间考虑: 先选堆排序, 再选快速排序, 最后选择归并排序;
从排序结果的稳定性考虑: 选择归并排序。堆排序和快速排序都是不稳定排序;
从平均情况下排序最快考虑: 先选择快速排序。
12. (1) 堆排序, 快速排序, 归并排序 (2) 归并排序 (3) 快速排序 (4) 堆排序
13. 平均比较次数最少: 快速排序; 占用空间最多: 归并排序; 不稳定排序算法: 快速排序、堆排序、希尔排序。
14. 求前k个最大元素选堆排序较好。因为在建含n个元素的堆时, 总共进行的关键字的比较次数不超过4n, 调整建新堆时的比较次数不超过2log₂n次。在n个元素中求前k个最大元素, 在堆排序情况下比较次数最多不超过4n+2klog₂n。

稳定分类是指, 若排序序列中存在两个关键字值相同的记录R_i与R_j (K_i=K_j, i≠j) 且R_i领先于R_j, 若排序后R_i与R_j的相对次序保持不变, 则称这类分类是稳定分类(排序), 否则为不稳定分类。

A, C和E是稳定分类(排序), B和D是不稳定分类(排序)。



- 15.
16. (1) 此为直接插入排序算法, 该算法稳定。
(2) r[0]的作用是监视哨, 免去每次检测文件是否到尾, 提高了排序效率。
采用x.key<=r[j].key描述算法后, 算法变为不稳定排序, 但能正常工作。
17. (1) 横线内容: ①m ②1 ③0 ④1
(2) flag起标志作用。若未发生交换, 表明待排序列已有序, 无需进行下趟排序。
(3) 最大比较次数n(n-1)/2, 最大移动次数3n(n-1)/2 (4) 稳定
18. (1) ①499 ②A[j]>A[j+1] ③b:=true (2) 冒泡排序 (3) 499次比较, 0次交换
(4) n(n-1)/2次比较, n(n-1)/2次交换 (相当3n(n-1)/2次移动), 本题中n=500, 故有124750次比较和交换 (相当373250次移动)。
19. 答: 此排序为双向起泡排序: 从前向后一趟排序下来得到一个最大值, 若其中发生交换, 则再从后向前一趟排序, 得到一个最小值。
一趟: 12, 23, 35, 16, 25, 36, 19, 21, 16, 47
二趟: 12, 16, 23, 35, 16, 25, 36, 19, 21, 47 三趟: 12, 16, 23, 16, 25, 35, 19, 21, 36, 47
四趟: 12, 16, 16, 23, 19, 25, 35, 21, 36, 47 五趟: 12, 16, 16, 19, 23, 25, 21, 35, 36, 47
六趟: 12, 16, 16, 19, 21, 23, 25, 35, 36, 47 七趟: 12, 16, 16, 19, 21, 23, 25, 35, 36, 47
20. 对冒泡算法而言, 初始序列为反序时交换次数最多。若要求从大到小排序, 则表现为初始是上升序。
21. 证明: 起泡排序思想是相邻两个记录的关键字比较, 若反序则交换, 一趟排序完成得到一个极值。由题假设知R_j在R_i之前且K_j>R_i, 即说明R_j和R_i是反序; 设对于R_i之前全部记录1—R_{i-1} (其中包括K_j) 中关键字最大为K_{max}, 则K_{max}≥K_j, 故经过起泡排序前i-2次后, R_{i-1}的关键字一定为K_{max}, 又因K_{max}≥K_j>R_i, 故R_{i-1}和R_i为反序, 由此可知R_{i-1}和R_i必定交换, 证毕。
22. 采用直接插入排序算法, 因为记录序列已基本有序, 直接插入排序比较次数少, 且由于少量次序不对的记录与正确位置不远, 使直接插入排序记录移动次数也相对较少, 故选直接插入排序算法。
23. 各带标号语句的频度: (1)n (2)n-1 (3) (n+2) (n-1)/2 (4) n(n-1)/2 (5) n-1
时间复杂度O(n²), 属于直接选择排序。
24. 6, 13, 28, 39, 41, 72, 85, 20
i=1 ↑ m=4 ↑ r=7 ↑
20<39 i=1 ↑ m=2 ↑ r=3 ↑
20>13 i=3 r=3 m=3
20<28 r=2 i=3
将r+1 (即第3个) 后的元素向后移动, 并将20放入r+1处, 结果为6, 13, 20, 28, 39, 41, 72, 85。
(1) 使用二分法插入排序所要进行的比较次数, 与待排序的记录初始状态无关。不论待排序序列是否有顺序, 已形成的部分子序列是有序的。二分法插入首先查找插入位置, 插入位置是判定树查找失败的位置。失败位置只能在判定树的最下两层上。
(2) 一些特殊情况下, 二分法插入排序要比直接插入排序要执行更多的比较。例如, 在待排序序列已有序的情况下就是如此。
25. (1) 直接插入排序
第一趟 (3) [8, 3], 2, 5, 9, 1, 6 第二趟 (2) [8, 3, 2], 5, 9, 1, 6 第三趟 (5) [8, 5, 3, 2], 9, 1, 6
第四趟 (9) [9, 8, 5, 3, 2], 1, 6 第五趟 (1) [9, 8, 5, 3, 2, 1], 6 第六趟 (6) [9, 8, 6, 5, 3, 2, 1]

(2) 直接选择排序 (第六趟后仅剩一个元素, 是最小的, 直接选择排序结束)

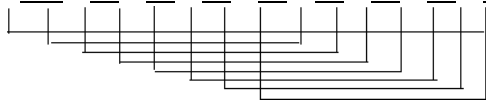
第一趟 (9) [9], 3, 2, 5, 8, 1, 6 第二趟 (8) [9, 8], 2, 5, 3, 1, 6 第三趟 (6) [9, 8, 6], 5, 3, 1, 2

第四趟 (5) [9, 8, 6, 5], 3, 1, 2 第五趟 (3) [9, 8, 6, 5, 3], 1, 2 第六趟 (2) [9, 8, 6, 5, 3, 2], 1

(3) 直接插入排序是稳定排序, 直接选择排序是不稳定排序。

26. 这种看法不对。本题的叙述与稳定性的定义无关, 不能据此来判断排序中的稳定性。例如 5, 4, 1, 2, 3 在第一趟冒泡排序后得到 4, 1, 2, 3, 5。其中 4 向前移动, 与其最终位置相反。但冒泡排序是稳定排序。快速排序中无这种现象。

27. 设 $D=7$ 125, 11, 22, 34, 15, 44, 76, 66, 100, 8, 14, 20, 2, 5, 1



$D=3$ 1, 11, 8, 14, 15, 2, 5, 66, 100, 22, 34, 20, 44, 76, 125

$D=1$ 1, 11, 2, 5, 15, 8, 14, 34, 20, 22, 66, 100, 44, 76, 125

$D=1$ 1, 2, 5, 8, 11, 14, 15, 20, 22, 34, 44, 66, 76, 100, 125

28. 设待排序记录的个数为 n , 则快速排序的最小递归深度为 $\lceil \log_2 n \rceil + 1$, 最大递归深度 n 。

29. 平均性能最佳的排序方法是快速排序, 该排序方法不稳定。

初始序列: 50, 10, 50, 40, 45, 85, 80

一趟排序: [45, 10, 50, 40] 50 [85, 80] 二趟排序: [40, 10] 45 [50] 50 [80] 85

三趟排序: 10, 40, 45, 50, 50, 80, 85

30. 快速排序。

31. (1) 在最好情况下, 假设每次划分能得到两个长度相等的子文件, 文件的长度 $n=2^k-1$, 那么第一遍划分得到两个长度均为 $n/2$ 的子文件, 第二遍划分得到 4 个长度均为 $n/4$ 的子文件, 以此类推, 总共进行 $k=\log_2(n+1)$ 遍划分, 各子文件的长度均为 1, 排序完毕。当 $n=7$ 时, $k=3$, 在最好情况下, 第一遍需比较 6 次, 第二遍分别对两个子文件 (长度均为 3, $k=2$) 进行排序, 各需 2 次, 共 10 次即可。

(2) 在最好情况下快速排序的原始序列实例: 4, 1, 3, 2, 6, 5, 7。

(3) 在最坏情况下, 若每次用来划分的记录的关键字具有最大值 (或最小值), 那么只能得到左 (或右) 子文件, 其长度比原长度少 1。因此, 若原文件中的记录按关键字递减次序排列, 而要求排序后按递增次序排列时, 快速排序的效率与冒泡排序相同, 其时间复杂度为 $O(n^2)$ 。所以当 $n=7$ 时, 最坏情况下的比较次数为 21 次。

(4) 在最坏情况下快速排序的初始序列实例: 7, 6, 5, 4, 3, 2, 1, 要求按递增排序。

32. 该排序方法为快速排序。

33. 不是。因为当序列已有序时, 快速排序将退化成冒泡排序, 时间复杂度为 $O(n^2)$ 。当待排序序列无序, 使每次划分完成后, 枢轴两侧子文件长度相当, 此时快速排序性能最好。

34. 初始序列: [28], 07, 39, 10, 65, 14, 61, 17, 50, 21 21 移动: 21, 07, 39, 10, 65, 14, 61, 17, 50, []

39 移动: 21, 07, [], 10, 65, 14, 61, 17, 50, 39 17 移动: 21, 07, 17, 10, 65, 14, 61, [], 50, 39

65 移动: 21, 07, 17, 10, [], 14, 61, 65, 50, 39 14 移动: 21, 07, 17, 10, 14, [28], 61, 65, 50, 39

类似本题的另外叙述题的解答:

(1): 快速排序思想: 首先将待排序记录序列中的所有记录作为当前待排序区域, 以第一个记录的关键字作为枢轴 (或支点) (pivot), 凡其关键字不大于枢轴的记录均移动至该记录之前, 凡关键字不小于枢轴的记录均移动至该记录之后。致使一趟排序之后, 记录的无序序列 $R[s..t]$ 将分割成两部分: $R[s..i-1]$ 和 $R[i+1..t]$, 且 $R[j].key \leq R[i].key \leq R[k].key (s \leq j < i, i < k \leq t)$, 然后再递归地将 $R[s..i-1]$ 和 $R[i+1..t]$ 进行快速排序。快速排序在记录有序时蜕变为冒泡排序, 可用 “三者取中” 法改善其性能, 避免最坏情况的出现。具体排序实例不再解答。

35. (1) 不可以, 若 $m+1$ 到 n 之间关键字都大于 m 的关键字时, $<=k$ 可将 j 定位到 m 上, 若为 $<k$ 则 j 将定位到 $m-1$ 上, 将出界线, 会造成错误。

(2) 不稳定, 例如 2, 1, 2', ($m=1, n=3$) 对 2, 1, 2' 排序, 完成会变成 1, 2', 2。

(3) 各次调用 qsort1 的结果如下:

一次调用 $m=1, n=10$ 11, 3, 16, 4, 18, 22, 58, 60, 40, 30 $j=6$

二次调用 $m=7, n=10$ 11, 3, 16, 4, 18, 22, 40, 30, 58, 60 $j=9$ (右部)

三次调用 $m=10, n=10$ 不变, 返回 $m=7, n=8$ 11, 3, 16, 4, 18, 22, 30, 40, 58, 60 $j=8$

四次调用 $m=9, n=8$ 不变, 返回 $m=7, n=7$ 返回 $m=1, n=5$ 4, 3, 11, 16, 18, 22, 30, 40, 58, 60 $j=3$ (左部)

五次调用 $m=1, n=2$ 3, 4, 11, 16, 18, 22, 30, 40, 58, 60 $j=2$

六次调用 $m=1, n=1$ 不变, 返回 $m=3, n=2$ 返回 $m=4, n=5$ 3, 4, 11, 16, 18, 22, 30, 40, 58, 60 $j=4$

七次调用 $m=4, n=3$ 不变, 返回 (注: 一次划分后, 左、右两部分调用算两次)

(4) 最大栈空间用量为 $O(\log n)$ 。

36. 在具有 n 个元素的集合中找第 k ($1 \leq k \leq n$) 个最小元素, 应使用快速排序方法。其基本思想如下: 设 n 个元素

的集合用一维数组表示，其第一个元素的下标为1，最后一个元素下标为n。以第一个元素为“枢轴”，经过快速排序的一次划分，找到“枢轴”的位置i，若 $i=k$ ，则该位置的元素即为所求；若 $i>k$ ，则在1至i-1间继续进行快速排序的划分；若 $i<k$ ，则在i+1至n间继续进行快速排序的划分。这种划分一直进行到 $i=k$ 为止，第i位置上的元素就是第k ($1 \leq k \leq n$) 个最小元素。

37. 快速排序各次调用结果：

一次调用：18, 36, 77, 42, 23, 65, 84, 10, 59, 37, 61, 98

二次调用：10, 18, 77, 42, 23, 65, 84, 36, 59, 37, 61, 98

三次调用：10, 18, 61, 42, 23, 65, 37, 36, 59, 77, 84, 98

归并排序各次调用结果：

一次调用36, 98, 42, 77, 23, 65, 10, 84, 37, 59, 18, 61, (子文件长度为1, 合并后子文件长度为2)

二次调用36, 42, 77, 98, 10, 23, 65, 84, 18, 37, 59, 61 (子文件长度为2, 合并后子文件长度为4)

三次调用10, 23, 36, 42, 65, 77, 84, 98, 18, 37, 59, 61 (第一子文件长度8, 第二子文件长度为4)

38. 建立堆结构: 97, 87, 26, 61, 70, 12, 3, 45 (2) 70, 61, 26, 3, 45, 12, 87, 97

(4) 45, 12, 26, 3, 61, 70, 87, 97 (6) 12, 3, 26, 45, 61, 70, 87, 97

39. (1) 是大堆; (2) 是大堆; (4) 是小堆;

(3) 不是堆, 调成大堆 100, 98, 66, 85, 80, 60, 40, 77, 82, 10, 20

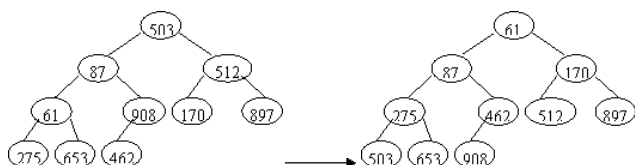
类似叙述(1): 不是堆 调成大顶堆 92, 86, 56, 70, 33, 33, 48, 65, 12

(2) ①是堆 ②不是堆 调成堆 100, 90, 80, 25, 85, 75, 60, 20, 10, 70, 65, 50

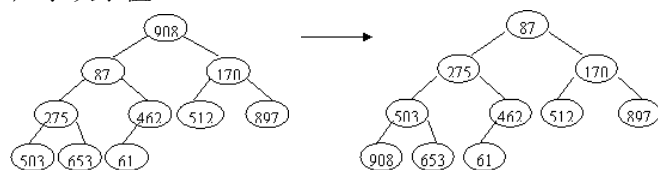
(3) ①是堆 ②不是堆 调成大堆 21, 9, 18, 8, 4, 17, 5, 6 (图略) (4): 略

40. 在内部排序方法中，一趟排序后只有简单选择排序和冒泡排序可以选出一个最大（或最小）元素，并加入到已有的有序子序列中，但要比较 $n-1$ 次。选次大元素要再比较 $n-2$ 次，其时间复杂度是 $O(n^2)$ 。从10000个元素中选10个元素不能使用这种方法。而快速排序、插入排序、归并排序、基数排序等时间性能好的排序，都要等到最后才能确定各元素位置。只有堆排序，在未结束全部排序前，可以有部分排序结果。建立堆后，堆顶元素就是最大（或最小，视大堆或小堆而定）元素，然后，调堆又选出次大（小）元素。凡要求在n个元素中选出k ($k \ll n, k > 2$) 个最大（或最小）元素，一般均使用堆排序。因为堆排序建堆比较次数至多不超过 $4n$ ，对深度为k的堆，在调堆算法中进行的关键字的比较次数至多为 $2(k-1)$ 次，且辅助空间为 $O(1)$ 。

41. (1) 建小堆



(2) 求次小值



42. 用堆排序方法，详见第40题的分析。从序列{59, 11, 26, 34, 14, 91, 25}得到{11, 17, 25}共用14次比较。其中建堆输出11比较8次，调堆输出17和25各需要比较4次和2次。

类似本题的另外叙述题的解答：

(1) 堆排序，分析同前，共 $20+5+4+5=34$ 次比较。

43. 对具体例子的手工堆排序略。

堆与败者树的区别：堆是n个元素的序列，在向量中存储，具有如下性质：

$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad (i=1, 2, \dots, \lfloor n/2 \rfloor)$$

由于堆的这个性质中下标i和2i、2i+1的关系，恰好和完全二叉树中第i个结点和其子树结点的序号间的关系一致，所以堆可以看作是n个结点的完全二叉树。而败者树是由参加比赛的n个元素作叶子结点而得到的完全二叉树。每个非叶（双亲）结点中存放的是两个子结点中的败者数据，而让胜者去参加更高一级的比赛。另外，还需增加一个结点，即结点0，存放比赛的全局获胜者。

44. (1) 堆的存储是顺序的

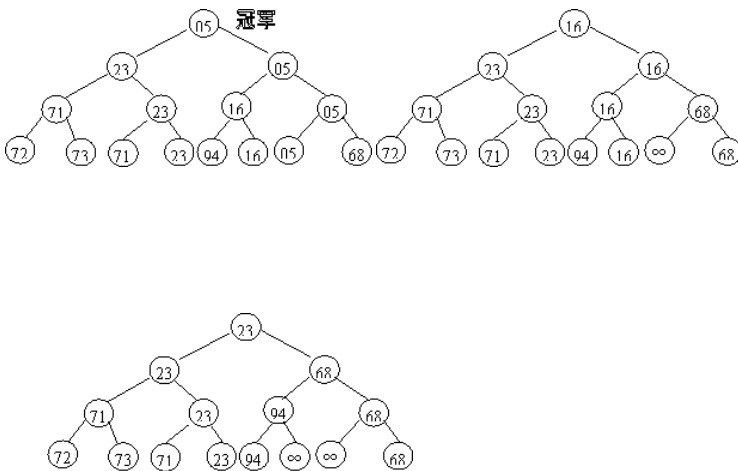
(2) 最大值元素一定是叶子结点，在最下两层上。

(3) 在建含有 n 个元素、深度为 h 的堆时，其比较次数不超过 $4n$ ，推导如下：

由于第 i 层上的结点数至多是 2^{i-1} ，以它为根的二叉树的深度为 $h-i+1$ ，则调用 $\lceil n/2 \rceil$ 次筛选算法时总共进行的关键字比较次数不超过下式之值：

$$\sum_{i=h-1}^1 2^{i-1} \cdot 2(h-i) = \sum_{i=h-1}^1 2^i (h-i) = \sum_{j=1}^{h-1} 2^{h-j} \cdot j \leq (2n) \sum_{j=1}^{h-1} j / 2^j \leq 4n$$

45. (1)



(2) 树形选择排序基本思想：首先对 n 个待排序记录的关键字进行两两比较，从中选出 $\lceil n/2 \rceil$ 个较小者再两两比较，直到选出关键字最小的记录为止，此为一趟排序。我们将一趟选出的关键字最小的记录称为“冠军”，而“亚军”是从与“冠军”比较失败的记录中找出，具体做法为：输出“冠军”后，将其叶子结点关键字改为“最大值”，然后从该叶子结点开始，和其左（或右）兄弟比较，修改从叶子结点到根结点路径上各结点的关键字，则根结点的关键字即为次小关键字。如此下去，可依次选出从小到大的全部关键字。

(3) 树形选择排序与直接选择排序相比较，其优点是从求第 2 个元素开始，从 $n-i+1$ 个元素中不必进行 $n-i$ 次比较，只比较 $\lceil \log_2 n \rceil$ 次，比较次数远小于直接选择排序；其缺点是辅助储存空间大。

(4) 堆排序基本思想是：堆是 n 个元素的序列，先建堆，即先选得一个关键字最大或最小的记录，然后与序列中最后一个记录交换，之后将序列中前 $n-1$ 记录重新调整为堆（调堆的过程称为“筛选”），再将堆顶记录和当前堆序列的最后一个记录交换，如此反复直至排序结束。优点是在时间性能与树形选择排序属同一量级的同时，堆排序只需要一个记录大小供交换用的辅助空间，调堆时子女只和双亲比较。避免了过多的辅助存储空间及和最大值的比较。

46. K_1 到 K_n 是堆，在 K_{n+1} 加入后，将 $K_1 \dots K_{n+1}$ 调成堆。设 $c=n+1$, $f=\lceil c/2 \rceil$, 若 $K_f \leq K_c$, 则调整完成。否则， K_f 与 K_c 交换；之后， $c=f$, $f=\lceil c/2 \rceil$, 继续比较，直到 $K_f \leq K_c$, 或 $f=0$, 即为根结点，调整结束。

47. (1) ① $\text{child}=\text{child}+1$; ② $\text{child}/2$ (2) 不能，调为大堆：92, 86, 56, 70, 33, 33, 48, 65, 12, 24

48. (1) 不需要。因为建堆后 $R[1]$ 到 $R[n]$ 是堆，将 $R[1]$ 与 $R[n]$ 交换后， $R[2]$ 到 $R[n-1]$ 仍是堆，故对 $R[1]$ 到 $R[n-1]$ 只需从 $R[1]$ 往下筛选即可。

(2) 堆是 n 个元素的序列，堆可以看作是 n 个结点的完全二叉树。而树型排序是 n 个元素作叶子结点的完全二叉树。因此堆占用的空间小。调堆时，利用堆本身就可以存放输出的有序数据，只需要一个记录大小供交换用的辅助空间。排序后，heap 数组中的关键字序列与堆是大堆还是小堆有关，若利用大堆，则为升序；若利用小堆则为降序。

49. 最高位优先 (MSD) 法：先对最高位关键字 K^0 进行排序，将序列分成若干子序列，每个子序列中的记录都具有相同的 K^0 值，然后，分别就每个子序列对关键字 K^1 进行排序，按 K^1 值不同再分成若干更小的子序列，……，依次重复，直至最后对最低位关键字排序完成，将所有子序列依次连接在一起，成为一个有序子序列。

最低位优先 (LSD) 法：先对最低位关键字 K^{d-1} 进行排序，然后对高级关键字 K^{d-2} 进行排序，依次重复，直至对最高位关键字 K^0 排序后便成为一个有序序列。进行排序时，不必分成子序列，对每个关键字都是整个序列参加排序，但对 K^i ($0 \leq i < d-1$) 排序时，只能用稳定的排序方法。另一方面，按 LSD 进行排序时，可以不通过关键字比较实现排序，而是通过若干次“分配”和“收集”来实现排序。

50. (1) 冒泡排序 (H, C, Q, P, A, M, S, R, D, F, X, Y)

(2) 初始步长为 4 的希尔排序 (P, A, C, S, Q, D, F, X, R, H, M, Y)

(3) 二路归并排序 (H, Q, C, Y, A, P, M, S, D, R, F, X) (4) 快速排序 (F, H, C, D, P, A, M, Q, R, S, Y, X)

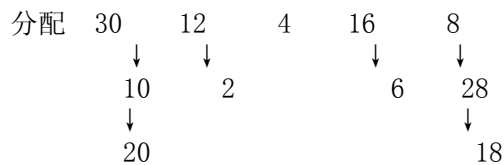
初始建堆: (A, D, C, R, F, Q, M, S, Y, P, H, X)

51. (1) 一趟希尔排序: 12, 2, 10, 20, 6, 18, 4, 16, 30, 8, 28 (D=5)

(2) 一趟快速排序: 6, 2, 10, 4, 8, 12, 28, 30, 20, 16, 18

(3) 链式基数排序 LSD [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

↓ ↓ ↓ ↓ ↓



收集: $\rightarrow 30 \rightarrow 10 \rightarrow 20 \rightarrow 12 \rightarrow 2 \rightarrow 4 \rightarrow 16 \rightarrow 6 \rightarrow 8 \rightarrow 28 \rightarrow 18$

52. (1) 2路归并 第一趟: 18, 29, 25, 47, 12, 58, 10, 51; 第二趟: 18, 25, 29, 47, 10, 12, 51, 58;

第三趟: 10, 12, 18, 25, 29, 47, 51, 58

(2) 快速排序 第一趟: 10, 18, 25, 12, 29, 58, 51, 47;

第二趟: 10, 18, 25, 12, 29, 47, 51, 88; 第三趟: 10, 12, 18, 25, 29, 47, 51, 88

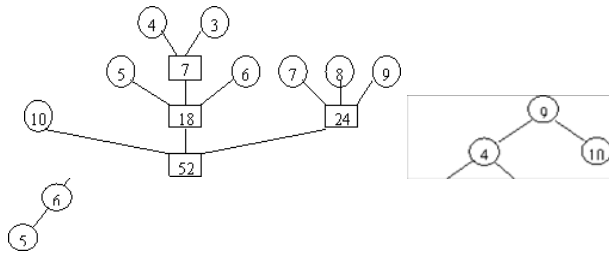
(3) 堆排序 建大堆: 58, 47, 51, 29, 18, 12, 25, 10;

① 51, 47, 25, 29, 18, 12, 10, 58; ② 47, 29, 25, 10, 18, 12, 51, 58;

③ 29, 18, 25, 10, 12, 47, 51, 58; ④ 25, 18, 12, 10, 29, 47, 51, 58;

⑤ 18, 10, 12, 25, 29, 47, 51, 58; ⑥ 12, 10, 18, 25, 29, 47, 51, 58; ⑦ 10, 12, 18, 25, 29, 47, 51, 58

类似叙述: (1) ① 设按3路归并 I/O次数=2*wp1=202次 ②



③④⑤ 略。

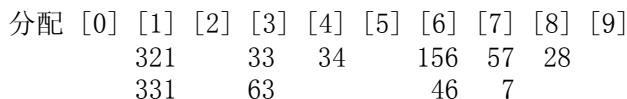
53. (1) 当至多进行n-1趟起泡排序, 或一趟起泡排序中未发生交换 (即已有序) 时, 结束排序。

(2) 希尔排序是对直接插入排序算法的改进, 它从“记录个数少”和“基本有序”出发, 将待排序的记录划分成几组 (缩小增量分组), 从而减少参与直接插入排序的数据量, 当经过几次分组排序后, 记录的排列已经基本有序, 这个时候再对所有的记录实施直接插入排序。

(3) 13, 24, 33, 65, 70, 56, 48, 92, 80, 112

(4) 采用树型锦标赛排序选出最小关键字至少要15次比较。再选出次小关键字比较4次。(两两比较8次选出8个优胜, 再两两比较4次选出4个优胜, 半决赛两场, 决赛一场, 共比较了15次。将冠军的叶子结点改为最大值, 均与兄弟比较, 共4次选出亚军。)

54. (1) 按LSD法 $\rightarrow 321 \rightarrow 156 \rightarrow 57 \rightarrow 46 \rightarrow 28 \rightarrow 7 \rightarrow 331 \rightarrow 33 \rightarrow 34 \rightarrow 63$



收集 $\rightarrow 321 \rightarrow 331 \rightarrow 33 \rightarrow 63 \rightarrow 34 \rightarrow 156 \rightarrow 46 \rightarrow 57 \rightarrow 7 \rightarrow 28$

类似叙述(1) 略。

55. ①快速排序 ②冒泡排序 ③直接插入排序 ④堆排序

56. A. $p[k] \leftarrow j$ B. $i \leftarrow i+1$ C. $k=0$ D. $m \leftarrow n$ E. $m < n$ F. $a[i] \leftarrow a[m]$ G. $a[m] \leftarrow t$

57. 一趟快速排序: 22, 19, 13, 6, 24, 38, 43, 32

初始大堆: 43, 38, 32, 22, 24, 6, 13, 19

二路并归: 第一趟: 19, 24, 32, 43, 6, 38, 13, 22

第二趟: 19, 24, 32, 43, 6, 13, 22, 38

第三趟: 6, 13, 19, 22, 24, 32, 38, 43

堆排序辅助空间最少, 最坏情况下快速排序时间复杂度最差。

58. (1) 排序结束条件为没有交换为止

第一趟奇数: 35, 70, 33, 65, 21, 24, 33

第二趟偶数: 35, 33, 70, 21, 65, 24, 33

第三趟奇数: 33, 35, 21, 70, 24, 65, 33

第四趟偶数: 33, 21, 35, 24, 70, 33, 65

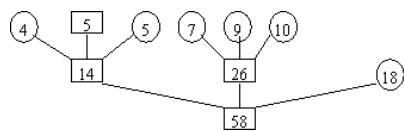
第五趟奇数: 21, 33, 24, 35, 33, 70, 65

第六趟偶数: 21, 24, 33, 33, 35, 65, 70

第七趟奇数: 21, 24, 33, 33, 35, 65, 70 (无交换) 第八趟偶数: 21, 24, 33, 33, 35, 65, 70 (无交换) 结束

59. 设归并路数为k, 归并趟数为s, 则 $s = \lceil \log_k 100 \rceil$, 因 $\lceil \log_k 100 \rceil = 3$, 所以k=5, 即最少5路归并。

60. 证明: 由置换选择排序思想, 第一个归并段中第一个元素是缓冲区中最小的元素, 以后每选一个元素都不应小于前一个选出的元素, 故当产生第一个归并段时 (即初始归并段), 缓冲区中m个元素中除最小元素之外, 其他m-1个元素均大于第一个选出的元素, 即当以后读入元素均小于输出元素时, 初始归并段中也至少能有原有的m个元素。证毕。

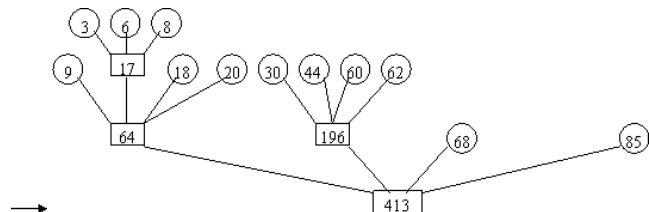


61. 因 $(8-1)\%(3-1)=1$, 故第一次归并时加一个“虚段”。

$$WPL=5*3+(4+5+7+9+10)*2+18*1=103$$

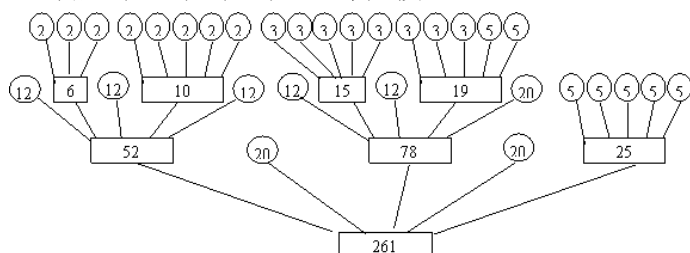
类似叙述:

(1) 加 $4-(12-1)\%(4-1)-1=1$ 个虚段



$$WPL=(3+6+8)*3+(9+18+20+30+44+60+62)*2+(68+85)*1=690$$
 (2) (3) (4) 略。

62. 加 $5-(31-1)\%(5-1)-1=2$ 个虚段。

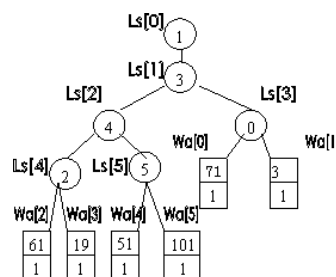


总读写次数为 $2*wp1=800$ 次。

类似叙述(1) (2) (3) 略。

63. 内部排序中的归并排序是在内存中进行的归并排序, 辅助空间为 $O(n)$ 。外部归并排序是将外存中的多个有序子文件合并成一个有序子文件, 将每个子文件中记录读入内存后的排序方法可采用多种内排序方法。外部排序的效率主要取决于读写外存的次数, 即归并的趟数。因为归并的趟数 $s=\lceil \log_k m \rceil$, 其中, m 是归并段个数, k 是归并路数。增大 k 和减少 m 都可减少归并趟数。应用中通过败者树进行多 (k) 路平衡归并和置换-选择排序减少 m , 来提高外部排序的效率。

64. 初始败者树



初始归并段:

$R_1: 3, 19, 31, 51, 61, 71, 100, 101$

$R_2: 9, 17, 19, 20, 30, 50, 55, 90$

$R_3: 6$

类似叙述题略。

65. (1) 4台磁带机：平衡归并只能用2路平衡归并，需归并 $\lceil \log_2 650 \rceil = 10$ 趟。多步归并进行3路归并，按3阶广义斐波那契序列。 $F_{11} < 650 < F_{12} = 653$ ，即应补3个虚段。 进行：12-2=10 趟归并。

$$t_1 = f_{11} + f_{10} + f_9 = 149 + 81 + 44 = 274$$

$$t_2 = f_{11} + f_{10} = 149 + 81 = 230$$

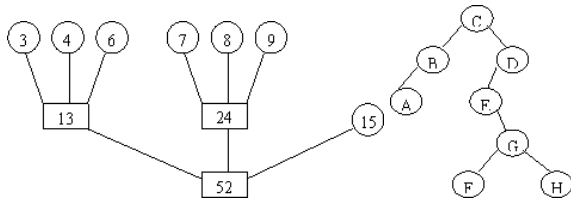
$$t_3 = f_{11} = 149$$

(2) 多步归并排序前五趟归并的情况如下：

T1	$1^{149}+1^{81}+1^{44}$	1步	T1	$1^{81}+1^{44}$	2步	T1	1^{44}	3步	T1		4步
T2	$1^{149}+1^{81}$		T2	1^{81}		T2			T2	9^{44}	
T3	1^{149}		T3			T3	5^{81}		T3	5^{37}	
T4			T4	3^{149}		T4	3^{68}		T4	3^{24}	

T1	17^{24}	5步	T1	17^{11}	注： m^p 表示p个长为m单位的归并段 1^{149} 表示149个长为1个单位的归并段。
T2	9^{20}		T2	9^7	
T3	5^{13}		T3		
T4			T4	31^{13}	类似叙述题略

66. (1)



(2)

类似叙述题略。

67. 外排序用k-路归并($k > 2$)是因为k越小，归并趟数越多，读写外存次数越多，时间效率越低，故，一般应大于最少的2路归并。 若将k-路归并的败者树思想单纯用于内排序，因其由胜者树改进而来，且辅助空间大，完全可由堆排序取代，故将其用于内排序效率并不高。

68. R_1 : 19, 48, 65, 74, 101

R_2 : 3, 17, 20, 21, 21, 33, 53, 99

五. 算法设计题

1. **void** BubbleSort2(**int** a[], **int** n) //相邻两趟向相反方向起泡的冒泡排序算法

{ change=1; low=0; high=n-1; //冒泡的上下界

while(low<high && change)

{ change=0; //设不发生交换

for(i=low; i<high; i++) //从上向下起泡

if(a[i]>a[i+1]) {a[i]<-->a[i+1]; change=1;} //有交换，修改标志change

high--; //修改上界

for(i=high; i>low; i--) //从下向上起泡

if(a[i]<a[i-1]) {a[i]<-->a[i-1]; change=1;} //有交换，修改标志change

low++; //修改下界

} //while

} //BubbleSort2

[算法讨论] 题目中“向上移”理解为向序列的右端，而“向下移”按向序列的左端来处理。

2. **typedef struct** node

{ ElemType data;

struct node *prior, *next;

} node, *DLinkedList;

void TwoWayBubbleSort(DLinkedList la)

//对存储在带头结点的双向链表la中的元素进行双向起泡排序。

{ **int** exchange=1; // 设标记

DLinkedList p, temp, tail;

head=la

//双向链表头，算法过程中是向下起泡的开始结点

```

tail=null;          //双向链表尾，算法过程中是向上起泡的开始结点
while (exchange)
{p=head->next;      //p是工作指针，指向当前结点
exchange=0;        //假定本趟无交换
while (p->next!=tail) // 向下（右）起泡，一趟有一最大元素沉底
if (p->data>p->next->data) //交换两结点指针，涉及6条链
{temp=p->next; exchange=1; //有交换
p->next=temp->next; temp->next->prior=p //先将结点从链表上摘下
temp->next=p; p->prior->next=temp; //将temp插到p结点前
temp->prior=p->prior; p->prior=temp;
}
else p=p->next; //无交换，指针后移
tail=p; //准备向上起泡
p=tail->prior;
while (exchange && p->prior!=head) //向上（左）起泡，一趟有一最小元素冒出
if (p->data<p->prior->data) //交换两结点指针，涉及6条链
{temp=p->prior; exchange=1; //有交换
p->prior=temp->prior; temp->prior->next=p; //先将temp结点从链表上摘下
temp->prior=p; p->next->prior=temp; //将temp插到p结点后（右）
temp->next=p->next; p->next=temp;
}
else p=p->prior; //无交换，指针前移
head=p; //准备向下起泡
} // while (exchange)
} //算法结束
3. PROCEDURE StraightInsertSort (VAR R: listtype; n: integer);
VAR i, j: integer;
BEGIN
FOR i:=2 TO n DO {假定第一个记录有序}
BEGIN
R[0]:=R[i]; j:=i-1; {将待排序记录放进监视哨}
WHILE R[0].key<R[j].key DO {从后向前查找插入位置，同时向后移动记录}
BEGIN R[j+1]:=R[j]; j:=j-1; END;
R[j+1]:=R[0] {将待排序记录放到合适位置}
END {FOR}
END;
4. TYPE pointer=↑node;
node=RECORD key:integer; link:pointer; END;
PROCEDURE LINSORT (L:pointer);
VAR t, p, q, s:pointer;
BEGIN
p:=L↑.link↑.link; {链表至少一个结点，p初始指向链表中第二结点（若存在）}
L↑.link↑.link=NIL; {初始假定第一个记录有序}
WHILE p<>NIL DO
BEGIN q:=p↑.link; {q指向p的后继结点}
s:=L;
WHILE (s↑.link<>NIL AND s↑.link↑.key<p↑.key) DO
s:=s↑.link; {向后找插入位置}
p↑.link:=s↑.link; s↑.link=p; {插入结点}
p:=q; {恢复p指向当前结点}
END {WHILE}
END; {LINSORT}
5. typedef struct
{ int num; float score; } RecType;
void SelectSort (RecType R[51], int n)
{ for(i=1; i<n; i++)
{ //选择第i大的记录，并交换到位
k=i; //假定第i个元素的关键字最大
for(j=i+1; j<=n; j++) //找最大元素的下标
if(R[j].score>R[k].score) k=j;
if(i!=k) R[i] <--> R[k]; //与第i个记录交换
} //for

```

```

    for(i=1; i<=n; i++) //输出成绩
    { printf("%d,%f", R[i].num, R[i].score); if(i%10==0) printf("\n"); }
} //SelectSort

```

6. typedef struct

```

    { int key; datatype info } RecType
void CountSort(RecType a[], b[], int n) //计数排序算法, 将a中记录排序放入b中
{ for(i=0; i<n; i++) //对每一个元素
  { for(j=0, cnt=0; j<n; j++)
    { if(a[j].key<a[i].key) cnt++; //统计关键字比它小的元素个数
      b[cnt]=a[i];
    }
  }
} //Count_Sort

```

(3) 对于有 n 个记录的表, 关键码比较 n^2 次。

(4) 简单选择排序算法比本算法好。简单选择排序比较次数是 $n(n-1)/2$, 且只用一个交换记录的空间; 而这种方法比较次数是 n^2 , 且需要另一数组空间。

[算法讨论]因题目要求“针对表中的每个记录, 扫描待排序的表一趟”, 所以比较次数是 n^2 次。若限制“对任意两个记录之间应该只进行一次比较”, 则可把以上算法中的比较语句改为:

```

for(i=0; i<n; i++) a[i].count=0; //各元素再增加一个计数域, 初始化为0
for(i=0; i<n; i++)
  for(j=i+1; j<n; j++)

```

```

    if(a[i].key<a[j].key) a[j].count++; else a[i].count++;

```

7. [题目分析]保存划分的第一个元素。以平均值作为枢轴, 进行普通的快速排序, 最后枢轴的位置存入已保存的第一个元素, 若此关键字小于平均值, 则它属于左半部, 否则属于右半部。

```

int partition (RecType r[], int l, h)
{ int i=l, j=h, avg=0;
  for(; i<=h; i++) avg+=R[i].key;
  i=l; avg=avg/(h-l+1);
  while (i<j)
  { while (i<j && R[j].key>=avg) j--;
    if (i<j) R[i]=R[j];
    while (i<j && R[i].key<=avg) i++;
    if (i<j) R[j]=R[i];
  }
  if(R[i].key<=avg) return i; else return i-1;
}

```

```

void quicksort (RecType R[], int S, T);
{ if (S<T)
  { k=partition (R, S, T);
    quicksort (R, S, k);
    quicksort (R, k+1, T); }
}

```

8. int Partition(RecType R[], int l, int h)

//一趟快速排序算法, 枢轴记录到位, 并返回其所在位置,

```

{ int i=l; j=h; R[0] = R[i]; x = R[i].key;
  while(i<j)
  { while(i<j && R[j].key>=x) j--;
    if (i<j) R[i] = R[j];
    while(i<j && R[i].key<=x) i++;
    if (i<j) R[j] = R[i];
  } //while
  R[i]=R[0];
  return i;
} //Partition

```

9. [题目分析]以 K_n 为枢轴的一趟快速排序。将上题算法改为以最后一个为枢轴先从前向后再从后向前。

```

int Partition(RecType K[], int l, int n)
{ //交换记录子序列K[1..n]中的记录, 使枢轴记录到位, 并返回其所在位置,
  //此时, 在它之前(后)的记录均不大(小)于它
  int i=l; j=n; K[0] = K[j]; x = K[j].key;
  while(i<j)
  { while(i<j && K[i].key<=x) i++;

```

```

        if (i<j) K[j]=K[i];
        while(i<j && K[j].key>=x) j--;
        if (i<j) K[i]=K[j];
    } //while
    K[i]=K[0]; return i;
} //Partition

```

10. [题目分析]把待查记录看作枢轴，先由后向前依次比较，若小于枢轴，则从前向后，直到查找成功返回其位置或失败返回0为止。

```

int index (RecType R[],int l,h,datatype key)
{
    int i=l, j=h;
    while (i<j)
    {
        while (i<=j && R[j].key>key) j--;
        if (R[j].key==key) return j;
        while (i<=j && R[i].key<key) i++;
        if (R[i].key==key) return i;
    }
    printf( "Not find" ); return 0;
} //index

```

11. (1) [题目分析]从第 n 个记录开始依次与其双亲 ($n/2$) 比较，若大于双亲则交换，继而与其双亲的双亲比较，以此类推直到根为止。

```

void sift(RecType R[],int n)
{
    //假设 R[1..n-1]是大堆，本算法把R[1..n]调成大堆
    j=n; R[0]=R[j];
    for (i=n/2;i>=1;i=i/2)
        if (R[0].key>R[i].key) { R[j]=R[i];j=i;} else break;
    R[j]=R[0];
} //sift

```

```

(2) void HeapBuilder(RecType R[],int n)
    { for (i=2;i<=n;i++) sift (R,i); }

```

```

12. void sort (RecType K[],int n)
    { for (i=1;i<=n;i++) T[i]=i;
      for (i=1;i<n;i++)
          for (j=1;j<=n-i;j++)
              if (K[T[j]]>K[T[j+1]]) {t=T[j];T[j]=T[j+1];T[j+1]=t;}
    } //sort

```

[算法讨论] 上述算法得到辅助地址表， $T[i]$ 的值是排序后 K 的第 i 个记录，要使序列 K 有序，则要按 T 再物理地重排 K 的各记录。算法如下：

```

void Rearrange(RecType K[],int T[],n)
//对有n个记录的序列K，按其辅助地址表T进行物理非递减排序
{for(i=1;i<=n;i++)
    if (T[i]!=i)
        {j=i; rc=K[i];          //暂存记录K[i]
         while (T[j]!=i)        //调整K[T[j]]到T[j]=i为止
             {m=T[j]; K[j]=K[m]; T[j]=j; j=m;}
         K[j]=rc; T[j]=j;       //记录R[i]到位
        } //if
} //Rearrange

```

13. (1) 堆排序是对树型选择排序的改进，克服了树型选择排序的缺点，其定义在前面已多次谈到，请参见上面“四、应用题”的43题和45题(4)。“筛选”是堆排序的基础算法。由于堆可以看作具有 n 个结点的完全二叉树，建堆过程是从待排序序列第一个非终端结点 $n/2$ 开始，直到根结点，进行“筛选”的过程。堆建成后，即可选得一个关键字最大或最小的记录。然后堆顶元素与序列中最后一个元素交换，再将序列中前 $n-1$ 记录重新调整为堆，可选得一个关键字次大或次小的记录。依次类推，则可得到元素的有序序列。

```

(2) void Sift(RecType R[], int i, int m)
{
    //假设R[i+1..m]中各元素满足堆的定义，本算法调整R[i]使序列R[i..m]中各元素满足堆的性质
    R[0]=R[i];
    for(j=2*i; j<=m; j*=2)
    {
        if(j<m && R[j].key<R[j+1].key) j++; //建大根堆
        if(R[0].key<R[j].key) { R[i]=R[j]; i=j;} else break;
    } //for
    R[i]=R[0];
} //Sift

void HeapSort(RecType R[], int n)
{
    //对记录序列R[1..n]进行堆排序。

```

```

for(i=n/2;i>0;i--) Sift(R,i,n);
for(i=n;i>1;i--){ R[1]<-->R[i]; Sift(R,1,i-1);} //for
} //HeapSort

```

(3) 堆排序的时间主要由建堆和筛选两部分时间开销构成。对深度为 h 的堆，“筛选”所需进行的关键字比较的次数至多为 $2(h-1)$ ；对 n 个关键字，建成深度为 $h(=\lceil \log_2 n \rceil + 1)$ 的堆，所需进行的关键字比较的次数至多 $C(n)$ ，它满足下式：

$$\begin{aligned}
 C(n) &= \sum_{i=h-1}^1 2^{i-1} \times 2(h-1) = \sum_{i=h-1}^1 2^i \times (h-1) \\
 &= 2^{h-1} + 2^{h-2} \times 2 + 2^{h-3} \times 3 + \dots + 2 \times (h-1) \\
 &= 2^h (1/2 + 2/2^2 + 3/2^3 + \dots + (h-1)/2^{h-1}) \\
 &\leq 2^h \times 2 \leq 2 \times 2^{(\log_2 n)+1} = 4n
 \end{aligned}$$

调整“堆顶” $n-1$ 次，总共进行的关键字比较的次数不超过： $2(\lceil \log_2(n-1) \rceil + 1 + \dots + \log_2 2) < 2n(\lceil \log_2 n \rceil)$ 因此，堆排序的时间复杂度为 $O(n \log_2 n)$ 。

14. PROC LinkedListSelectSort(head: pointer);

// 本算法一趟找出一个关键字最小的结点，其数据和当前结点进行交换；若要交换指针，则须记下

// 当前结点和最小结点的前驱指针

p:=head↑.next;

WHILE p<>NIL DO

[q:=p↑.next; r:=p; // 设 r 是指向关键字最小的结点的指针

WHILE <q<>NIL DO

[IF q↑.data<r↑.data THEN r:=q;

q:=q↑.next;

]

IF r<>p THEN r↑.data<-->p↑.data

p:=p↑.next;

]

ENDP;

15. void QuickSort(rectype r[n+1]; int n)

// 对 r[1..n] 进行快速排序的非递归算法

{typedef struct

{ int low,high; }node

node s[n+1]; // 栈，容量足够大

int quickpass(rectype r[],int,int); // 函数声明

int top=1; s[top].low=1; s[top].high=n;

while (top>0)

{ss=s[top].low; tt=s[top].high; top--;

if (ss<tt)

{k=quickpass(r,ss,tt);

if (k-ss>1) {s[++top].low=ss; s[top].high=k-1;}

if (tt-k>1) {s[++top].low=k+1; s[top].high=tt;}

}

} // 算法结束

int quickpass(rectype r[];int s,t)

{i=s; j=t; rp=r[i]; x=r[i].key;

while (i<j)

{while (i<j && x<=r[j].key) j--;

if (i<j) r[i++]=r[j];

while (i<j && x>=r[j].key) i++;

if (i<j) r[j--]=r[i];;

]

r[i]=rp;

return (i);

} // 一次划分算法结束

[算法讨论] 可对以上算法进行两点改进：一是在一次划分后，先处理较短部分，较长的子序列进栈；二是用“三者取中法”改善快速排序在最坏情况下的性能。下面是部分语句片段：

int top=1; s[top].low=1; s[top].high=n;

ss=s[top].low; tt=s[top].high; top--; flag=true;

while (flag || top>0)

{k=quickpass(r,ss,tt);

if (k-ss>tt-k) // 一趟排序后分割成左右两部分

{if (k-ss>1) // 左部子序列长度大于右部，左部进栈

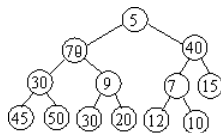
{s[++top].low=ss; s[top].high=k-1; }

if (tt-k>1) ss=k+1; // 右部短的直接处理

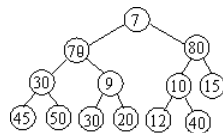
```

        else flag=false; // 右部处理完, 需退栈
    }
    else if (tt-k>1) //右部子序列长度大于左部, 右部进栈
        {s[++top].low=k+1; s[top].high=tt; }
        if (k-ss>1) tt=k-1 // 左部短的直接处理
        else flag=false // 左部处理完, 需退栈
    }
    if (!flag && top>0)
        {ss=s[top].low; tt=s[top].high; top--; flag=true;}
    } // end of while (flag || top>0)
} // 算法结束
int quickpass(rectype r[];int s,t)
// 用“三者取中法”进行快速排序的一次划分
{ int i=s, j=t, mid=(s+t)/2;
  rectype tmp;
  if (r[i].key>r[mid].key) {tmp=r[i];r[i]=r[mid];r[mid]=tmp }
  if (r[mid].key>r[j].key)
      {tmp=r[j];r[j]=r[mid];
       if (tmp>r[i]) r[mid]=tmp; else {r[mid]=r[i];r[i]=tmp }
      }
  {tmp=r[i];r[i]=r[mid];r[mid]=tmp }
  // 三者取中: 最佳2次比较3次赋值; 最差3次比较10次赋值
  rp=r[i]; x=r[i].key;
  while (i<j)
      {while (i<j && x<=r[j].key) j--;
       if (i<j) r[i++]=r[j];
       while (i<j && x>=r[j].key) i++;
       if (i<j) r[j--]=r[i];
      }
  r[i]=rp;
  return (i);
} // 一次划分算法结束

```



16. (1)
加入5后



(2)
加入80后

(3) [题目分析]从插入位置进行调整, 调整过程由下到上。首先根据元素个数求出插入元素所在层数, 以确定其插入层是最大层还是最小层。若插入元素在最大层, 则先比较插入元素是否比双亲小, 如是, 则先交换, 之后, 将小堆与祖先调堆, 直到满足小堆定义或到达根结点; 若插入元素不小于双亲, 则调大堆, 直到满足大堆定义。若插入结点在最小层, 则先比较插入元素是否比双亲大, 如是, 则先交换, 之后, 将大堆与祖先调堆; 若插入结点在最小层且小于双亲, 则将小堆与祖先调堆, 直到满足小堆定义或到达根结点。

```

(4) void MinMaxHeapIns(RecType R[],int n)
{ //假设R[1..n-1]是最小最大堆, 插入第n个元素, 把R[1..n]调成最小最大堆
  j=n; R[0]=R[j];
  h=⌈log2n⌉+1; //求高度
  if (h%2==0) //插入元素在偶数层, 是最大层
      {i=n/2;
       if (R[0].key<R[i].key) //插入元素小于双亲, 先与双亲交换, 然后调小堆
           {R[j]=R[i];
            j=i/4;
            while (j>0 && R[j]>R[i]) //调小堆
                {R[i]=R[j]; i=j; j=i/4; }
            R[i]=R[0];
           }
      }
  else //插入元素大于双亲, 调大堆
      {i=n; j=i/4;

```

```

        while (j>0 && R[j]<R[i])
            {R[i]=R[j]; i=j; j=i/4; }
        R[i]=R[0];
    }
}
else //插入元素在奇数层, 是最小层
{
    i=n/2;
    if (R[0].key>R[i].key) //插入元素大于双亲, 先与双亲交换, 然后调大堆
        {R[j]=R[i];
        j=i/4;
        while (j>0 && R[j]<R[i]) //调大堆
            {R[i]=R[j]; i=j; j=i/4; }
        R[i]=R[0];
        }
    else //插入元素小于双亲, 调小堆
        {i=n; j=i/4;
        while (j>0 && R[j]>R[i])
            {R[i]=R[j]; i=j; j=i/4; }
        R[i]=R[0];
        }
}
} //MinMaxHeapIns
17. void BiInsertSort(RecType R[], int n)
    { //二路插入排序的算法
    int d[n+1]; //辅助存储
    d[1]=R[1]; first=1; final=1;
    for(i=2; i<=n; i++)
        { if(R[i].key>=d[1].key) //插入后部
            { low=1; high=final;
            while (low<=high) //折半查找插入位置
                { m=(low+high)/2;
                if(R[i].key< d[m].key) high=m-1; else low=m+1;
                } //while
            for (j=final; j>=high+1; j--) d[j+1] = d[j]; //移动元素
            d[high+1]=R[i]; final++; //插入有序位置
            }
        else //插入前部
            { if(first==1){ first=n; d[n]=R[i]; }
            else{ low=first; high=n;
            while (low<=high)
                { m=(low+high)/2;
                if(R[i].key< d[m].key) high=m-1; else low=m+1;
                } //while
            for (j=first; j<=high; j++) d[j+1] = d[j]; //移动元素
            d[high+1]=R[i]; first++;
            } //if
            } //if
        } //for
    R[1] =d[1];
    for(i=first%n+1, j=2; i!=first; i=i%n+1, j++) R[j] =d[i]; //将序列复制回去
} //BiInsertSort

```

18. 手工模拟排序过程略。

```

#define n 待排序记录的个数
typedef struct
{
    int key[d]; //关键字由d个分量组成
    int next; //静态链域
    AnyType other; //记录的其它数据域
} SLRecType;
SLRecType R[n+1]; //R[1..n]存放n个记录
typedef struct
{
    int f, e; //队列的头、尾指针
} SLQueue;
SLQueue B[m] //用队列表示桶, 共m个

```

```

int RadixSort(SLRecType R[], int n)
{ //对R[1..n]进行基数排序, 返回收集用的链头指针
  for(i=1; i<n; i++) R[i].next=i+1; //将R[1..n]链成一个静态链表
  R[n].next=-1; p=1; //将初始链表的终端结点指针置空, p指向链表的第一个结点
  for(j=d-1; j>=0; j--) //进行d趟排序
  {
    for(i=0; i<m; i++) { B[i].f=-1; B[i].e=-1; } //初始化桶
    while(p!=-1) //按关键字的第j个分量进行分配
    {
      k=R[p].key[j]; //k为桶的序号
      if(B[k].f==-1) B[k].f=p; //将R[p]链到桶头
      else R[B[k].e].next=p; //将R[p]链到桶尾
      B[k].e=p; //修改桶的尾指针,
      p=R[p].next; //扫描下一个记录
    } //while
    i=0;
    while(B[i].f==-1) i++; //找第一个非空的桶
    t=B[i].e; p=B[i].f //p为收集链表的头指针, t为尾指针
    while(i<m-1)
    {
      i++;
      if(B[i].f!=-1) { R[t].next=B[i].f; t=B[i].e; } //连接非空桶
    } //while
    R[t].next=-1; //本趟收集完毕, 将链表的终端结点指针置空
  } //for
  return p;
} //RadixSort

```

19. [题目分析] 本题是基数排序的特殊情况, 关键字只含一位数字的整数。

```

typedef struct
{
  int key;
  int next;
} SLRecType;
SLRecType R[N+1];
typedef struct
{
  int, f, e;
} SLQueue;
SLQueue B[10];
int Radixsort(SLRecType R[], int n) //设各关键字已输入到R数组中
{
  for (i=1; i<n; i++) R[i].next=i+1;
  R[n].next=-1; p=1; // -1表示静态链表结束
  for (i=0; i<=9; i++) { B[i].f=-1; B[i].e=-1; } //设置队头队尾指针初值
  while (p!=-1) //一趟分配
  {
    k=R[p].key; //取关键字
    if(B[k].f==-1) B[k].f=p; //修改队头指针
    else R[B[k].e].next=p;
    B[k].e=p;
    p=R[p].next; //下一记录
  }
  i=0; //一趟收集
  while (B[i].f==-1) i++;
  t=B[i].e; p=B[i].f;
  while (i<9)
  {
    i++;
    if (B[i].f!=-1)
    {
      R[t].next=B[i].f; t=B[i].e;
    }
    R[t].next=-1;
  }
  return p; //返回第一个记录指针
}

```

[算法讨论] 若关键字含d位, 则要进行d趟分配和d趟收集。关键字最好放入字符数组, 以便取关键字的某位。

20. void Adjust(int T[], int s)

```

{ //选得最小关键字记录后, 从叶到根调整败者树, 选下一个最小关键字
  //沿从叶子结点R[s]到根结点T[0]的路径调整败者树
  t=(s+k)/2; //T[t]是R[s]的双亲结点
  while(t>0)

```

```

    { if(R[s].key>R[T[t]].key) s<-->T[t]; //s指示新的胜者
      t=t/2;
    } //while
    T[0]=s;
  } //Adjust
  void CreateLoserTree(int T[])
  { //建立败者树, 已知R[0]到R[k-1]为完全二叉树T的叶子结点, 存有k个关键字, 沿
    //从叶子到根的k条路径将T调整为败者树
    R[k].key=MINKEY; //MINKEY是最小关键字
    for(i=0;i<k;i++) T[i]=k; //设置T中“败者”的初值
    //依次从R[k-1], R[k-2], ..., R[0]出发调整败者
    for(i=k-1;k>=0;i--) Adjust(T, i);
  } //CreateLoserTree

```

21、[题目分析]利用快速排序思想解决。由于要求“对每粒砾石的颜色只能看一次”，设3个指针*i*, *j*和*k*, 分别指向红色、白色砾石的后一位置和待处理的当前元素。从*k=n*开始, 从右向左搜索, 若该元素是兰色, 则元素不动, 指针左移(即*k-1*)；若当前元素是红色砾石, 分*i>=j*(这时尚没有白色砾石)和*i<j*两种情况。前一情况执行第*i*个元素和第*k*个元素交换, 之后*i+1*；后一情况, *i*所指的元素已处理过(白色), *j*所指的元素尚未处理, 应先将*i*和*j*所指元素交换, 再将*i*和*k*所指元素交换。对当前元素是白色砾石的情况, 也可类似处理。

为方便处理, 将三种砾石的颜色用整数1、2和3表示。

```

void QkSort(rectype r[], int n)
// r为含有n个元素的线性表, 元素是具有红、白和兰色的砾石, 用顺序存储结构存储,
// 本算法对其排序, 使所有红色砾石在前, 白色居中, 兰色在最后。
{ int i=1, j=1, k=n, temp;
  while (k!=j)
  { while (r[k].key==3) k--; // 当前元素是兰色砾石, 指针左移
    if (r[k].key==1) // 当前元素是红色砾石
    { if (i>=j) { temp=r[k]; r[k]=r[i]; r[i]=temp; i++; }
      // 左侧只有红色砾石, 交换r[k]和r[i]
    } else { temp=r[j]; r[j]=r[i]; r[i]=temp; j++;
      // 左侧已有红色和白色砾石, 先交换白色砾石到位
      temp=r[k]; r[k]=r[i]; r[i]=temp; i++;
      // 白色砾石(i所指)和待处理砾石(j所指)
    } // 再交换r[k]和r[i], 使红色砾石入位。
    if (r[k].key==2)
    { if (i<=j) { temp=r[k]; r[k]=r[j]; r[j]=temp; j++; }
      // 左侧已有白色砾石, 交换r[k]和r[j]
    } else { temp=r[k]; r[k]=r[i]; r[i]=temp; j=i+1; }
      // i、j分别指向红、白色砾石的后一位置
    } // while
    if (r[k]==2) j++; /* 处理最后一粒砾石
  else if (r[k]==1) { temp=r[j]; r[j]=r[i]; r[i]=temp; i++; j++; }
  // 最后红、白、兰色砾石的个数分别为: i-1; j-i; n-j+1
} // 结束QkSort算法

```

[算法讨论]若将*j*(上面指向白色)看作工作指针, 将*r[1..j-1]*作为红色, *r[j..k-1]*为白色, *r[k..n]*为兰色。从*j=1*开始查看, 若*r[j]*为白色, 则*j=j+1*；若*r[j]*为红色, 则交换*r[j]*与*r[i]*, 且*j=j+1*, *i=i+1*；若*r[j]*为兰色, 则交换*r[j]*与*r[k]*; *k=k-1*。算法进行到*j>k*为止。

算法片段如下:

```

int i=1, j=1, k=n;
while(j<=k)
{ if (r[j]==1) // 当前元素是红色
  { temp=r[i]; r[i]=r[j]; r[j]=temp; i++; j++; }
  else if (r[j]==2) j++; // 当前元素是白色
  else // (r[j]==3 当前元素是兰色
  { temp=r[j]; r[j]=r[k]; r[k]=temp; k--; }
}

```

对比两种算法, 可以看出, 正确选择变量(指针)的重要性。

22、[题目分析]根据定义, DEAP是一棵完全二叉树, 树根不包含元素, 其左子树是一小堆(MINHEAP, 下称小堆), 其右子树是一大堆(MAXHEAP, 下称大堆), 故左右子树可分别用一维数组*l[]*和*r[]*存储, 用*m*和*n*分别表示当前两完全二叉树的结点数, 左右子树的高度差至多为1, 且左子树的高度始终大于等于右子树的高度。

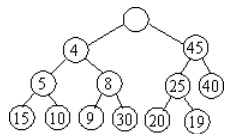
我们再分析插入情况: 当均为空二叉树或满二叉树($m=2^h-1$)时, 应在小堆插入; 小堆满(二叉树)后在大堆插入。即当 $m \geq n$ 且 $m < 2^h-1$ 且 $\lceil \log_2 m \rceil - \lceil \log_2 n \rceil \leq 1$ 在小堆插入, 否则在大堆插入。

最后分析调堆情况: 在小堆*m*处插入结点*x*后, 若*x*的值不大于大堆的*m/2*结点的值, 则在小堆调堆; 否则, 结点*x*与大堆的*m/2*结点交换, 然后进行大堆调堆。在大堆*n*处插入结点*x*后, 若*x*不小于小堆的*n*结点, 则在大堆调

堆；否则，结点x与小堆的n结点交换，然后进行小堆调堆。

(1) 在DEAP中插入结点4后的结果如图：

4先插入到大堆，因为4小于小堆中对应位置的19，所以和19交换。交换后只需调整小堆，从叶子到根结点。这时，大堆不需调整，因为插入小堆19时，要求19必须小于对应大堆双亲位置的25，否则，要进行交换。



```
void InsertDEAP(int l[], r[], m, n, x)
//在DEAP中插入元素x, l[]是小堆, r[]是大堆, m和n分别是两堆的元素个数, x是待插入元素。
{if (m>n && m<>2h-1 && ⌊log2m⌋-⌊log2n⌋≤1) // 在小堆插入x, h是二叉树的高度
    {m++; //m增1
    if (x>r[m/2]) //若x大于大堆中相应结点的双亲, 则进行交换
        {l[m]=r[m/2];
        c=m/2; f=c/2;
        while (f>0 && r[f]<x) //调大堆
            {r[c]=r[f]; c=f; f=c/2;}
        r[c]=x;
    } //结束调大堆
    else //调小堆
        {c=m; f=c/2;
        while (f>0 && l[f]>x)
            {l[c]=l[f]; c=f; f=c/2;}
        l[c]=x;
    }
else //在大堆插入x
    {n++; //n增1
    if (x<l[n]) //若x小于小堆中相应结点, 则进行交换
        {r[n]=l[n];
        c=n; f=c/2;
        while (f>0 && l[f]>x) //调小堆
            {l[c]=l[f]; c=f; f=c/2;}
        l[c]=x;
    } //结束调小堆
    else //调大堆
        {c=n; f=c/2;
        while (f>0 && r[f]<x)
            {r[c]=r[f]; c=f; f=c/2;}
        r[c]=x;
    }
} //结束InsertDEAP算法
```

第十一章 文件

一. 选择题

1. D	2. A	3. B	4. A	5. B	6. B	7. B													
------	------	------	------	------	------	------	--	--	--	--	--	--	--	--	--	--	--	--	--

二. 判断题

1. √	2. √	3. ×	4. √	5. ×	6. ×	7. ×	8. ×	9. ×	10. ×	11. √	
------	------	------	------	------	------	------	------	------	-------	-------	--

三. 填空题

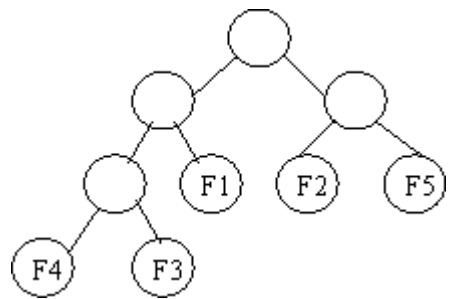
- 操作系统文件 数据库 2. 单关键字文件 多关键字文件
- (1) 数据库 (2) 文本 (3) 顺序组织 (4) 随机组织 (5) 链组织
- (6) 随机组织 (7) m (8) é/2ù (9) 2 (10) k
4. 记录 数据项 5. 串联文件 6. 第I-1 7. 随机
8. 提高查找速度 9. 树 10. 检索记录快
11. (1) 关键字 (2) 记录号 (3) 记录号 (4) 顺序 (5) 直接
12. 构造散列函数 解决冲突的方法 13. 索引集 顺序集 数据集
14. 分配和释放存储空间 重组 对插入的记录

四. 应用题

- 文件是由大量性质相同的记录组成的集合, 按记录类型不同可分为操作系统文件和数据库文件。
- 文件的基本组织方式有顺序组织、索引组织、散列组织和链组织。文件的存储结构可以采用将基本组织结合的方法, 常用的结构有顺序结构、索引结构、散列结构。
 - (1) 顺序结构, 相应文件为顺序文件, 其记录按存入文件的先后次序顺序存放。顺序文件本质上就是顺序表。若逻辑上相邻的两个记录在存储位置上相邻, 则为连续文件; 若记录之间以指针相链接, 则称为串联文件。顺序文件只能顺序存取, 要更新某个记录, 必须复制整个文件。顺序文件连续存取的速度快, 主要适用于顺序存取, 批量修改的情况。
 - (2) 带索引的结构, 相应文件为索引文件。索引文件包括索引表和数据表, 索引表中的索引项包括数据表中数据的关键字和相应地址, 索引表有序, 其物理顺序体现了文件的逻辑次序, 实现了文件的线性结构。索引文件只能是磁盘文件, 既能顺序存取, 又能随机存取。
 - (3) 散列结构, 也称计算寻址结构, 相应文件称为散列文件, 其记录是根据关键字值经散列函数计算确定其地址, 存取速度快, 不需索引, 节省存储空间。不能顺序存取, 只能随机存取。其它文件均由以上文件派生而得。

文件采用何种存储结构应综合考虑各种因素, 如: 存储介质类型、记录的类型、大小和关键字的数目以及对文件作何种操作。
- 在主文件外, 再建立索引表指示关键字及其物理记录的地址间一一对应关系。这种由索引表和主文件一起构成的文件称为索引文件。索引表依关键字有序。主文件若按关键字有序称为索引顺序文件, 否则称为索引非顺序文件 (通常简称索引文件)。索引顺序文件因主文件有序, 一般用稀疏索引, 占用空间较少。常用索引顺序文件有ISAM和VSAM。ISAM采用静态索引结构, 而VSAM采用B+树的动态索引结构。索引文件既能顺序存取, 也能随机存取。
- 在索引文件中, 若 (数据区) 主文件中关键字有序, 则文件称为索引顺序文件, 参见上题3。
- ISAM是专为磁盘存取设计的文件组织方式。即使主文件关键字有序, 但因磁盘是以盘组、柱面和磁道 (盘面) 三级地址存取的设备, 因此通常对磁盘上的数据文件建立盘组、柱面和磁道 (盘面) 三级索引。在ISAM文件上检索记录时, 先从主索引 (柱面索引的索引) 找到相应柱面索引。再从柱面索引找到记录所在柱面的磁道索引, 最后从磁道索引找到记录所在磁道的第一个记录的位置, 由此出发在该磁道上进行顺序查找直到查到为止; 反之, 若找遍该磁道而未找到所查记录, 则文件中无此记录。
- ISAM是一种专为磁盘存取设计的文件组织形式, 采用静态索引结构, 对磁盘上的数据文件建立盘组、柱面、磁道三级索引。ISAM文件中记录按关键字顺序存放, 插入记录时需移动记录并将同一磁道上最后的一个记录移至溢出区, 同时修改磁道索引项, 删除记录只需在存储位置作标记, 不需移动记录 and 修改指针。经过多次插入和删除记录后, 文件结构变得不合理, 需周期整理ISAM文件。
- VSAM文件采用B+树动态索引结构, 文件只有控制区间和控制区域等逻辑存储单位, 与外存储器中柱面、磁道等具体存储单位没有必然联系。VSAM文件结构包括索引集、顺序集和数据集三部分, 记录存于数据集中, 顺序集和索引集构成B+树, 作为文件的索引部分可实现顺链查找和从根结点开始的随机查找。
- 与ISAM文件相比, VSAM文件有如下优点: 动态分配和释放存储空间, 不需对文件进行重组; 能保持较高的查找效率, 且查找先后插入记录所需时间相同。因此, 基于B+树的VSAM文件通常作为大型索引顺序文件的标准组织。
- ISAM文件有三级索引: 磁盘组、柱面和磁道, 柱面索引存放在某个柱面上, 若柱面索引较大, 占多个磁道时, 可建立柱面索引的索引—主索引。故本题中所指的两级索引是盘组和磁道。
- 倒排文件是一种多关键字的文件, 主数据文件按关键字顺序构成串联文件, 并建立主关键字索引。对次关键字也建立索引, 该索引称为倒排表。倒排表包括两项, 一项是次关键字, 另一项是具有同一次关键字值的记录的物理记录号 (若数据文件非串联文件, 而是索引顺序文件—如ISAM, 则倒排表中存放记录的主关键字而不是物理记录号)。倒排表作索引的优点是索引记录快, 缺点是维护困难。在同一索引表中, 不同的关键字其记录数不同, 各倒排表的长度不同, 同一倒排表中各项长度也不相等。
- 因倒排文件组织中, 倒排表有关键字值及同一关键字值的记录的所有物理记录号, 可方便地查询具有同一关键字值的所有记录; 而多重表文件中次关键字索引结构不同, 删除关键字域后查询性能受到影响。

10. 多重表文件是把索引与链接结合而形成的组织方式。记录按主关键字顺序构成一个串联文件，建立主关键字的索引（主索引）。对每一次关键字建立次关键字索引，具有同一关键字的记录构成一个链表。主索引为非稠密索引，次索引为稠密索引，每个索引项包括次关键字，头指针和链表长度。多重表文件易于编程，也易于插入，但删除繁琐。需在各次关键字链表中删除。倒排文件的特点见上面题8。
11. 倒排表作索引的优点是索引记录快，因为从次关键字值直接找到各相关记录的物理记录号，倒排因此而得名（因通常的查询是从关键字查到记录）。在插入和删除记录时，倒排表随之修改，倒排表中具有相同次关键字的记录号是有序的。
12. 排表有两项，一是次关键字值，二是具有相同次关键字值的物理记录号，这些记录号有序且顺序存储，不使用多重表中的指针链接，因而节省了空间。
13. （1）顺序文件只能顺序查找，优点是批量检索速度快，不适于单个记录的检索。顺序文件不能象顺序表那样插入、删除和修改，因文件中的记录不能象向量空间中的元素那样“移动”，只能通过复制整个文件实现上述操作。
- （2）索引非顺序文件适合随机存取，不适合顺序存取，因主关键字未排序，若顺序存取会引起磁头频繁移动。索引顺序文件是最常用的文件组织，因主文件有序，既可顺序存取也可随机存取。索引非顺序文件是稠密索引，可以“预查找”，索引顺序文件是稀疏索引，不能“预查找”，但由于索引占空间较少，管理要求低，提高了索引的查找速度。
- （3）散列文件也称直接存取文件，根据关键字的散列函数值和处理冲突的方法，将记录散列到外存上。这种文件组织只适用于像磁盘那样的直接存取设备，其优点是文件随机存放，记录不必排序，插入、删除方便，存取速度快，无需索引区，节省存储空间。缺点是散列文件不能顺序存取，且只限于简单查询。经多次插入、删除后，文件结构不合理，需重组文件，这很费时。
14. 类似最优二叉树（哈夫曼树），可先合并含较少记录的文件，后合并较多记录的文件，使移动次数减少。见下面的哈夫曼树。



15. [问题分析]在职务项中增加一个指针项，指向其领导者。因题目中未提出具体的隶属关系，如哪个系的系主任，哪个系哪个室的室主任，哪个室的教员等。这里假设每个室主任隶属于他前边离他最近的那个系主任，每个教员隶属于他前边离他最近的那个室主任，见下面多重表文件。在职称项中增加一个指针项，指向同一职称的下一个职工，增加一个次关键字索引表：

关键字	头指针	长度
讲师	001	2
副教授	004	2
教授	002	6

“职称”索引表

记录号	职工号	职工姓名	职务		职称	
01	001	张军	教员	04	讲师	08
02	002	沈灵	系主任	03	教授	03
03	003	叶明	校长	ù	教授	05
04	004	张莲	室主任	02	副教授	10
05	005	叶宏	系主任	03	教授	06
06	006	周芳	教员	04	教授	07
07	007	刘光	系主任	03	教授	09
08	008	黄兵	教员	04	讲师	ù

09 10	009 010 ...	李民 赵松 ...	室主任 教员 ...	07 09	教授 副教授 ...	ù ù
----------	-------------------	-----------------	------------------	----------	------------------	--------

多重表文件