

答案

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) $n(n+1)/2$ 。
- $1 + (1+2) + (1+2+3) + \dots + (1+2+\dots+n) = n(n+1)(n+2)/6$ $O(n^3)$
- $\log_2 n$ 12. $n \log_2 n$ 13. $\log_2 n^2$ 14. $(n+3)(n-2)/2$ 15. $O(n)$
- ① (1) 1 (2) 1 (3) $f(m, n-1)$ (4) n ② 9 17. $n(n-1)/2$

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

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层
 报名热线：025-83535877、18951896587、18951896993、18951896967

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。

4. (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。数据的运算是对于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。

(2) 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。

(3) 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。

(4) 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、完整的刻画了问题的基本特征；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。）

5. 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）。

6. (1) 见上面题 3 (2) 见上面题 4 (3) 见上面题 3

(4) 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。

(5) 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。

(6) 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：

$$F_n = F_{n-1} + F_{n-2}$$

$$= 2F_{n-2} + F_{n-3}$$

$$= 3F_{n-3} + 2F_{n-4}$$

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

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. √								

部分答案解释如下。

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** $pa \rightarrow data \neq pb \rightarrow data$

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

(3) **return**(inclusion(pa, pb));

(4) pb:=pb^.next;

(5) **return**(false);

非递归算法:

(1)pre:=pb; (2) pa<>NIL AND pb<>NIL AND pb^.data=pa^.data (3)pa:=pa^.next; pb:=pb->next;

(4)pb:=pre^.next;pre:=pb;pa:=pa^.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^.data:=k D. q^.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^.next:=p; (3) r^.next:=q; (4) **IF** (q=NIL) **THEN** r^.next:=p;

22. A: r^.link^.data<>max **AND** q^.link^.data<>max

B: r:=r^.link C: q^.link D: q^.link E: r^.link F: r^.link

G: r:=s (或 r:=r^.link) H: r:=r^.link I: q^.link:=s^.link

23. (1) 1a (2) 0 (3) j<i-1 (4) p↑.next (5) i<1

24. (1) head^.left:=s //head 的前驱指针指向插入结点

(2) j:=1;

(3) p:=p^.right //工作指针后移

(4) s^.left:=p

(5) p^.right^.left:=s; //p 后继的前驱是 s

(6) s^.left:=p;

25. (1) i<=L.last //L.last 为元素个数

(2) j:=j+1 //有值不相等的元素

(3) L.elem[j]:=L.elem[i] //元素前移

(4) L.last:=j //元素个数

26. (A) p^.link:=q; //拉上链,前驱指向后继

(B) p:=q; //新的前驱

(C) p^.link:=head; //形成循环链表

(D) j:=0; //计数器,记被删结点

(E) q:=p^.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) $p := p \rightarrow next$ // 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) $p \neq null$ // 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) $L = L \rightarrow next$; // 暂存后继
 (2) $q = L$; // 待逆置结点
 (3) $L = p$; // 头指针仍为 L
32. (1) $p \rightarrow next \neq p_0$ (2) $r := p \rightarrow next$ (3) $p \rightarrow next := q_0$;
 (4) $q_0 := p$; (5) $p := r$
33. (1) r (2) NIL (3) $x < head \rightarrow data$ (4) $p \rightarrow data < x$
 (5) $p := p \rightarrow next$ (6) $p \rightarrow data > x$; (7) r (8) p
 (9) r (10) NIL (11) NIL
34. (1) $pa \neq ha$ // 或 $pa \rightarrow exp \neq -1$
 (2) $pa \rightarrow exp == 0$ // 若指数为 0, 即本项为常数项
 (3) $q \rightarrow next = pa \rightarrow next$ // 删常数项
 (4) $q \rightarrow next$ // 取下一元素
 (5) $= pa \rightarrow coef * pa \rightarrow exp$
 (6) $--$ // 指数项减 1
 (7) pa // 前驱后移, 或 $q \rightarrow next$
 (8) $pa \rightarrow next$ // 取下一元素
35. (1) $q := p$; // q 是工作指针 p 的前驱
 (2) $p \rightarrow data > m$ // p 是工作指针
 (3) $r := q$; // r 记最大值的前驱,
 (4) $q := p$; // 或 $q := q \rightarrow next$;
 (5) $r \rightarrow next := q \rightarrow next$; // 或 $r \rightarrow next := r \rightarrow next \rightarrow next$ 删最大值结点
36. (1) $L \rightarrow next = null$ // 置空链表, 然后将原链表结点逐个插入到有序表中
 (2) $p \neq null$ // 当链表尚未到尾, p 为工作指针
 (3) $q \neq null$ // 查 p 结点在链表中的插入位置, 这时 q 是工作指针。
 (4) $p \rightarrow next = r \rightarrow next$ // 将 p 结点链入链表中
 (5) $r \rightarrow next = p$ // 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) $(A \neq null \ \&\& \ B \neq null)$ // 两均未空时循环
 (2) $A \rightarrow element == B \rightarrow element$ // 两表中相等元素不作结果元素
 (3) $B = B \rightarrow link$ // 向后移动 B 表指针
 (4) $A \neq null$ // 将 A 表剩余部分放入结果表中
 (5) $last \rightarrow link = null$ // 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。

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

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 sqliisttp=**RECORD**

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

last:0..maxlen;

END;

链式存储结构的定义是:

TYPE pointer= \uparrow 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 pre=head;

while(pre \rightarrow next!=p) pre=pre \rightarrow next;

s \rightarrow next=p; pre \rightarrow next=s;

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

(1) **while**(p!=null && p \rightarrow data!=X) p=p \rightarrow next;

if(p==null) **return**(null); // 查找失败

else return(p); // 查找成功

(2) **while**(p!=null && p \rightarrow data<X) p=p \rightarrow next;

if(p==null || p \rightarrow data>X) **return**(null); // 查找失败

else return(p);

```
(3) while(p!=null && p->data>X) p=p->next;
    if(p==null || p->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 $q:=p.^{llink}$; 则

```
q.^{rlink}:=p.^{rlink}; p.^{rlink}.^{llink}:=q; p.^{llink}:=q.^{llink};
q.^{llink}.^{rlink}:=p; p.^{rlink}:=q; q.^{llink}:=p
```

17. (1) 前两个语句改为:

```
p.^{llink}.^{rlink} <- p.^{rlink};
p.^{rlink}.^{llink} <- p.^{llink};
```

(2) 后三个语句序列应改为:

```
q.^{rlink} <- p.^{rlink}; // 以下三句的顺序不能变
p.^{rlink}.^{llink} <- q;
p.^{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) $f1 \neq \text{NIL}$ 并且 $f2 \neq \text{NIL}$

(B) $f1 \uparrow . \text{data} < f2 \uparrow . \text{data}$

(C) $f2 \uparrow . \text{data} < f1 \uparrow . \text{data}$

(D) $f3 \uparrow . \text{data} < f1 \uparrow . \text{data}$

(E) $f1 \leftarrow f1 \uparrow . \text{link}$ 或 $f2 \leftarrow f2 \uparrow . \text{link}$;

21. 1) 本算法功能是将双向循环链表结点的数据域按值自小到大排序, 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) $r \rightarrow \text{prior} = q \rightarrow \text{prior}$; // 将 q 结点摘下, 以便插入到适当位置。

(2) $p \rightarrow \text{next} \rightarrow \text{prior} = q$; // (2) (3) 将 q 结点插入

(3) $p \rightarrow \text{next} = q$;

(4) $r = r \rightarrow \text{next}$; 或 $r = q \rightarrow \text{next}$; // 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
```

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967


```

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 表的公共元素就退出 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
}
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** ($i=low$; $i \leq num$; $i++$) $A[i+1]=A[i]$ ，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 $O(\log n)$ ，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 是结果集合中元素个数, 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 $p != \text{null}$ ），结束循环也未用指针判断。注意最后一句 $q->\text{next} = \text{pre}->\text{next}$ ，实现了从原第 i 个结点到原第 m 个结点的循环。最后 $\text{pre}->\text{next}$ 正是指向原第 m 个结点，不可用 $p->\text{next}$ 代替 $\text{pre}->\text{next}$ 。

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。

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

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 $pre \rightarrow next = p$ 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 $pre \rightarrow next = p$ 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, $pre \rightarrow next = p$ 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: sqliisttp; 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: sqliisttp)

 // 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 **DOWNTO** 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->freq=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->freq);
    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 中查找有无该数据。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 3 章 栈和队列答案

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. √	2. √	3. √	4. √	5. ×	6. √	7. √	8. √	9. √	10. ×	11. √	12. ×
13. ×	14. ×	15. √	16. ×	17. √	18. ×	19. √	20. √				

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

- 1、操作受限（或限定仅在表尾进行插入和删除操作） 后进先出
- 2、栈 3、3 1 2 4、23 100CH 5、0 $n+1$ $top[1]+1=top[2]$
- 6、两栈顶指针值相减的绝对值为 1（或两栈顶指针相邻）。
- 7、(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）
- 8、链式存储结构 9、 $S \times SS \times S \times \times$ 10、 $data[++top]=x$;
- 11、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）
- 12、假溢出时大量移动数据元素。
- 13、 $(M+1) \bmod N$ $(M+1) \% N$; 14、队列 15、先进先出 16、先进先出
- 17、 $s=(LinkedList)malloc(sizeof(LNode));$ $s->data=x;s->next=r->next;$ $r->next=s;$ $r=s;$
- 18、牺牲一个存储单元 设标记
- 19、 $(TAIL+1) \bmod M=FRONT$ （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）
- 20、 $sq.front=(sq.front+1) \% (M+1);$ **return**($sq.data(sq.front)$); $(sq.rear+1) \% (M+1) == sq.front$;
- 21、栈 22、 $(rear-front+m) \% m$; 23、 $(R-P+N) \% N$;
- 24、(1) $a[i]$ 或 $a[1]$ (2) $a[i]$ (3) $pop(s)$ 或 $s[1]$;
- 25、(1) **PUSH** (OPTR, w) (2) **POP** (OPTR) (3) **PUSH** (OPND, operate (a, theta, b))

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

26、(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 AAADDAADADDD。

(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。

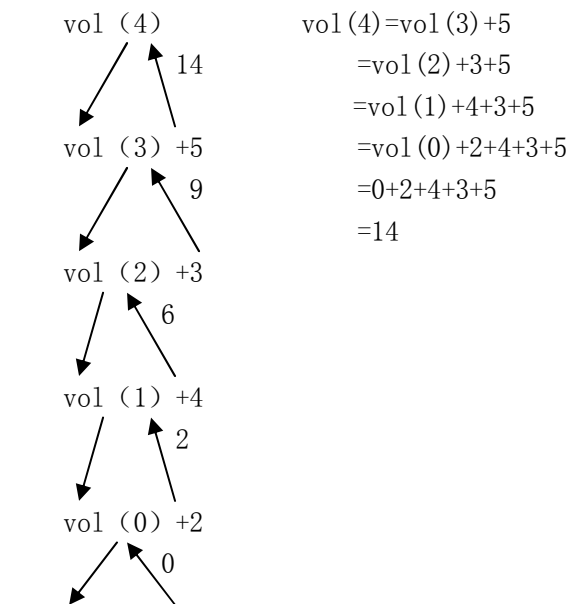
(2) 递归程序的优点是程序结构简单、清晰，易证明其正确性。缺点是执行中占内存空间较多，运行效率低。

(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

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

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

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

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

•

•

•

$$= 2^k H_{n-k} + 2^{k-1} + 2^{k-2} + \dots + 2^1 + 2^0$$

$$= 2^{n-1} H_1 + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0$$

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 $S[\text{MAX}]$ ，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 $\text{push}(i, x)$ 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。

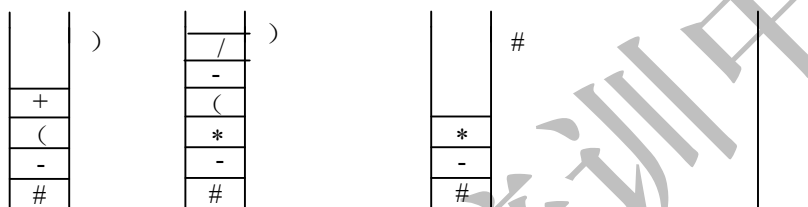
```
{if (ds.top[1]-ds.top[0]==1) {printf ( "栈满\n" ); return (0); }
switch (i)
{case 0: ds.s[++ds.top[i]]=x; break;
case 1: ds.s[--ds.top[i]]=x;
return (1); }//入栈成功。
}
```

18、本程序段查找栈 S 中有无整数为 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步骤：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 / - * -。

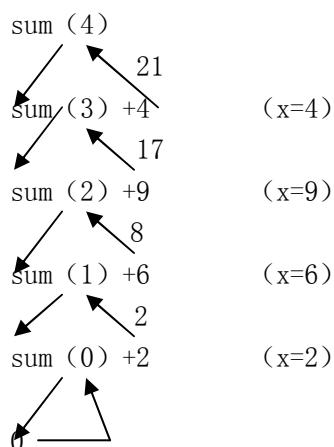
可用类似方法将中缀表达式转为前缀表达式。

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	# -	$A-B*C/D-E \uparrow F\#$	PUSH(OPTR, ' -')
3	AB	# -	$A-B*C/D-E \uparrow F\#$	PUSH(OPND, B)
4	AB	# - *	$A-B*C/D-E \uparrow F\#$	PUSH(OPTR, ' *')
5	ABC	# - *	$A-B*C/D-E \uparrow F\#$	PUSH(OPND, C)
6	AT ($T=B*C$)	# - /	$A-B*C/D-E \uparrow F\#$	PUSH(OPND, POP(OPND)*POP(OPND)) PUSH(OPTR, ' /')
7	ATD	# - /	$A-B*C/D-E \uparrow F\#$	PUSH(OPND, D)
8	AT ($T=T/D$) $T(T=A-T)$	# - # -	$A-B*C/D-E \uparrow F\#$	$x=\text{POP(OPND)}; y=\text{POP(OPND)}$ PUSH(OPND, y/x) ; $x=\text{POP(OPND)}; y=\text{POP(OPND)}$; PUSH(OPND, $y-x$) PUSH(OPTR, ' -')
9	TE	# -	$A-B*C/D-E \uparrow F\#$	PUSH(OPND, E)

10	TE	# - ↑	↑ 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、XSXXXSSSXXSXXSSSS

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

```
{element elemq[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 未空，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 $s.\text{front}=s.\text{rear}$; //设 s 是 sequeuetp 类型变量

(2) 队满: $(s.\text{rear}+1) \text{ MOD } \text{MAXSIZE}=s.\text{front}$ //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 $(\text{REAR}-\text{FRONT}+N) \% N$ 。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 $(\text{rear}+1) \% \text{表长}=\text{front}$ 为队满，或通过设标记 tag。若 tag=0, front==rear 则为队空；若 tag=1，因入队而使得 front==rear，则为队满。

本题中队列尾指针 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 $(\text{rear}+1) \% n=\text{front}$ 时为队满。出队操作（在队列不空情况下）队头指针是 $\text{front}=(\text{front}+1) \% n$,

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

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

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, $s1$ 栈顶指针为 -1 , $s2$ 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 $s1$, $i=1$ 表示右边的栈 $s2$, 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 $s1$ 栈, $i=1$ 时为 $s2$ 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, $s1$ 栈是通常意义下的栈, 而 $s2$ 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

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

```

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<=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!=-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]!= ‘#’ ) //逐字符处理字符串表达式的数组。
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!='$')
{
    switch
    {
        case '0' <=x<='9': while((x>='0' && x<='9') || x=='.' ) //拼数
            {
                if(x!='.') //处理整数
                {
                    num=num*10+(ord(x)-ord('0')); scanf("%c",&x);
                }
                else //处理小数部分。
                {
                    scale=10.0; scanf("%c",&x);
                    while(x>='0' && x<='9')
                    {
                        num=num+(ord(x)-ord('0'))/scale;
                        scale=scale*10; scanf("%c",&x);
                    }
                }
                push(OPND, num); num=0.0; //数压入栈，下个数字初始化
            }
        case x=' ': break; //遇空格，继续读下一个字符。
        case x='+': push(OPND, pop(OPND)+pop(OPND)); break;
        case x='-': x1=pop(OPND); x2=pop(OPND); push(OPND, x2-x1); break;
        case x='*': push(OPND, pop(OPND)*pop(OPND)); break;
        case x='/': x1=pop(OPND); x2=pop(OPND); push(OPND, x2/x1); break;
        default: //其它符号不作处理。
    }
} //结束 switch
scanf("%c",&x); //读入表达式中下一个字符。
} //结束 while (x != '$')
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(Sempty(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(!Sempty(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, ElementType 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
```

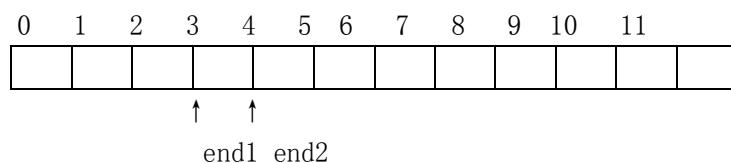
void enqueue (cycqueue Q, elemtp x)

// 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 $maxsize = 12$ ）



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 0F

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 0F

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
}
```

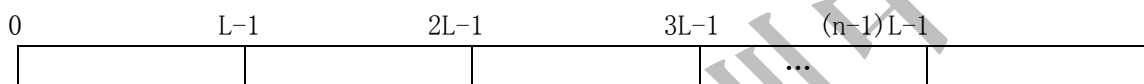
(3) 出队 **int deleteq (int i)**

// 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)**

// 设整数序列存于数组 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, 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));
```

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

}//算法结束

使用栈，消除递归的非递归算法如下：

```
int gcd(int m,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,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. √	2. √	3. √									
------	------	------	--	--	--	--	--	--	--	--	--

三、填空题

1. (1) 由空格字符 (ASCII 值 32) 所组成的字符串 (2) 空格个数 2. 字符
3. 任意个连续的字符组成的子序列 4. 5 5. $0(m+n)$
6. 01122312 7. 01010421 8. (1) 模式匹配 (2) 模式串
9. (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
10. 两串的长度相等且两串中对应位置的字符也相等。
11. 'xyxyxywwy' 12. $*s++=*t++$ 或 $(*s++=*t++) != '\0'$
13. (1) **char** s[] (2) j++ (3) i >= j
14. [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 ($1 \leq i \leq s.len$)。t 串用 j 指针 ($1 \leq j \leq t.len$)。算法思想是对每个 i ($1 \leq i \leq s.len$, 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 ($1 \leq j \leq t.len$, 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) ($i+k \leq s.len$) AND ($j+k \leq t.len$) 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) $i+k \leq s.len$ && $j+k \leq t.len$ && $s[i+k] = t[j+k]$ //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 S_2 是串 S_1 的子串, 且在 S_1 中的位置是 1。开始求出最大公共子串的长度恰是串 S_2 的长度, 一般情况下, $T(m, n) = O(m \cdot n)$ 。

4. 朴素的模式匹配 (Brute-Force) 时间复杂度是 $O(m \cdot n)$, 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. p_1 的 next 和 nextval 值分别为: 0112234 和 0102102; p_2 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

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

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)

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 算法的时间复杂性是 $O(m+n)$ 。

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, $next[1]=0$ 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 $1 \leq k < j$ 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 $next[j]$ 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 $next[j]$)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

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

replace(s1,i,j,s2): 置换函数,用 s2 串替换 s1 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) s1=substr(s,3,1) //取出字符: 'y'
- (2) s2=substr(s,6,1) //取出字符: '+'
- (3) s3=substr(s,1,5) //取出子串: '(xyz)'
- (4) s4=substr(s,7,1) //取出字符: '*'
- (5) s5=replace(s3,3,1,s2) //形成部分串: '(x+z)'
- (6) s=s5//s4//s1 //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 串的最后一个字符的下标。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] 教材中介绍的串替换有两种形式：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.cursor`）移动 `t.cursor-j` 绝对值个位置（以便将 `t` 串插入）：若 `j>t.cursor`，则向左移；若 `j<t.cursor`，则向右移动；若 `j=t.cursor`，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.cursor+s.cursor-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.cursor) //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.cursor-1;k>=i+j-1;k--) s.ch[k+t.cursor-j]=s.ch[k];
else if (j>t.cursor) //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.cursor-1;k++) s.ch[k-(j-t.cursor)]=s.ch[k];
    for(k=0;k<t.cursor;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.cursor) s.cursor=s.cursor-(j-t.cursor);else s.cursor=s.cursor+(t.cursor-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.cursor;k++) s1.ch[i+k]=t.ch[k] //将 t 串接到 s1 的尾部
l=s.cursor+t.cursor-j-1;
for(k=s.cursor-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.cursor+t.cursor-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 s 串中再找串 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，在串长相等的情况下，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' bcb' 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;

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=TOKEN.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. √										

部分答案解析如下。

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

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

15. 非零元很少 ($t \ll m \times n$) 且分布没有规律 16. 节省存储空间。
 17. 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: $k = ((i-1) \times (2n-i+2))/2 + (j-i+1) = (i-1)(2n-i)/2 + j$ ($i \leq j$)
 18. 93 19. $i(i-1)/2+j$ 20. 线性表 21. 其余元素组成的表
 22. (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

23. 深度 24. (1) () (2) (()) (3) 2 (4) 2
 25. head(head(tail(tail(head(tail(tail(A)))))))
 26. 表展开后所含括号的层数 27. (1) 5 (2) 3
 28. head(head(tail(LS))) 29. head(tail(tail(head(tail(head(A))))))
 30. head(tail(head(tail(H)))) 31. (b) 32. (x, y, z) 33. (d, e)
 34. GetHead(GetHead(GetTail(L)))
 35. 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) $j \leq m$
 36. (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 $k-1$)。内层 do 循环以 t ($t=a[low]$) 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 $\text{if}(i==k)$ **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

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

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) $(p \rightarrow \text{tag} == 0)$ // 处理原子
 (2) $h = \text{reverse}(p \rightarrow \text{val.ptr.hp})$ // 处理表头
 (3) $(p \rightarrow \text{val.ptr.tp})$ // 产生表尾的逆置广义表
 (4) $s \rightarrow \text{val.ptr.tp} = t$; // 连接
 (5) $q \rightarrow \text{val.ptr.hp} = h$ // 头结点指向广义表
 38. 本题要求将 $1, 2, \dots, n \times n$ 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) $k \geq n$ // 修改副对角线下方的下标 i 和 j 。
 (5) $m++$; 或 $m = m + 1$ // 为填下个数字作准备, 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。

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

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) $(s+m-1) \text{ MOD } i$ //计算出圈人 s_1
 - (2) $s_1:=i$ //若 $s_1=0$,说明是第 i 个人出圈 ($i\%i=0$)
 - (3) $s_1 \text{ TO } i-1$ //从 s_1 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:

$$\text{LOC}(A_{ijk}) = \text{LOC}(A_{c_1c_2c_3}) + [(i-c_1)V_2V_3 + (j-c_2)V_3 + (k-c_3)] * L + 1$$

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 $\lfloor b/2 \rfloor$ 条对角线(为叙述方便,下面设 $a = \lfloor b/2 \rfloor$)。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 $(n-2a)b + a*(a+b)$ 。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: $\text{Loc}(A_{ijkl}) = 100(\text{基地址}) + [(i-c_1)V_2V_3V_4 + (j-c_2)V_3V_4 + (k-c_3)V_4 + (l-c_4)] * 4$)

5. $1210+108L$ ($\text{LOC}(A[1, 3, -2]) = 1210 + [(k-c_3)V_2V_1 + (j-c_2)V_1 + (i-c_1)] * L$ (设每个元素占 L 个存储单元))

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

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 m_u 、列数 n_u 和非零元素个数 t_u (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 $m*n$ 个单元,只有当 $3(t+1) < m*n$ 时,用 LTMA 表示 A 才有意义。解不等式得 $t < m*n/3 - 1$ 。

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) $10^3 \times 10^3 - (3 \times 10^3 - 2)$

15. 稀疏矩阵 A 采用二维数组存储时，需要 $n \times n$ 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 ($t \ll m \times n$)，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. $LOC(A[i, j]) = LOC(A[3, 2]) + [(i-3) \times 5 + (j-2)] \times 2$ (按行存放)

$LOC(A[i, j]) = LOC(A[3, 2]) + [(j-2) \times 6 + (i-3)] \times 2$ (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 $i-j+1$ 。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) $i=k/3+1; (1 \leq k \leq 3n-2) \quad // \quad k/3$ 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： $O(4N \cdot \log_2 2N) \approx O(N \log(2 \cdot N))$ ；最差情况： $O((4N+3N) \cdot \log_2 2N) \approx O(N \log(2 \cdot N))$ 。

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：

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

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

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

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

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<=i<=n)中存放 A 数组中值为 i 的元素个数。

(b) c[i] (1<=i<=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：

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

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

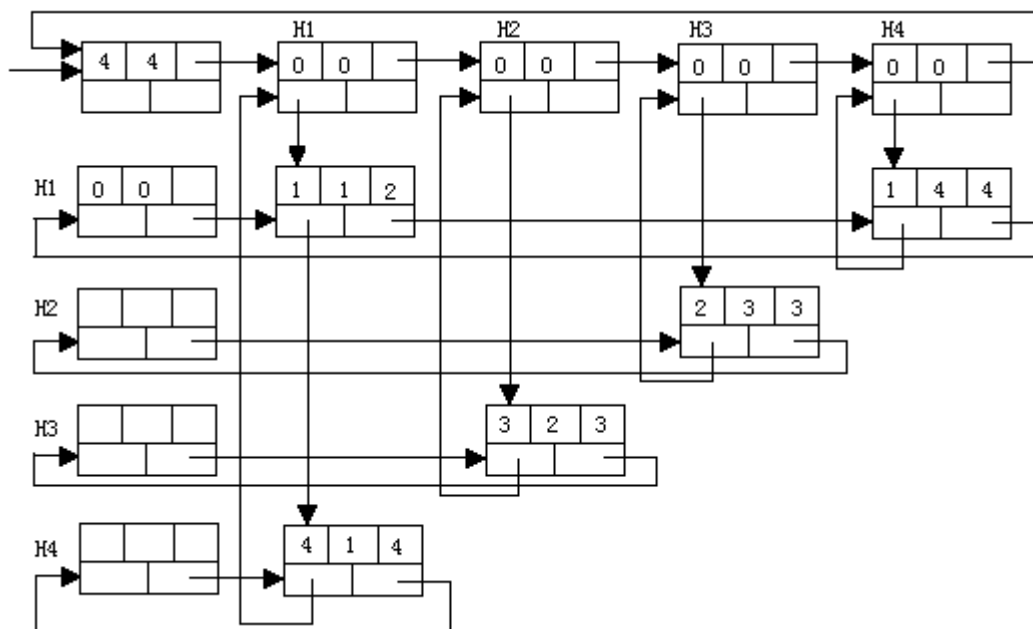
报名热线：025-83535877、18951896587、18951896993、18951896967

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) $k = 2(i-1) + j$ ($1 \leq i, j \leq n, |i-j| \leq 1$)

$i = \text{floor}(k/3) + 1$ // $\text{floor}(a)$ 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 $i=n+1-j$ ($1 \leq i, j \leq n$)

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

(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 $k=2(i-1)$ ($i > j, n/2+1 \leq i, j \leq n, 1 \leq k \leq 2n-1$)。

$$(3) a_{ij} \text{ 在 B 中的位置 } K = \begin{cases} A0 + 2(i-1) & (i=j, 1 \leq i, j \leq n) \\ A0 + 2(i-1) + 1 & (i < j, 1 \leq i, j \leq n/2) \\ A0 + 2(i-1) - 1 & (i > j, n/2+1 \leq i, j \leq n) \end{cases}$$

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:

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

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 mu, nu 和 tu 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 nu, mu 和 tu。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 nu 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 $O(n^3P)$, 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, 第一个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))))))

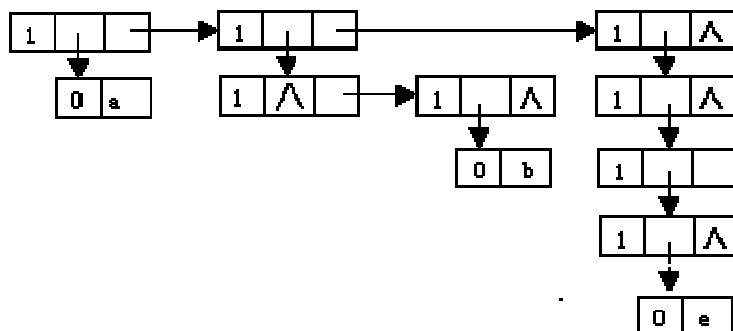
报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

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))

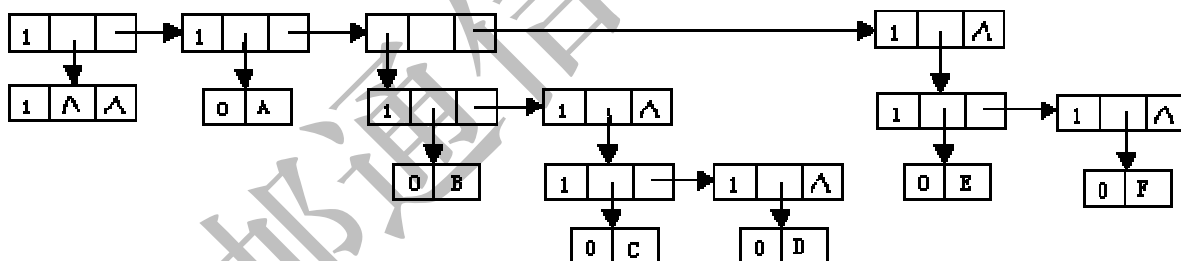
head (head (tail (L))) = (c,d)

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

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

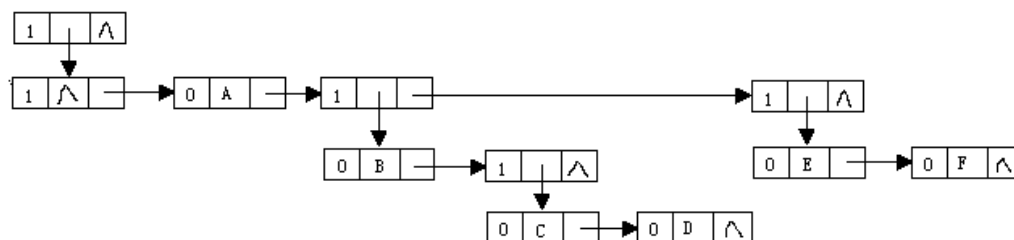
(6) head (tail (head (head (tail (tail (A)))))

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。

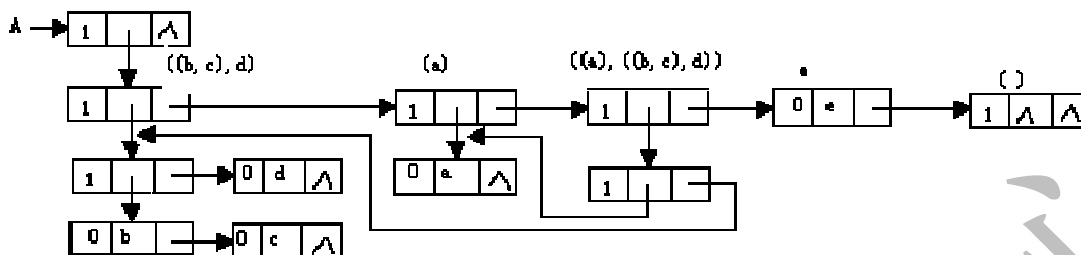
44. 深度为 5，长度为 2

45. (1) 略

(2) 表的长度为 5，深度为 4

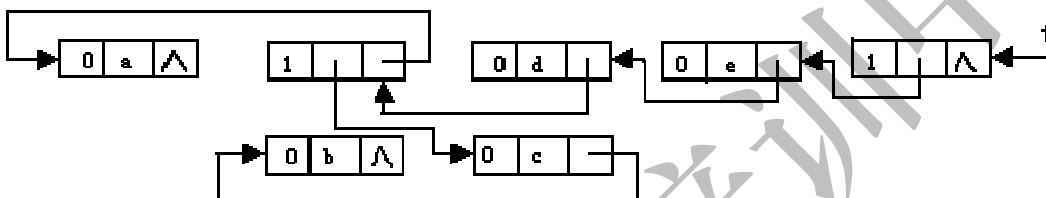
(3) head (tail (head (head (head (tail (tail (tail (tail (A))))))))))

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。

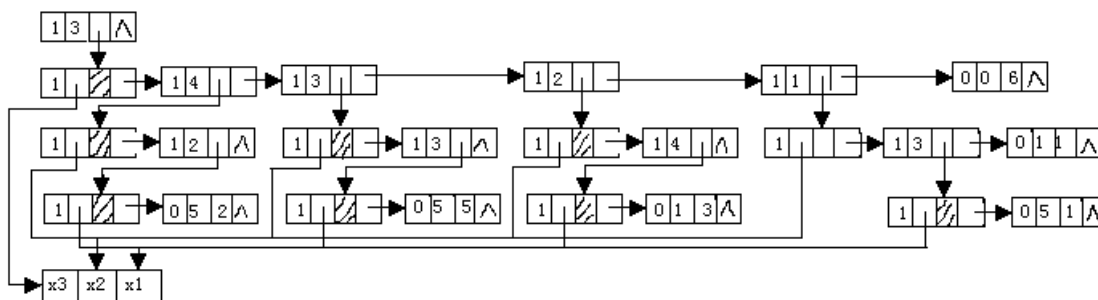


48. (a, b)

49. (d)

50. 否。广义表的长度不是广义表中原子个数，而是指广义表中所含元素的个数，广义表中的元素可以是原子，也可以是子表。广义表元素多于 1 个时，元素间用逗号分开。

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



52. (1) H(A(a₁, a₂), B(b₁), C(c₁, c₂), x)

HEAD(TAIL(HEAD(H))) = a₂

(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 ≤ i ≤ 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。
```

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967


```

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: elemtp;

END;

sparmattp=**RECORD**

 mu, nu, tu: integer;

 data: ARRAY[1..maxnum] OF tuple;

END;

PROC AddMatrix (**VAR** A: sparmattp; B: sparmattp);

// 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 个数的所有组合。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 DOWNT0 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;

```

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 l 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维循环的查找。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 减小到小于 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>-1)          //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>-1 && 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 JudgEqual(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 JudgEqual 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 $m \times n$ 个元素，第 1 个元素同其它 $m \times n - 1$ 个元素比较，第 2 个元素同其它 $m \times n - 2$ 个元素比较，……，第 $m \times n - 1$ 个元素同最后一个元素 ($m \times n$) 比较一次，所以在元素互不相等时总的比较次数为 $(m \times n - 1) + (m \times n - 2) + \dots + 2 + 1 = (m \times n)(m \times n - 1) / 2$ 。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 $(m \times n - 1)$ 个位置上均可能相同，这时的平均比较次数约为 $(m \times n)(m \times n - 1) / 4$ ，总的时间复杂度是 $O(n^4)$ 。

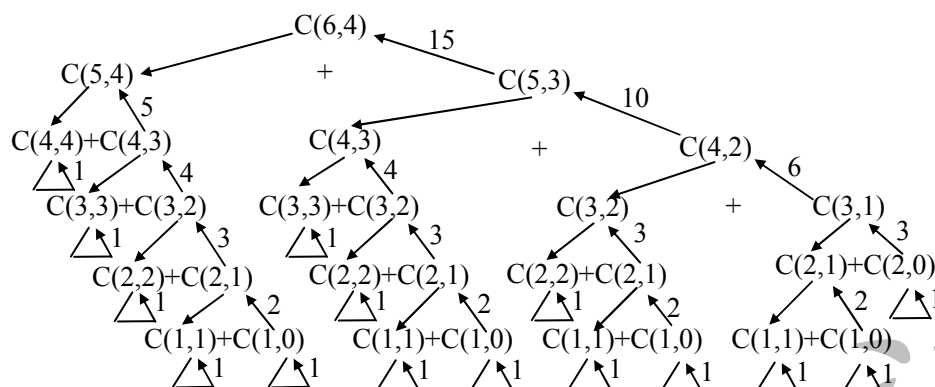
11. 二项式 $(a+b)^n$ 展开式的系数的递归定义为：

$$C(n, k) = \begin{cases} 1 & \text{当 } k=0 \text{ 或 } k=n (n \geq 0) \\ C(n-1, k) + C(n-1, k-1) & \text{当 } (0 < k < n) \end{cases} \quad C(n, k) = C_n^k = \frac{n(n-1)\dots(n-k+1)}{1*2*\dots*(k-1)*k}$$

```

(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 $A_{m \times n}$, H_m 是总表头指针。设 rch 是行列表头指针，则 $rch \rightarrow right = rch$ 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 ($k \ll m$) 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)

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967


```
//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, i, 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, i, 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]=j; // 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，只需要注意 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(Elentype 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
```

第 6 章 树和二叉树

一、选择题

1. D	2. B	3. C	4. D	5. D	6. A	7. 1C	7. 2A	7. 3C	7. 4A	7. 5C	8. B
9. C	10. D	11. B	12. E	13. D	14. D	15. C	16. B	17. C	18. C	19. B	20. D
21. A	22. A	23. C	24. C	25. C	26. C	27. C	28. C	29. B	30. C	31. D	32. B
33. A	34. D	35. B	36. B	37. C	38. B	39. B	40. B	41. 1F	41. 2B	42. C	43. B
44. C	45. C	46. B	47. D	48. B	49. C	50. A	51. C	52. C	53. C	54. D	55. C
56. B	57. A	58. D	59. D	60. B	61. 1B	61. 2A	61. 3G	62. B	63. B	64. D	65. D
66. 1C	66. 2D	66. 3F	66. 4H	66. 5I							

部分答案解析如下。

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层
 报名热线：025-83535877、18951896587、18951896993、18951896967

12. 由二叉树结点的公式: $n=n_0+n_1+n_2=n_0+n_1+(n_0-1)=2n_0+n_1-1$, 因为 $n=1001$, 所以 $1002=2n_0+n_1$, 在完全二叉树中, n_1 只能取 0 或 1, 在本题中只能取 0, 故 $n=501$, 因此选 E。
42. 前序序列是“根左右”, 后序序列是“左右根”, 若要这两个序列相反, 只有单支树, 所以本题的 A 和 B 均对, 单支树的特点是只有一个叶子结点, 故 C 是最合适的, 选 C。A 或 B 都不全。由本题可解答 44 题。
47. 左子树为空的二叉树的根结点的左线索为空(无前驱), 先序序列的最后结点的右线索为空(无后继), 共 2 个空链域。
52. 线索二叉树是利用二叉树的空链域加上线索, n 个结点的二叉树有 $n+1$ 个空链域。

二、判断题

1. ×	2. ×	3. ×	4. √	5. √	6. √	7. √	8. ×	9. √	10. ×	11. ×	12. ×
13. ×	14. √	15. ×	16. ×	17. √	18. √	19. ×	20. √	21. ×	22. √	23. ×	24. ×
25. √	26. ×	27. ×	28. ×	29. √	30. ×	31. ×	32. √	33. ×	34. ×	35. ×	36. √
37. √	38. ×	39. ×	40. ×	41. (3)	42. √	43. √	44. ×	45. √	46. ×	47. ×	48. ×
49. √	50. √										

部分答案解析如下。

6. 只有在确定何序(前序、中序、后序或层次)遍历后, 遍历结果才唯一。
19. 任何结点至多只有左子树的二叉树的遍历就不需要栈。
24. 只对完全二叉树适用, 编号为 i 的结点的左儿子的编号为 $2i$ ($2i \leq n$), 右儿子是 $2i+1$ ($2i+1 \leq n$)
37. 其中序前驱是其左子树上按中序遍历的最右边的结点(叶子或无右子女), 该结点无右孩子。
38. 新插入的结点都是叶子结点。
42. 在二叉树上, 对有左右子女的结点, 其中序前驱是其左子树上按中序遍历的最右边的结点(该结点的后继指针指向祖先), 中序后继是其右子树上按中序遍历的最左边的结点(该结点的前驱指针指向祖先)。
44. 非空二叉树中序遍历第一个结点无前驱, 最后一个结点无后继, 这两个结点的前驱线索和后继线索为空指针。

三. 填空题

1. (1)根结点 (2)左子树 (3)右子树 2. (1)双亲链表表示法 (2)孩子链表表示法 (3)孩子兄弟表示法
3. $p \rightarrow lchild == \text{null} \ \&\& \ p \rightarrow rchild == \text{null}$ 4. (1) $++a * b * 3 * 4 - cd$ (2) 18 5. 平衡因子
6. 9 7. 12 8. (1) 2^{k-1} (2) $2^k - 1$ 9. (1) 2^{H-1} (2) $2^H - 1$ (3) $H = \lfloor \log_2 N \rfloor + 1$
10. 用顺序存储二叉树时, 要按完全二叉树的形式存储, 非完全二叉树存储时, 要加“虚结点”。设编号为 i 和 j 的结点在顺序存储中的下标为 s 和 t , 则结点 i 和 j 在同一层上的条件是 $\lfloor \log_2 s \rfloor = \lfloor \log_2 t \rfloor$ 。
11. $\lfloor \log_2 i \rfloor = \lfloor \log_2 j \rfloor$ 12. (1) 0 (2) $(n-1)/2$ (3) $(n+1)/2$ (4) $\lfloor \log_2 n \rfloor + 1$ 13. n
14. $N+1$ 15. (1) $2^{k+1} - 1$ (2) $k+1$ 16. $\lfloor N/2 \rfloor$ 17. 2^{k-2} 18. 64
19. 99 20. 11 21. (1) $n_1 - 1$ (2) $n_2 + n_3$
22. (1) $2^{k-2} + 1$ (第 k 层 1 个结点, 总结点个数是 2^{H-1} , 其双亲是 $2^{H-1}/2 = 2^{k-2}$) (2) $\lfloor \log_2 i \rfloor + 1$ 23. 69
24. 4 25. 3^{h-1} 26. $\lfloor n/2 \rfloor$ 27. $\lceil \log_2 k \rceil + 1$
28. (1)完全二叉树 (2)单枝树, 树中任一结点(除最后一个结点是叶子外), 只有左子女或只有右子女。
29. $N+1$ 30. (1) 128(第七层满, 加第八层 1 个) (2) 7
31. 0 至多个。任意二叉树, 度为 1 的结点数没限制。只有完全二叉树, 度为 1 的结点数才至多为 1。
32. 21 33. (1) 2 (2) $n-1$ (3) 1 (4) n (5) 1 (6) $n-1$
34. (1) FEGHDCB (2) BEF (该二叉树转换成森林, 含三棵树, 其第一棵树的先根次序是 BEF)
35. (1)先序 (2)中序 36. (1)EACBDGF (2) 2 37. 任何结点至多只有右子女的二叉树。
38. (1)a (2) dbe (3) hfeg 39. (1) .D.G.B.A.E.H.C.F. (2) ...GD.B...HE..FCA
40. DGEBFCA 41. (1) 5 (2) 略 42. 二叉排序树 43. 二叉树 44. 前序
45. (1)先根次序 (2)中根次序 46. 双亲的右子树中最左下的叶子结点 47. 2 48. $(n+1)/2$

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

49. 31 (x 的后继是经 x 的双亲 y 的右子树中最左下的叶结点) 50. (1) 前驱 (2) 后继
51. (1) 1 (2) y[^].lchild (3) 0 (4) x (5) 1 (6) y (7) x (编者注: 本题按中序线索化)
52. 带权路径长度最小的二叉树, 又称最优二叉树 53. 69 54. (1) 6 (2) 261
55. (1) 80 (2) 001 (不唯一) 56. $2n_0-1$
57. 本题①是表达式求值, ②是在二叉排序树中删除值为 x 的结点。首先查找 x, 若没有 x, 则结束。否则分成四种情况讨论: x 结点有左右子树; 只有左子树; 只有右子树和本身是叶子。
- (1) Postorder_eval(t[^].lchild) (2) Postorder_eval(t[^].rchild) (3) ERROR (无此运算符) (4) A
- (5) tempA[^].lchild (6) tempA=NULL (7) q[^].rchild (8) q (9) tempA[^].rchild (10) tempA[^].Item<r[^].Item
58. (1) IF t=NIL THEN num:=0 ELSE num:=num(t[^].l)+num(t[^].r)+1
- (2) IF (t=NIL) AND (m≤n) OR (t<>NIL) AND (m>n) THEN all:=false
- ELSE BEGIN chk(t[^].l, 2*m);chk (t[^].r, 2*m+1);END
59. (1) p->rchild (2) p->lchild (3) p->lchild (4) ADDQ(Q, p->lchild) (5) ADDQ(Q, p->rchild)
60. (1) t->rchild!=null (2) t->rchild!=null (3) NO++ (4) count(t->lchild) (5) count(t->rchild)
61. (1) p (2) 0 (3) height(p->lchild) (4) 0 (5) height(p->rchild) (6) lh+1 (7) rh+1 (8) 0
62. (1) p<>NIL (2) addx(p) (3) addx(tree) (4) r[^].rchild
63. (1) stack[tp]=t (2) p=stack[tp--] (3) p (4) ++tp
64. ① 本算法将二叉树的左右子树交换
- ② (1) new(s) //初始化, 申请结点 (2) s[^].next=NIL //s 是带头结点的链栈
- (3) s[^].next[^].data //取栈顶元素 (4) s[^].next:= p[^].next //栈顶指针下移
- (5) dispose(p) //回收空间 (6) p[^].next:=s[^].next //将新结点入链栈
- (7) push(s, p[^].rchild) //先沿树的左分支向下, 将 p 的右子女入栈保存
- (8) NOT empty(s) (9) finishe:=true //已完成 (10) finish=true (或 s[^].next=NIL)
65. (1) new(t) (2) 2*i≤n (3) t[^].lchild, 2*i (4) 2*i+1≤n (5) t[^].rchild, 2*i+1 (6) 1
66. (1) Push(s, p) (2) K=2 (3) p->data=ch (4) BT=p (5) ins>>ch
67. (1) result; (2) p:=p[^].link; (3) q:=q[^].pre ((2)(3) 顺序可变)
68. (1) top++ (2) stack[top]=p->rchild (3) top++ (4) stack[top]=p->lchild
69. (1) (i<=j) AND (x<=y) (2) A[i]<>B[k] (3) k-x
- (4) creatBT(i+1, i+L, x, k-1, s[^].lchild) (5) creatBT(i+L+1, j, k+1, y, s[^].rchild)
70. (1) push(s, bt) (2) pop(s) (3) push(s, p[^].rchild) // p 的右子树进栈
71. (1) p=p->lchild // 沿左子树向下 (2) p=p->rchild
72. (1) 0 (2) hl>hr (3) hr=hl
73. (1) top>0 (2) t*2 // 沿左分枝向下 (3) top-1 // 退栈
74. (1) p:=p[^].lchild (2) (3) p:=S.data[s.top][^].rchild (4) s.top=0
75. (1) *ppos // 根结点 (2) rpos=ipos (3) rpos - ipos (4) ipos (5) ppos+1
76. (1) top>0 (2) stack[top]:=nd[^].right (3) nd[^].left<>NIL (4) top:=top+1 (左子树非空)
77. (1) p<>thr // 未循环结束 (2) p->ltag=0 (3) p->lchild
- (4) p->rtag=1 && p->rchild!=thr (5) p=p->rchild (6) p=p->rchild
78. 若 p[^].rtag=1, 则 p[^].rchild 为后继, 否则 p 的后继是 p 的右子树中最左下的结点
- (1) q=p[^].rchild (2) q[^].ltag=0 (3) q[^].lchild
79. (1) tree->lchild (2) null (3) pre->rchild
- (4) pre->rtag=1 (5) pre->right=tree; (6) tree->right (注(4)和(5)顺序可换)
80. (1) node->rflag=0 (2) *x=bt (3) *x=node->right

四. 应用题

1. 树的孩子兄弟链表表示法和二叉树二叉链表表示法, 本质是一样的, 只是解释不同, 也就是说树(树

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

是森林的特例，即森林中只有一棵树的特殊情况）可用二叉树唯一表示，并可使用二叉树的一些算法去解决树和森林中的问题。

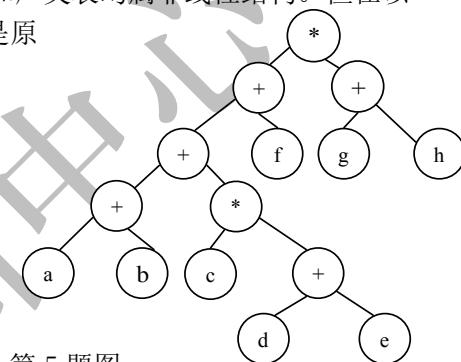
树和二叉树的区别有三：一是二叉树的度至多为 2，树无此限制；二是二叉树有左右子树之分，即使在只有一个分枝的情况下，也必须指出是左子树还是右子树，树无此限制；三是二叉树允许为空，树一般不允许为空（个别书上允许为空）。

2. 树和二叉树逻辑上都是树形结构，区别有以上题 1 所述三点。二叉树不是树的特例。

3. 线性表属于约束最强的线性结构，在非空线性表中，只有一个“第一个”元素，也只有一个“最后一个”元素；除第一个元素外，每个元素有唯一前驱；除最后一个元素外，每个元素有唯一后继。树是一种层次结构，有且只有一个根结点，每个结点可以有多个子女，但只有一个双亲（根无双亲），从这个意义上说存在一（双亲）对多（子女）的关系。广义表中的元素既可以是原子，也可以是子表，子表可以为它表共享。从表中套表意义上说，广义表也是层次结构。从逻辑上讲，树和广义表均属非线性结构。但在以下意义上，又蜕变为线性结构。如度为 1 的树，以及广义表中的元素都是原子时。另外，广义表从元素之间的关系可看成前驱和后继，也符合线性表，但这时元素有原子，也有子表，即元素并不属于同一数据对象。

4. 方法有二。一是对该算术表达式（二叉树）进行后序遍历，得到表达式的后序遍历序列，再按后缀表达式求值；二是递归求出左子树表达式的值，再递归求出右子树表达式的值，最后按根结点运算符（+、-、*、/ 等）进行最后求值。

5. 该算术表达式转化的二叉树如右图所示。



第 5 题图

6. n ($n > 0$) 个结点的 d 度树共有 nd 个链域，除根结点外，每个结点均有一个指针所指，故该树的空链域有 $nd - (n-1) = n(d-1) + 1$ 个。

7. 证明：设二叉树度为 0 和 2 的结点数及总的结点数分别为 n_0 , n_2 和 n ，则 $n = n_0 + n_2 \dots (1)$

再设二叉树的分支数为 B ，除根结点外，每个结点都有一个分支所指，则 $n = B + 1 \dots (2)$
度为零的结点是叶子，没有分支，而度为 2 的结点有两个分支，因此 (2) 式可写为

$$n = 2n_2 + 1 \dots (3)$$

由 (1)、(3) 得 $n_2 = n_0 - 1$ ，代入 (1)，并由 (1) 和 (2) 得 $B = 2(n_0 - 1)$ 。证毕。

8. (1) k^{h-1} (h 为层数)

(2) 因为该树每层上均有 k^{h-1} 个结点，从根开始编号为 1，则结点 i 的从右向左数第 2 个孩子的结点编号为 ki 。设 n 为结点 i 的子女，则关系式 $(i-1)k+2 \leq n \leq ik+1$ 成立，因 i 是整数，故结点 n 的双亲 i 的编号为 $\lfloor (n-2)/k \rfloor + 1$ 。

(3) 结点 n ($n > 1$) 的前一结点编号为 $n-1$ （其最右边子女编号是 $(n-1)*k+1$ ），故结点 n 的第 i 个孩子的编号是 $(n-1)*k+1+i$ 。

(4) 根据以上分析，结点 n 有右兄弟的条件是，它不是双亲的从右数的第一子女，即 $(n-1) \% k \neq 0$ ，其右兄弟编号是 $n+1$ 。

9. 最低高度二叉树的特点是，除最下层结点个数不满外，其余各层的结点数都应达到各层的最大值。设 n 个结点的二叉树的最低高度是 h ，则 n 应满足 $2^{h-1} < n \leq 2^h - 1$ 关系式。解此不等式，并考虑 h 是整数，则有 $h = \lfloor \log_2 n \rfloor + 1$ ，即任一结点数为 n 的二叉树的高度至少为 $O(\log n)$ 。

10. $2^n - 1$ (本题等价于高度为 n 的满二叉树有多少叶子结点，从根结点到各叶子结点的单枝树是不同的二叉树。)

11. 235。由于本题求二叉树的结点数最多是多少，第 7 层共有 $2^{7-1} = 64$ 个结点，已知有 10 个叶子，其余 54 个结点均为分支结点。它在第八层上有 108 个叶子结点。所以该二叉树的结点数最多可达 $(2^7 - 1 + 108) = 235$ 。(注意：本题并未明说完全二叉树的高度，但根据题意，只能 8 层。)

12. 1023 ($= 2^{10} - 1$)

13. 证明：设度为 1 和 2 的结点数是 n_1 和 n_2 ，则二叉树结点数 n 为 $n = m + n_1 + n_2 \dots (1)$

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

由于二叉树根结点没有分枝所指，度为 1 和 2 的结点各有 1 个和 2 个分枝，度为 0 的结点没有分枝，故二叉树的结点数 n 与分枝数 B 有如下关系

$$n=B+1=n_1+2n_2+1\cdots\cdots\cdots (2)$$

由 (1) 和 (2)，得 $n_2=m-1$ 。即 n 个结点的二叉树，若叶子结点数是 m ，则非叶子结点中有 $(m-1)$ 个度为 2，其余度为 1。

14. 根据顺序存储的完全二叉树的性质，编号为 i 的结点的双亲的编号是 $\lfloor i/2 \rfloor$ ，故 $A[i]$ 和 $A[j]$ 的最近公共祖先可如下求出：

```
while(i/2!=j/2)
    if(i>j) i=i/2; else j=j/2;
```

退出 **while** 后，若 $i/2=0$ ，则最近公共祖先为根结点，否则最近公共祖先是 $i/2$ 。

15. N 个结点的 K 叉树，最大高度 N （只有一个叶结点的任意 k 叉树）。设最小高度为 H ，第 i ($1 \leq i \leq H$) 层的结点数 K^{i-1} ，则 $N=1+k+k^2+\cdots+k^{H-1}$ ，由此得 $H=\lfloor \log_K(N(K-1)+1) \rfloor$

16. 结点个数在 20 到 40 的满二叉树且结点数是素数的数是 31，其叶子数是 16。

17. 设分枝结点和叶子结点数分别为 n_k 和 n_0 ，因此有 $n=n_0+n_k$ (1)

另外从树的分支数 B 与结点的关系有 $n=B+1=K*n_k+1$ (2)

由 (1) 和 (2) 有 $n_0=n-n_k=(n(K-1)+1)/K$

18. 用顺序存储结构存储 n 个结点的完全二叉树。编号为 i 的结点，其双亲编号是 $\lfloor i/2 \rfloor$ ($i=1$ 时无双亲)，其左子女是 $2i$ (若 $2i \leq n$ ，否则 i 无左子女)，右子女是 $2i+1$ (若 $2i+1 \leq n$ ，否则无右子女)。

19. 根据完全二叉树的性质，最后一个结点（编号为 n ）的双亲结点的编号是 $\lfloor n/2 \rfloor$ ，这是最后一个分枝结点，在它之后是第一个终端（叶子）结点，故序号最小的叶子结点的下标是 $\lfloor n/2 \rfloor + 1$ 。

20. 按前序遍历对顶点编号，即根结点从 1 开始，对前序遍历序列的结点从小到大编号。

21. 设树的结点数为 n ，分枝数为 B ，则下面二式成立

$$n=n_0+n_1+n_2+\cdots+n_m \quad (1)$$

$$n=B+1=n_1+2n_2+\cdots+mn_m \quad (2)$$

$$\sum_{i=1}^m (i-1)n_i$$

由 (1) 和 (2) 得叶子结点数 $n_0=1+\sum_{i=1}^m (i-1)n_i$

22. $\lceil \log_2 n \rceil + 1$ 23. 15

24. 该结论不成立。对于任一 $a \in A$ ，可在 B 中找到最近祖先 f 。 a 在 f 的左子树上。对于从 f 到根结点路径上所有 $b \in B$ ，有可能 f 在 b 的右子树上，因而 a 也就在 b 的右子树上，这时 $a > b$ ，因此 $a < b$ 不成立。同理可以证明 $b < c$ 不成立。而对于任何 $a \in A, c \in C$ 均有 $a < c$ 。

25. n 个结点的 m 次树，共有 $n*m$ 个指针。除根结点外，其余 $n-1$ 个结点均有指针所指，故空指针数为 $n*m-(n-1)=n*(m-1)+1$ 。证毕。

26. 证明 设度为 1 和 2 及叶子结点数分别为 n_0, n_1 和 n_2 ，则二叉树结点数 n 为 $n=n_0+n_1+n_2$ (1)

再看二叉树的分支数，除根结点外，其余结点都有一个分支进入，设 B 为分支总数，则 $n=B+1$ 。度为 1 和 2 的结点各有 1 个和 2 个分支，度为 0 的结点没有分支，故 $n=n_1+2n_2+1$ (2)

由 (1) 和 (2)，得 $n_0=n_2+1$ 。

27. 参见题 26。

28. 设完全二叉树中叶子结点数为 n ，则根据完全二叉树的性质，度为 2 的结点数是 $n-1$ ，而完全二叉树中，度为 1 的结点数至多为 1，所以具有 n 个叶子结点的完全二叉树结点数是 $n+(n-1)+1=2n$ 或 $2n-1$ （有或无度为 1 的结点）。由于具有 $2n$ (或 $2n-1$) 个结点的完全二叉树的深度是 $\lfloor \log_2(2n) \rfloor + 1$ ($\lfloor \log_2(2n-1) \rfloor + 1$)，即 $\lceil \log_2 n \rceil + 1$ ，故 n 个叶结点的非满的完全二叉树的高度是 $\lceil \log_2 n \rceil + 1$ 。（最下层结点数 ≥ 2 ）。

29. (1) 根据二叉树度为 2 结点个数等于叶子结点个数减 1 的性质，故具有 n 个叶子结点且非叶子结点均有左左子树的二叉树的结点数是 $2n-1$ 。

(2) 证明：当 $i=1$ 时， $2^{-(i-1)}=2^0=1$ ，公式成立。设当 $i=n-1$ 时公式成立，证明当 $i=n$ 时公式仍成立。

设某叶子结点的层号为 t ，当将该结点变为内部结点，从而再增加两个叶子结点时，这两个叶子结点的层号都是 $t+1$ ，对于公式的变化，是减少了一个原来的叶子结点，增加了两个新叶子结点，反映到公式中，因为 $2^{-(t-1)} = 2^{-(t+1-1)} + 2^{-(t+1-1)}$ ，所以结果不变，这就证明当 $i=n$ 时公式仍成立。证毕。

30. 结点数的最大值 2^h-1 （满二叉树），最小值 $2h-1$ （第一层根结点，其余每层均两个结点）。

31. (1) $k(u-1)+1+i$ (2) $\lfloor (v-2)/k \rfloor + 1$ （参见第 8 题推导）

32. 该二叉树是按前序遍历顺序编号，以根结点为编号 1，前序遍历的顺序是“根左右”。

33. (1) 设 $n=1$ ，则 $e=0+2*1=2$ （只有一个根结点时，有两个外部结点），公式成立。

设有 n 个结点时，公式成立，即

$$E_n = I_n + 2n \quad (1)$$

现在要证明，当有 $n+1$ 个结点时公式成立。

增加一个内部结点，设路径长度为 1，则

$$I_{n+1} = I_n + 1 \quad (2)$$

该内部结点，其实是从一个外部结点变来的，即这时相当于也增加了一个外部结点（原外部结点变成内部结点时，增加两个外部结点），则

$$E_{n+1} = E_n + 1 + 2 \quad (3)$$

由 (1) 和 (2)，则 (3) 推导为

$$\begin{aligned} E_{n+1} &= I_n + 2n + 1 + 2 = I_{n+1} - 1 + 2n + 1 + 2 \\ &= I_{n+1} + 2(n+1) \end{aligned}$$

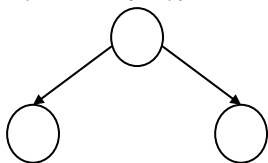
故命题成立

(2) 成功查找的平均比较次数 $s = I/n$

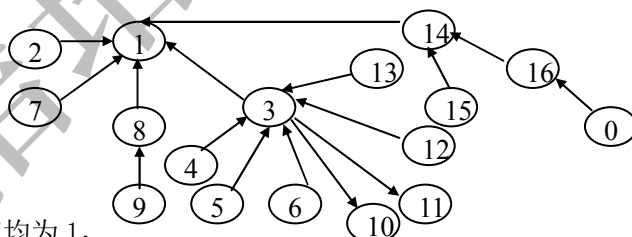
不成功查找的平均比较次数 $u = (E-n) / (n+1) = (I+n) / (n+1)$

由以上二式，有 $s = (1+1/n) * u - 1$ 。

34.



该有向图只有一个顶点入度为 0，其余顶点入度均为 1，它不是有向树。

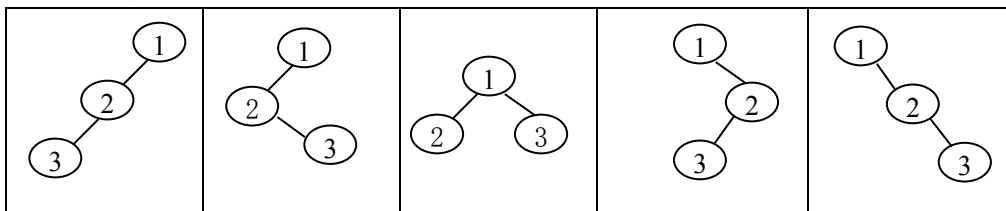


35. 题图

36. 参见 26 题

37. 由于二叉树前序遍历序列和中序遍历序列可唯一确定一棵二叉树，因此，若入栈序列为 1, 2, 3, ..., n，相当于前序遍历序列是 1, 2, 3, ..., n，出栈序列就是该前序遍历对应的二叉树的中序序列的数目。因为中序遍历的实质就是一个结点进栈和出栈的过程，二叉树的形态确定了结点进栈和出栈的顺序，也就确定了结点的中序序列。

下图以入栈序列 1, 2, 3，（解释为二叉树的前序序列）为例，说明不同形态的二叉树在中序遍历时栈的状态和访问结点次序的关系：



栈状态	访问	栈状态	访问	栈状态	访问	栈状态	访问	栈状态	访问
空		空		空		空		空	
1		1		1		1		1	
1 2		1 2		1 2		空	1	空	1
1 2 3		1	2	1	2	2		2	
1 2	3	1 3		空	1	2 3		空	2
1	2	1	3	3		2	3	3	
空	1	空	1	空	3	空	2	空	3

38. 给定二叉树结点的前序序列和对称序（中序）序列，可以唯一确定该二叉树。因为前序序列的第一个元素是根结点，该元素将二叉树中序序列分成两部分，左边（设 1 个元素）表示左子树，若左边无元素，则说明左子树为空；右边（设 r 个元素）是右子树，若为空，则右子树为空。根据前序遍历中“根—左子树—右子树”的顺序，则由从第二元素开始的 1 个结点序列和中序序列根左边的 1 个结点序列构造左子树，由前序序列最后 r 个元素序列与中序序列根右边的 r 个元素序列构造右子树。

由二叉树的前序序列和后序序列不能唯一确定一棵二叉树，因无法确定左右子树两部分。例如，任何结点只有左子树的二叉树和任何结点只有右子树的二叉树，其前序序列相同，后序序列相同，但却是两棵不同的二叉树。

39. 前序遍历是“根左右”，中序遍历是“左根右”，后序遍历是“左右根”。三种遍历中只是访问“根”结点的时机不同，对左右子树均是按左右顺序来遍历的，因此所有叶子都按相同的相对位置出现。

40. 在第 38 题，已经说明由二叉树的前序序列和中序序列可以确定一棵二叉树，现在来证明由二叉树的中序序列和后序序列，也可以唯一确定一棵二叉树。

当 $n=1$ 时，只有一个根结点，由中序序列和后序序列可以确定这棵二叉树。

设当 $n=m-1$ 时结论成立，现证明当 $n=m$ 时结论成立。

设中序序列为 S_1, S_2, \dots, S_m ，后序序列是 P_1, P_2, \dots, P_m 。因后序序列最后一个元素 P_m 是根，则在中序序列中可找到与 P_m 相等的结点（设二叉树中各结点互不相同） $S_i (1 \leq i \leq m)$ ，因中序序列是由中序遍历而得，所以 S_i 是根结点， S_1, S_2, \dots, S_{i-1} 是左子树的中序序列，而 $S_{i+1}, S_{i+2}, \dots, S_m$ 是右子树的中序序列。

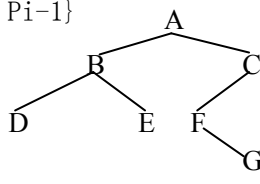
若 $i=1$ ，则 S_1 是根，这时二叉树的左子树为空，右子树的结点数是 $m-1$ ，则 $\{S_2, S_3, \dots, S_m\}$ 和 $\{P_1, P_2, \dots, P_{m-1}\}$ 可以唯一确定右子树，从而也确定了二叉树。

若 $i=m$ ，则 S_m 是根，这时二叉树的右子树为空，左子树的结点数是 $m-1$ ，则 $\{S_1, S_2, \dots, S_{m-1}\}$ 和 $\{P_1, P_2, \dots, P_{m-1}\}$ 唯一确定左子树，从而也确定了二叉树。

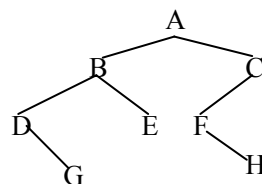
最后，当 $1 < i < m$ 时， S_i 把中序序列分成 $\{S_1, S_2, \dots, S_{i-1}\}$ 和 $\{S_{i+1}, S_{i+2}, \dots, S_m\}$ 。由于后序遍历是“左子树—右子树—根结点”，所以 $\{P_1, P_2, \dots, P_{i-1}\}$ 和 $\{P_i, P_{i+1}, \dots, P_{m-1}\}$ 是二叉树的左子树和右子树的后序遍历序列。因而由 $\{S_1, S_2, \dots, S_{i-1}\}$ 和 $\{P_1, P_2, \dots, P_{i-1}\}$

可唯一确定二叉树的左子树，由 $\{S_{i+1}, S_{i+2}, \dots, S_m\}$ 和 $\{P_i, P_{i+1}, \dots, P_{m-1}\}$ 可唯一确定二叉树的右子树。

由中序序列 DBEAFGC 和后序序列 DEBGFCA 构造的二叉树如右图：



第 40 题图



第 41 题图

41. 证明请参见第 40 题和第 38 题

由前序序列 ABDGECFH 和中序序列 DGBEAFHC 构造的二叉树如图：

42. 参见第 38 题

43. 先序遍历二叉树的顺序是“根—左子树—右子树”，中序遍历“左子树—根—右子树”，后序遍历顺序是：“左子树—右子树—根”，根据以上原则，本题解答如下：

- (1) 若先序序列与后序序列相同，则或为空树，或为只有根结点的二叉树
- (2) 若中序序列与后序序列相同，则或为空树，或为任一结点至多只有左子树的二叉树。
- (3) 若先序序列与中序序列相同，则或为空树，或为任一结点至多只有右子树的二叉树。

(4) 若中序序列与层次遍历序列相同, 则或为空树,

或为任一结点至多只有右子树的二叉树

由中序序列 DBEAFIHCG 和后序序列 DEBHIFGCA 确定的二叉树略。

44. 森林转为二叉树的三步:

(1) 连线 (将兄弟结点相连, 各树的根看作兄弟);

(2) 切线 (保留最左边子女为独生子女, 将其它子女分枝切掉);

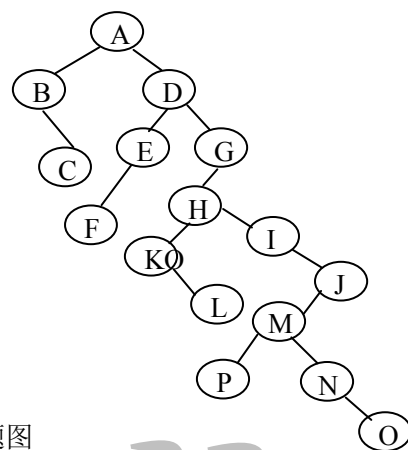
(3) 旋转 (以最左边树的根为轴, 顺时针向下旋转 45 度)。

其实经过 (1) 和 (2), 已转为二叉树,

执行 (3) 只是为了与平时的二叉树的画法一致。

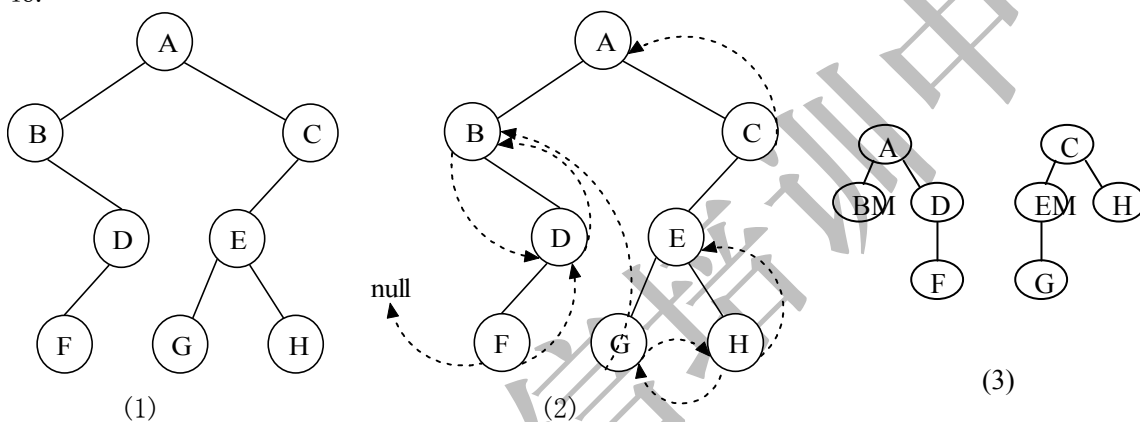
45. (1) ① $\text{tree}[p] \cdot l \rightarrow p$ ② $\text{tree}[p] \cdot r \rightarrow p$ ③ $p=0$

(2) 框 (A) 移至 III 处, 成为前序遍历。

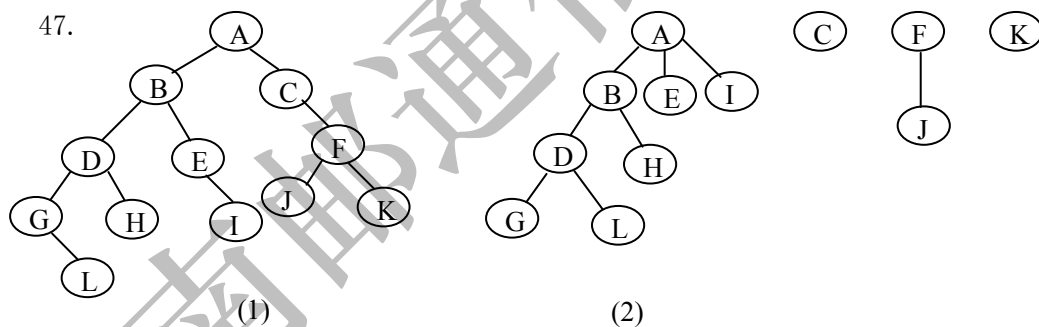


44 题图

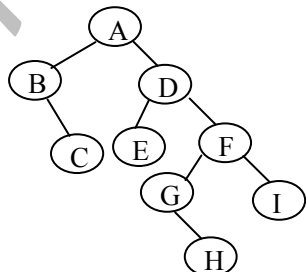
46.



47.



48. (1)



(2) 设二叉树的前序遍历序列为 P_1, P_2, \dots, P_m , 中序遍历序列为 S_1, S_2, \dots, S_m . 因为前序遍历是“根左右”, 中序遍历是“左根右”, 则前序遍历序列中第一个结点是根结点 (P_1). 到中序序列中查询到 $S_i = P_1$, 根据中序遍历根结点将中序序列分成两部分的原则, 有:

若 $i=1$, 即 $S_1 = P_1$, 则这时的二叉树没有左子树; 否则, S_1, S_2, \dots, S_{i-1} 是左子树的中序遍历序列, 用该

序列和前序序列 P_2, P_3, \dots, P_i 去构造该二叉树的左子树。

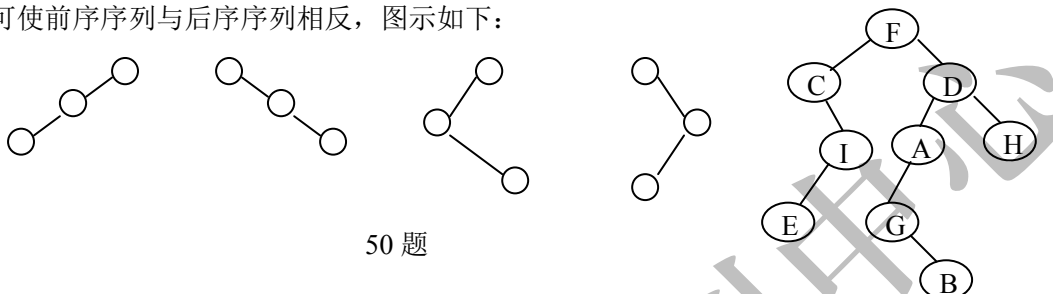
若 $i=m$, 即 $S_m=P_1$, 则这时的二叉树没有右子树; 否则, $S_{i+1}, S_{i+2}, \dots, S_m$ 是右子树的中序遍历序列, 用该序列和前序序列中 $P_{i+1}, P_{i+2}, \dots, P_m$ 去构造该二叉树的右子树。算法描述请参见下面算法设计第 56 题。

(3) 若前序序列是 $abcd$, 并非由这四个字母的任意组合 ($4!=24$) 都能构造出二叉树。因为以 $abcd$ 为输入序列, 通过栈只能得到 $1/(n+1)*2n!/(n!*n!)=14$ 种, 即以 $abcd$ 为前序序列的二叉树的数目是 14。任取以 $abcd$ 作为中序遍历序列, 并不全能与前序的 $abcd$ 序列构成二叉树。例如: 若取中序序列 $dcab$ 就不能。

该 14 棵二叉树的形态及中序序列略。

49. 不能。因 $DABC$ 并不是 $ABCD$ 的合法出栈序列, 参照第 37、48 (3) 的解释。

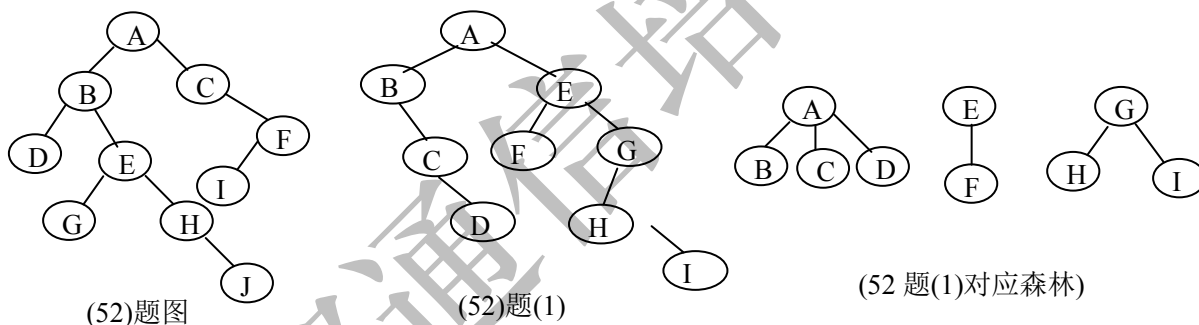
50. 先序序列是“根左右” 后序序列是“左右根”, 可见对任意结点, 若至多只有左子女或至多只有右子女, 均可使前序序列与后序序列相反, 图示如下:



50 题

51.

52. 按层次遍历, 第一个结点 (若树不空) 为根, 该结点在中序序列中把序列分成左右两部分—左子树和右子树。若左子树不空, 层次序列中第二个结点是左子树的根; 若左子树为空, 则层次序列中第二个结点是右子树的根。对右子树也作类似的分析。层次序列的特点是: 从左到右每个结点或是当前情况下子树的根或是叶子。

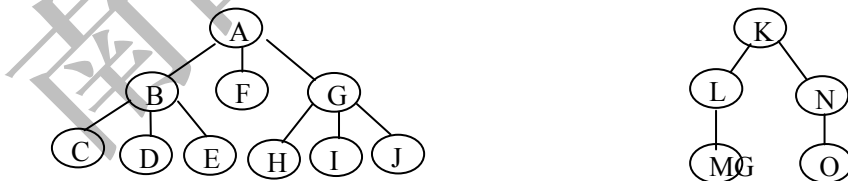


(52)题图

(52)题(1)

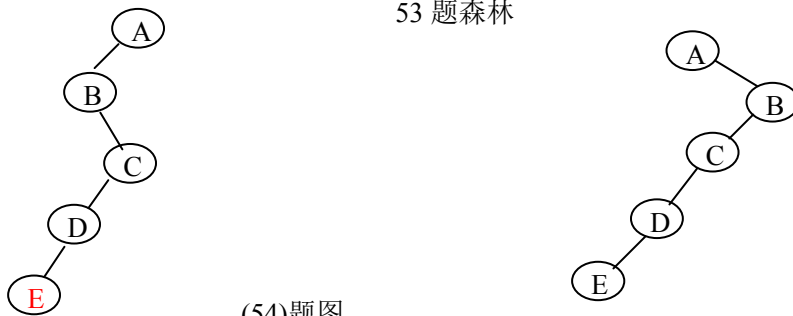
(52 题(1)对应森林)

53. 森林的先序序列和后序序列对应其转换的二叉树的先序序列和中序序列, 应先据此构造二叉树, 再构造出森林。



54.

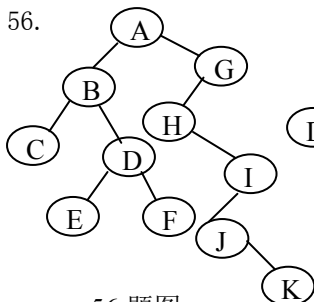
53 题森林



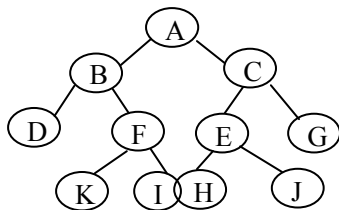
(54)题图

55. HIDJKEBLFGCA

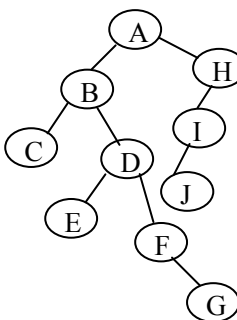
56.



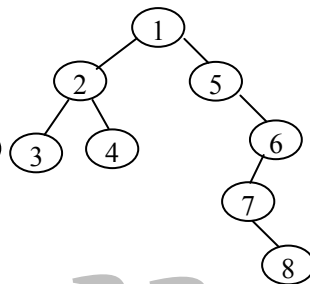
56 题图



56 (1)



56 (2)



56 (3)

57. M 叉树的前序和后序遍历分别与它转换成的二叉树的先序和中序遍历对应。

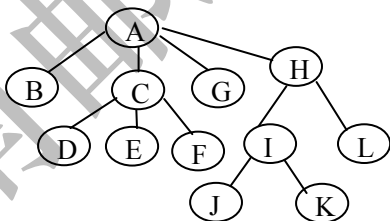
58. 前序遍历是“根左右”，中序遍历是“左根右”，后序遍历是“左右根”。若将“根”去掉，三种遍历就剩“左右”。三种遍历中的差别就是访问根结点的时机不同。二叉树是递归定义的，对左右子树均是按左右顺序来遍历的，因此所有叶子结点间的先后关系都是相同的。

59. 本题的核心是三种遍历的顺序：“根左右”-“左根右”-“左右根”，但对本题的解答必须先定义结点间相互关系的“左右”。本解答中将 N 是 M 的左子女，当作 N 在 M 的左边，而 N 是 M 的右子女，当作 N 在 M 的右边。若定义 P 是 M 和 N 的最近公共祖先，N 在 P 的左子树中，M 在 P 的右子树中，称 N 在 M 的左边，那时的答案是不一样的。

	先根遍历时 n 先被访问	中根遍历时 n 先被访问	后根遍历时 n 先被访问
N 在 M 的左边		√	√
N 在 M 的右边			√
N 是 M 的祖先	√		
N 是 M 的子孙			√

60. HIDJKEBLFGCA

61.



62. 后序遍历的顺序是“左子树—右子树—根结点”。因此，二叉树最左下的叶子结点是遍历的第一个结点。下面的语句段说明了这一过程（设 p 是二叉树根结点的指针）。

```

if(p!=null)
{
    while (p->lchild!=null || p->rchild!=null)
    {
        while(p->lchild!=null) p=p->lchild;
        if(p->rchild!=null) p=p->rchild; } }
return(p); //返回后序序列第一个结点的指针;

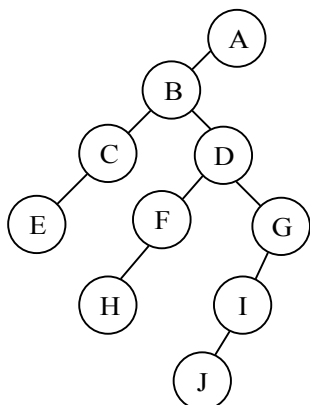
```

63. 采用前序和后序两个序列来判断二叉树上结点 n1 必定是结点 n2 的祖先。

在前序序列中某结点的祖先都排在其前。若结点 n1 是 n2 的祖先，则 n1 必定在 n2 之前。而在后序序列中，某结点的祖先排在其后，即若结点 n1 是 n2 的祖先，则 n1 必在 n2 之后。根据这条规则来判断若结点

n1 在前序序列中在 n2 之前，在后序序列中又在 n2 之后，则它必是结点 n2 的祖先。

64. (1)

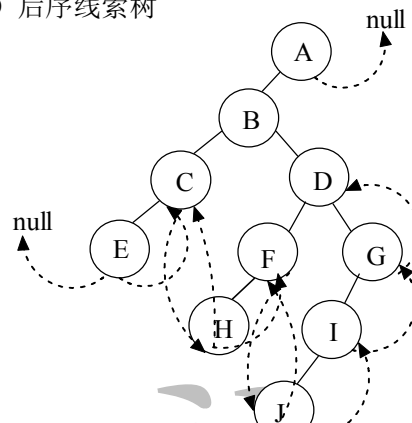


(2) 前序序列: ABCEDFHGIJ

中序序列: ECBHFDJIGA

后序序列: ECHFJIGDBA

(3) 后序线索树



65. 最后一个递归调用语句所保留的参数没有意义。这类递归因其在算法最后，通常被称为“尾递归”，可不用栈且将其（递归）消除。如中序遍历递归算法中，最后的递归语句 `inorder (bt->rchild)` 可改为下列语句段：

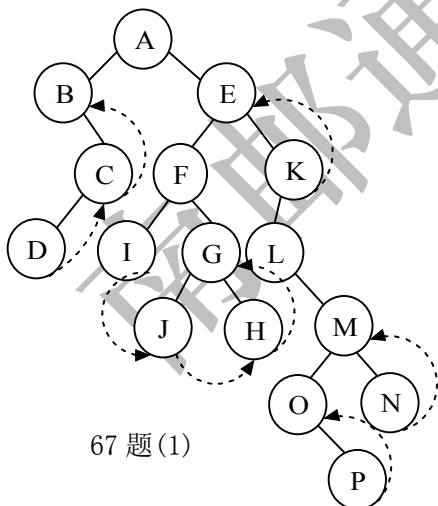
```
bt=bt->rchild;
```

```
while (bt->rchild!=null)
```

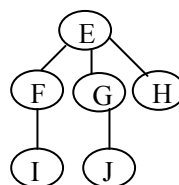
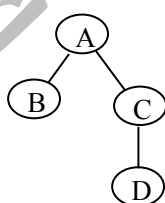
```
{inorder (bt ->lchild); printf( "%c", bt ->data); //访问根结点，假定结点数据域为字符
  bt=bt ->rchild; }
```

66. 在二叉链表表示的二叉树中，引入线索的目的主要是便于查找结点的前驱和后继。因为若知道各结点的后继，二叉树的遍历就变成非常简单。二叉链表结构查结点的左右子女非常方便，但其前驱和后继是在遍历中形成的。为了将非线性结构二叉树的结点排成线性序列，利用结点的空链域，左链为空时用作前驱指针，右链为空时作为后继指针。再引入左右标记 `ltag` 和 `rtag`，规定 `ltag=0`, `lchild` 指向左子女, `ltag=1` 时, `lchild` 指向前驱；当 `rtag=0` 时, `rchild` 指向右子女, `rtag=1` 时, `rchild` 指向后继。这样，在线索二叉树（特别是中序线索二叉树）上遍历就消除了递归，也不使用栈（后序线索二叉树查后继仍需要栈。）

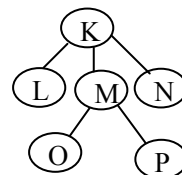
67.



67 题 (1)



67 题 (2)



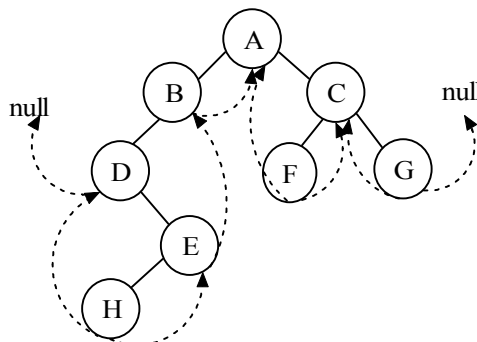
(3) 后根遍历森林，结点序列为：

BDCAIFJGHELOPMNK

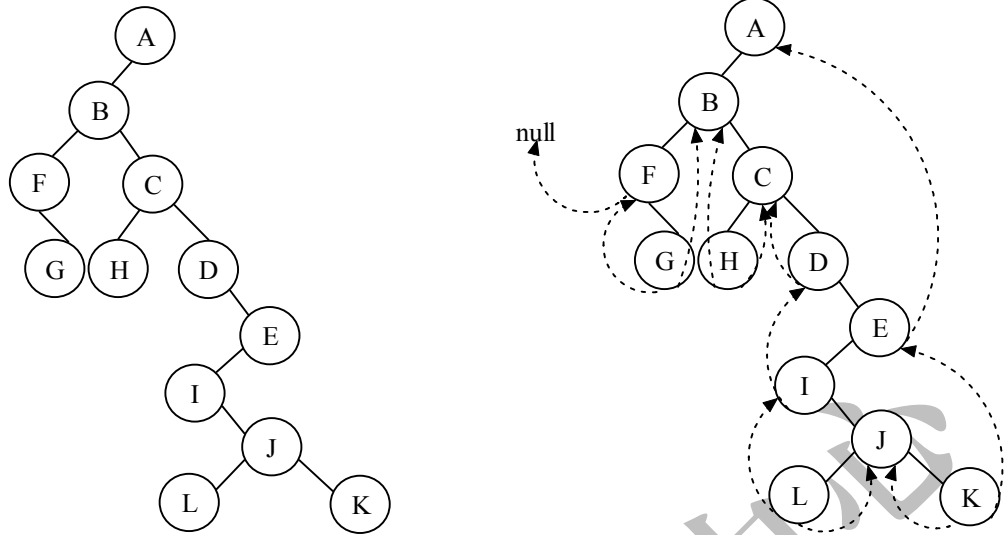
68. (1) 前序序列: ABDEHCFG

(2) 中序序列: DHEBAFCG

(3) 后序序列: HEDBFGCA



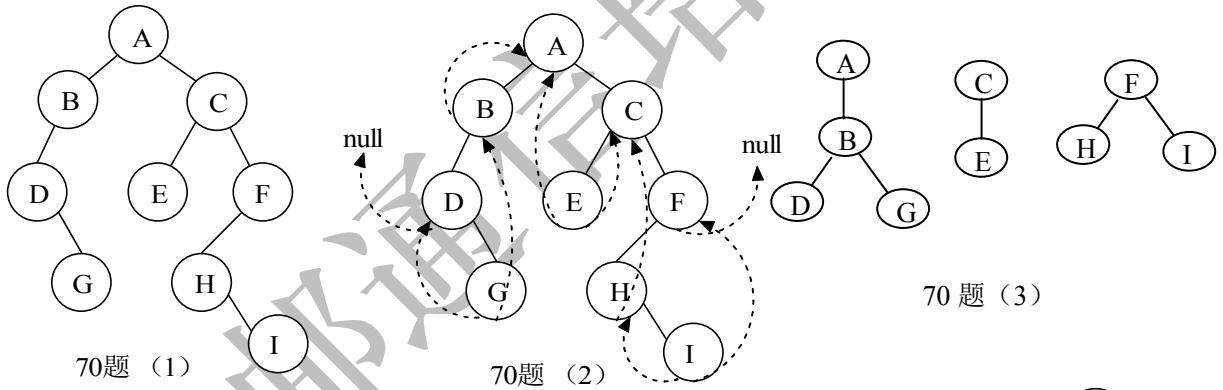
69. (1)



(2) BiTree INORDER-PRIOR(N, X) //在中序线索二叉树上查找结点 N 的前驱结点 X

```
{if(n->ltag==1) {X=N->lchild; return (X);}
  else {p=N->lchild;
        while (p->rtag==0) p=p->rchild;
        X=p;return(p);} }
```

70.



70题 (1)

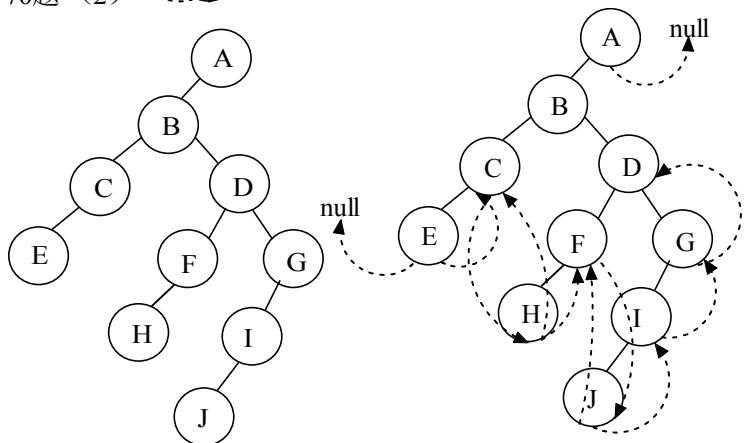
70题 (2)

70 题 (3)

71.

前序序列: A B C E D F H G I J
中序序列: E C B H F D J I G A
后序序列: E C H F J I G D B A

71 题 (2)



71 题 (3)

对前序线索二叉树，当二叉树非空时，若其结点有左子女 (ltag=0)，左子女是后继；否则，若结点有右子女 (rtag=0)，则右子女是后继；若无右子女 (rtag=1)，右子女是线索，也指向后继。所以，对任何前序线索二叉树进行前序遍历均不需使用栈。

74. **if**(p->ltag==0) **return**(p->lchild); //左子女不空, 左子女为直接后继结点

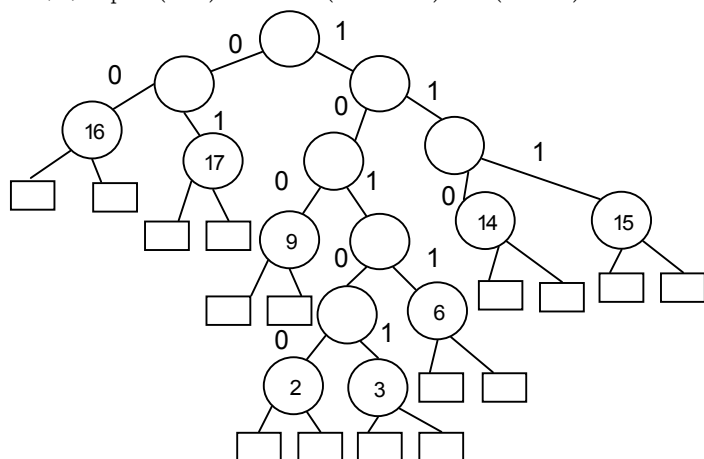
75. 后序线索树中结点的后继（根结点无后继），要么是其右线索（当结点的 $rtag=1$ 时），要么是其双亲结点右子树中最左下的叶子结点。对中序线索二叉树某结点，若其左标记等于 1，则左子女为线索，指向直接前驱；否则，其前驱是其左子树上按中序遍历的最后一个结点。

Data	A	B	C	D	E	F	G	H	I	J	K
Ltag	0	1	0	0	0	1	0	1	1	1	1
Fch	2	null	5	7	8	5	11	2	8	9	3
Rtag	1	0	0	1	0	1	1	0	0	1	1
Nsib	null	3	4	1	6	3	4	9	10	5	7

77题中序线索树

78. 字符 A, B, C, D 出现的次数为 9, 1, 5, 3。其哈夫曼编码如下 A:1, B:000, C:01, D:001

79. (2) $wpl = (2+3)*5+6*4+(9+14+15)*3+(16+17)*2=229$



79题 (1)

(3) 编码为: 15:111, 3:10101, 14:110, 2:10100, 6:1011, 9:100, 16:00, 17:01

(4) 常用哈夫曼树为通讯用的字符编码, 本题中集合的数值解释为字符发生的频率(次数)。由哈夫曼树构造出哈夫曼编码。译码时, 进行编码的“匹配”, 即从左往右扫描对方发来的“编码串”, 用字符编码去匹配, 得到原来的元素(本题中的数)。

80. 首先确定是否需加“虚权值”(即权值为0), 对 m 个权值, 建 k 叉树, 若 $(m-1) \% (k-1) = 0$, 则不需要加“虚权值”, 否则, 第一次归并时需 $(m-1) \% (k-1) + 1$ 个权值归并。建立 k 叉树的过程如下:

(1) 将 m 个权值看作 m 棵只有根结点的 k 叉树的集合 $F = \{T_1, T_2, \dots, T_m\}$ 。

(2) 从 F 中选 k (若需加虚权值, 则第一次选 $(m-1) \% (k-1) + 1$) 个权值最小的树作子树, 构成一棵 k 叉树, k 叉树根结点的权值为所选的 k 个树根结点权值之和, 在 F 中删除这 k 棵子树, 并将新 k 叉树加入到 F 中。

(3) 从 F 中选 k 个权值最小的树作子树, 构成一棵 k 叉树, 其根结点权值等于所选的 k 棵树根结点权值之和, 在 F 中删除这 k 棵树, 并将新得到的树加到 F 中。

(4) 重复 (3), 直到 F 中只有一棵树为止, 这就是最优的 k 叉树。

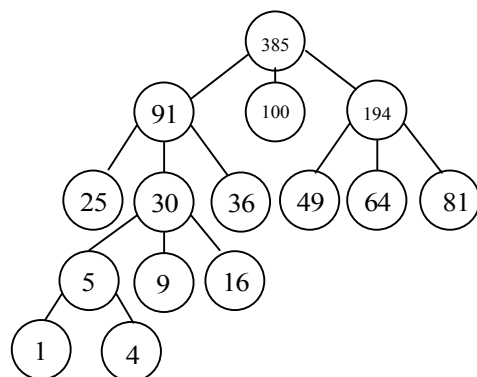
对本题 10 个权值, 构造最优三叉树。

因 $(10-1) \% (3-1) = 1$,

所以第一次用 2 个权值合并。

最小加权路径长度:

$$(1+4)*4+(9+16)*3+(25+36+49+64+81)*2+100*1=705$$



	字符	weight	Parent	lch	rch
1	A	3	0	0	0
2	B	12	0	0	0
3	C	7	0	0	0
4	D	4	0	0	0
5	E	2	0	0	0
6	F	8	0	0	0
7	G	11	0	0	0
8			0	0	0
9			0	0	0
10			0	0	0
11			0	0	0
12			0	0	0
13			0	0	0

	字符	weight	parent	lch	rch
1	A	3	8	0	0
2	B	12	12	0	0
3	C	7	10	0	0
4	D	4	9	0	0
5	E	2	8	0	0
6	F	8	10	0	0
7	G	11	11	0	0
8		5	9	5	1
9		9	11	4	8
10		15	12	3	6
11		20	13	9	7
12		27	13	2	10
13		47	0	11	12

81. 初态图

终态图

82. 前缀码是一编码不是任何其它编码前缀的编码。例如，0 和 01 就不是前缀码，因为编码 0 是编码 01 的前缀。仅从编码来看，0 和 01 是前缀码，但因历史的原因，它不被称为前缀码，而是把一编码不是另一编码前缀的编码称为前缀码。

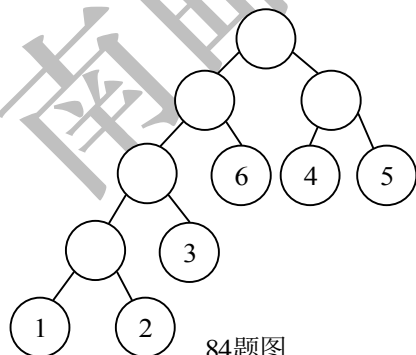
利用二叉树可以构造前缀码，例如，以 A, B, C, D 为叶子可构成二叉树，将左分枝解释为 0，右分枝解释为 1，从根结点到叶子结点的 0、1 串就是叶子的前缀码。用哈夫曼树可构造出最优二叉树，使编码长度最短。

83. 哈夫曼树只有度为 0 的叶子结点和度为 2 的分枝结点，设数量分别为 n_0 和 n_2 ，则树的结点数 n 为 $n=n_0+n_2$ 。另根据二叉树性质：任意二叉树中度为 0 的结点数 n_0 和度为 2 的结点数 n_2 间的关系是 $n_2=n_0-1$ ，代入上式，则 $n=n_0+n_2=2n_0-1$ 。

84. (1) T 树的最大深度 $K_{\max}=6$ (除根外，每层均是两个结点)

T 树的最小深度 $K_{\min}=4$ (具有 6 个叶子的完全二叉树是其中的一种形态)

(2) 非叶子结点数是 5。($n_2=n_0-1$) (3) 哈夫曼树见下图，其带权路径长度 $wpl=51$



84题图

85. (1) 错误，循环结束条件 $top=0$ 不能满足，因为在 $top>1$ 情况下，执行 $top:=top-1$

(2) 错误 (3) 错误 (4) 正确 (5) 结点的深度与其右孩子深度相同，比左孩子深度少 1。

五. 算法设计题

1. [题目分析]以二叉树表示算术表达式，根结点用于存储运算符。若能先分别求出左子树和右子树表示的子表达式的值，最后就可以根据根结点的运算符的要求，计算出表达式的最后结果。

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

```

typedef struct node
{
    ElemType data; float val;
    char optr; //只取 '+', '-', '*', '/'
    struct node *lchild, *rchild } BiNode, *BiTree;

float PostEval(BiTree bt) // 以后序遍历算法求以二叉树表示的算术表达式的值
{
    float lv, rv;
    if(bt!=null)
    {
        lv=PostEval(bt->lchild); // 求左子树表示的子表达式的值
        rv=PostEval(bt->rchild); // 求右子树表示的子表达式的值
        switch(bt->optr)
        {
            case '+': value=lv+rv; break;
            case '-': value=lv-rv; break;
            case '*': value=lv*rv; break;
            case '/': value=lv/rv;
        }
    }
    return(value);
}

```

2. [题目分析] 本题是将符号算术表达式用二叉树表示的逆问题，即将二叉树表示的表达式还原成原表达式。二叉树的中序遍历序列与原算术表达式基本相同，差别仅在于二叉树表示中消除了括号。将中序序列加上括号就恢复原貌。当根结点运算符优先级高于左子树（或右子树）根结点运算符时，就需要加括号。

```

int Precede(char optr1, optr2)
// 比较运算符级别高低，optr1 级别高于 optr2 时返回 1，相等时返回 0，低于时返回 -1
{
    switch(optr1)
    {
        case '+': case '-': if(optr2=='+' || optr2=='-') return(0); else return(-1);
        case '*': case '/': if(optr1=='*' || optr2=='/') return(0); else return(1);
    }
}

void InorderExp (BiTree bt)
// 输出二叉树表示的算术表达式，设二叉树的数据域是运算符或变量名
{
    int bracket;
    if(bt)
    {
        if(bt->lchild!=null)
        {
            bracket=Precede(bt->data, bt->lchild->data) // 比较双亲与左子女运算符优先级
            if(bracket==1) printf( '(' );
            InorderExp(bt->lchild); // 输出左子女表示的算术表达式
            if(bracket==1) printf( ')' ); // 加上右括号
        }
        printf(bt->data); // 输出根结点
        if(bt->rchild!=null) // 输出右子树表示的算术表达式
        {
            bracket=Precede(bt->data, bt->rchild->data)
            if (bracket==1) printf( "(" ); // 右子女级别低，加括号
            InorderExp (bt->rchild);
            if(bracket==1) printf( ")" );
        }
    }
} // 结束 Inorder Exp

```

3. [题目分析] 首先通过对二叉树后序遍历形成后缀表达式，这可通过全局变量的字符数组存放后缀表达式；接着对后缀表达式求值，借助于一栈存放运算结果。从左到右扫描后缀表达式，遇操作数就压入栈中，遇运算符就从栈中弹出两个操作数，作运算符要求的运算，并把运算结果压入栈中，如此下去，直到

后缀表达式结束，这时栈中只有一个数，这就是表达式的值。

```
char ar[maxsize]; //maxsize 是后缀表达式所能达到的最大长度
int i=1;
void PostOrder(BiTree t) //后序遍历二叉树 t, 以得到后缀表达式
{
    if(t)
    {
        PostOrder(t->lchild); PostOrder(t->rchild); ar[i++] = t->data;
    }
    //结束 PostOrder
}
void EXPVALUE()
//对二叉树表示的算术表达式，进行后缀表达式的求值
{
    ar[i] = '\0'; //给后缀表达式加上结束标记
    char value[]; //存放操作数及部分运算结果
    i=1; ch=ar[i++];
    while(ch != '\0')
    {
        switch(ch)
        {
            case ch in op: opnd1=pop(value); opnd2=pop(value); //处理运算符
                        push(operate(opnd2, ch, opnd1)); break;
            default:      push(value, ch); //处理操作数，压入栈中
        }
        ch=ar[i++]; //读入后缀表达式
    }
    printf(value[1]); //栈中只剩下一个操作数，即运算结束。
} //结束 EXPVALUE
```

[算法讨论] 根据题意，操作数是单字母变量，存放运算结果的栈也用了字符数组。实际上，操作数可能是变量，也可以是常量。运算中，两个操作数（opnd1 和 opnd2）也不会直接运算，即两个操作数要从字符转换成数（如‘3’是字符，而数值 3=‘3’-‘0’）。数在压入字符栈也必须转换，算法中的 operate 也是一个需要编写的函数，可参见上面算法设计题 1，其细节不再深入讨论。

4. [题目分析] 当森林（树）以孩子兄弟表示法存储时，若结点没有孩子（fch=null），则它必是叶子，总的叶子结点个数是孩子子树（fch）上的叶子数和兄弟（nsib）子树上叶结点个数之和。

```
typedef struct node
{
    ElemType data; //数据域
    struct node *fch, *nsib; //孩子与兄弟域
} *Tree;
int Leaves(Tree t)
//计算以孩子-兄弟表示法存储的森林的叶子数
{
    if(t)
    {
        if(t->fch == null) //若结点无孩子，则该结点必是叶子
            return(1+Leaves(t->nsib)); //返回叶子结点和其兄弟子树中的叶子结点数
        else return (Leaves(t->fch)+Leaves(t->nsib)); //孩子子树和兄弟子树中叶子数之和
    }
    //结束 Leaves
```

5. [题目分析] 由指示结点 i 左儿子和右儿子的两个一维数组 L[i] 和 R[i]，很容易建立指示结点 i 的双亲的一维数组 T[i]，根据 T 数组，判断结点 U 是否是结点 V 后代的算法，转为判断结点 V 是否是结点 U 的祖先的问题。

```
int Generation(int U, V, N, L[], R[], T[])
//L[] 和 R[] 是含有 N 个元素且指示二叉树结点 i 左儿子和右儿子的一维数组，
//本算法据此建立结点 i 的双亲数组 T，并判断结点 U 是否是结点 V 的后代。
{
    for(i=1; i<=N; i++) T[i]=0; //T 数组初始化
    for (i=1; i<=N; i++) //根据 L 和 R 填写 T
```

```

if(L[i]!=0) T[L[i]]=i;    //若结点 i 的左子女是 L, 则结点 L 的双亲是结点 i
for(i=1;i<=N;i++)
    if (R[i]!=0) T[R[i]]=i;    //i 的右子女是 r, 则 r 的双亲是 i
int parent=U;                //判断 U 是否是 V 的后代
while (parent!=V && parent!=0) parent=T[parent];
if (parent==V) {printf(“结点 U 是结点 V 的后代”);return(1);}
else{ printf(“结点 U 不是结点 V 的后代”);return(0);}
}结束 Generation

```

6. [题目分析]二叉树是递归定义的,以递归方式建立最简单。判定是否是完全二叉树,可以使用队列,在遍历中利用完全二叉树“若某结点无左子女就不应有右子女”的原则进行判断。

```

BiTree Creat()                //建立二叉树的二叉链表形式的存储结构
{ElemType x; BiTree bt;
scanf(“%d”,&x); //本题假定结点数据域为整型
if(x==0) bt=null;
else if(x>0)
    {bt=(BiNode *)malloc(sizeof(BiNode));
    bt->data=x; bt->lchild=creat(); bt->rchild=creat();
    }
    else error(“输入错误”);
return(bt);
} //结束 BiTree

int JudgeComplete(BiTree bt) //判断二叉树是否是完全二叉树,如是,返回 1, 否则, 返回 0
{int tag=0; BiTree p=bt, Q[]; // Q 是队列, 元素是二叉树结点指针, 容量足够大
if(p==null) return (1);
QueueInit(Q); QueueIn(Q,p); //初始化队列, 根结点指针入队
while (!QueueEmpty(Q))
    {p=QueueOut(Q); //出队
    if (p->lchild && !tag) QueueIn(Q,p->lchild); //左子女入队
    else {if (p->lchild) return 0; //前边已有结点为空, 本结点不空
          else tag=1; //首次出现结点为空
    }
    if (p->rchild && !tag) QueueIn(Q,p->rchild); //右子女入队
    else if (p->rchild) return 0; else tag=1;
    } //while
return 1; } //JudgeComplete

```

[算法讨论]完全二叉树证明还有其它方法。判断时易犯的错误的证明其左子树和右子数都是完全二叉树,由此推出整棵二叉树必是完全二叉树的错误结论。

```

7. BiTree Creat(ElemType A[],int i)
//n 个结点的完全二叉树存于一维数组 A 中, 本算法据此建立以二叉链表表示的完全二叉树
{BiTree tree;
if (i<=n) {tree=(BiTree)malloc(sizeof(BiNode)); tree->data=A[i];
    if(2*i>n) tree->lchild=null; else tree->lchild=Creat(A,2*i);
    if(2*i+1>n) tree->rchild=null; else tree->rchild=Creat(A,2*i+1);
}
return (tree); } //Creat

```

[算法讨论]初始调用时, i=1。

8. [题目分析]二叉树高度可递归计算如下: 若二叉树为空, 则高度为零, 否则, 二叉树的高度等于左

右子树高度的大者加 1。这里二叉树为空的标记不是 null 而是 0。设根结点层号为 1，则每个结点的层号等于其双亲层号加 1。

现将二叉树的存储结构定义如下：

```
typedef struct node
{
    int L[]; //编号为 i 的结点的左儿子
    int R[]; //编号为 i 的结点的右儿子
    int D[]; //编号为 i 的结点的层号
    int i;   //存储结点的顺序号（下标）
} tnode;
```

(1) **int** Height(tnode t, **int** i) //求二叉树高度，调用时 i=1

```
{
    int lh, rh;
    if (i==0) return (0);
    else {lh=Height(t, t.L[i]); rh=Height(t, t.R[i]);
        if (lh>rh) return (lh+1); else return (rh+1);
    }
} //结束 Height
```

(2) **int** Level (tnode t) //求二叉树各结点的层号，已知编号为 1 的结点是根，且层号为 1

```
{
    t.D[1]=1;
    for (i=1; i<=n; i++) {depth=t.D[i]; //取出根结点层号
        if (t.L[i]!=0) t.D[t.L[i]]=depth+1; //i 结点的左儿子层号
        if (t.R[i]!=0) t.D[t.R[i]]=depth+1; //i 结点的右儿子层号
    }
} //结束 level
```

9. [题目分析] 二叉树采用顺序存储结构（一维数组）是按完全二叉树的形状存储的，不是完全二叉树的二叉树顺序存储时，要加“虚结点”。数组中的第一个元素是根结点。本题中采用队列结构。

```
typedef struct
```

```
{
    BiTree bt; //二叉树结点指针
    int num; } tnode // num 是结点在一维数组中的编号
tnode Q[maxsize]; //循环队列, 容量足够大
```

```
void creat(BiTree T, ElemType BT[ ])
```

//深度 h 的二叉树存于一维数组 BT[1:2^h-1] 中，本算法生成该二叉树的二叉链表存储结构

```
{
    tnode tq; //tq 是队列元素
```

```
    int len=2h-1; //数组长度
```

```
    T=(BiTree) malloc(sizeof(BiNode)); //申请结点
```

```
    T->data=BT[1]; //根结点数据
```

```
    tq.bt=T; tq.num=1;
```

```
    Q[1]=tq; //根入队列
```

```
    front=0; rear=1; //循环队列头、尾指针
```

```
    while (front!=rear) //当队列不空时循环
```

```
    {
        front=(front+1) % maxsize ;
```

```
        tq=Q[front] ; p=tq.bt; i=tq.num ; //出队，取出结点及编号
```

```
        if (BT[2*i]=='#' || 2*i>len) p->lchild=null; //左子树为空，'#' 表示虚结点
```

```
        else //建立左子女结点并入队列
```

```
            {p->lchild=(BiTree) malloc(sizeof(BiNode)); //申请结点空间
```

```
                p->lchild->data=BT[2*i]; //左子女数据
```

```
                tq.bt=p->lchild; tq.num=2*i; rear=(rear+1) % maxsize ; //计算队尾位置
```

```

    Q[rear]=tq;           //左子女结点及其编号入队
}
if(BT[2*i+1]== '#' || 2*i+1>len) p->rchild=null; //右子树为空
else //建立右子女结点并入队列
{
    p->rchild=(BiTree)malloc(sizeof (BiNode)); //申请结点空间
    p->rchild->data=BT[2*i+1]; tq.bt=p->rchild; tq.num=2*i+1;
    rear=(rear+1)%maxsize; Q[rear]=tq; //计算队尾位置,右子女及其编号入队
}
} //while
} //结束 creat

```

[算法讨论] 本题中的虚结点用‘#’表示。应根据二叉树的结点数据的类型而定。

10. [题目分析] 本题静态链表中结点是按动态二叉链表的前序遍历顺序存放的。首先对动态二叉链表的二叉树进行前序遍历, 填写静态链表的“下标”和 data 域, 再对动态二叉链表的二叉树进行层次遍历, 设队列 Q, 填写静态链表的 lchild 域和 rchild 域。

```

typedef struct node //静态链表结点结构
{
    ElemType data; //结点数据
    int row, lchild, rchild; //下标, 左右子女
} component;
component st[]; //st 容量足够大
struct node {BiTree t; int idx; } qbt;
static int num=0;
void PreOrder(BiTree bt);
// 前序遍历二叉树, 填写静态链表的“下标”和 data 域
{
    if (bt)
    {
        st[++num].data=bt->data; st[num].row=num;
        PreOrder(bt->lchild); PreOrder(bt->rchild);
    }
}
int Locate(ElemType x)
//在静态链表中查二叉树结点的下标
{
    for (i=1; i<=num; i++) if (st[i].data==x) return (i);
}
void DynaToST (BiTree bt) //将二叉树的动态二叉链表结构转为静态链表结构
{
    int i=0; //i 为静态链表 st 的下标
    if (bt!=null)
    {
        QueueInit(Q); //Q 是队列, 容量足够大, 队列元素是 qbt
        qbt.t=bt; qbt.idx=1; QueueIn(Q, qbt); st[1].data=bt->data;
        while(!QueueEmpty(Q))
        {
            qbt=QueueOut(Q); //出队列
            p=qbt.t; i=qbt.idx; //二叉树结点及在静态链表中的下标
            if (p->lchild!=null) //若左子女存在, 查其在静态链表中的下标, 填 lchild 域值
            {
                lch=Locate(p->lchild->data); st[i].lchild=lch;
                qbt.t=p->lchild; qbt.idx=lch; QueueIn(Q, qbt);
            }
            else st[i].lchild=0; //无左子女, 其 lchild 域填 0
            if (p->rchild!=null) //若右子女存在, 查其在静态链表中的下标, 填 rchild 域值
            {
                rch=Locate(p->rchild->data); st[i].rchild=rch;
            }
        }
    }
}

```

```

        qbt.t=p->rchild; qbt.idx=rch; QueueIn(Q,qbt); }
    else st[i].rchild=0; //无左子女, 其 lchild 域填 0
} //while
} //结束 DynaToST

```

11. [题目分析] 由于以双亲表示法作树的存储结构, 找结点的双亲容易。因此我们可求出每一结点的层次, 取其最大层次就是树有深度。对每一结点, 找其双亲, 双亲的双亲, 直至(根)结点双亲为 0 为止。

```

int Depth(Ptree t) //求以双亲表示法为存储结构的树的深度, Ptree 的定义参看教材
{
    int maxdepth=0;
    for(i=1; i<=t.n; i++)
    {
        temp=0; f=i;
        while(f>0) {temp++; f=t.nodes[f].parent; } // 深度加 1, 并取新的双亲
        if(temp>maxdepth) maxdepth=temp; //最大深度更新
    }
    return(maxdepth); //返回树的深度
} //结束 Depth

```

12. [题目分析] 二叉树是递归定义的, 其运算最好采取递归方式

```

int Height(btree bt) //求二叉树 bt 的深度
{
    int hl, hr;
    if (bt==null) return(0);
    else {hl=Height(bt->lch); hr=Height(bt->rch);
        if(hl>hr) return (hl+1); else return (hr+1);
    }
}

```

13. [题目分析] 求二叉树高度的算法见上题。求最大宽度可采用层次遍历的方法, 记下各层结点数, 每层遍历完毕, 若结点数大于原先最大宽度, 则修改最大宽度。

```

int Width(BiTree bt) //求二叉树 bt 的最大宽度
{
    if (bt==null) return (0); //空二叉树宽度为 0
    else
    {
        BiTree Q[]; //Q 是队列, 元素为二叉树结点指针, 容量足够大
        front=1; rear=1; last=1; //front 队头指针, rear 队尾指针, last 同层最右结点在队列中的位置
        temp=0; maxw=0; //temp 记局部宽度, maxw 记最大宽度
        Q[rear]=bt; //根结点入队列
        while(front<=last)
        {
            p=Q[front++]; temp++; //同层元素数加 1
            if (p->lchild!=null) Q[++rear]=p->lchild; //左子女入队
            if (p->rchild!=null) Q[++rear]=p->rchild; //右子女入队
            if (front>last) //一层结束,
            {
                last=rear;
                if(temp>maxw) maxw=temp; //last 指向下层最右元素, 更新当前最大宽度
                temp=0;
            } //if
        } //while
        return (maxw);
    }
} //结束 width

```

14. [题目分析] 由孩子兄弟链表表示的树, 求高度的递归模型是: 若树为空, 高度为零; 若第一子女为空, 高度为 1 和兄弟子树的高度的大者; 否则, 高度为第一子女树高度加 1 和兄弟子树高度的大者。其非递归

算法使用队列，逐层遍历树，取得树的高度。

```

int Height(CSTree bt)    //递归求以孩子兄弟链表表示的树的深度
{
    int hc, hs;
    if (bt==null) return (0);
    else if (!bt->firstchild) return (1+height(bt->nextsibling)); //子女空，查兄弟的深度
    else // 结点既有第一子女又有兄弟，高度取子女高度+1 和兄弟子树高度的大者
        {
            hc=height(bt->firstchild); //第一子女树高
            hs=height(bt->nextsibling); //兄弟树高
            if(hc+1>hs) return(hc+1); else return (hs);
        }
} //结束 height

int height(CSTree t)    //非递归遍历求以孩子兄弟链表表示的树的深度
{
    if(t==null) return(0);
    else{
        int front=1, rear=1; //front, rear 是队头队尾元素的指针
        int last=1, h=0; //last 指向树中同层结点中最后一个结点，h 是树的高度
        Q[rear]=t; //Q 是以树中结点为元素的队列
        while(front<=last)
        {
            t=Q[front++]; //队头出列
            while(t!=null) //层次遍历
            {
                if (t->firstchild) Q[++rear]=t->firstchild; //第一子女入队
                t=t->nextsibling; //同层兄弟指针后移
            }
            if(front>last) //本层结束，深度加 1（初始深度为 0）
                {h++; last=rear;} //last 再移到指向当前层最右一个结点
        } //while(front<=last)
    } //else
} //Height

```

15. [题目分析] 后序遍历最后访问根结点，即在递归算法中，根是压在栈底的。采用后序非递归算法，栈中存放二叉树结点的指针，当访问到某结点时，栈中所有元素均为该结点的祖先。本题要找 p 和 q 的最近共同祖先结点 r，不失一般性，设 p 在 q 的左边。后序遍历必然先遍历到结点 p，栈中元素均为 p 的祖先。将栈拷入另一辅助栈中。再继续遍历到结点 q 时，将栈中元素从栈顶开始逐个到辅助栈中去匹配，第一个匹配（即相等）的元素就是结点 p 和 q 的最近公共祖先。

```

typedef struct
{
    BiTree t; int tag; //tag=0 表示结点的左子女已被访问，tag=1 表示结点的右子女已被访问
} stack;

stack s[], s1[]; //栈，容量够大

BiTree Ancestor(BiTree ROOT, p, q, r) //求二叉树上结点 p 和 q 的最近共同祖先结点 r。
{
    top=0; bt=ROOT;
    while(bt!=null || top>0)
    {
        while(bt!=null && bt!=p && bt!=q) //结点入栈
            {s[++top].t=bt; s[top].tag=0; bt=bt->lchild;} //沿左分枝向下
        if(bt==p) //不失一般性，假定 p 在 q 的左侧，遇结点 p 时，栈中元素均为 p 的祖先结点
            {for(i=1; i<=top; i++) s1[i]=s[i]; top1=top;} //将栈 s 的元素转入辅助栈 s1 保存
        if(bt==q) //找到 q 结点。
            for(i=top; i>0; i--) //将栈中元素的树结点到 s1 去匹配

```



```

{pp=s[i].t;
  for (j=top1;j>0;j--)
    if(s1[j].t==pp) {printf(“p 和 q 的最近共同的祖先已找到”); return (pp);}
}
while(top!=0 && s[top].tag==1) top--; //退栈
if (top!=0) {s[top].tag=1;bt=s[top].t->rchild;} //沿右分枝向下遍历
} //结束 while(bt!=null || top>0)
return(null); // q、p 无公共祖先
} //结束 Ancestor

```

16. [题目分析] 二叉树顺序存储，是按完全二叉树的格式存储，利用完全二叉树双亲结点与子女结点编号间的关系，求下标为 i 和 j 的两结点的双亲，双亲的双亲，等等，直至找到最近的公共祖先。

void Ancestor(ElmType A[], **int** n, i, j)

// 二叉树顺序存储在数组 A[1..n] 中，本算法求下标分别为 i 和 j 的结点的最近公共祖先结点的值。

```

{while(i!=j)
  if(i>j) i=i/2; //下标为 i 的结点的双亲结点的下标
  else j=j/2; //下标为 j 的结点的双亲结点的下标
  printf(“所查结点的最近公共祖先的下标是%d, 值是%d”, i, A[i]); //设元素类型整型。
} // Ancestor

```

17. [题目分析] 用二叉树表示出父子，夫妻和兄弟三种关系，可以用根结点表示父（祖先），根结点的左子女表示妻，妻的右子女表示子。这种二叉树可以看成类似树的孩子兄弟链表表示法；根结点是父，根无右子女，左子女表示妻，妻的右子女（右子女的右子女等）均可看成兄弟（即父的所有儿子），兄弟结点又成为新的父，其左子女是兄弟（父的儿子）妻，妻的右子女（右子女的右子女等）又为儿子的儿子等等。首先递归查找某父亲结点，若查找成功，则其左子女是妻，妻的右子女及右子女的右子女等均为父亲的孩子。

```

BiTree Search(BiTree t, ElmType father) //在二叉树上查找值为 father 的结点
{if(t==null) return (null); //二叉树上无 father 结点
 else if(t->data==father) return(t); //查找成功
  p=Search(t->lchild, father); p=Search(t->rchild, father); }
} //结束 Search

void PrintSons(BiTree t, ElmType p) //在二叉树上查找结点值为 p 的所有的儿子
{p=Search(t, p); //在二叉树 t 上查找父结点 p
 if(p!=null) //存在父结点
  {q=p->lchild; q=q->rchild; //先指向其妻结点，再找到第一个儿子
   while(q!=null) {printf(q->data); q=q->rchild;} //输出父的所有儿子
  }
} //结束 PrintSons

```

18. [题目分析] 后序遍历最后访问根结点，当访问到值为 x 的结点时，栈中所有元素均为该结点的祖先。

void Search(BiTree bt, ElmType x) //在二叉树 bt 中，查找值为 x 的结点，并打印其所有祖先

```

{typedef struct
{BiTree t; int tag; }stack; //tag=0 表示左子女被访问，tag=1 表示右子女被访问
stack s[]; //栈容量足够大
top=0;
while(bt!=null || top>0)
{while(bt!=null && bt->data!=x) //结点入栈
  {s[++top].t=bt; s[top].tag=0; bt=bt->lchild;} //沿左分枝向下
  if(bt->data==x) { printf(“所查结点的所有祖先结点的值为:\n”); //找到 x

```

```

    for(i=1;i<=top;i++) printf(s[i].t->data); return; } //输出祖先值后结束
    while(top!=0 && s[top].tag==1) top--; //退栈（空遍历）
    if(top!=0) {s[top].tag=1;bt=s[top].t->rchild;} //沿右分枝向下遍历
} // while(bt!=null||top>0)
}结束 search

```

因为查找的过程就是后序遍历的过程，使用的栈的深度不超过树的深度，算法复杂度为 $O(\log n)$ 。

19. [题目分析] 先序遍历二叉树的非递归算法，要求进栈元素少，意味着空指针不进栈。

```

void PreOrder(Bitree bt)//对二叉数 bt 进行非递归遍历
{int top=0; Bitree s[]; //top 是栈 s 的栈顶指针，栈中元素是树结点指针，栈容量足够大
 while(bt!=null || top>0)
 {while(bt!=null)
  {printf(bt->data); //访问根结点
   if(bt->rchild) s[++top]=bt->rchild; //若有右子女，则右子女进栈
   bt=bt->lchild; }
  if (top>0) bt=s[top--];
 }
}

```

本题中的二叉树中需进栈的元素有 C, H, K, F。

20. [题目分析] 二叉树的顺序存储是按完全二叉树的顺序存储格式，双亲与子女结点下标间有确定关系。对顺序存储结构的二叉树进行遍历，与二叉链表类似。在顺序存储结构下，判二叉树为空时，用结点的下标大于 n （完全二叉树）或 0（对一般二叉树的“虚结点”）。本题是完全二叉树。

```

void PreOrder(ElemType bt, int n)//对以顺序结构存储的完全二叉树 bt 进行前序遍历
{int top=0, s[]; //top 是栈 s 的栈顶指针，栈容量足够大
 while(i<=n||top>0)
 {while(i<=n)
  { printf(bt[i]); //访问根结点;
   if (2*i+1<=n) s[++top]=2*i+1; //右子女的下标位置进栈
   i=2*i; } //沿左子女向下
  if(top>0) i=s[top--]; } //取出栈顶元素
 } //结束 PreOrder
}

```

21. [题目分析] 本题使用的存储结构是一种双亲表示法，对每个结点，直接给出其双亲（的下标），而且用正或负表示该结点是双亲的右子女或左子女。该类结构不适于直接进行前序遍历（即若直接前序遍历，算法要很复杂），较好的办法是将这类结构转为结点及其左右子女的顺序存储结构，即

```

Tree2=ARRAY[1..max] OF RECORD data: char; //结点数据
                                lc,rc: integer; END; //结点的左右子女在数组中的下标
void Change (Tree t, Tree2 bt, int *root) //先将 t 转为如上定义类型的变量 bt;
{for(i=1;i<=max;i++) {bt[i].lc=bt[i].rc=0;} //先将结点的左右子女初始化为 0
 for(i=1;i<=max;i++) //填入结点数据，和结点左右子女的信息
 {bt[i].data=t[i].data;
  if(t[i].parent<0) bt[-t[i].parent].lc=i; //左子女
  else if(t[i].parent>0) bt[t[i].parent].rc=i; //右子女
  else *root=i; //root 记住根结点
 } } //change
void PreOrder(Tree2 bt) //对二叉树进行前序遍历
{int *root, top=0; int s[]; //s 是栈
 change(t, bt, root); int i=*root;

```

```

while(i!=0||top>0)
{
    while (i!=0)
    {
        printf (bt[i].data); if(bt[i].rc!=0) s[++top]=bt[i].rc; //右子女进栈
        i=bt[i].lc;
    }
    if (top>0) i=s[top--];
} } //结束 preorder

```

[算法讨论]本题的前序递归算法如下

```

void PreOrder(int root)//root 是二叉树根结点在顺序存储中的下标, 本算法前序遍历二叉树 bt
{
    if(root!=0) {printf(bt[root].data); //访问根结点
        PreOrder(bt[root].lc); //前序遍历左子树
        PreOrder(bt[root].rc); //前序遍历右子树
    } } //结束 preorder, 初始调用时, root 是根结点的下标

```

这类问题的求解方法值得注意。当给定数据存储结构不合适时, 可由已给结构来构造好的数据结构, 以便于运算。象上面第 5 题也是这样, 先根据 L 和 R 数组, 构造一个结点的双亲的数组 T。

22. [题目分析]二叉树先序序列的最后一个结点, 若二叉树有右子树, 则是右子树中最右下的叶子结点; 若无右子树, 仅有左子树, 则是左子树最右下的叶子结点; 若二叉树无左右子树, 则返回根结点。

BiTree LastNode(BiTree bt)//返回二叉树 bt 先序序列的最后一个结点的指针

```

BiTree p=bt;
if(bt==null) return(null);
else while(p)
{
    if (p->rchild) p=p->rchild; //若右子树不空, 沿右子树向下
    else if (p->lchild) p=p->lchild; //右子树空, 左子树不空, 沿左子树向下
    else return(p); //p 即为所求
} //结束 lastnode

```

23. [题目分析]高度为 K 的二叉树, 按顺序方式存储, 要占用 $2^K - 1$ 个存储单元, 与实际结点个数 n 关系不大, 对不是完全二叉树的二叉树, 要增加“虚结点”, 使其在形态上成为完全二叉树。

```

int m=2^K - 1; //全局变量
void PreOrder(ElemType bt[], i )
//递归遍历以顺序方式存储的二叉树 bt, i 是根结点下标 (初始调用时为 1)。
{
    if (i<=m) //设虚结点以 0 表示
    {
        printf(bt[i]); //访问根结点
        if(2*i<=m && bt[2*i]!=0) PreOrder(bt, 2*i); //先序遍历左子树
        if(2*i+1<=m && bt[2*i+1]!=0) PreOrder(bt, 2*i+1); //先序遍历右子树
    } } //结束 PreOrder

```

二叉树中最大序号的叶子结点, 是在顺序存储方式下编号最大的结点

```

void Ancesstor(ElemType bt[]) //打印最大序号叶子结点的全部祖先
{
    c=m; while(bt[c]==0) c--; //找最大序号叶子结点, 该结点存储时在最后
    f=c/2; //c 的双亲结点 f
    while(f!=0) //从结点 c 的双亲结点直到根结点, 路径上所有结点均为祖先结点
    {
        printf(bt[f]); f=f/2; } //逆序输出, 最老的祖先最后输出
} //结束

```

24. void InOrder(BiTree bt)

```

BiTree s[], p=bt; //s 是元素为二叉树结点指针的栈, 容量足够大
int top=0;

```

```
while(p || top>0)
{while(p) {s[++top]=p; bt=p->lchild;} //中序遍历左子树
  if(top>0) {p=s[top--]; printf(p->data); p=p->rchild;} //退栈, 访问, 转右子树
}
```

25. [题目分析] 二叉树用顺序方式存储, 其遍历方法与用二叉链表方式存储类似。判空时, 在二叉链表方式下用结点指针是否等于 null, 在顺序存储方式下, 一是下标是否是“虚结点”, 二是下标值是否超过最大值 (高为 H 的二叉树要有 2^H-1 个存储单元, 与实际结点个数关系不大)。当然, 顺序存储方式下, 要告诉根结点的下标。

```
void InOrder(int i) //对顺序存储的二叉树进行中序遍历, i 是根结点的下标
{if(i!=0)
  {InOrder(ar[i].Lc); //中序遍历左子树
    printf(ar[i].data); //访问根结点
    InOrder(ar[i].Rc); //中序遍历右子树
  } } // 结束 InOrder
```

26. [题目分析] 借助队列和栈, 最后弹出栈中元素实现对二叉树按自下至上, 自右至左的层次遍历

```
void InvertLevel(biTree bt) // 对二叉树按自下至上, 自右至左的进行层次遍历
{if(bt!=null)
  {StackInit(s); //栈初始化, 栈中存放二叉树结点的指针
    QueueInit(Q); //队列初始化。队列中存放二叉树结点的指针
    QueueIn(Q, bt);
    while(!QueueEmpty(Q)) //从上而下层次遍历
      {p=QueueOut(Q); push(s, p); //出队, 入栈
        if(p->lchild) QueueIn(Q, p->lchild); //若左子女不空, 则入队列
        if(p->rchild) QueueIn(Q, p->rchild); //若右子女不空, 则入队列
        while(!StackEmpty(s)) {p=pop(s); printf(p->data);} //自下而上, 从右到左的层次遍历
      } //if(bt!=null)
  } //结束 InvertLevel
```

27. int Level(BiTree bt) //层次遍历二叉树, 并统计度为 1 的结点的个数

```
{int num=0; //num 统计度为 1 的结点的个数
  if(bt) {QueueInit(Q); QueueIn(Q, bt); //Q 是以二叉树结点指针为元素的队列
    while(!QueueEmpty(Q))
      {p=QueueOut(Q); printf(p->data); //出队, 访问结点
        if(p->lchild && !p->rchild || !p->lchild && p->rchild) num++; //度为 1 的结点
        if(p->lchild) QueueIn(Q, p->lchild); //非空左子女入队
        if(p->rchild) QueueIn(Q, p->rchild); //非空右子女入队
      } } //if(bt)
  return(num); } //返回度为 1 的结点的个数
```

28. void exchange(BiTree bt) //将二叉树 bt 所有结点的左右子树交换

```
{if(bt) {BiTree s;
  s=bt->lchild; bt->lchild=bt->rchild; bt->rchild=s; //左右子女交换
  exchange(bt->lchild); //交换左子树上所有结点的左右子树
  exchange(bt->rchild); //交换右子树上所有结点的左右子树
}}
```

[算法讨论] 将上述算法中两个递归调用语句放在前面, 将交换语句放在最后, 则是以后序遍历方式交换所有结点的左右子树。中序遍历不适合本题。下面是本题 (1) 要求的非递归算法

```
(1) void exchange(BiTree t) //交换二叉树中各结点的左右孩子的非递归算法
{int top=0; BiTree s[],p; //s 是二叉树的结点指针的栈, 容量足够大
  if(bt)
  {s[++top]=t;
   while(top>0)
   {t=s[top--];
    if(t->lchild||t->rchild) {p=t->lchild;t->lchild=t->rchild;t->rchild=p;} //交换左右
    if(t->lchild) s[++top]=t->lchild; //左子女入栈
    if(t->rchild) s[++top]=t->rchild; //右子女入栈
   } //while(top>0)
  } //if(bt) } // 结束 exchange
```

29. [题目分析]对一般二叉树, 仅根据一个先序、中序、后序遍历, 不能确定另一个遍历序列。但对于满二叉树, 任一结点的左右子树均含有数量相等的结点, 根据此性质, 可将任一遍历序列转为另一遍历序列(即任一遍历序列均可确定一棵二叉树)。

```
void PreToPost(ElemType pre[], post[], int l1, h1, l2, h2)
//将满二叉树的先序序列转为后序序列, l1, h1, l2, h2 是序列初始和最后结点的下标。
{if(h1>=l1)
 {post[h2]=pre[l1]; //根结点
  half=(h1-l1)/2; //左或右子树的结点数
  PreToPost(pre, post, l1+1, l1+half, l2, l2+half-1) //将左子树先序序列转为后序序列
  PreToPost(pre, post, l1+half+1, h1, l2+half, h2-1) //将右子树先序序列转为后序序列
} } //PreToPost
```

```
30. BiTree IntoPost(ElemType in[], post[], int l1, h1, l2, h2)
//in 和 post 是二叉树的中序序列和后序序列, l1, h1, l2, h2 分别是两序列第一和最后结点的下标
{BiTree bt=(BiTree)malloc(sizeof(BiNode)); //申请结点
bt->data=post[h2]; //后序遍历序列最后一个元素是根结点数据
for(i=l1; i<=h1; i++) if(in[i]==post[h2]) break; //在中序序列中查找根结点
if(i==l1) bt->lchild=NULL; //处理左子树
else bt->lchild=IntoPost(in, post, l1, i-1, l2, l2+i-l1-1);
if(i==h1) bt->rchild=NULL; //处理右子树
else bt->rchild=IntoPost(in, post, i+1, h1, l2+i-l1, h2-1);
return(bt); }
```

```
31. TYPE bitreptr= binode;
      binode=RECORD data:ElemType; lchild,rchild:bitreptr END;
PROC PreOrder(bt:bitreptr); //非递归前序遍历二叉树
VAR S:ARRAY[1..max] OF bitreptr; //max 是栈容量, 足够大
inits(S); //栈初始化
WHILE (bt<>NIL) OR (NOT empty(S)) DO
[WHILE (bt<>NIL) DO
 [write(bt↑data); push(S, bt->rchild); bt:=bt↑.lchild;] //访问结点, 右子女进栈
 WHILE (NOT empty(S) AND top(S)=NIL) bt:=pop(S); // 退栈
 IF NOT empty(S) THEN bt:=pop(S);
] ENDP;
```

[算法讨论]若不要求使用 top(S), 以上算法还可简化。

32. [题目分析]二叉树采取顺序结构存储, 是按完全二叉树格式存储的。对非完全二叉树要补上“虚结点”。

由于不是完全二叉树，在顺序结构存储中对叶子结点的判定是根据其左右子女为 0。叶子和双亲结点下标间的关系满足完全二叉树的性质。

```
int Leaves(int h) //求深度为 h 以顺序结构存储的二叉树的叶子结点数
{
    int BT[]; int len=2h-1, count=0; //BT 是二叉树结点值一维数组，容量为 2h
    for (i=1; i<=len; i++) //数组元素从下标 1 开始存放
        if (BT[i]!=0) //假定二叉树结点值是整数，“虚结点”用 0 填充
            if (i*2>len) count++; //第 i 个结点没子女，肯定是叶子
            else if (BT[2*i]==0 && 2*i+1<=len && BT[2*i+1]==0) count++; //无左右子女的结点是叶子
    return (count);
} //结束 Leaves
```

33. [题目分析] 计算每层中结点值大于 50 的结点个数，应按层次遍历。设一队列 Q，用 front 和 rear 分别指向队头和队尾元素，last 指向各层最右结点的位置。存放值大于 50 的结点结构为

```
typedef struct {int level, value, idx; } node; //元素所在层号、值和本层中的序号
node a[], s;
void ValueGT50(BiTree bt) //查各层中结点值大于 50 的结点个数，输出其值及序号
{
    if (bt!=null)
    {
        int front=0, last=1, rear=1, level=1, i=0, num=0; //num 记>50 的结点个数
        BiTree Q[]; Q[1]=bt; //根结点入队
        while (front<=last)
        {
            bt=Q[++front];
            if (bt->data>50) {s.level=level; s.idx=++i; s.value=bt->data; a[++num]=s;}
            if (bt->lchild!=null) Q[++rear]=bt->lchild; //左子女入队列
            if (bt->rchild!=null) Q[++rear]=bt->rchild; //右子女入队列
            if (front==last) {last=rear; level++; i=0;} //本层最后一个结点已处理完
        } //初始化下层：last 指向下层最右结点，层号加 1，下层>50 的序号初始为 0
    } //while
    for (i=1; i<=num; i++) //输出 data 域数值大于 50 的结点的层号、data 域的数值和序号
        printf("层号=%3d, 本层序号=%3d, 值=%3d", a[i].level, a[i].idx, a[i].value);
} //算法 ValueGT50 结束
```

```
34. PROC print(bt:BiTree; xy:integer)
    //将二叉树逆时针旋转 90 度打印，xy 是根结点基准坐标，调用时 xy=40
    IF bt<>NIL THEN [ print(bt↑.rchild, xy+5); //中序遍历右子树
                      writeln(bt->data:xy); //访问根结点
                      print(bt↑.lchild, xy+5); ] //中序遍历左子树
    ENDP;
```

```
35. BiTree creat() //生成并中序输出二叉排序树
{
    scanf("%c", &ch) //ch 是二叉排序树结点值的类型变量，假定是字符变量
    BiTree bst=null, f=null;
    while (ch!='#') // '#' 是输入结束标记
    {
        s=(BiTree)malloc(sizeof(BiNode)); //申请结点
        s->data=ch; s->lchild=s->rchild=null;
        if (bst==null) bst=s; //根结点
        else //查找插入结点
        {
            p=bst;
            while (p)
            {
                if (p->data>ch) p=p->lchild;
                else p=p->rchild;
            }
            if (p==null) p=bst;
            if (p->data>ch) p->lchild=s;
            else p->rchild=s;
        }
    }
    return bst;
}
```

```

        if (ch>p->data) {f=p; p=p->rchild;}           //沿右分枝查, f 是双亲
        else {f=p; p=p->lchild;}                     //沿左分枝查
        if(f->data<ch) f->rchild=s; else f->lchild=s;} //将 s 结点插入树中
        scanf( "%c" ,&ch); //读入下一数据
    } //while (ch!= '#' )
    return(bst); } //结束 creat
void InOrder(BiTree bst)      //bst 是二叉排序树, 中序遍历输出二叉排序树
{if(bst)
    {InOrder (bst->lchild); printf(bst->data); InOrder(bst->rchild); }
} //结束 InOrder

```

36. [题目分析] 二叉树结点 p 所对应子树的第一个后序遍历结点 q 的求法如下: 若 p 有左子树, 则 q 是 p 的左子树上最左下的叶子结点; 若 p 无左子树, 仅有右子树, 则 q 是 p 的右子树上最左下的叶子结点。

```

BiTree PostFirst(p)
{BiTree q=p;
 if (!q) return(null); else
 while(q->lchild || q->rchild); //找最左下的叶子结点
 if(q->lchild) q=q->lchild;    //优先沿左分枝向下去查“最左下”的叶子结点
 else q=q->rchild;            //沿右分枝去查“最左下”的叶子结点
 return(q);
}

```

[算法讨论] 题目“求 p 所对应子树的第一个后序遍历结点”, 蕴涵 p 是子树的根。若 p 是叶子结点, 求其后继要通过双亲。

37. (1) 由先序序列 $A[1..n]$ 和中序序列 $B[1..n]$, 可以确定一棵二叉树, 详见本章四第 38 题。

(2) void PreInCreat(ElemTypeA[], B[], int l1, h1, l2, h2)

//由二叉树前序序列 $A[1..n]$ 和中序序列 $B[1..n]$ 建立二叉树, $l1, h1$, 和 $l2, h2$ 分别为先序序列和
//中序序列第一和最后结点的下标. 初始调用时 $l1=l2=1, h1=h2=n$ 。

```

{typedef struct {int l1,h1,l2,h2; BiTree t; }node;
BiTree bt;
int top=0, i; node s[],p; //s 为栈, 容量足够大
bt=(BiTree)malloc(sizeof(BiNode)); //申请结点空间
p.l1=l1; p.h1=h1; p.l2=l2; p.h2=h2; p.t=bt; s[++top]=p; //初始化
while(top>0)
{p=s[top--]; bt=p.t; l1=p.l1; h1=p.h1; l2=p.l2 ;h2=p.h2; //取出栈顶数据
 for(i=l2; i<=h2; i++) if(B[i]==A[l1]) break; //到中序序列中查根结点的值
 bt->data=A[l1]; //A[l1]为根结点的值
 if(i==l2) bt->lchild=null; //bt 无左子树
 else //将建立左子树的数据入栈
 {bt->lchild=(BiTree)malloc(sizeof(BiNode)); p.t=bt->lchild;
  p.l1=l1+1; p.h1=l1+i-l2; p.l2=l2; p.h2=i-1; s[++top]=p; }
 if(i==h2) bt->rchild=null; //bt 无右子树
 else {bt->rchild=(BiTree)malloc(sizeof(BiNode)); p.t=bt->rchild;
  p.l1=l1+i-l2+1; p.h1=h1; p.l2=i+1; p.h2=h2; s[++top]=p; } //右子树数据入栈
} //while
} //结束 PreInCreat

```

(3) 当二叉树为单支树时, 栈深 n ; 当二叉树左右子树高相等时, 栈深 $\log n$ 。时间复杂度 $O(n)$ 。

38. [题目分析]知二叉树中序序列与后序序列，第 30 题以递归算法建立了二叉树，本题是非递归算法。

```
void InPostCreat (ElemType IN[], POST[], int l1, h1, l2, h2)
//由二叉树的中序序列 IN[] 和后序序列 POST[] 建立二叉树，l1, h1 和 l2, h2 分别是中序序列和
//后序序列第一和最后元素的下标，初始调用时，l1=l2=1, h1=h2=n。
{typedef struct {int l1, h1, l2, h2; BiTree t; }node;
node s[], p; //s 为栈，容量足够大
BiTree bt=(BiTree)malloc(sizeof(BiNode)); int top=0, i;
p.l1=l1; p.h1=h1; p.l2=l2; p.h2=h2; p.t=bt; s[++top]=p; //初始化
while(top>0)
{p=s[top--]; bt=p.t; l1=p.l1; h1=p.h1; l2=p.l2; h2=p.h2; //取出栈顶数据
for(i=l1; i<=h1; i++) if(IN[i]==POST[h2]) break; //在中序序列中查等于 POST[h2]的结点
bt->data=POST[h2]; //根结点的值
if(i==l1) bt->lchild=null; //bt 无左子树
else //将建立左子树的数据入栈
{bt->lchild=(BiTree)malloc(sizeof(BiNode)); p.t=bt->lchild;
p.l1=l1; p.h1=i-1; p.l2=l2; p.h2=l2+i-l1-1; s[++top]=p; }
if(i==h1) bt->rchild=null; //bt 无右子树
else {bt->rchild=(BiTree)malloc(sizeof(BiNode)); p.t=bt->rchild;
p.l1=i+1; p.h1=h1; p.l2=l2+i-l1; p.h2=h2-1; s[++top]=p; } //右子树数据入栈
} // while(top>0)
}结束 InPostCreat
```

39. BiTree Copy (BiTree t) //复制二叉树 t

```
{BiTree bt;
if (t==null) bt=null;
else {bt=(BiTree)malloc(sizeof(BiNode)); bt->data=t->data;
bt->lchild=Copy(t->lchild);
bt->rchild=Copy(t->rchild);
}
return(bt); } //结束 Copy
```

40. [题目分析]森林在先根次序遍历时，首先遍历第一棵子树的根，接着是第一棵子树的结点；之后是第二棵树，……，最后一棵树。本题中 $E[i]$ 是 $H[i]$ 所指结点的次数，次数就是结点的分支个数 B ，而分支数 B 与树的结点数 N 的关系是 $N=B+1$ （除根结点外，任何一个结点都有一个分支所指）。所以，从 $E[i]$ 的第一个单元开始，将值累加，当累加到第 i 个单元，其值正好等于 $i-1$ 时，就是第一棵树。接着，用相同方法，将其它树分开，进行到第 n 个单元，将所有树分开。例如，上面应用题第 47 题 (2) 的森林可按本题图示如下。从左往右将次数加到下标 8 ($=B+1$) 时，正好结束第一棵树。

i	1	2	3	4	5	6	7	8	9	10	11	12
H[i]	A	B	D	G	L	H	E	I	C	F	J	K
E[i]	3	2	2	0	0	0	0	0	0	1	0	0

void Forest (ElemType H[], int E[], int, n)

```
// H[i] 是森林 F 在先根次序下结点的地址排列，E[i] 是 H[i] 所指结点的次数，本算法计算森林
//F 的树形个数，并计算森林 F 的最后一个树形的根结点地址
{int i=1, sum=0, j=0, m=0; //sum 记一棵树的分支数，j 记树的棵数，m 记一棵树的结点数
int tree[]; //tree 记每棵树先序遍历最后一个结点在 H[i] 中的地址
while (i<=n) //n 是森林中结点数，题目已给出
{sum+=E[i]; m++;
```

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967


```

if (sum+1==m && i<=n) //记树先序最后结点的地址，为下步初始化
    {sum=0; m=0; tree[++j]=i;}
i++;
} //while
if (j==1) return (1);    //只有一棵树时，第一个结点是根
else return (tree[j-1]+1)
} //forest

```

41. [题目分析] 叶子结点只有在遍历中才能知道，这里使用中序递归遍历。设置前驱结点指针 pre，初始为空。第一个叶子结点由指针 head 指向，遍历到叶子结点时，就将它前驱的 rchild 指针指向它，最后叶子结点的 rchild 为空。

```

LinkedList head, pre=null; //全局变量
LinkedList InOrder(BiTree bt)
    //中序遍历二叉树 bt，将叶子结点从左到右链成一个单链表，表头指针为 head
    {if(bt) {InOrder(bt->lchild); //中序遍历左子树
        if(bt->lchild==null && bt->rchild==null) //叶子结点
            if(pre==null) {head=bt; pre=bt;} //处理第一个叶子结点
            else {pre->rchild=bt; pre=bt;} //将叶子结点链入链表
        InOrder(bt->rchild); //中序遍历右子树
        pre->rchild=null; //设置链表尾
    }
    return(head); } //InOrder

```

时间复杂度为 $O(n)$ ，辅助变量使用 head 和 pre，栈空间复杂度 $O(n)$

42. 层次遍历，参见上面算法第 33 题。中序遍历序列和后序序列可确定一棵二叉树，详见上面应用题第 40 题。先序序列和后序序列无法确定一棵二叉树，因为无法将二叉树分成左右子树。如先序序列 AB 和后序序列 BA，A 是根，但 B 可以是左子女，也可以是右子女。

43. [题目分析] 此树看作度为 2 的有序树，先将根结点入队列。当队列不空，重复以下动作：结点出队；若结点有两个子女，先将第二（右）子女入队列，并转向第一子女；若结点有一个子女，则入队列；如此下去，直到碰到叶子结点或只有一个子女的结点，再重复上述动作，直至队列为空。定义树结构如下：

```

typedef struct node
{ElemType data; struct node *firstchild, *secondchild;} *Tree;
void TreeTravers(Tree t) //按字母顺序输出树中各结点的值
{Tree p, Q[]; //Q 为队列，元素是树结点指针，容量是够大
if (t)
    {QueueInit(Q); QueueIn(Q, t); //根结点入队
    while(!QueueEmpty(Q))
        {p=QueueOut(Q); //出队
        while (p->firstchild && p->secondchild) //当有双子女时
            {QueueIn(Q, p->secondchild); printf(p->data); //访问结点
            p=p->firstchild;} //沿第一子女向下
        if (p->firstchild) {printf(p->data); QueueIn(Q, p->firstchild)} //一个子女树入队
        if (!p->firstchild && !p->secondchild) printf(p->data); //访问叶子结点
        } //while(!QueueEmpty(Q))
    } //if
} //算法结束

```

44. [题目分析] 由定义，结点的平衡因子 bf 等于结点的左子树高度与右子树高度之差，设计一遍历算法，

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

在遍历结点时，求结点的左子树和右子树的高度，然后得到结点的平衡因子。

```
int Height(BiTree bt)//求二叉树 bt 的深度
{int hl,hr;
  if (bt==null) return(0);
  else {hl=Height(bt->lchild); hr=Height(bt->rchild);
        if(hl>hr) return (hl+1); else return(hr+1);
      } }// Height
void Balance(BiTree bt)
//计算二叉树 bt 各结点的平衡因子
{if (bt)
  {Balance(bt->lchild); //后序遍历左子树
   Balance(bt->rchild); //后序遍历右子树
   hl=Height(bt->lchild); hr=Height(bt->rchild); //求左右子树的高度
   bt->bf=hl-hr; //结点的平衡因子 bf
  } } //算法结束
```

45. [题目分析] 本题应采用层次遍历方式。若树不空，则二叉树根结点入队，然后当队列不空且队列长不超过 n ，重复如下操作：出队，若出队元素不为空，则记住其下标，且将其左右子女入队列；若出队元素为空，当作虚结点，也将其“虚子女”入队列。为节省空间，就用树 T 的顺序存储结构 $A[1..n]$ 作队列，队头指针 $front$ ，队尾指针 $rear$ ，元素最大下标 $last$ 。

```
void Traverse(BiTree bt ,int n)
// 求二叉树 bt 的顺序存储结构 A[1..n]，下标超过 n 报错，给出实际的最大下标
{BiTree A[], p;
  if(bt!=null)
  {int front=0, rear=1, last=1; A[1]=bt;
   while(front<=rear)
   {p=A[++front]; if(p) last=front; // 出队;用 last 记住最后一个结点的下标
    rear=2*front; //计算结点（包括虚结点）“左子女”下标
    if (p) //二叉树的实际结点
    {if(rear>n) printf(“%c 结点无左子女”); else A[rear]=p->lchild;
     if(rear+1>n) printf(“%c 结点无右子女”); else A[rear+1]=p->rchild;
    }
    else //p 是虚结点
    { if(rear<=n) A[rear]=null; if(rear+1<=n) A[rear+1]=null; }
   } // while(front<=rear)
   printf(“实际的最大下标是%d”, last);
  } //if(bt!=null) } //Traverse
```

46. [题目分析] 两棵空二叉树或仅有根结点的二叉树相似；对非空二叉树，可判左右子树是否相似，采用递归算法。

```
int Similar(BiTree p,q) //判断二叉树 p 和 q 是否相似
{if(p==null && q==null) return (1);
  else if(!p && q || p && !q) return (0);
  else return(Similar(p->lchild, q->lchild) && Similar(p->rchild, q->rchild))
} //结束 Similar
```

47. [题目分析] 根据树的双亲表示法创建树的孩子兄弟链表表示法，首先给出根结点在双亲表示法中的下标，但找根结点的子女要遍历双亲表示法的整个静态链表，根结点的第一个子女是孩子兄弟表示法中的孩

子，其它子女结点作兄弟。对双亲表示法中的任一结点，均递归建立其孩子兄弟链表子树。

```
CSTree PtreeToCstree (PTree pt, int root)
//本算法将双亲表示法的树 pt 转为孩子兄弟链表表示的树，root 是根结点在双亲表示法中的下标。
{CSTree child, sibling; int firstchild;
CSTree cst=(CSTree)malloc(sizeof(CSNode)); //申请结点空间
cst->data=pt.nodes[root].data; cst->firstchild=null; cst->nextsibling=null; //根结点
firstchild=1;
for(i=1; i<=pt.n; i++) //查找 root 的孩子
    if(pt.nodes[i].parent==root)
        {child=PtreeToCstree(pt, i);
        if(firstchild) {cst->firstchild=child; firstchild=0; sibling=cst->firstchild;}
        else //child 不是 root 的第一个孩子，作兄弟处理
            {sibling->nextsibling=child; sibling=sibling->nextsibling;}
        } //if
} //end for
return cst; } //结束 PtreeToCstree
```

48. [题目分析] 删除以元素值 x 为根的子树，只要能删除其左右子树，就可以释放值为 x 的根结点，因此宜采用后序遍历。删除值为 x 结点，意味着应将其父结点的左(右)子女指针置空，用层次遍历易于找到某结点的父结点。本题要求删除树中每一个元素值为 x 的结点的子树，因此要遍历完整棵二叉树。

```
void DeleteXTree(BiTree bt) //删除以 bt 为根的子树
{DeleteXTree(bt->lchild); DeleteXTree(bt->rchild); //删除 bt 的左子树、右子树
free(bt); } // DeleteXTree //释放被删结点所占的存储空间

void Search(B:Tree bt, ElemType x)
//在二叉树上查找所有以  $x$  为元素值的结点，并删除以其为根的子树
{BiTree Q[]; //Q 是存放二叉树结点指针的队列，容量足够大
if(bt)
    {if(bt->data==x) {DeleteXTree(bt); exit(0);} //若根结点的值为  $x$ ，则删除整棵树
    QueueInit(Q); QueueIn(Q, bt);
    while(!QueueEmpty(Q))
        {p=QueueOut(Q);
        if(p->lchild) // 若左子女非空
            if(p->lchild->data==x) //左子女结点值为  $x$ ，应删除当前结点的左子树
                {DeleteXTree(p->lchild); p->lchild=null;} //父结点的左子女置空
            else Enqueue(Q, p->lchild); // 左子女入队列
        if(p->rchild) // 若右子女非空
            if(p->rchild->data==x) //右子女结点值为  $x$ ，应删除当前结点的右子树
                {DeleteXTree(p->rchild); p->rchild=null;} //父结点的右子女置空
            else Enqueue(Q, p->rchild); // 右子女入队列
        } //while
    } //if(bt) } //search
```

49. [题目分析] 在二叉树上建立三叉链表，若二叉树已用二叉链表表示，则可象 48 题那样，给每个结点加上指向双亲的指针（根结点的双亲指针为空）。至于删除元素值为 x 的结点以及以 x 为根的子树，与 48 题完全一样，请参照 48 题。下面给出建立用三叉链表表示的二叉树的算法。二叉树按完全二叉树格式输入，对非完全二叉树，要补上“虚结点”。按完全二叉树双亲和子女存储下标间的关系，完成双亲和子女间指针的链接。‘#’是输入结束标志，‘\$’是虚结点标志。

```

BiTree creat() //生成三叉链表的二叉树 (题目给出 PASCAL 定义, 下面的用类 C 书写)
{BiTree p, Q[], root; //Q 是二叉树结点指针的一维数组, 容量足够大
  char ch; int rear=0; //一维数组最后元素的下标
  scanf(&ch);
  while(ch!= '#')
  {p=null;
   if(ch!= '$') {p=(BiTree)malloc(sizeof(nodetp));
                 p->data=ch; p->lchild=p->rchild=null; }
   Q[++rear]=p; //元素或虚结点
   if(p) {if(rear==1) {root=p; root->parent=null; } //根结点
          else {Q[rear]->parent=Q[rear/2]; //双亲结点和子女结点用指针链上
                if (rear%2==0) Q[rear/2]->lchild=Q[rear]; else Q[rear/2]->rchild=Q[rear];
               }
          scanf("%c", &ch);
        } //while
  return(root); } //结束 creat

50. int BTLC(BiTree T, int *c) //对二叉树 T 的结点计数
    {if(T)
     {*c++;
      BTLC(T->lchild, &c); //统计左子树结点
      BTLC(T->rchild, &c); //统计右子树结点
     } } //结束 Count, 调用时 *c=0

51. int Count(CSTree t) //统计以孩子兄弟链表表示的树的叶子结点个数
    {if(t==null) return(0);
     else if(t->firstlchild==null) //左子女为空, 结点必为叶子
       return(1+Count(t->nextsibling)); // (叶子) + 兄弟子树上的叶子结点
     else return(Count(t->firstchild)+Count(t->nextsibling)); //子女子树+兄弟子树
    } //Count

52. void Count(BiTree bt, int *n0, *n) //统计二叉树 bt 上叶子结点数 n0 和非叶子结点数 n
    {if(bt)
     {if (bt->lchild==null && bt->rchild==null) *n0++; //叶子结点
      else *n++; //非叶结点
      Count(bt->lchild, &n0, &n);
      Count(bt->rchild, &n0, &n);
     } } //结束 Count

53. int Count (BiTree bt) // 非递归遍历求二叉树上的叶子结点个数
    {int num=0;
     BiTree s[]; //s 是栈, 栈中元素是二叉树结点指针, 栈容量足够大
     while(bt!=null || top>0)
     {while(bt!=null) {push(s, bt); bt=bt->lchild; } //沿左分支向下
      if(!StackEmpty(s))
        {bt=pop(s); if(bt->lchild==null && bt->rchild==null) num++; //叶子结点
         bt=bt->rchild;
        }
     }
     return(num);
    }

```

```
    } //结束 Count
```

54. [题目分析]对二叉树的某层上的结点进行运算，采用队列结构按层次遍历最适宜。

```
int LeafKlevel(BiTree bt, int k) //求二叉树 bt 的第 k(k>1) 层上叶子结点个数
{
    if(bt==null || k<1) return(0);
    BiTree p=bt, Q[];           //Q 是队列，元素是二叉树结点指针，容量足够大
    int front=0, rear=1, leaf=0; //front 和 rear 是队头和队尾指针，leaf 是叶子结点数
    int last=1, level=1; Q[1]=p; //last 是二叉树同层最右结点的指针，level 是二叉树的层数
    while(front<=rear)
    {
        p=Q[++front];
        if(level==k && !p->lchild && !p->rchild) leaf++; //叶子结点
        if(p->lchild) Q[++rear]=p->lchild;           //左子女入队
        if(p->rchild) Q[++rear]=p->rchild;           //右子女入队
        if(front==last) {level++;                  //二叉树同层最右结点已处理，层数增 1
                        last=rear; }                //last 移到指向下层最右一元素
    }
    if(level>k) return (leaf);                      //层数大于 k 后退出运行
} //while } //结束 LeafKLevel
```

55. [题目分析]按题目要求，每个结点的编号大于其左右孩子的编号，结点左孩子的编号小于右孩子的编号。由此看出，从小到大按“左右根”顺序，这正是后序遍历的顺序，故对二叉树进行后序遍历，在遍历中对结点进行编号，现将二叉树结点结构定义如下：

```
typedef struct node
{
    ElemType data; int num; struct node *lchild, *rchild; } Bnode, *Btree;

void PostOrder(Btree t)
//对二叉树从 1 开始编号，结点编号大于其左右子女结点编号，结点的左子女编号小于其右子女编号
{
    typedef struct {Btree t; int tag; } node;
    Btree p=t; node sn, s[]; //s 为栈，容量足够大
    int k=0, top=0; //k 为结点编号，top 为栈顶指针
    while(p!=null || top>0)
    {
        while(p) {sn.t=p; sn.tag=0; s[++top]=sn; p=p->lchild;} //沿左分枝向下
        while(top>0 && s[top].tag==1) {sn=s[top--]; sn.t->num=++k;} //左右孩子已遍历，结点赋编号
        if (top>0) {s[top].tag=1; p=s[top].t->rchild;}
    } //while(p!=null || top>0)
} //结束 PostOrder
```

56. [题目分析]非递归算法参考上面第 37 题。下面给出递归算法。

```
void PreInCreat(BiTree root, ElemType pre[], in[], int l1, h1, l2, h2)
//根据二叉树前序序列 pre 和中序序列 in 建立二叉树。l1, h1, l2, h2 是序列第一和最后元素下标。
{
    root=(BiTree)malloc(sizeof(BiNode)); //申请结点
    root->data=pre[l1]; //pre[l1] 是根
    for(i=l2; i<=h2; i++) if(in[i]==pre[l1]) break; //在中序序列中，根结点将树分成左右子树
    if(i==l2) root->lchild=null; //无左子树
    else PreInCreat(root->lchild, pre, in, l1+1, l1+(i-l2), l2, i-1); //递归建立左子树
    if(i==h2) root->rchild=null; //无右子树
    else PreInCreat((root->rchild), pre, in, l1+(i-l2)+1, h1, i+1, h2) //递归建立右子树
} //结束 PreInCreat
```

57 (1) 略 (2) 根据中序和后序序列，建立二叉树的递归算法见上面第 30 题，非递归算法见第 38 题。

58. [题目分析]采用后序非递归遍历二叉树，栈中保留从根结点到当前结点的路径上的所有结点。

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

```

void PrintPath(BiTree bt, p) //打印从根结点 bt 到结点 p 之间路径上的所有结点
{
    BiTree q=bt, s[]; //s 是元素为二叉树结点指针的栈，容量足够大
    int top=0; tag[]; //tag 是数组，元素值为 0 或 1，访问左、右子树的标志，tag 和 s 同步
    if (q==p) {printf(q->data); return;} //根结点就是所找结点
    while(q!=null || top>0)
    {
        while(q!=null) //左子女入栈，并置标记
        {
            if (q==p) //找到结点 p，栈中元素均为结点 p 的祖先
            {
                printf("从根结点到 p 结点的路径为\n");
                for(i=1; i<=top; i++) printf(s[i]->data); printf(q->data); return;
            }
            else {s[++top]=q; tag[top]=0; q=q->lchild;} //沿左分支向下
        }
        while(top>0 && tag[top]==1) top--; //本题不要求输出遍历序列，这里只退栈
        if (top>0) {q=s[top]; q=q->rchild; tag[top]=1;} //沿右分支向下
    } //while(q!=null || top>0)
} //结束算法 PrintPath

```

59. [题目分析]上题是打印从根结点到某结点 p 的路径上所有祖先结点，本题是打印由根结点到叶子结点的所有路径。只要在上题基础上把 q 是否等于 p 的判断改为 q 是否是叶子结点即可。其语句段如下：

```

if(q->lchild==null&&q->rchild==null) //q 为叶子结点
{
    printf("从根结点到 p 结点的路径为\n");
    for(i=1; i<=top; i++) printf(s[i]->data); //输出从根到叶子路径上，叶子 q 的所有祖先
    printf(q->data);
}

```

60. [题目分析]因为后序遍历栈中保留当前结点的祖先的信息，用一变量保存栈的最高栈顶指针，每当退栈时，栈顶指针高于保存最高栈顶指针的值时，则将该栈倒入辅助栈中，辅助栈始终保存最长路径长度上的结点，直至后序遍历完毕，则辅助栈中内容即为所求。

```

void LongestPath(BiTree bt) //求二叉树中的第一条最长路径长度
{
    BiTree p=bt, l[], s[]; //l, s 是栈，元素是二叉树结点指针，l 中保留当前最长路径中的结点
    int i, top=0, tag[], longest=0;
    while(p || top>0)
    {
        while(p) {s[++top]=p; tag[top]=0; p=p->Lc;} //沿左分枝向下
        if(tag[top]==1) //当前结点的右分枝已遍历
        {
            if(!s[top]->Lc && !s[top]->Rc) //只有到叶子结点时，才查看路径长度
            {
                if(top>longest) {for(i=1; i<=top; i++) l[i]=s[i]; longest=top; top--;}
            } //保留当前最长路径到 l 栈，记住最高栈顶指针，退栈
        }
        else if(top>0) {tag[top]=1; p=s[top].Rc;} //沿右子分枝向下
    } //while(p!=null || top>0)
} //结束 LongestPath

```

61. [题目分析]在线索二叉树上插入结点，破坏了与被插入结点的线索。因此，插入结点时，必须修复线索。在结点 y 的右侧插入结点 x，因为是后序线索树，要区分结点 y 有无左子树的情况。

```

void TreeInsert(BiTree t, y, x) //在二叉树 t 的结点 y 的右侧，插入结点 x
{
    if(y->ltag==0) //y 有左子女
    {
        p=y->lchild; if (p->rtag==1) p->rchild=x; //x 是 y 的左子女的后序后继
        x->ltag=1; x->lchild=p; //x 的左线索是 y 的左子女
    }
    else //y 无左子女

```

```

{x->ltag=1; x->lchild=y->lchild; //y 的左线索成为 x 的左线索
if(y->lchild->rtag==1) //若 y 的后序前驱的右标记为 1
    y->lchild->rchild=x; //则将 y 的后序前驱的后继改为 x
}
x->rtag=1; x->rchild=y; y->rtag=0; y->rchild=x; //x 作 y 的右子树
} //结束 TreeInsert

```

62. [题目分析]在中序全线索化 T 树的 P 结点上, 插入以 X 为根的中序全线索化二叉树, 要对 X 有无左右子树进行讨论, 并修改 X 左(右)子树上最左结点和最右结点的线索。在中序线索树上查找某结点 p 的前驱的规律是: 若 $p \rightarrow ltag=1$, 则 $p \rightarrow lchild$ 就指向前驱, 否则, 其前驱是 p 的左子树上按中序遍历的最后一个结点; 查找某结点 p 的后继的规律是: 若 $p \rightarrow rtag=1$, 则 $p \rightarrow rchild$ 就指向后继, 否则, 其后继是 p 的右子树上按中序遍历的第一个结点。

```

int TreeThrInsert(BiThrTree T, P, X)
//在中序全线索二叉树 T 的结点 P 上, 插入以 X 为根的中序全线索二叉树, 返回插入结果信息。
{if(P->ltag==0 && P->rtag==0) {printf("P 有左右子女, 插入失败\n"); return(0); }
if(P->ltag==0) //P 有左子女, 将 X 插为 P 的右子女
{if(X->ltag==1) X->lchild=P; //若 X 无左子树, X 的左线索(前驱)是 P
else //寻找 X 的左子树上最左(下)边的结点
{q=X->lchild;
while(q->ltag==0) q=q->lchild;
q->lchild=P;
}
if(X->rtag==1) //修改 X 的右线索
X->rchild=P->rchild; //将 P 的右线索改为 X 的右线索
else //找 X 右子树最右面的结点
{q=X->rchild; while(q->rtag==0) q=q->rchild;
q->rchild=P->rchild;
}
P->rtag=0; P->rchild=X; //将 X 作为 P 的右子树
} //结束将 X 插入为 P 的右子树
else //P 有右子女, 将 X 插入为 P 的左子女
{if(X->ltag==1) //X 无左子女
X->lchild=P->lchild; //将 P 的左线索改为 X 的左线索
else //X 有左子女, 找 X 左子树上最左边的结点
{q=X->lchild;
while(q->ltag==0) q=q->lchild;
q->lchild=P->lchild;
}
if(X->rtag==1) X->rchild=P; //若 X 无右子树, 则 X 的右线索是 P
else //X 有右子树, 查其右子树中最右边的结点, 将该结点的后继修改为 P
{q=X->rchild;
while(q->rtag==0) q=q->rchild;
q->rchild=P;
}
P->ltag=0; //最后将 P 的左标记置 0
P->lchild=X; //P 的左子女链接到 X
}

```

```
    } //结束将 X 插入为 P 的左子女
} //结束 Tree ThrInser
```

63. [题目分析]在中序线索树中，非递归查找数据域为 A 的结点（设该结点存在，其指针为 P）并将数据域为 x 的 Q 结点插入到左子树中。若 P 无左子女，则 Q 成为 P 的左子女，原 P 的左线索成为 Q 的左线索，Q 的右线索为 P；若 P 有左子树，设 P 左子树中最右结点的右线索是结点 Q，结点 Q 的右线索是 P。

```
void InThrInsert(BiThrTree T, Q; ElemType A)
//在中序线索二叉树 T 中，查找其数据域为 A 的结点，并在该结点的左子树上插入结点 Q
{BiThrTree P=T;
while(P)
{while(P->LT==0 && P->data!=A) P=P->LL; //沿左子树向下
if (P->data==A) break; //找到数据域为 A 的结点，退出循环
while(P->RT==1) P=P->RL; //还没找到数据域为 A 的结点沿右线索找后继
P=P->RL; //沿右子树向下
}
if(P->LT==1) //P 没有左子树，Q 结点插入作 P 的左子女
{Q->LL=P->LL; Q->LT=1; //将 P 的左线索作为 Q 的左线索
}
else //P 有左子树，应修改 P 的左子树最右结点的线索
{Q->LL=P->LL; Q->LT=0; //Q 成为 P 的左子女
s=Q->LL; //s 指向原 P 的左子女
while(s->RT==0) s=s->RL; //查找 P 的左子树最右边的结点
s->RL=Q; //原 P 左子树上最右结点的右线索是新插入结点 Q
}
P->LT=0; P->LL=Q; //修改 P 的标记和指针
Q->RT=1; Q->RL=P; //将 Q 链为 P 的左子女，其中序后继是 P;
} //结束 InThrInsert
```

64. [题目分析]“双链”就利用二叉树结点的左右指针，重新定义左指针为指向前驱的指针，右指针是指向后继的指针，链表在遍历中建立，下面采用中序遍历二叉树。

```
BiTree head=null, pre; //全局变量链表头指针 head, pre
void CreatLeafList(BiTree bt) //将 BiTree 树中所有叶子结点链成带头结点的双链表，
{if (bt) //若 bt 不空
{CreatLeafList (bt->lchild); //中序遍历左子树
if(bt->lchild==null && bt->rchild==null) //叶子结点
if(head==null) //第一个叶子结点
{head=(BiTree)malloc(sizeof(BiNode)); //生成头结点
head->lchild=null; head->rchild=bt; //头结点的左链为空，右链指向第一个叶子结点
bt->lchild=head; pre=bt; //第一个叶子结点左链指向头结点，pre 指向当前叶子结点
}
else //已不是第一个叶子结点
{pre->rchild=bt; bt->lchild=pre; pre=bt;} //当前叶子结点链入双链表
CreatLeafList (bt->rchild); //中序遍历右子树
pre->rchild=null; //最后一个叶子结点的右链置空（链表结束标记）
} //if(bt) } //结束 CreatLeafList;
```

65. [题目分析]求中序全线索树任意结点 p 的前序后继，其规则如下：若 p 有左子女，则左子女就是其前序后继；若 p 无左子女而有右子女，则 p 的右子女就是 p 的前序后继；若 p 无左右子女，这时沿 p 的右

线索往上，直到 p 的右标志为 0（非线索），这时若 p 的右子女为空，则表示这是中序遍历最后一个结点，故指定结点无前序后继，否则，该结点就是指定结点的前序后继。程序段如下：

```
if(p->ltag==0 && p->lchild!=null) return(p->lchild); //p 的左子女是 p 的前序后继
else if(p->rtag==0 && p->rchild!=null) return(p->rchild); //p 右子女是其前序后继
else //p 无左右子女，应沿右线索向上（找其前序后继），直到某结点右标记为 0
{while (p->rtag==1) p=p->rchild;
  if (p->rchild) return(p->rchild);else return(null); } //指定结点的前序后继
```

[算法讨论] 请注意题目“中序序列第一结点的左标志和最后结点的右标志皆为 0（非线索），对应指针皆为空”的说明。若无这一说明，只要结点的左标记为 0，其左子女就是其前序后继。最后，当 p 无子女，沿右线索向上找其前序后继时，若最后结点的右标志为 0，但对对应指针为空，p 也无前序后继。

66. [题目分析] 不借助辅助堆栈实现中序遍历，必须解决如何查找后继的问题。使用线索树就行。为此，将结点结构修改为(ltag, lchild, data, rchild, rtag)。各字段的含义在上面已多次使用，不再介绍。设二叉树已中序线索化。下面首先编写一查中序后继的函数，接着是中序遍历的非递归算法。

```
BiTree After(BiThrTree t) //查中序线索二叉树上结点 t 的后继
{if (t->rtag==1) return(t->rchild);
 p=t->rchild;
 while(p->ltag==0) p=p->lchild; //p 右子树中最左下的结点是 p 的中序后继
 return(p); } //if
void InOrder(BiThrTree bt)
//非递归中序遍历带头结点的中序线索二叉树 bt
{p=bt->lchild; //p 指向原二叉树的根结点
if (p!=bt) //二叉树非空
{while (p->ltag==0) p=p->lchild; //找中序遍历的第一个结点
 while (p!=bt) //没回到头结点，就一直找后继并遍历
 {visit(*p); p=After(p); }
 } //if }结束算法 InOrder
```

67. [题目分析] 在中序穿线树中找结点的双亲，最简单情况是顺线索就可找到。例如，结点的左子女的右线索和右子女的左线索都指向双亲。但对于有左右子女的结点来说，则要利用中序穿线树中线索“向上”指向祖先的特点：若结点 p 是结点 q 右子树中中序遍历最左下的结点，p 的左线索指向 q；若结点 p 是结点 q 左子树中中序遍历最右下的结点，p 的右线索指向是 q。反过来，通过祖先找子女就容易了。另外，若结点 q 的后继是中序穿线树的头结点，则应特殊考虑。

```
void FFA(BiThrTree t, p, q) //在中序穿线树 t 上，求结点 p 的双亲结点 q
{q=p; //暂存
 while(q->RTAG==0) q=q->RLINK; //找 p 的中序最右下的结点
 q=q->RLINK; //顺右线索找到 q 的后继（p 的祖先结点）
 if (q==t) q=t->LLINK; //若后继是头结点，则转到根结点
 if (q==p) {printf(“根结点无双亲\n”); return; }
 if (q->LLINK==p) return(q); else q=q->LLINK; //准备到左子树中找 p
 while (q->RLINK!=p) q=q->RLINK; return(q); } //找最右结点的过程中回找到 p
} //结束 FFA
```

[算法讨论] 本题也可以先求结点 p 最左下结点的前驱线索，请读者自己写出算法。

68. [题目分析] 带头结点的中序线索树，其头结点的 lchild 指向二叉树的根结点，头结点的 rchild 域指向中序遍历的最后一个结点。而二叉树按中序遍历的第一个结点的 lchild 和最后一个结点的 rchild 指向头结点。故从头结点找到根结点后，顺“后继”访问二叉树。在中序线索树中，找前序的后继，已在第 65 题进行了详细的讨论，这里不再赘述。中序线索树在上面的“四、应用题”已画过多个，这里也不重复。

```

void PreorderInThreat (BiThrTree tbt)
    //前序遍历一中序全线索二叉树 tbt, tbt 是头结点指针
{bt=tbt->lchild;
    while(bt)
        {while(bt->ltag==0) {printf(bt->data); bt=bt->lchild;} //沿左分枝向下
          printf(bt->data);    //遍历其左标志为 1 的结点, 准备右转
          while(bt->rtag==1 && bt->rchild!=tbt) bt=bt->rchild; //沿右链向上
          if (bt->rchild!=tbt) bt=bt->rchild; //沿右分枝向下
        }
    } //结束 PreorderInThreat
    时间复杂度 O(n)。

```

69. [题目分析]线索化是在遍历中完成的, 因此, 对于二叉树进行前序、中序、后序遍历, 在“访问根结点”处进行加线索的改造, 就可实现前序, 中序和后序的线索化

```

BiThrTree pre=null; //设置前驱
void PreOrderThreat (BiThrTree BT)
    //对以线索链表为存储结构的二叉树 BT 进行前序线索化
{if (BT!=null)
    {if (BT->lchild==null) {BT->ltag=1; BT->lchild=pre;} //设置左线索
      if (pre!=null && pre->rtag==1) pre->rchild=BT;    //设置前驱的右线索;
      if (BT->rchild==null) BT->rtag=1;                //为建立右链作准备
      pre=BT; //前驱后移
      if (BT->ltag==0) PreOrderThreat (BT->lchild);    //左子树前序线索化
      PreOrderThreat (BT->rchild);                    //右子树前序线索化
    } //if (BT!=null) } 结束 PreOrderThreat

```

70. BiThrTree pre==null;

```

void InOrderThreat (BiThrTree T) //对二叉树进行中序线索化
{if (T)
    {InOrderThreat (T->lchild); //左子树中序线索化
      if (T->lchild==null) {T->ltag=1; T->lchild=pre;} //左线索为 pre;
      if (pre!=null && pre->rtag==1) pre->rchild=T;    //给前驱加后继线索
      if (T->rchild==null) T->rtag=1;                //置右标记, 为右线索作准备
      pre=BT; //前驱指针后移
      InOrderThreat (T->rchild);                    //右子树中序线索化
    } } //结束 InOrderThreat

```

71. void InOrderThreat (BiThrTree thrt)

```

    //thrt 是指向中序全线索化头结点的指针, 本算法中序遍历该二叉树
    {p=thrt->lchild; //p 指向二叉树的根结点, 当二叉树为空时, p 指向 thrt
      while(p!=thrt)
          {while (p->ltag==0) p=p->lchild; //沿左子女向下
            visit(*p); //访问左子树为空的结点
            while(p->rtag==1 && p->rchild!=thrt) {p=p->rchild; visit(*p);} //沿右线索访问后继结点
            p=p->rchild; //转向右子树
          } } //结束 InOrderThreat

```

72. [题目分析]若使新插入的叶子结点 S 成 T 右子树中序序列的第一个结点, 则应在 T 的右子树中最左面的结点 (设为 p) 处插入, 使 S 成为结点 p 的左子女。则 S 的前驱是 T, 后继是 p。

```
void ThrTreeInsert (BiThrTree T, S)
```

//在中序线索二叉树 T 的右子树上插入结点 S，使 S 成为 T 右子树中序遍历第一个结点

```
{p=T->rchild; //用 p 去指向 T 的右子树中最左面的结点
```

```
while(p->ltag==0) p=p->lchild;
```

```
S->ltag=1;S->rtag=1; //S 是叶子，其左右标记均为 1
```

```
S->lchild=T;S->rchild=p; //S 的前驱是根结点 T，后继是结点 p
```

```
p->lchild=S;p->ltag=0; //将 p 的左子女指向 S，并修改左标志为 0
```

```
} //结束 ThrTreeInsert
```

73. BiThrTree InOrder (BiThrTree T, ElemType x)

//先在带头结点的中序线索二叉树 T 中查找给定值为 x 的结点，假定值为 x 的结点存在

```
{p=T->lchild; //设 p 指向二叉树的根结点
```

```
while(p!=T)
```

```
{while(p->ltag==0 && p->data!=x) p=p->lc;
```

```
if(p->data==x) return(p);
```

```
while(p->rtag==1 && p->rc!=T) {p=p->rc; if(p->data==x) return(p);}
```

```
p=p->rc; }
```

```
} //结束 InOrder
```

BiThrTree AfterXNode (BiThrTree T) //在中序线索二叉树 T 中，求给定值为 x 的结点的后继结点

```
{BiThrTree p=InOrder(T, x); //首先在 T 树上查找给定值为 x 的结点，由 p 指向
```

```
if(p->rtag==1) return(p->rc); //若 p 的左标志为 1，则 p 的 rc 指针指向其后继
```

```
else {q=p->rc; while(q->ltag==0) q=q->lc; return(q); }
```

//结点 p 的右子树中最左面的结点是结点 p 的中序后继

```
} } //结束 AfterXNode
```

74. [题目分析]后序遍历是“左-右-根”，因此，若结点有右子女，则右子女是其前驱，否则，左子女（或左线索）指向其后序前驱。

BiThrTree PostSucc (BiThrTree T, p) //在后序线索二叉树 T 中，查找指定结点 p 的直接前驱 q

```
{if(p->rtag==0) q=p->rchild; //若 p 有右子女，则右子女为其前驱
```

```
else q=p->lchild; //若 p 无右子女，左子女或左线索就是 p 的后序前驱
```

```
return (q);
```

```
} //结束 PostSucc
```

75. BiThrTree InSucc (BiThrTree T, p) //在对称序穿线树 T 中，查找给定结点 p 的中序后继

```
{if (p->rtag==1) q=p->rchild; //若 p 的右标志为 1，用其右指针指向后继
```

```
else {q=p->rchild; while(q->ltag==0) q=q->lchild; } //p 的后继为其右子树中最左下的结点
```

```
return (q);
```

```
} //结束 InSucc
```

76. [题目分析]在后序序列中，若结点 p 有右子女，则右子女是其前驱，若无右子女而有左子女，则左子女是其前驱。若结点 p 左右子女均无，设其中序左线索指向某祖先结点 f (p 是 f 右子树中按中序遍历的第一个结点)，若 f 有左子女，则其左子女是结点 p 在后序下的前驱；若 f 无左子女，则顺其前驱找双亲的双亲，一直继续到双亲有左子女（这时左子女是 p 的前驱）。还有一种情况，若 p 是中序遍历的第一个结点，结点 p 在中序和后序下均无前驱。

BiThrTree InPostPre (BiThrTree t, p)

//在中序线索二叉树 t 中，求指定结点 p 在后序下的前驱结点 q

```
{BiThrTree q;
```

```
if (p->rtag==0) q=p->rchild; //若 p 有右子女，则右子女是其后序前驱
```

```
else if (p->ltag==0) q=p->lchild; //若 p 无右子女而有左子女，左子女是其后序前驱。
```

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

```

else if(p->lchild==null) q=null; //p 是中序序列第一结点, 无后序前驱
else //顺左线索向上找 p 的祖先, 若存在, 再找祖先的左子女
{while(p->ltag==1 && p->lchild!=null) p=p->lchild;
  if(p->ltag==0) q=p->lchild; //p 结点的祖先的左子女是其后序前驱
  else q=null; //仅右单枝树 (p 是叶子), 已上到根结点, p 结点无后序前驱
}

return(q); } //结束 InPostPre

```

77. [题目分析]在中序穿线二叉树中, 求某结点 p 的后序后继 q , 需要知道 p 的双亲结点 f 的信息。(中序线索树的性质: 若 p 是 f 的左子女, 则 f 是 p 的最右子孙的右线索; 若 p 是 f 的右子女, 则 f 是 p 的最左子孙的左线索。)找到 f 后, 若 p 是 f 的右子女, 则 p 的后继是 f ; 若 p 是 f 的左子女, 且 f 无右子女, 则 p 的后继也是 f ; 若 p 是 f 的左子女, 且 f 有右子女, 则 f 的右子树中最左边的叶子结点是 p 的后继。因此, 算法思路是, 先找 p 的双亲 f , 根据 p 是 f 的左/右子女再确定 p 的后序后继。

```

BiThrTree InThrPostSucc(BiThrTree r, p)
//在中序线索二叉树 r 上, 求结点 p (假定存在) 的后序后继结点 q)
{if(p==r) return(null) //若 p 为根结点, 后序后继为空
T=p
while(T->LT==1) T=T->LL; //找 p 的最左子孙的左线索
q=T->LL; //q 是 p 最左子孙的左线索, 是 p 的祖先
if(q->RL==p) return(q); //p 是 q 的右子女, 则 q 是 p 后序后继。
T=p;
while(T->RT==1) T=T->RL; //找 p 的最右子孙的右线索
q=T->RL; //q 是 p 最右子孙的右线索
if(q->LL==p) //若 p 是 q 的左子女
  if(q->RT==0) return(q); //若 p 是 q 的左子女且 q 无右子女, 则 p 的后序后继是 q
else //p 的双亲 q 有右子树, 则 q 的右子树上最左下的叶子结点是 p 的后继
{q=q->RL;
  while(q->LT==1 || q->RT==1) //找 q 右子树中最左下的叶子结点
  {while(q->LT==1) q=q->LL; //向左下
    if(q->RT==1) q=q->RL; //若 q 无左子女但有右子女, 则向右下, 直到叶子结点
  }
  return(q); //q 是 p 的后继
}
} //结束 InThrPostSucc

```

[算法讨论] 请注意本题条件: 标记为 0 时是线索, 而为 1 时是指向子女。

78. [题目分析]第 77 题已讨论了在中序线索树中查找结点 p 的后序后继问题, 本题要求在中序线索树上进行后序遍历。因后序遍历是“左右根”, 最后访问根结点, 即只有从右子树返回时才能访问根结点, 为此设一标志 `returnflag`, 当其为 1 时表示从右侧返回, 可以访问根结点。为了找当前结点的后继, 需知道双亲结点的信息, 在中序线索树中, 某结点最左子孙的左线索和最右子孙的右线索均指向其双亲, 因此设立两个函数 `LeftMost` 和 `RightMost` 求结点的最左和最右子孙, 为了判定是否是从右子树返回, 再设一函数 `IsRightChild`。

```

BiThrTree LeftMost(BiThrTree t) //求结点 t 最左子孙的左线索
{BiThrTree p=t;
while(p->ltag==0) p=p->lchild; //沿左分枝向下
if (p->lchild!=null) return(p->lchild); else return(null);
} //LeftMost

```

```

BiThrTree RightMost (BiThrTree t) //求结点 t 最右子孙的右线索
{
    BiThrTree p=t;
    while(p->rtag==0) p=p->rchild;    //沿右分枝向下
    if (p->rchild!=null) return (p->rchild); else return(null);
} //RightMost

int IsRightChild(BiThrTree t, father)    //若 t 是 father 的右孩子, 返回 1, 否则返回 0
{
    father=LeftMost(t);
    if(father && father->rchild==t) return(1); else return(0);
} //Is RightChild;

void PostOrderInThr (BiThrTree bt) //后序遍历中序线索二叉树 bt
{
    BiThrTree father, p=bt;
    int flag;
    while(p!=null)
    {
        while(p->ltag==0 ) p=p->lchild; // 沿左分枝向下
        if(p->rtag==0) flag=0; //左孩子为线索, 右孩子为链, 相当从左返回
        else flag=1;          //p 为叶子, 相当从右返回
        while(flag==1)
        {
            visit(*p); //访问结点
            if(IsRightChild(p, father)) {p=father; flag=1;} //修改 p 指向双亲
            else //p 是左子女, 用最右子孙的右线索找双亲
            {
                p=RightMost(p);
                if(p->rtag==1) flag=1; else flag=0;
            }
        } // while(flag==1)
        if(flag==0 && p!=null) p=p->rchild; //转向当前结点右分枝
    } //结束 PostOrderInThr
}

```

79. (1) 哈夫曼树的构造过程

① 根据给定的 n 个权值 $\{W_1, W_2, W_3, \dots, W_n\}$ 构成 n 棵二叉树的集合 $F=\{T_1, T_2, \dots, T_n\}$, 其中每棵二叉树 T_i 只有权值为 W_i 的根结点, 其左右子树均为空。

② 在 F 中选取两棵根结点的权值最小的树作左右子树构造一棵新二叉树, 新二叉树根结点的权值为其左右子树上根结点的权值之和。

③ 在 F 中删除这两棵树, 同时将新得到的二叉树加入 F 中。

④ 重复②和③, 直到 F 中只剩一棵树为止。这棵树便是哈夫曼树。

(2) 含有 n 个叶子结点的哈夫曼树共有 $2n-1$ 个结点, 采用静态链表作为存储结构, 设置大小为 $2n-1$ 的数组。现将哈夫曼树的结点及树的结构定义如下:

```

typedef struct {float weight;    //权值
               int parent, lc, rc; //双亲、左、右子女 } node;

typedef node HufmTree[2*n-1];

void Huffman(int n, float w[n], HufmTree T)
{
    //构造 n 个叶子结点的哈夫曼树 T, n 个权值已放在 W[n] 数组中
    {int i, j, p1, p2          //p1, p2 为最小值和次最小值的坐标
    float small1, small2;    //small1 和 small2 为权值的最小值和次小值
    for(i=0; i<2*n-1; i++)    //置初值, 结点的权、左、右子女, 双亲
        {T[i].parent=-1; T[i].lc=-1; T[i].rc=-1;
        if(i<n) T[i].weight=w[i]; else T[i].weight=0;

```

```

}
for (i=n ;i<2*n-1;i++)          //构造新二叉树
{p1=p2=0;small1=small2=maxint; //初值
for(j=0;j<i;j++)
    if(T[j].weight<small1 && T[j].parent!=-1)          //最小值
        {p2=p1; small2=small1; p1=j; small1=T[j].weight;}
    else if(T[j].weight<small2 && T[j].parent!=-1)      //次小值
        {p2=j;small2=T[j].weight;}
T[i].weight=T[p1].weight+T[p2].weight;                  //合并成一棵新二叉树
T[i].lc=p1; T[i].rc=p2;                                //置双亲的左右子女
T[p1].parent=i; T[p2].parent=i;                        //置左、右子女的双亲
} //for(i=0;i<2*n-1;i++) } //结束 huffman
求哈夫曼编码的算法
typedef struct {char bit[n]; int start;} codetype;
void HuffmanCode(CodeType code[n], HufmTree T) //哈夫曼树 T 已求出，现求其哈夫曼编码
{int i, j, c, p;
CodeType cd;
for(i=0;i<n;i++)
{cd.start=n;c=i;p=T[i].parent;
while(p!=-1)
{cd.start--;
if(T[p].lc==c) cd.bit[cd.start]='0' //左分枝生成代码 '0'
else cd.bit[cd.start]='1';          // 右分枝生成代码 '1'
c=p; p=T[p].parent;                //双亲变为新子女，再求双亲的双亲
}
code[i]=cd;                        //成组赋值，求出一个叶子结点的哈夫曼编码
} //for } //结束 HuffmanCode

```

80. [题目分析] 二叉树的层次遍历序列的第一个结点是二叉树的根。实际上, 层次遍历序列中的每个结点都是“局部根”。确定根后, 到二叉树的中序序列中, 查到该结点, 该结点将二叉树分为“左根右”三部分。若左、右子树均有, 则层次序列根结点的后面应是左右子树的根; 若中序序列中只有左子树或只有右子树, 则在层次序列的根结点后也只有左子树的根或右子树的根。这样, 定义一个全局变量指针 R, 指向层次序列待处理元素。算法中先处理根结点, 将根结点和左右子女的信息入队列。然后, 在队列不空的条件下, 循环处理二叉树的结点。队列中元素的数据结构定义如下:

```
typedef struct
{
    int lvl;           //层次序列指针，总是指向当前“根结点”在层次序列中的位置
    int l, h;         //中序序列的下上界
    int f;             //层次序列中当前“根结点”的双亲结点的指针
    int lr;            // 1—双亲的左子树  2—双亲的右子树
}qnode;

BiTree Creat(datatype in[], level[], int n)
//由二叉树的层次序列 level[n]和中序序列 in[n]生成二叉树。 n 是二叉树的结点数
{
    if (n<1) {printf(“参数错误\n”); exit(0);}
    qnode s, Q[];      //Q 是元素为 qnode 类型的队列，容量足够大
    init(Q); int R=0;   //R 是层次序列指针，指向当前待处理的结点
    BiTree p=(BiTree)malloc(sizeof(BiNode)); //生成根结点
```

```

p->data=level[0]; p->lchild=null; p->rchild=null; //填写该结点数据
for (i=0; i<n; i++) //在中序序列中查找根结点, 然后, 左右子女信息入队列
    if (in[i]==level[0]) break;
if (i==0) //根结点无左子树, 遍历序列的 1—n-1 是右子树
{p->lchild=null;
 s.lvl=++R; s.l=i+1; s.h=n-1; s.f=p; s.lr=2; enqueue(Q, s);
}
else if (i==n-1) //根结点无右子树, 遍历序列的 1—n-1 是左子树
{p->rchild=null;
 s.lvl=++R; s.l=1; s.h=i-1; s.f=p; s.lr=1; enqueue(Q, s);
}
else //根结点有左子树和右子树
{s.lvl=++R; s.l=0; s.h=i-1; s.f=p; s.lr=1; enqueue(Q, s); //左子树有关信息入队列
 s.lvl=++R; s.l=i+1; s.h=n-1; s.f=p; s.lr=2; enqueue(Q, s); //右子树有关信息入队列
}
while (!empty(Q)) //当队列不空, 进行循环, 构造二叉树的左右子树
{ s=delqueue(Q); father=s.f;
 for (i=s.l; i<=s.h; i++)
     if (in[i]==level[s.lvl]) break;
 p=(bitreptr)malloc(sizeof(binode)); //申请结点空间
 p->data=level[s.lvl]; p->lchild=null; p->rchild=null; //填写该结点数据
 if (s.lr==1) father->lchild=p;
 else father->rchild=p; //让双亲的子女指针指向该结点
 if (i==s.l)
 {p->lchild=null; //处理无左子女
 s.lvl=++R; s.l=i+1; s.f=p; s.lr=2; enqueue(Q, s);
 }
 else if (i==s.h)
 {p->rchild=null; //处理无右子女
 s.lvl=++R; s.h=i-1; s.f=p; s.lr=1; enqueue(Q, s);
 }
 else {s.lvl=++R; s.h=i-1; s.f=p; s.lr=1; enqueue(Q, s); //左子树有关信息入队列
 s.lvl=++R; s.l=i+1; s.f=p; s.lr=2; enqueue(Q, s); //右子树有关信息入队列
 }
 } //结束 while (!empty(Q))
return(p);
} //算法结束

```

第 7 章 图

一. 选择题

1. A	2. B	3. A	4. B	5. D	6. 1B	6. 2D	7. 1B	7. 2C	8. A	9. A	10. 1C
10. 2BDE	11. B	12. 1B	12. 2B	12. 3D	13. B	14. C	15. C	16. D	17. D	18. 1C	18. 2C
19. AB	20. B	21. 1C	21. 2A	21. 3B	21. 4A	22. A	23. A	24. D	25. A	26. A	27. B
28. D	29. B	30. A	31. C	32. B							

二. 判断题

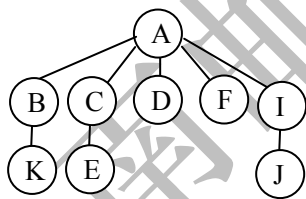
1. √	2. ×	3. ×	4. ×	5. ×	6. √	7. ×	8. ×	9. ×	10. ×	11. √	12. ×
13. √	14. ×	15. ×	16. ×	17. √	18. ×	19. ×	20. ×	21. ×	22. ×	23. ×	24. ×
25. ×	26. √	27. ×	28. √	29. ×	30. ×	31. √	32. √	33. ×	34. ×	35. ×	36. √
37. ×	38. ×	39. ×	40. ×	41. √	42. ×	43. ×	44. √	45. ×	46. ×	47. ×	48. √
49. ×											

部分答案解释如下。

2. 不一定是连通图，可能有若干连通分量
11. 对称矩阵可存储上（下）三角矩阵
14. 只有有向完全图的邻接矩阵是对称的
16. 邻接矩阵中元素值可以存储权值
21. 只有无向连通图才有生成树
22. 最小生成树不唯一，但最小生成树上权值之和相等
26. 是自由树，即根结点不确定
35. 对有向无环图，拓扑排序成功；否则，图中有环，不能说算法不适合。
42. AOV 网是用顶点代表活动，弧表示活动间的优先关系的有向图，叫顶点表示活动的网。
45. 能求出关键路径的 AOE 网一定是有向无环图
46. 只有该关键活动为各关键路径所共有，且减少它尚不能改变关键路径的前提下，才可缩短工期。
48. 按着定义，AOE 网中关键路径是从“源点”到“汇点”路径长度最长的路径。自然，关键路径上活动的时间延长多少，整个工程的时间也就随之延长多少。

三. 填空题

1. 有 n 个顶点， $n-1$ 条边的无向连通图
2. 有向图的极大强连通子图
3. 生成树
4. 45
5. $n(n-1)/2$
6. .
7. 9
8. n
9. $2(n-1)$
10. $N-1$
11. $n-1$
12. n
13. $N-1$
14. n
15. N
16. 3
17. $2(N-1)$
18. 度 出度
19. 第 I 列非零元素个数
20. $n \quad 2e$
21. (1) 查找顶点的邻接点的过程 (2) $O(n+e)$ (3) $O(n+e)$ (4) 访问顶点的顺序不同 (5) 队列和栈
22. 深度优先
23. 宽度优先遍历
24. 队列
25. 因未给出存储结构，答案不唯一。本题按邻接表存储结构，邻接点按字典序排列。



25 题 (1)

26. 普里姆 (prim) 算法和克鲁斯卡尔 (Kruskal) 算法
27. 克鲁斯卡尔
28. 边稠密 边稀疏
29. $O(e \log e)$ 边稀疏
30. $O(n^2)$ $O(e \log e)$
31. (1) (V_i, V_j) 边上的权值 都大的数 (2) 1 负值 (3) 为负 边
32. (1) $n-1$ (2) 普里姆 (3) 最小生成树
33. 不存在环
34. 递增 负值
35. 160
36. $O(n^2)$
37. 50, 经过中间顶点④
38. 75
39. $O(n+e)$
40. (1) 活动 (2) 活动间的优先关系 (3) 事件 (4) 活动 边上的权代表活动持续时间
41. 关键路径
42. (1) 某项活动以自己为先决条件 (2) 荒谬 (3) 死循环
43. (1) 零 (2) V_k 度减 1, 若 V_k 入度已减到零, 则 V_k 顶点入栈 (3) 环
44. (1) $p < \text{nil}$ (2) $\text{visited}[v] = \text{true}$ (3) $p = g[v].\text{firstarc}$ (4) $p = p.\text{nextarc}$
45. (1) $g[0].\text{vexdata} = v$ (2) $g[j].\text{firstin}$ (3) $g[j].\text{firstin}$ (4) $g[i].\text{firstout}$ (5) $g[i].\text{firstout}$

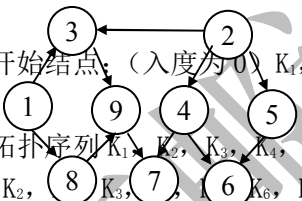
报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

- (6) $p^{\wedge}.vexj$ (7) $g[i].firstout$ (8) $p:=p^{\wedge}.nexti$ (9) $p<>nil$ (10) $p^{\wedge}.vexj=j$
 (11) $firstadj(g, v_0)$ (12) $not\ visited[w]$ (13) $nextadj(g, v_0, w)$
 46. (1) 0 (2) j (3) i (4) 0 (5) $indegree[i]==0$ (6) $[vex][i]$ (7) $k==1$ (8) $indegree[i]==0$
 47. (1) $p^{\wedge}.link:=ch[u].head$ (2) $ch[u].head:=p$ (3) $top<>0$ (4) $j:=top$ (5) $top:=ch[j].count$
 (6) $t:=t^{\wedge}.link$
 48. (1) $V1\ V4\ V3\ V6\ V2\ V5$ (尽管图以邻接表为存储结构, 但因没规定邻接点的排列, 所以结果是不唯一的。本答案是按邻接点升序排列给出的。)
 (2) ① $top==1$ ② $top=graph[j].count$ ③ $graph[k].count==0$

四. 应用题

1. (1) $G1$ 最多 $n(n-1)/2$ 条边, 最少 $n-1$ 条边 (2) $G2$ 最多 $n(n-1)$ 条边, 最少 n 条边
 (3) $G3$ 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 $K_1, K_2, K_3, K_4, K_5, K_6, K_8, K_9, K_7$
 $K_2, K_3, K_8, K_7, K_6, K_8, K_9, K_7$
 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。

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:
 ($a \rightarrow b \rightarrow e \rightarrow i$), ($a \rightarrow c \rightarrow g \rightarrow i$),
 ($a \rightarrow c \rightarrow b \rightarrow e \rightarrow i$)

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（ $i\text{link}$ ）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 $i\text{link}$ 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 $\langle i, j \rangle$ ，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 $\text{dfs}(v)$ 时，若在退出 $\text{dfs}(v)$ 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

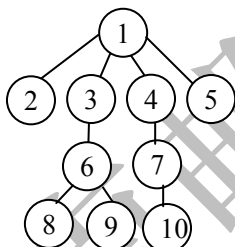
注：（1）邻接表不唯一，这里顶点的邻接点按升序排列

（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） $V_1V_2V_4V_3V_5V_6$ （2） $V_1V_2V_3V_4V_5V_6$

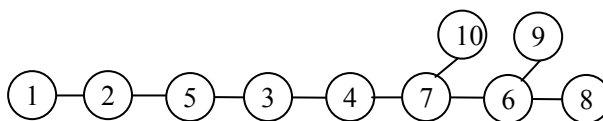
16. (1)



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)

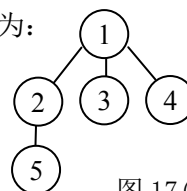


图 17(2)

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。

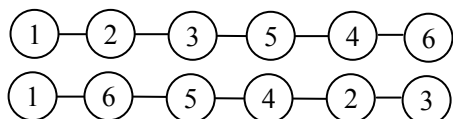
19. (1) 邻接矩阵： $(6 \times 6 \text{ 个元素}) \times 2 \text{ 字节/元素} = 72 \text{ 字节}$

邻接表：表头向量 $6 \times (4+2) + \text{边结点 } 9 \times (2+2+4) \times 2 = 180 \text{ 字节}$

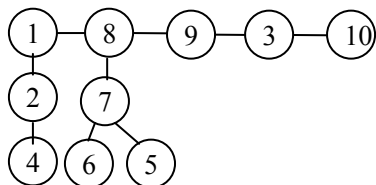
邻接多重表：表头向量 $6 \times (4+2) + \text{边结点 } 9 \times (2+2+2+4+4) = 162 \text{ 字节}$

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：

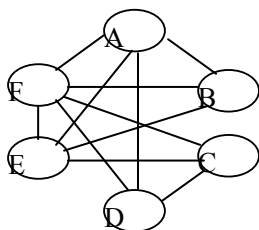


20. 未确定存储结构，其 DFS 树不唯一，其中之一（按邻接点逆序排列）是



(2) 关节点有 3, 1, 8, 7, 2

21. (1)



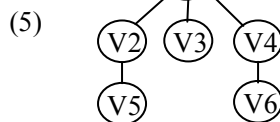
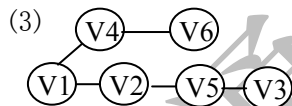
(2) AFEDBC

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



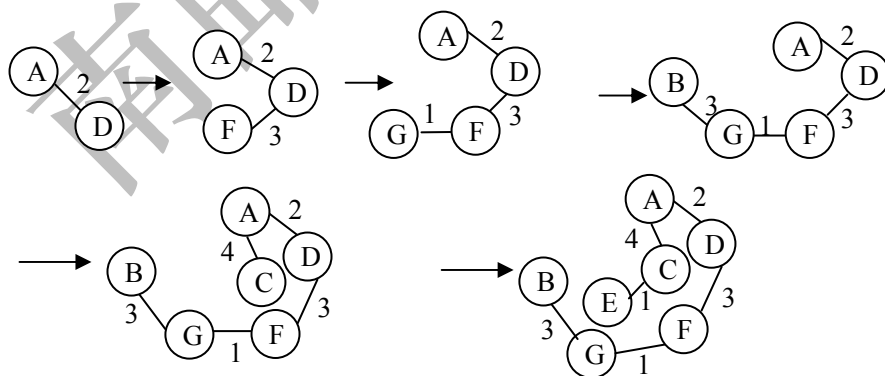
(6) 见本章五算法设计第 6 题

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)

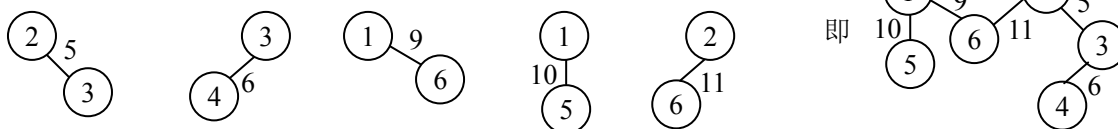


25. 在有相同权值边时生成不同的 MST，在这种情况下，用 Prim 或 Kruskal 也会生成不同的 MST。

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 e_i ”，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. $V(G) = \{1, 2, 3, 4, 5, 6, 7\}$

$E(G) = \{(1, 6, 4), (1, 7, 6), (2, 3, 5), (2, 4, 8), (2, 5, 12), (1, 2, 18)\}$

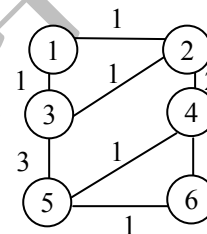
30. $V(G) = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$E(G) = \{(3, 8, 2), (4, 7, 2), (3, 4, 3), (5, 8, 3), (2, 5, 4), (6, 7, 4), (1, 2, 5)\}$

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	2	3	4	5	6	7	8	U	V-U
Closedge	2	3	4					{1}	{2, 3, 4, 5, 6, 7, 8}
Vex	①	①							
Lowcost	2	3							
Vex		①	②						
Lowcost		3							
[错误] 系统维护中									

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
```

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

```

    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(g2,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=(OrArcNode *)malloc(sizeof(OrArcNode)); //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  $v_i$  和  $v_j$  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  $v_i$  各  $v_j$  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  $v_i$  或  $v_j$  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  $v_i$  和  $v_j$  间有路径。
      else {visited[i]=1; stack[++top]=i;}//else
    }//else
  }while
  return(0); } //顶点  $v_i$  和  $v_j$  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->tailink; //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, vtype 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 (visied[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。

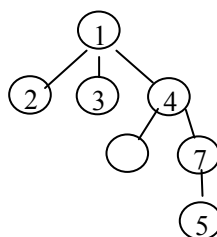
27. (1) [题目分析]D_搜索类似 BFS, 只是用栈代替队列, 入出队列改为入出栈。查某顶点的邻接点时, 若其邻接点尚未遍历, 则遍历之, 并将其压入栈中。当一个顶点的所有邻接点被搜索后, 栈顶顶点是下一个搜索出发点。

```

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",j); 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 $\langle i, j \rangle$, 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 $\langle i, j \rangle$, 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。
```

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

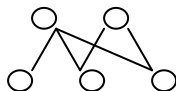
报名热线: 025-83535877、18951896587、 18951896993、 18951896967

```

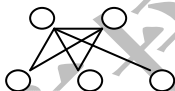
node v[]; //顶点元素数组。
int c[]; //中间变量
for (i=1;i<=n;i++) //顶点信息数组初始化, 设图有 n 个顶点。
    {v[i].vertex=i; v[i].out=0; v[i].count=1; c[i]=1;} //count=1 为最小
for (i=1;i<=n;i++) //计算每个顶点的出度。
    for (j=1;j<=n;j++) v[i].out+=g1[i][j];
for (i=n;i>=2;i--) //对 v 的出度域进行计数排序, 出度大者, count 域中值小。
    for (j=i-1;j>=1;j--)
        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.



二部图



非二部图

[题目分析] 将顶点放在两个集合 V_1 和 V_2 。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 就可遍历完全部结点，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 ,vertype 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:

$$A^{(0)} =$$

$$A^{(5)} =$$

$$\text{第 33 题结果矩阵 } A^{(6)} =$$

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，偏心度初值为机器最大数。
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度。
    if (s<dist) {dist=s; v=j;} //各偏心度中最小值为中心点。
} //for
printf("有向图 g 的中心点是顶点%d, 偏心度%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, vertype 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：

$low(v) = \text{Min}(visited[v], low[w], visited[k])$

其中 $(v, w) \in Edge$, $(v, k) \in Edge$, 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 $low[w] \geq visited[v]$ ，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)
{p=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; //将该顶点插入链表
} //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 8 章 动态存储管理

一. 选择题 1C

二. 判断题 1. 错误 2. 正确

三. 填空题

1. (1) $480+8=488$ ($480 \% 2^{3+1}=0$) (2) $480-32=448$

2. (1) 011011110100 (2) 011011100000

3. 用户不再使用而系统没有回收的结构和变量。例如, $p=\text{malloc}(\text{size}); \dots, p=\text{null};$

四. 应用题

- 在伙伴系统中, 无论占用块或空闲块, 其大小均为 2 的 k (k 为 ≥ 0 的正整数) 次幂。若内存容量为 2^n , 则空闲块大小只能是 $2^0, 2^1, 2^2, \dots, 2^n$ 。由同一大块分裂而得的两个小块互称“伙伴空间”, 如内存大小为 2^{10} 的块分裂成两个大小为 2^9 的块。只有两个“伙伴空间”才能合并成一个大空间。

起始地址为 p , 大小为 2^k 的内存块, 其伙伴的起始地址为:

$\text{buddy}(p, k)=p+2^k$ (若 $p \% 2^{k+1}=0$), 或 $\text{buddy}(p, k)=p-2^k$ (若 $p \% 2^{k+1}=2^k$)

- 首次拟合法: 从链表头指针开始查找, 找到第一个 \geq 所需空间的结点即分配。

最佳拟合法: 链表结点大小增序排列, 找到第一个 \geq 所需空间的结点即分配。

最差拟合法: 链表结点大小逆序排列, 总从第一个结点开始分配, 将分配后结点所剩空间插入到链表适当位置。

首次拟合法适合事先不知道请求分配和释放信息的情况, 分配时需查询, 释放时插在表头。最佳拟合法适用于请求分配内存大小范围较宽的系统, 释放时容易产生存储量很小难以利用的内存碎片, 同时保留那些很大的内存块以备将来可能发生的大内存量的需求, 分配与回收均需查询。最差拟合法适合请求分配内存大小范围较窄的系统, 分配时不查询, 回收时查询, 以便插入适当位置。

3. 011011110100

4. 011011100000

5. (1) $\text{buddy}(1664, 7)=1664-128=1536$ (2) $\text{buddy}(2816, 6)=2816+64=2880$

- 动态存储分配伙伴系统的基本思想请参见上面题 1。边界标识法在每块的首尾均有“占用”/“空闲”标志, 空闲块合并方便。伙伴系统算法简单, 速度快, 但只有互为伙伴的两个空闲块才可合并, 因而易产生虽空闲但不能归并的碎片。

- 组织成循环链表的可利用空间表的结点大小按递增序排列时, 首次适配策略就转变为最佳适配策略。

- 因为 $512=2^9$, 可利用空间表的初始状态图如 8-1 所示。

当用户申请大小为 23 的内存块时, 因 $2^4 < 23 < 2^5$, 但没有大小为 2^5 的块, 只有大小为 2^9 的块, 故将 2^9 的块分裂成两个大小为 2^8 的块, 其中大小为 2^8 的一块挂到可利用空间表上, 另一块再分裂成两个大小为 2^7 的块。

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

又将其大小为 2^7 的一块挂到可利用空间表上, 另一块再分裂成两个大小为 2^6 的块, 一块 2^6 的块挂到可利用空间表上, 另一块分裂成两个大小为 2^5 的块, 其中一块挂到可利用空间表上, 另一块分给用户 (地址 0—31)。如此下去, 最后每个用户得到的存储空间的起始地址如图 8-2, 6 个用户分配所需要的存储空间后可利用空间表的状态如图 8-3。

在回收时, 因为给申请 45 的用户分配了 2^6 , 其伙伴地址是 0, 在占用中, 不能合并, 只能挂到可利用空间表上。在回收大小为 52 的占用块时, 其伙伴地址是 192, 也在占用。回收大小为 11 的占用块时, 其伙伴地址是 48, 可以合并为大小 2^5 的块, 挂到可利用空间表上。回收 3 个占用块之后可利用空间表的状态如图 8-4。

存储大小	起始地址
23	0
45	64
52	128
100	256
11	32
19	192

图 8-2

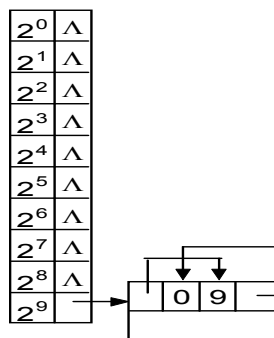


图 8-1

(注: 在图 8.3 和 8.4 画上了占用块, 从原理上, 只有空闲块才出现在“可利用空间表”中。)

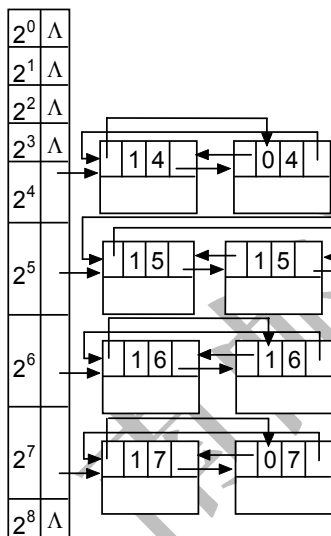


图 8-3

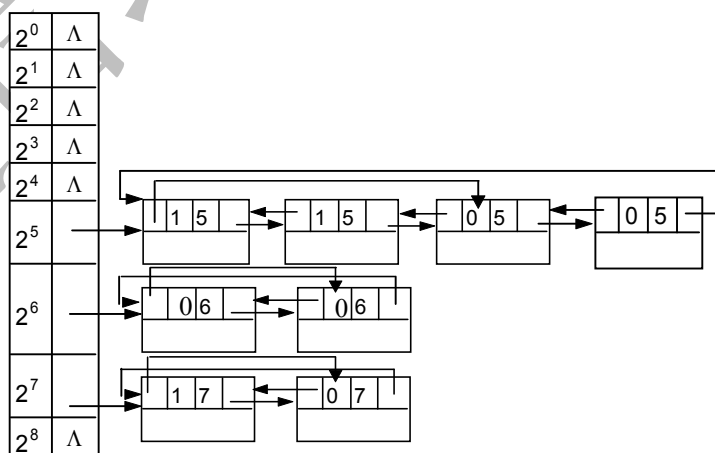
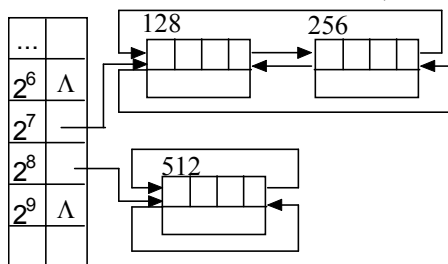
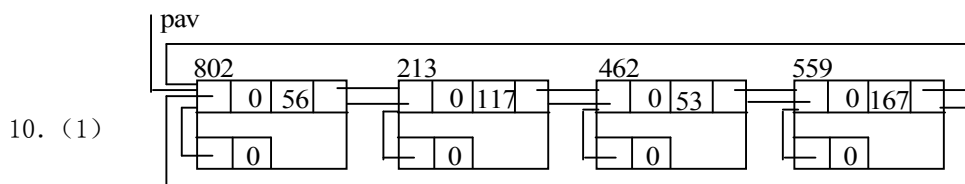


图 8-4

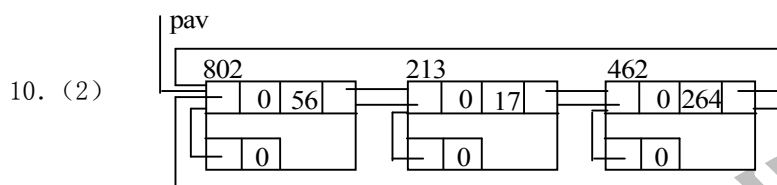
9. 因为 $768 \% 2^{7+1}=0$, 所以 768 和 $768+2^7=896$ 互为伙伴, 伙伴合并后, 首址为 768, 块大小为 2^8 。因为 $768 \% 2^{8+1}=2^8$, 所以, 所以首址 768 大小为 2^8 的块和首址 512 大小为 2^8 的块合并, 成为首址 512 大小为 2^9 的空闲块。因为 $128 \% 2^{7+1}=2^7$, 其伙伴地址为 $128-2^7=0$, 将其插入可利用空间表中。回收后该伙伴系统的状态图如下。



10. (1) 系统回收一个起始地址为 559，大小为 45 的空闲块后，因右侧起始地址 604 为空闲块，应与之合并。合并后，起始地址为 559，大小为 167 的空闲块。链表状态如图 10. (1) 所示。



(2) 系统在接收存储块大小为 100 的请求后，将大小为 117 的空闲块分出 100 给予用户。在回收一个起始地址为 515，大小为 44 的空闲块之后，因左侧起始地址 462 大小 53 和右侧起始地址 559 大小 167 均为空闲块，应与之合并。合并后，起始地址为 462，大小为 264 的空闲块。链表状态如图 10. (2) 所示。



第 9 章 集合

一. 选择题

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

二. 判断题

1. √	2. √	3. ×	4. ×	5. ×	6. √	7. √	8. ×	9. ×	10. ×	11. ×	12. √
13. √	14. ×	15. ×	16. ×	17. √	18. ×	19. √	20. ×	21. ×	22. ×	23. ×	24. ×
25. √	26. ×	27. ×	28. √	29. √	30. ×	31. ×	32. √	33. √	34. ×	35. √	36. √

部分答案解释如下。

4. 不能说哪种哈希函数的选取方法最好，各种选取方法有自己的适用范围。
8. 哈希表的结点中可以包括指针，指向其元素。
11. 单链表不能使用折半查找方法。
20. 按插入后中序遍历是递增序列的原则，若某结点只有右子树，而插入元素的关键字小于该结点的关键字，则会插入到该结点的左侧，成为其左孩子。这种插入就不是插入到叶子下面。
21. 从平衡因子定义看，完全二叉树任一结点的平衡因子的绝对值确实是小于等于 1。但是，平衡二叉树本质上是二叉排序树，完全二叉树不一定是排序树。故不能说完全二叉树是平衡二叉树。
23. 某结点的左子树根结点不一定是它的中序前驱，其右子树根结点也不一定是它的中序后继。
24. 在等概率下，查找成功时的平均查找长度相同，查找失败时的平均查找长度不相同。
26. 只有被删除结点是叶子结点时命题才正确。

三. 填空题

1. n n+1
2. 4
3. 6, 9, 11, 12
4. 5

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

5. 26 (第4层是叶子结点, 每个结点两个关键字) 6. 1, 3, 6, 8, 11, 13, 16, 19
 7. 5, 96 8. $m-1, \lceil m/2 \rceil -1$ 9. 2, 4, 3
 10. (1) 哈希函数 (2) 解决冲突的方法 (3) 选择好的哈希函数 (4) 处理冲突的方法 (5) 均匀 (6) 简单
 11. AVL 树 (高度平衡树, 高度平衡的二叉排序树), 或为空二叉树, 或二叉树中任意结点左右子树高度与右子树高度差的绝对值小于等于 1。
 12. 小于等于表长的最大素数或不包含小于 20 的质因子的合数 13. 16 14. $\lfloor \log_2 n \rfloor + 1$
 15. (1) 45 (2) 45 (3) 46 (块内顺序查找) 16. $k(k+1)/2$ 17. 30, 31.5 (块内顺序查找)
 18. (1) 顺序存储或链式存储 (2) 顺序存储且有序 (3) 块内顺序存储, 块间有序 (4) 散列存储
 19. $(n+1)/2$ 20. $(n+1)/n \cdot \log_2(n+1) - 1$ 21. 结点的左子树的高度减去结点的右子树的高度
 22. (1) 顺序表 (2) 树表 (3) 哈希表 (4) 开放定址方法 (5) 链地址方法 (6) 再哈希 (7) 建立公共溢出区

23. 直接定址法 24. $\log_{\lceil m/2 \rceil}(\frac{n+1}{2}) + 1$ 25. $O(N)$ 26. $n(n+1)/2$
 27. 54 28. 31 29. 37/12 30. 主关键字 31. 左子树 右子树
 32. 插入 删除 33. 14 34. (1) 126 (2) 64 (3) 33 (4) 65
 35. (1) $low \leq high$ (2) $(low+high) \text{ DIV } 2$ (3) $binsrch:=mid$ (4) $binsrch:=0$
 36. (1) k (2) $I < n+1$ 37. (1) $rear=mid-1$ (2) $head=mid+1$ (3) $head > rear$
 38. (1) $p! = null$ (2) $pf=p$ (3) $p! = *t$ (4) $*t = null$

四. 应用题

- 概念是基本知识的主要部分, 要牢固掌握。这里只列出一部分, 目的是引起重视, 解答略。
- (1) 散列表存储的基本思想是用关键字的值决定数据元素的存储地址
 (2) 散列表存储中解决碰撞的基本方法:
 - 开放定址法 形成地址序列的公式是: $H_i = (H(\text{key}) + d_i) \% m$, 其中 m 是表长, d_i 是增量。根据 d_i 取法不同, 又分为三种:
 - $d_i = 1, 2, \dots, m-1$ 称为线性探测再散列, 其特点是逐个探测表空间, 只要散列表中有空闲空间, 就可解决碰撞, 缺点是容易造成“聚集”, 即不是同义词的关键字争夺同一散列地址。
 - $d_i = 1^2, -1^2, 2^2, -2^2, \dots, \pm k^2 (k \leq m/2)$ 称为二次探测再散列, 它减少了聚集, 但不容易探测到全部表空间, 只有当表长为形如 $4j+3$ (j 为整数) 的素数时才有可能。
 - d_i = 伪随机数序列, 称为随机探测再散列。
 - 再散列法 $H_i = RH_i(\text{key})$ $i=1, 2, \dots, k$, 是不同的散列函数, 即在同义词产生碰撞时, 用另一散列函数计算散列地址, 直到解决碰撞。该方法不易产生“聚集”, 但增加了计算时间。
 - 链地址法 将关键字为同义词的记录存储在同一链表中, 散列表地址区间用 $H[0..m-1]$ 表示, 分量初始值为空指针。凡散列地址为 i ($0 \leq i \leq m-1$) 的记录均插在以 $H[i]$ 为头指针的链表中。这种解决方法中数据元素个数不受表长限制, 插入和删除操作方便, 但增加了指针的空间开销。这种散列表常称为开散列表, 而①中的散列表称闭散列表, 含义是元素个数受表长限制。
 - 建立公共溢出区 设 $H[0..m-1]$ 为基本表, 凡关键字为同义词的记录, 都填入溢出区 $O[0..m-1]$ 。

(3) 用分离的同义词表和结合的同义词表解决碰撞均属于链地址法。链地址向量空间中的每个元素不是简单的地址, 而是关键字和指针两个域, 散列地址为 i ($0 \leq i \leq m-1$) 的第一个关键字存储在地址空间向量第 i 个分量的“关键字”域。前者的指针域是动态指针, 指向同义词的链表, 具有上面③的优缺点; 后者实际是静态链表, 同义词存在同一地址向量空间 (从最后向前找空闲单元), 以指针相连。节省了空间, 但易产生“堆积”, 查找效率低。

(4) 要在被删除结点的散列地址处作标记, 不能物理的删除。否则, 中断了查找通路。

(5) 记录 负载因子

3. 评价哈希函数优劣的因素有：能否将关键字均匀影射到哈希空间上，有无好的解决冲突的方法，计算哈希函数是否简单高效。由于哈希函数是压缩映像，冲突难以避免。解决冲突的方法见上面 2 题。
4. 哈希方法的平均查找路长主要取决于负载因子（表中实有元素数与表长之比），它反映了哈希表的装满程度，该值一般取 0.65~0.9。解决冲突方法见上面 2 题。
5. 不一定相邻。哈希地址为 i ($0 \leq i \leq m-1$) 的关键字，和为解决冲突形成的探测序列 i 的同义词，都争夺哈希地址 i 。
- 6.

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	14	01	9	23	84	27	55	20		
比较次数	1	1	1	2	3	4	1	2		

平均查找长度： $ASL_{succ} = (1+1+1+2+3+4+1+2) / 8 = 15/8$

以关键字 27 为例： $H(27) = 27\%7 = 6$ （冲突） $H_1 = (6+1)\%10 = 7$ （冲突）

$H_2 = (6+2^2)\%10 = 0$ （冲突） $H_3 = (6+3^3)\%10 = 5$ 所以比较了 4 次。

7. 由于装填因子为 0.8，关键字有 8 个，所以表长为 $8/0.8 = 10$ 。

(1) 用除留余数法，哈希函数为 $H(key) = key \% 7$

(2)

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	21	15	30	36	25	40	26	37		
比较次数	1	1	1	3	1	1	2	6		

- (3) 计算查找失败时的平均查找长度，必须计算不在表中的关键字，当其哈希地址为 i ($0 \leq i \leq m-1$) 时的查找次数。本例中 $m=10$ 。故查找失败时的平均查找长度为：

$ASL_{unsucc} = (9+8+7+6+5+4+3+2+1+1) / 10 = 4.6$ $ASL_{succ} = 16/8 = 2$

(4) `int Delete(int h[n], int k)`

// 从哈希表 $h[n]$ 中删除元素 k ，若删除成功返回 1，否则返回 0

$\{i = k\%7$; // 哈希函数用上面 (1)，即 $H(key) = key \% 7$

`if (h[i] == maxint) // maxint 解释成空地址`

`printf(“无关键字%d\n”, k); return (0); }`

`if (h[i] == k) {h[i] = -maxint; return (1); }` // 被删元素换成最大机器数的负数

`else` // 采用线性探测再散列解决冲突

`{j = i;`

`for (d = 1; d ≤ n-1; d++)`

`{i = (j+d) % n;` // n 为表长，此处为 10

`if (h[i] == maxint) return (0); // maxint 解释成空地址`

`if (h[i] == k) { h[i] = -maxint; return (1); }`

`} // for`

`}`

`printf(“无关键字%d\n”, k); return (0)`

`}`

8.

散列地址	0	1	2	3	4	5	6	7	8	9
关键字		15		24	10	19	17	38	18	40
比较次数		1		1	2	1	4	5	5	5

哈希表 a: $ASL_{succ} = 24/8 = 3$;

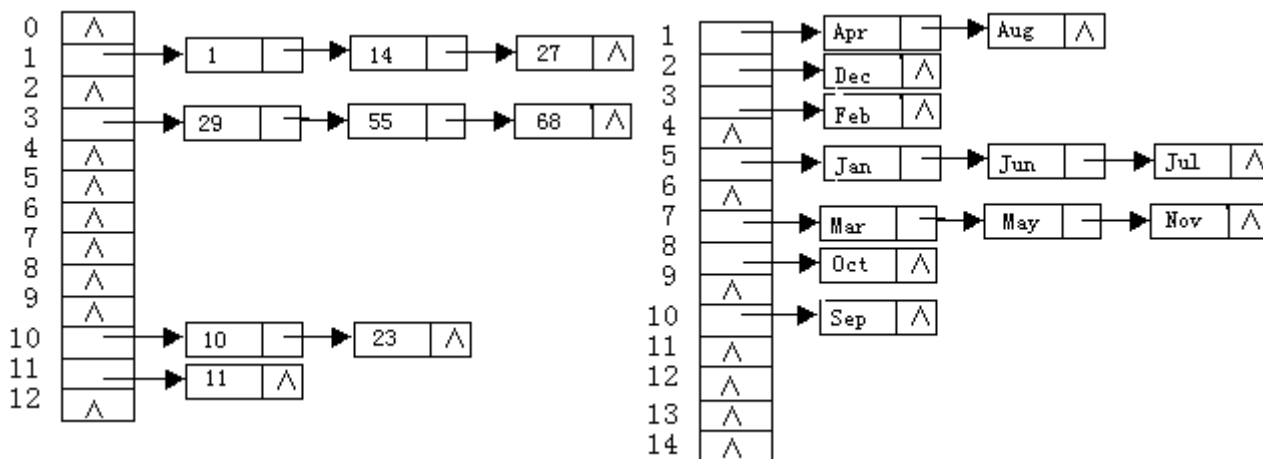
散列地址	0	1	2	3	4	5	6	7	8	9
------	---	---	---	---	---	---	---	---	---	---

关键字		15	17	24	10	19	40	38	18	
比较次数		1	3	1	2	1	2	4	4	

哈希表 b: $ASL_{succ} = 18/8$

9. (1)

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键字	13	22		53	1		41	67	46		51		30
比较次数	1	1		1	2		1	2	1		1		1



(2) 装填因子 $= 9/13 = 0.7$ (3) $ASL_{succ} = 11/9$ (4) $ASL_{unsucc} = 29/13$

10.

11. $ASL_{succ} = 19/12$

12. 常用构造哈希函数的方法有:

- (1) 数字分析法 该法事先需知道关键字集合,且关键字位数比散列表地址位数多,应选数字分布均匀的位。
- (2) 平方取中法 将关键字值的平方取中间几位作哈希地址。
- (3) 除留余数法 $H(key) = key \% p$, 通常 p 取小于等于表长的最大素数。
- (4) 折叠法 将关键字分成长度相等(最后一段可不等)的几部分,进行移位叠加或间界叠加,其值作哈希地址。
- (5) 基数转换法 两基数要互素,且后一基数要大于前一基数。

在哈希表中删除一个记录,在拉链法情况下可以物理地删除。在开放定址法下,不能物理地删除,只能作删除标记。该地址可能是该记录的同义词查找路径上的地址,物理的删除就中断了查找路径。因为查找时碰到空地址就认为是查找失败。

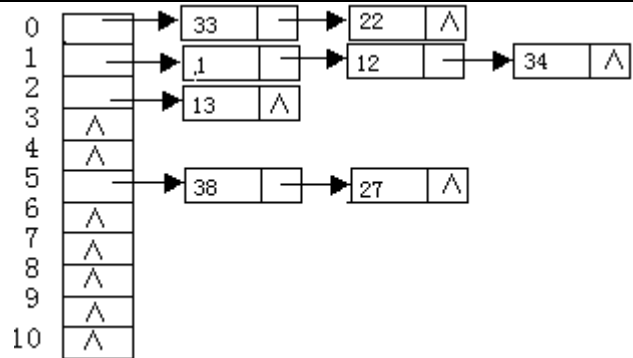
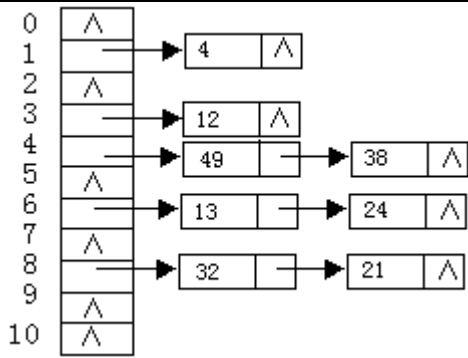
散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
关键字		14	01	68	27	55	19	20	84	79	23	11	10			
比较次数		1	2	1	4	3	1	1	3	9	1	1	3			

13. (1)

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键字		4		12	49	38	13	24	32	21	
比较次数		1		1	1	2	1	2	1	2	

$$ASL_{succ} = (1+1+1+2+1+2+1+2) / 8 = 11/8$$

$$ASL_{unsucc} = (1+2+1+8+7+6+5+4+3+2+1) / 11 = 40/11$$



(2)

13 题图 $ASL_{succ} = 11/8$ $ASL_{unsucc} = 19/11$ 14 题 (2) $ASL_{succ} = 13/8$ $ASL_{unsucc} = 19/11$

值得指出，对用拉链法求查找失败时的平均查找长度有两种观点。其一，认为比较到空指针算失败。以本题为例，哈希地址 0、2、5、7、9 和 10 均为比较 1 次失败，而哈希地址 1 和 3 比较 2 次失败，其余哈希地址均为比较 3 次失败，因此，查找失败时的平均查找长度为 $19/11$ ，我们持这种观点。还有另一种理解，他们认为只有和关键字比较才计算比较次数，而和空指针比较不计算。照这种观点，本题的 $ASL_{unsucc} = (1+1+2+2+2)/11 = 8/11$

14. 由 $hashf(x) = x \bmod 11$ 可知，散列地址空间是 0 到 10，由于有 8 个数据，装载因子取 0.7。

(1)

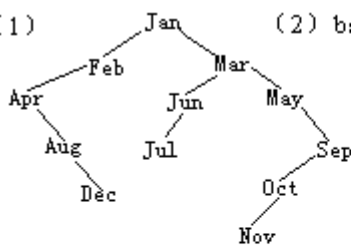
散列地址	0	1	2	3	4	5	6	7	8	9	10
关键字	33	1	13	12	34	38	27	22			
比较次数	1	1	1	3	4	1	2	8			

$ASL_{succ} = 21/8$

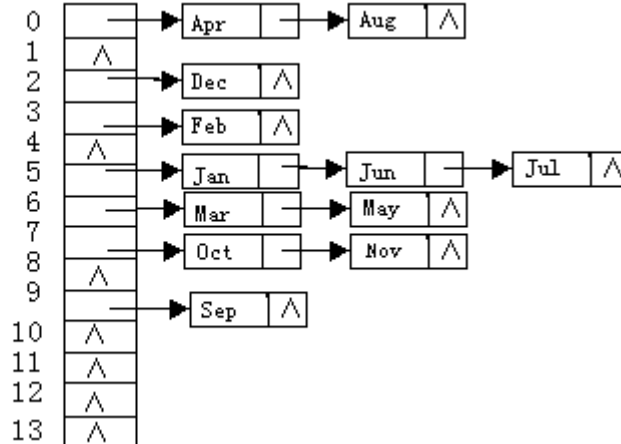
$ASL_{unsucc} = 47/11$

15.

(1)



(2) b:



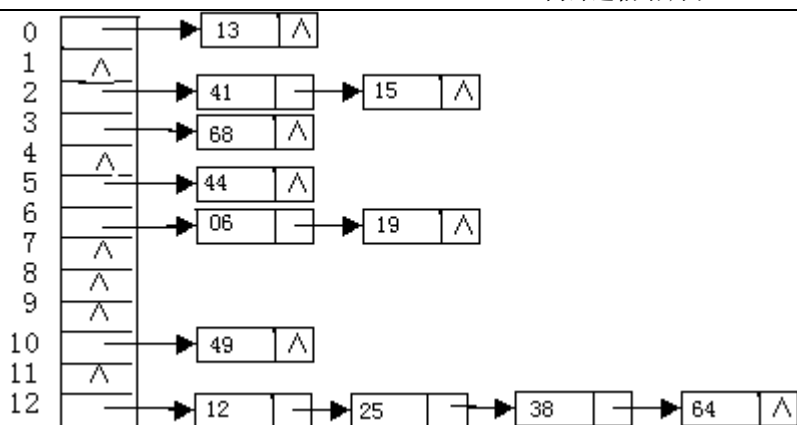
(1) $ASL = 42/12$

(2) a:

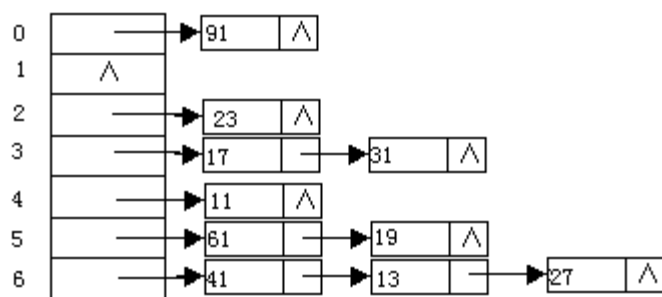
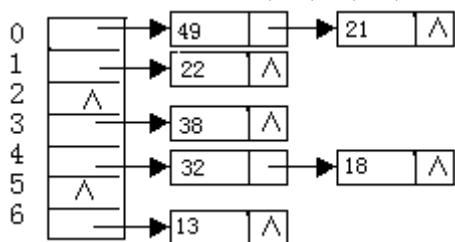
散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
关键字	Apr	Aug	Dec	Feb	Jan	Mar	May	Jun	Jul	Sep	Oct	Nov					
比较次数	1	2	1	1	1	1	2	4	5	2	5	6					

(2) a: $ASL_{succ} = 31/12$ (2) b: $ASL_{succ} = 18/12$ (注：本题 $[x]$ 取小于等于 x 的最大整数)

16.

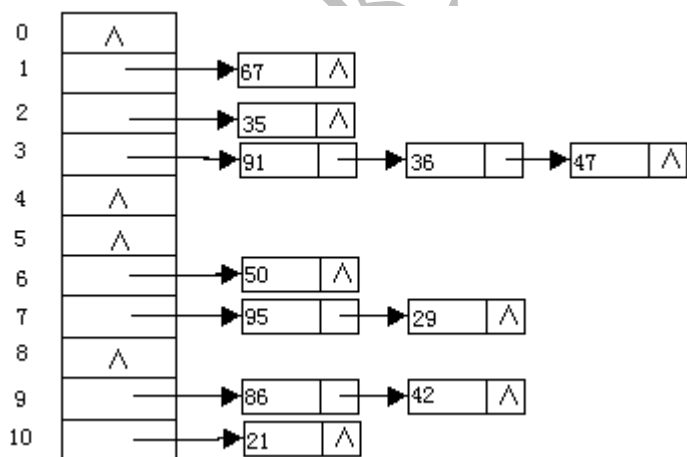


17. 查找时，对关键字 49, 22, 38, 32, 13 各比较一次，对 21, 18 各比较两次



18. $ASL_{succ} = 15/10$

19. $ASL_{succ} = 16/11$



20.

散列地址	0	1	2	3	4	5	6	7	8	9	10	11
关键字	231	89	79	25	47	16	38	82	51	39	151	
比较次数	1	1	1	1	2	1	2	3	2	4	3	

$$ASL_{succ} = 21/11$$

21.

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键字	22	33	46	13	01	67			41	53	30
比较次数	1	2	1	2	4	5			1	1	3

22.

(1)

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
关键字	32	17	63	49					24	40	10				30	31	46	47
比较次数	1	1	6	3					1	2	1				1	1	3	3

(2) 查找关键字 63, $H(k) = 63 \text{ MOD } 16 = 15$, 依次与 31, 46, 47, 32, 17, 63 比较。

(3) 查找关键字 60, $H(k) = 60 \text{ MOD } 16 = 12$, 散列地址 12 内为空, 查找失败。

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
关键字				LAN	CHA			WANG		YANG	LIU	CHANG	CAI		WEN		YUN	CHEN	ZHAO	LONG	CAO	WU	LI
比较次数				1	1			1		1	1	2	1		1		1	1	1	1	3	1	2

$$(4) ASL_{succ} = 23/11$$

23. 设用线性探测再散列解决冲突, 根据公式 $Sn1 \approx (1 + 1/(1 - \alpha)) / 2$ 。可求出负载因子为 $\alpha = 0.67$ 。再根据数据个数和装载因子, 可求出表长 $m = 15/0.67$, 取 $m = 23$ 。设哈希函数 $H(\text{key}) = (\text{关键字首尾字母在字母表中序号之和}) \text{ MOD } 23$ 。

从上表求出查找成功时的平均查找长度为 $ASL_{succ} = 19/15 < 2.0$, 满足要求。

24. (1) 哈希函数 $H(\text{key}) = (\text{关键字各字符编码之和}) \text{ MOD } 7$

(2)

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	be	cd	aa	ab	ac	ad	bd	bc	ae	ce
比较次数	1	2	1	1	1	1	1	3	3	9

25. $\alpha = 0.7$, 所以表长取 $m = 7/0.7 = 10$

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	SAT		WED			SUN	MON	TUE	THU	FRI
比较次数	6		1			1	1	2	3	4

$$ASL_{succ} = 18/7$$

$$ASL_{unsucc} = 32/10$$

26. (1)

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键字	27	53	2			31	19	20	8	18			
比较次数	3	1	1			1	1	1	1	2			

$$(2) ASL_{succ} = 11/8$$

27. (1)

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键字	0			3	29	200	32	45	58	100	10	126	400
比较次数	1			1	2	1	1	2	3	1	1	3	3

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

关键字 29 和 45 各发生一次碰撞，关键字 58, 126 和 400 各发生两次碰撞，其余关键字无碰撞。

(2)

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键字	58	10	100	3									

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	

部分答案解释如下：

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. √	2. ×	3. ×	4. ×	5. ×	6. ×	7. ×	8. ×	9. ×	10. ×	11. ×	12. ×	13. ×
14. √	15. √	16. ×	17. ×	18. ×	19. ×	20. ×	21. ×	22. ×	23. ×	24. ×	25. √	26. ×
27. √	28. ×	29. ×	30. ×	31. √								

部分答案解释如下：

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) $p \rightarrow \text{next}$

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

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) $r! = p$ (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) $r[\max] \leftarrow r[n-i+1]$
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 $\lfloor n/2 \rfloor$ 的子文件, 第二遍划分得到 4 个长度 $\lfloor n/4 \rfloor$ 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) $\text{finished} := \text{false}$ (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) $\text{sift}(r, i, n)$ (9) $r[1] := r[i]$ (10) $\text{sift}(r, 1, i-1)$
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) $\text{finish} := \text{false}$ (2) $h[i] := h[j]; i := j; j := 2*j;$ (3) $h[i] := x$ (4) h, k, n (5) $\text{sift}(h, 1, r-1)$
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. $\lceil n/m \rceil$
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比较, 即在它们之间存在着这样一个关系 $K_{S_1} \leq K_{S_2} \leq \dots \leq K_{S_n}$, 按此固有关系将 n 个记录序列重新排列为 $\{ R_{S_1}, R_{S_2}, \dots, R_{S_n} \}$ 。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	$O(d*(rd+n))$	$O(d*(rd+n))$	$O(rd)$	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。

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

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 $\text{Max} = \{\text{Max}_A, \text{Max}_B\}$ 和 $\text{Min} = \{\text{Min}_A, \text{Min}_B\}$ 。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: $C(n) = \lceil 3n/2 \rceil - 2$ 。显然, 当 $n \geq 3$ 时, $2n-3 > 3n/2-2$ 。事实上, $\lceil 3n/2 \rceil - 2$ 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, 则说明尚有二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 $\lceil \log_2 u \rceil + 1$ 。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 $\lceil \log_2(n!) \rceil$ 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 $\lceil \log_2(n!) \rceil$ 。根据斯特林公式, 有 $\lceil \log_2(n!) \rceil = O(n \log_2 n)$ 。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 $i (1 \leq i < n)$ 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 $\lceil n/2 \rceil$ 个长度为 2 的有序序列, 再进行两两归并, 得到 $\lceil n/4 \rceil$ 个长度为 4 的有序序列。如此重复, 经过 $\lceil \log_2 n \rceil$ 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 $n/2$ 。

$$\text{若不等概率, 则平均移动个数为 } \sum_{i=0}^{n-1} (n-i)/(n * (n+1)/2) * (n-i) = \frac{2n+1}{3}$$

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

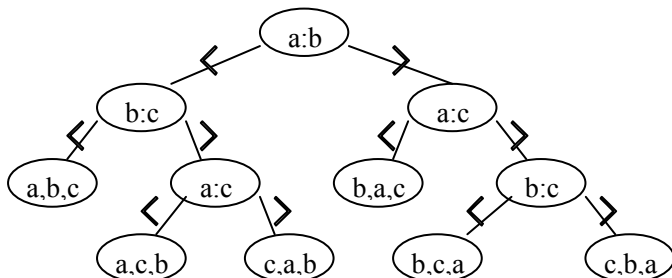
13. 平均比较次数最少: 快速排序; 占用空间最多: 归并排序; 不稳定排序算法: 快速排序、堆排序、希尔排序。

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 $2\log_2 n$ 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 $4n+2k\log_2 n$ 。

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 ($K_i=K_j, i \neq j$) 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 $x.key \leq r[j].key$ 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: ① m ② 1 ③ 0 ④ 1

(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) ① 499 ② $A[j] > A[j+1]$ ③ $b:=true$ (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 $1 \sim R_{i-1}$ (其中包括 K_j) 中关键字最大为 K_{max} ，则 $K_{max} \geq K_j$ ，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 $K_{max} \geq K_j > R_i$ ，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 $O(n^2)$ ，属于直接选择排序。

24. 6, 13, 28, 39, 41, 72, 85, 20

$i=1 \uparrow$ $m=4 \uparrow$ $r=7 \uparrow$

$20 < 39$ $i=1 \uparrow$ $m=2 \uparrow$ $r=3 \uparrow$

$20 > 13$ $i=3$ $r=3$ $m=3$

$20 < 28$ $r=2$ $i=3$

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。

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

(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

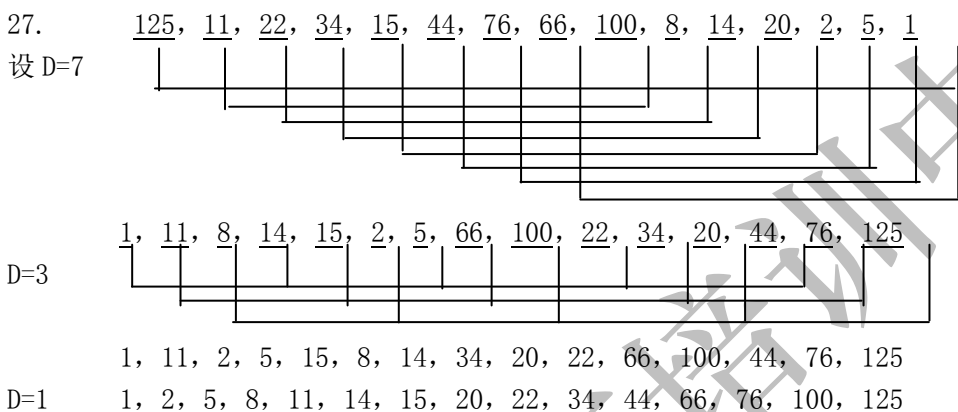
第一趟 (3) [8, 3], 2, 5, 9, 1, 6 第二趟 (2) [8, 3, 2], 5, 9, 1, 6 第三趟 (5) [8, 5, 3, 2], 9, 1, 6
第四趟 (9) [9, 8, 5, 3, 2], 1, 6 第五趟 (1) [9, 8, 5, 3, 2, 1], 6 第六趟 (6) [9, 8, 6, 5, 3, 2, 1]

(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。



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 $\lfloor \log_2 n \rfloor + 1$ ，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 $\lfloor n/2 \rfloor$ 的子文件，第二遍划分得到 4 个长度均为 $\lfloor n/4 \rfloor$ 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:

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

(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

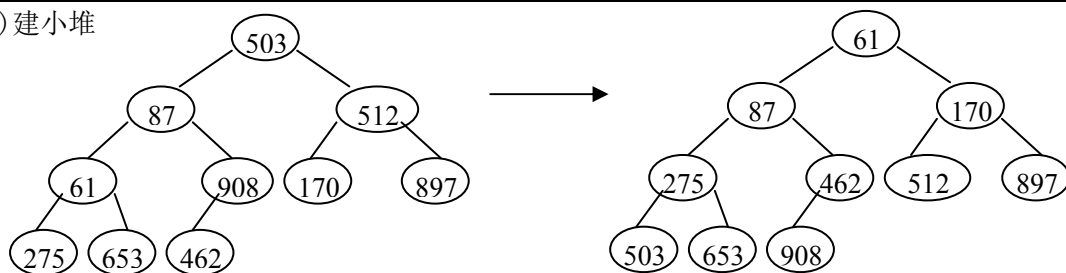
类似叙述(1): 不是堆 调成大顶堆 92, 86, 56, 70, 33, 33, 48, 65, 12

(2) ①是堆 ②不是堆 调成堆 100, 90, 80, 25, 85, 75, 60, 20, 10, 70, 65, 50

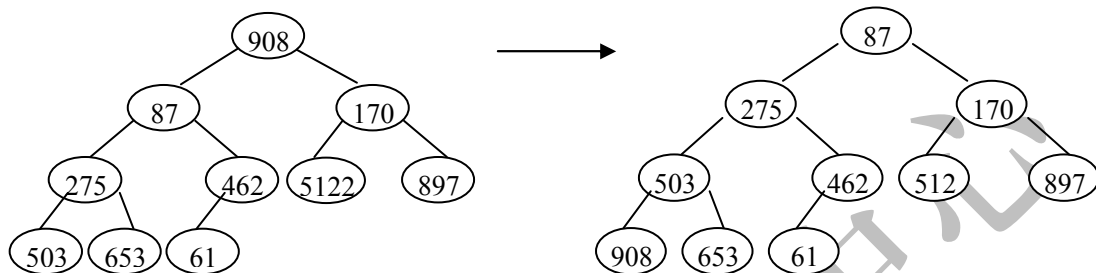
(3) ①是堆 ②不是堆 调成大堆 21, 9, 18, 8, 4, 17, 5, 6 (图略) (4): 略

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 ($k < n, k > 2$) 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

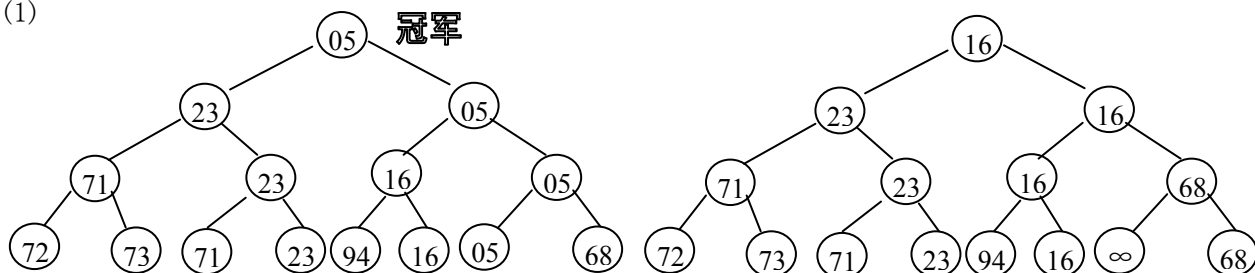
(2) 最大值元素一定是叶子结点, 在最下两层上。

(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 $\lfloor n/2 \rfloor$ 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:

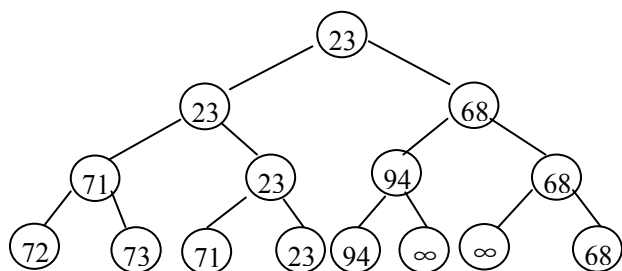
$$\sum_{i=h-1}^1 2^{i-1} \cdot 2(h-i) = \sum_{i=h-1}^1 2^i \cdot (h-i) = \sum_{j=1}^{h-1} 2^{h-j} \cdot j \leq (2n) \sum_{j=1}^{h-1} j/2^j \leq 4n$$

45. (1)



报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967



(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 $\lfloor \log_2 n \rfloor$ 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 $c=n+1, f=\lfloor c/2 \rfloor$ ，若 $K_f \leq K_c$ ，则调整完成。否则， K_f 与 K_c 交换；之后， $c=f, f=\lfloor c/2 \rfloor$ ，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 $child=child+1$ ；② $child/2$ (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

报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

(3) 链式基数排序 LSD [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

	↓	↓	↓	↓	↓
分配	30	12	4	16	8
	↓	↓		↓	↓
	10	2		6	28
	↓				↓
	20				18

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

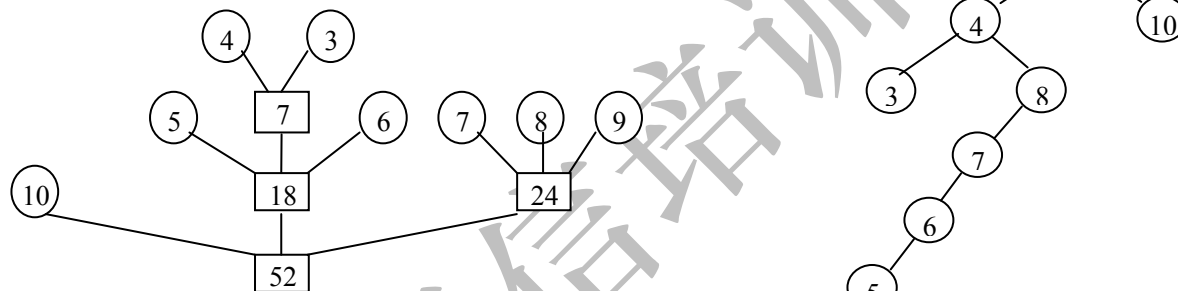
(3) 堆排序 建大堆: 58, 47, 51, 29, 18, 12, 25, 10;

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 = $2 * wpl = 202$ 次



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 $\rightarrow 321 \rightarrow 156 \rightarrow 57 \rightarrow 46 \rightarrow 28 \rightarrow 7 \rightarrow 331 \rightarrow 33 \rightarrow 34 \rightarrow 63$

分配 [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

321 33 34 156 57 28

331 63 46 7

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。

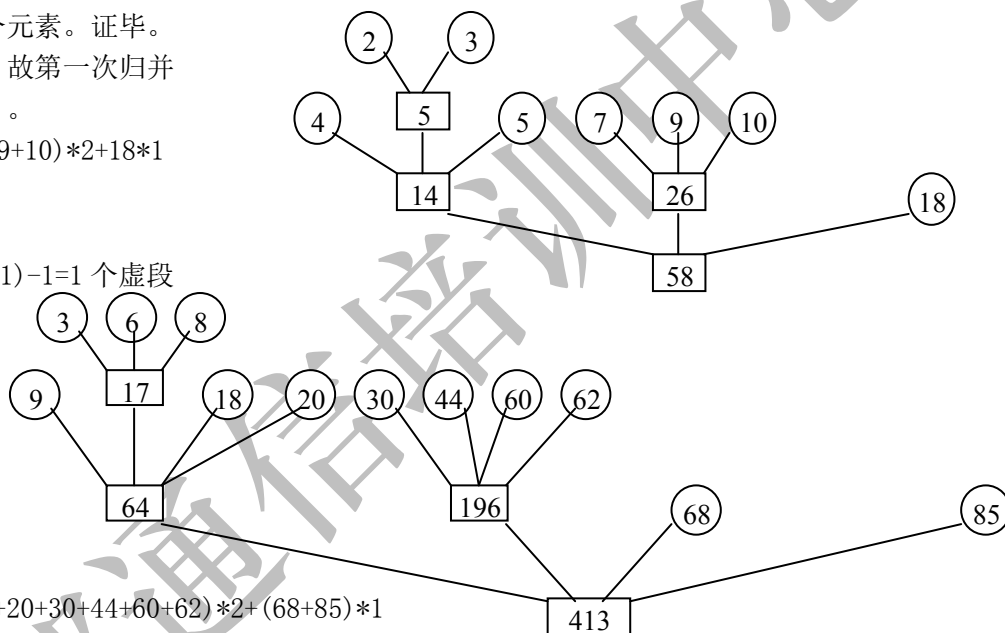
61. 因 $(8-1) \% (3-1) = 1$ ，故第一次归并

时加一个“虚段”。

$$WPL = 5 \times 3 + (4 + 5 + 7 + 9 + 10) \times 2 + 18 \times 1 \\ = 103$$

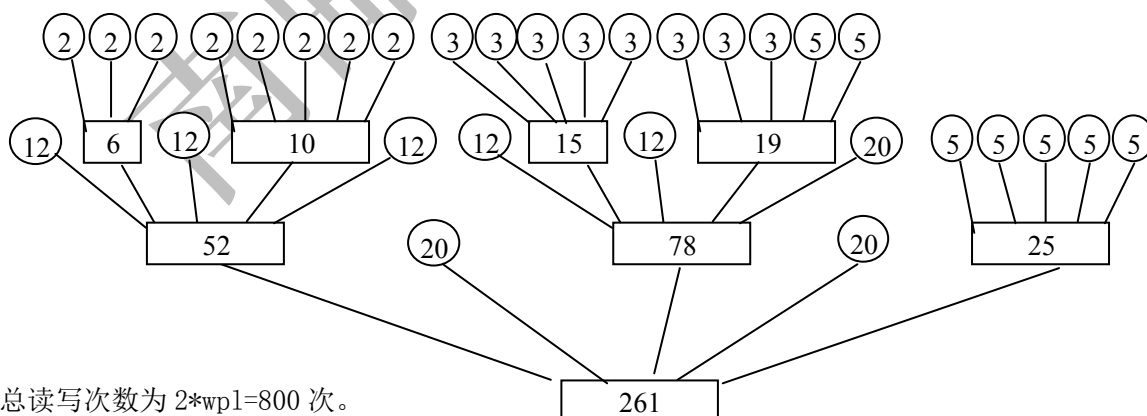
类似叙述：

(1) 加 $4 - (12-1) \% (4-1) - 1 = 1$ 个虚段



$$WPL = (3 + 6 + 8) \times 3 + (9 + 18 + 20 + 30 + 44 + 60 + 62) \times 2 + (68 + 85) \times 1 \\ = 690 \quad (2) (3) (4) \text{ 略。}$$

62. 加 $5 - (31-1) \% (5-1) - 1 = 2$ 个虚段。



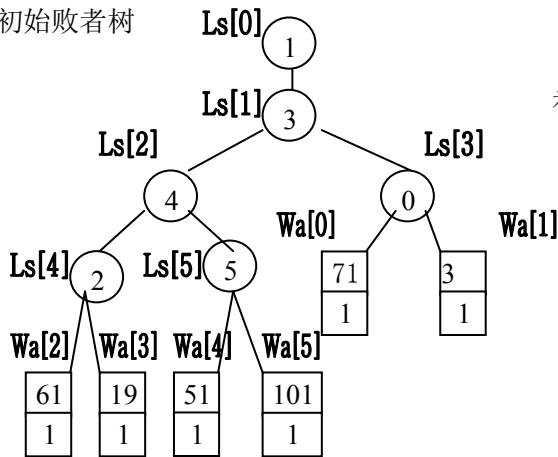
总读写次数为 $2 \times wpl = 800$ 次。

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。

$$t_1 = f_{11} + f_{10} + f_9 = 149 + 81 + 44 = 274$$

$$t_2 = f_{11} + f_{10} = 149 + 81 = 230$$

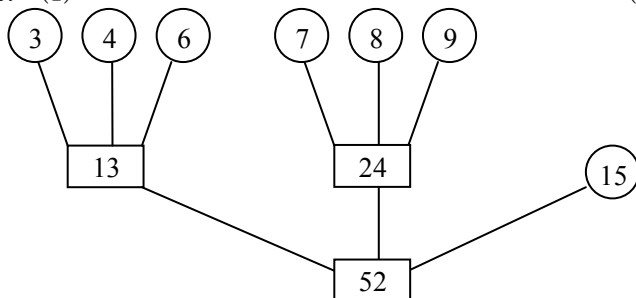
$$t_3 = f_{11} = 149$$

(2) 多步归并排序前五趟归并的情况如下:

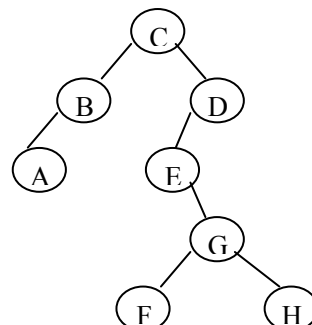
T1	$1^{149} + 1^{81} + 1^{44}$	1 步	T1	$1^{81} + 1^{44}$	2 步	T1	1^{44}	3 步	T1		4 步
T2	$1^{149} + 1^{81}$		T2	1^{81}		T2			T2	9^{44}	
T3	1^{149}		T3			T3	5^{81}		T3	5^{37}	
T4			T4	3^{149}		T4	3^{68}		T4	3^{24}	
T1	17^{24}	5 步	T1	17^{11}	注: m^p 表示 p 个长为 m 单位的归并段 1^{149} 表示 149 个长为 1 个单位的归并段。						
T2	9^{20}		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;}
    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=p->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
```

报名地址: 南京邮电大学仙林校区梅兰西街 梅苑 01101-1; 南京邮电大学三牌楼校区综合科研楼 19 层

报名热线: 025-83535877、18951896587、18951896993、18951896967

```

        {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[l..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 $\lfloor n/2 \rfloor$ 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 $h(\lfloor \log_2 n \rfloor + 1)$ 的堆, 所需进行的关键字比较的次数至多 $O(n)$, 它满足下式:

$$\begin{aligned}
 C(n) &= \sum_{i=h-1}^1 2^{i-1} \times 2(h-1) = \sum_{i=h-1}^1 2^i \times (h-1) \\
 &= 2^{h-1} + 2^{h-2} \times 2 + 2^{h-3} \times 3 + \cdots + 2 \times (h-1) \\
 &= 2^h (1/2 + 2/2^2 + 3/2^3 + \cdots + (h-1)/2^{h-1}) \\
 &\leq 2^h \times 2 \leq 2 \times 2^{(\log_2 n)+1} = 4n
 \end{aligned}$$

调整“堆顶” $n-1$ 次，总共进行的关键字比较的次数不超过： $2(\lfloor \log_2(n-1) \rfloor + \lfloor \log_2(n-2) \rfloor + \cdots + \log_2 2) < 2n(\lfloor \log_2 n \rfloor)$ 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];
  }
}

```

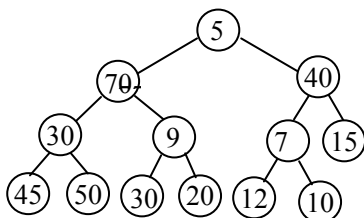
```

    r[i]=rp;
    return (i);
} // 一次划分算法结束

```

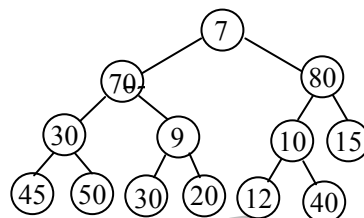
16. (1)

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次数,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= $\lfloor \log_2 n \rfloor + 1$; //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[first];
        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) 和特定砾石 (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 QkSor 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 $\lfloor \log_2 m \rfloor - \lfloor \log_2 n \rfloor < 1$ 在小堆插入，否则在大堆插入。

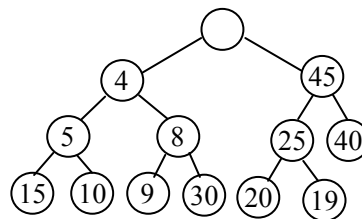
报名地址：南京邮电大学仙林校区梅兰西街 梅苑 01101-1；南京邮电大学三牌楼校区综合科研楼 19 层

报名热线：025-83535877、18951896587、18951896993、18951896967

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
{r[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
```

[illegible]

1. \checkmark	2. \checkmark	3. \times	4. \checkmark	5. \times	6. \times	7. \times	8. \times	9. \times	10. \times	11. \checkmark	
-----------------	-----------------	-------------	-----------------	-------------	-------------	-------------	-------------	-------------	--------------	------------------	--

1. 操作系统文件	数据库	2. 单关键字文件	多关键字文件
3. (1) 数据库	(2) 文本	(3) 顺序组织	(4) 随机组织
(6) 随机组织	(7) m	(8) $\lceil m/2 \rceil$	(9) 2
4. 记录	数据项	5. 串联文件	6. 第 I-1
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
12. 构造散列函数	解决冲突的方法		(5) 直接
14. 分配和释放存储空间	重组		13. 索引集
			顺序集
			数据集
			对插入的记录

5. 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。

6. 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,不需移动记录和修改指针。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。

7. 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。

8. 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。

9. 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。

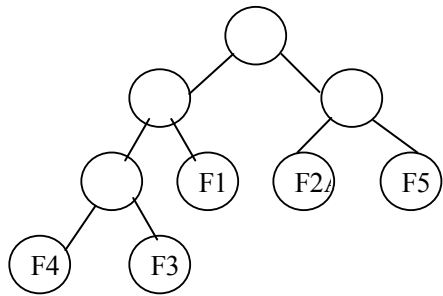
12. 排表有两项,一是次关键字值,二是具有相同次关键字值的物理记录号,这些记录号有序且顺序存储,不使用多重表中的指针链接,因而节省了空间。

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	^	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	^
09	009	李民	室主任	07	教授	^
10	010	赵松	教员	09	副教授	^
	

多重表文件