数据结构（C语言版）（第2版）
课后习题答案

李冬梅

2015.3

目　录

第 1 章　绪论

1．简述下列概念：数据、数据元素、数据项、数据对象、数据结构、逻辑结构、存储结构、抽象数据类型。

答案：

数据：是客观事物的符号表示，指所有能输入到计算机中并被计算机程序处理的符号的总称。如数学计算中用到的整数和实数，文本编辑所用到的字符串，多媒体程序处理的图形、图像、声音、动画等通过特殊编码定义后的数据。

数据元素：是数据的基本单位，在计算机中通常作为一个整体进行考虑和处理。在有些情况下，数据元素也称为元素、结点、记录等。数据元素用于完整地描述一个对象，如一个学生记录，树中棋盘的一个格局（状态）、图中的一个顶点等。

数据项：是组成数据元素的、有独立含义的、不可分割的最小单位。例如，学生基本信息表中的学号、姓名、性别等都是数据项。

数据对象：是性质相同的数据元素的集合，是数据的一个子集。例如：整数数据对象是集合 N={0，±1，±2，…}，字母字符数据对象是集合 C={‘A’，‘B’，…，‘Z’，‘a’，‘b’，…，‘z’}，学生基本信息表也可是一个数据对象。

数据结构：是相互之间存在一种或多种特定关系的数据元素的集合。换句话说，数据结构是带“结构”的数据元素的集合，“结构”就是指数据元素之间存在的关系。

逻辑结构：从逻辑关系上描述数据，它与数据的存储无关，是独立于计算机的。因此，数据的逻辑结构可以看作是从具体问题抽象出来的数学模型。

存储结构：数据对象在计算机中的存储表示，也称为**物理结构**。

抽象数据类型：由用户定义的，表示应用问题的数学模型，以及定义在这个模型上的一组操作的总称。具体包括三部分：数据对象、数据对象上关系的集合和对数据对象的基本操作的集合。

2．试举一个数据结构的例子，叙述其逻辑结构和存储结构两方面的含义和相互关系。

答案：

例如有一张学生基本信息表，包括学生的学号、姓名、性别、籍贯、专业等。每个学生基本信息记录对应一个数据元素，学生记录按顺序号排列，形成了学生基本信息记录的线性序列。对于整个表来说，只有一个开始结点(它的前面无记录)和一个终端结点(它的后面无记录)，其他的结点则各有一个也只有一个直接前趋和直接后继。学生记录之间的这种关系就确定了学生表的逻辑结构，即线性结构。

这些学生记录在计算机中的存储表示就是存储结构。如果用连续的存储单元(如用数组表示)来存放这些记录，则称为顺序存储结构；如果存储单元不连续，而是随机存放各个记录，然后用指针进行链接，则称为链式存储结构。

即相同的逻辑结构，可以对应不同的存储结构。

3．简述逻辑结构的四种基本关系并画出它们的关系图。

答案：

（1）集合结构

数据元素之间除了“属于同一集合”的关系外，别无其他关系。例如，确定一名学生是否为班级成员，只需将班级看做一个集合结构。

（2）线性结构

数据元素之间存在一对一的关系。例如，将学生信息数据按照其入学报到的时间先后顺序进行排列，将组成一个线性结构。

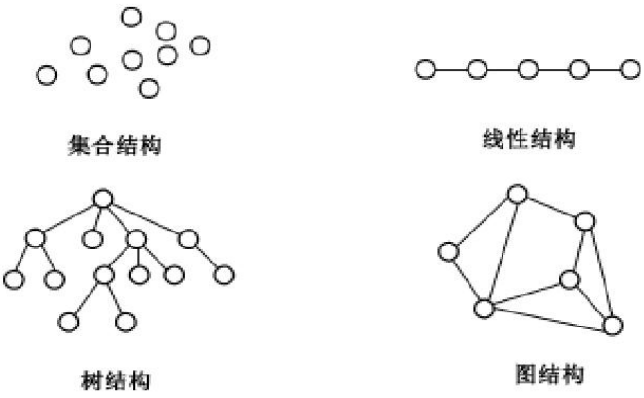
（3）树结构

数据元素之间存在一对多的关系。例如，在班级的管理体系中，班长管理多个组长，每位组长管理多名组员，从而构成树形结构。

（4）图结构或网状结构

数据元素之间存在多对多的关系。例如，多位同学之间的朋友关系，任何两位同学都可以是朋友，从而构成图形结构或网状结构。

其中树结构和图结构都属于非线性结构。



四类基本逻辑结构关系图

4．存储结构由哪两种基本的存储方法实现？

答案：

（1）顺序存储结构

顺序存储结构是借助元素在存储器中的相对位置来表示数据元素之间的逻辑关系，通常借助程序设计语言的数组类型来描述。

（2）链式存储结构

顺序存储结构要求所有的元素依次存放在一片连续的存储空间中，而链式存储结构，无需占用一整块存储空间。但为了表示结点之间的关系，需要给每个结点附加指针字段，用于存放后继元素的存储地址。所以链式存储结构通常借助于程序设计语言的指针类型来描述。

5．选择题

（1）在数据结构中，从逻辑上可以把数据结构分成（　　）。

A．动态结构和静态结构　　B．紧凑结构和非紧凑结构

C．线性结构和非线性结构　　D．内部结构和外部结构

答案：C

（2）与数据元素本身的形式、内容、相对位置、个数无关的是数据的（　　）。

A．存储结构　　B．存储实现

C．逻辑结构　　D．运算实现

答案：C

（3）通常要求同一逻辑结构中的所有数据元素具有相同的特性，这意味着（　　）。

A．数据具有同一特点

B．不仅数据元素所包含的数据项的个数要相同，而且对应数据项的类型要一致

C．每个数据元素都一样

D．数据元素所包含的数据项的个数要相等

答案：B

（4）以下说法正确的是（　　）。

A．数据元素是数据的最小单位

B．数据项是数据的基本单位

C．数据结构是带有结构的各数据项的集合

D．一些表面上很不相同的数据可以有相同的逻辑结构

答案：D

解释：数据元素是数据的基本单位，数据项是数据的最小单位，数据结构是带有结构的各数据元素的集合。

（5）算法的时间复杂度取决于（　　）。

A．问题的规模　　B．待处理数据的初态

C．计算机的配置　　D．A 和 B

答案：D

解释：算法的时间复杂度不仅与问题的规模有关，还与问题的其他因素有关。如某些排序的算法，其执行时间与待排序记录的初始状态有关。为此，有时会对算法有最好、最坏以及平均时间复杂度的评价。

（6）以下数据结构中，（　）是非线性数据结构

A．树　　B．字符串　　C．队列　　D．栈

答案：A

6．试分析下面各程序段的时间复杂度。

（1）
```
x=90; y=100;
while(y>0)
    if(x>100)
     {x=x-10;y--;}
    else x++;
```

答案：O(1)

解释：程序的执行次数为常数阶。

（2）
```
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
      a[i][j]=0;
```
答案：$O(m*n)$

解释：语句 a[i][j]=0;的执行次数为 m*n。

（3）
```
s=0;
for i=0; i<n; i++)
  for(j=0; j<n; j++)
    s+=B[i][j];
sum=s;
```
答案：$O(n^2)$

解释：语句 s+=B[i][j];的执行次数为 n^2。

（4）
```
i=1;
while(i<=n)
    i=i*3;
```
答案：$O(\log_3 n)$

解释：语句 i=i*3;的执行次数为 $\lfloor \log_3 n \rfloor$。

（5）
```
x=0;
for(i=1; i<n; i++)
  for (j=1; j<=n-i; j++)
    x++;
```
答案：$O(n^2)$

解释：语句 x++;的执行次数为 n-1+n-2+……＋1= n(n-1)/2。

（6）
```
x=n; //n>1
y=0;
while(x≥(y+1)* (y+1))
  y++;
```
答案：$O(\sqrt{n})$

解释：语句 y++;的执行次数为 $\lfloor \sqrt{n} \rfloor$。

第 2 章　线性表

1．选择题

（1）顺序表中第一个元素的存储地址是 100，每个元素的长度为 2，则第 5 个元素的地址是（　　）。

A．110　　B．108　　C．100　　D．120

答案：B

解释：顺序表中的数据连续存储，所以第 5 个元素的地址为：100+2*4=108。

（2）在 n 个结点的顺序表中，算法的时间复杂度是 O(1)的操作是（　　）。

A．访问第 i 个结点（1≤i≤n）和求第 i 个结点的直接前驱（2≤i≤n）

B．在第 i 个结点后插入一个新结点（1≤i≤n）

C．删除第 i 个结点（1≤i≤n）

D．将 n 个结点从小到大排序

答案：A

解释：在顺序表中插入一个结点的时间复杂度都是 $O(n^2)$，排序的时间复杂度为 $O(n^2)$ 或 $O(n\log_2 n)$。顺序表是一种随机存取结构，访问第 i 个结点和求第 i 个结点的直接前驱都可以直接通过数组的下标直接定位，时间复杂度是 O(1)。

（3） 向一个有 127 个元素的顺序表中插入一个新元素并保持原来顺序不变，平均要移动__的元素个数为（　　）。

A．8　　B．63.5　　C．63　　D．7

答案：B

解释：平均要移动的元素个数为：n/2。

（4）链接存储的存储结构所占存储空间（　　）。

A．分两部分，一部分存放结点值，另一部分存放表示结点间关系的指针

B．只有一部分，存放结点值

C．只有一部分，存储表示结点间关系的指针

D．分两部分，一部分存放结点值，另一部分存放结点所占单元数

答案：A

（5）线性表若采用链式存储结构时，要求内存中可用存储单元的地址（　　）。

A．必须是连续的　　B．部分地址必须是连续的

C．一定是不连续的　　D．连续或不连续都可以

答案：D

（6）线性表 L 在（　　）情况下适用于使用链式结构实现。

A．需经常修改 L 中的结点值　　B．需不断对 L 进行删除插入

C．L 中含有大量的结点　　D．L 中结点结构复杂

答案：B

解释：链表最大的优点在于插入和删除时不需要移动数据，直接修改指针即可。

（7）单链表的存储密度（　　）。

A．大于 1　　B．等于 1　　C．小于 1　　D．不能确定

答案：C

解释：存储密度是指一个结点数据本身所占的存储空间和整个结点所占的存储空间之比，假设单链表一个结点本身所占的空间为 D，指针域所占的空间为 N，则存储密度为：D/(D+N)，一定小于 1。

（8）将两个各有 n 个元素的有序表归并成一个有序表，其最少的比较次数是（　　）。

A．n　　B．2n-1　　C．2n　　D．n-1

答案：A

解释：当第一个有序表中所有的元素都小于（或大于）第二个表中的元素，只需要用第二个表中的第一个元素依次与第一个表的元素比较，总计比较 n 次。

（9）在一个长度为 n 的顺序表中，在第 i 个元素（1≤i≤n+1）之前插入一个新元素时须向后移动（　　）个元素。

A．n-i　　B．n-i+1　　C．n-i-1　　D．I

答案：B

(10) 线性表 L=(a_1，a_2,……a_n)，下列说法正确的是（　　）。

A．每个元素都有一个直接前驱和一个直接后继

B．线性表中至少有一个元素

C．表中诸元素的排列必须是由小到大或由大到小

D．除第一个和最后一个元素外，其余每个元素都有一个且仅有一个直接前驱和直接后继。

答案：D

(11) 创建一个包括 *n* 个结点的有序单链表的时间复杂度是（　　）。

A．O(1)　　B．O(n)　　C．O(n^2)　　D．O($n\log_2 n$)

答案：C

解释：单链表创建的时间复杂度是 O(n)，而要建立一个有序的单链表，则每生成一个新结点时需要和已有的结点进行比较，确定合适的插入位置，所以时间复杂度是 O(n2)。

(12) 以下说法错误的是（　　）。

A．求表长、定位这两种运算在采用顺序存储结构时实现的效率不比采用链式存储结构时实现的效率低

B．顺序存储的线性表可以随机存取

C．由于顺序存储要求连续的存储区域，所以在存储管理上不够灵活

D．线性表的链式存储结构优于顺序存储结构

答案：D

解释：链式存储结构和顺序存储结构各有优缺点，有不同的适用场合。

(13) 在单链表中，要将 s 所指结点插入到 p 所指结点之后，其语句应为（　　）。

A．s->next=p+1; p->next=s;

B．(*p).next=s; (*s).next=(*p).next;

C．s->next=p->next; p->next=s->next;

D．s->next=p->next; p->next=s;

答案：D

(14) 在双向链表存储结构中，删除 p 所指的结点时须修改指针（　　）。

A．p->next->prior=p->prior; p->prior->next=p->next;

B．p->next=p->next->next; p->next->prior=p;

C．p->prior->next=p; p->prior=p->prior->prior;

D．p->prior=p->next->next; p->next=p->prior->prior;

答案：A

(15) 在双向循环链表中，在 p 指针所指的结点后插入 q 所指向的新结点，其修改指针的操作是（　　）。

A．p->next=q; q->prior=p; p->next->prior=q; q->next=q;

B．p->next=q; p->next->prior=q; q->prior=p; q->next=p->next;

C．q->prior=p; q->next=p->next; p->next->prior=q; p->next=q;

D．q->prior=p; q->next=p->next; p->next=q; p->next->prior=q;

答案：C

2．算法设计题

（1）将两个递增的有序链表合并为一个递增的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其它的存储空间。表中不允许有重复的数据。

[题目分析]

合并后的新表使用头指针 Lc 指向，pa 和 pb 分别是链表 La 和 Lb 的工作指针,初始化为相应链表的第一个结点，从第一个结点开始进行比较，当两个链表 La 和 Lb 均为到达表尾结点时，依次摘取其中较小者重新链接在 Lc 表的最后。如果两个表中的元素相等，只摘取 La 表中的元素，删除 Lb 表中的元素，这样确保合并后表中无重复的元素。当一个表到达表尾结点，为空时，将非空表的剩余元素直接链接在 Lc 表的最后。

[算法描述]

```
void MergeList(LinkList &La,LinkList &Lb,LinkList &Lc)
{//合并链表 La 和 Lb，合并后的新表使用头指针 Lc 指向
  pa=La->next;  pb=Lb->next;
   //pa 和 pb 分别是链表 La 和 Lb 的工作指针,初始化为相应链表的第一个结点
   Lc=pc=La;  //用 La 的头结点作为 Lc 的头结点
   while(pa && pb)
   {if(pa->data<pb->data){pc->next=pa;pc=pa;pa=pa->next;}
     //取较小者 La 中的元素，将 pa 链接在 pc 的后面，pa 指针后移
     else if(pa->data>pb->data) {pc->next=pb; pc=pb; pb=pb->next;}
      //取较小者 Lb 中的元素，将 pb 链接在 pc 的后面，pb 指针后移
```

```
        else //相等时取 La 中的元素，删除 Lb 中的元素
        {pc->next=pa;pc=pa;pa=pa->next;
         q=pb->next;delete pb ;pb =q;
         }
        }
       pc->next=pa?pa:pb;     //插入剩余段
        delete Lb;             //释放 Lb 的头结点
    }
```

（2）将两个非递减的有序链表合并为一个非递增的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其它的存储空间。表中允许有重复的数据。

[题目分析]

合并后的新表使用头指针 Lc 指向，pa 和 pb 分别是链表 La 和 Lb 的工作指针,初始化为相应链表的第一个结点，从第一个结点开始进行比较，当两个链表 La 和 Lb 均为到达表尾结点时，依次摘取其中较小者重新链接在 Lc 表的表头结点之后，如果两个表中的元素相等，只摘取 La 表中的元素，保留 Lb 表中的元素。当一个表到达表尾结点，为空时，将非空表的剩余元素依次摘取，链接在 Lc 表的表头结点之后。

[算法描述]

```
void MergeList(LinkList& La, LinkList& Lb, LinkList& Lc, )
{//合并链表 La 和 Lb，合并后的新表使用头指针 Lc 指向
  pa=La->next;  pb=Lb->next;
//pa 和 pb 分别是链表 La 和 Lb 的工作指针,初始化为相应链表的第一个结点
  Lc=pc=La; //用 La 的头结点作为 Lc 的头结点
  Lc->next=NULL;
  while(pa||pb )
{//只要存在一个非空表，用 q 指向待摘取的元素
    if(!pa)  {q=pb;  pb=pb->next;}
//La 表为空，用 q 指向 pb，pb 指针后移
    else if(!pb)  {q=pa;  pa=pa->next;}
//Lb 表为空，用 q 指向 pa，pa 指针后移
    else if(pa->data<=pb->data)  {q=pa;  pa=pa->next;}
//取较小者（包括相等）La 中的元素，用 q 指向 pa，pa 指针后移
    else {q=pb;  pb=pb->next;}
//取较小者 Lb 中的元素，用 q 指向 pb，pb 指针后移
     q->next = Lc->next;  Lc->next = q;
//将 q 指向的结点插在 Lc 表的表头结点之后
    }
    delete Lb;             //释放 Lb 的头结点
}
```

（3）已知两个链表 A 和 B 分别表示两个集合，其元素递增排列。请设计算法求出 A 与 B 的交集，并存放于 A 链表中。

[题目分析]

只有同时出现在两集合中的元素才出现在结果表中,合并后的新表使用头指针 Lc 指向。pa 和 pb 分别是链表 La 和 Lb 的工作指针,初始化为相应链表的第一个结点，从第一个结点开始进行比较，当两个链表 La 和 Lb 均为到达表尾结点时，如果两个表中相等的元素时，摘取 La 表中的元素，删除 Lb 表中的元素；如果其中一个表中的元素较小时，删除此表中较小的元素，此表的工作指针后移。当链表 La 和 Lb 有一个到达表尾结点，为空时，依次删除另一个非空表中的所有元素。

[算法描述]

```
void Mix(LinkList& La, LinkList& Lb, LinkList& Lc)
{  pa=La->next;pb=Lb->next;
   pa 和 pb 分别是链表 La 和 Lb 的工作指针,初始化为相应链表的第一个结点
   Lc=pc=La; //用 La 的头结点作为 Lc 的头结点
   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; delete u;}
     else if(pa->data<pb->data) {u=pa;pa=pa->next; delete u;}
     else {u=pb; pb=pb->next; delete u;}
   }
   while(pa) {u=pa; pa=pa->next; delete u;}// 释放结点空间
   while(pb) {u=pb; pb=pb->next; delete u;}//释放结点空间
   pc->next=null;//置链表尾标记。
   delete Lb;  //释放 Lb 的头结点
}
```

（4）已知两个链表 A 和 B 分别表示两个集合，其元素递增排列。请设计算法求出两个集合 A 和 B 的差集（即仅由在 A 中出现而不在 B 中出现的元素所构成的集合），并以同样的形式存储，同时返回该集合的元素个数。

[题目分析]

求两个集合 A 和 B 的差集是指在 A 中删除 A 和 B 中共有的元素，即删除链表中的相应结点,所以要保存待删除结点的前驱，使用指针 pre 指向前驱结点。pa 和 pb 分别是链表 La 和 Lb 的工作指针,初始化为相应链表的第一个结点，从第一个结点开始进行比较，当两个链表 La 和 Lb 均为到达表尾结点时，如果 La 表中的元素小于 Lb 表中的元素，pre 置为 La 表的工作指针 pa 删除 Lb 表中的元素；如果其中一个表中的元素较小时，删除此表中较小的元素，此表的工作指针后移。当链表 La 和 Lb 有一个为空时，依次删除另一个非空表中的所有元素。

[算法描述]

```
void Difference（LinkList& La, LinkList& Lb,int *n）
{//差集的结果存储于单链表 La 中，*n 是结果集合中元素个数，调用时为 0
  pa=La->next; pb=Lb->next;
  //pa 和 pb 分别是链表 La 和 Lb 的工作指针,初始化为相应链表的第一个结点
  pre=La;            //pre 为 La 中 pa 所指结点的前驱结点的指针
  while（pa&&pb）
  {if（pa->data<q->data）{pre=pa;pa=pa->next;*n++;}
    // A 链表中当前结点指针后移
    else if（pa->data>q->data）q=q->next;      //B 链表中当前结点指针后移
    else {pre->next=pa->next;       //处理 A，B 中元素值相同的结点，应删除
          u=pa; pa=pa->next; delete u;}    //删除结点
   }
}
```

（5）设计算法将一个带头结点的单链表 A 分解为两个具有相同结构的链表 B、C，其中 B 表的结点为 A 表中值小于零的结点，而 C 表的结点为 A 表中值大于零的结点（链表 A 中的元素为非零整数，要求 B、C 表利用 A 表的结点）。

[题目分析]

B 表的头结点使用原来 A 表的头结点，为 C 表新申请一个头结点。从 A 表的第一个结点开始，依次取其每个结点 p，判断结点 p 的值是否小于 0，利用前插法，将小于 0 的结点插入 B 表,大于等于 0 的结点插入 C 表。

[算法描述]

```
void DisCompose(LinkedList A)
 {   B=A;
     B->next= NULL;     //B 表初始化
     C=new 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
     while(p!= NULL)
     { r=p->next;       //暂存 p 的后继
       if(p->data<0)
        {p->next=B->next; B->next=p; }//将小于 0 的结点链入 B 表,前插法
       else {p->next=C->next; C->next=p; }//将大于等于 0 的结点链入 C 表,前插法
       p=r;//p 指向新的待处理结点。
     }
   }
```

（6）设计一个算法，通过一趟遍历在单链表中确定值最大的结点。

[题目分析]

假定第一个结点中数据具有最大值，依次与下一个元素比较，若其小于下一个元素，则设其下一个元素为最大值，反复进行比较，直到遍历完该链表。

[算法描述]

```
ElemType Max (LinkList L ){
    if(L->next==NULL) return NULL;
    pmax=L->next; //假定第一个结点中数据具有最大值
    p=L->next->next;
    while(p != NULL ){//如果下一个结点存在
        if(p->data > pmax->data) pmax=p;//如果 p 的值大于 pmax 的值，则重新赋值
        p=p->next;//遍历链表
    }
    return pmax->data;
```

（7）设计一个算法，通过遍历一趟，将链表中所有结点的链接方向逆转，仍利用原表的存储空间。

[题目分析]

从首元结点开始，逐个地把链表 L 的当前结点 p 插入新的链表头部。

[算法描述]

```
void  inverse(LinkList &L)
{// 逆置带头结点的单链表 L
    p=L->next;  L->next=NULL;
    while ( p) {
        q=p->next;     // q 指向*p 的后继
        p->next=L->next;
        L->next=p;        // *p 插入在头结点之后
        p = q;
    }
}
```

（8）设计一个算法，删除递增有序链表中值大于 mink 且小于 maxk 的所有元素（mink 和 maxk 是给定的两个参数，其值可以和表中的元素相同，也可以不同 ）。

[题目分析]

分别查找第一个值>mink 的结点和第一个值 ≥maxk 的结点，再修改指针，删除值大于 mink 且小于 maxk 的所有元素。

[算法描述]

```
void delete(LinkList &L, int mink, int maxk) {
   p=L->next; //首元结点
   while (p && p->data<=mink)
      { pre=p;  p=p->next; } //查找第一个值>mink 的结点
   if (p)
```

```
{while (p && p->data<maxk)  p=p->next;
                    // 查找第一个值 ≥maxk 的结点
  q=pre->next;   pre->next=p;  // 修改指针
  while (q!=p)
     { s=q->next;  delete q;  q=s; } // 释放结点空间
 }//if
}
```

（9）已知 p 指向双向循环链表中的一个结点，其结点结构为 data、prior、next 三个域，写出算法 change(p)，交换 p 所指向的结点和它的前缀结点的顺序。

[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。

[算法描述]

```
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。
```

（10）已知长度为 n 的线性表 A 采用顺序存储结构，请写一时间复杂度为 O(n)、空间复杂度为 O(1)的算法，该算法删除线性表中所有值为 item 的数据元素。

[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。

[算法描述]

```
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--；//若右端元素为 item，指针左移
    if（i<j）A[i++]=A[j--]；}
```

第 3 章　栈和队列

1．选择题

（1）若让元素 1，2，3，4，5 依次进栈，则出栈次序不可能出现在（　）种情况。

A．5，4，3，2，1　B．2，1，5，4，3　C．4，3，1，2，5　D．2，3，5，4，1

答案：C

解释：栈是后进先出的线性表，不难发现 C 选项中元素 1 比元素 2 先出栈，违背了栈的后进先出原则，所以不可能出现 C 选项所示的情况。

（2）若已知一个栈的入栈序列是 1，2，3，…，n，其输出序列为 p1，p2，p3，…，pn，若 p1=n，则 pi 为（　）。

A．i　B．n-i　C．n-i+1　D．不确定

答案：C

解释：栈是后进先出的线性表，一个栈的入栈序列是 1，2，3，…，n，而输出序列的第一个元素为 n，说明 1，2，3，…，n 一次性全部进栈，再进行输出，所以 p1=n，p2=n-1，…，pi=n-i+1。

（3）数组 Q［n］用来表示一个循环队列，f 为当前队列头元素的前一位置，r 为队尾元素的位置，假定队列中元素的个数小于 n，计算队列中元素个数的公式为（　）。

A．r-f　B．(n+f-r)%n　C．n+r-f　D．(n+r-f)%n

答案：D

解释：对于非循环队列，尾指针和头指针的差值便是队列的长度，而对于循环队列，差值可能为负数，所以需要将差值加上 MAXSIZE（本题为 n），然后与 MAXSIZE（本题为 n）求余，即（n+r-f)%n。

（4）链式栈结点为：(data,link)，top 指向栈顶.若想摘除栈顶结点，并将删除结点的值保存到 x 中,则应执行操作（　）。

A．x=top->data;top=top->link;　B．top=top->link;x=top->link;

C．x=top;top=top->link;　D．x=top->link;

答案：A

解释：x=top->data 将结点的值保存到 x 中，top=top->link 栈顶指针指向栈顶下一结点，即摘除栈顶结点。

（5）设有一个递归算法如下

```
int fact(int n) {  //n 大于等于 0
    if(n<=0) return 1;
    else return n*fact(n-1);        }
```

则计算 fact(n)需要调用该函数的次数为（　）。

A．n+1　B．n-1　C．n　D．n+2

答案：A

解释：特殊值法。设 n=0，易知仅调用一次 fact(n)函数，故选 A。

（6）栈在（　）中有所应用。

A．递归调用　　B．函数调用　　C．表达式求值　　D．前三个选项都有

答案：D

解释：递归调用、函数调用、表达式求值均用到了栈的后进先出性质。

（7）为解决计算机主机与打印机间速度不匹配问题，通常设一个打印数据缓冲区。主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是（　）。

A．队列　　B．栈　　C．线性表　　D．有序表

答案：A

解释：解决缓冲区问题应利用一种先进先出的线性表，而队列正是一种先进先出的线性表。

（8）设栈 S 和队列 Q 的初始状态为空，元素 e1、e2、e3、e4、e5 和 e6 依次进入栈 S，一个元素出栈后即进入 Q，若 6 个元素出队的序列是 e2、e4、e3、e6、e5 和 e1，则栈 S 的容量至少应该是（　）。

A．2　　B．3　　C．4　　D．6

答案：B

解释：元素出队的序列是 e2、e4、e3、e6、e5 和 e1，可知元素入队的序列是 e2、e4、e3、e6、e5 和 e1，即元素出栈的序列也是 e2、e4、e3、e6、e5 和 e1，而元素 e1、e2、e3、e4、e5 和 e6 依次进入栈，易知栈 S 中最多同时存在 3 个元素，故栈 S 的容量至少为 3。

（9）若一个栈以向量 V[1..n]存储，初始栈顶指针 top 设为 n+1，则元素 x 进栈的正确操作是(　　)。

A．top++; V[top]=x;　　B．V[top]=x; top++;

C．top--; V[top]=x;　　D．V[top]=x; top--;

答案：C

解释：初始栈顶指针 top 为 n+1，说明元素从数组向量的高端地址进栈，又因为元素存储在向量空间 V[1..n]中，所以进栈时 top 指针先下移变为 n，之后将元素 x 存储在 V[n]。

（10）设计一个判别表达式中左，右括号是否配对出现的算法，采用（　）数据结构最佳。

A．线性表的顺序存储结构　　B．队列

C．线性表的链式存储结构　　D．栈

答案：D

解释：利用栈的后进先出原则。

（11）用链接方式存储的队列，在进行删除运算时（　）。

A．仅修改头指针　　B．仅修改尾指针

C．头、尾指针都要修改　　D．头、尾指针可能都要修改

答案：D

解释：一般情况下只修改头指针，但是，当删除的是队列中最后一个元素时，队尾指针也丢失了，因此需对队尾指针重新赋值。

（12）循环队列存储在数组 A[0..m]中，则入队时的操作为（　）。

A. rear=rear+1　　B. rear=(rear+1)%(m-1)

C. rear=(rear+1)%m　　D. rear=(rear+1)%(m+1)

答案：D

解释：数组 A[0..m]中共含有 m+1 个元素，故在求模运算时应除以 m+1。

（13）最大容量为 n 的循环队列，队尾指针是 rear，队头是 front，则队空的条件是（　）。

A. (rear+1)%n==front　　B. rear==front

C. rear+1==front　　D. (rear-1)%n==front

答案：B

解释：最大容量为 n 的循环队列，队满条件是(rear+1)%n==front，队空条件是 rear==front。

（14）栈和队列的共同点是（　）。

A. 都是先进先出　　B. 都是先进后出

C. 只允许在端点处插入和删除元素　　D. 没有共同点

答案：C

解释：栈只允许在栈顶处进行插入和删除元素，队列只允许在队尾插入元素和在队头删除元素。

（15）一个递归算法必须包括（　）。

A. 递归部分　　B. 终止条件和递归部分

C. 迭代部分　　D. 终止条件和迭代部分

答案：B

2．算法设计题

（1）将编号为 0 和 1 的两个栈存放于一个数组空间 V[m]中，栈底分别处于数组的两端。当第 0 号栈的栈顶指针 top[0]等于-1 时该栈为空，当第 1 号栈的栈顶指针 top[1]等于 m 时该栈为空。两个栈均从两端向中间增长。试编写双栈初始化，判断栈空、栈满、进栈和出栈等算法的函数。双栈数据结构的定义如下：

```
Typedef struct
{int top[2],bot[2];          //栈顶和栈底指针
 SElemType *V;               //栈数组
 int m;                      //栈最大可容纳元素个数
}DblStack
```

[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，左栈顶指针为-1，右栈顶为 m。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。

[算法描述]

(1) 栈初始化

```
int Init()
 {S.top[0]=-1;
  S.top[1]=m;
  return 1; //初始化成功
}
```

(2) 入栈操作：

```
int push(stk S ,int i,int x)
//i 为栈号，i=0 表示左栈，i=1 为右栈，x 是入栈元素。入栈成功返回 1，失败返回 0
{if(i<0||i>1){ cout<<“栈号输入不对”<<endl;exit(0);}
if(S.top[1]-S.top[0]==1) {cout<<“栈已满”<<endl;return(0);}
switch(i)
 {case 0: S.V[++S.top[0]]=x; return(1); break;
  case 1: S.V[--S.top[1]]=x; return(1);
  }
}//push
```

(3) 退栈操作

```
ElemType pop(stk S,int i)
//退栈。i 代表栈号，i=0 时为左栈，i=1 时为右栈。退栈成功时返回退栈元素
//否则返回-1
{if(i<0 || i>1){cout<<“栈号输入错误”<<endl； exit(0);}
 switch(i)
  {case 0: if(S.top[0]==-1) {cout<<“栈空”<<endl； return（-1）；}
    else return(S.V[S.top[0]--]);
    case 1: if(S.top[1]==m { cout<<“栈空”<<endl; return(-1);}
    else return(S.V[S.top[1]++]);
   }//switch
}//算法结束
```

(4) 判断栈空

```
int Empty();
{return (S.top[0]==-1 && S.top[1]==m);
}
```

[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。左栈是通常意义下的栈，而右栈入栈操作时，其栈顶指针左移（减 1），退栈时，栈顶指针右移（加 1）。

（2）回文是指正读反读均相同的字符序列，如“abba”和“abdba”均是回文，但“good”不是回文。试写一个算法判定给定的字符向量是否为回文。(提示：将一半字符入栈)

[题目分析]

将字符串前一半入栈，然后，栈中元素和字符串后一半进行比较。即将第一个出栈元素和后一半串中第一个字符比较，若相等，则再出栈一个元素与后一个字符比较，……，直至栈空，结论为字符序列是回文。在出栈元素与串中字符比较不等时，结论字符序列不是回文。

[算法描述]

```
#define StackSize 100 //假定预分配的栈空间最多为 100 个元素
typedef char DataType;//假定栈元素的数据类型为字符
typedef struct
{DataType data[StackSize];
 int top;
}SeqStack;

int IsHuiwen( char *t)
{//判断 t 字符向量是否为回文，若是，返回 1，否则返回 0
 SeqStack s;
 int i , len;
 char temp;
 InitStack( &s);
 len=strlen(t); //求向量长度
 for ( i=0; i<len/2; i++)//将一半字符入栈
    Push( &s, t[i]);
 while( !EmptyStack( &s))
 {// 每弹出一个字符与相应字符比较
  temp=Pop (&s);
  if( temp!=S[i])   return 0 ;// 不等则返回 0
  else i++;
 }
 return 1 ; // 比较完毕均相等则返回 1
}
```

（3）设从键盘输入一整数的序列：a_1, a_2, a_3，…，a_n，试编写算法实现：用栈结构存储输入的整数，当 $a_i \neq -1$ 时，将 a_i 进栈；当 $a_i=-1$ 时，输出栈顶整数并出栈。算法应对异常情况（入栈满等）给出相应的信息。

[算法描述]

```
#define maxsize 栈空间容量
void InOutS(int s[maxsize])
//s 是元素为整数的栈，本算法进行入栈和退栈操作。
```

```
{int top=0;                    //top 为栈顶指针，定义 top=0 时为栈空。
 for(i=1; i<=n; i++)           //n 个整数序列作处理。
    {cin>>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){cout<<“栈满”<<endl;exit(0);}
      else s[++top]=x; //x 入栈。
     }
     else   //读入的整数等于-1 时退栈。
     {if(top==0){ cout<<“栈空”<<endl;exit(0);}
      else cout<<“出栈元素是”<< s[top--]<<endl;}
     }
}//算法结束。
```

（4）从键盘上输入一个后缀表达式，试编写算法计算表达式的值。规定：逆波兰表达式的长度不超过一行，以$符作为输入结束，操作数之间用空格分隔,操作符只可能有+、-、*、/四种运算。例如：234 34+2*$。

[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。

[算法描述]

```
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。
 cin>>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’ )) ; cin>>x;}
                else           //处理小数部分。
                {scale=10.0; cin>>x;
                while(x>=’ 0’ &&x<=’ 9’ )
                {num=num+(ord(x)-ord( ‘0’ )/scale;
```

```
                scale=scale*10;  cin>>x; }
                }//else
            push(OPND,num); num=0.0;//数压入栈，下个数初始化
        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
      cin>>x;//读入表达式中下一个字符。
    }//结束 while（x! ='$'）
    cout<<"后缀表达式的值为"<<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）假设以 I 和 O 分别表示入栈和出栈操作。栈的初态和终态均为空，入栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列，称可以操作的序列为合法序列，否则称为非法序列。

①下面所示的序列中哪些是合法的？

A. IOIIOIOO　　B. IOOIOIIO　　C. IIIOIOIO　　D. IIIOOIOO

②通过对①的分析，写出一个算法，判定所给的操作序列是否合法。若合法，返回 true，否则返回 false（假定被判定的操作序列已存入一维数组中）。

答案：

①A 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 '0' : k++; if(k>j){cout<<"序列非法"<<ednl; exit(0);}
       }
     i++; //不论 A[i]是 'I' 或 '0', 指针 i 均后移。}
   if(j!=k) {cout<<"序列非法"<<endl; return(false);}
   else { cout<<"序列合法"<<endl; return(true);}
  }//算法结束。
```

[算法讨论]在入栈出栈序列（即由'I'和'0'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'0'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）假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素站点(注意不设头指针) ，试编写相应的置空队、判队空 、入队和出队等算法。

[题目分析]

置空队就是建立一个头节点，并把头尾指针都指向头节点，头节点是不存放数据的；判队空就是当头指针等于尾指针时，队空；入队时，将新的节点插入到链队列的尾部，同时将尾指针指向这个节点；出队时，删除的是队头节点，要注意队列的长度大于 1 还是等于 1 的情况，这个时候要注意尾指针的修改，如果等于 1，则要删除尾指针指向的节点。

[算法描述]

```
//先定义链队结构:
typedef struct queuenode
{Datatype data;
 struct queuenode *next;
}QueueNode; //以上是结点类型的定义
typedef struct
{queuenode *rear;
}LinkQueue; //只设一个指向队尾元素的指针

(1) 置空队
void InitQueue( LinkQueue *Q)
{ //置空队：就是使头结点成为队尾元素
   QueueNode *s;
   Q->rear = Q->rear->next;//将队尾指针指向头结点
   while (Q->rear!=Q->rear->next)//当队列非空，将队中元素逐个出队
   {s=Q->rear->next;
    Q->rear->next=s->next;
```

```
    delete s;
  }//回收结点空间
}

(2) 判队空
int EmptyQueue( LinkQueue *Q)
{ //判队空。当头结点的 next 指针指向自己时为空队
  return Q->rear->next->next==Q->rear->next;
}

(3) 入队
void EnQueue( LinkQueue *Q, Datatype x)
{ //入队。也就是在尾结点处插入元素
 QueueNode *p=new QueueNode;//申请新结点
 p->data=x; p->next=Q->rear->next;//初始化新结点并链入
 Q-rear->next=p;
 Q->rear=p;//将尾指针移至新结点
}

(4) 出队
Datatype DeQueue( LinkQueue *Q)
{//出队,把头结点之后的元素摘下
 Datatype t;
 QueueNode *p;
 if(EmptyQueue( Q ))
  Error("Queue underflow");
 p=Q->rear->next->next; //p 指向将要摘下的结点
 x=p->data; //保存结点中数据
 if (p==Q->rear)
  {//当队列中只有一个结点时，p 结点出队后，要将队尾指针指向头结点
   Q->rear = Q->rear->next;
   Q->rear->next=p->next;
  }
 else
  Q->rear->next->next=p->next;//摘下结点 p
 delete p;//释放被删结点
 return x;
 }
```

（7）假设以数组 Q[*m*]存放循环队列中的元素，同时设置一个标志 *tag*，以 *tag* == 0 和 *tag* == 1 来区别在队头指针(*front*)和队尾指针(*rear*)相等时，队列状态为“空”还是“满”。试编写与此结构相应的插入(*enqueue*)和删除(*dlqueue*)算法。

[算法描述]

(1)初始化

```
SeQueue QueueInit(SeQueue Q)
{//初始化队列
 Q.front=Q.rear=0; Q.tag=0;
 return  Q;
}
```

(2)入队

```
SeQueue QueueIn(SeQueue Q,int e)
{//入队列
 if((Q.tag==1) && (Q.rear==Q.front)) cout<<"队列已满"<<endl;
 else
 {Q.rear=(Q.rear+1) % m;
  Q.data[Q.rear]=e;
  if(Q.tag==0) Q.tag=1; //队列已不空
  }
return  Q;
}
```

(3)出队

```
ElemType QueueOut(SeQueue Q)
{//出队列
if(Q.tag==0) { cout<<"队列为空"<<endl; exit(0);}
else
{Q.front=(Q.front+1) % m;
 e=Q.data[Q.front];
 if(Q.front==Q.rear) Q.tag=0;  //空队列
 }
return(e);
}
```

（8）如果允许在循环队列的两端都可以进行插入和删除操作。要求：

① 写出循环队列的类型定义；

② 写出“从队尾删除”和“从队头插入”的算法。

[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。

[算法描述]

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)  { cout<<"队列空"<<endl; 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)  { cout<<"队满"<<endl; 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。
```

（9）已知 Ackermann 函数定义如下：

$$Ack(m,n)=\begin{cases} n+1 & \text{当 } m=0 \text{ 时} \\ Ack(m-1,1) & \text{当 } m\neq 0, n=0 \text{ 时} \\ Ack(m-1,Ack(m,n-1)) & \text{当 } m\neq 0, n\neq 0 \text{ 时} \end{cases}$$

① 写出计算 Ack(m,n)的递归算法，并根据此算法给出出 Ack(2,1)的计算过程。

② 写出计算 Ack(m,n)的非递归算法。

[算法描述]

```
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
```

（10）已知 f 为单链表的表头指针，链表中存储的都是整型数据，试写出实现下列运算的递归算法：

① 求链表中的最大整数；

② 求链表的结点个数；

③ 求所有整数的平均值。

[算法描述]

①

```
int GetMax(LinkList p)
{
    if(!p->next)
```

```
        return p->data;
    else
    {
        int max=GetMax(p->next);
        return p->data>=max ? p->data:max;
    }
}
```

②

```
int GetLength(LinkList p)
{
    if(!p->next)
        return 1;
    else
    {
        return GetLength(p->next)+1;
    }
}
```

③

```
double GetAverage(LinkList p , int n)
{
    if(!p->next)
        return p->data;
    else
    {
        double ave=GetAverage(p->next,n-1);
        return (ave*(n-1)+p->data)/n;
    }
}
```

第 4 章　串、数组和广义表

1．选择题

（1）串是一种特殊的线性表，其特殊性体现在（　）。

A．可以顺序存储　　B．数据元素是一个字符

C．可以链式存储　　D．数据元素可以是多个字符若

答案：B

（2）串下面关于串的的叙述中，（　）是不正确的？

A．串是字符的有限序列　　B．空串是由空格构成的串

C．模式匹配是串的一种重要运算　D．串既可以采用顺序存储，也可以采用链式存储

答案：B

解释：空格常常是串的字符集合中的一个元素，有一个或多个空格组成的串成为空格串，零个字符的串成为空串，其长度为零。

（3）串“abababaaababaa”的 next 数组为（　）。

A．012345678999　B．012121111212　C．011234223456　D．0123012322345

答案：C

（4）串“ababaabab”的 nextval 为（　）。

A．010104101　　B．010102101　　C．010100011　　D．010101011

答案：A

（5）串的长度是指（　）。

A．串中所含不同字母的个数　　B．串中所含字符的个数

C．串中所含不同字符的个数　　D．串中所含非空格字符的个数

答案：B

解释：串中字符的数目称为串的长度。

（6）假设以行序为主序存储二维数组 A=array[1..100,1..100]，设每个数据元素占 2 个存储单元，基地址为 10，则 LOC[5,5]=（　）。

A．808　　B．818　　C．1010　　D．1020

答案：B

解释：以行序为主，则 LOC[5,5]=[（5-1）*100+（5-1）]*2+10=818。

（7）设有数组 A[i,j]，数组的每个元素长度为 3 字节，i 的值为 1 到 8，j 的值为 1 到 10，数组从内存首地址 BA 开始顺序存放，当用以列为主存放时，元素 A[5,8]的存储首地址为（　）。

A．BA+141　　B．BA+180　　C．BA+222　　D．BA+225

答案：B

解释：以列序为主，则 LOC[5,8]=[（8-1）*8+（5-1）]*3+BA=BA+180。

（8）设有一个 10 阶的对称矩阵 A，采用压缩存储方式，以行序为主存储，a_{11} 为第一元素，其存储地址为 1，每个元素占一个地址空间，则 a_{85} 的地址为（　）。

A．13　　B．32　　C．33　　D．40

答案：C

（9）若对 n 阶对称矩阵 A 以行序为主序方式将其下三角形的元素(包括主对角线上所有元素)依次存放于一维数组 B[1..(n(n+1))/2]中，则在 B 中确定 a_{ij}（i<j）的位置 k 的关系为（　）。

A．i*(i-1)/2+j　　B．j*(j-1)/2+i　　C．i*(i+1)/2+j　　D．j*(j+1)/2+i

答案：B

（10）二维数组 A 的每个元素是由 10 个字符组成的串，其行下标 i=0, 1, …, 8, 列下标 j=1, 2, …, 10。若 A 按行先存储，元素 A[8, 5]的起始地址与当 A 按列先存储时的元素（　　）的起始地址相同。设每个字符占一个字节。

A．A[8,5]　　B．A[3,10]　　C. A[5,8]　　D．A[0,9]

答案：B

解释：设数组从内存首地址 M 开始顺序存放，若数组按行先存储，元素 A[8,5]的起始地址为：M+[（8-0）*10+（5-1）]*1=M+84；若数组按列先存储，易计算出元素 A[3,10]的起始地址为：M+[（10-1）*9+（3-0）]*1=M+84。故选 B。

（11）设二维数组 A[1.. m，1.. n]（即 m 行 n 列）按行存储在数组 B[1.. m*n]中，则二维数组元素 A[i,j]在一维数组 B 中的下标为（　）。

A．(i-1)*n+j　　B．(i-1)*n+j-1　　C．i*(j-1)　　D．j*m+i-1

答案：A

解释：特殊值法。取 i=j=1，易知 A[1,1]的的下标为 1，四个选项中仅有 A 选项能确定的值为 1，故选 A。

（12）数组 A[0..4,-1..-3,5..7]中含有元素的个数（　）。

A．55　　B．45　　C．36　　D．16

答案：B

解释：共有 5*3*3=45 个元素。

（13）广义表 A=(a,b,(c,d),(e,(f,g)))，则 Head(Tail(Head(Tail(Tail(A)))))的值为（　）。

A．(g)　　B．(d)　　C．c　　D．d

答案：D

解释：Tail(A)=(b,(c,d),(e,(f,g)))；Tail(Tail(A))=((c,d),(e,(f,g)))；　Head(Tail(Tail(A)))=(c,d)；Tail(Head(Tail(Tail(A))))=(d)；Head(Tail(Head(Tail(Tail(A)))))=d。

（14）广义表((a,b,c,d))的表头是（　），表尾是（　）。

A．a　　B．( )　　C．(a,b,c,d)　　D．(b,c,d)

答案：C、B

解释：表头为非空广义表的第一个元素，可以是一个单原子，也可以是一个子表，((a,b,c,d))的表头为一个子表(a,b,c,d)；表尾为除去表头之外，由其余元素构成的表，表为一定是个广义表，((a,b,c,d))的表尾为空表( )。

（15）设广义表 L=((a,b,c))，则 L 的长度和深度分别为（　）。

A．1 和 1　　B．1 和 3　　C．1 和 2　　D．2 和 3

答案：C

解释：广义表的深度是指广义表中展开后所含括号的层数，广义表的长度是指广义表中所含元素的个数。根据定义易知 L 的长度为 1，深度为 2。

2．应用题

（1）已知模式串 t='abcaabbabcab'写出用 KMP 法求得的每个字符对应的 next 和 nextval 函数值。

答案：

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

（2）设目标为 t=“abcaabbabcabaacbacba”，模式为 p=“abcabaa”

① 计算模式 p 的 naxtval 函数值；

② 不写出算法，只画出利用 KMP 算法进行模式匹配时每一趟的匹配过程。

答案：

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)

（3）数组 A 中，每个元素 A[i, j]的长度均为 32 个二进位，行下标从-1 到 9，列下标从 1 到 11，从首地址 S 开始连续存放主存储器中，主存储器字长为 16 位。求：

① 存放该数组所需多少单元？

② 存放数组第 4 列所有元素至少需多少单元？

③ 数组按行存放时，元素 A[7, 4]的起始地址是多少？

④ 数组按列存放时，元素 A[4, 7]的起始地址是多少？

答案：

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)请将香蕉 banana 用工具 H( )—Head( ), T( )—Tail( )从 L 中取出。

L=(apple,(orange,(strawberry,(banana)),peach),pear)

答案：H（H（T（H（T（H（T（L））））））

3．算法设计题

（1）写一个算法统计在输入字符串中各个不同字符出现的频度并将结果存入文件（字符串中的合法字符为 A-Z 这 26 个字母和 0-9 这 10 个数字）。

[题目分析] 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。

[算法描述]

```
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
       cout<<“数字”<<i<< “的个数=”<<num[i]<<endl;
  for（i＝10; i<36; i++）// 求出字母字符的个数
       cout<<“字母字符”<<i+55<< “的个数=”<<num[i]<<endl;
 }
```

（2）写一个递归算法来实现字符串逆序存储，要求不另设串存储空间。

[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。

[算法描述]

```
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
 cin>>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
  }
```

（3）编写算法，实现下面函数的功能。函数 void insert(char*s,char*t,int pos)将字符串 t 插入到字符串 s 中，插入位置为 pos。假设分配给字符串 s 的空间足够让字符串 t 插入。（说明：不得使用任何库函数）

[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。

[算法描述]

```
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) {cout<<“pos 参数位置非法”<<endl;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') { cout<<pos<<"位置大于字符串 s 的长度";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。

（4）已知字符串 S1 中存放一段英文，写出算法 format(s1,s2,s3,n),将其按给定的长度 n 格式化成两端对齐的字符串 S2，其多余的字符送 S3。

[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。

[算法描述]

```
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') {cout<<"字符串 s1 为空串或空格串"<<endl;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'){cout<<"字符串 s1 没有"<<n<<"个有效字符"<<endl; 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')
       {cout<<"s1 串没有"<<n<<"个两端对齐的字符串"<<endl; 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
 }
```

（5）设二维数组 a[1..m, 1..n] 含有 m*n 个整数。

① 写一个算法判断 a 中所有元素是否互不相同?输出相关信息(yes/no)；

② 试分析算法的时间复杂度。

①

[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。

[算法描述]

```
int JudgEqual(ing 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]) {cout<<“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]) { cout<<“no”; return(0); }
      }// for(j=0;j<n-1;j++)
    cout<<“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*n 个元素，第 1 个元素同其它 m*n-1 个元素比较，第 2 个元素同其它 m*n-2 个元素比较，……，第 m*n-1 个元素同最后一个元素(m*n)比较一次,所以在元素互不相等时总的比较次数为 (m*n-1)+(m*n-2)+…+2+1=（m*n）(m*n-1)/2。在有相同元素时,可能第一次比较就相同,也可能最后一次比较时相同,设在(m*n-1)个位置上均可能相同,这时的平均比较次数约为（m*n）(m*n-1)/4，总的时间复杂度是 $O(n^4)$。

(6)设任意 n 个整数存放于数组 A(1:n)中，试编写算法，将所有正数排在所有负数前面（要求算法复杂度为 O(n)）。

[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。

[算法描述]

```
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]
  }// while(i<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).

第 5 章　树和二叉树

1．选择题

（1）把一棵树转换为二叉树后，这棵二叉树的形态是（　　）。

A．唯一的　　B．有多种

C．有多种，但根结点都没有左孩子　　D．有多种，但根结点都没有右孩子

答案：A

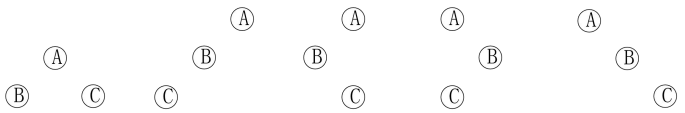
解释：因为二叉树有左孩子、右孩子之分，故一棵树转换为二叉树后，这棵二叉树的形态是唯一的。

（2）由 3 个结点可以构造出多少种不同的二叉树？（　　）

A．2　　B．3　　C．4　　D．5

答案：D

解释：五种情况如下：



（3）一棵完全二叉树上有 1001 个结点，其中叶子结点的个数是（　）。

A．250　　B．500　　C．254　　D．501

答案：D

解释：设度为 0 结点（叶子结点）个数为 A，度为 1 的结点个数为 B，度为 2 的结点个数为 C，有 A=C+1，A+B+C=1001，可得 2C+B=1000，由完全二叉树的性质可得 B=0 或 1，又因为 C 为整数，所以 B=0，C=500，A=501，即有 501 个叶子结点。

（4）一个具有 1025 个结点的二叉树的高 h 为（　）。

A．11　　B．10　　C．11 至 1025 之间　　D．10 至 1024 之间

答案：C

解释：若每层仅有一个结点，则树高 h 为 1025；且其最小树高为 $\lfloor \log_2 1025 \rfloor + 1 = 11$，即 h 在 11 至 1025 之间。

（5）深度为 h 的满 m 叉树的第 k 层有（　）个结点。(1=<k=<h)

A．m^{k-1}　　B．m^k-1　　C．m^{h-1}　　D．m^h-1

答案：A

解释：深度为 h 的满 m 叉树共有 m^h-1 个结点，第 k 层有 m^{k-1} 个结点。

（6）利用二叉链表存储树，则根结点的右指针是（　）。

A．指向最左孩子　　B．指向最右孩子　　C．空　　D．非空

答案：C

解释：利用二叉链表存储树时，右指针指向兄弟结点，因为根节点没有兄弟结点，故根节点的右指针指向空。

（7）对二叉树的结点从 1 开始进行连续编号，要求每个结点的编号大于其左、右孩子的编号，同一结点的左右孩子中，其左孩子的编号小于其右孩子的编号，可采用（　）遍历实现编号。

A．先序　　B．中序　　C．后序　　D．从根开始按层次遍历

答案：C

解释：根据题意可知按照先左孩子、再右孩子、最后双亲结点的顺序遍历二叉树，即后序遍历二叉树。

（8）若二叉树采用二叉链表存储结构，要交换其所有分支结点左、右子树的位置，利用（　）遍历方法最合适。

A．前序　　B．中序　　C．后序　　D．按层次

答案：C

解释：后续遍历和层次遍历均可实现左右子树的交换，不过层次遍历的实现消耗比后续大，后序遍历方法最合适。

（9）在下列存储形式中，（　）不是树的存储形式？

A．双亲表示法　B．孩子链表表示法　C．孩子兄弟表示法　D．顺序存储表示法

答案：D

解释：树的存储结构有三种：双亲表示法、孩子表示法、孩子兄弟表示法，其中孩子兄弟表示法是常用的表示法，任意一棵树都能通过孩子兄弟表示法转换为二叉树进行存储。

（10）一棵非空的二叉树的先序遍历序列与后序遍历序列正好相反，则该二叉树一定满足（　）。

A．所有的结点均无左孩子　　B．所有的结点均无右孩子

C．只有一个叶子结点　　D．是任意一棵二叉树

答案：C

解释：因为先序遍历结果是“中左右”，后序遍历结果是“左右中”，当没有左子树时，就是“中右”和“右中”；当没有右子树时，就是“中左”和“左中”。则所有的结点均无左孩子或所有的结点均无右孩子均可，所以 A、B 不能选，又所有的结点均无左孩子与所有的结点均无右孩子时，均只有一个叶子结点，故选 C。

（11）设哈夫曼树中有 199 个结点，则该哈夫曼树中有（　　）个叶子结点。

A．99　　B．100

C．101　　D．102

答案：B

解释：在哈夫曼树中没有度为 1 的结点，只有度为 0（叶子结点）和度为 2 的结点。设叶子结点的个数为 n0，度为 2 的结点的个数为 n2，由二叉树的性质 n0=n2+1，则总结点数 n=n0+n2=2*n0-1，得到 n0=100。

（12）若 X 是二叉中序线索树中一个有左孩子的结点，且 X 不为根，则 X 的前驱为（　）。

A．X 的双亲　　B．X 的右子树中最左的结点

C．X 的左子树中最右结点　　D．X 的左子树中最右叶结点

答案：C

（13）引入二叉线索树的目的是（　）。

A．加快查找结点的前驱或后继的速度　　B．为了能在二叉树中方便的进行插入与删除

C．为了能方便的找到双亲　　D．使二叉树的遍历结果唯一

答案：A

（14）设 F 是一个森林，B 是由 F 变换得的二叉树。若 F 中有 n 个非终端结点，则 B 中右指针域为空的结点有（　）个。

A．n-1　　B．n　　C．n + 1　　D．n + 2

答案：C

（15）n(n≥2)个权值均不相同的字符构成哈夫曼树，关于该树的叙述中，错误的是（　）。

A．该树一定是一棵完全二叉树

B．树中一定没有度为 1 的结点

C．树中两个权值最小的结点一定是兄弟结点

D．树中任一非叶结点的权值一定不小于下一层任一结点的权值

答案：A

解释：哈夫曼树的构造过程是每次都选取权值最小的树作为左右子树构造一棵新的二叉树，所以树中一定没有度为 1 的结点、两个权值最小的结点一定是兄弟结点、任一非叶结点的权值一定不小于下一层任一结点的权值。

2．应用题

（1）试找出满足下列条件的二叉树

① 先序序列与后序序列相同　　②中序序列与后序序列相同

③ 先序序列与中序序列相同　　④中序序列与层次遍历序列相同

答案：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＂，根据以上原则有

① 或为空树，或为只有根结点的二叉树

② 或为空树，或为任一结点至多只有左子树的二叉树.

③ 或为空树，或为任一结点至多只有右子树的二叉树.

④ 或为空树，或为任一结点至多只有右子树的二叉树

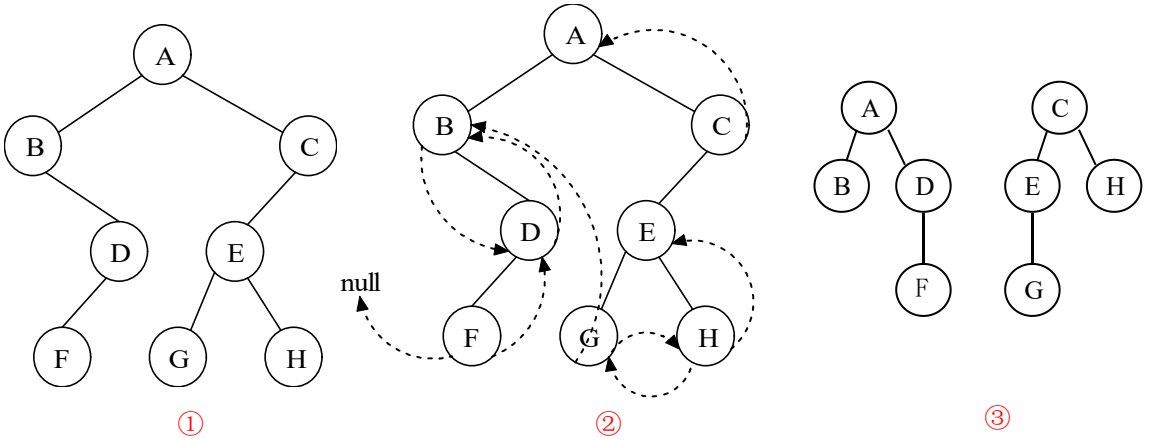
（2）设一棵二叉树的先序序列： A B D F C E G H ，中序序列： B F D A G E H C

①画出这棵二叉树。

②画出这棵二叉树的后序线索树。

③将这棵二叉树转换成对应的树（或森林）。

答案：



（3） 假设用于通信的电文仅由 8 个字母组成，字母在电文中出现的频率分别为 0.07，0.19，0.02，0.06，0.32，0.03，0.21，0.10。

① 试为这 8 个字母设计赫夫曼编码。

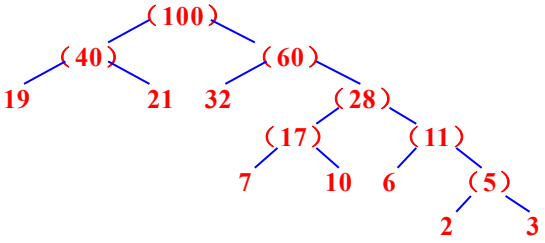
② 试设计另一种由二进制表示的等长编码方案。

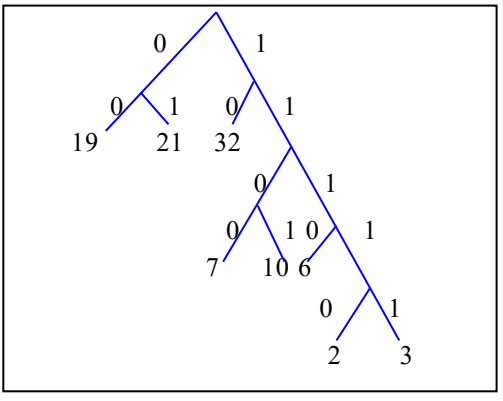
③ 对于上述实例，比较两种方案的优缺点。

答案：方案 1；哈夫曼编码

先将概率放大 100 倍，以方便构造哈夫曼树。

w={7,19,2,6,32,3,21,10}，按哈夫曼规则：【[（2,3），6], (7,10)】，……19, 21, 32





方案比较：

字母编号	对应编码	出现频率
1	1100	0.07
2	00	0.19
3	11110	0.02
4	1110	0.06
5	10	0.32
6	11111	0.03
7	01	0.21
8	1101	0.10

字母编号	对应编码	出现频率
1	000	0.07
2	001	0.19
3	010	0.02
4	011	0.06
5	100	0.32
6	101	0.03
7	110	0.21
8	111	0.10

方案 1 的 WPL＝2(0.19+0.32+0.21)+4(0.07+0.06+0.10)+5(0.02+0.03)=1.44+0.92+0.25=2.61

方案 2 的 WPL＝3(0.19+0.32+0.21+0.07+0.06+0.10+0.02+0.03)=3

结论：哈夫曼编码优于等长二进制编码

（4）已知下列字符 A、B、C、D、E、F、G 的权值分别为 3、12、7、4、2、8，11，试填写出其对应哈夫曼树 HT 的存储结构的初态和终态。

答案：

初态：

	weight	parent	lchild	rchild
1	3	0	0	0
2	12	0	0	0
3	7	0	0	0
4	4	0	0	0
5	2	0	0	0
6	8	0	0	0
7	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

终态：

	weight	parent	lchild	rchild
1	3	8	0	0
2	12	12	0	0
3	7	10	0	0
4	4	9	0	0
5	2	8	0	0
6	8	10	0	0
7	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

3．算法设计题

以二叉链表作为二叉树的存储结构，编写以下算法：

（1）统计二叉树的叶结点个数。

[题目分析]如果二叉树为空，返回 0，如果二叉树不为空且左右子树为空，返回 1，如果二叉树不为空，且左右子树不同时为空，返回左子树中叶子节点个数加上右子树中叶子节点个数。

[算法描述]

```
int LeafNodeCount(BiTree T)
{
    if(T==NULL)
        return 0; //如果是空树，则叶子结点个数为 0
    else if(T->lchild==NULL&&T->rchild==NULL)
        return 1; //判断结点是否是叶子结点（左孩子右孩子都为空），若是则返回 1
    else
        return LeafNodeCount(T->lchild)+LeafNodeCount(T->rchild);
}
```

（2）判别两棵树是否相等。

[题目分析]先判断当前节点是否相等(需要处理为空、是否都为空、是否相等)，如果当前节点不相等，直接返回两棵树不相等;如果当前节点相等，那么就递归的判断他们的左右孩子是否相等。

[算法描述]

```
int compareTree(TreeNode* tree1, TreeNode* tree2)
//用分治的方法做，比较当前根，然后比较左子树和右子树
{bool tree1IsNull = (tree1==NULL);
 bool tree2IsNull = (tree2==NULL);
 if(tree1IsNull != tree2IsNull)
  {
   return 1;
  }
 if(tree1IsNull && tree2IsNull)
 {//如果两个都是 NULL，则相等
  return 0;
 }//如果根节点不相等，直接返回不相等，否则的话，看看他们孩子相等不相等
 if(tree1->c != tree2->c)
 {
  return 1;
 }
 return (compareTree(tree1->left,tree2->left)&compareTree(tree1->right,tree2->right))
        (compareTree(tree1->left,tree2->right)&compareTree(tree1->right,tree2->left));
}//算法结束
```

（3）交换二叉树每个结点的左孩子和右孩子。

[题目分析]如果某结点左右子树为空，返回，否则交换该结点左右孩子，然后递归交换左右子树。

[算法描述]

```
void ChangeLR(BiTree &T)
{
    BiTree temp;
    if(T->lchild==NULL&&T->rchild==NULL)
        return;
    else
    {
        temp = T->lchild;
        T->lchild = T->rchild;
        T->rchild = temp;
```

```
    }//交换左右孩子
    ChangeLR(T->lchild);  //递归交换左子树
    ChangeLR(T->rchild);  //递归交换右子树
}
```

（4）设计二叉树的双序遍历算法（双序遍历是指对于二叉树的每一个结点来说，先访问这个结点，再按双序遍历它的左子树，然后再一次访问这个结点，接下来按双序遍历它的右子树）。

[题目分析]若树为空，返回；若某结点为叶子结点，则仅输出该结点；否则先输出该结点，递归遍历其左子树，再输出该结点，递归遍历其右子树。

[算法描述]

```
void DoubleTraverse(BiTree T)
{
    if(T == NULL)
        return;
    else if(T->lchild==NULL&&T->rchild==NULL)
        cout<<T->data;      //叶子结点输出
    else
    {
        cout<<T->data;
        DoubleTraverse(T->lchild);   //递归遍历左子树
        cout<<T->data;
        DoubleTraverse(T->rchild);   //递归遍历右子树
    }
}
```

（5）计算二叉树最大的宽度（二叉树的最大宽度是指二叉树所有层中结点个数的最大值）。

[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。

[算法描述]

```
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
```

（6）用按层次顺序遍历二叉树的方法，统计树中具有度为 1 的结点数目。

[题目分析]

若某个结点左子树空右子树非空或者右子树空左子树非空，则该结点为度为 1 的结点

[算法描述]

```
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); cout<<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
         } // while(!QueueEmpty(Q))
       }//if(bt)
  return(num);
 }//返回度为 1 的结点的个数
```

（7）求任意二叉树中第一条最长的路径长度，并输出此路径上各结点的值。

[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。

[算法描述]

```
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
```

（8）输出二叉树中从每个叶子结点到根结点的路径。

[题目分析]采用先序遍历的递归方法，当找到叶子结点*b 时，由于*b 叶子结点尚未添加到 path 中，因此在输出路径时还需输出 b->data 值。

[算法描述]

```
void AllPath(BTNode *b,ElemType path[],int pathlen)
{int i;
 if (b!=NULL)
    {if (b->lchild==NULL && b->rchild==NULL) //*b 为叶子结点
       {cout << " " << b->data << "到根结点路径:" << b->data;
        for (i=pathlen-1;i>=0;i--)
        cout << endl;
       }
      else
       {path[pathlen]=b->data;              //将当前结点放入路径中
        pathlen++;                          //路径长度增 1
        AllPath(b->lchild,path,pathlen);    //递归扫描左子树
        AllPath(b->rchild,path,pathlen);    //递归扫描右子树
        pathlen--;                          //恢复环境
       }
  }// if (b!=NULL)
}//算法结束
```

第 6 章 图

1．选择题

（1）在一个图中，所有顶点的度数之和等于图的边数的（　　）倍。

A．1/2　　B．1　　C．2　　D．4

答案：C

（2）在一个有向图中，所有顶点的入度之和等于所有顶点的出度之和的（　　）倍。

A．1/2　　B．1　　C．2　　D．4

答案：B

解释：有向图所有顶点入度之和等于所有顶点出度之和。

（3）具有 n 个顶点的有向图最多有（　　）条边。

A．n　　B．n(n-1)　　C．n(n+1)　　D．n^2

答案：B

解释：有向图的边有方向之分，即为从 n 个顶点中选取 2 个顶点有序排列，结果为 n(n-1)。

（4）n 个顶点的连通图用邻接距阵表示时，该距阵至少有（　　）个非零元素。

A．n　　B．2(n-1)　　C．n/2　　D．n^2

答案：B

（5）G 是一个非连通无向图，共有 28 条边，则该图至少有（　　）个顶点。

A．7　　B．8　　C．9　　D．10

答案：C

解释：8 个顶点的无向图最多有 8*7/2=28 条边，再添加一个点即构成非连通无向图，故至少有 9 个顶点。

（6）若从无向图的任意一个顶点出发进行一次深度优先搜索可以访问图中所有的顶点，则该图一定是（　　）图。

A．非连通　　B．连通　　C．强连通　　D．有向

答案：B

解释：即从该无向图任意一个顶点出发有到各个顶点的路径，所以该无向图是连通图。

（7）下面（　）算法适合构造一个稠密图 G 的最小生成树。

A．　Prim 算法　　B．Kruskal 算法　　C．Floyd 算法　　D．Dijkstra 算法

答案：A

解释：Prim 算法适合构造一个稠密图 G 的最小生成树，Kruskal 算法适合构造一个稀疏图 G 的最小生成树。

（8）用邻接表表示图进行广度优先遍历时，通常借助（　　）来实现算法。

A．栈　　B. 队列　　C.　树　　D．图

答案：B

解释：广度优先遍历通常借助队列来实现算法，深度优先遍历通常借助栈来实现算法。

（9）用邻接表表示图进行深度优先遍历时，通常借助（　　）来实现算法。

A．栈　　B．队列　　C．树　　D．图

答案：A

解释：广度优先遍历通常借助队列来实现算法，深度优先遍历通常借助栈来实现算法。

（10）深度优先遍历类似于二叉树的（　　）。

A．先序遍历　　B．中序遍历　　C．后序遍历　　D．层次遍历

答案：A

（11）广度优先遍历类似于二叉树的（　　）。

A．先序遍历　　B．中序遍历　　C．后序遍历　　D．层次遍历

答案：D

（12）图的 BFS 生成树的树高比 DFS 生成树的树高（　　）。

A．小　　B．相等　　C．小或相等　　D．大或相等

答案：C

解释：对于一些特殊的图，比如只有一个顶点的图，其 BFS 生成树的树高和 DFS 生成树的树高相等。一般的图，根据图的 BFS 生成树和 DFS 树的算法思想，BFS 生成树的树高比 DFS 生成树的树高小。

（13）已知图的邻接矩阵如图 6.30 所示，则从顶点 v_0 出发按深度优先遍历的结果是（　　）。

v_0	0	1	1	1	1	0	1
v_1	1	0	0	1	0	0	1
v_2	1	0	0	0	1	0	0
v_3	1	1	0	0	1	1	0
v_4	1	0	1	1	0	1	0
v_5	0	0	0	1	1	0	1
v_6	1	1	0	0	0	1	0

图 6.30　邻接矩阵

A．0 2 4 3 1 5 6

B．0 1 3 6 5 4 2

C．0 1 3 4 2 5 6

D．0 3 6 1 5 4 2

（14）已知图的邻接表如图 6.31 所示，则从顶点 v_0 出发按广度优先遍历的结果是（　　），按深度优先遍历的结果是（　　）。

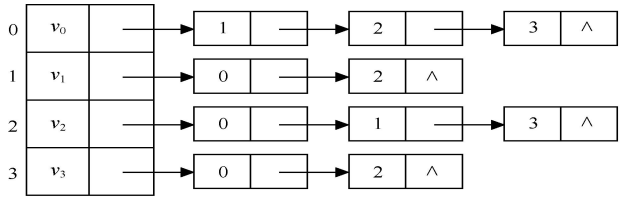


图 6.31　邻接表

A．0 1 3 2　　B．0 2 3 1　　C．0 3 2 1　　D．0 1 2 3

答案：D、D

（15）下面（　　）方法可以判断出一个有向图是否有环。

A．深度优先遍历　　B．拓扑排序　　C．求最短路径　　D．求关键路径

答案：B

2．应用题

（1）已知图 6.32 所示的有向图，请给出：

① 每个顶点的入度和出度；

② 邻接矩阵；

③ 邻接表；

④ 逆邻接表。

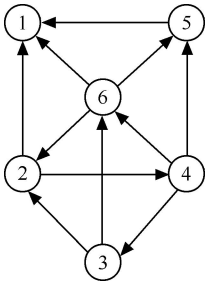


图 6.32　有向图

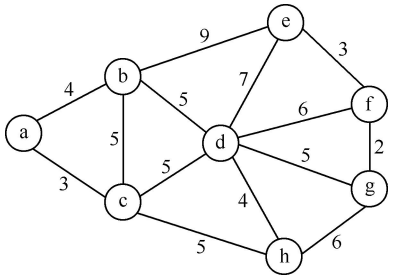


图 6.33　无向网

答案：

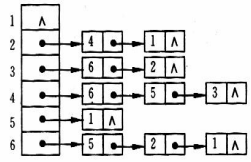
(1)

顶点	1	2	3	4	5	6
入度	3	2	1	1	2	2
出度	0	2	2	3	1	3

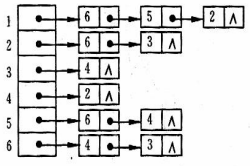
(2) 邻接矩阵

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(3) 邻接表



(4) 逆邻接表



（2）已知如图 6.33 所示的无向网，请给出：

① 邻接矩阵；

② 邻接表；

③ 最小生成树

答案：

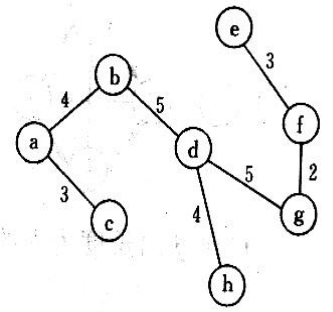
①

$$\begin{bmatrix} \infty & 4 & 3 & \infty & \infty & \infty & \infty & \infty \\ 4 & \infty & 5 & 5 & 9 & \infty & \infty & \infty \\ 3 & 5 & \infty & 5 & \infty & \infty & \infty & 5 \\ \infty & 5 & 5 & \infty & 7 & 6 & 5 & 4 \\ \infty & 9 & \infty & 7 & \infty & 3 & \infty & \infty \\ \infty & \infty & \infty & 6 & 3 & \infty & 2 & \infty \\ \infty & \infty & \infty & 5 & \infty & 2 & \infty & 6 \\ \infty & \infty & 5 & 4 & \infty & \infty & 6 & \infty \end{bmatrix}$$

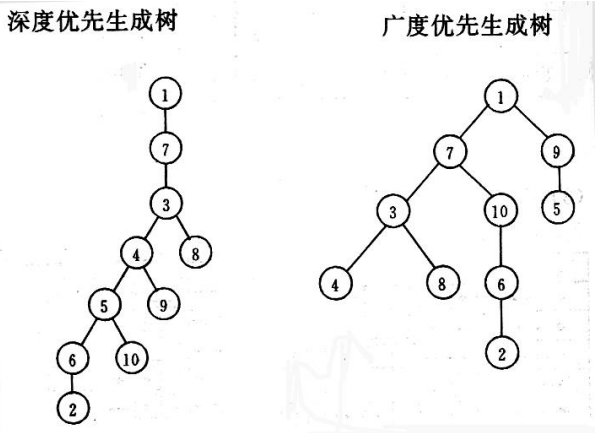
②

a	→	b	4	→	c	3									
b	→	a	4	→	c	5	→	d	5	→	e	9			
c	→	a	3	→	b	5	→	d	5	→	h	5			
d	→	b	5	→	c	5	→	e	7	→	f	6	→	g	5	→	h	4
e	→	b	9	→	d	7	→	f	3						
f	→	d	6	→	e	3	→	g	2						
g	→	d	5	→	f	2	→	h	6						

③



（3）已知图的邻接矩阵如图 6.34 所示。试分别画出自顶点 1 出发进行遍历所得的深度优先生成树和广度优先生成树。



（4）有向网如图 6.35 所示，试用迪杰斯特拉算法求出从顶点 a 到其他各顶点间的最短路径，完成表 6.9。

图 6.28　邻接矩阵

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	1	0	1	0
2	0	0	1	0	0	0	1	0	0	0
3	0	0	0	1	0	0	0	1	0	0
4	0	0	0	0	1	0	0	0	1	0
5	0	0	0	0	0	1	0	0	0	1
6	1	1	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0	1
8	1	0	0	1	0	0	0	0	1	0
9	0	0	0	0	1	0	1	0	0	1
10	1	0	0	0	0	1	0	0	0	0

图 6.34　邻接矩阵

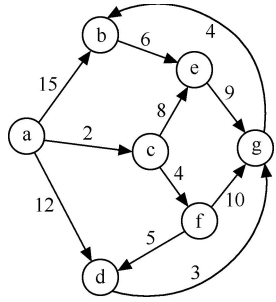


图 6.35　有向网

表 6.9

D 终点	i=1	i=2	i=3	i=4	i=5	i=6
b	15 (a,b)	15 (a,b)	15 (a,b)	15 (a,b)	15 (a,b)	**15** **(a,b)**
c	**2** **(a,c)**					
d	12 (a,d)	12 (a,d)	11 (a,c,f,d)	**11** **(a,c,f,d)**		
e	∞	10 (a,c,e)	**10** **(a,c,e)**			
f	∞	**6** **(a,c,f)**				
g	∞	∞	16 (a,c,f,g)	16 (a,c,f,g)	**14** **(a,c,f,d,g)**	
S 终点集	{a,c}	{a,c,f}	{a,c,f,e}	{a,c,f,e,d}	{a,c,f,e,d,g}	{a,c,f,e,d,g,b}

（5）试对图 6.36 所示的 AOE-网：

① 求这个工程最早可能在什么时间结束；

② 求每个活动的最早开始时间和最迟开始时间；

③ 确定哪些活动是关键活动

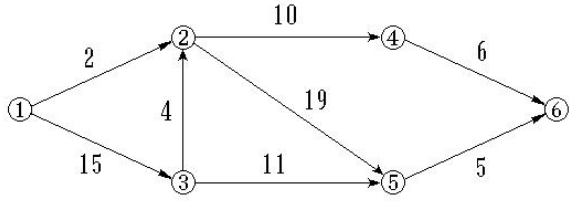


图 6.36　AOE-网

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

	1 ∂	2 ÷	3 •	4 ≠	5 ≡	6 ≈
Ve	0	19	15	29	38	43
Vl	0	19	15	37	38	43

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

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

3．算法设计题

（1）分别以邻接矩阵和邻接表作为存储结构，实现以下图的基本操作：

① 增加一个新顶点 v，InsertVex(G, v)；

② 删除顶点 v 及其相关的边，DeleteVex(G, v);

③ 增加一条边<v，w>，InsertArc(G, v, w);

④ 删除一条边<v，w>，DeleteArc(G, v, w)。

［算法描述］

假设图 G 为有向无权图，以邻接矩阵作为存储结构四个算法分别如下：

① 增加一个新顶点 v

```
Status Insert_Vex(MGraph &G, char v)//在邻接矩阵表示的图 G 上插入顶点 v
{
  if(G.vexnum+1)>MAX_VERTEX_NUM return INFEASIBLE;
  G.vexs[++G.vexnum]=v;
  return OK;
}//Insert_Vex
```

② 删除顶点 v 及其相关的边，

```
Status Delete_Vex(MGraph &G,char v)//在邻接矩阵表示的图 G 上删除顶点 v
{
n=G.vexnum;
if((m=LocateVex(G,v))<0) return ERROR;
G.vexs[m]<->G.vexs[n]; //将待删除顶点交换到最后一个顶点
for(i=0;i<n;i++)
```

```
{
G.arcs[m]=G.arcs[n];
G.arcs[m]=G.arcs[n]; //将边的关系随之交换
}
G.arcs[m][m].adj=0;
G.vexnum--;
return OK;
}//Delete_Vex
```

分析:如果不把待删除顶点交换到最后一个顶点的话,算法将会比较复杂,而伴随着大量元素的移动,时间复杂度也会大大增加。

③ 增加一条边<v，w>

```
Status Insert_Arc(MGraph &G,char v,char w)//在邻接矩阵表示的图 G 上插入边(v,w)
{
  if((i=LocateVex(G,v))<0) return ERROR;
  if((j=LocateVex(G,w))<0) return ERROR;
  if(i==j) return ERROR;
  if(!G.arcs[j].adj)
{
  G.arcs[j].adj=1;
  G.arcnum++;
}
     return OK;
}//Insert_Arc
```

④ 删除一条边<v，w>

```
Status Delete_Arc(MGraph &G,char v,char w)//在邻接矩阵表示的图 G 上删除边(v,w)
{
if((i=LocateVex(G,v))<0) return ERROR;
if((j=LocateVex(G,w))<0) return ERROR;
if(G.arcs[j].adj)
{
G.arcs[j].adj=0;
G.arcnum--;
}
return OK;
}//Delete_Arc
```

以邻接表作为存储结构，本题只给出 Insert_Arc 算法.其余算法类似。

```
Status Insert_Arc(ALGraph &G,char v,char w)//在邻接表表示的图 G 上插入边(v,w)
{
if((i=LocateVex(G,v))<0) return ERROR;
if((j=LocateVex(G,w))<0) return ERROR;
p=new ArcNode;
p->adjvex=j;p->nextarc=NULL;
if(!G.vertices.firstarc) G.vertices.firstarc=p;
else
{
for(q=G.vertices.firstarc;q->q->nextarc;q=q->nextarc)
if(q->adjvex==j) return ERROR; //边已经存在
q->nextarc=p;
}
G.arcnum++;
return OK;
}//Insert_Arc
```

（2）一个连通图采用邻接表作为存储结构，设计一个算法，实现从顶点 v 出发的深度优先遍历的非递归过程。

[算法描述]

```
Void DFSn(Graph G,int v)
{  //从第 v 个顶点出发非递归实现深度优先遍历图 G
    Stack s;
    SetEmpty(s);
    Push(s,v);
    While(!StackEmpty(s))
    {   //栈空时第 v 个顶点所在的连通分量已遍历完
        Pop(s,k);
        If(!visited[k])
        {   visited[k]=TRUE;
            VisitFunc(k);              //访问第 k 个顶点
            //将第 k 个顶点的所有邻接点进栈
        for(w=FirstAdjVex(G,k);w;w=NextAdjVex(G,k,w))
        {
            if(!visited[w]&&w!=GetTop(s)) Push(s,w);      //图中有环时 w==GetTop(s)
        }
    }
```

```
}
```

（3）设计一个算法，求图 G 中距离顶点 v 的最短路径长度最大的一个顶点，设 v 可达其余各个顶点。

[题目分析]

利用 Dijkstra 算法求 v0 到其它所有顶点的最短路径，分别保存在数组 D[i]中，然后求出 D[i]中值最大的数组下标 m 即可。

[算法描述]

```
int ShortestPath_MAX(AMGraph G, int v0){
    //用 Dijkstra 算法求距离顶点 v0 的最短路径长度最大的一个顶点 m
    n=G.vexnum;                          //n 为 G 中顶点的个数
    for(v = 0; v<n; ++v){                //n 个顶点依次初始化
      S[v] = false;                      //S 初始为空集
      D[v] = G.arcs[v0][v];              //将 v0 到各个终点的最短路径长度初始化
      if(D[v]< MaxInt)   Path [v]=v0;    //如果 v0 和 v 之间有弧，则将 v 的前驱置为 v0
      else Path [v]=-1;                  //如果 v0 和 v 之间无弧，则将 v 的前驱置为-1
    }//for
    S[v0]=true;                          //将 v0 加入 S
    D[v0]=0;                             //源点到源点的距离为 0
    /*开始主循环，每次求得 v0 到某个顶点 v 的最短路径，将 v 加到 S 集*/
    for(i=1;i<n; ++i){                   //对其余 n−1 个顶点，依次进行计算
      min= MaxInt;
      for(w=0;w<n; ++w)
        if(!S[w]&&D[w]<min)
            {v=w; min=D[w];}             //选择一条当前的最短路径，终点为 v
      S[v]=true;                         //将 v 加入 S
      for(w=0;w<n; ++w)                  //更新从 v0 到 V−S 上所有顶点的最短路径长度
      if(!S[w]&&(D[v]+G.arcs[v][w]<D[w])){
          D[w]=D[v]+G.arcs[v][w];        //更新 D[w]
          Path [w]=v;                    //更改 w 的前驱为 v
      }//if
```

```
    }//for
/*最短路径求解完毕，设距离顶点 v0 的最短路径长度最大的一个顶点为 m */
Max=D[0];
m=0;
for(i=1;i<n;i++)
    if(Max<D[i]) m=i;
return m;
}
```

（4）试基于图的深度优先搜索策略写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 v_i 到顶点 v_j 的路径（$i \neq j$）。

```
[题目分析]
引入一变量 level 来控制递归进行的层数
[算法描述]
int visited[MAXSIZE]; //指示顶点是否在当前路径上
int level＝1;//递归进行的层数
int exist_path_DFS(ALGraph G,int i,int j)//深度优先判断有向图 G 中顶点 i 到顶点 j
是否有路径,是则返回 1,否则返回 0
{
  if(i==j) return 1; //i 就是 j
  else
  {
    visited[i]=1;
    for(p=G.vertices[i].firstarc;p;p=p->nextarc， level--)
    { level++;
      k=p->adjvex;
      if(!visited[k]&&exist_path(k,j)) return 1;//i 下游的顶点到 j 有路径
}//for
  }//else
if (level==1)   return 0;
}//exist_path_DFS
```

（5）采用邻接表存储结构，编写一个算法，判别无向图中任意给定的两个顶点之间是否存在一条长度为为 k 的简单路径。

```
[算法描述]
int visited[MAXSIZE];
int exist_path_len(ALGraph G,int i,int j,int k)
//判断邻接表方式存储的有向图 G 的顶点 i 到 j 是否存在长度为 k 的简单路径
{if(i==j&&k==0) return 1; //找到了一条路径,且长度符合要求
 else if(k>0)
```

```
    {visited[i]=1;
     for(p=G.vertices[i].firstarc;p;p=p->nextarc)
       {l=p->adjvex;
        if(!visited[l])
            if(exist_path_len(G,l,j,k-1)) return 1; //剩余路径长度减一
       }//for
     visited[i]=0; //本题允许曾经被访问过的结点出现在另一条路径中
    }//else
  return 0; //没找到
}//exist_path_len
```

第 7 章　查找

1．选择题

（1）对 n 个元素的表做顺序查找时，若查找每个元素的概率相同，则平均查找长度为（　　）。

A．(n-1)/2　　B．n/2　　C．(n+1)/2　　D．n

答案：C

解释：总查找次数 N=1+2+3+…+n=n(n+1)/2，则平均查找长度为 N/n=(n+1)/2。

（2）适用于折半查找的表的存储方式及元素排列要求为（　　）。

A．链接方式存储，元素无序　　B．链接方式存储，元素有序

C．顺序方式存储，元素无序　　D．顺序方式存储，元素有序

答案：D

解释：折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列。

（3）如果要求一个线性表既能较快的查找，又能适应动态变化的要求，最好采用(　　)查找法。

A．顺序查找　　B．折半查找

C．分块查找　　D．哈希查找

答案：C

解释：分块查找的优点是：在表中插入和删除数据元素时，只要找到该元素对应的块，就可以在该块内进行插入和删除运算。由于块内是无序的，故插入和删除比较容易，无需进行大量移动。如果线性表既要快速查找又经常动态变化，则可采用分块查找。

（4）折半查找有序表（4，6，10，12，20，30，50，70，88，100）。若查找表中元素 58，则它将依次与表中（　　）比较大小，查找结果是失败。

A．20，70，30，50　　B．30，88，70，50

C．20，50　　D．30，88，50

答案：A

解释：表中共 10 个元素，第一次取⌊(1+10)/2⌋=5，与第五个元素 20 比较，58 大于 20，再取⌊(6+10)/2⌋=8，与第八个元素 70 比较，依次类推再与 30、50 比较，最终查找失败。

（5）对 22 个记录的有序表作折半查找，当查找失败时，至少需要比较（　　）次关键字。

A．3　　B．4　　C．5　　D．6

答案：B

解释：22 个记录的有序表，其折半查找的判定树深度为⌊$\log_2 22$⌋+ 1=5，且该判定树不是满二叉树，即查找失败时至多比较 5 次，至少比较 4 次。

（6）折半搜索与二叉排序树的时间性能（　　）。

A．相同　　B．完全不同

C．有时不相同　　D．数量级都是 O($\log_2 n$)

答案：C

（7）分别以下列序列构造二叉排序树，与用其它三个序列所构造的结果不同的是（　　）。

A.（100，80，　90，　60，　120，110，130）

B.（100，120，110，130，80，　60，　90）

C.（100，60，　80，　90，　120，110，130）

D.(100，80，　60，　90，　120，130，110)

答案：C

解释：A、B、C、D 四个选项构造二叉排序树都以 100 为根，易知 A、B、D 三个序列中第一个比 100 小的关键字为 80，即 100 的左孩子为 80，而 C 选项中 100 的左孩子为 60，故选 C。

（8）在平衡二叉树中插入一个结点后造成了不平衡，设最低的不平衡结点为 A，并已知 A 的左孩子的平衡因子为 0 右孩子的平衡因子为 1，则应作（　　）型调整以使其平衡。

A．LL　　B．LR　　C．RL　　D．RR

答案：C

（9）下列关于 m 阶 B-树的说法错误的是（　　）。

A．根结点至多有 m 棵子树

B．所有叶子都在同一层次上

C．非叶结点至少有 m/2 (m 为偶数)或 m/2+1（m 为奇数）棵子树

D．根结点中的数据是有序的

答案：D

（10）下面关于 B-和 B+树的叙述中，不正确的是（　　）。

A．B-树和 B+树都是平衡的多叉树　　B．B-树和 B+树都可用于文件的索引结构

C．B-树和 B+树都能有效地支持顺序检索　D．B-树和 B+树都能有效地支持随机检索

答案：C

（11）m 阶 B-树是一棵（　　）。

A．m 叉排序树　　B．m 叉平衡排序树

C．m-1 叉平衡排序树　　D．m+1 叉平衡排序树

答案：B

（12）下面关于哈希查找的说法，正确的是（　　）。

A．哈希函数构造的越复杂越好，因为这样随机性好，冲突小

B．除留余数法是所有哈希函数中最好的

C．不存在特别好与坏的哈希函数，要视情况而定

D．哈希表的平均查找长度有时也和记录总数有关

答案：C

（13）下面关于哈希查找的说法，不正确的是（　　）。

A．采用链地址法处理冲突时，查找一个元素的时间是相同的

B．采用链地址法处理冲突时，若插入规定总是在链首，则插入任一个元素的时间是相同的

C．用链地址法处理冲突，不会引起二次聚集现象

D．用链地址法处理冲突，适合表长不确定的情况

答案：A

解释：在同义词构成的单链表中，查找该单链表表中不同元素，所消耗的时间不同。

（14）设哈希表长为 14，哈希函数是 H(key)=key%11，表中已有数据的关键字为 15，38，61，84 共四个，现要将关键字为 49 的元素加到表中，用二次探测法解决冲突，则放入的位置是（　　）。

A．8　　B．3　　C．5　　D．9

答案：D

解释：关键字 15 放入位置 4，关键字 38 放入位置 5，关键字 61 放入位置 6，关键字 84 放入位置 7，再添加关键字 49，计算得到地址为 5，冲突，用二次探测法解决冲突得到新地址为 6，仍冲突，再用用二次探测法解决冲突，得到新地址为 4，仍冲突，再用用二次探测法解决冲突，得到新地址为 9，不冲突，即将关键字 49 放入位置 9。

（15）采用线性探测法处理冲突，可能要探测多个位置，在查找成功的情况下，所探测的这些位置上的关键字（　　）。

A．不一定都是同义词　　B．一定都是同义词

C．一定都不是同义词　　D．都相同

答案：A

解释：所探测的这些关键字可能是在处理其它关键字冲突过程中放入该位置的。

2．应用题

（1）假定对有序表：（3，4，5，7，24，30，42，54，63，72，87，95）进行折半查找，试回答下列问题：

① 画出描述折半查找过程的判定树；

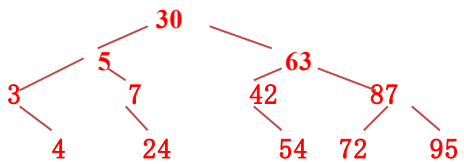
② 若查找元素 54，需依次与哪些元素比较？

③ 若查找元素 90，需依次与哪些元素比较？

④ 假定每个元素的查找概率相等，求查找成功时的平均查找长度。

答案：

①先画出判定树如下（注：mid=⌊(1+12)/2⌋=6）：



②查找元素 54，需依次与 30, 63, 42, 54 元素比较；

③查找元素 90，需依次与 30, 63,87, 95 元素比较；

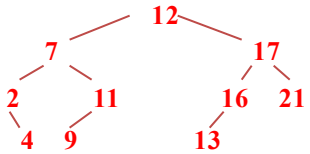
④求 ASL 之前，需要统计每个元素的查找次数。判定树的前 3 层共查找 1＋2×2＋4×3=17 次；

但最后一层未满，不能用 8×4，只能用 5×4=20 次，

所以 ASL＝1/12（17＋20）＝37/12≈3.08

（2）在一棵空的二叉排序树中依次插入关键字序列为 12，7，17，11，16，2，13，9，21，4，请画出所得到的二叉排序树。

答案：



验算方法：　用中序遍历应得到排序结果：2,4,7,9,11,12,13,16,17,21

（3）已知如下所示长度为 12 的表：（Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec）

① 试按表中元素的顺序依次插入一棵初始为空的二叉排序树，画出插入完成之后的二叉排序树，并求其在等概率的情况下查找成功的平均查找长度。

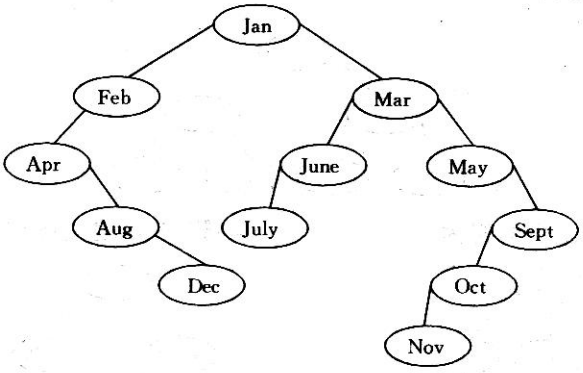
② 若对表中元素先进行排序构成有序表，求在等概率的情况下对此有序表进行折半查找时查找成功的平均查找长度。

③ 按表中元素顺序构造一棵平衡二叉排序树，并求其在等概率的情况下查找成功的平均查找长度。

答案：

（1）求得的二叉排序树如下图所示，在等概率情况下查找成功的平均查找长度为

$$ASL_{succ}=\frac{1}{12}(1\times 1+2\times 2+3\times 3+4\times 3+5\times 2+6\times 1)=\frac{42}{12}$$



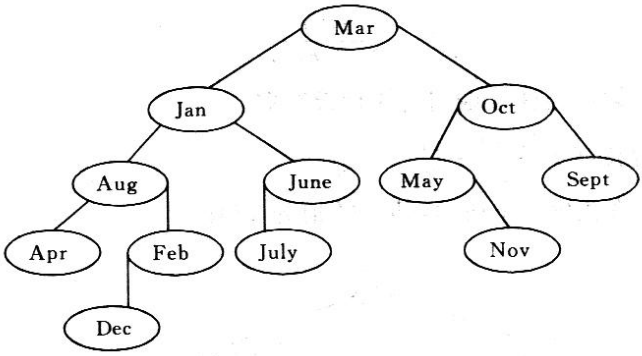
(2) 经排序后的表及在折半查找时找到表中元素所经比较的次数对照如下：

Apr	Aug	Dec	Feb	Jan	July	June	Mar	May	Nov	Oct	Sept
3	4	2	3	4	1	3	4	2	4	3	4

等概率情况下查找成功时的平均查找长度为

$$ASL_{succ} = \frac{1}{12}(1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 5) = \frac{37}{12}$$

(3) 平衡二叉树为



它在等概率情况下的平均查找长度为

$$ASL = \frac{1}{12}(1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 4 + 5 \times 1) = \frac{38}{12}$$

(4) 对图 7.31 所示的 3 阶 B-树，依次执行下列操作，画出各步操作的结果。

① 插入 90　② 插入 25　③ 插入 45　④ 删除 60

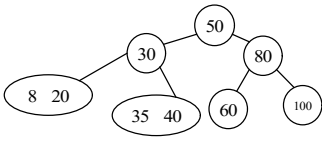
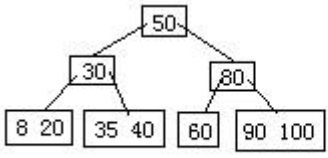
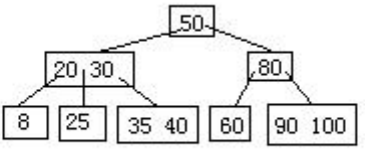


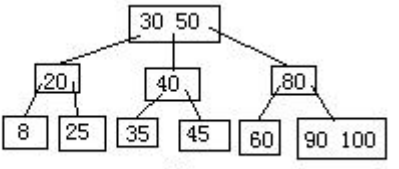
图 7.31　3 阶 B-树



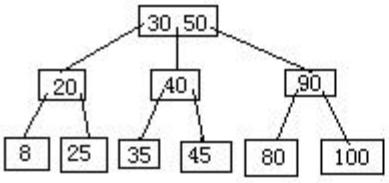
(1) 插入90



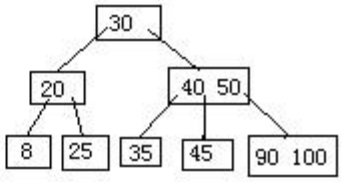
(2) 插入25



(3) 插入45



（4）删除60



（5）删除80

（5）设哈希表的地址范围为 0～17，哈希函数为：H（key）=key%16。用线性探测法处理冲突，输入关键字序列：（10，24，32，17，31，30，46，47，40，63，49），构造哈希表，试回答下列问题：

① 画出哈希表的示意图；

② 若查找关键字 63，需要依次与哪些关键字进行比较？

③ 若查找关键字 60，需要依次与哪些关键字比较？

④ 假定每个关键字的查找概率相等，求查找成功时的平均查找长度。

答案：

①画表如下：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
32	17	63	49					24	40	10				30	31	46	47

②查找 63,首先要与 H(63)=63%16=15 号单元内容比较，即 63 与 31 比较 ,不匹配；然后顺移，与 46,47,32,17,63 相比，一共比较了 6 次！

③查找 60,首先要与 H(60)=60%16=12 号单元内容比较，但因为 12 号单元为空（应当有空标记），所以应当只比较这一次即可。

④对于黑色数据元素，各比较 1 次；共 6 次；

对红色元素则各不相同，要统计移位的位数。“63”需要 6 次，“49”需要 3 次，“40”需要 2 次，“46”需要 3 次，“47”需要 3 次，

所以 ASL=1/11（6＋2＋3×3+6）＝23/11

（6）设有一组关键字（9，01，23，14，55，20，84，27），采用哈希函数：H（key）=key %7 ，表长为 10，用开放地址法的二次探测法处理冲突。要求：对该关键字序列构造哈希表，并计算查找成功的平均查找长度。

答案：

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	14	1	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）设哈希函数 H（K）=3 K mod 11，哈希地址空间为 0～10，对关键字序列（32，13，49，24，38，21，4，12），按下述两种解决冲突的方法构造哈希表，并分别求出等概率下查找成功时和查找失败时的平均查找长度 ASL_{succ} 和 ASL_{unsucc}。

① 线性探测法；

② 链地址法。

答案：

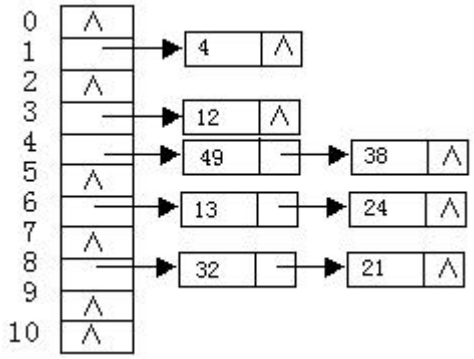
①

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



ASL_{succ} =（1*5+2*3）/8=11/8

ASL_{unsucc}=（1+2+1+2+3+1+3+1+3+1+1）/11=19/11

3．算法设计题

（1）试写出折半查找的递归算法。

[算法描述]

```
int  BinSrch（rectype r[ ], int  k, low, high）
//在长为 n 的有序表中查找关键字 k，若查找成功，返回 k 所在位置，查找失败返回 0。
{if（low≤high）  //low 和 high 分别是有序表的下界和上界
  {mid=（low+high）/2;
   if（r[mid].key==k）return  （mid）;
   else if（r[mid].key>k）return  （BinSrch（r,k, mid+1, high））;
        else return  （BinSrch（r,k, low, mid-1））;
   }
```

```
    else  return  (0); //查找失败。
  }//算法结束
```

（2）试写一个判别给定二叉树是否为二叉排序树的算法。

[题目分析] 根据二叉排序树中序遍历所得结点值为增序的性质，在遍历中将当前遍历结点与其前驱结点值比较，即可得出结论，为此设全局指针变量 pre（初值为 null）和全局变量 flag，初值为 true。若非二叉排序树，则置 flag 为 false。

[算法描述]

```
#define true 1
#define false 0
typedef struct node
{datatype data; struct node *lchild,*rchild;} *BTree;
void  JudgeBST(BTree T,int  flag)
// 判断二叉树是否是二叉排序树，本算法结束后，在调用程序中由 flag 得出结论。
{ if(T!=null && flag)
     { Judgebst(T->lchild,flag); // 中序遍历左子树
       if(pre==null) pre=T; // 中序遍历的第一个结点不必判断
       else  if(pre->data<T->data) pre=T; //前驱指针指向当前结点
               else{flag=flase; }   //不是完全二叉树
       Judgebst (T->rchild,flag); // 中序遍历右子树
}//JudgeBST 算法结束
```

（3）已知二叉排序树采用二叉链表存储结构，根结点的指针为 T，链结点的结构为（lchild,data,rchild），其中 lchild，rchild 分别指向该结点左、右孩子的指针，data 域存放结点的数据信息。请写出递归算法，从小到大输出二叉排序树中所有数据值>=x 的结点的数据。要求先找到第一个满足条件的结点后，再依次输出其他满足条件的结点。

[题目分析]本题算法之一是如上题一样，中序遍历二叉树，在“访问根结点”处判断结点值是否≥x，如是则输出，并记住第一个≥x 值结点的指针。这里给出另一个算法，利用二叉排序树的性质，如果根结点的值>=x,则除左分枝中可能有<x 的结点外都应输出。所以从根结点开始查找，找到结点值<x 的结点后，将其与双亲断开输出整棵二叉排序树。如果根结点的值<x,则沿右子树查找第一个≥x 的结点，找到后，与上面同样处理。

```
    void  Print(BSTree  t)
       // 中序输出以 t 为根的二叉排序树的结点
     {if(t) {Print(t->lchild);
             Cout<<t-data;
             Print(t->rchild);
            }
     }
    void  PrintAllx(BSTree bst, datatype x)
```

```
        //在二叉排序树 bst 中，查找值≥x 的结点并输出
{p=bst;
 if（p）
      {while（p && p->data<x）p=p->rchild；//沿右分枝找第一个值≥x 的结点
       bst=p; //bst 所指结点是值≥x 的结点的树的根
       if（p）
         {f=p; p=p->lchild ；//找第一个值<x 的结点
          while（p && p->data≥x）//沿左分枝向下，找第一个值<x 的结点
            {f=p; p=p->lchild ；} //f 是 p 的双亲结点的指针，指向第一个值≥x 的结点
             if(p) f->lchild=null; //双亲与找到的第一个值<x 的结点断开
          Print(bst)；//输出以 bst 为根的子树
            }//while
         }//内层 if（p）
      }//第一层 if（p）
 }//PrintAllx
```

（4）已知二叉树 T 的结点形式为（lling,data,count,rlink），在树中查找值为 X 的结点，若找到，则记数（count）加 1，否则，作为一个新结点插入树中，插入后仍为二叉排序树，写出其非递归算法。

[算法描述]

```
void SearchBST(BiTree &T,int target){
 BiTree s,q,f;   //以数据值 target,新建结点 s
 s=new BiTNode;
 s->data.x=target;
 s->data.count=0;
 s->lchild=s->rchild=NULL;

 if(!T){
      T=s;
      return ;
 }    //如果该树为空则跳出该函数
 f=NULL;
 q=T;
 while (q){
      if (q->data.x==target){
           q->data.count++;
           return ;
      }  //如果找到该值则计数加一
```

```
        f=q;
        if (q->data.x>target)     //如果查找值比目标值大，则为该树左孩子
            q=q->lchild;
        else                      //否则为右孩子
            q=q->rchild;
    }  //将新结点插入树中
    if(f->data.x>target)
        f->lchild=s;
    else
        f->rchild=s;
}
```

（5）假设一棵平衡二叉树的每个结点都表明了平衡因子 b，试设计一个算法，求平衡二叉树的高度。

[题目分析]因为二叉树各结点已标明了平衡因子 b，故从根结点开始记树的层次。根结点的层次为 1，每下一层，层次加 1，直到层数最大的叶子结点，这就是平衡二叉树的高度。当结点的平衡因子 b 为 0 时，任选左右一分枝向下查找，若 b 不为 0，则沿左（当 b=1 时）或右（当 b=-1 时）向下查找。

[算法描述]

```
int   Height（BSTree t）
  // 求平衡二叉树 t 的高度
  {level=0; p=t;
   while（p）
    {level++;  // 树的高度增 1
  if（p->bf<0）p=p->rchild; //bf=-1 沿右分枝向下
  //bf 是平衡因子，是二叉树 t 结点的一个域，因篇幅所限，没有写出其存储定义
     else p=p->lchild;          //bf>=0 沿左分枝向下
  }//while
   return  （level）; //平衡二叉树的高度
} //算法结束
```

（6）分别写出在散列表中插入和删除关键字为 K 的一个记录的算法，设散列函数为 H，解决冲突的方法为链地址法。

[算法描述]

```
bool insert(){
 int data;
 cin>>data;
 int ant=hash(data);
 LinkList p=HT[ant];    //初始化散列表
```

```
 while (p->next){
       if(p->next->data==data)
            return false;
       p=p->next;
 }     //找到插入位置
 LinkList s;
 s=new LNode;
 s->data=data;
 s->next=p->next;
 p->next=s;   //插入该结点
 return true;
}

bool deletes(){
 int data;
 cin>>data;
 int ant=hash(data);
 LinkList p=HT[ant];   //初始化散列表
 while (p->next){
       if(p->next->data==data){
            LinkList s=p->next;
            p->next=s->next;
            delete s;   //删除该结点
            return true;
       }   //找到删除位置
       p=p->next;   //遍历下一个结点
 }
 return false;
}
```

第 8 章　排序

1．选择题

（1）从未排序序列中依次取出元素与已排序序列中的元素进行比较，将其放入已排序序列的正确位置上的方法，这种排序方法称为（　　）。

A．归并排序　　B．冒泡排序　　C．插入排序　　D．选择排序

答案：C

（2）从未排序序列中挑选元素，并将其依次放入已排序序列（初始时为空）的一端的方法，称为（　　）。

A．归并排序　　B．冒泡排序　　C．插入排序　　D．选择排序

答案：D

（3）对 n 个不同的关键字由小到大进行冒泡排序，在下列（　　）情况下比较的次数最多。

A．从小到大排列好的　　B．从大到小排列好的

C．元素无序　　D．元素基本有序

答案：B

解释：对关键字进行冒泡排序，关键字逆序时比较次数最多。

（4）对 n 个不同的排序码进行冒泡排序，在元素无序的情况下比较的次数最多为（　　）。

A．n+1　　B．n　　C．n-1　　D．n(n-1)/2

答案：D

解释：比较次数最多时，第一次比较 n-1 次，第二次比较 n-2 次……最后一次比较 1 次，即(n-1)+(n-2)+…+1= n(n-1)/2。

（5）快速排序在下列（　　）情况下最易发挥其长处。

A．被排序的数据中含有多个相同排序码

B．被排序的数据已基本有序

C．被排序的数据完全无序

D．被排序的数据中的最大值和最小值相差悬殊

答案：C

解释：B 选项是快速排序的最坏情况。

（6）对 n 个关键字作快速排序，在最坏情况下，算法的时间复杂度是（　　）。

A．O(n)　　B．$O(n^2)$　　C．$O(n\log_2 n)$　　D．$O(n^3)$

答案：B

解释：快速排序的平均时间复杂度为 $O(n\log_2 n)$，但在最坏情况下，即关键字基本排好序的情况下，时间复杂度为 $O(n^2)$。

（7）若一组记录的排序码为（46, 79，56，38，40，84），则利用快速排序的方法，以第一个记录为基准得到的一次划分结果为（　　）。

A．38，40，46，56，79，84　　B．40，38，46，79，56，84

C．40，38，46，56，79，84　　D．40，38，46，84，56，79

答案：C

（8）下列关键字序列中，（　　）是堆。

A．16，72，31，23，94，53　　B．94，23，31，72，16，53

C．16，53，23，94，31，72　　D．16，23，53，31，94，72

答案：D

解释：D 选项为小根堆

（9）堆是一种（　　）排序。

A．插入　　B．选择　　C．交换　　D．归并

答案：B

（10）堆的形状是一棵（　　）。

A．二叉排序树　　B．满二叉树　　C．完全二叉树　　D．平衡二叉树

答案：C

（11）若一组记录的排序码为（46，79，56，38，40，84），则利用堆排序的方法建立的初始堆为（　　）。

A．79，46，56，38，40，84　　B．84，79，56，38，40，46

C．84，79，56，46，40，38　　D．84，56，79，40，46，38

答案：B

（12）下述几种排序方法中，要求内存最大的是（　　）。

A．希尔排序　　B．快速排序　　C．归并排序　　D．堆排序

答案：C

解释：堆排序、希尔排序的空间复杂度为 O(1)，快速排序的空间复杂度为 $O(\log_2 n)$，归并排序的空间复杂度为 O(n)。

（13）下述几种排序方法中，（　　）是稳定的排序方法。

A．希尔排序　　B．快速排序　　C．归并排序　　D．堆排序

答案：C

解释：不稳定排序有希尔排序、简单选择排序、快速排序、堆排序；稳定排序有直接插入排序、折半插入排序、冒泡排序、归并排序、基数排序。

（14）数据表中有 10000 个元素，如果仅要求求出其中最大的 10 个元素，则采用(　　)算法最节省时间。

A．冒泡排序　　B．快速排序　　C．简单选择排序　　D．堆排序

答案：D

（15）下列排序算法中，（　　）不能保证每趟排序至少能将一个元素放到其最终的位置上。

A．希尔排序　　B．快速排序　　C．冒泡排序　　D．堆排序

答案：A

解释：快速排序的每趟排序能将作为枢轴的元素放到最终位置；冒泡排序的每趟排序能将最大或最小的元素放到最终位置；堆排序的每趟排序能将最大或最小的元素放到最终位置。

2．应用题

（1）设待排序的关键字序列为{12，2，16，30，28，10，16*，20，6，18}，试分别写出使用以下排序方法，每趟排序结束后关键字序列的状态。

① 直接插入排序

② 折半插入排序

③ 希尔排序（增量选取 5，3，1）

④ 冒泡排序

⑤ 快速排序

⑥ 简单选择排序

⑦ 堆排序

⑧ 二路归并排序

答案：

①直接插入排序

[2 12] 16 30 28 10 16* 20 6 18
[2 12 16] 30 28 10 16* 20 6 18
[2 12 16 30] 28 10 16* 20 6 18
[2 12 16 28 30] 10 16* 20 6 18
[2 10 12 16 28 30] 16* 20 6 18
[2 10 12 16 16* 28 30] 20 6 18
[2 10 12 16 16* 20 28 30] 6 18
[2 6 10 12 16 16* 20 28 30] 18
[2 6 10 12 16 16* 18 20 28 30]

② 折半插入排序 排序过程同①

③ 希尔排序（增量选取 5，3，1）

10 2 16 6 18 12 16* 20 30 28 （增量选取 5）
6 2 12 10 18 16 16* 20 30 28 （增量选取 3）
2 6 10 12 16 16* 18 20 28 30 （增量选取 1）

④ 冒泡排序

2 12 16 28 10 16* 20 6 18 [30]
2 12 16 10 16* 20 6 18 [28 30]
2 12 10 16 16* 6 18 [20 28 30]
2 10 12 16 6 16* [18 20 28 30]

2 10 12 6 16 [16* 18 20 28 30]
2 10 6 12 [16 16* 18 20 28 30]
2 6 10 [12 16 16* 18 20 28 30]
2 6 10 12 16 16* 18 20 28 30]

⑤ 快速排序

12 [6 2 10] **12** [28 30 16* 20 16 18]
6 [2] **6** [10] 12 [28 30 16* 20 16 18]
28 2 6 10 12 [18 16 16* 20] **28** [30]
18 2 6 10 12 [16* 16] **18** [20] 28 30
16* 2 6 10 12 16* [16] 18 20 28 30
左子序列递归深度为 1，右子序列递归深度为 3

⑥ 简单选择排序

2 [12 16 30 28 10 16* 20 6 18]
2 6 [16 30 28 10 16* 20 12 18]
2 6 10 [30 28 16 16* 20 12 18]
2 6 10 12 [28 16 16* 20 30 18]
2 6 10 12 16 [28 16* 20 30 18]
2 6 10 12 16 16* [28 20 30 18]
2 6 10 12 16 16* 18 [20 30 28]
2 6 10 12 16 16* 18 20 [28 30]
2 6 10 12 16 16* 18 20 28 [30]

⑧ 二路归并排序

2 12 16 30 10 28 16 * 20 6 18
2 12 16 30 10 16* 20 28 6 18
2 10 12 16 16* 20 28 30 6 18
2 6 10 12 16 16* 18 20 28 30

（2）给出如下关键字序列｛321，156，57，46，28，7，331，33，34，63｝，试按链式基数排序方法，列出每一趟分配和收集的过程。

答案：

按最低位优先法 →321→156→57→46→28→7→331→33→34→63

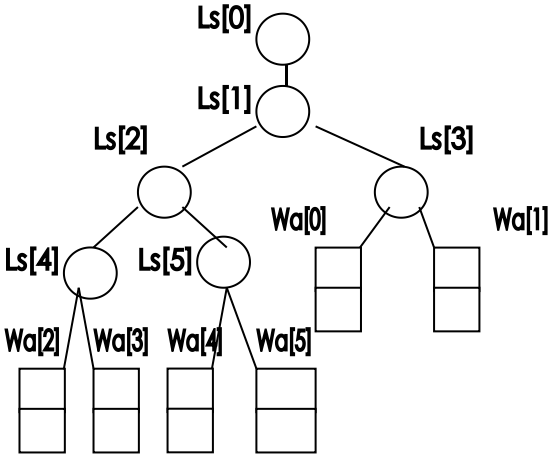
分配	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		321		33	34		156	57	28	
		331		63			46	7		

收集　→321→331→33→63→34→156→46→57→7→28

（3）对输入文件（101，51，19，61，3，71，31，17，19，100，55，20，9，30，50，6，90）；当 k=6 时，使用置换-选择算法，写出建立的初始败者树及生成的初始归并段。

答案：

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

3．算法设计题

（1）试以单链表为存储结构，实现简单选择排序算法。

[算法描述]：

```
void LinkedListSelectSort(LinkedList head)
//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!=null)
      {q=p->next;  r=p;     //设 r 是指向关键字最小的结点的指针
       while (q!=null)
        {if(q->data<r->data) r=q;
    q:=q->next;
   }
       if(r!=p)  r->data<-->p->data;
       p=p->next;
      }
```

（2）有 n 个记录存储在带头结点的双向链表中，现用双向冒泡排序法对其按上升序进行排序，请写出这种排序的算法。（注：双向冒泡排序即相邻两趟排序向相反方向冒泡）。

[算法描述]：

```
typedef struct node
   { ElemType data;
     struct node *prior,*next;
   }node, *DLinkedList;
  void  TwoWayBubbleSort(DLinkedList la)
    //对存储在带头结点的双向链表 la 中的元素进行双向起泡排序。
    {int exchange=1;  // 设标记
     DLinkedList p,temp,tail;
     head=la          //双向链表头，算法过程中是向下起泡的开始结点
     tail=null;       //双向链表尾，算法过程中是向上起泡的开始结点
     while (exchange)
      {p=head->next;     //p 是工作指针，指向当前结点
       exchange=0;       //假定本趟无交换
       while (p->next!=tail)  // 向下（右）起泡，一趟有一最大元素沉底
        if (p->data>p->next->data) //交换两结点指针，涉及 6 条链
         {temp=p->next; exchange=1;//有交换
          p->next=temp->next;temp->next->prior=p //先将结点从链表上摘下
          temp->next=p; p->prior->next=temp;      //将 temp 插到 p 结点前
          temp->prior=p->prior; p->prior=temp;
          }
         else p=p->next; //无交换，指针后移
         tail=p; //准备向上起泡
         p=tail->prior;
        while (exchange && p->prior!=head)
        //向上（左）起泡，一趟有一最小元素冒出
          if (p->data<p->prior->data)        //交换两结点指针，涉及 6 条链
            {temp=p->prior; exchange=1;      //有交换
             p->prior=temp->prior;temp->prior->next=p;
             //先将 temp 结点从链表上摘下
             temp->prior=p; p->next->prior=temp; //将 temp 插到 p 结点后（右）
             temp->next=p->next; p->next=temp;
             }
           else p=p->prior;  //无交换，指针前移
         head=p;             //准备向下起泡
```

```
    }// while (exchange)
  } //算法结束
```

（3）设有顺序放置的 n 个桶，每个桶中装有一粒砾石，每粒砾石的颜色是红，白，蓝之一。要求重新安排这些砾石，使得所有红色砾石在前，所有白色砾石居中，所有蓝色砾石居后，重新安排时对每粒砾石的颜色只能看一次，并且只允许交换操作来调整砾石的位置。

[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。

[算法描述]:

```
void QkSort(rectype r[],int n) {
  // r 为含有 n 个元素的线性表，元素是具有红、白和兰色的砾石，用顺序存储结构存储，
  //本算法对其排序，使所有红色砾石在前，白色居中，兰色在最后。
  int i=1,j=1,k=n,temp;
  while (k!=j){
    while (r[k].key==3) k--;// 当前元素是兰色砾石，指针左移
    if (r[k].key==1)          // 当前元素是红色砾石
        if (i>=j){temp=r[k];r[k]=r[i];r[i]=temp; i++;}
                  //左侧只有红色砾石，交换 r[k]和 r[i]
        else      {temp=r[j];r[j]=r[i];r[i]=temp; j++;
                  //左侧已有红色和白色砾石，先交换白色砾石到位
                  temp=r[k];r[k]=r[i];r[i]=temp; i++;
                   //白色砾石（i 所指）和待定砾石（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。

（4）编写算法，对 n 个关键字取整数值的记录序列进行整理，以使所有关键字为负值的记录排在关键字为非负值的记录之前，要求：

① 采用顺序存储结构，至多使用一个记录的辅助存储空间；

② 算法的时间复杂度为 O(n)。

[算法描述]

```
void  process (int A[n]){
  low = 0;
  high = n-1;
  while ( low<high ){
    while (low<high && A[low]<0)
    low++;
    while (low<high && A[high]>0)
    high++;
    if (low<high){
      x=A[low];
      A[low]=A[high];
      A[high]=x;
      low++;
      high--;
    }
  }
    return;
```

```
}
```

（5）借助于快速排序的算法思想，在一组无序的记录中查找给定关键字值等于 key 的记录。设此组记录存放于数组 r[l..n]中。若查找成功，则输出该记录在 r 数组中的位置及其值，否则显示“not find”信息。请简要说明算法思想并编写算法。

[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。

[算法描述]

```
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;
     }
    cout<<“Not find”; return  0;
 }//index
```

（6）有一种简单的排序算法，叫做计数排序。这种排序算法对一个待排序的表进行排序，并将排序结果存放到另一个新的表中。必须注意的是，表中所有待排序的关键字互不相同，计数排序算法针对表中的每个记录，扫描待排序的表一趟，统计表中有多少个记录的关键字比该记录的关键字小。假设针对某一个记录，统计出的计数值为 c，那么，这个记录在新的有序表中的合适的存放位置即为 c。

① 给出适用于计数排序的顺序表定义；

② 编写实现计数排序的算法；

③ 对于有 n 个记录的表，关键字比较次数是多少？

④ 与简单选择排序相比较，这种方法是否更好？为什么？

[算法描述]

①
```
typedef struct
   {int key;
    datatype info
   }RecType
```

②
```
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++;
```