何谓数据结构

数据结构是在整个计算机科学与技术领域上广泛被使用的术语。它用来反映一个数据的内部构成，即一个数据由那些成分数据构成，以什么方式构成，呈什么结构。数据结构有逻辑上的数据结构和物理上的数据结构之分。逻辑上的数据结构反映成分数据之间的逻辑关系，而物理上的数据结构反映成分数据在计算机内部的存储安排。数据结构是数据存在的形式。

数据结构是信息的一种组织方式，其目的是为了提高算法的效率，它通常与一组算法的集合相对应，通过这组算法集合可以对数据结构中的数据进行某种操作。

数据结构主要研究什么？

数据结构作为一门学科主要研究数据的各种逻辑结构和存储结构，以及对数据的各种操作。因此，主要有三个方面的内容：数据的逻辑结构；数据的物理存储结构；对数据的操作（或算法）。通常，算法的设计取决于数据的逻辑结构，算法的实现取决于数据的物理存储结构。

什么是数据结构？什么是逻辑结构和物理结构？

数据是指由有限的符号（比如，"0"和"1"，具有其自己的结构、操作、和相应的语义）组成的元素的集合。**结构**是元素之间的关系的集合。通常来说，一个**数据结构 *DS*** 可以表示为一个二元组：

DS=(D,S), //i.e., *data-structure=(data-part,logic-structure-part)*

这里 *D* 是数据元素的集合（或者是“结点”，可能还含有“数据项”或“数据域”），*S* 是定义在 *D*（或其他集合）上的关系的集合，*S* = { *R* | *R* : *D*×*D*×...}，称之为元素的**逻辑结构**。

逻辑结构有四种基本类型：集合结构、线性结构、树状结构和网络结构。表和树是最常用的两种高效数据结构，许多高效的算法可以用这两种数据结构来设计实现。表是线性结构的（全序关系），树(偏序或层次关系)和图（局部有序(weak/local orders)）是非线性结构。

数据结构的物理结构是指逻辑结构的存储镜像(image)。数据结构 *DS* 的物理结构 *P* 对应于从 *DS* 的数据元素到存储区 *M*（维护着逻辑结构 *S*）的一个映射：

P:(*D*,*S*) --> *M*

存储器模型：一个存储器 *M* 是一系列固定大小的存储单元，每个单元 *U* 有一个唯一的地址 *A*(*U*)，该地址被连续地编码。每个单元 *U* 有一个唯一的后继单元 *U'*=succ(*U*)。

***P* 的四种基本映射模型：**顺序（sequential）、链接（linked）、索引（indexed）和散列（hashing）映射。

因此，我们至少可以得到 4×4 种可能的物理数据结构：

sequential	**(sets)**
linked	**lists**
indexed	**trees**
hash	**graphs**

（并不是所有的可能组合都合理）

数据结构 *DS* 上的操作：所有的定义在 *DS* 上的操作在改变数据元素（节点）或节点的域时必须保持 *DS* 的逻辑和物理结构。

DS 上的基本操作：任何其他对 *DS* 的高级操作都可以用这些基本操作来实现。最好将 DS 和他的所有基本操作看作一个整体——称之为**模块**。我们可以进一步将该模块抽象为数据类型（其中 *DS* 的存储结构被表示为私有成员，基本操作被表示为公共方法），称之为 **ADT**。作为 ADT，堆栈和队列都是一种特殊的表，他们拥有表的操作的子集。

对于 DATs 的高级操作可以被设计为（不封装的）算法，利用基本操作对 *DS* 进行处理。

好的和坏的 DS：如果一个 *DS* 可以通过某种“线性规则”被转化为线性的 *DS*（例如线性表），则称它为好的 *DS*。好的 *DS* 通常对应于好的（高效的）算法。这是由计算机的计算能力决定的，因为计算机本质上只能存取逻辑连续的内存单元，因此如何没有线性化的结构逻辑上是不可计算的。比如对一个图进行操作，要访问图的所有结点，则必须按照某种顺序来依次访问所有节点（要形成一个偏序），必须通过某种方式将图固有的非线性结构转化为线性结构才能对图进行操作。

树是好的 DS——它有非常简单而高效的线性化规则，因此可以利用树设计出许多非常高效的算法。树的实现和使用都很简单，但可以解决大量特殊的复杂问题，因此树是实际编程中最重要和最有用的一种数据结构。树的结构本质上有递归的性质——每一个叶节点可以被一棵子树所替代，反之亦然。实际上，每一种递归的结构都可以被转化为（或等价于）树形结构。

计算机中数据的描述方式

我们知道，数据可以用不同的形式进行描述或存储在计算机存储器中。最常见的数据描述方法有：公式化描述、链接描述、间接寻址和模拟指针。

- 公式化描述借助数学公式来确定元素表中的每个元素分别存储在何处，也就通过公式计算元素的存储器地址。最简单的情形就是把所有元素依次连续存储在一片连续的存储空间中，这就是通常所说的连续线性表，即数组。复杂一点的情形是利用复杂的函数关系根据元素的某些特征来计算元素在内存中的位置，这种技术称为散列技术(Hash，经常音译为哈希技术)。
- 在链接描述中，元素表中的每个元素可以存储在存储器的不同区域中，每个元素都包含一个指向下一个元素的指针。这就是通常所说的链表。这种描述方法的好处是，知道了第一个元素的位置，就可以依次找到第 n 个元素的位置，而且在其中插入元素非常方便，缺点是查找某个元素要遍历所有在该元素之前的元素，实际应用中经常和公式化描述结合起来使用。
- 在间接寻址方式中，元素表中的每个元素也可以存储在存储器的不同区域中，不同的是，此时必须保存一张表，该表的第 i 项指向元素表中的第 i 个元素，所以这张表是一个用来存储元素地址的表。指针数组（元素为指针的数组）就是这种描述法的应用。这种描述方法是公式化描述和链接描述的一种折衷方案，同时具有两种描述方法的优点，但是需要额外的内存开销。
- 模拟指针非常类似于链接描述，区别在于它用整数代替了指针，整数所扮演的角色与指针所扮演的角色完全相同。模拟指针的描述方式是链接描述和公式化描述的结合，元素被存储在不同的区域中，每个元素包含一个指示下一个元素位置的整数，可以通过某种公式由该整数计算出下一个元素的存储器地址。线性表的游标实现就是模拟指针描述法。

算法 Algorithm

算法是在有限步骤内求解某一问题所使用的一组定义明确的规则。通俗点说，就是计算机解题的过程。在这个过程中，无论是形成解题思路还是编写程序，都是在实施某种算法。前者是推理实现的算法，后者是操作实现的算法。

一个算法应该具有以下五个重要的特征：

1. **有穷性：** 一个算法必须保证执行有限步之后结束；
2. **确切性：** 算法的每一步骤必须有确切的定义；
3. **输入：**一个算法有 0 个或多个输入，以刻画运算对象的初始情况，所谓 0 个输入是指算法本身定除了初始条件；
4. **输出：**一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的；
5. **可行性：** 算法原则上能够精确地运行，而且人们用笔和纸做有限次运算后即可完成。

Did you know

Algorithm 一词的由来

Algorithm(算法)一词本身就十分有趣。初看起来，这个词好像是某人打算要写“Logarithm”(对数)一词但却把头四个字母写的前后颠倒了。这个词一直到 1957 年之前在 *Webster's New World Dictionary(《韦氏新世界词典》)*中还未出现，我们只能找到带有它的古代涵义的较老形式的“Algorism”(算术)，指的是用阿拉伯数字进行算术运算的过程。在中世纪时，珠算家用算盘进行计算，而算术家用算术进行计算。中世纪之后，对这个词的起源已经拿不准了，早期的语言学家试图推断它的来历，认为它是从把 algiros(费力的)+arithmos(数字)组合起来派生而成的，但另一些人则不同意这种说法，认为这个词是从“喀斯迪尔国王 Algor”派生而来的。最后，数学史学家发现了 algorism(算术)一词的真实起源：它来源于著名的 *Persian Textbook(《波斯教科书》)*的作者的名字 Abu Ja'far Mohammed ibn Mûsâ al-Khowârizm（约公元前 825 年）——从字面上看，这个名字的意思是“Ja'far 的父亲，Mohammed 和 Mûsâ 的儿子，Khowârizm 的本地人”。Khowârizm 是前苏联 ХИВА(基发)的小城镇 。Al-Khowârizm 写了著名的书 *Kitab al jabr w'al-muqabala (《复原和化简的规则》)*；另一个词，“algebra”(代数)，是从他的书的标题引出来的，尽管这本书实际上根本不是讲代数的。

逐渐地，“algorism”的形式和意义就变得面目全非了。如牛津英语字典所说明的，这个词是由于同 arithmetic(算术)相混淆而形成的错拼词。由 algorism 又变成 algorithm。一本早期的德文数学词典 *Vollstandiges Mathematisches Lexicon (《数学大全辞典》)* ，给出了 Algorithmus (算法)一词的如下定义：“在这个名称之下，组合了四种类型的算术计算的概念，即加法、乘法、减法、除法”。拉丁短语 *algorithmus infinitesimalis* (无限小方法) ，在当时就用来表示 Leibnitz(莱布尼兹)所发明的以无限小量进行计算的微积分方法。

1950 年左右，algorithm 一词经常地同欧几里德算法(Euclid's algorithm)联系在一起。这个算法就是在欧几里德的 *《几何原本》*(*Euclid's Elements ,第 VII 卷，命题 i 和 ii*)中所阐述的求两个数的最大公约数的过程(即辗转相除法)。

伪代码的使用 Usage of Pseudocode

伪代码(Pseudocode)是一种算法描述语言。使用为代码的目的是为了使被描述的算法可以容易地以任何一种编程语言(Pascal, C, Java, etc)实现。因此，伪代码必须结构清晰，代码简单，可读性好，并且类似自然语言。

下面介绍一种类 Pascal 语言的伪代码的语法规则。

伪代码的语法规则

1. 在伪代码中，每一条指令占一行(else if 例外，)，指令后不跟任何符号（Pascal 和 C 中语句要以分号结尾）；
2. 书写上的“缩进”表示程序中的分支程序结构。这种缩进风格也适用于 if-then-else 语句。用缩进取代传统 Pascal 中的 begin 和 end 语句来表示程序的块结构可以大大提高代码的清晰性；同一模块的语句有相同的缩进量，次一级模块的语句相对与其父级模块的语句缩进；

例如：

```
line 1
line 2
  sub line 1
  sub line 2
    sub sub line 1
   sub sub line 2
  sub line 3
line 3
```

而在 Pascal 中这种关系用 begin 和 end 的嵌套来表示，

```
 line 1
 line 2
 begin
   sub line 1
   sub line 2
   begin
     sub sub line 1
     sub sub line 2
   end;
   sub line 3
 end;
 line 3
```

在 C 中这种关系用{ 和 } 的嵌套来表示，

```
 line 1
 line 2
 {
    sub line 1
   sub line 2
   {
      sub sub line 1
   sub sub line 2
    }
   sub line 3
```

```
}
line 3
```

3. 在伪代码中，通常用连续的数字或字母来标示同一即模块中的连续语句，有时也可省略标号。

例如：

```
1. line 1
2. line 2
   a. sub line 1
   b. sub line 2
      1. sub sub line 1
      2. sub sub line 2
   c. sub line 3
3. line 3
```

4. 符号△后的内容表示注释；
5. 在伪代码中，变量名和保留字**不区分大小写**，这一点和 Pascal 相同，与 C 或 C++不同；
6. 在伪代码中，变量不需声明，但变量局部于特定过程，不能不加显示的说明就使用全局变量；
7. 赋值语句用符号←表示，x←exp 表示将 exp 的值赋给 x，其中 x 是一个变量，exp 是一个与 x 同类型的变量或表达式（该表达式的结果与 x 同类型）；多重赋值 i←j←e 是将表达式 e 的值赋给变量 i 和 j，这种表示与 j←e 和 i←e 等价。

例如：

```
x←y
x←20*(y+1)
x←y←30
```

以上语句用 Pascal 分别表示为：

```
x := y;
x := 20*(y+1);
x := 30; y := 30;
```

以上语句用 C 分别表示为：

```
x = y;
x = 20*(y+1);
x = y = 30;
```

8. 选择语句用 if-then-else 来表示，并且这种 if-then-else 可以嵌套，与 Pascal 中的 if-then-else 没有什么区别。

例如:

```
if (Condition1)
  then [ Block 1 ]
  else if (Condition2)
         then [ Block 2 ]
         else [ Block 3 ]
```

9. 循环语句有三种：while 循环、repeat-until 循环和 for 循环，其语法均与 Pascal 类似，只是用缩进代替 begin - end；

例如：

```
1. x ← 0
2. y ← 0
3. z ← 0
4. while x < N
     1. do x ← x + 1
     2.    y ← x + y
     3.    for t ← 0 to 10
             1. do z ← ( z + x * y ) / 100
             2.    repeat
                     1. y ← y + 1
                     2. z ← z - y
             3.    until z < 0
     4.    z ← x * y
5. y ← y / 2
```

上述语句用 Pascal 来描述是：

```
 x := 0;
 y := 0;
 z := 0;
 while x < N do
 begin
   x := x + 1;
   y := x + y;
```

```
   for t := 0 to 10 do
   begin
     z := ( z + x * y ) / 100;
     repeat
       y := y + 1;
       z := z - y;
     until z < 0;
   end;
   z := x * y;
 end;
 y := y / 2;
```

上述语句用 C 或 C++来描述是：

```
x = y = z = 0;
while( z < N )
{
  x ++;
  y += x;
  for( t = 0; t < 10; t++ )
  {
     z = ( z + x * y ) / 100;
     do {
        y ++;
        z -= y;
     } while( z >= 0 );
  }
  z = x * y;
}
y /= 2;
```

10.　数组元素的存取有数组名后跟“[下标]”表示。例如 A[j]指示数组 A 的第 j 个元素。符号“ …”用来指示数组中值的范围。

例如：

A[1…j]表示含元素 A[1], A[2], … , A[j]的子数组；

11. 复合数据用对象(Object)来表示，对象由属性(attribute)和域(field)构成。域的存取是由域名后接由方括号括住的对象名表示。

例如：

数组可被看作是一个对象，其属性有 length，表示其中元素的个数，则 length[A]就表示数组 A 中的元素的个数。在表示数组元素和对象属性时都要用方括号，一般来说从上下文可以看出其含义。

用于表示一个数组或对象的变量被看作是指向表示数组或对象的数据的一个指针。对于某个对象 x 的所有域 f，赋值 y←x 就使 f[y]=f[x]，更进一步，若有 f[x]←3，则不仅有 f[x]=3，同时有 f[y]=3，换言之，在赋值 y←x 后，x 和 y 指向同一个对象。

有时，一个指针不指向任何对象，这时我们赋给他 **nil**。

12. 函数和过程语法与 Pascal 类似。

函数值利用 “return (函数返回值)” 语句来返回，调用方法与 Pascal 类似；过程用 “call 过程名”语句来调用；

例如：

```
1. x ← t + 10
2. y ← sin(x)
3. call CalValue(x,y)
```

参数用按值传递方式传给一个过程：被调用过程接受参数的一份副本，若他对某个参数赋值，则这种变化对发出调用的过程是不可见的。当传递一个对象时，只是拷贝指向该对象的指针，而不拷贝其各个域。

算法的复杂性

傅清祥 王晓东

算法与数据结构，电子工业出版社,1998

摘要

本文介绍了算法的复杂性的概念和衡量方法，并提供了一些计算算法的复杂性的渐近阶的方法。

目录

简介

算法的复杂性是算法效率的度量，是评价算法优劣的重要依据。一个算法的复杂性的高低体现在运行该算法所需要的计算机资源的多少上面，所需的资源越多，我们就说该算法的复杂性越高；反之，所需的资源越低，则该算法的复杂性越低。

计算机的资源，最重要的是时间和空间（即存储器）资源。因而，算法的复杂性有**时间复杂性**和**空间复杂性**之分。

不言而喻，对于任意给定的问题，设计出复杂性尽可能地的算法是我们在设计算法是追求的一个重要目标；另一方面，当给定的问题已有多种算法时，选择其中复杂性最低者，是我们在选用算法适应遵循的一个重要准则。因此，算法的复杂性分析对算法的设计或选用有着重要的指导意义和实用价值。

关于算法的复杂性，有两个问题要弄清楚：

1. 用怎样的一个量来表达一个算法的复杂性；
2. 对于给定的一个算法，怎样具体计算它的复杂性。

让我们从比较两对具体算法的效率开始。

比较两对算法的效率

考虑问题 1：已知不重复且已经按从小到大排好的 m 个整数的数组 A[1..m]（为简单起见。还设 $m=2^k$，k 是一个确定的非负整数）。对于给定的整数 c，要求寻找一个下标 i，使得 A[i]=c；若找不到，则返回一个 0。

问题 1 的一个简单的算法是：从头到尾扫描数组 A。照此，或者扫到 A 的第 i 个分量，经检测满足 A[i]=c；或者扫到 A 的最后一个分量，经检测仍不满足 A[i]=c。我们用一个函数 Search 来表达这个算法：

```
Function Search (c:integer):integer;
Var J:integer;
Begin
 J:=1; {初始化}
 {在还没有到达 A 的最后一个分量且等于 c 的分量还没有找到时，
 查找下一个分量并且进行检测}
 While (A[i]<c)and(j<m) do
         j:=j+1;
 If A[j]=c then search:=j {在数组 A 中找到等于 c 的分量，且此分量的下标为 j}
           else Search:=0; {在数组中找不到等于 c 的分量}
End;
```

容易看出，在最坏的情况下，这个算法要检测 A 的所有 m 个分量才能判断在 A 中找不到等于 c 的分量。

解决问题 1 的另一个算法利用到已知条件中 A 已排好序的性质。它首先拿 A 的中间分量 A[m/2]与 c 比较，如果 A[m/2]=c 则解已找到。如果 A[m/2]>c，则 c 只可能在 A[1],A[2],..,A[m/2-1]之中，因而下一步只要在 A[1], A[2], .. ，A[m/2-1]中继续查找；如果 A[m/2]<c，则 c 只可能在 A[m/2+1],A[m/2+2],..,A[m]之中，因而下一步只要在 A[m/2+1],A[m/2+2],..,A[m]中继续查找。不管哪一种情形，都把下一步需要继续查找的范围缩小了一半。再拿这一半的子数组的中间分量与 c 比较，重复上述步骤。照此重复下去，总

有一个时候，或者找到一个 i 使得 A[i]=c，或者子数组为空（即子数组下界大于上界）。前一种情况找到了等于 c 的分量，后一种情况则找不到。

这个新算法因为有反复把供查找的数组分成两半，然后在其中一半继续查找的特征，我们称为二分查找算法。它可以用函数 B_Search 来表达：

```
Function B_Search ( c: integer):integer;
Var
 L,U,I : integer;  {U 和 L 分别是要查找的数组的下标的上界和下界}
 Found: boolean;
Begin
 L:=1; U:=m;   {初始化数组下标的上下界}
 Found:=false; {当前要查找的范围是 A[L]..A[U]。}
 {当等于 c 的分量还没有找到且 U>=L 时，继续查找}
 While (not Found) and (U>=L) do
  Begin
   I:=(U+L) div 2;  {找数组的中间分量}
   If c=A[I] then Found:=Ture
             else if c>A[I] then L:=I+1
                            else U:=I-1;
  End;
 If Found then B_Search:=1
          else B_Search:=0;
End;
```

容易理解，在最坏的情况下最多只要测 A 中的 k+1(k=logm,这里的 log 以 2 为底，下同)个分量，就判断 c 是否在 A 中。

算法 Search 和 B_Search 解决的是同一个问题，但在最坏的情况下（所给定的 c 不在 A 中），两个算法所需要检测的分量个数却大不相同，前者要 $m=2^k$ 个，后者只要 k+1 个。可见算法 B_Search 比算法 Search 高效得多。

以上例子说明：解同一个问题，算法不同，则计算的工作量也不同，所需的计算时间随之不同，即复杂性不同。

上图是运行这两种算法的时间曲线。该图表明，当 m 适当大（$m>m_0$）时，算法 B_Search 比算法 Search 省时，而且当 m 更大时，节省的时间急剧增加。

不过，应该指出：用实例的运行时间来度量算法的时间复杂性并不合适，因为这个实例时间与运行该算法的实际计算机性能有关。换句话说，这个实例时间不单纯反映算法的效率而是反映包括运行该算法的计算机在内的综合效率。我们引入算法复杂性的概念是为了比较解决同一个问题的不同算法的效率，而不想去比较运行该算法的计算机的性能。因而，不应该取算法运行的实例时间作为算法复杂性的尺度。我们希望，尽量单纯地反映作为算法精

髓的计算方法本身的效率，而且在不实际运行该算法的情况下就能分析出它所需要的时间和空间。

复杂性的计量

算法的复杂性是算法运行所需要的计算机资源的量，需要的时间资源的量称作时间复杂性，需要的空间（即存储器）资源的量称作空间复杂性。这个量应该集中反映算法中所采用的方法的效率，而从运行该算法的实际计算机中抽象出来。换句话说，这个量应该是只依赖于算法要解的问题的规模、算法的输入和算法本身的函数。如果分别用 N、I 和 A 来表示算法要解问题的规模、算法的输入和算法本身，用 C 表示算法的复杂性，那么应该有：

$$C=F(N,I,A)$$

其中 $F(N,I,A)$是 N,I,A 的一个确定的三元函数。如果把时间复杂性和空间复杂性分开，并分别用 T 和 S 来表示，那么应该有：

$$T=T(N,I,A) \quad (2.1)$$

和 $$S=S(N,I,A) \quad (2.2)$$

通常，我们让 A 隐含在复杂性函数名当中，因而将（2.1）和（2.2）分别简写为

$$T=T(N,I)$$

和 $$S=S(N,I)$$

由于时间复杂性和空间复杂性概念类同，计算方法相似，且空间复杂性分析相对地简单些，所以下文将主要地讨论时间复杂性。

下面以 $T(N,I)$为例，将复杂性函数具体化。

根据 $T(N,I)$的概念，它应该是算法在一台抽象的计算机上运行所需的时间。设此抽象的计算机所提供的元运算有 k 种，他们分别记为 $O_1,O_2,..,O_k$；再设这些元运算每执行一次所需要的时间分别为 $t_1,t_2,..,t_k$ 。对于给定的算法 A，设经过统计，用到元运算 O_i 的次数为 e_i，$i=1,2,..,k$ ，很明显，对于每一个 i，$1<=i<=k$，e_i 是 N 和 I 的函数，即 $e_i=e_i(N,I)$。那么有：

(2.3)

其中 t_i，i=1,2,..,k,是与 N,I 无关的常数。

显然，我们不可能对规模 N 的每一种合法的输入 I 都去统计 $e_i(N,I)$,i=1,2,…,k。因此 $T(N,I)$ 的表达式还得进一步简化，或者说，我们只能在规模为 N 的某些或某类有代表性的合法输入中统计相应的 e_i，i=$1,2,…,k$，评价时间复杂性。

下面只考虑三种情况的复杂性，即最坏情况、最好情况和平均情况下的时间复杂性，并分别记为 $T_{max}(N)$、$T_{min}(N)$和 $T_{avg}(N)$。在数学上有：

(2.4)

(2.5)

(2.6)

其中，D_N 是规模为 N 的合法输入的集合；I^* 是 D_N 中一个使 $T(N,I^*)$ 达到 $T_{max}(N)$ 的合法输入，

是 D_N 中一个使 $T(N,\ \ )$ 到 $T_{min}(N)$ 的合法输入；而 $P(I)$ 是在算法的应用中出现输入 I 的概率。

以上三种情况下的时间复杂性各从某一个角度来反映算法的效率，各有各的用处，也各有各的局限性。但实践表明可操作性最好的且最有实际价值的是最坏情况下的时间复杂性。下面我们将把对时间复杂性分析的主要兴趣放在这种情形上。

一般来说，最好情况和最坏情况的时间复杂性是很难计量的，原因是对于问题的任意确定的规模 N 达到了 $T_{max}(N)$ 的合法输入难以确定，而规模 N 的每一个输入的概率也难以预测或确定。我们有时也按平均情况计量时间复杂性，但那时在对 $P(I)$ 做了一些人为的假设（比如等概率）之后才进行的。所做的假设是否符合实际总是缺乏根据。因此，在最好情况和平均情况下的时间复杂性分析还仅仅是停留在理论上。

现在以上一章提到的问题 1 的算法 Search 为例来说明如何利用(2.4)-(2.6)对它的 T_{max}、T_{min} 和 T_{avg} 进行计量。这里问题的规模以 m 计算，算法重用到的元运算有赋值、测试和加法等三种，它们每执行一次所需的时间常数分别为 $a,t,$和 s 。对于这个例子，如假设 c 在 A 中，那么容易直接看出最坏情况的输入出现在 c=A[m]的情形，这时：

$$T_{max}(m)=a+2mt+(m-1)s+(m-1)a+t+a=(m+1)a+(2m+1)t+(m-1)s \quad (2.7)$$

而最好情况的输入出现在 c=A[1]的情形。这时：

(2.8)

至于 $T_{avg}(m)$，如前所述，必须对 D_m 上的概率分布做出假设才能计量。为简单起见，我们做最简单的假设：D_m 上的概率分布是均等的，即 $P(A[i]=c)=1/m$ 。若记 $T_i=T(m,I_i)$，其中 I_i 表示 $A[i]=c$ 的合法输入，那么：

(2.9)

而根据与(2.7)类似的推导，有：

代入(2.9) ，则：

这里碰巧有：

$T_{avg}(m)=(T_{max}(m)+T_{min}(m))/2$

但必须指出，上式并不具有一般性。

类似地，对于算法 B_Search 照样可以按(2.4)-(2.6)计算相应的 $T_{max}(m)$、$T_{min}(m)$和 $T_{avg}(m)$ 。不过，我们这里只计算 $T_{max}(m)$ 。为了与 Search 比较，仍假设 c 在 A 中，即最坏情况的输入仍出现在 c=A[m]时。这时，while 循环的循环体恰好被执行了 log$m$ +1 即 k+1 次。因为第一次执行时数据的规模为 m，第二次执行时规模为 m/2 等等，最后一次执行时规模为 1。另外，与 Search 少有不同的是这里除了用到赋值、测试和加法三种原运算外，还用到减法和除法两种元运算。补记后两种元运算每执行一次所需时间为 b 和 d ，则可以推演出：

(2.10)

比较(2.7)和(2.10) ，我们看到 m 充分大时，在最坏情况下 B_Search 的时间复杂性远小于 Search 的时间复杂性。

复杂性的渐近性态及其阶

随着经济的发展、社会的进步、科学研究的深入，要求用计算机解决的问题越来越复杂，规模越来越大。但是，如果对这类问题的算法进行分析用的是第二段所提供的方法，把所有的元运算都考虑进去，精打细算，那么，由于问题的规模很大且结构复杂，算法分析的工作量之大、步骤之繁将令人难以承受。因此，人们提出了对于规模充分大、结构又十分复杂的问题的求解算法，其复杂性分析应如何简化的问题。

我们先要引入复杂性渐近性态的概念。设 $T(N)$是在第二段中所定义的关于算法 A 的复杂性函数。一般说来，当 N 单调增加且趋于∞时,$T(N)$也将单调增加趋于∞。对于 $T(N)$，如果存在 $T'(N)$，使得当 $N\to\infty$时有：

$(T(N)-T'(N))/T(N)\to 0$

那么，我们就说 $T'(N)$是 $T(N)$当 $N\to\infty$时的**渐近性态**，或叫 $T'(N)$为算法 A 当 $N\to\infty$的**渐近复杂性**而与 $T(N)$相区别，因为在数学上，$T'(N)$是 $T(N)$当 $N\to\infty$时的渐近表达式。

直观上，$T'(N)$是 $T(N)$中略去低阶项所留下的主项。所以它无疑比 $T(N)$来得简单。比如当

$T(N)=3N^2+4N\log_2 N+7$

时，$T'(N)$的一个答案是 $3N^2$，因为这时有：

显然 $3N^2$ 比 $3N^2+4N\log_2 N+7$ 简单得多。

由于当 $N\to\infty$ 时 $T(N)$ 渐近于 $T'(N)$，我们有理由用 $T'(N)$ 来替代 $T(N)$ 作为算法 A 在 $N\to\infty$ 时的复杂性的度量。而且由于于 $T'(N)$ 明显地比 $T(N)$ 简单，这种替代明显地是对复杂性分析的一种简化。

进一步，考虑到分析算法的复杂性的目的在于比较求解同一问题的两个不同算法的效率，而当要比较的两个算法的渐近复杂性的阶不相同时，只要能确定出各自的阶，就可以判定哪一个算法的效率高。换句话说，这时的渐近复杂性分析只要关心 $T'(N)$ 的阶就够了，不必关心包含在 $T'(N)$ 中的常数因子。所以，我们常常又对 $T'(N)$ 的分析进--步简化，即假设算法中用到的所有不同的元运算各执行一次，所需要的时间都是一个单位时间。

综上所述，我们已经给出了简化算法复杂性分析的方法和步骤，即只要考察当问题的规模充分大时，算法复杂性在渐近意义下的阶。与此简化的复杂性分析方法相配套，需要引入五个渐近意义下的记号：O、Ω、θ、o 和 ω。

以下设 $f(N)$ 和 $g(N)$ 是定义在正数集上的正函数。

如果存在正的常数 C 和自然数 N_0，使得当 $N\geq N_0$ 时有 $f(N)\leq Cg(N)$。则称函数 $f(N)$ 当 N 充分大时上有界，且 $g(N)$ 是它的一个**上界**，记为 $f(N)=O(g(N))$。这时我们还说 $f(N)$ 的阶不高于 $g(N)$ 的阶。

举几个例子：

(1)因为对所有的 $N\geq 1$ 有 $3N\leq 4N$，我们有 $3N=O(N)$；

(2)因为当 $N\geq 1$ 时有 $N+1024\leq 1025N$，我们有 $N+1024=O(N)$；

(3)因为当 $N\geq 10$ 时有 $2N^2+11N-10\leq 3N^2$，我们有 $2N^2+11N-10=O(N^2)$；

(4)因为对所有 $N\geq 1$ 有 $N^2\leq N^3$，我们有 $N^2=O(N^3)$；

(5)作为一个反例 $N^3\neq O(N^2)$。因为若不然，则存在正的常数 C 和自然数 N_0，使得当 $N\geq N_0$ 时有 $N^3\leq C\,N^2$，即 $N\leq C$ 。显然当取 $N=\max(N_0,[C]+1)$ 时这个不等式不成立，所以 $N^3\neq O(N^2)$。

按照大 O 的定义，容易证明它有如下运算规则：

1. $O(f)+O(g)=O(\max(f,g))$；
2. $O(f)+O(g)=O(f+g)$；
3. $O(f)\cdot O(g)=O(f\cdot g)$；
4. 如果 $g(N)=O(f(N))$，则 $O(f)+O(g)=O(f)$；
5. $O(Cf(N))=O(f(N))$，其中 C 是一个正的常数；
6. $f=O(f)$；

规则 1 的证明：

设 $F(N)=O(f)$ 。根据记号 O 的定义，存在正常数 C_1 和自然数 N_1，使得对所有的 $N\geq N_1$，有 $F(N)\leq C_1 f(N)$。类似地，设 $G(N)=O(g)$，则存在正的常数 C_2 和自然数 N_2 使得对所有的 $N\geq N_2$ 有 $G(N)\leq C_2 g(N)$，今令：

$C_3=\max(C_1, C_2)$

$N_3=\max(N_1, N_2)$

和对任意的非负整数 N，

$h(N)=\max(f,g)$,

则对所有的 $N\geq N_3$ 有：

$$F(N)\leq C_1 f(N)\leq C_1 h(N)\leq C_3 h(N)$$

类似地，有：

$$G(N)\leq C_2 g(N)\leq C_2 h(N)\leq C_3 h(N)$$

因而

$$\begin{aligned}O(f)+O(g) &= F(N)+G(N)\leq C_3 h(N)+C_3 h(N)\\ &= 2C_3 h(N)\\ &= O(h)\\ &= O(\max(f,g))\end{aligned}$$

其余规则的证明类似，请读者自行证明。

应用这些规则的一个例子：对于第一章中的算法 search，在第二章给出了它的最坏情况下时间复杂性 $T_{\max}(m)$ 和平均情况下的时间复杂性 $T_{avg}(m)$ 的表达式。如果利用上述规则，立即有:

$$T_{\max}(m)=O(m)$$

和 $T_{avg}(m)=O(m)+O(m)+O(m)=O(m)$

另一个例子：估计下面二重循环算法段在最坏情况下的时间复杂性 $T(N)$的阶。

```
for i:=l to N do
  for j:=1 to i do
    begin
     S1;
     S2;
     S3;
     S4;
    end;
```

其中 S_k (k=1,2,3,4)是单一的赋值语句。对于内循环体，显然只需 O(l)时间。因而内循环只需

时间。累加起来便是外循环的时间复杂性：

应该指出，根据记号 O 的定义，用它评估算法的复杂性，得到的只是当规模充分大时的一个上界。这个上界的阶越低则评估就越精确，结果就越有价值。

关于记号 Ω，文献里有两种不同的定义。本文只采用其中的一种，定义如下：如果存在正的常数 C 和自然数 N_0，使得当 $N\geq N_0$ 时有 $f(N)\geq Cg(N)$，则称函数 $f(N)$当 N 充分大时**下有界**，且 $g(N)$是它的一个**下界**，记为 $f(N)=\Omega(g(N))$。这时我们还说 $f(N)$的阶不低于 $g(N)$的阶。

Ω 的这个定义的优点是与 O 的定义对称，缺点是当 $f(N)$对自然数的不同无穷子集有不同的表达式，且有不同的阶时，未能很好地刻画出 $f(N)$的下界。比如当：

时，如果按上述定义，只能得到 $f(N)=\Omega(1)$，这是一个平凡的下界，对算法分析没有什么价值。

然而，考虑到 Ω 的上述定义有与 O 的定义的对称性，又考虑到常用的算法都没出现上例中那种情况，所以本文还是选用它。

我们同样也可以列举 Ω 的一些运算规则。但这里从略，只提供一个应用的例子。还是考虑算法 Search 在最坏情况下的时间复杂性函数 $T_{max}(m)$。由它的表达式(2.7)及已知 a,s,t 均为大于 0 的常数，可推得，当 $m\geq 1$ 时有：

$T_{max}(m)\geq(m+1)a+(2m+1)t>ma+2mt=(a+2t)m$ ，

于是 $T_{max}(m)=\Omega(m)$。

我们同样要指出，用 Ω 评估算法的复杂性，得到的只是该复杂性的一个下界。这个下界的阶越高，则评估就越精确，结果就越有价值。再则，这里的 Ω 只对问题的一个算法而言。如果它是对一个问题的所有算法或某类算法而言，即对于一个问题和任意给定的充分大的规模 N，下界在该问题的所有算法或某类算法的复杂性中取，那么它将更有意义。这时得到的相应下界，我们称之为问题的下界或某类算法的下界。它常常与 O 配合以证明某问题的一个特定算法是该问题的最优算法或该问题在某算法类中的最优算法。

明白了记号 O 和 Ω 之后，记号 θ 将随之清楚，因为我们定义 $f(N)=\theta(g(N))$ 则 $f(N)=O(g(N))$ 且 $f(N)=\Omega(g(N))$。这时，我们说 $f(N)$ 与 $g(N)$ **同阶**。比如，对于算法 Search 在最坏情况下的时间复杂性 $T_{\max}(m)$。已有 $T_{\max}(m)=O(m)$ 和 $T_{\max}(m)=\Omega(m)$，所以有 $T_{max}(m)=\theta(m)$，这是对 $T_{\max}(m)$ 的阶的精确估计。

最后，如果对于任意给定的 $\varepsilon\geq 0$，都存在非负整数 N_0，使得当 $N\geq N_0$ 时有 $f(N)\leq \varepsilon g(N)$，则称函数 $f(N)$ 当 N 充分大时比 $g(N)$ **低阶**，记为 $f(N)=o(g(N))$，例如：

$4N\log N+7=o(3N^2+4N\log N+7)$；而 $f(N)=\omega(g(N))$ 定义为 $g(N)=o(f(N))$。

即当 N 充分大时 $f(N)$ 的阶比 $g(N)$ 高。我们看到 o 对于 O 有如 ω 对于 Ω。

复杂性渐近阶的重要性

计算机的设计和制造技术在突飞猛进，一代又一代的计算机的计算速度和存储容量在直线增长。有的人因此认为不必要再去苦苦地追求高效率的算法，从而不必要再去无谓地进行复杂性的分析。他们以为低效的算法可以由高速的计算机来弥补，以为在可接受的一定时间内用低效的算法完不成的任务，只要移植到高速的计算机上就能完成。这是一种错觉。造成这种错觉的原因是他们没看到：随着经济的发展、社会的进步、科学研究的深入，要求计算机解决的问题越来越复杂、规模越来越大，也呈线性增长之势；而问题复杂程度和规模的线性增长导致的时耗的增长和空间需求的增长，对低效算法来说，都是超线性的，决非计算机速度和容量的线性增长带来的时耗减少和存储空间的扩大所能抵销。事实上，我们只要对效率上有代表性的几个档次的算法作些简单的分析对比就能明白这一点。

我们还是以时间效率为例。设 $A_1,A_2,\ldots$ 和 A_6。是求解同一间题的 6 个不同的算法，它们的渐近时间复杂性分别为 N，$N\log N$，N^2，N^3，2^N，$N!$。让这六种算法各在 C_1 和 C_2 两台计算机上运行，并设计算机 C_2 的计算速度是计算机 C_1 的 10 倍。在可接受的一段时间内，设在 C_1 上算法 A_i 可能求解的问题的规模为 N_{1i}，而在 C_2 上可能求解的问题的规模为 N_{2i}，那么，我们就应该有 $T_i(N_{2i})=10T_i(N_{1i})$，其中 $T_i(N)$ 是算法 A_i 渐近的时间复杂性，i=1,2,…,6。分别解出 N_{2i} 和 N_{1i} 的关系，可列成下表：

表 4-1 算法与渐近时间复杂性的关系

算法	渐进时间复杂性 $T(N)$	在 C_1 上可解的规模 N_1	在 C_2 上可解的规模 N_2	N_1 和 N_2 的关系
A_1	N	N_{11}	N_{21}	$N_{21}=10N_{11}$
A_2	$N\log N$	N_{12}	N_{22}	$N_{22}\approx 10N_{12}$
A_3	N^2	N_{13}	N_{23}	
A_4	N^3	N_{14}	N_{24}	
A_5	2^N	N_{15}	N_{25}	$N_{25}=N_{15}+\log 10$
A_6	N!	N_{16}	N_{26}	$N_{26}=N_{16}$+小的常数

从表 4-1 的最后一列可以清楚地看到，对于高效的算法 A_1，计算机的计算速度增长 10 倍，可求解的规模同步增长 10 倍；对于 A_2，可求解的问题的规模的增长与计算机的计算速度的增长接近同步；但对于低效的算法 A_3，情况就大不相同，计算机的计算速度增长 10 倍只换取可求解的问题的规模增加 log10。当问题的规模充分大时，这个增加的数字是微不足道的。换句话说，对于低效的算法，计算机的计算速度成倍乃至数 10 倍地增长基本上不带来求解规模的增益。因此，对于低效算法要扩大解题规模，不能寄希望于移植算法到高速的计算机上，而应该把着眼点放在算法的改进上。

从表 4-l 的最后一列我们还看到，限制求解问题规模的关键因素是算法渐近复杂性的阶，对于表中的前四种算法，其渐近的时间复杂性与规模 N 的一个确定的幂同阶，相应地，计算机的计算速度的乘法增长带来的是求解问题的规模的乘法增长，只是随着幂次的提高，规

模增长的倍数在降低。我们把渐近复杂性与规模 N 的幂同阶的这类算法称为多项式算法。对于表中的后两种算法，其渐近的时间复杂性与规模 N 的一个指数函数同阶，相应地计算机的计算速度的乘法增长只带来求解问题规模的加法增长。我们把渐近复杂性与规模 N 的指数同阶的这类算法称为指数型算法。多项式算法和指数型算法是在效率上有质的区别的两类算法。这两类算法的区别的内在原因是算法渐近复杂性的阶的区别。可见，算法的渐近复杂性的阶对于算法的效率有着决定性的意义。所以在讨论算法的复杂性时基本上都只关心它的渐近阶。

多项式算法是有效的算法。绝大多数的问题都有多项式算法。但也有一些问题还未找到多项式算法，只找到指数型算法。

我们在讨论算法复杂性的渐近阶的重要性的同时，有两条要记住：

1. “复杂性的渐近阶比较低的算法比复杂性的渐近阶比较高的算法有效”这个结论，只是在问题的求解规模充分大时才成立。比如算法 A_4 比 A_5 有效只是在 $N^3<2^N$，即 $N\geq c$ 时才成立。其中 c 是方程 $N^3=2^N$ 的解。当 $N<c$ 时，A_5 反而比 A_4 有效。所以对于规模小的问题，不要盲目地选用复杂性阶比较低的算法。其原因一方面是如上所说，复杂性阶比较低的算法在规模小时不一定比复杂性阶比较高的算法更有效；另方面，在规模小时，决定工作效率的可能不是算法的效率而是算法的简单性，哪一种算法简单，实现起来快，就选用那一种算法。
2. 当要比较的两个算法的渐近复杂性的阶相同时，必须进一步考察渐近复杂性表达式中常数因子才能判别它们谁好谁差。显然常数因子小的优于常数因子大的算法。比如渐近复杂性为 $N1\log N/l00$ 的算法显然比渐近复杂性为 $l00N\log N$ 的算法来得有效。

算法复杂性渐近阶的分析

前两段讲的是算法复杂性渐近阶的概念和对它进行分析的重要性。本段要讲如何具体地分析一个算法的复杂性的渐近阶，给出一套可操作的规则。算法最终要落实到用某种程序设计语言（如 Pascal）编写成的程序。因此算法复杂性渐近阶的分析可代之以对表达该算法的程序的复杂性渐近阶的分析。

如前所提出，对于算法的复杂性，我们只考虑最坏、最好和平均三种情况，而通常又着重于最坏情况。为了明确起见，本段限于针对最坏情况。

仍然以时间复杂性为例。这里给出分析时间复杂性渐近阶的八条规则。这八条规则已覆盖了用 Pascal 语言程序所能表达的各种算法在最坏情况下的时间复杂性渐近阶的分析。

在逐条地列出并解释这入条规则之前，应该指出，当我们分析程序的某一局部(如一个语句，一个分程序，一个程序段，一个过程或函数)时，可以用具体程序的输入的规模 N 作为复杂性函数的自变量，也可以用局部的规模参数作为自变量。但是，作为最终结果的整体程序的复杂性函数只能以整体程序的输入规模为自变量。

对于串行的算法，相应的 Pascal 程序是一个串行的 Pascal 语句序列，因此，很明显，该算法的时间复杂性(即所需要的时间)等于相应的 Pascal 程序的每一个语句的时间复杂性(即所需要的时间)之和。所以，如果执行 Pascal 语句中的每一种语句所需要的时间都有计量的规则，那么，执行一个程序，即执行一个算法所需要的时间的计量便只是一个代数问题。接着，应用本节第三段所提供的 O、Ω 和 θ 等运算规则就可以分析出算法时间复杂性的渐近阶。

因此，我们的时间计量规则只需要针对 Pascal 有限的几种基本运算和几种基本语句。下面是这些规则的罗列和必要的说明。

规则(1)

赋值、比较、算术运算、逻辑运算、读写单个常量或单个变量等，只需要 1 个单位时间。

规则(2)

条件语句"if C then S_1 else S_2"只需要 $T_c+\max(T_{s1}, T_{s2})$的时间，其中 T_c 是计算条件表达式 C 需要的时间，而 T_{s1} 和 T_{s2} 分别是执行语句 S_1 和 S_2 需要的时间。

规则(3)

选择语句"Case A of a1:S1; a2:S2; … ;am:Sm; end"，需要 max(Ts1, Ts2，…，Tsm)的时间，其中 Tsii 是执行语句 Si 所需要的时间，i=l,2,…,m。

规则(4)

访问一个数组的单个分量或一个记录的单个域，只需要 1 个单位时间。

规则(5)

执行一个 for 循环语句需要的时间等于执行该循环体所需要的时间乘上循环的次数。

规则(6)

执行一个 while 循环语句"while C do S"或一个 repeat 循环语句" repeat S until C"，需要的时间等于计算条件表达式 C 需要的时间与执行循环 S 体需要的时间之和乘以循环的次数。与规则 5 不同，这里的循环次数是隐含的。

例如，b_search 函数中的 while 循环语句。按规则(1)-(4)，计算条件表达式" (not found)and(U≥=L)"与执行循环体

```
I:=(U+L)div 2;
if c=A[I] then found:=true
          else  if  c>A[I] then L:=I+1
                           else U:=I-1;
```

只需要 $\theta(1)$时间，而循环次数为 log$m$，所以，执行此 while 语句只需要 $\theta(\log m)$时间。

在许多情况下，运用规则(5)和(6)常常须要借助具体算法的内涵来确定循环的次数，才不致使时间的估计过于保守。这里举一个例子。

考察程序段：

```
Size:=m;                                                    1
                          i:=1;                             1
                    while i<n do
                          begin
                          i:=i+1;
                          S1;                               θ(n)
                    if  Size>0  then                        1
                          begin
              在 1 到 Size 的范围内任选一个数赋值给 t;      θ(1)
                          Size:=Size-t;                     2
                    for j:=l  to  t  do
                          S2                                θ(n)
                          end;
                          end;
```

程序在各行右端顶格处标注着执行相应各行所需要的时间。如果不对算法的内涵作较深入的考察，只看到 1≤t≤Size≤m，就草率地估计 while 的内循环 for 的循环次数为 $O(m)$，那么，程序在最坏情况下的时间复杂性将被估计为 $O(n^2+m\cdot n^2)$。反之，如果对算法的内涵认真地分析，结果将两样。事实上，在 while 的循环体内 t 是动态的,size 也是动态的，它们都取决 while 的循环参数 i，即 $t=t$(i)记为 t_i；size=size(i)记为 $size_i$，i=l,2,…,n-1。对于各个 i，1≤i≤n-1，t_i与 m 的关系是隐含的，这给准确地计算 for 循环的循环体 S_2 被执行的次数带来困难。上面的估计比较保守的原因在于我们把 S_2 的执行次数的统计过于局部化。如果不

局限于 for 循环，而是在整个程序段上统计 S_2 被执行的总次数，那么，这个总次数等于　　，

又根据算法中 t_i 的取法及 $size_{i+1}=size_i-t_i$，i=1,2,…,n-1 有 $size_n=size_1-$　　。最后利用

$size_1=m$ 和 $size_n=0$ 得到　　=m 。于是在整个程序段上，S_2 被执行的总次数为 m，所需要的时间为 $\theta(mn)$。执行其他语句所需要的时间直接运用规则(l)-(6)容易计算。累加起来，整个程序段在最坏情况下时间复杂性渐近阶为 $\theta(n^2+mn)$。这个结果显然比前面粗糙的估计准确。

规则(7)

对于 goto 语句。在 Pascal 中为了便于表达从循环体的中途跳转到循环体的结束或跳转到循环语句的后面语句，引入 goto 语句。如果我们的程序按照这一初衷使用 goto 语句，那么，在时间复杂性分析时可以假设它不需要任何额外的时间。因为这样做既不会低估也不会高估程序在最坏情况下的运行时间的阶。如果有的程序滥用了 goto 语句，即控制转移到前面的语句，那么情况将变得复杂起来。当这种转移造成某种循环时，只要与别的循环不交叉，保持循环的内外嵌套，则可以比照规则(1)-(6)进行分析。当由于使用 goto 语句而使程序结构混乱时，建议改写程序然后再做分析。

规则(8)

对于过程调用和函数调用语句，它们需要的时间包括两部分，一部分用于实现控制转移，另一部分用于执行过程(或函数)本身，这时可以根据过程(或函数)调用的层次，由里向外运用规则(l)-(7)进行分析，一层一层地剥，直到计算出最外层的运行时间便是所求。如果过程(或函数)出现直接或间接的递归调用，则上述由里向外逐层剥的分析行不通。这时我们可以对其中的各个递归过程(或函数)，所需要的时间假设为一个相应规模的待定函数。然后一一根据过程(或函数)的内涵建立起这些待定函数之间的递归关系得到递归方程。最后用求递归方程解的渐进阶的方法确定最坏情况下的复杂性的渐进阶。

递归方程的种类很多，求它们的解的渐近阶的方法也很多，我们将在下一段比较系统地给予介绍。本段只举一个简单递归过程(或函数)的例子来说明如何建立相应的递归方程，同时不加推导地给出它们在最坏情况下的时间复杂性的渐近阶。

例：再次考察函数 b_search，这里将它改写成一个递归函数。为了简明，我们已经运用前面的规则(l)-(6)，统计出执行各行语句所需要的时间，并标注在相应行的右端：

```
Function b_search(C,L,U:integer):integer;                    单位时间数
var index,element:integer;
 begin
   if (U<L) then                                                  1
         b_search:=0;                                             1
   else
    begin
     index:=(L+U) div 2;                                          3
     element:=A[index];                                           2
     if element=C then                                            1
       b_search:=index                                            1
   else if element>C then
       b_search:=b_search(C,L,index-1)                          3+T(m/2)
     else
       b_search:=b_search(C,index+1,U);                         3+T(m/2)
    end;
end;
```

其中 $T(m)$是当问题的规模 $U-L+1=m$ 时 b_search 在最坏情况下(这时，数组 A[L..U]中没有给定的 C)的时间复杂性。根据规则(l)-(8)，我们有：

或化简为

这是一个关于 T(m)的递归方程。用下一段将介绍的迭代法，容易解得：

$T(m)=11\log m+l3=\theta(\log m)$

在结束这一段之前，我们要提一下关于算法在最坏情况下的空间复杂性分析。我们照样可以给出与分析时间复杂性类似的规则。这里不赘述。然而应该指出，在出现过程(或函数)递归调用时要考虑到其中隐含的存储空间的额外开销。因为现有的实现过程(或函数)递归调用的编程技术需要一个隐含的、额外(即不出现在程序的说明中)的栈来支持。过程(或函数)的递归调用每深人一层就把本层的现场局部信息及调用的返回地址存放在栈顶备用，直到调用的最里层。因此递归调用一个过程(或函数)所需要的额外存储空间的大小即栈的规模与递归调用的深度成正比，其比例因子等于每深入一层需要保存的数据量。比如本段前面所举的递归函数 b_search，在最坏情况下，递归调用的深度为 $\log m$，因而在最坏情况下调用它所需要的额外存储空间为 $\theta(\log m)$。

递归方程解的渐近阶的求法

上一章所介绍的递归算法在最坏情况下的时间复杂性渐近阶的分析，都转化为求相应的一个递归方程的解的渐近阶。因此，求递归方程的解的渐近阶是对递归算法进行分析的关键步骤。

递归方程的形式多种多样，求其解的渐近阶的方法也多种多样。这里只介绍比较实用的五种方法。

1. **代入法** 这个方法的基本步骤是先推测递归方程的显式解，然后用数学归纳法证明这一推测的正确性。那么，显式解的渐近阶即为所求。
2. **迭代法** 这个方法的基本步骤是通过反复迭代，将递归方程的右端变换成一个级数，然后求级数的和，再估计和的渐近阶；或者，不求级数的和而直接估计级数的渐近阶，从而达到对递归方程解的渐近阶的估计。
3. **套用公式法** 这个方法针对形如：$T(n)=aT(n/b)+f(n)$ 的递归方程，给出三种情况下方程解的渐近阶的三个相应估计公式供套用。
4. **差分方程法** 有些递归方程可以看成一个差分方程，因而可以用解差分方程(初值问题)的方法来解递归方程。然后对得到的解作渐近阶的估计。
5. **母函数法** 这是一个有广泛适用性的方法。它不仅可以用来求解线性常系数高阶齐次和非齐次的递归方程，而且可以用来求解线性变系数高阶齐次和非齐次的递归方程，甚至可以用来求解非线性递归方程。方法的基本思想是设定递归方程解的母函数，努力建立一个关于母函数的可解方程，将其解出，然后返回递归方程的解。

本章将逐一地介绍上述五种井法，并分别举例加以说明。

本来，递归方程都带有初始条件，为了简明起见，我们在下面的讨论中略去这些初始条件。

递归方程组解的渐进阶的求法——代入法

用这个办法既可估计上界也可估计下界。如前面所指出，方法的关键步骤在于预先对解答作出推测，然后用数学归纳法证明推测的正确性。

例如，我们要估计 T(n)的上界，T(n)满足递归方程：

其中　是地板(floors)函数的记号，　表示不大于 n 的最大整数。

我们推测 T(n)=O(n*log* n)，即推测存在正的常数 C 和自然数 n_0，使得当 $n \geq n_0$ 时有：

$T(n) \leq Cn \log n$ (6.2)

事实上，取 $n_0=2^2=4$，并取

那么，当 $n_0 \leq n \leq 2n_0$ 时，(6.2)成立。今归纳假设当 $2^{k-1}n_0 \leq n \leq 2^k n_0$ ，$k \geq 1$ 时，(1.1.16)成立。那么，当 $2^k n_0 \leq n \leq 2^{k+1} n_0$ 时，我们有：

即(6.2)仍然成立，于是对所有 $n \geq n_0$，(6.2)成立。可见我们的推测是正确的。因而得出结论：递归方程(6.1)的解的渐近阶为 $O(n \log n)$。

这个方法的局限性在于它只适合容易推测出答案的递归方程或善于进行推测的高手。推测递归方程的正确解，没有一般的方法，得靠经验的积累和洞察力。我们在这里提三点建议：

(1) 如果一个递归方程类似于你从前见过的已知其解的方程，那么推测它有类似的解是合理的。作为例子，考虑递归方程：

右边项的变元中加了一个数 17，使得方程看起来难于推测。但是它在形式上与(6.1)很类似。实际上，当 n 充分大时

与

相差无几。因此可以推测(6.3)与(6.1)有类似的上界 $T(n)=O(n\log n)$。进一步，数学归纳将证明此推测是正确的。

(2)从较宽松的界开始推测，逐步逼近精确界。比如对于递归方程(6.1)，要估计其解的渐近下界。由于明显地有 $T(n)\geq n$，我们可以从推测 $T(n)=\Omega(n)$开始，发现太松后，把推测的阶往上提，就可以得到 $T(n)=\Omega(n\log n)$的精确估计。

(3)作变元的替换有时会使一个未知其解的递归方程变成类似于你曾见过的已知其解的方程，从而使得只要将变换后的方程的正确解的变元作逆变换，便可得到所需要的解。例如考虑递归方程：

看起来很复杂，因为右端变元中带根号。但是，如果作变元替换 $m=\log n$，即令 $n=2^m$ ，将其代入(6.4)，则(6.4)变成：

把 m 限制在正偶数集上，则(6.5)又可改写为：

$$T(2^m)=2T(2^{m/2})+m$$

若令 $S(m)=T(2^m)$，则 $S(m)$满足的递归方程：

$$S(m)=2S(m/2)+m \ ,$$

与(6.1)类似，因而有：

$$S(m)=O(m\log m),$$

进而得到 $T(n)=T(2^m)=S(m)=O(m\log m)=O(\log n\log\log n)$ (6.6)

上面的论证只能表明：当(充分大的)n 是 2 的正偶次幂或换句话说是 4 的正整数次幂时(6.6)才成立。进一步的分析表明(6.6)对所有充分大的正整数 n 都成立，从而，递归方程(6.4)解的渐近阶得到估计。

在使用代入法时，有三点要提醒：

(1)记号 O 不能滥用。比如，在估计(6.1)解的上界时，有人可能会推测 $T(n)=O(n)$，即对于充分大的 n，有 $T(n)\leq Cn$ ，其中 C 是确定的正的常数。他进一步运用数学归纳法，推出：

从而认为推测 $T(n)=O(n)$是正确的。实际上，这个推测是错误的，原因是他滥用了记号 O ，错误地把$(C+1)n$ 与 Cn 等同起来。

(2)当对递归方程解的渐近阶的推测无可非议，但用数学归纳法去论证又通不过时，不妨在原有推测的基础上减去一个低阶项再试试。作为一个例子，考虑递归方程

其中　　是天花板(floors)函数的记号。我们推测解的渐近上界为 $O(n)$。我们要设法证明对于适当选择的正常数 C 和自然数 n_0，当 $n\geq n_0$ 时有 $T(n)\leq Cn$。把我们的推测代入递归方程，得到：

我们不能由此推断 $T(n)\leq Cn$，归纳法碰到障碍。原因在于(6.8)的右端比 Cn 多出一个低阶常量。为了抵消这一低阶量，我们可在原推测中减去一个待定的低阶量 b，即修改原来的推测为 $T(n)\leq Cn-b$ 。现在将它代人(6.7)，得到：

只要 $b\geq 1$，新的推测在归纳法中将得到通过。

(3)因为我们要估计的是递归方程解的渐近阶，所以不必要求所作的推测对递归方程的初始条件（如 $T(0)$、$T(1)$）成立，而只要对 $T(n)$成立，其中 n 充分大。比如，我们推测(6.1)的解 $T(n)\leq Cn\log n$，而且已被证明是正确的，但是当 $n=1$ 时，这个推测却不成立，因为 $(Cn\log n)|_{n=1}=0$ 而 $T(1)>0$。

递归方程组解的渐进阶的求法——迭代法

用这个方法估计递归方程解的渐近阶不要求推测解的渐近表达式，但要求较多的代数运算。方法的思想是迭代地展开递归方程的右端，使之成为一个非递归的和式，然后通过对和式的估计来达到对方程左端即方程的解的估计。

作为一个例子，考虑递归方程：

接连迭代二次可将右端项展开为：

由于对地板函数有恒等式：

(6.10)式可化简为：

这仍然是一个递归方程，右端项还应该继续展开。容易看出，迭代 i 次后，将有

（6.11）

而且当

时，(6.11)不再是递归方程。这时：

(6.13)

又因为$[a]\leq a$，由(6.13)可得：

而由(6.12)，知 $i\leq \log_4 n$ ，从而

，

代入(6.14)得：

即方程(6.9)的解 $T(n)=O(n)$。

从这个例子可见迭代法导致繁杂的代数运算。但认真观察一下，要点在于确定达到初始条件的迭代次数和抓住每次迭代产生出来的"自由项"(与 T 无关的项)遵循的规律。顺便指出，迭代法的前几步迭代的结果常常能启发我们给出递归方程解的渐近阶的正确推测。这时若换用代入法，将可免去上述繁杂的代数运算。

图 6-1 与方程(6.15)相应的递归树

为了使迭代法的步骤直观简明、图表化，我们引入**递归树**。靠着递归树，人们可以很快地得到递归方程解的渐近阶。它对描述分治算法的递归方程特别有效。我们以递归方程

$$T(n)=2T(n/2)+n^2 \quad (6.15)$$

为例加以说明。图 6-1 展示出(6.15)在迭代过程中递归树的演变。为了方便，我们假设 n 恰好是 2 的幂。在这里，递归树是一棵二叉树，因为(6.15)右端的递归项 $2T(n/2)$可看成 $T(n/2)+T(n/2)$。图 6-1(a)表示 $T(n)$集中在递归树的根处，(b)表示 $T(n)$已按(6.15)展开。也就是将组成它的自由项 n^2 留在原处，而将 2 个递归项 $T(n/2)$分别摊给它的 2 个儿子结点。(c)表示迭代被执行一次。图 6-1(d)展示出迭代的最终结果。

图 6-1 中的每一棵递归树的所有结点的值之和都等于 $T(n)$。特别，已不含递归项的递归树(d)中所有结点的值之和亦然。我们的目的是估计这个和 $T(n)$。我们看到有一个表格化的办法：先按横向求出每层结点的值之和，并记录在各相应层右端顶格处，然后从根到叶逐层地将顶格处的结果加起来便是我们要求的结果。照此，我们得到(6.15)解的渐近阶为 $\theta(n^2)$。

再举一个例子。递归方程：

$$T(n)= T(n/3)+ T(2n/3)+n \quad (6.16)$$

的迭代过程相应的递归树如图 6-2 所示。其中，为了简明，再一次略去地板函数和天花板函数。

图 6-2 迭代法解(6.16)的递归树

当我们累计递归树各层的值时，得到每一层的和都等于 n，从根到叶的最长路径是

设最长路径的长度为 k，则应该有

，

得

，

于是

即 $T(n)=O(n\log n)$ 。

以上两个例子表明，借助于递归树，迭代法变得十分简单易行。

递归方程组解的渐进阶的求法——套用公式法

这个方法为估计形如：

$T(n)=aT(n/b)+f(n)$ (6.17)

的递归方程解的渐近阶提供三个可套用的公式。(6.17)中的 $a\geq 1$ 和 $b\geq 1$ 是常数，$f(n)$是一个确定的正函数。

(6.17)是一类分治法的时间复杂性所满足的递归关系，即一个规模为 n 的问题被分成规模均为 n/b 的 a 个子问题，递归地求解这 a 个子问题，然后通过对这 a 个子问题的解的综合，得到原问题的解。如果用 $T(n)$表示规模为 n 的原问题的复杂性，用 $f(n)$表示把原问题分成 a 个子问题和将 a 个子问题的解综合为原问题的解所需要的时间，我们便有方程(6.17)。

这个方法依据的是如下的定理：设 $a\geq 1$ 和 $b\geq 1$ 是常数 $f(n)$是定义在非负整数上的一个确定的非负函数。又设 $T(n)$也是定义在非负整数上的一个非负函数，且满足递归方程(6.17)。方程(6.17)中的 n/b 可以是$[n/b]$，也可以是 n/b。那么，在 $f(n)$的三类情况下，我们有 $T(n)$的渐近估计式：

1. 若对于某常数 $\varepsilon>0$，有

   ，

   则

   ；

2. 若

   ，

   则

   ；

3. 若对其常数 $\varepsilon>0$，有

   且对于某常数 $c>1$ 和所有充分大的正整数 n 有 $af(n/b)\leq cf(n)$，则 $T(n)=\theta(f(n))$。

这里省略定理的证明。

在应用这个定理到一些实例之前，让我们先指出定理的直观含义，以帮助读者理解这个定理。读者可能已经注意到，这里涉及的三类情况，都是拿 $f(n)$与　　作比较。定理直观地告诉我们，递归方程解的渐近阶由这两个函数中的较大者决定。在第一类情况下，函数　　较大，则 $T(n)=\theta($　　$)$；在第三类情况下，函数 $f(n)$较大，则 $T(n)=\theta(f(n))$；在第二类情况下，两个函数一样大，则 $T(n)=\theta($　　$)$，即以 n 的对数作为因子乘上 $f(n)$与 $T(n)$的同阶。

此外，定理中的一些细节不能忽视。在第一类情况下 $f(n)$不仅必须比　　小，而且必须是多项式地比　　小，即 $f(n)$必须渐近地小于　　与　　的积，ε 是一个正的常数；在第三类情况下 $f(n)$不仅必须比　　大，而且必须是多项式地比　　大，还要满足附加的“正规性”条件：$af(n/b)\leq cf(n)$。这个附加的“正规性”条件的直观含义是 a 个子问题的再分解和再综合所需要的时间最多与原问题的分解和综合所需要的时间同阶。我们在一般情况下将碰到的以多项式为界的函数基本上都满足这个正规性条件。

还有一点很重要，即要认识到上述三类情况并没有覆盖所有可能的 $f(n)$。在第一类情况和第二类情况之间有一个间隙：$f(n)$小于但不是多项式地小于　　；类似地，在第二类情

况和第三类情况之间也有一个间隙：$f(n)$大于但不是多项式地大于　　。如果函数 $f(n)$落在这两个间隙之一中，或者虽有　　，但正规性条件不满足，那么，本定理无能为力。

下面是几个应用例子。

例 1 考虑

$$T(n)=9T(n/3)+n_0$$

对照(6.17)，我们有 $a=9$，$b=3$， $f(n)=n$，　　，取　　，便有

　　，可套用第一类情况的公式，得 $T(n)=\theta(n^2)$。

例 2 考虑

$$T(n)=T(2n/3)+1$$

对照(6.17)，我们有 a=1，b=3/2， f(n)=1，　　，可套用第二类情况的公式，得 T(n)=θ(logn)。

例 3 考虑

$$T(n)=3T(n/4)+n\log n$$

对照(6.17)，我们有 a=3，b=4，f(n)=nlog n，　　，只要取　　，便有　　。进一步，检查正规性条件：

只要取 c=3/4，便有 af(n/b)≤cf(n)，即正规性条件也满足。可套用第三类情况的公式，得 T(n)=θ(f(n))=θ(nlogn)。

最后举一个本方法对之无能为力的例子。

考虑

$$T(n)=2T(n/2)+n\log n$$

对照(6.17)，我们有 $a=2$，$b=2$， $f(n)=n\log n$，　　，虽然 $f(n)$渐近地大于　　，但 $f(n)$并不是多项式地大于　　，因为对于任意的正常数 ε，

，

即 $f(n)$在第二类情况与第三类情况的间隙里，本方法对它无能为力。

递归方程组解的渐进阶的求法——差分方程法

这里只考虑形如：

$$T(n)=c_1T(n\text{-}1)+c_2T(n\text{-}2)+\ldots+ c_kT(n\text{-}k)+f(n)，\ n\geq k \quad (6.18)$$

的递归方程。其中 c_i (i=I，2，…，k)为实常数，且 $c_k\neq 0$。它可改写为一个线性常系数 k 阶非齐次的差分方程：

$$T(n)-c_1T(n\text{-}1)- c_2T(n\text{-}2)-\ldots-c_kT(n\text{-}k)=f(n)，\ n\geq k \quad (6.19)$$

(6.19)与线性常系数k阶非齐次常微分方程的结构十分相似，因而解法类同。限于篇幅，这里直接给出(6.19)的解法，略去其正确性的证明。

第一步，求(6.19)所对应的齐次方程：

$$T(n)-c_1T(n\text{-}1)- c_2T(n\text{-}2)-\ldots-c_kT(n\text{-}k)=0 \quad (6.20)$$

的基本解系：写出(6.20)的特征方程：

$$C(t)=t^k-c_1t^{k-1}-c_2t^{k-2} -\ldots-c_k=0 \quad (6.21)$$

若 $t=r$ 是(6.21)的 m 重实根，则得(6.20)的 m 个基础解 r^n，nr^n，n^2r^n，…，$n^{m-1}r^n$；若 $\rho e^{i\theta}$ 和 $\rho e^{-i\theta}$是(6.21)的一对l重的共扼复根，则得(6.20)的$2l$个基础解$\rho^n\cos n\theta$，$\rho^n\sin n\theta$，$n\rho^n\cos n\theta$，$n\rho^n\sin n\theta$，…，$n^{l-1}\rho^n\cos n\theta$，$n^{l-1}\rho^n\cos n\theta$。如此，求出(6.21)的所有的根，就可以得到(6.20)的 k 个的基础解。而且，这 k 个基础解构成了(6.20)的基础解系。即(6.20)的任意一个解都可以表示成这 k 个基础解的线性组合。

第二步，求(6.19)的一个特解。理论上，(6.19)的特解可以用 Lagrange 常数变易法得到。但其中要用到(6.20)的通解的显式表达，即(6.20)的基础解系的线性组合，十分麻烦。因此在实际中，常常采用试探法，也就是根据 $f(n)$的特点推测特解的形式，留下若干可调的常数，将推测解代人(6.19)后确定。由于(6.19)的特殊性，可以利用迭加原理，将 $f(n)$线性分解为若干个单项之和并求出各单项相应的特解，然后迭加便得到 $f(n)$相应的特解。这使得试探法更为有效。为了方便，这里对三种特殊形式的 $f(n)$，给出(6.19)的相应特解并列在表 6-1 中，可供直接套用。其中 p_i，i=1,2,…,s 是待定常数。

表 6-1 方程(6.19)的常用特解形式

$f(n)$的形式	**条　件**	**方程(6.19)的特解的形式**
a^n	$C(a)\neq 0$	

	a 是 $C(t)$的 m 重根	
n^s	$C(1)\neq 0$	
	1 是 $C(t)$的 m 重根	
$n^s a^n$	$C(a)\neq 0$	
	a 是 $C(t)$的 m 重根	

第三步，写出(6.19)即(6.18)的通解

(6.22)

其中$\{T_i(n)$，i=0,1,2,…,$n\}$是(6.20)的基础解系，$g(n)$是(6.19)的一个特解。然后由(6.18)的初始条件

$T(i)=T_i$ ，i=1,2,…,k-1

来确定(6.22)中的待定的组合常数$\{a_i\}$，即依靠线性方程组

或

解出$\{a_i\}$，并代回(6.22)。其中 $\beta_j=T_j-g(j)$，j=0,1,2,…,k-1。

第四步，估计(6.22)的渐近阶，即为所要求。

下面用两个例子加以说明。

例 l 考虑递归方程

它的相应特征方程为：

$$C(t)=t^2-t-1=0$$

解之得两个单根　　和　　。相应的(6.20)的基础解系为$\{r_0^n, r_1^n\}$。
相应的(6.19)的一个特解为 $F^*(n)=-8$，因而相应的(6.19)的通解为：

$$F(n)=a_0r_0^n+a_1r_1^n-8$$

令其满足初始条件，得二阶线性方程组：

或

或

解之得　　，　　，从而

于是

。

例 2 考虑递归方程

$$T(n)=4T(n-1)-4T(n-2)+2^n n \quad (6.23)$$

和初始条件 $T(0)=0$，$T(1)=4/3$。

它对应的特征方程(6.21)为

$$C(t)=t^2-4t+4=0$$

有一个两重根 $r=2$。故相应的(6.20)的基础解系为$\{2^n, 2^n n\}$。由于 $f(n)=2^n n$，利用表 6-1，
相应的(6.19)的一个特解为

$T^*(n)=n^2(p_0+p_1n)2^n$，

代人(6.23)，定出 $p_0=1/2$，$p_1=1/6$。因此相应的(6.19)的通解为：

$T(n)=a_02^n+a_1n2^n+n^2(1/2+n/6)2^n$，

令其满足初始条件得 $a_0=a_1=0$，从而

$T(n)=n^2(1/2+n/6)2^n$

于是 $T(n)=\theta(n^32^n)$。

递归方程组解的渐进阶的求法——母函数法

关于 $T(n)$的递归方程的解的母函数通常设为：

(6.24)

当(6.24)右端由于 $T(n)$增长太快而仅在 $x=0$ 处收敛时可另设

(6.25)

如果我们可以利用递归方程建立 $A(x)$的一个定解方程并将其解出，那么，把 $A(x)$展开成幂级数，则 x^n 或 $x^n/n!$项的系数便是所求的递归方程的解。其渐近阶可接着进行估计。

下面举两个例子加以说明。

例 1 考虑线性变系数二阶齐次递归方程

$(n-1)T(n)=(n-2)T(n-1)+2T(n-2)$ ，$n\geq 2$ (6.26)

和初始条件 $T(0)=0$，$T(1)=1$。根据初始条件及(6.26)，可计算 $T(2)=0$，$T(3)=T(1)=1$。

设$\{T(n)\}$的母函数为：

由于 $T(0)=T(2)=0$，$T(1)=1$，有 ：

令 $B(x)= A (x)/x$，即：

那么：

利用(6.26)并代入 T (3)= 1，得

即

两边同时沿[0,x]积分，并注意到 $B(0)=1$，有：

把 $B(x)$展开成幂级数，得

从而

最后得

例 2 考虑线性变系数一阶非齐次递归方程

$$D(n)=nD(n\text{-}1)+(-1)^n \quad n\geq 1 \quad (6.27)$$

及初始条件 $D(0)=1$

很明显 $D(n)$随 n 的增大而急剧增长。如果仍采用(6.24)形式的函数，则(6.24)的右端可能仅在 $x=0$ 处收敛，所以这里的母函数设为：

用 $x^n/n!$乘以(6.27)的两端，然后从 1 到∞求和得：

化简并用母函数表达，有：

$$A(x)-1=xA(x)+e^{-x}-1$$

或

$$(1-x)A(x)=e^{-x}$$

从而

$$A(x)=e^{-x}/(1-x)$$

展成幂级数，则：

故

算法设计策略

这里介绍了一般的算法设计策略，阐述各方法的理论基础、主要思想及其适用范围。同时针对一些具体问题来讲述如何用这些一般的理论以及各种抽象数据类型对问题进行抽象描述，并用最有效的方式设计出解决问题的高效算法。它们将生动地再现计算机程序设计方法学的理论、抽象和设计三个过程，而且，通过对算法正确性的证明和复杂性的分析，深化对大问题的复杂性、概念和形式模型、效率和抽象的层次、折衷和结论等在计算机学科中重复出现的概念的理解。

必须强调指出，对于某些问题(如 NP--完全问题)而言，用这里的方法和任何已知的方法都不可能设计出有效的算法。对于这种问题，人们常常考虑利用具体输入的某些特点来设计有效算法或设计求问题近似解的有效算法。这一部分内容我们将在高级专题中讨论。

在对有关算法进行形式描述时我们采用类 Pascal 的伪代码，并作了一些简化，略去不言而喻的一些说明，如函数、形参、变量等类型说明。

这里主要讨论的算法设计策略有：

- 递归技术 —— 最常用的算法设计思想，体现于许多优秀算法之中
- 分治法 —— 分而制之的算法思想，体现了一分为二的哲学思想
- 模拟法 —— 用计算机模拟实际场景，经常用于与概率有关的问题
- 贪心算法 —— 采用贪心策略的算法设计
- 状态空间搜索法 —— 被称为“万能算法”的算法设计策略
- 随机算法 —— 利用随机选择自适应地决定优先搜索的方向
- 动态规划 —— 常用的最优化问题解决方法

摘要

本文介绍了分治法的基本思想和基本步骤，通过实例讨论了利用分治策略设计算法的途径。

目录

- 其他资料

参考文献

- *现代计算机常用数据结构和算法*，潘金贵 等 编著，南京大学出版社，1992
- *算法与数据结构*，傅清祥 王晓东 编著，电子工业出版社，1998
- *Dictionary of Algorithms, Data Structures, and Problems* ， Paul E. Black ， http://hissa.nist.gov/dads/ ，下载该网站的镜像（1,682KB）

简介

对于一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做**分治法**。

分治法的基本思想

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于 n 个元素的排序问题，当 n=1 时，不需任何计算。n=2 时，只要作一次比较即可排好序。n=3 时只要作 3 次比较即可，…。而当 n 较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。

分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

如果原问题可分割成 k 个子问题，$1<k\leq n$ ，且这些子问题都可解，并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

1. 该问题的规模缩小到一定的程度就可以容易地解决；
2. 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。
3. 利用该问题分解出的子问题的解可以合并为该问题的解；
4. 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

上述的第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；第二条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用；第三条特征是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑贪心法或动态规划法。第四条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好。

分治法的基本步骤

分治法在每一层递归上都有三个步骤：

1. **分解**：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；
2. **解决**：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；
3. **合并**：将各个子问题的解合并为原问题的解。

它的一般的算法设计模式如下：

```
Divide-and-Conquer(P)
1.  if |P|≤n0
2.    then return(ADHOC(P))
3.  将 P 分解为较小的子问题 P1 ,P2 ,...,Pk
4.  for i←1 to k
5.    do yi ← Divide-and-Conquer(Pi)      △ 递归解决 Pi
6.  T ← MERGE(y1,y2,...,yk)               △ 合并子问题
7.  return(T)
```

其中|P|表示问题 P 的规模；n_0 为一阈值，表示当问题 P 的规模不超过 n_0 时，问题已容易直接解出，不必再继续分解。ADHOC(P)是该分治法中的基本子算法，用于直接解小规模的问题 P。因此，当 P 的规模不超过 n_0 时，直接用算法 ADHOC(P)求解。算法 MERGE(y_1,y_2,...,y_k)是该分治法中的合并子算法，用于将 P 的子问题 P_1 ,P_2 ,...,P_k 的相应的解 y_1,y_2,...,y_k 合并为 P 的解。

根据分治法的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才为适当？这些问题很难予以肯定的回答。但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。许多问题可以取 k=2。这种使子问题规模大致相等的做法是出自一种**平衡(balancing)子问题**的思想，它几乎总是比子问题规模不等的做法要好。

分治法的合并步骤是算法的关键所在。有些问题的合并方法比较明显，如下面的例 1，例 2；有些问题合并方法比较复杂，或者是有多种合并方案，如例 3，例 4；或者是合并方案不明显，如例 5。究竟应该怎样合并，没有统一的模式，需要具体问题具体分析。

分治法的复杂性分析

从分治法的一般设计模式可以看出，用它设计出的程序一般是一个递归过程。因此，分治法的计算效率通常可以用递归方程来进行分析。为方便起见，设分解阈值 n_0=1，且算法 ADHOC 解规模为 1 的问题耗费 1 个单位时间。又设分治法将规模为 n 的问题分成 k 个规模为 n/m 的子问题去解，而且，将原问题分解为 k 个子问题以及用算法 MERGE 将 k 个子问题的解合并为原问题的解需用 f(n)个单位时间。如果用 T(n)表示该分治法 Divide-and-Conquer(P)解规模为|P|=n 的问题 P 所需的计算时间，则有:

（1）

用算法的复杂性中递归方程解的渐进阶的解法介绍的解递归方程的迭代法，可以求得(1)的解:

（2）

注意，递归方程(1)及其解(2)只给出 n 等于 m 的方幂时 T(n)的值，但是如果认为 T(n)足够平滑，那么由等于 m 的方幂时 T(n)的值可以估计 T(n)的增长速度。通常，我们可以假定 T(n)是单调上升的，从而当 $m^i \le n < m^{i+1}$ 时，$T(m^i) \le T(n) < T(m^{i+1})$。

另一个需要注意的问题是，在分析分治法的计算效率时，通常得到的是递归不等式：

(3)

由于我们关心的一般是最坏情况下的计算时间复杂度的上界，所以用等于号(=)还是小于或等于号(≤)是没有本质区别的。

分治法的几种变形

1. **二分法 dichotomy**

   一种每次将原问题分解为两个子问题的分治法，是一分为二的哲学思想的应用。这种方法很常用，由此法产生了许多经典的算法和数据结构。

2. **分解并在解决之前合并法 divide and marriage before conquest**

   一种分治法的变形，其特点是将分解出的子问题在解决之前合并。

3. **管道传输分治法 pipelined divide and conquer**

   一种分治法的变形，它利用某种称为“管道”的数据结构在递归调用结束前将其中的某些结果返回。此方法经常用来减少算法的深度。

注: divide and marriage before conquest 和 pipelined divide and conquer 方法我并不太了解，只在某些参考文献上看过其名称。其原文定义如下：
divide and marriage before conquest:A variant of divide and conquer in which subproblems created in the "divide" step are merged before the "conquer" step.
pipelined divide and conquer:A divide and conquer paradigm in which partial results from recursive calls can be used before the calls complete. The technique is often useful for reducing the depth of an algorithm.
如果你有关于这两种算法的资料请告诉我(mailto:Starfish.h@china.com)。

分治法的实例分析

以上讨论的是分治法的基本思想和一般原则，下面我们用具体的例子来说明如何针对具体问题用分治法来设计有效解法。

例 1 和例 2 是分治法的经典范例，其分解和合并过程都比较简单明显；例 3 和例 4 的合并方法有多种选择，只有选择最好的合并方法才能够改进算法的复杂度；例 5 是一个计算几何学中的问题，它的合并步骤需要较高的技巧。例 6 则是 IOI'95 的试题 Wires and Switches 。

- 例 1 二分查找
- 例 2 快速排序
- 例 3 大整数乘法
- 例 4 Strassen 矩阵乘法
- 例 5 最接近点对问题
- 例 6 导线和开关

更多实例请参阅分治法问题集

其他资料

请参阅以下文章：

- 递归技术
- 贪心法
- 动态规划
- 分治法问题集

用递归中序遍历二叉树

```
#include<stdlib.h>
struct tree              //声明树的结构
{
    struct tree *left;
    int data;
    struct tree *right;
};
typedef struct tree treenode;
type treenode *b_tree;           //声明二叉树链表
//插入二叉树的节点
b_tree insert_node(b_tree root,int node)
{
    b_tree newnode;
    b_tree currentnode;
    b_tree parentnode;

    newnode=(b_tree)malloc(sizeof(treenode));    //建立新节点的内存空间
    newnode->data=node;
    newnode->right=NULL;
    newnode->left=NULL;
    if(root=NULL)
        return newnode;
    else
    {   currentnode=root;
        while(currentnode!=NULL)
        {   parentnode=currentnode;
            if( currentnode->data>node)
                currentnode=currentnode->left;
            else   currentnode=currentnode->right;
        }
        if(parentnode->data>node)
            parentnode->left=newnode;
        else  parentnode->right=newnode;
    }
    return root;
}
// 建立二叉树
b_tree create_btree(int *data,int len)
{
    b_tree root=NULL;
    int i;
    for(i=0;i<len;i++)
```

```
        root=insert_node(root,data[i]);
    return root;
}
//二叉树中序遍历
void inorder(b_tree point)
{
    if(point!=NULL)
    {
        inorder(point->left);
        printf("%d",point->data);
        inorder(point->right);
    }
}
//主程序
void main( )
{
    b_tree root=NULL;
    int i,index;
    int value;
    int nodelist[20];
    printf("\n pleaase input the elements of binary tree(exit for 0 ):\n");
    index=0;
    //读取数值存到数组中
    scanf("%d",&value);

    while(value!=0)
    {
        nodelist[index]=value];
        index=index+1;
        scanf("%d",&value);
    }
    //建立二叉树
    root=create_btree(nodelist,index);
    //中序遍历二叉树
    printf("\nThe inorder traversal result is :");
    inorder(root);
    printf("\n");
}
```

Bresenham 高效画线算法

画线的算法不少，但要作到高速、简单并不容易。斜率相乘法是最简单的方法之一，但计算每个点均要花费不少时间用于乘、除法运算；下面介绍的是 Bresenham's 高效画线算法，对每个点的坐标计算只要加、减法就能完成。

简化算法用伪 Pascal 语言描述如下：

```
procedure DrawLine(x1, y1, x2, y2: Integer);
var
x, y, DeltaX, DeltaY, HalfX, ErrorTerm, i: Integer;
begin
DeltaX := x2 - x1;
DeltaY := y2 - y1;
HalfX := (x2 - x1) shr 1;
ErrorTerm := 0;
x := x1;
y := y1;
for i:=0 to DeltaX do
begin
Plot(X, Y);
Inc(x);
ErrorTerm := ErrorTerm + DeltaY;
if ErrorTerm>HalfX then
begin
ErrorTerm := ErrorTerm - DeltaX;
Inc(y);
end;
end;
end;
```

为方便阅读，上述程序作了简化。实际程序应略作修正，以分别处理 DeltaX 与 DeltaY 比较大小，必要时交换起始、结束点等。

修正后的的伪 Pascal 算法如下：

```
procedure DrawLine(x1, y1, x2, y2: Integer);
var
x, y, DeltaX, DeltaY, HalfCount, ErrorTerm, i, Flag: Integer;
begin
DeltaX := x2 - x1;
DeltaY := y2 - y1;
if Abs(DeltaY)<Abs(DeltaX) then
begin
if DeltaX<0 then
begin
i := x1; x1 := x2; x2 := i;
```

```
i := y1; y1 := y2; y2 := i;
DeltaX := x2 - x1;
DeltaY := y2 - y1;
end;
if DeltaY<0 then Flag := -1
else Flag := 1;
DeltaY := Abs(DeltaY);
HalfCount := DeltaX shr 1;
ErrorTerm := 0;
x := x1;
y := y1;
for i:=0 to DeltaX do
begin
Plot(X, Y);
Inc(x);
ErrorTerm := ErrorTerm + DeltaY;
if ErrorTerm>HalfCount then
begin
ErrorTerm := ErrorTerm - DeltaX;
y := y + Flag;
end;
end;
end
else
begin
if DeltaY<0 then
begin
i := x1; x1 := x2; x2 := i;
i := y1; y1 := y2; y2 := i;
DeltaX := x2 - x1;
DeltaY := y2 - y1;
end;
if DeltaX<0 then Flag := -1
else Flag := 1;
DeltaX := Abs(DeltaX);
HalfCount := DeltaY shr 1;
ErrorTerm := 0;
x := x1;
y := y1;
for i:=0 to DeltaY do
begin
Plot(X, Y);
Inc(y);
ErrorTerm := ErrorTerm + DeltaX;
```

```
if ErrorTerm>HalfCount then
begin
ErrorTerm := ErrorTerm - DeltaY;
x := x + Flag;
end;
end;
end;
end;
```

C++的沉迷与爱恋

每年的 09/28 对於我都是一个特殊的日子 -- 不只是因为教师节。今年很特殊地没有普天同庆，那麼我就写篇文章自己庆祝一下好了。

我於今年七月发表了一本着作〈多型与虚拟〉和一本译作〈深度探索 C++ 物件模型〉，获得很大的回响。这些作品都不是针对 C++ 的完全初学者所写，但从初阶到高阶为数众多的 C++ guy，热情地表达了他们对这些主题的喜悦。

在许多来信中，我看到一些有趣的现象，也感受到一些值得整理下来的想法。所以，根据我个人的学习过往、我的教学经验、以及周遭朋友的心得交流，写下这篇文章，或可为後学者戒。

●〈多型与虚拟〉序言节录

首先让我节录〈多型与虚拟〉一书序言：

〈多型与虚拟〉序 节录（侯俊杰/松岗/1998/07）

一般而言，C++ 是一个难学易用的语言。

C++ 的难学，初始在於其重重的布幕，布幕之中编译器对我们的程式码做了太多的手脚，使我们惯於循序思考的工程脑袋一无所措。及长，又面临新的思维模式，使我们必须扭转惯常的思考习惯。

C++ 的易用则在於其巨大的弹性，能够以多型（polymorphism）、虚拟（virtual）、模板（template）、泛型（generalization）等种种型式，让既有的码去处理未知的、未来的资料型态。

当然，易用必须先能用。用不好或不能用的话，「写 C++ 程式」最後就成了只是「使用 C++ 编译器」，这是大家常拿来彼此调侃的笑话。

在「难学」的背景下，「易用」是使我们依然前仆後继的动力。愈来愈多的

大学资讯科系把 C++ 开在大一课程，这虽然说明 C++ 是多麼地重要，可也苦了资讯新兵们。

其实「难学」的最大症结在於，很难得有一本书，能够一针见血地指出多型与虚拟的重要性；在我们粗具语法基础之後，直接把我们导引到最核心最重要的思想，并且在建立这个思想的过程中，提供足够的必要基础。

●困难度之一

「C++ 是个难学易用的语言」，这句话相信很多人心有戚戚。C++ 的学习难度，一在於语言本身太多的「幕」，一在於 "paradigm shift"（思考模式的移转）。

传统循序语言如 C, Pascal, Basic, Fortran...，除了模样看起来稍有不同，基本上都是函式 call 来 call 去，大同小异，很容易掌握。你想做的动作，在 code 中都看得一清二楚。你所看不到的，荦荦大者也不过就是编译器为你的函式加上用以处理堆叠的一小段码（prologue 和 epilogue），这一小段码基本上做的是 housekeeping 工作，你没看到也没有关系（更好），并不影响你对程式逻辑的思考。

C++ 不一样，C++ 有太多和程式逻辑息息相关的动作是编译器为我们加上去的。换句话说 C++ 编译器为我们「加码」。如果不识清这一节，学习 C++ 有如雾里看花，雾非雾，花非花。

编译器为我们的 C++ 程式加了什麼码呢？很多！物件诞生时 ctor 会被唤起，物件死亡时 dtor 会被唤起，这都是加码的结果。ctor 中设定 vtpr 和 vtbl，这也是加码的结果。new 单一物件时会产生 memory block cookie，new 物件阵列时会产生一个内部结构记录着 object size 和 class ctor...，这也都是布幕後的工作。可以说，程式码中看不到而却必须完成的所有与程式逻辑有关的动作，统统都是 C++ 编译器加码後的结果。

当「继承」发生，整个情况变得稍微复杂起来。「多重继承」又更复杂一些，「虚拟继承」再更复杂一些。

这些布幕後的主题，统可归类为所谓的 C++ object model（物件模型）。如果不知道这些底层机制，你就只能够把 "make destructors virtual in base classes"（<Effective C++>, item14）或 "never treat arrays

polymorphically" (<More Effective C++>, item 3) 这类规则硬背下来，却不明白它的道理。

用一样东西，却不明白它的道理，林语堂如是说：『不高明』。只知道 how，不知道 why，侯捷如是说：『不高明』。

●困难度之二

C++ 的第二个学习难度在於 "paradigm shift"（思考模式的移转）。别说自己设计 classes 了，光使用别人的 classes，就都是一种思考模式和行为模式的移转。MFC（或 OWL 或 VCL）programmer 必然甚能够领略并体会我的意思。

使用所谓的 application framework（一种大型的、凝聚性强的、有着物件导向公共基础建设的 class library），你的码和 framework 之间究竟是怎样的关系呢？framework 提供的一大堆可改写的虚拟函式的意义与价值究竟在哪里呢？为什麼 framework 所设计的种种美好性质以及各式各样的演算法竟然可以施行於我们自己设计的 class types 身上呢？framework 被设计时，并不知道我们的存在呀！

这正是物件导向中的多型（polymorphism）的威力。

稍早所说的 C++ 物件模型，偏属程式设计的低层面；这里所说的思考模式移转，则是程式设计的高层面。能够把新思维模式的威力发挥得最淋漓尽致的，当推物件导向的 polymorphism（多型）和 generalization（泛型）。如果你没有使用这两项特性，等於入 C++ 宝山而空手返。

●反覆炼，循环震荡

想像 C++ 是一把用来解决程式问题的刀，要它坚韧，要它锋利，就必须经过多次的回火，在高热和骤冷之间炼。

初学 C++ 语法（syntax）之後，你应该尽快尝试体验 polymorphism （大致而言也就是虚拟函式的运用）。等到对 OOP 的精神有了大局掌控的能力，但对 C++ 的许多小细节不甚清楚，就是回到 C++ 物件模型炼的时机。

成长，是在高阶（polymorphism）和低阶（object model）之间反覆震荡，才能够震荡到更高的位阶，而不是平平庸庸於中阶（C++ syntax）的一滩死水。

●不要沉沦於 C++ syntax

100 个人跟我说他懂 C++/OOP，只有 10% 不到可以让我认为他没有胡吹大气。太多的人，上嘛上不到 polymorphism，下嘛又下不到 object model。就这样不上不下地卡在 C++ 语法层面。大一学了 C++，到大四快毕业了，连 virtual functions 是怎麼回事都期期艾艾支支吾吾说不出个道理。

有时候我觉得，太苛责同学也於心不忍，因为同学们事实上处於一种无知的状态，既不知道 C++/OOP 该怎麼学，也不知道哪些书可以教他们那麼学。所以，苛责同学，不如责怪老师。

众所周知，大学教授泰半是动口不动手，普遍的心态是「论文第一，升等为要；程式语言？哎，末流！」。「末流」课程通常由教授们轮流教，谁倒楣谁来教；於是就常常有「下学期要教 C++ 语言了，这学期寒（暑）假赶快去要本书来恶补」的情况发生。偏偏程式语言这东西，只动口不管用，一定要动手，而且要常动手。老师自己没有摸到 C++/OOP 的精神，学生又能学到什麼？

有些学校资讯系并不教特定的程式语言，老师们的态度是「语言是一种自己学就好了的东西嘛，拿到大学殿堂来，哎，不入流」！於是应该好好为学生打下实际基础的课程，却天马行空地腾云驾雾起来，大谈抽象意念。饱读经书的老师们可能忽略了，一个完全没有技术基础的学子，要的不是形而上的道，而是形而下的器。

我们是先能够欣赏具象画，还是先能够欣赏抽象画？我们不都是先对毕卡索的画大骂「这是什麼东西」，直到自己的艺术涵养够丰富了、人生阅练更饱满了、能够举一隅以三隅反了、能够接触类旁通左右逢源了，才转而能够欣赏甚至进入毕卡索的抽象意境吗？

老师们各有专长，要老师们来教非彼专长的大班课、基础课，我又觉得似乎也太为难老师了。那麼，苛责老师，不如责怪学校当局。如果学校当局能够聘请经验老道又有教学热诚的工程师来教这类实务学科，不是三方皆大欢喜吗？不要说什麼制度僵化啦，难以突破啦，大学是高度自治区，礼聘几位兼任老师，不全都是系上的权责范围内吗？

当学子们在课程上学不到他要的东西，就只好闭门自修。但是，循序性

（sequential）语言尚有自修学会的可能，物件导向语言嘛，以大学生的程度来讲，我认为自修实在困难，只会修出个四不像、半瓶水。

管不到学校！管不到教授！自求多福的情况下，希望看到这篇文章的你，知道 C++/OOP 该怎麼学。

●不要沉迷於 C++ semantics 和 C++ object model

对於底层知识有浓厚兴趣的朋友，下探到 object model 领域，一定会非常开心地在 object size、object layout、vptr/vtbl、以及许多布幕後的技术之间玩将起来。了解这些东西，当然是好的，但是由於一探究竟得其奥秘的快感与成就感，使得一些朋友们在这个层面里「玩」起来了，小地方玩得很精，玩得不亦乐乎，玩得忽略了 C++/OO 的最终目标。

最终目标是 polymorphism!

我要说，在 C++ syntax 以及相对低阶的 C++ semantics 里，不要玩得太过火。过犹不及，会伤身的。C++ 经典名书内附的一些习题，在我看来颇有点玩得过火的味道。至於什麼百题精选、题库大成，除了修练基本功之外，都满无趣的东西。

Programming 应该是一种天马行空的想像力与创意的组合；如果你能够自己想题目，譬如说实作一个天体运行的 class 体系、或是实作一个生物分类（界门纲目科属种）体系，不是很有趣吗？准备资料的过程中，查查百科全书，你也因此查到了太阳系九大行星的几何资料，哈雷慧星的轨道周期，或是黑面琵鹭的「界门纲目科属种」英文名称，这难道不比钻研於 ++++i 或 ----i 或 *&*&p 之类的头脑体操题目有趣吗？（看过不少这类好笑题目，没一个记下来，只好胡乱写几个运算式。诸位应该知道我说的那种头脑体操题目）

固然，在科学与工程的领域里头，无技术无以为立，但别把自己弄得过於僵化，过於匠气。僵化与匠气是我们教育体系的最大沉疴。到了高专层次，败象显露无遗。

●名书推荐

如果没有介绍几本好书，我就是为德不卒。

让我再节录〈多型与虚拟〉的二刷感言：

〈多型与虚拟〉一版二刷感言 （侯俊杰/松岗/1998/08）... C++ 相关书籍，如天上繁星，如过江之鲫。广博如四库全书者有之（如 The C++ Programming Language、C++ Primer），深奥宛如山重水复有之（如 Inside The C++ Object Model），独沽一味者有之（如 C++ Programming Style、More Effective C++），独树一帜者有之（如 The Design and Evolution of C++），另辟蹊径者亦有之（如 STL tutorial Reference Guide）。...

以下是我认为你应该要拥有的书籍。有趣的是，我才在自己班上做了一个调查（我教的是物件导向 Windows 程式设计，学生应该要有良好的 C++/OOP 基础），拥有以下 1~5 本书的人举手。举手人数都很少，而且老是那几位（最高记录是拥有四本）。这让我感觉，强者恒强，弱者恒弱。悲夫！

1. C++ Primer (3/e), Lippman/A.W./1998

（听说 1999 将有中译本）

2. The C++ Programming Language (3/e), Bjarne/A.W./1997

（听说 1999 将有中译本）

以上两本书是 C++ 经典百科。就内容水平而言，我认为同为瑜亮。 普遍的印象是，第一本较易接受，第二本涩味稍重。第二本书 作者 Bjarne 是 C++ 语言的创造者，所以有其权威性。我认识的多 位 C++/OOP 高手，都是两书齐具。

3. Inside The C++ Object Model, Lippman/A.W./1996

（中译本〈深度探索 C++ 物件模型〉）

全书讲解 C++ object model，上穷碧落下黄泉。内容很好，层次也高， 可惜原文书大大小小的错误繁如晨星，阅读时需小心。

4. Effective C++, Meyers/A.W./1992

（印象似有中译本，名称忘了，谁可补充说明？）

5. More Effective C++, Meyers/A.W./1996

（有中译本吗？我不知道，谁可补充说明？）

同一作者的这两本书，专讲 C++ programming 的重要观念，使你的程式 更稳健更有效率。书中许多观念涉及 C++ object model，与 (3) 书混合看，如鱼得水。

6. Polymorphism in C++〈多型与虚拟〉侯俊杰/松岗/1998

（没有中译本 -- 它本身就是中文书）

在语法粗具的基础上，直接把读者导引到最核心最重要的思想，并且 在建立这个思想的过程中，提供足够的必要基础。

我只列出一本中文书，是因为这方面的中文书我看得少，英文书看得多。「恐有遗珠之憾」这类「八方得体」的话，还是说一下好了 :)。

注意，这些都只是强本固元用来扎基础的书籍而已，要观摩大型程式经验，还有诸如 Large Scale C++ Software Design（John Lakos/A.W./1996）可以阅读。

OO 的世界，不止 OOP，还有 OOA/OOD，那又是一缸子的学问和一缸子的书。

C++语言程序设计试题

2001 年 1 月

说明：在本试卷中规定整型(int)数据占用 4 个字节的存储单元。

一、选择题(每小题 1 分，共 6 分)

1．由 C++目标文件连接而成的可执行文件的缺省扩展名为_____。

A． cpp　　B． exe

C． obj　　D． lik

2．在下面的一维数组定义中，哪一个有语法错误。_____

A． int a[ ]={1,2,3}　　B． int a[10]={0}

C． int a[ ]　　D． int a[5]

3．在下面的函数声明中，存在着语法错误的是_____。

A． void BC(int a , int)　　B． void BD(int , int)

C． void BE(int , int=5)　　D． int BF(int x ; int y)

4. 假定 AB 为一个类，则该类的拷贝构造函数的声明语句为_____。

A． AB &(AB x)　　B． AB(AB x)

C． AB(AB &)　　D． AB(AB * x)

5．对于结构中定义的成员，其隐含访问权限为_____。

A． public　　B． protected

C． private　　D． static

6. 当使用 fstream 流类定义一个流对象并打开一个磁盘文件时，文件的隐含打开方式为____。

A． ios::in　　B． ios::out

C． ios::int | ios::out　　D． 没有

二、填空题(每小题 2 分，共 24 分)

1．执行“cout <<43<<’-‘<<18<<’=’<<43-18<<endl;”语句后得到的输出结果为____。

2．已知’A’～’Z’的 ASCII 码为 65～90，当执行“char ch=14*5+2; cout <<ch<<endl;”

语句序列后，得到的输出结果为_____。
3．使用 const 语句定义一个标识符常量时，则必须对它同时进行_____。
4．表达式 x=x+1 表示成增量表达式为_____。
5．若 x=5，y=10，则 x>y 和 x<=y 的逻辑值分别为_____和_____。
6．执行“typedef int ABC[10]；”语句把 ABC 定义为具有 10 个整型元素的_____。
7．假定 p 所指对象的值为 25，p+1 所指对象的值为 46，则执行“(*p)++；”语句后，p 所指对象的值为_____。
8．假定一个二维数组为 a[M][N]，则 a[i]的地址值(以字节为单位)为_____。
9．假定要访问一个结构指针 p 所指对象中的 b 指针成员所指的对象，则表示方法为_____。
10．设 px 是指向一个类动态对象的指针变量，则执行“delete px；”语句时，将自动调用该类的_____。
11．若需要把一个函数“void F( )；”定义为一个类 AB 的友元函数，则应在类 AB 的定义中加入一条语句：_____。
12．若要在程序文件中进行标准输入输出操作，则必须在开始的 # include 命令中使用_____头文件。

三、下列程序运行后的输出结果(每小题 6 分，共 36 分)

1．
```
# include <iostream.h>
void main()
{
    int s=0;
    for (int i=1;  ; i++) {
        if (s>50) break;
        if (i%2==0)  s+=i;
    }
    cout  <<"i,s="<<i<<","<<s<<endl;
}
```

2．
```
# include <iostream.h>
void main()
{
    char a[]="abcdabcabfgacd";
    int i1=0,i2=0,i=0;
    while (a[i]) {
        if (a[i]=='a') i1++;
        if (a[i]=='b') i2++;
        i++;
    }
    cout <<i1<<' '<<i2<<endl;
}
```

3．
```
# include <iomanip.h>
void main()
{
    int a[9]={2,4,6,8,10,12,14,16,18};
    for (int i=0; i<9; i++) {
        cout <<setw(5)<<*(a+i);
        if ((i+1)%3==0) cout <<endl;
    }
}
```

4．
```
# include <iomanip.h>
void LE(int * a,int * b) {
    int x=*a;
```

```
        *a=*b;   *b=x;
        cout <<*a<<' '<<*b<<endl;
    }
    void main()
    {
        int x=10,y=25;
        LE(&x,&y); cout <<x<<' '<<y<<endl;
}
```

5.
```
# include <iostream.h>
class A {
    int a,b;
public :
    A() { a=b=0; }
    A(int aa,int bb) {
        a=aa; b=bb;
        cout <<a<<' '<<b<<endl;
    }
};
void main()
{
    A x,y(2,3),z(4,5);
}
```

6.
```
# include <iostream.h>
template <class TT>
class FF {
    TT a1,a2,a3;
public :
    FF(TT b1,TT b2,TT b3) {
        a1=b1; a2=b2; a3=b3;
    }
    TT Sum() { return a1+a2+a3; }
};
void main()
{
    FF <int> x(2,3,4),y(5,7,9);
    cout <<x.Sum()<<' '<<y.Sum()<<endl;
}
```

四、写出下列每个函数的功能(每小题 6 分，共 24 分)

1.
```
double SF(double x,int n) {
    // n 为大于等于 0 的整数
    double p=1,s=1;
    for (int i=1; i<=n; i++) {
        p*=x;
        s+=p/(i+1);
    }
    return s;
}
```

2.
```
float FH() {
    float x,y=0,n=0;
    cin >>x;
    while (x!=-1) {
        n++; y+=x;
```

```
            cin >>x;
        }
        if (n==0) return y; else return y/n;
    }
```

3.
```
    # include <iostream.h>
    void WA(int a[],int n) {
        for (int i=0; i<n-1; i++) {
            int k=i;
            for (int j=i+1; j<n; j++)
                if (a[j]<a[k]) k=j;
                int x=a[i]; a[i]=a[k]; a[k]=x;
        }
    }
```

4.
```
    # include <iomanip.h>
    # include <fstream.h>
    void JB(char * fname)
        // 可把以 fname 所指字符串作为文件标识符的文件称为 fname 文件
        // 假定该文件中保存着一批字符串，每个字符串的长度均小于 20
    {
        ifstream fin(fname);
        char a[20];
        int i=0;
        while (fin>>a) {
            cout <<a<<endl;
            i++;
        }
        fin.close();
        cout <<"i="<<i<<endl;
    }
```

五、编写一个函数，统计出具有 n 个元素的一维数组中大于等于所有元素平均值的元素个数并返回。(10 分)

```
int   Count(double a[],int n);   // 此为该函数的声明。
```

C++语言程序设计

试题答案及评分标准

(供参考)

2001 年 1 月

一、选择题(每小题 1 分，共 6 分)

评分标准：选对者得 1 分，否则不得分。

1. B　　2. C　　3. D
1. C　　5. A　　6. D

二、填空题(每小题 2 分，共 24 分)

评分标准：每题与参考答案相同者得 2 分，否则酌情给分。

1. 43-18=25　　2. H
3. 初始化　　4. ++x
5. false 或 0　true 或 1　　6. 数组类型
7. 26

8. a+(i*N)*sizeof(a[0][0]) 或 a+i*sizeof(a[i])
9. *(p->b) 10. 析构函数
11. friend void F( ); 11. iostream.h 或 iomanip.h

三、下列程序运行后的输出结果(每小题 6 分，共 36 分)

评分标准：每题与参考答案的数据和显示格式完全相同者得 6 分，否则酌情给分。

1. i,s=15,56
2. 4 3
3. 2 4 6
   8 10 12
   14 16 18
4. 25 10
   25 10
5. 2 3
   4 5
6. 9 21

四、写出下列每个函数的功能(每小题 6 分，共 24 分)

评分标准：每题与参考答案的叙述含义相同者得 6 分，否则酌情给分。

1．计算 $1+X/2+X^2/3+\ldots+X^n/(n+1)$的值并返回。

2．求出从键盘上输入的一批常数的平均值，以-1 作为结束输入的标志。

3．采用选择排序的方法对数组 a 中的 n 个整数按照从小到大有次序重新排列。

4．从向文件 fname 中依次读取每个字符串并输出到屏幕上显示出来，同时统计并显示出文件中的字符串个数。

五、编写一个函数，统计出具有 n 个元素的一维数组中大于等于所有元素平均值的元素个数并返回。(10 分)

评分标准见参考程序中的注释。

```
int Count(double a[],int n) {
    double m=0;
    int i;
    for (i=0; i<n; i++) m+=a[i];   // 计算出所有元素之和得 3 分
    m=m/n;    // 计算出平均值得 1 分
    int c=0;
    for (i=0; i<n; i++)
        if (a[i]>m) c++;    // 按条件统计出元素个数得 4 分
    return c;    // 返回统计结果得 2 分
}
```

C++语言程序设计试题

2001 年 3 月

说明：在本试卷中统一规定整型(int)数据占用 4 个字节的存储单元。

一、单选题(每小题 1 分，共 6 分)

1、在每个 C++程序中都必须包含有这样一个函数，该函数的函数名为_____。
A. main B. MAIN C. name D. function

2、设 x 和 y 均为 bool 量，则 x&&y 为真的条件是_____。
A. 它们均为真 B. 其中一个为真 C. 它们均为假 D. 其中一个为假

3、下面的哪个保留字不能作为函数的返回类型？_____。
A. void B. int C. new D. long

4、假定 a 为一个整型数组名，则元素 a[4]的字节地址为_____。
A. a+4 B. a+8 C. a+16 D. a+32

5、假定 AB 为一个类，则执行“AB a(4) , b[3] , * p[2] ;”语句时，自动调用该类构造函数的次数为_____。

A. 3　　B. 4　　C. 6　　D. 9

6、假定要对类 AB 定义加号操作符重载成员函数，实现两个 AB 类对象的加法，并返回相加结果，则该成员函数的声明语句为：_____。

A. AB operator+(AB & a , AB & b)　　B. AB operator+(AB & a)

C. operator+(AB a)　　D. AB & operator+( )

二、填空题(每小题 2 分，共 24 分)

1、C++语言中的每条基本语句以_____作为结束符，每条复合语句以_____作为结束符。

2、执行“cout <<char('A'+2)<<endl;”语句后得到的输出结果为_____。

3、float 和 double 类型的大小分别为_____和_____。

4、算术表达式　　对应的 C++表达式为_____。

5、关系表达式 x+y>5 的相反表达式为_____。

6、假定一个一维数组的定义为“char * a[8] ;”，则该数组所含元素的个数为_____，所占存储空间的字节数为_____。

7、变量分为全局和局部两种，_____变量没有赋初值时，其值是不确定的。

8、假定 a 是一个二维数组，则 a[i][j]的指针访问方式为_____。

9、假定一个结构类型定义为

“struct D { int a ; union { int b ; double c ; } ; D * d[2] ; } ;” ，

则该类型的大小为_____字节。

10、对一个类中的数据成员的初始化可以通过构造函数中的_____实现，也可以通过构造函数中的_____实现。

11、假定 AB 为一个类，则执行“AB a[10];”语句时，系统自动调用该类的构造函数的次数为_____。

12、假定类 AB 中有一个公用属性的静态数据成员 bb，在类外不通过对象名访问该成员 bb 的写法为_____ 。

三、给出下列程序运行后的输出结果(每小题 6 分，共 36 分)

1、
```
# include <iostream.h>
void SB(char ch) {
    switch(ch){
    case 'A': case 'a':
        cout <<"well!"; break;
    case 'B': case 'b':
        cout <<"good!"; break;
    case 'C': case 'c':
        cout <<"pass!"; break;
    default:
        cout <<"nad!"; break;
    }
}
void main() {
    char a1='b',a2='C',a3='f';
    SB(a1);SB(a2);SB(a3);SB('A');
    cout <<endl;
```

```
}
```

2、
```
# include <iostream.h>
# include <string.h>

void main() {
    char *a[5]={"student","worker","cadre","soldier","peasant"};
    char *p1,*p2;
    p1=p2=a[0];
    for (int i=0; i<5; i++) {
        if (strcmp(a[i],p1)>0) p1=a[i];
        if (strcmp(a[i],p2)<0) p2=a[i];
    }
    cout <<p1<<' '<<p2<<endl;
}
```

3、
```
# include <iostream.h>
int a=5;
void main() {
    int a=10,b=20;
    cout <<a<<' '<<b<<endl;
    {   int a=0,b=0;
        for (int i=1; i<6; i++) {
          a+=i; b+=a;
    }
        cout <<a<<' '<<b<<' '<<::a<<endl;
    }
    cout <<a<<' '<<b<<endl;
}
```

4、
```
# include <iomanip.h>
int LB(int *a,int n) {
    int s=1;
    for (int i=0; i<n; i++)
        s*=*a++;
    return s;
}
void main() {
    int a[]={1,2,3,4,5,6,7,8};
    int b=LB(a,5)+LB(&a[3],3);
    cout <<"b="<<b<<endl;
}
```

5、
```
# include <iostream.h>
# include <string.h>
struct Worker{
    char name[15];   // 姓名
    int age;         // 年龄
    float pay;       // 工资
};
void main() {
    Worker x;
    char *t="liouting";
    int d=38; float f=493;
    strcpy(x.name,t);
    x.age=d; x.pay=f;
```

```
    cout <<x.name<<' '<<x.age<<' '<<x.pay<<endl;
}
```

6、
```
# include <iostream.h>
class A {
    int a;
public:
    A(int aa=0) { a=aa; }
    ~A() { cout <<"Destructor A!"<<a<<endl;  }
};
class B:public A {
    int b;
public:
    B(int aa=0,int bb=0):A(aa) { b=bb; }
    ~B() { cout <<"Destructor B!"<<b<<endl; }
};
void main() {
    B x(5),y(6,7);  // 后定义的变量将先被释放
}
```

四、写出下列每个函数的功能(每小题 6 分，共 24 分)

1、
```
# include <iostream.h>
int SA(int a,int b) {
    if (a>b) return 1;
    else if (a==b) return 0;
    else return -1;
}
```

2、
```
float FI(int n) {
    // n 为大于等于 1 的整数
    float x,y=0;
    do {
        cin >>x;
        n--; y+=x*x;
    } while (n>0);
    return  y;
}
```

3、
```
template <class Type>
void WE(Type a[],Type b[],int n) {
    for (int i=0; i<n; i++)
        b[n-i-1]=a[i];
}
```

4、
```
struct StrNode {
    char name[15];     // 字符串域
    StrNode * next;    // 指针域
};
void QB(StrNode * & f ,int n) {
    if (n==0) { f=NULL; return; }
    f=new StrNode;
    cin >>f->name;
    StrNode * p=f;
    while (--n) {
        p=p->next=new StrNode;
        cin >>p->name;
```

```
    }
    p->next=NULL;
  }
```

五、编写程序，把从键盘上输入的一批整数(以-1 作为终止输入的标志)保存到文本文件“a:xxk1.dat”中。(10 分)

****

C++语言程序设计试题
答案及评分标准
(供参考)

2001 年 3 月

一、单选题(每小题 1 分，共 6 分)

评分标准：选对者得 1 分，否则不得分。

1、A　　2、A　　3、C　　4、C　　5、B　　6、B

二、填空题(每小题 2 分，共 24 分)

评分标准：每题与参考答案相同者得 2 分，否则不得分。

1、　:　　}　　2、　C

3、　4　8　　4、　(x*y*y)/(3*a)+4*b-1

5、　x+y<=5　　6、　8　32

7、　局部　　8、　*(a[i]+j)　或　*(*(a+i)+j)

9、　20　　10、　初始化表　函数体

11、　10　　12、　AB::bb

三、给出下列程序运行后的输出结果(每小题 6 分，共 36 分)

评分标准：每题与参考答案的数据和显示格式完全相同者得 6 分，否则酌情给分。

1、　good!　pass!　bad!　well!

2、　worker　cadre

3、　10　20
　　15　35　5
　　10　20

4、　b=240

5、　liouting　38　493

6、　Destructor　B!　7
　　Destructor　A!　6
　　Destructor　B!　0
　　Destructor　A!　5

四、写出下列每个函数的功能(每小题 6 分，共 24 分)

评分标准：每题与参考答案的叙述含义相同者得 6 分，否则酌情给分。

1、比较两个整数 a 和 b 的大小，若 a>b 则返回 1，若 a==b 则返回 0，若 a<b 则返回-1。

2、求出从键盘上输入的 n 个常数的平方和并返回。

3、模板函数，把数组 a 的每个元素按逆序放入数组 b 中。

4、建立一个具有 n 个结点的链表，每个结点的字符串值由键盘输入，链表的表头指针由引用变量 f 带回。

五、编写程序，把从键盘上输入的一批整数(以-1 作为终止输入的标志)保存到文本文件“a:xxk1.dat”中。(10 分)

评分标准：见参考程序中的注释。

```
# include <iostream.h>    // 使用此命令得 1 分
```

```
# include <fstream.h>
# include <stdlib.h>

void main() {
    ofstream fout("a:xxk1.dat");  // 定义输出文件流并打开文件得 2 分
    if (!fout){
        cerr <<"文件没有打开！"<<endl;
        exit(1);
    }       // 可有可无
    int x;
    cin >>x;
    while (x!=-1) {
        fout <<x<<' ';
        cin >>x;
    }   // 能够从键盘向文件正确输出数据得 6 分
    fout.close();   // 关闭输出文件流得 1 分
}
```

DES 加密算法破解方法

DES(数据加密标准)在 1977 年被美国国家标准技术协会认可成为均衡加密算法的标准,用于加密非机密的信息.des 广泛应用于各个行业的加密领域,如银行业.这麼样一种古老的加密算法,到今天还有人在用,真是让人想不明白.这种按照摩尔定律早该淘汰的东西,怎么可能会没有办法破解呢??

以下是 6 种破解 des 的方法:

1.暴力破解

上一次的主页更新已经介绍过了,在这里不再复述.

2.分布式计算

通过网络联合数台计算机一起计算.可以大大缩短时间.

3.专用设备破解(破解机)

暴力破解实在是太费时间,但是个人计算机不是最快的破解工具,PC 终归是一种通用设备.在 1998 年,EFF 为了向世人证明 des 不是一种安全的加密方式而制造了一台专用于破解 des 的机器,这台机器叫做 Deep Crack ,总共耗资 20 万美元,该机器使用 1536 个专用处理器,平均破解(穷举)出一个正确的 key 需耗时 4 天左右. 每秒钟可以穷举 920 亿个 key.

4.时间与数据量折衷法.

这是马丁赫尔曼先生于 1980 年提出的一种可行的破解 des 的算法, 可以想象这样一种情况,我们有无穷多的存储器,我们预先把所有可能的 key(A)和与某个明文通过这个 key 所得到的相应的密文(B)组成一对(A,B)存在存储器中.我们就可以通过数据库快速的找到我们需要的 key,当我们有足够的存储器的时候,这是最快的方法,那么需要多少存储器呢??你可以自己算一下.:)))

当然,我们没有那么多的硬盘来村这些数据,但是马丁赫尔曼提出了一种新的算法来解决这个问题,按照一定的规则选一部分 key 把相应的数据对(A,B)存在硬盘中,再按照相应的算法通过数据库的搜索结果,把正确的 key 锁定在很小的范围内,然后在这一范围内进行穷举.按照这一方法,一台普通的微机只需要 1000G 的硬盘和 3 天左右的时间就可以找到正确的 key.

5.微分密码分析法.

1990 年,;两名以色列密码专家发明了一种新的方法来破解 des,这就是微分密码分析法. 按照这一方法只需要对特殊的明文和密文成对采样 247 对,通过短时间的分析便可以得到正确的 key.

具体算法吗.....hehehe...:))
6.线性分析法.
日本三菱电子 1994 年发明的方法,按照这一方法如果我们有 2^43==8'796'093'022'208 个明文和密文对(约 64'000 GB),我们可以在短时间内计算出正确的 key.
ok,大家有信心了吧.

DES 算法及其应用误区

在银行金融界及非金融界，越来越多地用到了 DES 算法， DES 全称为 Data Encryption Standard 即数据加密算法，它是 IBM 公司于 1975 年研究成功并公开发表的，目前在国内，随着三金工程尤其是金卡工程的启动，DES 算法在 POS、ATM、磁卡及智能卡（IC 卡）、加油站、高速公路收费站等领域被广泛应用，以此来实现关键数据的保密，如信用卡持卡人的 PIN 的加密传输，IC 卡与 POS 间的双向认证、金融交易数据包的 MAC 校验等，均用到 DES 算法。

DES 算法的入口参数有三个：Key、Data、Mode。其中 Key 为 8 个字节共 64 位，是 DES 算法的工作密钥；Data 也为 8 个字节 64 位，是要被加密或被解密的数据；Mode 为 DES 的工作方式，有两种：加密或解密。

DES 算法是这样工作的：如 Mode 为加密，则用 Key 去把数据 Data 进行加密， 生成 Data 的密码形式（64 位）作为 DES 的输出结果；如 Mode 为解密，则用 Key 去把密码形式的数据 Data 解密，还原为 Data 的明码形式（64 位）作为 DES 的输出结果。在通信网络的两端，双方约定一致的 Key，在通信的源点用 Key 对核心数据进行 DES 加密，然后以密码形式在公共通信网（如电话网）中传输到通信网络的终点，数据到达目的地后，用同样的 Key 对密码数据进行解密，便再现了明码形式的核心数据。这样，便保证了核心数据（如 PIN、MAC 等）在公共通信网中传输的安全性和可靠性。

通过定期在通信网络的源端和目的端同时改用新的 Key，便能更进一步提高数据的保密性，这正是现在金融交易网络的流行做法。

DES 算法详述

DES 算法把 64 位的明文输入块变为 64 位的密文输出块，它所使用的密钥也是 64 位，整个算法的主流程图如下：
其功能是把输入的 64 位数据块按位重新组合，并把输出分为 L0、R0 两部分，每部分各长 32 位，其置换规则见下表：
58,50,12,34,26,18,10,2,60,52,44,36,28,20,12,4,
62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
57,49,41,33,25,17, 9,1,59,51,43,35,27,19,11,3,
61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7,

即将输入的第 58 位换到第一位，第 50 位换到第 2 位，...，依此类推，最后一位是原来的第 7 位。L0、R0 则是换位输出后的两部分，L0 是输出的左 32 位，R0 是右 32 位，例：设置换前的输入值为 D1D2D3.....D64，则经过初始置换后的结果为：L0=D58D50...D8；R0=D57D49...D7。

经过 26 次迭代运算后。得到 L16、R16，将此作为输入，进行逆置换，即得到密文输出。逆置换正好是初始置的逆运算，例如，第 1 位经过初始置换后，处于第 40 位，而通过逆置换，又将第 40 位换回到第 1 位，其逆置换规则如下表所示：
40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,
38,6,46,14,54,22,62,30,37,5,45,13,53,21,61,29,
36,4,44,12,52,20,60,28,35,3,43,11,51,19,59,27,

34,2,42,10,50,18,58 26,33,1,41, 9,49,17,57,25,

放大换位表

32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9, 10,11,

12,13,12,13,14,15,16,17,16,17,18,19,20,21,20,21,

22,23,24,25,24,25,26,27,28,29,28,29,30,31,32, 1,

单纯换位表

16,7,20,21,29,12,28,17, 1,15,23,26, 5,18,31,10,

2,8,24,14,32,27, 3, 9,19,13,30, 6,22,11, 4,25,

在 f(Ri,Ki)算法描述图中，S1,S2...S8 为选择函数，其功能是把 6bit 数据变为 4bit 数据。下面给出选择函数 Si(i=1,2.....8)的功能表：

选择函数 Si

S1:

14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,

0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,

4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,

15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13,

S2:

15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10,

3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5,

0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,

13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9,

S3:

10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,

13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,

13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,

1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12,

S4:

7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,

13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,

10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,

3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14,

S5:

2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,

14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,

4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,

11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3,

S6:

12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,

10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,

9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,

4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13,

S7:

4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,

13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,

1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,

6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12,

S8:

13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,

1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,

7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,

2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11,

在此以 S1 为例说明其功能，我们可以看到：在 S1 中，共有 4 行数据，命名为 0，1、2、3 行；每行有 16 列，命名为 0、1、2、3，.....，14、15 列。

现设输入为： D＝D1D2D3D4D5D6

令：列＝D2D3D4D5

行＝D1D6

然后在 S1 表中查得对应的数，以 4 位二进制表示，此即为选择函数 S1 的输出。下面给出子密钥 Ki(48bit)的生成算法

从子密钥 Ki 的生成算法描述图中我们可以看到：初始 Key 值为 64 位，但 DES 算法规定，其中第 8、16、.....64 位是奇偶校验位，不参与 DES 运算。故 Key 实际可用位数便只有 56 位。即：经过缩小选择换位表 1 的变换后，Key 的位数由 64 位变成了 56 位，此 56 位分为 C0、D0 两部分，各 28 位，然后分别进行第 1 次循环左移，得到 C1、D1，将 C1（28 位）、D1（28 位）合并得到 56 位，再经过缩小选择换位 2，从而便得到了密钥 K0（48 位）。依此类推，便可得到 K1、K2、.....、K15，不过需要注意的是，16 次循环左移对应的左移位数要依据下述规则进行：

循环左移位数

1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1

以上介绍了 DES 算法的加密过程。DES 算法的解密过程是一样的，区别仅仅在于第一次迭代时用子密钥 K15，第二次 K14、.....，最后一次用 K0，算法本身并没有任何变化。

DES 算法的应用误区

DES 算法具有极高安全性，到目前为止，除了用穷举搜索法对 DES 算法进行攻击外，还没有发现更有效的办法。而 56 位长的密钥的穷举空间为 256，这意味着如果一台计算机的速度是每一秒种检测一百万个密钥，则它搜索完全部密钥就需要将近 2285 年的时间，可见，这是难以实现的，当然，随着科学技术的发展，当出现超高速计算机后，我们可考虑把 DES 密钥的长度再增长一些，以此来达到更高的保密程度。

由上述 DES 算法介绍我们可以看到：DES 算法中只用到 64 位密钥中的其中 56 位，而第 8、16、24、.....64 位 8 个位并未参与 DES 运算，这一点，向我们提出了一个应用上的要求，即 DES 的安全性是基于除了 8，16，24，.....64 位外的其余 56 位的组合变化 256 才得以保证的。因此，在实际应用中，我们应避开使用第 8，16，24，.....64 位作为有效数据位，而使用其它的 56 位作为有效数据位，才能保证 DES 算法安全可靠地发挥作用。如果不了解这一点，把密钥 Key 的 8，16，24，..... .64 位作为有效数据使用，将不能保证 DES 加密数据的安全性，对运用 DES 来达到保密作用的系统产生数据被破译的危险，这正是 DES 算法在应用上的误区，是各级技术人员、各级领导在使用过程中应绝对避免的，而当今国内各金融部门及非金融部门，在运用 DES 工作，掌握 DES 工作密钥 Key 的领导、主管们，极易忽略，给使用中貌似安全的系统，留下了被人攻击、被人破译的极大隐患。

DES 算法应用误区的验证数据

笔者用 Turbo C 编写了 DES 算法程序，并在 PC 机上对上述的 DES 算法的应用误区进行了骓，其验证数据如下：

Key: 0x30 0x30 0x30 0x30.....0x30（8 个字节）
Data: 0x31 0x31 0x31 0x31.....0x31（8 个字节）
Mode: Encryption
结果：65 5e a6 28 cf 62 58 5f

如果把上述的 Key 换为 8 个字节的 0x31，而 Data 和 Mode 均不变，则执行 DES 后得到的密文完全一样。类似地，用 Key:8 个 0x32 和用 Key:8 个 0x33 去加密 Data （8 个 0x31），二者的图文输出也是相同的：5e c3 ac e9 53 71 3b ba

我们可以得到出结论：

Key 用 0x30 与用 0x31 是一样的；

Key 用 0x32 与用 0x33 是一样的，.....

当 Key 由 8 个 0x32 换成 8 个 0x31 后，貌似换成了新的 Key，但由于 0x30 和 0x31 仅仅是在第 8，16，24.....64 有变化，而 DES 算法并不使用 Key 的第 8，16，.....64 位作为 Key 的有效数据位，故：加密出的结果是一样的。

DES 解密的验证数据：

Key: 0x31 0x31.....0x31（8 个 0x31）
Data: 65 5e a6 28 cf 62 58 5f
Mode: Decryption
结果：0x31 0x31.....0x31（8 个 0x31）

由以上看出：DES 算法加密与解密均工作正确。唯一需要避免的是：在应用中，避开使用 Key 的第 8，16.....64 位作为有效数据位，从而便避开了 DES 算法在应用中的误区。

避开 DES 算法应用误区的具体操作

在 DES 密钥 Key 的使用、管理及密钥更换的过程中，应绝对避开 DES 算法的应用误区，即：绝对不能把 Key 的第 8，16，24.....64 位作为有效数据位，来对 Key 进行管理。这一点，特别推荐给金融银行界及非金融业界的领导及决策者们，尤其是负责管理密钥的人，要对此点予以高度重视。有的银行金融交易网络，利用定期更换 DES 密钥 Key 的办法来进一步提高系统的安全性和可靠性，如果忽略了上述应用误区，那么，更换新密钥将是徒劳的，对金融交易网络的安全运行将是十分危险的，所以更换密钥一定要保证新 Key 与旧 Key 真正的不同，即除了第 8，16，24，...64 位外其它位数据发生了变化，请务必对此保持高度重视！

N 皇后问题

一【题目】N 皇后问题(含八皇后问题的扩展，规则同八皇后)：在 N*N 的棋盘上，放置 N 个皇
后，要求每一横行每一列，每一对角线上均只能放置一个皇后，求解可能的方案及方案数。
下面是算法的实现源码，请大家讨论。

```
const max=8;
var i,j:integer;
a:array[1..max] of 0..max; {放皇后数组}
b:array[2..2*max] of boolean; {/对角线标志数组}
c:array[-(max-1)..max-1] of boolean; {\对角线标志数组}
col:array[1..max] of boolean; {列标志数组}
```

```
  total:integer; {统计总数}
  procedure output; {输出}
  var i:integer;
begin
  write('No.':4,'[',total+1:2,']');
  for i:=1 to max do
        write(a[i]:3);write(' ');
  if (total+1) mod 2 =0
       then writeln; inc(total);
end;
 function ok(i,dep:integer):boolean; {判断第 dep 行第 i 列可放否}
    begin
      ok:=false;
       if ( b[i+dep]=true) and ( c[dep-i]=true) {and (a[dep]=0)} and
                (col[i]=true)
       then ok:=true
     end;
  procedure try(dep:integer);
     var i,j:integer;
    begin
       for i:=1 to max do {每一行均有 max 种放法}
          if ok(i,dep) then
           begin
           a[dep]:=i;
             b[i+dep]:=false; {/对角线已放标志}
             c[dep-i]:=false; {\对角线已放标志}
           col[i]:=false; {列已放标志}
       if dep=max then output
          else try(dep+1); {递归下一层}
        a[dep]:=0; {取走皇后,回溯}
        b[i+dep]:=true; {恢复标志数组}
        c[dep-i]:=true;
        col[i]:=true;
   end;
  end;
begin
    for i:=1 to max do
        begin
         a[i]:=0;col[i]:=true;
        end;
        for i:=2 to 2*max do
          b[i]:=true;
        for i:=-(max-1) to max-1 do
           c[i]:=true;
```

```
        total:=0;
        try(1);
        writeln('total:',total);
    end.
```

采用循环双向链表，能实现多个长整型进行加法运算

```
/*
*  文件名: 1_2.c
*  实验环境: Turbo C 2.0
*  完成时间: 2003 年 2 月 17 日
*-------------------------------------------------------------------
*  改进说明:  采用循环双向链表，能实现多个长整型进行加法运算.
*/

#include <math.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0
#define OPERAND_NUM 2
#define POSITIVE 1
#define NEGATIVE 0

typedef int ElemType;
typedef int status;
typedef struct NodeType
{
ElemType data;
struct NodeType *prior;
struct NodeType *next;
} NodeType, *LinkType;

status CreateOpHeadBuff(LinkType **, int);
status CreateOperandList(LinkType *, int);
void CreateResultList(LinkType *, LinkType *, int);
status DeleteZero(LinkType *);
status PushDataToList(LinkType *, int, int);
status AppendNodeToList(LinkType *, LinkType);
LinkType ComputePNList(LinkType, LinkType, int);
LinkType ComputePPNNList(LinkType, LinkType, int);
status MakeNode(LinkType *, ElemType);
status PrintList(LinkType);
status ClearMemory(LinkType *, int);
status DeleteList(LinkType);
status ErrorProcess(char[], int);

int main(void)
{
int iCounter,
iOpNum = 2;/*  操作数的个数，默认为 2 */
```

```
char strNum[10], cOrder[5];
LinkType ResultList = NULL, /* 结果链表的头指针 */
*ListHeadBuff = NULL; /* 指向操作数头指针缓冲 */

do
{
printf("请输入需要的操作数的个数, 注意至少为 2: ");
gets(strNum);
iOpNum = atoi(strNum);
} while (iOpNum < 2);
/* 构造操作数链表的头指针缓冲区 */
CreateOpHeadBuff(&ListHeadBuff, iOpNum);
/* 提示用户输入数据,并构造操作数链表 */
while (!CreateOperandList(ListHeadBuff, iOpNum))
{
printf("\n 出现非法输入, 需要退出吗?\n");
printf("键入 Y 则退出, 键入 N 重新输入(Y/N):");
gets(cOrder);
if (cOrder[0] == 'Y' || cOrder[0] == 'y')
{
ClearMemory(ListHeadBuff, iOpNum);
return 0;
}
}
printf("打印输入情况:\n");
for (iCounter = 0; iCounter < iOpNum; iCounter++)
{
printf("- - - 第%d 个操作数 - - -\n", iCounter + 1);
DeleteZero(ListHeadBuff + iCounter);
PrintList(*(ListHeadBuff + iCounter));
}

/* 相加所有操作数链表的结果，并存放在 ResultList 中*/
CreateResultList(&ResultList, ListHeadBuff, iOpNum);
printf("打印结果:\n");
PrintList(ResultList);

ClearMemory(ListHeadBuff, iOpNum);
DeleteList(ResultList);
printf("运算完毕!");
getch();

return 0;
}

status CreateOpHeadBuff(LinkType **pBuff, int size)
{
int iCounter;

*pBuff = (LinkType *)malloc(sizeof(LinkType) * size);
if (!*pBuff)
{
printf("Error, the memory is overflow!\n");
return FALSE;
}
for (iCounter = 0; iCounter < size; iCounter++)
```

```
*(*pBuff + iCounter) = NULL;

return TRUE;
}

status CreateOperandList(LinkType *headBuff, int iOpNum)
{
int iCounter = 0, iTemp = 0,
iNodeNum = 0, /* 记录每一个操作数链表中加入的操作数个数 */
iSign = POSITIVE; /* 标识操作数的正(1)负(0)，初始为正的 */
char strScrNum[150], /* 用户输入的所有操作数字符 */
*cpCurr, /* 当前操作数字符尾 */
*cpCurrNum, /* 当前操作数字符头 */
strTsl[7]; /* 准备转换的操作数字符 */
LinkType NewNode;

printf("请输入所有操作数\n");
printf("例如输入 3 个操作数: \n\
1111, 2222; -3333, 4444; -3333, 9999, 0202;\n: ");
gets(strScrNum);

/* 检测输入正确性 */
if (!ErrorProcess(strScrNum, iOpNum)) return FALSE;

for (cpCurr = cpCurrNum = strScrNum; *cpCurr != '\0'; cpCurr++)
{
if (*cpCurr == ',' || *cpCurr == ';')
{
if (*(cpCurr + 1) == '\0') cpCurr++;
strncpy(strTsl, cpCurrNum, cpCurr - cpCurrNum);
strTsl[cpCurr - cpCurrNum] = '\0';
iTemp = atol(strTsl);
/* 异常处理,如 strTsl=="-3333","10000" */
if (0 > iTemp || iTemp > 9999)
{
printf("\n 出现非法输入 2!\n");
return FALSE;
}
/* 为操作数链表加入结点 */
MakeNode(&NewNode, iTemp);
AppendNodeToList(headBuff + iCounter, NewNode);
iNodeNum++; /* 当前链表已经加入的一个结点 */
if (*cpCurr == ';')
{
/* 将控制结点插在链表头 */
PushDataToList(headBuff + iCounter, iNodeNum, iSign);
iNodeNum = 0; /* 逻辑结点个数初始化为 0 */
iSign = POSITIVE; /* 符号标识默认为正的 */
if ((iCounter + 1) < iOpNum)
iCounter++; /* 标识下一个链表头指针 */
}
cpCurrNum = cpCurr + 1;
}
else if (*cpCurr == '-')
{
```

```
iSign = NEGATIVE; /* 符号标识改为负的 */
cpCurr++;
cpCurrNum = cpCurr;
}
else if (*cpCurr == '+');
/* 读完后停止构造操作数链表 */
if (*cpCurr == '\0')
{
PushDataToList(headBuff + iCounter, iNodeNum, iSign);
break;
}
} /* end for */

return TRUE;
}

/*
* 正正,结果为正的.
* 负负,结果为负的.
* 长正短负,结果为正的.
* 长负短正,要变为长正短负,结果为负的.
* 异号时同样长
* 注意要删除每次算出的中间链表，最后传回 Result
*/
void CreateResultList(LinkType *ResultHead,
LinkType *headBuff, int iOpNum)
{
int iCounter, iSign;
LinkType ResultList = NULL, TempList, CurrNode_1, CurrNode_2;

for (ResultList = *headBuff, iCounter = 1;
iCounter < iOpNum; iCounter++)
{
TempList = ResultList;
if (ResultList->data > 0 &&
(*(headBuff + iCounter))->data > 0)/* 正正,结果为正的 */
ResultList = ComputePPNNList(
TempList, *(headBuff + iCounter), POSITIVE);
else if (ResultList->data < 0 &&
(*(headBuff + iCounter))->data < 0)/* 负负,结果为负的 */
ResultList = ComputePPNNList(
TempList, *(headBuff + iCounter), NEGATIVE);
else
{
if (ResultList->data + (*(headBuff + iCounter))->data == 0)
{ /* 异号时同样长 */
CurrNode_1 = ResultList;
CurrNode_2 = *(headBuff + iCounter);
do
{ /* 直到找到第一个不等值的结点 */
if (CurrNode_1->data > CurrNode_2->data)
{
iSign = (ResultList->data > 0) ?
POSITIVE : NEGATIVE;
ResultList = ComputePNList(
TempList, *(headBuff + iCounter), iSign);
```

```
break;
}
else if (CurrNode_1->data < CurrNode_2->data)
{
iSign = ((*(headBuff + iCounter))->data > 0) ?
POSITIVE : NEGATIVE;
ResultList = ComputePNList(
*(headBuff + iCounter), TempList, iSign);
break;
}
CurrNode_1 = CurrNode_1->next;
CurrNode_2 = CurrNode_2->next;
} while (CurrNode_1 != ResultList);
}
else if (fabs(ResultList->data) >
fabs((*(headBuff + iCounter))->data))
{
iSign = (ResultList->data > 0) ? POSITIVE : NEGATIVE;
ResultList = ComputePNList(
TempList, *(headBuff + iCounter), iSign);
}
else if (fabs(ResultList->data) <
fabs((*(headBuff + iCounter))->data))
{
iSign = ((*(headBuff + iCounter))->data > 0) ?
POSITIVE : NEGATIVE;
ResultList = ComputePNList(
*(headBuff + iCounter), TempList, iSign);
}

}
if (*headBuff > TempList || TempList > *(headBuff + iCounter))
DeleteList(TempList); /* 清除上次的中间链表 */
/* 删除多出的 0,如删除 0000,0010,3333 中的 0000 为 0010,3333*/
DeleteZero(&ResultList);
}
*ResultHead = ResultList;
}

/*
* 每次只处理两个操作数链表,符号相异时 List_1 为正的, List_2 为负的
* 如果两个操作数链表不一样长则 List_1 为长的结果链表的结构和操作
* 数链表一样, 最后返回结果链表
*/
LinkType ComputePNList(LinkType List_1, LinkType List_2, int iSign)
{
int iResult = 0, iBorrow = 0, iResultNodeNum = 0;
LinkType CurrNodeArray[2],
NewNode = NULL, ResultList = NULL;

/* 初始为每一个操作数链表的尾结点地址 */
CurrNodeArray[0] = (List_1)->prior;
CurrNodeArray[1] = (List_2)->prior;

while ((CurrNodeArray[0] != List_1)
|| (CurrNodeArray[1] != List_2))
{
```

```
if (iBorrow < 0) /* 处理前一位的借位 */
if (CurrNodeArray[0]->data > 0)
{
iBorrow = 0;
iResult = -1;
}
else if (CurrNodeArray[0]->data == 0)
{
iBorrow = -1; /* 继续向高位借 1 位 */
iResult = 9999;
}

if ((CurrNodeArray[0] != List_1)
&& (CurrNodeArray[1] != List_2))
{
if ((CurrNodeArray[0]->data < CurrNodeArray[1]->data)
&& iBorrow == 0)
{
iBorrow = -1; /* 不够减则向高位借 1 位 */
iResult += 10000;
}
iResult += CurrNodeArray[0]->data -
CurrNodeArray[1]->data;

CurrNodeArray[0] = CurrNodeArray[0]->prior;
CurrNodeArray[1] = CurrNodeArray[1]->prior;
}
else if (List_1 != CurrNodeArray[0]) /* 处理剩下的链表 */
{
iResult += CurrNodeArray[0]->data;
CurrNodeArray[0] = CurrNodeArray[0]->prior;
}

/* 将算好的结点加入结果链表 */
PushDataToList(&ResultList, iResult, POSITIVE);
iResultNodeNum++;
if ((CurrNodeArray[0] == List_1)
&& (CurrNodeArray[1] == List_2))
{
/* 在链表头插入控制结点 */
MakeNode(&NewNode, iResultNodeNum);
PushDataToList(&ResultList, iResultNodeNum, iSign);
}

iResult = 0; /* 准备计算下一个结点 */
}

return ResultList;
}

/* 每次只处理两个操作数链表,正正,结果为正的,负负,结果为负的 */
LinkType ComputePPNNList(LinkType List_1, LinkType List_2, int iSign)
{
int iResult = 0, iCarry = 0, iResultNodeNum = 0;
LinkType CurrNodeArray[2],
NewNode = NULL, ResultList = NULL;
```

```
/* 初始为每一个操作数链表的尾结点地址 */
CurrNodeArray[0] = (List_1)->prior;
CurrNodeArray[1] = (List_2)->prior;

while (TRUE)
{
if (iCarry > 0) /* 处理前一位的进位 */
{
iResult += iCarry;
iCarry = 0;
}

if (CurrNodeArray[0] != List_1 &&
CurrNodeArray[1] != List_2)
{
iResult += CurrNodeArray[0]->data + CurrNodeArray[1]->data;
CurrNodeArray[0] = CurrNodeArray[0]->prior;
CurrNodeArray[1] = CurrNodeArray[1]->prior;
}
else if (CurrNodeArray[0] != List_1)
{
iResult += CurrNodeArray[0]->data;
CurrNodeArray[0] = CurrNodeArray[0]->prior;
}
else if (CurrNodeArray[1] != List_2)
{
iResult += CurrNodeArray[1]->data;
CurrNodeArray[1] = CurrNodeArray[1]->prior;
}

if (iResult >= 10000)
{
iCarry = iResult / 10000;
iResult = iResult % 10000;
}

PushDataToList(&ResultList, iResult, POSITIVE);
iResultNodeNum++;
if (iCarry == 0 && CurrNodeArray[0] == List_1
&& CurrNodeArray[1] == List_2)
{
MakeNode(&NewNode, iResultNodeNum);
PushDataToList( &ResultList, iResultNodeNum, iSign);
break;
}

iResult = 0; /* 准备计算下一个结点 */
}

return ResultList;
}

/*
* 删除多出的 0,如删除 0000,0010,3333 中的 0000 为 0010,3333
* ,但链表为只有一个逻辑结点为 0 时则不删除.
*/
```

```
status DeleteZero(LinkType *List)
{
LinkType CurrNode, DelNode;

/*
*  一旦遇到第一个不为 0 的结点则退出，但
*  链表为只有一个逻辑结点为 0 时则不删除
*/
CurrNode = DelNode = (*List)->next;
while (fabs((*List)->data) > 1 && CurrNode->data == 0)
{
(*List)->next = CurrNode->next;
CurrNode->next->prior = *List;
DelNode = CurrNode;
CurrNode = CurrNode->next;
free(DelNode);
/*  控制结点减少一个逻辑结点的个数  */
(*List)->data += ((*List)->data > 0) ? -1 : 1;
}

return TRUE;
}

status PushDataToList(LinkType *head, int iNodeNum, int sign)
{
LinkType NewNode;

/* sign 为 1 时为正的, sign 为 0 时为负的  */
iNodeNum *= (sign == POSITIVE) ? 1 : -1;
MakeNode(&NewNode, iNodeNum);
if (*head != NULL)
{
/*  将 NewNode 所指的结点插入链表，使成为头结点  */
NewNode->next = *head;
NewNode->prior = (*head)->prior;
(*head)->prior = NewNode;
NewNode->prior->next = NewNode;
}
*head = NewNode;

return TRUE;
}

status AppendNodeToList(LinkType *head, LinkType NewNode)
{
static LinkType CurrNode = NULL;

if (*head == NULL)
*head = CurrNode = NewNode;
else
{
/*  在链表尾部添加结点  */
NewNode->next = CurrNode->next;
CurrNode->next = NewNode;
NewNode->prior = CurrNode;
NewNode->next->prior = NewNode;
```

```
/* 当前指针向前一步 */
CurrNode = CurrNode->next;
}

return TRUE;
}

status MakeNode(LinkType *p, ElemType e)
{
*p = (LinkType)malloc(sizeof(NodeType) * 1);
if (!(*p))
{
printf("Error, the memory is overflow!\n");
return FALSE;
}
(*p)->data = e;
(*p)->prior = (*p)->next = (*p);

return TRUE;
}

status PrintList(LinkType head)
{
/* LinkType CurrNode = head; use for debug */
LinkType CurrNode = head->next;

if (head == NULL) return FALSE;
if (head->data < 0) printf("-");
while (TRUE)
{
printf(" %04d", CurrNode->data);
CurrNode = CurrNode->next;
if (CurrNode == head) break;
printf("%c", ',');
}
printf("\n");

return TRUE;
}

status ClearMemory(LinkType *headBuff, int iOpNum)
{
int iCounter;

for (iCounter = 0; iCounter < iOpNum; iCounter++)
DeleteList(*(headBuff + iCounter));
free(headBuff);

return TRUE;
}

status DeleteList(LinkType head)
{
LinkType CurrNode;

if (head == NULL) return FALSE;
while (1)
```

```
{
CurrNode = head;
CurrNode->next->prior = CurrNode->prior;
CurrNode->prior->next = CurrNode->next;
if (head == head->next)
{
free(CurrNode);
break;
}
head = head->next;
free(CurrNode);
}

return TRUE;
}

/* 输入异常处理 */
status ErrorProcess(char strScrNum[], int iOpNum)
{
int iTemp = 0;
char *cpCurr;

if (!strlen(strScrNum))
{
printf("你没有输入数据!");
return FALSE;
}
for (cpCurr = strScrNum; *cpCurr != '\0'; cpCurr++)
{
if (!(*cpCurr == ' ' || *cpCurr == ',' || *cpCurr == ';'
|| *cpCurr == '-' || *cpCurr == '+'
|| ('0' <= *cpCurr && *cpCurr <= '9'))
|| (*(cpCurr + 1) == '\0' && *cpCurr != ';')
|| (*(cpCurr + 1) == '+' && *cpCurr == '-')
|| (*(cpCurr + 1) == '-' && *cpCurr == '+')
|| (*(cpCurr + 1) == '-' && *cpCurr == '-')
|| (*(cpCurr + 1) == '+' && *cpCurr == '+'))
{
printf("\n 出现非法输入 1!\n");
return FALSE;
}
if (*cpCurr == ';') iTemp++;
}
if (iTemp != iOpNum) return FALSE;

return TRUE;
}
```

插入排序(Inseretion Sort)

插入排序的基本思想是，经过 i-1 遍处理后,$a_1,a_2,..,a_{i-1}$ 已排好序。第 i 遍处理仅将 a_i 插入 $a_1,a_2,..,a_{i-1}$ 的适当位置，使得 $a_1,a_2,..,a_i$ 又是排好序的序列。要达到这个目的，我们可以用顺序比较的方法。首先比较 a_i 和 a_{i-1}，如果 $a_{i-1} \le a_i$，则 $a_1,a_2,..,a_i$ 已排好序，第 i 遍处理就结束了；否则交换 a_i 与 a_{i-1} 的位置，继续比较 a_{i-1} 和 a_{i-2}，直到找到某一个位置 $j(1 \le j \le i-1)$，使得 $a_j \le a_{j+1}$ 时为止。图 1 演示了对 4 个元素进行插入排序的过程。

图 1 对 4 个元素进行插入排序

在下面的插入排序算法中，为了写程序方便我们可以引入一个哨兵元素 a_0，它小于 $a_1,a_2,..,a_n$ 中任一记录。所以，我们设元素的类型 TElement 中有一个常量-∞，它比可能出现的任何记录都小。如果常量-∞不好事先确定，就必须在决定 a_i 是否向前移动之前检查当前位置是否为 1，若当前位置已经为 1 时就应结束第 i 遍的处理。另一个办法是在第 i 遍处理开始时，就将 a_i 放入 a_0 中，这样也可以保证在适当的时候结束第 i 遍处理。

例：

初始序列:	8	3	2	5	9	1	6
i=2	3	8	2	5	9	1	6
i=3	2	3	8	5	9	1	6
i=4	2	3	5	8	9	1	6
i=5	1	2	3	5	8	9	6
i=6	1	2	3	5	8	9	6
i=7	1	2	3	5	6	8	9

程序如下：

直接插入排序:

```
void docr(float *in,int count)
{
  int i,j,x;
  float temp;
  for(i=1;i<count;i++)
  {
    for(x=0;x<i;x++)
    {
      if((*(in+i))<=(*(in+x)))
      break;
    }
    temp=*(in+i);
    for(j=i;j>x;j--)
    {
      *(in+j)=*(in+j-1);
    }
    *(in+x)=temp;
  }
}
```

最小比较次数：n　最大比较次数：n*n/2

最小移动次数：2n　最大移动次数：n*n/2

二分法插入排序:(比较部分用二分法)常用算法

```
void docr(float *in,int count)
{
int i,j,x;
int l,r,m;
float temp;
for(i=1;i<count;i++)
    {
    l=0;
    r=i-1;
    temp=*(in+i);
    while(l<=r)
    {

        m=(int)((l+r)/2);
        if(temp<=(*(in+m)))
        {
            r=m-1;
        }
        else
        {
            l=m+1;
        }
    }
    for(j=i;j>l;j--)
    {
        *(in+j)=*(in+j-1);
    }
    *(in+l)=temp;
    }
}
```

移动次数: n*log2(n)

最小移动次数：2n　最大移动次数：n*n/2

程序设计：哈希表的一个应用

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
```

```
#define L 50 /*定义哈希表长*/
#define M 47   /*定义 p 值*/
#define N 30    /*定义名单长*/
char z[22];
struct old{char *name;char *py;int k;};
struct old oldlist[L];/*原始表*/
struct   hterm
{ char *name;char *py;
int k;int si;
};
struct hterm hlist[L];/*哈希表*/
int i,adr,sum,d;
char ch1;
float average;
/**********************************/
void chash()
{for (i=0;i<L;i++)
     {hlist[i].name="";
      hlist[i].py="";
      hlist[i].k=0;
      hlist[i].si=0;
     };
 for (i=0;i<N;i++)
     { sum=0;
     adr=(oldlist[i].k)%M;
     d=adr;
     if(hlist[adr].si==0)
       {hlist[adr].k=oldlist[i].k;
        hlist[adr].name=oldlist[i].name;
        hlist[adr].py=oldlist[i].py;
        hlist[adr].si=1;
       }
     else
       {do
   {d=(d+((oldlist[i].k))%10+1)%M;/*伪随机*/
          sum=sum+1;
         }
        while (hlist[d].k!=0);
        hlist[d].k=oldlist[i].k;
        hlist[d].name=oldlist[i].name;
        hlist[d].py=oldlist[i].py;
        hlist[d].si=sum+1;
```

```
      }
   }
}
/*************************************/
void findhlist()
{ int s0;char r,g;
  clrscr();/*清屏*/
  for (r=0;r<20;r++){z[r]=0;};
   gotoxy(1,1);printf("查找：copyright by 姚建飞 2003.6");
 gotoxy(5,10);printf("请拼音后回车！ ");
 gotoxy(5,12);scanf("%s",z);
      s0=0;
      for (r=0;r<20;r++){s0=z[r]+s0;};
  gotoxy(5,13);     printf("%d",s0);
 /*for (i=0;i<L;i++)*/
      sum=1;
     adr=s0%M;
    d=adr;
     if(hlist[adr].k==s0)
       {
        gotoxy(18,18);printf("               ");
             gotoxy(18,18);printf("%s",hlist[d].name);
             gotoxy(18,19);printf("%s",hlist[d].py);
               gotoxy(18,20);
                  printf("搜索 %d 次",sum);
          getch();
       }
     else
       {if (hlist[adr].k==0)
            {gotoxy (18,18);
             printf("无记录!                    ");
             getch();
            }
        else
           {g=0;
            for (i=0;g==0;i++)
          {d=(d+s0%10+1)%M;    /*伪随机*/
               sum=sum+1;
               if (hlist[d].k==0)
                  {gotoxy (18,18);
                   printf("无记录!                ");
                   g=1;getch();
```

```
                };
              gotoxy(18,18);
                printf("%s",hlist[d].name);
                gotoxy(18,19);
                printf("%s",hlist[d].py);
                gotoxy(18,20);
                printf("搜索  %d  次",sum);
                getch();
              if (hlist[d].k==s0)
                {    g=1;
                 gotoxy(18,21);
                printf("搜索  %d  次成功!",sum);
                 getch();
                };
              };
              };
      };
}

/****************************************/
void inp() /*输入表*/
{
 char *f;
 int r,s0;
 oldlist[0].name="桂芳芳";oldlist[0].py="guifanfan";
 oldlist[1].name="姚建飞";oldlist[1].py="yaojianfei";
 oldlist[2].name="杨扬";oldlist[2].py="yangyang";
 oldlist[3].name="朱玉环";oldlist[3].py="zhuyuhuang";
 oldlist[5].name="陈曦";oldlist[5].py="chenxi";
 oldlist[6].name="张雷";oldlist[6].py="zhanglei";
 oldlist[7].name="盛永海";oldlist[7].py="shenyonghai";
 oldlist[8].name="陈道全";oldlist[8].py="chengdaoquan";
 oldlist[9].name="陆道清";oldlist[9].py="ludaoqing";
 oldlist[10].name="龚云祥";oldlist[10].py="gongyunxiang";
 oldlist[11].name="孙振兴";oldlist[11].py="sunzhenxing";
 oldlist[12].name="孙容飞";oldlist[12].py="sunrongfei";
 oldlist[13].name="孙明龙";oldlist[13].py="sunminglong";
 oldlist[14].name="张浩";oldlist[14].py="zhanghao";
 oldlist[15].name="田苗";oldlist[15].py="tianmiao";
 oldlist[16].name="姚建中";oldlist[16].py="yaojianzhong";
 oldlist[17].name="姚建清";oldlist[17].py="yaojianqing";
 oldlist[18].name="姚建华";oldlist[18].py="yaojianhua";
```

```
 oldlist[19].name="张海峰";oldlist[19].py="yaohaifeng";
 oldlist[20].name="陈言号";oldlist[20].py="chengyanhao";
 oldlist[21].name="姚秋锋";oldlist[21].py="yaoqiufeng";
 oldlist[22].name="钱鹏程";oldlist[22].py="qianpengcheng";
 oldlist[23].name="姚海峰";oldlist[23].py="yaohaifeng";
 oldlist[24].name="卞艳";oldlist[24].py="bianyan";
 oldlist[25].name="凌蕾";oldlist[25].py="linglei";
 oldlist[26].name="李伟";oldlist[26].py="liwei";
 oldlist[27].name="黄海燕";oldlist[27].py="huanhaiyan";
 oldlist[28].name="刘殿琴";oldlist[28].py="liudianqin";
 oldlist[29].name="李云";oldlist[29].py="liyun";
/*
请在此输入数据,同时修改程序开头的 M   L N


*/
for (i=0;i<N;i++)
     {
 s0=0;
        f=oldlist[i].py;
        for (r=0;*(f+r) != '\0';r++){s0=*(f+r)+s0;};
        oldlist[i].k=s0;

     };
}

/*****************************************/
void   dhash() /*显示哈希表*/
{   char LON=17;
   clrscr();
   if (LON>L){LON=L;};
   gotoxy(1,1);printf("哈希表：copyright by 姚建飞 2003.6");
   gotoxy(1,2);printf("地址:");
   for(i=0;i<LON;i++)
     {gotoxy(1,i+3);
      printf("%-3d",i);
     };
   gotoxy(9,2);printf("关键字:");
   for(i=0;i<LON;i++)
     {gotoxy(10,i+3);
      printf("%-6d",hlist[i].k);
     };
```

```
 gotoxy(19,2);printf("姓名:");
 for(i=0;i<LON;i++)
    {gotoxy(19,3+i);
     printf("%s",hlist[i].name);
    };
gotoxy(28,2);printf("拼音:");
 for(i=0;i<LON;i++)
    {gotoxy(28,i+3);
     printf("%s",hlist[i].py);
    };
 gotoxy(40,2);printf("搜索长度:");
 for(i=0;i<LON;i++)
    {gotoxy(43,i+3);
     printf("%2d",hlist[i].si);
    };
gotoxy(53,2);printf("H(key):");
 for(i=0;i<LON;i++)
    {gotoxy(53,i+3);
     printf("%2d",(hlist[i].k)%M);
    };
 average=0;
 for (i=0;i<L;i++)
     {average=average+hlist[i].si;};
 average=average/N;
 gotoxy(10,23);
printf("平均搜索长度：ASL(%d)=%f",N,average);
gotoxy(20,24);
 printf("任意键下一屏!");
 ch1=getch();

  if (L>15)
  {
 clrscr();
 if (LON>L-15){LON=L-15;};
  gotoxy(1,1);printf("哈希表：copyright by 姚建飞 2003.6");
  gotoxy(1,2);printf("地址:");
  for(i=0;i<LON;i++)
    {gotoxy(1,i+3);
     printf("%-3d",i+15);
    };
  gotoxy(9,2);printf("关键字:");
  for(i=0;i<LON;i++)
```

```
    {gotoxy(10,i+3);
     printf("%-6d",hlist[i+15].k);
    };
  gotoxy(19,2);printf("姓名:");
  for(i=0;i<LON;i++)
    {gotoxy(19,3+i);
     printf("%s",hlist[i+15].name);
    };
  gotoxy(28,2);printf("拼音:");
  for(i=0;i<LON;i++)
    {gotoxy(28,i+3);
    printf("%s",hlist[i+15].py);
    };
  gotoxy(40,2);printf("搜索长度:");
  for(i=0;i<LON;i++)
    {gotoxy(43,i+3);
     printf("%2d",hlist[i+15].si);
    };
  gotoxy(53,2);printf("H(key):");
  for(i=0;i<LON;i++)
    {gotoxy(53,i+3);
     printf("%2d",(hlist[i+15].k)%M);
    };
  average=0;
  for (i=0;i<L;i++)
     {average=average+hlist[i].si;};
  average=average/N;
  gotoxy(10,23);
  printf("平均搜索长度：ASL(%d)=%f",N,average);
  gotoxy(20,24);
  printf("任意键下一屏!   ");
  ch1=getch();
   };
  if (L>30)
  {
 clrscr();
 if (LON>L-30){LON=L-30;};
  gotoxy(1,1);printf("哈希表：copyright by 姚建飞 2003.6");
  gotoxy(1,2);printf("地址:");
  for(i=0;i<LON;i++)
    {gotoxy(1,i+3);
     printf("%-3d",i+30);
```

```
   };
gotoxy(9,2);printf("关键字:");
for(i=0;i<LON;i++)
   {gotoxy(10,i+3);
    printf("%-6d",hlist[i+30].k);
   };
gotoxy(19,2);printf("姓名:");
for(i=0;i<LON;i++)
   {gotoxy(19,3+i);
    printf("%s",hlist[i+30].name);
   };
gotoxy(28,2);printf("拼音:");
for(i=0;i<LON;i++)
   {gotoxy(28,i+3);
   printf("%s",hlist[i+30].py);
   };
gotoxy(40,2);printf("搜索长度:");
for(i=0;i<LON;i++)
   {gotoxy(43,i+3);
    printf("%2d",hlist[i+30].si);
   };
gotoxy(53,2);printf("H(key):");
for(i=0;i<LON;i++)
   {gotoxy(53,i+3);
    printf("%2d",(hlist[i+30].k)%M);
   };
average=0;
for (i=0;i<L;i++)
    {average=average+hlist[i].si;};
average=average/N;
gotoxy(10,23);
printf("平均搜索长度：ASL(%d)=%f",N,average);
gotoxy(20,24);
printf("任意键下一屏!   ");
ch1=getch();
};
if (L>45)
{
clrscr();
if (LON>L-45){LON=L-45;};
gotoxy(1,1);printf("哈希表：copyright by 姚建飞 2003.6");
gotoxy(1,2);printf("地址:");
```

```
    for(i=0;i<LON;i++)
       {gotoxy(1,i+3);
        printf("%-3d",i+45);
       };
    gotoxy(9,2);printf("关键字:");
    for(i=0;i<LON;i++)
       {gotoxy(10,i+3);
        printf("%-6d",hlist[i+45].k);
       };
    gotoxy(19,2);printf("姓名:");
    for(i=0;i<LON;i++)
       {gotoxy(19,3+i);
        printf("%s",hlist[i+45].name);
       };
    gotoxy(28,2);printf("拼音:");
    for(i=0;i<LON;i++)
       {gotoxy(28,i+3);
       printf("%s",hlist[i+45].py);
       };
    gotoxy(40,2);printf("搜索长度:");
    for(i=0;i<LON;i++)
       {gotoxy(43,i+3);
        printf("%2d",hlist[i+45].si);
       };
    gotoxy(53,2);printf("H(key):");
    for(i=0;i<LON;i++)
       {gotoxy(53,i+3);
        printf("%2d",(hlist[i+45].k)%M);
       };
    average=0;
    for (i=0;i<L;i++)
       {average=average+hlist[i].si;};
    average=average/N;
    gotoxy(10,23);
    printf("平均搜索长度：ASL(%d)=%f",N,average);
    gotoxy(20,24);
    printf("任意键返回!   ");
    ch1=getch();
    };
}
/***************************************/
void main()
```

```
    {inp();   /*输入原表*/
     chash ();/*建哈希表*/
  a: clrscr();
     gotoxy(21,2);
     textcolor(GREEN);
     cprintf("欢迎使用本程序----------编者:姚建飞");
     printf("\n");
     gotoxy(22, 4);
     textcolor(GREEN);
     cprintf("    1. 显示哈希表");
     printf("\n");
     gotoxy(22, 6);
     textcolor(GREEN);
     cprintf("    2. 查找");
     printf("\n");
     gotoxy(22, 8);
     textcolor(GREEN);
     cprintf("    x. 退出");
     printf("\n");
     gotoxy(22, 12);
     cprintf("  请输入选择: ");
     printf("\n");
     gotoxy(24,14);
     ch1=getch();
     if (ch1==0x78){ textcolor(GREEN);
     cprintf("谢谢使用本程序,你已经退出本程序!");printf("\n"); exit();};/*"x":退出*/
     if (ch1==0x31){dhash();};/*表的属性*/
     if (ch1==0x32){ findhlist();};/*查找*/
       goto a;
     }
```

多维数组下标操作符重载一法

刚学习 C++操作符重载时，我就想过如何重载多维下标操作符“[][]”的问题，这里仅就二维的特殊情况提出一种方法，希望可以起到抛砖引玉的作用。

我们假设有一个 Matrix 类用来封装矩阵，并希望用[][]来取得矩阵对应位置的元素值。分析一下 m.[a][b]，可以看做是 m.operator [a].operator [b]。所以要让第一个[]返回一个也重载了[]的辅助类，再把 a 传到那个辅助类中，想办法在后一个类的[]中实现对应元素的返回。代码如下（忽略了其他成员函数和越界检查）：

```
class assis//辅助类
{
```

```
public:
int operator [](int index)
{
index2=index-1;
return pdata[index1*ncol+index2];
};
int * pData;
int index1,index2;
int ncol,nrow;
};
class Matrix//矩阵类
{
public:
...//其他成员函数
assis operator [](int index)
{
assis ret;
ret.pData=pData;
ret.ncol=ncol;
ret.nrow=nrow;
ret.index1=index-1;
return ret;
};
private:
int * pData;
int nrow,ncol;
...//其他成员变量
};
```

如果哪位读者有其他更好的办法，请来信与我交流。

汉诺塔的非递归（演示、动画）

Hanoi,非递归,演示,动画效果

kensta

有动画演示，move2()是标准解的调用

move()是用于演示动画或显示移动秩序和包含监测有无错误移动的调用

使用 Borland c++ 3.0(Turbo c++ 3.0,Turbo c 2.0 也可)编译通过，图形方式使用 Tc 的 bgi

```
*/
/************************************/
/*
```

```
 about error process
*/
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
/*
 if debuging use #define ERROR_DEBUG
 otherwise remove it.
*/
//#define ERROR_DEBUG
#ifdef  ERROR_DEBUG
 #define error(x) error_debug(x)
 #define report() report_debug()
 #define initerror() initerror_debug()
char *err[10];
int errs=0;
void initerror_debug(){
 int i;
 for(i=0;i<10;i++)err[i]=NULL;
}
void error_debug(char *a){
 if(errs>9)return;
 err[errs]=(char *)malloc(strlen(a)+1);
 strcpy(err[errs],a);
 printf(a);
 errs++;
}
void report_debug(){
 int i;
 if(!errs)return;
 for(i=0;i<errs;i++){
  printf(err[i]);
  free(err[i]);
 }
}
#else
 #define error(x)
 #define report()
 #define initerror()
#endif
/*************************************/
/*
```

```
 about stack
*/
#define STACK_SIZE 31
typedef struct {
 int data[STACK_SIZE];
 int top;
}stack;
int clear(stack *a);
int create(stack **a);
int push(stack *a,int data);
int pop(stack *a,int *data);
int gettop(stack *a,int *data);
int dispose(stack *a);
int pop(stack *a,int *data){
 if(a->top){
  *data=a->data[--a->top];
  return 1;
 }else{
  error("pop(stack *,int *):stack empty!\n");
  return 0;
 }
}
int push(stack *a,int data){
 if(a->top<STACK_SIZE){
  a->data[a->top++]=data;
  return 1;
 }else {
  error("push(stack *,int):stack full!\n");
  return 0;
 }
}
int create(stack **a){
 *a=(stack *)malloc(sizeof(stack));
 if(*a)return clear(*a);
 else{
  error("create(stack **):create error! Not enough momery!\n");
  return 0;
 }
}
int clear(stack *a){
 if(a){
  a->top=0;
```

```
 return 1;
 }else {
  error("clear(stack *):stack not exist!\n");
  return 0;
 }
}
int gettop(stack *a,int *data){
 if(a->top){
  *data=a->data[a->top-1];
  return 1;
 }else{
  error("gettop(stack *,int *):stack empty!\n");
  return 0;
 }
}
int dispose(stack *a){
 if(a){
  free(a);
  return 1;
 }else{
  error("dispose(stack *):stack not exist!\n");
  return 0;
 }
}
/*************************************/
/*
 about Hanoi the game
*/
#include <graphics.h>
#include <dos.h>
#define MAX_LEVEL STACK_SIZE
int position[MAX_LEVEL+1];
stack *theStack[3];
int depth;
int mode;
int print;
int initgame(int d){
 int i;
 int x,y;
 int h=5;
 int w;
 initerror();
```

```
 if(mode){
  int gdriver = DETECT, gmode, errorcode;
  /* initialize graphics mode */
  initgraph(&gdriver, &gmode, "");
  setfillstyle(1,7);
 }
 for(i=0;i<3;i++)
  if(!create(&theStack[i]))
   break;
 if(i!=3){
  for(;i>=0;i--)dispose(theStack[i]);
  error("initgame(int):can not init stack!\n");
  return 0;
 }
 depth=d;
 for(i=d;i;i--){
  push(theStack[0],i);
  if(mode){
   y=200+100-theStack[0]->top*(h+1);
   w=i*10;
   x=150-w/2;
   setcolor(i);
   setfillstyle(1,i);
   bar(x,y,x+w,y+h);
  }
  position[i]=0;
 }
 if(mode){
  setcolor(15);
  for(i=0;i<3;i++)
   rectangle(150+i*150-1,120,150+i*150+1,300);
  line(50,300,500,300);
 }
 return 1;
}
int endgame(){
 int i=2;
 for(;i>=0;i--)dispose(theStack[i]);
 printf("report:");
 report();
 if(mode)closegraph();
 return 1;
```

```
}
void show(int p,int from,int to){
 int i;
 int x,y;
 int newx,newy;
 int h=5;
 int w=p*10;
 y=200+100-(theStack[from]->top+1)*(h+1);
 x=from*150+150-w/2;
 newx=to*150+150-w/2;
 newy=200+100-theStack[to]->top*(h+1);
 while(y>100){
  setcolor(0);
  setfillstyle(1,0);
  bar(x,y,x+w,y+h);
  y-=(h+1);
  setcolor(15);
  rectangle(150+from*150-1,120,150+from*150+1,300);
  setcolor(p);
  setfillstyle(1,p);
  bar(x,y,x+w,y+h);
  delay(10);
 }
 while(x!=newx){
  setcolor(0);
  setfillstyle(1,0);
  bar(x,y,x+w,y+h);
  (x>newx)?x--:x++;
  setcolor(p);
  setfillstyle(1,p);
  bar(x,y,x+w,y+h);
  delay(2);
 }
 while(y<newy){
  setcolor(0);
  setfillstyle(1,0);
  bar(x,y,x+w,y+h);
  setcolor(15);
  rectangle(150+to*150-1,120,150+to*150+1,300);
  y+=(h+1);
  setcolor(p);
  setfillstyle(1,p);
```

```
  bar(x,y,x+w,y+h);
  delay(10);
 }
}
int move(int p){
 int t,s;
 if(!gettop(theStack[position[p>,&t)){
  error("move(int):the stack is empty\n");
  return 0;
 }
 if(t==p){
  pop(theStack[position[p>,&t);
  if(!mode&&print)printf("%c -> ",'A'+position[p]);
  /* another important core line */
  s=(position[p]+1+(depth%2?p%2:(p+1)%2) )%3;
  if(gettop(theStack[s],&t)&&t<p){
   error("move(int):can not move big level above small one\n");
   return 0;
  }
  push(theStack[s],p);
  if(mode)show(p,position[p],s);
  else if(print)printf("%c\t",'A'+s);
  position[p]=s;
 }else error("move(int):position error\n");
 return 1;
}
int move2(int p){
 int t,s;
 s=(position[p]+1+(depth%2?p%2:(p+1)%2) )%3;
 if(print)printf("%c->%c\t",'A'+position[p],'A'+s);
 position[p]=s;
 return 1;
}
#include <conio.h>
void main(){
 unsigned long i;
 unsigned long N=10;
 unsigned long p,q;
 printf("Welcome to Hanoi\n");
 printf("Note that this Hanoi is not write by recurrence!\n");
 printf("And not calculate with any stack.\n");
 printf("but i want to check if the a is right.\n");
```

```
printf("i use 3 stack to show if there is any violent move happens.:)\n");
printf("\nEnter a number as level(1 to 30):");
scanf("%d",&N);
if(N<1||N>30){
 printf("error: not 1 to 30\n");
 return;
}
printf("\n Select show mode('c' in TEXT 'g' in GRAPHICS)\n");
printf("Note that if the level is to big you'd better not use 'g' for speed.\n");
printf("19 is about 20 seconds. 20 is about double of that. etc.\n");
printf("I test on a intel 166mmx cpu. 30 may be 40*1024 seconds.\n");
printf("wish you succeed!\n");
switch(getch()){
 case 'c':
  printf("do you want to show the result?(y/n)\n");
  printf("print result will be slow!!!\n");
  do{
   mode=getch();
   if(mode=='y')print=1;
   if(mode=='n')print=0;
  }while(mode!='y'&&mode!='n');
  mode=0;
  break;
 case 'g':mode=1;break;
 default:printf("error: neither 'c' nor 'g'\n");return;
}
printf("processing...\n");
initgame(N);
/*
 core here!!!
 only 8 lines, ha!
 here get the level queue
 as 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
*/
for(i=1;i<(1L<<N);i++){
 q=1L<<N;p=N+1;
 while(q&&i%q){
  q>>=1;
  p--;
 }
 if(mode||print)move(p);
 else move2(p);
```

```
 }
 printf("ok\n");
 getch();
 endgame();
}
```

回朔法一例

关键字 回溯法例题
原作者姓名 wjiang_dt(王大)
介绍
这道是用来填 3*3 方格的，要求相邻两个空个数字之和为质数
99 年的题，邮电的书上有解释
正文

```
/**************************
* 在 9（3*3）个方格的方阵中填入数字 1 到 N(N>=10)内的某 9 个数字
* 每个方格填一个整数，要求两个方格的两个整数之和为质数。
* 试求所有的解
* 回溯法例题
**************************/
#include <stdio.h>
#define N 12
void write(int a[]) /*输出满足条件的结果*/
{
  int i,j;
  for(i=0;i<3;i++）
  {
    for(j=0;j<3;j++)
      printf("%3d",a[3*i+j]);
      printf("\n");
  }
  scanf("%*c");
}
int b[N+1];
/*b 是一个数组，下标是 0 到 N，其中 b[x] (其中 1<=x<=N)的值表示数 x 是否已填
入方阵 a 当中*/
int a[10];  /*a 是一个数组，下标是 0~9，其中 a[0]~a[8]存放存方阵的值*/
int isPrime(int m)  /*判断是否素数*/
{
  int i;
```

```
  int primes[]={2,3,5,7,11,13,17,19,23,29,-1};
  if(m==1||m%2==0) return 0;
  for(i=0;primes[i]>0;i++)
    if(m==primes[i]) return 1;
  for(i=3;i*i<=m;)
  {
    if(m%i==0) return 0;
    i+=2;
  }
  return 1;
}
int checkMatrix[][3]={
                    {-1},{0,-1},{1,-1},{0,-1},{1,3,-1},{2,4,-1},{3,-1},{4,6,-1},{5,7,-1}
                  };
int selectNum(int start)  /**/
{/*1~N 这些数字只能被使用一次（在方阵中只能出现一次），假设数字 6 被使用了，则 b[6]=0*/
  int j;
  for(j=start;j<=N;j++)
    if(b[j]) return j;
  return 0;/*start~N 都已被使用了，b[start]~b[N]都为 0，说明无值可选，*/
}
int check(int pos)  /*检查数字是否满足条件*/
{
  int i,j;
  if(pos<0) return 0;
  for(i=0;(j=checkMatrix[pos][i]>=0;i++)
    if(!isPrime(a[pos]+a[j])) return 0;
  return 1;
}
int extend(int pos)
{
a[++pos]=selectNum(1); /*对限制有何作用？？？？？？？？？？*/
b[a[pos>=0;
return pos;
}
int change(int pos)//更改 a[pos]的值，
{
  int j;
  while(pos>=0&&(j=selectNum(a[pos]+1))==0)/*selectNum(a[pos]+1))  ==0 说明无值可选择*/
  b[a[pos-->=1;  /*a[pos]被放弃，该值还可以被使用，所以 b[a[pos>=1，然后 pos--*/
```

```
  if(pos<0) return -1;
  b[a[pos>=1;
  a[pos]=j;
  b[j]=0;
  return pos;
}
void find()   /*其中的变量控制不是很了解*/
{
  int ok=1;
  int pos=0;
  a[pos]=1; /* a[0]取 1*/
  b[a[pos>=0;/*b[1]取 0，由于 1 被使用了，所以 b[1]就为 0，表示不可以再次被使用*/
  do{
     if(ok)
     {
       if(pos==8)/*a[0]~a[8]都选好了数，即方阵已经建好*/
       {
         write(a);/*将数组 a 中所有值打出来*/
         pos=change(pos);/*从 1~N 中为 a[8]选择下一个值，如果无值  可选，则为 a[7]选择下一个值，如果 a[7]无值可选，则为 a[6]选择下一个值，直到 pos<0，如果 pos<0，说明所有可能的情况都已经尝试过了，*/
       }
       else
       pos=extend(pos);/*由于 a[0]~a[pos]都是符合要求的，这里继续为 a[pos+1]选择值，进行试探*/
     }
     else
     pos=change(pos);//由于 a[pos]不符合要求，则为 a[pos]选择其它值
     ok=check(pos);//检查重选的 a[pos]是否符合要求
   }
   while(pos>=0);/* change 函数的返回值传给 pos，如果 pos<0，说明所有可能的情况都已经尝试过了，结束程序。*/
}
void main()
{
  int i;
  for(i=1;i<=N;i++) b[i]=1;//初始化 b，1~N 都可以选
  find();
}
```

几道有趣的算法题

。

给定一个 N 进制正整数，把它的各位数字上数字倒过来排列组成一个新数，然后与原数相加，如果是回文数则停止，如果不是，则重复这个操作，直到和为回文数为止。

如果 N 超过 10，使用英文字母来表示那些大于 9 的数码。例如对 16 进制数来说，用 A 表示 10，用 B 表示 11，用 C 表示 12，用 D 表示 13，用 E 表示 14，用 F 表示 15。

例如：10 进制 87 则有：

STEP1: 87+78=165

STEP2: 165+561=726

STEP3: 726+627=1353

STEP4: 1353+3531=4884

编写一个程序，输入 N(2<=N<=16)进制数 M（1<=M<=30000(10 进制)）,输出最少经过几步可以得到回文数。如果在 30 步以内（含 30 步）不可能得到回文数，则输出 0。输入的数保证不为回文数。

【输入】

第一行一个整数 L，代表输入数据的组数。

接下来 L 行，每行两个整数 N,M

【输出】

输出 L 行，对于每个数据组，按题目要求输出结果，并占一行。

【样例输入】

2

10 87

2 110

【样例输出】

4

1

B 题 恺撒的规划

【问题描述】

亚特兰蒂斯是一块富饶美丽的土地。恺撒大帝率领他的大军，经过了一整年的浴血奋战，终于将它纳入了罗马帝国的版图。然而，长期的战火彻底抹去了这里的繁华，昔日的富庶之地如今一片荒芜。恺撒大帝作为一位有着雄才大略的君主，决心在战争的废墟上建起一座更为宏伟的城市。所以，在建城之前，他需要对整个城市进行规划。

亚特兰蒂斯是一块矩形平原，恺撒准备在上面修建一些建筑。为了规划方便，他将矩形划分成 N*M 格。棘手的是，部分古老的神庙残存下来，散布在某些格子内。亚特兰蒂斯的原住民原本就十分信奉神灵，而这些经过战火洗礼的神庙更是被他们视为圣物，是万万不能拆除的，否则将激起民愤，甚至引发暴动。恺撒深知这一点，因此，他的新建筑在选址时要避开这些神庙。

假设新的建筑物有 P 种规格，每种建筑物都是正方形的，占地为 Ti Ti 格 (1<=i<=P)。恺撒想知道对于每种规格的建筑，有多少种不同的合适选址方案（一种合适的选址方案指的是在该建筑所占的正方形区域内不存在神庙）。作为他的内务部长，这个光荣而艰巨的任务自然交给你来完成。

【输入】

输入文件第一行包含三个数，分别代表 N,M,P (1<=N,M<=100,1<=P<=100)。随后的 n 行，每行有 m 个 0 或 1（1 表示该格为废墟，0 表示该格有神庙）。接下来的 P 行每行有一个整数

(1< <=max(M,N))，代表的第 i 种建筑物的边长。

【输出】

输出文件有 P 行，每行一个整数，第行的数代表边长为 的建筑物选址方案数。

【样例输入】

4 4 2
1011
1111
1110
1110
2
3

【样例输出】

5
1

C 题　车 站

【问题描述】

辖区内新开了一条高速公路，公路上有两个车站，坐标分别为 A（xa，ya）、B（xb，yb），每天都有车辆从 A 站开往 B 站。公路附近有两个村庄（公路可能从村庄中穿过），村庄分布在如图所示的带状区域内，坐标为 C（xc，yc），D（xd，yd），C、D 两村每天都分别有 m 人要前往 B 站。

因为高速公路不可随意出入，所以需要在两车站之间的公路上合理地设置一些汽车停靠点，村民可步行至停靠点后进入高速公路，并免费乘车前往 B 站。每个村民每步行一千米（一个单位看作一千米）所得到的政府补贴为 t 元，政府维护一个停靠点所需花费为 p 元/年。应如何设置这些停靠点，才能使政府的支出最小？

给出一个年份 year，请你设计一个方案，使得镇政府从该年起的 n 年内总支出最小，注意考虑闰年情况。

注意，村民只能进入停靠点而不能直接进入车站，但允许在车站处设置停靠点。

【输入】

第一行四个数： xa ya xb yb
第二行四个数： xc yc xd yd
第三行四个数： m n t p （0<=3000，0<=10）
第四行一个数： year （2000<year<3000）
以上数字，m，year，n 为正整数，p，t 为正实数，其余均为实数。

【输出】

第一行 最小费用 c（单位：元）
第二行 设置的停靠点数 N（N 为正整数）
以下 N 行，每行两个实数，代表停靠点的坐标
如有多解，任意输出一解即可。
所有实数保留四位小数。

【样例输入】

-5 0 5 0
-1 -1 1 1
1 1 1 500

2001
【样例输出】
1532.3759
1
0.0000 0.0000

阶梯问题的递归解法

```
#include <stdio.h>
#define N 10
int steped[N];
int i=0;
void steping(int n){                    \\走楼梯
     if(n==0){                          \\已走完
         for(int j=0;j<i;j++){
              printf(" %d ",steped[j]);      \\打印
         }
     printf("\n");
     }
     if(n>=1){                          \\若剩下的阶梯数大等于 1
         steped[i++]=1;                 \\迈 1 个阶梯
         steping(n-1);                  \\走剩下的阶梯
         i--;                           \\退一个阶梯,寻找其它上法
     }
     if(n>=2){                          \\若剩下的阶梯数大等于 2
         steped[i++]=2;待                \\迈两个阶梯
         steping(n-2);                  \\走剩下的阶梯
         i--;                           \\退两个阶梯,寻找其它上法
    }
    if(n>=3){                           \\同上.....略
        steped[i++]=3;
        steping(n-3);
        i--;
    }
}
void main(){
     int n;
     n=N;
     steping(n);
}
```

“迭代法”也称“辗转法”是一种不断用变量的旧值递推新值的过程。迭代法又分为精确迭代和近似迭代。“二分法”和“牛顿迭代法”，这两种属于“近似迭代法”。在这里我们用的是精确迭代法求两正整数的最大公约数。以后我会讲到“近似迭代法”求二元方程。

求两正整数的最大公约数

原理：1.比较两数的大小，用大数除以小数，将得到一个余数；2.再将小数作为先前的大数，余数作为先前的小数，再重复第一步；3.直到所得的余数为 0 时停止，那么我们得到的最后那一个余数就是他们的最大公约数。

模型：假设两正整数为 m 和 n，且 m>n.

u=m,v=n

当 r=m/n 不为 0 时，则有：

u=v；

v=r；

当 r 为 0 时，输出 v。

当然这不是求最大公约数的唯一方法，还可以用穷举法等，这里我就不多说了。

例如下：(C++)

```
#include<iostream.h>
main()
{
 int u,v,r;
 cout<<"Enter two positive integers:";
 cin>>m>>n;
 if(m>n)
    u=m;
 else
    v=n;
 if(u*v!=0)
 {
  while(r=u%v)
  {
   u=v;
   v=r;
```

```
        }
        cout<<"The gcd is:"<<v<<endl;
        }
        else cout<<"Divided by zero!"<<endl;
    }
```

矩阵求逆的快速算法

算法介绍

矩阵求逆在 3D 程序中很常见，主要应用于求 Billboard 矩阵。按照定义的计算方法乘法运算，严重影响了性能。在需要大量 Billboard 矩阵运算时，矩阵求逆的优化能极大提高性能。这里要介绍的矩阵求逆算法称为全选主元高斯-约旦法。

高斯-约旦法（全选主元）求逆的步骤如下：

首先，对于 k 从 0 到 n - 1 作如下几步：

从第 k 行、第 k 列开始的右下角子阵中选取绝对值最大的元素，并记住次元素所在的行号和列号，在通过行交换和列交换将它交换到主元素位置上。这一步称为全选主元。

m(k, k) = 1 / m(k, k)

m(k, j) = m(k, j) * m(k, k)，j = 0, 1, ..., n-1；j != k

m(i, j) = m(i, j) - m(i, k) * m(k, j)，i, j = 0, 1, ..., n-1；i, j != k

m(i, k) = -m(i, k) * m(k, k)，i = 0, 1, ..., n-1；i != k

最后，根据在全选主元过程中所记录的行、列交换的信息进行恢复，恢复的原则如下：在全选主元过程中，先交换的行（列）后进行恢复；原来的行（列）交换用列（行）交换来恢复。

实现(4 阶矩阵)

```
float Inverse(CLAYMATRIX& mOut, const CLAYMATRIX& rhs)
{
CLAYMATRIX m(rhs);
DWORD is[4];
DWORD js[4];
float fDet = 1.0f;
int f = 1;
for (int k = 0; k < 4; k ++)
{
// 第一步，全选主元
float fMax = 0.0f;
for (DWORD i = k; i < 4; i ++)
{
for (DWORD j = k; j < 4; j ++)
{
const float f = Abs(m(i, j));
if (f > fMax)
{
fMax = f;
is[k] = i;
js[k] = j;
```

```
}
}
}
if (Abs(fMax) < 0.0001f)
return 0;

if (is[k] != k)
{
f = -f;
swap(m(k, 0), m(is[k], 0));
swap(m(k, 1), m(is[k], 1));
swap(m(k, 2), m(is[k], 2));
swap(m(k, 3), m(is[k], 3));
}
if (js[k] != k)
{
f = -f;
swap(m(0, k), m(0, js[k]));
swap(m(1, k), m(1, js[k]));
swap(m(2, k), m(2, js[k]));
swap(m(3, k), m(3, js[k]));
}
// 计算行列值
fDet *= m(k, k);
// 计算逆矩阵
// 第二步
m(k, k) = 1.0f / m(k, k);
// 第三步
for (DWORD j = 0; j < 4; j ++)
{
if (j != k)
m(k, j) *= m(k, k);
}
// 第四步
for (DWORD i = 0; i < 4; i ++)
{
if (i != k)
{
for (j = 0; j < 4; j ++)
{
if (j != k)
m(i, j) = m(i, j) - m(i, k) * m(k, j);
}
}
```

```
}
// 第五步
for (i = 0; i < 4; i ++)
{
if (i != k)
m(i, k) *= -m(k, k);
}
}
for (k = 3; k >= 0; k --)
{
if (js[k] != k)
{
swap(m(k, 0), m(js[k], 0));
swap(m(k, 1), m(js[k], 1));
swap(m(k, 2), m(js[k], 2));
swap(m(k, 3), m(js[k], 3));
}
if (is[k] != k)
{
swap(m(0, k), m(0, is[k]));
swap(m(1, k), m(1, is[k]));
swap(m(2, k), m(2, is[k]));
swap(m(3, k), m(3, is[k]));
}
}
mOut = m;
return fDet * f;
}
```

比较
原算法 原算法(经过高度优化) 新算法
加法次数 103 61 39
乘法次数 170 116 69
需要额外空间 16 * sizeof(float) 34 * sizeof(float) 25 * sizeof(float)
结果不言而喻吧。

快 速 排 序

基本思路：通过一次分割，将无序序列分成两部分，其中一部分的元素值均不大于后一部分的元素值。然后用同样的方法对每一部分进行分割，一直到每一个子序列的长度小于或等于 1 为止。

具体过程：设序列 P 进行分割。首先，从第一个、中间一个、最后一个元素中选出中项，设为 P[k]，并将 P[k]赋给 t，再将序列中的第一个元素移到

P[k]的位置上。然后，再设两个指针 i、j 分别指向起始和最后位置上：（1）将 i 逐渐增大，与此同时比较 P[i]与 t 的大小，直到发现 P[i]>t，将 P[i]移到 P[j]的位置上；（2）将 j 逐渐减小，与此同时比较 P[j]与 t 的大小，直到发现 P[j]<t，将 P[j]移到 P[i]的位置上；直到 i=j 为止，这时将 t 移到 P[i]的位置上。最后就得到了排序结果。

C 函数如下：

```
void prqck(p,n)
int n;double p[];
{
 int m,i0,*i;
 double *s;
 void rsplit();
 i=&i0;
 if(n>1)
   {
    rsplit(p,n,i);
    m=i0;prqck(p,m);
    s=p+(i0+1);m=n-(i0+1);prqck(s,m);
    }
 return;
}
static void rsplit(p,n,m)
int n,*m;double[];
{
 int i,j,k,l;
 double t;
 i=0;j=n-1;k=(i+j)/2;
 if((p[i]>=p[j])&&(p[j]>=p[k])) l=j;
    else if((p[i]>=p[k])&&(p[k]>=p[j])) l=k;
 else l=i;
 t=p[l];p[l]=p[i];
 while(i!=j)
      {
       while((i<j)&&(p[j]>=t)) j=j-1;
             if(i<j)
```

```
                        {
                         p[i]=p[j];i=i+1;
                         while((i<j)&&(p[j]<=t))
          i=i-1;
                               if(i<j)
                                  {p[j]=p[i];j=j-1;}
                          }
                  }
           p[i]=t;*m=i;
           return;
          }
```

马踏棋盘问题

```
#include <stdio.h>
#define N 5
void main(){
 int x,y;
 void horse(int i,int j);
 printf("Please input start position:");
 scanf("%d%d",&x,&y);
 horse(x-1,y-1);
}
void horse(int i,int j){
 int a[N][N]={0},start=0,
  h[]={1,2,2,1,-1,-2,-2,-1},
  v[]={2,1,-1,-2,2,1,-1,-2},
  save[N*N]={0},posnum=0,ti,tj,count=0;
 int jump(int i,int j,int a[N][N]);
 void outplan(int a[N][N]);
 a[i][j]=posnum+1;
 while(posnum>=0){
  ti=i;tj=j;
  for(start=save[posnum];start<8;++start){
   ti+=h[start];tj+=v[start];
   if(jump(ti,tj,a))
    break;
   ti-=h[start];tj-=v[start];
```

```
  }
  if(start<8){
   save[posnum]=start;
   a[ti][tj]=++posnum+1;
   i=ti;j=tj;save[posnum]=0;
   if(posnum==N*N-1){
    //outplan(a);
    count++;
   }
  }
  else{
   a[i][j]=0;
   posnum--;
   i-=h[save[posnum>;j-=v[save[posnum>;
   save[posnum]++;
  }
 }
 printf("%5d",count);
}
int jump(int i,int j,int a[N][N]){
 if(i<N&&i>=0&&j<N&&j>=0&&a[i][j]==0)
  return 1;
 return 0;
}
void outplan(int a[N][N]){
 int i,j;
 for(i=0;i<N;i++){
  for(j=0;j<N;j++)
   printf("%3d",a[i][j]);
  printf("\n");
 }
 printf("\n");
 //getchar();
}
```

用回溯法得到所有的解，但效率较低，只能算出 5 行 5 列的。

冒 泡 法：

目 的：按要求从大到小或从小到大排序。

基 本 思 路：对尚未排序的各元素从头到尾依次依次比较相邻的两个元素是否逆序（与欲排顺序相反），若逆序就交换这两元

素，经过第一轮比较排序后便可把最大（或最小）的元素排好，然后再用同样的方法把剩下的元素逐个进行比较，就得到了你所要的顺序。可以看出如果有 N 个元素，那么一共要进行 n-1 轮比较，第 I 轮要进行 j=n-i 次比较。（如：有 5 个元素，则要进行 5-1 轮比较。第 3 轮则要进行 5-3 次比较）

C++为例：这例中用到外部静态数组，也可以不用，用参数传递的方法。

```
#include<iostream.h>
static int age[5]={21,18,20,17,19};
void main()
{
 void sort_age();    //函数原形声明
 void prt_age();     //函数原形声明
 sort_age();
 prt_age();
}
void sort_age()      //冒泡实现函数
{
 extern int size;     //引用性声明
 size=sizeof(age)/2       //计算数组大
小
 for(int i=1;i<=size-1;i++)
     for(int j=0;j<=size-i;j++)
         if(age[j]>age[j+1])
         {
          int temp;
          temp=age[j];
          age[j]=age[j+1];
          age[j+1]=temp;
          }
}
void prt_age()
{
 extern int size;      //引用性声明
 for(int i=0;i<size;i++)
 cout<<"age["<<i<<"]:"<<age[i]<<endl;
```

```
}
int size      //定义性声明
```

C 语言为例：注意哦：本例 a[0]不用，只用 a[1]～a[10]，以符合人们的习惯，所以定义为 a[11]。

```
main()
{
 int a[11];
 int i,j,k;
 printf("intput 10 numbers:\n");
 for(i=1;i<11;i++)
     scanf("%d",&a[i]);
 printf("\n");
 for(i=1;i<=9;i++)
     for(j=1;j<=10-i;j++)
         if(a[j]>a[j+1])
         {
          k=a[j];a[j]=a[j+1];a[j+1]=k;
          }
  printf("the sorted numbers:\n");
  for(i=1;i<11;i++)
      printf("%d ",a[i]);
}
```

排序算法五例

一、插入排序(Insertion Sort)

1. 基本思想：

每次将一个待排序的数据元素，插入到前面已经排好序的数列中的适当位置，使数列依然有序；直到待排序数据元素全部插入完为止。

2. 排序过程：

【示例】：

[初始关键字] [49] 38 65 97 76 13 27 49

J=2(38) [38 49] 65 97 76 13 27 49

J=3(65) [38 49 65] 97 76 13 27 49

J=4(97) [38 49 65 97] 76 13 27 49
J=5(76) [38 49 65 76 97] 13 27 49
J=6(13) [13 38 49 65 76 97] 27 49
J=7(27) [13 27 38 49 65 76 97] 49
J=8(49) [13 27 38 49 49 65 76 97]

```
Procedure InsertSort(Var R : FileType);
//对 R[1..N]按递增序进行插入排序, R[0]是监视哨//
 Begin
  for I := 2 To N Do //依次插入 R[2],...,R[n]//
  begin
    R[0] := R[I]; J := I - 1;
    While R[0] < R[J] Do //查找 R[I]的插入位置//
     begin
      R[J+1] := R[J]; //将大于 R[I]的元素后移//
      J := J - 1
     end
    R[J + 1] := R[0] ; //插入 R[I] //
  end
 End; //InsertSort //
```

二、选择排序

1. 基本思想：

每一趟从待排序的数据元素中选出最小（或最大）的一个元素，顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。

2. 排序过程：

【示例】：

初始关键字 [49 38 65 97 76 13 27 49]
第一趟排序后 13 ［38 65 97 76 49 27 49]
第二趟排序后 13 27 ［65 97 76 49 38 49]
第三趟排序后 13 27 38 [97 76 49 65 49]
第四趟排序后 13 27 38 49 [49 97 65 76]
第五趟排序后 13 27 38 49 49 [97 97 76]
第六趟排序后 13 27 38 49 49 76 [76 97]
第七趟排序后 13 27 38 49 49 76 76 [97]
最后排序结果 13 27 38 49 49 76 76 97

```
Procedure SelectSort(Var R : FileType); //对 R[1..N]进行直接选择排序 //
 Begin
  for I := 1 To N - 1 Do //做 N - 1 趟选择排序//
   begin
    K := I;
    For J := I + 1 To N Do //在当前无序区 R[I..N]中选最小的元素 R[K]//
```

```
    begin
     If R[J] < R[K] Then K := J
    end;
   If K <> I Then //交换 R[I]和 R[K] //
    begin Temp := R[I]; R[I] := R[K]; R[K] := Temp; end;
  end
End; //SelectSort //
```

三、冒泡排序(BubbleSort)

1. 基本思想：

两两比较待排序数据元素的大小，发现两个数据元素的次序相反时即进行交换，直到没有反序的数据元素为止。

2. 排序过程：

设想被排序的数组 R［1..N］垂直竖立，将每个数据元素看作有重量的气泡，根据轻气泡不能在重气泡之下的原则，从下往上扫描数组 R，凡扫描到违反本原则的轻气泡，就使其向上"漂浮"，如此反复进行，直至最后任何两个气泡都是轻者在上，重者在下为止。

【示例】：

```
49 13 13 13 13 13 13 13
38 49 27 27 27 27 27 27
65 38 49 38 38 38 38 38
97 65 38 49 49 49 49 49
76 97 65 49 49 49 49 49
13 76 97 65 65 65 65 65
27 27 76 97 76 76 76 76
49 49 49 76 97 97 97 97
```

```
Procedure BubbleSort(Var R : FileType) //从下往上扫描的起泡排序//
Begin
 For I := 1 To N-1 Do //做 N-1 趟排序//
  begin
   NoSwap := True; //置未排序的标志//
   For J := N - 1 DownTo 1 Do //从底部往上扫描//
    begin
     If R[J+1]< R[J] Then //交换元素//
      begin
       Temp := R[J+1]; R[J+1 := R[J]; R[J] := Temp;
       NoSwap := False
      end;
    end;
   If NoSwap Then Return//本趟排序中未发生交换，则终止算法//
  end
End; //BubbleSort//
```

四、快速排序（Quick Sort）

1. 基本思想：

在当前无序区 R[1..H]中任取一个数据元素作为比较的"基准"(不妨记为 X)，用此基准将当前无序区划分为左右两个较小的无序区：R[1..I-1]和 R[I+1..H]，且左边的无序子区中数据元素均小于等于基准元素，右边的无序子区中数据元素均大于等于基准元素，而基准 X 则位于最终排序的位置上，即 R[1..I-1]≤X.Key≤R[I+1..H](1≤I≤H)，当 R[1..I-1]和 R[I+1..H]均非空时，分别对它们进行上述的划分过程，直至所有无序子区中的数据元素均已排序为止。

2. 排序过程：

【示例】：

初始关键字 [49 38 65 97 76 13 27 49］

第一次交换后 ［27 38 65 97 76 13 49 49］

第二次交换后 ［27 38 49 97 76 13 65 49］

J 向左扫描，位置不变，第三次交换后 ［27 38 13 97 76 49 65 49］

I 向右扫描，位置不变，第四次交换后 ［27 38 13 49 76 97 65 49］

J 向左扫描 ［27 38 13 49 76 97 65 49］

（一次划分过程）

初始关键字 ［49 38 65 97 76 13 27 49］

一趟排序之后 ［27 38 13］ 49 ［76 97 65 49］

二趟排序之后 ［13］ 27 ［38］ 49 ［49 65］76 ［97］

三趟排序之后 13 27 38 49 49 ［65］76 97

最后的排序结果 13 27 38 49 49 65 76 97

各趟排序之后的状态

```
Procedure Parttion(Var R : FileType; L, H : Integer; Var I : Integer);
//对无序区 R[1,H]做划分，I 给以出本次划分后已被定位的基准元素的位置 //
Begin
 I := 1; J := H; X := R[I] ;//初始化，X 为基准//
 Repeat
  While (R[J] >= X) And (I < J) Do
   begin
    J := J - 1 //从右向左扫描，查找第 1 个小于 X 的元素//
    If I < J Then //已找到 R[J] 〈X//
     begin
      R[I] := R[J]; //相当于交换 R[I]和 R[J]//
      I := I + 1
     end;
    While (R[I] <= X) And (I < J) Do
     I := I + 1 //从左向右扫描，查找第 1 个大于 X 的元素///
   end;
  If I < J Then //已找到 R[I] > X //
   begin       R[J] := R[I]; //相当于交换 R[I]和 R[J]//
    J := J - 1
```

```
    end
  Until I = J;
  R[I] := X //基准 X 已被最终定位//
End; //Parttion //
Procedure QuickSort(Var R :FileType; S,T: Integer); //对 R[S..T]快速排序//
Begin
  If S < T Then //当 R[S..T]为空或只有一个元素是无需排序//
    begin
      Partion(R, S, T, I); //对 R[S..T]做划分//
      QuickSort(R, S, I-1);//递归处理左区间 R[S,I-1]//
      QuickSort(R, I+1,T);//递归处理右区间 R[I+1..T] //
    end;
End; //QuickSort//
```

五、堆排序(Heap Sort)

1. 基本思想：

堆排序是一树形选择排序，在排序过程中，将 R[1..N]看成是一颗完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系来选择最小的元素。

2. 堆的定义: N 个元素的序列 K1,K2,K3,...,Kn.称为堆，当且仅当该序列满足特性：

Ki≤K2i Ki ≤K2i+1(1≤ I≤ [N/2])

堆实质上是满足如下性质的完全二叉树：树中任一非叶子结点的关键字均大于等于其孩子结点的关键字。例如序列 10,15,56,25,30,70 就是一个堆，它对应的完全二叉树如上图所示。这种堆中根结点（称为堆顶）的关键字最小，我们把它称为小根堆。反之，若完全二叉树中任一非叶子结点的关键字均大于等于其孩子的关键字，则称之为大根堆。

3. 排序过程：

堆排序正是利用小根堆（或大根堆）来选取当前无序区中关键字小（或最大）的记录实现排序的。我们不妨利用大根堆来排序。每一趟排序的基本操作是：将当前无序区调整为一个大根堆，选取关键字最大的堆顶记录，将它和无序区中的最后一个记录交换。这样，正好和直接选择排序相反，有序区是在原记录区的尾部形成并逐步向前扩大到整个记录区。

【示例】：对关键字序列 42，13，91，23，24，16，05，88 建堆

```
Procedure Sift(Var R :FileType; I, M : Integer);
//在数组 R[I..M]中调用 R[I]，使得以它为完全二叉树构成堆。事先已知其左、右子树(2I+1 <=M 时)均是堆//
Begin
  X := R[I]; J := 2*I; //若 J <=M, R[J]是 R[I]的左孩子//
  While J <= M Do //若当前被调整结点 R[I]有左孩子 R[J]//
   begin
    If (J < M) And R[J].Key < R[J+1].Key Then
     J := J + 1 //令 J 指向关键字较大的右孩子//
       //J 指向 R[I]的左、右孩子中关键字较大者//
    If X.Key < R[J].Key Then //孩子结点关键字较大//
      begin
        R[I] := R[J]; //将 R[J]换到双亲位置上//
        I := J ; J := 2*I //继续以 R[J]为当前被调整结点往下层调整//
      end;
     Else
      Exit//调整完毕，退出循环//
   end
  R[I] := X;//将最初被调整的结点放入正确位置//
End;//Sift//
Procedure HeapSort(Var R : FileType); //对 R[1..N]进行堆排序//
    Begin
       For I := N Div Downto 1 Do //建立初始堆//
          Sift(R, I , N)
       For I := N Downto 2 do //进行 N-1 趟排序//
          begin
             T := R[1]; R[1] := R[I]; R[I] := T;//将当前堆顶记录和堆中最后一个记录交换//
             Sift(R, 1, I-1) //将 R[1..I-1]重成堆//
          end
End; //HeapSort//
```

六、几种排序算法的比较和选择

1. 选取排序方法需要考虑的因素：

(1) 待排序的元素数目 n；

(2) 元素本身信息量的大小；

(3) 关键字的结构及其分布情况；

(4) 语言工具的条件，辅助空间的大小等。

2. 小结：

(1) 若 n 较小(n <= 50)，则可以采用直接插入排序或直接选择排序。由于直接插入排序所需的记录移动操作较直接选择排序多，因而当记录本身信息量较大时，用直接选择排序较好。

(2) 若文件的初始状态已按关键字基本有序，则选用直接插入或冒泡排序为宜。

(3) 若 n 较大，则应采用时间复杂度为 O(nlog2n)的排序方法：快速排序、堆排序或归并排序。 快速排序是目前基于比较的内部排序法中被认为是最好的方法。

(4) 在基于比较排序方法中，每次比较两个关键字的大小之后，仅仅出现两种可能的转移，因此可以用一棵二叉树来描述比较判定过程，由此可以证明：当文件的 n 个关键字随机分布时，任何借助于"比较"的排序算法，至少需要 O(nlog2n)的时间。

(5) 当记录本身信息量较大时，为避免耗费大量时间移动记录，可以用链表作为存储结构。

排序算法一览

第 10 章 排序

10.1 基本概念

排序(Sorting)是计算机程序设计中的一种重要操作，其功能是对一个数据元素集合或序列重新排列成一个按数据元素某个项值有序的序列。作为排序依据的数据项称为“排序码”，也即数据元素的关键码。为了便于查找，通常希望计算机中的数据表是按关键码有序的。如有序表的折半查找，查找效率较高。还有，二叉排序树、B-树和 B+树的构造过程就是一个排序过程。若关键码是主关键码，则对于任意待排序序列，经排序后得到的结果是唯一的；若关键码是次关键码，排序结果可能不唯一，这是因为具有相同关键码的数据元素，这些元素在排序结果中，它们之间的的位置关系与排序前不能保持。

若对任意的数据元素序列，使用某个排序方法，对它按关键码进行排序：若相同关键码元素间的位置关系，排序前与排序后保持一致，称此排序方法是稳定的；而不能保持一致的排序方法则称为不稳定的。

排序分为两类：内排序和外排序。

内排序：指待排序列完全存放在内存中所进行的排序过程，适合不太大的元素序列。

外排序：指排序过程中还需访问外存储器，足够大的元素序列，因不能完全放入内存，只能使用外排序。

10.2 插入排序

10.2.1 直接插入排序

设有 n 个记录，存放在数组 r 中，重新安排记录在数组中的存放顺序，使得按关键码有序。即 r[1].key≤r[2].key≤……≤r[n].key

先来看看向有序表中插入一个记录的方法：

设 1 < j ≤ n，r[1].key≤r[2].key≤……≤r[j-1].key，将 r[j]插入，重新安排存放顺序，使得 r[1].key≤r[2].key≤……≤r[j].key，得到新的有序表，记录数增 1 。

【算法 10.1】

① r[0]=r[j]； //r[j]送 r[0]中，使 r[j]为待插入记录空位
i=j-1； //从第 i 个记录向前测试插入位置，用 r[0]为辅助单元， 可免去测试 i<1。
② 若 r[0].key≥r[i].key，转④。 //插入位置确定
③ 若 r[0].key < r[i].key 时，
r[i+1]=r[i]；i=i-1；转②。 //调整待插入位置
④ r[i+1]=r[0]；结束。 //存放待插入记录
【例 10.1】向有序表中插入一个记录的过程如下：
r[1] r[2] r[3] r[4] r[5] 存储单元
2 10 18 25 9 将 r[5]插入四个记录的有序表中，j=5
r[0]=r[j]；i=j-1； 初始化，设置待插入位置
2 10 18 25 □ r[i+1]为待插入位置
i=4，r[0] < r[i]，r[i+1]=r[i]；i--； 调整待插入位置
2 10 18 □ 25
i=3，r[0] < r[i]，r[i+1]=r[i]；i--； 调整待插入位置
2 10 □ 18 25
i=2，r[0] < r[i]，r[i+1]=r[i]；i--； 调整待插入位置
2 □ 10 18 25
i=1，r[0] ≥r[i]，r[i+1]=r[0]； 插入位置确定，向空位填入插入记录
2 9 10 18 25 向有序表中插入一个记录的过程结束
直接插入排序方法：仅有一个记录的表总是有序的，因此，对 n 个记录的表，可从第二个记录开始直到第 n 个记录，逐个向有序表中进行插入操作，从而得到 n 个记录按关键码有序的表。
【算法 10.2】

```
void InsertSort(S_TBL &p)
{ for(i=2；i<=p->length；i++)
if(p->elem[i].key < p->elem[i-1].key) /*小于时，需将 elem[i]插入有序表*/
{ p->elem[0].key=p->elem[i].key； /*为统一算法设置监测*/
for(j=i-1；p->elem[0].key < p->elem[j].key；j--)
p->elem[j+1].key=p->elem[j].key； /*记录后移*/
p->elem[j+1].key=p->elem[0].key； /*插入到正确位置*/
}
}
```

【效率分析】
空间效率：仅用了一个辅助单元。
时间效率：向有序表中逐个插入记录的操作，进行了 n-1 趟，每趟操作分为比较关键码和移动记录，而比较的次数和移动记录的次数取决于待排序列按关键码的初始排列。
最好情况下：即待排序列已按关键码有序，每趟操作只需 1 次比较 2 次移动。
总比较次数=n-1 次
总移动次数=2(n-1)次
最坏情况下：即第 j 趟操作，插入记录需要同前面的 j 个记录进行 j 次关键码比较，移动记录的次数为 j+2 次。
平均情况下：即第 j 趟操作，插入记录大约同前面的 j/2 个记录进行关键码比较，移动记录的次数为 j/2+2 次。
由此，直接插入排序的时间复杂度为 O(n2)。是一个稳定的排序方法。
10.2.2 折半插入排序

直接插入排序的基本操作是向有序表中插入一个记录，插入位置的确定通过对有序表中记录按关键码逐个比较得到的。平均情况下总比较次数约为 n2/4。既然是在有序表中确定插入位置，可以不断二分有序表来确定插入位置，即一次比较，通过待插入记录与有序表居中的记录按关键码比较，将有序表一分为二，下次比较在其中一个有序子表中进行，将子表又一分为二。这样继续下去，直到要比较的子表中只有一个记录时，比较一次便确定了插入位置。

二分判定有序表插入位置方法：

① low=1；high=j-1；r[0]=r[j]； // 有序表长度为 j-1，第 j 个记录为待插入记录

//设置有序表区间，待插入记录送辅助单元

② 若 low>high，得到插入位置，转⑤

③ low≤high，m=(low+high)/2； // 取表的中点，并将表一分为二，确定待插入区间*/

④ 若 r[0].key<r[m].key，high=m-1； //插入位置在低半区

否则，low=m+1； // 插入位置在高半区

转②

⑤ high+1 即为待插入位置，从 j-1 到 high+1 的记录，逐个后移，r[high+1]=r[0]；放置待插入记录。

【算法 10.3】

```
void InsertSort(S_TBL *s)
{ /* 对顺序表 s 作折半插入排序 */
for(i=2；i<=s->length；i++)
{ s->elem[0]=s->elem[i]； /* 保存待插入元素 */
low=i；high=i-1； /* 设置初始区间 */
while(low<=high) /* 该循环语句完成确定插入位置 */
{ mid=(low+high)/2；
if(s->elem[0].key>s->elem[mid].key)
low=mid+1； /* 插入位置在高半区中 */
else high=mid-1； /* 插入位置在低半区中 */
}/* while */
for(j=i-1；j>=high+1；j--) /* high+1 为插入位置 */
s->elem[j+1]=s->elem[j]； /* 后移元素，留出插入空位 */
s->elem[high+1]=s->elem[0]； /* 将元素插入 */
}/* for */
}/* InsertSort */
```

【时间效率】

确定插入位置所进行的折半查找，关键码的比较次数至多为 ，次，移动记录的次数和直接插入排序相同，故时间复杂度仍为 O(n2)。是一个稳定的排序方法。

10.2.3 表插入排序

直接插入排序、折半插入排序均要大量移动记录，时间开销大。若要不移动记录完成排序，需要改变存储结构，进行表插入排序。所谓表插入排序，就是通过链接指针，按关键码的大小，实现从小到大的链接过程，为此需增设一个指针项。操作方法与直接插入排序类似，所不同的是直接插入排序要移动记录，而表插入排序是修改链接指针。用静态链表来说明。

```
#define SIZE 200
typedef struct{
ElemType elem； /*元素类型*/
int next； /*指针项*/
```

}NodeType； /*表结点类型*/
typedef struct{
NodeType r[SIZE]； /*静态链表*/
int length； /*表长度*/
}L_TBL； /*静态链表类型*/
假设数据元素已存储在链表中，且 0 号单元作为头结点，不移动记录而只是改变链指针域，将记录按关键码建为一个有序链表。首先，设置空的循环链表，即头结点指针域置 0，并在头结点数据域中存放比所有记录关键码都大的整数。接下来，逐个结点向链表中插入即可。
【例 10.2】表插入排序示例
MAXINT 49 38 65 97 76 13 27 49
0 - - - - - - - -
MAXINT 49 38 65 97 76 13 27 49
1 0 - - - - - - -
MAXINT 49 38 65 97 76 13 27 49
2 0 1 - - - - - -
MAXINT 49 38 65 97 76 13 27 49
2 3 1 0 - - - - -
MAXINT 49 38 65 97 76 13 27 49
2 3 1 4 0 - - - -
MAXINT 49 38 65 97 76 13 27 49
2 3 1 5 0 4 - - -
MAXINT 49 38 65 97 76 13 27 49
6 3 1 5 0 4 2 - -
MAXINT 49 38 65 97 76 13 27 49
6 3 1 5 0 4 7 2 -
MAXINT 49 38 65 97 76 13 27 49
6 8 1 5 0 4 7 2 3
图 10.1
表插入排序得到一个有序的链表，查找则只能进行顺序查找，而不能进行随机查找，如折半查找。为此，还需要对记录进行重排。
重排记录方法：按链表顺序扫描各结点，将第 i 个结点中的数据元素调整到数组的第 i 个分量数据域。因为第 i 个结点可能是数组的第 j 个分量，数据元素调整仅需将两个数组分量中数据元素交换即可，但为了能对所有数据元素进行正常调整，指针域也需处理。
【算法 10.3】
1. j=l->r[0].next；i=1； //指向第一个记录位置，从第一个记录开始调整
2. 若 i=l->length 时，调整结束；否则，
a. 若 i=j，j=l->r[j].next；i++；转(2) //数据元素应在这分量中，不用调整，处理下一个结点
b. 若 j>i，l->r[i].elem<-->l->r[j].elem； //交换数据元素
p=l->r[j].next； // 保存下一个结点地址
l->r[j].next=l->[i].next；l->[i].next=j； // 保持后续链表不被中断
j=p；i++；转(2) // 指向下一个处理的结点
c. 若 j<i，while(j<i) j=l->r[j].next；//j 分量中原记录已移走，沿 j 的指针域找寻原记录的位置转到(a)
【例 10.3】对表插入排序结果进行重排示例

MAXINT 49 38 65 97 76 13 27 49
6 8 1 5 0 4 7 2 3
MAXINT 13 38 65 97 76 49 27 49
6 (6) 1 5 0 4 8 2 3
MAXINT 13 27 65 97 76 49 38 49
6 (6) (7) 5 0 4 8 1 3
MAXINT 13 27 38 97 76 49 65 49
6 (6) (7) (7) 0 4 8 5 3
MAXINT 13 27 38 49 76 97 65 49
6 (6) (7) (7) (6) 4 0 5 3
MAXINT 13 27 38 49 49 97 65 76
6 (6) (7) (7) (6) (8) 0 5 4
MAXINT 13 27 38 49 49 65 97 76
6 (6) (7) (7) (6) (8) (7) 0 4
MAXINT 13 27 38 49 49 65 76 97
6 (6) (7) (7) (6) (8) (7) (8) 0
图 10.2

【时效分析】

表插入排序的基本操作是将一个记录插入到已排好序的有序链表中，设有序表长度为 i，则需要比较至多 i+1 次，修改指针两次。因此，总比较次数与直接插入排序相同，修改指针总次数为 2n 次。所以，时间复杂度仍为 O(n2)

10.2.4 希尔排序(Shell's Sort)

希尔排序又称缩小增量排序，是 1959 年由 D.L.Shell 提出来的，较前述几种插入排序方法有较大的改进。

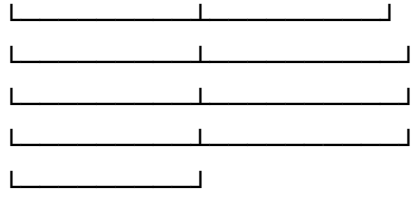
直接插入排序算法简单，在 n 值较小时，效率比较高，在 n 值很大时，若序列按关键码基本有序，效率依然较高，其时间效率可提高到 O(n)。希尔排序即是从这两点出发，给出插入排序的改进方法。

希尔排序方法：

1. 选择一个步长序列 t1，t2，…，tk，其中 ti>tj，tk=1；
2. 按步长序列个数 k，对序列进行 k 趟排序；
3. 每趟排序，根据对应的步长 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅步长因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

【例 10.4】待排序列为 39，80，76，41，13，29，50，78，30，11，100，7，41，86。步长因子分别取 5、3、1，则排序过程如下：

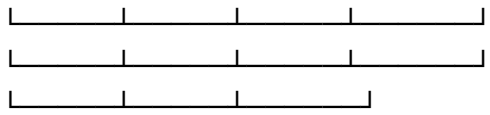
p=5 39 80 76 41 13 29 50 78 30 11 100 7 41 86



子序列分别为{39，29，100}，{80，50，7}，{76，78，41}，{41，30，86}，{13，11}。

第一趟排序结果：

p=3 29 7 41 30 11 39 50 76 41 13 100 80 78 86



子序列分别为{29，30，50，13，78}，{7，11，76，100，86}，{41，39，41，80}。
第二趟排序结果：
p=1 13 7 39 29 11 41 30 76 41 50 86 80 78 100
此时，序列基本“有序”，对其进行直接插入排序，得到最终结果：
7 11 13 29 30 39 41 41 50 76 78 80 86 100
图 10.3
【算法 10.5】

```
void ShellInsert(S_TBL &p，int dk)
{ /*一趟增量为 dk 的插入排序，dk 为步长因子*/
for(i=dk+1；i<=p->length；i++)
if(p->elem[i].key < p->elem[i-dk].key) /*小于时，需 elem[i]将插入有序表*/
{ p->elem[0]=p->elem[i]； /*为统一算法设置监测*/
for(j=i-dk；j>0&&p->elem[0].key < p->elem[j].key；j=j-dk)
p->elem[j+dk]=p->elem[j]； /*记录后移*/
p->elem[j+dk]=p->elem[0]； /*插入到正确位置*/
}
}
void ShellSort(S_TBL *p，int dlta[]，int t)
{ /*按增量序列 dlta[0，1…，t-1]对顺序表*p 作希尔排序*/
for(k=0；k<t；t++)
ShellSort(p，dlta[k])； /*一趟增量为 dlta[k]的插入排序*/
}
```

【时效分析】
希尔排序时效分析很难，关键码的比较次数与记录移动次数依赖于步长因子序列的选取，特定情况下可以准确估算出关键码的比较次数和记录的移动次数。目前还没有人给出选取最好的步长因子序列的方法。步长因子序列可以有各种取法，有取奇数的，也有取质数的，但需要注意：步长因子中除 1 外没有公因子，且最后一个步长因子必须为 1。希尔排序方法是一个不稳定的排序方法。

10.3 交换排序

交换排序主要是通过两两比较待排记录的关键码，若发生与排序要求相逆，则交换之。

10.3.1 冒泡排序(Bubble Sort)

先来看看待排序列一趟冒泡的过程：设 1<j≤n，r[1],r[2],…,r[j]为待排序列，
通过两两比较、交换，重新安排存放顺序，使得 r[j]是序列中关键码最大的记录。一趟冒泡方法为：
① i=1； //设置从第一个记录开始进行两两比较
② 若 i≥j，一趟冒泡结束。
③ 比较 r[i].key 与 r[i+1].key，若 r[i].key≤r[i+1].key，不交换，转⑤
④ 当 r[i].key>r[i+1].key 时， r[0]=r[i]；r[i]=r[i+1]；r[i+1]=r[0]；
将 r[i]与 r[i+1]交换
⑤ i=i+1； 调整对下两个记录进行两两比较，转②
冒泡排序方法：对 n 个记录的表，第一趟冒泡得到一个关键码最大的记录 r[n]，第二趟冒泡对

n-1 个记录的表，再得到一个关键码最大的记录 r[n-1]，如此重复，直到 n 个记录按关键码有序的表。

【算法 10.6】

① j=n； //从 n 记录的表开始

② 若 j<2，排序结束

③ i=1； //一趟冒泡，设置从第一个记录开始进行两两比较，

④ 若 i≥j，一趟冒泡结束，j=j-1；冒泡表的记录数-1，转②

⑤ 比较 r[i].key 与 r[i+1].key，若 r[i].key≤r[i+1].key，不交换，转⑤

⑥ 当 r[i].key>r[i+1].key 时， r[i]<-->r[i+1]； 将 r[i]与 r[i+1]交换

⑦ i=i+1； 调整对下两个记录进行两两比较，转④

【效率分析】

空间效率：仅用了一个辅助单元。

时间效率：总共要进行 n-1 趟冒泡，对 j 个记录的表进行一趟冒泡需要 j-1 次关键码比较。

移动次数：

最好情况下：待排序列已有序，不需移动。

10.3.2 快速排序

快速排序是通过比较关键码、交换记录，以某个记录为界(该记录称为支点)，将待排序列分成两部分。其中，一部分所有记录的关键码大于等于支点记录的关键码，另一部分所有记录的关键码小于支点记录的关键码。我们将待排序列按关键码以支点记录分成两部分的过程，称为一次划分。对各部分不断划分，直到整个序列按关键码有序。

一次划分方法：

设 1≤p<q≤n，r[p],r[p+1],...,r[q]为待排序列

① low=p；high=q； //设置两个搜索指针，low 是向后搜索指针，high 是向前搜索指针

r[0]=r[low]； //取第一个记录为支点记录，low 位置暂设为支点空位

② 若 low=high，支点空位确定，即为 low。

r[low]=r[0]； //填入支点记录，一次划分结束

否则，low<high，搜索需要交换的记录，并交换之

③ 若 low<high 且 r[high].key≥r[0].key //从 high 所指位置向前搜索，至多到 low+1 位置

high=high-1；转③ //寻找 r[high].key<r[0].key

r[low]=r[high]； //找到 r[high].key<r[0].key，设置 high 为新支点位置，

//小于支点记录关键码的记录前移。

④ 若 low<high 且 r[low].key<r[0].key //从 low 所指位置向后搜索，至多到 high-1 位置

low=low+1；转④ //寻找 r[low].key≥r[0].key

r[high]=r[low]； //找到 r[low].key≥r[0].key，设置 low 为新支点位置，

//大于等于支点记录关键码的记录后移。

转② //继续寻找支点空位

【算法 10.7】

```
int Partition(S_TBL *tbl,int low,int high) /*一趟快排序*/
{ /*交换顺序表 tbl 中子表 tbl->[low…high]的记录，使支点记录到位，并反回其所在位置*/
/*此时，在它之前(后)的记录均不大(小)于它*/
tbl->r[0]=tbl->r[low]; /*以子表的第一个记录作为支点记录*/
pivotkey=tbl->r[low].key; /*取支点记录关键码*/
while(low<higu) /*从表的两端交替地向中间扫描*/
{ while(low<high&&tbl->r[high].key>=pivotkey) high--;
```

tbl->r[low]=tbl->r[high]; /*将比支点记录小的交换到低端*/
while(low<high&&tbl-g>r[high].key<=pivotkey) low++;
tbl->r[low]=tbl->r[high]; /*将比支点记录大的交换到低端*/
}
tbl->r[low]=tbl->r[0]; /*支点记录到位*/
return low; /*反回支点记录所在位置*/
}
【例 10.5】一趟快排序过程示例
r[1] r[2] r[3] r[4] r[5] r[6] r[7] r[8] r[9] r[10] 存储单元
49 14 38 74 96 65 8 49 55 27 记录中关键码
low=1；high=10； 设置两个搜索指针， r[0]=r[low]； 支点记录送辅助单元，
□ 14 38 74 96 65 8 49 55 27
↑ ↑
low high
第一次搜索交换
从 high 向前搜索小于 r[0].key 的记录，得到结果
27 14 38 74 96 65 8 49 55 □
↑ ↑
low high
从 low 向后搜索大于 r[0].key 的记录，得到结果
27 14 38 □ 96 65 8 49 55 74
↑ ↑
low high
第二次搜索交换
从 high 向前搜索小于 r[0].key 的记录，得到结果
27 14 38 8 96 65 □ 49 55 74
↑ ↑
low high
从 low 向后搜索大于 r[0].key 的记录，得到结果
27 14 38 8 □ 65 96 49 55 74
↑ ↑
low high
第三次搜索交换
从 high 向前搜索小于 r[0].key 的记录，得到结果
27 14 38 8 □ 65 96 49 55 74
↑↑
low high
从 low 向后搜索大于 r[0].key 的记录，得到结果
27 14 38 8 □ 65 96 49 55 74
↑↑
low high
low=high，划分结束，填入支点记录
27 14 38 8 49 65 96 49 55 74
【算法 10.8】

```
void QSort(S_TBL *tbl,int low,int high) /*递归形式的快排序*/
{ /*对顺序表 tbl 中的子序列 tbl->[low…high]作快排序*/
if(low<high)
{ pivotloc=partition(tbl,low,high); /*将表一分为二*/
QSort(tbl,low,pivotloc-1); /*对低子表递归排序*/
QSort(tbl,pivotloc+1,high); /*对高子表递归排序*/
}
}
```

【效率分析】

空间效率：快速排序是递归的，每层递归调用时的指针和参数均要用栈来存放，递归调用层次数与上述二叉树的深度一致。因而，存储开销在理想情况下为 O(log2n)，即树的高度；在最坏情况下，即二叉树是一个单链，为 O(n)。

时间效率：在 n 个记录的待排序列中，一次划分需要约 n 次关键码比较，时效为 O(n)，若设 T(n)为对 n 个记录的待排序列进行快速排序所需时间。

理想情况下：每次划分，正好将分成两个等长的子序列，则

T(n)≤cn+2T(n/2) c 是一个常数

≤cn+2(cn/2+2T(n/4))=2cn+4T(n/4)

≤2cn+4(cn/4+T(n/8))=3cn+8T(n/8)

.....

≤cnlog2n+nT(1)=O(nlog2n)

最坏情况下：即每次划分，只得到一个子序列，时效为 O(n2)。

快速排序是通常被认为在同数量级（O(nlog2n)）的排序方法中平均性能最好的。但若初始序列按关键码有序或基本有序时，快排序反而蜕化为冒泡排序。为改进之，通常以“三者取中法”来选取支点记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。快速排序是一个不稳定的排序方法。

10.4 选择排序

选择排序主要是每一趟从待排序列中选取一个关键码最小的记录，也即第一趟从 n 个记录中选取关键码最小的记录，第二趟从剩下的 n-1 个记录中选取关键码最小的记录，直到整个序列的记录选完。这样，由选取记录的顺序，便得到按关键码有序的序列。

10.4.1 简单选择排序

操作方法：第一趟，从 n 个记录中找出关键码最小的记录与第一个记录交换；第二趟，从第二个记录开始的 n-1 个记录中再选出关键码最小的记录与第二个记录交换；如此，第 i 趟，则从第 i 个记录开始的 n-i+1 个记录中选出关键码最小的记录与第 i 个记录交换，直到整个序列按关键码有序。

【算法 10.9】

```
void SelectSort(S_TBL *s)
{ for(i=1；i<s->length；i++)
{ /* 作 length-1 趟选取 */
for(j=i+1，t=i；j<=s->length；j++)
{ /* 在 i 开始的 length-n+1 个记录中选关键码最小的记录 */
if(s->elem[t].key>s->elem[j].key)
t=j； /* t 中存放关键码最小记录的下标 */
}
```

s->elem[t]<-->s->elem[i]； /* 关键码最小的记录与第 i 个记录交换 */
}
}
从程序中可看出，简单选择排序移动记录的次数较少，但关键码的比较次数依然是
10.4.2 树形选择排序
按照锦标赛的思想进行，将 n 个参赛的选手看成完全二叉树的叶结点，则该完全二叉树有 2n-2 或 2n-1 个结点。首先，两两进行比赛(在树中是兄弟的进行，否则轮空，直接进入下一轮)，胜出的再兄弟间再两两进行比较，直到产生第一名；接下来，将作为第一名的结点看成最差的，并从该结点开始，沿该结点到根路径上，依次进行各分枝结点子女间的比较，胜出的就是第二名。因为和他比赛的均是刚刚输给第一名的选手。如此，继续进行下去，直到所有选手的名次排定。
【例 10.6】16 个选手的比赛(n=24)
图 10.5

图 10.6
图 10.5 中，从叶结点开始的兄弟间两两比赛，胜者上升到父结点；胜者兄弟间再两两比赛，直到根结点，产生第一名 91。比较次数为 23+22+21+20=24-1=n-1。
图 10.6 中，将第一名的结点置为最差的，与其兄弟比赛，胜者上升到父结点，胜者兄弟间再比赛，直到根结点，产生第二名 83。比较次数为 4，即 log2n 次。其后各结点的名次均是这样产生的，所以，对于 n 个参赛选手来说，即对 1，故时间复杂度为 O(nlog2n)。该方法占用空间较多，除需输出排序结果的 n 个单元外，尚需 n-1 个辅助单元。–n+1)log2n–n 个记录进行树形选择排序，总的关键码比较次数至多为(n
10.4.3 堆排序(Heap Sort)
设有 n 个元素的序列 k1，k2，…，kn，当且仅当满足下述关系之一时，称之为堆。

图 10.7 两个堆示例
若以一维数组存储一个堆，则堆对应一棵完全二叉树，且所有非叶结点的值均不大于(或不小于)其子女的值，根结点的值是最小(或最大)的。
设有 n 个元素，将其按关键码排序。首先将这 n 个元素按关键码建成堆，将堆顶元素输出，得到 n 个元素中关键码最小(或最大)的元素。然后，再对剩下的 n-1 个元素建成堆，输出堆顶元素，得到 n 个元素中关键码次小(或次大)的元素。如此反复，便得到一个按关键码有序的序列。称这个过程为堆排序。
因此，实现堆排序需解决两个问题：
1. 如何将 n 个元素的序列按关键码建成堆；
2. 输出堆顶元素后，怎样调整剩余 n-1 个元素，使其按关键码成为一个新堆。
首先，讨论输出堆顶元素后，对剩余元素重新建成堆的调整过程。
调整方法：设有 m 个元素的堆，输出堆顶元素后，剩下 m-1 个元素。将堆底元素送入堆顶，堆被破坏，其原因仅是根结点不满足堆的性质。将根结点与左、右子女中较小(或小大)的进行交换。若与左子女交换，则左子树堆被破坏，且仅左子树的根结点不满足堆的性质；若与右子女交换，则右子树堆被破坏，且仅右子树的根结点不满足堆的性质。继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，堆被建成。称这个自根结点到叶子结点的调整过程为筛选。
【例 10.6】
图 10.8 自堆顶到叶子的调整过程

再讨论对 n 个元素初始建堆的过程。
建堆方法：对初始序列建堆的过程，就是一个反复进行筛选的过程。n 个结点的完全
子树成为堆，之后向前依次对各结点为根的子树进行筛选，使之成为堆，直到根结点。
【例 10.7】

堆排序：对 n 个元素的序列进行堆排序，先将其建成堆，以根结点与第 n 个结点交换；调整前 n-1 个结点成为堆，再以根结点与第 n-1 个结点交换；重复上述操作，直到整个序列有序。
【算法 10.10】

```
void HeapAdjust(S_TBL *h，int s，int m)
{/*r[s…m]中的记录关键码除 r[s]外均满足堆的定义，本函数将对第 s 个结点为根的子树筛选，使其成为大顶堆*/
rc=h->r[s];
for(j=2*s；j<=m；j=j*2) /* 沿关键码较大的子女结点向下筛选 */
{ if(j<m&&h->r[j].key<h->r[j+1].key)
j=j+1； /* 为关键码较大的元素下标*/
if(rc.key<h->r[j].key) break； /* rc 应插入在位置 s 上*/
h->r[s]=h->r[j]； s=j； /* 使 s 结点满足堆定义 */
}
h->r[s]=rc； /* 插入 */
}
void HeapSort(S_TBL *h)
{ for(i=h->length/2；i>0；i--) /* 将 r[1..length]建成堆 */
HeapAdjust(h，i，h->length)；
for(i=h->length；i>1；i--)
{ h->r[1]<-->h->r[i]； /* 堆顶与堆低元素交换 */
HeapAdjust(h，1，i-1)； /*将 r[1..i-1]重新调整为堆*/
}
}
```

次，交换记录至多 k 次。所以，在建好堆后，排序过程中的筛选次数不超过下式：
+ … +⌋2)–log2(n⌊ + ⌋1)–log2(n⌊(2)⌋log22 < nlog2n2
而建堆时的比较次数不超过 4n 次，因此堆排序最坏情况下，时间复杂度也为 O(nlog2n)。
10.5 二路归并排序
二路归并排序的基本操作是将两个有序表合并为一个有序表。
设 r[u…t]由两个有序子表 r[u…v-1]和 r[v…t]组成，两个子表长度分别为 v-u、t-v+1。合并方法为：
⑴ i=u；j=v；k=u； //置两个子表的起始下标及辅助数组的起始下标
⑵ 若 i>v 或 j>t，转⑷ //其中一个子表已合并完，比较选取结束
⑶ //选取 r[i]和 r[j]关键码较小的存入辅助数组 rf
如果 r[i].key<r[j].key，rf[k]=r[i]； i++； k++； 转⑵
否则，rf[k]=r[j]； j++； k++； 转⑵
⑷ //将尚未处理完的子表中元素存入 rf
如果 i<v，将 r[i…v-1]存入 rf[k…t] //前一子表非空
如果 j<=t，将 r[i…v]存入 rf[k…t] //后一子表非空
⑸ 合并结束。

【算法 10.11】

```
void Merge(ElemType *r，ElemType *rf，int u，int v，int t)
{
for(i=u，j=v，k=u；i<v&&j<=t；k++)
{ if(r[i].key<r[j].key)
{ rf[k]=r[i]；i++；}
else
{ rf[k]=r[j]；j++；}
}
if(i<v) rf[k…t]=r[i…v-1]；
if(j<=t) rf[k…t]=r[j…t]；
}
```

一.两路归并的迭代算法

1 个元素的表总是有序的。所以对 n 个元素的待排序列，每个元素可看成 1 个有序子表长度均为 2。再进行两两合并，直到生成 n 个元素按关键码有序的表。

【算法 10.12】

```
void MergeSort(S_TBL *p，ElemType *rf)
{ /*对*p 表归并排序，*rf 为与*p 表等长的辅助数组*/
ElemType *q1，*q2；
q1=rf；q2=p->elem；
for(len=1；len<p->length；len=2*len) /*从 q2 归并到 q1*/
{ for(i=1；i+2*len-1<=p->length；i=i+2*len)
Merge(q2，q1，i，i+len，i+2*len-1)； /*对等长的两个子表合并*/
if(i+len-1<p->length)
Merge(q2，q1，i，i+len，p->length)； /*对不等长的两个子表合并*/
else if(i<=p->length)
while(i<=p->length) /*若还剩下一个子表，则直接传入*/
q1[i]=q2[i]；
q1<-->q2； /*交换，以保证下一趟归并时，仍从 q2 归并到 q1*/
if(q1!=p->elem) /*若最终结果不在*p 表中，则传入之*/
for(i=1；i<=p->length；i++)
p->elem[i]=q1[i]；
}
}
```

二.两路归并的递归算法

【算法 10.13】

```
void MSort(ElemType *p，ElemType *p1，int s，int t)
{ /*将 p[s…t]归并排序为 p1[s…t]*/
if(s==t) p1[s]=p[s]
else
{ m=(s+t)/2； /*平分*p 表*/
MSort(p，p2，s，m)； /*递归地将 p[s…m]归并为有序的 p2[s…m]*/
MSort(p，p2，m+1，t)； /*递归地将 p[m+1…t]归并为有序的 p2[m+1…t]*/
Merge(p2，p1，s，m+1，t)； /*将 p2[s…m]和 p2[m+1…t]归并到 p1[s…t]*/
```

```
}
}
void MergeSort(S_TBL *p)
{ /*对顺序表*p 作归并排序*/
MSort(p->elem，p->elem，1，p->length)；
}
```

【效率分析】

需要一个与表等长的辅助元素数组空间，所以空间复杂度为 O(n)。

对 n 个元素的表，将这 n 个元素看作叶结点，若将两两归并生成的子表看作它们的父结点，则归并过程对应由叶向根生成一棵二叉树的过程。所以归并趟数约等于二叉树的高度-1，即 log2n，每趟归并需移动记录 n 次，故时间复杂度为 O(nlog2n)。

10.6 基数排序

基数排序是一种借助于多关键码排序的思想，是将单关键码按基数分成“多关键码”进行排序的方法。

10.6.1 多关键码排序

扑克牌中 52 张牌，可按花色和面值分成两个字段，其大小关系为：

花色： 梅花 < 方块 < 红心 < 黑心

面值： 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

若对扑克牌按花色、面值进行升序排序，得到如下序列：

梅花 2,3,...,A，方块 2,3,...,A，红心 2,3,...,A，黑心 2,3,...,A

即两张牌，若花色不同，不论面值怎样，花色低的那张牌小于花色高的，只有在同花色情况下，大小关系才由面值的大小确定。这就是多关键码排序。

为得到排序结果，我们讨论两种排序方法。

方法 1：先对花色排序，将其分为 4 个组，即梅花组、方块组、红心组、黑心组。再对每个组分别按面值进行排序，最后，将 4 个组连接起来即可。

方法 2：先按 13 个面值给出 13 个编号组(2 号，3 号，...，A 号)，将牌按面值依次放入对应的编号组，分成 13 堆。再按花色给出 4 个编号组(梅花、方块、红心、黑心)，将 2 号组中牌取出分别放入对应花色组，再将 3 号组中牌取出分别放入对应花色组，……，这样，4 个花色组中均按面值有序，然后，将 4 个花色组依次连接起来即可。

设 n 个元素的待排序列包含 d 个关键码{k1，k2，…，kd}，则称序列对关键码{k1，k2，…，kd}有序是指：对于序列中任两个记录 r[i]和 r[j](1≤i≤j≤n)都满足下列有序关系：

其中 k1 称为最主位关键码，kd 称为最次位关键码。

多关键码排序按照从最主位关键码到最次位关键码或从最次位到最主位关键码的顺序逐次排序，分两种方法：

最高位优先(Most Significant Digit first)法，简称 MSD 法：先按 k1 排序分组，同一组中记录，关键码 k1 相等，再对各组按 k2 排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码 kd 对各子组排序后。再将各组连接起来，便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法一即是 MSD 法。

最低位优先(Least Significant Digit first)法，简称 LSD 法：先从 kd 开始排序，再对 kd-1 进行排序，依次重复，直到对 k1 排序后便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法二即是 LSD 法。

10.6.2 链式基数排序

将关键码拆分为若干项，每项作为一个“关键码”，则对单关键码的排序可按多关键码排序方法进行。比如，关键码为 4 位的整数，可以每位对应一项，拆分成 4 项；又如，关键码由 5 个字符组成的字符串，可以每个字符作为一个关键码。由于这样拆分后，每个关键码都在相同的范围内(对数字是 0～9，字符是'a'～'z')，称这样的关键码可能出现的符号个数为“基”，记作 RADIX。上述取数字为关键码的“基”为 10；取字符为关键码的“基”为 26。基于这一特性，用 LSD 法排序较为方便。

基数排序：从最低位关键码起，按关键码的不同值将序列中的记录“分配”到 RADIX 个队列中，然后再“收集”之。如此重复 d 次即可。链式基数排序是用 RADIX 个链队列作为分配队列，关键码相同的记录存入同一个链队列中，收集则是将各链队列按关键码大小顺序链接起来。

【例 10.8】以静态链表存储待排记录，头结点指向第一个记录。链式基数排序过程如下图。

图(a)：初始记录的静态链表。

图(b):第一趟按个位数分配，修改结点指针域，将链表中的记录分配到相应链队列中。

图(c):第一趟收集：将各队列链接起来，形成单链表。

图(d):第二趟按十位数分配，修改结点指针域，将链表中的记录分配到相应链队列中。

图(e):第二趟收集：将各队列链接起来，形成单链表。

图(f):第三趟按百位数分配，修改结点指针域，将链表中的记录分配到相应链队列中。

图(g):第三趟收集：将各队列链接起来，形成单链表。此时,序列已有序。

图 10.10

【算法 10.14】

```
#define MAX_KEY_NUM 8 /*关键码项数最大值*/
#define RADIX 10 /*关键码基数，此时为十进制整数的基数*/
#define MAX_SPACE 1000 /*分配的最大可利用存储空间*/
typedef struct{
KeyType keys[MAX_KEY_NUM]；  /*关键码字段*/
InfoType otheritems；  /*其它字段*/
int next；  /*指针字段*/
}NodeType；  /*表结点类型*/
typedef struct{
NodeType r[MAX_SPACE]；  /*静态链表，r[0]为头结点*/
int keynum；  /*关键码个数*/
int length；  /*当前表中记录数*/
}L_TBL；  /*链表类型*/
typedef int ArrayPtr[radix]；  /*数组指针，分别指向各队列*/
void Distribute(NodeType *s，int i，ArrayPtr *f，ArrayPtr *e)
{ /*静态链表 ltbl 的 r 域中记录已按(kye[0]，keys[1]，…，keys[i-1])有序)*/
/*本算法按第 i 个关键码 keys[i]建立 RADIX 个子表，使同一子表中的记录的 keys[i]相同*/
/*f[0…RADIX-1]和 e[0…RADIX-1]分别指向各子表的第一个和最后一个记录*/
for(j=0；j<RADIX；j++) f[j]=0；  /*  各子表初始化为空表*/
for(p=r[0].next；p；p=r[p].next)
{ j=ord(r[p].keys[i])；  /*ord 将记录中第 i 个关键码映射到[0…RADIX-1]*/
if(!f[j]) f[j]=p；
else r[e[j]].next=p；
e[j]=p；  /*  将 p 所指的结点插入到第 j 个子表中*/
```

```
}
}
void Collect(NodeType *r，int i，ArrayPtr f，ArrayPtr e)
{/*本算法按 keys[i]自小到大地将 f[0…RADIX-1]所指各子表依次链接成一个链表
*e[0…RADIX-1]为各子表的尾指针*/
for(j=0；!f[j]；j=succ(j))； /*找第一个非空子表，succ 为求后继函数*/
r[0].next=f[j]；t=e[j]； /*r[0].next 指向第一个非空子表中第一个结点*/
while(j<RADIX)
{ for(j=succ(j)；j<RADIX-1&&!f[j]；j=succ(j))； /*找下一个非空子表*/
if(f[j]) {r[t].next=f[j]；t=e[j]；} /*链接两个非空子表*/
}
r[t].next=0； /*t 指向最后一个非空子表中的最后一个结点*/
}
void RadixSort(L_TBL *ltbl)
{ /*对 ltbl 作基数排序，使其成为按关键码升序的静态链表，ltbl->r[0]为头结点*/
for(i=0；i<ltbl->length；i++) ltbl->r[i].next=i+1；
ltbl->r[ltbl->length].next=0； /*将 ltbl 改为静态链表*/
for(i=0；i<ltbl->keynum；i++) /*按最低位优先依次对各关键码进行分配和收集*/
{ Distribute(ltbl->r，i，f，e)； /*第 i 趟分配*/
Collect(ltbl->r，i，f，e)； /*第 i 趟收集*/
}
}
```

【效率分析】

时间效率：设待排序列为 n 个记录，d 个关键码，关键码的取值范围为 radix，则进行链式基数排序的时间复杂度为 O(d(n+radix))，其中，一趟分配时间复杂度为 O(n)，一趟收集时间复杂度为 O(radix)，共进行 d 趟分配和收集。

空间效率：需要 2*radix 个指向队列的辅助空间，以及用于静态链表的 n 个指针。

10.7 外排序

10.7.1 外部排序的方法

外部排序基本上由两个相互独立的阶段组成。首先，按可用内存大小，将外存上含 n 个记录的文件分成若干长度为 k 的子文件或段(segment)，依次读入内存并利用有效的内部排序方法对它们进行排序，并将排序后得到的有序子文件重新写入外存。通常称这些有序子文件为归并段或顺串；然后，对这些归并段进行逐趟归并，使归并段(有序子文件)逐渐由小到大，直至得到整个有序文件为止。

显然，第一阶段的工作已经讨论过。以下主要讨论第二阶段即归并的过程。先从一个例子来看外排序中的归并是如何进行的？

假设有一个含 10000 个记录的文件，首先通过 10 次内部排序得到 10 个初始归并段 R1～R10 ，其中每一段都含 1000 个记录。然后对它们作如图 10.11 所示的两两归并，直至得到一个有序文件为止。

从图 10.11 可见，由 10 个初始归并段到一个有序文件，共进行了四趟归并，每一趟

将两个有序段归并成一个有序段的过程，若在内存中进行，则很简单，前面讨论的 2-路归并排序中的 Merge 函数便可实现此归并。但是，在外部排序中实现两两归并时，不仅要调用 Merge 函数，而且要进行外存的读/写，这是由于我们不可能将两个有序段及归并结果同时放在内存中的缘故。对外存上信息的读/写是以“物理块”为单位。假设在上例中每个物理块可以容纳 200 个记录，则每一趟归并需进行 50 次“读”和 50 次“写”，四趟归并加上内部排序时所需进行的读/写，使得在外排序中总共需进行 500 次的读/写。

一般情况下，外部排序所需总时间=

内部排序(产生初始归并段)所需时间 m*tis

+外存信息读写的时间 d*tio

+内部归并排序所需时间 s*utmg

其中：tis 是为得到一个初始归并段进行的内部排序所需时间的均值；tio 是进行一次外存读/写时间的均值；utmg 是对 u 个记录进行内部归并所需时间；m 为经过内部排序之后得到的初始归并段的个数；s 为归并的趟数；d 为总的读/写次数。由此，上例 10000 个记录利用 2-路归并进行排序所需总的时间为：

10*tis+500*tio+4*10000tmg

其中 tio 取决于所用的外存设备，显然，tio 较 tmg 要大的多。因此，提高排序效率应主要着眼于减少外存信息读写的次数 d。

下面来分析 d 和“归并过程”的关系。若对上例中所得的 10 个初始归并段进行 5-平衡归并(即每一趟将 5 个或 5 个以下的有序子文件归并成一个有序子文件)，则从下图可见，仅需进行二趟归并，外部排序时总的读/写次数便减少至 2×100+100=300，比 2-路归并减少了 200 次的读/写。

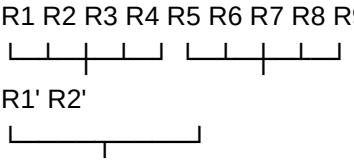


图 10.12

可见，对同一文件而言，进行外部排序时所需读/写外存的次数和归并的趟数 s 成正比。而在一般情况下，对 m 个初始归并段进行 k-路平衡归并时，归并的趟数

可见，若增加 k 或减少 m 便能减少 s。下面分别就这两个方面讨论之。

10.7.2 多路平衡归并的实现

从上式可见，增加 k 可以减少 s，从而减少外存读/写的次数。但是，从下面的讨论中又可发现，单纯增加 k 将导致增加内部归并的时间 utmg。那末，如何解决这个矛盾呢？

先看 2-路归并。令 u 个记录分布在两个归并段上，按 Merge 函数进行归并。每得到归并后的含 u 个记录的归并段需进行 u-1 次比较。

再看 k-路归并。令 u 个记录分布在 k 个归并段上，显然，归并后的第一个记录应是 k 个归并段中关键码最小的记录，即应从每个归并段的第一个记录的相互比较中选出最小者，这需要进行 k-1 次比较。同理，每得到归并后的有序段中的一个记录，都要进行 k-1 次比较。显然，为得到含 u 个记录的归并段需进行(u-1)(k-1)次比较。由此，对 n 个记录的文件进行外部排序时，在内部归并过程中进行的总的比较次数为 s(k-1)(n-1)。假设所得初始归并段为 m 个，则可得内部归并过程中进行比较的总的次数为

k 而减少外存信息读写时间所得效益，这是我们所不希望的。然而，若在进行 k-路归并时利用“败者树”(Tree of Loser)，则可使在 k 个记录中选出关键码最小的记录时仅需进

它不再随 k 的增长而增长。
何谓“败者树”？它是树形选择排序的一种变型。相对地，我们可称图 10.5 和图 10.6 中二叉树为“胜者树”，因为每个非终端结点均表示其左、右子女结点中“胜者”。反之，若在双亲结点中记下刚进行完的这场比赛中的败者，而让胜者去参加更高一层的比赛，便可得到一棵“败者树”。
【例 10.9】

(a) (b)
图 10.13 实现 5-路归并的败者树
图 10.13(a)即为一棵实现 5-路归并的败者树 ls[0…4]，图中方形结点表示叶子结点(也可看成是外结点)，分别为 5 个归并段中当前参加归并的待选择记录的关键码；败者树中根结点 ls[1]的双亲结点 ls[0]为“冠军”，在此指示各归并段中的最小关键码记录为第三段中的记录；结点 ls[3]指示 b1 和 b2 两个叶子结点中的败者即是 b2，而胜者 b1 和 b3(b3 是叶子结点 b3、b4 和 b0 经过两场比赛后选出的获胜者)进行比较，结点 ls[1]则指示它们中的败者为 b1。在选得最小关键码的记录之后，只要修改叶子结点 b3 中的值，使其为同一归并段中的下一个记录的关键码，然后从该结点向上和双亲结点所指的关键码进行比较，败者留在该双亲，胜者继续向上直至树根的双亲。如图 10.13(b)所示。当第 3 个归并段中第 2 个记录参加归并时，选得最小关键码记录为第一个归并段中的记录。为了防止在归并过程中某个归并段变为空，可以在每个归并段中附加一个关键码为最大的记录。当选出的“冠军”记录的关键码为最大值时，表明此次归并已完成。由于实现 k-路归并的败者树
的初始化也容易实现，只要先令所有的非终端结点指向一个含最小关键码的叶子结点，然后从各叶子结点出发调整非终端结点为新的败者即可。
下面程序中简单描述了利用败者树进行 k-路归并的过程，为了突出如何利用败者树进行归并，避开了外存信息存取的细节，可以认为归并段已存在。
【算法 10.15】

```
typedef int LoserTree[k]; /*败者树是完全二叉树且不含叶子，可采用顺序存储结构*/
typedef struct{
KeyType key;
}ExNode,External[k]; /*外结点，只存放待归并记录的关键码*/
void K_Merge(LoserTree *ls,External *b) /*k-路归并处理程序*/
{ /*利用败者树 ls 将编号从 0 到 k-1 的 k 个输入归并段中的记录归并到输出归并段*/
/*b[0]到 b[k-1]为败者树上的 k 个叶子结点，分别存放 k 个输入归并段中当前记录的关键码*/
for(i=0;i<k;i++) input(b[i].key); /*分别从 k 个输入归并段读入该段当前第一个记录的*/
/*关键码到外结点*/
CreateLoserTree(ls); /*建败者树 ls，选得最小关键码为 b[0].key*/
while(b[ls[0]].key!=MAXKEY)
{ q=ls[0]; /*q 指示当前最小关键码所在归并段*/
output(q); /*将编号为 q 的归并段中当前(关键码为 b[q].key 的记录写至输出归并段)*/
input(b[q].key); /*从编号为 q 的输入归并段中读入下一个记录的关键码*/
Adjust(ls,q); /*调整败者树，选择新的最小关键码*/
}
output(ls[0]); /*将含最大关键码 MAXKEY 的记录写至输出归并段*/
}

void Adjust(LoserTree *ls,int s) /*选得最小关键码记录后，从叶到根调整败者树，选下一个最
```

```
小关键码*/
{ /*沿从叶子结点 b[s]到根结点 ls[0]的路径调整败者树*/
t=(s+k)/2; /*ls[t]是 b[s]的双亲结点*/
while(t>0)
{ if(b[s].key>b[ls[t]].key) s<-->ls[t]; /*s 指示新的胜者*/
t=t/2;
}
ls[0]=s;
}
void CreateLoserTree(LoserTree *ls) /*建立败者树*/
{ /*已知 b[0]到 b[k-1]为完全二叉树 ls 的叶子结点存有 k 个关键码，沿从叶子到根的 k 条路径*/
/*将 ls 调整为败者树*/
b[k].key=MINKEY; /*设 MINKEY 为关键码可能的最小值*/
for(i=0;i<k;i++) ls[i]=k; /*设置 ls 中“败者”的初值*/
for(i=k-1;k>0;i--) Adjust(ls,i); /*依次从 b[k-1],b[k-2],…,b[0]出发调整败者*/
}
```

最后要提及一点，k 值的选择并非越大越好，如何选择合适的 k 是一个需要综合考虑的问题。

穷举密码算法

```
//*************************************************************************
//在许多情况下我们需要穷举组合的算法，比如密码词典。
//这个算法的关键是密码下标进位的问题。
//另外本例子中的写文件语句效率比较低，为了降低算法复杂度没有优化。
//如果要提高写文件的效率，可以使用缓冲区，分批写入。
//*********************************************breath.cnpick.com*****
void createpassword()
{
#define passwordmax 8//将生成密码的最大长度
char a[]="0123456789abcdefghijklmnopqrstuvwxyz";//可能的字符
long ndictcount=sizeof(a);//获得密码词典长度
char cpass[passwordmax+2];//将生成的密码
long nminl=1,nmaxl=3;//本例中密码长度从 1-3
long array[passwordmax];//密码词典下标
assert(nminl<=nmaxl && nmaxl<=passwordmax);//容错保证
long nlength=nminl;
register long j,i=0;
bool bnext;
cstdiofile file;
file.open("c:\\dict.txt",cfile::modecreate|cfile::modewrite);
```

```
while(nlength<=nmaxl)
{
for(i=0;i<passwordmax;i++)
array[i]=0;
bnext=true;
while(bnext)
{
for(i=0;i<nlength;i++)
cpass[i]=a[array[i>;
cpass[i]='\0';
file.writestring(cpass);
file.writestring("\n");
for(j=nlength-1;j>=0;j--)//密码指针进位
{
array[j]++;
if(array[j]!=ndictcount-1)break;
else
{
array[j]=0;
if(j==0)bnext=false;
}
}
}
nlength++;
}
file.close();
}
```

如何实现 DES 算法

如何实现 DES 算法

原文：Matthew Fischer

翻译：小榕软件实验室

DES(Data Encryption Standard)算法，于 1977 年得到美国政府的正式许可，是一种用 56 位密钥来加密 64 位数据的方法。DES 算法以被应用于许多需要安全加密的场合。（如：UNIX 的密码算法就是以 DES 算法为基础的）。下面是关于如何实现 DES 算法的语言性描述，如果您要其源代码，可以到 Http//Assassin.yeah.net 下载，后者您有任何问题也可以写信给我（Assassin@ynmail.com）。

1-1、变换密钥

取得 64 位的密钥，每个第 8 位作为奇偶校验位。

1-2、变换密钥。

1-2-1、舍弃 64 位密钥中的奇偶校验位，根据下表（PC-1）进行密钥变换得到 56 位的密钥，在变换中，奇偶校验位以被舍弃。

Permuted Choice 1 (PC-1)
57 49 41 33 25 17 9
1 58 50 42 34 26 18
10 2 59 51 43 35 27
19 11 3 60 52 44 36
63 55 47 39 31 23 15
7 62 54 46 38 30 22
14 6 61 53 45 37 29
21 13 5 28 20 12 4
1-2-2、将变换后的密钥分为两个部分，开始的 28 位称为 C[0]，最后的 28 位称为 D[0]。
1-2-3、生成 16 个子密钥，初始 I=1。
1-2-3-1、同时将 C[I]、D[I]左移 1 位或 2 位，根据 I 值决定左移的位数。见下表
I： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
左移位数： 1 1 2 2 2 2 2 2 1 2 2 2 2 2 2 1
1-2-3-2、将 C[I]D[I]作为一个整体按下表（PC-2）变换，得到 48 位的 K[I]

Permuted Choice 2 (PC-2)
14 17 11 24 1 5
3 28 15 6 21 10
23 19 12 4 26 8
16 7 27 20 13 2
41 52 31 37 47 55
30 40 51 45 33 48
44 49 39 56 34 53
46 42 50 36 29 32
1-2-3-3、从 1-2-3-1 处循环执行，直到 K[16]被计算完成。

2、处理 64 位的数据
2-1、取得 64 位的数据，如果数据长度不足 64 位，应该将其扩展为 64 位（例如补零）
2-2、将 64 位数据按下表变换（IP）
Initial Permutation (IP)
58 50 42 34 26 18 10 2
60 52 44 36 28 20 12 4
62 54 46 38 30 22 14 6
64 56 48 40 32 24 16 8
57 49 41 33 25 17 9 1
59 51 43 35 27 19 11 3
61 53 45 37 29 21 13 5
63 55 47 39 31 23 15 7
2-3、将变换后的数据分为两部分，开始的 32 位称为 L[0]，最后的 32 位称为 R[0]。
2-4、用 16 个子密钥加密数据，初始 I=1。
2-4-1、将 32 位的 R[I-1]按下表（E）扩展为 48 位的 E[I-1]
Expansion (E)
32 1 2 3 4 5

4 5 6 7 8 9
8 9 10 11 12 13
12 13 14 15 16 17
16 17 18 19 20 21
20 21 22 23 24 25
24 25 26 27 28 29
28 29 30 31 32 1
2-4-2、异或 E[I-1]和 K[I]，即 E[I-1] XOR K[I]
2-4-3、将异或后的结果分为 8 个 6 位长的部分，第 1 位到第 6 位称为 B[1]，第 7 位到第 12 位称为 B[2]，依此类推，第 43 位到第 48 位称为 B[8]。
2-4-4、按 S 表变换所有的 B[J]，初始 J=1。所有在 S 表的值都被当作 4 位长度处理。
2-4-4-1、将 B[J]的第 1 位和第 6 位组合为一个 2 位长度的变量 M，M 作为在 S[J]中的行号。
2-4-4-2、将 B[J]的第 2 位到第 5 位组合，作为一个 4 位长度的变量 N，N 作为在 S[J]中的列号。
2-4-4-3、用 S[J][M][N]来取代 B[J]。
Substitution Box 1 (S[1])
14 4 13 1 2 15 11 8 3 10 6 12 5 9 0 7
0 15 7 4 14 2 13 1 10 6 12 11 9 5 3 8
4 1 14 8 13 6 2 11 15 12 9 7 3 10 5 0
15 12 8 2 4 9 1 7 5 11 3 14 10 0 6 13
S[2]
15 1 8 14 6 11 3 4 9 7 2 13 12 0 5 10
3 13 4 7 15 2 8 14 12 0 1 10 6 9 11 5
0 14 7 11 10 4 13 1 5 8 12 6 9 3 2 15
13 8 10 1 3 15 4 2 11 6 7 12 0 5 14 9
S[3]
10 0 9 14 6 3 15 5 1 13 12 7 11 4 2 8
13 7 0 9 3 4 6 10 2 8 5 14 12 11 15 1
13 6 4 9 8 15 3 0 11 1 2 12 5 10 14 7
1 10 13 0 6 9 8 7 4 15 14 3 11 5 2 12
S[4]
7 13 14 3 0 6 9 10 1 2 8 5 11 12 4 15
13 8 11 5 6 15 0 3 4 7 2 12 1 10 14 9
10 6 9 0 12 11 7 13 15 1 3 14 5 2 8 4
3 15 0 6 10 1 13 8 9 4 5 11 12 7 2 14
S[5]
2 12 4 1 7 10 11 6 8 5 3 15 13 0 14 9
14 11 2 12 4 7 13 1 5 0 15 10 3 9 8 6
4 2 1 11 10 13 7 8 15 9 12 5 6 3 0 14
11 8 12 7 1 14 2 13 6 15 0 9 10 4 5 3
S[6]
12 1 10 15 9 2 6 8 0 13 3 4 14 7 5 11
10 15 4 2 7 12 9 5 6 1 13 14 0 11 3 8
9 14 15 5 2 8 12 3 7 0 4 10 1 13 11 6

4 3 2 12 9 5 15 10 11 14 1 7 6 0 8 13
S[7]
4 11 2 14 15 0 8 13 3 12 9 7 5 10 6 1
13 0 11 7 4 9 1 10 14 3 5 12 2 15 8 6
1 4 11 13 12 3 7 14 10 15 6 8 0 5 9 2
6 11 13 8 1 4 10 7 9 5 0 15 14 2 3 12
S[8]
13 2 8 4 6 15 11 1 10 9 3 14 5 0 12 7
1 15 13 8 10 3 7 4 12 5 6 11 0 14 9 2
7 11 4 1 9 12 14 2 0 6 10 13 15 3 5 8
2 1 14 7 4 10 8 13 15 12 9 0 3 5 6 11
2-4-4-4、从 2-4-4-1 处循环执行，直到 B[8]被替代完成。
2-4-4-5、将 B[1]到 B[8]组合，按下表（P）变换，得到 P。
Permutation P
16 7 20 21
29 12 28 17
1 15 23 26
5 18 31 10
2 8 24 14
32 27 3 9
19 13 30 6
22 11 4 25
2-4-6、异或 P 和 L[I-1]结果放在 R[I]，即 R[I]=P XOR L[I-1]。
2-4-7、L[I]=R[I-1]
2-4-8、从 2-4-1 处开始循环执行，直到 K[16]被变换完成。
2-4-5、组合变换后的 R[16]L[16]（注意：R 作为开始的 32 位），按下表（IP-1）变换得到最后的结果。
Final Permutation (IP**-1)
40 8 48 16 56 24 64 32
39 7 47 15 55 23 63 31
38 6 46 14 54 22 62 30
37 5 45 13 53 21 61 29
36 4 44 12 52 20 60 28
35 3 43 11 51 19 59 27
34 2 42 10 50 18 58 26
33 1 41 9 49 17 57 25
以上就是 DES 算法的描述。

入栈与出栈的所有排列可能性

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <bios.h>
#include <string.h>
#include <graphics.h>
/*定义全局变量*/
int pu=0,po=0,t=0;
char tp[130][12];

/*用栈排出入栈出栈的顺序*/
struct train{
  int numb;
  struct train *next;
  };
struct push{
  int a;
  int b;
  char data[24];
  struct push *next;
  };
struct push *creat(struct push *top)
{
  top=(struct push *)malloc(sizeof(struct push));
  top->next=NULL;
  return top;
}
struct push *pup(struct push *top,int a,int b,int n)
{
  struct push *p,*q;
  q=top->next;
  p=(struct push *)malloc(sizeof(struct push));
  p->a=a;
  p->b=b;
  if(q)
    strcpy(p->data,q->data);
  if(q->a<n||!q)
   {
    p->data[a+b-1]='r';
    p->data[a+b]='\0';
   }
  else
   {
```

```
      p->data[a+b-1]='c';
      p->data[a+b]='\0';
    }
   p->next=top->next;
   top->next=p;
return top;
}
struct push *pop(struct push *top)
{
   top=top->next;
   return top;
}
struct push *apaili(struct push *top,int numb)   /*向后移动一个出

命令*/

{
   struct push *q;
   q=top->next;
   if(pu<numb)
    {
      pu++;
      top=pup(top,pu,po,numb);
      top=apaili(top,numb);
     }
    if(po<numb)
     {
       po++;
       top=pup(top,pu,po,numb);
       top=apaili(top,numb);
      }
      return top;
}


struct push *bpaili(struct push *top,int numb) /*移动第一个出命令

到最后*/

{
   int a,b,cir;
   char x[22];
   struct push *p;
```

```
    do{
  p=top->next;
  p=p->next;
  a=p->a;
  b=p->b;
  if(p->data[a+b-1]=='r')
  break;
  else
  {
  do{
     top=pop(top);
     p=top->next;
     a=p->a;
     b=p->b;
   }while(p->data[a+b-1]=='c');
  if(a==1)
    {
     cir=1;
     break;
     }
  top=pop(top);
  a--;
  b++;
  top=pup(top,a,b,numb);
  top->next->data[a+b-1]='c';
  pu=a;
  po=b;
  top=apaili(top,numb);
  strcpy(x,top->next->data);
  if(jc(x))
  {
    strcpy(tp[t],x);
    t++;
  }
  }
     }while(a+b<2*numb);
     if(cir==1)
     return top;
     top=pop(top);
     top=bpaili(top,numb);
}
int jc(char c[22])
```

```
{
  int i=0,k=0;
  if(c[i]!=0)
  do{
     if(c[i]=='r')
     k++;
     if(c[i]=='c')
     k--;
     if(k<0)
     return 0;
     i++;
     }while(c[i]!='\0');
     return 1;
}
/*调用排好的顺序进行入栈与出栈操作*/
struct train *tcreat(struct train *ttop)
{
  ttop=(struct train *)malloc(sizeof(struct train));
  ttop->next=NULL;
  return ttop;
 }
struct train *tpup(struct train *ttop,int data)
{
  struct train *p;
  p=(struct train *)malloc(sizeof(struct train));
  p->numb=data;
  p->next=ttop->next;
  ttop->next=p;
  return ttop;
  }
struct train *tpop(struct train *ttop)
{
 int data;
 struct train *p;
 p=ttop->next;
 data=p->numb;
 ttop=ttop->next;
 printf("%d ",data);
 return ttop;
 }
inso(char c[22],int n)
```

```
{
  int i,k=0,begin[12];
  struct train *ttop;
  ttop=tcreat(ttop);
  for(i=0;i<n;i++)
  begin[i]=i+1;
  i=0;
  if(c[i]!='\0')
  while(c[i]!='\0')
  {
    switch(c[i])
    {
      case 'r': ttop=tpup(ttop,begin[k]);
 k++;
 break;
      case 'c': ttop=tpop(ttop);
 break;
      }
    i++;
    }
}

/*输出所有情况*/
print(int k)
{
 int i,j=0,m;
 for(i=0;i<t;i++)
 {
  if(i%2==0)
  {
   m=8+j;
   gotoxy(5,m);
  }
  j++;
  printf("第%3d 种排列:",i+1);
  inso(tp[i],k);
  gotoxy(32,m);
  if((i+1)%16==0)
  {
   getch();
```

```
    j=0;
    }
 }
}
/*产生随机数功能函数*/
int ram()
{
   int a,b;
   b=random(100);
   if(b<=100)
   a=4;
   if(b<=60)
   a=3;
   if(b<=30)
   a=2;
   if(b<=10)
   a=1;
   return a;
   }
int aram()
{
 int a,b;
 b=random(120);
 if(b<=120)
 a=3;
 if(b<=60)
 a=2;
 if(b<=20)
 a=1;
 if(b<=5)
 a=0;
 return a;
}
/*清屏功能函数*/
void clears(int x,int y,int n,int m)
{
    setfillstyle(1,8);
    bar(x,y,n,m);
}
```

```
/*提示栏信息库*/
tel(int n)
{
 switch(n)
 {
 case 1:clears(459,300,625,465);
   gotoxy(59,21);
   printf("这是输入平台一");
   gotoxy(59,23);
   printf("输入要调用货车厢个数");
         gotoxy(59,25);
         printf("它必须不大于现有个数");
   break;
   case 2:clears(459,300,625,465);
   gotoxy(59,21);
   printf("这是输入平台二");
   gotoxy(59,23);
   printf("你想要调用客车厢个数");
          gotoxy(59,25);
         printf("它必须不大于现有个数");
   break;
   case 3:clears(459,300,625,465);
   gotoxy(59,21);
   printf("这是输入平台三");
   gotoxy(59,23);
   printf("邮件车厢要用到多少个");
         gotoxy(59,25);
         printf("它必须不大于现有个数");
   break;
   case 4:clears(459,300,625,465);
   gotoxy(59,21);
   printf("这里是操作提示框");
```

```
  gotoxy(59,23);
        printf("它可以给你操作提示");
        gotoxy(59,25);
  printf("看清信息后任意键继续");
  break;
  case 5:clears(459,300,625,465);
  gotoxy(59,21);
  printf("正在输出资料");
  gotoxy(59,23);
  printf("任意键查看下一页");
  break;
  case 6:clears(459,300,625,465);
  moveto(460,310);
  setcolor(12);
  outtext("WARING !");
  gotoxy(59,23);
  printf("输入的数字是非法的");
  gotoxy(59,25);
  printf("看清信息，任意键重输");
  break;
  case 7:clears(459,300,625,465);
  moveto(460,310);
  setcolor(12);
  outtext("WARING !");
  gotoxy(59,23);
  printf("输入的数字错误的");
  gotoxy(59,25);
  printf("看清信息，按键重输");
  break;
  case 8:clears(459,300,625,465);
  moveto(460,310);
  setcolor(12);
  outtext("ok!");
  gotoxy(59,23);
```

```
   printf("车厢已经排列好");
   gotoxy(59,25);
   printf("按键查看所有排列顺序");
   break;
   case 9:clears(459,300,625,465);
   gotoxy(59,21);
   printf("这是输入平台四");
   gotoxy(59,23);
   printf("输入你要的排列的序号");
   gotoxy(59,25);
   printf("选择后");
   gotoxy(59,27);
   printf("系统将排列演示");
   break;
  case 10:clears(459,300,625,465);
   gotoxy(59,21);
   printf("正在根据你的选择");
   gotoxy(59,23);
   printf("进行调度演示");
   break;
  case 11:clears(459,300,625,465);
   gotoxy(59,21);
   printf("车厢调度成功");
   gotoxy(59,23);
   printf("顺序如右图所示");
   break;
  case 12:clears(459,300,625,465);
   gotoxy(59,21);
   printf("又有车头进站");
   gotoxy(59,23);
   printf("要为它安排车厢吗");
```

```
   gotoxy(59,25);
   printf("(Y/N)");
   break;
    }
}
/*车头信息库*/
int atri(void)
{
   gotoxy(60,5);
   printf("这是一辆 AT 型车头");
   gotoxy(60,6);
   printf("最多可以带 2 个车厢");
   return 2;
}
int btri(void)
{
   gotoxy(60,5);
   printf("这是一辆 BT 型车头");
   gotoxy(60,6);
   printf("最多可以带 3 个车厢");
   return 3;
 }
int ctri(void)
{
   gotoxy(60,5);
   printf("这是一辆 CT 型车头");
   gotoxy(60,6);
   printf("最多可以带 4 个车厢");
   return 4;
 }
int dtri(void)
{
   gotoxy(60,5);
   printf("这是一辆 DT 型车头");
   gotoxy(60,6);
```

```
   printf("最多可以带 5 个车厢");
   return 5;
}

/*车厢信息库*/
truk(int a,int b)
{
   setcolor(BROWN);
   rectangle(a,b,a+70,b+20);
   setfillstyle(6,6);
   bar(a+1,b+1,a+69,b+19);
}
passage(int a,int b)
{
   setcolor(BROWN);
   rectangle(a,b,a+70,b+20);
   setfillstyle(2,1);
   bar(a+1,b+1,a+69,b+19);
}
mail(int a,int b)
{
   setcolor(BROWN);
   rectangle(a,b,a+70,b+20);
   setfillstyle(2,WHITE);
   bar(a+1,b+1,a+69,b+19);
}
carbox()
{
   truk(460,160);
   passage(460,200);
   mail(460,240);
}
/*车厢个数显示信息*/
ttel(a,b,c)
{
   gotoxy(69,11);
   printf("货厢%d 个",a);
   gotoxy(69,14);
```

```
   printf("客厢%d 个",b);
   gotoxy(69,16);
   printf("邮厢%d 个",c);
}
/*图象*/
door(int a,int b,int c)
{
 setcolor(YELLOW);
 rectangle(1,1,637,478);
 rectangle(10,27,628,468);
 line(1,20,637,20);
 setfillstyle(1,3);
 bar(2,2,636,19);
 setfillstyle(1,8);
 bar(11,28,627,467);
 moveto(7,7);
 settextstyle(5,0,4);
 setcolor(5);
 outtext("J01126 WANGBIN");
 setcolor(YELLOW);
 rectangle(450,27,457,468);
 rectangle(11,400,450,407);
 rectangle(457,120,628,138);
 rectangle(457,280,628,298);
 setfillstyle(1,9);
 bar(451,28,456,467);
 bar(12,401,449,406);
 bar(2,21,636,26);
 bar(2,21,9,477);
 bar(2,469,636,477);
 bar(629,21,636,477);
 line(458,45,627,45);
 setfillstyle(1,4);
 bar(458,28,627,44);
 bar(458,121,627,137);
 bar(458,281,627,297);
 setcolor(GREEN);
 moveto(460,35);
 outtext("STATION'S INFORMATION");
```

```
 moveto(460,128);
 outtext("COMPATE'S INFORMATION");
 moveto(465,287);
 outtext("MESSAGE");
 carbox();
 ttel(a,b,c);
 }
int nform(void)
{
   int numb;
   setcolor(YELLOW);
   rectangle(150,150,350,230);
   rectangle(153,153,347,227);
   line(150,150,155,155);
   line(150,230,155,225);
   line(350,230,345,225);
   line(350,150,345,155);
   setfillstyle(1,9);
   bar(154,154,346,226);
   moveto(170,170);
   outtext("INPUT THE NUMBER ");
   setfillstyle(1,8);
   bar(170,190,290,210);
   gotoxy(30,13);
   scanf("%d",&numb);
   return numb;
 }
void initt()
{
   int graphdrive=IBM8514LO;
   int graphmode=IBM8514HI;
   initgraph(&graphdrive,&graphmode," ");
   setbkcolor(8);
   }
/*动态信息*/
int headtrain()
{
   int i,z;
   i=ram();
   switch(i)
   {
```

```
    case 1:clears(459,47,625,110);
    z=atri();
    break;
    case 2:clears(459,47,625,110);
    z=btri();
    break;
    case 3:clears(459,47,625,110);
    z=ctri();
    break;
    case 4:clears(459,47,625,110);
    z=dtri();
    break;
    }
    return z;
}
ptrain(int a,int b,int c)
{
 int i;
 if(a==0)
 for(i=0;i<c;i++)
 truk(15+(b-c+i)*75,425);
 if(a==1)
 for(i=0;i<c;i++)
 passage(15+(b-c+i)*75,425);
 if(a==2)
 for(i=0;i<c;i++)
 mail(15+(b-c+i)*75,425);
}
/*主键面*/
first()
{
 int k,i,m,tr,pa,ma,ci;
 char c[22];
 struct push *top,*p;
 clrscr();
 tr=aram();
 do{
 pa=aram();
 }while(pa==tr);
 do{
 ma=aram();
```

```
}while(ma==pa||ma==tr);
door(tr,pa,ma);
m=headtrain();
tel(4);
do{
    getch();
    k=0;
    for(i=0;i<3;i++)
    {
     tel(i+1);
     ci=nform();
     if((i==0&&ci>tr)||(i==1&&ci>pa)||(i==2&&ci>ma))
     {
      i--;
      tel(7);
      getch();
      continue;
      }
    k+=ci;
    if(k<=m)
    {
    ptrain(i,k,ci);
    savetrain(i,k,ci);
    }
    else
    {
    tel(6);
    clears(12,420,380,450);
    break;
    }
    }
}while(k>m);
clears(140,140,360,240);
tel(8);
getch();
top=creat(top);
top=apaili(top,k);
strcpy(tp[t],top->next->data);
t++;
if(k!=1)
top=bpaili(top,k);
tel(5);
```

```
 print(k);
 getch();
}
/*动画*/
struct run{
int data;
char kind;
}ttrai[5];
savetrain(int i,int j,int c)
{
 int a;
 if(i==0&&c!=0)
 for(a=j-c;a<j;a++)
 {
 ttrai[a].data=a+1;
 ttrai[a].kind='t';
 }
 if(i==1&&c!=0)
 for(a=j-c;a<j;a++)
 {
   ttrai[a].data=a+1;
   ttrai[a].kind='p';
   }
 if(i==2&&c!=0)
 for(a=j-c;a<j;a++)
 {
   ttrai[a].data=a+1;
   ttrai[a].kind='m';
   }
}
runtrain(int k)
{
 int i=0,a;
 char c;
 c=ttrai[k].kind;
 switch(c)
 {
   case 't':for(a=0;a<210;a+=3)
     {
       clears(12,190,440,275);
       if(a<=160)
```

```
       truk(365-a,200);
       else
       truk(205,39+a);
       delay(1900);
      }
    break;
  case 'p':for(a=0;a<210;a+=3)
    {
       clears(12,190,440,275);
       if(a<=160)
       passage(365-a,200);
       else
       passage(215,39+a);
       delay(1900);
      }
    break;
  case 'm':for(a=0;a<210;a+=3)
    {
     clears(12,190,440,275);
     if(a<=160)
     mail(365-a,200);
     else
     mail(205,39+a);
     delay(1900);
    }
    break;
    }
}

bruntrain(char c)
{
 int a;
 switch(c)
 {
  case 't':for(a=0;a<220;a+=3)
    {
       clears(12,190,440,275);
       if(a<50)
       truk(205,249-a);
       else
       truk(258-a,200);
       delay(1900);
```

```
      }
      break;
   case 'p':for(a=0;a<220;a+=3)
     {
       clears(12,190,440,275);
       if(a<50)
       passage(205,249-a);
       else
       passage(258-a,200);
       delay(1900);
      }
     break;
   case 'm':for(a=0;a<220;a+=3)
     {
      clears(12,190,440,275);
      if(a<50)
      mail(205,249-a);
      else
      mail(258-a,200);
      delay(1900);
     }
     break;
     }
}
/*演示时的键面*/
face()
{
   setcolor(10);
   rectangle(400,170,449,189);
   rectangle(11,170,60,189);
   setcolor(YELLOW);
   rectangle(200,276,290,400);
   setfillstyle(1,11);
   bar(401,172,448,187);
   bar(12,172,59,187);
   setfillstyle(1,7);
   bar(201,277,289,399);
   setcolor(RED);
   moveto(411,178);
   outtext("ENTER");
   moveto(18,178);
```

```
  outtext("OUT");
  moveto(220,340);
  outtext("STATION");
}

ztrain(int k)
{
  int i=0,j=0,a=0,e=0,b,d;
  char c[12];
  struct run tpu[5];
  strcpy(c,tp[k]);
  face();
  if(c[i]!='\0')
  do
  {
   switch(c[i])
   {
     case 'r':clears(15+j*73,424,88+j*75,445);
       for(b=0;b<10;b++)
       delay(10000);
       runtrain(j);
       clears(12,190,440,275);
       for(b=0;b<10;b++)
       delay(10000);
       tpu[e].data=ttrai[j].data;
       tpu[e].kind=ttrai[j].kind;
       j++;
       e++;
       break;
     case 'c':ttrai[a].data=a+1;
       ttrai[a].kind=tpu[e-1].kind;
       bruntrain(tpu[e-1].kind);
       clears(12,190,440,275);
       for(b=0;b<5;b++)
       delay(10000);
       if(ttrai[a].kind=='t')
       d=0;
       if(ttrai[a].kind=='p')
       d=1;
       if(ttrai[a].kind=='m')
       d=2;
       ptrain(d,a+1,1);
```

```
        for(b=0;b<5;b++)
        delay(10000);
        a++;
        e--;
        break;
      }
    i++;
 }while(c[i]!='\0');
}
/*开始动画*/
begi()
{
 int i;
 void *trp;
 setcolor(7);
 rectangle(50,330,130,350);
 rectangle(50,310,80,330);
 rectangle(115,320,120,330);
 rectangle(0,359,637,362);
 for(i=0;i<4;i++)
 {
   if(i>1)
   circle(63+i*18,352,6);
   else
   circle(63+i*18,350,8);
 }
 trp=malloc(sizeof(imagesize(49,299,131,357)));
 getimage(49,299,131,357,trp);
 for(i=0;i<500;i++)
 putimage(50+i,300,trp,COPY_PUT);
 clrscr();
 setcolor(YELLOW);
 rectangle(100,125,500,300);
 rectangle(120,150,480,280);
 setfillstyle(1,9);
 bar(101,126,499,299);
 setfillstyle(1,8);
 bar(122,152,478,278);
 gotoxy(20,12);
 printf("有一辆车头进站，请立即安排车厢");
```

```
 gotoxy(20,14);
 printf("G 键进入系统，E 键退出");
}

atta()
{
 int i;
 setcolor(13);
 for(i=0;i<10;i++)
 {
 moveto(100,200);
 clears(11,35,449,400);
 delay(90000);
 outtext("HAVE A NEW MESSAGE");
 delay(90000);
 }
}

main()
{
 int x,i;
 char c;
 initt();
 begi();
 c=getch();
 if(c=='g'||c=='G')
 {
   while(1)
   {
   first();
   clears(12,35,430,380);
   do{
   tel(9);
   x=nform();
   if(x<=0||x>t)
    {
     tel(7);
     getch();
    }
   }while(x<=0||x>t);
   clears(12,35,430,380);
```

```
  tel(10);
  ztrain(x-1);
  tel(11);
  for(i=0;i<30;i++)
  delay(10000);
  tel(12);
  atta();
  c=getch();
  if(c=='n'||c=='N')
  exit(0);
  pu=0;
  po=0;
  t=0;
  }
 }
 else
 exit(0);
}
```

三维图形的消隐算法分析

---- 摘要造型是计算机三维图形处理的基础，而消隐则是三维造型的关键。本文剖析了当前在CAD三维图形处理中最主要的8种物体空间消隐算法和4种图象空间消隐算法。

---- 关键词造型、消隐、物体空间法、图象空间法

---- 分类号

---- 造型（modeling）是计算机三维图形处理的基础，而消除隐藏面（hidden surface，简称消隐）则是三维造型的关键。所谓消隐就是不画出即隐藏从当前观察点看不见的三维模型表面。消隐算法的核心就是判断三维模型的表面是否可见。

---- 抽象来看，一种消隐算法可以看作一个五元组，即

HA = (I, O, D, P, S)

---- 其中，I为要进行消隐处理的三维对象的集合；

---- O为经过消隐处理的二维对象的集合；

---- D为进行消隐处理时所采用的数据结构；

---- P为进行消隐处理所需基本操作过程的集合，主要包括

分类、排序

三维坐标变换

透视投影变换

基本图形元素间的求交计算

两个区域重叠判断

点与区域的包含测试

面的朝向测试

---- S为消隐策略，即规定P中各基本操作过程被采用的先后次序。

---- 因此，设计消隐算法时应考虑上述五个要素及它们之间的相

互关系。

---- 在计算机图形学中，为了简化算法，一般是利用多面体去逼近曲面体，因此多面体的消隐算法是曲面体的基础。本文的消隐算法讨论主要是基于多面体的消隐问题，对曲面体进行多次多面体近似，对每一多面体运用多面体消隐算法就能实现曲面体的消隐。

---- 基于B-rep 模型（Boundary Representative Model）和CSG 模型（Constructive Solid Geometry Model）的三维造型消隐算法可以分为两大类，即物体空间法和图象空间法。物体空间法利用三维环境信息或三维视图（主要使用三维观察坐标，有时也使用三维世界坐标）来消除隐藏面，即根据空间中各物体三维模型的几何关系，来判断哪些表面可见，哪些表面不可见。图象空间法基于物体三维模型的二维显示图形（使用二维显示坐标）来确定物体或表面与观察点的远近关系，从而判断哪些表面遮挡了其它表面。

---- 本文将分析当前在CAD 三维图形处理中最主要的12 中消隐算法，其中8 种属于物体空间法，4 种属于图象空间法。在实际处理中，由于物体表面形态的复杂性和提高消隐处理的效率，通常都是结合使用多种消隐算法来完成物体的消隐处理的。

---- 一、物体空间法

---- 物体空间法是在三维坐标系中，通过分析物体模型间的几何关系，如物体的几何位置、与观察点的相对位置等，来进行隐藏面判断的消隐算法。世界坐标系是描述物体的原始坐标系，物体的世界坐标描述了物体的基本形状。为了更好地观察和描述物体，经常需要对其世界坐标进行平移和旋转，而得到物体的观察坐标。物体的观察坐标能得到描述物体的更好视角，所以物体空间法通常都是在观察坐标系中进行的。观察坐标系的原点一般即是观察点。

---- 物体空间法消隐包括两个基本步骤，即三维坐标变换和选取适当的隐藏面判断算法。

---- （一）三维坐标变换

---- 选择合适的观察坐标系不但可以更好地描述物体，而且可以大大简化和降低消隐算法的运算。因此，利用物体空间法进行消隐的第一步往往是将物体所处的坐标系转换为适当的观察坐标系。这需要对物体进行三维旋转和平移变换。

---- 设物体顶点的原始坐标为(x, y, z)，变化后的观察坐标为(x*, y*, z*)，则

---- 1．旋转变换

物体绕Y 轴旋转的角度α的三维旋转变换公式为

[x* y * z* 1]= [x y z] [cosα sinα 0 0

-sinα cosα 0 0

0 0 1 0

0 0 0 1]

物体绕Z 轴旋转的角度β的三维旋转变换公式为

[x* y * z* 1]= [x y z] [cosβ 0 -sinβ 0
0 1 0 0
sinβ 0 cosβ 0
0 0 0 1]

物体绕X轴旋转的角度γ的三维旋转变换公式为
[x* y * z* 1]= [x y z] [1 0 0 0
0 corγ sinγ 0
0 -sinγ cosγ 0
0 0 0 1]

将上述表达式展开后可得如下公式
{ x'=cosα×x-sinα×z;
y'=cosβ×y-sinβ×x';
z'=sinα×x+cosα×z;
x*=cosβ×x'+sinβ×y;
y*=sinγ×z'+cosγ×y';
z*=cosγ×z'-sin γ×y';

---- 其中，x'、y' 和z' 是运算中间变量。
---- 2．平移变换
---- 三维平移变换公式如下
[x* y * z* 1]= [x y z 1] [1 0 0 0
0 1 0 0
0 0 1 0
Δx Δy Δz 1]

---- 展开后即得
[x*=x+Δx
y*=y+Δy
z*=z+Δz

---- 其中Δx、Δy、Δz分别为物体沿X、Y、Z轴的平移量。
---- 二）消隐算法
---- 常用的物体空间消隐算法包括平面公式法、背面消除法、径向预排序法、径向排序法、隔离平面法、深度排序法、光线跟踪法和分解法。其中前四种算法最常用，它们的基础都是背面消隐原理。所谓背面消隐原理，即是相对观察点来说朝向后面的物体表面是不可见的，应被隐藏。
---- 1．平面公式法
---- 根据解析几何原理，通过标准的平面方程可以判断给定点是在平面的正面还是背面。平面公式法利用此原理来判断观察点位于物体表面的哪一面，如位于背面一侧，则表面不可见，应被消隐；反之则可见。

---- 对物体得任意表面，可将其划分为若干个平面，在根据平面上任意三点的坐标可以求得其平面方程。标准得平面方程为
Ax+By+Cz+D = 0;
---- 其中A、B、C、D为决定平面得常数。如果（x1, y1, z1）、（x2, y2, z2）、（x3, y3, z3）为平面上已知得三点坐标，则可求得A、B、C、D如下：
{ A=y1(x2-x3)+y2(z3-z1)+y3(z1-z2);
B=z1(x2-x3)+z2(x3-x1)+z3(x1-x2);
C=x1(y2-y3)+x2(y3-y1)+x3(y1-y2);
D=-x1(y2z3-y3z2)-x2(y3z1-y1z3)-x3(y1z2-y2z1);

---- 设观察点坐标为（x, y, z），如果
Ax+By+Cz+D = 0，则观察点（x, y, z）位于平面上；
Ax+By+Cz+D > 0，则观察点（x, y, z）位于平面背面一侧，平面不可见，应被隐藏；
Ax+By+Cz+D < 0，则观察点（x, y, z）位于平面正面一侧，平面是可见面，应被画出。
---- 通过对物体进行适当旋转和平移后，可将物体变换到以观察点为原点得观察坐标系中，如果在观察坐标系中求得了平面得方程Ax+By+Cz+D = 0，将观察点坐标（0，0，0）代入上面得判断准则，则可得出如下得简单判据：
D > 0，则平面不可见，应被隐藏；
D < 0，则平面是可见面，应被画出。
---- 平面公式法算法简便，是在实际中使用最频繁得消隐算法。但它只能用于凸面体得消隐，而不适用于凹面体消隐。
---- 2．背面消除法
---- 背面消除法是直接运用背面消隐原理的消隐算法。在数学上，物体表面的法向量即是表面的朝向，因此，法向量方向背向观察点的物体表面都应被消隐。

---- 表面的法向量是否背向观察点可以通过表面法向量与视向量的点积来决定。如图1所示，设经坐标变换后，坐标系的原点O即是观察点，空间中任意平面ABC的法向量为，法向量为与平面的交点为P，则从向量OP即是平面ABC的视向量。
---- 如果> 0，则物体表面是可见的朝向观察点的面；如果，则物体表面是不可见的背向观察点的面，应被消隐。
---- 设θ为向量和之间的夹角，视向量的长度为线段OP的长度|OP|，则根据向量点积的定义可知=|OP|||cos θ。如果> 0，则cos θ> 0（即> θ> 0）；反之，如果，则cos θ 0（即 θ）。
---- 因此，背面消除法的判据简化为：
cos θ 0，则物体表面不可见，应被消隐；
cos θ> 0，则物体表面可见，应被画出。
---- 根据平面法向量的定义可知，在平面上按逆时针方向选取P1（x1, y1, z1）、P2（x2, y2, z2）、P3（x3, y3, z3）三点，则
---- (公式略)

---- 其 中：
---- (公式略)
---- 经 过 投 影 变 化 后，视 向 量 与Z 轴 是 平 行 的，因 此 向 量 和 之 间 的 夹 角 θ 即 为Z 轴 与 向 量 的 夹 角， 所 以
---- 由 于||>0，所 以cos θ 的 正 负 取 决 于C，因 此 背 面 消 除 法 的 判 据 转 化 化 为：
---- (公式略)
C 0， 则 物 体 表 面 不 可 见， 应 被 消 隐；
C > 0， 则 物 体 表 面 可 见， 应 被 画 出。
---- 3 ． 径 向 预 排 序 法
---- 径 向 预 排 序 法 根 据 物 体 在 三 维 坐 标 系XY 平 面 中 的 角 位 置 来 判 断 哪 些 物 体 挡 住 了 其 它 物 体， 物 体 的 哪 些 表 面 挡 住 了 其 它 表 面。 对 具 有 相 同 角 位 置 的 物 体 或 表 面， 与 观 察 点 较 近 的 将 挡 住 较 远 的。 如 图2 所 示。

图2 径 向 预 排 序 法 示 例
---- 径 向 预 排 序 法 消 隐 的 要 点 是 先 对 物 体 及 物 体 的 表 面 进 行 由 远 及 近 的 排 序，对 具 有 相 同 角 位 置 的 物 体 或 表 面，先 画 较 远 的，后 画 较 近 的， 这 样 如 果 较 近 的 物 体 或 表 面 挡 住 了 较 远 的 物 体 或 表 面，则 被 遮 挡 的 部 分 被 覆 盖 而 实 现 消 隐。但 对 具 有 不 同 角 位 置 的 物 体 或 表 面， 先 画 哪 一 个 可 根 据 需 要 来 决 定。 如 果 存 在 凹 面 物 体 的 消 隐，一 般 应 先 画 物 体 中 心 部 分，再 画 物 体 的 两 侧，以 正 确 地 表 现 互 相 重 叠 的 凹 面 模 型。
---- 径 向 预 排 序 法 可 以 对 任 意 形 状 的 物 体 进 行 消 隐 处 理。 但 需 要 预 先 知 道 观 察 角 度， 并 根 据 角 位 置 对 物 体 的 画 图 顺 序 预 先 排 序。 而 且 构 造 模 型 的 编 码 受 到 这 种 排 序 的 限 制，模 型 不 能 进 行 旋 转 变 换。
---- 4 ． 径 向 排 序 法
---- 径 向 排 序 法 是 对 径 向 预 排 序 法 的 改 进 算 法， 使 得 构 造 模 型 的 编 码 能 根 据 观 察 角 度 的 变 化，来 自 动 调 整 物 体 或 表 面 的 远 近 顺 序 即 画 图 顺 序， 以 实 现 对 模 型 的 旋 转 变 换， 以 便 能 从 不 同 的 角 度 来 观 察 物 体。 算 法 需 要 检 测 旋 转 变 换 的 角 度， 并 随 角 度 的 变 化 而 调 整 物 体 或 表 面 的 远 近 顺 序。
---- 5 ． 隔 离 平 面 法
---- 隔 离 平 面 法 主 要 用 于 多 个 物 体 之 间 的 消 隐 处 理， 其 基 础 是 平 面 公 式 法。 其 基 本 原 理 是， 在 需 要 进 行 消 隐 处 理 的 两 个 物 体 之 间 建 立 一 个 虚 拟 平 面，并 根 据 平 面 公 式 法 判 断 出 两 个 物 体 分 别 位 于 该 平 面 的 哪 一 侧， 以 及 该 平 面 的 哪 一 侧 朝 向 观 察 点， 则 可 以 推 论 得 到 位 于 平 面 朝 向 观 察 点 一 侧 的 物 体 离 观 察 点 较 近，将 遮 挡 位 于 平 面 背 向 观 察 点 一 侧 的 物 体。即 位 于 平 面 背 向 观 察 点 一 侧 的 物 体 应 被 首 先 画 出， 且 应 进 行 消 隐。
---- 6 ． 深 度 排 序 法
---- 深 度 排 序 法 也 是 主 要 用 于 分 析 多 个 物 体 之 间 是 否 存 在 表 面 遮 挡 的 消 隐 算 法。 其 原 理 是 比 较 不 同 物 体 或 表 面 的 表 示 远 近 的z 坐

标，在观察点位于原点的观察坐标系中，|z| 值越大的物体或表面离观察点越远，被消隐的可能性越大，应先画出；|z| 值越小的物体或表面离观察点越近，将可能遮挡较远的物体或表面，应后画出。

---- 深度排序法需要用到深度信息和绘图顺序，通常用于模型数据中包含深度信息和绘图顺序的物体造型。

---- 7．光线跟踪法

---- 光线跟踪法的基本原理是，人能看见物体是因为物体能反射光，因此，跟踪从光源发出的光线，光线投射到物体上，再从物体反射到观察点，在光线轨迹中离观察点最近的物体表面将遮挡其它物体表面。

---- 光线跟踪法需要分析物体表面的每一点的光反射状态，因此需要的内存空间较大，运算速度也较慢。但这种方法可同时生成物体的光照模型，产生的消隐效果和真实感都很好。

---- 8．分解法

---- 分解法是对CSG 模型的一种消隐算法，首先将复杂物体分解为一系列的立方体，离观察点近的立方体将遮挡远的立方体，从而实现消隐。分解法算法复杂，需要的内存空间大，速度也满，近仅用于一些特殊的场合。

---- 二、图象空间法

---- 图象空间法基于物体三维模型的二维显示图形来确定物体或表面上的每一点与观察点的远近关系，从而判断哪些表面遮挡了其它表面。为了获得三维物体的二维显示图形，在对物体进行旋转和平移变化后，还需对物体进行透视投影变换。

图3 透视投影变换示意

---- 如图3 所示，三维空间中点P(x, y, z) 由透视点E 沿Z 轴透视投影变换到XOY 二维平面中的点P*(x*, y*, z*)。设E 点距原点O 的距离为r，则透视投影变换公式为：

----（x*, y*, z*, 1）=（x, y, z, 1）MPeMPr

---- 式中MPe、MPr 分别为透视变换矩阵和投影变换矩阵，它们的表达式如下：

---- (公式略)

---- 将透视投影变换公式展开后可得：

---- (公式略)

---- 1．Z 缓冲区法

---- Z 缓冲区法首先建立一个大的缓冲区，用来存储三维物体沿Z 轴透视投影而得到的二维图形的所有象素的值，因此叫做Z 缓冲区。Z 缓冲区的单元个数与屏幕上象素点的个数相同，也和帧缓冲区的单元个数相同，而且它们之间是一一对应的。Z 缓冲区每个单元的大小取决于图形在观察坐标系中Z 方向的变化范围。Z 缓冲区的每个单元的值是对应象素点所对应的物体表面点的Z 坐标值。

---- 利用Z 缓冲区法进行消隐和造型的过程就是对屏幕中每一点进行判断并给帧缓冲区和Z 缓冲区中相应单元进行赋值的过程。现用形式化语言描述该算法如下：

Z 缓冲区消隐算法
{
1）将帧缓冲区各单元的值置为背景色值;
2）将 Z 缓冲区各单元的值置为 Z 坐标可能出现的最大值;
3）循环：对每一物体
{
循环：对物体每一面的每一点（x, y, z）
{
i)对（x, y, z）做透视投影变换，得到变换后的 X、Y 坐标（x*, y*）；
ii)如果 Z 缓冲区中（x*, y*）对应单元的值小于 z，则
{
a)将 Z 缓冲区中（x*, y*）对应单元的值置为 z;
b)将帧缓冲区中（x*, y*）对应单元的值置为点
（x, y, z）的属性值（通常是亮度、颜色值或颜色查找表的索引值）;
}
iii)如果 Z 缓冲区中（x*, y*）对应单元的值大于 z，则
{
a)说明目前帧缓冲区中（x*, y*）对应单元的
所表示的物体上点比点（x, y, z）
更接近观察点，即点（x, y, z）应被消隐；
b)将 Z 缓冲区和帧缓冲区中（x*, y*）对应单元的值均保持不变；
}
}
}
4）循环：对屏幕上每一点（x*, y*）
根据帧缓冲区中（x*, y*）对应单元的值画出象素点。
}
---- Z 缓冲区消隐算法简单、可靠，而且消隐和表现效果很好。但需要的内存容量大，运算复杂，费时。
---- 2．扫描线法
---- 扫描线法是对 Z 缓冲区法进行改进而派生出来的消隐算法。为了克服 Z 缓冲区法需要分配与屏幕上象素点的个数相同单元的巨大内存这一缺点，可以将整个屏幕分成若干区域，一个区一个区地进行处理，这样可以将 Z 缓冲区的单元个数减少为屏幕上一个区域的象素点的个数。将屏幕的一行作为这样的区域，便得到了扫描线法，又称为扫描线 Z 缓冲区法，Z 缓冲区的单元个数仅为屏幕上一行的象素点的个数。
---- 扫描线法的形式化语言描述如下：
扫描线消隐算法
{
1）循环：从上到下对屏幕中每一条扫描线
{
i)将帧缓冲区对应行各单元的值置为背景色值;
ii)将 Z 缓冲区各单元的值置为 Z 坐标可能出现的最大值;

iii)循环：对每一物体
{
循环：对物体每一表面
{
如果当前扫描线与当前物体表面相交，则
{
循环：扫描线与当前物体表面的交点是成对的，
对每对交点之间的每一点（x, y, z）
{
A)对（x, y, z）做透视投影变换，
得到变换后的 X、Y 坐标（x*, y*）；
B)如果 Z 缓冲区中（x*, y*）对应单元的值小于 z，则
{
a)将 Z 缓冲区中（x*, y*）对应单元的值置为 z;
b)将帧缓冲区中（x*, y*）对应单元的值置为点（x, y, z）
的属性值（通常是亮度、颜色值或颜色查找表的索引值）;
}
C)如果 Z 缓冲区中（x*, y*）对应单元的值大于 z，则
{
a)说明目前帧缓冲区中（x*, y*）对应单元的所表示
的物体上点比点（x, y, z）更接近观察点，即点（x, y, z）应被消隐；
b)将 Z 缓冲区和帧缓冲区中（x*, y*）对应单元的值均保持不变；
}
}
}
}
}
iv)循环：对屏幕上每一点（x*, y*）
根据帧缓冲区中（x*, y*）对应单元的值画出象素点。
}
}
---- 3 ． 视 线 投 射 法
---- 视 线 投 射 法 的 基 本 原 理 是 把 物 体 的 二 维 显 示 图 像 看 成 是 从 眼 睛 到 物 体 的 视 线 把 物 体 的 可 见 点 投 射 到 显 示 屏 上 的 投 影。该 算 法 的 形 式 化 语 言 描 述 如 下：
视线投射消隐算法
{
1）循环：对屏幕上每一象素（x*, y*）
{
确定经过视点和象素（x*, y*）的直线 Ray；
判断直线 Ray（x, y）与物体是否相交；
如果存在交点，则
{

实用算法(基础算法-递推法-01)

有一类试题，每相邻两项数之间的变化有一定的规律性，我们可将这种规律归纳成如下简捷的递推关系式：

$F_n=g(F_{n-1})$

这就在数的序列中，建立起后项和前项之间的关系，然后从初始条件(或最终结果)入手，一步步地按递推关系递推，直至求出最终结果(或初始值)。很多程序就是按这样的方法逐步求解的。如果对一个试题，我们要是能找到后一项与前一项的关系并清楚其起始条件(最终结果)，问题就好解决，让计算机一步步算就是了，让高速的计算机做这种重复运算，可真正起到“物尽其用”的效果。

递推分倒推法和顺推法两种形式。一般分析思路：

```
if 求解条件 F1
  then begin{倒推}
    由题意(或递推关系)确定最终结果 Fa;
    求出倒推关系式 Fi-1=g'(Fi);
    i=n;{从最终结果 Fn 出发进行倒推}
    while  当前结果 Fi 非初始值 F1 do 由 Fi-1=g(F1)倒推前项;
    输出倒推结果 F1 和倒推过程;
    end {then}
  else begin{顺推}
    由题意(或顺推关系)确定初始值 F1(边界条件);
    求出顺推关系式 F1=g(Fi-1);
    i=1;{由边界条件 F1 出发进行顺推}
    while  当前结果 Fi 非最终结果 Fn do 由 Fi=g(Fi-1)顺推后项;
    输出顺推结果 Fn 和顺推过程;
  end; {else}
```

一、倒推法

所谓倒推法，就是在不知初始值的情况下，经某种递推关系而获知问题的解或目标，再倒推过来，推知它的初始条件。因为这类问题的运算过程是一一映射的，故可分析得其递推公式。然后再从这个解或目标出发，采用倒推手段，一步步地倒推到这个问题的初始陈述。

下面举例说明。

[例 1] 贮油点

一辆重型卡车欲穿过 1000 公里的沙漠，卡车耗油为 1 升/公里，卡车总载油能力为 500 公升。显然卡车一次是过不了沙漠的。因此司机必须设法在沿途建立几个储油点，使卡车能顺利穿越沙漠，试问司机如何建立这些储油点？每一储油点

应存多少油，才能使卡车以消耗最少油的代价通过沙漠？

算法分析：

编程计算及打印建立的贮油点序号，各贮油点距沙漠边沿出发的距离以及存油量。

No.	Distance(k.m.)	oil(litre)
1	X X	X X
2	X X	X X
3	X X	X X
...	.....	.....

设 dis[i]　为第 i 个贮油点至终点(i=0)的距离；

oil[i]　为第 i 个贮油点的存贮油量；

我们可以用倒推法来解决这个问题。从终点向始点倒推，逐一求出每个贮油点的位置及存油量。

下图表示倒推时的返回点：

从贮油点 i 向贮油点 i+1 倒推的策略是，卡车在点 i 和点 i+1 间往返若干次。卡车每次返回 i+1 处时正好耗尽 500 公升汽油，而每次从 i+1 出发时又必须装足 500 公升汽油。两点之间的距离必须满足在耗油最少的条件下使 i 点贮足 i*500 分升汽油的要求(0<=i<=n-1)。具体地讲，第一个贮油点 i=1 应距终点 i=0 处 500km 且在该处贮藏 500 公升汽油，这样才能保证卡车能由 i=1 处到达终点 i=0 处，这就是说

dis[1]=500　　oil[1]=500;

为了在 i=1 处贮藏 500 公升汽油，卡车至少从 i=2 处开两趟满载油的车至 i=1 处。所以 i=2 处至少贮有 2*500 公升汽油，即 oil[2]=500*2=1000。另外，再加上从 i=1 返回至 i=2 处的一趟空载，合计往返 3 次。三次往返路程的耗油量按最省要求只能为 500 公升。即 d_{12}=500/3km

dis[2]=dis[1]+d_{12}=dis[1]+500/3

为了在 i=2 处贮存 1000 公升汽油，卡车至少从 i=3 处开三趟满载油的车至 i=2 处。报以 i=3 处至少贮有 3*500 公升汽油，即 oil[3]=500*3=1500。加上 i=2 至 i=3 处的二趟返程空车，合计 5 次。路途耗油量也应为 500 公升，即 d_{23}=500/5,

dis[3]=dis[2]+d_{23}=dis[2]+500/5;

依此类推，为了在 i=k 处贮藏 k*500 公升汽油，卡车至少从 i=k+1 处开 k 趟满载车至 i=k 处，即

oil[k+1]=[k+1]*500=oil[k]+500，加上从 i=k 处返回 i=k+1 的 k-1 趟返程空间，合计 2k-1 次。这 2k-1 次总耗油量按最省要求为 500 公升，即

$d_{k,k+1}$=500/(2k-1)

dis[k+1]=dis[k]+$d_{k,k+1}$

=dis[k]+500/(2k-1);

最后，i=n 至始点的距离为 1000-dis[n],oil[n]=500*n。为了在 i=n 处取得 n*500 公升汽油，卡车至少从始点开 n+1 次满载车至 i=n，加上从 i=n 返回始点的 n 趟返程空车，合计 2n+1 次，2n+1 趟的总耗油量应正好为(1000-dis[n])*(2n+1)，即始点藏油为 oil[n]+(1000-dis[n])*(2n+1)。

下面为程序代码：

```
program oil_lib;
var
k:integer;  {贮油点位置序号}
d,          {累计终点至当前贮油点的距离}
d1:real;    {i=n 至始点的距离}
oil,dis:array[1..10] of real;
i:integer;   {辅助变量}
begin
   writeln('NO.','distance(k.m)':30,'oil(1.)':80);
   k:=1;
```

```
    d:=500;    { 从 i=1 处开始向始点倒推}
    dis[1]:=500;
    oil[1]:=500;
    repeat
       k:=k+1;
       d:=d+500/(2*k-1);
       dis[k]:=d;
       oil[k]:=oil[k-1]+500;
    until d>=1000;

    dis[k]:=1000;       {置始点至终点的距离值}
    d1:=1000-dis[k-1];    {求 i=n 处至始点的距离}
    oil[k]:=d1*(2*k+1)+oil[k-1];    {求始点藏油量}
    for i:=0 to k do       {由始点开始，逐一打印始点至当前贮油点的距离和藏油量}
       writeln(i,1000-dis[k-i]:30,oil[k-i]:80);
end. {main}
```

转换为 C 语言程序如下：

```
#include<stdio.h>
void main()
{
    int k;          /*贮油点位置序号*/
    float d,d1;       /*d:累计终点至当前贮油点的距离,d1:i=n 至始点的距离*/
   float oil[10],dis[10];
   int i;
   printf("NO. distance(k.m.)\toil(l.)\n");
   k=1;
   d=500;        /*从 i=1 处开始向始点倒推*/
   dis[1]=500;
   oil[1]=500;
    do{
      k=k+1;
      d=d+500/(2*k-1);
      dis[k]=d;
      oil[k]=oil[k-1]+500;
   }while(!(d>=1000));
    dis[k]=1000;       /*置始点至终点的距离值*/
    d1=1000-dis[k-1];    /*求 i=n 处至始点的距离*/
    oil[k]=d1*(2*k+1)+oil[k-1];    /*求始点藏油量*/
    for(i=0;i<k;i++)       /*由始点开始逐一打印始点至当前贮油点的距离和藏油量
*/
```

```
    printf("%d\t%f\t%f\t\n",i,1000-dis[k-i],oil[k-i]);
}
```

实用算法(基础算法-递推法-02)

发表日期：2003 年 4 月 10 日 出处：实用算法的分析和程序设计 作者：C 语言之家整理 已经有 1317 位读者读过此文

顺推法

倒推法的逆过程就是顺推法，即由边界条件出发，通过递推关系式推出后项值，再由后项值按递推关系式推出再后项值.....，依次递推，直至从问题初始陈述向前推进到这个问题的解为止。

实数数列：一个实数数列共有 N 项，已知

ai=(a_i-1-a_{i+1})/2+d, (1<i<N)(N<60)

键盘输入 N,d,a_1,a_n,m,输出 a_m

输入数据均不需判错。

算法分析：

分析该题，对公式：

$A_i=(A_{i-1}-A_{i+1})/2+d$ (1<i<N) (n<60)

作一翻推敲，探讨其数字变换规律。不然的话会无从下手。

令 $X=A_2$ $s_2[i]=(p_i,Q_i,R_i)$表示 $A_i=P_iX+Q_iD+R_iA_1$

我们可以根据

$A_i=A_{i-2}-2A_{i-1}+2D$

$=P_iX+Q_iD+R_iA_1$

推出公式

$P_iX+Q_iD+R_iA_1=(P_{i-2}-2P_{i-1})X+(Q_{i-2}-2Q_{i-1}+2)D+(R_{i-2}-2R_{i-1})A_1$

比较等号两端 X,D 和 A1 的系数项，可得

$P_i=P_{i-2}-2P_{i-1}$

$Q_i=Q_{i-2}-2Q_{i-1}+2$

$R_i=R_{i-2}-2R_{i-1}$

加上两个边界条件

$P_1=0$ $Q_1=0$ $R_1=1$ ($A_1=A_1$)

$P_2=1$ $Q_2=0$ $R_2=0$ ($A_2=A_2$)

根据 Pi、Qi、Ri 的递推式，可以计算出

$S_2[1]=(0,0,1)$;

$S_2[3]=(-2,2,1)$;

$S_2[4]=(5,-2,-2)$;

$S_2[5]=(-12,8,5)$;

.....

$S_2[i]=(P_i,Q_i,R_i)$;

.....

$S_2[N]=(P_N,Q_N,R_N)$;

有了上述基础，AM 便不难求得。有两种方法：

1、由于 A_N、A_1 和 P_N、Q_N、R_N 已知，因此可以先根据公式：

$A_2=A_N-Q_ND-R_NA_1/P_N$

求出 A_2。然后将 A_2 代入公式

$A_3=A_1-2A_2+2D$

求出 A_3。然后将 A_3 代入公式

$A_4=A_2-2A_3+2D$

求出 A_4。然后将 A_4 代入公式

.....

求出 A_{i-1}。然后将 A_{i-1} 代入公式

$A_i=A_{i-2}-2A_{i-1}+2D$

求出 A_i。依此类推，直至递推至 A_M 为止。

上述算法的缺陷是由于 A2 是两数相除的结果，而除数 PN 递增，因此精度误差在所难免，以后的递推过程又不断地将误差扩大，以至当 M 超过 40 时，求出的 AM 明显偏离正确值。显然这种方法简单但不可靠。

2、我们令 $A_2=A_2,A_3=X$，由 $S_3[i]=(P_i,Q_i,R_i)$表示 $A_i=P_iX+Q_iD+R_iA_2$ (i>=2) 可计算出：

$S_3[2]=(0,0,1)=S_2[1]$;

$S_3[3]=(1,0,0)=S_2[2]$;

$S_3[4]=(-2,2,1)=S_2[3]$;

$S_3[5]=(5,-2-2)=S_2[4]$;

.....

$S_3[i]=(.....)=S_2[i-1]$;

.....

$S_3[N]=(.....)=S_2[N-1]$;

再令 $A_3=A_3,A_4=X$,由 $S_4[i]=(p_i,Q_i,R_i)$表示 $A_i=P_iX+Q_iD+R_iA_3$ (i>=3) 可计算得出：

$S_4[3]=(0,0,1)=S_3[2]=S_2[1]$;

$S_4[4]=(1,0,0)=S_3[3]=S_2[2]$;

$S_4[5]=(-22,1)=S_3[4]=S_2[3]$;

.....

$S_4[i]=(.....)=S_3[i-1]=S_2[i-2]$;

.....

$S_4[N]=(.....)=S_3[N-1]=S_2[N-2]$;

依此类推，我们可以发现一个有趣的式子：

$A_N=P_{N-i+2}*A_i+Q_{N-i+2}*D+R_{N-i+2}*A_{i-1}$，即

$A_i=(A_N-Q_{N-i+2}*D-R_{N-i+2}*A_{i-1})/P_{N-i+2}$

我们从已知量 A_1 和 A_N 出发，依据上述公式顺序递推 A_2、A_3、...、A_M.由于 P_{N-i+2} 递减，因此最后得出的 A_M 要比第一种算法趋于精确。

程序代码如下：

```
program ND1P4;
const
   maxn   =60;
var
  n,m,i   :integer;
  d       :real;
   list     :array[1..maxn] of real;       {list[i]-------对应 ai}
   s       :array[1..maxn,1..3] of real;  {s[i,1]--------对应 Pi}
                                          {s[i,2]--------对应 Qi}
                                          {s[i,3]--------对应 Ri}
procedure init;
  begin
    write('n m d =');
    readln(n,m,d);          {输入项数，输出项序号和常数}
    write('a1 a',n,'=');
    readln(list[1],list[n]);  {输入 a1 和 an}
  end;   {init}
procedure solve;
  begin
    s[1,1]:=0;s[1,2]:=0;s[1,3]:=1;   {求递推边界(P1,Q1,R1)和(P2,Q2,R2)}
    s[2,1]:=1;s[2,2]:=0;s[2,3]:=0;  {根据公式 Pi<---Pi-2 - 2*Pi-1}
                                    {Qi<---Qi-2 - 2*Qi-1}
                                    {Ri<---Ri-2 - 2*Ri-1}
                                    {递推(P3,Q3,R3).....Pn,Qn,Rn)}
    for i:=3 to n do
      begin
        s[i,1]:=s[i-2,1]-2*s[i-1,1];
        s[i,2]:=s[i-2,2]-2*s[i-1,2]+2;
        s[i,3]:=s[i-2,3]-2*s[i-1,3];
      end; {for}
  end;{solve}
procedure main;
  begin
    solve;       {求(P1,Q1,R1)..(Pn,Qn,Rn)}
                {根据公式 Ai=(An-Qn-i+2 * d-Rn-i+2 * Ai-1)/Pn-i+2}
                 {递推 A2..Am}
    for i:=2 to m do
      list[i]:=(list[n]-s[n-i+2,2]*d-s[n-i+2,3]*list[i-1])/s[n-i+2,1];
```

```
        writeln('a',m,'=',list[m]:20:10);   {输出 Am}
    end;   {main}
begin
    init;      {输入数据}
    main;      {递推和输出 Am}
    readln;
end.   {main}
```

a)求出直线 Ray 与物体的第一个交点（x, y, z）；
b)交点（x, y, z）是可见点，其余交点都应消隐；
c)将点（x, y, z）的属性值（通常是亮度、颜色值或
颜色查找表的索引值）赋给象素（x*, y*）；
}
}
2）循环：对屏幕上每一点（x*, y*）
画出象素（x*, y*）对应的属性值。
}
---- 4 ． 极 值 检 测 法
---- 极 值 检 测 法 需 与 与 其 它 消 隐 算 法 结 合 适 用， 主 要 用 来 提 高 消 隐 速 度。极 值 检 测 法 通 过 计 算 物 体 表 面 的 显 示 坐 标 的 极 大 和 极 小 值 来 判 断 这 两 个 表 面 是 否 存 在 重 叠。 如 果 一 个 表 面 的x 显 示 坐 标 的 极 大 值 小 于 另 一 个 表 面 的x 显 示 坐 标 的 极 小 值， 则 这 两 个 表 面 不 重 叠， 可 以 按 任 意 顺 序 直 接 画 出。 否 则， 这 两 个 表 面 存 在 重 叠， 需 要 用 其 它 消 隐 算 法 进 行 消 隐 处 理。
---- 通 常 先 用 极 值 检 测 法 画 出 不 发 生 重 叠 的 表 面， 然 后 在 用 其 它 算 法 处 理 重 叠 的 表 面。
---- 参 考 文 献
Newmann W M，Sproull R F，Principles of Interactive Computer Graphics，MacGrawHill，1979
Giloi W K，Interactive Computer Graphics -- Data Structure，Algorithms，Languages，Printice-Hall，1978
Hamlin G，Gear G，Raster-scan Hidden Surface Algorithm Techniques，Computer Graphics，Vol. 11，pp206-213，1977
Griffiths J G，Bibliography of Hidden-line and Hidden-surface Algorithms，Comput. Aided Des. ，Vol. 10，No. 3，pp203-206，1978
Atherton P R，A Scan-line Hidden Surface Removal Procedure for Constructive Solid Geometry，Computer Graphics，Vol. 17，No. 3，1983
---- An Analysis of Algorithms for 3-D Graphics Hidden Surface Removal
---- Zhou Zhangfa
---- （Department of Automatic Engineering, Beijing Institute of Light Industry, Beijing

100037）
---- Abstract Modeling is the foundation for 3-D computer graphics, while hidden surface removal is the key of 3-D modeling. This paper discusses 11 algorithms widely used in modern 3-D computer graphics and CAD, 8 of which belong to object space algorithms, the other belong to image space algorithms.
---- Key words Modeling Hidden Surface Removal Object Space Algorithm Image Space Algorithm
---- E-mail：zfzhou@tonghua.com.cn
---- 电 话：68905574
---- 地 址： 阜 成 路 11 号 轻 甲 1 楼 510
---- 邮 编：100037

数据结构：哈夫曼树的应用

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<conio.h>a
#include<graphics.h>
#define MAXVALUE 200          /*权值的最大值*/
#define MAXBIT   30           /*最大的编码位数*/
#define MAXNODE 30            /*初始的最大的结点数*/
 struct haffnode
        {char data;
  int weight;
                int flag;
                int parent;       /*双亲结点的下标*/
                int leftchild;    /*左孩子下标*/
                int rightchild;   /*右孩子下标*/
        };
 struct haffcode
        {int bit[MAXNODE];
                int start;        /*编码的起始下标*/
  char data;
  int weight;       /*字符权值*/
        };


/*函数说明*/
/*************************************************************************/
```

```
void pprintf(struct haffcode haffcode[],int n);
/*输出函数*/
void haffmantree(int weight[],int n,struct haffnode hafftree[],char data[]);
/*建立哈夫曼树*/
void haffmancode(struct haffnode hafftree[],int n,struct haffcode haffcode[]);
/*求哈夫曼编码*/
void test(struct haffcode haffcode[],int n);
/*测试函数*/
void end();
/*结束界面函数*/
/*************************************************************************/


void haffmantree(int weight[],int n,struct haffnode hafftree[],char data[])
   /*建立叶结点个数为 n，权值数组为 weight[]的哈夫曼树*/
   {int i,j,m1,m2,x1,x2;
    /*哈夫曼树 hafftree[]初始化，n 个叶结点共有 2n-1 个结点*/
       for(i=0;i<2*n-1;i++)
      {if(i<n)  {hafftree[i].data=data[i];
   hafftree[i].weight=weight[i];   /*叶结点*/
        }
       else     {hafftree[i].weight=0;          /*非叶结点*/
   hafftree[i].data='\0';
   }
       hafftree[i].parent=0;                  /*初始化没有双亲结点*/
                    hafftree[i].flag=0;
       hafftree[i].leftchild=-1;
                    hafftree[i].rightchild=-1;
                   }
 for(i=0;i<n-1;i++)                        /*构造哈夫曼树 n-1 个非叶结点*/
                   {m1=m2=MAXVALUE;
                    x1=x2=0;
     for(j=0;j<n+i;j++)
    {if(hafftree[j].weight<m1&&hafftree[j].flag==0)
                              {m2=m1;
                               x2=x1;
                               m1=hafftree[j].weight;
                               x1=j;
                              }
     else if(hafftree[j].weight<m2&&hafftree[j].flag==0)
                                {m2=hafftree[j].weight;
                                 x2=j;
```

```
                    }
                }
                hafftree[x1].parent=n+i;
                hafftree[x2].parent=n+i;
                hafftree[x1].flag=1;
                hafftree[x2].flag=1;
                hafftree[n+i].weight=hafftree[x1].weight+hafftree[x2].weight;
                hafftree[n+i].leftchild=x1;
                hafftree[n+i].rightchild=x2;
        }
            }
void haffmancode(struct haffnode hafftree[],int n,struct haffcode haffcode[])
  {/*由 n 个结点的哈夫曼树 hafftree[]构成的哈夫曼编码 haffcode[]*/
   int i,j,child,parent;
   struct haffcode newcode;
   struct haffcode *cd;
   cd=&newcode;
   for(i=0;i<n;i++)                    /*求 n 个结点的哈夫曼编码*/
      {cd->start=MAXBIT-1;                    /*不等长编码的最后一位是 n-1*/
  cd->weight=hafftree[i].weight;
  cd->data=hafftree[i].data;    /*取得编码对应值的字符*/
            child=i;
            parent=hafftree[child].parent;
            while(parent!=0)
      {if(hafftree[parent].leftchild==child)
                cd->bit[cd->start]=0;          /*左孩子编码为 0*/
              else
       cd->bit[cd->start]=1;          /*右孩子编码为 1*/
              cd->start--;
              child=parent;
              parent=hafftree[child].parent;
             }
   for(j=cd->start+1;j<MAXBIT;j++)              /*保存每个叶结点的编码和等长编
码的起始位*/
       haffcode[i].bit[j]=cd->bit[j];
   haffcode[i].data=cd->data;
   haffcode[i].start=cd->start;
   haffcode[i].weight=cd->weight;
      }
  }
void pprintf(struct haffcode myhaffcode[],int n)
     {int i,j,count=0;
```

```
        clrscr();
        for(i=0;i<n;i++)
    {textcolor(YELLOW);
     cprintf("字符=%c",myhaffcode[i].data);
     printf("              ");
     textcolor(YELLOW);
     cprintf("weight=%3d",myhaffcode[i].weight);
     printf("              ");
     textcolor(YELLOW);
     cprintf("haffcode=");
     for(j=myhaffcode[i].start+1;j<MAXBIT;j++)
         cprintf("%d",myhaffcode[i].bit[j]);
     printf("\n");
     count++;
     if(count==21)
     getch();
    }
      }
void test(struct haffcode haffcode[],int n)
     {int i,j,k,s;
      char sstring[MAXNODE];
      struct haffcode newhaffcode[MAXNODE];
      j=0;
      clrscr();
      textcolor(YELLOW);
      cprintf("请输入哈夫曼编码测试数据，在此建议为'this programme is my favorite'");
      printf("\n");
      cprintf("注意小写,空格由大写字母 T 代替，并且字符数小于 27.\n");
      scanf("%s",sstring);
      if(strlen(sstring)>=MAXNODE)
      {printf("you input the data number >=MAXNODE.");
      exit(1);
      }
      for(i=0;i<strlen(sstring);i++)
  {
    for(j=0;j<MAXBIT;j++)
     if(sstring[i]==haffcode[j].data)
     {
      k=j;
      break;
      }
    if(k<0||k>MAXNODE-1)
```

```
    {printf("在系统中找不到与第个%d 字符相匹配的编码\n",i+1);
     continue;
    }
    newhaffcode[i].start=haffcode[k].start;
    newhaffcode[i].weight=haffcode[k].weight;
    newhaffcode[i].data=haffcode[k].data;
    for(s=haffcode[k].start+1;s<MAXBIT;s++)
     newhaffcode[i].bit[s]=haffcode[k].bit[s];
  }
     pprintf(newhaffcode,strlen(sstring));
    }
void end()
   {int driver,mode;
  driver=VGA;
  mode=VGAHI;
  initgraph(&driver,&mode," ");
  setlinestyle(0,0,2);
  setfillstyle(1,9);
  bar(120,60,520,80);
  setfillstyle(1,9);
  bar(90,100,550,350);
  moveto(121,65);
  settextstyle(5,0,6);
  setcolor(7);
  outtext("This programme is designed by Dou Zheren");
  settextstyle(3,0,3);
  setcolor(7);
  moveto(150,200);
  outtext("thank you use this programme.");
  moveto(100,300);
  settextstyle(3,0,2);
  setcolor(7);
  outtext("please press anykey to end this programme.");
   }
void main()
    {
     int i,j,n=27;
     int driver=VGA,mode=VGAHI;
     char ch;
     int weight[27]={186,64,13,22,32,103,21,15,47,
       57,1,5,32,20,57,63,15,1,48,
       51,80,23,8,18,1,16,1};
```

```
    char data[28]={'T','a','b','c','d','e','f','g','h',
     'i','j','k','l','m','n','o','p','q',
     'r','s','t','u','v','w','x','y','z'};
    struct haffnode newhaffnode[2*MAXNODE-1];
    struct haffcode newcode[MAXNODE];
    struct haffnode *myhafftree=newhaffnode;
    struct haffcode *myhaffcode=newcode;
    if(n>MAXNODE)
  {printf("you input the haffnode > MAXNODE,so you input the data is wrong");
   printf("\n");
              exit(1);
  }
    clrscr();
    textcolor(YELLOW);
    cprintf("WELCOME!这是一个求哈夫曼编码的问题");
    printf("\n");
    cprintf("即对所有的字母进行编码后，在根据用户的需要，对用户的要求进行编
码。");
    printf("\n");
    cprintf("注意：本程序只支持小写字母，空格用大写字母 T 代替！  ");
    printf("\n");
    getch();
    textcolor(YELLOW);
    cprintf("Ready?Enter,if you want to begin!\n");
    printf("\n");
    getch();
    cprintf("Now，开始演示哈夫曼编码.");
    getch();
    haffmantree(weight,n,myhafftree,data);
        haffmancode(myhafftree,n,myhaffcode);
    pprintf(myhaffcode,n);
    clrscr();
    printf("若执行自定义编译，请输入 y 继续。否则程序将结束.");
    if((ch=getch())=='y'||ch=='Y')
    test(myhaffcode,n);
    getch();
    clrscr();
    end();
    getch();
    exit(1);
  }
```

递归法和回溯法

有人说，回溯实际上是递归的展开，但实际上。两者的指导思想并不一致。

打个比方吧，递归法好比是一个军队要通过一个迷宫，到了第一个分岔口，有 3 条路，将军命令 3 个小队分别去探哪条路能到出口，3 个小队沿着 3 条路分别前进，各自到达了路上的下一个分岔口，于是小队长再分派人手各自去探路——只要人手足够（对照而言，就是计算机的堆栈足够），最后必将有人找到出口，从这人开始只要层层上报直属领导，最后，将军将得到一条通路。所不同的是，计算机的递归法是把这个并行过程串行化了。

而回溯法则是一个人走迷宫的思维模拟——他只能寄希望于自己的记忆力，如果他没有办法在分岔口留下标记（电视里一演到什么迷宫寻宝，总有恶人去改好人的标记）。

想到这里突然有点明白为什么都喜欢递归了，他能够满足人心最底层的虚荣——难道你不觉得使用递归就象那个分派士兵的将军吗？想想汉诺塔的解法，也有这个倾向，“你们把上面的 N－1 个拿走，我就能把下面的挪过去，然后你们在把那 N－1 个搬过来”。笑谈，切勿当真。

这两种方法的例程，我不给出了，网上很多。我只想对书上的递归解法发表点看法，因为书上的解法有偷梁换柱的嫌疑——迷宫的储存不是用的二维数组，居然直接用岔路口之间的连接表示的——简直是人为的降低了问题的难度。实际上，如果把迷宫抽象成（岔路口）点的连接，迷宫就变成了一个“图”，求解入口到出口的路线，完全可以用图的遍历算法来解决，只要从入口 DFS 到出口就可以了；然而，从二维数组表示的迷宫转化为图是个很复杂的过程。并且这种转化，实际上就是没走迷宫之前就知道了迷宫的结构，显然是不合理的。对此，我只能说这是为了递归而递归，然后还自己给自己开绿灯。

但迷宫并不是只能用上面的方法来走，前提是，迷宫只要走出去就可以了，不需要找出一条可能上的最短路线——确实，迷宫只是前进中的障碍，一旦走通了，没人走第二遍。下面的方法是一位游戏玩家提出来的，既不需要递归，也不

需要栈来回溯——玩游戏还是有收获的。

另一种解法

请注意我在迷宫中用粗线描出的路线，实际上，在迷宫中，只要从入口始终沿着一边的墙走，就一定能走到出口，那位玩家称之为“靠一边走”——如果你不把迷宫的通路看成一条线，而是一个有面积的图形，很快你就知道为什么。编程实现起来也很简单。

下面的程序在 TC2 中编译，不能在 VC6 中编译——为了动态的表现人的移动情况，使用了 gotoxy()，VC6 是没有这个函数的，而且堆砌迷宫的 219 号字符是不能在使用中文页码的操作系统的 32 位的 console 程序显示出来的。如果要在 VC6 中实现 gotoxy()的功能还得用 API，为了一个简单的程序没有必要，所以，就用 TC2 写了，突然换到 C 语言还有点不适应。

```
#include <stdio.h>
typedef struct hero {int x,y,face;} HERO;
void set_hero(HERO* h,int x,int y,int
face){h->x=x;h->y=y;h->face=face;}
void go(HERO* h){if(h->face%2) h->x+=2-h->face;else
h->y+=h->face-1;}
void goleft(HERO* h){if(h->face%2) h->y+=h->face-2;else
h->x+=h->face-1;}
void turnleft(HERO* h){h->face=(h->face+3)%4;}
void turnright(HERO* h){h->face=(h->face+1)%4;}
void print_hero(HERO* h, int b)
{
   gotoxy(h->x + 1, h->y + 1);
   if (b)
   {
       switch (h->face)
       {
```

```
        case 0: printf("%c", 24); break;
        case 1: printf("%c", 16); break;
        case 2: printf("%c", 25); break;
        case 3: printf("%c", 27); break;
        default: break;
        }
    }
    else printf(" ");
}
int maze[10][10] =
{
    0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
    1, 0, 1, 1, 0, 1, 1, 1, 1, 0,
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 0, 1, 1, 0, 1, 1, 1,
    0, 0, 1, 0, 1, 1, 0, 0, 0, 1,
    1, 0, 1, 0, 1, 1, 0, 1, 0, 1,
    0, 0, 1, 0, 1, 1, 0, 1, 0, 1,
    0, 1, 1, 0, 0, 0, 0, 1, 0, 1,
    0, 0, 0, 0, 1, 0, 1, 1, 0, 1,
    0, 1, 1, 1, 1, 0, 0, 0, 0, 0
};
void print_maze()
{
    int i, j;
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 10; j++)
        {
            if (maze[i][j]) printf("%c", 219);
```

```
            else printf(" ");
        }
        printf("\n");
    }
}
int gomaze(HERO* h)
{
    HERO t = *h; int i;
    for (i = 0; i < 2; t = *h)
    {
        print_hero(h, 1); sleep(1); go(&t);
        if (t.x >= 0 && t.x < 10 && t.y >= 0 && t.y < 10
&& !maze[t.y][t.x])
        {
                print_hero(h, 0); go(h);/*前方可走则向前走*/
                if (h->x == 9 && h->y == 9) return 1; goleft(&t);
                if (h->x == 0 && h->y == 0) i++;
                if (t.x >= 0 && t.x < 10 && t.y >= 0 && t.y < 10
&& !maze[t.y][t.x]) turnleft(h);/*左方无墙向左转*/
        }
        else turnright(h);/*前方不可走向右转*/
    }
    return 0;
}

main()
{
    HERO Tom;/*有个英雄叫Tom*/
    set_hero(&Tom, 0, 0, 0);/*放在(0,0)面朝北*/
    clrscr();
```

```
    print_maze();
    gomaze(&Tom);/*Tom 走迷宫*/
}
```

总结

书上讲的基本上就这些了，要是细说起来，几天几夜也说不完。前面我并没有讲如何写递归算法，实际上给出的都是非递归的方法，我也觉得有点文不对题。我的目的是使大家明白，能写出什么算法，主要看你解决问题的指导思想，换而言之，就是对问题的认识程度。所以初学者现在就去追求“漂亮”的递归算法，是不现实的，结果往往就是削足适履，搞的一团糟——有位仁兄写了个骑马游世界的“递归”程序，在我机器上 10 分钟没反映。其实优秀的递归算法是在对问题有了清楚的认识后才会得出的。

最后说说用汇编语言写递归函数。我的汇编水平并不高，不过我想说的是用汇编写递归函数，绝对不像《汇编与 c 解决递归问题之比较》http://www.csdn.net/develop/article/17/17597.shtm 那篇文章说的，实际上比高级语言并不复杂，甚至在 masm32v7 中，和高级语言一样，因为那里面有一句很象代参函数调用的 INVOKE expression [, arguments]。那位作者显然连教科书都没看全，因为在我们的讲 8086 汇编语言的书上就有一个阶乘的递归函数例程，如果他看过，就不会有那个结论了。

迷宫

关于迷宫，有一个引人入胜的希腊神话，这也是为什么现今每当人们提到这个问题，总是兴致勃勃（对于年青人，估计是 RPG 玩多了），正如虽然九宫图连小学生都能做出来，我们总是自豪的说那叫“洛书”。这个神话我不复述了，有兴趣的可以在搜索引擎上输入“希腊神话 迷宫”，就能找到很多的介绍。

迷宫的神话讲述了一位英雄如何靠着“线团”杀死了牛头怪（玩过《英雄无敌》的朋友一定知道要想造牛头怪，就必须建迷宫，也是从这里来的），我看到的一本编程书上援引这段神话讲述迷宫算法的时候，不知是有意杜撰，还是考证

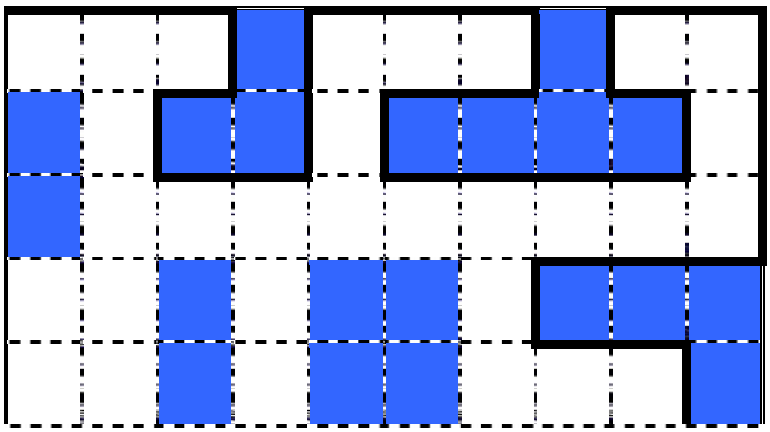
不严，把这个过程叙述成：英雄靠着线团的帮助——在走过的路上铺线，每到分岔口向没铺线的方向前进，如果遇到死胡同，沿铺的线返回，并铺第二条线——走进了迷宫深处，杀死了牛头怪。然而，神话传说讲的是，英雄被当成贡品和其他的孩子送到了迷宫的深处，英雄杀死了牛头怪，靠着线团标识的路线退出了迷宫。实际上，这个线团只是个“栈”，远没有现代人赋予给它的“神奇作用”。我想作者也是 RPG 玩多了，总想着怎样“勇者斗恶龙”，然而，实际上却是“胜利大逃亡”。

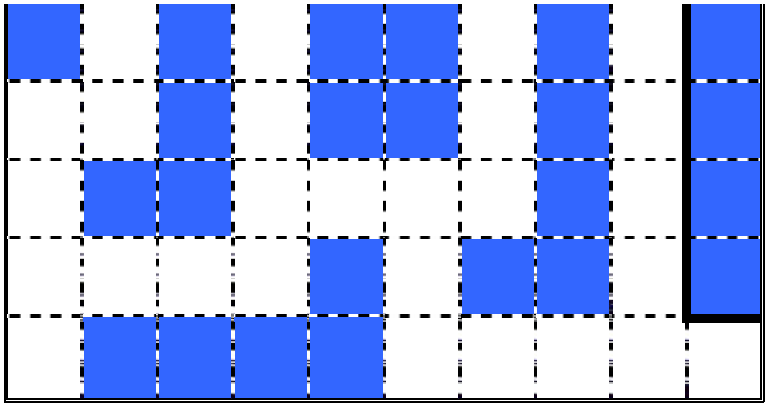
迷宫问题实际上是一个心理测试，它反映了测试者控制心理稳定的能力——在一次次失败后，是否失去冷静最终陷在迷宫之中，也正体现了一句诗，“不识庐山真面目，只缘身在此山中”。换而言之，我们研究迷宫的计算机解法，并没有什么意义，迷宫就是为人设计的，而不是为机器设计的，它之所以称为“迷”宫，前提是人的记忆准确性不够高；假设人有机器那样的准确的记忆，只要他不傻，都能走出迷宫。现在可能有人用智能机器人的研究来反驳我，实际上，智能机器人是在更高的层面上模拟人的思考过程，只要它完全再现了人的寻路过程，它就能走出迷宫。但是，研究迷宫生成的计算机方法，却是有意义的，因为人们总是有虐待自己的倾向（不少人在 RPG 里的迷宫转了三天三夜也不知道疲倦），呵呵，笑谈。

不管怎么说，还是亲自研究一下计算机怎么走迷宫吧。

迷宫的存储

按照惯例，用一个二维数组来表示迷宫，0 表示墙，1 表示通路，以后我们的程序都走下面这个迷宫。





```
#include <iostream>
#include <vector>
using namespace std;
class Needle
{
public:
    Needle() { a.push_back(100); }//每一个柱子都有一个底座
    void push(int n) { a.push_back(n); }
    int top() { return a.back(); }
    int pop() { int n = a.back(); a.pop_back(); return n; }
    int movenum(int n) { int i = 1;while (a[i] > n) i++; return a.size()
- i; }
    int size() { return a.size(); }
    int operator [] (int n) { return a[n]; }
private:
    vector<int> a;
};
void  Hanoi(int n)
{
    Needle needle[3], ns;//3 个柱子，ns 是转换柱子时的保存栈，借用了
Needle 的栈结构
    int source = 0, target, target_m = 2, disk, m = n;
    for (int i = n; i > 0; i--) needle[0].push(i);//在 A 柱上放 n 个盘子
```

```
while (n)//问题规模为 n，开始搬动
{
    if (!m) { source = ns.pop(); target_m = ns.pop();
        m = needle[source].movenum(ns.pop()); }//障碍盘子搬走后，回到原来的当前柱
    if (m % 2) target = target_m; else target = 3 - source - target_m;//规律 1 的实现
    if (needle[source].top() < needle[target].top())//当前柱顶端盘子可以搬动时，移动盘子
    {
        disk = needle[source].top();m--;
        cout << disk << " move " << (char)(source + 0x41) << " to "<< (char)(target + 0x41) << endl;//显示搬动过程
        needle[target].push(needle[source].pop());//在目标柱上面放盘子
        if (disk == n) { source = 1 - source; target_m = 2; m = --n; }规律 3 的实现
    }
    else//规律 2 的实现
    {
        ns.push(needle[source][needle[source].size() - m]);
        ns.push(target_m); ns.push(source);
        m = needle[target].movenum(needle[source].top());
        target_m = 3 - source - target; source = target;
    }
}
```

这个算法实现比递归算法复杂了很多（递归算法在网上、书上随便都可以找到），而且还慢很多，似乎是多余的，然而，这是有现实意义的。我不知道现在

还在搬 64 个盘子的僧人是怎么搬的，不过我猜想一定不是先递归到 1 个盘子，然后再搬——等递归出来，估计胡子一打把了（能不能在人世还两说）。我们一定是马上决定下一步怎么搬，就如我上面写的那样，这才是人的正常思维，而用递归来思考，想出来怎么搬的时候，黄瓜菜都凉了。正像我们做事的方法，虽然我今生今世完不成这项事业，但我一定要为后人完成我能完成的，而不是在那空想后人应该怎么完成——如果达不到最终的结果，那也一定保证向正确的方向前进，而不是呆在原地空想。

由此看出，计算机编程实际上和正常的做事步骤的差距还是很大的——我们的做事步骤如果直接用计算机来实现的话，其实并不能最优，原因就是，实际中的相关性在计算机中可能并不存在——比如人脑的逆推深度是有限的，而计算机要比人脑深很多，论记忆的准确性，计算机要比人脑强很多。这也导致了一个普通的程序员和一个资深的程序员写的算法的速度常常有天壤之别。因为，后者知道计算机喜欢怎么思考。

据结构学习（C++）——双向链表

原书这部分内容很多，至少相对于循环链表是很多。相信当你把单链表的指针清楚后，这部分应该难不倒你。现在我的问题是，能不能从单链表派生出双向链表

你可以有几种做法：

一种就是先定义一个双链节点——但是，它的名字必须叫 Node，这是没办法

不然你就只好拷贝一份单链表的实现文件，把其中的 Node 全都替换成你的双点名字，但是这就不叫继承了。

另一种做法就是先定义一种结构例如这样的：

```
mplate <class Type> class newtype

blic:
  Type data;
  Node<newtype> *link;
```

当你派生双向链表时，这样写 template <calss Type> class DblList : public List<newtype<Type> >，注意连续的两个“>”之间要有空格。或者根本不定义这样的结构，直接拿 Node 类型来做，例如我下面给出的。但是，请注意要完成“==”的重载，否则，你又要重写 Find 函数，并且其他的某些操作也不方便。

在开始完成你的从单链表派生出来的双向链表之前，要在单链表这个基类中添加修改当前指针和当前前驱指针的接口，如下所示：

```
protected:
    void Put(Node<Type> *p)//尽量不用，双向链表将使用这个完成向前移动
    {
        current = p;
    }

    void PutPrior(Node<Type> *p)//尽量不用，原因同上
    {
        prior = p;
    }
```

因为这个接口很危险，而且几乎用不到，所以我在前面并没有给出，但要完成双向链表最“杰出”的优点——向前移动当前指针，必须要使用。另外说的是，我从前也从来没计划从单链表派生双链表，下面你将看到，这个过程很让人烦人，甚至不如重写一个来的省事，执行效率也不是很好，这种费力不讨好的事做它有什么意思呢？的确，我也觉得我在钻牛角尖。（别拿鸡蛋丢我）

定义和实现

```
#ifndef DblList_H
#define DblList_H

#include "List.h"

template <class Type> class DblList : public List< Node<Type> >
```

```
{
public:
	Type *Get()
	{
		if (pGet() != NULL) return &pGet()->data.data;
		else return NULL;
	}

	Type *Next()
	{
		pNext();
		return Get();
	}

	Type *Prior()
	{
		if (pGetPrior != NULL)
		{
			Put(pGetPrior());
			PutPrior( (Node< Node<Type> >*)pGet()->data.link);
			return Get();
		}
		return NULL;
	}

	void Insert(const Type &value)
	{
		Node<Type> newdata(value, (Node<Type>*)pGet());
		List< Node<Type> >::Insert(newdata);
```

```
        if (pGetNext()->link != NULL)
            pGetNext()->link->data.link = (Node<Type>*)pGetNext();
    }

    BOOL Remove()
    {
        if (List< Node<Type> >::Remove())
        {
            pGet()->data.link = (Node<Type>*)pGetPrior();
            return TURE;
        }
        return FALSE;
    }

};

#endif
```

【说明】只完成了最重要的 Insert 和 Remove 函数和最具特点的 Prior()函数，其他的没有重新实现。所以，你在这里使用单链表的其他方法，我不保证一定正确。并且，这里的指针类型转换依赖于编译器实现，我也不能肯定其他的编译器编译出来也能正确。对于让不让 Prior 返回头节点的 data，我考虑再三，反正用 First();Get();这样的组合也能返回，所以就不在乎他了，所以要是用 Prior 遍历直到返回 NULL，就会将头节点的 data 输出来了。

【补充】至于双向循环链表，也可以从这个双向链表派生（仿照派生循环链表的方法）；或者从循环链表派生（仿照派生双向链表的方法），就不一一举例了（再这样下去，我就真闹心的要吐血了）。至此，可以得出一个结论，链表的各种结构都是能从单链表派生出来的。换句话说，单链表是根本所在，如果研究透了单链表，各种链式结构都不难。

一小段测试程序

```
void DblListTest_int()
{
	DblList<int> a;
	for (int i = 10; i > 1; i--) a.Insert(i);
	for (i = 10; i > 1; i--) cout << *a.Next() << " ";
	a.First();
	cout << endl;
	cout << *a.Next() << endl;
	cout << *a.Next() << endl;
	cout << *a.Next() << endl;
	cout << *a.Next() << endl;
	a.Remove();
	cout << *a.Get() << endl;
	cout << *a.Prior() << endl;
	cout << *a.Prior() << endl;
	cout << *a.Prior() << endl;
}
```

【后记】从我对双向链表不负责任的实现来看，我并不想这么来实现双向链表，我只是尝试怎样最大限度的利用已有的类来实现这种类型。实践证明，不如重写一个。别人看起来也好看一些，自己写起来也不用这样闹心。不过，这个过程让我对函数的调用和返回的理解又更深了一步。如果你能第一次就写对这里的 Insert 函数，相信你一定对 C++有一定的感触了。我也觉得，只有做一些创新，才能最已经很成熟的东西更深入的了解。比如，这些数据结构，在 C++的标准库（STL）中都可以直接拿来用，我们为什么还辛辛苦苦的写，结果还不如人家原来的好。为了学习，这就是理由，这也是一切看起来很笨的事发生的理由。

水波算法实例

```
//*********************************************************
```

```
//根据波能数据缓冲区对离屏页面进行渲染
//******************************************************
void RenderRipple()
{
//锁定两个离屏页面
DDSURFACEDESC ddsd1, ddsd2;
ddsd1.dwSize = sizeof (DDSURFACEDESC);
ddsd2.dwSize = sizeof(DDSURFACEDESC);
lpDDSPic1->Lock(NULL, &ddsd1, DDLOCK_WAIT, NULL);
lpDDSPic2->Lock(NULL, &ddsd2, DDLOCK_WAIT, NULL);
//取得页面象素位深度，和页面内存指针
int depth=ddsd1.ddpfPixelFormat.dwRGBBitCount/8;
BYTE *Bitmap1 = (BYTE*)ddsd1.lpSurface;
BYTE *Bitmap2 = (BYTE*)ddsd2.lpSurface;

//下面进行页面渲染
int xoff, yoff;
int k = BACKWIDTH;
for (int i=1; i<BACKHEIGHT-1; i++)
{
for (int j=0; j<BACKWIDTH; j++)
{
//计算偏移量
xoff = buf1[k-1]-buf1[k+1];
yoff = buf1[k-BACKWIDTH]-buf1[k+BACKWIDTH];
//判断坐标是否在窗口范围内
if ((i+yoff )< 0 ) {k++; continue;}
if ((i+yoff )> BACKHEIGHT) {k++; continue;}
if ((j+xoff )< 0 ) {k++; continue;}
if ((j+xoff )> BACKWIDTH ) {k++; continue;}
//计算出偏移象素和原始象素的内存地址偏移量
int pos1, pos2;
pos1=ddsd1.lPitch*(i+yoff)+ depth*(j+xoff);
pos2=ddsd2.lPitch*i+ depth*j;
//复制象素
for (int d=0; d<depth; d++)
Bitmap2[pos2++]=Bitmap1[pos1++];
k++;
}
}
//解锁页面
lpDDSPic1->Unlock(&ddsd1);
lpDDSPic2->Unlock(&ddsd2);
}
```

增加波源

俗话说：无风不起浪，为了形成水波，我们必须在水池中加入波源，你可以想象成向水中投入石头，形成的波源的大小和能量与石头的半径和你扔石头的力量都有关系。知道了这些，那么好，我们只要修改波能数据缓冲区 buf，让它在石头入水的地点来一个负的“尖脉冲”，即让 buf[x,y]=-n。经过实验，n 的范围在（32~128）之间比较合适。

控制波源半径也好办，你只要以石头入水中心点为圆心，画一个以石头半径为半径的圆，让这个圆中所有的点都来这么一个负的“尖脉冲”就可以了（这里也做了近似处理）。

增加波源的代码如下：

```
//******************************************************
//增加波源
//******************************************************
void DropStone(int x,//x 坐标
int y,//y 坐标
int stonesize,//波源半径
int stoneweight)//波源能量
{
//判断坐标是否在屏幕范围内
if ((x+stonesize)>BACKWIDTH ||
(y+stonesize)>BACKHEIGHT||
(x-stonesize)<0||
(y-stonesize)<0)
return;
for (int posx=x-stonesize; posx<x+stonesize; posx++)
for (int posy=y-stonesize; posy<y+stonesize; posy++)
if ((posx-x)*(posx-x) + (posy-y)*(posy-y) < stonesize*stonesize)
buf1[BACKWIDTH*posy+posx] = -stoneweight;
}
```

好了，至此，水波特效的制作原理就此就全部揭示了。在上面的推导中，每一步都进行了很多看似非常过分的近似处理，但是，你完全不必担心，事实证明，用这种方法，在速度和图象上都可以获得非常好的效果。源程序中有非常详尽的注释，仔细推敲一下，看懂它们应该不成问题。

这个程序是 Win32 下的 DirectX 编程，没有使用任何包装库。在我的电脑上（AMDK6-200、2MVRam、64MSRam），320x240 的画面大小，每秒可以达到 25 帧。与前几个程序不一样，这个程序使用了窗口模式，所以调试起来很方便。如果你对窗口模式编程不熟悉，这个程序也是一个很好的例子。

这种用数据缓冲区对图象进行水波处理的方法，有个最大的好处就是，程序运算和其示的速度与水波的复杂程度是没有关系的，无论水面是风平浪静还是波涛汹涌，程序的 fps 始终保持不变，这一点你研究一下程序就应该可以看出来。实际上，如果你掌握了这种方法，将这种方法推广一下，完全可以做出另外一些特殊的效果，如烟雾、大气、阳光等，我现在也正在研究这些特效的制作，相信不久以后就会有新的收获

平摊分析

在平摊分析中，执行一系列数据结构操作所需要的时间是通过对执行的所有操作求平均而得出的。平摊分析可用来证明在一系列操作中，即使单一的操作具有较大的代价，通过

对所有操作求平均后，平均代价还是很小的。平摊分析与平均情况分析的不同之处在于它不牵涉到概率。这种分析保证了在最坏情况下每个操作具有平均性能。

本文将讨论平摊分析技术中最常用的三种技术：

- 聚集方法 —— 可以用这种方法确定一个 n 个操作的序列的总代价的上界 T(n)。每个操作的平摊代价可表示为 T(n)/n；
- 会计方法 —— 用它可确定每个操作的平摊代价。当有一种以上的操作时，每种操作都可有一个不同的平摊代价。这种方法对操作序列中的某些操作先“多记帐”，将多记的部分作为对数据结构中的特定对象上预付的存款存起来。在该序列中稍后要用到这些存款以补偿那些对它们记的“帐”少于其实际代价的操作。
- 势能方法 —— 它与会计方法的相似之处在于要确定每个操作的代价，且先对某些操作多记帐以补偿以后的不足记帐。这种方法将存数作为数据结构的“势能”来维护，而不是将存款与数据结构中的单个对象联系起来。

我们将用两个例子来说明这三个模型。第一个例子是个栈，它有一个新的操作 MULTIPOP，它可一次弹出几个对象。另一个例子是个二进计数器，它利用操作 INCREMENT 从 0 开始计数。

在阅读本文时，读者应记住在平摊分析中所记的“帐”只是为了分析之用，它们不应出现在代码中。例如，当在采用会计方法时，如果将一存款赋予一个对象，在代码中就没有必要对其属性 credit[x]赋一个相应的值了。

通过做平摊分析而获得的对某种数据结构特性的认识有助于优化设计。例如，后文中我们将用势能方法来分析一个动态扩充和收缩的表。

聚集方法

在平摊分析的聚集方法中，我们要证明对所有的 n，n 个操作构成的序列在最坏情况下总的时间为 T(n)。在最坏情况下，每个操作的平摊代价就为 T(n)/n。请注意这个平摊代价对每个操作都是成立的，即使当序列中存在几种类型的操作时也是一样的。后文中要研究的另两种方法—— 会计方法和势能方法——对不同类型的操作则可能赋予不同的平摊代价。

栈操作

在关于聚集方法的第一个例子中，我们要来分析增加了一个新操作的栈。我们知道，栈的两种基本操作 ——PUSH 和 POP，每个的时间代价都是 O(1)：

- PUSH(S,x) —— 将对象 x 压入栈 S；
- POP(S) —— 弹出并返回 S 的顶端元素；

因为这两个操作的运行时间都为 O(1)，故我们可把每个操作的代价视为 1。这样，一个 n 个 PUSH 和 POP 操作的序列的总代价就为 n，而这 n 个操作的实际运行时间就为 O(n)。

如果再增加一个栈操作 MULTIPOP(S,k)，则情况会变得更有趣。该操作去掉 S 的 k 个顶端对象，或当它包含少于 k 个对象时弹出整个栈。在下面的伪代码中，如果当前栈中没有对象则操作 STACK-EMPTY 返回 TRUE，否则它返回 FALSE。

```
MULTIPOP(S,k)
1 while not STACK-EMPTY(S) and k≠0
2    do POP(S)
```

```
3        k ← k-1
```

图 1 演示了 MULTIPOP 的一个例子，初始情况见图 1(a)。最上面的四个对象由 MULTIPOP(S,4)弹出，其结果如(b)中所示。下一个操作是 MULTIPOP(S,7)，它将栈清空——如图 1(c)中所示——因为余下的对象已经不足七个了。

图 1 MULTIPOP 作用于栈 S 上的动作

MULTIPOP(S，k)作用于一个包含 s 个对象的栈上的运行时间怎样呢？实际的运行时间与实际执行的 POP 操作数成线性关系，因而只要按 PUSH 和 POP 具有抽象代价 1 来分析 MULTIPOP 就是够了。代码中 while 循环执行的次数即从栈中弹比的对象数 min(s,k)。对该循环的每一次执行，在第 2 行中都要调用一次 POP。这样，MULTIPOP 的总代价即址为 min(s,k)，而实际运行时间则为这个代价的一个线性函数。

现在来对作用于一个初始为空的栈上的 n 个 PUSH，POP 和 MULTIPOP 操作构成的序列作个分析。序列中一次 MULTIPOP 操作的最坏情况代价为 O(n)，因为栈的大小至多为 n。这样，任意栈操作的最坏情况时间就是 O(n)，而 n 个操作的总代价就是 $O(n^2)$，因为 MULTIPOP 操作可能会有 O(n)个，每个的代价为 O(n)。虽然这个分析是正确的，但通过分析每个操作的最坏情况代价而得的 $O(n^2)$的结论却是不够准确的。

利用平摊分析中的聚集方法，我们可以或的一个考虑到了整个操作序列的更好的上界。事实上，虽然某一次 MULTIPOP 操作的代价可能较高，但作用于初始为空的栈上的任意一个包含 n 个 PUSH，POP 和 MULTIPOP 操作的序列的代价至多为 O(n)。为什么会是这样呢？一个对象在每次被压入栈后至多被弹出一次。所以，在一个非空栈上调用 POP 的次数(包括在 MULTIPOP 内的调用)至多等于 PUSH 操作的次数，即至多为 n。对任意的 n 值，包含 n 个 PUSH，POP 和 MULTIPOP 操作的序列的总时间为 O(n)。据此，每个操作的平摊代价为：O(n)/n=O(1)。

我们想再一次强调一下，虽然我们已说明了每个栈操作的平均代价(或平均运行时间)为 O(1)，**但没有用到任何概率推理**。实际上是给出了一列 n 个操作的最坏情况界 O(n)。用 n 来除这个总代价即可得每个操作的平均代价(或说平摊代价)。

二进制计数器

作为聚集方法的另一个例子，考虑实现一个由 0 开始向上计数的 k 位二进制计数器的问题。我们用一个数组 A[0..k-1](此处 length[A]=k)作为计数器。存储在计数器中的一个二进数 x 的最低位在 A[0]中，最高位在 A[k-1]中，故

开始时，x=0，故 A[i]=0，i=0,1,...,k-1。为将计数器中的值加 1(模 2^k)，我们可用下面的过程：

```
INCREMENT(A)
1  i←0
2  while i<length[A] and A[i]=1
3    do A[i]←0
4       i←i+1
5  if i<length[A]
6    then A[i]←1
```

这个算法与硬件实现的行波进位计数器基本上是一样的。图 2 演示了一个二进制计数器从 0 至 16 的 16 次增值的过程。发生翻转而取得下一个值的位都加了阴影。右边示出了位翻转所需的代价。注意总代价始终不超过 INCREMENT 操作总次数的两倍。在第 2-4 行中每次 while 循环的开始，我们希望在位置 i 处加 1。如果 A[i]=1，则加 1 后就将位置 i 处的数位置为 0，并产生一个进位 1，它在循环的下一次执行中加到位置 i+1 上；否则，循环结束；然后，如果 i<k，我们知道 A[i]=0，故将 1 加到位置 i 后，使 0 变为 1，这在第 6 行中完成。每次 INCREMENT 操作的代价与被改变值的位数成线性关系。

像在栈的例子中一样，大致分析一下只能得到一个正确但不紧确的界。在最坏情况下，INCREMENT 的每次执行要花 O(k)时间，此时数组 A 中包含全 1。这样，在最坏情况下，作用于一个初始为零的计数器上的 n 个 INCREMENT 操作的时间就为 O(nk)。

如果我们分析得更精确一些的话，则可得到 n 次 INCREMENT 操作的序列的最坏情况代价为 O(n)。关键是要注意到在每次调用 INCREMENT 中，并不是所有的位都发生变化：作用于初始为零的计数器上的 n 次 INCREMENT 操作导致 A[1]变化了$\lfloor n/2 \rfloor$次。类似地，位 A[2]在 n 次 INCREMENT 操作中共变化$\lfloor n/4 \rfloor$次。一般地，对 i=0,1,..,$\lfloor \log n \rfloor$，位 A[i]在一个作用于初始为零的计数器上的 n 次 INCREMENT 操作的序列中共要翻转$\lfloor n/2^i \rfloor$次。对 i >$\lfloor \log n \rfloor$，位 A[i]始终不发生变化。这样，在序列中发生的位翻转的总次数为

由此可知，作用于一个初始为零的计数器上的 n 次 INCREMENT 操作的最坏情况时间为 O(n)，因而每次操作的平摊代价为 O(n)/n=O(1)。

Counter Value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3

3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

图 2 在 16 次 INCREMENT 操作作用下，一个八位二进制计数器的值从 0 变到 16

会计方法

在平摊分析的会计方法中，我们对不同的操作赋予不同的费值，某些操作的费值比它们的实际代价或多或少。对每一操作所记的费值即其平摊代价。当一个操作的平摊代价超过了它的实际代价时，两者的差值就被作为存款赋给数据结构中一些特定的对象。存款可在以后用于补偿那些其平摊代价低于其实际代价的操作。这样，我们就可将一操作的平摊代价看作为两部分——实际代价与存款(或被储蓄或被使用)。这与聚集方法有很大不同，后者所有操作都具有相同的平摊代价。

在选择操作的平摊代价时是要非常小心的。如果我们希望通过对平摊代价的分析说明每次操作具有较小的最坏情况平均代价，则操作序列的总的平摊代价就必须是该序列的总的实际代价的一个上界。而且，像在聚集方法中一样，这种关系必须对所有的操作序列都成立。这样，与该数据结构相联系的存款始终应该是非负的，因为它表示了总的平摊代价超过总的实际代价的部分。如果允许总的存款为负的话(开始时对某些操作的费值记得过低)，则在某一时刻总的平摊代价就会低于总的实际代价。对到该时刻为止的操作序列来说，总的平摊代价就不会是总的实际代价的一个上界。所以，我们必须始终注意数据结构中的总存款不能是负的。

栈操作

为了说明平摊分析中的会计方法，我们再回过头看看栈的例子。各栈操作的实际代价为：

- PUSH 1
- POP 1
- MULTIPOP min(k,s)

其中 k 为 MULTIPOP 的一个参数，s 为调用该操作时栈的大小。现对它们赋予以下的平摊代价：

- PUSH 2
- POP O
- MULTIPOP O

请注意 MULTIPOP 的平摊代价是个常数 0，而它的实际代价却是个变量。此处所有的三个平摊代价都是 O(1)，但一般来说所考虑的各种操作的平摊代价会渐近地变化。

现在我们来说明只需要用平摊代价就支付任何的栈操作序列。假设我们用 1 元钱来表示代价的单位。开始时栈是空的。栈数据结构与在餐馆中一堆迭放的盘子类似。当将一个盘子压入堆上时，我们用 1 元来支付该压入动作的实际代价，并有 1 元的存款(记的是 2 元的帐)，将该 1 元钱放在刚压入的盘子的上面。在任何一个时间点上，堆中每个盘子的上而都有 l 元钱的余款。

盘中所存的钱是用来预付将盘从栈中弹出所需代价的。当我们在执行了一个 POP 操作时，对该操作不用收任何费，只要用盘中所存放的余款来支付其实际代价即可。为弹出一个盘子，我们拿掉该盘子上的 1 元余款，并用它来支付弹出操作的实际代价。这样，在对 PUSH 操作多收了一点费后，就无需对 POP 操作收取任何费用。

更进一步，我们对 MULTIPOP 操作也无需收费。为弹出第一个盘子，我们取出其中的 1 元余款并用它支付一次 POP 操作的实际代价。为弹出第二个盘子，再取出该盘子上的 1 元余款来支付第二次 POP 操作，等等。这样，对任意的包含 n 次 PUSH，POP 和 MULTIPOP 操作的序列，总的平摊代价就是其总的实际代价的一个上界。又因为总的平摊代价为 O(n)，故总的实际代价也为 O(n)。

二进制计数器的增值

为进一步说明会计方法，我们再来分析一下作用于一个初始为 0 的二进制计数器上的 INCREMENT 操作。我们前面已经说过，这个操作的运行时间与发生翻转的位数是成正比的，而位数在本例中即为代价。我们还是用 1 元钱来表示位数代价(此例中即为某一位的翻转)。

为进行平摊分析，我们规定对将某一位置为 1 的操作收取 2 元的平摊费用。当某数位被设定后，我们用 2 元中的 1 元来支付置位操作的实际代价，而将另 1 元存在该位上作为余款。在任何时间点上，计数器中每个 1 上都有 1 元余款。这样在将某位复位成 0 时不用支付任何费用，只要取出该位上的 1 元余款即可。

现在就可以来确定 INCREMENT 的平摊代价了。在 while 循环中复位操作的代价是由有关位上的余款来支付的。在 INCREMENT 的第 6 行中至多有一位被复位，所以一次 INCREMENT 操作的代价至多为 2 元。又因为计数器中为 1 的位数始终是非负的，故其中总的余款额也是非负的。对 n 次 INCREMENT 操作，总的平摊代价为 2n 元，即 O(n)，这就给出了总的实际代价的一个界。

势能方法

平摊分析中的势能方法不是将已预付的工作作为存储在数据结构特定对象中的存款来

表示，而是表示成--种“势能”，或“势”，它在需要时可释放出来以支付后面操作的代价。势是与整个数据结构而不是其中的个别对象发生联系的。

势能方法的工作过程是这样：开始时先对一个初始数据结构 D_0 执行 n 个操作，对每个 i=1,2,..n，设 c_i 为第 i 个操作的实际代价，D_i 为对数据结构 D_{i-1} 作用第 i 个操作的结果。势函数Φ将每个数据结构 D_i 映射为一个实数$\Phi(D_i)$，即与数据结构 D_i 相联系的势。第 i 个操作的平摊代价定义为：

(1)

从这个式子可以看出，每个操作的平摊代价为其实际代价加上由于该操作所增加的势。根据等式(1)，n 个操作的总的平摊代价为：

（2）

如果我们能定义--个势函数Φ使得$\Phi(D_n)\geqq\Phi(D_0)$，则总的平摊代价就是总的实际代价的一个上界。在实践中，我们并不总是
知道要执行多少个操作，所以，如果要求对所有的 i 有$\Phi(D_i)\geqq\Phi(D_0)$，则就应像在会计方法中一样，保证预先支付。通常为了方便起见，我们定义$\Phi(D_0)$为 0，然后再证明对所有 i 有$\Phi(D_i)\geqq O$；倘若$\Phi(D_0)\neq0$，只要构造势函数Φ′，使得$\Phi'(i)=\Phi(D_i)-\Phi(D_0)$即可。

从直觉上看，如果第 i 个操作的势差$\Phi(D_i)-\Phi(D_{i-1})$是正的，则平摊代价　就表示对第 i 个操作多收了费，同时数据结构的势也随之增加了。如果势差是负的，则平摊代价就表示对第 i 个操作的不足收费，这时就可通过减少势来支付该操作的实际代价。

由等式(1)和(2)所定义的平摊代价依赖于所选择的势函数Φ，不同的势函数可能会产生不同的平摊代价，但它们都是实际代价的上界。在选择一个势函数时常要作一些权衡，可选用的最佳势函数的选择要取决于所需的时间界。

栈操作

为了说明势能方法，我们再一次来研究栈操作 PUSH，POP 和 MULTIPOP。定义栈上的势函数Φ为栈中对象的个数。开始时我们要处理的是空栈 D_0，$\Phi(D_0)=0$。因为栈中的对象数始终是非负的，故在第 i 个操作之后的栈 D_i 就具有非负的势，且有

以Φ表示的 n 个操作的平摊代价的总和就表示了实际代价的一个上界。

现在我们来计算各栈操作的平摊代价。如果作用于一个包含 s 个对象的栈上的第 i 个操作是个 PUSH 操作，则势差为

根据等式(1)，该 PUSH 操作的代价为

假设第 i 个操作是 MULTIPOP(S,k)，且弹出了 k'=min(k,s)个对象。该操作的实际代价为 k'，势差为

这样，MULTIPOP 操作的平摊代价为

类似地，POP 操作的平摊代价也是 0。

三种栈操作中每一种的平摊代价都是 O(1)，这样包含 n 个操作的序列的总平摊代价就是 O(n)。因为我们已经证明了$\Phi(D_i) \geqq \Phi(D_0)$，故 n 个操作的总平摊代价即为总的实际代价的一个上界。这样 n 个操作的最坏情况代价为 O(n)。

二进制计数器的增值

作为说明势能方法的另一个例子，我们再来看看二进计数器的增值问题。这---次，我们定义在第 i 次 INCREMENT 操作后计数器的势为 b_i，即第 i 次操作后计数器中 1 的个数。

我们来计算一下一次 INCREMENT 操作的平摊代价。假设第 i 次 INCREMENT 操作对 t_i 个位进行了复位。该操作的实际代价至多为 t_i+1，因为除了将 t_i 个位复位外，它至多将一位置成 1。所以，在第 i 次操作后计数器中 1 的个数为 $b_i \leqq b_{i-1}-t_i+1$，势差为

平摊代价为

如果计数器开始时为 0，则$\Phi(D_0)=0$。因为对所有 i 有$\Phi(D_i) \geqq 0$，故 n 次 INCREMENT 操作的序列的总平摊代价就为总的实际代价的一个上界，且 n 次 INCREMENT 操作的最坏情况代价为 O(n)。

势能方法给我们提供了一个简易方法来分析开始时不为零的计数器。开始时有 b_0 个 1，在 n 次 INCREMENT 操作之后有 b_n 个 1，此处 $0 \leqq b_0, b_n \leqq k$。我们可以将等式（2）重写为：

对所有 $1 \leqq i \leqq n$ 有 $c_i \leqq 2$。因为$\Phi(D_0)=b_0$，$\Phi(D_n)=b_n$，n 次 INCREMENT 操作的总的实际代价为：

请注意因为 $b_0 \leqq k$，如果我们执行了至少 $n=\Omega(k)$次 INCREMENT 操作，则无论计数器中包含什么样的初始值，总的实际代价都是 O(n)。

动态表

在有些应用中，在开始的时候无法预知在表中要存储多少个对象。可以先为该表分配一定的空间，但后来可能会觉得不够，这样就要为该表分配一个更大的空间，而表中的对象就复制到新表中。类似地，如果有许多对象被从表中删去了，就应该给原表分配一个更小的空间。在后文中，我们要研究表的动态扩张和收缩的问题。利用平摊分析，我们要证明插入和删除操作的平摊代价为 O(1)，即使当它们引起了表的扩张和收缩时具有较大的实际代价也时一样的。此外，我们将看到如何来保证某一动态表中未用的空间始终不超过整个空间的一部分。

假设动态表支持 Insert 和 Delete 操作。Insert 将某一元素插入表中，该元素占据一个槽（即一个元素占据的空间）。同样地，Delete 将一个元素从表中去掉，从而释放了一个槽。用来构造这种表的数据结构方法的细节不重要，可以选用的有栈，堆或杂凑表结构。我们将用一个数组或一组数组来实现对象存储。

大家将会发现在采用了杂凑表中分析杂凑技术时引入的装载因子概念后会很方便。定义一个非空表 T 的装载因子$\alpha(T)$为表中存储的对象项数和表的大小(槽的个数)的比值。对一个空表（其中没有元素）定义其大小为 0，其装载因子为 1。如果某一动态表的装载因子以一个常数为上界，则表中未使用的空间就始终不会超过整个空间的一常数部分。

我们先开始分析只对之做插入的动态表，然后再考虑既允许插入又允许删除的更一般的情况。

表的扩张

假设一个表的存储空间分配为一个槽的数组，当所有的槽都被占用的时候这个表就被填满了，这时候其装载因子为 1。在某些软件环境中，如果试图向一个满的表中插入一个项，就只会导致错误。此处假设我们的软件环境提供了存储管理系统，它能够根据请求来分配或释放存储块。这样，当向一满的表中插入一个项时，我们就能对原表进行扩张，即分配一个包含比原表更多槽的新表，再将原表中各项数据复制到新表中去。

一种常用的启发技术是分配一个比原表大一倍的新表。如果只对表进行插入操作，则表的装载因子总是至少为 1/2，这样浪费掉的空间就始终不会超过表的总空间的一半。

在下面的伪代码中，我们假设对象 T 表示一个表，域 table[T]包含了一个指向表的存储

块的指针，域 num[T]包含了表中的项数，域 size[T]为表中总的槽数。开始时，表是空的：num[T]=size[T]=0

```
Insert(T, x)
1  if size[T]=0
2    then 给 table[T]分配一个槽的空间
3         size[T]←1
4  if num[T]=size[T]
5    then 分配一个有 2*size[T]个槽的空间的新表
6         将 table[T]中所有的项插入到新表中
7         释放 Table[T]
8         table[T]指向新表的存储块地址
9         size[T]←2*size[T]
10 将 x 插入 table[T]
11 num[T]←num[T]+1
```

请注意，这里有两种“插入”过程：Insert 过程本身与第 6 行和第 l0 行中的基本插入。可以根据基本插入的操作数来分析 Insert 的运行时间，每个基本插入操作的代价为 1。假淀 Insert 的实际运行时间与插入项的时间成线性关系，使得在第 2 行中分配初始表的开销为常数，而第 5 行与第 7 行中分配和释放存储的开销由第 6 行中转移表中所有项的开销决定。我们称第 5-9 行中执行 then 语句的事件为一次扩张。

现在我们来分析一下作用于一个初始为空的表上的 n 次 Insert 操作的序列。设第 i 次操作的代价 c_i，如果在当前的表中还有空间（或该操作是第一个操作），则 $c_i=1$，因为这时我们只需在第 10 行中执行一次基本插入操作即可。如果当前的表是满的，则发生一次扩张，这时 $c_i=i$；第 10 行中基本插入操作的代价 1 再加上第 6 行中将原表中的项夏制到新表中的代价 i-1。如果执行了 n 次操作，则一次操作的最坏情况代价为 O(n)，由此可得 n 次操作的总的运行时间的上界 O(n)。

但这个界不很紧确，因为在执行 n 次 Insert 操作的过程中并不常常包括扩张表的代价。特别地，仅当 i-1 为 2 的整数幕时第 i 次操作才会引起一次表的扩张。实际上，每一次操作的平摊代价为 O(1)，这一点我们可以用聚集方法加以证明。第 i 次操作的代价为

由此，n 次 Insert 操作的总代价为

因为至多有 次操作的代价为 1，而余下的操作的代价就构成了一个几何级数。因为 n 次 Insert 操作的总代价为 3n，故每次操作的平摊代价为 3。

通过采用会计方法，我们可以对为什么一次 Insert 操作的平摊代价会是 3 有一些认识。从直觉上看，每一项要支付三次基本插入操作：将其自身插入现行表中，当表扩张时对其自身的移动，以及对另一个在扩张表时已经移动过的另一项的移动。例如，假设刚刚完成扩张后某一表的大小为 m，那么表中共有 m/2 项，且没有“存款”。对每一次插入操作要收费 3 元。立即发生的基本插入的代价为 1 元，另有 1 元放在刚插入的元素上作为存款，余下的 1 元放在已在表中的 m/2 个项上的某一项上作为存款。填满该表另需要 m/2 次插入，这样，到该表包含了 m 个项时，该表已满，每一项上都有 1 元钱以支付在表扩张期间的插入。

也可以用势能方法来分析一系列 n 个 Insert 操作，我们还将在后文中用此方法来设计一个平摊代价为 O(1)的 Delete 的操作。开始时我们先定义一个势函数Φ，在完成扩张时它为 0，当表满时它也达到表的大小，这样下一次扩张的代价就可由存储的势来支付了。函数

(4)

是一种可能的选择。在刚刚完成一次扩张后，我们有 num[T]=size[T]/2，于是有$\Phi(T)=0$，这正是所希望的。在就要做一次扩张前，有 num[T]=size[T]，于是$\Phi(T)=num[T]$，这也正是我们希望的。势的初值为 0，又因为表总是至少为半满，$num[T]\geqq size[T]/2$，这就意味着$\Phi(T)$总是非负的。所以，n 次 Insert 操作的总的平摊代价就是总的实际代价的一个上界。

为了分析第 i 次 Insert 操作的平摊代价，我们用 num_i 来表示在第 i 次操作后表中所存放的项数，用 $size_i$ 表示在第 i 次操作之后表的大小，Φ_i 表示第 i 次操作之后的势。开始时，$num_0=O$，$size_0=0$ 和$\Phi_0=0$。

如果第 i 次 Insert 操作没有能触发一次表的扩张，则 $num_i=num_{i-1}+1$，$size_i=size_{i-1}$，且该操作的平摊代价为

如果第 i 次操作确实触发了一次扩张，则 $num_i=num_{i-1}+1$，$size_i = 2size_{i-1} = 2num_{i-1}=2(num_i -1)$，且该操作的平摊代价为

图 3 画出了 num_i，$size_i$ 和Φ_i 的各个值。在第 i 次操作后对这些量中的每一个都要加以计算。图中红线表示 num_i，紫线表示 $size_i$，蓝线表示Φ_i。注意在每一次扩张前，势已增长到等于表中的项目数，因而可以支付将所有元素移到新表中去的代价。此后，势降为 0，但一但引起扩张的项目被插入时其值就立即增加 2。

图 3 对表中项目数 num_i，表中的空位数 $size_i$，以及势Φ_i作用 n 次 Insert 操作的效果

表扩张和收缩

为了实现 Delete 操作，只要将指定的项从表中去掉即可。但是，当某一表的装载因子过小时，我们就希望对表进行收缩，使得浪费的空间不致太大。表收缩与表扩张是类似的：当表中的项数降得过低时，我们就要分配一个新的、更小的表，而后将旧表的各项复制到新表中。旧表所占用的存储空间则可被释放，归还到存储管理系统中去。在理想情况下，我们希望下面两个性质成立：

- 动态表的装载因子由一常数作为下界；
- 各表操作的平摊代价由一常数作为下界。

另外，我们假设用基本插入和删除操作来测度代价。

关于表收缩和扩张的一个自然的策略是当向表中插入一个项时将表的规模扩大一倍，而当从表中删除一项就导致表的状态小于半满时，则将表缩小一半。这个策略保证了表的装载因子始终不会低于 1/2，但不幸的是，这样又会导致各表操作具有较大的平摊代价。请考虑一下下面这种情况：我们对某一表 T 执行 n 次操作，此处 n 为 2 的整数幂。前 n/2 个操作是插入，由前面的分析可知其总代价为 O(n)。在这一系列插入操作的结束处，num[T]=size[T]=n/2。对后面的 n/2 个操作，我们执行下面这样一个序列：I,D,D,I,I,D,D,I,I，……，其中 I 表示插入，D 表示删除。第一次插入导致表扩张至规模 n。紧接的两次删除又将表的大小收缩至 n/2；紧接的两次插入又导致表的另一次扩张，等等。每次扩张和收缩的代价为 Θ(n)，共有 Θ(n)次扩张或收缩。这样，n 次操作的总代价为 $\Theta(n^2)$，而每一次操作的平摊代价为 Θ(n)。

这种策略的困难性是显而易见的：在一次扩张之后，我们没有做足够的删除来支付一次收缩的代价。类似地，在一次收缩后，我们也没有做足够的插入以支付一次扩张的代价。

我们可以对这个策略加以改进，即允许装载因子低于 1/2。具体来说，当向满的表中插

入一项时，还是将表扩大一倍，但当删除一项而引起表不足 1/4 满时，我们就将表缩小为原来的一半。这样，表的装载因子就以常数 1/4 为下限界。这种做法的基本思想是使扩张以后表的装载因子为 1/2。因而，在发生一次收缩前要删除表中一半的项，因为只有当装载因子低于 1/4 时方会发生收缩。同理，在收缩之后，表的装载因子也是 1/2。这样，在发生扩张前要通过扩张将表中的项数增加一倍，因为只有当表的装载因子超过 1 时方能发生扩张。

我们略去了 Delete 的代码，因为它与 Insert 的代码是类似的。为了方便分析，我们假定如果表中的项数降至 0，就释放该表所占存储空间。亦即，如果 num[T]=0，则 size[T]=0。

现在我们用势能方法来分析由 n 个 Insert 和 Delete 操作构成的序列的代价。先完义一个势函数Φ，它在刚完成一次扩张或收缩时值为 0，并随着装载因子增至 1 或降至 1/4 而变化。我们用α(T)= num[T]/size[T]来表示一个非空表 T 的装载因子。对一个空表，因为有 num[T]=size[T]=0，且α(T)=1，故总有 num[T]=α(T)*size[T]，无论该表是否为空。我们采用的势函数为

(5)

请注意，空表的势为 0；势总是非负的。这样，以Φ表示的一列操作的总平摊代价即为其实际代价的--个上界。

在进行详细分析之前，我们先来看看势函数的某些性质。当装载因子为 1/2 时，势为 0。当它为 1 时，有 size[T]=num[T]，这就意味着Φ(T)=num[T]，这样当因插入一项而引起一次扩张时，就可用势来支付其代价。当装载因子为 1/4 时，我们有 size[T]=4*num[T]，它意味着Φ(T)=num[T]，因而当删除某个项引起一次收缩时就可用势来支付其代价。图 4 说明了对一系列操作势是如何变化的。

图 4 对表中的项目数 num_i、表中的空位数 $size_i$ 及势Φ_i
作用由 n 个 Insert 和 Delete 操作构成的操作序列的效果

图 4 中红线表示 num_i，紫线表示 $size_i$，蓝线表示Φ_i。注意在每一次扩张前，势已增长到等于表中的项目数，因而可以支付将所有元素移到新表中去的代价。类似地，在一次收缩之前，势也增加到等于表中的项目数。

为分析 n 个 Insert 和 Delete 的操作序列，我们用　　来表示第 i 次操作的实际代价，c_i 表示其参照Φ的平摊代价，num_i 表示在第 i 次操作之后表中存储的项数，$size_i$ 表示第 i 次操作后表的大小，α_i 表示第 i 次操作后表的装载因子，Φ_i 表示第 i 次操作后的势。开始时，num_0=O，$size_0$=0，α_0=1，Φ_0=0。

我们从第 i 次操作是 Insert 的情况开始分析。如果$\alpha_{i-1}\geq 1/2$，则所要做的分析就与对表扩张的分析完全一样。无论表是否进行了扩张，该操作的平摊代价 c_i 都至多是 3。如果$\alpha_{i-1}<1/2$，则表不会因该操作而扩张，因为仅当$\alpha_{i-1}=1$ 时才发生扩张。如果还有$\alpha_i<1/2$，则第 i 个操作的平摊代价为

如果$\alpha_{i-1}< 1/2$，但$\alpha_i\geq 1/2$，那么

因此，一次 Insert 操作的平摊代价至多为 3。

现在我们再来分析一下第 i 个操作是 Delete 的情形。这时，$num_i=num_{i-1}-1$。如果$\alpha_{i-1}< 1/2$，我们就要考虑该操作是否会引起一次收缩。如果没有，则 $size_i=size_{i-1}$，而该操作的平摊代价则为

如果$\alpha_{i-1}< 1/2$ 且第 i 个操作触发一次收缩，则该操作的实际代价为 $c_i=num_i+1$，因为我们删除了一项，移动了 num_i 项。这时，$size_i/2 = size_{i-1}/4 = num_i+1$，而该操作的平摊代价

为

当第 i 次操作为 Delete 且$\alpha_{i-1}\geq 1/2$ 时，其平摊代价仍有一常数上界。具体的分析从略。

总之，因为每个操作的平摊代价都有一常数上界，所以作用于一动态表上的 n 个操作的实际时间为 O(n)。

算法表达中的抽象机制

傅清祥 王晓东

算法与数据结构，电子工业出版社,1998

摘要

本文介绍了算法表达中的抽象机制，引入了抽象数据类型 ADT 的概念，提供一种相应的自顶向下逐步求精、模块化的程序设计方法，即运用抽象数据类型来描述程序的方法。

目录

简介

要用计算机解决一个稍为复杂的实际问题，大体都要经历如下的步骤。

1. 将实际问题数学化，即把实际问题抽象为一个带有一般性的数学问题。这一步要引入一些数学概念，精确地阐述数学问题，弄清问题的已知条件、所要求的结果、以及在已知条件和所要求的结果之间存在着的隐式或显式的联系。
2. 对于确定的数学问题，设计其求解的方法，即所谓的算法设计。这一步要建立问题的求解模型，即确定问题的数据模型并在此模型上定义一组运算，然后借助于对这组运算的调用和控制，从已知数据出发导向所要求的结果，形成算法并用自然语言来表述。这种语言还不是程序设计语言，不能被计算机所接受。
3. 用计算机上的一种程序设计语言来表达已设计好的算法。换句话说，将非形式自然语言表达的算法转变为一种程序设计语言表达的算法。这一步叫程序设计或程序编制。
4. 在计算机上编辑、调试和测试编制好的程序，直到输出所要求的结果。

在这里，我们只关心第 3 步，而且把注意力集中在算法程序表达的抽象机制上，目的是引人一个重要的概念--抽象数据类型，同时为大型程序设计提供一种相应的自顶向下逐步求精、模块化的具体方法，即运用抽象数据类型来描述程序的方法。

从机器语言到高级语言的抽象

我们知道，算法被定义为一个运算序列。这个运算序列中的所有运算定义在一类特定的数据模型上，并以解决一类特定问题为目标。这个运算序列应该具备下列四个特征。

1. 有限性，即序列的项数有限，且每一运算项都可在有限的时间内完成;
2. 确定性，即序列的每一项运算都有明确的定义，无二义性;
3. 可以没有输入运算项，但一定要有输出运算项;
4. 可行性，即对于任意给定的合法的输入都能得到相应的正确的输出。

这些特征可以用来判别一个确定的运算序列是否称得上是一个算法。

但是，我们现在的问题不是要判别一个确定的运算序列是否称得上是一个算法，而是要对一个已经称得上是算法的运算序列，回顾我们曾经如何用程序设计语言去表达它。

算法的程序表达，归根到底是算法要素的程序表达，因为一旦算法的每一项要素都用程序清楚地表达，整个算法的程序表达也就不成问题。

作为运算序列的算法，有三个要素。

1. 作为运算序列中各种运算的运算对象和运算结果的数据;
2. 运算序列中的各种运算;
3. 运算序列中的控制转移。

这三种要素依序分别简称为**数据**、**运算**和**控制**。

由于算法层出不穷，变化万千，其中的运算所作用的对象数据和所得到的结果数据名目繁多，不胜枚举。最简单最基本的有布尔值数据、字符数据、整数和实数数据等;稍复杂的有向量、矩阵、记录等数据;更复杂的有集合、树和图，还有声音、图形、图像等数据。

同样由于算法层出不穷，变化万千，其中运算的种类五花八门、多姿多彩。最基本最初等的有赋值运算、算术运算、逻辑运算和关系运算等;稍复杂的有算术表达式和逻辑表达式等;更复杂的有函数值计算、向量运算、矩阵运算、集合运算，以及表、栈、队列、树和图上的运算等:此外，还可能有以上列举的运算的复合和嵌套。

关于控制转移，相对单纯。在串行计算中，它只有顺序、分支、循环、递归和无条件转移等几种。

我们来回顾一下，自从计算机问世以来，算法的上述三要素的程序表达，经历过一个怎样的过程。

最早的程序设计语言是机器语言，即具体的计算机上的一个指令集。当时，要在计算机上运行的所有算法都必须直接用机器语言来表达，计算机才能接受。算法的运算序列包括运算对象和运算结果都必须转换为指令序列。其中的每一条指令都以编码(指令码和地址码)的形式出现。与算法语言表达的算法，相差十万八千里。对于没受过程序设计专门训练的人来说，一份程序恰似一份"天书"，让人看了不知所云，可读性极差。

用机器语言表达算法的运算、数据和控制十分繁杂琐碎，因为机器语言所提供的指令太初等、原始。机器语言只接受算术运算、按位逻辑运算和数的大小比较运算等。对于稍复杂的运算，都必须一一分解，直到到达最初等的运算才能用相应的指令替代之。机器语言能直接表达的数据只有最原始的位、字节、和字三种。算法中即使是最简单的数据如布尔值、字符、整数、和实数，也必须一一地映射到位、字节和字中，还得一一分配它们的存储单元。

对于算法中有结构的数据的表达则要麻烦得多。机器语言所提供的控制转移指令也只有无条件转移、条件转移、进入子程序和从子程序返回等最基本的几种。用它们来构造循环、形成分支、调用函数和过程得事先做许多的准备，还得靠许多的技巧。

直接用机器语言表达算法有许多缺点。

1. 大量繁杂琐碎的细节牵制着程序员，使他们不可能有更多的时间和精力去从事创造性的劳动，执行对他们来说更为重要的任务。如确保程序的正确性、高效性。
2. 程序员既要驾驭程序设计的全局又要深入每一个局部直到实现的细节，即使智力超群的程序员也常常会顾此失彼，屡出差错，因而所编出的程序可靠性差，且开发周期长。
3. 由于用机器语言进行程序设计的思维和表达方式与人们的习惯大相径庭，只有经过较长时间职业训练的程序员才能胜任，使得程序设计曲高和寡。
4. 因为它的书面形式全是"密"码，所以可读性差，不便于交流与合作。
5. 因为它严重地依赖于具体的计算机，所以可移植性差，重用性差。

这些弊端造成当时的计算机应用未能迅速得到推广。

克服上述缺点的出路在于程序设计语言的抽象，让它尽可能地接近于算法语言。

为此，人们首先注意到的是可读性和可移植性，因为它们相对地容易通过抽象而得到改善。于是，很快就出现汇编语言。这种语言对机器语言的抽象，首先表现在将机器语言的每一条指令符号化:指令码代之以记忆符号，地址码代之以符号地址，使得其含义显现在符号上而不再隐藏在编码中，可让人望"文"生义。其次表现在这种语言摆脱了具体计算机的限制，可在不同指令集的计算机上运行，只要该计算机配上汇编语言的一个汇编程序。这无疑是机器语言朝算法语言靠拢迈出的一步。但是，它离算法语言还太远，以致程序员还不能从分解算法的数据、运算和控制到汇编才能直接表达的指令等繁杂琐碎的事务中解脱出来。

到了 50 年代中期，出现程序设计的高级语言如 Fortran，Algol60，以及后来的 PL/I，Pascal 等，算法的程序表达才产生一次大的飞跃。

诚然，算法最终要表达为具体计算机上的机器语言才能在该计算机上运行，得到所需要的结果。但汇编语言的实践启发人们，表达成机器语言不必一步到位，可以分两步走或者可以筑桥过河。即先表达成一种中介语言，然后转成机器语言。汇编语言作为一种中介语言，并没有获得很大成功，原因是它离算法语言还太远。这便指引人们去设计一种尽量接近算法语言的规范语言，即所谓的高级语言，让程序员可以用它方便地表达算法，然后借助于规范的高级语言到规范的机器语言的"翻译"，最终将算法表达为机器语言。而且，由于高级语言和机器语言都具有规范性，这里的"翻译"完全可以机械化地由计算机来完成，就像汇编语言被翻译成机器语言一样，只要计算机配上一个编译程序。

上述两步，前一步由程序员去完成，后一步可以由编译程序去完成。在规定清楚它们各自该做什么之后，这两步是完全独立的。它们各自该如何做互不相干。前一步要做的只是用高级语言正确地表达给定的算法，产生一个高级语言程序;后一步要做的只是将第一步得到的高级语言程序翻译成机器语言程序。至于程序员如何用高级语言表达算法和编译程序如何将高级语言表达的算法翻译成机器语言表达的算法，显然毫不相干。

处理从算法语言最终表达成机器语言这一复杂过程的上述思想方法就是一种抽象。汇编语言和高级语言的出现都是这种抽象的范例。

与汇编语言相比，高级语言的巨大成功在于它在数据、运算和控制三方面的表达中引

入许多接近算法语言的概念和工具，大大地提高抽象地表达算法的能力。

在运算方面，高级语言如 Pascal，除允许原封不动地运用算法语言的四则运算、逻辑运算、关系运算、算术表达式、逻辑表达式外，还引入强有力的函数与过程的工具，并让用户自定义。这一工具的重要性不仅在于它精简了重复的程序文本段，而且在于它反映出程序的两级抽象。在函数与过程调用级，人们只关心它能做什么，不必关心它如何做。只是到函数与过程的定义时，人们才给出如何做的细节。用过高级语言的读者都知道，一旦函数与过程的名称、参数和功能被规定清楚，那么，在程序中调用它们便与在程序的头部说明它们完全分开。你可以修改甚至更换函数体与过程体，而不影响它们的被调用。如果把函数与过程名看成是运算名，把参数看成是运算的对象或运算的结果，那么，函数与过程的调用和初等运算的引用没有两样。利用函数和过程以及它们的复合或嵌套可以很自然地表达算法语言中任何复杂的运算。

在数据方面，高级语言如 Pascal 引人了数据类型的概念，即把所有的数据加以分类。每一个数据(包括表达式)或每一个数据变量都属于其中确定的一类。称这一类数据为一个数据类型。 因此，数据类型是数据或数据变量类属的说明，它指示该数据或数据变量可能取的值的全体。对于无结构的数据，高级语言如 Pascal，除提供标准的基本数据类型--布尔型、字符型、整型和实型外，还提供用户可自定义的枚举类型、子界类型和指针类型。这些类型(除指针外)，其使用方式都顺应人们在算法语言中使用的习惯。对于有结构的数据，高级语言如 Pascal，提供了数组、记录、有限制的集合和文件等四种标准的结构数据类型。其中，数组是科学计算中的向量、矩阵的抽象;记录是商业和管理中的记录的抽象;有限制的集合是数学中足够小的集合的势集的抽象;文件是诸如磁盘等外存储数据的抽象。人们可以利用所提供的基本数据类型(包括标准的和自定义的)，按数组、记录、有限制的集合和文件的构造规则构造有结构的数据。 此外，还允许用户利用标准的结构数据类型，通过复合或嵌套构造更复杂更高层的结构数据。这使得高级语言中的数据类型呈明显的分层，如图 1-6 所示。

高级语言中数据类型的分层是没有穷尽的，因而用它们可以表达算法语言中任何复杂层次的数据。

在控制方面，高级语言如 Pascal，提供了表达算法控制转移的六种方式。

(1)缺省的顺序控制";"。

(2)条件(分支)控制:"if 表达式(为真)then S1 else S2;" 。
(3)选择(情况)控制:
"Case 表达式 of
值 1: S1
值 2: S2
...
值 n: Sn
end"
(4)循环控制:
"while 表达式(为真) do S;" 或
"repeat S until 表达式(为真);" 或
"for 变量名:=初值 to/downto 终值 do S;"
(5)函数和过程的调用，包括递归函数和递归过程的调用。
(6)无条件转移 goto。

这六种表达方式不仅覆盖了算法语言中所有控制表达的要求，而且不再像机器语言或汇编语言那样原始、那样繁琐、那样隐晦，而是如上面所看到的，与自然语言的表达相差无几。

程序设计语言从机器语言到高级语言的抽象，带来的主要好处是：

1. 高级语言接近算法语言，易学、易掌握，一般工程技术人员只要几周时间的培训就可以胜任程序员的工作；
2. 高级语言为程序员提供了结构化程序设计的环境和工具，使得设计出来的程序可读性好，可维护性强，可靠性高；
3. 高级语言远离机器语言，与具体的计算机硬件关系不大，因而所写出来的程序可移植性好，重用率高；
4. 由于把繁杂琐碎的事务交给了编译程序去做，所以自动化程度高，开发周期短，且程序员得到解脱，可以集中时间和精力去从事对于他们来说更为重要的创造性劳动，以提高程序的质量。

抽象数据类型

与机器语言、汇编语言相比，高级语言的出现大大地简便了程序设计。但算法从非形式的自然语言表达到形式化的高级语言表达，仍然是一个复杂的过程，仍然要做很多繁杂琐碎的事情，因而仍然需要抽象。

对于一个明确的数学问题，设计它的算法，总是先选用该问题的一个数据模型。接着，弄清该问题所选用的数据模型在已知条件下的初始状态和要求的结果状态，以及隐含着的两个状态之间的关系。然后探索从数据模型的已知初始状态出发到达要求的结果状态所必需的运算步骤。把这些运算步骤记录下来，就是该问题的求解算法。

按照自顶向下逐步求精的原则，我们在探索运算步骤时，首先应该考虑算法顶层的运算步骤，然后再考虑底层的运算步骤。所谓顶层的运算步骤是指定义在数据模型级上的运算步骤，或叫宏观运算。它们组成算法的主干部分。表达这部分算法的程序就是主程序。其中涉及的数据是数据模型中的一个变量，暂时不关

心它的数据结构;涉及的运算以数据模型中的数据变量作为运算对象，或作为运算结果，或二者兼而为之，简称为定义在数据模型上的运算。由于暂时不关心变量的数据结构，这些运算都带有抽象性质，不含运算的细节。所谓底层的运算步骤是指顶层抽象的运算的具体实现。它们依赖于数据模型的结构，依赖于数据模型结构的具体表示。因此，底层的运算步骤包括两部分:一是数据模型的具体表示;二是定义在该数据模型上的运算的具体实现。我们可以把它们理解为微观运算。于是，底层运算是顶层运算的细化;底层运算为顶层运算服务。为了将顶层算法与底层算法隔开，使二者在设计时不会互相牵制、互相影响，必须对二者的接口进行一次抽象。让底层只通过这个接口为顶层服务，顶层也只通过这个接口调用底层的运算。这个接口就是**抽象数据类型**。其英文术语是 **Abstract Data Types**，简记 ADT。

抽象数据类型是算法设计和程序设计中的重要概念。严格地说，它是算法的一个数据模型连同定义在该模型上、作为该算法构件的一组运算。这个概念明确地把数据模型与作用在该模型上的运算紧密地联系起来。事实正是如此。一方面，如前面指出过的，数据模型上的运算依赖于数据模型的具体表示，因为数据模型上的运算以数据模型中的数据变量作为运算对象，或作为运算结果，或二者兼而为之;另方面，有了数据模型的具体表示，有了数据模型上运算的具体实现，运算的效率随之确定。于是，就有这样的一个问题:如何选择数据模型的具体表示使该模型上的各种运算的效率都尽可能地高?很明显，对于不同的运算组，为使组中所有运算的效率都尽可能地高，其相应的数据模型具体表示的选择将是不同的。在这个意义下，数据模型的具体表示又反过来依赖于数据模型上定义的那些运算。特别是，当不同运算的效率互相制约时，还必须事先将所有的运算的相应使用频度排序，让所选择的数据模型的具体表示优先保证使用频度较高的运算有较高的效率。数据模型与定义在该模型上的运算之间存在着的这种密不可分的联系，是抽象数据类型的概念产生的背景和依据。

应该指出，抽象数据类型的概念并不是全新的概念。它实际上是我们熟悉的基本数据类型概念的引伸和发展。用过高级语言进行算法设计和程序设计的人都知道，基本数据类型已隐含着数据模型和定义在该模型上的运算的统一，只是当时还没有形成抽象数据类型的概念罢了。事实上，大家都清楚，基本数据类型中的逻辑类型就是逻辑值数据模型和或(∨)、与(∧)、非(┐)三种逻辑运算的统一体;整数类型就是整数值数据模型和加(+)、减(-)、乘(*)、除(div)四种运算的统一体;实型和字符型等也类同。每一种基本类型都连带着一组基本运算。只是由于这些基本数据类型中的数据模型的具体表示和基本运算的具体实现都很规范，都可以通过内置(built-in)而隐蔽起来，使人们看不到它们的封装。许多人已习惯于在算法与程序设计中用基本数据类型名和相关的运算名，而不问其究竟。所以没有意识到抽象数据类型的概念已经孕育在基本数据类型的概念之中。

回到定义算法的顶层和底层的接口，即定义抽象数据类型。根据抽象数据类型的概念，对抽象数据类型进行定义就是约定抽象数据类型的名字，同时，约定在该类型上定义的一组运算的各个运算的名字，明确各个运算分别要有多少个参数，这些参数的含义和顺序，以及运算的功能。一旦定义清楚，算法的顶层就可以像引用基本数据类型那样，十分简便地引用抽象数据类型;同时，算法的底层就有了设计的依据和目标。顶层和底层都与抽象数据类型的定义打交道。顶层运

算和底层运算没有直接的联系。因此，只要严格按照定义办，顶层算法的设计和底层算法的设计就可以互相独立，互不影响，实现对它们的隔离，达到抽象的目的。

在定义了抽象数据类型之后，算法底层的设计任务就可以明确为:

1. 赋每一个抽象数据类型名予具体的构造数据类型，或者说，赋每一个抽象数据类型名予具体的数据结构;
2. 赋每一个抽象数据类型上的每个运算名予具体的运算内容，或者说，赋予具体的过程或函数。

因此，落实下来，算法底层的设计就是数据结构的设计和过程与函数的设计。用高级语言表达，就是构造数据类型的定义和过程与函数的说明。

不言而喻，由于实际问题千奇百怪，数据模型千姿百态，问题求解的算法千变万化，抽象数据类型的设计和实现不可能像基本数据类型那样可以规范、内置、一劳永逸。它要求算法设计和程序设计人员因时因地制宜，自行筹划，目标是使抽象数据类型对外的整体效率尽可能地高。

下面用一个例子来说明，对于一个具体的问题，抽象数据类型是如何定义的。

考虑拓扑排序问题:已知一个集合 S={a_1,a_2, ... ,a_m}，S 上已规定了一个部分序<。要求给出 S 的一个线性序{a_1',a_2', ... ,a_m'}，即 S 的一个重排，使得对于任意的 1<=j<k<=m，不得有 a_k'<a_j'。这里所谓 S 上的部分序<，是指 S 上的一种序关系，它对于 S 中的任意元素 x,y 和 z，具有如下三个性质:

1. 不得有 x<x；(反自反性）
2. 若 x<y，则不得有 y<x；（反对称性）
3. 若 x<y，,且 y<z，则 x<z；(传递性)。

其中 x<y 读作 x 先于 y,或等价地读作 x 是 y 的前驱，或 y 是 x 是后继。

由于已知的 S 上的部分序<可以用一个有向图 G 来表示，而要求的 S 的线性序可以用一个队列 Q 来表示，所以问题的数据模型包括一类有向图和一类队列。我们将其分别取名为 Digraph 和 Queue。其中 G=G(V，E)是 Digraph 中的一个有向图，结点集 V=S，有向边集 E 是由<决定的 S 的元素间的有向连线的全体;Q=S={a_1,a_2, ... ,a_m}是 Queue 中的一个队列。在 G 中，a_i 和 a_j 之间有一条起于 a_i 止于 a_j 的有向连线的充分必要条件是 a_i<a_j。具体地说，比如 S={a_1，a_2, ... ,a_{10}}，而<如表 1-3 所示，则 G(V,E)如图 1-7，而 Q={a_7，a_9，a_1，a_2，a_4，a_6，a_3，a_5，a_8，a_{10}}。这个 Q 只是问题的一个解。显然问题的解不唯一，容易举出 Q'={a_1，a_2，a_7,a_9，a_{10}，a_4，a_6，a_3，a_5，a_8}是另一个解。

$a_1<a_2$
$a_2<a_4$
$a_4<a_6$
$a_2<a_{10}$
$a_4<a_8$
$a_6<a_3$
$a_1<a_3$
$a_3<a_5$
$a_5<a_8$
$a_7<a_5$
$a_7<a_9$
$a_9<a_4$
$a_9<a_{10}$

表 1-3 $S=\{a_1,a_{2,}...,a_{10}\}$中的部分序

在数据模型 Digraph 和 Queue 的基础上，容易拟定出算法高层的宏观运算步骤，我们称之为算法的主干部分，并用非形式的自然语言表述如下：

1.φ->Q;
2.检测 G。
(1)当 G≠φ 时;
①在 G 中出任意一个无前驱的结点，记为 a；
②将 a 加到 Q 的末尾;
③在 G 中删去结点 a 以及以 a 为起点的所有有向边;
④转向 2。
(2)当 C=φ 时，算法结束，问题的解在 Q 中。

用高级语言中的控制结构语句成分，替换上述主干算法中自然语言的控制转移术语，则主干算法可用自然语言和高级语言的混合语言改述如下:

```
φ->Q;
while G≠φ do
 begin
  a:=G 中任意一个无前驱的顶点;
  将 a 加到 Q 的末尾; 从 G 中删去结点 a 以及以 a 为起点的所有有向边;
 end;
```

我们看到，其中那些还未能用高级语言表达的语句或语句成分，正是算法需要定义在数据 模型 Digraph 和 Queue 上的运算。现分别将它们列出。

对于 Digraph 中的 G:

1. 检测 G 是否非空图;
2. 在 G 中找任意一个无前驱的结点;
3. 在 G 中删去一个无前驱的结点，以及以该结点为起点的所有有向边。

对于 Queue 中的 Q:

1. 初始化 Q 为空队列;
2. 将一个结点加到 Q 的末尾。

如果还考虑到已知 G 的初始状态如何由输入形成和 Q 的结果状态的输出，那么，对于 Digraph 和 Queue 还需要补充定义若干有关的运算。为了简单，这里从略。

由于高级语言为抽象数据类型的定义提供了很好的环境和工具，再复杂的数据模型都可 以通过构造数据类型来表达，再复杂的运算都可以借助过程或函数来描述。因此，上述由数据模型和数据模型上定义的运算综合起来的抽象数据类型很容易用高级语言来定义。

对于抽象数据类型 mgraph，定义如下三个运算:

(l)function G_empty(G:Digraph):boolean;
{检测图 G 是否非空。如果 G=ϕ，则函数返回 true，否则返回 false}
(2)function G_front(G:Digraph):nodetype；
{在有向图 G 中找一个无前驱的结点。nodetype 是结点类型名，它有待用户定义，下同}
(3)Procedure delete_G_front(var G:Digraph;a:nodetype);
{在 G 中删去结点 a 以及以 a 为起点的所有有向边}

对抽象数据类型 Queue，定义如下两个运算:

(l)Procedure init_Q(var Q:Queue); {初始化队列 Q 为空队列}
(2)Procedure add_Q_rear(a:nodetype;var Q:Queue) {将结点 a 加到队列 Q 的末尾}

这样，我们便定义了 ADT Digraph 和 ADT Queue。

有了抽象数据类型 Digraph 和 Queue 的上述定义，拓扑排序问题的主干算法即可完全由高级语言表达成主程序。

```
Program topsort(input，ouput);
type
 nodetype=…
 Digraph=…
 Queue=…

Function G_empty(G:Digraph):boolean;
          ...

Function G_front(G:Dlgraph):nodetype;
          ...

Procedure delete_G_front(var G:Digraph;a:nodetype);
          ...

Procedure init_Q(var Q:Queue);
          ...

Procedure add_Q_rear(a:nodetype;var Q:Queue);
          ...

var
a:nodetype;
G:Digraph;
Q:Queue;

begin
```

```
…        {输入并形成 G 的初始状态即拓扑排序前的状态}
init_Q(Q);
while not G_empty(G) do
 begin
  a:=G_front(G);
  add_Q_rear(a，Q);
  delete_G_front(G，a);
 end;
…
{输出 Q 中的结果}
end;
```

为了简明，我们在其中略去了输入、拓扑排序前 G 的状态的形成和结果输出三个部分。至于构造数据类型 nodetype，Digraph 和 Queue 的表示，函数 G_empty，G_front，过程 delete_G_front，init_Q 和 add_Q_rear 等的实现，则留待算法的底层设计去完成。需要指出的是，nodetype 通常用记录表示，而 Digraph 和 Queue 都有多种表示方式。因而 G_empty，G_front，delete_G_front，init_Q 和 add_Q_rear 也有多种的实现方式。

但是，只要抽象数据类型 Digraph 和 Queue 的定义不变，不管上述构造数据类型的表示和过程与函数的实现如何改变，主程序的表达都不会改变;反过来，不管主程序在哪里调用抽象数据类型上的函数或过程，上述构造数据类型的表示和过程与函数的实现都不必改变。算法顶层的设计与底层的设计之间的这种独立性，显然得益于抽象数据类型的引人。而这种独立性给算法和程序设计带来了许多好处。

使用抽象数据类型带来的好处

使用抽象数据类型将给算法和程序设计带来很多好处，其中主要的有下面几条。

1. 算法顶层的设计与底层的设计被隔开，使得在进行顶层设计时不必考虑它所用到的数据和运算分别如何表示和实现;反过来，在进行数据表示和运算实现等底层设计时，只要抽象数据类型定义清楚，也不必考虑它在什么场合被引用。这样做，算法和程序设计的复杂性降低了，条理性增强了。既有助于迅速开发出程序的原型，又有助于在开发过程中少出差错，保证编出的程序有较高的可靠性。
2. 算法设计与数据结构设计隔开，允许数据结构自由选择，从中比较，可优化算法和提高程序运行的效率。
3. 数据模型和该模型上的运算统一一在抽象数据类型中，反映了它们之间内在的互相依赖和互相制约的关系，便于空间和时间耗费的折衷，满足用户的要求。
4. 由于顶层设计和底层设计被局部化，在设计中，如果出现差错，将是局部的，因而容易查我也容易纠正。在设计中常常要做的增、删、改也都是局部的，因而也都很容易进行。因此，可以肯定，用抽象数据类型表述的程序具有很好的可维护性。

5. 编出来的程序自然地呈现模块化，而且，抽象的数据类型的表示和实现都可以封装起来，便于移植和重用。
6. 为自顶向下逐步求精和模块化提供一种有效的途径和工具。
7. 编出来的程序结构清晰，层次分明，便于程序正确性的证明和复杂性的分析。

数据结构、数据类型和抽象数据类型

数据结构、数据类型和抽象数据类型，这三个术语在字面上既不同又相近，反映出它们在含义上既有区别又有联系。

数据结构是在整个计算机科学与技术领域上广泛被使用的术语。它用来反映一个数据的内部构成，即一个数据由哪些成分数据构成，以什么方式构成，呈什么结构。数据结构有逻辑上的数据结构和物理上的数据结构之分。逻辑上的数据结构反映成分数据之间的逻辑关系，物理上的数据结构反映成分数据在计算机内的存储安排。数据结构是数据存在的形式。

数据是按照数据结构分类的，具有相同数据结构的数据属同一类。同一类数据的全体称为一个数据类型。在程序设计高级语言中，数据类型用来说明一个数据在数据分类中的归属。它是数据的一种属性。这个属性限定了该数据的变化范围。为了解题的需要，根据数据结构的种类，高级语言定义了一系列的数据类型。不同的高级语言所定义的数据类型不尽相同。Pascal 语言所定义的数据类型的种类如图 1-8 所示。

其中，简单数据类型对应于简单的数据结构;构造数据类型对应于复杂的数据结构;在复杂的数据结构里，允许成分数据本身具有复杂的数据结构，因而，构造数据类型允许复合嵌套;指针类型对应于数据结构中成分数据之间的关系，表面上属简单数据类型，实际上都指向复杂的成分数据即构造数据类型中的数据，因此这里没有把它划入简单数据类型，也没有划入构造数据类型，而单独划出一类。

数据结构反映数据内部的构成方式，它常常用一个结构图来描述:数据中的每一项成分数据被看作一个结点，并用方框或圆圈表示，成分数据之间的关系用相应的结点之间带箭号的连线表示。如果成分数据本身又有它自身的结构，则结构出现嵌套。这里嵌套还允许是递归的嵌套。

由于指针数据的引入，使构造各种复杂的数据结构成为可能。按数据结构中的成分数据之间的关系，数据结构有线性与非线性之分。在非线性数据结构中又有层次与网状之分。

由于数据类型是按照数据结构划分的，因此，一类数据结构对应着一种数据类型。数据类型按照该类型中的数据所呈现的结构也有线性与非线性之分，层次与网状之分。一个数据变量，在高级语言中的类型说明必须是读变量所具有的数据结构所对应的数据类型。

最常用的数据结构是数组结构和记录结构。数组结构的特点是：

1. 成分数据的个数固定，它们之间的逻辑关系由成分数据的序号(或叫数组的下标)来体现。这些成分数据按照序号的先后顺序一个挨一个地排列起来。
2. 每一个成分数据具有相同的结构(可以是简单结构，也可以是复杂结构)，因而属于同一个数据类型(相应地是简单数据类型或构造数据类型)。这种同一的数据类型称为基类型。
3. 所有的成分数据被依序安排在一片连续的存储单元中。

概括起来，数组结构是一个线性的、均匀的、其成分数据可随机访问的结构。由于这种结构有这些良好的特性，所以最常被人们所采用。在高级语言中，与数组结构相对应的数据类型是数组类型，即数组结构的数据变量必须说明为 array [i] of T_0 ，其中 i 是数组结构的下标类型，而 T_0 是数组结构的基类型。

记录结构是另一种常用的数据结构。它的特点是:

1. 与数组结构一样，成分数据的个数固定。但成分数据之间没有自然序，它们处于平等地位。每一个成分数据被称为一个域并赋予域名。不同的域有不同的域名。
2. 不同的域允许有不同的结构，因而允许属于不同的数据类型。
3. 与数组结构一样，它们可以随机访问，但访问的途径靠的是域名。

在高级语言中记录结构对应的数据类型是记录类型。记录结构的数据的变量必须说明为记录类型。

抽象数据类型的含义在上一段已作了专门叙述。它可理解为数据类型的进一步抽象。即把数据类型和数据类型上的运算捆在一起，进行封装。引入抽象数据类型的目的是把数据类型的表示和数据类型上运算的实现与这些数据类型和运算在程序中的引用隔开，使它们相互独立。对于抽象数据类型的描述，除了必须描述它的数据结构外，还必须描述定义在它上面的运算(过程或函数)。抽象数据类型上定义的过程和函数以该抽象数据类型的数据所应具有的数据结构为基础。

随机数算法

```
/*1.从同一个种子开始*/
#include <stdio.h>
#include <conio.h>
static unsigned long int next=1;
int rand0(void)
{
next=next*1103515245+12345;
```

```
return (unsigned int)(next/65536)%32768;
}
int main(void)
{
int count;
for(count=0;count<5;count++)
   printf("%hd\n",rand0());
getch();
return 0;
}
/*2.重置种子*/
#include <stdio.h>
#include <conio.h>
static unsigned long int next=1;
int rand1(void)
{
next=next*1103515245+12345;
return (unsigned int)(next/65536)%32768;
}
void srand1(unsigned int seed)
{
next=seed;
}
int main(void)
{
int count;
unsigned int seed;
printf("please input seed:");
scanf("%u",&seed);
srand1(seed);
for(count=0;count<5;count++)
   printf("%hd\n",rand1());
getch();
return 0;
}
/*3.利用利用时钟产生种子
ANSI C 程序库提供了 rand()函数来产生随机数；
ANSI C 程序库提供了 srand()函数来产生种子；
ANSI C 程序库提供了 time()函数返回系统时间。
*/
#include <time.h>
#include <stdio.h>
```

```
#include <dos.h>
#include <conio.h>
#include <stdlib.h>
int main(void)
{
  int i;
  time_t t;
  clrscr();
  t = time(NULL);
  srand((unsigned) t);
  for(i=0; i<10; i++) printf("%d\n", rand()%10);
  getch();
  return 0;
}
```

台阶问题

某人上楼梯,他一步可以迈一个台阶,两个台阶或三个台阶,共有 n 个台阶,编程输出他所有可能上法.

如:有 4 个台阶,输出应是:

```
1   1   1   1
1   1   2
1   2   1
1   3
2   1   1
2   2
3   1
```

```
/* stair.c
 The problem of stair
 Copyright By Jimmy, 2002.11
 All Rights Reserved.
*/
#define STAIR_NUM 5
int total=0;
int index;
int que[STAIR_NUM];
void outputstep()
{
  int i;
```

```
   for(i=0;i<index;i++)
     printf("-%d",que[i]);
     printf("-\n");
}

void step(int n)
{
  if (n==0)
  {
    total++;
    printf("---------the NO.%d ----------\n",total);
    outputstep();
    return ;
  }
  que[index++]=1;
  step(n-1);
  --index;
  if (n>1)
  {
    que[index++]=2;
    step(n-2);
    --index;
  }
  if(n>2)
  {
    que[index++]=3;
    step(n-3);
    --index;
  }
}
main()
{
  printf("\n");
  printf("----------------------------------------\n");
  printf("             stair step               \n");
  printf("----------------------------------------\n");
  step(STAIR_NUM);
  printf("\n the total is %d \n",total);
}
```

通用冒泡排序

在着里就不在重述冒泡法的原理，前面已经说过了，需要提出的是这里用到了 C++中的模板函数 template 函数，使它能对任何类型的一组数据进行排序，其他的地方都没什么变化。

C 函数如下：

```
template<class Type>

void gensort(Type * base,int n)
{
 int i,j;
 for(i=1;i<n;i++)
     for(j=0;j<i-1;j++)
         if(base[j]>base[j+1])
         {
          Type temp=base[j];
          base[j]=base[j+1];
          base[j+1]=temp;
            }
}
```

图遍历应用

```
/* ======================================== */
/*          图形的遍历              */
/* ======================================== */
#include <stdlib.h>
#define MAXQUEUE 70            /* 伫列的最大容量      */
struct node                 /* 图形顶点结构宣告     */
{
   int vertex;              /* 顶点资料          */
   struct node *nextnode;       /* 指下一顶点的指标     */
};
typedef struct node *graph;      /* 图形的结构新型态     */
struct node head[61];          /* 图形顶点结构数组     */
int visited[61];             /* 遍历记录数组        */
int queue[MAXQUEUE];           /* 伫列的数组宣告      */
```

```
int front = -1;                 /*  佇列的前端          */
int rear = -1;                  /*  佇列的后端          */
/* ---------------------------------------- */
/*   建立图形                              */
/* ---------------------------------------- */
void creategraph(int *node,int num)
{
   graph newnode;               /*  新顶点指标          */
   graph ptr;
   int from;                    /*  边线的起点          */
   int to;                      /*  边线的终点          */
   int i;
   for ( i = 0; i < num; i++ )    /*  读取边线的回路       */
   {
      from = node[i*2];         /*  边线的起点          */
      to = node[i*2+1];         /*  边线的终点          */
      /*  建立新顶点记忆体  */
      newnode = ( graph ) malloc(sizeof(struct node));
      newnode->vertex = to;       /*  建立顶点内容        */
      newnode->nextnode = NULL;   /*  设定指标初值        */
      ptr = &(head[from]);        /*  顶点位置            */
      while ( ptr->nextnode != NULL ) /*  遍历至链表尾     */
         ptr = ptr->nextnode;     /*  下一个顶点          */
      ptr->nextnode = newnode;    /*  插入结尾            */
   }
}
/* ---------------------------------------- */
/*   佇列资料的存入                        */
/* ---------------------------------------- */
int enqueue(int value)
{
   if ( rear >= MAXQUEUE )       /*  检查佇列是否全满     */
      return -1;                 /*  无法存入            */
   rear++;                       /*  后端指标往前移       */
   queue[rear] = value;          /*  存入佇列            */
}
/* ---------------------------------------- */
/*   佇列资料的取出                        */
/* ---------------------------------------- */
int dequeue()
{
   if ( front   == rear )        /*  检查佇列是否是空     */
```

```
    return -1;                /*  无法取出            */
  front++;                    /*  前端指标往前移       */
  return queue[front];        /*  伫列取出            */
}
/* ---------------------------------------- */
/*   图形的广度优先搜寻法                   */
/* ---------------------------------------- */
void bfs(int current)
{
  graph ptr;
  /*  处理第一个顶点  */
  enqueue(current);           /*  将顶点存入伫列       */
  visited[current] = 1;       /*  记录已遍历过         */
  printf("[%d]    ",current);   /*  印出遍历顶点值       */
  while ( front != rear )     /*  伫列是否是空的       */
  {
    current = dequeue();      /*  将顶点从伫列取出     */
    ptr = head[current].nextnode;   /*  顶点位置         */
    while ( ptr != NULL )     /*  遍历至链表尾         */
    {
      if ( visited[ptr->vertex] == 0 ) /*  如过没遍历过  */
      {
        enqueue(ptr->vertex);     /*  递回遍历呼叫     */
        visited[ptr->vertex] = 1; /*  记录已遍历过     */
        /*  印出遍历顶点值  */
   printf("[%d]    ",ptr->vertex);
      }
      ptr = ptr->nextnode;    /*  下一个顶点           */
    }
  }
}
/* ---------------------------------------- */
/*   图形的深度优先搜寻法                   */
/* ---------------------------------------- */
void dfs(int current)
{
  graph ptr;
  visited[current] = 1;       /*  记录已遍历过         */
  printf("[%d]    ",current);   /*  印出遍历顶点值      */
  ptr = head[current].nextnode;   /*  顶点位置         */
  while ( ptr != NULL )       /*  遍历至链表尾         */
  {
```

```
      if ( visited[ptr->vertex] == 0 )  /*  如过没遍历过  */
        dfs(ptr->vertex);              /*  递回遍历呼叫  */
      ptr = ptr->nextnode;             /*  下一个顶点    */
    }
}

/* ---------------------------------------- */
/*   主程式: 建立图形后,将遍历内容印出.      */
/* ---------------------------------------- */
void main()
{clrscr();
while(1)
{
   char c,a;
   graph ptr;
   int i;
   int node[60][2] = { {1, 10}, {10, 1},   /*  边线数组       */
        {2, 10}, {10, 2},
        {2, 3}, {3, 2},
        {3, 4}, {4, 3},
        {3, 12}, {12, 3},
        {4, 13}, {13, 4},
        {4, 5}, {5, 4},
        {5, 6}, {6, 5},
        {5, 7}, {7, 5},
        {7, 8}, {8, 7},
        {9, 10}, {10, 9},
        {10, 11}, {11, 10},
        {11, 14}, {14, 11},
        {11, 12}, {12, 11},
        {12, 15}, {15, 12},
        {12, 13}, {13, 12},
        {13, 16}, {16, 13},
        {14, 17}, {17, 14},
        {14, 18}, {18, 14},
        {15, 19}, {19, 15},
        {16, 20}, {20, 16},
        {17, 18}, {18, 17},
        {18, 23}, {23, 18},
        {18, 19}, {19, 18},
        {19, 23}, {23, 19},
        {19, 24}, {24, 19},
```

```
        {19, 20}, {20, 19},
        {20, 21}, {21, 20},
        {22, 23}, {23, 22},
        {24, 25}, {25,24}
        };
clrscr();
printf("\n\n\n");
printf("/*----------------------------------------------------*/\n");
printf("/*              欢 迎 使 用 本 程 序              */\n");
printf("/*----------------------------------------------------*/\n");
printf("    本 程 序 是 有 关 图 的 遍 历 的 算 法 演 示,\n");
printf("如 果 有 不 足 之 处 敬 请 原 谅!\n\n");
printf("请 问 你 是 否 要 运 行 以 下 的 程 序:\n\n");
printf("  图 的 深 度 遍 历 和 广 度 遍 历? Y/N?\n");
c=getch();
if(c!='y'&&'Y')
exit(0);
 clrscr();
printf("\n\n");
printf("请 注 意 以 下 为 各 城 市 的 代 码:\n\n");
printf("1:乌鲁木齐; 2:呼和浩特; 3:北京; 4:天津; 5:沈阳; \n");
printf("6:大连; 7:长春; 8:哈尔滨; 9:西宁; 10:兰州;\n");
printf("11:西安; 12:郑州; 13:徐州; 14:成都; 15:武汉; \n");
printf("16:上海; 17:昆明; 18:贵阳; 19:株州; 20:南昌;\n");
printf("21:福州; 22:南宁; 23:柳州; 24:广州; 25:深圳.\n");
   for (i=1;i<=25;i++ )
   {
     head[i].vertex=i;        /* 设定顶点值          */
     head[i].nextnode=NULL;   /* 清除图形指标        */
     visited[i]=0;            /* 设定遍历初值        */
   }
   creategraph(node,60);        /* 建立图形           */
   printf("图 形 的 邻 接 链 表 内 容:\n");
   for (i=1;i<=25;i++)
   {  if(i%3==0)printf("\n");
     printf("顶点%d=>",head[i].vertex); /* 顶点值      */
     ptr=head[i].nextnode;          /* 顶点位置     */
     while(ptr!=NULL)       /* 遍历至链表尾        */
     {
  printf("%d  ",ptr->vertex);  /* 印出顶点内容     */
  ptr=ptr->nextnode;        /* 下一个顶点        */
     }
```

```
    }
printf("\n\n");
printf("请 选 择 你 需 要 的 操 作\n");
printf("1、图形的广度优先遍历请输入:'g'或'G'\n");
printf("2、图形的深度优先遍历请输入:'s'或'S'\n");
 c=getch();
   switch(c)
{
   case'g':case'G':
    printf("\n 请 你 输 入 你 需 要 的 起 始 顶 点:\n");
    scanf("%d",&i);
    clrscr();
    printf("\n\n");
    printf("请 注 意 以 下 为 各 城 市 的 代 码:\n\n");
    printf("1:乌鲁木齐; 2:呼和浩特; 3:北京; 4:天津; 5:沈阳; \n");
    printf("6:大连; 7:长春; 8:哈尔滨; 9:西宁; 10:兰州;\n");
    printf("11:西安; 12:郑州; 13:徐州; 14:成都; 15:武汉; \n");
    printf("16:上海; 17:昆明; 18:贵阳; 19:株州; 20:南昌;\n");
    printf("21:福州; 22:南宁; 23:柳州; 24:广州; 25:深圳.\n");
    printf("图 形 的 广 度 优 先 遍 历 的 顶 点 内 容:\n");
    bfs(i);                    /* 印出遍历过程        */
    printf("\n");     /*     换行      */
    break;
 case's':case'S':
     printf("\n 请 输 入 你 需 要 的 起 始 顶 点:\n");
     scanf("%d",&i);
     clrscr();
    printf("\n\n");
     printf("请 注 意 以 下 为 各 城 市 的 代 码:\n\n");
     printf("1:乌鲁木齐; 2:呼和浩特; 3:北京; 4:天津; 5:沈阳; \n");
     printf("6:大连; 7:长春; 8:哈尔滨; 9:西宁; 10:兰州;\n");
     printf("11:西安; 12:郑州; 13:徐州; 14:成都; 15:武汉; \n");
     printf("16:上海; 17:昆明; 18:贵阳; 19:株州; 20:南昌;\n");
     printf("21:福州; 22:南宁; 23:柳州; 24:广州; 25:深圳.\n");
     printf("图 形 的 深 度 优 先 遍 历 的 顶 点 内 容:\n");
     dfs(i);                    /* 印出遍历过程       */
     printf("\n");                /* 换行             */
      break;
}
printf("\n 请 问 你 是 否 要 继 续:y/n");
a=getch();
if(a!='y'&&'Y')
```

```
exit(0);
}
}
```

图象扭曲算法

图象扭曲是平面图形变化的一种，它可用于许多场合，如在以前介绍的火焰特效中加入扭曲效果，会使火焰更逼真(当然代码要有更高的的效率才行)，如果在字幕当中加入扭曲效果，会给人一种怪异的感觉。

图象扭曲的算法并不复杂，但要解释清楚却不是一件容易的事，为了说明问题只好借用图片了，网路慢的朋友多多包涵了。算法例程源码可点这里下载，编译需 VC++、DXSDK、DXGuide。

图一 图二 图三

首先我们来看图一，大家可看出在图中有一些网格线，这里假定这些网格线是一些有弹性的细绳，在图一中假定网格线是与底层分离的，接下来我们要在网格线的结点处施加外力，网格线受外力后就会变成象图二的形状，大家要仔细看图一和图二的底图，变化的仅仅是网格线，而底图目前为止还没改变。

再下来就是关键的地方了，到目前为止，我们还是假定网格线是与底图分离开的，接下来我们要把图二中网格线附着在底图上，然后撤消外力，记住网格线是有弹性的，这时底图在网格线的带动下发生变形，直到网格线回复到原样，如图三。大家再仔细看看图三的底图，是不是已被扭曲？

是不是恍然大悟？接下来就好解释了，我们再来看看图二到图三中某个固定的网格是如何形变的，

```
// 单元块扭曲算法
inline void CFeedBackApp::TextureBlock(int xo, int yo)
{
// 投影平面
float fLeftOffX, fLeftOffY; // 各行左端点相对于上一行左端点的偏移
float fRightOffY, fRightOffX; // 各行右端点相对于上一行右端点的偏移
float TX1, TY1, TX2, TY2; // 当前行左、右端点的坐标
float HDx, HDy; // 当前行各点间的平均偏移量
float tx, ty; // 当前投影点坐标
// 渲染平面
int x, y; // 当前渲染点坐标
int xi=(xo<<4), yi=(yo<<4); // 当前渲染块左上角坐标
WORD *Tptr;
Tptr = &(m_awBuf1[xi + m_nMul640[yi]]);
fLeftOffX = (m_offset[xo] [yo+1].xint - m_offset[xo] [yo].xint) /16; // 计算平均偏移
fLeftOffY = (m_offset[xo] [yo+1].yint - m_offset[xo] [yo].yint) /16;
fRightOffX = (m_offset[xo+1][yo+1].xint - m_offset[xo+1][yo].xint) /16;
fRightOffY = (m_offset[xo+1][yo+1].yint - m_offset[xo+1][yo].yint) /16; // 计算平均偏移
TX1 = m_offset[xo] [yo].xint; // 取投影图块第一行左端点坐标
TY1 = m_offset[xo] [yo].yint;
TX2 = m_offset[xo+1][yo].xint; // 取投影图块第一行右端点坐标
```

```
TY2 = m_offset[xo+1][yo].yint;
for (y=yi; y < (yi+16); y++)
{
HDx = (TX2-TX1) / 16; // 计算投影图块当前行各点的平均偏移
HDy = ((TY2-TY1) / 16);
tx = TX1; // 投影平面当前行左端点坐标
ty = TY1;
for (x=xi; x < (xi+16); x++)
{
*Tptr++ = m_awBuf2[int(tx) + m_nMul640[int(ty)] ];
tx += HDx; // 下一点
ty += HDy;
}
Tptr += (SCRWIDTH-16); // 下一行
TX1 += fLeftOffX; // 计算投影平面中下一行左、右端点的坐标
TY1 += fLeftOffY;
TX2 += fRightOffX;
TY2 += fRightOffY;
}
}
```

图四

图二中的网格是不规则形状的四边形，图三中的则是正方形。网格的形变其实就是四边形的挤压和拉伸的过程，如图四，蓝色是变形前的图象，绿色就是变形后的图象，则算法只是简单的缩放运算而已。假定正方形是 16*16 点的图块，将正方形各点投射到不规则四边形上，则在正方形上各点的颜色值就取不规则四边形上相应的投影点处的颜色值就行了。具体算法见例程源码。

图象增强--梯度锐化

1、微分法

在图象中，边缘是由灰度级和相邻域点不同的象素点构成的。因而，若想增强边缘，就应该突出相邻点间的灰度级的变化。微分运算可用来求信号的变化率，因而具有加强高频分量的作用。如果将其应用在图象上，可使图象的轮廓清晰。由于我们常常无法事先确定轮廓的取向，因而挑选用于轮廓增强的微分算子时，必须选择那些不具备空间方向性的和具有旋转不变的线形微分算子。

图象处理中最常用的微分方法是求梯度。对于图象 f(x, y)，它在点（x,y）处的梯度是一个矢量。

微分运算一般用差分来代替。常用的差分形式有两种：

1) GM(x,y)=|f(x,y)-f(x+1,y)|+|f(x,y)-f(x,y+1)|
2) GM(x,y)=|f(x,y)-f(x+1,y+1)|+|f(x+1,y)-f(x,y+1)|

利用差分运算时，图象的最后一行和最后一列的象素的梯度无法求得，一般用前一行或前一列的梯度值近似代替。

算出梯度后让梯度图象的灰度值 g(x,y)等于该点的梯度幅度，即 g(x,y)=GM(x,y)。这是常用的方法。

还有就是：

1)

2)

Lg 为一指定的灰度值。

3)

Lb 为一对背景指定的灰度值。

4)

Lg 和 Lb 的意义同上。

2、卷积

一般可使用如下高通滤波矩阵：

0 -1 0
-1 5 -1
0 -1 0

-1 -1 -1
-1 9 -1
-1 -1 -1

1 -2 1
-2 5 -2
1 -2 1

-1 -2 -1
-2 19 -2
-1 -2 -1

-2 1 -2
1 6 1
-2 1 -2

下面是微分法第一种方法的程序，适用于灰阶图象。

```
/* Contents
a_gradient Sharpen images with differential method
*/
#include <stdio.h>
#include <vicdefs.h>
#include <vicfcts.h>
#include <vicerror.h>
#include <string.h>
#include <math.h>
#include <float.h>
extern int _cdecl checkrange_(imgdes *);
/* Sharpen images. Returns NO_ERROR,
```

```
BAD_RANGE, BAD_FAC, NO_EMM, EMM_ERR, NO_XMM, or XMM_ERR
*/
int _cdcel a_gradient(imgdes * srcimg, imgdes * desimg, int kind_method)
{
int pixel_gray_1, pixel_gray_2, pixel_gray_3, pixel_gray_4, j, k, i, l;
int rcode=NO_ERROR;
int sx, sy, ex, ey;
int gradient;
int kind;
kind = kind_method;
/* Check range of start, end position */
if (checkrange_(srcimg))
return (BAD_RANGE);
if (kind != 1 && kind != 2)
return (BAD_FAC);

sx = srcimg->stx;
sy = srcimg->sty;
ex = srcimg->endx;
ey = srcimg->endy;
copyimgdes (srcimg, desimg);
rcode = copyimage(srcimg, desimg);
if (rcode != NO_ERROR) return (rcode);
for (k= sx; k<= ex-1; k++)
{
for (j=sy; j<= ey-1; j++)
{
gradient = 0;
if (kind ==1) {
pixel_gray_1 = getpixelgray (srcimg, k, j);
pixel_gray_2 = getpixelgray (srcimg, k+1, j);
pixel_gray_3 = getpixelgray (srcimg, k, j);
pixel_gray_4 = getpixelgray (srcimg, k, j+1);
}
else
{
pixel_gray_1 = getpixelgray (srcimg, k, j);
pixel_gray_2 = getpixelgray (srcimg, k+1, j+1);
pixel_gray_3 = getpixelgray (srcimg, k+1, j);
pixel_gray_4 = getpixelgray (srcimg, k, j+1);
}
if (pixel_gray_1 < 0) return (pixel_gray_1);
if (pixel_gray_2 < 0) return (pixel_gray_2);
if (pixel_gray_3 < 0) return (pixel_gray_3);
```

```
if (pixel_gray_4 < 0) return (pixel_gray_4);

gradient = (int)(abs(pixel_gray_1 - pixel_gray_2)) +
(int)(abs(pixel_gray_3 - pixel_gray_4));

if (gradient > 255)
gradient = 255;

rcode = setpixelgray(desimg, k, j, (UCHAR)gradient);
if (rcode!=NO_ERROR) return (rcode);
}
}
/* for last column and row */
for (j=sy; j<= ey-1; j++)
{
rcode=getpixelgray(desimg, ex-1, j);
if (rcode!=NO_ERROR) return (rcode);
rcode=setpixelgray(desimg, ex, j, rcode);
if (rcode!=NO_ERROR) return (rcode);
}
for (k=sx; k<=sx-1; k++)
{
rcode = getpixelgray(desimg, k, ey-1);
if (rcode!= NO_ERROR) return (rcode);
rcode = setpixelgray (desimg, k, ey, rcode);
if (rcode != NO_ERROR) return (rcode);
}

rcode = getpixelgray (desimg, ex-1, ey-1);
if (rcode!= NO_ERROR) return (rcode);
rcode = setpixelgray (desimg, ex, ey, rcode);
if (rcode != NO_ERROR) return (rcode);
}
```

程序中用参数 KIND 来选择近似计算梯度的两种方法。

五子棋算法

作者：添翼虎
网址：http://tyhweb.163.net
邮箱：tyhweb@163.net
==========

任何一种棋类游戏其关键是对当前棋局是否有正确的评分,评分越准确则电脑的 AI 越高。五子棋游戏也是

如此，但在打分之前，我们先扫描整个棋盘，把每个空位从八个方向上的棋型填入数组
gStyle(2, 15, 15, 8, 2)，其中第一个下标为 1 时表示黑棋，为 2 时表示白棋，第二和第三个下标表示(x,y)
第四个下标表示 8 个方向，最后一个下标为 1 时表示棋子数，为 2 时表示空格数，如：
gStyle(1,2,2,1,1)=3 表示与坐标(2,2)在第 1 个方向上相邻的黑棋棋子数为 3
gstyle(1,2,2,1,2)=4 表示与坐标(2,2)在第 1 个方向上的最近的空格数为 4
在定义方向时，也应该注意一定的技巧，表示两个相反的方向的数应该差 4，在程序中我是这样定义的：

```
Const DIR_UP = 1
Const DIR_UPRIGHT = 2
Const DIR_RIGHT = 3
Const DIR_RIGHTDOWN = 4
Const DIR_DOWN = 5
Const DIR_DOWNLEFT = 6
Const DIR_LEFT = 7
Const DIR_LEFTUP = 8
```

这样我们前四个方向可以通过加四得到另一个方向的值。如果你还是不太明白，请看下面的图：

```
---------
---------
---oo----
-ox*xx---
---------
---------
```

图中的*点从标为(4,4),（打*的位置是空位）,则:
gStyle(2,4,4,1,1)=1 在(4,4)点相邻的上方白棋数为 1
gStyle(2,4,4,1,2)=2 在(4,4)点的上方距上方白棋最近的空格数为 2
gStyle(1,4,4,3,1)=2 在(4,4)点相邻的右方黑棋数为 2
gStyle(1,4,4,3,2)=1 在(4,4)点的右方距右方黑棋最近的空格数为 3
...
一旦把所有空点的棋型值填完，我们很容易地得出黑棋水平方向上点(4,4)的价值，由一个冲 1（我把有界的棋称为冲）
和活 2（两边无界的棋称为活）组成的。对于而白棋在垂直方向上点(4,4)的价值是一个活 1，而在/方向也是活 1
所以，只要我们把该点的对于黑棋和白棋的价值算出来，然后我们就取棋盘上各个空点的这两个值的和的最大一点
作为下棋的点。
然而，对各种棋型应该取什么值呢？我们可以先作如下假设：
Fn 表示先手 n 个棋子的活棋型，如：F4 表示先手活四
Fn'表示先手 n 个棋子的冲棋型，如：F4'表示先手冲四
Ln 表示后手 n 个棋子的活棋型，如：L3 表示后手活三
Ln'表示后手 n 个棋子的冲棋型，如：L3'表示后手冲三
.
.
.

根据在一行中的棋型分析，得到如下关系：
L1'<=F1'<L2'<=F2'<=L1<F1<L2<F2<L3'<=F3'<L4'<F4'=F4
从这个关系包含了进攻和防守的关系（当然，这个关系是由我定的，你可以自己定义这些关系）。对这些
关系再进一步细化，如在一个可下棋的点，其四个方向上都有活三，也比不上一个冲四，所以我们可以又得到
4*F3<L4'这个关系，同样，我们还可以得到其它的关系，如：4*F2<L3、4*L3<F3...，这些的关系由于你的定法
和我的定法制可能不一样，这样计算机的 AI 也就不一样，最后我们把分值最小的 L1'值定为 1，则我们就得到了
下面各种棋型的分值，由 C 语言表示为：
F[2][5]={{0,2,5,50,16000},{0,10,30,750,16000}};
L[2][5]={{0,1,5,50,3750},{0,10,30,150,4000}};
F 数组表示先手，第一个下标为 0 时表示冲型，第二个下标表示棋子数，则 F2'对应 F[0][2]
L 数组表示后手，第一个下标为 0 时表示冲型，第二个下标表示棋子数，则 L2 对应 F[1][2]
Ok，棋型的分值关系确定好了以后，我们把每一个可下点的四个方向的棋型值相加（包括先手和后手的分
值），最后选择一个最大值，并把这一点作为计算机要下的点就 OK 了:)。
后话：
1、得到最大值也许不止一个点，但在我的程序中只选择第一个最大点，当然你可以用于个随机数来决定
选择那一个最大值点，也可以对这些最大值点再作进一步的分析。
2、在这个算法中我只考虑了周围有棋子的点，而其它点我没有考虑。
3、可以再更进一步，用这个算法来预测以后的几步棋，再选择预测值最好的一步，这样电脑的 AI 就更高了
4、这个算法没有考虑黑棋的禁手（双 3、双四和多于五子的连棋）。因为在平时我下的五子棋是没有这些
禁手的。

希 尔 排 序

基本思想：将整个无序序列分割成若干小的子序列分别进行插入排序。

序列分割方法：将相隔某个增量 h 的元素构成一个子序列。在排序过程中，逐次减小这个增量，最后当 h 减到 1 时，进行一次插入排序，排序就完成。

增量序列一般采用：$h_t=2^t-1, 1\leqslant t\leqslant[\log_2 n]$，其中 n 为待排序序列的长度。

C 函数如下：

```
void prshl(p,n)
int n;double p[];
{
 int k,j,i;
 double t;
```

```
    k=n/2;
    while(k>0)
    {
     for(j=k;j<=n-1;j++)
     {
       t=p[j];i=j-k;
       while((i>=0)&&(p[i]>t))
         {
          p[i+k]=p[i];i=i-k;
          }
        p[i+k]=t;
       }
       k=k/2;
      }
    return;
   }
```

数据结构学习（C++）——线性链式结构总结

在开始写这些文章之前，我曾经有个想法，能不能以单链表为基础，完成所有的线性链式结构？实践证明，是可以的，就像你看到的这样。我做这个尝试的起因是不惯现在教科书凌乱的结构：罗列了一大堆 ADT 或者是 templat class，好像这要你去记似的。殊不知，只有提取共性，突出个性，才能更明显的表现出各种数构的差异，显示数据结构的进化发展的过程，看出变化的内在需求。借用《C＋＋录》作者的一句话，“避免重复”。在以后的应用中，你可能需要单独的写出一ass——打个比方，为了实现栈，你没有必要先写一个线性表，然后再继承得到栈假如你已经有了一个线性表了，你还需要另起炉灶再重写一个栈吗？对于一本书本来就是假定读者在看后面的章节的时候，已经对前面的章节有所了解；即使C＋＋沉思录》这样的从以前发表在杂志上的文章整理出的书，也能看到循序渐进子。如果在后面的章节又把前面的东西重复一遍，我只能认为他是在骗稿费。

我把以前的代码总结一下，列表如下（因为我画不好图）：

单链表（List<Type>）				
多项式节点	表达式节点	修改部分操作	添加向前指针域	限制操作
多项式	表达式	循环链表	双向链表	栈和队列

我们从单链表出发，通过对应的变化，得到了绿色的一行。突然之间我觉得自己很可悲，看了 100 多页书，最后得到竟然只是这么一个表。不管怎么说，这也是我看了这么多页书的结晶，让我们看看能从这表里得到什么。

首先，单链表是一个容器，他是为了存取数据而存在的。如果里面存的是多项式的节点，那么他就是一个多项式；如果里面存的是表达式节点，那么他就是一个表达式。这让我想起了以前曾经困扰我的语义问题：多项式究竟是一个存了多项式节点的单链表呢，还是包含了一个存了多项式节点的单链表？现在我的想法是，这并不重要，怎么理解都行。

其次，循环链表、双向链表、栈和队列也是容器，只是具体的操作实现或者对外的行为和单链表有所差别。但是内在的存取机制是相同的，或者还有更多的地方相同，我们应当最大限度的利用这些相同之处。

上面的提法好像很无聊——他们当然是容器，还用你小子废话吗——请注意我下面的问题：数组是容器吗？你做过仅针对数组里的元素操作的练习吗？你做过数组应用的练习吗？现在请你把数组换成单链表，检查一下有没有答案变化了。这真是对大学教育的一个讽刺，我们学完单链表，竟然只会插入删除节点，合并两个链表，将一个单链表逆序，等等；我们居然不知道为什么要学这个，怎样用他。最终的结果就是，当我们走向工作岗位后，又如获至宝的捧起这本书，然后用语重心长的口气对学弟们说：“数据结构很重要，你一定要好好学”。

或许你说这没什么，大学里许多的功课不都是这样的吗——临到用时才发现重要。是的，习惯成自然，但是习惯并不都是好的，我们也并不是一定要“书到用时方恨少”，然而，这并不只是取决于我们的年少“不”轻狂。让我们回想一下我们的教科书，一章一节，井井有条，丝丝入扣，当然这是写的好的，其他的就是追求这个目标，画虎不成反类犬。前面的章节为后面的章节铺垫，逻辑推理性极强，请不要认为我这是夸那些书，你不觉得和看那些论文一个感觉？你所得到的仅仅是“是什么”，“我说的是有道理的”；你能看到“为什么有这些”吗？

对于论文，我不能要求作者给出他的思维过程，而他写论文的目的也仅仅是为了告诉别人：“我研究出这个了，你们看，我说的是有道理的”，并且，看他的论文的人也有和他相当的知识构成，我想，现在看我文章的人也不会对哲学论文感兴趣（即使你可能会看黑格尔的书，你也决不会对现在那群玄人的梦呓感兴趣；正如那群人虽然用电脑写文章，你想要跟他们说内存管理，他们也会把耳朵塞起来）。

对于教科书，仅仅用论文的要求来要求，显然是太低了。首先，书所面对的读者都是对这门课程一无所知的人——你假设和论文一样，都有一个基本认识了，我还学你这门课干什么，我直接看最新的论文好不好。其次，写作目的是让读者了解这门学科，认识这门学科的规律，最终能够运用这门学科，甚至有所发展。正如学写字先描红，不先模仿，怎么创新呢？如果连现有的结论和成果都不知道来龙去脉，谈何再有新的突破呢？

教科书不是论文，当然不能象写论文那样来写，但现在国内的作者好像乐此不疲，就好像不写得层次严谨就不能体现自己功力扎实似的。这也是为什么越来越多的人选用国外的教科书，即使他们的英文并不好——当然，不少人是借此学习英文。

正如哲学上讲的，只有遵从人类的认知规律，人们才能更容易的认识新事物。但论文的写作，恰恰是对这个过程的逆向总结——他是和人类的认识过程相反的。而教科书为了达到他传授知识的目的，就必须遵从人的认知规律，而决不是象论文那样反其道而行之。

这样看来，数据结构的书决不应该象现在这样来写。从数据结构的发展来看，他是应问题的需要而出现的，并为解决问题而服务。换而言之，对于数据结构的讲解，应当把重点放在算法上面，在各种典型问题上提出新的数据结构；最终得出的认识是，为了特定的问题和算法，而选用特定的数据结构；为了改进算法而改进数据结构；为了新的问题和算法而创造出新的数据结构。而决不是象现在这样能得出的认识：我学了数据结构能解决什么问题。乍一看，好像没什么差别，但现在这个认识，你不是数据结构的主人，你只是数据结构的奴隶。

最后举个讽刺性的例子。我最初知道数据结构的时候是高中，那时我只会 BASIC，数据结构的概念源自一本名叫《中小学生电脑操作与程序设计》的书。那本书使用的是 GWBASIC，实现了大学教科书里的数据结构：链表、栈、树、图。来看看他的目录：八皇后、迷宫、骑马游世界、链表和约瑟夫、树和背包、一笔画、追捕罪犯、四种颜

色就够了。——我们甚至在中小学时，用 BASIC 就可以清楚的有趣的讲解数据结构；现在居然让大学生学完之后，甚至连中小学生的水平都不如。我不知道你怎么看，但我觉得，至少，我们的教科书应该改变一下写作的风格了。

结构学习（C++）——循环链表

书对循环链表的介绍很简略，实现部分也不完整（当然了，如果完整就又是重复）。而我也没觉得循环链表有什么别的用，他更应该是为了一个特殊的问题而产这只是个人的看法。我从链表类派生出了循环链表，这需要注意几个细节。

构造函数：派生类实例化时，先调用基类的构造函数；因此，初始化循环链表工作就是将带表头的空链表的表头节点的 link 指向表头节点，从而构成一个圈

析构函数：释放对象时，先调用派生类的析构函数，然后调用基类的析构函数因此，释放循环链表只需要将循环链表变成普通的单链表，然后这个单链表会基类的析构函数释放。这里假定不使用这种语句 Base *p = new Drived;delete 因为我在~List()前面没有加 virtual。你可以参阅各种 C++书籍搞清这类问题。

判空函数：条件不是检测头节点的 link 是否为空，而是是否指向头节点。

置空函数：原来的显然不能工作了，实际上只要从表头位置不断后删直到表空可以了。

Next()：遇到表头节点要跳过去。

Remove()：当前节点是表头节点时不能删，删了表头节点的后果自己想吧（因在循环链表中 prior 指针不一定为空——其实应该是一定不空，但是由于继承部 List 函数，所以就是不一定了，以至于原来的 Remove()检查可能无效）；如果除的是表尾节点，删除后当前指针将指向表头节点，要跳过去指向下一个。总之使用 Next()和 Remove()时，不能让外界觉察到表头节点的存在，否则，当你循计数时，表头节点就被算进去了。

“<<”必须重写，否则，当你执行 cout << CircList;这种东西时，天哪，自己去吧；当然了，只需要拷贝过来修改一下循环判断就可以了。

End()将不能工作，考虑到如果按照原来的功能来实现，效率很低，而且用处不

所以修改 End()功能定义为更正 last 指针。为避免混淆，将其放在 private，对外不提供这个功能。

定义和实现

```
#ifndef CircList_H
#define CircList_H

include “List.h”

template <class Type> class CircList : public List<Type>
{
public:
    CircList() { pGetFirst()->link = pGetFirst(); }
    ~CircList() { End(); pGetLast()->link = NULL; }
    BOOL IsEmpty()
    {
        Node<Type> *p = pGetFirst();
        return p->link == p;
    }

    Type *Next()
    {
        if (pNext() == pGetFirst()) pNext();
        return Get();
    }


BOOLRemove()
{
```

```
if(!IsEmpty())
{
if (pGet() == pGetFirst()) return FALSE;
List<Type>::Remove();
if (pGet() == pGetFirst()) pNext();
returnTURE;
}
return FALSE;
}


        void MakeEmpty()
        {
                First();
                while (!IsEmpty()) RemoveAfter();
        }

        void LastInsert(const Type &value)
        {
                End();
                List<Type>::LastInsert(value);
        }
private:
        void End()
        {
                if (pGetLast()->link != pGetFirst())
                {
                        Node<Type> *pfirst = pGetFirst();
                        for (Node<Type> *p = pGet(); p->link != pfirst; p = pNext());
```

```
            PutLast(p);
        }
    }

friend ostream & operator << (ostream & strm, CircList<Type> &cl)
{
        cl.First();
        Node<Type> *pfirst = cl.pGetFirst();
        while (cl.pGet()->link != pfirst) strm << *cl.Next() << " " ;
        strm << endl;
        cl.First();
        return strm;
}

};

#endif
```

【说明】为了后面的约瑟夫问题，我添加了 LastInsert。如果使用 Insert 是倒插，当然可以倒输入来解决，但这样的做法有将就的嫌疑，而不断的 Locate 显然效率太低。显然，Find，Loacte，Length 这类继承过来的函数，运行起来会发生意想不到的事，我没有在这里给出重新的实现出于以下原因：

- Find 可以把原来的代码拷过来修改一下循环判定，但也可以不改。方法是，采用后面介绍的查找表的办法，在表头节点放查找值，这样就一定会查找成功了。然后检查当前节点是否为头节点就可以判断是否真正查找成功，如果你自己完成这个函数，将会有很多收获。给个例子：

```
BOOL Find(const Type &value)
    {
        pGetFirst()->data = value;
        List<Type>::Find(value);
        if (pGet() == pGetFirst()) return FALSE;
```

```
        return TURE;
    }
```

- Locate 原来的实现在这里其实也没什么语义的毛病，无非是转圈吗，当然怎么改随你。建议配合 Length 检查定位值的合法，这样可以把转圈提前扼杀。
- Length 你说循环链表有多长？就像无论多长的长跑比赛都可以在 400 米的跑道进行一样。这里建议增加一个私有数据成员 length，在每次插入删除时调整，因为改动的地方比较多，所以我就偷懒了，主要是觉得很少用。

约瑟夫问题

几乎提到循环链表总要提到约瑟夫问题，而我当年还在学 BASIC 时，就告诉我解决这个问题要构造一个循环链表，当然了，在 BASIC 里是静态链表。真的好像循环链表就是为了这个问题而存在的。为了照顾没听说过的人，我简单介绍一下这个问题：

说是一个旅行社要从 n 名旅客中选出一名幸运旅客，为他提供免费环球旅行服务。方法是，大家站成圈，然后选定一个 m，从第 1 个人开始报数，报到 m 时，这个人 OUT，然后从下一个人开始重新从 1 报数，重复这个过程，直到最后剩下一个人就是幸运之星。问题就是谁是幸运者呢？或者说是怎样才能赢大奖。

我用这个问题测试了整数情况下循环链表各个成员函数的正确性，相应函数如下：

```
void CircListTest_int()
{
    CircList<int> a;
    cout << endl << "整型循环链表测试：求解约瑟夫问题" << endl;
    int n, m;//n 是总人数，m 是报数值
    cout << "输入总人数：";
    cin >> n;
    cout << "输入报数值：";
    cin >> m;
    for (int i = 1; i <= n; i++) a.LastInsert(i);
```

```
    cout << a;
    a.Locate(0);
    for (i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < m - 1; j++) a.Next();
        cout << "第" << *a.Get() << "位旅客被淘汰" << endl;
        a.Remove();
    }
    cout << "第" << *a.Get() << "位旅客获胜" << endl;
    cout << a;
    a.MakeEmpty();
    cout << a;

}
```

【后记】就是做了循环链表这个派生类，我才发现了原来写的链表类的许多隐患，比如原来的 Find，在整数链表里当你寻找 0 时，无一例外的会停在表头，所以你现在看到的链表类已经和我当初写的有很大的变化了，也可能有些我还没有发现，欢迎指正。

“二分法”求二元方程的解

前面说到了用“精确迭代法”求两个数的最大公约数，这里的“二分法”也属于迭代法——近似迭代。另外还有“牛顿迭代”也属于近似迭代。

思想：二分法属于数学问题，但为了说清楚问题就再说一下原理。先取二元方程 f(x)的两个初略解 x1 和 x2，若 f(x1)与 f(x2)的符号相反，则方程 f(x)=0 在[x1，x2]区间至少有一个根；若 f(x)在[x1，x2]区间单调，则至少有一个实根；所以取 x3=(x1+x2)/2，并在 x1 和 x2 中舍去和 f(x3)同号者，那么解就在 x3 和另外那个没有舍去的初略解组成的区间里；如此反复取舍，直到 xn 与 xn-1 之差满足要求时，那么 xn 便是方程 f(x)的近似根。

所以有算法：

while(误差>给定误差)

```
if(f(x)==0)
   x 就是根，不在迭代；
else  if(f(x)*f(x1)<0)       /*这里的 x 相当于上面所说的 x3*/
          x2=x;
     else
          x1=x;
```

例：用二分法求方程 $x^2-2-x=0$ 在[0，3]区间的根。

```
float f(float x)
 {return (x*x-x-2);}
#include<iostream.h>
#include<math.h>
main()
{
 float x1=0,x2=3,x,root;
 const float err=.5e-5              //给定精度
 while(fabs(x1-x2)>err)             //求根
       {
        if(f(x1)==0)
           {root=x1;break;}
        if(f(x2)==0)
            {root=x2;break;}
         x=(x1+x2)/2;
         if(f(x)==0)
             {root=x;break;}
          if(f(x)*f(x1)<0)
             x2=x;
          else
             x1=x;
          }
 root=x1;
 cout<<"The root is:"<<root<<endl;
}
```

数据结构教程

第一课

本课主题：数据结构的基本概念和术语

教学目的：了解数据结构的基本概念，理解常用术语

教学重点：基本概念：数据与数据元素

教学难点：数据元素间的四种结构关系。

授课内容：

一、数据、数据元素、数据对象、数据结构的定义

1、数据的定义

定义一：数据是客观事物的符号表示。

学号	姓名	语文	数学	C 语言
6201001	张三	85	54	92
6201002	李四	92	84	64
6201003	王五	87	74	73
6201004				
. . .				

例：张三的 C 语言考试成绩为 92 分，92 就是该同学的成绩数据。

定义二：能输入到计算机中并被计算机程序处理的符号的总称。

例：图像、声音等。

总结：现实世界信息的分析、复制、传播首先要符号化，这样才便于处理，尤其

是便于计算机的处理。家长、社会要了解一个学生的学习成绩和能力，要看他的学习档案，而学习档案即是说明该学生学习情况的数据。

2、数据元素、数据项

数据元素是数据的基本单位，它也可以再由不可分割的数据项组成。如图示：

3、数据对象

是性质相同的数据元素的集合。如上例：一个班级的成绩表可以看作一个数据对象。

4、数据结构

定义一、数据元素集合（也可称数据对象）中各元素的关系。

定义二、相互之间存在特定关系的数据元素集合。

数据结构的种类：

	特征	示例
集合	元素间为松散的关系	
线性结构	元素间为严格的一对一关系	如上面的成绩表中各元素

树形结构	元素间为严格的一对多关系	
图状结构(或网状结构)	元素间为多对多关系	

数据结构的形式定义：

数据结构名称=（D，S）

其中 D 为数据元素的有限集，S 是 D 上关系的有限集

逻辑结构		“数据结构”定义中的“关系”指数据间的逻辑关系，故也称数据结构为逻辑结构。
存储结构		数据结构在计算机中的表示称为物理结构。又称存储结构。
	顺序存储结构	
	链式存储结构	

存储结构详解：

计算机中存储信息的最小单位：**位**，8 位为一字节，两个字节为一字，字节、字或更多的二进制位可称为**位串**。在逻辑描述中，把位串称为元素或**结点**。

当数据元素由若干数据项组成时，位串中对应于各个数据项的子位串称为**数据域**(Data Field)。

例：上述成绩表数据用 C 语言的结构体数组 classonestu[50]来存储：

```
struct stu {
int stuno;/*数据项，也称 stu 位串中的一个子位串，或叫做数据域*/
char name[20];
int maths;
int language;
```

```
int c_language;
} classonestu[50];
```

二、数据类型

1、定义：数据类型是一个值的集合和定义在这个值集上的一组操作的总称。

例：C 语言中的整型，其内涵为一定范围的自然数集合，及定义在该集合上的加减乘除及取模、比较大小操作。而实型则无取模操作。当然整型也不需四舍五入。

2、数据类型的种类：

	特征	例
原子类型	值在逻辑上不可分解	int float
结构类型	值由若干成分按某种结构组成	struct stu

数据类型封装了数据存储与操作的具体细节。

三、总结

数据->数据元素

具有特定关系的数据元素集合->数据结构

数据结构的逻辑表示与物理存储->逻辑结构与存储结构

人们不仅关心数据的逻辑结构、存储结构，还关心数据的处理方法（算法）与处理结果->数据类型

数据类型->分类

第二课

本课主题： 抽象数据类型的表示与实现

教学目的： 了解抽象数据类型的定义、表示和实现方法

教学重点： 抽象数据类型表示法、类 C 语言语法

教学难点： 抽象数据类型表示法

授课内容：

一、抽象数据类型定义（ADT）

作用：抽象数据类型可以使我们更容易描述现实世界。例：用线性表描述学生成绩表，用树或图描述遗传关系。

定义：一个数学模型以及定义在该模型上的一组操作。

关键：使用它的人可以只关心它的逻辑特征，不需要了解它的存储方式。定义它

的人同样不必要关心它如何存储。

例：线性表这样的抽象数据类型，其数学模型是：数据元素的集合，该集合内的元素有这样的关系：除第一个和最后一个外，每个元素有唯一的前趋和唯一的后继。可以有这样一些操作：插入一个元素、删除一个元素等。

抽象数据类型分类	
原子类型	值不可分解，如 int
固定聚合类型	值由确定数目的成分按某种结构组成，如复数
可变聚合类型	值的成分数目不确定如学生基本情况

抽象数据类型表示法：

一、

三元组表示：（D，S，P）

其中 D 是数据对象，S 是 D 上的关系集，P 是对 D 的基本操作集。

二、书中的定义格式：

ADT 抽象数据类型名{

数据对象：<数据对象的定义>

数据关系：<数据关系的定义>

基本操作：<基本操作的定义>

}ADT 抽象数据类型名

例：线性表的表示

名称	线性表	
数据对象	D={a_i| a_i(-ElemSet,i=1,2,...,n,n>=0}	任意数据元素的集合
数据关系	R1={<a_{i-1},a_i>| a_{i-1},a_i(- D,i=2,...,n}	除第一个和最后一个外，每个元素有唯一的直接前趋和唯一的直接后继
基本操作	ListInsert(&L,i,e)	L 为线性表，i 为位置，e 为数据元素。
	ListDelete(&L,i,e)	
	...	

二、类 C 语言语法

类 C 语言语法示例				
1、预定义常量和类型	#define TRUE 1 #define FALSE 0 #define OK 1 #define ERROR 0 #define INFEASIBLE -1 #define OVERFLOW -2 typedef in Status; //Status 是函数的类型，其值是函数结果状态代码。			
2、数据结构的存储结构	typedef ElemType first;			

3、基本操作的算法	函数类型 函数名（函数参数表）{
	//算法说明
	语句序列
	}//函数名			

4、赋值语句	简单赋值：	变量名=表达式；		
	串联赋值：	变量名 1=变量名 2=...=变量名 k=表达式；		
			成组赋值：	（变量名 1，...,变量名 k)=(表达式 1,...,表达式 k)； 结构名=结构名； 结构名=（值 1，...，值 k)； 变量名[]=表达式； 变量名[起始下标..终止下标]=变量名[起始下标..终止下标]；

	交换赋值：	变量名<-->变量名；
	条件赋值：	变量名=条件表达式？表达式？ 表达式 T：表达式 F

	5、选择语句	1、if（表达式） 语句；
		2、if（表达式） 语句；
		else 语句；
		3、switch(表达式) {
		case 值 1：语句序列 1；break;
		...
		case 值 n：语句序列 n；break;
		default:语句序列 n+1；break;
		}
		4、switch{
		case 条件 1：语句序列 1；break;
		...
		case 条件 n：语句序列 n；break;
		default:语句序列 n+1；break;
		}

6、循环语句	for(赋初值表达式；条件；修改表达式序列) 语句； while（条件）语句； do{ 语句序列}while（条件）;	

7、结束语句	return [表达式];
	return; //函数结束语句
	break; //case 结束语句
	exit(异常代码); //异常结束语句	

8、输入和输出语句	scanf（[格式串]，变量 1，...,变量 n);	

9、注释	//文字序列
10、基本函数	max（表达式 1，...，表达式 n） min,abs,floor,ceil,eof,eoln
11、逻辑运算	&&与运算；||或运算

例：线性表的实现：

ADT List{

数据对象：D={a_i| a_i(-ElemSet,i=1,2,...,n,n>=0}

数据关系：R1={<a_{i-1},a_i>| a_{i-1},a_i(- D,i=2,...,n}

基本操作：

InitList(&L)

DestroyList(&L)

ListInsert(&L,i,e)

ListDelete(&L,i,&e)

}ADT List

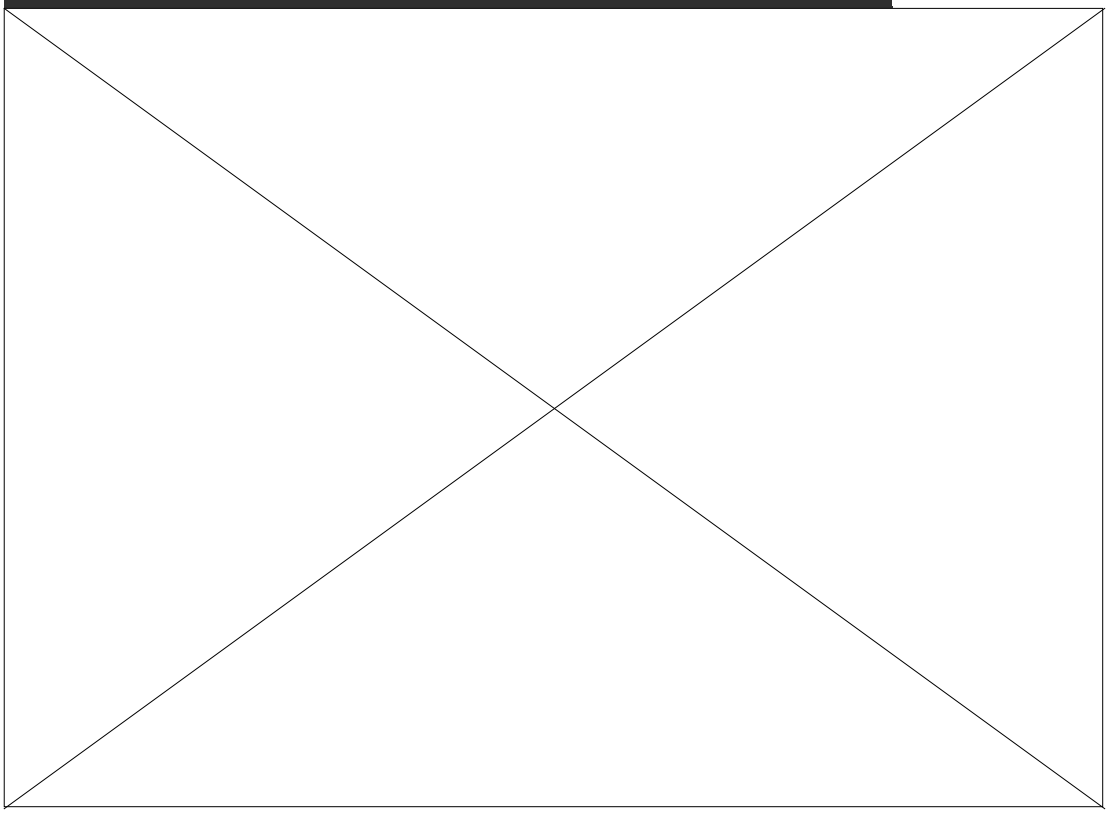
```
ListInsert(List &L,int i,ElemType e)
{if(i<1||i>L.length+) return ERROR;
q=&(L.elem[i-1]);
for(p=&(L.elem[L.length-1]);p>=q;--p) *(p+1)=*p;
*q=e;
++L.length;
return OK;
}
```

下面是 C 语言编译通过的示例：

```
#define ERROR 0
#define OK 1
struct STU
{ char name[20];
char stuno[10];
int age; int score;
}stu[50];
struct LIST
{ struct STU stu[50];
int length;
}L;

int printlist(struct LIST L)
{ int i;
printf("name stuno age score\n");
for(i=0;i<L.length;i++)
printf("%s %s\t%d\t%d\n", L.stu[i].name, L.stu[i].stuno,
L.stu[i].age, L.stu[i].score);
printf("\n");
}

int listinsert(struct LIST *L,int i,struct STU e)
{ struct STU *p,*q;
if (i<1||i>L->length+1)
return ERROR;
q=&(L->stu[i-1]);
for(p=&L->stu[L->length-1];p>=q;--p)
*(p+1)=*p; *q=e; ++L->length;
return OK;
```

```
}/*ListInsert Before i */

main()
{ struct STU e;
L.length=0;
strcpy(e.name,"zmofun");
strcpy(e.stuno,"100001");
e.age=80;
e.score=1000;
listinsert(&L,1,e);
printlist(L);
printf("List length now is %d.\n\n",L.length);
strcpy(e.name,"bobjin");
strcpy(e.stuno,"100002");
e.age=80;
e.score=1000;
listinsert(&L,1,e);
printlist(L);
printf("List length now is %d.\n\n",L.length);
}
```

```
E:\ZM\Zmdoc\datastru\class02>listdemo

name stuno age score

zmofun 100001 80 1000

List length now is 1.

name stuno age score

bobjin 100002 80 1000

zmofun 100001 80 1000

List length now is 2.
```



三、总结

抽象数据类型定义；

抽象数据类型实现方法：一、类 C 语言实现 二、C 语言实现

回目录 上一课 下一课

第三课

本课主题： 算法及算法设计要求

教学目的： 掌握算法的定义及特性，算法设计的要求

教学重点： 算法的特性，算法设计要求

教学难点： 算法设计的要求

授课内容：

一、算法的定义及特性

1、定义：

```
ispass(int num[4][4])
{ int i,j;
for(i=0;i<4;i++)
for(j=0;j<4;j++)
if(num[i][j]!=i*4+j+1)/*一条指令，多个操作*/
return 0;
return 1;
}/*上面是一个类似华容道游戏中判断游戏是否结束的算法*/
```

算法是对特定问题求解步骤的一种描述，它是指令的有限序列，其中每一条指令表示一个或多个操作；此外，一个算法还具有下列五个重要特性：

2、算法的五个特性：

有穷性	一个算法必须总是（对任何合法的输入值）在执行有穷步之后结束，且每一步都可在有穷时间内完成；
确定性	算法中每一条指令必须有确切的含义，读者理解时不会产生二义性。有任何条件下，算法只有唯一的一条执行路径，即对于相同的输入只能得出相同的输出。
可行性	一个算法是能行的，即算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现的。
输入	一个算法有零个或多个的输入，这些输入取自于某个特定的对象的集合。
输出	一个算法有一个或多个的输出。这些输出是同输入有着某些特定关系的量。

例：

有穷性	haha()
	{/*only a joke,do nothing.*/
	}
	main()
	{printf("请稍等...您将知道世界的末日...");
	while(1)
	haha();
	}

确定性	float average(int *a,int num)
	{int i;long sum=0;
	for(i=0;i<num;i++)
	sum+=*(a++);
	return sum/num;
	}
	main()
	{int score[10]={1,2,3,4,5,6,7,8,9,0};
	printf("%f",average(score,10);
	}

可行性	
输入	

输出	getsum(int num)
	{
	int i,sum=0;
	for(i=1;i<=num;i++)
	sum+=i;
	} /*无输出的算法没有任何意义，

二、算法设计的要求

1、正确性

算法正确性的四个层次	

程序不含语法错误。	max(int a,int b,int c)
	{
	if (a>b)
	{if(a>c) return c;
	else return a;
	}
	}

程序对于几组输入数据能够得出满足规格说明要求的结果。	max(int a,int b,int c)
	{
	if (a>b)
	{if(a>c) return a;
	else return c;
	}
	} /* 8,6,7 */ /* 9,3,2 */

程序对于精心选择的典型、苛刻而带有刁难性的几组输入数据能够得出满足规格说明要求的结果。	max(int a,int b,int c)
	{
	if (a>b)
	{if(a>c) return a;
	else return c;
	}
	else
	{if(b>c) return b;
	else return c;
	}
	}

程序对于一切合法的输入数据都能产生满足规格说明要求的结果。	

2、可读性

3、健壮性

4、效率与低存储量需求

效率指的是算法执行时间。对于解决同一问题的多个算法，执行时间短的算法效率高。

存储量需求指算法执行过程中所需要的最大存储空间。

两者都与问题的规模有关。

	算法一	算法二

在三个整数中求最大者	max(int a,int b,int c)
	{if (a>b)
	{if(a>c) return a;
	else return c;
	}
	else
	{if(b>c) return b;
	else return c;
	}/*无需额外存储空间，只需两次比较*/
		max(int a[3])
		{int c,int i;
		c=a[0];
		for(i=1;i<3;i++)
		if (a[i]>c) c=a[i];
		return c;
		}
		/*需要两个额外的存储空间，两次比较，至少一次赋值*/
		
		/*共需 5 个整型数空间*/

求 100 个整数中最大者	同上的算法难写，难读	max(int a[100])
		{int c,int i;
		c=a[0];
		for(i=1;i<100;i++)
		if (a[i]>c) c=a[i];
		return c;
		}
		/*共需 102 个整型数空间*/

三、总结

1、算法的特性

2、算法设计要求：正确性、可读性、健壮性、效率与低存储量需求。

回目录 上一课 下一课

第四课

本课主题： 算法效率的度量和存储空间需求

教学目的： 掌握算法的渐近时间复杂度和空间复杂度的意义与作用

教学重点： 渐近时间复杂度的意义与作用及计算方法

教学难点： 渐近时间复杂度的意义

授课内容：

一、算法效率的度量

算法执行的时间是算法优劣和问题规模的函数。评价一个算法的优劣，可以在相同的规模下，考察算法执行时间的长短来进行判断。而一个程序的执行时间通常有两种方法：

1、事后统计的方法。

缺点：不利于较大范围内的算法比较。（异地，异时，异境）

2、事前分析估算的方法。

程序在计算机上运行所需时间的影响因素	
算法本身选用的策略	
问题的规模	规模越大，消耗时间越多
书写程序的语言	语言越高级，消耗时间越多
编译产生的机器代码质量	
机器执行指令的速度	

综上所述，为便于比较算法本身的优劣，应排除其它影响算法效率的因素。

从算法中选取一种对于所研究的问题来说是基本操作的原操作，以该基本操作重复执行的次数作为算法的时间量度。（原操作在所有该问题的算法中都相同）

$$T(n)=O(f(n))$$

上示表示随问题规模 n 的增大，算法执行时间的增长率和 f(n) 的增长率相同，称作算法的**渐近时间复杂度**，简称**时间复杂度**。

求 4*4 矩阵元素和，T(4)=0(f(4))
f=n*n;
	sum(int num[4][4])
	{ int i,j,r=0;
	for(i=0;i<4;i++)
	for(j=0;j<4;j++)
	r+=num[i][j]; /*原操作*/
	return r;
	}

最好情况：
T（4）=0（0）
最坏情况：
T（4）=0（n*n）
	ispass(int num[4][4])
	{ int i,j;
	for(i=0;i<4;i++)
	for(j=0;j<4;j++)
	if(num[i][j]!=i*4+j+1)
	return 0;
	return 1;
	}

原操作执行次数和包含它的语句的频度相同。语句的频度指的是该语句重复执行的次数。

语句	频度	时间复杂度
{++x;s=0;}	1	0(1)
for(i=1;i<=n;++i) {++x;s+=x;}	n	0(n)

for(j=1;j<=n;++j)
for(k=1;k<=n;++k)
{++x;s+=x;}
	n*n	0(n*n)

		0(log n)

基本操作的执行次数不确定时的时间复杂度	
平均时间复杂度	依基本操作执行次数概率计算平均
最坏情况下时间复杂度	在最坏情况下基本操作执行次数

二、算法的存储空间需求

类似于算法的时间复杂度，空间复杂度可以作为算法所需存储空间的量度。

记作：

$$S(n)=O(f(n))$$

若额外空间相对于输入数据量来说是常数，则称此算法为**原地工作**。

如果所占空间量依赖于特定的输入，则除特别指明外，均按最坏情况来分析。

三、总结

渐近时间复杂度

空间复杂度

回目录 上一课 下一课

第五课

本课主题： 线性表的类型定义

教学目的： 掌握线性表的概念和类型定义

教学重点： 线性表的类型定义

教学难点： 线性表的类型定义

授课内容：

复习：数据结构的种类

线性结构的特点：

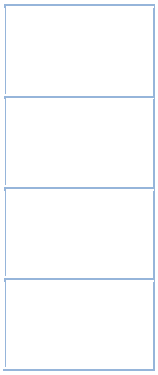
在数据元素的非空有限集中，

（1）存在唯一的一个被称做“第一个”的数据元素；

（2）存在唯一的一个被称做“最后一个”的数据元素；

（3）除第一个之外，集合中的每个数据元素均只有一个前驱；

（4）除最后一个之外，集合中每个数据元素均只有一个后继。



一、线性表的定义

线性表是最常用且最简单的一种数据结构。

一个线性表是 n 个数据元素的有限序列。

数据元素可以是一个数、一个符号、也可以是一幅图、一页书或更复杂的信息。

线性表例：

1、

1	2	3	4	5	6	7

2、

3、

学号	姓名	语文	数学	C 语言
6201001	张三	85	54	92
6201002	李四	92	84	64
6201003	王五	87	74	73
6201004				
...				

数据元素也可由若干个**数据项**组成（如上例 3）。这时常把数据元素称为**记录**。

含有大量记录的线性表又称**文件**。

线性表中的数据元素类型多种多样，但同一线性表中的元素必定具有相同特性，即属同一数据对象，相邻数据元素之间存在着序偶关系。

a_1	...	a_{i-1}	a_i	a_{i+1}	...	a_n

a_i 是 a_{i+1} 的**直接前驱**元素，a_{i+1} 是 a_i 的**直接后继**元素。

线性表中元素的个数 n 定义为线性表的长度，为 0 时称为空表。在非空表中的每

个数据元素都有一个确定的位置。a_i 是第 i 个元素，把 i 称为数据元素 a_i 在线性中的**位序**。

二、线性表的类型定义

1、抽象数据类型线性表的定义如下：

ADT List{

数据对象：D={a_i| a_i(-ElemSet,i=1,2,...,n,n>=0}

数据关系：R1={<a_{i-1},a_i>| a_{i-1},a_i(- D,i=2,...,n}

基本操作：

InitList(&L)

DestroyList(&L)

ClearList(&L)

ListEmpty(L)

ListLength(L)

GetElem(L,i,&e)

LocateElem(L,e,compare())

PriorElem(L,cur_e,&pre_e)

NextElem(L,cur_e,&next_e)

ListInsert(&L,i,e)

ListDelete(&L,i,&e)

ListTraverse(L,visit())

union(List &La,List &Lb)

}ADT List

2、部分操作的类 C 实现：

InitList(&L)

```
{L.elem=(ElemType *)malloc(LIST_INIT_SIZE*sizeof(ElemType));
if(!L.elem)exit(OVERFLOW);
L.length=0;
L.listsize=LIST_INIT_SIZE;
return OK;
```

```
}//InitList
GetElem(L,i,&e)
{*e=L.lem[i]
}//GetElem
ListInsert(List &L,int i,ElemType e)
{if(i<1||i>L.length+) return ERROR;
q=&(L.elem[i-1]);
for(p=&(L.elem[L.length-1]);p>=q;--p) *(p+1)=*p;
*q=e;
++L.length;
return OK;
}//ListInsert
void union(List &La,List &Lb)
{La_len=ListLength(La);Lb_len=ListLength(Lb);
for(i=1;i<=Lb_len;i++){
GetElem(Lb,i,e);
if(!LocateElem(La,e,equal))
ListInsert(La,++La_len,e);
}//union
void MergeList(List La,List Lb,List &Lc)
{InitList(Lc);
i=j=1;k=0;
La_len=ListLength(La);Lb_len=ListLength(Lb);
while((i<=La_len)&&(j<Lb_len)){
GetElem(La,i,ai);GetElem(Lb,j,bj);
if(ai<=bj){ListInsert(Lc,++k,ai);++i;}
else{ListInsert(Lc,++k,bj);++j;}
}
while(k<=La_len){
GetElem(La,i++,ai);ListInsert(Lc,++k,ai);
```

```
}
while(j<=Lb_len){
GetElem(Lb,j++,bj);ListInsert(Lc,++k,bj);
}
}//MergeList
```

3、部分操作的C语言实现,下面是程序运行的结果：

```
--------------------List Demo is running...-----------------
First is InsertList function.
name stuno age score
stu1 100001 80 1000
stu2 100002 80 1000
List A length now is 2.
name stuno age score
stu1 100001 80 1000
stu2 100002 80 1000
stu3 100003 80 1000
List A length now is 3.
name stuno age score
zmofun 100001 80 1000
bobjin 100002 80 1000
stu1 100001 80 1000
List B length now is 3.
Second is UnionList function.
Now union List A and List B.....
name stuno age score
stu1 100001 80 1000
stu2 100002 80 1000
stu3 100003 80 1000
zmofun 100001 80 1000
```

```
bobjin 100002 80 1000
List A length now is 5.
Welcome to visit http://zmofun.heha.net !
```

三、总结

线性表的定义

线性表的类型定义

回目录 上一课 下一课

第六课

本课主题： 线性表的顺序表示和实现

教学目的： 掌握线性表的顺序表示和实现方法

教学重点： 线性表的顺序表示和实现方法

教学难点： 线性表的顺序存储的实现方法

授课内容：

复习

1、存储结构

逻辑结构		“数据结构”定义中的“关系”指数据间的逻辑关系，故也称数据结构为逻辑结构。
存储结构		数据结构在计算机中的表示称为物理结构。又称存储结构。
	顺序存储结构	
	链式存储结构	

2、线性表的类型定义

一、线性表的顺序表示

用一组地址连续的存储单元依次存储线性表的数据元素。C 语言中的数组即采用顺序存储方式。

2000:0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
2000:0003	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2
2000:0005	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	3
2000:0007	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	4
2000:0009	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	5
2000:0011	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	6
2000:0013	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7
2000:0015	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	8
2000:0017	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	9
...																	
2000:1001																	
2000:1003																	

a[9]

假设线性表的每个元素需占用 *1* 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。则存在如下关系：

$$LOC(a_{i+1})=LOC(a_i)+1$$

$$LOC(a_i)=LOC(a_1)+(i-1)*1$$

式中 LOC(a_1)是线性表的第一个数据元素的存储位置，通常称做线性表的起始位置或基地址。常用 *b* 表示。

线性表的这种机内表示称做线性表的顺序存储结构或顺序映象。

称顺序存储结构的线性表为顺序表。顺序表的特点是以元素在计算机内物理位置相邻来表示线性表中数据元素之间的逻辑关系。

二、顺序存储结构的线性表类 C 语言表示：

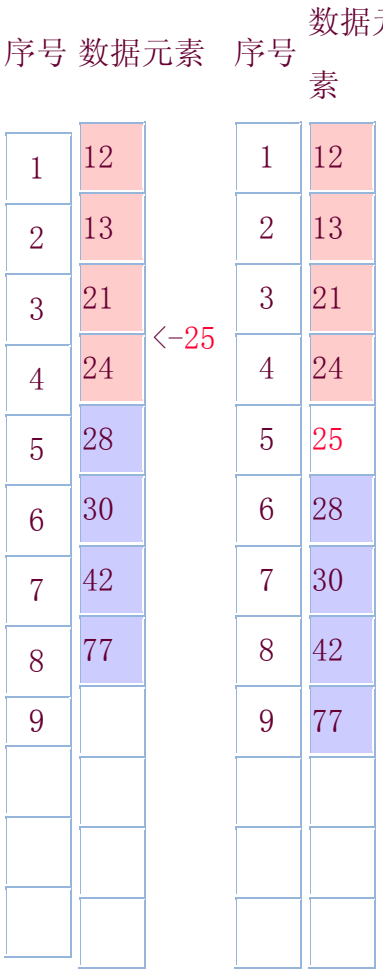
线性表的动态分配顺序存储结构

```
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
```

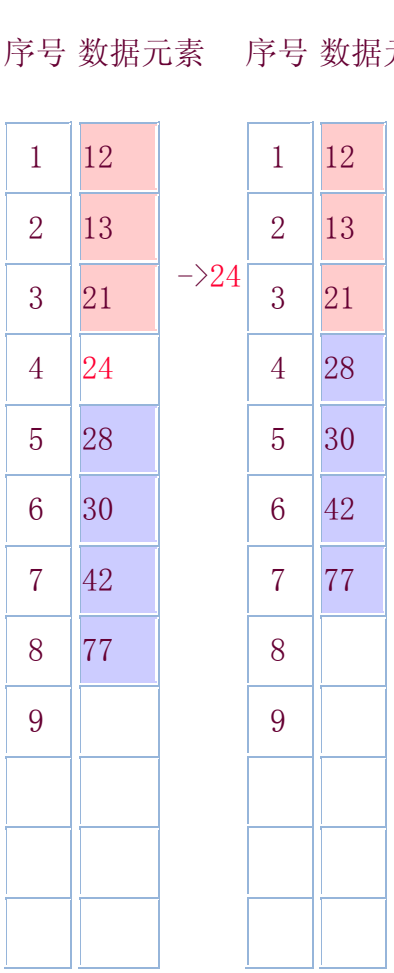
```
typedef struct{
ElemType *elem; //存储空间基址
int length; //当前长度
int listsize; //当前分配的存储容量以一数据元素存储长度为单位
}SqList;
```

三、顺序存储结构的线性表操作及 C 语言实现：

顺序表的插入与删除操作：



插入前 n=8;插入后 n=9;



删除前 n=8;删除后 n=7;

顺序表的插入算法

```
status ListInsert(List *L,int i,ElemType e) {
struct STU *p,*q;
if (i<1||i>L->length+1) return ERROR;
```

```
q=&(L->elem[i-1]);
for(p=&L->elem[L->length-1];p>=q;--p)
*(p+1)=*p;
*q=e;
++L->length;
return OK;
}/*ListInsert Before i */
```

顺序表的合并算法

```
void MergeList(List *La,List *Lb,List *Lc) {
ElemType *pa,*pb,*pc,*pa_last,*pb_last;
pa=La->elem;pb=Lb->elem;
Lc->listsize = Lc->length = La->length + Lb->length;
pc = Lc->elem =
(ElemType *)malloc(Lc->listsize * sizeof(ElemType));
if(!Lc->elem) exit(OVERFLOW);
pa_last = La->elem + La->length - 1;
pb_last = Lb->elem + Lb->length - 1;
while(pa<=pa_last && pb<=pb_last) {
if(Less_EqualList(pa,pb)) *pc++=*pa++;
else *pc++=*pb++;
}
while(pa<=pa_last) *pc++=*pa++;
while(pb<=pb_last) *pc++=*pb++;
}
```

顺序表的查找算法

```
int LocateElem(List *La,ElemType e,int type) {
int i;
switch (type) {
case EQUAL:
```

```
for(i=0;i<length;i++)
if(EqualList(&La->elem[i],&e))
return 1;
break;
default:
break;
}
return 0;
}
```

顺序表的联合算法

```
void UnionList(List *La, List *Lb) {
int La_len,Lb_len; int i; ElemType e;
La_len=ListLength(La); Lb_len=ListLength(Lb);
for(i=0;i<Lb_len;i++) {
GetElem(*Lb,i,&e);
if(!LocateElem(La,e,EQUAL))
ListInsert(La,++La_len,e);
}
}
```

三、C 语言源程序范例

四、总结

线性表的顺序表示

顺序表的插入算法

顺序表的合并算法

回目录 上一课 下一课

第七课

本课主题： 实验一 线性表的顺序存储实验

教学目的： 掌握顺序表的定义及操作的 C 语言实现方法

教学重点： 顺序表的操作的 C 语言实现方法

教学难点： 顺序表的操作的 C 语言实现方法

实验内容：

利用顺序表完成一个班级的一个学期的所有课程的管理：能够增加、删除、修改学生的成绩记录。

实验要求：

在上机前写出全部源程序。

回目录 上一课 下一课

第八课

本课主题： 线性表的链式表示与实现

教学目的： 掌握线性链表、单链表、静态链表的概念、表示及实现方法

教学重点： 线性链表之单链表的表示及实现方法。

教学难点： 线性链表的概念。

授课内容：

一、复习顺序表的定义。

二、线性链表的概念：

以链式结构存储的线性表称之为线性链表。

特点是该线性表中的数据元素可以用任意的存储单元来存储。线性表中逻辑相邻的两元素的存储空间可以是不连续的。为表示逻辑上的顺序关系，对表的每个数据元素除存储本身的信息之外，还需存储一个指示其直接衙继的信息。这两部分信息组成数据元素的存储映象，称为结点。

2000:1000	头指针 2000:1006	2000:1030	
2000:1010	a3	2000:1040	
2000:1020	a6	NULL	
2000:1030	a1	2000:1060	<-数据域+指针域
2000:1040	a4	2000:1050	
2000:1050	a5	2000:1020	
2000:1060	a2	2000:1010	
...	数据域	指针域	
2000:4000			

例:下图是若干抽屉，每个抽屉中放一个数据元素和一个指向后继元素的指针，一号抽屉中放线性表的第一个元素，它的下一个即第二个元素的位置标为 5，即放在第 5 个抽屉中，而第三个放在 2 号抽屉中。第三个元素即为最后一个，它的下一个元素的指针标为空，用 0 表示。

用线性链表表示线性表时,数据元素之间的逻辑关系是由结点中的指针指示的

二、线性链表的存储实现

```
struct LNODE{
ElemType data;
struct LNODE *next;
};
typedef struct LNODE LNode;
```

```
typedef struct LNODE * LinkList;
```

头指针与头结点的区别：

头指针只相当于结点的指针域，头结点即整个线性链表的第一个结点，它的数据域可以放数据元素，也可以放线性表的长度等附加信息，也可以不存储任何信息。

三、线性表的操作实现(类 C 语言)

1 初始化操作

```
Status Init_L(LinkList L){
if (L=(LinkList *)malloc(sizeof(LNode)))
{L->next=NULL;return 1;}
else return 0;
}
```

2 插入操作

```
Status ListInsert_L(LinkList &L,int i,ElemType e){
p=L,j=0;
while(p&&j<i-1){p=p->next;++j;}
if(!p||j>i-1) return ERROR;
s=(LinkList)malloc(sizeof(LNode));
s->data=e;s->next=p->next;
p->next=s;
return OK;
}//ListInsert_L
```

3 删除操作

```
Status ListDelete_L(LinkList &L,int i,ElemType &e){
p=L,j=0;
while(p&&j<i-1){p=p->next;++j;}
if(!p->next||j>i-1) return ERROR;
q=p->next;p->next=q->next;
e=q->data;free(q);
return OK;
}//ListDelete_L
```

4 取某序号元素的操作

```
Status GetElem_L(LinkList &L,int i,ElemType &e){
p=L->next,j=1;
while(p&&j<i){p=p->next;++j;}
if(!p||j>i) return ERROR;
e=p->data;
return OK;
}//GetElem_L
```

5 归并两个单链表的算法

```
void MergeList_L(LinkList &La,LinkList &Lb,LinkList &Lc){
//已知单链线性表 La 和 Lb 的元素按值非递减排列
//归并后得到新的单链线性表 Lc,元素也按值非递减排列
pa=La->next;pb=Lb->next;
Lc=pc=La;
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;}
}
pc->next=pa?pa:pb;
free(Lb);
}//MergeList_L
```

C 语言实现的例子。

四、总结

1、线性链表的概念。

2、线性链表的存储

3、线性链表的操作

回目录 上一课 下一课

第九课

本课主题： 循环链表与双向链表

教学目的： 掌握循环链表的概念，掌握双向链表的的表示与实现

教学重点： 双向链表的表示与实现

教学难点： 双向链表的存储表示

授课内容：

一、复习线性链表的存储结构

二、循环链表的存储结构

循环链表是加一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点。

循环链表的操作和线性链表基本一致，差别仅在于算法中的循环条件不是 p 或 p->next 是否为空，而是它们是否等于头指针。

三、双向链表的存储结构

提问：单向链表的缺点是什么？

提示：如何寻找结点的直接前趋。

双向链表可以克服单链表的单向性的缺点。

在双向链表的结点中有两个指针域，其一指向直接后继，另一指向直接前趋。

1、线性表的双向链表存储结构

```
typedef struct DulNode{
struct DulNode *prior;
ElemType data;
struct DulNode *next;
}DulNode,*DuLinkList;
```

对指向双向链表任一结点的指针 d，有下面的关系：

d->next->priou=d->priou->next=d

即：当前结点后继的前趋是自身，当前结点前趋的后继也是自身。

2、双向链表的删除操作

```
Status ListDelete_DuL(DuLinkList &L,int i,ElemType &e){
if(!(p=GetElemP_DuL(L,i)))
return ERROR;
e=p->data;
p->prior->next=p->next;
p->next->prior=p->pror;
free(p);
```

```
return OK;
}//ListDelete_DuL
```

3、双向链表的插入操作

```
Status ListInsert_DuL(DuLinkList &L,int i,ElemType &e){
if(!(p=GetElemP_DuL(L,i)))
return ERROR;
if(!(s=(DuLinkList)malloc(sizeof(DuLNode)))) return ERROR;
s->data=e;
s->prior=p->prior;
p->prior->next=s;
s->next=p;
p->prior=s;
return OK;
}//ListInsert_DuL
```

四、一个完整的带头结点的线性边表类型定义：

```
typedef struct LNode{
ElemType data;
struct LNode *next;
}*Link,*Position;

typedef struct{
Link head,tail;
int len;
```

```
}LinkList;


Status MakeNode(Link &p,ElemType e);
//分配由 p 指向的值为 e 的结点，并返回 OK；若分配失败，则返回 ERROR
void FreeNode(Link &p);
//释放 p 所指结点
Status InitLinst(LinkList &L);
//构造一个空的线性链表 L
Status DestroyLinst(LinkList &L);
//销毁线性链表 L，L 不再存在
Status ClearList(LinkList &L);
//将线性链表 L 重置为空表，并释放原链表的结点空间
Status InsFirst(Link h,Link s);
//已知 h 指向线性链表的头结点，将 s 所指结点插入在第一个结点之前
Status DelFirst(Link h,Link &q);
//已知 h 指向线性链表的头结点，删除链表中的第一个结点并以 q 返回
Status Append(LinkList &L,Link s);
//将指针 s 所指(彼此以指针相链)的一串结点链接在线性链表 L 的最后一个结点
//之后，并改变链表 L 的尾指针指向新的尾结点
Status Remove(LinkList &L,Link &q);
//删除线性链表 L 中的尾结点并以 q 返回，改变链表 L 的尾指针指向新的尾结点
Status InsBefore(LinkList &L,Link &p,Link s);
//已知 p 指向线性链表 L 中的一个结点，将 s 所指结点插入在 p 所指结点之前，
//并修改指针 p 指向新插入的结点
Status InsAfter(LinkList &L,Link &p,Link s);
//已知 p 指向线性链表 L 中的一个结点，将 s 所指结点插入在 p 所指结点之后，
//并修改指针 p 指向新插入的结点
Status SetCurElem(Link &p,ElemType e);
//已知 p 指向线性链表中的一个结点，用 e 更新 p 所指结点中数据元素的值
```

```
ElemType GetCurElem(Link p);
//已知 p 指向线性链表中的一个结点，返回 p 所指结点中数据元素的值
Status ListEmpty(LinkList L);
//若线性链表 L 为空表，则返回 TRUE,否则返回 FALSE
int ListLength(LinkList L);
//返回线性链表 L 中的元素个数
Position GetHead(LinkList L);
//返回线性链表 L 中头结点的位置
Position GetLast(LinkList L);
//返回线性链表 L 中最后一个结点的位置
Position PriorPos(LinkList L,Link p);
//已知 p 指向线性链表 L 中的一个结点，返回 p 所指结点的直接前趋的值
//若无前趋，返回 NULL
Position NextPos(LinkList L,Link p);
//已知 p 指向线性链表 L 中的一个结点，返回 p 所指结点的直接后继的值
//若无后继，返回 NULL
Status LocatePos(LinkList L,int i,Link &p);
//返回 p 指示线性链表 L 中第 i 个结点的位置并返回 OK,i 值不合法时返回 ERROR
Position LocateElem(LinkList L,ElemType e,
Status(*compare)(ElemType,ElemType));
//返回线性链表 L 中第 1 个与 e 满足函数 compare()判定关系的元素的位置，
//若下存在这样的元素，则返回 NULL
Status ListTraverse(LinkList L,Status(*visit)());
//依次对 L 的每个元素调用函数 visit()。一旦 visit()失败，则操作失败。
```

五、总结本课内容

循环链表的存储结构

双向链表的存储结构

回目录 上一课 下一课

第十课

本课主题： 栈的表示与实现

教学目的： 栈的数据类型定义、栈的顺序存储表示与实现

教学重点： 栈的顺序存储表示与实现方法

教学难点： 栈的定义

授课内容：

一、栈的定义

栈是限定仅在**表尾**进行插入或删除操作的线性表。

栈的表尾称为栈顶，表头称为栈底，不含元素的空表称为空栈。

栈的抽象数据类型定义：

ADT Stack{

数据对象:D={a_i|a_i(- ElemSet,i=1,2,...,n,n>=0}

数据关系:R1={<a_{i-1},a_i>|a_{i-1},a_i(- D,i=2,...,n}

基本操作:

InitStack(&S) 构造一个空栈 S

DestroyStack(&S) 栈 S 存在则栈 S 被销毁

ClearStack(&S) 栈 S 存在则清为空栈

StackEmpty(S) 栈 S 存在则返回 TRUE,否则 FALSE

StackLength(S) 栈 S 存在则返回 S 的元素个数,即栈的长度

GetTop(S,&e) 栈 S 存在且非空则返回 S 的栈顶元素

Push(&S,e) 栈 S 存在则插入元素 e 为新的栈顶元素

Pop(&S,&e) 栈 S 存在且非空则删除 S 的栈顶元素并用 e 返回其值

StackTraverse(S,visit())栈 S 存在且非空则从栈底到栈顶依次对 S 的每个数据元素调用函数 visit()一旦 visit()失败,则操作失败

}ADT Stack

二、栈的表示和实现

栈的存储方式：

1、顺序栈:利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素,同时附设指针 top 指示栈顶元素在顺序栈中的位置

2、链栈:利用链表实现

顺序栈的类 C 语言定义:

```
typedef struct{
SElemType *base;
SElemType *top; //设栈顶栈底两指针的目的是便于判断栈是否为空
int StackSize; //栈的当前可使用的最大容量.
}SqStack;
```

顺序栈的的模块说明:

```
struct STACK {
SElemType *base;
```

```
SElemType *top;
int stacksize;
};
typedef struct STACK Sqstack;
Status InitStack(SqStack &S);
Status DestroyStack(SqStack &S);
Status ClearStack(SqStack &S);
Status StackEmpty(SqStack S);
int StackLength(SqStack S);
Status GetTop(SqStack S,SElemType &e);
Status Push(SqStack &S,SElemType e);
Status Pop(SqStack &S,SElemType &e);
Status StackTraverse(SqStack S,Status (*visit)());

Status InitStack(SqStack &S) {
S.base=(SelemType *)malloc(STACK_INIT_SIZE *sizeof(ElemType));
if(!S.base)exit(OVERFLOW);
S.top=S.base;
S.stacksize=STACK_INI_SIZE;
return OK;
```

```
}//IniStack

Status DestroyStack(SqStack &S); {

}//DestroyStack

Status ClearStack(SqStack &S); {

S.top=S.base;

} //ClearStack

Status StackEmpty(SqStack S); {

if(S.top==S.base) return TRUE;

else return FALSE;

} //StackEmpty

int StackLength(SqStack S); {

int i; SElemType *p;

i=0;

p=S.top;

while(p!=S.base) {p++; i++; }

} //stackLength

Status GetTop(SqStack S,SElemType &e); {

if(S.top==S.base) return ERROR;

e=*(S.top-1);

return OK;
```

```
} //GetTop

Status Push(SqStack &S,SElemType e); {

if(S.top - s.base>=S.stacksize) {

S.base=(ElemType *) realloc(S.base,

(S.stacksize + STACKINCREMENT) * sizeof(ElemType));

if(!S.base)exit(OVERFLOW);

S.top=S.base+S.stacksize;

S.stacksize+=STACKINCREMENT;

}

*S.top++=e;

return OK;

} //Push

Status Pop(SqStack &S,SElemType &e); {

if(S.top==S.base)

return ERROR;

e=*--S.top;

return OK;

}//Pop

Status StackTraverse(SqStack S,Status (*visit)()); {

}//StackTraverse
```

以上伪代码的 C 语言源码

三、总结

栈的定义

栈的顺序存储实现

回目录 上一课 下一课

第十一课

本课主题： 栈的应用

教学目的： 掌握栈的应用方法,理解栈的重要作用

教学重点： 利用栈实现行编辑,利用栈实现表达式求值

教学难点： 利用栈实现表达式求值

授课内容：

一、栈应用之一：数制转换

将十进制数转换成其它进制的数有一种简单的方法：

例：十进制转换成八进制：$(66)_{10}=(102)_8$

66/8=8 余 2

8/8=1 余 0

1/8=0 余 1

结果为余数的逆序:102 。先求得的余数在写出结果时最后写出，最后求出的余数最先写出，符合栈的先入后出性质，故可用栈来实现数制转换：

```
void conversion() {
pSqStack S;
SElemType e;
int n;
InitStack(&S);
printf("Input a number to convert to OCT:\n");
scanf("%d",&n);
```

```
if(n<0)
{ printf("\nThe number must be over 0.");
return;}
if(!n) Push(S,0);
while(n){
Push(S,n%8);
n=n/8; }
printf("the result is: ");
while(!StackEmpty(*S)){
Pop(S,&e); printf("%d",e);}
}
```

请看：数制转换的 C 源程序

二、栈应用之二：行编辑

一个简单的行编辑程序的功能是：接受用户从终端输入的程序或数据，并存入用户的数据区。允许用户输入出错时可以及时更正。可以约定＃为退格符，以表示前一个字符无效，@为退行符，表示当前行所有字符均无效。

例：在终端上用户输入为

whli##ilr#e(s#*s) 应为

while(*s)

```
void LineEdit() {
pSqStack S,T; char str[1000];
int strlen=0; char e; char ch;
InitStack(&S);
InitStack(&T);
ch=getchar();
while(ch!=EOFILE) {
while(ch!=EOFILE&&ch!='\n') {
switch(ch){
case '#': Pop(S,&ch); break;
```

```
case '@': ClearStack(S); break;
default: Push(S,ch); break; }
ch=getchar();
}
if(ch=='\n') Push(S,ch);
while(!StackEmpty(*S)) { Pop(S,&e); Push(T,e); }
while(!StackEmpty(*T)) { Pop(T,&e); str[strlen++]=e; }
if(ch!=EOFILE) ch=getchar();
}
str[strlen]='\0';
printf("\n%s",str);
DestroyStack(S);
DestroyStack(T);
}
```

请看:行编辑的 C 源程序

三、栈应用之三：表达式求值

一个程序设计语言应该允许设计者根据需要用表达式描述计算过程，编译器则应该能分析表达式并计算出结果。表达式的要素是运算符、操作数、界定符、算符优先级关系。例：1+2*3 有+,*两个运算符，*的优先级高,1,2,3 是操作数。 界定符有括号和表达式结束符等。

算法基本思想：

1 首先置操作数栈为空栈，表达式起始符＃为运算符栈的栈底元素；

2 依次讲稿表达式中每个字符，若是操作数则进 OPND 栈，若是运算符，则和 OPTR 栈的栈顶运算符比较优先权后作相应操作，直至整个表达式求值完毕。

```
char EvaluateExpression() {
SqStack *OPND,*OPTR;
char c,x,theta; char a,b;
InitStack(&OPTR); Push(OPTR,'#');
```

```
InitStack(&OPND);
c=getchar();
while(c!='#'||GetTop(*OPTR)!='#') {
if(!In(c,OP)) {Push(OPND,c);c=getchar();}
else
switch(Precede(GetTop(*OPTR),c)) {
case '<': Push(OPTR,c); c=getchar(); break;
case '=': Pop(OPTR,&x); c=getchar(); break;
case '>': Pop(OPTR,&theta);
Pop(OPND,&b); Pop(OPND,&a);
Push(OPND,Operate(a,theta,b));
break;
}
}
c=GetTop(*OPND);
DestroyStack(OPTR);
DestroyStack(OPND);
return c;
}
```

请看:表达式求值的 C 源程序

四、总结

栈的先进后出、后进先出的特性。

回目录 上一课 下一课

第十二课

本课主题： 实验二 循环链表实验

教学目的： 掌握单向链表的实现方法

教学重点： 单向链表的存储表示及操作

教学难点： 单向链表的操作实现

授课内容：

一、单向链表的存储表示

C 源程序

二、单向链表的基本操作

C 源程序

回目录 上一课 下一课

第十三课

本课主题： 队列

教学目的： 掌握队列的类型定义,掌握链队列的表示与实现方法

教学重点： 链队列的表示与实现

教学难点： 链队列的表示与实现

授课内容：

一、队列的定义:

队列是一种先进先出的线性表。它只允许在表的一端进行插入，而在另一端删除元素。象日常生活中的排队，最早入队的最早离开。

在队列中，允许插入的的一端叫队尾，允许删除的一端则称为队头。

抽象数据类型队列:

ADT Queue{

数据对象: D={ai| ai(-ElemSet,i=1,2,...,n,n>=0}

数据关系: R1={<ai-1,ai> | ai-1,ai(- D,i=2,...,n}

基本操作:

InitQueue(&Q) 构造一个空队列 Q

Destroyqueue(&Q) 队列 Q 存在则销毁 Q

ClearQueue(&Q) 队列 Q 存在则将 Q 清为空队列

QueueEmpty(Q) 队列 Q 存在,若 Q 为空队列则返回 TRUE,否则返回 FALSE

QueueLenght(Q) 队列 Q 存在,返回 Q 的元素个数,即队列的长度

GetHead(Q,&e) Q 为非空队列,用 e 返回 Q 的队头元素

EnQueue(&Q,e) 队列 Q 存在,插入元素 e 为 Q 的队尾元素

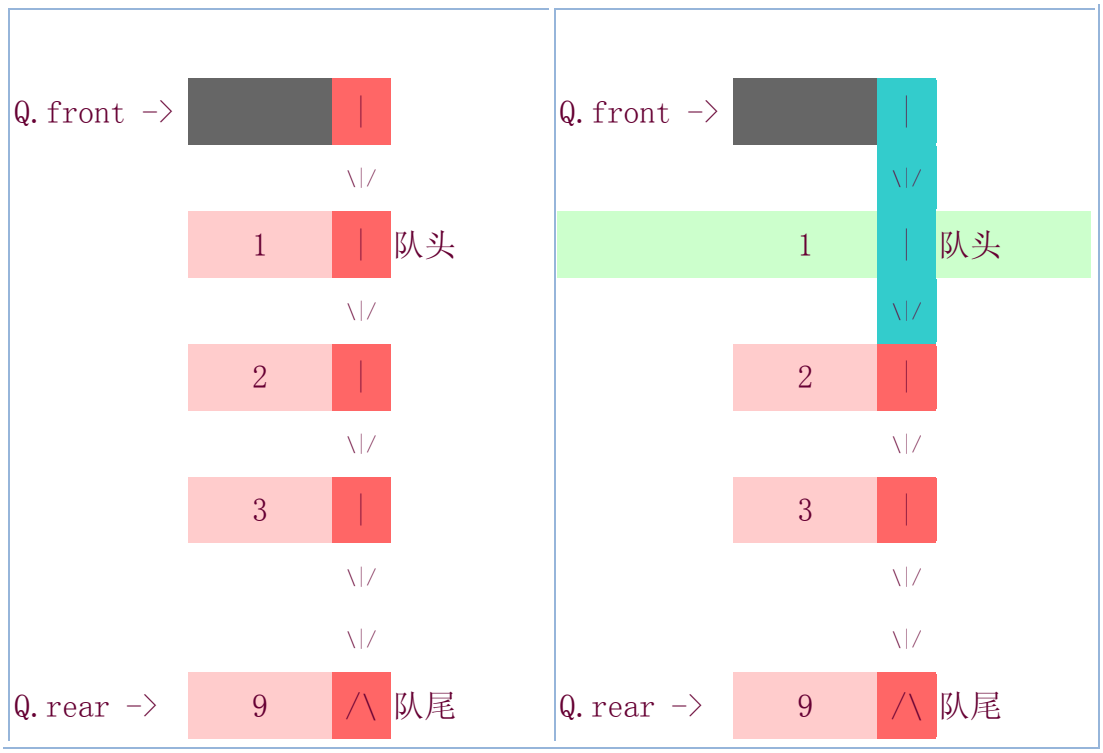
DeQueue(&Q,&e) Q 为非空队列,删除 Q 的队头元素,并用 e 返回其值

QueueTraverse(Q,vivsit()) Q 存在且非空,从队头到队尾,依次对 Q 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失败

}ADT Queue

二、链队列—队列的链式表示和实现

用链表表示的队列简称为链队列。一个链队列显然需要两个分别指示队头和队尾的指针。



链队列表示和实现：

```
//存储表示
typedef struct QNode{
QElemType data;
struct QNode *next;
}QNode,*QueuePtr;
typedef struct{
```

```
QueuePtr front;
QueuePtr rear;
}LinkQueue;
//操作说明
Status InitQueue(LinkQueue &Q)
//构造一个空队列 Q
Status Destroyqueue(LinkQueue &Q)
//队列 Q 存在则销毁 Q
Status ClearQueue(LinkQueue &Q)
//队列 Q 存在则将 Q 清为空队列
Status QueueEmpty(LinkQueue Q)
// 队列 Q 存在,若 Q 为空队列则返回 TRUE,否则返回 FALSE
Status QueueLenght(LinkQueue Q)
// 队列 Q 存在,返回 Q 的元素个数,即队列的长度
Status GetHead(LinkQueue Q,QElemType &e)
//Q 为非空队列,用 e 返回 Q 的队头元素
Status EnQueue(LinkQueue &Q,QElemType e)
//队列 Q 存在,插入元素 e 为 Q 的队尾元素
Status DeQueue(LinkQueue &Q,QElemType &e)
//Q 为非空队列,删除 Q 的队头元素,并用 e 返回其值
Status QueueTraverse(LinkQueue Q,QElemType vivsit())
//Q 存在且非空,从队头到队尾,依次对 Q 的每个数据元素调用函数 visit()。一
旦 visit()失败，则操作失败
//操作的实现
Status InitQueue(LinkQueue &Q) {
//构造一个空队列 Q
Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
if(!Q.front)exit(OVERFLOW);
Q.front->next=NULL;
return OK;}
```

```
Status Destroyqueue(LinkQueue &Q) {
//队列Q存在则销毁Q
while(Q.front){
Q.rear=Q.front->next;
free(Q.front);
Q.front=Q.rear;
}
return OK;}
Status EnQueue(LinkQueue &Q,QElemType e) {
//队列Q存在,插入元素e为Q的队尾元素
p=(QueuePtr)malloc(sizeof(QNode));
if(!p) exit(OVERFLOW);
p->data=e;p->next=NULL;
Q.rear->next=p;
Q.rear=p;
return OK;}
Status DeQueue(LinkQueue &Q,QElemType &e) {
//Q为非空队列,删除Q的队头元素,并用e返回其值
if(Q.front==Q.rear)return ERROR;
p=Q.front->next;
e=p->data;
Q.front->next=p->next;
if(Q.rear==p)Q.rear=Q.front;
free(p);
return OK;}
```

三、总结

链队列的存储表示

链队列的操作及实现

回目录 上一课 下一课

第十四课

本课主题： 串的定义

教学目的： 掌握串的定义及作用

教学重点： 串的类型定义

教学难点： 串的类型定义

授课内容：

一、串定义

串（或字符串），是由零个或多个字符组成的有限序列。一般记为：

s='$a_1a_2...a_n$'(n>=0)

其中 s 是串的名，用单引号括起来的字符序列是串的值；串中字符的数目 n 称为串的长度。零个字符的串称为空串，它的长度为零。

串中任意个连续的字符组成的子序列称为该串的子串。包含子串的串相应地称为主串。通常称字符在序列中的称为该字符在串中的位置。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

例：a='BEI',b='JING',c='BEIJING',d='BEI JING'

串长分别为 3,4,7,8,且 a,b 都是 c,d 的子串。

称两个串是相等的，当且仅当这两个串的值相等。

二、串的抽象数据类型的定义：

ADT String{

数据对象：D={ai|ai(-CharacterSet,i=1,2,...,n,n>=0}

数据关系：R1={<ai-1,ai>|ai-1,ai(-D,i=2,...,n}

基本操作：

StrAssign(&T,chars)

chars 是字符常量。生成一个其值等于 chars 的串 T。

StrCopy(&T,S)

串 S 存在则由串 S 复制得串 T

StrEmpty(S)

串 S 存在则若 S 为空串,返回真否则返回假

StrCompare(S,T)

串 S 和 T 存在,若 S>T,则返回值大于 0,若 S=T,则返回值=0,若 S<T,则返回值<0

StrLength(S)

串 S 存在返回 S 的元素个数称为串的长度.

ClearString(&S)

串 S 存在将 S 清为空串

Concat(&T,S1,S2)

串 S1 和 S2 存在用 T 返回由 S1 和 S2 联接而成的新串

SubString(&Sub,S,pos,len)

串 S 存在,1<=pos<=StrLength(S)且 0<=len<=StrLength(S)-pos+1

Index(S,T,pos)

串 S 和 T 存在,T 是非空,1<=pos<=StrLength(S),若主串 S 中存在和串 T 值相同的子串,则返回它在主串 S 中第 pos 个字符之后第一次出现的位置,否则函数值为 0

Replace(&S,T,V)

串 S,T 和 V 存在,T 是非空串,用 V 替换主串 S 中出现的所有与 T 相等的不重叠的子串

StrInsert(&S,pos,T)

串 S 和 T 存在,1<=pos<=StrLength(S)+1,在串 S 的第 pos 个字符之前插入串 T

StrDelete(&S,pos,len)

串 S 存在,1<=pos<=StrLength(S)-len+1 从串中删除第 pos 个字符起长度为 len 的子串

DestroyString(&S)

串 S 存在,则串 S 被销毁

}ADT String

三、串操作应用举例:

1 文字处理中常用的:串的查找(比较,定位)与替换

在 TC 集成环境中可用^QF 快速查找变量 在 WORD 中可用搜索与替换批量改变文本

2 串的截断与连接

可用求子串及串连接的方法进行文字处理

四、总结

找出几个自己亲自做过的串操作例子。

回目录 上一课 下一课

第十五课

本课主题： 串的表示和实现

教学目的： 掌握串的几种实现方法

教学重点： 定长顺序存储表示法堆分配存储表示法

教学难点： 堆分配存储表示法

授课内容：

一、复习串的定义

串的定义

二、定长顺序存储表示

类似于线性表的顺序存储结构,用一组地址连续的存储单元存储串值的字符序列.

```
#define MAXSTRLEN 255
typedef unsigned char SString[MAXSTRLEN+1] //0 号单元存放串长
```

串的实际长度可在这予定义长度的范围内随意,超过予定义长度的串值则被舍去

串长可用下标为 0 的数组元素存储,也可在串值后设特殊标记

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	...	a[n]
3	a	b	c				pascal
a	b	c	\0				c

1 串联接的实现 Concat(&T,S1,S2)

假设 S1,S2 和 T 都是 SString 型的串变量,且串 T 是由串 S1 联结串 S2 得到的,即串 T 的值的前一段和串 S1 的值相等,串 T 的值的后一段和串 S2 的值相等,则只要进行相应的"串值复制"操作即可,对超长部分实施"截断"操作

以下是串联接可能出现的三种情况:

S1	S2	T
4	2	6
a	d	a
b	e	b
c		c
d		d
		e
		f

S1,S2 串长和小于最大值

S1	S2	T
6	6	8
a	g	a
b	h	b
c	i	c
d	j	d
e	k	e
f	l	f
		g
		h

S1,S2 串长和超过最大串长

S1	S2	T
8	2	8
a	i	a
b	j	b
c		c
d		d
e		e
f		f
g		g
h		h

S1 串长已等于最大串长

算法描述如下:

```
Status Concat(SString &T,SString S1,SString S2){
if(S1[0]+S2[0]<=MAXSTRLEN){
T[1..S1[0]]=S1[1..S1[0]];
T[S1[0]+1..S1[0]+S2[0]]=S2[1..S2[0]];
T[0]=S1[0]+S2[0]uncut=TRUE;
}
else if(S1[0]<MAXSTRSIZE){
T[1..S1[0]]=S1[1..S1[0]];
T[S1[0]+1..MAXSTRLEN]=S2[1..MAXSTRLEN-S1[0]];
T[0]=MAXSTRLEN;uncut=FALSE;
}
else{
```

```
T[0..MAXSTRLEN]=S1[0..MAXSTRLEN];
uncut=FALSE;
}
return uncut;
}
```

三、堆分配存储表示

这种存储表示的特点是,仍以一组地址连续的存储单元存放串值字符序列,但它们的存储空间是在程序执行过程中动态分配而得

在 C 语言中,存在一个称之为堆的自由存储区,并由 C 语言的动态分配函数 malloc()和 free()来管理.利用函数 malloc()为每个新产生的串分配一块实际串长所需存储空间,为处理方便,约定串长也作为存储结构的一部分

```
typedef struct{
char *ch;//若是非空串,则按串长分配存储区,否则 ch 为 NULL
int length; //串长度
}HString
Status StrInsert(HString &S,int pos,HString T){
if(pox<1||pos>S.length+1) return ERROR;
if(T.length){
if(!(S.ch=(char *)realloc(S.ch,(S.length+T.length)*sizeof(char))))
exit(OVERFLOW);
for(i=S.length-1;i>=pos-1;--i)
S.ch[i+T.length]=S.ch[i];
S.ch[pos-1..pos+T.lenght-2]=T.ch[0..T.length-1];
S.length+=T.length;
}
return OK;
}
```

四、总结

思考两种存储表示方法的优缺点

回目录 上一课 下一课

第十六课

本课主题： 串操作应用举例

教学目的： 掌握文本编辑的基本原理及方法

教学重点： 简单文本编辑

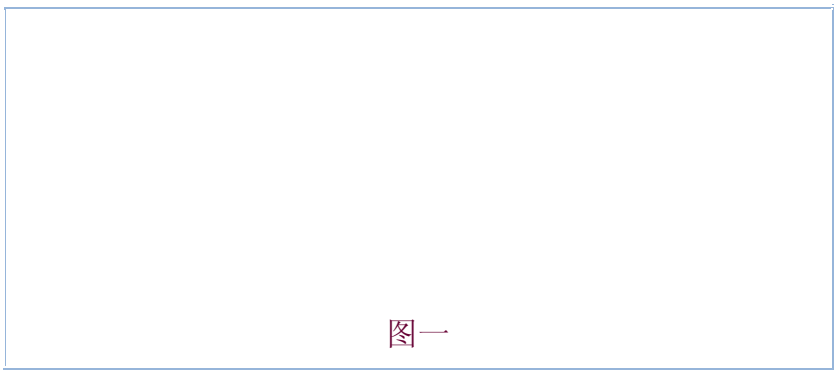
教学难点： 串的存储管理

授课内容：

一、复习串的堆分配存储表示

堆分配存储表示

二、文本编辑基本原理



图一

文本编辑可以用于源程序的输入和修改（如图一），也可用于报刊和书籍的编辑排版以及办公室的公文书信的起草和润色（如图二）。

图二

可用于文本编辑的程序很多，功能强弱差别很大，但基本操作是一致的：都包括串的查找，插入和删除等基本操作。

对用户来讲，一个文本（文件）可以包括若干页，每页包括若干行，每行包括若干文字。

对文本编辑程序来讲，可把整个文本看成一个长字符串，称文本串，页是文本串的子串，行又是页的子串。为简化程序复杂程度，可简单地把文本分成若干行。

例：下面的一段源程序可以看成一个文本串，

```
main(){
float a,b,max;
scanf("%f,%f",&a,&b);
if (a>b) max=a;
else max=b;
};
```

这个文本串在内存中的存储映像可为：

m	a	i	n	(	)	{	\n			f	l	o	a	t		a	,	b	,
m	a	x	;	\n			s	c	a	n	f	(	"	%	f	,	%	f	"
,	&	a	,	&	b	)	;	\n			i	f		a	>	b			m
a	x	=	a	;	\n			e	l	s	e			m	a	x	=	b	;
\n	}	\n																	

在编辑时，为指示当前编辑位置，程序中要设立页指针、行指针、字符指针，分别指示当前页，当前行，当前字符。因此程序中要设立页表、行表便于查找。

三、简单行编辑程序例

源程序

回目录 上一课 下一课

第十七课

本课主题： 实验三：栈的表示与实现及栈的应用

教学目的： 掌握栈的存储表示方式和栈基本操作的实现方法

教学重点： 栈的基本操作实现方法，栈的应用

教学难点： 栈的存储表示

实验内容：

一、栈的实现

实现栈的顺序存储。

栈实现示例

二、栈的应用

１、利用栈实现数制转换　２、利用栈实现单行编辑

以上任选一题。

数制转换示例

单行编辑示例

这里是实现栈的头文件

回目录 上一课 下一课

第十八课

本课主题： 数组的顺序表示与实现

教学目的： 掌握数组的定义,数组的顺序表示方法

教学重点： 数组的定义,数组的顺序表示方法

教学难点： 数组的顺序表示方法

授课内容：

一、数组的定义

几乎所有的程序设计语言都把数组类型设定为固有类型。

以抽象数据类型的形式讨论数组的定义和实现,可以让我们加深对数组类型的理解。

数组的定义：

ADT Array{

数据对象:j_i=0,...,b_i-1,i=1,2,...,n;

D={$a_{j1j2...jn}$|n(>0)称为数组的维数,b_i是数组第 i 维的长度,j_i是数组元素的第 i

维下标,$a_{j1j2...jn}$ (-ElemSet}

数据关系:R={R_1,R_2,...R_n|

Ri={<$a_{j1...ji...jn}$,$a_{j1...ji+1...jn}$>|

0<=j_k<=b_{k-1},1<=k<=n 且 k<>i,

0<=j_i<=b_{i-2},$a_{j1...ji...jn}$,

$a_{j1...ji+1...jn}$(-D,i=2,...n}

基本操作:

InitArray(&A,n,bound1,...,boundn)

若维数和各维长度合法,则构造相应的数组 A,并返回 OK.

DestroyArray(&A)

操作结果:销毁数组 A.

Value(A,&e,index1,...,indexn)

初始条件:A 是 n 维数组,e 为元素变量,随后是 n 个下标值.

操作结果:若各下标不超界,则 e 赋值为所指定的 A 的元素值,并返回 OK.

Assign(&A,e,index1,...,indexn)

初始条件:A 是 n 维数组,e 为元素变量,随后是 n 个下标值.

操作结果:若下标不超界,则将 e 的值赋给 所指定的 A 的元素,并返回 OK.

}ADT Array

列向量的一维数组:

a_{00}	a_{01}	a_{02}	...	$a_{0,n-1}$
a_{10}	a_{11}	a_{12}	...	$a_{1,n-1}$
...	...	...	...	...
$a_{m-1,0}$	$a_{m-1,1}$	$a_{m-1,2}$	...	$a_{m-1,n-1}$

行向量的一维数组:

把二维数组中的每一行看成是一个数据元素,这些数据元素组成了一个一维数组 A.

A_0	a_{00}	a_{01}	a_{02}	…	$a_{0,n-1}$
A_1	a_{10}	a_{11}	a_{12}	…	$a_{1,n-1}$
…	…	…	…	…	…
A_m	$a_{m-1,0}$	$a_{m-1,1}$	$a_{m-1,2}$	…	$a_{m-1,n-1}$

二、数组的顺序表示和实现

以行序为主序的存储方式:

a_{00}	a_{01}	a_{02}	…	$a_{0,n-1}$	a_{10}	a_{11}	a_{12}	…	$a_{1,n-1}$	…	$a_{m-1,0}$	$a_{m-1,1}$	$a_{m-1,2}$	…	$a_{m-1,n-1}$

数组的顺序存储表示和实现:

```
#include<stdarg.h>
#define MAX_ARRAY_DIM 8
typedef struct {
ElemType *base;
int dim;
int *bounds;
int *constants;
}Array;
Status InitArray(Array &A,int dim,...);
Status DestroyArray(Array &A);
Status Value(Array A,ElemType &e,...);
Status Assign(Array &A,ElemType e,...);
```

基本操作的算法描述:

```
Status InitArray(Array &A,int dim,...){
if(dim<1||dim>MAX_ARRAY_DIM) return ERROR;
A.dim=dim;
```

```
A.bounds=(int *)malloc(dim *sizeof(int));
if(!A.bounds) exit(OVERFLOW);
elemtotal=1;
va_start(ap,dim);
for(i=1;i<dim;++i){
A.bounds[i]=va_arg(ap,int);
if(A.bounds[i]<0) return UNDERFLOW;
elemtotal*=A.bounds[i];
}
va_end(ap);
A.base=(ElemType *)malloc(elemtotal*sizeof(ElemType));
if(!A.base) exit(OVERFLOW);
A.constants=(int *)malloc(dim*sizeof(int));
if(!A.constants) exit(OVERFLOW);
A.constants[dim-1]=1;
for(i=dim-2;i>=0;--i)
A.constants[i]=A.bounds[i+1]*A.constants[i+1];
return OK;
}
Status DestoyArray(Array &A){
if(!A.base) return ERROR;
free(A.base); A.base=NULL;
if !(A.bounds) return ERROR;
free(A.bounds); A.bounds=NULL;
if!(A.constatns) return ERROR;
free(A.constants); A.constants=NULL;
return OK;
}
Status Locate(Array A,va_list ap,int &off){
off=0;
```

```
for(i=0;i<A.dim;++i){
ind=va_arg(ap,int);
if(ind<0||ind>=A.bounds[i]) return OVERFLOW;
off+=A.constants[i]*ind;
}
return OK;
}
Status Value(Array A,ElemType &e,...){
va_start(ap,e);
if((result=Locate(A,ap,off))<=0 return result;
e=*(A.base+off);
return OK;
}
Status Assign(Array &A,ElemType e,...){
va_start(ap,e);
if((result=Locate(A,ap,off))<=0) return result;
*(A.base+off)=e;
return OK;
}
```

三、小结

数组的存储方式。

数组的基本操作种类。

回目录 上一课 下一课

第十九课

本课主题： 实验四 串的实现实验

教学目的： 掌握 PASCAL 串类型的实现方法

教学重点： 串的操作

教学难点： 串的联接操作

授课内容：

一、PASCAL 串类型的存储表示:

```
#define MAXSTRLEN 255

typedef char SString[MAXSTRLEN+1];
```

二、串的操作：

１、串的联接

```
mystrcat(SString s1,SString s2,SString t);
```

２、求子串

```
mysubstr(SString t,int pos,int len,SString sub);
```

３、子串定位

```
mystrindex(SString t,SString sub,int *index);
```

回目录 上一课 下一课

第二十课

本课主题： 广义表

教学目的： 广义表的定义及存储结构

教学重点： 广义表的操作及意义

教学难点： 广义表存储结构

授课内容：

一、广义表的定义

广义表是线性表的推广，其表中的元素可以是另一个广义表,或其自身.

广义表的定义:

ADT GList{

数据对象:D={i=1,2,...,n>=0;ei(-AtomSet 或 ei(-GList,

AtomSet 为某个数据对象}

数据关系:R1={<ei-1,ei>|ei-1,ei(-D,2=<i<=n}

基本操作:

InitGlist(&L);

操作结果:创建空的广义表 L

CreateGList(&L,S);

初始条件:S 是广义表的书写形式串

操作结果:由 S 创建广义表 L

DestroyGlist(&L);

初始条件:广义表 L 存在

操作结果:销毁广义表 L

CopyGlist(&T,L);

初始条件:广义表 L 存在

操作结果:由广义表 L 复制得到广义表 T

GListLength(L);

初始条件:广义表 L 存在

操作结果:求广义表 L 的长度,即元素个数

GListDepth(L);

初始条件:广义表 L 存在

操作结果:求广义表 L 的深度

GlistEmpty(L);

初始条件:广义表 L 存在

操作结果:判断广义表 L 是否为空

GetHead(L);

初始条件:广义表 L 存在

操作结果:取广义表 L 的头

GetTail(L);

初始条件:广义表 L 存在

操作结果:取广义表 L 的尾

InsertFirst_GL(&L,e);

初始条件:广义表 L 存在

操作结果:插入元素 e 作为广义表 L 的第一元素

DeleteFirst_GL(&L,&e);

初始条件:广义表 L 存在

操作结果:删除广义表 L 的第一元素,并用 e 返回其值

Traverse_GL(L,Visit());

初始条件:广义表 L 存在

操作结果:遍历广义表 L,用函数 Visit 处理每个元素

}ADT GList

广义表一般记作:LS=(a1,a2,...,an)

其中 LS 是广义表的名称,n 是它的长度,ai 可以是单个元素也可是广义表,分别称为原子和子表,当广义表非空时,称第一个元素 a1 为 LS 的表头称其余元素组成的广义表为表尾.

二、广义表的存储结构

广义表的头尾链表存储表示

typedef emnu{ATOM,LIST} ElemTag;

typedef struct GLNode{

ElemTag tag;

union{

AtomType atom;

struct{struct GLNode *hp,*tp;}ptr;

}

}

有 A、B、C、D、E 五个广义表的描述如下：

A=() A 是一个空表,它的长度为零

B=(e) 列表 B 只有一个原子 e,B 的长度为 1.

C=(a,(b,c,d)) 列表 C 的长度为 2,两个元素分别为原子 a 和子表(b,c,d)

D=(A,B,C) 列表 D 的长度为 3,三个元素都是列表,显然,将子表的值代入后,则有 D=((),(e),(a,(b,c,d)))

E=(a,E) 这是一个递归的表,它的长度为 2,E 相当于一个无限的列表 E=(a,(a,(a,...)))

上述五个广义表用以上的存储结构的存储映像如下:

回目录 上一课 下一课

第二十一课

本课主题： 树、二叉树定义及术语

教学目的： 掌握树、二叉树的基本概念和术语，二叉树的性质

教学重点： 二叉树的定义、二叉树的性质

教学难点： 二叉树的性质

授课内容：

一、树的定义：

树是 n(n>=0)个结点的有限集。在任意一棵非空树中：

(1)有且仅有一个特定的称为根的结点；

(2)当 n>1 时,其余结点可分为 m(m>0)个互不相交的有限集 T1,T2,...Tm,其中每一个集合本身又是一棵树,并且称为根的子树.

二、树的基本概念：

树的结点包含一个数据元素及若干指向其子树的分支。

结点拥有的子树数称为结点的度。

度为 0 的结点称为**叶子**或**终端结点**。

度不为 0 的结点称为**非终端结点**或**分支结点**。

树的度是树内各结点的度的最大值。

结点的子树的根称为该结点的**孩子**，相应地，该结点称为孩子的**双亲**。

同一个双亲的孩子之间互称**兄弟**。

结点的**祖先**是从根到该结点所经分支上的所有结点。

以某结点为根的子树中的任一结点都称为该结点的**子孙**。

结点的层次从根开始定义起，根为第一层，根的孩子为第二层。其双亲在同一层的结点互为堂兄弟。树中结点的最大层次称为树的深度，或高度。

如果将树中结点的各子树看成从左至右是有次序的，则称该树为有序树，否则称为无序树。

森林是 m(m>=0)棵互不相交的树的集合。

三、二叉树的定义

二叉树是另一种树型结构，它的特点是每个结点至多只有二棵子树（即二叉树中不存在度大于 2 的结点),并且,二叉树的子树有左右之分,其次序不能任意颠倒。一棵深度为 k 且有 2(k)-1 个结点的二叉树称为满二叉树，如图(a)，按图示给每个结点编号，如果有深度为 k 的，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时,称之为完全二叉树。

二叉树的定义如下：

ADT BinaryTree{

数据对象 D:D 是具有相同特性的数据元素的集合。

数据关系 R:

基本操作 P:

InitBiTree(&T);

DestroyBiTree(&T);

CreateBiTree(&T,definition);

ClearBiTree(&T);

BiTreeEmpty(T);

BiTreeDepth(T);

Root(T);

Value(T,e);

Assign(T,&e,value);

Parent(T,e);

LeftChild(T,e);

RightChild(T,e);

LeftSibling(T,e);

RightSibling(T,e);

InsertChild(T,p,LR,c);

DeleteChild(T,p,LR);

PreOrderTraverse(T,visit());

InOrderTraverse(T,visit());

PostOrderTraverse(T,visit());

LevelOrderTraverse(T,Visit());

}ADT BinaryTree

三、二叉树的性质

性质 1：	在二叉树的第 i 层上至多有 2 的 i-1 次方个结点 (i>=1)。	
性质 2：	深度为 k 的二叉树至多有 2 的 k 次方减 1 个结点	

	(k>=1)。	
性质 3：	对任何一棵二叉树 T，如果其终端结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。	
性质 4：	具有 n 个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor+1$	
性质 5：	如果对一棵有 n 个结点的完全二叉树的结点按层序编号，则对任一结点 i(1=<i=<n)有： (1)如果 i=1，则结点 i 是二叉树的根，无双亲；如果 i>1，则双亲 PARENT(i)是结点 i/2 (2)如果 2i>n，则结点 i 无左孩子(结点 i 为叶子结点)；否则其左孩子 LCHILD(i)是结点 2i (3)如果 2i+1>n，则结点 i 无右孩子；否则其右孩子 RCHILD(i)是结点 2i+1	

四、总结

回目录 上一课 下一课

第二十二课

本课主题： 实验五 数组实验

教学目的： 掌握二维数组的实现方法

教学重点： 二维数组的存储表示,二维数组的基本操作

教学难点： 二维数组的基本操作

授课内容：

数组的顺序存储表示和实现:

```
#include<stdarg.h>

#define MAX_ARRAY_DIM 8
```

```
typedef struct {
ElemType *base;
int dim;
int *bounds;
int *constants;
}Array;
Status InitArray(Array &A,int dim,...);
Status DestroyArray(Array &A);
Status Value(Array A,ElemType &e,...);
Status Assign(Array &A,ElemType e,...);
```

基本操作的算法描述:

```
Status InitArray(Array &A,int dim,...){
if(dim<1||dim>MAX_ARRAY_DIM) return ERROR;
A.dim=dim;
A.bounds=(int *)malloc(dim *sizeof(int));
if(!A.bounds) exit(OVERFLOW);
elemtotal=1;
va_start(ap,dim);
for(i=1;i<dim;++i){
A.bounds[i]=va_arg(ap,int);
```

```
if(A.bounds[i]<0) return UNDERFLOW;
elemtotal*=A.bounds[i];
}
va_end(ap);
A.base=(ElemType *)malloc(elemtotal*sizeof(ElemType));
if(!A.base) exit(OVERFLOW);
A.constants=(int *)malloc(dim*sizeof(int));
if(!A.constants) exit(OVERFLOW);
A.constants[dim-1]=1;
for(i=dim-2;i>=0;--i)
A.constants[i]=A.bounds[i+1]*A.constants[i+1];
return OK;
}
Status DestoyArray(Array &A){
if(!A.base) return ERROR;
free(A.base); A.base=NULL;
if !(A.bounds) return ERROR;
free(A.bounds); A.bounds=NULL;
if!(A.constatns) return ERROR;
free(A.constants); A.constants=NULL;
```

```
return OK;
}
Status Locate(Array A,va_list ap,int &off){
off=0;
for(i=0;i<A.dim;++i){
ind=va_arg(ap,int);
if(ind<0||ind>=A.bounds[i]) return OVERFLOW;
off+=A.constants[i]*ind;
}
return OK;
}
Status Value(Array A,ElemType &e,...){
va_start(ap,e);
if((result=Locate(A,ap,off))<=0 return result;
e=*(A.base+off);
return OK;
}
Status Assign(Array &A,ElemType e,...){
va_start(ap,e);
if((result=Locate(A,ap,off))<=0) return result;
```

```
*(A.base+off)=e;

return OK;

}
```

回目录 上一课 下一课

第二十三课

本课主题： 二叉树的存储结构

教学目的： 掌握二叉树的两种存储结构

教学重点： 链式存储结构

教学难点： 链式存储二叉树的基本操作

授课内容：

一、复习二叉树的定义

二叉树的基本特征：每个结点的度不大于 2 。

二、顺序存储结构

```
#define MAX_TREE_SIZE 100
typedef TElemType SqBiTree[MAX_TREE_SIZE];
SqBiTree bt;
```

结点编号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
结点值	1	2	3	4	5	0	0	0	0	6	7	0	0	0	0

第 i 号结点的左右孩子一定保存在第 2i 及 2i+1 号单元中。

缺点：对非完全二叉树而言，浪费存储空间

三、链式存储结构

一个二叉树的结点至少保存三种信息：数据元素、左孩子位置、右孩子位置

对应地，链式存储二叉树的结点至少包含三个域：数据域、左、右指针域。

也可以在结点中加上指向父结点的指针域 P。

对结点有二个指针域的存储方式有以下表示方法：

```
typedef struct BiTNode{
TElemType data;
struct BitNode *lchild,*rchild;
}BiTNode,*BiTree;
```

基于该存储结构的二叉树基本操作有：

```
Status CreteBiTree(BiTree &T);
//按先序次序输入二叉树中结点的值（一个字符），空格字符表示空树，
//构造二叉链表表示的二叉树 T。
Status PreOrderTraverse(BiTree T,Status(*Visit)(TElemType e));
//采用二叉链表存储结构,Visit 是对结点操作的应用函数
//先序遍历二叉树 T,对每个结点调用函数 Visit 一次且仅一次
//一旦 visit()失败,则操作失败
```

Status InOrderTraverse(BiTree T,Status(*Visit)(TElemType e));

//采用二叉链表存储结构,Visit 是对结点操作的应用函数

//中序遍历二叉树 T,对每个结点调用函数 Visit 一次且仅一次

//一旦 visit()失败,则操作失败

Status PostOrderTraverse(BiTree T,Status(*Visit)(TElemType e));

//采用二叉链表存储结构,Visit 是对结点操作的应用函数

//后序遍历二叉树 T,对每个结点调用函数 Visit 一次且仅一次

//一旦 visit()失败,则操作失败

Status LevelOrderTraverse(BiTree T,Status(*Visit)(TElemType e));

//采用二叉链表存储结构,Visit 是对结点操作的应用函数

//层序遍历二叉树 T,对每个结点调用函数 Visit 一次且仅一次

//一旦 visit()失败,则操作失败

四、总结本课内容

顺序存储与链式存储的优缺点。

回目录 上一课 下一课

第二十四课

本课主题： 遍历二叉树

教学目的： 掌握二叉树遍历的三种方法

教学重点： 二叉树的遍历算法

教学难点： 中序与后序遍历的非递归算法

授课内容：

一、复习二叉树的定义

二叉树由三个基本单元组成：根结点、左子树、右子树

问题：如何不重复地访问二叉树中每一个结点？

二、遍历二叉树的三种方法：

先序	1	访问根结点
	2	先序访问左子树
	3	先序访问右子树
中序	1	中序访问左子树

	2	中序访问根结点
	3	中序访问右子树
后序	1	后序访问左子树
	2	后序访问右子树
	3	访问根结点

三、递归法遍历二叉树

先序：

```
Status(PreOrderTraverse(BiTree T,Status(*Visit)(TElemType e)){
if(T){
if(Visit(T->data))
if(PreOrderTraverse(t->lchild,Visit))
if(PreOrderTraverse(T->rchild,Visit)) return OK;
return ERROR;
}else return OK;
}
```

遍历结果：1,2,4,5,6,7,3

四、非递归法遍历二叉树

中序一：

```
Status InorderTraverse(BiTree T,Status(*Visit)(TElemType e)){
InitStack(S);Push(S,T);
while(!StackEmpty(S)){
while(GetTop(S,p)&&p)Push(S,p->lchild);
Pop(S,p);
if(!StackEmpty(S)){
Pop(S,p); if(!Visit(p->data)) return ERROR;
Push(S,p->rchild);
}
}
return OK;
}
```

中序二：

```
Status InorderTraverse(BiTree T,Status(*Visit)(TElemType e)){
InitStack(S);p=T;
while(p||!StackEmpty(S)){
if(p){Push(S,p);p=p->lchild;}
else{
Pop(S,p); if(!Visit(p->data)) return ERROR;
p=p->rchild);
}//else
}//while
return OK;
}
```

五、总结

二叉树遍历的意义

回目录 上一课 下一课

第二十五课

本课主题： 单元测验

教学目的： 复习前面所学的内容,检验学习效果,拾遗补缺

教学重点：

教学难点：

授课内容：

测验题:

一，填空：

1. 基本数据结构有_____，_____，_____，_____四种。

2. 存储结构可根据数据元素在机器中的位置是否连续分为_____，_____。

3. 算法的基本要求有_____，_____，_____，_____。

4. 度量算法效率可通过_____，_____两方面进行。

5. 栈的定义：_____。

二，简答：

1. 举例说明数据对象、数据元素、数据项的定义。

2. 类C语言和C语言有哪些主要区别？

3. 线性表的基本操作有哪些？

4. 写出类C语言定义的线性表的静态分配顺序存储结构。

三，算法设计：

1. 下面是线性表的存储结构和插入算法，请补充算法中空缺部分。

```
#define LIST_INIT_SIZE 100

#define LISTINCREMENT 10

typedef struct{

ElemType *elem; //存储空间基址

int length; //当前长度

int listsize; //当前分配的存储容量以一数据元素存储长度为单位

}SqList;
```

```
status ListInsert(List *L,int i,ElemType e) {

_____ *p,*q;

if (i<1||i>L->length+1) return ERROR;

q=&(L->elem[i-1]);

for(p=&L->elem[L->length-1];p>=q;--p)

_____;

*q=e;

_____;

return OK;

}/*ListInsert Before i */
```

2. 下面是栈的顺序存储结构和入栈、出栈算法，请补充算法中空缺部分。

```
typedef struct{

SElemType *base;

SElemType *top; //设栈顶栈底两指针的目的是便于判断栈是否为空

int StackSize; //栈的当前可使用的最大容量.

}SqStack;

Status Push(SqStack &S,SElemType e); {

if(S.top - s.base>=S.stacksize) {

S.base=(ElemType *) realloc(S.base,

(S.stacksize + STACKINCREMENT) * sizeof(ElemType));
```

```
if(!S.base)exit(OVERFLOW);
S.top=S.base+S.stacksize;
S.stacksize+=STACKINCREMENT;
}
*S.top++=_____;
return OK;
} //Push
Status Pop(SqStack &S,SElemType &e); {
if(_____)
return ERROR;
_____=*--S.top;
return OK;
}//Pop
```

四，问答：

1. 用图示法说明在单向线性链表中插入结点的过程。
2. 有一学生成绩单，画出用链式存储结构时的成绩单数据的存储映像。
3. 用C语言实现单向线性链表。写出存储结构定义及基本算法。

回目录 上一课 下一课

第二十六课

本课主题： 图的定义与术语

教学目的： 掌握图的定义及常用术语

教学重点： 图的常用术语

教学难点： 图的常用术语

授课内容：

一、图的定义

图是一种数据元素间为多对多关系的数据结构，加上一组基本操作构成的抽象数据类型。

ADT Graph{

数据对象 V ：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系 R：

R={VR}

VR={<v,w>|v,w(-V 且 P(v,w),<v,w>表示从 v 到 w 的弧，谓词 P(v,w)定义了弧<v,w>的意义或信息}

基本操作 P：

CreateGraph(&G,V,VR);

初始条件：V 是图的顶点集，VR 是图中弧的集合。

操作结果：按 V 和 VR 的定义构造图 G

DestroyGraph(&G);

初始条件：图 G 存在

操作结果：销毁图 G

LocateVex(G,u);

初始条件：图 G 存在，u 一 G 中顶点有相同特征

操作结果：若 G 中存在顶点 u, 则返回该顶点在图中位置；否则返回其它信息。

GetVex(G,v);

初始条件：图 G 存在，v 是 G 中某个顶点

操作结果：返回 v 的值。

PutVex(&G,v,value);

初始条件：图 G 存在，v 是 G 中某个顶点

操作结果：对 v 赋值 value

FirstAdjVex(G,v);

初始条件：图 G 存在，v 是 G 中某个顶点

操作结果：返回 v 的第一个邻接顶点。若顶点在 G 中没有邻接顶点，则返回“空”

NextAdjVex(G,v,w);

初始条件：图 G 存在,v 是 G 中某个顶点,w 是 v 的邻接顶点。

操作结果：返回 v 的（相对于 w 的）下一个邻接顶点。若 w 是 v 的最后一个邻接点，则返回“空”

InsertVex(&G,v);

初始条件：图 G 存在，v 和图中顶点有相同特征

操作结果：在图 G 中增添新顶点 v

DeleteVex(&G,v);

初始条件：图 G 存在，v 是 G 中某个顶点

操作结果：删除 G 中顶点 v 及其相关的弧

InsertAcr(&G,v,w);

初始条件：图 G 存在，v 和 w 是 G 中两个顶点

操作结果：在 G 中增添弧<v,w>,若 G 是无向的，则还增添对称弧<w,v>

DeleteArc(&G,v,w);

初始条件：图 G 存在，v 和 w 是 G 中两个顶点

操作结果：在 G 中删除弧<v,w>,若 G 是无向的，则还删除对称弧<w,v>

DFSTraverser(G,v,Visit());

初始条件：图 G 存在，v 是 G 中某个顶点，Visit 是顶点的应用函数

操作结果：从顶点 v 起深度优先遍历图 G，并对每个顶点调用函数 Visit 一次。一旦 Visit()失败，则操作失败。

BFSTRaverse(G,v,Visit());

初始条件：图 G 存在，v 是 G 中某个顶点，Visit 是顶点的应用函数

操作结果：从顶点 v 起广度优先遍历图 G，并对每个顶点调用函数 Visit 一次。一旦 Visit()失败，则操作失败。

}ADT Graph

二、图的常用术语

对上图有：G1=(V1,{A1})

其中：V1={v1,v2,v3,v4} A1={<v1,v2>,<v1,v3>,<v3,v4>,<v4,v1>}

如果用 n 表示图中顶点数目，用 e 表示边或弧的数目，则有：

对于无向图，e 的取值范围是 0 到 n(n-1)/2,有 n(n-1)/2 条边的无向图称为完全图。

对于有向图，e 有取值范围是 0 到 n(n-1)。具有 n(n-1)条弧的有向图称为有向完全图。

有很少条边或弧的图称为稀疏图，反之称为稠密图。

v1 与 v2 互为邻接点

e1 依附于顶点 v1 和 v2

v1 和 v2 相关联

v1 的度为 3

对有向图，如果每一对顶点之间都有通路，则称该图为强连通图。

三、总结

图的特征

有向图与无向图的主要区别

回目录 上一课 下一课

第二十七课

本课主题： 实验六 二叉树实验

教学目的： 掌握二叉树的链式存储结构

教学重点： 二叉树的链式存储实现方法

教学难点：

授课内容：

生成如下二叉树，并得出三种遍历结果：

一、二叉树的链式存储结构表示

```
typedef struct BiTNode{
TElemType data;
struct BitNode *lchild,*rchild;
}BiTNode,*BiTree;
```

二、二叉树的链式存储算法实现

```
CreateBiTree(&T,definition);
InsertChild(T,p,LR,c);
```

三、二叉树的递归法遍历

```
PreOrderTraverse(T,Visit());
InOrderTraverse(T,Visit());
PostOrderTraverse(T,Visit());
```

示例源程序

第二十八课

本课主题： 图的存储结构

教学目的： 掌握图的二种存储表示方法

教学重点： 图的数组表示及邻接表表示法

教学难点： 邻接表表示法

授课内容：

一、数组表示法

用两个数组分别存储数据元素（顶点）的信息和数据元素之间的关系（边或弧）的信息。

```
// 图的数组（邻接矩阵）存储表示
#define INFINITY INT_MAX //最大值无穷大
#define MAX_VERTEX_NUM 20 //最大顶点个数
typedef enum{DG,DN,AG,AN} GraphKind;//有向图，有向网，无向图，无向网
typedef struct ArcCell{
VRType adj; //VRType 是顶点关系类型。对无权图，用 1 或 0 表示相邻否，对带权图，则为权值类型
InfoType *info; //该弧相关停息的指针
}ArcCell,AdjMatrix[max_vertex_num][max_vertex_num];
tpyedef struct{
VertexType vexs[MAX_VERTEX_NUM]; //顶点向量
AdjMatrix arcs; //邻接矩阵
int vexnum,arcnum; //图的当前顶点数和弧数
GraphKind kind; //图的种类标志
}MGraph;
```

二、邻接表

邻接表是图的一种链式存储结构。

在邻接表中，对图中每个顶点建立一个单链表，第 i 个单链表中的结点表示依附于顶点 vi 的边（对有向图是以顶点 vi 为尾的弧）。每个结点由三个域组成，其中邻接点域(adjvex)指示与顶点 vi 邻接的点在图中的位置，链域(nextarc)指示下一条边或弧的结点；数据域(info)存储和边或弧相关的信息，如权值等。每个链表上附设一个表头结点，包含链域(firstarc)指向链表中第一个结点，还设有存储顶点 vi 的名或其它有关信息的数据域(data)。如：

表结点	adjvex	nextarc	info
头结点	data	firstarc	

```
#define MAX_VERTEX_NUM 20
typedef struct ArcNode{
int adjvex; //该弧所指向的顶点的位置
struct ArcNode *nextarc; //指向下一条弧的指针
InfoType *info; //该弧相关信息的指针
}ArcNode;
typedef struct VNode{
VertexType data; //顶点信息
ArcNode *firstarc; //指向第一条依附该顶点的弧的指针
}VNode,AdjList[MAX_VERTEX_NUM];
typedef struct {
AdjList vertices; //图的当前顶点数和弧数
int vexnum,arcnum; //图的种类标志
int kind;
```

}ALGraph;

三、总结

图的存储包括哪些要素？

回目录 上一课 下一课

第二十九课

本课主题： 静态查找表（一）顺序表的查找

教学目的： 掌握查找的基本概念，顺序表查找的性能分析

教学重点： 查找的基本概念

教学难点： 顺序表查找的性能分析

授课内容：

一、查找的基本概念

查找表：	是由同一类型的数据元素（或记录）构成的集合。
查找表的操作：	1、查询某个“特定的”数据元素是否在查找表中。 2、检索某个“特定的”数据元素的各种属性。 3、在查找表中插入一个数据元素；

	4、从查找表中删去某个数据元素。
静态查找表	对查找表只作前两种操作
动态查找表	在查找过程中查找表元素集合动态改变
关键字	是数据元素（或记录）中某个数据项的值
主关键字	可以唯一的地标识一个记录
次关键字	用以识别若干记录
查找	根据给定的某个值，在查找表中确定一个其关键字等于给定的记录或数据元素。若表中存在这样的一个记录，则称查找是成功的，此时查找的结果为给出整个记录的信息，或指示该记录在查找表中的位置；若表中不存在关键字等于给定值的记录，则称查找不成功。

一些约定：

典型的关键字类型说明：

```
typedef float KeyType;//实型
typedef int KeyType;//整型
typedef char *KeyType;//字符串型
```

数据元素类型定义为：

```
typedef struct{
KeyType key; // 关键字域
...
}ElemType;
```

对两个关键字的比较约定为如下的宏定义：

```
对数值型关键字
#define EQ(a,b) ((a)==(b))
#define LT(a,b) ((a)<(b))
#define LQ(a,b) ((a)<=(b))
对字符串型关键字
#define EQ(a,b) (!strcmp((a),(b)))
#define LT(a,b) (strcmp((a),(b))<0)
#define LQ(a,b) (strcmp((a),(b))<=0)
```

二、静态查找表

静态查找表的类型定义：

```
ADT StaticSearchTable{
数据对象 D：D 是具有相同特性的数据元素的集合。各个数据元素均含有类型相同，可唯一标识数据元素的关键字。
数据关系 R：数据元素同属一个集合。
基本操作 P：
Create(&ST,n);
操作结果：构造一个含 n 个数据元素的静态查找表 ST。
Destroy(&ST);
初始条件：静态查找表 ST 存在。
操作结果：销毁表 ST。
Search(ST,key);
初始条件：静态查找表 ST 存在，key 为和关键字类型相同的给定值。
操作结果：若 ST 中在在其关键字等于 key 的数据元素，则函数值为该元素的值或在表中的位置，否则为“空”。
Traverse(ST,Visit());
初始条件：静态查找表 ST 存在，Visit 是对元素操作的应用函数。
操作结果：按某种次序对 ST 的每个元素调用函数 visit()一次且仅一次。一旦 visit()失败，则操作失败。
}ADT StaticSearchTable
```

三、顺序表的查找

静态查找表的顺序存储结构

```
typedef struct {
ElemType *elem;
int length;
}SSTable;
```

顺序查找：从表中最后一个记录开始，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值比较相等，则查找成功，找到所查记录；反之，查找不成功。

```
int Search_Seq(SSTable ST,KeyType key){
ST.elme[0].key=key;
for(i=ST.length; !EQ(ST.elem[i].key,key); --i);
return i;
}
```

查找操作的性能分析：

查找算法中的基本操作是将记录的关键字和给定值进行比较，，通常以“其关键字和给定值进行过比较的记录个数的平均值”作为衡量依据。

平均查找长度：

为确定记录在查找表中的位置，需用和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的平均查找长度。

	其中：Pi 为查找表中第 i 个记录的概率，且　　； Ci 为找到表中其关键字与给定值相等的第 i 个记录时，和给定值已进行过比较的关键字个数。
	等概率条件下有：

假设查找成功与不成功的概率相同：	

四、总结

什么是查找表

顺序表的查找过程

回目录 上一课 下一课

第三十课

本课主题： 静态查找表（二）有序表的查找

教学目的： 掌握有序表的折半查找法

教学重点： 折半查找

教学难点： 折半查找

授课内容：

一、折半查找的查找过程

以有序表表示静态查找表时，Search 函数可用折半查找来实现。

先确定待查记录所在的范围（区间），然后逐步缩小范围直到找到或找不到该记录为止。

二、折半查找的查找实现

```
int Search_Bin(SSTable ST,KeyType key){

low=1;high=ST.length;

while(low<=high){

mid=(low+high)/2;

if EQ(key,ST.elem[mid].key) return mid;

else if LT(key,ST.elem[mid].key) high=mid -1;

else low=mid +1 ;
```

}

return 0;

}//Search_Bin;

三、折半查找的性能分析

折半查找在查找成功时和给定值进行比较的关键字个数至多为

回目录 上一课 下一课

第三十一课

本课主题： 动态查找表

教学目的： 掌握二叉排序树的实现方法

教学重点： 二叉排序树的实现

教学难点： 构造二叉排序树的方法

授课内容：

一、动态查找表的定义

动态查找表的特点是：

表结构本身是在查找过程中动态生成的，即对于给定值 key,若表中存在其关键字等于 key 的记录，则查找成功返回，否则插入关键字等于 key 的记录。以政是动态查找表的定义：

ADT DymanicSearchTable{

数据对象 D：D 是具有相同特性的数据元素的集合。各个数据元素均含有类型相同，可唯一标识数据元素的关键字。

数据关系Ｒ：数据元素同属一个集合。

基本操作Ｐ：

InitDSTable(&DT);

DestroyDSTable(&DT);

SearchDSTable(DT,key);

InsertDSTable(&DT,e);

DeleteDSTable(&DT,key);

TraverseDSTable(DT,Visit());

}ADT DynamicSearchTable

二、二叉排序树及其查找过程

二叉排序树或者是一棵空树；或者是具有下列性质的二叉树：

１、若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

２、若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

３、它的左、右子树了分别为二叉排序树。

如果取二叉链表作为二叉排序树的存储结构，则上述查找过程如下：

```
BiTree SearchBST(BiTree T,KeyType key){
if(!T)||EQ(key,T->data.key)) return (T);
else if LT(key,T->data.key) return (SearchBST(T->lchild,key));
else return (SearchBST(T->rchild.key));
}//SearchBST
```

三、二叉排序树的插入和删除

二叉排序树是一种动态树表，其特点是，树的结构通常不是一资生成的，面是在查找过程中，当树中不存在关键字等于给定值的结点时再进行插入。新插入的结点一定是一个新添加的叶子结点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。

```
Status SearchBST(BiTree T,KeyType key,BiTree f,BiTree &p){
if(!T) {p=f;return FALSE;}
else if EQ(key,T->data.key){ p=T;return TRUE;}
else if LT(key,T->data.key) SearchBsT(T->lchild,key,T,p);
else SearchBST(T->rchild,key,T,p);
}//SearchBST
```

插入算法：

```
Status InsertBST(BiTree &T,ElemType e){
if(!SearchBST(T,e.key,NULL,p){
s=(BiTree)malloc(sizeof(BiTNode));
s->data=e;s->lchild=s->rchild=NULL;
if(!p) T=s;
else if (LT(e.key,p->data.key) p->lchild=s;
else p->rchild=s;
return TRUE;
}
```

```
else return FALSE;

}//InsertBST
```

在二叉排序树中删除一个节点的算法：

```
Status DeleteBST(BiTree &T,KeyType key){

if(!T) return FALSE;

else{

if EQ(key,T->data.key) Delete(T);

else if LT(key,T->data.key) DeleteBST(T->lchild,key);

else DeleteBST(T->rchild,key);

return TRUE;

}

}

void Delete(BiTree &p){

if(!p->rchild){

q=p; p=p->lchild; free(q);

}

else if(!p->lchild){

q=p;p=p->rchild; free(q);

}
```

```
else{
//方法一：如图示


q=p;s=p->lchild;
while(s->rchild){q=s;s=s->rchild}//转左，然后向右到尽头
p->data=s->data; //s 指向被删结点的"前驱"
if(q!=p)q->rchild=s->lchild; //重接*q 的右子树
else q->lchild=s->lchild;//重接*q 的左子树（方法一结束）
//或可用方法二：
//q=s=(*p)->l;
//while(s->r) s=s->r;
//s->r=(*p)->r;
```

```
//free(*p);

//(*p)=q;









}

}
```

请看一个示例源程序。

四、总结

回目录 上一课 下一课

第三十二课

本课主题： 哈希表（一）

教学目的： 掌握哈希表的概念作用及意义，哈希表的构造方法

教学重点： 哈希表的构造方法

教学难点： 哈希表的构造方法

授课内容：

一、哈希表的概念及作用

一般的线性表，树中，记录在结构中的相对位置是随机的，即和记录的关键字之间不存在确定的关系，因此，在结构中查找记录时需进行一系列和关键字的比较。这一类查找方法建立在“比较“的基础上，查找的效率依赖于查找过程中所进行的比较次数。

理想的情况是能直接找到需要的记录，因此必须在记录的存储位置和它的关键字之间建立一个确定的对应关系 f，使每个关键字和结构中一个唯一的存储位置相对应。

哈希表最常见的例子是以学生学号为关键字的成绩表，1 号学生的记录位置在第一条， 1 0 号学生的记录位置在第 1 0 条...

如果我们以学生姓名为关键字，如何建立查找表，使得根据姓名可以直接找到相应记录呢？

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
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

	刘丽	刘宏英	吴军	吴小艳	李秋梅	陈伟	...
姓名中各字拼音首字母	ll	lhy	wj	wxy	lqm	cw	...
用所有首字母编号值相加求和	24	46	33	72	42	26	...
最小值可能为 3 最大值可能为 78 可放 75 个学生							

用上述得到的数值作为对应记录在表中的位置，得到下表：

		成绩一	成绩二...
3	...		
...	...		
24	刘丽	82	95
25	...		
26	陈伟		
...	...		
33	吴军		
...	...		

42	李秋梅		
...	...		
46	刘宏英		
...	...		
72	吴小艳		
...	...		
78	...		

上面这张表即哈希表。

如果将来要查李秋梅的成绩，可以用上述方法求出该记录所在位置：

李秋梅:lqm 12+17+13=42 取表中第 42 条记录即可。

问题：如果两个同学分别叫刘丽 刘兰 该如何处理这两条记录？

这个问题是哈希表不可避免的，即冲突现象：对不同的关键字可能得到同一哈希地址。

二、哈希表的构造方法

1、直接定址法

例如：有一个从 1 到 100 岁的人口数字统计表，其中，年龄作为关键字，哈希函数取关键字自身。

地址	01	02	...	25	26	27	...	100
年龄	1	2	...	25	26	27	...	...
人数	3000	2000	...	1050	...	...	...	...
...								

2、数字分析法

有学生的生日数据如下：

年.月.日

75.10.03

75.11.23

76.03.02

76.07.12

75.04.21

76.02.15

...

经分析,第一位，第二位，第三位重复的可能性大，取这三位造成冲突的机会增加，所以尽量不取前三位，取后三位比较好。

３、平方取中法

取关键字平方后的中间几位为哈希地址。

４、折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为哈希地址，这方法称为折叠法。

例如：每一种西文图书都有一个国际标准图书编号，它是一个 10 位的十进制数字，若要以它作关键字建立一个哈希表，当馆藏书种类不到 10,000 时，可采用此法构造一个四位数的哈希函数。如果一本书的编号为 0-442-20586-4,则：

```
          5864                    5864
          4220                    0224
    +)      04              +)      04
    ----------              ----------
         10088                   6092
      H(key)=0088             H(key)=6092

       (a)移位叠加              (b)间界叠加
```

５、除留余数法

取关键字被某个不大于哈希表表长 m 的数 p 除后所得余数为哈希地址。

H(key)=key MOD p (p<=m)

６、随机数法

选择一个随机函数，取关键字的随机函数值为它的哈希地址，即

H(key)=random(key) ,其中 random 为随机函数。通常用于关键字长度不等时采用此法。

三、总结

哈希表的优缺点

四、作业

预习如何处理冲突及哈希表的查找。

回目录 上一课 下一课

第三十三课

本课主题： 哈希表（二）

教学目的： 掌握哈希表处理冲突的方法及哈希表的查找算法

教学重点： 哈希表处理冲突的方法

教学难点： 开放定址法

授课内容：

一、复习上次课内容

什么是哈希表？如何构造哈希表？

提出问题：如何处理冲突？

二、处理冲突的方法

		成绩一	成绩二...
3	...		
...	...		
24	刘丽	82	95
25	...		
26	陈伟		
...	...		
33	吴军		
...	...		
42	李秋梅		
...	...		
46	刘宏英		
...	...		

72	吴小艳		
...	...		
78	...		

如果两个同学分别叫刘丽 刘兰，当加入刘兰时，地址 24 发生了冲突，我们可以以某种规律使用其它的存储位置，如果选择的一个其它位置仍有冲突，则再选下一个，直到找到没有冲突的位置。选择其它位置的方法有：

1 、开放定址法

Hi=(H(key)+di) MOD m i=1,2,...,k(k<=m-1)

其中 m 为表长，di 为增量序列

如果 di 值可能为 1,2,3,...m-1，称线性探测再散列。

如果 di 取值可能为 1,-1,2,-2,4,-4,9,-9,16,-16,...k*k,-k*k(k<=m/2)

称二次探测再散列。

如果 di 取值可能为伪随机数列。称伪随机探测再散列。

例：在长度为 11 的哈希表中已填有关键字分别为 17,60,29 的记录，现有第四个记录，其关键字为 38，由哈希函数得到地址为 5，若用线性探测再散列，如下：

0	1	2	3	4	5	6	7	8	9	10
					60	17	29			

(a)插入前

0	1	2	3	4	5	6	7	8	9	10
					60	17	29	38		

(b)线性探测再散列

0	1	2	3	4	5	6	7	8	9	10
					60	17	29			

(c)二次探测再散列

0	1	2	3	4	5	6	7	8	9	10
			38		60	17	29			

(d)伪随机探测再散列

伪随机数列为 9,5,3,8,1...

２、再哈希法

当发生冲突时，使用第二个、第三个、哈希函数计算地址，直到无冲突时。缺点：计算时间增加。

３、链地址法

将所有关键字为同义词的记录存储在同一线性链表中。

４、建立一个公共溢出区

假设哈希函数的值域为[0,m-1],则设向量 HashTable[0..m-1]为基本表，另外设立存储空间向量 OverTable[0..v]用以存储发生冲突的记录。

三、哈希表的查找

```
//开放定址哈希表的存储结构
int hashsize[]={997,...};
typedef struct{
ElemType *elem;
int count;
int sizeindex;
}HashTable;
#define SUCCESS 1
#define UNSUCCESS 0
#define DUPLICATE -1
Status SearchHash(HashTable H,KeyType K,int &p,int &c){
```

```
p=Hash(K);
while(H.elem[p].key!=NULLKEY && !EQ(K,H.elem[p].key))
collision(p,++c);
if(EQ(K,H.elem[p].key)
return SUCCESS;
else return UNSUCCESS;
}
Status InsertHash(HashTable &H,EleType e){
c=0;
if(SearchHash(H,e.key,p,c))
return DUPLICATE;
else if(c<hashsize[H.sizeindex]/2){
H.elem[p]=e; ++H.count; return OK;
}
else RecreateHashTable(H);
}
```

四、总结

处理冲突的要求是什么？

回目录 上一课 下一课

第三十四课

本课主题： 插入排序，快速排序

教学目的： 掌握排序的基本概念，插入排序、快速排序的算法

教学重点： 插入排序、快速排序的算法

教学难点： 快速排序算法

授课内容：

一、排序概述

排序：将一个数据元素的无序序列重新排列成一个按关键字有序的序列。

姓名	年龄	体重
1 李由	57	62

2 王天	54	76
3 七大	24	75
4 张强	24	72
5 陈华	24	53

上表按年龄无序，如果按关键字年龄用某方法排序后得到下表：

姓名	年龄	体重
3 七大	24	75
4 张强	24	72
5 陈华	24	53
2 王天	54	76
1 李由	57	62

注意反色的三条记录保持原有排列顺序，则称该排序方法是稳定的！

如果另一方法排序后得到下表：

姓名	年龄	体重
4 张强	24	72
3 七大	24	75
5 陈华	24	53
2 王天	54	76
1 李由	57	62

原 3,4,5 记录顺序改变，则称该排序方法是不稳定的！

内部排序：待排序记录存放在计算机随机存储器中进行的排序过程；

外部排序：待排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。

二、插入排序

1、直接插入排序

基本操作是将一个记录插入到已排好序的有序表中，从而得到一个新的、记录数增 1 的有序表。排序过程：

38	49	65	97	76	13	27	49	...	

38	49	65	76	97	13	27	49	...	

13	38	49	65	76	97	27	49	...	

13	27	38	49	65	76	97	49	...	

13	27	38	49	49	65	76	97	...	

2、折半插入排序

在直接插入排序中，为了找到插入位置，采用了顺序查找的方法。为了提高查找速度，可以采用折半查找，这种排序称折半插入排序。

3、2－路插入排序

为减少排序过程中移动记录的次数，在折半插入排序的基础上加以改进：

49	38	65	97	78	13	27	49	...	

i=1	49								
	first								
i=2	49								38
	final								first
i=3	49	65							38
		final							first
i=4	49	65	97						38
			final						first
i=5	49	65	76	97					38
				final					first
i=6	49	65	76	97				13	38

				final			first	
i=7	49	65	76	97		13	27	38
				final		first		
i=8	49	49	65	76	97	13	27	38
					final	first		

三、快速排序

1、起泡排序

首先将第一个记录的关键字和第二个记录的关键字进行比较，若为逆序，则将两个记录交换之，然后比较第二个记录和第三个记录的关键字。直至第 n-1 个记录和第 n 个记录的关键字进行过比较为止。

然后进行第二趟起泡排序，对前 n-1 个记录进行同样操作。

...直到在某趟排序过程中没有进行过交换记录的操作为止。

49	38	38	38	38	13	13
38	49	49	49	13	27	27
65	65	65	13	27	38	38
97	76	13	27	49	49	
76	13	27	49	49		
13	27	49	65			
27	49	78				
49	97					
初始	第一趟	第二趟	第三趟	第四趟	第五趟	第六趟

2、快速排序

通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

初始关键字	49	38	65	97	76	13	27	49
	i						j	j
1 次交换之后	27	38	65	97	76	13		49

	i		i				j	
2 次交换之后	27	38		97	76	13	65	49
			i			j	j	
3 次交换之后	27	38	13	97	76		65	49
			i	i		j		
4 次交换之后	27	38	13		76	97	65	49
				ij		j		
完成一趟排序	27	38	13	49	76	97	65	49

初始状态	49	38	65	97	76	13	27	49
一次划分	27	38	13	49	76	97	65	49
分别进行	13	27	38					
	结束		结束		49	65	76	97
					49	65		结束
						结束		
有序序列	13	27	38	49	49	65	76	97

四、总结

几种排序的简单分析与比较。（时间、空间复杂度）

五、作业

实现直接插入排序、起泡排序、快速排序算法。

回目录 上一课 下一课

第三十五课

本课主题： 实验七 查找

教学目的： 练习顺序查找、折半查找及二叉排序树的实现

教学重点：

教学难点：

授课内容：

顺序查找

折半查找

顺序查找及折半查找示例

二叉排序树

示例

回目录 上一课 下一课

第三十六课

本课主题： 选择排序，归并排序

教学目的： 掌握选择排序，归并排序算法

教学重点： 选择排序之堆排序，归并排序算法

教学难点： 堆排序算法

授课内容：

一、选择排序

每一趟在 n-i+1(i=1,2,...n-1)个记录中选取关键字最小的记录作为有序序列中第 i 个记录。

二、简单选择排序

算法：

```
Smp_Selecpass(ListType &r,int i)
{
k=i;
for(j=i+1;j<n;i++)
if (r[j].key<r[k].key)
k=j;
if (k!=i)
{ t=r[i];r[i]=r[k];r[k]=t;}
}
Smp_Sort(ListType &r)
{
for(i=1;i<n-1;i++)
Smp_Selecpass(r,i);
}
```

三、树形选择排序

又称锦标赛排序，首先对 n 个记录的关键字进行两两比较，然后在其中一半较小者之间再进行两两比较，如此重复，直到选出最小关键字的记录为止。

四、堆排序

只需要一个记录大小的辅助空间，每个待排序的记录仅占有一个存储空间。

什么是堆？n 个元素的序列{k1,k2,...,kn}当且仅当满足下列关系时，称之为堆。关系一：ki<=k2i 关系二：ki<=k2i+1（i=1,2,...,n/2)

堆排序要解决两个问题：1、如何由一个无序序列建成一个堆？2、如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？

问题 2 的解决方法：

```
sift(ListType &r,int k,int m)
    {
    i=k;j=2*i;x=r[k].key;finished=FALSE;
    t=r[k];
    while((j<=m)&&(!finished))
    {
    if ((j<m)&&(r[j].key>r[j+1].key)) j++;
```

```
if (x<=r[j].key)

finished:=TRUE;

else {r[i]=r[j];i=j;j=2*i;}

}

r[i]=t;

}

HeapSort(ListType &r)

{

for(i=n/2;i>0;i--) sift(r,i,n);

for(i=n;i>1;i--){

r[1]<->r[i];

sift(r,i,i-1)

}

}
```

五、归并排序

将两个或两个以上的有序表组合成一个新的有序表的方法叫归并。

假设初始序列含有 n 个记录，则可看成是 n 个有序的子序列，每个子序列的长度为 1，然后两两归并，得到 n/2 个长度为 2 或 1 的有序子序列；再两两归并，如此重复。

```
merge(ListType r,int l,int m,int n,ListType &r2)

{

i=l;j=m+1;k=l-1;

while(i<=m) and(j<n) do

{

k=k+1;

if (r[i].key<=r[j].key) {r2[k]=r[i];i++;}

else {r2[i]=r[j];j++}

}

if (i<=m) r2[k+1..n]=r[i..m];

if (j<=n) r2[k+1..n]=r[j..n];

}

mergesort(ListType &r,ListType &r1,int s,int t)

{

if (s==t)
```

```
r1[s]=r[s];
else
{
mergesort(r,r2,s,s+t/2);
mergesort(r,r2,s+t/2+1,t);
merge(r2,s,s+t/2,t,r1);
}
]
```

六、总结

回目录 上一课 下一课

第三十七课

本课主题： 实验八 排序实验

教学目的： 掌握简单插入排序、快速排序、堆排序的算法并加以应用。

教学重点：

教学难点：

授课内容：

实现下述三种算法，并用以下无序序列加以验证：

49，38，65，97，76，13，27，49

一、简单插入排序

二、快速排序

三、堆排序

以上算法的 C 源程序。

回目录 上一课 下一课

第三十八课

本课主题： 文件概念，顺序文件

教学目的： 掌握文件基本概念，顺序文件的概念。

教学重点： 文件基本概念

教学难点： 逻辑结构与物理结构的关系。

授课内容：

一、表与文件

和表类似，文件是大量记录的集合。习惯上称存储在主存储器（内存储器）中的记录集合为表，称存储在二级存储器（外存储器）中的记录集合为文件。

二、文件基本概念

文件：是由大量性质相同的记录组成的集合。

<table>
<tr><td rowspan="2">文件按记录类型不同分类</td><td>操作系统的文件</td><td>一维的连续的字符序列</td></tr>
<tr><td>数据库文件</td><td>带有结构的记录的集合,每条记录是由一个或多个数据项组成的集合。
<table>
<tr><td>姓名</td><td>准考证号</td><td>政治</td><td>语文</td><td>数学</td><td>外语</td></tr>
<tr><td>刘青</td><td>1501</td><td>78</td><td>90</td><td>100</td><td>95</td></tr>
<tr><td>张朋</td><td>1502</td><td>64</td><td>88</td><td>90</td><td>74</td></tr>
<tr><td>崔永</td><td>1503</td><td>90</td><td>100</td><td>85</td><td>89</td></tr>
<tr><td>郑琳</td><td>1504</td><td>85</td><td>73</td><td>90</td><td>91</td></tr>
<tr><td>...</td><td></td><td></td><td></td><td></td><td></td></tr>
</table></td></tr>
</table>

文件按记录长度是否相同分类	定长记录文件	文件中每个记录含有信息长度相同。
	不定长记录文件	文件中每个记录含有信息长度不等。

记录的逻辑结构是指记录在用户或应用程序员面前呈现的方式，是用户对数据的表示和存取方式。

姓名	准考证号	政治	语文	数学	外语
刘青	1501	78	90	100	95
张朋	1502	64	88	90	74
崔永	1503	90	100	85	89
郑琳	1504	85	73	90	91
. . .					

这张成绩表呈现的结构即是逻辑结构。

记录的物理结构是数据在物理存储器上存储的方式。一条物理记录指的是计算机用一条 I/O 命令进行读写的基本数据单位。

三、顺序文件

顺序文件中的物理记录的顺序和逻辑记录的顺序是一致的。

四、总结

回目录 上一课 下一课

第三十九课

本课主题： 索引文件

教学目的： 掌握索引文件的有关概念

教学重点： 索引文件的基本概念，索引文件的重要意义

教学难点： 索引文件的建立

授课内容：

一、索引文件的基本概念

除了文件本身（称作数据区）之外，别建立一张指示逻辑记录和物理记录之间一一对应关系的表--索引表。

索引表中的每一项称作索引项。不论主文件是否按关键字有序，索引表中的索引项总是按关键字（或逻辑记录号）顺序排列。

若数据区中的记录也按关键字顺序排列，则称索引顺序文件。反之，若数据区中记录不按关键字顺序排列，则称非顺序文件。

数据区：

物理记录号	姓名	年龄	体重（关键字）
1	李由	57	62
2	王天	54	76
3	七大	24	75
4	张强	24	72
5	陈华	24	53

索引表：

体重（关键字）	物理记录号
53	5
62	1
72	4
75	3
76	2

有了按体重索引的索引表后，按体重查找学生可先在索引表中查找（因索引表中按体重有序，所以可用效率高的查找算法）然后得到对应的物理记录号后到数据区取出对应物理记录。

索引文件可以大大提高表查找的速度。因为索引表容量小，且索引表按关键字有序。

二、索引文件的建立

在记录输入建立数据区的同时建立一个索引表，表中的索引项按记录输入的先后次序排列，待全部记录输入完毕后再对索引表进行排序。

回目录 上一课 下一课

第四十课

本课主题： 总复习

教学目的： 数据结构综述

教学重点： 数据结构课程的核心

教学难点： 理解概念

授课内容：

一、学习数据结构的意义

设想一下，你决定向一个公司投资，而你对某个公司的了解只限于该公司的一条生产线每分钟可生产 2000 件产品，你会作出投资的决定吗？如果你是一个公司的管理者，这个公司日常的每笔交易的详细情况对你来讲的确重要，但如果你把时间花在这些数据上面，你就无法站在宏观的高度上把握公司的经营方向。

不管是经营一个公司，还是管理一个国家，对描述事物特征的数据必须加以分析与加工，现实事物是普遍联系的，描述这些事物属性及特征的数据之间也是普遍联系的，把这些数据之间的关系进行总结，得到集合、线性、树、图这四种基本关系，由此得到四类基本数据结构。而每种结构类型的数据，相同的操作（如遍历、查找等）需要采用不同的方法（算法），不同结构类型可进行的操作也有区别。通过应用这些算法，可得到事物的总体抽象特征。如：一个公司的年产值，年利润总额，利润率等。

反过来，为了描述一个复杂的事物，必须分析它的组成部分，既要描述每个部分的特征，又要描述各个部分之间的关系，如此细分下去，便于最终用计算机进行处理，而计算机的基本数据类型不适合描述复杂的结构，且仅用基本数据类型也不便于人的理解与记忆，所以使用介于两者之间的抽象数据类型成了计算机语言描述现实事物的纽带。人可以方便的把事物用抽象数据类型描述，也可以方便的把抽象数据类型用基本数据类型来实现，为用计算机处理现实问题提供了解

决方法。

二、数据结构的学习重点

如何描述一种新的抽象数据类型？

如何分析算法的优劣？

线性表的主要特征。

线性表的存储表示（顺序表示、单向链表、循环链表、双向链表）

特殊的线性表：栈、队列、串

二叉树的定义、性质、存储结构、遍历算法

图的定义、术语、存储结构

静态查找表、二叉排序树、哈希函数的构造及冲突处理方法。

插入排序、快速排序、选择排序

回目录 上一课