金牌奥林匹克丛书

信息学解题思路与方法

Xinxixue Jieti Silu yu Fangfa

江 涛等 编著

深入剖析

巧妙点拨

融会贯通

举一反三

安徽科学技术出版社

《金牌奥林匹克》丛书

信息学解题思路与方法

江　涛
张　辰　编著

安徽科学技术出版社

图书在版编目(CIP)数据

信息学解题思路与方法/江涛,张辰编著.—合肥:安徽科学技术出版社,2002
(金牌奥林匹克丛书)
ISBN 7-5337-2417-8

Ⅰ.信… Ⅱ.①江…②张… Ⅲ.计算机课-中学-竞赛题-解题 Ⅳ.G634.675

中国版本图书馆 CIP 数据核字(2001)第 081035 号

*

安徽科学技术出版社出版
(合肥市跃进路 1 号新闻出版大厦)
邮政编码:230063
电话号码:(0551)2825419
新华书店经销 合肥中德印刷培训中心印刷厂印刷

*

开本:850×1168 1/32 印张:7.5 字数:145 千
2002 年 5 月第 1 版 2002 年 5 月第 1 次印刷
印数:4 000
ISBN 7-5337-2417-8/TP·74 定价:11.00 元

前　言

经常有外地的老师(选手)见到我的第一件事就是问应该教学生(他们)做些什么题目？但一经了解，发现选手学了一本程序设计语言的书后，要么认为接下来就是做各种各样的竞赛题了，要么就不知道下一步该做什么了。其实，一些必要的数据结构和编程知识是非常重要的。但对于初级水平的竞赛选手，如果一开始就按部就班地学习“数据结构”和“实用算法”等知识，是有很大难度的。所以，作者出此书的目的是希望读者通过本书的内容在信息学竞赛中能快速入门，更希望本书能起到抛砖引玉的作用。

本书框架结构及内容简介如下：

第一章至第四章的内容是“数据结构”和“实用算法”等知识的精选，并用深入浅出的语言代替了抽象严谨的数学描述，即使是初级选手也可以很快掌握。

第五章至第六章是通过对目前极为流行的解题方法——动态规划的介绍，使读者了解当前最新的解题思想和方法，并从中体会到信息学竞赛中分析问题、解决问题的一般思路和方法。此外，为了特别强调数学分析的重要性，本书将递推关系也单独作了介绍。

最后四章是对信息学竞赛中经常考的几个基本问题的总结，目的是让初学者学会如何总结知识，如何研究问题。

本书是由芜湖一中江涛老师根据平时竞赛、培训等讲义改编而成，部分内容由学生张辰编写。

作　者

目　录

第一章　基本数据结构快速入门

一谈到数据结构有些同学就觉得是新东西，比较难懂。其实，它与我们在程序设计语言中的简单数据类型很相似，可以看成是它的一种延伸。

简单数据类型就是把常用的一些数据类型直接做到语言系统中，并提供相应的运算，供程序员直接使用。例如，TP7.0 提供了实数类型及其相应的一些运算（+，-，*，/等），我们在计算两个实数相加时只要写 x+y 就可以了。至于它在计算机内是怎么表示的、怎样完成加法的，都由编译系统完成，我们不必去一一研究。

但是任何一个程序设计语言都不可能把所有常用的数据类型——特别是一些需要在具体应用中加以变化的数据表示及其方法——都预先做好，提供给程序员。因此，很多数学模型的数据表示方法以及对它们的操作，都要根据具体情况来即时构造。

研究问题的共性是形成一门学科的开始。数据结构就是把解决问题过程中常用的一些数据表示——数据描述及其相应的一些运算操作——方法总结、抽象出来，上升到理论高度分类并加以研究，以便在解决问题的具体算法中更好地应用。

计算机解决现实世界中的问题时，多数是研究处理对象的性质及其之间的关系，例如，城市之间架设通信线路问题。我们可以这样认为：各个城市是要处理的对象，它们之间的通信线路是它们之间的关系。抽象地说，我们用节点表示对象，线表示关系，就得到一个抽象的数学模型——图结构。

科学研究的一个常用方法是从简单到复杂，从特殊到一般。研究点与线的关系的学问称为“图论”，在数据结构中我们称之为

图的问题。图的关系比较复杂，我们可以先研究其特殊情况，这就是树。更简单的有一种特殊“树”结构——线性数据结构，如图 1－1 所示：

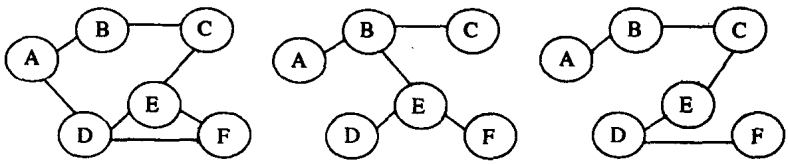


图 1－1

《数据结构》是一门独立的课程，有很多内容，这里介绍一些最常用的内容。下面分三节，从简单到复杂，分别讨论线性数据结构、树、图三种基本数据结构。以后我们可以看到，程序设计中，这三种数据结构是最常用的，其他一些具体的、复杂的数据结构也常常由它们变换或综合而得到。

§1－1 线性数据结构

简单地讲，线性数据结构就是把处理对象排成一排。

设有节点 $A_1, A_2, A_3, \cdots, A_n$，它们之间只有相邻结点才有前后关系，即关系表示为：

$$\begin{cases} A_i < A_{i+1} & i<j, i=1,\cdots,n-1 \\ A_i > A_{i-1} & i<j, i=2,\cdots,n \end{cases}$$

这里＜，＞表示前后关系，前后是两个相反方向，因此，我们通常认为线性数据结构* 是有方向的。用抽象的节点与线表示有下面几种。

线性数据结构具体到程序设计中又怎么来实现呢？

---

* 后面用“表”作为“线性数据结构”的简称。

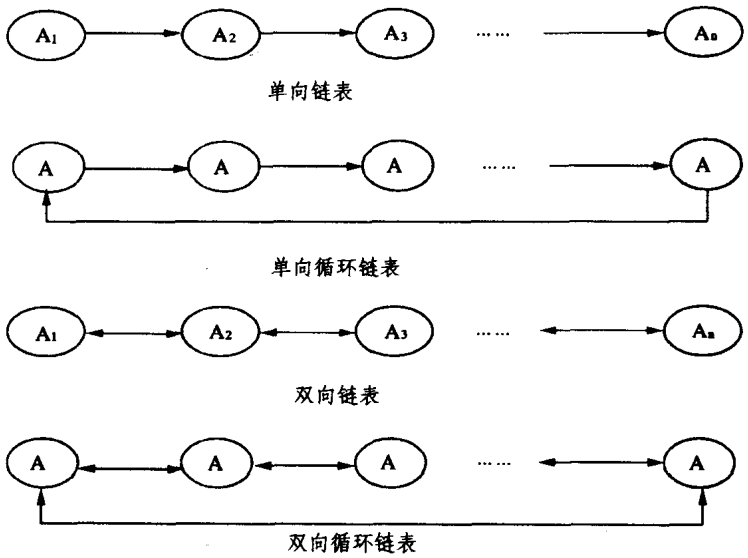


图 1-2

一、表的数组实现

在程序设计中,一种最简单的表的实现方法就是用数组。数组的下标自然具有连续性,即第 i+1 个元素为第 i 个的后一个,第 i-1 个元素是第 i 个的前一个。因此,可以看成是单向或双向链表。而且,数组不仅能知道第 i 项的前后,更能方便、直接地任意存取第 i 项,我们称其为随机存储结构。这个性质在很多情况下非常有用(可惜不是所有表的实现都有这个性质),所以这正是数组被广泛应用的原因。

例 1-1 求 Fibonacci 数列的前 N 项。

我们知道 Fibonacci 数列的定义为:

$$\begin{cases} f(n)=f(n-1)+f(n-2), n>2 \\ f(1)=1, f(2)=2 \end{cases}$$

定义中的递推式使用数组正好可以得到很好地解决:

```
f[1]:=1;  f[2]:=1;
for i:=3 to n do
    f[i]:=f[i-1]+f[i-2];
```

用数组实现循环链表也很方便，请看下面的例子。

例 1-2 猴子选大王。

N 只猴子围成一圈，从第 P 个开始，每隔 M 只报数，报到的退出，直至剩下一只为止，最后剩下的为猴子大王。问：猴大王是原来的第几只猴子。

显然，数猴子时是循环进行的，用循环链表比较好。但如果用数组实现，最简单的方法是：

```
const
  MaxN=1000;
Var
  monkey:array[1..MaxN]of integer;  {N个猴子的标记}
  n,nn,p,m,i,j      :integer;
function No(i:integer):integer;   {处理循环计数}
  begin
    while  i>n  do i:=i-n;
    No:=i;
  end;
begin
  readln(n,p,m);
  for i:=1 to n do monkey[i]:=i;
  p:=No(p+m-1); nn:=n;
  for i:=1 to n-1 do
  begin
     dec(nn);
     for j:=p to nn do monkey[j]:=monkey[j+1];
     p:=No(p+1);
  end;
  writeln('The king of Monkey is ',monkey[p]);
end.
```

二、表的指针实现

1. 串联成表

将数据节点用指针依次串联起来形成一条链，称为表的指针实现。

由于需要用指针才能知道其相邻的节点地址，因此每个节点都要加一个辅助空间来放指针。另外，我们还需要一个指针来指出这个表的第一个节点的地址。通常加一个节点，称其为头节点或哨兵，具体实现参见例 1-3。

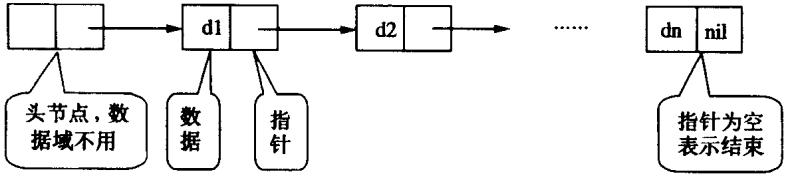


图 1-3

注意 头节点(哨兵)技术虽然多用了一个节点，但方便了编程。

例 1-3 单词分组。

读入 N(N<10000)个姓名，每个姓名由不多于 20 个小写字母组成。试编程按姓名的第一个字母分成 26 个集合。

本题的解法显然是开 26 个链表。由于事先不知道每个链表的长度(最大可能为 N)，加上空间问题不好解决，用数组的方法当然也不好。因此用指针来实现。下面给出了完整的程序：

```
program example3;
type
  TPnode   =^Tnode;  { 节点的定义 }
  Tnode    =record
              name :string[20];
              next :TPnode;
            end;
var
```

```
    index :array['a'..'z'] of TPnode;
    N,i   :word;
    NewName   :string[20];
    c         :char;
    temp      :TPNode;
begin
  assign(input,'3.in'); reset(input);
   for c:='a' to 'z' do  {初始化 26 个表头节点}
     begin  new(index[c]); index[c]^.next:=nil; end;
   readln(N);
   for i:=1 to N do
   begin
       readln(NewName);
       c:=NewName[1];
       new(temp); temp^.name:=NewName;
       temp^.next:=index[c]^.next;  { 插入一个节点到表头 }
       index[c]^.next:=temp;
   end;
   for c:='a'to'z'do  { 打印 26 个表 }
   begin
     while index[c]^.next<>nil do
     begin  write(index[c]^.next^.name,'  ');
            index[c]^.next:=index[c]^.next^.next;
     end;
     if index[c]^.next<>nil then      writeln;
   end;
   close(input);
   end.
```

2. 循环链表的简单实现

循环链表的指针实现非常简单,只要将最后一个节点的指针域取头节点的指针域即可,如图 1-4 所示。

其双向链表也很容易实现,只要每个节点再加一个指向前的指针域。这里就不详述了,大家可以自己做一做。

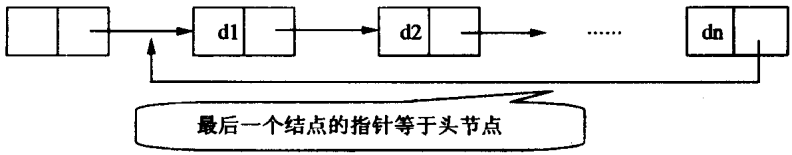


图 1-4

3. 用数组下标模拟指针的方法

一个指针占用 4 个字节，比较浪费，并且在 TP7.0 环境下调试也不方便。在知道数据空间不太大，且指针链表又比较方便时，我们可以用数组来存贮所有结点，利用数组下标来模拟指针，从而实现线性表的功能。

例 1-4 猴子选大王新解。

在例 1-2 中，退出一只猴子，为了保证数组的连续性，要移动很多元素。下面用数组下标模拟指针的方法实现，可以避免这种情况。程序如下：

```
const
  maxN=10000;
var
  monkey:array[1..maxN]of  record
                             No, next  :integer;
                           end;
  n,p,m,i  :integer;
begin
  readln(n,p,m);
  for i:=1 to n do
    begin monkey[i].No:=i; monkey[i].next:=i+1; end;
  monkey[n].next:=1;
  p:=p-1; if p=0 then p:=n;

  while n>1 do
  begin
    for i:=1 to (m-1) mod n  do
      p:=monkey[p].next;
```

```
        monkey[p].next:=monkey[monkey[p].next].next;
        n:=n-1;
      end;
      writeln('The king of Monkey is ',p);
end.
```

注 例 1-4 的解法虽然比例 1-2 的解法好,但也是比较简单的。当数据规模比较大时有速度问题,更好的算法参见后面的数据结构的综合运用。

三、表的基本操作

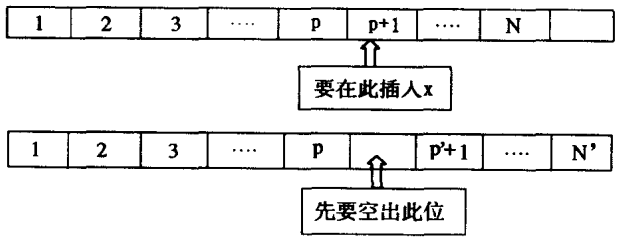
从上面两节表的实现例子中,我们已经看到了很多线性表的基本操作。线性数据结构的基本操作有:插入、删除、查找,下面分别一一介绍。

1. 插入运算 Insert(x,p,L)

将元素 x 插入到表 L 中位置为 p 的元素后面。

(1)对于数组实现,算法大致为:

```
for i:=N downto p+1 do  data[i+1]:=data[i];
data[p]:=x;
N:=N+1;  { N为表的长度 }
```

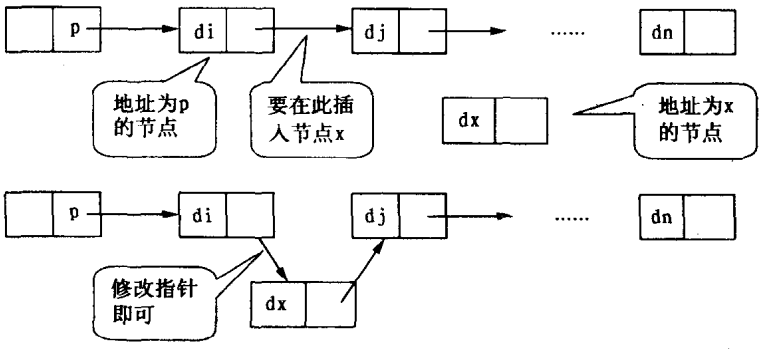


算法示意图

图 1-5

(2)对于指针实现的算法大致为:

```
x^.next:=p^.next;
p^.next:=x;
N:=N+1;
```



算法示意图

图 1-6

2. 删除运算 Delete(p,L)

将表 L 中位置为 p 的元素删除掉。

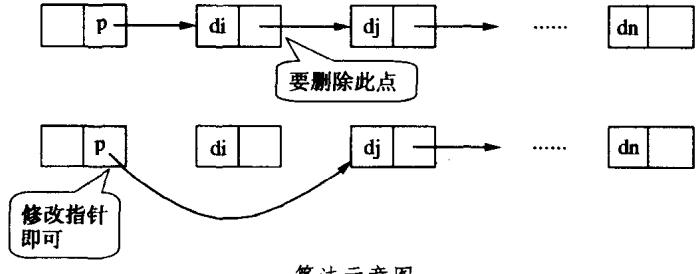
(1)对于数组实现,算法大致为:

```
for i:=p+1 to N do
    data[i-1]:=data[i];
N:=N-1;
```

算法说明 为了保持数组的连续性,必须将删除的元素补上,但又要保留原先的关系(否则,只要把最后一个调过来即可),只好移动 N-p 个元素。

(2)对于指针实现,则算法大致为:

```
temp:=p^.next; p^.next:=p^.next^.next; dispose(temp);
N:=N-1;
```



算法示意图

图 1-7

3. 查找运算

查找运算有两种，一种是找表中的第 K 个元素，另一种是查找值为 x 的元素。我们把这两种运算定义为：Find(k,L)和 Locate(x,L)。

(1)对于数组实现，算法大致为：

```
function  Find(k,L):TypeData;
 begin
      Find:=data[k];
 end;

Fuinction Locate(x,L):TypeData;
begin
  for i:=1 to N do
    if data[i]:=x then begin Locate:=i; exit; end;
  Locate:=0;  {表示没有查找到}
end;
```

(2)对于指针实现，算法大致为：

```
function  Find(k,L):TypeData;
 begin
    p:=L.head;     {取表头}
    while k>1 do begin
       p:=p^.next;
       k:=k-1;
       end;
       Find:=P^.next;
end;

Function Locate(x,L):TypeData;
begin
     p:=L.head;     {取表头}
     while (p^.next<> nil ) and (p^.next^.data<>x) do p:=p^.next;
  Locate:=p^.next;
end;
```

总结 下面是各种运算的平均时间复杂度表：

表 1-1

	插入运算	删除运算	查找
数组实现	O(N)	O(N)	O(1),O(N)
指针实现	O(1)	O(1)	O(N),O(N)

可以看出，数组实现在时间复杂度上只有一项——Find(x,L)——比指针实现好，但这并不说明指针实现就好。原因有下面两点：

(1)数组实现编程比较方便，不易错，调试也容易。对同样多的操作，其时间复杂度的系数比较小，速度更快。

(2)插入运算和删除运算的前提常常是先用 Locate(x,L)操作定位，这时指针实现插入运算和删除运算总的复杂度也成了O(N)，因此，多数情况下不占优。

所以说，在编程中到底用什么样的数据结构及怎样实现都要具体情况具体分析。灵活应用的前提是对基本数据结构有全面的了解，并在平时的解题当中多分析、多思考具体问题的需要。下面就介绍两个在一定情况下常用的特殊线性数据结构。

四、两种特殊表——队列和栈

1. 先进先出的队列

如果规定，只能在表尾插入元素，在表头删除元素，则这样的表我们称为队列。显然，队列具有先进先出(First In First Out)的性质，有时简称为 FIFO 表。另外，为了操作方便，增加两个变量：队头、队尾。

用数组实现队列自然很方便，但是如果不断的有数据进、出，可能会导致数组很长，甚至有存储空间不够的问题。如果知道了队列的最长长度，可以用循环数组的技巧来实现，具体的实现请看下面例子：