算法竞赛宝典

第三部
基础数据结构

张新华 编著

3

清华大学出版社

算法竞赛宝典

基础数据结构

张新华 编著

清华大学出版社
北 京

图书在版编目(CIP)数据

基础数据结构/张新华编著. --北京：清华大学出版社，2016
(算法竞赛宝典)
ISBN 978-7-302-40950-2

Ⅰ. ①基…　Ⅱ. ①张…　Ⅲ. ①程序设计－数据结构－数学参考资料　Ⅳ. ①TP311.12

中国版本图书馆 CIP 数据核字(2015)第 166259 号

责任编辑：纪海虹　刘美玉
封面设计：张新华
责任校对：王荣静
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京密云胶印厂
经　　销：全国新华书店
开　　本：185mm×260mm　　印　张：17.75　　字　　数：427 千字
版　　次：2016 年 4 月第 1 版　　印　　次：2016 年 4 月第 1 次印刷
印　　数：1～4000
定　　价：42.00 元

产品编号：062996-01

FOREWORD

<<< 前言

为什么编写这套书

随着计算机逐步深入人类生活的各个方面，利用计算机及其程序设计来分析、解决问题的算法在计算机科学乃至整个科学界的作用日益明显。相应地，各类以算法为主的程序设计竞赛也层出不穷：在国内，有全国青少年信息学奥林匹克联赛（National Olympiad in Informatics in Provinces，NOIP），该联赛与全国中学生生物学联赛、全国中学生物理竞赛、全国高中数学联赛、全国高中学生化学竞赛，并称为国内影响力最大的“五大奥赛”；在国际，有中学生的国际信息学奥林匹克竞赛（International Olympiad in Informatics，IOI）、面向亚太地区在校中学生的信息学科竞赛（即亚洲与太平洋地区信息学奥林匹克，Asia-Pacific Informatics Olympiad，APIO）和国际大学生程序设计竞赛（ACM International Collegiate Programming Contest，ACM/ICPC）等。

各类算法竞赛要求参赛选手不仅必须具有深厚的计算机算法功底、快速、准确的编程能力以及创造性的思维，而且还必须具备团队合作精神和抗压工作的能力，因此算法竞赛在高校、IT 公司和其他社会各界中获得越来越多的认同和重视。算法竞赛的优胜者，更是微软、Google、百度、Facebook 等全球著名 IT 公司争相高薪招募的对象。因此，不仅是各类参加算法竞赛的选手，即使是不参加此类竞赛的很多研究工作者和从事 IT 行业的人士，都希望能获得这方面的专业训练并从中得到一定的收获。

但是，长期以来，算法竞赛的内容以其高难度、高要求而困住了许多有志于参加算法竞赛的爱好者的求索脚步。部分已出版的同类书籍或艰深难懂、或教条灌输，使不少爱好者望而却步，半途而废，以至于研究算法成了一种“曲高和寡”的学问，严重制约了算法竞赛优秀人才的培养。

殊不知，书上多数看似复杂难懂的知识或内容，其实总是可以找到一种深入浅出、化难为易的表述方法，使读者能看得懂、学得透，并最终达到触类旁通、举一反三的学习目标。因此本套丛书的写作初衷，是试图将看似晦涩难懂的算法表述得轻松、活泼、易懂。同时也是为了弥补国内算法竞赛以及普及读物方面的不足。

希望本套丛书的出版，能够给那些学有余力的中学生、计算机专业的大学生、程序算法爱好者以及 IT 从业者提供一个学习计算机科学的机会。

为什么要学习算法

经常有人说：我不学算法也照样可以编程写软件。那么，为什么还要学习算法呢？

首先，算法（algorithm）一词源于算术（algorism），具体地说，算术方法是一个由已知推

求未知的运算过程。后来，人们把它推广到一般，即把进行某一工作的方法和步骤称为算法。一个程序要完成一个任务，其背后肯定要涉及算法的实现，算法的优劣，直接决定了程序的优劣，因此，算法是程序的灵魂。学好了算法，就能够设计出更加有效的软件，以最为有效的方式完成复杂的功能。例如，设计完成一个具有较强人工智能的人机对弈棋类游戏，程序员没有深厚的算法功底是根本不可能实现的。

其次，算法是对事物本质的数学抽象，是初等数学、高等数学、线性代数、计算几何、离散数学、概率论、数理统计和计算方法等知识的具体运用。真正学通计算机的人（不只是"编程匠"）都对数学有相当的造诣，既能用科学家的严谨思维来求证，也能用工程师的务实手段来解决问题——而这种思维和手段的最佳演绎就是"算法"。学习算法，能锻炼我们的思维，使思维变得更清晰、更有逻辑，变得更有深度和广度。学习算法更是培养逻辑推理能力的非常好的载体之一。因此，学会算法思想，其意义不仅仅在算法本身，更重要的是，对日后的学习生活和思维方式也会产生深远的影响。

最后，学习算法本身很有意思、很有趣味。所谓"技术做到极致就是艺术"，当一个人真的沉浸到算法研究中时，他会感受到一个个精妙绝伦的算法的艺术之美，他会为它惊人的运行速度和巧夺天工的构思而深深震撼，并从中体会到一种不可言喻的美感和愉悦。当然，算法的那份优雅与精巧虽然吸引人，却也令很多人望而生畏。事实证明，对很多人来说，学习算法是一件很痛苦的事情。

本套丛书的特色

本套丛书的第一个特点是具备一定理解能力的读者可以凭自学看懂书中的多数内容。由于地区的不平衡性等原因，很多算法竞赛选手及算法爱好者面对的一个最大难题是没有合适的教材以及找不到合适的导师给予引导。考虑到这种情况，本书作者尽可能地对书中每一个难点和重点都进行了深入仔细的讲解，此外，书中的绝大多数题目均附有带注释的完整源代码及测试数据供读者参考和检测。当然，不管怎样，你永远也不要指望阅读一本算法书能像阅读一本通俗小说那般轻松惬意。

本套丛书的第二个特点是摒弃了一些算法书过于"学究气"，片面强调学术性、严谨性而忽略了可读性的"四平八稳"的表述手法，代之以轻松、活泼的故事情节和穿插以历史、文化、科技等要素的情境导入，读者不仅在学习过程中获得编程的乐趣，还能够跟随书中的众多人物，一同经历一场奇妙的冒险之旅。本书作者热切希望读者不仅最终能够在计算机科学的领域大展宏图，更能够在故事中获得感悟，对社会、人生、理想、价值和信念等有更深刻的思考和认识。

本套丛书的第三个特点是绝大多数题目均采用了"多向思考""一题多解"和"一题多变"的解决方法，其目的主要有三点：一是为了充分调动读者思维的积极性，提高综合运用已学知识解答问题的技能技巧；二是为了锻炼读者思维的灵活性，促进知识和智慧的增长；三是训练思维深度和广度，引导读者灵活地掌握知识的纵横联系，培养和发挥读者的创造性。所以，这种训练，决不能简单地看作是编程技巧的花哨卖弄，而是通过这种训练，培养读者思维的敏捷性，促进读者智力和思维的发展，提高读者的变通能力与综合运用所学知识的能力。读者若能坚持这种思维训练方法，必能起到意想不到的良好学习效果。

本套丛书内容简介

本套丛书的第一部——《语言及算法入门》，主要介绍在算法竞赛中需要用到的 C++语

言的语法知识及一些简单算法的运用。但与一般C++语言入门书不同的是，本部书在介绍C++语言的同时，更加侧重于数学思维的培养和简单算法的应用，因此其学习难度远高于一般市面上的C++语言入门书。书中很多表面看上去似乎非常简单的题目，由于采取了“一题多解”及“数学求解”等方法，其程序复杂度直线上升。因此这就要求读者具备一定的数学功底和思维能力，并且需要花费相当长的时间去思考和练习，才可能深刻理解题目的本质和内涵。此外，为了防止初学者原封不动地照抄源代码而不理解代码真实含义的“恶习”，书中凡可写可不写源代码的地方，一律采用伪代码形式表述，当然，在出版社网站上，读者可以下载到书中所有的源代码及测试数据等资源。

本套丛书的第二部——《基础算法艺术》，重点介绍了各种基础算法，如分治算法、贪心算法、枚举算法、动态规划算法等。书中的绝大多数题目都采用了“多向思考”、“一题多解”和“一题多变”的方式来解决。读者不仅可以通过出版社网站下载的简单测试数据验证所写程序的正确性，还可以根据书中标注的题目原始出处，访问相关的在线评测网站提交所写代码进行测试。需要注意的是，与多数教材从头读到尾的习惯不同的是，本书的各章节划分并不是严格按从难到易、从简到繁的顺序编排，各章节基本都可以独立阅读，互不影响。读者在阅读过程中，有时可根据需要翻到其他章节学习完相关内容后再返回本章节继续学习。此外，每一道题的各种算法按难易程度粗略地以★号表示，★号越多，表示难度越大，读者若遇到难度过大的题，不必硬“啃”，完全可以在学习完其他章节的相关内容后再来学习。

本套丛书的第三部——《基础数据结构》，详细介绍了链表、堆栈、队列、树、图等基础数据结构的相关知识。为了便于读者的理解，本书对数据结构众多知识点进行了详细的解释和分析，随书还配有难易适中的练习题。本书中的多数题目未配置相应测试数据，读者编写的代码正确与否，需要去相关的在线评测网站提交代码进行测试。这样做是培养读者善于应用无限网络资源的能力，使读者能逐渐脱离书本的束缚，最终达到独立、自主学习的目的。

全套书中以调侃化的形式出现的公司和人名等，均为故事情节而服务，与现实无更多关联。

适合阅读本套丛书的读者

本套丛书可作为全国青少年信息学奥林匹克联赛（NOIP）系列的复赛教材及ACM国际大学生程序设计竞赛（ACM/ICPC）的参考和学习用书，还可作为计算机系学生、IT工程师、科研人员、算法爱好者的参考和学习用书。

致谢

感谢原乌鲁木齐高级中学（前身为乌鲁木齐市第六中学）的李念、刘悠然、朱生龙、陆智、李智杰、刘毅东、吕亚伟、徐兆鹏、戴文轩、杜同心、胡亚欣、徐菲鹏、戴维、张欢、郑斐、彭胡杰、肖阳等同学；感谢浙江省瑞安中学钱瑞源、潘伟达、张元煌、周望威、林展、杨丰、张颖洁、卢众意、沈小洲、肖煜伟、郑立言、金泽豪、章奎亮、蔡子豪、陈云帆、方佳豪、陈士俊、李心怡、孙一言、蔡凡、黄海同、王络、肖瑞宇、潘洲阳、陈可、张夏悦、陈振哲、林江浩、魏书扬等同学，他们在笔者的指导下，用简洁易懂的程序代码实现了本书的部分算法。

感谢全国各省市中学、大学的信息学奥赛指导老师们，他们给本套丛书提了许多真诚而有益的建议，并对本人在写书过程中遇到的一些困惑和问题给予了热心的解答。

感谢浙江省瑞安中学为本人搭建的学习、研究、竞赛的平台以及对本人工作的大力支持。

本套丛书在完成过程中，使用了 NOIP 的部分原题、在线评测网站的部分题目，并参考和收集了其他创作者发表在互联网、杂志等媒体上的相关资料，无法一一列举，在此一并表示衷心感谢。

另外，感谢我的女儿，你是我完成这套书的动力，因为这套书专为你而写。

最后要说的话

本套丛书花费了笔者巨大的精力和时间，这些年来的多数闲暇时间，都用在了这套丛书的编写和反复校验上。但尽管如此，由于时间和水平所限，书中难免存在不妥和错误之处，欢迎同仁或读者赐正，如果在阅读中发现任何问题，请发送电子邮件到 hiapollo@sohu.com 告诉本人，更希望读者对本书提出建设性意见，以便修订再版时改进。

针对本套书的题库网站正在不断完善中，网址：www.razxhoi.com。

张新华

2014 年 8 月于浙江瑞安

CONTENTS

<<< 目录

第一章

链　表

何谓链表

我们知道，一般用数组存放一组数据时，必须事先定义固定的长度(即元素个数)。这在某些问题的解决中，并不是特别的适用。例如，记录不同班级的学生数据时，由于各班人数不同，会出现开辟过大的数组导致内存浪费、开辟过小的数组导致数组元素不够用的情况。而链表可以根据需要动态开辟内存单元，是一种常见的重要数据结构。图 1.1 所示为最简单的一种链表结构。

图　1.1

链表如同铁链一样，一环扣一环，中间是不能断开的。打个通俗的比方：幼儿园老师带领小朋友出来散步，老师牵着第一个小朋友的手，第一个小朋友的手牵着第二个小朋友的手……这就是一个“链”，最后一个小朋友的手是空的。

老师即“头指针”变量，图 1.1 中以 Head 表示，它存放一个地址。链表中每一个元素称为“结点”，每个结点都应该包括两部分：一为实际元素值，一为下一结点的地址。

最后一个元素不指向其他元素，它被称为“表尾”，以“NULL”表示，“NULL”在 C++语言里指向“空地址”。

很显然，这种链表的数据结构，必须要用指针变量才能实现。

简单静态链表

下面的代码是一个简单的静态链表，它由 3 个学生的数据(学号，成绩)的结点组成。请考虑：(1)head 的作用？(2)p 的作用？

```
1   //简单静态链表
2   #include <iostream>
3   using namespace std;
4
5   struct  student
6   {
7     long num;
8     float score;
9     struct student *next;                    //该指针指向 student 类型的结构体
10  };                                         //注意必须有分号
11
12  int main()
13  {
14    struct student a,b,c, *head, *p;
15    a.num=34341;  a.score=81.5;              //赋值
16    b.num=34343;  b.score=97;
17    c.num=34344;  c.score=82;
18    head=&a;                                 //将 a 的地址给 head
19    a.next=&b;
20    b.next=&c;
21    c.next=NULL;
22    p=head;
23    do                                       //输出记录
24    {
25        cout<<p->num<<" "<<p->score<<endl;
26        p=p->next;
27    }while(p!=NULL);
28    getchar();
29  }
```

处理动态链表的函数

1. malloc

函数原型为

```
void *malloc(unsigned int size);
```

作用是在内存的动态存储区中分配一个长度为 size 的连续空间。此函数返回的是一个指向分配域起始地址的指针，如果此函数未能成功地执行(如内存空间不足)，则返回空指针(NULL)。

2. calloc

函数原型为

```
void *calloc(unsigned n,unsigned size);
```

作用是在内存的动态存储区中分配 n 个长度为 size 的连续空间。函数返回一个指向分配域起始地址的指针；如果分配不成功，则返回 NULL。

3. free()

函数原型为

```
void free(void *p);
```

作用是释放由 p 指向的内存区，使这部分内存区能被其他变量使用。

动态链表的准备工作

一个完善的动态链表程序应该具有以下基本功能：建立链表、插入结点、删除结点、打印链表、释放链表等。扩展的动态链表程序还可能有获得链表长度、获得当前结点、查找结点位置、连接两个链表、比较两个链表等功能。下面将逐个实现其功能代码。

为了程序的易读性和可扩展性，有时需要在程序开头先进行预定义处理。请务必领会下面代码的用意，否则将影响对以后代码的理解。

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   typedef int ElemType;                //以 ElemType 代表 int 型数据
4   typedef struct List *link;           //以 link 代表链表指针
5   typedef struct List Lnode;           //以 Lnode 代表链表结点
6
7   struct List
8   {
9     ElemType data;                     //此处仅以一个整型变量为例
10    struct List *next;
11  };
```

主函数的建立：下面的主函数只是一个简单调用各功能子函数的示范例子，读者可自行修改和添加代码以完成更复杂的任务。请根据主函数的代码考虑各功能子函数的原型应如何建立。

```
1   int main()
2   {
3     int l;link head1;link head2;
4     head1=create(head1);                          //建立链表 1
5     head2=create(head2);                          //建立链表 2
6     connect(head1,head2);                         //连接两个链表
7     head1=insert(head1,888,5);                    //在位置为 5 处插入元素 888
8     head1=del(head1,3);                           //删除一个结点
9     display(head1);                               //打印链表
10    printf("\n  lenth is %d\n", lenth(head1));    //打印链表长度
11    printf("\n  get is %d\n", get(head1,3));      //获得当前结点值
12    printf("\n  locate 12 is %d", locate(head1,12));  //查找元素 12 所在的位置
13    head=setnull(head);                           //释放链表
14  }
```

链表的建立

由主函数调用 create()函数的方式可知，该函数应该返回一个结点的指针，输入的参数也应该是一个结点指针。参考代码如下。

```
link create(link Head)
{
  ElemType newData;
  link NewPoint;
  /*先建立一个结点*/
  Head=(link)malloc(sizeof(Lnode));
  printf("please input number: \n");
  scanf("%d",&newData);
  Head->data=newData;                    //结点赋值
  Head->next=NULL;                       //结点指向空地址

  while(1)                               //继续添加结点
  {
    NewPoint=(link)malloc(sizeof(Lnode));  //开辟一个结点空间
    if(NewPoint==NULL)                   //如果开辟空间失败,则返回
      break;                             //此判断语句在某些类型的竞赛中用处不大,可忽略
    printf("please input number: input '-1' means exit\n");
    scanf("%d",&newData);
    if (newData==-1)                     //输入-1 则添加结点结束并返回 head
      return Head;
    NewPoint->data=newData;
    NewPoint->next=Head;
    Head=NewPoint;
  }
  return Head;
}
```

链表的显示

该子函数无返回值，输入参数为链表的头指针，请考虑：指针 p 的作用是什么？如果不用指针 p，直接用 Head 这个指针会出现什么后果？

```
void display(link Head)
{
  link p;p=Head;
  if(p==NULL)
    printf("\nList is empty\n");
  else
    do
    {
      printf("%d ",p->data);
```

```
        p=p->next;
    }while(p!=NULL);
}
```

结点的插入

```
link insert(link Head,ElemType x,int i)
{
  link NewPoint,p=Head;
  int j=1;
  NewPoint=(link)malloc(sizeof(Lnode));
  NewPoint->data=x;
  if(i==1)      //如果插入位置为第一个结点时
  {   NewPoint->next=Head;   Head=NewPoint; }
  else
  {
    while(j<i-1 && p->next!=NULL)
    {      p=p->next;    j++;   }
    if(j==i-1)
    {
      NewPoint->next=p->next;
      p->next=NewPoint;
    }
    else   printf("insert is Failure,i is not right!");
  }
  return Head;
}
```

结点的删除

例如下面这个链表：

(A)→(B)→(C)→(D)→(E)→…→NULL

因为链表中唯一能够找到元素的办法是通过它上一个元素的指针，所以如果我们将某个元素直接删除，这个元素所指向的元素和它之后的所有元素都没有办法再找到了，为了解决这个问题，一般采取这样的办法：记录下要删除的元素之前的那个元素，在删除元素之前，把这个元素的指针指向要删除的那个元素指针指向的元素，比如说现在要删除这个链表的元素B，先找到它之前的元素 A，在删除 B 之前将 A 的指针指向 C，然后再删除 B，这样在删除 B 之前，就已经把 B 从链表里面剔了出来，如图 1.2 所示。

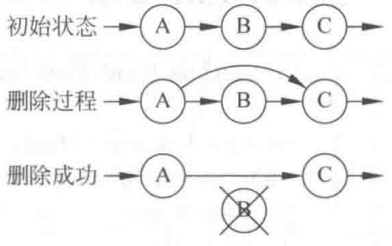


图 1.2

```
link del(link Head,int i)
{
```

```
3     int j=1; link p,t; p=Head;
4     if(i==1)
5     {   p=p->next;   free(Head);   Head=p; }
6     else
7     {
8        while(j<i-1 && p->next !=NULL)
9        {  p=p->next;       j++;     }
10       if(p->next!=NULL && j==i-1)
11          {  t=p->next;       p->next=t->next;     }
12       if(t!=NULL)   free(t);
13    }
14    return Head;
15 }
```

上面的例子中操作符 free 用来删除一个变量，即在内存中释放某个变量所对应的地址，使得之后系统可以利用这块内存做别的事情，这样比较节省内存空间。我们应养成及时释放无用的内存空间的好习惯。

获得结点元素值

```
1  ElemType get(link Head,int i)
2  {
3     int j=1;
4     link p; p=Head;
5     while(j<i && p!=NULL)
6     {     p=p->next;     j++;   }
7     if(p!=NULL)
8          return(p->data);
9     else
10         printf("data is error!");
11         return -1;
12 }
```

查找结点元素 X 的位置

```
1  int locate(link Head,ElemType x)
2  {
3     int n=0;link p;p=Head;
4     while(p!=NULL && p->data !=x)
5     {  p=p->next;      n++;   }
6     if(p==NULL)
7          return -1;
8     else
9          return n+1;
10 }
```

返回链表的长度

```
int lenth(link Head)
{ int len=0;   link p;   p=Head;
  while(p!=NULL)
  {      len++;      p=p->next;   }
  return len;
}
```

连接两个链表

```
link connect(link Head1,link Head2)
{
    link p; p=Head1;
    while(p->next !=NULL)
    { p=p->next;   }
    p->next=Head2;
    return Head1;
}
```

比较两个链表是否相同

```
int compare(link Head1,link Head2)
{
    link p1,p2; p1=Head1; p2=Head2;
    while(1)
    {
        if((p1->next==NULL) &&(p2->next==NULL))
            return 1;
        if(p1->data!=p2->data)
            return 0;
        else
        {      p1=p1->next;      p2=p2->next;    }
    }
}
```

释放链表

```
link setnull(link Head)
{
    link p;p=Head;
    while(p!=NULL)
    {
        p=p->next;
```

```
        free(Head);
        Head=p;
    }
    return Head;
}
```

完整的链表程序

参考代码如下。

```
#include <stdlib.h>
#include <stdio.h>
typedef int ElemType;
typedef struct List *link;
typedef struct List Lnode;

struct List
{
  ElemType data;
  struct List *next;
};

link setnull(link Head)
{
  link p;p=Head;
  while(p!=NULL)
  {
     p=p->next;
     free(Head);
     Head=p;
  }
  return Head;
}

link insert(link Head,ElemType x,int i)
{
  link NewPoint,p=Head;
  int j=1;
  NewPoint=(link)malloc(sizeof(Lnode));
  NewPoint->data=x;
  if(i==1)
  {
     NewPoint->next=Head;
     Head=NewPoint;
  }
  else
  {
     while(j<i-1 && p->next!=NULL)
     {
```

```
40          p=p->next;
41          j++;
42        }
43        if(j==i-1)
44        {
45          NewPoint->next=p->next;
46          p->next=NewPoint;
47        }
48        else  printf("insert is Failure,i is not right!");
49      }
50      return Head;
51    }
52
53    link create(link Head)
54    {
55      ElemType newData;
56      link NewPoint;
57
58      Head=(link)malloc(sizeof(Lnode));
59      printf("please input number: \n");
60      scanf("%d",&newData);
61      Head->data=newData;
62      Head->next=NULL;
63
64      while(1)
65      {
66        NewPoint=(link)malloc(sizeof(Lnode));
67        if(NewPoint==NULL)
68          break;
69        printf("please input number: input '-1' means exit\n");
70        scanf("%d",&newData);
71        if (newData==-1)
72          return Head;
73        NewPoint->data=newData;
74        NewPoint->next=Head;
75        Head=NewPoint;
76      }
77      return Head;
78    }
79
80    int lenth(link Head)
81    {
82      int len=0;
83      link p;
84      p=Head;
85      while(p!=NULL)
86      {
87        len++;
88        p=p->next;
89      }
```

```
90      return len;
91  }
92
93  ElemType get(link Head,int i)
94  {
95      int j=1;
96      link p; p=Head;
97      while(j<i && p!=NULL)
98      {
99          p=p->next;
100         j++;
101     }
102     if(p!=NULL)
103         return(p->data);
104     else
105         printf("data is error!");
106     return -1;
107 }
108
109 int locate(link Head,ElemType x)
110 {
111     int n=0;link p;p=Head;
112     while(p!=NULL && p->data !=x)
113     {
114         p=p->next;
115         n++;
116     }
117     if(p==NULL)
118         return -1;
119     else
120         return n+1;
121 }
122
123 void display(link Head)
124 {
125     link p;p=Head;
126     if(p==NULL)
127         printf("\nList is empty\n");
128     else
129         do
130         {
131             printf("%d ",p->data);
132             p=p->next;
133         }while(p!=NULL);
134 }
135
136 link connect(link Head1,link Head2)
137 {
138     link p;
139     p=Head1;
```

```
140     while(p->next !=NULL)
141     {
142       p=p->next;
143     }
144     p->next=Head2;
145
146     return Head1;
147   }
148
149   link del(link Head,int i)
150   {
151     int j=1;
152     link p,t;
153     p=Head;
154     if(i==1)
155     {
156       p=p->next;
157       free(Head);
158       Head=p;
159     }
160     else
161     {
162       while(j<i-1 && p->next !=NULL)
163       {
164         p=p->next;
165         j++;
166       }
167       if(p->next!=NULL && j==i-1)
168       {
169         t=p->next;
170         p->next=t->next;
171       }
172
173       if(t!=NULL)
174         free(t);
175     }
176     return Head;
177   }
178
179   int compare(link Head1,link Head2)
180   {
181     link p1,p2;
182     p1=Head1;
183     p2=Head2;
184     while(1)
185     {
186       if((p1->next==NULL) &&(p2->next==NULL))
187         return 1;
188       if(p1->data!=p2->data)
189         return 0;
```

```
190        else
191        {
192            p1=p1->next;
193            p2=p2->next;
194        }
195    }
196 }
197
198 int main()
199 {
200     int l;
201     link head1;link head2;
202
203     head1=create(head1);
204     printf("\nHead1 is\n");
205     display(head1);
206
207     head2=create(head2);
208     printf("\nHead2 is\n");
209     display(head2);
210
211     l=compare(head1,head2);
212     printf("\nl is %d\n",l);
213
214     connect(head1,head2);
215     printf("\nHead1+Head2 is\n");
216     display(head1);
217
218     l=lenth(head1);
219     printf("\nlenth is %d\n",l);
220
221     l=get(head1,3);
222     printf("\n get is %d\n",l);
223
224     l=locate(head1,12);
225     printf("\n locate 12 is %d",l);
226
227     head1=insert(head1,888,5);
228     display(head1);
229
230     head1=del(head1,5);
231     display(head1);
232
233     head1=setnull(head1);
234     display(head1);
235 }
```

数组仿真链表

实际上，可以使用数组仿真链表的各项功能，当然数组仿真链表不具备链表动态开辟内存空间的特点。参考代码如下。

```
1   //数组仿真链表的演示程序
2   #include<cstdio>
3   #define MAXN 100001                    //链表中最多能容纳的元素数+1
4   using namespace std;
5
6   int linklst_data[MAXN] ;               //linklst_data 为链表中元素的数据
7   int linklst_point[MAXN] ;              //linklst_point 为链表中元素的指针
8   int head = -1;                         /*链表的头指针*/
9
10  void del_by_data(int del_data)         //删除一个含有 del_data 的元素
11  {
12    int p=head,pre=-1;                   //从头开始遍历链表
13    while(p != -1)
14    {
15      if(linklst_data[p] == del_data)    //如找到了要删除的值,则进行删除操作
16      {
17        if(p == head)                    //如果删除的是头指针,则更新头指针
18          head = linklst_point[head];
19        else     //这个 else 等同于 if (pre != 1)因为当 p!=head 时 p 前面已经有元素了
20          linklst_point[pre] = linklst_point[p];
21        linklst_data[p] = -1;            //删除 p 指向的元素的值*/
22        linklst_point[p] = -1;
23        return ;                         /*结束删除操作*/
24      }
25        pre = p;
26        p = linklst_point[p];
27    }
28  }
29
30  void add_front(int add_data)           //在链表前段加入元素
31  {
32    int p = 1;
33    while(linklst_data[p] != -1 && p < MAXN)  //找一空位存储数据,这里可以优化
34      ++p;
35    linklst_data[p] = add_data;          //将要加入的结点选好的空位赋值
36    linklst_point[p] = head;             //将当前加入元素的指针指向 head
37    head = p;                            //使当前的元素成为链表头指针
38  }
39
40  void add_rear(int add_data)            //在链表末尾加入元素
41  {
42    int p = 1,pre;
43    while(linklst_data[p] != -1 && p < MAXN )  //找到空位
```

```
44        ++p;
45      linklst_data[p] = add_data;              //该空位赋值
46      if( head != -1 )                         //链表不为空
47      {
48        pre = head;                            //找到链表中的最后一个元素 */
49        while(linklst_point[pre] != -1)        //找到空指针
50          pre = linklst_point[pre];
51        linklst_point[pre] = p;          //将当前链表中最后一个元素的指针指向要加入的元素
52      }
53      else                                     //否则直接对 head 所指的元素赋值
54        head = p;
55      return ;
56    }
57
58    void output()
59    {
60      int p=head;
61      cout<<"list is: ";
62      while(p != -1)
63      {
64        cout<<linklst_data[p]<<" ";
65        p = linklst_point[p];
66      }
67      cout<<"\n";
68    }
69
70    void init()                              //初始化数组指针值
71    {
72      for(int i = 0;i < MAXN;i++)              //链表中的空值设定为-1
73      {
74        linklst_point[i] = -1;
75        linklst_data[i] = -1;
76      }
77    }
78
79    int main()                               //演示链表的操作
80    {
81      int ins,data;
82      init();
83      while(1)
84      {
85        cout<<"1.insert a value in front \n";
86        cout<<"2.insert a value in rear \n";
87        cout<<"3.delete a value \n";
88        cout<<"4.quit \n";
89        cin>>ins;
90        switch(ins)
91        {
92          case 1:
93            cout<<"please insert a value :";
```

```
94              cin>>data;
95              add_front(data);
96              break;
97          case 2:
98              cout<<"please insert a value :";
99              cin>>data;
100             add_rear(data);
101             break;
102         case 3:
103             cout<<"please insert a value :";
104             cin>>data;
105             del_by_data(data);
106             break;
107         default:
108             return 0;
109     }
110     system("cls");                          //清空屏幕,此命令只能在 Windows 平台下使用
111     output();
112 }
113 return 0;
114 }
```

数组仿真链表的优化

在上面的程序中,每次加入一个新值都要在数组当中寻找一个空位来储存数据,这样做会有一个问题,就是当链表中加入了 n 个元素时,每一次都会用线性的时间复杂度去寻找空位,造成了很大的时间上的浪费,因此需要进行优化(本节内容涉及堆栈的应用,请在学完本书第二章的内容后再研究本节内容)。

用一个堆栈记录所有没有被加入到链表中的数组元素下标,栈的元素可以初始化为 1…MAXN−1,以方便使用。每次对链表进行删除操作时,再将删除的数组元素所对应的数组下标入栈,下次再添加新元素进链表时,只需取出栈顶的数组下标即可,因为这个下标对应的数组元素显然是空的。

这个优化方法将 O(n)寻找空位的过程直接变成了 O(1)。

```
1   //数组链表的演示程序(栈优化)
2   #include<iostream>
3   #define MAXN 100001
4   using namespace std;
5
6   int linklst_data[MAXN] ;
7   int linklst_point[MAXN] ;
8   int stack[MAXN];                    //用于记录没有加入链表的数组下标的栈
9   int head = -1;                      //链表的头指针
10  int stack_head = 0;                 //栈顶指针
11
12  void del_by_data(int del_data)      //删除一个含有 del_data 的元素
```

```
13  {
14    int p=head,pre=-1;                    //从头开始遍历链表
15    while(p != -1)
16    {
17      if(linklst_data[p] == del_data)     //找到要删除的值,进行删除操作
18      {
19
20        if(p == head)                     //如果删除的是头指针,则更新头指针
21          head = linklst_point[head];
22        else     //这个 else 等同于 if (pre != 1),因为当 p!=head 时 p 前面已经有元素了
23          linklst_point[pre] = linklst_point[p];
24
25        linklst_data[p] = -1;             //删除 p 指向的元素的值
26        linklst_point[p] = -1;
27        stack[--stack_head] = p;          //将删除元素的数组下标入栈
28        return ;
29      }
30        pre = p;
31        p = linklst_point[p];
32    }
33    return ;
34  }
35
36  void add_front(int add_data)            //在链表前段加入元素
37  {
38    int p = 1;
39    p = stack[stack_head++];              //直接取出栈中的空位
40    linklst_data[p] = add_data;           //将要加入的结点选好的空位赋值
41    linklst_point[p] = head;              //将当前加入元素的指针指向 head
42    head = p;                             //使当前的元素成为链表头指针
43  }
44
45  void add_rear(int add_data)             //在链表末尾加入元素
46  {
47    int p = 1,pre;
48    p = stack[stack_head++];              //直接取出栈中的空位
49    linklst_data[p] = add_data;           //对要加入的结点的空位进行赋值
50    if( head != -1 )                      //链表不为空
51    {
52      pre = head;                         //找到链表中的最后一个元素
53      while(linklst_point[pre] != -1)
54        pre = linklst_point[pre];
55      linklst_point[pre] = p;             //将当前链表中最后一个元素的指针指向要加入的元素
56    }
57    else                                  //否则直接对 head 所指的元素赋值
58      head = p;
59  }
60
61  void output()
62  {
```

```
63      int p=head;
64      while(p != -1)
65      {
66        cout<<linklst_data[p]<<" ";
67        p = linklst_point[p];
68      }
69      cout<<"\n";
70      return ;
71    }
72
73
74    void init()                          //初始化数组指针值
75    {
76      for(int i = 0;i < MAXN;i++)
77      {
78        linklst_point[i] = -1;
79        linklst_data[i] = -1;
80        stack[i] = i + 1;                //从 1 到 MAXN 给栈加入数组下标
81      }
82      return ;
83    }
84
85    int main()                           //演示链表的操作
86    {
87      int ins,data;
88      init();
89      while(1)
90      {
91        cout<<"1.insert a value in front \n";
92        cout<<"2.insert a value in rear \n";
93        cout<<"3.delete a value \n";
94        cout<<"4.quit \n";
95        cin>>ins;
96        switch(ins)
97        {
98          case 1:
99            cout<<"please insert a value :";
100           cin>>data;
101           add_front(data);
102           break;
103         case 2:
104           cout<<"please insert a value :";
105           cin>>data;
106           add_rear(data);
107           break;
108         case 3:
109           cout<<"please insert a value :";
110           cin>>data;
111           del_by_data(data);
112           break;
```

```
113         default:
114             return 0;
115         }
116     system("cls");                    //此命令只能在 Windows 平台下使用
117     output();
118 }
119 return 0;
120 }
```

指针仿真链表

作为比较，使用指针方式重写以上的程序。

```
1   //使用指针模拟链表数据结构
2   #include<iostream>
3   using namespace std;
4
5   struct node
6   {
7     int data;
8     node *next;
9   };
10
11  node *head={NULL};
12
13  void del_by_data(int del_data)        //从链表头向后查找，找到 del_data 并删除
14  {
15    node *p = NULL, *pre = NULL;
16    p = head;
17    while(p != NULL)
18    {
19      if(del_data == p->data)
20      {
21        if(pre != NULL)
22          pre->next = p->next;
23        if(p == head)
24          head = p->next;
25        delete p;
26        return ;
27      }
28      pre = p;
29      p = p->next;
30    }
31    return ;
32  }
33
34  void add_rear(int add_data)           //在链表末尾加入一个元素
35  {
36    if(head != NULL)
```

```
37      {
38        node * add_point, * tmp_point;
39        add_point = new node;
40        add_point->data = add_data;
41        add_point->next = NULL;
42        tmp_point = head;
43        while(tmp_point->next != NULL)
44          tmp_point = tmp_point->next;
45        tmp_point->next = add_point;
46      }
47      else
48      {
49        head = new node;
50        head->data = add_data;
51        head->next = NULL;
52      }
53    }
54
55    void add_front(int add_data)          //在链表前端加入一个元素
56    {
57      if(head != NULL)
58      {
59        node * add_point;
60        add_point = new node;
61        add_point->data = add_data;
62        add_point->next = head;
63        head = add_point;
64      }
65      else
66      {
67        head = new node;
68        head->data = add_data;
69        head->next = NULL;
70      }
71    }
72
73    void output()
74    {
75      node * p;
76      p = head;
77      while(p != NULL)
78      {
79        cout<<p->data<<" ";
80        p = p->next;
81      }
82      cout<<"\n";
83    }
84
85    int main()                            //演示链表的操作
86    {
```

```
87      int ins,data;
88      while(1)
89      {
90        cout<<"1.insert a value in front \n";
91        cout<<"2.insert a value in rear \n";
92        cout<<"3.delete a value \n";
93        cout<<"4.quit \n";
94        cin>>ins;
95        switch(ins)
96         {
97           case 1:
98             cout<<"please insert a value :";
99             cin>>data;
100            add_front(data);
101            break;
102          case 2:
103            cout<<"please insert a value :";
104            cin>>data;
105            add_rear(data);
106            break;
107          case 3:
108            cout<<"please insert a value :";
109            cin>>data;
110            del_by_data(data);
111            break;
112          default:
113            return 0;
114        }
115      system("cls");                    //此命令只能在 Windows 平台下使用
116      output();
117    }
118    return 0;
119  }
```

指针与数组链表的比较

一个常见的编程问题：遍历同样大小的数组和链表，哪个比较快？如果按照某些教科书上的分析方法，你会得出结论，这二者一样快，因为时间复杂度都是 O(n)。但是在实践中，这二者却有极大的差异。通过下面的分析你会发现，其实数组比链表要快很多。

首先介绍一个概念：memory hierarchy（存储层次结构），计算机中存在多种不同的存储器，各存储器的平均存取速度相差很大，如表 1.1 所示。

各级别的存储器速度差异非常大，CPU 寄存器速度是内存速度的 100 倍！这就是为什么 CPU 生产厂家发明了 CPU 缓存的原因。而这个 CPU 缓存，就是数组链表和指针链表区别的关键所在。

CPU 缓存会把一片连续的内存空间读入，因为数组结构是连续的内存地址，所以数组

全部或者部分元素被连续存在 CPU 缓存里面，平均读取每个元素的时间只要 3 个 CPU 时钟周期。而链表的结点是分散在堆空间里面的，这时候 CPU 缓存帮不上忙，只能是去读取内存，平均读取时间需要 100 个 CPU 时钟周期。这样算下来，数组访问的速度比链表快 33 倍！（以上皆为理论数字，具体的数字视 CPU 型号及环境不同而略有差异）

表 1.1

存储器名称	存取速度
CPU 寄存器(CPU Registers)	0～1 个 CPU 时钟周期
CPU L1 缓存(L2 CPU caches)	3 个 CPU 时钟周期
CPU L2 缓存(L2 CPU caches)	10 个 CPU 时钟周期
内存(RAM)	100 个 CPU 时钟周期
硬盘(Disk)	大于 1000000 个 CPU 时钟周期

因此，程序中尽量使用连续的数据结构，这样可以充分发挥 CPU 缓存的威力。这种对缓存友好的算法称为 Cache-oblivious algorithm，有兴趣可以参考相关资料。再举一个简单例子，如表 1.2 所示。

表 1.2

```
//程序 1
#include <iostream>
#define N 1000
using namespace std;

int main()
{
  int a[N];
  int b[100][N];
  int c[N];
  for(int i=0;i<N;i++)
   for(int j=0;j<N;j++)
     for(int k=0;k<100;k++)
       c[i]+=a[k]*b[k][j];
       // B[k][j]非逐行连续读取
  cout<<clock()<<endl;
  system("pause");
}
```

```
//程序 2
#include <iostream>
#define N 1000
using namespace std;

int main()
{
  int a[N];
  int b[100][N];
  int c[N];
  for(int i=0;i<N;i++)
    for(int k=0;k<100;k++)
      for(int j=0;j<N;j++)
        c[i]+=a[k]*b[k][j];
        // B[k][j]为逐行连续读取
  cout<<clock()<<endl;
  system("pause");
}
```

虽然两者执行结果一样，算法复杂度也一样，但是你会发现第二种写法要快很多。

总结一下，各种存储器的速度差异很大，在编程中绝对有必要考虑这个因素。比如，内存速度比硬盘快 1 万倍，所以程序中应该尽量避免频繁的硬盘读写；CPU 缓存比内存快几十倍，在程序中尽量多加利用。

为了验证理论的正确性，可以通过随机化读写的测试来比较两者的差异。

测试数据如下：

data1 100000 次随机插入删除操作，数据在[0,32767]之间；（删除操作较少）

data2 100000 次随机插入删除操作，数据在[0,256]之间；（删除较多）

data3 100000 次随机插入删除操作，数据在[0,32]之间；（删除比较频繁）

data4 1000 次随机插入删除操作，数据在[0,32767]之间；

data5 1000 次随机插入删除操作，数据在[0,256]之间；

data6 1000 次随机插入删除操作，数据在[0,32]之间；

data7 10 次随机插入删除操作，数据在[0,32767]之间；

data8 10 次随机插入删除操作，数据在[0,256]之间；

data9 10 次随机插入删除操作，数据在[0,32]之间；

Cena 测试结果如图 1.3 所示。

指针链表

测试点	评测结果	得分	用时	内存
1	正确	10	16.89s	1788KB
2	正确	10	6.04s	1268KB
3	正确	10	5.15s	1264KB
4	正确	10	0.01s	244KB
5	正确	10	0.01s	236KB
6	正确	10	0.01s	232KB
7	正确	10	0.01s	220KB
8	正确	10	0.01s	220KB
9	正确	10	0.01s	220KB

得分：90 用时：28.14s

数组链表

测试点	评测结果	得分	用时	内存
1	正确	10	16.68s	1000KB
2	正确	10	7.64s	1000KB
3	正确	10	7.46s	1000KB
4	正确	10	0.01s	1000KB
5	正确	10	0.01s	1000KB
6	正确	10	0.01s	1000KB
7	正确	10	0.01s	1000KB
8	正确	10	0.01s	1000KB
9	正确	10	0.01s	1000KB

得分：90 用时：31.84s

数组链表(栈优化)

测试点	评测结果	得分	用时	内存
1	正确	10	10.17s	1392KB
2	正确	10	3.37s	1392KB
3	正确	10	2.98s	1392KB
4	正确	10	0.01s	1392KB
5	正确	10	0.01s	1392KB
6	正确	10	0.01s	1392KB
7	正确	10	0.01s	1392KB
8	正确	10	0.01s	1392KB
9	正确	10	0.01s	1392KB

得分：90 用时：16.58s

图 1.3

以上测试在 Thinkpad T60 ：2.0GHz Dou CPU，1GB RAM windows XP SP3 系统环境下进行。

求两个一元多项式之和

【题目描述】 求两个一元多项式之和

与天顶星人的战争已经结束了十年，这十年间，魔法世界的人们一方面积极进行战后的重建工作；另一方面积极探索上古文明在这个世界留下的遗迹。现已知有一处上古文明遗迹建在地下 100 千米处，进入遗迹的超合金门需要两个人同时插入电子钥匙，每把电子钥匙会随机产生一个一元多项式，只有正确地算出两个一元多项式的和，大门才会打开。

现用链表求两个一元多项式之和。例如，求 $3x^3+2x^2-5x+6$ 与 $-2x^3-2x^2+5x+4$ 之和。

【输入格式】

输入第一个一元多项式，以 0,0 标志结束。

要求：1. 按幂从大到小次序输入各结点的系数和幂

2. 没有两个结点具有相同的幂

第 1 次：3,3

第 2 次：2,2

第 3 次：−5,1

第 4 次：6,0

第 5 次：0,0

输入第二个一元多项式，以 0,0 标志结束。

要求：1. 按幂从大到小次序输入各结点的系数和幂

2. 没有两个结点具有相同的幂

第 1 次：−2,3

第 2 次：−2,2

第 3 次：5,1

第 4 次：4,0

第 5 次：0,0

说明：第一个一元多项式为：3x^3+2x^2−5x+6

第二个一元多项式为：−2x^3−2x^2+5x+4

【输出格式】

相加后一元多项式为：1x^3+10

参考代码如下。

```
//两个一元多项式之和
#include <stdlib.h>
#include <stdio.h>
typedef struct List *Link;
typedef struct List Lnode;

struct List
{
```

```
9     int coef;                                  //系数
10    int expn;                                  //幂
11    struct List *next;
12  };
13
14  Link create()
15  {
16    int a,n,i=1;
17    Link Head,s,p;
18    Head=(Link)malloc(sizeof(Lnode));
19    Head->next=NULL;
20    p=Head;
21    do
22    {
23      printf("\n第 %d 次->系数,幂:",i++);
24      scanf("%d %d",&a,&n);
25      if(a!=0 ||n!=0)
26      {
27        s=(Link)malloc(sizeof(Lnode));
28        s->coef=a;s->expn=n;s->next=NULL;
29        p->next=s;p=s;
30
31      }
32    }while(a!=0 ||n!=0);
33    printf("\n");
34    return(Head->next);
35  }
36
37  Link add(Link pa,Link pb)
38  {
39    int n;
40    Link pc,s,p;
41    pc=(Link)malloc(sizeof(Lnode));
42    pc->next=NULL;
43    p=pc;
44    while(pa!=NULL && pb!=NULL)        //当两个链表都没有到末尾时
45    {
46      if(pa->expn>pb->expn)              //如 a 的幂大于 b 的幂
47      {
48        s=(Link)malloc(sizeof(Lnode));
49        s->coef=pa->coef;s->expn=pa->expn;
50        s->next=NULL;p->next=s;p=s;
51        pa=pa->next;
52      }
53      else if(pa->expn<pb->expn)         //如 a 的幂小于 b 的幂
54      {
55        s=(Link)malloc(sizeof(Lnode));
56        s->coef=pb->coef;s->expn=pb->expn;
57        s->next=NULL;p->next=s;p=s;
58        pb=pb->next;
```

```
59        }
60        else                                    //如 a 的幂等于 b 的幂
61        {
62            n=pa->coef+pb->coef;
63            if(n!=0)
64            {
65                s=(Link)malloc(sizeof(Lnode));
66                s->coef=n;s->expn=pb->expn;s->next=NULL;
67                p->next=s;p=s;
68            }
69            pa=pa->next;pb=pb->next;
70        }
71    }
72    while(pa!=NULL)                             //当 a 还有余项时
73    {
74        s=(Link)malloc(sizeof(Lnode));
75        s->coef=pa->coef;s->expn=pa->expn;
76        s->next=NULL;p->next=s;
77        p=s;pa=pa->next;
78    }
79    while(pb!=NULL)                             //当 b 还有余项时
80    {
81        s=(Link)malloc(sizeof(Lnode));
82        s->coef=pb->coef;s->expn=pb->expn;
83        s->next=NULL;p->next=s;
84        p=s;pb=pb->next;
85    }
86    return(pc->next);
87  }
88
89  void display(Link Head)
90  {
91    int first=1;
92    Link p=Head;
93    while(p!=NULL)
94    {
95        if(first)                               //显示第一个结点时,不显示+号
96        {
97            if(p->expn==1)
98                printf("%dx",p->coef);
99            else if(p->expn==0)
100               printf("%d",p->coef);
101           else
102               printf("%d^%d",p->coef,p->expn);
103           first=0;
104       }
105       else                                    //显示其他结点时,显示+号
106       {
107           if(p->expn==1)
108               printf("+%dx",p->coef);
```

```
109            else if(p->expn==0)
110                printf("+%d",p->coef);
111            else
112                printf("+%dx^%d",p->coef,p->expn);
113        }
114        p=p->next;
115    }
116    printf("\n");
117 }
118
119 int main()
120 {
121    Link Link1,Link2,Link3;
122    printf("\n建立第一个一元多项式(以0,0标志结束):\n");
123    printf("要求:1.按幂从大到小次序输入各结点.\n");
124    printf("      2.没有两个结点具有相同的幂.\n");
125    Link1=create();
126
127    printf("\n建立第二个一元多项式(以0,0标志结束):\n");
128    printf("要求:1.按幂从大到小次序输入各结点.\n");
129    printf("      2.没有两个结点具有相同的幂.\n");
130    Link2=create();
131
132    Link3=add(Link1,Link2);
133    printf("第一个一元多项式为:");
134    display(Link1);
135    printf("第二个一元多项式为:");
136    display(Link2);
137    printf("\n相加后的一元多项式为:");
138    display(Link3);
139    system("pause");
140    return 0;
141 }
```

密钥

【题目描述】 密钥

输入正确的两个一元多项式的和后，上古文明遗迹的超合金门上忽然全息投影出了一圈围坐着的 n 只猴子以及一个整数 m，据说这是设计者受猴子选大王问题的启发而设计的。这 n 只猴子按顺时针方向从 1 到 n 编号。然后从 1 号猴子开始沿顺时针方向从 1 开始报数，报到 m 的猴子出局，再从刚出局猴子的下一个位置重新开始报数，如此重复，直至剩下一只猴子，它的编号就是密钥。

现设计编写一个基于链表的程序，实现如下功能：

(1) 要求由用户输入开始时的猴子数 n、报的最后一个数 m。

(2) 找出最后一只猴子的编号，即密钥。

【算法分析】

我们可以建立具有 n 个结点的循环单链表(链表尾指向链表头,形成一个环),表头结点为 head,然后由指针 q 指向表头结点,开始循环,直到该链表中只有一个结点(q→next==q)为止。在循环过程中,由计数器 count 计数扫描过的结点个数,当 count 累加至 m-1 时,说明下一个结点便是要出队的结点,故删除该结点,这样,链表中的结点个数越来越少,直到只有一个结点,它的编号就是密钥。参考程序如下。

```
1   /*密钥*/
2   #include <stdio.h>
3   struct Node
4   {
5     int data;
6     struct Node *next;
7   };
8
9   int main()
10  {
11    struct Node *head, *s, *q, *t;
12    int n,m,count=0,i;
13    printf("输入猴子个数 n m 号:");
14    scanf("%d,%d",&n,&m);
15    for (i=1;i<=n;i++)
16    {
17      s=(struct Node *)malloc(sizeof(struct Node));      /*建立结点 s*/
18      s->data=i;
19      s->next=NULL;
20      if (i==1)                                          /*建立表头结点*/
21      {
22        head=s,q=head;          /*head 作为表头结点,q 总是指向链表的最后一个结点*/
23      }
24      else                                               /*建立其他结点*/
25      {
26        q->next=s;
27        q=q->next;
28      }
29    }
30    q->next=head;                                        /*建立循环单链表*/
31    printf("出队前:");
32    q=head;                                              /*q 先指向表头结点*/
33    while (q->next!=head)                                /*显示出队前的顺序*/
34    {
35      printf("%d ",q->data);q=q->next;
36    }
37    printf("%d\n",q->data);
38    printf("出队后:");
39    q=head;                                              /*q 先指向表头结点*/
40    do
41    {
42      count++;                                           /*计数器增 1*/
```

```
43        if (count==m-1)                          /*第i个猴子退出*/
44        {
45          t=q->next;                             /*t指向要删除的结点*/
46          q->next=t->next;                       /*删除结点t*/
47          count=0;                               /*计数器重置为0*/
48          printf("%d ",t->data);
49        }
50        q=q->next;
51      } while (q->next!=q);                      /*循环直到只剩一个结点*/
52      printf("%d\n",q->data);                    /*显示最后的一个结点值*/
53      system("pause");
54      return 0;
55  }
```

课后练习

1. 试利用单链表编写一个学生成绩系统，该系统具有查询成绩、修改成绩、删除成绩、添加成绩、计算全班平均成绩等功能。数据如下：

学生座号	学生姓名	语文成绩	英语成绩	数学成绩
6	Mike	86	87	74
15	Rose	70	74	54
17	Bill	95	64	86
21	Peter	64	86	63
23	Jack	78	82	74
34	Helen	92	73	83

2. 请设计一个双向链表。所谓双向链表是指每一个结点都有前后指针，一个指向它的前趋结点，一个指向它的后继结点。很显然，双向链表在处理插入、删除数据等操作时是非常快速和方便的，但双向链表是通过消耗内存的代价来换取速度的。

第二章

堆　　栈

堆栈的定义

何谓堆栈

堆栈(简称栈)是一种最常用和最重要的数据结构,是一种只能在一端进行插入或删除数据操作的线性表。表中允许进行插入、删除操作的一端称为栈顶。栈顶当前位置是动态的,栈顶当前位置由一个称为栈顶指针的位置指示器表示。表的另一端称为栈底。当栈中没有数据元素时,称为空栈。栈的插入操作通常称为进栈或入栈。栈的删除操作通常称为退栈或出栈。

栈的主要特点是"后进先出",即后入栈的元素先弹出。每次进栈的数据元素都放在原当前栈顶元素之上,成为新的栈顶元素,每次出栈的数据都是原当前栈顶元素,如图 2.1 所示。

栈的基本操作

常用的栈操作有以下几个:

初始化栈——init()

入栈——push()

出栈——pop()

取栈顶元素——gettop()

判断栈是否为空——empty()

显示栈元素——display()

释放栈——setnull()

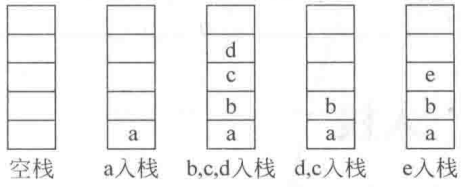


图　2.1

建立堆栈的准备工作

初始化设置

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

```
typedef int ElemType;
typedef struct Stack *link;
typedef struct Stack Snode;

struct Stack
{
  ElemType data;
  struct Stack *next;
};
```

建立主函数

为了简便起见，下面只给出主函数的基本框架，具体功能的实现请参见后面的完整代码。

```
int main()
{
    int i, x;
    link head1;
    head1=init();
    ......
}
```

初始化栈

```
link init()
{
  link p;
  p=NULL;
  return p;
}
```

入栈

```
link push(link Head, ElemType x)
{
  link p;
  p=(link)malloc(sizeof(Snode));
  if(p==NULL)
  {
      printf("\nMemory Error\n");
      return Head;
  }
  p->data=x;
  p->next=Head;
  return p;
}
```

出栈

```
link pop(link Head)
{
 link p;p=Head;
 if(p==NULL)
   printf("\nStack is Empty!\n");
 else
 {
   p=p->next;
   free(Head);
 }
   return p;
}
```

取栈顶元素

```
ElemType gettop(link Head)
{
  if(Head==NULL)
  {
      printf("\nStack is Empty!\n");
      return -1;
  }
  else
      return Head->data;
}
```

判断栈是否为空

```
int empty(link Head)
{
 if(Head==NULL)
    return 1;
 else
    return 0;
}
```

显示栈元素

```
void display(link Head)
{
 link p;p=Head;
 if(p==NULL)
```

```
        printf("\nStack is empty\n");
    else
      do
       {
          printf("%d ",p->data);
          p=p->next;
       }while(p!=NULL);
}
```

释放栈

```
link setnull(link Head)
{
  link p;p=Head;
  while(p!=NULL)
  {
      p=p->next;   free(Head);    Head=p;
  }
  return Head;
}
```

指针仿真堆栈

```
/*堆栈演示!*/
#include <iostream>
using namespace std;
typedef struct Stack *link;
typedef struct Stack Snode;

struct Stack
{
  int data;
  struct Stack *next;
};

link init()                    //初始化栈
{
  link p;
  p=NULL;
  return p;
}

link push(link Head,int x)     //入栈
{
  link p;
  p=new Snode;
  if(p==NULL)
```

```
25      {
26          cout<<"\nMemory Error\n";
27          return Head;
28      }
29      p->data=x;
30      p->next=Head;
31      return p;
32  }
33
34  link pop(link Head)              //出栈
35  {
36      link p;p=Head;
37      if(p==NULL)
38          cout<<"\nStack is Empty!\n";
39      else
40      {
41          p=p->next;
42          delete(Head);
43      }
44      return p;
45  }
46
47  link setnull(link Head)          //释放栈
48  {
49      link p;p=Head;
50      while(p!=NULL)
51      {
52          p=p->next;
53          delete(Head);
54          Head=p;
55      }
56      return Head;
57  }
58
59  int lenth(link Head)             //获得栈内元素个数
60  {
61      int len=0;
62      link p;
63      p=Head;
64      while(p!=NULL)
65      {
66          len++;
67          p=p->next;
68      }
69      return len;
70  }
71
72  int gettop(link Head)            //获得栈顶元素
73  {
74      if(Head==NULL)
```

```
75      {
76        cout<<"\nStack is Empty!\n";
77        return -1;
78      }
79      else
80        return Head->data;
81  }
82
83  void display(link Head)          //显示栈内元素
84  {
85    link p;p=Head;
86    if(p==NULL)
87      cout<<"\nStack is empty\n";
88    else
89      do
90      {
91        cout<<p->data<<" ";
92        p=p->next;
93      }while(p!=NULL);
94  }
95
96  int empty(link Head)             //判断栈是否为空
97  {
98    if(Head==NULL)
99      return 1;
100   else
101     return 0;
102 }
103
104 int main()
105 {
106   int i,x;
107   link head1;
108   head1=init();
109   while(i!=6)
110   {
111     system("cls");
112     cout<<"\n 1.Input a stack data";
113     cout<<"\n 2.Output a stack data";
114     cout<<"\n 3.Empty or Not";
115     cout<<"\n 4.Display a top of stack";
116     cout<<"\n 5.Display the lenth of stack";
117     cout<<"\n 6.Exit and Free Stack\n";
118     cout<<"\n   Stack is:   ";
119     display(head1);
120     cout<<"\n";
121
122     cin>>i;
123     switch(i)
124     {
```

```
125         case 1: while(1)
126                 {
127                  system("cls");
128                   cout<<"\n -. Input a stack data";
129                   cout<<"\n -. Output a stack data";
130                   cout<<"\n -. Empty or Not";
131                   cout<<"\n -. Display a top of stack";
132                   cout<<"\n -. Display the lenth of stack";
133                   cout<<"\n -. Exit and Free Stack\n";
134                   cout<<"\n   Stack is:   ";
135                  display(head1);
136                   cout<<"\n";
137                   cout<<"\ninput a number: until enter -1:\n";
138                   cin>>x;
139                   if(x==-1)
140                      break;
141                   head1=push(head1,x);
142                 }
143                 break;
144       case 2: head1=pop(head1);
145               break;
146       case 3: if(empty(head1))
147                 cout<<"\nStack is empty\n";
148               else
149                 cout<<"\nStack is not empty\n";
150              system("pause");
151               break;
152       case 4: cout<<"\n The top is   "<<gettop(head1)<<endl;
153               system("pause");
154               break;
155       case 5: cout<<"\n The length of stack is "<<lenth(head1)<<endl;
156               system("pause");
157               break;
158     }
159   }
160   system("cls");;
161   head1=setnull(head1);
162   display(head1);
163   getchar();
164   return 0;
165 }
```

数组仿真堆栈

不仅用链表可以仿真堆栈，还可以用数组仿真堆栈。堆栈数组声明如下：

```
int stack[MaxSize];
int top=-1;
```

其中 MaxSize 是该堆栈的最大容量，top 表示当前堆栈顶端的索引值，初始值设为－1表示堆栈为空。

数组仿真堆栈代码实现

```
1  //简单数组模拟栈
2  #include<iostream>
3  #define MAXN 1000                    //栈能容纳的最多元素个数
4  using namespace std;
5
6  int stack[MAXN];
7  int top = -1;                        //初始化栈顶指针为-1
8
9  int pop()                            //栈顶元素出栈并获取出栈的元素值
10 {
11    int temp;
12    if(top<0)
13    {
14       cout<<"\nThe stack is empty!\n";
15       return -1;
16    }
17    temp=stack[top--];
18    return temp;
19 }
20
21 void push(int value)
22 {
23    if(top>=MAXN)
24       cout<<"\nThe stack is full!\n";
25    else
26       stack[++top]=value;
27 }
28
29 void display()                       //显示栈中元素
30 {
31    for(int tmp = top ; tmp >= 0 ; -- tmp)
32       cout<<stack[tmp]<<" ";
33    cout<<"\n";
34 }
35
36 int main()
37 {
38    int ins;
39    while(1)
40    {
41       cout<<"Please enter a value,(0=exit,-1=pop)\n";
42       cin>>ins;
43       if(ins==0)
44          exit(0);
45       else if(ins!=-1)
46          push(ins);
```

```
47        else if(ins==-1)
48            pop();
49        system("cls");
50        display();
51    }
52    return 0;
53 }
```

值得一提的是，用字符串作字符栈也是一种很好的方法。此时的压栈操作，只是将相应字符加在串首(尾)，而出栈操作则是删除串的第(最后)一个字符，完全不用考虑栈顶指针的问题。

数制转换

【题目描述】 十进制转 d 进制(NchangeX. c/cpp/pas)

据说打开上古文明遗迹的大门可以获得失落已久的上古智慧结晶，任何种族若获得此智慧结晶加以应用，不仅能极大地促进自身技术和文明的进步，甚至还有最终傲立于整个宇宙文明之林的可能。所以，进入上古文明遗迹的条件十分苛刻，不仅进入者必须是人类血统，甚至连产生的密钥也经过了处理，具体来说就是获得的密钥是一个十进制数，你需要用堆栈法把它转成 d(2≤d≤36)进制数输入。

【输入格式】

输入文件名 NchangeX. in，为两个整数，即 N 和 d。

【输出格式】

输出文件名 NchangeX. out，为一个整数，即转换的 d 进制数。

【输入样例】

10 2

【输出样例】

1010

```
1  //十进制转 d 进制
2  #include<iostream>
3  #define MAXN 1000              //栈能容纳的最多元素个数
4  using namespace std;
5  char c[36]={'0','1','2','3','4','5','6','7','8','9','A','B',
6              'C','D','E','F','G','H','I','J','K','L','M','N',
7              'O','P','Q','R','S','T','U','V','W','X','Y','Z'};
8  int stack[MAXN];
9  int top = -1;                  //初始化栈顶指针为-1
10
11 int StackEmpty()
12 {
13     if(top<0)
14         return 1;
15     else
```

```
16        return 0;
17    }
18
19    char pop()                          //栈顶元素出栈并获取出栈的元素值
20    {
21      int temp=stack[top--];
22      return c[temp];
23    }
24
25    void push(int value)
26    {
27      stack[++top]=value;
28    }
29
30    void conversion(int N,int d)
31    {
32      while(N)
33      {
34        push(N%d);
35        N=N/d;
36      }
37      while(!StackEmpty())
38        cout<<pop();
39    }
40
41    int main()
42    {
43      int number,d;
44      cin>>number>>d;
45      conversion(number,d);
46      cout<<endl;
47      system("pause");
48      return 0;
49    }
```

判断回文数

【题目描述】 判断回文数(palindrome. c/cpp/pas)

进入上古文明遗迹的第二个陷阱是产生的密钥有可能是一个回文数，如果用户不小心输入这个回文数，就会引发危险，所以你需要判断一个数是否是回文数。

要求：由于输入的一个回文数可能无穷大，所以用单链表存储该数，将用户输入的数以一个单链表的方式存储，从头扫描该单链表，将前面的一半元素入栈，若元素总个数为奇数，则跳过中间的那个元素，然后开始循环：边退栈边在单链表中后移指针，若当前栈顶元素与单链表中当前结点的值域不相等，则退出循环。最后如果栈空且链表比较完毕，则是回文数，否则不是回文数。

【输入格式】

输入文件为 palindrome.in，为一个字符串，即需判断的数。

【输出格式】

输出文件为 palindrome.out，为一行字符串，即某数是回文数或某数不是回文数。

【输入样例】

111122221111

【输出样例】

111122221111 是回文数。

参考代码如下。

```
//判断回文数
#include <iostream>
using namespace std;
#define MAXCHAR 40
struct node
{
  char data;
  struct node *next;
};

int ishs(struct node *head,int n)
{
  char stack[MAXCHAR/2];
  struct node *p=head;
  int top=0;
  while (top<n/2)                    /*将前半部分元素入栈*/
  {
    stack[top]=p->data;
top++;p=p->next;
  }
  if (n%2==1)                        /*n为奇数*/
    p=p->next;
    top--;
  while (top>=0 && p!=NULL && stack[top]==p->data)   /*边退栈边比较*/
  {
    top--;p=p->next;
  }
  if (top==-1 && p==NULL)            /*若栈空且链表比较完毕,则是回文数*/
    return(1);
  else
    return(0);      /*否则不是回文数*/
  }

int main()
{
  char s[MAXCHAR];
  struct node *head=NULL,*p,*q;
```

```
38    int i=0;
39    printf("输入一个数:");
40    scanf("%s",s);
41    while (s[i]!='\0')      /*建立字符串的单链表*/
42    {
43      p=(struct node *)malloc(sizeof(struct node));
44      p->data=s[i];p->next=NULL;
45      if (head==NULL)
46      {
47        head=p;q=p;      /*q总是指向最后一个结点*/
48      }
49      else
50      {
51        q->next=p;q=p;
52      }
53      i++;                /*i为head链表的长度*/
54    }
55    if (ishs(head,i))
56      printf("%s是回文数\n",s);
57    else
58      printf("%s不是回文数\n",s);
59    system("pause");
60  }
```

行编辑程序

【题目描述】 行编辑程序(LineEditor.c/cpp/pas)

为了保证用户的正确输入,上古文明遗迹入口提供了一个简单的行编辑程序,它的功能是:接收用户从终端输入的程序或数据,并存入用户的数据区。由于用户在终端上进行输入时,不能保证不出差错,因此若在编辑程序中,"每接收一个字符即存入用户数据区的做法"显然不是最恰当的。较好的做法是:设立一个输入缓冲区,用以接收用户输入的一行字符,然后逐行存入用户数据区。允许用户输入出差错,并在发现有误时可以及时更正。例如,当用户发现刚刚键入的一个字符是错的时,可以补进一个退格符"#",以表示前一个字符无效;如果发现当前键入的行内差错较多或难以补救,则可以键入一个退行符"@",以表示当前行中的字符均无效。例如,假设从终端接收了这样两行字符:

```
Whil#lr#e(s#*s)
outcha@putchar(*s=#++);
```

则实际有效的是下列两行:

```
While(*s)
putchar(*s++);
```

为此,可设这个输入缓冲区为一个栈结构,每当从终端接收了一个字符之后先作如下判别:如果它既不是退格符也不是退行符,则将该字符压入栈顶;如果是一个退格符,则从栈

顶删去一个字符；如果它是一个退行符，则将字符栈清为空栈。

【输入格式】

输入文件为 LineEditor.in，

【输出格式】

输出文件为 LineEditor.out，为最终的正确字符串。

【输入样例】

Whil#lr#e(s#*s)

【输出样例】

While(*s)

```
1   //行编辑器
2   #include<fstream>
3   #include<string>
4   using namespace std;
5   ifstream cin("LineEditor.in");
6   ofstream cout("LineEditor.out");
7   string s;
8   char a[5005];
9   int p;
10  int main()
11  {
12      int i,len;
13      cin>>s;
14      len=s.size();
15      for(i=0;i<len;++i)
16      {
17          if(s[i]=='@')
18              p=0;
19          else if(s[i]=='#' && p>0)
20              --p;
21          else if(s[i]!='#')
22              a[++p]=s[i];
23      }
24      for(i=1;i<=p;++i)
25          cout<<a[i];
26      cout<<'\n';
27      return 0;
28  }
```

后序表达式

【题目描述】 后序表达式(ReversePolish.c/cpp/pas)

由于进入上古文明遗迹危险重重，所以对探险者的编程能力要求很高，例如，他至少能够实现求一个表达式的值的能力，即用户输入一个包含“+”、“-”、“*”、“/”、正整数和圆括号的合法数学表达式，程序可以计算出该表达式的运算结果。

【输入格式】

输入文件名为 ReversePolish.in，输入一个合法数学表达式，以“#”结束。

【输出格式】

输出文件名为 ReversePolish.out，为该表达式的运算结果。

【输入样例】

(30－2＊5)－3＊8/4#

【输出样例】

14

【算法分析】

一个表达式是由操作数、运算符及分隔符组成的，一般我们使用的是中序表示法，也就是将运算符写在两个操作数之间。例如：

X＋Y

X＋Y＊Z

由于表达式的运算符有优先级，故要用计算机运算中序表达式是很不方便的，所以要在计算机中有效率地计算表达式的值，一般是将中序表达式先转换成较易求值的前序或后序表达式，再进而求算式之值。其中后序表达式又称逆波兰式。

前序表达式是将运算符写在两个操作数之前的，例如：

＋XY

＋X＊YZ

后序表达式是将运算符写在两个操作数之后的，例如：

XY＋

XYZ＊＋

例如，用户在键盘上输入(5-4)/(6＊2)－8＋2 这样的式子。计算机只有通过字符串接收该字符串并根据运算优先级的先后顺序将之转化为前序表达式或后序表达式方可进行正确的运算。

转后序表达式的方法

对表达式中的字符 ch 依次扫描：

1. 若 ch 为数字，将后续的所有数字依次输出；
2. 若 ch 为左括号，将此括号插入栈中；
3. 若 ch 为右括号，将栈中左括号“(”以前的字符依次删除并输出，然后将左括号“(”删除；
4. 若 ch 为＋，－，则将当前栈中左括号“(”以前的所有字符依次删除并输出，然后将 ch 插入栈中(因为优先级低)；
5. 若 ch 为＊ /，则将当前栈中的栈顶端连续的＊ /删除并依次输出，然后将 ch 插入栈中。

例如，将 A/B-C＊(D＋E)转换成后序表达式，结果如表 2.1 所示。

具体转换过程为：

1. 读到 A，直接输出；
2. 读到/，存入堆栈；
3. 读到 B，直接输出；
4. 读到－，因优先权低，则输出堆栈中的运算符再存入堆栈；

表 2.1

后序表达式	堆 栈	后序表达式	堆 栈
A		AB/CDE	－＊(＋
A	/	AB/CDE＋	－＊
AB	/	AB/CDE＋＊－	
AB/	－		

5. 依此处理，直到读到 E；

6. 当读到右括号时，将堆栈中运算符输出，直到取出左括号；

7. 表达式读完，依次输出堆栈中的运算符。

后序表达式的运算

对后序表达式依次扫描，遇到数字存入栈中；遇到运算符，就执行两次删除栈中数的操作并进行运算，将结果压入栈中。重复执行即可。

如后序表达式：AB/CDE＋＊－

1. 堆栈中存入 AB，在读到/时，取出 A 和 B，进行除法运算后将结果存入堆栈；

2. 堆栈中存入 CDE，遇到＋，取出 DE 相加后将结果存入堆栈；

3. 遇到＊，取出 D＋E 的结果，与 C 相乘后存入堆栈；

4. 遇到－，则取出堆栈中仅剩的两个数进行相减得出结果。

本题中假设输入中序表达式中的符号以字符形式由键盘输入，并存放在字符型数组 str 中，其后缀表达式存放在字符型数组 exp 中，在将中序表达式转换成后缀表达式的过程中用一个字符型数组 stack 作为栈。设字符“#”为表达式的终止符。则转换过程为依次从键盘输入表达式中的字符 ch，对于每一个 ch 做如下操作：

(1) 若 ch 为数字，将后续的所有数字依次存入数组 exp 中，并以字符“#”标示数字结束。

(2) 若 ch 为左括号“(”，则将此括号插入栈 stack。

(3) 若 ch 为右括号“)”，则将栈 stack 中左括号“(”以前的字符依次删除并存入数组 exp 中，然后将左括号“(”删除。

(4) 若 ch 为“＋”或“－”，则将当前栈 stack 中的栈顶端连续的“＊”或“/”删除并依次存入数组 exp 中，然后将 ch 插入栈 stack 中。

(5) 若 ch 为“＊”或“\”，则将当前栈 stack 中的栈顶端连续的“＊”或“/”删除并依次存入数组 exp 中，然后将 ch 插入栈 stack 中。

(6) 若 ch 为“#”，则将栈 stack 中的所有运算符依次删除并存入数组 exp 中，然后再将 ch 存入数组 exp 中。最后可在数组 exp 中得到表达式的后缀表示。

对于表达式“(56－20)/(4＋2)#”，其转换成后缀表达式的过程如表 2.2 所示。

表 2.2

stack	exp	说 明
(		ch 为“(”，将此括号插入栈 stack
(	56#	ch 为数字，将 56 存入数组 exp 中，并插入一个字符“#”
(－	56#	ch 为“－”，由于 stack 中“(”以前没有字符，故直接将 ch 插入栈 stack 中

续表

stack	exp	说　明
(−	56#20#	ch 为数字，将 20# 存入数组 exp 中
	56#20#−	ch 为“)”，将栈 stack 中“(”以前的字符依次删除并存入数组 exp 中，然后将“(”删除
/	56#20#−	ch 为“/”，将 ch 插入栈 stack 中
/(	56#20#−	ch 为“(”，将此括号插入栈 stack 中
/(	56#20#−4#	ch 为数字，将 4# 存入数组 exp 中
/(+	56#20#−4#	ch 为“+”，由于 stack 中“(”以前没有字符，故直接将 ch 插入栈 stack 中
/(+	56#20#-4#2#	ch 为数字，将 2# 存入数组 exp 中
/	56#20#−4#2#+	ch 为“)”，故将栈 stack 中“(”以前的字符依次删除并存入数组 exp 中，然后将“(”删除
	56#20#−4#2#+/	ch 为“#”，故将栈 stack 中的所有字符依次弹出并存入数组 exp 中，然后再将 ch 存入数组 exp 中，得到后缀表达式

参考程序如下。

```
1   //求表达式值
2   #include <stdio.h>
3   #define MAX 100
4   char exp[MAX];                          /*存储转换成的后缀表达式*/
5
6   void trans()                            /*将算术表达式转换成后缀表达式*/
7   {
8     char str[MAX];                        /*存储原算术表达式*/
9     char stack[MAX];                      /*作为栈使用*/
10    char ch;
11    int sum,i,j,t,top=0;
12    /*t作为exp的下标,top作为stack的下标,i作为str的下标*/
13
14  printf("*****************************************************************\n");
15    printf("* 输入一个求值的表达式,以#结束。只能包含+,-,*,/运算符和正整数 *\n");
16
17
18  printf("*****************************************************************\n");
19    printf("算术表达式:");
20    i=0;                                  /*获取用户输入的表达式*/
21    do
22    {
23       i++;
24       scanf("%c",&str[i]);
25    } while (str[i]!='#' && i!=MAX);
26    sum=i;                                /*记录输入表达式总的字符个数*/
27    t=1;i=1;
28    ch=str[i];i++;
29    while (ch!='#')
30    {
31       switch(ch)
```

```
32        {
33          case '(':                                  /*判定为左括号*/
34              top++;stack[top]=ch;
35              break;
36          case ')':                                  /*判定为右括号*/
37              while (stack[top]!='(')
38              {
39                exp[t]=stack[top];top--;t++;
40              }
41              top--;
42              break;
43          case '+':                                  /*判定为加减号*/
44          case '-':
45              while (top!=0 && stack[top]!='(')
46              {
47                exp[t]=stack[top];top--;t++;
48              }
49              top++;stack[top]=ch;
50              break;
51          case '*':  /*判定为'*'或'/'号*/
52          case '/':
53              while (stack[top]=='*' || stack[top]=='/')
54              {
55                  exp[t]=stack[top];top--;t++;
56              }
57              top++;stack[top]=ch;
58              break;
59          case ' ':break;
60          default:
61              while (ch>='0' && ch<='9')  /*判定为数字*/
62              {
63                exp[t]=ch;t++;
64                ch=str[i];i++;
65              }
66              i--;
67              exp[t]='#';t++;
68        }
69        ch=str[i];i++;
70    }
71    while (top!=0)
72    {
73      exp[t]=stack[top];t++;top--;
74    }
75    exp[t]='#';
76    printf("\n\t原来表达式:");
77    for (j=1;j<sum;j++) printf("%c",str[j]);
78      printf("\n\t后缀表达式:",exp);
79    for (j=1;j<t;j++)
```

```
80        printf("%c",exp[j]);
81    }
82
83    void compvalue()                          /*计算后缀表达式的值*/
84    {
85      float stack[MAX],d;                     /*作为栈使用*/
86      char ch;
87      int t=1,top=0;                          /*t作为exp的下标,top作为stack的下标*/
88      ch=exp[t];t++;
89      while (ch!='#')
90      {
91        switch (ch)
92        {
93          case '+':stack[top-1]=stack[top-1]+stack[top];
94              top--;break;
95          case '-':stack[top-1]=stack[top-1]-stack[top];
96              top--;break;
97          case '*':stack[top-1]=stack[top-1]*stack[top];
98              top--;break;
99          case '/':if (stack[top]!=0)
100                 stack[top-1]=stack[top-1]/stack[top];
101               else
102               {
103                 printf("\n\t除零错误!\n");
104                exit(0);                      /*异常退出*/
105               }
106               top--;break;
107         default:d=0;
108               while (ch>='0' && ch<='9') /*判定为数字字符*/
109               {
110                 d=10*d+ch-'0';               /*将数字字符转换成对应的数值*/
111                 ch=exp[t];t++;
112               }
113               top++;
114               stack[top]=d;
115        }
116        ch=exp[t];t++;
117      }
118      printf("\n\t计算结果:%g\n",stack[top]);
119    }
120
121    int main()
122    {
123      trans();
124      compvalue();
125      system("pause");
126    }
```

机器人搬运问题

【题目描述】 机器人搬运问题

考验探险者能力的另一个测试是编写一个机器人搬运圆盘的程序，如图 2.2 所示。有两根初始柱 a、b，每根柱子最多只能叠加四块中间有孔的圆盘。初始时每根柱子各放了四块编号为 1 到 8 的圆盘，圆盘的叠放顺序是随机的。有四根过渡柱 c、d、e、f，每根过渡柱子上最多只能叠加 2 块圆盘。机器人每次只能移动柱子上的最上面一块圆盘，问如何用最少的步骤将圆盘有序地叠到目标盘 g、h 上。注意：从 a、b 柱移走的圆盘，不能再放回 a、b 柱，放到 g、h 上的圆盘，不能再次移动；此外，为了保证时间效率，从 a 柱取下的圆盘若要放到过渡柱上，应尽可能先放到同侧的 c、d 柱上，同理，从 b 柱上取下的圆盘，要尽可能先放在同侧的 e、f 柱上。

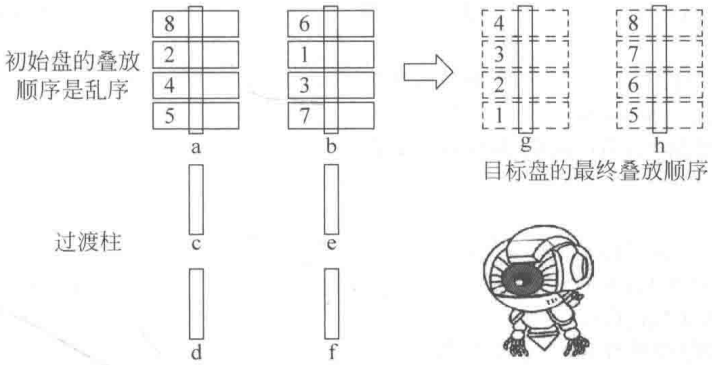


图 2.2

【算法分析】

可以使用两个一维数组 a[4]、b[4]保存初始状态，使用两个一维数组 X[4]、Y[4]保存目标状态用于对比。使用两个一维数组 A[4]、B[4]模拟搬运中目标柱的变化过程，使用四个一维数组 c[2]、d[2]、e[2]、f[2]模拟 c、d、e、f 四根过渡柱的变化过程。在搬运过程中，当 A[4]和 X[4]的每个元素相等，且 B[4]和 Y[4]的每个元素都相等时，即为完成任务。

接下来就是数组仿真堆栈操作即可，参考代码如下。

```
//机器人顺序放 8 块积木
#include <stdio.h>

int a[4]={1,4,3,2},b[4]={5,8,7,6};   //初始柱
int c[2],d[2],e[2],f[2];             //过渡柱
int A[4],B[4];                       //结果柱
int X[4]={1,2,3,4},Y[4]={5,6,7,8};   //对比柱
int aa,bb,cc,dd,ee,ff,AA,BB,XX,YY;   //柱子们的指针
int STEP;                            //统计步数

int OK()                             //通过目标盘和参考盘对比，判断是否成功
{
```

```
13      int i;
14      for(i=0;i<=3;i++)
15        if(X[i]!=A[i])
16          return 0;
17      for(i=0;i<=3;i++)
18        if(Y[i]!=B[i])
19          return 0;
20      return 1;
21    }
22
23    void out()                          //调试用,非正式程序
24    {
25      int i;
26      printf("a: ");
27      for(i=0;i<=3;i++)
28        printf("%2d ",a[i]);
29      printf("   ");
30
31      printf("b: ");
32      for(i=0;i<=3;i++)
33        printf("%2d ",b[i]);
34      printf("      ");
35
36      printf("        c: ");
37      for(i=0;i<=1;i++)
38        printf("%d ",c[i]);
39      printf("   ");
40
41      printf("d: ");
42      for(i=0;i<=1;i++)
43        printf("%d ",d[i]);
44      printf("   ");
45
46      printf("e: ");
47      for(i=0;i<=1;i++)
48        printf("%d ",e[i]);
49      printf(" ");
50
51      printf("f: ");
52      for(i=0;i<=1;i++)
53        printf("%d ",f[i]);
54      printf("\n");
55
56      printf("A: ");
57      for(i=0;i<=3;i++)
58        printf("%2d ",A[i]);
59      printf("   ");
60
61      printf("B: ");
62      for(i=0;i<=3;i++)
```

```
63      printf("%2d ",B[i]);
64    //printf("\n");
65    /*
66    printf("a:%d=%d,b:%d=%d  ",aa,a[aa],bb,b[bb]);
67    printf("  c:%d=%d d:%d=%d ",cc,c[cc],dd,d[dd]);
68    printf("e:%d=%d f:%d=%d   ",ee,e[ee],ff,f[ff]);
69    printf("  A:%d=%d,B:%d=%d  ",AA,A[AA],BB,B[BB]);
70    printf("  X:%d=%d,Y:%d=%d\n\n",XX,X[XX],YY,Y[YY]);   */
71  }
72
73  void step(char x,char y)                    //统计步数并打印步数,实际移动,可用此作接口
74  {
75      ++STEP;
76      printf("%c->%c\n",x,y);
77  }
78
79  void move(char W)                           //移动到过渡
80  {
81    if(W=='a')                                //如果是移动 a 柱
82    {
83      if(cc==-1)                              //如果 c 柱为空,则放 c 柱
84      {
85        step('a','c');
86        c[++cc]=a[aa];
87        a[aa--]=-1;
88      }
89      else if(cc==0 && a[aa]<c[cc])           //如果 c 柱已有一个,并且待放的比这个要小
90      {
91        step('a','c');                        //则再放一个,保证上面的一定比下面的小,以防止卡死
92        c[++cc]=a[aa];
93        a[aa--]=-1;
94      }
95      else if(ee==-1)                         //如果 e 柱为空,则放
96      {
97        step('a','e');
98        e[++ee]=a[aa];
99        a[aa--]=-1;
100     }
101     else if(ee==0 && a[aa]<e[ee])           //如果 e 柱已有一个,则要保证待放的比这个要小
102     {
103       step('a','e');
104       e[++ee]=a[aa];
105       a[aa--]=-1;
106     }
107     else if(dd==-1)                         //如果推荐的过渡柱都不能放,则只能借用另两个柱子
108     {
109       step('a','d');
110       d[++dd]=a[aa];
111       a[aa--]=-1;
112     }
```

```
113        else if(ff==-1)
114        {
115          step('a','f');
116          f[++ff]=a[aa];
117          a[aa--]=-1;
118        }
119      }
120      if(W=='b')                          //同上
121      {
122        if(dd==-1)
123        {
124          step('b','d');
125          d[++dd]=b[bb];
126          b[bb--]=-1;
127        }
128        else if(dd==0 && b[bb]<d[dd])
129        {
130          step('b','d');
131          d[++dd]=b[bb];
132          b[bb--]=-1;
133        }
134        else if(ff==-1)
135        {
136          step('b','f');
137          f[++ff]=b[bb];
138          b[bb--]=-1;
139        }
140        else if(ff==0 && b[bb]<f[ff])
141        {
142          step('b','f');
143          f[++ff]=b[bb];
144          b[bb--]=-1;
145        }
146        else if(cc==-1 )
147        {
148          step('b','c');
149          c[++cc]=b[bb];
150          b[bb--]=-1;
151
152        }
153        else if(ee==-1 )
154        {
155          step('b','e');
156          e[++ee]=b[bb];
157          b[bb--]=-1;
158        }
159      }
160    }
161
162
```

```
163 int main()
164 {
165   int i;
166
167   /*for(i=0;i<=3;i++)                    //输入初始盘
168     scanf("%d",&a[i]);
169   for(i=0;i<=3;i++)
170     scanf("%d",&b[i]);*/
171
172   //for(i=0;i<=3;i++)                    //输入目标参考盘
173   //   X[i]=i+1;                         //此处默认 1,2,3,4,5,6,7,8
174   //   scanf("%d",&X[i]);
175   //for(i=0;i<=3;i++)
176   //   Y[i]=i+5;
177   //   scanf("%d",&Y[i]);
178   XX=0;YY=0;                             //比较柱指针指向最底层
179   aa=3;bb=3;                             //指针指向 3
180   cc=-1,dd=-1,ee=-1,ff=-1,AA=-1,BB=-1;  //指针初始化
181   while(!OK())                    //当没有成功时,摆放策略是能放就放,不能就暂移到过渡柱
182   {
183     if(a[aa]==X[XX] && aa>=0 && XX<=3) //a 柱,后面的并列条件是防止指针越界
184     {
185       step('a','A');
186       A[++AA]=a[aa];
187       a[aa--]=-1;
188       XX++;
189     }
190     else if(a[aa]==Y[YY] && aa>=0 && YY<=3) //必须要防止指针越界,否则会错
191     {
192       step('a','B');
193       B[++BB]=a[aa];
194       a[aa--]=-1;
195       YY++;
196     }
197     else if(b[bb]==X[XX] && bb>=0 && XX<=3) //b 柱
198     {
199       step('b','A');
200       A[++AA]=b[bb];
201       b[bb--]=-1;
202       XX++;
203     }
204     else if(b[bb]==Y[YY] && bb>=0 && YY<=3)
205     {
206       step('b','B');
207       B[++BB]=b[bb];
208       b[bb--]=-1;
209       YY++;
210     }
211     else if(c[cc]==X[XX] && cc>=0 && cc<=1 && XX<=3)     //c 柱
212     {
```

```
213            step('c','A');
214            A[++AA]=c[cc];
215            c[cc--]=-1;
216            XX++;
217        }
218        else if(c[cc]==Y[YY] && cc>=0 && cc<=1 && YY<=3)
219        {
220            step('c','B');
221            B[++BB]=c[cc];
222            c[cc--]=-1;
223            YY++;
224        }
225        else if(d[dd]==X[XX] && dd>=0 && dd<=1 && XX<=3)    //d 柱
226        {
227            step('d','A');
228            A[++AA]=d[dd];
229            d[dd--]=-1;
230            XX++;
231        }
232        else if(d[dd]==Y[YY] && dd>=0 && dd<=1 && YY<=3)
233        {
234            step('d','B');
235            B[++BB]=d[dd];
236            d[dd--]=-1;
237            YY++;
238        }
239        else if(e[ee]==X[XX] && ee>=0 && ee<=1 && XX<=3)    //e 柱
240        {
241            step('e','A');
242            A[++AA]=e[ee];
243            e[ee--]=-1;
244            XX++;
245        }
246        else if(e[ee]==Y[YY] && ee>=0 && ee<=1 && YY<=3)
247        {
248            step('e','B');
249            B[++BB]=e[ee];
250            e[ee--]=-1;
251            YY++;
252        }
253        else if(f[ff]==X[XX] && ff>=0 && ff<=1 && XX<=3)    //f 柱
254        {
255            step('f','A');
256            A[++AA]=f[ff];
257            f[ff--]=-1;
258            XX++;
259        }
260        else if(f[ff]==Y[YY] && ff>=0 && ff<=1 && YY<=3)
261        {
```

```
262         step('f','B');
263         B[++BB]=f[ff];
264         f[ff--]=-1;
265         YY++;
266       }
267       else                          //如果不能直接放,就暂移到过渡柱上
268       {
269         int u,v;
270         if(aa<0)                    //如果 a 柱已经空了
271           move('b');                //那么就移 b 柱
272         else if(bb<0)               //如果 b 柱已经空了
273           move('a');                //那么就移 a 柱
274         else                        //找到最容易移的柱子,即目标盘最靠上的柱子
275         {
276           for(u=aa;u>=0;u--)
277             if(a[u]==X[XX] || a[u]==Y[YY])
278               break;
279           for(v=bb;v>=0;v--)
280             if(b[v]==X[XX] || b[v]==Y[YY])
281               break;
282           if(u>v)
283             move('a');
284           else
285             move('b');
286         }
287       }
288       out();
289       getchar();
290     }
291     printf("Use %d steps,OK!",STEP);
292     getchar();
293     return 0;
294   }
```

课后练习

1. 若已知一个栈的入栈顺序是 1,2,3,…,n,其输出序列为 P1,P2,P3,…,Pn,若 P1 是 n,则 Pi 是(　　)。(第七届 NOIP 提高组初赛选择题第 13 题)

A. i　　B. n−1　　C. n−i+1　　D. 不确定

2. 设栈 S 的初始状态为空,元素 a、b、c、d、e、f、g 依次入栈,以下出栈序列不可能出现的有(　　)。(第十一届 NOIP 提高组初赛试题选择题第 14 题)

A. a,b,c,e,d,f,g　　B. b,c,a,f,e,g,d

C. a,e,c,b,d,f,g　　D. d,c,f,e,b,a,g

E. g,e,f,d,c,b,a

3. 某个车站呈狭长形,宽度只能容下一台车,并且只有一个出入口。已知某时刻该车

站状态为空，从这一时刻开始的出入记录为："进，出，进，进，出，进，进，进，出，出，进，出"。假设车辆入站的顺序为1,2,3,…，则车辆出站的顺序为(　　)。(第十届NOIP提高组初赛试题第3题)

A. 1,2,3,4,5　　B. 1,2,4,5,7

C. 1,3,5,4,6　　D. 1,3,5,6,7

E. 1,3,6,5,7

4. 已知元素(8,25,14,87,51,90,6,19,20)，问这些元素以怎样的顺序进入栈，才能使出栈的顺序满足：8在51前面；90在87的后面；20在14的后面；25在6的前面；19在90的后面。(　　)(第九届NOIP提高组初赛试题第19题)

A. 20,6,8,51,90,25,14,19,87　　B. 51,6,19,20,14,8,87,90,25

C. 19,20,90,8,6,25,51,14,87　　D. 6,25,51,8,20,19,90,87,14

E. 25,6,8,51,87,90,19,14,20

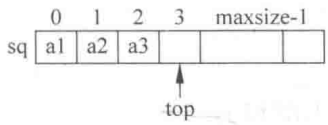
5. 设栈S的初始状态为空，现有5个元素组成的序列(1,2,3,4,5)，对该序列在S栈上依次进行如下操作(从序列中的1开始，出栈后不再进栈)：进栈，进栈，进栈，出栈，进栈，出栈，进栈，试问出栈的元素序列是(　　)。(第四届NOIP提高组初赛试题第10题)

A. 5,4,3,2,1　　B. 2,1　　C. 2,3　　D. 3,4

6. 设有一顺序栈S，元素s1,s2,s3,s4,s5,s6依次进栈，如果6个元素出栈的顺序是s2,s3,s6,s5,s4,s1，则栈的容量至少是(　　)。

A. 2　　B. 3　　C. 4　　D. 5

7. 设有一顺序栈已含3个元素，如下图所示，元素a4正等待进栈。那么下列4个序列中不可能出现的出栈序列是(　　)。



A. a3,a1,a4,a2　　B. a3,a2,a4,a1　　C. a3,a4,a2,a1　　D. a4,a3,a2,a1

8. 若一个栈的输入序列为1,2,3,…,n，输出序列的第一个元素是i，则第j个输出元素是(　　)。

A. i−j−1　　B. i−j　　C. j−i+1　　D. 不确定

9. 设一个栈的输入序列是1,2,3,4,5，则下列序列中，是栈的合法输出序列的是(　　)。

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

答案：1. C；2. CE；3. E；4. D；5. D；6. C；7. A；8. D；9. D

第三章

队　　列

队列的定义

队列简称队，它也是一种运算受限的线性表，其限制仅允许在表的一端进行插入，而在表的另一端进行删除。我们把进行插入的一端称作队尾(rear)，进行删除的一端称作队首(front)。向队列中插入新元素称为进队或入队，新元素进队后就成为新的队尾元素；从队列中删除元素称为离队或出队，元素离队后，其后继元素就成为队首元素。

由于队列的插入和删除操作分别是在各自的一端进行的，每个元素必然按照进入的次序离队，所以又把队列称为先进先出表。

图 3.1 是一个队列的动态示意图。

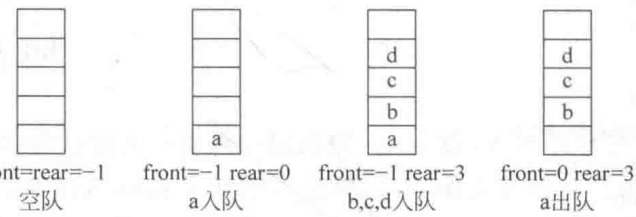


图　3.1

队列的基本操作

初始化队列——initqueue(Q)：建立一个新的空队列 Q。

入队列——enter(Q,x)：将元素 x 插入到队列 Q 中。

出队列——delete(Q)：从队列 Q 中退出一个元素。

取队首元素——gethead(Q)：返回当前的队首元素。

判断队列是否为空——empty(Q)：若队列 Q 为空，则返回 1；否则返回 0。

显示队列中元素——display(Q)：从队首到队尾顺序显示队中所有元素。

数组仿真队列

最简单的，可以使用数组仿真队列，请仔细观察下面的参考程序，看是否有可以改进的地方。

```
1   //数组仿真队列
2   #include <iostream>
3   using namespace std;
4   int queue[5]={0};
5   int front =-1;                          //头指针
6   int rear =-1;                           //尾指针
7
8   void addqueue(int value)
9   {
10    if(rear>=5)
11      cout<<"The queue is full!\n";
12    else
13    {
14      rear++;
15      queue[rear]=value;
16    }
17  }
18
19  int delqueue()
20  {
21    int temp;
22    if(front==rear)
23      return -1;
24    else
25    {
26      front++;
27      temp=queue[front];
28      queue[front]=0;
29      return temp;
30    }
31  }
32
33  void display()
34  {
35    int i;
36    for(i=0;i<5;i++)
37    {
38      if(queue[i]!=0)
39        cout<<queue[i]<<' ';
40    }
41    cout<<endl;
42  }
43
```

```
44  int main()
45  {
46    int select;
47    int i,temp;
48    while(1)
49    {
50      cout<<"1.Input a data\n";
51      cout<<"2.Output a data\n";
52      cout<<"3.Exit\n";
53      cin>>select;
54      switch(select)
55      {
56        case 1:cout<<"Please input a value";
57               cin>>temp;
58               addqueue(temp);
59               break;
60        case 2:if(delqueue()==-1)        //如果队列是空
61                 cout<<"The queue is empty!\n";
62               else
63                 cout<<"The out value is "<<temp<<endl;
64               break;
65
66        case 3:return 0;
67      }
68      system("cls");
69      display();
70    }
71    return 0;
72  }
```

数组循环队列

如果数组仿真队列进行插入一次删一次的操作，只要 2n 次数组就会被用光，当数组仿真队列的元素出队后，队的首部会空出许多位置，而队尾指针指向队列中最后一个元素位置，空出的位置将无法再被利用，导致队列空间的浪费，并且在新的数据元素入队时，会造成“假溢出”，如图 3.2 所示。

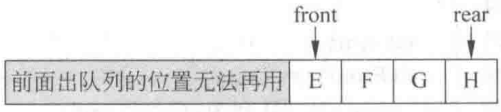


图 3.2

解决的方法是将线形数组模拟成环形，图 3.3 所示是一个有四个元素的环状数组 a[4] 的入队出队操作。

可以看出：

队满条件是 rear=front。

队空条件是 rear=front。

所以这种方法无法判断究竟是队满还是队空。

为了解决这个问题，我们要在入队时少用一个数据元素空间。

所以现在队满的条件可用(rear+1) %MaxSize=front 来判断。

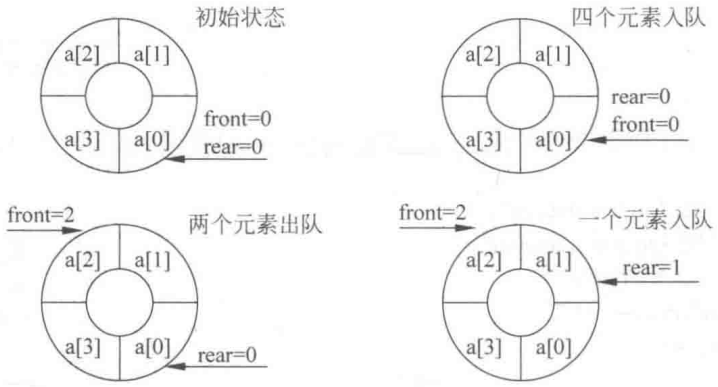


图 3.3

队空条件仍为 rear=front。

```
//循环队列演示程序
#include <iostream>
#define MAXN 5                          /*队列的最多元素个数*/
using namespace std;

int queue[MAXN];
int Front=0;                            //头指针
int Rear=0;                             //尾指针

void addqueue(int value)                //输入 value 值入队列
{
  if(Front==0 && (Rear+1)%MAXN==MAXN)
    cout<<"队满"<<endl;                 //元素从未出队情况下的队满
  else if((Rear+1)%MAXN==Front)
    cout<<"队满"<<endl;                 //已有元素出队后的情况下的队满
  else
  {
    queue[Rear]=value;
    Rear=(Rear+1)%MAXN;
  }
}

int delqueue()                          //输出队列数据
{
  int temp;
  if(Front==Rear)
    cout<<"队列为空"<<endl;
  else
  {
     temp=queue[Front];
     queue[Front]=-1;                   //取出后该位置设置为-1
     Front=(Front+1)%MAXN;              //指向下一位置
     return temp;                       //输出元素值
  }
```

```
35  }
36
37  void display()                          //显示队列里所有的元素
38  {
39    for(int i = 0 ;i < MAXN; ++ i)
40      cout<<queue[i]<<" ";
41    cout<<endl;
42  }
43
44  void init()                             //初始化队列,设定空值为-1*/
45  {
46    for(int i = 0 ;i < MAXN; ++ i)
47      queue[i] = -1;
48  }
49
50  int main()
51  {
52    int select,temp;
53    init();
54    while(1)
55    {
56      cout<<"1.Input a data\n";
57      cout<<"2.Output a data\n";
58      cout<<"3.Display the queue\n";
59      cout<<"4.Exit\n";
60      cin>>select;
61      switch(select)
62       {
63        case 1:cout<<"Please input a value";
64               cin>>temp;
65               addqueue(temp);
66               break;
67        case 2:delqueue();
68               break;
69        case 3:display();
70               break;
71        case 4:return 0;
72      }
73      system("cls");
74      display();
75    }
76    return 0;
77  }
```

链表仿真队列

我们可以使用链表的方式仿真队列,参考程序如下。

```
1  //链表仿真队列
2  #include  <iostream>
```

```
3   using namespace std;
4   typedef struct QueueNode node;
5   typedef struct QueueNode *link;
6   struct QueueNode
7   {
8     int data;
9     struct QueueNode *next;
10  };
11
12  link front =NULL;                    //队头
13  link rear =NULL;                     //队尾
14
15  void addqueue(int value)
16  {
17     link new_node;
18     new_node=(link)malloc(sizeof(node));
19     new_node->data=value;
20     new_node->next=NULL;
21     if(front==NULL)                   //若为队列第一个元素
22        front=new_node;
23     else
24        rear->next=new_node;
25     rear=new_node;
26  }
27
28  int delqueue()
29  {
30     link top;
31     int temp;
32     if(front!=NULL)
33     {
34        top=front;
35        front=front->next;
36        temp=top->data;
37        free(top);
38        return temp;
39     }
40     else
41     {
42        rear=NULL;
43        return -1;
44     }
45  }
46
47  void display()
48  {
49     link point;
50     point=front;
51     while(point!=NULL)
52     {
```

```
53        cout<<point->data<<' ';
54        point=point->next;
55    }
56    cout<<endl;
57  }
58
59  int main()
60  {
61    int select;
62    int i,temp;
63    while(1)
64    {
65      cout<<"1.Input a data\n";
66      cout<<"2.Output a data\n";
67      cout<<"3.Exit\n";
68      cin>>select;
69      switch(select)
70       {
71        case 1:cout<<"Please input a value";
72               cin>>temp;
73               addqueue(temp);
74               break;
75        case 2:if(delqueue()==-1)  //如果队列是空
76                 cout<<"The queue is empty!\n";
77               else
78                 cout<<"The out value is"<<temp<<endl;
79               break;
80
81        case 3:return 0;
82       }
83      system("cls");
84      display();
85    }
86    return 0;
87  }
```

队列排序

【题目描述】 队列排序

因为报名参加探险的人很多(有 n 个,0<n<100000),为了方便管理,给每个人发了一个编号从 0~9 的号码牌,每个人要根据自己的号码牌找到自己所在的队伍排好队。抽象来说,就是输入 n 个 10 以内的数,每输入 i(0≤i≤9),就把它插入到第 i 号队列中。最后把 10 个队列中的非空队列按队列号以从小到大的顺序串接成一条链,并输出该链的所有元素。

【输入格式】

两行,第一行一个整数,表示排序的数 n。

第二行有 n 个整数,即各待排序的数,以空格间隔。

【输出格式】

输出排好序的数，以空格间隔。

【输入样例】

3

4 2 1

【输出样例】

1 2 4

参考程序如下。

```
//队列排序
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
#define N 100
struct node
{
  int data;
  struct node *next;
};

struct hnode
{
  struct node *link;
} HQ[MAX];

int main()
{
  int A[N],n,i,first=1;
  struct node *p,*s,*head;
  printf("请输入数值的个数 n:");
  scanf("%d",&n);
  for (i=0;i<MAX;i++)
    HQ[i].link=NULL;
  for (i=0;i<n;i++)
    do
    {
      printf("第%d 个数:",i+1);
      scanf("%d",&A[i]);
    } while (A[i]<0 || A[i]>9);
  for (i=0;i<n;i++)
  {
    s=(struct node *)malloc(sizeof(struct node));
    s->data=A[i];s->next=NULL;
     if (HQ[A[i]].link==NULL)
        HQ[A[i]].link=s;
    else
    {
```

```
40          p=HQ[A[i]].link;
41          while (p->next!=NULL)
42            p=p->next;                    /*p指向该链表的最后一个结点*/
43          p->next=s;
44        }
45      }
46      head=(struct node *)malloc(sizeof(struct node));
47      head->next=NULL;
48      for (i=0;i<MAX;i++)
49      {
50        if (HQ[i].link!=NULL)
51        {
52          if (first)
53          {
54            head->next=HQ[i].link;first=0;
55             p=head;
56             while (p->next!=NULL)
57               p=p->next;                 /*p指向最后结点*/
58          }
59          else
60          {
61            p->next=HQ[i].link;
62            p=HQ[i].link;
63             while (p->next!=NULL)
64               p=p->next;
65          }
66        }
67      }
68      printf("\n显示所有元素:\n");
69      if (head->next!=NULL)
70        head=head->next;
71      while (head!=NULL)
72      {
73        printf("%d ",head->data);
74        head=head->next;
75      }
76      system("pause");
77      return 0;
78    }
```

搭档问题

【题目描述】 搭档问题

每一队的报名者，在测试能力时必须男女随机搭档，所以男士和女士进入考场前，各自排成一队。测试开始后，依次从男队和女队的队头上各出一人配成搭档。若两队初始人数不相同，则较长的一队中未配对者等待下一轮测试。已知测试有无数场，采取循环测试的方式，现要求输出第 k 轮测试的匹配情况。

【输入格式】

共三行，第一行为三个数字，即男士数 M、女士数 F 和 k 值。M、N≤26，K≤100。

第二行为一行长度为 M 的字符串，每个字符表示一个男士的标志。

第三行为一行长度为 F 的字符串，每个字符表示一个女士的标志。

【输出格式】

输出第 k 场考试的匹配情况。每一行为一个匹配。

【输入样例】

5 4 3

ABCDE

abcd

【输出样例】

D a

E b

A c

B d

参考程序如下。

```
//搭档问题
#include <iostream>
using namespace std;
#define maxsize 100

typedef struct
{
  char data[maxsize];
  int front,rear;
}seqQueue;


void iniqueue(seqQueue *M,int m)          //队列初始化
{
   scanf("%s",M->data);
   M->front=0;
   M->rear=m;
}

void addqueue(seqQueue *q,int m)          //队列插入结点
{
   q->front=(q->front+1)%m;
   q->rear=(q->rear+1)%m;
}

char outqueue(seqQueue *q,int m)          //队列删除结点
{
  if(q->rear==q->front)
    return NULL;
```

```
30      else
31      {
32        q->front=(q->front+1)%m;
33        return q->data[q->front];
34      }
35    }
36
37    int main()
38    {
39      int n,m,k,i,j;
40
41      seqQueue M,F;
42      scanf("%d%d%d",&n,&m,&k);
43
44      iniqueue(&M,n);
45      iniqueue(&F,m);
46      if(n<m)
47      {
48        for(i=0;i<k-1;i++)
49        {
50          for(j=0;j<n;j++)
51          {
52            addqueue(&M,n);
53            addqueue(&F,m);
54          }
55        }
56        for(i=0;i<n;i++)
57        {
58          printf("%c ",M.data[M.front]);
59          printf("%c\n",F.data[F.front]);
60          addqueue(&M,n);
61          addqueue(&F,m);
62        }
63      }
64      else
65      {
66        for(i=0;i<k-1;i++)
67        {
68          for(j=0;j<m;j++)
69          {
70            addqueue(&F,m);
71            addqueue(&M,n);
72          }
73        }
74        for(i=0;i<m;i++)
75        {
76          printf("%c ",M.data[M.front]);
77          printf("%c\n",F.data[F.front]);
78          addqueue(&M,n);
79          addqueue(&F,m);
```

```
        }
    }
    system("pause");
    return 0;
}
```

封闭面积问题

【题目描述】 封闭面积问题(area.c/cpp/pas)

据已探明的情报，上古文明遗迹的面积大得惊人，设遗迹为由“*”号围成的下列图形，其面积的计算方法是统计“*”号所围成的闭合曲线中水平线和垂直线交点的数目。如图 3.4 所示，在 10×10 的二维数组中，由“*”号围住 15 个点。因此面积为 15。

【输入格式】

第一行两个整数 M，N，即矩阵的列数和行数，行列数不超过 90000。以下 N 行描述矩阵，以空格间隔。

【输出格式】

输出面积。

【输入样例】

```
4  3
0  0  0  0
0  *  *  0
*  0  *  0
```

【输出样例】

1

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 * * * 0 0 0
0 0 0 0 * 0 0 * 0 0
0 0 0 0 0 * 0 0 * 0
0 0 * 0 0 0 * 0 * 0
0 * 0 * 0 * 0 0 * 0
0 * 0 0 * * 0 * * 0
0 0 * 0 0 0 0 * 0 0
0 0 0 * * * * * 0 0
0 0 0 0 0 0 0 0 0 0
```

图 3.4

【算法分析】

可以用 10×10 的二维数组存储这张图，我们先统计“封闭曲线”外的“0”的个数，把总个数减去外围“0”的个数，再减去“*”号的个数，剩下的就是图形面积。外围“0”的个数统计方法采取从左上角的“0”开始，找所有与它相邻的“0”，找到后存入队列。一个“0”的相邻点找完后，再找队列下一个“0”的相邻点。这样继续下去，直到所有“0”找到为止，为使已找过的点不再重复计算，设存入队列的点的标志为 2。

参考程序如下。

```
//封闭面积问题
#include<iostream>
#include<fstream>
#define MAXN 90000
using namespace std;

ifstream fin("area.in");
ofstream fout("area.out");

struct img
```

```
11  {
12    short int x;
13    short int y;
14  }a[MAXN];
15
16
17  int main()
18  {
19      char temp;
20      short int dx[4],dy[4];
21
22      dx[0]=0;  dy[0]=1;              //建立方向
23      dx[1]=1;  dy[1]=0;
24      dx[2]=0;  dy[2]=-1;
25      dx[3]=-1;  dy[3]=0;
26
27      int m,n,tt,kk,xing=0,ttxx,ttyy;
28      fin>>m>>n;
29      bool pos[n+2][m+2];
30      memset(pos,0,sizeof(pos));
31
32      int i,j;
33      for(i=1;i<=n;++i)               //输入图像
34        for(j=1;j<=m;++j)
35        {  fin>>temp;
36           if(temp=='*')  {pos[i][j]=1;  xing++;}
37           else if(temp=='0')  pos[i][j]=0;
38        }
39      for(i=1;i<=n+1;++i)
40          pos[i][m+1]=1;
41      for(i=1;i<=m+1;++i)
42          pos[n+1][i]=1;
43      for(i=0;i<=m+1;++i)
44          pos[0][i]=1;
45      for(i=1;i<=n+1;++i)
46          pos[i][0]=1;
47
48    int sum=1,front,rear;
49    tt=1;
50    kk=1;
51    a[1].x=1;      a[1].y=1;
52    pos[1][1]=1;
53
54    while(tt<=kk)
55    {
56        for(i=0;i<=3;++i)           //各个方向各试一遍
57        {
58            front=tt%MAXN;
59            ttxx=a[front].x+dx[i];;
60            ttyy=a[front].y+dy[i];
```

```
61                if( pos[ttyy][ttxx]==0  )
62                {
63                    kk++;
64                    rear=kk%MAXN;
65                    a[rear].x=ttxx;
66                    a[rear].y=ttyy;
67                    sum++;
68                    pos[ttyy][ttxx]=1;
69                }
70            }
71            tt++;
72
73        }
74
75        fout<<m*n-sum-xing<<endl;
76        return 0;
77        fin.close();
78        fout.close();
79    }
```

第四章

树

基本概念

树状结构是一种重要的非线性结构，它用于描述数据元素之间的层次关系。每一棵树必有一特定的结点，称作根结点(root)。根结点之下可以有零个以上的子结点(可以没有)。而各子结点也可为子树，拥有自己的子结点。

图 4.1 中，结点 A 为树 T 的根结点(root)，B，C，D，M 则为结点 A 的子结点。若包含其下拥有的所有子结点，则为 T 的子树。例如，B 是 A 的子结点，P，Q 皆是 B 的子结点。而 B，P，Q 为树 T 的子树。

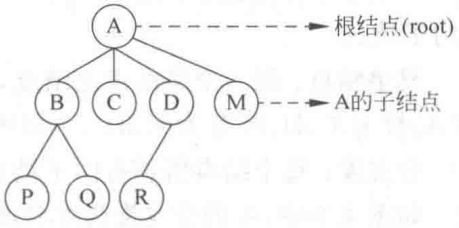


图 4.1

若一棵树中的结点最多可以有 n 个子结点，则称这样的树为 n 元树。例如，二叉树中的结点，最多只能有两个子结点。

树的表示

树的表示有树型表示法、文氏图表示法、凹入表示法和括号表示法等，如图 4.2 所示。

(1) 树型表示法：这是树的最基本表示，用一棵倒置的树表示树结构非常直观和形象。

图 4.2

(2) 文氏图表示法：使用集合以及集合的包含关系描述树结构。

(3) 凹入表示法：使用线段的伸缩描述树结构。

(4) 括号表示法：将树的根结点写在括号的左边，除根结点之外的结点写在括号中并用逗号隔开来描述树结构。

树的术语

根结点：一棵树中没有父结点的结点，称为根结点。

叶结点：一棵树中没有子结点的结点，称为叶结点。

非终端结点：除了叶结点以外的其他结点，称为非终端结点。

结点示意图如图 4.3 所示。

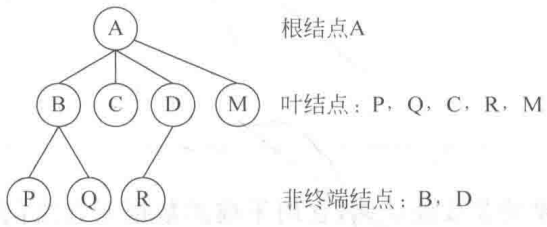


图 4.3

父结点和子结点：若结点 x 有一个以结点 y 为树根的子树，则 x 为 y 的父结点，而 y 为 x 的子结点。

兄弟结点：同一个父结点之结点，称为兄弟结点。如图 4.3 中，B、C、D、M 的父结点均为 A，故 B、C、D、M 互为兄弟。

分支度：每个结点所拥有的子结点个数。而一棵树中最大分支度值，即为该树的分支度。如图 4.3 中，A 的分支度为 4，B 的分支度为 2，D 的分支度为 1，故该树的分支度为 4。

阶层：阶层为结点之特性值，将根结点之阶层设为 1，其子结点为 2，以此类推。如图 4.3 中，阶层为 1 有结点 A，阶层为 2 有结点 B、C、D、M，阶层为 3 有 P、Q、R。

高度或深度：一棵树中的最大阶层值，称为树的高度或深度。如图 4.3 中，最大阶层数为 3，故树之高度为 3。

祖先：由某结点 x 到根结点之路径上的所有结点，均称为 x 之祖先。如图 4.3，P 点的祖先为 A、B。

树林：n≥0 个的集合称为树林。若将一树的根结点移去，所剩的恰是一树林，如图 4.4 所示。

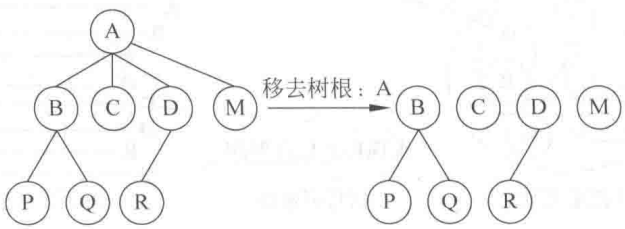


图 4.4

有序树和无序树：如果一棵树中结点的各子树从左到右是有次序的，即若交换了某结点各子树的相对位置构成了不同的树，则称这棵树为有序树，反之则为无序树。

二叉树的概念

二叉树(binary tree)是树的一种，二叉树中的结点至多只能有两个子结点。

二叉树的定义如下：

(1) 由有限个结点所构成之集合，此集合可以为空的。

(2) 二叉树的根结点下可分成两个子树，称为左子树和右子树，左子树和右子树亦称二叉树，如图 4.5 所示。

由于二叉树之子树有顺序关系，分为左子树和右子树，所以图 4.6 中的两棵树是相同的树，但却是不同的二叉树。

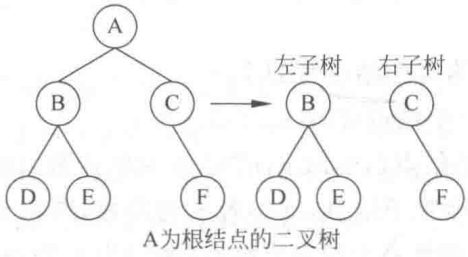


图 4.5

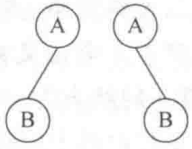


图 4.6

二叉树和树的比较：

(1) 二叉树可为空，而树不可以为空(至少要有根结点)。

(2) 二叉树的子树有顺序关系，而树没有。

(3) 二叉树的分支度必为 0,1 或 2，而树的分支度可大于 2。

歪斜树：一般二叉树可视为根结点、左子树和右子树的集合。若一棵树中，所有左结点均不存在，则此树为右歪斜树；反之，若所有右结点均不存在，则此树为左歪斜树，如图 4.7 所示。

满二叉树：一棵树中所有叶结点均在同一阶层，而其他非终端结点之分支度均为 2，则此树为一满二叉树。若该树的高度为 h，则此满二叉树的结点为 2^h-1，如图 4.8 所示。

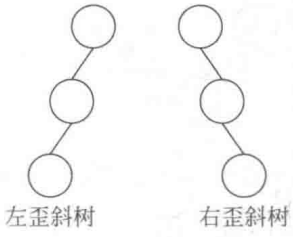


图 4.7

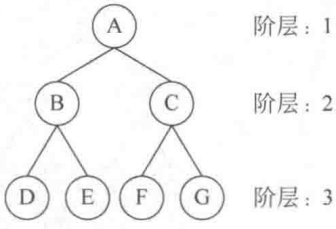


图 4.8

图 4.8 中二叉树高度为 3，结点数为 $2^3-1=7$，为满二叉树。

完全二叉树：一棵树去除掉最大阶层那一层后为一满二叉树，且阶层最大那层的结点

均向左靠齐，则该二叉树称为完全二叉树，如图 4.9 所示。

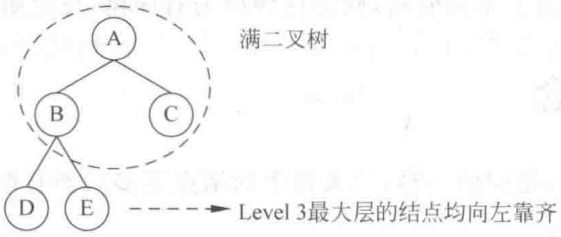


图 4.9

图 4.9 中若去除 Level 3 这一层后，A，B，C 为一满二叉树，且 D，E 在 Level 3 中均向左靠齐，故称其为一完全二叉树。

二叉树的性质

性质 1：二叉树上叶子结点数等于度为 2 的结点数加 1。

性质 2：二叉树的第 i 层上至多有 2^{i-1} 个结点（$i \geqslant 1$）。

性质 3：对于完全二叉树中编号为 i 的结点（$1 \leqslant i \leqslant n$，$n \geqslant 1$，n 为结点数），有：(1)若 $2 \times i \leqslant n$，则编号为 i 的结点为分支结点，否则为叶子结点。(2)若 n 为奇数，则每个分支结点都既有左孩子，又有右孩子；若 n 为偶数，则编号最大的分支结点（编号为 n/2）只有左孩子，没有右孩子，其余分支结点左、右孩子都有。

二叉树数组表示法

我们可将一个二叉树按阶层从低到高、由左到右、从 1 开始依序编号，再根据编号存入相对应索引编号的数组中，若该树不是满二叉树，也可对各结点编成在满二叉树中相同位置的结点编号值，再以相同的方式存入数组中，若某一编号没有结点存在，则不存值于数组中。

从图 4.10 所示可以看出以下两点。

(1) 左子结点为父结点乘以 2：$2 \times n$。

(2) 右子结点为父结点乘以 2 加 1：$2 \times n + 1$。

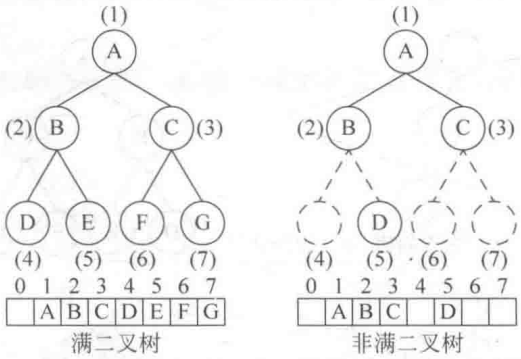


图 4.10

例如，图 4.10 左图中 A 结点的编号为 1，则左子结点 B 编号为 $2\times1=2$，右子结点 C 编号为 $2\times1+1=3$，故若要用 1 个一维数组来代表一棵树二叉树，所需的数组长度为 2^h-1，h 为最大阶层数。

建立二叉树结点数据的规则如下：

(1) 以第 1 个建立的元素为根结点。

(2) 依序将元素值与根结点做比较。

① 若元素值大于根结点值，则将元素值往根结点的右子结点移动，若此右子结点为空，则将元素值存入；否则就重复比较，直到找到适当的空结点为止。

② 若元素值小于根结点值，则将元素值往根结点的左子结点移动，若此左子结点为空，则将元素值存入；否则就重复比较，直到找到适当的空结点为止。

上述的二叉树建立原则满足了二叉排序树(Binary Sort Tree)又称二叉查找树的条件。关于二叉查找树的条件将在后面做详细介绍。

例如，输入数据的顺序为 5、4、6、3、9、8、2、4.5、5.5，则建立的二叉树结构如图 4.11 所示。

二叉树数组表示的优点是：对于任一个结点都能很容易地找到其父结点、子结点及兄弟结点，而且每个结点的存储空间不大，只占用数组的一个内存空间。但当二叉树之深度和结点数之比例偏高时(二叉树分布不平均，如歪斜树)，则内存的利用率会偏低，容易造成空间的浪费。另外，由于数组表示法是用循序的方式处理，故在插入或删除结点时，需要移动其他元素方可完成。

例如，图 4.12 所示的二叉树在数组中的分布如图 4.13 所示。

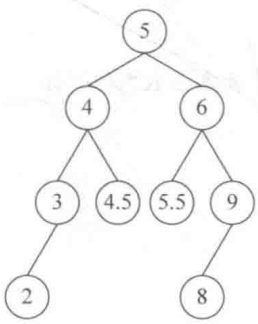


图 4.11

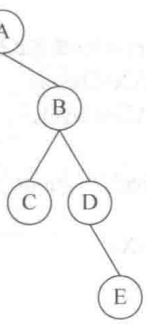


图 4.12

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A		B			C	D								E

图 4.13

【题目描述】 构造二叉查找树(arraytree.c/cpp/pas)

有一棵深度为 N(N≤20)的树，请用数组建立二叉查找树。

【输入格式】

第一行为一整数 N，表示有多少个元素，第二行为 N 个元素。

【输出格式】

按数组顺序输出即可。

【输入样例】

9

6 3 8 5 2 9 4 7 10

【输出样例】 6 3 8 2 5 7 9 0 0 4 0 0 0 0 10

【内存限制】 400KB

参考程序如下。

```
// 构造数组二叉树
#include <iostream>
#define MAXN 32768                    //2^15
using namespace std;
int tree[MAXN];                       //所有数组均从 1 开始

int create(int _tree[ ],int _node[ ],int len)
{
  int i,MAX=1;
  _tree[1]=_node[1];
  int level=1;
  for(i=2;i<=len;i++)
  {
    level=1;
    while(_tree[level]!=0)            //判断当前位置是否有值
    {
      if(_node[i]<_tree[level])
        level=level*2;
      else
        level=level*2+1;
      if(MAX<level)                   //判断最后一个元素占数组的下标
        MAX=level;
    }
    _tree[level]=_node[i];
  }
  return MAX;
}

int main()
{
  freopen("arraytree.in","r",stdin);
  freopen("arraytree.out","w",stdout);
  int n,num;
  cin>>n;
  int node[n+1];
  for(int i=1;i<=n;i++)
    cin>>node[i];
  num=create(tree,node,n);
  for(int i=1;i<=num-1;i++)
    cout<<tree[i]<<' ';
  cout<<tree[num]<<endl;
  return 0;
}
```

二叉树结构数组法

由于二叉树的结点最多只能有两个子结点，所以我们可以用结构的类型来声明结点的存储方式。此结构包含 3 个字段，其中一个字段用来存放结点的数据内容，而另两个字段则分别存放左子树和右子树在数组中的索引值，如图 4.14 所示。

data：存放结点的数据内容。

left：存放左子树在数组中的索引值。

right：存放右子树在数组中的索引值。

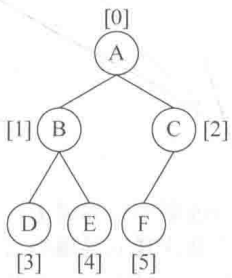
left	data	right

图 4.14

结构数组的结构体声明如下：

```
struct
{
  int left;
  int data;
  int right;
}b_tree[15];
```

声明完成后，b_tree 即用来存放二叉树各结点的结构数组。在结构数组中，会将根结点置于数组结构中索引值为 0 之处，将结点值存在 data 字段，而 left 和 right 字段则分别存储左右子树在数组结构中的索引值，若子树不存在则存值－1。例如，有一棵二叉树其树状结构与结构数组表示法如图 4.15 所示。



索引	left	data	right
0	1	A	2
1	3	B	4
2	5	C	−1
3	−1	D	−1
4	−1	E	−1
5	−1	F	−1

图 4.15

【题目描述】 结构数组法建立二叉树(arraytree.c/cpp/pas)

有一棵深度为 N(N≤20)的树，请用结构数组法建立二叉树。

【输入格式】

第一行为一整数 N，表示有多少个元素，第二行为 N 个元素。

【输出格式】 按数组顺序输出即可。

【输入样例】

9

6 3 8 5 2 9 4 7 10

【输出样例】

2 6 3

5 3 4

8 8 6
7 5 −1
−1 2 −1
−1 9 9
−1 4 −1
−1 7 −1
−1 10 −1

【内存限制】 800KB

参考程序如下。

```
//二叉树结构数组表示法
#include <iostream>
using namespace std;
#define MAXN 32768                          //2^15
int n;
struct tree
{
  int Left;
  int Date;
  int Right;
};

typedef struct tree treenode;
treenode b_tree[MAXN];

void create(int * node,int len)
{
  int i;
  int level;
  int position;                             //左树−1,右树 1
  for(i=1;i<=len;i++)                       //依次建立其他结点
  {
     b_tree[i].Date=node[i];                //元素值存入结点
     level=0;                               //树根为 0
     position=0;
     while(position==0)
     {
       if(node[i]>b_tree[level].Date)       //判断是否为右子树
         if(b_tree[level].Right!=−1)
           level=b_tree[level].Right;
         else
           position=−1;                     //设为右树
       else                                 //判断是否为左子树
         if(b_tree[level].Left!=−1)
           level=b_tree[level].Left;
         else
           position=1;                      //设为左树
     }
```

```
39          if(position==1)
40            b_tree[level].Left=i;                    //连接左子树
41          else
42            b_tree[level].Right=i;                   //连接右子树
43        }
44    }
45
46    void print()
47    {
48      int i;
49      for(i=1;i<=n;i++)
50        cout<<b_tree[i].Left<<' '<<b_tree[i].Date<<' '<<b_tree[i].Right<<'\n';
51    }
52
53    int main()
54    {
55      freopen("arraytree.in","r",stdin);
56      freopen("arraytree.out","w",stdout);
57      int i;
58      cin>>n;
59      int node[n+1];
60      for(i=1;i<=n;i++)
61        cin>>node[i];
62      for(i=0;i<MAXN;i++)
63      {
64        b_tree[i].Left=-1;
65        b_tree[i].Date=0;
66        b_tree[i].Right=-1;
67      }
68      create(node,n);
69      print();
70    }
```

二叉树链表表示法

二叉树的链表表示法与结构数组法很相似，都是用 3 个字段来存储结点信息。两者之间的不同在于结构数组是以循序静态的方式处理，而链表是运用动态内存配置的方法来建立二叉树，其中左右字段是用来连结左右子树的指针。链表的结点结构如图 4.16 所示。

date：存放结点的数据内容。

left：指向左子树的指针。

right：指向右子树的指针。

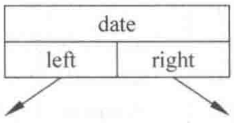


图 4.16

二叉树链表结构的声明如下：

```
struct tree
{
  struct tree *left;
  int data;
```

```
  struct tree *right;
};
typedef struct tree treenode;
treenode *b_tree;
```

【题目描述】 用链表建立二叉树(linktree.c/cpp/pas)

有一棵深度为N(N≤20)的树,请用链表建立二叉树。

【输入格式】

第一行为一整数N,表示有多少个元素,第二行为N个元素。

【输出格式】

按数组顺序输出即可。

【输入样例】

9

6 3 8 5 2 9 4 7 10

【输出样例】

6 3 2 5 4 8 7 9 10

(注意:输出的最后一个数字后为一个空格加换行)

参考程序如下。

```
1  //构造链表二叉树
2  #include <iostream>
3  using namespace std;
4
5  struct tree
6  {
7    struct tree *left;
8    int date;
9    struct tree *right;
10 };
11 typedef struct tree treenode;
12 typedef struct tree *b_tree;
13
14 b_tree insert(b_tree root,int node)
15 {
16   b_tree newnode;
17   b_tree currentnode;
18   b_tree parentnode;
19
20   newnode=(b_tree)malloc(sizeof(treenode));
21   newnode->date=node;
22   newnode->left=NULL;
23   newnode->right=NULL;
24
25   if(root==NULL)                    //第一个结点建立
26     return newnode;
27   else
28   {
```

```
29        currentnode=root;                          //存储当前结点
30        while(currentnode!=NULL)
31        {
32          parentnode=currentnode;                  //存储父结点
33          if(currentnode->date>node)
34            currentnode=currentnode->left;         //左子树
35          else
36            currentnode=currentnode->right;        //右子树
37        }
38        if(parentnode->date>node)
39          parentnode->left=newnode;
40        else
41          parentnode->right=newnode;
42      }
43      return root;
44    }
45
46    b_tree create(int * data,int len)              //建立二叉树
47    {
48      int i;
49      b_tree root=NULL;
50      for(i=1;i<=len;i++)
51        root=insert(root,data[i]);
52      return root;
53    }
54
55    void print(b_tree root)                        //递归打印二叉树
56    {
57      if(root!=NULL)
58      {
59        printf("%d ",root->date);
60        print(root->left);
61        print(root->right);
62      }
63    }
64
65    int main()
66    {
67      freopen("linktree.in","r",stdin);
68      freopen("linktree.out","w",stdout);
69      int i,n;
70      b_tree root=NULL;
71      cin>>n;
72      int node[n+1];
73      for(int i=1;i<=n;i++)
74        cin>>node[i];
75
76      root=create(node,n);
77      print(root);
78      cout<<endl;
```

```
    return 0;
}
```

数组结构转链表结构

二叉树的数组表示法中，是将二叉树的结点置入适当的数组位置中，如果二叉树的结点分布不均时，容易造成内存空间的浪费，现给定一个二叉树数组结构，如何使用递归方式建立一棵二叉树？

【题目描述】 数组结构转链表结构(array2link.c/cpp/pas)

请编程将数组结构的二叉树转为链表结构。

【输入格式】

输入数组结构的二叉树。

【输出格式】

中序输出链表结构的二叉树。

【输入样例】

5 2 9 1 4 7 0 0 0 3 0 6 8 0 0(注意：数组从下标1开始)

【输出样例】

1 2 3 4 5 6 7 8 9(注意：最后一个元素后面为一空格加换行)

需要说明的是，该程序递归层次不能过深，一般当数组超30000时可能会导致程序的崩溃，并且数据中应不包括0。参考程序如下。

```
//数组结构二叉树递归转链表结构二叉树
#include <iostream>
using namespace std;

struct tree
{
    struct tree *left;
    int date;
    struct tree *right;
};
typedef struct tree treenode;
typedef struct tree *b_tree;
int N;

b_tree create(int *node,int position)                    //递归建立二叉树
{
    b_tree newnode;
    if(node[position]==0 || position>N)
        return NULL;
    else
    {
        newnode=(b_tree)malloc(sizeof(treenode));
```

```
23        newnode->date=node[position];
24        newnode->left=create(node,2*position);
25        newnode->right=create(node,2*position+1);
26        return newnode;
27    }
28 }
29
30 void print(b_tree root)                    //中序遍历
31 {
32    if(root!=NULL)
33    {
34        print(root->left);
35        printf("%d ",root->date);
36        print(root->right);
37    }
38 }
39
40 int main()
41 {
42    freopen("array2link.in","r",stdin);
43    freopen("array2link.out","w",stdout);
44    int i=1,temp;
45    int node[10000];
46    b_tree root=NULL;
47    while(cin>>temp)
48        node[i++]=temp;
49    N=i-1;
50    root=create(node,1);
51    print(root);
52    cout<<endl;
53    return 0;
54 }
```

二叉树的前序遍历

【题目描述】 前序遍历(preorder_traversal.cpp/c/pas)

上古文明遗迹里有很多的岔路,我们可以把它看成一棵二叉树。作为探险小分队的一员,你的任务是建立二叉树后输出前序遍历。

【输入格式】

第一行为一整数 N,表示有多少个元素,第二行为 N 个元素。

【输出格式】

依次输出数据,每个数据间有一个空格。

【输入样例】

9

6 3 8 5 2 9 4 7 10

【输出样例】

6 3 2 5 4 8 7 9 10

【算法分析】

前序遍历是先遍历根结点，再遍历左子树，最后才遍历右子树，如图 4.17 所示。其前序遍历为：从根结点 A 开始先往左子树 B 再到 D，由于 D 没有左子树，故转向右子树 G。再回到 B，因为 B 没有右子树，所以此时 A 的左子树遍历完毕。则转向 A 之右子树 C，再往左边继续遍历，以此类推，故可得到前序遍历的顺序为：A，B，D，G，C，E，H，I，F。

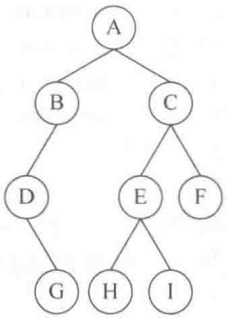


图 4.17

参考代码如下。

```
1   //前序遍历
2   #include <iostream>
3   using namespace std;
4
5   struct tree
6   {
7     struct tree *left;
8     int date;
9     struct tree *right;
10  };
11  typedef struct tree treenode;
12  typedef struct tree *b_tree;
13
14  b_tree insert(b_tree root,int node)                    //插入结点
15  {
16    b_tree newnode;
17    b_tree currentnode;
18    b_tree parentnode;
19
20    newnode=(b_tree)malloc(sizeof(treenode));
21    newnode->date=node;
22    newnode->left=NULL;
23    newnode->right=NULL;
24
25    if(root==NULL)                                       //第一个结点建立
26      return newnode;
27    else
28    {
29      currentnode=root;                                  //存储当前结点
30      while(currentnode!=NULL)
31      {
32        parentnode=currentnode;                          //存储父结点
33        if(currentnode->date>node)
34          currentnode=currentnode->left;                 //左子树
35        else
36          currentnode=currentnode->right;                //右子树
37      }
```

```
38        if(parentnode->date>node)
39            parentnode->left=newnode;
40        else
41            parentnode->right=newnode;
42    }
43    return root;
44 }
45
46 b_tree create(int * data,int len)                    //建立二叉树
47 {
48    int i;
49    b_tree root=NULL;
50    for(i=1;i<=len;i++)
51        root=insert(root,data[i]);
52    return root;
53 }
54
55 void print(b_tree root)                              //前序遍历
56 {
57    if(root!=NULL)
58    {
59        cout<<root->date<<' ';
60        print(root->left);
61        print(root->right);
62    }
63 }
64
65 int main()
66 {
67    freopen("preorder_traversal.in","r",stdin);
68    freopen("preorder_traversal.out","w",stdout);
69    int n,i;
70    b_tree root=NULL;
71    cin>>n;
72    int node[n+1];
73    for(i=1;i<=n;i++)
74        cin>>node[i];
75    root=create(node,n);
76    print(root);
77    cout<<endl;
78    return 0;
79 }
```

二叉树的中序遍历

【题目描述】 中序遍历(inorder_traversal.cpp/c/pas)

上古文明遗迹里有很多的岔路,我们可以把它看成一棵二叉树,作为探险小分队的一员,你的任务是建立二叉树后输出中序遍历。

【输入格式】

第一行为一整数 N,表示有多少个元素,第二行为 N 个元素。

【输出格式】 依次输出数据,每个数据间有一个空格。

【输入样例】

9

6 3 8 5 2 9 4 7 10

【输出样例】

2 3 4 5 6 7 8 9 10

【算法分析】

中序遍历是先遍历左子树,再遍历根结点,最后才遍历右子树。例如,图 4.18 的二叉树的中序遍历为:从结点 A 开始,一直往左走到 D 再无法前进,则处理 D,再往 D 的右方到 G。此时已遍历完 B 的左子树,接着处理 B,再往 B 的右方前进。由于 B 没有右子树,故 A 的左子树遍历完毕,可处理结点 A,再往 A 的右子树前进,以此类推,故可得到中序遍历顺序为:D,G,B,A,H,E,I,C,F。

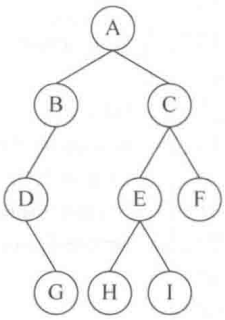


图 4.18

其程序除遍历输出部分不同外,其他与前序遍历程序完全相同。现给出输出部分代码如下。

```
1  void print(b_tree root)        //中序遍历
2  {
3    if(root!=NULL)
4    {
5      print(root->left);
6      cout<<root->date<<' ';
7      print(root->right);
8    }
9  }
```

二叉树的后序遍历

【题目描述】 后序遍历(postorder_traversal.cpp/c/pas)

上古文明遗迹里有很多的岔路,我们可以把它看成一棵二叉树。作为探险小分队的一员,你的任务是建立二叉树后输出后序遍历。

【输入格式】

第一行为一整数 N,表示有多少个元素;第二行为 N 个元素。

【输出格式】 依次输出数据,每个数据间有一个空格。

【输入样例】

9

6 3 8 5 2 9 4 7 10

【输出样例】

2 4 5 3 7 10 9 8 6

【算法分析】

后序遍历是先遍历左子树，再遍历右子树，最后才遍历根结点。例如，图 4.19 的二叉树后序遍历为：从结点 A 开始一直往左走到 D 无法再前进，则往 D 的右方前进到 G，由于 G 没有左、右子树，故处理结点 G。之后由于 D 的右子树遍历完毕，故进而处理 D，而 B 的左子树也相对地完成。且结点 B 没有右子树，故可接着处理 B。此时结点 A 的左子树已遍历完毕，可进而往 A 的右子树 C 前进，以此类推，当 A 的右子树遍历完毕后，方可处理根结点 A，故可得到后序遍历的顺序为：G，D，B，H，I，E，F，C，A。

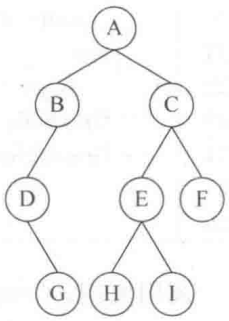


图 4.19

其程序除遍历输出部分不同外，其他与前序遍历程序完全相同。现给出输出部分代码如下。

```
1  void print(b_tree root)          //后序遍历
2  {
3    if(root!=NULL)
4    {
5      print(root->left);
6      print(root->right);
7      cout<<root->date<<' ';
8    }
9  }
```

二叉树的图形化显示

类似于树的数据结构，虽然通过前面的程序可以打印出遍历顺序，但毕竟过于抽象，如果能直观地在屏幕上显示出树的结构，那么对于初学者来说，程序的调试就会更加轻松方便。网上有很多关于图形化显示二叉树的讨论，例如调用绘图模式等，但这些技术过于复杂和专业，此处提供一种简单的“图形化”显示方法以方便程序的调试。其关键代码如下。

```
1  //需在程序起始位置加入#include <windows.h>
2  //函数形参中的 x,y 为输出结点的坐标,k=0,1,2 分别代表结点为根、左子、右子,space 控制输
3  出的树宽度
4
5  void GraphiShow(b_tree root,int x,int y,int k,int space)
6  {
7    if(root!=NULL)
8    {
9      //此段代码用于定位光标,可另写成函数调用的形式
10     HANDLE hOutput;
11     COORD location;
12     location.X=x;
13     location.Y=y;
14     hOutput=GetStdHandle(STD_OUTPUT_HANDLE);        //获得屏幕句柄
15     SetConsoleCursorPosition(hOutput,location);     //定位光标到坐标(x,y)处
16
```

```
17        if(k==1)
18            cout<<root->date<<"/";                    //输出表示左子树
19        else if(k==2)
20            cout<<"\\"<<root->date;                   //输出表示右子树
21        else
22            cout<<root->date;                         //输出根结点
23        GraphiShow (root->left,x-space,y+1,1,space/2);
24        GraphiShow (root->right,x+space,y+1,2,space/2);
25    }
26 }
```

调用实例：GraphiShow (root,40,3,0,20);

说明：由于屏幕一行是 80 个字符，所以 x 的起始位置设为 40，即根结点在中间显示。

该程序由于调用了 windows.h 头文件，因此仅能在 Windows 平台上运行，但对于初学者调试程序来讲，已经足够。

一个完整的程序如下。

```
1  //树的图形化显示
2  #include <iostream>
3  #include <windows.h>
4  using namespace std;
5
6  struct tree
7  {
8      struct tree *left;
9      int date;
10     struct tree *right;
11 };
12 typedef struct tree treenode;
13 typedef struct tree *b_tree;
14
15 b_tree insert(b_tree root,int node)              //插入结点
16 {
17     b_tree newnode;
18     b_tree currentnode;
19     b_tree parentnode;
20
21     newnode=(b_tree)malloc(sizeof(treenode));
22     newnode->date=node;
23     newnode->left=NULL;
24     newnode->right=NULL;
25
26     if(root==NULL)                               //第一个结点建立
27         return newnode;
28     else
29     {
30         currentnode=root;                        //存储当前结点
31         while(currentnode!=NULL)
32         {
```

```
33          parentnode=currentnode;                    //存储父结点
34          if(currentnode->date>node)
35            currentnode=currentnode->left;           //左子树
36          else
37            currentnode=currentnode->right;          //右子树
38        }
39        if(parentnode->date>node)
40          parentnode->left=newnode;
41        else
42          parentnode->right=newnode;
43      }
44      return root;
45    }
46
47    b_tree create(int * data,int len)                //建立二叉树
48    {
49      int i;
50      b_tree root=NULL;
51      for(i=1;i<=len;i++)
52        root=insert(root,data[i]);
53      return root;
54    }
55
56    void GraphiShow(b_tree root,int x,int y,int k,int space)
57    {
58      if(root!=NULL)
59      {
60                                                //此段代码用于定位光标,可另写成函数调用的形式
61        HANDLE hOutput;
62        COORD location;
63        location.X=x;
64        location.Y=y;
65        hOutput=GetStdHandle(STD_OUTPUT_HANDLE);        //获得屏幕句柄
66        SetConsoleCursorPosition(hOutput,location);     //定位光标到坐标(x,y)处
67
68        if(k==1)
69          cout<<root->date<<"/";                        //输出表示左子树
70        else if(k==2)
71          cout<<"\\"<<root->date;                       //输出表示右子树
72        else
73          cout<<root->date;                             //输出根结点
74        GraphiShow (root->left,x-space,y+1,1,space/2);
75        GraphiShow (root->right,x+space,y+1,2,space/2);
76      }
77    }
78
79    int main()
80    {
81      int n,i;
82      b_tree root=NULL;
```

```
    cin>>n;
    int node[n+1];
    for(i=1;i<=n;i++)
      cin>>node[i];
    root=create(node,n);
    GraphiShow (root,40,3,0,20);
    cout<<endl;
    system("pause");
}
```

程序在屏幕上显示结果类似于图 4.20 所示。

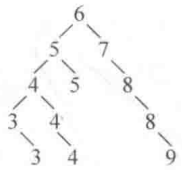


图 4.20

FBI 树

【题目描述】 FBI 树(fbi. c/cpp/pas)NOIP 2004

上古文明遗迹的探险任务最终决定交由墨老师带领完成。他们打开第一个宝藏的条件是解决 FBI 树问题。所谓 FBI 树的描述如下：

我们可以把由“0”和“1”组成的字符串分为三类：全“0”串称为 B 串，全“1”串称为 I 串，既含“0”又含“1”的串则称为 F 串。FBI 树是一种二叉树，它的结点类型也包括 F 结点、B 结点和 I 结点三种。由一个长度为 2^N 的“01”串 S 可以构造出一棵 FBI 树 T，递归的构造方法如下：

(1) T 的根结点为 R，其类型与串 S 的类型相同。

(2) 若串 S 的长度大于 1，将串 S 从中间分开，分为等长的左右子串 S1 和 S2；由左子串 S1 构造 R 的左子树 T1，由右子串 S2 构造 R 的右子树 T2。

现在给定一个长度为 2^N 的“01”串，请用上述构造方法构造出一棵 FBI 树，并输出它的后序遍历序列。

【输入文件】

输入文件 fbi. in 的第一行是一个整数 N($0\leqslant N\leqslant 10$)，第二行是一个长度为 2N 的“01”串。

【输出文件】

输出文件 fbi. out 包括一行，这一行只包含一个字符串，即 FBI 树的后序遍历序列。

【样例输入】

3

10001011

【样例输出】

IBFBBBFIBFIIIFF

【数据规模】

对于40%的数据，N≤2；

对于全部的数据，N≤10。

【算法分析】

如果用数组来描述，递归的算法比较难写，而利用指针建立二叉树，再后序遍历很简单。

主串的长度为2^n，逐层递减，是一种二分结构，由于要保存结点以便后序遍历，使用二叉树结构，共需要使用$2^{(n+1)}$个结点，树的高度为lgn。

可以使用最后一层的结点来保存2^n个值，则它们的父亲结点的字符值就已经由左右儿子的'B'，'I'决定了，故不用保存串，只需要记录字符值。

参考程序如下。

```
1   //FBI树
2   #include <iostream>
3   using namespace std;
4   const int NMAX = ( 1 << 11 ) + 1;
5
6   struct node
7   {
8       char c;
9       int value;
10      int leftson, rightson;
11  };
12  int N, i, j;
13  node tree[NMAX];
14
15  char check( int left , int right )
16  {
17      if( tree[left].c == 'B' && tree[right].c == 'B' )
18          return 'B';
19      else if( tree[left].c == 'I' && tree[right].c == 'I' )
20          return 'I';
21      else
22          return 'F';
23  }
24
25  void LRD( int root )
26  {
27      if( root != -1 ){
28          LRD( tree[root].leftson );
29          LRD( tree[root].rightson );
30          printf( "%c", tree[root].c );
31      }
32  }
33
34  int main()
```

```
35  {
36      freopen("fbi.in","r",stdin);
37      freopen("fbi.out","w",stdout);
38      scanf( "%d", &N );
39      for( i = ( 1 << N ) ; i < ( 1 << ( N + 1 ) ) ; i++ )
40      {
41          scanf( "%1d", &tree[i].value );
42          tree[i].leftson = -1;
43          tree[i].rightson = -1;
44          if( tree[i].value )
45              tree[i].c = 'I';
46          else
47              tree[i].c = 'B';
48      }
49      int temp = N - 1 ;
50      while( temp >= 0 )
51      {
52          for( i = 1 << temp ; i < 1 << ( temp + 1 ) ; i++ )
53          {
54              tree[i].leftson = 2 * i;
55              tree[i].rightson = 2 * i + 1;
56              tree[i].c = check( 2 * i , 2 * i + 1 );
57          }
58          temp -- ;
59      }
60      LRD( 1 );
61      printf( "\n" );
62      return 0;
63  }
```

已知前序中序求后序

常见的遍历问题一般是已知中序和另一种遍历的情况下求第三种遍历。因为仅知道前序和后序遍历是无法确定一棵树的，所以“已知前序后序求中序遍历”的问题放在最后讨论。

先来看“已知前序中序求后序”。

前序：ABDEFGCH　　中序：DFEGBAHC

显然地，A 点一定是该树的根结点(在前序遍历中最先访问)。我们再从中序遍历中找到 A 点，因为中序遍历先遍历了左子树才访问根结点，所以 DFEGB 是 A 结点左子树的中序遍历，HC 是 A 结点右子树的中序遍历，如图 4.21 所示。

图　4.21

由于前序遍历也是先左子树后右子树，因此前序中 A 之后的 BDEFG 则是 A 结点左子树的前序遍历。同样地，CH，HC 也分别是 A 结点右子树的前、中序遍历。于是问题转化为已知 A 结点左子树的前序中序求后序和已知 A 结点右子树的前序中序求后序，如图 4.22 所示。

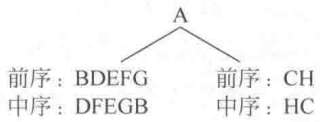


图　4.22

显然可以递归求解，并在这两个过程结束后输出根结点 A（符合后序遍历的顺序）。

下面是“已知前序中序求后序”的主要代码。

```
//已知前序中序求后序
//a b 分别是前序和中序遍历
void hwg(string a,string b)
{
int i,ll,lr;
for(i=0;i<b.length();i++)
  if(a[0]==b[i])
  {
    ll=i;
    lr=b.length()-1-ll;                          //计算出左右子树的长度
    if(ll)
    {
      string part1(a,1,ll),part2(b,0,ll);        //取出左子树
      hwg(part1,part2);                          //递归求左子树
    }
    if(lr)
    {
      string part3(a,1+ll,lr),part4(b,1+ll,lr);  //取出右子树
      hwg(part3,part4);                          //递归求右子树
    }
    cout<<a[0];                                  //按照后序遍历，最后输出根结点
    break;
  }
}
```

已知后序中序求前序

“已知后序中序求前序”的方法与此类似，无须多说，唯一需要注意的是根据前序遍历的顺序，程序必须先输出根结点再访问左右子树。

已知前序后序求中序

熟悉了这一过程，我们再来看“已知前序后序求中序”。

前序：ABDEFGCH　　后序：FGEDBHCA

显然，A 是根结点。我们看到，前序中 B 在 A 后，后序中 C 在 A 前。那该结点左右子树肯定都有，且子树根结点分别为 B 和 C，如图 4.23 所示。

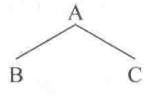


图　4.23

先来看 A 的左子树，前序：BDEFG　　后序：FGEDB

用同样的方法分析，却发现前序中 B 点之后是 D 点，后序中 B 点之前还是 D 点。左右子树结点不可能相同，唯一的解释是，B 只有一棵子树，且其根结点为 D。然而我们无法判定这棵树究竟是左子树还是右子树。事实上，这是都有可能的。这也是仅知道前后序遍历无法确定一棵树的唯一原因。不过我们可以分两种可能情况来讨论，如图 4.24 所示。

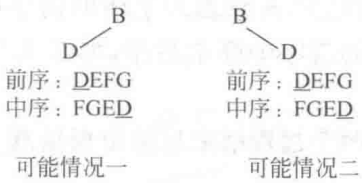


图　4.24

在这两种可能情况的分类讨论中，B 的子树 D 的前序为 DEFG，后序为 FGED，同理，可以发现 D 也只有一棵子树，其子树根结点为 E，而 E 可能是左子树，也可能是右子树，这又需要分类讨论，如图 4.25 所示。

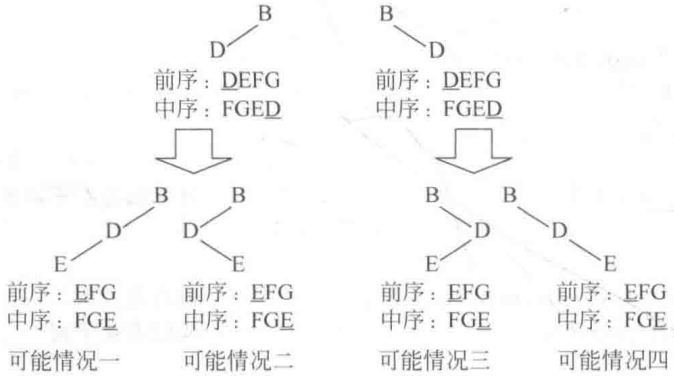


图　4.25

依次按照这种方式处理完毕，我们就可以输出所有可能的中序遍历。

具体程序实现时，我们以字符串来表示所有可能的情况，如样例所对应的字符串应为图 4.26 所示。

图　4.26

并列的大括号表示可能的两种情况，每次碰到大括号时任选一个，最后就得到一个可能的中序遍历。那么如何任选或者说跳转呢？我们可以采用数组的方式来记录该字符串需要跳转的位置，数组 cho[]各元素初始值为－1，对于任意并列的两对大括号，处理规则为：第一个大括号处记录第三个大括号的位置，第二、四个大括号处记录第四个大括号所在位置后的第一个位置，第三个大括号处值不变，表示不应跳转。标记好的数组见图 4.27。

处理完毕后，就可以用深度优先搜索算法(DFS)来输出所有结果了，操作规则如下：

(1) 遇到字母就记录下来。

(2) 遇到左大括号(且该位置数组元素值非－1)，就分开两条路。

① 直接向前走一步。

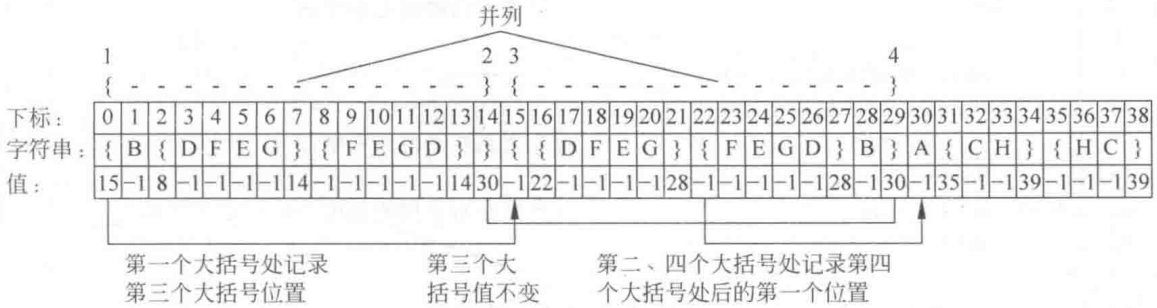


图　4.27

② 跳到该位置数组元素所记录的位置处。

(3) 遇到右括号就移动到该位置数组元素所记录的位置。

下面是完整的代码。

```
//已知前序中序求后序
#include<iostream>
#include<string>
#define ANSN 10000
using namespace std;

string rx78;                          //用来记录上例中的那个包含括号和字母的字符串
int final,cho[ANSN+1];                //用来跳转的数组

void hwg(string a,string b)
{
  int i,mark,ll,lr,len=a.length();
  if(len>1)                           //如果不止有一个结点
  {
    if(a[1]==b[len-2])                //如果符合 ABXXX,XXXBA 的形式,就要分类处理
    {
      string p1(a,1,len-1),p2(b,0,len-1); //取出子树
                                      //可能性 1,该树是右子树,先访问根结点
      rx78+='{';
      mark=rx78.length()-1;           //记录下左括号的位置
      rx78+=a[0];
      hwg(p1,p2);                     //打下标记后处理子树
      rx78+='}';
      cho[mark]=rx78.length();        //给左括号赋下跳转的位置
      mark=rx78.length()-1;           //记录下右括号的位置

                                      //可能性 2,该树是右子树,后访问根结点
      rx78+='{';
      hwg(p1,p2);                     //同上
      rx78+=a[0];
      rx78+='}';
      cho[rx78.length()-1]=rx78.length();
      cho[mark]=rx78.length();        //给两个右括号赋上跳转的位置
    }
```

```
35        else                                    //如果可以确定左右子树
36        {
37          for(i=0;i<len-2;i++)
38            if(a[1]==b[i])
39              break;                            //先找到左子树的根结点
40          ll=i+1;
41          lr=len-1-ll;                          //计算左右子树的长度
42          if(ll)
43          {
44            string part1(a,1,ll),part2(b,0,ll);//取出左子树
45            hwg(part1,part2);                   //递归求左子树
46          }
47          rx78+=a[0];                         //中序遍历,根结点在访问左子树后,访问右子树前访问
48          if(lr)
49          {
50            string part3(a,1+ll,lr),part4(b,ll,lr);  //取出右子树
51            hwg(part3,part4);                   //递归求右子树
52          }
53        }
54      }
55      else                                      //如果只有一个结点
56        rx78+=a[0];
57    }
58
59    void dfs(string have,int pos)
60    {
61      char ty=rx78[pos];
62      if(pos>=final)                            //如果深搜到尽头
63        cout<<have<<endl;
64      else if(ty=='{')                          //当碰到左括号时
65      {
66        if(cho[pos]==-1)                        //如果这个左括号不能跳转,就只能向前走一格
67          dfs(have,pos+1);
68        else                                    //如果可以跳转
69        {
70          dfs(have,pos+1);                      //要么直接向前走一格
71          dfs(have,cho[pos]);                   //要么跳转
72        }
73      }
74      else if(ty=='}')
75        dfs(have,cho[pos]);                     //碰到右括号都跳转
76      else
77      {
78        have+=rx78[pos];
79        dfs(have,pos+1);                        //如果碰到字符就存入,并继续深搜
80      }
81    }
82
83    int main()
84    {
```

```
    int i;
    string front,back;
    cin>>front>>back;
    for(i=0;i<=ANSN;i++)
      cho[i]=-1;
    hwg(front,back);
    final=rx78.length();
    dfs("",0);
    return 0;
}
```

最后值得一提的是，如果仅要计算可能的中序遍历的数量而非输出所有的答案，则要简单许多，前面我们已经知道：只有前序、后序分别出现 ABXXXXX，XXXXXBA 的情况时，才需要分两种情况讨论。所以我们可以先定义可能的情况数 s=1，枚举前序遍历中的每一个点，看看它在前序遍历中之后的一个点，和在后序遍历中之前的一个点是否相同，如果相同，s=s×2。在之前的例子里，我们看到 B，D，C 三个点符合条件，所以可能的情况数是 2 的三次方即 8。

二叉查找树

二叉查找树(Binary Search Tree)，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、右子树也分别为二叉排序树。

上述性质被称为 BST 性质，如图 4.28 所示。

二叉排序树的查找过程和次优二叉树类似，通常采取二叉链表作为二叉排序树的存储结构。中序遍历二叉排序树可得到一个关键字的有序序列，一个无序序列可以通过构造一棵二叉排序树变成一个有序序列，构造树的过程即为对无序序列进行排序的过程。每次插入的新的结点都是二叉排序树上新的叶子结点，在进行插入操作时，不必移动其他结点，只需改动某个结点的指针，由空变为非空即可。搜索、插入、删除的复杂度等于树高、期望 O(logn)、最坏 O(n)(数列有序，树退化成线性表)。

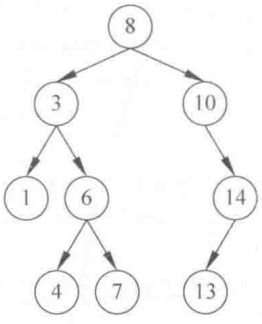


图　4.28

虽然二叉排序树的最坏效率是 O(n)，但它支持动态查询，且有很多改进版的二叉排序树可以使树高为 O(logn)，如 SBT、AVL、红黑树等。故不失为一种好的动态排序方法。

在下面的介绍当中，对于树上某结点，data[x]为结点的数据，left[x]为结点的左孩子，right[x]为结点的右孩子，p[x]为结点的父结点，root 为树根。

下面查找、插入和删除的过程通过伪代码来介绍。

查找

在二叉查找树中寻找一个值时，首先从根结点出发，每次判断当前的结点是否与要查找

的结点相等，在不相等的情况下，如果当前结点的值小于寻找的值，则查找其右子树，否则查找其左子树。

这个过程的伪代码如下。

```
//最开始的寻找 tree_search(root,k)
tree_search( x, k)k 为要查找的值
{
    if(x == NULL or k == data[x])     /*查找完了整棵树或查找到要找的结点*/
        return x;
    if(k > data[x])
        tree_search(left[x],k);
    else
        tree_search(right[x],k);
}
```

【题目描述】 追溯过去

修罗王和邪狼自与天顶星人的最后一战后，就失去了踪影，这十年来，无数的魔法师努力想还原当时的情形。有魔法师根据量子多宇宙理论，提出了一种搜索方法，该理论是基于这么一个假想：当宇宙面临一种量子选择时，它就分裂成两个不同的宇宙，我们可以将此看作一棵二叉树，每个结点数据大于左子结点的数据，且要小于右子结点的数据。请根据该二叉树的特征，查找出要找的关键结点。

参考程序如下。

```
//二叉树二分查找法
#include <stdio.h>
#include <stdlib.h>
struct tree
{
  struct tree *left;
  int date;
  struct tree *right;
};
typedef struct tree treenode;
typedef struct tree *b_tree;

b_tree create(int *node,int position)          //递归建立二叉树
{
  b_tree newnode;
  if(node[position]==0 || position>15)
    return NULL;
  else
  {
    newnode=(b_tree)malloc(sizeof(treenode));
    newnode->date=node[position];
    newnode->left=create(node,2*position);
    newnode->right=create(node,2*position+1);
    return newnode;
  }
```

```
26  }
27
28  void print(b_tree root)                          //中序遍历
29  {
30     if(root!=NULL)
31     {
32        print(root->left);
33        printf("%d ",root->date);
34        print(root->right);
35     }
36  }
37
38  b_tree search(b_tree point,int locate)           //二分查找
39  {
40     while(point!=NULL)
41     {
42        if(point->date==locate)                    //如找到,则返回地址
43           return point;
44        else
45           if(point->date>locate)
46              point=point->left;                   //往左子树找
47           else
48              point=point->right;                  //往右子树找
49     }
50     return NULL;                                  //未找到结点,则返回 NULL
51  }
52
53  int main()
54  {
55     int i,locate;
56     b_tree root=NULL;
57     b_tree point=NULL;
58     int node[16]={0,5,2,9,1,4,7,0,0,0,3,0,6,8,0,0};
59     root=create(node,1);
60     print(root);
61     printf("\n 请输入欲查找的结点值:  ");
62     scanf("%d",&locate);
63
64     point=search(root,locate);                    //调用二分查找函数
65     if(point!=NULL)
66        printf("The value is [%d]\n",point->date);
67     else
68        printf("Not find!\n");
69
70     system("pause");
71  }
```

插入

二叉查找树的插入和查找很相似,从根结点开始,我们每次将要插入的值与当前结点做比较,如果要插入的值小于当前结点,则向当前结点左子树走,否则向右,沿树下降直到叶子

结点。然后把要插入的值变成结点连接到这个叶结点，使其成为这个叶子结点的左孩子或者右孩子(比较两者的值进行判断)。下面是二叉查找树插入的伪代码，假设这时已经建立了一个树结点存储要插入的数据，z 就是这个结点的下标。

```
tree_insert( z )
{
步骤 1: 确定要插入的结点在树中的位置
    x = root 从树根开始扫描
    while( x 指针对应的结点有值)
      {
          y = x;y 为 x 的父结点
          if(data[z] < data[x])
              x = left[x];
          else
              x = right[x];
}

步骤 2: 确定要插入的结点和树上结点的关系
   p[z] = y;
   if( y 不存在) 如果这棵树 T 是空的
      root = z ;z 就是该树的结点
   else if(data[z] < data[y])
        left[y] = z;
   else
        right[y] = z;
}
```

删除

二叉查找树的删除过程有点复杂，我们这里分三种情况讨论。这里的方法讨论如何删除 z 指针指向的结点。在这之前，我们先了解一下如何求在中序遍历下树的前驱和后继，后面所谓树的前驱与后继，都是说的中序遍历之下的顺序。树上某个结点的前驱与后继就是中序遍历下这个结点的前面遍历的结点和后面遍历的结点。

如果要寻找一个结点的后继，我们首先考察一个结点的右孩子是否存在，若它的右孩子存在，那么这个后继一定就在它的右子树上，而且必然是右子树上的最左(小)结点(显然是这样的，因为你求一棵二叉树的中序遍历的时候，第一个遍历到的结点必然是这棵树上的最左(小)结点)。

另外，如果这个结点的右孩子不存在，且这个结点存在有一个后继，那么这个后继一定是这个结点的最低祖先结点(想一下中序遍历的过程就明白了)。

下面的过程显示了如何求一个结点的后继。

```
tree_successor(x)
{
如果这个结点存在有右子树，那么返回右子树上的最左结点
if right[x] != NULL
     return tree_minimum(right[x]);

```

```
如果这个结点没有右子树,且这个结点有后继,那么求这个结点的最低祖先
    y  = p[x];
    while(y != NULL and x == right[y])
     {
        x = y;
        y = p[y];
     }
    return y;
}
```

假如现在要删除结点 z,那么可能出现下列三种情况:

如果这个结点没有孩子,那么则直接将 z 删除,然后清除其父结点指向它的指针就可以了。

如果 z 只有一个孩子,那么就把 z 的父结点和 z 的关系(z 是其父结点的左孩子或者右孩子)指向 z 的那个孩子,然后再将 z 删除,最后别忘了将 z 的那个孩子的父结点变为 z 的父结点。

如果 z 有两个孩子,那么把这个结点的后继 y 从树上取下来(就是删除 y,但是保留 y 的数据),然后用 y 的数据替换掉 z 的数据就可以了。

```
tree_delete( z )
{
    if(left[z] == NULL or right[z] == NULL)
        y  = z;如果该结点没有孩子,或只有一个孩子,就直接删除该结点
    else                        //y 赋值为 z 的后继
        y  = tree_successor(z);
如果 z 有孩子,则寻找 z 的后继
x 为 y 的左孩子或右孩子(如果有的话)
    if(left[y] != NULL)
      x = left[y];
    else
      x = right[y];
如果 x 存在,
    if(x != NULL)
      P[x] = P[y];改变 x 的父结点

    if(p[y] == NULL)删除的点是根结点
      root = x;
    else if y == left[p[y]] y 是其父结点的左孩子
      left[p[y]] = x;
    else y 是其父结点的右孩子
      right[p[y]] = x;

    if( y != z)把 z 赋上 y 的数据,然后删除 y
    {
      data[z] = data[y];
      还有其他数据也在这一步复制到 z
     }
    return y;                   //删除 y 指针对应的结点
```

注意以上的算法中，y 不可能有两个定点，因为在 y 作为 z 的后继时，y 必然是右子树上最左结点，如果 y 有两个孩子，那么这个后继必然还在更左的地方，所以 y 至多只有一个定点。

二叉查找树样例

这是一个演示二叉查找树基本操作的样例，这里使用数组存储二叉树结点但不同于前面用数组构造的二叉树，这里是用数组构造了链表的结构。如同第一章数组链表的结构。

```
1  //一个简单的二叉查找树的例子，用数组模拟二叉树的结构，使用栈优化
2  #include<iostream>
3  #define MAX_TREE_NODE 1000000          /* 树中最多结点数 */
4  using namespace std;
5
6  /* data 为数据 lc, rc 为结点的左右孩子 */
7  int
8  data[MAX_TREE_NODE], p[MAX_TREE_NODE], lc[MAX_TREE_NODE], rc[MAX_TREE_
   NODE];
9  int root = -1;
10
11 /* 用于提供数组空位的栈 */
12 int stack[MAX_TREE_NODE];
13 int top = 0;
14
15 /* 搜索树上值为 k 的结点，并返回其在数组中的位置，如果不存在，则返回-1 */
16 int tree_search(int x, int k)
17 {
18   /* 这里使用较快的迭代方式 */
19   while(x != -1 and data[x] != k)
20     {
21       if(k < data[x])
22     x = lc[x];
23       else
24     x = rc[x];
25     }
26   return x;
27 }
28
29 /* 在树中插入一个结点 z，这个结点在 pre_insert()过程中已经赋值 */
30 void tree_insert(int z)
31 {
32   /* insert node */
33   int x = root, y = -1;
34   while(x != -1)
35     {
36       y = x;
37       if(data[z] < data[x])
38     x = lc[x];
```

```
39          else
40        x = rc[x];
41        }
42
43    p[z] = y;
44
45    if(y == -1)
46        root = z;
47    else if (data[z] < data[y])
48        lc[y] = z;
49    else
50        rc[y] = z;
51
52    return ;
53  }
54
55  /*在数组中找到一个空位并且给这个结点赋值,将其插入到树中*/
56  void pre_insert(int ins_data)
57  {
58    int z;
59    z = stack[top++];
60    data[z] = ins_data;
61    tree_insert(z);
62    return ;
63  }
64
65  /*找树上的最小值*/
66  int tree_minimum(int x)
67  {
68    while(lc[x] != -1 )
69        x = lc[x];
70    return x;
71  }
72
73  /*求某结点的后继*/
74  int tree_successor(int x)
75  {
76    /*若该结点存在右子树,则寻找其右子树最左结点*/
77    if(rc[x] != -1)
78        return tree_minimum(rc[x]);
79    /*否则查找其最低祖先,查找直到遇到某个结点是其父结点的左孩子结点为止*/
80    int y = p[x];
81    while(y != -1 and x == rc[y])
82        {
83            x = y;
84            y = p[y];
85        }
86    return y;
87  }
88
```

```
89  /*删除数组z位置上的结点,这个结点在pre_delete()过程中已经确定了*/
90  void tree_delete(int z)
91  {
92    int x,y;
93    /*如果这个结点至多有一个孩子,则删除该结点*/
94    if(lc[z] == -1 or rc[z] == -1)
95      y = z;
96    else/*否则寻找其后继*/
97      y = tree_successor(z);
98    /*将x指向y的左孩子或右孩子(如果有的话)*/
99    if(lc[y] != -1)
100     x = lc[y];
101   else
102     x = rc[y];
103
104   /*如果y有孩子,那么则将在上面步骤中指定过的孩子的父结点指向y的父结点*/
105   if(x != -1)
106     p[x] = p[y];
107
108   if(p[y] == -1)/*删除的结点为根结点*/
109     root = x;
110   else if(y == lc[p[y]])/*删除的结点y为y的左孩子*/
111     lc[p[y]] = x;
112   else/*删除的结点y为y的右孩子*/
113     rc[p[y]] = x;
114
115   if(y != z)/*将y的值赋给z*/
116     data[z] = data[y];
117
118   /*delete node y*/
119   data[y] = -1;
120   lc[y] = -1;
121   rc[y] = -1;
122   p[y] = -1;
123   stack[--top] = y;
124   return ;
125 }
126
127 /*删除之前的操作,在树中找到删除结点所在的位置*/
128 void pre_delete(int data)
129 {
130   int k;
131   k = tree_search(root,data);
132   if(k != -1)
133     tree_delete(k);
134   return ;
135 }
136
137 /*初始化所有结点,默认空值为-1*/
138 void init()
```

```
139 {
140   int i;
141   for(i = 0 ;i < MAX_TREE_NODE; ++ i)
142     {
143       data[i] = rc[i] = lc[i] = p[i] = -1;
144       stack[i] = i;
145     }
146   return ;
147 }
148
149 /* 以中序遍历的方式显示二叉树中所有的结点及每个结点的信息 */
150 void display(int x)
151 {
152   if(x == -1) return ;
153   if(lc[x] != -1)
154     display(lc[x]);
155   cout<<x<<" value : "<<data[x]<<" left: "<<lc[x]<<" right: "<<rc[x]<<
    " parent : "<<p[x]<<"\n";
156
157   if(rc[x] != -1)
158     display(rc[x]);
159   return ;
160 }
161
162 /* 演示部分 */
163 int main()
164 {
165   int ins,tmp;
166
167   cout<<"please insert a instruction:\n";
168   cout<<"1.insert a value\n";
169   cout<<"2.delete a value\n";
170   cout<<"3.display the tree in middle order\n";
171
172   init();
173
174   while(cin>>ins)
175     {
176       //cout<<"top : "<<top<<" val : "<<stack[top]<<"\n";
177       switch(ins)
178     {
179     case 1:/* 输入 1,插入一个值 */
180       cin>>tmp;
181       cout<<"please input insert data \n";
182       pre_insert(tmp);
183       display(root);
184       break;
185
186     case 2:/* 输入 2,删除一个值,输入的删除值 */
187       cin>>tmp;
```

```
188            cout<<"please input the data need to delete\n";
189            pre_delete(tmp);
190            display(root);
191            break;
192
193        case 3:/* 显示二叉树中的所有结点 */
194            display(root);
195            break;
196
197        default :
198            return 0;
199        }
200        }
201    return 0;
202 }
```

最优二叉树

最优二叉树又叫哈夫曼树，这种树的所有叶子结点都带有权值，从中构造出带权路径长度最短的二叉树，即最优二叉树。

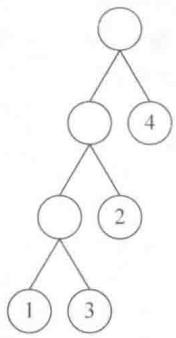
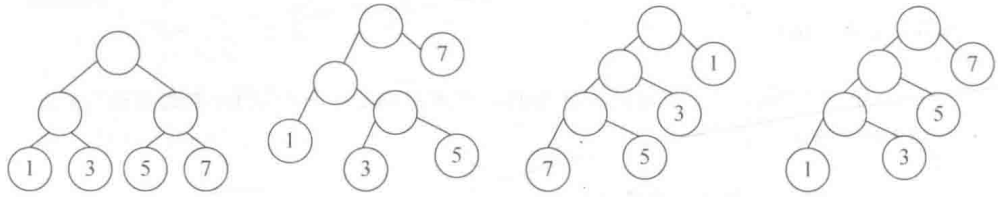


图 4.29

设二叉树具有 n 个带权值的叶子结点，那么从根结点到各个叶子结点的路径长度与相应结点权值的乘积之和，叫作二叉树的带权路径长度。例如，图 4.29 所示的二叉树，它的带权路径长度值为 $1\times3+3\times3+2\times2+4\times1=20$。

如果给定一组具有确定权值的叶子结点，那么可以构造出不同的带权二叉树，它们的带权路径长度并不相同，我们把其中具有最小带权路径长度的二叉树称为哈夫曼树。

图 4.30 中的四棵二叉树具有相同的叶子结点，它们的带权路径长度各不一样。可以证明，图 4.30 中的 D 树所示的二叉树是一棵最优二叉树。



A.1*2+3*2+5*2+7*2=32　B.1*2+3*3+5*3+7*1=33　C.7*3+5*3+3*2+1*1=43　D.1*3+3*3+5*7+7*1=29

图 4.30

构造最优二叉树

根据最优二叉树的定义，一棵二叉树要想使其带权路径长度值最小，就必须使权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根结点。那么如何构造一棵哈夫

曼树呢？其方法如下：

(1) 由给定的 n 个权值构造 n 根只有一个叶子结点的二叉树，从而得到一个二叉树的集合。

(2) 在二叉树集合中选取根结点权值最小和次小的两棵二叉树作为左右子树构造一棵新的二叉树，这棵新的二叉树根结点的权值为其左右子树根结点权值之和。

(3) 在二叉树集合中删除作为左右子树的两棵二叉树，并将新建立的二叉树加入到集合中。

(4) 重复步骤(2)和(3)，直到剩下一棵二叉树。

请看图 4.31 所示的例子。

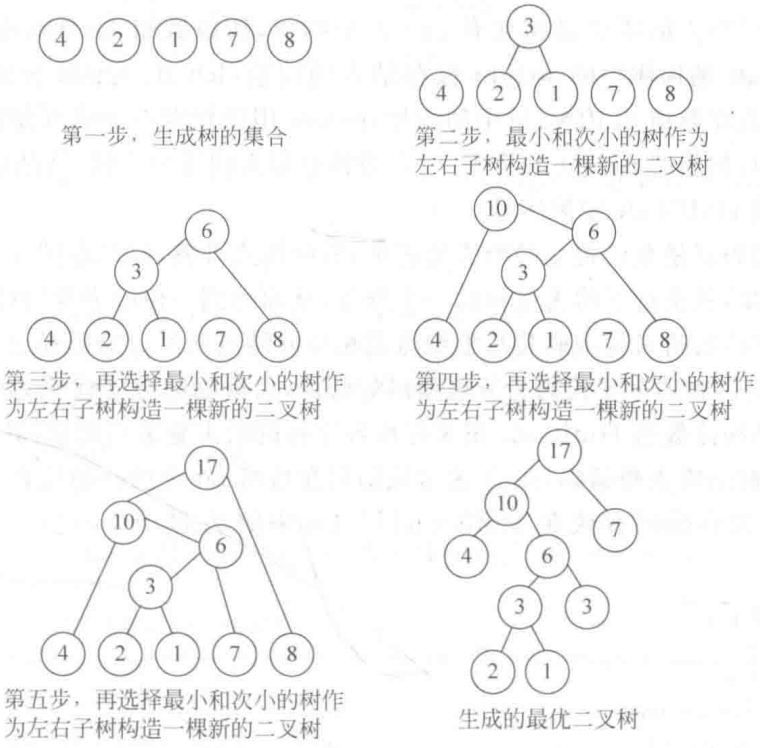
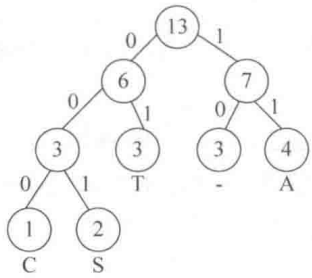


图　4.31

哈夫曼编码

【题目描述】 哈夫曼编码

探险者之间的联系方式采用哈夫曼编码方式，所谓哈夫曼编码方式是利用哈夫曼树(最优二叉树)构造的用于通信的二进制编码。例如，有一段电文“CAST_ TAT_ A_SA”，统计电文中字母的频度 f(C)=1，f(S)=2，f(T)=3，f(_)=3，f(A)=4。用频度{1，2，3，3，4}为权值生成的哈夫曼树，并在每个叶子上注明对应的字符。树中从根到每个叶子都有一条路径，对路径上的各分支约定指向左子树根的分支表示“0”码，指向右子树的分支表示“1”码，取每条路径上的“0”或“1”的序列作为和各个叶子对应的字符的编码，这就是哈夫曼编码。对应图 4.32 左图的哈夫曼树和字符编码见图 4.32 右图。



字符	编码
C	000
S	001
T	01
_	10
A	11

图 4.32

设置一个结构数组 HuffNode 保存哈夫曼树中各结点的信息。根据二叉树的性质可知，具有 n 个叶子结点的哈夫曼树共有 2n－1 个结点，所以数组 HuffNode 的大小设置为 2n－1。HuffNode 结构体中的 weight 保存结点的权值，lchild，rchild 分别保存该结点的左、右孩子的结点在数组 HuffNode 中的序号，parent 用于判定一个结点是否已加入到要建立的哈夫曼树中，初始时 parent 的值为－1。当结点加入到树中去时，该结点 parent 的值为其父结点在数组 HuffNode 中的序号。

求哈夫曼编码就是在已建立的哈夫曼树中，从叶结点开始，沿结点的双亲链域回退到根结点，每回退一步，就走过了哈夫曼树的一个分支，从而得到一位哈夫曼码值。

由于一个字符的哈夫曼编码是从根结点到相应叶结点所经过的路径上各分支所组成的 0，1 序列，因此先得到的分支代码为所求编码的低位，后得到的分支代码为所求编码的高位码。设置一个结构体数组 HuffCode 用来存放各字符的哈夫曼编码信息，其中 bit 为一维数组，用来保存字符的哈夫曼编码，start 表示该编码在数组 bit 中的开始位置。所以，对于第 i 个字符，它的哈夫曼编码存放在 HuffCode[i]. bit 中的从 HuffCode[i]. start 到 n 的 bit 位中。

参考程序如下。

```
1   //哈夫曼编码
2   #include <iostream>
3   using namespace std;
4
5   #define MAXN 1000
6   #define MAXLEAF      30
7   #define MAXNODE      MAXLEAF * 2 -1
8
9   typedef struct                          //编码结构体
10  {
11    int bit[100];                         //保存哈夫曼编码
12    int start;                            //编码的开始位置
13  }HCode;
14
15  typedef struct                          //结点结构体
16  {
17    int weight;
18    int parent;
19    int lchild;
```

```
20      int rchild;
21  }HNode;
22
23  HNode HuffNode[MAXNODE];          //定义一个结点结构体数组
24  HCode HuffCode[MAXLEAF];          //定义一个编码结构体数组
25
26  void CreatHuffmanTree(HNode HNode[MAXNODE],int n)  //构造一棵哈夫曼树
27  {
28      int i, j;
29      int m1, m2;                   //构造树过程中两个最小权值结点的权值
30      int x1, x2;                   //构造树过程中两个最小权值结点在数组中的序号
31
32      for (i=0; i<n-1; i++)         //循环构造 Huffmantree
33      {
34          m1=m2=10000;              //m1,m2 中存放无父结点且结点权值最小的两个结点
35          x1=x2=0;
36
37          for (j=0; j<n+i; j++)     //找出所有结点中权值最小、无父结点的两个结点,并
38                                      合并为一棵二叉树
39
40          {
41              if (HNode[j].weight < m1 && HNode[j].parent==-1)
42              {
43                  m2=m1;
44                  x2=x1;
45                  m1=HNode[j].weight;
46                  x1=j;
47              }
48              else if (HNode[j].weight < m2 && HNode[j].parent==-1)
49              {
50                  m2=HNode[j].weight;
51                  x2=j;
52              }
53          }
54          //设置找到的两个子结点 x1、x2 的父结点信息
55          HNode[x1].parent  = n+i;
56          HNode[x2].parent  = n+i;
57          HNode[n+i].weight = HNode[x1].weight + HNode[x2].weight;
58          HNode[n+i].lchild = x1;
59          HNode[n+i].rchild = x2;
60      }
61  }
62
63  void CreatHuffmanCode(HNode HuffNode[MAXNODE],int n)
64  {
65      HCode cd;                     //定义一个临时变量来存放求解编码时的信息
66      for (int i=0; i < n; i++)
67      {
68          cd.start = n-1;
69          int c = i;
```

```
70        int p = HuffNode[c].parent;
71        while (p != -1)                    /* 父结点存在 */
72        {
73          if (HuffNode[p].lchild == c)
74            cd.bit[cd.start] = 0;
75          else
76            cd.bit[cd.start] = 1;
77          cd.start--;                      /* 求编码的低一位 */
78          c=p;
78          p=HuffNode[c].parent;            /* 设置下一循环条件 */
80        }
81        for (int j=cd.start+1; j<n; j++)  //保存求出的每个叶结点的哈夫曼编码和编码的起始位
82
83
84        {
85          HuffCode[i].bit[j] = cd.bit[j];
86        }
87        HuffCode[i].start = cd.start;
88    }
89  }
90
91  int init(HNode HNode[],int n)            //初始化
92  {
93    for (int i=0; i<2*n-1; i++)
94    {
95      HNode[i].weight = 0;
96      HNode[i].parent =-1;
97      HNode[i].lchild =-1;
98      HNode[i].lchild =-1;
99    }
100   for (int i=0; i<n; i++)
101   {
102     cout<<"请输入第"<<i<<"个结点的权重";
103     cin>>HNode[i].weight;
104   }
105 }
106
107 int main()
108 {
109   int i, j,n;
110   cout<<"请输入字符个数 n:\n";
111   cin>>n;
112   init(HuffNode,n);                      //初始化
113   CreatHuffmanTree(HuffNode, n);         //建立哈夫曼树
114   CreatHuffmanCode(HuffNode, n);         //生成哈夫曼编码
115
116   for (i=0; i<n; i++)                    //输出哈夫曼编码
117   {
118     cout<<i<<"的哈夫曼编码为:";
119     for (j=HuffCode[i].start+1; j < n; j++)
```

```
120            cout<<HuffCode[i].bit[j];
121        cout<<"\n";
122    }
123    getchar();
124    getchar();
125    return 0;
126 }
```

表达式处理

在二叉树的应用当中，最常见的就是表达式处理。由于表达式的处理需考虑计算的优先级，而二叉树的左右子树也有顺序之分，故若能将表达式数据整理成二叉查找树，则可用来方便处理表达式。

例如中序表达式如下：

4 * 8+6/2

将运算值当做叶结点，运算符号当做非终端结点，并考虑到运算的优先级，其步骤如图 4.33 所示。

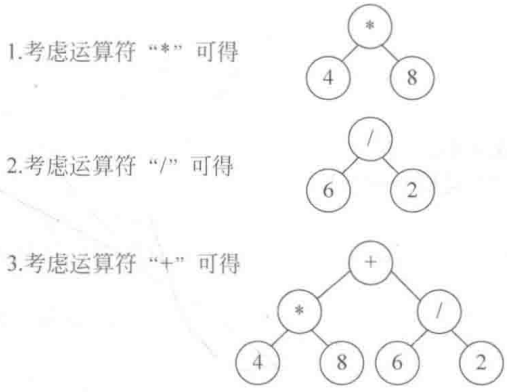


图 4.33

此二叉树当我们用前序、中序、后序遍历时，其结果如下。

前序：+ * 48/62

中序：4 * 8+6/2

后序：48 * 62/+

我们可以看到，这正好是前序表达式、中序表达式、后序表达式。

参考程序如下。（不含括号，输入的每个数字不超过两位数）

```
1  //简单表达式处理
2  #include<iostream>
3  using namespace std;
4
5  typedef struct node
6  {
7      char data;
```

```
8     struct node *lchild;
9     struct node *rchild;
10  }BTNode;
11
12  BTNode *CRTree(char s[],int i,int j)
13  {
14    int k,posi,plus=0;
15    BTNode *p;
16    if(i==j)                          //当只有一个数字时
17    {
18      p=(BTNode *)malloc(sizeof(BTNode));
19      p->data=s[i];
20      p->lchild=NULL;
21      p->rchild=NULL;
22      return p;
23    }
24    for(k=i;k<=j;k++)
25    {
26      if(s[k]=='+'||s[k]=='-')
27      {
28        plus++;
29        posi=k;
30      }
31    }
32    if(plus==0)
33      for(k=i;k<=j;k++)
34        if(s[k]=='*'||s[k]=='/')
35        {
36          plus++;
37          posi=k;
38        }
39
40    if(plus!=0)
41    {
42      p=(BTNode *)malloc(sizeof(BTNode));
43      p->data=s[posi];
44      p->lchild=CRTree(s,i,posi-1);
45      p->rchild=CRTree(s,posi+1,j);
46      return p;
47    }
48  }
49
50  void DispPre(BTNode *T)           //打印前序表达式
51  {
52    if(T!=NULL)
53    {
54      printf("%c ",T->data);
55      DispPre(T->lchild);
56      DispPre(T->rchild);
57    }
```

```
58  }
59
60  void DispIn(BTNode *T)              //打印中序表达式
61  {
62    if(T!=NULL)
63    {
64      DispIn(T->lchild);
65      printf("%c ",T->data);
66      DispIn(T->rchild);
67    }
68  }
69
70  void DispPost(BTNode *T)            //打印后序表达式
71  {
72    if(T!=NULL)
73    {
74      DispPost(T->lchild);
75      DispPost(T->rchild);
76      printf("%c ",T->data);
77    }
78  }
79
80  int main()
81  {
82    char s[20];
83    BTNode *T;
84    gets(s);
85    T=CRTree(s,0,strlen(s)-1);
86    printf("\n前序表达式为:\n");
87    DispPre(T);
88    printf("\n中序表达式为:\n");
89    DispIn(T);
90    printf("\n后序表达式为:\n");
91    DispPost(T);
92    getchar();
93    getchar();
94    return 0;
95  }
```

再看更复杂的字符串表达式：

((−3+5)*7−9)*2−8−(7−4)*6

此处为简化问题，假定表达式中数字均只有一位，且不含除法，不出现中间结果为小数和0的情况。

形成表达式树的过程是：

(1) 找出表达式中运算优先级低最右边的运算符，以此为表达式的根结点。以此运算符为界，将表达式分成两部分，左边放入左子树中，右边放到右子树中。

(2) 对左右子树按上述方式继续进行，但遇下面的情况分别对待：

若表达式两端为括号，则去括号；

若括号不在表达式两端，则识括号及其内部字符为一个整体；

若表达式为一数字，则为表达式树的叶结点。

如图 4.34 所示。

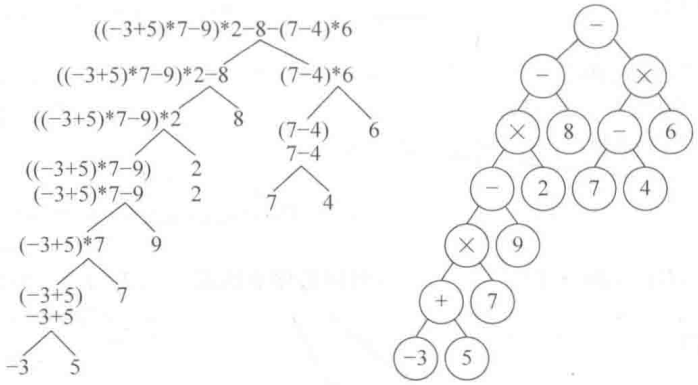


图 4.34

参考程序如下。

```
1  //复杂表达式处理
2  #include<iostream>
3  #include<string>
4  #include<iomanip>                    //横向显示生成的树，头文件
5  #include<windows.h>                  //纵向显示生成的树，头文件
6  #define max 1001
7  using namespace std;
8
9  string tree[max];
10 int qkh(string str[],int i,int *n)    //去括号
11 {
12   string a[max];
13   int j,k=0;
14   int sum=0;
15   if(i==*n&&str[1]=="(")             //判断整个表达式是否被()包住
16   {
17     for(j=1;j<=i;j++)
18       if(str[j]=="("||str[j]==")")   //统计括号的位置与方向
19       {
20         k++;
21         a[k]=str[j];
22       }
23     if((a[2]=="("&&a[k-1]==")")||k==2)
24     //如(1+2)*(3+4)是表达式前后为()，整个表达式却不是在一个括号内，还需分类讨论
25
26     {
27       for(j=1;j<=*n-2;j++)
28         str[j]=str[j+1];
29       str[*n]="\0";str[*n-1]="\0";
30       *n=*n-2;
```

```
31          return *n+1;
32        }
33    }
34    for(j=i;j>=0;j--)                    //如果括号没有覆盖整个字符串则跳过括号内的内容
35    {
36        if(str[j]==")")
37            sum++;
38        if(str[j]=="(")
39            sum--;
40        if(sum==0)
41            break;
42    }
43    return j;
44  }
45
46  void zhh(int n,string str2[],int level,int m)
47  {
48    if(m==1)                           //标记下一个存放的位置,此句为左子树
49        level=2*level;
50    if(m==-1)                          //同上,右子树
51        level=2*level+1;
52
53    int l=0,r=0,i,j;
54    int a=1,k;
55    /*for(i=1;i<=n;i++)
56        cout<<str2[i];cout<<endl;*/
57    //过程中的子串,可取消注释
58    if(n!=1)
59        for(i=n;i>=1;i--)
60        {
61            if((str2[i]=="*"||str2[i]=="/")&&a==1)
62            //a保证此语句运行一次,即括号外的、最右边的*、/号
63                {k=i;a--;}
64            if(str2[i]==")")               //对括号进行处理
65                {i=qkh(str2,i,&n);continue;}
66            else if(str2[i]=="+"||str2[i]=="-")
67            //提取最右边括号外的+、-号进行左右子树的分节
68                {l=i-1;r=n-i;break;}
69        }
70    if(i==0&&a==0&&l==0&&r==0)
71    //若括号外无+、-号,则去*、/进行运算
72        {l=k-1;r=n-k;}
73    if(l>0)                            //左子树
74    {
75        string part1[l+1];             //对左子树赋值
76        for(i=0;i<l+1;i++)
77            part1[i]=str2[i];
78        zhh(l,part1,level,1);
79    }
80    if(r>0)                            //右子树
```

```
81      {
82        string part2[r+1];                    //对右子树赋值
83        for(i=0,j=l+1;i<=r+1,j<=n;i++,j++)
84          part2[i]=str2[j];
85        zhh(r,part2,level,-1);
86      }
87      tree[level]=str2[l+1];
88    }
89
90    void qx(int level)                        //前序遍历
91    {
92      cout<<tree[level];
93      if(tree[2*level]!="\0")
94        qx(2*level);
95      if(tree[2*level+1]!="\0")
96        qx(2*level+1);
97    }
98
99    void hx(int level)                        //后序遍历
100   {
101     if(tree[2*level]!="\0")
102       hx(2*level);
103     if(tree[2*level+1]!="\0")
104       hx(2*level+1);
105     cout<<tree[level];
106   }
107
108   void draw1(int loo,int level)             //画出二叉树
109   {
110     if(tree[level]!="\0")
111     {
112       draw1(loo+8,2*level+1);
113         cout<<setw(loo)<<tree[level]<<endl;
114       draw1(loo+8,2*level);
115     }
116   }
117   void draw2(int level,int x,int y,int k,int space)
118   {
119     if(tree[level]!="\0")
120     {
121       HANDLE hOutput;
122       COORD location;
123       location.X=x;
124       location.Y=y;
125       hOutput=GetStdHandle(STD_OUTPUT_HANDLE);
126       SetConsoleCursorPosition(hOutput,location);
127       if(k==1)
128         cout<<tree[level]<<"/";
129       else if(k==2)
130         cout<<"\\"<<tree[level];
```

```
131         else
132           cout<<tree[level];
133         draw2(2*level,x-space,y+1,1,space/2);
134         draw2(2*level+1,x+space,y+1,2,space/2);
135       }
136   }
137
138   int main()
139   {
140     char a;
141     string str[max];
142     int num=1;
143
144     while(1)
145     {
146       if((a=getchar())=='\n')
147         break;
148       else
149       {
150         str[num]=a;
151         num++;
152       }
153     }
154     num--;
155     zhh(num,str,1,0);
156     cout<<"中序表达式:";
157     for(int i=1;i<=num;i++)
158       cout<<str[i];
159     cout<<endl;
160     cout<<"前序表达式:";
161     qx(1);
162     cout<<endl;
163     cout<<"后序表达式:";
164     hx(1);
165     cout<<endl;
166     //draw1(0,1);                    //横向显示树
167     cout<<endl;
168     draw2(1,40,3,0,20);            //纵向显示树
169     getchar();
170     getchar();
171     return 0;
172   }
```

一般树转二叉树

如果将一般树转成二叉树，应该如何操作呢？主要有四个步骤：

(1) 保留所有结点与其左子结点的链接。

(2) 连结所有兄弟结点。(拥有同一个父结点的子结点)

(3) 打断所有结点原本与右子结点的链接。

(4) 将兄弟结点顺时针转 45°。

例如,将图 4.35 转化成二叉树。

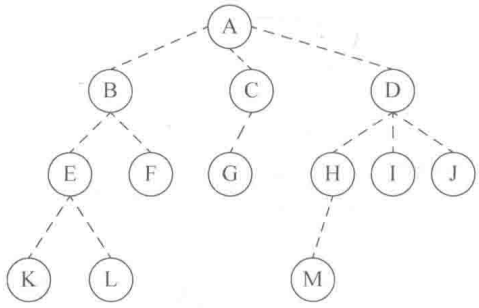


图 4.35

转化方法如图 4.36 所示。

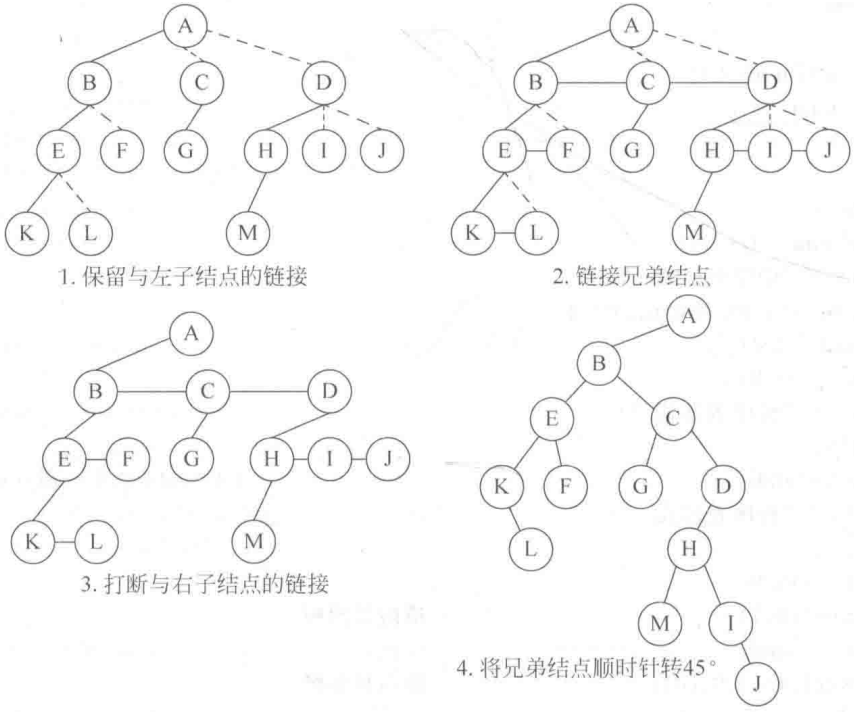


图 4.36

堆排序

直接选择排序(Straight Select Sorting)是一种简单的排序方法,它的基本思想是:对于一个由 R[1]~R[n]组成的无序序列,第一次从 R[1]~R[n]中选取最小值,与 R[1]交换,第二次从 R{2}~R[n]中选取最小值,与 R[2]交换……总共通过 n-1 次交换,完成从小到大排列的有序序列。这种方法,每一次选取最小值时,都需要对剩下的无序序列重新进行一次扫描,但实际上,在上一次扫描过程中,有许多的比较可能已经做过。

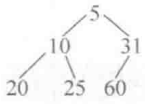
堆排序(Heapsort)是一种树型排序方法,是对直接选择排序的一种改进,可以通过树形结构保存部分比较结果,以减少比较次数。堆排序是不稳定的排序方法,时间复杂度为O(nlogn)。

我们先来看堆的定义:n个元素的序列$\{K_1,K_2,\cdots,K_n\}$,当且仅满足以下两个条件之中的一个时,称为堆。

(1) $k_i \leqslant k_{2i}$且$k_i \leqslant k_{2i+1}$,$i=1,2,\cdots,n/2$

(2) $k_i \geqslant k_{2i}$且$k_i \geqslant k_{2i+1}$,$i=1,2,\cdots,n/2$

堆又分为小根堆和大根堆。满足条件(1)的为小根堆,满足条件(2)的为大根堆。堆可以被看作一种完全二叉树,树中每个结点与数组中存放该结点值的那个元素对应。树的每一层都是填满的,最后一层可能除外(最后一层从一个结点的左子树开始填),因此存储时我们可以用一维数组例如a[]连续存储。图4.37即是小根堆与大根堆的示例。

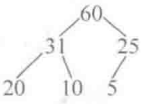


树结构

下标	1	2	3	4	5	6
值	5	10	31	20	25	60

存储结构

(a) 小根堆示例



树结构

下标	1	2	3	4	5	6
值	60	31	25	20	10	5

存储结构

(b) 大根堆示例

图 4.37

在这样的结构下,给定某结点的下标i,a[i]即为该结点数据,其父结点p[i]、左儿子left[i]和右儿子right[i]的下标都可以简单地计算出来:

$$p[i] = i / 2$$
$$left[i] = i \times 2$$
$$right[i] = i \times 2 + 1$$

堆所对应的完全二叉树中,树中任意非叶子结点的关键字均不大于(或不小于)其左右孩子(若存在)结点的关键字。

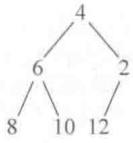
堆顶(根结点)的关键字必为堆里所有结点关键字中最小值(或最大值)。

堆中任一子树亦是堆。

需要说明的是,我们这里讨论的堆实际上是二叉堆(Binary Heap),类似地可定义K叉堆。

堆排序有两个关键步骤,一个是构造堆,一个是调整堆,我们先来讨论如何构造(大根)堆。

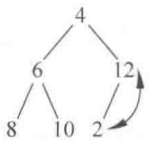
1. 读入无序的n个元素,例如,4,6,2,8,10,12共6个元素。我们将之顺序存入一维数组a[]中,如图4.38所示。



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
空	4	6	2	8	10	12

图 4.38

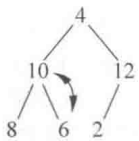
2. 调整堆，直接从 n/2 处即 a[3]开始调整即可。调整规则如下：如当前结点没有孩子，则调整结束，否则，将左右儿子与当前结点进行比较，如当前结点小于左右孩子结点，则将最大的孩子结点与当前结点互换。重复执行该过程，直到当前结点大于左右孩子为止。调整结果为 a[3]与 a[6]互换，互换后 a[6]无孩子，调整结束，如图 4.39 所示。



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
空	4	6	12	8	10	2

图 4.39

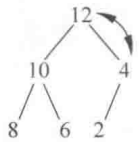
3. 再调整 a[2]。按规则，a[2]与 a[5]互换，互换后 a[5]无孩子，调整结束，如图 4.40 所示。



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
空	4	10	12	8	6	2

图 4.40

4. 调整 a[1]。由于 a[1]<a[2]<a[3]，因此将最大的 a[3]与 a[1]互换。互换后，a[3]还有孩子 a[6]，但 a[6]<a[3]，所以 a[3]的调整结束，如图 4.41 所示。



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
空	12	10	4	8	6	2

图 4.41

以上是构造大根堆的过程，构造小根堆的过程原理是一样的。

堆排序过程(大根堆)是这样的：

1. 将 a[1]与 a[n]互换。
2. 排除元素 a[n]后，调整剩余堆为大根堆。
3. 回到第 1 步，依次反复直至堆内仅有两个元素。

根据样例我们模拟其过程如图 4.42 所示。

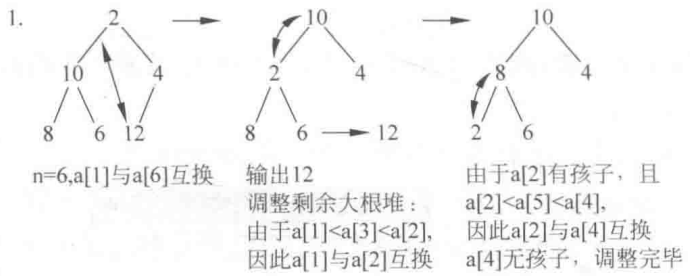


图 4.42

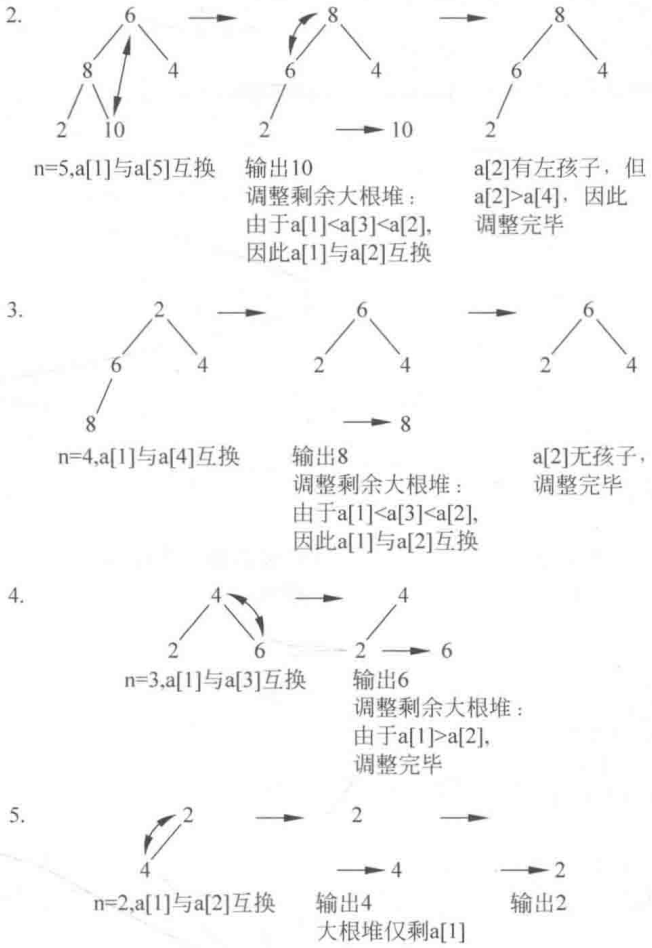


图 4.42(续)

该过程从大到小依次输出各元素，小根堆的排序过程与此类似。

下面的参考程序是小根堆排序，使用数组模拟堆，时间复杂度为 $O(n\log_2 n)$。

```
1   //小根堆排序
2   #include <iostream>
3   using namespace std;
4   #define N 100000
5   int a[N+1];
6
7   void adjust_down(int i,int m)              //调整
8   {
9     int x;
10    while(i*2<=m)
11    {
12      i=i*2;
13      if((i<m)&&(a[i+1]>a[i]))               //检查右兄弟
14        i++;
15      if(a[i]>a[i/2])
16      {
```

```
17          x=a[i/2];
18          a[i/2]=a[i];
19          a[i]=x;
20        }
21        else
22          break;
23    }
24  }
25
26  int main()
27  {
28    int n,i;
29    cin>>n;
30    for(i=1;i<=n;i++)
31      cin>>a[i];
32
33    for(i=n/2;i>=1;i--)                    //只需调整一半即可
34      adjust_down(i,n);                    //建堆
35    for(i=n;i>=2;i--)
36    {
37      a[i]=a[i]^a[1];                      //交换
38      a[1]=a[1]^a[i];
39      a[i]=a[i]^a[1];
40      adjust_down(1,i-1);
41    }
42    for(i=1;i<=n;i++)
43      cout<<a[i]<<' ';
44    return 0;
45  }
```

另一个大根堆的参考程序如下。

```
1   //堆排序
2   #include<iostream>
3   #define MAXN 100
4   using std::cin;
5   using std::cout;
6   int a[MAXN],n,heapsize;
7
8   void maxheapify(int i)                  /*定义根结点*/
9   {
10    int l,r,largest,t;
11    /*计算该结点的左右孩子*/
12    l=i<<1;
13    r=(i<<1)+1;
14    /*记录最大元素的下标*/
15    if(l<=heapsize and a[i]<a[l])
16      largest=l;
17    else
18      largest=i;
```

```
19      if(r<=heapsize and a[r]>a[largest])
20        largest=r;
21      /*交换最大元素与i的值,并进行递归检查交换后的结点是否满足大根堆性质*/
22      if(largest != i)
23      {
24        t=a[i];a[i]=a[largest];a[largest]=t;
25        maxheapify(largest);
26      }
27      return ;
28    }
29
30    void BuildMaxHeap()                    //建堆
31    {
32      int i;
33      heapsize=n;
34      for(i=n/2;i>=1;i--)
35        maxheapify(i);
36      return ;
37    }
38
39    void heapsort()
40    {
41      int i,t;
42      BuildMaxHeap();                      /*建立最大堆*/
43      for(i=n;i>=2;i--)
44      {
45        t=a[1];a[1]=a[i];a[i]=t;           /*将a[i]与a[1]交换*/
46        --heapsize;                        /*剔除交换后的元素a[i]*/
47        maxheapify(1);                     /*维护最大堆性质*/
48      }
49      return ;
50    }
51
52    int main()
53    {
54      int i;
55      cin>>n;
56      for(i=1;i<=n;i++)
57        cin>>a[i];
58      heapsort();
59      for(i=1;i<=n;i++)
60        cout<<a[i]<<" ";
61      system("pause");
62      return 0;
63    }
```

优先队列

虽然堆排序是一个很优美的算法,但在实际中,容易实现的快排往往优于堆排序,所以像二叉堆这种数据结构通常被用来实现优先队列(priority queue)。优先队列是不同于先进

先出的一般队列，它每次从队列中取出的是具有最高优先权的元素。对优先队列的操作一般有查找、插入新元素、删除等。我们举一个例子来说明。

【题目描述】 虚拟现实游戏

魔法世界利用天顶星人的技术开发出了虚拟现实游戏，游戏者只要穿上传感衣，戴上传感头盔，就可以沉浸在虚拟现实的游戏中。由于虚拟现实游戏逼真到使游戏者无法分辨出真实世界与虚拟游戏世界，因此吸引了大批游戏玩家为之狂热，游戏厅里每天玩游戏的玩家都排成了长长的队，黑心的老板为了让游戏机挣到更多的钱，他定下了如下规定：

(1)任何一个玩家玩的时间均为 T 分钟；

(2)排队根据玩家玩一次愿意付出的钱数排序，即玩家愿意付出的费用越高，就越排在队列前面；

(3)排队的玩家随时可以增加费用以获得更高的优先权；

(4)队列中随时可以插入新的玩家，排列顺序按玩家愿付出的费用排序。

请你为游戏厅老板编程实现上述功能。

下面的程序将演示基于堆的优先队列实现的各种功能。

```
1   //优先队列
2   #include<iostream>
3   #include<windows.h>
4   #define MAXH 100000                    /*堆中最大元素数量 + 1*/
5   #define INF 0xffffff
6   using std::cin;
7   using std::cout;
8
9   int n,heapsize;                        /*堆的大小*/
10  int a[MAXH];                           /*堆中元素的值*/
11
12  void maxheapify(int i)                 //维护最大堆
13  {
14    int l,r,largest,t;
15
16    l=i<<1;                              //计算该结点的左右孩子
17    r=(i<<1)+1;
18
19    if(l<=heapsize && a[i]<a[l])         //记录最大元素的下标
20      largest=l;
21    else
22      largest=i;
23    if(r<=heapsize && a[r]>a[largest])
24      largest=r;
25    //交换最大元素与 i 的值,并进行递归检查交换后的结点是否满足最大堆性质
26    if(largest != i)
27      {
28        t=a[i];a[i]=a[largest];a[largest]=t;
29        maxheapify(largest);
30      }
31    return ;
```

```
32  }
33
34  void BuildMaxHeap()                    //建立最大堆
35  {
36     int i;
37     heapsize=n;
38     for(i=n/2;i>=1;i--)
39        maxheapify(i);
40     return ;
41  }
42
43  int ExtractMax(int a[ ])               //弹出并返回堆中最大元素
44  {
45     int max;
46     max=a[1];
47     a[1]=a[heapsize];
48     a[heapsize]=-1;
49     heapsize--;
50     maxheapify(1);                     //维护弹出后堆的性质
51     return max;
52  }
53
54  void HeapIncreaseKey(int i,int key)    //改变结点当前的值
55  {
56     int t;
57     if(a[i]>key)                       //更改的值必须要大于当前的值
58     {
59        cout<<"CURRENT VERX BIGGER THAN KEY\n";
60        return;
61     }
62     a[i]=key;
63     while(i>1 and a[i>>1]<a[i])        /* 旋转并保持堆的性质 */
64     {
65        t=a[i];a[i]=a[i>>1];a[i>>1]=t;i=i>>1;
66     }
67     return ;
68  }
69
70  void MaxHeapInsert(int key)            //往堆中插入一个元素
71  {
72     heapsize++;
73     a[heapsize]=-INF;                  //初始化插入元素
74     HeapIncreaseKey(heapsize,key);     //插入元素
75     return;
76  }
77
78  int main()                             //演示操作
79  {
80     int i,key,v;
81     cin>>n;
```

```
82      for(i=1;i<=n;i++)
83        cin>>a[i];
84      BuildMaxHeap();                    //构建堆
85
86      while(1)
87      {
88        system("cls");
89        cout<<"Please Insert A Instruction:\n";
90        cout<<"1.Show max verx\n";       //查找最大优先值
91        cout<<"2.push max verx\n";       //最大优先值出列
92        cout<<"3.Change verx\n";         //改变某元素的优先值
93        cout<<"4.Insert new Verx\n";     //插入新的元素
94        cout<<"Other.Exit\n";
95        Gout(1,40,6,0,20);
96        cin>>i;
97        if(i==1)
98      cout<<a[1]<<"\n";
99        else if(i==2)
100     cout<<ExtractMax(a)<<"\n";
101       else if(i==3)
102       {
103         cin>>v>>key;
104         HeapIncreaseKey(i,key);
105       }
106       else if(i==4)
107       {
108         cin>>key;
109         MaxHeapInsert(key);
110       }
111       else
112     break;
113       getchar();
114     }
115   }
```

可以看出，插入元素实际上就是先扩充一个结点，将该结点初始化为负无穷，接着再调用函数 HeapIncreaseKey()改变这个结点的值为真正的值。

烽火传递

【题目描述】 烽火传递(balefire.c/cpp/pas)

上古文明遗迹内有一种奇怪的力场，它使得所有的魔法力都无法使用，为了通信方便，墨老师一行不得不采用古代烽火传递的方法。所谓烽火传递，是指在远古时代，防御外敌入侵的方法是使用烽火台报讯。烽火台又称烽燧，一般建在险要处或交通要道上。一旦有敌情发生，白天燃烧柴草，通过浓烟表达信息；夜晚燃烧干柴，以火光传递军情。在某两座城市之间有 n 个烽火台，每个烽火台发出信号都有一定的代价。为了使情报准确地传递，在 m 个烽火台中至少要有一个发出信号。现输入 n，m 和每个烽火台发出的信号的代价，请计算

总共最少需要花费多少代价，才能使敌军来袭之时，情报能在这两座城市之间准确地传递。

例如，有5个烽火台，它们发出信号的代价依次为1、2、5、6、2，且m为3，则总共最少花费的代价为4，即由第2个和第5个烽火台发出信号。

【输入格式】

第一行有两个数n，m分别表示n个烽火台，在m个烽火台中至少要有一个发出信号。

第二行为n个数，表示每一个烽火台的代价。

【输出格式】

一个数，即最小代价。

【输入样例】

5 3 (n，m≤100)

1 2 5 6 2

【输出样例】

4

【算法分析】

本题算法是动态规划＋小根堆排序。

设F[i]表示将信息从1传到i所需花费的最小代价，点燃i之前，必须先点燃[i－M＋1，i－1]区间中的某一个烽火台，因此状态转移方程为F[i]＝MIN(F[k]＋value[i])；(K[i－M＋1，i－1])。

由于小根堆对F[k]中的值进行排序，并将最小的值赋给heap[1]，因此状态转移方程即化简为F[i]＝heap[1]＋value[i]。

```
1   //烽火传递
2   #include <iostream>
3   using namespace std;
4   const int size=100;
5   int n,m,r,value[size],heap[size],pos[size],home[size],opt[size];
6   //heap[i]表示用顺序数组存储的堆heap中第i个元素的值
7   //pos[i]表示opt[i]在堆heap中的位置，即heap[pos[i]]=opt[i]
8   //home[i]表示heap[i]在序列opt中的位置，即opt[home[i]=heap[i]
9
10  void swap(int i,int j)                //交换堆中的第i个和第j个元素
11  {
12    int tmp;
13    pos[home[i]]=j;
14    pos[home[j]]=i;
15    tmp=heap[i];
16    heap[i]=heap[j];
17    heap[j]=tmp;
18    tmp=home[i];
19    home[i]=home[j];
20    home[j]=tmp;
21  }
22
23  void add(int k)                       //在堆中插入opt[k]
```

```
24  {
25    int i;
26    r++;
27    heap[r]=opt[k];
28    pos[k]=r;
29    home[r] = k;
30    i=r;
31    while((i>1) && (heap[i]<heap[i/2]))
32    {
33      swap(i,i/2);
34      i/=2;
35    }
36  }
37
38  void remove(int k)                    //在堆中删除 opt[k]
39  {
40    int i,j;
41    i=pos[k];
42    swap(i,r);
43    r--;
44    if(i==r+1)
45      return;
46    while((i>1) && (heap[i]<heap[i/2]))
47    {
48      swap(i,i/2);
49      i/=2;
50    }
51    while(i+i<=r)
52    {
53      if((i+i+1<=r) && (heap[i+i+1]<heap[i+i]))
54        j=i+i+1;
55      else
56        j = i + i;
57      if(heap[i]>heap[j])
58      {
59        swap(i, j);
60        i=j;
61      }
62      else
63        break;
64    }
65  }
66
67  int main()
68  {
69    int i;
70    freopen("balefire.in","r",stdin);
71    freopen("balefire.out","w",stdout);
72    cin>>n>>m;
73    for(i=1;i<=n;i++)
```

```
74        cin>>value[i];
75    r=0;
76    for(i=1;i<=m;i++)
77    {
78        opt[i]=value[i];
79        add(i);
80    }
81    for(i=m+1;i<=n;i++)
82    {
83        opt[i]= value[i] + heap[1];
84        remove( i - m);
85        add(i);
86    }
87    cout<<heap[1]<<endl;
88    return 0;
89 }
```

此题的递归算法如下。

```
1  //烽火传递递归算法
2  #include<iostream>
3  #include<fstream>
4  using namespace std;
5  int n,m;
6  int *v, *b;
7  int f(int i)
8  {
9      int j,minxx=INT_MAX,sum;
10     if(i+m>n)
11         return 0;
12     else if(b[i])return b[i];
13     else
14         for(j=i+1;j<=i+m;j++)
15         {
16             if(minxx>(sum=f(j)+v[j]))
17                 minxx=sum;
18         }
19     b[i]=minxx;
20     return minxx;
21 }
22 int main()
23 {
24     ifstream fin("balefire.in");
25     ofstream fout("balefire.out");
26     fin>>n>>m;
27     v=new int [n+1];
28     b=new int [n+1];
29     for(int i=1;i<=n;i++)
30         {fin>>v[i];b[i]=0;}
31     v[0]=b[0]=0;
```

```
32        fout<<f(0)<<endl;
33        fin.close();
34        fout.close();
35        return 0;
36    }
```

此题的回溯算法如下。

```
1    //烽火传递回溯算法
2    #include <iostream>
3    #include <fstream>
4    using namespace std;
5    int main()
6    {
7        int i,k,num,n,m,sum,Min=100000;
8        int a[121],d[121];
9        ifstream    fin("balefire.in");
10       ofstream    fout("balefire.out");
11       fin>>n>>m;
12       memset(d,0,sizeof(d));
13       for(i=1;i<=n;i++)
14           fin>>a[i];
15       num=sum=0;
16       k=0;
17       while(k>=0)
18       {
19           d[k]++;
20           if(num>=n-m+1)
21           {
22               if(sum<Min)
23                   Min=sum;
24               sum-=a[num];
25               d[k]=0;
26               k--;
27               num-=d[k];
28           }
29           else
30           {
31               if(sum>=Min)
32               {
33                   d[k]=0;
34                   sum-=a[num];
35                   k--;
36                   num-=d[k];
37               }
38               else if(d[k]<=m)
39               {
40                   num+=d[k];
41                   sum+=a[num];
42                   k++;
```

```
43                  }
44                  else
45                  {
46                      sum-=a[num];
47                      d[k]=0;
48                      k--;
49                      num-=d[k];
50                  }
51              }
52          }
53          fout<<Min<<endl;
54          fin.close();
55          fout.close();
56          return 0;
57      }
```

普通排序法程序如下。

```
1   #include<iostream>
2   #include<fstream>
3   using namespace std;
4   int *v, *line, *pos, *head;
5   void sort(int i,int j)
6   {
7       int n=i,m=j,k=head[(i+j)/2],t;
8       while(n<=m)
9           {
10              while(n<j && head[n]>k)n++;
11              while(m>i && head[m]<k)m--;
12              if(n<=m)
13              {
14                  t=head[n];
15                  head[n]=head[m];
16                  head[m]=t;
17                  t=pos[line[n]];
18                  pos[line[n]]=pos[line[m]];
19                  pos[line[m]]=t;
20                  t=line[m];
21                  line[m]=line[n];
22                  line[n]=t;
23                  n++;
24                  m--;
25              }
26          }
27      if(n<j)
28          sort(n,j);
29      if(m>i)
30          sort(i,m);
31  }
32  int main()
```

```
33  {
34      ifstream fin("balefire.in");
35      ofstream fout("balefire.out");
36      int n,m,i;
37      fin>>n>>m;
38      v=new int[n+2];
39      line=new int [m+2];
40      pos=new int [n+2];
41      head=new int [m+2];
42      for(i=1;i<=n;i++)
43          {fin>>v[i];pos[i]=i;}
44      for(i=1;i<=m;i++)
45          {head[i]=v[i];pos[i]=i;line[i]=i;}
46      head[m+1]=0;
47       sort(1,m);
48      for(i=m+1;i<=n;i++)
49          {
50              v[i]+=head[m];
51              pos[i]=m+1;
52              head[m+1]=v[i];
53              head[pos[i-m]]=0;
54              line[m+1]=i;
55              sort(1,m+1);
56          }
57      fout<<head[m]<<endl;
58      fin.close();
59      fout.close();
60      return 0;
61  }
```

第五章

图

图的基本概念

图是一种非线性结构，它比树形结构更复杂。在图中，数据元素之间是多对多的关系，因此图用于表达数据元素之间存在着的复杂关系，我们称这种关系为网状结构关系。

著名的“欧拉七桥问题”就是最早的图问题。古时在Koenigsberg(哥尼斯堡)这个地方，有一条河，将两岸和小岛分隔出4个区域(A,B,C,D)，连接这4个区域的是7座桥(a,b,c,d,e,f,g)，如图5.1所示。而七桥问题是“是否可以从某一个区域出发，经过每座桥一次，而回到原先出发的区域?”

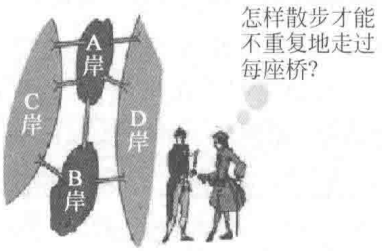


图 5.1

图5.1实际上就可以转化为图的问题，而且欧拉发现：如果任一个区域上所连接的桥数为偶数时，才有可能从任一座桥出发，经过每一座桥。因此，我们可以发现，图5.1实际上是没有解的。

无向图形：在图形中任一顶点上的边都是没有方向性的。例如，图5.2就是一个无向图形。因为任意两个顶点间的边都没有方向性，如顶点1和顶点2之间的边表示可以从顶点1到顶点2，也可以从顶点2到顶点1。

有向图形：在图形中任一顶点上的边都是有方向性的。例如图5.3就是一个有向图形，因为任意两个顶点间的边都是有方向的，如顶点1和顶点2之间的边表示可从顶点1到顶点2，但是不可以从顶点2到顶点1。

在有向图形和无向图形之中，我们还必须加上一些限制。

1. 图形中不允许自身循环，如图5.4中的顶点3。

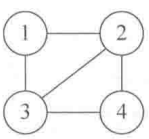


图 5.2

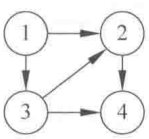


图 5.3

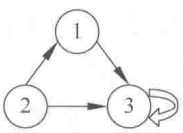


图 5.4

2. 图形中两顶点的边，不可以重复。边重复的图形，我们称为多边形图，如图 5.5 中顶点 2 到顶点 3，有 3 条重复的边。

完全图形：在无向图形中任意两顶点上都存在一个边，例如图 5.6 就是一个完全图形，因为任意两个顶点间都存在一个边。

子图形：由图形 G 中取出的部分集合，如图 5.7 为图形 G。则图 5.8 中的所有图皆为其子图形。

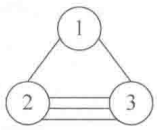


图 5.5

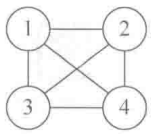


图 5.6

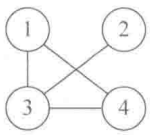


图 5.7

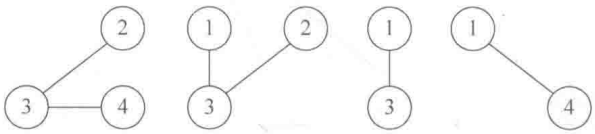


图 5.8

路径：在图形中从顶点 A 到达顶点 B 所经过的所有的边。例如图 5.9 从顶点 1 到顶点 5 的路径为<V1,V2>、<V2,V4>、<V4,V5>，而路径长度为经过的边数，在这个例子中，路径的长度为 3。

简单路径：在图形中，除了起点和终点可以重复(不重复亦可)外，其余的顶点皆不相同的路径，如图 5.9 中的<V1,V2>、<V2,V4>、<V4,V5>为简单路径。

回路：在图形中，起点和终点相同的简单路径，如图 5.10 中，<V1,V2>、<V2,V4>、<V4,V1>就是一条回路。除第一个和最后一个顶点外，其余各顶点均不重复出现的回路称为简单回路。

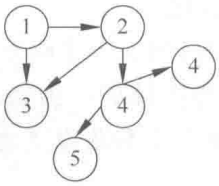


图 5.9

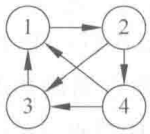


图 5.10

连通顶点：在无向图形中，顶点 A 到顶点 B 间存在一条路径，则称顶点 A 和顶点 B 为连通顶点。

连通图形：如果在无向图形中，任意两个顶点皆连通，则称为连通图形。即任意两个顶点皆存在一条路径可到达。

连通单元：所谓连通单元是将无向图形分为多个分离的子图形之后，原图形的连通顶点仍在同一个子图中。

强连通顶点：在有向图形中，顶点 A 到顶点 B 间存在一条路径，而顶点 B 到顶点 A 也存在一条路径，则称顶点 A 和顶点 B 为强连通顶点。

强连通图形：如果在有向图形中，任意两个顶点间皆存在一条路径可到对方，则称为强连通图形。

强连通单元：将有向图形分为多个分离的子图形之后，原图形的强连通顶点仍在同一个子图中。

度：有向图形中，因为边皆是有方向性的，我们还定义内分支度和外分支度。内分支度是指由其他顶点前往此顶点的边数，而外分支度是指由此顶点前往其他顶点的边数。如图 5.11 所示，顶点 1 的内分支度为 2，外分支度为 1。在有向图形中，各顶点的内分支度或外分支度总和，即为此图形的边数。

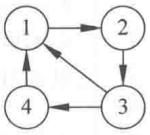


图　5.11

邻接数组表示法

邻接数组表示法是以一个 n×n 的数组来表示一个具有 n 个顶点的图形。我们以数组的索引值来表示顶点，以数组的内容值来表示顶点间的边是否存在（以 1 表示存在边；以 0 表示不存在边）。如图 5.12 所示的无向图形，其邻接数组为表 5.1 所示。

表　5.1

	0	1	2	3	4
0	0	1	1	1	0
1	1	0	0	1	1
2	1	0	0	1	0
3	1	1	1	0	0
4	0	1	0	0	0

可以看出，在无向图形中，邻接数组中的内容值呈现出一种对称的关系，因为如果顶点 A 和顶点 B 存在公共边，即表示顶点 A 可到达顶点 B，顶点 B 也可到达顶点 A。

再来看看有向图形的邻接数组如何表示，如图 5.13 所示。

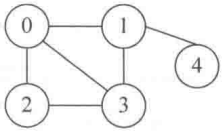


图　5.12

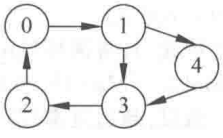


图　5.13

其邻接数组为表 5.2 所示。

表　5.2

	0	1	2	3	4
0	0	1	0	0	0
1	0	0	0	1	1
2	1	0	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0

现在我们设计一个将上述图形转成邻接数组的程序，参考程序如下。

```
1  //邻接数组
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define Max 6
5  int graph[Max][Max]={0};
6
7  void printGraph()
8  {
9     int i,j;
10    for(i=0;i<Max;i++)
11    {
12       for(j=0;j<Max;j++)
13          printf("%d  ",graph[i][j]);
14       printf("\n");
15    }
16 }
17
18 int main()
19 {
20    int i,j;
21    int node1;
22    int node2;
23    while(1)
24    {
25       printf("------------------------\n");
26       printf("请输入起始点:  ");
27       scanf("%d",&node1);
28       if(node1==-1)
29          break;
30       printf("请输入结束点:  ");
31       scanf("%d",&node2);
32
33       if(node1==node2)
34          printf("错误,自身循环!\n");
35       else if(node1>=Max || node2>=Max)
36          printf("错误,超出范围!");
37       else
38          graph[node1][node2]=1;
39    }
40    printf("\邻接数组为:\n");
41    printGraph();
42    system("pause");
43    return 0;
44 }
```

采用邻接矩阵表示图，直观方便，很容易查找图中任两个顶点 i 和 j 之间有无边或边上的权值，因为可以根据 i,j 的值随机查找存取，所以时间复杂度为 O(1)。也很容易计算一个顶点的度和邻接点，其时间复杂度为 O(n)。但是空间复杂度为 O(n×n)，如果用来表示稀

疏图，则会造成巨大的空间浪费。

邻接列表表示法

邻接列表法是以链表来记录各顶点的邻接顶点。其结点结构如下：

邻接顶点	下一个邻接顶点

如图 5.14 所示的有向图形：

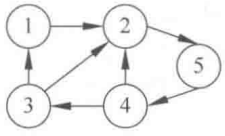
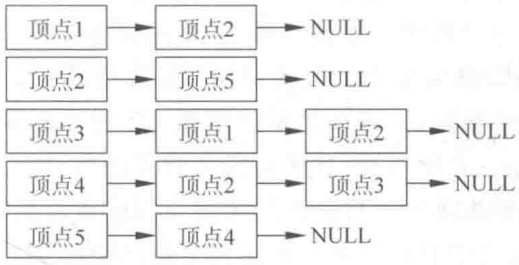


图 5.14

其邻接列表为：



在 C++程序中，邻接列表的结构声明如下：

```
struct Node
{
  int Vertex;
  struct Node* next;
};
```

参考程序如下所示。

```
//邻接列表法
#include <iostream>
using namespace std;
#define Num 6

struct Node
{
   int vertex;
   struct Node* next;
};
typedef struct Node* Graph;
Node Head[Num];

```

```
14  void CreateG(int node1,int node2)
15  {
16     Graph pointer;
17     Graph New;
18
19     New=(Graph)malloc(sizeof(struct Node));          //开辟新结点
20     if(New!=NULL)                                    //如果结点开辟成功
21     {
22        New->vertex=node2;
23        New->next=NULL;
24        pointer=&(Head[node1]);
25        while(pointer->next!=NULL)
26           pointer=pointer->next;
27        pointer->next=New;                            //将新结点链接
28     }
29  }
30
31  void printGraph()                                   //打印邻接列表
32  {
33     Graph pointer;
34     int i,j;
35     for(i=0;i<Num;i++)
36     {
37        pointer=Head[i].next;
38        printf("Head[%d]",i);
39        while(pointer!=NULL)
40        {
41           printf("-> %d ",pointer->vertex);
42           pointer=pointer->next;
43        }
44        printf("\n");
45     }
46  }
47
48  int main()
49  {
50     int i,j;
51     int node1;
52     int node2;
53     for(i=0;i<Num;i++)                               //赋初值
54     {
55        Head[i].vertex=i;
56        Head[i].next=NULL;
57     }
58     while(1)
59     {
60        printf("------------------------\n");
61        printf("请输入起始点:  ");
62        scanf("%d",&node1);
63        if(node1==-1)
```

```
64            break;
65        printf("请输入结束点:  ");
66        scanf("%d",&node2);
67
68        if(node1==node2)
69            printf("错误,自身循环!\n");
70        else if(node1>=Num || node2>=Num)
71            printf("错误,超出范围!");
72        else
73            CreateG(node1,node2);
74    }
75    printf("\n邻接列表为:\n");
76    printGraph();
77    system("pause");
78    return 0;
79 }
```

邻接列表表示法便于查找任一顶点的关联边及邻接边，只要从表头向量中取出对应表头指针，然后进行查找即可。由于无向图的每个顶点的单链表平均长度为 2e/n，所以运算的时间复杂度为 O(e/n)。对于有向图来说，想要查找一个顶点的后继顶点和以该顶点为起点的边，包括求该顶点的出度都很容易；但要查找一个顶点的前驱顶点和以此顶点为终点的边，以及该顶点的入度就不方便了。需要扫描整个表，时间复杂度为 O(n+e)。所以对于查找顶点入度或以该顶点为终点的关联边的运算时，可以建立一个逆邻接表，该表中每个顶点的单链表存储的是所有以该点为终点的关联边信息。甚至还可以把邻接表和逆邻接表结合起来，构造成“十字邻接表”，此时，每个边结点的数据信息包含五个域：起点、终点、权，以及以该顶点为终点的关联边的链接，以该顶点为起点的关联边的链接。表头向量的结点也包括三个域：顶点编号，以该点为终点的表头指针域，以该点为起点的表头指针域。

加权边的图形

在七桥问题中，每个桥都是有长度的，为了表示这些信息，我们在图形的边上加上一些数字来表示，这样的图，我们称为“加权边的图形”。例如图 5.15 就是一个加权边的图形。

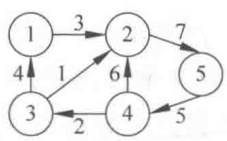


图 5.15

它的邻接数组应该如表 5.3 这样表示。

表 5.3

	1	2	3	4	5
1	0	3	0	0	0
2	0	0	0	0	7
3	4	1	0	0	0
4	0	6	2	0	0
5	0	0	0	5	0

如果使用邻接列表来表示图形，则列表的结点必须添加一个记录加权值的字段。加权边的邻接列表结点结构如下：

加权值	邻接顶点	下一个邻接顶点

深度优先法

深度优先法(DFS)：想象一只老鼠在一座不见天日的迷宫内，老鼠在入口处进去，要从出口出来。那老鼠会怎么走？当然是这样的：老鼠如果遇到直路，就一直往前走，如果遇到分叉路口，就任意选择其中的一个继续往下走，如果遇到死胡同，就退回到最近的一个分叉路口，选择另一条道路再走下去，如果遇到了出口，老鼠的旅途就算结束了。深度优先搜索法的基本原则就是这样：按照某种条件往前试探搜索，如果前进中遭到失败(正如老鼠遇到死胡同)，则退回头另选通路继续搜索(用堆栈实现)，直到找到条件的目标为止。

例如对于图 5.16 所示的无向图形：

使用深度优先法的过程如下。

① 如果从顶点 1 开始深度优先搜索，顶点 1 存入堆栈，如图 5.17 所示。

② 顶点 1 的邻接顶点为顶点 2 和顶点 3，我们任意选择一个顶点例如顶点 2 继续往下深度优先搜索，将顶点 2 存入堆栈，如图 5.18 所示。

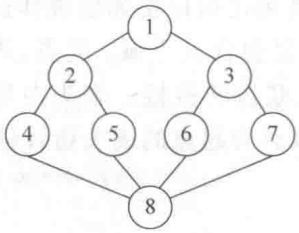


图 5.16

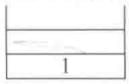


图 5.17

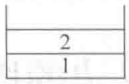


图 5.18

③ 由于顶点 2 的邻接顶点为顶点 4 和顶点 5，选择顶点 4 往下继续深度优先搜索。将顶点 4 存入堆栈，如图 5.19 所示。

④ 顶点 4 的邻接顶点为顶点 8，将顶点 8 存入堆栈，如图 5.20 所示。

⑤ 由于顶点 8 的邻接顶点为顶点 4、顶点 5、顶点 6 和顶点 7，顶点 4 已经查找过，因此选择顶点 5 继续深度优先搜索。将顶点 5 存入堆栈，如图 5.21 所示。

⑥ 发现顶点 5 的邻接顶点为顶点 2，顶点 2 已经查找过，返回到顶点 8。将顶点 5 从堆栈取出，如图 5.22 所示。

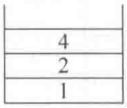


图 5.19

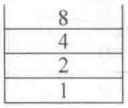


图 5.20

图 5.21

图 5.22

⑦ 顶点 8 的邻接顶点为顶点 4 和顶点 5、顶点 6 和顶点 7，顶点 4 和顶点 5 已经查找过，选择顶点 6 继续深度优先搜索。将顶点 6 存入堆栈，如图 5.23 所示。

⑧ 发现顶点 6 的邻接顶点为顶点 3，选择顶点 3 继续深度优先搜索。将顶点 3 存入堆栈，如图 5.24 所示。

⑨ 发现顶点 3 的邻接顶点为顶点 7，选择顶点 7 继续深度优先搜索。将顶点 7 存入堆栈，如图 5.25 所示。

⑩ 顶点 7 的邻接顶点皆查找完，取出堆栈中的顶点。堆栈中所有顶点的邻接皆已查找，此时堆栈为空。结束查找，如图 5.26 所示。

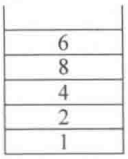


图 5.23

图 5.24

图 5.25

图 5.26

所以查找顺序为：1,2,4,8,5,6,3,7。但因为在深度优先搜索时，可选择同一深度的邻接顶点中的任何一个继续进行深度查找，所以深度优先搜索的顺序并不是唯一的。

参考程序如下。

```
1   //深度优先搜索
2   #include <iostream>
3   using namespace std;
4   #define Num 9
5
6   struct Node
7   {
8     int vertex;
9     struct Node* next;
10  };
11
12  typedef struct Node* Graph;
13  Node Head[Num];                    //9 个顶点
14  int visited[Num]={0};              //标记是否访问过
15
16  void CreateG(int node1,int node2)  //建立邻接列表
17  {
18    Graph pointer;
19    Graph New;
20    New=(Graph)malloc(sizeof(struct Node));
21    if(New!=NULL)
22    {
23      New->vertex=node2;
24      New->next=NULL;
25      pointer=&(Head[node1]);
26      while(pointer->next!=NULL)
27        pointer=pointer->next;
```

```
28          pointer->next=New;
29      }
30  }
31
32  void printG()                                    //打印图
33  {
34      Graph pointer;
35      int i,j;
36      for(i=0;i<Num;i++)
37      {
38          pointer=Head[i].next;
39          printf("Head[%d]",i);
40          while(pointer!=NULL)
41          {
42              printf("-> %d ",pointer->vertex);
43              pointer=pointer->next;
44          }
45          cout<<"\n";
46      }
47  }
48
49  void DFS(int vertex)                             //深度优先搜索
50  {
51      Graph pointer;
52      visited[vertex]=1;
53      printf("[%d]->",vertex);
54      pointer=Head[vertex].next;
55      while(pointer!=NULL)
56      {
57          if(visited[pointer->vertex]==0)
58              DFS(pointer->vertex);
59          pointer=pointer->next;
60      }
61  }
62
63  int main()
64  {
65      int i;
66      int node[20][2]={{1,2},{2,1},{1,3},{3,1},{2,4},
67                       {4,2},{2,5},{5,2},{3,6},{6,3},
68                       {3,7},{7,3},{4,8},{8,4},{5,8},
69                       {8,5},{6,8},{8,6},{7,8},{8,7}};
70      for(i=0;i<Num;i++)
71      {
72          Head[i].vertex=i;
73          Head[i].next=NULL;
74      }
75      for(i=0;i<20;i++)
76          CreateG(node[i][0],node[i][1]);
77      printG();
```

```
    DFS(1);
    cout<<"\n\n";
    system("pause");
    return 0;
}
```

运行结果如图 5.27 所示。

```
Head[0]
Head[1]-> 2 -> 3
Head[2]-> 1 -> 4 -> 5
Head[3]-> 1 -> 6 -> 7
Head[4]-> 2 -> 8
Head[5]-> 2 -> 8
Head[6]-> 3 -> 8
Head[7]-> 3 -> 8
Head[8]-> 4 -> 5 -> 6 -> 7
[1]->[2]->[4]->[8]->[5]->[6]->[3]->[7]->
```

图 5.27

广度优先法

广度优先法(BFS)是指在图形中,如果以顶点 V 作为起始点开始查找,我们从顶点 V 的邻接列表中选择一个未查找过的顶点 W,将顶点 V 的所有邻接顶点查找过后,再继续对顶点 W 的所有邻接顶点进行广度优先法的查找,然后再继续查找顶点 V 的下一个邻接顶点的所有邻接顶点,重复进行广度优先搜索,直到所有的邻接顶点皆查找过为止。通常是使用队列来存储邻接顶点,每查找一个邻接顶点便把其所有的邻接顶点存入队列中,直到队列空了才结束广度优先搜索。

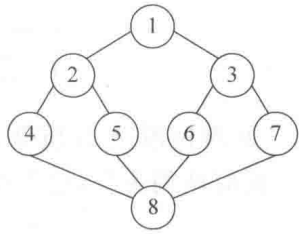


图 5.28

例如图 5.28 所示的无向图形:

① 如果从顶点 4 开始广度优先搜索,将顶点 4 存入队列中。

4					

② 查找顶点 4,将顶点 4 的邻接顶点 2 和顶点 8 存入队列中,将顶点 4 从队列中取出。

2	8				

③ 查找顶点 2,将顶点 2 的邻接顶点存入队列中,邻接顶点 4 已查找过,不必存入队列中,将顶点 2 从队列中取出。

8	1	5			

④ 查找顶点 8,将顶点 8 的邻接顶点存入队列中,邻接顶点 4 已查找过和邻接顶点 5 已在队列中,所以不必存入队列,将顶点 8 从队列中取出。

1	5	6	7		

⑤ 查找顶点 1，将顶点 1 的邻接顶点存入队列中，邻接顶点 2 已查找过，不必存在队列中。将顶点 1 从队列中取出。

5	6	7	3		

⑥ 查找顶点 5，将顶点 5 的邻接顶点存入队列中，邻接顶点 2 和邻接顶点 8 已查找过，不必存入队列中，将顶点 5 从队列中取出。

6	7	3			

⑦ 查找顶点 6，将顶点 6 的邻接顶点存入队列中，邻接顶点 8 已查找过和邻接顶点 3 已在队列中，所以不必存入队列。将顶点 6 从队列中取出。

7	3				

⑧ 查找顶点 7，将顶点 7 的邻接顶点存入队列中，邻接顶点 8 已查找过和邻接顶点 3 已在队列中，所以不必存入队列。将顶点 7 从队列中取出。

3					

⑨ 查找顶点 3，顶点 3 的邻接顶点皆已查找过，不必存入队列。将顶点 3 从队列中取出。此时队列为空，结束查找。

所以查找的顺序为：4，2，8，1，5，6，7，3。而且由于广度优先搜索时，同一广度的邻接顶点，可选择其中一个继续进行邻接顶点的广度查找，因此广度优先搜索的顺序也不是唯一的。

参考程序如下。

```
1   //广度优先法
2   #include <iostream>
3   using namespace std;
4   #define Num 9
5   #define QueueNum 10
6
7   struct Node
8   {
9     int vertex;
10    struct Node* next;
11  };
12  typedef struct Node* Graph;
13  Node Head[Num];                    //9 个顶点
```

```
14  int visited[Num]={0};                    //标记是否访问过
15  int Queue[QueueNum];
16  int front=-1;
17  int rear=-1;
18
19  int Enqueue(int vertex)                  //入队列
20  {
21    if(rear>=QueueNum)
22      return -1;
23    else
24    {
25      rear++;
26      Queue[rear]=vertex;
27      return 1;
28    }
29  }
30
31  int Dequeue()                            //出队列
32  {
33    if(front==rear)
34      return -1;
35    else
36    {
37      front++;
38      return Queue[front];
39    }
40  }
41
42  void BFS(int vertex)                     //宽度优先搜索
43  {
44    Graph pointer;
45    Enqueue(vertex);
46    visited[vertex]=1;
47    printf("%d->",vertex);
48    while(front!=rear)
49    {
50      vertex=Dequeue();                    //出队列
51      pointer=Head[vertex].next;           //指向下一个结点
52      while(pointer!=NULL)
53      {
54        if(visited[pointer->vertex]==0)    //如结点未被访问,即做标记
55        {
56          Enqueue(pointer->vertex);
57          visited[pointer->vertex]=1;
58          printf("%d->",pointer->vertex);
59        }
60        pointer=pointer->next;
61      }
62    }
63  }
```

```
64
65  void CreateG(int node1,int node2)                //建立邻接列表
66  {
67     Graph pointer;
68     Graph New;
69     New=(Graph)malloc(sizeof(struct Node));
70     if(New!=NULL)
71     {
72        New->vertex=node2;
73        New->next=NULL;
74        pointer=&(Head[node1]);
75        while(pointer->next!=NULL)
76           pointer=pointer->next;
77        pointer->next=New;
78     }
79  }
80
81  void printG()                                    //打印图
82  {
83     Graph pointer;
84     int i,j;
85     for(i=0;i<Num;i++)
86     {
87        pointer=Head[i].next;
88        printf("Head[%d]",i);
89        while(pointer!=NULL)
90        {
91           printf("-> %d ",pointer->vertex);
92           pointer=pointer->next;
93        }
94        printf("\n");
95     }
96  }
97
98  void DFS(int vertex)                             //深度优先搜索
99  {
100    Graph pointer;
101    visited[vertex]=1;
102    printf("[%d]->",vertex);
103    pointer=Head[vertex].next;
104    while(pointer!=NULL)
105    {
106       if(visited[pointer->vertex]==0)
107          DFS(pointer->vertex);
108       pointer=pointer->next;
109    }
110 }
111
112 int main()
113 {
```

```
114     int i;
115     int node[20][2]={{1,2},{2,1},{1,3},{3,1},{2,4},
116                      {4,2},{2,5},{5,2},{3,6},{6,3},
117                      {3,7},{7,3},{4,8},{8,4},{5,8},
118                      {8,5},{6,8},{8,6},{7,8},{8,7}};
119     for(i=0;i<Num;i++)
120     {
121       Head[i].vertex=i;
122       Head[i].next=NULL;
123     }
124     for(i=0;i<20;i++)
125       CreateG(node[i][0],node[i][1]);
126     printG();
127     BFS(1);
128     printf("\n\n\n");
129     system("pause");
130   }
```

运行结果如图 5.29 所示。

```
Head[0]
Head[1]-> 2 -> 3
Head[2]-> 1 -> 4 -> 5
Head[3]-> 1 -> 6 -> 7
Head[4]-> 2 -> 8
Head[5]-> 2 -> 8
Head[6]-> 3 -> 8
Head[7]-> 3 -> 8
Head[8]-> 4 -> 5 -> 6 -> 7
1 ->2->3->4->5->6->7->8->
```

图 5.29

生成树问题

一个图形，如果有 N 个顶点，则至少要有 N－1 个边才能将 N 个顶点给相连起来，形成连通图形。这种用 N－1 个边的连通图形，我们称为生成树。

例如图 5.30 所示的无向图形，我们可以以深度优先搜索法或广度优先搜索法生成树。以顶点 1 开始的深度优先搜索，所生成的树如图 5.31 所示。

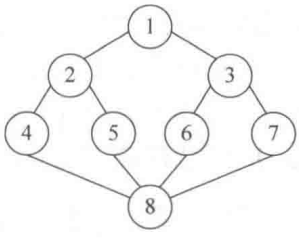


图 5.30

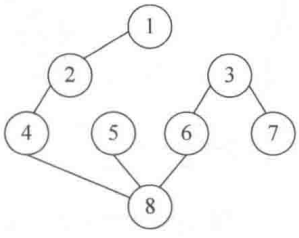


图 5.31

以顶点 4 开始的广度优先搜索，所生成的树如图 5.32 所示。

最小生成树(MST)：在一个有加权边的图形生成的所有生成树中，加权值总和最小的生成树，我们称为最小生成树。例如：魔法学院有 5 栋大楼，如图 5.33 所示，我们需要用网络来连接各栋大楼，已知每栋楼之间的距离，问：怎样连线距离最短？

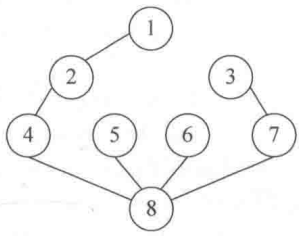


图 5.32

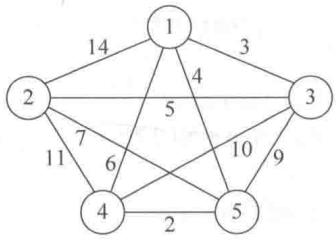


图 5.33

如果我们能够找出这张图形的最小生成树，就可以用最少的网络线连接 5 栋大楼。

Kruskal 算法

Kruskal 算法是根据边的加权值以递增的方式，依次找出加权值最低的边来建最小生成树，并且每次添加的边不能造成生成树有回路，直到找到 N－1 个边为止。当边集比较少时，可考虑用此法。我们以图 5.34 为例。

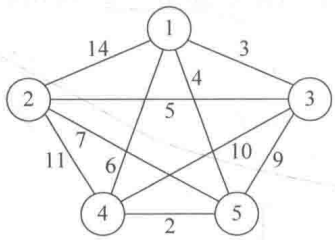


图 5.34

首先将图形中所有的边递增排序(快排)，结果如表 5.4 所示。

表 5.4

邻接边	4,5	1,3	1,5	2,3	1,4	2,5	3,5	3,4	2,4	1,2
加权值	2	3	4	5	6	7	9	10	11	14

第一步，将(4,5)的边加到生成树中，如图 5.35 所示。

第二步，将(1,3)加入生成树中，如图 5.36 所示。

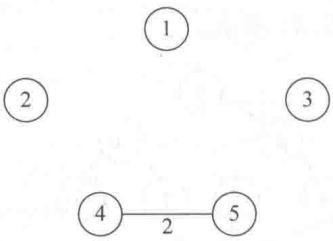


图 5.35

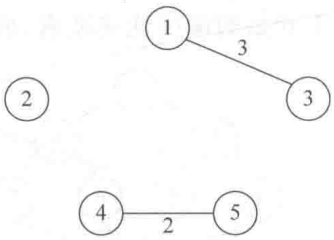


图 5.36

第三步，将(1,5)加入生成树中，如图 5.37 所示。

第四步，将(2,3)加入生成树中，如图 5.38 所示。

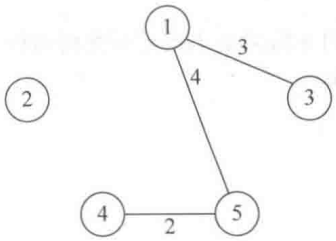


图 5.37

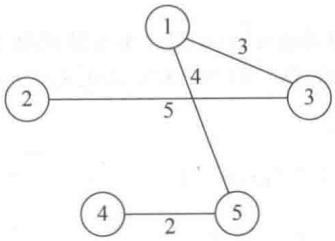


图 5.38

由于共有 5 个顶点，而现在生成树已达 4 个边，生成树建立完成。加权值总和为 14。

其实这个例子举得并不是十分恰当，容易造成只要排序后加 N－1 条边就可以的错觉。然而在很多条件下情况并非如此，比如说图 5.39 所示的这个例子。

这里就需要并查集来判断两个结点是否属于同一棵树从而判断是否应该将这条边加入。

即每加一条边时，你需要考虑：

1. 它是不是剩下边中最小的一条；
2. 加入它会不会造成图的回路。

以图 5.40 为例。

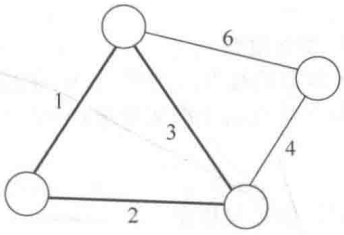


图 5.39

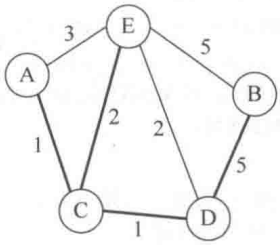


图 5.40

先让每个结点的父结点 father[]＝自身编号；

第一次加入(A,C)边，C 的父结点设为 A；

第二次要加入(C,D)边，D 的父结点设置为 C，而 C 的父结点是 A，则 D 的父结点也为 A；

第三次加入(C,E)边，同理，E 的父结点为 A；

第四次要加入(D,E)边，因为 D,E 的父结点都为 A，表示 E,D 在一个分支中，会形成回路，拒绝加入；

第五次要加入(A,E)边，因为 A,E 的父结点都为 A，拒绝加入；

第六次加入(B,D)边，D 的父结点为 A。

判断是否为相同集合的代码片断如下。

```
int dad[1000];                          //每个结点属于的父结点集合
int x[1000],y[1000],w[1000];            //边结点 x,边结点 y,权值 w

int getfather(int x)
```

```
5   {
6     //如 dad[x]=x,则 x 本身即树根,返回 x;否则返回 x 的父亲 dad[x]所在树的根结点
7     return dad[x]==x?x:dad[x]=getfather(dad[x]);
8   }
9
10  for(i=1;i<e;i++)
11  {
12    x=getfather(x[i]);
13    y=getfather(y[i]);
14    if(x!=y)                          //如果不在同一集合
15    {
16      ans += w[i];
17      dad[x]=y;
18      count++;
19    }
20  }
```

参考程序如下所示。

```
1   //Kruskal 算法
2   #include<iostream>
3   #define MAXN 2000
4   #define INF 99999999
5   using namespace std;
6   int n,e;                          /*点数和边数*/
7   int x[MAXN],y[MAXN],w[MAXN];      /*x 为边的起点,y 为终点,w 为边的权值*/
8   int dad[MAXN];                    /*每个结点属于的父亲集合*/
9
10
11  void sort(int i,int j)            //快排,给边权排序
12  {
13    if(i>=j)
14      return ;
15    int m,n,t,k;
16    m=i,n=j;
17    k=w[ (i+j) >> 1];               //取中值
18    while(m<=n)
19    {
20      while(w[m]<k)
21        m++;
22      while(w[n]>k)
23        n--;
24      if(m<=n)
25        {
26        t=x[m];
27        x[m]=x[n];
28        x[n]=t;
29        t=y[m];
30        y[m]=y[n];
31        y[n]=t;
```

```
32          t=w[m];
33          w[m]=w[n];
34          w[n]=t;
35          m++;
36          n--;
37        }
38    }
39    sort(i,n);
40    sort(m,j);
41  }
42
43  int getfather(int x)                    /*查找点 x 属于的集合*/
44  {
45    if(x==dad[x])
46      return x;
47    dad[x]=getfather(dad[x]);  //查找过程中更新 x 的父亲集合,相当于并查集归并过程
48    return (dad[x]);
49  }
50
51  void kruskal()
52  {
53    int i,p,ans;                          /*p:已经加入的边数,ans:加入边的边权和*/
54    for(i=1;i<=n;i++)
55      dad[i]=i;                           /*初始化点的集合*/
56    p=1,ans=0;          //p 为顶点数,为加快计算速度 p,初始化为 1,所以当 p=n 时结束计算
57
58    for(i=1;i<=e;i++)
59    {
60      if(getfather(x[i])!=getfather(y[i]))  //如果边的两点不在同一个集合内,则归并两个集合
61
62        {
63          ans+=w[i];                      /*加入这条边并统计其权值*/
64          dad[getfather(x[i])]=y[i];      //合并为同一父亲集合,此处是取 y[i]的值
65          p++;
66          if(p == n)                      //如果最小生成树中的顶点数等于全部顶点数 - 1
67          {
68            cout<<ans<<"\n";
69            return ;
70          }
71        }
72      }
73      return ;
74  }
75
76  int main()
77  {
78    int i,j;
79    cin>>n>>e;                            //输入结点数和边数
80    for(i=1;i<=e;i++)
81      cin>>x[i]>>y[i]>>w[i];
82    sort(1,e);                            /*将边按权值大小排序*/
```

```
    kruskal();
    system("pause");
    return 0;
}
```

请尝试使用堆排序完成上述代码。

Prims 算法

Prims 算法是一种基于“贪心”的求最小树算法，以每次加入一个的邻接边来建立最小生成树，直到找到 N－1 个边为止。Prims 的规则是以开始时生成树的集合为起始的顶点，然后找出与生成树集合邻接的边中，加权值最小的边来生成树，为了确定新加入的边不会造成回路，所以每一个新加入的边，只允许有一个顶点在生成树集合中。重复执行此步骤，直到找到 N－1 个边为止。

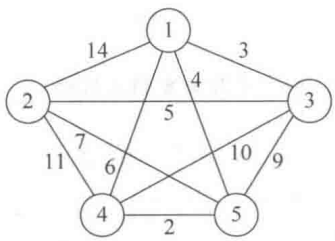


图 5.41

Prims 算法适合稠密图，其时间复杂度为 $O(n^2)$，其时间复杂度与边的数目无关，而 Kruskal 算法的时间复杂度为 O(eloge)，跟边的数目有关，适合稀疏图。我们以图 5.41 为例：

设数组 mincount[]，其中 mincount[i]表示集合 U 中的顶点到集合 U 外的顶点 i 的最小边权。设 mincount[1]＝0，则因为 mincount[1]＝0 最小，将顶点 1 放入集合 U＝{1}，此时与集合相邻的边为{(1,2),(1,3),(1,4),(1,5)}，更新 mincount[2]、mincount[3]、mincount[4]、mincount[5]的值，如图 5.42 所示。

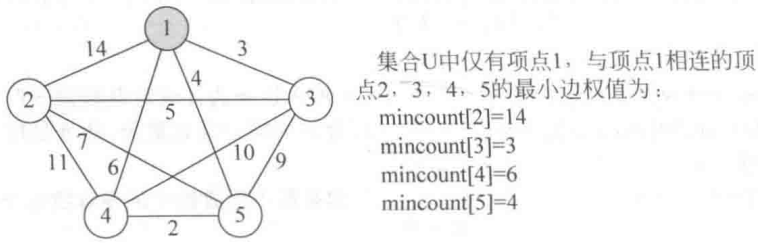


图 5.42

由于 mincount[3]＝3 的值最小，将顶点 3 加入集合 U，即 U＝{1,3}，此时与顶点 3 相邻的顶点为 2,4,5，更新 mincount[2]、mincount[4]、mincount[5]的值，如图 5.43 所示。

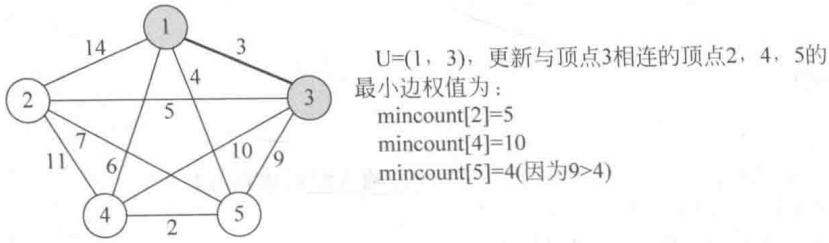


图 5.43

由于 mincount[5]=4 最小，将顶点 5 加入集合 U，即 U={1,3,5}，此时与顶点 5 相邻的顶点为 4,2，更新 mincount[4]、mincount[2]的值，如图 5.44 所示。

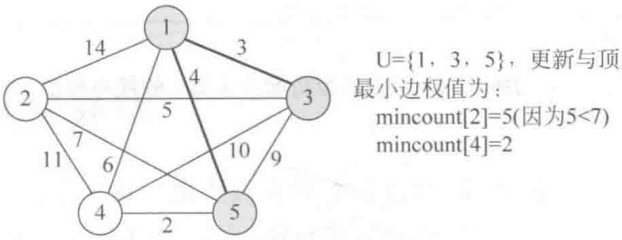


图 5.44

由于 mincount[4]=2 最小，将顶点 4 加入集合 U，即 U={1,3,4,5}，此时与顶点 4 相邻的顶点为 2，更新 mincount[2]的值，如图 5.45 所示。

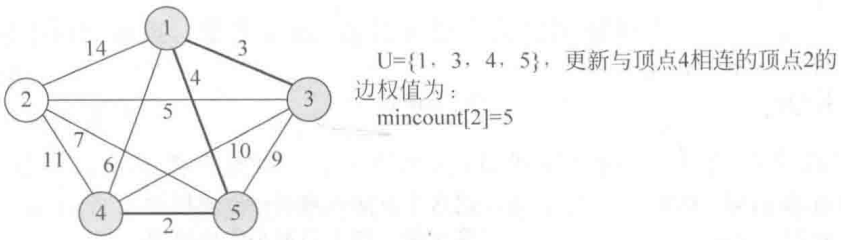


图 5.45

此时仅剩一个顶点 2，连接顶点 2，如图 5.46 所示。

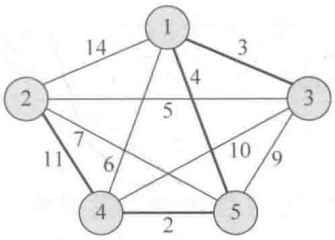


图 5.46

参考程序如下。

```
/*
程序名称：Prims 算法的演示程序
程序说明：这里使用无向图
*/
#include<iostream>
#define MAXN 2001
#define INF 999999999
using namespace std;
int n,e;
int w[MAXN][MAXN];              /*两点之间边的权值*/
int mincount[MAXN];             /*从初始顶点到该顶点的最小权值*/

```

```
13  void prim(int s)
14  {
15    int i,j,count=0,min;                /*count 为生成树所有边的权值和*/
16    int k;
17    for(i=1;i<=n;i++)
18      mincount[i]=w[s][i];   /*初始化,设 w(1,i)为初始点 k 到 i 的最小权值,如果没有则为
19  +∞*/
20    mincount[s]=0;
21
22    for(i=1;i<n;i++)
23    {
24      min=INF;
25      for(j=1;j<=n;j++)
26       if(mincount[j]!=0 and mincount[j]<min)   //如该结点没有被加入到 SMT 中并且该点
27  权值为当前最小
28
29      {
30        min=mincount[j];
31        k=j;                          //记录该点
32      }
33
34      mincount[k]=0;                  //把这个点加入到最小生成树中
35      count+=min;                     //将这条边加入到最小生成树中
36
37      for(j=1;j<=n;j++)               /*修正初始点到每个点的最小权值*/
38       if(w[k][j]<mincount[j])
39         mincount[j]=w[k][j];
40    }
41    cout<<count<<"\n";
42  }
43
44  void init()
45  {
46    int i,j,tx,ty;
47    for(i=0;i<=MAXN;i++)
48      for(j=0;j<=MAXN;j++)
49        w[i][j]=INF;
50
51    cin>>n>>e;
52    for(i=1;i<=e;i++)
53    {
54      cin>>tx>>ty>>w[tx][ty];
55      w[ty][tx]=w[tx][ty];
56    }
57  }
58
59  int main()                          //演示过程,从标号为 1 的顶点开始构造生成树
60  {
61    init();
```

```
    prim(1);
    return 0;
}
```

Dijkstra 算法

墨老师一行手中有一张上古文明遗迹的地图，如图 5.47 所示。现在他们要从 A 点出发，找到一条最短的路径到 B 点，应该如何解决呢？

我们在这里可以采用一种叫 Dijkstra 的算法。Dijkstra 算法的规则为：设置顶点集合 S，首先将起始点加入该集合，然后根据起始点到其他顶点的路径长度，选择路径长度最小的顶点加入集合，根据所加入顶点更新源点到其他顶点的路径长度，然后再选取最小边的顶点。依次来做，直到求解出到达所有顶点的路径长度。

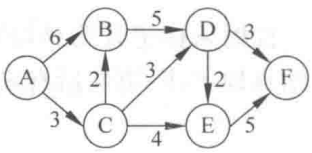
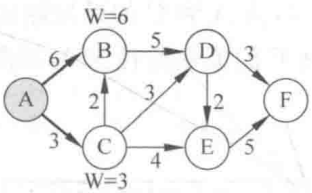


图 5.47

我们以样例来说明。

首先，将起点 A 标记为已访问，从 A 点出发，找出所有和 A 邻近，且有路径的城市即 B 和 C。我们在 B 和 C 上标记现在的路径总和 W，则从 A 到 B 的总和路径为 6，从 A 到 C 的总和路径为 3。(A 点到 A 点自身为 0)如图 5.48 所示。

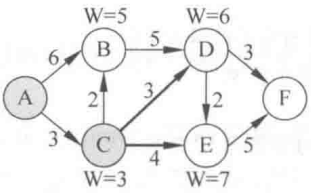


找出未标记且最短的结点C

结点	A	B	C	D	E	F
A至各点距离W	0	6	3	Max	Max	Max
标记已访问						

图 5.48

标记 C 结点为已访问，再从总和路径最短的 C 出发，找出与 C 有路径所有未标记结点即 D 和 E，更新 A 至各未标记结点的最小距离，这时我们会发现，从 A 经由 C 到达 B 的总和路径为 5，比 A 直接到 B 的路径 6 还短，所以我们决定选择从 A 经 C 到达 B，如图 5.49 所示。



初始值d(A,D)=Max
d(A,C)+d(C,D)=6
因此更新最小值为6

初始值d(A,B)=6
d(A,C)+d(C,B)=5
因此更新最小值为5

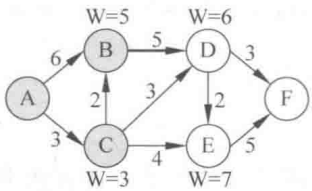
初始值d(A,E)=Max
d(A,C)+d(C,E)=7
因此更新最小值为7

结点	A	B	C	D	E	F
A至各点距离	0	5	3	6	7	Max
标记已访问						

更新所有未标记结点的最短路径

图 5.49

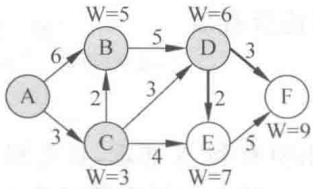
此时未标记且路径最短的结点为B,将B标为已访问,再找到与B点有路径的所有未标记结点即D,更新所有未标记结点的最短路径。可以发现没有值被改变,如图5.50所示。



结点	A	B	C	D	E	F
A至各点距离	0	5	3	6	7	Max
标记已访问						

图 5.50

此时未标记且路径最短的结点为D,将D标为已访问,找到与D点有路径的所有未标记结点即E,F,更新所有未标记结点的最短路径,如图5.51所示。

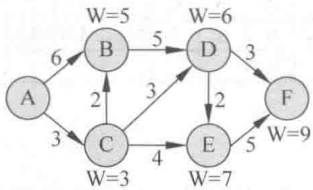


初始值d(A,F)=Max
d(A,D)+d(D,F)=9
因此更新最小值为9

结点	A	B	C	D	E	F
A至各点距离	0	5	3	6	7	9
标记已访问						

图 5.51

以此类推,最后结果为:从A到B的最短路径为5,从A到C的最短路径为3,从A到D的最短路径为6,从A到E的最短路径为7,从A到F的最短路径为9,如图5.52所示。



结点	A	B	C	D	E	F
A至各点距离	0	5	3	6	7	9
标记已访问						

图 5.52

参考代码如下。

```
//Dijkstra 算法
#include<iostream>
using namespace std;
#define Max 0xffffff
int n;
int map[1000+1][1000+1];          //存储图
int visit[1000+1];                //设置访问标记
int d[1000+1];                    //源点到各结点的最小距离

void Dijkstra(int x)              //求从 x 点到其他的单源最短路径
{
  int i,j,Min,p;
  for(i=1;i<=n;i++)               //算出从起始点到各顶点的最短距离
```

```
14        d[i]=map[x][i];
15      visit[x]=1;                          //设源点为已访问过
16      d[x]=0;                              //自身到自身为零
17      for(i=1;i<n;i++)
18      {
19        Min=Max;                           //最小边
20        for(j=1;j<=n;j++)                  //找出总和最短的路径
21          if(!visit[j] && Min>d[j])        //找没查找过的邻接边
22          {
23            p=j;                           //找出最短边
24            Min=d[j];
25          }
26        visit[p]=1;
27        for(j=1;j<=n;j++)                  //更新未查找顶点的最小距离总和
28          if(!visit[j] && Min+map[p][j]<d[j])
29            d[j]=Min+map[p][j];
30      }
31      for(int i=1;i<=n;i++)                //打印出从点 x 到任意点的最短距离
32        cout<<d[i]<<' ';
33      cout<<endl;
34    }
35
36    void init()
37    {
38      cin>>n;                              //n 个结点
39      int i,j;
40      for(i=1;i<=n;i++)
41        for(j=1;j<=n;j++)
42        {
43          cin>>map[i][j];
44          if(map[i][j]==0)                 //若不存在路径,则为+∞。计算时要小心溢出
45            map[i][j]=Max;
46        }
47    }
48
49    int main()
50    {
51      init();                              //读入图
52      Dijkstra(1);
53      return 0;
54    }
```

Dijkstra 算法的堆优化

朴素的 Dijkstra 算法使用邻接矩阵来实现，其外循环 n－1 次，内循环 n 次，时间复杂度为 $O(n^2)$。在数据量大的情况下，可以采用二叉堆来存储，从而使复杂度降低到 O(nlogn)。

参考代码如下所示。

```
//Dijkstra 堆优化
#include<iostream>
using namespace std;
#define Max 0xffffff
int n,m,ms;
int map[1000+1][1000+1];                //存储图
int visit[1000+1];                      //设置访问标记
int d[1000+1];                          //源点到各结点的最小距离
int heap[1000+1];                       //堆
int pos[1000+1],posh[1000+1];
//pos[i]存储堆中第 i 点在图中的位置,posh[i]存储图中的点 i 在堆中的位置

void minheap(int x)                     //维护最小堆
{
  int l,r,t;
  t=x;
  l=t<<1;
  r=(t<<1)+1;
  if(l<=ms && heap[l]<heap[t])
    t=l;
  if(r<=ms && heap[r]<heap[t])
    t=r;
  if(t!=x)
  {
    swap(heap[x],heap[t]);
    swap(posh[pos[x]],posh[pos[t]]);
    swap(pos[x],pos[t]);
    minheap(t);
  }
}

void build()                            //构建堆
{
  int i;
  for(i=1;i<=n;i++)
    pos[i]=posh[i]=i;
  for(i=n>>1;i>=1;i--)
    minheap(i);
}

void insert(int x)                      //改变堆中某个元素的值
{
  while(x>1 && heap[x]<heap[x>>1])
  {
    swap(heap[x],heap[x>>1]);
    swap(posh[pos[x]],posh[pos[x>>1]]);
    swap(pos[x],pos[x>>1]);
    x>>=1;
  }
}
```

```
51
52  void Dijkstra(int x)                    //求从 x 点到其他的单源最短路径
53  {
54    int i,j,Min,p,q;
55    for(i=1;i<=n;i++)                     //初始化堆中元素
56      heap[i]=map[1][i];
57    heap[x]=0;
58    build();                              //建堆
59    for(i=1;i<n;i++)
60    {
61      p=pos[1];                           //将最小的元素提出来,并标记该点
62      q=heap[1];
63      visit[p]=1;
64      swap(heap[1],heap[ms]);
65      swap(posh[pos[1]],posh[pos[ms]]);
66      swap(pos[1],pos[ms]);               //将最大的元素和最小的元素交换,这样才能维护堆
67      ms--;                               //每确定一个点,堆都会缩小 1
68      minheap(1);                         //维护最小堆
69      for(j=1;j<=n;j++)                   //更新未查找顶点的最小距离总和
70        if(!visit[j] && q+map[p][j]<heap[posh[j]])
71        {
72          heap[posh[j]]=q+map[p][j];
73          insert(posh[j]);                //修改 j 点在堆里的位置
74        }
75    }
76    for(i=1;i<=n;i++)                     //打印最短路径
77      cout<<heap[posh[i]]<<' ';
78    cout<<endl;
79  }
80
81  void init()
82  {
83    cin>>n;                               //n 个结点
84    int i,j;
85    for(i=1;i<=n;i++)
86      for(j=1;j<=n;j++)
87      {
88        cin>>map[i][j];
89        if(map[i][j]==0)                  //若不存在路径,则为+∞。计算时要小心溢出
90          map[i][j]=Max;
91      }
92    ms=n;
93  }
94
95  int main()
96  {
97    init();                               //读入图
98    Dijkstra(1);
99    return 0;
100 }
```

【题目描述】 最优乘车(NOI94)

H 城是一个旅游胜地,每年都有成千上万的人前来观光。为方便游客,巴士公司在各个景点都设置了巴士站并开通了一些单程巴士线路。每条单程巴士线路从某个巴士站出发,依次途经若干个巴士站,最终到达终点巴士站。楚继光最近到 H 城旅游,他很想去 S 公园游玩,但如果从他所在的饭店没有一路巴士可以直接到达 S 公园,现用整数 1,2,…,N 给 H 城的所有巴士站编号,约定楚继光所在饭店巴士站编号为 1,…,S,公园巴士站编号为 N,编写程序,帮助楚继光寻找最优乘车方案,使他在从饭店乘车到 S 公园的过程中换车次数最少。

【输入格式】

输入文件第一行有两个数字 M,N(1≤M≤100,1<N≤500),表示开通了 M 条单程巴士线路,总共有 N 个车站。从第二行到第 M 行依次给出了第 1 条到第 M 条巴士线路的信息。其中第 i+1 行给出的是第 i 条巴士线路的信息,从左至右按运行顺序依次给出了该线路上所有站号,相邻两个站号之间用一个空格隔开。

【输出格式】

输出文件只有一行,如果无法乘巴士从饭店到达 S 公园,则输出"NO",否则输出你的程序所找到的最少换车次数,换车次数为 0 表示不需换车即可到达。

【输入样例】

3 7

6 7

4 7 3 6

2 1 3 5

【输出格式】

2

【算法分析】

根据题目所给线路构建有向图模型,以给定一条线路为例,建立图论模型,例如 1 2 3 4,表明 1 站可以到达 2,3,4; 2 站可以到达 3,4; 3 站可以到达 4,权值为 1。对于样例中的数据,可建立图 5.53。

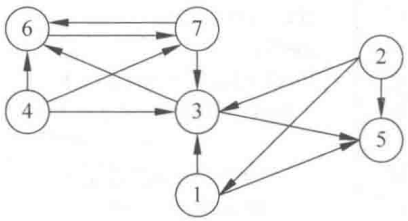


图 5.53

分析此模型,可以求从站 1 到 N 的最短路径长度,而换乘次数是最短路的长度减 1,因此本题就转换成为求两点间的最短路长度。

参考程序如下。

```
//最优乘车
#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;

int a[500+1][500+1];
```

```
8   char v[500+1];
9   long s[500+1];
10  int t[500+1];
11  int p,x,y,i,j,Max,k,m,n,l;
12
13  int main()
14  {
15     freopen("input.txt","r",stdin);
16     freopen("output.txt","w",stdout);
17     scanf("%d%d",&m,&n);
18     memset(a,0,sizeof(a));
19     for(l=1;l<=m;l++)
20     {
21        j=0;
22        memset(t,0,sizeof(t));
23        while(!feof(stdin))
24        {
25           j++;
26           scanf("%d",&t[j]);
27        }
28        for(i=1;i<=j-1;i++)
29           for(k=i+1;k<=j;k++)
30              a[t[i]][t[k]]=1;
31     }
32     memset(v,0,sizeof(v));
33     v[1]=1;
34     for(j=2;j<=n;j++)
35     {
36        Max=99999999;
37        for(i=1;i<=n;i++)
38           if(!v[i]&&(a[1][i]!=0)&&(a[1][i]<Max))
39           {
40              Max=a[1][i];
41              p=i;
42           }
43        v[p]=1;
44        for(i=1;i<=n;i++)
45           if((a[1][p]!=0)&&(a[p][i]!=0))
46              if((a[1][i]>a[1][p]+a[p][i])||(a[1][i]==0))
47                 a[1][i]=a[1][p]+a[p][i];
48     }
49     if(a[1][n]-1==-1)
50        printf("NO\n");
51     else
52        printf("%d\n",a[1][n]-1);
53     return 0;
54  }
```

Bellman-Ford 算法

用 Dijkstra 算法求有向图的单源最短路径，条件是图中任意一条边都是正权。但在求单源最短路中可能存在有负权边的有向图，这时 Dijkstra 算法就无法计算了，因为总可以顺着“最短路”再穿过负权回路从而得到更小的最短路径，解决这种问题，我们要使用 Bellman-Ford 算法。

我们通过实例逐步分解 Bellman-Ford 算法来理解其本质，例如根据图 5.54 求顶点 1 到其他各点的最短距离。

设 W[i,j]为各顶点之间的距离，则其邻接表如图 5.55 所示。

设 D[i]为从顶点 1 至顶点 i 的最短路径，设 D[1]=0，其他各元素值为无穷大，则初始化的数组如图 5.56 所示。

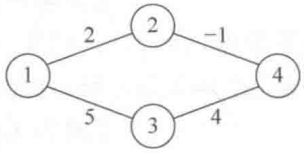


图 5.54

j \ i	1	2	3	4
1	∞	2	5	∞
2	2	∞	∞	-1
3	5	∞	∞	4
4	∞	-1	4	∞

图 5.55

下标	1	2	3	4
D[i]	0	∞	∞	∞

图 5.56

现在进行迭代求解，反复对每条边进行松弛操作，使得 D[i]的值逐步逼近实际值。

(1)求 D[2]的值：

因为 D[1]+W[1,2]<D[2]+W[2,2]<D[3]+W[3,2]<D[4]+W[4,2]，所以 D[2]=D[1]+W[1,2]=2。

(2)求 D[3]的值：

因为 D[1]+W[1,3]<D[2]+W[2,3]<D[3]+W[3,3]<D[4]+W[4,3]，所以 D[3]=D[1]+W[1,3]=5。

(3)求 D[4]的值：

因为 D[2]+W[2,4]<D[1]+W[1,4]<D[3]+W[3,4]<D[4]+W[4,4]，所以 D[4]=D[2]+W[2,4]=1。

可以看到，我们共进行了 3 次(即 n−1 次)松弛操作，每次松弛操作确定了一条最短路径，D 数组值如图 5.57 所示。

下标	1	2	3	4
D[i]	0	2	5	1

图 5.57

继续进行松弛操作(仍然是 n−1 次)以检查是否存在负权边，如发现 D[i]的值仍然可以继续变小，则说明存在负权边并退出，否则输出最短路径。显然，我们很容易就可以看出：

D[4]+W[4,2]<D[4]，这足以说明该图存在负权边了。

综上所述，Bellman-Ford 算法流程分为三个阶段。

(1) 初始化：将除源点外的所有顶点的最短距离估计值 D[i]=+∞，D[sourse] =0。

(2) 迭代求解：反复对每条边进行松弛操作，使得每个顶点的最短距离 D[i]估计值逐步逼近其最短距离；(运行 n−1 次)。

(3) 检验负权回路：通过松弛操作判断每一条边的两个端点是否收敛。如果存在未收敛的顶点，则算法返回 false，表明问题无解；否则算法返回 true，并且输出 D[i]的值。

实际上，松弛操作的原理是基于“三角形两边和大于第三边”这条著名的定理。其时间复杂度为 O(V E)[其中 V 为图的结点数，E 为图的边数，如果图以邻接矩阵存储，则时间复杂度为 $O(V^3)$]，所以通常并不用这种算法来求单源点最短路径。但是由于 Bellman-Ford 算法适用于负权边，并且还能够判断图中是否存在负权环，所以在一些特殊的情况下会用到这种算法。

我们通过下面这道例题给出 Bellman-Ford 算法的标准程序。

【题目描述】 虫洞

对上古文明遗迹的探索加速了魔法世界的科技进步，魔法历 2163 年，虫洞被发现了。虫洞是时空中连接两个星系的子空间通道。虫洞有一些特别的性质：

(1) 虫洞是单向通道。

(2) 通过虫洞的用时可忽略。

(3) 虫洞有两个口，每个口都在星系里。

(4) 从太阳系出发总是能通过一系列的虫洞到达任何其他星系。

(5) 对于任何一对星系，其间每个方向上至少有一个虫洞。

(6) 没有虫洞两个口都在一个星系里。

所有虫洞都有自己恒定的时差，例如，某个虫洞可以让人到未来 15 年后，另一个虫洞可能让人回到 42 年前。

一个科学家想通过虫洞研究宇宙诞生时的大爆炸，她准备通过虫洞实现，她希望到达一个虫洞环(虫洞环是一系列首尾相连的虫洞)并在环上走一圈后能让她回到过去，如此一来，她只要在环上通行足够多次，就能回到宇宙的开端，亲眼看到大爆炸。写一个程序帮助她找到这样的虫洞环。

【输入格式】

第一行为星系个数 n($1\leqslant n\leqslant 1000$)和虫洞个数 m($1\leqslant m\leqslant 2000$)。星系编号从 0(太阳系)到 n−1。下面 m 行描述 m 个虫洞，格式均为 x,y,t。其中 x,y 是虫洞两个口位于的星系编号，t($-1000\leqslant t\leqslant 1000$)为虫洞的时差。

【输出格式】

如找到合适的虫洞环，输出 Possible，否则输出 Not possible。

【输入样例 1】

```
3 3
0 1 1000
1 2 15
2 1 -42
```

【输入样例 1】

```
Possible
```

【输入样例 2】

```
4 4
0 1 10
```

1 2 20

2 3 30

3 0 −60

【输入样例 2】

Not possible

【算法分析】

我们以星系为顶点、虫洞为边作图 G，由于“虫洞是单向通道”，说明 G 为有向图；“从太阳系出发总是能通过一系列的虫洞到达任何星系”，说明从源点出发能到达任何其他顶点；“对于任何一对星系，其间每个方向上至多有一个虫洞”说明 G 不含平行边；“没有虫洞两个口都在一个星系里”，说明 G 不含自环。所以，G 是一个带权简单有向图，且每个顶点是源点可到达的。

本题要求判断是否存在一个虫洞环，走上一圈后，能回到过去，实际即是求 G 的负权回路，如果存在负权回路，则通过反复通行该负权回路，即可回到无限的过去。可以用 Bellman-Ford 算法。

参考程序如下所示。

```
1   //虫洞
2   #include <iostream>
3   using namespace std;
4   int w[1000+1][1000+1],d[1000+1];
5   int n,m;
6   char change;
7
8   void init()
9   {
10    cin>>n>>m;                    //读入星系数和蛀洞的个数
11    int x,y,v,i,j;
12    for(i=1;i<=n;i++)
13      for(j=1;j<=n;j++)
14        w[i][j]=INT_MAX;
15    for(i=1;i<=m;i++)
16    {
17      cin>>x>>y>>v;
18      w[x][y]=v;                  //注意此为单向通道
19    }
20  }
21
22  void bellman_ford(int x)
23  {
24    int i,j,k;
25    for(i=1;i<=n;i++)
26      d[i]=w[x][i];
27    d[x]=0;
28    for(k=1;k<=n-1;k++)           //松弛 n−1 次
29      for(j=1;j<=n;j++)           //枚举每个点，每个点都有可能会更新最短路径
30        for(i=1;i<=n;i++)         //经过某点 i 到达 j
```

```
31          if((w[i][j]!=INT_MAX)&&(d[i]!=INT_MAX)&&(d[j]>d[i]+w[i][j]))
32            d[j]=d[i]+w[i][j];
33    change=1;                          //判断有无回路,默认不存在,change=1
34    for(i=1;i<=n;i++)                  //松弛操作判断是否存在负权回路
35      for(j=1;j<=n;j++)
36        if((w[i][j]!=INT_MAX)&&(d[i]!=INT_MAX)&&(d[j]>d[i]+w[i][j]))
37  //如果d[j]还能缩短,则存在负权回路
38
39          change=0;                    //设置 change=0
40    if(change)
41      cout<<"Not possible"<<endl;
42    else
43      cout<<"Possible"<<endl;
44  }
45
46  int main()
47  {
48    init();
49    bellman_ford(1);                   //从顶点 1 出发
50    return 0;
51  }
```

Bellman-Ford 算法的优化

分析 Bellman-Ford 算法,可以看出其最外层循环的迭代次数为 n−1,但实际上,如果不存在负权回路,一般情况下其外层循环次数应该是远远小于 n−1 的。

例如在某一遍迭代中,如松弛操作未被执行,则说明该遍迭代所有的边都没有被松弛,这实际上即表示任意两点间的距离在以后的迭代中已不再可能被缩小了,因此其迭代过程应该被提前结束。

我们可以设定一个布尔型标志变量 relaxed,初值为 false。在某次迭代过程中,如有边被成功松弛时,则 relaxed 设置为 true。否则提前结束外层循环。

优化后的 Bellman-ford 函数如下所示。

```
1   bool Bellman_ford(int x)
2   {
3     int i,j,k;
4     bool relaxed;                    //标记是否进行了松弛操作
5     for(i=1;i<=n;i++)                //初始化
6       d[i]=INT_MAX;
7     d[x]=0;
8     for(i=1;i<n;i++)
9     {
10      relaxed=0;                     //默认未进行松弛操作
11      for(j=1;j<=n;j++)
12        for(k=1;k<=n;k++)
13          if(w[j][k]!=INT_MAX && d[k]!=INT_MAX && d[j]>d[k]+w[j][k])
14          {
```

```
15                d[j]=d[k]+w[j][k];
16                relaxed=1;              //进行了松弛操作
17            }
18        if(!relaxed)                    //若未进行松弛操作,跳出循环
19            break;
20    }
21    for(i=1;i<=n;i++)                   //验证是否有负环
22        for(j=1;j<=n;j++)
23            if(w[j][k]!=INT_MAX && d[k]!=INT_MAX && d[j]>d[k]+w[j][k])
24                return 1;               //有负环
25    return 0;                           //无负环
26 }
```

优化后的算法的时间复杂度已经和用堆优化的 Dijkstra 算法相近了，而且其编码复杂度低，能处理负边权情况，因此该算法是非常优秀的。但要注意的是：在处理有负权回路的测试数据时，由于每次都会有边被松弛，因而不可能提前终止外层循环。其时间复杂度仍旧为 O(VE)。

SPFA

SPFA(Shortest Path Faster Algorithm)是 Bellman-Ford 算法的进一步优化，其时间复杂度一般认为是 O(kE)，可以证明 $k \leqslant 2$。

SPFA 的实质与 Bellman-Ford 算法一样，每次都要去更新最短路径的估计值。但 SPFA 对 Bellman-Ford 算法优化的关键之处在于意识到：只有那些在前一遍松弛中改变了距离估计值的点，才可能引起它们的邻接点的距离估计值的改变。

实际编写代码时，我们是通过队列来缩小搜索范围的。首先将各点距离估计值设为 $+\infty$，并将起始点加入队列。如果通过队列中的点 i 到相邻点 j 的距离小于原来到点 j 的距离，即 d[j]>d[i]+w[i][j]，则 d[j]=d[i]+w[i][j]，并将 j 点加入队列。当队列为空时，即已求出从起始点到任意点的最短距离。为了防止同一个点多次出现在队列里，需要对该点作标记以确定该点是否存在于队列中。

参考程序如下所示。

```
1   //SPFA
2   #include <iostream>
3   #define MAX 99999999
4   using namespace std;
5   struct node   //用链表存储边降低空间复杂度,如用数组可降低代码复杂度
6   {
7       int n;
8       int v;                          //边的权值
9       node* next;
10      node()                          //初始化
11      {
12          n=0;
13          next=NULL;
```

```
14      }
15  }* e[1001];                          //存放每一个点到别的点的边
16  int d[1001];                         //估计值
17  bool c[1001];                        //判断该顶点是否存在于队列
18  int line[2002];                      //队列
19  int rear;                            //队列末尾
20  int n,m;                             //n 为顶点数,m 为边数
21
22  void init()
23  {
24      cin>>n>>m;
25      node* p;
26      int i,j;
27      int x,y,v;
28      for(i=1;i<=m;i++)
29      {
30          cin>>x>>y>>v;
31          p=new node();
32          p->n=y;
33          p->v=v;
34          if(e[x]==NULL)
35              e[x]=p;
36          else
37          {
38              p->next=e[x]->next;
39              e[x]->next=p;
40          }
41      }
42  }
43
44  void add(int x)
45  {
46      rear++;
47      line[rear]=x;
48  }
49
50  void SPFA(int x)
51  {
52      int i,j;
53      node* p;
54      add(x);                          //将起始点加入队列
55      for(i=1;i<=n;i++)
56          d[i]=MAX;                    //初始化估计值
57      d[x]=0;
58      for(i=1;i<=rear;i++)             //读取队列
59      {
60          p=e[line[i]];                //和 line[i]点连接的边
61          while(p!=NULL)               //如果还有边的话
62          {
63              if(d[line[i]]+p->v<d[p->n])  //若通过 line[i]点到达 p->n 点的距离比原来短
```

```
        {
          d[p->n]=d[line[i]]+p->v;    //更新距离
          if(!c[p->n])                //如果 p->n 点不在队列里
          {
            c[p->n]=1;                //标记为已加入队列
            add(p->n);                //加入队列
          }
        }
        p=p->next;                    //继续 i 点的下一点边
      }
      c[line[i]]=0;                   //line[i]点出队
    }
    for(i=1;i<=n;i++)                 //打印从 x 点到任意点的最短距离
      cout<<d[i]<<' ';
    cout<<endl;
}

int main()
{
    init();
    SPFA(1);
    return 0;
}
```

需要注意的是：仅当图不存在负权回路时，SPFA 能正常工作。如果图存在负权回路，由于负权回路上的顶点无法收敛，总有顶点在入队和出队往返，队列无法为空，这种情况下 SPFA 无法正常结束。

判断负权回路的方案很多，常用的方法是记录每个结点进队次数，超过 n 次表示有负权。

还有种方法是记录这个结点在路径中所处的位置 ord[i]，每次更新的时候 ord[i]=ord[x]+1，若超过 n，则表示有负圈。第二种方法速度要优于第一种方法。

Floyd 算法

如想求得每对顶点之间的最短路径，我们可以以一个顶点为源点，重复执行 Dijkstra 算法 n 次，这样可以求得每一对顶点之间的最短路径。时间复杂度是 $O(n^3)$。

另外一种算法是 Floyd 算法，其本质是动态规划，时间复杂度也是 $O(n^3)$，但形式较简单。

设 D[i,j]为从 i 点至 j 点的最短路径，则其动态转移方程为

$$D[i,j]=\max(D[i,k]+D[k,j])(1\leqslant k\leqslant n)$$

我们以下面的一道例题来展示 Floyd 算法。

【题目描述】 最小交通费用问题

某游乐城内有 N 个活动场所，某些活动场所之间有公路连接，任意两个场所可以通过公路直接或者间接到达，并且任意有公路连接的两个场所之间，来回使用的交通工具不一

样，所以费用也不一样，张琪曼和李旭琳在 A 场所时，李旭琳有点累，想休息一下，此时张琪曼只能单独去 B 场所，请设计一条来回最小交通费用的线路。

【输入格式】

第一行有两个数 N，M(N 为场所个数；M 为场所间交通路线图)。

第二行至第 M+1 行分别有三个数字，前两个为场所编号；第三个为它们的交通费用。

第 M+2 行有两个数字，为两个求解的场所编号。

【输出格式】

一个整数(最小交通费)

【输入样例】

3 5

1 2 4

2 1 6

1 3 11

3 1 3

2 3 2

1 2

【输出样例】

9

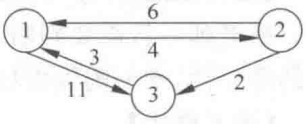


图 5.58

【算法分析】

此题为多源最短路径问题，使用 Floyd 算法即可。样例数据如图 5.58 所示。

参考代码如下所示。

```
//Floyd 算法
#include <iostream>
#define maxn 100
using namespace std;
int n,s,t;
int a[maxn+1][maxn+1];

void floyd()
{
  int i,j,k;
  for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
        if(a[i][k]!=-1 && a[k][j]!=-1)//要注意选择 MAX_INT 相加会变成负数
          if(a[i][k]+a[k][j]<a[i][j])
            a[i][j]=a[i][k]+a[k][j];
}

void init()
{
  int m,i,u,v;
```

```
22    cin>>n>>m;
23    for(u=1;u<=n;u++)
24      for(v=1;v<=n;v++)
25        a[u][v]=-1;                    //此处不可设为 MAX_INT,否则相加会成负数
26    for(i=1;i<=m;i++)
27      cin>>u>>v>>a[u][v];
28    cin>>s>>t;
29  }
30
31  int main()
32  {
33    init();
34    floyd();
35    cout<<a[s][t]+a[t][s]<<endl;
36    return 0;
37  }
```

次小生成树算法

【题目描述】 唯一的最短路(PKU 1679)

墨老师一行需要判断上古文明遗迹的最短路径是否唯一,简单说来,就是判断生成最小树的唯一性,唯一则输出权值,不唯一则输出 Not Unique!。

【输入格式】

第一行两个整数 n 和 m,即 n(1≤ n≤100)个点 m 条边。以下 m 行包括三个数(xi,yi,wi),表示 xi 与 yi 两点相连,其权值为 wi,对于任何两点,最少有一条边相连。

【输出格式】

输出权值或 Not Unique!

【输入样例 1】

3 3

1 2 1

2 3 2

3 1 3

【输出样例 1】

3

【输入样例 2】

4 4

1 2 2

2 3 2

3 4 2

4 1 2

【输出样例 2】

Not Unique!

【算法分析】

求次小生成树即求第二小的生成树，次小生成树可由最小生成树换一条边得到。一般采用的是求出最小生成树后，依次删除最小生成树上的每一条边，然后生成 n－1 次最小生成树，记录下这个过程中的最小的生成树的值，那么这个就是第二小生成树了。用 Kruskal 这种算法的复杂度为 O(n×elog2e)，当图比较稠密时，复杂度接近 $O(n^3)$。

参考代码如下所示。

```
1   //次小生成树 1
2   #include<stdio.h>
3   #include<stdlib.h>
4   #include <iostream>
5   using namespace std;
6   #define maxn 10000
7
8   struct data
9   {
10    int x;
11    int y;
12    int w;
13  }sides[maxn+1];                          //记录边的起点和终点，还有权值
14  int set[110];                            //并查集的数组
15  bool flag[maxn+1];                 //记录某条边在最小生成树中是否使用过，用过则赋值为 1
16
17  int cmp(const void * a,const void  * b)
18  {
19    return ((data  * )a)->w - ((data  * )b)->w;
20  }
21
22  int find(int x)                          //寻找根结点；
23  {
24    if(set[x]==x)
25      return x;
26    set[x]=find(set[x]);                   //寻找的过程中，更新 set[x]的值，路径压缩
27    return set[x];
28  }
29
30  int main()
31  {
32    int n,i,j,ans,t,ans1,rx,ry,before,m;
33    bool OK;                               //标志是否找到答案
34    scanf("%d%d",&n,&m);
35    OK=0;
36    for(i=1;i<=m;i++)
37      scanf("%d%d%d",&sides[i].x,&sides[i].y,&sides[i].w);
38    qsort(&sides[1],m,sizeof(sides[0]),cmp);   //贪心，排序，按权值
39    for(i=1;i<=n;i++)
40      set[i]=i;                            //并查集初始化
41    for(i=1;i<=m;i++)                      //初始化为未被使用过
42      flag[i]=0;
```

```
43    ans=0;
44    i=0;
45    j=0;
46    while(i<n-1)                          //kruscal
47    {
48      j++;
49      rx=find(sides[j].x);                //寻找根结点
50      ry=find(sides[j].y);
51      if(rx!=ry)
52      {
53        i++;
54        ans+=sides[j].w;
55        set[rx]=set[ry];                  //合并集合
56        flag[j]=1;
57      }
58    }
59    if(m==n-1)                            //唯一最小生成树,直接输出结果
60    {
61      printf("%d\n",ans);
62      return 0;
63    }
64    //开始寻找次小生成树
65    before=1;
66    flag[before]=0;
67    for(t=1;t<=n-1;t++)                   //进行 n-1 次删边的操作
68    {
69      for(i=1;i<=n;i++)                   //重更新并查集数组
70        set[i]=i;
71      i=0;
72      j=0;
73      ans1=0;
74      while(i<n-1)                        //同样的操作,求最小生成树
75      {
76        j++;
77        if(j!=before)                     //每次删除一条边
78        {
79          rx=find(sides[j].x);
80          ry=find(sides[j].y);
81          if(rx!=ry)
82          {
83            i++;
84            ans1+=sides[j].w;
85            set[rx]=set[ry];              //并到同一集合
86          }
87        }
88      }
89      if(ans1==ans)
90      {
91        OK=1; break;
92      }
```

```
93        for(i=before+1;i<=m;i++)                    //寻找一条边,赋值 before,给删除用的
94          if(flag[i])                               //如果该边在最小生成树中
95          {
96            before=i;                               //确定下一个要删除的边
97            flag[before]=0;
98            break;
99          }
100     }
101     if(OK)
102       printf("Not Unique\n");
103     else
104       printf("%d\n",ans);
105     system("pause");
106     return 0;
107   }
```

但有一种更简单的方法：先求最小生成树 T,枚举添加不在 T 中的边,则添加后一定会形成环。例如图 5.59 中的最小树,添加了结点 3 到结点 5 的边后,形成了环。

找到环上边值第二大的边(即环中属于 T 中的最大边),把它删掉,计算当前生成树的权值,取所有枚举修改的生成树的最小值,即为次小生成树。

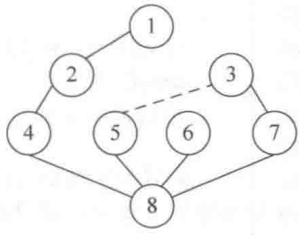


图 5.59

这种方法在实现时有更简单的方法：首先求最小生成树 T,然后从每个结点 u 遍历最小生成树 T,用一个二维数组 Max[u][v]记录结点 u 到结点 v 的路径上边的最大值(即最大边的值)。然后枚举不在 T 中的边(u,v),计算 T− Max[u][v] + w(u,v)的最小值,即为次小生成树的权值。显然,这种方法的时间复杂度为 $O(n^2+e)$。

由于只用求一次最小生成树,可以用最简单的 Prim,时间复杂度为 $O(N^2)$。算法的瓶颈不在求最小生成树,而在 $O(N^2+e)$的枚举加边修改,所以用更好的最小生成树算法是没有必要的。

顺便说一下,k 小生成树的算法可由此衍生,即求得次小生成树后再按照此法求次次小生成树,直到求得 k 小生成树。

```
1   //次小生成树 2
2   #include<cstdio>
3   #include<cstring>
4   #include<cstdlib>
5   #include <iostream>
6   using namespace std;
7   const int inf = 0x3f3f3f3f;
8   const int MAXN = 105;
9
10  bool Tlink[MAXN][MAXN],vis[MAXN];   //Tlink 记录边是否在 MST 中,vis[i]表示 i 点是
11  否在 MST
12
13  int w[MAXN][MAXN];                  //w[i][j]为 i 到 j 的边的权值
```

```
14  int lowc[MAXN], pre[MAXN];
15  int Max[MAXN][MAXN];                    //Max[i,j]为 MST 中连接 i,j 唯一路中最大边权值
16  int n, m;
17
18  int MAX( int a, int b)
19  {
20      return a > b ? a : b;
21  }
22
23  int prim()
24  {
25    int i, j, p, k;
26    int minc, res = 0;
27    memset( vis, false, sizeof vis);
28    memset( pre, 0, sizeof pre);
29    memset( Max, 0, sizeof Max);
30    vis[1] = 1;                           //从第一个结点开始
31    pre[1] = 1;                           //前驱是结点 1
32    for( i = 2; i <= n; i ++)             //初始化
33    {
34      lowc[i] = w[1][i];
35      pre[i] = 1;
36    }
37
38    for( i = 2; i <= n; i ++)             //prim 算法
39    {
40      minc = inf; p = -1;
41      for( j = 1; j <= n; j ++)
42      {
43        if( !vis[j] && lowc[j] < minc)    //找出结点没有被加入 MST 的最小权值边
44        {
45          minc = lowc[j];
46          p = j;
47        }
48      }
49      vis[p] = true;                      //该结点加入 MST
50      res += minc;                        //最小树权值累加
51      Max[ pre[p] ][p] = minc;
52      Tlink[ pre[p] ][p] = true;          //将这条边标记为最小树的边
53      Tlink[p][ pre[p] ] = true;          //将这条边标记为最小树的边
54      for( k = 1; k <= n; k ++)
55        Max[k][p] = MAX( Max[ pre[p] ][p], Max[k][p]);  //更新连接该边的最大值
56      for( j = 1; j <= n; j ++)           //更新 p 点到 j 点的最小边权值
57        if( !vis[j] && lowc[j] > w[p][j])
58        {
59          lowc[j] = w[p][j];
60          pre[j] = p;                     //确定前驱结点
61        }
62    }
63    return res;
```

```
64  }
65
66  int main()
67  {
68    int s, e, t, Ans, ans;
69    bool ok = true;                          //是否唯一最小生成树的标志
70    cin>>n>>m;
71    for( int i = 1; i <= n; i ++)
72      for( int j = 1; j <= n; j ++)
73        w[i][j] = inf;
74    memset( Tlink, false, sizeof Tlink);
75    for( int i = 1; i <= m; i ++)
76    {
77      cin>>s>>e>>t;
78      w[s][e] = t;
79      w[e][s] = t;
80    }
81    Ans = prim();                            //prim 算法求出最小生成树权值
82    for( int i = 1; i <= n; i ++)
83    {
84      for( int j = 1; j <= n; j ++)
85      {
86        if( w[i][j] == inf || Tlink[i][j]) //如果该点在最小生成树或者无边则忽略
87          continue;
88        ans = Ans + w[i][j] - Max[i][j]; //Max[i,j]:MST 中连结 i,j 中最大边权值
89        if( Ans == ans)          //如果发现加非最小树的一条边后权值与原最小树权值相等
90        {
91          ok = false;                        //则非唯一最小树
92          break;
93        }
94      }
95      if( !ok)
96        break;
97    }
98    if( ok)
99      printf( "%d\n", Ans);
100   else
101     printf( "Not Unique!\n");
102   return 0;
103 }
```

请尝试用其他最小树生成法完成该题。

度限制生成树

【题目描述】 野餐(PKU 1639)

马戏团的小丑们有一个特异功能，无论一个车子有多小，他们都能钻进去，也就是说，一辆车子能够容纳无限个小丑。

现在小丑们要去一个公园野餐，他们住在不同的地方，为了节约路费(石油)，要使得所

有车子加起来走的路程最小，往往是几个小丑各自开车到另一个小丑家停车后，再挤在一辆车上开车到公园。

但是公园的停车位是有限制的，公园最多只能停 k 辆车。一旦某个小丑开车到了公园，那么就必须停在公园，不能再回去载其他小丑了。

求所有小丑开车的最短总路程。

【输入格式】

第一行是一个整数 n，表示小丑与小丑之间及与公园之间的连续数。以下 n 行表示从 A 地到 B 地的路程数，路是双向的，小丑数不会超过 20 人，并且地名的字符长度不会超过 10 个字符。每一个小丑家到公园的路是连通的，不会出现无解的情况。

【输出格式】

最小路径数

【输入样例】

10

Alphonzo Bernardo 32

Alphonzo Park 57

Alphonzo Eduardo 43

Bernardo Park 19

Bernardo Clemenzi 82

Clemenzi Park 65

Clemenzi Herb 90

Clemenzi Eduardo 109

Park Herb 24

Herb Eduardo 79

3

【输出样例】

Total miles driven: 183

【算法分析】

此题是最小生成树问题的扩展，之前的最小生成树中每个结点的度数是任意的，但如果对无向图中的某一个结点（例如 V_0）的度数做出限制，即 $d(V_0)\leqslant k$。那么把 $d(V_0)=k$ 的最小生成树称为无向图的最小 K 度限制生成树，其权值和记为 tree[k]。显然求无向图的最小度限制生成树等价于求 $\min\{tree[i] \mid 1\leqslant i\leqslant k\}$。

那么现在问题是，对于结点 V_0，K 有没有最小限制？观察图 5.60。

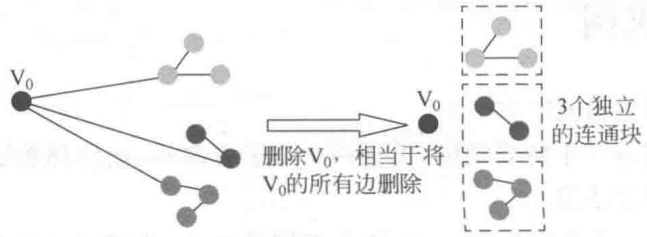


图 5.60

可以看到，将 V_0 的所有边删除后，右边出现了 3 个独立的连通块，可见，当 $d(V_0)<3$ 时，问题无解。即 V_0 的度最少应为 3。

可知，对于任意一个无向图，将 V_0 点去除后，原图剩余部分有 K_0 个连通块，那么当 $k<K_0$ 时肯定不存在 k 度最小度限制生成树。

现假设对于图 5.60 的 V_0，已求出 tree[3]的值，那么如何求 tree[4]，tree[5]，tree[6]，...的值呢？换言之，已知 tree[m]，是否能求出 tree[m+1]的值？

以图 5.61 为例，对于 V_0 来说，由于去掉 V_0 的几个连通块相对独立，因此不妨只考虑 1 个连通块的情况下的 tree[1]的值如何求。

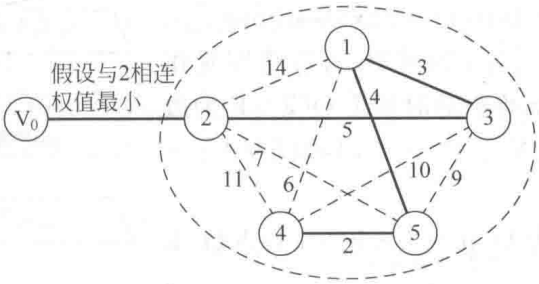


图 5.61

很显然，该图的最小 1 度限制生成树等价于求连通块的最小生成树，再从 V_0 枚举一条到连通块中的边使得权值和最小即可。

那么 2 度限制最小生成树如何求呢？我们尝试在 1 度限制最小生成树的基础上增加一条边以达到度限制要求。显然在增加一条边后，图 5.62 中出现了环，因此我们需要删除一条边(不与 V_0 相关联)。

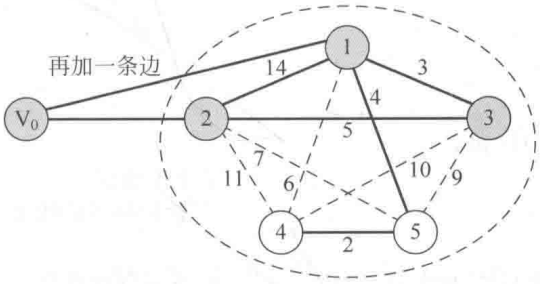


图 5.62

那么删除哪条边呢？显然取环中的最大权值边(不与 V_0 相关联)，即为 2 度限制最小生成树，如图 5.63 所示。

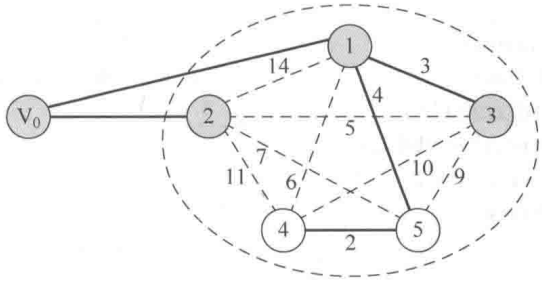


图 5.63

由此，可看出求解 K 度限制最小生成树的关键步骤为：

1. 增加一条从 V_0 出发的边，记作(V_0,i)。

2. 删除环中的最大边(a,b)。

3. 则 tree[k+1]=tree[k]+cost[V_0,i]=cost[a,b]。该操作称为差额最小添删操作。

则可得算法流程如图 5.64 所示。

但该算法流程中，最大的瓶颈在于找环中的边权最大值，我们需要使用动态规划进行预处理：

设 best[i]为 V_0 到 i 路径上与 V_0 无关联边权的最大值，则状态转移方程为 best[i]=max{best[pre[i]], g[pre[i]][i]}，其中 pre[i]为 i 的前驱，可以将图看作以 V_0 为根的一棵树，那么维护的时间为 O(E)，E 为边数。边界条件为 best[V_0]=$-\infty$，Best[V']=$-\infty$|(V_0,V')∈E(T)。

算法总时间复杂度为 O(n^2+k×E)，n 为点数，E 为边数。

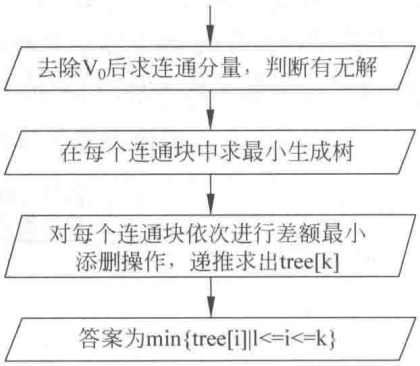


图 5.64

参考代码如下所示。

```
//深度优先搜索
#include <iostream>
using namespace std;
#define Num 9

struct Node
{
   int vertex;
   struct Node* next;
};
typedef struct Node* Graph;
Node Head[Num];                          //9 个顶点
int visited[Num]={0};                    //标记是否访问过

void CreateG(int node1,int node2)        //建立邻接列表
{
   Graph pointer;
   Graph New;
   New=(Graph)malloc(sizeof(struct Node));
   if(New!=NULL)
   {
      New->vertex=node2;
      New->next=NULL;
      pointer=&(Head[node1]);
      while(pointer->next!=NULL)
         pointer=pointer->next;
      pointer->next=New;
   }
}
```

```
30
31  void printG()                                    //打印图
32  {
33     Graph pointer;
34     int i,j;
35     for(i=0;i<Num;i++)
36     {
37        pointer=Head[i].next;
38        printf("Head[%d]",i);
39        while(pointer!=NULL)
40        {
41           printf("-> %d ",pointer->vertex);
42           pointer=pointer->next;
43        }
44        printf("\n");
45     }
46  }
47
48  void DFS(int vertex)                             //深度优先搜索
49  {
50     Graph pointer;
51     visited[vertex]=1;
52     printf("[%d]->",vertex);
53     pointer=Head[vertex].next;
54     while(pointer!=NULL)
55     {
56        if(visited[pointer->vertex]==0)
57           DFS(pointer->vertex);
58        pointer=pointer->next;
59     }
60  }
61
62  int main()
63  {
64     int i;
65     int node[20][2]={{1,2},{2,1},{1,3},{3,1},{2,4},
66                      {4,2},{2,5},{5,2},{3,6},{6,3},
67                      {3,7},{7,3},{4,8},{8,4},{5,8},
68                      {8,5},{6,8},{8,6},{7,8},{8,7}};
69     for(i=0;i<Num;i++)
70     {
71        Head[i].vertex=i;
72        Head[i].next=NULL;
73     }
74     for(i=0;i<20;i++)
75        CreateG(node[i][0],node[i][1]);
76     printG();
77     DFS(1);
78     printf("\n\n\n");
79     system("pause");
80  }
```

前向星表示法

在做图的题目时，经常会遇到图很大，边很少的情况，如果我们用矩阵存图，空间复杂度就会过大，我们可以用链表存数据，或者用前向星方式。

前向星是一种星形表示法，把所有的边集中在一起形成一个边列表，比如说表示第 i 条边(u,v)=w 就应当分别把起点、终点、权值存在三个数组中但是下标相同。星形表示法分为前向星和后向星。前向星按照起始结点从小到大排序，前向星的好处是比链表节省时间和空间，只要不增加或者删除边，就能很快找到从一个点出发的所有边。除了不能直接用起点终点定位以外，前向星几乎是完美的。

例如有一个图结点有 100000 个，普通的矩阵存图存的即是 100000×100000。而前向星是保存边，一共定义三个数组 a[i]、b[i]和 e[i]，它们分别表示第 i 条边的起点编号、第 i 条边的终点编号、第 i 条边的权值。

于是我们就可以这样读数据：

```
cin>>n>>m;
for(int i=1;i<=m;i++)
  cin>>a[i]>>b[i]>>e[i];
```

然后对数组 a 进行排序，即 qsort(a,1,m)，为的是让这些起始编号有序(让终止编号有序也可以，不过一般都是排起始编号)。

这就是前向星了，结构非常简单，空间复杂度视边的个数(也就是 m)而定。

【问题描述】 游戏厅

虚拟现实游戏的火爆导致在魔法学院附近出现了众多的游戏厅。虽然如此，但是由于学院的学员众多(≤3000)，所以抢到游戏机玩的难度非常之大。学院到游戏厅的路是错综复杂的，以至于到一个自己想去的游戏厅都有非常多的路线可以选择，而路线的长度又不相同，这样就决定了要花费的时间，因此想要尽快到达，选择一条最佳的线路是很有必要的。

为了简化问题，我们把学院与周边的游戏厅看作图中的顶点，学校与游戏厅，游戏厅与游戏厅之间的路线看作边，每个边都有一个权，表示我们走完这条路的时间，由于放学人流量大，如果反向走会有危险，因此这是一个有向图。设学院在 S 点，想要去的游戏厅在 T 点。你的任务就是选择一条最佳路线，使得从学校到目的地游戏厅的时间最短，你只需要输出最短到达时间即可。

【输入文件】

输入文件共有 M+2 行数据，第一行两个整数 N,M，表示点数和边数。

然后 M 行每行 3 个正整数(u,v,t)，表示有一条可由 u 到 v 耗时为 t 的边。最后一行两个正整数 S,T。

【输出文件】

输出文件只有一行，一个整数表示最短时间。如果 S,T 之间不存在通路，则输出“No Solution!”(双引号不输出，“!”为西文标点)。

【输入样例】

4 4

1 2 3

2 4 10

1 3 5

3 4 5

1 4

【输出样例】

10

【数据规模】

对于30%的数据保证有1<N≤1000,1≤M≤1000;

对于全部的数据保证有1<N≤10000,1≤M≤100000。

【算法分析】

采用前向星+Bellman-ford即可。

```
1   //游戏厅
2   #include <iostream>
3   using namespace std;
4   int x[10000+1],y[10000+1],e[10000+1];
5   int d[10000+1];
6   int n,m,s,t;
7   char change;
8   void init()
9   {
10      cin>>n>>m;                  //读入点的个数和边的个数
11      int i,j;
12      for(i=1;i<=m;i++)           //采取前向星方式读入
13          cin>>x[i]>>y[i]>>e[i];
14      cin>>s>>t;
15  }
16  void bellman_ford(int s)        //bellman-ford
17  {
18      int i,j,k;
19      for(i=1;i<=n;i++)
20          d[i]=INT_MAX;
21      d[s]=0;
22      for(i=1;i<n;i++)
23          for(j=1;j<=m;j++)
24              if(d[x[j]]!=INT_MAX && d[y[j]]>d[x[j]]+e[j])
25                  d[y[j]]=d[x[j]]+e[j];
26      if(d[t]!=INT_MAX)
27          cout<<d[t]<<endl;
28      else
29          cout<<"No Solution!"<<endl;
30  }
31  int main()
```

```
{
    init();
    bellman_ford(s);
    return 0;
}
```

前向星一般是用来优化 SPFA 的。

前向星的缺点是不能用起点直接终点定位，如果程序必须要定位的话，可以用两种方法来进行定位：

(1) 就是用二分的思想进行查找(也就是常说的对半查找)；

(2) 可以定义一个数组 F[i]表示以 a[i]为起点的第一条边。

二分查找代码如下所示。

```
int binsearch(int i,int j,int x)
{
    int mid;
    while(i<j)
    {
        mid=(i+j)>>1;
        if(a[mid]>x)
            i=mid+1;
        else
            j=mid;
    }
    return i;
}
```

一笔画问题

【问题描述】 一笔画问题

墨老师带领的探险小队圆满地完成了任务，但他们在回去的路上遇到了危险。墨老师在将其他队员送出险境后，他必须独自想办法选择一条安全的路逃离危险。简言之，他不能够走相同的路，而且必须把所有的路全走到。这就是所谓的一笔画问题。例如对于图 5.65 给出的两个图形，是否能够一笔画画完呢？若能一笔画，请找出一笔画的先后顺序；若不能，则打印信息："无解"。

【算法分析】

一笔画的算法为：

(1) 建立邻接矩阵。

(2) 算出每个顶点的度数，并统计奇点的个数。

(3) 若无奇点，则可从任意一点出发开始一笔画。

(4) 若有两个奇点，则可从其中一个奇点出发开始一笔画。

(5) 若奇点的个数超过两个，则不能一笔画。

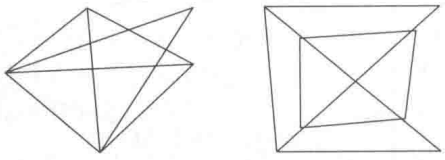


图 5.65

参考程序如下所示。

```
//一笔画问题
//此程序样例与例图无关
# include <stdio.h>
int graph[7][7]={0,0,0,0,0,0,0,
                 0,0,1,0,0,1,1,
                 0,1,0,1,1,0,1,
                 0,0,1,0,1,0,0,
                 0,0,1,1,0,1,1,
                 0,1,0,0,1,0,1,
                 0,1,1,0,1,1,0};

int a[7]={0},total=0,edge=0;

int draw(int v)
{
  int i,k=0;

  if(total==edge) return 1;

  for(i=1;i<7;i++)
    {
      if(graph[v][i]==1)
        {
          k=1;
          graph[v][i]=0;
          graph[i][v]=0;
          edge+=2;
          if(draw(i))
            {
              printf("%3d",i);
              return 1;
            }
          else
            {
              graph[v][i]=1;
              graph[i][v]=1;
              edge-=2;
              k=0;
            }
        }
    }
  if(k==0) return 0;
}

int main()
{
  int i,j,v,k=0;

```

```
49    for(i=1;i<7;i++)
50      {
51        for(j=1;j<7;j++)
52         if(graph[i][j]==1) a[i]++;
53
54        total+=a[i];
55        if(a[i]%2==1)
56         {
57           k++;
58           v=i;
59         }
60      }
61
62    if(k>2)
63      printf("No solution\n");
64    else
65    {
66      draw(v);
67      printf("%3d",v);
68    }
69    return 0;
70  }
```

实际上，能够一笔画完的图为欧拉图或半欧拉图，如果一个图能用一笔多起点开始，不重复通过同一条边而走遍全部边又回到起点，则为欧拉图。若虽然不能回到起点，但能走遍所有边，则为半欧拉图。

如果一个连通图的所有顶点的度均是偶数，则一定是欧拉图，反之亦然。如果一个连通图只有两个度为奇数的顶点，则为半欧拉图。若要一笔画完半欧拉图，则必须以这两个奇点为起始点和终止点。

另外，如果一个图有 N 对奇数顶点，则这个图必须要用 N 笔才能画成(没有一个图的奇点有奇数个)。

与欧拉图相似的是哈密尔顿图，即能够从一个结点出发，对每个结点仅经历一次，又回到起始结点的图称其为哈密尔顿图。这样对图中每个结点仅经历一次所形成的一个回路，称其为哈密尔顿图或哈密尔顿环。哈密尔顿图目前为止还没有找到好的解决方法和有效求回路的算法。

补天计划

【问题描述】 补天计划(fence. cpp/c/pas)USACO3. 3. 1

与天顶星人的战争造成了宇宙时空通道多处的破损，魔法世界的“女娲号”飞船需要巡视以发现和修复破损的时空通道，这称为“补天计划”。

但是飞船从来不经过同一个时空通道两次。你必须编一个程序，读入时空通道的描述，并计算出一条巡视时空通道的路径，使每个时空通道都恰好被经过一次。飞船能从任何一个顶点(即两个结点的交点)开始出发，在任意一个顶点结束。

每一个时空通道连接两个顶点，顶点用 1～500 标号（虽然有的时空并没有 500 个顶点）。一个顶点上可连接任意多（≥1）个时空通道。所有时空通道都是连通的（也就是你可以从任意一个顶点到达另外的所有时空通道）。

你的程序必须输出飞船的路径（用路径上依次经过的顶点号码表示）。我们如果把输出的路径看成一个 500 进制的数，那么当存在多组解的情况下，输出 500 进制表示法中最小的一个（也就是输出第一个数较小的，如果还有多组解，输出第二个数较小的……）。

输入数据保证至少有一个解。

【输入格式】

第 1 行：一个整数 F（1 ≤ F≤1024），表示时空通道的数目。

第 2 行：每行两个整数 i，j（1≤i，j≤500）表示这条时空通道连接 i 与 j 号顶点。

【输出格式】

输出应当有 F+1 行，每行一个整数，依次表示路径经过的顶点号。注意数据可能有多组解，但是只有上面题目要求的那一组解是正确的。

【输入样例】

```
9
1 2
2 3
3 4
4 2
4 5
2 5
5 6
5 7
4 6
```

【输出样例】

```
1
2
3
4
2
5
4
6
5
7
```

【算法分析】

这道题是要求我们求出一条欧拉路，所以我们要首先判断图中是否有欧拉路。对于一个无向图，如果它每个点的度都是偶数，那么它存在一条欧拉回路；如果有且仅有 2 个点的度为奇数，那么它存在一条欧拉路；如果超过 2 个点的度为奇数，那么它就不存在欧拉

路了。

由于题目中说数据保证至少有1个解，因此一定存在欧拉路了。但是我们还要选一个点作为起点。如果没有点的度为奇数，那么任何一个点都能做起点。如果有2个奇点，那么就只能以这两个点之一为起点，另一个为终点。但是我们要注意，题目要求我们输出的是进行进制转换之后最小的(也就是输出第一个数较小的，如果还有多组解，输出第二个数较小的……)，所以我们要以最小的点做起点。

参考程序如下所示。

```
//补天计划
#include<cstdlib>
#include<cstring>
#include<iostream>
#define N 510
using namespace std;

int du[N],ans[N*3];
int map[N][N];
int m,ans_tot,st;

void Init()
{
  scanf("%d",&m);
  int x,y;
  for (int i=1;i<=m;i++)
  {
    scanf("%d%d",&x,&y);
    map[x][y]++;
    map[y][x]++;
    du[x]++;
    du[y]++;
  }
  st=0;
  for (int i=1;i<=500;i++)
    if (du[i]%2==1)
    {
      st=i;
      break;
    }
  if (st==0)
  {
    for (int i=1;i<=500;i++)
      if (du[i])
      {
        st=i;
        break;
      }
  }
}
```

```
41
42  void Dfs(int x)
43  {
44    for (int i=1;i<=500;i++)
45      if (map[x][i])
46      {
47        map[x][i]--;
48        map[i][x]--;
49        du[i]--;
50        du[x]--;
51        Dfs(i);
52      }
53    ans[++ans_tot]=x;
54  }
55
56  void Do()
57  {
58    Dfs(st);
59    for (int i=ans_tot;i>=1;i--)
60      printf("%d\n",ans[i]);
61  }
62
63  int main()
64  {
65    freopen("fence.in","r",stdin);
66    freopen("fence.out","w",stdout);
67    Init();
68    Do();
69    fclose(stdin); fclose(stdout);
70    return 0;
71  }
```

拓扑排序

一项工程由若干个子工程组成，这些子工程之间存在一种先后关系，即某些子工程必须在其他一些工程完成之后才能开始，我们用一个有向图来表示工程间的关系，子工程为顶点，活动之间的先后关系为有向边，这种有向图称为顶点表示活动的网络，又称为 AOV 网。在 AOV 网中，如果有一条从顶点 V_i 到 V_j 的路径，则说 V_i 是 V_j 的前趋，V_j 是 V_i 的后续。如果有弧(V_i,V_j)，则称 V_i 是 V_j 的直接前趋，V_j 是 V_i 的直接后续。

拓扑排序是把 AOV 网中所有顶点排成一个线性序列，如果有弧(V_i,V_j)，则序列中顶点 V_i 在 V_j 之前。

现以实例来说明：

有 N 个魔法学员($1\leqslant N\leqslant 26$)，编号依次为 A，B，C，D，…，队列训练时，指挥官要把一些学员从高到矮依次排成一行。但现在指挥官不能直接获得每个人的身高信息，只能获得“某某比某某高”这样的比较结果，例如 A>B、B>D、F>D 等，学员的身高关系对应一张有向图，图中的顶点对应一个学员，学员的身高关系图可如图 5.66 所示。

我们按照高矮将学员排出一个线性的顺序关系，即为对有向图的顶点进行拓扑排序。注意：有环图是不能拓扑排序的，如图 5.67 所示。

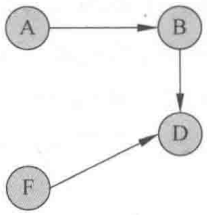


图 5.66

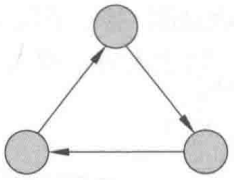


图 5.67

拓扑序列的计算方法：

(1) 从图中选择一个入度为 0 的顶点且输出之。

(2) 从图中删除该顶点及其所有出边(即与之相邻的所有顶点的入度减 1)。

(3) 反复执行上面两个步骤，真到整个拓扑排序完成。若再没有入度为 0 的顶点，说明有环无解。

例如有图 5.68 所示的学员身高关系图。

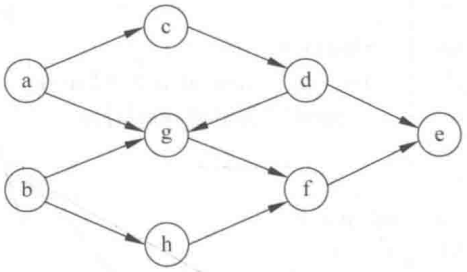


图 5.68

开始时，有 a，b 入度均为 0，则任选一点例如 b(由此可看出，拓扑排序可能并非唯一)，将 b 输出，并删除 b 顶点及其出边，如图 5.69 所示。

此时有 a，h 入度均为 0，任选一点例如 h，将 h 输出，并删除 h 顶点及其出边，如图 5.70 所示。

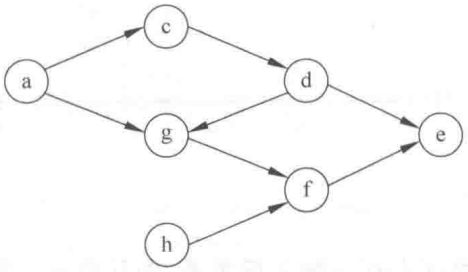


图 5.69

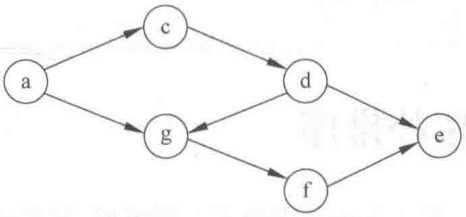


图 5.70

以此方法直到所有顶点均被删除，则原图按顶点输出顺序形成的一种拓扑排序为 bhacdgfe(可能有多种拓扑排序)。

参考程序如下所示。

```
//拓扑排序
#include<iostream>
#define MAXN 1001
using namespace std;

int n,m;                    /*n 为结点数,m 为边数*/
int top;                    /*栈顶指针*/
```

```
8   int mark[MAXN],ind[MAXN],stack[MAXN];
9   /*mark 为结点的拓扑序,ind 为结点的入度数,stack 为一个栈*/
10  bool w[MAXN][MAXN];                    /*邻接表*/
11
12  void sort()
13  {
14    int i,u,t;
15    top = -1;
16    /*将入度为 0 的结点入栈并且做出标记*/
17    for(i = 1 ; i <= m ; i ++)
18      if(ind[i] == 0 )
19      {
20        mark[i] = 1;
21        stack[++top] = i;
22        ind[i] = -1;
23      }
24
25    /*用 DFS 的方式标记结点的拓扑序*/
26    do{
27      /*取出栈中的一个结点并将与该结点相邻的结点的入度减 1*/
28      u = stack[top--];
29      for(i=1;i<=n;i++)
30        if(w[u][i] and ind[i] not_eq -1)
31      ind[i]--;
32
33      /*查找当前是否有入度为 0 且没有入栈的结点并将其入栈*/
34      for(i=1;i<=n;i++)
35        if(ind[i] == 0)
36         {
37          mark[i] = mark[u] + 1;
38          stack[++top] = i;
39          ind[i] = -1;
40         }
41    }while(top >= 0);
42
43    for(i=1;i<=n;i++)
44      cout<<mark[i]<<" ";
45    cout<<"\n";
46    return ;
47  }
48
49  int main()
50  {
51    int i,u,v;
52    cin>>n>>m;                    //输入一个符合拓扑序的图的结点数和边数 n,m
53
54    for(i=1;i<=m;i++)
55    {
56      cin>>u>>v;
57      w[u][v] = true;
```

```
        ind[v]++;
    }
    sort();
    return 0;
}
```

再介绍 dfs 方式的拓扑排序，对图 dfs 遍历时顶点的出栈顺序的反序即拓扑排序的结果。

其伪代码为：

```
void Topological_Sort(G);
{
 1.调用 DFS(G)计算每个顶点的完成时间 f[v];
 2.当每个顶点完成后，把它插入链表前端;
 3.返回由顶点组成的链表;
}
```

例如，对图 5.71 所示的有向无环图利用深度优先搜索进行拓扑排序，退出 DFS 函数的顺序为 e f g d c a h b，则此图的一个拓扑序为 b h a c d g f e。

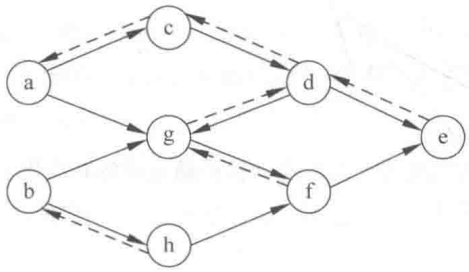


图 5.71

参考程序如下所示。

```
//拓扑排序算法 2
#include<iostream>
using namespace std;
const int MAX = 100;
int List[MAX],color[MAX],k;
struct
{
    int n;
    int adjvex[MAX];
}Edge[MAX];

void DFS(int cur)
{
    int i,next;
    color[cur] = -1;
    for(i=0;i<Edge[cur].n;i++)              //遍历所有的连接边
    {
```

```
18          next = Edge[cur].adjvex[i];                //找到下一个连接边
19          if(color[next]==0)
20             DFS(next);
21          else if(color[next]==-1)                   //有回边出现意味着有回路
22          {
23             cout<<"Circuit exists!"<<endl;
24             system("pause");
25             exit(0);
26          }
27       }
28       color[cur] = 1;
29       List[--k] = cur;                              //按结点完成的顺序入列,逆序保存
30    }
31
32    int main()
33    {
34       int n,e,a,b;
35       int i,j;
36       cin>>n>>e;                                    //输入顶点数 n 和边数 e
37       for(i=1;i<=n;i++)
38          Edge[i].n = 0;
39       for(i=0;i<e;i++)                              //输入边
40       {
41          cin>>a>>b;
42          Edge[a].adjvex[ Edge[a].n++ ] = b;
43       }
44       k=n;
45       DFS(1);
46       for(i=0;i<n;i++)
47          cout<<List[i]<<" ";
48       system("pause");
49       return 0;
50    }
```

关键路径

拓扑排序应用的一个典型的例子就是关键路径(Critical Path)的计算。所谓关键路径可以这么理解,比如有一群人约好去某个地方,大家从同一个地方同一时间开始出发,有些人选择骑车,有些人选择走路,有些选择公交……,那么最迟到达的那个人需要的时间就相当于关键路径。

又如某一项工程必须要完成 N 个子工程,每个子工程需要不同的时间,其中一些子工程必须以另一些子工程完成为基础,这种关系可以被表示成一个有向图即 AOE(Activity On Edge Network)。所谓 AOE 是指用顶点表示事件,弧表示活动,弧上的权值表示活动持续的时间的有向图,如图 5.72 所示的 AOE 网包括 11 项活动、9 个事件,每个事件都有所需的完成时间。

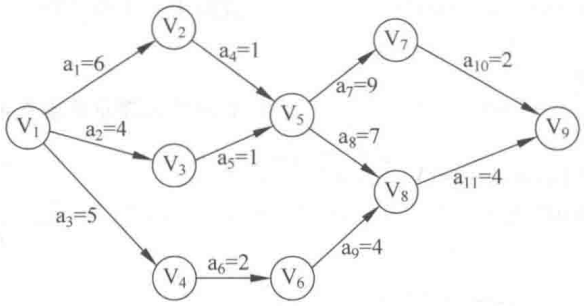


图 5.72

AOE 的性质如下：

(1) 只有在某顶点所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始。

(2) 只有在进入某顶点的各有向边所代表的活动都已结束，该顶点所代表的事件才能发生。

(3) 由于实际工程只有一个开始点和一个结束点，因此 AOE 存在唯一的入度为 0 的开始点(又称源点)和唯一的出度为 0 的结束点(又称汇点)。

(4) AOE 网应当是无环的。

我们现在需要解决的问题：

(1) 完成整项工程至少需要多少时间(最短时间)；

(2) 哪些活动是影响工程进度的关键(关键活动)。

用 e(i)表示活动最早开始时间，l(i)表示在不拖延整个工期的条件下，该活动的允许的最迟开始时间，则 l(i)－e(i)为完成该活动的时间余量。则根据图 5.72 所示的样例可列表如表 5.5 所示(计算过程见后面的分析)。

表 5.5

活动	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
e(i)	0	0	0	6	4	5	7	7	7	16	14
l(i)	0	2	3	6	6	8	7	7	10	16	14
l(i)－e(i)	0	2	3	0	2	3	0	0	3	0	0

若 l[k] = e[k](0<k<12)，则 ak 活动就是关键活动，否则为非关键活动，显然，关键活动的延期，会使整个工程延期，但非关键活动只要它的延期量不超过它的最大可利用时间，就不会影响整个工期。由此可得关键路径如图 5.73 所示。

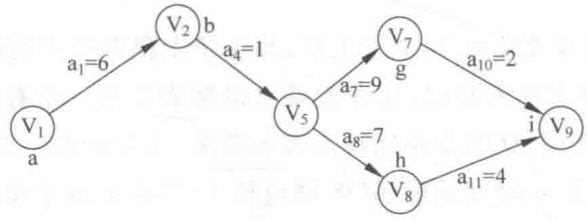


图 5.73

可以看出，一个 AOE 的关键路径可以不止一条，如图 5.73 的 AOE 中有两条关键路径，$(v_1, v_2, v_5, v_7, v_9)$和$(v_1, v_2, v_5, v_8, v_9)$，它们的路径长度都是 18。

现在的问题是，如何求得 l[k]和 e[k]呢？我们先进行如下定义：

(1) 设 AOE 的起点为 v_0，终点为 v_n。

(2) 设 AOE 中，从事件 i 到 j 的路径中，活动时间最大者称为 i 到 j 的关键路径(Critical Path)，记为 cp(i,j)。特别地，始点 0 到终点 n 的关键路径 cp(0,n)是整个 AOE 的关键路径。

(3) 设事件 vi 的最早发生时间 ve(i)定义为：从始点到 vi 的最大活动时间长度，即 cp(0,i)。

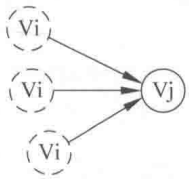
(4) 设事件 vi 的最晚发生时间 vl(i)定义为：在不拖延整个工期的条件下，vi 的可能的最晚发生时间。即 vl(i) = ve(n)－cp(i,n)。

(5) 设弧<vi,vj>为活动 ak，则 ak 的最早开始时间 e(k)等于事件 vi 的最早发生时间，即 e(k) = ve(i) = cp(0,i)。

(6) 设弧<vi,vj>为活动 ak，则 ak 的最晚开始时间 l(k)定义为：在不拖延整个工期的条件下，该活动的允许的最迟开始时间，即 l(k) = vl(j)－len(i,j)，其中 len(i,j)是 ak 的权。

由定义(5)中的 e(k) = ve(i) = cp(0,i)和定义(6)中的 l(k) = vl(j)－len(i,j)可知，要求 e(k)和 l(k)的值，须先求出 ve[i]和 vl[i]的值：

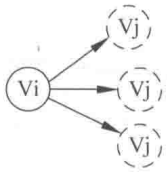
(1) 求事件 vj 的最早发生时间 ve[j]需按拓扑次序递推，即计算 ve(j)前，应已求得 j 的各前趋结点的 ve 值。如图 5.74 所示：



ve(1)=0
ve(j)=MAX{ve(i)+len(i，j)}
其中len(i，j)为所有到达Vj的有向边集合

图 5.74

(2) 求事件 vj 的最晚发生时间 ve[j]需按逆拓扑次序递推(注意此处是逆拓扑排序)，即计算 vl(i)前，应已求得 i 的各后继结点的 vl 值。如图 5.75 所示。

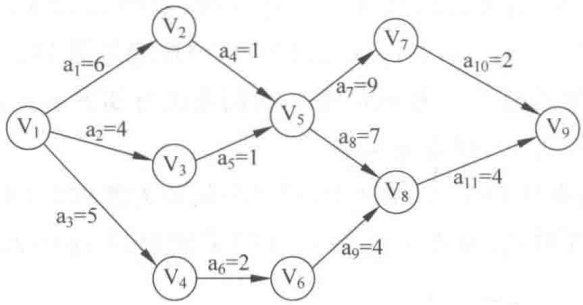


vl(n)=ve(n)
vl(i)=MIN{vl(j)-len(i，j)}
其中len(i，j)为所有从Vi发出的
有向边的集合

图 5.75

则根据图 5.72 的样例，推出 ve(k)和 vl(k)的过程如图 5.76 所示。

根据 e(k) = ve(i)和 l(k) = vl(j)－len(i,j)，推出 e(k)和 l(k)如图 5.77 所示。



根据递推式

ve(j)=MAX{ve(i)+len(i, j)}
vl(i)=MIN{vl(j)-len(i, j)}

推出下表

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
ve(k)	0	6	4	5	7	7	16	14	18
vl(k)	0	6	6	8	7	10	16	14	18

图 5.76

e(k)=ve(i)
l(k)=vl(j)-len(i, j)

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
e(i)	0	0	0	6	4	5	7	7	7	16	14
i(i)	0	2	3	6	6	8	7	7	10	16	14

图 5.77

参考代码如下所示。

```
//关键路径
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <algorithm>
using namespace std;

const int Ns=1e5;
const int Ms=1e7;

//邻接链表
int p[Ns],cnt;
struct edge{int v,w,next;}e[Ms];
void adde(int u,int v,int w)
{
    e[++cnt]=(edge){v,w,p[u]};
    p[u]=cnt;
}

struct queue                    //双端队列
```

```
21  {
22    int head,tail,a[Ns];
23    queue(){tail=1;}
24    void push(int k){a[++tail]=k;}
25    void pop(){++head;}
26    void pop_back(){--tail;}
27    int front(){return a[head];}
28    int back(){return a[tail];}
29    bool empty(){return head>tail;}
30  }q;
31
32  //im[i]表示入度,ve[i]表示最早,vl[i]表示最迟的开始时间
33  int N,M,im[Ns],ve[Ns],vl[Ns];
34
35  void work()
36  {
37    int i,u,v,w;
38    scanf("%d%d",&N,&M);      //输入事件数,活动数
39    for (i=0;i<M;++i)
40    {
41      scanf("%d%d%d",&u,&v,&w);      //从 u 到 v 顶点的权值为 w
42      adde(u,v,w);
43      ++im[v];
44    }
45    //拓扑排序正向更新 ve
46    for (i=1;i<=N;++i)
47      if (!im[i]) q.push(i);
48
49    while(!q.empty())
50    {
51      u=q.front();q.pop();
52      for (i=p[u];i;i=e[i].next)
53      {
54        ve[e[i].v]=max(ve[e[i].v],ve[u]+e[i].w);
55        if (!(--im[e[i].v]))
56          q.push(e[i].v);
57      }
58    }
59    //反向更新 vl
60    q.head=1;
61    while(!q.empty())
62    {
63      u=q.back();q.pop_back();
64      for (i=p[u];i;i=e[i].next)
65      {
66        if (!vl[e[i].v])
67          vl[e[i].v]=ve[e[i].v];
68        if (!vl[u])
69          vl[u]=vl[e[i].v]-e[i].w;
70        else
```

```
                vl[u]=min(vl[u],vl[e[i].v]-e[i].w);
        }
    }

    for (i=1;i<=N;++i)          //输出
        printf("POINT_%d ,ve[%d]=%d ,vl[%d]=%d\n",i,i,ve[i],i,vl[i]);
    /*
        if ve[i]==vl[i] ,则 i 点为一个关键点,属于一个关键路径。
        标记完所有点后,若要输出关键路径,DFS 即可
    */
}

int main()
{
    work();
    return 0;
}
```

【问题描述】 工程计划(HDU 4109)

随着虫洞的发现,浩瀚无垠的宇宙越来越吸引了人们的目光,终于在魔法历 2165 年,魔法世界开始实施一项史上最宏伟的计划——"星际争霸计划",这项计划由无数的子计划组成,这些子计划之间存在一种先后关系,即某些子计划必须在其他一些计划完成一段时间之后才能开始。除此之外,各子计划可以同步进行且各子计划完成的时间为 1 个月。现在给出各子计划之间的关系图,请分析完成所有的计划需要多少时间?

【输入格式】

输入包含多组测试数据。

第一行包含两个整数 N,M (N ≤ 1000,M ≤10000),表示有 N 个项目和 M 个依存关系。

随后 M 行,每行包含三个整数 X,Y,Z,表示必须完成 X 子计划才能执行 Y 子计划,且花费时间为 Z。各子计划的标号为 0 ~N-1。

【输出格式】

输出完成所有计划需要的时间。

【输入样例】

5 2

1 2 1

3 4 1

【输出样例】

2

【样例说明】

第 1 个月,子计划 0、1 和 3 被完成;

第 2 个月,子计划 2 和 4 被完成;

所以最短完成时间为 2 个月。

【算法分析】

标准的求关键路径题目。

参考程序如下所示。

```
//工程计划
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <algorithm>
#include <queue>
using namespace std;

const int Ns=20005;
const int Ms=20005;

int N,M,im[Ns],vmin[Ns];

int p[Ns],cnt;

struct edge
{
  int v,w,next;
}e[Ms];

void adde(int u,int v,int w)
{
  e[++cnt]=(edge){v,w,p[u]};
  p[u]=cnt;
}

int Min(int a,int b)
{
  if (!a)
    return b;
  else
  {
    if (a<b)
      return a;
    else
      return b;
  }
}

void work()
{
  int i,u,v,w;
  while(scanf("%d%d",&N,&M)!=EOF)
  {
    memset(im,0,sizeof(im));
    memset(p,0,sizeof(p));cnt=0;
```

```
47        memset(vmin,0,sizeof(vmin));
48        memset(e,0,sizeof(e));
49
50        for (i=0;i<M;++i)
51        {
52          scanf("%d%d%d",&u,&v,&w);
53          adde(u+1,v+1,w);
54          ++im[v+1];
55        }
56
57        queue<int> q;
58        for (i=1;i<=N;++i)
59          if (!im[i])
60            q.push(i);
61        while(!q.empty())
62        {
63          u=q.front();q.pop();
64          for (i=p[u];i;i=e[i].next)
65          {
66            if (!vmin[e[i].v])
67              vmin[e[i].v]=e[i].w+vmin[u];
68            else
69              vmin[e[i].v]=max(vmin[e[i].v],e[i].w+vmin[u]);
70          --im[e[i].v];
71            if (!im[e[i].v])
72              q.push(e[i].v);
73          }
74        }
75        int ans=0;
76        for (i=1;i<=N;++i)
77          ans=max(ans,vmin[i]);
78        printf("%d\n",ans+1);
79      }
80    }
81
82    int main()
83    {
84      work();
85      return 0;
86    }
```

顺便一提的是，本题的另一种解法是使用差分约束系统。

第六章

哈 希 表

哈希表

哈希表查找又叫杂凑法或散列法。它通过在结点的存储位置和其关键字之间建立某种直接关系，使得查找时，无须做比较或做很少次的比较即能找到相应的记录。

再具体点，设一个长度为 m 的表 A，我们用一个哈希函数 H 把数据集合中的数据结点映射到表 A 中，如果要查找元素 i，我们可以通过哈希函数 H(i)找到 i 在表中的存储位置。例如之前我们使用康托展开式来作为对 n 个数的排列进行状态的压缩和存储的哈希函数，这种映射是一对一的关系，不会产生冲突。

但在更广泛的范围内，我们很难设计出一个均匀映射的哈希函数来保证对于任意不同的 K_i 和 K_j，有 $H(K_i)!=H(K_j)$。当 $K_i!=K_j$，而 $H(K_j)=H(K_i)$的现象称为冲突现象。由于冲突是难免的，所以当发生冲突时，必须要有相应的解决冲突的方法，因此构造合适的哈希函数和建立解决冲突的方法是哈希表查找的两个任务。

例如：对于数据序列(22,41,53,46,30,13,1)，我们构建哈希函数为 $H(K_i)=3K_i\%11$，当发生冲突时，其求下一个地址的函数为：

$d_1=H(K_i)$

$d_i=(d_{i-1}+(7K_i\%10)+1)\%11\ (i=2,3,\cdots)$

哈希表空间为(0～10)的散列地址空间。

则哈希表构造过程如下：

(1) $H(22)=3\times22\%11=0$

(2) $H(41)=3\times41\%11=2$

(3) $H(53)=3\times53\%11=5$

(4) $H(46)=3\times46\%11=6$

(5) $H(30)=3\times30\%11=2$ (产生冲突)

$d_1=H(30)=2$

$H(30)=(2+(7\times30\%10)+1)\%11)=3$

(6) H(13)=3×13%11=6 (产生冲突)

d_1=H(13)=6

H(13)=(6+(7×13%10)+1)%11=8

(7) H(1)=3×1%11=3 (产生冲突)

d_1=H(1)=3

H(1)=(3+(1×7%10)+1)%11=0(产生冲突)

d_2=H(1)=0

H(1)=0+((1×7%10)+1)%11=8(产生冲突)

d_3=H(1)=8

H(1)=(8+(1×7%10)+1)%11=5 (产生冲突)

D_4=H(1)=5

H(1)=(5+(1×7%10)+1)%11=2(产生冲突)

D_5=H(1)=2

H(1)=(2+(1×7%10)+1)%11=10

参考程序如下所示。

```
1  //哈希表
2  #include <iostream>
3  using namespace std;
4  #define M   11
5  #define N   7
6
7  struct hterm
8  {
9     int key;                          /*关键字值*/
10    int si;                           /*散列次数*/
11 };
12
13 struct hterm hashlist[M+1];
14 int i,address,sum,d,x[N+1];
15 float average;
16
17 int main()
18 {
19    for (i=1;i<=M;i++)                //置初值
20    {
21       hashlist[i].key=0;
22       hashlist[i].si=0;
23    }
24    x[1]=22;  x[2]=41;  x[3]=53;
25    x[4]=46;  x[5]=30;  x[6]=13;
26    x[7]=1;
27    for (i=1;i<=N;i++)
28    {
29       sum=0;
30       address=(3*x[i]) % M;
```

```
31        d=address;
32        if (hashlist[address].key==0)        //若没有冲突则保存
33        {
34          hashlist[address].key=x[i];
35          hashlist[address].si=1;
36        }
37        else                                 //处理冲突
38        {
39          do
40          {
41              d=(d+(x[i]*7) % 10 +1) % 11;
42            sum=sum+1;
43          }while (hashlist[d].key!=0);
44          hashlist[d].key=x[i];
45          hashlist[d].si=sum+1;
46        }
47      }
48      printf("哈希表地址:   ");
49      for (i=0;i<M;i++)
50        printf("%-4d",i);
51      printf("\n");
52      printf("哈希表关键字: ");
53      for (i=0;i<M;i++)
54        printf("%-4d",hashlist[i].key);
55      printf("\n");
56      printf("搜索长度:     ");
57      for (i=0;i<M;i++)
58        printf("%-4d",hashlist[i].si);
59      printf("\n");
60      average=0;
61      for (i=0;i<M;i++)
62        average=average+hashlist[i].si;
63      average=average/N;
64      printf("平均搜索长度:ASL(%d)=%g",N,average);
65      system("pause");
66      return 0;
67    }
```

下面介绍几种哈希函数的构建方法：

数字分析法：数字分析法是从数据结点中提取数字分布比较均匀的若干位作为哈希地址。如对于 $K_1 \sim K_4$ 的数据序列(1100001134,1100001214,1000001316,1000001523)，可以取第 7 位和第 8 位作为哈希地址。即 $H(K_1)=11$，$H(K_2)=12$，$H(K_3)=13$，$H(K_4)=15$。

除留余数法：是用关键字 K_i 除以一个合适的不大于哈希表长度的正整数 P，所得余数作为哈希地址的方法。对应的哈希函数 $H(K_i)$ 为 $H(K_i)=K_i\%P$。由此方法产生的哈希函数的好坏由 P 决定，当 P 取小于哈希表长的最大质数时，产生的哈希函数较好。该方法是一种简单而有效的构造哈希函数的方法。经验上我们一般选择在元素值域的 1～2 倍的范围内的素数，例如当值域在[0,600]，那么 701 即是一个不错的选择。

乘积取整法：我们用关键字 K 乘以一个在(0～1)中的实数种子 A(最好是无理数)，得

到一个(0,k)之间的实数；取出其小数部分，乘以表容量 M，再取整数部分，即得到在哈希表中的位置。例如表容量 M=12，种子 A=(sqrt(5)−1)/2 时，关键值 K=100 的哈希表位置即 H(100)=9。这类函数主要适用于小数。

我们再介绍一种简单易实现的哈希表并通过模拟它的插入与查找方法以进一步理解哈希表，假设有下面一些数要插入哈希表：13,7,15,21,58,89,25,87,92,33,74,65,97,14,18,37 共 16 个数。

首先定义一个有 19 个元素的数组，令哈希函数为 x%19，则哈希表的形成过程如下：

插入 13,7,15,21,58 的过程非常顺利，但插入 89 时，由于 89%19=13，而 13 的位置先前已经插入 13 这个数了，这就造成了冲突，则将 89 插入 13 后的空位上，如表 6.1 所示。

表 6.1

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
		58	21					7						13	89	15				

以后各数的插入过程类似，特别是插入 37 时，在表尾冲突而插入了表头，如表 6.2 所示。

表 6.2

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	37	58	21	97	33		25	7	65			87		13	89	15	92	17	14	18

我们再来模拟哈希表的查找过程，假设待查数列为 7,24,89,14 和 34，则查找过程如表 6.3 所示。

表 6.3

待查数	哈希值	查找过程	停止位置	查找结论
7	7	a[7]=7	a[7]=7	成功
24	5	a[5]=0	a[5]=0	失败
89	13	a[13]>0 and a[13]≠89	a[14]=89	成功
14	14	a[14..17]≠0 and a[14..17]≠14	a[18]=14	成功
93	17	a[17..19,0..4]≠0 and ≠93	a[5]=0	失败

我们可得出以下结论：

(1) 我们应定义合适长度的哈希表，表越长，冲突的可能性越小，查找速度越快，但浪费的空间越多，表越小，则冲突的次数就会升高，查找时间就会延长，最坏的是哈希表充满了数据，则一个失败的查询可能永远无法完成。

(2) 应尽可能设计好哈希函数使数据分布比较分散，例如，哈希表的最大下标选择质数等。

单词拼写检查

【题目描述】 单词拼写检查

张琪曼小时候曾经拥有过一个宠物机器人，它可以检查主人输入的单词是否有拼写错误。这是因为宠物机器人内置有一单词库，里面存有常用的单词。待主人输入一个单词后，它就到库中查找是否有这种拼写，以判断正误。

现给定一个单词库，库中存入 10000 个以下的单词(单词长度 3～20)，且单词没有按任何顺序排列，另给定 10000 个以下的待查单词，要求编程输出这些待查单词中有多少个词拼写错误。题中单词均由小写字母组成。

【输入格式】

第一行为一个数字 N，表示单词库里有 N 个单词，随后是 N 行单词。紧接着一个数 M，表示待查单词数，随后是 M 行待查单词。

【输出格式】

输出拼写错误的单词数。

【输入样例】

```
5
apple
be
love
up
down
3
up
down
bee
```

【输出样例】

```
1
```

【算法分析】

此题需建立哈希表才可能在限定的时间里得出解。但建立哈希表，仍需要一些技巧，例如，我们可以取单词的首、中、尾的字符顺序码加权。

参考程序如下所示。

```
//单词拼写检查
#include <iostream>
#include <string>
using namespace std;
const int maxn=1123357;          //此处的质数过大，请选择合适质数
int total;
string h[maxn+1];
```

```
8
9   int hash(string x)
10  {
11     int t,l,m;
12     l=x.length();
13     m=l/2;
14     t=(x[0]-65)*10000+(x[m]-65)*100+(x[l-1]-65);
15     return t%maxn;
16  }
17
18  void insert(string x)
19  {
20     int t=hash(x);
21     while(h[t]!="" && h[t]!=x)
22     {
23        t++;
24        if(t==maxn)
25           t=0;
26     }
27     h[t]=x;
28  }
29
30  void find(string x)
31  {
32     int t=hash(x);
33     while(h[t]!="" && h[t]!=x)
34     {
35        t++;
36        if(t==maxn)
37           t=0;
38     }
39     if(h[t]=="")
40        total++;
41  }
42
43  int main()
44  {
45     int i,n,m;
46     string wrd;
47     freopen("in.txt","r",stdin);
48     freopen("ou.txt","w",stdout);
49
50     cin>>n;                    //单词库的单词个数
51     for(i=1;i<=n;i++)
52     {
53        cin>>wrd;
54        insert(wrd);
55     }
```

```
56    cin>>m;                    //待查单词数
57    for(i=1;i<=m;i++)
58    {
59      cin>>wrd;
60      find(wrd);
61    }
62    cout<<total;
63    system("pause");
64    return 0;
65  }
```

相同的雪花

【题目描述】 相同的雪花（POJ 3349）

墨老师为了拯救其他探险队员而落入了名为“无限轮回”的魔法陷阱，这种上古魔法营造了一个个逼真的虚幻世界，误入者若无强大的意志力和自省能力，就会逐渐迷失其中而永远无法回归真实的世界……

多年后，曾有人问墨老师是怎么发现他在陷阱中所处的世界其实是虚幻的并且最终成功逃离时，墨老师笑着说：“因为在真实的世界中，没有两片相同的雪花。”现在你的任务是编写一个程序比较你所处的世界是否有相同的雪花，已知每朵雪花有六瓣，如果两朵雪花彼此对应的花瓣有相同的长度则说明是一样的。

【输入格式】

第一行包括一个整数 $n(0 < n \leqslant 100000)$ 表示雪花的数目。随后 n 行描述每一朵雪花。每朵雪花有六个整数，每个整数范围在(0～10000000)之间，表示雪花六个花瓣的长度。六个整数的先后出现顺序可能是雪花花瓣的顺时针顺序也可能是逆时针顺序，并且可能是从任一个花瓣位置开始的。比如说，对同一个雪花，描述的方法可能是 1 2 3 4 5 6 或者 4 3 2 1 6 5。

【输出格式】

如果没有相同的雪花，输出“No two snowflakes are alike.”，否则输出“Twin snowflakes found.”。

【输入样例】

2

1 2 3 4 5 6

4 3 2 1 6 5

【输出样例】

Twin snowflakes found.

【算法分析】

如果采用最简单的枚举，其时间复杂度为 $O(n \times n)$。若采用哈希算法，即每读入一片雪花，就将该雪花进行哈希操作，并判断哈希表里是否有相同的哈希值，若有相同的哈希值就从链表中一一取出并判断是否同构即可。

参考代码如下。

```
//相同的雪花
#include <iostream>
#include <stdlib.h>
#include <vector>
using namespace std;

const int M = 90001;                    //哈希函数,取余的数

int snow[100005][6];                    //存储雪花信息
vector<int> hash[M];                    //哈希表,表中存储的是 snow 数组的下标

bool isSame(int a, int b)               //判断 a 与 b 是否同样
{
  for(int i=0;i<6;i++)
  {
                                        //顺时针
    if((snow[a][0] == snow[b][i] &&
          snow[a][1] == snow[b][(i+1)%6] &&
          snow[a][2] == snow[b][(i+2)%6] &&
          snow[a][3] == snow[b][(i+3)%6] &&
          snow[a][4] == snow[b][(i+4)%6] &&
          snow[a][5] == snow[b][(i+5)%6])
          ||                            //逆时针
          (snow[a][0] == snow[b][i] &&
         snow[a][1] == snow[b][(i+5)%6] &&
         snow[a][2] == snow[b][(i+4)%6] &&
         snow[a][3] == snow[b][(i+3)%6] &&
         snow[a][4] == snow[b][(i+2)%6] &&
         snow[a][5] == snow[b][(i+1)%6]))

      return true;
  }
  return false;
}

int main()
{
  int n;
  int i,j;
  cin>>n;
  for( i = 0; i < n; i++)
    for( j = 0; j < 6; j++)
      cin>>snow[i][j];

  int sum, key;
  for(i = 0; i < n; i++)
  {
    sum = 0;                            //求出雪花六个花瓣的和
```

```
49        for( j = 0; j < 6; j++)
50          sum += snow[i][j];
51        key = sum % M;                    //求出 key
52
53        //判断是否与 hash 表中 hash[key]存储的雪花相同
54        for(vector<int>::size_type j = 0; j < hash[key].size(); j++)
55         {
56           if(isSame(hash[key][j], i))  //如相同
57            {
58              cout<<"Twin snowflakes found."<<endl;
59              exit(0);
60            }
61         }
52         hash[key].push_back(i);          //若没找到相同的
63       }
64       cout<<"No two snowflakes are alike."<<endl;
65       return 0;
66     }
```

零和游戏

【题目描述】 零和游戏(POJ 2785)

被探险小分队带出上古文明遗址的资料中最有价值的部分，并不是种种令人心驰神往的高科技资料，而是取回了失落已久的古文明精魂。简言之，魔法世界几千年来一直在错误地玩着零和游戏，所谓零和游戏，是指一项游戏中，游戏者有输有赢，一方所赢正是另一方所输，而游戏的总成绩永远为零。这从魔法世界的方方面面都可以验证：例如大肆开发资源，留给后人的便越来越少；人类活动的范围越广，生物物种消失的速度就越来越快；工业的发展越快，环境污染就越来越严重……，而找到了失落已久的古文明精魂，就可以逐步纠正这一错误，使魔法世界的发展走向正轨。

现在的问题是，有一个魔法师错误地理解了零和游戏的含意，并提出了一个很无厘头的问题，即有一个 N 行 4 列的数字矩阵中，从四列中分别选取一个数使得总和为 0，问一共有多少种取法？

【输入格式】

第一行为一个数字 N，表示 N 行，N 不超过 4000，N 行 4 列的矩阵中每个数的绝对值不超过 2^{28}。

【输出格式】

输出共有多少种和为零的取法。

【输入样例】

6

−45 22 42 −16

−41 −27 56 30

−36 53 −37 77

−36 30 −75 −46

26 −38 −10 62

−32 −54 −6 45

【输出样例】

5

【算法分析】

注意如果第一列有两个 0，用第一个 0 和第二个 0 是两种不同的取法。

哈希表的参考程序如下。

```
//零和游戏-哈希表
#include <iostream>
#define MAX 1000000000
#define size 20345677
#define key 745
using namespace std;

int n,a[4040],b[4040],c[4040],d[4040],ans;
int hash[size],sum[size];

void Insert(int num)
{
  int tmp=num;
  num=(num+MAX)%size;
  while(hash[num]!=MAX && hash[num]!=tmp)
    num=(num+key)%size;
  hash[num]=tmp;
  sum[num]++;
}

int Find(int num)
{
  int tmp=num;
  num=(num+MAX)%size;
  while(hash[num]!=MAX && hash[num]!=tmp)
    num=(num+key)%size;
  if(hash[num]==MAX)
    return 0;
  else
    return sum[num];
}

int main()
{
  int i,j;
  cin>>n;
  for(i=0;i<n;i++)
    cin>>a[i]>>b[i]>>c[i]>>d[i];
  for(i=0;i<size;i++)
    hash[i]=MAX;
```

```
41    for(i=0;i<n;i++)                    //前两列的和
42      for(j=0;j<n;j++)
43        Insert(a[i]+b[j]);
44    for(i=0;i<n;i++)                    //后两列和的相反数
45      for(j=0;j<n;j++)
46        ans+=Find(-(c[i]+d[j]));
47    cout<<ans<<endl;
48    return 0;
49  }
```

另一种方法是把4个数字分成两份，分别两两求和，得到两个长度n×n的一维数组，排序后比较进行匹配即可。其参考代码如下所示。

```
1   //零和游戏2
2   #include <iostream>
3   #include <cstdlib>
4   #include <algorithm>
5   using namespace std;
6   #define maxn 4004
7
8   int n;
9   int f1[maxn * maxn];
10  int f2[maxn * maxn];
11  int a[maxn], b[maxn], c[maxn], d[maxn];
12
13  int main()
14  {
15    cin>>n;
16    for (int i = 0; i < n; i++)
17      cin>>a[i]>>b[i]>>c[i]>>d[i];
18    for (int i = 0; i < n; i++)
19      for (int j = 0; j < n; j++)
20        f1[i * n + j] = a[i] + b[j];
21    for (int i = 0; i < n; i++)
22      for (int j = 0; j < n; j++)
23        f2[i * n + j] = c[i] + d[j];
24    sort(f1, f1 + n * n);
25    sort(f2, f2 + n * n);
26    int r = n * n - 1;
27    int ans = 0;
28    for (int i = 0; i < n * n; i++)
29    {
30      while (r >= 0 && f1[i] + f2[r] > 0)
31        r--;
32      if (r < 0)
33        break;
34      int temp = r;
35      while (temp >= 0 && f1[i] + f2[temp] == 0)
36        ans++, temp--;
37    }
```

```
    cout<<ans<<endl;
    return 0;
}
```

古书密码

【题目描述】 古书密码(POJ 1200)

解读古文明精魂具体含义的密码隐藏在魔法学院图书馆的那些用“神族文字”书写的魔法书中,它拥有保护人类的力量。数千年来,为了维护这个秘密,无数的人不惜献出宝贵的生命。

现在,你需要编写一个程序破解出密码,简单来说,就是给你一个很长的字符串,给你子字符串的长度 N 和这个字符串中共有多少个不同的字符 NC,求这个长的字符串有多少中长度为 N 的子字符串。

举例来说,考虑 N=3,NC=4 并且字符串为"daababac"时,符合条件的字符串有"daa"、"aab"、"aba"、"bab"和"bac"。因此,密码应该是 5。

【输入格式】

第一行为两个整数 N 和 NC,随后是一个字符串。并且保证通过该字符串可能形成的子字符串的最大数目不超过 1600 万。

【输出格式】

输出产生的子字符串数目。

【输入样例】

3 4

daababac

【输出样例】

5

【算法分析】

由于一个字符串有 n 种字符构成,把每种字符对应为 0~n-1 中的一个数字,把字母换成对应的数字之后,对于固定长度的串,每个串都与一个唯一的 n 进制数对应,这样就可以使用哈希表了。

参考程序如下所示。

```
//古书密码
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;
#define maxn 16000005

bool hash[maxn];
int name[260];
int n, nc;
char st[maxn];
```

```
12
13  int main()
14  {
15      cin>>n>>nc>>st;
16      memset(name, 0, sizeof(name));
17      memset(hash, 0, sizeof(hash));
18      int len = strlen(st);
19      int t =0;
20      for (int i =0; i < len; i++)
21          if (name[st[i]] ==0)
22              name[st[i]] = t++;
23      int temp =0;
24      t = nc;
25      for (int i =0; i < n -1; i++)
26      {
27          temp = temp * nc + name[st[i]];
28          t * = nc;
29      }
30      int ans =0;
31      for (int i = n -1; i < len; i++)
32      {
33          temp = (temp * nc + name[st[i]]) % t;
34          if (!hash[temp])
35          {
36              ans++;
37              hash[temp] =true;
38          }
39      }
40      cout<<ans<<endl;
41      return 0;
42  }
```

第七章

并 查 集

并查集算法

并查集是一种树型的数据结构，用于处理一些不相交集合(Disjoint Sets)的合并及查询问题。

为了形象地理解并查集的含义，我们先来看一道例题：

【题目描述】 亲戚(Relation)

魔法世界的王族传承了几千年，时至今日已成为一个庞大的家族，要判断家族的两个人是否是亲戚，是很不容易的一件事，现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。

规定：x 和 y 是亲戚，y 和 z 是亲戚，那么 x 和 z 也是亲戚。如果 x，y 是亲戚，那么 x 的亲戚都是 y 的亲戚，y 的亲戚也都是 x 的亲戚。

【数据输入】

第一行：三个整数 n，m，p，($n \leqslant 20000$，$m \leqslant 1000000$，$p \leqslant 1000000$)，分别表示有 n 个人，m 个亲戚关系，询问 p 对亲戚关系。

以下 m 行：每行两个数 M_i，M_j，$1 \leqslant M_i, M_j \leqslant N$，表示 A_i 和 B_i 具有亲戚关系。

接下来 p 行：每行两个数 P_i，P_j，询问 P_i 和 P_j 是否具有亲戚关系。

【数据输出】

p 行，每行一个"Yes"或"No"。表示第 i 个询问的答案为“具有”或“不具有”亲戚关系。

【输入样例】

```
6 5 3
1 2
1 5
3 4
5 2
```

1 3

1 4

2 3

5 6

【输出样例】

Yes

Yes

No

【算法分析】

初步分析本题是一个图论中判断两个点是否在同一个连通子图中的问题。我们以样例建立无向图如图 7.1 所示(人为顶点,关系为边):

当判断某两人是否为亲戚时,只需检查相应的两顶点是否在同一个连通子图中即可。如果两顶点在同一个连通子图中,则他们是亲戚,否则就不是。显然除 6 以外,1,2,3,4,5 互为亲戚。

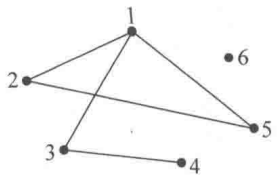


图　7.1

但是这种算法的最大问题是:我们无法存下多至 2000000 条边的图,更不可能在规定的时限内算出结果。

因此我们引入并查集算法:

首先给每个人建立一个集合,集合的元素值只有他自己,表示最开始时他不知道任何人是他的亲戚。以后每次给出一个亲戚关系 a,b,即 a 和他的亲戚与 b 和他的亲戚互为亲戚,则将 a 所在集合与 b 所在集合合并。对于样例数据的操作全过程如下:

1. 初始时每个人均为一个集合如图 7.2 所示。

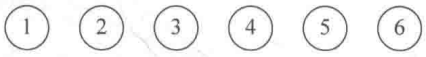


图　7.2

2. 根据 1 和 2 是亲戚的条件,合并 1 的集合和 2 的集合,如图 7.3 所示。

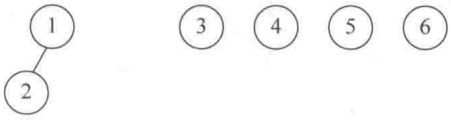


图　7.3

3. 根据 1 和 5 是亲戚的条件,合并 1 的集合和 5 的集合,如图 7.4 所示。

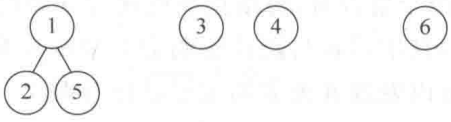


图　7.4

4. 根据 3 和 4 是亲戚的条件，合并 3 的集合和 4 的集合，如图 7.5 所示。

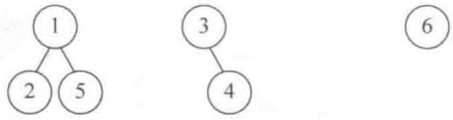


图 7.5

5. 由于 5 和 2 已在同一集合，所以无须合并，再根据 5 和 3 是亲戚的条件，合并 5 的集合和 3 的集合，最后结果为两棵树形成的森林，如图 7.6 所示。

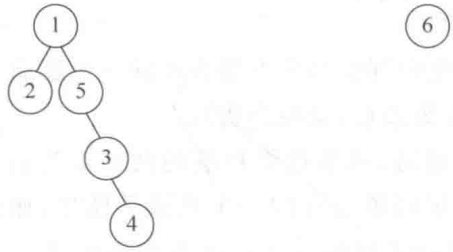


图 7.6

观察图 7.6，我们发现 1，2，3，4，5 形成的树分布不平均，很多结点在树的较深层次，如果当我们查找某个结点是否属于该树时，会花费较多时间，这种树即是所谓的“退化树”。解决该问题我们有两种优化方法：

1. 路径压缩，即查找完某结点至根结点的路径后，一般将这条路径上所有结点都设为根结点。实现方法是查找两遍，第 1 遍找到树根，第 2 遍将经过结点的父结点改为根结点，这样搜索某个结点的时间复杂度将降为 O(1)，如图 7.7 所示。

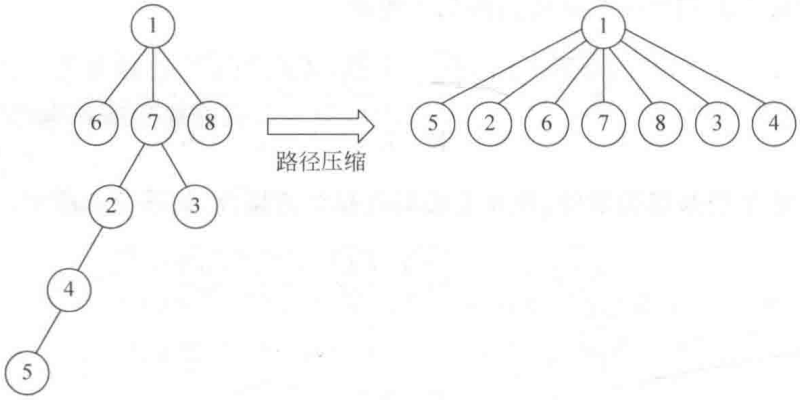


图 7.7

2. 启发式合并，在合并两个集合时，将结点少的树合并到结点多的树上，也可以把高度小的数合并到高度大的树上，其中后者与路径压缩会有少许冲突。

通过上面的分析，我们可以发现并查集的主要操作有两个：

1. 合并两个不相交集合；

2. 判断两个元素是否属于同一集合。

参考程序如下所示。

```
1   //并查集
2   #include <iostream>
3   using namespace std;
4   int N,M,Q;
5   int father[20001];
6
7   int find(int x)                          //判断 x 所属的集合
8   {
9     int root=x;
10    while(father[root]>=0)                 //找到最终的根结点
11      root=father[root];
12    while(root!=x)                         //压缩路径
13    {
14      int temp=father[x];
15      father[x]=root;
16      x=temp;
17    }
18    return root;
19  }
20
21  void Union(int a,int b)                  //合并两个不同集合的元素
22  {
23    int temp=father[a]+father[b];
24    if(father[a]>father[b])                //较小的树成为较大的树的子树
25    {
26      father[a]=b;
27      father[b]=temp;
28    }
29    else
30    {
31      father[b]=a;
32      father[a]=temp;
33    }
34  }
35
36  int main()
37  {
38    int i,a,b;
39    cin>>N>>M>>Q;
40    for(i=1;i<=N;i++)                      //初始化
41      father[i]=-1;                        //各结点的父结点均为-1
42    for(i=1;i<=M;i++)
43    {
44      cin>>a>>b;
45      if(find(a)!=find(b))
46        Union(a,b);                        //联通
47    }
48    for(i=1;i<=Q;i++)
49    {
50      cin>>a>>b;
```

```
        if(find(a)==find(b))
            cout<<"YES"<<endl;
        else
            cout<<"NO"<<endl;
    }
    return 0;
}
```

并查集进行 n 次查找的时间复杂度是 O(na(m,n))(执行 n−1 次合并和 m≥n 次查找)。其中 a 是阿克曼函数(Ackermann Function)的某个反函数,是个增长极其缓慢的函数,它可以看作是小于 5 的,所以可以认为并查集的时间复杂度几乎是线性的。

无所不在的宗教

【题目描述】 无所不在的宗教(POJ 2524)

宗教是人类文明的一种普遍的思想传统,魔法世界几万年来先后诞生和兴亡更替的各种文明,大多都受到宗教的影响。因此,即使由天顶星人编造的创世神谎言被揭穿,魔法世界仍有为数众多的人群信仰着各种各样的宗教。

现在魔法学院有 n(0 < n ≤50000)个学员,已知有 m(0 ≤m≤n(n−1)/2)对宗教信仰相同的学生,请估算这 n 个学生中最少有多少种宗教信仰。

【输入格式】

输入数据有好几组,每一组以两个数字 n,m 开始,代表有 n 个学生,m 对宗教信仰相同的学生,随后的 m 行中,每行包含两个数字 i 和 j,表示学生 i 和学生 j 的宗教信仰相同。最后一行两个 0 代表输入结束。

【输出格式】

输出整数 ans 表示最少有多少种宗教信仰,注意 ans 前面加上 Case+组数+“:”。

【输入样例】

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

5 8

0 0

【输出样例】

Case 1：1

Case 2：7

【算法分析】

并查集的简单应用，先把宗教数初始化为学生数 n，对给出的每对学员 i 和 j，如果他们在不同的集合，那么就合并，并将宗教的数量减一。下面的程序中，合并过程采用的启发函数是树的深度。请与前一个程序比较异同。

参考程序如下所示。

```
//无所不在的宗教
#include<iostream>
using namespace std;

int father[50001];
int rank[50001];                    //rank 值记录树的高度

int find(int x)                     //查找，并把查找路径上的结点都指向跟，以减小树的高度
{
  if(x!=father[x])
    father[x]=find(father[x]);      //路径压缩
  return father[x];
}

void Union(int x,int y)             //合并
{
  int r1=find(x);
  int r2=find(y);
  if(r1==r2)                        //两元素如在同一集合则退出
    return;
  if(rank[r1]<rank[r2])             //rank 值较小的集合合并到大的集合中
    father[r2]=r1;
  else
  {
    if(rank[r1] == rank[r2])        //rank 值相等的树合并后 rank 要增加一
      rank[r2]++;
    father[r1]=r2;
  }
}

int main()
{
  int m,n, a,b,Case=0,i=0;
  while(cin>>n>>m)
  {
    Case++;
    for(i=0;i<n;i++)
```

```
38        {
39            father[i]=i;                    //注意根结点指向自身
40            rank[i]=0;                      //树的初始深度为 0
41        }
42        for(i=0;i<m;i++)
43        {
44            cin>>a>>b;
45            int x=find(a);
46            int y=find(b);
47            if(x!=y)                        //如在不同的集合,则合并
48            {
49                n--;                        //宗教数减一
50                Union(x,y);
51            }
52        }
53        cout<<"Case "<<Case<<": "<<n<<endl;
54    }
55    return 0;
56 }
```

星际争霸

【题目描述】 星际争霸(galaxy.cpp/c/pas)(NOI 2002)

"30 年过去了,可是墨老师还是没有回来。"李旭琳在太空战舰指挥舱通过量子传输技术对一万光年之外的张琪曼说。

"是啊,如果没有他保护探险队员顺利带回技术资料,我们实施星际争霸的计划至少会推迟几百年。"张琪曼坐在地面总指挥部说。

"放心吧,我有一种直觉,墨老师很快就会回来的,哦,演习时间到了,你们准备好了吗?"楚继光坐在另一太空战舰指挥舱说。

魔法历 2193 年,魔法世界第一次大规模星际演习正式开始,由红方李旭琳、蓝方楚继光各率领十万余艘太空战舰模拟对抗。

楚继光将星域战场划分成 30000 列,每列依次编号为 1,2,…,30000。之后,他把自己的战舰也依次编号为 1,2,…,30000,让第 i 号战舰处于第 i 列(i = 1,2,…,30000),形成"一字长蛇阵",诱敌深入。这是初始阵形。当进犯之敌到达时,楚继光会多次发布合并指令,将大部分战舰集中在某几列上,实施密集攻击。合并指令为 M i j,含义为让第 i 号战舰所在的整个战舰队列,作为一个整体(头在前尾在后)接至第 j 号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

但李旭琳早已在战略上取得了主动。在交战中,她可以通过庞大的情报网络随时监听楚继光的舰队调动指令。

在楚继光发布指令调动舰队的同时,李旭琳为了及时了解当前楚继光的战舰分布情况,也会发出一些询问指令:C i j。该指令意思是,询问计算机,楚继光的第 i 号战舰与第 j 号战舰当前是否在同一列中,如果在同一列中,那么它们之间布置有多少战舰。

作为一个资深的高级程序设计员，你被要求编写程序分析楚继光的指令，以及回答李旭琳的询问。

【输入格式】

第1行有1个整数T(1≤T≤500000)，表示总共有T条指令。

以下有T行，每行有一条指令。指令有两种格式：

1. M i j：i和j是两个整数(1≤i,j≤30000)，表示指令涉及的战舰编号。该指令是李旭琳窃听到的楚继光发布的舰队调动指令，并且保证第i号战舰与第j号战舰不在同一列。

2. C i j：i和j是两个整数(1≤i,j≤30000)，表示指令涉及的战舰编号。该指令是楚继光发布的询问指令。

【输出格式】

你的程序应当依次对输入的每一条指令进行分析和处理：

如果是楚继光发布的舰队调动指令，则表示舰队排列发生了变化，你的程序要注意到这一点，但是不要输出任何信息；

如果是李旭琳发布的询问指令，你的程序要输出一行，仅包含一个整数，表示在同一列上，第i号战舰与第j号战舰之间布置的战舰数目。如果第i号战舰与第j号战舰当前不在同一列上，则输出-1。

【输入样例】

4
M 2 3
C 1 2
M 2 4
C 4 2

【输出样例】

-1
1

【算法分析】

本题是并查集的典型应用。由于题目要求第i号战舰所在的整个战舰队列，作为一个整体(头在前尾在后)接至第j号战舰所在的战舰队列的尾部。所以我们即不能进行路径压缩也不能按大小合并，而只需要使用并查集的普通查找和合并的算法即可。

时间复杂度：O(M×logN)，其中M表示M棵等价类树；N表示树的深度。

空间复杂度：O(MAX)，MAX表示集合成员数量。

参考程序如下所示。

方法一：

```
1  //星际争霸1
2  #include<fstream>
3  using namespace std;
4  ifstream cin("galaxy.in");
5  ofstream cout("galaxy.out");
6  int FindFather(int father[], int pos);
```

```
7   void Unite(int father[ ], int posI, int posJ);
8   int Search(int father[ ], int posI, int posJ);
9
10  int main()
11  {
12     const int MAX = 30000;
13     int *father = new int[MAX+1];
14     for (int i=0; i<=MAX; i++)          //所有的数组元素值均初始化为自身序号的相反数
15        father[i] = -i;                  //father[i]  为自己的父亲,而根的 father 为尾结点
16
17     int T;
18     cin >> T;
19
20     char ch;
21     int  posI, posJ;
22
23     for (int i=0; i<T; i++)
24     {
25        cin >> ch;
26        cin >> posI >> posJ;
27        if (ch == 'M')
28           Unite(father, posI, posJ);
29        else
30           cout << Search(father, posI, posJ) << endl;
31     }
32     delete [ ]father;
33     return 0;
34  }
35
36  int FindFather(int father[ ], int pos)  //寻找第 i 号战舰所在的整个战舰队列的对头
37  {
38     if (father[pos] < 0)
39        return pos;
40     return FindFather(father, father[pos]);
41  }
42
43  void Unite(int father[ ], int posI, int posJ)//合并指令
44  {
45     int fI = FindFather(father, posI);  //首先各自去寻找所在的整个战舰队列的对头
46     int fJ = FindFather(father, posJ);
47     //第 i 号战舰所在的整个战舰队列,
48     //作为一个整体(头在前尾在后)接至第 j 号战舰所在的战舰队列的尾部
49     int rear = -father[fJ];             //posJ 所在队列的队尾
50     father[fJ] = father[fI];
51     father[fI] = rear;
52  }
53
54  int Search(int father[ ], int posI, int posJ)//查询指令
55  {
56     int fI = FindFather(father, posI);      //posI 所在队列的队头
```

```
    int rear = -father[fI];                    //posI 所在队列的队尾

    while (rear != posI && rear != posJ) //寻找 posI 或 posJ 在队列中的位置
        rear = father[rear];

    if (rear == posJ)                          //如果先找到 posJ,则设 posJ = posI,继续寻找 posJ
        posJ = posI;

    int len = -1;
    while (rear != father[fI] && rear != posJ)  //寻找 posJ 在队列中的位置
    {
        len++;
        rear = father[rear];
    }

    //如果第 i 号战舰与第 j 号战舰当前不在同一列上,则输出-1
    if (rear != posJ)
        len = -1;

    return len;
}
```

方法二：

```
//星际争霸 2
#include<cstdio>
#include<cmath>
#include<iostream>
using namespace std;
int fa[30001],s[30001],t[30001],m;
/*
fa[i]:i 的父亲,初始值为 i,自己
s[i]:i 所在队列中,i 之前的元素的个数,初始值为 0
t[i]:i 所在队列中,所有的元素和,初始值为 1
*/

/*查找的时候用路径压缩,进行路径压缩的时候 s 也要跟着修改:
s[i]=s[i]+s[fa[i]]  因为 i 要接在根上,  这里要更新下 S */
int find(int v)
{
    int p;
    if(fa[v]==v)return(v);
    p=find(fa[v]);
    s[v]+=s[fa[v]];
    fa[v]=p;
    return(fa[v]);
}

int main()
{
```

```
27    int i,fx,fy,x,y;
28    char ch;
29    freopen("galaxy.in","r",stdin);
30    freopen("galaxy.out","w",stdout);
31    scanf("%d\n",&m);
32
33    for(i=1;i<=30000;i++)
34    {
35      fa[i]=i;
36       s[i]=0;
37       t[i]=1;
38    }
39
40    for(i=1;i<=m;i++)
41    {
42      scanf("%c %d %d\n",&ch,&x,&y);
43      fx=find(x);
44      fy=find(y);
45      if(ch=='M')
46      {
47       if(fx!=fy)
48       {
49           fa[fx]=fy;
50           s[fx]+=t[fy];
51           t[fy]+=t[fx];
52       }
53      }
54      else{
55      if(fx!=fy)
56           printf("-1\n");
57      else
58           if(x==y)printf("0\n");
59           else printf("%d\n",int(abs(s[x]-s[y])-1));
60      }
61    }
62    fclose(stdin);
63    fclose(stdout);
64    return(0);
65  }
```

宇宙食物链

【题目描述】 宇宙食物链(POJ 1182)

宇宙食物链可以通俗地用动物世界中的动物来比喻,假设动物世界中有三类动物 A,B,C,这三类动物的食物链构成了有趣的环形。A 吃 B,B 吃 C,C 吃 A。现有 N 个动物,以 1～N 编号。每个动物都是 A,B,C 中的一种,但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

第一种说法是"1 X Y"，表示 X 和 Y 是同类。

第二种说法是"2 X Y"，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

(1) 当前的话与前面的某些真的话冲突，就是假话；

(2) 当前的话中 X 或 Y 比 N 大，就是假话；

(3) 当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 N(1 ≤N≤50000)和 K 句话(0 ≤ K ≤100000)，输出假话的总数。

【输入格式】

第一行是两个整数 N 和 K，以一个空格分隔。以下 K 行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中 D 表示说法的种类。若 D=1，则表示 X 和 Y 是同类。若 D=2，则表示 X 吃 Y。

【输出格式】

只有一个整数，表示假话的数目。

【输入样例】

```
100 7
1 101 1
2 1 2
2 2 3
2 3 3
1 1 3
2 3 1
1 5 5
```

【输出样例】

```
3
```

【算法分析】

用动物之间"相对"的关系来确定一个并查集。设 father[x]表示 x 的根结点。rank[x]表示 father[x]与 x 的关系。rank[x] == 0 表示 father[x]与 x 同类；1 表示 father[x]吃 x；2 表示 x 吃 father[x]。

怎样判断一句话是不是假话？

假设已读入 D，X，Y，先利用 find_set()函数得到 X，Y 所在集合的代表元素 xf，yf，若它们在同一集合(即 xf == yf)则可以判断这句话的真伪。

若 D == 1 而 rank[X] != rank[Y]则此话为假。(D == 1 表示 X 与 Y 为同类，而从 rank[X] != rank[Y]可以推出 X 与 Y 不同类。矛盾。)

若 D == 2 而 rank[X] == rank[Y] (X 与 Y 为同类)或者 rank[X] == (rank[Y] + 1) % 3 (Y 吃 X)则此话为假。

式子 r[X] == (r[Y] + 1) % 3 推导过程为：假设有 Y 吃 X，那么 r[X]和 r[Y]的值

是怎样的？

我们来列举一下：r[X] = 0 && r[Y] = 2

r[X] = 1 && r[Y] = 0

r[X] = 2 && r[Y] = 1

稍微观察一下就知道 r[X] = (r[Y] + 1) % 3；事实上，对于上个问题有更一般的判断方法：

若(r[Y] − r[X] + 3) % 3 != D − 1，则此话为假。

参考代码如下所示。

```
1   //宇宙食物链
2   #include <iostream>
3   using namespace std;
4
5   int father[50005];
6   int rank[50005];
7
8   void Sent(int x)                          //初始化集合
9   {
10    father[x]=x;
11    rank[x]=0;
12  }
13
14  //查找 x 的集合，回溯时压缩路径，并修改 x 与 father[x]的关系
15  int Find_set(int x)
16  {
17    int t;
18    if(x!=father[x])
19    {
20      t = father[x];
21      father[x]= Find_set(father[x]);
22      //更新 x 与 father[X]的关系
23      rank[x] = (rank[x]+rank[t])%3;
24    }
25    return father[x];
26  }
27
28  void Union(int x,int y,int d)             //合并 x,y 所在的集合
29  {
30    int xf = Find_set(x);
31    int yf = Find_set(y);
32    father[xf] = yf;                        //将集合 xf 合并到 yf 集合上
33    rank[xf]=(rank[y]-rank[x]+3+d)%3;       //更新 xf 与 father[xf]的关系
34  }
35
36  int main()
37  {
38    int total=0;
39    int i,n,k,x,y,d,xf,yf;
```

```
40    cin>>n>>k;
41    for(i=1;i<=n;++i)
42      Sent(i);
43    while(k--)
44    {
45      cin>>d>>x>>y;
46      if(x>n||y>n||(d==2 && x == y))  //如果 x 或 y 比 n 大,或 x 吃 x,是假话
47        total++;
48      else
49      {
50        xf = Find_set(x);
51        yf = Find_set(y);
52        //如果 x,f 的父结点相同,那么可以判断给出的关系是否正确的
53        if(xf == yf)
54        {
55          if((rank[x]-rank[y]+3)%3 != d-1)
56            total++;
57        }
58        else
59          Union(x,y,d-1);                    //否则合并 x,y
60      }
61    }
62    cout<<total<<endl;
63    return 0;
64  }
```

第八章

线 段 树

太空堡垒

【题目描述】 太空堡垒(HDU 1166)

楚继光的太空舰队在太空中沿直线布置了N个太空堡垒,由于李旭琳的舰队采取了某种先进的监测手段,所以楚继光的每个堡垒的飞船数李旭琳都掌握得一清二楚,每个堡垒的飞船数都有可能发生变动,可能增加或减少若干飞船,但这些都逃不过李旭琳舰队的监视。

作为演习总指挥的张琪曼,需要经常了解某一段连续的堡垒一共有多少飞船,例如张琪曼问:"红方指挥官,马上汇报第3个堡垒到第10个堡垒共有多少飞船!"红方指挥官李旭琳就要派你马上开始计算这一段的总飞船数并汇报。但堡垒的飞船数经常变动,而张琪曼每次询问的段都不一样,所以你不得不每次都一个一个汇报堡垒的去数,因此很快就筋疲力尽了。为了避免这种情况的发生,你能写个程序完成这项工作吗?

【输入格式】

第一行一个整数T,表示有T组数据。

每组数据第一行一个正整数N(N≤50000),表示有N个堡垒,接下来有N个正整数,第i个正整数ai代表第i个堡垒里开始时有ai个飞船(1≤ai≤50)。

接下来每行有一条命令,命令有4种形式:

(1) Add i j,i和j为正整数,表示第i个堡垒增加j个飞船(j不超过30);

(2) Sub i j,i和j为正整数,表示第i个堡垒减少j个飞船(j不超过30);

(3) Query i j,i和j为正整数,i≤j,表示询问第i到第j个堡垒的总飞船数;

(4) End表示结束,这条命令在每组数据最后出现。

每组数据最多有40000条命令。

【输出格式】

对第i组数据,首先输出"Case i:"和回车,对于每个Query询问,输出一个整数并回车,表示询问的段中的总飞船数,这个数保持在int以内。

【输入样例】

1
10
1 2 3 4 5 6 7 8 9 10
Query 1 3
Add 3 6
Query 2 7
Sub 10 2
Add 6 3
Query 3 10
End

【输出样例】

Case 1:
6
33
59

【算法分析】

类似这样与区间操作有关的题目。要想在规定时间内出解，需要用到“线段树”这种特殊的数据结构。

例如根据该题的输入样例，可做典型的线段树如图 8.1 所示。

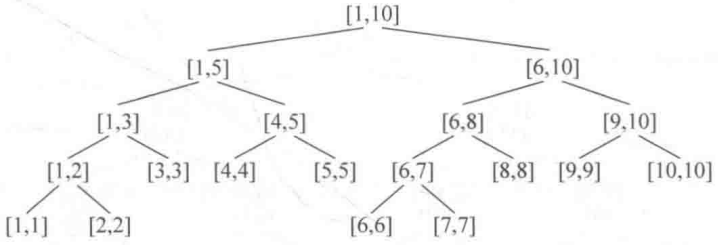


图 8.1

可以看出，线段树是一种二叉平衡树，记为 T[a,b]，参数 a,b 表示区间[a,b]，其中 b−a 称为区间的长度，记为 L。线段树 T[a,b]可递归定义为：

若 L>1，则[a,(a+b)/2]为 T 的左儿子；[(a+b)/2,b]为 T 的右儿子。

若 L=1，则 T 为叶子结点。

线段树的深度不超过 Log2L，线段树把区间上的任意一条线段都分成不超过 2log2L 条线段。所以线段树能在 O(log2L)的时间内完成一条线段的插入、删除和查找等工作。

每个结点可用结构体表示为：

```
struct
{
  int a,b,sum;            //a 为左端点,b 为右端点,sum 为 a,b 端点的总飞船数
}t[140000];               //使用数组
int people[50010];        // 存放每个堡垒的飞船数
```

由于使用了数组保存树结点，因此当父结点下标为 i 时，其左子结点下标为 2×i，其右子结点下标为 2×i+1。例如当根结点[1,10]设为 t[1]时，则其左儿子[1,5]和右儿子[5,10]分别为 t[2]和 t[3]。

此外，使用数组 people[50010]保存每个堡垒的飞船数。

构造初始的线段树由根结点开始依次往下递归构造，代码如下：

```
1   void make(int x,int y,int num)              //x 为左端点，y 为右端点，num 为数组下标
2   {
3     t[num].a=x;                               //确定左端点为 x
4     t[num].b=y;                               //确定右端点为 y
5     if(x==y)                                  //如果 x==y，说明已经是叶子结点了
6       t[num].sum=people[y];                   //则人数为单个的堡垒飞船数即 r[y]
7     else
8     {
9       make(x,(x+y)/2,num+num);                //递归构造左儿子树
10      make((x+y)/2+1,y,num+num+1);            //递归构造右儿子树
11      t[num].sum=t[num+num].sum+t[num+num+1].sum;
12  //父结点的飞船数等于子结点飞船数之和，线段被分成两段
13
14    }
15  }
```

则当堡垒总数为 n 时，执行 make(1,n,1)语句，即 t[1]为根结点。根据本题的输入样例，构造的线段树如图 8.2 所示。

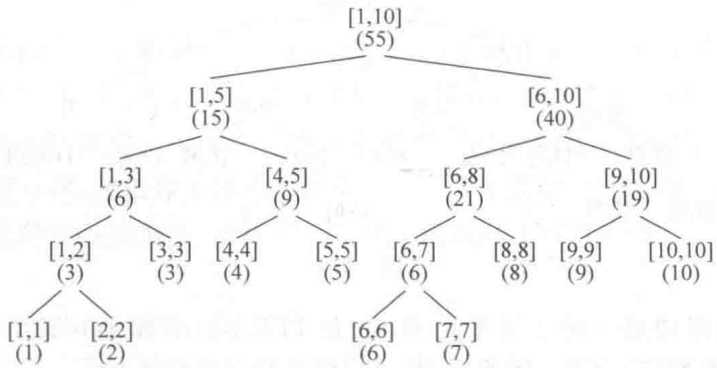


图 8.2

当第 i 个堡垒增加 j 个飞船时，则从根结点 t[1]开始，不断往下递归更改飞船数，即只要包含该点 i 的线段都增加相应的人数 j。函数代码如下：

```
1   void add(int i,int j,int num)               //第 i 个堡垒加 j 个船，初始 num 为 1，即从根结点开始
2   {
3     //从根结点不断往下更改，只要包含该点 i 的线段子都增加相应的数量 j
4     t[num].sum+=j;
5     if(t[num].a==i && t[num].b==i) //如果找到 i 的叶子结点，则停止
6       return;
7     if(i>(t[num].a+t[num].b)/2)          //如果点 i 在该线段的右边
8       add(i,j,num+num+1);                //则递归进入右子结点
```

```
9    else
10      add(i,j,num+num);                //否则递归进入左子结点
11   }
```

与此类似，当第 i 个堡垒减少 j 个飞船时，函数代码如下。

```
1    void sub(int i,int j,int num)
2    {
3        t[num].sum-=j;
4        if(t[num].a==i && t[num].b==i)
5          return;
6        if(i>(t[num].a+t[num].b)/2)
7          sub(i,j,num+num+1);
8        else sub(i,j,num+num);
9    }
```

询问从第 i 到第 j 堡垒的总飞船数的代码如下。

```
1    void query(int i,int j,int num)          //初始 num 为 1,即从根结点开始查找
2    {
3      if(i<=t[num].a && j>=t[num].b)   //找到要求的线段区间,返回其值
4        SUM+=t[num].sum;              //SUM 初始为零,不断累加飞船数,最后返回的是 SUM 值
5      else
6      {
7        int min=(t[num].a+t[num].b)/2;
8        if(i>min)                        //要查询的线段在该线段右边,查询该线段的右子结点
9          query(i,j,num+num+1);
10       else if(j<=min)                  //要查询的线段在该线段左边,查询该线段的左子结点
11         query(i,j,num+num);
12       else
13       {
14         //要查询的线段在该线段中间,分段查询,左右结点都查
15         query(i,j,num+num);
16         query(i,j,num+num+1);
17       }
18     }
19   }
```

参数中的 num 初始值为 1，即每次都从根结点 t[1]开始递归查找，如 query(a,b,1)。此题的关键点是更新结点，区间求和。其完整的代码如下所示。

```
1    //太空堡垒
2    #include<iostream>
3    using namespace std;
4    struct
5    {
6      int a,b,sum;
7    }t[140000];
8    int people[50010],SUM;          //people[50010]存放每个点上的船数,SUM 为输出查询结果
9
```

```
10  void make(int x,int y,int num)    //构造线段树
11  {
12    t[num].a=x;                     //确定左端点为 x
13    t[num].b=y;                     //确定右端点为 y
14    if(x==y)                        //如果 x==y,说明已经是叶子结点了
15      t[num].sum=people[y];         //则人数为单个的堡垒飞船数即 r[y]
16    else
17    {
18      make(x,(x+y)/2,num+num);            //递归构造左儿子树
19      make((x+y)/2+1,y,num+num+1);        //递归构造右儿子树
20      //父结点的人数等于子结点飞船数之和,线段被分成两段
21      t[num].sum=t[num+num].sum+t[num+num+1].sum;
22    }
23  }
24
25  void query(int i,int j,int num)                //初始 num 为 1,即从根结点开始查找
26  {
27    if(i<=t[num].a && j>=t[num].b)      //找到要求的线段区间,返回其值
28      SUM+=t[num].sum;            //SUM 初始为零,不断累加飞船数,最后返回的是 SUM 值
29    else
30    {
31      int min=(t[num].a+t[num].b)/2;
32      if(i>min)                   //要查询的线段在该线段右边,查询该线段的右子结点
33        query(i,j,num+num+1);
34      else if(j<=min)             //要查询的线段在该线段左边,查询该线段的左子结点
35        query(i,j,num+num);
36      else
37      {
38        //要查询的线段在该线段中间,分段查询,左右结点都查
39        query(i,j,num+num);
40        query(i,j,num+num+1);
41      }
42    }
43  }
44
45  void add(int i,int j,int num)                  //第 i 个堡垒增加 j 个飞船,初始 num 为 1
46  {
47    //从根结点不断往下更改,只要包含该点 i 的线段子都增加相应的数量 j
48    t[num].sum+=j;
49    if(t[num].a==i && t[num].b==i)      //如果找到 i 的叶子结点,则停止
50      return;
51    if(i>(t[num].a+t[num].b)/2)         //如果点 i 在该线段的右边
52      add(i,j,num+num+1);               //则递归进入右子结点
53    else
54      add(i,j,num+num);                 //否则递归进入左子结点
55  }
56
57  void sub(int i,int j,int num)
58  {
59    t[num].sum-=j;
```

```
60      if(t[num].a==i && t[num].b==i)
61          return;
62      if(i>(t[num].a+t[num].b)/2)
63          sub(i,j,num+num+1);
64      else
65          sub(i,j,num+num);
66  }
67
68  int main(int argc, char* argv[])
69  {
70      int n,t,i,j;
71      char command[6];
72      cin>>t;
73      j=0;
74      while(t--)
75      {
76          int temp,a,b;
77          cin>>n;
78          people[0]=0;
79          for(i=1;i<=n;i++)
80              cin>>people[i];
81          make(1,n,1);                    //从根结点 t[1]开始
82          cout<<"Case "<<++j<<":"<<endl;
83          while(cin>>command)
84          {
85              if(strcmp(command,"End")==0)
86                  break;
87              else if(strcmp(command,"Query")==0)
88              {
89                  cin>>a>>b;
90                  SUM=0;                  //初始为零
91                  query(a,b,1);           //从根结点 t[1]开始
92                  cout<<SUM<<endl;
93              }
94              else if(strcmp(command,"Add")==0)
95              {
96                  cin>>a>>b;
97                  add(a,b,1);             //从根结点 t[1]开始
98              }
99              else if(strcmp(command,"Sub")==0)
100             {
101                 cin>>a>>b;
102                 sub(a,b,1);             //从根结点 t[1]开始
103             }
104         }
105     }
106     return 0;
107 }
```

天网

【题目描述】 天网(HDU 1754)

除了了解楚继光舰队的某一段连续的堡垒一共有多少飞船以外,张琪曼还会询问从某某堡垒到某某堡垒当中,飞船数最多的是多少。

现在请你写一个程序,模拟回答张琪曼的询问。当然,楚继光有时候会改变某个堡垒中的飞船数。

【输入格式】

本题目包含多组测试,请处理到文件结束。

在每个测试的第一行,有两个正整数 N 和 M ($0<N\leqslant 200000$,$0<M<5000$),分别代表堡垒的数目和操作的数目。

堡垒编号分别从 1 编到 N。

第二行包含 N 个整数,代表这 N 个堡垒的初始飞船数,其中第 i 个数代表 ID 为 i 的堡垒的飞船数。

接下来有 M 行。每一行有一个字符 C (只取'Q'或'U'),和两个正整数 A,B。

当 C 为'Q'的时候,表示这是一条询问操作,它询问 ID 从 A 到 B(包括 A,B)的堡垒当中,飞船数最多的是多少。

当 C 为'U'的时候,表示这是一条更新操作,要求把 ID 为 A 的堡垒的飞船数更改为 B。

【输出格式】

对于每一次询问操作,在一行里面输出最多飞船数。

【输入样例】

```
5 6
1 2 3 4 5
Q 1 5
U 3 6
Q 3 4
Q 4 5
U 2 9
Q 1 5
```

【输出样例】

```
5
6
5
9
```

【算法分析】

这个题是线段树的一个基础应用,即建一个静态树,然后维护各个区间上的最值。用到了三个基本操作即建树、更新和查询。其关键点是结点更新,区间求最值。

完整参考代码如下所示。(还可优化)

```
//天网
#include<stdio.h>
#define Max 200001
struct data
{
  int l,r,max;
}node[3*Max];

int score[Max];                                  //保存分数

int max(int a,int b)
{
    return a>b? a:b;
}

void BuildTree(int left,int right,int num)       //建树
{
  node[num].l=left;
  node[num].r=right;
  if(left==right)
  {
    node[num].max=score[left];
  }
  else
  {
    BuildTree(left,(left+right)>>1,2*num);
    BuildTree(((left+right)>>1)+1,right,2*num+1);
    node[num].max=max(node[2*num].max,node[2*num+1].max);
  }
}

void Update(int stu,int val,int num)             //更新成绩
{
  node[num].max=max(val,node[num].max);          //只更新最大值即可
  if(node[num].l==node[num].r)
    return;
  if(stu<=node[2*num].r)
    Update(stu,val,2*num);
  else
    Update(stu,val,2*num+1);
}

int Query(int left,int right,int num)            //询问操作
{
  if(node[num].l==left && node[num].r==right)
    return node[num].max;
  if(right<=node[2*num].r)
    return Query(left,right,2*num);
```

```
49      if(left>=node[2*num+1].l)
50          return Query(left,right,2*num+1);
51      int mid=(node[num].l+node[num].r)>>1;
52      return max(Query(left,mid,2*num),Query(mid+1,right,2*num+1));
53  }
54
55  int main()
56  {
57      int N,M;
58      while(scanf("%d%d",&N,&M)!=EOF)
59      {
60          int i;
61          for(i=1;i<=N;i++)
62          {
63              scanf("%d",&score[i]);
64          }
65          getchar();
66          char c;
67          int s,e;
68          BuildTree(1,N,1);
69          for(i=0;i<M;i++)
70          {
71              scanf("%c%d%d",&c,&s,&e);
72              getchar();
73              if(c=='U')
74              {
75                  Update(s,e,1);
76              }
77              if(c=='Q')
78              {
79                  printf("%d/n",Query(s,e,1));
80              }
81          }
82      }
83      return 0;
84  }
```

致命武器

【题目描述】 致命武器(HDU 1698)

让很多人想不到的是,在"星际争霸"演习中双方舰队所使用的最强大的武器并不是暗物质炮,也不是黑洞炸弹,更不是时空粉碎机,而是一种绰号为"屠夫的肉钩"的致命武器。这让人不由联想起一个流传已久的游戏——DOTA。

在 DOTA 游戏中,屠夫的肉钩是最可怕的武器,它是由一系列连续的相同长度的金属棒组成,金属棒编号为 1 到 N。初始时金属棒为铜。

屠夫可以改变从 X 到 Y 的连续金属棒为铜、银或金。

钩的总价值计算为 N 的金属棍棒的值的总和。更确切地说,每一种棒的价值计算公式

如下：

对于每个铜棒，值为 1。

对于每一个银棒，价值 2。

对于每一个金色的棍子，值为 3。

现在计算每次操作后的钩子的总价值。

【输入格式】

输入数据包括多组数据，第一行为组数，组数不超过 10 组。

每一组数据中，第一行为数字 N(1≤N≤100000)；第二行为操作数量 Q(0≤Q≤100000)

接下来的 Q 行，每一行包括三个整数 X，Y，1≤X≤Y≤N，Z，1≤Z≤3，表示将从 X 到 Y 区间的金属棒变为 Z，其中 Z=1 表示铜；Z=2 表示银；Z=3 表示金。

【输出格式】

每一组数据，打印出操作后的钩子的总价值。

【输入样例】

1

10

2

1 5 2

5 9 3

【输出样例】

Case 1：The total value of the hook is 24.

【算法分析】

此题的关键点是成段更新，总区间求和。其线段树的每个结点保存颜色值，即 0，1，2，3，其中 0 表示杂色，更新时，当某段结点为纯色，则其左右儿子也为纯色。

参考程序如下所示。

```
//致命武器
#include <iostream>
#include <stdio.h>
#include <string.h>
#define lson l,m,k<<1
#define rson m+1,r,k<<1|1
#define N 100001
using namespace std;
int st[N<<2];                    //st 数组里面记录的是标记,0,1,2,3;0 代表杂色,不是纯色
int flag;

void down(int &k)
{
   st[k<<1]=st[k<<1|1]=st[k];        //左儿子和右儿子均设为该颜色
   st[k]=0;                          //设为杂色//////////////////
}

void build(int l,int r,int k)        //建线段树,均初始为 1
```

```
19  {
20      st[k]=flag;
21      if(l==r)
22          return ;
23      int m=(l+r)>>1;                          //二分
24      build(lson);
25      build(rson);
26  }
27
28  void updata(int &L,int &R,int l,int r,int k)  //更新 L--R,要从 1--n,st[1]开始
29  {
30      if(L<=l&&R>=r)                           //如果更新部分恰好被包含在某线段树内
31      {
32          st[k]=flag;                          //则整个一段都更新为新颜色
33          return ;
34      }
35      if(st[k])                                //如果该结点为纯色,则左右儿子颜色也为该纯色
36          down(k);
37      int m=(l+r)>>1;
38      if(L<=m)
39          updata(L,R,lson);                    //递归更新左儿子
40      if(R>m)
41          updata(L,R,rson);                    //递归更新右儿子
42  }
43
44  int query(int l,int r,int k)
45  {
46      if(st[k])                                //为纯色时,直接成段相乘得出结果,并返回
47          return st[k] * (r-l+1);
48      int m=(l+r)>>1,t1,t2;
49      t1=query(lson);
50      t2=query(rson);
51      return t1+t2;
52  }
53
54  int main()
55  {
56      int T,n,t=1;
57      int L,R,q;
58      scanf("%d",&T);
59      while(T--)
60      {
61          scanf("%d%d",&n,&q);
62          flag=1;
63          build(1,n,1);
64          while(q--)
65          {
66              scanf("%d%d%d",&L,&R,&flag);
67              updata(L,R,1,n,1);
68          }
```

```
        printf("Case %d: The total value of the hook is %d.\n",t++,query(1,n,1));
    }
    return 0;
}
```

无限轮回

【题目描述】 无限轮回(HDU 1394)

墨老师落入的名为“无限轮回”的魔法陷阱，其本质是将历史上不同时空点发生的事件随机虚拟成了一个个镜像世界并因此而形成了一个封闭的时间环，要想用最小代价打破这个封闭时间环需要用到求逆序数的方法。简单来讲，我们将镜像世界中不同事件的时间点以数字形式排成一个序列 a1,a2,…,an，那么该序列的逆序数是这么一对数(ai,aj)，其中 i < j 并且 ai > aj。如 2431 中，21,43,41,31 是逆序，逆序数是 4。

对于一个给字的序列 a1,a2,…,an，如果我们移动第 m(m≥0)个数到序列末尾，则会获得另一个序列，例如 n 的序列的全部序列如下：

a1,a2,…,an−1,an (初始序列，即 m = 0)

a2,a3,…,an,a1 (当 m = 1 时)

a3,a4,…,an,a1,a2 (当 m = 2 时)

…

an,a1,a2,…,an−1 (当 m = n−1 时)

请从上述所有排列中，找出最小的逆序数。

【输入格式】

输入第一行为测试组数，每组数据包括两行，第一行为整数 n (n≤5000)；第二行为从 0 到 n−1 的 n 个整数的排列。

【输出格式】

每一组测试数据，输出最小逆序数。

【输入样例】

10

1 3 6 9 0 8 5 7 4 2

【输出样例】

16

【样例说明】

当序列为 4 2 1 3 6 9 0 8 5 7 时，有 42,41,43,40,21,20,10,30,60,65,90,98,95,97,85,87 共 16 对逆序数。

【算法分析】

由于测试数据范围较小，此题可以使用暴力枚举法，观察图 8.3，可以看出：

由于给定的序列实际上是从 0～n−1 的 n 个连续数的某种排列，因此如果用数组 a[0～n−1]来存原始数据。只需求出原始序列的逆序数 sum，然后对于 a[i](0≤i <n−1)，每挪一个，用 sum 减去挪之前它右边比它小的数的个数(也就是 a[i])，再用 sum 加上挪之后

当序列为 7 4 2 1 3 6 9 0 8 5 时，逆序数为：

74 72 71 73 76 70 75 42 41 43 40 21 20 10 30 60 65 90 98 95 85

当右移一位即 4 2 1 3 6 9 0 8 5 7 时，逆序数为：

42 41 43 40 21 20 10 30 60 65 90 98 95 85 97 87

图 8.3

左边比它大的数的个数(也就是 n－a[i]－1)，就是挪了 a[i]的逆序数。即 sum＝sum－a[i]＋(n－a[i]－1)。

参考代码如下：

```
//无限轮回——暴力枚举
#include <iostream>
using namespace std;

int a[5005];
int main()
{
  int n;
  while (scanf ("%d",&n)!=-1)
  {
    for (int i=0; i<n; i++)
      scanf ("%d",&a[i]);
    int sum=0;
    for (int i=0; i<n; i++)        //计算初始序列的逆序数
      for (int j=i+1; j<n; j++)
        if (a[i]>a[j])
          sum++;

    int minn=sum;
    for (int i=0; i<n; i++)
    {
      sum=sum-a[i]+(n-a[i]-1);
      minn=min(sum,minn);
    }
    printf ("%d/n",minn);
  }
  return 0;
}
```

此代码分为两个主要步骤，第一步是暴力穷举用 sum 把逆序数统计出来；第二步是使用公式推出结果。基本上第二步已很难进行优化，但第一步还有优化的空间。此处我们使用线段树来优化，其时间复杂度为 O(nlogn)。

根据输入样例，建立初始线段树如图 8.4 所示。

当插入第一个元素 1 时，先查找[1,9]区间的数的个数，再更新线段树如图 8.5 所示。

当插入第二个元素 3 时，先查找[3,9]区间的数的个数，再更新线段树如图 8.6 所示。

当插入第三个元素 6 时，先查找[6,9]区间的数的个数，再更新线段树如图 8.7 所示。

依此类推，当输入最后一个元素 2 时，先查找[2,9]区间的数的个数，再更新线段树如图 8.8 所示。

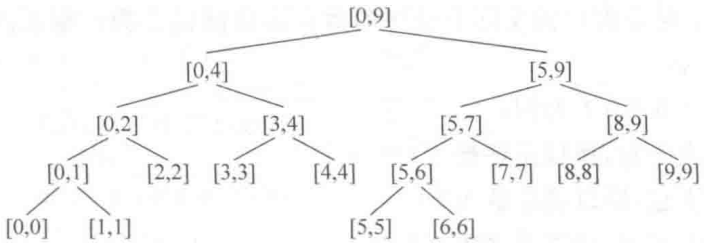


图　8.4

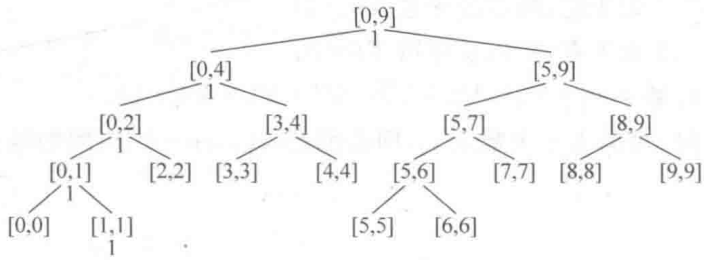


图　8.5

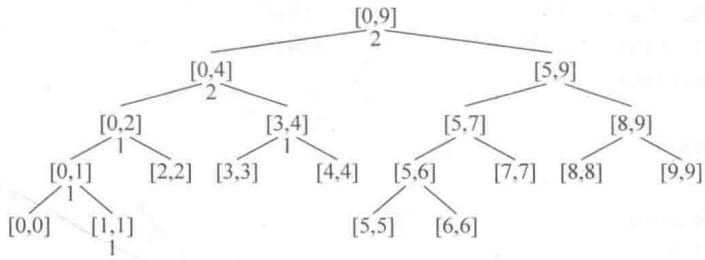


图　8.6

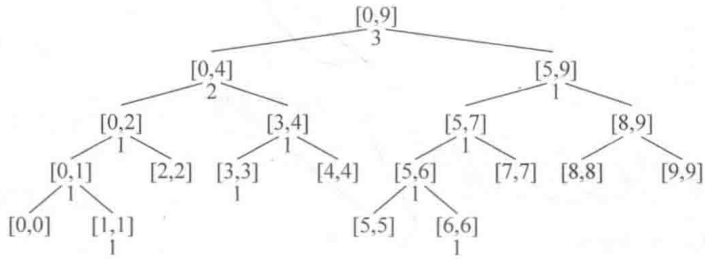


图　8.7

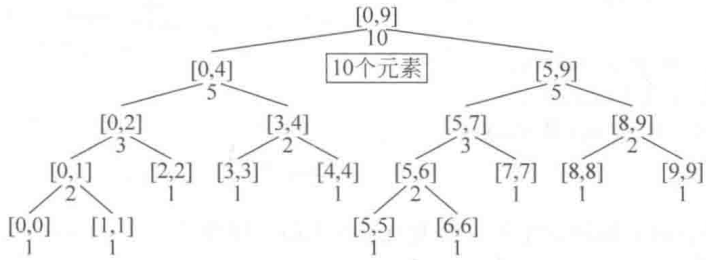


图　8.8

可以看出，每个结点保存的实际上是序列数在该区间的个数。那么如何计算该序列逆序数 sum 值呢？

我们以序列 3 2 5 4 0 1 为例：

3 之前没有数大于它，所以逆序数 V1=0；

2 之前有 3 大于它，所以逆序数 V2=1；

5 之前没有数大于它，所以逆序数 V3=0；

4 之前有 5 大于它，所以逆序数 V4=1；

0 之前有 3,2,5,4 大于它，所以逆序数 V5=4；

1 之前有 3,2,5,4 大于它，所以逆序数 V6=5；

则该序列的逆序数 sum=V1+V2+V3+V4+V5+V6=11。

因此实际操作时，每插入一个数 a[i]，即查询在[a[i],n-1]区间的数有多少，最后依次累加即可。

参考代码如下：

```
1   //无限轮回——线段树
2   #include<iostream>
3   #define LL(x) ((x)<<1)                    //两倍；
4   #define RR(x) ((x)<<1|1)                  //两倍+1；
5   using namespace std;
6
7   struct Seg_Tree
8   {
9     int left,right,val;
10    int calmid()
11    {
12          return (left+right)/2;
13    }
14  }tt[15000];
15
16  int val[5001];                            //保存序列
17
18  void build(int left,int right,int idx)
19  {
20      tt[idx].left = left;
21      tt[idx].right = right;
22      tt[idx].val = 0;
23      if(left == right)
24        return ;
25      int mid = tt[idx].calmid();
26      build(left,mid,LL(idx));
27      build(mid+1,right,RR(idx));
28  }
29
30  //查询在[left,right]区间中有多少数，即比 left 大的逆序数
31  int query(int left,int right,int idx)
32  {
33    if(left == tt[idx].left && right == tt[idx].right)
```

```
34        return tt[idx].val;
35    int mid = tt[idx].calmid();
36    if(right <= mid)
37        return query(left,right,LL(idx));
38    else if(mid < left)
39        return query(left,right,RR(idx));
40    else
41        return query(left,mid,LL(idx)) + query(mid+1,right,RR(idx));
42  }
43
44  //更新所有包含 id 这个数的区间的 val 值,都+1;
45  void update(int id,int idx)
46  {
47    tt[idx].val ++;
48    if(tt[idx].left == tt[idx].right)
49        return ;
50    int mid = tt[idx].calmid();
51    if(id <= mid)
52        update(id,LL(idx));
53    else
54        update(id,RR(idx));
55  }
56
57  int main()
58  {
59    int i,n,sum;
60    while(scanf("%d",&n) == 1)
61    {
62        build(0,n-1,1);
63        for(i=0;i<n;i++)
64            sum = 0;
65        for(i=0;i<n;i++)
66        {
67            scanf("%d",&val[i]);
68            sum += query(val[i],n-1,1);        //此时 val[i]还未插入树中
69            update(val[i],1);                  //此时插入 val[i],即更新树
70        }
71        int ret = sum;
72        for(i=0;i<n;i++)
73        {
74            sum = sum - val[i] + (n - val[i] - 1);
75            ret=min(ret,sum);
76        }
77        printf("%d\n",ret);
78    }
79    return 0;
80  }
```

此题也可以使用树状数组、归并排序等算法解决。

时间锁链

【题目描述】 时间锁链(POJ 2777)

当墨老师找到封闭时间环中最小的逆序对数后,他就可以将时间环展开成有一个长 L 的时间锁链(我们可以将之看成是一根很长的管子),其中 L 是整数,所以我们可以将该管子分为 L 段,并从左到右标记为 1,2,…,L。现在对管子有两种操作:

1. "C A B C"将 A 到 B 的数都标记为 C(我们可形象地看成是染成 C 这种颜色)。

2. "P A B"输出 A 和 B 之间不同颜色的数目。

颜色有 T 种,标记为 1,2,3,…,T,T 是一个很小的值,初始时管子的颜色为 1。

【输入格式】

第一行为 L (1≤L≤100000),T (1≤T≤30)和 O (1≤O ≤100000),其中 O 表示操作数。除后 O 行为操作命令即"C A B C"或"P A B",其中 A 可能比 B 值大。

【输出格式】

输出操作的结果。

【输入样例】

2 2 4

C 1 1 2

P 1 2

C 2 2 2

P 1 2

【输出样例】

2

1

【算法分析】

因为颜色数不多只有 30 种,所以可以巧妙运用二进制位运算,用一个 int 就可以表示当前区间段的颜色情况。比如 1001 表示有两种颜色,如果左子树的当前颜色情况是 101,而右子树的颜色情况是 011,那么父亲的颜色情况就是两者的位或运算,这样就可以避免掉重复的情况。

在更新时,如果我们考虑保证所有的结点都正确地代表段内的颜色的集合,这样将导致所需的更新操作远远增加,即每一次的更新,都要涉及其所有父亲,以及其所有孩子,极端情况下就会需要 O(n)个子段要更新。所以这样更新不现实,另外,考虑线段树,是否可以只更新那些组成该线段的子段呢?

此处使用 lazy 思想。所谓 lazy,就是懒惰,每次不想做太多,只要插入的区间完全覆盖了当前结点所管理的区间就不再往下做了,在当前结点上打上一个 lazy 标记,然后直接返回。下次如果遇到当前结点有 lazy 标记的话,直接传递给两个儿子,自己的标记清空。

我们以染色为例,可以这样想,如果当前结点和它的子孙都有 lazy 标记的话,必定是子孙的先标记,因为如果是自己先标记,那么在访问子孙的时候,必定会将自己的标记下传给儿子,而自己的标记必定会清空,那么 lazy 标记也就不存在了。所以可以肯定,当前的 lazy

标记必定覆盖了子孙的，所以直接下传即可，不需要做任何判断。

本题的关键点是成段更新，区间统计和位运算加速。

参考代码如下所示。

```
//时间链
#include<iostream>
using namespace std;
#define N 100005
int ans;

struct node
{
 int left, right;
 int color;                                          //二进制码标记颜色
 int lazy;                                           //标记是否更新
 int mid()
 {
    return (left+right)>>1;
 }
}E[N*4];

void build(int root, int l, int r)                   //建树
{
   E[root].left=l;
   E[root].right=r;
   E[root].color=1;
   E[root].lazy=0;                                   //初始为未标记
   if(l==r)
      return;
   else
   {
      int mid=E[root].mid();
      build(root<<1, l, mid);
      build(root<<1|1, mid+1, r);
   }
}

void update(int root, int l, int r, int c)           //lazy 思想，延迟更新
{
   if(E[root].left==l && E[root].right==r)           //恰好为该段
   {
      E[root].color=c;
      if(l!=r)
         E[root].lazy=1;
      return;
   }
   else
   {
      if(E[root].lazy==1)                            //如果已做标记
      {
```

```
47          E[root<<1].lazy=E[root<<1|1].lazy=1;                    //两子结点也做标记
48          E[root<<1].color=E[root<<1|1].color=E[root].color;  //两子结点更新父结点颜色
49
50          E[root].lazy=0;                                         //自身清空
51        }
52        int mid=E[root].mid();
53        if(r<=mid)
54          update(root<<1,l,r,c);                                  //更新左结点
55        else if(l>mid)
56          update(root<<1|1,l,r,c);                                //更新右结点
57        else
58        {
59          update(root<<1,l,mid,c);
60          update(root<<1|1,mid+1,r,c);
61        }
62    }
63    E[root].color=E[root<<1].color|E[root<<1|1].color;          //位或
64  }
65
66  int query(int root,int l,int r)
67  {
68    if((E[root].left==1 && E[root].right==r) || E[root].lazy==1)
69      return E[root].color;
70    else
71    {
72      int mid=E[root].mid();
73      if(r<=mid)
74        return query(root<<1,l,r);                                //递归左结点
75      else if(l>mid)
76       return query(root<<1|1,l,r);                               //递归右结点
77      else
78       return query(root<<1,l,mid)|query(root<<1|1,mid+1,r); //两段的合并
79    }
80  }
81
82  int code(int x)                                                 //将 x 转换为二进制码
83  {
84   return 1<<(x-1);                          //例如 x=7 时,返回值为 64,二进制即 1000000
85  }
86
87  int uncode(int x)                          //计算二进制的 x 中有多少个 1
88  {
89    int t=0;
90    while(x>0)
91    {
92      if(x%2==1)
93        t++;
94      x/=2;
95    }
96    return t;
```

```
97  }
98
99  int main()
100 {
101     int i,L,m,t;
102     char c;
103     while(scanf("%d%d%d",&L,&t,&m)!=EOF)
104     {
105         build(1,1,L);
106         for(i=0;i<m;i++)
107         {
108             scanf(" %c",&c);
109             if(c=='C')
110             {
111                 int a,b,c;
112                 scanf("%d%d%d",&a,&b,&c);
113                 c=code(c);                          //将颜色转为二进制
114                 cout<<"c="<<c<<endl;
115                 if(a<b)
116                     update(1,a,b,c);
117                 else
118                     update(1,b,a,c);
119             }
120             else if(c=='P')
121             {
122                 int st,en;
123                 scanf("%d%d",&st,&en);
124                 if(st<en)
125                     ans=uncode(query(1,st,en));
126                 else
127                     ans=uncode(query(1,en,st));
128                 printf("%d\n",ans);
129             }
130         }
131     }
132     return 0;
133 }
```

第九章

二 分 图

二分图最大匹配问题

【问题描述】

“当我怀疑一切事物的存在时，我却不用怀疑我本身的思想，因为此时我唯一可以确定的事就是我自己思想的存在。”这是笛卡儿所说的“我思故我在”的简洁释义。

话说墨老师为了救出其他探险队员，而不慎触动了名为“无限轮回”的魔法陷阱。在某一个虚幻世界中，墨老师成了一名乒乓球教练，乒乓球队中有 N 名男运动员和 M 名女运动员。由于某种原因，在混双比赛中，某些男运动员和某些女运动员不能配对比赛，这使得墨老师很苦恼，他希望你能帮他找出一种最优的配对方案，组成尽可能最多的混双配对。

【输入格式】

第一行三个正整数 N，M，K（$N+M\leqslant 1000$），表示男运动员个数、女运动员个数和可配对数。以下 K 行，每行两个整数 a，b，表示男运动员 a 可与女运动员 b 配对比赛。

【输出格式】

只有一个整数，为最多的混双配对。

【输入样例】

```
5 4 14
1 1
1 2
2 3
3 2
4 2
4 3
4 4
5 4
5 2
```

5 3
3 3
2 4
1 3
2 1

【输出样例】

4

【算法分析】

如果我们将每个运动员看作图的顶点 V，将每个可配对的男女运动员以边线 E 连接，则题目中的样例可作无向图如图 9.1 所示。

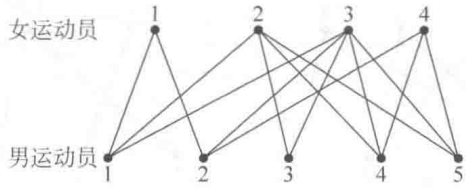


图　9.1

图 9.1 即是所谓的二分图，其定义如下：图 G=[V,E]是一个无向图，顶点集合 V 分成两部分 X 和 Y，G 中每条边的两个端点一定是一个属于 X 而另一个属于 Y。例如运动员分成两部分，一部分是男运动员，一部分是女运动员，只能在男女运动员之间连线，相同性别的运动员之间无连线。其二分图可简记为 G=(X,Y,E)。完全二分图是一种特殊的二分图，可以把图中的顶点分成两个集合，使得第一个集合 X 中的所有顶点都与第二个集合 Y 中的所有顶点相连。记作 Kx,y，如图 9.2 所示。

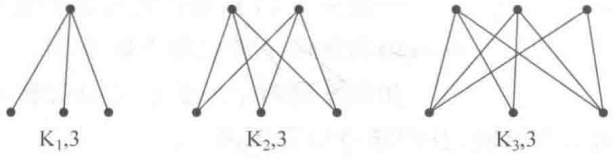


图　9.2

定理：当且仅当无向图 G 的每一个回路的长度均为偶数时，G 才是一个二分图。如果无回路，相当于任一回路的长度为 0，0 视为偶数。

题目要求组成尽可能最多的混双配对，其实质即是求二分图的最大匹配。如果我们将可配对的男女运动员以粗线表示，则其中两种最大匹配方案见图 9.3。

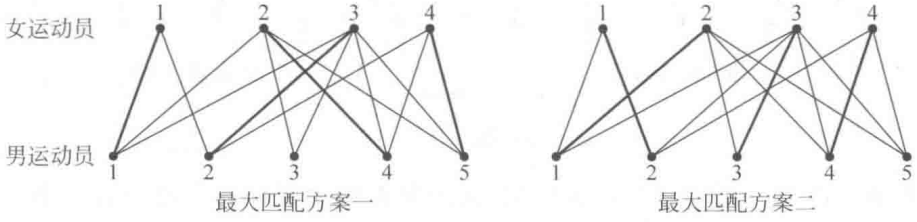


图　9.3

需要注意的是，最大匹配方案可能还会有很多种，但无论如何匹配，最大匹配数是一样的。

稍作观察即可发现，如果将所有的粗线看作一个集合 M，则 M 中的任意两条边都没有公共端点。因此匹配的定义是：设 G=[V,E]是一个无向图，M∈E 是 G 的若干条边的集合，如果 M 中的任意两条边都没有公共端点，就称 M 是一个匹配。

要解决二分图的最大匹配问题，首先我们需要了解交错轨的概念，所谓交错轨，就是设 P 是图 G 的一条路径，如果该路径的任意两条相邻边一定是一条属于 M 而另一条不属于 M，就称 P 是一条交错轨。

如果轨 P 仅含一条边，那么无论该边是否属于匹配 M，P 一定是一条交错轨。

例如上面的两个最大匹配方案中，我们可以找出交错轨如图 9.4 所示。

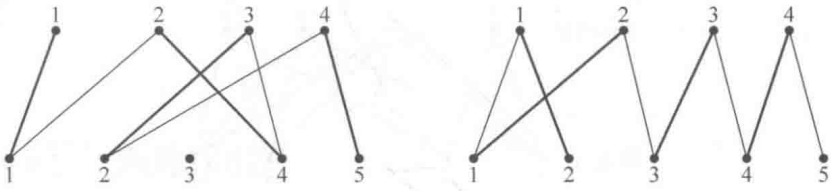


图 9.4

显然，如果从某个顶点出发，不重复地经过其他顶点所形成的一条边的路径，必是交错轨。交错轨中有两个端点都是未盖点的交错轨叫可增广轨。所谓未盖点，直接地讲，在给定了一个匹配 M 以后，我们可以认为每一条属于 M 的边 E "盖住"了两个顶点，就是 E 的两个端点。而其余没有被盖住的顶点，自然就称为未盖点。例如图 9.5 即是可增广轨，两个端点 1 和 5 为未盖点。

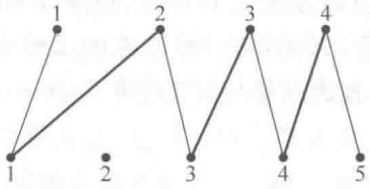


图 9.5

一条关于 M 的增广路径的长度必为奇数，且路上的第一条边和最后一条边都不属于 M。

如果发现两个未盖点之间仅含一条边，则该边必为可增广轨。因为一条单边为交错轨，且两端点均为未盖点。

那么为什么叫可增广轨呢？因为对于 G 的一个匹配 M 来说，如果能够找到一条可增广轨，那么这个匹配 M 一定可以改成一个原匹配数+1 的匹配。方法是将该交错轨粗线改为细线，细线改为粗线，如图 9.6 所示。

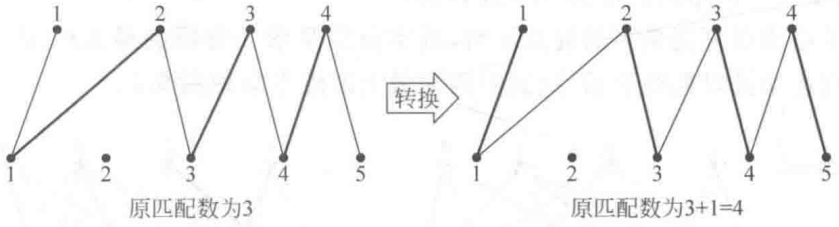


图 9.6

但这种方法并不一定它就是最大匹配，判断是否最大匹配的方法只有一条：若匹配 M 中不存在可增广轨，那么 M 一定是最大匹配。

匈牙利算法

Edmonds 于 1965 年提出一种求二分图最大匹配的有效算法——匈牙利算法，该算法原理是应用了可增广轨的特性，即：初始时，置 M 为空集，然后反复在二分图中找一条关于 M 的增广路径 p，然后进行异或变化以增加一个匹配，直至二分图中不存在 M 的增广路径。

显然，从上面的例子可以看出，搜寻增广路径的方法就是 DFS，可以写成一个递归函数。当然，用 BFS 也完全可以实现。

另外，要完成匈牙利算法，还基于一个重要的定理：

如果从一个点 A 出发，没有找到增广路径，那么无论再从别的点出发找到多少增广路径来改变现在的匹配，从 A 出发都永远找不到增广路径。

DFS 算法的参考代码如下。

```
1   #include <iostream>
2   using namespace std;
3
4   int n, m, k;
5   int link[101];                    //例如 link[1]=3 代表女 1 与男 3 相匹配
6   bool visit [101];                 //记录女运动员结点是否被访问过
7   bool map[101][101];               //map[a][b]=1 代表 a,b 顶点有边相连
8
9   bool dfs(int a)
10  {
11    for (int i=1;i<=m;i++)          //遍历女运动员
12      if (map[a][i]==1&&!visit[i])  //如顶点 i 与 a 有边且未被访问
13      {
14        visit[i] = true;            //标记 i 为已访问
15        if(link[i]==0||dfs(link[i]))//如 i 未在匹配 M 中,或 i 匹配的结点有增广路
16        {
17          link[i]=a;                //异或反转
18          return true;              //返回
19        }
20      }
21    return false;                   //无增广路则返回 false
22  }
23
24  int main()
25  {
26    int a,b,ans=0;
27    cin>>n>>m>>k;                   //n,m 为男、女运动员个数,k 为边数
28    for(int i=1;i<=k;i++)
29    {
30      cin>>a>>b;
31      map[a][b]=true;
32    }
33    for (int i=1; i<=n; i++)        //穷举男运动员顶点
34    {
35      memset(visit, 0, sizeof(visit));//清空上次结点访问标记
```

```
36        if (dfs(i))
37            ans++;                          //从结点 i 尝试扩展
38    }
39    cout<<ans<<endl;
40    return 0;
41 }
```

如果二分图的某一边共有 n 个点，那么最多找 n 条增广路径。如果图中共有 m 条边，那么每找一条增广路径(DFS 或 BFS)时最多把所有边遍历一遍，所花时间也就是 m。所以总的时间大概就是 $O(n\times m)$。

该程序实现简单，理解容易，适用于稠密图，因为边多，所以查找增广路径速度很快，时间复杂度 $O(n^3)$。

图 9.7 是其递归算法的具体运行过程。

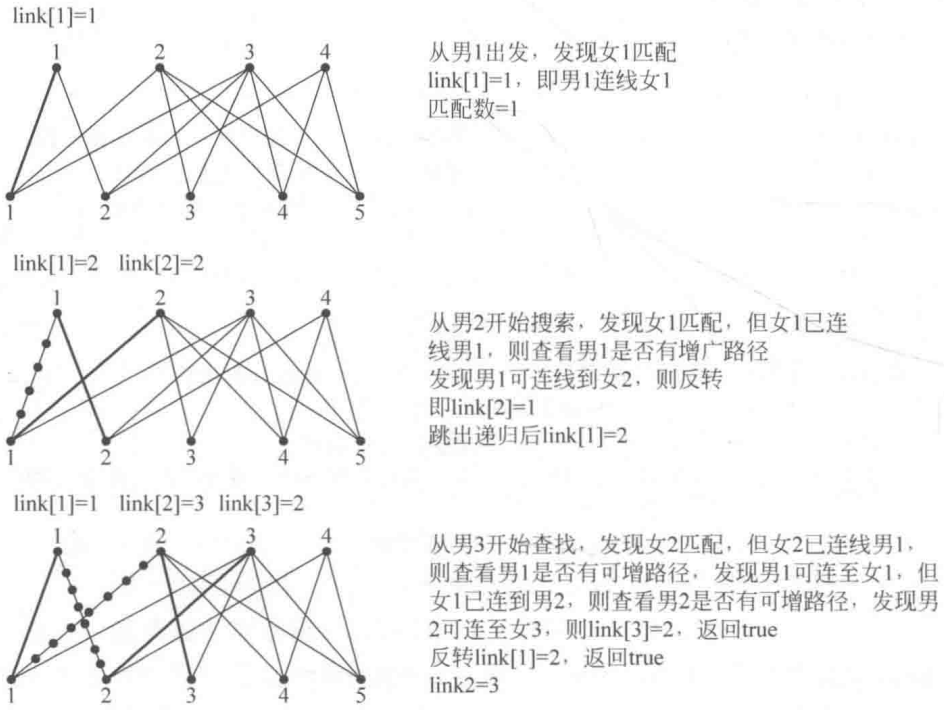


图 9.7

依此类推即可。

BFS 算法适用于稀疏二分图，边少，增广路短。时间复杂度 $O(n^3)$，结构体方式的参考程序如下所示。

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  //定义链表
6  struct link
7  {
```

```
8       int data;                                   //存放数据
9       link * next;                                //指向下一个结点
10      link(int=0);
11  };
12  link::link(int n)                               //d 定义 link 类的一种继承关系
13  {
14      data=n;
15      next=NULL;
16  }
17
18  int n1,n2,m,ans=0;
19  int result[101];                                //记录 n1 中的点匹配的点的编号
20  bool state [101];                               //记录 n1 中的每个点是否被搜索过
21  link *head [101];                               //记录 n2 中的点的邻接结点
22  link *last [101];                               //邻接表的终止位置记录
23
24  //判断能否找到从结点 n 开始的增广路
25  bool find(const int n)
26  {
27      link * t=head[n];
28      while (t!=NULL)                             //n 仍有未查找的邻接结点时
29      {
30          if (!(state[t->data]))                  //如果邻接点 t->data 未被查找过
31          {
32              state[t->data]=true;                //标记 t->data 为已经被找过
33              if ((result[t->data]==0) ||         //如果 t->data 不属于前一个匹配 M
34              (find(result[t->data])))            //如 t->data 匹配到的结点可以寻找到增广路
35              {
36                  result[t->data]=n;              //可更新匹配 M',其中 n1 中的点 t->data 匹配 n
37                  return true;                    //返回匹配成功的标志
38              }
39          }
40          t=t->next;                              //继续查找下一个 n 的邻接结点
41      }
42      return false;
43  }
44
45  int main()
46  {
47      int t1=0, t2=0;
48      cin>>n1>>n2>>m;
49      for (int i=0; i<m; i++)
50      {
51          cin>>t1>>t2;
52          if (last[t1]==NULL)
53              last[t1]=head[t1]=new link(t2);
54          else
55              last[t1]=last[t1]->next=new link(t2);
56      }
57      for (int i=1; i<=n1; i++)
```

```
    {
        memset(state, 0, sizeof(state));
        if (find(i)) ans++;
    }
    cout<<ans<<endl;
    return 0;
}
```

小狗散步

现实生活中，许多问题都可以转化成二分图的最大匹配问题，例如下例：

【题目描述】 小狗散步(SHOI 2001 上海市选拔赛)

李旭琳养了一条宠物小狗 Candy，李旭琳喜欢带着她的小狗 Candy 散步。李旭琳以一定的速度沿着固定路线走，该路线可能自交。Candy 喜欢游览沿途的景点，不过会在给定的 N 个点和主人相遇。小狗和李旭琳同时从(X1，Y1)点出发，并同时在(Xn，Yn)点汇合。小狗的速度最快是李旭琳的两倍。当主人从一个点以直线走向另一个点时，Candy 跑向一个它感兴趣的景点。Candy 每次与李旭琳相遇之前最多只去一个景点。

你现在的任务是：为 Candy 寻找一条路线(有可能与主人的路线部分相同)，使它能够游览最多的景点，并能够准时与主人在给定地点相遇或者汇合。

【输入格式】

输入文件第一行是两个整数 N 和 M(1≤N，M≤100)；

输入文件第二行的 N 个坐标给出了李旭琳的散步路线，即 Candy 和主人相遇地点；

输入文件第三行的 M 个坐标给出了所有 Candy 感兴趣的景点。

所有输入的坐标均不相同，且绝对值不超过 1000。

【输出格式】

输出小狗的移动路线。

第一行是经过的点数；第二行依次为经过的点的坐标(直角坐标系)。

【输入样例】

```
4 5
1 4 5 7 5 2 -2 4
-4 -2 3 9 1 2 -1 3 8 -3
```

【输出样例】

```
6
1 4 3 9 5 7 5 2 1 2 -2 4
```

注意：答案不一定唯一。

【算法分析】

如果直接考虑小狗的行进路线，每到达一个新的汇合点，都要考虑小狗的行动方案。总的计算量为 P(m，n)指数级复杂度的算法无疑是低效、不切实际的。

由于“当主人从一个点以直线走向另一个点时，Candy 跑向一个它感兴趣的景点”，并且主人的移动路线是确定的，所以寻找一条路线可以看作确定小狗在每一段路线的选择：或

是跑向哪一个景点，或是主人的路线重合。显然，如果把主人的移动的每一段抽象成点，其性质和小狗的所向往的景点的性质是不同的。所以图中的点被自然地分成两类，题目要求的也就是这两类之间的关系。

由于在一条线段上，小狗只能跑向一个景点，所以景点是否可达，只和景点到线段两端点的距离有关。

由于一条线段只能和一个景点发生关联，而一个景点也只在第一次访问的时候才有意义，因此任何两项关联都不会有任何交点，这显然满足匹配的定义。

两类定点组成一个二分图，目标求两个点集的最大匹配。

参考程序如下所示。

```
//小狗散步
#include<iostream>
#include<cmath>
using namespace std;
#define maxn 110

struct node
{
  int x,y;
}X[maxn],Y[maxn];

int map[maxn][maxn];
int b[maxn],match[maxn];
bool vis[maxn];
int n,m;

double dis(node x,node y)
{
  return sqrt(abs(x.x-y.x)*abs(x.x-y.x)+abs(x.y-y.y)*abs(x.y-y.y));
}

bool dfs(int i)
{
  int j,cnt;
  for(j=1;j<=n-1;j++)
  {
    //j=map[i][cnt];
    //cout<<i<<":"<<j<<endl;
    if(!vis[j] && map[i][j])
    {
      vis[j]=1;
      if(match[j]==0 || dfs(match[j]))
      {
        match[j]=i;
        return true;
      }
    }
```

```
38      }
39      return false;
40  }
41  int main()
42  {
43
44      int i,j,k;
45      while(scanf("%d%d",&n,&m)!=EOF)
46      {
47          for(i=1;i<=n;i++)
48              scanf("%d%d",&X[i].x,&X[i].y);
49          for(j=1;j<=m;j++)
50              scanf("%d%d",&Y[j].x,&Y[j].y);
51          memset(b,0,sizeof(b));
52          memset(map,0,sizeof(map));
53          for(i=1;i<=n-1;i++)
54          {
55              for(j=1;j<=m;j++)
56              {
57                  //cout<<(dis(X[i],Y[j]) + dis(Y[j],X[i+1]))/2<<endl;
58                  if( dis(X[i],X[i+1]) >(dis(X[i],Y[j]) + dis(Y[j],X[i+1]))/2 )
59                  {
60                      map[j][i]=1;
61                      //cout<<i<<":"<<j<<endl;
62                  }
63              }
64          }
65          int ans=0;
66          memset(match,0,sizeof(match));
67          for(i=1;i<=m;i++)
68          {
69              memset(vis,0,sizeof(vis));
70              if(dfs(i))
71              {
72                  ans++;
73              }
74          }
75          printf("%d\n",ans+n);
76          for(i=1;i<=n;i++)
77          {
78              printf("%d %d",X[i].x,X[i].y);
79              if(i==n)
80                  printf("\n");
81              else
82              {
83                  printf(" ");
84                  if(match[i]!=0)
85                      printf("%d %d ",Y[match[i]].x,Y[match[i]].y);
86              }
87          }
```

```
    }
    return 0;
}
```

最小边覆盖问题

在某一个虚幻世界中，墨老师是一名消防员，他正在规划如何放置消防车以控制如图 9.8 所示的公路网，图 G 中的顶点看成是村庄，每条边看成是一段公路。如果在一段公路上放上一辆消防车，就可以把这段公路两端的两个村庄控制起来了，现在问至少用几辆消防车才能把所有村庄全控制住？这些消防车应该放在哪些公路上？

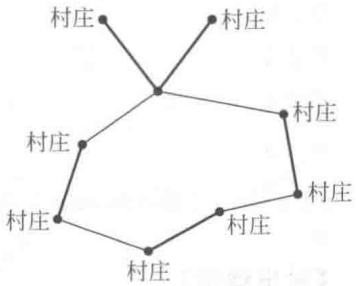


图　9.8

其数学模型是：求无向图 G=[V,E]的一个边集合，这个边集合恰好把图的所有顶点都盖住且含边数最少。这就是所谓的最小边覆盖。

由于一条边只能盖住两个顶点，因此对于一条有 N 个顶点的图 G 来说，盖住所有顶点的边数不会少于 N/2 条，即最小边覆盖中的边数至少为 N/2 条。注意：如果图中有不和任何边关联的孤立点，这个图就不可能有边覆盖。

求最小边覆盖的算法其实很简单：找出最大匹配 M，因为这样盖住的顶点最多，剩下那些未盖点，想要用一条边盖住两个顶点就不可能了，不得不对一个顶点用一条边来盖。

最小点覆盖问题

二分图中，选取最少的点数，使这些点和所有的边都有关联（把所有的边覆盖），叫作最小点覆盖。

Konig 定理：二分图的最小顶点覆盖数＝最大匹配数。

机器安排

【题目描述】 机器安排(POJ 1325)

在某一个虚幻世界中，墨老师是一个工厂的负责人，现有两台机器 A 和 B，A 机器有 0～n－1 种工作模式，B 机器有 0～m－1 种工作模式，初始时两台机器均在 0 工作模式。

现在有 k 项任务，每项任务都能在 A 机器或 B 机器的某个模式中完成，例如任务 0 可在 A 机器的模式 3 或 B 机器的模式 4 中完成，任务 1 可在 A 机器的模式 2 或 B 机器的模式 4 中完成，我们以(i,x,y)表示任务 i 可工作在 A 机器的 x 模式或 B 机器的 y 模式。

显然，要完成所有任务，我们需要时时变动机器的模式，请问如何才能使变动模式数最小？

【输入格式】

第一行包括三个整数 n,m (n,m < 100)和 k (k < 1000)。以下 k 行，每行三个参数即 i,x,y。最末一行以 0 结束。

【输出格式】

一个整数，即最小变动数。

【输入样例】

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

【输出样例】

3

【算法分析】

A，B 两台机器构成一个二分图，按照给出的条件连边。考虑每一个工作其实是由一条边来代表的，那么我们只要用最少的顶点来覆盖所有的边即可。这就是最小覆盖。根据公式：最小覆盖＝最大匹配；对原二分图做一次最大匹配即可。

参考程序如下所示。

```
//机器安排
#include<iostream>
#include<vector>
using namespace std;

int n,m,k;
vector<int>v[201];
int vis[201];
int match[201];

int dfs(int i)
{
    for(int j=0;j<v[i].size();++j)
    {
        if(!vis[v[i][j]+n])        //+n 是调整编号
        {
            vis[v[i][j]+n]=1;
            if(match[v[i][j]+n]==-1||dfs(match[v[i][j]+n]))
            {
                match[v[i][j]+n]=i;
                return 1;
```

```
        }
      }
    }
    return 0;
  }

  int main()
  {
    int i,a,b,c;
    while(scanf("%d",&n),n)
    {
      scanf("%d%d",&m,&k);
      for(i=0;i<=m+n;++i)
        v[i].clear();
      for(i=1;i<=k;++i)
      {
        scanf("%d%d%d",&a,&b,&c);
        if(b&&c)
          v[b].push_back(c);   //只需要从 X 集指向 Y 集就好了 dfs()的时候就只是从 X 到 Y

      }
      a=0;                     //匹配部分
      memset(match,-1,sizeof(match));
      for(i=1;i<=n;i++)
      {
        memset(vis,0,sizeof(vis));
        if(dfs(i))
          a++;
      }
      cout<<a<<endl;
    }
    return 0;
  }
```

最小路径覆盖问题

路径覆盖的定义是：在有向图中找一些路径，使之覆盖了图中的所有顶点，就是任意一个顶点都跟那些路径中的某一条相关联，且任何一个顶点有且只有一条路径与之关联，一个单独的顶点是一条路径。最小路径覆盖就是最少的路径覆盖数。如图 9.9 所示：

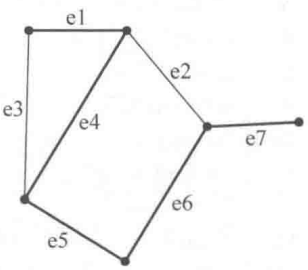


图　9.9

图 9.9 的粗线即为最小路径覆盖的那条路，最小路径覆盖数为 1。如果不考虑图中存在回路，那么每条路径就是一个弱连通子集。

定理：最小路径覆盖＝图的顶点数－最大匹配数。

其实那个最大匹配数并非原图的最大匹配数，而是最小路径覆盖的边的条数，是把图中每个点拆成两个点，再算出来的

最大匹配数。很容易证明两者是相同的。

伞兵任务

【题目描述】 伞兵任务(AJU 1525)

在某一个虚幻世界中,墨老师是伞兵部队中的一员,他们需要占领的某小镇有 m 个路口和 n 条路,这些路都是单向的而且无环路。伞兵可以在任何路口着陆,也可以沿着单行道的方向行走,但不能走到已经走过了的街道。凡是伞兵走过的路口就可以看成被你占领。现在你的任务是写一个程序,用最少的伞兵占领这个城镇所有的路口。

【输入格式】

第一行为测试数据组数,每组数据第一行为路口数 m(0<m≤120),第二行为一个正整数表示 n 条路,以下 n 行每行两个整数,代表一条路起点和终点,为无序排列。

【输出格式】

对于每组测试数据,输出最少伞兵数。

【输入样例】

```
2
4
3
3 4
1 3
2 3
3
3
1 3
1 2
2 3
```

【输出样例】

```
2
1
```

参考程序如下所示。

```
//伞兵任务
#include <stdio.h>
#include <string.h>
int g[125][125];
bool v[125];
int link[125];
int m,n;
int find(int x)
{
  int i;
  for (i=1;i<=m;i++)
  {
```

```
13        if (g[x][i] && !v[i])
14        {
15          v[i]=1;
16          if (!link[i] || find(link[i]))
17          {
18            link[i]=x;
19            return 1;
20          }
21        }
22      }
23      return 0;
24    }
25
26    int main ()
27    {
28      int t,i,x,y;
29      scanf("%d",&t);
30      while (t--)
31      {
32        scanf("%d %d",&m,&n);
33        memset(g,0,sizeof(g));
34        for (i=0;i<n;i++)
35        {
36          scanf("%d %d",&x,&y);
37          g[x][y]=1;
38        }
39        int sum=0;
40        memset(link,0,sizeof(link));
41        for (i=1;i<=m;i++)
42        {
43          memset(v,0,sizeof(v));
44          if (find(i))
45            sum++;
46        }
47        printf ("%d\n",m-sum);
48      }
49      return 0;
50    }
```

出租车

【题目描述】 出租车(POJ 2060)

某一个虚幻世界中，墨老师是一名出租车公司的老板，他所在的城市可看作是一个矩形网格，有 m 个出租车任务，告诉你每个请求的出发时间 s，起点坐标(a,b)，终点坐标(c,d)。出租车从(a,b)到(c,d)需要的时间为|a − c| + |b − d|。若一辆出租车完成某项任务后，能及时赶到另一个任务出发点，则继续完成该任务。注意有些任务可能半夜才能结束。请求出完成所有请求所需要的最少出租车辆数。

【输入格式】

第一行为一整数 N，表示有几组数据，每一组数据第一行有一个整数 M，0 < M < 500，表示出租车任务。下面 M 行中，每行为起始时间，格式为 hh:mm（00:00～23:59），两个整数 a，b 为起始地址坐标，c，d 为目标地址。所有坐标在(0～200)之间。每个任务已按起始时间排序。

【输出格式】

每组数据输出一行，每行只有一个整数，表示最少出租车数。

【输入样例】

2
2
08:00 10 11 9 16
08:07 9 16 10 11
2
08:00 10 11 9 16
08:06 9 16 10 11

【输出样例】

1
2

【算法分析】

将一个请求作为有向无环图的一个结点，若请求 u 完成后能够赶到请求 v，那么在有向图 u 到 v 之间连有向边。然后求这个有向图的最小路径覆盖。注意是有向的！最小路径覆盖数＝结点数 m－最大匹配数。

```
1   //出租车
2   #include <iostream>
3   using namespace std;
4
5   struct P
6   {
7     int ti;
8     int a,b,c,d;
9   };
10
11  P A[505];
12  bool used[505],map[505][505];
13  int Link[505],n;
14
15  int dfs(int x)
16  {
17    int i,j;
18    for(i=0;i<n;i++)
19      if(map[x][i] && !used[i])
20      {
21        used[i]=1;
```

```
22          j=Link[i];
23          Link[i]=x;
24          if(j==-1 || dfs(j))
25            return true;
26          Link[i]=j;
27        }
28    return false;
29  }
30
31  int fun()
32  {
33    int i,num=0;
34    memset(Link,-1,sizeof(Link));
35    for(i=0;i<n;i++)
36    {
37      memset(used,0,sizeof(used));
38      if(dfs(i))
39        num++;
40    }
41    return n-num;
42  }
43
44  bool pd(int x,int y)
45  {
46    int temp;
47    if(x==y)
48      return 0;
49    //做 a 任务所用的时间+a 任务开始的时间
50    temp=abs(A[x].a-A[x].c)+abs(A[x].b-A[x].d)+A[x].ti;
51    //a 任务做完后再做 b 任务的时间
52    temp+=abs(A[x].c-A[y].a)+abs(A[x].d-A[y].b);
53    if(temp<A[y].ti)
54      return 1;
55    return 0;
56  }
57
58  int main()
59  {
60    int h,m,N;
61    cin>>N;
62    while(N--)
63    {
64      cin>>n;
65      for(int i=0;i<n;i++)
66      {
67    cin>>h>>m;
68        A[i].ti=h*60+m;
69        cin>>A[i].a>>A[i].b>>A[i].c>>A[i].d;
70      }
71      for(int i=0;i<n;i++)
```

```
72        {
73            for(int j=0;j<n;j++)
74                map[i][j]=pd(i,j);
75        }
76        cout<<fun()<<endl;
77    }
78    return 0;
79 }
```

最佳匹配问题

KM 算法

在某一个虚幻世界中，墨老师是一个公司的老板，该公司有工作人员 X1，X2，…，Xm，他们去做工作 Y1，Y2，…，Yn，每人适合做其中的一项或几项工作，但是不同的人做不同的工作效益未必一致。请制定一个分工方案，使公司的效益最大。

其数学模型为：在一个二分图内，左顶点为 X，右顶点为 Y，现对于每组左右连接 X_iY_j 有权 w_{ij}，求一种匹配使得所有 w_{ij} 的和最大。

解决此问题使用库恩－曼克莱斯算法（KM 算法）。该算法是数学家 Kuhn 在 1955 年提出、后经数学家 Munkres 完善的一种通过调整完全二分图的顶点标号来计算最佳匹配的方法。

KM 算法是通过给每个顶点一个标号（叫作顶标）来把求最大权匹配的问题转化为求完备匹配的问题的。设顶点 Xi 的顶标为 A[i]，顶点 Yj 的顶标为 B[j]，顶点 Xi 与 Yj 之间的边权为 w[i,j]。在算法执行过程中的任一时刻，对于任一条边(i,j)，A[i]+B[j]≥w[i,j]始终成立。

我们试举一个简单的例子：如下例：当 n=5 时，各边权值如表 9.1 所示。

表 9.1

	Y1	Y2	Y3	Y4	Y5
X1	3	5	5	4	1
X2	2	2	0	2	2
X3	2	4	4	1	0
X4	0	1	1	0	0
X5	1	2	1	3	3

其最大权值为 14，图 9.10 是两种可能的方案：

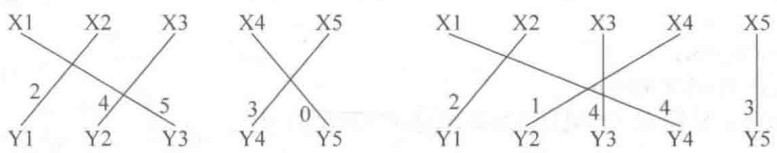


图 9.10

KM 算法的具体运算过程为：

(1) 初始化，为了保证 A[i]+B[j]≥w[i,j]恒成立，令 A[i]为所有与顶点 Xi 关联的边的最大权，B[j]=0。则各顶点的标号(顶标)如下：

A[1]=max{3,5,5,4,1}=5

A[2]=max{2,2,0,2,2}=2

A[3]=max{2,4,4,1,0}=4

A[4]=max{0,1,1,0,0}=1

A[5]=max{1,2,1,3,3}=3

B[1]= B[2]=B[3]=B[4]=B[5]=0

(2) 对于图中的每个顶点，调用匈牙利算法的改进算法，以找到一个完备匹配，使其构成的子图(称作相等子图)中，所有的边(i,j)满足 A[i]+B[j]=w[i,j]。这是基于以下定理：

若由二分图中所有满足 A[i]+B[j]=w[i,j]的边(i,j)构成的子图(称作相等子图)有完备匹配，那么这个完备匹配就是二分图的最大权匹配。

即如果所有满足 A[i]+B[j]=w[i,j]的边组成的导出子图中存在一个完美匹配，那么这个完美匹配肯定就是原图中的最大权匹配，理由很简单，这个匹配的权值之和恰等于所有顶标的和，由于上面的那个不等式，另外的任何匹配方案的权值和都不会大于所有顶标的和。但问题是，对于当前的顶标的导出子图并不一定存在完美匹配，因此需要用某种方法对顶标进行调整。

该定理的证明：对于二分图的任意一个匹配，如果它包含于相等子图，那么它的边权和等于所有顶点的顶标和；如果它有的边不包含于相等子图，那么它的边权和小于所有顶点的顶标和。所以相等子图的完备匹配一定是二分图的最大权匹配。

匈牙利算法的改进算法与原始算法不同点在于：某个顶点的增广路径必须满足 A[i]+B[j]=w[i,j]的条件，而不是原始算法中只要有边相连即可。显然，在不改变各顶标值的情况下，一般不可能顺利地找到最大权匹配，这就需要更改各顶标的值。

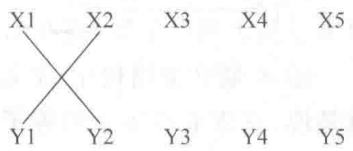


图 9.11

(3) 更改各顶标的值。例如在对 X1,X2,依次寻找增广路径后，其匹配如图 9.11 所示。

现在对 X3 寻找增广路径，其匹配如图 9.12 所示。

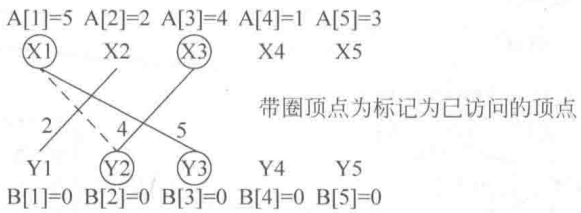


图 9.12

可以看出，由上面三条边构成的子图，均满足 A[i]+B[i]=w[i,j]的条件。现在从顶点 X4 起，调用匈牙利算法，首先标记 X4 为已访问，再寻找增广路径时，找不到满足条件的

边，因此求当前的相同子图的完备匹配失败了。我们实际上得到了一个交错树如图 9.13 所示。

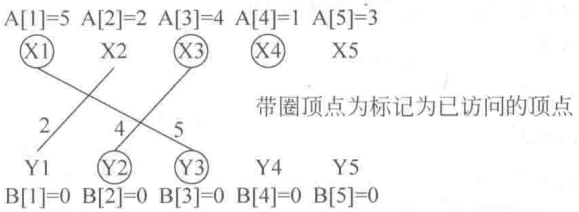


图 9.13

这时我们需要更改各顶标的值。使得有一条边满足 A[i]+B[j]=w[i,j]。具体方法是对已标记访问的顶点的顶标 A[i]和 B[i]，使 A[i]=A[i]+d，B[i]=B[i]−d。其中 d 值的求法是：从所有已标记访问过的 X[i]顶点出发，查找未被标记访问过的 Y[j]顶点集合，则 d 等于 min{A[i]+B[j]−w[i,j]}，此处求得 d 值为 1。这是因为 d 值如果过小了不会使得交错数有所增广，过大有可能会跳过最优解，则更改的各顶标值如下：

A[1]=5−1=4，A[3]=4−1=3，A[4]=1−1=0；

B[2]=0+1=1，B[3]=0+1=1

此时会出现四种可能情况：

① 两端都在交错树中的边(i,j)，A[i]+B[j]的值没有变化。也就是说，它原来属于相等子图，现在仍属于相等子图。如 X1 和 Y3。

② 两端都不在交错树中的边(i,j)，A[i]和 B[j]都没有变化。也就是说，它原来属于(或不属于)相等子图，现在仍属于(或不属于)相等子图。如 X5 和 Y5。

③ X 端不在交错树中，Y 端在交错树中的边(i,j)，它的 A[i]+B[j]的值有所增大。它原来不属于相等子图，现在仍不属于相等子图。如 X5 和 Y2。

④ X 端在交错树中，Y 端不在交错树中的边(i,j)，它的 A[i]+B[j]的值有所减小。也就是说，它原来不属于相等子图，现在可能进入了相等子图，因而使相等子图得到了扩大。如 X4 和 Y1。

此时由于 X4 与 Y1 现在满足 A[4]+B[1]=w[4][1]，因此调用匈牙利算法时，该边加入匹配。其匹配结果如图 9.14 所示。

依此类推。

KM 的数学构造法证明：

设矩阵 C 如图 9.15 所示。

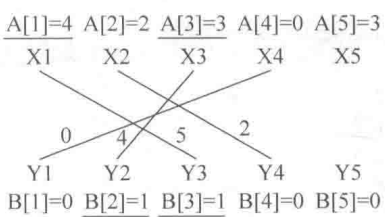


图 9.14

$$C=\begin{array}{c} \left\{\begin{array}{ccccc} 3 & 5 & 5 & 4 & 1 \\ 2 & 2 & 0 & 2 & 2 \\ 2 & 4 & 4 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 3 & 3 \end{array}\right\} \begin{array}{c} x1 \\ x2 \\ x3 \\ x4 \\ x5 \end{array} \\ \begin{array}{ccccc} y1 & y2 & y3 & y4 & y5 \end{array} \end{array}$$

图 9.15

初始标号如下：

A[1]=max{3,5,5,4,1}=5

A[2]=max{2,2,0,2,2}=2

A[3]=max{2,4,4,1,0}=4

A[4]=max{0,1,1,0,0}=1

A[5]=max{1,2,1,3,3}=3

B[1]= B[2]=B[3]=B[4]=B[5]=0

设矩阵 B,使得 Bij=A[i]+B[j]−w[i,j],则如图 9.16 所示。

我们可以看到,矩阵 B 的所有零元素可被第 2、5 行和第 2、3 列所覆盖。则我们对未覆盖的行所对应的顶标进行调整,即 A[1]=A[1]+1,A[3]=A[3]+1,A[4]=A[4]+1,对覆盖的列所对应的顶标进行调整,即 B[2]=B[2]−1,B[3]=B[3]−1,则矩阵 B 将变为如图 9.17 所示。

此时已不能用少于数目 5 的行和列覆盖住矩阵 B 的所有元素了。则我们可以根据该矩阵 B 的零元素所在的行和列作二分图,例如 B12=B13=B14=0,则二分图中与 X1 相连的边有(X1,Y2),(X1,Y3),(X1,Y4),其余类推,再利用匈牙利算法对该二分图进行最大匹配,则该匹配即为最佳匹配,如图 9.18 所示。

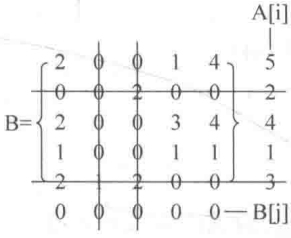


图　9.16

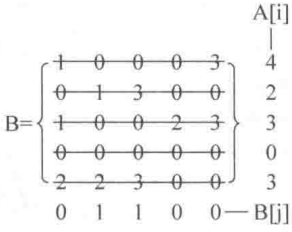


图　9.17

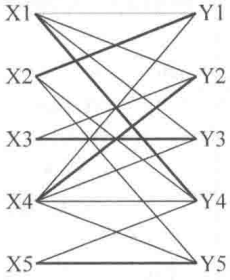


图　9.18

参考程序如下：

```
//KM 算法
#include <algorithm>
#include <iostream>
using namespace std;

const int MAX = 1024;

int n;                                  // X 的大小
int w[MAX] [MAX];                       // w[i] [j]表示从 i 到 j 的权重
int A[MAX],B[MAX];                      // 顶标
bool VisitX [MAX], VisitY [MAX];        //标记是否被搜索过
int match [MAX];                        // match[1]=3 表示顶点 1 与顶点 3 匹配

void init (int size)                    // 初始化权重
{
   n = size;
```

```
17    for (int i = 0; i < n; i ++)
18      for (int j = 0; j < n; j ++)
19        cin>>w[i][j];
20  }
21
22  bool dfs(int u)                                  // 从 X(u)寻找增广路
23  {
24    VisitX [u] = true;
25    for (int v = 0; v < n; v ++)
26      if (!VisitY[v] && A[u]+B[v]==w[u][v]) //满足 A[u]+B[v]==w[u][v]才可加入
27      {
28        VisitY[v]=true;
29        if (match[v]==-1 || dfs(match[v]))
30        {
31          match [v]=u;                             //重置匹配
32          return true;                             //有增广路则返回 true
33        }
34      }
35    return false;
36  }
37
38  int km(bool maxsum)
39  {
40    int i, j;
41    if (!maxsum)                                   //如果是求最小权
42      for (i = 0; i < n; i ++)
43        for (j = 0; j < n; j ++)
44          w[i] [j] = -w[i] [j];
45
46    for (i = 0; i < n; i ++)                       // 初始化标号,即 A[i]=max,B[i]=0
47    {
48      A[i] = -0x1FFFFFFF;
49      B[i] = 0;
50      for (j = 0; j < n; j ++)
51        if (A[i] < w[i] [j])
52          A[i] = w[i] [j];
53    }
54    memset (match, -1, sizeof (match));            //均未匹配
55    for (int u = 0; u < n; u ++)
56      while (1)
57      {
58        memset (VisitX, 0, sizeof (VisitX));
59        memset (VisitY, 0, sizeof (VisitY));
60        if (dfs (u))                               //如有增广路径,退出
61          break;
62
63        int d=0x7FFFFFFF;                          // 如无增广路径,计算 d 值
64        for (i = 0; i < n; i ++)
65          if (VisitX [i])
66            for (j = 0; j < n; j ++)
```

```
67                if(!VisitY [j])
68                    d= min(A[i]+B[j]-w[i][j],d);
69
70            for (i = 0; i < n; i ++)                    //更改标号
71            {
72              if (VisitX [i])
73                A[i] -= d;
74              if (VisitY [i])
75                B[i] += d;
76            }
77        }
78
79      int ans = 0;
80      for (i = 0; i < n; i ++)
81        ans += w[match[i]][i];
82
83      if (!maxsum)                                      //如果是最小权
84      {
85        ans = -ans;
86        for (i = 0; i < n; i ++)
87          for (j = 0; j < n; j ++)
88            w[i] [j] = -w[i][j];                        //恢复原 weight□□的值
89      }
90      return ans;
91    }
92
93    int main()
94    {
95      int n;
96      cin>>n;
97      init (n);
98      int cost = km(true);                              //求最大权
99      cout<<cost<<endl;
100     return 0;
101   }
```

回家

【题目描述】 回家(POJ 2195)

名为“无限轮回”的魔法陷阱扭曲了时间流逝的速度,即魔法陷阱内的 1 天,相当于魔法世界的 30 年,而对于魔法陷阱中的人来说,却感觉度过了漫长的 500 年。墨老师在经历了无数的虚幻世界后,并没有磨灭自我意识,并且最终,他成功地找到了跳出魔法陷阱回到真实世界的方法。现在的问题是,他所剩的魔法力不多了,他如何耗费最小的魔法力回家呢?

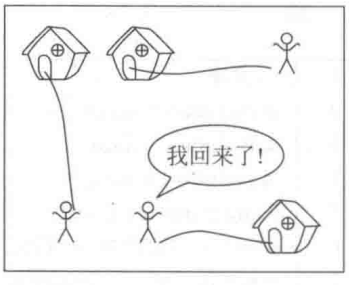


图 9.19

简单来说,就是在一个格子地图上有 n 个人(以 m 表示)和 n 个空间(以 H 表示,因为你可以把它想象成房子)如图 9.19 所示,每个人每次能向水平方向或者竖直方向移动一个格子,每移动一格要付出 1 个单位的魔法力,直到他进

入某个房子里，每个房子只能有一个人。最后要求怎么安排这些人，使房子和人一一配对，使最后的代价最小。

【输入格式】

输入有多组测试数据，每组数据第一行为两个整数 N 和 M，N 为图的行数，M 为图的列数，以下 N 行描述该图，N 和 M 的值在(2～100)之间，空间数和人数为同样数字。空间不会超过 100 个。结束以 0 0 标志。

【输出格式】

每组数据输出一个最少付出的魔法力。

【输入样例】

```
2 2
.m
H.
5 5
HH..m
.....
.....
.....
mm..H
7 8
...H....
...H....
...H....
mmmHmmmm
...H....
...H....
...H....
0 0
```

【输出样例】

```
2
10
28
```

```
//回家
#include<iostream>
#include<vector>
#include<cmath>
using namespace std;
const int MAXN = 110;
const int inf = -0x3fffffff;
struct node
```

```
9   {
10    int pos_x, pos_y;
11    int node_num;
12    node()
13    {
14      pos_x=0, pos_y=0;
15      node_num=0;
16    }
17  };
18  int bin_map[MAXN][MAXN];
19  char ch[MAXN][MAXN];
20  bool vis_x[MAXN], vis_y[MAXN];
21  int lx[MAXN], ly[MAXN];
22  int result[MAXN];
23
24  bool find(int a, int n)                    //判断第 a 能否找到房子
25  {
26    vis_x[a] = true;
27    for(int i=1;i!=n;i++)
28    {
29      if(!vis_y[i]&&bin_map[a][i]==lx[a]+ly[i])
30      {
31        vis_y[i] = true;
32        if(result[i]==-1||find(result[i],n))
33        {
34  result[i] = a;
35  return true;
36        }
37      }
38    }
39    return false;
40  }
41  int best_match(int n)                      //开始匹配
42  {
43    memset(ly,0,sizeof(ly));
44    for(int i=1;i!=n;i++)
45    {
46      lx[i] = inf;
47      for(int j=1;j!=n;j++)
48      {
49        if(lx[i]<bin_map[i][j])
50          lx[i] = bin_map[i][j];
51        }
52    }
53    memset(result,-1,sizeof(result));
54    for(int u=1;u!=n;u++)
55    {
56      while(1)
57      {
58        memset(vis_x,false,sizeof(vis_x));
```

```
59          memset(vis_y,false,sizeof(vis_y));
60          if(find(u,n))//
61            break;
62          int dx = -inf;
63          for(int i=1;i!=n;i++)
64          {
65            if(vis_x[i])
66            {
67              for(int j=1;j!=n;j++)
68              {
69                if(!vis_y[j])
70                  dx = min(dx,lx[i]+ly[j]-bin_map[i][j]);
71              }
72            }
73          }
74          for(int i=1;i!=n+1;i++)
75          {
76            if(vis_x[i])
77             lx[i]-=dx;
78            if(vis_y[i])
79             ly[i]+=dx;
80          }
81        }
82      }
83      int sum = 0;
84      for(int i=1;i!=n;i++)
85      {
86        sum-=bin_map[result[i]][i];
87      }
88      return sum;
89    }
90
91    int main()
92    {
93      int m,n;
94      while(cin>>m>>n&&(m!=0&&n!=0))
95      {
96        vector<node> men;
97        vector<node> house;
98        int count_m(0),count_h(0);
99        for(int i=1;i!=m+1;i++)
100       {
101         for(int j=1;j!=n+1;j++)
102         {
103           cin>>ch[i][j];
104           if(ch[i][j]=='m')
105           {
106             count_m++;
107             node N;N.pos_x = j;N.pos_y = i;N.node_num = count_m;
108             men.push_back(N);
```

```
109            }
110            if(ch[i][j]==='H')
111            {
112              count_h++;
113              node N;N.pos_x = j;N.pos_y = i;N.node_num = count_h;
114              house.push_back(N);
115            }
116          }
117        }
118        memset(bin_map,0,sizeof(bin_map));
119        for(vector<node> ::size_type t=0;t!=house.size();t++)   //构造二分图
120        {
121          node N1 = men[t];
122          for(vector<node> ::size_type i=0;i!=house.size();i++)
123          {
124            int x2 = house[i].pos_x;int y2 = house[i].pos_y;
125            int x1 = N1.pos_x;int y1 = N1.pos_y;
126            int num = -(abs(x1-x2)+abs(y1-y2));   //值变为负数转化为,求最大匹配
127            bin_map[N1.node_num][house[i].node_num] = num;
128          }
129        }
130        int cost = best_match(count_m+1);
131        cout<<cost<<endl;
132      }system("pause");
133      return 0;
134    }
```

参考文献

[1] 董永建. 信息学奥赛一本通. 北京：科学技术文献出版社，2013.

[2] 汪长喜. 信息技术与信息学竞赛. 北京：清华大学出版社，2008.

[3] 腾伟. 青少年信息学奥林匹克竞赛规范教程. 北京：清华大学出版社，2006.

[4] [美]Thomas H. Cormen Charles E. Leiserson. 算法导论. 北京：机械工业出版社，2006.

[5] 林厚从. 数学与程序设计. 南京：东南大学出版社，2008.

[6] 刘汝佳. 算法艺术与信息学竞赛. 北京：清华大学出版社，2004.

[7] 李建江. 全国青少年信息学联赛培训教材. 杭州：浙江大学出版社，2004.

[8] 曹文. 全国青少年奥林匹克联赛培训教材. 南京：南京大学出版社，2004.

[9] 戴胜军. 信息学奥林匹克普及版. 长沙：湖南科学技术出版社，2006.

[10] 陈平. 信息学(计算机)奥林匹克(高级本). 南京：南京大学出版社，1999.

[11] 叶至军. C++ STL 开发技术导引. 北京：人民邮电出版社，2007.

[12] 张新华. 信息学初级算法艺术 C++版. 长春：吉林人民出版社，2012.

[13] 吴耀斌. 信息学奥林匹克教程. 长沙：湖南师范大学出版社，2003.

[14] 谭浩强. C 程序设计. 北京：清华大学出版社，1995.

[15] 郑莉，董渊. C++语言程序设计. 北京：清华大学出版社，2004.

[16] 李春葆，陶红艳. C++语言程序设计. 北京：清华大学出版社，2008.

[17] 吴文虎. 全国信息学奥林匹克联赛培训教程(二). 北京：清华大学出版社，2004.

[18] 刘艳飞. C 语言范例开发大全. 北京：清华大学出版社，2010.

[19] 吴文虎，王建德. 图论的算法与程序设计. 北京：清华大学出版社，2002.

[20] Matrix67，www. matrix67. com/blog/.

算法竞赛宝典

第一部　语言及算法入门
第二部　基础算法艺术
第三部　基础数据结构

一套你能看懂的算法书
一套写了十年的精心之作
NOI、ACM算法竞赛的金牌之路
应聘全球著名IT企业的必备技能

手机扫二维码可下载本书资源包
（含源文件和测试数据）

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

ISBN 978-7-302-40950-2
9 787302 409502 >
定价：42.00元

[General Information]
SS号=13978123