基本数据结构在信息学竞赛中的应用

安徽省芜湖市第一中学 朱晨光

目 录

摘要

本文介绍了几种基本数据结构（例如线性表、队列）在信息学竞赛中的应用，并通过文中的几道例题集中体现了这些数据结构的重要作用。全文可以分为如下几个部分：

一、介绍几种常用的基本数据结构；

二、通过几道例题说明基本数据结构的重要作用；

三、总结全文并探讨应用基本数据结构在思想上带来的启示。

关键字

基本数据结构　线性表　队列　双向链表　栈
编程复杂度　时间复杂度
辨证关系　螺旋式发展

正文

引言

在当今的信息学竞赛中，各种高难度的题目层出不穷。而与这些题目相伴而来的便是很高的编程复杂度。随着计算机科学的不断进步，有越来越多高效而实用的数据结构应运而生。但是，其惊人的编程复杂度使得我们在比赛时必须小心翼翼，慎之又慎，并且经常因为一个微小的疏漏而导致全盘皆输。

然而，并非所有的题目都只能运用复杂的数据结构加以解决。有些时候，常常被我们忽略的基本数据结构也大有用武之地。灵活地运用基本数据结构，可以使我们在紧张的信息学比赛中赢得宝贵的时间，增加成功的概率。

第一部分——基本数据结构的介绍

其实，这一部分的内容可以在所有介绍数据结构的书中找到，这里只做比较简单的介绍。

一、 线性表

线性表是最常用且最简单的一种数据结构。简言之，一个线性表是 n 个数据元素的有限序列。对线性表进行的基本操作有如下几种：

1) INITIATE(L) 初始化操作
2) LENGTH(L) 求长度函数
3) GET(L,i) 取元素函数
4) PRIOR(L,element) 求前驱函数
5) NEXT(L,element) 求后继函数

6) LOCATE(L,x) 定位函数
7) INSERT(L,i,b) 前插操作
8) DELETE(L,i) 删除操作
9) EMPTY(L) 判空表操作
10) CLEAR(L) 表置空操作

线性表的顺序存储结构

在计算机内，可以用不同的方式来表示线性表，其中最简单和最常用的方式是用一组地址连续的存储单元依次存储线性表的元素（即我们通常所说的“一维数组”）（如图 1）。

存储地址	内存状态	元素在线性表中的序号	
b	a_1	1	
b+L	a_2	2	
……	……	……	
b+(i-1)L	a_i	i	
……	……	……	
b+(n-1)L	a_n	n	
b+nL			空闲
……			
b+(maxlen-1)L			

图 1

应用这种存储结构，可以实现对线性表中任一数据元素的随机存取。所以线性表的顺序存储结构是一种随机存取的存储结构。在 Pascal 与 C 语言中可以用一维数组描述之。

可以很容易证明，在线性表的顺序存储结构中，删除与插入任意一个元素的时间复杂度为 O(N)，而定位一个元素的时间复杂度为 O(1).

线性表的链式存储结构

1、 线性链表

线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素。对于每个数据元素 a_i 来说，除了存储其本身的信息之外，还需存储一个指示其直接后继的信息。这两部分信息组成数据元素 a_i 的存储映象，称为结点，其中包括存储数据元素信息的数据域和存储直接后继存储位置的指针域（称为指针或链）。n 个节点的链组成一个链表，称为线性链表或单链表（如图 2）。

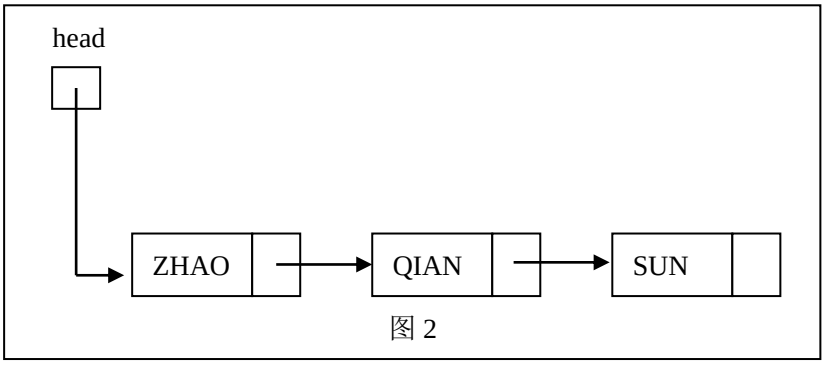


图 2

其中 head 为头节点，它的指针域存储第一个元素结点的存储位置。

2、 循环链表

循环链表是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环（如图 3）。

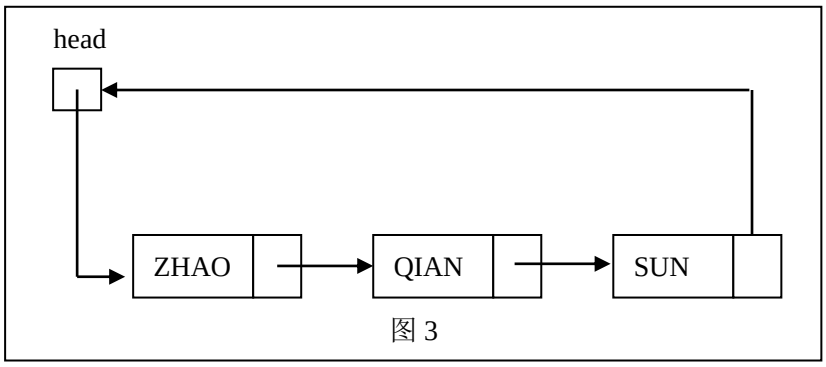


图 3

3、 双向链表

顾名思义，双向链表的结点中有两个指针域，其一指向直接后继，另一指向直接前驱。当然，双向链表也有循环表，其结构如图 4 所示。

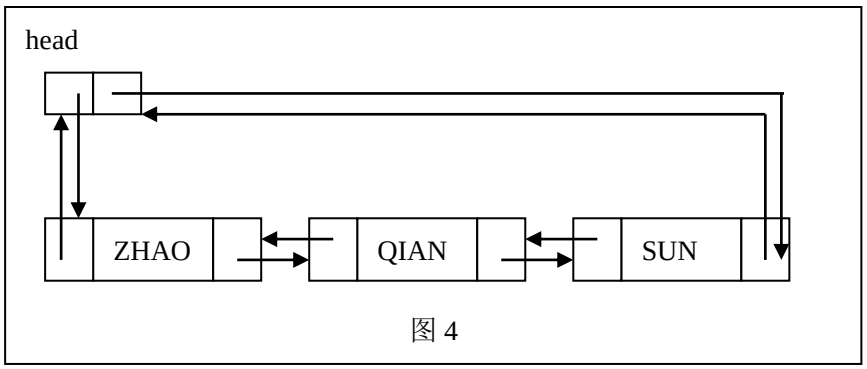


图 4

可以很容易证明，在线性表的链式存储结构中，删除与插入任意一个已经定位的元素的时间复杂度为 O(1)，而定位一个元素的时间复杂度为 O(N).

二、 栈

栈是限定仅在表尾进行插入或删除操作的线性表。因此，表尾端称为栈顶，相应地，表头端称为栈底。不含元素的空表称为空栈。

根据图 5 可以很明显地看出，栈的修改是按后进先出的原则进行的。因此，栈又称后进先出（Last In First Out）的线性表（简称 LIFO 结构）。

栈的基本操作有：

INISTACK(S) 初始化操作

EMPTY(S) 判栈空操作

PUSH(S,x)　　入栈操作
POP(S,x)　　出栈函数
GETTOP(S)　取栈顶元素函数
CLEAR(S)　　栈置空操作
CURRENT_SIZE(S)　求当前栈中元素个数函数

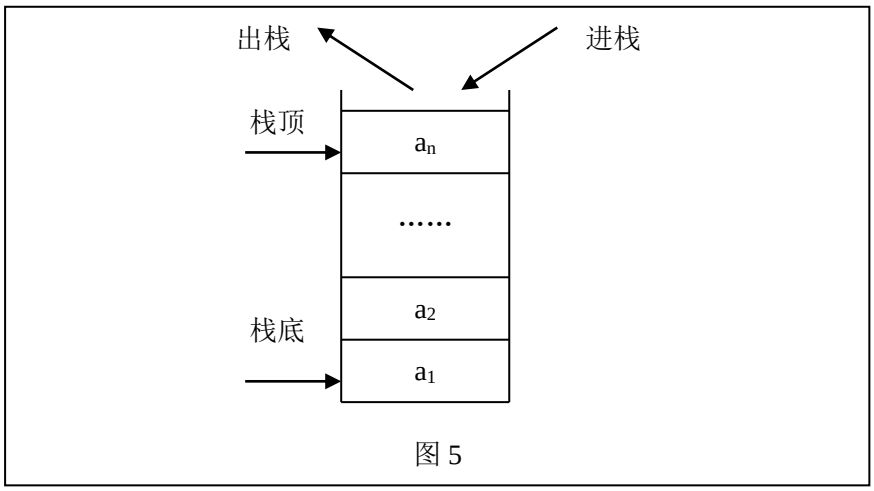


图 5

可以很容易证明,栈的各项基本操作的时间复杂度均为 O(1).

三、　队列

和栈相反，队列是一种先进先出（First In First Out，缩写为 FIFO）的线性表。它只允许在表的一端进行插入，而在另一端删除元素。在队列中，允许插入的一端叫做队尾，允许删除的一端则称为队头。 图 6 为队列的示意图。

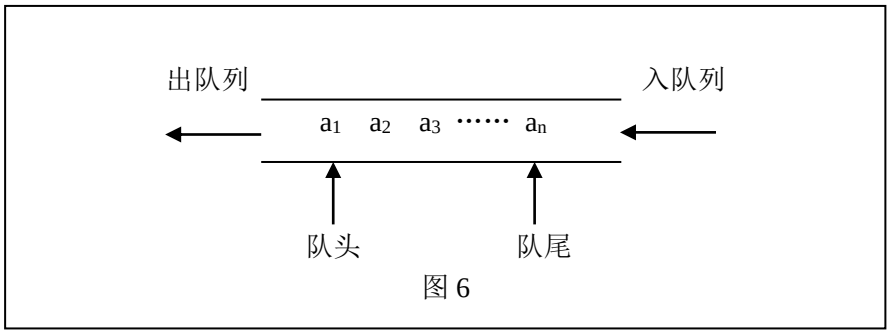


图 6

队列的基本操作有如下几种：
INIQUEUE(Q)　　初始化操作
EMPTY(Q)　　判空函数
ENQUEUE(Q,x)　入队列操作
DLQUEUE(Q)　　出队列函数
GETHEAD(Q)　　取队头元素函数
CLEAR(Q)　　队列置空操作
CURRENT_SIZE(Q)　求已知队列 Q 当前所含元素个数

可以很容易证明，队列的各项基本操作的时间复杂度均为 O(1).

除了栈和队列之外，还有一种限定性数据结构是双端队列。在双端队列中，每个端点都可以只输出、只输入或同时输出输入。因此前面所介绍的队列可以看成是双端队列的一种特殊情况。

除了上面介绍的数据结构以外，基本数据结构还包括串、有根树的存储结构等。限于文章篇幅，这里不进行具体介绍，而将重点阐述栈、队列以及线性表的应用。

第二部分——基本数据结构的应用

一、栈的应用

例1　　求 01 矩阵中最大的全零矩形[1]

给定一个 M*N 的 01 矩阵，求其中全部由 0 组成的面积最大的矩形（输出面积即可）。

分析：

这道题目的描述非常简单。我们可以很容易设计出一个简单易行的方法，即枚举矩形的左上角与右下角，然后逐格判断此矩形是否全部由 0 组成。这个算法的时间复杂度为 $O(M^3N^3)$，难以承受。当然，利用部分和技术可以使得判断部分的时间复杂度降为 O(1)，从而使整个算法的时间复杂度降为 $O(M^2N^2)$.

其实，时间复杂度更低的算法是存在的。在 2003 年国家集训队论文中，王知昆同学曾经对这个问题进行过深入研究，并且得到了时间复杂度为 O(MN)的优秀算法，其本质为动态规划算法。由于不是本文讨论的重点，这里不再赘述，请有兴趣的同学参见这篇论文（参考文献[3]）。

根据王知昆同学论文中的理论，一个极大矩形（即无法继续扩大的全 0 矩形）中一定存在着一条竖直的“悬线”（即从一点开始不断向上扩展，直到遇到 1 或者整个矩阵边界为止所形成的竖线），并且这个矩形就是由这条“悬线”不断向左向右扩展得到。下文中的算法依然要用到这条性质。

首先，悬线的长度可以很容易用动态规划算法得到。设方格(x,y)的悬线长度为 h(x,y)，则 h(x,y)的递推式为：

$$h(x,y)=\begin{cases}0, & \text{方格}(x,y)\text{上是 1，}\\ 1, & \text{方格}(x,y)\text{上是 0 且 } x=1,\\ h(x-1,y)+1, & \text{方格}(x,y)\text{上是 0 且 } x>1.\end{cases}$$

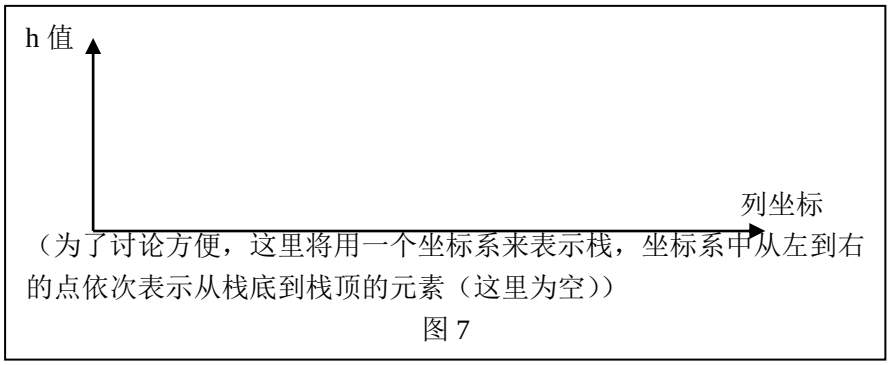
这个递推式的正确性是十分显然的，这里就不加证明了。

现在的任务是，对于矩阵的每一行，在求出这行中所有 h(x,y)的值后，需要得到每一条悬线能够最多往左往右各延伸多长的距离而不碰到障碍“1”（这里用 left(x,y)记录(x,y)格上的悬线能够向左延伸的最长距离，用 right(x,y)记录能够延伸的最长距离）. 与王知昆同学论文中的动态规划算法不同，这里将运用一种基本数据结构——栈来解决问题。

[1] 经典问题

首先，为了描述算法的方便以及去除对特殊情况处理的考虑，我们可以在每一行的末尾添加一个哨兵值“-1”，即令 h(x,n+1)=-1，从后文中可以体现出这个哨兵的作用。

在处理每一行之前，先将栈置空（如图 7）：



（为了讨论方便，这里将用一个坐标系来表示栈，坐标系中从左到右的点依次表示从栈底到栈顶的元素（这里为空））

图 7

接着，从 h(x,1)开始，依次向栈里插入值。但是与简单的入栈不同，这里必须考虑 h(x,i)与栈里元素之间的关系。

设栈顶元素的值为 A，对应的列坐标值为 j：

如果栈为空或待插入的 h(x,i)>A，则将 h(x,i)插入栈，成为新的栈顶元素（图 8）；

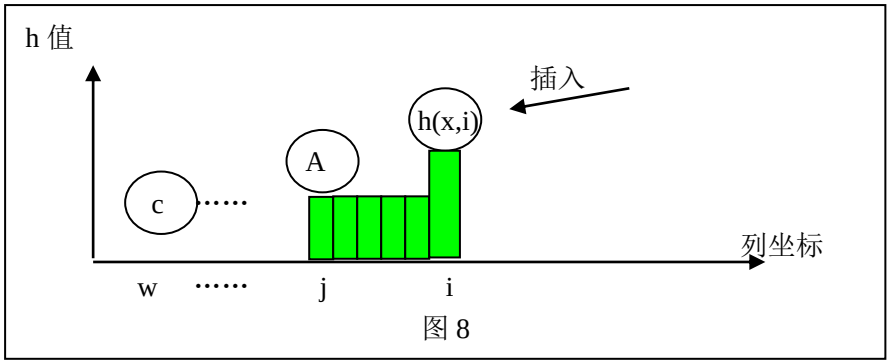


图 8

如果 h(x,i)=A，left(x,i)还不能马上确定。但是由于 h(x,i)=A，所以第 j 列的悬线向左向右扩展所得出的矩形与第 i 列的悬线向左向右扩展所得出的矩形完全一样。又由于 j 表示这条悬线最左能够到达的位置，所以此时不对栈做任何改动（图 9）；

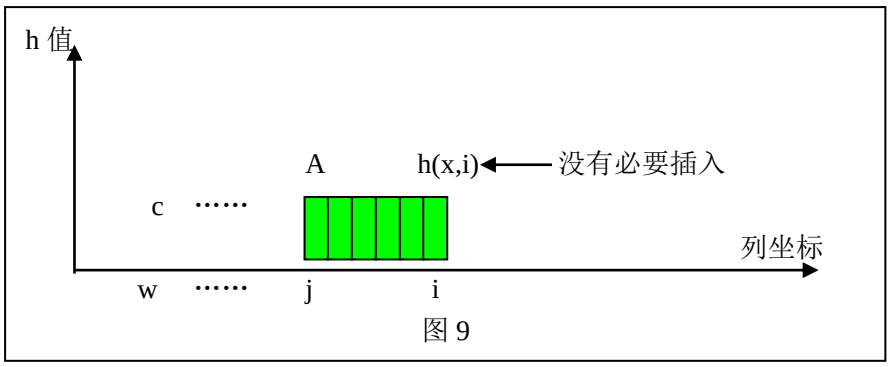


图 9

如果 h(x,i)<A，left(x,i)还不能马上确定。我们需要不断弹出栈顶元素（因为这些比 h(x,i)高的悬线是不可能再向第 i 列后面延伸了，后面比 h(x,i)高的悬线也不可能向第 i 列前面延伸了）。在弹元素出栈的过程中，对于每一个出栈的元素 B，设它的列坐标值为 j，则此时可以确定它对应的悬线最多向右扩展(i-j)格，而且可以用 B*(i-j)与已得最大全 0 矩形面积比较。弹出的过程一直进行到栈为空或者栈顶元素的值 B≤h(x,i). 如果

B=h(x,i)，则按照“h(x,i)=A”的情况进行处理（即不做改动）（图 10）。否则，由于我们至少弹出了一个元素，所以一定有一个最后弹出的元素，设它的值为 C，列坐标值为 j. 显然，(x,i)格上的悬线能够一直往左扩展到第 j 列。此时将 h(x,i)插入栈，但是需要设它的列坐标值为 j（而不是 i），表示它能够一直往左扩展到 j（图 11）.

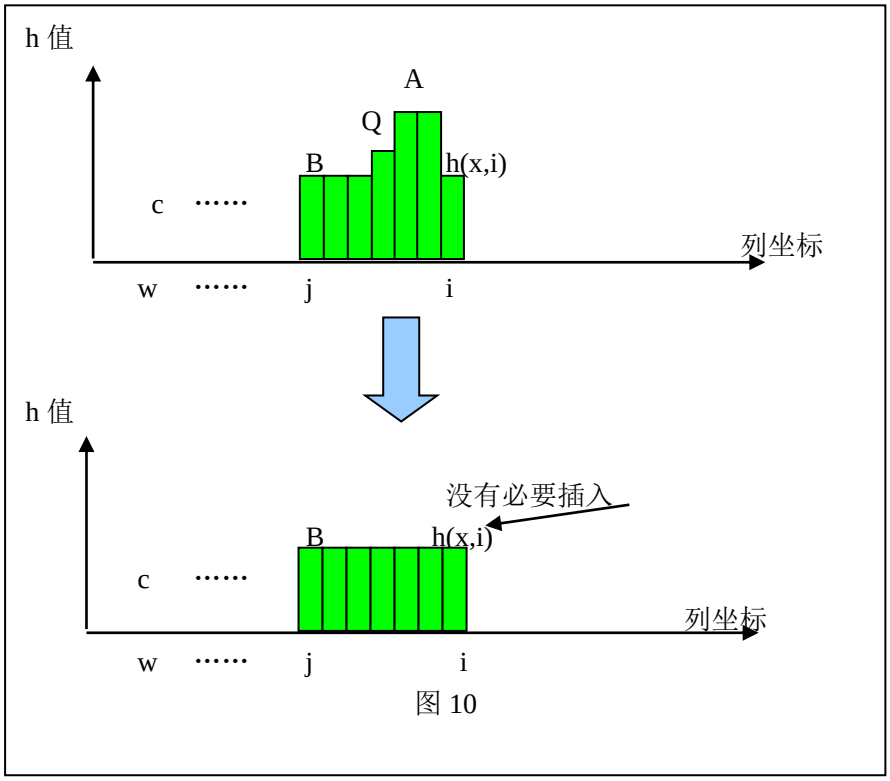


图 10

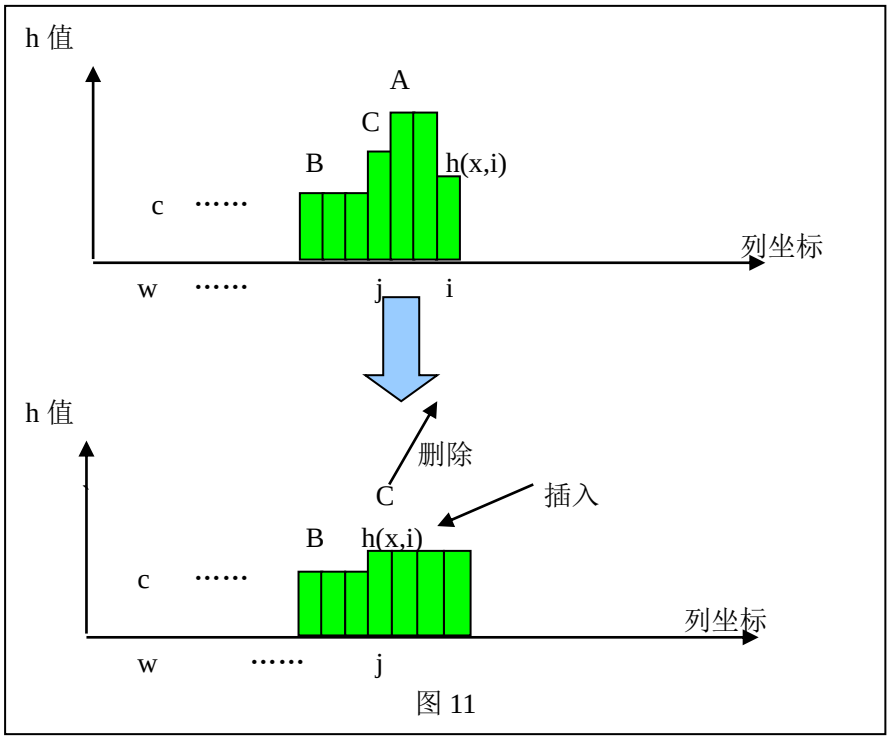


图 11

可以看出，在栈中，一个元素的列坐标值实际上表示以这个元素的值为高的悬线最左能够延伸到的位置。它被弹出的时刻也就是它向右延伸“碰壁”的时刻。

由于在行尾有一个哨兵 h(x,n+1)=-1，所以最终栈中非哨兵元素会被全部弹出，从而不会遗漏任何一个值。

算法的伪代码如下：

```
maxarea=0;
for (x=1; x<=m; x++)
{
    计算 h(x,i)，设哨兵 h(x,n+1)=-1;
    设栈指针为 0;
    for (i=1; i<=n+1; i++)
    {
        (设栈顶元素为 A，对应列坐标为 j)
        if ((栈为空) or (h(x,i)>A))
          插入 h(x,i); continue;
        if (h(x,i)=A)
          continue;
        while (h(x,i)<A)
          {
            area=A*(i-j);
            if (area>maxarea)
              then maxarea=area;
            从栈中删除 A;
          }
        if (A==h(x,i))
          continue;
        设最后删除的元素为 C，对应列坐标为 j
        插入元素 h(x,i)，列坐标设为 j
    }
}
```

算法 1

这个算法的时间复杂度是 O(MN)，空间复杂度可以用滚动数组降为 O(N)，是一个十分优秀的算法。从解题过程中可以看出，基本数据结构之一的栈发挥了很重要的作用。整个算法都是围绕栈展开的。但是，与普通的栈不同，这个算法中的栈需要考虑到元素之间的大小关系，还要记录相应列坐标的值。可以很容易证明，算法 1 中的栈任何时候都满足从栈底到栈顶的元素组成一个严格递增的序列。这便是与一般的栈的不同之处。

通过巧妙地利用与改造栈，我们成功地解决了这道例题。下面将通过例 2 来体现另一种重要的基本数据结构——线性表的应用。

二、线性表的应用

例2　　营业额统计[2]

给定 N（1≤N≤32767）天的营业额 $a_1,a_2,a_3,\cdots\cdots a_n$. 定义最小波动值：

该天的最小波动值=min{|该天以前某一天的营业额-该天营业额|}. 特别地，第一天的最小波动值即为 a_1. 试求 N 天的最小波动值之和。

分析：

这道题目的规模很大，如果简单地用两层循环解决，时间复杂度高达 $O(N^2)$，难以在时限内出解。算法低效的原因在于没有高效地将数据组织起来，而是松散地存储在数组中，导致对于每一个营业额都需要检查前面所有的营业额。实际上，有一种高级数据结构——平衡树[3]可以解决这个问题。如图 12 所示，我们可以在将一天的营业额插入平衡树的过程中得到该天的最小波动值。方法是求出所有在插入路径上的数字与改天营业额差的绝对值，从中取出最小值（如图中取 min{|5-7|,|8-7|,|6-7|}=1）。

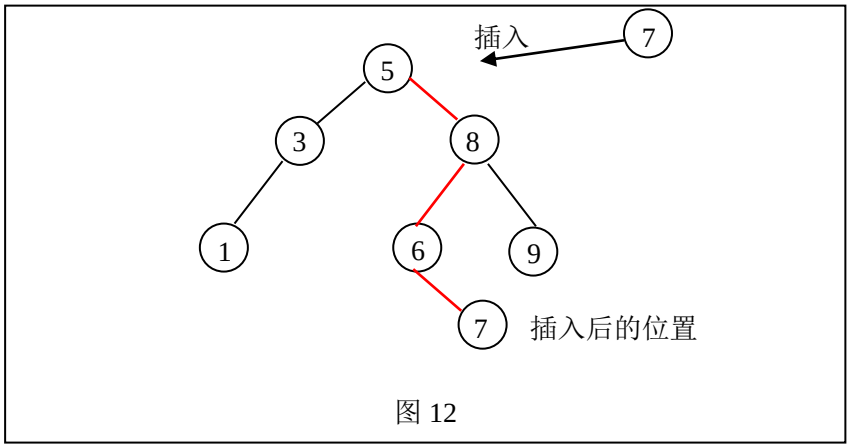


图 12

这样，应用平衡树可以在 $O(N\log_2 N)$时间内得到问题的解。

然而，尽管平衡树十分高效，但是它的编程复杂度却非常高。各种左旋、右旋甚至双旋都比较复杂，稍不注意就可能出错，导致整个程序的失败。这使得我们在紧张的信息学比赛中必须谨慎使用这样“高效+高出错率”的数据结构。那么，基本数据结构能否在这里得到应用呢？是的。

首先，将这 N 个元素进行排序，同时记录下原来第 i 个元素在排序后的位置 b_i. 接着，将排序后的序列建成一个双向链表。然后，按照从 a_n 到 a_1 的顺序依次处理每个元素。对于 a_n，由于知道它在排序后的序列中的位置 b_n，可以查看 b_n 的前驱 $pre[b_n]$ 与后继 $next[b_n]$所指的数。很显然，由于数据在双向链表中是有序的，所以最小波动值必然是 a_n 与这两个数中的某一个的差的绝对值。当然，如果前驱或后继为空，就不需要考虑相应的位置了。处理完 a_n 后，我们把它从双向链表中删除（由前文可知，这一步操作的时间复杂度仅为 O(1)），接着处理 a_{n-1}. 这样，每当处理 a_i 时，a_{i+1}，

[2] 湖南省 2002 年省队选拔赛试题

[3] 平衡树本质上是一棵二叉排序树，但是它的各项基本操作（如取最值、插入、删除等）的时间复杂度为 $O(\log_2 N)$或平摊为 $O(\log_2 N)$. 关于平衡树的介绍详见参考书目[2].

a_{i+2}，……， a_n 已被从双向链表中删除，而此时 b_i 的前驱与后继所指的数就等于将 a_1，a_2，……， a_i 排序后 a_i 的前驱与后继。整个算法分为排序，建表与处理三部分。很显然，建表与处理的时间复杂度均为 O(N)，所以此算法的时间复杂度与排序的时间复杂度同阶，为 $O(N\log_2 N)$.

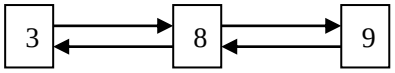
图 13 为对一组数据处理的图示。

原数据：a_1=9　a_2=3　a_3=8

排序后：3,8,9

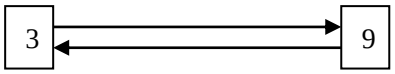
　　b_1=3　b_2=1　b_3=2

双向链表



处理 a_3：b_3=2，前驱为 1，后继为 3，最小波动值为 min{|8-3|,|8-9|}=1

删除 a_3 后：



处理 a_2：b_2=1，无前驱，后继为 3，最小波动值为 min{|9-3|}=6

删除 a_2 后：

处理 a_1：根据定义，最小波动值即为 9

图 13

相应的伪代码：

1、输入 N，a_1，a_2，a_3，……，a_n

2、将 a_1，a_2，a_3，……，a_n 按照从小到大的顺序排序，得到序列 c_1，c_2，c_3，……，c_n，并记录下每个 a_i 在新序列中的位置 b_i

3、在新的序列上建立双向链表

4、按照从 a_n 到 a_1 的顺序依次处理每个元素：

```
result=0;
for (i=n; i>=2; i--)
{
  result+=min{|ai-c pre[bi] |,|ai- c next[bi] |};
（如果 pre[bi]=0 或 next[bi]=0 则不予考虑）
}
result+=a1;
```

5、输出 result

算法 2

在本题中，平衡树可以高效地得出答案，但是由于其很高的编程复杂度，导致我们在

比赛中难以保证程序的正确性。而之后运用双向链表的方法十分简洁，既没有增加算法的时间复杂度，又大大降低了编程复杂度，实为比赛时的首选，并且其清晰的思路与独到的见解也值得学习与推广。

以上通过两道例题简单介绍了两种基本数据结构——栈与线性表的应用。下文中的例 3 将展示另一种基本数据结构——队列的应用。

三、队列的应用

例 3　　瑰丽华尔兹[4]

给定一个 N 行 M 列的矩阵，矩阵中的某些方格上有障碍物。有一个人从矩阵中的某个方格开始滑行。每次滑行都是向一个方向最多连续前进 c 格（也可以原地不动）（两次滑行的 c 值不一定相同）。但是这个人在滑行中不能碰到障碍物。现按顺序给出 K 次滑行的方向（东、南、西、北中的一个）以及对应的 c，试求这个人能够滑行的最长距离（即格子数）。

数据范围：1≤N,M≤200，K≤200，$\sum_{i=1}^{k} c_i \le 40000$

分析：

本题是一个求最值的问题。根据题目中 K 次滑行的有序性以及数据范围，可以很容易设计出这样一种动态规划算法：

令 f(k,x,y)=此人 k 次滑行后到达(x,y)方格时已经滑行的最长距离。动态规划的状态转移方程如下（以下仅给出向东滑行的状态转移方程，其他 3 个方向上的转移方程可以类似地推出）：

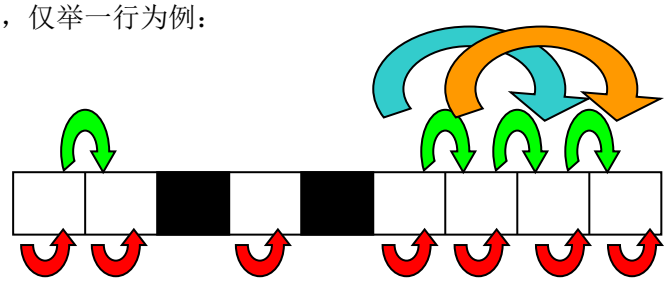
f(0,startx,starty)=0

f(k,x,y)=max{f(k-1,x,y),f(k-1,x,y-1)+1,f(k-1,x,y-2)+2,……,f(k-1,x,y')+y-y'}

（其中 y'为满足 y=1 或(x,y'-1)上有障碍或 y'=y-c_k 的最大值）

从图 14 中可以很清楚地看出动态规划转移的条件：

令 c_k=2，仅举一行为例：



其中，黑色方格表示障碍，每个箭头表示发出箭头的状态可以去改进被箭头指向的状态（当然，这两个状态的 k 值相差 1）。

图 14

现在来分析这个动态规划算法的时间复杂度。

显然，状态总数为 O(KMN)，而每次状态转移在最坏情况下的时间复杂度为 O(max{M,N})，因此总的时间复杂度为 O(KMN*max{M,N})=O($1.6*10^9$)，难以承受。因

[4] NOI2005 Day1 adv1900

此，我们需要对这个动态规划算法进行优化。

先不考虑障碍，可以看出，在求 f(k,x,y)与 f(k,x,y+1)时，很多状态我们都重复考虑了。以图 15 为例，求 f(k,1,3)与 f(k,1,4)时都用到了 f(k-1,1,2)和 f(k-1,1,3). 在先前介绍的动态规划算法中，这样的重复大量出现，导致算法的低效。那么，如何才能有效地防止这类重复计算的发生呢？让我们先来研究动态规划方程：

f(k,x,y)=max{f(k-1,x,y),f(k-1,x,y-1)+1,f(k-1,x,y-2)+2,……,f(k-1,x,y')+y-y'}

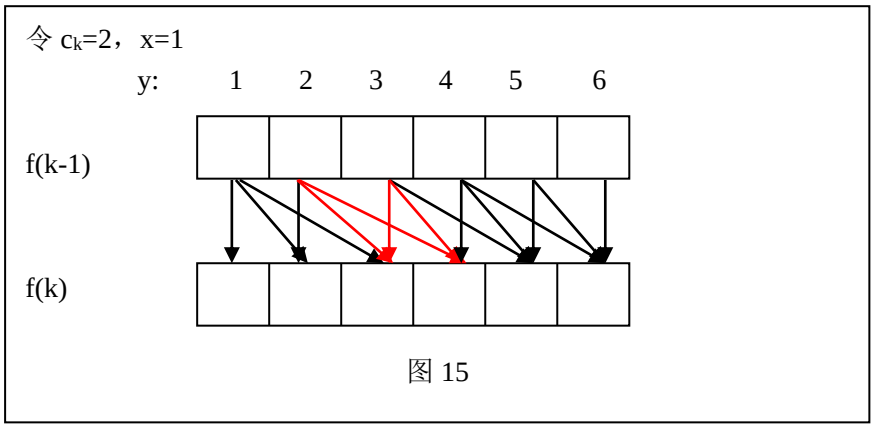


图 15

对于一个具体的例子 k=2，x=1，c_2=2 可以列出如下等式：

f(2,1,1)=max{f(1,1,1)}

f(2,1,2)=max{f(1,1,1)+1,f(1,1,2)}

f(2,1,3)=max{f(1,1,1)+2,f(1,1,2)+1,f(1,1,3)}

f(2,1,4)=max{f(1,1,2)+2,f(1,1,3)+1,f(1,1,4)}

……

如果我们定义一个序列 a，使得 a_i=f(1,1,i)-i+1，则以上等式可以写成：

f(2,1,1)=max{a_1}=max{a_1}

f(2,1,2)=max{a_1+1,a_2+1}=max{a_1,a_2}+1

f(2,1,3)=max{a_1+2,a_2+2,a_3+2}=max{a_1,a_2,a_3}+2

f(2,1,4)=max{a_1+3,a_2+3,a_3+3,a_4+3}=max{a_2,a_3,a_4}+3

……

显然，在应用了 a 序列之后，我们就可以只关注 a 序列而不必为每个 a_i 加上一个不同的值，从而简化了操作。

现在，我们可以加入对障碍物的考虑。例如对于图 15 中的状态，我们依次要求 max{a_1},max{a_1,a_2}，max{a_4}，max{a_6}，max{a_6,a_7}，max{a_6,a_7,a_8}，max{a_7,a_8,a_9}. 考虑 max 函数中的序列，可以发现，每次都是在序列的尾部添上一个 a 值（遇到障碍物除外），并有时在头部删去一些 a 值（如果区间长度超过 c_k+1，就删去 1 个 a 值；如果遇到一个障碍物，则清空整个序列），而且这个序列中 a 的下标一定是连续的。有了这些条件与限制，就可以运用一种专门计算区间最值的数据结构——线段树[5]。每次根据 a 值建立一棵线段树，然后对于需要求最大值的区间，直接在线段树中查找。对于一行来说，建立线段树的时间复杂度为 O(M)，每次查找的时间复杂度为 O($\log_2$M)，而对于前文所说的区间处理，由于每个 a 值最多进入区间一次，被删除一次，所以维护区间的总的时间复杂度为 O(M). 这样，整个算法的时间复杂度降为 O(KMN$\log_2$(max{m,n})).

[5] 线段树是一棵以线段作为基本单位的二叉树，在线段树中进行的区间插入、删除以及询问的时间复杂度均为 O($\log_2$N). 关于线段树的介绍详见参考书目[2].

然而，线段树的编程复杂度较高，初始化、插入、查找分别需要一个子过程，容易出错。并且 $O(KMN\log_2(\max\{m,n\}))$的复杂度再乘以线段树操作中的系数，还是比较慢，不能令人满意。实际上，有一种基本数据结构——队列可以非常好地解决这个问题。

前面已经对于序列的插入与删除进行过讨论。其中只在一端插入，另一端删除的特性恰好符合队列的性质。但是，这里是要求队列中所有数的最大值，普通的队列可以胜任这个操作吗？让我们首先来分析一下如何存储队列中的数。

如图 16 所示，对于已经出现在队列中的 a_2 与 a_3，如果 $a_2 \le a_3$，则 a_2 是没有必要出现在队列中的。因为根据队列的插入与删除原则可以推导出，如果队列中已经出现 a_3 了，则在 a_2 被删除之前，a_3 是一定不会被删除的。因此，a_2 与 a_3 会一直同时出现在队列中，直至 a_2 被删除。但是 $a_2 \le a_3$，因此队列中的最大值永远不会是 a_2，也就没有必要存储 a_2.

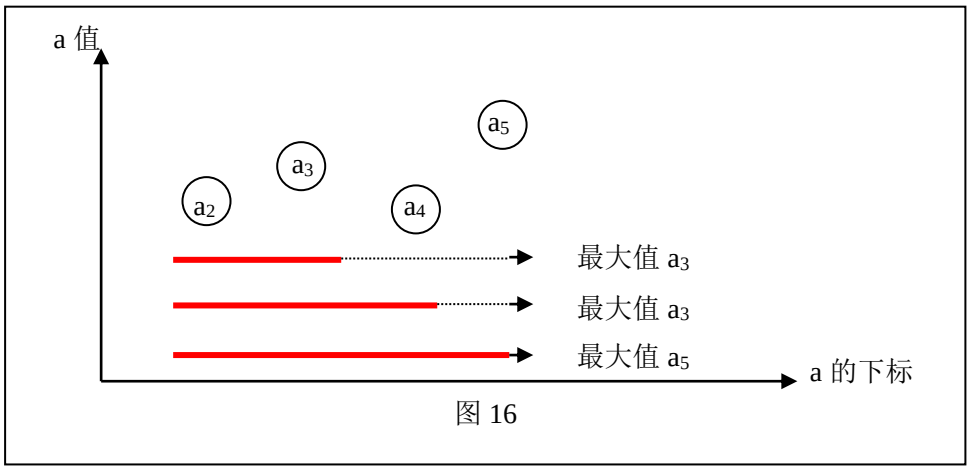


图 16

根据这一条重要的性质，可以立刻推导出“有必要”存储在队列中的 a 值的大小关系——严格递减。也就是说，存储在队列中的 a 值是依次减小的，而队头元素的值为最大值，也就是当前队列中所有数字（无论是否存储在队列中）的最大值。这样，每次可以取出位于队头的 a 值作为最大值。但是一个新的问题摆在我们面前——如何实时维护队列，即如何正确地插入或删除元素。

首先研究删除操作。很显然，如果队头 a 值的下标对应的方格与当前处理的方格之间的距离已经大于 c_k，则直接将它从队列中删除，即队头指针加一。

接着是插入操作。根据前文中对于队列中元素大小关系的讨论可以得知，插入一个元素 a_i 后，队列中不能有元素 a_j 满足 $a_j \le a_i$. 于是，我们可以从队尾开始，依次删除掉不大于 a_i 的 a 值，直到队列中剩下的元素都大于 a_i. 此时就可以将 a_i 插入队尾。（插入与删除过程见图 17）。

很显然，每次求队列中所有元素的最大值可以直接查看队头元素。根据此队列性质，队头元素一定是队列中所有元素里最大的一个。

由于在一行中，一个元素最多被插入一次，删除一次，所以插入与删除的总时间复杂度为 $O(\max\{M,N\})$，而每次询问最大值的时间复杂度仅为 $O(1)$. 综上所述，此算法的时间复杂度为 $O(KMN)$，是一个十分优秀的算法。

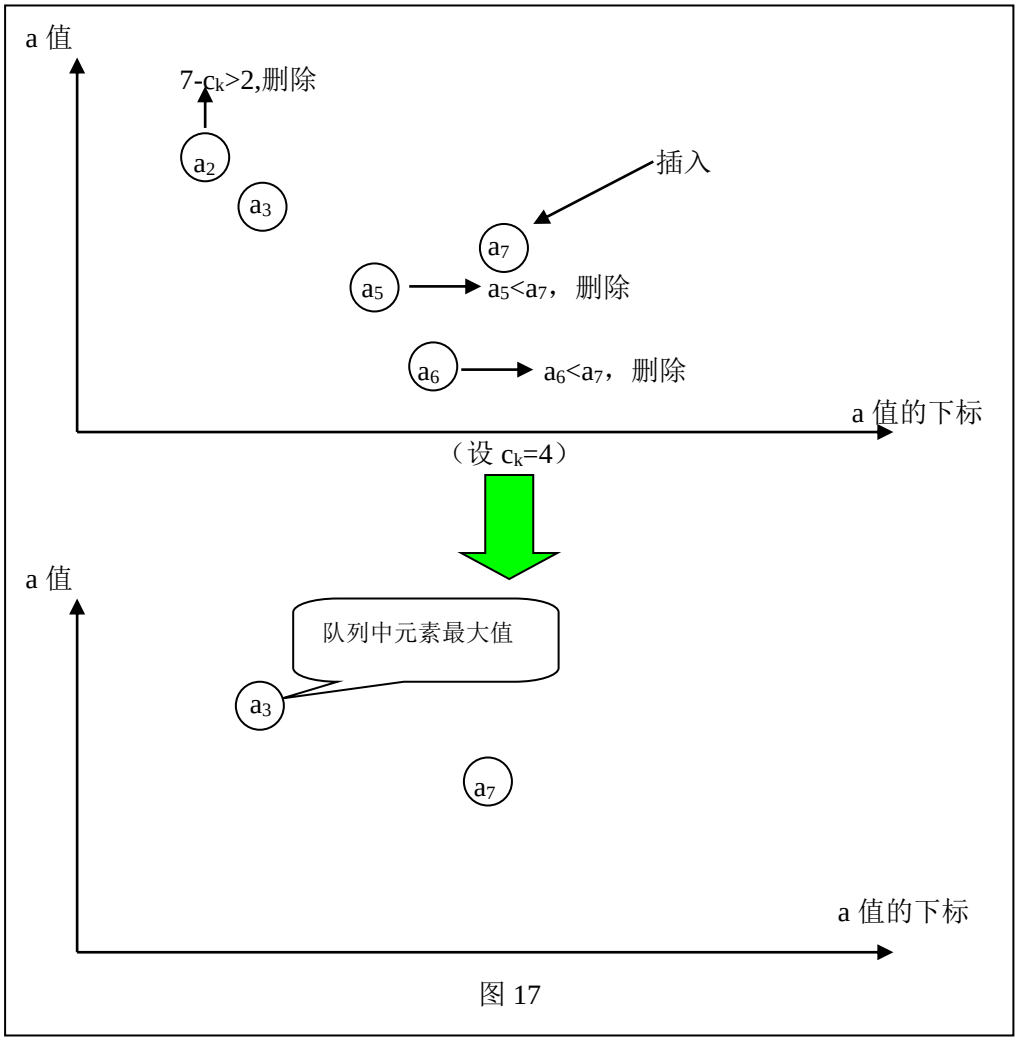


图 17

相应的伪代码（见下页）：

```
(以下仅考虑向东滑行的情况，向其他方向滑行的情况可以类似推出)
for (x=1; x<=n; x++)
  for (y=1; y<=m; y++)
    f[0][x][y]=-∞;
f[0][startx][starty]=0;
for (k=1; k<=slidenum; k++)
  for (x=1; x<=n; x++)
  {
    head=tail=0;
    for (y=1; y<=m; y++)
    {
      if (方格(x,y)上有障碍)
        {f[k][x][y]=-∞;    head=tail=0;    continue;}
      if (queue[head]<y-c[k])
        head++;
      while ((tail>head)&&(f[k-1][x][queue[tail].num]-queue[tail].num+1<=f[k-1][x][y]-y+1))
        tail--;
      queue[++tail]=y;
      f[k][x][y]=f[k-1][x][queue[head].num]+y-queue[head].num;
    }
  }
return max{f[slidenum][1][1], f[slidenum][1][2],……,f[slidenum][n][m]};
```

算法 3

回顾整个过程，我们首先找到了一个动态规划的解法。但是由于每次转移的时间复杂度太高，使得我们必须减少冗余运算。文中提到的线段树是一个不错的解决方案，但是其编程复杂度与时间复杂度都还不能令人十分满意。而经常被我们所忽略的基本数据结构——队列却在这里发挥了巨大的作用。由前文可以看出，运用队列的解法巧妙而简洁，不但降低了时间复杂度，还大大减少了由于程序复杂而导致编程错误的可能性，很好地解决了这个问题。这再一次说明，在新趋势下的信息学竞赛中，基本数据结构的作用没有减小，而是变得更加重要了。在我们找到一种好算法而畏于其复杂的程序实现时，不妨转换思路，尝试用基本数据结构加以解决，说不定会有事半功倍的效果。

总结

本文简要介绍了基本数据结构及其在信息学竞赛中的应用。实际上，基本数据结构在许多领域都有其建树。限于篇幅，本文难以涵盖基本数据结构灵活多变的应用之万一。基本数据结构之所以成为经典，其根本在于它易于实现，普遍适用。基本数据结构是数据结构中的精华，它的理论已经经过数十年时间的检验，应用于算法中的诸多方面。其实，基本数据结构的应用还远远没有被研究彻底。在计算机科学不断发展的今天，不时有它的身影出现，并巧妙地运用它解决了诸多问题。在未来的计算机科学研究中，一定会有越来越多现在还不为人所知的基本数据结构的应用被发现和研究。

其实，在思想上，基本数据结构的应用也给我们带来了很多的启示。我们在学习信息学竞赛相关知识时，首先对于基本数据结构进行了一定的了解。在熟练掌握这些基本知识后，又开始学习各种高级数据结构。现在对于基本数据结构作用的重新发现与挖掘，并不是一种简单的回归，更不是对于高级数据结构的摒弃。从技术层面上来说，基本数据结构并不是万能的，它也存在着自己的局限性。因此，在有些情况下，我们必须借助于专门设计的数据结构（如平衡树、线段树等）。但是，基本数据结构的确能在很多情况下帮助我们巧妙地解决问题。因此，灵活掌握基本数据结构及其操作并对某些高级数据结构有所了解才能更好地帮助我们，使我们在面对问题时能够做到以不变应万变，从而更有效地解决问题。而这体现了基本数据结构与高级数据结构之间的**辨证关系**。

更重要的是，对于基本数据结构的应用并不是以原先的理解解决新的问题，而是站在一个更高的高度对其进行研究。在整个过程中，我们经历了从基本数据结构到高级数据结构再到基本数据结构的过程。而这个过程并不是一种简单的回归，而是一种**螺旋式发展**。从表面上看，我们又回到了基本数据结构的阶段。但事实上，对于基本数据结构的再次运用是基于对整个数据结构知识的深刻领悟。在对于基本数据结构进一步的理解与运用中，我们的知识与思想也得到了升华。

参考文献

[1] 严蔚敏 吴伟民 ，1992，《数据结构（第二版）》。 北京：清华大学出版社

[2] Thomas H.Cormen Charles E.Leiserson Ronald L.Rivest Clifford Stein, 2001 ，《Introduction to Algorithms, Second Edition》. The MIT Press.

[3] 王知昆 IOI2003 国家集训队论文 《浅谈用极大化思想解决最大子矩形问题》

感谢

衷心感谢江涛老师在论文创作过程给予笔者的指导和帮助。

衷心感谢刘汝佳教练与许智磊同学给笔者提出了重要的意见与建议。

附录

一、文中例题的原题

1、例 1 原题[6]

Big Barn[7]

Farmer John wants to place a big square barn on his square farm. He hates to cut down trees on his farm and wants to find a location for his barn that enables him to build it only on land that is already clear of trees. For our purposes, his land is divided

[6] 这里采用一道类似于经典题目的例题

[7] USACO Training 1.5.4

into N x N parcels. The input contains a list of parcels that contain trees. Your job is to determine and report the largest possible square barn that can be placed on his land without having to clear away trees. The barn sides must be parallel to the horizontal or vertical axis.

PROGRAM NAME: bigbrn

INPUT FORMAT

Line 1:	Two integers: N (1 <= N <= 1000), the number of parcels on a side, and T (1 <= T <= 10,000) the number of parcels with trees
Lines 2..T+1:	Two integers (1 <= each integer <= N), the row and column of a tree parcel

SAMPLE INPUT (file bigbrn.in)
8 3
2 2
2 6
6 3

OUTPUT FORMAT
The output file should consist of exactly one line, the maximum side length of John's barn.

SAMPLE OUTPUT (file bigbrn.out)
5

2、例 2 原题[8]

营业额统计

(turnover.exe)

Tiger 最近被公司升任为营业部经理，他上任后接受公司交给的第一项任务便是统计并分析公司成立以来的营业情况。

Tiger 拿出了公司的账本，账本上记录了公司成立以来每天的营业额。分析营业情况是一项相当复杂的工作。由于节假日，大减价或者是其他情况的时候，营业额会出现一定的波动，当然一定的波动是能够接受的，但是在某些时候营业额突变得很高或是很低，这就证明公司此时的经营状况出现了问题。经济管理学上定义了一种**最小波动值**来衡量这种情况：

$$\text{该天的最小波动值} = \min\{\ |\text{该天以前某一天的营业额} - \text{该天营业额}|\ \}$$

当最小波动值越大时，就说明营业情况越不稳定。

而分析整个公司的从成立到现在营业情况是否稳定，只需要把每一天的最小波动值加起来就可以了。你的任务就是编写一个程序帮助 Tiger 来计算这一个值。

[8]湖南省 2002 年省队选拔赛试题

第一天的最小波动值为第一天的营业额。

- **输入输出要求**

输入由文件'turnover.in'读入。

第一行为正整数 $n(n \le 32767)$，表示该公司从成立一直到现在的天数，接下来的 n 行每行有一个正整数 $a_i(a_i \le 1000000)$，表示第 i 天公司的营业额。

输出到文件'turnover.out'。

输出文件仅有一个正整数，即 $\sum$每一天的最小波动值。结果小于 2^{31}。

- **输入输出样例**

Turnover.in	Turnover.out
6 5 1 2 5 4 6	12

结果说明：5+|1-5|+|2-1|+|5-5|+|4-5|+|6-5|=5+4+1+0+1+1=12

3、例 3 原题[9]

瑰丽华尔兹

主文件名：adv1900

时限：1s

【任务描述】

你跳过华尔兹吗？当音乐响起，当你随着旋律滑动舞步，是不是有一种漫步仙境的惬意？众所周知，跳华尔兹时，最重要的是有好的音乐。但是很少有几个人知道，世界上最伟大的钢琴家一生都漂泊在大海上，他的名字叫丹尼?布德曼?T.D.?柠檬?1900，朋友们都叫他 1900。

1900 出生于 20 世纪的第一年出生在往返于欧美的邮轮弗吉尼亚号上，然后就被抛弃了。1900 刚出生就成了孤儿，孤独的成长在弗吉尼亚号上，从未离开过这个摇晃的世界；也许是对他命运的补偿，上帝派可爱的小天使艾米丽照顾他。

可能是天使的点化，1900 拥有不可思议的钢琴天赋，从未有人教，从没看过乐谱，但他却能凭着自己的感觉弹出最沁人心脾的旋律。当 1900 的音乐获得邮轮上所有人的欢迎时，他才 8 岁，而此时他已经乘着海轮往返欧美 50 多次了。

虽说是钢琴奇才，但 1900 还是个 8 岁的孩子，他有着和一般男孩一样的好奇的调皮，不过可能更有一层浪漫的色彩罢了：

这是一个风雨交加的夜晚，海风卷起层层巨浪拍打着弗吉尼亚号，邮轮随着巨浪剧烈的摇摆。船上的新萨克斯手迈克斯?托尼晕船了，1900 将他邀请到舞厅，然后——，然后松开了固定

[9] NOI2005 Day1 adv1900

钢琴的闸，于是，钢琴随着海轮的倾斜滑动起来。准确的说，我们的主角 1900、钢琴、邮轮随着 1900 的旋律一起跳起了华尔兹，所有的事物好像都化为一体，随着“强弱弱”的节奏，托尼的晕船症也奇迹般地一点一点恢复。正如托尼在回忆录上这样写道：

大海摇晃着我们
使我们转来转去
快速的掠过灯和家具
我意识到我们正在和大海一起跳舞
真是完美而疯狂的舞者
晚上在金色的地板上快乐的跳着华尔兹是不是很惬意呢？也许，我们忘记了一个人，那就是艾米丽，她可没闲着：她必须在适当的时候施魔法帮助 1900，不让钢琴碰上舞厅里的家具。而艾米丽还小，她无法施展魔法改变钢琴的运动方向或速度，而只能让钢琴停一下。

不妨认为舞厅是一个 N 行 M 列的矩阵，矩阵中的某些方格上堆放了一些家具，其他的则是空地。钢琴可以在空地上滑动，但不能撞上家具或滑出舞厅，否则会损坏钢琴和家具，引来难缠的船长。
每个时刻，钢琴都会随着船体倾斜的方向向相邻的方格滑动一格，其中相邻的方格可以是向东、向西、向南或向北的。而艾米丽可以选择施魔法或不施魔法，如果不施魔法，则钢琴会滑动，而如果施魔法，则钢琴会原地不动。
艾米丽是个天使，她知道每段时间的船体的倾斜情况。她想使钢琴尽量长时间在舞厅里滑行，这样 1900 会非常高兴，同时也有利于治疗托尼的晕船。但艾米丽还太小，不会算，所以希望你能帮助她。
【输入格式】
输入文件的第一行包含 5 个数 N, M, x, y 和 K。N 和 M 描述舞厅的大小，x 和 y 为在第 1 时刻初钢琴的位置（x 行 y 列）；我们对船体倾斜情况是按时间的区间来描述的，比如“在[1, 3]时间里向东倾斜，[4, 5]时间里向北倾斜”，因此这里的 K 表示区间的数目。
以下 N 行，每行 M 个字符，描述舞厅里的家具。第 i 行第 j 列的字符若为‘ . ’，则表示该位置是空地；若为‘ x ’，则表示有家具。
以下 K 行，顺序描述 K 个时间区间，格式为：s_i t_i d_i($1 \leqslant i \leqslant K$)。表示在时间区间[si, ti]内，船体都是向 di 方向倾斜的。di 为 1, 2, 3, 4 中的一个，依次表示北、南、西、东（分别对应矩阵中的上、下、左、右）。输入保证区间是连续的，即

$s_1 = 1$

$t_i = s_{i-1} + 1 \quad (1 < i \leqslant K)$

$t_K = T$

【输出格式】
输出文件仅有 1 行，包含一个整数，表示钢琴滑行的最长距离(即格子数)。
【输入样例】
4 5 4 1 3
..xx.
.....
...x.
.....
1 3 4
4 5 1

6 7 2

【输出样例】
6

【样例说明】
钢琴的滑行路线：

□□■■□
←←↑□□
□□↑■□
→→×□□

钢琴在“×位置上时天使使用一次魔法，因此滑动总长度为 6。
【评分方法】
本题没有部分分，你的程序的输出只有和我们的答案完全一致才能获得满分，否则不得分。
【数据范围】
50%的数据中，1≤N, M≤200，T≤200；
100%的数据中，1≤N, M≤200，K≤200，T≤40000。

二、参考程序：
1、例 1 程序

```
#include <stdio.h>
#define infile "bigbrn.in"
#define outfile "bigbrn.out"
#define maxn 1010

struct Tstack{
      long v,j;
}df[maxn+1];

char a[maxn+1][maxn+1];
long h[maxn+1],
    n,dftop=0,result=0;

FILE *fin=fopen(infile,"r"),
    *fout=fopen(outfile,"w");

void init()
{
 long i,j,ge,x,y;
 fscanf(fin,"%ld%ld",&n,&ge);
 for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
```

```
   a[i][j]=0;
 for (i=1; i<=ge; i++)
  {
   fscanf(fin,"%ld%ld",&x,&y);
   a[x][y]=1;
  }
 fclose(fin);
}

long min(long a,long b)
{
 if (a<b)
  return a;
  else return b;
}

void work()
{
 long x,i,j,t;
 for (i=1; i<=n; i++)
  h[i]=0;
 result=0;
 for (x=1; x<=n; x++)
  {
   for (i=1; i<=n; i++)
    if (!a[x][i])
     h[i]++;
     else h[i]=0;
   h[n+1]=-1;    //special
   dftop=0;
   for (i=1; i<=n+1; i++)
    {
     if ((!dftop)||(h[i]>df[dftop].v))
      {
       df[++dftop].v=h[i];
       df[dftop].j=i;
       continue;
      }
     if (h[i]==df[dftop].v)
      continue;
     while ((dftop)&&(h[i]<df[dftop].v))
      {
       t=min(df[dftop].v,i-df[dftop].j);
       if (t>result)
```

```
       result=t;
      dftop--;
     }
    if ((dftop)&&(h[i]==df[dftop].v))
     continue;
    dftop++;
    df[dftop].v=h[i];
   }
  }
}

void output()
{
 fprintf(fout,"%ld\n",result);
 fclose(fout);
}

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

2、例 2 程序

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define infile "turnover.in"
#define outfile "turnover.out"
#define maxn 33000

long a[maxn+1],xl[maxn+1],
    next[maxn+1],pre[maxn+1],
    n,result=0;

FILE *fin=fopen(infile,"r"),
    *fout=fopen(outfile,"w");

void pass(long start,long stop,long &mid)
{
 long t,x;
```

```
 t=rand()%(stop-start+1)+start;
 x=xl[t];
 xl[t]=xl[start];
 while (start<stop)
  {
   while ((start<stop)&&(a[x]<a[xl[stop]]))
    stop--;
   xl[start]=xl[stop];
   if (start<stop)
    start++;
   while ((start<stop)&&(a[xl[start]]<a[x]))
    start++;
   xl[stop]=xl[start];
   if (start<stop)
    stop--;
  }
 mid=start;
 xl[start]=x;
}

void px(long start,long stop)
{
 long mid;
 if (start<stop)
  {
   pass(start,stop,mid);
   px(start,mid-1);
   px(mid+1,stop);
  }
}

void init()
{
 long i;
 fscanf(fin,"%ld",&n);
 for (i=1; i<=n; i++)
  {
   fscanf(fin,"%ld",&a[i]);
   xl[i]=i;
  }
 fclose(fin);
 srand(time(0));
 px(1,n);
 next[0]=xl[1];
```

```
 pre[0]=xl[n];
 pre[xl[1]]=next[xl[n]]=0;
 for (i=1; i<n; i++)
  next[xl[i]]=xl[i+1];
 for (i=2; i<=n; i++)
  pre[xl[i]]=xl[i-1];
}

void work()
{
 long i,j,k,min,t;
 result=a[1];
 for (i=n; i>=2; i--)
  {
   min=2147483647;
   if (pre[i])
    {
     t=labs(a[pre[i]]-a[i]);
     if (t<min)
      min=t;
    }
   if (next[i])
    {
     t=labs(a[next[i]]-a[i]);
     if (t<min)
      min=t;
    }
   result+=min;
   next[pre[i]]=next[i];
   pre[next[i]]=pre[i];
  }
}

void output()
{
 fprintf(fout,"%ld\n",result);
 fclose(fout);
}

int main()
{
 init();
 work();
 output();
```

```
 return 0;
}
```

3、例 3 程序

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define infile "adv1900.in"
#define outfile "adv1900.out"
#define maxm 210
#define maxn 210
#define maxge 210
#define minnum -1000000000

struct Txl{
      long x,y;
}xl[maxn+1];

struct Tzl{
      long start,stop,d;
}zl[maxge+1];

char map[maxm+1][maxn+1];
long f[2][maxm+1][maxn+1],
     head,tail,m,n,old,now,
     startx,starty,ge,len;

FILE *fin=fopen(infile,"r"),
     *fout=fopen(outfile,"w");

void init()
{
 long i,j;
 fscanf(fin,"%ld%ld%ld%ld%ld",&m,&n,&startx,&starty,&ge);
 for (i=1; i<=m; i++)
  {
   fscanf(fin,"%s",map[i]);
   for (j=n; j>=1; j--)
    map[i][j]=map[i][j-1];
   map[i][0]=' '; //just for looking
  }
 for (i=1; i<=ge; i++)
  fscanf(fin,"%ld%ld%ld",&zl[i].start,&zl[i].stop,&zl[i].d);
 fclose(fin);
 now=0;
```

```
 for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
   f[now][i][j]=minnum;
 f[now][startx][starty]=0;
}

inline long dist(long x1,long y1,long x2,long y2)
{
 return (labs(x1-x2)+labs(y1-y2));
}

void cl(long x,long y,long d)
{
 if (map[x][y]=='x')
  {
   head=tail+1;
   return;
  }
 if (d==1)
  {
   while ((head<=tail)&&(xl[head].x-x>len))
    head++;
   while
((head<=tail)&&(f[old][x][y]>=xl[tail].x-x+f[old][xl[tail].x][xl[tail
].y]))
    tail--;
  }
  else if (d==2)
  {
      while ((head<=tail)&&(x-xl[head].x>len))
    head++;
   while
((head<=tail)&&(f[old][x][y]>=x-xl[tail].x+f[old][xl[tail].x][xl[tail
].y]))
    tail--;
  }
  else if (d==3)
  {
   while ((head<=tail)&&(xl[head].y-y>len))
    head++;
   while
((head<=tail)&&(f[old][x][y]>=xl[tail].y-y+f[old][xl[tail].x][xl[tail
].y]))
    tail--;
```

```
  }
  else {
       while ((head<=tail)&&(y-xl[head].y>len))
        head++;
       while
((head<=tail)&&(f[old][x][y]>=y-xl[tail].y+f[old][xl[tail].x][xl[tail
].y]))
        tail--;
      }
 xl[++tail].x=x;
 xl[tail].y=y;

f[now][x][y]=f[old][xl[head].x][xl[head].y]+labs(xl[head].x-x)+labs(x
l[head].y-y);
}

void work()
{
 long i,j,k,w,x,y;
 for (w=1; w<=ge; w++)
  {
   old=now;
   now=1-now;
   len=zl[w].stop-zl[w].start+1;
   if (zl[w].d==1)
    {
     for (j=1; j<=n; j++)
      {
       head=1;
       tail=0;
       for (i=m; i>=1; i--)
        cl(i,j,1);
      }
    }
   if (zl[w].d==2)
    {
     for (j=1; j<=n; j++)
      {
       head=1;
       tail=0;
       for (i=1; i<=m; i++)
        cl(i,j,2);
      }
    }
```

```
   if (zl[w].d==3)
    {
     for (i=1; i<=m; i++)
      {
       head=1;
       tail=0;
       for (j=n; j>=1; j--)
        cl(i,j,3);
      }
    }
   if (zl[w].d==4)
    {
     for (i=1; i<=m; i++)
      {
       head=1;
       tail=0;
       for (j=1; j<=n; j++)
        cl(i,j,4);
      }
    }
  }
}

void output()
{
 long i,j,result=0;
 for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
   if ((map[i][j]=='.')&&(f[now][i][j]>result))
    result=f[now][i][j];
 fprintf(fout,"%ld\n",result);
 fclose(fout);
}

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