数据结构的在程序设计中的应用

长沙市一中　　肖洲

【关键字】 逻辑结构　存储结构　算法优化

【摘要】

数据结构作为程序设计的基础，其对算法效率的影响必然是不可忽视的。本文就如何合理选择数据结构来优化算法这一问题，对选择数据结构的原则和方法进行了一些探讨。首先对数据逻辑结构的重要性进行了分析，提出了选择逻辑结构的两个基本原则；接着又比较了顺序和链式两种存储结构的优点和缺点，并讨论了选择数据存储结构的方法；最后本文从选择数据结构的的另一角度出发，进一步探讨了如何将多种数据结构进行结合的方法。在讨论方法的同时，本文还结合实际，选用了一些较具有代表性的信息学竞赛试题举例进行了分析。

【正文】

一、引论

“数据结构＋算法＝程序”，这就说明程序设计的实质就是对确定的问题选择一种合适的数据结构，加上设计一种好的算法。由此可见，数据结构在程序设计中有着十分重要的地位。

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。因为这其中的“关系”，指的是数据元素之间的逻辑关系，因此数据结构又称为数据的**逻辑结构**。而相对于逻辑结构这个比较抽象的概念，我们将数据结构在计算机中的表示又称为数据的**存储结构**。

建立问题的数学模型，进而设计问题的算法，直至编出程序并进行调试通过，

这就是我们解决信息学问题的一般步骤。我们要建立问题的数学模型，必须首先找出问题中各对象之间的关系，也就是确定所使用的逻辑结构；同时，设计算法和程序实现的过程，必须确定如何实现对各个对象的操作，而操作的方法是决定于数据所采用的存储结构的。因此，数据逻辑结构和存储结构的好坏，将直接影响到程序的效率。

二、选择合理的逻辑结构

在程序设计中，逻辑结构的选用就是要分析题目中的数据元素之间的关系，并根据这些特定关系来选用合适的逻辑结构以实现对问题的数学描述，进一步解决问题。逻辑结构实际上是用数学的方法来描述问题中所涉及的操作对象及对象之间的关系，将操作对象抽象为数学元素，将对象之间的复杂关系用数学语言描述出来。

根据数据元素之间关系的不同特性，通常有以下四种基本逻辑结构：集合、线性结构、树形结构、图状（网状）结构。这四种结构中，除了集合中的数据元素之间只有“同属于一个集合”的关系外，其它三种结构数据元素之间分别为“一对一”、“一对多”、“多对多”的关系。

因此，在选择逻辑结构之前，我们应首先把题目中的操作对象和对象之间的关系分析清楚，然后再根据这些关系的特点来合理的选用逻辑结构。尤其是在某些复杂的问题中，数据之间的关系相当复杂，且选用不同逻辑结构都可以解决这一问题，但选用不同逻辑结构实现的算法效率大不一样。

对于这一类问题，我们应采用怎样的标准对逻辑结构进行选择呢？

下文将探讨选择合理逻辑结构应充分考虑的两个因素。

一、　充分利用“可直接使用”的信息。

首先，我们这里所讲的“信息”，指的是元素与元素之间的关系。

对于待处理的信息，大致可分为“可直接使用”和“不可直接使用”两类。对于“可直接使用”的信息，我们使用时十分方便，只需直接拿来就可以了。而对于“不可直接使用”的这一类，我们也可以通过某些间接的方式，使之成为可以使用的信息，但其中转化的过程显然是比较浪费时间的。

由此可见，我们所需要的是尽量多的“可直接使用”的信息。这样的信息越多，算法的效率就会越高。

对于不同的逻辑结构，其包含的信息是不同的，算法对信息的利用也会出现不同的复杂程度。因此，要使算法能够充分利用“可直接使用”的信息，而避免

算法在信息由“不可直接使用”向“可直接使用”的转化过程中浪费过多的时间，我们必然需要采用一种合理的逻辑结构，使其包含更多“可直接使用”的信息。

〖问题一〗 IOI99 的《隐藏的码字》。

〖问题描述〗

问题中给出了一些码字和一个文本，要求编程找出文本中包含这些码字的所有项目，并将找出的项目组成一个最优的“答案”，使得答案中各项目所包含的码字长度总和最大。每一个项目包括一个码字，以及该码字在文本中的一个覆盖序列（如’abcadc’就是码字’abac’的一个覆盖序列），并且覆盖序列的长度不超过1000。同时，“答案”要求其中每个项目的覆盖序列互相没有重叠。

〖问题分析〗

对于此题，一种较容易得出的基本算法是：对覆盖序列在文本中的终止位置进行循环，再判断包含了哪些码字，找出所有项目，并最后使用动态规划的方法将项目组成最优的“答案”。

算法的其它方面我们暂且不做考虑，而先对问题所采用的逻辑结构进行选择。

如果我们采用线性的逻辑结构（如循环队列），那么我们在判断是否包含某个码字 t 时，所用的方法为：初始时用指针 p 指向终止位置，接着通过 p 的不断前移，依次找出码字 t 从尾到头的各个字母。例如码字为“ABDCAB”，而文本图1-1，终止位置为最右边的箭头符号，每个箭头代表依次找到的码字的各个字母。

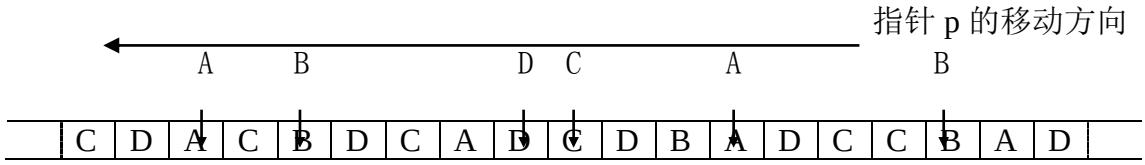

图 1-1

由于题目规定码字的覆盖序列长度不超过 1000，所以进行这样的一次是否包含的判断，其复杂度为 O(1000)。

由于码字 t 中相邻两字母在文本中的位置，并非只有相邻(如图 1-1 中的’D’和’C’)这一种关系，中间还可能间隔了许多的字母(如图 1-1 中’C’和’A’就间隔了 2 个字母)，而线性结构中拥有的信息，仅仅只存在于相邻的两元素之间。通过这样简单的信息来寻找码字的某一个字母，其效率显然不高。

如果我们建立一个有向图，其中顶点 i(即文本的第 i 位)用 52 条弧分别连接’a’..’z’,’A’..’Z’这 52 个字母在 i 位以前最后出现的位置（如图 1-2 的连接方式），我们要寻找码字中某个字母的前一个字母，就可以直接利用已连接的边，而不需用枚举的方法。我们也可以把问题看为：从有向图的一个顶点出发，寻找一条长度为 length(t)-1 的路径，并且路径中经过的顶点，按照码字 t 中的字母有序。

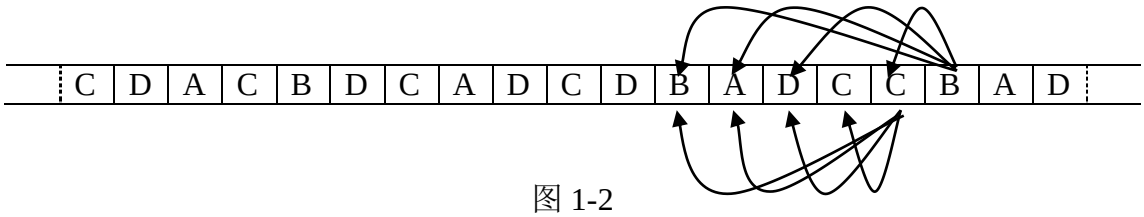

图 1-2

通过计算，用图进行记录在空间上完全可以承受(记录 1000 个点×52 条弧×4 字节的长整型=200k 左右)。在时间上，由于可以充分利用第 i 位和第 i+1 位弧的连接方式变化不大这一点(如图 1-2 所示，第 i 位和第 i+1 位只有一条弧的指向发生了变化，即第 i+1 位将其中一条弧指向了第 i 位)，所以要对图中的弧进行记录，只需对弧的指向进行整体赋值，并改变其中的某一条弧即可。

因此，我们通过采用图的逻辑结构，使得寻找字母的效率大大提高，其判断的复杂度为 O(length(t))，最坏为 O(100)，比原来方法的判断效率提高了 10 倍。

（附程序 codes.pas）

对于这个例子，虽然用线性的数据结构也可以解决，但由于判断的特殊性，每次需要的信息并不能从相邻的元素中找到，而线性结构中只有相邻元素之间存在关系的这一点，就成为了一个很明显的缺点。因此，问题一线性结构中的信息，就属于“不可直接使用”的信息。相对而言，图的结构就正好满足了我们的需要，将所有可能产生关系的点都用弧连接起来，使我们可以利用弧的关系，高效地进行判断寻找的过程。虽然图的结构更加复杂，但却将“不可直接使用”的信息，转化成为了“可直接使用”的信息，算法效率的提高，自然在情理之中。。

二、　不记录“无用”信息。

从问题一中我们看到，由于图结构的信息量大，所以其中的信息基本上都是“可用”的。但是，这并不表示我们就一定要使用图的结构。在某些情况下，图结构中的“可用”信息，是有些多余的。

信息都“可用”自然是好事，但倘若其中“无用”（不需要）的信息太多，就只会增加我们思考分析和处理问题时的复杂程度，反而不利于我们解决问题了。

〖问题二〗　湖南省 1997 年组队赛的《乘船问题》

〖问题描述〗

有 N 个人需要乘船，而每船最多只能载两人，且必须同名或同姓。求最少需要多少条船。

〖问题分析〗

看到这道题，很多人都会想到图的数据结构：将 N 个人看作无向图的 N 个点，

凡同名或同姓的人之间都连上边。

要满足用船最少的条件，就是需要尽量多的两人共乘一条船，表现在图中就是要用最少的边完成对所有顶点的覆盖。这就正好对应了图论的典型问题：求最小边的覆盖。所用的算法为“求任意图最大匹配”的算法。

使用“求任意图最大匹配”的算法比较复杂(要用到扩展交错树，对花的收缩等等)，效率也不是很高。因此，我们必须寻找一个更简单高效的方法。

首先，由于图中任两个连通分量都是相对独立的，也就是说任一条匹配边的两顶点，都只属于同一个连通分量。因此，我们可以对每个连通分量分别进行处理，而不会影响最终的结果。

同时，我们还可以对需要船只 s 的下限进行估计：

对于一个包含 P_i 个顶点的连通分量，其最小覆盖边数显然为$[P_i/2]$。若图中共有 L 个连通分量，则 $s=\sum[P_i/2](1<=i<=L)$。

然后，我们通过多次尝试，可得出一个猜想：

实际需要的覆盖边数完全等于我们求出的下限$\sum[P_i/2](1<=i<=L)$。

要用图的结构对上述猜想进行证明，可参照以下两步进行：

1．连通分量中若不存在度为 1 的点，就必然存在回路。

2．从图中删去度为 1 的点及其相邻的点，或删去回路中的任何一边，连通分量依然连通，即连通分量必然存在非桥边。

由于图的方法不是这里的重点，所以具体证明不做详述。而由采用图的数据结构得出的算法为：每次输出一条非桥的边，并从图中将边的两顶点删去。此算法的时间复杂度为 $O(n^3)$。（寻找一条非桥边的复杂度为 $O(n^2)$，寻找覆盖边操作的复杂度为 O(n)）

由于受到图结构的限制，时间复杂度已经无法降低，所以如果我们要继续对算法进行优化，只有考虑使用另一种逻辑结构。这里，我想到了使用二叉树的结构，具体说就是将图中的连通分量都转化为二叉树，用二叉树来解决问题。

首先，我们以连通分量中任一个顶点作为树根，然后我们来确定建树的方法。

1．找出与根结点 i 同姓的点 j（j 不在二叉树中）作为 i 的左儿子，再以 j 为树根建立子树。

2．找出与根结点 i 同名的点 k（k 不在二叉树中）作为 i 的右儿子，再以 k 为树根建立子树。

如图 2-1-1 中的连通分量，我们通过上面的建树方法，可以使其成为图 2-1-2 中的二叉树的结构(以结点 1 为根)。（两点间用实线表示同姓，虚线表示同名）

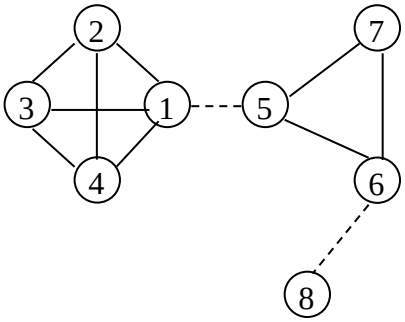

图 2-1-1

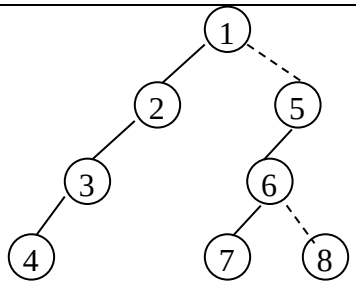

图 2-1-2

接着，我就来证明这棵树一定包含了连通分量中的所有顶点。

【引理 2.1】

若二叉树 T 中包含了某个结点 p，那么连通分量中所有与 p 同姓的点一定都在 T 中。

证明：

为了论证的方便，我们约定：s 表示与 p 同姓的顶点集合；lc[p,0]表示结点 p，lc[p,i](i>0)表示 lc[p,i-1]的左儿子，显然 lc[p,i]与 p 是同姓的。

假设存在某个点 q，满足 $q\in s$ 且 $q\notin T$。由于 s 是有限集合，因而必然存在某个 lc[p,k]无左儿子。则我们可以令 lc[p,k+1]=q，所以 $q\in T$，与假设 $q\notin T$ 相矛盾。

所以假设不成立，原命题得证。

由引理 2.1 的证明方法，我们同理可证引理 2.2。

【引理 2.2】

若二叉树 T 中包含了某个结点 p，那么连通分量中所有与 p 同名的点一定都在 T 中。

有了上面的两个引理，我们就不难得出下面的定理了。

【定理一】

以连通分量中的任一点 p 作为根结点的二叉树，必然能够包含连通分量中的所有顶点。

证明：

由引理 2.1 和引理 2.2，所有与 p 同姓或同名的点都一定在二叉树中，即连通分量中所有与 p 有边相连的点都在二叉树中。由连通分量中任两点间都存在路径的特性，该连通分量中的所有点都在二叉树中。

在证明二叉树中包含了连通分量的所有顶点后，我们接着就需要证明我们的猜想，也就是下面的定理：

【定理二】包含 m 个结点的二叉树 T_m，只需要船的数量为 boat[m]=[m/2](m∈N)。

证明：

(i) 当 m=1,m=2,m=3 时命题显然成立。

(ii) 假设当 m<k(k>3)时命题成立，那么当 m=k 时，我们首先从树中找到一个层次最深的结点，并假设这个结点的父亲为 p。那么，此时有且只有以下三种情况（结点中带有阴影的是 p 结点）：

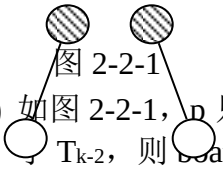

图 2-2-1

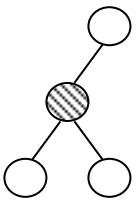

图 2-2-2

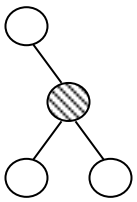

图 2-2-3

(1) 如图 2-2-1，p 只有一个儿子。此时删去 p 和 p 唯一的儿子，T_k 就成为了 T_{k-2}，则 boat[k]=boat[k-2]+1=[(k-2)/2]+1=[k/2]。

(2) 如图 2-2-2，p 有两个儿子，并且 p 是其父亲的左儿子。此时可删去 p 和 p 的右儿子，并可将 p 的左儿子放到 p 的位置上。同样地，T_k 成为了 T_{k-2}，boat[k]=boat[k-2]+1=[k/2]。

(3) 如图 2-2-3，p 有两个儿子，并且 p 是其父亲的右儿子。此时可删去 p 和 p 的左儿子，并可将 p 的右儿子放到 p 的位置上。情况与(2)十分相似，易得此时得 boat[k]=boat[k-2]+1=[k/2]。

综合(1)、(2)、(3)，当 m=k 时，boat[k]=[k/2]。

最后，综合(i)、(ii)，对于一切 m∈N，boat[m]=[m/2]。

由上述证明，我们将问题中数据的图结构转化为树结构后，可以得出求一棵二叉树的乘船方案的算法：

```
proc try(father:integer;var root:integer;var rest:byte);
{输出 root 为树根的子树的乘船方案，father=0 表示 root 是其父亲的左儿子，
father=1 表示 root 是其父亲的右儿子，rest 表示输出子树的乘船方案后，
是否还剩下一个根结点未乘船}
begin
    visit[root]:=true;  {标记 root 已访问}
    找到一个与 root 同姓且未访问的结点 j;
    if j<>n+1 then try(0,j,lrest);
    找到一个与 root 同姓且未访问的结点 k;
    if k<>n+1 then try(1,k,rrest);
    if (lrest=1) xor (rrest=1) then begin {判断 root 是否只有一个儿子，情况一}
        if lrest=1 then print(lrest,root) else print(rrest,root);
        rest:=0;
    end
    else if (lrest=1) and (rrest=1) then begin {判断 root 是否有两个儿子}
```

```
                if father=0 then begin
                    print(rrest,root);root:=j; {情况二}
                end
                else begin
                        print(lrest,root);root:=k;   {情况三}
                    end;
                rest:=1;
            end
            else rest:=1;
end;
```

这只是输出一棵二叉树的乘船方案的算法，要输出所有人的乘船方案，我们还需再加一层循环，用于寻找各棵二叉树的根结点，但由于每个点都只会访问一次，寻找其左右儿子各需进行一次循环，所以算法的时间复杂度为 $O(n^2)$。（附程序 boat.pas）

最后，我们对两种结构得出不同时间复杂度算法的原因进行分析。其中最关键的一点就是因为二叉树虽然结构相对较简单，但已经包含了几乎全部都“有用”的信息。由我们寻找乘船方案的算法可知，二叉树中的所有边不仅都发挥了作用，而且没有重复的使用，可见信息的利用率也是相当之高的。

既然采用树结构已经足够，图结构中的一些信息就显然就成为了“无用”的信息。这些多余的“无用”信息，使我们在分析问题时难于发现规律，也很难找到高效的算法进行解决。这正如迷宫中的墙一样，越多越难走。“无用”的信息，只会干扰问题的规律性，使我们更难找出解决问题的方法。

小结

我们对数据的逻辑结构进行选择，是构造数学模型一大关键，而算法又是用来解决数学模型的。要使算法效率高，首先必须选好数据的逻辑结构。上面已经提出了选择逻辑结构的两个条件（思考方向），总之目的是提高信息的利用效果。利用“可直接使用”的信息，由于中间不需其它操作，利用的效率自然很高；不不记录“无用”的信息，就会使我们更加专心地研究分析“有用”的信息，对信息的使用也必然会更加优化。

总之，在解决问题的过程中，选择合理的逻辑结构是相当重要的环节。

三、 选择合理的存储结构

数据的存储结构，分为顺序存储结构和链式存储结构。顺序存储结构的特点是借助元素在存储器中的相对位置来表示数据元素之间的逻辑关系；链式存储结构则是借助指示元素存储地址的指针表示数据元素之间的逻辑关系。

因为两种存储结构的不同，导致这两种存储结构在具体使用时也分别存在着优点和缺点。

这里有一个较简单的例子：我们需要记录一个 n×n 的矩阵，矩阵中包含的非 0 元素为 m 个。

此时，我们若采用顺序存储结构，就会使用一个 n×n 的二维数组，将所有数据元素全部记录下来；若采用链式存储结构，则需要使用一个包含 m 个结点的链表，记录所有非 0 的 m 个数据元素。由这样两种不同的记录方式，我们可以通过对数据的不同操作来分析它们的优点和缺点。

1. 随机访问矩阵中任意元素。由于顺序结构在物理位置上是相邻的，所以可以很容易地获得任意元素的存储地址，其复杂度为 O(1)；对于链式结构，由于不具备物理位置相邻的特点，所以首先必须对整个链表进行一次遍历，寻找需进行访问的元素的存储地址，其复杂度为 O(m)。此时使用顺序结构显然效率更高。
2. 对所有数据进行遍历。两种存储结构对于这种操作的复杂度是显而易见的，顺序结构的复杂度为 $O(n^2)$，链式结构为 O(m)。由于在一般情况下 m 要远小于 n^2，所以此时链式结构的效率要高上许多。

除上述两种操作外，对于其它的操作，这两种结构都不存在很明显的优点和缺点，如对链表进行删除或插入操作，在顺序结构中可表示为改变相应位置的数据元素。

既然两种存储结构对于不同的操作，其效率存在较大的差异，那么我们在确定存储结构时，必须仔细分析算法中操作的需要，合理地选择一种能够“扬长避短”的存储结构。

一、合理采用顺序存储结构。

我们在平常做题时，大多都是使用顺序存储结构对数据进行存储。究其原因，一方面是出于顺序结构操作方便的考虑，另一方面是在程序实现的过程中，使用顺序结构相对于链式结构更便于对程序进行调试和查找错误。因此，大多数人习惯上认为，能够使用顺序结构进行存储的问题，最“好”采用顺序存储结构。

其实，这个所谓的“好”只是一个相对的标准，是建立在以下两个前提条件之下的：

1. 链式结构存储的结点与顺序结构存储的结点数目相差不大。这种情况下，

由于存储的结点数目比较接近，使用链式结构完全不能体现出记录结点少的优点，并且可能会由于指针操作较慢而降低算法的效率。更有甚者，由于指针自身占用的空间较大，且结点数目较多，因而算法对空间的要求可能根本无法得到满足。

2．并非算法效率的瓶颈所在。由于不是算法最费时间的地方，这里是否进行改进，显然是不会对整个算法构成太大影响的，若使用链式结构反而会显得操作过于繁琐。

二、必要时采用链式存储结构。

上面我对使用顺序存储结构的条件进行了分析，最后就只剩下何时应该采用链式存储结构的问题了。

由于链式结构中指针操作确实较繁琐，并且速度也较慢，调试也不方便，因而大家一般都不太愿意用链式的存储结构。但是，这只是一般的观点，当链式结构确实对算法有很大改进时，我们还是不得不进行考虑的。

〖问题三〗 IOI99 的《地下城市》。

〖问题描述〗

已知一个城市的地图，但未给出你的初始位置。你需要通过一系列的移动和探索，以确定初始时所在的位置。题目的限制是：

1．不能移动到有墙的方格。

2．只能探索当前所在位置四个方向上的相邻方格。

在这两个限制条件下，要求我们的探索次数（不包括移动）尽可能的少。

〖问题分析〗

由于存储结构要由算法的需要确定，因此我们首先来确定问题的算法。

经过对问题的分析，我们得出解题的基本思想：先假设所有无墙的方格都可能是初始位置，再通过探索一步步地缩小初始位置的范围，最终得到真正的初始位置。同时，为提高算法效率，我们还用到了分治的思想，使我们每一次探索都尽量多的缩小初始位置的范围(使程序尽量减少对运气的依赖)。

接着，我们来确定此题的存储结构。

由于这道题的地图是一个二维的矩阵，所以一般来讲，采用顺序存储结构理所当然。但是，顺序存储结构在这道题中暴露了很大的缺点。我们所进行的最多的操作，一是对初始位置的范围进行筛选，二是判断要选择哪个位置进行探索。而这两种操作，所需要用到的数据，只是庞大地图中很少的一部分。如果采用顺序存储结构(如图 3-1 中阴影部分表示已标记)，无论你需要用到多少数据，始终都要完全的遍历整个地图。

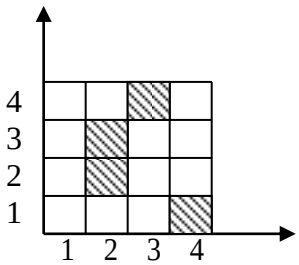

图 3-1

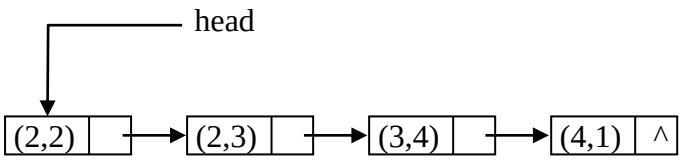

图 3-2

然而，如果我们采用的是链式存储结构(如图 3-2 的链表)，那么我们需要多少数据，就只会遍历多少数据，这样不仅充分发挥了链式存储结构的优点，而且由于不需单独对某一个数据进行提取，每次都是对所有数据进行判断，从而避免了链式结构的最大缺点。

我们使用链式存储结构，虽然没有降低问题的时间复杂度(链式存储结构在最坏情况下的存储量与顺序存储结构的存储量几乎相同)，但由于体现了前文所述选择存储结构时扬长避短的原则，因而算法的效率也大为提高。（程序对不同数据的运行时间见表 3-3）

测试数据编号	使用顺序存储结构的程序	使用链式存储结构的程序
1	0.06s	0.02s
2	1.73s	0.07s
3	1.14s	0.06s
4	3.86s	0.14s
5	32.84s	0.21s
6	141.16s	0.23s
7	0.91s	0.12s
8	6.92s	0.29s
9	6.10s	0.23s
10	17.41s	0.20s

表 3-3

（附使用链式存储结构的程序 under.pas）

我们选择链式的存储结构，虽然操作上可能稍复杂一些，但由于改进了算法的瓶颈，算法的效率自然也今非昔比。由此可见，必要时选择链式结构这一方法，其效果是不容忽视的。

小结

合理选择逻辑结构，由于牵涉建立数学模型的问题，可能大家都会比较注意。但是对存储结构的选择，由于不会对算法复杂度构成影响，所以比较容易忽视。

那么，这种不能降低算法复杂度的方法是否需要重视呢？

大家都知道，剪枝作为一种常用的优化算法的方法，被广泛地使用，但剪枝同样是无法改变算法的复杂度的。因此，作用与剪枝相似的存储结构的合理选择，也是同样很值得重视的。

总之，我们在设计算法的过程中，必须充分考虑存储结构所带来的不同影响，选择最合理的存储结构。

四、　多种数据结构相结合

上文所探讨的，都是如何对数据结构进行选择，其中包含了逻辑结构的选择和存储结构的选择，是一种具有较大普遍性的算法优化方法。对于多数的问题，我们都可以通过选择一种合理的逻辑结构和存储结构以达到优化算法的目的。

但是，有些问题却往往不如人愿，要对这类问题的数据结构进行选择，常常会顾此失彼，有时甚至根本就不存在某一种合适的数据结构。此时，我们是无法选择出某一种合适的数据结构的，以上的方法就有些不太适用了。

为解决数据结构难以选择的问题，我们可以采用将多种数据结构进行结合的方法。通过多种数据结构相结合，达到取长补短的作用，使不同的数据结构在算法中发挥出各自的优势。

这只是我们将多种数据结构进行结合的总思想，具体如何进行结合，我们可以先看下面的例子。

〖问题四〗 广东省 97 年省赛的《最小序列》问题。

〖问题描述〗

给定一个 N×N（N<=100）的正整数矩阵。需要在矩阵中寻找一条从起始位置到终止位置的路径(可沿上下左右四个方向)，并且要求路径中经过的所有数字，其相邻数字之差的绝对值之和最小。

〖问题分析〗

这道题的基本算法很简单，只要用 Dijkstra 算法求出从起始位置到终止位置的最短路径即可。但这当中存在一个很大的问题：N<=100。这就是说图中点的数目可能多达 10000 个。此时复杂度为 $O(n^2)$的 Dijkstra 算法就显得有些力不从心了。

我们继续对算法进行分析。由于 Dijstra 算法通常采用的是线性的数组结构，所以当我们每次寻找下一条最短路径时，有两步需要进行：

1．找出一个不在最短路径起点集合内，并且到终点距离最短的顶点 i。这一步的复杂度显然为 O(n)。

2．修改从与 i 相邻的顶点到终点的路径长度。由于最多只有 4 个点(四个方向)与 i 相邻，所以这一步的复杂度为 O(1)即常数。

这两步中，虽然第二步最多只改变了到 4 个点的路径长度，但是第一步却还是需要枚举所有的数组元素，这显然很浪费时间。出于对第一步进行改进的考虑，我们可以想到树结构中二叉堆的结构。堆结构的优点是：根结点就是最优的结点，并且改变堆中某个结点的值以后，只需 $O(\log_2 n)$的复杂度就可以完成堆化的过程(可分为从二叉树中自下而上和自上而下两种堆化方法，并且复杂度都一样)，使堆的结构发生改变后，通过堆化仍然能够保留堆的性质。

如果我们采用二叉堆的结构存储距离，第一步就只需将根结点取出，并且用堆中的另一结点代替后进行一次自上而下的堆化。因而第一步的复杂度可降为 $O(\log_2 n)$。但是，此时的堆结构在进行第二步时暴露了很大的缺点：无法预知 i 点的相邻点在堆中的位置。如果我们用对堆进行遍历来寻找 i 的相邻点，第二步的复杂度又成为了 O(n)。

我们从这两种数据结构中不难发现这样规律：采用线性结构的数组，易于进行第二步；采用树结构的二叉堆，做第一步时效果比较理想。

那么，我们是否可以将这两种数据结构进行结合，取长补短呢？

答案是肯定的。

我们可以采用映射的方法，将线性结构中的元素与堆中间的结点一一对应起来，若线性的数组中的元素发生变化，堆中相应的结点也接着变化，堆中的结点发生变化，数组中相应的元素也跟着变化。

将两种结构进行结合后，无论是第一步还是第二步，我们都不需对所有元素进行遍历，只需进行常数次复杂度为 $O(\log_2 n)$的堆化操作。这样，整个时间复杂度就成为了 $O(n\log_2 n)$，算法效率无疑得到了很大提高。

这个例子只是对逻辑结构的结合进行了分析，对于存储结构的结合，实际上也是相同的道理。我们同样可以在链式存储结构和顺序存储结构中建立对应的关系，并且将链式存储结构中插入和删除的操作，与顺序结构中数据元素标志的改变同时进行。这样，我们在访问数据元素时，如果需要进行遍历，就使用链式结构；如果只需要判断某一个元素，就直接访问其在顺序结构中的标记。通过存储结构的结合，我们就将两种存储结构的优点完全发挥了出来。

由此，我们又得到了在各种数据结构优劣难辩、难以取舍时，选择和使用各种数据结构的方法：用映射来达到对各数据结构进行结合的目的。这无疑又增加了一种数据结构的使用方法和技巧。

五、 总结

我们平常使用数据结构，往往只将其作为建立模型和算法实现的工具，而没有考虑这种工具对程序效率所产生的影响。信息学问题随着难度的不断增大，对算法时空效率的要求也越来越高，而算法的时空效率，在很大程度上都受到了数据结构的制约。

因此，数据结构作为信息学竞赛的基础，也是大家最为熟悉的内容，其对算法时间和空间都起到了至关重要的作用。我们惟有对数据结构优化算法的方法从原则上进行分析，在做题时充分考虑“扬长避短”的选择原则和“取长补短”的结合方法，才能更加有效地对算法进行改进。

随着数据结构的不断发展，人们对数据结构的认识也不断深入。本文只是对数据结构进行了初步的探讨，要对数据结构进行全面的了解和掌握，还需要我们更进一步的进行分析和总结。正所谓“养兵千日，用兵一时”，我们只有在平常打好基础，总结经验，才能在信息学竞赛中运用自如。

【附录】

[1] 本文对数据结构的定义，是采用的《数据结构（第二版）》中的定义，而在其它书中，数据结构的定义可能会有所不同。

[2] 由于本文的重点是数据结构而不是算法，所以文中对问题进行分析时，可能未十分详尽地写出每个问题的算法推导过程。

[3] 在对《地下城市》中使用顺序存储结构和链式存储结构的程序进行比较时，所选用数据为 IOI99 的标准数据。测试机型为 6x86 200MHz。

【程序清单】

本文由于是对数据结构的选择方法进行探讨，所以举例分析时大多采用了对不同数据结构进行比较的方法。以下仅给出了采用较好的数据结构的程序。

〖程序 1〗隐藏的码字（使用有向图的结构）：codes.pas

出于提高程序效率的目的，程序中还进行了其它的一些优化措施，如对码字排序、对求出的项目择优存储等。程序的输入输出都按照试卷的标准格式。

```
{$A+,B-,D+,E+,F-,G-,I-,L+,N-,O-,P-,Q-,R-,S-,T-,V+,X+,Y+}
{$M 2048,0,655360}
const st11='words.inp';      {记录码字的文件名}
      st12='text.inp';       {记录文本的文件名}
      st2='codes.out';       {输出文件名}
      max=1023;              {使用循环队列记录的字母数目上限}
```

```
type arr1=array[0..10000] of longint;
     arr2=array[0..10000] of shortint;
     arr3=array['A'..'z'] of longint;
var i,j,now:longint;  {now--标志已读到文本的第 now 位}
    m,n:integer;      {n--码字的个数,m—得出的项目个数}
    buf:array[1..4095] of char; {输入文件的缓冲区}
    t:array[1..100] of string[100];      {t[i]--记录各个码字}
    last:array[0..max] of ^arr3;
    {文本中位置 i 之前最后一次出现字母 c 的位置记为 last[i and max,c]}
    nearest:arr3;  {nearest[c]--当前读入的第 now 位之前最后出现字母 c 的位置}
    bb,ee:array['A'..'z'] of byte;
    {进行排序后首字母为 c 的码字在码字列表中位置为 bb[c]-ee[c],
     bb[c]>ee[c]表示首字母为 c 的码字不在码字列表中}
    no:array[1..100] of byte;    {no[i]--排序后列表中第 i 个码字在原列表中的位置}
    ss,tt:^arr1;
    nn:^arr2;
    {ss[i],tt[i],nn[i]分别记录找到的第 i 个项目中的覆盖序列的首位置,
     尾位置和其对应码字在排序后的码字列表中的编号}
    f:text;
    b:char;

procedure readcod;  {读入码字列表并对其进行排序}
var f:text;
    i,j,k,l:integer;
    st:string;
begin
     assign(f,st11);reset(f);
     readln(f,n);
     for i:=1 to n do begin
         readln(f,t[i]);no[i]:=i;
     end;
     fillchar(bb,sizeof(bb),1);
     fillchar(ee,sizeof(ee),0);
     {初始时,令所有 bb[c]>ee[c],以后找到字母 c 后再改变 bb[c]和 ee[c]的值}
     for i:=1 to n do begin
         k:=i;
         for j:=i+1 to n do  {按尾字母进行排序,在尾字母相同时按码字由长到短排序}
             if (t[j,length(t[j])]<t[k,length(t[k])]) or
                (t[j,length(t[j])]=t[k,length(t[k])]) and
                (length(t[j])>length(t[k])) then k:=j;
         if i<>k then begin
            st:=t[i];t[i]:=t[k];t[k]:=st;
            l:=no[i];no[i]:=no[k];no[k]:=l;
         end;
```

```
          {下面的判断为确定对应各尾字母的码字在新列表中的位置}
          if t[i,length(t[i])]=b then inc(ee[b])
          else begin
                    b:=t[i,length(t[i])];bb[b]:=i;ee[b]:=i;
               end;
     end;
     close(f);
end;

procedure find(st,ed:longint;e:char);
     {寻找项目的过程 ,已确定其终止位置为 ed,初始位置不小于 st,
      其包含的码字的尾字母为 e}
var i,k,l:integer;
    j:longint;
    can:boolean;
begin
     for k:=bb[e] to ee[e] do begin
          j:=ed;i:=length(t[k])-1;
          while (i>0) and (j>=st) do begin
                j:=last[j and max]^[t[k,i]];dec(i);
          end;
          {依次找出码字的各字母}
          if j>=st then begin
             l:=m;can:=true;
             while (l>0) and (tt^[l]>=j) do begin
                   if (ss^[l]>=j) and (length(t[nn^[l]])>=length(t[k])) then begin
                      can:=false;break;
                   end;
                   dec(l);
             end;
             {对找出的项目进行择优，若 can=false 表示新找到的项目不如以前的}
             if can then begin
                inc(m);ss^[m]:=j;tt^[m]:=ed;nn^[m]:=k;
             end;
          end;
   end;
end;

procedure findanswer;  {求最优“答案”}
var i,j,k:integer;
    f:text;
    len,next:^arr1;
    {len[i]表示在前 i 个项目(不必包含 i)中得出的最优“答案”包含的码字长度}
    {len[i]=max(len[j])+项目 i 的码字长度(i<j<=m,且项目 i 和项目 j 无公共部分)}
```

```
    {我们用 next[i]记录上式中对 j 所作的选择}
begin
     new(next);next^[0]:=0;
     new(len);len^[0]:=0;
     k:=0;
     for i:=1 to m do begin
         len^[i]:=len^[i-1];    {不选第 i 个项目}
         {求选第 i 个项目时可得到的最优“答案”}
         j:=i-1;
         while (j>0) and (tt^[j]>=ss^[i]) do dec(j);
         if len^[j]+length(t[nn^[i]])>len^[i] then begin
            {判断选第 i 个项目是否更优}
            len^[i]:=len^[j]+length(t[nn^[i]]);next^[i]:=j;
         end;
     end;
     assign(f,st2);rewrite(f);
     writeln(f,len^[m]);  {输出“答案”的总和值}
     k:=m;    {寻找“答案”中所包含的各个项目}
     while k<>0 do begin
           while len^[k]=len^[k-1] do dec(k);{等式成立时不必选项目 k}
           writeln(f,no[nn^[k]],' ',ss^[k],' ',tt^[k]);
           k:=next^[k];
     end;
     close(f);
end;

begin
     readcod;
     new(ss);new(tt);new(nn);
     assign(f,st12);
     settextbuf(f,buf);
     reset(f);
     fillchar(nearest,sizeof(nearest),0);
     for i:=0 to max do begin
         new(last[i]);fillchar(last[i]^,sizeof(last[i]^),0);
     end;
     {初始化}
     while not seekeoln(f) do begin
           inc(now);
           last[now and max]^:=nearest;
           read(f,b);nearest[b]:=now;  {将最后一次出现字母 b 的位置更新为 now}
           if bb[b]<=ee[b] then  {判断是否包含尾字母为 b 的码字}
              if now>=1000 then find(now-999,now,b)
              else find(1,now,b);
```

```
                {确定起始位置的下限 max(now-999,1)，并对项目进行寻找}
     end;
     close(f);
     findanswer;
end.
```

〖程序 2〗乘船问题（采用二叉树结构）：boat.pas

本题输入文件的格式为：第一行为总人数，以下每行为一个人的姓和名，中间用一个空格分开(姓和名分别为长度不超过 5 和 10 的字符串)。输出文件每行给出了一艘船上坐船的人的姓名，最后一行为需要船只的总数。

```
{$A+,B-,D+,E+,F-,G-,I-,L+,N-,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 65520,0,655360}
const Maxn=5000;  {最多的人数}
type Fnamearr=array[1..Maxn] of string[5];
     Gnamearr=array[1..Maxn] of string[10];
var Fname:^Fnamearr;   {记录每个人的姓}
    Gname:^Gnamearr;   {记录每个人的名}
    visit:array[1..Maxn] of boolean; {visit[i]记录第 i 个人是否已访问过}
    total,n:integer;   {total--需要船的数目，n--总人数}
    fo:text;

procedure init;    {读入所有人的姓名}
var f:text;
    st,t:string;
    i,j:integer;
begin
     new(Fname);new(Gname);
     write('Input file name:');readln(st);assign(f,st);reset(f);
     readln(f,n);
     for i:=1 to n do begin
         readln(f,t);
         j:=pos(' ',t);
         Fname^[i]:=copy(t,1,j-1);
         delete(t,1,j);
         Gname^[i]:=t;
     end;
     close(f);
end;

procedure print(i,j:integer);  {输出 i 和 j 同坐一条船}
var t:integer;
begin
     t:=17-length(Fname^[i])-length(Gname^[i]);
```

```
     writeln(fo,Fname^[i],' ',Gname^[i],'':t,'--  ',Fname^[j],' ',Gname^[j]);
     inc(total);
end;

procedure try(father:integer;var root:integer;var rest:byte);
  {建立以 root 为根结点的子树，并输出在这棵子树中的所有人的乘船方案}
  {father 表示 root 是其父亲的哪个儿子，0 表示左儿子，1 表示右儿子}
  {rest 表示在输出该子树中所有人的乘船方案以后，是否有一个人独坐一船}
  {rest=1 表示有，并可将这个独坐一船的人换到子树根结点 root 的位置上}
var j,k:integer;
    Lrest,Rrest:byte;
    {Lrest 和 Rrest 分别表示以 root 为根的子树中，root 的左右两儿子的 rest 值}
begin
     visit[root]:=true;
     j:=1;
     while (j<=n) and (visit[j] or (Fname^[root]<>Fname^[j])) do inc(j);
     {寻找一个与 root 同姓并且从未访问过的人 j}
     Lrest:=0;
     if j<=n then try(0,j,Lrest);{以 j 为根建立子树并输出其子树中所有人的乘船方案}
     k:=1;
     while (k<=n) and (visit[k] or (Gname^[root]<>Gname^[k])) do inc(k);
     {寻找一个与 root 同名并且从未访问过的人 k}
     Rrest:=0;
     if k<=n then try(1,k,Rrest);{以 j 为根建立子树并输出其子树中所有人的乘船方案}
     {以下按照左右两子树中是否还有人剩下，对乘坐的船进行分配}
     if (Lrest=1) xor (Rrest=1) then begin  {只有一棵子树有人剩下}
        if Lrest=1 then print(root,j) else print(root,k);
        {判断是哪棵子树有人剩下，并将那个人与 root 分配在同一条船上}
        rest:=0;
     end
     else if (Lrest=1) and (Rrest=1) then begin  {两子树都有人剩下}
             {以下由 root 和其父亲的关系，在左右两子树间选择与 root 同船的人，}
             {并将剩下的人移动到根结点 root 的位置上}
             if father=0 then begin
                print(root,k);root:=j;
             end
             else begin
                       print(root,j);root:=k
                  end;
             rest:=1;
          end
          else rest:=1;
end;
```

```
procedure findanswer;   {寻找森林中的每棵子树，也就是图中的所有连通分量}
var i,j:integer;
    rest:byte;
begin
     assign(fo,'output.txt');rewrite(fo);
     for j:=1 to n do
         if not visit[j] then begin  {若有一个人未访问，则证明其处在另一棵子树中}
            i:=j;
            try(1,i,rest);
           {以这个未访问的点为根结点建立子树，并输出该子树中所有人的乘船方案}
            if rest=1 then begin  {最后是否还剩下一个人}
               writeln(fo,Fname^[i],' ',Gname^[i]);
               inc(total);
            end;
         end;
     writeln(fo,'The total of boats is ',total);  {输出船的总数}
     close(fo);
end;

begin
     init;
     findanswer;
end.
```

〖程序 3〗地下城市（使用链式存储结构）：under.pas

程序的输入输出按照试卷的标准格式。

```
{$A+,B-,D+,E+,F-,G-,I-,L+,N-,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 65520,0,655360}
uses undertpu;
const c:array[1..4,1..2] of shortint=((0,-1),(-1,0),(1,0),(0,1));
      {四个方向的坐标变化值}
      d:array[1..4] of char=('S','W','E','N');   {与 c 中四个方向对应的字母}
      st1='under.inp';  {输入文件名}
type aa=record
              x,y:shortint;
              next:integer;
     end;
     arr=array[0..10000] of aa;
var a:array[1..100,1..100] of char;
    {地下城市地图，坐标(x,y)对应 a[x,y]。a[x,y]='O'表示无墙，a[x,y]='W'表示有墙}
    z:array[-100..100,-100..100] of char;
    {已知信息组成的新地图,(0,0)为出发位置。'O','W'分别表示无墙和有墙}
    {另外，我们用'Q'和'U'表示未知方格，但'U'表示的方格可以进行探索，'Q'就不行}
```

```
    uk,can:^arr;
    {用数组实现链表，其中链表的头结点为数组中的下标为 0 数组元素}
    {链表的结点类型为：x 和 y 为坐标，next 为下个结点的在数组中地址}
    {uk 记录目前能够探索且未知其状态的所有位置，can 记录剩下的可能出发点的坐标}
    tail,rest,v,u:integer;
    {rest 表示剩下的可能出发点数目}
    {tail 为链表 uk 的尾指针，同时也表示链表 uk 已使用 0-tail-1 的空间}
    {u 和 v 表示地图的列数和行数}

procedure readp;  {读入城市地图}
var f:text;
    i,j:integer;
begin
     assign(f,st1);reset(f);
     readln(f,u,v);
     for i:=v downto 1 do begin
         for j:=1 to u do read(f,a[j,i]);
         readln(f);
     end;
     close(f);
end;

procedure main;   {寻找出发点的主过程}
var nowx,nowy,nextx,nexty,o:integer;
 {当前相对于出发点的位置为(nowx,nowy),下一步探索目标的相对位置为(nextx,nexty)}
    e:char;

    procedure changeUK(xx,yy:integer);
    {新增一个无墙方格(xx,yy)后，我们就可以对其相邻的未知方格进行探索}
    var i,j:integer;
    begin
         for i:=1 to 4 do
             if z[xx+c[i,1],yy+c[i,2]]='Q' then begin {判断四个相邻的方格}
                z[xx+c[i,1],yy+c[i,2]]:='U';  {设置为“可探索”}
                with uk^[tail] do begin{增加一个可探索的方格，插入链表尾指针位置}
                     x:=xx+c[i,1];y:=yy+c[i,2];
                     next:=tail+1;
                end;
                inc(tail);  {使 tail 指向一个未使用的空间}
             end;
    end;

    procedure getready;   {探索前的准备}
    var i,j:integer;
```

```
begin
     nowx:=0;nowy:=0;
     fillchar(z,sizeof(z),'Q');
     z[0,0]:='0';  {为新地图设置初始值，即除起点无墙外，其它位置都未知}
     new(can);can^[0].next:=1;
     {下面对所有可能是起点的方格进行统计，放入链表 can 中}
     for i:=1 to u do
         for j:=1 to v do
             if a[i,j]='0' then begin
                inc(rest);
                with can^[rest] do begin
                     x:=i;y:=j;next:=rest+1;
                end;
             end;
     can^[rest].next:=0;
     new(uk);tail:=1;uk^[0].next:=1;
     changeUK(0,0);  {将与起点相邻的四个方格的状态改为可探索方格}
end;

procedure change;  {找出不需探索也可确定状态的方格}
var px,py,i,j,d:integer;
   {d 记录某一方格的可能状态，1 表示有墙，2 表示无墙，3 表示两种情况都可能存在}
begin
     i:=0;
     while uk^[i].next<>tail do begin  {枚举所有状态未知且可探索的方格}
           px:=uk^[uk^[i].next].x;py:=uk^[uk^[i].next].y;
           d:=0;j:=0;
           while (d<3) and (can^[j].next<>0) do begin
                 j:=can^[j].next;
                 if a[can^[j].x+px,can^[j].y+py]='W'
                    then d:=d or 1    {记录该方格可能有墙}
                    else d:=d or 2;   {记录该方格可能无墙}
           end;
           if d<3 then begin  {如果不是两种情况都可能}
              uk^[i].next:=uk^[uk^[i].next].next;
              {该方格的状态可确定，从链表中删除}
              if d=1 then z[px,py]:='W';  {判断是否只可能是有墙}
              if d=2 then begin           {判断是否只可能是无墙}
                 z[px,py]:='0';
                 changeUK(px,py);  {增加与(px,py)相邻的可探索方格}
              end;
           end
           else i:=uk^[i].next;
     end;
```

```
end;

procedure find(var xx,yy:integer);  {寻找最佳探索位置，记录为(xx,yy)}
var min,px,py,d1,d2,i,j,o:integer;
{min--当前最佳探索位置对运气的依赖程度，越小越好}
begin
     min:=maxint;i:=0;
     while uk^[i].next<>tail do begin
           px:=uk^[uk^[i].next].x;py:=uk^[uk^[i].next].y;
           d1:=0;j:=0;
           while can^[j].next<>0 do begin
                 j:=can^[j].next;
                 if a[can^[j].x+px,can^[j].y+py]='W'
                    then inc(d1);  {有墙的可能性加 1}
           end;
           d2:=rest-d1;  {计算无墙的可能性}
           if abs(d1-d2)<min then begin {判断选择(px,py)是否对运气的依赖更少}
              min:=abs(d1-d2);o:=i;xx:=px;yy:=py;
           end;
           i:=uk^[i].next;
     end;
     uk^[o].next:=uk^[uk^[o].next].next; {将探索目标从链表中删除}
end;

procedure path(x1,y1,x2,y2:integer);
{寻找移动和探索的方案，并进行探索和对出发点的筛选}
var s,t,k,l,i,j:integer;
    p:arr;
begin
     {使用广度优先搜索寻找从(x2,y2)到(x1,y1)的路径}
     p[1].x:=x2;p[1].y:=y2;p[1].next:=0;s:=1;t:=1;
     while (p[s].x<>x1) or (p[s].y<>y1) do begin
           for k:=1 to 4 do
               if z[p[s].x+c[k,1],p[s].y+c[k,2]]='0' then begin
                  inc(t);
                  p[t].x:=p[s].x+c[k,1];p[t].y:=p[s].y+c[k,2];
                  z[p[t].x,p[t].y]:='P';  {对找过的位置在地图上进行标记}
                  p[t].next:=s;   {标记 t 的父亲结点}
               end;
           inc(s);
     end;
     {通过移动到达要探索的位置(x2,y2)的相邻方格}
     l:=s;j:=p[l].next;
     while j<>1 do begin
```

```
                for i:=1 to 4 do
                    if (p[l].x+c[i,1]=p[j].x) and
                       (p[l].y+c[i,2]=p[j].y) then break;
                move(d[i]);
                l:=j;j:=p[l].next;
            end;
            {寻找探索的方向}
            nowx:=p[l].x;nowy:=p[l].y;
            for i:=1 to 4 do
                if (p[l].x+c[i,1]=p[1].x) and
                   (p[l].y+c[i,2]=p[1].y) then break;
            e:=look(d[i]);  {探索}
            z[x2,y2]:=e;    {在地图上进行标记}
            if e='0' then changeUK(x2,y2);  {如果出现无墙方格，则增加可探索的位置}
            for i:=-u to u do
                for j:=-v to v do
                    if z[i,j]='P' then z[i,j]:='0';  {还原地图}
            j:=0;
            while can^[j].next<>0 do begin  {筛选出发点}
                k:=j;
                j:=can^[j].next;
                if a[can^[j].x+x2,can^[j].y+y2]<>e then begin
                   {判断相对位置的状态是否不符合探索的结果}
                   dec(rest);can^[k].next:=can^[j].next;j:=k;
                end;
            end;
        end;

begin
    getready;  {探索前的准备}
    start;     {开始探索}
    while rest>1 do begin
          change;   {寻找不需探索也可确定状态的方格}
          find(nextx,nexty);   {寻找探索的最佳位置}
          path(nowx,nowy,nextx,nexty);
          {寻找移动和探索的具体过程，并进行对出发点的筛选}
    end;
    o:=can^[0].next;
    finish(can^[o].x,can^[o].y);  {找到出发的位置}
end;

begin
    readp;
    main;
```

```
end.
```

〖程序 4〗最小序列（线性结构和二叉堆结构相结合）：sequence.pas

本题程序的输入为：第一行为矩阵的边长 N，接下来 N 行是 N×N 的正整数矩阵，最后一行为四个数字，代表起点和终点的位置(按照先行后列的顺序)。输出文件的第一行为序列相邻项之差的绝对值之和的最小值，第二行就是其对应的相邻序列。

```
{$A+,B-,D+,E+,F-,G-,I+,L+,N-,O-,P-,Q-,R-,S+,T-,V+,X+}
{$M 16384,0,655360}
const b:array[1..4,1..2] of shortint=((1,0),(0,1),(-1,0),(0,-1));
type arr=array[0..101,0..101] of integer;
     aa=record
     {堆中结点的类型，w 表示到终点的距离，(x,y)表示该结点在矩阵中的位置}
              w:integer;
              x,y:byte;
     end;
var a,d:array[0..101,0..101] of byte;
    {a[i,j]--记录矩阵的数组,d[i,j]--从(i,j)到终点的最短路径中下一步应选择的方向}
    pos:^arr;
    {pos[i,j]--用于不同数据结构间进行映射的数组，表示从矩阵中(i,j)位置到终点的}
    {距离在堆结构中存储的位置，即将距离记录在 h[pos[i,j]].w}
    h:array[0..10001] of aa;
    {用数组表示的二叉堆，其中 h[i]的左儿子为 h[i*2]，右儿子为 h[i*2+1]}
    x1,y1,x2,y2,n:byte;  {n--矩阵的边长，(x1,y1),(x2,y2)--起点和终点位置}
    t:integer;  {堆中的结点总数}

procedure readp;  {读入数据}
var f:text;
    st:string;
    i,j:byte;
begin
     new(pos);
     write('File name:');readln(st);assign(f,st);reset(f);
     readln(f,n);
     for i:=1 to n do
         for j:=1 to n do
             read(f,a[i,j]);
     readln(f,x1,y1,x2,y2);
     close(f);
end;

procedure swap(i,j:integer);  {交换堆中的第 i 个和第 j 个结点}
var k:aa;
begin
```

```
     k:=h[i];h[i]:=h[j];h[j]:=k;   {交换距离}
     pos^[h[i].x,h[i].y]:=i;pos^[h[j].x,h[j].y]:=j;  {交换映射数组指向的位置}
end;

procedure heap1(i:integer);  {从 i 结点向下进行堆化}
var j:integer;
begin
     while i*2<=t do begin
           if (i*2+1<=t) and (h[i*2+1].w<h[i*2].w)
              {寻找左右儿子中到终点距离最短的}
              then j:=i*2+1
              else j:=i*2;
           if h[j].w<h[i].w then begin  {是否需要与根结点交换}
              swap(i,j);
              i:=j;
           end
           else break;
     end;
end;

procedure heap2(i:integer);  {从 i 结点向上进行堆化}
begin
     while (i<>1) and (h[i div 2].w>h[i].w) do begin  {判断是否需要与其父亲交换}
           swap(i div 2,i);
           i:=i div 2;
     end;
end;

procedure main;   {寻找从所有位置到终点的最短路径}
var xx,yy,i,j:byte;
    fo:text;
begin
     h[10001].w:=9999;
     for i:=1 to n do
         for j:=1 to n do
             pos^[i,j]:=10001;
     {设定从各位置到终点的距离为∞}
     h[0].w:=0;
     for i:=1 to n do begin
         pos^[i,0]:=0;pos^[0,i]:=0;pos^[i,n+1]:=0;pos^[n+1,i]:=0;
     end;
     {为了不移到矩阵外，在矩阵四周加上一圈，并假设到这一圈的距离都为 0}
     {因为 0 是不可能被变得更小的，所以也就不会移出矩阵了}
     with h[1] do begin
```

```
          w:=0;x:=x2;y:=y2;
     end;
     pos^[x2,y2]:=1;t:=1; {将终点作为堆的根结点}
     repeat
          {堆中的根结点就是最短路径的起点}
          for i:=1 to 4 do begin  {改变从根结点的相邻点到终点的距离}
              xx:=h[1].x+b[i,1];yy:=h[1].y+b[i,2];  {计算相邻点的坐标}
              if h[pos^[xx,yy]].w=9999 then begin  {判断是否在堆中}
                 inc(t);
                 with h[t] do begin {堆中增加一个新结点}
                      w:=h[1].w+abs(a[h[1].x,h[1].y]-a[xx,yy]);
                      x:=xx;y:=yy;
                      pos^[xx,yy]:=t;d[xx,yy]:=i;
                 end;
                 heap2(t);  {从新增的结点开始(向上)堆化}
              end
              else if h[1].w+abs(a[h[1].x,h[1].y]-a[xx,yy])<h[pos^[xx,yy]].w
                      then begin  {判断是否得到更短的距离}
                      h[pos^[xx,yy]].w:=h[1].w+abs(a[h[1].x,h[1].y]-a[xx,yy]);
                      {更新距离}
                      heap2(pos^[xx,yy]);  {从堆中的对应点开始(向上)堆化}
                      d[xx,yy]:=i;
                   end;
          end;
          swap(1,t);pos^[h[t].x,h[t].y]:=0;dec(t);  {取出根结点}
          heap1(1);  {从根结点开始(向下)堆化}
     until (h[1].x=x1) and (h[1].y=y1);  {直到找到从给出的起点到终点的最短距离}
     assign(fo,'output.txt');rewrite(fo);
     writeln(fo,h[1].w);  {输出距离}
     {输出路径中经过的点}
     while (x1<>x2) or (y1<>y2) do begin
          write(fo,a[x1,y1],' ');i:=x1;j:=y1;
          dec(x1,b[d[i,j],1]);dec(y1,b[d[i,j],2]);
     end;
     writeln(fo,a[x2,y2]);
     close(fo);
end;

begin
     readp;
     main;
end.
```

【参考书目】

1. 《数据结构（第二版）》，严蔚敏，吴伟民编著，清华大学出版社。
2. 《实用算法的分析与程序设计》，吴文虎，王建德编著，电子工业出版社。
3. 《青少年国际和全国信息学（计算机）奥林匹克竞赛指导——图论的算法与程序设计》，吴文虎，王建德编著。
4. 《信息学奥林匹克》（季刊），1998 年第一、二期。
5. IOI99 试题以及湖南省历届省赛试题。