

线性表讲稿

一、线性表的逻辑结构

线性表是最基本、最简单、也是最常用的一种数据结构。线性表的逻辑结构描述为：

1. 存在唯一的“**第一个元素**”（**第一个结点**）；
2. 存在唯一的“**最后一个元素**”（**最后一个结点**）；
3. 除最后一个元素之外，其余元素均有 **唯一的前驱**（**前驱继结点**）；
4. 除第一个元素之外，其余元素均有 **唯一的后继**（**后继结点**）。

线性表类似一列火车，火车的每个车厢就是线性表的一个数据元素，车头没有前驱，车尾没有后继。每个车厢所装内容都是同类型的，即线性表是若干相同类型的数据元素排成的一个线性序列。

二、线性表的存储结构和操作

【引例】、线性表维护

给定一个长度为 n 的整数序列。现在有 m 个操作，操作分为下面几类：

- 1 i ：询问序列中第 i 个元素的值，保证 i 小于等于当前序列长度。
- 2 v ：询问 v 在序列中出现的最早位置，如果找不到，输出 "not found!"。
- 3 i v ：在序列的第 i 个元素前插入一个元素 v ，保证 $1 \leq i \leq n+1$ （ n 表示当前序列的长度）。
- 4 i ：删除序列中第 i 个元素，保证 $1 \leq i \leq n$ （ n 表示当前序列的长度）。

【输入格式】

第 1 行包含两个整数： n, m ，分别表示序列的最初长度和后面要操作数。

第 2 行包含 n 个整数，表示序列从头到尾的每个元素。

接下来的 m 行，每行表示题目所述的 4 类操作之一。

【输出格式】

按输入顺序输出操作 1 和 2 的结果。

【数据范围】

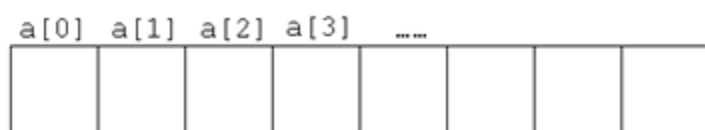
$N, m \leq 1000$ ，元素值是在 int 范围内。

【讲解】

线性表有两种存储结构：顺序存储和链式存储：

1、顺序存储（数组）

数组是线性表一种重要存储结构，基本特点是物理的存储地址与线性表元素的逻辑关系一致，对于数组： $\text{int } a[20005]$ ；在内存中是一段连续的空间， $a[1]$ 是第一个元素， $a[N]$ 是最后一个元素， $a[i]$ 的物理地址是 $a+i$ ，前驱是 $a[i-1]$ 物理地址为 $a+i-1$ ，后继 $a[i+1]$ 的物理地址是 $a+i+1$ 。



因为数组中逻辑上相邻的元素在物理存储地址上也是相邻的，所以在数组中插入或删除元素都需要移动一些元素，具体来说：

1)、插入：在 $a[i]$ 前插入一个元素 v ，类似在 n 个人队列中的第 i 个位置中插入一个人，那么第 i 到 n 个人都依次后移一个位置，才能腾出一个空位。一次插入操作最好情况是插入到最后的位置，此时不需要移动元素，最坏情况是插入第一个位置，此时需要移动 n 个元素，所以平均情况下需要的移动次数为 $\frac{n}{2}$ ，时间复杂度为 $O(n)$ 。

2)、删除：删出第 i 个元素，类似在 n 人的队列中第 i 个人出列，那么后面的第 $i+1$ 到 n 个人需要依次前移一个位置（填补空位）。同理一次删出操作最好情况是删出最后一个元素，此时不需要移动元素，最坏情况是删出第一个元素，此时需要移动 $n-1$ 个元素，所以平均情况下需要的移动次数为 $\frac{n}{2}$ ，时间复杂

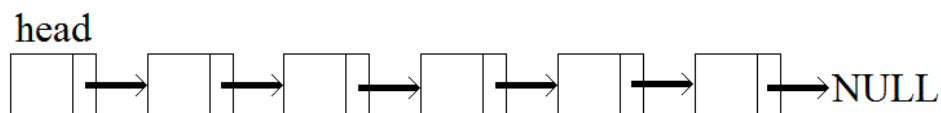
度为 $O(n)$ 。

```
int n,m,a[2005];
void solve()
{
    int i,v,op;
    while(m--)
    {
        scanf("%d",&op);
        if(op==1)
        {
            scanf("%d",&i);
            printf("%d\n",a[i]);
        }
        else if(op==2)
        {
            scanf("%d",&v);
            i=Find(v);
            if(i<0) printf("not found!\n");
            else printf("%d\n",i);
        }
        else if(op==3)
        {
            scanf("%d%d",&i,&v);
            Insert(i,v);
        }
        else if(op==4)
        {
            scanf("%d",&i);
            Delete(i);
        }
    }
}
```

```
int Find(int v) //查找元素 v
{
    for(int i=1;i<=n;i++)
        if(a[i]==v) return i;
    return -1;
}
void Insert(int pos,int v) //pos 前插入 v
{
    for(int i=n;i>=pos;i--) //腾位置
        a[i+1]=a[i];
    a[pos]=v;
    n++;
}
void Delete(int pos) //删出 pos 位置上的元素
{
    for(int i=pos;i<n;i++) //填补空位
        a[i]=a[i+1];
    n--;
}
```

2、链式存储（链表）

线性表的链式存储结构相对于顺序存储结构的特点是：逻辑上相邻但物理存储地址上不一定相邻。所以链式存储结构用地址指针体现相邻元素的关系——当前元素的下一个元素存储在在哪里，或者上一个元素存储在在哪里。此时线性表的每个元素是一个节点：元素值 和 下一个元素或上一个的指针。链式存储结构也称为链表。下图是一个单向链表示意图：



在 C++ 中，链表可以用指针来实现，也可以用数组来模拟实现，在这里介绍数组方法。

1)、结点类型定义

```
const int maxn=2005 //线性表可能的最多结点数!!
struct node
{
    int v;           //元素值
    int next;        //下一个元素的位置
} a[maxn];          //链表存储数组（存储池）
int head=0,np=0;    // head 是链表头，np 用于申请新结点； 注意 head 不是第一个元素，a[head].next 才是！
```

2) 链表的 Link 操作

把元素 $a[p]$ 与 $a[q]$ 链接起来 $\text{Link}(p, q)$ ，使 $a[q]$ 成为 $a[p]$ 的后继，是单链表的基本操作：

<pre>void Link(int p,int q) { a[p].next=q; }</pre>	<p>请理解下面程序，并写出程序输出的内容：</p> <pre>for(int i=1;i<=5;i++) a[i].v=i; head=0; Link(head,4); Link(4,3); Link(3,5); Link(5,1); Link(1,2); Link(2,0); int p=head; //跟随指针 for(p=a[p].next;a[p].next!=0;p=a[p].next) printf("%d->",a[p]); printf("%d\n",a[p]);</pre> <p>程序输出：4 3 5 2 1</p>
--	---

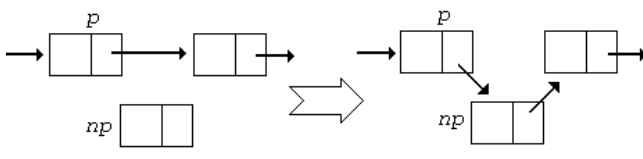
3)、在链表中获取线性表的第 i 个元素的存储位置 (时间复杂度 $O(n)$)

<pre>int Getpos(int i) //第 i 个元素的节点地址 { int p=head; for(int j=1;j<=i;j++) p=a[p].next; return p; }</pre>	<p>注意：在链表中，$a[i]$ 不是第 i 个元素，而是某个元素存储的位置，甚至可能不是线性表的元素。那么线性表的第 i 个元素在哪里呢，需要从头开始遍历：</p>
---	---

4)、在链表中查找第一个值为 v 元素是线性序列的第几个 (时间复杂度 $O(n)$)

<pre>int Find(int v) //查找元是第几个元素 { int p=head,cnt=0; for(p=a[p].next; p!=0; p=a[p].next) { cnt++; if(a[p].v==v) return cnt; } return -1; //找不到 }</pre>	<p>顺序查找：从链表头开始依次询问！</p>
--	-------------------------

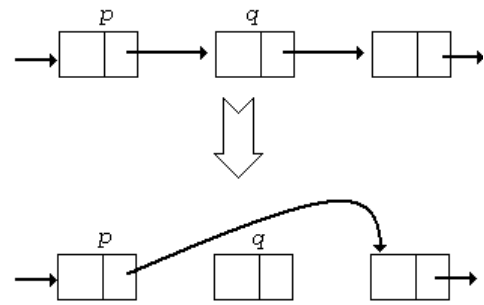
5)、在链表中插入一个元素 (时间复杂度 $O(n)$)

<pre>void Insert(int i,int v) { int p=Getpos(i-1); a[++np].v=v; Link(np,a[p].next); Link(p,np); n++; }</pre>	<p>在线性表的第 i 个元素前插入一个元素 v，先找到第 i 个元素前驱的位置 p，再新建节点 np，左后把 np 插入到 p 的后继位置：</p> 
--	---

6)、在链表中删出一个元素 (时间复杂度 $O(n)$)

```
void Delete(int i)
{
    int p=Getpos(i-1);
    int q=a[p].next;
    Link(p,a[q].next);
    n--;
}
```

先找到第 i 个元素前驱的位置 p , 然后删出 p 的后继元素 q :

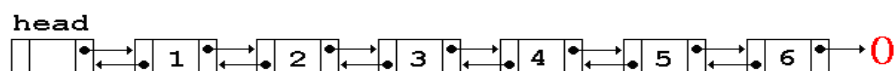


对上述 1) ~6) 充分理解后, 可以代码主程序就是调用以上函数:

```
void solve()
{
    scanf("%d%d", &n, &m);
    head=0;
    for(int i=1; i<=n; i++) //建立初始链表
    {
        scanf("%d", &a[i].v);
        Link(i-1, i);
    }
    Link(n, 0);
    np=n;
    while(m--)
    {
        scanf("%d", &op);
        if(op==1)
        {
            scanf("%d", &i);
            int pos=Getpos(i);
            printf("%d\n", a[pos].v);
        }
        else if(op==2)
        {
            scanf("%d", &v);
            i=Find(v);
            if(i<0) printf("not found!\n"); else printf("%d\n", i);
        }
        else if(op==3)
        {
            scanf("%d%d", &i, &v);
            Insert(i, v);
        }
        else if(op==4)
        {
            scanf("%d", &i);
            Delete(i);
        }
    }
}
```

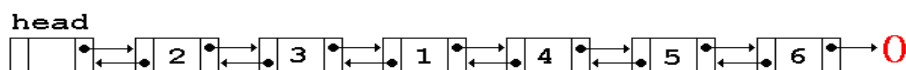
引例 2、双向链表操作 (P1528)

你有一些小球，从前到后依次编号 $1, 2, \dots, n$ 。如图所示

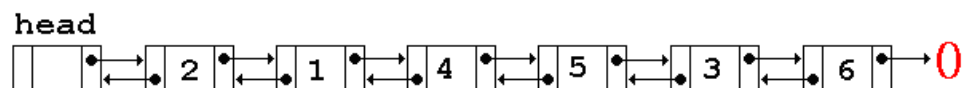


你可以执行两种命令，其中， $A \ x \ y$ 表示小球 x 移到小球 y 后边， $B \ x \ y$ 表示小球 x 移到小球 y 前边。指令保证合法。

例如，在初始状态下执行 $A \ 1 \ 4$ 后，小球 1 被移到小球 4 的前边，如下图所示：



如果执行 $B \ 3 \ 5$ ，小球 3 会移到小球 5 的后边，如下图所示：

**【样例】**

```
6 2 //n 和 m: n<=500000, m<=100000
A 1 4
B 3 5
```

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

【讲解】

由题目描述可知：每个结点不仅有一个后继指针，也有一个前驱指针，我们把这个链表称为双向链表。

1、结点定义：由于本题的每个结点只有一个信息——编号，所以没必要定义结构：

```
const int maxn=500005;
struct node
{
    int id;    //小球编号
    int pred,next; //前驱、后继指针
}a[maxn];
int head=0,np=0; //head 是表头、np 用于记录空间申请
```

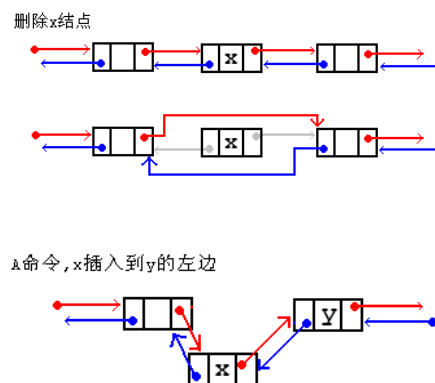
2、结点连接操作，因为每个结点有两个指针，所以最后专门写一个函数连接两个结点

```
void Link(int p,int q)
{
    a[p].next=q, a[q].pred=p;
}
```



3、解答程序：所有操作都不会移动小球，而是前驱和后继元素的变化，即左右指针的变化

```
int main()
{
    scanf("%d%d",&n,&m);
    head=0;
    for(int i=1;i<=n;i++) //建立链表
    {
        a[i].id=i;
        Link(i-1,i);
    }
    Link(n,0);
    np=n;
    for(int i=1;i<=m;i++)
    {
        scanf("%s%d%d",op,&x,&y);
```

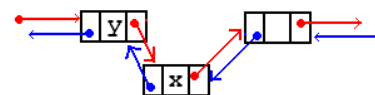


```

Link(a[x].pred,a[x].next);
if (op[0]=='A')
{
    Link(a[y].pred,x);
    Link(x,y);
}
else
{
    Link(x,a[y].next);
    Link(y,x);
}
}
for(int p=a[head].next;p!=0;p=a[p].next)
    printf("%d ",a[p].id);
printf("\n");
return 0;
}

```

B命令，x插入到y的右边



三、链表应用例题

例 3、约瑟夫问题[2] (P1527)

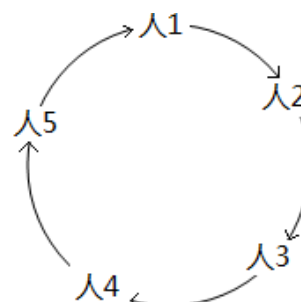
编号为：1、2、3、...、N 的 N 个人按顺时针方向围坐一圈，每人持有一个密码（正整数）。从指定编号为 1 的人开始，按顺时针方向自 1 开始顺序报数，报到指定数 M 时停止报数，报 M 的人出列，并将他的密码作为新的 M 值，从他在顺时针方向的下一个人开始，重新从 1 报数，依此类推，直至所有的人全部出列为止。请设计一个程序求出列的顺序。

【样例】

5 3 //N,M: 1 < N <= 1000 1<= M <=500	3
2	4
3	1
1	5
2	2
3	

【讲解】

N 个人排成一圈，为了删除方便，把这个 N 个人看成一个链表，且是**循环链表**。所谓循环链表，就是把单链表的尾部接点的后继指针指向链表的第一个结点（Link(N,a[head].next)，就形成了一个**循环链表**，这样的存储结构与题目描述的“N 个人按顺时针方向围坐一圈”正好相符！



在链表中删除一个结点无须移动结点，时间复杂度为 $O(1)$ ，所以整个算法的时间复杂度为 $O(N*M)$ ，可以通过此题。

数据结构定义及算法框架	建立单循环链表
<pre> const int maxn=1005; struct node { int id,m; //人的密码 int next; //后继指针 }a[maxn]; int head=0,np=0; </pre>	<pre> int main() { scanf("%d%d",&N,&M); head=0; for(int i=1;i<=N;i++) //建立循环链表 { scanf("%d",&a[i].m); a[i].id=i; } </pre>

	<pre> Link(i-1,i); } Link(N,a[head].next); //循环 int p=head,q; for(int i=1;i<=N;i++) //总共需要删出N次 { for(int k=1;k<M;k++) p=a[p].next; //报数 q=a[p].next; printf("%d\n",a[q].id); m=a[q].m; Link(p,a[q].next); //删出 } return 0; } </pre>
--	--

例 4、约瑟夫问题[3] (P1531)

M 个人围成一圈，任意指定一个人为其编号为 1，余下的人按顺时针依次编号为 2 到 M，其中编号为 M 的人与编号为 1 的人相邻。

现在以编号为 S ($1 \leq S \leq M$) 的人为起点，开始顺时针报数，报到 N 的人出列；然后以出列人的左边的人为起点，开始逆时针报数，报到 K 的人出列；接着再以出列的人右边的人为起点，开始顺时针报数，报到 N 的人出列……。就这样按顺时针和逆时针方向交替不断报数，直到全部人出列为止。请你输出出列人的编号序列。

【样例】

6 2 3 1 //M,N,K,S: N<=20000	2 5 1 3 6 4 //依次出列编号
-----------------------------	----------------------

【讲解】

相对于上题区别在于：报数是顺时针和逆时针交替报数。所以需要建立双向循环链表：

<pre> int M,N,K,S; struct node { int id; int pred,next; }a[20005]; int head=0,np=0; </pre>	<pre> int main() { scanf("%d%d%d%d",&M,&N,&K,&S); head=0; for(int i=1;i<=M;i++) { a[i].id=i; Link(i-1,i); } Link(M,a[head].next); //双向循环 int p=a[S].pred,q; for(int i=1;i<=M;i++) { if(i%2) //顺时针数 { for(int j=1;j<N;j++) p=a[p].next; q=a[p].next; printf("%d ",a[q].id); Link(p,a[q].next); p=a[p].next; } else //逆时针数 { </pre>
--	---

	<pre> for(int j=1;j<K;j++) p=a[p].pred; q=a[p].pred; printf("%d ",a[q].id); Link(a[q].pred,p); p=a[p].pred; } } return 0; } </pre>
--	--

例 5、小 M 的问题

小 M 生活在一个神奇的国家，这个国家有 N 个城市（编号为 $1..N$ ），还有 M 条道路，每条道路连接着两个不同的城市，通过这条道路，这两个城市可以相互直接到达。现在小 A 想知道每个城市可以直接到达那些城市，请你来帮助它。

【输入格式】

第 1 行包含两个整数 N, M ($N \leq 100000$ $M \leq 500000$)。

接下来的 M 行，每行两个整数，描述了一条道路连接两个城市的编号。

【输出格式】

输出 N 行，每行有若干个整数，其中第 i 行的整数表示城市 i 能直接到达的城市编号，输出顺序按照输入的先后顺序出现。

【输入样例】

4 5	3 4 2
2 3	3 4 1
3 1	2 1
1 4	1 2
2 4	
1 2	

【讲解】

信息分类存储。按城市分类，把每个城市能直达的城市组织成一个线性表，则共有 N 个线性表。如果用二维数组 $g[i][j]=1$ 来表示城市 i 能直达的第 j 个城市的编号为 k ，因为每个城市能直达的城市最多可能有 $N-1$ 个，那么总共需要 $N*N$ 的二维数组，这样算下来数组大小需要 9G 的空间，无法承受！

这里我们介绍另一种方法：定义一个存储池（数组： $E[2*maxm]$ ），来存储 N 个线性表的每个元素，同一个线性表的元素用 $next$ 指针串联起来；再设置一个 $head[u]$ 和 $tail[u]$ 分别指向城市 u 能直达城市组成的链表的第一个节点和最后一个节点。如果增加一条道路 (u, v) ，则在城市 u 的链表尾部增加一个节点 v ，同时也在城市 v 的链表尾部增加一个节点 u 。这样存储池就只需要道路数量的两倍（ $2*M$ ）大小即可，大大节省了空间。

样例图	存储结构
	<div> <div>first[u]</div> <div>E</div> <div>tail[u]</div> </div>

程序如下：时间复杂度为 $O(m)$

数据定义	代码
<pre> const int maxm=1000005; const int maxn=1000005; struct edge //链表节点信息 { int to; //直达的目标城市 int next; }E[2*maxm]; //内存池 int first[maxn]={0},tail[maxn]={0},np=0; int n,m; </pre>	<pre> void addedge(int u,int v) //新加一个结点 { E[++np]=(edge){v,0}; //新建一个节点 if(first[u]==0) //v 是 u 的第一个直达城市 first[u]=np; else //插入尾部 E[tail[u]].next=np; tail[u]=np; //u 的直达城市表的尾部 } int main() { scanf("%d%d",&n,&m); int x,y; for(int i=1;i<=m;i++) { scanf("%d%d",&x,&y); addedge(x,y); addedge(y,x); } for(int i=1;i<=n;i++) { for(int k=first[i];k!=0;k=E[k].next) printf("%d ",E[k].to); printf("\n"); } return 0; } </pre>

注意，还可以用 STL 变长数组 `vector` 来分类存储每个城市直达城市的列表，使得每个城市直达城市的线性表有多少个就申请相应大小的空间。请自行实现。

四、STL 中的双向链表 `list`

1、`list` 容器

<pre> #include<list> 定义格式: list<元素类型>链表名称 </pre>	<p>例如:</p> <pre>list<int>A;</pre>
--	---

2、`list` 常用操作

操作函数	功能	时间复杂度
<code>A.size()</code>	返回链表中元素数目	$O(1)$
<code>A.begin()</code>	链表第一个元素的迭代器地址	$O(1)$
<code>A.end()</code>	链表最后一个元素后一个位置的迭代器地址	$O(1)$
<code>A.push_front(x)</code>	在表头插入一个元素	$O(1)$
<code>A.push_back(x)</code>	在表尾插入一个元素	$O(1)$
<code>A.pop_front()</code>	删出表的第一个元素	$O(1)$
<code>A.pop_back()</code>	删出表的最后一个元素	$O(1)$
<code>A.insert(p,x)</code>	在迭代器地址 <code>p</code> 处插入一个元素	$O(1)$
<code>A.erase(p)</code>	删出迭代器 <code>p</code> 指向的元素	$O(1)$
<code>A.clear()</code>	清空表	$O(n)$

3、示例

下面的代码是用 `list` 来实现 引例 1 的代码：

<pre>#include<cstdio> #include<list> using namespace std; list<int>A; int n,m,np; list<int>::iterator Getpos(int i) { list<int>::iterator p=A.begin(); for(int j=2;j<=i;j++) p++; return p; } int Find(int v) //查找元是第几个元素 { list<int>::iterator p; int cnt=0; for(p=A.begin();p!=A.end();p++) { cnt++; if(*p==v) return cnt; } return -1; } void Insert(int i,int v) { list<int>::iterator p=Getpos(i); A.insert(p,v); } void Delete(int i) { list<int>::iterator p=Getpos(i); A.erase(p); }</pre>	<pre>int main() { list<int>::iterator it; int i,v,op,p; scanf("%d%d",&n,&m); for(int i=1;i<=n;i++) { scanf("%d",&v); A.push_back(v); } while(m--) { scanf("%d",&op); if(op==1) { scanf("%d",&i); it=Getpos(i); printf("%d\n",*it); } else if(op==2) { scanf("%d",&v); i=Find(v); if(i<0) printf("not found!\n"); else printf("%d\n",i); } else if(op==3) { scanf("%d%d",&i,&v); Insert(i,v); } else if(op==4) { scanf("%d",&i); Delete(i); } } return 0; }</pre>
--	--

另外：变长数组 `vector` 也是一种线性表的存储结构，它的相关操作前面已经做过介绍！