链表

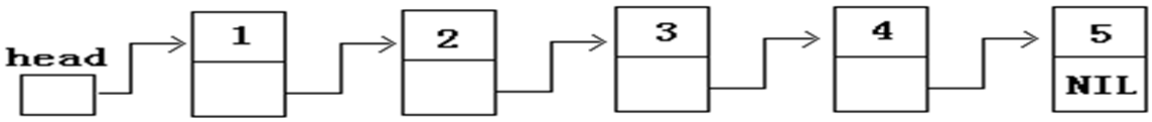
链表，是一种线性数据结构。它的优势我们可以通过与数组的对比来看一下。

数组，优势在于可以随机访问，调用非常方便。但是劣势也非常明显：1.删除和插入的操作中会涉及到大量数据的移动。2.定义的空间必须是连续的，这就导致当需要一个较大空间的数组时，系统将面临无法分配处如此之大的连续空间的问题。

链表，是一种物理上分散，但逻辑上连续的数据结构。它可以将各个数据存入到内存中任意的空余空间，哪怕这些空间是分散的。但在空间与空间之间建立一条隧道，即通过指针将空间串联起来，使得我们可以有序访问。在逻辑上，它便是连续的。当我们需要插入和修改时，只需更改其指针的值即可。

单链表

head为头节点，每个元素中data存储数据的值，后继指针next存储下一个数据点的内存地址。每访问到一个数据点，将可沿着指针继续访问下一个数据点。



每一个数据点的定义：

```
struct node{
    int data;//存数据
    node * next;//存下一个数据点的地址
};
```

```
node * p,*head=NULL;//定义指针,NULL是一个常量，代表空
```

向系统申请空间：

```
p=new node;
```

释放空间：

```
delete p;
```

节点的赋值：

```
p->data=x;
```

由于p是一个指针，也有人会用另一种写法，

```
(*p).data=x;
```

但相比较而言，第一种写法更为通用。

单链表的基本操作：

1.构建单链表

```
void creat_list(){
    int n,x;
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>x;
        p=new node;//申请新的空间
        p->data=x;//将x封装成node节点
        //反向插入写法，从头部插入
        p->next=head;//next指向此时head所指向的地址
        head=p; //将head指向自己。
    }
}
```

2.输出单链表：

```
void print_list(){
    p=head;
    while(p!=NULL){//注意结束条件
        cout<<p->data<<" ";
        p=p->next;
    }
    cout<<endl;
}
```

3.插入操作：

```
void insert_list(int k,int x){//在第k个节点插入x
    p=new node;
    p->data=x;
    if(k==1){//头部插入
        p->next=head;//next指向此时head指向的节点，即目前第一个节点
        head=p; //head指向自己，称谓新的首节点
    }else{//非头部插入
        int i=1;
        node * q=head;
        while(i<k-1){//找到第k-1个节点
            q=q->next;
            i++;
        }
        p->next=q->next;//q为第k-1个节点，它的next指向此时第k个节点。将此值赋予p的next后，p将成为新的第k个节点
        q->next=p;//第k-1个节点的next指向p，即指向了新的第k个节点
    }
}
```

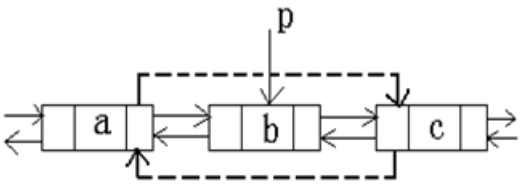
4.删除操作：

```
void delete_list(int k){
    if(k==1){//头部删除
        p=head;
        head=p->next;
        delete p;
    }else{
```

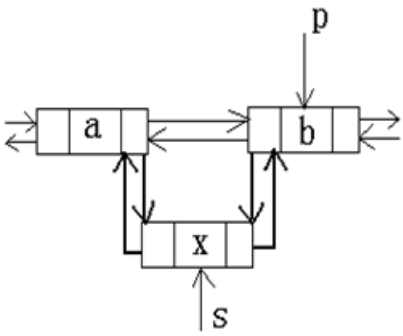
```
        int i=1;
        p=head;
        while(i<k-1){//p为第k-1个节点
            p=p->next;
            i++;
        }
        node *q=p->next;//q为待删除的第k个节点
        p->next=q->next;//第k-1个节点p将指向q的下一个节点，即跳过第k个节点，指向第k+1个节点
        delete q;//将q所在空间释放
    }
}
```

双向链表

链表中每个节点有两个指针，一个指向前驱，一个指向后继。需要注意的是，在双向链表中若要进行插入和删除的操作，有四个指针的值需要修改。



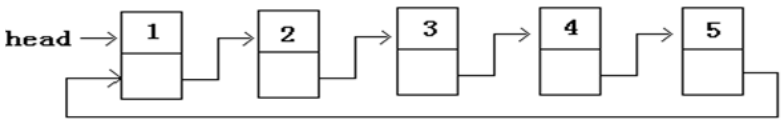
删除P结点前后的指针变化



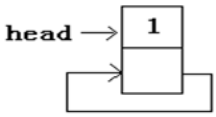
在P结点之前插入S结点前后的指针变化

循环链表

大体与单链表相似，但是最后一个节点的next指针指向头结点，则使得整个链表形成一个环。同理，也可构建出双向循环链表。



非空表



空表 (head^.next:=head)

单向循环链表

练习：

1.链表不具有的特点是( )

A. 插入删除不需要移动元素

B.不必事 先估计存储空间

C.所需空间与线性表长度成正比

D.可随机访问任一元素

2.线性表若采用链表存储结构，要求内存中可用存储单元地址( )。

A.必须连续

B. 部分地址必须连续

C.一定不连续

D.连续不连续均可

3.向一个栈顶指针为hs的链式栈中插入一个指针s指向的结点时，应执行()。

A. hs->next = S;

B. s->next = hs; hs = s;

C. s->next = hs->next; hs->next= s;

D. s->next = hs; hs = hs->next;

4.双向链表中有两个指针域，llink 和rlink,分别指回前驱及后继，设p指向链表中的一个结点， q指向一待插入结点， 现要求在p前插入 q，则正确的插入为 ()。

A. p->llink = q; q->rlink = p;

p->llink->rlink = q; q->llink = p->llink;

B. q->llink = p->llink; p->llink->rlink = q;

q->rlink = p; p->llink = q->rlink;

C. q->rlink = p; p->rlink = q;

p->llink->rlink = q; q->rlink = p;

D. p->llink->rlink = q; q->rlink = p;

q->llink = p->llink; p->llink = q;

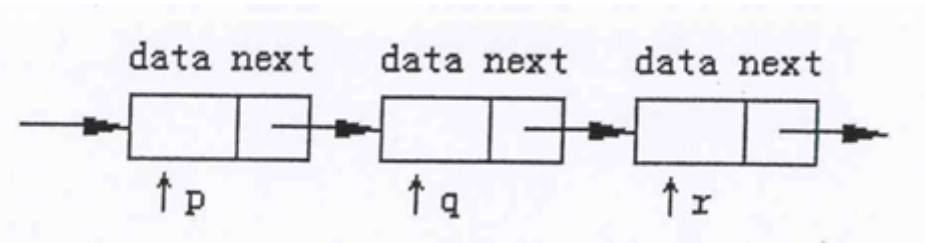
5.有以下结构体说明和变量定义，如图所示，指针p、q、r分别指向一个链表中的三个连续结点。

```
struct node {

    int data;

    node *next;

}*p,*q, *r;
```



现要将q和r所指结点的先后位置交换，同时要保持链表的连续，以下程序段中错误的是()。

A. q->next = r->next; p->next = r;r->next = q;

B. p->next = r; q->next = r->next; r->next = q;

C. q->next = r->next; r->next = q; p->next = r;

D. r->next = q; q->next = r->next; p->next = r;

【答案】1.D 2.D 3.C 4.D

静态链表

在日常使用中，静态链表更为常用。对比链表每次使用都适用new去申请一个节点的空间，静态链表是通过数组预先申请一块数据域，每次使用将数据填入每个单元，然后去构建链即可。

在预先生成的数据空间，每个但愿都有两个参数，一个是数据data，一个是用数组下标来表示地址next。定义代码如下：

```
struct node{
    int data;
    int next;
}a[1000];//预先生成1000个元素点
int head=-1;
```

构建静态链表：

```
void createList(){
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>a[++tot].data;//通过tot的递增每次将数据分配给新的空间
        a[tot].next=h;
        h=tot;
    }
}
```

同样，输出、插入、删除等插入思路与单链表的原理一致，区别仅是用数组的方式来实现代码。

练习：

使用静态链表构建在线排序，输入数值，构建成有序链表，将结果输出。

【代码】

```
#include<iostream>
using namespace std;
struct node{
    int data;
    int next;
}a[1000];
int n,x,head=-1,tot;
int search(int x){
    if(head==-1) return -1;
    if(a[head].data>x) return -1;
    int p=head;
    while(a[p].next!=-1&&a[a[p].next].data<x){
        p=a[p].next;
    }//注意：找到需要插入位置的前一位
    return p;
}
void insert(int x){
    a[++tot].data=x;
    int p=search(x);//查询插入位置
    if(p==-1){//头部插入
        a[tot].next=head;
        head=tot;
    }else{
```

```
        a[tot].next=a[p].next;
        a[p].next=tot;
    }
}
void print(int p){//静态链表输出
    while(p!=-1){
        cout<<a[p].data<<" ";
        p=a[p].next;
    }
    cout<<endl;
}
int main(){
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>x;
        insert(x);
    }
    print(head);
}
```

List标准库

STL中的List是一种双向链表，可高效进行插入和删除元素。

使用前必须调用list头文件

```
#include<list>
```

List的定义：

```
list< Node > lst;
```

List中还有多个函数可供调用：

函数名	功能	复杂度
Size()	返回表的元素数	O(1)
Begin()	返回指向表开头的迭代器	O(1)
End()	返回指向表末尾的迭代器	O(1)
Push_front(x)	在表的开头添加元素x	O(1)
Push_back(x)	在表的末尾添加元素x	O(1)
Pop_front()	删除表头元素	O(1)
Pop_back()	删除表尾元素	O(1)
Insert(p,x)	在位置p插入元素x	O(1)
Erase(p)	删除位置p的元素	O(1)
Clear()	删除表中所有元素	O(1)

其中需要注意的是，虽然这部分函数的时间复杂度为O(1)，但是插入和删除函数使用前需要先查找到待处理的位置p，这个准备工作所需时间接近O(n)。

list的运用中要注意输出的操作，需要定义一个迭代器，其含义与指针类似。

举个例子：

```
int * p;
list<int>::iterator it;
```

在上例两个变量定义中，指针p的类型为"int *"，指向的是int类型，同理，迭代器it的类型为list::iterator，其中iterator和指针定义中的*类似，it所指向的类型为list。

如果我们定义一个队列:

```
queue<int> q;
```

那么这个队列的迭代器可以如下定义：

```
queue<int>::iterator it;
```

【代码】

```
#include<iostream>
#include<list>
using namespace std;
list<int> lst;
list<int>::iterator it;//定义迭代器it
int n,x;
int main(){
    cin>>n;
    for(int i=1;i<=n;i++){
        lst.push_back(x);
    }
    for(it=lst.begin();it!=lst.end();it++){
        //需要注意的是循环条件it!=lst.end()，不能用大于小于来判断，必须用不等于
        cout<<*it<<" ";//注意it为指针
    }
}
```

A.字符串

【题目描述】

现在给一个字符串，你要做的就是当这个字符串中存在两个挨着的字符是相同的时就将这两个字符消除。需要注意的是，当把这两个字符消除后，可能又产生一对新的挨着的字符是相同的。比如，初始的字符串是abcddc，dd是两个挨着的相同的字符，当把"dd"消除后，得到的字符串是abcc，这时cc又是两个挨着的相同的字符，所以又应该把cc消除。重复以上操作直到剩下的串中不存在两个挨着的字符是相同的为止，输出最终剩下的串。另外需要注意的是，多对相同字符的消除顺序是不会对答案产生影响的，可以证明最后他们都会达到唯一的结果，比如，对于初始字符串adccdeed，无论是adccdeed->addeed->aeed->ad还是adccdeed->adccdd->adcc->ad，最终的输出结果都是ad。

【输入】

输入的第一行，包含一个字符串，为初始字符串，所有的字符均为小写字母。

【输出】

输出为一行，包含一个字符串，为执行多次消除操作后最终剩下的字符串。

样例输入	样例输出
adccdeed	ad

【提示】

对于100%的数据，字符串的长度在1到200000之间。

【分析】本题也可通过栈结构来解决，因为在list的使用中通过在头或尾的插入，也可以实现队和栈的功能。利用链表，每次将数据与表尾的值进行判断是否相等，相等则将表尾元素删除，不相等则从表尾插入。

【代码】

```
#include<iostream>
#include<list>
using namespace std;
list<char> l;
list<char>::iterator it;//迭代器,对链表进行遍历
char ch;
int main(){
    cin>>ch;
    l.push_back(ch);
    while(cin>>ch){
        if(ch==l.back()) {
            l.pop_back();
        }else{
            l.push_back(ch);
        }
    }
    for(it=l.begin();it!=l.end();it++){
        cout<<*it;
    }
}
```

B.打字机

【题目描述】

现在有一台打字机，打字机只有'a'-'z'和'L'(光标往左移动一格),'R'（光标往右移动一格)这28个按钮，现在给出按键顺序，问最后打出来的字符串是什么。注意：光标到了已经打出的字符串最左端的时候，即使按'L'光标也不会往左移动一格；最右端也是，即使按'R'光标也不会往右移动一格）

【输入】

一行只有'a'-'z'和'L','R'的字符串

【输出】

一行，为打出来的字符串

样例输入	样例输出
abLcd	acdb
icpLLLLLacmRRRRRRRRRRRRc	acmicpc

【分析】

本题充分体现了链表使用的优势，避免在插入时大量数据的移动。

【代码】

```
#include<iostream>
#include<list>
using namespace std;
list<char> l;
list<char>::iterator it;
char ch;
int main(){
    it=l.begin();
    while(cin>>ch){
        if(ch=='L'&&it!=l.begin()){//注意边界限制
            it--;
        }
        if(ch=='R'&&it!=l.end()){
            it++;
        }
        if(ch>='a'&&ch<='z'){
            l.insert(it,ch);//注意：插入后it会自动往后跳
        }
    }
    for(it=l.begin();it!=l.end();it++) cout<<*it;
}
```

C.饭堂插队

【题目描述】

在饭堂排队真是太累了，于是同学们就想插队。
每个班的同学因为熟悉，所以可以互相插队。具体地说，饭堂有一个窗口（惨啊），中午前，饭堂是空的。当一个人到来时，他会找到他们班的在队中最前的一个同学，然后排在他的前面（哈哈，这样后面的人发现不了），如果没有自己班的人，只好排在最后了。饭堂的工作人员已经知道同学到达饭堂的情况，他们需要你帮他们重现这个中午的情况。
总共会有两种事件。
A x y；
x班的y号同学到了饭堂，这种情况无需输出（同一个班同一个学号可能会来两次）。
B；
将队伍中第一个同学处理掉（以后来的同班同学将无法发现他）。
输出该同学的班级和学号。

【输入】

一个整数m表示有m个时间。
接下来m行描述m个事件。
A x y或B

【输出】

若干行，每行对应一个B，输出处理的人的班级和学号，用空格隔开。如果此时队伍为空，输出”no one”（不带引号）。

样例输入	样例输出
1 B A 1 1 A 2 1 A 1 2 B B B	no one 1 2 1 1 2 1
15 A 1 66494298 B A 0 89114043 A 1 64513030 A 0 11463757 A 1 95868919 A 0 20016348 A 1 13171423 B B B B B B B	1 66494298 0 20016348 0 11463757 0 89114043 1 13171423 1 95868919 1 64513030 no one

【提示】

【解释】 1班2号插队插到了一班一号前。
【数据范围】
%10 $1 <= m <= 100$
%30 $1 <= m <= 1000$
%100 $1 <= m <= 1000000$
所有班级的标号不超过10^6,所有学号不超过10^8。并且非负整数。

【分析】本题要注意处理好代码的细节，根据题意进行模拟即可。实现过程利用队列来维护队伍中存在的班级及顺序，利用list的表头插入模拟插队过程。

【代码】

```
#include<iostream>
#include<list>
#include<queue>
#include<cstdio>
using namespace std;
int m,c,x,y;
queue<int> q;
list<int> mylist[1000001];
```

```
bool checkc[1000001];
char ch[2];
int main(){
    scanf("%d",&m);
    for(int i=1;i<=m;i++){
        scanf("%s",&ch);//注意读入方式
        if(ch[0]=='B'){
            if(q.empty()) printf("no one\n");//队伍为空
            else{
                c=q.front();//队首的班级
                printf("%d %d\n",c,mylist[c].front());
                mylist[c].pop_front();//处理队首班级的首位
                if(mylist[c].empty()){//处理后为若为空，则本班标记从队中删除。班级标记为false
                    q.pop();checkc[c]=false;
                }
            }
        }else{
            scanf("%d%d",&x,&y);
            if(checkc[x]){//本班有人在队
                mylist[x].push_front(y);//插队
            }else{
                q.push(x);//本班排队尾
                checkc[x]=true;//本班标记为true
                mylist[x].push_front(y);//本班首位入list
            }
        }
    }
}
```

D.乐乐的数字

【题目描述】

乐乐做完数学作业，突发奇想定义了一种新的数：乐乐数。乐乐把n个数排成一行，一个数的“乐乐数”是指：在这个数的左边且比它小的数中最靠近它（即最靠右）的那个数。依次给出这n个数，请求出所有这n个数相对应的“乐乐数”。

【输入】

数据的第一行是一个正整数n，表示一共有多少个数。
第二行有n个用空格隔开的正整数，它们从左至右给出了数列中的n个数。这些数保证小于2^31。

【输出】

输出一行用空格隔开的n个数。
这些数对应于输入数据中的数的“乐乐数”。如果输入中某个数没有“乐乐数”（即它左边的数都不比它小），请输出0。

样例输入	样例输出
7 3 1 2 7 6 7 4	0 0 1 2 2 6 2
5 5 6 3 10 9	0 5 0 3 3

【分析】本题有多种解题方式，可利用线性动态规划解决，也可利用栈解决。本题解从利用List维护序列的单调性，即可快速找到“乐乐数”

【代码】

```
#include<iostream>
#include<list>
#include<cstdio>
using namespace std;
list<int> lst;
int n,x;
int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%d",&x);
        if(lst.size()==0){
            printf("0 ");
            lst.push_front(x);
        }else{
            while(lst.size()>0){
                if(lst.front()<x) break;
                lst.pop_front();
            }
            if(lst.size()!=0) printf("%d ",lst.front());
            else printf("0 ");
            lst.push_front(x);
        }
    }
}
```

E.重新排列

【题目描述】

FJ带他的N头奶牛去参加比赛, 比赛前N头奶牛自觉的排好队, 每头奶牛以它自己名字的首字母给出. 但FJ对他们的顺序不满意,现在FJ想要把奶牛们重新排列,以达到他们的队列在字典顺序中最小. FJ的操作是这样的: 他打算把奶牛从旧队列中一个一个拉出来, 排到一个新队列去(刚开始新队列为空),他每次只能把旧队列目前剩下的第一位奶牛或最后一位奶牛拉出来, 排到新队列的最后一位去。

【输入】

第一行: 一个整数N , 1 <= N <= 2,000。
接下来有N行,每行一个大写字母: 'A'---'Z'。
这N行字母就是题目描述的奶牛们们自觉排好队的旧队列。

【输出】

每行输出80个字母, 最后一行若不够80个字母,则按实际个数输出。

样例输入	样例输出
6 A C D B C B	ABCBCD

【提示】

说明: 6头牛的旧队列: ACDBCB
样例输出: ABCBCD

【分析】模拟操作，当首尾不相等时，输出字典序较小的那一位。当首尾相等时，需要从两端往中间遍历，找到不相等的两位中字典序较小的一端。注意输出格式。

```
#include<iostream>
#include<list>
using namespace std;
int n;
char ch;
list<char> lis;
list<char>::iterator ifront;
list<char>::iterator iback;
int main(){
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>ch;
        lis.push_back(ch);
    }
    for(int i=1;!lis.empty();i++){
        if(lis.front()!=lis.back()){//首尾不相同情况
            if(lis.front()>lis.back()){
                cout<<lis.back();
                lis.pop_back();
            }
            else{
                cout<<lis.front();
                lis.pop_front();
            }
        }else{//首尾相同情况
            ifront=lis.begin();
            iback=lis.end();
            iback--;
            while(*ifront==*iback){
                ifront++;
                iback--;
                if(ifront==iback){//判断至仅剩一个元素的情况
                    while(!lis.empty()){
                        cout<<lis.front();
                        lis.pop_front();
                    }
                    return 0;
                }
```

```
            }
            if(*ifront<*iback){
                cout<<lis.front();
                lis.pop_front();
            } else{
                cout<<lis.back();
                lis.pop_back();
            }
        }
        if(i%80==0) cout<<endl;
    }
}
```

F.整理内存

【题目描述】

每新开一个应用程序时，总要占用一定的内存，如何给这个程序在内存中分配位置呢？一种方法就是：从前往后找，找到第一块放得下的可用空间，就把它安排在这里。并把这一片空间标记为不可用。应用程序退出时，就把那一块空间再标为可用。这样反复多次后，内存中就会出现很多小块的可用空间，但太小了，无法利用。这样就造成了浪费，并且有时总的可用空间是够用的，但都分散了，导致内存不足。这时就需要进行内存整理。
所谓内存整理，就是把内存中所有的程序全部向前移动，使它们全部"紧紧贴在一块儿"。例如，在0到10被占用，20到30被占用，其余空间空闲时，进行整理内存，结果是0到20被占用，其余空闲。
现在已知总内存量，以及应用程序的开始结束情况和它们占用的内存分别是多少。求如果按照上面的方案分配内存的话，需要多少次内存整理。
内存量统一以M为单位。

【输入】

第一行是内存总量T，和一个正整数n，表示共进行n次打开或关闭应用程序
的操作。($0 < T <= 10000.0$,$1 <= n <= 1000$)
以下n行，每行描述了一个操作。
每行第一个数是这个应用程序的编号D,($1 <= D <= 1000$)
如果这个编号是第一次出现，那么这行后面还有一个实数F，表示这个程序的内存占用量。(
$0 < F <= 10000.0$)
如果这个编号是第二次出现，那么此行结束，表示这个编号的程序退出了。

【输出】

共一行，共需整理内存的次数。如果内存整理了还不够用，则输出"Impossible"。（引号不需要输出）

样例输入	样例输出
100 4 1 30 2 50 1 3 50	1

【分析】本题理解题意进行模拟即可。建立链表，记录每次操作前空间的空余内存段。当遇到有新的应用程序打开，可通过迭代器对链表进行遍历，查看是否有空余的内存空间存储，如有则需将占用后的剩余空间重新插入链，将原元素删除。若不够，则需考虑进行空间的整理，实质上就是将所有元素的值求和，清空链表后将这个总值插入。而当有应用程序关闭时，则有相应的空余空间产生，需将其插入链。

【代码】

```
#include<iostream>
#include<list>
using namespace std;
int n,d,ans;
double t,a[1001];
bool p,imp;
list<double> lis;//记录空余内存段
list<double>::iterator it;
int main(){
    cin>>t>>n;
    lis.push_back(t);
    for(int i=1;i<=n;i++){
        cin>>d;
        p=false;
        if(a[d]!=0){
            lis.push_back(a[d]);
        }else{
            cin>>a[d];
            if(imp) continue;
            for(it=lis.begin();it!=lis.end();it++){//遍历判断是否有足够的空间可供
                if((int)(*it-a[d])>=0){//注意数据类型
                    if((int)(*it-a[d])>0)lis.push_back(*it-a[d]);
                    lis.erase(it);
                    p=true;
                    break;
                }
            }
            if(!p){//暂无足够的空间空闲的情况，进行内存空间整理
                double sum=0.0;
                for(it=lis.begin();it!=lis.end();it++){
                    sum+=*it;
                }
                lis.clear();
                if(sum<a[d]){
                    imp=true;
                    continue;
                }
                lis.push_back(sum);
                ans++;
            }
        }
    }
    if(imp) cout<<"Impossible";
    else cout<<ans;
}
```