数据结构-链表

信智学院　陈琰宏

主要内容

01 链表的存储方式(结构体、数组)

02 单链表的操作（增删改查等）

03 单链表与双链表

04 案例实现

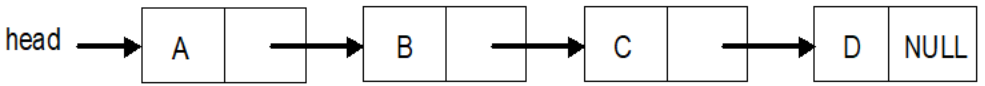
1 链表概念

链表是一种重要的基础数据结构，也是实现复杂数据结构的重要手段。它不按照线性的顺序存储数据，而是由若干个同一结构类型的“结点”依次串接而成的，即每一个结点里保存着下一个结点的地址（指针）。

链表又分单向链表，双向链表以及循环链表等。

1.1 单链表-结构体表示

单向链表的结构



使用结构的嵌套来定义单向链表结点的数据类型。如：

```
struct Node{
    ElementType   Data;
    struct Node *Next;
};
// ElementType代表所有可能的数据类型
```

```
struct Node *p;
p = (struct Node *) malloc(sizeof(struct Node));
```

类型定义typedef

除了使用C语言提供的标准类型和自己定义的一些结构体、枚举等类型外，还可以用typedef语句来建立已经定义好的数据类型的别名。

typedef *原有类型名* *新类型名*

```
typedef  struct Node * NodePtr;
```

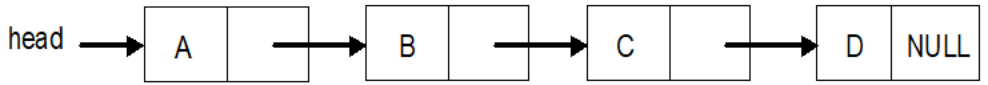
typedef struct Node * NodePtr;

这样，Reverse函数头就可以写成：

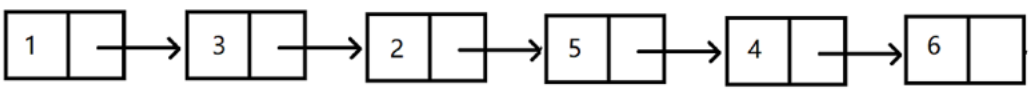
NodePtr Reverse(NodePtr L)

```
NodePtr  Reverse( NodePtr L )
/*  struct Node *Reverse(struct Node *L)  */
{   struct Node *p, *q, *t;
    p = L, q = NULL;
    while ( p != NULL ) {
            t = p->Next;
            p->Next = q;  q = p;
            p = t;
    }  return q;
}
```

1.2 单链表-数组表示



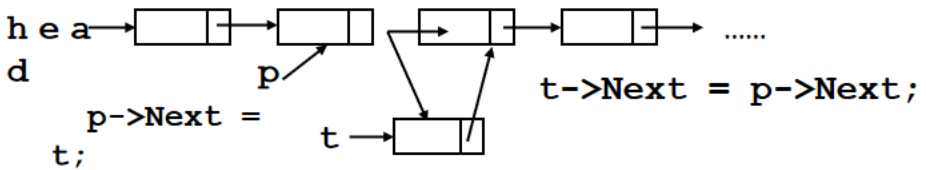
head 表示头节点的下标
e[i] 表示节点 i 的值
ne[i] 表示节点 i 的 next 指针是多少
idx 存储当前已经用到了哪个节点



e[1]=1　　e[2]= 3　表示当前节点的值
ne[1]=2　　ne[2]=3　表示当前节点next域指向的下一个节点编号

2.1 单链表操作-插入结点

（1） 插入结点（p之后插入新结点t）



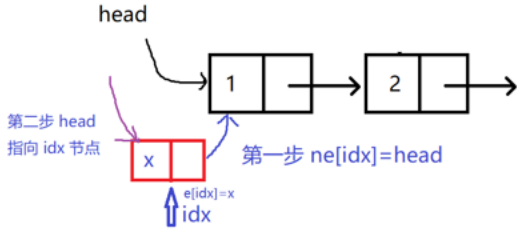

```
1  s=new LinkList();
2  //   malloc(sizeof(struct Node));
3  s->data=e;          //创建新结点*s,其data域置为e
4  s->next=p->next;    //将*s结点插入到*p结点之后
5  p->next=s;
```

单链表操作-插入结点 （结构体实现）

bool ListInsert(int i, T e)　//插入数据元素

```
{    int j=0; LinkList  *s, *p;
     if (i<1) return false;              //i<1时i错误,返回false
     p=head;                  //p指向头结点,j置为0(即头结点的序号为0)
     while (j<i-1 && p!=NULL)  //查找第i-1个结点*p
     {    j++;
          p=p->next;
     }
     if (p==NULL) return false;   //未找到第i-1个结点,返回false
     else   //找到第i-1个结点*p,插入新结点并返回true
     {
          s=new LinkList();    //   malloc(sizeof(struct Node));
          s->data=e;                    //创建新结点*s,其data域置为e
          s->next=p->next;              //将*s结点插入到*p结点之后
          p->next=s;
          return true;
     }
}
```

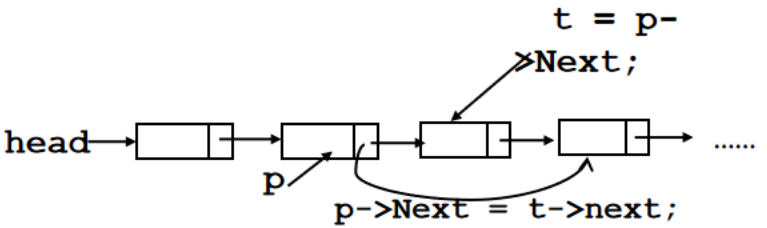
单链表-插入结点（数组表示）




```
1  //* 将 x 插到头节点
2  void add_to_head(int x)
3  {
4    e[idx]  = x;     //将第一个的值 变成我们插入的值
5    ne[idx] = head; //当前点指向 head 后的点
6    head    = idx;  //将 head 指到 当前的头部
7    idx++;          //移到下一个
8  }
```

```
9
10 //*将 x 这个点 插入到下标是 k 的点后面
11 void add(int k, int x)
12 {
13   e[idx]  = x;      //先把 x 这个值存下来
14   ne[idx] = ne[k]; //把新点的指针插入 k这个点指向的下一个位置
15   ne[k]   = idx;    //把 k 指向 要插入节点的位置
16   idx++;
17 }
```

2.2 单链表-删除结点

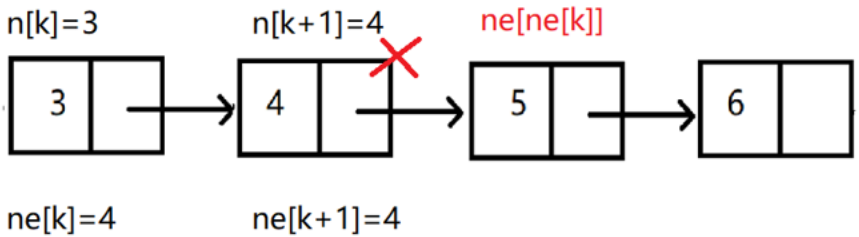



```
1  t=p->next;          //q指向第i个结点
2  p->next=t->next;      //从单链表中删除*q结点
3  delete q;           //释放*q结点
```

单链表-删除结点

```
bool ListDelete(int i)          //删除数据元素
{   int j=0; LinkList *q, *p;
    if (i<1) return false;              //i<1时i错误,返回false
    p=head;                             //p指向头结点,j置为0(即头结点的序号为0)
    while (j<i-1 && p!=NULL)  //查找第i-1个结点*p
    {   j++;
        p=p->next;
    }
    if (p==NULL) return false;  //未找到第i-1个结点,返回false
    else                                //找到第i-1个结点*p
    {   q=p->next;                      //q指向第i个结点
        if (q==NULL) return false;          //若不存在第i个结点,返回false
        p->next=q->next;            //从单链表中删除*q结点
        delete q;                   //释放*q结点
        return true;                //返回true表示成功删除第i个结点
    }
}
```

单链表-删除结点




```
1  //* 将位置 k 后面的点删除
2  void remove(int k)
3  {
4      ne[k] = ne[ne[k]];
5  }
```

2.3 单链表-初始化

```
LinkList  MakeEmpty()          //创建一个空单链表
{    head=new LinkList();
      head->next=NULL;
     return head;
}
```

```
1  void init()
2  {
3    head = -1;
4    idx  = 0;
5  }
```

2.4 单链表-遍历

```
void DispList() //输出单链表所有结点值
{     LinkList *p;
      p=head->next;              //p指向开始结点
      while (p!=NULL)            //p不为NULL,输出*p结点的data域
      {  cout << p->data << " ";
         p=p->next;              //p移向下一个结点
      }
      cout << endl;
}

while(cnt<m){
        x=ne[x];cnt++;
 }
```

2.5 单链表-求长度

```
int ListLength()    //求单链表数据结点个数
{
    int i=0; LinkList  *p;
     p=head;                  //p指向头结点,n置为0(即头结点的序号为0)
     while (p->next!=NULL)
     {    i++;
          p=p->next;
     }
     return (i);              //循环结束,p指向尾结点,其序号i为结点个数
}
```

2.6 单链表的建立

有两种常见的插入结点方式：

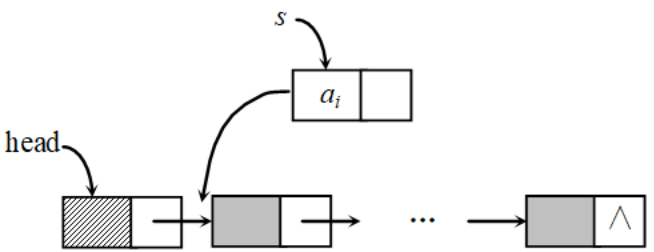
（1）在链表的头上不断插入新结点；

（2）在链表的尾部不断插入新结点。

如果是后者，一般需要有一个临时的结点指针一直指向当前链表的最后一个结点，以方便新结点的插入。

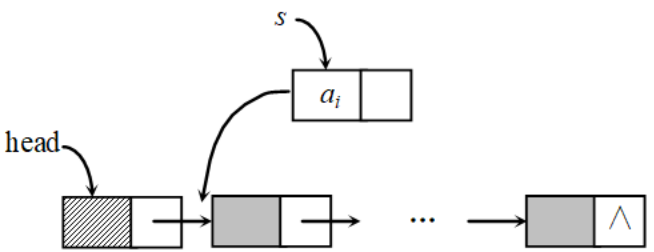
2.6.1 头插法

该方法从一个空表开始，读取数组*a*中的元素，生成新结点*s*，将读取的数据存放到新结点的数据域中，然后将新结点*s*插入到当前链表的表头上，如图所示。重复这一过程直到*a*数组的所有元素读完为止。

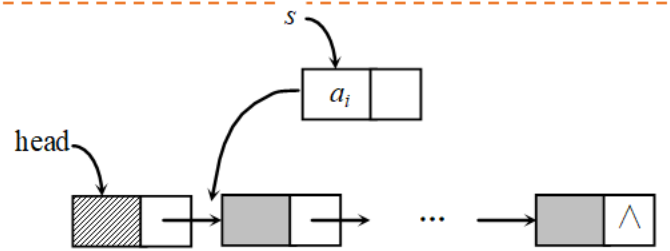


2.6.1 头插法

该方法从一个空表开始，读取数组*a*中的元素，生成新结点*s*，将读取的数据存放到新结点的数据域中，然后将新结点*s*插入到当前链表的表头上，如图所示。重复这一过程直到*a*数组的所有元素读完为止。



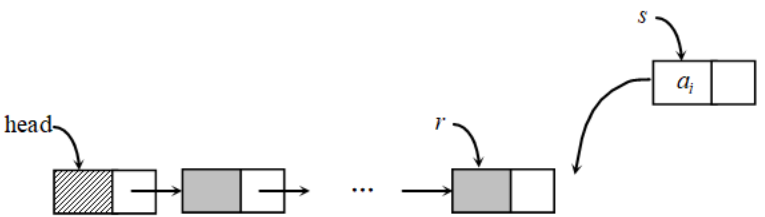
2.6.1 头插法




```
void CreateListF(T a[],int n)          //头插法建立单链表
{    LinkList*s; int i;
     head->next=NULL;                          //将头结点的next域置为NULL
     for (i=0; i<n; i++)                       //循环建立数据结点
     {    s=new LinkList();                 // malloc(sizeof(struct Linklist))
          s->data=a[i];                        //创建数据结点*s
          s->next=head->next; //将*s结点插入到开始结点之前,头结点之后
          head->next=s;
     }
}
```

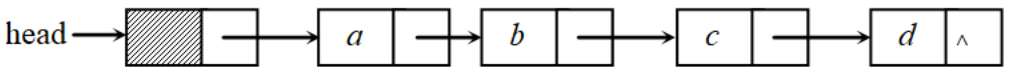
2.6.2 尾插法

头插法建立链表虽然算法简单，但生成的链表中结点的次序和原数组元素的顺序相反。若希望两者次序一致，可采用尾插法建立。该方法是将新结点*s*插到当前链表的表尾上，为此必须增加一个尾指针*r*，使其始终指向当前链表的尾结点



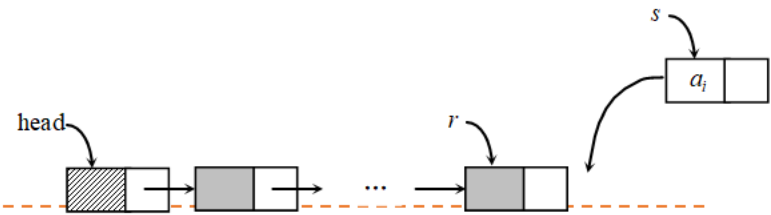
2.6.2 尾插法

若数组*a*包含4个元素'*a*'、'*b*'、'*c*'和'*d*'，调用CreateListR(a,4)建立的单链表如图所示。从中看到，尾插法建立的单链表中数据结点的次序与*a*数组中的次序正好相同。



2.6.2 尾插法

```
void CreateListR(T a[],int n)    //尾插法建立单链表
{     LinkList *s,*r; int i;
      r=head;                    //r始终指向尾结点,开始时指向头结点
      for (i=0; i<n; i++)        //循环建立数据结点
      {   s=new LinkList ();
          s->data=a[i];          //创建数据结点*s
          r->next=s;             //将*s结点插入*r结点之后
          r=s;
      }
      r->next=NULL;              //将尾结点的next域置为NULL
}
```



练习

1. 链表不具备的特点是（　　）
A.可随机访问任何一个元素　　B.插入、删除操作不需要移动元素
C.无需事先估计存储空间大小　　D.所需存储空间与存储元素个数成正比

2. 在单链表中，若p所指的结点不是最后结点，在p之后插入s所指结点，则执行（　）
A) s->next=p; p->next=s;　　B) s->next=p->next; p=s;
C)　s->next=p->next; p->next=s;　　D) p->next=s; s->next=p;

3. 设h为不带头结点的单向链表。在h的头上插入一个新结点t的语句是：（　）
A) h=t; t->next=h->next;　　B) t->next=h->next; h=t;
C) h=t; t->next=h;　　D) t->next=h; h=t;

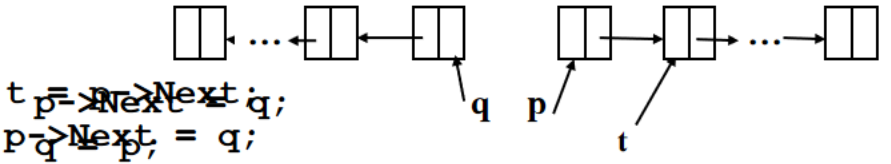
A　C　D

[例] 给定一个单链表L，请设计函数Reverse将链表L就地逆转，即不需要申请新的结点，将链表的第一个元素转为最后一个元素，第二个元素转为倒数第二个元素……

```
struct Node *Reverse(struct Node *L)
{ struct Node *p, *q, *t;
  p = L; q = NULL;
  while (p) {
    t = p->Next;
    p->Next = q;
    q = p;
    p = t;
  }
  return q;
}
```

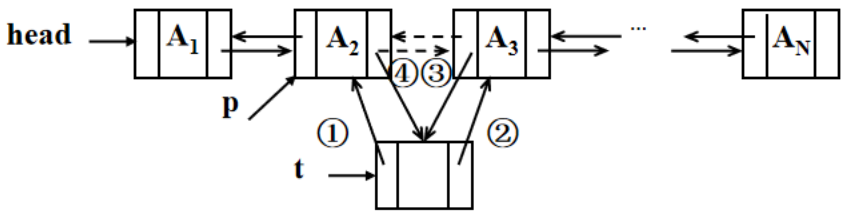
【分析】基本思路是：

- 利用循环，从链表头开始逐个处理。
- 如何把握住循环不变式。（循环不变式表示一种在循环过程进行时不变的性质，不依赖于前面所执行过程的重复次数的断言。）
- 在每轮循环开始前我们都面临两个序列，其中p是一个待逆转的序列，而q是一个已经逆转好的序列，如下图。
- 每轮循环的目的是把p中的第一个元素插入到q的头上，使这轮循环执行好后，p和q还是分别指向新的待逆转序列和已经逆转好的序列。



3.1 双向链表

双向链表也叫双链表，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。**双向链表的插入、删除和遍历基本思路与单向链表相同，但需要同时考虑前后两个指针。**




```
struct DNode {
        ElementType Data;
        struct DNode *Next;
        struct DNode *Previous;
} *p,*t;
```

指针操作顺序：

① t->Previous = p;

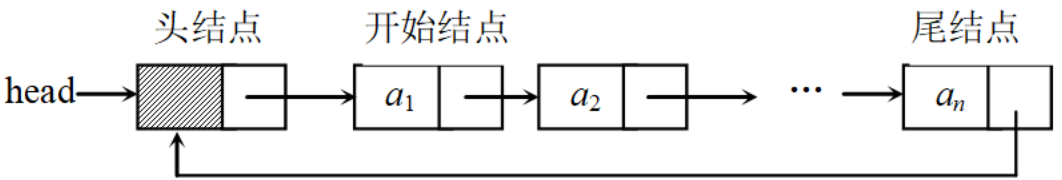
② t->Next = p->Next;

③ p->Next->Previous = t;

④ p->Next = t;

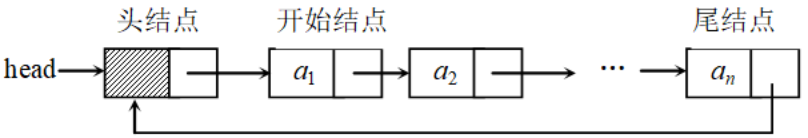
3.2 循环链表

带头结点head的循环单链表如下图所示，表中尾结点的指针域不再是空，而是指向头结点，整个链表形成一个环。其特点是从表中任一结点出发均可找到链表中其他结点。



3.2 循环链表

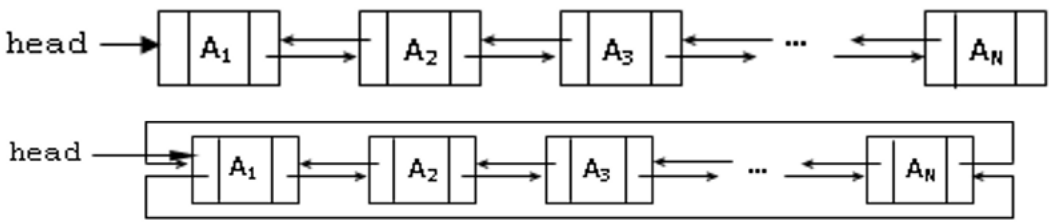
循环单链表的基本运算实现算法与非循环单链表的相似，只是对表尾的判断作了改变。例如，在循环单链表head中，判断表尾结点*p*的条件是p->next==head。



3.2 双向循环链表

如果将双向链表最后一个单元的Next指针指向链表的第一个单元，而第一个单元的Previous指针指向链表的最后一个单元，这样构成的链表称为双向循环链表。

```
struct Node{
    ElementType   Data;
    struct Node *Next;
    struct Node * Previous;
};
```



双向链表中有两个指针域llink和rlink，分别指向该结点的前驱及后继。设p指向链表中的一个结点，它的左右结点均非空。现要求删除结点p，则下面语句序列中错误的是（ ）。

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

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

C. p->rlink->llink = p->llink;
   p->rlink->llink->rlink = p->rlink; delete p;

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

A

1.3.7.1 [1115]多项式相加

一条单链表可以表示一个一元多项式，每个节点包含三个域：指数、系数和后继节点（指针或引用），如多项式$3X^4-6X^2+5X-10$的单链表。给定两个多项式，实现两个多项式相加算法。

输入　第一行输入包含两个整数m,n;后续为m行 和 n行数据。m，n分别代表两个多项式的项数。后续每一行代表多项式的项，包含a,b两个数据，表示该项的系数和指数。
输出　从较高指数到较低指数，依次输出求得的和。
每行一项，格式与输入相同，但无需输出项数，系数为0的项也不输出。

样例输入

```
2 3
1 2
1 1
2 2
1 1
2 0
```

样例输出

```
3 2
2 1
2 0
```

1.3.7.1 [1155]多项式相加-数组模拟

```
#include <iostream>
using namespace std;
int ans[200];
int main()
{
    int n,m;
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++){
        int x,y;//输入一个表达式的各项参数
        scanf("%d%d",&x,&y);
        ans[y]+=x;//y表示指数，相同指数的系数相加
    }
    for(int i=1;i<=m;i++){
        int x,y;////输入第二个表达式的各项参数
        scanf("%d%d",&x,&y);
        ans[y]+=x;
    }
    for(int i=100;i>=0;i--){
        if(ans[i]!=0){
            printf("%d %d\n",ans[i],i);
        }
    }
}
```

1.3.7.1 [1115]多项式相加

方法2：采用顺序存储结构表示多项式的非零项。

每个非零项 $a_i x^i$ 涉及两个信息：指数 i 和系数 a_i，
可以将一个多项式看成是一个（a_i, i）二元组的集合。

例如： $P_1(x) = 9x^{12} + 15x^8 + 3x^2$ **和** $P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$

- **最后得到的结果多项式是：（(26,19), (9,12), (11,8), (–13,6), (3,2), (82,0)）**

表示成： $P_2(x) = 26x^{19} + 9x^{12} + 11x^8 - 13x^6 + 3x^2 + 82$

数组下标i	0	1	2	……
系数	9	15	3	–
指数	12	8	2	–

(a) $P_1(x)$

数组下标i	0	1	2	3	……
系数	26	–4	–13	82	–
指数	19	8	6	0	–

(b) $P_2(x)$

1.3.7.1 多项式相加

方法3：采用链表结构来存储多项式的非零项。

每个链表结点存储多项式中的一个非零项，包括系数和指数两个数据域以及一个指针域，表示为：

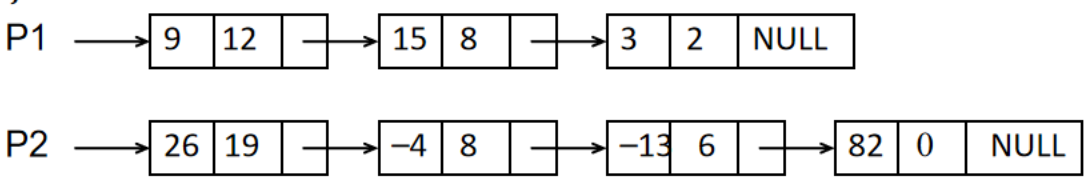

```
typedef struct PolyNode *Polynomial;
typedef struct PolyNode {
        int coef;tuoyuan
        int expon;
        Polynomial link;
}
```

例如：

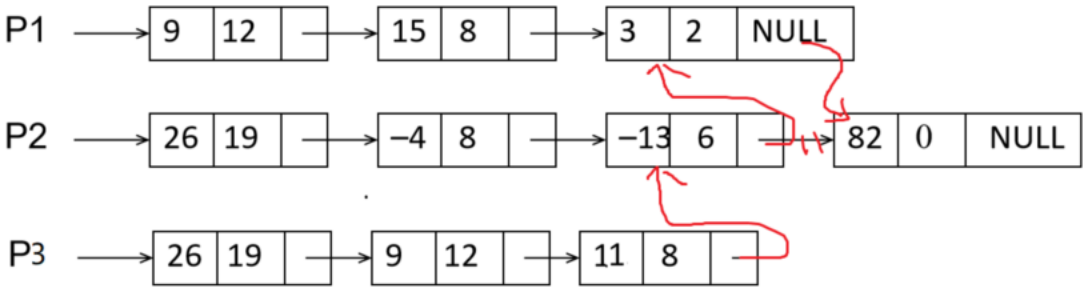
$$P_1(x) = 9x^{12} + 15x^8 + 3x^2$$

$$P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$$

链表存储形式为：

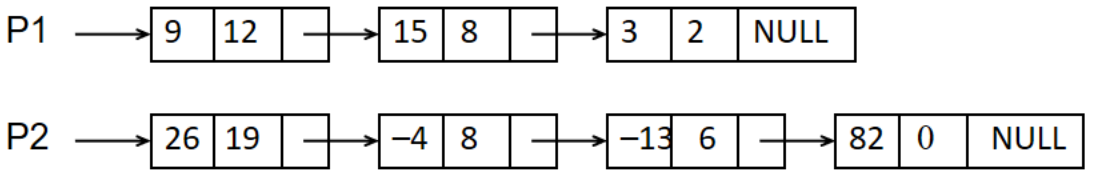


a. 总思路



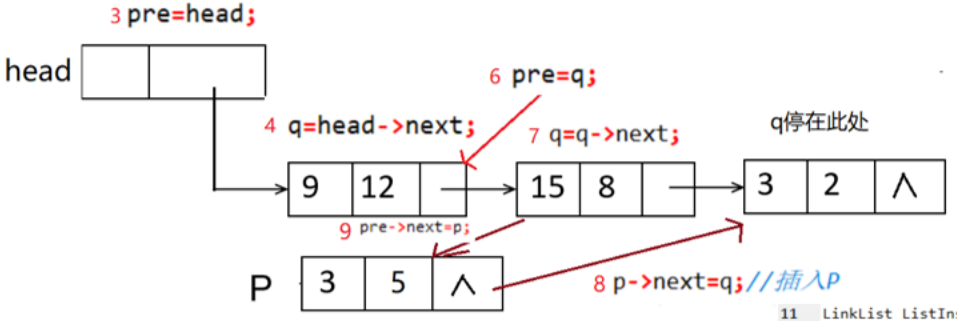
P3是一个虚拟链表，没有生成新的节点，就是在P1,P2上产生了一条新的链。

b. 节点定义




```
3  typedef struct LNode
4  {
5      int coef,expn;// coef系数  expn指数
6      struct LNode *next;
7  }LNode,*LinkList;
```

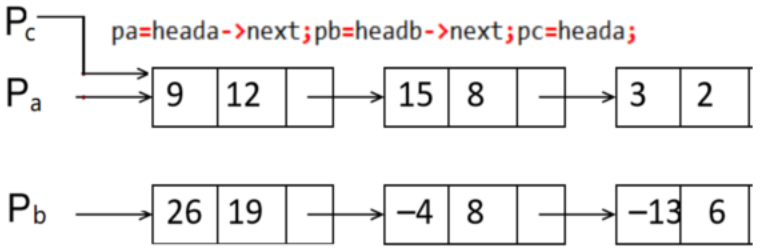
c. 创建链表



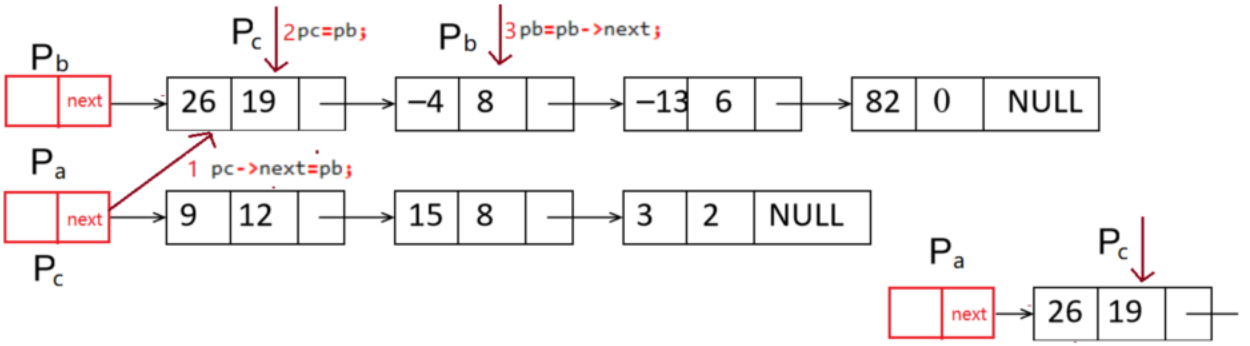

```
1 p=new LNode();//新建插入的点
2 cin>>p->coef>>p->expn;
5 while(q&&q->expn>p->expn)
  {//比较指数大小，如果新节点P的指数 <q.
   //如果没到末尾并且找到合适的位置
    6 pre=q;
    7 q=q->next;
  }
```

```
11  LinkList ListInsert(int n)
12  {//创建链表
13      LinkList head,p,q,pre,ttt;
14      head=new LNode();//新建头结点
15      head->next=NULL;//初始头结点的next为NULL
16      for(int i=0;i<n;i++)
17      {
18          p=new LNode();//新建插入的点
19          cin>>p->coef>>p->expn;
20          pre=head;//
21          q=head->next;
22          while(q&&q->expn>p->expn)
23          {//比较指数大小，如果新节点P的指数 <q.expn
24           //如果没到末尾并且找到合适的位置
25              pre=q;
26              q=q->next;
27          }
28          p->next=q;//插入P
29          pre->next=p;
30      }
31      return head;
32  }
```

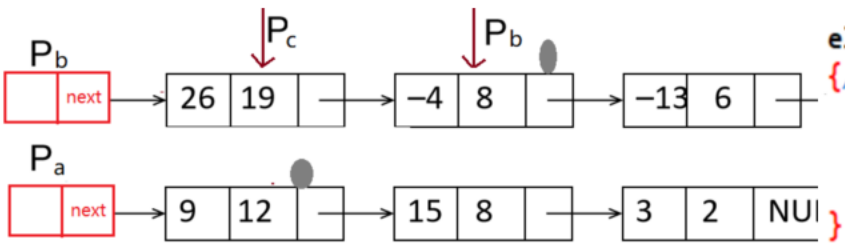
d. 相加操作（pa<pb)



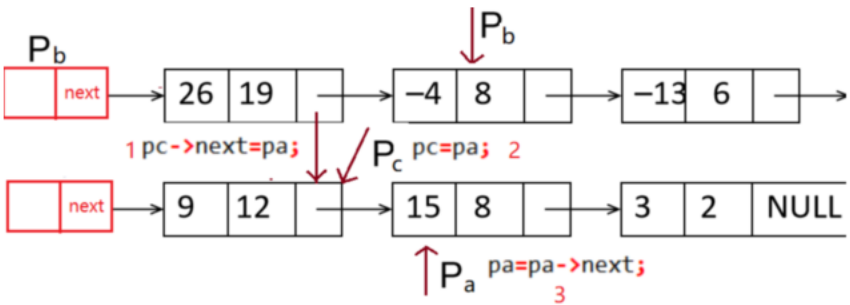

```
else if(pa->expn<pb->expn)
{//把pb项链接到pc链中
    pc->next=pb;
    pc=pb;
    pb=pb->next;
}
```

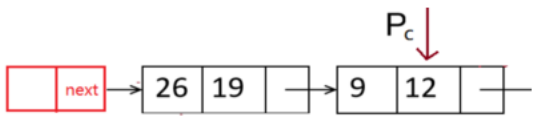


e. 相加操作（pa>pb)

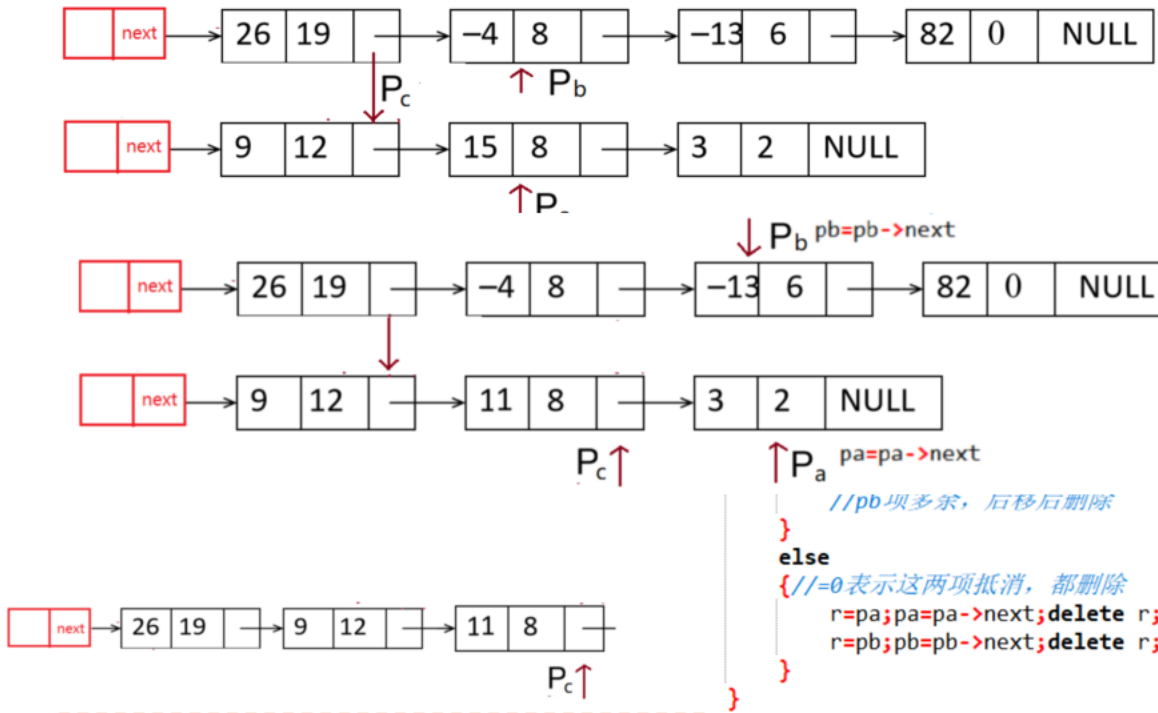



```
else if(pa->expn>pb->expn)
{//把pa项链接到pc链中
  1 pc->next=pa;
  2 pc=pa;
  3 pa=pa->next;
}
```





f. 相加操作（pa==pb）




```
        //pb项多余，后移后删除
    }
    else
    {//=0表示这两项抵消，都删除
        r=pa;pa=pa->next;delete r;
        r=pb;pb=pb->next;delete r;
    }
}
```

1.3.7.1 [1115]多项式相加-链表标准实现

```
#include<bits/stdc++.h>
using namespace std;
typedef struct LNode
{
    int coef,expn;// coef系数  expn指数
    struct LNode *next;
}LNode,*LinkList;
LinkList ListInsert(int n)
{//创建链表
void _plus(LinkList heada,LinkList headb)
{
void print(LinkList head)
{
int main()
{
    LinkList head1,head2;
    int m,n;
    cin>>m>>n;
    head1 = ListInsert(m);
    head2 = ListInsert(n);
    _plus(head1,head2);
    print(head1);
    return 0;
}
```

```
LinkList ListInsert(int n)
{//创建链表
    LinkList head,p,q,pre,tail;
    head=new LNode();
    head->next=NULL;
    tail=head;
    for(int i=0;i<n;i++)
    {
        p=new LNode();
        cin>>p->coef>>p->expn;
        pre=head;
        q=head->next;
        while(q&&q->expn>p->expn)
        {//没到末尾并且找到合适的位置
            pre=q;
            q=q->next;
        }
        p->next=q;
        pre->next=p;
    }
    return head;
}
```

1.3.7.1 [1115]多项式相加-链表标准实现

```
void _plus(LinkList heada,LinkList headb)
{
    int sum;
    LinkList pa,pb,pc,r;
    pa=heada->next;pb=headb->next;pc=heada;
    //pc初始为头节点，pa/pb指向第一个节点
    //pc为相加后新的链
    while(pa&&pb)//不为空即不到尾部
    {
        if(pa->expn==pb->expn)
        {
            sum=pa->coef+pb->coef;
            if(sum!=0)
            {
                pa->coef=sum;//相加形成新系数
                pc->next=pa;pc=pa;//pc链向后走一项
                pa=pa->next;//pa链 向后走一项
                r=pb;pb=pb->next;delete r;
                //pb项多余，后移后删除
            }
            else
            {//=0表示这两项抵消，都删除
                r=pa;pa=pa->next;delete r;
                r=pb;pb=pb->next;delete r;
            }
        }
        else if(pa->expn<pb->expn)
        {//把pb项链接到pc链中
            pc->next=pb;
            pc=pb;
            pb=pb->next;
        }
        else
        {//把pa项链接到pc链中
            pc->next=pa;
            pc=pa;
            pa=pa->next;
        }
    }
    pc->next=pa?pa:pb;
    //有一个后为空把剩下的链接到pc
}
```

1.3.7.1 [1115]多项式相加-链表标准实现

```
72  void print(LinkList head)
73  {
74      LinkList p;
75      p=head->next;
76      while(p!=NULL)
77      {
78          cout<<p->coef<<" "<<p->expn<<endl;
79          p=p->next;
80      }
81  }
```

4.1 [3204] 约瑟夫问题

编写一个程序求解约瑟夫（Joseph）问题。有n个小孩围成一圈，给他们从1开始依次编号，从编号为1的小孩开始报数，数到第m个小孩出列，然后从出列的下一个小孩重新开始报数，数到第m个小孩又出列，…，如此反复直到所有的小孩全部出列为止，求整个出列序列。

如当n=6，m=5时的出列序列是5，4，6，2，3，1。

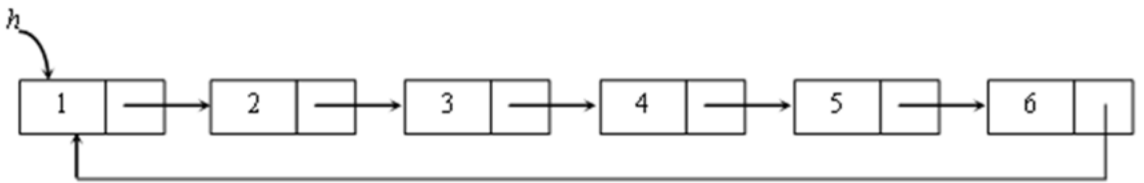
4.1.1 结构体链表解法

（1）设计存储结构

本题采用循环单链表存放小孩圈，其结点类型如下：

```
struct Child                          //小孩结点类型
{     int no;                         //小孩编号
      Child *next;            //指向下一个结点指针
};
```

依本题操作，小孩圈循环单链表不带头结点，例如，n=6时的初始循环单链表如图所示，h指向开始报数的小孩结点，初始时指向首结点。



n=6 时的初始循环单链表

4.1.1 结构体链表解法

```
1   void  Init( int n1,int m1) //建立有n1个结点的循环单链表
2   {
3       int i;
4       Child *p,*r; //r指向新建循环单链表的尾结点
5       n=n1; m=m1;
6       h=new Child();
7       h->no=1; //先建立只有一个no为1结点的单链表
8       r=h;
9       for (i=2;i<=n;i++)
10      {
11          p=new Child();  //建立一个新结点*p
12          p->no=i;        //新结点存放编号i
13          r->next=p; r=p; //将*p结点链到末尾
14      }
15      r->next=h; //构成一个首结点为h的循环单链表
16  }
```

4.1.1 结构体链表解法

```
1  void Jsequence()     //求解约瑟夫序列
2  {
3      int i,j;
4      Child *p,*q;
5      for (i=1;i<=n;i++)  //共出列n个小孩
6      {
7          p=h; j=1;
8          while (j<m-1) //从*h结点开始报数，报到第m-1个结点
9          {
10             j++;     //报数递增
11             p=p->next; //移到下一个结点
12         }
13         q=p->next; //q指向第m个结点
14         cout << q->no << " "; //该结点出列
15         p->next=q->next;     //删除*q结点
16         delete q;          //释放其空间
17         h=p->next;  //从下一个结点重新开始
18     }
19 }
```

4.1.1 结构体链表解法

设计如下主函数求解n=6，m=5的约瑟夫序列：

```
int main()
{

        cin>>n>>m;
        Init(n,m);
        Jsequence();
        cout << endl;
        return 0;
}
```

本程序的执行结果如下：

```
n=6,m=5的约瑟夫序列:5 4 6 2 3 1
```

4.1.2 数组链表解法

```
1   #include<bits/stdc++.h>
2   using namespace std;
3   const int N=2e5+10;
4   int ne[N];//存储下一个节点的编号
5   int n,m;
6   int x,cnt,peo;
7   void init()
8   {//x当前节点值  peo/cnt 计数变量
9       x=n,cnt=1,peo=1;
10  }
11  void add()
12  {//建立链表
13      for(int i=1;i<=n;i++)
14          ne[i]=i+1;//当前点i指向 i+1
15      ne[n]=1;//连成一个环
16  }
```

4.1.2 数组链表解法

```
17  void Remove()
18  {
19      while(peo<=n)
20      {
21          while(cnt<m){
22              x=ne[x];cnt++;
23          }//遍历下一个点
24          cout<<ne[x]<<' ';
25          ne[x]=ne[ne[x]];//删除节点
26          cnt=1;//重新计数
27          peo++;
28      }
29  }
30  int main()
31  {
32      cin>>n>>m;
33      init();
34      add();
35      Remove();
36      return 0;
37  }
```

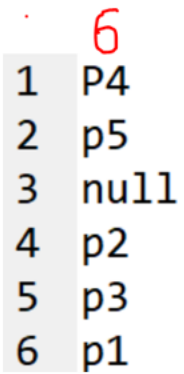
4.2 [1892] 链表结点的物理顺序与逻辑顺序

已知一个单向链表各结点在存储器中的物理顺序，以及各结点之间的指向关系，要求输出该链表各结点的逻辑顺序。

例如有6个结点，按这6个结点在存储器中的物理顺序依次编号为1～6。第1个结点的指针域为p4，表示它指向第4个结点。符号“∧”表示空指针。

因此，这个链表6个结点的逻辑顺序为：

6->1->4->2->5->3。



4.2 [1892] 链表结点的物理顺序与逻辑顺序

```
1  #include<iostream>
2  using namespace std;
3  struct node{//定义一个节点
4      int id;
5      node *next;
6  };
7
8  char ch[5];//输入的字符，如 p4
9  int n,digit[100];//digit[i]=j;表示 第i个节点指向第j个节点
10 int tail;//tail表示最后一个节点
```

4.2 [1892] 链表结点的物理顺序与逻辑顺序

```
11  int main(){
12      while(cin>>n){
13          if(n==0)break;
14          node *head,*t;
15          for(int i=1;i<=n;i++){//循环n次，输入n个字符串
16              cin>>ch;
17              if(ch[1]=='u'){//u即null，查找最后一个节点
18                  tail=i;//记录最后一个节点的数据域
19                  digit[i]=-1;//用-1表示空
20              }
21              else
22                  digit[i]=ch[1]-'0';//digit[i]=j;记录节点的指向关系
23          }//1获取数据
```

4.2 [1892] 链表结点的物理顺序与逻辑顺序

```
25          head=new node();// 新建节点
26          head->id=tail,head->next=NULL;//初始头节点
27          int j=1;
28          while(j<n){//扫描、建立链表的过程
29              for(int i=1;i<=n;i++){
30                  if(digit[i]==head->id){//反向扫描，digit[i]=j，即i->j
31                      t=new node();
32                      t->id=i,t->next=head;//建立 i->j
33                      head=t,j++;
34                  }
35              }
36          }//2 建立链表
37
38          while(head->next!=NULL){
39              cout<<head->id<<"->";
40              head=head->next;
41          }//3 输出
42          cout<<tail<<endl;
43      }
44      return 0;
45  }
```

4.3 [7287] 单链表

实现一个单链表，链表初始为空，支持三种操作：
(1) 向链表头插入一个数；
(2) 删除第k个插入的数后面的数；
(3) 在第k个插入的数后插入一个数
现在要对该链表进行M次操作，进行完所有操作后，从头到尾输出整个链表。
注意:题目中第k个插入的数并不是指当前链表的第k个数。例如操作过程中一共插入了n个数，则按照插入的时间顺序，这n个数依次为：第1个插入的数，第2个插入的数，…第n个插入的数。

4.3 [7287] 单链表

输入格式
第一行包含整数M，表示操作次数。
接下来M行，每行包含一个操作命令，操作命令可能为以下几种：
(1) “H x”，表示向链表头插入一个数x。
(2) “D k”，表示删除第k个输入的数后面的数（当k为0时，表示删除头结点）。
(3) “I k x”，表示在第k个输入的数后面插入一个数x（此操作中k均大于0）。
输出格式
共一行，将整个链表从头到尾输出。
数据范围
1≤M≤100000
所有操作保证合法。

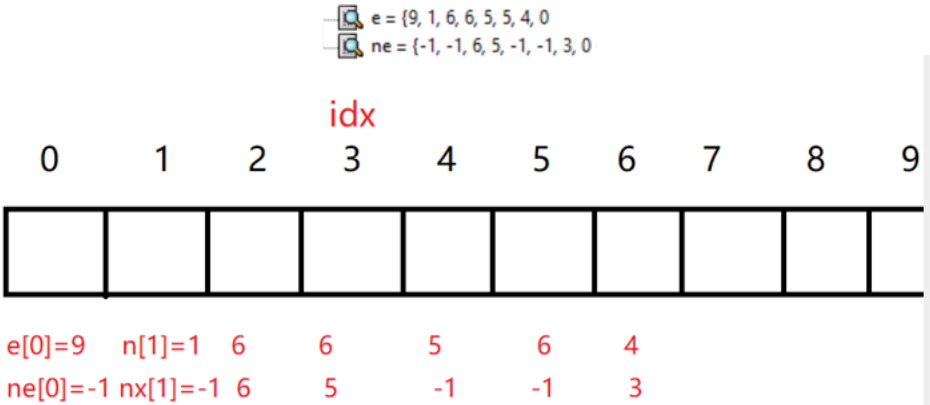
样例输入

```
10
H 9
I 1 1
D 1
D 0
H 6
I 3 6
I 4 5
I 4 5
I 3 4
D 6
```

样例输出

```
6 4 6 5
```

4.3 [7287] 单链表



样例输入

```
10
H 9
I 1 1
D 1
D 0
H 6
I 3 6
I 4 5
I 4 5
I 3 4
D 6
```

样例输出

```
6 4 6 5
```

4.3 [7287] 单链表

```cpp
#include <iostream>
using namespace std;
const int N = 1e5 + 10;
//* head 表示头节点的下标
//* e[i] 表示节点 i 的值
//* ne[i] 表示节点 i 的 next 指针是多少
//* idx 存储当前已经用到了哪个节点
int head, e[N], ne[N], idx;
//* 初始化
void init()
{
  head = -1;
  idx  = 0;
}
//* 将 x 插到头节点
void add_to_head(int x)
{
  e[idx]  = x;     //! 将第一个的值 变成我们插入的值
  ne[idx] = head; //! 因为是插入，所以当前的 head 是插入后的
  head    = idx;  //! 将 head 指到 当前的头部
  idx++;          //! 移到下一个
}
```

4.3 [7287] 单链表

```
23  //*将 x 这个点 插入到下标是 k 的点后面
24  void add(int k, int x)
25  {
26    e[idx]  = x;      //! 先把 x 这个值存下来
27    ne[idx] = ne[k]; //! 把新点的指针插入 k这个点指向的下一个位置
28    ne[k]   = idx;    //! 把 k 指向 要插入节点的位置
29    idx++;
30  }
31  //* 将位置 k 后面的点删除
32  void remove(int k)
33  {
34    ne[k] = ne[ne[k]];
35  }
36
```

4.3 [7287] 单链表

```
37  int main()
38  {
39    // todo 第k个插入的数 就是下标是 k-1的点
40    int m;
41    cin >> m;
42    init(); //! 初始化
43    while(m--)
44    {
45      int  x, k;
46      char op;
47      cin >> op;
48      if(op == 'H')
49      {
50        cin >> x;
51        add_to_head(x);
52      }
53      else if(op == 'D')
54      {
55        cin >> k;
56        if(k == 0)
57          head = ne[head];
58        else
59          remove(k - 1); // todo 下标是 k-1
60      }
61      else
62      {
63        cin >> k >> x;
64        add(k - 1, x); // todo 下标是 k-1
65      }
66    }
67    for(int i = head; i != -1; i = ne[i]) cout << e[i] << ' ';
68    cout << endl;
69    return 0;
70  }
```

今天的课程结束啦……

下课了…
同学们再见！