Part 1 线性表及其实现

- 例1 多项式的表示
  - 方法1：顺序存储结构直接表示
  - 方法2：顺序存储结构只表示非零项
  - 方法3：链表结构存储非零项
  - 启示
- 线性表 Linear List
- 线性表抽象数据类型描述
- 实现
  - 顺序存储实现
  - 主要操作实现
    1. 初始化（建立空的顺序表）
    2. 查找（查找值为 `X` 的元素的位置）
    3. 插入（在第 `i`（`1 ≤ i ≤ n + 1`）个位置（即 `Data[n - 1]`）插入一个值为 `X` 的新元素）
    4. 删除（删除第 `i`（`1 ≤ i ≤ n`）个位置上的元素）
  - 链式存储实现
  - 主要操作实现
    1. 求表长
    2. 查找
    3. 插入（在第 `i` 个节点后插入一个值为X的新节点，返回链表 `head`）
    4. 删除（删除链表第 `i` 个位置上的节点，返回链表 `head`）
- 广义表 Generalized List：表中元素可以是单元素或广义表
- 多重链表：链表中节点可能同时隶属于多个链
  - 例：矩阵存储

Part 2 堆栈 Stack

- 表达式求值之后缀表达式
- 堆栈的抽象数据类型描述
- 实现
  - 堆栈的顺序存储实现
  - 尝试一个数组实现两个堆栈，并充分利用空间
  - 堆栈的链式存储实现
- 中缀表达式转换为后缀表达式

Part 3 队列 Queue

- 队列的抽象数据类型描述
- 队列的顺序存储实现

队列的链式存储实现

编程练习

1. **两个有序链表序列的合并**

   函数接口定义：

   裁判测试程序样例：

   输入样例：

   输出样例：

   实现

2. 一元多项式的乘法与加法运算

   输入格式:

   输出格式:

   输入样例:

   输出样例:

   实现

3. Reversing Linked List 单链表翻转

   Input Specification:

   Output Specification:

   Sample Input:

   Sample Output:

   实现

4. **Pop Sequence**

   Input Specification:

   Output Specification:

   Sample Input:

   Sample Output:

   实现

Part 1 线性表及其实现

例1 多项式的表示

$$f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n \tag{1}$$

- 多项式的主要运算有 相加，相减，相乘等
- 多项式的关键数据有 多项式的项数n，各项的系数a_i及对应的指数i

方法1：顺序存储结构直接表示

`a[i]`表示项x^i的系数a_i

例如：　$f(x)=4x^5-3x^2+1$

表示成：

下标i	0	1	2	3	4	5	……
a[i]	**1**	**0**	**–3**	**0**	**0**	**4**	**……**
	1		-3x²			4x⁵	

多项式相加直接由两个数组对应位置相加即可

缺点：我们如果用这个方法表示一个系数为0的项数过多的多项式，会很浪费空间，例如$x+9x^{9000}$

方法2：顺序存储结构只表示非零项

我们定义一种结构体（二元组）来表示多项式非零项，包含两个属性，即系数与指数；这样，定义一个对应的结构数组就可以表示整个多项式，但是要注意这个结构数组的存储需要按照指数的递增或递减顺序存放，以便进行加减

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

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

(a) $P_1(x)$

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

(b) $P_2(x)$

这里假设是按照指数递减的顺序存储的结构体，两个多项式相加，只需要从头开始对比，指数大的直接放入结果，指数相同的系数相加再放入结果即可，例如

P1: (9,12), (15,8), (3,2)

P2: (26,19), (-4,8), (-13,6), (82,0)

P3: (26,19) (9,12) (11,8) (-13,6) (3,2) (82,0)

$$P_3(x) = 26x^{19} + 9x^{12} + 11x^8 - 13x^6 + 3x^2 + 82$$

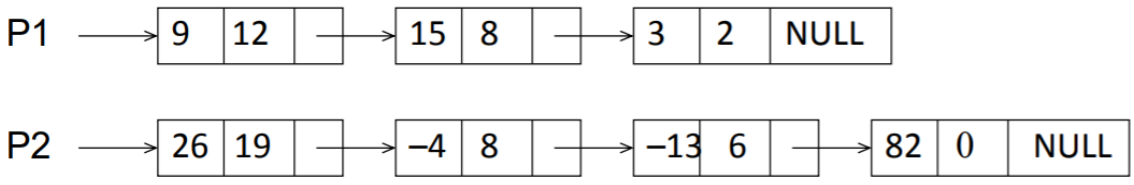
• 方法3：链表结构存储非零项

链表的每个节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和一个指数域

```
typedef struct PolyNode* Polynomial;
struct PolyNode{
    int coef;   // 系数
    int expon;  // 指数
    Polynomial link;    // 下一个节点的地址
}
```

例如

$$\begin{aligned} P_1(x) &= 9x^{12} + 15x^8 + 3x^2 \\ P_2(x) &= 26x^{19} - 4x^8 - 13x^6 + 82 \end{aligned} \tag{2}$$



• 启示

- 同一个问题可以有不同的表示（存储）方法（数组，链表）
- 有一类共性的问题：有序线性序列的组织和管理

线性表 Linear List

由**同类型数据元素**构成**有序序列**的线性结构

- 表中元素个数称为线性表的**长度**
- 线性表没有元素时，称为**空表**
- 表起始位置称为**表头**，结束位置称为**表尾**

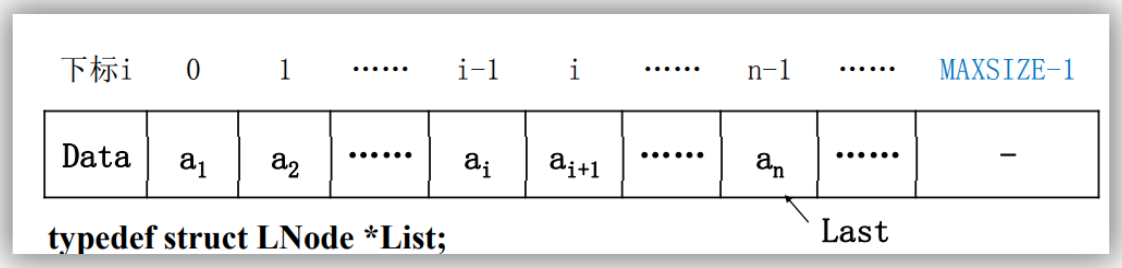
线性表抽象数据类型描述

- 类型名称：线性表 (List)
- 数据对象集：线性表是由$n \geq 0$个元素构成的有序序列
- 操作集：线性表L ∈ List，整数i表示位置，元素X ∈ ElementType，基本操作有

```
List MakeEmpty();    //初始化一个空线性表L
ElementType FindKth(int K, List L); //根据索引K，返回相应元素
int Find(ElementType X, List L);     //在L中查找X的第一次出现的位置
void Insert(ElementType X, int i, List L);  //在L的索引i前插入一个新元素X
void Delete(int i, List L); //删除索引为i的元素
int Length(List L); //返回线性表L的长度n
```

实现

顺序存储实现



```
typedef struct LNode* List;
struct LNode{
    ElementType Data[MAXSIZE];  //存放线性表的数组
    int Last;   // 线性表的最后一个元素的索引
};
struct LNode L;
List PtrL;
```

访问下标为 `i` 的元素：`L.Data[i]` 或 `PtrL->Data[i]`

线性表的长度：`L.Last + 1` 或 `PtrL->Last + 1`

• 主要操作实现

- 1. 初始化（建立空的顺序表）

```
List MakeEmpty()
{
    List PtrL;
    PtrL = (List)malloc(sizeof(struct LNode));
    PtrL->Last = -1;    //顺序表中存在一个元素是这里才是0
    return PtrL;
}
```

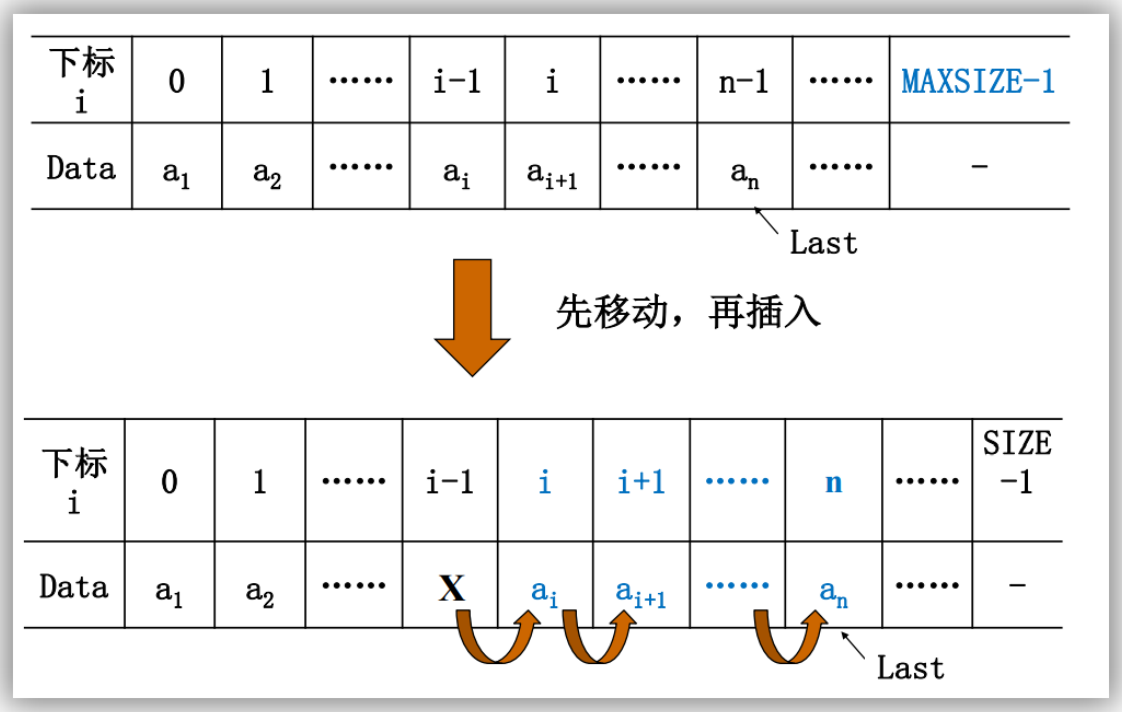
- 2. 查找（查找值为 `X` 的元素的位置）

```
 int Find(ElementType X, List PtrL)
 {
     int i = 0;
     while(i <= PtrL->Last && PtrL->Data[i] != X)
         i++;
     if(PtrL->Last <= i) // 没找到，返回-1
         return -1;
     else
         return i;   // 找到了，返回对应位置（索引）
 }
```

查找成功的平均比较次数是 `(n + 1) / 2`，平均时间性能为 `O(n)`

- 3. 插入（在第 `i`（`1 ≤ i ≤ n + 1`）个位置（即 `Data[n - 1]`）插入一个值为 `X` 的新元素）

先把要插入位置及其之后的元素统一向后移动一位之后，再在该位置插入元素

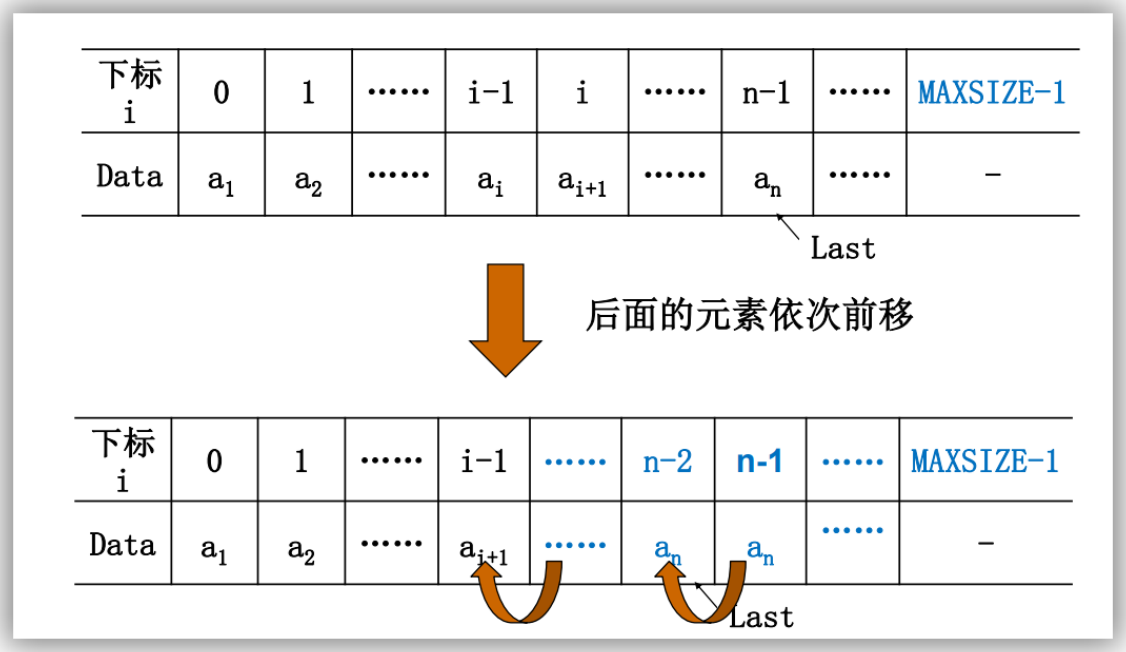



```c
void Insert(ElementType X, int i, List PtrL)
{
    if(PtrL->Last == MAXSIZE - 1)    // 表空间已满，不能插入
    {
        printf("表满，无法插入");  return;
    }
    if(i < 1 || PtrL->Last + 2 < i)
    {
        printf("位置不合法，无法插入");    return;
    }
    // 将 a[i]~a[n]倒序向后移动，倒序是为了避免覆盖
    for(int j = PtrL->Last; i - 1 <= j; j--)
        PtrL->Data[j + 1] = PtrL->Data[j];
    PtrL->Data[i - 1] = X;   // 新元素插入
    PtrL->Last++;    // 保证Last依然指向最后元素
}
```

平均移动次数为 `n / 2`，平均时间性能为 `O(n)`

- 4. 删除（删除第 `i`（`1 ≤ i ≤ n`）个位置上的元素）

将要删除位置后面的元素统一向前移动一位覆盖要删除的位置即可，这里需要顺序移动

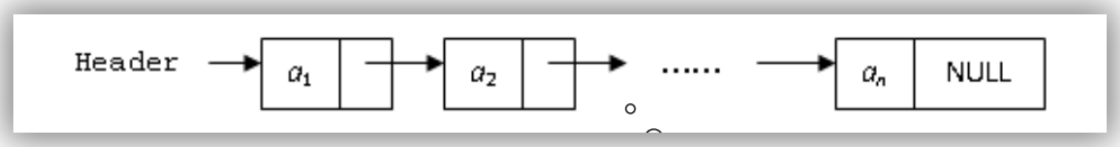



```c
void Delete(int i, List PtrL)
{
    if(i < 1 || PtrL->Last + 1 < i)
    {
        printf("不存在第%d个元素", i); return;
    }
    for(int j = i; j <= PtrL->Last; j++)
        PtrL->Data[j - 1] = PtrL->Data[j];
    PtrL->Last--;    // 保证Last依然指向最后元素
}
```

平均移动次数为 `(n - 1) / 2`，平均时间性能为 `O(n)`

• 链式存储实现

顺序存储结构中的元素逻辑相邻的同时存放地址也是相邻的；而链式存储则不要求逻辑上相邻的两个元素物理上也相邻，通过 **链** 建立起数据元素之间的逻辑关系，这种存储方式的主要优势就体现在插入和删除不需要移动数据元素，而只是修改 **链**




```
typedef struct LNode* List;
struct LNode{
    ElementType Data;   // 节点数据
    List Next;  // 下一个节点的位置
};
struct LNode L;
List PtrL;
```

• 主要操作实现

- 1. 求表长

```
int Length(List PtrL)
{
    List p = PtrL;  // 指向第一个节点
    int j = 0;
    while(p)
    {
        p = p->Next;
        j++;
    }
    return j;
}
```

时间性能为 `O(n)`

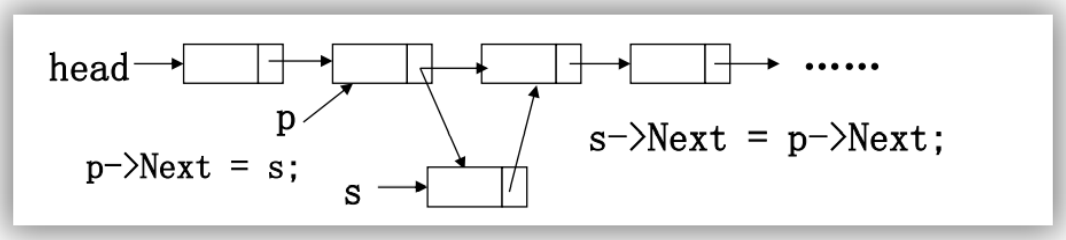
2. 查找

```
// 按序号查找，找第K个元素
List FindKth(int K, List PtrL)
{
    List p = PtrL;
    int i = 1;
    while(p != NULL && i < K)
    {
        p = p->Next;
        i++;
    }
    if(i == K)  return p;    // 找到第K个元素，返回指针
    else return NULL;    // 找不到，返回NULL

}
// 按值查找
List Find(ElementType X, List PtrL)
{
    List p = PtrL;
    while(p != NULL && p->Data != X)
        p = p->Next;
    return p;
}
```

两种查找平均时间性能均为 `O(n)`

3. 插入（在第 `i` 个节点后插入一个值为X的新节点，返回链表 `head` ）

1. 构造一个新节点，用 `s` 指向
2. 找到链表的第 `i- 1` 个节点，用 `p` 指向
3. 修改指针，先将 `p->next` 赋给 `s->next` ，再将 `s` 赋给 `p->next` ，注意这个过程不能颠倒



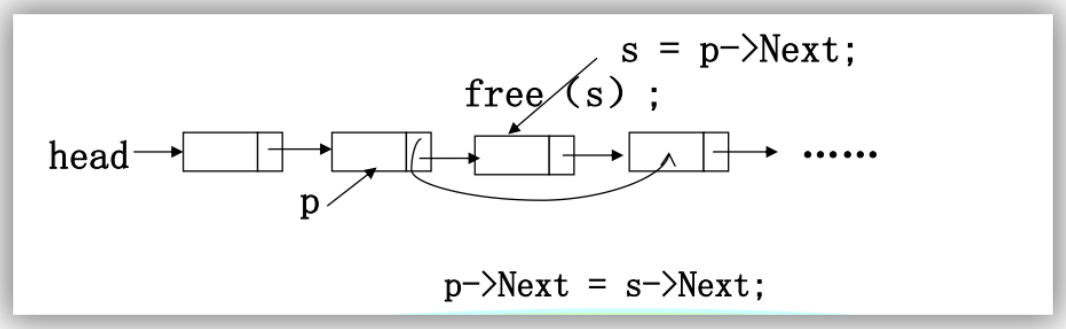
```
List Insert(ElementType X, int i, List PtrL)
{
    List p, s = (List)malloc(sizeof(struct LNode)); // 申请，填装节点;
    // 新节点插在表头，新节点前面没有第i-1个节点，因此不需要p->Next = s
    if(i == 1)
    {
        s->Data = X;
        s->Next = PtrL;
        return s;
    }
    p = FindKth(i - 1, PtrL);    //查找第i-1个节点
    if(p == NULL)    // 第i-1个节点不存在，则不能插入
    {
        printf("参数i有误"); return NULL;
    }
    else
    {
        s->Data = X;
        s->Next = p->Next;
        p->Next = s;
        return PtrL;
    }
}
```

除了查找之外的操作都是常数时间完成，因此插入的平均时间性能为 `O(n)`

- 4. 删除（删除链表第 `i` 个位置上的节点，返回链表 `head` ）

1. 找到第 `i - 1` 个节点，用 `p` 指向
2. `s` 指向要被删除的节点，即 `p->next`
3. 修改 `p->next` 为 `s->next` （或者是 `p->next->next` ）

4. 释放 s 所指节点的空间




```
List Delete(int i, List PtrL)
{
    List p, s;
    if(i == 1)
    {
        s = PtrL;
        if(PtrL != NULL)    PtrL = PtrL->next;
        else return NULL;
        free(s);
        return PtrL;
    }
    p = FindKth(i - 1, PtrL);    // 寻找第i-1个节点
    // 第i个节点或第i-1个节点不存在都无法删除
    if(p == NULL || p->Next == NULL)
    {
        printf("第%d个节点不存在", i); return NULL;
    }
    else
    {
        s = p->next;
        p->next = s->next;
        free(s);
        return PtrL;
    }
}
```

除了查找之外的操作都是常数时间完成，因此删除的平均时间性能为 O(n)

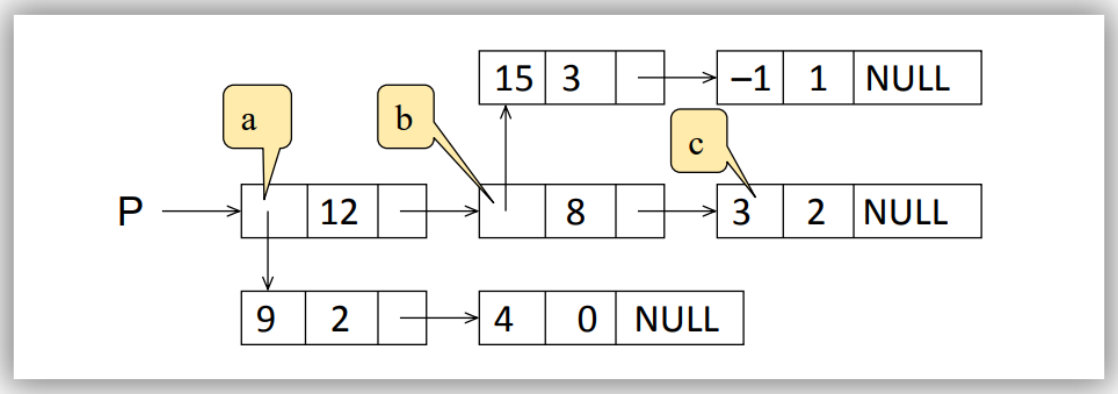
广义表 Generalized List：表中元素可以是单元素或广义表

我们知道了一元多项式的表示，那么二元多项式又该如何表示?

$$P(x,y)=9x^{12}y^2+4x^{12}+15x^8y^3-x^8y+3x^2 \tag{3}$$

一种方法是将其看成是关于x的一元多项式，而这个一元多项式的系数中又包含了关于y的一元多项式，$P(x,y)=\left(9y^2+4\right)x^{12}+\left(15y^3-y\right)x^8+3x^2$

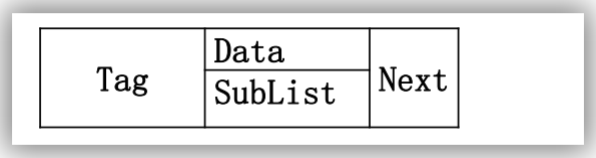
我们用一种复杂的链表来表示这种二元多项式，这就是一种广义表



- 广义表是**线性表的推广**
- 对于线性表而言，n 个元素都是基本的单元素
- 广义表的元素可以是单元素，也可以是另一个广义表（包括线性表，及其无限套娃）

例如：(1, 2, 3, (4, 5, (6, 0, (99, 94))), (88, 99, 32))是一个长度为5的广义表，它的后两个元素是它的子表

```
typedef struct GNode* GList;
struct GNode
{
    int Tag;     // 标志域，0表示节点是单元素，1表示节点是广义表
    union{       // 子表指针域Sublist与单元素数据域Data复用，即共用存储空间
        ElementType Data;
        GList SubList;
    } URegion;
    GList Next; // 后继节点
}
```



多重链表：链表中节点可能同时隶属于多个链

- 多重链表中节点的**指针域会有多个**，实际上广义表中的例子就是一个多重链表，包含了 `Next` 和 `SubList` 两个指针域
- **含有多个指针域的链表不一定是多重链表**，比如双向链表，指针域中的指针必须指向不同的链才行
- 多重链表应用广泛，例如树，图这种复杂的数据结构都可以使用多重链表来存储

例：矩阵存储

用二维数组存储矩阵有两个缺点

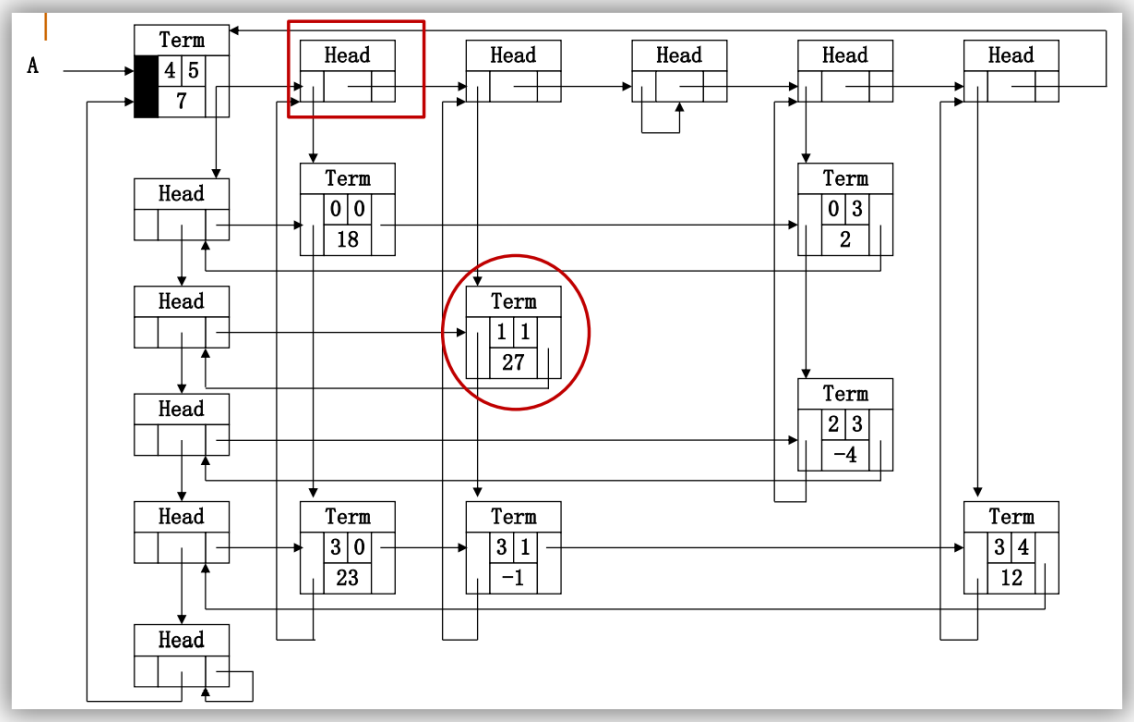
- 事先需要确认数组的大小
- 如果是稀疏矩阵，则会造成大量的存储空间浪费

我们采用一种典型的多重链表---十字链表来存储稀疏矩阵

- 只存储矩阵的非0元素项
- 每个非零元素项节点的数据域包括：行坐标Row，列坐标Col，数值Value

- 每个节点通过两个指针域：行（向右）指针 Right，列（向下）指针 Down，把同行、同列串起来

下图是矩阵 $\begin{bmatrix} 18 & 0 & 0 & 2 & 0 \\ 0 & 27 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 0 \\ 23 & -1 & 0 & 0 & 12 \end{bmatrix}$ 的十字链表表示



- 一共存在两种节点，一种是头节点标识节点，标识值为 `Head`，一种是非零元素节点，标识值为 `Term`
- 注意左上角为矩阵的入口节点，它的Row是矩阵行数，Col是矩阵列数，Value是矩阵非零元素个数，它通过两个指针域连接了行的头节点和列的头节点
- 每一行和每一列对于的链表都是循环链表

节点总体结构

Tag		
Down	*URegion*	Right

矩阵非零元素节点

Term		
Down	Row Col	Right
	Value	

头节点

Head		
Down	Next	Right

Part 2 堆栈 Stack

表达式求值之后缀表达式

我们平时常用的表达式是中缀表达式，运算符号位于两个运算数之间，如 `a + b * c - d / e`

后缀表达式是运算符号位于两个运算数之后，如 `a b c * + d e / -`

我们对后缀表达式求值的策略是从左往右扫描，根据下面规则逐个处理运算数和运算符号

- 遇到运算数，记住未参与运算的数
- 遇到运算符号，取出最近未参与运算的两个数

例如 `a b c * + d e / -`，先记住 `a,b,c`，扫描到 `*`，执行 `b * c`，扫描到 `+`，执行 `a + b * c`，扫描到 `d,e` 则先记住，扫描到 `/`，执行 `d / e`，扫描到 `-`，执行 `a + b * c - d / e`

由上可知，我们需要一种数据结构可以**顺序存储**运算数，当我们需要运算时**倒序输出**这些运算数，堆栈就是这样一种数据结构

堆栈的抽象数据类型描述

堆栈：具有一定操作约束的线性表，只在栈顶Top做插入删除

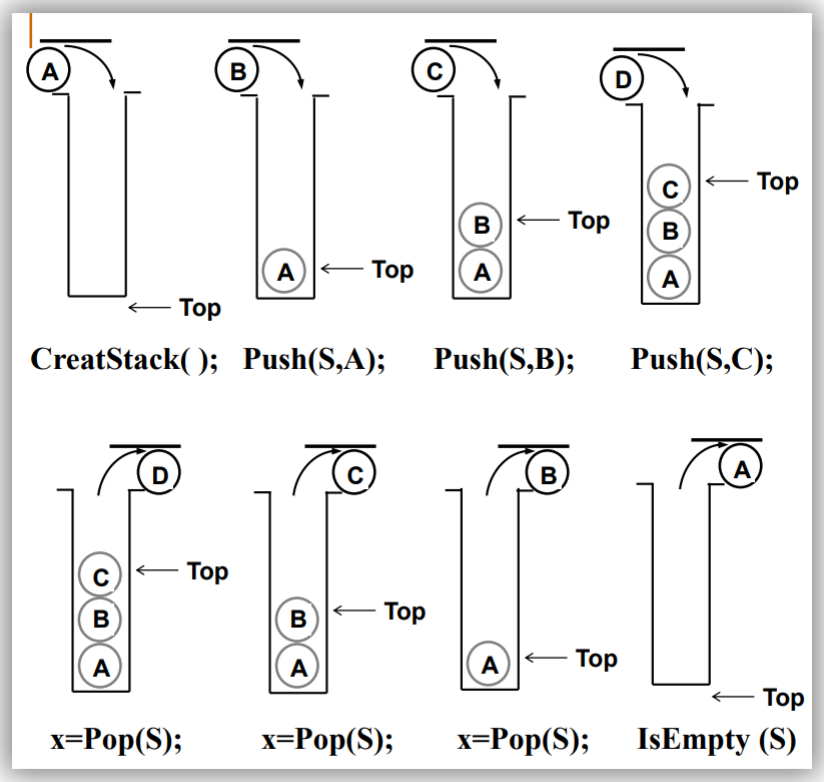
- 插入数据：入栈 Push
- 删除数据：出栈 Pop
- 后入先出：Last In First Out LIFO

类型名称：堆栈　Stack

数据对象集：一个有 0 个或多个元素的有穷线性表

操作集：长度MaxSize的堆栈S ∈ Stack，堆栈元素item ∈ ElementType

```
Stack CreateStack(int MaxSize);          //生成空堆栈，最大长度为
MaxSize
int IsFull(Stack S, int MaxSize);        //判断堆栈S是否已满
void Push(Stack S, ElementType item);    //将元素item压入堆栈
int IsEmpty(Stack S);                    //判断堆栈S是否为空
ElementType Pop(Stack S);                //删除并返回栈顶元素
```



考虑三个字符A，B，C顺序压入堆栈，那么C，A，B这样的出栈顺序是不可能出现的！

实现

堆栈的顺序存储实现

通常由一个一维数组和一个记录栈顶元素位置的变量组成

```
const int MaxSize = (存储数据元素的最大个数);
typedef struct SNode* Stack;
struct SNode
{
    ElementType Data[MaxSize];
    int Top;
};
```

```
 void Push(Stack PtrS, ElementType item)
 {
     if(PtrS->Top == MaxSize - 1)
     {
         printf("堆栈已满"); return;
     }
     else
     {
         PtrL->Data[++(PtrL->Top)] = item;
         return;
     }
 }
```

```
ElementType Pop(Stack PtrS)
{
    if(PtrS->Top < 0)
    {
        printf("堆栈为空"); return ERROR;    // ElementType的特殊值，标志错误
    }
    else
        return (PtrS->Data[(PtrL->top)--]);
}
```

• 尝试一个数组实现两个堆栈，并充分利用空间

将这两个堆栈分别从数组的两边开始向中间生长，当两个栈的栈顶指针相遇时，表示两个栈都满了

```
const int MaxSize = (存储数据元素的最大个数);
struct DStack
{
    ElementType Data[MaxSize];
    int Top1;
    int Top2;
}S;
S.Top1 = -1;
S.Top2 = MaxSize;
```

```
void Push(struct DStack* PtrS, ElementType item, int Tag)
{
    if(PtrS->Top2 - PtrS->Top1 <= 1)
    {
        printf("堆栈满");  return;
    }
    if(Tag == 1)     //对第一个堆栈进行操作
        PtrS->Data[++(PtrS->Top1)] = item;
    else             //对第二个堆栈进行操作
        PtrS->Data[--(PtrS->Top2)] = item;
}
```

```
ElementType Pop( struct DStack *PtrS, int Tag )
{
    if(Tag == 1)
        if(PtrS->Top1 < 0)  //堆栈1空
        {
            pirntf("堆栈1为空");    return NULL;
        }
        else return PtrS->Data[(PtrS->Top1)--];
    else
        if(MaxSize - 1 < PtrS->Top2)    //堆栈2空
        {
            printf("堆栈2为空");    return NULL;
        }
        else return PtrS->Data[(PtrS->Top2)++];
}
```

堆栈的链式存储实现

实际上一个单链表，称为链栈，插入和删除都在链栈的栈顶进行

注意：栈顶只能定义在单链表的头部，不能定义到单链表的尾部，因为栈顶若在单链表的尾部，删除操作是我们将栈顶删除，随后无法找到新的栈顶了（单链表只有前一个节点指向后一个节点，反之，通过后一个节点找不到前一个节点）！

上面说明了单链表的一个共性：**链表头可以插入和删除，链表尾只能删除**，后面队列的链式实现也会体现这一点

```
typedef struct SNode* Stack;
struct SNode
{
    ElementType Data;
    Stack Next;
};

Stack CreateStack()
{
    Stack S;
    S = (Stack)malloc(sizeof(struct SNode));    //S不存储具体的节点
    S->Next = NULL;
    return S; //构建了一个头节点并返回头节点指针
}

int IsEmpty(Stack S)
{
    //堆栈的Top元素在S->Next，如果是空，就说明栈为空
    return (S->Next == NULL);
}

void Push(ElementType item, Stack S)
{
    struct SNode *TmpCell;
    // 通过不断申请空间完成push操作，所以栈的容量没有具体限制
    TmpCell = (Stack)malloc(sizeof(struct SNode));
    TmpCell->Element = item;
    TmpCell->Next = S->Next;
    S->Next = TmpCell;
}
```

```
31
32  ElementType Pop(Stack S)
33  {
34      struct SNode* FirstCell;
35      ElementType TopElem;
36      if(IsEmpty(S))
37      {
38          printf("堆栈为空"); return NULL;
39      }
40      else
41      {
42          FirstCell = S->Next;
43          S->Next = FirstCell->Next;
44          TopElem = FirstCell->Element;
45          free(FirstCell);
46          return TopElem;
47      }
48  }
```

中缀表达式转换为后缀表达式

从头到尾读取中缀表达式的每个对象，对不同对象按不同情况处理

- 运算数：直接输出
- 左括号：压入堆栈
- 右括号：将栈顶的运算符弹出并输出，直至弹出左括号（不输出）
- 运算符
    - 优先级大于栈顶运算符时，将其压栈
    - 优先级小于等于栈顶运算符时，将栈顶元素弹出并输出；再比较新的栈顶元素重复上述操作直至该运算符大于栈顶运算符为止，将其压栈

表达式读取完毕后，将堆栈剩余运算符依次输出

步骤	待处理表达式	堆栈状态 （底←→顶）	输出状态
1	2*（9+6/3-5）+4		
2	*（9+6/3-5）+4		2
3	（9+6/3-5）+4	*	2
4	9+6/3-5）+4	*（	2
5	+6/3-5）+4	*（	2 9
6	6/3-5）+4	*（+	2 9
7	/3-5）+4	*（+	2 9 6
8	3-5）+4	*（+/	2 9 6
9	-5）+4	*（+/	2 9 6 3
10	5）+4	*（-	2 9 6 3 / +
11	）+4	*（-	2 9 6 3 / + 5
12	+4	*	2 9 6 3 / + 5 -
13	4	+	2 9 6 3 / + 5 - *
14		+	2 9 6 3 / + 5 - * 4
15			2 9 6 3 / + 5 - * 4 +

Part 3 队列 Queue

队列的抽象数据类型描述

- 队列 Queue：具有一定操作约束的线性表（只能在一端插入，而在另一端删除）
- 数据插入：入队
- 数据删除：出队
- 先来先服务，先进先出 FIFO

类型名称：队列(Queue)

数据对象集：一个有0个或多个元素的有穷线性表。

操作集：长度为MaxSize的队列Q ∈ Queue，队列元素item ∈ ElementType

```
Queue CreatQueue(int MaxSize);       //生成长度为MaxSize的空队列;
int IsFullQ(Queue Q, int MaxSize);   //判断队列Q是否已满;
void AddQ(Queue Q, ElementType item);//将数据元素item插入队列Q中;
int IsEmptyQ(Queue Q);               //判断队列Q是否为空;
ElementType DeleteQ(Queue Q);        //将队头数据元素从队列中删除并返回
```

队列的顺序存储实现

```
const int MaxSize = (存储数据元素的最大个数);
struct QNode
{
    ElementType Data[MaxSize];
    int rear;   //记录尾元素位置
    int front;  //记录头元素位置
};
typedef struct QNode* Queue;
```

初始状态，`rear`和`front`均设为`-1`，此时`rear == front`，是判定队列为空的一种方法

当有元素入队时，`rear++`，确保它指向新加入的元素；当有元素出队的时候，`front++`，确保它始终指向队列第一个元素的前一个位置

在不断的入队出队操作之后，可能`rear`已经指向数组的最后一个位置，而`front`前面还要大量空闲的数组空间，我们要使得新入队的元素在数组后面没有位置的情况下可以放到数组的前面空闲部分，以免浪费空间，我们称之为**循环队列**

对于大小为`n`的队列，队列装载元素的情况有`n + 1`种，`0(空),1,2,3,...,n-1,n`；而我们根据`front`和`rear`的差距来判断队列装载了多少元素，`0,1,2,...,n-1`，最多表示`n`种情况，这就出现了矛盾，在循环队列种，空队列和满队列都有`rear == front`，解决方法有以下两种

- 使用额外标记：`Size`来记录当前队列元素个数或`tag`记录前一次操作是出队还是入队，如果是出队导致`rear == front`那就是队空，反之，就是队满

- 仅使用 `n - 1` 个数组空间，队列种有 `n - 1` 个元素时，就判定队满，使得 `rear - front` 可以表示全部队列装载元素的情况，下面使用这种解决方法给出具体实现

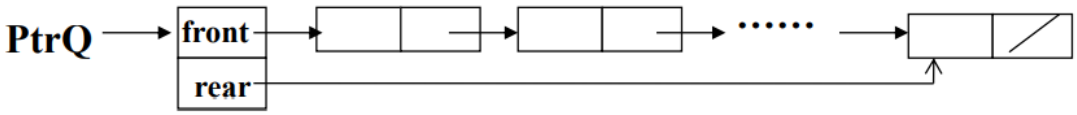
```
void AddQ(Queue PtrQ, ElementType item)
{
    if((PtrQ->rear + 1) % MaxSize == PtrQ->front)
    {
        printf("队列已满"); return;
    }
    PtrQ->rear = (PtrQ->rear + 1) % MaxSize;
    PtrQ->Data[PtrQ->rear] = item;
}

ElementType DeleteQ(Queue PtrQ)
{
    if(PtrQ->front == PtrQ->rear)
    {
        printf("队列为空"); return ERROR;
    }
    else
    {
        PtrQ->front = (PtrQ->front + 1) % MaxSize;
        return PtrQ->Data[PtrQ->front];
    }
}
```

队列的链式存储实现

链表头可以执行删除和插入，而链表尾只能执行删除，所以 `front` （删除）应指向链表尾， `rear` （插入）应指向链表头

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

```
6
7   struct QNode
8   {
9       struct Node* rear;
10      struct Node* front;
11  };
12
13  typedef struct QNode* Queue;
14  Queue PtrQ;
```



`PtrQ->front == PtrQ->rear == NULL` 则该队列为空

```
1   ElementType DeleteQ(Queue PtrQ)
2   {
3       struct Node* FrontCell;
4       ElementType FrontElem;
5
6       if(PtrQ->front == NULL)
7       {
8           printf("队列为空"); return ERROR;
9       }
10
11     FrontCell = PtrQ->front;
12      if(PtrQ->front == PtrQ->rear)
13          PtrQ->front = PtrQ->rear = NULL;
14      else
15          PtrQ->front = PtrQ->front->Next;
16      FrontElem = FrontCell->Data;
17      free(FrontCell);
18      return FrontElem;
19  }
20
21  void AddQ(Queue PtrQ, ElementType item)
22  {
23      struct Node* TmpCell;
24      // 通过不断申请空间完成add操作，所以该队列的容量没有具体限制
25      TmpCell = (Node*)malloc(sizeof(struct Node));
26      TmpCell->Data = item;
```

```
27        TmpCell->Next = PtrQ->Next;
28        PtrQ->Next = TmpCell;
29    }
```

编程练习

1. 两个有序链表序列的合并

本题要求实现一个函数，将两个链表表示的递增整数序列合并为一个非递减的整数序列。

• 函数接口定义：

```
1   List Merge( List L1, List L2 );
```

其中 `List` 结构定义如下：

```
1   typedef struct Node *PtrToNode;
2   struct Node {
3       ElementType Data; /* 存储结点数据 */
4       PtrToNode   Next; /* 指向下一个结点的指针 */
5   };
6   typedef PtrToNode List; /* 定义单链表类型 */
```

`L1` 和 `L2` 是给定的带头结点的单链表，其结点存储的数据是递增有序的；函数 `Merge` 要将 `L1` 和 `L2` 合并为一个非递减的整数序列。应直接使用原序列中的结点，返回归并后的带头结点的链表头指针。

- **裁判测试程序样例：**

```c
#include <stdio.h>
#include <stdlib.h>

typedef int ElementType;
typedef struct Node* PtrToNode;
struct Node {
    ElementType Data;
    PtrToNode   Next;
};
typedef PtrToNode List;

List Read(); /* 细节在此不表 */
void Print( List L ); /* 细节在此不表；空链表将输出NULL */

List Merge( List L1, List L2 );

int main()
{
    List L1, L2, L;
    L1 = Read();
    L2 = Read();
    L = Merge(L1, L2);
    Print(L);
    Print(L1);
    Print(L2);
    return 0;
}

/* 你的代码将被嵌在这里 */
```

- **输入样例：**

```
3
1 3 5
5
2 4 6 8 10
```

• 输出样例：

```
1 2 3 4 5 6 8 10
NULL
NULL
```

• 实现

注意：

- 链表的头节点不是含有数据的节点而是链表的开始入口，`头节点->Next == NULL` 说明这个节点为空
- 题中要求直接使用原序列中的节点，也就是直接 `L1,L2` 的节点赋值给 `tmp->Next` ，而不是对 `tmp->Data` 进行赋值，这样尝试过，未通过

```
List Merge(List L1, List L2)
{
    List L, tmp, tmp1, tmp2;
    L = (List)malloc(sizeof(struct Node));
    tmp = L, tmp1 = L1->Next, tmp2 = L2->Next;
    while(tmp1 && tmp2)
    {
        if(tmp1->Data <= tmp2->Data)
        {
            tmp->Next = tmp1;
            tmp = tmp1;
            tmp1 = tmp1->Next;
        }
        else
        {
            tmp->Next = tmp2;
            tmp = tmp2;
            tmp2 = tmp2->Next;
        }
    }
    while(tmp1)
```

```
        {
            tmp->Next = tmp1;
            tmp = tmp1;
            tmp1 = tmp1->Next;
        }
        while(tmp2)
        {
            tmp->Next = tmp2;
            tmp = tmp2;
            tmp2 = tmp2->Next;
        }
        L1->Next = NULL, L2->Next = NULL;
        return L;
    }
```

2. 一元多项式的乘法与加法运算

设计函数分别求两个一元多项式的乘积与和。

- **输入格式:**

输入分2行，每行分别先给出多项式非零项的个数，再以指数递降方式输入一个多项式非零项系数和指数（绝对值均为不超过1000的整数）。数字间以空格分隔。

- **输出格式:**

输出分2行，分别以指数递降方式输出乘积多项式以及和多项式非零项的系数和指数。数字间以空格分隔，但结尾不能有多余空格。零多项式应输出 `0 0` 。

- **输入样例:**

```
4 3 4 -5 2  6 1  -2 0
3 5 20  -7 4  3 1
```

- **输出样例:**

```
15 24 -25 22 30 21 -10 20 -21 8 35 6 -33 5 14 4 -15 3 18 2 -6 1
5 20 -4 4 -5 2 9 1 -2 0
```

- **实现**

```
#include <iostream>
#include <cstdio>
using namespace std;

typedef struct PolyNode *Polynomial;
struct PolyNode
{
    int coef;  //系数
    int expon; //指数
    Polynomial link;
};

void Attach(int c, int e, Polynomial *pRear)
{
    //建立一个新节点，并对其进行赋值
    Polynomial P;
    P = (Polynomial)malloc(sizeof(struct PolyNode));
    P->coef = c;
    P->expon = e;
    P->link = NULL;
    //将之前的Rear指向新建的节点
    (*pRear)->link = P;
    //将Rear设置为新建的节点，为下一次插入准备
    *pRear = P;
}

Polynomial ReadPoly()
{
    int c, e, cnt;
    Polynomial P, Rear, t;

    cin >> cnt;
```

```
33      P = (Polynomial)malloc(sizeof(struct PolyNode)); //链表头
    空节点
34      P->link = NULL;
35      Rear = P;
36      while (cnt--)
37      {
38          cin >> c >> e;
39          Attach(c, e, &Rear); //不断将新节点插到尾部
40      }
41      t = P;
42      P = P->link;
43      free(t);  //
44      return P; //返回读入链表的头节点
45  }
46
47  Polynomial Add(Polynomial P1, Polynomial P2)
48  {
49      Polynomial front, rear, temp;
50      int sum;
51      rear = (Polynomial)malloc(sizeof(struct PolyNode));
52      front = rear; //记录头节点
53      while (P1 && P2)
54      {
55          if (P1->expon == P2->expon)
56          {
57              sum = P1->coef + P2->coef;
58              if (sum)
59                  Attach(sum, P1->expon, &rear);
60              P1 = P1->link;
61              P2 = P2->link;
62          }
63          else if (P2->expon <= P1->expon)
64          {
65              Attach(P1->coef, P1->expon, &rear);
66              P1 = P1->link;
67          }
68          else
69          {
70              Attach(P2->coef, P2->expon, &rear);
71              P2 = P2->link;
72          }
73      }
74      while (P1)
```

```
75      {
76          Attach(P1->coef, P1->expon, &rear);
77          P1 = P1->link;
78      }
79      while (P2)
80      {
81          Attach(P2->coef, P2->expon, &rear);
82          P2 = P2->link;
83      }
84      rear->link = NULL;
85      temp = front;
86      front = front->link; //令front指向第一个非零项
87      free(temp);
88      return front;
89  }
90
91  Polynomial Mult(Polynomial P1, Polynomial P2)
92  {
93      if (!(P1 && P2)) // P1或P2为空都返回NULL
94          return NULL;
95      Polynomial P, rear, t1, t2, t;
96      int c, e;
97      t1 = P1, t2 = P2;
98      P = (Polynomial)malloc(sizeof(struct PolyNode));
99      P->link = NULL;
100     rear = P;
101     while (t2) // P1的第一项乘以P2，得到P
102     {
103         Attach(t1->coef * t2->coef, t1->expon + t2->expon,
    &rear);
104         t2 = t2->link;
105     }
106     t1 = t1->link;
107     while (t1)
108     {
109         t2 = P2, rear = P;
110         while (t2)
111         {
112             e = t1->expon + t2->expon;
113             c = t1->coef * t2->coef;
114             while (rear->link && e < rear->link->expon)
115                 rear = rear->link;
116             if (rear->link && rear->link->expon == e)
```

```
                if (rear->link->coef + c)
                    rear->link->coef += c;
                else
                {
                    t = rear->link;
                    rear->link = t->link;
                    free(t);
                }
            else
            {
                t = (Polynomial)malloc(sizeof(struct
    PolyNode));
                t->coef = c;
                t->expon = e;
                t->link = rear->link;
                rear->link = t;
                rear = rear->link;
            }
            t2 = t2->link;
        }
        t1 = t1->link;
    }
    t2 = P;
    P = P->link;
    free(t2);
    return P;
}

void PrintPoly(Polynomial P)
{
    int flag = 0;
    if (!P)
    {
        printf("0 0\n");
        return;
    }
    while (P)
    {
        if (!flag)
            flag = 1; //确保第一个输出的前面没有空格
        else
            printf(" ");
        cout << P->coef << ' ' << P->expon;
```

```
            P = P->link;
        }
        cout << endl;
    }

    int main()
    {
        Polynomial P1, P2, PP, PS;
        P1 = ReadPoly();
        P2 = ReadPoly();
        PP = Mult(P1, P2);
        PrintPoly(PP);
        PS = Add(P1, P2);
        PrintPoly(PS);
        return 0;
    }
```

3. Reversing Linked List 单链表翻转

Given a constant K and a singly linked list L, you are supposed to reverse the links of every K elements on L. For example, given L being 1→2→3→4→5→6, if K=3, then you must output 3→2→1→6→5→4; if K=4, you must output 4→3→2→1→5→6.

• Input Specification:

Each input file contains one test case. For each case, the first line contains the address of the first node, a positive N (≤105) which is the total number of nodes, and a positive K ($\le N$) which is the length of the sublist to be reversed. The address of a node is a 5-digit nonnegative integer, and NULL is represented by -1.

输入的第一行包括三个数，分别是第一个节点的地址、这个链表包含的节点数N、需要反转的节点的个数K

节点地址是五位非负数，如果这个节点是NULL，它的地址用-1表示

Then N lines follow, each describes a node in the format:

```
Address Data Next
```

where `Address` is the position of the node, `Data` is an integer, and `Next` is the position of the next node.

接下来的N行每行三个数，分别是每个节点的地址，节点的数据，节点的下一个节点的地址

• Output Specification:

For each case, output the resulting ordered linked list. Each node occupies a line, and is printed in the same format as in the input.

输出翻转后的链表，每个节点占一行，和输入是同样的模式

• Sample Input:

```
00100 6 4
00000 4 99999
00100 1 12309
68237 6 -1
33218 3 00000
99999 5 68237
12309 2 33218
```

• Sample Output:

```
00000 4 33218
33218 3 12309
12309 2 00100
00100 1 99999
99999 5 68237
68237 6 -1
```

• 实现

题意理解错了，是每K个节点翻转一次，不是前K个节点翻转一次，有两个点没过

测试点	提示	结果	分数	耗时	内存
0	sample 有尾巴不反转, 地址取上下界	答案正确	12	4 ms	328 KB
1	正好全反转	答案错误	0	5 ms	300 KB
2	K=N全反转	答案正确	2	4 ms	316 KB
3	K=1不用反转	答案正确	2	4 ms	308 KB
4	N=1 最小case	答案正确	2	6 ms	424 KB
5	最大N,最后剩K-1不反转	答案错误	0	109 ms	3004 KB
6	有多余结点不在链表上	答案正确	1	4 ms	304 KB

```cpp
#include <iostream>
using namespace std;
const int maxn = 100010;
struct node
{
    int data;
    int next;
} List[maxn];
int head, N, K; //节点数N，翻转前K个节点

void Input()
{
    cin >> head >> N >> K;
    int tmp_index, tmp_data, tmp_next;
    int n = N;
    while (n--)
    {
        cin >> tmp_index >> tmp_data >> tmp_next;
        List[tmp_index].data = tmp_data;
        List[tmp_index].next = tmp_next;
    }
}

void ReverseList()
{
    if (K == 1) // K=1时不需要进行任何操作即可
        return;
    //查找输入的链表有多少节点
```

```
29      int i = head, cnt = 1;
30      while (List[i].next != -1)
31      {
32          cnt++;
33          i = List[i].next;
34      } //最后cnt的值就是这个链表的节点个数
35
36      //先完成第K个节点和头节点的交换
37      if (1 < K)
38      {
39          int beforeK = head;
40          int step = K - 2; //从head开始向前走K-1步到第K个节点，走K-
    2步到第K-1个节点
41          //首先找到第K个节点的前一个节点的索引
42          while (step--)
43              beforeK = List[beforeK].next;
44
45          int K_index = List[beforeK].next; //当前K的索引
46          int head_next = List[head].next;  //head的下一个节点的索
    引
47          int K_next = List[K_index].next;  //第K个节点的下一个节
    点的索引
48
49          if (beforeK == head)
50          {
51              List[K_index].next = head; //1
52              List[head].next = K_next;  //2
53          }
54          else
55          {
56              List[K_index].next = head_next; //K的next设为head
    的next
57              List[beforeK].next = head;      //K的前一个节点的
    next设为head
58              List[head].next = K_next;       //head的next设为原
    K的next
59          }
60          head = K_index; //头节点的索引已经变为第K个节点的索引
61      }
62
63      for (int i = 2, j = K - 1; i < j; i++, j--)
64      {
```

```
65          int fstep = i - 2, rstep = j - 2, before_index1 =
    head, before_index2 = head;
66
67          //第一个互换节点的前一个节点，从head开始向后走i-2步到达
68          while (fstep--)
69              before_index1 = List[before_index1].next;
70          //第二个互换节点的前一个节点，从head开始向后走j-2步到达
71          while (rstep--)
72              before_index2 = List[before_index2].next;
73
74          //下面处理两个节点与前后节点的关系
75
76          int index1 = List[before_index1].next;
77          int index2 = List[before_index2].next;
78          if (List[index1].next != index2) //要交换的节点不相邻，需
    要给两个点连各自左右的两条链，共四条链
79          {
80              int next1 = List[index1].next;
81              int next2 = List[index2].next;
82              //把index2插到iindex1原先的位置
83              List[before_index1].next = index2; //1
84              List[index2].next = next1;         //2
85              //把index1插到index2原先的位置
86              List[before_index2].next = index1; //3
87              List[index1].next = next2;         //4
88          }
89          else //交换的节点是相邻节点，需要连接第一个节点前的一条链，两
    个节点中间的一条链，第二个节点后的一条链
90          {
91              List[before_index1].next = index2; //1
92              int next2 = List[index2].next;
93              List[index2].next = index1; //2
94              List[index1].next = next2;  //3
95          }
96      }
97  }
98  void PrintList()
99  {
100     int cur = head, i = N;
101     while (i--)
102     {
103
104         if (List[cur].next == -1)
```

```
105             {
106                 printf("%05d %d %d\n", cur, List[cur].data,
    List[cur].next);
107                 break;
108             }
109             printf("%05d %d %05d\n", cur, List[cur].data,
    List[cur].next);
110             cur = List[cur].next;
111         }
112     }
113
114     int main()
115     {
116         Input();
117         ReverseList();
118         PrintList();
119         return 0;
120     }
```

4. Pop Sequence

Given a stack which can keep M numbers at most. Push N numbers in the order of 1, 2, 3, ..., N and pop randomly. You are supposed to tell if a given sequence of numbers is a possible pop sequence of the stack. For example, if M is 5 and N is 7, we can obtain 1, 2, 3, 4, 5, 6, 7 from the stack, but not 3, 2, 1, 7, 5, 6, 4.

• Input Specification:

Each input file contains one test case. For each case, the first line contains 3 numbers (all no more than 1000): M (the maximum capacity of the stack), N (the length of push sequence), and K (the number of pop sequences to be checked). Then K lines follow, each contains a pop sequence of N numbers. All the numbers in a line are separated by a space.

一行三个数，M-栈的最大容量，N-入栈序列为1,2,3,...,N，K-需要判断的出栈序列个数

接下来K行，每行一个出栈序列进行判断

• Output Specification:

For each pop sequence, print in one line "YES" if it is indeed a possible pop sequence of the stack, or "NO" if not.

• Sample Input:

```
5 7 5
1 2 3 4 5 6 7
3 2 1 7 5 6 4
7 6 5 4 3 2 1
5 6 4 3 7 2 1
1 7 6 5 4 3 2
```

• Sample Output:

```
YES
NO
NO
YES
NO
```

• 实现

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

int main()
{
    int m, n, k;
    cin >> m >> n >> k;
    while (k--)
    {
        vector<int> pool(n, 0);
        for (int i = 0; i < n; i++)
```

```
14              cin >> pool[i]; //输入一行输出序列,存到数字池pool中
15
16          stack<int> s;
17          int pool_now = 0; //当前指向序列的索引
18          int num = 1;
19          bool flag = true;
20
21          while (pool_now < n)
22          {
23              if (s.empty())      //当前栈为空
24                  s.push(num++); //按顺序放一个数入栈
25              else                //栈不空
26              {
27                  //如果栈顶元素和当前数字池指向的数字相同
28                  if (s.top() == pool[pool_now])
29                  {
30                      s.pop();     //直接出栈即可
31                      pool_now++; //指向数字池下一个元素
32                  }
33                  //栈顶元素较小
34                  else if (s.top() < pool[pool_now])
35                  {
36                      //检查是否满栈,若满了则不可能出栈为pool_now指向的
    数，该序列不可能是出栈序列
37                      if (s.size() == m)
38                      {
39                          flag = false;
40                          break;
41                      }
42                      else //若没满，继续入栈
43                          s.push(num++);
44                  }
45                  //栈顶元素大于当前pool_now指向的数字，该序列不可能是出
    栈序列
46                  else
47                  {
48                      flag = false;
49                      break;
50                  }
51              }
52          }
53          if (flag)
54              cout << "YES" << endl;
```

```
55            else
56                cout << "NO" << endl;
57        }
58        return 0;
59    }
```