

# 课前签到

数据结构与算法2023秋季



微信扫一扫，使用小程序

1. 微信扫码+**实名**
2. 点击今日签到

签到时间：

9:45~10:15

签到地方：

珠海校区-教学  
大楼-C407

人脸识别；智能定位

# 上课纪律



手机静音

上课期间手机静音：

1. 关闭手机
2. 飞行模式



# 数据结构与算法 表、栈和队列

余建兴

中山大学人工智能学院

## 提纲

- 1 线性表
- 2 堆栈
- 3 队列
- 4 应用实例

## 2.1 线性表

### 什么是线性表



# 多项式表示

## [例] 一元多项式及其运算

一元多项式：

$$f(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n$$

主要运算：多项式相加、相减、相乘等

## [分析] 如何表示多项式？

多项式的关键数据：

- 多项式项数  $n$
- 各项系数  $a_i$  及指数  $i$


# 多项式表示

## • 方法1：顺序存储结构直接表示

- 数组各分量对应多项式各项

- $a[i]$ ：项  $x^i$  的系数  $a_i$

- 例如：  $f(x) = 4x^5 - 3x^2 + 1$



下标 $i$	0	1	2	3	4	5	.....
$a[i]$	1	0	-3	0	0	4	.....
	1		$-3x^2$			$4x^5$	

- 两个多项式相加：两个数组对应分量相加
- 问题：如何表示多项式  $x + 3x^{2000}$ ？

# 多项式表示

- 方法2：顺序结构存储非零项

- 每个非零项  $a_i x^i$  涉及两个信息：系数  $a_i$  和指数  $i$
- 可以将一个多项式看成是一个  $(a_i, i)$  二元组的集合
- 用结构表示：数组分量是由系数  $a_i$ 、指数  $i$  组成的结构，对应一个非零项

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

下标i	0	1	2	.....
系数 $a_i$	9	15	3	-
指数i	12	8	2	-

(a)  $P_1(x)$

下标i	0	1	2	3	.....
系数 $a_i$	26	-4	-13	82	-
指数i	19	8	6	0	-

(b)  $P_2(x)$

**按指数大小有序存储！**

# 多项式表示

- 方法2：顺序结构存储非零项

- 相加过程：从头开始，比较两个多项式当前对应的指数

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)

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

# 多项式表示

- 方法3：链表结构存储非零项

- 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**

- 两个数据域及一个**指针域**，表示为：

coef	expon	link
------	-------	------

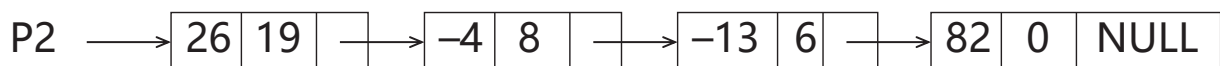
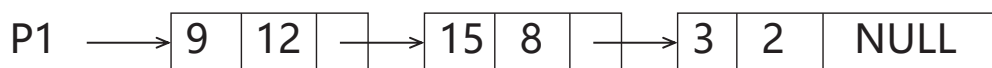
```
typedef struct PolyNode *Polynomial;
typedef struct PolyNode {
    int coef;
    int expon;
    Polynomial link;
}
```

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：



## 什么是线性表

多项式表示问题的启示：

1. 同一个问题可以有不同的表示（存储）方法
2. 有一类共性问题：**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**表头**，表结束位置称**表尾**

# 线性表的抽象数据类型描述

- **类型名称：**线性表(List)
- **数据对象集：**线性表是 $n(\geq 0)$ 个元素构成的有序序列 $(a_1, a_2, \dots, a_n)$
- **操作集：**线性表 $L \in \text{List}$ ，整数 $i$ 表示位置，元素 $X \in \text{ElementType}$ ，线性表基本操作主要有：
  - **List MakeEmpty():** 初始化一个空线性表L;
  - **ElementType FindKth( int K, List L ):** 根据位序K, 返回相应元素 ;
  - **int Find( ElementType X, List L ):** 在线性表L中查找X的第一次出现位置;
  - **void Insert( ElementType X, int i, List L ):** 在位序i前插入一个新元素X;
  - **void Delete( int i, List L ):** 删除指定位序i的元素;
  - **int Length( List L ):** 返回线性表L的长度n。

## 线性表的顺序存储实现

- 利用数组的**连续存储空间顺序存放**线性表的各元素

下标i	0	1	...	i-1	i	...	n-1	...	MAXSIZE-1
Data	$a_1$	$a_2$	...	$a_i$	$a_{i+1}$	...	$a_n$	...	-

```
typedef struct {  
    ElementType Data[MAXSIZE];  
    int Last;  
} List;  
List L, *PtrL;
```

↖ Last

- 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(List) );
    PtrL->Last = -1;
    return PtrL;
}
```

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

## 2. 查找

```
int Find( ElementType X, List *PtrL )
{
    int i = 0;
    while( i <= PtrL->Last && PtrL->Data[i] != X )
        i++;
    if ( i > PtrL->Last) return -1; /*如果没找到, 返回-1*/
    else return i; /* 找到后返回的是存储位置 */
}
```

# 顺序表的查找演示

## 顺序查找

bobo

顺序查找是一种最简单的查找方法。其基本思想是：从表的一端开始，顺序扫描线性表，依次将扫描到的结点关键字和给定值K相比较。若当前扫描到的结点关键字与K相等，则查找成功；若扫描结束后，仍未找到关键字等于K的结点，则查找失败。

*说明：*本演示中，顺序表中最多可含16个结点，各结点关键字之间用半角逗号分隔，并且各关键字为100以内的正整数。

输入顺序表中的关键字：3, 17, 2, 8, 9, 10, 21, 18, 16

输入查找的给定值K：8

开始查找



# 主要操作实现

## ● 3. 插入(第 $i$ ( $1 \leq i \leq n + 1$ )个位置上插入)

下标 $i$	0	1	...	$i-1$	$i$	...	$n-1$	...	MAXSIZE-1
Data	$a_1$	$a_2$	...	$a_i$	$a_{i+1}$	...	$a_n$	...	-

↓ Last  
先移动，再插入

下标 $i$	0	1	...	$i-1$	$i$	$i+1$	...	$n$	...	SIZE-1
Data	$a_1$	$a_2$	...	X	$a_i$	$a_{i+1}$	...	$a_n$	...	-

↻ ↻ ↻ Last

## 顺序表的插入演示

### 顺序表的插入

bobo

InsertList(L, , ) ---- 请输入要插入的位置  $i$   
----- 请输入要插入的字符  $x$

▶ 执行 ◀ 重来

```
#define ListSize 10
typedef int DataType;
typedef struct {
    DataType data[ListSize];
    int length;
```

```
} SeqList;
```

```
SeqList *L;
```

```
L->data
```

```
L->length
```

6

a1	a2	a3	a4	a5	a6				
1	2	3	4	5	6	7	8	9	10

# 顺序表的插入演示

数据变化

关键代码

1

2

3

4

5

6

7

空闲空间

```
Status ListInsert(SqList *L,int i,ElemType e)
```

```
{  
    int k;  
    if (L->length==MAXSIZE)  
        return ERROR;  
    if (i<1 || i>L->length+1)  
        return ERROR;  
    if (i<=L->length)  
    {  
        for(k=L->length-1;k>=i-1;k--)  
            L->data[k+1]=L->data[k];  
    }  
    L->data[i-1]=e;  
    L->length++;  
  
    return OK;  
}
```

i=3, e=8, L->length=7, k=6

# 顺序表的插入演示

数据变化

关键代码

1

2

3

4

5

6

7

空闲空间

```
Status ListInsert(SqList *L,int i,ElemType e)
```

```
{  
    int k;  
    if (L->length==MAXSIZE)  
        return ERROR;  
    if (i<1 || i>L->length+1)  
        return ERROR;  
    if (i<=L->length)  
    {  
        for(k=L->length-1;k>=i-1;k--)  
            L->data[k+1]=L->data[k];  
    }  
    L->data[i-1]=e;  
    L->length++;  
  
    return OK;  
}
```

i=3, e=8, L->length=7

# 顺序表的插入演示

数据变化

关键代码

1 2 8 3 4 5 6 7 空闲空间

```
Status ListInsert(SqList *L,int i,ElemType e)
{
    int k;
    if (L->length==MAXSIZE)
        return ERROR;
    if (i<1 || i>L->length+1)
        return ERROR;
    if (i<=L->length)
    {
        for(k=L->length-1;k>=i-1;k--)
            L->data[k+1]=L->data[k];
    }
    L->data[i-1]=e;
    L->length++;
    return OK;
}
```

$i=3$ ,  $e=8$ ,  $L->length=7$

## 主要操作实现

### ● 4. 删除(表第 $i(1 \leq i \leq n)$ 个位置上的元素)

下标 $i$	0	1	...	$i-1$	$i$	...	$n-1$	...	MAXSIZE-1
Data	$a_1$	$a_2$	...	$a_i$	$a_{i+1}$	...	$a_n$	...	-

Last



后面的元素依次前移

下标 $i$	0	1	...	$i-1$	...	$n-2$	$n-1$	...	MAXSIZE-1
Data	$a_1$	$a_2$	...	$a_{i+1}$	...	$a_n$	$a_n$	...	-

Last

# 顺序表的删除演示

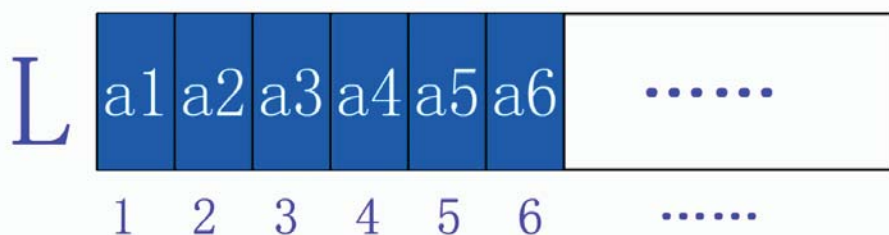
## 顺序表的删除运算

bobo

DeleteList (L, )

执行 重来

请输入你要删除的存储单元地址



## 顺序表的删除样例

数据变化

关键代码



```
Status ListDelete(SqList *L,int i,ElemType *e)
```

```
{  
    int k;  
    if (L->length==0)  
        return ERROR;  
    if (i<1 || i>L->length)  
        return ERROR;  
    *e=L->data[i-1];  
    if (i<L->length)  
    {  
        for(k=i;k<L->length;k++)  
            L->data[k-1]=L->data[k];  
    }  
    L->length--;  
    return OK;  
}
```

i=3, e=null, L->length=8

# 顺序表的删除样例

数据变化



关键代码

```
Status ListDelete(SqList *L,int i,ElemType *e)
{
    int k;
    if (L->length==0)
        return ERROR;
    if (i<1 || i>L->length)
        return ERROR;
    *e=L->data[i-1];
    if (i<L->length)
    {
        for(k=i;k<L->length;k++)
            L->data[k-1]=L->data[k];
    }
    L->length--;
    return OK;
}
```

8

i=3, e=8, L->length=7

## 数组存储实现优缺点

### 优点

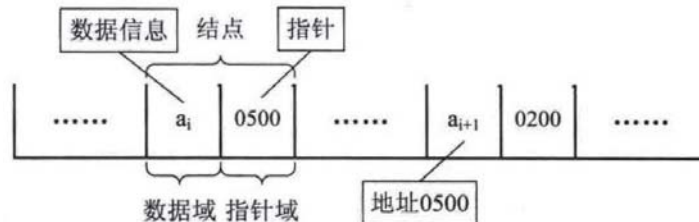
- 无须为表示表中元素之间的逻辑关系而增加额外的存储空间
- 可以快速地存取表中任一位置的元素

### 缺点

- 插入和删除操作需要移动大量元素
- 当线性表长度变化较大时，难以确定存储空间的容量
- 造成存储空间的“碎片”

# 线性表的链式存储实现

- 不要求逻辑上相邻的两个数据元素物理上也相邻; 通过“链”建立起数据元素之间的逻辑关系
- 插入、删除不需要移动数据元素, 只需要修改“链”



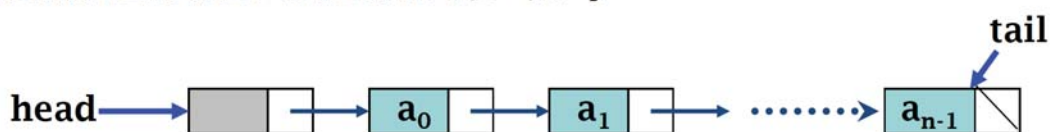
```
typedef struct Node{
    ElementType Data;
    struct Node *Next;
} List;
List L, *PtrL;
```

访问序号为  $i$  的元素?  
求线性表的长度?

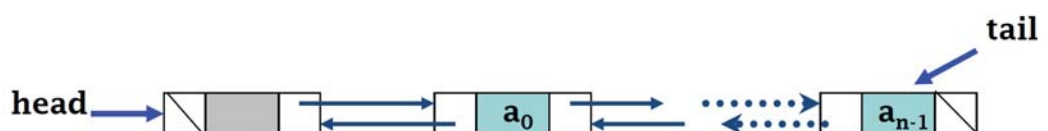
# 线性表的链式存储实现

## 分类（根据链接方式和指针多寡）

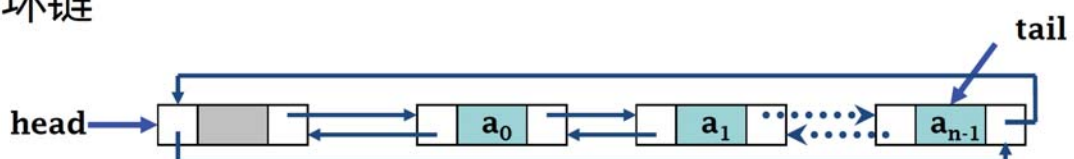
- 单链



- 双链



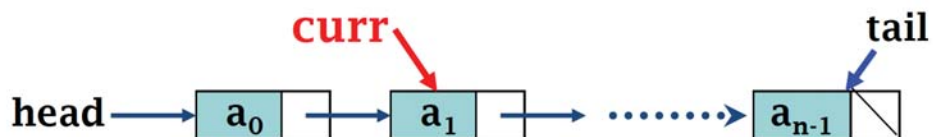
- 循环链



# 单链表

## · 简单的单链表

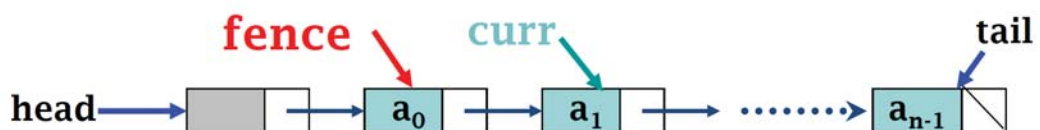
- 整个单链表：head
- 第一个结点：head
- 空表判断：head == NULL
- 当前结点  $a_1$ ：curr



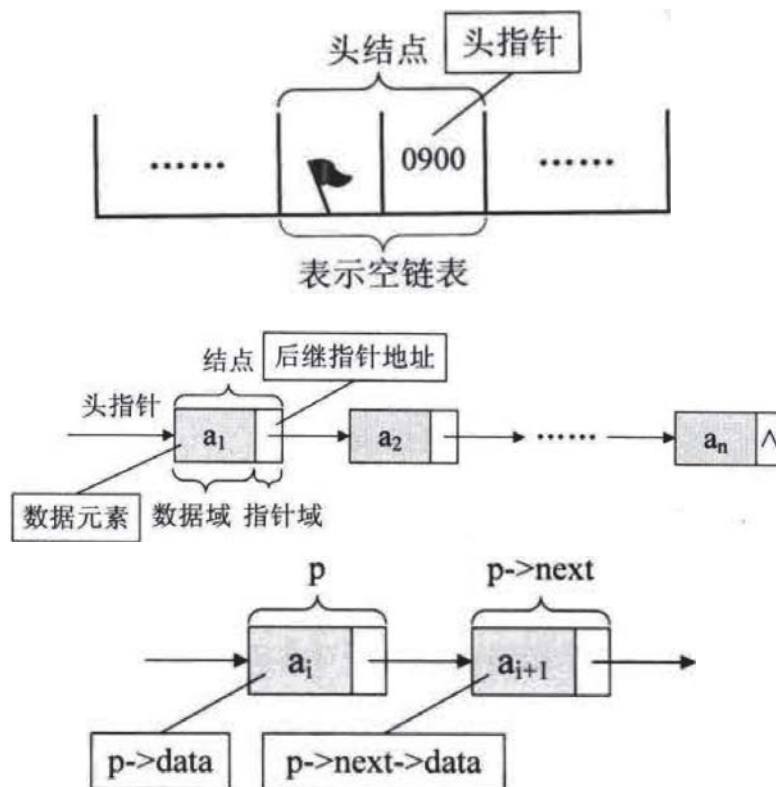
# 单链表

## · 带头结点的单链表

- 整个单链表：head
- 第一个结点：head->next, head ≠ NULL
- 空表判断：head->next == NULL
- 当前结点  $a_1$ ：fence->next (curr 隐含)



# 线性表的链式存储实现



## 主要操作实现

### ● 1. 求表长

```
int Length ( List PtrL )
{   List p = PtrL;   /* p指向表的第一个结点*/
  int  j = 0;
  while ( p ) {
      p = p->Next; /* 当前p指向的是第 j 个结点*/
      j++;
  }
  return j;
}
```

时间性能为  $O(n)$ 。



# 主要操作实现

## ● 2. 查找

### (1) 按序号查找: FindKth

```
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, 返回指针 */
  else return NULL;
  /* 否则返回空 */
}
```

### (2) 按值查找: Find

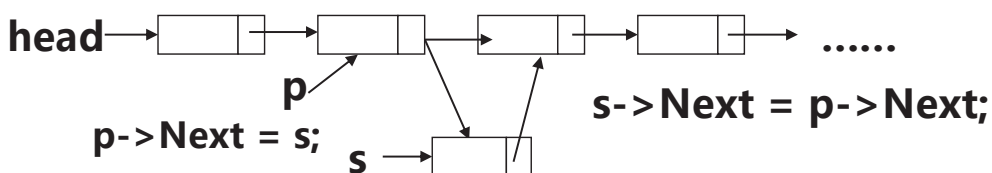
```
List Find( int K, List PtrL )
{ List p = PtrL;
  while ( p != NULL && p->Data
    != X )
    p = p->Next;
  return p;
}
```

平均时间性能为  $O(n)$ 。

# 主要操作实现

## ● 3. 插入(第 $i - 1$ ( $1 \leq i \leq n + 1$ )个结点后插入)

- (1) 先构造一个新结点, 用s指向
- (2) 再找到链表的第 $i - 1$ 个结点, 用p指向
- (3) 然后修改指针, 插入结点(p之后插入新结点是s)

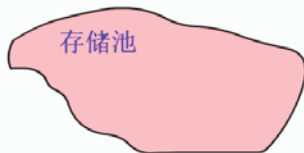


思考: 修改指针的两个步骤如果交换一下, 将会发生什么?

# 单链表结点的插入操作演示

## 单链表结点的插入

bobo



本课件由于屏幕空间所限，给定的链表长度为5。可输入的位置*i*应介于1和6之间。  
*x*表示一个可插入字符。  
本课件只演示插入一个结点的过程。



InsertList(head, , )

➡ 插入 ⬅ 重来

请输入要插入的字符*x*

请输入要插入的位置*i*

# 头插法建单链表演示

## 头插法建无头结点单链表

bobo



请输入插入链表中的字符串

apple



建立



重建

# 尾插法建单链表演示

## 尾插法建表

bobo



本课件中,存储池代表一片存储空间。由于屏幕空间所限,这里建立的单链表最多只能有6个结点。



请输入插入链表中的字符串

apple



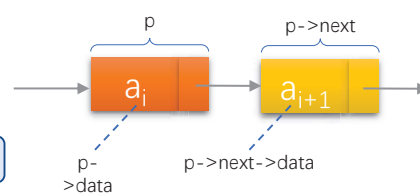
建立



重建

## 单链表结点的插入操作样例

数据变化



关键代码

```
Status ListInsert(LinkList *L,int i,ElemType e)
{
    int j;
    LinkList p,s;
    p = *L;
    j = 1;
    while (p && j < i)
    {
        p = p->next;
        ++j;
    }
    if (!p || j > i)
        return ERROR;
    s = (LinkList)malloc(sizeof(Node));
    s->data = e;
    s->next = p->next;
    p->next = s;
    return OK;
}
```

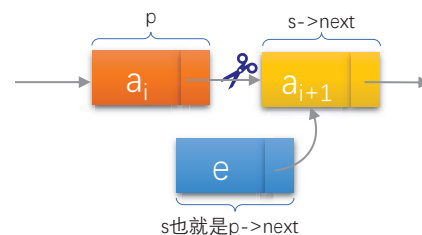
i=3, j=1,2,3

# 单链表结点的插入操作样例

## 关键代码

```
Status ListInsert(LinkList *L,int i,ElemType e)
{
    int j;
    LinkList p,s;
    p = *L;
    j = 1;
    while (p && j < i)
    {
        p = p->next;
        ++j;
    }
    if (!p || j > i)
        return ERROR;
    s = (LinkList)malloc(sizeof(Node));
    s->data = e;
    s->next = p->next;
    p->next = s;
    return OK;
}
```

## 数据变化

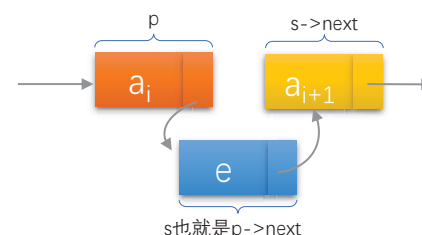


# 单链表结点的插入操作样例

## 关键代码

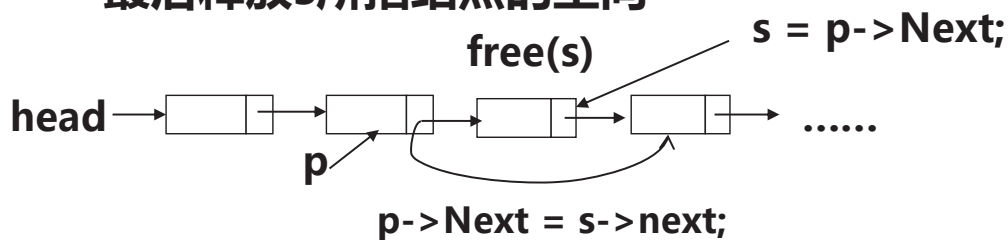
```
Status ListInsert(LinkList *L,int i,ElemType e)
{
    int j;
    LinkList p,s;
    p = *L;
    j = 1;
    while (p && j < i)
    {
        p = p->next;
        ++j;
    }
    if (!p || j > i)
        return ERROR;
    s = (LinkList)malloc(sizeof(Node));
    s->data = e;
    s->next = p->next;
    p->next = s;
    return OK;
}
```

## 数据变化



# 主要操作实现

- 4. 删除(第 $i$ ( $1 \leq i \leq n$ )个位置上的结点)
  - 先找到链表的第 $i-1$ 个结点, 用 $p$ 指向
  - 再用指针 $s$ 指向要被删除的结点( $p$ 的下一个结点)
  - 然后修改指针, 删除 $s$ 所指结点
  - 最后释放 $s$ 所指结点的空间



思考: 操作指针的几个步骤如果随意改变, 将会发生什么?

## 单链表结点的删除操作演示

### 单链表结点的删除

bobo



由于屏幕空间所限, 课件的链表长度是5, 所以您输入的删除位置 $i$ 应大于0小于等于5。我们仅删除一个结点以做演示。

存储池

DeleteList(head, )

删除 重来

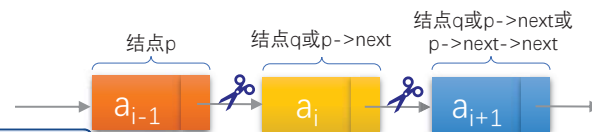
请输入要删除的位置 $i$

# 单链表结点的删除操作样例

## 关键代码

```
Status ListDelete(LinkList *L,int i,ElemType *e)
{
    int j;
    LinkList p,q;
    p = *L;
    j = 1;
    while (p->next && j < i)
    {
        p = p->next;
        ++j;
    }
    if (!(p->next) || j > i)
        return ERROR;
    q = p->next;
    p->next = q->next;
    *e = q->data;
    free(q);
    return OK;
}
```

## 数据变化



# 单链表结点的删除操作样例

## 关键代码

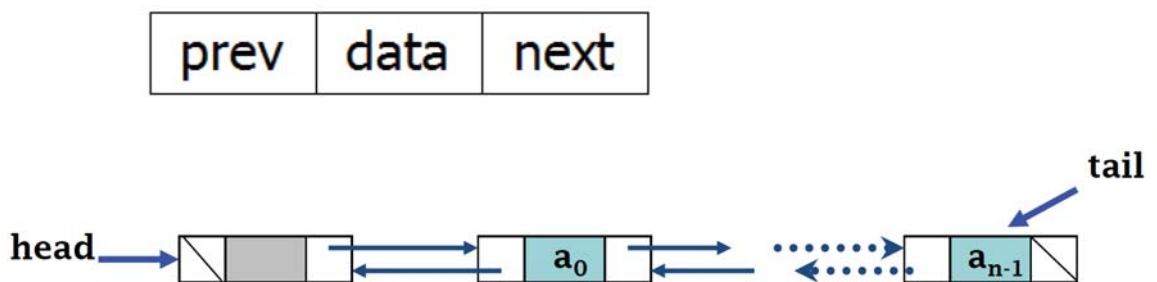
```
Status ListDelete(LinkList *L,int i,ElemType *e)
{
    int j;
    LinkList p,q;
    p = *L;
    j = 1;
    while (p->next && j < i)
    {
        p = p->next;
        ++j;
    }
    if (!(p->next) || j > i)
        return ERROR;
    q = p->next;
    p->next = q->next;
    *e = q->data;
    free(q);
    return OK;
}
```

## 数据变化



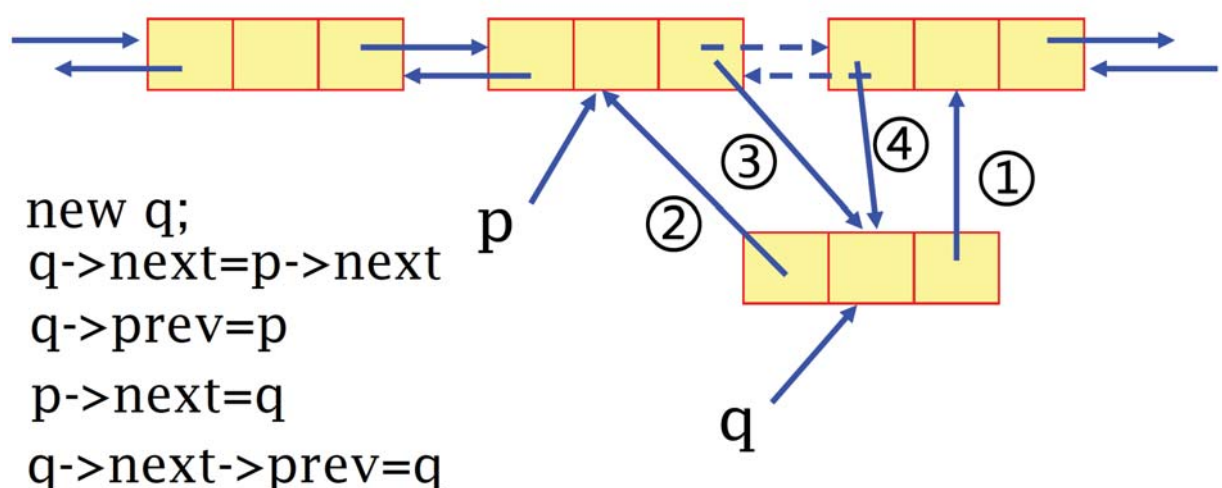
# 双链表

- 为弥补单链表的不足,而产生双链表
  - 单链表的 next 字段仅仅指向后继结点,不能有效地找到前驱,反之亦然
  - 增加一个指向前驱的指针

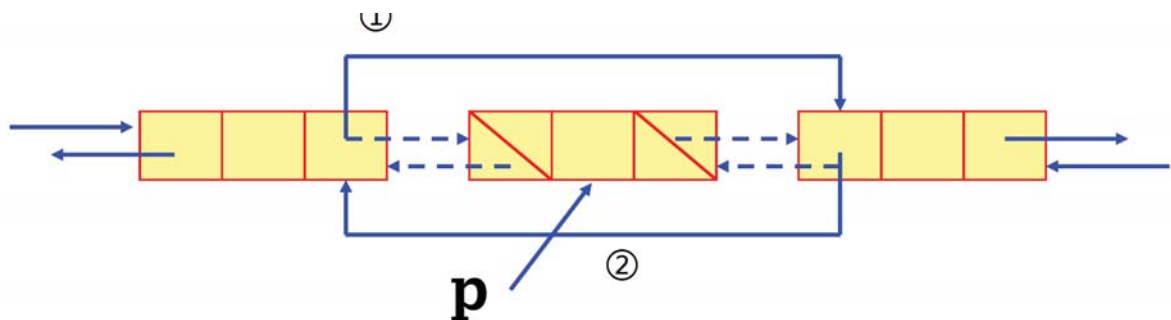


## 双链表的插入

在 p 所指结点后面插入一个新结点



# 双链表的删除



删除  $p$  所指的结点

$p \rightarrow \text{prev} \rightarrow \text{next} = p \rightarrow \text{next}$

$p \rightarrow \text{next} \rightarrow \text{prev} = p \rightarrow \text{prev}$

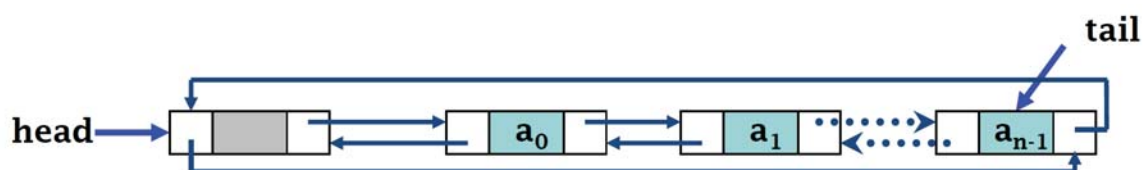
$p \rightarrow \text{next} = \text{NULL}$

$p \rightarrow \text{prev} = \text{NULL}$

- 如果马上删除  $p$ 
  - 则可以不赋空

# 循环链表

- 将单链表或者双链表的头尾结点链接起来，就是一个循环链表
- 不增加额外存储花销，却给不少操作带来了方便
  - 从循环表中任一结点出发，都能访问到表中其他结点





# 链表的边界条件

---

- 几个特殊点的处理
  - 头指针处理
  - 非循环链表尾结点的指针域保持为 NULL
  - 循环链表尾结点的指针回指头结点
- 链表处理
  - 空链表的特殊处理
  - 插入或删除结点时指针勾链的顺序
  - 指针移动的正确性
    - 插入
    - 查找或遍历

## 课间习题

---

- 下面关于线性表的叙述中，正确的是哪些？
  - A. 线性表采用链接存储，便于插入和删除操作。
  - B. 线性表采用顺序存储，便于进行插入和删除操作。
  - C. 线性表采用顺序存储，必须占用一片连续的存储单元。
  - D. 线性表采用链接存储，不必占用一片连续的存储单元。

## 课间习题

---

### ● 下面的叙述中正确的是：

- A. 线性表在链式存储时，插入第 $i$ 个元素的时间与 $i$ 的数值成正比。
- B. 线性表在顺序存储时，查找第 $i$ 个元素的时间与 $i$ 的数值无关。
- C. 线性表在顺序存储时，查找第 $i$ 个元素的时间与 $i$ 的数值成正比。
- D. 线性表在链式存储时，查找第 $i$ 个元素的时间与 $i$ 的数值无关。

## 顺序表和链表的比较

---

- 顺序表的主要优点
  - 没有使用指针，不用花费额外开销
  - 线性表元素的读访问非常简洁便利
- 链表的主要优点
  - 无需事先了解线性表的长度
  - 允许线性表的长度动态变化
  - 能够适应经常插入删除内部元素的情况
- 总结
  - 顺序表是存储静态数据的不二选择
  - 链表是存储动态变化数据的良方

# 顺序表和链表的比较

---

- 顺序表
  - 插入、删除运算时间代价  $O(n)$ ，查找则可常数时间完成
  - 预先申请固定长度的连续空间
  - 如果整个数组元素很满，则没有结构性存储开销
- 链表
  - 插入、删除运算时间代价  $O(1)$ ，但找第 $i$ 个元素运算时间代价  $O(n)$
  - 存储利用指针，动态地按照需要为表中新的元素分配存储空间
  - 每个元素都有结构性存储开销

## 顺序表和链表的存储密度

---

$n$  表示线性表中当前元素的数目，  
 $P$  表示指针的存储单元大小（通常为 4 bytes）  
 $E$  表示数据元素的存储单元大小  
 $D$  表示可以在数组中存储的线性表元素的最大数目

- 空间需求
  - 顺序表的空间需求为  $DE$
  - 链表的空间需求为  $n(P + E)$
- $n$  的临界值，即  $n > DE / (P+E)$ 
  - $n$  越大，顺序表的空间效率就更高
  - 如果  $P = E$ ，则临界值为  $n = D / 2$

# 应用场合选择

---

- 顺序表不适用的场合
  - 经常插入删除时，不宜使用顺序表
  - 线性表的最大长度也是一个重要因素
- 链表不适用的场合
  - 当读操作比插入删除操作频率大时，不应选择链表
  - 当指针的存储开销，和整个结点内容所占空间相比其比例较大时，应该慎重选择

# 应用场合选择

---

- 顺序表
  - 结点总数目大概可以估计
  - 线性表中结点比较稳定（插入删除少）
  - $n > DE / (P + E)$
- 链表
  - 结点数目无法预知
  - 线性表中结点动态变化（插入删除多）
  - $n < DE / (P + E)$

# 休息时间



第3节休息

## 广义表(Generalized List)

**[例]** 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？比如，给定二元多项式：

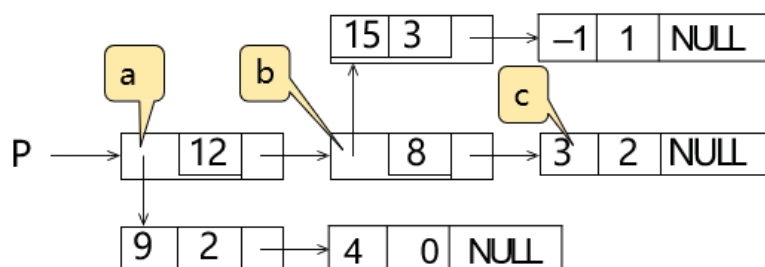
$$P(x, y) = 9x^{12}y^2 + 4x^{12} + 15x^8y^3 - x^8y + 3x^2$$

**[分析]** 可将上述二元多项式看成关于x的一元多项式

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

$$ax^{12} + bx^8 + cx^2$$

所以，上述二元多项式可以用“复杂”链表表示为：



# 广义表(Generalized List)

- 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
- 广义表中，这些元素不仅可以是单元素也可以是另一个广义表

```
typedef struct GNode{  
    int Tag; /*标志域：0表示该结点是单元素，1表示该结点是广义表 */  
    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;  
        struct GNode *SubList;  
    } URegion;  
    struct GNode *Next; /* 指向后继结点 */  
} GList;
```

Tag	Data	Next
	SubList	

## 多重链表

- 多重链表：链表中的节点可能同时隶属于多个链
  - 多重链表中结点的指针域会有多个，如前面例子包含了Next和SubList两个指针域
  - 但包含两个指针域的链表并不一定是多重链表，比如双向链表不是多重链表
- 多重链表有广泛的用途
  - 基本上如树、图这样相对复杂的数据结构都可以采用多重链表方式实现存储



# 多重链表

**[例]** 矩阵可以用二维数组表示，但存在两个缺陷：

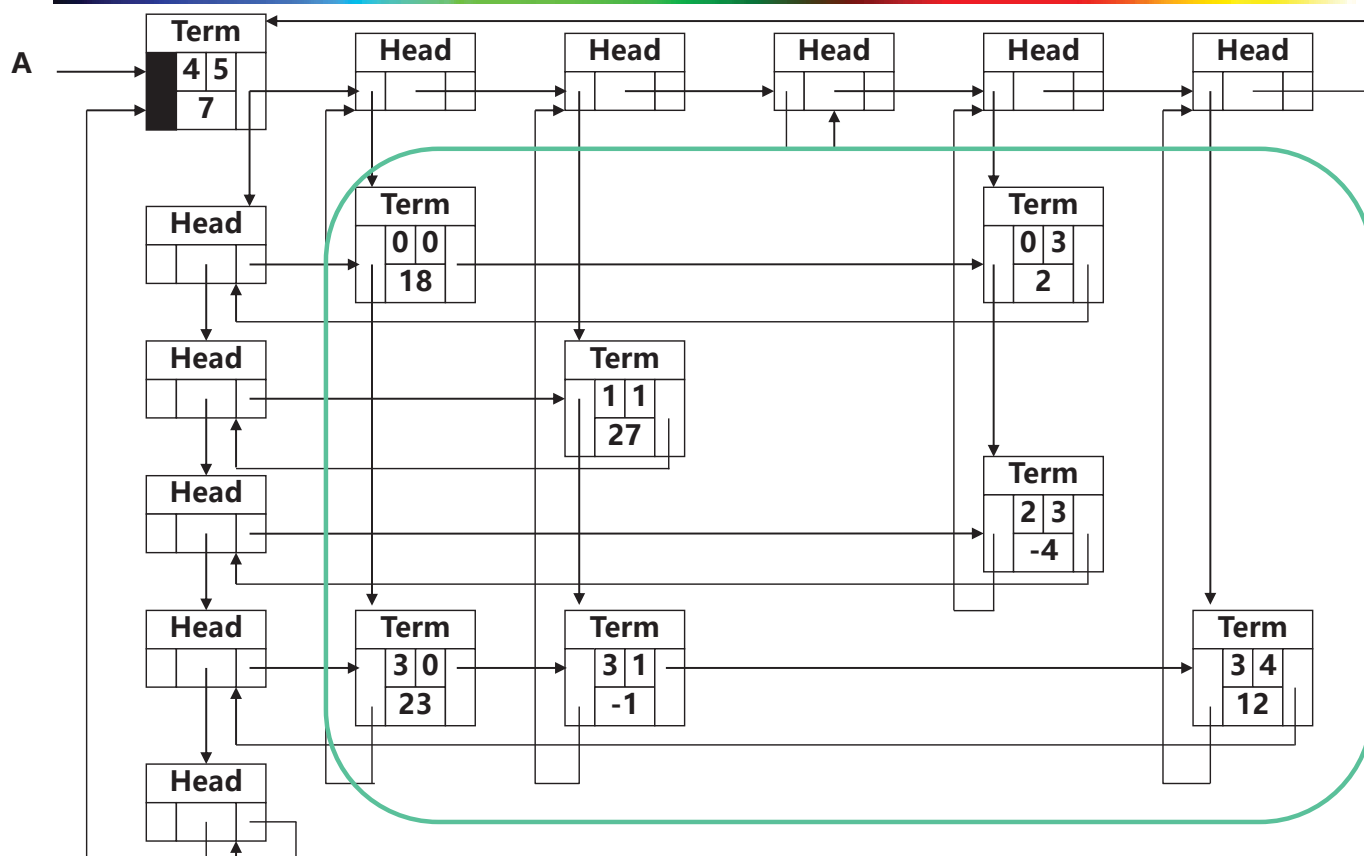
- 数组的**大小需要事先确定**
- 对于“**稀疏矩阵**”，将造成大量的**存储空间浪费**

$$A = \begin{bmatrix} 18 & 0 & 0 & 2 & 0 \\ 0 & 27 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 0 \\ 23 & -1 & 0 & 0 & 12 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 2 & 11 & 0 & 0 & 0 \\ 3 & -4 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 13 & 0 \\ 0 & -2 & 0 & 0 & 10 & 7 \\ 6 & 0 & 0 & 5 & 0 & 0 \end{bmatrix}$$

**[分析]** 采用一种典型的多重链表——**十字链表**来存储稀疏矩阵

- 只存储矩阵非0元素项
  - 结点的**数据域**：行坐标Row、列坐标Col、数值Value
- 每个结点通过**两个指针域**，把同行、同列串起来
  - 行指针(或称为向右指针)**Right**
  - 列指针(或称为向下指针)**Down**

## 矩阵A的多重链表图



# 多重链表抽象数据结构

- 用一个标识域Tag来区分头结点和非0元素结点
- 头节点的标识值为“Head”，矩阵非0元素结点的标识值为“Term”
- 稀疏矩阵的数据结构可定义为：

```
#define MAXSIZE 50      /* 矩阵最大非0元素个数 */
typedef enum { Head, Term } NodeTag;
typedef struct MatrixNode *PtrMatrix;
typedef struct TermNode {
    int Row;
    int Col;
    ElementType Value;
};
typedef struct MatrixNode {
    PtrMatrix Down;
    PtrMatrix Right;
    NodeTag Tag;
    union {
        PtrMatrix Next;
        TermNode Term;
    } URegion;
};
PtrMatrix HeadNode [MAXSIZE];
```

Tag		
Down	URegion	Right

(a) 结点的总体结构

Term			
Down	Row	Col	Right
	Value		

(b) 矩阵非0元素结点

Head		
Down	Next	Right

(c) 头结点

## 小结

- 单链表结构和顺序存储结构对比

### 存储分配方式

- 顺序存储结构用一段连续的存储单元依次存储线性表的数据元素
- 单链表采用链式存储结构，用一组任意的存储单元存放线性表的元素

### 时间性能

- 查找
  - 顺序存储结构 $O(1)$
  - 单链表 $O(n)$
- 插入和删除
  - 顺序存储结构需要平均移动表长一半的元素，时间为 $O(n)$
  - 单链表在线出某位置的指针后，插入和删除时间仅为 $O(1)$

### 空间性能

- 顺序存储结构需要预分配存储空间，分大了，浪费，分小了易发生上溢
- 单链表不需要分配存储空间，只要有就可以分配，元素个数也不受限制



# 小结

## 线性表

顺序存储结构

链式存储结构

单链表

静态链表

循环链表

双向链表

# 习题

- 阅读算法，回答下列问题

问1：说明语句S1的功能

```
LinkedList mynote(LinkedList L) {  
    if(L && L->next) {  
        q = L; L = L->next; p = L;  
        S1: while(p->next) p=p->next;  
        S2: p->next = 1; q->next = NULL;  
    }  
    return L;  
}
```

问2：说明语句S2的功能

问3：设链表表示的线性表为 $(a_1, a_2, \dots, a_n)$ ，写出算法返回值所表示的线性表

# 习题

---

- 已知两个链表A和B分别表示两个集合，其元素递增排列。请设计算法求出A与B的交集，并存放于A链表中

---

## 2.2 堆栈

---

# 什么是堆栈

- 计算机如何进行表达式求值？

**[例]** 算术表达式  $5 + 6/2 - 3*4$ 。正确理解：

$$5 + 6/2 - 3*4 = 5 + 3 - 3*4 = 8 - 3*4 = 8 - 12 = -4$$

- 由两类对象构成的：
  - 运算数，如2、3、4
  - 运算符，如+、-、\*、/
- 不同运算符优先级不一样

## 后缀表达式

- **中缀表达式**：运算符位于两个运算数之间。  
如， $a + b * c - d / e$
- **后缀表达式**：运算符位于两个运算数之后。  
如， $a b c * + d e / -$

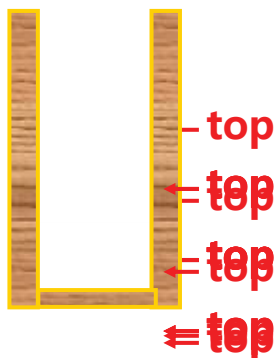
**[例]**  $6\ 2\ /\ 3\ -\ 4\ 2\ *\ +\ =\ ?$

- **后缀表达式求值策略**：从左向右“扫描”，逐个处理运算数和运算符
  - 遇到运算数怎么办？如何“记住”目前还未参与运算的数？
  - 遇到运算符怎么办？对应的运算数是什么？

**启示：**需要有存储方法，能顺序存储运算数，并在需要时“倒序”输出！

# 后缀表达式示例

**[例]** 6 2 / 3 - 4 2 \* + = ?      8



对象: 6 (运算数)	对象: 2 (运算数)
对象: / (运算符)	对象: 3 (运算数)
对象: - (运算符)	对象: 4 (运算数)
对象: 2 (运算数)	对象: * (运算符)
对象: + (运算符)	Pop: 8

$\pi(N) = O(N)$ 。不需要知道运算符的优先规则。

## 堆栈的抽象数据类型描述

- **堆栈(Stack):** 具有一定操作约束的线性表
  - 只在一端(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)**

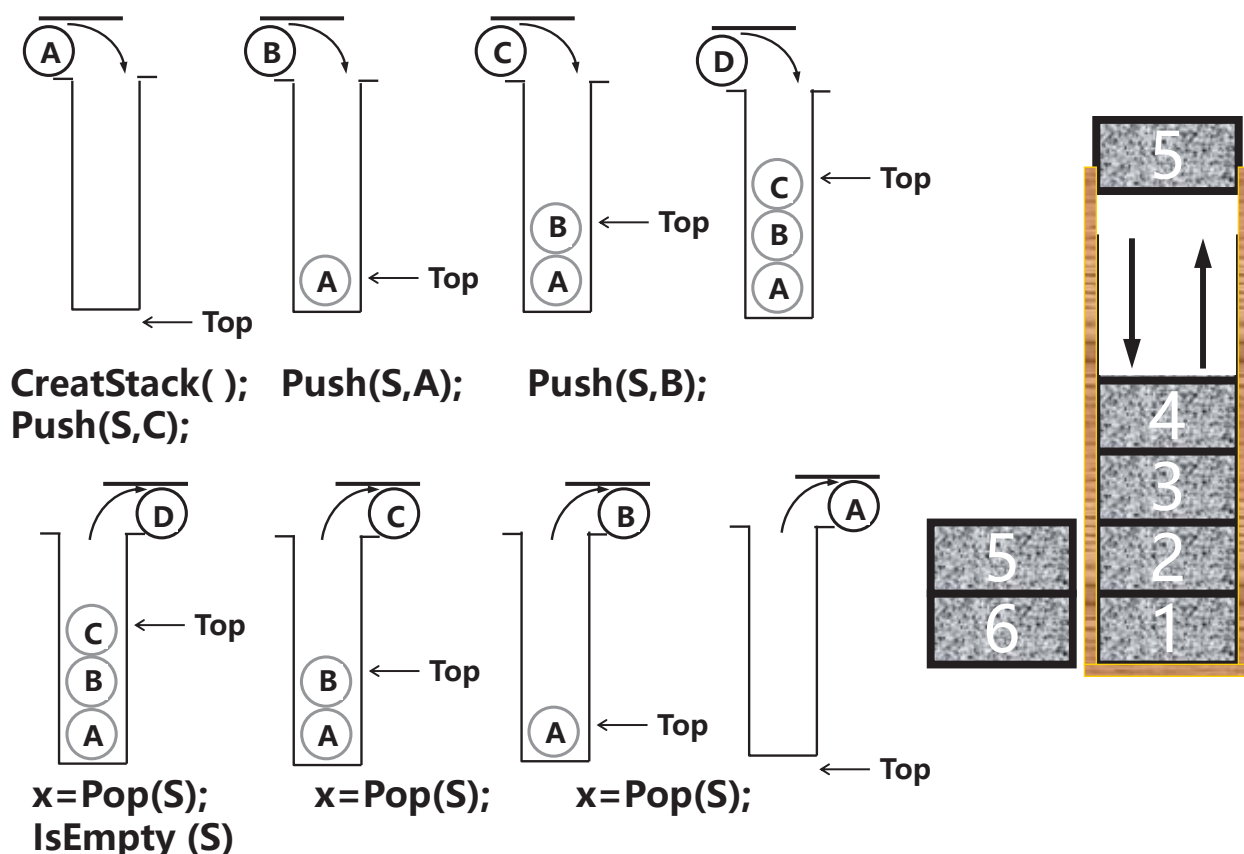


# 堆栈的抽象数据类型描述

- **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：**长度为MaxSize的堆栈S E Stack，堆栈元素item E ElementType

- **Stack CreateStack( int MaxSize )：**生成空堆栈，其最大长度为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;

# 堆栈的抽象数据类型描述




# 堆栈的抽象数据类型描述

- Push和Pop可以穿插交替进行
- 按照操作系列：
  - Push(S, A)->Push(S, B)->Push(S, C)->Pop(S)  
->Pop(S) ->Pop(S)。堆栈输出是？
  - Push(S, A) ->Pop(S) ->Push(S, B) ->Push(S, C)  
->Pop(S) ->Pop(S)。堆栈输出是？

**[例]** 如果三个字符按ABC顺序压入堆栈

- ABC的所有排列都可能是出栈的序列吗？
- 可以产生CAB这样的序列吗？

## 堆栈操作演示

栈与递归 F=Factorial()-请输入n的值

```
(1) int Factorial(int n);  
(2) { int temp;  
(3)   if(n==0)  
(4)     return 1;  
(5)   else{  
(6)     temp=Factorial(n-1);  
(7)     temp=n*temp;  
(8)     return temp;  
(9)   }  
(10) }
```

参数表      返回地址


Top →

Start

Cooling

# 栈的顺序存储实现

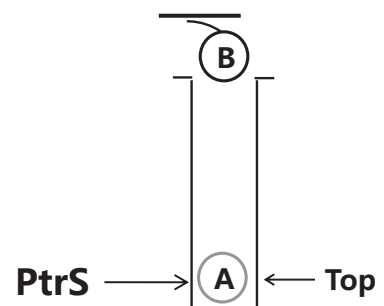
- 栈的顺序存储结构通常由一个**一维数组**和一个记录**栈顶**元素位置的变量组成

```
#define MaxSize <储存数据元素的最大个数>
typedef struct {
    ElementType Data[MaxSize];
    int Top;
} Stack;
```

# 栈的顺序存储实现

## (1)入栈

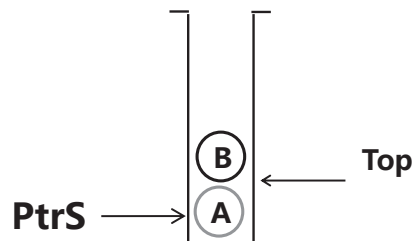
```
void Push( Stack *PtrS, ElementType item )
{
    if ( PtrS->Top == MaxSize-1 ) {
        printf( "堆栈满" ); return;
    }else {
        PtrS->Data[++(PtrS->Top)] = item;
        return;
    }
}
```



# 栈的顺序存储实现

## (2)出栈

```
ElementType Pop( Stack *PtrS )
{
    if ( PtrS->Top == -1 ) {
        printf( "堆栈空" );
        return ERROR; /* ERROR是ElementType的特殊值，标志错误 */
    } else
        return ( PtrS->Data[(PtrS->Top)--] );
}
```



# 栈的顺序存储实现

**[例]** 请用一个数组实现两个堆栈，要求最大地利用数组空间，使数组只要有空间入栈操作就可以成功

**[分析]** 一种比较聪明的方法是使这两个栈分别从数组的**两头**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。此时，最大化地利用了数组空间。

```
#define MaxSize <存储数据元素的最大个数>
struct DStack {
    ElementType Data[MaxSize];
    int Top1; /* 堆栈 1 的栈顶指针 */
    int Top2; /* 堆栈 2 的栈顶指针 */
} S;

S.Top1 = -1;
S.Top2 = MaxSize;
```



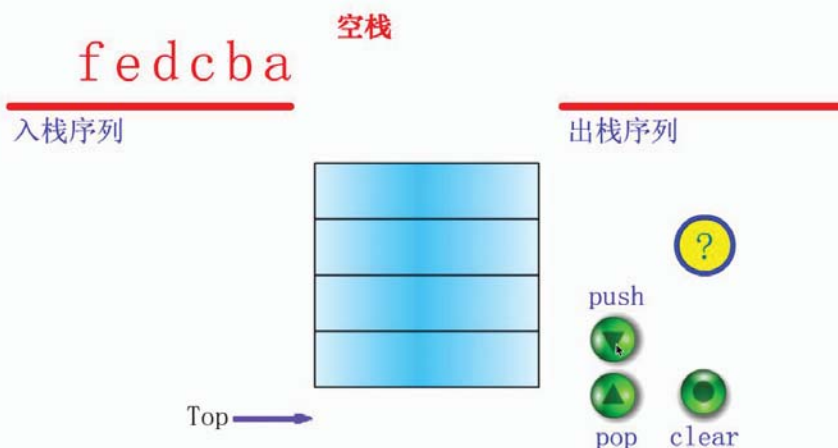
# 栈的顺序存储实现

```
void Push( struct DStack *PtrS, ElementType item, int Tag )
{ /* Tag作为区分两个堆栈的标志, 取值为1和2 */
  if ( PtrS->Top2 - PtrS->Top1 == 1 ) { /*堆栈满*/
    printf( "堆栈满" ); return ;
  }
  if ( Tag == 1 ) /* 对第一个堆栈操作 */
    PtrS->Data[++(PtrS->Top1)] = item;
  else /* 对第二个堆栈操作 */
    PtrS->Data[--(PtrS->Top2)] = item;
}

ElementType Pop( struct DStack *PtrS, int Tag )
{ /* Tag作为区分两个堆栈的标志, 取值为1和2 */
  if ( Tag == 1 ) { /* 对第一个堆栈操作 */
    if ( PtrS->Top1 == -1 ) { /*堆栈1空*/
      printf( "堆栈1空" ); return NULL;
    } else return PtrS->Data[(PtrS->Top1)--];
  } else { /* 对第二个堆栈操作 */
    if ( PtrS->Top2 == MaxSize ) { /*堆栈2空*/
      printf( "堆栈2空" ); return NULL;
    } else return PtrS->Data[(PtrS->Top2)++];
  }
}
```

## 栈操作演示

顺序栈 (假设为四个存储空间)



# 栈操作

最先进栈的元素，是不是就只能最后出栈呢？

## 栈的链式存储实现

- 栈的链式存储结构实际上就是一个单链表，叫做链栈。插入和删除操作只能在链栈的栈顶进行

栈顶指针Top应该在链表的哪一头？

```
typedef struct Node{
    ElementType Data;
    struct Node *Next;
} LinkStack;
LinkStack *Top;
```

- (1) 堆栈初始化（建立空栈）
- (2) 判断堆栈S是否为空



```
LinkStack *CreateStack()
{ /* 构建一个堆栈的头结点，返回指针 */
    LinkStack *S;
    S = malloc( sizeof(struct Node ));
    S->Next = NULL;
    return S;
}

int IsEmpty( LinkStack *S )
{ /*判断堆栈S是否为空，若为空函数返回整数1，
   否则返回0 */
    return ( S->Next == NULL );
}
```

# 栈的链式存储实现

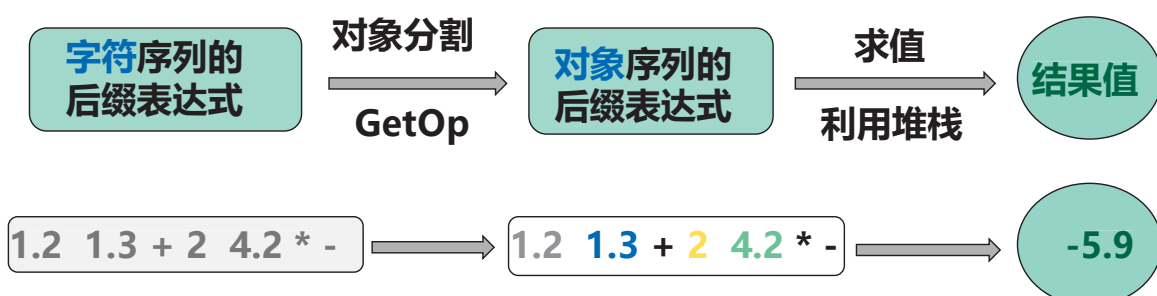
```
void Push( ElementType item, LinkStack *S )
{ /* 将元素item压入堆栈S */
    struct Node *TmpCell;
    TmpCell = malloc( sizeof( struct Node ) );
    TmpCell->Element = item;
    TmpCell->Next = S->Next;
    S->Next = TmpCell;
}

ElementType Pop( LinkStack *S )
{ /* 删除并返回堆栈S的栈顶元素 */
    struct Node *FirstCell;
    ElementType TopElem;
    if( IsEmpty( S ) ) {
        printf( "堆栈空" ); return NULL;
    } else {
        FirstCell = S->Next;
        S->Next = FirstCell->Next;
        TopElem = FirstCell->Element;
        free(FirstCell);
        return TopElem;
    }
}
```

## 堆栈应用：表达式求值

- 回忆：应用堆栈实现后缀表达式求值的基本过程：
  - 从左到右读入后缀表达式的各项(运算符或运算数)

- 1.运算数：入栈；
- 2.运算符：从堆栈中弹出适当数量的运算数，计算并结果入栈；
- 3.最后，堆栈顶上的元素就是表达式的结果值。



# 中缀表达式求值

- **基本策略：**将中缀表达式转换为后缀表达式，然后求值——如何将中缀表达式转换为后缀？
- **观察一个简单例子：**  $2+9/3-5 \rightarrow 2\ 9\ 3\ /\ +\ 5\ -$ 
  - 1. 运算数相对顺序不变
  - 2. 运算符顺序发生改变
- 需要存储“等待中”的运算符
- 要将当前运算符与“等待中”的最后一个运算符比较

堆栈!

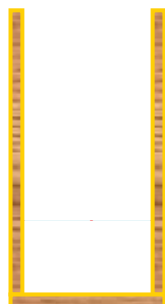
有括号怎么办?

## 中缀表达式求值示例

[例]  $a * (b + c) / d = ?$

$a\ b\ c\ +\ *\ d\ /\$

输出:  $a\ b\ c\ +\ *\ d\ /\$



← top

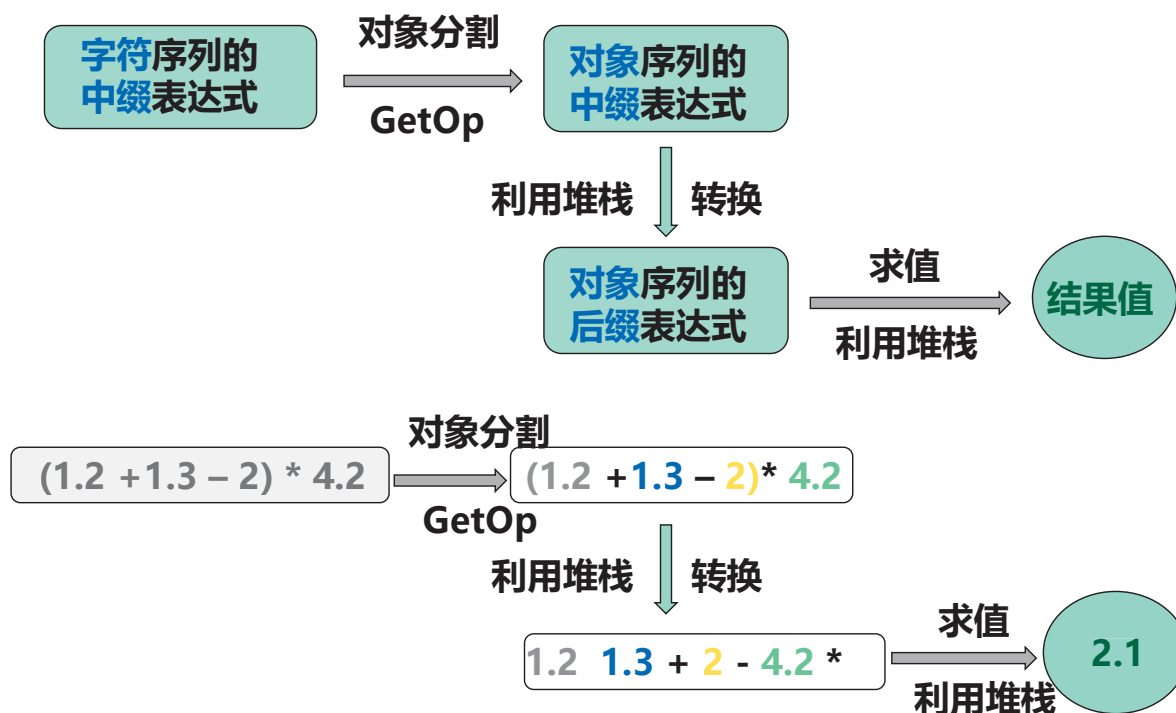
输入对象: $a$ (操作数)	输入对象: $*$ (乘法)
输入对象: $($ (左括号)	输入对象: $b$ (操作数)
输入对象: $+$ (加法)	输入对象: $c$ (操作数)
输入对象: $)$ (右括号)	输入对象: $/$ (除法)
输入对象: $d$ (操作数)	

$* \geq / ?$

$T(N) = O(N)$

NO?!

# 中缀表达式求值示例



## 中缀表达式如何转为后缀表达式

- 从头到尾读取中缀表达式的每个对象，对不同对象按不同的情况处理
  - 1. 运算数：直接输出；
  - 2. 左括号：压入堆栈；
  - 3. 右括号：将栈顶的运算符弹出并输出，直到遇到左括号(出栈，不输出)；
  - 4. 运算符：
    - 若优先级大于栈顶运算符时，则把它压栈；
    - 若优先级小于等于栈顶运算符时，将栈顶运算符弹出并输出；
    - 再比较新的栈顶运算符，直到该运算符大于栈顶运算符优先级为止，然后将该运算符压栈；
  - 5. 若各对象处理完毕，则把堆栈中存留的运算符一并输出。

# 中缀转换为后缀示例 $(2*(9+6/3-5)+4)$

步骤	待处理表达式	堆栈状态 (底 $\leftrightarrow$ 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 +

## 堆栈的其他应用

- 函数调用及递归实现
- 深度优先搜索
- 回溯算法

# 习题

---

- 若已知一个栈的入栈序列是 $1, 2, 3, \dots, n$ , 其输出序列为 $p_1, p_2, p_3, \dots, p_n$ , 若 $p_1=n$ , 则 $p_i$ 为 \_\_\_\_\_
- 若一个栈以向量 $V[1\dots n]$ 存储, 初始栈顶指针 $top$ 设为 $n+1$ , 则元素 $x$ 进展的正确操作是 \_\_\_\_\_

# 习题

---

- 回文是指正读反读均相同的字符序列, 如"abba"和"abdba"均是回文, 但"good"不是。写一个算法判断给定的字符向量是否为回文



# 休息时间

---



第4节休息

---

## 2.3 队列

---



# 什么是队列

---

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

## 队列的抽象数据类型描述

---

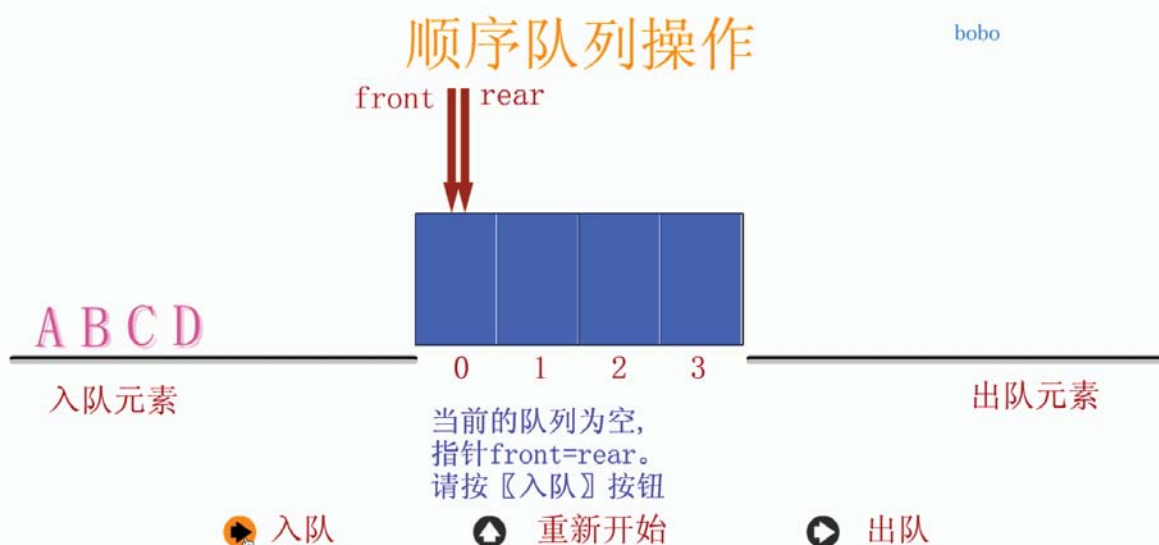
- **类型名称:** 队列(Queue)
- **数据对象集:** 一个有0个或多个元素的有穷线性表
- **操作集:** 长度为MaxSize的队列Q Queue, 队列元素  $item \in ElementType$ 
  - **Queue CreatQueue(int MaxSize):** 生成长度为MaxSize的空队列;
  - **int IsFullQ(Queue Q, int MaxSize):** 判断队列Q是否已满;
  - **void AddQ(Queue Q, ElementType item):** 将数据元素  item插入队列Q中;
  - **int 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

- 队列的顺序存储结构通常由一个**一维数组**和一个记录队列头元素位置的变量**front**以及一个记录队列尾元素位置的变量**rear**组成

```
#define MaxSize <储存数据元素的最大个数>
struct QNode {
    ElementType Data[ MaxSize ];
    int rear;
    int front;
};
typedef struct QNode *Queue;
```

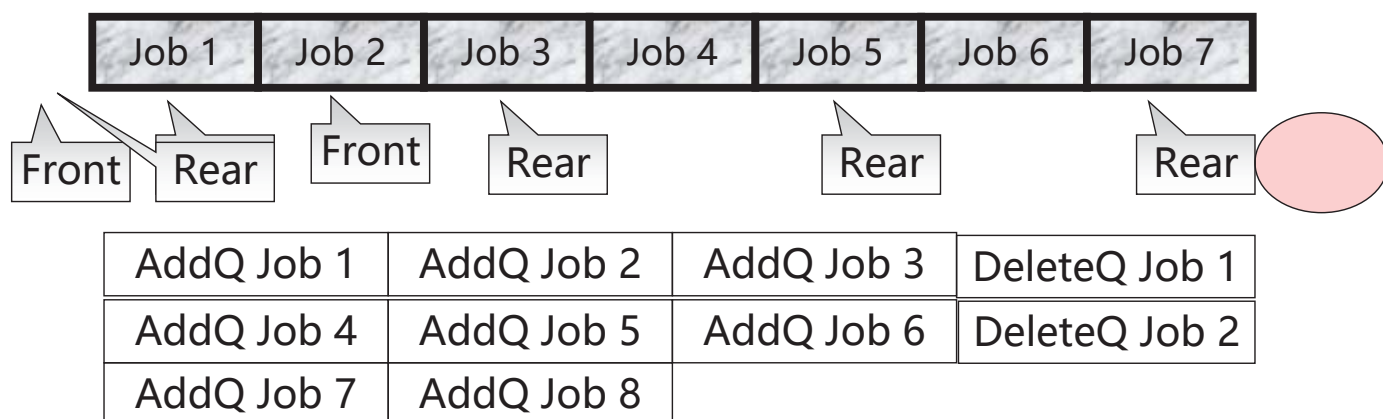
## 顺序队列操作演示



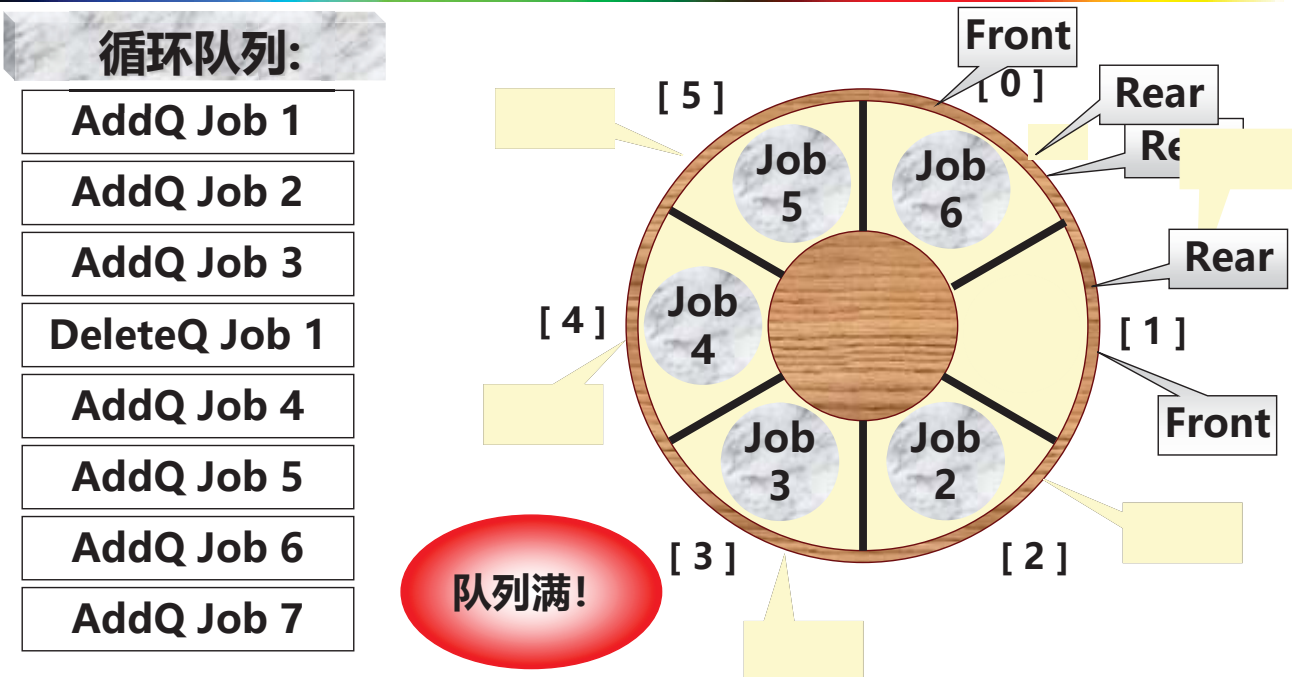
# 队列的顺序存储实现

- 队列的顺序存储结构通常由一个**一维数组**和一个记录队列头元素位置的变量**front**以及一个记录队列尾元素位置的变量**rear**组成

## [例] 一个工作队列



# 队列的顺序存储实现



**注:** 如果数据结构中增加一个 **Size** 域, 用来区分队列“空”和“满”的话, 可以“节省”一个数据元素的存储单元。但是会带来算法描述的复杂性。

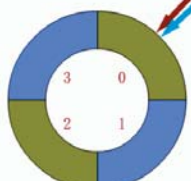
# 循环队列操作演示

循环队列操作演示

bobo

当前队列中元素总数

0



A B C D E F G H

入队元素

出队元素

循环队列当前状态为空。请按【入队】将元素入队

注意：  
本课件仅提供8个元素入队作示范，如入队元素超过8个请按【重新开始】

入队 重新开始 出队

## 队列的顺序存储实现

### (1)入队列

Front和rear指针的移动采用“加1取余”法，体现了顺序存储的“循环使用”。

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

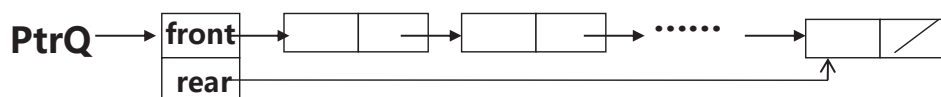
```
ElementType DeleteQ ( Queue *PtrQ )
{
    if ( PtrQ->front == PtrQ->rear ) {
        printf( "队列空" );
        return ERROR;
    } else {
        PtrQ->front = (PtrQ->front+1)% MaxSize;
        return PtrQ->Data[PtrQ->front];
    }
}
```

# 队列的链式存储实现

- 队列的链式存储结构也可以用一个单链表实现。插入和删除操作分别在链表的两头进行

队列指针front和rear应该分别指向链表的哪一头？

```
typedef struct Node{
    ElementType Data;
    struct Node *Next;
}QNode;
typedef struct {    /* 链队列结构 */
    QNode *rear;    /* 指向队尾结点 */
    QNode *front;   /* 指向队头结点 */
} LinkQueue;
LinkQueue *PtrQ ;
```



# 队列的链式存储实现

- 不带头结点的链式队列出队操作的一个示例：

```
ElementType DeleteQ ( LinkQueue *PtrQ )
{
    QNode *FrontCell;
    ElementType FrontElem;

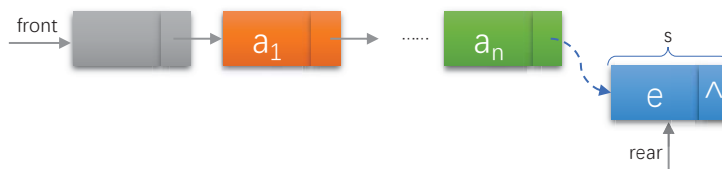
    if ( PtrQ->front == NULL ) {
        printf( "队列空" ); return ERROR;
    }
    FrontCell = PtrQ->front;
    if ( PtrQ->front == PtrQ->rear ) /* 若队列只有一个元素 */
        PtrQ->front = PtrQ->rear = NULL; /* 删除后队列置为空 */
    /*
    else
        PtrQ->front = PtrQ->front->Next;
    FrontElem = FrontCell->Data;
    free( FrontCell ); /* 释放被删除结点空间 */
    return FrontElem;
}
```

# 链式存储入队样例

## 关键代码

```
Status EnQueue(LinkQueue *Q, QElemType e)
{
    QueuePtr s=(QueuePtr)malloc(sizeof(QNode));
    if(!s)
        exit(OVERFLOW);
    s->data=e;
    s->next=NULL;
    Q->rear->next=s;
    Q->rear=s;
    return OK;
}
```

## 数据变化



# 链式存储出队样例

## 关键代码

```
Status DeQueue(LinkQueue *Q, QElemType *e)
{
    QueuePtr p;
    if(Q->front==Q->rear)
        return ERROR;
    p=Q->front->next;
    *e=p->data;
    Q->front->next=p->next;
    if(Q->rear==p)
        Q->rear=Q->front;
    free(p);
    return OK;
}
```

## 数据变化



# 习题

- 设栈S和队列Q的初始状态为空，元素 $e_1$ 、 $e_2$ 、 $e_3$ 、 $e_4$ 、 $e_5$ 和 $e_6$ 依次进入栈S，一个元素出栈后即进入Q，若6个元素出队的序列是 $e_2$ 、 $e_4$ 、 $e_3$ 、 $e_6$ 、 $e_5$ 和 $e_1$ ，则栈S的容量至少应该是 \_\_\_\_\_

# 习题

- 数组Q[n]用来表示一个循环队列，f 为当前队列头元素的前一位置，r 为队尾元素的位置，假定队列中元素的个数小于n，计算队列中元素个数的公式为 \_\_\_\_\_
- 用两个栈实现一个队列的功能，给出算法及思路



# 习题

---

- 如果允许在循环队列的两端都可以进行插入和删除操作。要求：
  - (1) 写出循环队列的类型定义；
  - (2) 写出“从队尾删除”和“从队头插入”的算法。

---

## 2.4 应用实例

---

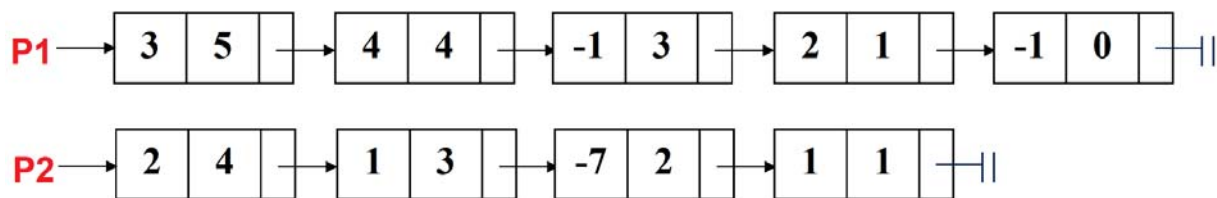
# 应用1：多项式加法运算

$$\begin{array}{r} P1 = 3X^5 + 4X^4 - X^3 + 2X - 1 \\ + P2 = \phantom{3X^5 + } 2X^4 + X^3 - 7X^2 + X \\ \hline P = 3X^5 + 6X^4 - 7X^2 + 3X - 1 \end{array}$$

主要思路：相同指数的项系数相加，其余部分进行拷贝

## 多项式加法运算

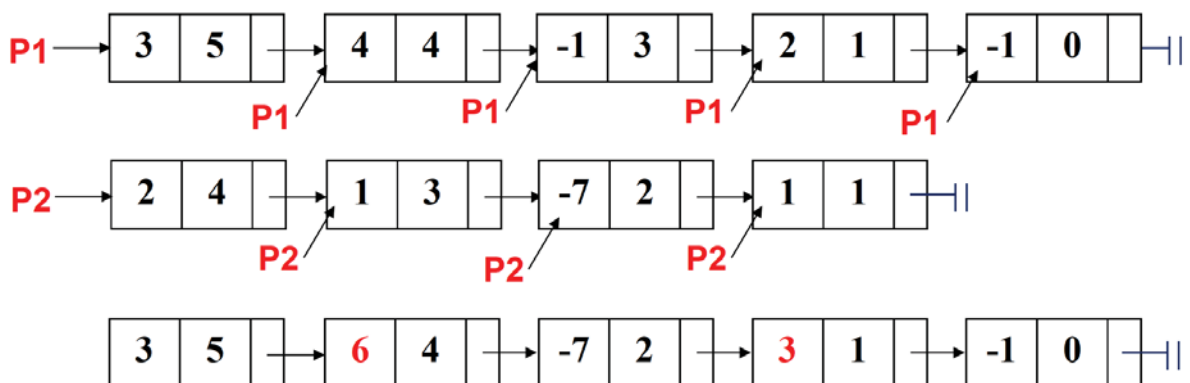
- 采用不带头结点的**单向链表**，按照**指数递减**的顺序排列各项



```
struct PolyNode
{
    // 系数
    int coef;
    // 指数
    int
    expon;
    struct PolyNode *link;    // 指向下一个节点的指针
};
typedef struct PolyNode *Polynomial;
Polynomial P1, P2;
```

# 多项式加法运算

- **算法思路：**两个指针P1和P2分别指向这两个多项式第一个结点，不断循环：
  - $P1 \rightarrow \text{expon} == P2 \rightarrow \text{expon}$ : 系数相加，若结果不为0，则作为结果多项式对应项的系数。同时，P1和P2都分别指向下一项
  - $P1 \rightarrow \text{expon} > P2 \rightarrow \text{expon}$ : 将P1的当前项存入结果多项式，并使P1指向下一项
  - $P1 \rightarrow \text{expon} < P2 \rightarrow \text{expon}$ : 将P2的当前项存入结果多项式，并使P2指向下一项
- 当某一多项式处理完时，将另一个多项式的所有结点依次复制到结果多项式中去



# 多项式加法运算

Polynomial PolyAdd (Polynomial P1, Polynomial P2)

```
{
    Polynomial front, rear, temp; int
    sum;
    rear = (Polynomial) malloc(sizeof(struct PolyNode));
    front = rear; /* 由front记录结果多项式链表头结点 */
    while ( P1 && P2 ) /* 当两个多项式都有非零项待处理时 */
    switch ( Compare(P1->expon, P2->expon) ) {
        case 1:
            Attach( P1->coef, P1->expon, &rear);
            P1 = P1->link;
            break;
        case -1:
            Attach(P2->coef, P2->expon, &rear);
            P2 = P2->link;
            break;
        case 0:
            sum = P1->coef + P2->coef;
            if ( sum ) Attach(sum, P1->expon, &rear);
            P1 = P1->link;
            P2 = P2->link;
            break;
    }
    /* 将未处理完的另一个多项式的所有节点依次复制到结果多项式中去 */
    for ( ; P1; P1 = P1->link ) Attach(P1->coef, P1->expon, &rear);
    for ( ; P2; P2 = P2->link ) Attach(P2->coef, P2->expon, &rear);
    rear->link = NULL;
    temp = front;
    front = front->link; /* 令front指向结果多项式第一个非零项 */
    free(temp);          /* 释放临时空表头结点 */
    return front;
}
```

为方便表头插入，先产生一个临时空结点作为结果多项式链表头

P1中的数据项指数较大

P2中的数据项指数较大

两数据项指数相等

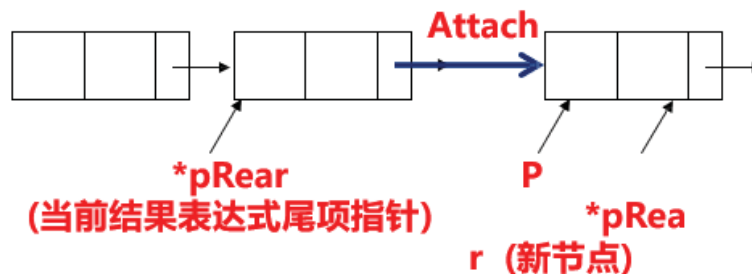
注意判断系数和是否为0

# 多项式加法运算

```
void Attach( int c, int e, Polynomial *pRear )
{
    /* 由于在本函数中需要改变当前结果表达式尾项指针的值, */
    /* 所以函数传递进来的是结点指针的地址, *pRear指向尾项*/
    Polynomial P;

    P = (Polynomial)malloc(sizeof(struct PolyNode)); /* 申请新结点 */
    P->coef = c;                                     /* 对新结点赋值 */
    P->expon = e;                                     /* */
    P->link=NULL;

    /* 将P指向的新结点插入到当前结果表达式尾项的后面 */
    /* (*pRear)->link = P; */
    *pRear = P; /* 修改pRear值 */
}
```



## 应用2：一元多项式的加法与乘法运算

- 设计函数分别求两个一元多项式的乘积与和

已知两个多项式:

$$(1) 3x^4 - 5x^2 + 6x - 2$$

$$(2) 5x^{20} - 7x^4 + 3x$$

多项式和:

$$5x^{20} - 4x^4 - 5x^2 + 9x - 2$$

## 题意理解

- 设计函数分别求两个一元多项式的乘积与和

已知两个多项式:

$$(1) 3x^4 - 5x^2 + 6x - 2$$

$$(2) 5x^{20} - 7x^4 + 3x$$

多项式的乘积:

$$(a+b)(c+d) = ac+ad+bc+bd$$

多项式乘积:

$$15x^{24}-25x^{22}+30x^{21}-10x^{20}-21x^8+35x^6-33x^5+14x^4-15x^3+18x^2-6x$$

## 题意理解

- 设计函数分别求两个一元多项式的乘积与和

### 输入样例:

4 3 4 -5 26 1 -2 0

3 5 20 -7 4 3 1

$$3x^4-5x^2+6x-2$$

$$5x^{20}-7x^4+3x$$

### 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

$$15x^{24}-25x^{22}+30x^{21}-10x^{20}-21x^8+35x^6-33x^5+14x^4-15x^3+18x^2-6x$$

$$5x^{20}-4x^4-5x^2+9x-2$$

# 求解思路

---

- 多项式表示
- 程序框架
- 读多项式
- 加法实现
- 乘法实现
- 多项式输出

## 多项式的表示

---

- 仅表示非零项

数组：

- 👍 编程简单、调试容易
- 👎 需要事先确定数组大小

链表：

- 👍 动态性强
- 👎 编程略为复杂、调试比较困难

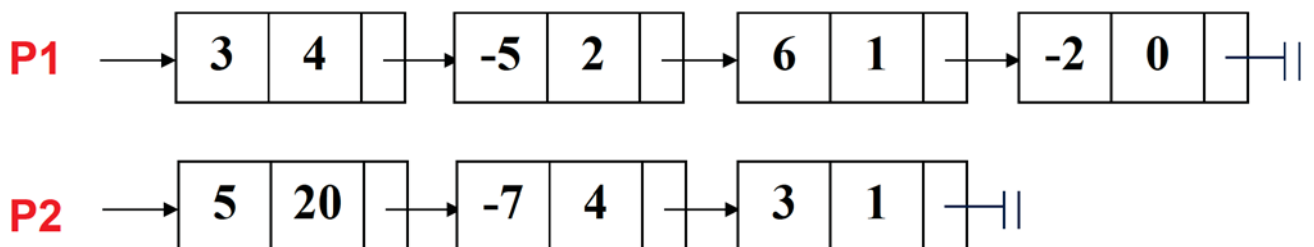
- 一种比较好的实现方法是：动态数组

下面介绍链表表示

# 多项式的表示

## ● 数据结构设计

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



## 程序框架搭建

```
int main()
{
    读入多项式 1
    读入多项式 2
    乘法运算并输出
    加法运算并输出

    return 0;
}
```

需要设计的函数:

- 读一个多项式
- 两多项式相乘
- 两多项式相加
- 多项式输出

```
int main()
{
    Polynomial P1, P2, PP, PS;

    P1 = ReadPoly();
    P2 = ReadPoly();
    PP = Mult( P1, P2 );
    PrintPoly( PP );
    PS = Add( P1, P2 );
    PrintPoly( PS );

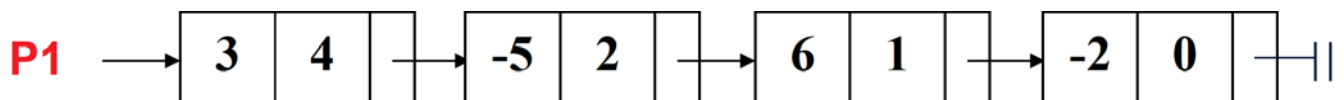
    return 0;
}
```

# 如何读入多项式

```
Polynomial ReadPoly()
```

```
{  
    .....  
    scanf("%d", &N);  
    .....  
    while ( N-- ) {  
        scanf("%d %d", &c, &e);  
        Attach(c, e, &Rear);  
    }  
    ..... return P;  
}
```

4 3 4 -5 2 6 1 -2 0



# 如何读入多项式

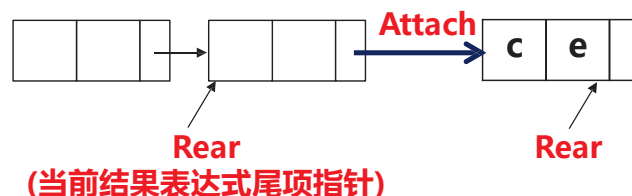
```
Polynomial ReadPoly()
```

```
{  
    .....  
    scanf("%d", &N);  
    .....  
    while ( N-- ) {  
        scanf("%d %d", &c, &e);  
        Attach(c, e, &Rear);  
    }  
    ..... return P;  
}
```

**Rear初值是多少?**

**两种处理方法:**

- 1. Rear初值为NULL, 在 Attach函数中根据Rear是否为NULL做不同处理





# 如何读入多项式

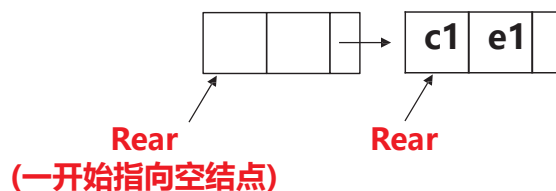
```
Polynomial ReadPoly()
```

```
{
    .....
    scanf("%d", &N);
    .....
    while ( N-- ) {
        scanf("%d %d", &c, &e);
        Attach(c, e, &Rear);
    }
    ..... return P;
}
```

**Rear初值是多少?**

**两种处理方法:**

- 1. Rear初值为NULL
- 2. **Rear指向一个空结点**

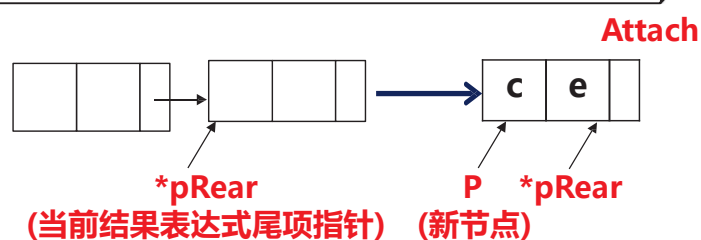


# 如何读入多项式

```
void Attach( int c, int e, Polynomial *pRear )
```

```
{
    Polynomial P;

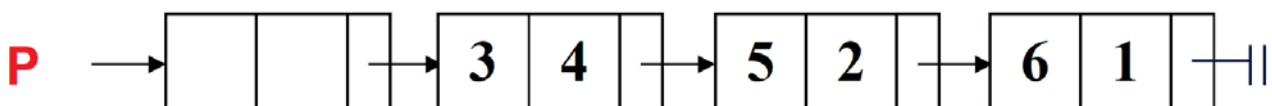
    P = (Polynomial)malloc(sizeof(struct PolyNode));
    P->coef = c; /* 对新结点赋值 */
    P->expon = e;
    P->link = NULL;
    (*pRear)->link = P;
    *pRear = P; /* 修改pRear值 */
}
```



# 如何读入多项式

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

    scanf("%d", &N);
    P = (Polynomial)malloc(sizeof(struct PolyNode)); /* 链表头空结点 */
    P->link = NULL;
    Rear = P;
    while ( N-- ) {
        scanf("%d %d", &c, &e);
        Attach(c, e, &Rear); /* 将当前项插入多项式尾部 */
    }
    t = P; P = P->link; free(t); /* 删除临时生成的头结点 */
    return P;
}
```



# 如何将两个多项式相加

```
Polynomial Add( Polynomial P1, Polynomial P2 )
{
    .....
    t1 = P1; t2 = P2;
    P = (Polynomial)malloc(sizeof(struct PolyNode));
    P->link = NULL; Rear = P;
    while (t1 && t2) {
        if (t1->expon == t2->expon) {
            .....
        }
        else if (t1->expon > t2->expon) {
            .....
        }
        else {
            .....
        }
    }
    while (t1) {
        .....
    }
    while (t2) {
        .....
    }
    .....
    return P;
}
```

# 如何将两个多项式相乘

## ● 方法

### ● 将乘法运算转换为加法运算

- 将P1当前项( $c_i, e_i$ )乘P2多项式, 再加入到结果多项式里

```
t1 = P1; t2 = P2;
P = (Polynomial)malloc(sizeof(struct PolyNode)); P->link = NULL;
Rear = P;
while (t2)
{
    Attach(t1->coef*t2->coef, t1->expon+t2->expon,
    &Rear); t2 = t2->link;
}
```

### ● 逐项插入

- 将P1当前项( $c1_i, e1_i$ )乘P2当前项 ( $c2_i, e2_i$ ),并插入到结果多项式中。关键是要找到插入位置
- 初始结果多项式可由P1第一项乘P2获得 (如上)

# 如何将两个多项式相乘

```
Polynomial Mult( Polynomial P1, Polynomial P2 )
{
    .....
    t1 = P1; t2 = P2;
    .....
    while (t2) { /* 先用P1的第1项乘以P2, 得到P */
        .....
    }
    t1 = t1->link;
    while (t1) {
        t2 = P2; Rear = P;
        while (t2) {
            e = t1->expon + t2->expon;
            c = t1->coef * t2->coef;
            .....
            t2 = t2->link;
        }
        t1 = t1->link;
    }
    .....
}
```

# 如何将两个多项式相乘

```
Polynomial Mult( Polynomial P1, Polynomial P2 )
{
    Polynomial P, Rear, t1, t2, t;
    int c, e;
    if (!P1 || !P2) return NULL;
    t1 = P1; t2 = P2;
    P = (Polynomial)malloc(sizeof(struct PolyNode));
    P->link = NULL; Rear = P;
    while (t2) {          /* 先用P1的第1项乘以P2, 得到P */
        Attach(t1->coef*t2->coef, t1->expon+t2->expon, &Rear);
        t2 = t2->link;
    }
    t1 = t1->link;
    while (t1) {
        t2 = P2; Rear = P;
        while (t2) {
            .....
            t2 = t2->link;
        }
        t1 = t1->link;
    }
    .....
}
```

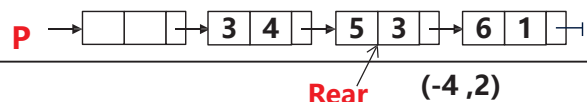
# 如何将两个多项式相乘

```
Polynomial Mult( Polynomial P1, Polynomial P2 )
{
    .....
    while (t1) {          P → [ ] [ ] → 3 4 → 5 3 → 6 1 → null
        t2 = P2; Rear = P;          插入: (-4, 2)
        while (t2) {
            e = t1->expon + t2->expon;
            c = t1->coef * t2->coef;
            while (Rear->link && Rear->link->expon > e)
                Rear = Rear->link;
            if (Rear->link && Rear->link->expon == e) {
                .....
            }
            else {
                .....
            }
            t2 = t2->link;
        }
        t1 = t1->link;
    }
    .....
}
```

# 如何将两个多项式相乘

Polynomial Mult( Polynomial P1, Polynomial P2 )

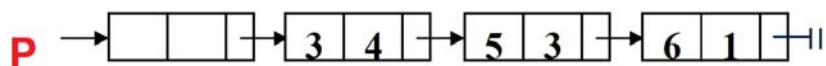
```
{
    .....
    while (Rear->link && Rear->link->expon > e)
        Rear = Rear->link;
    if (Rear->link && Rear->link->expon == e) {
        if (Rear->link->coef + c)
            Rear->link->coef += c;
        else {
            t = Rear->link;
            Rear->link = t->link;
            free(t);
        }
    }
    else {
        t = (Polynomial)malloc(sizeof(struct PolyNode));
        t->coef = c; t->expon = e;
        t->link = Rear->link;
        Rear->link = t; Rear = Rear->link;
    }
    .....
}
```



# 如何将两个多项式相乘

Polynomial Mult( Polynomial P1, Polynomial P2 )

```
{
    .....
    t1 = P1; t2 = P2;
    .....
    while (t2) { /* 先用P1的第1项乘以P2, 得到P */
        .....
    }
    t1 = t1->link;
    while (t1) {
        t2 = P2; Rear = P;
        while (t2) {
            e = t1->expon + t2->expon;
            c = t1->coef * t2->coef;
            .....
            t2 = t2->link;
        }
        t1 = t1->link;
    }
    t2 = P; P = P->link;
    free(t2);
    return P;
}
```



# 如何将两个多项式相乘

```
void PrintPoly( Polynomial P )
{ /* 输出多项式 */
    int flag = 0;                /* 辅助调整输出格式用 */
    if (!P) {printf("0 0\n"); return;}

    while ( P ) {
        if (!flag)
            flag = 1;
        else
            printf(" ");
        printf("%d %d", P->coef, P->expon);
        P = P->link;
    }
    printf("\n");
}
```

## 小结

### 栈

- 顺序栈
- 两栈共享空间
- 链栈

### 队列

- 顺序队列
- 循环队列
- 链队列