# 王道考研——数据结构

WWW.CSKAOYAN.COM

## 第三章 栈和队列

### 本章总览

本节内容

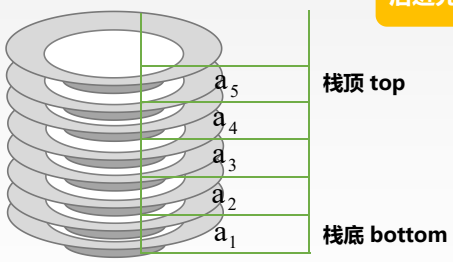
# 栈和队列

## 栈
## 栈的基本概念



## 栈的基本概念

后进先出(LIFO)




$S=(a_1, a_2, a_3, a_4, a_5)$

**栈(Stack)** 只允许在一端进行插入或删除操作的线性表

## 栈的基本概念

### 栈的基本操作

**InitStack(&S)**：初始化一个空栈S。
**StackEmpty(S)**：判断一个栈是否为空，若栈为空则返回true，否则返回false。
**Push(&S, x)**：进栈，若栈S未满，则将x加入使之成为新栈顶。
**Pop(&S, &x)**：出栈，若栈非空，则弹出栈顶元素，并用x返回。
**GetTop(S, &x)**：读栈顶元素，若栈非空则用x返回栈顶元素。
**ClearStack(&S)**：销毁栈，并释放S占用的内存空间。

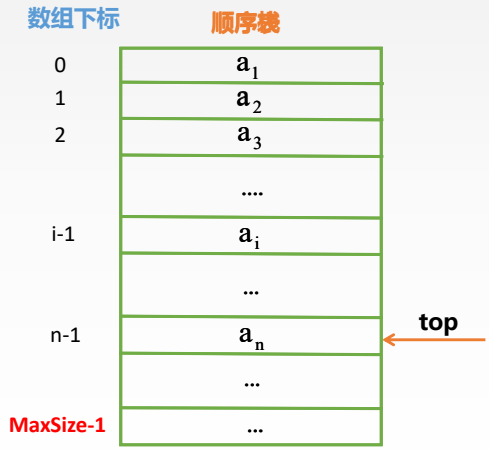


本节内容

# 栈和队列

## 栈
## 顺序存储结构

## 栈的顺序存储

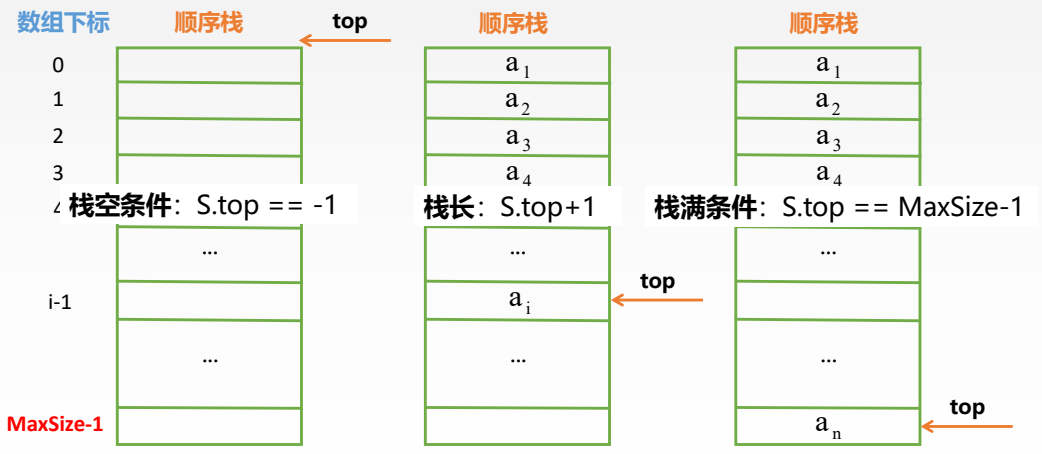



**顺序栈** 采用顺序存储的栈

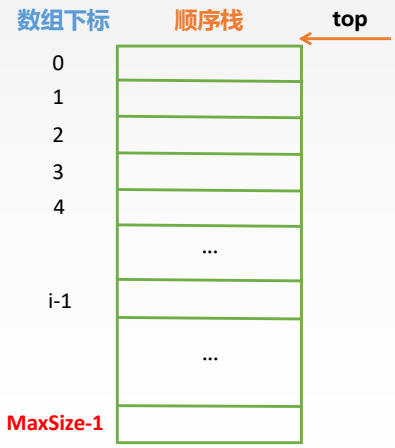
```
#define MaxSize 50
typedef struct{
    ElemType data[MaxSize];
    int top;
}SqStack;
```



## 栈的顺序存储



## 栈的顺序存储

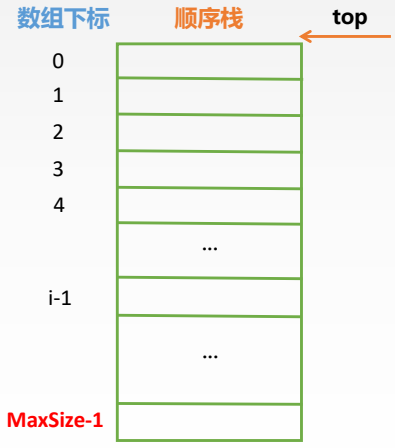



初始化

```
void InitStack(SqStack &S){
    S.top == -1;
}
```



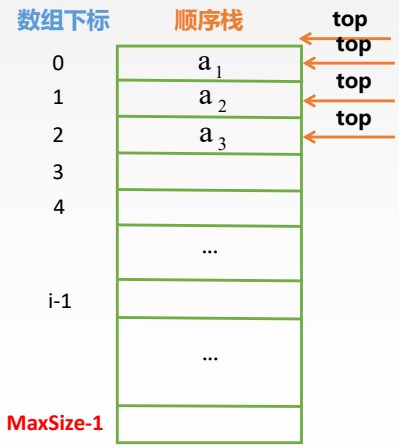
## 栈的顺序存储




判断栈空

```
bool StackEmpty(SqStack S){
    if(S.top == -1)
        return true;
    else
        return false;
}
```

## 栈的顺序存储

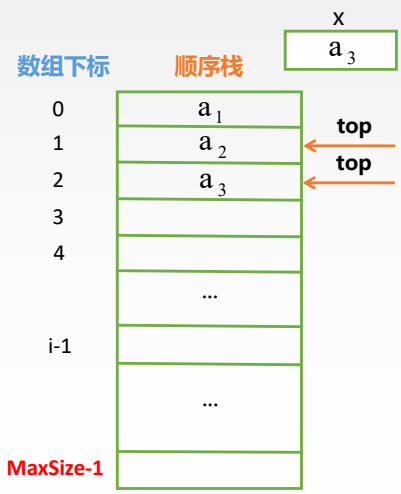



进栈

```
bool Push(SqStack &S, ElemType x){
    if(S.top == MaxSize-1)
        return false;
    S.data[++S.top] = x;
    return true;
}
```



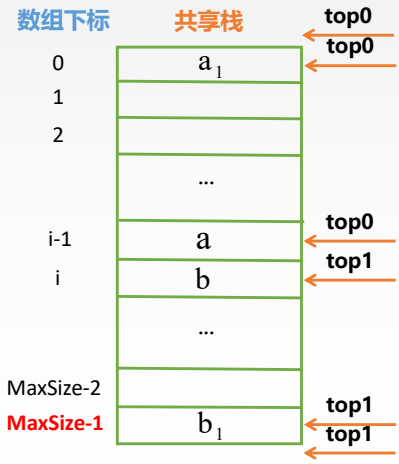
## 栈的顺序存储




出栈

```
bool Pop(SqStack &S, ElemType &x){
    if(S.top == -1)
        return false;
    x = S.data[S.top--];
    return true;
}
```

## 栈的顺序存储




**共享栈**

将两个栈底设置在共享空间的两端，栈顶向空间中间延伸

**判空**：0号栈 top == -1
1号栈 top == MaxSize

**栈满**：top1-top0 == 1

**优点** 存取时间复杂度仍为O(1)，但空间利用更加有效



本节内容

线性表

线性表
链式表示

本节内容

# 栈和队列

## 栈
## 链式存储结构



## 栈的链式存储

**链栈** 采用链式存储的栈


```
typedef struct Linknode{
    ElemType data;
    struct Linknode *next;
} *LiStack;
```

★ 所有的操作都在表头进行

本节内容

# 栈和队列

## 队列
### 队列的基本概念



## 队列的基本概念

先进先出(FIFO)


**队列(Queue)** 只允许在表的一端进行插入，表的另一端进行删除操作的线性表

## 队列的基本概念

### 队列的基本操作

**InitQueue(&Q)**：初始化队列，构造一个空队列Q。
**QueueEmpty(Q)**：判队列空，若队列Q为空返回true，否则返回false。
**EnQueue(&Q, x)**：入队，若队列Q未满，则将x加入使之成为新的队尾。
**DeQueue(&Q, &x)**：出队，若队列Q非空，则删除队头元素，并用x返回。
**GetHead(Q, &x)**：读队头元素，若队列Q非空则用x返回队头元素。
**ClearQueue(&Q)**：销毁队列，并释放队列Q占用的内存空间。

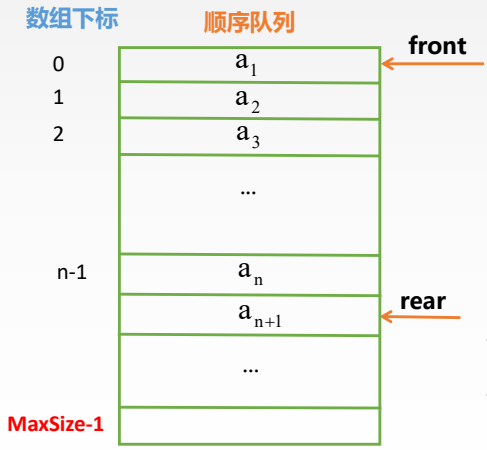


本节内容

# 栈和队列

# 队列
## 顺序存储结构

## 队列的顺序存储



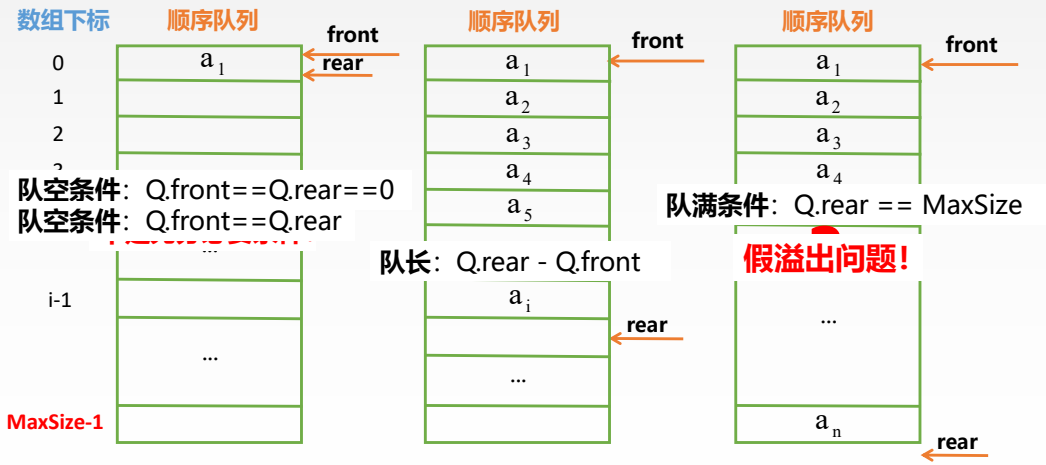

**顺序队** 采用顺序存储的队列

```
#define MaxSize 50
typedef struct{
    ElemType data[MaxSize];
    int front, rear;
}SqQueue;
```

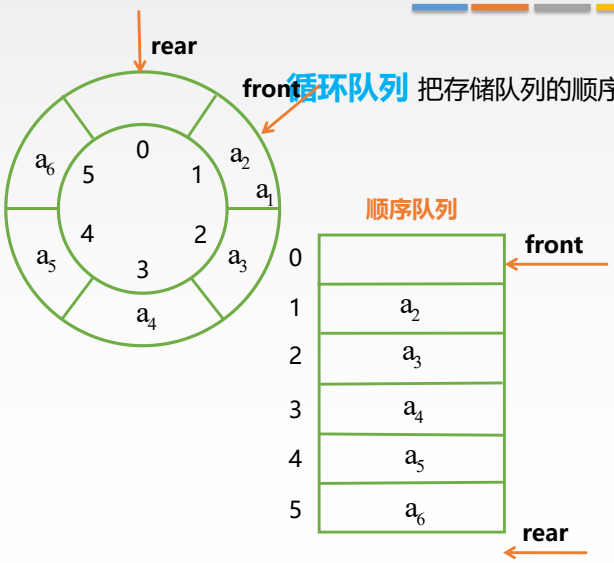
- front指向队头元素，rear指向队尾元素的下一位置
  （或front指向队头元素的前一位置，rear指向队尾元素）
- 初始时front == rear == 0



## 栈的顺序存储



## 队列的顺序存储



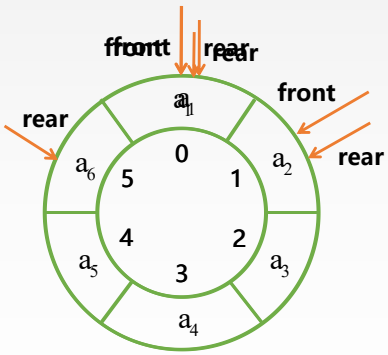

**循环队列** 把存储队列的顺序队列在逻辑上视为一个环

**取余（%MaxSize）**

**front指针移动:**
Q.front = (Q.front + 1) % MaxSize
**rear指针移动:**
Q.rear = (Q.rear + 1) % MaxSize
**队列长度:**
(Q.rear + MaxSize - Q.front) % MaxSize



## 队列的顺序存储



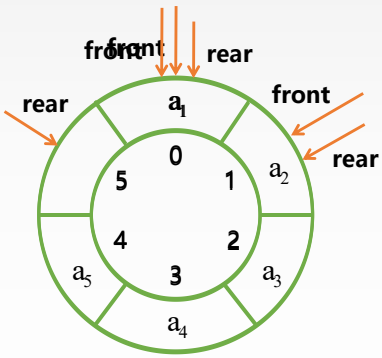

**队空条件**：Q.front==Q.rear

**队满条件**：Q.front==Q.rear

**队空条件 == 队满条件**

## 队列的顺序存储

### 方法一：牺牲一个存储单元



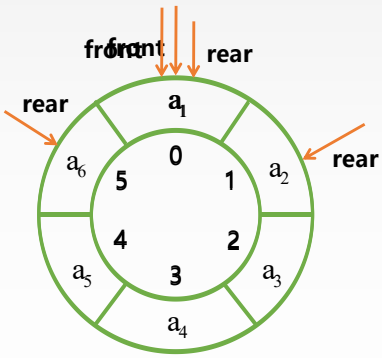

**队空条件**：Q.front==Q.rear

**队满条件**：
Q.front==（Q.rear+1）%MaxSize



## 队列的顺序存储

### 方法二：增加一个变量代表元素的个数



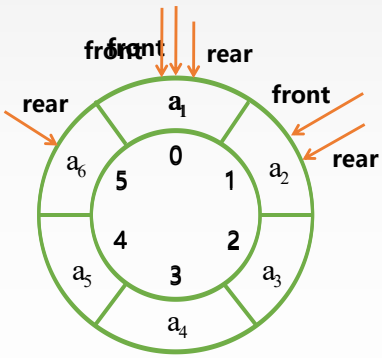

Q.size = 0

**队空条件**：Q.size== 0

**队满条件**：Q.size == MaxSize

## 队列的顺序存储

### 方法三：增加tag标识




tag = 0

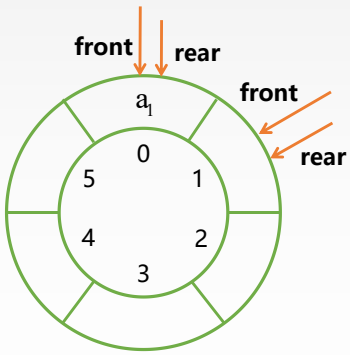
**队空条件**：Q.front==Q.rear&&tag==0

**队满条件**：Q.front==Q.rear&&tag==1



## 队列的顺序存储

### 循环队列的基本操作




**出队**

```
bool DeQueue(SqQueue &Q, ElemType &x){
    if(Q.rear == Q.front)
        return false;
    x = Q.data[Q.front];
    Q.front = (Q.front+1)%MaxSize;
    return true;
}
```

本节内容

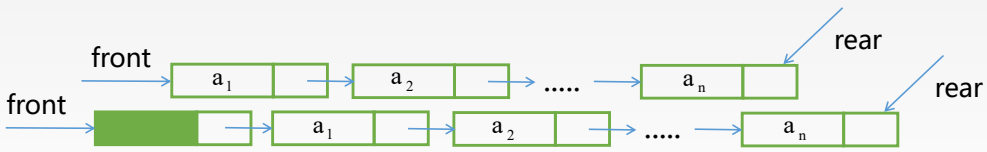
# 栈和队列

## 队列
### 链式存储结构



## 队列的链式存储

**链队** 采用链式存储的队列




```
typedef struct{
    ElemType data;
    struct LinkNode *next;
}LinkNode;

typedef struct{
    LinkNode *front, *rear;
}LinkQueue;
```

## 队列的链式存储

**初始化**


```
void InitQueue(LinkQueue &Q){
    Q.front = (LinkNode*)malloc(sizeof(LinkNode));
    Q.rear = Q.front;
    Q.front->next = NULL;
}
```



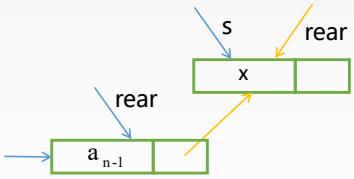
## 队列的链式存储

**判队空**


```
void isEmpty(LinkQueue Q){
    if(Q.front == Q.rear)
        return true;
    else
        return false;
}
```

## 队列的链式存储

**入队**

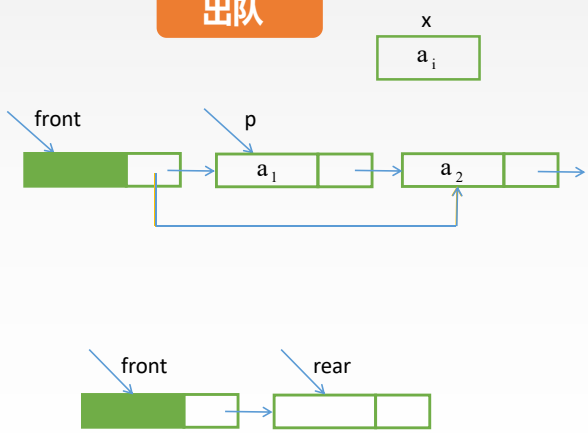



```
void EnQueue(LinkQueue &Q, ElemType x){
    LinkNode *s = (LinkNOde *)malloc(sizeof(LinkNode));
    s->data = x;
    s->next = NULL;
    Q.rear->next = s;
    Q.rear = s;
}
```



## 队列的链式存储

**出队**




```
bool DeQueue(LinkQueue &Q, ElemType &x){
    if(Q.front == Q.rear)
        return false;
    LinkNode *p = Q.front->next;
    x = p->data;
    Q.front->next = p->next;
    if(Q.rear == p)
        Q.rear = Q.front;
    free(p);
    return true;
}
```

本节内容

# 栈和队列

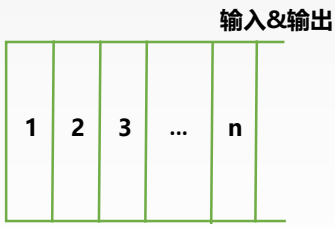
## 双端队列



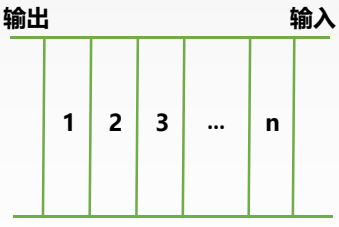
## 输出序列

### 连续输入和输出

输入序列：1, 2, 3, ..., n




输出序列：n,...,3,2,1

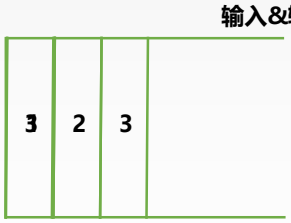



输出序列：1,2,3,...,n

## 输出序列

### 非连续输入和输出

**输入序列：1,2,3**




1 2 3
1 3 2
2 1 3
2 3 1
3 1 2 ✗
3 2 1



## 输出序列

### 非连续输入和输出

**输入序列：1,2,3,4**

| | | | | | |
|---|---|---|---|---|---|
| 1234 | 1243 | 1324 | 1342 | 1423 | 1432 |
| 2134 | 2143 | 2314 | 2341 | 2413 | 2431 |
| 3124 | 3142 | 3214 | 3241 | 3412 | 3421 |
| 4123 | 4132 | 4213 | 4231 | 4312 | 4321 |

★ 出栈序列中每一个元素后面所有比它小的元素组成一个递减序列

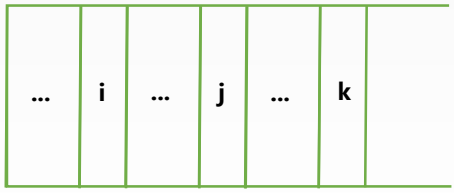
## 输出序列

### 非连续输入和输出

**进栈序列：1,2,3,.....,n**

★ 出栈序列中每一个元素后面所有比它小的元素组成一个递减序列

i<j<k




出栈序列：k,,j,,j,,i.,i



## 输出序列

### 合法出栈序列的个数

**进栈序列：1, 2, 3, ..., k, ..., n**

**出栈序列：**

...,1
...,2
.
.
...,k
.
.
...,n-1
...,n

f(n) = f(0) * f(n - 1) + f(1) * f(n - 2) + ... + f(n - 2) * f(1) + f(n - 1) * f(0)
且f(0) = f(1) = 1

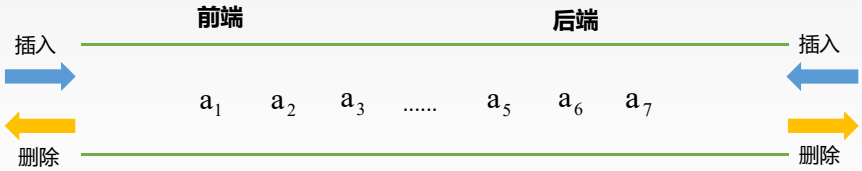
f(2) = f(0) * f(1) + f(1) * f(0) = 2
f(3) = f(0) * f(2) + f(1) * f(1) + f(2) * f(0) = 5
f(4) = f(0) * f(3) + f(1) * f(2) + f(2) * f(1) + f(3) * f(0) = 14

**f(n) = C(2n, n) / (n + 1)**

f(k-1)*f(n-k)

## 双端队列

**双端队列**允许两端都可以进行入队以及出队操作的队列

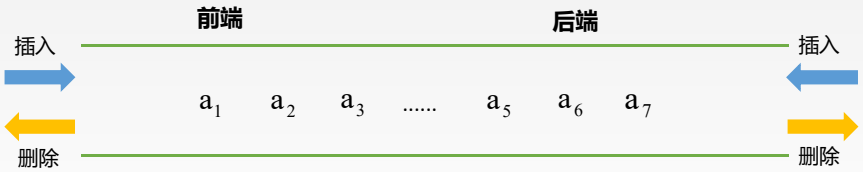



无论哪一端先出的元素在前，后出的元素在序列后

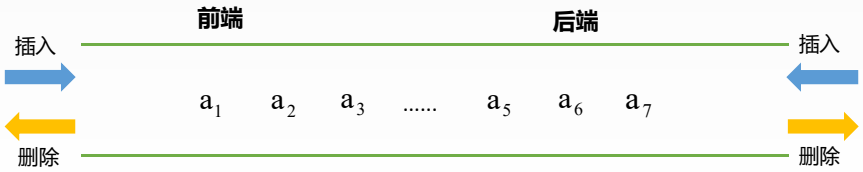


## 双端队列

**输出受限的双端队列**



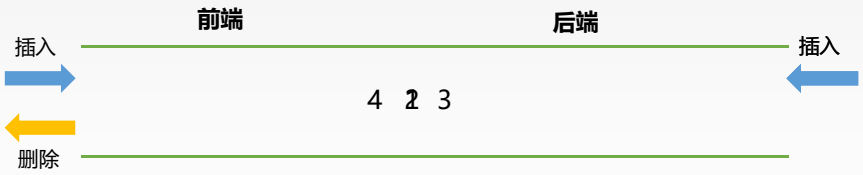

**输入受限的双端队列**



## 双端队列

**输入序列：1, 2, 3, 4**

**输出受限的双端队列**




4! -14 = 10

1423 2413 3124 3142 3412
4123 4132 4213 4231 4312

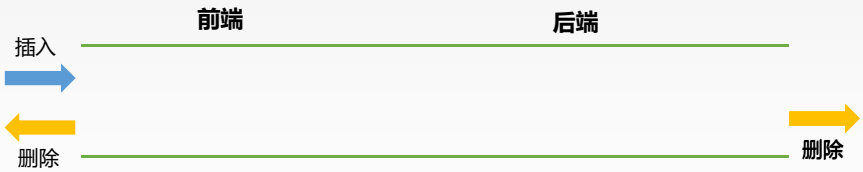
**4231 和 4132**



## 双端队列

**输入序列：1, 2, 3, 4**

**输入受限的双端队列**




4! -14 = 10

1423 2413 3124 3142 3412
4123 4132 4213 4231 4312

**4213 和 4231**

## 知识总结




本节内容

# 栈和队列

## 栈和队列的应用

## 栈的应用

### 括号匹配

[(A + B) * C] - [E - F]

[ ( ) ] [ ]

**匹配序列**

( [ ( ) ] )　　[ ] [ ] ( )　　( ) [ ( ) ]

**不匹配序列**

( [ ( ) ]　　] [ ] ( )　　( ] [ ( ) ]

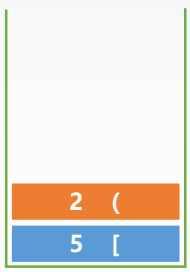


## 栈的应用

### 括号匹配

[ ( ) ] [ ]

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| [ | ( | ) | ] | [ | ] |




**算法思想：**

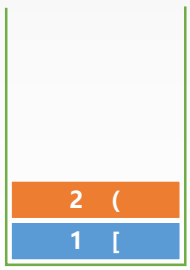
1) 初始一个空栈，顺序读入括号。
2) 若是右括号，则与栈顶元素进行匹配
   - 若匹配，则弹出栈顶元素并进行下一个元素
   - 若不匹配，则该序列不合法
3) 若是左括号，则压入栈中
4) 若全部元素遍历完毕，栈中非空则序列不合法

## 栈的应用

### 括号匹配

[ ( ] [ ]

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| [ | ( | ] | [ | ] |




**算法思想：**

1）初始一个空栈，顺序读入括号。
2）若是右括号，则与栈顶元素进行匹配
- 若匹配，则弹出栈顶元素并进行下一个元素
- 若不匹配，则该序列不合法

3）若是左括号，则压入栈中
4）若全部元素遍历完毕，栈中非空则序列不合法

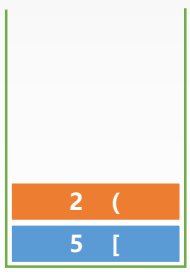


## 栈的应用

### 括号匹配

[ ( ) ] [

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| [ | ( | ) | ] | [ |




**算法思想：**

1）初始一个空栈，顺序读入括号。
2）若是右括号，则与栈顶元素进行匹配
- 若匹配，则弹出栈顶元素并进行下一个元素
- 若不匹配，则该序列不合法

3）若是左括号，则压入栈中
4）若全部元素遍历完毕，栈中非空则序列不合法

## 栈的应用

### 表达式求值

[(A + B)* C]- E[E F F]

| 前缀表达式 | 中缀表达式 | 后缀表达式 |
| --- | --- | --- |
| + AB | A + B | AB + |

[(A + B) * C] - [E - F]

?



## 栈的应用

### 表达式求值

[(A + B) * C] - [E - F]

- * + A B C - E F

前缀表达式 - * + A B C - E F

## 栈的应用

### 表达式求值

[(A + B) * C] - [E - F]

A B + C * E F - -

后缀表达式 **A B + C * E F - -**



## 栈的应用

### 表达式求值

( ( A + B ) * C ) - ( E - F )

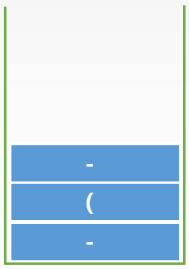
**算法思想：**
数字直接加入后缀表达式
运算符时：
**a.** 若为'(' ，入栈；
**b.** 若为')'，则依次把栈中的运算符加入后缀表达式，直到出现'('，并从栈中删除'('；
**c.** 若为'+'，'-'，'*'，'/'，
- 栈空，入栈；
- 栈顶元素为'('，入栈；
- 高于栈顶元素优先级，入栈；
- 否则，依次弹出栈顶运算符，直到一个优先级比它低的运算符或'('为止；

**d.** 遍历完成，若栈非空依次弹出所有元素。

## 栈的应用

### 表达式求值

( ( A + B ) * C ) - ( E - F )




**a.** 若为'('，入栈；
**b.** 若为')'，则依次把栈中的运算符加入后缀表达式，直到出现'('，并从栈中删除'('；
**c.** 若为'+'，'-'，'*'，'/'，
- 栈空，入栈；
- 栈顶元素为'('，入栈；
- 高于栈顶元素优先级，入栈；
- 否则，依次弹出栈顶运算符，直到一个优先级比它低的运算符或'('为止；

**d.** 遍历完成，若栈非空依次弹出所有元素。

**后缀表达式** A B + C * E F - -

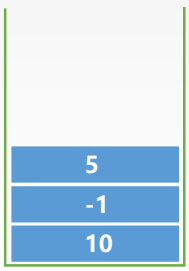


## 栈的应用

### 表达式求值

A B + C * E F - -

1 2 + 3 * 4 5 - -



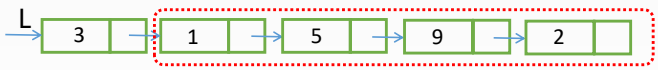

**算法思想：**
顺序扫描后缀序列
- 若是数字压入栈中；
- 若是操作符，连续弹出栈中两个元素，按操作符计算并把结果压入栈中。
- 扫描完毕后栈顶为最后结果。

( ( 1 + 2 ) * 3 ) - ( 4 - 5 ) = 10

## 栈的应用

**递归** 若在一个函数、过程或数据结构的定义中又应用了它自身，则称它为递归定义的，简称递归

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode, *LinkList;
```

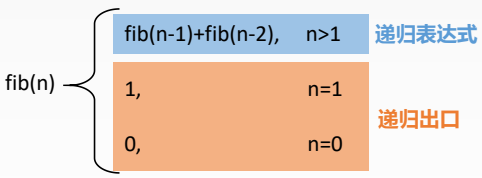





## 栈的应用

**递归** 若在一个函数、过程或数据结构的定义中又应用了它自身，则称它为递归定义的，简称递归

**斐波那契数列**：0，1，1，2，3，5，……




```
int Fib(int n){
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return Fib(n-1) + Fib(n-2);
}
```

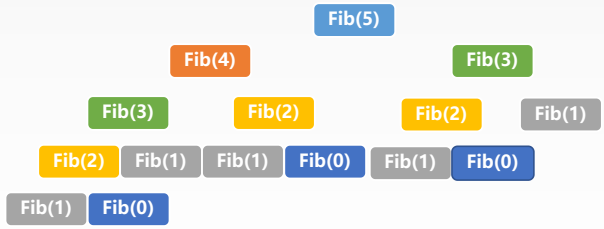
★ 递归的精髓在于能否将原始问题转换为属性相同但规模较小的问题

## 栈的应用

**递归产生的问题**

- 在递归调用过程中，系统为每一层的返回点、局部变量、传入实参等开辟了递归工作栈来进行数据存储，递归次数过多容易造成栈溢出。
- 通常情况下递归的效率并不高

```
int Fib(int n){
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return Fib(n-1) + Fib(n-2);
}
```




递归转换算法转换为非递归算法，往往需要借助栈来进行



本节内容

栈和队列

特殊矩阵
数组

## 数组

? 什么是数组

逻辑结构

**数组** 是由n (n>=1)个相同类型的数据元素构成的有限序列，每个数据元素称为一个数组元素，每个元素受n个线性关系的约束，每个元素在n个线性关系中的序号称为该元素的下标，并称该数组为n维数组。

**数组是线性表的推广！**



## 数组

**数组的维度**

**一维数组**


**二维数组**

## 数组

**数组的维度和维界不可变**


**数组一旦被定义，其维度和维界不可变，数组除初始化和销毁外，只有存取元素和修改元素的操作**



## 数组

**存储结构**

$(a_0, a_1, a_2, \cdots, a_{n-1}, a_n)$

**一维数组**

| 0 | 1 | 2 | | n-1 | n |
|---|---|---|---|---|---|
| $a_0$ | $a_1$ | $a_2$ | ..... | $a_{n-1}$ | $a_n$ |

**存储空间**

| |
|---|
| $a_0$ |
| $a_1$ |
| $a_2$ |
| ... |
| $a_{n-1}$ |
| $a_n$ |

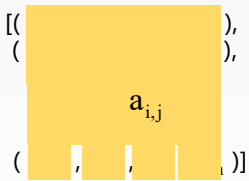
$LOC(a_0)$

$LOC(a_i) = LOC(a_0) + (i)*L \quad (0 \le i \le n)$

# 数组

## 存储结构

### 按行优先




$[(a_{0,0}, a_{0,1}, a_{0,2}),$
$(a_{1,0}, a_{1,1}, a_{1,2})]$

存储空间

| |
|---|
| $a_{0,0}$ |
| $a_{0,1}$ |
| $a_{0,2}$ |
| $a_{1,0}$ |
| $a_{1,1}$ |
| $a_{1,2}$ |

$LOC(a_{0,0})$

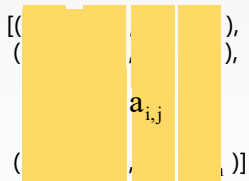
$LOC(a_{i,j}) = LOC(a_{0,0}) + i * (n+1)*L + j * L$



# 数组

## 存储结构

### 按列优先




$[(a_{0,0}, a_{0,1}, a_{0,2}),$
$(a_{1,0}, a_{1,1}, a_{1,2})]$

存储空间

| |
|---|
| $a_{0,0}$ |
| $a_{1,0}$ |
| $a_{0,1}$ |
| $a_{1,1}$ |
| $a_{0,2}$ |
| $a_{1,2}$ |

$LOC(a_{0,0})$

$LOC(a_{i,j}) = LOC(a_{0,0}) + j * (n+1)*L + i * L$

本节内容

# 栈和队列

## 特殊矩阵
压缩存储



## 特殊矩阵的压缩存储

? **矩阵**

## 特殊矩阵的压缩存储

? 矩阵的压缩存储



## 特殊矩阵的压缩存储

? 矩阵的压缩存储

王道考研/CSKAOYAN.COM

## 特殊矩阵的压缩存储

**？ 矩阵的压缩存储**

压缩存储：指多个值相同的元素只分配一个存储空间，对零元素不分配存储空间。

特殊矩阵：指具有许多相同矩阵元素或零元素，并且这些相同矩阵元素或零元素的分布有一定规律性的矩阵。

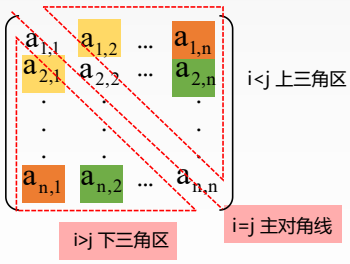
**Just Kidding**

特殊矩阵的压缩存储：找出特殊矩阵中值相同的矩阵元素的分布规律，把那些呈现规律性分布、值相同的多个矩阵元素压缩存储到一个存储空间上。



## 特殊矩阵的压缩存储

**对称矩阵** 若对一个n阶方阵A[1…n][1…n]中的任意元素 $a_{i,j}$ 都有 $a_{i,j} = a_{j,i}(1 \le i,j \le n)$，则称其为对称矩阵。

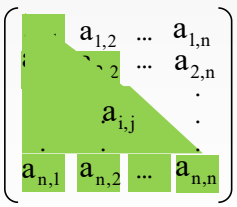



存放数组B[n(n+1)/2]

## 特殊矩阵的压缩存储

**对称矩阵** 若对一个n阶方阵A[1…n][1…n]中的任意元素 $a_{i,j}$都有 $a_{i,j}= a_{j,i}(1\le i,j\le n)$，则称其为对称矩阵。

**按行优先**




数组下标 k = 1+2+…+(i-1)+j-1+1-1

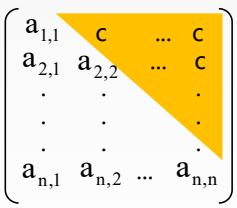
$$\text{数组下标 } k = \begin{cases} i(i-1)/2 + j-1 & i\ge j \\ j(j-1)/2 + i-1 & i<j \end{cases}$$



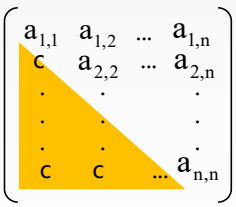
## 特殊矩阵的压缩存储

**三角矩阵** 若对一个n阶方阵A[1…n][1…n]中上（下）三角区元素均为同一常量，则称为下（上）三角矩阵。

**下三角矩阵**




**上三角矩阵**

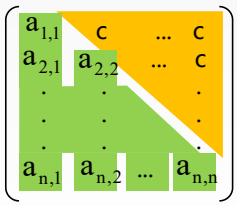



存放数组 B[n(n+1)/2+1]

## 特殊矩阵的压缩存储

**三角矩阵** 若对一个n阶方阵A[1...n][1...n]中上（下）三角区元素均为同一常量，则称为下（上）三角矩阵。

**下三角矩阵**




数组下标 k = { i(i-1)/2 + j-1　i≥j ; n(n+1)/2　i<j }

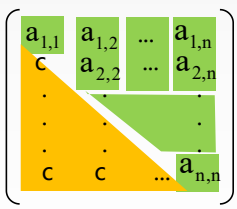
| | 0 | 1 | 2 | | n(n+1)/2 |
|---|---|---|---|---|---|
| B | $a_{1,1}$ | $a_{2,1}$ | $a_{2,2}$ | ..... | $a_{n,n}$ c |



## 特殊矩阵的压缩存储

**三角矩阵** 若对一个n阶方阵A[1...n][1...n]中上（下）三角区元素均为同一常量，则称为下（上）三角矩阵。

**上三角矩阵**



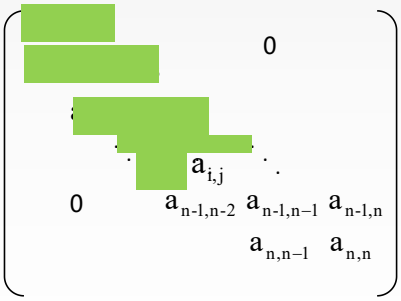

数组下标 k = { (i-1)(2n-i+2)/2+(j-i)　i≤j ; n(n+1)/2　i>j }

| | 0 | 1 | 2 | | n(n+1)/2 |
|---|---|---|---|---|---|
| B | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ | ..... | $a_{n,n}$ c |

王道考研/CSKAOYAN.COM

## 特殊矩阵的压缩存储

**三对角矩阵** 若对一个n阶方阵A中的任意元素 $a_{i,j}$，当|i - j|>1,有 $a_{i,j} = 0$（1≤i,j≤n），则称为三对角矩阵。




数组下标 k = 3*(i-1)-1+j-i+1 +1-1

= 2i+j-3

若k已知，求i，j?

$i = \lfloor (k+1)/3+1 \rfloor$

j = k-2i+3



## 特殊矩阵的压缩存储

**稀疏矩阵** 矩阵元素个数s相对于矩阵中非零元素的个数t来说非常多，即s>>t的矩阵称为稀疏矩阵。

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{pmatrix}$$

**三元组（行标，列标，值）**

| i | j | value |
|---|---|---|
| 0 | 0 | 1 |
| 2 | 3 | 8 |
| 3 | 2 | 2 |

稀疏矩阵压缩存储后失去了随机存储的特性

# 本章总览