第三章

栈和队列

Stack and Queue

主讲：顾为兵

栈和队列都是特殊的线性表

线性表：表中数据元素之间存在着线性关系

特殊性：对表的插入和删除操作受到了限制

第3章 栈和队列

目录

§ 3.1 栈(Stack)

目录

3.1.1 栈的定义

普通线性表:

插入位置有n+1个(1..n+1)

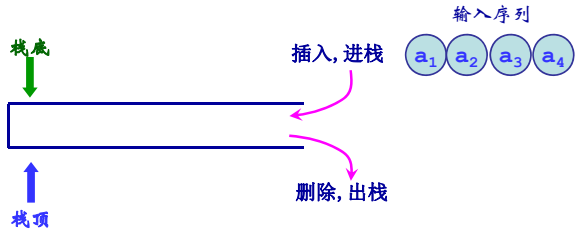
删除位置有n个(1..n)

栈:

插入位置和删除位置均只有一个，

限制在表的同一端(表尾端)进行。

栈模型：



特点：后进先出LIFO(Last In First Out)

栈 ● 栈的定义

举例：A, B, C, D依次进栈，出栈序列是：

D,C,B,A	A,B,C,D	B,C,D,A	C,D,B,A
~~D,C,A,B~~	A,B,D,C	B,C,A,D	~~C,D,A,B~~
~~D,B,C,A~~	A,C,B,D	B,D,C,A	C,B,D,A
~~D,B,A,C~~	A,C,D,B	~~B,D,A,C~~	C,B,A,D
~~D,B,C,A~~	A,D,C,B	B,A,C,D	~~C,A,B,D~~
~~D,B,A,C~~	~~A,D,B,C~~	B,A,D,C	~~C,A,D,B~~

栈 ● 栈的定义

栈的抽象数据类型定义：

```
ADT Stack{
   数据对象：D={ ai | ai∈ElemSet, i=1,2,...,n,n≥0 }
   数据关系：R={ <ai-1,ai> | ai-1,ai∈D, i=2,...,n }
          约定an端为栈顶，a1端为栈底。
   基本操作：
      InitStack (&S)          (初始化)
        操作结果：构造一个空栈S
      DestroyStack(&S)        (销毁栈)
        初始条件：栈S已经存在；  操作结果：栈S被销毁
      ClearStack(&S)          (清空栈)
        初始条件：栈S已经存在；
        操作结果：清空栈S清为空栈
```

（未完，接下页）

栈 ● 栈的定义

```
      StackEmpty(S)           (判栈空)
        初始条件：栈S已经存在。
        操作结果：若栈S为空，则返TRUE，否则返回FALSE。
      StackLength(S)          (求栈长)
        初始条件：栈S已经存在。操作结果：返回栈元素的个数。
      GetTop(S, &e)           (读栈顶)
        初始条件：栈S已存在且非空。操作结果：用e返回栈顶的值
      Push(S, e)              (进栈)
        初始条件：栈S已存在。操作结果：插入e为新的栈顶元素。
      Pop(S, &e)              (出栈)
        初始条件：栈S已存在且非空。
        操作结果：删除栈顶元素，并用e返回其值。
}ADT Stack
```

§3.1 栈

§3.1 栈(Stack)

目录

栈 ● 栈的表示与实现

3.2.2 栈的表示与实现

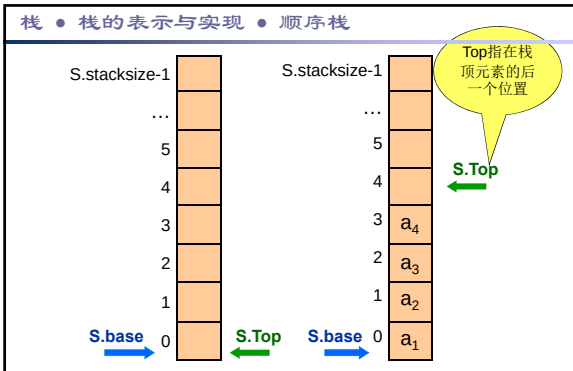
一、顺序存储结构----顺序栈

二、链式存储结构----链栈

栈 ● 栈的表示与实现 ● 顺序栈

一、顺序栈

```
typedef struct {
   SElemType      *base;
   SElemType      *top;
   int            stacksize;
}SqStack
```



判空栈：S.base == S.Top　　求栈长：S.Top – S.base

基本运算在顺序栈上的实现

读栈顶：

```
Status GetTop(SqStack S, SElemType &e){
      //若栈不空，则用e返回S的栈顶元素，并返OK，
      //否则返回ERROR
      if(S.top==S.base) return ERROR; //栈空
      e=*(S.top-1);
      return OK;
}//GetTop
```


基本运算在顺序栈上的实现

判栈空：

```
Status StackEmpty(SqStack S){
      //若栈空，返回TRUE；否则返回FALSE。
      if(S.top==S.base) return TRUE;  //栈空
      else return FALSE;
}//StackEmpty
```


基本运算在顺序栈上的实现

进栈操作：

```
Status Push(SqStack &S, SElemType e){
   //插入元素e为新的栈顶元素
   if(S.top-S.base>=S.stacksize) {//栈满，追加空间
      S.base=(SElemType*)realloc(S.base,
        (S.stacksize+STACKINCREMENT)*sizeof(SElemType));
      if(!S.base) exit(OVERFLOW);
      S.stacksize+=STACKINCREMENT;
    }
   *S.top++=e;
   return OK;
}//Push
```

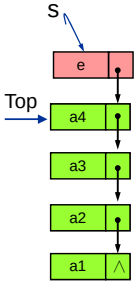

基本运算在顺序栈上的实现

出栈：

```
status Pop(SqStack &S, SElemType &e){
   //若栈不空，则删除栈顶元素，用e返回其值，并返回OK；
  //否则返回ERROR
   if(S.top==S.base) return ERROR;  //栈空
   e=*(--S.top);
   return OK;
}//Pop
```


二、链栈

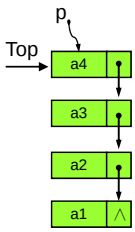
----用头指针TOP引导的不带头结点的单链表



新结点s进栈，

Push：

```
s->next=top;
top=S;
```



出栈Pop：

```
p=top;
top=top->next;
free(p);
```

判栈空条件：

```
Top==NULL;
```


基本运算在链栈上的实现

出栈：

```
status Push(LinkStack &S, SElemType e){
  //插入元素e为新的栈顶元素，S是链栈的头指针
  p=(SElemType*)malloc(sizeof(SElemType));
  if(!p)exit(OVERFLOW);
  p->data=e;
  p->next=S; S=p; //头插栈顶元素
  return OK;
}//Pusp
```


基本运算在链栈上的实现

出栈：

```
status Pop(LinkStack &S, SElemType &e){
  //若栈不空，则删除栈顶元素，用e返回其值，并返回OK；
 //否则返回ERROR
  if(S==NULL) return ERROR;  //栈空
  e=S->data;
  p=S; S=S->next; free(p); //删除栈顶元素
  return OK;
}//Pop
```

第3章 栈和队列

目录

§ 3.2 栈的应用举例

目录

3.2.1 数制转换

原理： $N = (N \text{ div } d) \times d + N \bmod d$

例： $(1348)_{10} = (2504)_8$

```
8 | 1348
8 | 168   4
8 | 21    0   结
8 | 2     5   果
  0       2
```

借助一个栈来反转计算结果的顺序

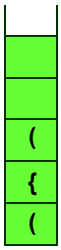
```
void  conversion( ){
   //输入非负十进制整数，打印对应的八进制数
      InitStack(S);
      scanf ("%d", &N);
      while(N){
         Push (S, N%8);   //余数进栈
         N=N\8;           //用8整除N
      }
      while(!StackEmpty(S)){
         Pop(S, e); printf("%d", e);
      }
} //conversion
```


§3.2 栈的应用举例

目录

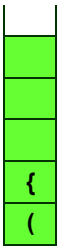
例：检验括号匹配

(…{…[…] …)…}…



例：检验括号匹配

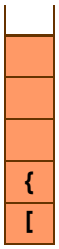
(…{…[…] …)…}…



右括号“)”与当前栈顶的左括号“}”错配

例：检验括号匹配

(…{…}…) …[…{…(…)…[…]



扫描结束了而栈非空，不匹配

括号配对的三种失配情况：

1. 当扫描到右括号时栈为空----无左括号相配；
2. 栈不空，但右括号与当前的栈顶元素不配对；
3. 字符串扫描结束，但栈不空----有多余的左括号未配完。

```
Status Check(char *exp){ //exp是待检查的字符串
    char *p=exp;
    InitStack(S);   //初始化栈S
    while((ch=*p)!='\0'){   //当exp未扫描完
       if(is_left(ch)) push(S,ch); //左括号进栈
    else if(is_right(ch)){
          if(!Pop(S,left)) return -1;//缺左括号
          if(!match(left,ch)) return -2;//左右不配
       }//else
       p++;    //继续扫描下一个字符
    }
   if(! StackEmpty(S))return -3; //有多余的左括号
    else return TRUE;
} //matching
```



```
Status is_left(char ch){
    //判断字符ch是否是左括号
    if(ch=='(' || ch=='[' || ch=='{')
       return TRUE;
    else  return FALSE;
}//is_left

Status is_right(char ch){
    //判断字符ch是否是右括号
    if(ch==')' || ch==']' || ch=='}')
       return TRUE;
    else  return FALSE;
}//is_right
```



```
Status match(char a, char b){
    //判断左括号a和右括号b是否配对
    if(a=='(' && b==')') return TRUE;
    if(a=='[' && b==']') return TRUE;
    if(a=='{' && b=='}') return TRUE;
    return FALSE;
}//match
```


§3.2 栈的应用举例

目录

3.2.3 行编辑程序

编辑一篇文档时，除了输入字母、数字、标点符号等可打印字符外，还会输入一些控制字符，如：

退格符（Backspace）： #
退行符： @
换行符（Enter）： \n
全文结束符： EOF

行编辑程序的算法思想：

从键盘输入字符，使用一个栈来保留文档中一行文字的可打印字符，并且针对不同的控制字符采用不同的栈操作来达到修改的目的：

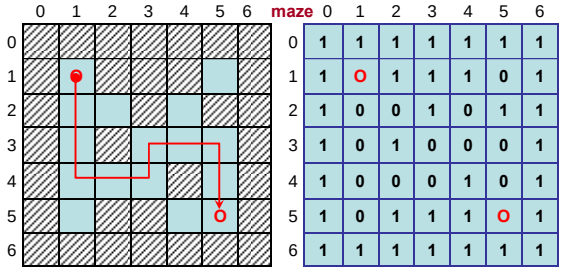
- 可打印字符： Push进栈
- 退格符＃： Pop出栈
- 退行符@： ClearStack清空栈
- 换行符\n： 将栈内的字符录入文档，清空栈，接收下一行字符
- 全文结束符EOF： 全文结束，销毁栈

§3.2 栈的应用举例

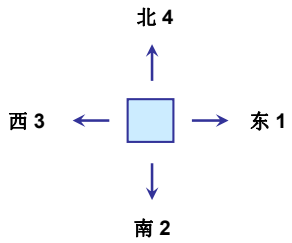
目录

3.2.4 迷宫求解

二维数组maze表示迷宫：
0----空地，2----路径
1----墙， 3----死路



每个通道块有四个探测方向：



通道块类型：

```
typedef  struct {
   int        i ;   //行下标
   int        j ;   //列下标
} PostType ;
```

栈元素类型：

```
typedef  struct {
   int       ord;  //通道块在路径上的序号
   PostType  seat; //通道块的下标
   int       di ;  //通道块当前的前进方向
} ElemType ;
```


算法3.3中的**Pass( )**函数：

```
Status Pass(PosType curpos) {
  //判断maze数组中块curpos是否可通过;
  if(maze[curpos.i][curpos.j]==0)
          return TRUE;
  return FALSE;
}
```


算法3.3中的**FootPrint( )**函数：

```
void  FootPrint(PosType curpos){
   //给maze数组中块curpos打上路径标记2
   maze[curpos.i][curpos.j]=2;
}
```

算法3.3中的**MarkPrint( )**函数：

```
void MarkPrint (PosType curpos) {
   //给maze数组中块curpos打上死路标记3
   maze[curpos.i][curpos.j] =3;
}
```

算法3.3中的**NextPos( )**函数：

```
PosType NextPos(PosType curpos, int di){
  //返回块curpos在di方向的邻接块的位置
  PosType   s=curpos;
  switch(di){
    case 1: s.j++; break;     //向东
    case 2: s.i++; break;     //向南
    case 3: s.j--; break;     //向西
    case 4: s.i--; break;     //向北
  } //end switch
  return s;
}  //end NextPos
```

	4	
3	i, j	1
	2	

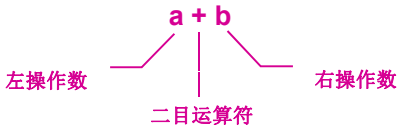
```
Status  MazePath( MazeType maze, PostType  start, PostType  end ){
  InitStack(S);  curpos=start;  curstep=1;     //初始化
  do{
    if(Pass(curpos)){           //如果当前块可通
      FootPrint(curpos);        //打下路径标记
      e=(curstep, curpos, 1 ); Push(S, e); //当前块进栈
      if(curpos==end) return TURE;  //当前块是终点，结束
      curpos=NextPos(curpos,1); curstep++;} //向东进入下一块
    else{                       //如果当前块不可通
      if(!StackEmpty(S)){       //未回溯完
        Pop(S, e);              //弹出栈顶
        while(e.di==4&&!StackEmpty(S)){   // 4个方向探完
          MarkPrint(e.seat); Pop(S, e);}
          if(e.di<4){                     //还有未探测的方向
           e.di++; Push(S, e);            //换方向
           curpos=NextPos(e.seat, e.di);
        } //end if
    } //end else
  }while(!StackEmpty);
   return False;   //迷宫不通，失败
}  //end MazePath
```


§3.2 栈的应用举例

目录

3.2.5 表达式求值

算术表达式：**(a+b)*(c-d)-d/e**



运算规则：先算乘除，后算加减；

先算括号里，后算括号外；

从左算到右。

算术表达式的另一种表示方法：

将运算符放在操作数的前面，称为前缀表达式，或波兰式；

+ a b

运算符放在操作数的后面，称为后缀表达式，或逆波兰式；

a b +

一般表达式：**(a+b)*(c-d)-e/f**

等价的后缀表达式：**ab+cd-*ef/-**

等价的前缀表达式：**-*+ab-cd/ef**

后缀表达式和前缀表达式的共同特点是：

1. 表达式中没有括号，
2. 运算符不存在优先级的差别。

后缀表达式的计算方法：

- 从左向右扫描表达式；
- 遇到运算符时，就将该算符左侧的两个操作数进行计算。

```
  5 2 + 9 4 - * 18 3 / -
= 7 9 4 - * 18 3 / -
= 7 5 * 18 3 / -
= 35 18 3 / -
= 35 6 -
= 29
```

(5+2)*(9-4)-18/3

求解逆波兰式的算法思想：

1. 初始化一个空栈S，用来存放操作数；
2. 输入θ；
3. if (θ==结束标志) { 打印栈顶(计算结果); 结束; }
4. if (θ是操作数)
   Push(S, θ);
   else {
   Pop(S, b); Pop(S, a);
   Push(S, a θb);
   }
5. 转 2。

前缀表达式的求值方法：

– * + 5 2 – 9 4 / 18 3

前缀表达式求值算法也要借助一个栈来实现。

注意，这是一个操作数和运算符“混装” 的栈。

求解波兰式的算法思想：

1. 初始化一个空栈S，用来存放操作数或运算符
2. 输入 y;
3. 如果 y 是结束标志，打印栈顶，算法结束。
4. if (y 是运算符)
   Push(S, y);
   else {
   while(当前栈顶是操作数) {
   Pop(S, x); Pop(S, θ); y = x θ y ;
   }
   Push(S, y);
   }
5. 转 2。

中缀表达式求值：

(a + b) * (c – d) – e / f

由于要处理括号和算符优先级，所以算法比较复杂，需要借助两个栈：

操作数栈：OPND

运算符栈：OPTR

 # … a θ_1 b θ_2 c … #

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	Φ
)	>	>	>	>	Φ	>	>
#	<	<	<	<	<	Φ	=

算法原理： # … a β b θ c … #

0. 初始化操作数栈OPND和运算符栈OPTR，将左＃压进栈OPTR
1. 从左至右逐个读取表达式的每一项θ
2. 如果θ是右＃且OPTR的栈顶是左＃，结束。
3. θ是操作数：Push（OPND，θ）；goto 1;
4. θ是运算符：

   β = GetTop (OPTR)，

   比较β与θ 的优先级：

   β<θ: Push (OPTR, θ); goto 1;

   β==θ: Pop (OPTR, θ)（脱括号）； goto 1;

   β>θ: Pop (OPND, b); Pop (OPND, a)；

   Pop (OPTR, β); Push (OPND，a β b)；

   goto 4;

第3章 栈和队列

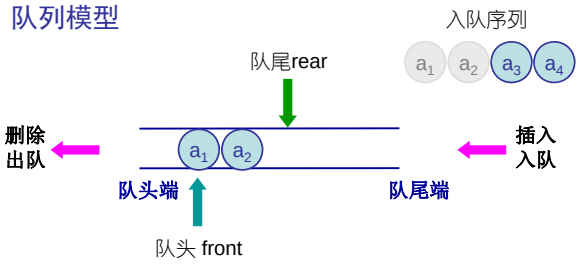
目 录

§ 3.4 队列(Queue)

3.3.1 队列的定义

- 队列是一种特殊的线性表；
- 插入位置只有1个，限制在表尾进行；
- 删除位置也只有1个，限制在表头进行。

队列模型



出队序列

特点：先进先出FIFO(First In First Out)

队列的抽象数据类型定义：

```
ADT Queue{
    数据对象：D={ ai | ai∈ElemSet, i=1,2,...,n,n≥0 }
    数据关系：R={ <ai-1,ai> | ai-1,ai∈D, i=2,...,n }
              约定a1端为队列头，an端为队列尾。
    基本操作：
      InitQueue (&Q)          (初始化空队列)
        操作结果：构造一个空队列Q
      DestroyQueue(&Q)        (销毁队列)
        初始条件：队列Q已经存在； 操作结果：队列Q被销毁
      ClearQueue(&Q)          (清空队列)
        初始条件：队列Q已经存在；
        操作结果：将Q清空为空队列
```

（未完，接下页）

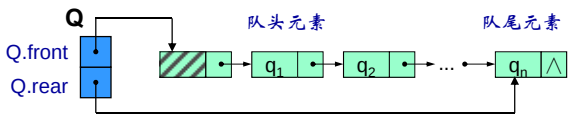
QueueEmpty(Q)　　（判队列空）
初始条件：队列Q已经存在。
操作结果：若队列Q为空，则返TRUE，否则返回FALSE。
QueueLength(Q)　　（求队列长）
初始条件：队列Q已经存在。操作结果：返回队元素的个数。
GetHead(Q, &e)　　（读队头）
初始条件：Q为非空队列。操作结果：用e返回队头的值
EnQueue(Q, e)　　（入队）
初始条件：Q已存在。操作结果：插入e为新的队尾元素。
DeQueue(Q, &e)　　（出队）
初始条件：Q为非空队列。
操作结果：删除队头元素，并用e返回其值。
}ADT Queue

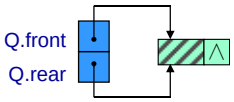
§ 3.4 队列(Queue)

链队列：　带头结点、头指针和尾指针的单链表：

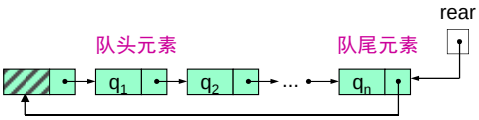
```
typedef struct  QNode {
      QElemType data;
      struct QNode *next;
} QNode, *QueuePtr;

typedef struct {
      QueuePtr    front, rear;      //队头指针和队尾指针
} LinkQueue;
```





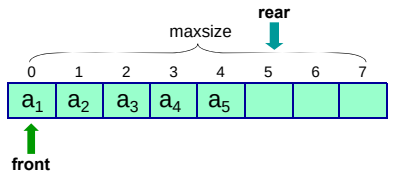
队空条件：
Q.front==Q.rear



这种用尾指针引导的循环链队列也是一种不错的选择

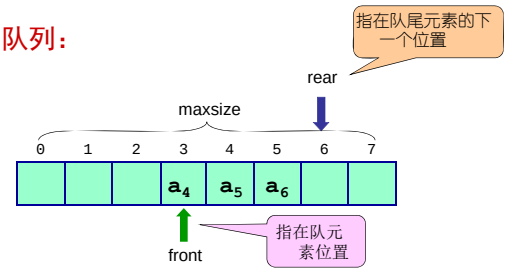
§ 3.4 队列(Queue)

队列 ● 队列的顺序表示和实现----循环队列




```
#define maxsize  100    //队列数组的大小
typedef struct {
   QElemType   *base;   //队列数组的基地址
   int         front;   //队头指示器
   int         rear;    //队尾指示器
}SqQueue;
```

顺序队列：



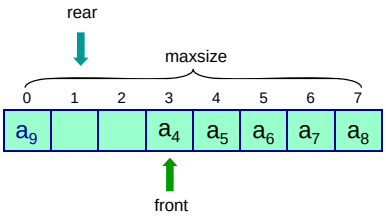
队空条件：front==rear　　队列长度：rear－front

入队操作：rear＋＋　　出队操作：front＋＋

队满条件：**真上溢** rear==maxsize && rear－front==maxsize

假上溢 rear==maxsize && rear － front < maxsize

队列 ● 队列的顺序表示和实现----循环队列

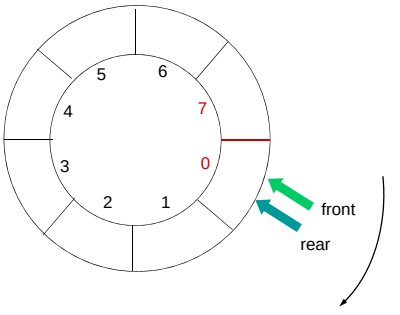


假上溢问题的解决方案：

允许rear<front，即当rear达到上界而有入队请求时，rear回到下界位置。这就是循环队列的思想。

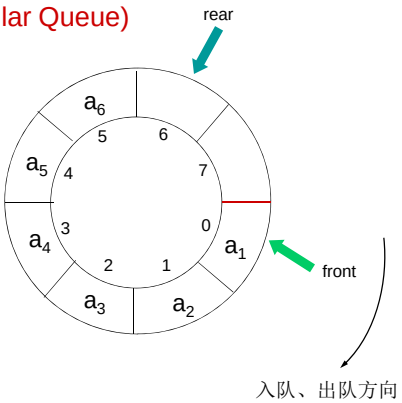
循环队列(Circular Queue)

空队列



循环队列(Circular Queue)

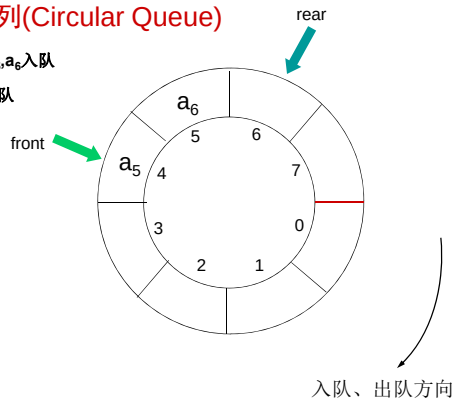
a_1,a_2,a_3,a_4,a_5,a_6入队



循环队列(Circular Queue)

a_1,a_2,a_3,a_4,a_5,a_6入队

a_1,a_2,a_3,a_4出队

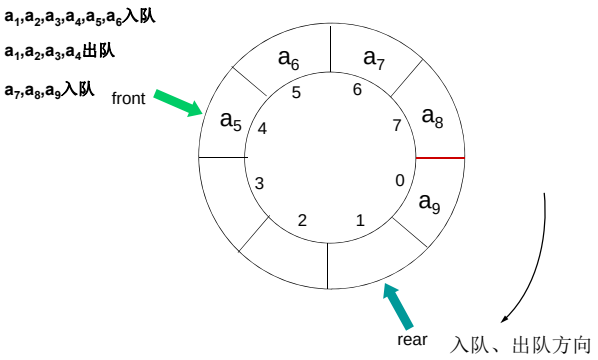


循环队列(Circular Queue)

a_1,a_2,a_3,a_4,a_5,a_6入队

a_1,a_2,a_3,a_4出队

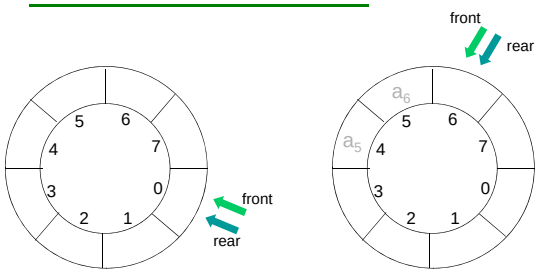
a_7,a_8,a_9入队



入队：
```
if(rear<maxsize-1) rear ++;
else  rear=0;
```
这个if语句可等价地写成：
```
rear=(rear+1) mod maxsize
```

出队：
```
if(front<maxsize-1) front++;
else  front=0;
```
这个if语句可等价地写成：
```
front=(front+1) mod maxsize
```

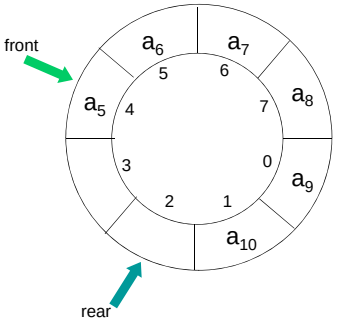

判队空的条件：front == rear



初态：front=rear=0

a5、a6出队后front=rear=6

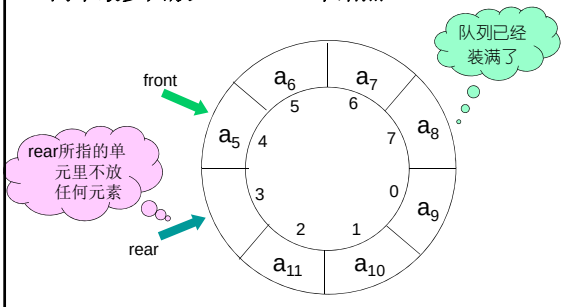
问题来了——队满的条件也是 front == rear ！



如果a_{11}和a_{12}相继入队后，rear就追上了front

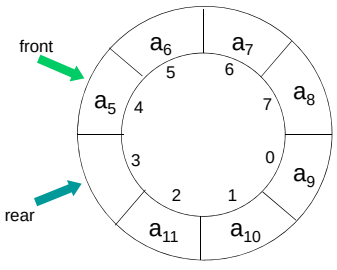
解决队满与队空条件相同的方案：

牺牲一个结点的空间——约定：maxsize的空间中最多只放maxsize−1个结点

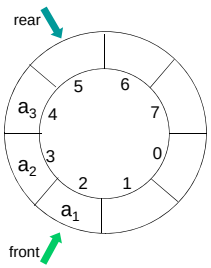


在这种约定下，当rear在顺时针方向距front相差一个单元时，队列就满了。

队满条件：front == (rear+1) mod maxsize

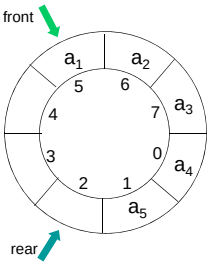


队列长度： (rear – front + maxsize) mod maxsize



front=2, rear=5

length = (5 – 2 + 8) mod 8 = 3



front=5, rear=2

length = (2 – 5 + 8) mod 8 = 5

总结：

队列Q[0..maxsize-1]

初态： rear = front = 0

入队： rear = (rear+1) mod maxsize

出队： front = (front +1) mod maxsize

队空条件： front == rear

队满条件： front == (rear+1) mod maxsize

队列长度： (rear-front+maxsize) mod maxsize

注意，当队列数组的下界为1时计算公式应为：

队列Q[1..maxsize]

初态： rear = front = 1

入队： rear = rear mod maxsize + 1

出队： front = front mod maxsize + 1

队空条件： front == rear

队满条件： front == rear mod maxsize + 1

队列长度： (rear-fornt+maxsize) mod maxsize

循环队列的类型定义：

```
#define MAXQSIZE  100;    //队列数组的大小

typedef struct {
  QElemType     *base;     //队列数组首地址
  int           front;     //队头指针，指向队头元素
  int           rear;      /*队尾指针，指向
                           队尾元素的下一个位置*/
 }SqQueue;
```


基本运算在循环队列上的实现：初始化队列

```
Status InitQueue(SqQueue &Q){
   //初始化一个空队列
   Q.base=(QElemType*)malloc(MAXQSIZE*
                          sizeof(QElemType));
   if(!Qbase) return(OVERFLOW);
   Q.front=Q.rear=0;
   return OK
}//InitQueue
```


求队列长度：

```
int QueueLength(SqQueue Q){
   len=(Q.rear-Q.front+MAXQSIZE) % MAXQSIZE;
   return len;
}//QueueLength
```

入队操作：

```
Status EnQueue(SqQueue &Q, int e){
  //入队操作,元素e成为新的队尾元素
   if(Q.front==(Q.rear+1)%MAXQSIZE)
       return ERROR;          //队列已满
   Q.elem[Q.rear]=e;
   Q.rear=(Q.rear+1) % MAXQSIZE; //修改队尾指针
}//EnQueue
```


出入队操作：

```
Status DeQueue(SqQueue &Q, int &e){
  //用e返回队头元素的值，队头元素出队，
   if(Q.front==Q.rear) return ERROR; //队空
   e=Q.elem[Q.front];
   Q.front=(Q.front+1)% MAXQSIZE; //修改队头指针
   return OK;
}//DeQueue
```


判队空：

```
bool QueueEmpty(SqQueue Q){
   return Q.front==Q.rear ? TRUE : FALSE;
}//QueueLength
```