第三章　栈和队列

对线性表 $L=(a_1, a_2, , \ldots, a_n)$，

可在任意第i(i=1, 2, , . . . n, n+1)个位置插入新元素，

或删除任意第i(i=1, 2, , . . . n)个元素。

受限数据结构---- 插入和删除受限制的线性表：

1. 栈(stack)
2. 队列(queue)
3. 双队列(deque)

栈和队列是限定插入和删除只能在表的“端点”进行的线性表。

线性表	栈	队列
Insert(L, i, x) 1≤i≤n+1	Insert(S, n+1, x)	Insert(Q, n+1, x)
Delete(L, i,&e) 1≤i≤n	Delete(S, n,&e)	Delete(Q, 1,&e)

栈和队列是两种常用数据类型，是典型的串行数据结构。

3.1栈(stack)

3.1.1栈的定义和操作

1.定义和术语

栈：限定在表尾作插入、删除操作的线性表。

$(a_1, a_2, \ldots, a_n)$ ←插入元素(进栈)

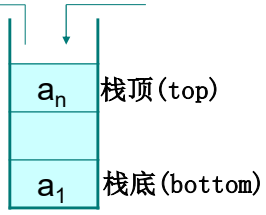
↑ ↑ ↘删除元素(出栈)

表头 表尾

(栈底) (栈顶)

3

出栈(pop) 进栈(push)



栈的示意图

进栈：插入一个元素到栈中。

或称:入栈、推入、压入、push。

出栈：从栈删除一个元素。

或称：退栈、上托、弹出、pop。

栈顶：允许插入、删除元素的一端(表尾)。

栈顶元素：处在栈顶位置的元素。

栈底：表中不允许插入、删除元素的一端。

空栈：不含元素的栈。

栈的元素的进出原则:

“后进先出”,“Last In First Out”。

栈的别名:“后进先出”表、“LIFO”表、

反转存储器、地窖、堆栈。

4

栈的类型定义

ADT Stack {

数据对象:

$D = \{ a_i \mid a_i \in ElemSet, i=1,2,...,n, n\geq 0 \}$

数据关系:

$R1 = \{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n \}$

约定a_n端为栈顶，a_1端为栈底。

基本操作:

} ADT Stack

5

2. 栈的基本操作

(1) Initstack(&s)： 置s为空栈。

(2) Push(&s,e)： 元素e进栈s。

若s已满，则发生溢出。

若不能解决溢出，重新分配空间失败，则插入失败。

(3) Pop(&s,&e)： 删除栈s的顶元素，并送入e 。

若s为空栈,发生“下溢”(underflow)；

为空栈时，表示某项任务已完成。

(4) GetTop(s,&e)： 栈s的顶元素拷贝到e。

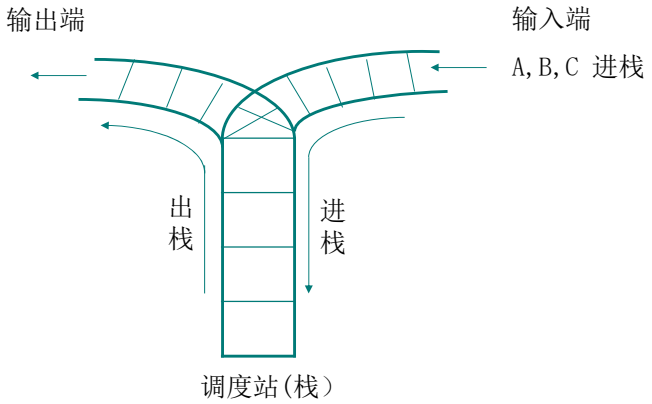
若s为空栈，则结束拷贝。

(5) StackEmpty(s)： 判断s是否为空栈。

若s为空栈,则StackEmpty(s)为true；否则为false。

6

3. 理解栈操作（模拟铁路调度站）

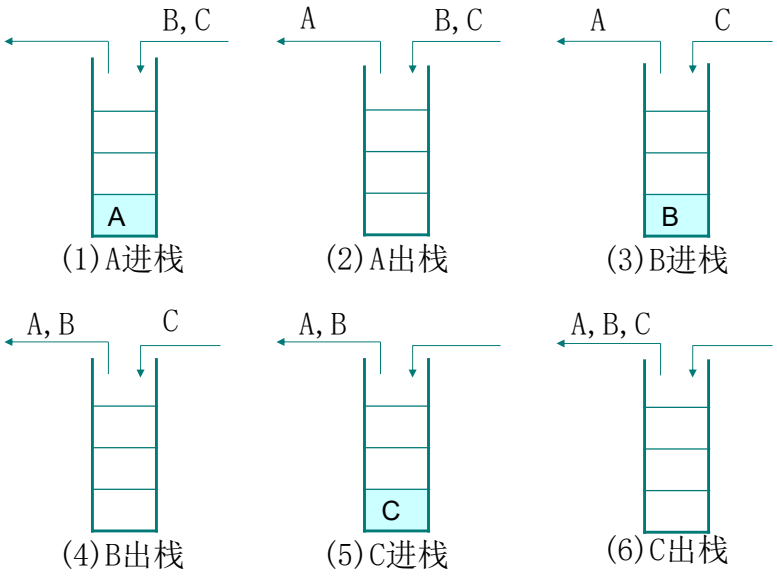


讨论：

假设依次输入3个元素(车辆)A, B, C到栈(调度站)中，可得到哪几种不同输出？

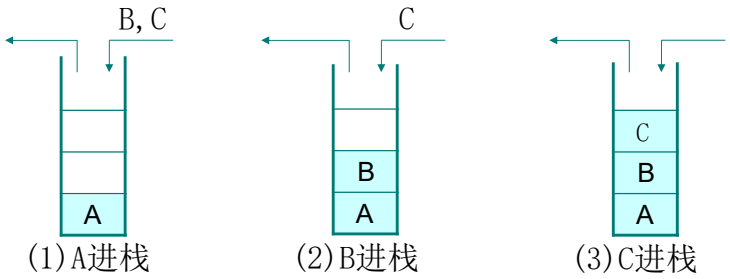
7

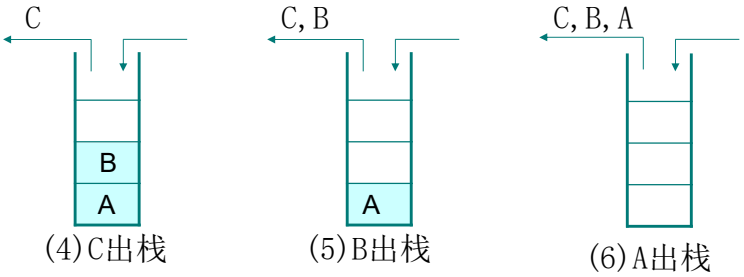
(1)输入A, B, C, 产生输出A, B, C的过程：



8

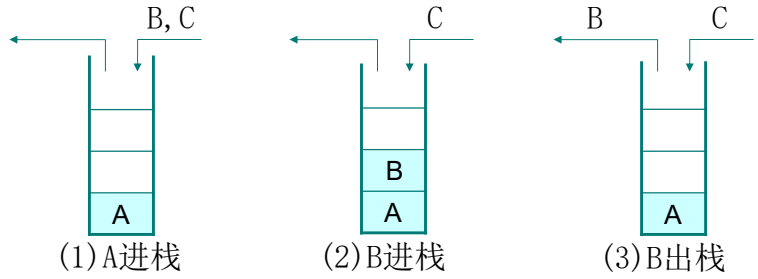
(2)输入A, B, C, 产生输出C, B, A的过程：

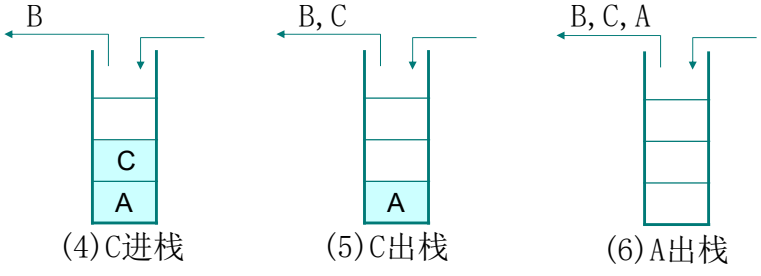




9

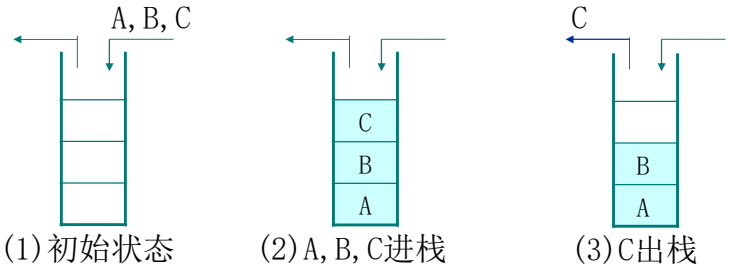
(3)输入A, B, C, 产生输出B, C, A的过程：





10

(4) 输入A, B, C, 不能产生输出C, A, B:

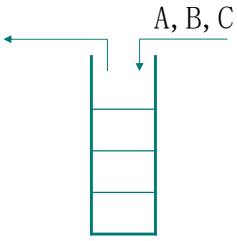


(1) 初始状态　(2) A, B, C进栈　(3) C出栈

当A, B, C依次进栈，C出栈后，由于栈顶元素是B, 栈底元素是A, 而A不能先于B出栈, 所以不能在输出序列中，使A成为C的直接后继, 即不可能由输入A, B, C产生输出C, A, B。

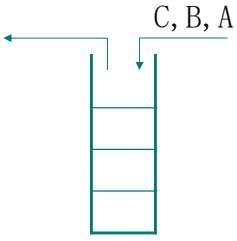
一般地，输入序列(..., a_i, ..., a_j, ..., a_k, ...)到栈中, 不能得到输出序列(..., a_k, ..., a_i, ..., a_j, ...)。

设依次输入元素A, B, C到栈中, 可得哪几种输出?



(1) A, B, C
(2) A, C, B
(3) B, A, C
(4) B, C, A
~~(5) C, A, B~~
(6) C, B, A

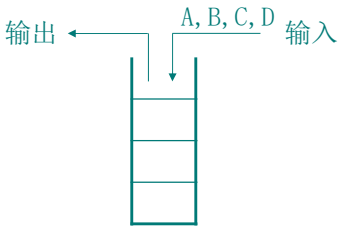
设依次输入元素C, B, A到栈中, 可得哪几种输出?



(1) A, B, C
~~(2) A, C, B~~
(3) B, A, C
(4) B, C, A
(5) C, A, B
(6) C, B, A

讨论：

假定输入元素 A,B,C,D 到栈中，能得到哪几种输出？不能得到哪几种输出序列？



(1) A,B,C,D	(7) B,A,C,D	~~(13) C,A,B,D~~	~~(19) D,B,C,A~~
(2) A,B,D,C	(8) B,A,D,C	~~(14) C,A,D,B~~	~~(20) D,B,A,C~~
(3) A,C,B,D	(9) B,C,A,D	(15) C,B,A,D	(21) D,C,B,A
(4) A,C,D,B	(10) B,C,D,A	(16) C,B,D,A	~~(22) D,C,A,B~~
~~(5) A,D,B,C~~	~~(11) B,D,A,C~~	~~(17) C,D,A,B~~	~~(23) D,A,B,C~~
(6) A,D,C,B	(12) B,D,C,A	(18) C,D,B,A	~~(24) D,A,C,B~~
5种	5种	3种	1种

共5+5+3+1=14种

13

3.1.2 栈的存储表示和操作实现

1.顺序栈： 用顺序空间表示的栈。

设计实现方案时需要考虑的因素：

- 如何分配存储空间

  动态分配或静态分配

  栈空间范围，如：s[0..maxleng-1]

- 如何设置进栈和出栈的标志top

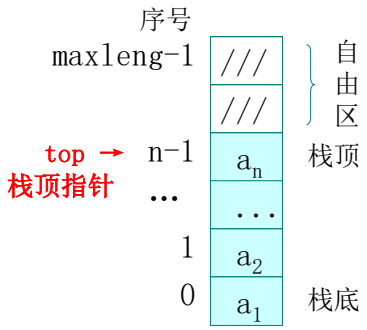
  如top指向栈顶元素或指向栈顶元素上一空单元等，作为进栈与出栈的依据。

- 分析满栈的条件，用于进栈操作。
- 分析空栈的条件，用于出栈操作。

14

(1) 方案1：栈空间范围为：s[0..maxleng-1]

顶指针指向顶元素所在位置：



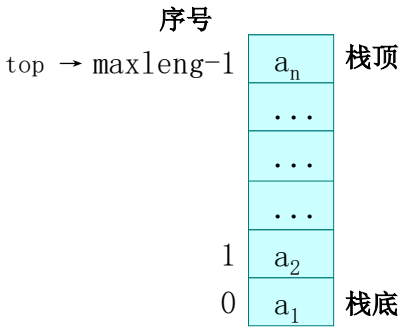
(a) 非空栈示意图

top>=0 栈顶元素=s[top]

进栈操作：先对top加1，指向下一空位置，将新数据送入top指向的位置，完成进栈操作。结束时top指向新栈顶元素所在位置。

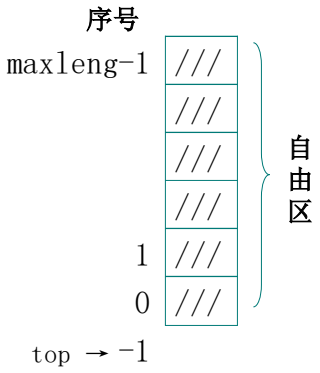
出栈操作：先根据top指向，取出栈顶数据元素；再对top减1。完成出栈操作。结束时top指向去掉原栈顶元素后的新栈顶元素所在位置。

(b) 进出栈说明



(c)满栈条件

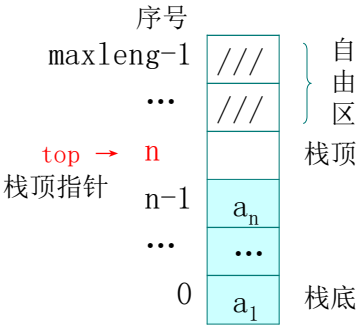
top==maxleng-1 若插入元素，将发生“溢出” “Overflow”



(d) 空栈条件，top==-1

若删除元素，将发生“下溢”

(2) 方案2：栈空间范围为：s[0..maxleng-1]
顶指针指向顶元素的下一空位置：

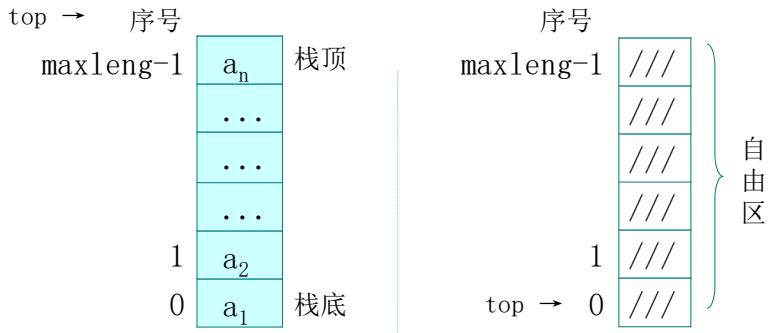


(a) 非空栈示意图
top>=1 栈顶元素=s[top-1]

进栈操作：先将新数据送入top指向的位置，再对top加1，指向下一空位置，完成进栈操作。结束时top正好指向新栈顶元素所在位置上的一空位置。

出栈操作：先对top减1,根据top指向取出栈顶数据元素。完成出栈操作。结束时top指向去掉原栈顶元素后的新栈顶元素所在位置上的一空位置。

(b) 进出栈说明



(c)满栈条件：
top==maxleng
若插入元素,将发生“溢出”

(d)空栈条件：
top==0
若删除元素，将发生“下溢”

2.顺序栈的实现

栈元素空间与顶指针合并定义为一个记录(结构)

约定：栈元素空间[0..maxleng-1]

top指向栈元素的下一空位置。

//top是栈顶标志，根据约定由top找栈顶元素。

存储空间分配方案

(a) 静态分配

```
typedef struct
 { SElemType elem[maxleng];  //栈元素空间
   int top;                  //栈顶指针
  }SqStack;                  //SqStack为结构类型
 SqStack s;                  //s为结构类型变量
```

其中：s.top---栈顶指针；s.elem[s.top-1]---栈顶元素

19

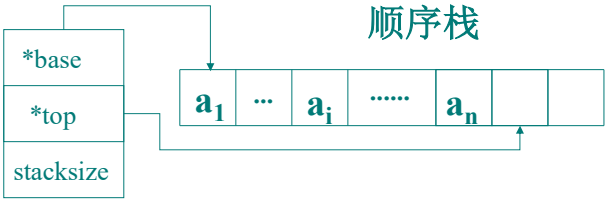
(b) 动态分配

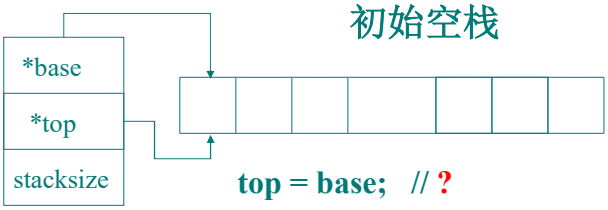
```
#define STACK_INIT_SIZE  100
#define STACKINCREMENT   10
typedef struct
  { SElemType *base;       //指向栈元素空间基址
    SElemType *top;        //栈顶指针指向栈顶元素后一单元
    int stacksize          //已分配的存储空间大小，
                           //以元素为单位
   } SqStack;              // SqStack为结构类型
 SqStack s;                //s为结构类型变量
```

其中：s.top —栈顶指针；*(s.top-1) —栈顶元素

20

顺序栈示意图





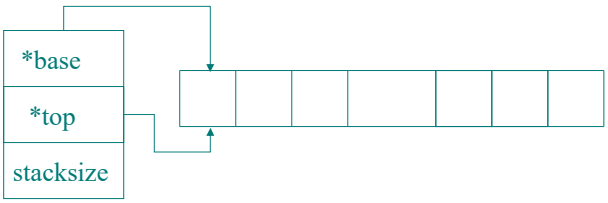
top = base; // ?

stacksize = STACK_INIT_SIZE

21

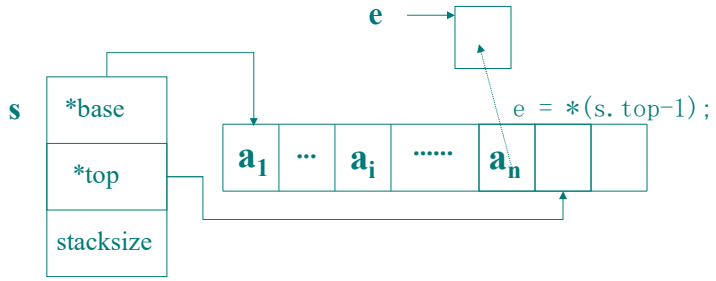
3. 顺序栈算法

```
Status InitStack (SqStack &S)
{// 构造一个空栈S
  S.base=(SElemType*)malloc(STACK_INIT_SIZE*
                            sizeof(ElemType));
  if (!S.base) exit (OVERFLOW); //存储分配失败
  S.top = S.base;
  S.stacksize = STACK_INIT_SIZE;
  return OK;
} // InitStack
```



22

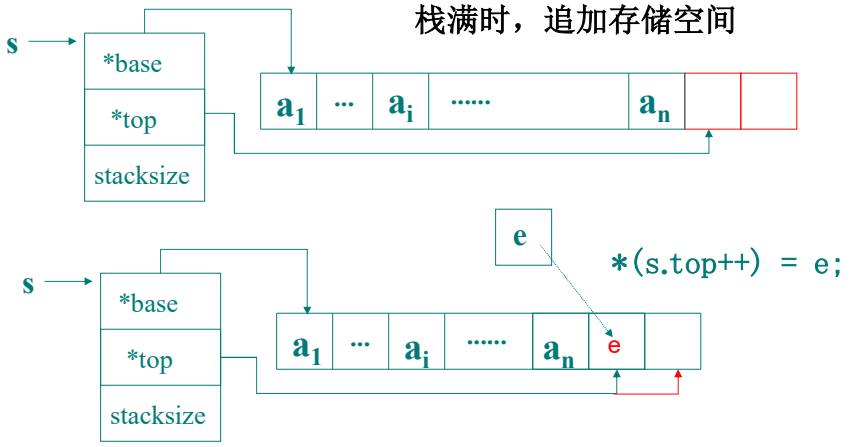
```
Status  GetTop(SqStack s, SElemType &e)  {
//栈S非空,则用e返回栈S中栈顶元素的值，并返回OK, 否则返回ERROR。
  if (s.top == s.base) return ERROR;
     e = *(s.top-1);
     return OK;
} // GetTop
```



23

(2) 进栈算法：插入元素e为新的栈顶元素

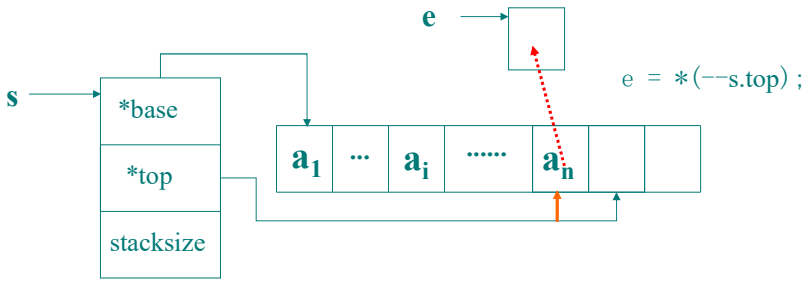
插入新的栈顶元素时，堆栈变化示意



24

```
Status Push (SqStack &S, SElemType e) {
  if (S.top - S.base >= S.stacksize) {    //栈满，追加存储空间
    SElemType *newbase;
    newbase = (SElemType *) realloc ( S.base,
          (S.stacksize + STACKINCREMENT) * sizeof (ElemType));
    if (!newbase) exit (OVERFLOW);        //存储分配失败
    S.base = newbase;
    S.top = S.base + S.stacksize;      //为何不是原来的S.top？
    S.stacksize += STACKINCREMENT;
  }
  *S.top++ = e;                        // *S.top= e; S.top=S.top+1
   return OK;
}
```


(3) 出栈算法



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



```
main()
{
 SqStack   S;
 SElemType  e;
 InitStack(S);
 push(S,10);
 if (push(S,20)==ERROR)          //最好能判断其返回值，
                                 //做出相应处理
     printf("进栈失败！");
   ……
 if (pop(S,e)==OK)
       {退栈成功，处理e的值s}
 else  {退栈失败，提示错误信息}
}
```

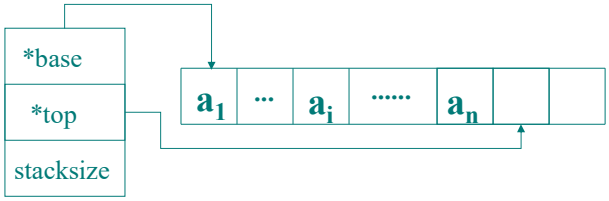
动态演示顺序栈的操作

复习：

- 什么是栈？它与一般线性表有什么不同？
  - ✓ 栈是一种特殊的线性表，它只能在表的一端（即栈顶）进行插入和删除运算。
  - ✓ 与一般线性表的区别：仅在于运算规则不同。

一般线性表	栈
逻辑结构：线性结构	逻辑结构：线性结构
存储结构：顺序表、链表	存储结构：顺序栈、链栈
运算规则：随机存取	运算规则：后进先出(LIFO)

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



29

4. 链式栈：

使用不带表头结点的单链表

(1)结点和指针的定义

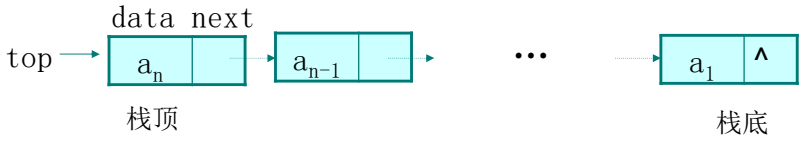
```
struct node
{ SElemType data;        //data为抽象元素类型
  struct node *next;     //next为指针类型
 } *top=NULL;            //初始化,置top为空栈
```

30

(2)非空链式栈的一般形式

假定元素进栈次序为：a_1、a_2、…、a_n。

用普通无头结点的单链表：



进栈将新结点作为首结点。

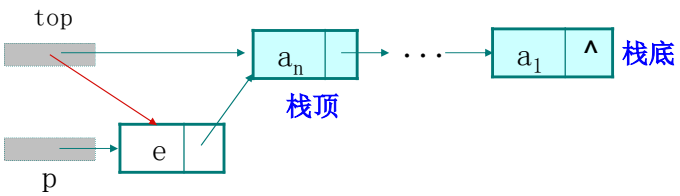
出栈时删除首结点。

优点：进出栈时间为常数。

31

(3)链式栈的进栈：

压入元素e到top为顶指针的链式栈

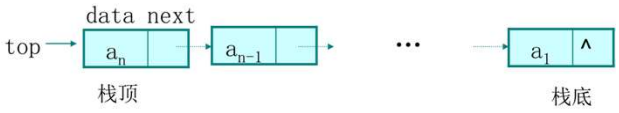



```
p=(struct node *)malloc(sizeof(struct node));
p->data=e;
p->next=top;
top=p;
```

32

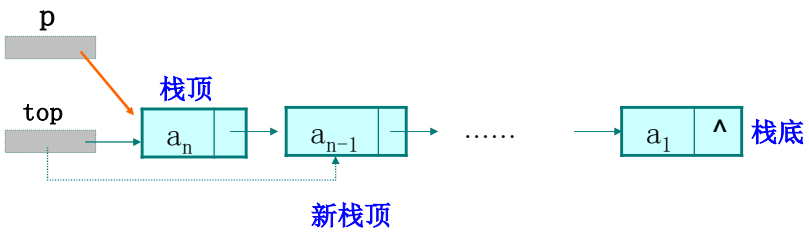
进栈算法：

```
struct node *push_link(struct node *top,SElemtype e)
{ struct node *p;
  int leng=sizeof(struct node);        //确定新结点空间的大小
  p=(struct node *)malloc(leng);       //生成新结点,需补充 ???
  p->data=e;                           //装入元素e
  p->next=top;                         //插入新结点
  top=p;                               //top指向新结点
  return top;                          //返回指针top
 }
```



33

(4).链式栈的退栈

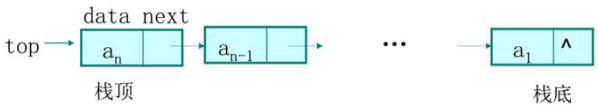



```
p=top;
top=top->next;
e=p->data;
free(p);
```

34

退栈算法

```
struct node *pop(struct node *top,SElemtype *e)
{ struct node *p;
  if (top==NULL) return NULL;     //空栈,返回NULL
  p=top;                          //p指向原栈的顶结点
  (*e)=p->data;                   //取出原栈的顶元素送（*e）
  top=top->next;                  //删除原栈的顶结点
  free(p);                        //释放原栈顶结点的空间
  return top;                     //返回新的栈顶指针top
 }
```



35

几点说明:

1) 链栈不必设头结点，因为栈顶（表头）操作频繁；

2) 链栈一般不会出现栈满情况，除非没有空间导致malloc分配失败。

3) 链栈的入栈、出栈操作就是栈顶的插入与删除操作，修改指针即可完成。

4) 采用链栈存储方式的优点是，可使多个栈共享空间；当栈中元素个数变化较大，且存在多个栈的情况下，链栈是栈的首选存储方式。

36

3.2 栈的应用举例

栈的基本用途----保存暂时不用的数或存储地址。

- 调用函数或子程序
- 实现递归运算
- 用于保护现场和恢复现场
- 简化程序设计

3.2.1 数制转换

算法基于原理：

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

例. 给定十进制数 N=1348,转换为八进制数 R=2504

1.依次求余数,并送入栈中，直到商为0。

(1) r_1=1348%8=4 //求余数

n_1=1348/8=168 //求商

(2) r_2=168%8=0 //求余数

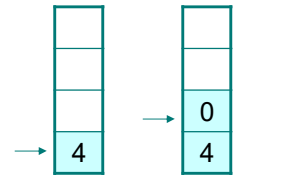
n_2=168/8=21 //求商

(3) r_3=21%8=5 //求余数

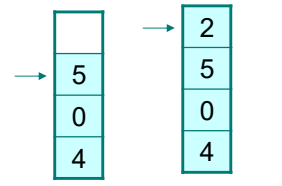
n_3=21/8=2 //求商

(4) r_4=2%8=2 //求余数

n_4=2/8=0 //求商



(1) 4进栈 (2) 0进栈



(3) 5进栈 (4) 2进栈

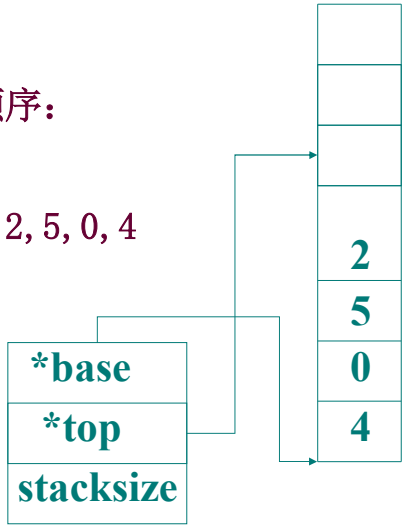
2.依次退栈,得R=2504

先进后出：

数据生成的顺序：

4, 0, 5, 2

读出的顺序：2, 5, 0, 4




```
void conversion () {
//输入任意十进制整数，打印输出与其等值的八进制数
    InitStack(S);
    scanf ("%d",N);
    while (N) {
      Push(S, N % 8);
      N = N/8;
    }
    while (!StackEmpty(S)) {
      Pop(S,e);
      printf ( "%d", e );
    }
} // conversion
```

3.2.2 判定表达式中的括号匹配

1. 括号匹配的表达式

例. {...(...( )...)...}

[...{...( )...( )...}...]

2. 括号不匹配的表达式

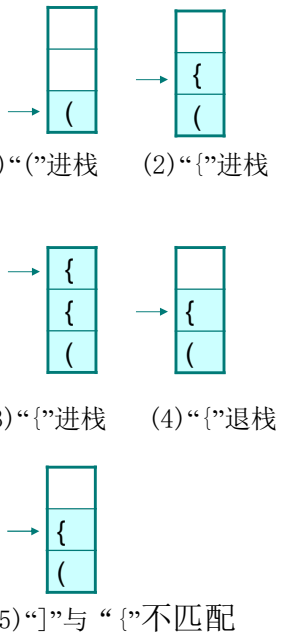
例. {...[}...]

[...(...( )...)...)

3. 判定括号不匹配的方法

例. (...{ ...{ ...}...]

(1) (2) (3) (4) (5)



(1)“(”进栈 (2)“{”进栈

(3)“{”进栈 (4)“{”退栈

(5)“]”与“{”不匹配

41

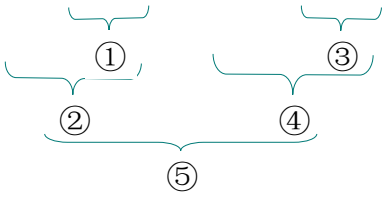
算法的设计思想：

- （1）凡出现左括弧，则进栈；
- （2）凡出现右括弧，首先检查栈是否空
  - 若栈空，则表明该“右括弧”多余
  - 否则和栈顶元素比较，
    - 若相匹配，则“左括弧出栈”
    - 否则表明不匹配
- （3）表达式检验结束时，
  - 若栈空，则表明表达式中匹配正确
  - 否则表明“左括弧”有余

42

3.2.3 表达式求值

例：4 + 2 * 3 – 10 / (7 – 5)



表达式由操作数、运算符和界限符组成。

- 操作数（operand)：常数或变量
- 运算符（operator)

  算术运算符：+、-、*、/、**等

  关系运算符：＜、≤、＝、≠、≥、＞

  逻辑运算符：AND、OR、NOT
- 界限符（delimiter): 左右括号、表达式结束符＃等

43

算符优先关系表

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

求值规则：

1. 先乘除，后加减；
2. 先括号内，后括号外；
3. 同类运算，从左至右。

约定：

θ_1----左算符

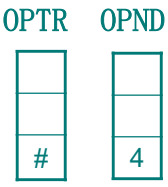
θ_2----右算符

θ_1=#，为开始符

θ_2=#，为结束符

44

表达式求值的算符优先算法



1．使用的数据结构

算符栈OPTR：有效算符；

操作数栈OPND：有效操作数，运算结果。

2．算法思想

（1）初始化：OPND置为空栈，将＃放入OPTR栈底.

（2）依次读入表达式中的每个字符(单词），

若是操作数，则入OPND栈，读取下一“单词”；

若是算符，则和OPTR栈顶算符进行优先级比较：

- 若栈顶算符优先，则执行相应运算？结果存入OPND栈；
- 若与栈顶算符相等，则作（＝）或＃＝＃处理？；
- 若栈顶算符低，该算符入OPTR栈，读取下一“单词”。

（3） 重复（2），直到表达式求值完毕

（读入的是#，且OPTR栈顶元素也为#）

45

```
OperandType Evaluateexpress( ) {
  InitStack(OPTR);  Push(OPTR,'#');
  InitStack(OPND);  c=getchar( );
  while (c!='#' || GetTop(OPTR)!='#') {
   if （c为操作数）{Push(OPND,c);c=getchar();} //不是运算符进栈
   else
      switch (Precede(GetTop(OPTR),c)){
        case '<':  // 栈顶元素优先权低
            Push(OPTR,c);  c=getchar();  break;
        case '=':  // 脱括号并接收下一个字符
            Pop(OPTR,x);   c=getchar();  break;
        case '>':  //退栈并将运算结果入栈
             Pop(OPTR,theta); Pop(OPND,b); Pop(OPND,a);
             Push(OPND,Operate(a,theta,b));  break;
      }//switch
   } //while
   return GetTop(OPND);
}// Evaluateexpress
```

46

例. 求表达式的值：**# 4 + 2 * 3 – 12 / (7 – 5) #**

动态演示栈的应用

步骤	操作数栈	运算符栈	输 入 串	下步操作说明
0		#	4+2*3–12/(7–5)#	操作数进栈
1	4	#	+2*3–12/(7–5)#	p(#)<p(+),进栈
2	4	# +	2*3–12/(7–5)#	操作数进栈
3	42	# +	*3–12/(7–5)#	p(+)<p(*),进栈
4	42	# + *	3–12/(7–5)#	操作数进栈
5	423	# + *	–12/(7–5)#	p(*)>p(-),退栈op=*
6	423	# +	–12/(7–5)#	操作数退栈b=3
7	42	# +	–12/(7–5)#	操作数退栈a=2
8	4	# +	–12/(7–5)#	a*b得c=6进栈

47

步骤	操作数栈	运算符栈	输 入 串	下步操作说明
8	4	#+	–12/(7–5)#	a*b得6进栈
9	46	#+	–12/(7–5)#	p(+)>p(-),退栈op=+
10	46	#	–12/(7–5)#	操作数退栈b=6
11	4	#	–12/(7–5)#	操作数退栈a=4
12		#	–12/(7–5)#	a+b得c=10进栈
13	10	#	–12/(7–5)#	p(#)<p(-),进栈
14	10	# -	12/(7–5)#	操作数进栈
15	1012	# -	/(7–5)#	p(-)<p(/),进栈
16	1012	# - /	(7–5)#	p(/)<p(( ), 进栈
17	1012	# - / (	7–5)#	操作数进栈

48

步骤	操作数栈	运算符栈	输 入 串	下步操作说明
17	1012	# - / (	7–5) #	操作数进栈
18	10127	# - / (	–5) #	p(()<p(-),进栈
19	10127	# - / (-	5) #	操作数进栈
20	101275	# - / (-	) #	p(-)>p()),退栈op=-
21	101275	# - / (	) #	操作数退栈b=5
22	10127	# - / (	) #	操作数退栈a=7
23	1012	# - / (	) #	a-b得c=2进栈
24	10122	# - / (	) #	p(()>p()),去括号
25	10122	# - /	#	p(/)>p(#),退栈op=/
26	10122	# -	#	操作数退栈b=2

49

步骤	操作数栈	运算符栈	输 入 串	下步操作说明
26	10122	# -	#	操作数退栈b=2
27	1012	# -	#	操作数退栈a=12
28	10	# -	#	a/b得c=6进栈
29	106	# -	#	p(-)>p(#),退栈op=-
30	106	#	#	操作数退栈b=6
31	10	#	#	操作数退栈a=10
32		#	#	a-b得c=4进栈
33	**4**	#	#	p(#)=p(#),算法结束

表达式的值

50

3.3 栈与递归的实现

什么是递归？一个直接或间接调用自身的函数被称为是递归（recursion）的。

例1 写出下面C程序段的执行结果。

```
long int fact(int n) {
① long f;
② if(n>1)
③     f=n*fact(n-1);
④ else f=1;
⑤ return(f);
}
```

```
main( )
{ int n;
  long y;
  n=5;
  y=fact(n);
  printf("%d,%ld\n",n,y); }
```

递归运算n！ 运行结果为： 5，120

递归工作栈：保存递归调用的工作记录。
（返址，实参，局部变量）

51

栈与递归的实现

```
long int fact(int n) {
① long f;
② if(n>1)
③     f=n*fact(n-1);
④ else f=1;
⑤ return(f);
}
```

f=5*fact(4);
f=4*fact(3);
f=3*fact(2);
f=2*fact(1);
fact(1) =1;

③	2	2*fact(1)
③	3	3*fact(2)
③	4	4*fact(3)
③	5	5*fact(4)
Addr.	n	f

52

编制递归算法要注意些什么？

- 递归进行是有条件的。一般常把判断语句加在递归语句以前。
- 递归的最底层（base case)应该有返回值，以供上层递归的调用。否则会死循环。
- 参量的初始化应该在递归以前。
- 递归调用需要利用栈。

- 每次调用要把本次调用的参数和局部变量保存在栈顶。
- 每次从下一层调用返回到上一层调用时：
  - ✓从栈顶恢复本层调用的参数和局部变量的值。

53

3.4 队列（排队，queue）

队列是限定仅能在表头进行删除，
表尾进行插入的线性表，
它是一种先进先出（FIFO）的线性表。

$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$

删除　　插入

54

3.4.1 队列及其操作

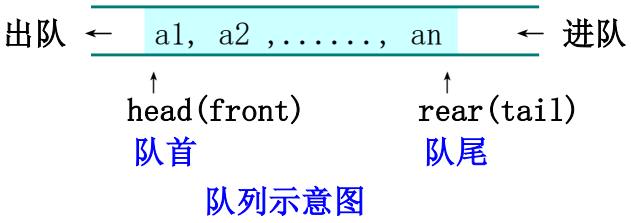
1. 定义和术语

- 队列----只允许在表的一端删除元素,在另一端插入元素的线性表。
- 空队列----不含元素的队列。
- 队首----队列中只允许删除元素的一端。head,front
- 队尾----队列中只允许插入元素的一端。rear,tail
- 队首元素----处于队首的元素。
- 队尾元素----处于队尾的元素。
- 进队----插入一个元素到队列中。又称：入队。
- 出队----从队列删除一个元素。

2. 元素的进出原则:“先进先出”，“First In First Out”

队列的别名:“先进先出”表，“FIFO”表，排队,queue

55



队列示意图

3. 队列的基本操作：

(1) InitQueue(&Q)---- 初始化,将Q置为空队列。

(2) QueueEmpty(Q)----判断Q是否为空队列。

(3) EnQueue(&Q,e)---- 将e插入队列Q的尾端。

(4) DeQueue(&Q,&e)---- 取走队列Q的首元素,送e。

(5) GetHead(Q,&e)---- 读取队列Q的首元素,送e 。

(6) ClearQueue(&Q)----置Q为空队列。

56

队列的抽象数据类型定义

ADT Queue{

数据对象：$D=\{a_i|a_i \in ElemSet;1\le i\le n,n\ge 0;\}$

数据关系：$R=\{<a_i,a_{i+1}>|\ a_i,a_{i+1}\in D,i=1,2,\ldots\ldots,n-1\}$，

约定a_1端为队列头，a_n端为队列尾

基本操作：

InitQueue(&Q)　　*DestroyQueue(&Q)*

QueueEmpty(Q)　　*QueueLength(Q)*

GetHead(Q,&e)　　*EnQueue(&Q,e)*

DeQueue(&Q,&e)　　*ClearQueue(&Q)*

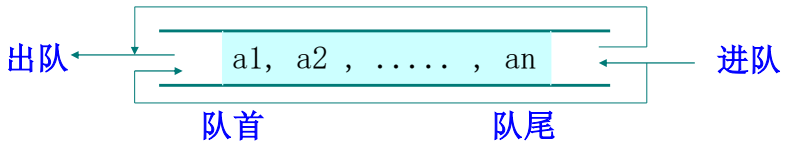
QueueTraverse(Q,visit( ))

}ADT Queue

57

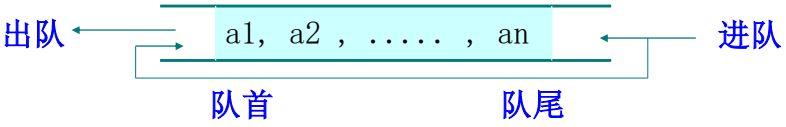
4. 双队列（双端队列，deque----double ended queue）

(1)双队列----只许在表的两端插入、删除元素的线性表。



双队列示意图

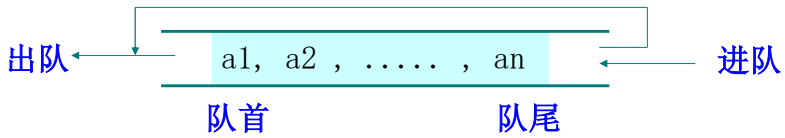
(2)输出受限双队列----只许在表的两端插入、在一端删除元素的线性表。



输出受限双队列示意图

58

(3)输入受限双队列----只允许在表的一端插入、在两端删除元素的线性表。



输入受限双队列示意图

队列类型的实现：

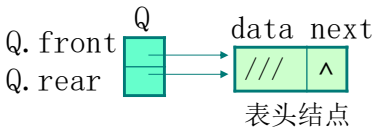
- 链式映象
- 顺序映象

59

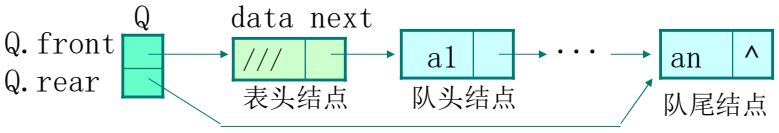
3.4.2 链式队列：用带表头结点的单链表表示队列

1.一般形式

(1)空队列：



(2)非空队列：



其中： Q.front----队头(首)指针,指向表头结点。

Q.rear----队尾指针，指向队尾结点。

Q.front->data 不放元素。

Q.front->next 指向队首结点a1。

60

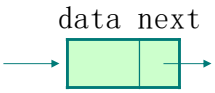
2. 定义结点类型

(1)存放元素的结点类型

```
typedef struct Qnode
{ QElemType data;          //data为抽象元素类型
  struct Qnode *next;      //next为指针类型
 }Qnode, *QueuePtr;        //结点类型, 指针类型
```

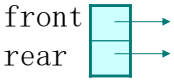
其中：Qnode----结点类型

QueuePtr----指向Qnode的指针类型

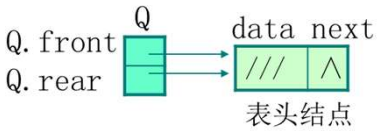


(2)由头、尾指针组成的队列类型

```
typedef struct
{ Qnode *front;   //头指针
  Qnode *rear;    //尾指针
 }LinkQueue;      //链式队列类型
```

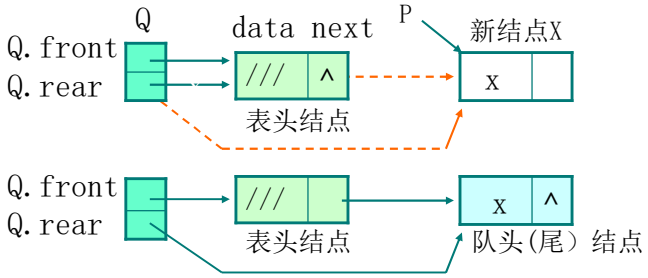


3. 生成空队列算法

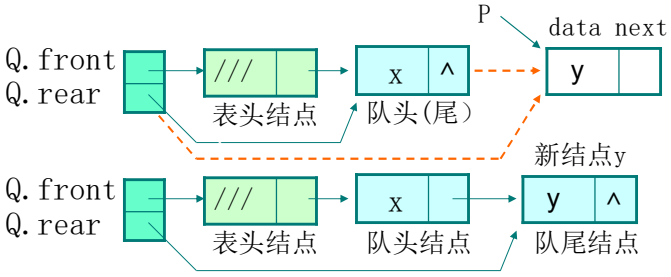



```
Status InitQueue (LinkQueue &Q) {
        // 构造一个空队列Q
  Q.front = Q.rear =(QueuePtr)malloc(sizeof(QNode));
        //生成表头结点
  if (!Q.front) exit (OVERFLOW);
                        //存储分配失败
  Q.front->next = NULL;  //表头结点的next为空指针
  return OK;
}
```

4. （空队列时）插入新元素x

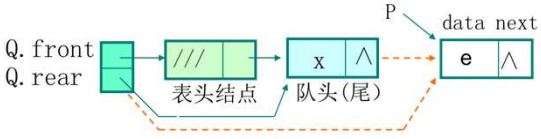


（非空队列时）插入新元素y



63

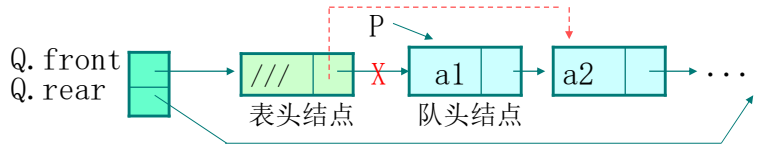
```
Status EnQueue (LinkQueue &Q, QElemType e) {
        // 插入元素e为Q的新的队尾元素
  p = (QueuePtr) malloc (sizeof (QNode));
                                    //生成新元素结点
  if (!p)  exit (OVERFLOW);         //存储分配失败
  p->data = e;                      //装入元素e
  p->next = NULL;                   //为队尾结点
  Q.rear->next = p;                 //插入新结点
  Q.rear = p;                       //修改尾指针
  return OK;
}
```



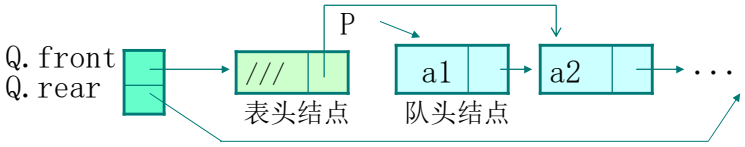
64

5. 出队----删除队头结点

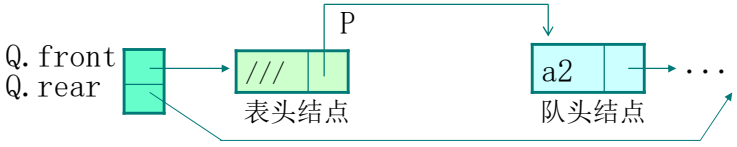
(1)若原队列有2个或2个以上的结点



(a)执行：Q. front->next=p->next;

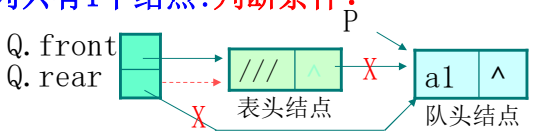


(b)执行：free(p);

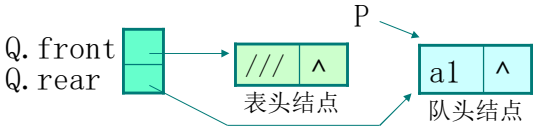


65

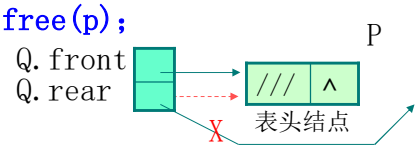
(2)若原队列只有1个结点:判断条件？



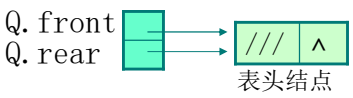
(a)执行：Q. front->next=p->next;



(b)执行：free(p);



(c)执行：Q. rear=Q. front;



66

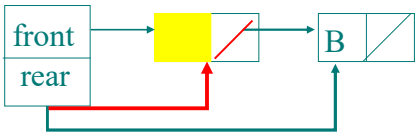
出队时的三种情形：

a.出队前已空；

b.出队前只有一个结点，出队后为空队列；

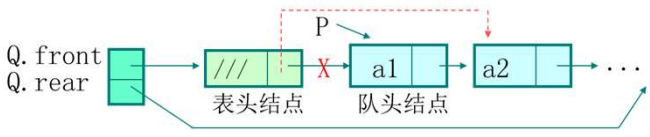
c.其他情形（出队前结点数>1）

思考：在什么情况下出队要修改Q. rear (队尾)指针？为什么？



67

```
Status DeQueue (LinkQueue &Q, QElemType &e) {
    // 若队列不空，则删除Q的队头元素，
    //用e返回其值，并返回OK；否则返回ERROR
  if (Q.front == Q.rear)    return ERROR;   //若原队列为空
  p = Q.front->next;                        //P指向队头结点
  e = p->data;
  Q.front->next = p->next;                  //删除队头结点
  if (Q.rear == p)                      //若原队列只有1个结点
        Q.rear = Q.front;               //修改尾指针
  free (p);
  return OK;
}
```



68

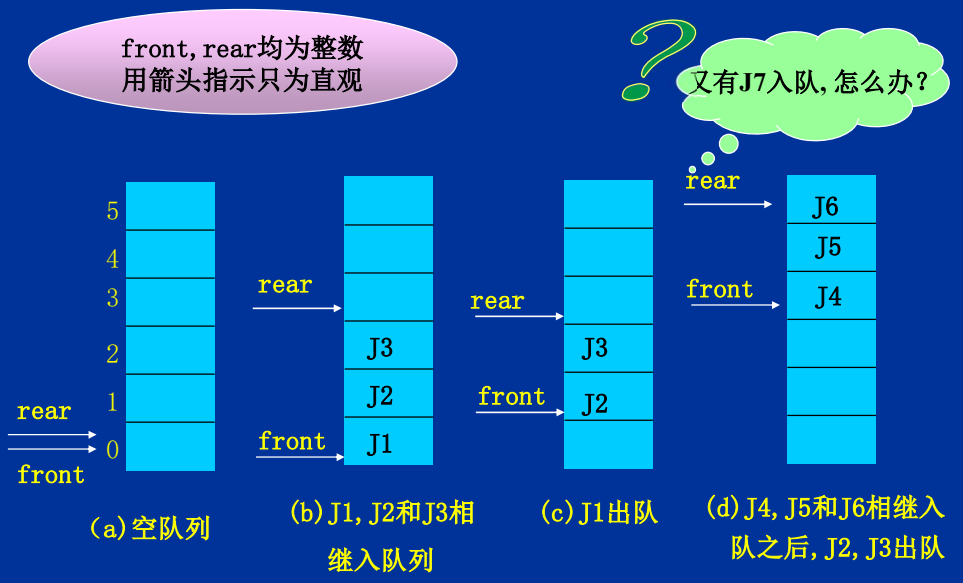
3.4.3 队列的顺序表示和实现

```
#define MAXQSIZE  100      //最大队列长度
typedef struct {
  QElemType  *base;        // 动态分配存储空间
  int  front;              // 头指针，若队列不空，指向队头元素
  int  rear;               // 尾指针，若队列不空，指向队尾元素的下一个位置？
} SqQueue;
```

队列采用顺序存储结构，约定：

- 初始空队列：　front = rear=0 ;
- 插入新的元素时：　rear++;
- 删除队头元素时：　front++;

69



70

顺序队列的“溢出”问题

- 真溢出
  - *Q.rear* − *Q.front* ≥ *Q.Maxsize*
- 假溢出
  - 顺序队列因多次入队和出队操作后出现的有空闲存储空间但不能进行入队操作的溢出。

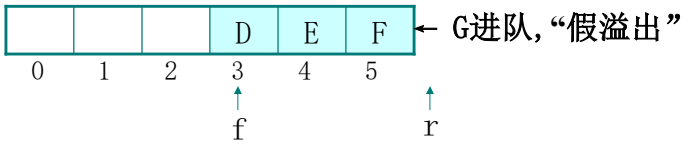
71

如何解决顺序队列的“假溢出”问题？

- 按最大可能的进队操作次数设置顺序队列的最大元素个数；
- 出队时移到：修改出队算法，使每次出队列后都把队列中剩余数据元素向队头方向移动一个位置；
- 假溢出时移到：修改入队算法，增加判断条件，当假溢出时，把队列中的数据元素向队头移动，然后完成入队操作；
- 采用循环队列。

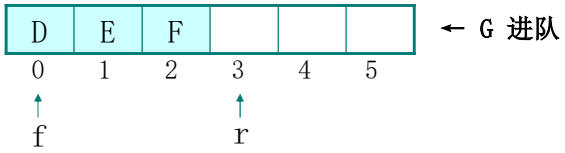
72

例 如下顺序队列，D,E,F 依次进队之后：

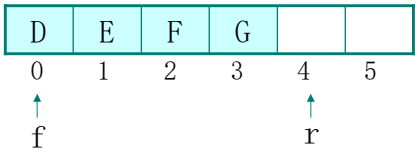


解决假溢出的方法： 移动元素

D,E,F移到前端后：



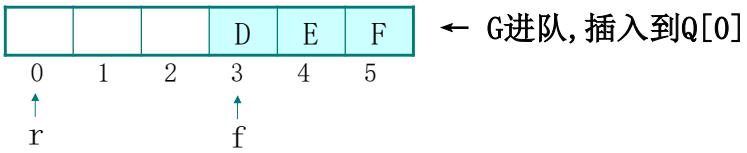
G进队之后：



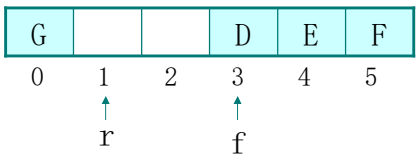
73

解决假溢出的方法： 将Q当循环表使用(循环队列)

(1)



(2) G进Q[0]之后：



74

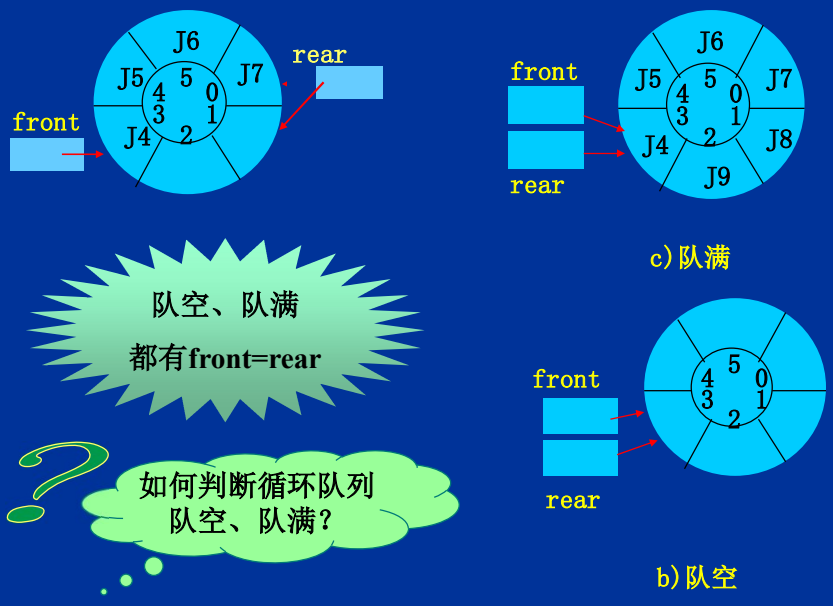
复习：顺序队列实现

```
#define MAXQSIZE  100      //最大队列长度
typedef struct {
  QElemType  *base;        // 动态分配存储空间
  int  front;              // 头指针，若队列不空，指向队列头元素
  int  rear;
                           // 尾指针，若队列不空，指向队列尾元素 的下一个位置
} SqQueue;
```

队列采用顺序存储结构，约定：

- 初始空队列：　front = rear=0 ;
- 插入新的元素时：　rear++;
- 删除队头元素时：　front++;

75



76

二义性问题：

- 在循环队列中，队空特征是*Q.front＝Q.rear*；
- 队满时也会有*Q.front＝Q.rear*；
- 判决条件将出现二义性！

解决方案有三：

- ①使用一个计数器记录队列中元素个数（即队列长度）；
  - 判队满：*count＝MAXQSIZE*
  - 判队空：*count==0*
- ②加设标志位
  - 判队满：*tag==1 && Q.rear==Q.front*
  - 判队空：*tag==0 && Q.rear==Q.front*
- ③ 少用一个存储单元
  - 判队满： *Q.front==(Q.rear+1) % MAXQSIZE*
  - 判队空： *Q.rear==Q.front*

77

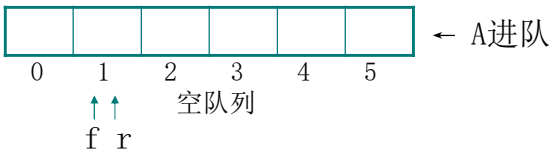
循环队列的实现方法：

设f指向队头元素；r指向队尾元素后一空单元。Q[0..5]为循环队列。

(1)初始化

f=r=1;

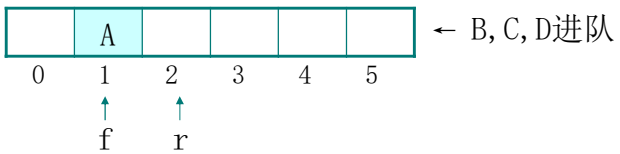
(只要在0到5的范围内相等即可)



(2)A进队

Q[r]=A;

++r;

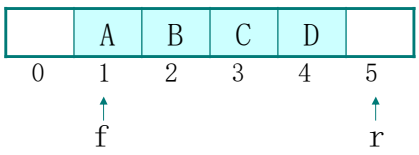


(3)B,C,D进队

Q[r++]=B;

Q[r++]=C;

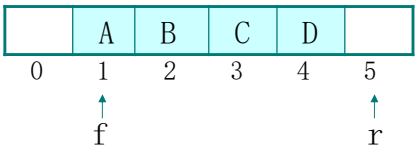
Q[r++]=D;



78

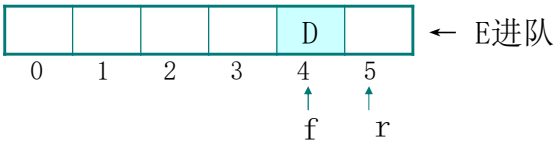
(3)B,C,D进队

```
Q[r++]=B;
Q[r++]=C;
Q[r++]=D;
```



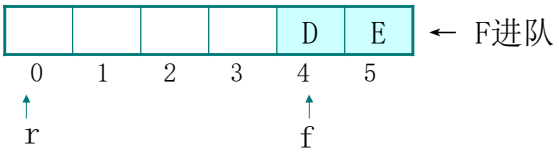
(4)A,B,C出队

```
e1=Q[f++];
e2=Q[f++];
e3=Q[f++];
```



(5)E进队

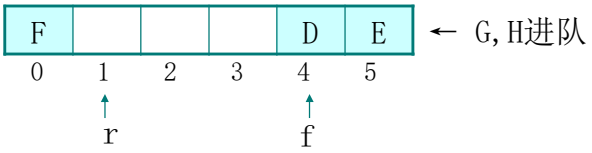
```
Q[r]=E;
r=(r+1)%6
```



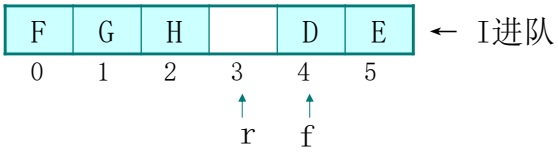
79

(6)F进队

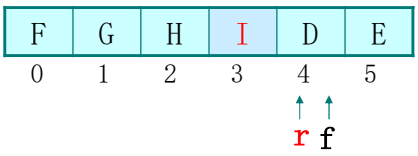
```
Q[r++]=F;
```



(7)G,H进队后：

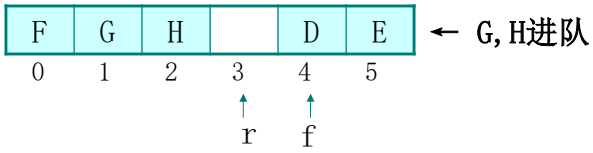


(8)I进队后，导致r=f,产生二义性。



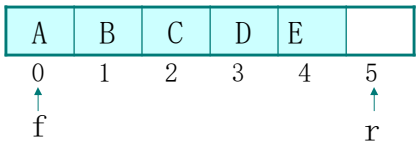
80

方案三少用一个存储单元的空，满队列条件分析：



(1)满队列条件：

若A, B, C, D, E
依次进队后：

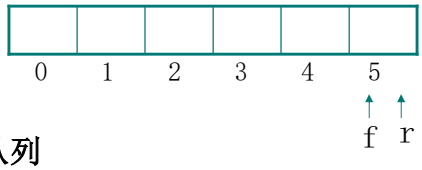


当 r+1==f 或 (f==0)&&(r==MAXQSIZE-1)

即：(r+1)% MAXQSIZE ==f 为满队列

(2)空队列条件：

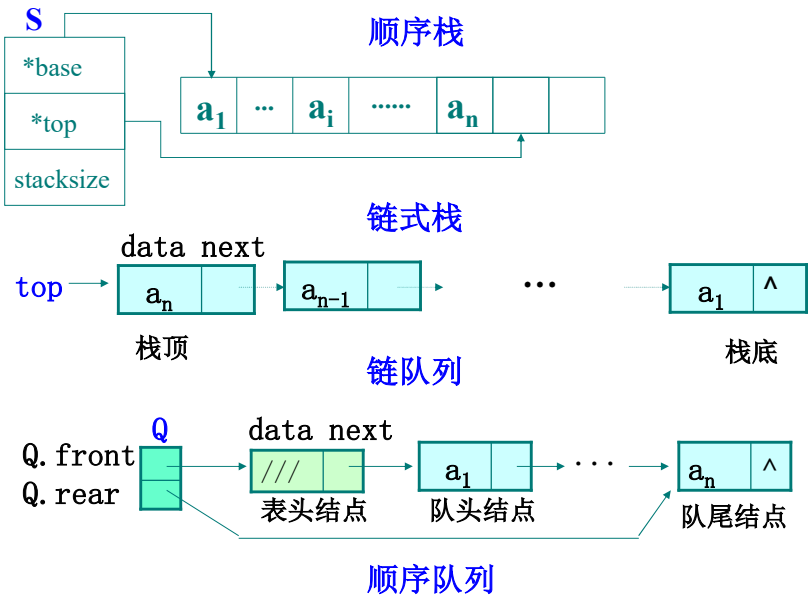
A, B, C, D, E
依次出队后：



当(f==r)为空队列

81

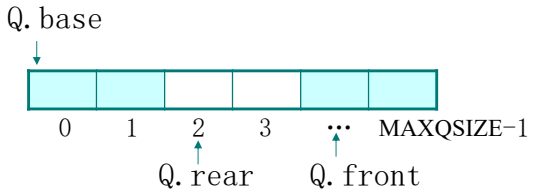
Review

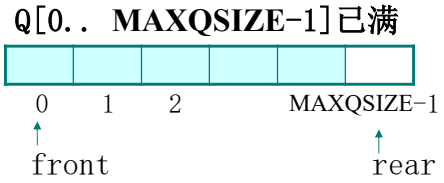


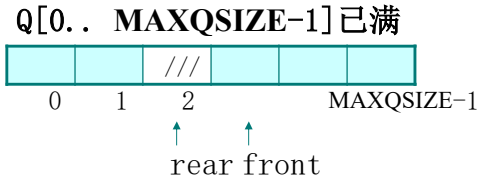
82

循环队列存储结构定义

```
#define MAXQSIZE 100
Typedef struct{
   QElemType *base;
   int front;
   int rear;
}SqQueue，Q;
```



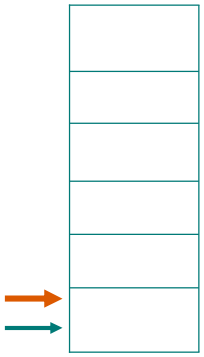




队满：(Q.rear+1)% **MAXQSIZE**==Q.front　　//少用一个单元

83

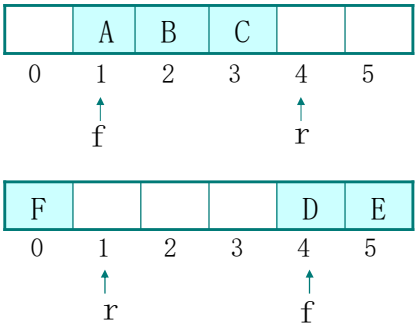
```
Status InitQueue (SqQueue &Q) {
        // 构造一个空队列Q
  Q.base = (QElemType *) malloc(MAXQSIZE *sizeof(QElemType));
  if (!Q.base) exit (OVERFLOW);
        // 存储分配失败
  Q.front = Q.rear = 0;
  return OK;
}
```



空队列

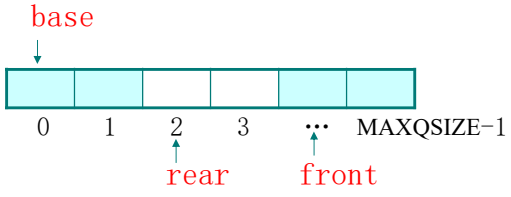
84

```
int QueueLength(SqQueue Q){
        //求队列的长度
  return (Q.rear－Q.front+MAXQSIZE)  % MAXQSIZE;
}
```



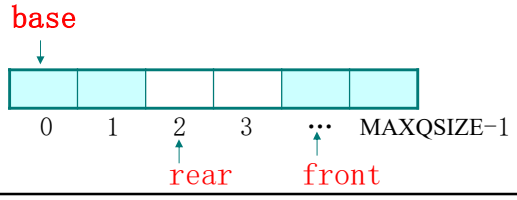
85

```
Status EnQueue (SqQueue &Q, QElemType e) {
      // 插入元素e为Q的新的队尾元素
  if ((Q.rear+1) % MAXQSIZE == Q.front)
    return ERROR;          //队列满
  Q.base[Q.rear] = e;      //装入新元素e
  Q.rear = (Q.rear+1) % MAXQSIZE;
                  //尾指针循环后移一个位置
  return OK;
}
```



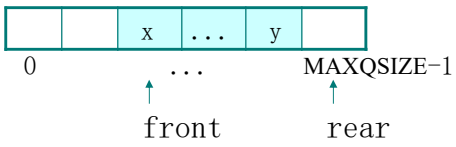
86

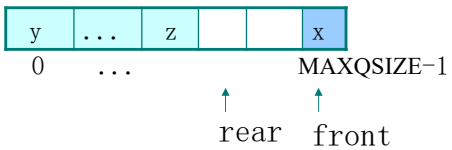
```
Status DeQueue (SqQueue &Q, QElemType &e) {
  // 若队列不空，则删除Q的队头元素，
  // 用e返回其值，并返回OK；否则返回ERROR
  if (Q.front == Q.rear)            // Q为空队列
    return ERROR;
  e = Q.base[Q.front];
  Q.front = (Q.front+1) % MAXQSIZE;
                    //头指针循环后移到一个位置
  return OK;
}
```



87

指针循环后移到一个位置的两种状态：





动态演示循环队列的操作

88

队列应用

判断一个字符序列是否是回文。

- ✓基本思想：将输入字符逐个分别存入队列和栈， 然后逐个出队列和退栈并比较出队列的字符和退栈的字符是否相等，若全部相等则该字符序列是回文，否则就不是回文。

CPU资源的竞争问题

主机与外部设备之间速度不匹配的问题：打印队列

应用系统中的事件规划与事件模拟：银行业务活动

89

本章小结

线性表、栈、队列的异同点：

- ❑ 相同点：
  - 逻辑结构相同，都是线性的；
  - 都可以用顺序存储或链表存储；
  - 栈和队列是两种特殊的线性表，即受限的线性表（只是对插入、删除运算加以限制）；
  - 栈与队列都是典型的串行数据结构。

90

- 不同点：
  - 运算规则不同：
    - ✓线性表为随机存取；
    - ✓而栈是只允许在一端进行插入和删除运算，因而是后进先出表*LIFO*；
    - ✓队列是只允许在一端进行插入、另一端进行删除运算，因而是先进先出表*FIFO*。
  - 用途不同：
    - ✓线性表较通用；
    - ✓栈用于函数调用、递归和简化程序设计等；
    - ✓队列用于离散事件模拟、操作系统作业调度等。

91

基本要求

1. 掌握栈和队列的特点，并能在相应的应用问题中正确选用它们。

2. 掌握栈类型的两种实现方法，注意栈满和栈空的条件以及它们的描述方法。

3. 熟练掌握循环队列和链队列的基本操作实现算法，注意队满和队空的描述方法。

课堂测试

92

讨　论

1. 填空题

(1) 设栈*S*和队列*Q*的初始状态皆为空，元素a_1，a_2，a_3，a_4，a_5和a_6依次通过栈*S*，一个元素出栈后即入队列*Q*，若6个元素出队列的顺序是a_3，a_5，a_4，a_6，a_2，a_1，则栈*S*至少应该容纳__4__个元素。

(2) 利用栈求表达式：

（（9－1）－2－（8－（5－1））） 的值，

运算符栈至少应该有__7__个单元 。

2. 试设计一种方案：用一个定长数组实现两个栈，共享并充分利用该数组存储空间。

93

3. 比较栈和队列的相同点和不同点，举例说明。

4. 对于算术表达式3×(5－2)＋7，用栈存储式子中的运算对象和运算符，说明该算术表达式的运算过程。

5.循环队列的优点是什么？如何判断它的空和满？

6.试述队列的链式存储结构和顺序存储结构的优缺点。

7.在循环队列中，如果用**front**指向队列的第一个元素，**length**表示元素的个数，如何设计如下算法：

```
void IniQueue(&Q);
int EnQueue(&Q,e);
int DeQueue(&Q,&e);
```

94

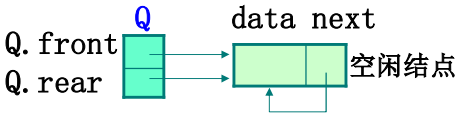
2019. 请设计一个队列，要求满足：初始时队列为空 ；入队时，允许增加队列占用空间；出队后，出队元素所占用的空间可重复使用，即整个队列所占用的空间只增不减；入队操作和出队操作的时间复杂度始终保持为O(1)。请回答下列问题：

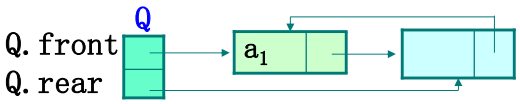
（1）该队列应该选择链式存储结构，还是顺序存储结构？

（2）画出队列的初始状态 ，并给出判断队空和队满的条件。

（3） 画出第一个元素入队后的队列状态。

（4） 给出入队操作和出队操作的基本过程。





95

课外思考：

严- 3.6，

3.28， 3.31.

96