



第三章 栈和队列

逻辑结构和线性表相同

运算受到了限制

根据所受限制的不同，分为栈和队列



栈----线性结构，插入和删除操作受限制

- 栈是限制在表的一端（表尾）进行插入和删除的线性表。
- $(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1}, a_n)$

出栈

插入顺序

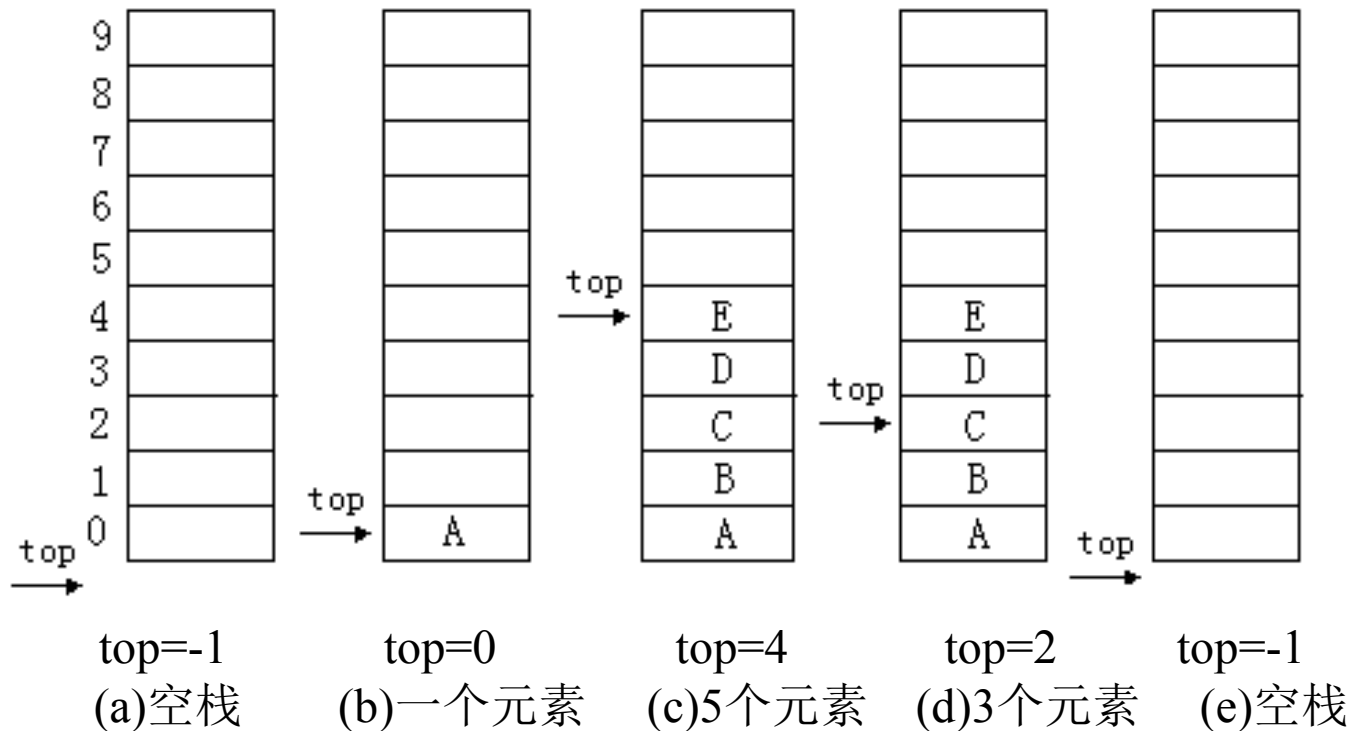
进栈

删除顺序

特点—先进后出

顺序存储结构存放—顺序栈

采用数组实现，需设栈顶指针，指示栈顶的位置





顺序栈

```
#define MAXSIZE 1024
```

```
typedef struct
```

```
{datatype data[MAXSIZE];
```

```
int top;
```

```
}SeqStack;
```



顺序栈

```
#define MAXSIZE 5  
  
typedef struct  
{datatype data[MAXSIZE];  
  int top;  
}SeqStack;  
  
SeqStack s ;
```



顺序栈

```
#define MAXSIZE 5  
  
typedef struct  
{int data[MAXSIZE];  
  int top;  
}SeqStack;  
  
SeqStack s ;
```



顺序栈

初始化: $s.top = -1;$

入栈: 只要有空间则

$s.data[++s.top] = x;$

$s.data[4]$

$s.data[3]$

$s.data[2]$

$s.data[1]$

$s.data[0]$



$s.top = -1$



顺序栈

初始化: $s.top = -1;$

入栈: 只要有空间则

$s.data[++s.top] = x;$

入栈 ($x = a_1$)

$s.data[4]$

$s.data[3]$

$s.data[2]$

$s.data[1]$

$s.data[0]$



← $s.top = 0$

初始化: $s.top = -1$;

顺序栈

入栈: 只要有空间则

$s.data[++s.top] = x$;

入栈 ($x = a_1$)

入栈 ($x = a_2$)

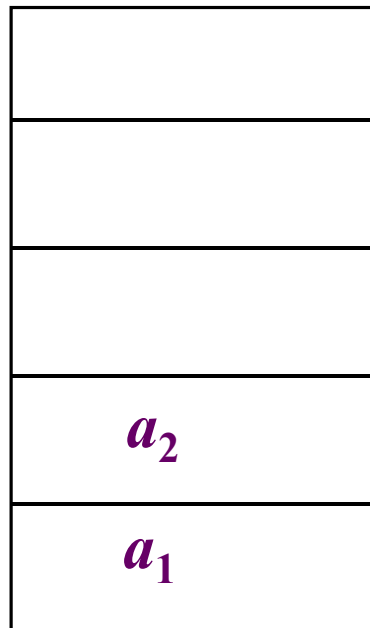
$s.data[4]$

$s.data[3]$

$s.data[2]$

$s.data[1]$

$s.data[0]$



← $s.top = 1$



顺序栈

初始化: $s.top = -1;$

入栈: 只要有空间则

$s.data[++s.top] = x;$

入栈 ($x = a_1$)

入栈 ($x = a_2$)

入栈 ($x = a_3$)

$s.data[4]$

$s.data[3]$

$s.data[2]$

$s.data[1]$

$s.data[0]$

a_3
a_2
a_1

← $s.top = 2$

顺序栈

初始化: $s.top = -1;$

入栈: 只要有空间则

$s.data[++s.top] = x;$

$s.data[4]$

$s.data[3]$

$s.data[2]$

$s.data[1]$

$s.data[0]$

a_4
a_3
a_2
a_1

← $s.top = 3$

入栈 ($x = a_1$)

入栈 ($x = a_2$)

入栈 ($x = a_3$)

入栈 ($x = a_4$)



顺序栈

初始化: $s.top = -1$;

入栈: 只要有空间则

$s.data[++s.top] = x$;

$s.data[4]$

a_5

← $s.top = 4$

$s.data[3]$

a_4

$s.data[2]$

a_3

$s.data[1]$

a_2

$s.data[0]$

a_1

入栈 ($x = a_1$)

入栈 ($x = a_2$)

入栈 ($x = a_3$)

入栈 ($x = a_4$)

入栈 ($x = a_5$)

顺序栈

初始化: $s.top = -1$;

入栈: 只要有空间则

$s.data[++s.top] = x$;

$s.data[4]$

a_5

$s.top = 4$

入栈 ($x = a_1$)

$s.data[3]$

a_4

入栈 ($x = a_2$)

$s.data[2]$

a_3

入栈 ($x = a_3$)

$s.data[1]$

a_2

入栈 ($x = a_4$)

$s.data[0]$

a_1

入栈 ($x = a_5$)

入栈 ($x = a_6$)

溢出

$if(s.top < MAXSIZE - 1)$

$s.data[++s.top] = x$;

$else \text{ printf}(\text{"overflow"});$



顺序栈—出栈

只要非空栈 $s.top--$;

$s.data[4]$	a_5	← $s.top=4$
$s.data[3]$	a_4	
$s.data[2]$	a_3	
$s.data[1]$	a_2	
$s.data[0]$	a_1	



顺序栈—出栈

只要非空栈 $s.top--$;

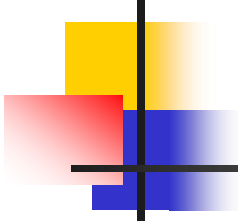
$s.data[4]$	a_5	$\leftarrow s.top=3$
$s.data[3]$	a_4	
$s.data[2]$	a_3	
$s.data[1]$	a_2	
$s.data[0]$	a_1	



顺序栈—出栈

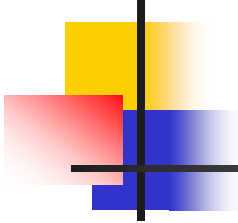
只要非空栈 $s.top--$;

$s.data[4]$	a_5	← $s.top=2$
$s.data[3]$	a_4	
$s.data[2]$	a_3	
$s.data[1]$	a_2	
$s.data[0]$	a_1	



顺序栈—入栈 ($x=a_6$)

s.data[4]	a_5	← s.top=2
s.data[3]	a_4	
s.data[2]	a_3	
s.data[1]	a_2	
s.data[0]	a_1	



顺序栈—入栈 ($x=a_6$)

s.data[4]	a_5	← s.top=3
s.data[3]	a_6	
s.data[2]	a_3	
s.data[1]	a_2	
s.data[0]	a_1	



顺序栈—出栈

只要非空栈 $s.top--$;

$s.data[4]$	a_5	$\leftarrow s.top=2$
$s.data[3]$	a_6	
$s.data[2]$	a_3	
$s.data[1]$	a_2	
$s.data[0]$	a_1	



顺序栈—出栈

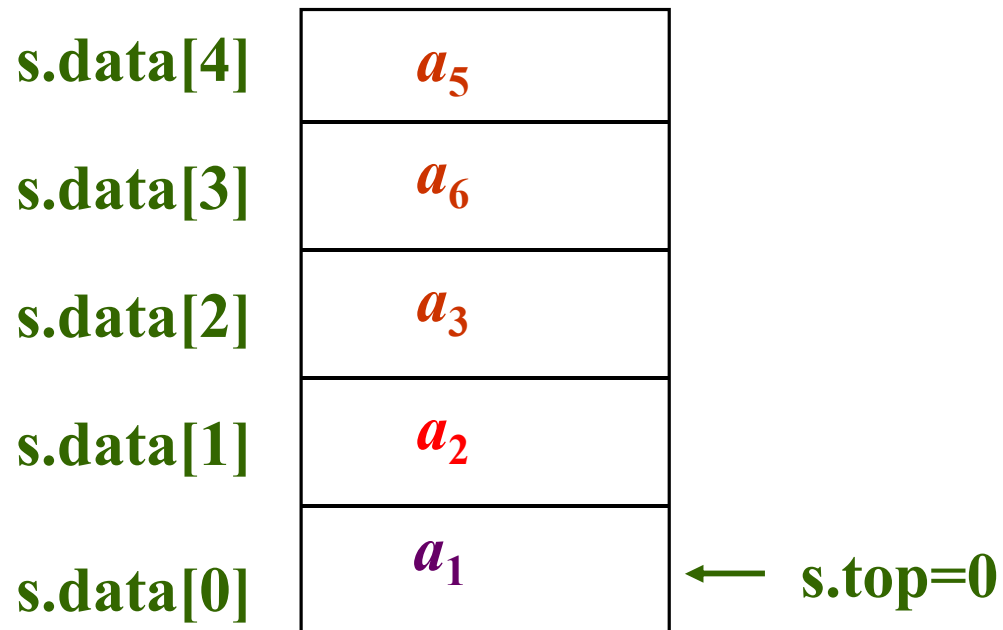
只要非空栈 $s.top--;$

$s.data[4]$	a_5	$\leftarrow s.top=1$
$s.data[3]$	a_6	
$s.data[2]$	a_3	
$s.data[1]$	a_2	
$s.data[0]$	a_1	



顺序栈—出栈

只要非空栈 $s.top--$;





顺序栈—出栈

只要非空栈 $s.top--$;

$if(s.top \geq 0) \ s.top--$;

$s.data[4]$

a_5

$s.data[3]$

a_6

$s.data[2]$

a_3

$s.data[1]$

a_2

$s.data[0]$

a_1

对空栈进行删除称为下溢出

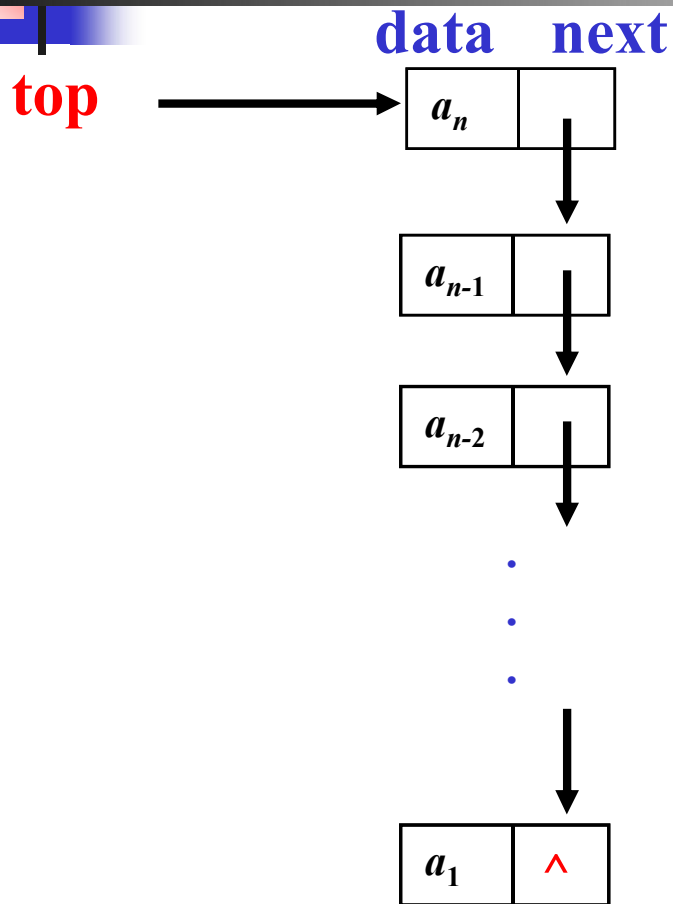
← $s.top = -1$



顺序栈

- 初始化: `s.top=-1;`
- 入栈: `if(s.top<MAXSIZE-1)`
`s.data[++s.top]=x;`
`else printf(“overflow”);`
- 出栈: `if(s.top>=0)`
`s.top--;else{...}`
- 判栈空: `s.top==-1`
- 判栈满: `s.top==MAXSIZE-1`
- 栈顶元素: `s.data[s.top]`

采用链式存储结构存放--链栈

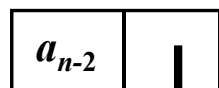
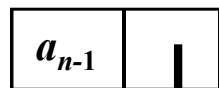
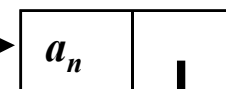


- 不带表头结点的单链表
- 根据栈的定义，存放每个数据元素的结点空间的指针----该数据元素的直接前驱
- 保留栈顶的位置即可----单链表的头指针对应为栈顶的位置
- 出栈在栈顶进行----删除单链表的第一个数据元素结点
- 入栈在栈顶进行----在单链表的第一个数据元素结点前插入一个新的数据元素

链栈

top

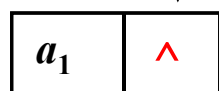
data next



.

.

.



```
■ typedef struct node{
    int data ;
    struct node *next; //直接前驱
}Node, *LinkList;
```

```
■ LinkList top;
```

```
■ //出栈
```

```
➤ if (top)
```

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

```
■ //入栈—将x入栈
```

```
➤ s=(LinkList)malloc(sizeof(Node));
```

```
➤ s->data=x;
```


```
➤ s->next=top;
```

```
➤ top=s;
```

栈的应用—括号匹配的检验

- 正确的表达式:
 - 括号要成对出现
 - 表达式中括号不允许出现骑跨现象
- 例： $a+\{2-[b+c]*(8*[8+g]/[m-e]-7)-p\}$
- 忽略表达式中的运算对象和运算符，只看括号：
 - $\{[]([][])\}$ ---- 成对出现 😊
 - $\{[()]\}$ ---- 不允许出现骑跨现象 ☹️

从左至右读取表达式，读到运算对象和运算符，不做任何动作，直接往下继续读，读到**括号**，**要对括号**的使用是否正确进行检查


$$2+\{a*[b+d]-(w/[17-8]*[7+7])\}$$

从左至右读取表达式，读到运算对象和运算符，不做任何动作，直接往下继续读，读到**括号**，要对**括号**的使用是否正确进行检查


$$2+\{a*[b+d]-(w/[17-8]*[7+7])\}$$

从左至右读取表达式，读到运算对象和运算符，不做任何动作，直接往下继续读，读到**括号**，要对**括号**的使用是否正确进行检查



{ [] ([] []) }

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

s.top=-1

借助栈实现表达式括号合法性检查--采用一个栈保存扫描过的未配对的左括号，从表达式中读到**左**括号时入栈保存，读到**右**括号取栈顶保存的左括号进行配对检查



s.data[4]

s.data[3]

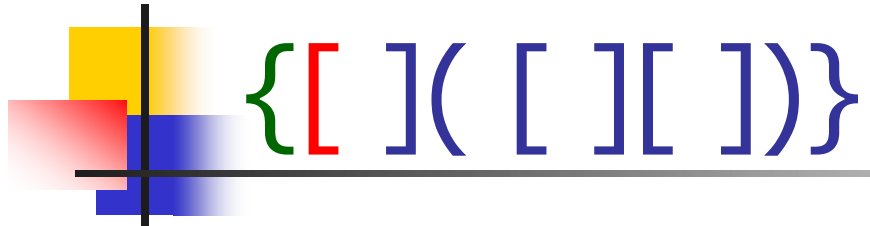
s.data[2]

s.data[1]

s.data[0]



← **s.top=0**



s.data[4]

s.data[3]

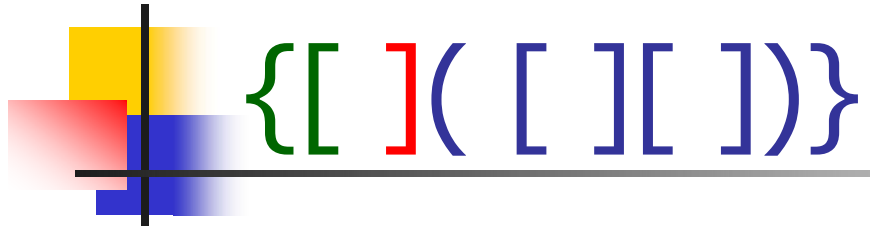
s.data[2]

s.data[1]

s.data[0]

[
{

← s.top=1



s.data[4]

s.data[3]

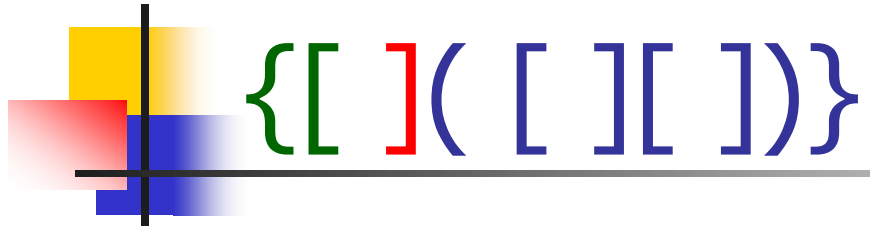
s.data[2]

s.data[1]

s.data[0]

[
{

← s.top=1



s.data[4]

s.data[3]

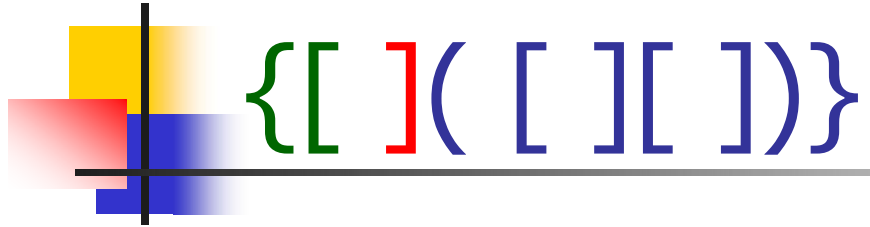
s.data[2]

s.data[1]

s.data[0]

[
{

← s.top=1



s.data[4]

s.data[3]

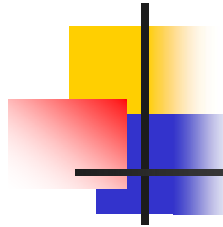
s.data[2]

s.data[1]

s.data[0]

[
{

← s.top=0



{ [] ([] []) }

s.data[4]

s.data[3]

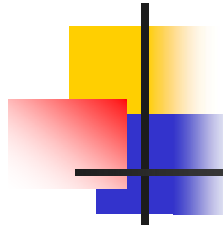
s.data[2]

s.data[1]

s.data[0]

[
{

← s.top=0



{ [] ([] []) }

s.data[4]

s.data[3]

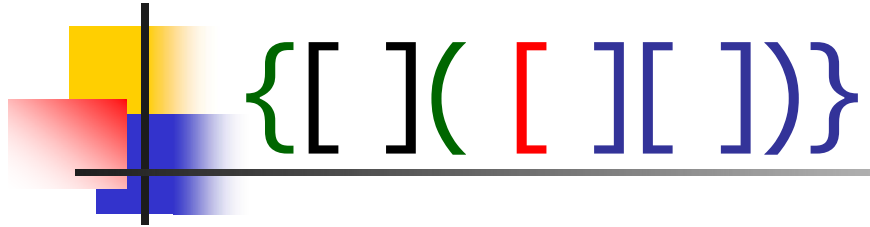
s.data[2]

s.data[1]

s.data[0]

(
{

← s.top=1



s.data[4]

s.data[3]

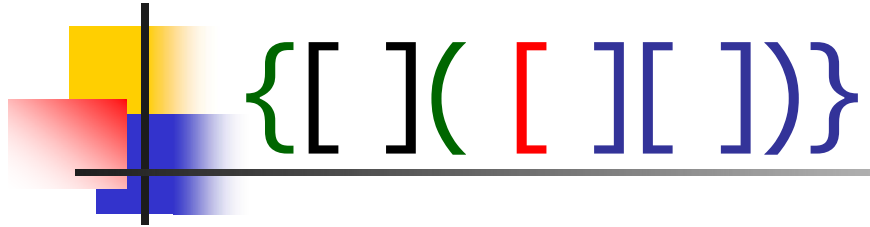
s.data[2]

s.data[1]

s.data[0]

(
{

← s.top=1



s.data[4]

s.data[3]

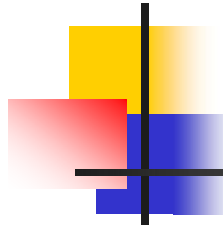
s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=2



{ [] ([] []) }

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=2



{ [] ([] []) }

s.data[4]

s.data[3]

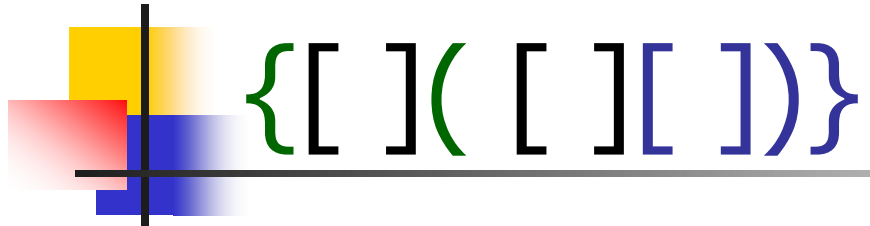
s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=2



s.data[4]

s.data[3]

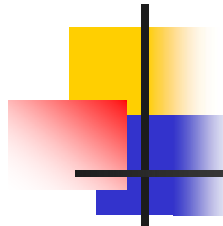
s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=1



{ [] ([] []) }

s.data[4]

s.data[3]

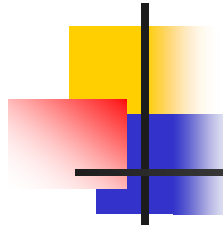
s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=1



{ [] ([] []) }

s.data[4]

s.data[3]

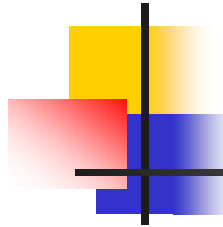
s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=2



{ [] ([] []) }

s.data[4]

s.data[3]

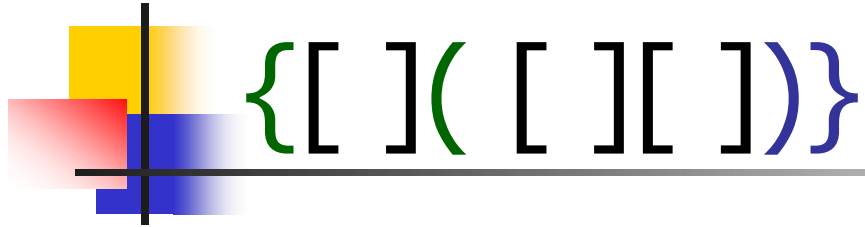
s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=2



s.data[4]

s.data[3]

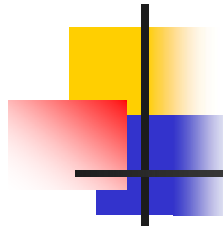
s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=1



{ [] ([] []) }

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=1



s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=0



s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=0



s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

[
(
{

← s.top=-1



问题

■ $\{ [] ([] []) \}$, $\{ [] ([(])) \}$

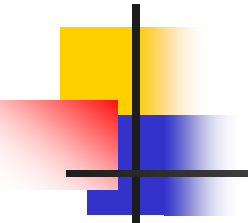
- 左右括号不配对
- 读取的右括号与栈顶保存的左括号形状不匹配

■ $\{ [] ([] []) \})$ 右括号多

- 读取了右括号，但栈空，没有左括号与之配对

■ $(\{ [] ([] []) \}$ 左括号多

- 表达式读取结束，栈中还有保留的左括号没配对
- 为了判断表达式是否读取结束，在表达式尾部加#
为表达式结束标志。
- 例： $\{ [] ([] []) \} \#$



1. 初始化一个空栈;

3. 读一个字符存入变量ch;

4. 若ch=='#', 转5; 否则:

4.1. 若ch为左括号, 进栈, 读下一个字符到ch; 转4;

4.2. 若ch为 ')', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '(', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.3. 若ch为 ']', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '[', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.4. 若ch为 '}', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '{', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.5. 读下一个字符到ch; 转4;

5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。



1. **s.top=-1;**

3. 读一个字符存入变量ch;

4. 若ch=='#', 转5; 否则:

4.1. 若ch为左括号, 进栈, 读下一个字符到ch; 转4;

4.2. 若ch为 ')', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '(', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.3. 若ch为 ']', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '[', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.4. 若ch为 '}', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '{', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.5. 读下一个字符到ch; 转4;

5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。



1. **s.top=-1;**

3. 读一个字符存入变量ch;

4. 若ch=='#', 转5; 否则:

4.1. 若ch为左括号, 进栈, 读下一个字符到ch; 转4;

4.2. 若ch为 ')', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '(', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.3. 若ch为 ']', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '[', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.4. 若ch为 '}', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '{', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.5. 读下一个字符到ch; 转4;

5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。



1. **s.top=-1;**

3. **scanf("%c",&ch);**

4. 若ch=='#', 转5; 否则:

4.1. 若ch为左括号, 进栈, 读下一个字符到ch; 转4;

4.2. 若ch为 ')', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '(', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.3. 若ch为 ']', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '[', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.4. 若ch为 '}', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '{', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.5. 读下一个字符到ch; 转4;

5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。



1. **s.top=-1;**

3. **scanf("%c",&ch);**

4. **while(ch!='#'){**

4.1. 若ch为左括号，进栈，读下一个字符到ch；转4；

4.2. 若ch为 ')', 若栈空，则“括号不匹配”，结束。

若栈顶不是 '(', 则“括号不匹配”，结束。

否则，退栈，读下一个字符到ch；转4；

4.3. 若ch为 ']', 若栈空，则“括号不匹配”，结束。

若栈顶不是 '[', 则“括号不匹配”，结束。

否则，退栈，读下一个字符到ch；转4；

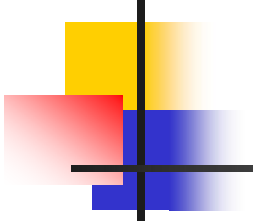
4.4. 若ch为 '}', 若栈空，则“括号不匹配”，结束。

若栈顶不是 '{', 则“括号不匹配”，结束。

否则，退栈，读下一个字符到ch；转4；

4.5. 读下一个字符到ch；转4；

5. 若栈空，则“括号匹配”，结束。否则“括号不匹配”，结束。

- 
1. `s.top=-1;`
 3. `scanf("%c",&ch);`
 4. `while(ch!='#'){`
 - 4.1. `if((ch=='(')||(ch=='[')||(ch=='{'))`
`if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}`
`else{s.data[++s.top]=ch; scanf("%c",&ch);}`
`else`
 - 4.2. 若ch为 ')', 若栈空, 则“括号不匹配”, 结束。
若栈顶不是 '(', 则“括号不匹配”, 结束。
否则, 退栈, 读下一个字符到ch; 转4;
 - 4.3. 若ch为 ']', 若栈空, 则“括号不匹配”, 结束。
若栈顶不是 '[', 则“括号不匹配”, 结束。
否则, 退栈, 读下一个字符到ch; 转4;
 - 4.4. 若ch为 '}', 若栈空, 则“括号不匹配”, 结束。
若栈顶不是 '{', 则“括号不匹配”, 结束。
否则, 退栈, 读下一个字符到ch; 转4;
 - 4.5. 读下一个字符到ch; 转4;
 5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。



1. **s.top=-1;**

3. **scanf("%c",&ch);**

4. **while(ch!='#'){**

4.1. **if((ch=='(')||(ch=='[')||(ch=='{'))**

if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}

else{s.data[++s.top]=ch; scanf("%c",&ch);}

else

4.2. **if(ch=='')** 若栈空，则“括号不匹配”，结束。

若栈顶不是 '(', 则“括号不匹配”，结束。

否则，退栈，读下一个字符到ch；转4；

4.3. 若ch为 ']', 若栈空，则“括号不匹配”，结束。

若栈顶不是 '[', 则“括号不匹配”，结束。

否则，退栈，读下一个字符到ch；转4；

4.4. 若ch为 '}', 若栈空，则“括号不匹配”，结束。

若栈顶不是 '}', 则“括号不匹配”，结束。

否则，退栈，读下一个字符到ch；转4；

4.5. 读下一个字符到ch；转4；

5. 若栈空，则“括号匹配”，结束。否则“括号不匹配”，结束。

- 
1. `s.top=-1;`
 3. `scanf("%c",&ch);`
 4. `while(ch!='#'){`
 - 4.1. `if((ch=='(')||(ch=='[')||(ch=='{'))`
`if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}`
`else{s.data[++s.top]=ch; scanf("%c",&ch);}`
`else`
 - 4.2. `if(ch==')')`
`if((s.top==-1) || (s.data[s.top]!='(')){printf("NoMatch");return;}`
`否则，退栈，读下一个字符到ch；转4；`
 - 4.3. 若ch为 ']', 若栈空，则“括号不匹配”，结束。
`若栈顶不是 '['，则“括号不匹配”，结束。`
`否则，退栈，读下一个字符到ch；转4；`
 - 4.4. 若ch为 '}', 若栈空，则“括号不匹配”，结束。
`若栈顶不是 '{'，则“括号不匹配”，结束。`
`否则，退栈，读下一个字符到ch；转4；`
 - 4.5. 读下一个字符到ch；转4；
 5. 若栈空，则“括号匹配”，结束。否则“括号不匹配”，结束。



1. `s.top=-1;`

3. `scanf("%c",&ch);`

4. `while(ch!='#'){`

4.1. `if((ch=='(')||(ch=='[')||(ch=='{'))`

`if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}`

`else{s.data[++s.top]=ch; scanf("%c",&ch);}`

`else`

4.2. `if(ch==')')`

`if((s.top==-1) || (s.data[s.top]!='(')){printf("NoMatch");return;}`

`else{s.top--; scanf("%c",&ch);}`

4.3. 若ch为 ']', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '[', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.4. 若ch为 '}', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '{', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.5. 读下一个字符到ch; 转4;

5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。



1. **s.top=-1;**

3. **scanf("%c",&ch);**

4. **while(ch!='#'){**

4.1. **if((ch=='(')||(ch=='[')||(ch=='{''))**

if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}

else{s.data[++s.top]=ch; scanf("%c",&ch);}

else

4.2. **if(ch==')')**

if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}

else{s.top--; scanf("%c",&ch);}

4.3. **else if(ch==']')** 若栈空，则“括号不匹配”，结束。

若栈顶不是 '['，则“括号不匹配”，结束。

否则，退栈，读下一个字符到ch；转4；

4.4. 若ch为 '}', 若栈空，则“括号不匹配”，结束。

若栈顶不是 '{'，则“括号不匹配”，结束。

否则，退栈，读下一个字符到ch；转4；

4.5.读下一个字符到ch；转4；

5. 若栈空，则“括号匹配”，结束。否则“括号不匹配”，结束。

- 
1. `s.top=-1;`
 3. `scanf("%c",&ch);`
 4. `while(ch!='#'){`
 - 4.1. `if((ch=='(')||(ch=='[')||(ch=='{'))`
`if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}`
`else{s.data[++s.top]=ch; scanf("%c",&ch);}`
`else`
 - 4.2. `if(ch==')')`
`if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}`
`else{s.top--; scanf("%c",&ch);}`
 - 4.3. `else if(ch==']')`
`if((s.top==-1) ||(s.data[s.top]!='[')){printf("NoMatch");return;}`
`否则，退栈，读下一个字符到ch；转4；`
 - 4.4. 若ch为 '}', 若栈空，则“括号不匹配”，结束。
`若栈顶不是 '}', 则“括号不匹配”，结束。`
`否则，退栈，读下一个字符到ch；转4；`
 - 4.5. 读下一个字符到ch；转4；
 5. 若栈空，则“括号匹配”，结束。否则“括号不匹配”，结束。

- 
1. `s.top=-1;`
 3. `scanf("%c",&ch);`
 4. `while(ch!='#'){`
 - 4.1. `if((ch=='(')||(ch=='[')||(ch=='{'))`
`if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}`
`else{s.data[++s.top]=ch; scanf("%c",&ch);}`
`else`
 - 4.2. `if(ch==')')`
`if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}`
`else{s.top--; scanf("%c",&ch);}`
 - 4.3. `else if(ch==']')`
`if((s.top==-1) ||(s.data[s.top]!='[')){printf("NoMatch");return;}`
`else{s.top--; scanf("%c",&ch);}`
 - 4.4. 若ch为 '}', 若栈空, 则“括号不匹配”, 结束。
若栈顶不是 '{', 则“括号不匹配”, 结束。
否则, 退栈, 读下一个字符到ch; 转4;
 - 4.5. 读下一个字符到ch; 转4;
 5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。

- 
1. `s.top=-1;`
 3. `scanf("%c",&ch);`
 4. `while(ch!='#'){`
 - 4.1. `if((ch=='(')||(ch=='[')||(ch=='{'))`
`if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}`
`else{s.data[++s.top]=ch; scanf("%c",&ch);}`
 - `else`
 - 4.2. `if(ch==')')`
`if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}`
`else{s.top--; scanf("%c",&ch);}`
 - 4.3. `else if(ch==']')`
`if((s.top==-1) ||(s.data[s.top]!='[')){printf("NoMatch");return;}`
`else{s.top--; scanf("%c",&ch);}`
 - 4.4. `else if(ch=='}')`
`if((s.top==-1) ||(s.data[s.top]!='{')){printf("NoMatch");return;}`
`else{s.top--; scanf("%c",&ch);}`
 - 4.5. 读下一个字符到ch; 转4;
 5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。



```
1. s.top=-1;
3. scanf("%c",&ch);
4. while(ch!='#'){
    4.1. if((ch=='(')||(ch=='[')||(ch=='{'))
        if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}
        else{s.data[++s.top]=ch; scanf("%c",&ch);}
    else
    4.2. if(ch==')')
        if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}
        else{s.top--; scanf("%c",&ch);}
    4.3. else if(ch==']')
        if((s.top==-1) ||(s.data[s.top]!='[')){printf("NoMatch");return;}
        else{s.top--; scanf("%c",&ch);}
    4.4. else if(ch=='}')
        if((s.top==-1) ||(s.data[s.top]!='{')){printf("NoMatch");return;}
        else{s.top--; scanf("%c",&ch);}
    4.5. else scanf("%c",&ch);}
5. 若栈空, 则 “括号匹配”, 结束。否则 “括号不匹配”, 结束。
```




```
1. s.top=-1;
3. scanf("%c",&ch);
4. while(ch!='#'){
    4.1. if((ch=='(')||(ch=='[')||(ch=='{'))
        if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}
        else{s.data[++s.top]=ch; scanf("%c",&ch);}
    else
    4.2. if(ch==')')
        if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}
        else{s.top--; scanf("%c",&ch);}
    4.3. else if(ch==']')
        if((s.top==-1) ||(s.data[s.top]!='[')){printf("NoMatch");return;}
        else{s.top--; scanf("%c",&ch);}
    4.4. else if(ch=='}')
        if((s.top==-1) ||(s.data[s.top]!='{')){printf("NoMatch");return;}
        else{s.top--; scanf("%c",&ch);}
    4.5. else scanf("%c",&ch);}
5. if(s.top==-1) printf("Match"); else printf("NoMatch");return;
```



```
s.top=-1;
scanf("%c",&ch);
while(ch!='#'){
    if((ch=='(')||(ch=='[')||(ch=='{'))
        if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}
        else{s.data[++s.top]=ch; scanf("%c",&ch);}
    else
        if(ch==')')
            if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}
            else{s.top--; scanf("%c",&ch);}
        else if(ch==']')
            if((s.top==-1) ||(s.data[s.top]!='[')){printf("NoMatch");return;}
            else{s.top--; scanf("%c",&ch);}
        else if(ch=='}')
            if((s.top==-1) ||(s.data[s.top]!='{')){printf("NoMatch");return;}
            else{s.top--; scanf("%c",&ch);}
        else scanf("%c",&ch);}
if(s.top==-1) printf("Match"); else printf("NoMatch");return;
```



```
void s1(){
s.top=-1;
scanf("%c",&ch);
while(ch!='#'){
if((ch=='(')||(ch=='[')||(ch=='{'))
if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}
else{s.data[++s.top]=ch; scanf("%c",&ch);}
else
if(ch==')')
if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}
else{s.top--; scanf("%c",&ch);}
else if(ch==']')
if((s.top==-1) ||(s.data[s.top]!='[')){printf("NoMatch");return;}
else{s.top--; scanf("%c",&ch);}
else if(ch=='}')
if((s.top==-1) ||(s.data[s.top]!='{')){printf("NoMatch");return;}
else{s.top--; scanf("%c",&ch);}
else scanf("%c",&ch);}
if(s.top==-1) printf("Match"); else printf("NoMatch");return;}
```



栈的应用

- 表达式中括号是否合法
- 将由 $+$, $-$, $*$, $/$ 和单字母变量组成的普通表达式转换成逆波兰式。
- 表达式求值




栈的应用

- 将由 $+$, $-$, $*$, $/$ 和单字母变量组成的普通表达式转换成逆波兰式。
- 表达式的表示形式
 - 1 前缀形式----波兰式
 - 2 中缀形式
 - 3 后缀形式----逆波兰式



前缀形式


■ $a+b$  $+ ab$

■ $a+b*c$

■ $a+b*c-e$



前缀形式

■ $a+b$  $+ab$

■ $a+b*c$  $+ a*bc$

■ $a+b*c-e$



前缀形式

■ $a+b$ \longrightarrow $+ab$

■ $a+b*c$ \longrightarrow $+a*bc$

■ $a+b*c-e$ \longrightarrow $-+a*bce$




后缀形式

- $a+b$
- $a+b*c$
- $a+b*c-e$



后缀形式


■ $a+b$  $ab+$

■ $a+b*c$

■ $a+b*c-e$



后缀形式

■ $a+b$  $ab+$

■ $a+b*c$  $abc*+$

■ $a+b*c-e$



后缀形式

■ $a+b$ \longrightarrow $ab+$

■ $a+b*c$ \longrightarrow $abc*+$

■ $a+b*c-e$ \longrightarrow $abc*+e-$



求逆波兰式

- $(a+b)*c-d+e/f$



求逆波兰式

- $(a+b)*c-d+e/f$
- $a*b+c-d/e$



求逆波兰式

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



求逆波兰式

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

s.top=-1



求逆波兰式

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

s.top=-1



求逆波兰式——*a*

- *a***b*+*c*-*d*/*e*#
- 构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

s.top=-1



求逆波兰式——*a*

- $a * b + c - d / e \#$
- 构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

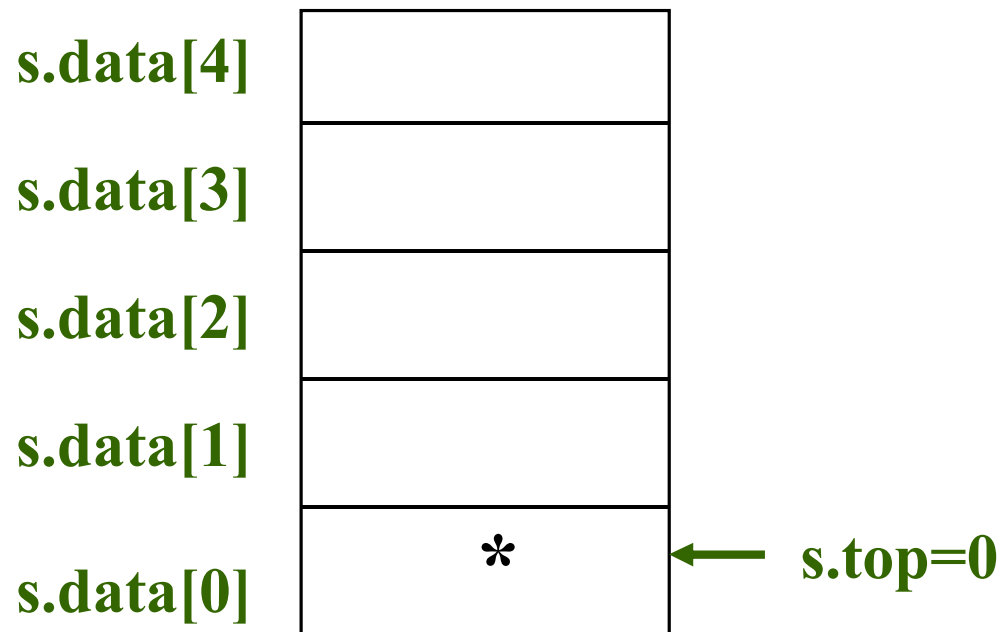
s.data[1]

s.data[0]

s.top=-1

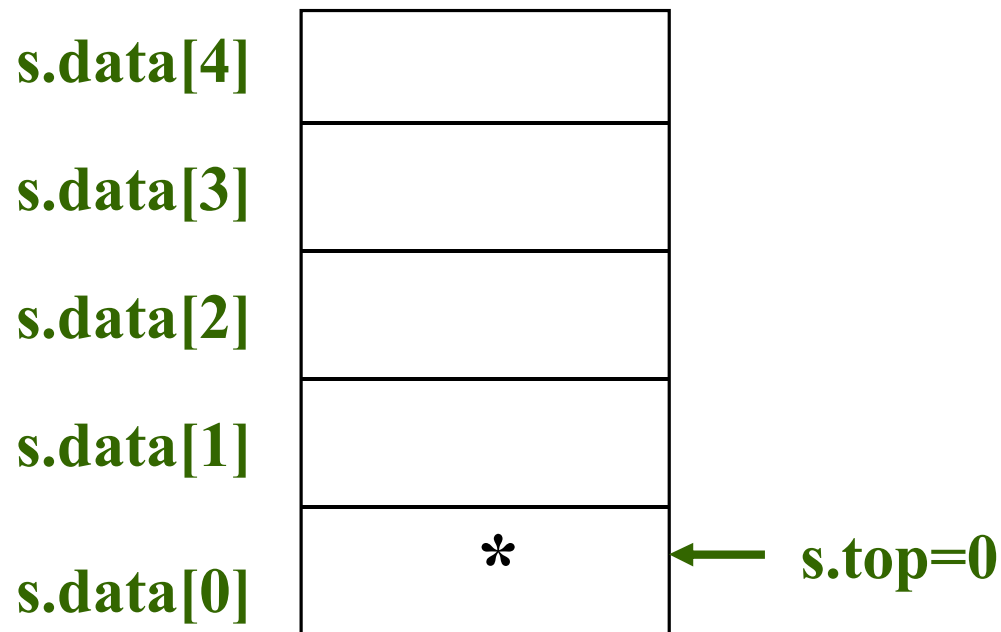
求逆波兰式——*a*

- $a * b + c - d / e \#$
- 构造一个栈保存相应的运算符



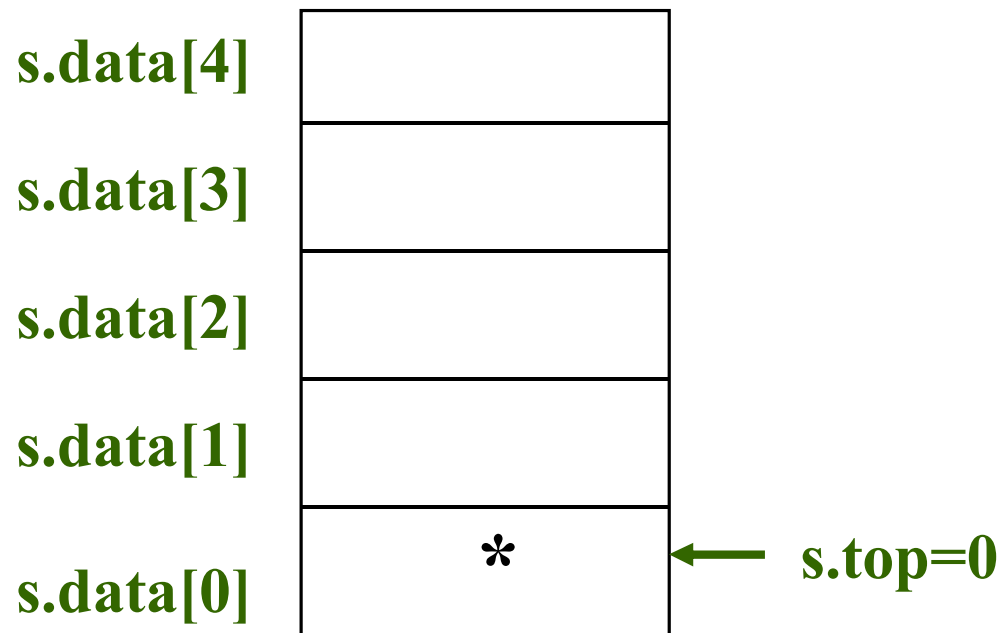
求逆波兰式——*a*

- $a * b + c - d / e \#$
- 构造一个栈保存相应的运算符



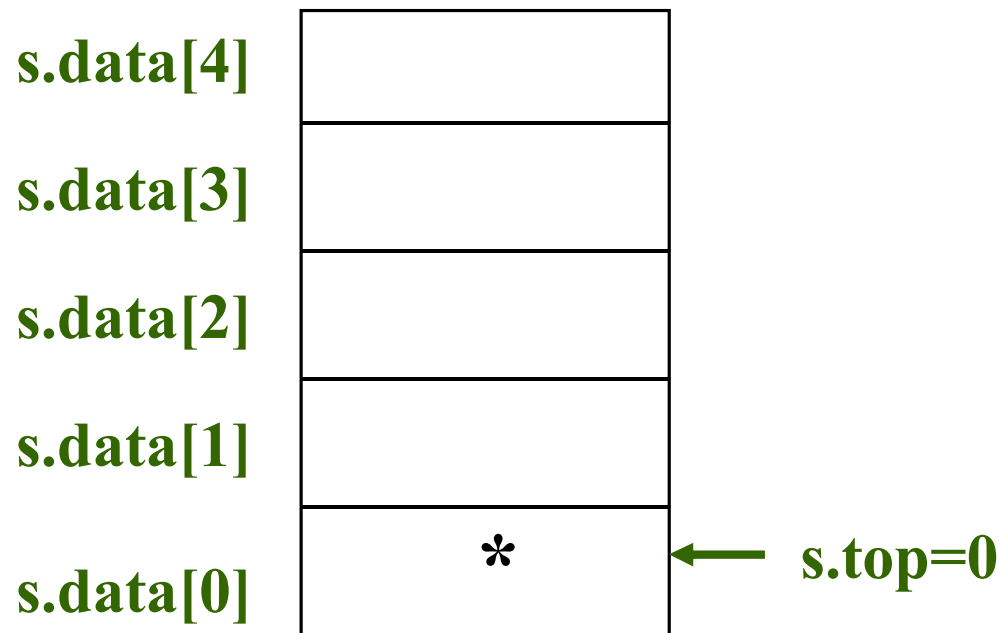
求逆波兰式— ab

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



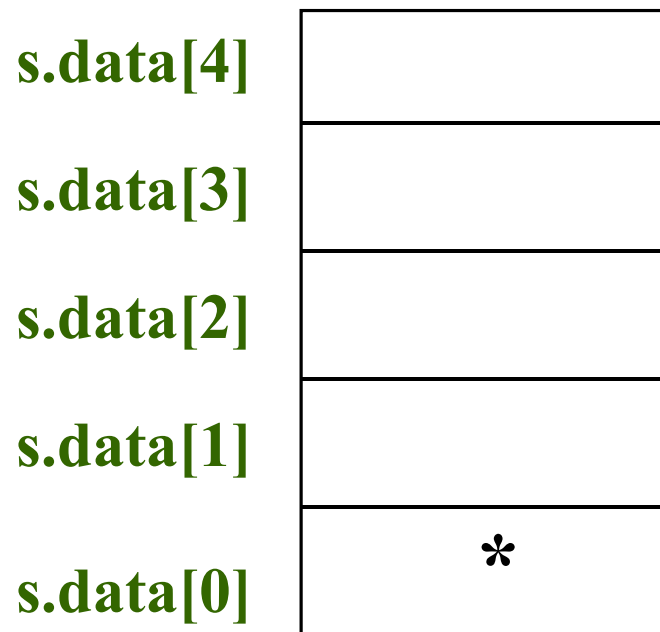
求逆波兰式— ab

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



求逆波兰式—— ab^*

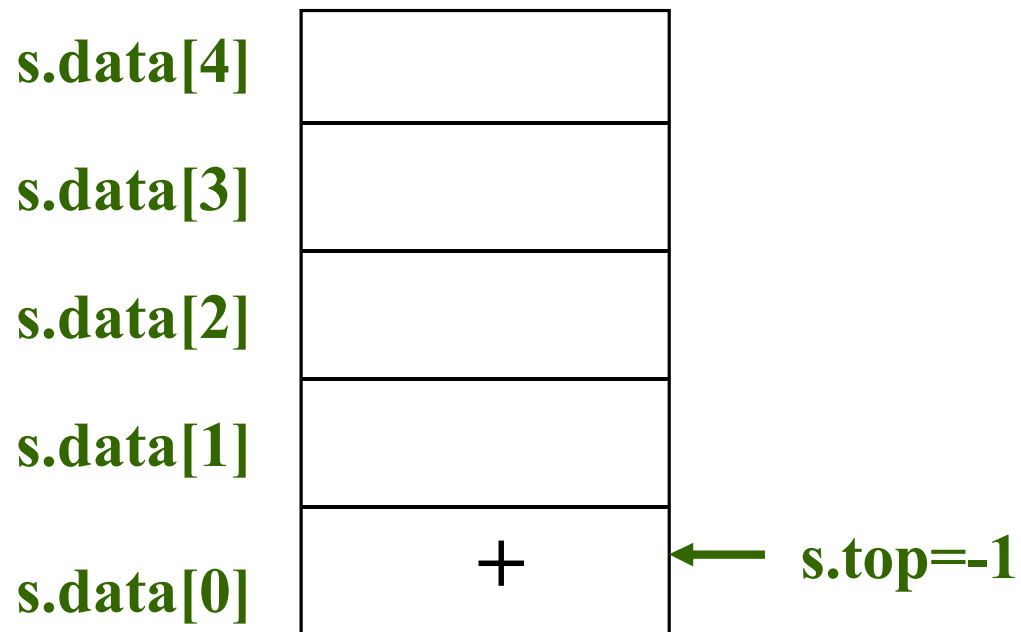
- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



← s.top=-1

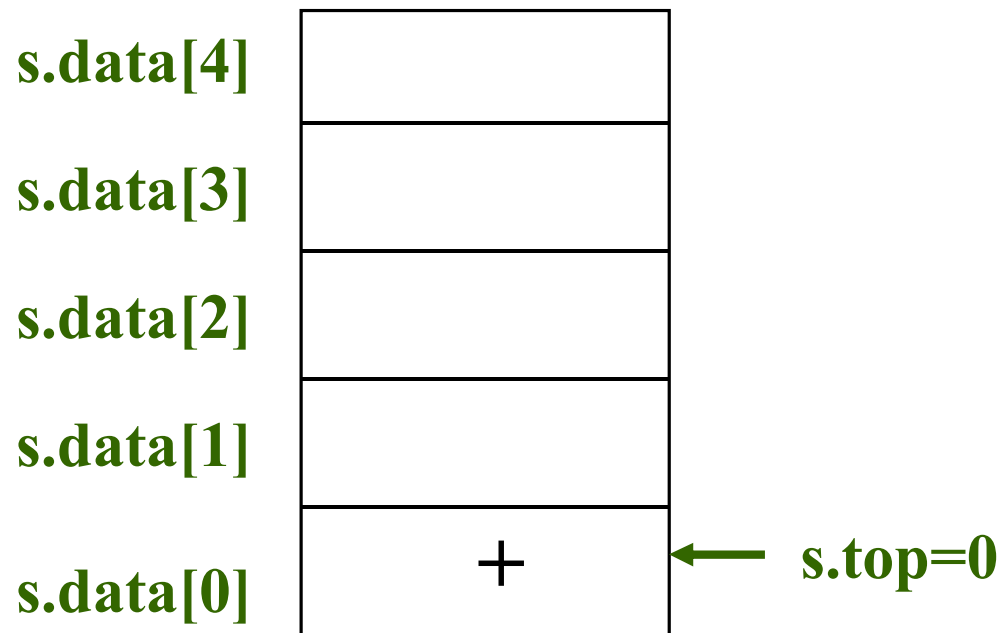
求逆波兰式—— ab^*

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



求逆波兰式—— ab^*

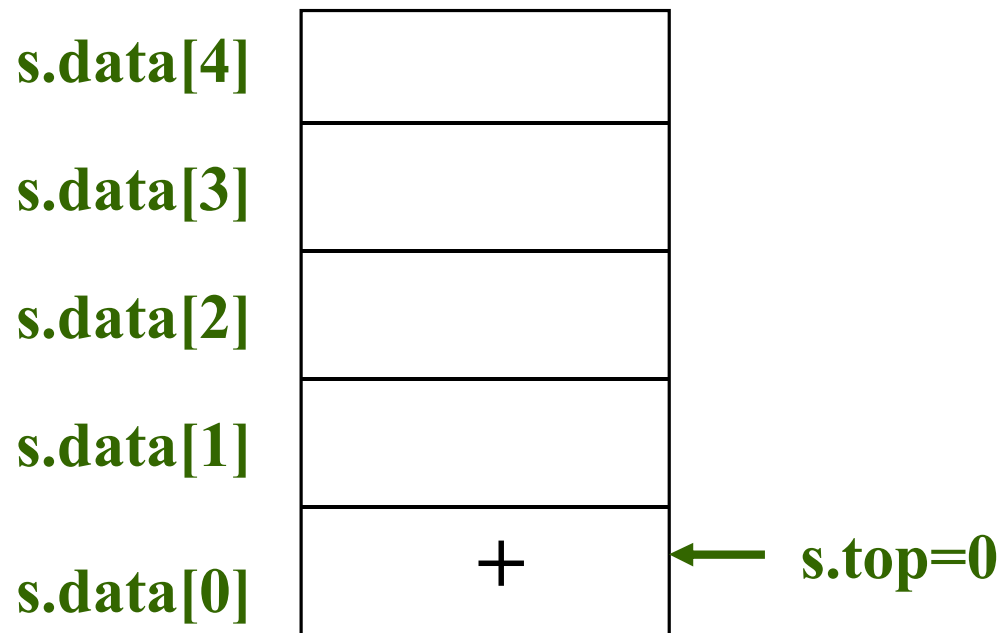
- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符





求逆波兰式—— $ab*c$

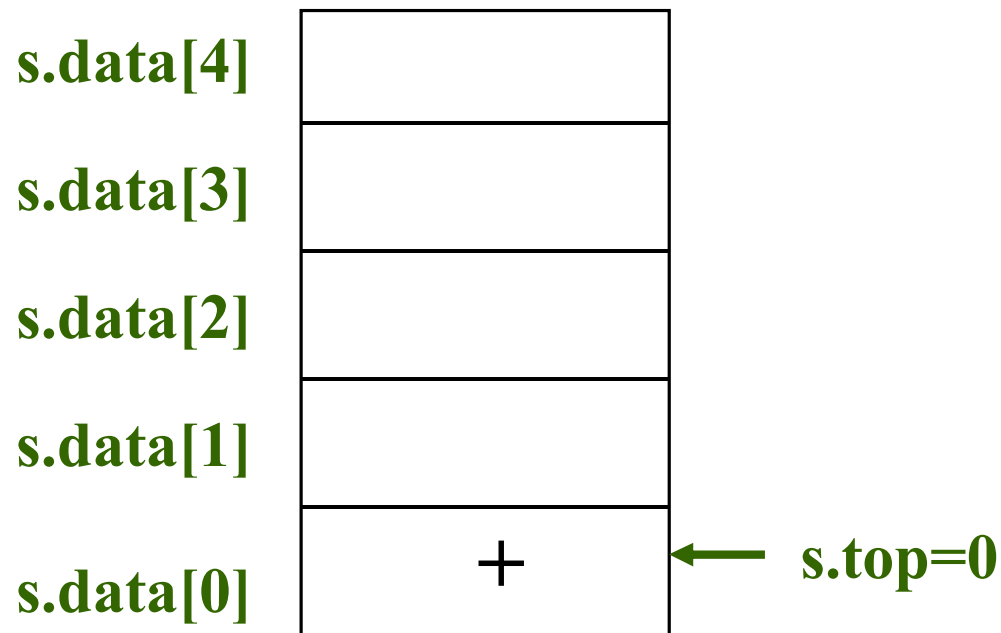
- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符





求逆波兰式—— $ab*c$

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



求逆波兰式—— $ab*c+$

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符

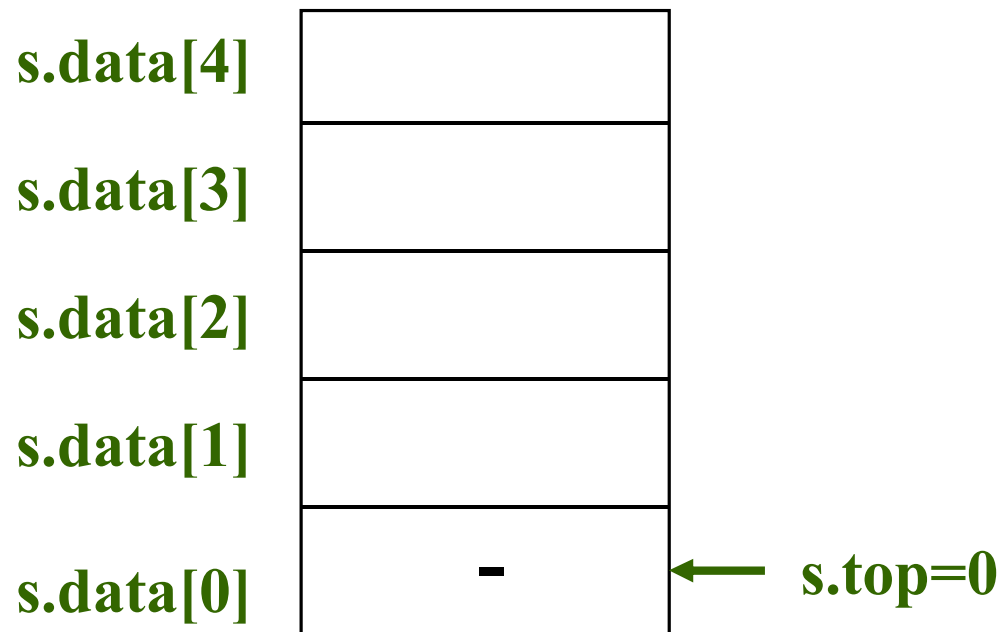
$s.data[4]$	
$s.data[3]$	
$s.data[2]$	
$s.data[1]$	
$s.data[0]$	+

← $s.top = -1$



求逆波兰式—— $ab*c+$

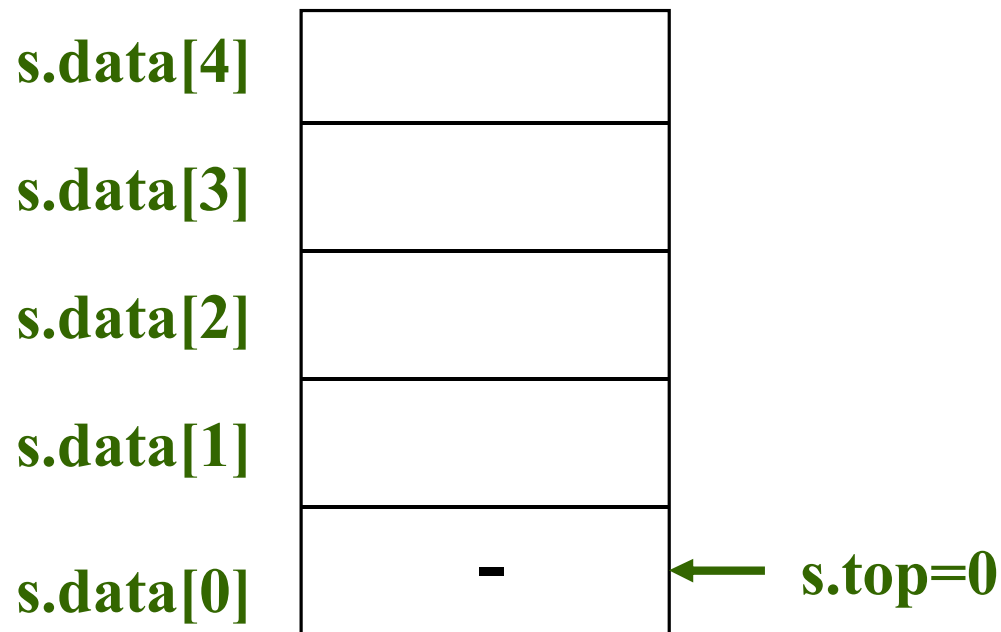
- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符





求逆波兰式—— $ab*c+$

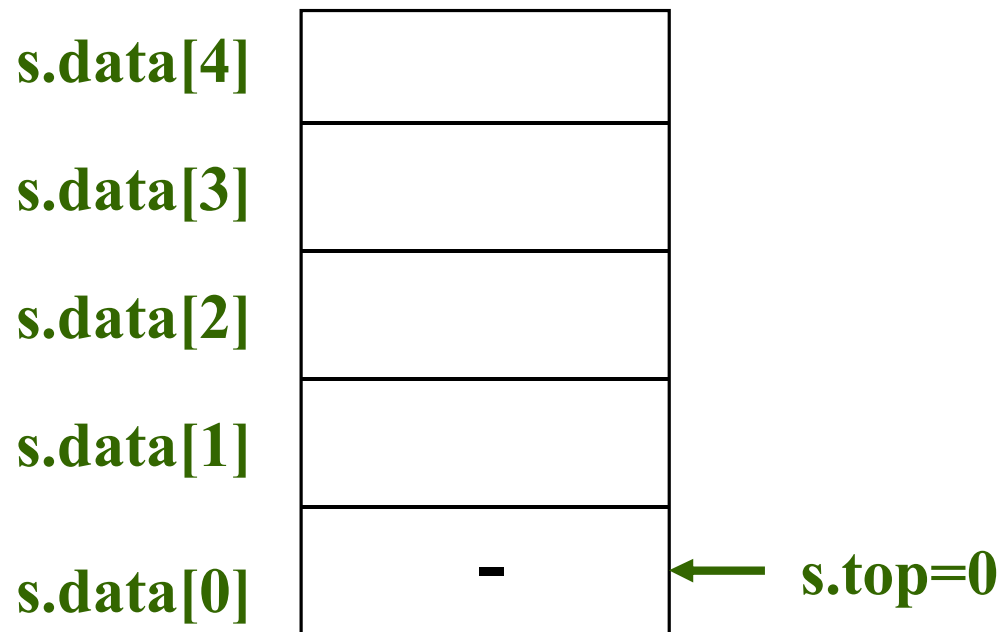
- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符





求逆波兰式—— $ab*c+d$

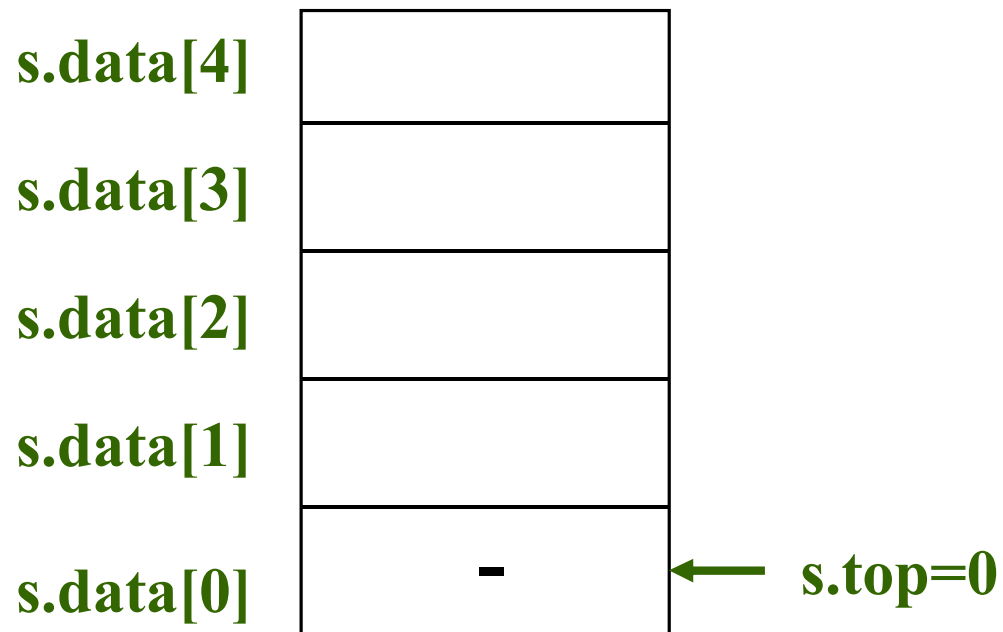
- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符





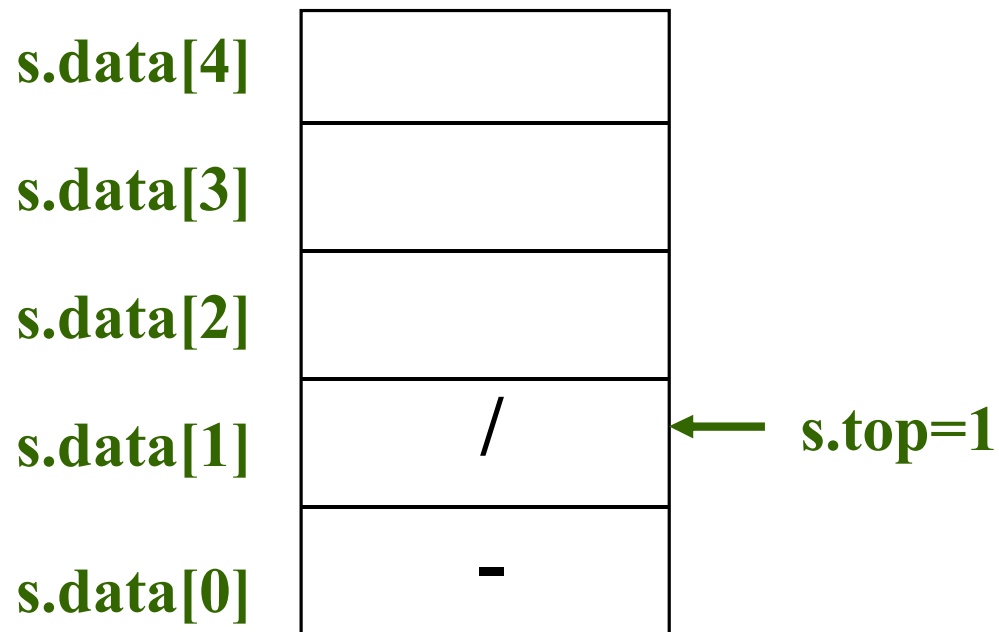
求逆波兰式—— $ab*c+d$

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



求逆波兰式—— $ab*c+d$

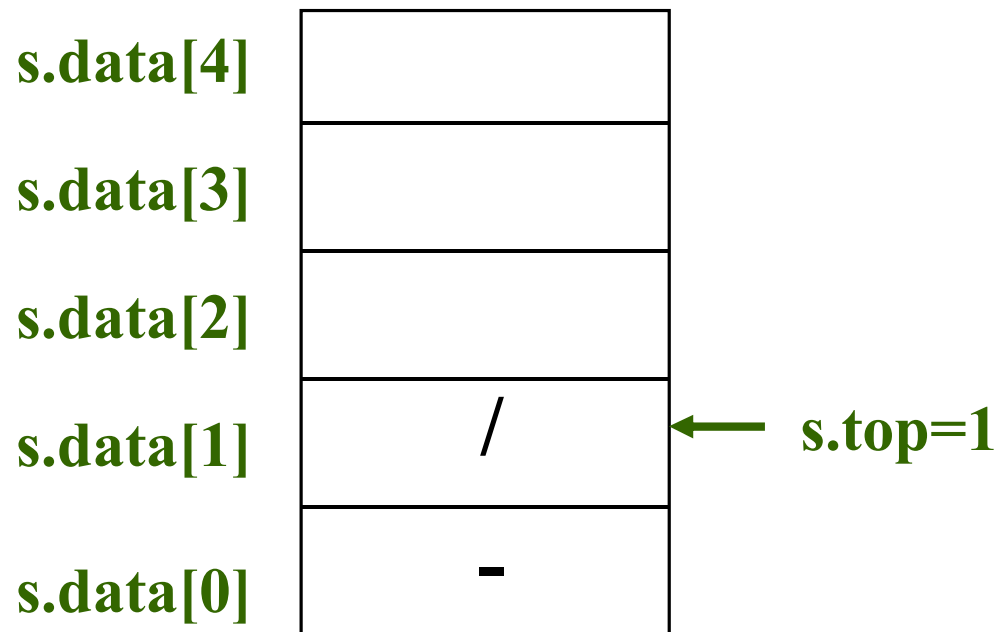
- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符





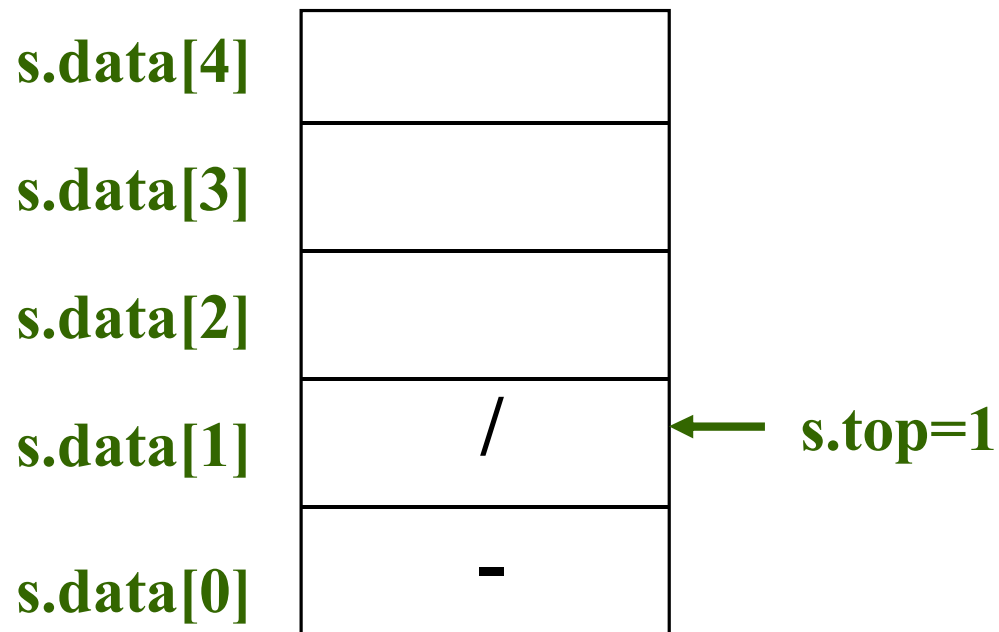
求逆波兰式—— $ab*c+d$

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



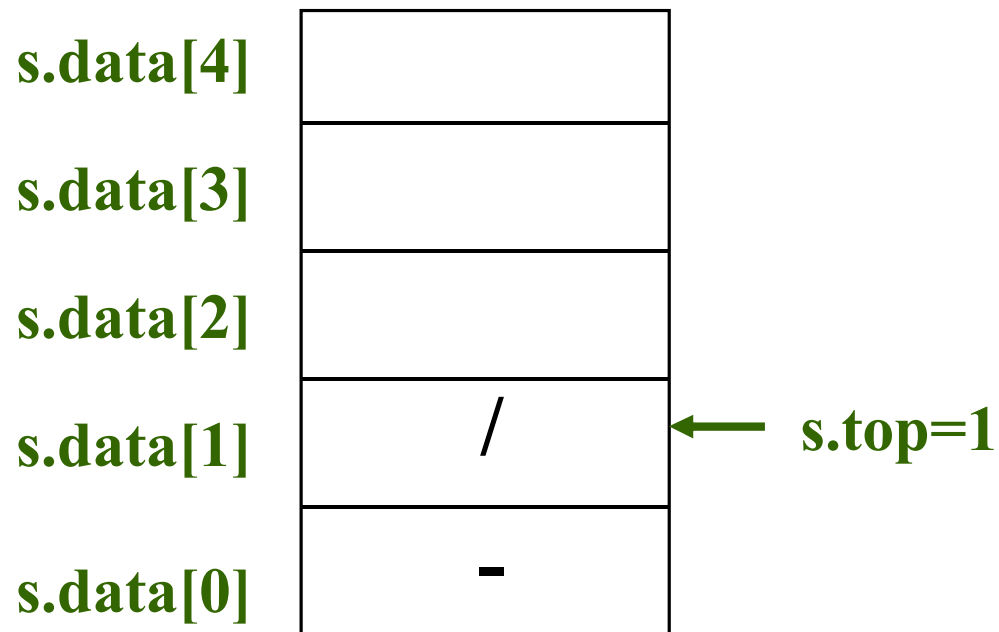
求逆波兰式—— $ab*c+de$

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



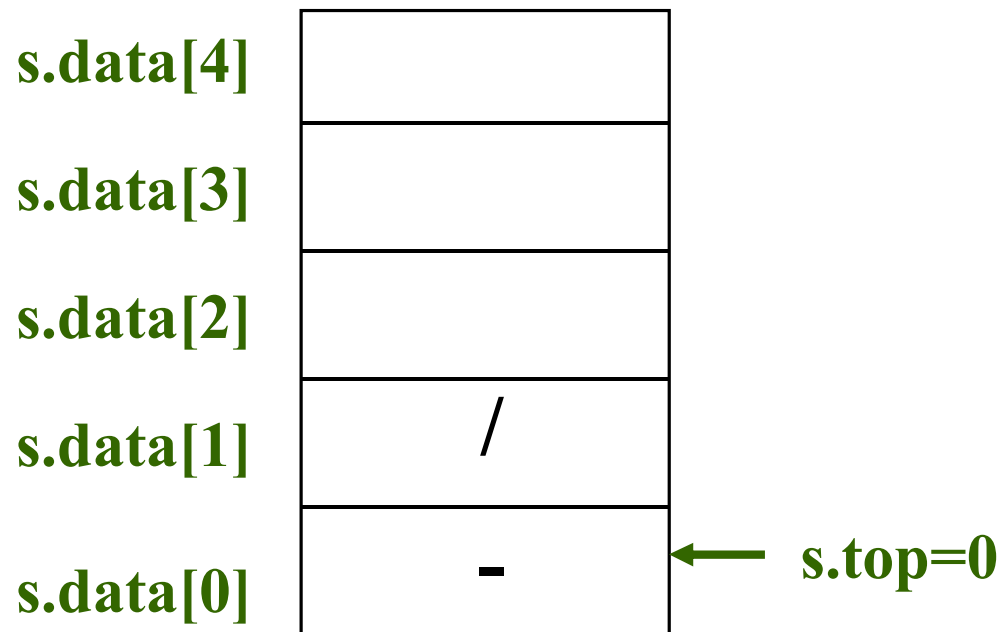
求逆波兰式—— $ab*c+de$

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



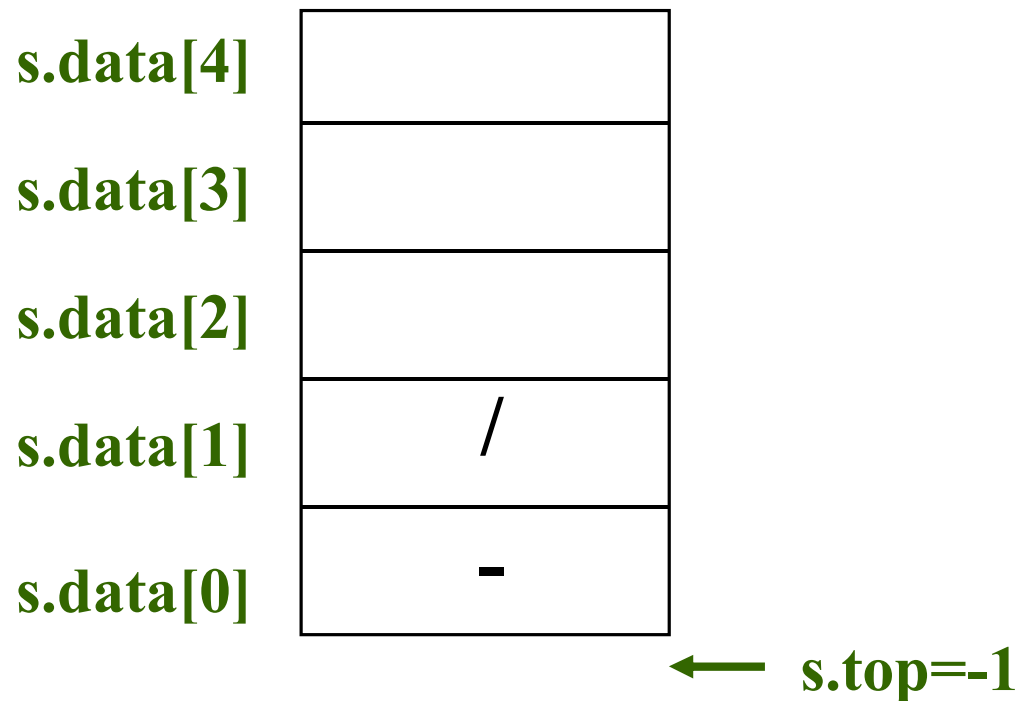
求逆波兰式—— $ab*c+de/$

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



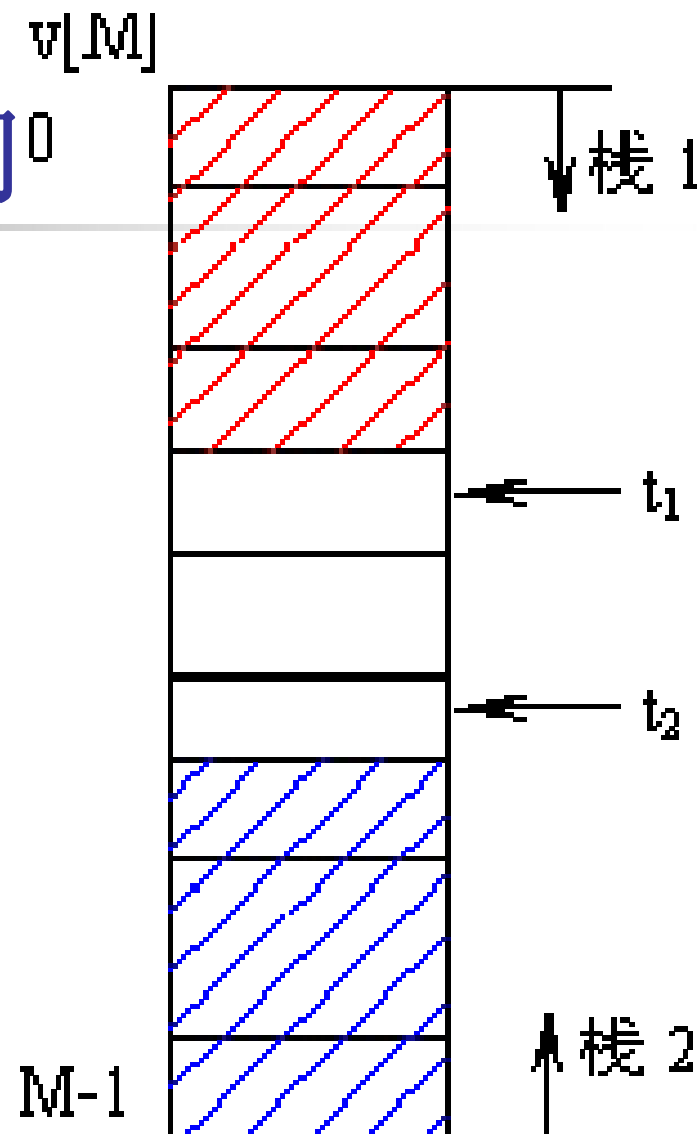
求逆波兰式—— $ab*c+de/-$

- $a*b+c-d/e\#$
- 构造一个栈保存相应的运算符



双栈共享存储空间

- 两个栈共用一个一维数组 $v[M]$, 栈底分别设在数组的**两端**, 各自向中间伸展, 第一个栈自顶向下伸展, 第二个栈自底向上伸展。两个栈共享存储空间, 可互补空缺, 使得某个栈实际可利用的空间大于 $M/2$



入队

队尾

队列

----线性结构，插入和删除操作受限制

- 限制插入在表一端（表尾）进行，而删除在表的另一端（表头）进行。

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

队首

出队

插入顺序



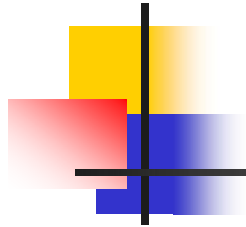
删除顺序

特点—先进先出



练习

- 三个数据按照1, 2, 3的顺序通过队的运算, 给出所有可能的出队序列。
- 主要操作:
- 入队、出队、初始化空队列、求队长
- 实现: 顺序队, 设置队头、队尾两个指针



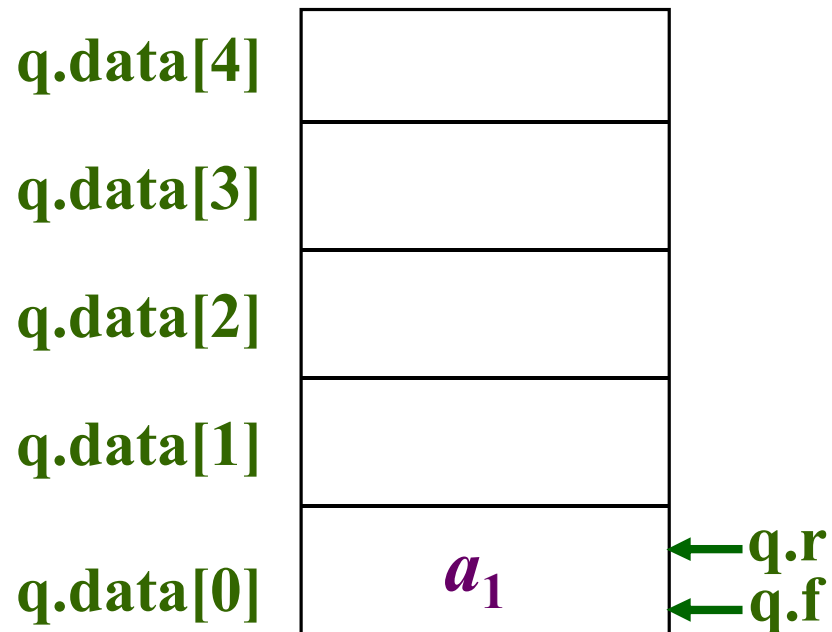
顺序队

```
define MAXSIZE 5  
typedef struct  
    {datatype data[MAXSIZE];  
        int r,f;  
    }SeQueue;  
SeQueue q;
```

初始化: $q.f=0; q.r=0;$

入队: 只要有空间
 $q.data[q.r]=x; q.r++;$

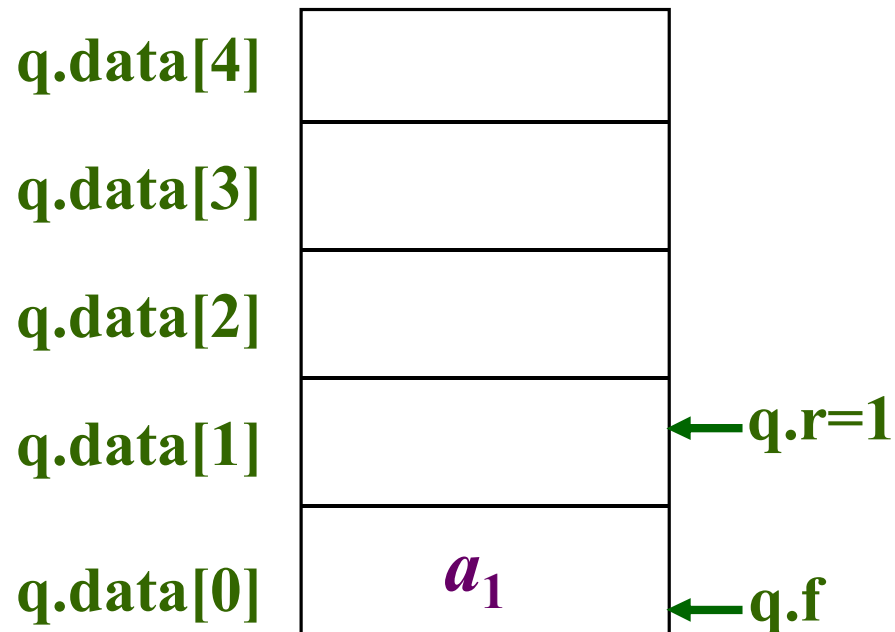
顺序队



初始化: $q.f=0; q.r=0;$

入队: 只要有空间
 $q.data[q.r]=x; q.r++;$

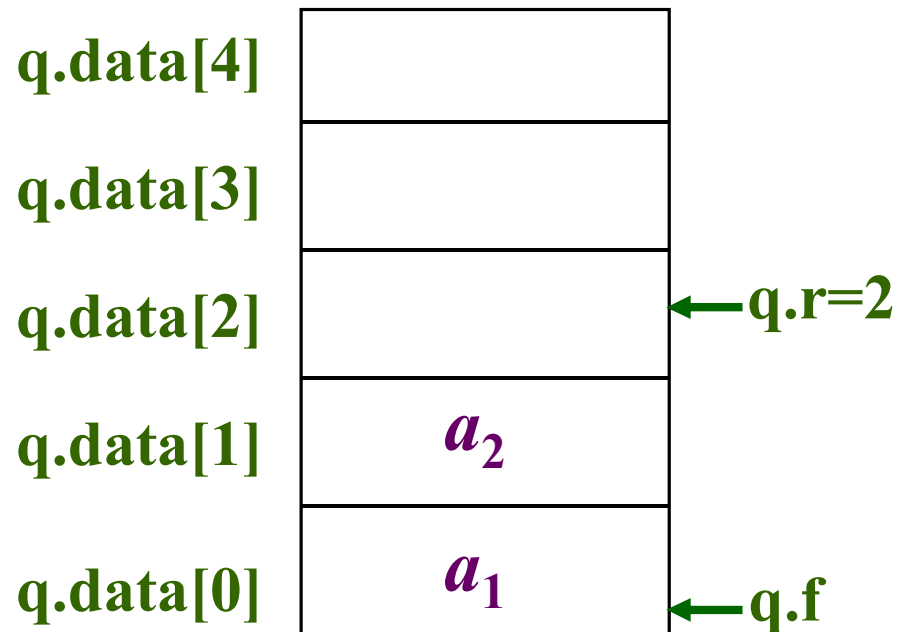
顺序队



初始化: $q.f=0; q.r=0;$

入队: 只要有空间
 $q.data[q.r]=x; q.r++;$

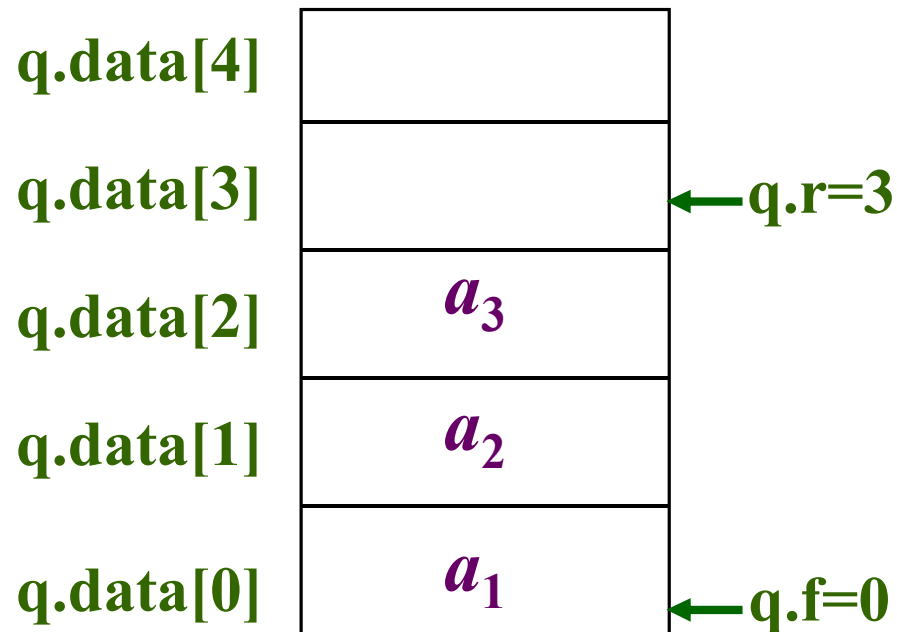
顺序队

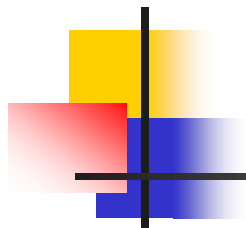


初始化: $q.f=0; q.r=0;$

入队: 只要有空间
 $q.data[q.r]=x; q.r++;$

顺序队

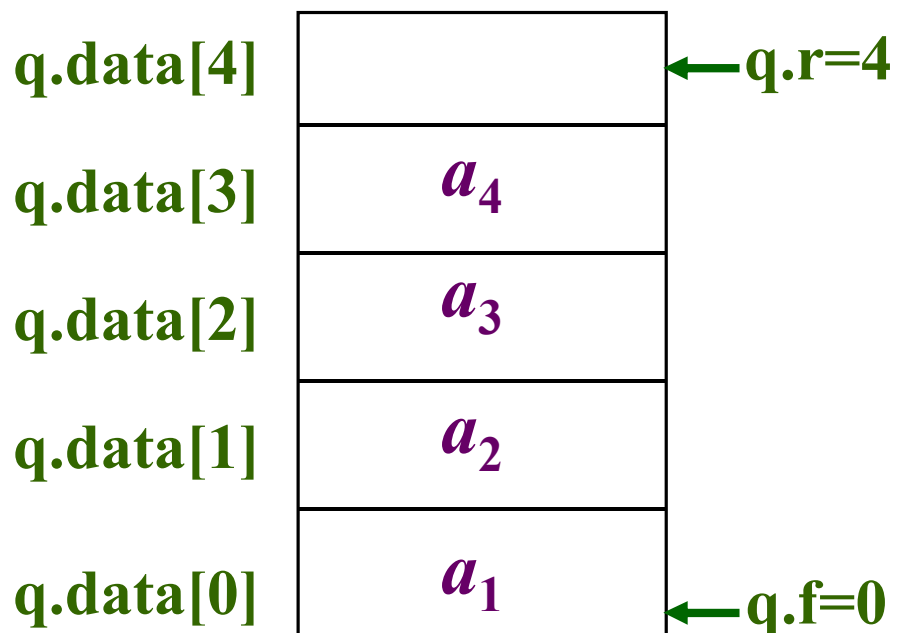


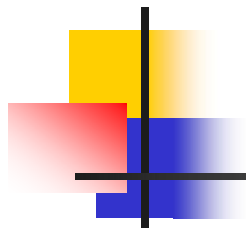


顺序队

初始化: $q.f=0; q.r=0;$

入队: 只要有空间
 $q.data[q.r]=x; q.r++;$





顺序队

初始化: $q.f=0; q.r=0;$

入队: 只要有空间 $q.r < \text{MAXSIZE}$
 $q.data[q.r]=x; q.r++;$

← $q.r=5$

$q.data[4]$	a_5
$q.data[3]$	a_4
$q.data[2]$	a_3
$q.data[1]$	a_2
$q.data[0]$	a_1

再插入, 无空间, 溢出

← $q.f=0$



顺序队

- 初始化: `q.f=0;q.r=0;`
- 入队: **if (`q.r<MAXSIZE`)**
 {q.data[q.r]=x;q.r++;}
 else printf(“overflow”);



顺序队

- 初始化: `q.f=0;q.r=0;`
- 入队: `if (q.r<MAXSIZE)`
 `{q.data[q.r]=x;q.r++;}`
 `else printf(“overflow”);`
- 出队: `if (队不空)`
 `q.f++;`



顺序队 — 出队

← q.r=5

q.data[4]	a_5
q.data[3]	a_4
q.data[2]	a_3
q.data[1]	a_2
q.data[0]	a_1

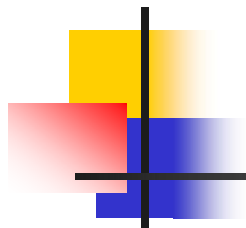
← q.f=0



顺序队 — 出队

← q.r=5

q.data[4]	a_5	
q.data[3]	a_4	
q.data[2]	a_3	
q.data[1]	a_2	← q.f=1
q.data[0]	a_1	



顺序队 — 出队

← q.r=5

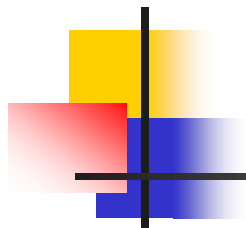
q.data[4]	a_5	
q.data[3]	a_4	
q.data[2]	a_3	← q.f=2
q.data[1]	a_2	
q.data[0]	a_1	



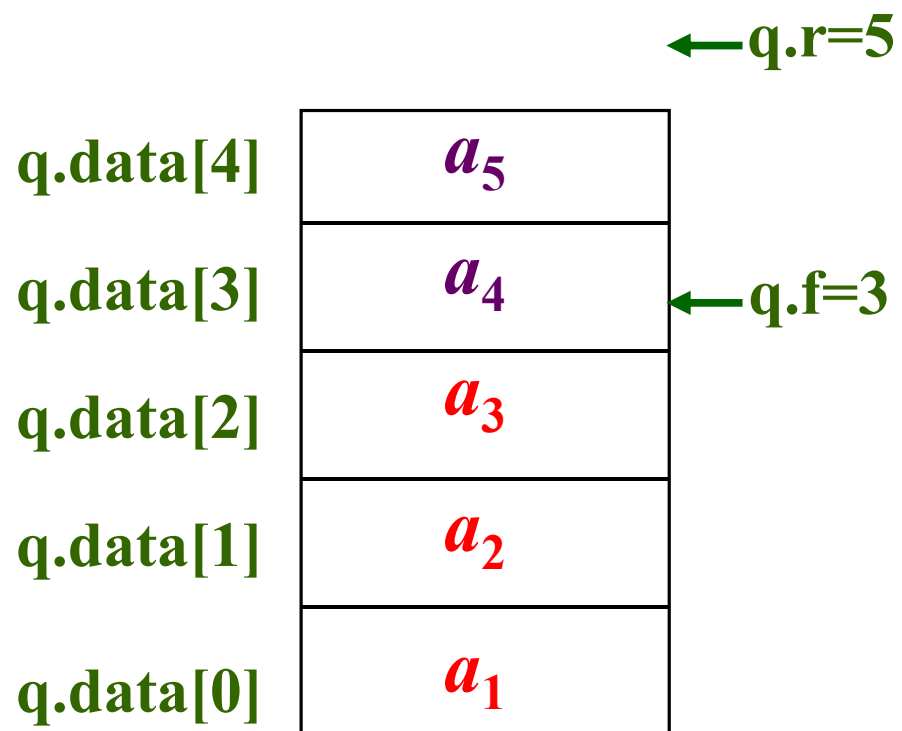
顺序队 —出队—判队空

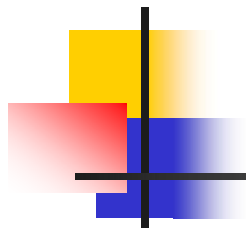
← q.r=5

q.data[4]	a_5	
q.data[3]	a_4	
q.data[2]	a_3	← q.f=2
q.data[1]	a_2	
q.data[0]	a_1	



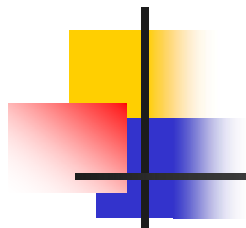
顺序队 — 出队



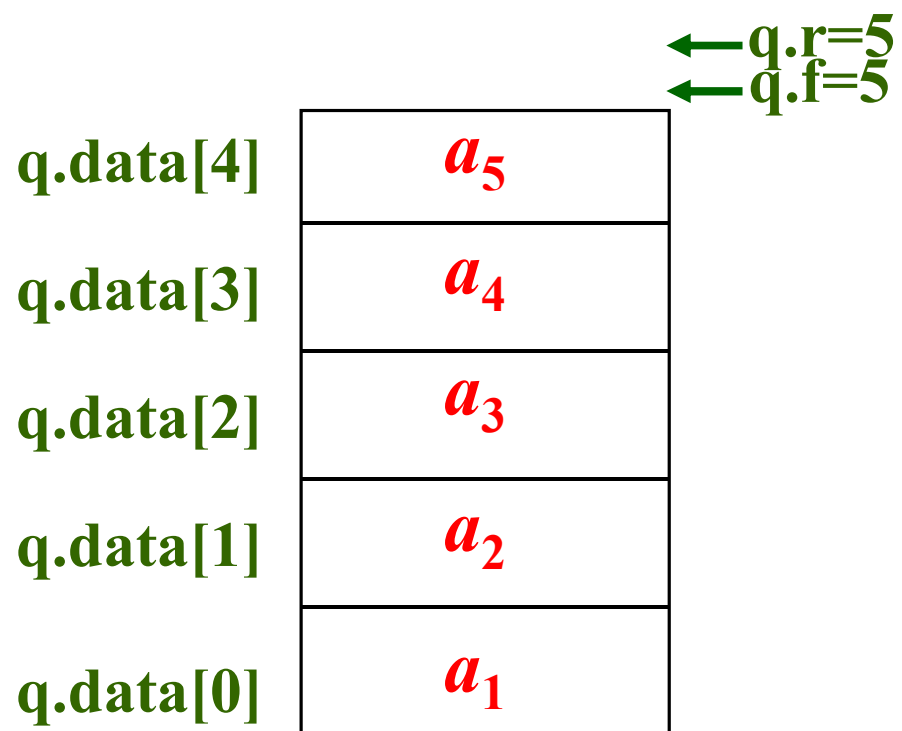


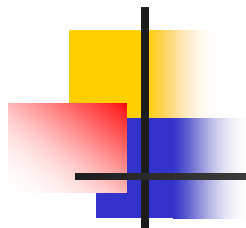
顺序队 — 出队

		← q.r=5
q.data[4]	a_5	← q.f=4
q.data[3]	a_4	
q.data[2]	a_3	
q.data[1]	a_2	
q.data[0]	a_1	

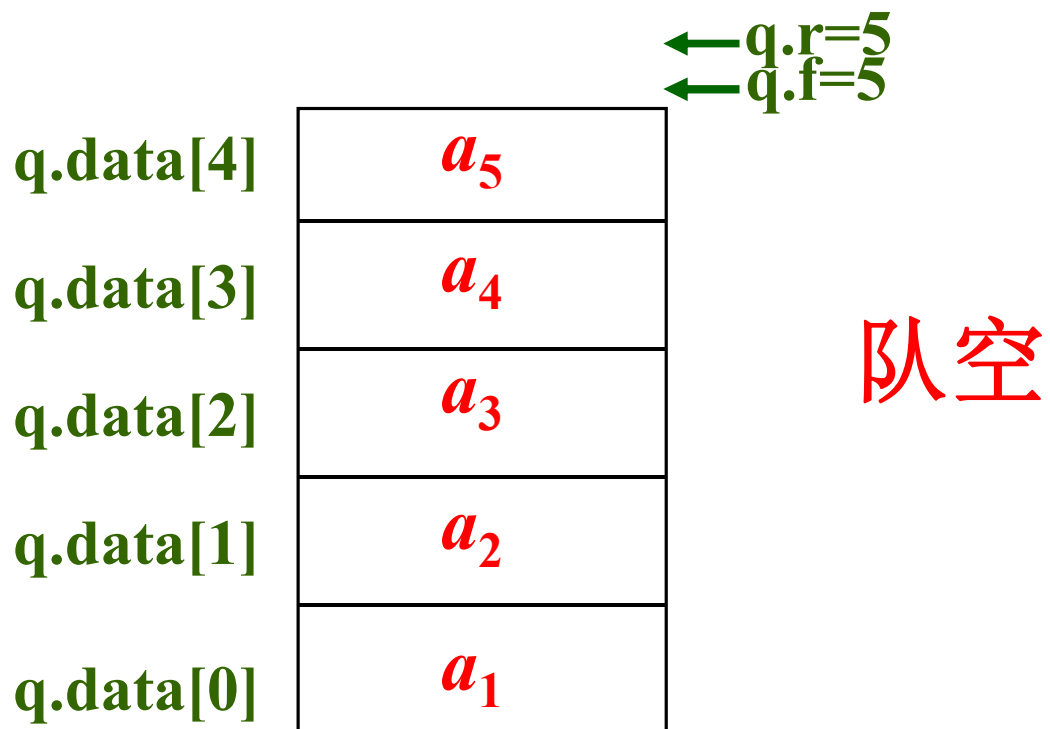


顺序队 — 出队





顺序队 —出队—出队





顺序队

- 初始化: `q.f=0;q.r=0;`
- 入队: `if (q.r<MAXSIZE)`
 `{q.data[q.r]=x;q.r++;}`
 `else printf(“overflow”);`
- 出队: `if(q.f!=q.r)//if(q.f<q.r)`
 `q.f++;`

顺序队 — 入队 $x=a_6$


	$\leftarrow q.r=5$ $\leftarrow q.f=5$
q.data[4]	a_5
q.data[3]	a_4
q.data[2]	a_3
q.data[1]	a_2
q.data[0]	a_1

此时队空，但是再次
插入数据元素却判断
没有空间——假溢出

顺序队 — 假溢出

◆ 顺序队列存在假溢出

◆ 解决方法

➤ 方法1：每删除一个数据元素，余下的所有数据元素顺次移动一个位置 

➤ 方法2：循环队列，将顺序队列data[0..
MAXSIZE - 1]看头尾相接的循环结构

初始化: $q.f=0; q.r=0;$

循环队列

将队列的数据区 $data[0..MAXSIZE-1]$ 看成头尾相接的循环结构

$q.data[4]$

$q.data[3]$

$q.data[2]$

$q.data[1]$

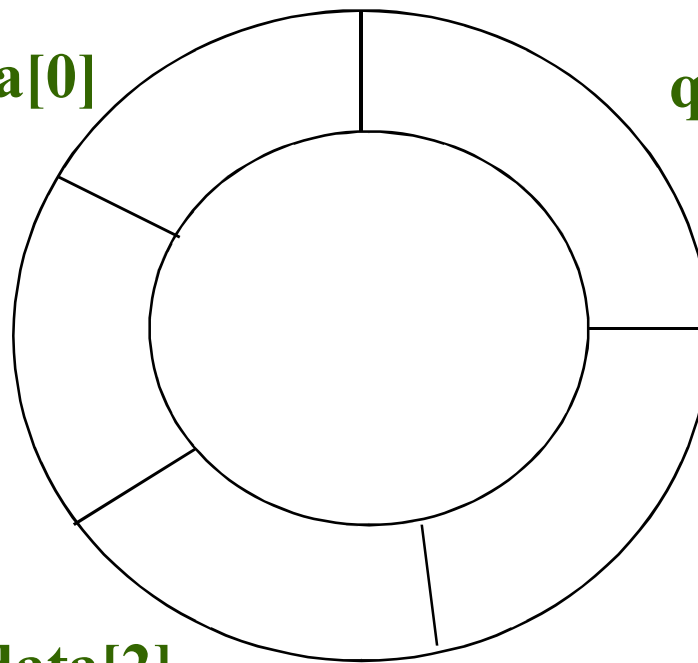
$q.data[0]$



$q.data[0]$

$q.data[1]$

$q.data[2]$



$q.data[4]$

$q.data[3]$



循环队列

- 初始化: `q.f=0;q.r=0;`
- 入队: `if (有空间)`
`{ q.data[q.r]=x;q.r++;}`
`else printf("overflow");`



循环队列

- 初始化: $q.f=0; q.r=0;$
- 入队: if (有空间)
 $\{q.data[q.r]=x; q.r=(q.r+1)\%MAXSIZE;\}$
 else printf(“overflow”);



循环队列

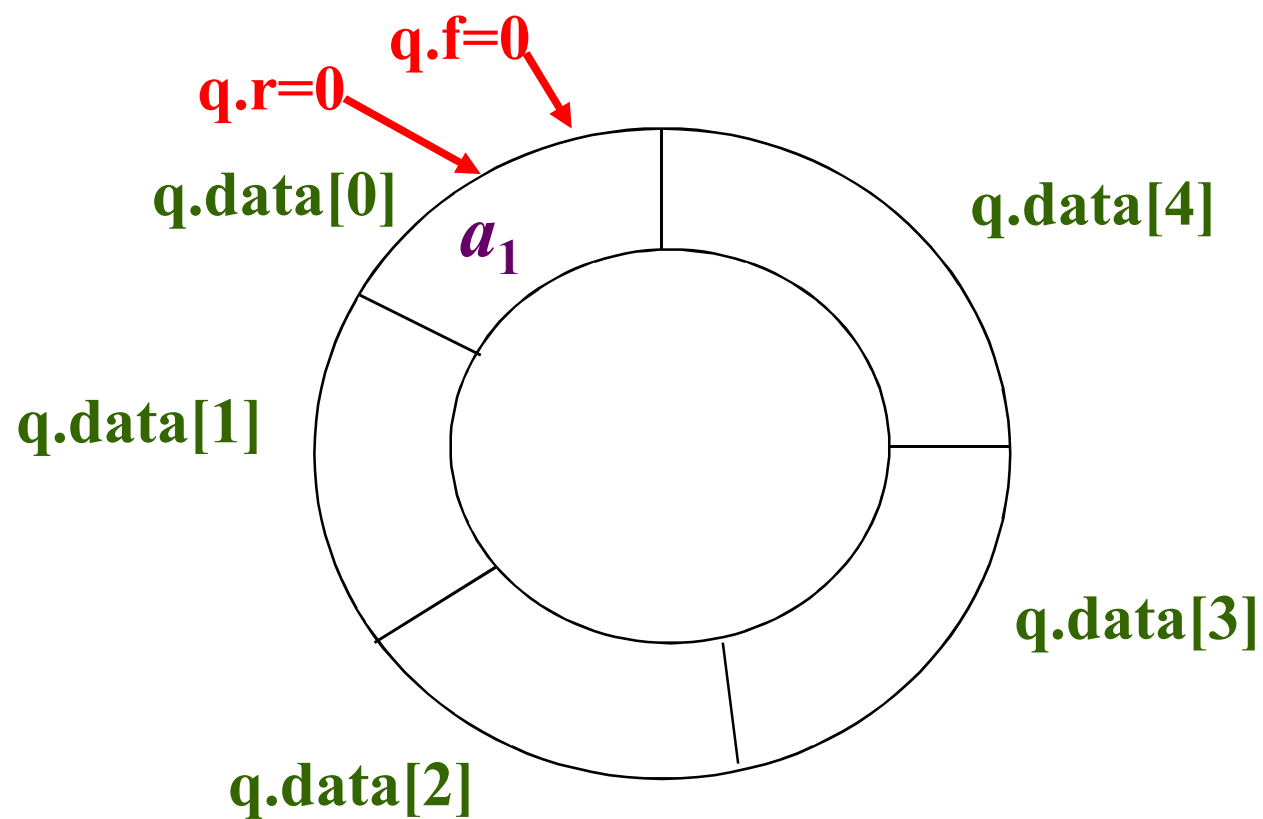
- 初始化: $q.f=0; q.r=0;$
- 入队: if (有空间)
 $\{q.data[q.r]=x; q.r=(q.r+1)\%MAXSIZE;\}$
 $\text{else printf(“overflow”);}$
- 出队: if (队不空)
 $q.f++$



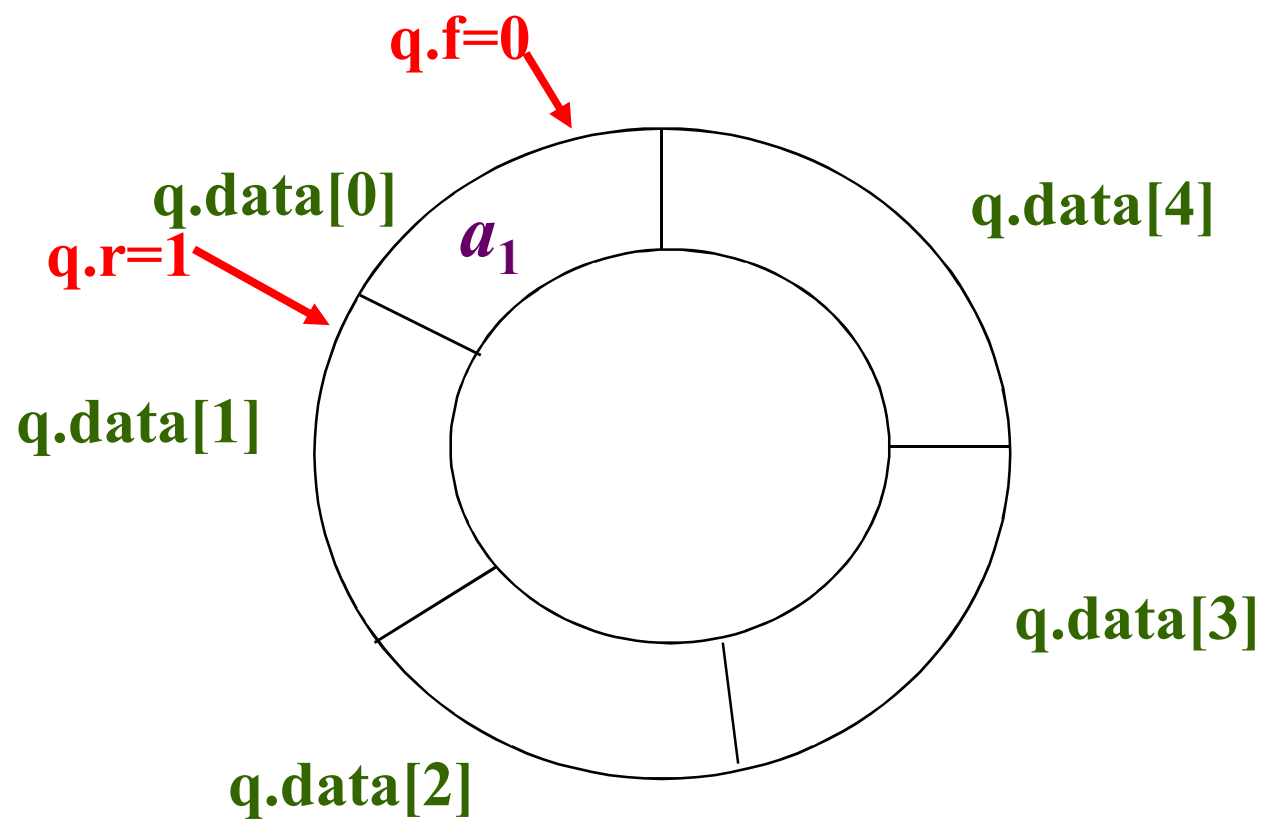
循环队列

- 初始化: $q.f=0; q.r=0;$
- 入队: if (有空间)
 $\{q.data[q.r]=x; q.r=(q.r+1)\%MAXSIZE;\}$
else printf(“overflow”);
- 出队: if (队不空)
 $q.f=(q.f+1)\%MAXSIZE;$

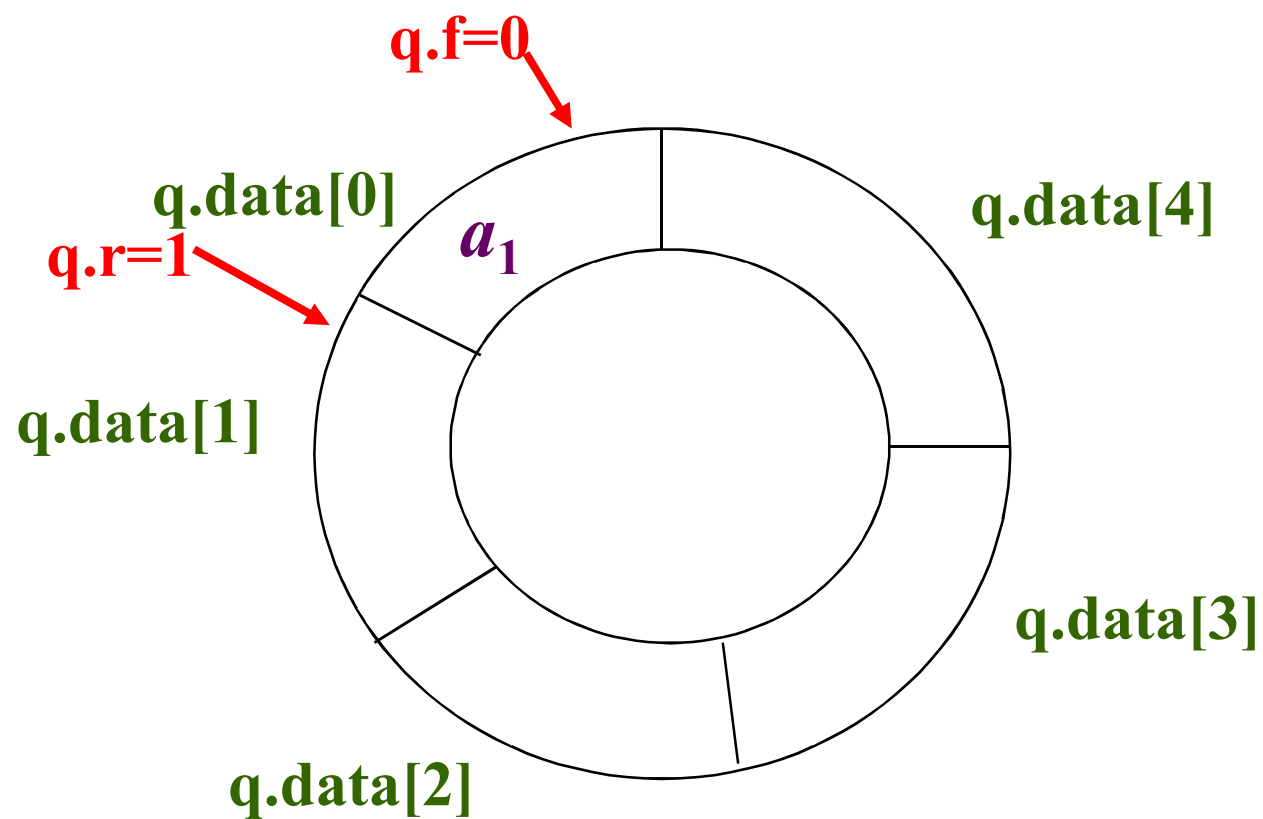
循环队列 -- 入队 $x=a_1$



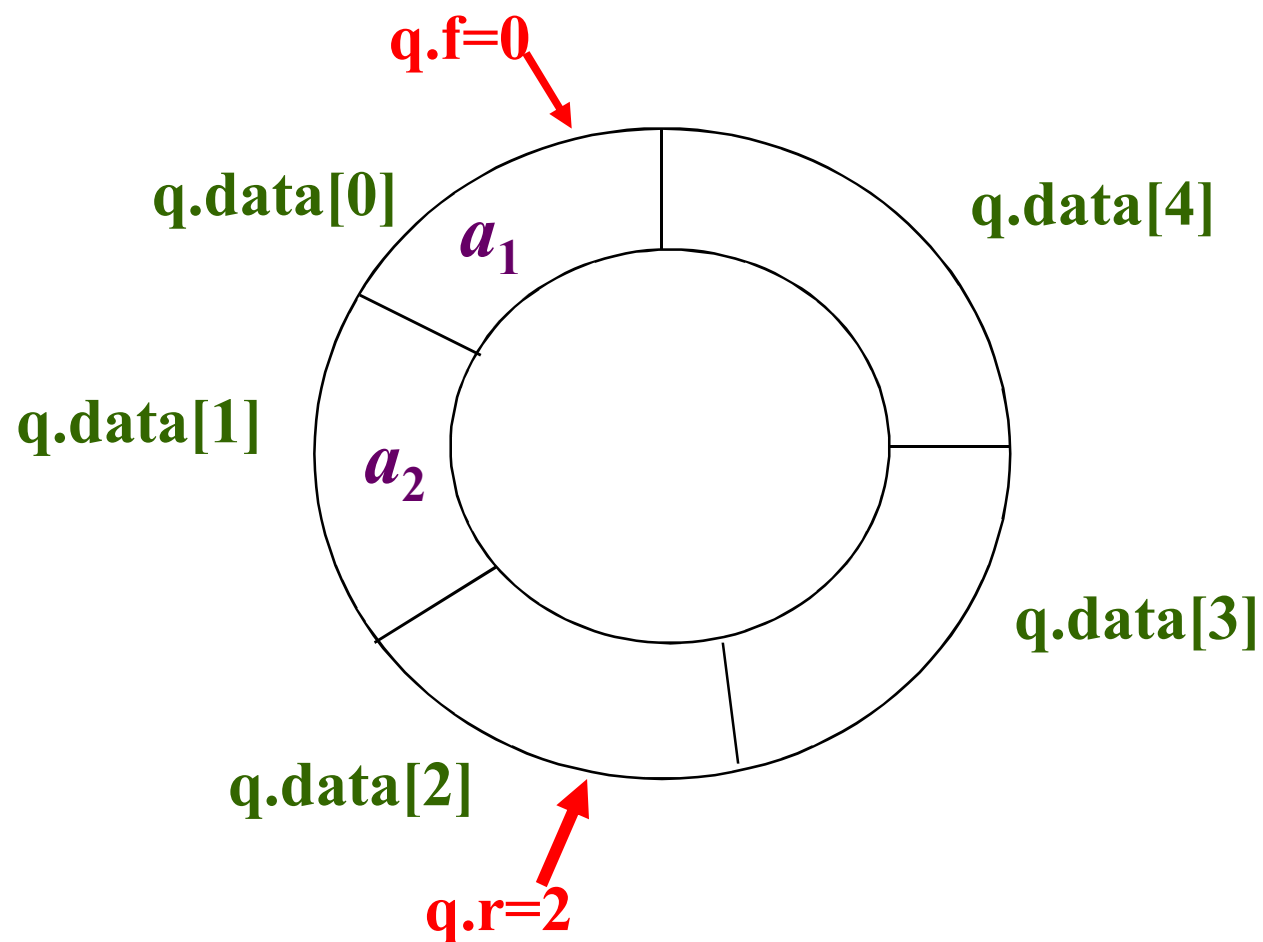
循环队列--入队 $x=a_1$



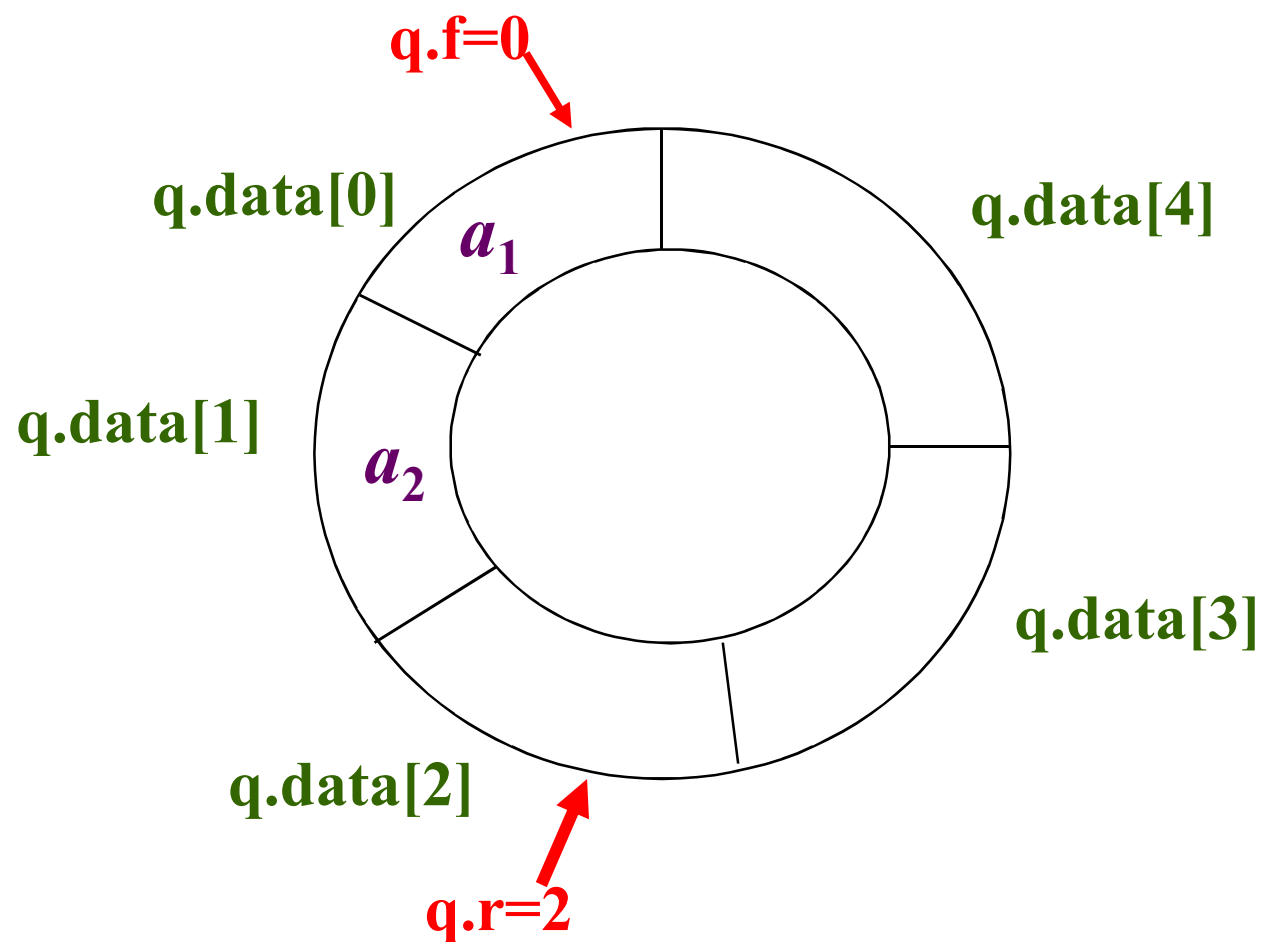
循环队列--入队 $x=a_2$



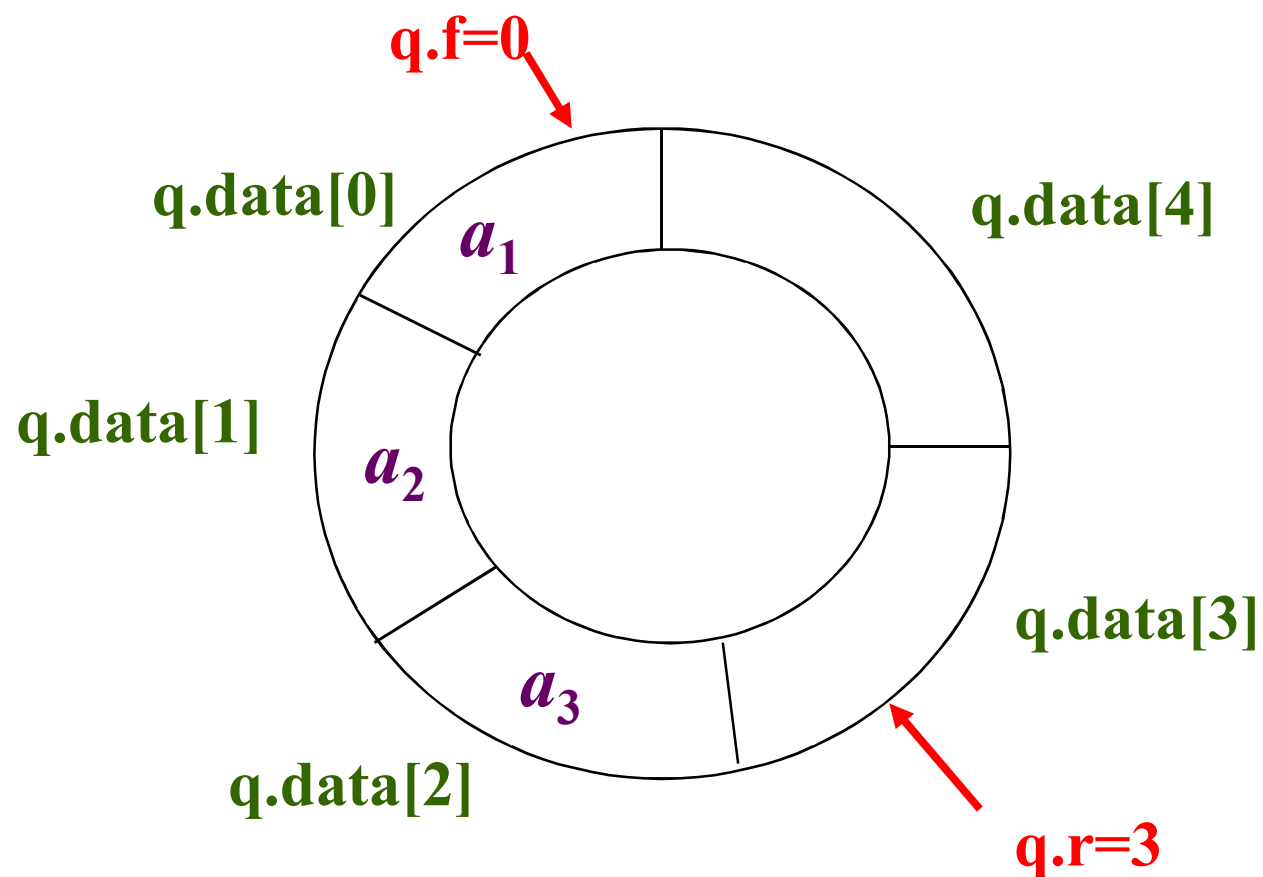
循环队列--入队 $x=a_2$



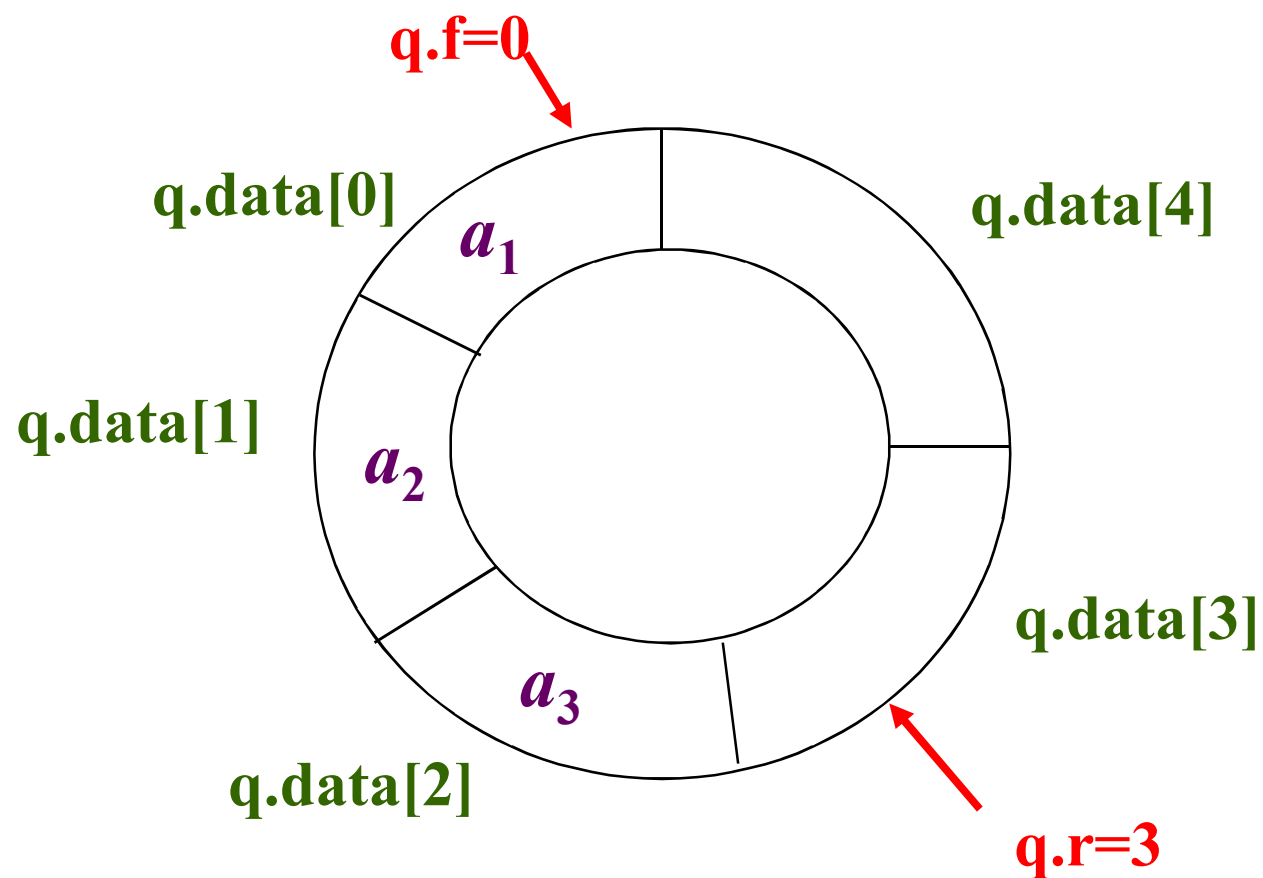
循环队列--入队 $x=a_3$



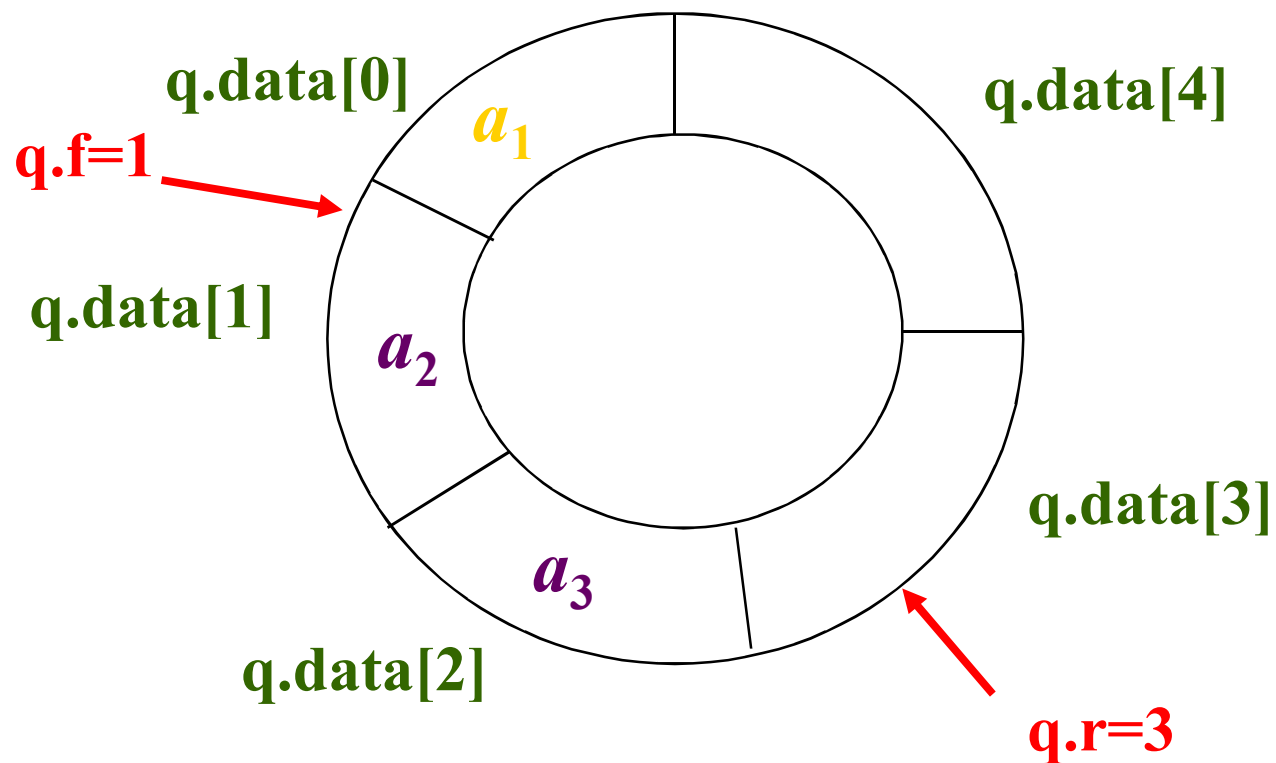
循环队列--入队 $x=a_3$



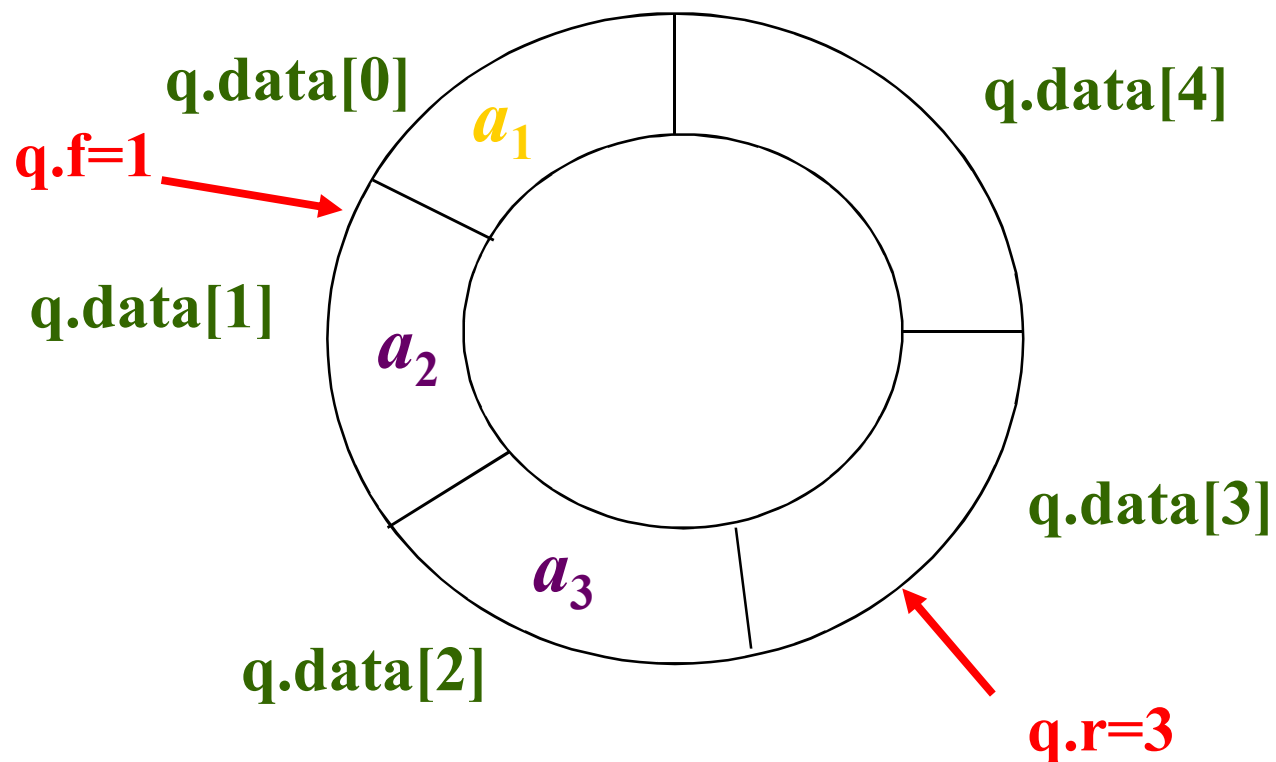
循环队列—出队



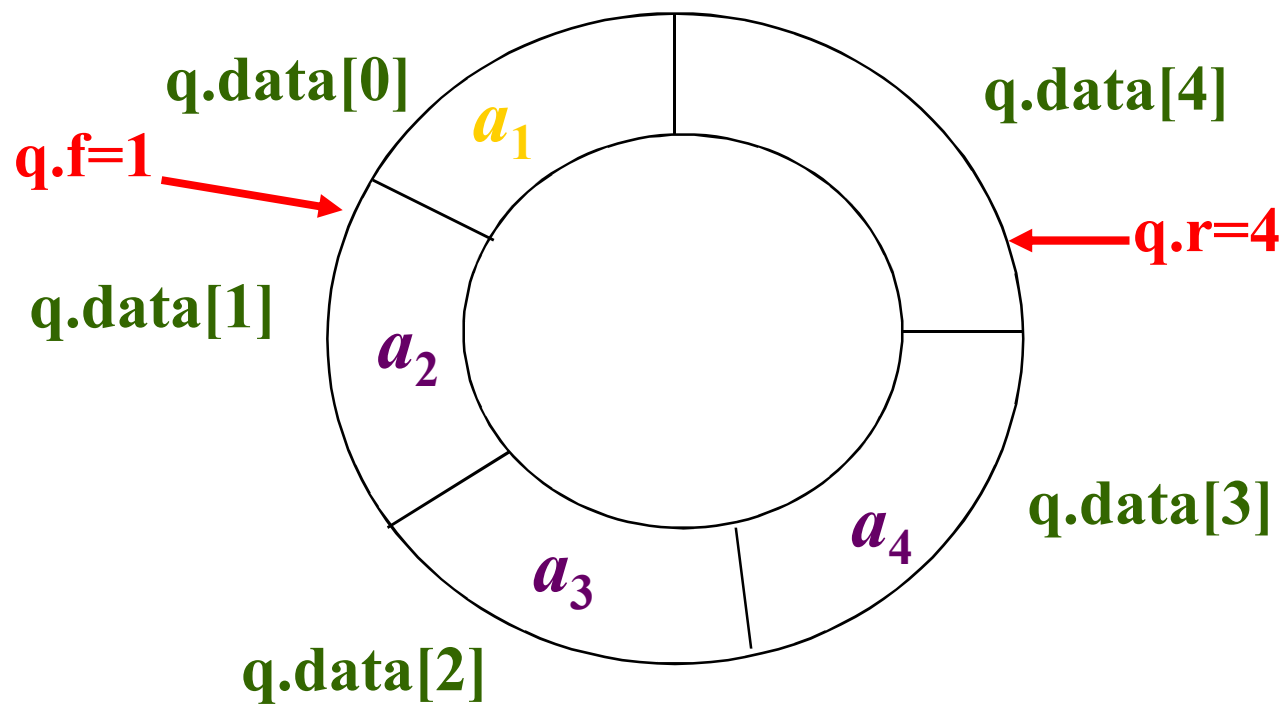
循环队列—出队



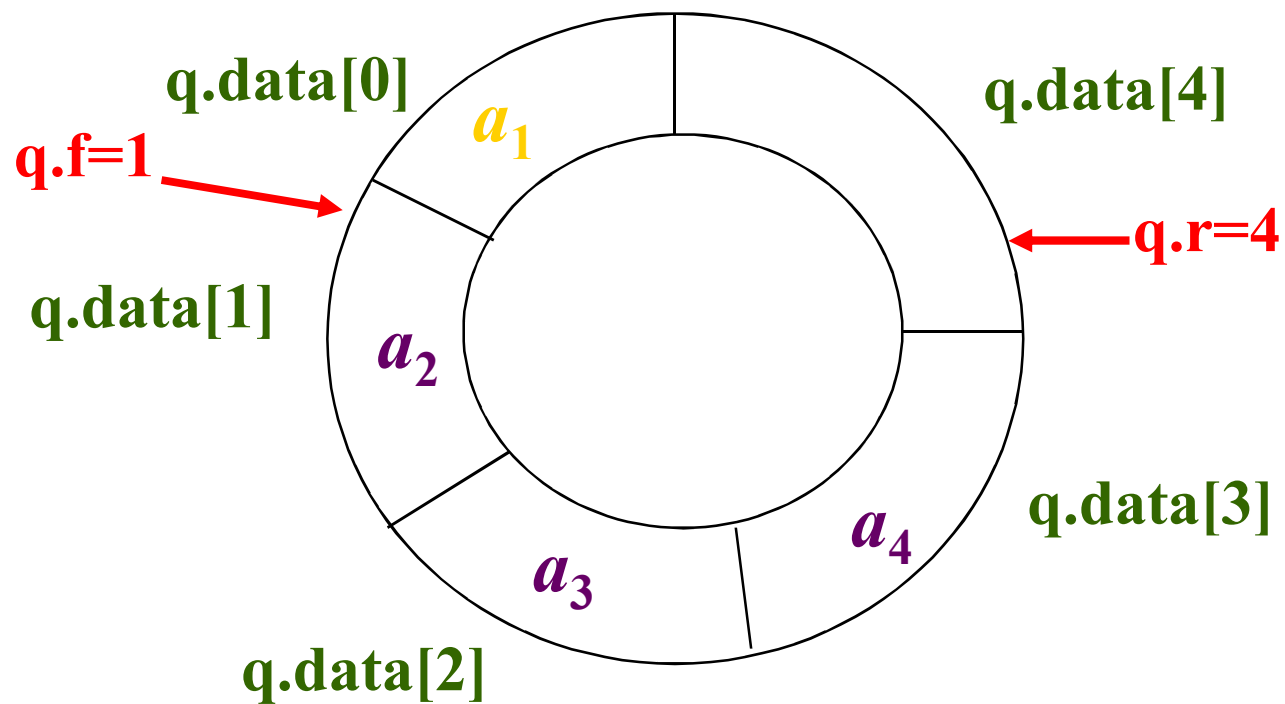
循环队列—入队 $x=a_4$



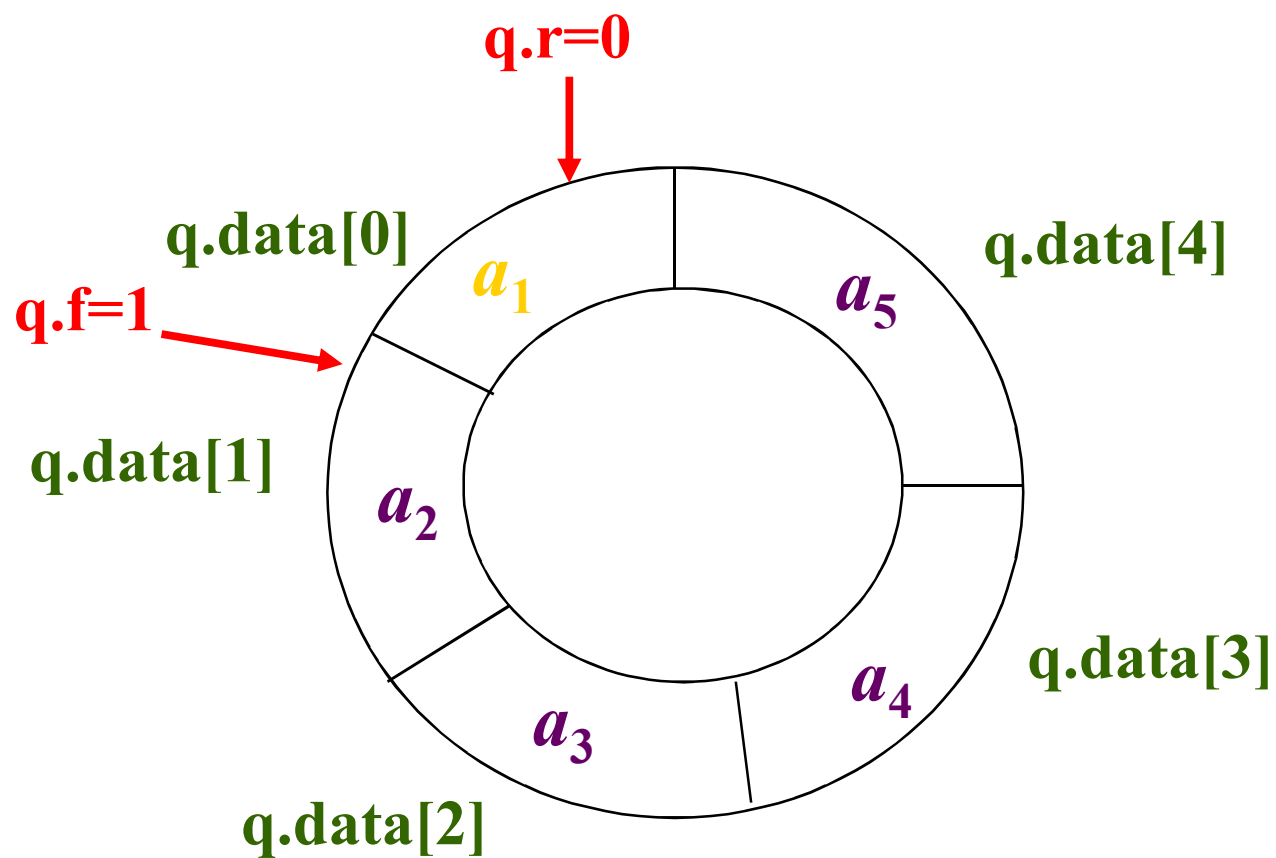
循环队列—入队 $x=a_4$



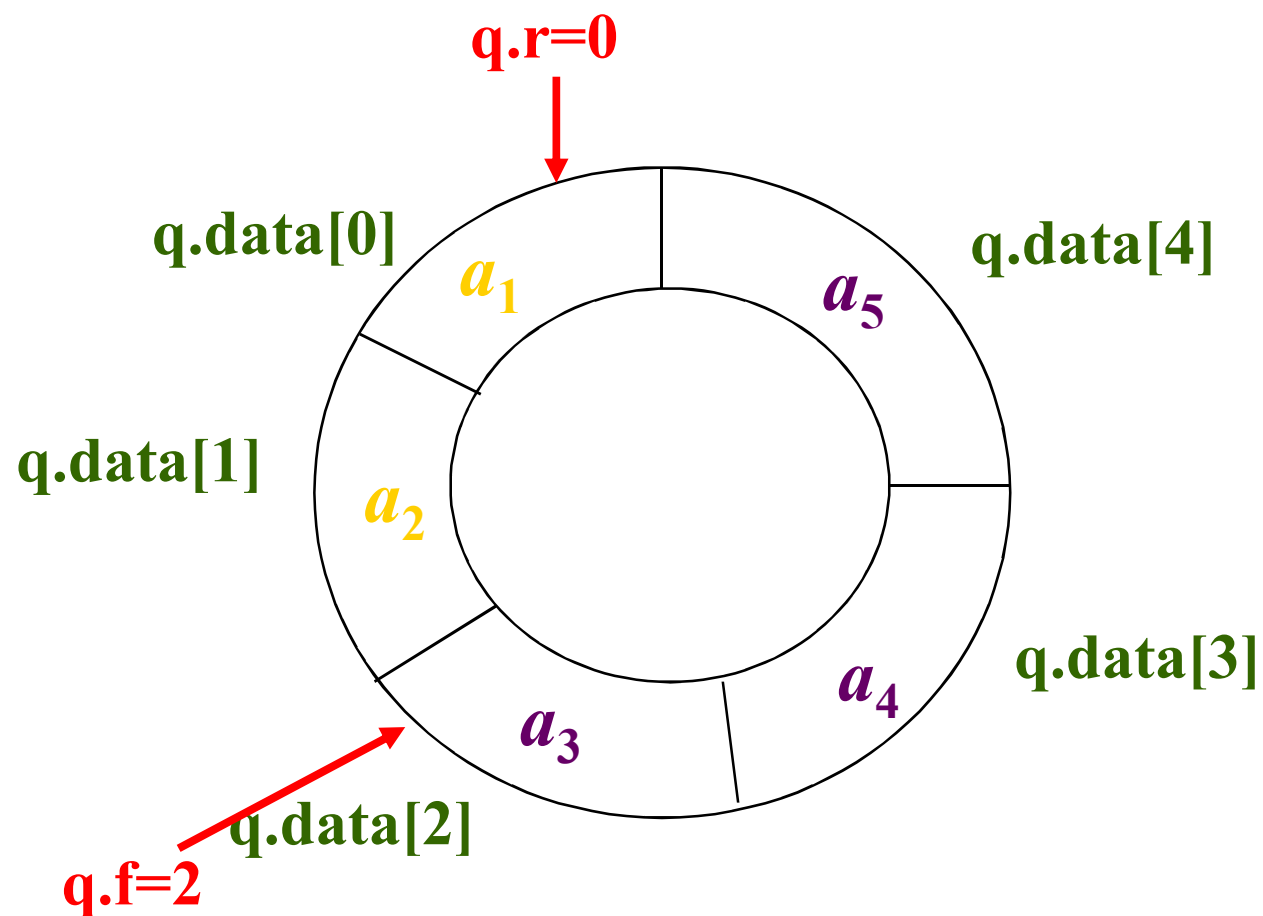
循环队列—入队 $x=a_5$



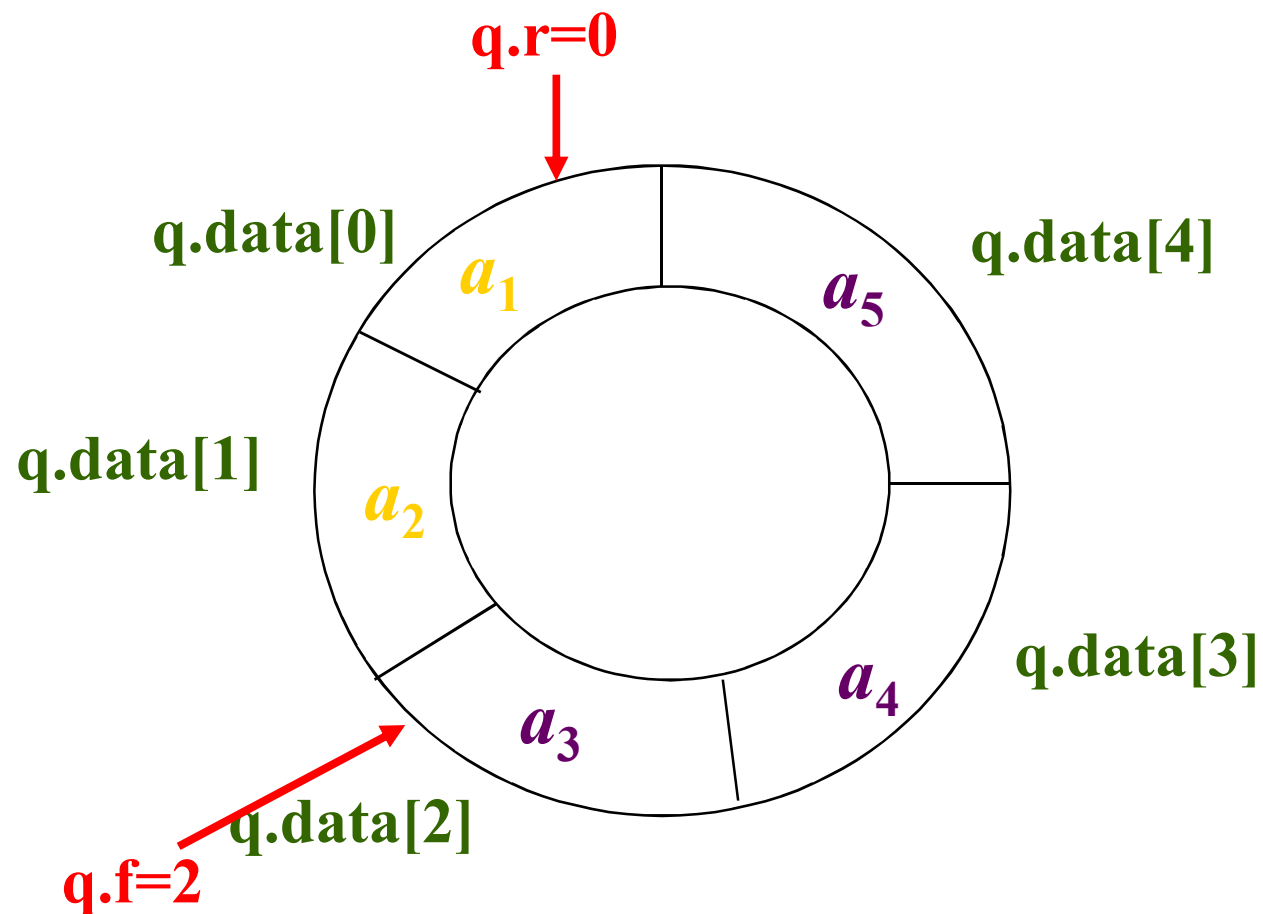
循环队列—入队 $x=a_5$



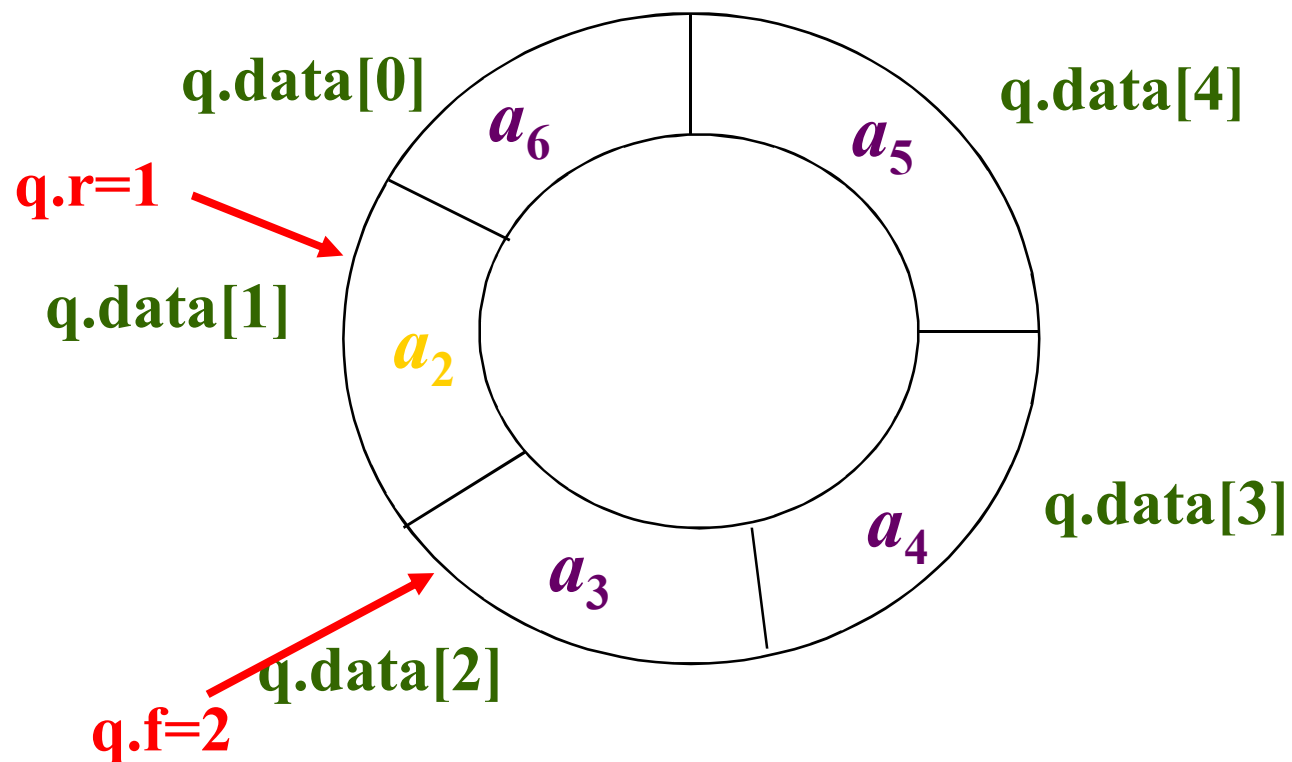
循环队列—出队



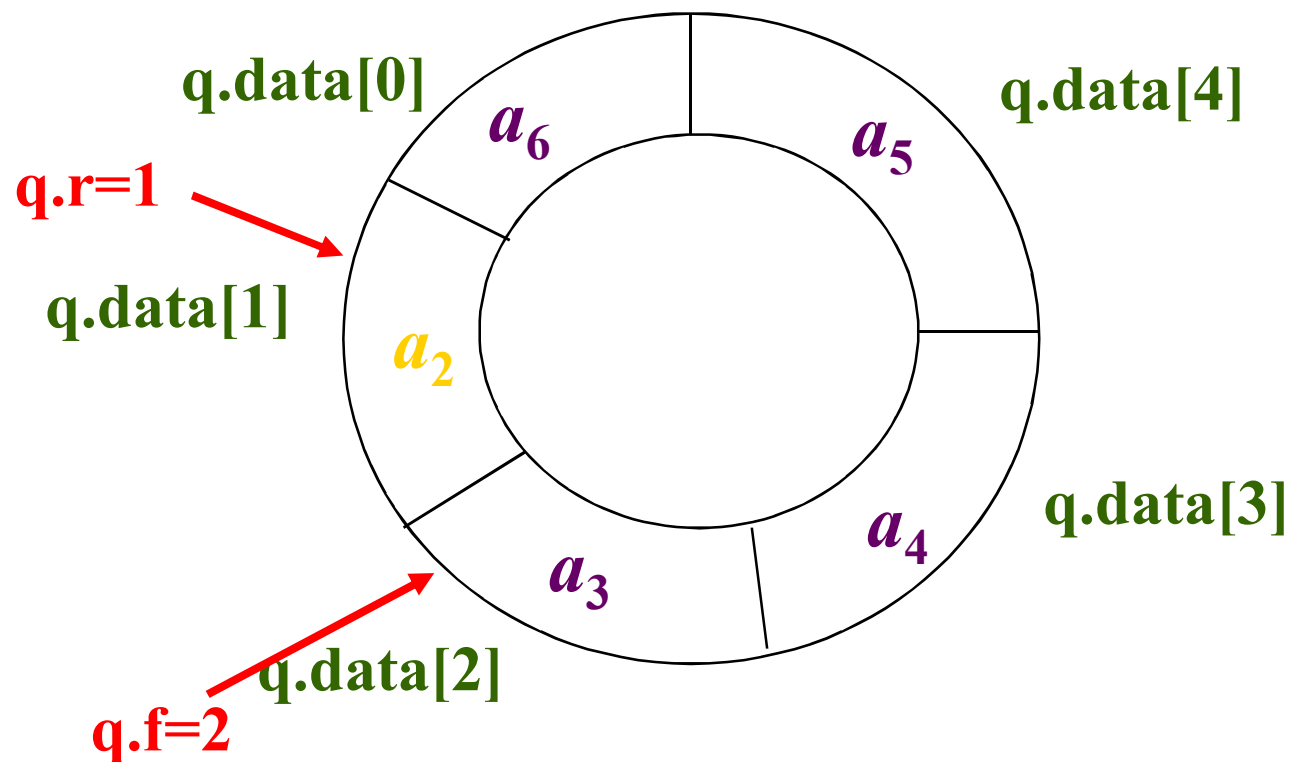
循环队列—入队 $x=a_6$



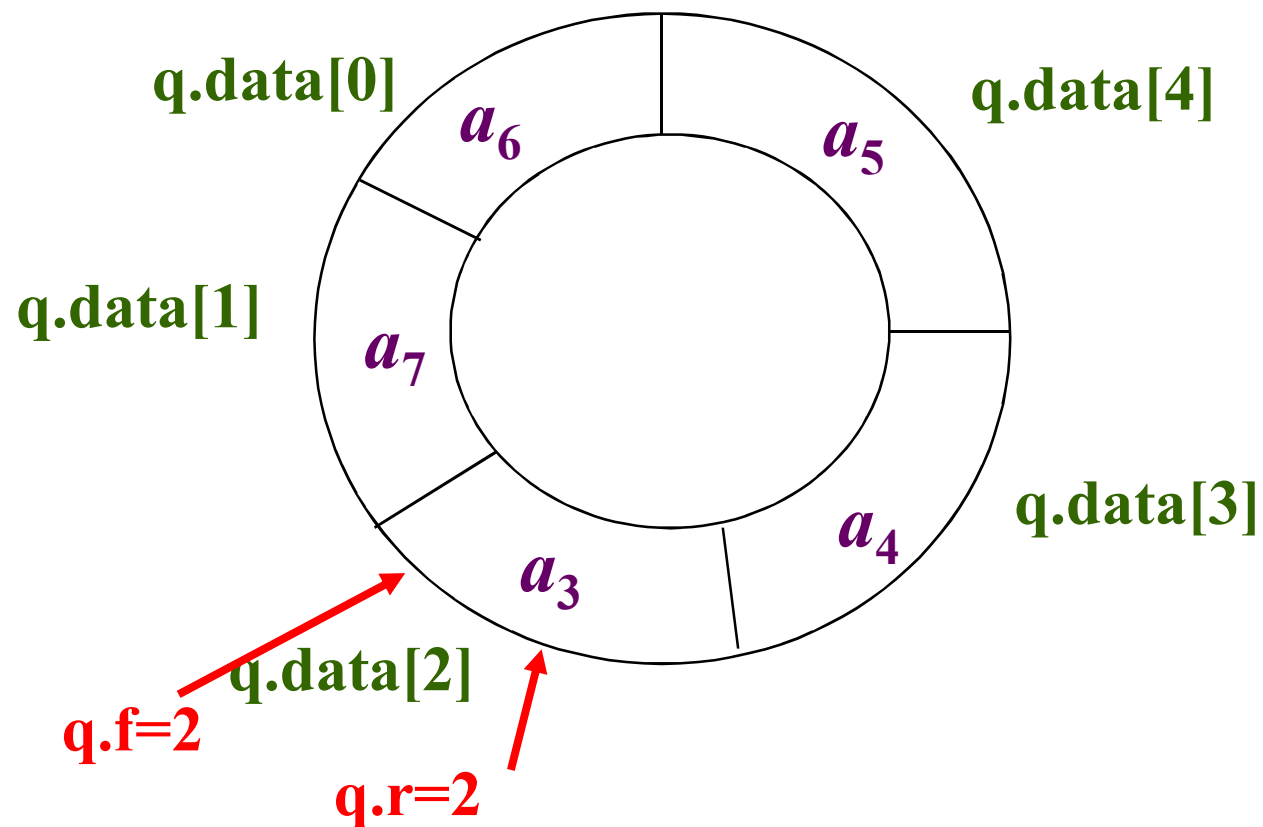
循环队列—入队 $x=a_6$



循环队列—入队 $x=a_7$



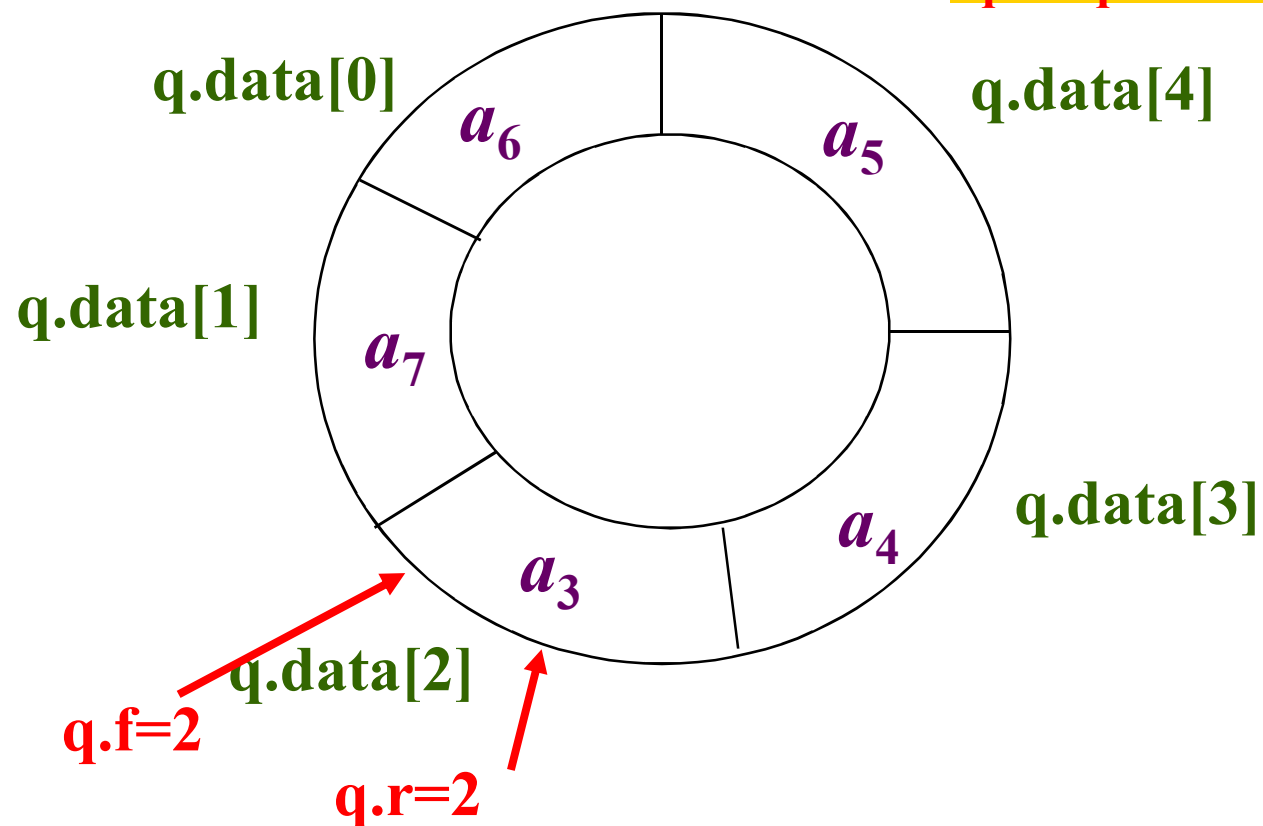
循环队列—入队 $x=a_7$



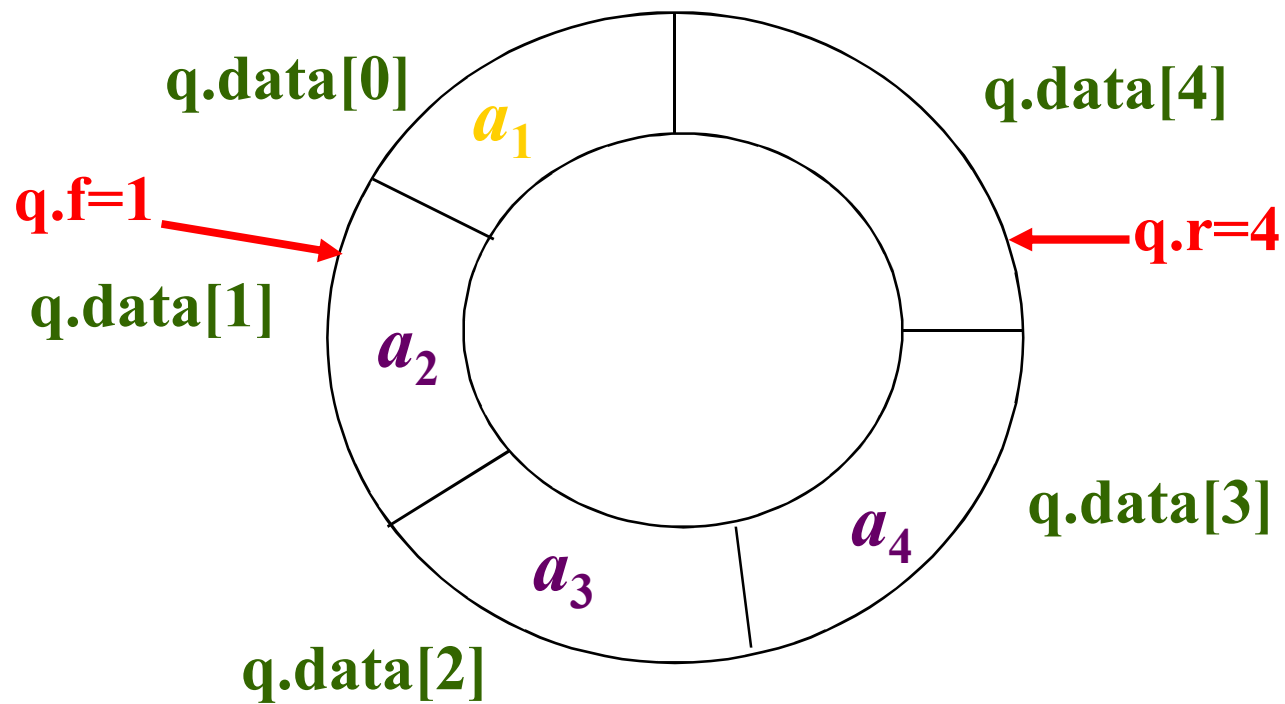
循环队列—入队 $x=a_8$

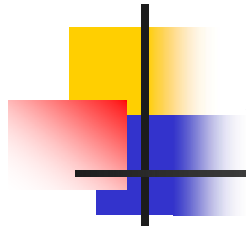
溢出，队满

$q.f == q.r$

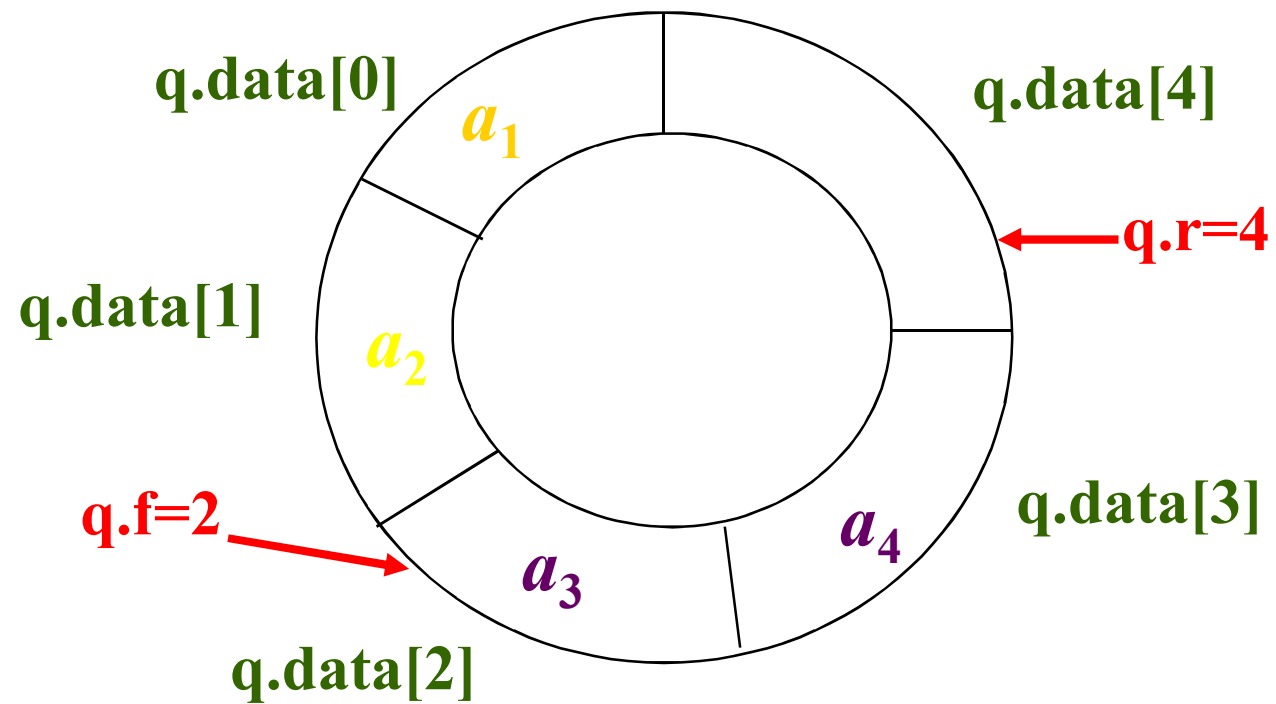


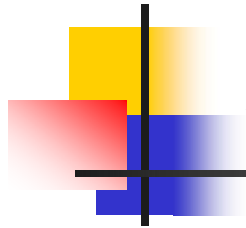
循环队列



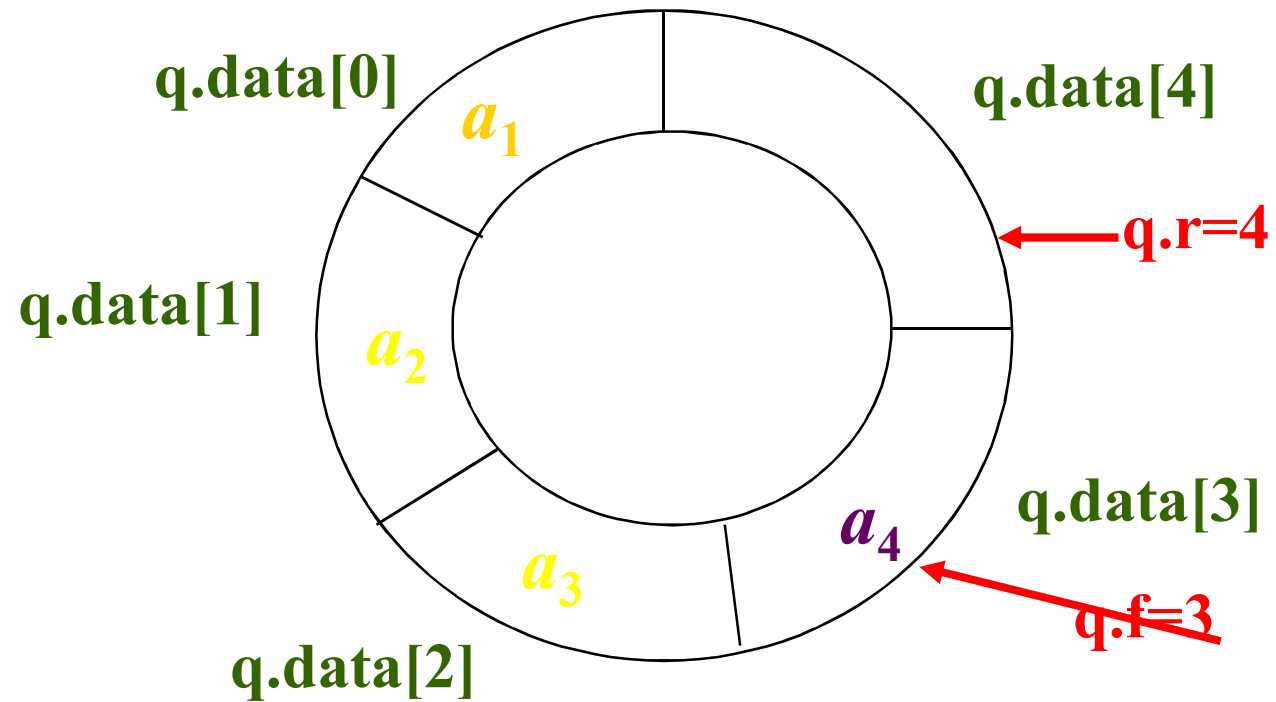


循环队列





循环队列

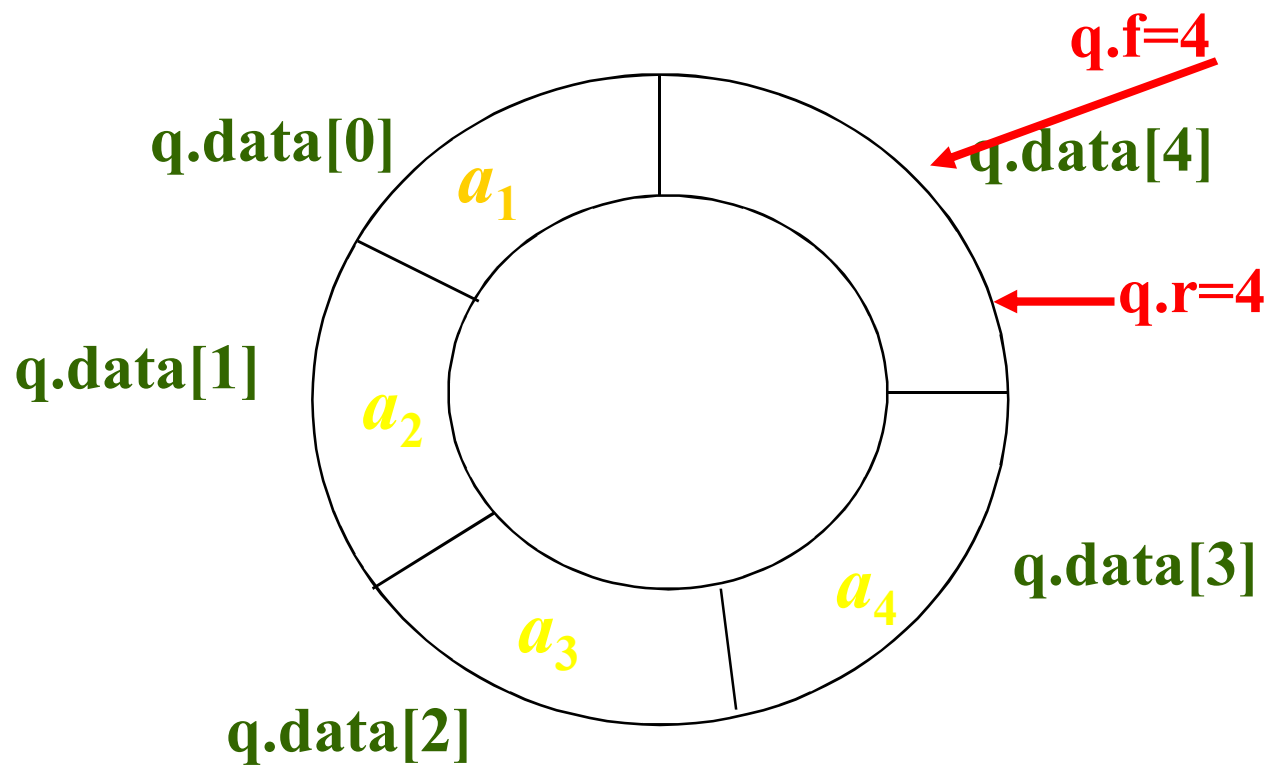


队满，队空时均有 $q.f == q.r$ 。如何区分何时队空？何时队满？

循环队列

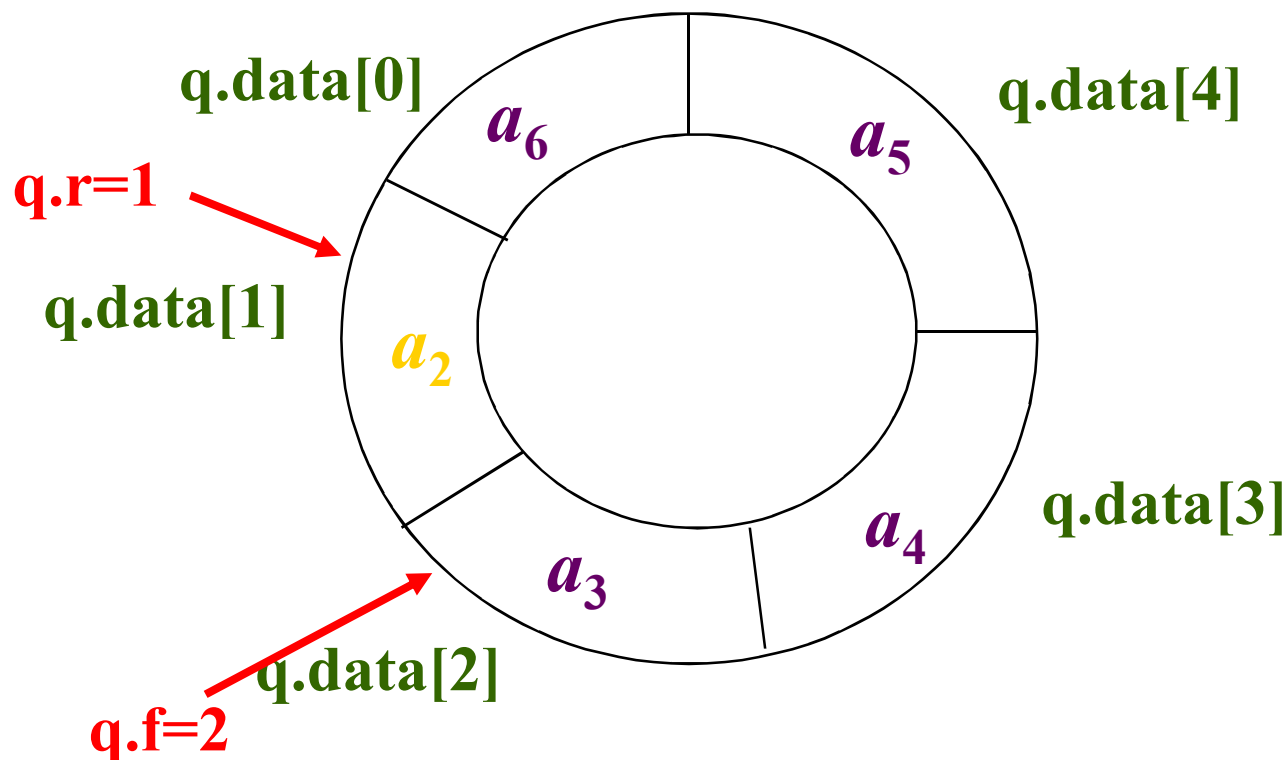
队空

$q.f == q.r$



循环队列—队空队满的区分

方法1: 为区分队空、队满, 牺牲一个存储位置,
当 $(q.r+1)\%MAXSIZE==q.f$ 时认为队满了, $q.r==q.f$ 为队空





循环队列—队空队满的区分

方法1:

队空: $q.r == q.f$

队满: $(q.r + 1) \% \text{MAXSIZE} == q.f$

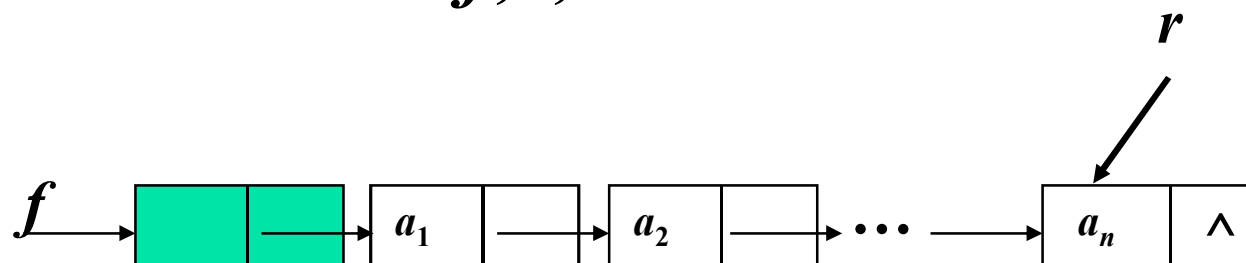
方法2: 设一计数器, 初始化时计数器清0, 入队时, 计数器+1, 出队时计数器-1

■ 练习

- 1 顺序队列如何解决假溢出?
- 2 循环队列如何判断队满和队空?

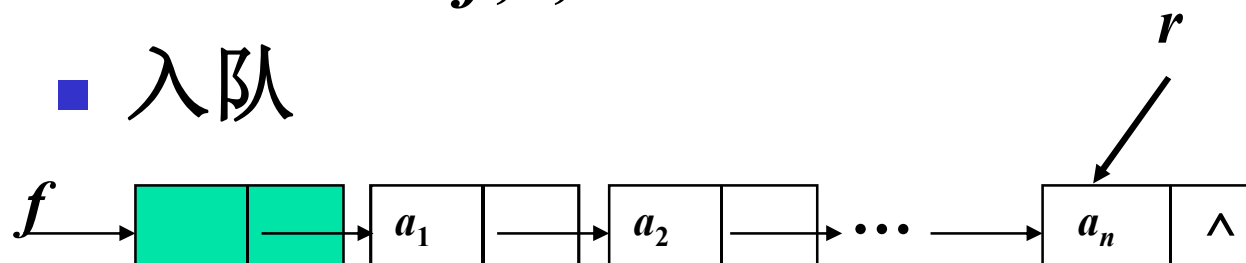
链队

- 定义
- **LinkList** f, r ;



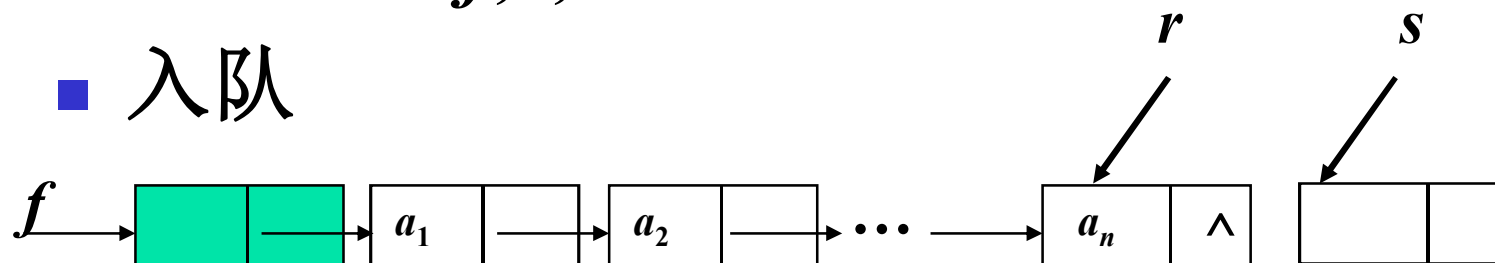
链队

- 定义
- **LinkList** f, r ;
- 入队



链队

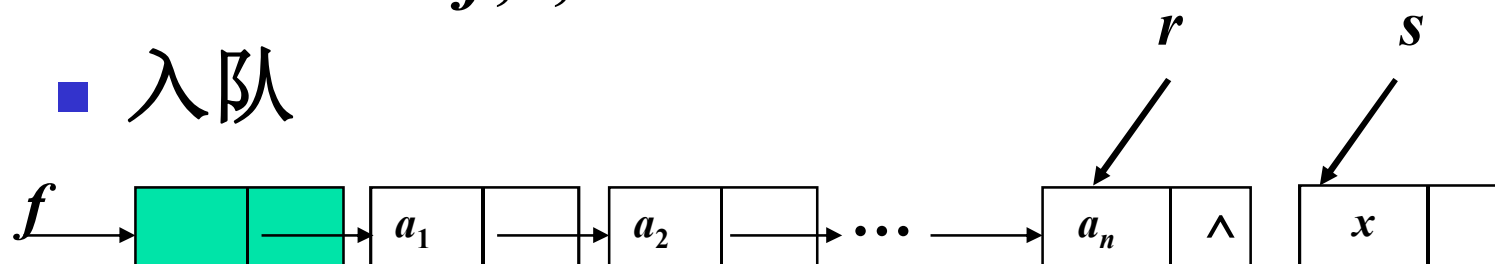
- 定义
- $\text{LinkList } f, r;$
- 入队



$s = (\text{LinkList})\text{malloc}(\text{sizeof}(\text{Node}));$

链队

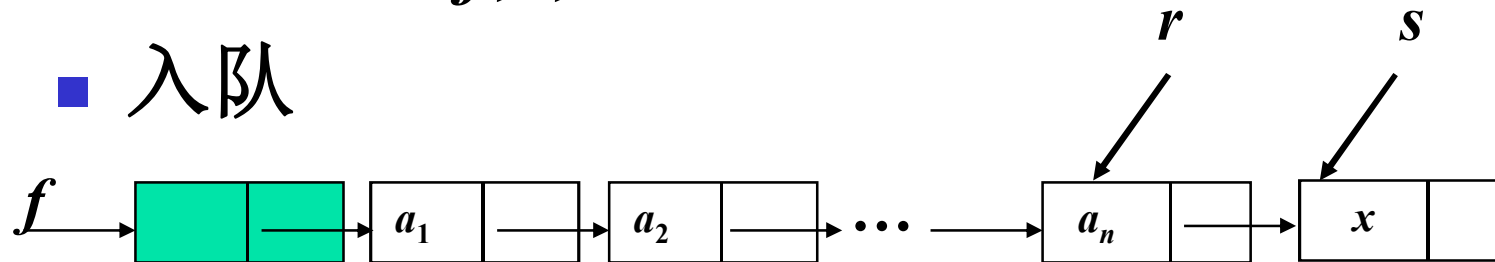
- 定义
- **LinkList** f, r ;
- 入队



```
 $s = (\text{LinkList})\text{malloc}(\text{sizeof}(\text{Node}));$   
 $s \rightarrow \text{data} = x;$ 
```

链队

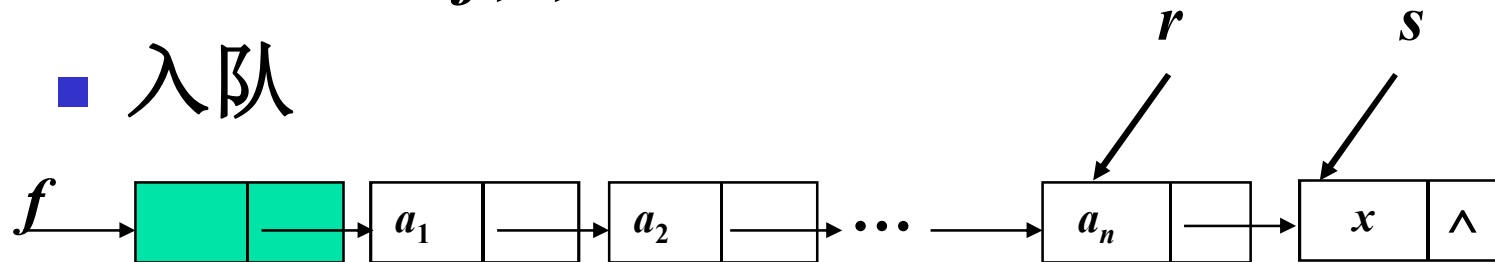
- 定义
- $\text{LinkList } f, r;$
- 入队



```
 $s = (\text{LinkList})\text{malloc}(\text{sizeof}(\text{Node}));$   
 $s \rightarrow \text{data} = x;$   
 $r \rightarrow \text{next} = s;$ 
```


链队

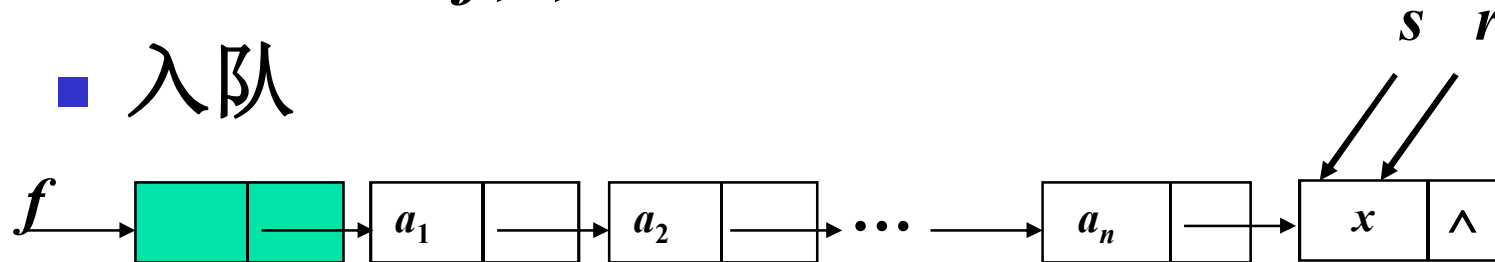
- 定义
- $\text{LinkList } f, r;$
- 入队



```
 $s = (\text{LinkList})\text{malloc}(\text{sizeof}(\text{Node}));$   
 $s \rightarrow \text{data} = x;$   
 $r \rightarrow \text{next} = s; s \rightarrow \text{next} = \text{NULL};$ 
```

链队

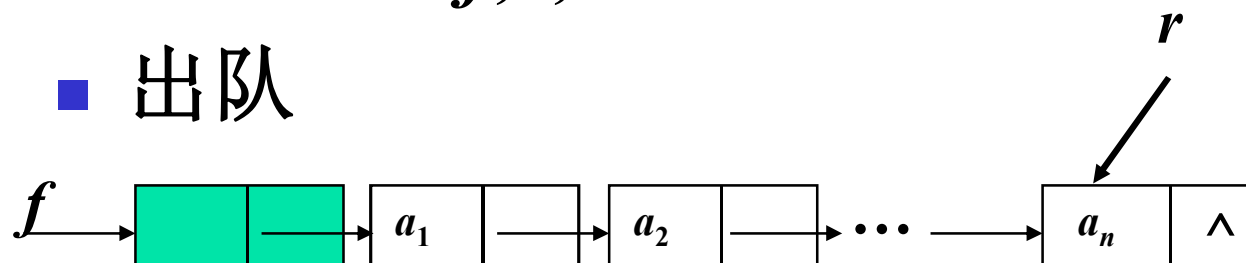
- 定义
- $\text{LinkList } f, r;$
- 入队



```
 $s = (\text{LinkList})\text{malloc}(\text{sizeof}(\text{Node}));$   
 $s \rightarrow \text{data} = x;$   
 $r \rightarrow \text{next} = s; s \rightarrow \text{next} = \text{NULL};$   
 $r = s;$ 
```

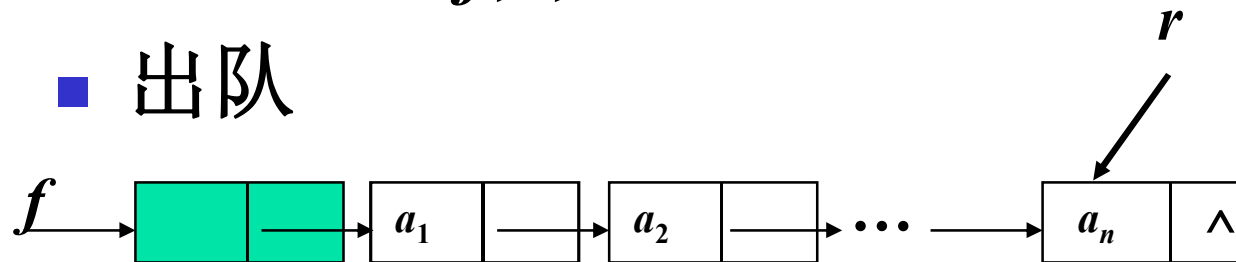
链队

- 定义
- **LinkList** f, r ;
- 出队



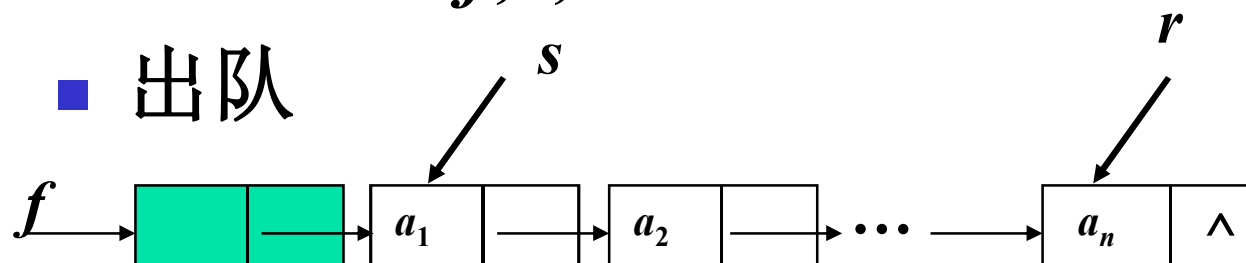
链队— $f \rightarrow \text{next} \neq \text{NULL}$

- 定义
- $\text{LinkList } f, r;$
- 出队



链队

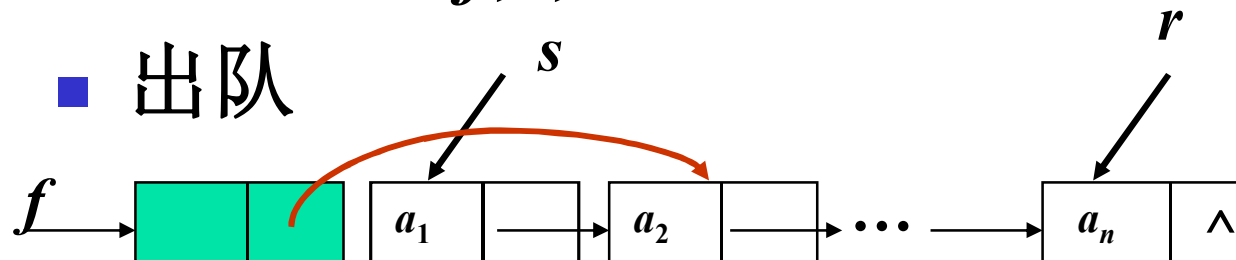
- 定义
- **LinkList f, r ;**
- 出队



$s=f \rightarrow \text{next};$

链队

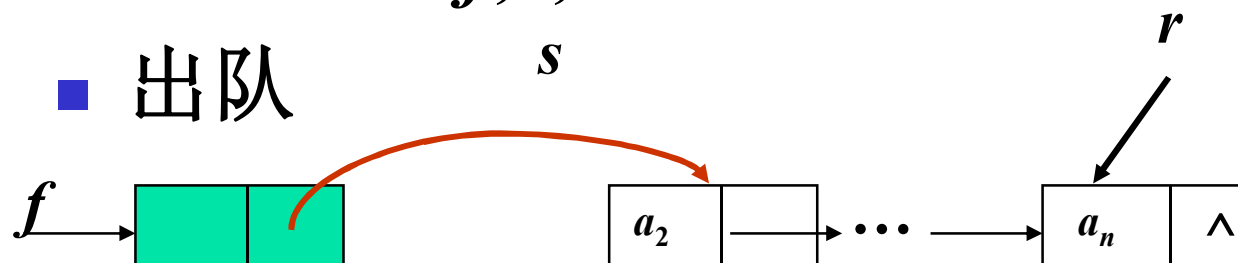
- 定义
- $\text{LinkList } f, r;$
- 出队



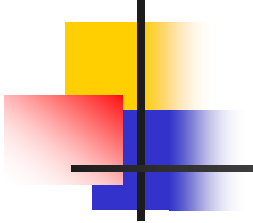
$s = f \rightarrow \text{next};$
 $f \rightarrow \text{next} = s \rightarrow \text{next};$

链队

- 定义
- $\text{LinkList } f, r;$
- 出队



$s=f \rightarrow \text{next};$
 $f \rightarrow \text{next}=s \rightarrow \text{next};$
 $\text{free}(s);$



双端队列：是限定插入和删除运算在表的两端进行的线性表，它好像一个特别书架，取、存书限定在两边进行。

超队列：是一种输入受限的双端队列，即删除仅可在一端进行，而插入仍允许在两端进行。它好像一种特殊的队列，允许有的刚插入的元素就可删除。