目录

第 四 章 栈和队列

1、栈

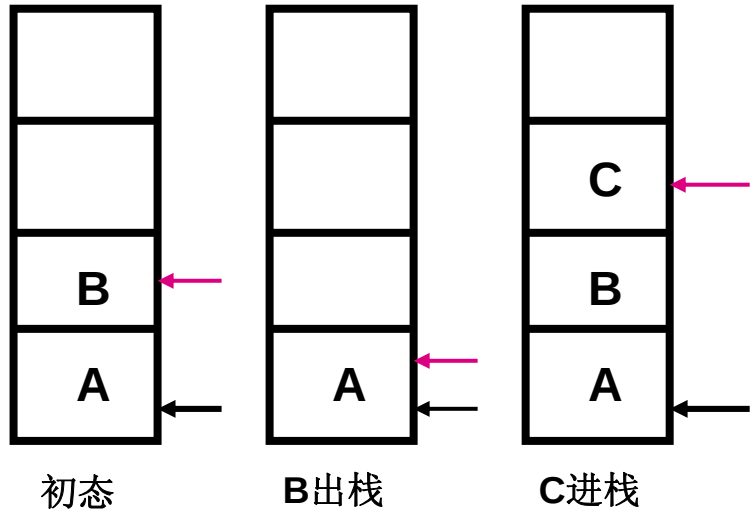
1、定义：

限定仅只能在表尾端进行插入和删除的线性表。

栈顶：表尾端被称之为栈顶。

栈底：和表尾相对应的另一端，称之为栈底。

时间有序表：**LIFO** 特征的线性结构。



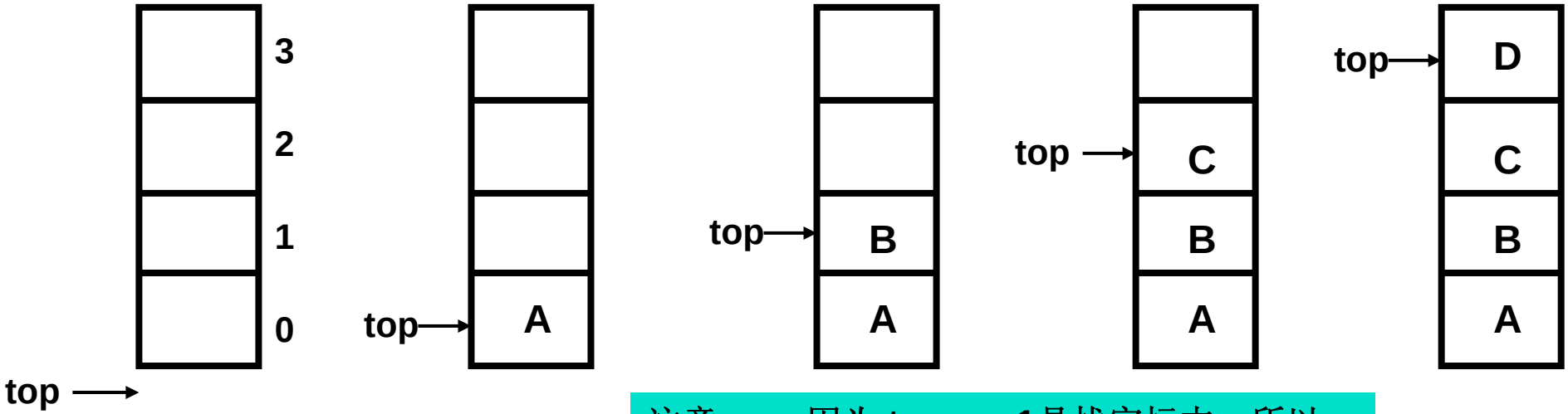
2、栈的应用：表达式计算的 ADT (Abstract Data Type)：

```
Template <class Type> class Stack {
public:
 Stack( int =10 ); //  初始化时栈的空间大小或栈可放结点个数
 void Push( const Type & item ); // 将  item  之值压入堆栈，压之前须判是否上溢。
 Type Pop( );      // 弹出栈顶元素的数据值返回，操作之前须判栈空吗。
 Type GetTop( );   // 取栈顶元素的数据值返回，取之前须判栈空吗。
 void MakeEmpty ( ); // 置空栈。
 int IsEmpty( ) const;  // 栈空为1，否则为 0
 int IsFull( ) const ;   //  栈满为 1，否则为 0
}
```

1、栈

2、栈的应用：表达式计算的表示：

· 顺序表示的堆栈的实现：



空栈

top == -1 是栈空标志

maxsize = = 4

注意： 因为 top == -1是栈空标志，所以 top 指针指示真正的栈顶元素的下标地址。

栈满时的处理方法：

1、报错。返回操作系统。

2、分配更大的空间，作为栈的存储空间。将 原栈的内容移入新栈。

1、栈

2、栈的应用：表达式计算的表示：

· 顺序表示的栈的程序实现：

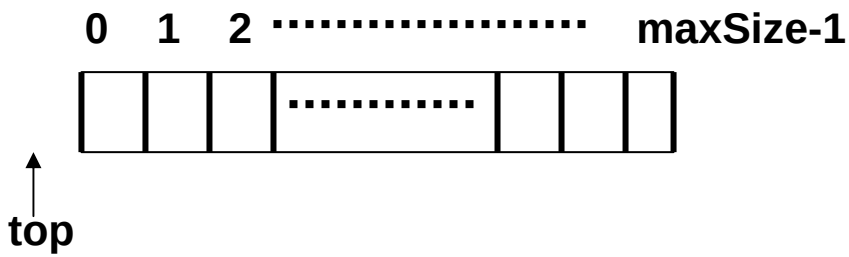
```
# include < assert.h >

Template <class Type> class Stack {
public:
 Stack( int =10 ); //  初始化时栈的空间大小或栈可放结点个数
 ～Stack( ){ delete [ ] elements; } //  初始化时栈的空间大小或栈可放结点个数
 void Push( const Type & item ); // 将  item  之值压入堆栈，压之前须判是否上溢。
 Type Pop( );      // 弹出栈顶元素的数据值返回，操作之前须判栈空吗。
 Type GetTop( );    // 取栈顶元素的数据值返回，取之前须判栈空吗。
 void MakeEmpty ( top = -1  ); // 置空栈。
 int IsEmpty( ) const { return top == -1; }   // 栈空为1，否则为 0
 int IsFull( ) const { return top ==  maxSize -1; }     //  栈满为 1，否则为 0
Private:
 int top;
 Type * elemets;
 int maxSize;
}
```

1、栈

2、栈的应用：表达式计算的表示：

· 顺序表示的栈的初始化操作：

```
Template <class Type>
Stack <Type> :: Stack(  int  s ) : top(-1), maxSize(s) {
 //  初始化时栈的存储空间的大小为 s 个结点
 elements = new Type[ maxSize ];
 assert( elements != 0  );
}
```

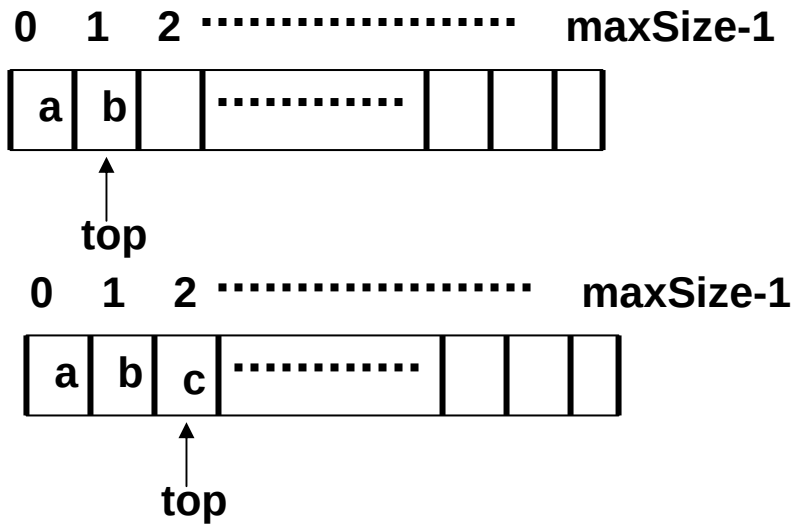


1、栈

2、栈的应用：表达式计算的表示：

· 顺序表示的栈的程序实现 **push** 操作：

```
Template <class Type>
void Stack < Type> :: Push(  const Type &  item ) {
 // 栈不满时，将 item 之值压入堆栈
 assert( ! IsFull );
elements[ ++top ] = item;
}
```

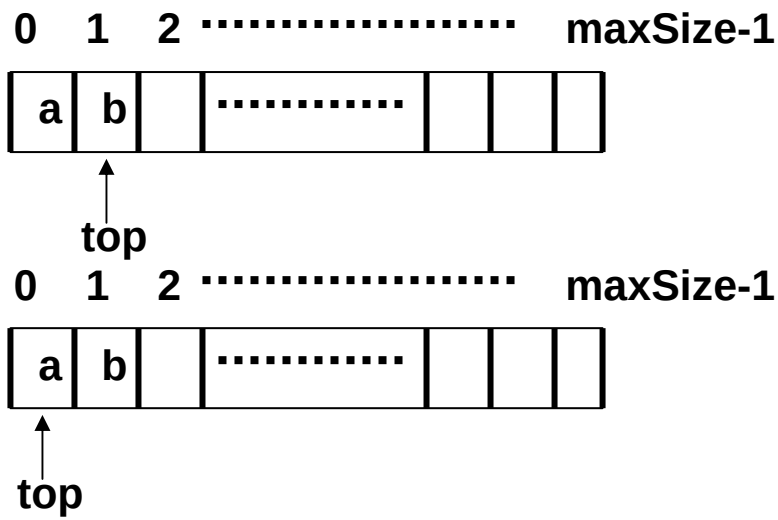


1、栈

2、栈的应用：表达式计算的表示：

· 顺序表示的栈的程序实现 **push** 操作：

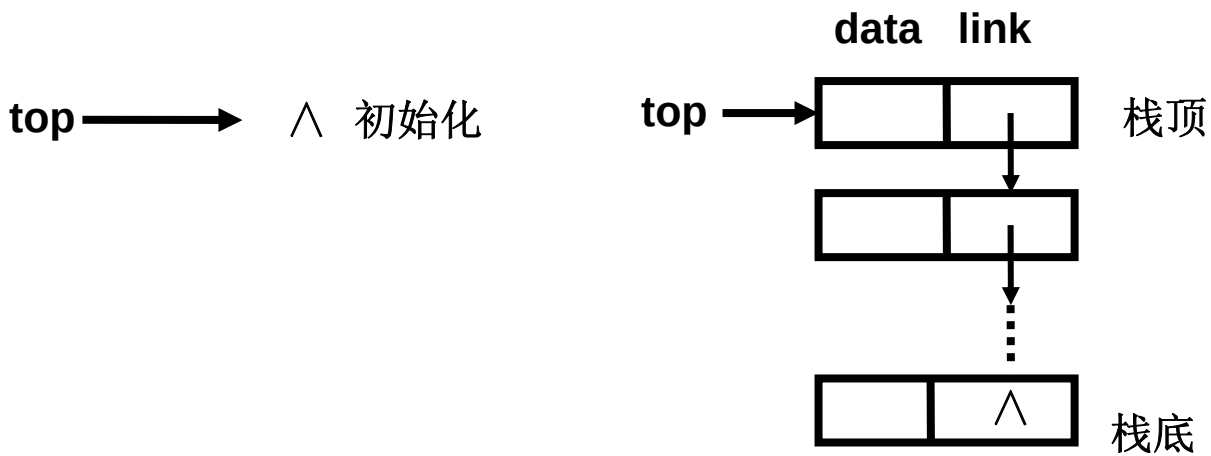
```
Template <class Type>
Type Stack < Type> :: Pop( ) {
 // 栈不空时，返回栈顶之值后，栈顶指针退 1
 assert( ! IsEmpty );
 return elements[ top-- ];
}
```



1、栈

· 链接表示的栈作：**top** 是栈顶指针。栈顶和栈底如图所示。

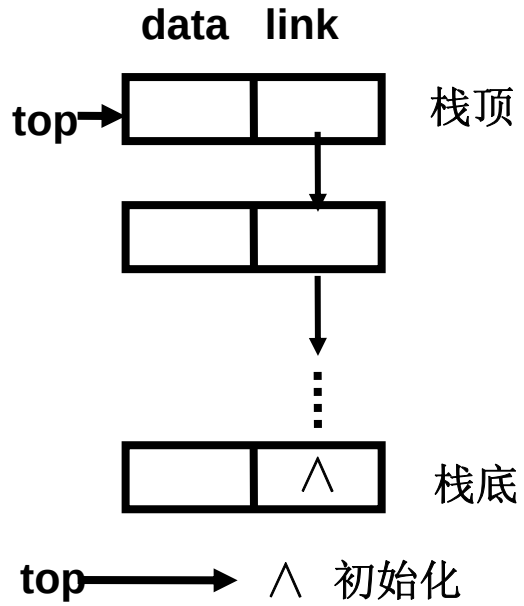
```
Template <class Type> class Stack;
Template <class Type> class StackNode {
friend class Stack < Type >;
private:
 Type data;
 StackNode <Type>  * link;
 StackNode ( Type d = 0; StackNode Type  * l= NULL):data(d), link( l );
}
```



1、栈

·链接表示的栈作：**top** 是栈顶指针。栈顶和栈底如图所示。

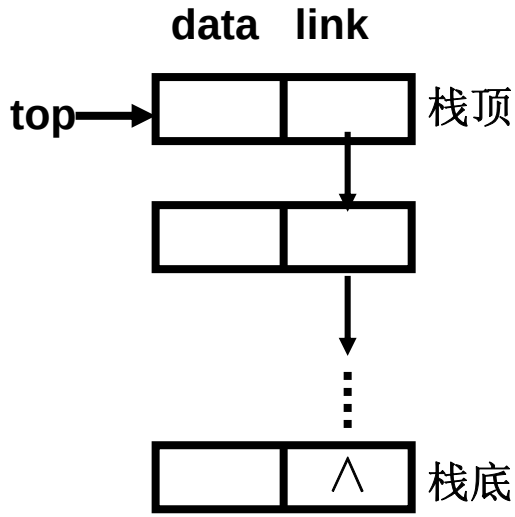
```
Template <class Type> class Stack {
public:
 Stack( ):top (NULL){ }
 ～Stack( );
 void Push( const Type & item );
 Type Pop( );
 Type GetTop( );
 void MakeEmpty ( );
 int IsEmpty( ) const { return top == NULL; }  // 栈空为1，否则为0
Private:
 StackNode< Type> * top;
}
```

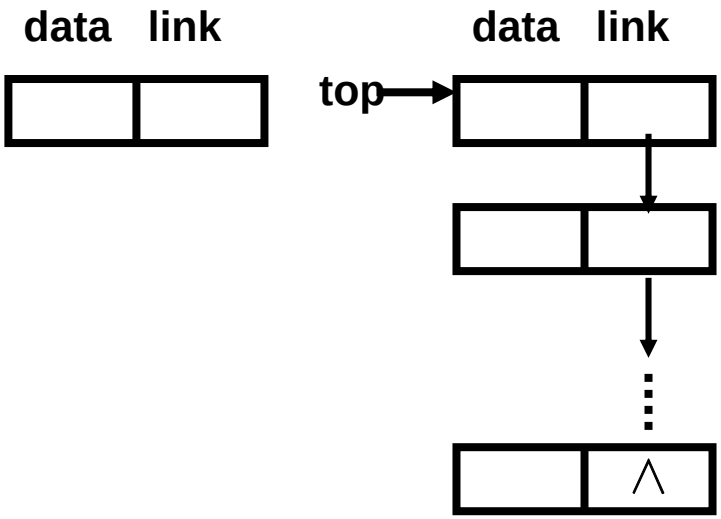


1、栈

· 链接表示的栈的 **Push** 操作：

```
Template <class Type>
void Stack < Type> :: Push(  const Type &  item ) {
 top = new StockNode< Type > (item,top);
 assert(  top != NULL );
}
```

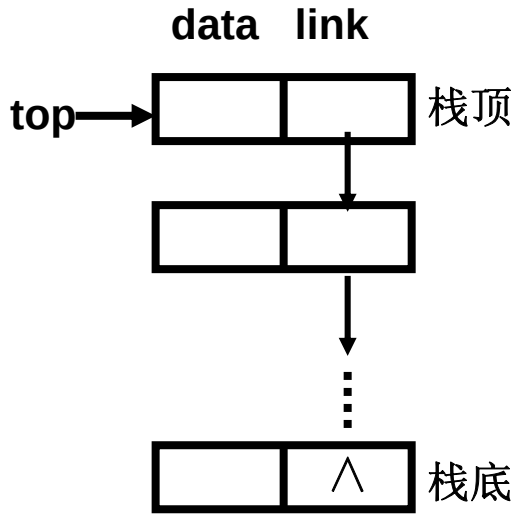


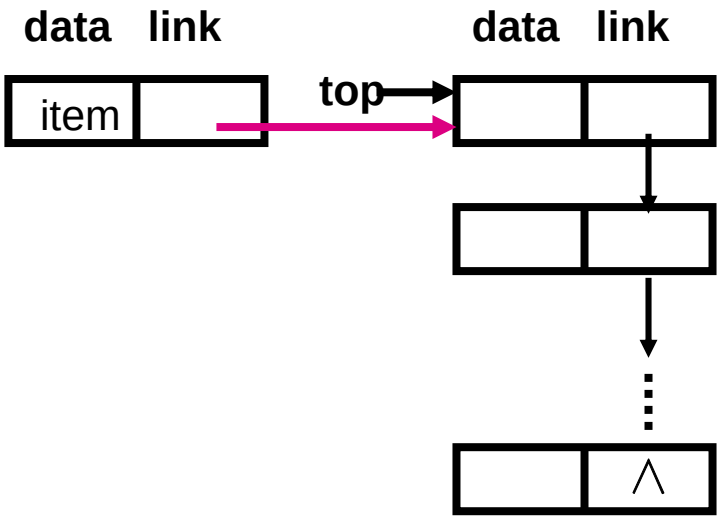


1、栈

· 链接表示的栈的 **Push** 操作：

```
Template <class Type>
void Stack < Type> :: Push(  const Type &  item ) {
 top = new StockNode< Type > (item,top);
 assert(  top != NULL );
}
```

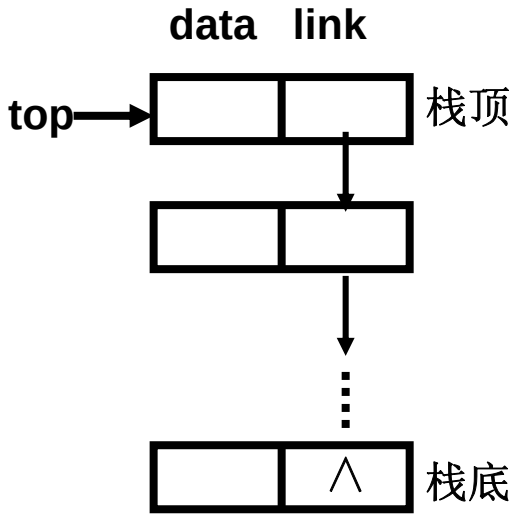


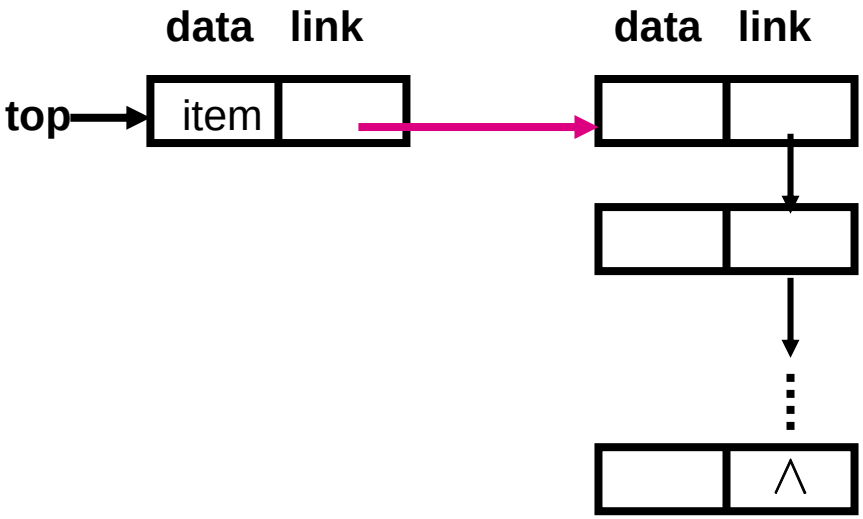


1、栈

· 链接表示的栈的 **Push** 操作：

```
Template <class Type>
void Stack < Type> :: Push(  const Type &  item ) {
 top = new StockNode< Type > (item,top);
 assert(  top != NULL );
}
```

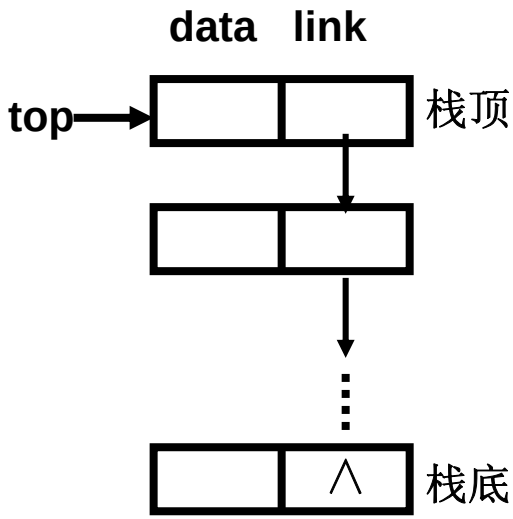




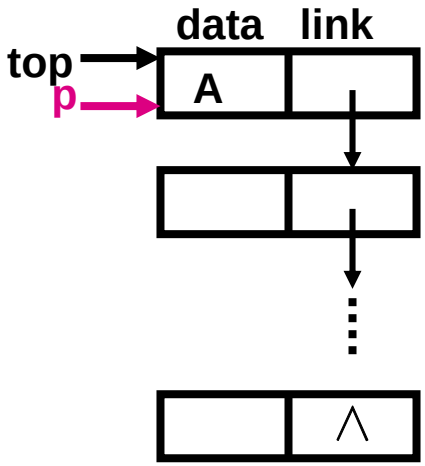
1、栈

· 链接表示的栈的操作：s 是栈顶指针。栈顶和栈底如图所示。

```
Template <class Type>
Type Stack < Type> :: Pop( ) {
 // 栈不空时，返回栈顶之值后，栈顶指针退 1
 assert( ! IsEmpty );
 StackNode< Type > *p = top;
 Type retvalue = p->data;
 top = top ->link; delete p;
 return retvalue;
}
```



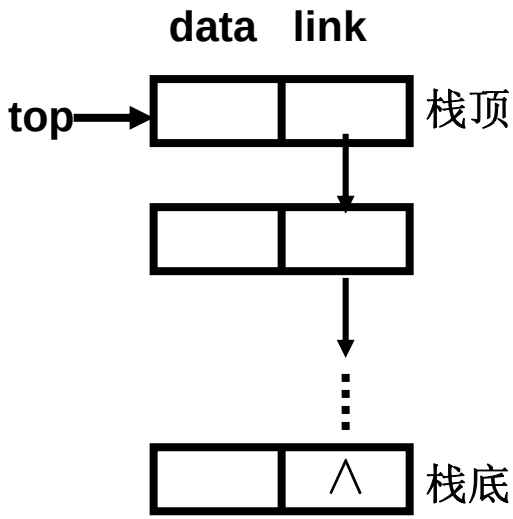
retvalue = ‘ A‘



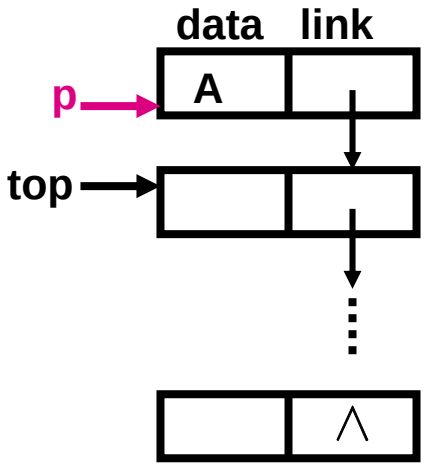
1、栈

· 链接表示的栈的操作：s 是栈顶指针。栈顶和栈底如图所示。

```
Template <class Type>
Type Stack < Type> :: Pop( ) {
 // 栈不空时，返回栈顶之值后，栈顶指针退 1
 assert( ! IsEmpty );
 StackNode< Type > *p = top;
 Type retvalue = p->data;
 top = top ->link; delete p;
 return retvalue;
}
```



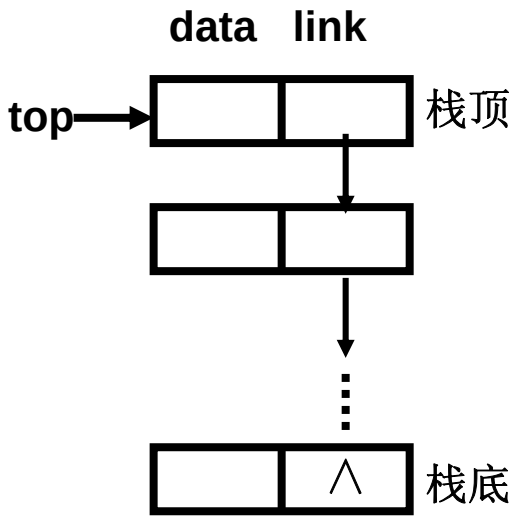
retvalue = ‘ A‘



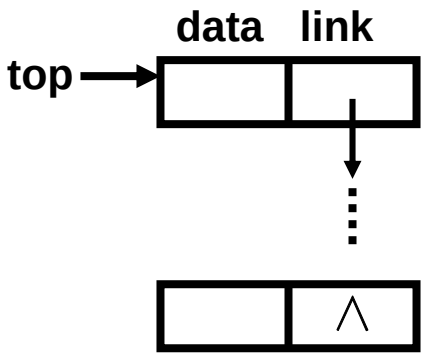
1、栈

· 链接表示的栈的操作：s 是栈顶指针。栈顶和栈底如图所示。

```
Template <class Type>
Type Stack < Type> :: Pop( ) {
 // 栈不空时，返回栈顶之值后，栈顶指针退 1
 assert( ! IsEmpty );
 StackNode< Type > *p = top;
 Type retvalue = p->data;
 top = top ->link; delete p;
 return retvalue;
}
```



retvalue = ‘ A‘



2、栈的应用：表达式计算的应用

· 数制转换：

例如：10 进制和 8 进制之间的数的转换。

$(1348)_{10} = 8^3 * a_3 + 8^2 * a_2 + 8 * a_1 + 8^0 * a_0$　　// 两边同除以 8

168 余 4 $= (8^2 * a_3 + 8^1 * a_2 + a_1)$ 余 4 即 $a_0 = 4$

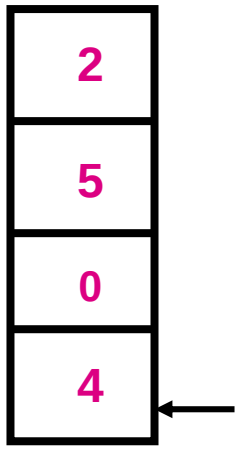
$168 = 8^2 * a_3 + 8^1 * a_2 + a_1$　　// 两边同除以 8

21 余 0 $= (8 * a_3 + a_2)$ 余 0　　即 $a_1 = 0$

$21 = 8 * a_3 + a_2$　　// 两边同除以 8

2 余 5 $= (a_3)$　　余 5　　即 $a_2 = 5$

$a_3 = 2$



2、栈的应用：表达式计算的应用

· 数制转换：

顺便提一句， 10 进制小数如何变成 2 进制小数：

例如： $(0.4)_{10} = (\ ?\ )_2$

$0.4 \times 8 = 3.2$ → $(\ 0.3\ )_2$ → $(\ 0.011\ )_2$

2、栈的应用：表达式计算的应用

· 数制转换：

顺便提一句， 10 进制小数如何变成 2 进制小数：

例如： $(0.4)_{10} = (\ ?\ )_2$

$0.4 \times 8 = 3.2$ → $(\ 0.3\ )_2$ → $(\ 0.011\ )_2$

$0.2 \times 8 = 1.6$ → $(\ 0.31\ )_2$ → $(\ 0.011001\ )_2$

2、栈的应用：表达式计算的应用

· 数制转换：

顺便提一句， 10 进制小数如何变成 2 进制小数：

例如： $(0.4)_{10} = (\ ?\ )_2$

$0.4 \times 8 = 3.2$ → $(\ 0.3\ )_2$ → $(\ 0.011\ )_2$

$0.2 \times 8 = 1.6$ → $(\ 0.31\ )_2$ → $(\ 0.011001\ )_2$

$0.6 \times 8 = 4.8$ → $(\ 0.314\ )_2$ → $(\ 0.011001100\ )_2$

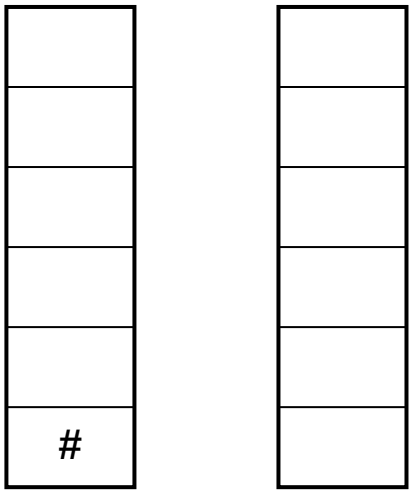
2、栈的应用：表达式计算

·表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

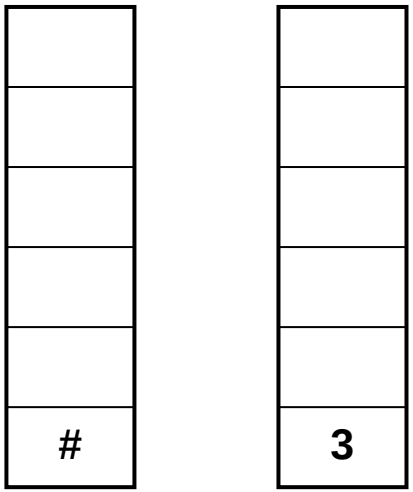
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



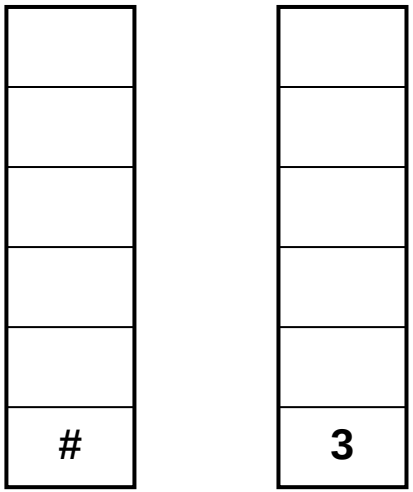
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



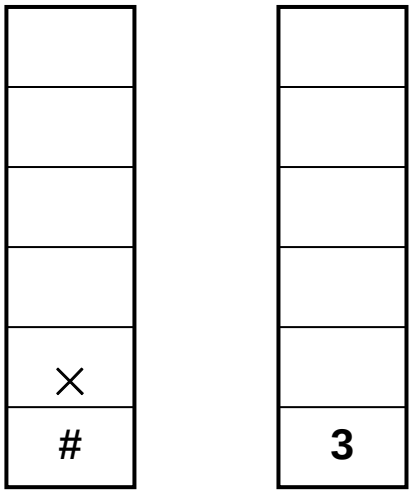
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

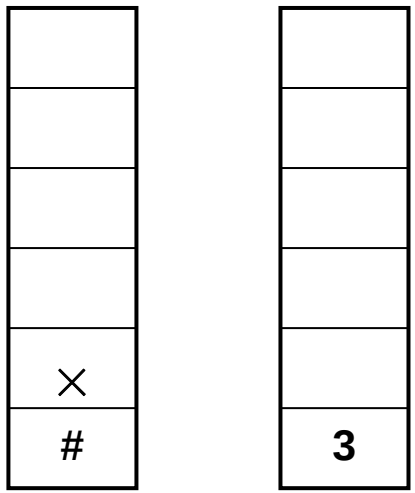
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

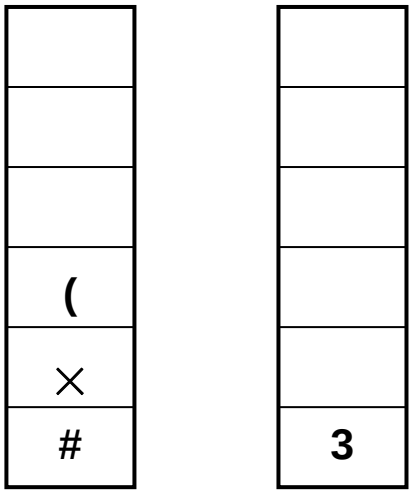
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

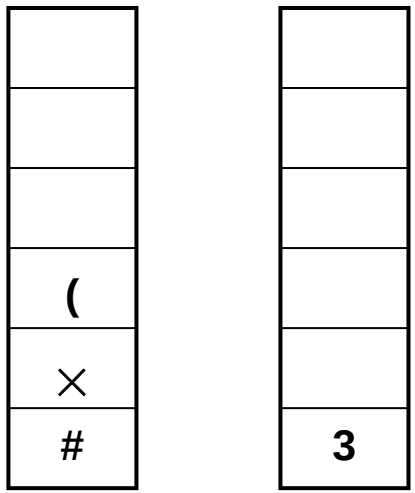
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

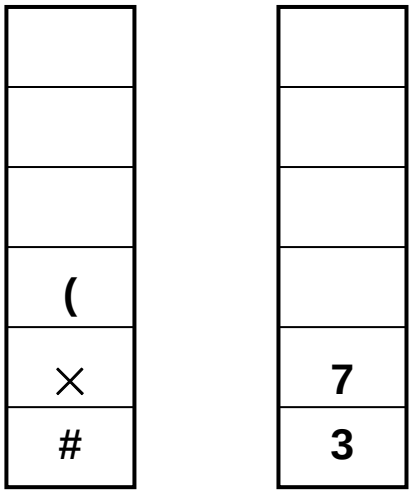
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：x = 3 × (7-2)

解：(≻ * / > + - ≻) #

如： x = 3 × (7-2)

执行过程：# 3 × (7-2) #



OPTR栈　　OPND栈

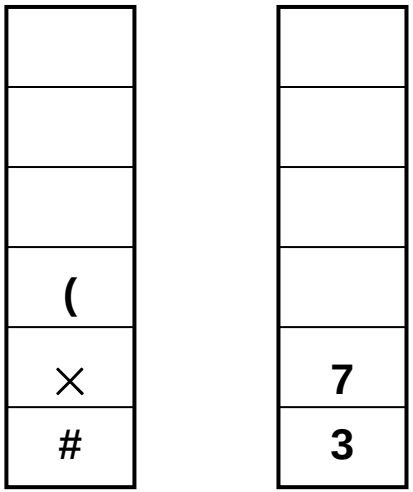
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

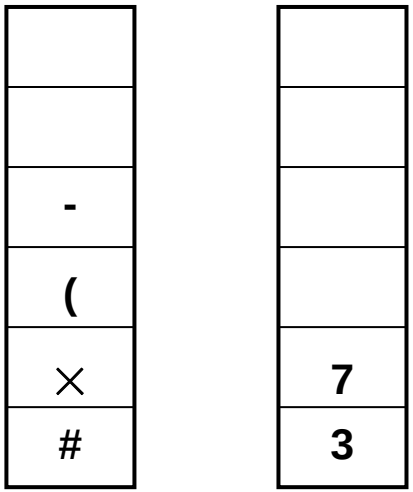
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

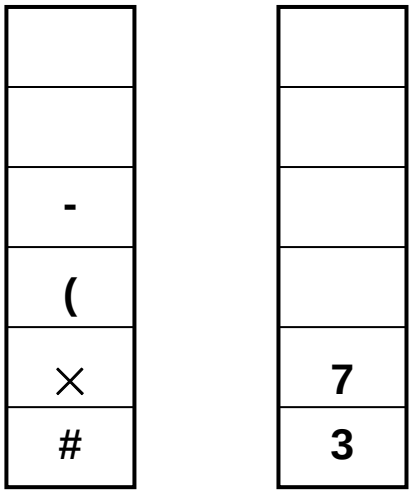
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：x = 3 × (7-2)

解：(≻ * / > + - ≻) #

如： x = 3 × (7-2)

执行过程：# 3 × (7-2) #



OPTR栈　　OPND栈

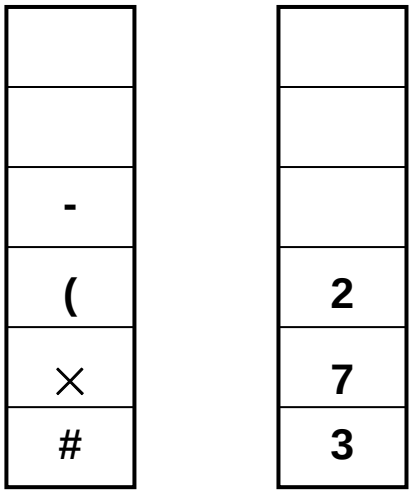
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

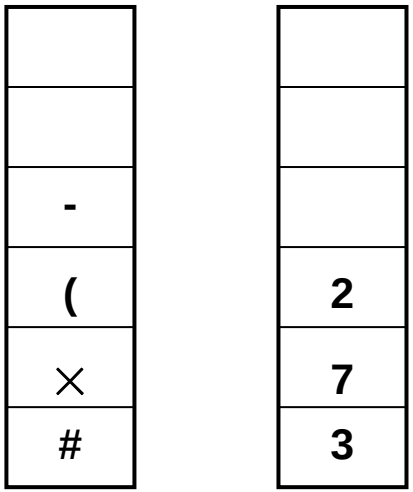
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

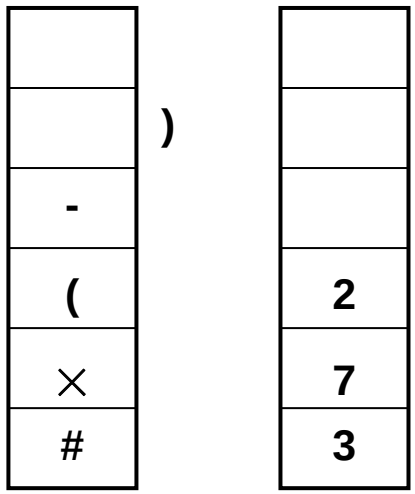
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：x = 3 × (7-2)

解：(≻ * / > + - ≻) #

如： x = 3 × (7-2)

执行过程：# 3 × (7-2) #



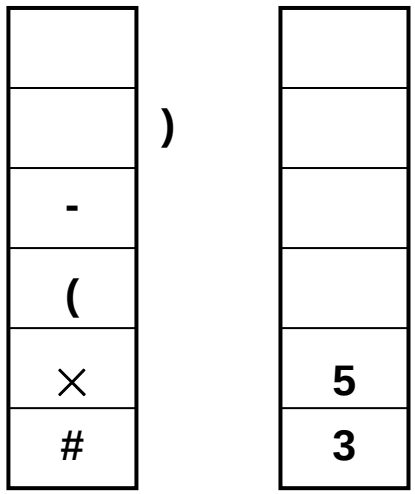
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：x = 3 × (7-2)

解：(≻ * / > + - ≻) #

如： x = 3 × (7-2)

执行过程：# 3 × (7-2) #



OPTR栈　　OPND栈

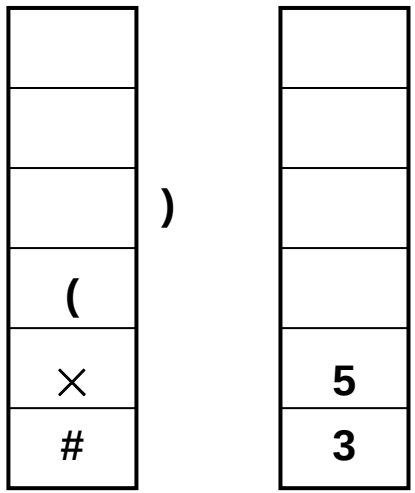
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：x = 3 × (7-2)

解：(≻ * / > + - ≻) #

如： x = 3 × (7-2)

执行过程：# 3 × (7-2) #



OPTR栈　　OPND栈

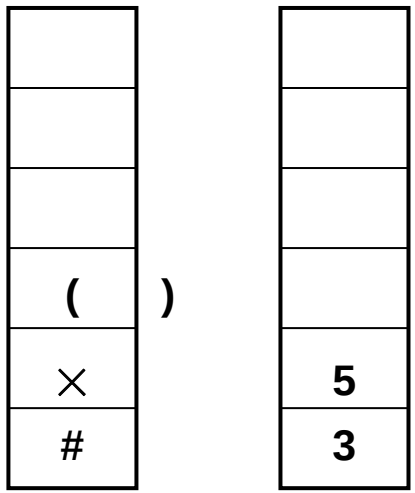
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：x = 3 × (7-2)

解：(≻ * / > + - ≻) #

如： x = 3 × (7-2)

执行过程：# 3 × (7-2) #



OPTR栈　　OPND栈

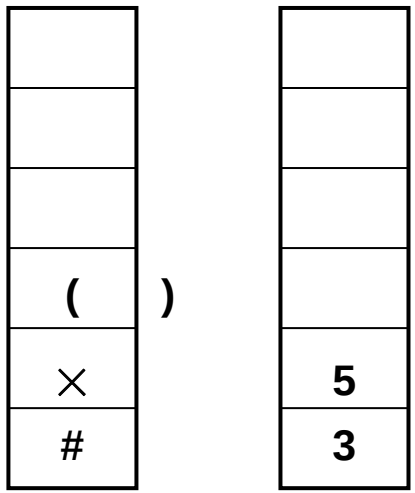
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：x = 3 × (7-2)

解：(≻ * / > + - ≻) #

如： x = 3 × (7-2)

执行过程：# 3 × (7-2) #



OPTR栈　　OPND栈

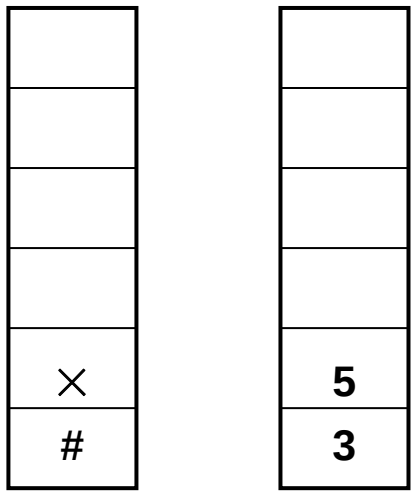
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

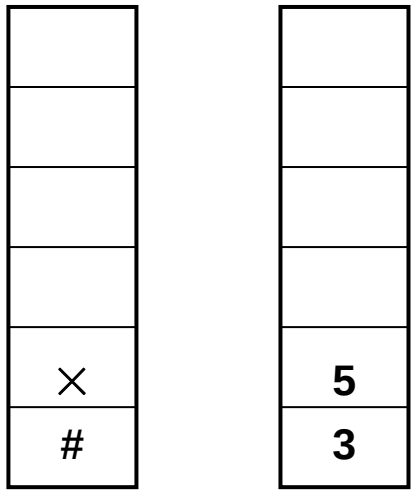
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

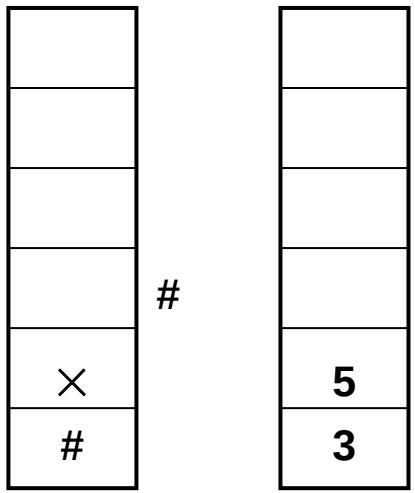
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

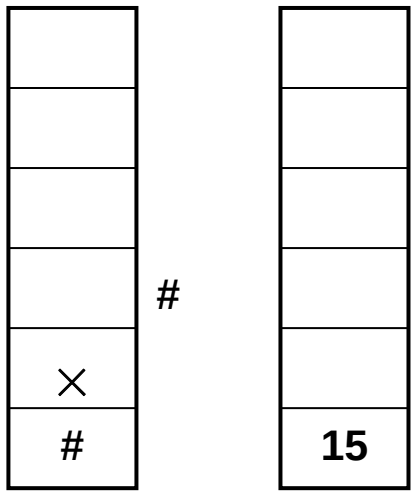
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

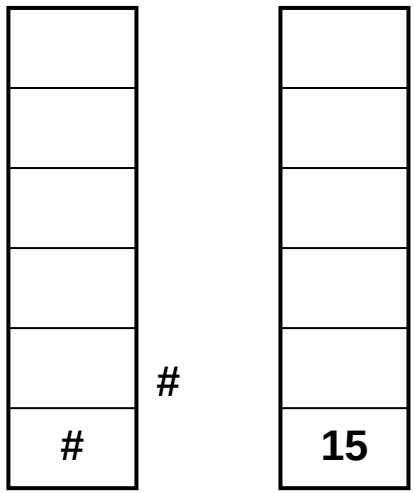
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



OPTR栈 **OPND栈**

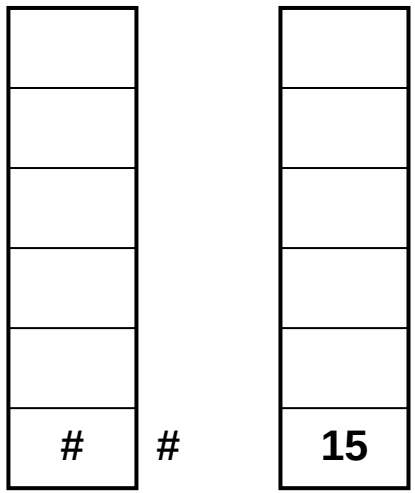
2、栈的应用：表达式计算

·编译中表达式的计算：如计算表达式的值：**x = 3 × (7-2)**

解：**(≻ * / > + - ≻) #**

如： **x = 3 × (7-2)**

执行过程：**# 3 × (7-2) #**



算法：

1. # 压入到 **OPTR** 栈。

2· 扫描中遇到操作数，进入**OPND** 栈。

3. 运算符及 **(** 则进 **OPTR** 栈。在同一 **( )** 表达式中，运算符进栈时，必须保持栈顶的运算符的优先级最高，否则应将 **OPND** 栈中的操作数完成该运算符对应的操作。并将该运算符自 **OPTR** 栈中弹出。

2、栈的应用：表达式计算

·编译中表达式的计算：

算符优先数

操作符	#	(	× / %	＋ －	)	
Isp: 栈内优先数	**0**	**1**	**5**	**3**	**8**	
Icp: 栈外优先数	**0**	**8**	**4**	**2**	**1**	

·如果表达式转换为后缀表达式，那么计算将非常简单。

1、 3 × (7-2) ⟶ #3 × (7-2)# ⟶ 3· 7·2·－×

2、 A＋B×(C－D)－E/F ⇨ #A＋B×(C－D)－E/F# ⇨ ABCD－×＋EF/-

2、栈的应用：表达式计算

·编译中表达式的计算：将中缀表达式变成后缀表达式

```
void postfix(expression e ) {
   Stack<char> s; char ch, y;
   s.MakeEmpty(  );   s.push( '#')
   while ( cin.get( ch ), ch != '#' ) {
   if ( Isdigit( ch ) cout < < ch;
   else  if ( ch == ')' )
            for ( y = s.Pop(); y != '( '; y = s.Pop( ) ) cout < < y;
         else {
                  for ( y = s.Pop(); isp( y ) > icp( ch );  y = s.Pop( ) cout < < y;
                  s.Push(  y ); s.Push( ch );
                } // else
    }  // while
  while( ! s.IsEmpty( ) { y = s.Pop( ); cout << y; }
}
```

3、队列

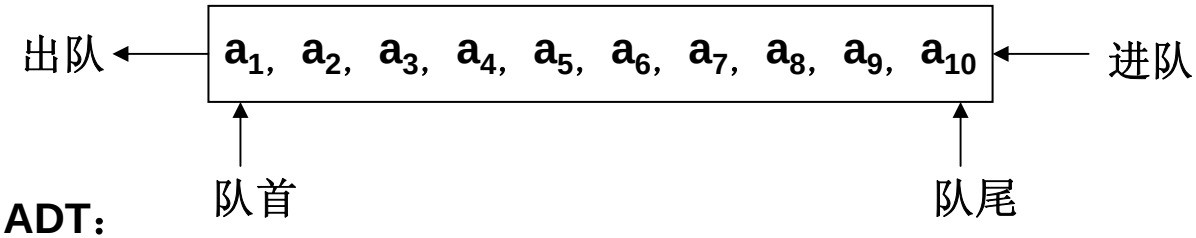
1、定义：

在表的一端进行插入，而在另一端进行删除的线性表。

队尾：进行插入的一端。

队首：进行删除的一端。

时间有序表：**FIFO** 特征的线性结构。



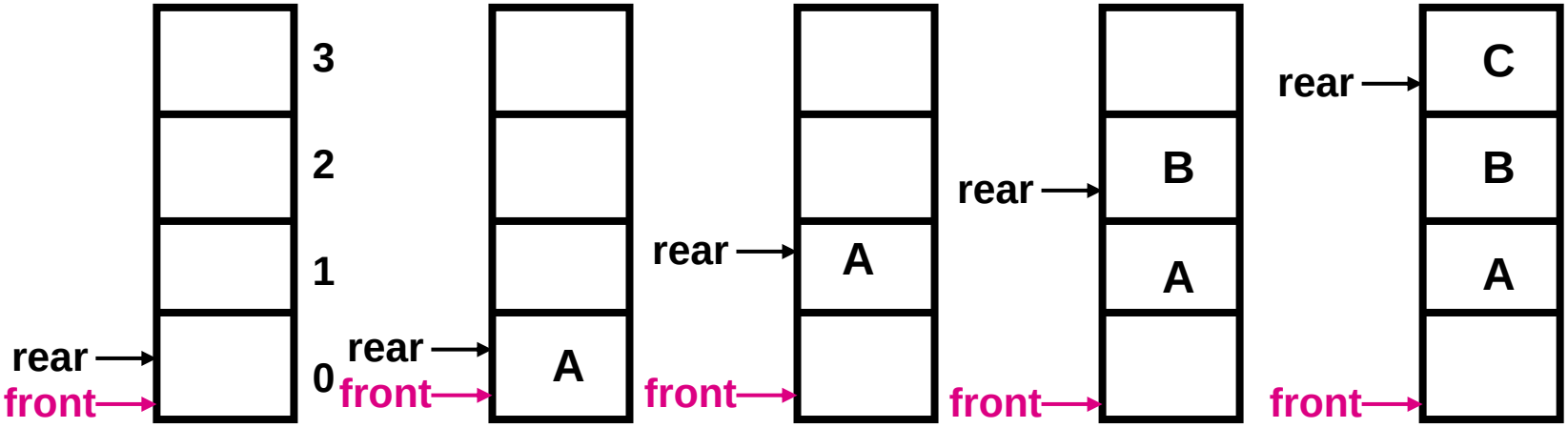
2、队列的 ADT：

```
Template <class Type> class Queue {
public:
 Queue( int =10 ); //  初始化时队列的空间大小或可放结点个数
 void EnQueue( const Type & item ); // 将 item 之值进队，进队之前须判是否队满。
 Type DeQueue( );      // 队首元素的出队且返回队首元素之值，操作前须判队空吗。
 Type Getfront( );    // 取队首元素的数据值返回，操作前须判队空吗。
 void MakeEmpty ( ); // 置空队。
 int IsEmpty( ) const;  // 队空为1，否则为 0
 int IsFull( ) const ;    //  队满为 1，否则为 0
}
```

3、队列

2、队列的表示：

· 顺序表示的队列—循环队列



空队

maxSize = 4

front == rear

是队空标志

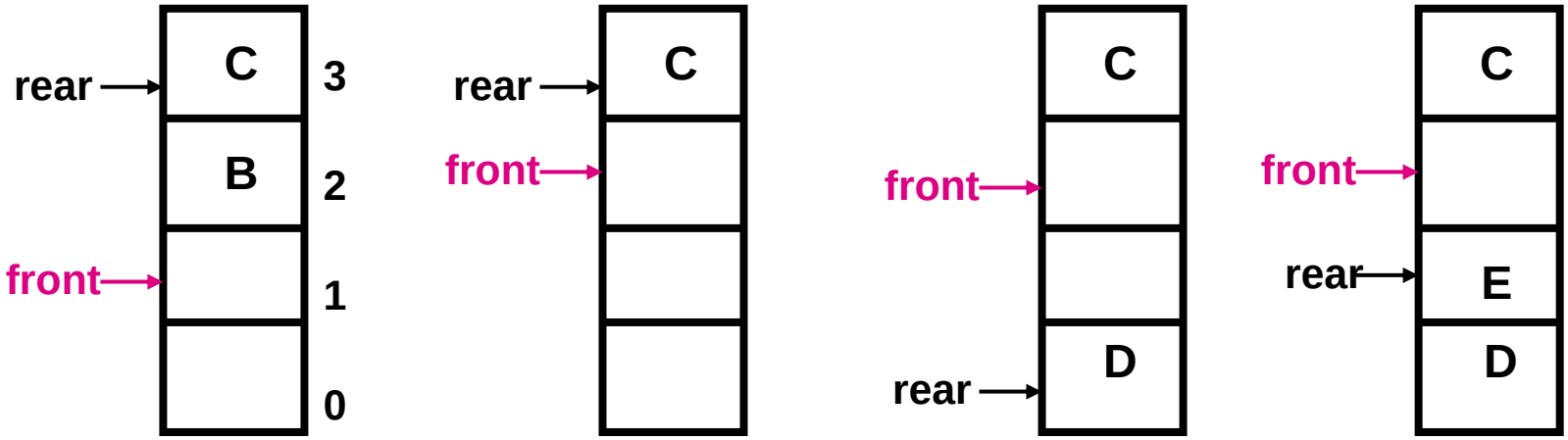
当 **A** 进队时，如仍设队首指针 **front** 和队尾指针 **rear** 指向真正的队首和队尾，则

front == rear

和队空标志矛盾。

因此，使 **front** 指向真正的队首的前一结点而 **rear**指向真正队尾(其它方案？)。

3、队列



空队

maxSize = 4

front == rear

是队空标志

当 **A** 进队时，如仍设队首指针 **front** 和队尾指针 **rear** 指向真正的队首和队尾，则

front == rear

和队空标志矛盾。

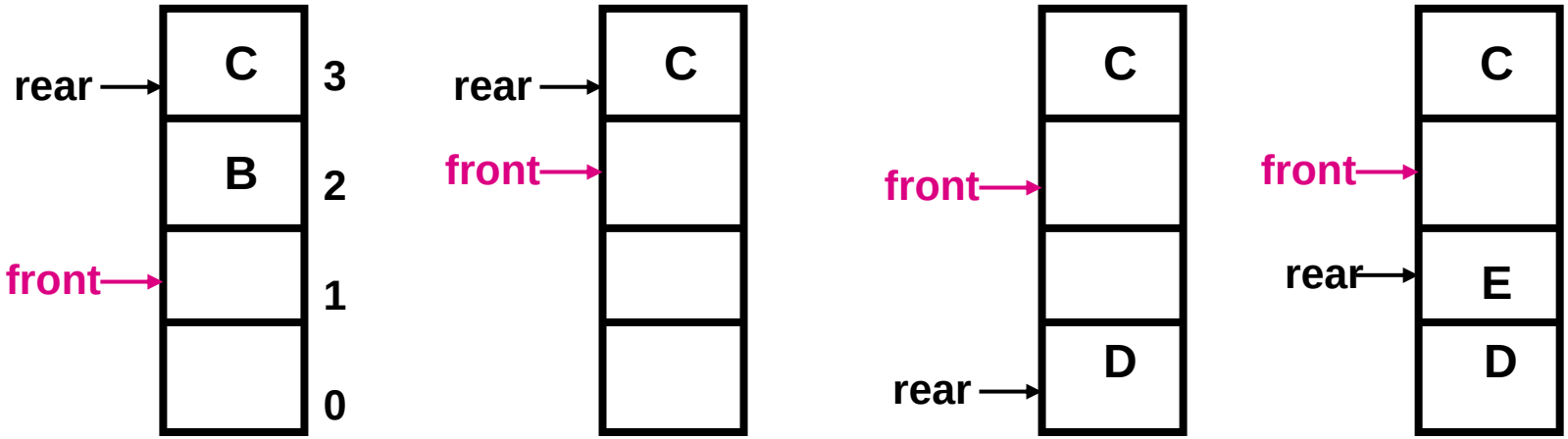
因此，使 **front** 指向真正的队首的前一结点而 **rear**指向真正队尾(其它方案？)。

当 **D** 进队时，队尾指针 **rear** 指向真正的队尾，由于此时 **0** 号单元已可以使用，所以

rear == 0

这就是所谓循环队列。认为具有最大下标的单元后面是最小下标的单元。

3、队列



空队

maxSize = 4

front == rear

是队空标志

当 **A** 进队时，如仍设队首指针 **front** 和队尾指针 **rear** 指向真正的队首和队尾，则

front == rear

和队空标志矛盾。

因此，使 **front** 指向真正的队首的前一结点而 **rear**指向真正队尾(其它方案？)。

当 **F** 进队时，队尾指针 **rear** 指向真正的队尾，这样就造成了如下情况：

front == rear

和队空标志矛盾。解决：

rear == front 就认为队列已满。

注意：单元未用足，少用一个单元。

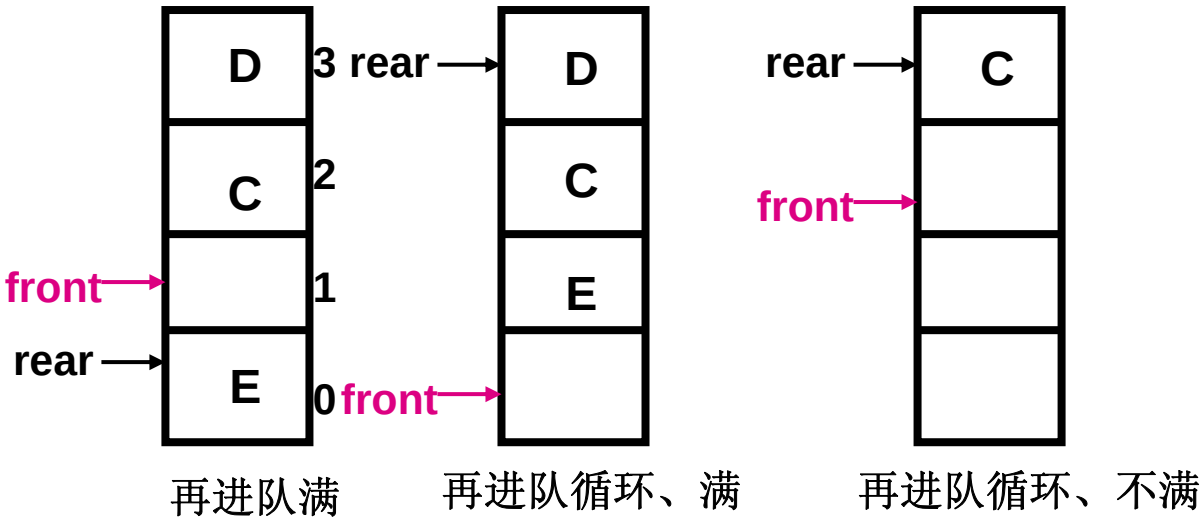
3、队列

- 出队时应先判队是否空：条件 rear == Q.front
  不空则出队，注意 front 是否会由最高下标跳至最低下标(循环)。
- 进队时应先判队是否满：条件 ((.rear + 1) % maxSize) == front
  不满则进队，注意 Q.rear 是否会由最高下标跳至最低下标(循环)。 。

3、队列

·基本操作的实现程序：**EnQueue (SqQueue &Q, QElemType e)**

```
Template <class Type>
void Queue < Type> :: EnQueue(  const Type &  item ) {
   assert(  ! IsFull );
   rear = ( rear +1 ) % maxSize;
   elements[rear] = item;
}
```

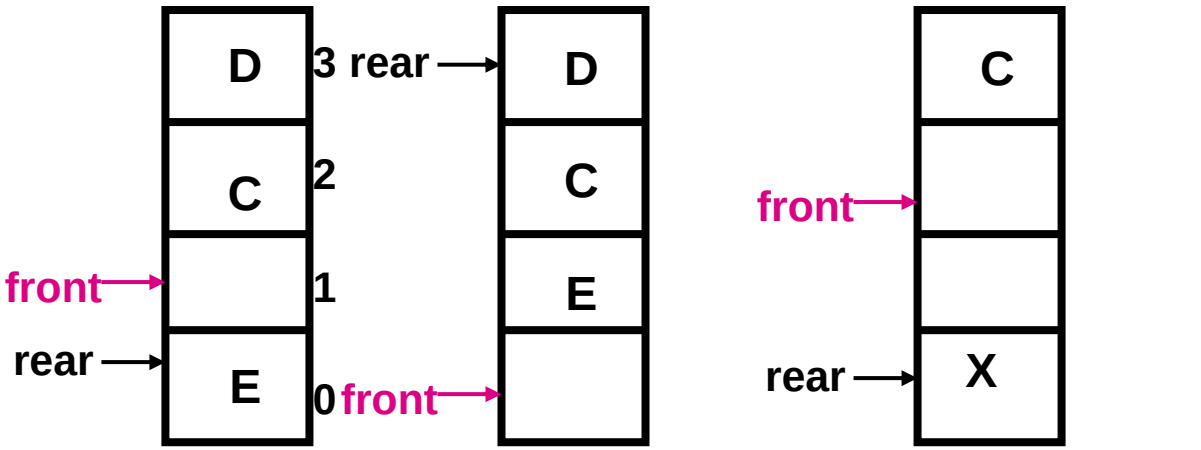


再进队满　　再进队循环、满　　再进队循环、不满

3、队列

·基本操作的实现程序：**EnQueue (SqQueue &Q, QElemType e)**

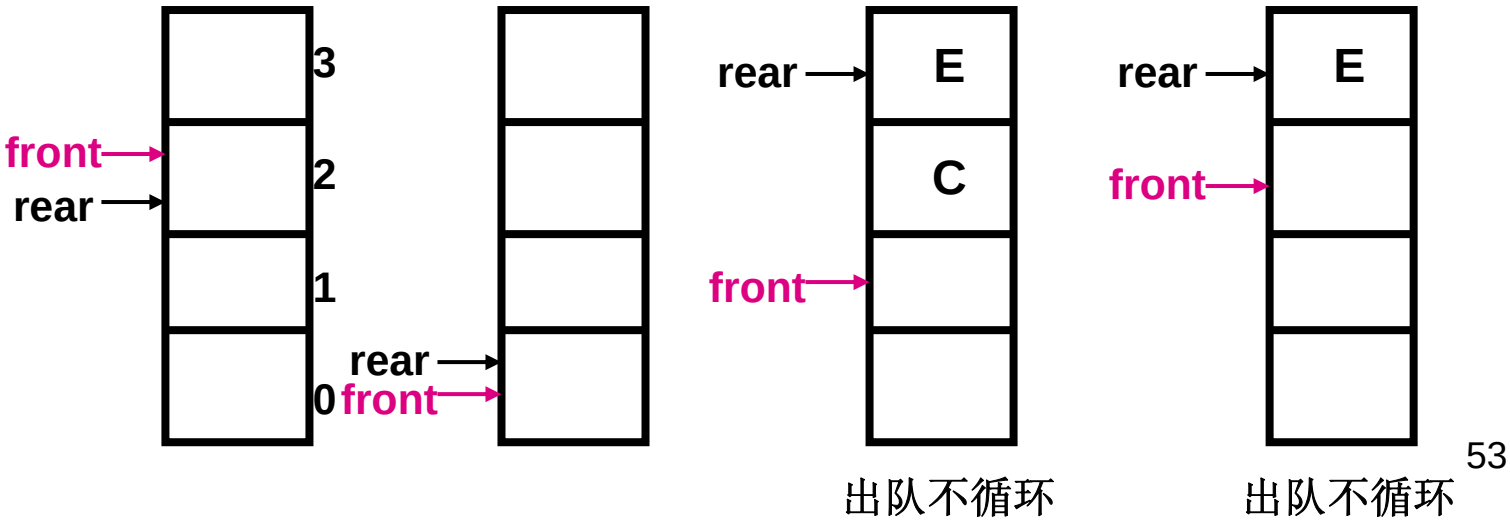
```
Template <class Type>
void Queue < Type> :: EnQueue(  const Type &  item ) {
   assert(  ! IsFull );
   rear = ( rear +1 ) % maxSize;
   elements[rear] = item;
}
```



3、队列

·基本操作的实现程序：**DeQueue ( )**

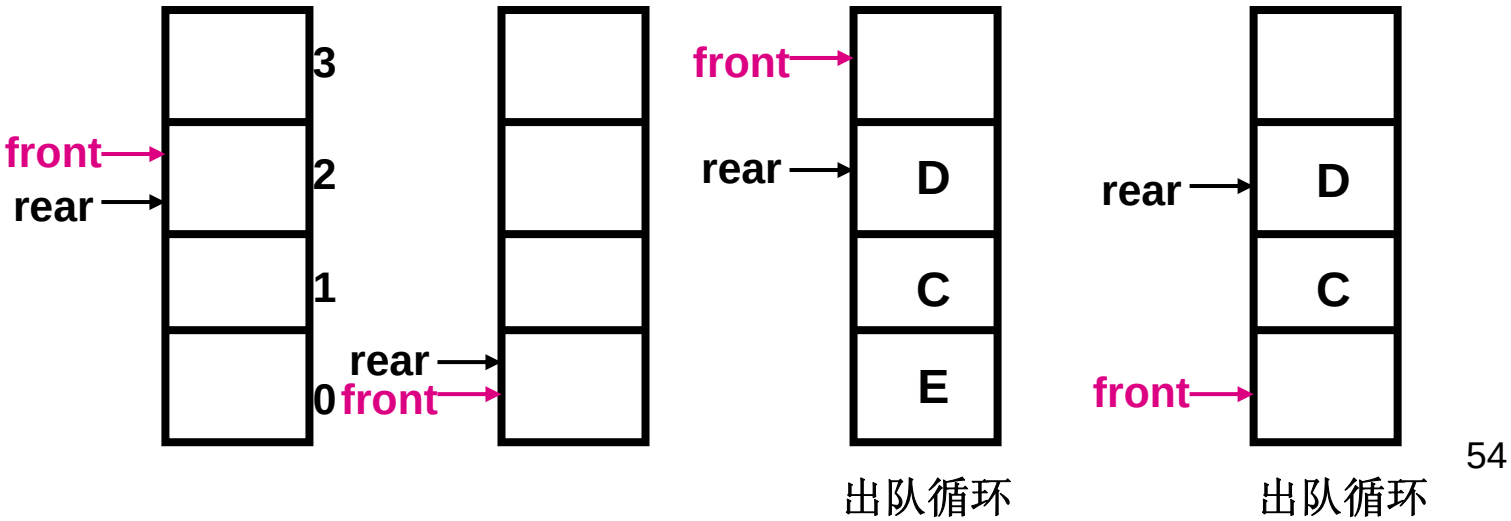
```
Template <class Type>
void Queue < Type> :: DeQueue( ) {
   assert( ! IsEmpty );
   front = ( front +1 ) % maxSize;
   return elements[front];
}
```



3、队列

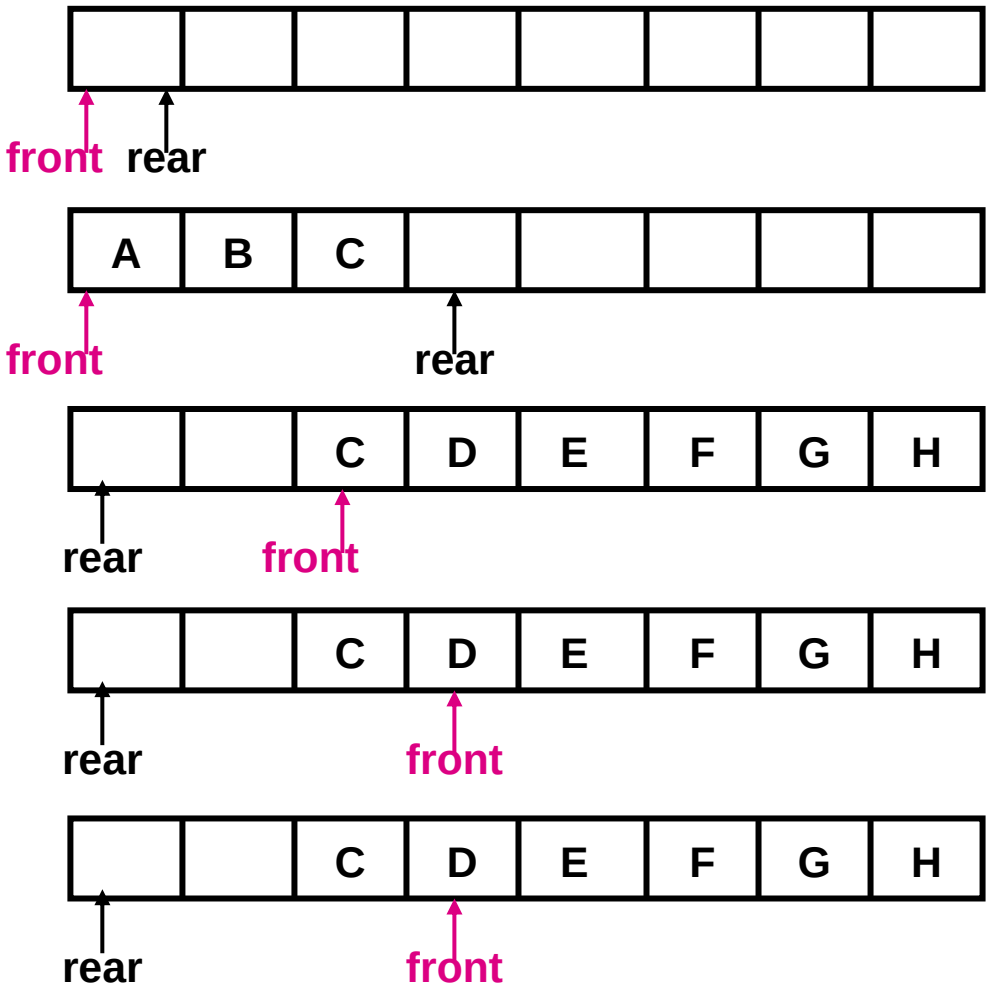
·基本操作的实现程序：**DeQueue ( )**

```
Template <class Type>
void Queue < Type> :: DeQueue(  ) {
   assert(  ! IsEmpty );
   front = ( front +1 ) % maxSize;
   return elements[front];
}
```



3、队列

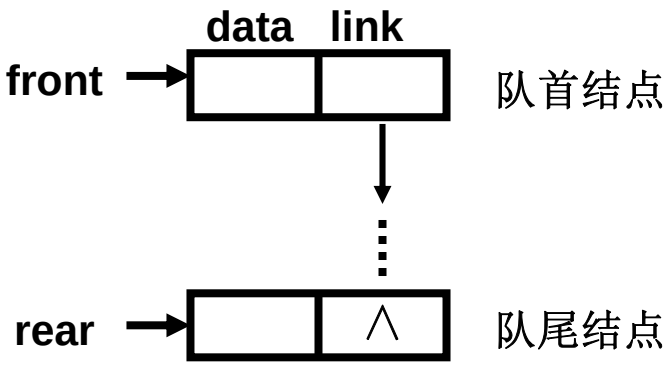
- 求队中结点的个数：(rear - front + maxSize) % maxSize
- 循环队列的应用实例：打印缓冲区的安排。



I、J、K 要进队不行，必须等待！！

3、队列

· 链接表示的队列：参照下图所示。其中 **front** 和 **rear** 分别是队首和队尾指针。它们指示着真正的队首和真正的队尾结点。

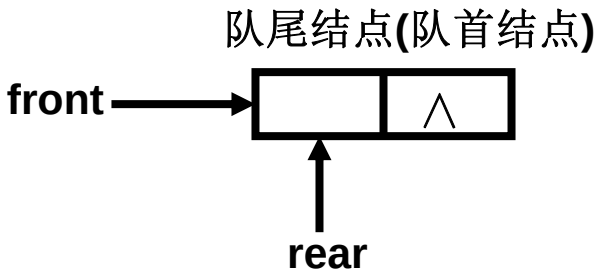


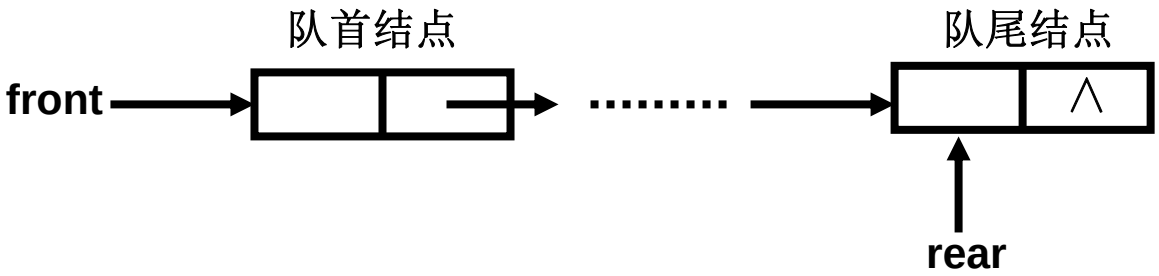
· 链接队列的操作：

3、队列

· 链接队列的操作：

front = rear = NULL

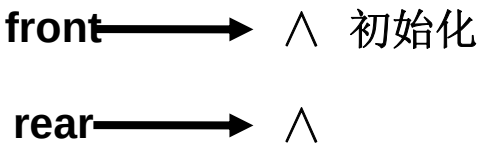


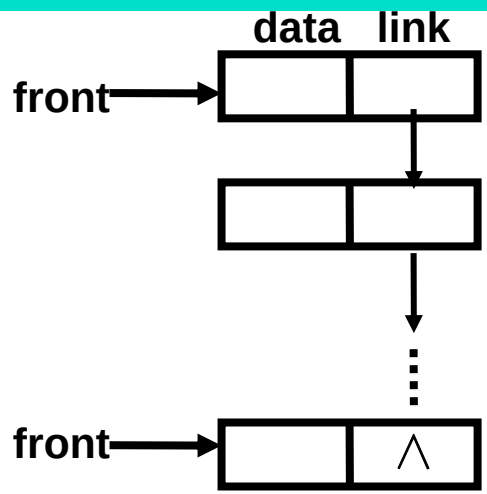


3、队列

·链接表示的队列：**front**、**rear** 分别是队列的队首和对尾指针。

```
Template <class Type> class Queue;
Template <class Type> class QueueNode {
friend class Queue<Type>;
private:
 Type data;
 StackNode <Type>  * link;
 StackNode ( Type d = 0; QueueNode <Type>  * l = NULL):data(d), link( l );
}
```

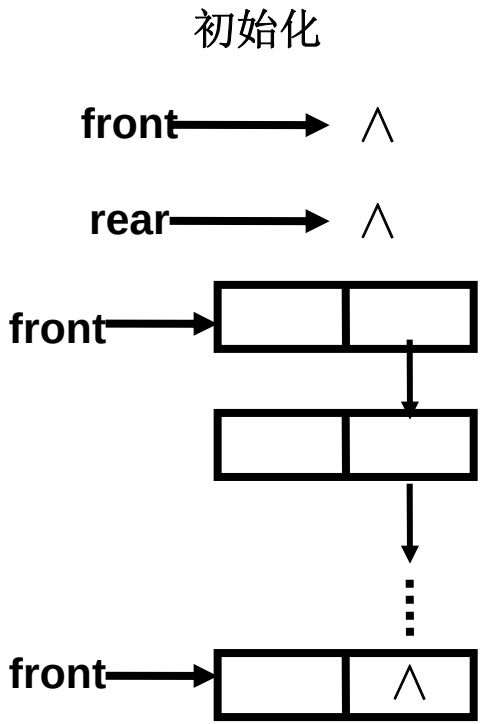




3、队列

· 链接表示的队列：**front**、**rear** 分别是队列的队首和对尾指针。

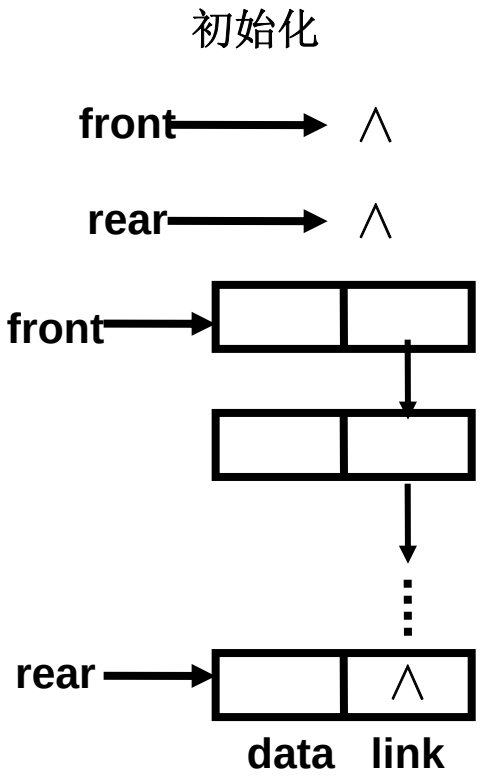
```
Template <class Type> class Queue {
public:
 Queue( ): rear(NULL), front( NULL)  { }
 ~Queue( );
 void EnQueue( const Type & item );
 Type DeQueue(  );
 Type GetFront(  );
 void MakeEmpty ( );
 int IsEmpty( ) const { return front == NULL; }   // 判队空吗
Private:
 Queue< Type> * front,* rear;
}
```



3、队列

· 链接表示的队列：进队操作，**front**、**rear** 分别是队列的队首和对尾指针。

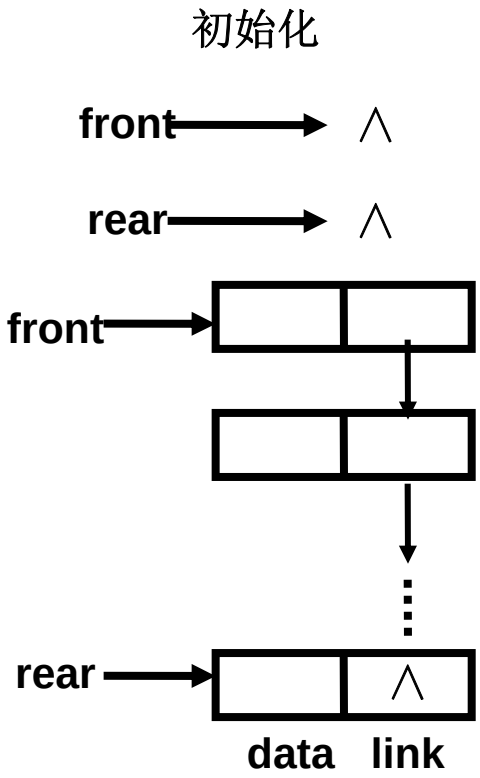
```
Template < class Type >
void Queue< Type > :: EnQueue( const Type & item ) {
    if ( front = = NULL )
        front = rear = new QueueNode < Type > ( item, NULL)
    else
        rear = rear->link = new QueueNode < Type > ( item, NULL)
 }
```



3、队列

·链接表示的队列：出队操作，**front**、**rear** 分别是队列的队首和对尾指针。

```
Template < class Type >
Type Queue< Type > :: DeQueue( ) {
   assert( ! IsEmpty( ) );
   QueueNode < Type > * p = front;
   Type retvaue = p->data;
    front =  front -> link;  delete p;
    return retvalue;
 }
```

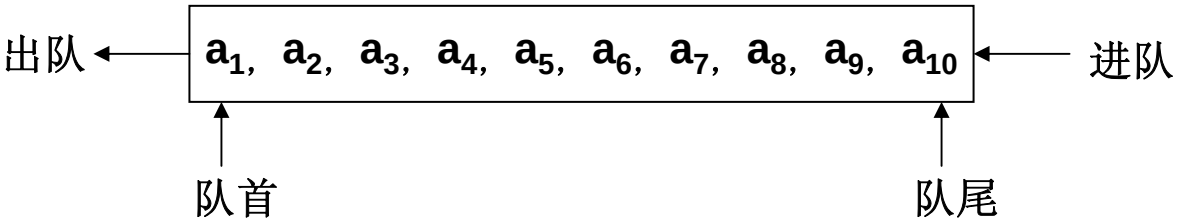


4、优先队列

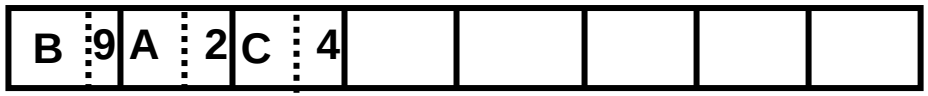
1、定义：

优先权有序表：具有特征高优先权的结点将先离开的线性结构。和到达时刻无关。

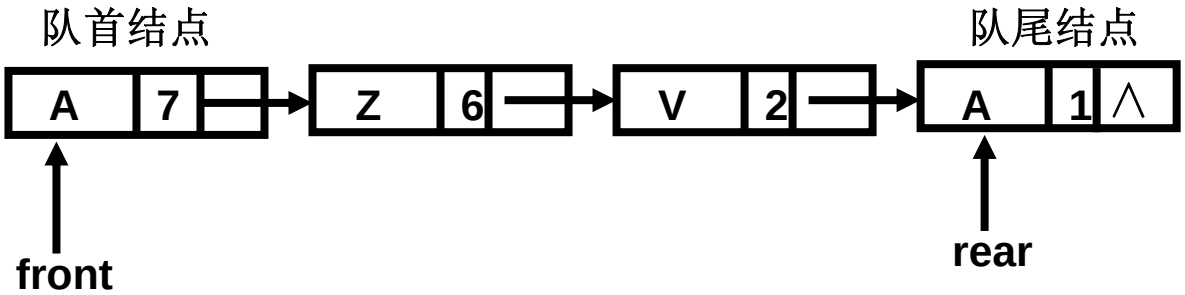
实现方法：结点中处包含数据场外，还有本结点的优先权数。



1。顺序存储的优先队列：用数组



2。用链接表：



4、优先队列

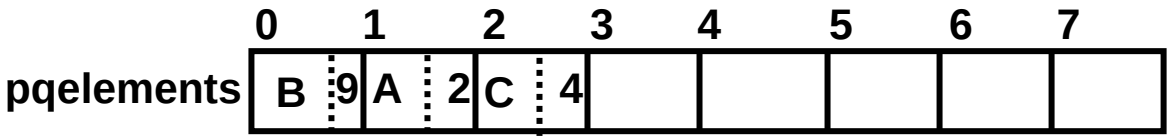
2、优先队列的 ADT：

```
# include  < assert.h >
# include  < iostream.h >
# include  < stdlib.h >
const int maxPQSize = 50

Template <class Type> class PQueue {
public:
 PQueue(  ); //  初始化时优先队列的空间
 ~PQueue(  ) { delete [  ] pqelements; }     // 置空队。
 void PQInsert ( const type & item );
 Type PQRemove ( );
 void MakeEmpty (  ) { count = 0; }     // 置空队。
 int IsEmpty( ) const { count = = 0; }    // 队空为1，否则为 0
 int IsFull(  ) const { return count = maxPQSize; } //  队满为 1，否则为 0
 int Length(  ) const { return count; }    //  队满为 1，否则为 0
private:
 Type * pqelements; // 存放结点的数组
  int count; // 结点个数计数器
}
```

4、优先队列

2、优先队列的删除结点的操作的实现：

```
Template <class Type>
Type Pqueue < Type > :: PQRemove ( ) {
  assert( ! IsEmpty( );
  Type min = pqelements[ 0 ];
  int minindex = 0;
  for ( int i = 1; i < count;  i++ )
     if ( pqelements[ i ] < min { min = pqelements[ i ]; minindex = i; }
 pqelements[ minindex ] = pqelements[ count  - 1 ];
 count- -;
 return min;
}
```



注意：**count == 3**