

4-1 改写顺序栈的进栈成员函数 `Push(x)`，要求当栈满时执行一个 `stackFull()` 操作进行栈满处理。其功能是：动态创建一个比原来的栈数组大二倍的新数组，代替原来的栈数组，原来栈数组中的元素占据新数组的前 `MaxSize` 位置。

【解答】`template<class Type>void stack<Type>::push(const Type & item){`

`if(isFull()) stackFull();` //栈满，做溢出处理

`elements[++top] = item;` //进栈

`}`

`template<class Type> void stack<Type>::stackFull(){`

`Type * temp = new Type [3 * maxSize];` //创建体积大二倍的数组

`for(int i = 0; i <= top; i++)` //传送原数组的数据

`temp[i] = elements[i];`

`delete [] elements;` //删去原数组

`maxSize *= 3;` //数组最大体积增长二倍

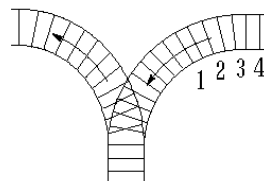
`elements = temp;` //新数组成为栈的数组空间

`}`

4-2 铁路进行列车调度时，常把站台设计成栈式结构的站台，如右图所示。试问：

(1) 设有编号为 1,2,3,4,5,6 的六辆列车，顺序开入栈式结构的站台，则可能的出栈序列有多少种？

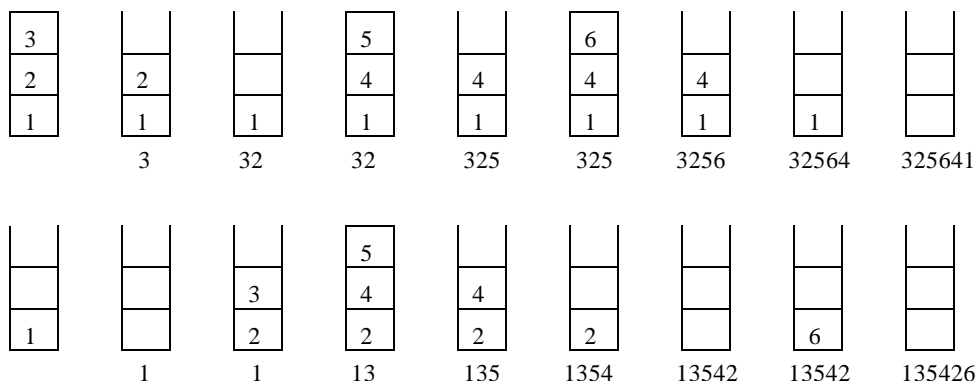
(2) 若进站的六辆列车顺序如上所述，那么是否能够得到 435612, 325641, 154623 和 135426 的出站序列，如果不能，说明为什么不能；如果能，说明如何得到(即写出"进栈"或"出栈"的序列)。



【解答】

(1) 可能的不同出栈序列有 $(1/(6+1)) * C_{12}^6 = 132$ 种。

(2) 不能得到 435612 和 154623 这样的出栈序列。因为若在 4, 3, 5, 6 之后再 1, 2 出栈，则 1, 2 必须一直在栈中，此时 1 先进栈，2 后进栈，2 应压在 1 上面，不可能 1 先于 2 出栈。154623 也是这种情况。出栈序列 325641 和 135426 可以得到。



4-3 试证明：若借助栈可由输入序列 $1, 2, 3, \dots, n$ 得到一个输出序列 $p_1, p_2, p_3, \dots, p_n$ (它是输入序列的某一种排列)，则在输出序列中不可能出现以下情况，即存在 $i < j < k$ ，使得 $p_j < p_k < p_i$ 。(提示：用反证法)

【解答】

因为借助栈由输入序列 $1, 2, 3, \dots, n$ ，可得到输出序列 $p_1, p_2, p_3, \dots, p_n$ ，如果存在下标 i, j, k ，满足 $i < j < k$ ，那么在输出序列中，可能出现如下 5 种情况：

① i 进栈, i 出栈, j 进栈, j 出栈, k 进栈, k 出栈。此时具有最小值的排在最前面 p_i 位置, 具有中间值的排在其后 p_j 位置, 具有最大值的排在 p_k 位置, 有 $p_i < p_j < p_k$, 不可能出现 $p_j < p_k < p_i$ 的情形;

② i 进栈, i 出栈, j 进栈, k 进栈, k 出栈, j 出栈。此时具有最小值的排在最前面 p_i 位置, 具有最大值的排在 p_j 位置, 具有中间值的排在最后 p_k 位置, 有 $p_i < p_k < p_j$, 不可能出现 $p_j < p_k < p_i$ 的情形;

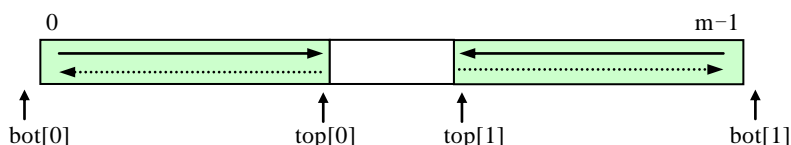
③ i 进栈, j 进栈, j 出栈, i 出栈, k 进栈, k 出栈。此时具有中间值的排在最前面 p_i 位置, 具有最小值的排在其后 p_j 位置, 有 $p_j < p_i < p_k$, 不可能出现 $p_j < p_k < p_i$ 的情形;

④ i 进栈, j 进栈, j 出栈, k 进栈, k 出栈, i 出栈。此时具有中间值的排在最前面 p_i 位置, 具有最大值的排在其后 p_j 位置, 具有最小值的排在 p_k 位置, 有 $p_k < p_i < p_j$, 也不可能出现 $p_j < p_k < p_i$ 的情形;

⑤ i 进栈, j 进栈, k 进栈, k 出栈, j 出栈, i 出栈。此时具有最大值的排在最前面 p_i 位置, 具有中间值的排在其后 p_j 位置, 具有最小值的排在 p_k 位置, 有 $p_k < p_j < p_i$, 也不可能出现 $p_j < p_k < p_i$ 的情形;

4-4 将编号为 0 和 1 的两个栈存放于一个数组空间 $V[m]$ 中, 栈底分别处于数组的两端。当第 0 号栈的栈顶指针 $\text{top}[0]$ 等于 -1 时该栈为空, 当第 1 号栈的栈顶指针 $\text{top}[1]$ 等于 m 时该栈为空。两个栈均从两端向中间增长。当向第 0 号栈插入一个新元素时, 使 $\text{top}[0]$ 增 1 得到新的栈顶位置, 当向第 1 号栈插入一个新元素时, 使 $\text{top}[1]$ 减 1 得到新的栈顶位置。当 $\text{top}[0]+1 == \text{top}[1]$ 时或 $\text{top}[0] == \text{top}[1]-1$ 时, 栈空间满, 此时不能再向任一栈加入新的元素。试定义这种双栈(Double Stack)结构的类定义, 并实现判栈空、判栈满、插入、删除算法。

【解答】



双栈的类定义如下:

```
#include <assert.h>
```

```
template <class Type> class DblStack {                                //双栈的类定义
private:
    int top[2], bot[2];                                              //双栈的栈顶指针和栈底指针
    Type *elements;                                                 //栈数组
    int m;                                                          //栈最大可容纳元素个数
public:
    DblStack ( int sz =10 );                                         //初始化双栈, 总体积 m 的默认值为 10
    ~DblStack ( ) { delete [ ] elements; }                          //析构函数
    void DblPush ( const Type& x, int i );                          //把 x 插入到栈 i 的栈顶
    int DblPop ( int i );                                           //退掉位于栈 i 栈顶的元素
    Type * DblGetTop ( int i );                                     //返回栈 i 栈顶元素的值
    int IsEmpty ( int i ) const { return top[i] == bot[i]; }        //判栈 i 空否, 空则返回 1, 否则返回 0
    int IsFull ( ) const { return top[0]+1 == top[1]; }            //判栈满否, 满则返回 1, 否则返回 0
    void MakeEmpty ( int i );                                       //清空栈 i 的内容
}

template <class Type> DblStack<Type> :: DblStack ( int sz ) : m(sz), top[0] (-1), bot[0](-1), top[1](sz), bot[1](sz) {
//建立一个最大尺寸为 sz 的空栈, 若分配不成功则错误处理。
    elements = new Type[m];                                         //创建栈的数组空间
    assert ( elements != NULL );                                    //断言: 动态存储分配成功与否
}
```

```
template <class Type> void DblStack<Type>::DblPush ( const Type& x, int i ) {
```

```
//如果 IsFull ( ), 则报错; 否则把 x 插入到栈 i 的栈顶
```

```
    assert ( !IsFull ( ) );
```

```
//断言: 栈满则出错处理, 停止执行
```

```
    if ( i == 0 ) elements[ ++top[0] ] = x;
```

```
//栈 0 情形: 栈顶指针先加 1, 然后按此地址进栈
```

```
    else elements[ --top[1] ] = x;
```

```
//栈 1 情形: 栈顶指针先减 1, 然后按此地址进栈
```

```
}
```

```
template <class Type> int DblStack<Type>::DblPop ( int i ) {
```

```
//如果 IsEmpty ( i ), 则不执行退栈, 返回 0; 否则退掉位于栈 i 栈顶的元素, 返回 1
```

```
    if ( IsEmpty ( i ) ) return 0;
```

```
//判栈空否, 若栈空则函数返回 0
```

```
    if ( i == 0 ) top[0]--;
```

```
//栈 0 情形: 栈顶指针减 1
```

```
    else top[1]++;
```

```
//栈 1 情形: 栈顶指针加 1
```

```
    return 1;
```

```
}
```

```
template <class Type> Type * DblStack<Type>::DblGetTop ( int i ) {
```

```
//若栈不空则函数返回该栈栈顶元素的地址。
```

```
    if ( IsEmpty ( i ) ) return NULL;
```

```
//判栈 i 空否, 若栈空则函数返回空指针
```

```
    return& elements[ top[i] ];
```

```
//返回栈顶元素的值
```

```
}
```

```
template <class Type> void MakeEmpty ( int i ) {
```

```
    if ( i == 0 ) top[0] = bot[0] = -1;
```

```
    else top[1] = bot[1] = m;
```

```
}
```

4-5 写出下列中缀表达式的后缀形式:

(1) $A * B * C$

(2) $- A + B - C + D$

(3) $A * - B + C$

(4) $(A + B) * D + E / (F + A * D) + C$

(5) $A \&\& B || ! (E > F)$ /*注: 按 C++ 的优先级*/

(6) $!(A \&\& !((B < C) || (C > D))) || (C < E)$

【解答】

(1) $A B * C *$

(2) $A - B + C - D +$

(3) $A B - * C +$

(4) $A B + D * E F A D * + / + C +$

(5) $A B \&\& E F > ! ||$

(6) $A B C < C D > || ! \&\& ! C E < ||$

4-6 根据课文中给出的优先级, 回答以下问题:

(1) 在函数 postfix 中, 如果表达式 e 含有 n 个运算符和分界符, 问栈中最多可存入多少个元素?

(2) 如果表达式 e 含有 n 个运算符, 且括号嵌套的最大深度为 6 层, 问栈中最多可存入多少个元素?

【解答】

(1) 在函数 postfix 中, 如果表达式 e 含有 n 个运算符和分界符, 则可能的运算对象有 $n+1$ 个。因此在利用后缀表达式求值时所用到的运算对象栈中最多可存入 $n+1$ 个元素。

(2) 同上。

4-7 设表达式的中缀表示为 $a * x - b / x \uparrow 2$, 试利用栈将它改为后缀表示 $ax * bx2 \uparrow / -$ 。写出转换过程中栈的变化。

【解答】

若设当前扫描到的运算符 ch 的优先级为 $icp(ch)$, 该运算符进栈后的优先级为 $isp(ch)$, 则可规定各个算术运算符的优先级如下表所示。

运算符	;	(^	*, /, %	+, -)
isp	0	1	7	5	3	8
icp	0	8	6	4	2	1

isp 也叫做栈内 (in stack priority) 优先数, icp 也叫做栈外 (in coming priority) 优先数。当刚扫描到的运算符 ch 的 $icp(ch)$ 大于 $isp(stack)$ 时, 则 ch 进栈; 当刚扫描到的运算符 ch 的 $icp(ch)$ 小于 $isp(stack)$ 时, 则位于栈顶的运算符退栈并输出。从表中可知, $icp("(")$ 最高, 但当 $("$ 进栈后, $isp("(")$ 变得极低。其它运算符进入栈中后优先数都升 1, 这样可体现在中缀表达式中相同优先级的运算符自左向右计算的要求。运算符优先数相等的情况只出现在括号配对 $)$ 或栈底的 $;$ 号与输入流最后的 $;$ 号配对时。前者将连续退出位于栈顶的运算符, 直到遇到 $("$ 为止。然后将 $("$ 退栈以对消括号, 后者将结束算法。

步序	扫描项	项类型	动作	栈的变化	输出
0			☞ ';' 进栈, 读下一符号	;	
1	a	操作数	☞ 直接输出, 读下一符号	;	A
2	*	操作符	☞ $isp(';') < icp('*')$, 进栈, 读下一符号	;*	A
3	x	操作数	☞ 直接输出, 读下一符号	;*	ax
4	-	操作符	☞ $isp('*') > icp('-')$, 退栈输出	;	ax*
			☞ $isp(';') < icp('-')$, 进栈, 读下一符号	;-	ax*
5	b	操作数	☞ 直接输出, 读下一符号	;-	ax*b
6	/	操作符	☞ $isp('-') < icp('/')$, 进栈, 读下一符号	;-/	ax*b
7	x	操作数	☞ 直接输出, 读下一符号	;-/	ax*b x
8	↑	操作符	☞ $isp('/') < icp('↑')$, 进栈, 读下一符号	;-/ ↑	ax*b x
9	2	操作数	☞ 直接输出, 读下一符号	;-/ ↑	ax*b x 2
10	;	操作符	☞ $isp('↑') > icp(';')$, 退栈输出	;-/	ax*b x 2 ↑
			☞ $isp('/') > icp(';')$, 退栈输出	;-	ax*b x 2 ↑ /
			☞ $isp('-') > icp(';')$, 退栈输出	;	ax*b x 2 ↑ / -
			☞ 结束		

4-8 试利用运算符优先数法, 画出对如下中缀算术表达式求值时运算符栈和运算对象栈的变化。

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

【解答】

设在表达式计算时各运算符的优先规则如上一题所示。因为直接对中缀算术表达式求值时必须使用两个栈, 分别对运算符和运算对象进行处理, 假设命名运算符栈为 OPTR (operator 的缩写), 运算对象栈为 OPND (operand 的缩写), 下面给出对中缀表达式求值的一般规则:

(1) 建立并初始化 OPTR 栈和 OPND 栈, 然后在 OPTR 栈中压入一个 $;$;

(2) 从头扫描中缀表达式, 取一字符送入 ch 。

(3) 当 ch 不等于 “;” 时, 执行以下工作, 否则结束算法。此时在 OPND 栈的栈顶得到运算结果。

① 如果 ch 是运算对象, 进 OPND 栈, 从中缀表达式取下一字符送入 ch ;

② 如果 ch 是运算符, 比较 ch 的优先级 $icp(ch)$ 和 OPTR 栈顶运算符 $isp(OPTR)$ 的优先级:

☞ 若 $icp(ch) > isp(OPTR)$, 则 ch 进 OPTR 栈, 从中缀表达式取下一字符送入 ch ;

☞ 若 $icp(ch) < isp(OPTR)$, 则从 OPND 栈退出一个运算符作为第 2 操作数 a_2 , 再退出一个运算符作为第 1 操作数 a_1 , 从 OPTR 栈退出一个运算符 θ 形成运算指令 $(a_1)\theta(a_2)$, 执行结果进 OPND 栈;

☞ 若 $icp(ch) == isp(OPTR)$ 且 $ch == "("$, 则从 OPTR 栈退出栈顶的 “(”, 对消括号, 然后从中缀表达式取下一字符送入 ch ;

根据以上规则, 给出计算 $a + b * (c - d) - e \uparrow f / g$ 时两个栈的变化。

步序	扫描项	项类型	动作	OPND 栈变化	OPTR 栈变化
0			☞ OPTR 栈与 OPND 栈初始化, ‘;’ 进 OPTR 栈, 取第一个符号		;
1	a	操作数	☞ a 进 OPND 栈, 取下一符号	a	;
2	+	操作符	☞ $icp(‘+’) > isp(‘;’)$, 进 OPTR 栈, 取下一符号	a	; +
3	b	操作数	☞ b 进 OPND 栈, 取下一符号	a b	; +
4	*	操作符	☞ $icp(‘*’) > isp(‘+’)$, 进 OPTR 栈, 取下一符号	a b	; + *
5	(操作符	☞ $icp(‘(’) > isp(‘*’)$, 进 OPTR 栈, 取下一符号	a b	; + * (
6	c	操作数	☞ c 进 OPND 栈, 取下一符号	a b c	; + * (
7	-	操作符	☞ $icp(‘-’) > isp(‘(’)$, 进 OPTR 栈, 取下一符号	a b	; + * (-
8	d	操作数	☞ d 进 OPND 栈, 取下一符号	a b c d	; + * (-
9)	操作符	☞ $icp(‘)’) < isp(‘-’)$, 退 OPND 栈 ‘d’, 退 OPND 栈 ‘c’, 退 OPTR 栈 ‘-’, 计算 $c - d \rightarrow s_1$, 结果进 OPND 栈	a b s_1	; + * (
10	同上	同上	☞ $icp(‘)’) == isp(‘(’)$, 退 OPTR 栈 ‘(’, 对消括号, 取下一符号	a b s_1	; + *
11	-	操作符	☞ $icp(‘-’) < isp(‘*’)$, 退 OPND 栈 ‘ s_1 ’, 退 OPND 栈 ‘b’, 退 OPTR 栈 ‘*’, 计算 $b * s_1 \rightarrow s_2$, 结果进 OPND 栈	a s_2	; +
12	同上	同上	☞ $icp(‘-’) < isp(‘+’)$, 退 OPND 栈 ‘ s_2 ’, 退 OPND 栈 ‘a’, 退 OPTR 栈 ‘+’, 计算 $a * s_2 \rightarrow s_3$, 结果进 OPND 栈	s_3	;
13	同上	同上	☞ $icp(‘-’) > isp(‘;’)$, 进 OPTR 栈, 取下一符号	s_3	; -
14	e	操作数	☞ e 进 OPND 栈, 取下一符号	s_3 e	; -
15	↑	操作符	☞ $icp(‘↑’) > isp(‘-’)$, 进 OPTR 栈, 取下一符号	s_3 e	; - ↑
16	f	操作数	☞ f 进 OPND 栈, 取下一符号	s_3 e f	; - ↑
17	/	操作符	☞ $icp(‘/’) < isp(‘↑’)$, 退 OPND 栈 ‘f’, 退 OPND 栈 ‘e’, 退 OPTR 栈 ‘↑’, 计算 $e \uparrow f \rightarrow s_4$, 结果进 OPND 栈	s_3 s_4	; -
18	同上	同上	☞ $icp(‘/’) > isp(‘-’)$, 进 OPTR 栈, 取下一符号	s_3 s_4	; - /
19	g	操作数	☞ g 进 OPND 栈, 取下一符号	s_3 s_4 g	; - /
20	;	操作符	☞ $icp(‘;’) < isp(‘/’)$, 退 OPND 栈 ‘g’, 退 OPND 栈 ‘ s_4 ’, 退 OPTR 栈 ‘/’, 计算 $s_4 / g \rightarrow s_5$, 结果进 OPND 栈	s_3 s_5	; -
21	同上	同上	☞ $icp(‘;’) < isp(‘-’)$, 退 OPND 栈 ‘ s_5 ’, 退 OPND 栈 ‘ s_3 ’, 退 OPTR 栈 ‘-’, 计算 $s_3 - s_5 \rightarrow s_6$, 结果进 OPND 栈	s_6	;

22	同上	同上	☞ <code>icp(' ; ') == isp(' ; ')</code> , 退 OPND 栈 's ₆ ', 结束		;
----	----	----	--	--	---

4-9 假设以数组 $Q[m]$ 存放循环队列中的元素, 同时以 `rear` 和 `length` 分别指示环形队列中的队尾位置和队列中所含元素的个数。试给出该循环队列的队空条件和队满条件, 并写出相应的插入(enqueue)和删除(dlqueue)元素的操作。

【解答】

循环队列类定义

```
#include <assert.h>

template <class Type> class Queue {                                //循环队列的类定义
public:
    Queue ( int=10 );
    ~Queue () { delete [ ] elements; }
    void EnQueue ( Type & item );
    Type DeQueue ();
    Type GetFront ();
    void MakeEmpty () { length = 0; }                               //置空队列
    int IsEmpty () const { return length == 0; }                    //判队列空否
    int IsFull () const { return length == maxSize; }               //判队列满否

private:
    int rear, length;                                              //队尾指针和队列长度
    Type *elements;                                                //存放队列元素的数组
    int maxSize;                                                  //队列最大可容纳元素个数
}
```

构造函数

```
template <class Type>
Queue<Type>:: Queue ( int sz ) : rear (maxSize-1), length (0), maxSize (sz) {
//建立一个最大具有 maxSize 个元素的空队列。
    elements = new Type[maxSize];                                //创建队列空间
    assert ( elements != 0 );                                     //断言: 动态存储分配成功与否
}
```

插入函数

```
template<class Type>
void Queue<Type>:: EnQueue ( Type &item ) {
    assert ( ! IsFull () );                                       //判队列是否不满, 满则出错处理
    length++;                                                      //长度加 1
    rear = ( rear +1 ) % maxSize;                                  //队尾位置进 1
    elements[rear] = item;                                         //进队列
}
```

删除函数

```
template<class Type>
Type Queue<Type>:: DeQueue () {
    assert ( ! IsEmpty () );                                       //判断队列是否不空, 空则出错处理
    length--;                                                      //队列长度减 1
    return elements[(rear-length+maxSize) % maxSize];             //返回原队头元素值
}
```

```
}
```

读取队头元素值函数

```
template<class Type>
Type Queue<Type>::GetFront () {
    assert ( ! IsEmpty ( ) );
    return elements[(rear-length+1+maxSize) % maxSize];    //返回队头元素值
}
```

4-10 假设以数组 $Q[m]$ 存放循环队列中的元素，同时设置一个标志 tag ，以 $tag == 0$ 和 $tag == 1$ 来区别在队头指针($front$)和队尾指针($rear$)相等时，队列状态为“空”还是“满”。试编写与此结构相应的插入($enqueue$)和删除($dequeue$)算法。

【解答】

循环队列类定义

```
#include <assert.h>

template <class Type> class Queue {    //循环队列的类定义
public:
    Queue ( int=10 );
    ~Queue ( ) { delete [ ] Q; }
    void EnQueue ( Type & item );
    Type DeQueue ( );
    Type GetFront ( );
    void MakeEmpty ( ) { front = rear = tag = 0; }    //置空队列
    int IsEmpty ( ) const { return front == rear && tag == 0; }    //判队列空否
    int IsFull ( ) const { return front == rear && tag == 1; }    //判队列满否

private:
    int rear, front, tag;    //队尾指针、队头指针和队满标志
    Type *Q;    //存放队列元素的数组
    int m;    //队列最大可容纳元素个数
}
```

构造函数

```
template <class Type>
Queue<Type>::Queue ( int sz ) : rear (0), front (0), tag(0), m (sz) {
    //建立一个最大具有 m 个元素的空队列。
    Q = new Type[m];    //创建队列空间
    assert ( Q != 0 );    //断言：动态存储分配成功与否
}
```

插入函数

```
template<class Type>
void Queue<Type>::EnQueue ( Type &item ) {
    assert ( ! IsFull ( ) );    //判队列是否不满，满则出错处理
    rear = ( rear + 1 ) % m;    //队尾位置进 1，队尾指针指示实际队尾位置
    Q[rear] = item;    //进队列
    tag = 1;    //标志改 1，表示队列不空
}
```

删除函数

```
template<class Type>
Type Queue<Type>::DeQueue () {
    assert ( ! IsEmpty ( ) );           //判断队列是否不空，空则出错处理
    front = ( front + 1 ) % m;          //队头位置进 1，队头指针指示实际队头的前一位置
    tag = 0;                            //标志改 0，表示栈不满
    return Q[front];                    //返回原队头元素的值
}
```

读取队头元素函数

```
template<class Type>
Type Queue<Type>::GetFront () {
    assert ( ! IsEmpty ( ) );           //判断队列是否不空，空则出错处理
    return Q[(front + 1) % m];          //返回队头元素的值
}
```

4-11 若使用循环链表来表示队列，p 是链表中的一个指针。试基于此结构给出队列的插入(enqueue)和删除(dequeue)算法，并给出 p 为何值时队列空。

【解答】

链式队列的类定义

```
template <class Type> class Queue;           //链式队列类的前视定义

template <class Type> class QueueNode {      //链式队列结点类定义
friend class Queue<Type>;
private:
    Type data;                             //数据域
    QueueNode<Type> *link;                 //链域
public:
    QueueNode ( Type d = 0, QueueNode *l = NULL ) : data (d), link (l) { } //构造函数
};

template <class Type> class Queue {          //链式队列类定义
public:
    Queue () : p ( NULL ) { }               //构造函数
    ~Queue ();                              //析构函数
    void EnQueue ( const Type & item );      //将 item 加入到队列中
    Type DeQueue ();                        //删除并返回队头元素
    Type GetFront ();                      //查看队头元素的值
    void MakeEmpty ();                     //置空队列，实现与~Queue () 相同
    int IsEmpty () const { return p == NULL; } //判队列空否
private:
    QueueNode<Type> *p;                    //队尾指针（在循环链表中）
};
```

队列的析构函数

```
template <class Type> Queue<Type>::~~Queue () { //队列的析构函数
```



```

QueueNode<Type> *s;
while ( p != NULL ) { s = p;  p = p->link;  delete s; }//逐个删除队列中的结点
}

```

队列的插入函数

```

template <class Type> void Queue<Type>::EnQueue ( const Type & item ) {
    if ( p == NULL ) { //队列空, 新结点成为第一个结点
        p = new QueueNode<Type> ( item, NULL );  p->link = p;
    }
    else { //队列不空, 新结点链入 p 之后
        QueueNode<Type> *s = new QueueNode<Type> ( item, NULL );
        s->link = p->link;  p = p->link = s; //结点 p 指向新的队尾
    }
}

```

队列的删除函数

```

template <class Type> Type Queue<Type>::DeQueue ( ) {
    if ( p == NULL ) { cout << "队列空, 不能删除!" << endl;  return 0; }
    QueueNode<Type> *s = p; //队头结点为 p 后一个结点
    p->link = s->link; //重新链接, 将结点 s 从链中摘下
    Type retvalue = s->data;  delete s; //保存原队头结点中的值, 释放原队头结点
    return retvalue; //返回数据存放地址
}

```

队空条件 $p == \text{NULL}$ 。

4-12 若将一个双端队列顺序表示在一维数组 $V[m]$ 中, 两个端点设为 end1 和 end2 , 并组织成一个循环队列。试写出双端队列所用指针 end1 和 end2 的初始化条件及队空与队满条件, 并编写基于此结构的相应的插入(enqueue)新元素和删除(dlqueue)算法。

【解答】

初始化条件 $\text{end1} = \text{end2} = 0$;

队空条件 $\text{end1} = \text{end2}$;

队满条件 $(\text{end1} + 1) \% m = \text{end2}$; //设 end1 端顺时针进栈, end2 端逆时针进栈

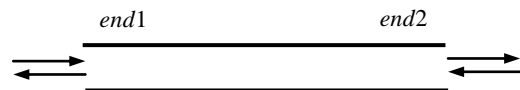
循环队列类定义

```

#include <assert.h>

template <class Type> class DoubleQueue { //循环队列的类定义
public:
    DoubleQueue ( int=10 );
    ~DoubleQueue ( ) { delete [ ] V; }
    void EnQueue ( Type & item, const int end );
    Type DeQueue ( const int end );
    Type GetFront ( const int end );
    void MakeEmpty ( ) { end1 = end2 = 0; } //置空队列
    int IsEmpty ( ) const { return end1 == end2; } //判两队列空否
    int IsFull ( ) const { return (end1+1) \% m == end2; } //判两队列满否
private:
    int end1, end2; //队列两端的指针
}

```



```

    Type *V;                                //存放队列元素的数组
    int m;                                    //队列最大可容纳元素个数
}

```

构造函数

```

template <class Type>
DoubleQueue<Type>::DoubleQueue ( int sz ) : end1 (0), end2 (0), m (sz) {
    //建立一个最大具有 m 个元素的空队列。
    V = new Type[m];                      //创建队列空间
    assert ( V != 0 );                    //断言：动态存储分配成功与否
}

```

插入函数

```

template<class Type>
void DoubleQueue<Type>::EnQueue ( Type &item, const int end ) {
    assert ( !IsFull ( ) );
    if ( end == 1 ) {
        end1 = ( end1 + 1 ) % m;            //end1 端指针先进 1，再按指针进栈
        V[end1] = item;                   //end1 指向实际队头位置
    }
    else {
        V[end2] = item;                   //end2 端先进队列，指针再进 1
        end2 = ( end2 + 1 + m ) % m;       //end2 指向实际队头的下一位置
    }
}

```

删除函数

```

template<class Type>
Type DoubleQueue<Type>::DeQueue ( const int end ) {
    assert ( !IsEmpty ( ) );
    Type& temp;
    if ( end == 1 ) {
        temp = V[end1];                   //先保存原队头元素的值, end1 端指针退 1
        end1 = ( end1 + m - 1 ) % m;
    }
    else {
        end2 = ( end2 + 1 ) % m;
        temp = V[end2];                   //end2 端指针先退 1。再保存原队头元素的值
    }
    return temp;
}

```

读取队头元素的值

```

template<class Type>
Type DoubleQueue<Type>::GetFront ( const int end ) {
    assert ( !IsEmpty ( ) );
    Type& temp;
    if ( end == 1 ) return V[end1];       //返回队头元素的值
}

```

```

        else return V[(end2+1) % m];
    }

```

4-13 设用链表表示一个双端队列，要求可在表的两端插入，但限制只能在表的一端删除。试编写基于此结构的队列的插入(enqueue)和删除(dequeue)算法，并给出队列空和队列满的条件。

【解答】

链式双端队列的类定义

```

template <class Type> class DoubleQueue;           //链式双端队列类的前视定义

template <class Type> class DoubleQueueNode {       //链式双端队列结点类定义
friend class DoubleQueue<Type>;
private:
    Type data;                                     //数据域
    DoubleQueueNode<Type> *link;                  //链域
public:
    DoubleQueueNode (Type d = 0, DoubleQueueNode *l = NULL) : data (d), link (l) { } //构造函数
};

template <class Type> class DoubleQueue {           //链式双端队列类定义
public:
    DoubleQueue ( );                               //构造函数
    ~DoubleQueue ( );                              //析构函数
    void EnDoubleQueue1 ( const Type& item );       //从队列 end1 端插入
    void EnDoubleQueue2 ( const Type& item );       //从队列 end2 端插入
    Type DeDoubleQueue ( );                         //删除并返回队头 end1 元素
    Type GetFront ( );                             //查看队头 end1 元素的值
    void MakeEmpty ( );                            //置空队列
    int IsEmpty ( ) const { return end1 == end1->link; } //判队列空否
private:
    QueueNode<Type> *end1, *end2;                  //end1 在链头，可插可删; end2 在链尾，可插不可删
};

```

队列的构造函数

```

template<class Type> doubleQueue<Type> :: doubleQueue ( ) { //构造函数
    end1 = end2 = new DoubleQueueNode<Type>( );           //创建循环链表的表头结点
    assert ( !end1 || !end2 );
    end1->link = end1;
}

```

队列的析构函数

```

template <class Type> Queue<Type> :: ~Queue ( ) {          //队列的析构函数
    QueueNode<Type> *p;                                     //逐个删除队列中的结点，包括表头结点
    while ( end1 != NULL ) { p = end1; end1 = end1->link; delete p; }
}

```

队列的插入函数

```

template<class Type>                                     //从队列 end1 端插入

```

```

void DoubleQueue<Type> :: EnDoubleQueue1 ( const Type& item ) {
    if ( end1 == end1->link )                //队列空, 新结点成为第一个结点
        end2 = end1->link = new DoubleQueueNode<Type> ( item, end1 );
    else                                     //队列不空, 新结点链入 end1 之后
        end1->link = new DoubleQueueNode<Type> ( item, end1->link );
}

template <class Type>                      //从队列 end2 端插入
void DoubleQueue<Type> :: EnDoubleQueue2 ( const Type& item ) {
    end2 = end2->link = new DoubleQueueNode<Type> ( item, end1 );
}

```

队列的删除函数

```

template <class Type>
Type DoubleQueue<Type> :: DeDoubleQueue ( ) {
    if ( IsEmpty ( ) ) return { cout << "队列空, 不能删除! " << endl; return 0; }
    DoubleQueueNode<Type> *p = end1->link;    //被删除结点
    end1->link = p->link;                     //重新链接
    Type retvalue = p->data; delete p;        //删除 end1 后的结点 p
    if ( IsEmpty ( ) ) end2 = end1;
    return retvalue;
}

```

读取队列 end1 端元素的内容

```

template <class Type> Type DoubleQueue<Type> :: GetFront ( ) {
    assert ( !IsEmpty ( ) );
    return end1->link->data;
}

```

置空队列

```

template <class Type> void Queue<Type>:: MakeEmpty ( ) {
    QueueNode<Type> *p;                      //逐个删除队列中的结点, 包括表头结点
    while ( end1 != end1->link ) { p = end1; end1 = end1->link; delete p; }
}

```

4-14 试建立一个继承结构, 以栈、队列和优先级队列为派生类, 建立它们的抽象基类——Bag 类。写出各个类的声明。统一命名各派生类的插入操作为 Add, 删除操作为 Remove, 存取操作为 Get 和 Put, 初始化操作为 MakeEmpty, 判空操作为 Empty, 判满操作为 Full, 计数操作为 Length。

【解答】

Bag 类的定义

```

template<class Type> class Bag {
public:
    Bag ( int sz = DefaultSize );           //构造函数
    virtual ~Bag ( );                         //析构函数
    virtual void Add ( const Type& item );     //插入函数
    virtual Type *Remove ( );                 //删除函数
    virtual int IsEmpty ( ) { return top == -1; } //判空函数
}

```

```

        virtual int IsFull () { return top == maxSize - 1; }           //判满函数
    private:
        virtual void Empty () { cout << "Data Structure is empty." << endl; }
        virtual void Full () { cerr << "DataStructure is full." << endl; }
        Type *elements;                                              //存储数组
        int maxSize;                                                  //数组的大小
        int top;                                                      //数组当前元素个数
};

```

Bag 类的构造函数

```

template<class Type> Bag<Type> :: Bag ( int MaxBagSize ) : MaxSize ( MaxBagSize ) {
    elements = new Type [ MaxSize ];
    top = -1;
}

```

Bag 类的析构函数

```

template<class Type> Bag<Type> :: ~Bag () {
    delete [ ] elements;
}

```

Bag 类的插入函数

```

template<class Type> void Bag<Type> :: Add ( const Type & item ) {
    if ( IsFull () ) Full ();
    else elements [ ++top ] = item;
}

```

Bag 类的删除函数

```

template <class Type> Type *Bag<Type> :: Remove () {
    if ( IsEmpty () ) { Empty (); return NULL; }
    Type & x = elements [0];                                //保存被删除元素的值
    for ( int i = 0; i < top; i++ )                          //后面元素填补上来
        elements [i] = elements [ i+1];
    top--;
    return &x;
}

```

栈的类定义（继承 Bag 类）

```

template<class Type> class Stack : public Bag {
public:
    Stack ( int sz = DefaultSize );                          //构造函数
    ~Stack ();                                                //析构函数
    Type *Remove ();                                         //删除函数
};

```

栈的构造函数

```

template<class Type> Stack<Type> :: Stack ( int sz ) : Bag ( sz ) { }
//栈的构造函数 Stack 将调用 Bag 的构造函数

```

栈的析构函数

```

template<class Type> Stack<Type> :: ~Stack () { }
//栈的析构函数将自动调用 Bag 的析构函数，以确保数组 elements 的释放

```

栈的删除函数

```
template<class Type> Type * Stack<Type> :: Remove ( ) {
    if ( IsEmpty ( ) ) { Empty ( ); return NULL; }
    Type& x = elements [ top-- ];
    return &x;
}
```

队列的类定义（继承 Bag 类）

```
template<class Type> class Queue : public Bag {
public:
    Queue ( int sz = DefaultSize );           //构造函数
    ~Queue ( );                               //析构函数
};
```

队列的构造函数

```
template<class Type> Queue<Type> :: Queue ( int sz ) : Bag ( sz ) { }
//队列的构造函数 Queue 将调用 Bag 的构造函数
```

优先级队列的类定义（继承 Bag 类）

```
template <class Type> class PQueue : public Bag {
public:
    PQueue ( int sz = DefaultSize );           //构造函数
    ~PQueue ( ) { }                           //析构函数
    Type *PQRemove ( );                       //删除函数
}
```

优先级队列的构造函数

```
template <class Type> PQueue<Type> :: PQueue ( int sz ) : Bag ( sz ) { }
//建立一个最大具有 sz 个元素的空优先级队列。top = -1。
```

优先级队列的删除函数

```
template <class Type> Type *PQueue<Type> :: Remove ( ) {
//若优先级队列不空则函数返回该队列具最大优先权(值最小)元素的值，同时将该元素删除。
    if ( IsEmpty ( ) ) { Empty ( ); return NULL; }
    Type& min = elements[0];                   //假设 elements[0]是最小值，继续找最小值
    int minindex = 0;
    for ( int i = 1; i <= top; i++ )
        if ( elements[i] < min ) { min = elements[i]; minindex = i; }
    elements[minindex] = elements[top];        //用最后一个元素填补要取走的最小值元素
    top--;
    return& min;                               //返回最小元素的值
}
```

4-15 试利用优先级队列实现栈和队列。

【解答】

```
template <class Type> class PQueue;           //前视的类定义

template <class Type> class PQueueNode {      //优先级队列结点类的定义
friend class PQueue<Type>;                  //PQueue 类作为友元类定义
```

public:

```
PQueueNode ( Type& value, int newpriority, PQueue<Type> * next )
    : data ( value ), priority ( newpriority ), link ( next ) { }           //构造函数
virtual Type GetData ( ) { return data; }                               //取得结点数据
virtual int GetPriority ( ) { return priority; }                         //取得结点优先级
virtual PQueueNode<Type> * GetLink ( ) { return link; }                 //取得下一结点地址
virtual void SetData ( Type& value ) { data = value; }                //修改结点数据
virtual void SetPriority ( int newpriority ) { priority = newpriority; } //修改结点优先级
virtual void SetLink ( PQueueNode<Type> * next ) { link = next; }     //修改指向下一结点的指针
```

private:

```
Type data;                      //数据
int priority;                   //优先级
ListNode<Type> *link;           //链指针
```

};

template <class Type> class PQueue { //优先级队列的类定义

public:

```
PQueue ( ) : front ( NULL ), rear ( NULL ) { }           //构造函数
virtual ~PQueue ( ) { MakeEmpty ( ); }                 //析构函数
virtual void Insert ( Type& value, int newpriority );   //插入新元素 value 到队尾
virtual Type Remove ( );                               //删除队头元素并返回
virtual Type Get ( );                                  //读取队头元素的值
virtual void MakeEmpty ( );                             //置空队列
virtual int IsEmpty ( ) { return front == NULL; }       //判队列空否
```

private:

```
PQueueNode<Type> *front, *rear;           //队头指针, 队尾指针
```

};

template<class Type>

```
void PQueue<Type> :: MakeEmpty ( ) {                     //将优先级队列置空
    PQueueNode<Type> *q;
    while ( front != NULL )                               //链不空时, 删去链中所有结点
        { q = front; front = front->link; delete q; }     //循链逐个删除
    rear = NULL;                                          //队尾指针置空
}
```

template<class Type>

```
void PQueue<Type> :: Insert ( Type& value, int newpriority ) { //插入函数
    PQueueNode<Type> *q = new PQueueNode ( value, newpriority, NULL );
    if ( IsEmpty ( ) ) front = rear = q;                 //队列空时新结点为第一个结点
    else {
        PQueueNode<Type> *p = front, *pr = NULL;        //寻找 q 的插入位置
        while ( p != NULL && p->priority >= newpriority ) //队列中按优先级从大到小链接
            { pr = p; p = p->link; }
    }
```

```

        if ( pr == NULL ) { q->link = front; front = q; } //插入在队头位置
        else { q->link = p; pr->link = q; //插入在队列中部或尾部
                if ( pr == rear ) rear = q;
            }
    }
}

```

```

template<class Type> Type PQueue<Type> :: Remove ( ) { //删除队头元素并返回
    if ( IsEmpty ( ) ) return NULL;
    PQueueNode<Type> *q = front; front = front->link; //将队头结点从链中摘下
    Type &retvalue = q->data; delete q;
    if ( front == NULL ) rear = NULL;
    return& retvalue;
}

```

```

template<class Type> Type PQueue<Type> :: Get ( ) { //读取队头元素的值
    if ( IsEmpty ( ) ) return NULL;
    else return front->data;
}

```

(1) 栈的定义与实现

```

template <class Type> class Stack : public PQueue { //栈类定义
public:
    Stack ( ) : front ( NULL ), rear ( NULL ) { } //构造函数
    void Insert ( Type & value ); //插入新元素 value 到队尾
}

template<class Type>
void Stack<Type> :: Insert ( Type& value ) { //插入函数
    PQueueNode<Type> * q = new PQueueNode ( value, 0, NULL );
    if ( IsEmpty ( ) ) front = rear = q; //栈空时新结点为第一个结点
    else { q->link = front; front = q; } //插入在前端
}

```

(2) 队列的定义与实现

```

template <class Type> class Queue : public PQueue { //队列类定义
public:
    Queue ( ) : front ( NULL ), rear ( NULL ) { } //构造函数
    void Insert ( Type& value ); //插入新元素 value 到队尾
}

template<class Type>
void Queue<Type> :: Insert ( Type & value ) { //插入函数
    PQueueNode<Type>* q = new PQueueNode ( value, 0, NULL );
    if ( IsEmpty ( ) ) front = rear = q; //队列空时新结点为第一个结点
    else rear = rear->link = q; //插入在队尾位置
}

```

4-15 所谓回文，是指从前向后顺读和从后向前倒读都一样的不含空白字符的串。例如 did, madamimadam,

pop 即是回文。试编写一个算法，以判断一个串是否是回文。

【解答 1】

将字符串中全部字符进栈，然后将栈中的字符逐个与原字符串中的字符进行比较。算法如下：

```
int palindrome ( char A[ ], int n ) {
    stack<char> st (n+1);
    int yes = 1, i = 0;
    while ( A[i] != "\0" ) { st.Push ( A[i] ); i++; }           //扫描字符串，所有字符进栈
    i = 0;
    while ( A[i] != "\0" )                                       //比较字符串
        if ( A[i] == st.GetTop ( ) ) { st.Pop ( ); i++; }
        else { yes = 0; break; }
    return yes;
}
```

【解答 2】

采用递归算法，判断从 s 到 e 中的字符串是否回文，通过函数返回是或不是。

```
int palindrome ( char A[ ], int s, int e ) {
    if ( A[s] != A[e] ) return 0;
    else if ( s > e ) return 1;
    else palindrome ( A, s+1, e-1 );
}
```

4-16 设有一个双端队列，元素进入该队列的顺序是 1, 2, 3, 4。试分别求出满足下列条件的输出序列。

- (1) 能由输入受限的双端队列得到，但不能由输出受限的双端队列得到的输出序列；
- (2) 能由输出受限的双端队列得到，但不能由输入受限的双端队列得到的输出序列；
- (3) 既不能由输入受限的双端队列得到，又不能由输出受限的双端队列得到的输出序列。

【解答】

允许在一端进行插入和删除，但在另一端只允许插入的双端队列叫做输出受限的双端队列，允许在一端进行插入和删除，但在另一端只允许删除的双端队列叫做输入受限的双端队列。

输出受限双端队列不能得到的输出序列有：

4 1 3 2 4 2 3 1

输入受限双端队列不能得到的输出序列有：

4 2 1 3 4 2 3 1

所以有：

- (1) 4 1 3 2
- (2) 4 2 1 3
- (3) 4 2 3 1

