第4章 栈和队列

**一、复习要点**

本章主要讨论3种线性结构：栈、队列与优先级队列。这3种结构都是顺序存取的表，而且都是限制存取点的表。栈限定只能在表的一端（栈顶）插入与删除，其特点是先进后出。队列和优先级队列限定只能在表的一端（队尾）插入在另一端（队头）删除，不过优先级队列在插入和删除时需要根据数据对象的优先级做适当的调整，令优先级最高的对象调整到队头，其特点是优先级高的先出。而队列不调整，其特点是先进先出。这几种结构在开发各种软件时非常有用。

本章复习的要点：

**1、**基本知识点

要求理解栈的定义和特点，栈的抽象数据类型和在递归和表达式计算中的使用，在栈式铁路调车线上当进栈序列为1, 2, 3, …, n时，可能的出栈序列计数，栈的顺序存储表示和链接存储表示，特别要注意，链式栈的栈顶应在链头，插入与删除都在链头进行。另外，需要理解队列的定义和特点，队列的抽象数据类型和在分层处理中的使用，队列的顺序存储表示（循环队列）和链接存储表示，需要注意的是，链式队列的队头应在链头，队尾应在链尾。还需要理解优先级队列的定义和特点。优先级队列的最佳存储表示是堆（heap），本章介绍的表示看懂即可。

**2、**算法设计

⮚ 栈的5种操作（进栈、退栈、取栈顶元素、判栈空、置空栈）的在顺序存储表示下的实现，以及在链接存储表示下的实现。

⮚ 使用栈的后缀表达式计算算法

⮚ 双栈共用一个数组的进栈、退栈、置空栈、判栈空算法及栈满、栈空条件

⮚ 使用两个栈模拟一个队列时的进队列和出队列算法

⮚ 循环队列的进队列、出队列、取队头元素、判队列空、置空队列操作的实现

⮚ 使用tag区分队列空和队列满的循环队列的进队列和出队列操作的实现

⮚ 链式队列的进队列、出队列、取队头元素、判队列空、置空队列操作的实现

⮚ 使用队尾指针rear和队列长度length的链式队列的进队列、出队列、取队头元素、判队列空、置空队列操作的实现

⮚ 队列在分层处理中的使用事例（杨辉三角形按层次打印）

⮚ 双端队列的顺序存储表示及其进队列、出队列算法及队空、队满条件

**二、难点和重点**

1、栈：栈的特性、栈的基本运算

⮚ 栈的数组实现、栈的链表实现

⮚ 栈满及栈空条件、抽象数据类型中的先决条件与后置条件

2、栈的应用：用后缀表示计算表达式，中缀表示改后缀表示

3、队列：队列的特性、队列的基本运算

⮚ 队列的数组实现：循环队列中队头与队尾指针的表示，队满及队空条件

⮚ 队列的链表实现：链式队列中的队头与队尾指针的表示、

4、双向队列：双向队列的插入与删除算法

5、优先级队列：优先级队列的插入与删除算法

**三、教材中习题的解析**

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

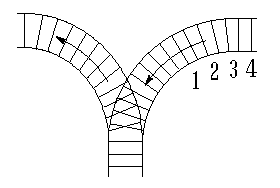
**for** ( **int** i = 0**;** i <= top**;** i++ ) temp[i] = elements[i]**;** //传送原数组的数据

**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 种。

(2) 不能得到435612和154623这样的出栈序列。因为若在4, 3, 5, 6之后再将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。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 |  |  |  |  |  | 5 |  |  |  | 6 |  |  |  |  |  |  |
| 2 |  | 2 |  |  |  | 4 |  | 4 |  | 4 |  | 4 |  |  |  |  |
| 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  |  |

3 32 32 325 325 3256 32564 325641

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | 5 |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 3 |  | 4 |  | 4 |  |  |  |  |  |  |  |  |
| 1 |  |  |  | 2 |  | 2 |  | 2 |  | 2 |  |  |  | 6 |  |  |

1 1 13 135 1354 13542 13542 135426

4-3 试证明：若借助栈可由输入序列1, 2, 3, …, n得到一个输出序列p1, p2, p3, …, pn (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pj < pk < pi。(提示：用反证法)

【解答】

充分性：由j < k，pj < pk，则pj必须在pk入栈之前就出栈；而i < j，pj < pi，则意味着pi必须先于pj进栈且pj必须先于pi出栈；此外，i < k，则表明pk必须在pi之后出栈，这与pj < pk < pi相矛盾（因为这意味着pj必须在pk之前和pi之后离开，但pi又出现在pk之后）。

下面详细解释一下。借助栈由输入序列1, 2, 3, …, n，可得到输出序列p1, p2, p3, …, pn ，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pi位置，具有中间值的排在其后pj位置，具有最大值的排在pk位置，有pi < pj < pk, 不可能出现pj < pk < pi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pi位置，具有最大值的排在pj位置，具有中间值的排在最后pk位置，有pi < pk < pj , 不可能出现pj < pk < pi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pi位置，具有最小值的排在其后pj位置，有pj < pi < pk, 不可能出现pj < pk < pi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pi 位置，具有最大值的排在其后pj位置，具有最小值的排在pk位置，有pk < pi < pj, 也不可能出现pj < pk < pi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pi 位置，具有中间值的排在其后pj位置，具有最小值的排在pk位置，有pk < pj < pi, 也不可能出现pj < pk < pi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top[0]等于-1时该栈为空，当第1号栈的栈顶指针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top[1]减1得到新的栈顶位置。当top[0]+1 *==* top[1]时或top[0] *==* 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】

0 m-1

bot[0] top[0] top[1] bot[1]

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 ( **int** 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↑2，试利用栈将它改为后缀表示ax \* bx2↑/ -。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 | **;** \* | a x |
| 4 | - | 操作符 | ☞ isp ( ' \* ' ) > icp ( ' - ' ), 退栈输出 | **;** | a x \* |
|  |  |  | ☞ isp ( ' **;** ' ) < icp ( ' - ' ), 进栈, 读下一符号 | **;** - | a x \* |
| 5 | b | 操作数 | ☞ 直接输出, 读下一符号 | **;** - | a x \* b |
| 6 | / | 操作符 | ☞ isp ( ' - ' ) < icp ( '/' ), 进栈, 读下一符号 | **;** -/ | a x \* b |
| 7 | x | 操作数 | ☞ 直接输出, 读下一符号 | **;** -/ | a x \* b x |
| 8 | ↑ | 操作符 | ☞ isp ( ' / ' ) < icp ( '↑' ), 进栈, 读下一符号 | **;** -/↑ | a x \* b x |
| 9 | 2 | 操作数 | ☞ 直接输出, 读下一符号 | **;** -/↑ | a x \* b x 2 |
| 10 | **;** | 操作符 | ☞ isp ( '↑' ) > icp ( ' **;** ' ), 退栈输出 | **;** -/ | a x \* b x 2↑ |
|  |  |  | ☞ isp ( ' / ' ) > icp ( ' **;** ' ), 退栈输出 | **;** - | a x \* b x 2↑/ |
|  |  |  | ☞ isp ( ' - ' ) > icp ( ' **;** ' ), 退栈输出 | **;** | a x \* 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↑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a2，再退出一个运算符作为第1操作数a1，从OPTR栈退出一个运算符θ形成运算指令 (a1)θ(a2)，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↑f / g时两个栈的变化。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 步序 | 扫描项 | 项类型 | 动作 | OPND栈 | OPTR栈 |
| 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 → s1, 结果  进OPND栈 | a b s1 | **;** + \* ( |
| 10 | 同上 | 同上 | ☞ icp (‘ ) ’ ) == isp (‘ ( ’ ), 退OPTR栈‘(’, 对消括号,  取下一符号 | a b s1 | **;** + \* |
| 11 | - | 操作符 | ☞ icp (‘ - ’ ) < isp (‘ \* ’ ), 退OPND栈 ‘s1’, 退OPND  栈 ‘b’, 退OPTR栈 ‘\*’, 计算 b \* s1 → s2, 结果  进OPND栈 | a s2 | **;** + |
| 12 | 同上 | 同上 | ☞ icp (‘ - ’ ) < isp (‘ + ’ ), 退OPND栈 ‘s2’, 退OPND  栈 ‘a’, 退OPTR栈 ‘+’, 计算 a \* s2 → s3, 结果  进OPND栈 | s3 | **;** |
| 13 | 同上 | 同上 | ☞ icp (‘ - ’ ) > isp (‘ **;** ’ ), 进OPTR栈, 取下一符号 | s3 | **;** - |
| 14 | e | 操作数 | ☞ e 进OPND栈, 取下一符号 | s3 e | **;** - |
| 15 | ↑ | 操作符 | ☞ icp (‘↑’ ) > isp (‘ -’ ), 进OPTR栈, 取下一符号 | s3 e | **;** -↑ |
| 16 | f | 操作数 | ☞ f 进OPND栈, 取下一符号 | s3 e f | **;** -↑ |
| 17 | / | 操作符 | ☞ icp (‘ / ’ ) < isp (‘↑’ ), 退OPND栈 ‘f’, 退OPND  栈 ‘e’, 退OPTR栈 ‘↑’, 计算 e↑f → s4, 结果  进OPND栈 | s3 s4 | **;** - |
| 18 | 同上 | 同上 | ☞ icp (‘ / ’ ) > isp (‘ -’ ), 进OPTR栈, 取下一符号 | s3 s4 | **;** - / |
| 19 | g | 操作数 | ☞ g 进OPND栈, 取下一符号 | s3 s4 g | **;** - / |
| 20 | **;** | 操作符 | ☞ icp (‘ **;** ’ ) < isp (‘ / ’ ), 退OPND栈 ‘g’, 退OPND  栈 ‘s4’, 退OPTR栈 ‘/’, 计算 s4 / g → s5, 结果  进OPND栈 | s3 s5 | **;** - |
| 21 | 同上 | 同上 | ☞ icp (‘ **;** ’ ) < isp (‘ - ’ ), 退OPND栈 ‘s5’, 退OPND  栈 ‘s3’, 退OPTR栈 ‘ - ’, 计算 s3 – s5 → s6, 结  果进OPND栈 | s6 | **;** |
| 22 | 同上 | 同上 | ☞ icp (‘ **;** ’ ) == isp (‘ **;** ’ ), 退OPND栈 ‘s6’, 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分别指示循环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

**}**

**template <class Type>** Queue**<Type>::** Queue ( **int** sz ) **:** rear (maxSize-1)**,** length (0)**,** maxSize (sz) **{**

//构造函数：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

**}**

**template<class Type> void** Queue<**Type**> **::** EnQueue ( **Type &**item) **{**

**//**插入函数

**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

**}**

**template<class Type> Type** Queue<**Type**> **::** DeQueue ( ) **{**

**//**删除函数

**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

**}**

**template<class Type> Type** Queue<**Type**> **::** GetFront ( ) **{**

**//**读取队头元素值函数

**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(dlqueue)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

**}**

**template <class Type>** Queue**<Type>::** Queue ( **int** sz ) **:** rear (0)**,** front (0)**,** tag(0)**,** m (sz) **{**

//构造函数：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

**}**

**template<class Type> void** Queue<**Type**> **::** EnQueue ( **Type &**item) **{**

**//**插入函数

**assert** ( ! IsFull ( ) )**;** //判队列是否不满，满则出错处理

rear = ( rear + 1 ) % m**;** //队尾进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

**}**

**template<class Type> Type** Queue<**Type**> **::** DeQueue ( ) **{**

**//**删除函数

**assert** ( ! IsEmpty ( ) )**;** //判断队列是否不空，空则出错处理

front = ( front + 1 ) % m**;** //队头进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

**}**

**template<class Type> Type** Queue<**Type**> **::** GetFront ( ) **{**

**//**读取队头元素函数

**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）

**};**

**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

**}**

**template <class Type> void** Queue**<Type> ::** EnQueue ( **const Type** **&** item ) **{**

**//**队列的插入函数

**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 ( ) **{**

**//**队列的插入函数

**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 == 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 end1 = end2 = 0**;**

end1

end2

队空条件 end1 = end2**;**

队满条件 ( end1 + 1 ) % m = 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

**}**

**template <class Type>** DoubleQueue**<Type> ::** DoubleQueue ( **int** sz ) **:** end1 (0)**,** end2 (0)**,** m (sz) **{**

//构造函数：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

**}**

**template<class Type> void** DoubleQueue<**Type**> **::** EnQueue ( **Type &**item**, const int** end ) **{**

**//**插入函数

**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

**}**

**}**

**template<class Type> Type** DoubleQueue<**Type**> **::** DeQueue ( **const int** end ) **{**

**//**删除函数

**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**;**

**}**

**template<class Type> Type** DoubleQueue<**Type**> **::** GetFront ( **const int** end ) **{**

**//**读取队头元素的值

**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

**};**

**template<class Type>** doubleQueue<**Type**> **::** doubleQueue ( ) **{**

//队列的构造函数

end1 = end2 = **new** DoubleQueueNode**<Type>**( )**;** //创建循环链表的表头结点

**assert (** !end1 || !end2 **);**

end1->link = end1**;**

**}**

**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**; }**

**}**

**template<class Type>** **void** DoubleQueue**<Type> ::** EnDoubleQueue1 ( **const Type&** item ) **{**

//队列的插入：从队列end1端插入

**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 )**;**

**}**

t**emplate <class Type> void** DoubleQueue<**Type**> **::** EnDoubleQueue2 ( **const Type&** item ) **{**

//队列的插入：从队列end2端插入

end2 = end2->link = **new** DoubleQueueNode<**Type**> ( item**,** end1 )**;**

**}**

**template <class Type> Type** DoubleQueue<**Type**> **::** DeDoubleQueue ( ) **{**

**//**队列的删除函数

**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**;**

**}**

**template <class Type> Type** DoubleQueue<**Type**> **::** GetFront ( ) **{**

**//**读取队列end1端元素的内容

**assert** ( !IsEmpty ( ) )**;**

**return** end1->link->data**;**

**}**

**template <class Type> void** Queue**<Type>::** MakeEmpty ( ) **{**

//置空队列

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

**protecred:**

**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

**};**

**template<class Type>** Bag<**Type**> **::** Bag ( **int** MaxBagSize ) **:** MaxSize ( MaxBagSize ) **{**

**//** Bag类的构造函数

elements = **new** **Type** [ MaxSize ]**;**

top = -1**;**

**}**

**template<class Type>** Bag<**Type**> **::** ~Bag ( ) **{**

**//** Bag类的析构函数

**delete** [ ] elements**;**

**}**

**template<class Type> void** Bag<**Type**> **::** Add ( **const Type &** item ) **{**

**//** Bag类的插入函数

**if** ( IsFull ( ) ) Full ( )**;**

**else** elements [ ++top ] = item**;**

**}**

**template <class Type> Type** \*Bag<**Type**> **::** Remove ( ) **{**

**//** Bag类的删除函数

**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

**template<class Type> class** Stack **: public** Bag **{**

**//**栈的类定义（继承Bag类）

**public:**

Stack ( **int** sz = DefaultSize )**;** //构造函数

~Stack ( )**;** //析构函数

**Type** \*Remove ( )**;** //删除函数

**};**

**template<class Type>** Stack**<Type> ::** Stack( **int** sz) **:** Bag ( sz ) **{ }**

//栈的构造函数：将调用Bag的构造函数

**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

**template<class Type> Type** \* Stack**<Type> ::** Remove ( ) **{**

**//**栈的删除函数

**if** ( IsEmpty ( ) ) **{** Empty ( )**; return** NULL**; }**

**Type** x = elements [ top-- ]**;**

**return &**x**;**

**}**

队列的类定义

**template<class Type> class** Queue **: public** Bag **{**

**//**队列的类定义（继承Bag类）

**public:**

Queue ( **int** sz = DefaultSize )**;** //构造函数

~Queue ( )**;** //析构函数

**};**

**template<class Type>** Queue**<Type> ::** Queue( **int** sz) **:** Bag ( sz ) **{ }**

//队列的构造函数：将调用Bag的构造函数

优先级队列的类定义

**template <class Type> class** PQueue **: public** Bag **{**

**//**优先级队列的类定义（继承Bag类）

**public:**

PQueue ( **int** sz = DefaultSize )**;** //构造函数

~PQueue ( ) **{ }** //析构函数

**Type** \*PQRemove ( )**;** //删除函数

**}**

**template <class Type>** PQueue**<Type> ::** PQueue ( **int** sz ) **:** Bag ( sz ) **{ }**

//优先级队列的构造函数：建立一个最大具有sz个元素的空优先级队列。top = -1。

**template <class Type> Type** \*PQueue**<Type> ::** Remove ( ) **{**

//优先级队列的删除函数：若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**; }**

**private:** //修改指向下一结点的指针

**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

**}**

**四、其他练习题**

4-16 单选题

(1) 栈的插入和删除操作在\_\_\_\_\_\_\_\_进行。

A 栈顶 B 栈底 C 任意位置 D 指定位置

(2) 当利用大小为n的数组顺序存储一个栈时，假定用top*==*n表示栈空，则向这个栈插入一个元素时，首先应执行\_\_\_\_\_\_\_\_语句修改top指针。

A top++**;** B top*--***;** C top = 0**;** D top**;**

(3) 若让元素1,2,3依次进栈，则出栈次序不可能出现\_\_\_\_\_\_\_\_种情况。

A 3,2,1 B 2,1,3 C 3,1,2 D 1,3,2

(4) 在一个顺序存储的循环队列中，队头指针指向队头元素的\_\_\_\_\_\_\_\_位置。

A 前一个 B 后一个 C 当前 D 后面

(5) 当利用大小为n的数组顺序存储一个队列时，该队列的最大长度为\_\_\_\_\_\_\_\_。

A n-2 B n-1 C n D n+1

(6) 从一个顺序存储的循环队列中删除一个元素时，首先需要\_\_\_\_\_\_\_\_。

A 队头指针加一 B 队头指针减一

C 取出队头指针所指的元素 D 取出队尾指针所指的元素

(7) 假定一个顺序存储的循环队列的队头和队尾指针分别为f和r，则判断队空的条件为\_\_\_\_\_\_\_\_。

A f+1 *==* r B r+1 *==* f C f *==* 0 D f *==* r

(8) 假定一个链式队列的队头和队尾指针分别为front和rear，则判断队空的条件为\_\_\_\_\_\_\_\_。

A front *==* rear B front != NULL C rear != NULL D front *==* NULL

【解答】

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

(7) D (8) D

4-17 填空题

(1) 队列的插入操作在\_\_\_\_\_\_\_\_进行，删除操作在\_\_\_\_\_\_\_\_进行。

(2) 栈又称为\_\_\_\_\_\_\_\_的表，队列又称为\_\_\_\_\_\_\_\_的表。

(3) 向一个顺序栈插入一个元素时，首先使\_\_\_\_\_\_\_\_后移一个位置，然后把待插入元素\_\_\_\_\_\_\_\_到这个位置上。

(4) 从一个栈删除元素时，需要前移一位\_\_\_\_\_\_\_\_。

(5) 在一个循环队列Q中，判断队空的条件为\_\_\_\_\_\_\_\_，判断队满的条件为\_\_\_\_\_\_\_\_。

(6) 在一个顺序栈中，若栈顶指针等于\_\_\_\_\_\_\_\_，则为空栈；若栈顶指针等于\_\_\_\_\_\_\_\_，则为满栈。

(7) 在一个链式栈中，若栈顶指针等于NULL则为\_\_\_\_\_\_\_\_；在一个链式队列中，若队头指针与队尾指针的值相同，则表示该队列为\_\_\_\_\_\_\_\_或该队列\_\_\_\_\_\_\_\_。

(8) 向一个链式栈插入一个新结点时，首先把栈顶指针的值赋给\_\_\_\_\_\_\_\_，然后把新结点的存储位置赋给\_\_\_\_\_\_\_\_。

(9) 向一个循环队列中插入元素时，需要首先移动\_\_\_\_\_\_\_\_，然后再向所指位置\_\_\_\_\_\_\_\_新插入的元素。

(10) 当用长度为n的数组顺序存储一个栈时，若用top *==* n表示栈空，则表示栈满的条件为\_\_\_\_\_\_\_\_。

(11) 向一个栈顶指针为top的链式栈中插入一个新结点\*p时，应执行\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_操作。

(12) 从一个栈顶指针为top的非空链式栈中删除结点并不需要返回栈顶结点的值和回收结点时，应执行\_\_\_\_\_\_\_\_操作。

(13) 假定front和rear分别为一个链式队列的队头和队尾指针，则该链式队列中只有一个结点的条件为\_\_\_\_\_\_\_\_。

(14) 中缀表达式3\*(x+2)-5所对应的后缀表达式为\_\_\_\_\_\_\_\_。

(15) 后缀表达式“4 5 \* 3 2 + -”的值为\_\_\_\_\_\_\_\_。

【解答】

(1) 队尾，队头

(2) 后进先出，先进先出

(3) 栈顶指针，写入

(4) 栈顶指针

(5) Q.front *==* Q.rear，(Q.rear+1) % MaxSize+1 *==* Q.front

(6) –1，MaxSize-1

(7) 空栈，空，只含有一个结点

(8) 新结点的指针域，栈顶指针

(9) 队尾指针，写入

(10) top *==* 0

(11) p->link = top，top = p

(12) top = top->link

(13) front == rear **&&** front != NULL或者front == rear **&&** rear != NULL

(14) 3 x 2 + \* 5 -

(15) 15

4-18 设有一个顺序栈S，元素s1, s2, s3, s4, s5, s6依次进栈，如果6个元素的出栈顺序为s2, s3, s4, s6, s5, s1，则顺序栈的容量至少应为多少？

【解答】

3

4-19 设链式栈中结点的结构为（data, link），且top是指向栈顶的指针。若想在链式栈的栈顶插入一个由指针s所指的结点，则应执行下列哪一个操作？

(1) top->link = s**;** (2)s->link = top->link**;** top->link = s**;**

(3) s->link = top**;** top = s**;** (4) s->link = top**;** top = top->link**;**

【解答】

(3)

4-20 设链式栈中结点的结构为（data, link），且top是指向栈顶的指针。若想摘除链式栈的栈顶结点，并将被摘除结点的值保存到x中，则应执行下列哪一个操作？

(1) x = top->data**;** top = top->link**;** (2) top = top->link**;** x = top->data**;**

(3) x = top**;** top = top->link**;** (4) x = top->data**;**

【解答】

(1)

4-21 设循环队列的结构是

**const int** MaxSize = 100**;**

**typedef int** DataType**;**

**typedef struct {**

DataType data[MaxSize]**;**

**int** front, rear**;**

**}** Queue**;**

若有一个Queue类型的队列Q，试问判断队列满的条件应是下列哪一个语句？

(1) Q.front *==* Q.rear**;** (2) Q.front - Q.rear == MaxSize**;**

(3) Q.front + Q.rear == MaxSize**;** (4) Q.front == (Q.rear+1) % MaxSize**;**

【解答】

(4)

4-22 设循环队列的结构如4-21。若有一个Queue类型的队列Q，试问应用下列哪一个语句计算队列元素个数？

(1) (Q.rear - Q.front + MaxSize ) % MaxSize**;** (2) Q.rear - Q.front +1**;**

(3) Q.rear - Q.front -1**;** (4) Q.rear - Qfront**;**

【解答】

(1)

4-23 假设以数组Q[m]存放循环队列中的元素, 同时以rear和length分别指示循环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】

队空：length *==* 0 队满：length *==* Maxsize

4-24 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 )**;**

**}**

4-25 借助栈实现单链表上的逆置运算。

【解答】

由于进栈与出栈顺序正好相反，因此，借助栈可以实现单链表的逆置运算。方法是让单链表中的结点依次进栈，再依次出栈。

#**include** "stack.h"

#**include** "LinkList.h"

**template<class Type> void** invert ( LinkList<**Type**> L ) **{**

Stack<ListNode<**Type**>\* > S**;**

ListNode<**Type**> \*p = L.First ( )**,** \*q**;**

**while** ( p != NULL ) **{** S.Push(p)**;**  p = p.Next ( )**; } //**依次将链中结点进栈

p = L.Firster ( )**;** p->setLink ( NULL )**; //**单链表表头结点链指针置空

**while** ( !S.IsEmpty( ) ) **{ //**将栈中保存的结点依次出栈

q = S.GetTop ( )**;** S.Pop ( )**;**

q->SetLink ( p->getLink ( ) )**;** //链入逆置后的链中

p->setLink ( q )**;**

p = p->getLink ( )**;**

**}**

**}**

4-26 编写一个算法，将一个非负的十进制整数N转换为另一个基为B的B进制数。

【解答】

可以利用栈解决数制转换问题。例如：

4910 = 6·81 +1·80 = 618，9910 = 1·26 +1·25 +0·24 +0·23 +0·22 +1·21 +1·20 = 11000112。

其转换规则是



其中，bi 表示B进制数的第i位上的数字。这样，十进制数N可以用长度为 ⎣logBN⎦ +1位的B进制数表示为b ⎣logBN⎦ … b2 b1 b0。若令j = ⎣logBN⎦，则有

N = bj Bj + bj-1 B j-1 + … + b1 B1 + b0 =

= ( bj B j-1 + bj-1 B j-2 +… + b1 ) B + b0 = ( N / B )·B + N % B ( "/"表示整除运算)

因此，可以先通过N % B求出b0，然后令N = N / B，再对新的N做除B求模运算可求出b1，…，如此重复直到某个N等于零结束。这个计算过程是从低位到高位逐个进行的，但输出过程是从高位到低位逐个打印的，为此需要利用栈来实现。

#**include** "stack.h"

**int** BaseTrans ( **int** N**, int** B ) **{**

**int** i, result = 0**;** Stack<**int**> S**;**

**while** ( N != 0 ) **{** i = N % B**;** N = N / B**;** S.Push ( i )**; }**

**while** ( !S.IsEmpty ( ) ) **{** i = S.GetTop( )**;** S.Pop( )**;** result = result\*10 + i**; }**

**return** result**;**

**}**

4-27试编写一个算法，检查一个程序中的花括号、方括号和圆括号是否配对，若能够全部配对则返回1，否则返回0。

【解答】

在算法中，扫描程序中的每一个字符，当扫描到每个花、中、圆左括号时，令其进栈，当扫描到花、中、圆右括号时，则检查栈顶是否为相应的左括号，若是则作退栈处理，若不是则表明出现了语法错误，应返回0。当扫描到程序文件结尾后，若栈为空则表明没有发现括号配对错误，应返回1，否则表明栈中还有未配对的括号，应返回0。另外，对于一对单引号或双引号内的字符不进行括号配对检查。

根据分析，编写出算法如下：

**#include** <iostream.h>

**#include** “stack.h”

**int** BracketsCheck ( **char**\* fname) **{**

//对由fname所指字符串为文件名的文件进行括号配对检查

**ifstream** ifstr ( fname, **ios::in|ios::nocreate** )**;**

//用文件输入流对象ifstr打开以fname所指字符串为文件名的文件

**if** ( !ifstr ) **{**

**cerr** << "File" << "\'" << fname << "\'"<< "not found!" << **endl;**

**exit** (1)**;**

**}**

stack<**char**> S**;** **char** ch**;** //定义一个栈

**while** ( ifstr >> ch ) **{** //顺序扫描文件中的每一个字符

**if** ( ch == 39 ) **{** //单引号内的字符不参与配对比较

**while** ( ifstr >> ch )

**if** ( ch == 39 ) **break;**  //39为单引号的ASCII值

**if** ( !ifstr ) **return** 0**;**

**}**

**else if** ( ch == 34 ) **{** //双引号内的字符不参与配对比较

**while** ( ifstr >> ch )

**if** ( ch == 34 ) **break;** //34为双引号的ASCII值

**if** ( !ifstr ) **return** 0**;**

**}**

**switch** ( ch ) **{**

**case** '{' **: case** '[' **: case** '(' **:** S.Push(ch)**; break;** //出现以上三种左括号则进栈

**case** '}' **: if** ( S.GetTop( ) == '{' ) S.Pop( )**;**  //栈顶的左花括号出栈

**else return** 0**; break;**

**case** ']' **: if** ( S.GetTop ( ) == '[' ) S.Pop( )**;** //栈顶的左中括号出栈

**else return** 0**; break;**

**case** ')' **: if** ( S.GetTop( ) == '(' ) S.Pop( )**;** //栈顶的左圆括号出栈

**else return** 0;

**}**

**}**

**if** ( S.IsEmpty( ) ) **return** 1**;**

**else return** 0**;**

**}**

4-28 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：

end1 end2

输出受限的双端队列

end1 end2

输入受限的双端队列

4 1 3 2 4 2 3 1

输人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