第 3 章　栈 和 队 列

50 多年前，与同学相约去北京郊区上方山云水洞旅游。当时景区尚未开发，我们拿着从老乡那里买来的火炬就进了云水洞。为不致迷路，按照当地老乡的嘱咐，我们一路走，一路在洞壁上、地上做标记，最后依靠这些标记退出洞来。这些标记记忆了回退的路径，实际上就是一个栈，回退时最先看到的是最后做的标记。

再看日常生活中其他的例子。如银行取款，一进门就需要拿一个号，然后等待叫号。这就是队列，银行里办理业务至少有 5 个队列：个人业务、对公业务、VIP、理财金和外币业务。基本上都是先来先服务。

不要看栈和队列简单，在计算机应用系统中应用极多。Windows 操作系统中就用到了 9000 多个栈。而且栈与队列都是顺序存取结构，比向量更容易使用。就像结构化编程相对于普通编程，具有更加优良的程序设计风格。

3.1　栈

栈、队列和双端队列是特殊的线性表，它们的逻辑结构和线性表相同，只是其运算规则较线性表有更多的限制，故又称它们为运算受限的线性表，或限制了存取点的线性表。

3.1.1　栈的概念

1. 栈的定义

栈(Stack)是只允许在表的一端进行插入和删除的线性表。允许插入和删除的一端称为栈顶(top)，而不允许插入和删除的另一端称为栈底(bottom)。当栈中没有任何元素时则成为空栈。

2. 栈的后进先出特性

设给定栈 $S=(a_1,a_2,\cdots,a_n)$，则称最后加入栈中的元素 a_n 为栈顶。栈中按 $a_1,a_2,\cdots,a_n$ 的顺序进栈。而退栈的顺序反过来，a_n 先退出，然后 a_{n-1} 才能退出，最后退出 a_1。换句话说，后进者先出。因此，栈又称为**后进先出**(Last In First Out，LIFO)的线性表。参看图 3-1。

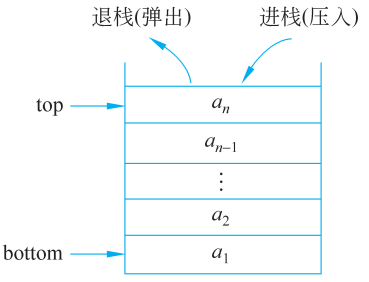


图 3-1　栈的示意图

3. 栈的主要操作

```
(1) void Stack ( Stack& S );
```

先决条件：无。

操作结果：为栈 S 分配存储空间，并对各数据成员赋初值。

```
(2) bool Push ( Stack& S, SElemType x );
```

先决条件：栈 S 已存在且未满。

操作结果：新元素 x 进栈 S 并成为新的栈顶。

```
(3) bool Pop ( Stack& S, SElemType & x );
```

先决条件：栈 S 已存在且栈非空。

操作结果：若栈 S 空，则函数返回 false，x 不可用；否则栈 S 的栈顶元素退栈，退出元素由 x 返回，函数返回 true。

```
(4) bool GetTop ( Stack& S, SElemType & x );
```

先决条件：栈 S 已存在且栈非空。

操作结果：若栈 S 空，则函数返回 false，x 不可用；否则由引用型参数 x 返回栈顶元素的值但不退栈，函数返回 true。

```
(5) bool StackEmpty ( Stack& S );
```

先决条件：栈 S 已存在。

操作结果：函数测试栈 S 空否。若栈空，则函数返回 true，否则函数返回 false。

```
(6) bool StackFull ( Stack& S );
```

先决条件：栈 S 已存在。

操作结果：函数测试栈 S 满否。若栈满，则函数返回 true，否则函数返回 false。

```
(7) int StackSize ( Stack& S );
```

先决条件：栈 S 已存在。

操作结果：函数返回栈 S 的长度，即栈 S 中元素个数。

举例说明栈的应用。利用栈实现向量 $\boldsymbol{A}$ 中所有元素的原地逆置。算法的设计思路是：若设向量 $\boldsymbol{A}$ 中数据原来的排列是$\{a_1,a_2,\cdots,a_n\}$，执行此算法时，把向量中的元素依次进栈。再从栈 S 中依次退栈，存入向量 $\boldsymbol{A}$，从而使得 $\boldsymbol{A}$ 中元素的排列变成$\{a_n,a_{n-1},\cdots,a_1\}$。所谓“原地”是指逆转后的结果仍占用原来的空间。参看程序 3-1。

程序 3-1　向量 A 中所有元素逆置的算法（存放于 prg3-1.cpp）

```
#include "SeqStack.cpp"
void Reverse ( SElemType A[], int n ) {
    Stack S;  InitStack(S);  int i;
    for ( i = 1; i <= n; i++ ) Push (S, A[i-1]);
    while ( !StackEmpty(S) ) Pop (S, A[i-1]);
}
```

想想看　为何栈元素的数据类型要用 SElemType 定义？在何处声明？

3.1.2　顺序栈

顺序栈是栈的顺序存储表示。实际上，顺序栈是指利用一块连续的存储单元作为栈元素的存储空间，只不过在 C 语言中是借用一维数组实现而已。

1. 顺序栈的结构定义

在顺序栈中设置一个一维数组 elem 存放栈的元素，同时附设指针 top 指示栈顶元素的实际位置。在 C 语言中，栈元素是从 elem[0]开始存放的。如图 3-2 所示。

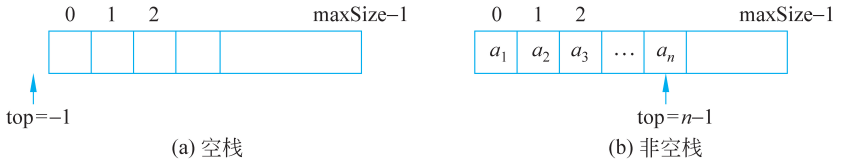


图 3-2 顺序栈的示意图

顺序栈的静态存储结构定义如程序 3-2 所示。

程序 3-2 顺序栈的静态存储结构

```
#define maxSize 100
typedef int SElemType;                          //栈元素数据类型
typedef struct {
    SElemType elem[maxSize];
    int top;
} SeqStack;
```

顺序栈的静态存储结构需要预先定义或申请栈的存储空间，也就是说栈空间的容量有限且一旦装满不能扩充。因此在顺序栈中，当一个元素进栈之前，需要判断是否栈满（栈空间中没有空闲单元），若栈满，则元素进栈会发生上溢现象。特别需要注意的是栈空情形。因为向量下标从 0 开始，栈空时应该 S.top ＜ 0，因此空栈时栈顶指针 S.top ＝＝ －1。

因为在解决应用问题的系统中往往不能精确估计所需栈容量的大小，需要设置足够大的空间。如果程序执行过程中发现上溢，系统就会报错而导致程序停止运行。因此，应采用动态存储分配方式来定义顺序栈，一旦栈满可以自行扩充，避免上溢现象。

顺序栈的动态存储结构定义如程序 3-3 所示。

程序 3-3 顺序栈的动态存储结构（存放于 SeqStack.h）

```
#include<stdio.h>
#include <stdlib.h>
#define initSize 20                             //栈空间初始大小
typedef int SElemType;                          //栈元素数据类型
typedef struct {                                //顺序栈的结构定义
    SElemType *elem;                            //栈元素存储数组
    int maxSize;                                //栈空间最大容量及栈顶指针(下标)
    int top;
} SeqStack;
```

2. 顺序栈主要操作的实现

（1）初始化。程序 3-4 给出动态顺序栈的初始化算法。算法首先按 initSize 大小为顺序栈动态分配存储空间，首地址为 S.elem，并以 initSize 作为最初的 S.maxSize。最后，置栈的初始状态为空（top ＝ －1）。top 指示栈顶，即最后加入元素的存储位置。

程序 3-4　动态顺序栈的初始化（存放于 SeqStack.cpp）

```
#include "SeqStack.h"
void initStack ( SeqStack& S ) {
//建立一个最大尺寸为 initSize 的空栈，若分配不成功则错误处理
    //创建栈空间
    S.elem = ( SElemType * ) malloc ( initSize * sizeof ( SElemType ) );
    if ( S.elem == NULL ) { printf ( "存储分配失败!\n" );  exit(1); }
    S.maxSize = initSize;  S.top = -1;
}
```

（2）进栈。程序 3-5 是动态顺序栈的进栈算法。主要步骤如下：

① 先判断栈是否已满。若栈顶指针 top ＝＝ maxSize－1，说明栈中所有位置均已使用，栈已满。这时新元素进栈将发生栈溢出，程序转入溢出处理，否则直接执行②。

② 先让栈顶指针 top 进 1，指到当前可加入新元素的位置，再按栈顶指针所指位置将新元素插入。这个新插入的元素将成为新的栈顶元素。

程序 3-5　顺序栈进栈操作的实现（存放于 SeqStack.cpp）

```
bool Push( SeqStack& S, SElemType x ) {
//进栈:若栈不满，则将元素 x 插入到该栈的栈顶，否则溢出处理后再进栈
    if ( stackFull (S) ) { printf ( "栈满!\n" );  return false; }     //栈满
    S.elem[++S.top] = x;                                    //栈顶指针先加 1，再进栈
    return true;
}
```

顺序栈进栈操作的示例如图 3-3 所示。栈顶指针指示最后加入元素位置。

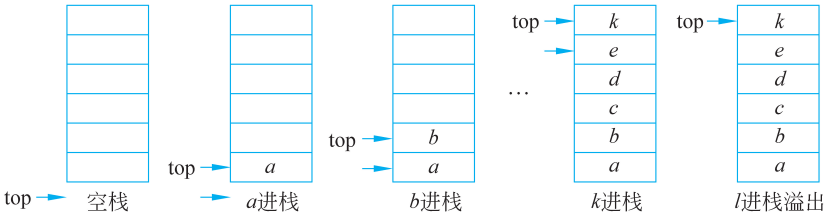


图 3-3　顺序栈的进栈

（3）退栈。程序 3-6 是动态顺序栈的退栈算法。主要步骤如下：

① 先判断是否栈空。若在退栈时发现栈是空的，则退栈操作将执行栈空处理。栈空处理通常表明使用这个栈的算法结束。若栈不空，执行②。

② 先将位于栈顶的元素取出，再让栈顶指针减 1，使栈顶退到次栈顶位置。

程序 3-6　顺序栈退栈操作的实现（存放于 SeqStack.cpp）

```
bool Pop ( SeqStack& S, SElemType& x ) {
    //退栈:若栈不空则函数通过引用参数 x 返回栈顶元素的值，同时栈顶指针退 1，函数返回
    //true;否则函数返回 false，且 x 的值不可引用
    if ( S.top == -1 ) return false;            //判栈空否，若栈空则函数返回 false
    x = S.elem[S.top--];  return true;          //先保存栈顶的值，再让栈顶指针退 1
}
```

顺序栈退栈操作的示例如图 3-4 所示。位于栈顶指针上方的栈空间中即使有元素，它

们也不是栈的元素了。

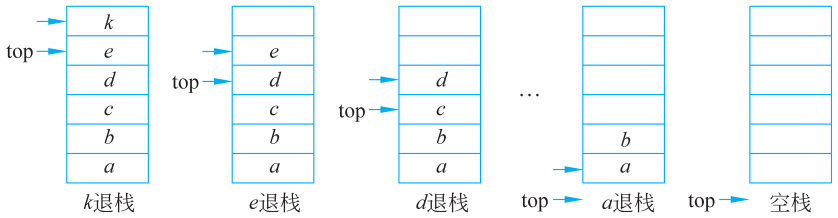


图 3-4　顺序栈的退栈

（4）顺序栈其他操作的实现。参看程序 3-7。需要注意的是 GetTop 操作与 Pop 操作的不同。GetTop 操作在读取栈顶元素的值时栈顶元素不退栈，因此栈顶指针不会被修改。

程序 3-7　顺序栈其他操作的实现（存放于 SeqStack.cpp）

```
bool getTop ( SeqStack& S, SElemType& x ) {
//读取栈顶元素的值:若栈不空则函数返回栈顶元素的值且函数返回 true,否则函数返回 false
    if ( S.top == -1 ) return false;            //判栈空否，若栈空则函数返回 false
    x = S.elem[S.top];  return true;            //返回栈顶元素的值
}
bool stackEmpty ( SeqStack& S ) {
//函数测试栈 S 空否。若栈空,则函数返回 true,否则函数返回 false
    return S.top == -1;
}
bool stackFull ( SeqStack& S ) {
//函数测试栈 S 满否。若栈满,则函数返回 true,否则函数返回 false
    return S.top == S.maxSize-1;
}
int StackSize ( SeqStack& S ) {
//函数返回栈 S 的长度,即栈 S 中元素个数
    return S.top+1;
};
void printStack ( SeqStack& S ) {
//从栈底到栈顶顺序输出栈内元素的值
    for ( int i = 0; i <= S.top; i++ )
        printf ( "%d ", S.elem[i] );
    printf ( "\n" );
}
```

想想看　为何栈操作只有这几个，表的其他一般操作，如 Search、Insert、Remove 是否也可以适用于栈？

（5）栈的扩充。在进栈时首先要判断是否栈已满？一旦发现栈已满就要做溢出处理。一种溢出处理的方法是扩充栈的大小。按 2×S.maxSize 大小申请新的连续存储空间，把原来存储空间中存放的所有栈元素转移到新的存储空间后释放原来的存储空间，再让 S.elem 指针指示新的存储空间，并修改 S.maxSize ＝ 2×S.maxSize，从而解决空间溢出的问题。

程序 3-8　进栈时的溢出处理（存放于 prg3-8.cpp）

```
#include "SeqStack.cpp"
void OverflowProcess ( SeqStack& S ) {
```

```
    //溢出处理:扩充栈的存储空间
    SElemType * temp =
              ( SElemType * ) malloc ( 2 * S.maxSize * sizeof ( SElemType ) );
    if ( temp == NULL ) { printf ( "存储分配失败!\n" );  exit(1); }
    //传送原栈内的数据
    for ( int i = 0; i <= S.top; i++ ) temp[i] = S.elem[i];
    free ( S.elem );                                          //删除原数组
    S.maxSize = 2 * S.maxSize;  S.elem = temp;                //数组最大体积增长一倍
}
void Push( SeqStack& S, SElemType x ) {
    //进栈操作:若栈不满, 则将元素 x 插入到该栈的栈顶, 否则溢出处理后再进栈
    if ( StackFull(S) ) OverflowProcess(S);                   //栈满则溢出处理
       S.elem[++S.top] = x;                                   //栈顶指针先加 1, 再进栈
};
```

因为 S.top 存放的是数组元素的下标,而不是实际存储地址,当 S.elem 数组扩充后,它还是保持原栈顶的位置,所以没有修改。

想想看 为何扩充栈空间要扩大为原来的 2 倍,而不是一点点地扩充?

3. 顺序栈主要操作的性能分析

顺序栈是限定了存取位置的线性表,除溢出处理操作外都只能顺序存取,这决定了各主要操作的性能都良好,设 n 是栈最大容量,则各主要操作的性能如表 3-1 所示。

表 3-1 顺序栈各操作的性能比较

操 作 名	时间复杂度	空间复杂度	操 作 名	时间复杂度	空间复杂度
初始化 InitStack	$O(1)$	$O(1)$	读栈顶 GetTop	$O(1)$	$O(1)$
进栈 Push	$O(1)$	$O(1)$	判栈空 StackEmpty	$O(1)$	$O(1)$
溢出处理 overflowProcess	$O(n)$	$O(n)$	判栈满 StackFull	$O(1)$	$O(1)$
退栈 Pop	$O(1)$	$O(1)$	求栈长 StackSize	$O(1)$	$O(1)$

溢出处理需要开辟一个大小为 2×S.maxSize 的地址连续的附加存储空间,所以该操作的空间复杂度较高,还要把原来栈中所有元素复制过去,时间复杂度也较高。除这个操作外,其他操作的时间和空间复杂度都很低。因此,顺序栈是一种高效的存储结构。

4. 双栈共享同一栈空间

程序中往往同时存在几个栈,因为各个栈所需的空间在运行中是动态变化着的。如果给几个栈分配同样大小的空间,可能在实际运行时,有的栈膨胀得快,很快就产生了溢出,而其他的栈可能此时还有许多空闲的空间。这时就必须调整栈的空间,防止栈的溢出。

当程序同时使用两个栈时,定义一个足够大的栈空间 Vector[maxSize],并将两个栈设为 0 号栈和 1 号栈。该空间的两端的外侧分别设为两个栈的栈底,用 $b[0](=-1)$ 和 $b[1]$ (= maxSize)指示。让两个栈的栈顶 $t[0]$ 和 $t[1]$ 都向中间伸展,直到两个栈的栈顶相遇,才认为发生了溢出,如图 3-5 所示。

进栈情形:对于 0 号栈,每次进栈时栈顶指针 $t[0]$ 加 1,对于 1 号栈,每次进栈时栈顶指针是 $t[1]$ 减 1,当两个栈的栈顶指针相遇,即 $t[0]+1 == t[1]$,才算栈满。

退栈情形:对于 0 号栈,每次退栈时栈顶指针是 $t[0]$ 减 1,对于 1 号栈,每次退栈时栈顶

指针是 $t[1]$加 1，只有当栈顶指针退到栈底才算栈空。

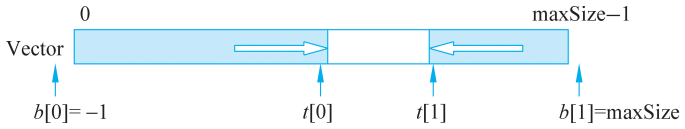


图 3-5　两个栈共享同一存储空间的情形

两栈的大小不是固定不变的。在实际运算过程中，一个栈有可能进栈元素多而体积大些，另一个则可能小些。两个栈共用一个栈空间，互相调剂，灵活性强。

两个栈共享一个栈空间时主要栈操作的实现如程序 3-9 所示。

程序 3-9　两栈共享同一栈空间时的进栈和退栈操作的实现（存放于 prg3-9.cpp）

```
#include<stdio.h>
#include <stdlib.h>
#define maxSize 20                                  //栈存储数组的大小
typedef int SElemType
typedef struct {                                    //双栈的结构定义
    int top[2], bot[2];                             //双栈的栈顶指针和栈底指针
    SElemType elem[maxSize];                        //栈数组
} DblStack;
void initStack ( DblStack& S ) {
//初始化函数:建立一个尺寸为 maxSize 的空栈，若分配不成功则错误处理。
    S.top[0] = S.bot[0] = -1;  S.top[1] = S.bot[1] = maxSize;
}
bool Push ( DblStack& S, SElemType x, int d ) {     //进栈运算
    if ( S.top[0]+1 == S.top[1] ) ) return false;   //栈满则返回 false,不能进栈
    if ( d == 0 ) S.elem[++S.top[0]] = x;           //d = 0:栈顶指针先加 1 再进栈
    else S.elem[--S.top[1]] = x;                    //d = 1:栈顶指针先减 1 再进栈
    return true;
}
bool Pop ( DblStack& S, SElemType& x, int d ) {
    //函数通过引用参数 x 返回退出栈 d 的栈顶元素的元素值,前提是栈不为空
    if ( S.top[d] == S.bot[d] ) return false;       //第 d 个栈栈空,不能退栈
    if ( d == 0 ) x = S.elem[S.top[0]--];           //栈 0:先出栈,栈顶指针减 1
    else x = S.elem[S.top[1]++];                    //栈 1:先出栈,栈顶指针加 1
    return true;
}
bool getTop ( DblStack& S, SElemType& x, int d ) {
    //函数通过引用参数 x 返回栈 d 的栈顶元素的元素值,前提是栈不为空
    if ( S.top[d] == S.bot[d] ) ) return false;
    x = S.elem[S.top[i]];
    return true;
}
```

算法的调用方式与顺序栈的基本运算类似，只是在参数表中多了一个 d，指示操作对象是哪一个栈：$d = 0$ 是 0 号栈，$d = 1$ 是 1 号栈。

如果要求更多的栈共享同一个栈空间，使用顺序存储方式，将使得处理十分复杂。在向其中一个已满的栈中插入一个新元素时，必须向后或向前寻找未满的栈，成块地移动存储，

压缩未满的栈的空间，为要插入元素的栈腾出空间。这导致大量的时间开销，降低了算法的时间效率。特别是当整个存储空间即将充满时，这个问题更加严重。解决的办法就是采用链接方式作为栈的存储表示。

3.1.3 链式栈

链式栈是栈的链接存储表示。采用链式栈来表示一个栈，便于结点的插入与删除。在程序中同时使用多个栈的情况下，用链接表示不仅能够提高效率，还可以达到共享存储空间的目的。链式栈的示意图可参看图 3-6。

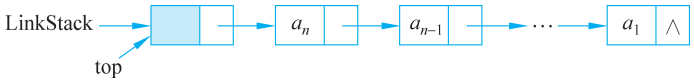


图 3-6 链式栈

从图 3-6 可知，链式栈的栈顶指针在链表头结点，但实际的栈顶结点在头结点后面的首元结点。因此，新结点的插入和栈顶结点的删除都在链表的首元结点，即栈顶进行。

1. 链式栈的结构定义

链式栈用单链表作为它的存储表示，其结构定义如程序 3-10 所示。它使用了一个链表的头指针来表示一个栈。对于需要同时使用多个栈的情形，只要我们声明一个链表指针向量，就能同时定义和使用多个链式栈，并且无须在运算时做存储空间的移动。

程序 3-10 链式栈的定义（存放于 LinkStack.h）

```
#include<stdio.h>
#include <stdlib.h>
typedef int SElemType;
typedef struct node {
    SElemType data;
    struct node * link;
} LinkNode, * LinkList, * LinkStack;
```

2. 链式栈主要操作的实现

链式栈主要操作的实现如程序 3-11 所示。对于链表结构，有一点需要注意：只要存储中还有可分配的空间，就可以申请和分配新的链表结点，使得链表延伸下去，所以理论上讲，链式栈没有栈满问题。但是它有栈空问题。

程序 3-11 链式栈主要操作的实现（存放于 LinkStack.cpp）

```
#include "LinkList.h"
void initStack ( LinkStack& S ) {
    //链式栈初始化:置栈顶指针,即链表头指针为空。
    S = ( LinkNode* ) malloc ( sizeof ( LinkNode ));              //创建头结点
    if ( S == NULL ) { printf ( "存储分配失败!\n" );  exit(1); }
    S->link = NULL;                                               //栈顶置空
};
bool Push ( LinkStack& S, SElemType x ) {
    //进栈:将元素 x 插入到链式栈的栈顶,即链头
    LinkNode *p = ( LinkNode* ) malloc( sizeof( LinkNode ));  //创建新结点 *p
```

```
    p->data = x;  p->link = S->link;  S->link = p;          //新结点插入在链头
    return true;
}
bool Pop ( LinkStack& S, SElemType& x ) {
    //退栈:若栈空,函数返回 false,参数 x 的值不可用;若栈不空,则函数返回 true,并通过
    //引用参数 x 返回被删栈顶元素的值
    if ( S->link == NULL ) return false;                    //栈空函数返回 false
    LinkNode *p = S->link;  x = p->data;                    //存栈顶元素
    S->link = p->link;  free (p);                           //栈顶指针退到次栈顶
    return true;
}
bool GetTop ( LinkStack& S, SElemType& x ) {
    //读取栈顶:若栈不空,函数通过引用参数 x 返回栈顶元素的值
    if ( S->link == NULL ) return false;                    //栈空函数返回 false
    x = S->link->data;  return true;                        //栈不空则返回 true
};
bool stackEmpty ( LinkStack& S ) {
    //判断栈是否为空:若栈空,则函数返回 true,否则函数返回 false
    return S->link == NULL;
}
int stackSize ( LinkStack& S ) {
    //求栈的长度:计算栈元素个数
    int k = 0;                                              //逐个结点计数
    for ( LinkNode *p = S->link; p != NULL, p = p->link ) k++;
    return k;
}
void printStack ( LinkStack& S ) {
    for ( LinkNode *p = S->link; p != NULL; p = p->link )
        printf ( "%d ", p->data );                          //逐个结点输出
    printf ( "\n" );
}
void createLinkStack ( LinkStack& S, SElemType A[ ], int n ) {
    //根据数组 A[n]建立单链表 S,要求 S 已存在并置空
    LinkList p;
    for ( int i = 0; i < n; i++ ) {                         //头插法
        p = ( LinkNode * ) malloc ( sizeof ( LinkNode ) );
        p->data = A[i];    p->link = S->link;  S->link = p;  //链入到头结点之后
    }
}
```

3. 链式栈主要操作的性能分析

链式栈主要操作的性能分析如表 3-2 所示，其中 n 是链式栈中元素个数。

除栈清空、栈建立和计算栈长度这三个操作需要逐个结点处理，时间复杂度达到 $O(n)$ 以外，其他操作的时间和空间性能都相当好。此外，如果事先无法根据问题要求确定栈空间大小时，使用链式栈更好些，因为它没有事先估计栈空间大小的困扰，也不需要事先分配一块足够大的地址连续的存储空间。但是由于每个结点增加了一个链接指针，导致存储密度较低，如果问题明确要求最省空间，可以优先考虑顺序栈。

表 3-2 链式栈各操作性能的比较

操 作 名	时间复杂度	空间复杂度	操 作 名	时间复杂度	空间复杂度
初始化 InitStack	$O(1)$	$O(1)$	读取栈顶 GetTop	$O(1)$	$O(1)$
清空 clearStack	$O(n)$	$O(1)$	判栈空 StackEmpty	$O(1)$	$O(1)$
进栈 Push	$O(1)$	$O(1)$	计算栈长 StackSize	$O(n)$	$O(1)$
退栈 Pop	$O(1)$	$O(1)$	建栈 createLinkStack	$O(n)$	$O(1)$

如果同时使用 n 个链式栈，其头指针数组可以用以下方式定义：

```
int i, n = 10;
LinkNode **st = ( LinkNode ** ) malloc ( n * sizeof (LinkNode * ) );
for ( i = 0; i < n; i++ ) {
    //创建第 i 个链表的头结点
    st[i] = ( LinkNode * ) malloc ( sizeof ( LinkNode ) );
    st[i]->link = NULL;                          //置空该链表
}
```

其中，st[i]是表头指针数组中第 i 个链表的表头指针，它指示第 i 个链表的头结点。

3.1.4 栈的混洗

设给定一个数据元素的序列，通过控制入栈和退栈的时机，可以得到不同的退栈序列，这就称为栈的混洗。在用栈辅助解决问题时，需要考虑混洗问题。那么，对于一个有 n 个元素的序列，如果让各元素按照元素的序号 $1,2,\cdots,n$ 的顺序进栈，可能的退栈序列有多少种？不可能的退栈序列有多少种？现在我们来进行讨论。

当进栈序列为 1 和 2 时，可能的退栈序列有 2 种：{1,2}和{2,1}；当进栈序列为 1、2 和 3 时，可能的退栈序列有 5 种：{1,2,3},{1,3,2},{2,1,3},{2,3,1},{3,2,1}。注意，{3,1,2}是不可能的退栈序列。

一般情形又如何呢？若设进栈序列为 $1,2,\cdots,n$，可能的退栈序列有 m_n 种，则

当 $n=0$ 时无整数进栈，$m_0=1$：退栈序列为{}。

当 $n=1$ 时 1 个整数进栈，$m_1=1$：退栈序列为{1}。

当 $n=2$ 时 2 个整数进栈，$m_2=2$：退栈序列中 1 在首位，1 左侧有 0 个数，右侧有 1 个数，有 $m_0\times m_1=1$ 种退栈序列为{1,2}；退栈序列中 1 在末位，1 左侧有 1 个数，右侧有 0 个数，有 $m_1\times m_0=1$ 种退栈序列{2,1}；总的可能退栈序列有 $m_0\times m_1+m_1\times m_0=2$ 种。

当 $n=3$ 时 3 个整数进栈，$m_3=5$：退栈序列中 1 在首位，1 左侧有 0 个数，右侧有 2 个数，有 $m_0\times m_2=2$ 种退栈序列{1,2,3}和{1,3,2}；退栈序列中 1 在第 2 位，1 左侧 1 个数，右侧 1 个数，有 $m_1\times m_1=1$ 种退栈序列{2,1,3}；退栈序列中 1 在第 3 位。1 左侧 2 个数，右侧 0 个数，有 $m_2\times m_0=2$ 种退栈序列{2,3,1}和{3,2,1}；因此，总的可能退栈序列有 $m_0\times m_2+m_1\times m_1+m_2\times m_0=5$ 种。

当 $n=4$ 时有 4 个整数依次进栈，按照上面的方法推理，可能的退栈序列栈 $m_4=m_0\times m_3+m_1\times m_2+m_2\times m_1+m_3\times m_0=1\times5+1\times2+2\times1+5\times1=14$ 种。它们是{1,2,3,4},{1,2,4,3},{1,3,2,4},{1,3,4,2},{1,4,3,2},{2,1,3,4},{2,1,4,3},{2,3,1,4},{3,2,

1,4},{2,3,4,1},{2,4,3,1},{3,2,4,1},{3,4,2,1},{4,3,2,1}。

一般地，设有 n 个元素按序号 1,2,…,n 进栈，轮流让 1 在退栈序列的第 1，第 2，…，第 n 位，则可能的退栈序列数为：

$$\sum_{i=0}^{n-1} m_i \times m_{n-i-1} = \frac{1}{n+1} C_{2n}^{n} = \frac{(2n)(2n-1)\cdots(n+2)}{n!}$$

再看不可能的退栈序列又是什么情况。设对于初始进栈序列 1,2,3,…,n，利用栈得到可能的退栈序列为 $p_1 p_2 \cdots p_i \cdots p_n$，如果序号 $i<j<k$，且在进栈序列中 $p_i<p_j<p_k$，即：

$$\cdots p_i, \cdots, p_j, \cdots, p_k \quad (p_i < p_j < p_k)$$

则：

$$\cdots p_k, \cdots, p_i, \cdots, p_j$$

就是不可能的退栈序列。因为 p_k 在 p_i 和 p_j 之后进栈，又先于 p_i 和 p_j 退栈，按照栈的后进先出的特性，p_i 压在 p_j 的下面，理应 p_j 先出，所以…p_k,…,p_i,…,p_j…是不可能的退栈序列。

那么，设一个进栈序列有 n 个互不相等的整数，如何求得所有可能的出栈序列？

考虑采用递归方法求解。算法需要使用一个辅助栈保存生成的出栈序列。假设 $n=3$，进栈序列为 1、2 和 3。若进栈标记为 I，出栈标记为 O，可用如图 3-7 所示的状态树来表示进栈与出栈情形。由于进栈操作(I)在任何情况下都应比它右边的出栈操作(O)多，不合理的情形都被剪枝，所以在状态树中剪枝部分都没有画出来，树中从左向右的状态分别是 $III OOO$(输出 321)，$IIOIOO$(输出 231)，$IIOOIO$(输出 213)，$IOIIOO$(输出 132)，$IOIOIO$(输出 123)。算法的思路就是遍历此状态树，按向左进栈，向右出栈，出栈时输出的原则进行处理。每个分支代表一个输出序列。

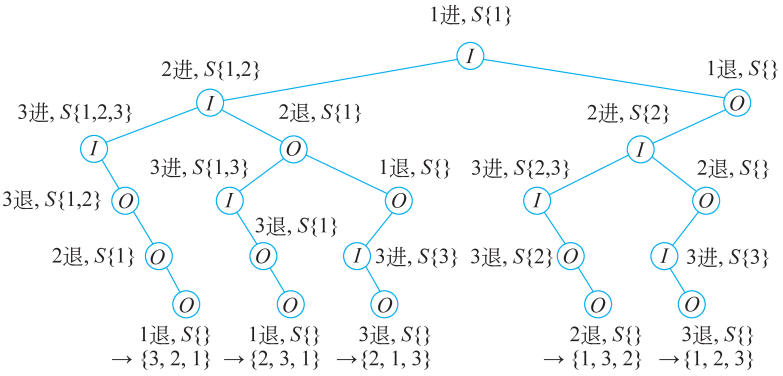


图 3-7 表示进/退栈的状态树

程序 3-12 求进栈序列的所有退栈序列(存放于 prg3-12.cpp)

```
#include "SeqStack.cpp"
#define N 3
void allOutSTK( int A[], int C[], SeqStack& S, int i, int k, int& count ) {
//设进栈序列有 N 个不同整数,算法参数表中数组 C[N]是进栈序列,A[k]是退栈序列,k
//是 A 中当前形成的序列中元素数(初值为 0),i 是 C 中当时取元素指针(初值为 0),S
//存放已进栈序列,count 用于统计出栈序列个数(初值为 0)
    int x, j;
```

```
    if ( i == N && stackEmpty (S) ) {          //i 已取完进栈元素且栈 S 空
        //输出当前形成的退栈序列
        for ( j = 0; j < k; j++ ) printf ( "%d ", C[A[j]] );
        printf ( "\n" );
        count++;                               //退栈序列数加 1
    }
    if ( i < N ) {                             //未取完进栈元素,向左一步
        Push ( S, i );                         //取进栈元素指针保存
        allOutSTK ( A, C, S, i+1, k, count );  //递归继续形成退栈序列
        Pop ( S, x );
    }
    if ( !stackEmpty ( S ) ) {                 //向右
        Pop ( S, x );  A[k] = x;               //退栈
        allOutSTK ( A, C, S, i, k+1, count );  //递归继续形成退栈序列
        Push ( S, x );
    }
}
```

图 3-8 是对进栈序列{1,2,3}执行程序 3-11 所得到的结果。

```
进栈序列 1 2 3
所有出栈序列为:
3 2 1
2 3 1
2 1 3
1 3 2
1 2 3
出栈序列个数 5
Press any key to continue
```

图 3-8　求进栈序列的所有退栈序列的示例

设进栈序列有 n 个元素,算法的时间复杂度为 $O(n^2)$,空间复杂度为 $O(n)$。

再看一个 $n=4$ 的进栈/退栈的例子。设一个进栈序列为 *abcd*,可能的退栈序列有 14 种,即 *abcd*、*abdc*、*acbd*、*acdb*、*adcb*、*bacd*、*badc*、*bcad*、*cbad*、*bcda*、*bdca*、*cbda*、*cdba*、*dcba*;不可能的退栈序列有 4! −14=24-10=10 种,原因参看表 3-3。

表 3-3　不可能的退栈序列

不可能退栈序列	不可能的原因
adbc	*b* 先于 *c* 进栈,*d* 退栈时 *b* 一定压在 *c* 下,不可能 *b* 先于 *c* 退栈
bdac	*a* 先于 *c* 进栈,*d* 退栈时 *a* 一定压在 *c* 下,不可能 *a* 先于 *c* 退栈
cabd	*a* 先于 *b* 进栈,*c* 退栈时 *a* 一定压在 *b* 下,不可能 *a* 先于 *b* 退栈
cadb	同上
cdab	同上
dabc	按照 *a*, *b*, *c*, *d* 顺序进栈,*d* 先退栈,*a*, *b*, *c* 一定还在栈内,且 *a* 压在最下面,*b* 压在 *c* 下面,不可能 *b* 先于 *c* 或 *a* 先于 *b* 退栈
dacb	*a* 先于 *b* 进栈,*d* 退栈时 *a* 一定压在 *b* 下,不可能 *a* 先于 *b* 退栈
dcab	同上

续表

不可能退栈序列	不可能的原因
dbac	*a* 先于 *c* 进栈，*d* 退栈时 *a* 一定压在 *c* 下，不可能 *a* 先于 *c* 退栈
dbca	*b* 先于 *c* 进栈，*d* 退栈时 *b* 一定压在 *c* 下，不可能 *b* 先于 *c* 退栈

3.2 队　　列

3.2.1 队列的概念

1. 队列的定义和特性

队列(Queue)是另一种限定存取位置的线性表。它只允许在表的一端插入，在另一端删除。允许插入的一端称为队尾(rear)，允许删除的一端称为队头(front)。参看图 3-9。每次在队尾加入新元素，因此元素加入队列的顺序依次为 $a_1,a_2,a_3,\cdots,a_n$。最先进入队列的元素最先退出队列，如同在铁路车站售票口排队买票一样。队列所具有的这种特性就称为先进先出 FIFO(First In/First Out)。

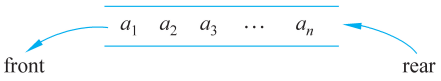


图 3-9　队列的示意图

2. 队列的主要操作

```
(1) 队列的初始化 InitQueue ( Queue& Q );
```

先决条件：无。

操作结果：为队列 *Q* 分配存储空间，并对各数据成员赋初值。

```
(2) void EnQueue ( Queue& Q, QElemType x );
```

先决条件：队列 *Q* 已存在且未满。

操作结果：新元素 *x* 进队列 *Q* 并成为新的队尾。

```
(3) int DeQueue ( Queue& Q, QElemType& x );
```

先决条件：队列 *Q* 已存在且队列 *Q* 非空。

操作结果：若队列 *Q* 空，则函数返回 0，*x* 不可用；否则队列 *Q* 的队头元素退队列，退出元素由 *x* 返回，函数返回 1。

```
(4) int GetFront( Queue& Q, QElemType& x );
```

先决条件：队列 *Q* 已存在且队列非空。

操作结果：若队列 *Q* 空，则函数返回 0，*x* 不可用；否则由引用型参数 *x* 返回队列元素的值但不退队列，函数返回 1。

```
(5) int QueueEmpty( Queue& Q );
```

先决条件：队列 Q 已存在。

操作结果：判队列 Q 空否。若队列空，则函数返回 1，否则函数返回 0。

```
(6) int QueueFull( Queue& Q );
```

先决条件：队列 Q 已存在。

操作结果：判队列 Q 满否。若队列满，则函数返回 1，否则函数返回 0。

```
(7) int QueueSize( Queue& Q );
```

先决条件：队列 Q 已存在。

操作结果：函数返回队列 Q 的长度，即队列 Q 中元素个数。

下面看一个例子。给出一个整数序列，判断它是否中心对称。中心对称是指一个整数序列以位于中间位置的整数为基准两侧的整数完全对称相等。例如，整数序列{1,3,5,3,1}或{1,3,5,5,3,1}即为中心对称，而{1,3,5,7,9}就不是中心对称。

我们可以利用一个队列和一个栈来解决此问题。首先把给定的整数序列分别存入栈和队列，然后依次从栈和队列中退出一个整数，比较出队列的整数和退栈的整数是否相等。若不相等，则该整数序列不是中心对称；若相等则从队列和栈中再退出一个整数，重复上述的比较。如果队列和栈同时为空，则说明对应整数全部相等，该整数序列是中心对称。

算法的实现如程序 3-13 所示。

程序 3-13　检测整数序列 A 是否中心对称的算法（存放于 prg3-13.cpp）

```
#include "SeqStack.cpp"
#include "CircQueue.cpp"
bool centroSym ( int A[], int n ) {
//函数判断整数数组 A[n]是否中心对称。是则函数返回 true,否则函数返回 false
    SeqStack S;  initStack(S);                          //定义一个栈 S 并初始化
    CircQueue Q;  initQueue(Q);                         //定义一个队列 Q 并初始化
    int i, j;
    for ( i = 0; i < n; i++ )                           //逐个整数处理
        { enQueue ( Q, A[i] );  Push ( S, A[i]); }      //分别进队列和进栈
    while ( !stackEmpty(S) && !queueEmpty(Q) ) {        //当队列和栈不空时
        Pop ( S, i );  deQueue (Q, j );                 //分别退出一个整数
        if ( i != j ) return false;                     //对应整数不等,不是中心对称
    }
    return true;                                        //全部相等,中心对称
};
```

3.2.2　循环队列

1. 循环队列的概念

队列的存储表示也有两种方式：一种是基于数组的存储表示，另一种是基于链表的存储表示。队列的基于数组的存储表示亦称为顺序队列，它是利用了一个一维数组 elem[maxSize]来存放队列元素，并且设置两个指针 front 和 rear，分别指示队列的队头和队尾位置。

图 3-10 即为顺序队列操作的示例。maxSize 是数组的最大长度。

从图 3-10 中可以看到，在队列刚建立时，需要首先对它初始化，令 front = rear = 0。

每当加入一个新元素时，先将新元素添加到 rear 所指位置，再让队尾指针 rear 进 1。因而指针 rear 指示了实际队尾位置的后一位置，即下一元素应当加入的位置。而队头指针 front 则不然，它指示真正队头元素所在位置。

如果要退出队头元素，应当首先把 front 所指位置上的元素值记录下来，再让队头指针 front 进 1，指示下一队头元素位置，最后把记录下来的元素值返回。

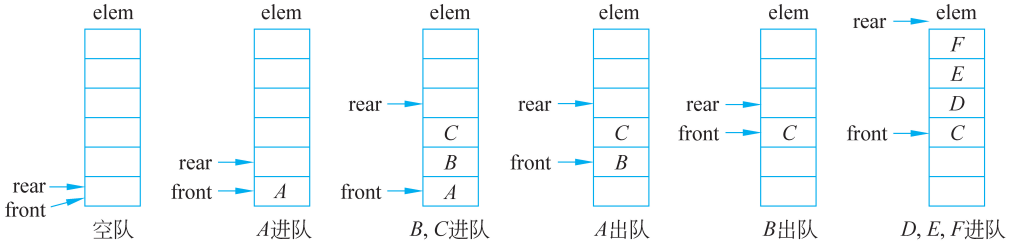


图 3-10 顺序队列的初始化、插入和删除的图示

从图 3-10 中还可以看到，如果队列指针 front == rear，队列为空；而当 rear == maxSize 时，队列满，如果再加入新元素，就会产生“溢出”。

但是，这种“溢出”可能是假溢出，因为在数组的前端可能还有空位置。为了能够充分地使用数组中的存储空间，把数组的前端和后端连接起来，形成一个环形表，即把存储队列元素的表从逻辑上看成一个环，成为循环队列(Circular Queue)。

如图 3-11 所示。循环队列的首尾相接，当队头指针 front 和队尾指针 rear 进到 maxSize－1 后，再前进一个位置就自动到 0。这可以利用整数除取余的运算(%)来实现。

队头指针进 1：

```
front = (front+1) % maxSize;
```

队尾指针进 1：

```
rear = (rear+1) % maxSize;
```

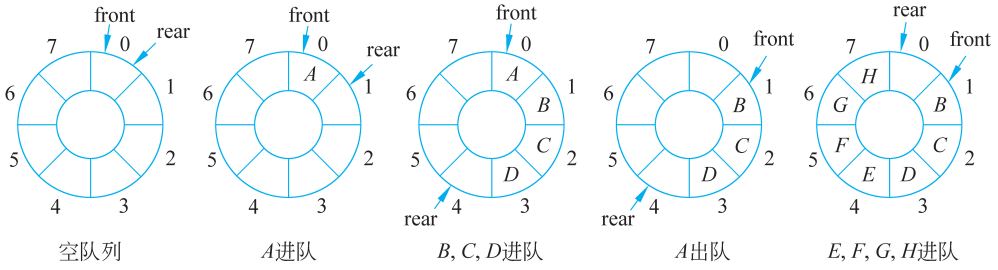


图 3-11 循环队列的初始化、插入和删除的示意图

循环队列的队头指针和队尾指针初始化时都置为 0：front = rear = 0。在队尾插入新元素和删除队头元素时，两个指针都按顺时针方向进 1。当它们进到 maxSize－1 时，并不

表示表的终结，只要有需要，利用 %（取模或取余数）运算可以前进到数组的 0 号位置。

如果循环队列取出元素的速度快于存入元素的速度，队头指针很快追上了队尾指针，一旦到达 front == rear，队列变空。反之，如果队列存入元素的速度快于取出元素的速度，队尾指针很快就赶上了队头指针，使得队列变满。为了区别于队空条件，用（rear+1）% maxSize == front 来判断是否队满，就是说，让 rear 指到 front 的前一位置就认为队已满。此时，因队尾指针指示实际队尾的后一位置，所以在队满时实际空了一个元素位置。如果不留这个空位置，让队尾指针一直走到这个位置。必然有 rear == front，则队空条件和队满条件就混淆了。除非另加队空/队满标志，否则无从分辨到底是队空，还是队满。

想想看 在循环队列中，最多只能存放 maxSize−1 个元素。在设计队列时，若估计队列需要容纳 n 个元素，则队列容量至少应设计多大？

循环队列还有其他可能的实现方式，如：

（1）使用队头指针 front、队尾指针 rear 加上一个进队/出队的标志 tag，可以实现循环队列。新元素进队列时置 tag = 1，以 rear == front 和 tag == 1 作为判断队列是否为满的条件；元素出队列时置 tag = 0，以 rear == front 和 tag == 0 作为判断队列是否为空的条件。队列的进队列和出队列操作仍然按加 1 取模来实现。

（2）使用队尾指针 rear 和队列长度 length，也可实现循环队列。

想想看 使用队尾指针 rear 时队头在何处？是在 rear+1 处，还是在 rear−length+1 处？是否需要用 % maxSize 处理一下？

2. 循环队列的结构定义

循环队列使用了一个容量为 maxSize 的元素数组 elem，还需要两个指针 front 和 rear，这样可定义如程序 3-14 所示的循环队列的结构。

程序 3-14 循环队列的结构定义（存放于 CircQueue.h）

```
#include <stdio.h>
#include <stdlib.h>
typedef char QElemType;
typedef struct {
    QElemType * elem;                    //循环队列的存储空间
    int maxSize;                         //最大存储单元数
    int front, rear;                     //front 指示队头、rear 指示队尾
} CircQueue;
```

队列有许多实际应用，例如，作为系统的输入缓冲区，一旦队列满，系统将产生中断，直到队列元素取走为止；作为系统的输出缓冲区，一旦队列空，系统将产生输出中断，直到队列中重新填充入元素为止。

3. 循环队列主要操作的实现

循环队列主要操作的实现如程序 3-15 所示。注意，队头和队尾指针都在同一方向进 1，不像栈的栈顶指针那样，进栈和退栈是相反的两个方向。

程序 3-15 循环队列主要操作的实现（存放于 CircQueue.cpp）

```
#include "CircQueue.h"
void initQueue ( CircQueue& Q ) {              //循环队列的初始化
```

```
    Q.elem = (QElemType *)malloc(queSize * sizeof(QElemType));
    if ( Q.elem == NULL ) { printf ( "存储分配失败!\n" );  exit(1); }
    Q.maxSize = queSize;  Q.front = 0;  Q.rear = 0;
}
bool queueEmpty ( CircQueue& Q ) {              //判队空否,队空则返回 true
    return Q.front == Q.rear;
}
bool queueFull ( CircQueue& Q ) {               //判队满否,队满则返回 true
    return (Q.rear+1) % Q.maxSize == Q.front;
}
int queueSize ( CircQueue& Q ) {                //求队列元素个数
    return (Q.rear-Q.front+Q.maxSize) % Q.maxSize;
}
bool enQueue ( CircQueue& Q, QElemType x ) {
//若队列不满，则将元素 x 插入队尾，函数返回 true,否则函数返回 false,不能进队
    if ( queueFull (Q) ) return false;
    Q.elem[Q.rear] = x;                         //按照队尾指针指示位置插入
    Q.rear = (Q.rear+1) % Q.maxSize;            //队尾指针进 1
    return true;
}
bool deQueue ( CircQueue& Q, QElemType& x ) {
//若队列已空则不能出队,函数返回 false;若队列不空则函数退掉队头元素并通过
//引用参数 x 返回出队元素的值,函数返回 true
    if ( queueEmpty (Q) ) return false;
    x = Q.elem[Q.front];                        //存队头元素的值
    Q.front = (Q.front+1) % Q.maxSize;          //队头指针进 1
    return true;
}
bool getFront( CircQueue& Q, QElemType& x ) {
//若队列不空则函数返回队头元素的值,不退出队头元素
    if ( queueEmpty (Q) ) return false;
    x = Q.elem[Q.front];  return true;          //x 返回队头元素的值
}
void printQueue ( CircQueue& Q ) {
    for ( int i = Q.front; i < Q.rear; i = (i+1) % Q.maxSize )
        printf ( "%c ", Q.elem[i] );
    printf ( "\n" );
}
```

想想看 Q.rear－Q.front 的结果在什么情况下是正数，在什么情况下是负数？

4. 循环队列主要操作的性能分析

循环队列主要操作的性能如表 3-4 所示。

表 3-4 循环队列各种操作的性能比较

操 作 名	时间复杂度	空间复杂度	操 作 名	时间复杂度	空间复杂度
初始化 initQueue	$O(1)$	$O(1)$	判队列空否 QueueEmpty	$O(1)$	$O(1)$
进队列 enQueue	$O(1)$	$O(1)$	判队列满否 QueueFull	$O(1)$	$O(1)$
出队列 deQueue	$O(1)$	$O(1)$	求队列长度 QueueSize	$O(1)$	$O(1)$
读取队头 getFront	$O(1)$	$O(1)$	输出队列 printQueue	$O(n)$	$O(1)$

除输出队列操作外，循环队列所有操作的时间复杂度和空间复杂度都为 $O(1)$，其性能良好。注意，空间复杂度是指操作中所使用的附加空间的复杂度，它是常数级的，包括 0 个附加空间。

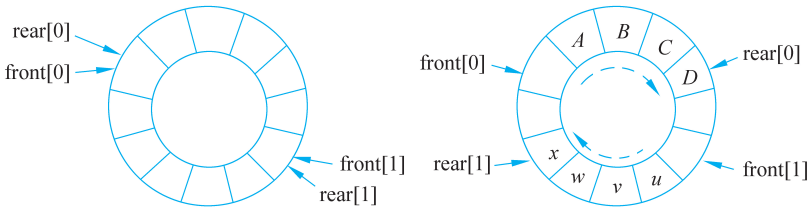
想想看 在后续章节，如树与二叉树、图、排序等，如果在它们的算法中使用了顺序栈或循环队列，这些栈或队列是那些算法的附加空间。在这种场合，栈或队列涉及的元素数组是否应计入算法的空间复杂度度量内？

3.2.3 双循环队列

假设两个队列共享一个首尾相连的环形向量空间，如图 3-12 所示，则称其为双循环队列。好处是两个队列空间的使用可能不相同，一个可能进得多一些，一个可能进得少一些，它们共享一个向量空间，可以互相调剂，更有效地利用存储空间。

1. 双循环队列的结构定义

双循环队列共享同一存储空间，其结构类型如程序 3-16 所示。



(a) 双循环队列初始状态　　(b) 加入数据之后的状态

图 3-12 双循环队列的示例

程序 3-16 双循环队列的结构定义（存放于 DualQueue.h）

```
#include<stdio.h>
#include <stdlib.h>
typedef int QElemType;
typedef struct {
    QElemType elem[maxSize];                    //共享存储空间
    int front[2], rear[2];                      //队列 0 和队列 1 的队头、队尾指针
} DualQueue;
```

初始时，队列 0 的队头指针 front[0]和队尾指针 rear[0]，队列 1 的队头指针 front[1]和队尾指针 rear[1]各自指向环形存储数组的同一个位置，如图 3-12(a)所示，它们相距 maxSize/2 个位置。在使用两个队列的过程中，它们的队头、队尾指针都向顺时针方向移动，如图 3-12 (b)所示。若一个队列编号为 0，那么另一个队列的编号就为 1。在程序中用 i (= 0 或 1)和 $1-i$(= 1 或 0)代表它们的编号。

进队方式是先让队尾指针进 1，再按队尾指针所指位置将新元素进队，出队方式是先让队头指针进 1，再把队头元素取出。这样，队列 i 的队尾指针 rear[i]指示实际队尾元素位置，队头指针 front[i]指示实际队头元素的前一位置，队列 i 空的条件为 rear[i] = front[i]，队列 i 满的条件为 rear[i] = front[$1-i$]，表明另一个的队尾追上了本队列的队头。

为充分利用存储空间，在队列 i 满的情况下，若又有新元素要进队，应当再判断另一个

队列是否已满，若未满，可以把另一个队列顺时针移动一个元素位置，为队列 i 空出一个位置，以插入新元素；若另一个队列已满，则报错。

2. 队列运算的实现

共享存储空间采用静态存储分配，在程序运行期间空间大小不会改变。所以存储空间被视为首尾相连的环形数组，类似于循环队列，指针顺时针移动时走到 maxSize－1 位置，下一步将进到 0 位置，这就用到整数的除余运算（%）。相应算法的实现参看程序 3-17。

程序 3-17　双循环队列主要操作的实现（存放于 DualQueue.cpp）

```
#include "DualQueue.h"
void initQueue ( DualQueue& Q ) {                          //双循环队列的初始化
    Q.front[0] = 0;  Q.rear[0] = 0;
    Q.front[1] = maxSize/2;  Q.rear[1] = maxSize/2;
}
bool queueFull ( DualQueue Q, int i ) {                    //队列 i 判满
    return Q.rear[i] == Q.front[1-i] || Q.rear[1-i] == Q.front[i];
}
bool queueEmpty ( DualQueue Q, int i ) {                   //队列 i 判空
    return Q.front[i] == Q.rear[i];
}
bool enQueue ( DualQueue& Q, int i, QElemType x ) {
//将 x 进到双循环队列 Q 中，i 指定进 0 号还是 1 号队列。若进队列成功函数返回 true
    if ( i < 0 || i > 1 ) return false;
    if ( Q.rear[i] == Q.front[1-i] ) {
        if ( Q.rear[1-i] == Q.front[i] ) return false;     //另一队列也满返回 0
        int j = Q.rear[1-i];                               //另一队列前移一个位置
        while ( j > Q.front[1-i] ) {
            Q.elem[(j+1) % maxSize] = Q.elem[j];           //顺时针前移元素
            j = ( j-1+maxSize ) % maxSize;
        }
        Q.rear[1-i] = ( Q.rear[1-i]+1 ) % maxSize;         //修改队列 1-i 的指针
        Q.front[1-i] = ( Q.front[1-i]+1 ) % maxSize;
    }
    Q.rear[i] = ( Q.rear[i]+1 ) % maxSize;                 //队列 i 的队尾指针进 1
    Q.elem[Q.rear[i]] = x;
      return true;
}
bool deQueue ( DualQueue& Q, int i, QElemType& x ) {
//从指定第 i 号队列中退出队头元素，且通过引用参数 x 返回其值，函数返回
//true；否则函数返回 false，引用参数 x 的值不可用
    if ( i < 0 || i > 1 ) return false;
    Q.front[i] = ( Q.front[i]+1 ) % maxSize;
    x = Q.elem[Q.front[i]];
    return true;
}
```

进队和出队算法的时间复杂度和空间复杂度均为 $O(1)$。

3.2.4　链式队列

1. 链式队列的概念

链式队列是队列的基于单链表的存储表示，如图 3-13 所示。在单链表的每一个结点中

有两个域：data 域存放队列元素的值，link 域存放单链表下一个结点的地址。

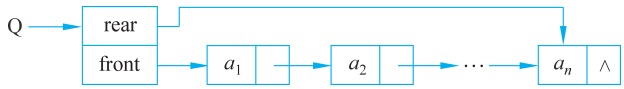


图 3-13　链式队列

为了操作方便，通常链式队列不设头结点，队列的队头指针直接指向单链表的首元结点，这意味着实际队头就在链表的首元结点。若要从队列中退出一个元素，只需从单链表中删除首元结点即可。而队列的队尾指针指向单链表的尾结点，存放新元素的结点应插在队列的队尾，即单链表的最后一个结点后面，这个新结点将成为新的队尾。

用单链表表示的链式队列特别适合于数据元素变动比较大的情形，而且不存在队列满而产生溢出的情况。另外，假若程序中要使用多个队列，与多个栈的情形一样，最好使用链式队列。这样不会出现存储分配不合理的问题，也不需要进行存储的移动。

2. 链式队列的结构定义

链式队列实际上就是单链表，每个结点的定义与单链表结点相同，而链表则设置了两个指针：队头指针 front 指向链表的头结点，而头结点的下一结点才是队列的队头结点；队尾指针 rear 指向链尾结点。队列所有结点的访问都必须从队头开始顺序访问。队列的结构定义如程序 3-18 所示。

程序 3-18　链式队列的结构定义（存放于 LinkQueue.h）

```
#include<stdio.h>
#include <stdlib.h>
typedef int QElemType;
typedef struct Node {                          //链式队列结点定义
    QElemType data;                            //结点的数据
    struct Node * link;                        //结点的链接指针
} LinkNode;
typedef struct {                               //链式队列定义
   LinkNode * front, * rear;                   //队列的队头和队尾指针
} LinkQueue;
```

3. 链式队列主要操作的实现

链式队列主要操作的实现如程序 3-19 所示，与单链表不同的是，插入和删除仅限于链表的两端：插入（进队列）在链尾，新结点成为新的链尾；删除（出队列）在链头，删除链表的首元结点，而首元结点的下一结点成为新的首元结点。

程序 3-19　链式队列主要操作的实现（存放于 LinkQueue.cpp）

```
#include "LinkQueue.h"
void initQueue ( LinkQueue& Q ) {                    //队列初始化
    Q.front = Q.rear = NULL;                         //队头与队尾指针初始化
};
void clearQueue ( LinkQueue& Q ) {                   //清空队列
    LinkNode * p;
    while ( Q.front != NULL ) {                      //逐个删除队列中的结点
        p = Q.front->link;  Q.front = p->link;       //摘下首元结点，修改队头
        free (p);
```