第3章　栈和队列

40多年前，笔者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，一路在洞壁上和地上做标记，最后依靠这些标记退出洞外。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，这两种结构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，这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一样。

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叫做栈顶(top)，而不允许插入和删除的另一端叫做栈底(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。栈中的元素按 $a_1,a_2,\cdots,a_n$ 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，即后进者先出。因此，栈又叫做后进先出(LIFO，Last In First Out)的线性表。栈的逻辑结构示意如图3-1所示。

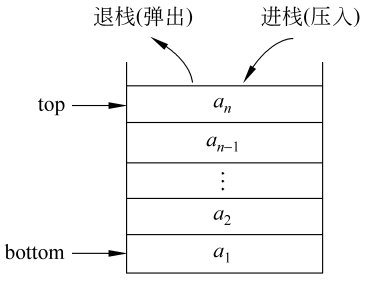

图3-1　栈的示意图

3. 栈的主要操作

(1) 栈的初始化 `InitStack(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) `void 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) int 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 0，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 1。

(4) int 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 0，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 1。

(5) int 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 1，否则函数返回 0

(6) int 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 1，否则函数返回 0。

(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 中的元素个数。

【例 3-1】 利用栈实现向量 A 中所有元素的原地逆置。算法的设计思路是：若设向量 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 A，从而使得 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。参见程序 3-1。

程序 3-1 向量 A 中所有元素逆置的算法

```
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向量 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，栈元素是从数组 elem[0]开始存放的，如图 3-2 所示。

顺序栈的静态存储结构定义如程序 3-2 所示。

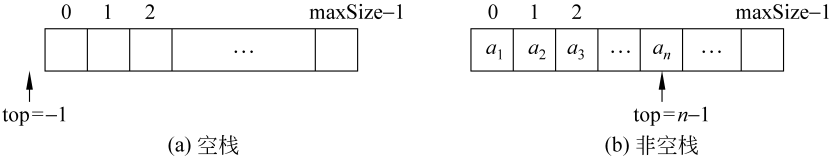

图 3-2　顺序栈的示意图

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

```
#define initSize 100                            //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,top;                            //栈空间最大容量及栈顶指针(下标)
}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，主要步骤如下：

① 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。

② 一旦需要扩充，按 2S. 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＝2S. maxSize，从而解决空间上溢的问题。

程序 3-4　动态顺序栈的初始化

```
void InitStack(SeqStack& S){
//建立一个最大尺寸为 initSize 的空栈,若分配不成功则执行错误处理
    S.elem=new SElemType[initSize];             //创建栈的存储空间
```

```
    if(!S.elem){cerr<<"存储分配失败!\n"; exit(1);}   //链接 stdlib.h 和 iostream.h
    S.maxSize=initSize; S.top=-1;
};
```

top 指示的是最后加入的元素的存储位置。

(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,然后做②;否则直接做②。

② 先让栈顶指针加 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 动态顺序栈进栈操作的实现

```
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,否则做溢出处理后再进栈
    if(StackFull(S))OverflowProcess(S);                    //栈满则做溢出处理
    S.elem[++S.top]=x;                                     //栈顶指针先加 1,再进栈
};
void OverflowProcess(SeqStack& S){
//溢出处理:扩充栈的存储空间
    SElemType * newArray=new SElemType[S.maxSize * 2];
    if(!newArray){cerr<<"存储分配失败!\n"; exit(1);}       //链接 stdlib.h 和 iostream.h
    for(int i=0;i<=S.top;i++)newArray[i]=S.elem[i];
    S.maxSize=2 * S.maxSize;
    delete []S.elem;
    S.elem=newArray;
};
```

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。

【例 3-2】 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的元素位置。

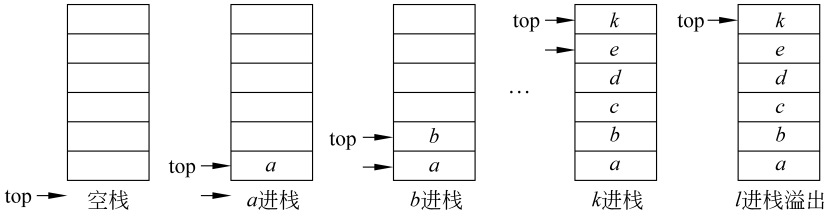

图 3-3 顺序栈的进栈操作示意图

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) 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,做②。

② 先将栈顶指针所指示栈顶的元素取出,再让栈顶指针减 1,等于栈顶退回到次栈顶位置。

程序 3-6　顺序栈退栈操作的实现

```
int Pop(SeqStack& S,SElemType& x){
//退栈：若栈不空则函数通过引用型参数 x 返回栈顶元素的值
//同时栈顶指针减 1,函数返回 1;否则函数返回 0,且 x 的值不可引用
    if(S.top==-1)return 0;              //判栈空否,若栈空则函数返回 0
    x=S.elem[S.top--];                  //栈顶指针减 1
    return 1;                           //退栈成功,函数返回 1
};
```

【例 3-3】 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。

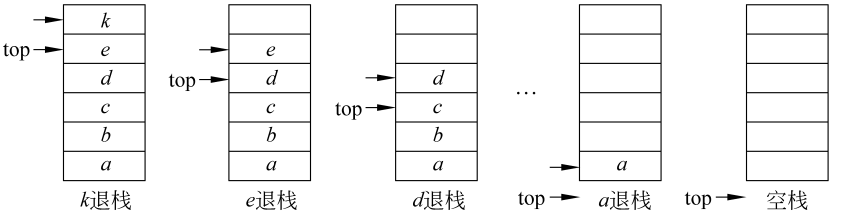

图 3-4　顺序栈的退栈操作示意图

(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

```
int 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 1,否则函数返回 0
    if(S.top==-1)return 0;              //判栈空否,若栈空则函数返回 0
    x=S.elem[S.top];                    //返回栈顶元素的值
    return 1;
};
int StackEmpty(SeqStack& S){
//函数测试栈 S 空否。若栈空,则函数返回 1,否则函数返回 0
    return S.top==-1;                   //函数返回布尔式 S.top==-1 的运算结果
};
int StackFull(SeqStack& S){
//函数测试栈 S 满否。若栈满,则函数返回 1,否则函数返回 0
    return S.top==S.maxSize;            //函数返回布尔式 S.top==S.maxSize 的运算结果
};
int StackSize(SeqStack& S){
//函数返回栈 S 的长度,即栈 S 中的元素个数
    return S.top+1;
};
```

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

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十分良好，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 2S.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，其他的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。

【例 3-4】 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。

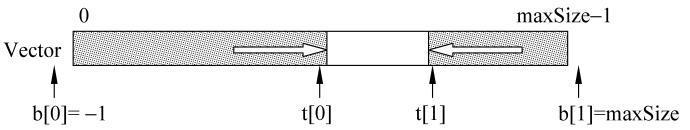

图 3-5　两个栈共享同一存储空间的情形

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 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 t[0]减 1；对于 1 号栈，每次退栈时栈顶指针 t[1]加 1。只有当栈顶指针退到栈底才算栈空。

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 3-8 所示。

程序 3-8　两栈共享同一栈空间时的进栈和退栈操作的实现

```
int push(DualStack& DS,SElemType x,int d){
//在 d 所指示的栈中插入元素 x。d=0,插在 0 号栈;d=1,插在 1 号栈
//若插入成功,函数返回 1,否则函数返回 0
    if(DS.t[0]+1==DS.t[1]||d!=0 && d!=1)return 0;
    if(d==0)DS.t[0]++;                          //栈顶指针加 1
```

```
    else DS.t[1]--;
    DS.Vector[DS.t[d]]=x;                          //进栈
    return 1;
};
int Pop(DualStack& DS,SElemType& x,int d){
//从 d 所指示的栈中退出栈顶元素,通过引用型参数 x 返回。d=0,从 0 号栈退栈;d=1,
//从 1 号栈退栈。若退栈成功,函数返回 1;否则函数返回 0
    if(d!=0 && d!=1|| DS.t[d]==DS.b[d])return 0;
    x=DS.Vector[DS.t[d]];                          //取出栈顶元素的值
    if(d==0)DS.t[0]--;                             //栈顶指针减 1
    else DS.t[1]++;
    return 1;
};
```

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。链式栈的示意如图 3-6 所示。

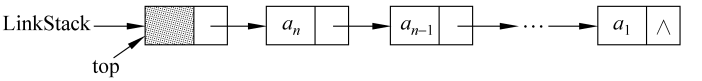

图 3-6 链式栈

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 3-9 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 3-9 链式栈的定义

```
typedef int SElemType;
typedef struct node{
    SElemType data;
    struct node * link;
} LinkNode, * 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 3-10 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，所以从理论上讲，链式栈没有栈满问题，但是它有栈空问题。

程序 3-10 链式栈主要操作的实现

```
void InitStack(LinkStack& S){
//链式栈初始化：置栈顶指针，即链表头指针为空
    S=new LinkNode;                                    //创建头结点
    if(!S){cerr<<"存储分配失败!\n"; exit(1);}          //链接 stdlib.h 和 iostream.h
    S->link=NULL;                                      //栈顶置空
};
void Push(LinkStack& S,SElemType x){
//进栈：将元素 x 插入到链式栈的栈顶，即链表头
    LinkNode * p=new LinkNode;                         //创建新结点
    if(!p){cerr<<"存储分配失败!\n"; exit(1);}
    p->data=x;
    p->link=S->link; S->link=p;                        //新结点插入在链表头
};
int Pop(LinkStack& S,SElemType& x){
//退栈：若栈不空，删除栈顶结点并通过引用型参数 x 返回被删栈顶元素的值
//若删除成功，函数返回 1，否则函数返回 0，此时 x 的值不可引用
    if(S->link==NULL)return 0;                         //若栈空则不退栈，返回 0
    LinkNode * p=S->link;                              //否则暂存栈顶元素
    S->link=p->link;                                   //栈顶指针退到新的栈顶位置
    x=p->data; delete p;                               //释放结点，返回退出元素的值
    return 1;
};
int GetTop(LinkStack& S,SElemType& x){
//读取栈顶：若栈不空，通过引用型参数 x 返回栈顶元素的值，且函数返回 1
//若栈空，则函数返回 0，此时 x 的值不可引用
    if(S->link==NULL)return 0;                         //若栈空则返回 0
    x=S->link->data;                                   //栈不空则返回栈顶元素的值
    return 1;
};
int StackEmpty(LinkStack& S){
//判断栈是否为空：若栈空，则函数返回 1，否则函数返回 0
    return S->link==NULL;                              //返回 S->link==NULL 结果
};
int StackSize(LinkStack& S){
//求栈的长度：计算栈元素个数
    LinkNode * p=S->link; int k=0;
    while(p!=NULL){p=p->link; k++;}                    //逐个结点计数
    return k;
};
```

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 是链式栈中的元素个数。

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	$O(1)$	$O(1)$			

除栈清空操作和计算栈长度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。

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：

```
LinkNode * s=new LinkNode[n];
```

在多个链式栈的情形中，link 域需要一些附加的空间，但其代价并不很大。

想想看　一种常用的链式栈是基于静态链表的。用一个整数数组 $S[n]$ 存放链接指针（游标），设初始时 top＝－1，表示栈空，则其进栈、出栈、判栈空等操作如何实现？

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，…，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？下面对此进行讨论。

当进栈序列为 1，2 时，可能的退栈序列有 2 种：{1，2}和{2，1}；当进栈序列为 1，2，3 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，…，n，可能的退栈序列有 m_n 种，则

当 n＝0 时，m_0＝1：退栈序列为{}。

当 n＝1 时，m_1＝1：退栈序列为{1}。

当 n＝2 时，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_0m_1＝1 种退栈序列：{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_1m_0＝1 种退栈序列：{2，1}。总的可能退栈序列有 $m_0m_1+m_1m_0$＝2 种。

当 n＝3 时，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_0m_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 左侧有 1 个数，右侧有 1 个数，有 m_1m_1＝1 种退栈序列：{2，1，3}；退栈序列中 1 在第 3 位，1 左侧有 2 个数，右侧有 0 个数，有 m_2m_0＝2 种退栈序列：{2，3，1}和{3，2，1}。总的可能退栈序列有 $m_0m_2+m_1m_1+m_2m_0$＝5 种。

当 n＝4 时，m_4＝14：以此类推，在退栈序列中分别置 1 在第 1 位、第 2 位、第 3 位和第

4 位，得到总的可能退栈序列种数有 $m_0m_3+m_1m_2+m_2m_1+m_3m_0=1\times5+1\times2+2\times1+5\times1=14$ 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，则可能的退栈序列种数为：

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

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 $p_1p_2\cdots p_i\cdots p_n$，如果序号 $i<j<k$，且在进栈序列中 $p_i<p_j<p_k$，即：

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

则…p_k…p_i…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。

【例 3-5】 已知一个进栈序列为 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，原因参见表 3-3。

表 3-3　例 3-5 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	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
cdab	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
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	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
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。允许插入的一端叫做队尾(rear)，允许删除的一端叫做队头(front)，如图 3-7 所示。每