第3章　栈 和 队 列

栈和队列在计算机系统中应用极多。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。

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

栈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，而不允许插入和删除的另一端叫做栈底。当栈中没有任何元素时则称为空栈。

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，栈又叫做后进先出（Last In First Out，LIFO）的线性表。参看图 3-1。

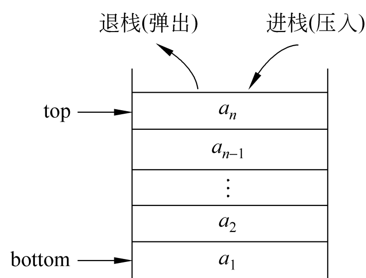


图 3-1　栈的示意图

栈的主要操作如下：

（1）void 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）int 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 中元素个数。

【例 3-1】 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 $\boldsymbol{A}$ 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]);
     i = 0;
     while ( !StackEmpty(S)) Pop (S, A[i++]);
};
```

思考题 *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。

在顺序栈中需要设置一个向量 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。顺序栈的结构示意如图 3-2 所示。

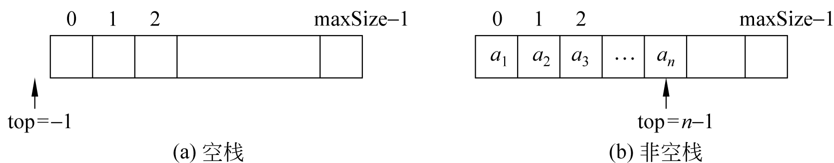


图 3-2 顺序栈的示意图

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（存放于头文件 SeqStack.h 中）

```
#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在程序编译时就分配了存储空间，栈空间一旦装满就不能扩充。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（保存于头文件 SeqStack.h 中）

```
#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, top;                 //栈空间最大容量及栈顶指针（下标）
} SeqStack;
```

下面讨论栈的主要操作的实现。

1．初始化操作

程序 3-4 给出动态顺序栈的初始化算法。算法的基本思路是：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。

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，若分配不成功则进行错误处理
    S.elem = ( SElemType* ) malloc ( initSize*sizeof ( SElemType ));
                                     //创建栈空间
    if ( S.elem == NULL ) { printf ( "存储分配失败！\n" );  exit(1); }
    S.maxSize = initSize;  S.top = -1;
}
```

2．进栈操作

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，函数报错并返回 0。

（2）若栈不满，先让栈顶指针进 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，最后函数返回 1。

【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元素位置。

程序 3-5 动态顺序栈进栈操作的实现

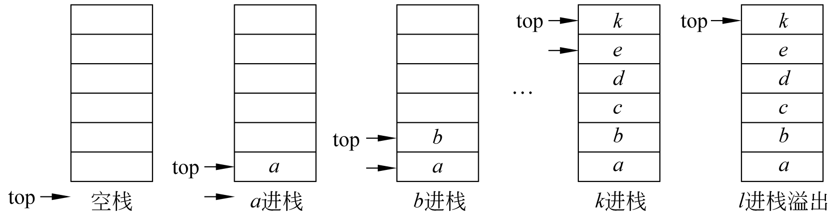


图 3-3　顺序栈的进栈操作示意图

```
int Push( SeqStack& S, DataType x ) {
//进栈操作：若栈不满，则将元素 x 插入栈顶，函数返回 1；否则栈溢出，函数返回 0
     if ( S.top == S.maxSize-1 ) return 0;       //栈满则溢出处理
     S.elem[++S.top] = x;                        //栈顶指针先加 1，再进栈
     return 1;
}
```

思考题　可否使用 void calloc(n, size)函数动态分配存储空间？如何做？

3．退栈操作

程序 3-6 是动态顺序栈的退栈算法。算法的基本思路是：先判断是否栈空。若在退栈时发现栈是空的，则执行退栈操作失败，函数返回 0；若栈不空，可先将栈顶元素取出，再让栈顶指针减 1，让栈顶退回到次栈顶位置，函数返回 1，表示退栈成功。

【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。

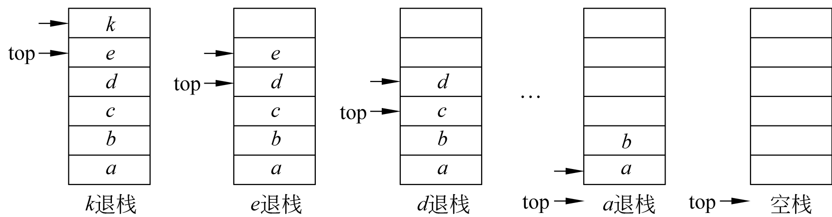


图 3-4　顺序栈的退栈操作示意图

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参数 x 返回栈顶元素值，栈顶指针退 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--];                 //栈顶指针退 1
    return 1;                            //退栈成功，函数返回 1
};
```

思考题　有一种说法：退栈时若栈空不是错误，而是表明使用栈的某种处理的结束，执行栈空处理即可。进栈时若栈满则是错误，表明栈溢出，需做溢出处理。这种说法对吗？

4．顺序栈其他操作的实现

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。

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 S.top == -1 的结果
};
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 S.top == S.maxSize 的结果
};
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;
};
```

思考题 *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？*

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	$O(1)$	$O(1)$	判栈空 StackEmpty	$O(1)$	$O(1)$
进栈 Push	$O(1)$	$O(1)$	判栈满 StackFull	$O(1)$	$O(1)$
退栈 Pop	$O(1)$	$O(1)$	求栈长 StackSize	$O(1)$	$O(1)$
读栈顶 GetTop	$O(1)$	$O(1)$			

从表 3-1 可知，顺序栈所有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。

3.1.3 扩展阅读：多栈处理

1．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，可定义一个足够大的栈空间 $V[m]$，并将两个栈设为 0 号栈和 1 号栈。在 $V[m]$两端的外侧分设两个栈的栈底，用 $b[0]$（= −1）和 $b[1]$（= m）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。

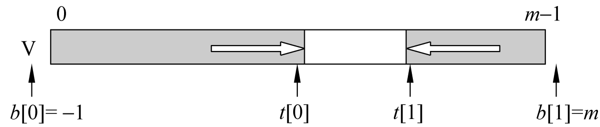


图 3-5　两个栈共享同一存储空间的情形

- 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是 *t*[1]减 1。当两个栈的栈顶指针相遇，即 *t*[0]+1 == *t*[1]时，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主要操作的实现（保存于头文件 DblStack.h 中）

```
#define m 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 V[m];                                   //栈数组
} DblStack;
void InitStack ( DblStack& S ) {
//初始化函数：建立一个大小为 m 的空栈
    S.top[0] = S.bot[0] = -1;  S.top[1] = S.bot[1] = m;
}
int Push ( DblStack& S, SElemType x, int i ) {  //进栈运算
     if (S.top[0]+1 == S.top[1] ) return 0;     //栈满则返回 0
     if ( i == 0 ) S.V[++S.top[0]] = x;         //栈 0：栈顶指针先加 1 再进栈
     else S.V[--S.top[1]] = x;                  //栈 1：栈顶指针先减 1 再进栈
     return 1;
}
int Pop ( DblStack& S, int i, SElemType& x ) {
//函数通过引用参数 x 返回退出栈 i 栈顶元素的元素值，前提是栈不为空
    if ( S.top[i] == S.bot[i] ) return 0;       //第 i 个栈栈空，不能退栈
    if ( i == 0 ) x = S.V[S.top[0]--];          //栈 0：先出栈，栈顶指针减 1
    else x = S.V[S.top[1]++];                   //栈 1：先出栈，栈顶指针加 1
    return 1;
}
```

2．多栈共享同一栈空间

当程序中同时使用 3 个或更多的栈时，如果它们共享同一个存储空间 *V*[*m*]，必须使用所谓的“栈浮动技术”来处理栈的进栈、退栈和溢出处理。

如果 *n* 个栈共同使用同一块存储空间 *V*[*m*]，可定义各栈的栈底指针 bot[*n*]和栈顶指针 top[*n*]，其中，bot[*i*]表示第 *i* 个栈的栈底，top[*i*]表示第 *i* 个栈的栈顶，如图 3-6 所示。

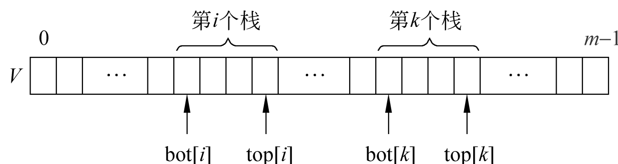
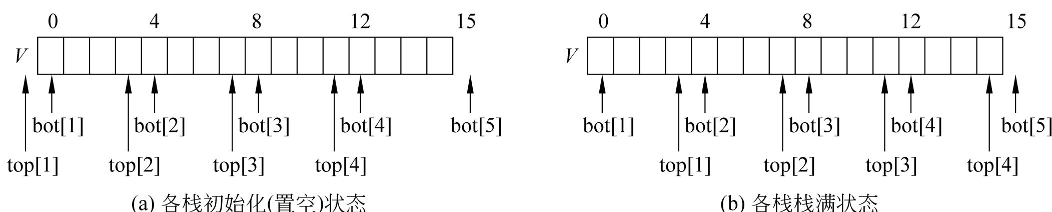


图 3-6　*n* 个顺序栈共享同一个存储空间的示意图

假设 *m* 可以整除 *n*，各栈的编号从 1 开始，*V*[*m*]平均分配给 *n* 个栈。各栈的初始分配将是 bot[*i*] = (*i*−1)×*m*/*n*，top[*i*] = bot[*i*]−1（1≤*i*≤*n*），bot[*n*+1] = *m*。

【例 3-5】 若 *m* = 16, *n* = 4，则各栈的初始分配是 bot[*i*] = 0, 4, 8, 12, 16，top[*i*] = −1, 3, 7, 11。注意，栈底指针比栈顶指针多一个。第 *i* 个栈的栈空条件是 bot[*i*]−1 == top[*i*]，栈满条件是 top[*i*]+1 == bot[*i*+1]，如图 3-7 所示。



(a) 各栈初始化(置空)状态　　　　(b) 各栈栈满状态

图 3-7　4 个栈共享同一个存储空间的示意图

- 第 *i* 个栈进栈的情形：若 top[*i*]+1==bot[*i*+1]，则栈满，发生“上溢”，需要做栈浮动，为第 *i* 个栈腾出进栈的空间才能继续进栈；否则置 V[++top[*i*]] = *x*，进栈成功。
- 第 *i* 个栈退栈的情形：若 top[*i*] == bot[*i*]−1，则第 *i* 个栈空，不能退栈；否则执行退栈操作，置 *x* = D[top[*i*]−−]，从 *x* 取得退出的元素。

一种较直观的栈浮动的方法是：当第 *i* 个栈发生“上溢”时，先从其右边第 *i*+1 个栈起往右边扫描，寻找未满的栈，如果找到，则移动第 *i* 栈右边相关栈的存储，为第 *i* 个栈腾出空间；否则向第 *i* 个栈左边的各栈扫描，寻找未满的栈；如果左边也找不到未满的栈，则说明整个存储区均已占满，报告失败信息。这一算法详述如下：

（1）向右寻找满足 *i*＜*k*≤*n* 且 top[*k*]+1＜bot[*k*+1]的最小的 *k*。如果存在这样的 *k*，则执行以下操作：

① 对于所有的存储单元 *j*（*j* = top[*k*], top[*k*]−1, …, bot[*i*+1]），执行 *V*[*j*+1] = *V*[*j*]，为第 *i* 个栈腾出一个栈顶存储单元，再执行 *V*[++top[*i*]] = *x*。

② 对于所有相关的栈 *j*（*i*＜*j*≤*k* ），修改栈顶与栈底，置 bot[*j*]++，top[*j*]++。

（2）如果找不到步骤（1）中的 *k*，但在 top[*i*]的左边找到了满足 1≤*k*＜*i* 且 top[*k*]+1＜bot[*k*+1]的最大的 *k*，则执行以下操作：

① 对于所有的存储单元 *j*（ *j* = bot[*k*+1], …, top[*i*]），执行 *V*[*j*−1] = *V*[*j*]，为第 *i* 个栈腾出一个栈顶存储单元，再执行 *V*[top[*i*]++] = *x*。

② 对于所有相关的栈 *j*（*k*＜*j*≤*i*），修改栈顶和栈底，置 bot[*j*]−−，top[*j*]−−。

（3）对于所有的 *k*≠*i*，如果均有 top[*k*]+1 == bot[*k*+1]，则表示存储空间已满，*n* 个栈都不能进栈了。

多个顺序栈共享同一个栈空间，使用栈浮动技术处理溢出，会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。

3.1.4 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 3-8。

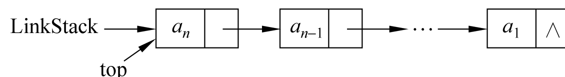


图 3-8 链式栈

从图 3-8 可知，链式栈无须附加头结点，栈顶指针在链表的首元结点。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。

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（保存于头文件 LinkStack.h 中）

```
typedef int SElemType;
typedef struct node {
     SElemType data;
     struct node *link;
} LinkNode, *LinkStack;
```

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：只要还有可分配的存储空间，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 3-10 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 = NULL;                                          //栈顶置空
};
int 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->data = x;  p->link = S;  S = p;                 //新结点插入在链头
    return 1;
}
int Pop ( LinkStack& S, SElemType& x ) {
//退栈：若栈空，函数返回 0，参数 x 的值不可用
//若栈不空，则函数返回 1，并通过引用参数 x 返回被删栈顶元素的值
    if ( S == NULL ) return 0;                         //栈空，函数返回 0
    LinkNode *p = S;  x = p->data;                     //存栈顶元素
    S = p->link;  free (p);  return 1;                 //栈顶指针退到新栈顶位置
```

```
}
int 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 == NULL ) return 0;                         //栈空，函数返回 0
    x = S->data;  return 1;                            //栈不空则返回 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 == NULL;
}
int StackSize ( LinkStack& S ) {
//求栈的长度：计算栈元素个数
    LinkNode *p = S;  int k = 0;
    while ( p != NULL ) { p = p->link;  k++; }         //逐个结点计数
    return k;
}
```

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。

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	$O(1)$	$O(1)$
进栈 Push	$O(1)$	$O(1)$	判栈空 StackEmpty	$O(1)$	$O(1)$
退栈 Pop	$O(1)$	$O(1)$	计算栈长 StackSize	$O(n)$	$O(1)$

除计算栈长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。

如果同时使用 n 个链式栈，其头指针数组可以用以下方式定义：

```
LinkStack *s = ( LinkStack* ) malloc ( n*sizeof ( LinkStack ));
```

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。

3.1.5　扩展阅读：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，这就叫做栈的混洗。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？现在来做一讨论。

当进栈序列为 1, 2 时，可能的退栈序列有两种：{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：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 $m_2 = m_0m_1+m_1m_0 = 2$ 种。
- 当 n = 3 时：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 $m_3 = m_0m_2+m_1m_1+m_2m_0 = 5$ 种。
- 当 $n = 4$ 时：在退栈序列中置 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数有 $m_4 = m_0m_3+m_1m_2+m_2m_1+m_3m_0 = 1×5+1×2+2×1+5×1 = 14$ 种。

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, 2, …, n 位，则可能的退栈序列数为

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

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, …, n，利用栈得到可能的退栈序列为 $p_1 p_2 \cdots p_i \cdots p_n$，如果进栈时按 p_i、p_j、p_k 次序，即…, 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，按照栈的后进先出的特性，p_i 压在 p_j 的下面，当 p_k 最先退栈时，p_i 不可能先于 p_j 退栈，所以…, p_k, …, p_i, …, p_j 是不可能的退栈序列。

【例 3-6】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	*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

队列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，允许删除的一端叫做队头，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 $a_1, a_2, \cdots, a_n$。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叫做先进先出（First In First Out，FIFO）。

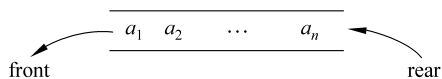


图 3-9　队列的示意图

队列的主要操作如下:

（1）队列的初始化：void InitQueue (Queue& Q)。

先决条件：无。

操作结果：队列 Q 置空，并对各数据成员赋初值。

（2）进队列：int 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）出队列：int DeQueue (Queue& Q, QElemType& x);

先决条件：队列 Q 已存在且非空。

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）读取队头：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）判队列空否：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）判队列满否：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）求队列长度：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。

3.2.2 循环队列

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，它利用一个一维数组 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。

【例 3-7】 顺序队列的初始化、插入和删除如图 3-10 所示。maxSize 是数组的最大长度。

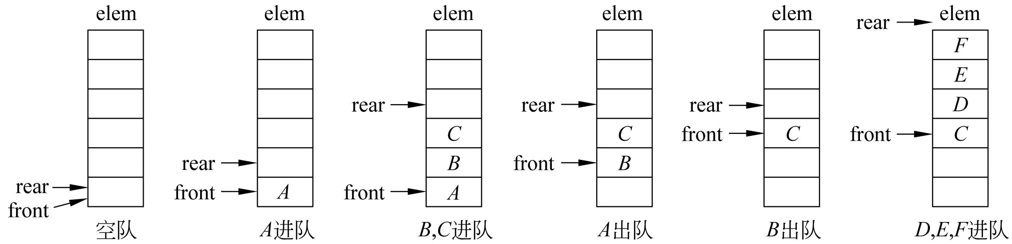


图 3-10　顺序队列的初始化、插入和删除的示意

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。所以，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。

从图 3-10 中还可以看到，当队列指针 front == rear 时，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，形成一个环形的表，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。

【例 3-8】 循环队列的初始化、插入和删除如图 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。

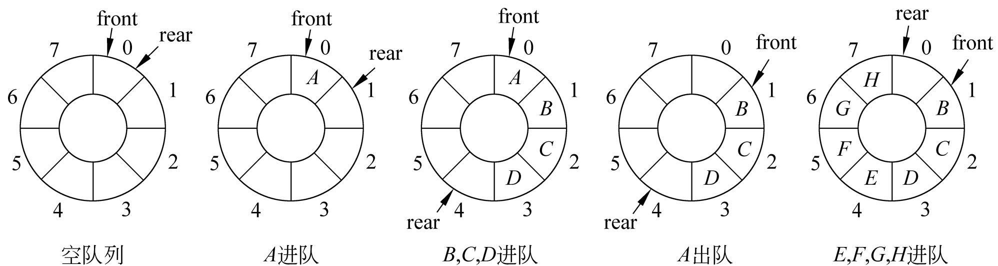


图 3-11　循环队列的初始化、插入和删除的示意

循环队列的队头指针 front 和队尾指针 rear 初始化时都置为 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。

思考题 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，例如：

（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（作为队列的控制变量），也可实现循环队列。

思考题 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？

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 3-11 所示的循环队列的结构。

程序 3-11 循环队列的结构定义（保存于头文件 CircQueue.h 中）

```
typedef int QElemType;
#define maxSize 20                          //循环队列的容量
typedef struct {
     QElemType elem[maxSize];               //元素数组
     int front, rear;                       //队头指针和队尾指针（数组下标）
} CircQueue;
```

在实际应用时循环队列多采用静态存储分配。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元素为止。

循环队列主要操作的实现如程序 3-12 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 3-12 循环队列主要操作的实现

```
void InitQueue ( CircQueue& Q ) {
//循环队列初始化：令队头指针和队尾指针归零
     Q.front = Q.rear = 0;
};
int EnQueue ( CircQueue& Q, QElemType x ) {
//若队列不满，则将元素 x 插入到队尾，函数返回 1，否则函数返回 0，不能进队列
    if ((Q.rear+1) % maxSize == Q.front) return 0; //队列满则不能插入,
                                                   //函数返回 0
    Q.elem[Q.rear] = x;                            //按照队尾指针指示位置插入
    Q.rear = (Q.rear+1) % maxSize;                 //队尾指针进 1
```

```
        return 1;                              //插入成功，函数返回1
    };
    int DeQueue ( CircQueue& Q, QElemType& x ) {
    //若队列不空，则函数退掉一个队头元素并通过引用型参数x返回，函数返回1，否则函数
    //返回0，此时x的值不可引用
        if ( Q.front == Q.rear ) return 0;     //队列空则不能删除，函数返回0
        x = Q.elem[Q.front];
        Q.front = (Q.front+1) % maxSize;       //队头指针进1
        return 1;                              //删除成功，函数返回1
    };
    int GetFront( CircQueue& Q, QElemType& x ) {
    //若队列不空，则函数通过引用参数x返回队头元素的值，函数返回1，否则函数返回
    //0，此时x的值不可引用
        if ( Q.front == Q.rear ) return 0;     //队列空则函数返回0
        x = Q.elem[Q.front];                   //返回队头元素的值
        return 1;
    };
    int QueueEmpty ( CircQueue& Q ) {
    //判队列空否。若队列空，则函数返回1；否则返回0
        return Q.front == Q.rear;              //返回front==rear的运算结果
    };
    int QueueFull ( CircQueue& Q ) {
    //判队列满否。若队列满，则函数返回1；否则返回0
        return (Q.rear+1) % maxSize == Q.front; //返回布尔式的运算结果
    };
    int QueueSize ( CircQueue& Q ) {
    //求队列元素个数
        return (Q.rear-Q.front+maxSize) % maxSize;
    };
```

思考题 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？

循环队列主要操作的性能如表3-4所示。

表3-4 循环队列各个操作的性能比较

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

循环队列所有操作的时间复杂度和空间复杂度都为$O(1)$，其性能十分良好。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。

思考题 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 链式队列

链式队列是队列的基于单链表的存储表示，如图 3-12 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。

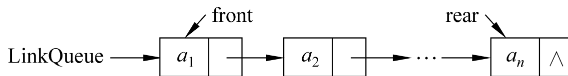


图 3-12 链式队列

在不设头结点的场合，队列的队头指针指向单链表的首元结点，若要从队列中退出一个元素，必须从单链表中删去首元结点。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，即单链表的尾结点后面，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，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。

思考题 在链式队列中设置头结点与不设置头结点有什么不同？

链式队列每个结点的定义与单链表结点相同，队列设置了两个指针：队头指针 front 指向首元结点，队尾指针 rear 指向链尾结点。链表中所有结点都必须通过这两个指针才能找到。队列的结构定义如程序 3-13 所示。

程序 3-13 链式队列的结构定义（保存于头文件 LinkQueue.h 中）

```
typedef int QElemType;
typedef struct Node {                              //链式队列结点
     QElemType data;                               //结点的数据
     struct Node *link;                            //结点的链接指针
} LinkNode;
typedef struct {                                   //链式队列
     LinkNode *front, *rear;                       //队列的队头和队尾指针
} LinkQueue;
```

链式队列主要操作的实现如程序 3-14 所示，与单链表不同的是，插入和删除仅限于队列的两端：插入（进队列）在队尾，新结点成为新的队尾；删除（出队列）在队头，队列的第二个结点成为新的首元结点。

程序 3-14 链式队列主要操作的实现

```
void InitQueue ( LinkQueue& Q ) {
//队列初始化：令队头指针和队尾指针均为 NULL
    Q.front = Q.rear = NULL;                       //队头与队尾指针初始化
};
int EnQueue( LinkQueue& Q, DataType x ) {
//进队列：将新元素 x 插入到队列的队尾（链尾）
    LinkNode *s = ( LinkNode* ) malloc (sizeof (LinkNode));  //创建新结点
    s->data = x;  s->link = NULL;
```

```
    if ( Q.rear == NULL ) Q.front = Q.rear = s;//空队列时新结点是唯一结点
    else { Q.rear->link = s;  Q.rear = s; }      //新结点成为新的队尾
    return 1;
}
int DeQueue ( LinkQueue& Q, QElemType& x ) {
//出队列：如果队列空，不能退队，函数返回 0，且引用参数 x 的值不可用
//如果队列不空，函数返回 1，且从引用参数 x 得到被删元素的值
    if ( Q.front == NULL ) return 0;             //队列空，不能退队，返回 0
    LinkNode *p = Q.front;  x = p->data;         //存队头元素的值
    Q.front = p->link;  free (p);                //队头修改，释放原队头结点
    if ( Q.front == NULL ) Q.rear = NULL;
    return 1;
}
int GetFront ( LinkQueue& Q, QElemType& x ) {
//读取队头元素的值：若队列空，则函数返回 0，且引用参数 x 的值不可用
//若队列不空，则函数返回 1，且从引用参数 x 可得到退出的队头元素的值
    if ( Q.front == NULL ) return 0;             //队列空，不能读，返回 0
    x = Q.front->data;   return 1;               //取队头元素中的数据值
}
int QueueEmpty ( LinkQueue& Q ) {
//判队列空否：队列空则函数返回 1，否则函数返回 0
     return Q.front == NULL;                     //返回布尔式的计算结果
};
int QueueSize ( LinkQueue& Q ) {
//求队列元素个数
     LinkNode *p = Q.front;  int k = 0;
     while ( p != NULL ) { p = p->link;  k++; }
     return k;
}
```

链式队列主要操作的性能如表 3-5 所示。

表 3-5　链式队列各操作的性能比较

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 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	$O(\mathrm{n})$	$O(1)$
读取队头 GetFront	$O(1)$	$O(1)$			

除求队列长度的操作外，其他链式队列的操作的时间和空间复杂度都很好。求队列长度的操作需要对链式队列所有结点逐个检测，若设链式队列中有 n 个结点，其时间复杂度达到 $O(n)$。

思考题　为何链式队列不采用循环单链表作为其存储表示？

3.3 栈的应用

栈在计算机科学与技术领域应用十分广泛，本节所涉及的仅是其中的一小部分。希望通过几个小的实例，让读者学会如何灵活使用栈来解决问题。

3.3.1 数制转换

在计算机基础课程中已经讲过如何利用“除 2 取余”法把一个十进制整数转换为二进制数。例如，一个十进制整数 111 转换为二进制数 1101111 的计算过程如图 3-13 所示。若想把一个十进制整数转换为八进制数也可以使用类似的方法。例如，一个十进制整数 1347 转换为八进制数 2503 的计算过程如图 3-14 所示。

整数 N	111	55	27	13	6	3	1
商（$N\div 2$）	55	27	13	6	3	1	0
余数（$N\%2$）	1	1	1	1	0	1	1

图 3-13　十进制数转换为二进制数的过程

整数 N	1347	168	21	2
商（$N\div 8$）	168	21	2	0
余数（$N\%8$）	3	0	5	2

图 3-14　十进制数转换为八进制数的过程

一般地，可把此方法推广到把十进制整数转换为 k 进制数。

【例 3-9】 可以利用栈解决数制转换问题。例如，$49_{10} = 1\times2^5+1\times2^4+1\times2^0 = 110001_2$。其转换规则如下：

$$N=\sum_{i=0}^{\lfloor \log_k N\rfloor} b_i\times k^i,\quad b_i=0,1,\cdots,k-1$$

其中，b_i 表示 k 进制数的第 i 位上的数字。

这样，十进制数 N 可以用长度为 $\lfloor\log_k N\rfloor+1$ 位的 k 进制数表示为 $b_{\lfloor\log_k N\rfloor}\cdots\ b_2\,b_1\,b_0$。若令 $j=\lfloor\log_k N\rfloor$，则有

$$N=b_j k^j+b_{j-1}k^{j-1}+\cdots+b_1k^1+b_0=(b_jk^{j-1}+b_{j-1}k^{j-2}+\cdots+b_1)\times k+b_0$$

$$b_jk^{j-1}+b_{j-1}k^{j-2}+\cdots+b_1=(b_jk^{j-2}+b_{j-1}k^{j-3}+\cdots+b_2)\times k$$

因此，可以先通过 $N\%k$ 求出 b_0，然后令 $N=N/k$，再对新的 N 做除 k 求模运算可求出 b_1……如此重复，直到算出的 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。算法如程序 3-15 所示。

程序 3-15　使用栈实现数制转换的算法

```
typedef int SElemType;
int BaseTrans ( int N ) {
     int i, result = 0;
     SeqStack S;  InitStack ( S );
     while ( N != 0 )
          { i = N % k;  N = N / k;  Push ( S, i ); }
     while ( ! StackEmpty ( S ) )
          { Pop ( S, i );  result = result*10+i; }
     return result;
}
```

在程序中，进栈和退栈都是“线性”的，这种“流水式”的处理体现了栈和队列的优越性。与一维数组中使用下标跳来跳去进行处理相比，使用栈和队列辅助算法的实现不仅思路清晰，而且正确性更好。

3.3.2　括号匹配

【例 3-10】 在一个用字符串描述的表达式“(a*(b+c)−d)”中，位置 0 和位置 3 有左括号“(”，位置 7 和位置 10 有右括号“)”。位置 0 的左括号匹配位置 10 的右括号，位置 3 的左括号匹配位置 7 的右括号。而对于字符串“(a+b))(”，位置 5 的右括号没有可匹配的左括号，位置 6 的左括号没有可匹配的右括号。

现在要建立一个算法，输入一个字符串，输出匹配的括号和没有匹配的括号。

可以观察到，如果从左向右扫描一个字符串，那么每一个右括号将与最近出现的那个未匹配的左括号相匹配。这个观察的结果使我们联想到可以在从左向右的扫描过程中把所遇到的左括号存放到栈中。每当在后续的扫描过程中遇到一个右括号时，就将它与栈顶的左括号（如果存在）相匹配，并删除栈顶的左括号。图 3-15 是利用栈对字符串“(a*(b+c)−d)”进行括号匹配的过程，图 3-16 是利用栈对字符串“(a+b))(”进行括号匹配的过程。

顺序	栈内容	当前字符	动作	剩余字符串	顺序	栈内容	当前字符	动作	剩余字符串
0	空	'('	进栈	"(a*(b+c)−d)"	6	'(', '('	'c'	跳过	"c)−d)"
1	'('	'a'	跳过	"a*(b+c)−d)"	7	'(', '('	')'	退栈	")−d)"
2	'('	'*'	跳过	"*(b+c)−d)"	8	'('	'−'	跳过	"−d)"
3	'('	'('	进栈	"(b+c)−d)"	9	'('	'd'	跳过	"d)"
4	'(', '('	'b'	跳过	"b+c)−d)"	10	'('	')'	退栈	")"
5	'(', '('	'+'	跳过	"+c)−d)"	11	空	无	结束	""

图 3-15　无错误的括号匹配过程

顺序	栈内容	当前字符	动作	剩余字符串	顺序	栈内容	当前字符	动作	剩余字符串
0	空	'('	进栈	"(a+b))("	4	'('	')'	退栈	"))("
1	'('	'a'	跳过	"a+b))("	5	空	')'	报错	")("
2	'('	'+'	跳过	"+b))("	6	空	'('	进栈	"("
3	'('	'b'	跳过	"b))("	7	'('	无	报错	""

图 3-16　有错误的括号匹配过程

程序 3-16 给出相应的算法。因为括号数目不定，采用了链式栈。其时间复杂度为 $O(n)$，其中 n 是输入串的长度。

程序 3-16　判断括号匹配的算法

```
#include <string.h>
#include <stdio.h>
#define stkSize 10
void PrintMatchedPairs ( char expr[] ) {
//参数 expr 应是已存在的表达式字符串，算法给出括号匹配的过程
```

```
    int S[stkSize];  int top = -1;              //设置栈 S 并置空
    int j, i = 0;  char ch = expr[i];
    while ( ch != '\0' ) {                      //在表达式中搜索'('和')'
        if ( ch == '(' ) S[++top] = i;          //左括号，位置进栈
        else if ( ch == ')' ) {                 //右括号
            if ( top != -1 ) {                  //如果栈不空，有括号匹配
                j = S[top--];                   //退栈
                printf ( "位置%d 的左括号与位置%d 的右括号匹配！\n", j, i );
            }
            else printf ( "栈空，没有与位置%d 的右括号匹配的左括号！\n", i );
        }
        ch = expr[++i];                         //跳过，取下一字符
    }
    while ( top != -1 ) {                       //串已处理完，但栈中还有左括号
        j = S[top--];                           //报错次数等于栈中左括号数目
        printf ( "没有与位置%d 的左括号相匹配的右括号！\n", j );
    }
}
```

算法扫描表达式，只要遇到左括号“(”立即将它进栈。如果遇到右括号“)”，就要看栈顶，如果栈不空，括号可配对；将栈顶退掉；如果栈空，表示右括号无左括号与之配对，报错。如果表达式扫描完，栈不空，表示栈内还有左括号，但再无右括号与之配对，报错。此程序修改一下，使用 3 个栈，就可以同时解决在文本中的“{”与“}”、“[”与“]”、“(”与“)”的匹配问题。

3.3.3 表达式的计算与优先级处理

在计算机中执行算术表达式的计算是通过栈来实现的。

如何将表达式翻译成能够正确求值的指令序列，是语言处理程序要解决的基本问题。作为栈的应用事例，下面讨论表达式的求值过程。

任何一个表达式都是由操作数、操作符和分界符组成。通常，算术表达式有 3 种表示：

（1）中缀表达式：<操作数> <操作符> <操作数>，例如 $A+B$。

（2）前缀表达式：<操作符> <操作数> <操作数>，例如$+AB$。

（3）后缀表达式：<操作数> <操作数> <操作符>，例如 $AB+$。

后缀表达式也叫做 RPN 或逆波兰记号。我们日常生活中所使用的表达式都是中缀表达式。如 $A+B*(C-D)-E/F$ 就是中缀表达式。

为了正确执行这种中缀表达式的计算，必须明确各个操作符的执行顺序。为此，C 语言为每一个操作符都规定了一个优先级。为简单起见，本节只讨论算术运算中的双目操作符。C 语言规定一个表达式中相邻的两个操作符的计算次序为：优先级高的先计算；如果优先级相同，则自左向右计算；当使用括号时，从最内层的括号开始计算。

对于编译程序来说，一般使用后缀表达式对表达式求值。因为用后缀表达式计算表达式的值，在计算过程中不需考虑操作符的优先级和括号，只需顺序处理表达式的操作符即可。

【例 3-11】 给出一个中缀表达式 $A+B*(C-D)-E/F$，求值的执行顺序如图 3-17 所示，R_1、R_2、R_3、R_4、R_5 为中间计算结果。与图 3-17 所示的表达式计算等价的后缀表达式计算过程如图 3-18 所示。与中缀表达式 $A+B*(C-D)-E/F$ 对应的后缀表达式为 $ABCD-*+EF/-$。

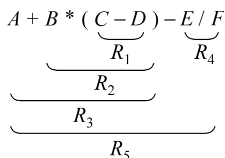


图 3-17　中缀表达式计算顺序

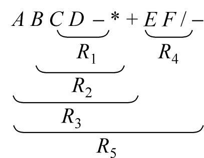


图 3-18　后缀表达式的计算顺序

通过后缀表示计算表达式值的过程为：顺序扫描表达式的每一项，然后根据它的类型做如下相应操作：如果该项是操作数，则进栈；如果是操作符<op>，则连续从栈中退出两个操作数 Y 和 X，形成运算指令 X <op> Y，并将计算结果重新进栈。当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。

【例 3-12】 对于后缀表达式 $ABCD-*+EF/-$的求值过程如图 3-19 所示。

步	扫描项	项类型	动　　作	栈中内容
1			置空栈	空
2	A	操作数	进栈	A
3	B	操作数	进栈	$A\ B$
4	C	操作数	进栈	$A\ B\ C$
5	D	操作数	进栈	$A\ B\ C\ D$
6	–	操作符	D、C 退栈，计算 $C–D$，结果 R_1 进栈	$A\ B\ R_1$
7	*	操作符	R_1、B 退栈，计算 $B*R_1$，结果 R_2 进栈	$A\ R_2$
8	+	操作符	R_2、A 退栈，计算 $A+R_2$，结果 R_3 进栈	R_3
9	E	操作数	进栈	$R_3\ E$
10	F	操作数	进栈	$R_3\ E\ F$
11	/	操作符	F、E 退栈，计算 E/F，结果 R_4 进栈	$R_3\ R_4$
12	–	操作符	R_4、R_3 退栈，计算 $R_3–R_4$，结果 R_5 进栈	R_5

图 3-19　使用操作符栈的后缀表达式的求值过程

程序 3-17 给出简单计算器的模拟。它要求从键盘读入一个字符串后缀表达式，计算表达式的值。该计算器接收的操作符包括 '+'、'-'、'*'、'/'，操作数在'0'～'9'之间。

程序 3-17　计算后缀表达式的值

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define stkSize 20                          //预设操作数栈的大小
int DoOperator ( int OPND[], int& top, char op ) {
//算法：从操作数栈 OPND 中取两个操作数，根据操作符 op 形成运算指令并计算
     int left, right;
     if ( top == -1 )                                     //检查栈空否？
```