第3章　栈和队列

栈和队列都是特殊形式的线性表，由于它们的应用十分广泛，人们早已把它们单列为新的数据结构。栈和队列在数据的插入和删除等操作上有所不同，把它们放在一起对比学习，有助于学生掌握这两种数据结构的不同特点，从而更好地在实践中加以应用。

本章将介绍栈和队列的基本概念、运算以及常见的实现方法，并通过一个有趣的案例介绍栈和队列的应用。

【技能目标】

- 理解栈和队列的定义，掌握栈和队列的特征和基本运算。
- 会使用栈和队列的顺序存储结构解决问题。
- 能实现栈和队列的各种基本运算。
- 会使用栈和队列的链式存储结构解决问题。
- 能实现链表的各种基本运算。

3.1　栈的定义和基本运算

让我们观察一下餐饮店里盘子的堆放和取用操作，可以发现以下一些特点：盘子一个个地叠放成一摞，可以看成是一个由盘子组成的线性表。每次将洗净的盘子放入盘叠，总是放在最顶部，而每次用盘子时，也总是先取用盘叠最上方的那个盘子。

当我们交考试试卷时，也可以发现这种情况的存在：第一个交卷的同学卷子放在最底下，而最后一个交卷的同学卷子放在最上面。当老师改卷时，总是先从最上面的试卷开始批阅。

从上述两个例子可以看出，在对事物的组织和管理上，采用的是同一机制，即使用一个线性表，且仅在表的一端允许插入和删除，这就是栈的概念。

3.1.1　栈的定义

栈是一种特殊的线性表，它仅允许在表的一端进行运算。在表中，允许插入和删除的一端称为“栈顶”，另一端称为“栈底”，将元素插入栈顶的操作成为“进栈”，称删除栈顶元素的操作为“出栈”，如图3.1所示。因为出栈操作时后进栈的元素先出，所以栈也被称为是一种“后进先出”表，简称为LIFO(Last In First Out)。

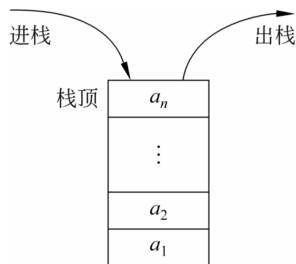

图3.1　堆栈

3.1.2　栈的基本运算

根据实际应用，通常认为，栈应该包含了以下一些基本运算。

(1) 栈初始化——置栈为空栈。

(2) 判断栈是否为空——若栈为空，则返回 true，否则返回 false。

(3) 求栈的长度——返回栈的元素个数。

(4) 进栈——将一个元素下推进栈。

(5) 出栈——将栈顶元素托出栈。

(6) 读栈顶——返回栈顶元素。

3.2　顺　序　栈

与线性表类似，栈的存储结构也分为顺序存储结构和链式存储结构。顺序存储结构的栈简称为顺序栈，链式存储结构的栈称为链栈。

3.2.1　顺序栈存储的定义

与顺序线性表类似，顺序栈也需要通过一个一维数组存储元素，同时设置栈顶元素的位置下标，即：

顺序栈＝一维数组＋栈顶指示

顺序栈的存储结构如图 3.2 所示。

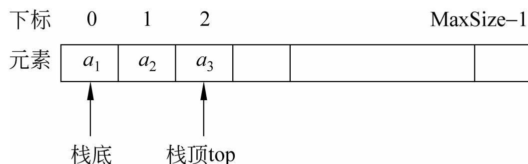

图 3.2　顺序栈的存储结构

具体地说，顺序栈的数据类型描述如下：

```
#define MAX_SIZE100                    /* 设置最大元素个数 */
typedef int Elemtype;
typedef struct
{
    Elemtype stack[MAX_SIZE];          /* 堆栈的元素个数 */
    int top;                           /* 栈顶位置 */
}sqstack;
```

若将顺序栈 st 定义为

```
SeqStack st=new SeqStack();
```

顺序栈 st 中序号为 i 的元素对应数组的下标是 $i-1$，即用 st.elem[$i-1$]表示，st 的栈

顶用 st. top 表示。

此外，在栈的上述存储表示下，不难得到以下栈空及栈满条件。

栈空条件：

```
st.top=-1
```

栈满条件：

```
st.top=MaxSize-1
```

3.2.2 顺序栈的基本运算

根据顺序栈的运算定义，可实现顺序栈的以下操作。

1. 栈初始化

栈的初始化实现比较简单，即将栈顶 top 的值设置为－1 即可。算法实现如下：

```
sqstack* StackInit()
{
    sqstack *s=(sqstack*)malloc(sizeof(sqstack));
    if (NULL==s)
        return NULL;
    s->top=-1 ;
    return s ;
}
```

2. 判断栈是否为空

在判断栈是否为空时，只需将栈顶指示 top 值与－1 相比即可，若 top 值为－1，则表示顺序栈中不包含任何元素。算法实现如下：

```
intStackEmpty(sqstack *s)
{
    if(s->top <0)
        return 1 ;
    return 0 ;
}
```

3. 求栈的长度

栈的长度即为栈中数组的元素个数，因为 top 值总是指向最后一个元素，考虑到当 top 值为 0 时，已经有一个元素存在，则元素的个数为 top＋1。算法实现如下：

```
int StackLength(sqstack *q)
{
    if (NULL==q)
        return 0;
    return (q->top+1);
}
```

4. 进栈操作

假设顺序栈中包含元素(a_1,a_2,a_3),当将元素 e 入栈时,实际就是要在栈顶位置插入该元素。相关算法如图 3.3 所示,具体描述为:

(1) 栈顶指示 top 朝栈的增长方向前进一步(即 top 值增 1)。

(2) 将元素放入栈中由当前栈顶 top 指向的位置上。

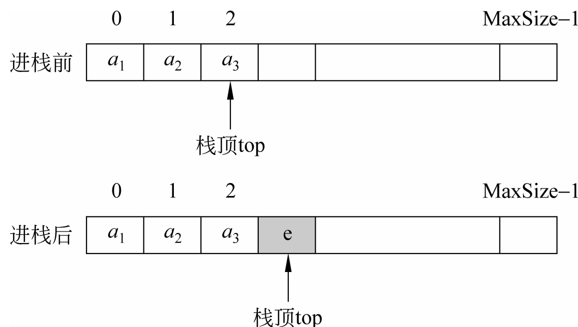

图 3.3　将元素入栈

应该注意的是,在栈的这种静态实现中,进行进栈运算时,必须先进行栈满检查,以避免错误。

```
//插入元素 x 为新的栈顶元素
int Push(sqstack *s, Elemtype x)
{
    if(s->top >= (MAX_SIZE -1))          /* 栈满 */
    {
        printf("溢出\n");
        return 0 ;
    }
    s->top++;                            /* 栈顶指针加 1 */
    s->stack[s->top]=x ;                 /* 将新元素 x 赋值给栈顶空间 */
    return 1 ;
}
```

5. 出栈操作

同样假设顺序栈中包含元素(a_1,a_2,a_3),现将 a_3 元素出栈,只需将栈顶指示 top 后退一步(即 top 值减 1)即可,如图 3.4 所示。同时若需要在出栈的同时返回该出栈元素,还需通

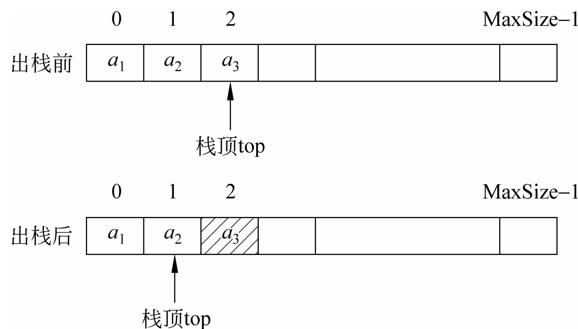

图 3.4　将元素出栈

过一个临时变量获取 a_3 并返回。应该注意的是，出栈前应进行栈空检查。

相关的算法实现如下：

```
//若栈不空,则删除 s 栈顶元素,返回其值,修改栈顶指针
Elemtype Pop(sqstack * s)
{
    Elemtype x ;
    if (s->top <0)                /* 检查堆栈是否为空 */
        return NULL ;
    x=s->stack[s->top];           /* 将要删除的栈顶元素返回 */
    s->top--;                     /* 栈顶指针减 1 */
    return x;
}
```

6. 获取栈顶元素

根据栈顶指示 top，可以直接获取最后入栈的元素。应该注意的是，在进行读取之前，也要进行栈空检查。

相关的算法实现如下：

```
Elemtype GetTop(sqstack * s)
{
    if(s->top<0)return NULL ;
        return (s->stack[s->top]);
}
```

要测试上述这些方法，可以使用如下语句。

```
int _tmain(int argc, _TCHAR* argv[])
{
    sqstack* myStack=StackInit();
    if (NULL==myStack)
        return -1;
    printf("IsEmpty: %d, Length: %d\n", StackEmpty(myStack), StackLength(myStack));
    Push(myStack, 100);
    Push(myStack, 200);
    Push(myStack, 300);
    printf("IsEmpty: %d, Length: %d\n", StackEmpty(myStack), StackLength(myStack));
    int val=Pop(myStack);
    printf("IsEmpty: %d, Length: %d\n", StackEmpty(myStack), StackLength(myStack));
    return 0;
}
```

程序运行的结果如图 3.5 所示。

大家可能注意到，这些栈的运算都极其简单，因此，在实际编程中，有时并不将这些操作设计为方法，而是直接以语句的方式操作。不过，当涉及的栈较多，或栈的元素较为复杂，或要在多个地方进行栈的操作，还是应该采用方法调用的方式，这既符合结构化程序设计的要