第3章 chapter 3

栈和队列

栈和队列广泛应用于计算机软硬件系统中。在编译系统、操作系统等系统软件和各类应用软件中经常需要使用栈和队列完成特定的算法设计。它们的逻辑结构和线性表相同，但它们是一种特殊的线性表。其特殊性在于运算操作受到了一定限制，因此栈和队列又被称为操作受限的线性表。栈按“后进先出”的规则进行操作，队列则按“先进先出”的规则进行操作。

【本章学习要求】

掌握：栈的基本概念，存储结构以及入栈、出栈等基本操作。

掌握：在处理实际问题中如何运用栈特点解决问题。

了解：栈在递归实现过程中的作用。

掌握：队列的基本概念，存储结构和入队、出队等基本操作。

了解：如何运用队列解决实际问题。

3.1 栈

3.1.1 栈的定义及基本操作

栈是限制在表的一端进行插入和删除的线性表。在线性表中允许插入、删除的这一端称为栈顶，栈顶的当前位置是动态变化的；不允许插入和删除的另一端称为栈底，栈底是固定不变的。当表中没有元素时称为空栈。栈的插入操作称为进栈、压栈或入栈，栈的删除操作称为退栈或出栈。如图 3.1 所示栈的进栈和出栈过程，进栈的顺序是 e_1、e_2、e_3，出栈的顺序为 e_3、e_2、e_1，所以栈又称为后进先出线性表(last in first out)，简称 LIFO 表或称先进后出线性表。

在日常生活中，有很多后进先出的例子，如食堂里碟子在叠放时是从下到上，从大到小，在取碟子时，则是从上到下，从小到大。在程序设计中，常常需要栈这样的数据结构，使得取数据与保存数据时呈相反的顺序。对于栈，常用的基本操作有：

(1) 栈初始化：Init_Stack()。

初始条件：栈 S 不存在。

操作结果：构造了一个空栈 S。

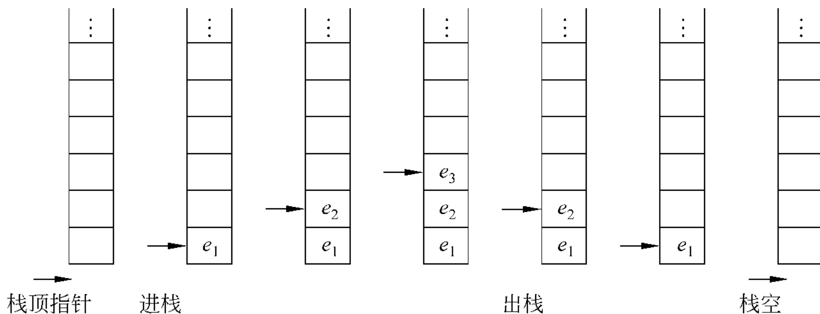

图 3.1 进出栈示意图

(2) 判栈空：Empty_Stack(S)。

操作结果：若 S 为空栈返回为 1，否则返回为 0。

(3) 入栈：Push_Stack(S,x)。

初始条件：栈 S 已存在。

操作结果：在栈 S 的顶部插入一个新元素 x，x 成为新的栈顶元素。栈发生变化。

(4) 出栈：Pop_Stack(S)。

初始条件：栈 S 存在且非空。

操作结果：栈 S 的顶部元素从栈中删除，栈中少了一个元素。栈发生变化。

(5) 取栈顶元素：GetTop_Stack(S)。

初始条件：栈 S 存在且非空。

操作结果：栈顶元素作为结果返回，栈不变化。

(6) 销毁栈：Destroy_Stack(S)。

初始条件：栈 S 已存在。

操作结果：销毁一个已存在的栈。

3.1.2 栈的顺序存储及操作实现

栈的存储与一般线性表的实现类似，也有两种存储方式：顺序存储和链式存储。

利用顺序存储方式实现的栈称为顺序栈。类似于顺序表的定义，要分配一块连续的存储空间存放栈中的元素，用一个一维数组来实现：DataType data[MAXSIZE]，栈底位置可以固定设置在数组的任一端，如固定在下标为－1 的位置，而栈顶指示当前实际的栈顶元素位置，它是随着插入和删除而变化的，用一个 int top 变量指明当前栈顶的位置，同样将 data 和 top 封装在一个结构中，顺序栈的类型描述如下：

```
#define MAXSIZE  100
typedef  struct {
    DataType  data[MAXSIZE];
    int  top;
}SeqStack, * PSeqStack;
```

定义一个指向顺序栈的指针：

```
PSeqStack  S;
S=(PSeqStack)malloc(sizeof(SeqStack));
```

栈的顺序存储如图 3.2 所示。

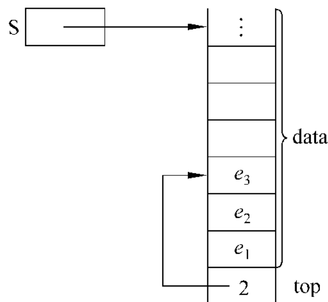

图 3.2　栈的存储示意图

由于顺序栈是静态分配存储，而栈的操作是一个动态过程：随着入栈的进行，有可能栈中元素的个数超过给栈分配的最大空间的大小，这时产生栈的溢出现象——称为上溢；随着出栈的进行，也有可能栈中元素全部出栈，这时栈中再也没有元素出栈了，也会造成栈的溢出现象——称为下溢。所以，在下面的操作实现中，请读者注意在出栈和入栈时要首先进行栈空和栈满的检测。

顺序栈的基本操作实现如下所示。

1. 初始化空栈

顺序栈的初始化即构造一个空栈，要返回一个指向顺序栈的指针。首先动态分配存储空间，然后，将栈中 top 置为−1，表示空栈。具体算法如下所示。

【算法 3.1】

```
PSeqStack  Init_SeqStack(void)
{  /*创建一顺序栈,入口参数无,返回一个指向顺序栈的指针,为零表示分配空间失败*/
    PSeqStack  S;
    S=(PSeqStack)malloc(sizeof(SeqStack));
    if (S)
      S->top=-1;
    return S;
}
```

2. 判栈空

判断栈中是否有元素，算法思想：只需判断 top 是否等于−1 即可。具体算法如下所示。

【算法 3.2】

```
int Empty_SeqStack(PSeqStack  S)
{  /*判断栈是否为空,入口参数:顺序栈,返回值:1表示为空,0表示非空*/
   if (S->top==-1)
      return 1;
   else
      return 0;
}
```

3. 入栈

入栈操作是在栈的顶部进行插入操作，也即相当于在线性表的表尾进行插入，因而无

须移动元素。算法思想是，首先判断栈是否已满，若满退出，否则，由于栈的 top 指向栈顶，只要将入栈元素赋到 top＋1 的位置，同时执行 top＋＋即可。具体算法如下所示。

【算法 3.3】

```
int Push_SeqStack (PSeqStack  S, DataType  x)
{  /* 在栈顶插入一新元素 x,入口参数:顺序栈,返回值:1 表示入栈成功,0 表示失败。 */
  if (S->top==MAXSIZE-1)
    return 0;                       /* 栈满不能入栈 */
  else
  {   S->top++;
      S->data[S->top]=x;
      return 1;
  }
}
```

4. 出栈

出栈操作是在栈的顶部进行删除操作，也即相当于在线性表的表尾进行删除，因而无须移动元素。算法思想：首先判断栈是否为空，若空退出，否则，由于栈的 top 指向栈顶，只要修改 top 为 top －1 即可。具体算法如下所示。

【算法 3.4】

```
int  Pop_SeqStack(PSeqStack  S, DataType*x)
{  /* 删除栈顶元素并保存在*x,入口参数:顺序栈,返回值:1 表示出栈成功,0 表示失败。 */
    if (Empty_SeqStack (S))
        return 0;                   /* 栈空不能出栈 */
    else
    {   *x=S->data[S->top];
       S->top--;
       return 1;
    }
}
```

5. 取栈顶元素

取栈顶元素操作是取出栈顶指针 top 所指的元素值。算法思想是，首先判断栈是否为空，若空退出，否则，由于栈的 top 指向栈顶，返回 top 所指单元的值，栈不发生变化。具体算法如下所示。

【算法 3.5】

```
int  GetTop_SeqStack(PSeqStack  S, DataType  *X)
{  /* 取出栈顶元素,入口参数:顺序栈,被取出的元素指针,这里用指针带出栈顶值 */
   /* 返回值:1 表示成功,0 表示失败 */
   if (Empty_SeqStack (S))
      return 0;                             /* 给出栈空信息 */
```

```
    else
        * x=S->data[S->top];                /* 栈顶元素存入 * x 中 * /
        return (1);

}
```

6. 销毁栈

顺序栈被构造，使用完后，必须要销毁，否则可能会造成申请的内存不能释放，顺序栈的销毁操作是初始化操作的逆运算。由于要修改栈的指针变量，所以要将指针地址传给该函数。首先判断要销毁的栈是否存在，然后在顺序表存在的情况下释放该顺序表所占用的空间，将顺序栈指针赋 0。具体算法如下所示。

【算法 3.6】

```
void Destroy_ SeqStack (PSeqStack  * S)
{  /* 销毁顺序栈,入口参数:为要销毁的顺序栈指针地址,无返回值 * /
   if ( * S)
      free ( * S);
   * S=NULL;
   return;
}
```

设调用函数为主函数，主函数对初始化函数和销毁函数的调用如下：

```
main()
{  PSeqStack  S;
   S=Init_SeqStack();
   ⋮
   Destroy_ SeqStack (&S);
}
```

3.1.3 栈的链式存储及操作实现

栈也可以用链式存储方式实现，一般链栈用单链表表示，其结点结构与单链表的结构相同。即结点结构为：

```
typedef  struct node {
    DataType data;
    struct node  * next;
}StackNode, * PStackNode;
```

因为栈的主要操作是在栈顶插入、删除，显然以链表的头部做栈顶是最方便的，而且没有必要像单链表那样为了操作方便附加一个头结点，同时为了方便操作和强调栈顶是栈的一个属性，我们定义一个栈：

```
typedef struct {
```

```
    PStackNode  top;
}LinkStack, * PLinkStack;
PLinkStack  S;
S=(PLinkStack)malloc(sizeof(LinkStack));
```

栈的链式存储如图3.3所示。

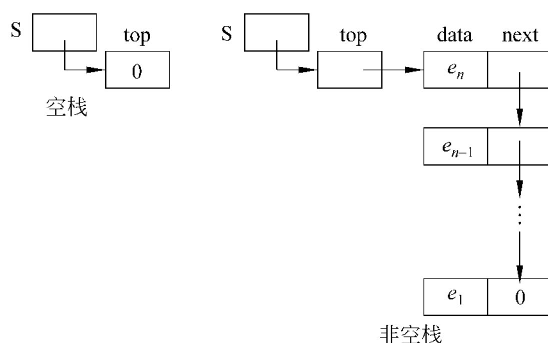

图3.3 栈的链式存储示意图

链栈基本操作的实现如下。

1. 初始化空栈

【算法3.7】

```
PLinkStack  Init_LinkStack(void)
{  /*初始化链栈,入口参数:空,返回值:链栈指针,null表示初始化失败*/
     PLinkStack S;
     S=(PLinkStack)malloc(sizeof(LinkStack));
     if (S)   S->top=NULL;
     return (S);
}
```

2. 判栈空

【算法3.8】

```
int  Empty_LinkStack(PLinkStack  S)
{  /*判断链栈是否为空,入口参数:链栈指针,返回值:1表示栈空,0表示栈非空*/
   return (S->top==NULL);
}
```

3. 入栈

【算法3.9】

```
int  Push_LinkStack(PLinkStack  S, DataType x)
```

```
{  /*进栈,入口参数:链栈指针,进栈元素,返回值:1表示入栈成功,0表示失败*/
   PStackNode p;
   p=(PStackNode) malloc(sizeof(StackNode));
   if (!p)
   {
      printf("内存溢出");
      return (0);
   }
   p->data=x;
   p->next=S->top;
   S->top=p;
   return (1);
}
```

4. 出栈

【算法 3.10】

```
int Pop_LinkStack (PLinkStack  S, DataType *x)
{  /*出栈,返回值:1表示出栈成功,0表示失败,*x保存被删除的元素值*/
   PStackNode p;
   if (Empty_LinkStack (S))
   {
      printf("栈空,不能出栈");
      return (0);
   }
   *x=S->top->data;
   p=S->top;
   S->top=S->top->next;
   free (p);
   return (1);
}
```

5. 取栈顶元素

【算法 3.11】

```
int GetTop_LinkStack (PLinkStack  S, DataType  *x)
{  /*得到栈顶元素,入口参数:链栈指针,出栈元素存放空间地址*/
   /*返回值:1表示出栈成功,0表示失败*/
   if (Empty_LinkStack (S))
   {
       printf("栈空");
       return (0);                      /*栈空*/
   }
```

```
    * x=S->top->data;                    /* 栈顶元素存入 * x 中 */
    return (1);
}
```

6. 销毁栈

链栈被构造,使用完后,必须要销毁,否则可能会造成申请的内存不能释放。具体算法如下:

【算法 3.12】

```
void Destroy_LinkStack (PLinkStack  * LS)
{   /* 销毁链栈,入口参数:要销毁的链栈指针地址,无返回值 */
    PStackNode  p, q
    if ( * LS)
    {
        p= ( * LS)->top;
        while(p)
        {
            q=p;
            p=p->next;
            free(q);
        }
        free ( * LS);
    }
    * LS=NULL;
    return;
}
```

主函数对初始化函数和销毁函数的调用方法类似算法 3.6。

3.2 栈的应用举例

由于栈的"后进先出"特点,在很多实际问题中都利用栈做一个辅助的数据结构来实现逆向操作的求解,下面通过几个例子进行说明。

【例 3.1】 数制转换问题。

将十进制数 N 转换为 r 进制的数,其转换方法利用辗转相除法:以 $N=1234$,$r=8$ 为例转换方法如下:

N	$N/8$(整除)	$N\%8$(求余)	
			低 ↑
1234	154	2	
154	19	2	
19	2	3	
2	0	2	高

所以：$(1234)_{10}=(2322)_8$

我们看到所转换的八进制数按从低位到高位的顺序产生，而通常的输出应该从高位到低位，与计算过程正好相反，因此转换过程中每得到一位八进制数则进栈保存，转换完毕后依次出栈则正好是转换结果。

算法思想如下：

(1) 初始化栈，初始化 N 为要转换的数，r 为进制数。

(2) 判断 N 的值，为 0 转步骤(4)，否则 $N\%r$ 压入栈 s 中。

(3) 用 N/r 代替 N 转步骤(2)。

(4) 出栈，出栈序列即为结果。

具体算法如下所示。

【算法 3.13】

```
typedef  int  DataType;
int conversion(int n,int  r)
{  PSeqStack  S;                              /*定义一个顺序栈*/
   DataType  x;
   if (!r)
   {
      printf("基数不能为 0");
      return(0);
   }
   S=Init_SeqStack();                         /*初始化栈*/
   if (!S)
   {
      printf("栈初始化失败");
      return(0);
   }
   while (n)
   {
      Push_SeqStack (S,n%r);                  /*余数入栈*/
      n=n/ r;                                 /*商作为被除数继续*/
   }
   while (!Empty_SeqStack(S))                 /*直到栈空退出循环*/
   {    Pop_SeqStack (S,&x);                  /*弹出栈顶元素*/
        printf ("%d ",x);                     /*输出栈顶元素*/
   }
   Destroy_ SeqStack (&S);                    /*销毁栈*/
}
```

当应用程序中需要一个与数据保存时顺序相反的数据时，通常使用栈。用顺序栈的情况较多。

【例 3.2】 利用栈实现迷宫的求解。

问题：这是实验心理学中的一个经典问题，心理学家把一只老鼠从一个无顶盖的大

盒子的入口处赶进迷宫。迷宫中设置很多隔壁,对前进方向形成了多处障碍,心理学家在迷宫的唯一出口处放置了一块奶酪,吸引老鼠在迷宫中寻找通路以到达出口。

求解思想:回溯法是一种不断试探且及时纠正错误的搜索方法。下面的求解过程即是回溯法。从入口出发,按某一方向向前探索,若能走通并且未走过,即某处可以到达,则到达新点,否则试探下一方向;若所有的方向均没有通路,则沿原路返回前一点,换下一个方向再继续试探,直到找到一条通路,或无路可走又返回到入口点。

在求解过程中,为了保证在到达某一点后不能向前继续行走(无路)时,能正确返回前一点以便继续从下一个方向向前试探,则需要用一个栈保存所能够到达的每一点的下标及从该点前进的方向。

实现该算法需要解决的四个问题。

1. 表示迷宫的数据结构

设迷宫为 m 行 n 列,利用 maze[m][n]来表示一个迷宫,maze[i][j]=0 或 1;其中:0 表示通路,1 表示不通,当从某点向下试探时,中间点有 4 个方向可以试探(见图 3.4),而 4 个角点有两个方向,其他边缘点有三个方向,为使问题简单化我们用 maze[$m+2$][$n+2$]来表示迷宫,而迷宫的四周的值全部为 1,这样做使问题简单了,每个点的试探方向全部为 4(实际上是 8 个方向,本书为将问题简单化只考虑 4 个方向),不用再判断当前点的试探方向有几个。

如图 3.4 表示的迷宫是一个 6×8 的迷宫。

入口(1,1)

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	0	1	1	1	1
2	1	0	0	0	0	1	1	1	1	1
3	1	0	1	0	0	0	0	0	1	1
4	1	0	1	1	1	0	0	1	1	1
5	1	1	0	0	1	1	0	0	0	1
6	1	0	1	1	0	0	1	1	0	1
7	1	1	1	1	1	1	1	1	1	1

出口(6,8)

图 3.4 用 maze[$m+2$][$n+2$]表示的迷宫

入口坐标为(1,1),出口坐标为(6,8)。

迷宫的定义如下:

```
#define  m   6                    /* 迷宫的实际行 */
#define  n   8                    /* 迷宫的实际列 */
int maze [m+2][n+2];
```

2. 试探方向

在上述表示迷宫的情况下,每个点有 4 个方向去试探,如当前点的坐标(x,y),与其