Chapter 3 第3章 栈和队列

内容提要

本章介绍两种常用的数据结构——栈和队列。它们的逻辑结构和线性表相同，其特点在于运算受到了限制：栈按“后进先出”的规则进行操作，队按“先进先出”的规则进行操作，故称运算受限制的线性表。本章还介绍了栈和队列的应用。

学习目标与重点

- 理解栈和队列的定义、特点；
- 掌握栈和队列的各种存储结构；
- 重点掌握在顺序栈和链栈上实现进栈、出栈、判栈空和判栈满等基本操作；
- 重点掌握在循环队列中实现进队列、出队列、判队列空和判队列满等基本操作；
- 了解栈和队列在计算机软件设计中的应用。

关键术语

栈；入栈；出栈；队列；入队；出队；表达式求值

3.1 栈

3.1.1 栈的定义

栈是限定仅在表尾进行插入或删除的线性表。允许插入、删除的这一端即表尾称为栈顶，表的另一端称为栈底。当表中没有元素时称为空栈。如图 3.1 所示的栈中有 n 个元素，进栈的顺序是 $a_1,a_2,\cdots,a_n$，当需要出栈时其顺序为 $a_n,a_{n-1},\cdots,a_1$，所以栈又称为后进先出的线性表(Last In First Out)，简称 LIFO 表。

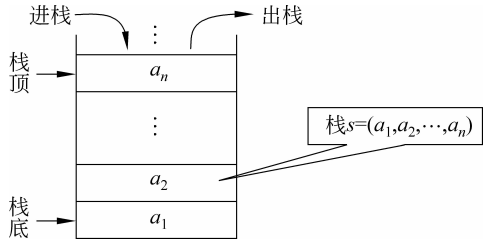

图 3.1 栈示意图

在日常生活中，有很多后进先出的例子，如食堂里的盘子，在叠放时是从下到上，从大到小，在取盘子时，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，使得与保存数据时相反顺序来使用这些数据，这时就需要用一个栈来实现。

抽象数据类型栈的定义如下：

```
ADT Stack {
    数据对象:
    D={ ai|ai∈ElemSet, i=1,2,…,n,  n≥0 }
    /* 称 n 为栈的长度; 称 n=0 时的栈为空栈 */
    数据关系:
    R1={ <ai-1,ai>|ai-1,ai∈D,  i=2,…,n }
    /* 设栈为(a1,a2,…,ai,…,an), 称 i 为 ai 在栈中的位序 */
    基本操作:
    (1) Init_Stack(s)
    初始条件: 栈 s 不存在
    操作结果: 构造了一个空栈
    (2) Empty_Stack(s)
    初始条件: 栈 s 已存在
    操作结果: 若 s 为空栈返回为 1,否则返回为 0
    (3) Push(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) Pop(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) Top_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
    …
} ADT Stack
```

3.1.2 顺序栈的存储结构和操作的实现

由于栈是运算受限的线性表,因此线性表的存储结构对栈也是适用的,只是操作不同而已。

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,栈中的数据元素用一个预设的足够长度的一维数组 elem 来实现。栈底位置可以设置在数组的任一个端点,用一个 base 来作为指向栈底的指针。base 为 NULL 时,表示栈不存在。栈顶是随着插入和删除而变化的,用一个 top 来作为指向栈顶的指针,指明当前栈顶的位置。同样,将 elem、top 和 base 封装在一个结构中。

顺序栈的类型描述如下:

```
#define  MAXSIZE  100
typedef  struct
{  ElemType  elem[MAXSIZE];
   int  base, top;
}SeqStack
```

定义一个指向顺序栈的指针的代码如下:

```
SeqStack  *s;
```

通常将 0 下标端设为栈底,指针 top 指向栈顶元素的后一个元素位置。空栈时栈顶

指针 top＝base＝0，此时出栈为下溢(Underflow)；入栈时，栈顶指针加 1，即 s－＞top＋＋。出栈时，栈顶指针减 1，即 s－＞top－－。栈操作的示意图如图 3.2 所示。

图 3.2(a)所示是空栈，图 3.2(d)所示是 A、B、C、D、E 、F 6 个元素依次入栈之后栈满的示意图。此时入栈，则上溢(Overflow)。通过这个示意图可以深刻理解栈顶指针的作用。

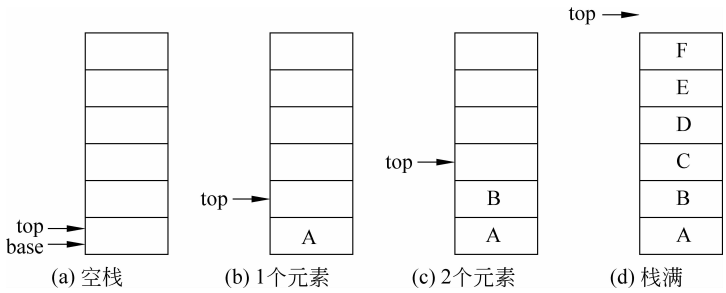

图 3.2　栈顶指针 top 与栈中数据元素的关系示意图

在上述存储结构上基本操作的实现如下。

(1) 置空栈：

```
SeqStack  * Init_SeqStack()
{
    SeqStack  * s;
    s=malloc(MAXSIZE * sizeof(SeqStack));
    s->top=s->base=s;
    s->top=0;  return s;
}
```

(2) 判空栈：

```
int Empty_SeqStack(SeqStack * s)
{
    if(s->top==s->base)  return 1;
    else  return 0;
}
```

(3) 入栈：

```
int Push(SeqStack * s, elemtype  x)
{  if(s->top==MAXSIZE)  return 0;      //栈满不能入栈
   else {
        s->elem[s->top]=x;
        s->top++;
        return 1;
   }
}
```

(4) 出栈：

```
int Pop(SeqStack * s, elemtype * x)
```

```
{  if  (Empty_SeqStack(s))  return 0;   //栈空不能出栈
    else  {
        s->top--;
        *x=s->elem[s->top];             //栈顶元素存入*x,返回
        return 1;
    }
}
```

(5) 取栈顶元素：

```
elemtype Top_SeqStack(SeqStack *s)
{
    if( Empty_SeqStack(s)) return 0;     //栈空
    else return(s->elem[s->top]);
}
```

说明：

(1) 对于顺序栈，入栈时首先判栈是否满了，栈满的条件为：s->top==MAXSIZE。栈满时，不能入栈；否则出现空间溢出，引起错误，这种现象称为上溢。

(2) 出栈和读栈顶元素操作，先判栈是否为空，为空时不能操作，否则产生错误。通常将栈空作为一种控制转移的条件。

3.1.3 链栈的存储结构和操作的实现

栈的链式存储结构称为链栈，它是运算受限的单链表。其结点结构与单链表的结构相同，在此用 LinkStack 表示，即：

```
typedef  struct  stacknode
{  ElemType data;
   struct stacknode *next;
}stacknode,* LinkStack;
```

说明：top 为栈顶指针的代码如下：

```
LinkStack  top;
```

因为栈的主要运算是在栈顶插入、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运算方便附加一个头结点。通常将链栈表示成如图 3.3 所示的形式。链栈基本操作的实现如下。

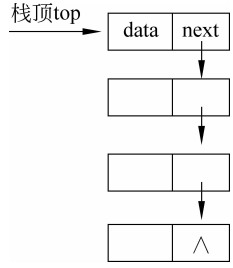

图 3.3 链栈示意图

(1) 置空栈：

```
LinkStack  Init_LinkStack()
{
    return  NULL;
}
```

(2) 判栈空：

```
int  Empty_LinkStack(LinkStack  top)
```

```
{
    if(top==NULL)  return 1;
    else  return  0;
}
```

(3) 入栈：

```
LinkStack  Push_LinkStack(LinkStack  top, datatype x)
{
     StackNode  * s;
     s=malloc(sizeof(StackNode));
     s->data=x;
     s->next=top;
     top=s;
     return top;
}
```

(4) 出栈：

```
LinkStack  Pop_LinkStack(LinkStack  top, datatype  * x)
{
    StackNode  * p;
    if(top==NULL)  return NULL;
    else {
         * x=top->data;
         p=top;
         top=top->next;
         free(p);
         return  top;
    }
}
```

3.2　栈的应用

由于栈具有“先进后出”的特点，在很多实际问题中都利用栈做一个辅助的数据结构来进行求解，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 d 进制的数，其转换方法利用辗转相除法。下面以 $N=1348$，$d=8$ 为例说明此转换方法。

N	$N/8$(整除)	$N\%8$(求余)	
1348	168	4	↑ 低
168	21	0	|
21	2	5	|
2	0	2	| 高

所以，$(1348)_{10}=(2504)_8$。

可以看出，转换得到的八进制数是按低位到高位的顺序产生的，而通常的输出是从高位到低位的，恰好与计算过程相反。因此，转换过程中每得到一位 8 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($N>0$)：

(1) 若 $N\neq0$，则将 $N\%r$ 压入栈 s 中，执行步骤(2)；若 $N=0$，将栈 s 的内容依次出栈，算法结束。

(2) 用 N/r 代替 N，转到步骤(1)。

算法如下：

```
void conversion()       //对于输入的任意一个非负十进制整数,输出等价的八进制数
{
    s=Init_SeqStack();
    scanf("%d",N);
    while(N)
    {
        Push(S,N%8);  N=N/8;
    }
    while(!Empty_SeqStack(S))
    {
        Pop(S,e); printf("%d",e);
    }
}
```

【例 3.2】 表达式求值。

表达式求值是程序设计语言编译中一个最基本的问题。它的实现是栈应用的一个典型例子。下面介绍一种简单直观、广为使用的表达式求值的算法——算符优先法。

表达式是由操作数、运算符和括号组成的有意义的式子。运算符从操作数的个数上分，有单目运算符和多目运算符；从运算类型上分，有算术运算符、关系运算符和逻辑运算符。在此仅限于讨论只含二目运算符的算术表达式。

(1) 中缀表达式求值。

中缀表达式是指每个二目运算符在两个操作数的中间的表达式。假设所讨论的算术运算符包括＋、－、＊、/、%、^和“()”，不难将它推广到更一般的表达式上。

运算符的优先级为：^、→、＊、/、%→＋、－→ ＃次序降低。即^的优先级高于＊、/、%，而＊、/、%三者的优先级相等，即＊、/、%的优先级高于＋、－，而＋、－二者优先级相等。＋、－的优先级高于＃，＃为表达式起始符和表达式结束符。

规定同等优先级的，栈顶优先级高，如＋和－优先级相同。但是如果＋在栈顶，则其优先级高于－；“(”和“)”优先级相等，“(”在栈顶时，小于除了“)”之外所有符号的优先级；“)”在栈顶时，大于除了“(”之外所有符号的优先级。

中缀表达式的求值过程如下。

① 需要两个栈：操作数栈 s_1 和运算符栈 s_2。首先置操作数栈为空栈，表达式起始符＃为运算符栈的栈底元素。

② 当自左至右扫描表达式的每一个字符时，若当前字符是操作数，则入操作数栈；是

运算符时，若这个运算符比栈顶运算符优先级高则入栈，继续向后处理；若优先级相等则出栈，继续向后处理；若这个运算符比栈顶运算符优先级低，则从操作数栈出栈两个操作数，从运算符栈出栈一个运算符进行运算，并将其运算结果入操作数栈。然后继续处理当前字符，直到遇到结束符。

中缀表达式“#3*2^(4+2*2-1*3)-5#”求值过程中两个栈的状态情况如表 3.1 所示。

表 3.1　中缀表达式 #3*2^(4+2*2-1*3)-5# 的求值过程

读字符	操作数栈 s_1	运算符栈 s_2	说　明
#	空	#	# 入栈 s_2
3	3	#	3 入栈 s_1
*	3	# *	* 入栈 s_2
2	3,2	# *	2 入栈 s_1
^	3,2	# * ^	^入栈 s_2
(	3,2	# * ^(	(入栈 s_2
4	3,2,4	# * ^(	4 入栈 s_1
+	3,2,4	# * ^(+	+ 入栈 s_2
2	3,2,4,2	# * ^(+	2 入栈 s_1
*	3,2,4,2	# * ^(+ *	* 入栈 s_2
2	3,2,4,2,2	# * ^(+ *	2 入栈 s_1
-	3,2,4,4	# * ^(+	计算 2*2=4，结果入栈 s_1
	3,2,8	# * ^(	计算 4+4=8，结果入栈 s_1
	3,2,8	# * ^(-	- 入栈 s_2
1	3,2,8,1	# * ^(-	1 入栈 s_1
*	3,2,8,1	# * ^(- *	* 入栈 s_2
3	3,2,8,1,3	# * ^(- *	3 入栈 s_1
)	3,2,8,3	# * ^(-	计算 1*3，结果 3 入栈 s_1
	3,2,5	# * ^(	计算 8-3，结果 5 入栈 s_2
	3,2,5	# * ^	(出栈
-	3,32	# *	计算 2^5，结果 32 入栈 s_1
	96	#	计算 3*32，结果 96 入栈 s_1
	96	#-	- 入栈 s_2
5	96,5	#-	5 入栈 s_1
#	91	#	计算 96-5，结果 91 入栈 s_1
	91	空	# 出栈 s_2，表达式结果为 91

为了处理方便，编译程序常把中缀表达式首先转换成等价的后缀表达式。后缀表达式的运算符在操作数之后。在后缀表达式中，不再引入括号，所有的计算按运算符出现的顺序，严格从左向右进行，而不用再考虑运算规则和级别。中缀表达式“3+2*(5-1)”的后缀表达式为“3251-*+”。

(2) 后缀表达式求值。计算一个后缀表达式,算法上比计算一个中缀表达式简单得多。这是因为后缀表达式中既无括号又无优先级的约束。

中缀表达式转化为后缀表达式的步骤如下。

① 使用一个栈,先初始化栈,#入栈。

② 当自左至右扫描表达式的每一个字符时,若当前字符是操作数,则输出;若当前字符是运算符时,若当前字符大于栈顶字符的优先级时,则入栈,读入下一字符;若当前字符等于栈顶字符的优先级时,则栈顶出栈,读入下一字符;若当前字符小于栈顶字符的优先级时,则输出栈顶。

③ 继续读入字符,遇到结束符#时,结束。

后缀表达式的求值过程如下。

使用一个栈,当从左向右扫描表达式时,每遇到一个操作数就送入栈中保存;每遇到一个运算符就从栈中取出两个操作数进行当前的计算;然后把结果再入栈,直到整个表达式结束,这时栈顶的值就是结果。

课堂小练习:将中缀表达式"3 * 2^(4+2 * 2-1 * 3)-5"转化为后缀表达式:"32422 * +13 * -^ * 5-"。要求写出转化过程,并求值。

【例 3.3】 栈与递归。

栈的一个重要应用是在程序设计语言中实现递归过程。现实中,有许多实际问题是递归定义的,这时用递归方法可以使许多问题的结果大大简化。下面以求 $n!$ 为例进行说明。

$n!$ 的定义如下:

$$n! = \begin{cases} 1 & n = 0 \quad // \text{递归终止条件} \\ n(n-1) & n > 0 \quad // \text{递归步骤} \end{cases}$$

根据定义可以很自然地写出相应的递归函数。

```
int fact(int n)
{
    if(n==0)  return 1 ;
    else return  (n * fact(n-1)) ;
}
```

递归函数都有一个终止递归的条件,如本例 n 值为 0 时,将不再继续递归下去。

递归函数的调用类似于多层函数的嵌套调用,只是调用单位和被调用单位是同一个函数。在每次调用时系统将属于各个递归层次的信息组成一个活动记录,这个记录中包含着本层调用的实参、返回地址、局部变量等信息,并将这个活动记录保存在系统的"递归工作栈"中。每当递归调用一次,就要在栈顶为过程建立一个新的活动记录。一旦本次调用结束,则将栈顶活动记录出栈,根据获得的返回地址信息返回到本次的调用处。下面以求 3!为例说明执行调用时工作栈中的状况。

为了方便将求阶乘,将程序修改如下:

```
main()
{
```

```
    int m,n=3 ;
    m=fact(n) ;
    printf("%d!=%d\n",n,m) ;
}

int fact(int n)
{
    int  f ;
    if(n==0)  f=1 ;
    else f=n * fact(n-1) ;
    return f  ;
}
```

程序的执行过程如图 3.4 所示。

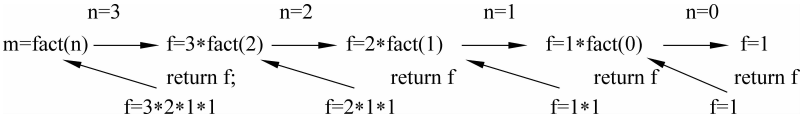

图 3.4　fact(3) 的执行过程

3.3　队列

3.3.1　队列的定义

前面所讲的栈是一种后进先出的数据结构，而在实际问题中还经常使用一种“先进先出”(First In First Out，FIFO)的数据结构：即限定只能在表的一端进行插入，在表的另一端进行删除。我们将这种数据结构称为队或队列，把允许插入的一端叫队尾(Rear)，允许删除的一端叫队头(Front)。如图 3.5 所示是一个有 5 个元素的队列。入队的顺序依次为 a_1、a_2、a_3、a_4、a_5，出队时的顺序将依然是 a_1、a_2、a_3、a_4、a_5。

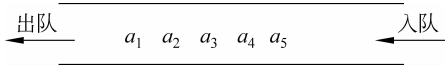

图 3.5　队列示意图

显然，队列也是一种运算受限制的线性表，所以又叫先进先出表。在日常生活中队列的例子很多，如排队买票、食堂排队买饭或自动取款机排队取款，排在队头的人处理完后从队头走掉，而后来的人则必须排在队尾等待。这是为了不造成次序的混乱而采取的一种让先到的客户比晚到的客户先得到服务的办法。

抽象数据类型队列的定义如下：

```
ADT Queue {
    数据对象：
    D={ a_i | a_i ∈ ElemSet, i=1,2,…,n,  n≥0 }  {称 n 为队列的长度}
    数据关系：
    R_1={ <a_(i-1),a_i> | a_(i-1),a_i ∈ D,  i=2,…,n} {约定其中 a_1 端为队列头，a_n 端为队列尾}
    基本操作：
    (1) 队列初始化：Init_Queue(q)
```

初始条件：队 q 不存在
操作结果：构造了一个空队
(2) 入队操作：In_Queue(q,x)
初始条件：队 q 存在
操作结果：对已存在的队列 q,插入一个元素 x 到队尾,队发生变化
(3) 出队操作：Out_Queue(q,x)
初始条件：队 q 存在且非空
操作结果：删除队首元素,并返回其值,队发生变化
(4) 读队头元素：Front_Queue(q,x)
初始条件：队 q 存在且非空
操作结果：读队头元素,并返回其值,队不变
(5) 判队空操作：Empty_Queue(q)
初始条件：队 q 存在
操作结果：若 q 为空队则返回为 1;否则返回为 0
…
} ADT Queue

3.3.2 链队列的存储结构和操作的实现

链式存储结构的队列称为链队列。和链栈类似,用单链表来实现链队列,它是限制仅在表头删除和表尾插入的单链表。显然仅有单链表的头指针不便于在表尾作插入操作,为此再增加一个尾指针,指向链表的最后一个结点,如图 3.6 所示。

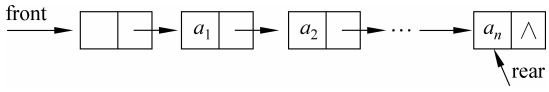

图 3.6 链队示意图

图 3.6 中,头指针 front 和尾指针 rear 是两个独立的指针变量,从结构性上考虑,通常将二者封装在一个结构中。

链队列的描述如下：

```
typedef struct node
{
    ElemType  data;
    struct  node * next;
} QNode;                //链队结点的类型

typedef struct
{
    QNnode   * front, * rear;
}LQueue;                //将头尾指针封装在一起的链队列
```

定义一个指向链队的指针。

```
LQueue  * q;
```

按这种思想建立的带头结点的链队列如图 3.7 所示。