第3章 栈和队列

栈(Stack)是一种最常用和最重要的数据结构,它在计算机科学中应用非常广泛,例如,编译器对表达式的计算和表达式括号的匹配、计算机系统处理函数调用时参数的传递和函数值的返回等。栈是一种特殊的线性表,其特殊性在于它的插入、删除等操作都是在线性表的一端进行,特点是按"后进先出"的规则进行操作,是一种运算受限制的线性表,因此,可称为限定性的数据结构。栈在程序设计中非常重要,程序的调试和运行都需要系统栈的支撑。

队列(Queue)是一种运算受限制的线性表。与栈不同的是:队列是限制在表的两端进行操作的线性表。队列只允许在表的一端插入数据元素而在另一端删除数据元素。队列是软件设计中常用的一种数据结构。队列在操作系统中有重要的应用[①]。队列的逻辑结构和线性表相同,其特点是按"先进先出"的规则进行操作。本章首先介绍栈的概念、栈的存储结构、有关栈的基本运算和栈的应用,然后介绍顺序队列(其中重要介绍循环队列)和链队列的概念及运算。

3.1 栈的概念及运算

3.1.1 栈的概念

1. 定义

栈是限定仅在表尾进行插入或删除操作的线性表,表尾称为栈顶(top),表头称为栈底(bottom)。例如,设有 n 个元素的栈 $S=(a_1,a_2,\cdots,a_n)$,则称 a_1 为栈底元素,a_n 为栈顶元素,如图 3.1 所示。

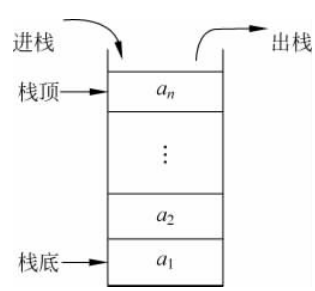


图 3.1 栈的示意图

2. 特点

栈的最主要特点就是"先进后出"(First In Last Out,FILO),或"后进先出"(Last In First Out,LIFO)。在日常生活中有很多栈的例子。例如,往箱子里放书,最先放进去的书

① 在操作系统中进程控制块(PCB)的组织就采用队列,称为 PCB 表。PCB 表示操作系统中最关键、最常用的数据,它的物理结构直接影响到系统的效率。

压在最底下，只能最后拿出来，而最后放进去的书在最上面，可以最先拿出来。又如食堂里洗碗，最先洗好的碗放在最下面，而最先被使用的碗是最后洗好的碗。另外，铁路调度站也采用栈的原理来调度铁路机车。

3.1.2 栈的基本运算

栈的基本运算主要有以下几种。

(1) 建立一个空栈：InitStack(S)。

初始条件：栈 S 不存在。

运算结果：构造一个空栈 S。

(2) 进栈：Push(S,x)。

初始条件：栈 S 存在且非满。

运算结果：在栈顶插入一个值为 x 的元素，栈中增加一个元素。

(3) 出栈：Pop(S,&x)。

初始条件：栈 S 存在且非空。

运算结果：将栈顶元素的值赋给 x，删除栈顶元素，栈中减少了一个元素。

(4) 读栈顶元素：ReadTop(S)。

初始条件：栈 S 存在且非空。

运算结果：返回栈顶元素的值，栈中元素保持不变。

(5) 判栈空：IsEmpty(S)。

初始条件：栈 S 已经存在。

运算结果：若栈空则返回 1，否则返回 0。

(6) 判栈满：IsFull(S)。

初始条件：栈 S 已经存在。

运算结果：若栈满则返回 1，否则返回 0。

(7) 显示栈中元素：ShowStack(S)。

初始条件：栈 S 已经存在且非空。

运算结果：显示栈中所有元素。

3.1.3 一个有趣的问题

有一张由 12 个区域组成的地图如图 3.2 所示，试用最少的颜色对该地图进行着色(每块区域只涂一种颜色)，且要求相邻的颜色互不相同，请打印出所有可能的着色方案。

根据四色定理[①]可以知道：任何多么复杂的地图只需要使用 4 种颜色就可以将相邻的

① 世界近代三大数学难题之一。四色猜想的提出来自英国。1852 年，毕业于伦敦大学的弗南西斯·格思里(Francis Guthrie)来到一家科研单位做地图着色工作时，发现了一种有趣的现象："每幅地图都可以用 4 种颜色着色，使得有共同边界的国家着上不同的颜色。"这个结论能不能从数学上加以严格证明？世界上许多一流的数学家都纷纷参加了四色猜想的大会战，直到 1976 年，美国数学家阿佩尔(Kenneth Appel)与哈肯(Wolfgang Haken)在美国伊利诺斯大学的两台不同的电子计算机上，用了 1200 个小时，做了 100 亿个判断，终于完成了四色定理的证明。四色定理是第一个主要由计算机证明的理论。

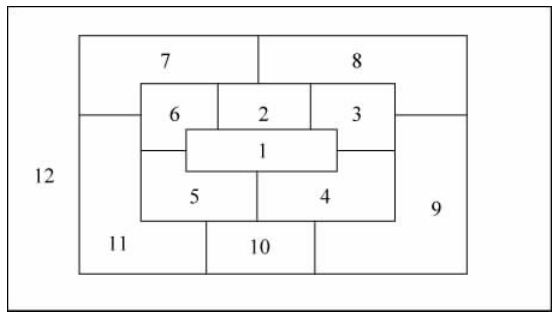


图 3.2 着色问题

区域(或国家)分开。

将 4 种颜色分别用 1、2、3、4 表示时,一种符合条件的着色方案可以如下所示:

1:1 2:2 3:3 4:2 5:3 6:4 7:1 8:4 9:1 10:4 11:2 12:3

由于要寻找所有可能的着色方案,需要使用回溯法,而利用栈可以方便地实现回溯法。栈和递归是紧密联系的,因而着色问题也可以使用递归算法实现。

3.2 栈的存储和实现

3.2.1 栈的顺序表示

用顺序存储方式实现的栈称为顺序栈,顺序栈对应于顺序表。

1. 顺序栈的实现

设栈中数据元素的类型是整型,用一个足够长的一维数组 *s* 来存放元素,数组的最大容量为 STACK_INTSIZE。同时假设栈顶指针 top。

注意:*在以下的程序中,top 不是指向当前的栈顶元素,而是指向下一次将要进栈的元素的存储位置。*

顺序栈可以用 C 语言描述如下:

```
#define  STACK_INTSIZE  50                    /* 分配栈的最大存储空间 */
DataType  s[STACK_INTSIZE];                   /* 用来存放栈中数据元素的内存空间 */
int  top;                                     /* 定义栈顶指针 */
```

可以用结构体数组来实现顺序栈:

```
#define  STACK_INTSIZE  50
#define  char  DataType                       /* 定义栈中数据元素的类型 */
typedef  struct
{ DataType  s[STACK_INTSIZE];
  int  top;
```

```
}Stack;
Stack * st;                          /* 指针 st 用来引用一个顺序栈 */
```

思考

如果栈顶指针 top 指向当前的栈顶元素，则下面顺序栈的基本运算算法如何修改？

2. 顺序栈的操作

设有一个一维数组 s[6]用来存放顺序栈，它的一些基本操作如图 3.3 所示，栈顶指针动态地反映了栈中元素的变化情况。

top＝0 时，表示空栈，如图 3.3 (a)所示。

top＝1 时，表示已经有一个元素进栈，如图 3.3 (b)中元素 E 已进栈；进栈时，栈顶指针 top 上移，top 加 1。

top＝6 时，也即 top＝STACK_INTSIZE，表示栈满，图 3.3 (c)是 6 个元素进栈后的状况，栈已满；出栈时，栈顶指针 top 下移，top 减 1，图 3.3(d)是元素 F 出栈后的状况。

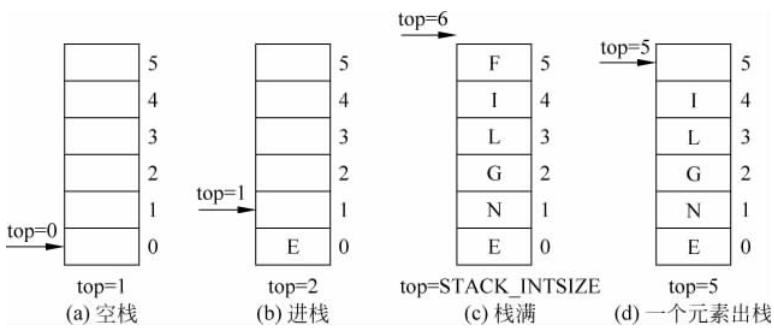


图 3.3　顺序栈操作示意图

3. 顺序栈基本操作的算法

1) 建立一个空栈

【算法 3.1】 顺序栈的建栈算法。

```
    Stack * InitStack()
1   {
2     Stack  * s;
3     s = malloc(sizeof(Stack));
4     s -> top = 0;
5     return  s;
6   }
```

思考

(1) 算法 3.1 中设置 s—> top＝0，表示什么含义？能否设置为 s—> top＝－1？

(2) 返回值 s 是什么类型?

(3) 根据本章的定义,算法中申请了多大的空间?

2) 进栈

【算法 3.2】 顺序栈的进栈算法。

```
    void Push(Stack * st, DataType x)
1   {  if (st->top>=STACK_INTSIZE)
2         printf("\t\t 栈已满,不能入栈!\n");      /* 若栈满则不能进栈,输出出错信息 */
3      else
4      { st->s[st->top]=x;                        /* 元素 x 进栈 */
5        st->top++;                               /* 栈顶指针 top 加 1 */
6      }
7   }
```

思考

(1) 在入栈时如果不判断栈是否满,会出现什么情况?

(2) 如果 st—> top 指向栈顶元素,那么如何判断栈满?

(3) 如果在算法 3.1 中设置 st—> top=—1,则算法 3.2 如何描述入栈操作?

3) 出栈

【算法 3.3】 顺序栈的出栈算法。

```
    void Pop(Stack * st, DataType x)
1   {  if(st->top==0)
2        printf("\t\t 栈空,不能出栈!\n");       /* 若栈空则不能出栈,且输出栈空的信息 */
3      else
4      { st->top--; x=st->s[st->top]; }   /* 栈非空则 top 减 1,元素出栈 */
5   }
```

思考

(1) 为什么只要将栈顶指针下移一个单元,就表示一个元素出栈了?

(2) 原来栈顶中的元素是否还存在,为什么?

(3) 如果 st—> top 指向栈顶元素,那么如何判断栈空?

(4) 如果希望出栈算法能够返回栈顶元素,怎么修改上述算法?

4) 读栈顶元素

【算法 3.4】 顺序栈的读栈顶元素算法。

```
    DataType ReadTop(Stack * st)
1   { DataType x;
2     if(st->top==0)
3       {  printf("\t\t 栈中无元素");  return(0); }        /* 若栈空则返回 0 */
4     else
5       {  st->top--;                                     /* 修改栈顶指针 */
6          x=st->s[st->top];                              /* 取栈顶元素 */
```

```
7        return(x);                    /* 返回 x 即栈顶元素的值 */
8     }
9  }
```

注意：此算法中元素并未出栈。

思考

(1) 该算法执行后栈顶元素是否在数组中？

(2) 如何用算法 3.3 和算法 3.4 实现显示栈中的所有元素 ShowStack(Stack * s)？

(3) 两个栈合用一个存储空间比一个栈单用一个存储空间有什么样的优势？

3.2.2　栈的链式表示

若是栈中元素的数目变化范围较大或不清楚栈中元素的数目，就应该考虑使用链式存储结构。将用链式存储结构表示的栈称为“链栈”，链栈对应于链表。链栈通常用一个无头结点的单链表表示。由于栈的插入删除操作只能在栈顶进行，而对于单链表来说，在首端插入和删除结点要比尾端相对地容易一些，因此将单链表的首端作为栈顶，即将单链表的头指针作为栈顶指针。

1. 链栈的实现

可以采用链表作为栈的存储结构，其结点类型与单链表的结点类型相同。

链栈可以用 C 语言描述如下：

```
typedef   struct node
{   DataType data;
    struct  node  * link;
}Node;
Node * top;                          /* 定义 top 指针指向链栈的栈顶结点 */
```

2. 链栈操作的示意图

链栈操作示意图如图 3.4～图 3.6 所示。

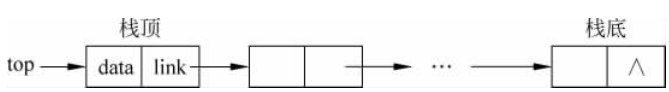


图 3.4　链栈示意图

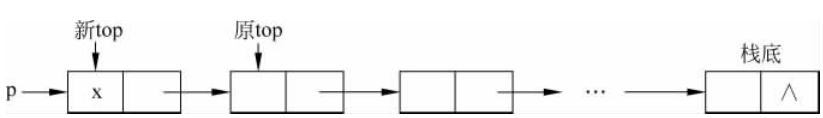


图 3.5　链栈的入栈操作

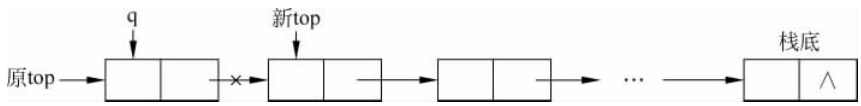


图 3.6 链栈的出栈操作

思考

链栈为什么不像线性链表那样需要一个人为的头结点?

3. 链栈操作的算法

1) 进栈

【算法 3.5】 链栈的进栈算法。

```
    void Push(Node * top,  DataType x)
1   {  Node * p;
2      p = (Node * )malloc(sizeof(Node));  /* 申请一个新的结点,并让指针 p 指向该结点 */
3      p -> data = x;                      /* 新结点的数据域被赋值为 x */
4      p -> link = top;                    /* 新结点的指针域赋值 */
5      top = p;                            /* 修改栈顶指针变量 */
6   }
```

思考

(1) 新插入结点和原来栈顶结点的逻辑关系是怎样的? 通过哪个语句体现的?

(2) 为什么要修改栈顶指针变量 top=p?

2) 出栈

【算法 3.6】 链栈的出栈算法。

```
    void  Pop (Node * top)
1   {  Node * q;
2      DataType  x;
3      if(top!= NULL)
4      {   q = top;                    /* q 指针指向原栈顶位置 */
5          x = q -> data;              /* x 用来保存原栈顶元素 */
6          top = top -> link;          /* 修改栈顶指针的位置 */
7          free(q);                    /* 回收结点 q */
8      }
9   }
```

思考

(1) 如果出栈时需要读出栈顶元素,如何修改算法?

(2) 如果出栈算法需要返回值为栈顶元素,如何修改算法?

3）读栈顶元素。

【算法 3.7】 读链栈栈顶元素的算法。

```
    DataType ReadTop(Node * top)
1   {  DataType  x;
2      if(top!= NULL)
3      {     x = top -> data;                /* x 用来保存原栈顶元素 */
4            return x;
5      }
6      else
7      {     printf("空栈\n");
8            exit( -1);
9       }
    }
```

思考

（1）栈顶指针是否发生了变化？

（2）如何用算法 3.7 实现算法 3.6 中的第一个思考问题？

（3）用算法 3.6 和算法 3.7 如何实现显示栈中的所有元素算法 ShowStack(Stack * s)？

3.3 栈的应用

3.3.1 数制转换

数值进位制的换算是计算机实现计算和处理的基本问题。例如，将十进制数 N 转换为 j 进制的数，其解决的方法很多，其中一个常用的算法是除 j 取余法（见图 3.7）。将十进制数每次除以 j，所得的余数依次进栈，然后按“后进先出”的次序便得到转换的结果。除 j 取余法的原理可以从下式看出：

$$(a_n a_{n-1} \cdots a_1)_j = a_n \times j^{n-1} + a_{n-2} \times j^{n-2} + \cdots + a_1 \times j^0 \tag{3-1}$$

其中 j 表示进位制的大小。从式(3-1)可知有下式成立：

$$N = (N/j) * j + N\%j \tag{3-2}$$

其中，/为整除，%为求余。

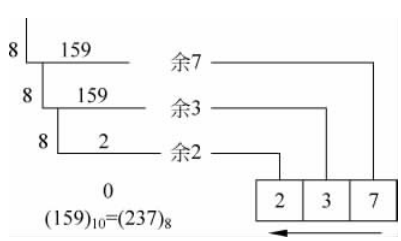


图 3.7 除 j 取余法

1. 算法思想

(1) 若 $N!=0$,则将 $N\%j$ 取得的余数压入栈 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/j 代替 N。

(3) 当 $N>0$,则重复步骤(1)、(2)。

2. 算法的实现

【算法 3.8】 十进制数转换算法(用顺序栈实现,进制转换中栈的变化情况如图 3.8 所示)。

```
    void  Conversion( )            /*将十进制数转换为八进制数*/
1   {  int x;
2      Stack *Dstack;
3      Dstack = InitStack();
4      scanf("%d",&x);
5      while(x!= 0)
6     { Push(Dstack,  x%8);        /*进栈次序是个位、十位……*/
7       x = x/8;
8      }
9      while(!IsEmpty(Dstack))
10    { Pop(Dstack,x);             /*先出栈的是高位,最后是个位*/
11      printf("%d",x);
12    }
13  }
```

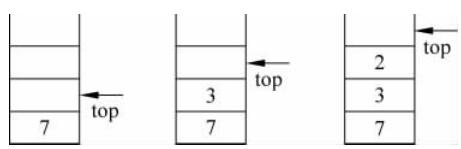


图 3.8 进制转换中栈的变化情况

思考

(1) 如何用链栈实现十进制数转换算法?

(2) 算法中使用了栈的哪些基本操作,这样做的好处是什么?

【例 3.1】 把十进制数 159 转换成八进制数(即 $N=159$, $j=8$)。

思考

(1) 试写出十进制转换为十六进制的算法。

(2) 使用链栈如何实现算法 3.8?

3.3.2 表达式求值

表达式是由运算对象、运算符、括号等组成的有意义的式子。表达式求值是程序设计语言编译中的一个基本问题。它的实现是栈应用的一个典型例子。

1. 中缀表达式

一般所用的表达式是将运算符号放在两运算对象的中间，如 $a+b$、c/d 等，把这样的式子称为中缀表达式。中缀表达式在运算中存在运算符号的优先权与结合性等问题，还存在括号优先处理的问题。

首先，要了解算术四则运算的规则：

(1) 先乘除，后加减；

(2) 从左到右进行计算；

(3) 先括号内，后括号外，多层括号，由内向外进行运算。

例如，要对表达式 $2+4\times3-10/2$ 求值，则计算顺序应为：

$$2+4\times3-10/2=2+12-10/2=2+12-5=14-5=9$$

任何一个表达式都是由操作数(operand)、运算符(operator)和界限符(delimiter)组成的，把运算符和界限符统称为算符。根据上述三条运算规则，在运算的每一步中，任意两个先后相继出现的算符 θ_1 和 θ_2 之间的优先关系有以下 3 种情况。

$\theta_1 < \theta_2$　θ_1 的优先权低于 θ_2

$\theta_1 = \theta_2$　θ_1 的优先权等于 θ_2

$\theta_1 > \theta_2$　θ_1 的优先权高于 θ_2

表 3.1 定义了算符之间的这种优先关系。

表 3.1　算符间的优先关系

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

例如，带阴影的单元就表示当相继出现两个“+”运算符时，第一个“+”的优先级比第二个“+”的优先级要高，换句话说就是第一个“+”优先进行运算。

“#”是表达式的结束符，为了算法简洁，在表达式的最左边也虚设一个“#”构成整个表达式的一对括号，“#”=‘#’表示整个表达式求值完毕。表中的“(”=“)”表示当左右括号相遇时，括号内的运算已经完成。

中缀表达式的求值采用“算符优先法”。为了实现算符优先算法，可以使用两个工作栈，一个称为 OPTR，用来存放运算符；另一个称为 OPND，用以存放操作数或运算结果。算法

的基本思想如下。

(1) 设置操作数栈为空栈,表达式起始符“#”为运算符栈(OPTR)的栈顶元素。

(2) 依次读入表达式中每个字符,若是操作数则进 OPND 栈,若是运算符(表中的 θ_2),则和 OPTR 栈的栈顶运算符(表中的 θ_1,也就是先入栈的运算符)比较优先权。如果 $\theta_1 < \theta_2$,则 θ_2 入 OPTR 栈;如果 $\theta_1 = \theta_2$,则 θ_1 出栈,而 θ_2 舍弃;如果 $\theta_1 > \theta_2$,则 θ_1 出栈与 OPND 栈中连续出栈的两个操作数进行运算,将运算得到的结果入 OPND 栈。然后将 θ_2 与 OPTR 栈顶的运算符再进行优先级比较,而进行相应的运算,直至整个表达式求值完毕。

【算法 3.9】 运算符比较算法。

```
    Char  Precede(char c1, char c2)
1  { char  optr1, optr2;
2    switch (c1)
3        {case '*':
4         case '/': optr1 = 4;break;
5         case '+':
6         case '-': optr1 = 2;break;
7         case '(': optr1 = 0;break;
8         case ')': optr1 = 5;break;
9         case '#': optr1 = -1;
10        }
11    switch (c2)
12        {case '*':
13         case '/': optr1 = 3;break;
14         case '+':
15         case '-': optr1 = 1;break;
16         case '(': optr1 = 5;break;
17         case ')': optr1 = 0;break;
18         case '#': optr1 = -1;
19        }
20    if(optr1 < optr2) return('<');
21    if(optr1 = optr2) return('=');
22    if(optr1 > optr2) return('>');
23 }
```

【算法 3.10】 表达式求值算法。

```
    int  Express( )
1  { char theta,  c;                    /* 用来表示运算符 */
2    Stack  OPTR; Push (&OPTR,'#');     /* 运算符栈 */
3    Stack  OPND;                       /* 运算数栈 */
4    c = getchar( );
5    while(c!= '#'|| ReadTop (&OPTR)!= '#')
6    /* '#'是表达式结束的标记,也是运算符栈为空的标记 */
7    { if(!In(c,op))
8     { Push(&OPND,c); c = getchar( );}   /* 不是运算符则进操作数栈 */
9     else                               /* 判断运算符栈顶元素和新输入运算符的优先权 */
10     switch(Precede(ReadTop(&OPTR),c))
```

```
11      {case '<':                                  /*栈顶优先权低*/
12         Push(&OPTR,c);     c=getchar( );         /*运算符进栈*/
13         break;
14      case '=':
15          Pop(&OPTR,&x);     c=getchar( );       /*运算符栈顶元素出栈*/
16           Break;
17       case '>':
18   /*将操作数栈顶两个元素和运算符栈顶的一个元素退栈,并结合进行运算,然后将结果入操作
     数栈*/
19          Pop(&OPTR,&theta);
20          Pop(&OPND,&b);     Pop(&OPND,&a);
21          Push(&OPND,Operate(a,theta,b));
22          break;
23       }
24    }
25    return ReadTop(&OPND);
26   }
```

【例 3.2】 计算 2×(3+9)/4−5。

栈的变化过程如表 3.2 所示。

表 3.2　中缀表达式求值中栈的变化情况

步骤	OPTR 栈	OPND 栈	输入字符	主要操作
1			#2*(3+9)/4−5#	Push(&OPTR,'#')
2	#	2	*(3+9)/4−5#	Push(&OPND,'2')
3	#	2	*(3+9)/4−5#	Push(&OPTR,'*')
4	# *	2	(3+9)/4−5#	Push(&OPTR,'(')
5	# * (	2	3+9)/4−5#	Push(&OPND,'3')
6	# * (	2 3	+9)/4−5#	Push(&OPTR,'+')
7	# * (+	2 3	9)/4−5#	Push(&OPND,'9')
8	# * (+	2 3 9	)/4−5#	Operate(3,'+',9)
9	# * (	2 12	/4−5#	Pop(&OPTR,theta){删除一对括号}
10	# *	2 12	/4−5#	Operate(2,'*',12)
11	#	24	4−5#	Push(&OPTR,'/')
12	# /	24	4−5#	Push(&OPND,'4')
13	# /	24 4	−5#	Operate(24,'/',4)
14	#	6	5#	Push(&OPTR,'−')
15	# −	6	5#	Push(&OPND,'5')
16	# −	6 5	#	Operate(6,'−',5)
17	#	1		return ReadTop(&OPND)

2. 中缀表达式转换成后缀表达式

将中缀表达式转换成为后缀表达式的过程也是一个栈应用的典型例子,将中缀表达式转换成后缀表达式,可以简化求值过程。

从中缀表达式“2*(3+9)/4−5”和其后缀表达式“239+*4/5−”可以看出,在这两种

表达形式中,操作数的次序是相同的。因此,顺序扫描中缀表达式,将它转换成后缀表达式的过程中,只要遇到操作数就可以直接输出,若是运算符(表中的 θ_2),则和 OPTR 栈的栈顶运算符(表中的 θ_1)比较优先权。如果 $\theta_1<\theta_2$,则 θ_2 入 OPTR 栈;如果 $\theta_1=\theta_2$,则 θ_1 出栈,而 θ_2 舍弃;如果 $\theta_1>\theta_2$,则输出 θ_1 出栈。然后将 θ_2 与 OPTR 栈顶的运算符再进行优先级比较,而进行相应的运算,直至整个表达式求值完毕。

【例 3.3】 将中缀表达式"2*(3+9)/4-5"转换成后缀表达式"239+*4/5-"的过程如表 3.3 表示。

表 3.3 中缀表达式转换成后缀表达式过程中栈的变化情况

步骤	OPTR 栈	输入字符	主要操作
1		#2*(3+9)/4-5#	Push(&OPTR,'#')
2	#	2*(3+9)/4-5#	输出'2'
3	#	*(3+9)/4-5#	Push(&OPTR,'*')
4	# *	(3+9)/4-5#	Push(&OPTR,'(')
5	# * (	3+9)/4-5#	输出'3'
6	# * (	+9)/4-5#	Push(&OPTR,'+')
7	# * (+	9)/4-5#	输出'9'
8	# * (+	)/4-5#	输出'+'
9	# * (	/4-5#	Pop(&OPTR,&theta){删除一对括号}
10	# *	/4-5#	输出'*'
11	#	4-5#	Push(&OPTR,'/')
12	# /	4-5#	输出'4'
13	# /	-5#	输出'/'
14	#	5#	Push(&OPTR,'-')
15	# -	5#	输出'5'
16	# -	#	输出'-'
17	#		

3. 后缀表达式

后缀表达式求值的运算只用一个栈便可实现,这是因为后缀表达式中既无括号又无优先级的约束。

后缀表达式求值的步骤如下。

(1) 读入表达式一个字符。

(2) 若是操作数,压入栈,转向步骤(4)。

(3) 若是运算符,从栈中弹出两个数,结合进行运算,并将运算结果再压入栈。

(4) 若表达式输入完毕,栈顶即表达式的值;若表达式未输入完,则转向步骤(1)。

【例 3.4】 计算 2×(3+9)/4-5。

先转化为后缀表达式,其后缀表达式为:239+*4/5-。

后缀表达式求值过程中栈的变化过程如表 3.4 所示。

表 3.4 后缀表达式求值中栈的变化情况

步骤	OPND 栈	输入字符	主要操作
1		239+*4/5−	Push(&OPND,'2')
2	2	39+*4/5−	Push(&OPND,'3')
3	2 3	9+*4/5−	Push(&OPND,'9')
4	2 3 9	+*4/5−	Operate(3,'+',9)
5	2 12	*4/5−	Operate(2,'*',12)
6	2 4	4/5−	Push(&OPND,'4')
7	2 4 4	/5−	Operate(24,'/',4)
8	6	5−	Push(&OPND,'5')
9	6 5	−	Operate(6,'−',5)
10	1		return ReadTop(&OPND)

表 3.3 和表 3.4 有何异同？

3.3.3 栈与递归

1. 函数的嵌套调用

在高级语言编制的程序中，调用子程序和被调用子程序之间的链接和信息交换需要通过一种特殊的栈来进行，这个栈就是系统栈。

通常，当在一个函数的运行期间调用另一个函数时，在运行被调用函数之前，系统需先完成三件工作：①将所有的实参、返回地址等信息传递给被调用函数保存；②为被调用函数的局部变量分配存储区；③将控制转移到被调用函数的入口。而从被调用函数返回调用函数之前，系统也应完成三件工作：①保存被调函数的计算结果；②释放被调函数的数据区；③依照被调用函数保存的返回地址将控制转移到调用函数。当有多个函数构成嵌套调用时，按照“最后调用最先返回”的原则，系统运行期间所需要的数据依次存放在系统栈中。当调用一个子程序时，就创建一个新的活动记录(或栈帧结构)，并通过入栈将其压入栈顶；每当从一个函数退出时，就通过出栈删除栈顶活动记录。

【例 3.5】 函数的嵌套调用。

```
main( )
{…
     R ( );
  …
}
R( )
{…
     S( );
…
```

```
    return;
}
S( )
{…
     K( );
…
     return;
}
K( )
{…
     return;
}
```

程序中栈的变化情况如图 3.9 所示。

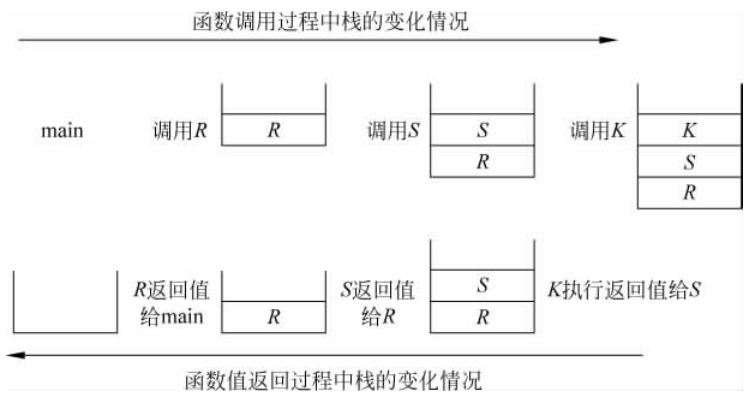


图 3.9 函数嵌套调用过程中栈的变化情况

2. 递归调用

函数直接或间接调用自身称为递归调用。实现过程类似函数的嵌套调用,建立一个递归工作栈。

【例 3.6】 函数的递归调用。

```
    void  write(int  w)
1  {int  i;
2  if ( w!= 0)
3    {write(w - 1);
4     for(i = 1;i < = w;++i)
5        printf(" % 3d",w);
6     printf("\n");
7    }
8  }
   main( )
   {write(3);}
```

程序运行结果：

```
1
2  2
3  3  3
```

以上程序中栈的变化情况如表 3.5 所示。由于例 3.6 所示的递归函数只含有一个参数，则每个工作记录包含两个数据项：返回地址和一个实在参数，并以递归函数中的语句行号表示返回地址，同时假设主函数的返回地址为 0。

表 3.5 递归函数运行的变化情况

递归运行的层次	递归工作栈状态（返址，w 值）	运行语句行号	说　明
1	top → 0, 3	1,2,3	主函数调用 write(3)，进入第一层递归后，运行至语句(行)3
2	top → 4, 2 / 0, 3	1,2,3	调用 write(2)，由第一层的语句(行)3 进入第二层递归，执行至语句(行)3
3	top → 4, 1 / 4, 2 / 0, 3	1,2,3	调用 write(1)，由第二层的语句(行)3 进入第三层递归，执行至语句(行)3
4	top → 4, 0 / 4, 1 / 4, 2 / 0, 3	1,2,8	调用 write(0)，由第三层的语句(行)3 进入第四层递归，执行至语句(行)2。从语句(行)8 退出第四层递归，返回至第三层的语句(行)4
3	top → 4, 1 / 4, 2 / 0, 3	4,5,6,7,8	此时 $w=1$，由条件输出语句输出一个 1。从语句(行)8 退出第三层递归，返回至第二层的语句(行)4
2	top → 4, 2 / 0, 3	4,5,6,7,8	此时 $w=2$，由条件输出语句输出两个 2。执行至语句(行)8 后退出第二层递归，返回至第一层的语句(行)4
1	top → 0, 3	4,5,6,7,8	此时 $w=3$，由条件输出语句中输出 3 个 3。执行至语句(行)8 后退出第一层递归，返回至主函数
0	栈空		继续运行主函数

递归法是描述算法的一种强有力的方法,其思想是:将 $N=n$ 时不能得出解的问题,设法将递归转化成为求 $n-1,n-2,n-3,\cdots$ 的问题,一直到 $N=0$ 或 $N=1$ 的初始情况,由于初始情况的解可以给出或方便得到,因此开始层层退栈得到 $N=2,3,\cdots,n$ 时的解,最终得到 $N=n$ 时问题的解。用递归法写出的程序结构清晰、简单易读,而且程序的正确性容易得到证明。但递归的次数受到计算机内存(或特定程序设计语言)的限制,因为每一次的递归函数调用都要压栈占用内存。值得注意的是递归思想贯穿在教材的各个章节中,如链表的定义、链表的建立、树和二叉树的定义和遍历、快速排序和堆排序等。递归算法具有简洁性的优点,但是如何将一个具体的问题抽象出递归关系却是个挑战。

3. 着色问题

【例 3.7】 有一张地图如图 3.2 所示,试用最少的颜色对该地图进行着色(每块区域只涂一种颜色),且要求相邻的颜色互不相同,请打印出所有可能的着色方案。

定义相邻关系矩阵(仅为方便起见,还可采取更好的节省存储空间的方法)(d_{ij}),$d_{ij}=0$ 表示第 i 区与第 j 区相邻,而 $d_{ij}=1$,则表示第 i 区与第 j 区不相邻。根据图 3.2,可写出相邻关系图 3.2 对应的相邻关系矩阵,如图 3.10 所示。

$$\begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0\\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0\\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0\\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0\\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0\\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1\\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1\\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1\\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1\\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0
\end{pmatrix}$$

图 3.10　图 3.2 地图对应的相邻矩阵

下面给出递归算法:

```
    int isAvailable(int color,int area,int dist[N][N],int s[])     //判断相邻地区重色
1   /*color 代表颜色,area 表示当前要染色的是第几个区域,
2   k 表示已经染色区域的颜色*/
3   {
4     int i=0;
5     while((i<area)&&(s[i]*dist[area][i]!=color))                  //判断是否重色
6     i++;
7     if(i<area)
8        return FALSE;
9     else
10       return TRUE;
11  }
```

```
12  void show(int s[])
13  {
14     int i;
15     for(i = 0;i < N;i++)
16       printf(" % d ",s[i]);
17     printf("\n");
18  }
19  void mapcolor(int s[],int dist[][N],int area,int color)     //利用递归来进行颜色试探
20  {
21     int i;
22     for(i = 1;i < color + 1;i++)                              //for 循环用于颜色试探
23     {
24           if(isAvailable(i,area,dist,s) == TRUE)              //判断重色
25              {
26                   s[area] = i;                               //如果不重色,给地图上色
27                   if(area == N - 1) show(s);                 //地图上完颜色,打印
28                   else mapcolor(s,dist,area + 1,color);      //递归,下一块区域上色
29                   s[area] = 0;
30              }
31       }
32  }
```

(1) 递归程序的缺点是什么？用程序测试你的计算机能够递归的层数。

(2) 如何把下面这个问题“聪明的学生”用递归表示求解？

一位教逻辑学的教授有三名非常善于推理且精于心算的学生 A、B 和 C。有一天，教授给他们 3 个出了一道题：教授在每个人脑门上贴了一张纸条并告诉他们，每个人的纸条上都写了一个正整数，且某两个数的和等于第三个。于是，每个学生都能看见贴在另外两个同学头上的整数，却看不见自己的数。

这时，教授先对学生 A 发问了：“你能猜出自己的数吗？”A 回答：“不能。”

教授又转身问学生 B：“你能猜出自己的数吗？”B 想了想，也回答：“不能。”

教授再问学生 C 同样的问题，C 思考了片刻后，摇了摇头：“不能。”

接着，教授又重新问 A 同样的问题，再问 B 和 C，经过若干轮后，当教授再次问某人时，此人露出了得意的笑容，把自己头上的数准确地说了出来。请问，已知 A、B 和 C 头上贴的数为 X_1、X_2、X_3，求教授至少需提问多少次，轮到回答问题的那个人才能猜出自己头上的数。

3.3.4　回溯法

回溯法又称为试探法，是寻找问题一个解或所有解的一种搜索策略。使用回溯法时，在使用某种方法寻找解的过程中，若中间项结果满足所解问题的条件，则一直沿这个方向搜索下去，直到无路可走或得到问题的一种解。若无路可走，则开始回溯，改变其前一项的方向(或值)继续搜索。若其上一项(或值)都已经测试过，还是无路可走，则在继续回溯到更前一

项,改变其方向(或值)继续搜索。若找到了一个符合条件的解,则停止或输出这个结果继续搜索;否则继续回溯下去,直到回溯到问题的开始处(不能再回溯),仍没有找到符合条件的解,则表示此问题无解或已经找到了全部的解。

使用回溯法求某个问题的全部解时,要注意在找到一组解时,将其及时输出或记录下来并统计解的个数。

【例 3.8】 使用回溯法来求解 3.1.3 节中的着色问题,图的定义如图 3.10 所示。

解题思路:要打印所有可能的着色方案,需要使用回溯法。

从第 1 号区域开始按照区域的编号顺序着色,在给每个区域选择着色时,都要检查该颜色号是否与其相邻区域的颜色号相同,只有颜色号与其相邻区域的颜色号均不相同时,该颜色号才能被选择。

用一维数组 $c[4]=\{1,2,3,4\}$来存储颜色,用一维数组 $e[12]$来表示着色方案,$e[n]$的值表示第 n 个区域的颜色号;G 表示着色方案的总数;M 表示一种着色方案用到的颜色的种数。

从第 1 号区域开始按照区域的编号顺序着色,在给每个区域选择着色时,都要检查该颜色号是否与其相邻区域的颜色号相同,只有颜色号与其相邻区域的颜色号均不相同时,该颜色号才能被选择。

算法描述:

```
输入:地图上需要着色区域的总数 N、地图对应的相邻关系矩阵 d、存储颜色号的数组 c。
初始化颜色栈:申请长度为 N 的一维数组 e 存放所求的着色方案。
令 G = 0, i = 1;
For (i = 1;i <= N;i++)
{
     按照颜色序号较小优先选择的原则,选择对区域 i 的合法着色 c[j];
     Push(e, c[j]);
}
ShowStack(e);                                  /* 输出结果就是一种着色方案 */
G = 1;
while(i > 0)
{    m = i;
     while(e[m]< 4)
     {
          如果存在颜色号大于 e[m]的合法颜色号 c[j]
          {
               Pop(e,&x);
               选择 c[j]为第 m 号区域新的着色;
               Push(e, c[j]);
          }
          if(m < N)
          {
               For(   ;m <= N;m++)
               {
                    按照颜色序号较小优先选择的原则,
                      i. 选择对区域 i 的合法着色 c[j];
                    Push(e, c[j]);
               }
```

```
28            }
29            ShowStack(e)(输出结果就是一种着色方案);
30            G++;
31       }
32       while(m>=i)
33            pop(e,&x);
34       i--;
35 }
36 输出 G
```

3.4　队列的概念及基本运算

3.4.1　队列的概念

队列是线性表的一种，是一种先进先出(First In First Out，FIFO)的线性表。队列限制在表的一端可进行插入而只能在另一端进行删除操作。跟排队购票一样，能够插入元素的一端称为**队尾**(rear)，允许删除元素的一端称为**队首**(front)。设有 n 个元素的队列 $Q=(a_1,a_2,a_3,\cdots,a_n)$，则称 a_1 为队首元素，a_n 为队尾元素。队列中的元素按 $a_1,a_2,a_3,a_4,\cdots,a_{n-1},a_n$ 的次序进队，按 $a_1,a_2,a_3,\cdots,a_{n-1},a_n$ 次序出队，即队列的操作是按照“**先进先出**”原则进行的，简称 FIFO 表，如图 3.11 所示。与线性表相类似，队列也有顺序存储和链式存储两种存储结构。

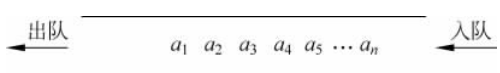


图 3.11　队列示意图

队列的实例如下。

(1) 在计算机处理文件打印时，为了解决高速的 CPU 与低速的打印机之间的矛盾，将多个打印请求存储在缓存中，按照它们提出请求的时间而顺序执行各个打印任务，即按照“先进先出”的原则形成打印队列。

(2) 现代制造业中的传送带上的产品也构成队列。

(3) 现代银行服务采用计算机同时处理多个窗口的多个队列[①]，以方便顾客和提高银行工作效率。

3.4.2　队列的基本运算

在队列上进行的运算如下。

(1) 队列初始化：InitQue(Q)。

初始条件：队列 Q 不存在。

① 详见严蔚敏、吴伟民编著的《数据结构》(C 语言版)，清华大学出版社。

运算结果：创建一个空队列。

(2) 入队操作：InsertQue(Q,x)。

初始条件：队列 Q 存在,但未满。

运算结果：插入一个元素 x 到队尾,长度加 1。

(3) 出队操作：ExitQue(Q)。

初始条件：队列 Q 存在,且非空。

运算结果：删除队首元素,长度减 1,需要时可获取队头元素。

(4) 判队空操作：EmptyQue(Q)。

初始条件：队列 Q 存在。

运算结果：若队列空则返回为 1,否则返回为 0。

(5) 求队列长度：LenQue(Q)。

初始条件：队列 Q 存在。

运算结果：返回队列的长度。

3.4.3 一个有趣的问题

队列在模拟排队一类的问题中用途非常广泛,无冲突分组问题就是其中的一个。例如,设有一个旅游团由 n 个人组成,这 n 个人中有的互有嫌隙。为了在旅游中彼此不发生冲突,应将人员分组。试问应如何才能使组数最少且任何互有嫌隙的人不被分在同一组中？这个问题的另外一种描述为：一个美食家急于吃遍某大饭店的 n 道菜,但这 n 道菜中却有不适于在同一餐中吃的,否则可能降低菜的营养价值。那么该人应如何点菜,才能尽快吃遍这 n 道菜又不降低菜的营养价值？

这个问题的抽象表示是：已知集合 $S=\{s_1, s_2, s_3, \cdots, s_n\}$ 和定义在 S 上的关系 $R=\{(s_i,s_j)\mid s_i, s_j\in S, 1\leqslant i,j\leqslant n$ 且 $i\neq j\}$,$(s_i,s_j)\in R$ 表示 s_i 与 s_j 有冲突。现在要求将 S 划分成若干个不相交的子集,使得子集数量最少且任何一个子集中的任何两个元素互不冲突。

例如,设 $S=\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\}$,$R=\{(s_2,s_8),(s_2,s_9),(s_2,s_6),(s_2,s_5),(s_2,s_1),(s_3,s_6),(s_3,s_7),(s_4,s_5),(s_4,s_9),(s_5,s_6),(s_5,s_7),(s_5,s_9),(s_6,s_7)\}$,则问题的一个解为$\{S_1, S_3, S_4, S_8\},\{S_2,S_7\},\{S_5\},\{S_6, S_9\}$。

3.5 队列的存储结构及运算

3.5.1 队列的顺序表示

顺序队列是按照队列中的数据元素的顺序将数据元素存放在一组连续的内存中,可以用一维数组 Q[MAXLEN]作为队列的顺序存储空间,其中 MAXLEN 为队列的容量,队列元素从 Q[1]单元开始存放,直到 Q[MAXLEN]单元。因为队头和队尾都是活动的,因此,除了队列的数据以外,一般还要有两个整型变量标记队首(front)和队尾(rear)两个数据元素的位序,这两个整型变量被称为队列的头指针和尾指针。为了方便,通常规定头指针 front 总是指向队列当前的队首元素的前一个位置(这个位置的地址编号更小),尾指针 rear

总是指向队列当前的队尾元素。

顺序队列用 C 语言定义如下：

```
typedef  struct
{ DataType *  Q;                          /* 存储队列元素的存储块的首地址 */
  int  front = -1, rear = -1;             /* 指示队头、队尾元素的位置,并置队列为空 */
}SeqQue;
SeqQue *  sp;                             /* 定义一个指向顺序队列的指针变量 */
sp = (Seque * ) malloc(MAXLEN * sizeof(SeqQue));      /* 申请一个顺序队列的存储空间 */
```

1. 入队操作

在无溢出的情况下，入队时队尾指针加 1，元素入队。操作如下：

```
sp->rear++;                               /* 先将队尾指针加 1 */
sp->Q[sp->rear] = x;                      /* x 入队 */
```

2. 出队操作

在队非空的情况下，出队时队头指针加 1，队头元素即可出队。操作如下：

```
sp->front++;
x = sp->Q[sp->front];                     /* 队头元素送 x */
```

求队列的长度：

队中元素的个数：$m=(sp->rear)-(sp->front)$。

队满时：$m=MAXLEN$。

队空时：$m=0$。

设队列长度 MAXLEN=5，则其示意图如图 3.12 所示。

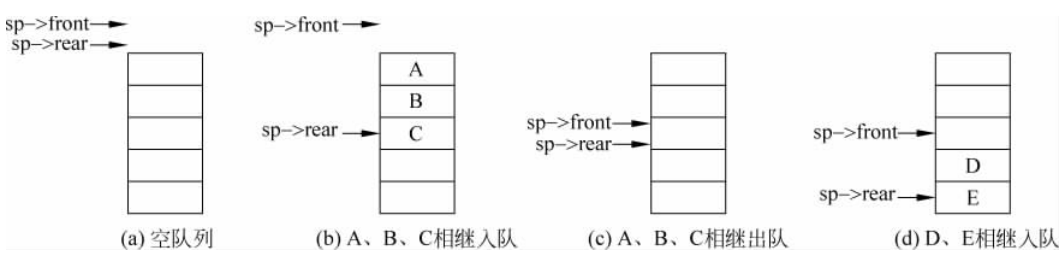


图 3.12　队列操作示意图

思考

(1) 为什么队列的长度不是(sp->rear)-(sp->front)+1?

(2) 从图 3.12(d)中可以看出，以上定义的这种队列有什么缺点?

从图 3.12 可以看到，随着入队、出队操作的不断进行，整个队列会整体向后移动，这样

就出现了图 3.12(d)中的现象：队尾指针已经移到了最后，而队列却未真满的“假溢出”现象，使得队列的空间没有得到有效的利用。解决的方法，可以将所有的数据往前移动，让空间留在队尾，这样新的数据就可以入队了。

(3) 食堂的排队是队列，为什么没有假溢出现象？

3.5.2 循环队列

为了解决上述队列的“假溢出”现象，要做移动操作，当移动数据较多时将会影响队列的操作速度。一个更有效的方法是将队列的数据区 Q[1：MAXLEN]假想成是首尾相连的环，即将 Q[1]与 Q[MAXLEN]假想成相邻的两个数组元素，形成一个环形表，这就成了循环队列，如图 3.13 所示。在循环队列中，仍旧规定头指针 front 总是指向队列当前的队首元素的前一个位置。

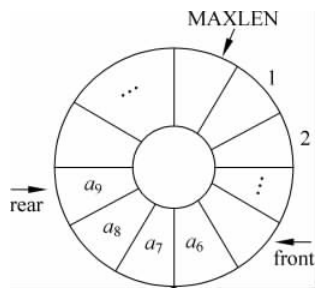


图 3.13 循环队列示意图

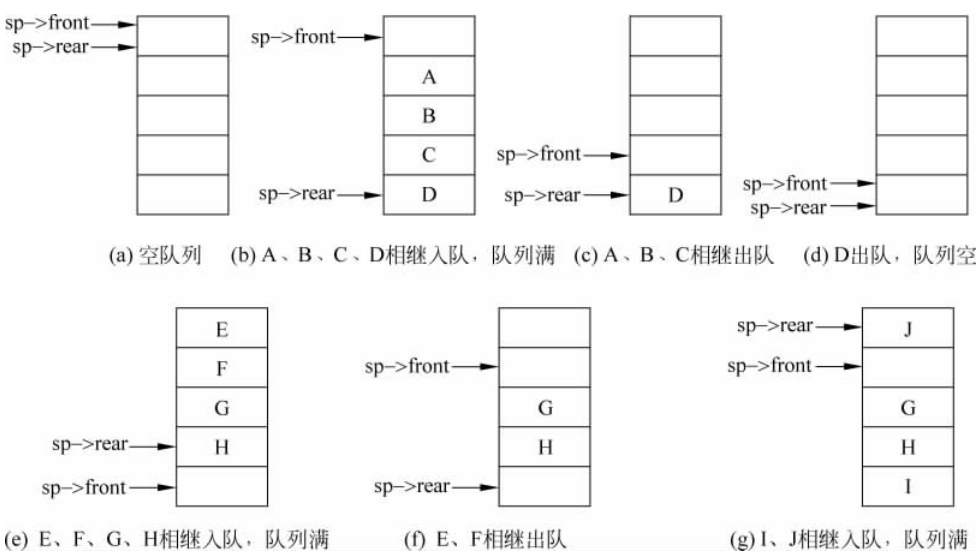
循环队列初始化时，定义 sp－> rear＝sp－> front＝1。因为是头尾相接的循环结构，入队操作修改为：

```
sp->rear = (sp->rear + 1) % MAXLEN;          /* 先将队尾指针加 1 */
sp->Q[sp->rear] = x;                         /* x 送队头元素中 */
```

出队操作修改为：

```
p->front = (p->front + 1) % MAXLEN;
x = sp->Q[sp->front];                        /* 队头元素送 x */
```

循环队列的入队和出队操作如图 3.14 所示。



(a) 空队列 (b) A、B、C、D相继入队，队列满 (c) A、B、C相继出队 (d) D出队，队列空

(e) E、F、G、H相继入队，队列满 (f) E、F相继出队 (g) I、J相继入队，队列满

图 3.14 循环队列操作示意图

从图 3.14 可以看出，循环队列的确可以解决“假溢出”问题，并且不需要移动数据元素。但是，循环队列还需要以空出一个存储空间的代价用于判断队列是否是“满”。如果不付出这样的一个存储空间，就无法判断队列是“满”还是“空”，这可以从对图 3.15 进行分析得到。

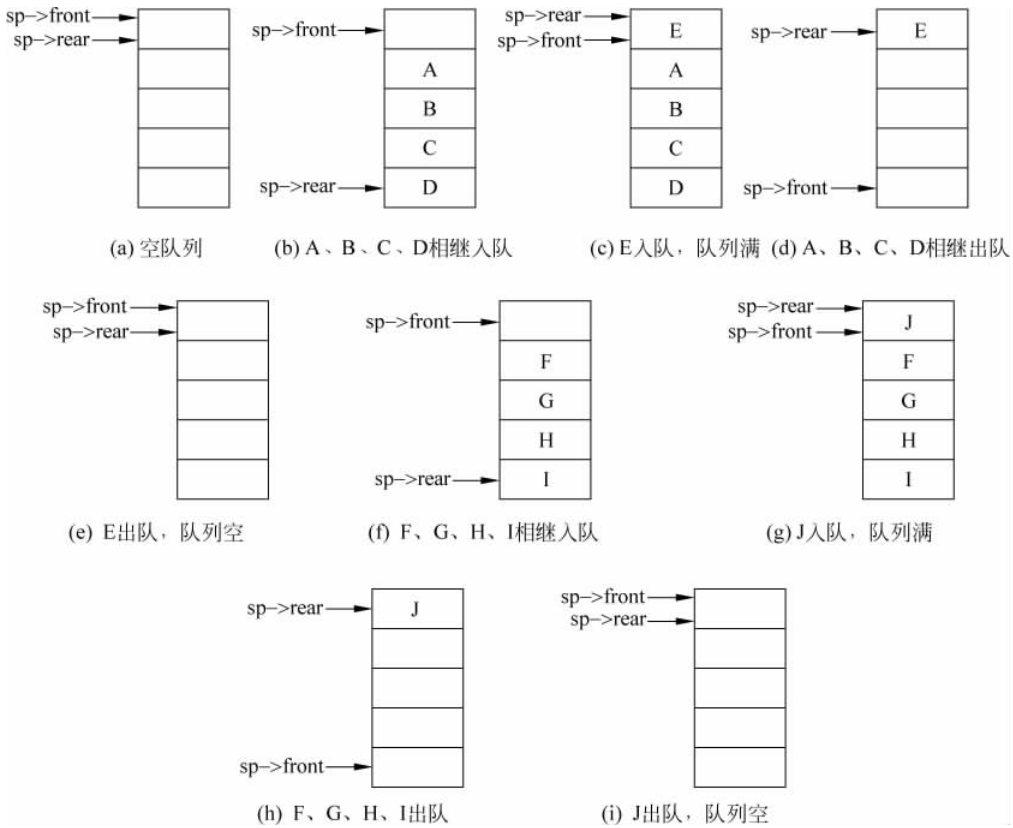


图 3.15　全部空间都用于存储数据元素时，循环队列操作示意图

从图 3.15 所示的循环队列可以看出：sp－>front==sp－>rear 可以判别队列空间是否是“空”，而(sp－>rear+1)% MAXLEN==sp－>front 可以判别队列空间是否是“满”。

从图 3.15 可以知道：当 sp－>rear 追上 sp－>front 时循环队列满，而当 sp－>front 追上 sp－>rear 时循环队列空。但是当 sp－>front==sp－>rear 时，无法判断队列是“空”还是“满”。要描述 sp－>rear 与 sp－>front 之间的这种“追赶”关系，那就必须要用一个变量来表述，因此为了便于判断循环队列是“空”还是“满”必须付出一个存储空间的代价。

循环队列类型定义如下：

```
typedef  struct
 {  DataType * Q;                    /* 数据的存储区 */
    int  front,rear;                 /* 队头、队尾指针 */
 }CycQue;                            /* 循环队列 */
```

循环队列的基本运算如下。

1. 置空队

【算法 3.11】 置空队示例如下：

```
    void  InitCycQue(CycQue * sp)
1   {  sp->Q=(DataType *)malloc(MAXLEN *sizeof(DataType));
2      sp->front=sp->rear=1;
3   }
```

思考

(1) 算法 3.11 中的第 1 行为什么要强制转换？

(2) 算法 3.11 中的第 2 行 sp—>front＝sp—>rear＝0 是否可以，对后面的算法有什么影响？

2. 入队算法

【算法 3.12】 入队算法示例如下：

```
    int  InsertCycQue(CycQue * sp,DataType x)
1   {    if ((sp->rear+1)%MAXLEN==sp->front)
2        {  printf ("队满");
3           return -1;                    /*队满不能入队*/
4        }
5        else
6        {  sp->rear=(sp->rear+1) % MAXLEN;
7           sp->Q[sp->rear]=x;
8           return 1;                     /*入队完成*/
9        }
10   }
```

思考

解释第 2 行判断循环队列满的含义。

3. 出队算法

【算法 3.13】 出队算法示例如下：

```
    int  ExitCycQue(CycQue * sp, DataType *  x)
1   {  if (sp->front==sp->rear)
2         {  printf ("队空");
3            return -1;                 /*队空不能出队*/
4         }
5     else
6       {    sp->front=(sp->front+1) % MAXLEN;
7            *x=sp->Q[sp->front];       /*读出队头元素*/
```

```
8            return 1;                    /* 出队完成 */
9        }
10   }
```

思考

(1) 算法 3.13 中的首行和第 7 行中的“*”号代表什么含义，含义是否相同？

(2) 在算法 3.13 中的首行和第 7 行中，如果不加“*”号是否可以？

4. 求队列长度

【算法 3.14】 求队列长度算法如下：

```
int  LenCycQue(CycQue * sp)
{  return (sp->rear - sp->front + MAXLEN) % MAXLEN;  }
```

思考

若用户无法估计所用队列的最大长度，该怎么办？

3.5.3　队列的链式表示

队列的链式存储称为链队列(或链队)。和链栈类似，链队列也可以用单链表来实现。根据“先进先出”原则，为了便于在队头删除和队尾插入，要分别为队列设置一个头指针和尾指针，队尾指针指向队列的最后一个元素。同样为了方便，给队列增加了一个头结点，队头指针就指向头结点，如图 3.16 和图 3.17 所示。

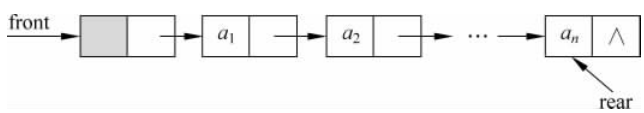


图 3.16　链队列示意图

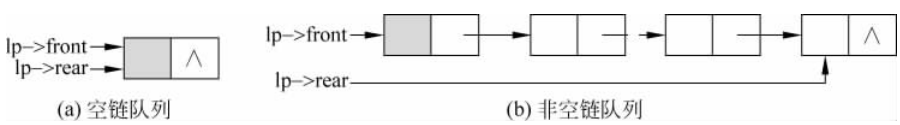


图 3.17　链队列

思考

在链队列与顺序队列中，头指针和尾指针的定义有何不同？

根据以上链队列的示意图，可以定义关于链队列结点和链队列的定义。

链队列结点的定义：

```
typedef  struct  node
     { DataType  data;                          /* 存储数据元素 */
       struct  node  next;                      /* 指向直接后继结点的指针 */
     } Quenode;
```

链队列的定义：

```
typedef  struct
     { Quenode   *front, *rear;                 /* 定义队列的头指针和尾指针 */
     }LinkQue;                                  /* 将头尾指针封装在一起的链队 */
LinkQue *  lp;                                  /* 定义一个指向链队的指针 lp */
```

队列为空时头指针和尾指针都指向头结点。

链队列的基本运算如下。

(1) 队列初始化,创建一个带头结点的空队列。

【算法 3.15】 创建空队列算法。

```
    void InitLQue (LinkQue* Lp)
1  {
2     Quenode*  Q;
3     Q = (Quenode* )malloc(sizeof(Quenode))      /* 申请链队头结点 */
4     lp->front = Q;                              /* 头指针指向头结点 */
5     lp->front->next = NULL;                     /* 置头结点指针域为空 */
6     lp->rear = lp->front;                       /* 尾指针指向头结点 */
7  }
```

思考

如果没有头结点,如何修改算法?

(2) 入队操作。

【算法 3.16】 入队算法。

```
    void InsertLQue (LinkQue *Lq ,  DataType x)
1  { Quenode *p;
2    p = (Quenode* )malloc (sizeof(Quenode));      /* 申请新结点 */
3    p->data = x;   p->next = NULL;
4    lq->rear->next = p;
5    lq->rear = p;
6  }
```

思考

如果没有头结点,如何修改入队算法,需要哪些特殊处理?

（3）判断队空操作。

【算法 3.17】 判断队空算法。

```
   int  EmptyLQue ( LinkQue *Lq)
1  { if (lq->front == Lq->rear)   return 0;
2    else  return 1;
3  }
```

思考

哪些情况需要判断队空操作？

（4）出队操作。

【算法 3.18】 出队算法。

```
    int ExitLQue (LinkQue* Lq ,  DataType*   x)
1  {  Quenode*q;
2     if (EmptyQue (Lq) )
3         {  printf ("队空"); return 0;
4         }                                          /*队空,出队失败*/
5     else
6         {  q=lq->front->next;
7            lq->front->next=lq->front->next->next;
8            *x=q->data;                             /*队头元素放x中*/
9            free(q);                                /*释放资源*/
10           return 1;
11        }
12 }
```

思考

（1）队列只有一个结点时，出队算法如何增加特殊处理？

（2）如果需要给出链队列的长度，怎样修改数据结构定义和各个算法？

（3）链队列有没有“假溢出”现象？

3.6 队列的应用

【例 3.9】 将二项式$(a+b)^n$展开，其系数构成杨辉[①]三角形，如图 3.18 所示。要求按行输出展开式系数的前 n 行。

从杨辉三角形的性质可以知道，除第 1 行以外，在打印第 i 行时，必须用到第 $i-1$ 行的数据。例如，在 $i=2$、3、4 时，每行的两侧各加上一个 0，如图 3.19 所示。设 s 是第 i 行第

① 杨辉是中国南宋时期杰出的数学家和数学教育家。在 13 世纪中叶活动于苏杭一带，其著作甚多。他著名的数学书共五种二十一卷，著有《详解九章算法》十二卷（1261 年）、《日用算法》二卷（1262 年）、《乘除通变本末》三卷（1274 年）、《田亩比类乘除算法》二卷（1275 年）、《续古摘奇算法》二卷（1275 年）。

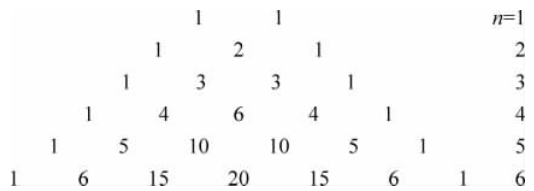


图 3.18 杨辉三角形

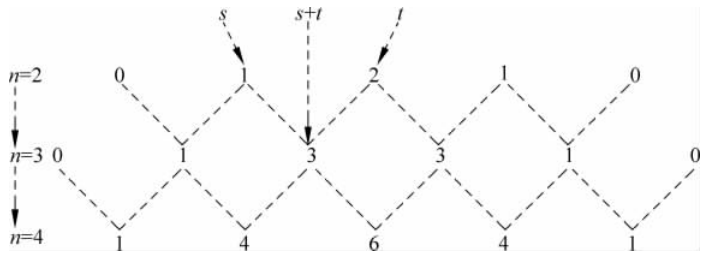


图 3.19 第 i 行元素与第 $i+1$ 行元素的关系

$j-1$ 个元素，t 是第 i 行第 j 个元素，则 $s+t$ 是第 $i+1$ 行的第 j 个元素。

设在数组 q 中已有第 i 行数据，且 $s=0$，则在第 i 行数据的后面再添加一个 0。这样，第 $i+1$ 行的数据可以通过第 i 行数据计算得到，而按照先算出先进入数组 q 的顺序将第 $i+1$ 行的数据依次入队进入数组 q，如图 3.20 所示，然后第 i 行的数据就依次出队。

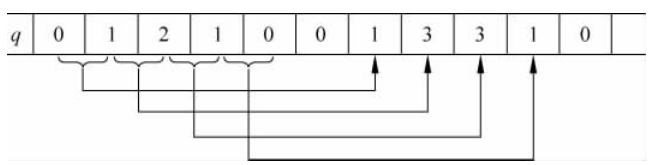


图 3.20 从第 2 行数据计算第 3 行数据

【算法 3.19】 杨辉三角形的算法如下：

```
    void YangHui(int n)
1   {int s, int t;
2   SeqQue   * q;
3   InitQue(q);
4   InsertQue(q,0);                          /* 在第 1 行的左侧添加 0 */
5   InsertQue(q,1), printf("1\t");
6   InsertQue(q,1) , printf("1\t");
7   InsertQue(q,0);                          /* 在第 1 行的右侧添加 0 */
8   for (int i = 2; i <= n; i++)             /* 计算第 2 行到第 n 行系数 */
9   {   InsertQue(q,0);                      /* 在第 i 行的左侧添加 0 */
10      printf("\n");
11      s = ExitQue(q);
12      for (int j = 1; j <= i + 1; j++)
13      {   t = ExitQue(q);
14          InsertQue(q,s + t);
```

```
15            if (j!= i + 1) printf(" % d\t",s + t);
16            s = t;
17          }
18          InsertQue(q,0);                    /* 在第 i 行的右侧添加 0 */
19        }
20        ExitQue(q);
21    }
```

思考

上述算法中所使用队列的容量如何确定？

【例 3.10】 某人急于吃遍某大饭店的 n 道菜，但这 n 道菜中却有不适于在同一餐中吃的，否则可能降低菜的营养价值。那么该人应如何点菜，才能尽快吃遍这 n 道菜又不降低菜的营养价值？这类问题的抽象表示是：已知集合 $S=\{s_1, s_2, s_3, \cdots, s_n\}$ 和定义在 S 上的关系 $R=\{(s_i,s_j) \mid s_i, s_j \in S, 1 \leqslant i, j \leqslant n$ 且 $i \neq j\}$，$(s_i,s_j) \in R$ 表示 s_i 与 s_j 有冲突。现在要求将 S 划分成若干个不相交的子集，使得子集数量最少且任何一个子集中的任何两个元素互不冲突。

解题思路： 将集合 S 的 n 个元素依序号放入一个队列中，并依次出队。若本次出队的元素与当前分组中的所有元素互不冲突，则将该元素放入当前组，否则重新入队。由于第一个再入队的元素的序号一定小于队尾元素的序号，因此当出队的元素的序号小于其前面出队元素的序号时，说明出队完成一个循环，形成一个分组。然后重开另一分组并重复上述过程直到队列空为止。

具体实现时，利用 n 阶二维数组来存放关系 R，若 $(s_i,s_j) \in R$，则令 $R[i][j]=R[j][i]=1$，否则 $R[i][j]=R[j][i]=0$，如图 3.21 所示。如果当前分组已有元素 $s_{i_1}, s_{i_2}, s_{i_3}, \cdots, s_{i_m}$ 时，要判断当前出栈 s_j 是否可加入当前分组，需要检查 s_j 与 $s_{i_1}, s_{i_2}, s_{i_3}, \cdots, s_{i_m}$ 各元素的关系，即检查 $R[j][i_1], R[j][i_2], R[j][i_3], \cdots, R[j][i_m]$ 是否全部为 0，如果是，则将 s_j 加入当前分组，否则不能加入当前分组。为了记录分组的结果，可设置一个有 n 个分量的一维数组 group，其第 i 个分量记录元素 s_i 的分组号。例如，有 group[5]=4 就表示第 5 个元素属于第 4 分组。以上例子的求解过程如表 3.6 所示。

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 3.21　关系矩阵

表 3.6　冲突分组的求解过程

当前分组	队列中剩余元素	出队元素	出队元素与当前分组中元素的冲突情况	所做的处理
{}	1,2,3,4,5,6,7,8,9		初始状态	1 号出队
{}	2,3,4,5,6,7,8,9	1	无冲突	1 号入当前组，2 号出队
{1}	3,4,5,6,7,8,9	2	2 号与 1 号冲突	2 号重新入队，3 号出队
{1}	4,5,6,7,8,9,2	3	3 号与当前组无冲突	3 号入当前组，4 号出队
{1,3}	5,6,7,8,9,2	4	4 号与当前组无冲突	4 号入当前组，5 号出队

续表

当前分组	队列中剩余元素	出队元素	出队元素与当前分组中元素的冲突情况	所做的处理
{1,3,4}	6,7,8,9,2	5	5号与4号冲突	5号重新入队,6号出队
{1,3,4}	7,8,9,2,5	6	6号与3号冲突	6号重新入队,7号出队
{1,3,4}	8,9,2,5,6	7	7号与3号冲突	7号重新入队,8号出队
{1,3,4}	9,2,5,6,7	8	8号与当前组无冲突	8号入当前组,9号出队
{1,3,4,8}	2,5,6,7	9	9号与4号冲突	9号重新入队,2号出队
{1,3,4,8}	5,6,7,9	2	2<9	本分组结束,另设一分组
{}	5,6,7,9		新的初始状态	2号入当前组,5号出队
{2}	6,7,9	5	5号与2号冲突	5号重新入队,6号出队
{2}	7,9,5	6	6号与2号冲突	6号重新入队,7号出队
{2}	9,5,6	7	7号与当前组无冲突	7号入当前组,9号出队
{2,7}	5,6	9	9号与2号冲突	9号重新入队,5号出队
{2,7}	6,9	5	5<9	本分组结束,另设一分组
{}	6,9		新的初始状态	5号入当前组,6号出队
{5}	9	6	6号与5号冲突	6号重新入队,9号出队
{5}	6	9	9号与5号冲突	9号重新入队,6号出队
{5}	9	6	6<9	本分组结束,另设一分组
{}	9		新的初始状态	6号入当前组,9号出队
{6}		9	9号与当前组无冲突	9号入当前组
{6,9}	队空			程序终止

根据表3.6中的求解过程,冲突分组算法的描述如下:

```
输入:元素数目n,关系矩阵R
初始化循环队列Q
for (i=1;i<=n;i++)
    InsertCycQue(&Q,i);
prior=1;
新建一空的分组;
while(!EmptyCycQue(&Q))
{   ExitCycQue(&Q,&j);
    if(j>prior)
    {   利用关系矩阵检测j号元素与当前分组中的元素是否有冲突;
        如果无冲突,则将j号元素归入当前分组;
            否则重新入队;
    }
    else
    {   新建另一空的分组;
        将j号元素归入该分组;
    }
}
输出:各元素的分组情况;
算法结束
```

习题 3

一、填空题

1. 向栈中压入元素的操作是先_____，后_____。

2. 设一个链栈的栈顶指针是 ls，栈中结点类型为 Node，则栈空的条件是_____。如果栈不为空，则退栈操作为 p=ls；_____；free(p)。

3. 栈是限制在表的一端进行插入和删除的线性表，插入、删除的这端称为_____，另一端称为_____。

4. 栈通常有_____和_____两种存储结构。

5. 在栈中存取数据遵循的原则是_____。

6. 在顺序栈 S 中，出栈操作时要执行的语句序列中有 S->top _____；进栈操作时要执行的语句序列中有 S->top _____。

7. 链栈中第一个结点代表栈的_____元素，最后一个结点代表栈的_____元素。

8. 在顺序栈中，假设栈所分配的最大空间为 MAXLEN，top=0 表示_____，此时再出栈会发生_____现象；top=MAXLEN 表示_____，此时再入栈就会发生_____现象。

9. 在高级语言编制的程序中，调用子程序和被调用子程序之间的链接和信息交换需要通过_____来进行。

10. 函数直接或间接调用自身称为_____。

11. 当有多个函数构成嵌套调用时，遵守_____的原则。

12. 2+4×(3+9)/(4-5)的后缀表达式为_____。

13. 在队列中存取数据应遵从的原则是_____。

14. FIFO 的含义是_____。

15. 在队列中，允许插入的一端称为_____，允许删除的一端称为_____。

16. 设长度为 n 的链队列用单循环链表表示，若只设头指针，则入队和出队操作的时间复杂度分别为_____和_____；若只设尾指针，则入队和出队操作的时间复杂度分别为_____和_____。

17. 在非空队列中，头指针指向_____，而尾指针指向_____。

18. 设采用顺序存储结构的循环队列头指针 front 指向队头元素，队尾指针 rear 指向队尾元素后的一个空闲元素，队列的最大空间为 Queuelen。

(1) 在循环队列中，队空标志为_____，队满标志为_____。

(2) 当 rear>=front 时，队列长度为_____；当 rear<front 时，队列长度是_____。

19. 队列也有_____和_____两种存储结构。

20. 为了方便，对顺序存储队列，通常规定头指针 front 总是指向_____，尾指针 rear 总是指向_____。

21. 队列的“假溢出”是指_____。

22. 循环队列付出了_____代价，在不移动数据的条件下解决了队列的“假溢出”问题。

二、选择题

1. 一个栈的入栈序列是a、b、c、d、e,则出栈序列不可能的是(　　)。

A. edcba　　B. dcbae　　C. dceab　　D. abcde

2. 判定一个顺序栈ST(最多元素为m)为空的条件是(　　)。

A. ST->top!=0　　B. ST->top==0

C. ST->top!=m　　D. ST->top==m

3. a*(b+c)-d的后缀表达式是(　　)。

A. abcdd+-　　B. abc+*d-　　C. abc*+d-　　D. -+*abcd

4. 有一栈,元素a、b、c、d只能依次进栈,则出栈序列中不可能得到的是(　　)。

A. d、c、b、a　　B. c、b、a、d　　C. a、b、c、d　　D. d、c、a、b

5. 如果以链表作为栈的存储结构,则出栈操作时(　　)。

A. 必须判别栈是否满　　B. 必须判别栈是否为空

C. 必须判别栈元素类型　　D. 可不做任何判断

6. 如果以链表作为栈的存储结构,则入栈操作时(　　)。

A. 必须判别栈是否满　　B. 必须判别栈是否为空

C. 必须判别栈元素类型　　D. 可不做任何判断

7. 插入和删除只能在一端进行线性表,称为(　　)。

A. 队列　　B. 循环队列　　C. 栈　　D. 循环栈

8. 一个顺序栈一旦说明,其占用空间的大小(　　)。

A. 已固定　　B. 可以变动　　C. 不能固定　　D. 动态变化

9. 向一个栈顶指针为H的链栈中插入一个s所指向的结点时,需执行(　　)。

A. H->link=s

B. s->link=H->link; H->link=s;

C. s->link=H;H=s;

D. s->link=H; H=H->link;

10. 向一个栈顶指针为H的链栈中执行出栈运算时,需执行(　　)。

A. p=H;H=H->link;free(p);

B. H=H->link;free(H);

C. p=H;H->link=H->link->link;free(p);

D. p=H;H=H->link;

11. 在队列中存取数据的原则是(　　)。

A. 先进先出　　B. 后进先出　　C. 先进后出　　D. 随意进出

12. 栈和队列的共同点是(　　)。

A. 都是先进先出　　B. 都是后进先出

C. 只允许在端点处插入和删除元素　　D. 没有共同点

13. 一个队列的入队序列是a、b、c、d,则出队序列是(　　)。

A. a,b,c,d　　B. a, c, b,d　　C. d, c,b,a　　D. a, c, b,d

14. 一个顺序队列的队头元素为(　　)。

A. Q[sp->front]　　B. Q[sp->front+1]

C. Q[sp->front-1]　　D. Q[sp->rear]

15. 队列是限定在(　　)进行操作的线性表。

A. 中间　　B. 队头　　C. 队尾　　D. 端点

16. 判断一个顺序存储的队列 sp 为空的条件是(　　)。

A. sp->front=sp->rear　　B. sp->front=sp->rear+1

C. sp->front=sp->rear-1　　D. sp->front=NULL

17. 在一个有头结点的链队列中,假设 f 和 r 分别为队首和队尾指针,则插入 s 所指的结点的运算是(　　)。

A. f->next=s; f=s;　　B. r->next=s; r=s;

C. s->next=r; r=s;　　D. s->next=f; f=s;

18. 在一个有头结点的链队列中,假设 f 和 r 分别为队首和队尾指针,则队头出队的运算是(　　)。

A. q=f->next; f->next=f->next->next; free(q);

B. q=f; f->next=f->next->next; free(q);

C. f->next=f->next->next; q=f->next; free(q);

D. q=f->next->next; f=f->next; free(q);

19. 判断一个循环队列 cq(最多元素为 m)为满的条件是(　　)。

A. cq->rear-cq-front=m;

B. (cq->rear+1)%m=cq->front;

C. cq->front=cq->rear;

D. cq->rear=m-1;

20. 判断一个循环队列 cq(最多元素为 m)为空的条件是(　　)。

A. cq->rear-cq-front=m;

B. (cq->rear+1)%m=cq->front;

C. cq->front=cq->rear;

D. cq->rear=m-1;

21. 循环队列用数组 A[0,m-1]存放其元素值,已知头尾指针分别是 front 和 rear,则当前队列中元素的数量是(　　)。

A. (rear-front+m)%m　　B. rear-front+1

C. rear-front-1　　D. rear-front

三、运算题

1. 设将整数 1、2、3、4 依次进栈,请回答下述问题。

(1) 若入、出栈秩序为 Push(1)、Pop()、Push(2)、Push(3)、Pop()、Pop( )、Push(4)、Pop( ),则出栈的数字序列是什么(这里 Push(i)表示 i 进栈,Pop( )表示出栈)?

(2) 能否得到出栈序列 1423 和 1432? 并说明为什么不能得到或者如何得到。

2. 假设 Q[1,7]是一个顺序队列,初始状态为 front=rear=0,求完成下列各运算后队列的头尾指针的值,若不能入队,请指出其元素,并解释理由,画出上述各运算过程。

(1) a、b、c 入队;

(2) a、b 出队;

(3) d、e、f 入队;

(4) c 出队;

(5) g、h 入队。

3. 如果第 2 题中的 Q[1,7]是循环队列,初始状态 front=rear=0,求完成与前题同样的操作,并画出运算过程。

四、程序设计题

1. 设计算法判断一个算术表达式的圆括号是否正确配对。

提示:对表达式进行扫描,凡遇到"("就进栈,遇")"就退掉栈顶的"(",表达式被扫描完毕,栈应为空。

2. 设计算法,要求用栈来判断输入的字符串(以"#"号结束)是否为回文。回文即字符串顺读与逆读一样(不含空格),如字符串"madam"即为回文。

3. 设计算法,要求用栈和队列进行回文判断。

4. 设计算法,要求用循环链表实现队列,设置一个指针指向队尾元素(注意不设置头指针),该队列至少具有创建空队列、入队和出队算法,并编写主函数对各个函数进行测试。

5. 编写一个函数从一给定的顺序表 L 中删除元素值在 x 到 $y(x\leqslant y)$之间的所有元素,要求以较高的效率来实现。要求:与队列基本运算的实现程序结合在一起,实现队列基本运算的扩充,上机调试通过。

6. 假设以数组 sequ[m]存放循环队列的元素,同时设变量 rear 和 quelen 分别指示循环队列中队尾元素的位置和队列中内含元素的个数。试给出判别此循环队列的队满条件,并写出相应的入队和出队算法。

五、实训题

1. 用递归算法求解"聪明的学生"问题。

2. 实现例 3.7 和例 3.9 中的算法。