

第3章 栈和队列

1. 选择题

(1) 若让元素 1, 2, 3, 4, 5 依次进栈, 则出栈次序不可能出现在 () 种情况。

A. 5, 4, 3, 2, 1 B. 2, 1, 5, 4, 3 C. 4, 3, 1, 2, 5 D. 2, 3, 5, 4, 1

答案: C

解释: 栈是后进先出的线性表, 不难发现 C 选项中元素 1 比元素 2 先出栈, 违背了栈的后进先出原则, 所以不可能出现 C 选项所示的情况。

(2) 若已知一个栈的入栈序列是 1, 2, 3, ..., n, 其输出序列为 $p_1, p_2, p_3, \dots, p_n$, 若 $p_1=n$, 则 p_i 为 ()。

A. i B. n-i C. n-i+1 D. 不确定

答案: C

解释: 栈是后进先出的线性表, 一个栈的入栈序列是 1, 2, 3, ..., n, 而输出序列的第一个元素为 n, 说明 1, 2, 3, ..., n 一次性全部进栈, 再进行输出, 所以 $p_1=n, p_2=n-1, \dots, p_i=n-i+1$ 。

(3) 数组 Q [n] 用来表示一个循环队列, f 为当前队列头元素的前一位置, r 为队尾元素的位置, 假定队列中元素的个数小于 n, 计算队列中元素个数的公式为 ()。

A. r-f B. (n+f-r)%n C. n+r-f D. (n+r-f)%n

答案: D

解释: 对于非循环队列, 尾指针和头指针的差值便是队列的长度, 而对于循环队列, 差值可能为负数, 所以需要将差值加上 MAXSIZE (本题为 n), 然后与 MAXSIZE (本题为 n) 求余, 即 $(n+r-f)\%n$ 。

(4) 链式栈结点为: (data,link), top 指向栈顶. 若想摘除栈顶结点, 并将删除结点的值保存到 x 中, 则应执行操作 ()。

A. $x=top->data; top=top->link;$ B. $top=top->link; x=top->link;$
C. $x=top; top=top->link;$ D. $x=top->link;$

答案: A

解释: $x=top->data$ 将结点的值保存到 x 中, $top=top->link$ 栈顶指针指向栈顶下一结点, 即摘除栈顶结点。

(5) 设有一个递归算法如下

```
int fact(int n) { //n 大于等于 0
    if(n<=0) return 1;
    else return n*fact(n-1); }
```

则计算 fact(n) 需要调用该函数的次数为 ()。

A. n+1 B. n-1 C. n D. n+2

答案: A

解释: 特殊值法。设 $n=0$, 易知仅调用一次 fact(n) 函数, 故选 A。

(6) 栈在 () 中有所应用。

A. 递归调用 B. 函数调用 C. 表达式求值 D. 前三个选项都有

答案: D

解释: 递归调用、函数调用、表达式求值均用到了栈的后进先出性质。

(7) 为解决计算机主机与打印机间速度不匹配问题, 通常设一个打印数据缓冲区。主机将要输出的数据依次写入该缓冲区, 而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是 ()。

A. 队列 B. 栈 C. 线性表 D. 有序表

答案: A

解释：解决缓冲区问题应利用一种先进先出的线性表，而队列正是一种先进先出的线性表。

(8) 设栈 S 和队列 Q 的初始状态为空，元素 e1、e2、e3、e4、e5 和 e6 依次进入栈 S，一个元素出栈后即进入 Q，若 6 个元素出队的序列是 e2、e4、e3、e6、e5 和 e1，则栈 S 的容量至少应该是 ()。

- A. 2 B. 3 C. 4 D. 6

答案：B

解释：元素出队的序列是 e2、e4、e3、e6、e5 和 e1，可知元素入队的序列是 e2、e4、e3、e6、e5 和 e1，即元素出栈的序列也是 e2、e4、e3、e6、e5 和 e1，而元素 e1、e2、e3、e4、e5 和 e6 依次进入栈，易知栈 S 中最多同时存在 3 个元素，故栈 S 的容量至少为 3。

(9) 若一个栈以向量 V[1..n] 存储，初始栈顶指针 top 设为 n+1，则元素 x 进栈的正确操作是 ()。

- A. top++; V[top]=x; B. V[top]=x; top++;
C. top--; V[top]=x; D. V[top]=x; top--;

答案：C

解释：初始栈顶指针 top 为 n+1，说明元素从数组向量的高端地址进栈，又因为元素存储在向量空间 V[1..n] 中，所以进栈时 top 指针先下移变为 n，之后将元素 x 存储在 V[n]。

(10) 设计一个判别表达式中左、右括号是否配对出现的算法，采用 () 数据结构最佳。

- A. 线性表的顺序存储结构 B. 队列
C. 线性表的链式存储结构 D. 栈

答案：D

解释：利用栈的后进先出原则。

(11) 用链接方式存储的队列，在进行删除运算时 ()。

- A. 仅修改头指针 B. 仅修改尾指针
C. 头、尾指针都要修改 D. 头、尾指针可能都要修改

答案：D

解释：一般情况下只修改头指针，但是，当删除的是队列中最后一个元素时，队尾指针也丢失了，因此需对队尾指针重新赋值。

(12) 循环队列存储在数组 A[0..m] 中，则入队时的操作为 ()。

- A. rear=rear+1 B. rear=(rear+1)%(m-1)
C. rear=(rear+1)%m D. rear=(rear+1)%(m+1)

答案：D

解释：数组 A[0..m] 中共含有 m+1 个元素，故在求模运算时应除以 m+1。

(13) 最大容量为 n 的循环队列，队尾指针是 rear，队头是 front，则队空的条件是 ()。

- A. (rear+1)%n==front B. rear==front
C. rear+1==front D. (rear-1)%n==front

答案：B

解释：最大容量为 n 的循环队列，队满条件是 (rear+1)%n==front，队空条件是 rear==front。

(14) 栈和队列的共同点是 ()。

- A. 都是先进先出 B. 都是先进后出
C. 只允许在端点处插入和删除元素 D. 没有共同点

答案：C

解释：栈只允许在栈顶处进行插入和删除元素，队列只允许在队尾插入元素和在队头删除元素。

(15) 一个递归算法必须包括 ()。

- A. 递归部分 B. 终止条件和递归部分
C. 迭代部分 D. 终止条件和迭代部分

答案：B

2. 算法设计题

(1) 将编号为 0 和 1 的两个栈存放于一个数组空间 $V[m]$ 中，栈底分别处于数组的两端。当第 0 号栈的栈顶指针 $top[0]$ 等于 -1 时该栈为空，当第 1 号栈的栈顶指针 $top[1]$ 等于 m 时该栈为空。两个栈均从两端向中间增长。试编写双栈初始化，判断栈空、栈满、进栈和出栈等算法的函数。双栈数据结构的定义如下：

```
Typedef struct
{int top[2],bot[2];           //栈顶和栈底指针
SElemType *V;                //栈数组
int m;                        //栈最大可容纳元素个数
}DblStack
```

[题目分析]

两栈共享向量空间，将两栈栈底设在向量两端，初始时，左栈顶指针为 -1，右栈顶为 m 。两栈顶指针相邻时为栈满。两栈顶相向、迎面增长，栈顶指针指向栈顶元素。

[算法描述]

(1) 栈初始化

```
int Init()
{S.top[0]=-1;
 S.top[1]=m;
 return 1; //初始化成功
}
```

(2) 入栈操作：

```
int push(stk S,int i,int x)
//i 为栈号，i=0 表示左栈，i=1 为右栈，x 是入栈元素。入栈成功返回 1，失败返回 0
{if(i<0||i>1){ cout<<"栈号输入不对"<<endl;exit(0);}
if(S.top[1]-S.top[0]==1) {cout<<"栈已满"<<endl;return(0);}
switch(i)
{case 0: S.V[++S.top[0]]=x; return(1); break;
 case 1: S.V[--S.top[1]]=x; return(1);
 }
} // push
```

(3) 退栈操作

```
ElemType pop(stk S,int i)
// 退栈。i 代表栈号，i=0 时为左栈，i=1 时为右栈。退栈成功时返回退栈元素
// 否则返回-1
{if(i<0 || i>1){cout<<"栈号输入错误"<<endl; exit(0);}
switch(i)
{case 0: if(S.top[0]==-1) {cout<<"栈空"<<endl; return (-1); }
 else return(S.V[S.top[0]--]);
 case 1: if(S.top[1]==m { cout<<"栈空"<<endl; return(-1);}
 else return(S.V[S.top[1]++]);
 } // switch
} // 算法结束
```

(4) 判断栈空

```
int Empty();
```

```
{return (S.top[0]==-1 && S.top[1]==m);
}
```

[算法讨论]

请注意算法中两栈入栈和退栈时的栈顶指针的计算。左栈是通常意义下的栈，而右栈入栈操作时，其栈顶指针左移（减1），退栈时，栈顶指针右移（加1）。

（2）回文是指正读反读均相同的字符序列，如“abba”和“abdba”均是回文，但“good”不是回文。试写一个算法判定给定的字符向量是否为回文。（提示：将一半字符入栈）

[题目分析]

将字符串前半入栈，然后，栈中元素和字符串后半进行比较。即将第一个出栈元素和后半串中第一个字符比较，若相等，则再出栈一个元素与后一个字符比较，……，直至栈空，结论为字符序列是回文。在出栈元素与串中字符比较不等时，结论字符序列不是回文。

[算法描述]

```
#define StackSize 100 //假定预分配的栈空间最多为 100 个元素
typedef char DataType; //假定栈元素的数据类型为字符
typedef struct
{DataType data[StackSize];
 int top;
}SeqStack;

int IsHuiwen( char *t)
//判断 t 字符向量是否为回文，若是，返回 1，否则返回 0
SeqStack s;
int i , len;
char temp;
InitStack( &s);
len=strlen(t); //求向量长度
for ( i=0; i<len/2; i++)//将一半字符入栈
    Push( &s, t[i]);
while( !EmptyStack( &s))
{// 每弹出一个字符与相应字符比较
    temp=Pop( &s);
    if( temp!=S[i]) return 0 ;// 不等则返回 0
    else i++;
}
return 1 ; // 比较完毕均相等则返回 1
}
```

（3）设从键盘输入一整数的序列： $a_1, a_2, a_3, \dots, a_n$ ，试编写算法实现：用栈结构存储输入的整数，当 $a_i \neq -1$ 时，将 a_i 进栈；当 $a_i = -1$ 时，输出栈顶整数并出栈。算法应对异常情况（入栈满等）给出相应的信息。

[算法描述]

```
#define maxsize 栈空间容量
void InOutS(int s[maxsize])
//s 是元素为整数的栈，本算法进行入栈和退栈操作。
{int top=0; //top 为栈顶指针，定义 top=0 时为栈空。
 for(i=1; i<=n; i++) //n 个整数序列作处理。
 {cin>>x; //从键盘读入整数序列。
  if(x!=-1) // 读入的整数不等于-1 时入栈。
  {if(top==maxsize-1){cout<<“栈满”<<endl;exit(0);}
   else s[++top]=x; //x 入栈。
  }
  else //读入的整数等于-1 时退栈。
  {cout<<s[top]<<endl;
   top--;
  }
 }
```

```

        {if(top==0){ cout<<“栈空”<<endl;exit(0);}
        else cout<<“出栈元素是”<< s[top--]<<endl;}
    }
} //算法结束。

```

(4) 从键盘上输入一个后缀表达式，试编写算法计算表达式的值。规定：逆波兰表达式的长度不超过一行，以\$符作为输入结束，操作数之间用空格分隔，操作符只可能有+、-、*、/四种运算。例如：
234 34+2*\$。

[题目分析]

逆波兰表达式(即后缀表达式)求值规则如下：设立运算数栈 OPND, 对表达式从左到右扫描(读入)，当表达式中扫描到数时，压入 OPND 栈。当扫描到运算符时，从 OPND 退出两个数，进行相应运算，结果再压入 OPND 栈。这个过程一直进行到读出表达式结束符\$，这时 OPND 栈中只有一个数，就是结果。

[算法描述]

```

float expr( )
//从键盘输入逆波兰表达式，以‘$’表示输入结束，本算法求逆波兰式表达式的值。
{float OPND[30]; // OPND 是操作数栈。
  init(OPND); //两栈初始化。
  float num=0.0; //数字初始化。
  cin>>x;//x 是字符型变量。
  while(x!='$')
  {switch
    {case '0' <=x<='9' :
      while((x>='0' && x<='9') || x=='.' ) //拼数
        if(x!='.') //处理整数
          {num=num*10+ (ord(x)-ord( '0' )) ; cin>>x;}
        else //处理小数部分。
          {scale=10.0; cin>>x;
            while(x>='0' && x<='9' )
              {num=num+(ord(x)-ord( '0' ))/scale;
                scale=scale*10; cin>>x; }
          } //else
      push(OPND,num); num=0.0;//数压入栈，下个数字初始化
    case x= ' ':break; //遇空格，继续读下一个字符。
    case x= '+':push(OPND, pop(OPND)+pop(OPND));break;
    case x= '-':x1=pop(OPND);x2=pop(OPND);push(OPND, x2-x1);break;
    case x= '*':push(OPND, pop(OPND)*pop(OPND));break;
    case x= '/':x1=pop(OPND);x2=pop(OPND);push(OPND, x2/x1);break;
    default: //其它符号不作处理。
  } //结束 switch
  cin>>x;//读入表达式中下一个字符。
} //结束 while (x!= '$')
cout<<“后缀表达式的值为”<<pop(OPND);
} //算法结束。

```

[算法讨论]假设输入的后缀表达式是正确的，未作错误检查。算法中拼数部分是核心。若遇到大于等于‘0’且小于等于‘9’的字符，认为是数。这种字符的序号减去字符‘0’的序号得出数。对于整

数，每读入一个数字字符，前面得到的部分数要乘上 10 再加新读入的数得到新的部分数。当读到小数点，认为数的整数部分已完，要接着处理小数部分。小数部分的数要除以 10（或 10 的幂数）变成十分位，百分位，千分位数等等，与前面部分数相加。在拼数过程中，若遇非数字字符，表示数已拼完，将数压入栈中，并且将变量 num 恢复为 0，准备下一个数。这时对新读入的字符进入 ‘+’、‘-’、‘*’、‘/’ 及空格的判断，因此在结束处理数字字符的 case 后，不能加入 break 语句。

（5）假设以 I 和 O 分别表示入栈和出栈操作。栈的初态和终态均为空，入栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列，称可以操作的序列为合法序列，否则称为非法序列。

①下面所示的序列中哪些是合法的？

A. IOIOIOIO B. IOOIOIOIO C. IIIIOIOIO D. IIIIOOIOO

②通过对①的分析，写出一个算法，判定所给的操作序列是否合法。若合法，返回 true，否则返回 false（假定被判定的操作序列已存入一维数组中）。

答案：

①A 和 D 是合法序列，B 和 C 是非法序列。

②设被判定的操作序列已存入一维数组 A 中。

```
int Judge(char A[])
//判断字符串 A 中的输入输出序列是否是合法序列。如是，返回 true，否则返回 false。
{i=0;                //i 为下标。
j=k=0;                //j 和 k 分别为 I 和字母 O 的个数。
while(A[i]!='\0') //当未到字符串尾就作。
{switch(A[i])
{case 'I' : j++; break; //入栈次数增 1。
case 'O' : k++; if(k>j){cout<<“序列非法”<<endl; exit(0);}
}
i++; //不论 A[i] 是 ‘I’ 或 ‘O’，指针 i 均后移。}
if(j!=k) {cout<<“序列非法”<<endl; return(false);}
else { cout<<“序列合法”<<endl; return(true);}
} //算法结束。
```

[算法讨论]在入栈出栈序列（即由 ‘I’ 和 ‘O’ 组成的字符串）的任一位置，入栈次数（‘I’ 的个数）都必须大于等于出栈次数（即 ‘O’ 的个数），否则视作非法序列，立即给出信息，退出算法。整个序列（即读到字符串中字符串的结束标记 ‘\0’），入栈次数必须等于出栈次数（题目中要求栈的初态和终态都为空），否则视为非法序列。

（6）假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素站点(注意不设头指针)，试编写相应的置空队、判队空、入队和出队等算法。

[题目分析]

置空队就是建立一个头节点，并把头尾指针都指向头节点，头节点是不存放数据的；判队空就是当头指针等于尾指针时，队空；入队时，将新的节点插入到链队列的尾部，同时将尾指针指向这个节点；出队时，删除的是队头节点，要注意队列的长度大于 1 还是等于 1 的情况，这个时候要注意尾指针的修改，如果等于 1，则要删除尾指针指向的节点。

[算法描述]

//先定义链队结构:

```
typedef struct queuenode
{Datatype data;
struct queuenode *next;
```

```

}QueueNode; //以上是结点类型的定义
typedef struct
{queuenode *rear;
}LinkQueue; //只设一个指向队尾元素的指针

```

(1) 置空队

```

void InitQueue(LinkQueue *Q)
{ //置空队：就是使头结点成为队尾元素
  QueueNode *s;
  Q->rear = Q->rear->next; //将队尾指针指向头结点
  while (Q->rear!=Q->rear->next) //当队列非空，将队中元素逐个出队
  {s=Q->rear->next;
   Q->rear->next=s->next;
   delete s;
  } //回收结点空间
}

```

(2) 判队空

```

int EmptyQueue(LinkQueue *Q)
{ //判队空。当头结点的 next 指针指向自己时为空队
  return Q->rear->next->next==Q->rear->next;
}

```

(3) 入队

```

void EnQueue(LinkQueue *Q, Datatype x)
{ //入队。也就是在尾结点处插入元素
  QueueNode *p=new QueueNode; //申请新结点
  p->data=x; p->next=Q->rear->next; //初始化新结点并链入
  Q->rear->next=p;
  Q->rear=p; //将尾指针移至新结点
}

```

(4) 出队

```

Datatype DeQueue(LinkQueue *Q)
{ //出队,把头结点之后的元素摘下
  Datatype t;
  QueueNode *p;
  if(EmptyQueue(Q))
    Error("Queue underflow");
  p=Q->rear->next->next; //p 指向将要摘下的结点
  x=p->data; //保存结点中数据
  if (p==Q->rear)
  { //当队列中只有一个结点时，p 结点出队后，要将队尾指针指向头结点
    Q->rear = Q->rear->next;
    Q->rear->next=p->next;
  }
  else
    Q->rear->next->next=p->next; //摘下结点 p
  delete p; //释放被删结点
  return x;
}

```

(7) 假设以数组 $Q[m]$ 存放循环队列中的元素, 同时设置一个标志 tag , 以 $tag == 0$ 和 $tag == 1$ 来区别在队头指针($front$)和队尾指针($rear$)相等时, 队列状态为“空”还是“满”。试编写与此结构相应的插入($enqueue$)和删除($dlqueue$)算法。

[算法描述]

(1) 初始化

```
SeQueue QueueInit (SeQueue Q)
{//初始化队列
    Q.front=Q.rear=0; Q.tag=0;
    return Q;
}
```

(2) 入队

```
SeQueue QueueIn (SeQueue Q, int e)
{//入队列
    if((Q.tag==1) && (Q.rear==Q.front)) cout<<"队列已满"<<endl;
    else
    {Q.rear=(Q.rear+1) % m;
      Q.data[Q.rear]=e;
      if(Q.tag==0) Q.tag=1; //队列已不空
    }
    return Q;
}
```

(3) 出队

```
ElemType QueueOut (SeQueue Q)
{//出队列
    if(Q.tag==0) { cout<<"队列为空"<<endl; exit(0);}
    else
    {Q.front=(Q.front+1) % m;
      e=Q.data[Q.front];
      if(Q.front==Q.rear) Q.tag=0; //空队列
    }
    return(e);
}
```

(8) 如果允许在循环队列的两端都可以进行插入和删除操作。要求：

① 写出循环队列的类型定义；

② 写出“从队尾删除”和“从队头插入”的算法。

[题目分析] 用一维数组 $v[0..M-1]$ 实现循环队列，其中 M 是队列长度。设队头指针 $front$ 和队尾指针 $rear$ ，约定 $front$ 指向队头元素的前一位置， $rear$ 指向队尾元素。定义 $front=rear$ 时为队空， $(rear+1)\%m=front$ 为队满。约定队头端入队向下标小的方向发展，队尾端入队向下标大的方向发展。

[算法描述]

①

```
#define M  队列可能达到的最大长度
typedef struct
{elemtp data[M];
    int front,rear;
}cycqueue;
```

②

```
elemtp delqueue ( cycqueue Q)
```

//Q 是如上定义的循环队列，本算法实现从队尾删除，若删除成功，返回被删除元素，否则给出出错信息。

```
{if (Q.front==Q.rear) { cout<<"队列空"<<endl; exit(0);}
    Q.rear=(Q.rear-1+M)%M;          //修改队尾指针。
    return(Q.data[(Q.rear+1+M)%M]); //返回出队元素。
} //从队尾删除算法结束
```

```
void enqueue (cycqueue Q, elemtp x)
```

// Q 是顺序存储的循环队列，本算法实现“从队头插入”元素 x。

```
{if (Q.rear==(Q.front-1+M)%M) { cout<<"队满"<<endl; exit(0);}
    Q.data[Q.front]=x;          //x 入队列
    Q.front=(Q.front-1+M)%M;    //修改队头指针。
} // 结束从队头插入算法。
```

(9) 已知 Ackermann 函数定义如下:

$$\text{Ack}(m, n) = \begin{cases} n+1 & \text{当 } m=0 \text{ 时} \\ \text{Ack}(m-1, 1) & \text{当 } m \neq 0, n=0 \text{ 时} \\ \text{Ack}(m-1, \text{Ack}(m, n-1)) & \text{当 } m \neq 0, n \neq 0 \text{ 时} \end{cases}$$

① 写出计算 Ack(m, n) 的递归算法，并根据此算法给出 Ack(2, 1) 的计算过程。

② 写出计算 Ack(m, n) 的非递归算法。

[算法描述]

```
int Ack(int m, n)
{if (m==0) return(n+1);
    else if (m!=0&& n==0) return(Ack(m-1, 1));
    else return(Ack(m-1, Ack(m, n-1)));
} //算法结束
```

① Ack(2, 1) 的计算过程

```
Ack(2, 1) = Ack(1, Ack(2, 0))          //因 m<>0, n<>0 而得
           = Ack(1, Ack(1, 1))          //因 m<>0, n=0 而得
           = Ack(1, Ack(0, Ack(1, 0)))   //因 m<>0, n<>0 而得
           = Ack(1, Ack(0, Ack(0, 1)))   //因 m<>0, n=0 而得
           = Ack(1, Ack(0, 2))           //因 m=0 而得
           = Ack(1, 3)                   //因 m=0 而得
           = Ack(0, Ack(1, 2))           //因 m<>0, n<>0 而得
           = Ack(0, Ack(0, Ack(1, 1)))   //因 m<>0, n<>0 而得
           = Ack(0, Ack(0, Ack(0, Ack(1, 0)))) //因 m<>0, n<>0 而得
           = Ack(0, Ack(0, Ack(0, Ack(0, 1)))) //因 m<>0, n=0 而得
           = Ack(0, Ack(0, Ack(0, 2)))   //因 m=0 而得
           = Ack(0, Ack(0, 3))           //因 m=0 而得
           = Ack(0, 4)                   //因 n=0 而得
           = 5                           //因 n=0 而得
```

②

```
int Ackerman(int m, int n)
{int akm[M][N]; int i, j;
```

```

for (j=0; j<N; j++) akm[0][j]=j+1;
for (i=1; i<m; i++)
{
    akm[i][0]=akm[i-1][1];
    for (j=1; j<N; j++)
        akm[i][j]=akm[i-1][akm[i][j-1]];
}
return (akm[m][n]);
} //算法结束

```

(10) 已知 f 为单链表的表头指针, 链表中存储的都是整型数据, 试写出实现下列运算的递归算法:

- ① 求链表中的最大整数;
- ② 求链表的结点个数;
- ③ 求所有整数的平均值。

[算法描述]

①

```

int GetMax(LinkList p)
{
    if(!p->next)
        return p->data;
    else
    {
        int max=GetMax(p->next);
        return p->data>=max ? p->data:max;
    }
}

```

②

```

int GetLength(LinkList p)
{
    if(!p->next)
        return 1;
    else
    {
        return GetLength(p->next)+1;
    }
}

```

③

```

double GetAverage(LinkList p, int n)
{
    if(!p->next)
        return p->data;
    else
    {
        double ave=GetAverage(p->next, n-1);
        return (ave*(n-1)+p->data)/n;
    }
}

```

}