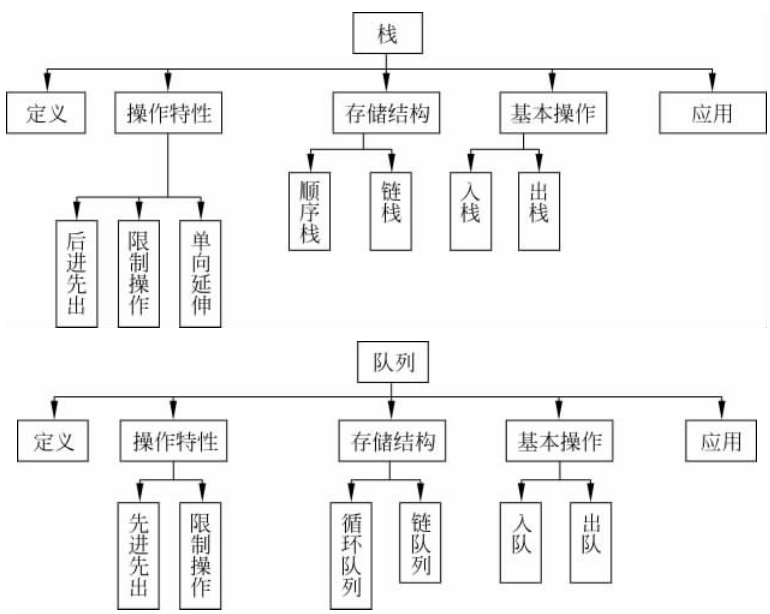
第3章 栈和队列

栈和队列是两种特殊的线性表，在算法设计中应用广泛。它们是操作受限的线性结构，一般地，在线性表上的插入和删除等操作是不受什么限制的；而栈只允许在线性表的同一端进行插入和删除；队列则只允许在线性表的一端进行插入而在另一端进行删除。本章除了讨论栈和队列的定义、表示方法和运算外，还给出了一些应用的案例。

本章知识结构图：



3.1 栈

3.1.1 栈的定义及操作特性

1. 栈的定义

栈(Stack)是一种操作受限的线性表，只允许在表的一端进行插入和删除的操作，允许插入和删除的一端称为栈顶(Top)，另一端称为栈底(Bottom)，又称为后进先出(Last In First out，LIFO)线性表。

如图 3-1 所示，设栈 $S=(a_1,a_2,\cdots,a_n)$，则称 a_1 为栈底元素，a_n 为栈顶元素。

在日常生活中有许多栈的例子，如经常使用的饼干圆筒，圆筒的底部可以看作栈底，开口的顶部可以看作栈顶，向圆筒中装饼干时，总是从底部一层一层地放进去，相当于入栈，当吃饼干时，只能从顶部一个一个地拿出，相当于出栈。

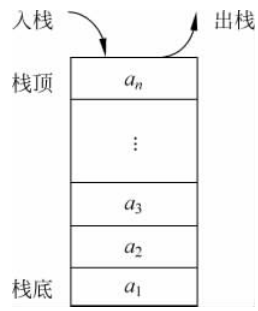


图 3-1 栈的示意图

2. 栈的操作特性

栈的操作特性是遵循“后进先出”原则。

向一个栈插入新元素称为进栈或入栈操作；删除一个元素称为出栈或退栈操作。也就是说，最先放入栈中的元素在栈底，最后放入栈中的元素在栈顶；最先出栈的是后入栈的栈顶元素，而先入栈的栈底元素最后出栈。

栈的基本运算主要有以下几种。

(1) 初始化运算：建立一个空栈。

(2) 销毁栈运算：释放栈占用的内存空间。

(3) 入栈运算：在栈顶插入一个新的数据元素。

(4) 出栈运算：在栈顶删除一个数据元素。

(5) 取栈顶运算：读栈顶元素的内容。

(6) 判空运算：判断一个栈是否为空。

例如，设一个栈 S 为(a,b,c)，a 为栈底元素，则 c 为栈顶元素。若向 S 中进行入栈运算，插入一个元素 d，则栈 S 变为(a,b,c,d)，此时 d 为栈顶元素；若接着从栈 S 中依次进行两个出栈运算，则先从栈 S 中删除 d，再删除 c，栈 S 变为(a,b)，栈顶元素为 b，如图 3-2 所示。

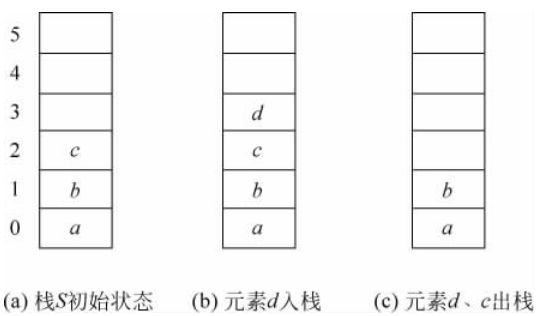


图 3-2 元素入栈、出栈示意图

栈也有两种存储表示方法，栈的顺序存储结构称为顺序栈，栈的链式存储结构称为链栈，两种结构各有优势。

3.1.2 栈的顺序存储结构及其基本运算的实现

栈的顺序存储结构，简称顺序栈，它是利用一组地址连续的存储单元依次存放自栈底到栈项的数据元素。即由一个一维数组 data 和一个记录栈顶元素位置的变量 top 组成。

顺序栈类型定义如下。

```
#define MaxSize 100                //顺序栈的初始分配空间大小
typedef struct
{   ElemType data[MaxSize];        //保存栈中元素,这里假设 ElemType 为 char 类型
    int top;                       //栈顶指针
}SqStack;
```

在上述顺序栈定义中,ElemType 为栈元素的数据类型,MaxSize 为一个常量,表示 data 数组中最多可以放的元素个数,data 元素的下标范围为 0～MaxSize－1。当 top＝－1 时表示栈空；当 top＝MaxSize－1 时表示栈满。

归纳起来,对于顺序栈 st,其初始时置 st.top＝－1,它的 4 个要素如下。

(1) 栈空条件：st.top＝＝－1。

(2) 栈满条件：st.top＝＝MaxSize－1。

(3) 元素 x 进栈操作：st.top＋＋；将元素 x 放在 st.data[st.top]中。

(4) 出栈元素 x 操作：取出栈元素 x＝st.data[st.top]；st.top－－。

顺序栈的基本运算如下。

1. 顺序栈的初始化算法

其主要操作是：设定栈顶指针 top 为－1。

算法 3.1

```
void InitStack(SqStack &st)        //st 为引用型参数
{
st.top = -1;
}
```

2. 顺序栈的销毁运算算法

顺序栈的内存空间是由系统自动分配的,在不再需要时由系统自动释放其空间。

3. 顺序栈的入栈运算算法

向顺序栈中插入一个新元素称为入栈或进栈运算。在非空栈中的栈顶指针始终在栈顶元素的位置上,每当插入新的栈顶元素时,指针 top 增 1。

其主要操作是：栈顶指针加 1,将进栈元素放在栈顶处。

算法 3.2

```
int Push(SqStack &st,ElemType x)
{   if(st.top == MaxSize - 1)      //栈满上溢出返回 0
    return 0;
    else
    { st.top++;
     st.data[st.top] = x;
     return 1;                     //成功进栈返回 1
    }
}
```

4. 顺序栈的出栈运算算法

将顺序栈的栈顶元素删除的运算称为出栈或退栈,删除栈顶元素时,指针 top 减 1。其主要操作是:先将栈顶元素取出,然后将栈顶指针减 1。

算法 3.3

```
int Pop(SqStack &st,ElemType &x)    //x 为引用型参数
{ if(st.top == -1)                  //栈空返回 0
  return 0;
  else
  { x = st.data[st.top];
    st.top--;
    return 1;                       //成功出栈返回 1
  }
}
```

5. 顺序栈取栈顶元素算法

其主要操作是:将栈顶指针 top 处的元素取出赋给变量 x。

算法 3.4

```
int GetTop(SqStack st,ElemType &x) //x 为引用型参数
{ if (st.top == -1)                //栈空返回 0
  return 0;
else
{ x = st.data[st.top];
  return 1;                        //成功取栈顶元素返回 1
  }
}
```

6. 判断栈空运算算法

其主要操作是:若栈为空(top==-1)则返回值 1,否则返回值 0。

算法 3.5

```
int StackEmpty(SqStack st)
{ if(st.top == -1)                 //栈空返回 1
    return 1;
  else
    return 0;                      //栈不空返回 0
}
```

当顺序栈的基本运算算法设计好后,给出主函数算法调用这些基本运算算法。

```
# include<stdio.h>
# include"SqStack.h"               //包含前面的顺序栈基本运算函数
void main()
{ SqStack st;                      //定义一个顺序栈 st
  ElemType e;
```

```
  printf("初始化 st\n");
  InitStack(st);
  printf("栈 %s\n",(StackEmpty(st) == 1?"空":"不空"));
  printf("a 进栈\n");Push(st,'a');
  printf("b 进栈\n");Push(st,'b');
  printf("c 进栈\n");Push(st,'c');
  printf("d 进栈\n");Push(st,'d');
  printf("栈 %s\n",(StackEmpty(st) == 1?"空":"不空"));
  GetTop(st,e);
  printf("栈顶元素:%c\n",e);
  printf("出栈次序:");
  While(!StackEmpty(st))          //栈不空循环
  { Pop(st,e);                    //出栈元素 e 并输出
    printf("%c",e);
  }
printf("\n");
}
```

3.1.3　栈的链式存储结构及其基本运算的实现

栈的链式存储结构简称链栈，它是一种特殊的单链表，也是一种动态存储结构，通常不会产生栈的溢出，不用预先分配存储空间。它是没有附加头结点且只允许在表头进行插入和删除运算的单链表，该单链表的头指针称为栈顶指针，链表中的每个结点包括数据域和指针域，如图 3-3 所示。

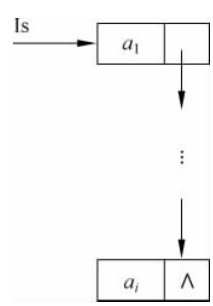


图 3-3　链栈示意图

链栈的类型定义如下：

```
typedef struct node{
    Elemtype data;        //存储结点数据,这里假设 Elemtype 为 char 型
    struct node *next;    //指针域
}LinkStack;
```

链栈 ls 初始时 ls＝NULL，其 4 个要素如下。

(1) 栈空条件：ls＝＝NULL。

(2) 栈满条件：不考虑。

(3) 元素 x 进栈操作：创建存放元素 x 的结点 $*p$，将其插入到栈顶位置。

(4) 出栈元素 x 操作：置 x 为栈顶结点的 data 域，并删除该结点。

链栈的基本运算有链栈的初始化、销毁栈、进栈、出栈、取栈顶元素、判栈空等，算法描述如下。

1. 初始化栈运算算法

其主要操作是：创建一个栈头结点 *ls，用 ls＝NULL 标示栈为空栈。

算法 3.6

```
void InitStack(LinkStack *&ls)                //ls 为引用型参数
{
```

```
  Is = NULL;
}
```

2. 销毁栈运算算法

链栈的所有结点空间都是通过 malloc 函数分配的,在不再需要时通过 free 函数释放所有结点的空间。

算法 3.7

```
void DestroyStack(LinkStack * &Is)
{ Linkstack * pre = Is, * p;
  if(pre == NULL) return;                    //考虑空间栈的情况
  p = pre -> next;
  while(p!= NULL)
  { free(pre);                               //释放 * pre 结点
    pre = p;p = p -> next;                   //pre、p 同步后移
  }
  free(pre);                                 //释放尾结点
}
```

3. 链栈的进栈运算算法

其主要操作是:先创建一个新结点,其 data 域值为 x;然后将该结点插入到 * Is 结点之后作为栈顶结点。

算法 3.8

```
void Push(LinkStack * &Is,ElemType x)        //Is 为引用型参数
{ LinkStack * p;
  p = (LinkStack * )malloc(sizeof(LinkStack));
  p -> data = x;                             //创建结点 * p 用于存放 x
  p -> next = Is;                            //插入 * p 结点作为栈顶结点
  Is = p;
  }
```

4. 链栈的出栈运算算法

其主要操作是:将栈顶结点的 data 域值赋给 x,然后删除该栈顶结点。

算法 3.9

```
int Pop(LinkStack * &Is,ElemType &x)         //Is 为引用型参数
{ LinkStack * p;
  if(Is == NULL)                             //栈空,下溢出返回 0
    return 0;
  else                                       //栈不空时出栈元素 x 并返回 1
  { p = Is;                                  //p 指向栈顶结点
    x = p -> data;                           //取栈顶元素 x
    Is = p -> next;                          //删除结点 * p
    free(p);                                 //释放 * p 结点
  return 1;
}
}
```

5. 取栈顶元素运算算法

其主要操作是：将栈顶结点的 data 域值赋给 x。

算法 3.10

```
int GetTop(LinkStack * Is,ElemType &x)
{ if(Is == NULL)                          //栈空,下溢出时返回 0
    return 0;
  else                                    //栈不空,去栈顶元素 x 并返回 1
    { x = Is -> data;
      return 1;
    }
}
```

6. 判断栈空运算算法

其主要操作是：若栈为空则返回值 1，否则返回值 0。

算法 3.11

```
int StackEmpty(LinkStack * Is)
{ if(Is == NULL) return 1;
  else return 0;
}
```

当链栈的基本运算算法设计好后，给出主函数算法调用这些基本运算算法。

```
#include <stdio.h>
#include"LinkStack.h"                     //包含前面链栈基本运算函数
void main()
{ ElemType e;
  LinkStack * st;                         //定义一个链栈 st
printf("初始化栈 st\n");
InitStack(st);
printf("栈 %s\n",(StackEmpty(st) == 1?"空":"不空"));
printf("a 进栈\n");Push(st,'a');
printf("b 进栈\n");Push(st,'b');
printf("c 进栈\n");Push(st,'c');
printf("d 进栈\n");Push(st,'d');
printf("栈 %s\n",(StackEmpty(st) == 1?"空":"不空"));
GetTop(st,e);
printf("栈顶元素:%c\n",e);
printf("出栈次序:");
while(!StackEmpty(st))                    //栈不空循环
  { Pop(st,e);                            //出栈元素 e 并输出
    printf("%c",e);
  }
printf("\n");
DestroyStack(st);
}
```

3.1.4 栈的应用举例

应用举例 1 表达式求值问题

表达式求值是栈应用的典型例子。算术表达式有三种表现形式：前缀表达式、中缀表达式和后缀表达式。中缀表达式求值通常采用的算法是算符优先法。实现这个算法需使用两个栈，一个是运算符栈，一个是存放运算对象和运算结果的栈。

算法的基本思想如下。

(1) 设立运算符栈和操作数栈；

(2) 预设运算符栈的栈底为#；

(3) 若当前字符是操作数，则进操作数栈；

(4) 若当前运算符的优先权高于栈顶运算符，则进运算符栈；

(5) 否则，退出栈顶运算符作相应运算；

(6) "("对它前后的运算符起隔离作用，")"可视为自相应左括号开始的表达式的结束符。

运算符间的优先关系见表 3-1。

表 3-1 运算符间的优先关系

$\&_1$ \ $\&_2$	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

例如，有中缀表达式 5+2 * 25/5，下面来看看如何利用栈求它的值。

设 op 为运算符栈，初始时 op 为空；设 num 为操作数栈，初始时 num 栈底放置了一个#符。

(1) 自左至右扫描该表达式，读到的第一个对象是数字 5，将它进栈 num。

(2) 继续扫描表达式，读到的第二个对象为运算符+，它的运算优先级高于 op 栈栈顶元素#，将它入栈 op。

(3) 继续扫描表达式，读到的第三个对象为数字 2，将它进栈 num。

(4) 继续扫描表达式，读到的第四个对象为运算符 *，它的运算优先级高于 op 栈栈顶元素+，将它进栈 op。

(5) 继续扫描表达式，读到的第五个对象为数字 25，将它进栈 num。

(6) 继续扫描表达式，读到的第六个对象为运算符/，由于它的优先级不高于 op 栈栈顶元素 *，因此将它暂不进栈，让 op 栈栈顶元素 * 出栈，将 num 栈栈顶两个元素依次出栈，执行操作 2×25，结果 50 进栈 num；然后将"/"与 op 栈栈顶运算比较，它比+优先级高，所以将"/"进栈 op。

(7) 继续扫描表达式，读到的第七个对象为数字 5，将它进栈 num。

(8) 继续扫描表达式,已经是表达式末尾,让 op 栈栈顶运算/出栈,让 num 栈栈顶的两个元素依次出栈,执行操作 50/5,结果 10 进栈 num。

(9) 继续将 op 栈栈顶元素+出栈,将 num 栈栈顶两个元素依次出栈,执行 5+10,将结果 15 进栈 num。

(10) 至此,op 栈已到栈底(遇到对象#),num 栈底的元素即表达式所求值。

为了处理方便,编译程序常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运算对象之后,并且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。

如何将一个中缀表达式转换为后缀表达式? 可以将中缀表达式存放到一个数组中,利用栈来存放扫描中缀表达式数组时遇到的运算符,将形成的后缀表达式存放到一个数组当中,最后依次输出后缀表达式数组当中的元素即可。具体步骤如下。

(1) 自左向右依次扫描中缀表达式里的各项,如果遇到的是操作数,则直接将它写入到存放后缀表达式的数组里。

(2) 如果遇到的是左括号(,意味着表达式中一个新的计算层次的开始,于是将它进栈。

(3) 如果遇到的是运算符,就把它和当前栈栈顶元素进行优先级比较,如果它的优先级小于等于栈顶元素的优先级,那么让栈顶元素出栈,将出栈元素写入后缀表达式数组中,重复此过程,直到其优先级大于栈顶元素的优先级,将其入栈。

(4) 如果遇到的是右括号),将栈顶元素出栈,同时写入后缀表达式数组,直到栈顶元素为左括号(,让左括号出栈,但不写入后缀表达式数组。

(5) 重复上述步骤,直到遇到中缀表达式的结束标记#,于是依次弹出栈中的元素,写入到后缀表达式数组中,结束算法。

以上述中缀表达式 5+2*25/5 为例,说明一下如何利用栈和数组将中缀表达式转换为后缀表达式。

(1) 自左向右依次扫描中缀表达式,遇到的第一个对象是数字 5,将其存入后缀表达式数组;

(2) 继续扫描中缀表达式,遇到的第二个对象是运算符+,因初始操作符栈 ss 为空,将+入栈 ss;

(3) 继续扫描中缀表达式,遇到的第三个对象是数字 2,存入后缀表达式数组;

(4) 继续扫描中缀表达式,遇到的第四个对象是运算符*,将其与 ss 栈顶元素+进行比较,因*优先级高于+的优先级,故将*入栈 ss;

(5) 继续扫描中缀表达式,遇到的第五个对象是数字 25,存入后缀表达式数组;

(6) 继续扫描中缀表达式,遇到的第六个对象是运算符/,将其与 ss 栈顶元素*进行比较,因两者的优先级相同,故将*出栈 ss,存入后缀表达式数组,再将/入栈 ss;

(7) 继续扫描中缀表达式,遇到的第七个对象是数字 5,存入后缀表达式数组;

(8) 继续扫描中缀表达式,遇到表达式结束符#,表明表达式结束;依次出栈 ss 中元素存入后缀表达式数组,输出后缀表达式数组元素(5,2,25,*,5,/,+),即得到所求后缀表达式。

算法 3.12

```
Pfix(ss,s1[ ],s2[ ])                    //ss 顺序栈,s1[ ]存放中缀表达式,s2[ ]存放后缀表达式
```

```
{  int i = 0,j = 0;
   ss.top++;
   ss[ss.top] = '#';                    //初始 ss 栈中存放'#'
   while(s1[i]!= '#')
   {  if(s1[i]>= '0'&&s1[i]<= '9')      //遇到数字,存入数组 s2
         s2[j++] = s1[i];
      else if(s1[i] == '(')             //遇到左括号,进栈 ss
       {  ss.top++;
          ss[ss.top] = s1[i];
        }
      else if(s1[i] == ')')
      { while(ss[ss.top]!= '(')         //遇到右括号,让栈 ss 中左括号前的元素出栈
         s2[j++] = ss[ss.top--];
        ss.top--;                       //让左括号出栈
       }
      else if(is(s1[i]))                //遇到运算符,比较优先级
      { while(pri(ss[ss.top])>= pri(s1[i]))          //运算符优先级小于等于栈顶元素的情况
           s2[j++] = ss[ss.top--]; //栈顶元素出栈,存入 s2
        ss.top++;
        ss[ss.top] = s1[i];             //当前运算符进栈
      }
      i++;                              //扫描 s1 中下一个元素
   }
     while(ss[ss.top]!= '#')            //中缀表达式扫描结束
       s2[j++] = ss[ss.top--];          //ss 栈中运算符依次出栈存入 s2
   }
   is(op)                               //判断一个字符是否是运算符
   { switch(op)
     { case'+':
       case'-':
       case'*':
       case'/':
         return 1;                      //是运算符返回 1
       default:
         return 0;                      //不是运算符返回 0
     }
   }
   pri(op)                              //计算运算符的优先级
     { switch(op);
        { case'#': return -1;
          case'(': return 0;
          case'+':
          case'-': return 1;
          case'*':
          case'/': return 2;
          default: return -1;
       }
   }
```

因为后缀表达式中既无括号又无优先级的约束，计算它的值要比计算中缀表达式的值简单得多。只需使用一个操作数栈，自左向右扫描后缀表达式时，每遇到一个操作数就入栈保存，每遇到一个运算符就从栈中出栈两个操作数进行计算，将结果入栈，直到整个表达式结束，这时栈顶元素存放的就是最后的计算结果。

下面以上述得到的后缀表达式(5,2,25,*,5,/,+)为例，看一下如何借助栈实现后缀表达式求值。

(1) 自左向右扫描后缀表达式，遇到的第一个对象是数字5，将其入栈num；

(2) 继续扫描后缀表达式，遇到的第二个对象是数字2，将其入栈num；

(3) 继续扫描后缀表达式，遇到的第三个对象是数字25，将其入栈num；

(4) 继续扫描后缀表达式，遇到的第四个对象是运算符*，将num栈出栈两个操作数25、2，计算2×25，得到数字50，将50入栈num；

(5) 继续扫描后缀表达式，遇到的第五个对象是数字5，将其入栈num；

(6) 继续扫描后缀表达式，遇到的第六个对象是运算符/，将num栈出栈两个操作数5、50，计算50/5，得到数字10，将10入栈num；

(7) 继续扫描后缀表达式，遇到的第七个对象是运算符+，将num栈出栈两个操作数10、5，计算5+10，得到数字15，将15入栈num；

(8) 继续扫描后缀表达式，遇到结束符#，表达式结束，出栈num栈顶元素，即最后计算结果是15。

算法 3.13

```
calexp(st,s[ ])                    //st 为顺序栈,s[ ]为存放后缀表达式的数组
{   int i = 0;
    int x1,x2;
    st.top++;
    st[st.top] = '#';              //将'#'入栈
    while(s[i]!= '#')
    {   if(s[i]>= '0'&&s[i]<= '9')  //扫描后缀表达式遇到的对象为数字
        st.top++;
        st[st.top] = s[i];         //将数字入栈
        i++;
    }
    else if(s[i] = '+')            //扫描后缀表达式遇到运算符'+'
    {   x2 = st[st.top--];
        x1 = st[st.top--];         //将栈顶两个元素出栈
        st.top++;
        st[st.top] = x1 + x2;      //将栈顶两个元素进行加法运算,结果入栈
        i++;
    }
    else if(s[i] = '-')            //扫描后缀表达式遇到运算符'-'
    {   x2 = st[st.top--];
        x1 = st[st.top--];         //将栈顶两个元素出栈
        st.top++;
        st[st.top] = x1 - x2;      //将栈顶两个元素进行减法运算,结果入栈
        i++;
```

```
        }
        else if(s[i] = '*')                    //扫描后缀表达式遇到运算符'*'
        {   x2 = st[st.top--];
            x1 = st[st.top--];                 //将栈顶两个元素出栈
            st.top++;
            st[st.top] = x1 * x2;              //将栈顶两个元素进行乘法运算,结果入栈
            i++;
        }
        else if(s[i] = '/')                    //扫描后缀表达式遇到运算符'/'
        {   x2 = st[st.top--];
            x1 = st[st.top--];                 //将栈顶两个元素出栈
            st.top++;
            st[st.top] = x1/x2;                //将栈顶两个元素进行除法运算,结果入栈
            i++;
        }
        return s(0);
    }
}
```

应用举例 2　括号匹配问题

假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，即([]())或[([][])]等为正确的格式，[(])或([())或(())均为不正确的格式。检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。例如，考虑下列括号序列：

```
[  (  [  ]  [  ]  )  ]
1  2  3  4  5  6  7  8
```

分析可能出现的不匹配的情况：

(1) 到来的右括号并非是所“期待”的；

(2) 到来的是“不速之客”；

(3) 直到结束，也没有到来所“期待”的。

算法 3.14

```
int match(char *exps)                          //exps 存放表达式
{   char st[MaxSize];
    int nomatch = 1,top = -1,i = 0;
    while(exps[i]!= '\0'&&nomatch == 1)        //遍历表达式 exps
    {   switch(exps[i])
        {
        case'(': case'[': case'{';             //左括号进栈
            top++; st[top] = exps[i];break;
        case')':                               //遇到)
            if(st[top] == '(') top--;          //判断栈顶是否为(
            else nomatch = 0;
            break;
        case']':                               //遇到]
            if(st[top] == '[') top--;          //判断栈顶是否为[
            else nomatch = 0;
            break;
```

```
        case'}':                                    //遇到}
            if(st[top] == '{') top -- ;             //判断栈顶是否为{
            else nomatch = 0;
            break;
        default:                                    //跳过其他字符
            break;
        }
        i++;
    }
   if(nomatch == 1&&top == -1)                      //栈空且符号匹配则返回 1
      return 1;
   else return 0;                                   //否则返回 0
}
```

应用举例 3　栈与递归

递归程序是通过调用自身来完成与自身要求相同的子问题的求解。在计算机内递归的过程是利用栈的技术来实现的。实现递归的算法简称为递归算法。

当一个程序调用另一个程序时,在运行被调用函数之前,要完成以下几件事情。

(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。

从被调用函数返回调用函数之前,应该完成以下几件事情。

(1) 保存被调函数的计算结果;

(2) 释放被调函数的数据区;

(3) 依照被调函数保存的返回地址将控制转移到调用函数。

可以看出它们的规则是:后调用先返回,可以利用栈技术实现。

汉诺塔问题是比较典型的递归问题,设有三座塔,分别命名为 A,B,C,在塔 A 上插有 n 个直径各不相同,从小到大编号为 1,2,3,…,n 的圆盘,编号大的圆盘放在塔的底部,编号小的圆盘在顶部。现在要求将圆盘从塔 A 移到塔 C 上,并且任按原来的顺序叠放,移动的过程中遵循以下原则。

(1) 每次只能移动一个圆盘;

(2) 圆盘可以插入在 A、B、C 的任一个塔座上;

(3) 任何时候都不能将一个较大的圆盘放在较小的圆盘上。

算法 3.15

```
void hanoi (int n, char x, char y, char z)
// 将塔座 x 上按直径由小到大且至上而下编号为 1 至 n
// 的 n 个圆盘按规则搬到塔座 z 上,y 可用作辅助塔座
{
  if (n == 1)
   move(x, 1, z);                                   // 将编号为 1 的圆盘从 x 移到 z
  else {
    hanoi(n - 1, x, z, y);                          // 将 x 上编号为 1 至 n - 1 的圆盘
                                                    // 移到 y, z 作辅助塔
```

```
        move(x, n, z);                          // 将编号为 n 的圆盘从 x 移到 z
        hanoi(n-1, y, x, z);                    // 将 y 上编号为 1 至 n-1 的圆盘
                                                // 移到 z, x 作辅助塔
    }
}
```

3.2 队列

3.2.1 队列的定义及操作特性

1. 队列的定义

队列(Queue)也是一种操作受限的线性表,它仅允许在表的一端进行插入操作,在表的另一端进行删除操作。允许插入(也称入队)的一端称为队尾(Rear),允许删除(也称出队)的一端称为队头(Front)。所以又把队列称为先进先出(First In First Out,FIFO)线性表,如图 3-4 所示。

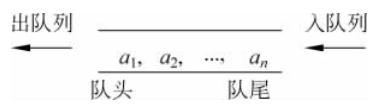


图 3-4 队列的示意图

在日常生活中有许多队列的例子,如人们非常熟悉的排队购物,排在队伍前面的顾客是队头,排在队伍后面的顾客为队尾;排在队头的顾客买完东西先离开,相当于出队,后来的顾客排在队尾等待购物,相当于入队。

2. 队列的操作特性

队列的特性为“先进先出”原则。

在队列的队尾插入一个元素的操作称为入队或进队,在队列的队头删除一个元素的操作称为出队或退队,即在队列中,最先入队的元素最先出队,最后入队的元素将最后出队。如果队列中没有任何元素,则称为空队列。

队列的基本运算主要有以下几种。

(1) 队列的初始化运算 InitQueue(qu),建立一个空队 qu。

(2) 销毁队运算 DestroyQueue(qu),释放队列 qu 占用的内存空间。

(3) 入队运算 EnQueue(qu,x),将 x 插入到 qu 的队尾。

(4) 出队运算 DeQueue(qu,x),将队列 qu 的队头元素出队并赋给 x。

(5) 取队头元素运算 GetQueue(qu,x),取出队列 qu 的队头元素并赋给 x,但该元素不出队。

(6) 判队空运算 QueueEmpty(qu),判断队列 qu 是否为空。

例如,设队列 $Q(a,b,c,d)$,其中元素 a 为队头元素,元素 d 为队尾元素。若元素 a 出队,则队列变为 $Q(b,c,d)$,此时元素 b 为队头元素;若新元素 e 入队,则队列变为 $Q(b,c,d,e)$,此时元素 e 为队尾元素;若元素 b,c,d,e 均出队,则队列 $Q()$为空队,如图 3-5 所示。

队列也有两种存储表示方法,队列的顺序存储结构称为顺序队列,队列的链式存储结构称为链队列。两种结构各有优势,各能实现不同的运算。

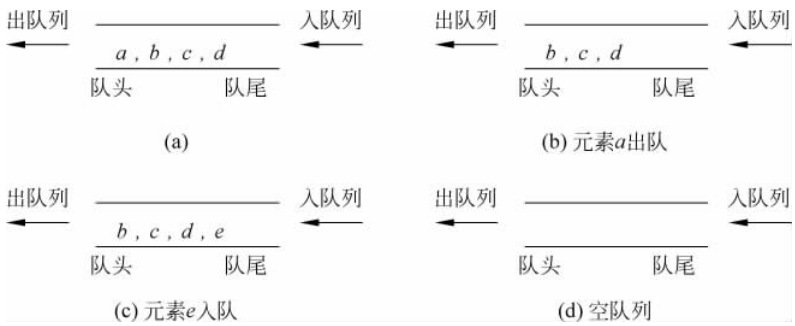


图 3-5　元素入队、出队示意图

3.2.2　队列的顺序存储结构及其基本运算的实现

1. 队列的顺序存储结构

队列的顺序存储结构称为顺序队。顺序队也是用一维数组来存放队列元素，除此之外还需附设两个指针 front 和 rear 分别指示队列头元素和队列尾元素的位置。通常约定队尾指针指示队尾元素的当前位置，队头指针指示队头元素的前一个位置。

顺序队的类型定义：

```
#define MaxSize 20                    //指定队列的容量
typedef struct
{ ElemType data[MaxSize];             //保存队中元素,这里假设 ElemType 为 int 类型
   int front,rear;                    //队头和队尾指针
}SqQueue;
```

顺序队的定义为一个结构类型，该类型变量有三个域：data、front、rear。其中，data 为存储队列中元素的一维数组。队头指针 front 和队尾指针 rear 定义为整型变量，实际取值范围是 0～MaxSize－1。

在初始化队列时，sq. front＝sq. rear＝－1；新元素入队时，将 sq. rear＋＋，新元素存入 sq. rear＋＋位置；出队时，将 sq. front＋＋，将 sq. front＋＋中的元素取出。依次操作下去，可以得出：当 rear＝ MaxSize－1 时，表示队满“真溢出”；当 front＝＝rear 时，表示队空。

例如，有如下顺序队列的操作，如图 3-6 所示。

```
MaxSize = 6
```

(1) 初始状态：sq. front ＝sq. rear＝－1；

(2) a_1、a_2、a_3入队；

(3) a_1、a_2出队；

(4) a_3出队，此时队列为空，有 sq. rear＝sq. front；

(5) a_4、a_5、a_6入队(此时已超出 MaxSize，不能再入队了)；

(6) a_7再入队时，此时队列的实际空间虽然不满，但却不能入队，产生“假溢出”现象；

(7) a_4、a_5、a_6出队，此时队列的实际空间为空，但却不能入队，产生“严重假溢出”现象，此时也无法再扩大分配存储空间，因为队列的实际可用空间并未满。

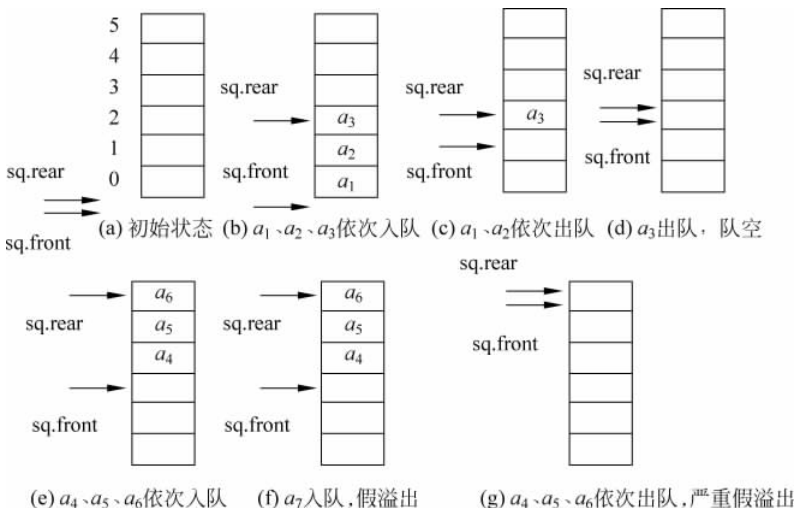


图 3-6 队列顺序存储示意图

为了解决上述矛盾现象，可以把数组的前端和后端连接起来，形成一个环形的表，即把存储队列元素的表从逻辑上看成一个环，这个环形的表叫作循环队列或环形队列，如图 3-7 所示。

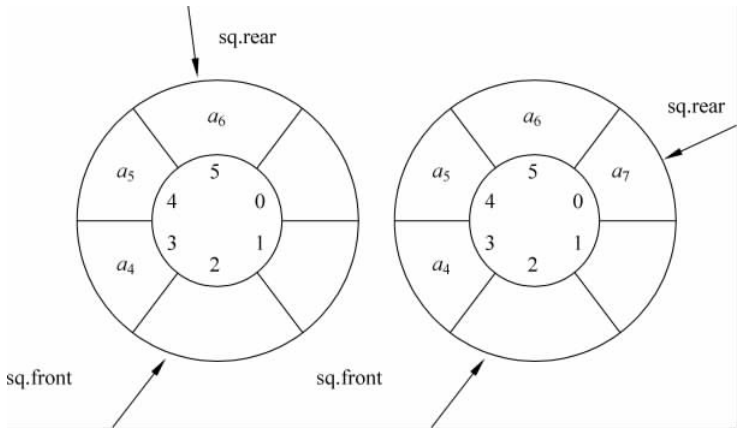


图 3-7 循环队列示意图

循环队列 sq 的 4 个要素如下。

(1) 队空条件：sq. front==sq. rear。

(2) 队满条件：(sq. rear+1)%MaxSize==sq. front。

(3) 进队操作：sq. rear 循环进 1；rear=(rear+1)%MaxSize 元素进队。

(4) 出队操作：sq. front 循环进 1；front=(front+1)%MaxSize 元素出队。

2. 循环队列的基本运算

循环队列的基本运算包括循环队列初始化、销毁队列、进队，出队、取队头元素和判断队空等。

1）循环队列的初始化运算算法

其主要操作是：sq.front＝sq.rear＝0。

算法 3.16

```
void InitQueue(SqQueue &sq)                    //sq 为引用型参数
{
  sq.rear = sq.front = 0;                      //指针初始化
}
```

2）销毁队列运算算法

这里顺序队的内存空间是由系统自动分配的，在不再需要时由系统自动释放其空间。

3）循环队列的进队运算算法

其主要操作是：先判断队列是否已满，若不满，让队尾指针循环进 1，在该位置存放 x。

算法 3.17

```
int EnQueue (SqQueue &sq, ElemType x) {
    if (( sq.rear + 1) % MaxSize == sq.front)  //队满上溢出
    return 0;
    sq.rear = (sq.rear + 1) % MaxSize;         //队尾循环进 1
    sq.data[sq.rear] = x;
    return 1;
}                                              //EnQueue
```

4）循环队列的出队运算算法

其主要操作是：先判断队列是否为空，若不空，让队头指针循环进 1，将该位置的元素值赋给 x。

算法 3.18

```
int DeQueue (SqQueue &sq, ElemType &x)         //x 为引用型参数
{   if (sq.rear == sq.front)                   //队空下溢出
       return 0;
   sq.front = (sq.front + 1) % MaxSize;        //队头循环进 1
   x = sq.data[sq.front];
   return 1;
}
```

5）取队头元素运算算法

其主要操作是：先判断队列是否为空，若不空，将队头指针上一个位置的元素值赋给 x。

算法 3.19

```
int GetHead(SqQueue sq,ElemType &x)            //x 为引用型参数
{   if(sq.rear == sq.front)                    //队空下溢出
       return 0;
   x = sq.data[(sq.front + 1) % maxsize];
   return 1;
}
```

6）判断队空运算算法

其主要操作是：若队列为空，则返回 1；否则返回 0。

算法 3.20

```
int QueueEmpty(SqQueue sq)
{   if(sq.rear = = sq.front)   return 1;
    else return 0;
}
```

当顺序队的基本运算算法设计好后，给出主函数算法，调用这些基本算法。

```
#include <stdio.h>
#include "SqQueue.h"                          //包含前面的顺序队基本运算函数
void main()
{   SqQueue sq;                               //定义一个顺序队 sq
    ElemType e;
    printf("初始化队列\n");
    InitQueue(sq);
    printf("队列 %s\n",(QueueEmpty(sq) == 1?"空":"不空"));
    printf("a 进队\n");EnQueue(sq,'a');
    printf("b 进队\n");EnQueue(sq,'b');
    printf("c 进队\n");EnQueue(sq,'c');
    printf("d 进队\n");EnQueue(sq,'d');
    printf("队 %s\n",(QueueEmpty(st) == 1?"空":"不空"));
    GetHead(sq,e);
    printf("队头元素：%c\n",e);
    printf("出队次序：");
While(!QueueEmpty(sq))                        //队不空循环
    {   DeQueue(sq,e);                        //出队元素 e
        printf("%c",e);                       //输出元素 e
    }
printf("\n");
DestroyQueue(sq);
}
```

3.2.3 队列的链式存储结构及其基本运算的实现

队列的链式存储结构称为链队，它实际上是一个同时带队头指针 front 和队尾指针 rear 的单链表。队头指针指向队头结点，队尾指针指向队尾结点即单链表的最后一个结点，并将队头和队尾指针结合起来构成链队结点，如图 3-8 所示。

其中，链队的数据结点类型定义如下。

```
typedef struct QNode
{
    ElemType data;                            //存放队中元素
    struct Node *next;                        //指向下一个结点的指针
}QType                                        //链队中结点的类型
```

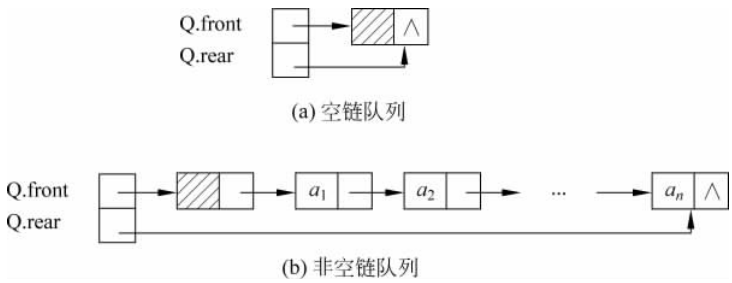


图 3-8 链队列示意图

链队结点的类型定义如下。

```
typedef struct qptr
{
  QType *front;                          //队头指针
  QType *rear;                           //队尾指针
}LinkQueue;                              //链队结点类型
```

归纳起来,链队 lq 的 4 个要素如下。

(1) 队空条件: lq->front==NULL。

(2) 队满条件: 不考虑,因为每个结点都是动态分配的。

(3) 进队操作: 创建结点 * p,将其插入到队尾,并由 lq->rear 指向它。

(4) 出队操作: 删除队头结点。

链队列的基本运算包括队列初始化、销毁队列、进队运算,出队、取队头元素,判断队空等。

1. 链队初始化运算算法

其主要操作是: 创建链队结点,并置该结点的 rear 和 front 为 NULL。

算法 3.21

```
void InitQueue(LinkQueue *&lq)           //lq 为引用型参数
{   lq=(LinkQueue *)malloc(sizeof(linkQueue));
    lq->rear=lq->front=NULL;             //初始时队头和队尾指针均为空
}
```

2. 销毁队列运算算法

链队的所有结点空间都是通过 malloc 函数分配的,在不再需要时需通过 free 函数释放所有结点空间。

算法 3.22

```
void DestroyQueue(LinkQueue *&lq)
{  QType *pre=lq->front, *p;
   if(pre!=NULL);                        //非空队的情况
   {if(pre==lq->rear)                    //只有一个数据结点的情况
       free(pre);                        //释放 *pre 结点
```

```
    else                                    //有两个或多个数据结点的情况
    { p = pre -> next;
      while(p!= NULL)
       { free(pre);                         //释放 * pre 结点
         pre = p;p = p -> next;             //pre、p 同步后移
       }
       free(pre);                           //释放尾结点
     }
    }
    free(lq);                               //释放链队结点
}
```

3. 链队的进队运算算法

其主要操作是：创建一个新结点，将其链接到链队的末尾，并由 rear 指向它。

算法 3.23

```
void EnQueue(LinkQueue * &lq , ElemType x)      //lq 为引用型参数
{   QType * s;
    s = (QType * ) malloc (sizeof (QType));     //创建新结点,插入到链队的末尾
    s -> data = x;s -> next = NULL;
    if (lq -> front == NULL)                    //原队为空队的情况
      lq -> rear = lq -> front = s;             //front 和 rear 均指向 * s 结点
    else                                        //原队不为空队的情况
    { lq -> rear -> next = s;                   //将 * s 链到队尾
     lq -> rear = s;                            //rear 指向它
    }
}
```

4. 链队的出队运算算法

其主要操作是：将 * front 结点的 data 域值赋给 x，并删除该结点。

算法 3.24

```
int DeQueue(LinkQueue * &lq , ElemType &x)      //lq,x 为引用型参数
{   QType * p;
    if (lq -> front == NULL);                   //原队为空队的情况
       return 0;
    p = lq -> front;                            //p 指向队头结点
    x = p -> data;                              //取队头元素值
    if (lq -> rear == lq -> front)              //若原队列中只有一个结点,删除后队列变空
      lq -> rear = lq -> front = NULL;
    else                                        //原队列有两个或两个以上结点的情况
      lq -> front = lq -> front -> next;
    free(p);
    return 1;
}
```

5. 链队的取队头元素运算算法

其主要操作是：将 * front 结点的 data 域值赋给 x。

算法 3.25

```
int GetHead(LinkQueue * lq , ElemType &x)        //x 为引用型参数
{   if (lq->front == NULL)                       //原队列为空的情况
       Return 0;
    x = lq->front->data;
    return 1;
}
```

6. 链队的判空运算算法

其主要操作是：若链队为空，则返回 1；否则返回 0。

算法 3.26

```
int QueueEmpty(LinkQueue * lq)
{   if (lq->front == NULL) return 1;             //队空返回 1
    else return 0;                               //队不空返回 0
}
```

当链队的基本运算算法设计好后，给出主函数算法，调用这些基本算法。

```
# include<stdio.h>
# include"LinkQueue.h"                           //包含前面的链队基本运算函数
void main()
{  LinkQueue * lq;                               //定义一个链队 lq
   ElemType e;
   printf("初始化队列\n");
   InitQueue(lq);
   printf("队 %s\n",(QueueEmpty(lq) == 1?"空":"不空"));
   printf("a 进队\n");EnQueue(lq,'a');
   printf("b 进队\n");EnQueue(lq,'b');
   printf("c 进队\n");EnQueue(lq,'c');
   printf("d 进队\n");EnQueue(lq,'d');
   printf("队 %s\n",(QueueEmpty(lq) == 1?"空":"不空"));
   GetHead(lq,e);
   printf("队头元素: %c\n",e);
   printf("出队次序: ");
   while(!QueueEmpty(lq))                        //队不空循环
   { DeQueue(lq,e);                              //出队元素 e
    printf("%c",e);                              //输出元素 e
   }
   printf("\n");
   DestroyQueue(lq);
}
```

3.2.4 队列的应用举例

队列满足“先进先出”的原则，所以一些满足“先进先出”原则的问题都可以采用队列作

为数据结构来解决问题。在计算机内部,队列的结构是必不可少的。例如在计算机系统中,解决主机与外设打印机数据传递速度不匹配的问题。

在打印机打印时,通常在内存中设置一个打印数据缓冲区。缓冲区是一块连续的存储空间,把它设计成循环队列结构,主机把要打印的数据依次输入到缓冲区中,输入满后就暂停输出,主机此时去进行其他工作。打印机就从缓冲区按照先进先出的原则依次打印数据。这样就解决了主机与打印机之间速度不匹配的问题,从而提高了计算机的工作效率。

3.3 栈和队列的应用案例

应用案例:回文问题

【案例说明】

常见的一种形如 abcba 的文字,这种文字从左到右读和从右到左读结果是一样的,这种文字称为回文。试设计一个程序可以判断给定的一段文字是否是回文。

【案例目的】

(1) 掌握栈的特性和表示方法。

(2) 掌握队列的特性和表示方法。

(3) 熟练掌握栈与队列的综合应用。

【实现要点】

栈的后进先出以及队列的先进先出,可以结合这两种结构来实现需要的功能,即将文字分别入队和入栈,然后依次输出判断是否有不相同的字符,一旦发现有不相同的文字证明不是回文。

【代码实现】

```
#include<stdio.h>
#include<stdlib.h>
#include<iostream.h>
#define STACKINCREMENT 10
#define STACK_INIT_SIZE 100
//队列的链式存储结构
typedef struct QNode{
    char data;
    struct QNode *next;
}QNode, *Queueptr;
typedef struct{
    Queueptr front;
    Queueptr rear;
}LinkQueue;
//栈的存储结构
typedef struct{
    char *base;
    char *top;
    int stacksize;
}SqStack;
void InitStack(SqStack &S) {
    //构造一个空栈
```

```
    S.base = (char *)malloc(STACK_INIT_SIZE * sizeof(char));
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    }
void Push(SqStack &S,char e) {
    //插入 e 为新的栈顶元素
    if(S.top - S.base >= S.stacksize){
      S.base = (char * ) realloc (S.base,(S.stacksize + STACKINCREMENT) * sizeof(char));
      if(!S.base) printf("存储分配失败!");
      S.top = S.base + S.stacksize;
      S.stacksize += STACKINCREMENT;
      }
    * S.top++ = e;
    }
char Pop(SqStack &S,char &e){
    //若栈不空,则删除 S 的栈顶元素,用 e 返回其值
    if(S.top == S.base) return 0;
    e = * -- S.top;
    return e;
    }
void InitQueue(linkQueue &Q) {
    //构造一个空队列
    Q.front = Q.rear = (Queueptr)malloc(sizeof(QNode));
    if(!Q.front )
    printf("存储分配失败!");
    Q.front -> next = NULL;
    }
void EnQueue(LinkQueue &Q,char e){
    //插入元素 e 为 Q 的新的队尾元素
    Queueptr p;
    p = (Queueptr) malloc (sizeof(QNode));
    if(!p) printf ("存储分配失败!");
    p -> data = e; p -> next = NULL;
    Q.rear -> next = p;
    Q.rear = p;
    }
char DeQueue(LinkQueue &Q,char &e){
    //若队列不空,则删除 Q 的对头元素,用 e 返回其值
    Queueptr p;
    p = Q.front -> next;
    e = p -> data;
    Q.front -> next = p -> next;
    if(Q.rear == p) Q.rear = Q.front;
    free(p);
    return e;
    }
void Main()
    {char a,b,c;
    SqStack S; LinkQueue Q;
    InitStack(S);
    InitQueue(Q);
    printf("请输入要判断的字符: ");
    scanf("%c",&c);
    while(c!= '#') {Push(S,c);EnQueue(Q,c); getchar(c);}
```

```
while(!((S.top == S.base)&&a == b)
  {Pop(S,a); DeQueue(Q,b);}
if(a!= b) printf ("This isn't a cycle\n");
else
    printf ("This is a cycle\n");
}
```

案例扩展：停车场管理系统

【案例说明】

假设有一个可以停放 n 辆汽车的狭长停车场，它只有一个大门可以供车辆进出。车辆到达停车场时间的早晚依次从停车场最里面向大门口处停放(最先到达的第一辆车放在停车场最里面)。如果停车场已放满 n 辆车，则后来的车辆只能停在停车场大门外的便道上等待，一旦停车场里有车开走，则排在便道上的第一辆车就进入停车场。若停车场内有某辆车要开走，在它之后进入停车场的车都必须先退出停车场，为它让路，待其开出停车场后，这些车再依原来的次序进场。每辆车离开停车场时，都应根据其在停车场的逗留时间交费。如果停留在便道上的车未进停车场就要离去，允许其离去，不收停车费，并且仍然保持在便道上等待的车辆顺序。编制一程序模拟停车场的管理。

利用栈和队列模拟此过程，算法思想如下。

停车场的管理流程如下。

(1) 当车辆要进入停车场时，检查停车场是否已满，如果未满则车辆进入停车场；如果停车场已满，则车辆进入便道等候。

(2) 当车辆要求出栈时，先让在它之后进入停车场的车辆退出停车场为它让路，再让该车退出停车场，让路的所有车辆再按其原来进入停车场的次序进入停车场。之后，再检查在便道上是否有车等候，有车则让最先等待的那辆车进入停车场。

【代码】略

小结

(1) 栈是一种操作受限的特殊线性表，它仅允许在线性表的同一端进行插入和删除操作，是一种后进先出的线性表；它的实现有顺序栈和链栈。

(2) 栈在日常生活和计算机程序设计中应用广泛。

(3) 队列也是一种操作受限的线性表，它仅允许在线性表的一端进行插入，在另一端进行删除，是一种先进先出的线性表；它的实现有循环队列和链队列。

重点、难点例题解析

一、选择题

1. 一个栈的入栈序列为 1，2，3，…，n，其出栈序列是 p_1，p_2，p_3，…，p_n。若 $p_2=3$，则 p_3 可能取值的个数是(　　)。【2013 年考研真题】

A. $n-3$　　B. $n-2$　　C. $n-1$　　D. 无法确定

【解答】 C

【分析】 本题主要考察对入栈出栈操作的理解。若 $p_2=3$，则 p_3 可以为 4…n 中任何一个，只要再考察 p_3 可不可为 1 或 2，均可，故 p_3 可能个数为 $n-1$。

2. 假设栈初始为空，将中缀表达式 $a/b+(c*d-e*f)/g$ 转化为等价后缀表达式的过程中，当扫描到 f 时，栈中的元素依次是(　　)。【2014 年考研真题】

A. +(*-　　B. +(-*　　C. /+(*-*　　D. /+-*

【解答】 B

【分析】 本题主要考察利用栈的数据结构将中缀表达式转化为后缀表达式的过程。

3. 循环队列存放在一维数组 $A[0..M-1]$中，end1 指向队头元素，end2 指向队尾元素的后一个位置。假设队列两端均可进行入队和出队操作，队列中最多能容纳 $M-1$ 个元素，初始时为空。下列判断队空和队满的条件中，正确的是(　　)。【2014 年考研真题】

A. 队空：end1==end2；队满：end1==(end2+1)mod M

B. 队空：end1==end2；队满：end2==(end1+1)mod($M-1$)

C. 队空：end2==(end1+1)mod M；队满：end1==(end2+1)mod M

D. 队空：end1==(end2+1)mod M；队满：end2==(end1+1)mod($M-1$)

【解答】 A

【分析】 本题主要考察对于循环队列数据结构判空和队满的判断。

4. 已知程序如下：

```
int s(int n)
{ return (n<=0) ? 0 : s(n-1) +n; }
void main()
{ cout<<s(1); }
```

程序运行时使用栈来保存调用过程的信息，自栈底到栈顶保存的信息依次对应的是(　　)。【2014 年考研真题】

A. main()->s(1)->s(0)　　B. s(0)->s(1)->main()

C. main()->s(0)->s(1)　　D. s(1)->s(0)->main()

【解答】 D

【分析】 本题主要考察程序调用过程中对栈的利用。

二、简答题

1. 若用一个大小为 6 的数组来实现循环队列，且当前 rear 和 front 的值分别为 0 和 3，当从队列中删除一个元素，再加入两个元素后，rear 和 front 的值分别为多少？

【解答】 2 和 4

【分析】 front 为 3，出队一个元素，front 变为 4；rear 为 0 时，再入队两个元素后，rear 变为 2。

2. 写出下列中缀表达式的后缀表达式。

(1) A*B*C

(2) (A+B)*C-D

(3) A * B+C/(D-E)

(4) (A+B) * D+E/(F+A * D)+C

【解答】

(1) ABC **

(2) AB+C * D-

(3) AB * CDE-/+

(4) AB+D * EFAD * +/+C+

3. 有 5 个数依次进栈：1,2,3,4,5。在各种出栈的序列中，以 3,4 先出的序列有哪几个？(3 在 4 之前出栈。)

【解答】 34215,34251, 34521

【分析】 3、4 出栈后，栈内有 2、1，栈外有 5。5 可以出现在 2 前、2 后和 1 前、1 后的任一位置上。

综合练习

一、选择题

1. 设有一个栈，元素的进栈次序为 A,B,C,D,E，下列不可能的出栈序列为(　　)。

A. A,B,C,D,E　　B. B,C,D,E,A

C. E,A,B,C,D　　D. E,D,C,B,A

2. 在一个具有 n 个单元的顺序栈中，假定以地址低端(即 0 单元)作为栈底，以 top 作为栈顶指针，当作出栈处理时，top 变化为(　　)。

A. top 不变　　B. top=0　　C. top--　　D. top++

3. 在具有 n 个单元的顺序存储的循环队列中，假定 front 和 rear 分别为队头指针和队尾指针，则判断队满的条件为(　　)。

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

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

4. 在具有 n 个单元的顺序存储的循环队列中，假定 front 和 rear 分别为队头指针和队尾指针，则判断队空的条件为(　　)。

A. rear%n==front　　B. front+1=rear

C. rear==front　　D. (rear+1)%n=front

5. 在一个链队列中，假定 front 和 rear 分别为队首和队尾指针，则删除一个结点的操作为(　　)。

A. front=front->next　　B. rear=rear->next

C. rear=front->next　　D. front=rear->next

二、填空题

1. 线性表、栈和队列都是(　　)结构，可以在线性表的(　　)位置插入和删除元素；对于栈只能在(　　)位置插入和删除元素；对于队列只能在(　　)位置插入元素和在(　　)

位置删除元素。

2. 对于一个栈作进栈运算时，应先判别栈是否为(　　)；作退栈运算时，应先判别栈是否为(　　)；当栈中元素个数为 m 时，作进栈运算时发生上溢，则说明栈的可用最大容量为(　　)。为了增加内存空间的利用率和减少发生上溢的可能性，由两个栈共享一片连续的内存空间时，应将两栈的(　　)分别设在这片内存空间的两端，这样只有当(　　)时才产生上溢。

3. 设有一空栈，现有输入序列 1,2,3,4,5，经过 push，push，pop，push，pop，push，push 后，输出序列是(　　)。

4. 无论对于顺序存储还是链式存储的栈和队列来说，进行插入或删除运算的时间复杂度均相同为(　　)。

三、应用题

1. 假定有 4 个元素 A，B，C，D 依次进栈，进栈过程中允许出栈，试写出所有可能的出栈序列。

2. 什么是队列的上溢现象？一般有几种解决方法？试简述之。

四、算法设计题

假定用一个单循环链表来表示队列(也称为循环队列)，该队列只设一个队尾指针，不设队首指针，试编写下列各种运算的算法。

(1) 向循环链队列中插入一个元素值为 x 的结点；

(2) 从循环链队列中删除一个结点。