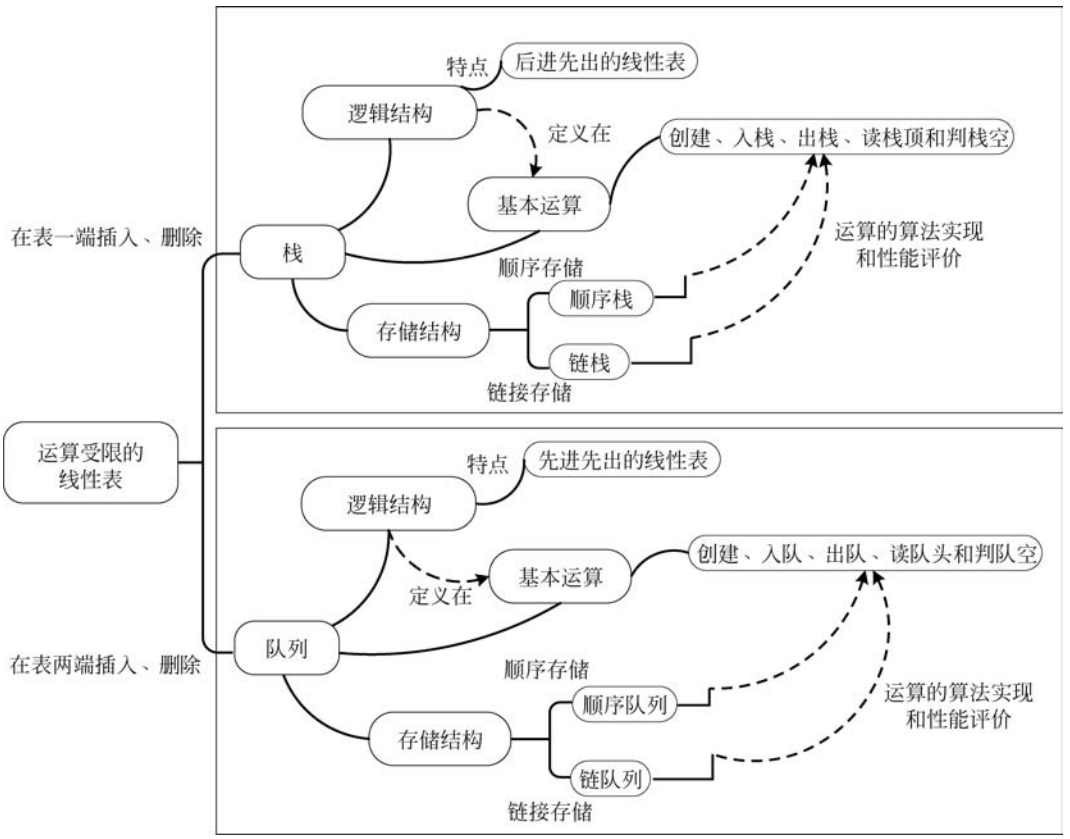
第3章 栈和队列

【学习目标】

- 理解栈和队列的概念、运算特点。
- 掌握栈和队列的存储以及运算的实现。
- 理解栈和队列在解决实际问题中的应用。
- 综合运用栈或队列解决实际应用问题。

【思维导图】



栈和队列是程序设计中广泛应用的两种数据结构。从数据元素间的逻辑关系看，栈和队列也是线性表，只是栈和队列的运算限定在表端完成，因此可以说栈和队列是两种特殊的线性表，其特殊性表现在运算受到限制。

3.1 栈

3.1.1 栈的定义及运算

栈(stack)是运算受限的线性表,限制它的插入和删除操作仅在表的一端进行。允许插入、删除的一端称为**栈顶**(top),另一端称为**栈底**(bottom)。当栈中的元素个数 $n=0$ 时称为**空栈**。向栈顶插入新元素称为**进栈**或**入栈**,从栈顶删除元素称为**出栈**或**退栈**。

假设栈 $S=(a_1,a_2,\cdots,a_n)$,如图 3.1 所示。栈中 a_1 为栈底元素,a_n 为栈顶元素。栈中元素进栈的顺序是 $a_1,a_2,\cdots,a_n$,若 n 个元素进栈后,再依次出栈,则出栈时其顺序为 $a_n,\cdots,a_2,a_1$。每次进栈操作总是将新元素放到栈顶,出栈操作是删除栈顶元素,即最后进栈的元素会最先出栈,而最先进栈的元素被放在栈的底部,最后才能出栈。由于元素是按后进先出的原则出入栈,因此栈又称为后进先出(last in first out,LIFO)的线性表,简称 **LIFO 表**。

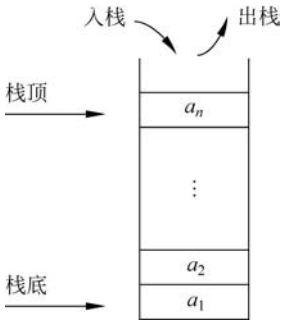


图 3.1 栈的图示

在实际应用中,有很多符合栈的特点的例子。例如,限定只能在一边进行放入或取出的一摞盘子或书、单行车道等。在计算机系统中,也有很多应用栈的例子。例如,Word、Photoshop 等很多软件中都具有“撤销”功能,用以恢复误操作,“撤销”时恢复内容的顺序与“操作”时输入内容的顺序恰好相反,即最先恢复的是最近的一次操作。因此,可以用栈来保存一定数量的操作,以方便“撤销”功能对操作顺序的要求。同理,浏览器中的“后退”按钮,单击后可以按访问顺序的逆序加载浏览过的页面,也是用栈的方式实现的。运行程序时,计算机操作系统对存储空间分配采取的方式之一就是栈式存储分配,即把程序运行时的数据存储空间组织成一个栈,按栈的方式进行分配和回收空间。每当调用一个函数时,在栈顶分配一块存储区域,每当一个函数被调用执行完毕返回时,释放它的存储区域。除此之外,计算机系统中还有很多应用栈的例子,读者可以自己列举。总之,栈中元素除了具有线性关系外,还具有后进先出的特点。所以,在应用时要根据这两个特点决定是否使用栈。

栈的基本运算有 5 种,定义如下:

(1) 初始化栈 initStack(s)。构造了一个空栈 s。

(2) 判栈空 empty(s)。若栈 s 为空栈,则返回值为“真”(1),否则返回值为“假”(0)。

(3) 入栈 push(s,x)。在栈 s 的顶部插入一个新元素 x,x 成为新的栈顶元素。

(4) 出栈 pop(s)。删除栈 s 的栈顶元素。

(5) 读栈顶元素 top(s)。栈顶元素作为结果返回,不改变栈的状态。

与线性表运算的定义相比,在栈的运算定义中,没有查找运算。因为栈是一种运算受限的线性表,它的插入、删除操作都只能在栈顶完成,即操作位置固定。在实际问题中需要使用的数据也一定是栈顶元素,根本不需要在栈的中间存取元素,因此不需要查找运算,这也是栈与线性表的不同之处。

由于栈的逻辑结构是线性表,因此线性表上的常用存储方式也适用于栈。采用顺序方

式存储的栈称为**顺序栈**(sequential stack),采用链接方式存储的栈称为**链栈**(linked stack)。

3.1.2 顺序栈及运算的算法实现

顺序栈类似于顺序表,利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素,在计算机高级语言中用一个预设的足够长的一维数组来实现;同时,设一个整型变量top来指明当前栈顶元素的位置,top也称为**栈顶指针**。通常将数组和栈顶指针top封装在一个结构体中。显然,顺序栈和顺序表的存储结构是一样的,只不过是将last换成top。

顺序栈存储结构C语言描述如下:

```
#define MAXSIZE  1024        /*栈可能达到的最大容量*/
typedef  int  DataType;
typedef struct
{   DataType  data[MAXSIZE];
    int  top;
}SeqStack ;
SeqStack   *s;               /*定义s是一个指向顺序栈的指针*/
```

在顺序栈中,通常数组下标为0的一端设为栈底,即s->data[0]是栈底元素,这样栈空时栈顶指针s->top=-1;当有元素入栈时,栈顶指针加1,即s->top++,s->top指示新的栈顶位置,然后再把元素赋值到这个新栈顶位置,栈顶元素为s->data[s->top];若s->top==MAXSIZE-1,表示栈满,则当再有新的元素需要入栈时,必将产生存储空间的溢出,这种情况称为"上溢"。在顺序栈中每当有元素出栈时,栈顶指针减1,即s->top--,s->top为新的栈顶位置,当s->top==-1时表示栈已经空了,这时再做出栈操作,也将产生溢出,这种情况称为"下溢"。栈空时说明栈中已经没有数据元素,不能再做出栈操作,因此出栈前一般先判定栈是否为空。入栈和出栈操作相当于在线性表的表尾进行插入和删除操作,其时间复杂度为$O(1)$。

栈的操作及栈顶指针变化情况如图3.2所示。图3.2(a)是空栈,图3.2(b)是元素A入栈后的状态,图3.2(c)是B、C、D、E这4个元素依次入栈之后的状态,如果再有元素入栈,就将产生上溢。图3.2(d)是E、D相继出栈,此时栈中还有3个元素,虽然最近出栈的元素D、E仍然在原先的单元存储着,但已经不再属于栈中的元素,top指针已经指向了新的栈顶,栈中元素为A、B、C,通过这个图示要深刻理解栈顶指针的作用。图3.2(e)是C、B、A依次出栈后的状态,此时栈已空,如果再做出栈操作,就将产生下溢。在这里,栈是竖着画的,并且下标依次从下向上递增,这样可以形象地表示出栈顶和栈底的上下关系。

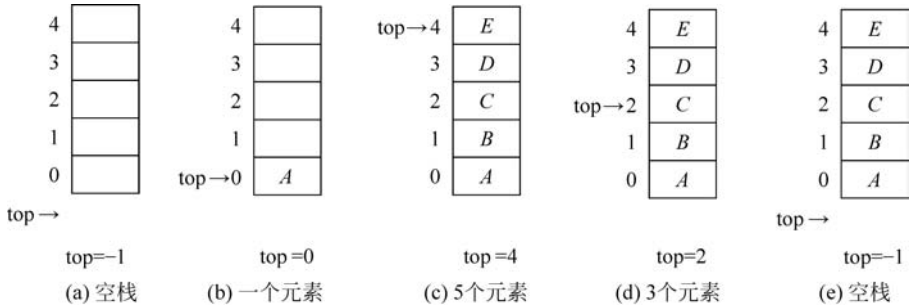


图3.2 栈顶指针top与栈中数据元素的关系

下面给出在顺序栈上实现5种基本运算的C函数。

(1) 初始化栈。

```
SeqStack * initSeqStack()
{   SeqStack * s;
    s = (SeqStack * )malloc(sizeof(SeqStack));        /* 申请栈的空间 */
    s -> top = - 1;                                   /* 初始化栈顶指针 */
    return s; }
```

(2) 判栈空。

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

(3) 入栈。

```
int push(SeqStack * s, DataType x)
{   if (s -> top == MAXSIZE - 1)
   {    printf("overflow"); return 0; }          /* 栈满不能入栈 */
   s -> top++;
   s -> data[s -> top] = x;
   return 1;
}
```

(4) 出栈。

```
void pop(SeqStack * s)                            /* 设栈不空 */
{ s -> top -- ;}
```

(5) 读栈顶元素。

```
DataType top(SeqStack * s)                        /* 设栈不空 */
{   return(s -> data[s -> top]);
}
```

注意：对出栈和读栈顶元素的操作，要求用户在调用此操作前，先判断栈是否为空，即先调用判栈空(empty(s))算法。

当程序中同时使用两个栈时，可以将两个栈的栈底分别设在数组向量空间的两端，让两个栈各自向中间延伸。这样，当一个栈中的元素较多，超过向量空间的一半时，只要另一个栈的元素不多，那么前者就可以占用后者的部分存储空间。只有当整个向量空间被两个栈占满(即两个栈顶相遇)时，才会发生上溢。因此，两个栈共享一个长度为 m 的向量空间和两个栈分别占用两个长度为 $\lfloor m/2 \rfloor$ 和 $\lceil m/2 \rceil$ 的向量空间比较，前者发生上溢的概率比后者要小得多。

3.1.3 链栈及运算的算法实现

通常链栈就是单链表,因此其节点结构与单链表的结构相同,栈顶指针就是单链表的头指针。

链栈存储结构的 C 语言描述如下:

```
typedef int DataType;
typedef struct Node
{    DataType data;
     struct Node * next;
} LinkStack;
LinkStack * top;            /* top 为栈顶指针 */
```

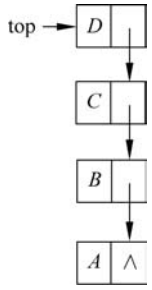


图 3.3 链栈存储结构的图示

栈的插入、删除操作都是在栈顶进行的,因此把单链表的表头一端作为栈顶最方便。表头指针即为**栈顶指针**,由栈顶指针指向的表头节点就是**栈顶元素**。由于只在链栈的头部进行插入和删除操作,因此没有必要像单链表那样为了运算方便附加一个头节点。链栈存储结构的图示如图 3.3 所示,top 是栈顶指针,它唯一地确定了一个链栈,当 top==NULL 时,栈为空。

下面给出在链栈上实现栈的 5 种基本运算的 C 函数。

(1) 初始化栈。

```
LinkStack * initLinkStack()
{    return NULL;
}
```

(2) 判栈空。

```
int emptyLinkStack(LinkStack * top)
{    if (!top) return 1;
     else  return  0; }
```

(3) 入栈。

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

(4) 出栈。

```
LinkStack * popLinkStack(LinkStack * top)          /* 设栈不空 */
{    LinkStack * p;
     p = top;
```

```
    top = top->next;
    free(p);
    return top;
}
```

(5) 读栈顶元素。

```
DataType topLinkStack(LinkStack *top)          /*设栈不空*/
{   return top->data;
}
```

从上面的基本运算的实现过程可以看出,链栈的入栈和出栈就相当于在单链表的表头进行插入和删除操作,其时间复杂度为$O(1)$。在链栈中,由于节点空间是动态申请和释放的,因此链栈没有溢出问题。

3.1.4　栈的应用

由于栈的"后进先出"特点,在很多实际问题中都利用栈作为一个辅助的数据结构进行求解。

【例 3.1】 将一个十进制正整数 N 转换为 r 进制的数。

以十进制正整数 N 转换为八进制的数为例,其转换方法是利用逐次除基数取余法。例如,十进制数 $N=1835$,基数 $r=8$,转换过程如下:

N	$N/8$(整除)	$N\%8$(取余)	
1835	229	3	低 ↑
229	28	5	
28	3	4	
3	0	3	高

所以,$(1835)_{10}=(3453)_8$。

在把十进制正整数转换为八进制数的过程中,是按照从低位到高位的顺序产生八进制的每位上的数字,而通常是从高位到低位输出每位上的数字,恰好与计算过程相反。这恰好与栈的"后进先出"的特点相符,因此可以利用栈来解决这个问题。在转换过程中,把得到的八进制数的每位进栈保存,转换完毕后按出栈顺序打印输出,则正好是所求的八进制数。十进制数转换为其他 r 进制数的原理与十进制数转换为八进制数相同,都利用的是"除基取余法",但当 $r>9$ 时,需要使用字母 $A,B,C,D,\cdots$,读者思考如何修改下面的算法。

转换算法中的主要步骤如下。

(1) 当 $N\neq 0$ 时,将 $N\%r$ 存入栈 s 中,然后用 N/r 代替 N,直到 $N=0$ 退出循环。

(2) 只要栈 s 不空,就读出栈顶元素输出并把栈顶元素出栈。

进制转换算法的 C 函数如下:

```
void convert(int N,int r)
{   SeqStack *s;                    /*定义一个顺序栈*/
```

```
    int x;
    s = initSeqStack();              /* 初始化栈为空 */
    while ( N )
    {   push(s ,N % r);              /* 余数入栈 */
        N = N / r ;                  /* 商作为被除数,继续循环 */
    }
    while(!empty(s))
    {   x = top(s) ; pop(s);
        printf(" %d ",x);            /* 读出栈顶元素并打印 */
    }
}
```

栈的引入简化了程序设计问题,使问题的求解层次更加清楚。上面的问题当然也可以用数组来实现,但是用数组实现不仅要关注问题本身,还要考虑数组下标的增减等细节问题。下面的算法中,就是把一个 int 型数组 S 加上一个 int 型变量 top 作为一个栈来使用,从而实现上述进制转换问题。

```
#define M 20
void conversion(int N,int r)
{    int s[M], x, top = -1;
     while(N)
     {   top++; s[top] = N % r; N = N / r; }
     while (top >= 0)
     {   x = s[top]; top -- ;
         printf(" %d",x);
     }
}
```

【例 3.2】 算术表达式中括号匹配的检查。

算术表达式中括号匹配的检查是栈的典型应用实例。假设表达式中只允许出现两种括号:方括号和圆括号,这两种括号可以嵌套,但必须成对出现。例如,[[()()]]或[()[]]等是正确的格式,而[](]或([)]]或[(][])等是不正确的格式。

上面的括号序列[[()()]]在依次输入前 3 个左括号时因为全是左括号都不可能得到匹配,但当遇到第 1 个")"时需要与第 3 个输入的"("匹配,而第 2 个")"要与它前面刚刚输入的"("匹配,第 1 个"]"要与第 2 个输入的"["匹配,而第 2 个"]"要与第 1 个输入的"["匹配。可以看出,右括号与左括号匹配时,后输入的左括号最先得到匹配,这正好与栈的性质相吻合。因此,可以用栈来实现括号匹配的检查。

括号不匹配的情况可能有如下 3 种:①刚到来的右括号不是所"期待"的,如需要匹配的是"(",而到来的却是"]";②到来的右括号已经没有与之匹配的左括号,这种情况说明右括号多了,而左括号却少了;③直到结束,也没有到来所"期待"的右括号,这种情况说明左括号有多余的,缺少右括号。

用栈来实现括号匹配检查的原则是对表达式从左到右扫描。①当遇到左括号时,左括号入栈。②当遇到右括号时,首先检查栈是否为空,若栈为空,则表明该右括号多余;否则

比较栈顶左括号是否与当前右括号匹配。若匹配，则将栈顶左括号出栈，继续操作；否则，表明不匹配，停止操作。③当表达式全部扫描完毕，若栈为空，则说明括号匹配，否则表明左括号有多余。

当然，括号匹配检查也可以延伸到任何成对出现的定界符，如引号、书名号等。对于出现在这些成对定界符之间的符号，在算法实现时只要跳过即可。判断一串带左右括号的字符序列是否匹配，其算法实现的 C 函数如下：

```
int match()
{    SeqStack *s;
     char c,e;
     s=initSeqStack();
     scanf("%c",&c);
     while (c!='#')                          /*以#作为输入结束标志*/
     {   if((c=='(') || (c=='['))            /*左括号入栈*/
            push(s,c);
         else if((c==')') || (c==']'))
         {    if(!empty(s))
              {    e=top(s);
                   if((c==')') &&(e=='(')||(c==']') &&(e=='['))
                            pop(s) ;         /*左右括号匹配栈顶元素出栈*/
                   else                      /*出栈的左括号与要求匹配的右括号匹配不上*/
                   {   printf("%c括号不匹配\n",c);
                       return 0;    }
              }
              else                           /*有右括号输入,栈为空,无匹配的左括号*/
              {   printf("右括号多了,不匹配\n") ;
                  return 0;
              }
         }
         scanf("%c",&c);
     }/*while*/
     if(!empty(s))                           /*输入结束后,栈中还有左括号,不匹配*/
     {   printf("左括号多了,括号不匹配\n") ;
         return 0;
    }
    else                                     /*输入结束后,栈为空,匹配*/
     {   printf("括号匹配\n") ;
         return 1;
     }
}
```

栈的应用例子还有很多，如行编辑程序、迷宫求解和表达式求值等。栈中元素的操作特点是“后进先出”，只要应用程序中的数据具有保存与使用时顺序相反，就可以考虑是否可以利用栈。

3.1.5 栈与递归

1. 递归

栈的一个重要应用是在计算机高级程序设计语言中实现递归。一个函数(过程)在其定义中直接或间接出现了对自身的引用就是一种递归,如阶乘运算。

$$n!=\begin{cases}1 & n=0 \quad /*\ 递归出口\ */\\ n\times(n-1)! & n>0 \quad /*\ 递归步骤\ */\end{cases}$$

所谓递归,是指在对问题进行分解时,得到的是与问题性质相同的子问题,二者的解决方法相同,只是所处理的对象有所不同,但所处理对象之间的关系是有规律变化的。简单地说,也就是在某个处理过程中又包含了同样的处理方法,这时就可以考虑采用递归方法来解决。在递归方法中,首先必须有一个明确的结束递归的条件,称为递归出口,否则递归会无止境地进行下去;其次必须保证每次递归都更接近递归出口,直到到达出口,使递归过程结束。

上面的阶乘问题,要求的是 $n!$,可以把问题转换为 $n\times(n-1)!$,而求 $(n-1)!$ 时,又可转换为 $(n-1)\times(n-2)!$,…,每次都是一个数和另一个数的阶乘相乘的问题,并且被处理的对象又是有规律递减的,分别是 n,$n-1$,$n-2$,…,可见处理问题的性质是相同的,由此可用递归实现。在这个问题中,递归结束条件是 $n=0$,阶乘值为 1;随着递归过程的不断进行,n 的值逐渐递减,直到到达递归出口($n=0$),使递归过程结束。

2. 栈实现递归

在支持递归调用的计算机高级语言(如 C 语言、C++、Java 等)中,可以用递归函数来实现递归。一个函数在其定义的内部直接调用自身,这个函数就称为直接递归函数。根据阶乘问题的定义可以很自然地写出相应的递归函数。

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

递归是程序设计中一个强有力的工具,在计算机科学和数学中有着广泛的应用。很多问题采用递归方法解决,可以使问题的描述和求解变得简洁和清晰。

一般来说,当问题本身或者所涉及的数据结构是递归定义的,设计算法时用递归方法比非递归方法更容易。本课程中的很多数据结构,由于结构本身固有的递归特性,使得某些运算可采用递归方法实现,如广义表的运算、二叉树的遍历和图的遍历等都可采用递归方法。此外,有些问题虽然没有明显的递归结构,但用递归求解更简单,如八皇后问题、汉诺塔问题等。

调用求阶乘函数的 main()函数如下:

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

程序的执行过程如图 3.4 所示。函数执行过程中,后调用的函数先返回,符合栈的特点。因此,对于函数调用时存储空间的分配和释放,系统以栈的方式来实现对存储空间的管理。

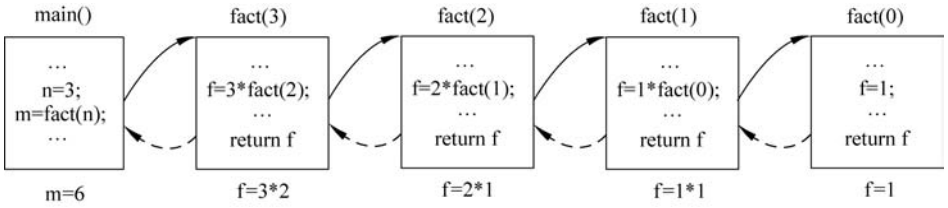
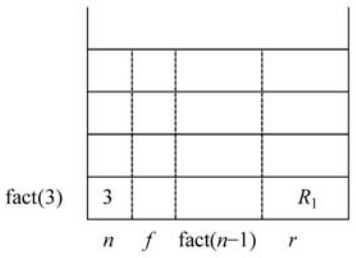
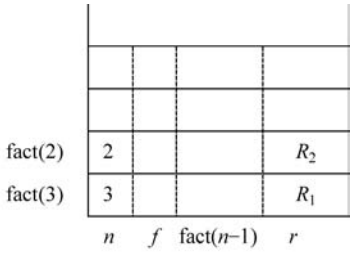


图 3.4 fact(3)的执行过程

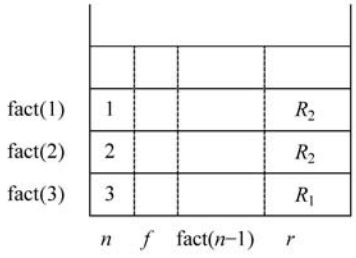
函数在一个生命周期(包括调用、执行、结束)中所需要的信息,保存在 1 字节连续的存储区域(这个连续的存储区域也称为活动记录)。在这个存储区域中包含了形式参数、返回地址、局部变量、临时变量等信息。函数调用时,操作系统把程序运行时所需的数据空间组织成一个栈,以栈的形式分配和回收存储区域,由此实现主调函数和被调用函数间的信息传递和控制转移。每当调用一个函数时,就在栈顶为函数分配一个新的字节连续的存储区域,一旦本次调用结束,则将栈顶存储区域出栈,根据获得的返回地址信息返回到主调函数。递归函数的调用也遵循上述规则,只是主调函数和被调用函数是同一个函数而已。图 3.5 所示为递归调用过程中栈及栈中数据的变化状况。



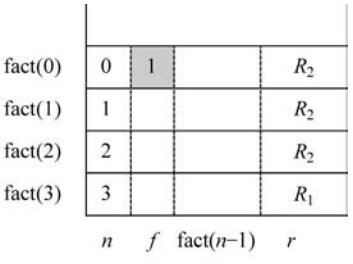
(a) 调用fact(3)时，栈及栈中数据



(b) 调用fact(2)时，栈及栈中数据

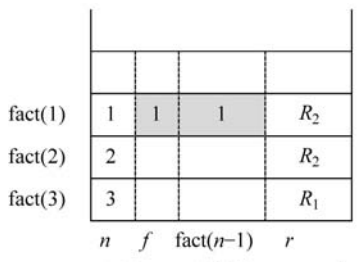


(c) 调用fact(1)时，栈及栈中数据

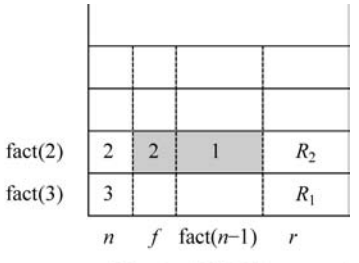


(d) 调用fact(0)时，栈及栈中数据

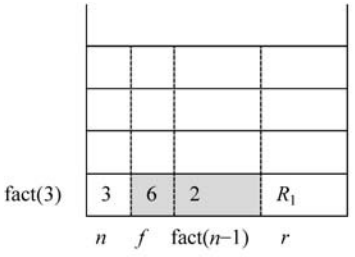
图 3.5 递归调用过程中栈及栈中数据的变化状况



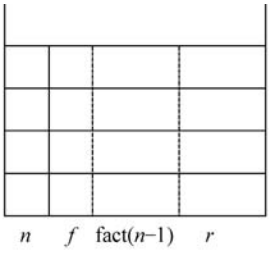
(e) fact(0)出栈后，继续执行fact(1)时



(f) fact(1)出栈后，继续执行fact(2)时



(g) fact(2)出栈后，继续执行fact(3)时



(h) fact(3)出栈后，返回主程序

图 3.5 （续）

(1) 当主函数执行到 m=fact(n)时，要中断当前主函数的执行，开始调用函数 fact(3)，此时在栈顶保存函数调用的现场信息。对于 fact()函数，需要保存的信息有 4 个：n、f、fact($n-1$)的值以及返回到主调函数的地址 r，其中 R_1 为主函数调用 fact()时返回点地址，R_2 为 fact()函数中递归调用 fact($n-1$)时返回点地址。

(2) 调用 fact(3)时，栈及栈中数据如图 3.5(a)所示，返回地址为 R_1，fact($n-1$)即 fact(2)值未知。当 fact(3)执行到 $f=3*$fact(2)时，由于 fact(2)未知，递归调用函数 fact(2)。这时系统在栈顶分配一块存储空间，由于都是 fact()函数，保存的信息相同，只不过其对应的值不同。调用 fact(2)时，栈及栈中数据如图 3.5(b)所示，返回地址是 R_2。同理，递归调用 fact(1)和 fact(0)，栈及栈中数据如图 3.5(c)和图 3.5(d)所示。

(3) 在执行 fact(0)时，由于 $n=0$，$f=1$，当 fact(0)执行到 return f 时，函数调用结束，这时释放 fact(0)所占用的空间，即 fact(0)数据空间出栈。根据返回地址 R_2 返回主调函数 fact(1)，并把 fact(0)的值 1 传给 fact(1)中的 fact($n-1$)处。注意，释放 fact(0)后栈顶正好是 fact(1)的数据空间。图 3.5(e)是 fact(0)出栈后继续执行 fact(1)但还未执行 return 语句时栈及栈中的数据情况，其中 f 值正好是 $n=1$ 和返回的 fact(0)=1 值的积。以此类推，fact(1)和 fact(2)调用结束时依次出栈。

(4) 当 fact(2)出栈后，栈顶是 fact(3)，根据 $n=3$ 和 fact(2)=2 计算出 $f=6$，执行 return f 结束调用，这时 fact(3)出栈，返回到 main()函数，把 fact(3)=6 值传给 m，$m=6$，从返回地址 R_1 继续执行 main()函数。fact(1)～fact(3)出栈后及栈中数据情况如图 3.5(f)～图 3.5(h)所示。

上述递归调用 fact(n)，栈使用的最大深度为 $n+2$，其空间复杂度为 $O(n)$。每次递归调用执行一次 if 条件语句，其时间复杂度为 $O(1)$，整个算法包括主程序，共进行了 $n+2$ 次调用，其时间复杂度也为 $O(n)$。

3. 递归到非递归的转换

一般情况下，递归函数写起来容易但理解起来往往比较困难。另外，在有些程序设计语言中不允许编写递归函数。因此，如何将递归函数转换为非递归函数，就是经常要探讨的一个问题。通过递归函数的实现过程可以看出，递归函数的完成需要借助一个系统栈来保存中间结果。实际上，用户也可以在算法中设置栈来模拟系统栈的作用，从而将递归函数转换为非递归函数。这种方法在数据结构中有较多实例，如二叉树遍历算法的非递归实现和图的深度优先遍历算法的非递归实现等。理论上，用这种方法可以把所有的递归算法转换为非递归算法。以求阶乘为例，用栈将递归算法转换为非递归算法。

阶乘的非递归算法实现如下：

```
long fact2(int n)                    /* 用栈实现非递归的阶乘运算 */
{   SeqStack *s;
    long f = 1;
    int i = n;
    s = initSeqStack();
    while(i > 0)
    {   push(s,i); i--; }
    while(!empty(s))
    {   i = top(s); f = f * i;  pop(s);  }
    return  f;
}
```

本算法的时间复杂度和空间复杂度都为 $O(n)$。

某些递归算法也可以通过循环结构转换为非递归算法实现。一般的尾递归函数(即递归调用语句是递归函数中的最后一条语句)都可以用这种方法转换为非递归实现。求阶乘运算的递归函数就是一个尾递归函数，可以采用循环结构转换为非递归算法。

用循环结构实现阶乘的非递归算法如下：

```
long fact1(int n)                    /* 用循环结构实现非递归的阶乘运算 */
{   long f = 1;
    int i;
    for (i = 1;i <= n;i++) f = f * i;
    return f;
}
```

本算法的时间复杂度为 $O(n)$，与前两个算法相同；空间复杂度为 $O(1)$，比前两个算法要好，从线性级 $O(n)$下降为常量级 $O(1)$。

递归函数写起来比较容易，理解起来困难。如果掌握栈在实现递归过程中的作用，则会对理解递归函数有很大的帮助。

3.2 队列

3.2.1 队列的定义及运算

栈是一种后进先出的线性表，在实际问题中还经常使用一种“先进先出”的线性表：只允许在表的一端进行插入，而在表的另一端进行删除，将这种线性表称为**队列**(queue)。显

然,队列也是一种运算受限的线性表。在队列中,把允许插入的一端称为**队尾**(rear),把允许删除的一端称为**队头**(front)。向队列中插入一个元素称为**入队**,从队列中删除元素称为**出队**。没有任何元素的队列称为**空队列**。队列中元素出队的顺序与其进入队列的顺序是一致的,最先入队的元素最先出队,所以队列又称为**先进先出**(first in first out,FIFO)的线性表,简称 **FIFO 表**。

图 3.6 所示是一个有 5 个元素的队列(a_1,a_2,a_3,a_4,a_5),其中 a_1 是队头元素,a_5 是队尾元素。队列中,入队的顺序依次为 a_1,a_2,a_3,a_4,a_5,出队的顺序将依然是 a_1,a_2,a_3,a_4,a_5。也就是说,只有在 a_1 出队后,a_2 才能出队。以此类推,只有在 a_1,a_2,a_3,a_4 出队后,a_5 才能出队。

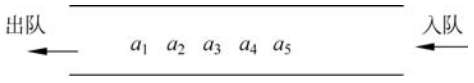


图 3.6 队列图示

在日常生活中队列的例子很多,如排队购物就是一个队列,排头的人先买完离开,新来的人排在队尾,等所有人买完离开后,就是一个空队列。程序设计中也经常使用队列。银行等机构实行的取号排队符合队列的特性,因此一般都采用队列实现。计算机操作系统允许多个进程同时运行,如果运行的结果都要通过通道输出,则会按请求的先后次序排队完成输出。在本书第 5 章二叉链表的建立和第 6 章图的广度优先遍历算法中,也应用了队列这种数据结构。

与栈的基本运算类似,队列上对应也有以下 5 个基本操作。

(1) 队列初始化 initQueue(q)。构造一个空队列。

(2) 判队空 emptyQueue(q)。若 q 为空队列则返回为 1,否则返回为 0。

(3) 入队 enQueue(q,x)。对已存在的队列 q,插入一个元素 x 到队尾,队发生变化。

(4) 出队 deQueue(q,x)。删除队头元素,并通过 x 返回其值,队发生变化。

(5) 读队头元素 frontQueue(q)。读队头元素,并返回其值,队不变。

在队列的运算定义中,也没有查找运算,这一点与栈的运算定义相同,因为队列也是运算受限的线性表,操作位置固定。

与栈相同,队列也可以采用顺序存储和链接存储方法。采用顺序存储方法的队列称为**顺序队列**(sequential queue)。采用链接存储方法的队列称为**链队列**(linked queue)。

3.2.2 顺序队列及运算的实现

顺序队列也是利用数组依次存放从队头到队尾的元素。类似顺序栈,可以将队头固定在数组的一端(下标为 0),设一个队尾指针指向队尾元素。然而,由于队头固定,这种顺序队列在做出队操作时,需要移动队列中的所有元素,大大影响了算法效率。为了不移动队列中的元素,增设队头指针指向队头元素。此时,队列的队头和队尾的位置都是变化的。

顺序队列存储结构的 C 语言描述如下:

```
#define MAXSIZE 1024          /* 队列的最大容量 */
typedef int DataType;
typedef struct
```

```
{    DataType data[MAXSIZE];          /* 队员的存储空间 */
     int rear, front;                 /* 队头、队尾指针 */
}SeQueue;
SeQueue   * sq;                       /* 定义一个指向队列的指针变量 */
```

使用队列时，同栈一样要将队列初始化为空队列。一般规定队头指针等于队尾指针时为空队列，队列初始化为空的操作为 sq－＞front＝sq－＞rear＝－1。为了避免只剩一个元素时，队头和队尾重合使处理变得麻烦，设计队列时可将队头指针 front 指向队头元素前面一个位置，队尾指针 rear 指向队尾元素位置(也可将 front 指向队头元素位置，队尾指针 rear 指向队尾元素的下一个位置)，这种方式设置的指针依然保证了空队列的条件为 sq－＞front＝＝sq－＞rear。

按照上述规定，入队、出队时头尾指针及队列中元素之间的关系如图 3.7 所示(设 MAXSIZE＝6)。在不考虑溢出的情况下，执行入队操作时，队尾指针加 1，指向原队尾元素的下一个位置，然后在新位置上存储元素，用 C 语言描述如下：

```
sq->rear = sq->rear + 1;
sq->data[sq->rear] = x;              /* 把 x 写入队尾位置 */
```

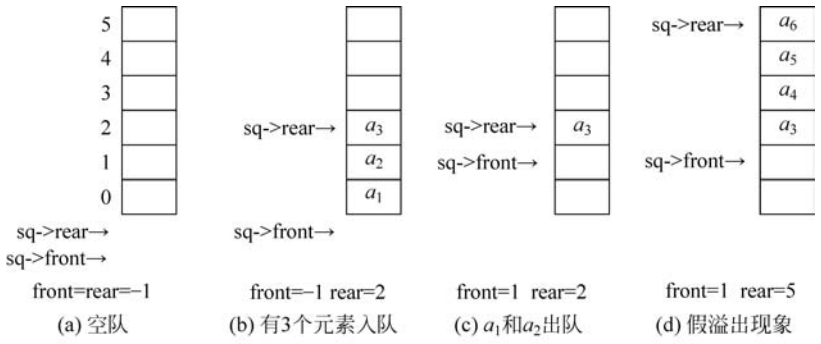


图 3.7 顺序队列头尾指针及队列中元素之间的关系

在不考虑队空的情况下，出队操作为队头指针加 1，指向原队头元素位置，表明队头元素出队，并把原队头元素保存在 x 中，用 C 语言描述如下：

```
sq->front = sq->front + 1;
x = sq->data[sq->front];             /* 把出队元素值赋给 x */
```

随着入队、出队的进行，当出现图 3.7(d)中的现象时，sq－＞rear＝MAXSIZE－1，队尾指针已经移至最后，再有元素入队就会出现“溢出”现象。但从图 3.7(d)中可以看出，队列中并未真正的“满员”，还有空闲的空间可供利用，这种现象称为“假溢出”。解决“假溢出”的简单方法是将队列的数据区假想成一个头尾相接的环形结构，也就是 sq－＞data[0]紧接在 sq－＞data[MAXSIZE－1]之后，如图 3.8 所示，称其为“循环队列”。在循环队列中，头尾指针的关系不变，进行入队、出队操作时，头尾指针顺时针方向移动。

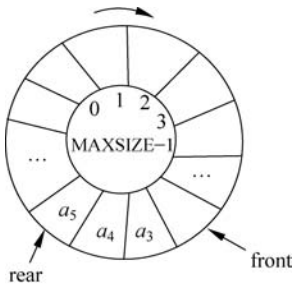


图 3.8 循环队列图示

设 MAXSIZE=6,图 3.9 是循环队列中指针变化情况图示。图 3.9(a)是循环队列的一般状态,具有 a_3,a_4 两个元素,队头元素是 a_3,队尾元素是 a_4;图 3.9(b)是 a_3 和 a_4 相继出队后,队空了,此时 sq->front==sq->rear==3;图 3.9(c)a_5,a_6,a_7 相继入队,当 a_6 入队后,sq->rear=5,到了数组的上界,当 a_7 入队时,由于是循环队列,且数组下标为 0 的单元为空闲,因此 a_7 存放在 sq->data[0]位置,sq->rear=0;图 3.9(d)a_8,a_9,a_{10} 相继入队,此时所有空间均被占用,队满了,且有 sq->front==sq->rear==3。

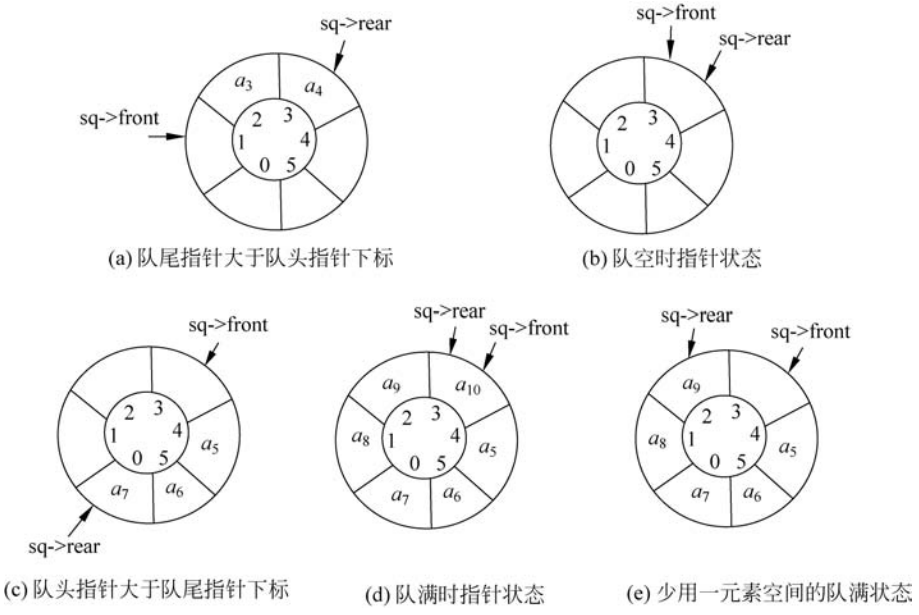


图 3.9 循环队列中指针变化情况图示

由此可以看出,无论是在队空情况下还是在队满情况下均有 sq->front==sq->rear,也就是说队满和队空的条件是相同的,出现这种情况显然不允许。解决方法有两种:一种是附设一个标志变量以区别是队空还是队满,例如可以设存储队列中元素个数的变量 num,当 num=0 时队空,当 num=MAXSIZE 时队满;另一种方法是在循环队列中少用一个元素空间,即把图 3.9(e)所示的情况视为队满,此时的状态是队尾指针加 1,就会从后面赶上队头指针。本节讨论的队列是采用后一种方法表示队满。

对于循环队列,**初始化队空**操作的 C 语言描述如下:

```
q->front = q->rear = MAXSIZE - 1
```

执行入队操作时,当 sq->rear==MAXSIZE-1 时,做 sq->rear+1 运算后,应使 sq->rear=0;否则 sq->rear=sq->rear+1,可以用 C 语言中的 if 语句实现:

```
if(sq->rear == MAXSIZE - 1) sq->rear = 0;
else sq->rear++;
```

入队操作时,队尾指针的移动更简单的方法是用“模运算”,用 C 语言描述如下:

```
sq->rear = (sq->rear + 1) % MAXSIZE;
```

出队操作时，队头指针移动的 C 语言描述如下：

```
sq->front = (sq->front + 1) % MAXSIZE;
```

判断队满的条件的 C 语言描述如下：

```
(sq->rear + 1) % MAXSIZE == sq->front;
```

队列中元素个数的 C 语言描述如下：

```
(sq->rear - sq->front + MAXSIZE) % MAXSIZE;
```

下面给出实现循环队列基本运算的 C 函数。

(1) 队列初始化。

```
SeQueue *initSeQueue()
 {   SeQueue *q;
     q=(SeQueue *) malloc(sizeof(SeQueue));
     q->front=q->rear=MAXSIZE-1;
     return q;
}
```

(2) 判队空。

```
    int emptySeQueue(SeQueue *sq)
    {   if(sq->front==sq->rear) return 1;
        else    return 0;
    }
```

(3) 入队。

```
int enSeQueue(SeQueue *sq, DataType x)
{   if((sq->rear+1) % MAXSIZE==sq->front)
     {  printf("队满"); return 0; }                    /*队满不能入队*/
     else
     {  sq->rear=(sq->rear+1) % MAXSIZE;
        sq->data[sq->rear]=x;
        return 1; }                                   /*入队完成*/
}
```

(4) 出队。

```
void deSeQueue(SeQueue *sq, DataType *x)
{   sq->front=(sq->front+1) % MAXSIZE;
    *x=sq->data[sq->front];          /*读出队头元素通过指针x返回主调函数*/
}
```

(5) 读队头元素。

```
DataType frontSeQueue(SeQueue *sq)
{   int front;
```

```
front = (sq -> front + 1) % MAXSIZE;
return sq -> data[front]; }
```

注意：用户在调用出队和读队头元素操作前，需要判断队列是否为空。同样地，后面的链队列调用出队和读队头元素操作前也要判断队列是否为空。循环队列的初态和终态都可以为空，因此队空可作为程序转移的判定条件。

3.2.3 链队列及运算的实现

采用带头节点的单链表来实现链队列，链队列中的节点类型与单链表相同。由于队列只允许在表头进行删除表尾进行插入，因此可以设一个头指针 front 指向队头节点，一个尾指针 rear 指向队尾节点，使得在队尾的入队和队头的出队操作时间复杂度都是 $O(1)$。如果只设头指针，虽然也可以实现在队尾的入队操作，但队尾入队的时间复杂度将为 $O(n)$。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体中，链队列的存储结构用 C 语言描述如下：

```
typedef int DataType;
typedef struct Node
{     DataType data;
      struct Node * next;
} LQNode;            /* 链队列节点的类型 */
typedef struct
{     LQNode * front, * rear;
}LQueue;             /* 将头、尾指针封装在一起的链队列 */
LQueue * q;          /* 定义一个指向链队列的指针 */
```

为了使空队列和非空队列上的操作尽可能相同，一般在链队列的队头元素前加一个头节点，这样头指针 front 指向头节点，尾指针 rear 指向队尾节点。按这种思想建立的带头节点的链队列一般情况如图 3.10(a)所示。图 3.10(b)是空链队列，头指针和尾指针都指向头节点，由此可知队空的判定条件是 q->front==q->rear。图 3.10(c)是只包含一个节点的链队列，此时元素出队，需要改变尾指针的指向。

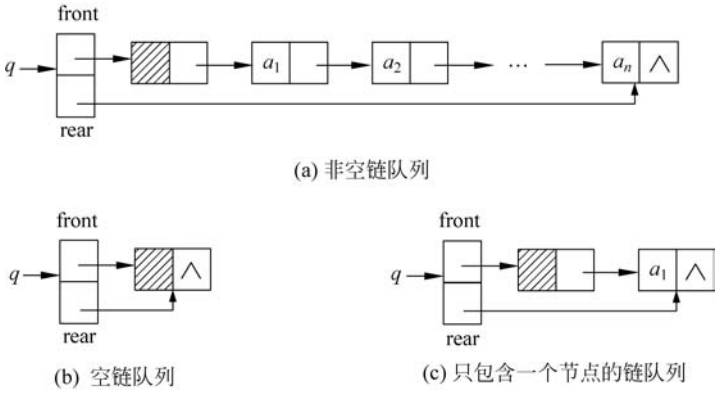


图 3.10 链队列图示

另外，还有一种更简单的链队列实现方法，就是用设尾指针的带头节点的单循环链表来存储队列中的元素。这种单循环链表在第 2 章介绍过，如图 3.11 所示。虽然只设了一个尾指针 r，但是找头节点的指针非常容易，即 r－＞next，队头元素的指针为 r－＞next－＞next，队空的判定条件是 r－＞next＝＝r。这种表示法比图 3.10(a)所示的链队列更简单，同时还减少了队头、队尾指针的节点(＊q)所占的空间。

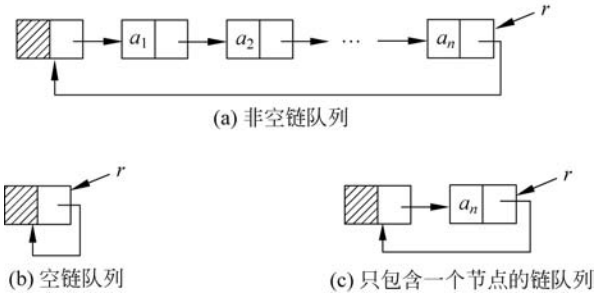


图 3.11　单循环链表链队列的图示

下面给出在图 3.11 所示的链队列上实现队列基本运算的 C 函数。

(1) 创建一个带头节点的空队。

```
LQNode * initLQueue()
{  LQNode * r;                              /* 链队列尾指针 */
   r = (LQNode * )malloc(sizeof(LQNode)); /* 申请链队列头节点空间,且使 r 指向头节点 */
   r -> next = r;
   return r;
}
```

(2) 判队空。

```
int  emptyLQueue(LQNode * r)
{    if (r -> next == r) return 1;
     else return 0;
}
```

(3) 入队。

```
LQNode * enLQueue(LQNode * r, DataType x)
{  LQNode * p;
   p = (LQNode * )malloc(sizeof(LQNode));          /* 申请新节点空间 */
   p -> data = x; p -> next = r -> next;
   r -> next = p; r = p;
return r;  }
```

在入队操作时，由于尾指针会改变，因此函数要返回尾指针的值。

(4) 出队。

```
LQNode * deLQueue(LQNode * r, DataType * x)
{  LQNode * p;
   p = r -> next -> next;                  /* 取队头元素 */
```

```
    if (p == r)     /* 只有一个元素时,出队后队空,此时要修改队尾指针使其指向头节点 */
    {   r = r -> next;r -> next = r; }
    else r -> next -> next = p -> next;   /* 队列中超过一个元素时,设置新的队头元素 */
     * x = p -> data;                     /* 原队头元素用指针 x 带回主调函数 */
    free(p);
    return r;
}
```

出队操作需要注意当链队列中只有一个节点时,如图 3.11(c)所示,尾指针指向唯一的这个节点,当这个节点被删除后,队列中无元素,尾指针无所指,此时还要修改队尾指针使其指向头节点,表明队列已为空。

(5) 读队头元素。

```
DataType frontLQueue(LQNode * r)
{    return r -> next -> next -> data;
}
```

顺序存储的循环队列需要预先设定一个队列的最大长度,若用户无法预先估计队列的长度,此时用链队列比较好。

3.3 栈与队列的比较

栈和队列都是运算受限的线性表,它们之间有很多的相似之处,但也有区别。

1. 具有相同的逻辑结构

从逻辑关系上看,栈和队列都是线性表,都具有线性结构的逻辑特征,元素之间都是一对一的线性关系。同线性表一样,栈或队列都可以用其结构中的元素序列来表示数据的逻辑结构,例如,栈 $S=(a_1,a_2,\cdots,a_n)$,队列 $Q=(a_1,a_2,\cdots,a_n)$。

2. 采用相同的存储方式

常用的两种存储方式包括顺序存储和链接存储,都适用于栈和队列存储结构的设计。用顺序存储方式的栈称为顺序栈,用链接存储方式的栈称为链栈。用顺序存储方式的队列称为顺序队列,用链接存储方式的队列称为链队列。

3. 具有不同的运算特点

栈和队列的主要区别是在运算上。同线性表的运算相比,栈和队列在操作上受到限制。线性表可以在元素序列的任何位置上进行插入、删除操作,一般在操作前需要先查找到相应的元素,然后才能进行相应的插入或删除操作。查找运算往往是线性表上使用频率最高的操作,查找、插入、删除的时间复杂度一般为 $O(n)$。栈和队列的插入和删除运算都限制在线性表的表端完成。若限制插入和删除运算都在表的一端完成即为栈;若限制插入运算在表的一端完成,删除运算在表的另一端完成,就是队列。完成栈的插入、删除运算的一端一般是线性表的终止端,所以栈的插入、删除操作与线性表在表尾进行的插入、删除操作相同。

对于队列，完成插入运算的一端通常为表尾，完成删除运算的一端通常为表头，所以队列的插入、删除操作等同于线性表在表尾插入、表头删除。由此可知，栈和队列的插入及删除运算时间复杂度都可以达到$O(1)$。由于栈和队列的插入、删除操作都在固定的位置，因此在栈和队列上不需要查找运算。总结栈和队列的运算特点，可得到栈和队列的另一种称谓：栈又叫“后进先出”(LIFO)的线性表；队列又叫“先进先出”(FIFO)的线性表。

4. 具有广泛的应用价值

计算机的软件系统中有很多使用栈和队列的例子，实际应用时主要根据它们的特点选择使用哪一种结构。前面介绍的栈与递归是栈的最典型应用之一，在函数调用过程中，操作系统利用栈的方式管理存储空间的申请和释放。在处理所有具有对称关系的数据(如{}、[]、()、“”等)时，都可以用栈来检查其匹配的正确性。此外，迷宫求解、汉诺塔、四则运算表达式求值、序列逆转等问题都可以通过栈来实现。队列的应用也很多。在操作系统中，经常使用的先来先服务原则就是通过队列来实现的。银行等机构的排号系统也可以通过队列实现。总之，要根据栈和队列的特点来充分发挥它们在程序设计中的作用。

小结

栈和队列是在解决实际问题中广泛使用的两种典型结构。从逻辑结构上看，它们都是线性表；从存储结构上看，它们可以和线性表一样，进行顺序存储和链接存储；与线性表不同的是，栈和队列上定义的运算不一样。所以，栈和队列都是运算受限的线性表。栈的插入和删除运算限制在表的一端完成；而队列的插入和删除运算分别限制在表的两端完成。由于栈和队列上的插入和删除运算位置已经固定，因此不需要查找运算。同样地，由于栈和队列在运算上的限制，使得它们各自呈现出明显的特点(LIFO 和 FIFO)。利用栈和队列的不同特点，根据不同应用中数据的使用特性，可以选择栈或者队列作为某些算法设计中的辅助数据结构。

本章重点：

(1) 栈与队列的定义及特点，两种结构中基本运算的定义。

(2) 根据栈和队列的特点，选择栈或者队列在解决实际问题中的具体体现。

(3) 循环队列的设计方法，队空、队满的判断条件，入队、出队的操作方法，队列中元素个数的计算公式。

习题

一、名词解释

1. 栈
2. 队列
3. 顺序栈
4. 链队列

二、选择题

1. 设一个栈的输入序列是 1,2,3,4,5,则下列序列中,栈的合法输出序列是(　　)。

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

2. 设有一个顺序栈,元素 1,2,3,4,5 依次进栈,如果出栈顺序是 2,4,3,5,1,则栈的容量至少是(　　)。

A. 1　　B. 2　　C. 3　　D. 4

3. 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,当入队一个元素,再出队两个元素后,rear 和 front 的值分别为(　　)。

A. 1 和 5　　B. 2 和 4　　C. 4 和 2　　D. 5 和 1

4. 递归函数调用时,需要使用(　　)数据结构来处理参数、返回地址等信息。

A. 队列　　B. 多维数组　　C. 栈　　D. 线性表

三、填空题

1. 栈和队列都是操作受限的线性表,栈的运算特点是_____,队列的运算特点是_____。

2. 若序列 a、b、c、d、e 按顺序入栈,假设 P 表示入栈操作,S 表示出栈操作,则操作序列 $PSPPSPSPSS$ 后得到的输出序列为_____。

3. 已知一个顺序栈 $*s$,栈顶指针是 top,它的容量为 MAXSIZE,则判断栈空的条件为_____,栈满的条件是_____。

4. 对于队列来说,允许进行删除的一端称为_____,允许进行插入的一端称为_____。

5. 某循环队列的容量 MAXSIZE=6,队头指针 front=3,队尾指针 rear=0,则该队列有_____个元素。

四、简答题

1. 栈上的基本运算有哪些?

2. 给出循环顺序队列的存储结构图示及 C 语言描述。

3. 简述栈和队列的联系与区别。

五、写出下列程序段的输出结果

1. 栈为本章定义的顺序栈,栈中的元素类型为 char。

```
void main()
{
    SeqStack *S;
    char x, y,ch;
    S = initSeqStack();
    x = 'o'; y = 'e';
    push(S, 'l');  push(S, x); push(S, 'o');  pop(S);
    push(S,'v'); push(S, 'k');  pop(S);push(S,y);
    while(!StackEmpty(S))
    {
        ch = top(S);
        printf("%c",ch);
    }
}
```

2. 队列为本章定义的循环队列，队列中的元素类型为 char。

```
void main()
{   SeQueue *Q;
    char ch1 = 'M',ch2 = 'G';
    char *x = &ch1, *y = &ch2;
    Q = initSeQueue();
    enSeQueue(Q,'I');
    enSeQueue(Q,'L');
    enSeQueue(Q, y);
    deSeQueue(Q, x);
    enSeQueue(Q, x);
    deSeQueue(Q, x);
    enSeQueue(Q,'R');
    While(!emptySeQueue(Q))
    {
       deSeQueue(Q,y);
       printf("%c", *y);
    }
    printf("%c", *x);
}
```

六、算法设计题

1. 通常称正读和反读都相同的字符序列为“回文”，如 abcdeedcba 和 abcdcba 都是回文。若字符序列存储在一个单链表中，编写算法判断此字符序列是否为回文(提示：将一半字符先依次进栈)。

2. 假设以数组 $a[m]$存放循环队列的元素，同时设变量 rear 和 length 分别作为队尾指针和队中元素个数，试给出判别此循环队列的队满条件，并写出相应入队和出队的算法(在出队的算法中要返回队头元素)。