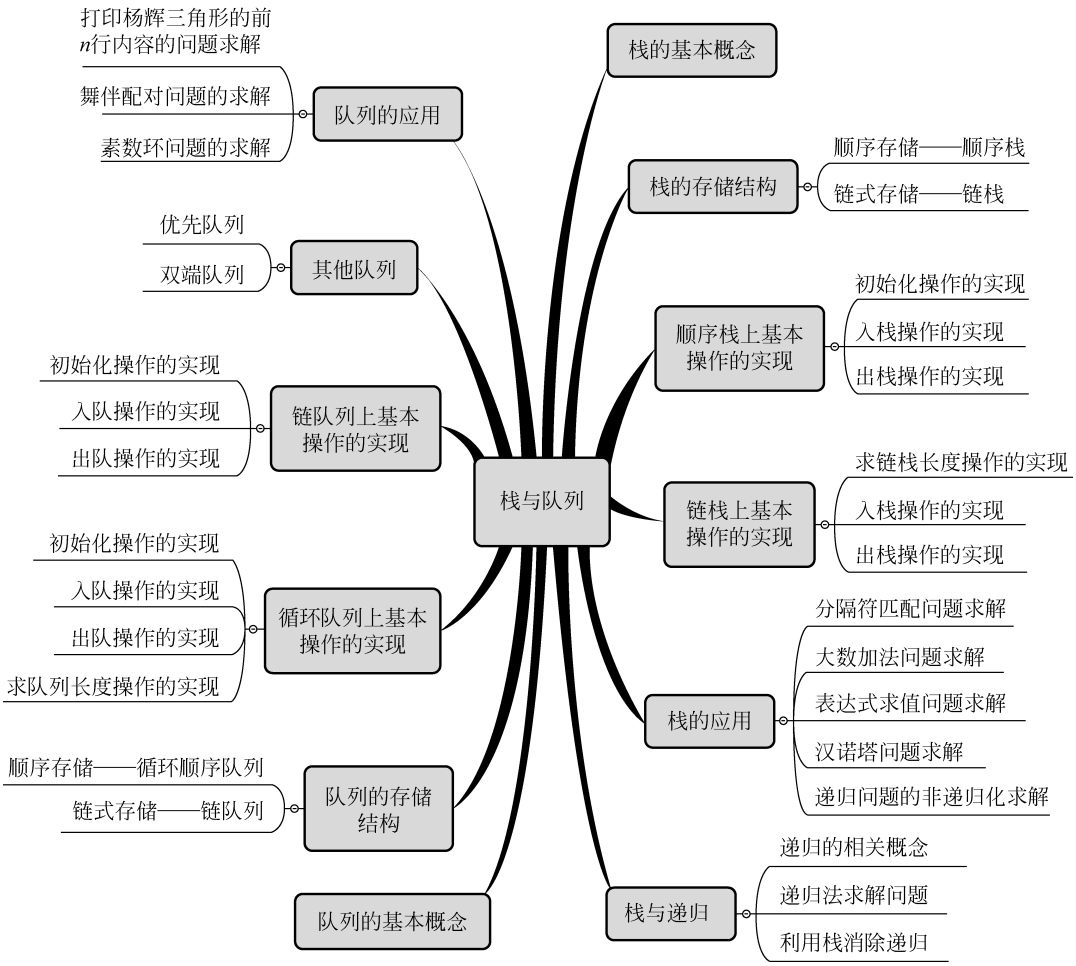
第3章 栈与队列

栈(Stack)和队列(Queue)是两种特殊的线性表,是两种应用非常广泛且极为重要的线性结构。例如,递归函数调用之间的链接和信息交换、编译器对程序的语法分析过程、操作系统实现对各种进程的管理等,都要应用到栈或队列。栈和队列与线性表之间的不同之处在于:它们可被看成是两种操作受限的特殊线性表,其特殊性体现在它们的插入和删除操作都是控制在线性表的一端或两端进行。正是由于其简洁性和规范性,栈与队列成为构建更复杂、更高级数据结构的基础。

【本章主要知识导图】



3.1 栈

3.1.1 栈的概念

栈是一种特殊的线性表，栈中的数据元素以及数据元素间的逻辑关系和线性表相同，两者之间的差别在于：线性表的插入和删除操作可以在表的任意位置进行，而栈的插入和删除操作只允许在表的尾端进行。其中，栈中允许进行插入和删除操作的一端称为**栈顶**(Stack top)，另一端称为**栈底**(Stack bottom)。假设栈中的数据元素序列为$\{a_1,a_2,a_3,\cdots,a_n\}$，则$a_1$称为栈底元素，$a_n$称为栈顶元素，$n$为栈中数据元素的个数或称为**栈的长度**(当$n=0$时，栈为空)。通常，我们将栈的插入操作称为**入栈**或进栈或压栈，而将栈的删除操作称为**出栈**或退栈或弹栈，如图 3-1 所示。

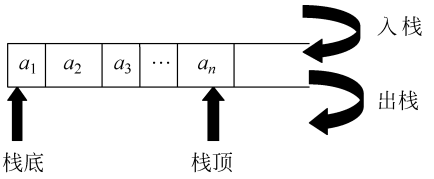


图 3-1　栈及其操作的示意图

从栈的概念可知，每次最先入栈的数据元素总是被放在栈的底部，成为栈底元素；而每次最先出栈的总是那个放在栈顶位置的数据元素，即栈顶元素。因此，栈是一种**后进先出**(Last-In First-Out，LIFO)，或**先进后出**(First-In Last-Out，FILO)的线性表。

在现实生活中有许多类似栈的实例。例如，叠成一摞的椅子或盘子可被看作是一个栈，任何时候要取出或叠放一把椅子或一个盘子只能在它的顶端实施，满足“后进先出”或“先进后出”的原则；还有火车的调度，以及抽纸盒中纸的放入和取出也都可被视为一个栈的模型。

尽管栈的特性降低了栈的插入与删除操作的灵活性，但这种限制却使栈的操作更为有效、更易实现。在计算机的算法中栈也经常可见，它成为算法设计的基础出发点。例如，浏览器对用户当前访问过的地址的管理，键盘缓冲区中对键盘输入信息的管理，文本编辑器中对用户的编辑操作，表达式的求值等，都采用了栈式结构。

思考：假设有编号为a、b、c和d的 4 辆列车，顺序进入一个栈式结构的站台，请写出这 4 辆列车开出车站的所有可能的顺序。

注意：一组元素序列依次入栈，并不能保证元素出栈的次序与其入栈的次序总相反，可以有多种出栈次序。若有n个元素依次进栈，则此n个元素出栈的可能次序有$\frac{1}{n+1}\times C_{2n}^{n}=\frac{1}{n+1}\frac{(2n)!}{n!\times n!}$种。出栈的次序是由每个元素之间的入栈、出栈序列所决定，只有当所有元素入栈后再全部出栈才能使元素进栈的次序与其出栈的次序正好相反，而当操作为“入栈、出栈、入栈、出栈……”时，元素进栈的次序与其出栈的次序是一致的。

3.1.2 栈的抽象数据类型描述

栈仍是由 $n(n\geqslant 0)$个数据元素所构成的有限序列,其数据元素的类型可以任意,只要是同一种类型即可。根据栈的定义,栈的抽象数据类型描述如下:

```
ADT Stack {
    数据对象:D = { ai | ai∈SElemType, 1≤i≤n,n≥0,SElemType 是约定的栈中数据元素类型,使
    用时用户需根据具体情况进行自定义}
    数据关系:R = {< ai,ai + 1 > | ai ,ai + 1∈D,1≤i≤n - 1}
    基本操作:
    InitStack(&S),初始化操作: 创建一个空栈 S.
    DestroyStack(&S),销毁操作: 释放一个已经存在的栈 S 的存储空间.
    ClearStack(&S),清空操作: 将一个已经存在的栈 S 置为空栈.
    StackEmpty (S),判空操作: 判断栈 S 是否为空.若为空,则函数返回 TRUE;否则,函数返回 FALSE.
    StackLength(S),求栈的长度操作: 求栈 S 中数据元素的个数并返回其值.
    GetTop(S, &e),取栈顶元素操作: 读取栈顶元素,并用 e 返回其值.
    Push(&S, e),入栈操作: 将数据元素 e 插入栈 S 中,并使其成为新的栈顶元素.
    Pop(&S,&e),出栈操作: 删除并用 e 返回栈顶元素.
    DisplayStack(S),输出操作: 输出栈 S 中各个数据元素的值.
    }ADT Stack
```

下面分别从顺序和链式两种不同的存储结构讨论栈的抽象数据类型的实现方法。其中,采用顺序存储结构的栈称为顺序栈,采用链式存储结构的栈称为链栈。

3.1.3 顺序栈及其基本操作的实现

1. 顺序栈的存储结构描述

与顺序表一样,顺序栈也可以用数组来实现。假设数组名为 base。由于入栈和出栈操作只能在栈顶进行,为操作方便需再增加一个变量 top 来指示栈顶元素的位置。top 有两种定义方式:一种是将其设置为指示栈顶元素存储位置的下一存储单元的位置;另一种是将其设置为指示栈顶元素的存储位置,本书中采用前一种方式来表示 top 的定义。为此可用下述的类型说明作为顺序栈的动态存储结构描述。

```
#define STACK_INIT_SIZE 80          //栈初始分配空间的容量
#define STACKINCREMENT 10           //栈增加分配空间的量
typedef struct {
    SElemType * base;               //栈的存储空间基地址(栈底指针)
    SElemType * top ;               //指示栈顶元素的下一存储单元的位置(栈顶指针)
    int stacksize;                  //栈当前的存储空间容量
}SqStack;
```

其中 SqStack 为顺序栈的类型名。根据上述描述,对于非空栈 $S=(39,17,30,65,35,54)$初始状态的顺序存储结构可用图 3-2 表示。

特别说明:对于已知如图 3-2 所示的顺序栈 S,其存储空间基地址的直接访问形式为 S.base;栈顶指针的直接访问形式为 S.top;栈的当前存储空间容量的直接访问形式为 S.stacksize。

2. 顺序栈基本操作的实现

对于一个说明为 SqStack 类型的栈 S,再结合图 3-2,首先需明确以下几个关键问题。

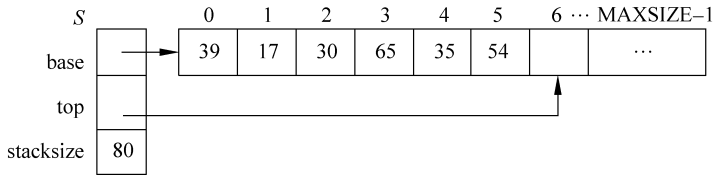


图 3-2　顺序栈 S 初始状态的存储结构示意图

(1) 顺序栈判空的条件是 S. top == S. base。

(2) 顺序栈为满的条件是 S. top－S. base≥S. stacksize。

(3) 栈的长度为 S. to－S. base。

(4) 栈顶元素就是指针 top 所指的前一个存储单元的值 *(S. top－1)。

在理解上述问题后,要描述顺序栈的置空、判空、求长度和取栈顶元素操作的算法就非常简单,请读者自行完成。下面仅对顺序栈的初始化、入栈和出栈操作的实现方法进行分析。

1) 顺序栈的初始化操作

顺序栈初始化操作 InitStack(&S)的实现与顺序表的初始化操作类似,先需按常量 STACK_INIT_SIZE 预定义值的大小分配数组空间,然后再将 top 和 stacksize 两个域置上相应值,使其形成一个空栈。操作结果将形成如图 3-3 所示的顺序栈。

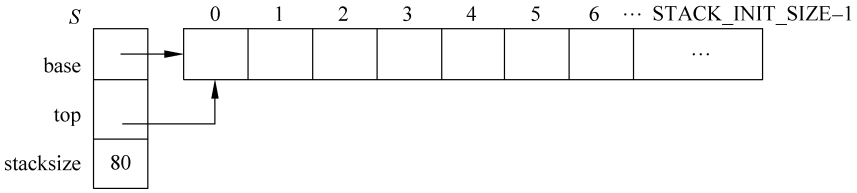


图 3-3　空顺序栈的存储结构示意图

【算法 3-1】 顺序栈的初始化操作算法。

```
Status InitStack( SqStack &S)
// 创建一个空的顺序栈 S
{  // 分配预定义大小的存储空间
    S.base = (SElemType * ) malloc (STACK_INIT_SIZE * sizeof (SElemType));
    if (!S.base)                          //如果空间分配失败
    {  printf("OVERFLOW");
       return ERROR;
    }
    S.top = S.base;                       //置当前栈顶指针指向栈底的位置
    S.stacsize = STACK_INIT_SIZE;         //置当前分配的存储空间容量为 STACK_INIT_SIZE 的值
    return OK;
}/ /算法 3 - 1 结束
```

2) 顺序栈的入栈操作

入栈操作 Push(&S,e)的基本要求是将数据元素 e 插入顺序栈 S 中,使其成为新的栈顶元素。完成此处理的主要步骤归纳如下。

(1) 判断顺序栈是否已满,若不满,则转(2),若已满,则对栈空间进行扩充,扩充成功后再转(2)。

(2) 将新的数据元素 e 存入 S.top 所指向的存储单元,使其成为新的栈顶元素。

(3) 栈顶指针 S.top 后移一位。

完成(2)和(3)所对应的语句为"*S.top++=e;"。

图 3-4 显示了在顺序栈上执行入栈操作时,栈顶元素和栈顶指针的变化情况。

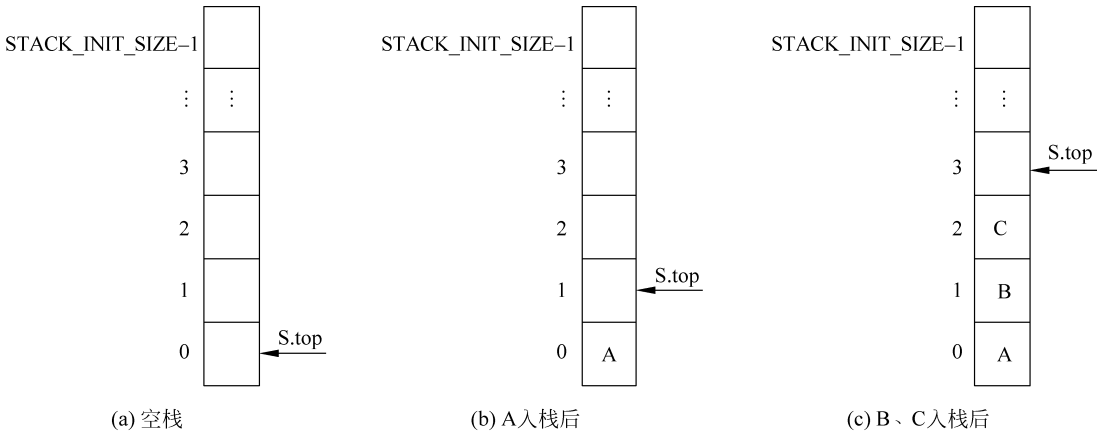


图 3-4 执行入栈操作时栈顶元素和栈顶指针的变化

【算法 3-2】 顺序栈的入栈操作算法。

```
Status Push(SqStack &S, SElemType e)
   // 在顺序栈 S 中插入新的元素 e, 使其成为新的栈顶元素
{  if (S.top - S.base >= S.stacksize)      //当前存储空间已满,则扩充空间
     {  S.base = (SElemType *)realloc(S.base,(S.stacksize + STACKINCREMENT) * sizeof (SElemType));
        if (!S.base)                       //如果空间分配失败
        {     printf("OVERFLOW");
              return ERROR;
        }
        S.top = S.base + S.stacksize;      //修改增加空间后的基址
        S.stacksize += STACKINCREMENT;     //修改增加空间后的存储空间容量
     }
      *S.top++ = e;                        //e 压栈后,top 指针再后移一位
      return OK;
} // 算法 3-2 结束
```

3) 顺序栈的出栈操作

出栈操作 Pop(&S,&e)的基本要求是将栈顶元素从栈 S 中移去,并用 e 返回被移去的栈顶元素值。完成此处理的主要步骤归纳如下。

(1) 判断顺序栈 S 是否为空,若为空,则报告栈的状态后结束算法,否则转②。

(2) 先将 S.top 减 1,使其栈顶指针指向栈顶元素。

(3) 用 e 返回 S. top 所指示的栈顶元素的值。

完成(2)和(3)所对应的语句为“e =*-- S. top;”。

图 3-5 显示了图 3-4(c)中顺序栈在执行出栈操作时,栈顶元素和栈顶指针的变化情况。

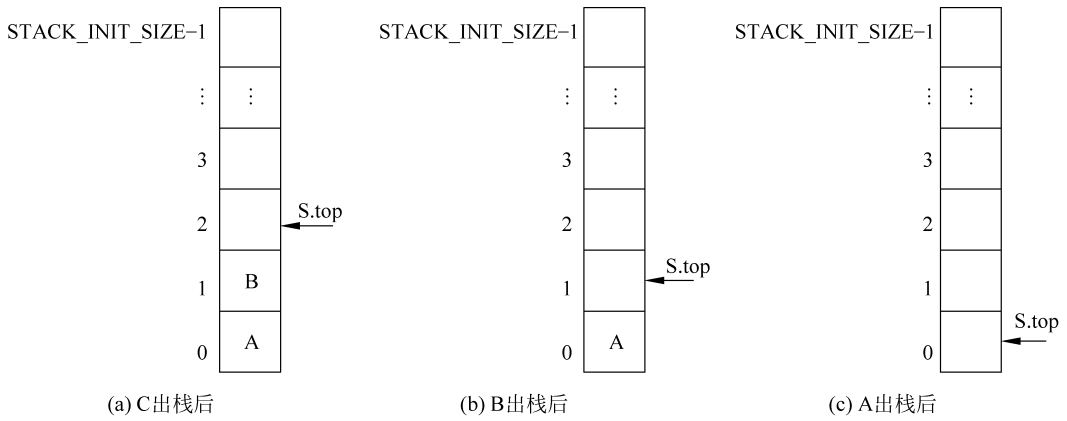


图 3-5 执行出栈操作时栈顶元素和栈顶指针的变化示意图

【算法 3-3】 顺序栈的出栈操作算法。

```
Status Pop(SqStack &S, SElemType &e)
   // 删除顺序栈 S 中的栈顶数据元素,并用 e 返回其值
{ if (S.base == S.top)                    //如果栈空
    {  printf("The Stack is NULL!\n");
       return ERROR;
    }
   e = * -- S.top;                        //删除栈顶元素并用 e 返回其值
   return OK;
} //算法 3-3 结束
```

所有有关顺序栈操作算法的时间复杂度都为 $O(1)$。

思考:如果将顺序栈的存储结构描述为静态的顺序存储:

```
#define MAXSIZE 80              //栈预分配空间的容量
typedef struct {
    SElemType base[MAXSIZE];    // 栈的存储空间
    int top;                    //指示栈顶元素的下一存储位置,是下标
}SqStackTp;
```

则对于一个说明为 SqStackTp 类型的栈 S 需明确以下问题。

(1) 顺序栈判空的条件是 S. top == 0 或 S. top == S. base。

(2) 顺序栈判满的条件是 S. top == MAXSIZE 或 S. top－S. base > = MAXSIZE。

(3) 栈的长度为 S. top 的值。

(4) 栈顶元素就是以 S. top－1 为下标的数组元素值 S. base[S. top－1]。

根据上述存储结构描述,请读者自行编写顺序栈对应的入栈和出栈操作算法。

3.1.4 链栈及其基本操作的实现

1. 链栈的存储结构

由于在栈中,入栈和出栈操作只能限制在栈顶进行,所以,宜采用不带表头结点的单链表作为栈的链式存储结构,而且直接将栈顶元素放在单链表的头部成为首结点。图 3-6 给出了链栈的存储结构示意图。

注意:链表的头指针 top 指向栈顶元素结点,称为**栈顶指针**;链中每一个结点的 next 域存储的不是指向其逻辑序列中后继结点的指针,而是指向其前驱结点的指针。

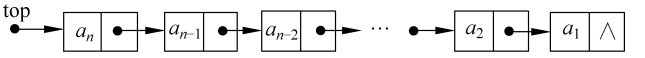


图 3-6 链栈的存储结构示意图

2. 链栈的存储结构描述

从图 3-6 可知,链栈中的结点结构与单链表中的结点结构完全相同。为此链栈的存储结构可描述为:

```
typedef int SElemType;          //为后续描述算法方便,特将数据元素类型自定义为整型
typedef struct SNode
{    SElemType data;
     struct SNode * next;
}SNode, * LinkStack;
```

其中,SNode 为链栈的结点类型名, LinkStack 为指向链栈中结点的指针类型名。

3. 链栈基本操作的实现

在讨论链栈上基本操作的实现方法时,读者要抓住链栈的特点:链栈是用不带表头结点的单链表作为存储结构的,作为一个空栈,其栈顶指针 top 就是空指针,作为一个非空栈其栈顶指针 top 直接指向栈顶元素结点。明确这些特点后,链栈上基本操作的实现方法就与在单链表上相应操作的实现方法相同。下面仅介绍求链栈的长度、入栈和出栈操作的实现方法及其算法描述。

1) 求链栈的长度操作

求链栈长度操作 StackLength(S)的基本要求是计算出链栈 S(S 为栈顶指针)中所包含的数据元素的个数并返回其值。此操作的基本思想与求单链表的长度相同:引进一个指针 p 和一个计数变量 length,p 的初始状态指向栈顶元素,length 的初始值为 0;然后逐个计数,即 p 沿着链栈中的后继指针进行逐个结点移动,同时 length 逐个加 1,直至 p 指向空为止,此时 length 值即为链栈的长度值。具体的实现算法描述如下。

【算法 3-4】 求链栈的长度操作算法。

```
int StackLength(LinkStack S)
 //返回链栈中数据元素的个数
{   int length = 0;
    LinkStack p = S;                          //p 指针指向链栈中第一个元素结点
    while(p)
    {   length++;                             //长度加 1
        p = p -> next;                        //p 指针后移
```

```
    }
    return length;
}//算法 3-4 结束
```

2）链栈的入栈操作

链栈的入栈操作 Push(&S,e)的基本要求是将数据元素值为 e 的新结点插入链栈 S 的栈顶，使其成为新的栈顶元素。此操作的基本思想与不带头结点的单链表上的插入操作类似，不相同的仅在于插入的位置对于链栈来说，是限制在表头(栈顶)进行的。链栈的入栈操作步骤归纳如下。

(1) 产生数据域值为 e 的新结点 p。

(2) 将新结点 p 直接链接到链栈的头部(栈顶)，并使其成为新的栈顶结点(首结点)。

图 3-7 显示了链栈的入栈操作后状态的变化情况。

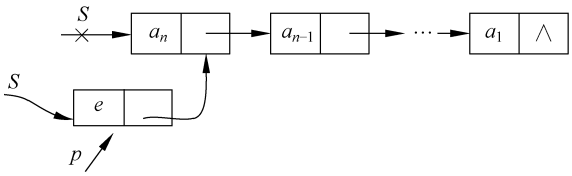


图 3-7　链栈的入栈操作状态变化示意图

【算法 3-5】 链栈的入栈操作算法。

```
Status Push(LinkStack &S, SElemType e)
 // 在链栈 S 的栈顶插入新的元素 e,使其成为新的栈顶元素
{  LinkStack p = (LinkStack)malloc(sizeof(SNode)); //为新结点 p 分配空间
   if (!p)                                         //空间分配失败
        return ERROR;
   p->data = e;
   p->next = S;                                    //修改链,让新结点插入链栈的栈顶
   S = p;                                          //使新结点成为新的栈顶结点
   return OK;
}//算法 3-5 结束
```

3）链栈的出栈操作

链栈的出栈操作 Pop(&S,&e)的基本要求是将栈顶结点(首结点)从链栈中移去，并用 e 返回该结点的数据域的值。此操作的基本思想与不带头结点的单链表上的删除操作类似，不相同的在于待删除的结点仅限制为链栈的栈顶结点。链栈的出栈操作步骤归纳如下。

(1) 判断链栈是否为空，若为空，则报告栈的状态后结束算法；否则，转(2)。

(2) 确定被删结点为栈顶结点。

(3) 修改相关指针域的值，使栈顶结点从链栈中移去，并用 e 返回被删的栈顶结点的数据域的值。

图 3-8 显示了链栈出栈操作后状态的变化情况。

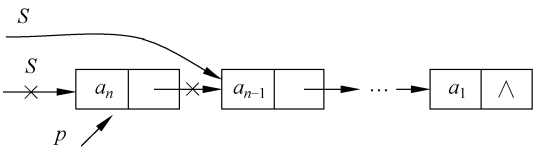


图 3-8　链栈的出栈操作状态变化示意图

【算法 3-6】 链栈的出栈操作算法。

```
Status Pop(LinkStack &S, SElemType &e)
 // 删除链栈 S 中的栈顶数据元素,并用 e 返回其值
{   if(S == NULL)
    {   printf("The Stack is NULL!\n");
        return ERROR;
    }
    LinkStack p = S;                          //p 指针指示链栈的栈顶元素结点
    e = p -> data;                            //用 e 保存待删结点的数据元素值
    S = p -> next;                            //栈顶指针 S 指向原栈顶元素的下一个元素结点
    free(p);                                  //释放待删结点空间
    return OK;
}//算法 3 - 6 结束
```

说明：链栈置空、判空、取栈顶元素、入栈与出栈操作的时间复杂度都为 $O(1)$；求栈的长度和栈的输出操作的时间复杂度为 $O(n)$，其中 n 为栈的长度。

3.1.5　栈的应用

栈是各种软件系统中应用极其广泛的数据结构之一，只要涉及先进后出处理特征的问题都可利用栈式结构。例如，函数递归调用中的地址和参数值的保存、文本编辑器中 undo 序列的保存、网页访问历史的记录保存、在编译软件设计中的括号匹配及表达式求值等问题。下面通过 3 个实例来说明栈在解决实际问题中的运用。

【例 3-1】 分隔符匹配问题：设计判断 C 语言语句中分隔符是否匹配的算法。

【分析】 分隔符的匹配是任意编译器的一部分，若分隔符不匹配，则程序就不可能正确。C 语言程序中有以下分隔符：圆括号“(”和“)”，方括号“[”和“]”，大括号“{”和“}”，以及注释分隔符“/ * ”和“ * /”。

以下是正确使用分隔符的例子：

```
a = b + (c + d) * (e - f);
s[4] = t[a[2]] + u/((i + j) * k);
if (i!= (n[8] + 1)) {p = 7; / * initialize p * / q = p + 2;}
```

以下是分隔符不匹配的例子：

```
a = (b + c/ (d * e) * f;                                  //左分隔符多余
s[4] = t[a[2]] + u/(i + j) * k);                          //右分隔符多余
while (i!= (n[8] + 1)] {p = 7; / * initialize p * / q = p + 2;}    //左、右分隔符不匹配
```

一个分隔符和它所匹配的分隔符可以被其他的分隔符分开，即分隔符允许嵌套。因此，一个给定的右分隔符只有在其前面的所有右分隔符都被匹配上后才可以进行匹配。例如，

条件语句 if(i! =(n[8]+1))中,第一个左圆括号必须与最后一个右圆括号相匹配,而且这只有在第二个左圆括号与倒数第二个右圆括号相匹配后才能进行;依次地,第二个左括号的匹配也只有在第三个左方括号与倒数第三个右方括号匹配后才能进行。可见,最先出现的左分隔符在最后才能进行匹配,这个处理与栈式结构的先进后出的特性相吻合,因此可借助栈来保存扫描过程中还未被匹配的左分隔符。分隔符匹配的主要操作步骤归纳如下。

从左到右扫描待判断的 C 语言语句,从语句中不断地读取字符,每次读取一个字符。若发现它是左分隔符,则将它压入栈中;当从语句中读到一个右分隔符时,则分以下两种情况处理。

(1) 当前栈非空,则弹出栈顶的左分隔符,并且查看它是否和当前右分隔符匹配,若它们不匹配,则匹配失败,否则当前的分隔符匹配成功。

(2) 当前栈为空,则表示栈中没有左分隔符与当前扫描到的右分隔符匹配,表示右分隔符多余,匹配失败。

如果语句的所有的字符都读入后,栈中仍留有左分隔符,表示左分隔符多余,匹配失败;如果语句的所有的字符都读入后,栈为空(即所有左右分隔符都已经匹配),则表示匹配成功。

根据上述步骤,再设定 C 语句用字符数组 str 存放,栈中的数据元素类型 SElemType 也设定为字符数组类型,则可得算法 3-7。

【算法 3-7】 例 3-1 中的相关设计算法。

```
typedef char SElemType[3];
int LEFT = 0;                                          // 记录分隔符为"左"分隔符
int RIGHT = 1;                                         // 记录分隔符为"右"分隔符
int OTHER = 2;                                         // 记录其他字符

int VerifyFlag(char * str)
// 判断 C 语句 str 中分隔符的类型,有 3 种:"左""右""其他"
{  if (!strcmp("(",str) || !strcmp("[",str)||!strcmp("{",str)||!strcmp("/*",str))
                                                       // 左分隔符
        return LEFT;
   else
        if (!strcmp(")",str) || !strcmp("]",str)|| !strcmp("}",str)||!strcmp("*/",str))
                                                       // 右分隔符
              return RIGHT;
        else                                           // 其他的字符
              return OTHER;
} // VerifyFlag

bool Matches(char * str1, char * str2)
// 检验左分隔符 str1 和右分隔符 str2 是否匹配,若匹配返回 TRUE,否则返回 FALSE
{  if ((!strcmp(str1,"(")&&!strcmp(str2,")"))||(!strcmp(str1,"{")&&!strcmp(str2,"}"))
     ||(! strcmp ( str1," [ ") &&! strcmp ( str2,"]")) || (! strcmp ( str1,"/* ") &&! strcmp
(str2,"*/")))
         return TRUE;
   else
         return FALSE;
}// Matches
```

```
bool IsLegal(char *str)
// 判断C语句str中的分隔符是否匹配,若匹配则返回TRUE,否则返回FALSE
{  if (strcmp("",str) && str != NULL)
   {  SqStack S;
      InitStack(S);                                  // 新建一个顺序栈
      int length = strlen(str);
      for (int i = 0; i < length; i++)
      {  char c = str[i];                            // 指定索引处的char值
         char t[3] = {c};                            // c字符转化成字符串t
         if (i != length-1)                          // c不是最后一个字符
         {  if (('/' == c && '*' == str[i+1])|| ('*' == c && '/' == str[i + 1]))
                                                     // 是分隔符"/*"或"*/"
            {  t[1] = str[i+1];                      // 将后一个字符连接到前一个字符的后面
               t[2] = '\0';                          // 为t串置上结束符
               ++i;                                  // 跳过一个字符
            }
         }
         if (LEFT == VerifyFlag(t))                  // 为左分隔符
            Push(S,t);                               // 压入栈
         else if (RIGHT == VerifyFlag(t))            // 为右分隔符
             {  if (StackEmpty(S))                   //栈空,即右分隔符多余
                {  printf("错误:右分隔符多余!\n"); // 报错
                   return FALSE;
                }
                else                                 // 右分隔符与栈顶元素不匹配
                {  SElemType t1;
                   Pop(S,t1);                        //栈顶左分隔符出栈并用t1记录
                   if (!Matches(t1,t))               //如果栈顶的左分隔符与右分隔符不匹配
                   {  printf("错误:左、右分隔符不匹配!\n");
                                                     // 报错
                      return FALSE;
                   }
                }
             }
         }
         if (!StackEmpty(S))                         // 栈非空,即栈中存在没有匹配的字符
         {  printf("错误:左分隔符多余!\n");        // 报错
            return FALSE;
         }
         else return TRUE;
      }
      else
      {  printf("C语言语句为空!\n");                // 输出异常
        return TRUE;
      }
}//算法3-7结束
```

【例3-2】 大数加法问题：设计算法实现两个大数的加法运算。

【分析】 整型数是有最大上限的。所谓大数是指超过整型数最大上限的数,例如

18 452 543 389 943 209 752 345 473 和 8 123 542 678 432 986 899 334 就是两个大数，它们是无法用整型变量来保存的，更不用说保存它们相加的和了。为解决两个大数的求和问题，可以把两个加数看成是数字字符串，将这些数的相应数字（从高位到低位）存储在两个栈中，并从两个栈中弹出对应位的数字，并依次执行加法即可得到结果。图 3-9 显示了以 784 和 8465 为例进行加法的计算过程。

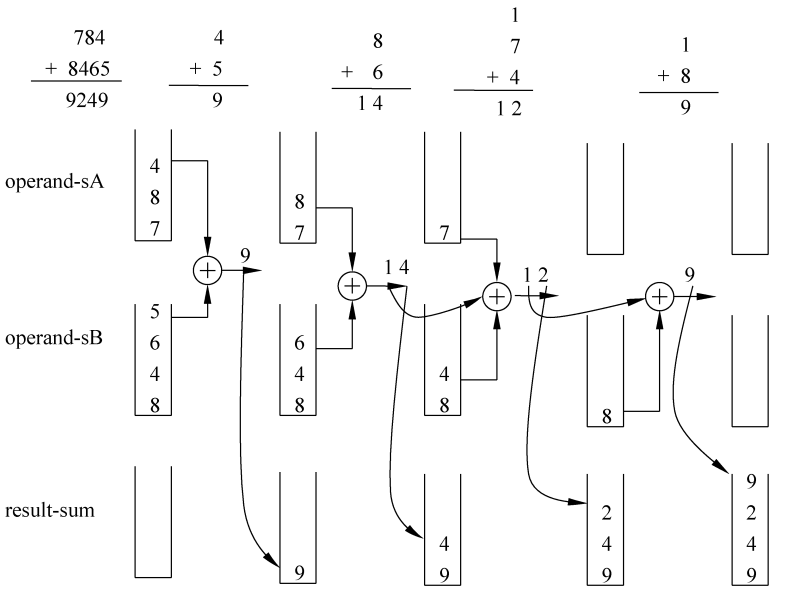


图 3-9　使用栈将 784 和 8465 相加

对于两个大数的加法，其操作步骤归纳如下。

（1）将两个加数的相应位从高位到低位依次压入栈 sA 和 sB 中。

（2）若两个加数栈均非空，则依次从栈中弹出栈顶数字并相加，和存入变量 partialSum 中；若和有进位，则将和的个位数压入结果栈 sum 中，并将进位数加到下一位数字相加的和中；若和没有进位，则直接将和压入结果栈 sum 中。

（3）若某个加数栈为空，则将非空加数栈中的栈顶数字依次弹出与进位相加，和的个位数压入结果栈 sum 中，直到此该栈为空为止。若最高位仍有进位，则最后将 1 压入栈 sum 中。

（4）若两个加数栈都为空，则栈 sum 中保存的就是计算结果。注意：栈顶是结果中的最高位数字。

根据上述步骤，再按需要设定两个大数的数字字符串存放在字符数组 a 和 b 中，栈中的数据元素类型 SElemType 仍设定为 int 类型，则可得算法 3-8。

【算法 3-8】 例 3-2 中的相关设计算法。

```
LinkStack NumSplit(char * str)
 // 将数字字符串 str 以单个字符的形式放入栈中,并去除字符串中空格,返回以单个字符为元素
 // 的栈
{  LinkStack s ;
   InitStack(s);                              //创建一个空的链栈
```

```
   for (int i = 0; i < strlen(str); i++)
   {  char c = str[i];                          // 指定索引处的 char 值
      if (' ' == c)                             // 去除空格
            continue;
      else if ('0' <= c && '9' >= c)            // 数字放入栈中
             Push(s,c);
      else                                      // 非数字型字符
      {  printf("错误:串中有非数字型字符!\n");
         exit(-1);
      }
   }
   return s;
}  //NumSplit

char *Add(char *a, char *b)
// 求两个大数的和,加数和被加数以字符串的形式输入(允许大数中出现空格)
// 计算的结果也以字符串的形式返回.
{  SElemType c1,c2,t;
   LinkStack sum;                               //大数的和存入栈 sum 中
   InitStack(sum);                              //创建一个空栈 sum
   LinkStack sA = NumSplit(a);                  //加数字符串以单个字符的形式存入栈 sA 中
   LinkStack sB = NumSplit(b);                  //被加数字符串以单个字符的形式存入栈 sB 中
   int partialSum;                              //记载两个位的求和
   bool isCarry = false;                        //进位标示
   while (!StackEmpty(sA) && !StackEmpty(sB))   //加数栈和被加数栈同时非空
   {  //下面对于栈中两个位求和,并在栈中去除加数和被加数中的该位
      Pop(sA,c1);                               //从栈顶取出加数 c1
      Pop(sB,c2);                               //从栈顶取出被加数 c2
      partialSum = c1 - '0' + c2 - '0';         //加数与被加数都转换成数字后再相加
      if (isCarry) {                            //低位进位
            partialSum++;                       //进位加到此位上
            isCarry = false;                    //重置进位标志
      }
      if (partialSum >= 10) {                   //需要进位
            partialSum -= 10;
            Push(sum,partialSum);
            isCarry = true;                     //标示有进位
      }
     else                                       //位和不需要进位
            Push(sum,partialSum);               //和放入栈中
}
LinkStack temp = !StackEmpty(sA) ? sA : sB;     //引用指向加数和被加数中非空栈
while (!StackEmpty(temp))
{   if (isCarry)                                //最后一次执行加法运算中需要进位
        {  Pop(temp,t);                         //取出加数或被加数中没有参加运算的位
           t = t - '0';                         //字符转换成数字
           ++t;                                 //进位加到此位上
           if (t >= 10)                         //需要进位
           {  t -= 10;
              Push(sum,t);
           }
```

```
            else
             {  Push(sum,t);
                isCarry = false;                    //重置进位标志
             }
          }
          else                                      //最后一次执行加法运算中不需要进位
          {  Pop(temp,t);
             Push(sum,t-'0');                  //把加数或被加数中非空的值转换成数字后放入和栈中
          }
       }
       if (isCarry)                                 //最高位需要进位
             Push(sum,1);                           //进位放入栈中
       char str[100];                               //说明一个字符数组
       int i=0;
       while (!StackEmpty(sum))                     //把栈中元素转化成字符串
       {  SElemType e;
          Pop(sum,e);
          str[i++]=e+'0';                        //将数字 e 先转换成字符连接到字符串 str 的尾部
       }
       str[i]='\0';                                 //为串置上结束符
       return str;
} //算法 3-8 结束
```

【例 3-3】 表达式求值问题：设计算法实现算术表达式的求值。

【问题分析】 算术表达式是由操作数、算术运算符和分隔符所组成的式子。为了方便，下面的讨论仅限于含有二元运算符且操作数是一位整数的算术表达式的运算。

表达式在计算机中一般有中缀表达式、后缀表达式和前缀表达式共 3 种表示形式。其中：中缀表达式是将运算符放在两个操作数的中间，这正是我们平时书写算术表达式的一种描述形式；后缀表达式(也称逆波兰表达式)是将运算符放在两个操作数之后；而前缀表达式是将运算符放在两个操作数之前。例如：中缀表达式 A+(B-C/D)*E，对应的后缀表达式为 ABCD/-E*+，对应的前缀表达式为+A*-B/CDE。

由于运算符有优先级，所以在计算机内部使用中缀表达式描述时，对计算是非常不方便的，特别是带括号时就更麻烦。而后缀表达式中既无运算符优先级，又无括号的约束问题，因为在后缀表达式中运算符出现的顺序正是计算的顺序，所以计算一个后缀表达式的值要比计算一个中缀表达式的值简单得多。由此，求算术表达式的值可以分成两步来进行，第一步先将原算术表达式转换成后缀表达式，第二步再对后缀表达式求值。下面分别对“如何计算后缀表达式的值”和“如何将算术表达式转换成后缀表达式”这两个问题进行分析讨论。

1. 计算后缀表达式的值

要计算后缀表达式的值比较简单，只要从左到右扫描后缀表达式，先找到运算符，再去找前面最后出现的两个操作数，从而构成一个最小的算术表达式进行运算。在计算过程中也需利用一个栈来保留后缀表达式中还未参与运算的操作数，此栈被称为**操作数栈**。现设定后缀表达式使用字符数组 postfix 存放，操作数栈采用顺序栈，计算后缀表达式值的主要步骤归纳如下。

(1) 初始化一个操作数栈为空栈。

(2) 从左到右顺序扫描后缀表达式中的每一项,根据它的类型做如下相应操作。

① 若该项是操作数,则将其压入操作数栈。

② 若该项是运算符,则从栈顶弹出两个操作数并分别作为第2个操作数和第1个操作数参与运算,再将运算结果重新压入操作数栈内。

(3) 重复步骤(2)直到后缀表达式扫描结束为止,则操作数栈中的栈顶元素即为后缀表达式的计算结果。

现要求计算结果为double类型,则应设定栈中数据元素类型SElemType也为double类型。按照以上的分析步骤,可得计算后缀表达式的值的算法3-9。

【算法3-9】 计算后缀表达式值的相关算法。

```
bool IsOperator(char c)
// 判断字符 c 是否为运算符
{   if ('+' == c || '-' == c || '*' == c || '/' == c || '^' == c|| '%' == c)
        return true;
   else
        return false;
}//IsOperator

double NumberCalculate(char * postfix)
// 计算后缀表达式 postfix 的值
{   double d,d1,d2,d3;
    char c;
    SqStack st;
    InitStack(st);                                          // 初始化一个操作数栈
    for (int i = 0; i < strlen(postfix); i++)
    {    c = postfix[i];                                    // 从后缀表达式中读取一个字符
         if (IsOperator(c))                                 // 当为操作符时
         {  Pop(st,d2);                                     // 取出两个操作数
            Pop(st,d1);
            if ('+' == c)                                   // 加法运算
                    d3 = d1 + d2;
            else if ('-' == c)                              // 减法运算
                        d3 = d1 - d2;
                    else if ('*' == c)                      // 乘法运算
                                d3 = d1 * d2;
                            else if ('/' == c)              // 除法运算
                                        d3 = d1 / d2;
                                    else if ('^' == c)      // 幂运算
                                              d3 = pow(d1, d2);
                                          else if ('%' == c)                //求模运算
                                                    d3 = (int)d1 % (int)d2;

            Push(st,d3);                                    //运算结果压栈
        }
        else                                                //当为操作数时
            Push(st,c-'0');                                 //数字字符转换成数字后压栈
    }//for
    Pop(st,d);                                              //从栈顶弹出最后的计算结果
    DestroyStack(st);                                       //销毁操作数栈
```

```
    return d;                                          // 返回计算结果
}// 算法 3-9
```

此算法如果测试输入的后缀表达式为 12＋52－＊22^/53%＋(算术表达式(1＋2)＊(5－2)/2^2＋5%3 的后缀表达式),则测试输出的计算结果为 4.25。

2. 将原算术表达式转换成后缀表达式

由于原算术表达式与后缀表达式中的操作数所出现的先后次序是完全一样的,只是运算符出现的先后次序不一样,所以转换的重点放在运算符的处理上。首先设定运算符的优先级如表 3-1 所示。

表 3-1　运算符的优先级

运算符	(（左括号）	＋(加)、－(减)	＊(乘)、/(除)、%(取模)	^(幂)
优先级	0	1	2	3

表 3-1 中的优先级从低到高依次用 0～3 的数字来表示,数字越大,表示其运算符的优先级越高。

要使运算符出现的次序与真正的算术运算顺序一致,就要使优先级高的以及括号内的运算符出现在前。为此,我们在把算术表达式转换成后缀表达式的过程中,使用了一个栈来保留还未送往后缀表达式的运算符,此栈被称为**运算符栈**。现设定表达式使用字符数组 exp 存放,运算符栈采用顺序栈,则原算术表达式转换成后缀表达式的主要步骤归纳如下:

(1) 初始化一个运算符栈为空栈。

(2) 从算术表达式 exp 中从左到右依次读取一个字符。

(3) 若当前字符是操作数,则直接送往后缀表达式。

(4) 若当前字符是左括号“(”时,将其压入运算符栈。

(5) 若当前字符为运算符时,则:

① 当运算符栈为空,则将其压入运算符栈。

② 当此运算符的优先级高于栈顶运算符,则将此运算符压入运算符栈;否则,重复弹出优先级更高的栈顶运算符并送往后缀表达式,再将当前运算符进栈。

(6) 若当前字符是右括号“)”时,反复将栈顶符号弹出,并送往后缀表达式,直到栈顶符号是左括号为止,再将左括号出栈并丢弃。

(7) 若读取还未完毕,则跳转到(2)。

(8) 若读取完毕,则将栈中剩余的所有运算符弹出并送往后缀表达式。

利用上述转换规则,将算术表达式(A ＋ B)＊(C － D)/E^F ＋ G%H,转换成后缀表达式的过程如表 3-2 所示。

表 3-2　算术表达式转换成后缀表达式的过程

步骤	算术表达式	运算符栈	后缀表达式	规则
1	(A ＋ B)＊(C － D)/E^F ＋ G%H	(		是左括号,进栈
2	A ＋ B)＊(C － D)/E^F ＋ G%H	(	A	是操作数,送往后缀表达式

续表

步骤	算术表达式	运算符栈	后缀表达式	规则
3	+ B) * (C - D)/E^F + G%H	(+	A	是运算符且优先级高于栈顶运算符,进栈
4	B) * (C - D)/E^F + G%H	(+	AB	是操作数,送往后缀表达式
5	) * (C - D)/E^F + G%H		AB+	是右括号,将栈中左括号之前的所有运算符送往后缀表达式并将栈中左括号弹出
6	* (C - D)/E^F + G%H	*	AB+	是运算符且栈为空,进栈
7	(C - D)/E^F + G%H	*(	AB+	是左括号,进栈
8	C - D)/E^F + G%H	*(	AB+C	是操作数,送往后缀表达式
9	- D)/E^F + G%H	*(-	AB+C	是运算符且优先级高于栈顶运算符,进栈
10	D)/E^F + G%H	*(-	AB+CD	是操作数,送往后缀表达式
11	)/E^F + G%H	*	AB+CD-	是右括号,将栈中左括号之前的所有运算符弹出送往后缀表达式并将栈中左括号弹出
12	/E^F + G%H	/	AB+CD- *	是运算符且优先级等于栈顶运算符,则弹出栈顶运算符送往后缀式,并将当前运算符进栈
13	E^F + G%H	/	AB+CD- * E	是操作数,送往后缀表达式
14	^F + G%H	/^	AB+CD- * E	是运算符且优先级高于栈顶运算符,进栈
15	F + G%H	/^	AB+CD- * EF	是操作数,送往后缀表达式
16	+ G%H	+	AB + CD - * EF^/	是运算符且优先级低于栈顶运算符,则重复弹出优先级更高的栈顶运算符送往后缀式,再将当前运算符进栈
17	G%H	+	AB + CD - * EF^/G	是操作数,送往后缀表达式
18	%H	+%	AB + CD - * EF^/G	是运算符且优先级高于栈顶运算符,进栈
19	H	+%	AB + CD - * EF^/GH	是操作数,送往后缀表达式
20	结束		AB + CD - * EF^/GH%+	弹出栈中剩余项并送往后缀表达式

假定已知的原算术表达式在计算机中用字符数组 exp 存放,且其中的操作数是由一位的整数组成,转换成的后缀表达式用字符数组 postfix 表示,则具体实现方法可用算法 3-10 描述。

【算法 3-10】 算术表达式转换成后缀表达式的相关算法。

```
int Priority(char c)
// 求运算符 c 的优先级
{  if (c == '^')                              //为幂运算
       return 3;
   if (c == '*' || c == '/' || c == '%')      //为乘、除、取模运算
       return 2;
    else if (c == '+' || c == '-')            //为加、减运算
       return 1;
    else                                      //其他
      return 0;
}//priority

void ConvertToPostfix(char * exp,char * postfix)
//将算术表达式 exp 转换为后缀表达式,并以 postfix 返回其值
{  SElemType ac;
   int j = 0;
   SqStack st;
   InitStack(st);                             //初始化一个运算符栈
   for (int i = 0; i < strlen(exp); i++)
   {  char c = exp[i];                        //从算术表达式中读取一个字符
      switch(c)
      { case '(':  Push(st,c);                //为左括号,进栈
                   break;
        case ')':  Pop(st,ac);                //为右括号,弹出栈顶元素
                   while(ac!= '(')            //一直到为左括号为止
                   {  postfix[j++] = ac;      //将 ac 送往后缀表达式
                      Pop(st,ac);
                   }
                   break;
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':
        case '%':                             //为运算符
              while (!StackEmpty(st))         //取出栈顶优先级高的运算符送往后缀表达式
              {  GetTop(st,ac);
                 if (Priority(ac)>=Priority(c))
                 {  postfix[j++] = ac;
                    Pop(st,ac);
                 }
                 else
                     break;
              }//while
              Push(st,c);                     //将当前扫描到的运算符进栈
              break;
        default:  postfix[j++] = c;           //为操作数,送往后缀表达式
        }//switch
    } //for
```

```
    while (!StackEmpty(st))                    //将栈中剩余的所有操作符送往后缀表达式
    {  Pop(st,ac);
       postfix[j++] = ac;
    }
    postfix = '\0';                            //为后缀表达式置上串结束标识
    DestroyStack(st);                          //销毁运算符栈
}//算法 3-10 结束
```

此算法如果测试输入的算术表达式为(1+2)*(5-2)/2^2+5%3,则测试输出的后缀表达式应为 12+52-*22^/53%+。

3.1.6 栈与递归

栈还有一个重要的应用就是在程序设计语言中实现函数调用。在 Windows 等大部分操作系统中,每个运行中的二进制程序都配有一个调用栈(call stack)和执行栈(execution stack)。借助调用栈可以跟踪属于同一程序的所有函数,记录它们之间的相互调用关系,并保证在每一调用实例执行完毕之后,可以准确返回。如图 3-10 显示了主函数 main()调用函数 A,函数 A 调用函数 B,函数 B 再自我调用时,其调用栈与执行栈的状况。

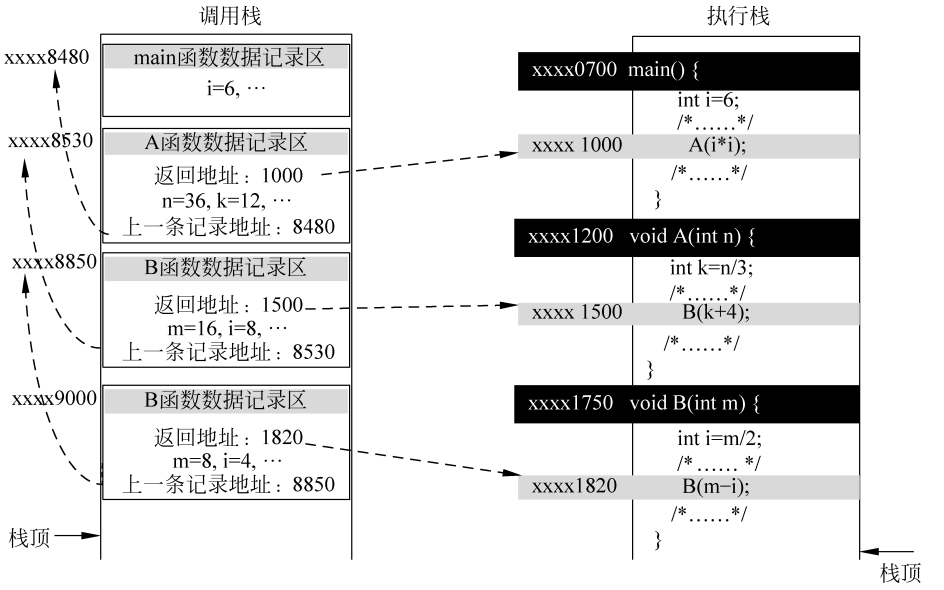


图 3-10 函数调用时栈的状况示意图

1. 递归的相关概念

若一个对象部分地包含它自己,或用它自己给自己定义,则称这个对象是递归定义的;若一个函数直接地或间接地调用自己,则称这个函数是递归的函数。类似地,如果一个算法直接或间接地调用自己,则称这个算法是递归算法。例如,以下 3 种情况都采用了递归方法。

1) 递归式的定义

在数学中,整数的阶乘定义如下:

$$n!=\begin{cases}1, & n=0 \qquad (3\text{-}1)\\ n*(n-1)!, & n>0 \qquad (3\text{-}2)\end{cases}$$

在上面的定义中，0!定义为1，如果 n 是一个大于0的整数，需要首先计算 $(n-1)!$，然后再将其与 n 相乘。为了求出 $(n-1)!$，要再一次应用定义。如果 $(n-1)>0$，则使用等式(3-2)；否则使用等式(3-1)。因此，对于一个大于0的整数 n，$n!$ 通过首先计算 $(n-1)!$（即 $n!$ 被简化成为一个比对自身更简单的形式），然后再将 $(n-1)!$ 乘以 n 获得；而 $(n-1)!$（若 $n-1>0$）又得先通过计算 $(n-2)!$，再乘以 $n-1$ 获得，如此层层递推下去，即可求得结果，这就是一个递归式的定义。

等式(3-1)中的解决方法是直接的，即等式中不包含阶乘符号，由它可直接求出解，这个等式称为**基本等式**。而等式(3-2)中的解决方法则是以一个比自身更简单的形式来定义的，因此称为**递归等式**。

总之，递归模式由基本等式和递归等式两部分组成。其中，必须包括一个（或多个）基本等式，基本等式是递归的终止条件（**递归出口**）；而递归等式（**递归体**）必须是由比自身问题更简单的形式定义，且最后必须能简化为一个基本等式。

2）递归式的数据结构

某些数据结构本身具有递归的特性，例如，前面学习的单链表就可看作是一个递归式的数据结构。因在一个单链表中，其中一个结点可以看作是指针域为NULL的单链表，也可以看作是其指针域仍然指向一个单链表。在后面要学习的树与二叉树等也是递归式的数据结构。

3）递归式的函数或算法

某些函数的定义也具有递归的特性，例如，二阶斐波那契(Fibonacci)数列：

$$\mathrm{Fib}(n)=\begin{cases}1, & n=1 \text{ 或 } n=2 \quad (3\text{-}3)\\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2), & \text{其他情形} \quad (3\text{-}4)\end{cases}$$

其中等式(3-3)是基本等式，也是终止递归的条件；等式(3-4)是递归等式，它是由比自身问题更小的形式定义的。

下面是求斐波那契数列的递归算法。

```
long  Fib(int n)
{  if (n==1||n==2)
        return 1;
    else
        return Fib(n-1)+Fib(n-2);
}
```

如图3-11显示了Fib(5)的执行过程。Fib最终返回的值为5，也就是第1和第2个数分别为1的斐波那契数列中的第5个数。

2. 递归法解决问题

递归是算法设计中常用的一种手段，是一种非常重要的解决问题的方法。它通常把一个大型复杂问题的描述和求解变得简洁和清晰。因此递归算法常常比非递归算法更易设计，尤其是当问题本身或所涉及的数据结构是递归定义的时候，使用递归方法是非常有效的解决途径。

【例3-4】 用递归法求解问题：设计算法实现汉诺塔(Hanoi)问题的求解。

n 阶汉诺塔问题描述：假设有3个分别命名为 x、y 和 z 的塔座，在塔座 x 上插有 n 个

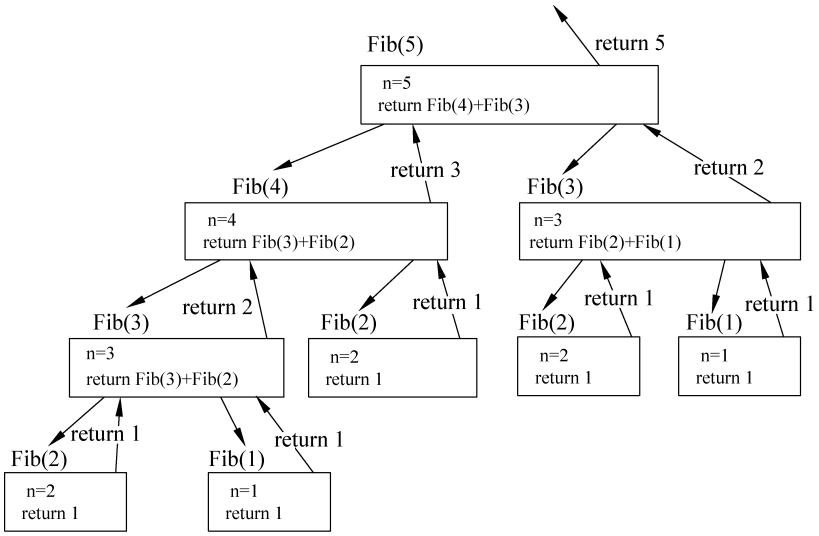


图 3-11　Fib(5)的执行过程

直径大小各不相同,且从小到大编号为 1,2,…,n 的圆盘。现要求将塔座 x 上的 n 个圆盘借助塔座 y 移至塔座 z 上,并仍按同样顺序叠排。圆盘移动时必须遵循下列规则。

(1) 每次只能移动一个圆盘。

(2) 圆盘可以插在 x、y 和 z 中的任何一个塔座上。

(3) 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

【问题分析】 当 $n=1$ 时,问题比较简单,只要将编号为 1 的圆盘从塔座 x 直接移动到塔座 z 上即可;当 $n>1$ 时,需利用塔座 y 作辅助塔座,若能先设法将压在编号为 n 的圆盘上的 $n-1$ 个圆盘从塔座 x 移到塔座 y 上,则可将编号为 n 的圆盘从塔座 x 移至塔座 z 上,然后再将塔座 y 上的 $n-1$ 个圆盘移至塔座 z 上。而如何将 $n-1$ 个圆盘从一个塔座移至另一个塔座是一个和原问题具有相同特征属性的问题,只是问题规模从 n 变为 $n-1$,减小了 1,因此可以用同样的方法求解。由此可知,求解 n 阶汉诺塔问题可以用递归分解的方法来进行。

【算法 3-11】 例 3-4 中的相关设计算法。

```
int c = 0;                              // 全局变量,对搬动计数
void Move(char x, int n, char z)
// 移动操作,将编号为 n 的圆盘从塔座 x 移到塔座 z
{
     printf("第 %d 次移动: %d 号圆盘, %c->%c\n",++c,n,x,z);
}//Move

void Hanoi(int n, char x, char y, char z)
// 将 n 个圆盘按规则从塔座 x 移到塔座 z 上,y 为辅助塔座
{  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
    }
}//算法 3-11 结束
```

以 Hanoi(3，'x'，'y'，'z')测试运行的结果如图 3-12 所示。

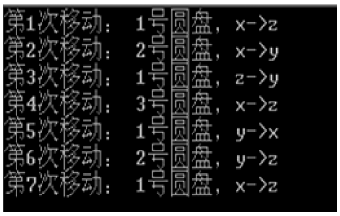


图 3-12　例 3-4 测试运行结果图

图 3-13 也给出了 Hanoi(3，'x'，'y'，'z')运行过程中圆盘的移动情况。

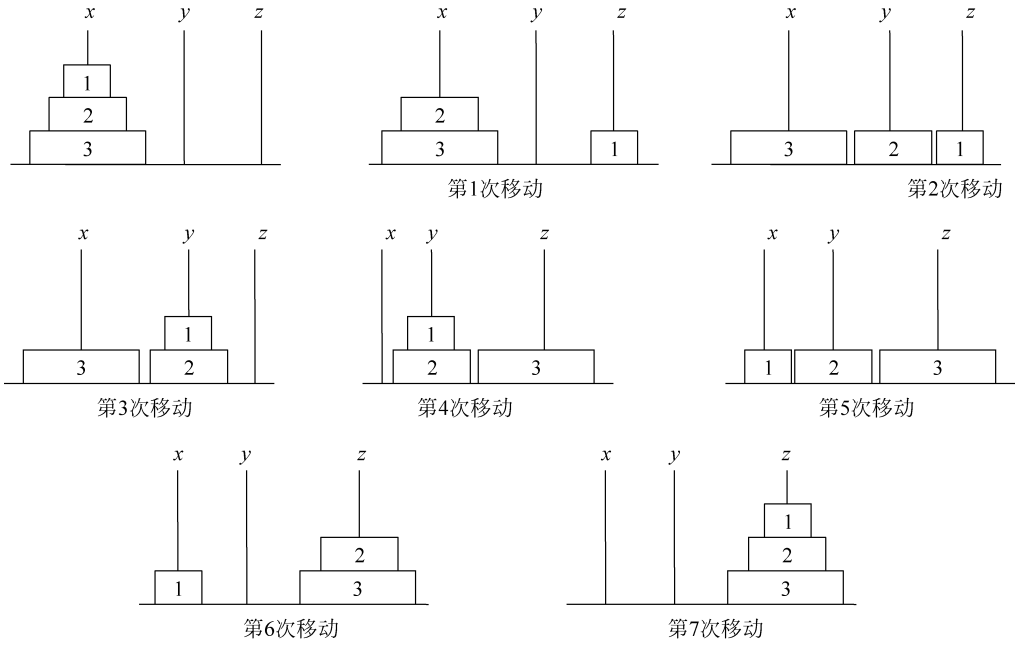


图 3-13　Hanoi(3，'x'，'y'，'z')运行过程中圆盘的移动情况示意图

【例 3-5】　用递归法求解问题：设计算法，实现以下递归函数的计算。

$$p_n(x)=\begin{cases}1, & n=0\\ 2x, & n=1\\ 2xp_{n-1}(x)-2(n-1)p_{n-2}(x), & n>1\end{cases}$$

【分析】　上面定义的函数本身就是一个递归函数，所以用递归方法设计算法就显得非常简单。

【算法 3-12】　例 3-5 中设计的递归算法。

```
1  double p(int n,double x)
2  //递归算法
```

```
3  {   if (n == 0)
4        return 1;
5     else if (n == 1)
6     return 2 * x;
7     else
8       return 2 * x * p(n - 1,x) - 2 * (n - 1) * p(n - 2,x);
9  }//算法 3 - 12
```

递归算法的实质就是把一个较为复杂的问题通过分解成规模更小的问题来简化实现原问题的求解。递归策略只需少量的代码就可以描述出递归过程所需要的多次重复计算,大大地减少了程序的代码量,提高了算法的可读性。但在递归调用的过程中,也隐含着某些代价。如图 3-10 所示,系统要为每一层调用的返回点、局部变量、传入实参等开辟递归工作栈来进行数据存储,这不仅增加了空间开销,同时还需要花费大量额外的时间以创建、维护和销毁各工作栈;除此之外,在通常情况下,递归调用过程中包含很多重复计算。下面以 $n=5$ 为例,列出算法 3-12 的递归调用执行过程,如图 3-14 所示。

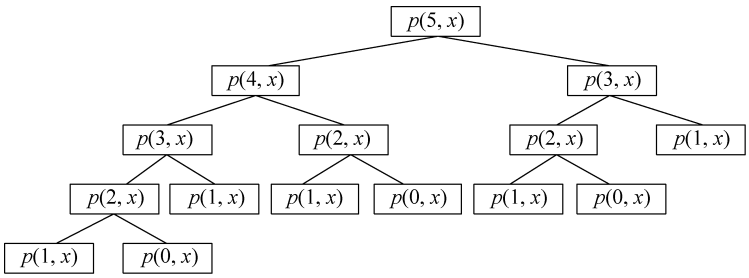


图 3-14　$p(5,x)$的递归执行过程

由图 3-14 可知,在递归调用的过程中,$p(3,x)$被计算了 2 次,$p(2,x)$被计算了 3 次,$p(1,x)$被计算了 5 次,$p(0,x)$被计算了 3 次。所以递归算法效率并不高。

例 3-4 中的汉诺塔问题的递归算法 3-11 和例 3-5 中的递归算法 3-12,其时间复杂度都为 $O(2^n)$,当 n 稍大时,这类算法是行不通的。

3. 利用栈消除递归

由上可见,递归算法并非十全十美,它虽然代码量少、结构简洁、可读性好,但在递归调用过程中需要系统提供隐式栈来实现,占用了大量额外的内存空间,运行效率较低。所以,在对运行速度有更高追求、存储空间需精打细算的场合,应尽可能地避免递归。这就要求能将递归算法改写成等价的非递归算法。既然递归本身就是操作系统隐式地维护一个调用栈来实现的,那么,就可以通过显式地利用栈来模拟调用栈的工作过程。

【例 3-6】 递归问题的非递归求解:设计算法,利用一个栈实现以下递归函数的非递归计算。

$$p_n(x)=\begin{cases}1 & n=0\\ 2x, & n=1\\ 2xp_{n-1}(x)-2(n-1)p_{n-2}(x), & n>1\end{cases}$$

【分析】 设置一个栈,栈中的每个元素含有两个域成员变量,分别用于保存 n 和对应的 $p_n(x)$值,并使栈中相邻元素的 $p_n(x)$值具有上述关系。解决办法是:先是将 2 到 n 的

数逆序压栈，再边出栈边计算 $p_n(x)$，等栈空后该值就计算出来了。其中栈的数据元素类型可说明如下：

```
typedef struct{
     int n;                          //保存 n 的值
     double val;                     //保存 pn(x)的值
}SElemType;                          //栈的数据元素类型
```

【算法 3-13】 例 3-6 中设计的相关算法。

```
double p(int n,double x)
//利用栈实现的非递归算法
{  SElemType e;
   SqStack s;
   InitStack(s);
   double f1 = 1;                    //n = 0 时的初值
   double f2 = 2 * x;                //n = 1 时的初值
   if (n == 0)
       return 1;
   for (int i = n; i >= 2; i -- )
   {  e.n = i;
      e.val = 0;
      Push(s,e);                     // 进栈
   }
   while(!StackEmpty(s))
   {  Pop(s,e);                      //出栈并获取到出栈的栈顶元素
      e.val = 2 * x * f2 - 2 * (e.n - 1) * f1;
      f1 = f2;
      f2 = e.val;
   }
   return f2;
} //算法 3 - 13 结束
```

说明：此算法的时间复杂度为 $O(n)$。对于递归函数的求值问题都可依照此题的解题思路进行求解，本章最后 3 道习题都要求用递归和非递归方法求解，请读者注意总结归纳，学以致用。

3.2 队　　列

3.2.1 队列的概念

队列是另一种特殊的线性表，它的特殊性体现在队列只允许在表尾插入数据元素，在表头删除数据元素，所以队列也是一种操作受限的特殊的线性表，它具有**先进先出**(first-in first-out，FIFO)或**后进后出**(last-in last-out，LILO)的特性。

允许进行插入的一端被称为**队尾**(rear)，允许进行删除的一端被称为**队首**(队头)(front)。假设队列中的数据元素序列为 $\{a_1, a_2, a_3, \cdots, a_n\}$，则其中 a_1 为队首

(队头)元素,a_n 为队尾元素,n 为队列中数据元素的个数,当 $n=0$ 时,称为空队列。队列的插入操作通常称为**入队**操作,而删除操作通常称为**出队**操作,如图 3-15 所示。

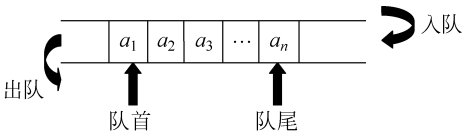


图 3-15　队列及其操作的示意图

队列在现实生活中处处可见,例如:人在食堂排队买饭、人在车站排队上车、汽车排队进站等。这些排队都满足一个规则,就是按先后次序,后来的只能到队伍的最后排队,先来的先处理再离开,不能插队。在需要公平且经济地对各种自然或社会资源进行分配的场合,无论是调试银行和医院的服务商品,还是轮耕的田地和轮伐的森林,队列都可大显身手。

队列在计算机领域中也随处可见,例如,计算机及其网络自身内部的各种计算机资源,无论是多进程共享的 CPU 资源,还是多用户共享的打印机资源,也都需要借助队列实现其合理和优化的分配。

3.2.2　队列的抽象数据类型描述

队列也是由 $n(n\geqslant 0)$ 个具有相同类型的数据元素所构成的有限序列。队列的基本操作与栈类似。队列的抽象数据类型描述如下:

ADT Queue {

数据对象:D = { ai | ai∈QElemType, 1≤i≤n,n≥0,QElemType 是约定的队列中数据元素类型,使用时用户需根据具体情况进行自定义}

数据关系:R = {< ai,ai + 1 > | ai 、ai + 1∈D,1≤i≤n - 1, 并约定其中 a1 为队首元素, an 为队尾元素}

基本操作:

InitQueue(&Q),**初始化操作:**创建一个空队列 Q。

DestroyQueue (&Q),**销毁操作:**释放一个已经存在的队列 Q 的存储空间。

ClearQueue (&Q),**清空操作:**将一个已经存在的队列 Q 置为空队列。

QueueEmpty (Q),**判空操作:**判断队列 Q 是否为空,若为空,则函数返回 TRUE;否则,函数返回 FALSE。

QueueLength(Q),**求栈的长度操作:**求队列 Q 中数据元素的个数并返回其值。

GetHead(Q, &e),**取队首元素操作:**读取队首元素,并用 e 返回其值。

EnQueue(&Q, e),**入队操作:**将数据元素 e 插入队列 Q 中,并使其成为新的队尾元素。

DeQueue(&Q,&e),**出队操作:**删除队首元素并用 e 返回其值。

DisplayStack(Q),**输出操作:**输出队列 Q 中各个数据元素的值。

}ADT Queue

同栈一样,队列也可用顺序和链式两种存储结构表示。顺序存储的队列称为**顺序队列**,链式存储的队列称为**链队列**。

3.2.3　顺序队列及其基本操作的实现

1. 顺序队列的存储结构描述

与顺序栈类似,在队列的顺序存储结构中,需要分配一块地址连续的存储区域来依次存放队列中从队首到队尾的所有元素。这样也可以使用一维数组来表示,假设数组的首地址

为 base，最大容量为 MAXQSIZE。由于队列的入队操作只能在当前队列的队尾进行，而出队操作只能在当前队列的队首进行，所以为了操作方便需加上变量 front 和 rear 来分别指示队首和队尾元素在数组中的位置。它们的位置可以说明为指针类型，也可以说明为整型。当说明为指针类型时，则意味着它们指示队首元素和队尾元素在数组中的存储单元地址；当说明为整型时，则意味着它们分别指示队首元素和队尾元素在数组中的下标。现假定 front 和 rear 为整型变量且在非空队列中，front 指示队首元素的存储位置，rear 指示队尾元素的**下一个**存储位置，则在初始化空队列时，front 和 rear 的初始值都为 0。顺序队列的动态存储结构描述如下：

```
#define MAXQSIZE 100                    //队列可能的最大长度
typedef struct {
    QElemType * base;                   // 队列存储空间基地址
    int front ;                         //指示队首元素存储单元的位置(队首指针)
    int rear;                           //指示队尾元素的下一存储单元的位置(队尾指针)
}SqQueue;
```

其中，SqQueue 为顺序队列的类型名。根据上述描述，对于非空队列 $Q=(39,17,30,65,35,54)$ 初始状态的顺序存储结构可用图 3-16 表示。

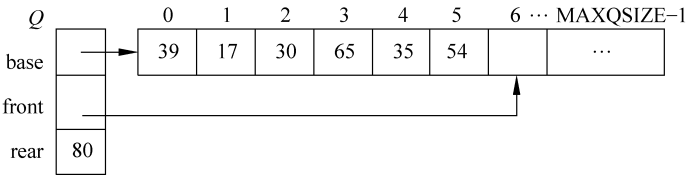


图 3-16　顺序队列 Q 初始状态的存储结构示意图

特别说明：对于已知如图 3-16 所示的顺序队列 Q，其存储空间基地址的直接访问形式为 Q. base；队首指针的直接访问形式为 Q. front，它的值即为队首元素在数组中存储单元的下标；队尾指针的直接访问形式为 Q. rear，它的值即为队尾元素在数组中下一存储单元的下标；队首元素的直接访问形式为 Q. base[Q. front]；队尾元素的直接访问形式为 Q. base[Q. rear-1]。

图 3-17 是一个在容量 MAXQSIZE＝6 的顺序队列 Q 上进行入队、出队操作后的动态示意图。

图 3-17 中描述了一个从空队列开始，先后经过 A、B、C 入队列；A、B 出队列；E、F、G 入队操作后，队列的顺序存储结构状态。

初始化队列时，令 Q. front＝Q. rear＝0；入队时，直接将新的数据元素存入 Q. rear 所指的存储单元中，然后将 Q. rear 值加 1；出队时，直接取出 Q. front 所指的存储单元中数据元素的值，然后将 Q. front 值加 1。

再仔细观察图 3-17(d)，若此时还需将数据元素 H 入队，H 应该存放于 Q. rear＝6 指示的位置处，顺序队列则会因数组下标越界而引起“溢出”，但此时顺序队列的首部还空出了两个数据元素的存储空间。因此，这时的“溢出”并不是由于数组空间不够而产生的。这种因顺序队列的多次入队和出队操作后出现有存储空间，但不能进行入队操作的溢出现象称为**“假溢出”**。

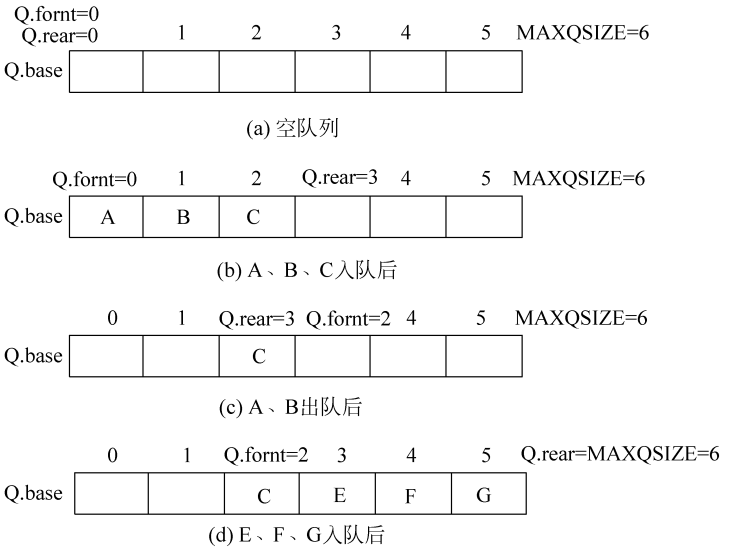


图 3-17　顺序队列的入队、出队操作的动态示意图

要解决“假溢出”问题，最好的办法就是把顺序队列所使用的存储空间看成是一个逻辑上首尾相连的循环队列。当 Q. rear 或 Q. front 到达 MAXQSIZE－1 后，再加 1 就自动到 0。这种转换可利用 C 语言中对整型数据求模(或取余)运算来实现，即令 Q. rear＝(Q. rear＋1)％MAXQSIZE。显然，当 Q. rear＝MAXQSIZE－1 时，Q. rear 加 1 再与 MAXQSIZE 求模运算后，Q. rear 的值为 0。这样，就可有效避免出现顺序队列数组的头部有空的存储空间，而队尾却因数组下标越界而引起的假溢出现象。

2. 循环顺序队列基本操作的实现

1) 循环顺序队列的初始化操作

循环顺序队列的初始化操作 InitQueue(&Q)的基本要求是创建一个空的循环顺序队列。仍然假设 MAXQSIZE＝6，循环顺序队列的初始化状态如图 3-18(a)所示，此时有 Q. front == Q. rear == 0 为真。要完成此操作的主要步骤可归纳如下：

(1) 分配预定义大小的数组空间。

用于存放队列中各数据元素的数组空间通过函数 malloc 进行分配，空间的大小仍遵守“足够应用”的原则。在此是通过符号常量 MQXQSIZE 表示，其值根据队列可能的最大长度值进行预先设定。

(2) 如果空间分配成功，则置队首和队尾指针值为 0。

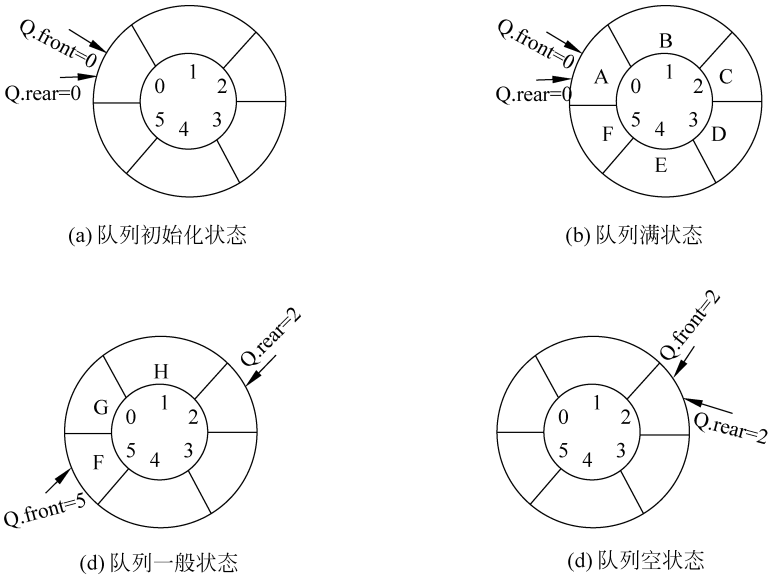
说明：如果假定队首指针指示队首元素的位置，而队尾指针指示队尾元素的位置，则队列初始化时应置队首指针值为 0，而队尾指针值为－1。

【算法 3-14】 循环顺序队列的初始化操作算法。

```
Status InitQueue( SqQueue &Q)
 // 创建一个空的循环顺序队列 Q
{  Q.base = (QElemType * ) malloc (MAXQSIZE * sizeof (QElemType));
   if (!Q.base) exit(OVERFLOW);          //如果空间分配失败
```

```
    Q.front = Q.rear = 0;
    return OK;
} //算法 3-14 结束
```

如图 3-18(a)所示的为空队列，当 A、B、C、D、E、F 分别入队后，循环顺序队列为满，其状态如图 3-18(b)所示，此时有条件 Q. front == Q. rear 为真；当 A、B、C、D、E 出队，而 G，H 又入队后，循环顺序队列的状态如图 3-18(c)所示，此时 Q. front＝5，Q. rear＝2；再将 F、G、H 出队后，循环顺序队列为空，如图 3-18(d)所示，此时也有条件 Q. front == Q. rear 为真。为此，在循环顺序队列中就引发一个新的问题：无法区分队空和队满的状态，这是因为循环顺序队列的队空和队满时都具备条件 Q. front == Q. rear 为真。



(a) 队列初始化状态　(b) 队列满状态

(d) 队列一般状态　(d) 队列空状态

图 3-18　循环顺序队列 Q 的 4 种状态图

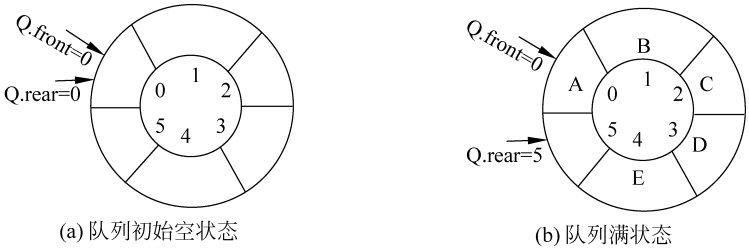
解决循环顺序队列的队空和队满的判断问题通常可采用如下 3 种方法。

(1) 少用一个存储单元。

当队列顺序存储空间的容量为 MAXQSIZE 时，只允许最多存放 MAXQSIZE－1 个数据元素。如图 3-19 所示为这种情况下的队空和队满的两种状态。此时：

队空的判断条件为 Q. front == Q. rear。

队满的判断条件为 Q. front == (Q. rear＋1)％MAXQSIZE。



(a) 队列初始空状态　(b) 队列满状态

图 3-19　少用一个存储单元时循环顺序队列 Q 的两种状态图

(2) 设置一个标志变量。

如果希望循环队列中的空间都得到利用,可以在程序设计过程中引进一个标志变量flag,其初始值置为0。每当入队操作成功后就置flag=1;每当出队操作成功后就置flag=0。此时:

队空的判断条件为Q.front == Q. rear && flag == 0。

队满的判断条件为Q.front == Q.rear && flag == 1。

(3) 设置一个计数器。

如果希望循环队列中的空间都得到利用,也可以在程序设计过程中引进一个计数变量num,其初始值置为0,每当入队操作成功后就将计数变量num的值加1;每当出队操作成功后就将计数变量num的值减1.此时:

队空的判断条件为Q. front == Q.rear&& num == 0或num == 0;

队满的判断条件为Q.front == Q.rear && num! = 0或num == MAXQSIZ。

下面假定在采用第二种方法区分队空和队满判断条件的前提下,给出循环顺序队列的入队、出队和求长度操作的实现方法及其算法描述。其中,标志变量flag说明为全局整型变量,并置初值0。

2) 循环顺序队列的入队操作

入队操作EnQueue(&Q,e)的基本要求是将新的数据元素e插入循环顺序队列Q的尾部,使其成为新的队尾元素。实现此操作的主要步骤归纳如下。

(1) 判断循环顺序队列是否为满,若满,则报告队列状态后结束算法,否则转(2)。

(2) 先将新的数据元素e存入Q.rear所指示的数组存储单元中,使其成为新的队尾元素,再将Q.rear值循环加1,使Q.rear始终指向队尾元素的下一个存储位置。实现这一步骤的操作语句为。

```
Q.base[Q.rear] = e ;                          //将e存入Q.rear所指的数组存储单元中
Q.rear = (Q.rear + 1) % MAXQSIZE;             //Q.rear值循环加1
```

【算法3-15】 循环顺序队列的入队操作算法。

```
Status EnQueue(SqQueue &Q, QElemType e)          //设置一个标志变量的方法
  // 在循环顺序队列Q中插入新的元素e, 使其成为新的队尾元素
{   if (Q.front == Q.rear&&flag == 1)            //当前队满
    {   printf("The Queue is OVERFLOW!\n");
        return ERROR;
    }
    Q.base[Q.rear] = e;                          //e入队
    Q.rear = (Q.rear + 1) % MAXQSIZE;            //队尾指针循环下移一位
    flag = 1;                                    //标志变量置为入队状态
    return OK;
} //算法3-15结束
```

3) 循环顺序队列的出队操作

出队操作DeQueue(&Q,&e)的基本要求是将队首元素从循环顺序队列Q中移去,并

用 e 返回被移去的队首元素的值。实现此操作的主要步骤归纳如下。

(1) 判断循环顺序队列是否为空,若为空,则报告队列状态后结束操作,否则转(2)。

(2) 先读取 Q.front 所指示的队首元素的值并用 e 保存,再将 Q.fornt 值循环加 1,使其指向新的队首元素。实现这一步骤的操作语句为:

```
e = Q.base[Q.front];                          //取出队首元素并用 e 保存
Q.front = (Q.front + 1) % MAXQSIZE;           //队首指针下移一位
```

【算法 3-16】 循环顺序队列的出队操作算法。

```
Status DeQueue (SqQueue &Q, QElemType &e)        //设置一个标志变量的方法
  // 删除循环顺序队列 Q 中的队首元素,并用 e 返回其值
{
     if (Q.front == Q.rear&&flag == 0)           //当前队空
     {  printf("The Queue is NULL!\n");
        return ERROR;
     }
    e = Q.base[Q.front];                         //用 e 返 队首元素
    Q.front = (Q.front + 1) % MAXQSIZE;          //队首指针循环下移一位
    flag = 0;                                    //标志变量置为出队状态
    return OK;
} //算法 3 - 16 结束
```

思考:请分别写出采用第一种和第三种方法区分队空和队满条件时,循环顺序队列的入队和出队操作算法。

4) 循环顺序队列求长度操作

由图 3-18(b)和图 3-18(c)可知,当队列为满时,队列的长度就是 MAXQSIZE;而在一般情况下,队列的长度可以根据队列的队尾指针和队首指针值计算而得。

【算法 3-17】 循环顺序队列求长度操作算法。

```
int QueueLength(SqQueue Q)                       //设置一个标志变量的方法
//返回循环顺序队列中数据元素的个数
{  if (Q.front == Q.rear&&flag == 1)
         return MAXQSIZE;
   else
         return (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
}//算法 3 - 17 结束
```

补充说明:若采用第一种方法区分队空和队满条件时,无论何时,队列的长度都可采用(Q.rear－Q.front＋MAXQSIZE)％MAXQSIZE 计算而得;若采用第三种方法区分队空和队满条件时,其中 num 的值就随时记录着队列的长度。

从上述算法描述可知:循环顺序队列的初始化、入队、出队及求长度操作算法的时间复杂度都为 $O(1)$。

3.2.4 链队列及其基本操作的实现

1. 链队列的存储结构描述

队列的链式存储结构用带头结点的单链表来实现。为了便于实现入队和出队操作,需

要引进两个指针 front 和 rear 来分别指向队首元素和队尾元素的结点。现将链队列的存储结构描述如下：

```
typedef struct QNode
{   QElemType data;
    struct QNode * next;
}QNode, * QueuePtr;                          //链队列的结点类型及指向结点的指针类型
typedef struct
{   QueuePtr front;                          //队首指针
    QueuePtr rear;                           //队尾指针
} LinkQueue;                                 //链队列类型
```

图 3-20 为空队列和非空队列 $a_1,a_2,\cdots,a_n$ 的链式存储结构示意图。

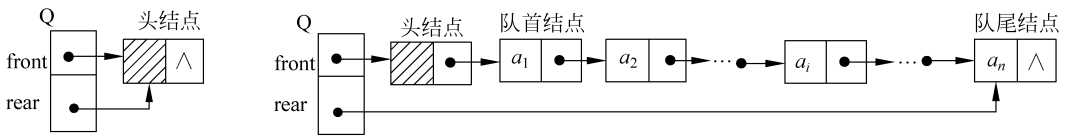


图 3-20　链队列的存储结构示意图

2. 链队列基本操作的实现

结合链队列的存储结构示意图，并借鉴第 2 章中，在带头结点的单链表上实现基本操作的主要步骤和算法描述，再来思考链队列相应操作的算法设计，还是比较容易的。在此只要记住队列的入队操作是在链表的表尾进行，而出队操作则是在链表的表头进行。对于一个空队列来说，其链中只含一个头结点，并且队首指针和队尾指针都指向头结点，如图 3-20(a)所示。因此，链队列的判空条件是 Q.rear == Q.front。下面也只讨论链队列的初始化、入队、出队的实现方法及其算法描述。

1) 链队列初始化操作

由图 3-17(a)可知，要创建一个空的链队列，只要为头结点分配一个结点大小的空间，并使队首指针和队尾指针都指向该结点即可。

【算法 3-18】 链队列初始化操作算法。

```
Status InitQueue(LinkQueue &Q)
//创建一个头结点的空链队列 Q
{   Q.front = (QueuePtr)malloc(sizeof(QNode));  //为链队列的头结点分配空间
    if (!Q.front)
        return OVERFLOW;                        //分配空间失败
    Q.front -> next = NULL;
    Q.rear = Q.front;                           //使队尾指针也指向头结点
    return OK;
}//算法 3 - 18 结束
```

2) 链队列的入队操作

链队列入队操作 EnQueue(&Q,e)的基本要求是将数据元素 e 所对应的结点插入链队列的尾部，使其成为新的队尾结点。此操作的基本思想与带头结点的单链表的插入操作类似，不相同之处为链队列的插入位置限

制在表尾进行。实现此操作的主要步骤归纳如下。

(1) 创建数据域值为 e 新结点。

(2) 修改链,使新结点链接到队列的尾部并使其成为新的队尾结点。

图 3-21 显示了链队列入队操作后状态的变化情况。

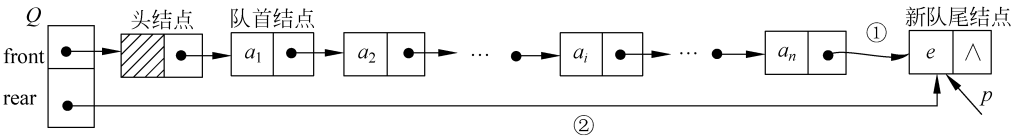


图 3-21 链队列的入队操作后状态变化示意图

【算法 3-19】 链队列的入队操作算法。

```
Status EnQueue(LinkQueue &Q, QElemType e)
  // 在链队列的队尾插入新的元素 e,使其成为新的队尾元素
{  QueuePtr p = (QueuePtr)malloc(sizeof(QNode));   //为新结点分配空间
   if (!p)                                         //空间分配失败
        return ERROR;
   p->data = e;                                    //e 存入新结点的数据域
   p->next = Q.rear->next;                         //修改链(1),让新结点插入到链队列的尾部
   Q.rear->next = p;
   Q.rear = p;                                     //修改链(2),让队尾指针指向新的队尾结点
   return OK;
}// 算法 3-19 结束
```

3) 链队列的出队操作

链队列出队操作 DeQueue(&Q,e)的基本要求是将非空队列中的队首结点从链队列中移去,并通过 e 返回被移去结点的数据元素值。此操作的基本思想也与带头结点的单链表的删除操作类似,不相同之处为链队列出队的结点一定是链中第一个数据元素结点。实现此操作的主要步骤归纳如下。

(1) 判断链队列是否为空,若为空,则报告队列状态后结束算法,否则转(2)。

(2) 用 p 指针指向待删除的队首结点,并用 e 保存其数据元素值。

(3) 修改链指针,使队首结点从链队列中脱离出来。

(4) 若待删除结点既是队首结点,又是队尾结点,则需修改队尾指针,使队列为空。

(5) 释放被删结点空间。

【算法 3-20】 链队列的出队操作算法。

```
1  Status DeQueue(LinkQueue &Q, QElemType &e)
2   // 删除链队列中的队首数据元素,并用 e 返回其值
3  {  if(Q.front == Q.rear)                         //队空
4     {  printf("The Queue is NULL!\n");
5        return ERROR;
6     }
7     QueuePtr p = Q.front->next;                   //p 指针指向待删除的队首结点
8     e = p->data;                                  //用 e 保存队首结点的数据元素值
9     Q.front->next = p->next;                      //修改链指针使队首结点从链中脱离
```

```
10      if (p == Q.rear)                          //如果被删的结点是队尾结点
11          Q.rear = Q.front;
12      free(p);                                  //释放待删结点空间
13      return OK;
14  }// 算法 3-20 结束
```


注意：在链队列中执行出队操作时，如果当前链队列只含一个数据元素结点，则待删除的结点既是队首结点，也是队尾结点。当此结点被删除后，则变成了一个空的链队列，所以此种情况下需修改队尾指针，使其与队首指针同指向队列的头结点，这就是算法3-20中第10～11行所描述的真正含义，请读者加以重视。

从上述算法描述可知，链队列的初始化、入队和出队操作算法的时间复杂度也都为$O(1)$。

3.2.5 其他队列

1. 优先级队列

在很多现实问题中，利用简单队列往往是不能完全解决问题的，队列的“先进先出”还会受到优先级的限制。例如：在车站排队买车票时，残疾人可以优先买，即当售票员工作时，尽管还有人排在队伍的前面，残疾人仍可以优先得到服务。又如：在计算机系统中，为了保证系统的正常运行和系统资源的有效利用，在某种情况下即使程序p_1比程序p_2早进入等待队列，但p_2仍可在p_1之前运行。这些情况就需要使用优先级队列。

优先级队列是一种带有优先级的队列。优先级可以反映使用频率、生日、工资、职位等不同的优先规则，也可以是程序队列中的预估运行时间。通常用数字来代表优先级，数字值越小表示优先级越高。

优先级队列可以用两种链表来表示。一种是优先级队列中的元素按优先级从高到低有序排列，也就是说优先级最高的总是排在队首，因此，在优先级队列中也是从队首删除元素，但元素的插入不一定是在队尾进行，而是顺序插入队列的合适位置，以确保队列的优先级顺序；另一种是优先队列中的元素按顺序进入队尾，而删除元素则要从优先队列中选择优先级最高的元素出队。这两种情况下的总操作时间复杂度都是$O(n)$，因为在按优先级有序排列的队列中，可以立即从队首取出一个元素出队，但入队操作需要的时间复杂度是$O(n)$，而在未按优先级排序的队列中，可以立即插入一个元素到队尾，但出队操作需要在队列中先选译优先级最高的元素，其时间复杂度是$O(n)$。

例如，计算机操作系统用一个优先队列来实现进程的调度管理。在一系列等待执行的进程中，每一个进程可以用一个数值来表示它的优先级，优先级越高，这个值越小。优先级高的进程应该最先获得处理器。假设操作系统中每个进程的数据由进程号和优先级两部分组成。进程号是每个不同进程的唯一标识，优先级通常是一个0～40的数值，规定0为优先级最高，40为优先级最低。如下，为一组模拟数据：

进程号	优先级
1	20
2	40
3	0
4	10
5	40

则图 3-22 表示当前按优先级从高到低所形成的优先队列存储结构示意图。

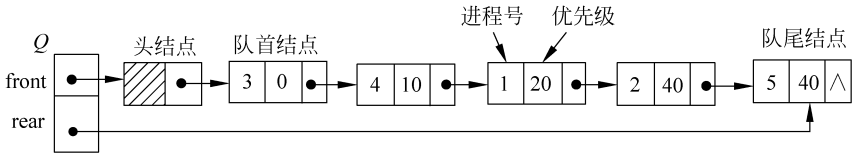


图 3-22　优先队列存储结构示意图

思考：请读者按如图 3-22 所示的存储结构示意图，写出其存储结构的类型描述，并设计此优先队列的入队操作算法（其出队操作的算法与一般队列的相同）。

2. 双端队列

双端队列也是一种操作受限的线性结构。

（1）双端队列：指允许两端都可以进行入队和出队操作的队列，其两端分别被称为前端和后端，如图 3-23 所示。在双端队列上实现入队操作时，前端入队的元素排列在从后端入队的元素前面，而从后端入队的元素排列在从前端入队的元素后面；在双端队列上实现出队操作时，无论从前端还是从后端出队，先取出的元素总是排列在后取出的元素的前面。

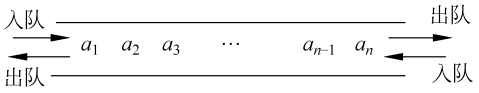


图 3-23　双端队列示意图

（2）输出受限的双端队列：指允许在一端进行出队操作，但允许两端进行入队操作的双端队列，如图 3-24 和图 3-25 所示。

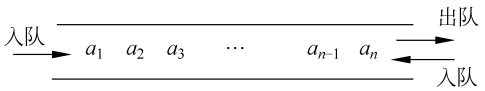


图 3-24　输出受限的双端队列示意图（1）

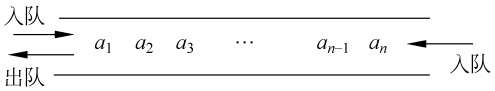


图 3-25　输出受限的双端队列示意图（2）

（3）输入受限的双端队列：指允许在一端进行入队操作，但允许在两端进行出队操作的双端队列，如图 3-26 和图 3-27 所示。如果限定双端队列从某端插入的元素只能从该端删除，则该双端队列就蜕变为两个栈底相邻接的栈了。

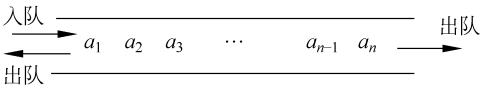


图 3-26　输入受限的双端队列示意图（1）

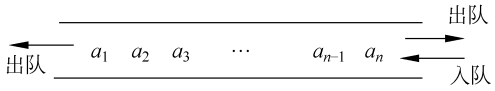


图 3-27　输入受限的双端队列示意图（2）

若以 1，2，3，4 作为双端队列的输入序列，满足以下条件的输出序列有：

（1）能由输入受限的双端队列得到，但不能由输出受限的双端队列得到的输出序列为 4，1，3，2。

（2）能由输出受限的双端队列得到，但不能由输入受限的双端队列得到的输出序列为 4，2，1，3。

（3）既不能由输入受限的双端队列得到，也不能由输出受限的双端队列得到的输出序列为 4，2，3，1。

3.2.6 队列的应用

由于队列是一种具有先进先出特性的线性表,所以在现实世界中,当求解具有先进先出特性的问题时可以使用队列。例如,操作系统中各种数据缓冲区的先进先出管理;应用系统中各种任务请求的排队管理;软件设计中对树的层次遍历和对图的广度遍历过程等,都需使用队列。队列的应用非常广泛,本节通过3个实例来说明队列在解决实际问题中的应用。

【例3-7】 设计算法,借助队列计算并打印杨辉三角形的前n行内容,如图3-28所示为8行的杨辉三角形。

【分析】 从图中观察可知,在杨辉三角形中,每行第一个和最后一个数都是1,从第3行开始的其余数等于上一行对应位置的左、右两个数之和。例如,第3行的第2个数2是第2行的第1个数1和第2个数1的和;第6行的第3个数10是第5行的第2个数4和第5行的第3个数6的和。也就是说,除每行的第一个数和最后一个数之外,其他数都可由上一行对应位置的两个数求出。为此,可以借助一个队列来存放上一行的数,每次由队列前两个数来求出下一行的一个数,运算完成后需将参与运算的当前队首数删除(出队),而运算结果再加入队列的尾部(入队)。

注意:

(1) 为了能正确求出从第2行起,每一行的第一个数1,需要在上一行的第一个数前面加一个0。图3-29展示了由第7行求得第8行对应数的过程。

(2) 每行的最后一个1不能通过上一行计算而得,所以只能直接输出这个1。

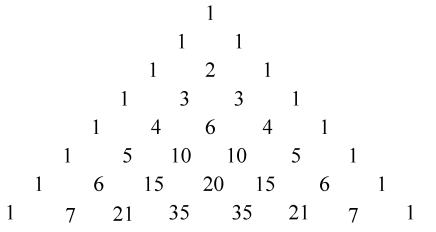


图3-28 杨辉三角形

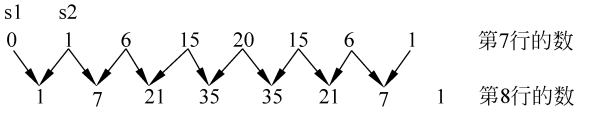


图3-29 杨辉三角形第8行的求解过程示意图

【算法3-21】 例3-7中设计的相关算法。

```
void OutYangHuiSanJiao(int n)
//打印杨辉三角中的前n行数据
{   int s1,s2,s;
    SqQueue Q;
    InitQueue(Q);
    for(int i=1;i<=4*(n-1);i++)          //输出第1行数1左边的空格
        printf(" ");
    printf("%8d\n",1);                   //输出第1行上的数1
    EnQueue(Q,1);                        //输出的1入队
    for(int i=2;i<=n;i++)
    {   s1=0;                            //存放前一个出队的数
        for(int k=1;k<=n*4-4*i;k++)      //输出第i行第一个数左边的空格
```

```
        printf(" ");
    for(int j = 1;j <= i - 1;j++)              //计算并输出第 i 行的数据
    {     DeQueue(Q,s2);                       //队首元素出队
       s = s1 + s2;                            //计算出第 i 行的一个数
       printf(" %8d",s1 + s2);                 //输出第 i 行的一个数
       EnQueue(Q,s);                           //输出的数入队
       s1 = s2;
    }
       printf(" %8d\n",1);                     //输出第 i 行的最后一个 1
       EnQueue(Q,1);                           //第 i 行最后一个 1 入队
  }
} //算法 3-21 结束
```

【例 3-8】 舞伴配对问题。

【问题描述】 假设在周末舞会上,男士们和女士们进入舞厅时,各自排成一队。跳舞开始时,依次从男队和女队的队头上各出一人配成舞伴。若两队初始人数不相同,则较长的那一队中未配对者等待下一轮舞曲。要求写一算法模拟上述舞伴配对问题,并编写其测试程序,且规定:

(1) 输入数据:进入舞厅的男士和女士的姓名和性别。

(2) 输出数据:如果是配成对的,则输出两个舞伴的人的姓名;如果是未配成对的,则输出等待配对的人数和下一轮舞曲开始时第一个可获得舞伴的人的姓名。

【问题分析】 该问题明显具有"先进先出"特性,所以可借助队列来模拟实现。在此不妨采用两个链队列来分别表示进入舞厅的男士和女士排成的队,初始时为空。再假设参加舞会的男士和女士总数为 num,其姓名和性别信息分别存放在一个结构体数组中作为算法的输入,然后依次扫描该数组的各元素,并根据性别来决定是进入男队还是女队。当这两个队列构造完成之后,再依次将两队列当前的队首元素出队来模拟舞伴配对,直至某队列变空为止。此时,若某队列仍有等待配对者,则输出此队列中等待配对的人数及排在队首的等待者的名字,他(或她)就是下一轮舞曲开始时的第一个可获得舞伴的人。主要步骤可具体归纳如下:

(1) 利用 InitQueue()操作创建两个空队列 Boys 和 Girls,分别用于表示进入舞厅的男士和女士排成的队。

(2) 从结构数组中依次读取每个元素的信息,若其性别为"男",则利用 EnQueue()操作将其加入 Boys 队列,否则加入 Girls 队列,直到 num 个人都入队为止。

(3) 利用 DeQueue()操作依次将 Boys 和 Girls 队列中的队首元素出队,并输出其姓名以示配对,直到某个队列为空为止。

(4) 利用 QueueLength()操作求出非空队列的长度并输出,这个长度值就是队列中等待配对的人数。

(5) 利用 GetHead()操作取出非空队列中的队首元素,并输出其姓名,他(她)就是下一轮舞曲开始时的第一个可获得舞伴的人。

【算法 3-22】 例 3-8 中设计的相关算法。

```
typedef struct{
    char name[10];                    //存放姓名
```

```
    char sex;                          //存放性别,'F'表示女性,'M'表示男性
}Person;
typedef Person QElemType;              //将队列中元素的数据类型定义为Person
void DancePartner(QElemType dancer[], int num)
//结构数组dancer中存放参加舞会的人员信息,num是参加舞会的人数
{
    int i;
    QElemType p;
    LinkQueue Boys,Girls;              //定义两个链队列,分别用于存放进入舞厅的男士和女士
    InitQueue(Boys);                   //初始化一个用来记录进入舞厅的男士队列
    InitQueue(Girls);                  //初始化一个用来记录进入舞厅的女士队列
    for(i=0;i<num;i++){                //将参加舞会者按其性别依次入男、女队列
        p=dancer[i];
        if(p.sex=='F')
            EnQueue(Girls,p);          //入女士队列
        else
            EnQueue(Boys,p);           //入男士队列
    }
    printf("配对成功的舞伴分别是:\n");
    while(!QueueEmpty(Girls)&&!QueueEmpty(Boys)){                    //依次输出男女舞伴名
        DeQueue(Girls,p);              //女士出队
        printf("%s",p.name);           //打印出队女士名
        DeQueue(Boys,p);               //男士出队
        printf("%s\n",p.name);         //打印出队男士名
    }
    if(!QueueEmpty(Girls)){            //输出女士剩余人数及队首女士的名字
        printf("还有%d名女士等待下一轮舞曲.\n",QueueLength(Girls));
        GetHead(Girls,p);              //取队首
        printf("%s将在下一轮中最先得到舞伴.\n",p.name);
    }
    else if(!QueueEmpty(Boys)){        //输出男队剩余人数及队首者男士的名字
        printf("还有%d名男士等待下一轮舞曲.\n",QueueLength(Boys));
        GetHead(Boys,p);
        printf("%s将在下一轮中最先得到舞伴.\n",p.name);
    }
} // 算法3-22结束
int main()
//测试算法3-22的主函数
{ int i,n;
    QElemType dancer[30];
    printf("请输入舞池中排队的人数:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("姓名:");
        scanf("%s",dancer[i].name);
        getchar();
        printf("性别:");
        scanf("%c",&dancer[i].sex);
    }
    DancePartner(dancer,n);
```

```
    return 0;
}
```

【运行结果】 此测试程序的运行结果如图 3-30 所示。

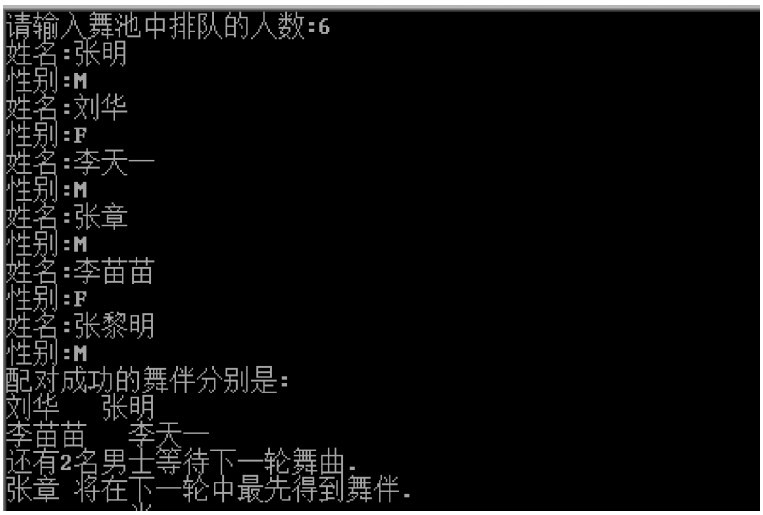


图 3-30 例 3-8 测试程序运行结果

【例 3-9】 设计算法：实现求解素数环问题。

【问题描述】 将 1～n 的 n 个自然数排列成环形，使得每相邻两数之和为素数，从而构成一个素数环。例如：当 $n=6$ 时，其素数环为 1,4,3,2,5,6；而当 $n=8$ 时，其素数环为 1,2,3,8,5,6,7,4。

【问题分析】 由问题描述可知，当 n 为奇数时，素数环一定不存在，所以可先排除此情况。对于偶数 n，则可借助线性表和队列的基本操作来求解此问题。本题是采用顺序表和顺序队列，其中顺序表是用来存放加入素数环中的自然数，而顺序队列是用来存放求解过程中还未加入素数环中的自然数。首先将自然数 1 加到素数环中，2～n 加入队列中。然后再依次判断 2～n 中的每个自然数能以何种顺序加入素数环。主要的步骤可归纳如下。

(1) 先创建一个空顺序表 L 和空顺序队列 Q。

(2) 将顺序表 L 和顺序队列 Q 置上初值：将 1 加入顺序表 L 中，将 2～n 的自然数全部加入 Q 队列中。

(3) 将队列 Q 的队首自然数 p 出队，并与素数环 L 中的最后一个自然数 q 相加。若两数之和是素数，则将 p 加入素数环 L 中，否则说明 p 暂时无法处理，必须再次入队等待，再重复此过程。若 p 为队列中最后一个自然数，则还需判断它与素数环的第一个自然数相加的和是否为素数。若是素数，则将 p 加入素数环，素数环构造成功，其结果以顺序表的值返回，求解结束；若不是素数，则说明不满足素数环条件，需将素数环中的最后一个自然数移至队列的尾部，再重复步骤(3)。如此重复，直到队列为空或已对队列中每一个数据元素都遍历了一次且未能加入素数环为止，此时意味着构造素数环不成功。

【算法 3-23】 例 3-19 中设计的相关算法。

```
bool IsPrime(int num)
```

```
// 判断正整数 num 是否为素数
{  if (num == 1)                        // 整数 1 返回 false
        return false;
   int n = (int)sqrt(num);              // 求平方根
   for (int i = 2; i <= n; i++)
       if (num % i == 0 )               // 模为 0, 返回 false
            return false;
   return true;
}//IsPrime

int InsertRing(SqList &L, SqQueue Q, int m, int n)
 // 判断队列 Q 中的队首自然数是否能加入素数环 L,若能则加入并返回 1,否则返回 0
 // 其中素数环 L 中有 m-1 个自然数,队列 Q 中有 n-m 个自然数
{  int count = 0;                       // 记录遍历队列中的数据元素的个数,防止在一次循环中
                                        // 重复遍历
   while (!QueueEmpty(Q)&& count <= n-m)
                                        // 队列非空,且未重复遍历队列中的自然数
   {  int p ;
      DeQueue(Q,p);                     // 队首自然数 p 出队
      int q;
      GetElem(L,ListLength(L),q);       // q 为顺序表中的最后一个数据元素
      if (m == n)                       // 为队列中的最后一个自然数
      {  if (IsPrime(p + q) && IsPrime(p + 1))
                                        // 满足素数环的条件
         {  ListInsert(L,ListLength(L) + 1, p);        // 将 p 插入顺序表表尾
            return 1;
         }
        else                            // 不满足素数环条件
            EnQueue(Q,p);               // p 重新入队
      }
      else if (IsPrime(p + q))          // 未遍历到队列的最后一个自然数,且满足相邻数和为素
                                        // 数的条件
            {  ListInsert(L,ListLength(L) + 1, p);      // 将 p 插入顺序表表尾
               if (InsertRing(L, Q, m + 1, n)!= 0)
                 // 递归调用函数,若返回值不为 0,即已成功找到素数环,返回 1
                   return 1;
               ListDelete(L,ListLength(L),p);         // 删除顺序表表尾的自然数 p
               EnQueue(Q,p);            //并将 p 重新入队
            }
           else
               EnQueue(Q,p);            // p 重新入队
        ++count;                        // 遍历次数加 1
    }
    return 0;
}//InsertRing

void MakePrimeRing(SqList &L,int n)
// 求 n 个正整数的素数环,结果以顺序表 L 返回
{  if (n % 2 != 0)                      // n 为奇数则素数环不存在
   {  printf("素数环不存在!\n");
      exit(0);
```

```
    }
    ClearList(L);                          // 清空已知的顺序表 L
    ListInsert(L,1,1);                     // 将自然数 1 加入表 L 中
    SqQueue Q;
    InitQueue(Q);                          // 创造一个空队列
    for (int i = 2; i <= n; i++)           //将自然数 2～n 分别加入队列
       EnQueue(Q,i);
    InsertRing(L, Q, 2, n);
} //算法 3-23 结束
```

3.3 栈与队列的比较

栈与队列既是两种重要的线性结构，也是两种操作受限的特殊线性表。为了让读者能够更好地掌握它们的使用特点，在此对栈和队列进行比较。

1. 栈与队列的相同点

(1) 都是线性结构，即数据元素之间具有“一对一”的逻辑关系。

(2) 插入操作都限制在表尾进行。

(3) 都可在顺序存储结构和链式存储结构上实现。

(4) 在时间代价上，插入与删除操作都需常数级时间；在空间代价上，情况也相同。

(5) 多链栈和多链队列的管理模式可以相同。在计算机系统软件中，经常会出现同时管理和使用两个以上栈或队列的情况，这种情景下，若采用顺序存储结构实现栈和队列，将会给处理带来极大不便，因而一般采用多个单链表来实现多个栈或队列。图 3-31 和图 3-32 是多链栈和多链队列的存储结构示意图。它们将多个链栈的栈顶指针或链队列的队首、队尾指针分别存放在一个一维数组中，从而很方便地实现了统一管理和使用多个栈或队列。

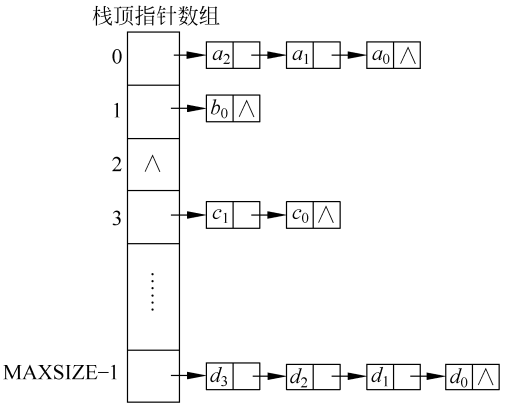


图 3-31　多链栈的存储结构示意图图

2. 栈与队列的不同点

(1) 删除数据元素操作的位置不同。栈的删除操作控制在表尾进行，而队列的删除操作控制在表头进行。

(2) 两者的应用场合不相同。具有后进先出(或先进后出)特性的应用需求，可使用栈式结构进行管理。例如，递归调用中现场信息、计算的中间结果和参数值的保存，图与树的

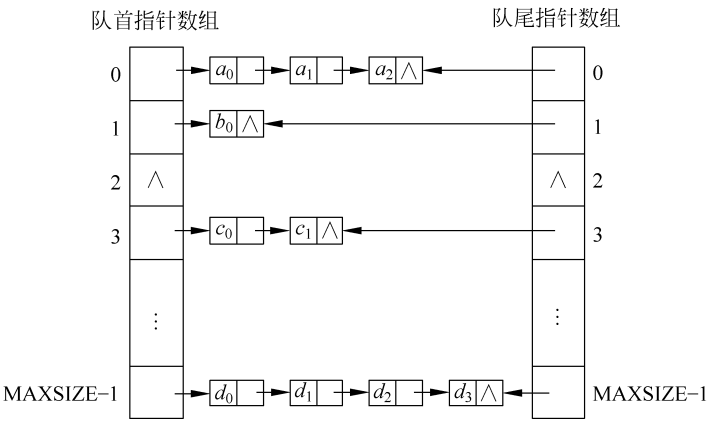


图 3-32　多链队列的存储结构示意图

深度优先搜索遍历都采用栈式存储结构加以实现。而具有先进先出(后进后出)特性的应用需求,则要使用队列结构进行管理。例如,消息缓冲器的管理,操作系统中对内存、打印机等各种资源进行管理,都使用了队列。同时可以根据不同优先级的服务请求,按优先级把服务请求组成多个不同的队列。队列也是图和树在广度搜索遍历过程中采用的数据结构。

(3) 顺序栈可实现多栈空间共享,而顺序队列则不能。实际应用中经常会出现在一个程序中需要同时使用两个栈或队列的情况。若采用顺序存储,就可以使用一个足够大的数组空间来存储多个栈,即让多个栈共享同一存储空间。图 3-33 是两个栈共享空间的示意图。其中:把数组的两端设置为两栈各自的栈底,两栈的栈顶从两端开始向中间延伸。可以充分利用顺序栈单向延伸的特性,使两个栈的空间形成互补,从而提高存储空间的利用率。然而,顺序队列就不能像顺序栈那样在同一个数组中存储两个队列,除非总有数据元素从一个队列转入另一个队列。

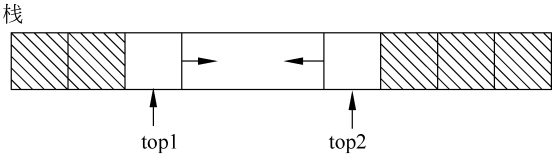


图 3-33　两个栈共享同一个数组空间

3.4　栈与队列的综合应用举例

栈和队列这两种特殊的线性表与基本线性表一样,在实际工作和生活中有着广泛的应用。本节通过一个实例再一次给出栈与队列的综合应用,使读者能更进一步地区分栈与队列的特性和它们各自的实现技巧。

【例 3-10】　停车场管理问题。

【问题描述】　假设停车场是一个可停放 n 辆车的狭长通道,并且只有一个大门可供汽车进出。在停车场内,汽车按到达的先后次序,由北向南依次排列(假设大门在最南端)。若车场内已停满 n 辆车,则后来的汽车需在门外的便道上等候,当有车开走时,便道上的第一

辆车即可驶入。当停车场内某辆车要离开时，在它之后进入的车辆必须先退出车场为它让路，待该辆车开出大门后，其他车辆再按原次序返回车场。每辆车离开停车场时，应按其停留时间的长短交费（在便道上停留的时间不收费）。

试编写程序，模拟上述管理过程。

【问题分析】 由于停车场中某辆车的离开是按在它之后进入的车辆必须先退出车场为它让路的原则下进行的，显然满足了“后进先出”的特性，所以可以用栈式结构来模拟停车场；而对于指定停车场，它的车位数是相对固定的，因而本题采用顺序栈来加以实现，并假设栈的容量为 10。又由于便道上的车是按先到先开进停车场的原则进行的，即满足“先进先出”的特性，所以可以用队列来模拟便道，而对于便道上停放车辆的数目是不固定的，因而本题采用链队列来加以实现。

为了能更好地模拟停车场管理，可以从终端读入汽车到达或离去的数据，每组数据应该包括二项：①汽车牌照号码；②是“到达”还是“离去”。程序中通过库函数 time(0)自动获取当前“到达”或“离去”的时间秒数。与每组输入信息相应的输出信息为：若是到达的车辆，则输出其在停车场中或便道上的位置；若是离去的车辆，则输出其在停车场中停留的时间和应交纳的费用。

费用的计算方法是：由离开时间减去到达时间求得停留时间的总秒数，因 time 函数返回的为 UNIX 时间戳，即从 1970 年 1 月 1 日(UTC/GMT 的午夜)开始所经过的秒数。然后再将秒数除以 60 使其转换成多少分钟，当不足整数分钟时，则以整数分钟计算。如停留时间是大于 5 分钟而小于 6 分钟，则以 6 分钟计算。最后每分钟的停车费乘以停车时间分钟数即可计算出总的停车费用。此题以每分钟 2 元的费用进行模拟计算。

【参考代码】

```
#include "Queue.cpp"          //Queue.cpp 中有队列抽象数据类型定义中的所有基本操作函数
#include "Stack.cpp"          //Stack.cpp 中有栈抽象数据类型定义中的所有基本操作函数
#include <stdio.h>
#include <time.h>
#include <string.h>
#define DEPARTURE 0;          // 标识车辆离开
#define ARRIVAL 1;            // 标识车辆到达
double fee 2.0                // 每分钟停车费用,全局变量
SqStack S;                    //顺序栈存放停车场内的车辆信息,全局变量
LinkQueue Q;                  //链队列存放便道上的车辆信息,全局变量

typedef struct
{   int state;                // 车辆状态,离开或到达
    time_t arrTime;           // 车辆达到时间
    time_t depTime;           // 车辆离开时间
    char *license;            // 车牌号
} CarInfo;                    //用于存放车辆信息的类型
typedef CarInfo QElemType;
typedef CarInfo SElemType;

// 停车场管理,参数 license 表示车牌号码,action 表示此车辆的动作,即到达或离开
void ParkingManag(char *license, char *action)
{   if (strcmp("arrive",action) == 0) {          // 车辆到达
```

```
            CarInfo info;           // 定义一个车辆信息类型变量
            info.license = license; // 修改车辆状态
            if (StackLength(S) < S.stacksize) {      // 停车场未满
                 info.arrTime = time(0);             // 当前时间初始化到达时间
                 info.state = ARRIVAL;
                 Push(S,info);
                 printf("    %s 停放在停车场第 %d 个位置!\n",info.license ,StackLength(S));
                 }
            else {                  // 停车场已满
                 EnQueue(Q,info); // 进入便道队列
                 printf("    %s 停放在便道第 %d 个位置\n" ,info.license ,QueueLength(Q));
                 }
            }
       else if (strcmp("depart",action) == 0)        // 车辆离开
            {  CarInfo info;
               int location = 0;                     // 车辆的位置
               SqStack S2;
               InitStack(S2);                        // 构造栈用于存放因车辆离开而导致的其他
                                                     // 车辆暂时退出车场
               for (int i = StackLength(S); i > 0; i -- ) {
                   Pop(S,info);
                   if (strcmp(info.license,license) == 0)
                                                     // 将离开的车辆
                   {  info.depTime = time(0);
                                                     // 用当前时间来初始化离开时间
                      info.state = DEPARTURE;
                      location = i;                  // 取得车辆位置信息
                      break;
                   }
                   else                              // 其他车辆暂时退出车场
                      Push(S2,info);
                   }//for
                  while (!StackEmpty(S2))            // 其他车辆重新进入停车场
                   {  CarInfo e;
                      Pop(S2,e);
                      Push(S,e);
                   }
                   if (location != 0)                // 停车场内存在指定车牌号码的车辆
                   {  double time = (double)(info.depTime - info.arrTime)/60;
                                                     // 计算停放时间,并把秒换成分钟
                      if ((int)time < time)          // 停车时间处理后输出
                          printf("%s 停放 %.2f 分钟,费用为:%.2f\n",info.license ,time,
                                 (int)(time + 1) * fee);
                      else
                        printf("%s 停放 %.2f 分钟,费用为:%.2f\n",info.license ,time,
                                 (int)time * fee);
                   }
                 if (!QueueEmpty(Q))                 // 便道上的第一辆车进入停车场
                 {  DeQueue(Q,info);
                    info.arrTime = time(0);          // 用当前时间来初始化到达时间
                    info.state = ARRIVAL;
```

```
                    Push(S,info);
                }
            }
}//ParkingManag

int main()
{   char license[20];
    char action[20];
    InitStack(S);                                   // 顺序栈存放停车场内的车辆信息
    InitQueue(Q);                                   // 链队列存放便道上的车辆信息
    for (int i = 1; i <= 12; i++)
    // 初始化 12 辆车,车牌号分别为 1,2,...,12,其中有 10 辆车停在停车场内,2 辆车停放在便道上
    {   printf("请输入第 %d 辆车的车牌号:",i);
        gets(license);
        ParkingManag(license, "arrive");
        }

        printf("请输入车牌号:");
        gets(license);
        printf("arrive or depart ?");
        gets(action);;
        ParkingManag(license, action);              // 调用停车场管理函数
        return 0;
}  //例 3-10 程序结束
```

此程序的运行结果如图 3-34 所示。

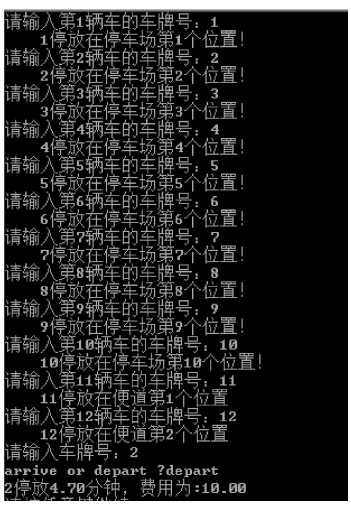


图 3-34　例 3-10 程序的运行结果

小　　结

本章介绍了两种特殊的线性表：栈和队列。栈的插入和删除操作限制在表尾一端进行的特殊线性表，无论是在栈中插入数据元素还是删除栈中数据元素，都只能固定在线性表的

表尾进行。通常将进行插入和删除操作的这一端称为“栈顶”,而另一端称为“栈底”。它是一种具有“后进先出”或“先进后出”特性的线性表。队列是插入操作只限制在表尾进行,而删除操作只限制在表头进行的特殊线性表。通常将允许进行插入操作的一端称为“队尾”,而将允许进行删除操作的另一端称为“队首”。它是一种具有“先进先出”或“后进后出”特性的线性表。

栈的基本操作主要包括判栈是否为空、出栈、入栈和取栈顶元素等。队列的基本操作主要包括判队列是否为空、出队、入队和取队首元素等。这些操作的时间复杂度都是$O(1)$。

栈与队列可用顺序和链式两种存储方式加以实现,为此有顺序栈和链栈、顺序队列和链队列之分。其中,顺序栈和顺序队列用数组实现;链栈和链队列则用单链表实现。要注意的是,链栈结点中指针的指向是从栈顶开始依次向后链接的,也就是说链栈中结点的指针不像单链表那样指向它在线性表中的后继,而是指向其在线性表中的前驱。链队列结点的链接方向与单链表相同,但为了便于实现插入和删除操作,链队列除了引进一个队首指针外,还引进了队尾指针来指向队尾元素,并将两者组合形成一个队列结构类型。

顺序栈比链栈的使用更为广泛。在顺序栈中,注意掌握入栈和出栈操作,特别是在入栈操作前要进行的判满条件,和在出栈操作前要进行的判空条件。在链栈中的操作与单链表操作类似,而且由于入栈与出栈操作都固定在链栈的栈顶位置进行,所以实现起来比单链表相应操作更为简单。

循环队列比非循环队列的使用更为广泛。在顺序存储方式下,也要注意掌握队列的入队和出队操作、队列判空与判满的条件,以及队列“假溢出”的处理方法。循环顺序队列就是为了避免“假溢出”现象而提出的一种队列。它是一个假想的环,是通过取模运算来使其首尾相连。特别要注意的是,在循环顺序队列中的入队和出队操作实现与在非循环顺序队列中的入队和出队操作实现的不同点在于,队首和队尾指针的变化不再是简单地加1或减1,而需加1或减1后再做取模运算。在循环顺序队列中为了区分队列的判空和判满条件,特别提出了3种解决方法:第一种是少用一个存储单元;第二种是设置一个标志变量;第三种是设置一个计数变量。链队列中的操作也与单链表中的操作类似,而且由于入队操作总是固定在链队列的队尾进行,而出队操作总是固定在链队列的队首进行,所以链队列的入队和出队操作实现起来非常简单。

栈与队列是两种十分重要的,并且应用非常广泛的数据结构。常见的栈的应用包括分隔符匹配问题的求解、表达式的转换和求值、函数调用和递归实现以及深度优先搜索遍历等。凡是遇到对数据元素的读取顺序与处理顺序相反的情况,都可考虑使用栈将读取到的而又未处理的数据元素保存在栈中。常见的队列的应用包括计算机系统中各种资源的管理、消息缓冲器的管理和广度优先搜索遍历等。凡是遇到对数据元素的读取顺序与处理顺序相同,都可考虑使用队列来保存读取到的而又未处理的数据元素。

优先级队列是带有优先级的队列。优先级队列中的每一个数据元素都有一个优先权。优先权可以比较大小,它既可以在数据元素被插入优先级队列时被人为赋予,也可以是数据元素本身所具有的某一属性。优先权的大小决定着该对象接受服务的先后顺序,所以也将其称为优先级。优先级队列不同于一般的队列,它是按照数据元素优先级的高低来决定出队的次序,而不是按照数据元素进入队列的次序来决定的。一般队列也可以被看作是一种特殊的优先级队列。在一般队列中,数据元素的优先级由其进入队列的时间确定,时间越

长，优先级越高。优先级队列的实现方法可以采用有序、无序线性表或栈来实现。本章中只介绍了在有序线性表上实现的优先级队列。在这种情况下，入队操作是进行数据元素的有序插入，即将待插入的数据元素插入队列中的适当位置，并使插入后的队列仍按照优先级从大到小的顺序排放。入队操作的时间复杂度为 $O(n)$。出队操作只要将队首元素（即优先级最高的元素）从队列中删除即可，其时间复杂度为 $O(1)$。在一些实际应用中，若需要采用一种数据结构来存储数据元素，对这种数据结构的要求是：数据元素加入的次序是无关紧要的，但每次取出的数据元素应是具有最高优先级的数据元素，这时就可以采用优先级队列来解决问题。

其实利用栈和队列的思想还可以设计出其他一些变种的栈和队列结构。例如双端队列、双端栈等，这些都是根据插入与删除操作位置受限的不相同而得名的。双端队列指插入和删除操作限制在线性表的两端进行；双端栈指两个底部相连的栈，它是一种添加限制的双端队列，并且规定从一端插入的数据元素只能从这一端删除。这些变种的栈和队列在某些特定情况下具有很好的应用价值。

习　题　3

一、客观测试题

扫码答题

二、算法设计题

1. 设计算法，要求借助一个栈把一个数组中的数据元素逆置。

2. 设计算法，判断一个字符序列是否为回文序列。所谓回文序列，就是正读与反读都相同的字符序列，例如：abba 和 abdba 均是回文序列。要求只借助栈来实现。

3. 设计算法，判断一个表达式中的花括号、圆括号和方括号是否配对。

4. 假设以一个数组实现两个栈：一个栈以数组的第一个存储单元作为栈底，另一个栈以数组的最后一个存储单元作为栈底，这种栈称为双向顺序栈或共享栈。试设计 3 个算法：一个是栈的初始化操作算法 InitStack(&S,maxSize)，此算法完成构造一个容量为 maxSize 的双向顺序空栈；一个是实现入栈操作的算法 Push(&S,i,x)，此算法完成将数据元素 x 压入到第 i($i=0$ 或 1)号栈中的操作；一个是实现出栈操作的算法 Pop(&S,i)，此算法完成将第 i 号栈的栈顶元素出栈的操作。

5. 假设循环顺序队列定义为：以域变量 rear 和 length 分别指示循环队列中队尾元素的位置和内含元素的个数。试设计实现相应的入队和出队的操作算法。

6. 假设采用带头结点的循环链表来表示队列，并且只设一个指向队尾元素的指针（不设队首指针），试设计相应的队列初始化、队列清空、队列判空、入队和出队的操作算法。

7. 设计算法，实现双端队列“队尾删除”和“队首插入”的操作算法，并且约定队首指针

指向队首元素的前一位置,队尾指针指向队尾元素。

8. 设计递归算法和非递归算法,实现将十进制数转换成k(2≤k≤9)进制的操作。

9. 设计递归算法和非递归算法,按反向输出一个单链表。

10. 已知递归函数(其中DIV为整除):

$$F(m)=\begin{cases}1, & m=0\\ mF(m\,\mathrm{DIV}\,2), & m>0\end{cases}$$

(1) 写出求$F(m)$的递归算法。

(2) 写出求$F(m)$的非递归算法。

习题3　主观题参考答案