第3章 栈和队列

CHAPTER 3

学习目标

1. 掌握栈和队列的定义、特点。

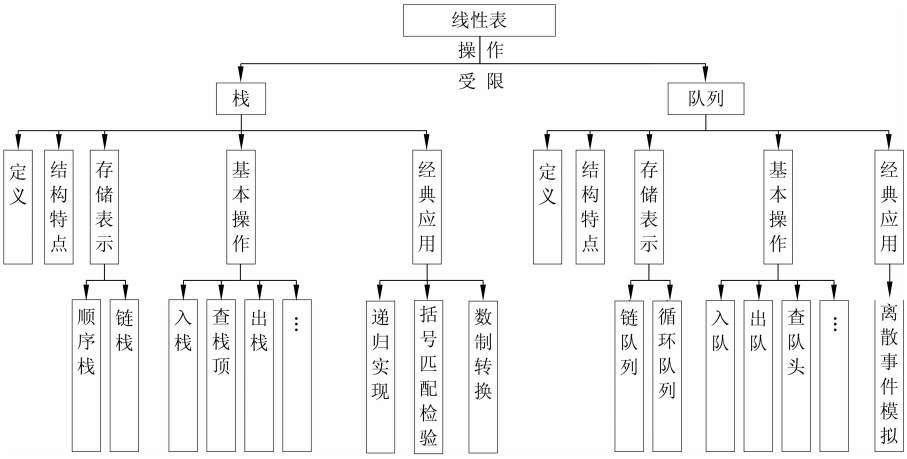
2. 掌握栈的顺序和链式存储表示及其在不同形式下基本操作的实现，注意栈空和栈满的条件。

3. 掌握队列的顺序和链式存储表示及其在不同形式下基本操作的实现，特别是循环队列中队列空和满的条件。

4. 理解递归的含义及其与栈的关系。

5. 能利用栈和队列的特点解决实际问题。

知识结构图



栈(stack)和队列(queue)作为两种重要的线性结构，在计算机科学中具有非常广泛的应用，从简单的表达式计算到编译器对程序语法的检查，再到操作系统对各种设备的管理等都会涉及。

从逻辑结构来说，栈和队列都是典型的线性结构。与线性表不同的是，基于栈和队列上的操作比较特殊，受到一定的限制，仅允许在线性表的一端或两端进行。因此，栈和队列常被称为操作受限的线性表，或者限制存取点的线性表。一般而言，栈的特点是后进先出，常用来处理具有递归

结构的数据；而队列的特点则是先进先出，在实际中体现出公平的原则，可以用来暂时存放需要按照一定次序处理但尚未处理的元素。

本章将对栈和队列及其典型应用进行介绍。

3.1 栈

3.1.1 栈的定义和特点

栈是限定仅在表尾进行插入和删除的线性表，通常，栈的插入和删除端为线性表的表尾，被称为栈顶(top)。与此相对，栈的另一端即线性表的表头端叫作栈底(bottom)。由此可知，最后插入栈中的元素是最先被删除或读取的元素，而最先压入的元素则被放在栈的底部，要到最后才能取出。换言之，栈的修改是按“后进先出”或者“先进后出”的原则进行。因此，通常栈被称为后进先出(Last In First Out，LIFO)或先进后出(First In Last Out，FILO)的线性表。

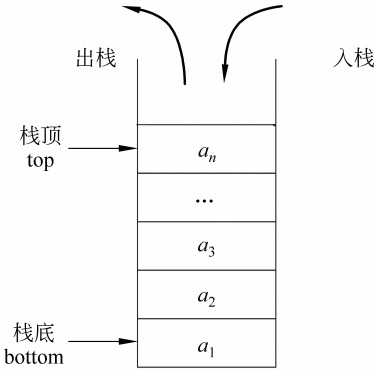


图 3-1 栈

假设栈 S=(a_1，a_2，…，a_n)，则称 a_1 为栈底元素，a_n 为栈顶元素。栈中元素按 a_1，a_2，…，a_n 的次序进栈，退栈的第一个元素应为栈顶元素，如图 3-1 所示。

在日常生活中，还有很多类似栈的例子。例如，叠放在一起的盘子可看成栈的一个实例，洗干净的盘子总是逐个叠放在上面，而用时从上往下逐个取用。尽管操作受限降低了栈的灵活性，但也正因为如此使得栈更有效且更容易实现。栈的操作特点正是上述实际应用的抽象。在程序设计中，如果需要按照与保存数据时相反的顺序来使用数据，则可以利用栈来实现。

如同线性表可以为空表一样，没有元素的栈称为空栈。例如，刚建立的栈一般是空栈，随着栈中所有元素的删除，栈也会变成空栈。

3.1.2 栈的类型定义

栈的应用非常广泛，并因此形成了栈的一些特殊术语。习惯上称往栈中插入元素为 push 操作，简称为进栈、压栈或入栈；删除栈顶元素被称为 pop 操作，简称为出栈、退栈或弹出。除此之外，栈的基本操作还有栈的初始化、栈空的判定，以及取栈顶元素等。下面给出栈的抽象数据类型定义：

```
ADT Stack {
    数据对象：D={aᵢ | aᵢ∈ElemSet, i=1, 2,…,n,n≥0}
    数据关系：R={<aᵢ₋₁, aᵢ>|aᵢ₋₁, aᵢ∈D, i=2,…,n }
    约定 aₙ端为栈顶，a₁ 为栈底。
    基本操作：
    InitStack(&S)          /* 初始化栈 */
```

```
        操作结果：构造一个空栈 S。
    StackEmpty(S)          /* 判断栈是否为空 */
        初始条件：栈 S 已存在。
        操作结果：若栈 S 为空栈，则返回 TRUE，否则返回 FALSE。
    StackLength(S)         /* 求栈长度 */
        初始条件：栈 S 已存在。
        操作结果：返回 S 的数据元素个数，即栈的长度。
    GetTop(S)              /* 取栈顶元素 */
        初始条件：栈 S 已存在且非空。
        操作结果：返回 S 的栈顶数据元素，不修改栈顶指针。
    Push(&S, e)            /* 入栈 */
        初始条件：栈 S 已存在。
        操作结果：将数据元素 e 插入为新的栈顶元素。
    Pop (&S, &e)           /* 出栈 */
        初始条件：栈 S 已存在且非空。
        操作结果：删除 S 的栈顶元素，并返回栈顶元素值 e。
    StackTraverse(S)       /* 栈的遍历 */
        初始条件：栈 S 已存在且非空。
        操作结果：从栈底到栈顶依次对 S 的每个数据元素进行访问。
}ADT Stack
```

本书在以后各章中引用的栈大多为如上定义的数据类型，栈的数据元素类型在应用程序内定义。

和线性表类似，栈也有两种存储表示方法，分别称为顺序栈和链栈。

3.1.3　顺序栈的表示和实现

1. 顺序栈及其表示

顺序栈是指利用顺序存储结构存储栈中的元素，即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，同时附设指针 top 指示栈顶元素在顺序栈中的位置。类似于顺序表的定义，顺序栈的存储结构可以有两种实现方式：一是对栈中的数据元素用一个预设的足够长度的一维数组来实现；二是栈中的数据元素不能确定，而且很难估计长度，对这样的栈用动态顺序结构来实现。无论采用何种方式，都需要附设栈顶指针 top，用来指示栈顶元素在顺序栈中的位置。另设指针 base 指示栈底元素在顺序栈中的位置。当 top 和 base 的值相等时，表示空栈。以动态顺序存储结构为例，定义如下：

```
//------顺序栈的存储结构------
#define STACK_INIT_SIZE 100        //存储空间初始分配量
#define STACKINCREMENT 10          //存储空间分配增量
typedef struct
```

```
{
    SElemType * base;          //栈底指针
    SElemType * top;           //栈顶指针
    int stacksize;             //栈当前可用的最大容量
}SqStack;
```

说明：

（1）在栈的类型定义中，base 为栈底指针，初始化完成后，栈底指针 base 始终指向栈底的位置，若 base 的值为 NULL，则表明栈结构不存在。

（2）top 为栈顶指针，为了操作方便，非空栈中的栈顶指针始终在栈最后一个元素的下一个位置上，其初值指向栈底，即当 top 和 base 的值相等时表示空栈；每当插入新的栈顶元素时，指针 top 增 1；删除栈顶元素时，指针 top 减 1，图 3-2 为空栈示意图。

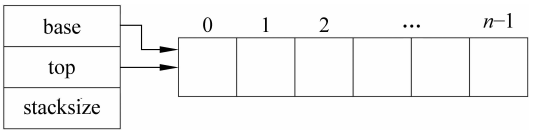


图 3-2　栈的动态分配存储示意图

（3）stacksize 指示当前栈可使用的最大容量，含义与顺序表中的 listsize 相似。

顺序栈中数据元素的入栈、出栈和栈指针之间的对应关系，如图 3-3 所示。

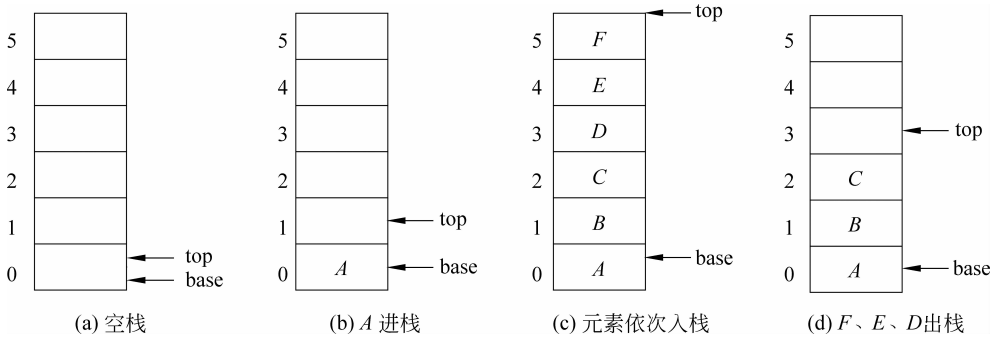


图 3-3　栈中元素和栈指针之间的关系

2. 顺序栈基本操作的实现

由于顺序栈的插入和删除只在栈顶进行，因此顺序栈的基本操作比顺序表要简单得多，以下给出顺序栈部分操作的实现。

1）初始化

顺序栈的初始化操作就是为顺序栈动态分配一个预定义大小的数组空间，其实质就是对栈定义中的 3 个分量赋值。

【算法步骤】

① 为顺序栈动态分配一个容量为 STACK_INIT_SIZE 的数组空间，使 base 指向这

段空间的基地址，即栈底。

② 栈顶指针 top 初始为 base，表示栈为空。

③ stacksize 置为栈的容量 STACK_INIT_SIZE。

算法 3-1　顺序栈的初始化

```
Status InitStack(SqStack &S)
{  //构造一个空栈 S
     S.base=(SElemType*)malloc(STACK_INIT_SIZE* sizeof(SElemType));
     //分配初始的数组空间
     if(!S.base) exit (OVERFLOW);      //存储分配失败
     S.top=S.base;                     //top 初始为 base,空栈
     S.stacksize=STACK_INIT_SIZE;      //stacksize 置为栈的最大容量 MAXSIZE
     return OK;
}
```

2）入栈

入栈操作是指在栈顶插入一个新的元素，同时修改栈顶指针。

【算法步骤】

① 判断栈是否满，若满：

- 追加增量空间，若不成功，则分配失败，退出；
- 设置新的栈底地址、栈顶地址、当前栈的空间长度 S. stacksize。

②将新元素压入栈顶，栈顶指针加 1。

算法 3-2　顺序栈的入栈

```
Status Push(SqStack &S, SElemType e)
{  //插入元素 e 为新的栈顶元素
     if (S.top-S.base==S.stacksize)
     {                                      //栈满
         newbase=(SElemType*)realloc(S.base,
         (S.stacksize+STACKINCREMENT)* sizeof(SElelmType));
         if(!newbase)exit(OVERFLOW);        //存储分配失败
         S.base=newbase;
         S.top=S.base+S.stacksize;
         S.stacksize+=STACKINCREMENT;
     }
         *S.top=e;                          //元素 e 压入栈顶
         S.top++;                           //栈顶指针加 1
         return OK;
}
```

3）出栈

出栈操作是将栈顶元素删除，同时修改栈顶指针。

【算法步骤】

① 判断栈是否空，若空则返回 ERROR。

② 栈顶指针减 1，栈顶元素出栈。

算法 3-3　顺序栈的出栈

```
Status Pop(SqStack &S,SElemType &e)
{  //删除 S 的栈顶元素,用 e 返回其值
     if(S.top==S.base) return ERROR;        //栈空
     --S.top;                               //栈顶指针减 1
     e= * S.top;                            //将栈顶元素赋给 e
     return OK;}
```

4）取栈顶元素

当栈非空时，此操作返回当前栈顶元素的值，栈顶指针保持不变。

算法 3-4　取顺序栈的栈顶元素

```
Status GetTop(SqStack &S,SElemType e)
{  //返回 S 的栈顶元素,不修改栈顶指针
     if (S.top==S.base)return ERROR;    //栈空
     e= * (S.top-1);                    //返回栈顶元素的值,栈顶指针不变
     return OK;
}
```

算法 3-3 和算法 3-4 的区别在于，出栈操作需要改变栈顶指针的指向，而取栈顶元素不需要改变栈顶指针位置。

由于栈插入、删除的特性，即使用顺序存储结构存储，进行插入、删除等操作的时候，仍不需要移动数据元素，因此时间复杂度为 $O(1)$。

【例 3-1】 将某序列数据依次进栈，出栈时可以将出栈操作按任何次序穿插在进栈序列中，例如，整数 1、2、3、4 依次进栈，每执行进栈一次，就执行出栈一次，就可得到 1、2、3、4；如果 4 个数全部进栈后，再执行 4 次出栈，即可得到 4、3、2、1 的输出顺序。对 1、2、3、4 入栈序列，问能否得到 2，3，4，1 和 1，4，2，3 的出栈序列，如能得到写出操作过程，如不能说明其原因。

【问题分析】 这类问题是典型的栈操作问题，在现实中有很多应用实例。针对这类问题，可以根据入栈、出栈顺序，来判断出栈序列是否可行。

【解】 2341 能得到，入栈、出栈操作如下：Push(1)、Push(2)、Pop()、Push(3)、Pop()、Push(4)、Pop()、Pop()即可。

1423 不能得到，按要求进行入栈、出栈操作 Push(1)、Pop()、Push(2)、Push(3)、Push(4)、Pop()，此时栈中还有 2、3，如果出栈只能是 3、2 顺序，因此不可能得到 1423。

3.1.4　链栈的表示和实现

1. 链栈的类型定义

栈的顺序存储结构仍然保留着线性表的顺序存储分配的固有缺点，若栈中元素数目变化范围较大或不清楚栈元素的数目时，可以采用栈的链式存储方式，定义如下：

```
typedef struct StackNode
{
    ElemType data;
    struct StackNode * next;
}StackNode, * LinkStack;

LinkStack top;
```

同线性表的链式存储结构一样，栈中每一个元素用一个链结点表示，数据类型为StackNode。栈顶指针 top 用来指向当前栈顶元素所在链结点的存储位置。当栈为空时，top＝NULL，链栈由栈顶指针唯一确定。因为栈的操作都是在栈顶进行，所以链栈通常不带头结点。如图 3-4 所示为具有 4 个结点的链栈示意图。

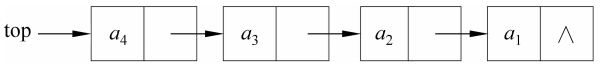


图 3-4　链栈示意图

2. 链栈基本操作的实现

1）初始化

链栈的初始化操作就是构造一个空栈，因为没必要设头结点，所以直接将栈顶指针置空即可。

算法 3-5　链栈元素 e 的初始化

```
Status InitStack(LinkStack &top)
{  //构造一个空栈 S,栈顶指针置空
     top=NULL;
     return OK;
}
```

2）入栈

根据栈的定义，在链栈中插入一个元素，实际上就是向 top 所指向的结点前插入新结点。和顺序栈的入栈操作不同的是，链栈在入栈前不需要判断栈是否满，只需要为入栈元素动态分配一个结点空间，如图 3-5 所示。

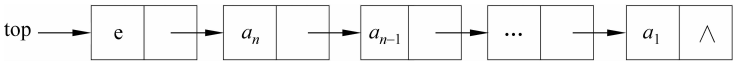


图 3-5　链栈元素 e 的入栈过程

【算法步骤】

① 为新结点申请空间，用指针 p 指向。

② 将新结点数据域置为 e。

③ 将新结点插入栈顶。

④ 修改栈顶指针为 p。

算法 3-6　链栈的入栈

```
Status Push(LinkStack &top, SElemType e)
{  //在栈顶插入元素 e
     p=(StackNode *)malloc(sizeof(StackNode));     //生成新结点
     p->data=e;                                    //将新结点数据域置为 e
     p->next=top;                                  //将新结点插入栈顶
     top=p;                                        //修改栈顶指针为 p
     return OK;
}
```

3）出栈

在删除链栈栈顶元素时，实际上就是删除栈顶指针 top 所指向链表的第一个结点。和顺序栈一样，链栈在出栈前也需要判断栈是否为空，不同的是，链栈在出栈后需要释放出栈前的栈顶空间，如图 3-6 所示。

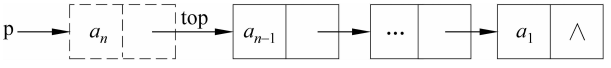


图 3-6　链栈的出栈过程

【算法步骤】

① 判断栈是否为空，若空则不能进行出栈操作，返回 ERROR。

② 将栈顶元素赋给 e。

③ 保存栈顶元素的指针，以备释放。

④ 修改栈顶指针，指向栈顶元素下一个结点。

⑤ 释放原栈顶元素的空间。

算法 3-7　链栈的出栈

```
Status Pop(LinkStack &top,SElemType &e)
{  //删除 top 的栈顶元素，用 e 返回其值
     if(top==NULL) return ERROR;          //栈空
     e=top->data;                         //将栈顶元素赋值给 e
     p=top;                               //用 p 保存栈顶元素指针，以备释放
     top=top->next;                       //修改栈顶指针
     free(p);                             //释放原栈顶元素的空间
     return OK;
}
```

4）顺序栈和链栈的比较

（1）时间性能比较：顺序栈和链栈插入或删除等基本操作的算法时间复杂度均为 $O(1)$。

（2）空间性能比较：初始时顺序栈必须确定一个固定的长度，所以有存储元素个数的限制和空间浪费的问题；链栈无栈满问题，只有当内存没有可用空间时才会出现栈满，但是每个元素都需要一个指针域，从而产生了结构性开销。

因此，当栈在使用过程中元素个数变化较大时，用链栈比较好，反之，应该采用顺序栈。

3.2　栈与递归

栈有一个重要应用是在程序设计语言中实现递归。递归是算法设计中最常用的手段之一，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，使用递归方法更加合适。为了增强理解和设计递归算法的能力，本节将介绍栈在递归算法的内部实现中所起的作用。

3.2.1　采用递归算法解决的问题

所谓递归，是指在一个函数内部直接或间接调用函数本身。根据调用方式的不同，将函数直接调用函数本身，称为直接递归；函数中调用另一函数，而在另一函数又调用函数本身的，称为间接递归。可以把递归现象分为如下 3 类。

1. 递归的定义

在数学中常用递归的方法定义函数，如阶乘函数：

$$\mathrm{Fact}(n)\begin{cases}1 & \text{若 } n=0\\ n*\mathrm{Fact}(n-1) & \text{若 } n>0\end{cases} \tag{3-1}$$

对于式(3-1) 中的阶乘函数，可以使用递归过程来求解，图 3-7 所示为主程序调用函数 Fact()的执行过程。

```
long Fact(long n)
{
    if(n==0) return 1;          //递归终止的条件
    else return n*Fact(n-1); //递归步骤
}
```

当 $n=4$ 时，在函数过程体中，else 语句以参数 3、2、1、0 执行递归调用。最后一次递归调用的函数因参数 n 为 0 执行 if 语句，递归终止，逐步返回，返回时依次计算 1*1、2*1、3*2、4*6，最后将计算结果 24 返回给主程序。

对于类似的复杂问题，若能够分解成几个相对简单且解法相同或类似的子问题来求

解，便称作递归求解。例如，在图 3-7 中，计算 4!时先计算 3!，然后再进一步分解进行求解，这种分解-求解的策略叫作“分治法”。

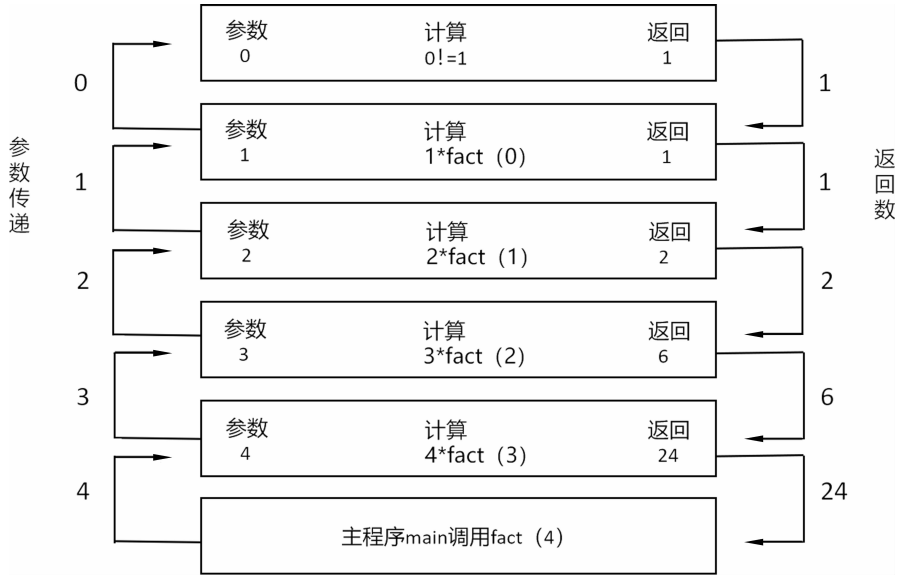


图 3-7　求解 4!的过程

采取“分治法”进行递归求解的问题需要满足以下 3 个条件。

(1) 能将一个问题转变成一个新问题，而新问题与原问题的解法相同或类同，不同的仅是处理的对象，并且这些处理对象更小且变化有规律。

(2) 可以通过上述转化而使问题简化。

(3) 必须有一个明确的递归出口，或称递归的边界。

“分治法”求解递归问题算法的一般形式为

```
void p(参数表)
{
    if(递归结束条件成立)可直接求解;          //递归终止的条件
    else p(较小的参数);                      //递归步骤
}
```

可见，上述阶乘函数的递归过程均与一般形式相对应。

2. 数据结构递归

某些数据结构本身具有递归的特性，因此用递归来描述它们的操作。

例如，对于链表，其结点 LNode 的定义由数据域 data 和指针域 next 组成，而指针域 next 是一种指向 LNode 类型的指针，即 LNode 的定义中又用到了其自身，所以链表是一种递归的数据结构。

对于递归的数据结构，相应算法采用递归的方法来实现特别方便。链表的创建和链

表结点的遍历输出都可以采用递归的方法。算法 3-8 是从前向后遍历输出链表结点的递归算法，调用此递归函数前，参数 p 指向单链表的首元结点，在递归过程中，p 不断指向后继结点，直到 p 为 NULL 时递归结束。显然，这个问题满足上述给出的采用“分治法”进行递归求解的问题需要满足的 3 个条件。

【算法步骤】

① 如果 p 为 NULL，递归结束返回。

② 否则输出 p->data，p 指向后继结点继续递归。

算法 3-8　遍历输出链表中各个结点的递归算法

```
void TraverseList(LinkList p)
{
    if(p==NULL) return;          //递归终止
    else
    {
        printf(p->data);         //输出当前结点的数据域
        TraverseList(p->next);   //p 指向后继指点继续递归
    }
}
```

后面章节要介绍的广义表、二叉树等也是典型的具有递归特性的数据结构，其相应算法也可采用递归的方法来实现。

3. 问题的解法是递归的

虽然问题本身并没有明显的递归结构，但用递归来理解并求解时比迭代更容易，程序更简单，如典型的 Hanoi 塔问题和迷宫问题等。

【例 3-2】 n 阶 Hanoi 塔问题，如图 3-8 所示：有 A、B 和 C 三个塔座，A 上套有 n 个直径不同的圆盘，按直径从小到大叠放，形如宝塔，编号 1，2，3，…，n。要求将 n 个圆盘从 A 移到 C，叠放顺序不变，移动过程中遵循下列原则：

（1）每次只能移一个圆盘；

（2）圆盘可在 3 个塔座上任意移动；

（3）任何时刻，每个塔座上不能将大盘压到小盘上。

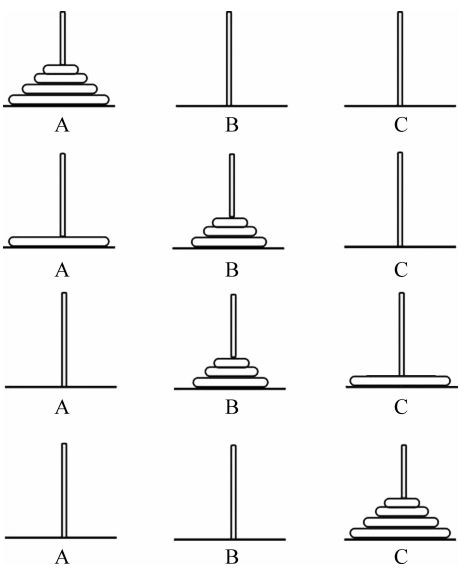


图 3-8　Hanoi 塔问题

【问题分析】 如何实现移动圆盘的操作呢？设 A 柱上最初的盘子总数为 n，则当 $n=1$ 时，只要将编号为 1 的圆盘从塔座 A 直接移至塔座 C 上即可；否则，执行以下 3 步：

（1）用 C 柱作为过渡，将 A 柱上的（$n-1$）个盘子移到 B 柱上；

（2）将 A 柱上最后一个盘子直接移到 C 柱上；

（3）用 A 柱作为过渡，将 B 柱上的($n-1$)个盘子移到 C 柱上。

所以 n 个圆盘的问题，转为 $n-1$ 个圆盘的问题，而根据这种解法，如何将 $n-1$ 个圆盘从一个塔座移至另一个塔座的问题是一个和原问题具有相同特征属性的问题，只是问题的规模小 1，因此可以用同样的方法求解。

为了便于描述算法，将搬动操作定义为 move(A,n,C)，是指将编号为 n 的圆盘从 A 移到 C。

【算法步骤】

① 如果 $n=1$，则直接将编号为 1 的圆盘从 A 移到 C，递归结束。

② 否则：

- 递归，将 A 上编号为 1 至 $n-1$ 的圆盘移到 B，C 做辅助塔；
- 直接将编号为 n 的圆盘从 A 移到 C；
- 递归，将 B 上编号为 1 至 $n-1$ 的圆盘移到 C，A 做辅助塔。

算法 3-9　Hanoi 塔问题的递归算法

```
void Hanoi(int n,char A,char B,char C)
{  //将塔座 A 上的 n 个圆盘按规则搬到 C 上，B 做辅助塔
     if(n==1) move(A,1,C);          //将编号为 1 的圆盘从 A 移到 C
     else
     {
         Hanoi(n-1,A,C,B);          //将 A 上编号为 1 至 n-1 的圆盘通过 C 移到 B
         move(A,n,C);               //将编号为 n 的圆盘从 A 移到 C
         Hanoi(n-1,B,A,C);          //将 B 上编号为 1 至 n-1 的圆盘通过 A 移到 C
     }
}
```

3.2.2　递归过程与递归工作栈

一个递归函数在函数的执行过程中，需多次进行自我调用。调用函数和被调用函数之间的链接及信息交换需通过栈来进行。

通常，当在一个函数的运行期间调用另一个函数时，在运行被调用函数之前，系统需先完成 3 件事：

（1）将所有的实参、返回地址等信息传递给被调用函数保存；

（2）为被调用函数的局部变量分配存储区；

（3）将控制转移到被调函数的入口。

而从被调用函数返回调用函数之前，系统也应完成 3 件工作：

（1）保存被调函数的计算结果；

（2）释放被调函数所占的存储空间；

（3）把执行控制按调用时保存的返回地址，转移到调用函数中调用语句的下一条语句。

当有多个函数构成嵌套调用时，按照“后调用先返回”的原则，上述函数之间的信息传递和控制转移必须通过“栈”来实现，即系统将整个程序运行时所需的数据空间安排在一个栈中，每当调用一个函数时，就为它在栈顶分配一个存储区，每当从一个函数退出时，就释放它的存储区，当前正运行的函数的数据区必在栈顶。

递归函数的运行过程与多个函数的嵌套调用类似，是通过层层自身调用来实现的。假设调用递归函数的主函数为第 0 层，则从主函数调用递归函数进入第 1 层，……，从第 i 层递归调用本函数进入第 $i+1$ 层，函数由上向下调用，直到遇到边界条件（出口）为止。当边界条件满足时，再将函数值层层向上返回，即第 i 层递归应返回至 $i-1$ 层。

为了保证递归函数正确执行，系统需设立一个“递归工作栈”作为整个递归函数运行期间使用的数据存储区。每一层递归所需信息构成一个工作记录，其中包括所有的实参、所有的局部变量，以及上一层的返回地址。每进入一层递归，就产生一个新的工作记录压入栈顶。每退出一层递归，就从栈顶弹出一个工作记录，当前执行层的工作记录必是递归工作栈栈顶的工作记录，称这个记录为“活动记录”，如图 3-9 所示。

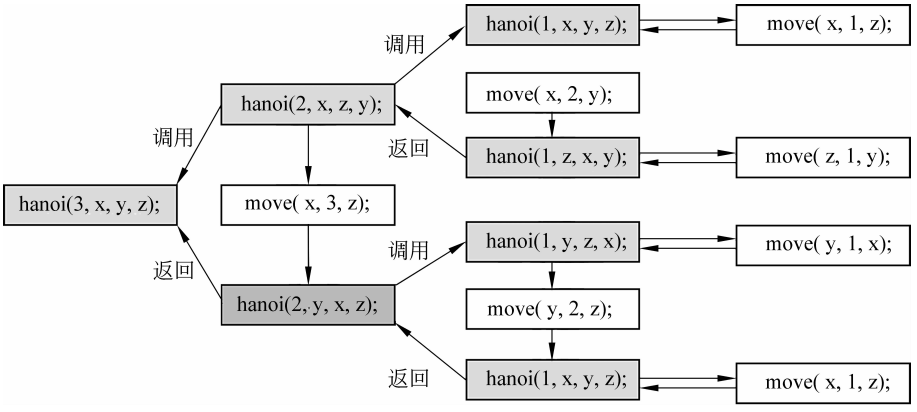


图 3-9　3 阶 hanoi 塔递归函数的运行过程

从上述递归调用过程分析可以看到，递归调用必须保存每次调用时的参数、变量和返回信息等，因此，从时间上讲并不经济，空间上也不节省。不过，递归程序比较紧凑，并且一般容易根据概念和定义直接编写，使得程序结构比较清晰。另外，在递归调用中系统开辟的工作栈数据区对用户来说是不可见的，这给用户编程或调试带来很大的方便。

3.3　队　　列

3.3.1　队列及其特点

队列是指只允许在表的一端进行插入操作，而在另一端进行删除操作的线性表。把允许进行插入操作的一端称为队尾（习惯用 rear 表示），把允许进行删除操作的一端称为队头（习惯用 front 表示）。队列的插入操作有时简称为入队，删除操作简称为出队。

队列的概念在日常生活中到处存在。如排队候车、排队买饭、排队参观等，任何一次

排队过程就形成了一个队列，它体现了“先到先服务”的处理原则，先来的排在前面，先得到服务，后来的只能排在队尾。因此，队列具有“先进先出”(First In First Out，FIFO)的特点，有时也叫先进先出表。

假设有一个队列结构 $Q=(a_1,a_2,\cdots,a_n)$，那么队头元素为 a_1，队尾元素为 a_n。如果队列的元素按 $a_1,a_2,\cdots,a_n$ 的顺序依次进入队列，则元素退出该队列也只能按照这个次序进行，如图 3-10 所示。

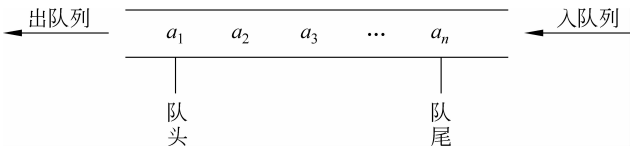


图 3-10 队列的示意图

3.3.2 队列的类型定义

队列的基本操作除插入、删除以外，还有队列的初始化、判空、求长度等。队列的抽象数据类型定义如下：

```
ADT Queue {
    数据对象：D={ai | ai∈ElemSet, i=1, 2, …, n, n>0}
    数据关系：R={<ai-1, ai> | ai-1, ai∈D, i=2,…,n}
              约定其中 a1 端为队列头，an端为队列尾。
    基本操作：
        InitQueue (&Q)          /*队列初始化*/
            操作结果：构造一个空队列 Q。
        QueueLength(Q)          /*求队列长度*/
            初始条件：队列 Q 已存在。
            操作结果：返回 Q 的元素个数，即队列的长度。
        GetHead(Q)              /*求队列首元素*/
            初始条件：Q 为非空队列。
            操作结果：返回 Q 的队头元素。
        EnQueue(&Q, e)          /*入队列*/
            初始条件：队列 Q 已存在。
            操作结果：插入元素 e 为 Q 的新的队尾元素。
        DeQueue(&Q, &e)         /*出队列*/
            初始条件：Q 为非空队列。
            操作结果：删除 Q 的队头元素，并用 e 返回其值。
        QueueTraverse(Q)        /*遍历队列*/
            初始条件：Q 已存在且非空。
            操作结果：从队头到队尾，依次对 Q 的每个数据元素访问。
}ADT Queue
```

和栈类似，在本书后面内容中引用的队列都是如上定义的队列类型，队列的数据元素

类型在应用程序内定义。

3.3.3　队列的顺序表示和实现

1. 顺序队列

队列有顺序表示和链式表示两种存储表示。队列的顺序存储结构称为顺序队列，是利用一组连续的存储单元（一维数组）依次存放从队首到队尾的各个元素。由于随着入队和出队操作的变化，队列的队头和队尾的位置是变动的，所以应设置两个整型变量 front 和 rear，分别指示队头和队尾在数组空间中的位置，通常称 front 为队头指针，rear 为队尾指针。队列的顺序存储结构表示如下：

```
//-----队列的顺序存储结构-----
#define MAXQSIZE 100        //队列可能达到的最大长度
typedef struct
{
    QElemType * base;       //存储空间的基地址
    int front;              //头指针
    int rear;               //尾指针
}SqQueue;

SqQueue Q;                  //定义队列变量
```

它们的初值在队列初始化时均应置为 0，并约定在非空队列里。

为了方便在 C 语言中描述，在此约定：初始化创建空队列时，令 Q. front＝Q. rear＝0，队首指针 Q. front 始终指向队头元素，队尾指针 Q. rear 始终指向队尾元素的下一个位置；每当插入新的队列尾元素时，尾指针 Q. rear 增 1；每当删除队列头元素时，头指针 Q. front 增 1，如图 3-11 所示。

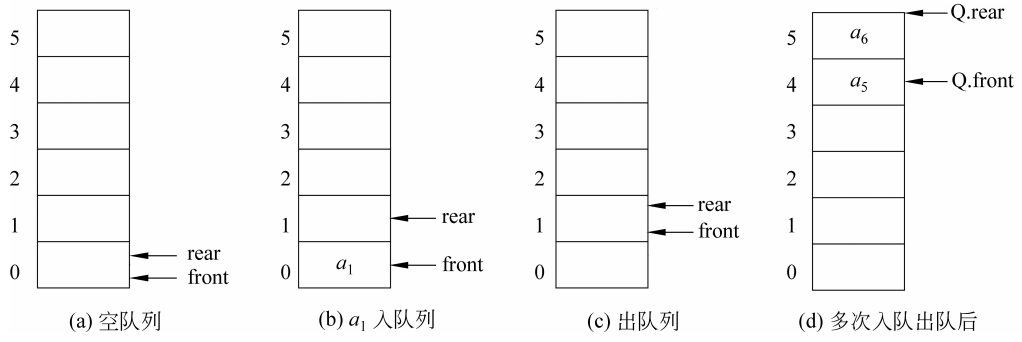


图 3-11　顺序分配的队列中头、尾指针和元素之间的关系

在队列刚建立时，Q. front＝Q. rear＝0，每当加入一个新元素时，先将新元素添加到 Q. rear 所指位置，再让队尾指针 Q. rear 加 1。因而指针 Q. rear 指示了实际队尾位置的后一位置，即下一元素应当加入的位置。而队头指针 Q. front 则不然，它指示真正队头元

素所在位置。所以，如果要退出队头元素，应当首先把 font 所指位置上的元素值记录下来，再让队头指针 Q. front＋1，指示下一队头元素位置，最后把记录下来的元素值返回。

假设当前队列分配的最大空间为 6，则当队列处于图 3-11(d)所示的状态时不可再继续插入新的队尾元素，否则会出现溢出现象，即因数组越界而导致程序的非法操作错误。事实上，此时队列的实际可用空间并未占满，所以这种现象称为“假溢出”。这是由“队尾入队，队头出队”这种受限制的操作造成的。

2. 循环队列

解决“假溢出”问题的一个较巧妙的办法就是循环队列，如图 3-12 所示。

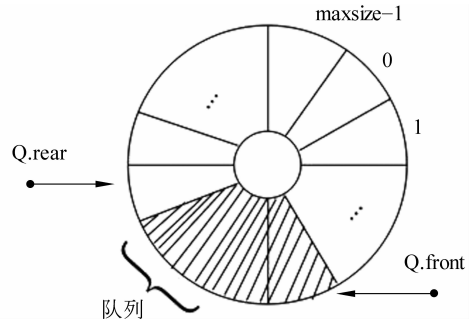


图 3-12　循环队列示意图

因为在数组的前端可能还有空位置。为了能够充分地使用数组中的存储空间，把数组的前端和后端连接起来，形成一个环形的表，即把存储队列元素的表从逻辑上看成一个环，成为循环队列(circular queue)，如图 3-12 所示，循环队列的首尾相接，头、尾指针以及队列元素之间的关系不变，当队头指针 Q. front 和队尾指针 Q. rear 进到 MAXQSIZE －1 后，再前进一个位置就自动到 0。这可以利用除法取余的运算(%)来实现，头指针和尾指针就可以在顺序表空间内以头尾衔接的方式“循环”移动。

队头指针进 1：Q. front＝(Q. front＋1)% MAXQSIZE；

队尾指针进 1：Q. rear＝(Q. rear＋1)% MAXQSIZE。

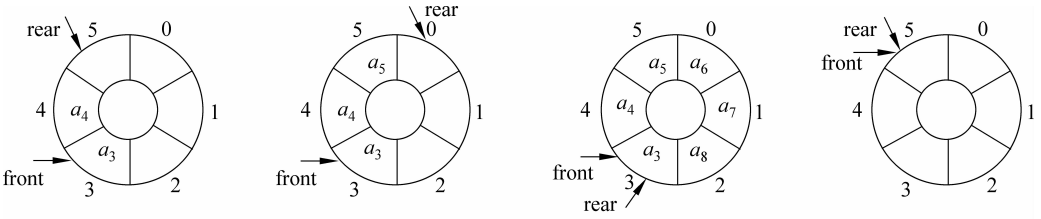
在图 3-13(a)中，Q. front 指向队头元素是 a_3，在元素 a_5 入队之前，Q. rear 指向位置 5，当元素 a_5 入队之后，通过“模”运算，Q. rear＝(Q. rear＋1)%6，得到 Q. rear 的值为 0，如图 3-13(b)所示。

在图 3-13(c)中，a_6、a_7、a_8 相继入队，队列空间被占满，此时头、尾指针相同。

在图 3-13(d)中，若 a_3 和 a_4 相继从图 3-13(a)所示的队列中出队，使队列此时呈“空”的状态，头、尾指针的值也是相同的。

由此可见，对于循环队列不能以头、尾指针的值是否相同来判别队列空间是“满”还是“空”。在这种情况下，如何区别队满还是队空呢？

通常有以下两种处理方法。



(a) 一般情况　(b) a_5 入队列　(c) a_6, a_7, a_8入队列　(d) a_3, a_4 出队列

图 3-13　循环队列中头、尾指针和元素之间的关系

(1) 少用一个元素空间，即队列空间大小为 MAXQSIZE 时，有 MAXQSIZE －1 个元素就认为是队满。这样判断队空的条件不变，即当头、尾指针的值相同时，则认为队空；而当尾指针在循环意义上加 1 后等于头指针，则认为队满。因此，在循环队列中队空和队满的条件如下。

① 队空的条件：Q. front＝Q. rear；

② 队满的条件：(Q. rear＋1)％MAXQSIZE＝Q. front。

如图 3-13(c)所示，当 a_6、a_7、a_8 进入图 3-13(b)所示的队列后，(Q. rear＋1)％MAXQSIZE 的值等于 Q. front，此时认为队满。

(2) 另设一个标志位以区别队列是“空”还是“满”。

3. 循环队列基本操作的实现

1) 初始化

循环队列的初始化操作就是动态分配一个预定义大小为 MAXQSIZE 的数组空间。

【算法步骤】

① 为队列分配一个最大容量为 MAXQSIZE 的数组空间，base 指向数组空间的首地址。

② 头指针和尾指针置为 0，表示队列为空。

算法 3-10　循环队列的初始化

```
Status InitQueue(SqQueue &Q)
{  //构造一个空队列 Q
     Q.base=(QElemType * )malloc(MAXQSIZE * sizeof(QElemType));
     //为队列分配一个最大容量为 MAXSIZE 的数组空间
     if(!Q.base) exit (OVERFLOW) ;          //存储分配失败
     Q.front=Q.rear=0;                      //头指针和尾指针置为零，队列为空
     return OK;
}
```

2) 求队列长度

对于非循环队列，尾指针和头指针之差便是队列长度，而对于循环队列，差值可能为负数，所以需要将差值加上 MAXQSIZE，然后与 MAXQSIZE 求余。

算法 3-11　求循环队列的长度

```
int QueueLength(SqQueue Q)
{  //返回 Q 的元素个数,即队列的长度
   return(Q.rear-Q.front+MAXQSIZE)% MAXQSIZE;
}
```

3）入队

入队操作是指在队尾插入一个新的元素。

【算法步骤】

① 判断队列是否满，若满则返回 ERROR。

② 将新元素插入队尾。

③ 队尾指针加 1。

算法 3-12　循环队列的入队

```
Status EnQueue(SqQueue &Q,QElemType e)
{  //插入元素 e 为 Q 的新的队尾元素
     if ((Q.rear+1) % MAXQSIZE==Q.front)     //表明队满
         return ERROR;
     Q.base[Q.rear]=e;                       //新元素插入队尾
     Q.rear=(Q.rear+1) % MAXQSIZE;           //队尾指针加 1
     return OK;
}
```

4）出队

出队操作是将队头元素删除。

【算法步骤】

① 判断队列是否为空，若空则返回 ERROR。

② 保存队头元素。

③ 队头指针加 1。

算法 3-13　循环队列的出队

```
Status DeQueue(SqQueue &Q,QElemType &e)
{  //删除 Q 的队头元素,用 e 返回其值
     if (Q.front==Q.rear) return ERROR;          //队空
     e=Q.base[Q.front] ;                         //保存队头元素
     Q.front=(Q.front+1) % MAXQSIZE;             //队头指针加 1
     return OK;
}
```

3.3.4　队列的链式表示和实现

循环队列在实现时必须为它设定一个最大队列长度，若用户无法预估所用队列的最

大长度，则宜采用链队列。

1. 链队列的定义和表示

所谓链队列，是指采用链式存储结构的队列，队列中每一个元素对应链表中的一个链结点。和链栈类似，一般用单链表来实现链队列，根据队列“先进先出”原则，为了操作方便，需要设置头指针和尾指针，分别指示队头和队尾，并限定删除在链表头、插入在链表尾进行。队列的链式存储结构表示如下：

```
//------队列的链式存储结构------
typedef struct QNode
{
    QElemType data;
    struct QNode * next;
}QNode, * QueuePtr;          /* 链队列结点类型 */

typedef struct{
    QueuePtr front;          /* 队头指针 */
    QueuePtr rear;           /* 队尾指针 */
}LinkQueue;                  /* 链队列类型 */
```

2. 两类不同指针类型的说明

设有如下两条变量的定义语句：

```
LinkQueue Q;
QueuePtr p;
```

(1) 其中 Q 是 LinkQueue 类型的变量，则 Q 有两个分量，一个是队头指针，表示为 Q. front，另一个是队尾指针，表示为 Q. rear。为了操作方便，给链队列添加一个头结点，并令头指针始终指向头结点，如图 3-14 所示。

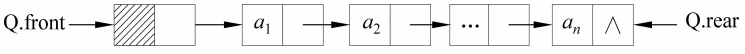


图 3-14　链队列示意图

(2) p 是 QueuePtr 类型的指针变量，p 是指向链式队列某结点的指针，其分量一个是 data，另一个是 next，该结点的数据域可表示为 p->data，该结点的指针域可表示为 p->next。

3. 链队列的基本操作

链队列的入队、出队操作即为单链表插入和删除操作的特殊情况，有时还需要进一步修改尾指针或头指针。下面给出链队列初始化、入

队、出队操作的实现。

1）初始化

链队列的初始化操作就是构造一个只有头结点的空队列，如图 3-15(a)所示。

【算法步骤】

① 为头结点开辟结点空间，队头和队尾指针指向此结点。

② 头结点的指针域置空。

算法 3-14　链队列的初始化

```
Status InitQueue(LinkQueue &Q)
{  //构造一个空队列 Q
     Q.front=Q.rear=(QNode*)malloc(sizeof(QNode));
                                        //生成头结点，也是队头和队尾指针
     Q.front->next=NULL;                //头结点的指针域置空
     return OK;
}
```

2）入队

链队列的结点是随着实际需求动态分配的，所以在入队前不需要判断队列是否为满，只需要直接为入队元素分配一个结点空间，如图 3-15(b)和(c)所示。

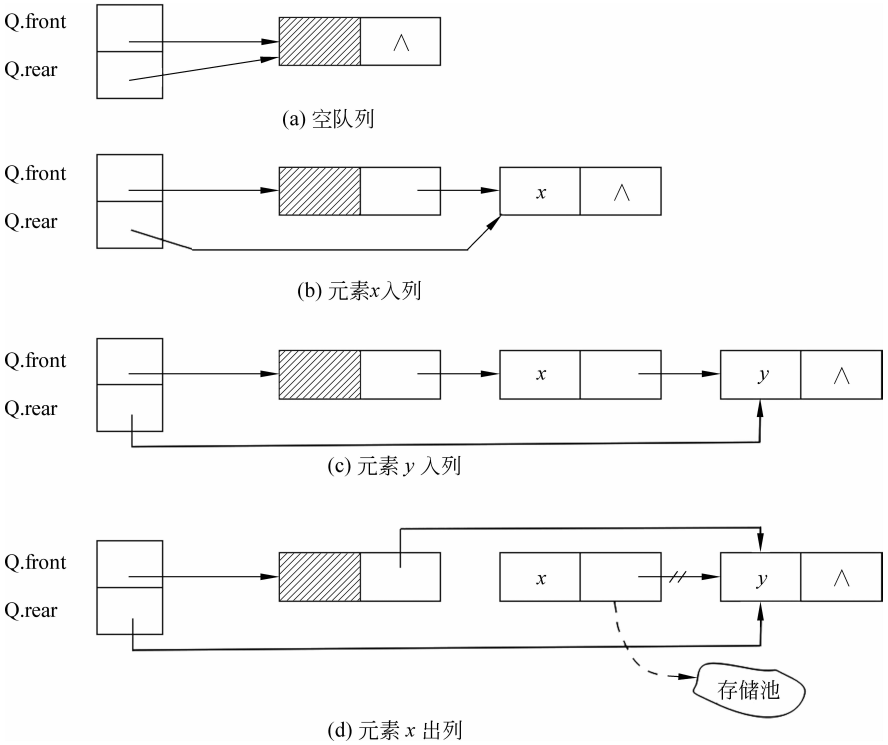


图 3-15　队列运算指针变化状况