第 3 章 栈和队列

CHAPTER 3

主要知识点

- 栈和队列的类型定义。
- 栈的存储表示及基于存储结构的基本操作实现。
- 队列的存储表示及基于存储结构的基本操作实现。

栈(Stack)和队列(Queue)是两种重要的线性结构,从数据结构的角度看,栈和队列也是线性表,其特殊性在于栈和队列的基本操作是线性表操作的子集,它们是操作受限制的线性表,因此可以称为限定性的数据结构。但从数据类型的角度看,它们是和线性表大不相同的两类抽象数据类型。栈和队列在操作系统、编译原理及大型应用软件系统中得到了广泛的应用。

第 9 集
微课视频

3.1 栈

栈的特点在于其基本操作的特殊性,即它必须按照“后进先出”的规则进行操作。与线性表相比,其插入和删除操作受到更多的约束和限定。

3.1.1 栈的类型定义

在日常生活中,有很多“后进先出”的例子。例如,把餐厅里洗净的一摞碗看作一个栈。通常,后洗净的碗总是摞在先洗净的碗上面,最后洗净的碗摞在最上面;而使用时却是从这摞碗顶端拿取,即后洗净的先取用。栈的操作特点正是上述实际的抽象。

1. 栈的定义

栈是限定仅可以在表尾进行插入和删除操作的线性表。允许插入和删除的一端称为栈顶(Top),栈顶将随着栈中数据元素的增减而浮动,且通过栈顶指针指明当前数据元素位置;不允许插入和删除的一端称为栈底(Bottom),栈底指针并不随着栈中数据元素的增减而移动,它是固定的。不含任何数据元素的栈称为空栈(Empty Stack)。

如图 3-1 所示,栈中有 3 个数据元素,插入元素(也称入栈)顺序是 a_1、a_2、a_3,当需要删除元素(也称出栈)时,其顺序只能为 a_3、a_2、a_1。换言之,在任何时候出栈的数据元素都只能是栈顶元素,即最后入栈者最先出栈,所以栈中元素除了具有线性关系外,还具有后进先出(Last In First Out,LIFO)的特性。栈也可以简称为 LIFO 表。

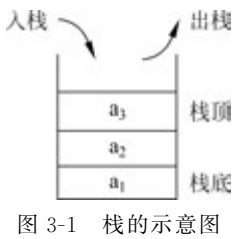


图 3-1 栈的示意图

图 3-2 所示的铁路调度站 A 是一个栈。车辆由轨道 B 进入调

度站 A,调度站 A 中的车辆从轨道 C 离去；后进入调度站 A 的车辆将先调出。

图 3-2　铁路调度示意图

在程序设计语言中,也有很多栈的应用例子。例如,在对程序设计语言编写的源程序进行编译时,类似于表达式括号匹配问题就是用栈来解决的。又如,计算机系统在处理子程序之间的调用关系时,用栈来保存处理执行过程中的调用层次等。

2. 栈的抽象数据类型

```
ADT Stack  {
    Data:
        D = {a_i | a_i ∈ ElemSet, i = 1,2, …,n, n≥0}(具有相同类型的数据元素集合)
    Relation:
        R = {< a_i-1 , a_i > | a_i-1 , a_i ∈ D, i = 2, …,n}(约定 a_n 端为栈顶,a_1 端为栈底)
    Operation:
        InitStack(&S)
            初始条件: 无。
            操作结果: 构造一个空栈 S。
        DestroyStack(&S)
            初始条件: 栈 S 已经存在。
            操作结果: 销毁 S。
        ClearStack(&S)
            初始条件: 栈 S 已经存在。
            操作结果: 重置 S 为空栈。
        StackLength(S)
            初始条件: 栈 S 已经存在。
            操作结果: 返回 S 中的数据元素个数。
        GetTop(S,&e)
            初始条件: 栈 S 已经存在且非空。
            操作结果: 用 e 返回 S 的栈顶数据元素。
        Push(&S,e)
            初始条件: 栈 S 已经存在。
            操作结果: 插入数据元素 e 为 S 的新栈顶元素。
        Pop(&S,&e)
            初始条件: 栈 S 已经存在且非空。
            操作结果: 删除 S 的栈顶数据元素,并用 e 返回其值。
} ADT Stack
```

3.1.2　栈的存储结构及操作实现

和线性表类似,栈也有两种存储结构：顺序存储结构和链式存储结构。

1. 顺序栈的定义

把自栈底到栈顶的元素按照逻辑顺序依次存放在一组地址连续的存储单元里的方式称为栈的顺序存储结构，采用这种存储结构的栈称为顺序栈(Sequential Stack)。同时附设指针 top 指示栈顶元素在顺序栈中的位置(相对地址)。

通常的做法是以 top＝0 表示“空栈”。鉴于 C 语言中数组下标约定从 0 开始，因此对用 C 语言描述的顺序栈需要以 top＝－1 表示空栈。图 3-3 给出了顺序栈中的栈顶指针和栈中元素之间的关系。

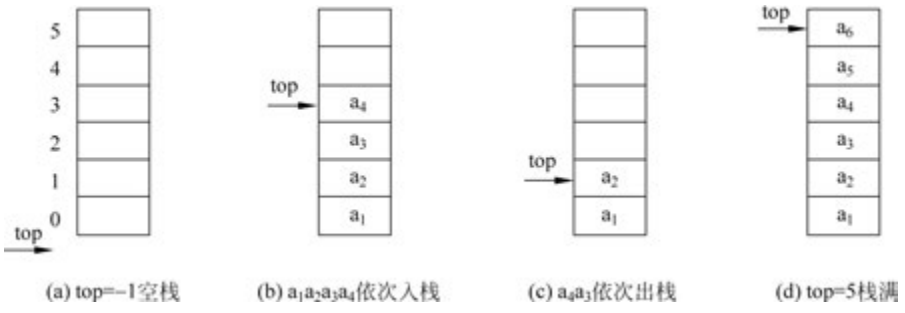


图 3-3　顺序栈中的栈顶指针和栈中元素之间的关系

2. 顺序栈的类型定义

采用结构类型来定义顺序栈类型，在顺序栈的结构定义中：

(1) 类似于顺序表，用一维数组描述顺序栈中数据元素的存储区域；

(2) 考虑到栈顶位置是随着入栈和出栈操作而变化的，应该设计一个整型量 top(通常称为栈顶指针)来指示当前栈顶数据元素的相对位置；

(3) 考虑栈的长度可变，需要设计一个表示栈当前存储空间的域。

```
#define STACK_INIT_SIZE   100                  // 栈存储空间的初始分配量
#define STACK_INCREMENT   10                   // 栈存储空间的分配增补量
typedef struct  {
    ElemType   * elem;                         // 栈存储空间基地址
    int         top;                           // 栈顶指针，栈非空时指向栈顶元素
    int         stacksize;                     // 当前分配的存储容量(以 ElemType 为单位)
} SqStack;
```

因为栈所需要的存储空间会随问题不同而异，所以为顺序栈定义了一个“存储空间的分配增补量”，为动态扩充数组容量提供方便；也就是说，一旦因为插入数据元素造成空间不足时可进行再分配，为顺序栈增加一个大小为 STACK_INCREMENT 个数据元素的空间。

3. 顺序栈的操作实现

(1) 初始化操作。

算法描述如算法 3-1 所示。

算法 3-1

```
void InitStack_Sq(SqStack &S)  {
// 构造一个空顺序栈 S
```

```
    S.elem = new ElemType[STACK_INIT_SIZE];
    if(!S.elem)  Error(" Overflow!");          // 存储分配失败
    S.top =- 1;
    S.stacksize = STACK_INIT_SIZE;
}                                              // InitStack_Sq
```

算法分析：算法 3-1 的时间复杂度为 O(1)。

(2) 销毁操作。

算法描述如算法 3-2 所示。

算法 3-2

```
void DestroyStack_Sq(SqStack S)  {
// 释放顺序栈 S 所占用的存储空间
    delete []S.elem;
    S.top =- 1;
    S.stacksize = 0;
} // DestroyStack_Sq
```

算法分析：算法 3-2 的时间复杂度为 O(1)。

(3) 清空操作。

算法描述如算法 3-3 所示。

算法 3-3

```
void ClearStack_Sq(SqStack &S)  {
// 重置顺序栈 S 为空栈
    S.top =- 1;
} // ClearStack_Sq
```

算法分析：算法 3-3 的时间复杂度为 O(1)。

(4) 求栈长操作。

算法描述如算法 3-4 所示。

算法 3-4

```
int StackLength_Sq(SqStack S)  {
// 返回顺序栈 S 的长度
    return(S.top + 1);
} // StackLength_Sq
```

算法分析：算法 3-4 的时间复杂度为 O(1)。

(5) 取栈顶元素值操作。

算法描述如算法 3-5 所示。

算法 3-5

```
void GetTop_Sq(SqStack S,ElemType &e) {
// 若顺序栈 S 为空,则给出相应信息并退出运行; 否则用 e 返回栈顶元素值
    if(S.top ==- 1)  Error("Stack Empty!");
    e = S.elem[S.top];
} // GetTop_Sq
```

算法分析：算法 3-5 的时间复杂度为 O(1)。

在该算法中，S.top==－1 表明栈已空，此时如果再作取栈顶数据元素值操作，则将因栈中无元素可取而发生溢出，称为“下溢(Below Overflow)”。栈的下溢常用来作为程序的控制条件。

(6) 入栈操作。

算法设计如下：

① 判断当前存储空间是否已满，如果已满，则需要系统增加空间；

② 先将栈顶指针加 1，再将新数据元素入栈顶。

算法描述如算法 3-6 所示。

算法 3-6

```
void Push_Sq(SqStack &S, ElemType e)  {
// 插入元素 e,作为新的栈顶元素
    if(S.top == (S.stacksize - 1))              // 若当前存储空间已满,则增加空间
        Increment(S);
    S.elem[++S.top] = e;                        // 栈顶指针先加 1,再将 e 压入栈顶
}                                               // Push_Sq

void Increment(SqStack &S)  {
// 为顺序栈 S 扩充 STACK_INCREMENT 个数据元素的空间
    newstack = new ElemType[S.stacksize + STACK_INCREMENT];
                                                // 增加 STACK_INCREMENT 个存储分配
    if(!newstack)  Error(" Overflow!");         // 存储分配失败
    for(i = 0;i <= S.top;i++)
        newstack[i] = S.elem[i];                // 腾挪原空间中的数据到 newstack 中
    delete []S.elem;                            // 释放元素所占用的原空间 S.elem
    S.elem = newstack;                          // 移交空间首地址
    S.stacksize += STACK_INCREMENT;             // 扩充后的顺序栈的最大空间
}                                               // Increment
```

算法分析：在正常情况下(不发生上溢)，算法 3-6 的时间复杂度为 O(1)；在非正常情况下(发生上溢)，算法 3-6 的时间复杂度为 O(n)。

在该算法中，如果 S.top==S.stacksize－1，则表明顺序栈已满，此时若再做入栈处理则将发生溢出，这种情况称为“上溢(Above Overflow)”。在实际应用中，栈的上溢一般有两种处理方法：一是给出诸如“栈满”“溢出”等出错信息，并退出运行，等候处理；二是为了不停止程序运行，重新分配空间(Increment(S))，即在原来顺序栈容量的基础上追加若干元素空间。显然，第二种处理方法的“扩充”操作比一次性申请空间要耗费时间。算法 3-6 中采用的是第二种方法。

(7) 出栈操作。

算法设计：

① 判断当前顺序栈是否为空，如果为空，则给出相应信息并退出运行；

② 先取出栈顶数据元素值，再将栈顶指针减 1。

算法描述如算法 3-7 所示。

算法 3-7

```
void Pop_Sq(SqStack &S,ElemType &e) {
// 若顺序栈 S 为空,给出相应信息并退出运行; 否则用 e 返回栈顶元素值并修改栈顶指针
    if(S.top ==- 1)  Error("Stack Empty!");     // 栈空,给出相应信息并退出运行
    e = S.elem[S.top -- ];
}                                               // Pop_Sq
```

算法分析：算法 3-7 的时间复杂度为 O(1)。

栈的链式存储结构如图 3-4 所示。由于栈的操作是线性表操作的特例，则链栈的操作类似单链表的操作，在此不做详细讨论，参见第 2 章第 2.3 节。

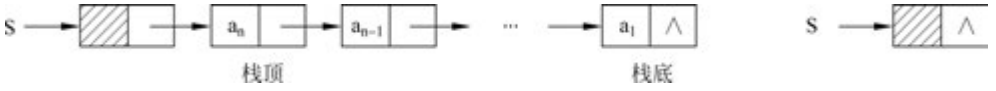


图 3-4　栈的链式存储结构

4. 栈的应用举例

例 3-1　数制转换问题。对于输入的任意一个非负十进制整数 N，输出与其等值的 D 进制数，例如八进制数、二进制数。

(1) 算法设计。

① 基本原理。

将一个非负十进制整数 N 转换为另一个等价的 D 进制数问题，可以通过“除 D 取余法”来解决。即

$$N=(N/D)\times D+N \text{ MOD } D$$

其中，MOD 为求余运算(取模)。

例如，将十进制数 1348 转换为八进制，即$(1348)_{10}=(2504)_8$，其运算过程如下。

N	N/8	N MOD 8
1348	168	4
168	21	0
21	2	5
2	0	2

又如，将十进制数 13 转换为二进制，即$(13)_{10}=(1101)_2$，其运算过程如下。

N	N/2	N MOD 2
13	6	1
6	3	0
3	1	1
1	0	1

② 实现方法。

由上可知，计算过程是按照从低位到高位的顺序来产生 D 进制数的各个数位；而输出过程是从高位到低位进行，恰好和计算过程相反。因此，如果将计算过程中得到的 D 进制

数的各位顺序入栈，则按照出栈序列打印输出的即为与输入对应的 D 进制数。

（2）算法描述。

算法 3-8

```
void Conversion() {
// 对于输入的任意非负十进制整数,打印输出与其等值的其他进制数
    InitStack_Sq(S);                          // 构造空顺序栈 S
    cin >> N;                                 // 输入任意非负十进制数 N
    cin >> D;                                 // 输入任意基数 D
    while(N)  {                               // 从低位到高位产生 D 进制数各个数位
        Push_Sq(S,N % D);                     // 将 N 取模 D 后入栈 S
        N = N/D;                              // 将 N 整除 D
    }
    while(!(S.top ==- 1)) {                   // 从高到低位输出 D 进制数各个数位
        Pop_Sq(S,e);                          // 将转换后的 D 进制数出栈 S
        cout << e;                            // 输出
    }
}                                             // Conversion
```

（3）算法分析。

① 问题规模：待转换的非负“十进制数”N 和“基”D。

② 基本操作：入栈操作、出栈操作。

③ 时间分析：算法执行时间主要花费在两个 while 循环的基本操作上，两个 while 循环的执行次数均为 $\log_D N$，因此算法 3-8 的时间复杂度为 $O(\log_D N)$。

例 3-2 括号匹配检验问题。对于输入的表达式中的任意括号串进行配对检测。

（1）算法设计。

① 基本原理。

在表达式中，可以包含圆括号“(”和“)”，且可以嵌套使用，但是需要检测表达式中的括号输入是否匹配：如果是“(( ))”序列或“(( ) ( ))”序列，则为配对正确；如果是“(( )”序列或“( ))(”序列，则为配对不正确。

检测表达式中的括号输入是否匹配的方法可以采用“期待的急迫程度”这个概念来描述。例如：

```
( …( …( …( …) …) …) …)
1   2   3   4   5   6   7   8
```

当计算机接受了第 1 个左括号后，它期待着与其匹配的第 8 个右括号出现，但等来的却是第 2 个左括号；此时第 1 个左括号只能暂时放到一边，而迫切期待着与第 2 个左括号匹配的第 7 个右括号出现；类似地，因为等来的是第 3 个左括号，其期待匹配的程度比第 2 个左括号更急迫，则第 2 个左括号也只能暂时放到一边；当等来的是第 4 个左括号时，由于其期待匹配的程度高于第 3 个左括号，所以第 3 个左括号也暂时放到一边；在接受了第 5 个右括号之后，第 4 个左括号的期待得到满足，消解之后，第 3 个左括号的匹配就成为当前最急迫的任务了，以此类推。可见，这个处理过程与栈的操作相同。

由此，可以自左至右扫描表达式（以按 Enter 键结束），并基于顺序栈设计一个算法来判断输入的一串括号是否配对正确。

② 实现方法。

左括号和右括号配对一定是先有左括号,后有右括号。因为括号是可以连续嵌套使用的,所以左括号允许单个或连续出现,并等待右括号出现而配对消解。左括号在等待右括号出现的过程中应暂时保存起来,当右括号出现时则一定是和最近出现的一个左括号配对并消解;当右括号出现而找不到左括号配对时则一定是发生了配对不正确的情况。左括号的这种保存和与右括号的配对消解过程与栈的"后进先出"原则是一致的。可以先将读到的左括号压入设定的栈中,当读到右括号时就和栈中左括号配对消解,即将栈顶的左括号弹出栈。如果栈顶弹不出左括号,则表示输入括号匹配出错;如果括号串已读完,栈中仍有左括号存在,则也表示输入括号匹配出错。

(2) 算法描述。

算法 3-9

```
int matching()  {
// 检验表达式中所含括号是否正确嵌套; 若是则返回 1,否则返回 0
    InitStack_Sq(S);                    // 构造空顺序栈 S
    Push_Sq(S,'#');                     // 将"#"入栈表示括号串开始
    ch = getchar();                     // 读取表达式中的一个字符
    state = 1;                          // state = 1,正确; state = 0,出错
    while((ch!= '\n')&&state)  {        // 只要表达式未结束且检验未出错继续检验
        if(ch == '(')  Push_Sq(S,ch);   // 若遇左括号则左括号入栈
        if(ch == ')')  {
            GetTop_Sq(S,e);             // 取栈顶元素
            if(e == '#')  state = 0;    // 若栈顶元素 = "#",则无左括号配对
            else  Pop_Sq(S,e);          // 若有左括号配对则消解
        }
        ch = getchar();
    }
    GetTop_Sq(S,e);
    if(e!= '#')  state = 0;             // 若栈顶元素≠"#",则左括号数多于右括号
    if(state)  return 1;                // 表达式中所含括号是正确嵌套,返回 1
    else  return 0;                     // 表达式中所含括号非正确嵌套,返回 0
}                                       // matching
```

(3) 算法分析。

① 问题规模:表达式串的"长度"(设值为 n)。

② 基本操作:字符比较。

③ 时间分析:算法执行时间主要花费在 while 循环的基本操作上,执行次数为 n,所以算法 3-9 的时间复杂度为 O(n)。

3.1.3 栈与递归问题

栈还有一个重要应用是在程序设计语言中实现递归。一个直接调用自己或通过一系列的调用语句间接地调用自己的函数,称为递归函数。

递归是程序设计中一种强有力的数学工具,它可以使问题的描述和求解变得简洁和清晰。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，使用递归算法特别合适。例如：

(1) 有很多数学函数是递归定义的，如非负整数的阶乘函数；

(2) 有的数据结构，如二叉树、广义表等，由于结构本身固有的递归特性，它们的操作可以递归地描述；

(3) 还有一类问题，虽然问题本身没有明显的递归结构，但用递归求解比迭代求解更简单，如八皇后问题、汉诺(Hanoi)塔问题等。

1. 递归算法的设计

(1) 递归基本步骤：将规模较大的原问题分解为一个或多个规模更小、但具有类似于原问题特性的子问题。即较大的问题递归地用较小的子问题来描述，解原问题的方法同样可以用来解这些子问题。

(2) 递归终止条件：确定一个或多个无须分解、可直接求解的最小子问题。

例如，非负整数 n 的阶乘可递归定义为

$$f(n)=\begin{cases}1, & n=0\\ n\times f(n-1), & n>0\end{cases} \tag{3-1}$$

2. 栈在递归算法的内部实现中所起的作用

1) 多个函数的嵌套调用

当在一个函数的运行期间调用另一个函数时，调用前系统需要先完成三件事情：

(1) 所有实参、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) 将控制权转移到被调用函数的入口。

调用后，系统也应该完成三件事情：

(1) 保存被调用函数的计算结果；

(2) 释放被调用函数数据区；

(3) 依照被调用函数保存的返回地址将控制权转移到调用函数。

当有多个函数构成嵌套调用时，按照“后调用先返回”的原则，函数之间的信息传递和控制必须通过“栈”来实现，即系统将整个程序运行时所需要的数据空间都安排在一个栈中：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。因此当前运行的函数的数据域必在栈顶。

2) 递归函数的运行过程

类似于多个函数的嵌套调用，只是调用函数和被调用函数是同一个函数，因此和每次调用相关的一个重要概念是递归函数运行的“层次”。假设调用该递归函数的主函数为第 0 层，则从主函数调用递归函数就为第 1 层；从第 i 层递归调用本函数为进入“下一层”，即第 i+1 层。反之，退出第 i 层递归应该返回至“上一层”，即第 i−1 层。为了保证递归函数正确执行，系统需要设立一个“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。每一层递归所需要的信息构成一个“工作记录”，其中包括所有实参、局部变量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，则当前执行层的工作记录必是递归工作栈栈顶的工作记录。

3. 汉诺塔问题

传说在世界刚创建时有一座钻石宝塔 A,塔上有 64 个金碟。所有碟子按照从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔 B 和 C。从世界创始之日起,牧师们就一直在试图把 A 上的碟子移到 C 上,其间借助于 B 的帮助。每次只能移动一个碟子,任何时候都不能把一个碟子放在比它小的碟子上面。

1) 算法设计

在该问题中,将一个金碟从一个塔移至另一个塔的问题是一个和原问题具有相同特征属性的问题,只是问题的规模小于 1,因此可以使用递归求解。递归模型如下。

基本项:当 n=1 时,将编号为 1 的金碟从 A 移至 C;

归纳项:当 n>1 时,将压在编号为 n 的金碟之上的 n-1 个金碟从 A 移至 B;将编号为 n 的金碟移至 C;将 B 上的 n-1 个金碟移至 C。

2) 算法描述

算法 3-10

```
1  void hanoi(int n,char A,char B,char C) {
   // 将 A 上按直径由大到小且自上而下编号为 1 到 n 的 n 个金碟按规则搬到 C 上,B 可用作辅助
2      if(n == 1) move(A,1,C);              // 将编号为 1 的金碟从 A 移到 C
3      else {
4          hanoi(n-1,A,C,B);                // 将 A 上编号为 n-1 的金碟移到 B,C 作辅助
5          move(A,n,C);                     // 将编号为 n 的金碟从 A 移到 C
6          hanoi(n-1,B,A,C);                // 将 B 上编号为 n-1 的金碟移到 C,A 作辅助
7      }
8  }                                        // hanoi
```

图 3-5 展示了语句 hanoi(3,A,B,C)的执行过程(从主函数进入递归函数到退出递归函数返回至主函数)中递归工作栈状态的变化情况。因为算法 3-10 中的递归函数中有 4 个实参数,所以每个工作记录包含 5 个数据项:4 个实参和返回地址,并以递归函数中的语句行号表示返回地址,同时假设主函数的返回地址为 0。

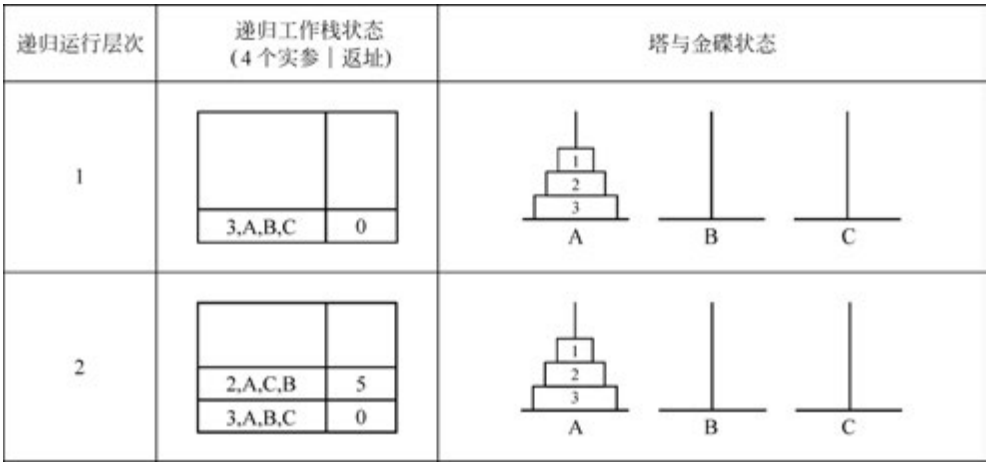


图 3-5 汉诺(Hanoi)塔的递归函数运行示意图

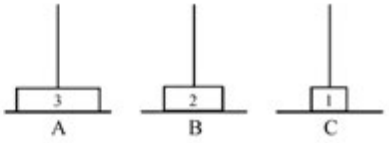
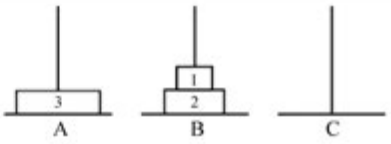
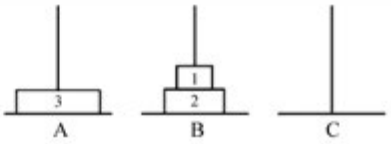
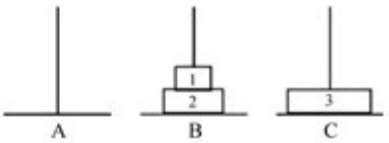
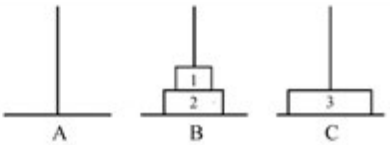
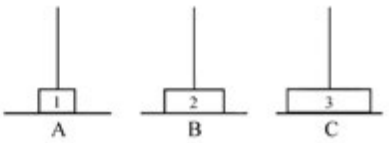
3	1,A,B,C　5 2,A,C,B　5 3,A,B,C　0	
2	2,A,C,B　5 3,A,B,C　0	
3	1,C,A,B　7 2,A,C,B　5 3,A,B,C　0	
2	2,A,C,B　5 3,A,B,C　0	
1	3,A,B,C　0	
2	2,B,A,C　7 3,A,B,C　0	
3	1,B,C,A　5 2,B,A,C　7 3,A,B,C　0	

图 3-5 （续）

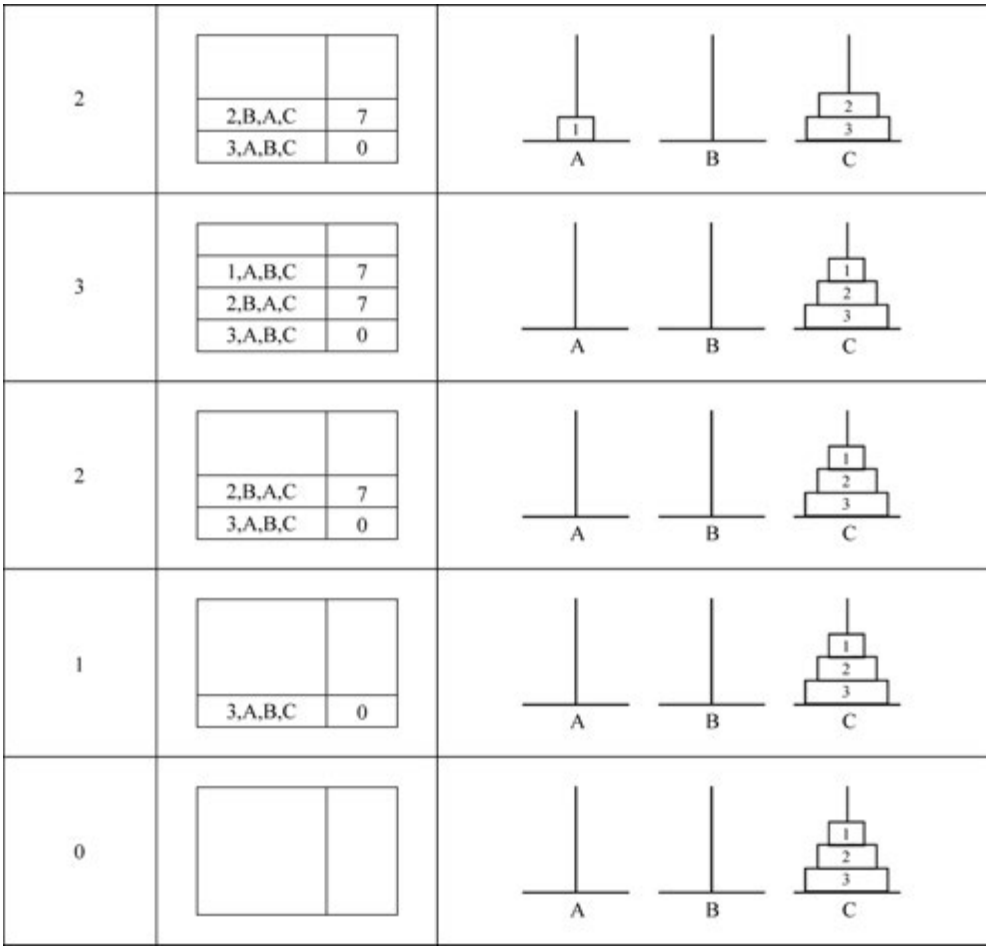


图 3-5 （续）

第 10 集
微课视频

由于递归函数结构清晰，程序易读，且其正确性也容易证明，因此利用允许递归调用的语言（如 C、C++语言）进行程序设计时，会为用户编制程序和调试程序带来很大方便。对这样一类递归问题编程时，不需要用户自己而是由系统来管理递归工作栈。

3.2 队列

队列的特点在于其基本操作的特殊性，即它必须按照“先进先出”规则进行操作。与线性表相比，它的插入和删除操作受到更多的约束和限定。

3.2.1 队列的类型定义

在日常生活中，经常会遇到为了维护社会正常秩序而需要排队的情景。在计算机程序设计中也经常出现类似问题。数据结构的“队列”与生活中的“排队”极为相似，也是按照“先到先办”的原则行事，并且严格限定：既不允许“加塞儿”，也不允许“中途离队”。

1. 队列的定义

队列是只允许在表的一端进行插入操作，而在表的另一端进行删除操作的线性表。允许插入的一端称为队尾(Rear)，队尾将随着队列中元素的增加而浮动，通过队尾指针指明队尾的位置；允许删除的一端称为队头(Front)，队头将随着队列中元素的减少而浮动，通过队头指针指明队头的位置。不含任何元素的队列称为空队列(Empty Queue)。

如图 3-6 所示，队列中有 5 个元素，插入元素(也称入队)顺序为 a_1、a_2、a_3、a_4、a_5，删除元素(也称出队)顺序依然是 a_1、a_2、a_3、a_4、a_5，即最先入队者最先出队，所以队列中的元素除了具有线性关系外，还具有先进先出(First In First Out，FIFO)的特性。队列也可以简称为 FIFO 表。

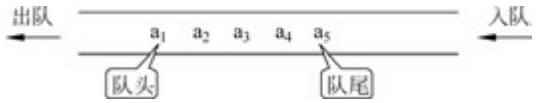


图 3-6　队列的示意图

在现实生活中有许多问题可以用队列来描述。例如，顾客服务部门的工作往往是按队列方式进行的，这类系统称为排队系统。在程序设计中，也经常使用队列记录需要按先进先出方式处理的数据，如键盘缓冲区、操作系统中的作业调度等。

例如，CPU 资源的竞争问题。在具有多个终端的计算机系统中，有多个用户需要使用 CPU 各自运行自己的程序，它们分别通过各自终端向操作系统提出使用 CPU 的请求，操作系统按照每个请求在时间上的先后顺序，将其排成一个队列，每次把 CPU 分配给队头用户使用，当相应的程序运行结束后令其出队，再把 CPU 分配给新的队头用户，直到所有用户任务处理完毕。

又如，主机与外部设备之间速度不匹配的问题。以主机和打印机为例来说明，主机输出数据给打印机打印，主机输出数据速度比打印机打印速度要快得多。直接把输出数据送给打印机打印，由于速度不匹配，显然是不行的，因此解决的方法是设置一个打印数据缓冲区，主机把要打印输出的数据依次写入这个缓冲区中，写满后就暂停输出，继而去做其他的事情，打印机就从缓冲区中按照先进先出的原则依次取出数据并打印，打印完后再向主机发出请求，主机接到请求后再向缓冲区写入打印数据，这样利用队列既保证了打印数据的正确，又使主机提高了效率。

2. 队列的抽象数据类型

```
ADT Queue {
    Data:
        D = {a_i | a_i ∈ ElemSet, i = 1,2,…,n, n≥0}(具有相同类型的数据元素集合)
    Relation:
        R = {< a_{i-1}, a_i > | a_{i-1}, a_i ∈ D, i = 2,…,n}(约定 a_1 端为队列头，a_n 端为队列尾)
    Operation:
        InitQueue(&Q)
            初始条件：无
            操作结果：构造一个空队列 Q。
        DestroyQueue(&Q)
            初始条件：队列 Q 已经存在。
```

```
            操作结果：销毁 Q。
        ClearQueue(&Q)
            初始条件：队列 Q 已经存在。
            操作结果：重置 Q 为空队列。
        QueueLength(Q)
            初始条件：队列 Q 已经存在。
            操作结果：返回 Q 的元素个数。
        GetHead(Q,&e)
            初始条件：队列 Q 已经存在且非空。
            操作结果：用 e 返回 Q 的队头元素。
        EnQueue(&Q,e)
            初始条件：队列 Q 已经存在。
            操作结果：插入元素 e 为 Q 的新队尾元素。
        DeQueue(&Q,&e)
            初始条件：队列 Q 已经存在且非空。
            操作结果：删除 Q 的队头元素，并用 e 返回其值。
} ADT Queue
```

3.2.2 队列的存储表示及操作实现

和线性表类似，队列也有两种存储表示：顺序存储表示和链式存储表示。

1. 循环队列及操作实现

1) 循环队列的定义

在队列的顺序存储结构中，除了用一组地址连续的存储单元依次存放队列头到队列尾的数据元素之外，还需要附设两个指针(front 和 rear)分别指示队列头元素和队列尾元素的位置。为了在 C 语言中描述方便起见，在此约定：初始化建立空队列时，令 front＝rear＝0，如图 3-7(a)所示；每当插入新的队列尾元素时，尾指针增 1；每当删除队列头元素时，头指针增 1。因此，在非空队列中，队列头指针 front 始终指向队列头元素，而队列尾指针 rear 始终指向队列尾元素的下一个位置。

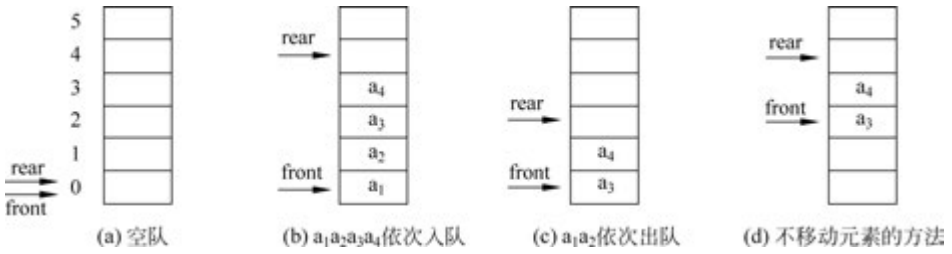


图 3-7 顺序存储队列中头、尾指针和队列中元素之间的关系

当队列有 n 个数据元素时，顺序存储的队列应该把队列的所有数据元素都依次存储在数组的前 n 个单元内。如果把队头元素放在数组中下标为 0 的一端，则入队操作(在队尾插入)的时间复杂度仅为 O(1)，此时的入队操作相当于追加，不需要移动元素，如图 3-7(b)所示；但出队操作(在队头删除)的时间复杂度为 O(n)，因为要保证剩下的 n－1 个元素仍然存储在数组的前 n－1 个单元，就必须将所有元素向前移动一个位置，如图 3-7(c)所示。如果把将队列的所有数据元素必须存储在数组的前 n 个单元这一条件放宽，只要求队列的元

素存储在数组中的连续位置上，则可以得到一种更为有效的存储方法，如图 3-7(d)所示。此时因为没有移动任何元素，入队和出队操作的时间复杂度都是 O(1)。图 3-7 给出了顺序存储队列中的队列头、尾指针和队列中数据元素之间的关系。

如果采用上述方法，则又遇到一个新的问题。随着入队和出队操作的进行，整个队列向数组中下标较大的位置移过去，从而产生了队列的“单向移动性”。当数据元素插入数组中下标最大的位置上之后，不可以再继续插入新的队尾元素，否则会因为数组越界导致程序代码被破坏；然而此时数组的低端还有空闲空间，这种现象称为“假溢出”，如图 3-8 所示，在这种状态下又不宜像顺序栈那样进行存储再分配扩大数组空间，因为队列的实际可用空间并未占满。解决“假溢出”的一种巧妙方法是将顺序队列臆造为一个环状的空间，如图 3-9 所示，即将存储队列的数组看成头尾相接的循环结构，允许队列直接从数组中下标最大的位置延续到下标最小的位置，也就是在逻辑上实现循环，而不是在物理上实现循环，如图 3-10 所示。利用数学上的取模操作很容易实现这种逻辑上的循环结构。队列的这种头尾相接的顺序存储结构称为循环队列(Circular Queue)。

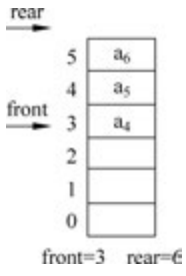


图 3-8　顺序队列假溢出示意图

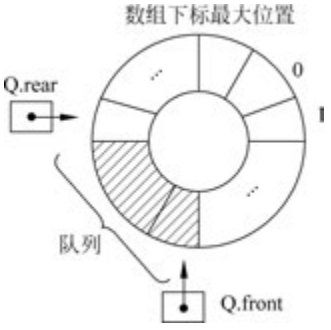


图 3-9　循环队列示意图

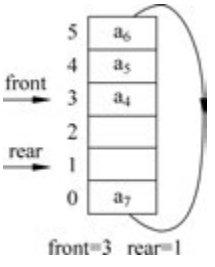


图 3-10　循环队列解决假溢出的示意图

设存储循环队列的数组长度为 QUEUE_MAX_SIZE，队列的长度用式(3-2)表示：

$$\text{QueueLength} = (\text{rear} - \text{front} + \text{Queue\_Size}) \% \text{QUEUE\_MAX\_SIZE} \tag{3-2}$$

当在循环队列的队尾插入元素后，队尾指针的修改用式(3-3)表示：

$$\text{rear} = (\text{rear} + 1) \% \text{QUEUE\_MAX\_SIZE} \tag{3-3}$$

当在循环队列的队头删除元素后，队头指针的修改用式(3-4)表示：

$$\text{front} = (\text{front} + 1) \% \text{QUEUE\_MAX\_SIZE} \tag{3-4}$$

在循环队列的操作中还要涉及两个很重要的问题：队列空和队列满的判定条件。在图 3-11(a)所示的队列中只有一个数据元素，如果此时执行出队操作，则队头指针加 1 后与队尾指针相等，即队列空的条件是 front＝rear，如图 3-11(b)所示。在如图 3-11(c)和图 3-11(e)所示的队列中只有一个空闲单元，如果此时执行入队操作，则队尾指针加 1 后与队头指针相等，即队列满的条件也是 front＝rear，如图 3-11(d)和图 3-11(f)所示。那么应该如何区分队列空和队列满的判定条件呢？此时可以有两种处理方法：

(1) 设立一个标志位，以区别循环队列是“空”还是“满”，但是这种方法会给算法设计带来复杂性。

(2) 少用一个数组元素空间，约定以“队列头指针在队列尾指针的下一个位置(指环状

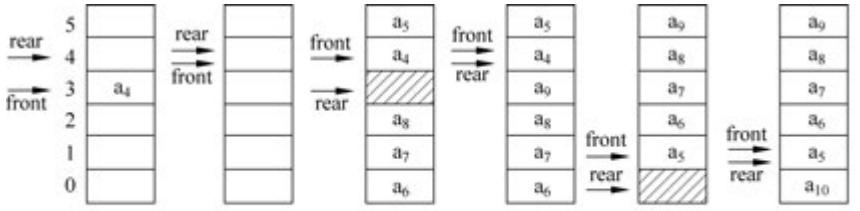


图 3-11　循环队列空和满的判定

的下一个位置)上"作为队列呈"满"状态的标志,即把图 3-11(c)和图 3-11(e)所示的情况视为队列满,此时队尾位置的指针和队头位置的指针正好相差 1,由此可以得到队列满和队列空不同的判定条件。该方法以牺牲一个元素空间为代价,换取算法设计的简单性,本书采用该方法。

队列满的判定条件用式(3-5)表示:

$$(\text{rear}+1)\%\text{QUEUE\_MAX\_SIZE}=\text{front} \tag{3-5}$$

队列空的判定条件用式(3-6)表示:

$$\text{front}=\text{rear} \tag{3-6}$$

2) 循环队列的类型定义

采用结构类型来定义循环队列类型,在循环队列的结构定义中:

(1) 类似于顺序表,用一维数组描述循环队列中数据元素的存储区域;

(2) 考虑到队头位置随着出队操作而变化,应该设计一个整型量 front(通常称为队头指针)来指示当前队头元素的相对位置;

(3) 考虑到队尾位置随着入队操作而变化,应该设计一个整型量 rear(通常称为队尾指针)来指示当前队尾元素的相对位置。

```
#define QUEUE_MAX_SIZE 100              // 循环队列大小,可根据实际需要而定
typedef struct {
    ElemType   *elem;                   // 存储空间基地址
    int         front;                  // 头指针,队列非空指向队头元素
    int         rear;                   // 尾指针,队列非空指向队尾元素下一位置
} SqQueue;
```

由以上分析可见,循环队列不能动态扩充数组容量。如果用户的应用程序中设有循环队列,则必须为它设定一个最大队列长度 QUEUE_MAX_SIZE;如果用户无法估计所用队列的最大长度,则宜采用链队列。

3) 循环队列的操作实现

(1) 初始化操作。

算法描述如算法 3-11 所示。

算法 3-11

```
void InitQueue_Sq(SqQueue &Q) {
// 构造一个空循环队列 Q
    Q.elem = new ElemType[QUEUE_MAX_SIZE];
    if(!Q.elem)  Error("Overflow!");     // 存储分配失败
```

```
    Q.front = Q.rear = 0;
}                                          // InitQueue_Sq
```

算法分析：算法 3-11 的时间复杂度为 O(1)。

(2) 销毁操作。

算法描述如算法 3-12 所示。

算法 3-12

```
void DestroyQueue_Sq(SqQueue &Q) {
// 释放循环队列 Q 所占用的存储空间
    delete []Q.elem;
    Q.front = Q.rear = 0;
} // DestroyQueue_Sq
```

算法分析：算法 3-12 的时间复杂度为 O(1)。

(3) 清空操作。

算法描述如算法 3-13 所示。

算法 3-13

```
void ClearQueue_Sq(SqStack &Q)  {
// 重置循环队列 Q 为空队列
    Q.front = Q.rear = 0;
} // ClearQueue_Sq
```

算法分析：算法 3-13 的时间复杂度为 O(1)。

(4) 求队列长操作。

算法描述如算法 3-14 所示。

算法 3-14

```
int QueueLength_Sq(SqQueue Q) {
// 返回循环队列 Q 的数据元素个数
    length = (Q.rear - Q.front + QUEUE_MAX_SIZE) % QUEUE_MAX_SIZE;
    return length;
} // QueueLength_Sq
```

算法分析：算法 3-14 的时间复杂度为 O(1)。

(5) 取队头元素值操作。

算法描述如算法 3-15 所示。

算法 3-15

```
void GetHead_Sq(SqQueue Q,ElemType &e) {
// 若循环队列 Q 为空,则给出相应信息并退出运行; 否则用返回队头元素值
    if(Q.front == Q.rear)  Error("Queue Empty!");
    e = Q.elem[Q.front];
} // GetHead_Sq
```

算法分析：算法 3-15 的时间复杂度为 O(1)。

在该算法中，Q.front==Q.rear 表明队列已经空，此时如果再作取队头元素值操作，则将因队列中无元素可取而发生溢出，称为“下溢(Below Overflow)”。

(6) 入队操作。

算法设计：

① 判断当前存储空间是否已满，如果已满，则给出相应信息并退出运行；

② 先将数据元素插入队尾，再根据式(3-3)修改队尾指针。

算法描述如算法 3-16 所示。

算法 3-16

```
void EnQueue_Sq(SqQueue &Q,ElemType e) {
// 若循环队列 Q 已满,则给出相应信息退出运行; 否则将元素 e 插入队尾,并修改队尾指针
    if(((Q.rear + 1) % QUEUE_MAX_SIZE) == Q.front)   Error("Queue Overflow!");
    Q.elem[Q.rear] = e;
    Q.rear = (Q.rear + 1) % Queue_Size;
} // EnQueue_Sq
```

算法分析：算法 3-16 的时间复杂度为 O(1)。

(7) 出队操作。

算法设计：

① 判断循环队列是否为空，如果为空，则给出相应信息并退出运行；

② 先取出队头数据元素值，再根据式(3-4)修改队头指针。

算法描述如算法 3-17 所示。

算法 3-17

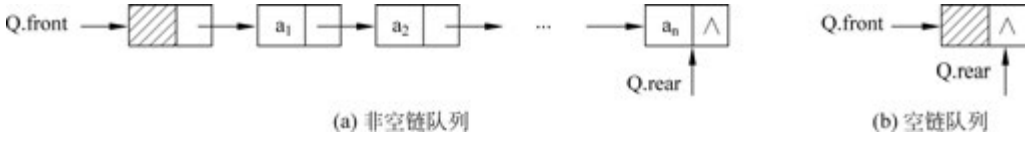
```
void DeQueue_Sq(SqQueue &Q,ElemType &e) {
// 若循环队列 Q 为空,则给出相应信息并退出运行; 否则用 e 返回队头元素,并修改队头指针
    if(Q.front == Q.rear)   Error("Queue Empty!");
    e = Q.elem[Q.front];
    Q.front = (Q.front + 1) % QUEUE_MAX_SIZE;
} // DeQueue_Sq
```

算法分析：算法 3-17 的时间复杂度为 O(1)。

2. 链队列及操作实现

1) 链队列的定义

用链表表示的队列称为链队列(Linked Queue)。一个链队列显然需要两个分别指示队头和队尾的指针(分别称为头指针和尾指针)才能唯一确定。这里，和线性表的单链表一样，为了操作方便起见，也给链队列添加一个同构节点，并令头指针指向头节点；而头节点指针域中的指针指向队列的第一个节点，尾指针指向队列的最后一个节点。当头指针和尾指针均指向头节点时，该链队列为空队列，如图 3-12 所示。链队列没有队列满的问题。



(a) 非空链队列　　(b) 空链队列

图 3-12　链队列的示意图

2) 链队列的类型定义

```
typedef struct QNode  {
```

```
    ElemType      data ;                  // 数据域
    struct QNode  * next ;                // 指针域
} QNode, * QueuePtr;
typedef struct  {
    QueuePtr      front;                  // 头指针,指向链队列头节点
    QueuePtr      rear;                   // 尾指针,指向链队列最后一个节点
} LinkQueue;
```

3) 链队列的操作实现

(1) 初始化操作。

算法描述如算法 3-18 所示。

算法 3-18

```
void InitQueue_L(LinkQueue &Q) {
// 构造一个空链队列 Q
    Q.front = Q.rear = new QNode;
    Q.front -> next = NULL;
} // InitQueue_L
```

算法分析：算法 3-18 的时间复杂度为 O(1)。

(2) 销毁操作。

算法描述如算法 3-19 所示。

算法 3-19

```
void DestroyQueue_L(LinkQueue &Q) {
// 释放链队列 Q 所占用的存储空间
    while(Q.front) {
        Q.rear = Q.front -> next;
        delete Q.front;
        Q.front = Q.rear;
    }
} // DestroyQueue_L
```

算法分析：

① 问题规模：队列的“当前长度”,即链队列的节点个数(设值为 n)。

② 基本操作：指针后移、释放节点存储空间。

③ 时间分析：算法执行时间主要花费在 while 循环的基本操作上,执行次数为 n+1,因此算法 3-20 的时间复杂度为 O(n)。

(3) 清空操作。

算法描述如算法 3-20 所示。

算法 3-20

```
void ClearQueue_L(LinkQueue &Q)  {
// 清空链队列 L,释放所有节点空间
    p = Q.front -> next;
    while(p)  {
        q = p;
        p = p -> next;
```

```
        delete q;
    }
    Q.front->next = NULL;   Q.rear = Q.front;
} // ClearQueue_L
```

算法分析：

① 问题规模：队列的“当前长度”，即链队列的节点个数(设值为 n)。

② 基本操作：指针后移、释放节点存储空间。

③ 时间分析：算法执行时间主要花费在 while 循环的基本操作上，执行次数为 n，因此算法 3-20 的时间复杂度为 O(n)。

(4) 求队列长操作。

算法描述如算法 3-21 所示。

算法 3-21

```
int QueueLength_L(LinkQueue Q) {
// 返回链队列 Q 的长度
    p = Q.front;                          // 设置指针，初始指向链队列头节点
    length = 0;                           // 设置计数器 length 初始为 0
    while(p->next)  {
        length++;
        p = p->next;
    }
    return length;
}                                         // QueueLength_L
```

算法分析：

① 问题规模：队列的“当前长度”，即链队列的节点个数(设值为 n)。

② 基本操作：计数器加 1、指针后移。

③ 时间分析：算法执行时间主要花费在 while 循环中的基本操作上，执行次数为 n，因此算法 3-21 的时间复杂度为 O(n)。

(5) 取队列头元素值操作。

算法描述如算法 3-22 所示。

算法 3-22

```
void GetHead_L(LinkQueue Q,ElemType &e) {
// 若链队列 Q 为空，则给出相应信息并退出运行；否则用 e 返回队头节点数据域的值
    if(Q.front == Q.rear)  Error("Queue Empty!");
    e = Q.front->next->data;
} // GetHead_L
```

算法分析：算法 3-22 的时间复杂度为 O(1)。

(6) 入队操作。

算法设计：

链队列的入队列操作只需要处理队尾节点，而不需要考虑其他节点。图 3-13 给出了链队列入队操作时指针的变化状况。

算法描述如算法 3-23 所示。

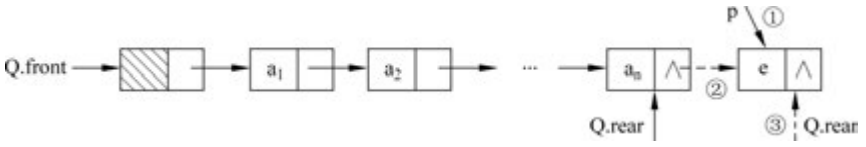


图 3-13　链队列入队操作时指针的变化状况

算法 3-23

```
void EnQueue_L(LinkQueue &Q;ElemType e) {
// 插入一个数据域的值为 e 的节点到链队列 Q 中,成为新的队尾节点
    p = new QNode;
    p->data = e;   p->next = NULL;          // 生成一个数据域的值为 e 的新节点
    Q.rear->next = p;                       // 将新节点插到队尾
    Q.rear = p;                             // 修改队尾指针,令其指向新节点
}                                           // EnQueue_L
```

算法分析：算法 3-23 的时间复杂度为 O(1)。

(7) 出队操作。

算法设计：

链队列的出队操作只需要处理队头节点,而不需要考虑其他节点。链队列的出队操作需要考虑两种情况：

① 当队列长度大于 1 时,用 e 返回链队列的队头节点数据域的值,并释放其存储空间。

② 当队列长度等于 1 时,即删除的既是队头节点又是队尾节点,还需要修改队尾指针。

图 3-14 给出了链队列出队操作时指针的变化状况。

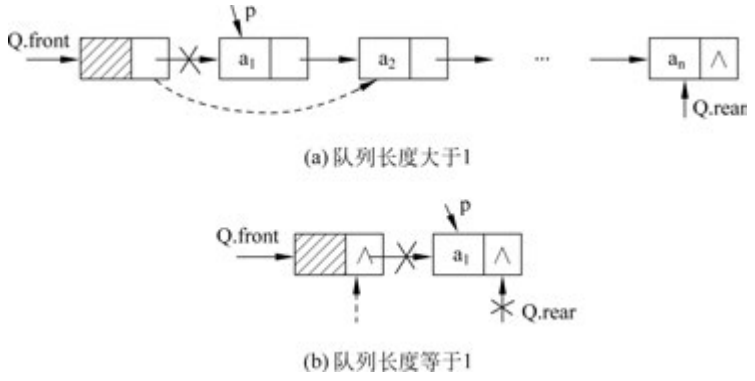


图 3-14　链队列出队操作指针的变化状况

算法描述如算法 3-24 所示。

算法 3-24

```
void DeQueue_L(LinkQueue &Q,ElemType &e) {
// 若链队列 Q 为空,则给出相应信息并退出运行; 否则用 e 返回队头节点数据域的值
    if(Q.front == Q.rear)  Error("Queue Empty!");
    p = Q.front->next;                      // 用指针 p 指向链队列 Q 的队头节点
    e = p->data;
    Q.front->next = p->next;                // 修改队头指针
```

```
    if(Q.rear == p)  Q.rear = Q.front;      // 若队列长度等于 1,则修改尾指针
    delete p;                               // 释放队头节点所占的存储空间
}                                           // DeQueue_L
```

算法分析：算法 3-24 的时间复杂度为 O(1)。

在链队列出队操作中应该注意队列长度等于 1 的特殊情况。一般情况下，删除队头节点时仅需要修改头节点指针域中的指针即可；但是当删除的既是链队列的队头节点又是队尾节点时，队列将变为空队列，此时队尾指针也丢失了，因此需要修改队尾指针，令其指向头节点。

3. 队列的应用举例

例 3-3 模拟患者在医院等候就诊的过程。患者到医院看病的顺序是：先排队等候，再看病治疗。在排队过程中重复做两件事情：一是当患者到达诊室时将病历交给护士，然后排到等候队列中候诊；二是护士从等候队列中取出下一个患者病历，让该患者进入诊室就诊。

(1) 算法设计。

患者等候就诊按照“先到先服务”的原则，因此利用队列存放患者的病历号：当“患者到达”时，即入队；当“护士让下一位患者就诊”时，即出队；当有“不再接收患者排队”时，即队列中所有元素出队。本算法采用链队列存储结构，“患者到达”用命令“A”(或“a”)表示，“护士让下一位患者就诊”用命令“N”(或“n”)表示，“不再接收患者排队”用“S”(或“s”)表示。

(2) 算法描述。

算法 3-25

```
void SeeDoctor() {
// 模拟患者在医院等候就诊的过程
    InitQueue_L(Q);                         // 初始化链队列 Q
    flag = 1;                               // flag = 1: 接收患者; = 0: 停止就诊
    while(flag) {
        cout <<"请输入命令: ";  cin >> ch;
        switch(ch) {
            case 'a' :
            case 'A' :
                cout <<"病历号: ";  cin > n;  cout << endl;
                EnQueue_L(Q,n);             // 入队等候就诊
                break;
            case 'n' :
            case 'N' :
                if(Q.front -> next) {
                    DeQueue_L(Q,n);         // 患者出队就诊
                    cout <<"病历号为"<< n <<"的患者就诊"<< endl;
                }
                else  cout <<"无患者等候就诊。"<< endl;
                break;
            case 's' :
            case 'S' :
                cout <<"下列排队的患者依次就诊: "
                while(Q.front -> next) {   // 所有患者依次出队就诊
```

```
                    DeQueue_L(Q,n);  cout << n <<" ";
                }
                cout << endl <<"今天不再接收患者排队!"<< endl;
                flag = 0;
                break;
            default :
                cout <<" 输入命令不合法!"<< endl;
        }
    }
}                                          // SeeDoctor
```

(3) 算法分析

① 问题规模：等候就诊的“患者”(设值为 n)。

② 基本操作：入队、出队。

③ 时间分析：算法 3-25 的时间复杂度为 O(n)。

例 3-4　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。如果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。

(1) 算法设计。

先入队的男士或女士也先出队配成舞伴，因此该问题具有典型的先进先出特性，可以采用队列作为算法的数据结构。本算法采用循环队列。

假设男士和女士的记录存放在一个数组中作为输入，然后依次扫描该数组的各数据元素，并根据性别来决定是进入男队还是女队。当这两个循环队列构造完成之后，依次将两队当前的队头元素出队来配成舞伴，直至某队列变空为止。此时，如果某队仍有等待配对者，则输出此队列中等待者的人数及排在队头的等待者的名字，他(或她)将是下一轮舞曲开始时第一个可以获得舞伴的人。

设计循环队列中数据元素的类型如下：

```
typedef struct {
    char   * name;                         // 姓名
    char   sex;                            // 性别,'F'表示女性,'M'表示男性
} Person;
typedef Person ElemType;                   // 将队列中数据元素类型改为 Person
```

(2) 算法描述。

算法 3-26

```
void DancePartner(Person dancer[],int num) {
// 结构数组 dancer 中存放跳舞的男女,结构数组中数据元素类型是 Person,num 是跳舞的人数
    InitQueue_Sq(Mdancers);                // 男士队列 Mdancers 初始化
    InitQueue_Sq(Fdancers);                // 女士队列 Fdancers 初始化
    for(i = 0;i < num;i++) {               // 依次将跳舞者依其性别入队
        p = dancer[i];
        if(p.sex == 'F')
            EnQueue_Sq(Fdancers,p);        // 排入女队
        else
            EnQueue_Sq(Mdancers,p);        // 排入男队
    }
```

```
    cout <<"The dancing partners are: \n \n";
    while(!(Fdancers.front == Fdancers.rear)&&!(Mdancers.front == Mdancers.rear)) {
                                                    // 依次输入男女舞伴名
        DeQueue_Sq(Fdancers,p);                     // 女士出队
        cout << p.name);                            // 打印出队女士名
        DeQueue_Sq(Mdancers,p);                     // 男士出队
        cout << p.name;                             // 打印出队男士名
        cout <<"\n";
    }
    if(!(Fdancers.front == Fdancers.rear)) {        // 输出女士剩余人数及队头女士的名字
        count = QueueLength_Sq(Fdancers);
        cout <<"There are"<< count <<" women wait in for the next round.\n "
        GetHead_Sq(Fdancers,p);                     // 取女队队头
        cout << p.name <<" will be the first to get a partner. \n"
    }
    if(!(Mdancers.front == Mdancers.rear)) {        // 输出男队剩余人数及队头者名字
        count = QueueLength_Sq(Mdancers);
        cout <<"\n There are"<< count <<" men wait in for the next round.\n "
        GetHead_Sq(Mdancers,p);                     // 取男队队头
        cout << p.name <<" will be the first to get a partner. \n"
    }
}                                                   // DancerPartners
```

(3) 算法分析。

① 问题规模：跳舞的人(设值为 num)。

② 基本操作：入队,出队。

③ 时间分析：算法 3-26 的时间复杂度为 O(num)。

习题

一、填空题

1. 线性表、栈和队列都属于(　　)结构。可在线性表(　　)位置插入和删除元素；对于栈只能在(　　)插入和删除元素；对于队列只能在(　　)插入元素和在(　　)删除元素。

2. 在操作序列 push(1),push(2),pop(),push(5),push(7),pop()和 push(6)之后,栈顶元素是(　　),栈底元素是(　　)。

3. 在操作序列 enqueue(1),enqueue(2),dequeue(),enqueue(5),enqueue(7),dequeue()和 enqueue(9)之后,队头元素是(　　),队尾元素是(　　)。

4. 实现递归函数调用的一种数据结构是(　　)。

5. 在具有 n 个单元的循环队列中,队满时共有(　　)个元素。

二、选择题

1. 如果一个栈的入栈序列是 1、2、3、4、5,则栈不可能的输出序列是(　　)。

(A) 54321　　(B) 45321　　(C) 43512　　(D) 12345

2. 如果一个队列的入队顺序是 1、2、3、4,则队列的输出顺序是(　　)。

(A) 4321　　(B) 1234　　(C) 1432　　(D) 3241

3. 在解决计算机主机与打印机之间速度不匹配问题时通常设置一个打印缓冲区，该缓冲区应该是一个(　　)结构。

(A) 栈　　(B) 队列　　(C) 数组　　(D) 线性表

4. 如果要从栈顶指针为 top 的链栈中删除一个节点，用 e 保存被删除节点的值，则执行(　　)。

(A) e=top; top=top->next;

(B) e=top->data;

(C) top=top->next; e=top->data;

(D) e=top->data; top=top->next;

5. 设栈 S 和队列 Q 的初始状态为空，元素 a_1，a_2，…，a_6 依次通过栈 S，一个元素出栈后即进入队列 Q。如果 6 个元素出队的顺序是 a_2、a_4、a_3、a_6、a_5、a_1，则栈 S 的容量至少应该是(　　)。

(A) 6　　(B) 4　　(C) 3　　(D) 2

三、问答题

1. 循环队列的引入是为了克服什么问题？

2. 设有一个栈，元素入栈的次序为 a、b、c、d、e，能否得到出栈序列 c、e、a、b、d 和 c、b、a、d、e？如果能，则写出操作序列；如果不能，则说明原因。

3. 如果用 Q[n]表示一个循环队列，用 front 表示队头元素的前一个位置，用 rear 表示队尾元素的位置，那么队列中元素个数是多少？

4. 已知一个栈的入栈序列是 1，2，…，n，其输出序列为 p_1，p_2，…，p_n，如果 $p_1=n$，则 p_i 是多少？

5. 如果利用两个栈 S1 和 S2 模拟一个队列，那么应该如何利用栈的运算实现队列的插入和删除操作？

四、算法设计题

1. 试设计一个算法：判定给定的字符向量是否为回文。回文即正读和反读均相同的字符序列，例如字符序列“abba”和字符序列“abdba”均是回文，但字符序列“good”不是回文。提示：将一半字符入栈。

2. 试设计一个算法：判断一个算术表达式的圆括号是否正确配对。提示：采用栈的结构，对表达式进行扫描时，凡遇到字符“(”就入栈，遇到字符“)”就退栈，表达式扫描完毕后栈应该为空。

3. 假设以带头节点的循环链表表示队列，且只设一个指针指向队尾元素。试设计一个算法：对该循环队列进行置空队、判队空、入队和出队操作。

4. 假设顺序栈 S 中有 2n 个元素，从栈顶到栈底的元素依次为 a_{2n}，a_{2n-1}，…，a_1。试设计一个算法：通过一个循环队列 Q 重新排列该栈中的元素，使得从栈顶到栈底的元素依次为 a_{2n}，a_{2n-2}，…，a_2，a_{2n-1}，a_{2n-3}，…，a_1。要求空间复杂度和时间复杂度均为 O(n)。

5. 假设编号为 1、2、3、4 的 4 列火车通过如图 3-2 所示的铁路调度站，可能得到的调度结果有哪些？如果有 n 列火车通过调度站，试设计一个算法：输出所有可能的调度结果。