第3章

栈、队列、数组

本章为栈和队列部分，首先给出本章学习目标、知识点导图，使读者对本章内容有整体了解；接着，介绍栈和队列；然后，围绕栈给出栈的定义、栈的存储结构（顺序栈、链栈、共享栈）、不同存储结构的基本操作以及栈的常见应用；之后，围绕队列给出队列的定义、队列的存储结构（循环队列、链队列、双端队列）、不同存储结构的基本操作以及队列的常见应用；最后一部分为特殊矩阵，包括对称矩阵、三角矩阵、三对角矩阵（带状矩阵）、稀疏矩阵。

本教材在每章各个需要讲解的部分配有微课视频和配套课件，读者可根据需要扫描对应部分的二维码获取；同时，考研真题部分也适当配有真题解析微课讲解，可根据需求扫描对应的二维码获取。


栈、队列、数组（一）


栈、队列、数组（二）

3.1 本章学习目标

（1）学习栈和队列的基本概念。

（2）掌握栈的存储结构（顺序栈、链栈、共享栈）、不同存储结构的基本操作。

（3）掌握队列的存储结构（循环队列、链队列、双端队列）、不同存储结构的基本操作。

（4）掌握栈和队列的常见应用。

（5）学习多维数组的存储和特殊矩阵的压缩存储。

3.2 知识点导图

栈、队列、数组知识点导图如图 3-1 所示。

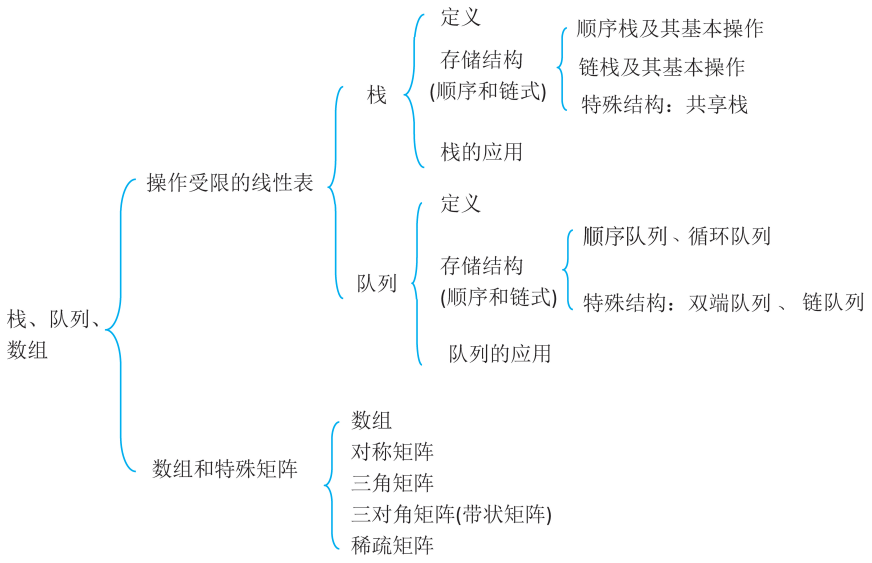


图 3-1 知识点导图

3.3 知识点归纳

3.3.1 栈

1. 栈概述

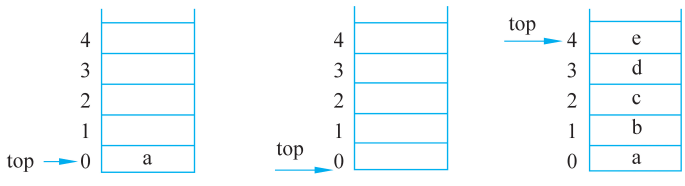
3.3.1 栈的概述

1) 栈的定义

栈(stack)是限定仅在表尾进行插入或删除操作的线性表。因此,对栈来说,表尾端有其特殊含义,称为**栈顶**(top),相应地,表头端称为**栈底**(bottom)。不含元素的空表称为**空栈**。

向栈顶插入元素的操作常称为“**入栈**”,删除栈顶元素的操作称为“**出栈**”。

如图 3-2 所示,分别为栈中只有一个元素、栈空和栈满状态(这里假定栈顶指针指向栈顶元素,要注意栈顶指针是指向栈顶元素还是栈顶元素的下一个位置)。由于在栈顶进行插入和删除操作,因此对应于图 3.2(c)栈满,出栈序列为 e,d,c,b,a。栈中访问结点时遵循**后**



(a) 栈中只有一个元素　(b) 栈空　(c) 栈满

图 3-2 栈示意图

进先出(Last In First Out,LIFO)的原则。

2) 栈的基本操作

从线性表角度总结归纳出栈的基本操作如下。

(1) 构造一个空栈 S,其基本操作为 InitStack(&S)。

(2) 销毁栈 S,S 不再存在,其基本操作为 DestroyStack(&S)。

(3) 判断栈是否为空,若栈 S 为空栈,则返回 1,否则返回 0,其基本操作为 StackEmpty(S)。

(4) 取栈顶元素,若栈不空,则用 e 返回 S 的栈顶元素,否则提示为“栈空”,其基本操作为 GetTop(S,&e)。

(5) 入栈操作,插入元素 e 为新的栈顶元素,其基本操作为 Push(&S,e)。

(6) 出栈操作,若栈不空,则删除 S 的栈顶元素,并返回其值,否则提示为“栈空”,其基本操作为 Pop(&S,&e)。

其中,栈的存储方式包括顺序栈和链栈两种。下面分别给出不同存储结构下对应的基本操作的算法实现。

2. 存储结构

1) 顺序栈及其基本操作

3.3.1 顺序栈及其基本操作

(1) 顺序栈的定义。栈的顺序存储结构称为顺序栈,顺序栈通常由一个一维数组和一个记录栈顶元素的变量(或两个变量,一个指向栈底,另一个指向栈顶)组成。

(2) 顺序栈的实现。顺序栈的存储结构描述如下。

```
#define LIST_INIT_SIZE 100                  //初始存储空间
typedef char ElemType;                      //数据元素类型
typedef struct{
    ElemType data[LIST_INIT_SIZE];          //存放栈中元素
    int top;                                //用于栈顶指针
    //int base;                             //用于栈底指针
}SqStack;                                   //顺序栈类型
```

由于栈在使用过程中所需的大小空间很难估计,一般在初始化时不应限定栈的最大容量,较合理的做法是先为栈分配基本容量,在应用过程中,当空间不足时再扩大,可再设一个 STACKINCREMENT(存储空间分配增量)。

(3) 顺序栈的基本操作算法实现。需要特别说明的是,这里假定栈顶指针指向栈顶元素,要注意栈顶指针是指向栈顶元素还是栈顶元素的下一个位置。顺序栈的存储结构示意图如图 3-2 所示。

① **初始化栈**。构造一个空栈 S,其基本操作为 InitSqStack(SqStack &S)。

```
void InitSqStack(SqStack &S){
    S.top=-1;
}
```

② **判断栈是否为空**。若栈 S 为空栈,则返回 1,否则返回 0,其基本操作为 StackEmpty(SqStack S)。

```
int SqStackEmpty(SqStack S){
    if(S.top!=-1)
        return 0;                          //非空
    else
        return 1;                          //空
}
```

③ **取栈顶元素**。若栈不空,则用 e 返回 S 的栈顶元素,否则返回 FALSE,其基本操作为 GetTop(SqStack S,ElemType &e)。

```
bool GetTop(SqStack S,ElemType &e){
    if(S.top==-1)
        return FALSE;                      //栈空
    else
        e=S.data[top];                     //非空,用 e 返回 S 的栈顶元素
    return TRUE;
}
```

④ **入栈操作**。插入元素 e 为新的栈顶元素,其基本操作为 Push(SqStack &S,ElemType e)。

```
bool Push(SqStack &S, ElemType e){
    if(S.top== LIST_INIT_SIZE-1)           //判满
        return FALSE;
    else
        S.data[++top]=e;                   //先自增,再入栈
    return TRUE;
}
```

⑤ **出栈操作**。若栈不空,则删除 S 的栈顶元素,并返回其值,否则返回 FALSE,其基本操作为 Pop(SqStack &S, ElemType &e)。

```
bool Pop(SqStack &S, ElemType &e){
    if(S.top== -1)                         //判空
        return FALSE;
    else
        e= S.data[top--];                  //先出栈,再自减
    return TRUE;
}
```

2) 链栈及其基本操作

3.3.1 链栈及其基本操作

(1) 链栈的定义。采用链式存储的栈称为**链栈**。链栈示意图如图 3-3 所示。通常采用单链表实现,不存在溢出问题,所有操作均在表头进行。注意,链栈中指针的方向。

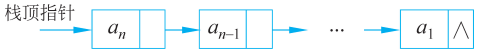


图 3-3　栈的链式(链栈)存储示意图

(2) 链栈的实现。栈的链式存储结构描述如下。

```
typedef char ElemType;                          //数据元素类型
typedef struct StackNode
{
    ElemType data;                              //数据域
    struct StackNode * next;                    //指针域
}StackNode;                                     //结点类型
typedef struct {
    struct StackNode *  top;                    //栈顶指针
    int   length;                               //栈中元素个数
} LStack;                                       //链栈类型
```

(3) 链栈的基本操作算法实现。

栈的链式存储上的操作与链表相似,其入栈和出栈对应于链表的插入和删除,需要注意的是均在表头进行。栈的链式存储便于插入和删除。这里,链栈可以带头结点也可以不带头结点,审题时需要注意,具体操作的实现会有所不同。以下基本操作假定不带头结点。

① **初始化栈**。构造一个空栈 S,其基本操作为 InitLStack(LStack &S)。

```
void InitLStack(LStack &S){
    //构造一个空栈 S
    S.top = NULL;
    //设栈顶指针的初值为 "空"
    S.length = 0;                               //空栈中元素的个数为 0
}
```

② **入栈操作**。插入元素 e 为新的栈顶元素,其基本操作为 Push(LStack &S, ElemType e)。

```
bool Push(LStack &S, ElemType e){
                                                //在栈顶之上插入元素 e,e 为新的栈顶元素
    p=new LNode;                                //建新的结点
    if(!p) return FALSE;                        //存储分配失败
    p->data=e;
    p->next=S.top;                              //链接到原来的栈顶
    S.top = p;                                  //移动栈顶指针
    ++S.length;                                 //栈的长度增 1
}
```

③ **出栈操作**。若栈不空,则删除 S 的栈顶元素,并返回其值,否则返回 FALSE,其基本操作为 Pop(LStack &S, ElemType &e)。

```
bool Pop ( LStack &S, ElemType &e ) {
if (!S.top)                                     //判空
            return FALSE;
    else {
        e=S.top->data;                          //返回栈顶元素
        q=S.top;
        S.top=S.top -> next;                    //修改栈顶指针
```

```
            --S.length;                          //栈的长度减 1
            delete q;                            //释放被删除的结点空间
            return TRUE;
        }
    }
```

3) 共享栈

3.3.1 共享栈

由于栈只有一端的地址随入栈操作动态变化,而另外一端不变,这一特点可以让两个栈共享一个连续空间,如图 3-4 所示,两个栈的栈底分别位于这段连续空间的两端,在执行入栈操作时,两个栈顶均向连续空间的中间位置移动,出栈操作时,栈顶则向连续空间的两端移动。

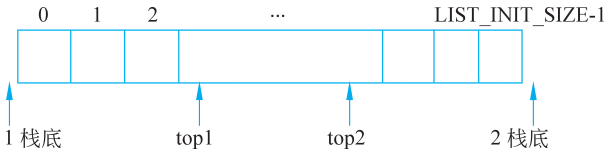


图 3-4 共享栈示意图

判空条件:top1==-1,top2== LIST_INIT_SIZE。

判满条件:top2-top1==1。

入栈:top1 先自增再赋值,top2 先自减再赋值。

出栈:top1 先得到值再自减,top2 先得到值再自增。

优点:两个栈的数据元素个数之和不大于连续的存储空间即可执行入栈操作;在满足入栈条件的前提下,两个栈共享空间,且两个栈的最大长度可变,空间利用率更高。

3. 栈的应用

3.3.1 栈的应用数制转换和括号匹配应用

1) 数制转换的应用

以十进制数转换为八进制数为例,给出进制转换的算法,其他进制转换与其相似。基本的算法思路为:即将需要转换的数除以 8,得到余数,余数的倒序输出即为所求结果,即对应的八进制数,因需倒序输出,故可采用栈作为辅助存储结构。

```
void func(int num){
    SqStack S;
InitSqStack(S);
int temp;
```

```
while(num)
{
    temp=num%8;
    Push(S,temp);
    num=num/8;
}
while(!SqStackEmpty(S))
{
    Pop(S,temp);
    printf("%d",temp);
}
}
```

2）括号匹配的应用

表达式中可能包含多种类型的括号，如()、[]、{}等，需要对应地去匹配。

基本思想为：每扫描一个左括号，将其入栈。若扫描到右括号，则取栈顶元素，判断其是否与其匹配，匹配即可出栈；若不匹配则为不合法情况，程序结束。若是左括号，则入栈，以此类推。最后，当算法结束时，栈为空，则序列括号匹配，否则为序列括号不匹配。

3）表达式求值的应用

3.3.1 栈的应用表达式求值的应用

（1）**中缀表达式转后缀表达式的方法**。中缀表达式需要关注运算符的优先级和括号，后缀表达式的运算符在操作数的后面、无括号，下面以一个例子说明中缀表达式转为后缀表达式和由后缀表达式求值的过程。

例题，将中缀表达式 $a+b/(c+d)-e*f$ 转换为后缀表达式（手动和计算机计算）。

① 方法一：手动模拟方法。

第一步，首先需要确定中缀表达式中各运算符的运算顺序：

$$a+b/(c+d)-e*f$$

顺序标号　　3 2　1　5 4

这里，可以观察到，运算符排序可以不唯一，可约定采用“靠左”的运算符排在前的原则，如果左边的运算符能先计算就将左边的运算符排在相对右边的运算符前面计算。

第二步，选择计算的运算符，组合为“左操作数 右操作数 运算符”后可作为新的操作数，直至所有运算符均处理完毕。

过程 1 为：$cd+$作为一个整体操作数，即 $c+d$ 的结果为新的操作数，变为 $a+b/$“$cd+$”$-e*f$。

过程 2 为：$bcd+/$作为一个整体操作数，即 $b/(c+d)$的结果为新的操作数，变为 $a+$“$bcd+/$”$-e*f$。

过程 3 为：$abcd+/+$作为一个整体操作数，即 $a+b/(c+d)$的结果为新的操作数，变为“$abcd+/+$”$-e*f$。

过程 4 为：$ef*$ 作为一个整体操作数，即 $e*f$ 的结果为新的操作数，变为“$abcd+/+$”—“$ef*$”。

过程 5 为：$abcd+/+ef*-$ 作为一个整体操作数，即“$abcd+/+$”—“$ef*$”的结果为最后结果，因此得到后缀表达式 $abcd+/+ef*-$。

② 方法二：计算机计算方法。设置运算符栈。

初始化栈，用于保存暂时不能确定运算顺序的运算符。从左至右依次处理如下。

第一步，当遇到操作数时，则加入后缀表达式。

第二步，当遇到括号时，左括号入栈，右括号依次弹出栈内运算符，加入到后缀表达式，直至弹出左括号，并且需要注意的是括号的匹配性和括号不加入到后缀表达式中。

第三步，当遇到操作符时，弹出运算符栈中优先级高于或等于当前运算符的所有运算符，同时需要加入到后缀表达式中，如果碰到左括号或栈空则停止，然后当前运算符入栈，最后可以认为有一个优先级最低的运算符 ♯（为了清空运算符栈），不需要入栈。其中，关于优先级，“*”和“/”高于“+”和“—”，“*”和“/”相等，“+”和“—”相等。可自行采用上述原则模拟计算机重新计算后缀表达式。

以 $a+b/(c+d)-e*f$ 为例，过程为：遇到 a 加入后缀表达式，遇到“+”入栈，遇到 b 加入后缀表达式，遇到“/”入栈，遇到“(”入栈，遇到 c 加入后缀表达式，遇到“+”入栈，遇到 d 加入后缀表达式，遇到“)”后“+”出栈并加入后缀表达式，“(”出栈，此时的后缀表达式为 $abcd+$，栈中运算符为“+”和“/”。接着，遇到“—”，依次弹出“/”和“+”并加入后缀表达式，“—”入栈，遇到 e 加入后缀表达式，遇到“*”入栈，此时栈中元素为“—”和“*”，遇到 f 加入后缀表达式，最后栈中运算符与“♯”比较优先级，依次弹出“*”和“—”，则最后的后缀表达式可得为 $abcd+/+\ ef*-$。

(2) **后缀表达式的计算方法**。分为手动模拟和计算机计算两种方法。

① 方法一：手动模拟方法。从左至右依次扫描，遇到运算符时，取出运算符前面的两个操作数执行运算，合为一个操作数，需要注意两个操作数的左右顺序。

例如，$abcd+/+\ ef*-$，先计算 $c+d$，再计算 $b/(c+d)$，然后 $a+b/(c+d)$，再计算 $e*f$，最后 $a+b/(c+d)-e*f$。

② 方法二：计算机计算方法。设置操作数栈。

第一步，从左至右依次扫描，若遇到操作数则入栈，否则执行第二步。

第二步，若为运算符，则依次弹出栈顶两个元素，执行运算，结果重新入栈，这里需要注意弹出的操作数的顺序，先弹出的为右操作数，后弹出的为左操作数。

(3) **中缀表达式的计算**。手动模拟方法较简单，可自行完成，本部分主要介绍计算机计算方法。需要操作数栈和运算符栈。

第一步，若扫描到操作数，则入操作数栈。

第二步，扫描到运算符或括号，则按照中缀转为后缀的方法压入运算符，每当弹出运算符时，同时需要弹出两个栈顶的操作数，执行运算后再将结果作为操作数压入操作数栈，直至结束。

(4) **中缀表达式转前缀表达式**。

第一步，确定中缀表达式中运算符的顺序，采用靠右原则，如果右边的运算符能先计算就将右边的运算符排在相对左边的运算符前面计算。

第二步，按顺序选择运算符，以“运算符 左操作数 右操作数”的组合计算作为一个新的

操作数整体，继续处理，直到没有新的操作数为止。例如：

$$a + b / (c + d) - e * f$$

顺序标号：　　5　3　　2　　4　1

过程为：首先，$*ef$ 为一个整体，作为新的操作数；然后 $+cd$ 为一个整体，作为新的操作数；接着是 $/b+cd$，再接着是 $-/b+cd*ef$，最后得到 $+a-/b+cd*ef$。

(5) **前缀表达式的计算**。通过栈(存放操作数)实现前缀表达式的计算。

从右往左扫描，当元素为操作数时，入栈，继续扫描，当元素为运算符时，弹出两个栈顶元素，执行运算符的运算，结果再入栈，继续扫描，分类处理，直至结束。

栈的应用除了以上提及的 3 类应用外，还有在递归方面的应用，可查阅程序设计语言(如 C 语言程序设计)相关书籍，这里不再赘述。

3.3.2　队列

1. 队列概述

3.3.2 队列概述

1) 队列的定义

队列(queue)是限定只能在表的一端进行插入并在另一端进行删除操作的线性表。在表中，允许插入的一端称为“**队列尾**(rear)”，允许删除的另一端称为“**队列头**(front)”。队列示意图如图 3-5 所示。当队列中没有任何元素时，称为**队空**。

图 3-5　队列示意图

与栈相反，队列是一种**先进先出**(First In First Out，FIFO)的线性表。这和我们日常生活中的排队是一致的，最早进入队列的元素最早离开。

2) 队列的基本操作

总结归纳出队列的基本操作如下。

(1) 构造一个空队列 Q，其基本操作为 InitQueue(&Q)。

(2) 销毁队列 Q，Q 不再存在，其基本操作为 DestroyQueue(&Q)。

(3) 判断队列是否为空，若队列 Q 为空队列，则返回 1，否则返回 0，可由调用它的函数根据返回值判断当前队列的状态，其基本操作为 QueueEmpty(Q)；

(4) 求队列的长度，其基本操作为 QueueLength(Q)。

(5) 取队头元素，若队列不为空，则用 e 返回 Q 的队头元素，否则提示为“队空”，其基本操作为 GetHead(Q,&e)。

(6) 入队操作，入队一个元素 e 为 Q 的新的元素，其基本操作为 EnQueue(&Q, e)。

(7) 出队操作，若队列不空，则出队一个元素，用 e 返回其值，其基本操作为 DeQueue(&Q,&e)。

其中，队列的存储方式有顺序队列和链队列。下面分别给出不同存储结构下对应的基本操作的算法实现。

2. 存储结构

1) 顺序队列

3.3.2 顺序队列及其基本操作

(1) 顺序队列的定义。顺序队列利用地址连续的存储空间,依次存放从队头到队尾的所有元素,即队列的顺序存储。顺序队列由一个一维数组、两个分别指示队头和队尾的变量组成,分别称为队头指针和队尾指针。这里,可以有两种方式:队头指针指向队头元素,队尾指针指向队尾元素的下一个位置;或者队头指针指向队头元素,队尾指针指向队尾元素。需要注意二者的区别。

(2) 顺序队列的实现。队列的顺序存储结构描述如下。

```
#define MaxSize  100
typedef struct {
    ElemType elem[MaxSize];                    //存储队列元素
    int rear;                                  //队尾指针
    int front;                                 //队头指针
}SqQueue;
```

(3) 顺序队列的基本操作。如图 3-6 所示为空队列和当前队列状态,在非空队列中,队头指针始终指向队头元素,而队尾指针指向队尾元素的"下一个"位置。

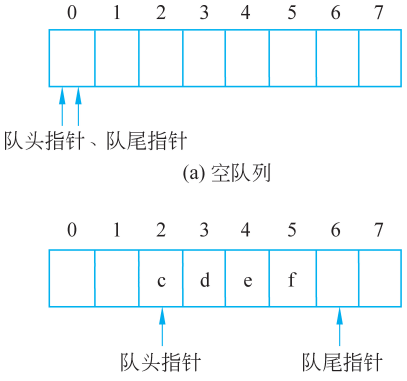


图 3-6 顺序队列示意图

初始状态 Q.front==Q.rear&&Q.front==0,入队操作时,由于头指针始终指向队头元素,而尾指针指向队尾元素的"下一个"位置,因此需要先入队,再执行 Q.rear++;出队操作时,需要先获取队头,再执行 Q.front++。

2) 循环队列及其基本操作

3.3.2 循环队列及其基本操作

（1）循环队列的定义。如图 3-6 所示，如果在这之后又有 g、h 入队列，而队列中的 c、d 出队列，此时队头指针指向 e，队尾指针则指到数组外面的位置，即指针越界。当下一个元素再入队时，入队操作无法进行，但是队列空间并未满，我们称这种现象为假溢出。

由此，可设想数组存储空间是个“环”，7 后的下一个位置是 0，首尾相接，由此引入了循环队列的概念，如图 3-7 和图 3-8 所示。

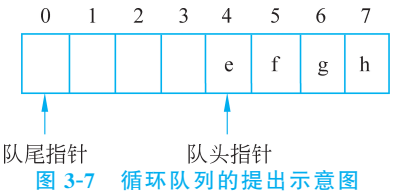


图 3-7　循环队列的提出示意图

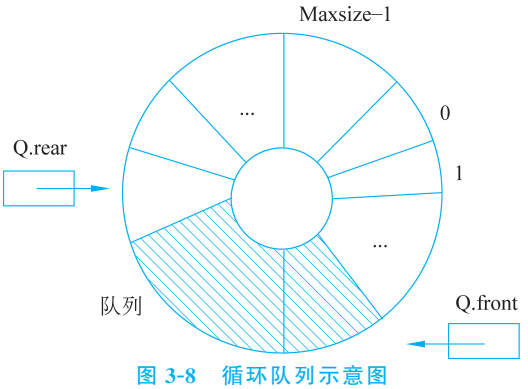


图 3-8　循环队列示意图

一般对于循环队列有 3 种设定方式，需要注意不同设定方式时队列空和队列满的判定条件。

第一种：为通常采用的方法，牺牲一个单元。约定在入队时少用一个单元，则当队头指针在队尾指针的下一位置时队列为满。在该种方法下，队列判空的条件为 Q.front==Q.rear，队列判满的条件为(Q.rear+1)%MaxSize==Q.front，计算当前队列中的元素个数为(Q.rear−Q.front+MaxSize)%MaxSize。

第二种：增加标记变量 tag，并约定每次删除成功时，tag=0；每次插入成功时，tag=1。这样 tag=0 时，若因删除导致 Q.front==Q.rear，则队列为空；tag=1 时，若因插入导致 Q.front==Q.rear，则队列为满。

第三种：类型中增设表示元素个数的数据成员，可以区分队列满还是队列空。则队列空的条件为 Q.length==0，队列满的条件为 Q.length==MaxSize。

（2）循环队列的实现。循环队列的存储结构描述如下。

```
#define MaxSize  100
typedef struct {
    ElemType *elem;                          //存储队列元素
    int rear;                                //队尾指针
    int front;                               //队头指针
}SqQueue;
```

（3）循环队列的基本操作算法实现。假设在非空队列中，头指针始终指向队头元素，而尾指针指向队尾元素的“下一个”位置。

① **初始化队列**。构造一个空队列 Q，其基本操作为 InitQueue(SqQueue &Q)。

```
void InitQueue(SqQueue &Q)
{
  Q.elem = new ElemType[MaxSize];
  //分配存储空间
```

```
    if (!Q.elem) exit(1);                          //分配失败
    Q.front=Q.rear=0;
}
```

② **求队列长度**。返回队列 Q 中元素的个数,即队列的长度,其基本操作为 QueueLength(SqQueue Q)。

```
int  QueueLength(SqQueue Q) {
    return ((Q.rear-Q.front+MaxSize)% MaxSize);
}
```

③ **入队操作**。若队列不满,入队一个元素 e,e 为 Q 的新的元素,其基本操作为 EnQueue(SqQueue &Q,ElemType e)。

```
bool EnQueue(SqQueue &Q,ElemType e) {
    if ((Q.rear + 1) % MaxSize==Q.front )
          return FALSE;
    Q.elem[Q.rear] = e;
    Q.rear = (Q.rear+1) % MaxSize;
    return TRUE;
}
```

④ **出队操作**。若队列不空,则出队一个元素,用 e 返回其值,其基本操作为 DeQueue (SqQueue &Q, ElemType &e)。

```
bool DeQueue (SqQueue &Q, ElemType &e) {
    if (Q.front == Q.rear)
              return FALSE;
    e = Q.elem[Q.front];
    Q.front = (Q.front+1) % MaxSize;
    return TRUE;
}//DeQueue
```

3) 链队列及其基本操作

3.3.2 链队列及其基本操作

(1) 链队列的定义。队列的链式存储结构称为**链队列**。它是一个带有队头指针和队尾指针的单链表,队头指针指向队头结点,队尾指针指向队尾结点,即单链表的最后一个结点。队头指针和队尾指针结合起来构成链队结点,如图 3-9 所示。注意示意图中链队列带头结点。通常为方便处理,链队列设置头结点。

(2) 链队列的实现。队列的链式存储结构描述如下。

```
typedef struct QNode{
    ElemType data;                                 //数据域
    struct QNode * next;                           //指针域
    } QNode;                                       //链队列结点类型
typedef struct{
```

```
    QNode * front;                          //指向队列头
    QNode * rear;                           //指向队列尾
} LinkQueue;                                //链队列类型
```

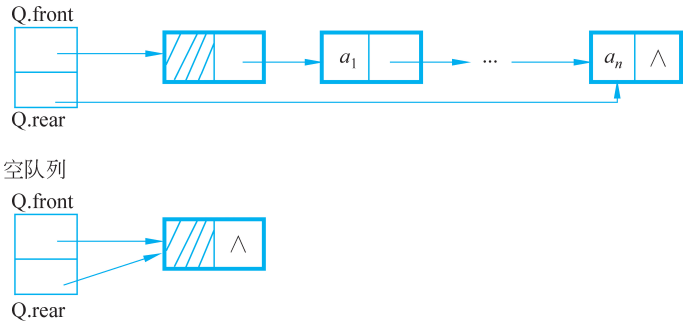


图 3-9　链队列示意图

(3) 链队列的基本操作算法实现。需要特别说明的是,为方便处理,这里假定的链队列为带有头结点的链队列。不带头结点的链队列相关操作可参看本章重难点解析部分。

① **初始化队列**。构造一个空队列 Q,其基本操作为 InitLinkQueue(LinkQueue &Q)。

```
InitLinkQueue(LinkQueue  &Q)
{ //构造一个空队列 Q
    Q.front=Q.rear=new QNode;
    if (!Q.front)
        exit(1);                            //存储空间分配失败
    Q.front->next=NULL;
}
```

② **判断队列是否为空**。若队列 Q 为空队列,则返回 1,否则返回 0,可由调用它的函数根据返回值判断当前队列的状态,其基本操作为 QueueLinkEmpty(LinkQueue &Q)。

```
int QueueLinkEmpty(LinkQueue &Q){
    if(Q.front==Q.rear)
        return 1;                           //队列空
    else
        return 0;                           //队列非空
}
```

③ **入队操作**。入队一个元素 e 为 Q 的新的元素,其基本操作为 EnLinkQueue(LinkQueue &Q, ElemType e)。

```
void EnLinkQueue(LinkQueue &Q,ElemType e) {
    s = new QNode;
    if (!s) exit(1);                        //存储空间分配失败
    s->data=e;   s->next = NULL;
    Q.rear->next=s;                         //修改尾结点指针
    Q.rear=s;                               //移动队尾指针
}
```

④ **出队操作**,若队列不空,则出队一个元素,用 e 返回其值,其基本操作为 DeLinkQueue(LinkQueue &Q, ElemType &e)。

```
bool DeLinkQueue(LinkQueue &Q, ElemType &e) {
    QNode * p;
    //若队列不空,则删除当前队头元素
    if(Q.front==Q.rear)//链队列中只有一个头结点
        return FALSE;
    p=Q.front->next;
    e=p->data;                              //返回被删元素
    Q.front->next=p->next;                  //修改头结点指针
    if(Q.rear==p)
        Q.rear=Q.front;                     //修改队尾指针
    delete p;                               //释放被删结点
    return TRUE;
}
```

4) 双端队列

3.3.2 双端队列

双端队列指队列的两端均可以进行入队和出队的特殊队列,如图 3-10 所示。

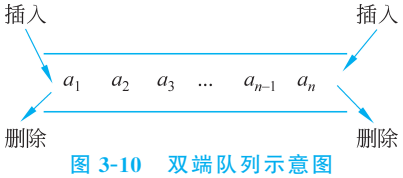


图 3-10 双端队列示意图

一般常见的还有输出受限的双端队列和输入受限的双端队列,如图 3-11 和图 3-12 所示。分别为一端可进行插入和删除,另一端只允许插入的双端队列;以及一端可进行插入和删除,另一端只允许删除的双端队列。如果再限定从一端插入的元素只能在这一端删除,则双端队列就变为两个栈底相邻的栈。

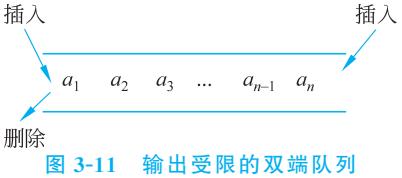


图 3-11 输出受限的双端队列

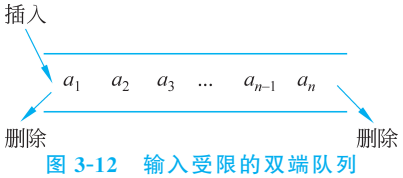


图 3-12 输入受限的双端队列

3. 队列的应用

3.3.2 队列的应用

1) 队列在计算机系统中的应用

例如,为解决计算机主机与打印机之间速度不匹配问题,通常设置一个打印数据缓冲区,

主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。再如，操作系统中多用户引起的资源竞争问题，需要时可查阅相关资料，这里仅简要提及，不再赘述。

2）队列在树的层次遍历和在图的搜索中的应用

队列在树的层次遍历和在图的搜索中均有应用，可参见后续树和图相关章节的具体内容，这里不再赘述。

3.3.3　数组和特殊矩阵

1. 数组

3.3.3 数组

矩阵是在科学工程计算中较常见的数学模型之一，数据结构中经常考虑如何使用较小的存储空间存放一组数据，如将 $m \times n$ 矩阵（m 行 n 列）更有效地存储在内存中，并且能够方便地获取元素。由于计算机的存储空间是线性的，因此通常程序设计语言使用一维数组存放二维的矩阵。这里，数组的概念可查阅相关程序设计语言教材，这里不再赘述。本部分主要关注数组的存储。

数组类型不做插入和删除的操作，因此只需通过顺序映像得到它的存储结构，可以借助数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系。

一个数组的所有元素在内存中占用一段连续的存储空间，一维数组的存储结构关系以 $a[n]$ 为例：$\mathrm{loc}(a_i)=\mathrm{loc}(a_0)+i\times \mathrm{sizeof}(\mathrm{ElemType})(n>i\geqslant 0)$。

注意数组的下标范围为 $n>i\geqslant 0$，ElemType 为数组元素类型，sizeof(ElemType)为数组元素所占用的存储单元。

1）多维数组的存储

通常有两种映像方法，即“按行优先”展开的映像方法和“按列优先”展开的映像方法。以下以二维数组为例说明两种方法。

（1）**按行优先**。行为主序，先行后列，将数组元素按行优先关系排列，先存储行号较小的元素，行号相等时先存储列号较小的元素，即同一行中的元素以列下标次序排列，如图 3-13 所示，第 $i+1$ 行的元素紧跟在第 i 行元素的后面。

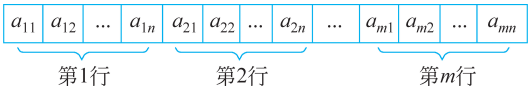


图 3-13　按行优先

（2）**按列优先**。列为主序，先列后行，将数组元素按列优先关系排列，先存储列号较小的元素，列号相等的时候先存储行号较小的元素，即同一列中的元素以行下标次序排列，如图 3-14 所示，第 $i+1$ 列的元素紧跟在第 i 列元素的后面。

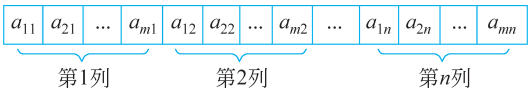


图 3-14　按列优先

2) 多维数组存储地址的计算

设每个数组元素需要 sizeof(ElemType)个存储单元存放;并记第一个数组元素的存储地址为 $\mathrm{loc}(a_{11})$,即存储区间的起始地址,也称为数组的基地址。

(1) **行为主序优先存储**。a_{ij} 是第 i 行第 j 列的数组元素,其前面已存放 $i-1$ 行共 $(i-1)\times n$ 个元素,在第 j 行中前面已经存放了 $j-1$ 个元素。

地址计算公式为

$$\mathrm{loc}(a_{ij})=\mathrm{loc}(a_{11})+((i-1)\times n+j-1)\times \mathrm{sizeof(ElemType)}$$

或者按照 C 语言数组下标,即

$$\mathrm{loc}(a_{ij})=\mathrm{loc}(a_{00})+(i\times n+j)\times \mathrm{sizeof(ElemType)}$$

(2) **列为主序优先存储**。以列为主序优先存储的地址计算公式为

$$\mathrm{loc}(a_{ij})=\mathrm{loc}(a_{11})+((j-1)\times m+i-1)\times \mathrm{sizeof(ElemType)}$$

或者按照 C 语言数组下标,即

$$\mathrm{loc}(a_{ij})=\mathrm{loc}(a_{00})+(j\times m+i)\times \mathrm{sizeof(ElemType)}$$

2. 特殊矩阵的压缩存储

3.3.3 特殊矩阵的压缩存储

对于 $m\times n$ 阶矩阵,当 m 和 n 较大时,需要占用大量的连续存储空间。通常用二维数组表示矩阵,则矩阵中的元素均可在二维数组中找到对应的存储位置。一些有下列特性的高阶矩阵,其元素值的分布具有规律性,矩阵中有很多值相同的元素或零值元素,为了节省存储空间,需要对它们进行“压缩存储”,不存或者少存这些值相同的元或零值元素,以降低矩阵对存储容量的需求。

常见的特殊矩阵包括对称矩阵、三角矩阵(上三角矩阵和下三角矩阵)和三对角矩阵(带状矩阵)等。

1) 对称矩阵

n 阶方阵,满足 $a_{ij}=a_{ji}$,其中 $n\geqslant i\geqslant 1$,$n\geqslant j\geqslant 1$,存储于二维数组 $a[n][n]$ 中,这种方阵被称为**对称矩阵**。可以将方阵划分为 3 个区域,分别为上三角区、主对角线和下三角区。如图 3-15 所示,以虚线为界分为 3 个区域。

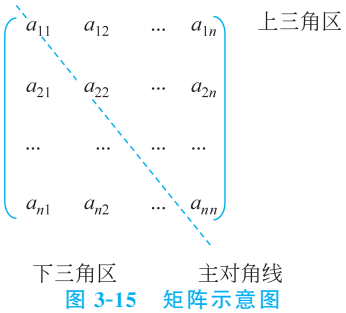


图 3-15 矩阵示意图

其中,上三角区元素 $i<j$,主对角线上元素 $i=j$,下三角区元素 $i>j$。因对称矩阵的上三角区元素与下三角区元素相同,因此可只存放上三角区和主对角元素或者下三角区和主对角元素在长度为 $n(n+1)/2$ 的一维数组中。

以只存放下三角部分为例:第一行存放 1 个元素,第二行 2 个元素,…,第 $i-1$ 行 $i-1$ 个元素,第 i 行 $j-1$ 个元素,则假设 a_{ij} 在一维数组中的下标为 k(下标从 0 开始),则 $k=1+2+\cdots+i-1+j-1=i(i-1)/2+j-1$。

得到下标对应关系如下:

$$k=\begin{cases} i\times(i-1)/2+j-1 & (i\geqslant j)\text{下三角区和主对角线元素} \\ j\times(j-1)/2+i-1 & (i<j)\text{上三角区元素} \end{cases}$$

2）三角矩阵

上三角矩阵，指矩阵的主对角线及以下的所有元素均为同一常数或 0 的 n 阶矩阵。

上三角矩阵的压缩存储方法。如图 3-16 所示，对处于下三角（不含主对角线）的常数，为其在最后分配一个存储空间；对含主对角线的处于上三角的每个元素按行为主序优先存储方法依次顺序存储分配空间，需要 $n\times(n+1)/2$ 个元素空间，加上最后分配的存储空间，共需要 $n\times(n+1)/2+1$ 个元素空间。

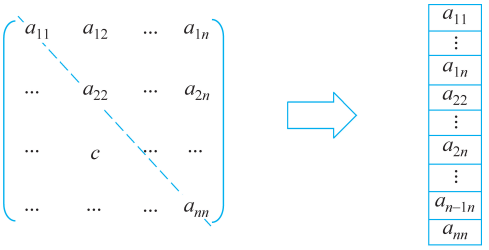


图 3-16　上三角矩阵的压缩存储示意图

得到下标对应关系如下：

$$k=\begin{cases} (i-1)\times(2n-i+2)/2+j-i & (i\leqslant j)\text{上三角区和主对角线元素} \\ n\times(n+1)/2 & (i>j)\text{下三角区元素} \end{cases}$$

注意，下标从 0 开始。

下三角矩阵，矩阵的主对角线以上的所有元素均为同一常数或 0 的 n 阶矩阵。

下三角矩阵的压缩存储方法。如图 3-17 所示，处在下三角的元素 a_{ij}，其前 $i-1$ 行共有 $i\times(i-1)/2$ 个元素，在第 i 行它处于第 j 个位置，则 a_{ij} 处于存储分配区的第 $i\times(i-1)/2+j$ 个。

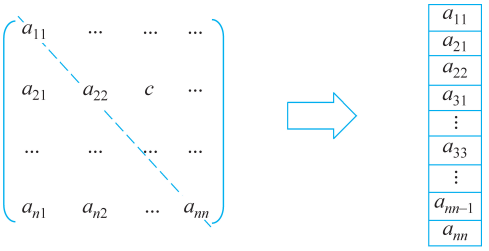


图 3-17　下三角矩阵的压缩存储示意图

有

$$k=\begin{cases} i\times(i-1)/2+j-1 & (i\geqslant j)\text{下三角区和主对角线元素} \\ n\times(n+1)/2 & (i<j)\text{上三角区元素} \end{cases}$$

注意，下标从 0 开始。

3）三对角矩阵（带状矩阵）

三对角矩阵是除主对角线及主对角线上下各一个元素外，其余元素都为零的矩阵。

对三对角矩阵的压缩存储方法是，按行为主序优先存储方法把非零区的三对角元素依次顺序存储到一片连续的存储空间中，如图 3-18 所示。

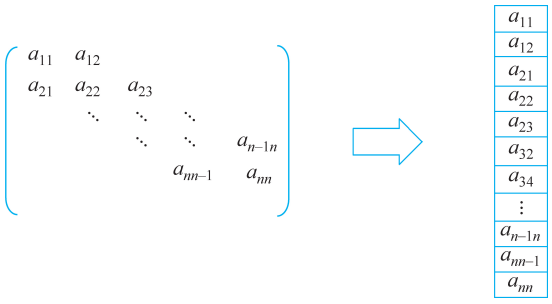


图 3-18 三对角矩阵的压缩存储示意图

由此可计算三条对角线上的元素 a_{ij}，$n \geqslant i \geqslant 1$，$n \geqslant j \geqslant 1$，在一维数组中存放的下标 $k = 2i + j - 3$，$|i - j| \leqslant 1$。注意，下标从 0 开始。

3. 稀疏矩阵

3.3.3 稀疏矩阵

矩阵中非零元素的个数相对于矩阵元素的个数来说非常少，则该矩阵被称为**稀疏矩阵**。假设一个 $m \times n$ 的稀疏矩阵，非零元素值较少，且零元素的分布无规律，用常规方法存储稀疏矩阵将浪费较多空间。可采用三元组的形式存储非零元素，三元组为“行标，列标，元素值”。三元组可以与稀疏矩阵中的非零值元一一对应，因此可用三元组顺序表来表示稀疏矩阵，并且三元组在顺序表中的元素为“以行为主”有序排列。

假设在 $m \times n$ 的矩阵中有 t 个非零值元，令

$$\delta = \frac{t}{m \times n}$$

称 δ 为矩阵的稀疏因子，则通常认定 $\delta \leqslant 0.05$ 的矩阵为稀疏矩阵。

如图 3-19 所示，为三元组举例。

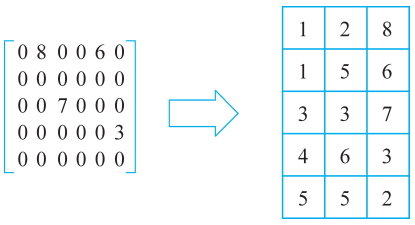


图 3-19 三元组示意图

需要注意的是，稀疏矩阵压缩存储后失去了随机存取特性。

三元组顺序表的结构定义如下。

```
#define MaxSize 100;
//假设非零元个数的最大值为 100
typedef struct {                                    //三元组结点结构
    int i, j;                                       //非零元的行号和列号
    ElemType e;                                     //非零元的值
} Triple;
typedef struct {
    Triple data[MaxSize+1];                         //非零元三元组表,data[0]未用
    int mu, nu, tu;                                 //矩阵的行数、列数和非零元的个数
} TSMatrix;                                         //三元组顺序表
```

3.4　重点和难点知识点详解

1. n 个不同元素进栈、出栈不同序列的个数计算

n 个不同元素进栈、出栈不同序列的个数为$\frac{1}{n+1}C_{2n}^{n}$，卡特兰(Catalan)数公式，请自行查阅证明方法。一般考查栈的出栈序列的合法性及出栈过程。

2. 链队列和链栈的优点

链队列和链栈不存在溢出问题，同时适合当存在多个队列和栈的情况，空间利用率较高。

3. 本章常见错误

(1) 需要特别说明的是，上述栈的基本操作假定栈顶指针指向栈顶元素，要注意栈顶指针是指向栈顶元素还是栈顶元素的下一个位置。在审题时需注意栈顶的设定，并随之更改栈的判空条件、判满条件、入栈具体操作、出栈具体操作。

(2) 链栈可以带头结点也可以不带头结点，审题时需要注意。

(3) 不带头结点的链队列注意事项。当链队列不带头结点时，判空条件为“Q.front==NULL&Q.rear==NULL;”。出队操作时，判断队列是否为空，若非空，则 Q.front 指向下一个结点，同时还需判断当前出队结点是否为最后一个结点，若是，则还需做 Q.rear=NULL 操作。入队操作时，Q.rear 指向新结点，同时还需要判断这个结点是否为当前队列中的第一个结点，若是，则还需做 Q.front 指向新结点的操作。

(4) 顺序队列的队头和队尾指向问题。队头指针指向队头，队尾指针指向队尾的下一个位置；也可队头指针指向队头，队尾指针指向队尾。二者会影响算法的设计，因此需要注意审题。

3.5　习 题 题 目

本部分习题形式包括单项选择题、综合应用题等题型，是专业课学习和考研的常见题型。知识点覆盖专业课程学习和考研知识点，因此本部分知识点相关习题设置较全面。

题目难度方面设置基础习题、进阶习题、考研真题，这样读者可根据自身情况合理安排学习规划，有针对性地逐步提升专业知识、解题能力和应试能力。

3.5.1　基础习题

一、单项选择题

1. 队列操作数据的原则是(　　)，栈操作数据的原则是(　　)。

A. 先进先出　　B. 后进先出　　C. 后进后出　　D. 无顺序

2. 栈的操作，入栈时需要判断栈是否为(　　)，出栈时需要判断栈是否为(　　)。

A. 满　　B. 空　　C. 下溢出　　D. 上溢出

3. 为解决计算机与打印机之间速度的不匹配问题，一般设置打印数据的缓冲区，主机可将要输出的数据先放入缓冲区，打印机再依次从缓冲区取出数据打印，则缓冲区的逻辑结

构为(　　)。

A. 队列　　B. 栈　　C. 图　　D. 树

4. 下列(　　)是栈的应用。

A. 表达式求值　　B. 递归调用　　C. 数制转换　　D. 以上均正确

5. 栈和队列都是(　　)。

A. 先进先出　　B. 后进先出

C. 限制存取点的线性结构　　D. 限制存取点的非线性结构

6. 假设栈和队列的初始状态均为空,元素 A、B、C、D、E、F、G 依次入栈,出栈后进入队列,现已知出队列的顺序为 B、D、C、F、E、A、G,则该栈的容量至少为(　　)。

A. 4　　B. 3　　C. 2　　D. 1

7. 链队列(链队列无头结点)在出队时需要修改的指针为(　　)。

A. 仅需修改队头指针

B. 仅需修改队尾指针

C. 队头指针、队尾指针均需修改

D. 有些情况下队头指针、队尾指针均需修改

8. 最大容量为 MaxSize 的循环队列存储在数组中(牺牲一个单元,入队时少用一个单元),则判断队列是否满的操作为(　　)。

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

C. rear + 1 ==front　　D. (rear − 1) % MaxSize==front

9. 栈的初始状态为空,A、B、C、D 4 个元素依次入栈,在所有可能出栈的序列中,以 D 开头的序列个数为(　　)。

A. 4　　B. 3　　C. 2　　D. 1

10. 用一个大小为 7 的数组实现循环队列,当前 rear=0、front=4,则出队一个元素、入队两个元素后,rear 和 front 的值分别为(　　)。

A. 5、1　　B. 1、5　　C. 4、2　　D. 2、5

11. 队列初始状态为空,入队序列为 A,B,C,D,则可能的出队序列为(　　)。

A. A,B,C,D　　B. D,C,B,A　　C. C,B,D,A　　D. A,D,C,B

12. 用带头结点的单链表表示链队列,则队尾指向链队列的(　　)。

A. 尾　　B. 中　　C. 头　　D. 以上均正确

13. 用循环单链表表示队列,其长度为 n,只设定头指针,队头在链表的表尾,则入队操作的时间复杂度为(　　)。

A. $O(1)$　　B. $O(n)$　　C. $O(n\log_2 n)$　　D. $O(n^2)$

14. 对于同一个问题,采用递归算法和非递归算法,下列说法正确的为(　　)。

A. 递归算法和非递归算法无法相比较　　B. 递归算法和非递归算法相同

C. 递归算法通常效率高　　D. 非递归算法通常效率高

15. 对稀疏矩阵进行压缩存储的目的是(　　)。

A. 降低运算的时间复杂度　　B. 节省存储空间

C. 便于运算　　D. 便于做输入输出操作