第3章 栈、队列和数组

教学重点	栈和队列的操作特性；栈和队列基本操作的实现；特殊矩阵的压缩存储方法及寻址方法
教学难点	循环队列的存储方法；循环队列中队空和队满的判定条件
教学目标	(1) 解释栈的定义及操作特性，根据入栈序列判断出栈序列是否合法； (2) 描述顺序栈和链栈存储方法，针对实际问题画出存储示意图； (3) 实现顺序栈和链栈的基本操作，比较两种存储结构的优缺点； (4) 在比较队列和栈的基础上，解释队列的定义及操作特性； (5) 辨明循环队列是顺序队列的改进，画出顺序队列和循环队列的存储示意图，描述循环队列存储方法； (6) 归纳循环队列的队空/队满判定条件，实现循环队列的基本操作； (7) 描述链队列存储方法，画出存储示意图，实现链队列的基本操作； (8) 解释(多维)数组的定义及存储方法，归纳寻址公式； (9) 描述对称矩阵、对角矩阵等特殊矩阵的压缩存储方法，归纳寻址公式； (10) 描述稀疏矩阵的顺序存储和链表存储方法，画出存储示意图
教学提示	对于栈和队列要抓住一条明线：逻辑结构→存储结构→算法实现→时间性能，从栈和队列的定义入手，在与线性表的定义和操作比较的基础上，得出栈和队列的操作特性和存储方法。注意将顺序栈与顺序表、链栈和单链表、顺序队列和循环队列、链队列和链表等进行比较。注意不要直接给出循环队列存储方法，要以提出问题、解决问题的方式逐渐引入，一方面训练学生的逻辑思维能力，另一方面深刻理解存储结构的含义。 对于(多维)数组，要把握数组的广义线性特征，基于数组的逻辑结构和操作特点，得出数组的顺序存储方法。对于特殊矩阵的压缩存储，要根据矩阵的特点设计压缩存储方法，强调特殊矩阵在压缩存储下仍然具有随机存取特性，归纳寻址公式。 本章的算法非常简单，但要求熟练掌握，在树结构和图结构中会使用栈和队列作为辅助数据结构实现遍历等复杂操作


课件 3-1

3.1 引　言

栈和队列是两种常用的数据结构,广泛应用在操作系统、编译程序等各种软件系统中。在实际问题的处理过程中,有些数据具有后到先处理或先到先处理的特点,请看下面几个例子。

例 3-1　括号匹配问题。C 语言对于算术表达式中括号的配对原则是:右括号“)”与其前面最近的尚未配对的左括号“(”配对。如何判断给定表达式中所含括号是否正确配对呢?如果顺序扫描表达式,当扫描到右括号“)”时,需要查找已经扫描过的最后一个尚未配对的左括号“(”,对于“(”具有最后扫描到最先配对的特点。那么,如何保存已经扫描过的尚未配对的左括号“(”,并对其实施配对操作呢?

例 3-2　函数的嵌套调用。函数的嵌套调用是在函数的执行过程中再调用其他函数,如图 3-1 所示,在函数 A 尚未执行结束时调用函数 B,在函数 B 尚未执行结束时又调用函数 C,那么,当函数 C 执行结束时,应该返回到什么位置呢?为保证函数嵌套调用的正确执行,系统自动设立了工作栈保存函数的调用次序。那么,系统工作栈是如何保证调用次序的正确性呢?

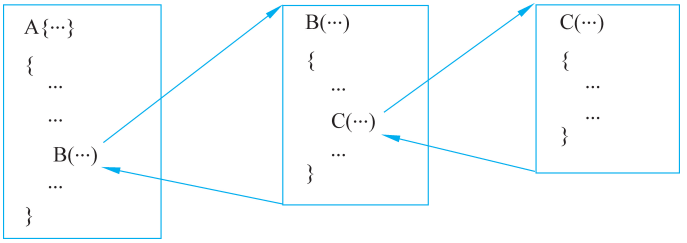


图 3-1　函数的嵌套调用

例 3-3　银行排队问题。在需要顺序操作但人群众多的场合,排队是现代文明的一种体现。例如,储户到银行办理个人储蓄业务,需要领取一张排队单,在排队单上打印了储户的顺序号以及前面的人数,储户只需坐在椅子上等待,窗口会按照先来先服务的原则顺次叫号。那么,如何实现这种先来先服务的功能呢?

例 3-4　打印缓冲区。在计算机系统中,经常会遇到两个设备在处理数据时速度不匹配的问题。例如,将计算机中的数据传送到打印机上打印输出,显然,打印机的打印速度远远小于计算机处理数据的速度,如果将数据直接送到打印机上,就会导致计算机处理完一批数据就要等待打印输出。为了提高计算机的效率,通常设置一个缓冲区,计算机将处理完的数据送到打印缓冲区中,打印机从打印缓冲区取出数据进行打印。由于打印机的速度比较慢,来不及打印的数据就在缓冲区排队等待,为了保证正确打印,这个缓冲区必须按照先来先服务的原则,从而解决了计算机处理速度与打印机输出速度不匹配的矛盾。

例 3-5　八皇后问题。八皇后问题是数学家高斯于 1850 年提出的。问题是在 8×8 的棋盘上摆放八个皇后,使其不能互相攻击,即任意两个皇后都不能处于同一行、同一列

或同一斜线上。八皇后问题首先要解决的问题就是如何表示棋盘？如何获得每个皇后的位置信息进而判断是否互相攻击？

3.2　栈


课件 3-2

3.2.1　栈的逻辑结构

1. 栈的定义

栈(stack)是限定仅在表的一端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，另一端称为栈底，不含任何数据元素的栈称为空栈。

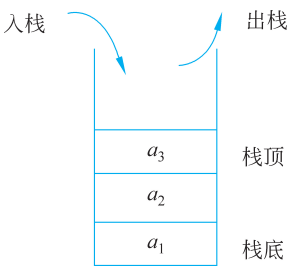


图 3-2　栈的示意图

如图 3-2 所示，栈中有 3 个元素，插入元素(也称为入栈、进栈、压栈)的顺序是 a_1、a_2、a_3，当需要删除元素(也称为出栈、弹栈)时只能删除 a_3，换言之，任何时刻出栈的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)[①]的特性。

在日常生活中，有很多栈的例子，例如，一叠摞在一起的盘子，要从这叠盘子中取出或放入一个盘子，只有在其顶部操作才是最方便的；早在计算机出现之前，会计就使用栈来记账；再如，火车扳道站、单车道死胡同等。

2. 栈的抽象数据类型定义

虽然对插入和删除操作的位置限制减少了栈操作的灵活性，但同时也使得栈的操作更有效更容易实现。其抽象数据类型定义为

```
ADT Stack
DataModel
    栈中元素具有后进先出特性，相邻元素具有前驱和后继关系
Operation
    InitStack
        输入：无
        功能：栈的初始化
        输出：一个空栈
    DestroyStack
        输入：无
        功能：栈的销毁
        输出：释放栈的存储空间
```

① 需要注意的是，栈只是对线性表的插入和删除操作的位置进行了限制，并没有限定插入和删除操作进行的时间，也就是说，出栈可随时进行，只要某个元素位于栈顶就可以出栈。例如，假定 3 个元素按 a、b、c 的次序依次进栈，且每个元素只允许进一次栈，则所有元素都出栈后，可能的出栈序列有 abc、acb、bac、bca、cba 5 种。

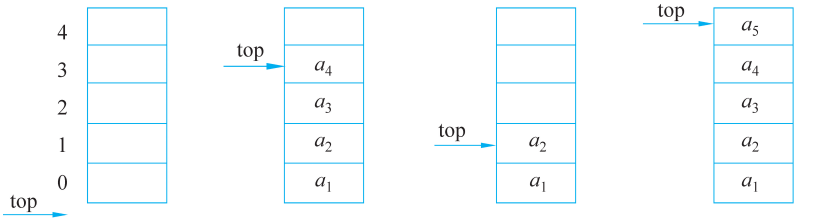
```
    Push
        输入：元素值 x
        功能：入栈操作，在栈顶插入一个元素 x
        输出：如果插入成功，栈顶增加了一个元素，否则返回插入失败信息
    Pop
        输入：无
        功能：出栈操作，删除栈顶元素
        输出：如果删除成功，返回被删除的元素值，否则返回删除失败信息
    GetTop
        输入：无
        功能：取栈顶元素，读取当前的栈顶元素
        输出：若栈不空，返回当前的栈顶元素值，否则返回取栈顶失败信息
    Empty
        输入：无
        功能：判空操作，判断栈是否为空
        输出：如果栈为空，返回 1，否则返回 0
endADT
```

3.2.2 栈的顺序存储结构及实现

1. 顺序栈的存储结构定义

栈的顺序存储结构称为**顺序栈**(sequential stack)。

顺序栈本质上是顺序表的简化，唯一需要确定的是用数组的哪一端表示栈底。通常把数组中下标为 0 的一端作为栈底，同时附设变量 top 指示栈顶元素在数组中的位置[①]。设存储栈的数组长度为 StackSize，则栈空时栈顶位置 top＝－1；栈满时栈顶位置 top＝StackSize－1。入栈时，栈顶位置 top 加 1；出栈时，栈顶位置 top 减 1。栈操作的示意图如图 3-3 所示。



(a) top=−1，栈空　(b) $a_1a_2a_3a_4$依次入栈　(c) a_4a_3依次出栈　(d) top=4，栈满

图 3-3　栈操作的示意图

下面给出顺序栈的存储结构定义：

① 在有些教材中，将 top 指向栈中第一个空闲位置，如果这样的话，空栈应该表示为 top ＝ 0。

```
#define StackSize 100            /* 假定栈元素最多 100 个 */
typedef int DataType;            /* 定义栈元素的数据类型,假设为 int 型 */
typedef struct
{
    DataType data[StackSize];    /* 存放栈元素的数组 */
    int top;                     /* 栈顶位置,栈顶元素在数组中的下标 */
} SeqStack;
```

2. 顺序栈的实现

根据栈的操作定义,容易写出顺序栈基本操作的算法,且时间复杂度均为 $O(1)$。

(1) 顺序栈的初始化。初始化一个空的顺序栈只需将栈顶位置 top 置为 -1,算法的 C 语言实现如下:

```
void InitStack(SeqStack * S)
{
    S->top = -1;
}
```

(2) 顺序栈的销毁。顺序栈是静态存储分配,在顺序栈变量退出作用域时自动释放顺序栈所占存储单元,因此,顺序栈无须销毁。

(3) 入栈操作。在栈中插入元素 x 只需将栈顶位置 top 加 1,然后在 top 的位置填入元素 x。函数 Push 的返回值表示插入操作是否成功,算法的 C 语言实现如下:

```
int Push(SeqStack * S,DataType x)
{
    if (S->top == StackSize-1) {printf("上溢错误,插入失败\n"); return 0;}
    S->data[++S->top] = x; return 1;
}
```

(4) 出栈操作。出栈操作只需取出栈顶元素,然后将栈顶位置 top 减 1。函数 Pop 的返回值表示删除操作是否成功,如果删除成功,被删元素通过指针参数 ptr 返回,算法的 C 语言实现如下:

```
int Pop(SeqStack * S,DataType * ptr)
{
    if (S->top == -1) {printf("下溢错误,删除失败\n"); return 0;}
     *ptr = S->data[S->top--]; return 1;
}
```

(5) 取栈顶元素。取栈顶元素只是将 top 位置的栈顶元素取出,并不修改栈顶位置。算法的 C 语言实现如下:

```
int GetTop(SeqStack * S,DataType * ptr)
{
    if (S->top == -1) {printf("下溢错误,取栈顶失败\n"); return 0;}
     *ptr = S->data[S->top]; return 1;
}
```

(6) 判空操作。顺序栈的判空操作只需判断 top 是否等于－1,算法的 C 语言实现如下:

```
int Empty(SeqStack * S)
{
    if (S->top == -1) return 1;              /* 栈空则返回 1 */
    else return 0;
}
```

3. 顺序栈的使用

在定义了顺序栈的存储结构 SeqStack 并实现了基本操作后,程序中就可以使用 SeqStack 类型来定义变量,可以调用实现基本操作的函数来完成相应功能。范例程序如下:


源代码 3-1

```
#include <stdio.h>
#include <stdlib.h>
/* 将顺序栈的存储结构定义和各个函数定义放到这里 */

int main( )
{
    DataType x;
    SeqStack S;                                  /* 定义结构体变量 S 为顺序栈类型 */
    InitStack(&S);                               /* 初始化顺序栈 S */
    printf("对 15 和 10 执行入栈操作,");
    Push(&S, 15);
    Push(&S, 10);
    if (GetTop(&S, &x) == 1)
        printf("当前栈顶元素为: %d\n", x);       /* 输出当前栈顶元素 10 */
    if (Pop(&S, &x) == 1)
        printf("执行一次出栈操作,删除元素: %d\n ", x);   /* 输出出栈元素 10 */
    if (GetTop(&S, &x) == 1)
        printf("当前栈顶元素为: %d\n", x);       /* 输出当前栈顶元素 15 */
    printf("请输入待入栈元素: ");
    scanf("%d", &x);
    Push(&S, x);
    if (Empty(&S) == 1)
```

```
        printf("栈为空\n");
    else
        printf("栈非空\n");                    /* 栈有 2 个元素,输出栈非空 */
    return 0;
}
```

3.2.3　栈的链接存储结构及实现

1. 链栈的存储结构定义

栈的链接存储结构称为**链栈**(linked stack),通常用单链表表示,其结点结构与单链表的结点结构相同,请参见 2.4.1 节。因为只能在栈顶执行插入和删除操作,显然以单链表的头部做栈顶是最方便的,通常将链栈表示成如图 3-4 所示的形式。

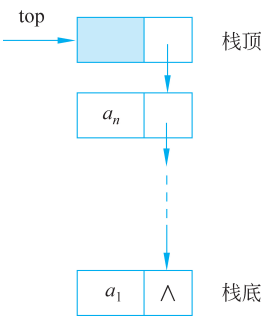


图 3-4　链栈示意图

2. 链栈的实现

链栈的基本操作本质上是单链表基本操作的简化,由于插入和删除操作仅在单链表的头部进行,因此,算法的时间复杂度均为 $O(1)$。

(1) 链栈的初始化。空链栈只有头结点,算法如下:

```
Node * InitStack( )
{
    Node * top = (Node * )malloc(sizeof(Node);
    top->next = NULL; return top;
}
```

(2) 链栈的销毁。链栈是动态存储分配,在链栈变量退出作用域前要释放链栈的存储空间。算法如下:

```
void DestroyStack(Node * top)
{
    Node * p = top;
    while (top != NULL)              /* 依次释放链栈的每一个结点 */
    {
        top = top->next;
        free(p);
        p = top
    }
}
```

(3) 入栈操作。链栈的插入操作只需处理栈顶的情况,其操作示意图如图 3-5 所

示,算法如下:

```
void Push(Node * top, DataType x)
{
    Node * s = (Node * )malloc(sizeof(Node));              /* 申请一个结点 s */
    s->data = x;
    s->next = top->next; top->next = s;                    /* 将结点 s 插在栈顶 */
}
```

(4) 出栈操作。链栈的删除操作只需处理栈顶的情况,其操作示意图如图 3-6 所示。函数 Pop 的返回值表示出栈操作是否正确执行,如果出栈正确,被删元素通过指针参数 ptr 返回,C 语言实现如下:

```
int Pop(Node * top, DataType * ptr)
{
    Node * p = top->next;
    if (top->next == NULL) {printf("下溢错误,删除失败\n"); return 0; }
    * ptr = p->data;                                       /* 存储栈顶元素 */
    top->next = p->next;                                   /* 将栈顶结点摘链 */
    free(p);
    return 1;
}
```

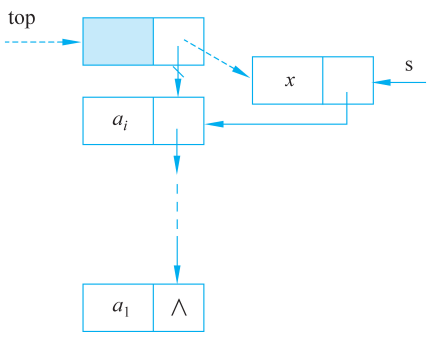


图 3-5 链栈插入操作示意图

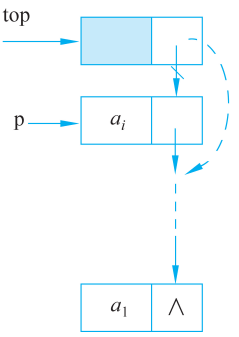


图 3-6 链栈删除操作示意图

(5) 取栈顶元素。取栈顶元素只需返回栈顶指针 top 所指结点的数据域,C 语言实现如下:

```
int GetTop(Node * top, DataType * ptr)
{
    if(top == NULL) {printf("下溢错误,取栈顶失败\n"); return 0; }
    * ptr = top->data; return 1;
}
```

(6) 判空操作。链栈的判空操作只需判断 top 是否等于 NULL,C 语言实现如下:

```
int Empty(Node * top)
{
    if(top == NULL) return 1;                    /* 栈空则返回 1 */
    else return 0;
}
```

3. 链栈的使用

在定义了单链表的结点结构 Node 并实现了链栈的基本操作后，就可以定义指向 Node 的栈顶指针来操作链栈，可以调用实现基本操作的函数来完成相应功能。范例程序如下：


源代码 3-2

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
/* 将单链表的结点结构定义和链栈的各个函数定义放到这里 */

int main( )
{
    DataType x;
    Node * top = NULL;                          /* 定义链栈的栈顶指针并初始化 */
    top = InitStack(top);                       /* 初始化链栈 */
    printf("对 15 和 10 执行入栈操作,");
    Push(top, 15);
    Push(top, 10);
    if (GetTop(top, &x) == 1)
        printf("当前栈顶元素为: %d\n", x);      /* 输出当前栈顶元素 10 */
    if (Pop(top, &x) != 0)
        printf("执行一次出栈操作,删除元素: %d\n ", x);     /* 输出出栈元素 10 */
    if (GetTop(top, &x) == 1)
        printf("当前栈顶元素为: %d\n", x);      /* 输出当前栈顶元素 15 */
    printf("请输入待插入元素: ");
    scanf("%d", &x);
    Push(top, x);
    if (Empty(top) == 1)
        printf("栈为空\n");
    else
        printf("栈非空\n");                     /* 栈有 2 个元素,输出栈非空 */
    DestroyStack(top);
    return 0;
}
```

3.2.4 顺序栈和链栈的比较

顺序栈和链栈基本算法的时间复杂度均为$O(1)$,因此唯一可以比较的是空间性能。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浪费空间的问题。链栈没有栈满的问题,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。所以当栈的使用过程中元素个数变化较大时,应该采用链栈,反之,应该采用顺序栈。


课件 3-3

3.3 队　　列

3.3.1 队列的逻辑结构

1. 队列的定义

队列(queue)是只允许在一端进行插入操作,在另一端进行删除操作的线性表,允许插入(也称为入队、进队)的一端称为队尾,允许删除(也称为出队)的一端称为队头。图3-7所示是一个有5个元素的队列,入队的顺序为a_1、a_2、a_3、a_4、a_5,出队的顺序依然是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)的特性。

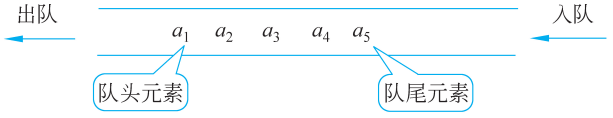


图3-7 队列的示意图

现实世界中有许多问题可以用队列描述,例如,对顾客服务部门(例如银行、电信等)的工作往往是按队列方式进行的,这类系统称作排队系统。在程序设计中,经常使用队列保存按先进先出方式处理的数据,例如键盘缓冲区、操作系统中的作业调度等。

2. 队列的抽象数据类型定义

```
ADT Queue
DataModel
    队列中元素具有先进先出特性,相邻元素具有前驱和后继关系
Operation
    InitQueue
        输入:无
        功能:队列的初始化
        输出:一个空队列
    DestroyQueue
        输入:无
        功能:队列的销毁
        输出:释放队列占用的存储空间
```

```
    EnQueue
        输入：元素值 x
        功能：入队操作，在队尾插入元素 x
        输出：如果插入成功，队尾增加了一个元素，否则返回插入失败信息
    DeQueue
        输入：无
        功能：出队操作，删除队头元素
        输出：如果删除成功，队头减少了一个元素，否则返回删除失败信息
    GetHead
        输入：无
        功能：读取队头元素
        输出：若队列不空，返回队头元素，否则返回取队头失败信息
    Empty
        输入：无
        功能：判空操作，判断队列是否为空
        输出：如果队列为空，返回 1，否则返回 0
endADT
```

3.3.2　队列的顺序存储结构及实现

1. 顺序队列

队列的顺序存储结构称为**顺序队列**(sequential queue)。

假设队列有 n 个元素，顺序队列把队列的所有元素存储在数组的前 n 个单元。如果把队头元素放在数组中下标为 0 的一端，则入队操作相当于追加，不需要移动元素，其时间性能为 $O(1)$；但是出队操作的时间性能为 $O(n)$，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，所有元素都要向前移动一个位置，如图 3-8(c)所示。

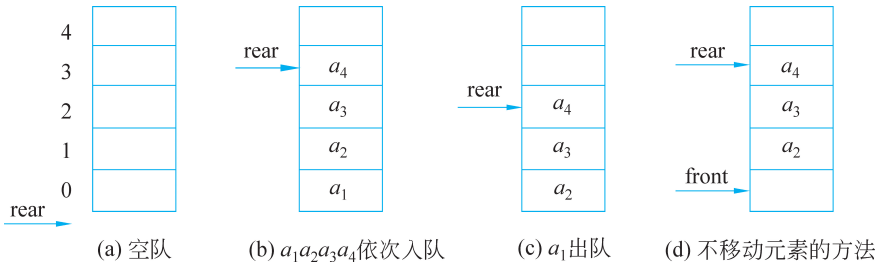


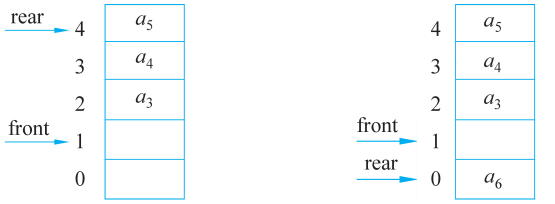
图 3-8　顺序队列的操作示意图

如果放宽队列的所有元素必须存储在数组的前 n 个单元这一条件，就可以得到一种更为有效的存储方法，如图 3-8(d)所示。此时入队和出队操作的时间性能都是 $O(1)$，因为没有移动任何元素，但是队列的队头和队尾都是活动的，因此，需要设置队头、队尾两个

位置变量 front 和 rear,并且约定:front 指向队头元素的前一个位置,rear 指向队尾元素[①]。

2. 循环队列的存储结构定义

在顺序队列中,随着队列的插入和删除操作,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"。当元素被插入数组中下标最大的位置之后,数组空间就用尽了,尽管此时数组的低端还有空闲空间,这种现象叫作"假溢出",如图 3-9(a)所示。



(a) front=1, rear=4 假溢出　　(b) rear=0 解决假溢出

图 3-9　循环队列的假溢出及其解决

解决假溢出的方法是将存储队列的数组看成是头尾相接的循环结构,即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,如图 3-9(b)所示,这可以通过取模操作来实现。队列的这种头尾相接的顺序存储结构称为**循环队列**(circular queue)。

下面给出循环队列的存储结构定义:

```
#define QueueSize 100          /* 定义数组的最大长度 */
typedef int DataType;          /* 定义队列元素的数据类型,假设为 int 型 */
typedef struct
{
    DataType data[QueueSize]; /* 存放队列元素的数组 */
    int front, rear;           /* 下标,队头元素和队尾元素的位置 */
} CirQueue;
```

3. 循环队列的实现

在循环队列中,如何判定队空和队满呢?如图 3-10(a)和(c)所示,队列中只有一个元素,执行出队操作,队头位置加 1 后与队尾位置相等,即队空的条件是 front = rear。

图 3-11(a)和(c)所示数组中只有一个空闲单元,执行入队操作,队尾位置加 1 后与队头位置相等,即队满的条件也是 front = rear。如何将队空和队满的判定条件区分开呢?

---

① 这样约定的目的是方便运算,如队尾位置减去队头位置正好等于队列的长度。有些参考书约定队头位置 front 指向队头元素,队尾位置 rear 指向队尾元素;或队头位置 front 指向队头元素,队尾位置 rear 指向队尾元素的后一个位置。

可以浪费一个数组元素空间[①]，把图 3-11(a)和(c)所示的情况视为队满，此时队尾位置和队头位置正好差 1，即队满的条件是(rear+1) % QueueSize = front。

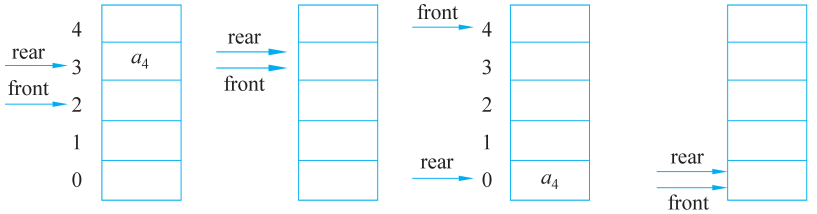


图 3-10　循环队列队空的判定

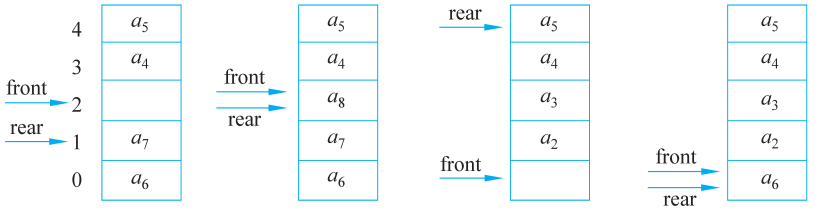


图 3-11　循环队列队满的判定

根据队列的操作定义，容易写出循环队列基本操作的算法，其时间复杂度均为 $O(1)$。

(1) 循环队列的初始化。初始化一个空的循环队列只需将队头 front 和队尾 rear 同时指向数组的某一个位置，一般是数组的高端，C 语言实现如下：

```
void InitQueue(CirQueue * Q)
{
    Q->front =Q->rear =QueueSize-1;
}
```

(2) 循环队列的销毁。循环队列是静态存储分配，在循环队列变量退出作用域时自动释放所占内存单元，因此，循环队列无须销毁。

(3) 入队操作。循环队列的入队操作只需将队尾位置 rear 在循环意义下加 1，然后将待插元素 x 插入队尾位置。函数 EnQueue 的返回值表示入队操作是否正确执行，C 语言实现如下：

① 算法设计有一个重要的原则——时空权衡。一般来说，牺牲空间或其他替代资源，通常可以减少时间代价。例如，在单链表的开始结点之前附设一个头结点，使得单链表的插入和删除等操作不用考虑表头的特殊情况；在双链表中，结点设置了指向前驱结点的指针和指向后继结点的指针，增加了指针的结构性开销，减少了查找前驱和后继的时间代价。

```
int EnQueue(CirQueue *Q, DataType x)
{
    if ((Q->rear+1) %QueueSize==Q->front) {
      printf("上溢错误,插入失败\n"); return 0;
  }
  Q->rear=(Q->rear+1) %QueueSize;          /*队尾位置在循环意义下加1*/
  Q->data[Q->rear]=x;                      /*在队尾处插入元素*/
  return 1;
}
```

(4) 出队操作。循环队列的出队操作只需将队头位置 front 在循环意义下加 1,然后读取并返回队头元素。函数 DeQueue 的返回值表示出队操作是否正确执行,如果正确出队,被删元素通过指针参数 ptr 返回,C 语言实现如下:

```
int DeQueue(CirQueue *Q,DataType *ptr)
{
    if (Q->rear==Q->front) {printf("下溢错误,删除失败\n"); return 0; }
    Q->front=(Q->front+1)%QueueSize;     /*队头位置在循环意义下加1*/
    *ptr=Q->data[Q->front];              /*读取出队前的队头元素*/
    return 1;
}
```

(5) 取队头元素。读取队头元素与出队操作类似,唯一的区别是不改变队头位置,C 语言实现如下:

```
int GetHead(CirQueue *Q,DataType *ptr)
{
    int i;
    if (Q->rear==Q->front) {printf("下溢错误,取队头失败\n"); return 0; }
    i=(Q->front+1)%QueueSize;            /*注意不改变队头位置*/
    *ptr=Q->data[i]; return 1;
}
```

(6) 判空操作。循环队列的判空操作只需判断 front 是否等于 rear,C 语言实现如下:

```
int Empty(CirQueue *Q)
{
    if(Q->rear==Q->front) return 1;      /*队列为空返回1*/
    else return 0;
}
```

4. 循环队列的使用

在定义了循环队列的存储结构 CirQueue 并实现了基本操作后，程序中就可以使用 CirQueue 类型来定义变量，可以调用实现基本操作的函数来完成相应功能。范例程序如下：


源代码 3-3

```
#include <stdio.h>
#include <stdlib.h>
/* 将循环队列的存储结构定义和各个函数定义放到这里 */

int main( )
{
    DataType x;
    CirQueue Q;                          /* 定义结构体变量 Q 为循环队列类型 */
    InitStack(&Q);                       /* 初始化循环队列 Q */
    printf("对 5 和 8 执行入队操作,");
    EnQueue(&Q, 5);
    EnQueue(&Q, 8);
    if (GetHead(&Q, &x) == 1)
        printf("当前队头元素为: %d\n", x);                  /* 输出当前队头元素 5 */
    if (DeQueue(&Q, &x) == 1)
        printf("执行一次出队操作,出队元素是: %d\n ", x);    /* 输出出队元素 5 */
    if (GetHead(&Q, &x) == 1)
        printf("当前队头元素为: %d\n", x);                  /* 输出当前队头元素 8 */
    printf("请输入入队元素: ");
    scanf("%d", &x);
    EnQueue(&Q, x);
    if (Empty(&Q) == 1)
        printf("队列为空\n");
    else
        printf("队列非空\n");                    /* 队列有两个元素,输出队列非空 */
    return 0;
}
```

3.3.3　队列的链接存储结构及实现

1. 链队列的存储结构定义

队列的链接存储结构称为**链队列**(linked queue)，通常用单链表表示，其结点结构与单链表的结点结构相同，请参见 2.4.1 节。为了使空队列和非空队列的操作一致，链队列也加上头结点。根据队列的先进先出特性，为了操作上的方便，设置队头指针指向链队列的头结点，队尾指针指向终端结点，如图 3-12 所示。

下面给出链队列的存储结构定义：

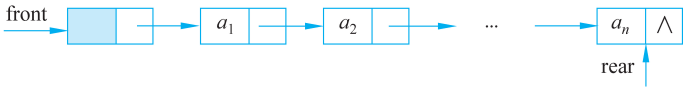


图 3-12 链队列示意图

```
typedef int DataType;                    /*定义队列元素的数据类型,假设为 int 型*/
typedef struct Node                      /*定义链队列的结点结构*/
{
    DataType data;
    struct Node *next;
} Node;
typedef struct                           /*定义链队列*/
{
    Node *front, *rear;
} LinkQueue;
```

2. 链队列的实现

链队列基本操作的实现本质上是单链表操作的简化,且时间复杂度均为 $O(1)$。

(1) 链队列的初始化。初始化链队列只需申请头结点,然后让队头指针和队尾指针均指向头结点,算法如下:

```
void InitQueue(LinkQueue *Q)
{
    Node *s=(Node *)malloc(sizeof(Node)); s->next=NULL;
    Q->front=Q->rear=s;                  /*队头指针和队尾指针均指向头结点*/
}
```

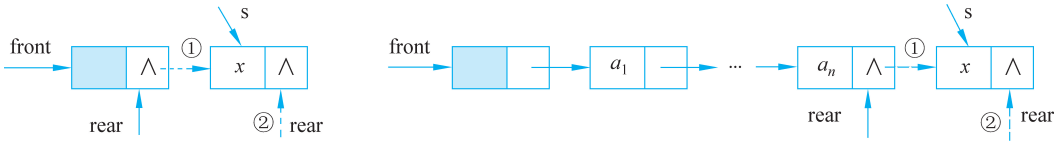
(2) 链队列的销毁。链队列是动态存储分配,需要释放链队列的所有结点的存储空间,算法如下:

```
void DestroyQueue(LinkQueue *Q)
{
    Node *p=Q->front,temp;
    while (p!=NULL)                      /*依次释放链队列的结点*/
    {
        temp=p-> next;
        free(p);
        p=temp;
    }
}
```

(3) 入队操作。链队列的插入操作只考虑在链表的尾部进行,由于链队列带头结点,空

链队列和非空链队列的插入操作语句一致，其操作示意图如图 3-13 所示。C 语言实现如下：

```
void EnQueue(LinkQueue * Q,DataType x)
{
    Node * s = (Node *)malloc(sizeof(Node));
    s->data = x; s->next = NULL;              /* 申请一个数据域为 x 的结点 s */
    Q->rear->next = s;Q->rear = s;            /* 将结点 s 插入队尾 */
}
```



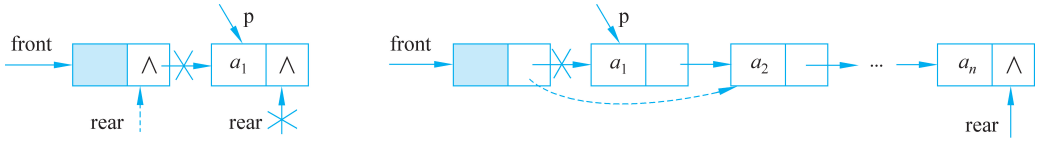
(a) 空链队列的入队操作　　(b) 非空链队列的入队操作

图 3-13　链队列的入队操作

（① rear—>next=s; ②rear=s;）

（4）出队操作。链队列的删除操作只考虑在链表的头部进行，注意队列长度等于 1 的特殊情况，其操作示意图如图 3-14 所示。函数 DeQueue 表示出队操作是否正确执行，如果正确出队，被删元素通过指针参数 ptr 返回，C 语言实现如下：

```
int DeQueue(LinkQueue * Q,DataType * ptr)
{
    Node * p;
    if (Q->rear == Q->front) {printf("下溢错误,删除失败\n"); return 0; }
    p = Q->front->next; * ptr = p->data;      /* 存储队头元素 */
    Q->front->next = p->next;                 /* 将队头元素所在结点摘链 */
    if (p->next == NULL)                      /* 判断出队前队列长度是否为 1 */
        Q->rear = Q->front;
    free(p);
    return 1;
}
```



(a) 特殊情况：队列长度等于1　　(b) 一般情况：队列长度大于1

图 3-14　链队列出队操作示意图

（5）取队头元素。取链队列的队头元素只需返回第一个元素结点的数据域，算法

如下:

```
int GetHead(LinkQueue *Q,DataType *ptr)
{
    Node *p=NULL;
    if (Q->rear==Q->front) {printf("下溢错误,取队头失败\n"); return 0; }
    p=Q->front->next;
    *ptr=p->data;
return 1;
}
```

(6) 判空操作。链队列的判空操作只需判断 front 是否等于 rear,算法如下:

```
int Empty(LinkQueue *Q)
{
    if (Q->rear==Q->front) return 1;          /*队列为空返回 1*/
    else return 0;
}
```

3. 链队列的使用

在定义了链队列的存储结构 LinkQueue 并实现了基本操作后,就可以使用 LinkQueue 类型定义变量,可以调用实现基本操作的函数来完成相应功能。范例程序如下:


源代码 3-4

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
/*将链队列的存储结构定义和各个函数定义放到这里*/

int main( )
{
    DataType x;
    LinkQueue Q;                              /*定义结构体变量 Q 为链队列类型*/
    InitQueue(&Q);                            /*初始化链队列 Q*/
    printf("对 5 和 8 执行入队操作,");
    EnQueue(&Q, 5);
    EnQueue(&Q, 8);
    if (GetHead(&Q, &x)==1)
        printf("当前队头元素为: %d\n", x);      /*输出当前队头元素 5*/
    if (DeQueue(&Q, &x)==1)
        printf("执行一次出队操作,出队元素是: %d\n ", x);   /*输出出队元素 5*/
    if (GetHead(&Q, &x)==1)
        printf("当前队头元素为: %d\n", x);      /*输出当前队头元素 8*/
    printf("请输入入队元素: ");
```

```
    scanf("%d", &x);
    EnQueue(&Q, 5);
    if (Empty(&Q) == 1)
        printf("队列为空\n");
    else
        printf("队列非空\n");              /* 队列有两个元素,输出队列非空 */
    DestroyQueue(&Q);
    return 0;
}
```

3.3.4　循环队列和链队列的比较

循环队列和链队列基本算法的时间复杂度均为 $O(1)$，因此，可以比较的只有空间性能。初始时循环队列必须确定一个固定的长度，所以有存储元素个数的限制和浪费空间的问题。链队列没有溢出的问题，只有当内存没有可用空间时才会出现溢出，但是每个元素都需要一个指针域，从而产生了结构性开销。所以当队列中元素个数变化较大时，应该采用链队列，反之，应该采用循环队列。如果确定不会发生溢出，也可以采用顺序队列。

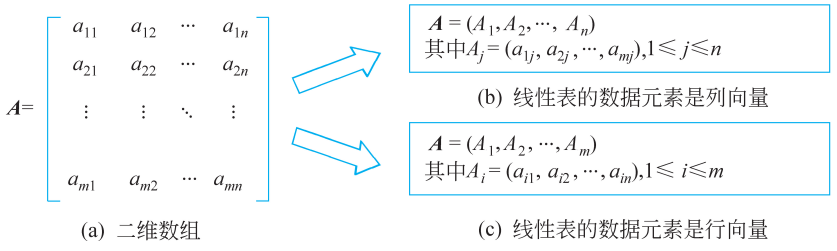
3.4　多维数组


课件 3-4

3.4.1　数组的逻辑结构

1. 数组的定义

数组(array)是由类型相同的数据元素构成的有序集合，每个数据元素称为一个数组元素(简称为元素)，每个元素受 $n(n\geqslant 1)$ 个线性关系的约束，每个元素在 n 个线性关系中的序号 $i_1,i_2,\cdots,i_n$，称为该元素的下标，并称该数组为 n 维数组。可以看出，数组的特点是数据元素本身可以具有某种结构，但属于同一数据类型。例如，一维数组可以看作一个线性表；二维数组可以看作元素是线性表的线性表；以此类推。所以，数组是线性表的推广，如图 3-15 所示。本章重点讨论二维数组。



(a) 二维数组　(b) 线性表的数据元素是列向量　(c) 线性表的数据元素是行向量

图 3-15　数组是线性表的推广

2. 数组的抽象数据类型定义

数组是一个具有固定格式和数量的数据集合，在数组上一般不能执行插入或删除某

个数组元素的操作。因此,数组中通常只有两种基本操作:①读操作:给定一组下标,读取相应的数组元素;②写操作:给定一组下标,存储或修改相应的数组元素。这两种操作本质上对应一种操作——寻址,即根据一组下标定位相应的数组元素。下面给出数组的抽象数据类型定义。

```
ADT Matrix
DataModel
    相同类型的数据元素的有序集合
    每个数据元素受 n(n≥1)个线性关系的约束并由一组下标唯一标识
Operation
    InitMatrix
        输入:数组的维数 n 和各维的长度 l1, l2, …, ln
        功能:数组的初始化
        输出:一个空的 n 维数组
    GetMatrix
        输入:一组下标 i1, i2, …, in
        功能:读操作,读取这组下标对应的数组元素
        输出:对应下标 i1, i2, …, in 的元素值
    SetMatrix
        输入:元素值 e,一组下标 i1, i2, …, in
        功能:写操作,存储或修改这组下标对应的数组元素
        输出:对应下标 i1, i2, …, in 的元素值改为 e
endADT
```

3.4.2 数组的存储结构与寻址

由于数组一般不执行插入和删除操作,也就是说,一旦建立了数组,其元素个数和元素之间的关系就不再发生变动,而且,数组是一种特殊的数据结构,通常要求能够随机存取,因此,数组采用顺序存储。由于内存单元是一维结构,而多维数组是多维结构,所以,采用顺序存储结构存储数组首先需要将多维结构映射到一维结构。常用的映射方法有两种:以行序为主序(row major order,即按行优先)和以列序为主序(column major order,即按列优先)。例如,C语言中的数组是按行优先存储的。

对于二维数组,按行优先存储的基本思想是先行后列,先存储行号较小的元素,行号相同者先存储列号较小的元素。设二维数组行下标与列下标的范围分别为$[l_1, h_1]$与$[l_2, h_2]$,则元素a_{ij}的存储地址可由下式确定:

$$\mathrm{LOC}(a_{ij}) = \mathrm{LOC}(a_{l_1 l_2}) + [(i - l_1) \times (h_2 - l_2 + 1) + (j - l_2)] \times c \qquad (3\text{-}1)$$

在式(3-1)中,$i \in [l_1, h_1]$,$j \in [l_2, h_2]$,且i与j均为整数;$\mathrm{LOC}(a_{ij})$是元素a_{ij}的存储地址;$\mathrm{LOC}(a_{l_1 l_2})$是二维数组中第一个元素$a_{l_1 l_2}$的存储地址,通常称为基地址;$c$是每个元素所占存储单元数目。二维数组的寻址过程如图3-16所示。

二维数组按列优先存储的基本思想是先列后行,先存储列号较小的元素,列号相同者先存储行号较小的元素。任一元素存储地址的计算与按行优先存储类似。