第5章

队　列

队列是一种先进先出的线性表，其操作规则类似于日常生活中人们的排队等候服务，在计算机系统中也有广泛的应用。本章主要介绍队列的概念、实现和应用，并简要说明双端队列和优先级队列。

扫一扫

视频讲解

5.1　队列的基本概念

与栈一样，队列(queue)也是一种特殊的线性表。从逻辑结构角度看，队列与普通线性表和栈没有不同。将队列定义为 n(n≥0)个数据元素构成的有限序列。当 n=0 时，称为空队列。当队列非空时，可记为 Q=(a_0，a_1，a_2，…，a_i，…，a_{n-1})，其中每个元素有一个固定的位序号。队列的特殊性是被限制在线性表的一端进行插入，在另一端进行删除，允许插入的一端称为**队尾**，允许删除的一端称为**队首**或**队头**。插入操作称为**入队**或**进队**，删除操作称为**出队**或**退队**，如图 5.1 所示。

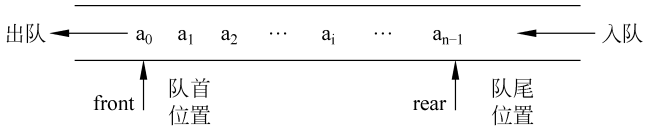


图 5.1　队列示意图

队列的特点是最先入队的元素最先出队，即具有“先进先出”(First In First Out，FIFO)的特性。因此，队列也被称为**先进先出表**。例如，一队人在游乐场门口等待检票进入游乐场，最先来的人排在队头最先进入游乐场，而最后来的人排在队尾最后进入游乐场。商店、食堂、影院等服务场所也都按照队列先进先出的原则处理客户请求。

队列广泛应用于计算机系统的资源分配、数据缓冲和任务调度等场景中。例如，局域

网用户申请使用网络打印机、多台终端访问 Web 服务器、键盘输入缓冲、操作系统的作业调度等都基于队列结构。

在程序设计中，队列也经常用于保存动态生成的多个任务或数据，以确保相应任务或数据按照先进先出的次序进行处理。

5.2 队列的抽象数据类型

T 类型元素构成的队列是由 T 类型元素构成的有限序列，并且具有以下基本操作。

(1) 构造一个空队列(__init__)。

(2) 判断一个队列是否为空(empty)。

(3) 求队列的长度(__len__)。

(4) 入队一个元素(append)。

(5) 读取队首元素并出队(serve)。

(6) 读取队首元素(retrieve)。

在以上 ADT 描述中，对队列逻辑结构的描述与对线性表和栈的描述完全一致，不同的是三者的操作不同，这里定义了对队列的 6 种操作。与栈的操作相比，不难发现，入队 append 操作类似于栈的 push 操作；出队 serve 操作类似于栈的 pop 操作；取队首 retrieve 操作则类似于栈的 get_top 操作。

5.3 队列的顺序存储及实现

扫一扫

视频讲解

5.3.1 物理模型法

在人们排队等候服务的物理模型中，新元素入队，即加入队列的尾部；而当队首元素得到服务离开队列时，队列中剩余的所有元素都向前移动一个位置，从而保证队列的队首元素始终在队列的最前端，以便顺利得到服务。

在计算机中也可以参考物理模型表示队列。将队列从队首至队尾的所有元素依次存储在数组中，队首固定为 0 位置；当入队一个新元素时，新元素加入数组的末尾，队尾后移一个位置；当出队队首元素时，数组中所有的剩余元素依次前移一个位置，队尾前移一个位置。

在 Python 中，假设用列表 entry 存储队列，从 entry 的 0 号位置开始依次存放从队首到队尾的所有元素，entry[0]即为队首元素，列表尾部即是队尾位置，如图 5.2 所示。

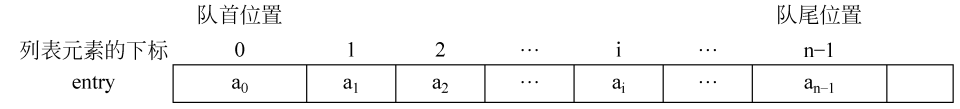


图 5.2 用列表实现物理模型法表示的队列

这样,判别队列是否为空对应于判别列表 entry 是否为空,求队列的长度对应于对列表 entry 求长度;入队操作对应于 entry 的 append 操作,出队操作对应于 entry 的 pop(0)操作,取队首元素 retrieve 操作则对应于读取列表的 0 号元素。这些方法实现的语句都很简单,需要注意,在出队(serve)和读取队首(retrieve)方法中需要判别队列是否为空,如果为空,则需要返回操作失败的信息,这里用返回 None 来表示操作失败。以下是物理模型队列类 PhysicalModelQueue 的定义:

```
class PhysicalModelQueue:
    def __init__(self):
        self._entry = []

    def __len__(self):
        return len(self._entry)

    def empty(self):
        return not self._entry

    def append(self, item):
        self._entry.append(item)

    def serve(self):
        if not self.empty():
            return self._entry.pop(0)
        else:
            return None

    def retrieve(self):
        if not self.empty():
            return self._entry[0]
        else:
            return None
```

虽然这种方案实现起来很简单,但出队算法效率差。队列的出队操作对应于列表 entry 的 pop(0)操作,0 号位置后的 n−1 个元素将全部往前平移一个位置,元素的移动次数为 n−1,算法的时间复杂度为 O(n)。即使 Python 列表在实现 pop 操作时会采用内存批量复制的方法,性能有所提升,但时间复杂度的数量级没有变化。在 Python 中,向前平移的是元素指针;而在 C++等语言中,向前平移的是元素本身,如果每个元素个体很大,时间花费更大。总之,当队列长度较长时不建议使用物理模型法表示队列。

扫一扫

视频讲解

5.3.2 线性顺序队列

假设使用初始容量为 capacity 的数组 entry 依次存储从队首到队尾的所有元素,并

设两个整型下标 front 和 rear 分别指示队列的队首位置和队尾位置，如图 5.3 所示。

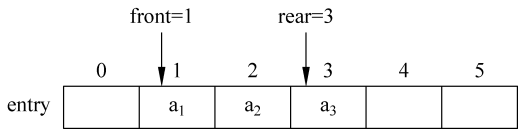


图 5.3 线性顺序队列示例

初始空队列时，生成初始容量为 capacity 的数组 entry，front 为 0，rear 为－1。入队时，队尾下标 rear 增 1，元素放在新队尾位置；出队时，获得队首元素，队首下标 front 增 1。

当队列空时，front＝rear＋1，或者也可用 front＞rear 来判别队列是否为空；当队列满时，rear 到达列表的尾端，即 rear≥capacity－1。

以下为 Python 实现的线性顺序队列类 LinearQueue。

```
class LinearQueue:
    def __init__(self, cap = 10):
        self._capacity = cap
        self._entry = [None for x in range(0, self._capacity)]
        self._front = 0
        self._rear = -1

    def empty(self):
        return self._front > self._rear

    def __len__(self):
        return self._rear - self._front + 1

    def append(self, item):
        if self._rear >= self._capacity - 1:
            raise Exception("overflow")
        else:
            self._rear += 1
            self._entry[self._rear] = item

    def serve(self):
        if self.empty():
            return None
        else:
            x = self._entry[self._front]
            self._front += 1
            return x

    def retrieve(self):
```

```
        if self.empty():
            return None
        else:
            return self._entry[self._front]
```

假设队列容量 capacity 为 6，如果对初始为空的线性顺序队列进行连续的入队和出队，通过图 5.4 的入队、出队操作看看会出现什么问题。

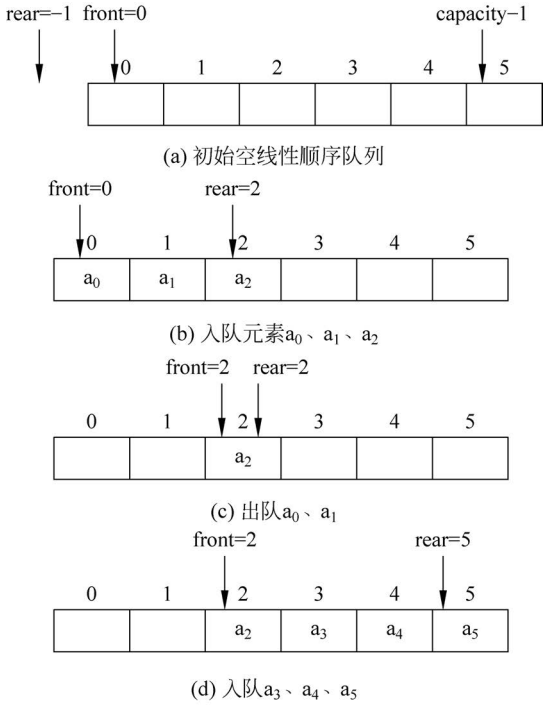


图 5.4　线性顺序队列的操作

从图 5.4(a)的空队列开始，在经过了若干次入队和出队后，到达图 5.4(d)的状态，队尾指示 rear 已到达数组尾部，无法再入队新的元素，是一种满的状态，如果需再入队，则会发生上溢出。也就是说，如果采用线性顺序队列存储结构，每出队一个元素，相应的空间就会遭到丢弃无法再利用。随着不断出队和入队，势必会产生这样的现象：数组的前端还有空余位置(即 front>0)，整个数组并没有占满，但队尾指示器 rear 已到达数组的末端而无法在当前空间入队新的元素，这种现象被称为**虚溢出**或**假溢出**。在最坏情况下，如对图 5.4(d)所示的队列继续出队 4 次后变为图 5.5 所示的状态，此时队列是空的，但仍然无法在当前队列空间入队元素。

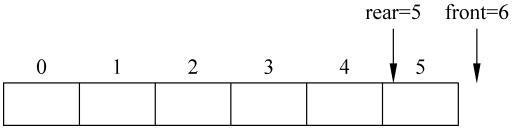


图 5.5　继续出队 a_2、a_3、a_4、a_5

由此可见，如果队列的存储区容量固定，多次入队、出队可能会造成假溢出。即使队列存储区可以像 Python 中的 list 一样自动增长，但 list 首端由于出队浪费的空间无法再利用，从而导致空间利用率低下。因此，必须对线性顺序队列方案加以改进才能达到实用的目的。

扫一扫

视频讲解

5.3.3 循环队列

为了解决线性顺序队列的假溢出问题，可以将数组 entry 的空间假想为首尾相连，即认为 capacity－1 后的空间为 0 号位置。当 rear 到达数组末端 capacity－1，而数组前端空间有空余(front＞0)时，将入队的新元素添加到 0 号位置，同时队尾指示器 rear 调整为 0。这就是队列顺序存储的**循环队列**方案，也是最常用、最有效的队列顺序存储方案。这种方案在线性顺序队列的基础上做了微调，入队时，如果 rear 下标在边界 capacity－1 处且数组前端空间有空余，则将 rear 调整为 0 后再入队；出队时，如果 front 下标在边界 capacity－1 处，元素出队后将 front 调整为 0。

在利用 Python 实现时，假设 entry 为存储队列元素的列表，入队新元素 item 的基本语句为：

```
self._rear = (self._rear + 1) % self._capacity
self._entry[self._rear] = item
```

出队队首元素的基本语句为：

```
item = self._entry[self._front]
self._front = (self._front + 1) % self._capacity
```

初始化空队列的方法可为：

```
self._front = 0
self._rear = self._capacity - 1
```

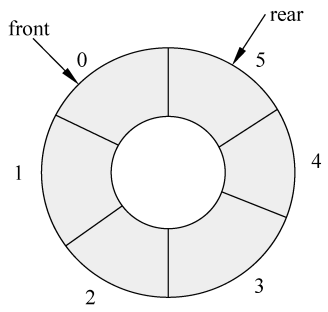
表 5.1 按照(a)～(f)的次序给出了容量为 6 的初始空循环队列依次入队、出队元素的过程。

对比表 5.1 中情况(a)和情况(c)的队空状态与情况(f)的队满状态，可以看到，队空和队满时 front 和 rear 存在相同的关系，即 front 是 rear 的后一个位置，(rear＋1) ％ capacity＝front，所以无法利用 rear 和 front 区分队列的状态是空还是满。但是，在做入队操作时必须判别出队列是否已满；在做出队操作时必须判别出队列是否为空。

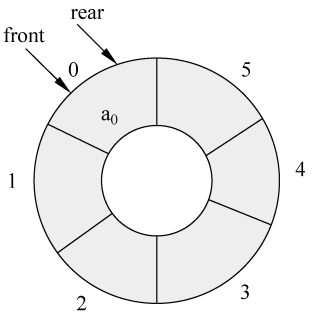
为了能够区别队空和队满的不同状态，需要增加其他的处理措施。例如，可以在类中增加队空或队满标志变量，或者增加队列长度计数变量等。

假设采用损失一个空间的做法，即在如表 5.1 中情况(e)的状态下，在 rear 和 front 相差两个位置，即数组中还剩一个空余位置时即认为已经队满，无法继续入队。因此，队空的判别条件为 front ＝＝ (rear＋1) ％ capacity；队满的判别条件为 front ＝＝ (rear＋2) ％ capacity。

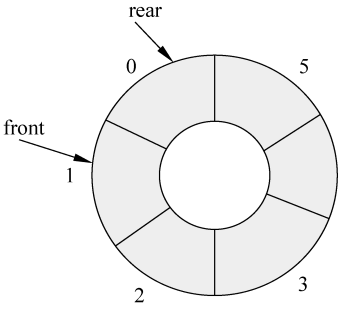
表 5.1　循环队列的入队、出队示例



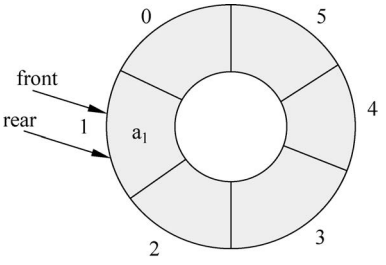
(a) 初始化空队列,front=0,rear=5



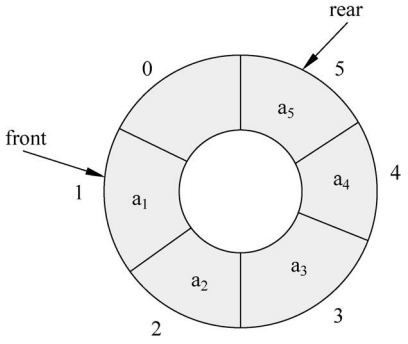
(b) 入队元素 a_0,front=0,rear=0



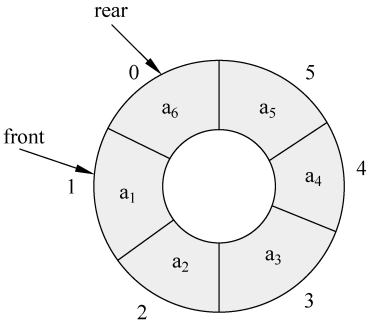
(c) 出队元素,front=1,rear=0,队列为空,rear 和 front 相差一个位置,即 front=(rear+1)%capacity



(d) 入队元素 a_1,front=1,rear=1



(e) 依次入队元素 a_2、a_3、a_4、a_5,front=1,rear=5



(f) 入队元素 a_6,front=1,rear=0,队列为满,此时 front 和 rear 的关系跟情况(c)完全相同

利用 Python 实现的循环队列类 CircularQueue 可定义如下:

```
class CircularQueue:
    def __init__(self, cap = 10):
        self._capacity = cap
        self._entry = [None for x in range(0, self._capacity)]
        self._front = 0
```

```
        self._rear = self._capacity - 1

    def empty(self):
        return  self._front == (self._rear + 1) % self._capacity

    def __len__(self):
        return  (self._rear - self._front + 1 + self._capacity) % self._capacity

    def append(self, item):
        if self._front == (self._rear + 2) % self._capacity:
            self.resize(2 * len(self._entry))
        self._rear = (self._rear + 1) % self._capacity
        self._entry[self._rear] = item

    def resize(self, cap):
        old = self._entry                    #用 old 指示原空间
        self._entry = [None] * cap           #entry 指示新分配空间
        p = self._front                      #p 在原空间中移动
        k = 0                                #k 在新空间中移动
        while p != self._rear:
            self._entry[k] = old[p]          #将原空间 p 位置的数据复制到新空间的 k 位置
            p = (1 + p) % self._capacity
            k += 1
        self._entry[k] = old[self._rear]     #最后一个队尾元素的复制
        self._front = 0
        self._rear = k
        self._capacity = cap

    def serve(self):
        if self.empty():                     #若队列为空,则出队失败,返回 None
            return None
        else:                          #非空,获得队首元素 item,更新 front 下标,返回 item
            item = self._entry[self._front]
            self._front = (self._front + 1) % self._capacity
            return item

    def retrieve(self):
        if self.empty():                     #若队列为空,返回 None
            return None
        else:
            return self._entry[self._front]  #非空,返回队首元素
```

在入队时遇到队满情况(还有一个剩余空间),append 方法调用 resize 方法扩大一倍空间,并把原队列从队首至队尾的所有元素复制到新列表空间从 0 号开始的连续位置。当空间已足够,接着更新 rear 为(rear + 1) % capacity,并将新元素 item 放在 rear 所指的位置。由于采用翻倍策略进行扩容,入队算法的摊销时间复杂度为 O(1)。

在循环队列下,出队算法的时间复杂度为 O(1)。

扫一扫

视频讲解

5.4 队列的链式存储及实现

与线性表和栈的链式存储相同,当借助链表存储队列时,在存储队列中每个元素的同时附加存储一个指针,指向其后继元素。所以,队列中每个元素的存储映像也包含两部分——元素值部分和指针部分,即第 3 章中所描述的结点。以链式存储结构表示的队列简称**链队列**。

1. 链队列结点类

链队列的结点结构与 3.4.1 节所描述的单链表结点结构一致,在类定义时仅修改了类名,定义如下:

```
class QueueNode:
    def __init__(self, data, link = None):
        self.entry = data
        self.next = link
```

2. 链队列类

假设队列中有 n 个元素,从队首到队尾分别为 a_0,a_1,…,a_{n-1},则队列非空时链队列的存储示意图如图 5.6(a)所示。与单链表类似,通常在链队列中加上表头结点。为方便对队列操作,给链队列设 front 和 rear 指针。当队列非空时,front 和 rear 分别指示表头结点和队尾结点,如图 5.6(a)所示;当队列为空时,front 和 rear 都指向表头结点,如图 5.6(b)所示。

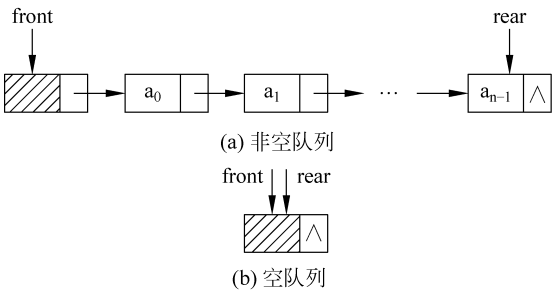


图 5.6 队列的链式结构

链队列类 LinkedQueue 的定义框架如下:

```
from QueueNode import QueueNode
class LinkedQueue:
    def __init__(self):
    def empty(self):
```

```
    def __len__(self):
    def append(self, item):
    def serve(self):
    def retrieve(self):
```

__init__方法初始化一个空队列，即生成表头结点，将指针 front 和 rear 都指向该表头结点。

```
def __init__(self):
    self._front = self._rear = QueueNode(None)
```

empty 方法判别指针 front 和 rear 是否指向同一个结点，如果是，则队列为空。

```
def empty(self):
    return self._front == self._rear
```

求队列长度需要从队首结点开始顺着链对链表中的所有结点进行计数，实现代码如下：

```
def __len__(self):
    p = self._front.next
    count = 0
    while p:
        count += 1
        p = p.next
    return count
```

图 5.7 示意了链队列的入队操作。首先生成一个值为 item 的新结点 new_rear，接着将 rear 所指原队尾结点的指针域指向新结点 new_rear，最后将队尾指针 rear 指向新结点。由于加了表头结点，空队列下的操作无须特殊处理。

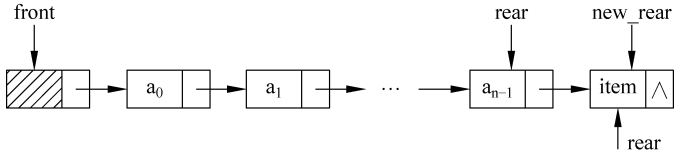


图 5.7　链队列的入队操作

入队算法的完整代码如下：

```
def append(self, item):
    new_rear = QueueNode(item)
    self._rear.next = new_rear
    self._rear = new_rear
```

接下来看出队操作。如果队列为空，如图 5.8(a)所示，则出队操作失败；如果队列只

有一个元素结点，如图 5.8(b)所示，注意唯一的元素结点出队后指针 rear 应指向表头结点；一般情况下，如图 5.8(c)所示，则将 front 所指表头结点的指针域指向原队首结点的下一结点。

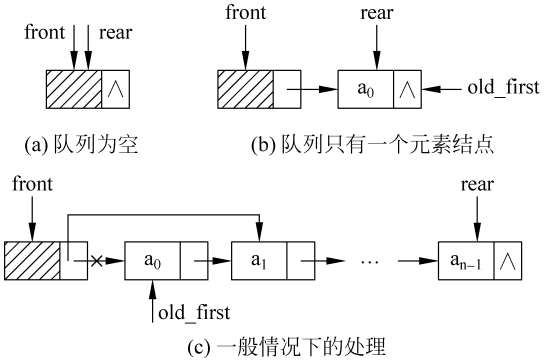


图 5.8　链队列的出队操作

出队算法的完整代码如下：

```
def serve(self):
    if self.empty():
        return None
    else:
        old_first = self._front.next
        self._front.next = old_first.next
        item = old_first.entry
        if self._rear == old_first:
            self._rear = self._front
        del old_first
        return item
```

获取队头元素的方法 retrieve 先判别队列是否为空，若为空时返回 None，非空时返回队首结点的值。

```
def retrieve(self):
    if self.empty():
        return None
    else:
        return self._front.next.entry
```

与链栈一样，除了__len__算法的时间复杂度为 O(n)，链队列的其他算法的时间复杂度都为 O(1)。

实际上，Python 标准库的 queue 模块中提供了 FIFO 队列类 Queue，它采用双向块链存储结构，具体请参考 5.6.3 节和 5.8 节。

5.5　队列的应用

5.5.1　杨辉三角形的输出

杨辉三角形的特点是两个腰上的数字都为1，其他位置上的数字是其上一行中与之相邻（上部和左上部）的两个整数之和。例如，当n＝7时，打印的杨辉三角形如下：

```
1
1    1
1    2    1
1    3    3    1
1    4    6    4    1
1    5   10   10    5    1
1    6   15   20   15    6    1
```

根据杨辉三角形的特点，可用第i－1行的元素来生成第i行的元素。例如，第2行1列和第2行2列的元素都是1，则第3行2列的元素是第2行1列的元素与第2行2列的元素之和2。可以设置一个初始全为0的n＋1行n＋1列的二维列表y，用y[i][j]表示三角形i行j列的元素，则

$$y[i][j]=\begin{cases}1 & i=1,j=1\\ y[i-1][j-1]+y[i-1][j] & i>1,j\geqslant 1,i\geqslant j\end{cases}$$

将y[1][1]赋值为1，通过以上迭代方法进行逐行计算，输出时只选二维列表中的杨辉三角形部分输出即可。该算法较为简单，读者可自行完成，算法的时间和空间复杂度都为$O(n^2)$。

如果用队列依次存放杨辉三角形第i行的所有（i个）元素，然后逐个出队并打印，同时生成第i＋1行的i＋1个元素并入队，重复出队、输出和入队操作，即可得到杨辉三角形。

算法步骤描述如下。

（1）初始化空队列。

（2）1行1列的元素1预先进入队列。

（3）外层循环执行n次，依次处理杨辉三角形的第i行（1≤i≤n）：

① 内层循环执行i次，即依次处理i行j列（1≤j≤i）元素：出队i行j列的元素t，输出t，假设t的前一列元素的值为s（j为1时s为0），求得i＋1行j列的元素s＋t并入队，s的值更新为t。

② 当第i行的i个元素全部出队并输出时，已得到i＋1行的前i个元素并入队，仅需将i＋1行的最后一个元素1入队。

③ 输出第i行的换行符。

具体算法如下：

```
def yang_hui(n):
    line = CircularQueue()              #采用循环队列,也可以用链队列
    line.append(1)                      #第1行的一个数入队
```

```
    for i in range(1, n+1):                  #输出杨辉三角形的 n 行
        #第 i 行数据已存放在队列中,在出队并输出第 i 行的全部元素的同时
        #生成第 i+1 行的全部 i+1 个元素,并入队
        s = 0                                #s 表示 i 行 j-1 列元素的值,初始为 0
        for j in range(1, i + 1):            #对于 i 行 j 列的元素
            t = line.serve()                 #出队
            print("%5d" % t, end="")         #输出
            line.append(s + t)               #生成 i+1 行 j 列的元素并入队
            s = t                            #s 的值更新为 t,用于生成 i+1 行的下一个元素
        line.append(1)                       #i+1 行的最后一个数 1 入队
        print()                              #换行
```

这是利用队列先进先出的特性,控制元素按照一定次序动态生成和输出的队列的典型应用,以后在二叉树层次遍历等算法中会多次遇到类似应用。本算法的时间复杂度为$O(n^2)$,空间复杂度为$O(n)$。

5.5.2 一元多项式的计算

1. 一元多项式的逻辑结构

一元多项式由若干非零项(term)构成,每个非零项由一对系数和指数共同确定,因此一个多项式可以看成若干系数、指数二元组构成的线性表。例如,$P(x)=-8x^{999}-2x^{12}+7x^3$ 可以表示为线性表((-8,999),(-2,12),(7,3))。注意,为了方便对多项式的处理,通常将多项式的非零项按指数递减或递增顺序排序。因此,多项式是一个有序线性表,表中的每个元素包含系数和指数两部分。

假设对上述用线性表表示的多项式进行加法、乘法等运算。例如,对多项式 A 和 B 求和,得到多项式 C,即通过 A 表和 B 表生成 C 表。很显然,当每次生成 C 的一个新项时,都是添加到 C 表的尾部,而当多项式 A 或多项式 B 的某一项被取出运算时,可将这一项从表首位置删除。因此,对相应线性表的操作方式是在头部删除、尾部插入,可以认为多项式线性表是一个队列。

2. 一元多项式的存储结构

在对两个多项式做加、减或乘等运算时,生成的目标多项式的项数事先无法确定,因此选用动态的链式存储结构会更加方便。在此将多项式类定义为链队列类的子类。

图 5.9 给出了多项式 $5x^{17}+9x^8+3x+7$ 的存储结构示意图。

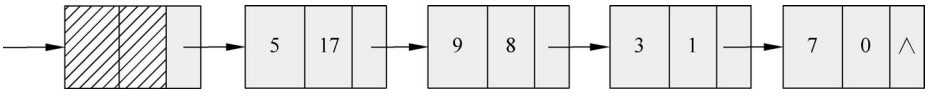


图 5.9 多项式的存储结构

3. 非零项 Term 类的定义

表示多项式非零项的 Term 类定义如下:

```
class Term:
    def __init__(self, scalar = 0.0, exponent = 0):
        self.coefficient = scalar          #非零项的系数
        self.degree = exponent             #非零项的指数
```

4. 多项式 Polynomial 类的定义

1）类定义及初始化方法

```
from LinkedQueue import LinkedQueue
class Polynomial(LinkedQueue):
    def __init__(self):
        super().__init__()
```

2）多项式清空方法

```
def clear(self):
    while not self.empty():
        self.serve()
```

3）求多项式最高次项的指数

exp 方法用于求解当前多项式最高次项的指数。例如，对于图 5.9 表示的多项式，返回 17；对于零多项式，即返回－1。

```
def exp(self):
    if self.empty():
        return -1
    term = self.retrieve()     #多项式按非零项指数递减有序,首项即对应最高次项
    return term.degree
```

4）多项式的读入

```
def read(self):
    print("请按指数递减序,一行输入一对系数和指数,输入完毕以#结束")
    data = input()
    while data != "#":
        temp = data.split()
        t = Term(float(temp[0]), int(temp[1]))
        self.append(t)
        data = input()
```

5）多项式的复制

```
def copy(self):
    r = Polynomial()
    q = self._front.next
    while q:
```

```
            t = Term(q.entry.coefficient, q.entry.degree)
            r.append(t)
            q = q.next
        return r
```

6）多项式的输出

print_out 对链队列进行一次遍历，依次输出每个结点所表示的非零项，根据该项是否为首项、系数的值、系数的正负以及指数的值的不同情况进行不同格式的输出。在输出每个非零项时，变元 X 和其后非 0 且非 1 的指数之间用“^”分隔，例如 3.0X^5。

```
def print_out(self):
    print_node = self._front.next
    first_term = True
    while print_node is not None:
        print_term = print_node.entry
        #以下 if…else 语句输出当前项的符号
        if first_term:                          #避免打印首项的"+"号
            first_term = False
            if print_term.coefficient < 0:
                print("-", end="")
        else:
            if print_term.coefficient < 0:
                print("-", end="")
            else:
                print("+", end="")
        #以下 5 行语句输出当前项系数的绝对值
        r = print_term.coefficient
        if r < 0:                               #保证 r 是系数的绝对值
            r = -r
        if r != 1:                              #系数为 1,无须输出
            print(r, end="")
        #以下 7 行语句输出 X^和指数部分,当指数为 1 或 0 时特别处理
        if print_term.degree > 1:
            print("X^", end="")
            print(print_term.degree, end="")
        if print_term.degree == 1:
            print("X", end="")
        if r == 1 and print_term.degree == 0:
            print("1", end="")
        print_node = print_node.next
    print()
```

5. 多项式的加法、减法和乘法运算

为使算法更加清晰，将多项式的加法等算术运算设计成使用 Polynomial 类的外部函数。

1）多项式的加法

例如，求多项式 first 和 second 的和多项式 result（图 5.10），可概括为如下步骤。

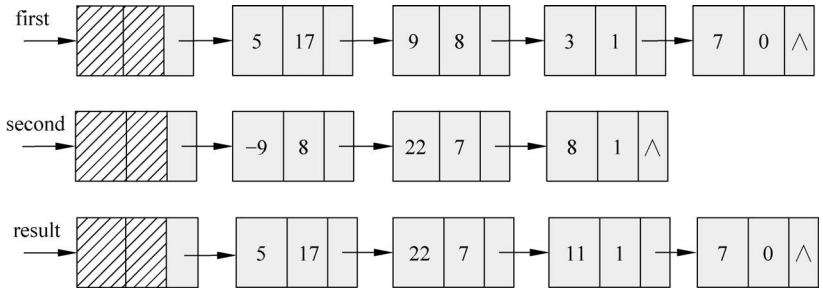


图 5.10　多项式的加法示例

当链队列 first 与 second 至少有一个非空时，循环执行：

（1）分别调用 first.exp()和 second.exp()获得 first 和 second 当前最高次项的指数 exp1 和 exp2。

（2）若 exp1＝exp2，则分别出队 first 链队列中的首项 p 和 second 链队列中的首项 q，如果 p 项和 q 项的系数之和非 0，则生成合并项添加到 result 的尾部。

（3）若 exp1＞exp2，则出队 first 链队列中的首项 p，生成 p 的复制项添加到 result 的尾部。

（4）若 exp1＜exp2，则出队 second 链队列中的首项 q，生成 q 的复制项添加到 result 的尾部。

以下是 add 函数实现多项式 first 和 second 的加法操作，并返回结果多项式。

```
def add(first, second):
    result = Polynomial()
    while not first.empty() or not second.empty():
        exp1 = first.exp()
        exp2 = second.exp()
        if exp1 == exp2:
            p = first.serve()
            q = second.serve()
            if p.coefficient + q.coefficient != 0:
                t = Term(p.coefficient + q.coefficient, p.degree)
                result.append(t)
        elif exp1 > exp2:
            p = first.serve()
            t = Term(p.coefficient, p.degree)
            result.append(t)
        else:
            q = second.serve()
            t = Term(q.coefficient, q.degree)
            result.append(t)
    return result
```

2）多项式的减法

可调用加法操作完成多项式的减法。在调用 add 函数之前，首先生成一个多项式 third，它的各非零项的系数分别为 second 多项式各非零项系数的相反数。以下是 subtract 函数实现多项式 first 和 second 的减法操作，并返回结果多项式。

```
def subtract(first, second):
    third = Polynomial()
    while not second.empty():
        q = second.serve()
        t = Term(-q.coefficient, q.degree)
        third.append(t)
    return add(first, third)
```

3）多项式与单个非零项的乘法

在介绍多项式乘法之前，首先说明多项式与一个非零项的乘法。例如，求多项式 current 与非零项 t 相乘的结果 result，则依次出队 current 的每项 p，将 p 和 t 的系数相乘、指数相加，以得到的新系数和新指数生成一个新项，依次入队到结果多项式 result 中。

```
def mult_term(current, t):
    result = Polynomial()
    while not current.empty():
        p = current.serve()
        result.append(Term(p.coefficient * t.coefficient,
                           p.degree + t.degree))
    return result
```

4）多项式的乘法

可调用加法操作和上述 mult_term 函数完成多项式的乘法。依次出队 first 多项式中的每项 t，将 t 项与 second 多项式的备份 third 相乘的结果 temp 依次加到结果 result 中。因为在做 mult_term 运算时多项式会逐渐出队而变空，所以不能直接用 second 对象参与 mult_term 运算。这也反映了采用队列表示多项式的一个缺点，即在做各种运算时可能会破坏当前的多项式，读者可思考多项式的其他表示方法。

```
def multiply(first, second):
    result = Polynomial()
    while not first.empty():
        t = first.serve()
        third = second.copy()
        temp = mult_term(third, t)
        result = add(result, temp)
    return result
```

5.5.3 基于队列的迷宫求解

1. 广度优先搜索迷宫求解策略

在第4章中利用栈进行回溯实现了迷宫求解，采用的是回溯法搜索迷宫的策略，接下来介绍一种基于广度优先搜索策略求解迷宫问题的方法。该方法的基本思想如下：

(1) 假设当前到达下标为(i,j)的可通位置P，如P为出口，则找到并输出路径，否则依次探索P位置的东、南、西、北4个邻居位置(假设为P_0、P_1、P_2、P_3)。若该邻居位置已为出口，则输出对应路径，算法结束；若该邻居位置不通或已经走过，则跳过该位置；若所有P_i位置都不通或已经走过，则说明无法从P到达终点，接下来只能通过其他位置继续探索。

(2) 如果P有可通邻居，则依次从P_0、P_1、P_2、P_3(假设都可通)位置开始重复步骤(1)，即在还没有找到终点并且还有未探索位置的情况下依次探索P_0的4个相邻位置P_{00}、P_{01}、P_{02}、P_{03}；P_1的4个相邻位置P_{10}、P_{11}、P_{12}、P_{13}；P_2的4个相邻位置P_{20}、P_{21}、P_{22}、P_{23}；P_3的4个相邻位置P_{30}、P_{31}、P_{32}、P_{33}。如果还没有找到路径并且还有未探索位置，则再按刚才各探索位置P_{ij}的次序探索各可通位置P_{ij}的可通未走过邻居位置，以此类推，直到找到一条路径或者所有可通位置都已探索但仍没有找到路径为止。

如图5.11所示，从位置(i,j)开始迷宫探索，依次探测其东、南、西、北4个相邻位置(A、B、C、D)；接着从东邻位置A(i,j+1)探测它的相邻位置，由于其西邻居位置已经走过无须探索，所以依次探测其东、南、北3个相邻位置(E、F、G)；然后从南邻位置B(i+1,j)探测它的相邻位置，由于东、北邻居位置已经走过无须探索，所以依次探测其南、西两个相邻位置(H、I)；接着从西邻位置C(i,j−1)探测它的未走过的西、北两个相邻位置(J、K)；最后从北邻位置D(i−1,j)探测它的未走过的北邻位置(L)；再依次探测这8个位置的未走过的各个相邻位置，在这个过程中遇到出口位置则求解结束。如果所有可通位置全部探测完毕但仍没有遇到出口，则整个迷宫没有可通的路径。在以上描述中，假设各个位置都是可通的，若某位置不可通，则直接跳过该位置。

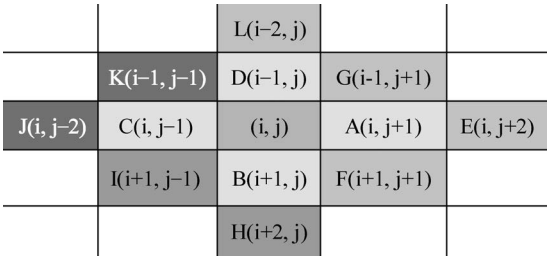


图5.11 从(i,j)位置开始探测迷宫示意图

可以用如图5.12所示的树结构图表示从(i,j)位置开始的迷宫探索过程。树中的每个方框称为结点，在此表示一个位置，带箭头的线段连接上、下两个结点P和Q，表示从P可以走到其邻居位置Q，在树结构中，称Q是P的孩子(请参考第9章)。由于每个位置最多有东、南、西、北4个可通的相邻位置，所以在该树中每个结点最多有4个孩子，实际

情况是由于部分孩子结点已经走过或不可通，某结点的孩子结点经常少于 4 个。从第 0 层的根结点开始探测，然后依次是第 1 层从左到右的 4 个位置，接着是第 2 层从左到右的各个未探测位置；以此类推，即按照从上到下、从左到右的次序对可通且未走过的位置进行探索，当遇到出口或已没有可通位置时求解结束。图中各结点旁标注的编号即是探测的次序。若第 2 层的 8 个有效结点都已探测但还没找到出口，则继续探测第 2 层结点的各个未探测可通邻居位置，图中由于篇幅关系，后续层次没有画出。

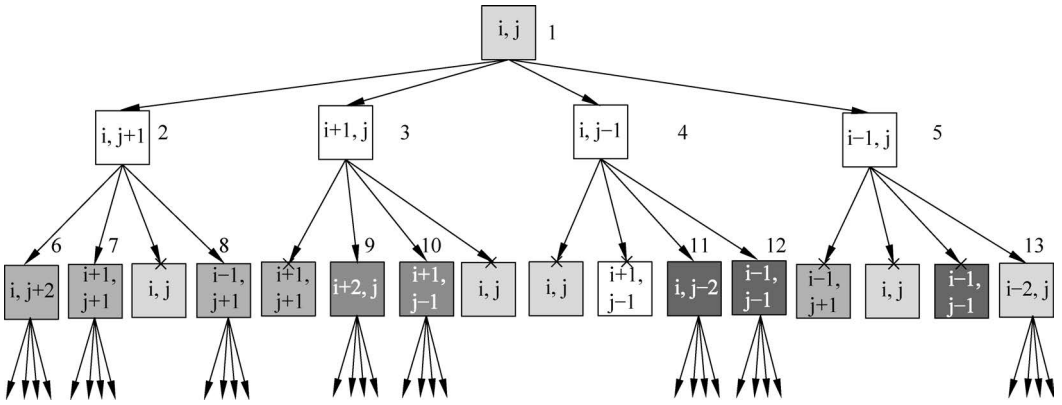


图 5.12　从(i,j)位置开始探测迷宫示意图

2. 算法步骤

类似于杨辉三角形打印，可以利用队列先进先出的特性得到这个从上到下、从左到右的次序。

为了得到第 i+1 层的每个位置的坐标，可以在队列中依次存放第 i 层的所有可通位置的坐标；然后逐个出队各位置 pos，同时得到 pos 位置的各个未走过可通相邻位置并入队，当 i 层位置全部出队时，第 i+1 层的位置则全部入队。重复出队和入队操作，直到遇到出口或队列为空。若遇到出口，则输出迷宫路径。另外，采用一个字典 precedent 记录每个可通位置的前趋位置，以方便最后输出路径。算法步骤如下。

(1) 若入口位置不通，找不到路径，输出相应信息，返回。

(2) 若入口位置即为出口位置，输出相应信息，返回。

(3) 初始化字典 precedent；小乌龟移动到入口位置，对该位置做已走过标记(黑色小圆点)；将入口位置放入初始为空的队列 q 中。

(4) 当队列 q 非空时循环(外层)执行：

① 出队当前位置 pos。

② 小乌龟移动到该位置(其运动轨迹不显示)。

③ 循环(内层)检查 pos 的 4 个相邻位置 nextPos，如果 nextPos 可通且未走过，则

- 小乌龟从 pos 移动到 nextPos，同时显示出其运动轨迹，并对 nextPos 位置做已走过标记(黑色圆点)。
- 若 nextPos 为出口，则调用 buildPath 方法显示出路径，算法结束。

- 否则，将 nextPos 位置入队，同时在 precedent 字典中设置 nextPos 的前趋为 pos，并且为了能在可视化界面中看清小乌龟将从 pos 走向下一个位置，此时将小乌龟再次定位到 pos 位置。

(5) 若队列为空，则没有找到路径。

3. 算法实现

根据上述分析，设计 Maze 类的基于队列的迷宫求解方法 findRouteByQueue。算法如下：

```
def findRouteByQueue(self):
    pos = self.startPosition
    if self[pos[0]][pos[1]] == OBSTACLE:
        print("入口不通")
        return
    if self.isExit(pos):                                #起点即为终点
        print("入口即出口")
        return [pos]
    q = CircularQueue()                                 #建立辅助队列 q
    precedent = dict()                                  #创建各位置的前趋字典 precedent
    #小乌龟移动到 pos 位置，做黑色小圆点标记
    self.updatePosition(pos[0], pos[1], TRIED)
    q.append(pos)                                       #将起点入队
    while not q.empty():                                #队列非空时
        pos = q.serve()                                 #出队一个位置
        self.gotoStart(pos[0], pos[1])                  #小乌龟移动到 pos 位置，不显示轨迹
        for i in range(4):                              #依次试探它的 4 个邻居 nextPos
            nextPos = (pos[0] + DIRECTIONS[i][0],
                       pos[1] + DIRECTIONS[i][1])
            if self.isPassable(nextPos):                #如果该邻居可通
                #小乌龟移动到 nextPos 位置，做黑色小圆点标记
                self.updatePosition(nextPos[0], nextPos[1], TRIED)
                if self.isExit(nextPos):                #若已经到达出口
                    precedent[nextPos] = pos            #出口位置的前趋位置为 pos
                    return self.buildPath(precedent)    #完成路径输出，返回
                q.append(nextPos)                       #该邻居位置入队
                precedent[nextPos] = pos                #该邻居的前趋位置设为 pos
                self.gotoStart(pos[0], pos[1])          #小乌龟回到 pos，不显示轨迹
    print("没有找到通过迷宫的路径")
```

4. Maze 类的其他方法

在上述迷宫求解算法中，遇到出口时调用 buildPath 方法，该方法动态显示并返回迷宫路径。具体算法如下：

```
def buildPath(self, precedent):
    """根据 precedent 中的位置前趋信息，产生起点到终点的路径存储到列表 path 中，
```

```
        同时将路径上的各位置做灰色大圆点标记"""
        start = self.startPosition
        end = self.endPosition
        path = [end]
        pos = end
        while pos != start:
            #对路径上的各位置做灰色大圆点标记
            self.updatePosition(pos[0], pos[1], PATH_PART)
            path.append(pos)
            pos = precedent[pos]
        path.append(start)                                    #此时得到的是 end 到 start 的路径
        self.updatePosition(start[0], start[1], PATH_PART) #起点位置做灰色大圆点标记
        self.wn.exitonclick()
        path.reverse()                                        #列表逆置
        return path
```

Maze 类的数据成员和其他方法与 4.5.4 节相同。对图 4.9 所示的迷宫,采用广度优先搜索路径的算法,从起点(1, 4)到终点(10, 10)的路径搜索过程如图 5.13 所示。其中,灰色大圆点连接的路径为迷宫路径。

扫一扫

视频讲解

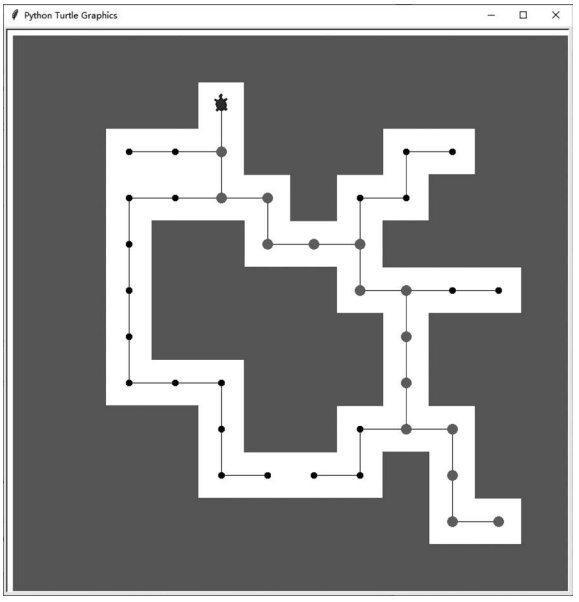


图 5.13 用队列实现迷宫求解的结果截图

5.6 双端队列

5.6.1 双端队列的基本概念

5.1 节中介绍的队列只允许在表的一端进行删除,在另一端进行插入。接下来介绍

一个类队列结构，它支持在线性表的两端进行插入和删除，这就是**双端队列**（double-ended queue 或者 deque，其发音为 deck）。图 5.14 为双端队列示意图，可以认为双端队列是栈和队列的泛化，栈和队列是双端队列的特例。

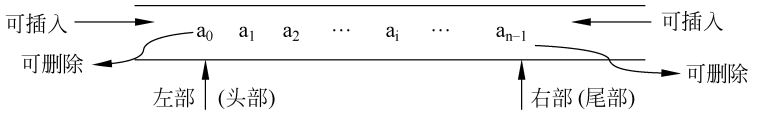


图 5.14 双端队列示意图

T 类型元素构成的双端队列是由 T 类型元素构成的有限序列，并且具有以下基本操作。

（1）构造一个空双端队列（__init__）。

（2）判断一个双端队列是否为空（empty）。

（3）求双端队列的长度（__len__）。

（4）在双端队列的尾部入队一个元素（append）。

（5）在双端队列的头部入队一个元素（appendleft）。

（6）在双端队列的尾部出队一个元素（pop）。

（7）在双端队列的头部出队一个元素（popleft）。

（8）读取双端队列的头部元素（getleft）。

（9）读取双端队列的尾部元素（getright）。

双端队列可以用顺序存储或链式存储方式进行存储，读者可以自行尝试用 Python 的 list 或链表等多种方式实现双端队列。

5.6.2 Python 的双端队列类

在 Python 的标准模块 collections 中提供了一个双端队列类 deque，在 Python 内部实现时采用双向块链表结构，即在双向链表中每个结点的数据域中存放多个元素，此时结点被称为块（block）。Python 双端队列的块结构如图 5.15 所示，data 是一个长度为 64 的数组，可存放 64 个元素（64 个 Python 对象的指针），leftlink 指针指向前趋块，rightlink 指针指向后继块。

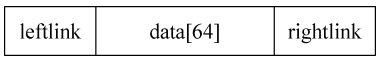


图 5.15 Python 双端队列的块结构

图 5.16 所示为一个含有多个块的非空 deque 对象，它是一个双向非循环链表，主要包含 leftblock、rightblock、leftindex 和 rightindex 等数据成员。其中，在 leftblock 指针指示的块中，data[leftindex]位置的元素对应于图 5.14 中的 a_0；在 rightblock 指针指示的块中，data[rightindex]位置的元素对应于图 5.14 中的 a_{n-1}。

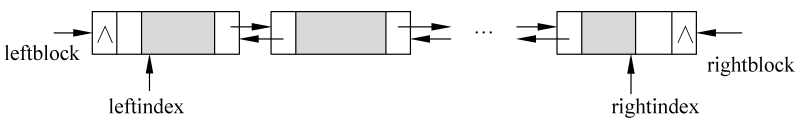


图 5.16 含有多个块的非空双端队列示意图

当生成一个初始空双端队列时即生成一个块，leftblock 和 rightblock 指针都指向该块，并且初始 leftindex 和 rightindex 为 data 数组的中间相邻位置，如图 5.17 所示。此时，如果在双端队列尾部入队(append)，则 rightindex 增 1，将元素放在 data[rightindex]位置；如果在双端队列头部入队(appendleft)，则 leftindex 减 1 后将元素放在该位置。当该块的 64 个空间用完后，若需要尾部入队(append)，则再生成新块插入 rightblock 的右边并调整 rightblock；若需要头部入队(appendleft)，则再生成新块插入 leftblock 的左边并调整 leftblock。

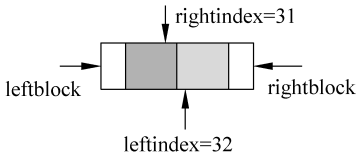


图 5.17　初始空双端队列示意图

总之，deque 对象的数据存储在长度固定的块构成的双链表中，并具有以下优点。

(1) 从基本操作的时间性能来看，双向块链表结构可以确保不管做哪个方向的 append 或 pop 操作，除了被操作的数据元素之外，任何其他元素都不会被移动，因此 append、appendleft、pop 和 popleft 算法的时间复杂度为 O(1)。

(2) 与顺序存储实现相比，链表结构可以完全避免顺序结构在初始分配空间用完时需重新分配更大空间并进行元素复制(类似于 resize 方法的功能)的问题，从而提高了性能的可预测性。

(3) 与 3.4.3 节中介绍的双向链表的实现相比，原双向链表的每个结点中存储一个数据元素和两个指针，存储密度为 1/3，并且每次元素的插入或删除都对应于一次结点的动态分配或回收(对应于底层实现时 C 语言的 malloc()和 free()函数)。而使用固定长度的块，元素的存储密度得到提高，同时避免了频繁调用 malloc()和 free()函数，提高了时间效率。

虽然 list 对象也支持两端的插入和删除操作，但由于采用顺序存储方案，其 pop(0)和 insert(0, v)操作会产生 O(n)内存移动成本，且会改变其他数据的位置。因此，在使用 Python 编写程序时，若需在表的两端做插入或删除，应优先选择 deque 而不是 list。Python 的 deque 对象支持的主要方法如表 5.2 所示。

表 5.2　Python 的 deque 对象支持的主要方法

方　法	说　明
__len__()	返回元素个数
clear()	删除 deque 中的所有元素，让它的长度为 0
append(x)	在 deque 的右边插入 x
appendleft(x)	在 deque 的左边插入 x
pop()	从 deque 的右侧删除并返回一个元素，如果不存在元素，则引发一个 IndexError
popleft()	从 deque 的左侧删除并返回一个元素，如果不存在元素，则引发一个 IndexError
[j]	索引访问，在两端都是 O(1)，但在中间减慢到 O(n)，对于快速随机访问，建议使用列表代替

5.6.3　双端队列的应用

基于上述双端队列类 deque，可以实现 4.2 节介绍的栈的抽象数据类型和 5.2 节介

绍的队列的抽象数据类型。

以下用 Python 的双端队列 deque 实现普通队列类，入队算法调用 deque 的 append 方法，出队算法则调用 deque 的 popleft 方法，类定义和各方法的具体实现如下：

```
from collections import deque
class Queue:
    def __init__(self):
        self._entry = deque()

    def empty(self):
        return len(self._entry) == 0

    def __len__(self):
        return len(self._entry)

    def append(self, value):
        return self._entry.append(value)

    def serve(self):
        if not self.empty():
            return self._entry.popleft()
        else:
            return None

    def retrieve(self):
        if not self.empty():
            return self._entry[0]
        else:
            return None

    def clear(self):
        return self._entry.clear()
```

如 5.4 节中所述，Python 标准库 queue 模块中的 Queue 类即采用双端队列实现。

5.7 优先级队列

5.1 节讨论的队列是一种特征为 FIFO 的数据结构，每次从队列中取出的是最早加入队列中的元素。但是，许多应用需要每次从队列中取出具有最高优先级的元素，这种队列就是**优先级队列**(priority queue)，也称为**优先权队列**或**优先队列**。

优先级队列是 0 个或多个元素的集合，每个元素都有一个与之关联的优先级。对于优先级队列，主要的操作如下。

(1) 查找优先级最高的值。

(2) 出队优先级最高的值。

(3) 入队一个任意优先级的值。

假设数值越小，表明该元素的优先级越高，则在如图 5.18 所示的优先级队列中，最先出队的是元素 10，入队新元素则可以直接放在 30 之后。

假设用无序表表示优先级队列，入队操作的时间复杂度可以为 O(1)，但查找和出队

操作都为 O(n)。如果用有序表表示优先级队列,查找和出队操作的效率可以为 O(1),但入队操作的时间复杂度为 O(n)。在第 8 章中将会介绍用“堆”实现优先级队列,此时入队和出队操作的时间效率都为 $O(\log_2 n)$。

优先级	20	50	40	10	30

图 5.18 优先级队列示例

5.8 Python 提供的多种队列

在 Python 标准库的 queue 模块中提供了 FIFO 队列类 Queue、LIFO 队列类 LifoQueue、优先级队列类 PriorityQueue; 另外,标准库 collections 模块中提供了双端队列类 deque。当然,Python 标准库中各个类的功能比我们所定义的抽象数据类型中的功能更强大,例如队列模块实现多生产者、多消费者队列,该模块中的 Queue 类可以在多个线程之间安全地交换信息,在多线程编程中特别有用。

从数据结构的逻辑结构和存储结构对这 4 个类进行分析,可以得出表 5.3 所列的对应关系。其中,LIFO 队列类 LifoQueue 对象的操作方法与栈相同。

表 5.3 Python 中的各队列及对应数据结构

类	引入语句	逻辑结构	存储结构
LifoQueue	from queue import LifoQueue	栈	顺序表,用 list 实现
Queue	from queue import Queue	队列	双向块链结构,用 deque 实现
PriorityQueue	from queue import PriorityQueue	优先级队列	二叉堆,用 heapq 实现,详见 8.5 节
deque	from collections import deque	双端队列	双向块链结构

5.9 上机实验

扫一扫

上机实验

习题 5

扫一扫

习题

扫一扫

自测题