

的任意3个连成一行、一列或者成对角线，则该玩家胜出。

这显然不是一个复杂的位置型游戏，甚至都没太大意思，因为一个不错的玩家O总能打成平局。三连棋游戏的可取之处在于该游戏能作为一个不错的简单例子来展示二维数组是如何运用于位置型游戏的。开发更复杂的位置型游戏，例如西洋棋、国际象棋或者流行的模拟游戏都是基于相同的方法，即此处论述的为三连棋游戏使用二维数组。

3×3 方格的代表是字符型列表的列表，'X'或'O'表明玩家的走法，" "表示空位置。例如，方格样例为

内部存储为

```
[['O', 'X', 'O'], [' ', 'X', ' '], [' ', 'O', 'X']]
```

我们编写了一个完整的Python类，是有两个玩家的三连棋方格。该类追踪了玩家走法并且宣布赢家，但它并不执行任何策略或允许其他人和计算机对弈三连棋。该程序细节虽超出了本章范围，但仍不失为一个好的课题项目（见练习P-8.68）。

给出该类的实现方法前，在代码段5-12中，我们使用一个简单的测试来展示它的公开接口。

代码段 5-12 对三连棋类的一个简单测试

```

1 game = TicTacToe()
2 # X moves:           # O moves:
3 game.mark(1, 1);    game.mark(0, 2)
4 game.mark(2, 2);    game.mark(0, 0)
5 game.mark(0, 1);    game.mark(2, 1)
6 game.mark(1, 2);    game.mark(1, 0)
7 game.mark(2, 0)
8
9 print(game)
10 winner = game.winner()
11 if winner is None:
12     print('Tie')
13 else:
14     print(winner, 'wins')

```

该类的基本操作为：一个新的游戏实例表示一个空的方格，`mark(i, j)`方法为当前玩家在指定的位置写入标号（用软件掌控轮流次序），该游戏方格能被打印并且由胜利者决定。代码段5-13给出了三连棋类的完整代码。`mark`方法执行错误检测以确保输入索引合法、位置没有被占用，并且当玩家已赢得比赛后，双方都不可再有进一步的动作。

代码段 5-13 管理三连棋游戏的完整 Python 类

```

1 class TicTacToe:
2     """Management of a Tic-Tac-Toe game (does not do strategy)."""
3
4     def __init__(self):
5         """Start a new game."""
6         self._board = [ [' '] * 3 for j in range(3) ]
7         self._player = 'X'
8
9     def mark(self, i, j):
10        """Put an X or O mark at position (i,j) for next player's turn."""
11        if not (0 <= i <= 2 and 0 <= j <= 2):
12            raise ValueError('Invalid board position')
13        if self._board[i][j] != ' ':
14            raise ValueError('Board position occupied')
15        if self.winner() is not None:
16            raise ValueError('Game is already complete')

```

O	X	O
	X	
O		X

```

17     self._board[i][j] = self._player
18     if self._player == 'X':
19         self._player = 'O'
20     else:
21         self._player = 'X'
22
23 def _is_win(self, mark):
24     """Check whether the board configuration is a win for the given player."""
25     board = self._board                         # local variable for shorthand
26     return (mark == board[0][0] == board[0][1] == board[0][2] or      # row 0
27             mark == board[1][0] == board[1][1] == board[1][2] or      # row 1
28             mark == board[2][0] == board[2][1] == board[2][2] or      # row 2
29             mark == board[0][0] == board[1][0] == board[2][0] or      # column 0
30             mark == board[0][1] == board[1][1] == board[2][1] or      # column 1
31             mark == board[0][2] == board[1][2] == board[2][2] or      # column 2
32             mark == board[0][0] == board[1][1] == board[2][2] or      # diagonal
33             mark == board[0][2] == board[1][1] == board[2][0])        # rev diag
34
35 def winner(self):
36     """Return mark of winning player, or None to indicate a tie."""
37     for mark in 'XO':
38         if self._is_win(mark):
39             return mark
40     return None
41
42 def __str__(self):
43     """Return string representation of current game board."""
44     rows = ['|'.join(self._board[r]) for r in range(3)]
45     return '\n-----\n'.join(rows)

```

5.7 练习

请访问 www.wiley.com/college/goodrich 以获得练习帮助。

巩固

- R-5.1 完成代码段 5-1 所示的实验，将运行的结果与在代码段 5-2 得出的结果进行比较。
- R-5.2 在代码段 5-1 中，我们通过实验将 Python 列表的长度与其底层内存占用情况做了比较，决定数组大小的顺序需要人工检查程序的输出。重新设计实验，当目前存储被耗尽时，程序仅输出 k 值。例如，在和代码段 5-2 结果保持一致的数组中，你的程序应该输出数组大小的序列为 0, 4, 8, 16, 25, ...
- R-5.3 修改代码段 5-1 所示的实验，证明当元素从 Python 列表中被取出时，列表偶尔会收缩底层数组大小。
- R-5.4 在代码段 5-3 给出的 DynamicArray 类中，`__getitem__` 方法不支持索引为负。更新该方法，使其更符合 Python 列表语义。
- R-5.5 重新证明命题 5-1，假设数组大小从 k 扩大到 $2k$ 需要 $3k$ 个网络硬币，则在摊销工作中，每次增添操作应收费多少？
- R-5.6 在代码段 5-5 中实现了 DynamicArray 类的 `insert` 方法，但效率较低。当改变数组大小时，改变操作需要花费时间把所有元素从旧数组复制到新数组，在随后的插入操作过程中，需要循环移动其中许多元素。对 `insert` 方法进行改进，使得在改变数组大小时，插入操作能将所有元素直接移动到其最终位置，以免循环移动。
- R-5.7 设 A 为数组，其大小 $n \geq 2$ ，包含 $1 \sim n - 1$ 的整数，其中恰有一个整数重复。描述一种快速算法，找到 A 中这个重复的整数。
- R-5.8 对于 Python 的 `list` 类的 `pop` 方法，当使用可变索引作为参数时（与我们在 5.4 节中对 `insert` 方法所采用的做法类似），利用实验，评估其效率。给出类似于表 5-5 的结果。

- R-5.9 解释在代码段 5-11 中本应该得出的改变，以至于能为情报执行凯撒密码，这些情报使用基于字符表的除英语之外的语言编写，比如希腊语、俄语或者希伯来语。
- R-5.10 在代码段 5-11 中，CaesarCipher 类的构造函数能够使用两行主要部分实现，这两行部分即通过把 join 方法和适当的理解语法结合使用，构造的加密前和加密后的字符串。请给出这样的一种实现。
- R-5.11 使用标准控制结构计算 $n \times n$ 数据集中所有编号的和，该数据集用列表的列表来表示。
- R-5.12 描述 python 内置的 sum 函数如何与理解语法相结合来计算 $n \times n$ 数据集中所有编号的和，该数据集用列表的列表来表示。
- 创新**
- C-5.13 在代码段 5-1 中，我们是以空列表开始的。假如在开始时，data 以非空长度被初始化，当底层数组扩大时，会影响值的序列吗？自己做实验，对你看到的有关初始长度和扩大序列间的任意关系给出评论。
- C-5.14 使用 random 模块提供的 shuffle 方法，对一张 python 列表重新排序，使得每种可能的顺序出现的概率相等。请实现这样的函数，可以使用 random 模块提供的 randrange(n) 函数，该函数返回 $0 \sim n - 1$ 的随机数字。
- C-5.15 思考动态数组的一种实现方法，当数组大小已满时，不是将元素复制到扩大 1 倍的数组中（即， $N \sim 2N$ ），而是复制到扩大 $\lceil N/4 \rceil$ 倍的数组中，数组大小从 N 变为 $N + \lceil N/4 \rceil$ 。证明：在这种情况下，连续执行 n 次 append 操作的运行时间为 $O(n)$ 。
- C-5.16 对代码段 5-3 给出的 DynamicArray 类，实现其 pop 方法，删除数组的最后一个元素，每当元素个数小于 $N/4$ 时，将数组大小缩小为原来的一半 (N 为数组大小)。
- C-5.17 正如前面的练习，当动态数组增大或缩小时，证明下述的连续 $2n$ 次操作的运行时间为 $O(n)$ ：在初始为空的数组上执行 n 次 append 操作，随后执行 n 次 pop 操作。
- C-5.18 给出形式证明：假如使用 C-5.16 描述的方法，对初始为空的动态数组连续执行 n 次 append 或 pop 操作，其运行时间为 $O(n)$ 。
- C-5.19 考虑 C-5.16 的一种变化形式，对于大小为 N 的数组，每次当其元素个数严格小于 $N/4$ 时，调整数组大小精确为元素的个数。给出形式证明：对初始为空的动态数组执行任意的连续 n 次 append 或 pop 操作，其运行时间为 $O(n)$ 。
- C-5.20 考虑 C-5.16 的一种变化形式，对于大小为 N 的数组，每次当其元素个数严格小于 $N/2$ 时，调整数组大小精确为元素的个数。证明存在一个连续 n 次操作，其运行时间为 $\Omega(n^2)$ 。
- C-5.21 在 5.4.2 节中，我们给出 4 种不同的方法组成长字符串：①重复连接；②增加一张临时列表，之后合并到该临时列表中；③使用 join 的列表推导式；④使用 join 的生成器理解法。做实验测试这 4 种方法的效率，给出你的发现。
- C-5.22 做实验比较 Python 的 list 类 extend 方法与重复调用 append 方法在完成等量任务时的相对效率。
- C-5.23 在 5.4 节“构造新列表”的讨论基础上，做实验比较 Python 的列表推导式与重复调用 append 方法构造列表之间的相对效率。
- C-5.24 做实验评估 Python 的 list 类 remove 方法的效率，正如我们在 5.4 节中对 insert 方法所做的那样。使用已知值使得每次删除操作要么出现在列表开头或中间，要么出现在尾部。给出类似于表 5-5 的结果。
- C-5.25 语句 data.remove(value) 仅仅删除 Python 的 data 列表中第一次出现的值为 value 的元素。实现 remove_all(data, value) 函数，使其能够在给出的列表中删除所有值为 value 的元素，对拥

有 n 个元素的列表，该函数的最坏运行时间为 $O(n)$ 。注意，并不是说重复调用 `remove` 方法不够高效。

- C-5.26 设 B 为数组，其大小 $n \geq 6$ ，包含 $1 \sim n - 5$ 的整数，恰有 5 个重复元素。给出一个不错的算法，找出 B 中这 5 个重复的整数。
- C-5.27 给出 Python 的 L 列表，该列表包含 n 个正整数，每个正整数用 $k = \lceil \log n \rceil + 1$ 位表示，给出一种运行时间为 $O(n)$ 的方法，该方法发现 k 位的整数不在 L 中。
- C-5.28 讨论：对于上一个问题的每种解决方案，为什么运行时间一定为 $\Omega(n)$ 。
- C-5.29 在数据库中，一个实用的操作为自然连接（natural join）。假如把数据库看作一张列表，该列表拥有许多有序的成对对象，比如 (x, y) 属于数据库 A ， (y, z) 属于数据库 B ，则 A 和 B 的自然连接即为所有有序三元组 (x, y, z) 所组成的列表。描述和分析一种高效算法，该算法能对包含 n 对对象的列表 A 和包含 m 对对象的列表 B 做自然连接。
- C-5.30 当 Bob 想要通过互联网给 Alice 发送一则消息 M 时，他把 M 分解成 n 个数据包（data packet），并按顺序给这些包编号，然后将它们发送到网络中。当数据包到达 Alice 的计算机时，可能已处于无序状态，因此，在确定自己已获得整个消息前，Alice 必须对 n 个包按序重组。假设 Alice 已知道 n 的值，为她描述一种有效方案去做这件事。这种算法的运行时间是多少？
- C-5.31 给出一种方法，在 $n \times n$ 数据集中，使用递归增加所有数，该数据集以列表的列表形式来表示。

项目

- P-5.32 使用 Python 编写一个函数，该函数给出 2 个三维数值型数据集，并以离散方式将它们相加。
- P-5.33 使用 Python 为矩阵类编写一个程序，该程序能够增加和乘以二维数值型数组，假设维数适应于此操作。
- P-5.34 为英语情报执行凯撒加密法编写程序，其中包括大小写字符。
- P-5.35 实现 `SubstitutionCipher` 类，该类的构造函数包含一个由 26 个大写字母以任意顺序组成的字符串，该字符串用于映射加密前字符串（类似于在代码段 5-11 中 `CaesarCipher` 类的 `self._forward` 字符串）。应能由加密前的字符串得到加密后的字符串。
- P-5.36 重新设计 `CaesarCipher` 类，并把它作为上个问题中 `SubstitutionCipher` 类的一个子类。
- P-5.37 设计 `RandomCipher` 类，并把它作为 P-5.35 中 `SubstitutionCipher` 类的一个子类，使得该类的每个实例拥有为其映射的随机排列的字符串。

扩展阅读

数组的基本数据结构属于计算机科学中的民间学说，Knuth 的重要著作《基本算法》^[64] 首次将它们写入了计算机科学文献。

栈、队列和双端队列

6.1 栈

栈是由一系列对象组成的一个集合，这些对象的插入和删除操作遵循后进先出（LIFO）的原则。用户可以在任何时刻向栈中插入一个对象，但只能取得或者删除最后一个插入的对象（即所谓的“栈顶”）。“栈”这个名字来源于自动售货机中用弹簧顶住的一堆盘子的隐喻。在这种情况下，其基本的操作只涉及向这个栈中取盘子或者放盘子。当需要从这个自动售货机中取一个新盘子时，我们“取”出这个栈顶的盘子。当需要向其中添加一个盘子的时候，我们将盘子“压”入栈顶，使其成为新的栈顶。或许一个更加有趣的例子是 PEZ 糖果售卖器，当售卖器的顶部打开时，它将存储在容器里面的糖果从顶部逐个弹出，如图 6-1 所示。栈是一个基本的数据结构。很多应用程序都会用到栈，下面是一些使用堆栈的示例。

例题 6-1：网络浏览器将最近浏览的网址

存放在一个栈中。每次当访问者访问一个新网站时，这个新网站的网址就被压入栈顶。这样，浏览器就可以在用户单击“后退”按钮时，弹出先前访问的网址，以回到其先前访问的网页。

例题 6-2：文本编辑器通常提供一个“撤销”机制以取消最近的编辑操作并返回到先前的文本状态。这个撤销操作就是通过将文本的变化状态保存在一个栈中得以实现的。

6.1.1 栈的抽象数据类型

栈是最简单的数据结构，但它同样也是最重要的数据结构。它们被用在一系列不同的应用中，并且在许多更加复杂的数据结构和算法中充当工具。从形式上而言，栈是一种支持以下两种操作的抽象数据类型（ADT），用 S 表示这一 ADT 实例：

- S.push(e)：将一个元素 e 添加到栈 S 的栈顶。
- S.pop(e)：从栈 S 中移除并且返回栈顶的元素，如果此时栈是空的，这个操作将出错。此外，为了方便，我们定义了以下访问方法：
- S.top()：在不移除栈顶元素的前提下，返回一个栈 S 的栈顶元素；若栈为空，这个操作会出错。
- S.is_empty()：如果栈中不包含任何元素，则返回一个布尔值“True”。

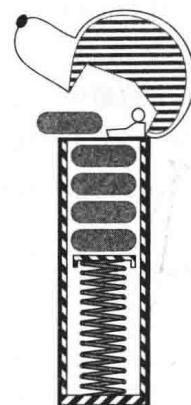


图 6-1 一个 PEZ 糖果售卖器的示意图；一个栈的物理实现过程（PEZ 是派斯糖果的注册商标）

- `len(S)`: 返回栈 S 中元素的数量; 在 Python 中, 我们用 `_len_` 这个特殊的方法实现它。

按照惯例, 我们假定一个新创建的栈是空的, 并且其容量也没有预先的限制。添加进栈的元素可以是任何类型的。

例题 6-3: 下表展示了在一个初始化为整数类型的空栈 S 中进行一系列操作的结果。

操作	返回值	栈的内容
<code>S.push(5)</code>	—	[5]
<code>S.push(3)</code>	—	[5, 3]
<code>len(S)</code>	2	[5, 3]
<code>S.pop()</code>	3	[5]
<code>S.is_empty()</code>	False	[5]
<code>S.pop()</code>	5	[]
<code>S.is_empty()</code>	True	[]
<code>S.pop()</code>	“error”	[]
<code>S.push(7)</code>	—	[7]
<code>S.push(9)</code>	—	[7, 9]
<code>S.top()</code>	9	[7, 9]
<code>S.push(4)</code>	—	[7, 9, 4]
<code>len(S)</code>	3	[7, 9, 4]
<code>S.pop()</code>	4	[7, 9]
<code>S.push(6)</code>	—	[7, 9, 6]
<code>S.push(8)</code>	—	[7, 9, 6, 8]
<code>S.pop()</code>	8	[7, 9, 6]

6.1.2 简单的基于数组的栈实现

我们可以简单地通过在 Python 列表中存储一些元素来实现一个栈。`list` 类已支持 `append` 方法, 用于添加一个元素到列表尾部, 并且支持 `pop` 方法, 用于移除列表中最后的元素, 所以我们可以很自然地将一个列表的尾部与一个栈的顶部相对应起来, 如图 6-2 所示。

虽然程序员可以直接用 `list` 类代替一个正式的 `stack` 类, 但是列表还包括一些不符合这种抽象数据类型的方法 (比如: 增加或者移除处于列表任何位置的元素)。同时, `list` 类所使用的术语也不能与栈这种抽象数据类型的传统命名方法精确对应, 特别是 `append` 方法和 `push` 方法之间的区别。相反, 我们将强调如何使用一个列表实现栈元素的内部存储, 并同时提供一个符合堆栈的公共接口。

适配器模式

适配器设计模式适用于任何上下文, 从而使我们可以有效地修改一个现有的类, 使它的方法能够与那些与其相关但又不同的类或接口相匹配。一个应用这种适配器模式的通用方法是以这样一种方式定义一个新类, 这种方式以包含一个现存类的实例作为隐藏域, 然后用这个隐藏实例变量的方法实现这个新类的方法。以这种方式应用适配器模式, 我们已经创建了一个新类, 它可以执行一些与现有类相同的函数功能, 却以一种更加方便的方式重新封装。对于栈这种抽象数据类型结构, 我们可以通过改编 Python 的 `list` 类中相应的内容来实现, 见表 6-1。

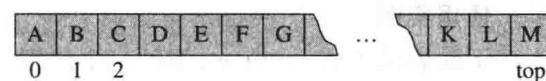


图 6-2 通过 Python 列表实现一个栈, 将其顶部元素存储在最右侧的单元中

表 6-1 通过改编一个 Python 列表 L 实现一个栈 S

栈方法	用 Python 列表实现
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[-1]
S.is_empty()	len(L)==0
len(S)	len(L)

用 Python 的 list 类实现一个栈

我们用适配器设计模式定义了一个 `ArrayStack` 类，并且使用一个基本的 Python 列表进行存储（之所以选择这个 `ArrayStack` 名字，是为了强调底层的存储是基于数组的）。现在还有一个问题，那就是当这个栈是空的时候，如果一个用户调用 `pop` 或者 `top` 方法时，代码应该怎样处理。ADT 给出的建议是触发一个错误，但是必须要决定这是一个什么类型的错误。当在一个空的 Python 列表中调用 `pop` 方法时，正常情况下会触发一个 `IndexError`（请求的索引超出序列范围），因为列表是基于索引的序列。对于栈而言，这个选择似乎并不恰当，因为这里并没有假定的索引。其实，定义一个新的异常类更为恰当。代码段 6-1 定义了这样一个 `Empty` 类作为 Python `Exception` 类的一个子类。

代码段 6-1 Empty 异常类的定义

```
class Empty(Exception):
    """Error attempting to access an element from an empty container."""
    pass
```

我们在代码段 6-2 中给出了 `ArrayStack` 类的正式定义。由于是内部存储，因此构造函数建立 `self._data` 数据成员作为一个初始化的空 Python 列表。余下的公共栈方法将根据表 6-1 中对应的方法实现。

使用实例

下面展示了 `ArrayStack` 类的一个使用实例，映射了例题 6-3 一开始列出的操作。

S = ArrayStack()	# contents: []
S.push(5)	# contents: [5]
S.push(3)	# contents: [5, 3]
print(len(S))	# contents: [5, 3]; outputs 2
print(S.pop())	# contents: [5]; outputs 3
print(S.is_empty())	# contents: [5]; outputs False
print(S.pop())	# contents: []; outputs 5
print(S.is_empty())	# contents: []; outputs True
S.push(7)	# contents: [7]
S.push(9)	# contents: [7, 9]
print(S.top())	# contents: [7, 9]; outputs 9
S.push(4)	# contents: [7, 9, 4]
print(len(S))	# contents: [7, 9, 4]; outputs 3
print(S.pop())	# contents: [7, 9]; outputs 4
S.push(6)	# contents: [7, 9, 6]

代码段 6-2 用 Python 列表作为存储实现一个栈

```
1 class ArrayStack:
2     """LIFO Stack implementation using a Python list as underlying storage."""
3
4     def __init__(self):
5         """Create an empty stack."""
6
7     def push(self, e):
8         self._data.append(e)
9
10    def pop(self):
11        if self.is_empty():
12            raise Empty('Stack is empty')
13        return self._data.pop()
14
15    def top(self):
16        if self.is_empty():
17            raise Empty('Stack is empty')
18        return self._data[-1]
19
20    def is_empty(self):
21        return len(self._data) == 0
22
23    def __len__(self):
24        return len(self._data)
```

```

6     self._data = []                      # nonpublic list instance
7
8     def __len__(self):
9         """Return the number of elements in the stack."""
10        return len(self._data)
11
12    def is_empty(self):
13        """Return True if the stack is empty."""
14        return len(self._data) == 0
15
16    def push(self, e):
17        """Add element e to the top of the stack."""
18        self._data.append(e)                 # new item stored at end of list
19
20    def top(self):
21        """Return (but do not remove) the element at the top of the stack.
22
23        Raise Empty exception if the stack is empty.
24
25        if self.is_empty():
26            raise Empty('Stack is empty')
27        return self._data[-1]               # the last item in the list
28
29    def pop(self):
30        """Remove and return the element from the top of the stack (i.e., LIFO).
31
32        Raise Empty exception if the stack is empty.
33
34        if self.is_empty():
35            raise Empty('Stack is empty')
36        return self._data.pop()           # remove last item from list

```

分析基于数组的栈的实现

表 6-2 展示了 `ArrayStack` 方法的运行时间。这个分析直接与 5.3 节给出的 `list` 类的分析相对应。在最坏的情况下，`top`、`is_empty` 和 `len` 方法均在常量时间内完成。对于 `push` 和 `pop` 操作的时间复杂度为 $O(1)$ ，指的是均摊计算的边界（参见 5.3.2 节）；对于这些方法中任何一个典型的调用都仅需要常量的时间，但是当一个操作导致了列表重新调整其内部数组的大小时，偶尔在最坏的情况下也会要 $O(n)$ 的时间开销，其中 n 是当前栈中元素的个数。对于栈的空间利用率是 $O(n)$ 。

表 6-2 基于数组实现的栈的性能。由于 `list` 类的范围相似，`push` 和 `pop` 操作的时间是摊销的。空间利用率是 $O(n)$ ，其中 n 是当前栈中元素的个数

操作	运行时间
<code>S.push(e)</code>	$O(1)^*$
<code>S.pop()</code>	$O(1)^*$
<code>S.top()</code>	$O(1)$
<code>S.is_empty()</code>	$O(1)$
<code>len(S)</code>	$O(1)$

* 摊销的

避免由于预留空间所导致的摊销

在某些情况下，会有额外的知识表明一个栈将会达到最大的尺寸。从代码段 6-2 看，`ArrayStack` 的实现开始于一个空的列表并且随着需要对它进行扩展。根据 5.4.1 节对列表的

分析，我们相信，在实际中构造一个最初长度为 n 的列表要比一开始就从一个空的列表开始逐步添加 n 项更加有效（即使两种方法均能在 $O(n)$ 时间内运行完毕）。

作为一个栈的替代模型，我们可能希望构造函数接受一个用于指定一个堆栈的最大容量并且初始化数据成员列表的长度。实现这样一个模型需要对代码段 6-2 做出大量改写。栈的长度将不再是列表的长度的同义词，并且对栈的 push 和 pop 操作也不再需要改变列表的长度。相反，我们建议单独维护一个整数作为实例变量以表示当前栈中元素的个数。这个实现过程的细节在课后练习 C-6.17 中展开讨论。

6.1.3 使用栈实现数据的逆置

由于 LIFO 协议，栈可以用作一种通用的工具，用于实现一个数据序列的逆置。例如，如果值 1、2、3 被顺序压入一个栈中，它们将会以 3、2、1 的顺序被逐个弹出。

这一思想可以被应用在各种设置中。例如，我们希望逆序打印一个文件的各行，目的是以降序（而非升序）的方式显示一个数据集。我们可以通过先逐行读出数据，然后压入一个栈中，再按照从栈中弹出的顺序来写入。这个方法的实现过程在代码段 6-3 中给出。

代码段 6-3 一个实现一个文件中各行的逆置函数

```

1 def reverse_file(filename):
2     """Overwrite given file with its contents line-by-line reversed."""
3     S = ArrayStack()
4     original = open(filename)
5     for line in original:
6         S.push(line.rstrip('\n'))      # we will re-insert newlines when writing
7     original.close()
8
9     # now we overwrite with contents in LIFO order
10    output = open(filename, 'w')    # reopening file overwrites original
11    while not S.is_empty():
12        output.write(S.pop() + '\n') # re-insert newline characters
13    output.close()
```

一个值得注意的技术细节是我们在读取时故意将行中的换行符去掉，然后在写入结果文件时重新在每一行中插入换行符。之所以这样做，是为了处理一种特殊的情况，这种特殊情况是在原始文件的最后一行并没有换行符。如果我们只是完全逆置地输出从文件中读取的每一行，那么这个原始文件的最后一行后面将紧跟着倒数第二行而没有新的换行符。我们的实现方法确保了结果中有分离换行符。

使用一个栈来实现数据集的逆置的思想也可以应用在其他类型的序列。例如，练习 R-6.5 就尝试使用栈来实现 Python 列表内容逆置的另一个解决方案（4.4.1 节中讨论了一个递归的解决方案）。一个更具有挑战性的任务是如何将存储在一个栈中的元素逆置。如果将它们从一个栈移到另一个栈中，那么它们将会被逆置，但是如果再次将它们放回原来的栈，那么它们将会再次被逆置，即又回到了最初的顺序。练习 C-6.18 对这个任务的解决方案进行了探索。

6.1.4 括号和 HTML 标记匹配

在本节中，我们将探索两个栈的相关应用，这两个应用都涉及对一串匹配分隔符的测试。在第一个应用中，我们设想算数表达式可能包含几组不同的成对符号，如：

- 小括号：“(” 和 “)”
- 大括号：“{” 和 “}”
- 中括号：“[” 和 “]”

每个开始符号必须与其相对应的结束符号相匹配，例如，一个左中括号 “[” 必须与一个相对应的右中括号 ”]” 相匹配，如表达式 $(5 + x) - (y + z)$ 。下面的例子进一步诠释了这一内容：

- 正确： $(0(0)\{([0])\})$
- 正确： $((0(0)\{([0])\}))$
- 错误： $)0(0)\{([0])\}$
- 错误： $\{([])\}$
- 错误： $($

我们在练习 R-6.6 给出了一组括号匹配的精确定义。

分隔符的匹配算法

在处理算术运算表达式时的一个重要任务是确保表达式中的分隔符匹配正确。代码段 6-4 给出了一个用 Python 实现这一功能的算法。

代码段 6-4 在算数表达式中分隔符匹配算法的函数实现

```

1 def is_matched(expr):
2     """ Return True if all delimiters are properly matched; False otherwise. """
3     lefty = '{[['
4     righty = '}}]'                                # opening delimiters
5     S = ArrayStack()
6     for c in expr:
7         if c in lefty:
8             S.push(c)                                # push left delimiter on stack
9         elif c in righty:
10            if S.is_empty():
11                return False                         # nothing to match with
12            if righty.index(c) != lefty.index(S.pop()):
13                return False                         # mismatched
14    return S.is_empty()                          # were all symbols matched?

```

假定输入的是字符序列如 $(5 + x) - (y + z)$ ，对原始的序列从左到右进行扫描，使用栈匹配这一组分隔符。每次遇到开始符时，我们都将其压入栈中；每次遇到结束符时，我们从栈顶弹出一个分隔符（假定栈不为空），并检查这两个分隔符是否能够组成有效的一对。如果扫描到表达式的最后并且栈为空，则表明原来的算数表达式匹配正确；否则，栈中一定存在一个开始分隔符没有被匹配。

如果原始算数表达式的长度为 n ，这个算法将最多 n 次调用 `push` 和 n 次调用 `pop`。即使假设这些调用的均摊复杂度边界 $O(1)$ ，这些调用总运行时间仍为 $O(n)$ 。对于给定的可能出现的分隔符 ({[，其大小为常量，追加测试如 `lefty` 中的 `c` 和 `righty.index(c)`，其实际运行时间都在 $O(1)$ 之内。结合这些操作，一个序列长度为 n 的匹配算法的运行时间为 $O(n)$ 。

标记语言的标签匹配

另一个分隔符匹配应用是在标记语言（如 HTML 或 XML）的验证中。HTML 是互联网上超文本文档的标准格式，XML 是用于各种数据集的扩展标记语言。图 6-3 所示即为一个 HTML 文件实例和一个可能的对应的翻译。

```

<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>

```

a) 一个 HTML 文件

The Little Boat

The storm tossed the little boat
like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but not
the tree salesman, who even as
a stowaway now felt that he had
overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

b) 它的翻译

图 6-3 HTML 标记的说明

在一个 HTML 文本中，部分文本是由 HTML 标签分隔的。一个简单的 HTML 开始标签的形式为“<name>”，相应的结束标签的则是“</name>”的形式。例如，我们在图 6-3a 的第一行中看到了标签 <body>，并在末尾看到了与其相匹配的标签 </body>。在这个例子中，其他一些经常使用的 HTML 标签如下：

- body：文档内容
- h1：节标题
- center：居中对齐
- p：段落
- ol：编号（命令）列表
- li：表项

理想情况下，一个 HTML 文本应该有相匹配的标记，尽管大多数浏览器能够容忍一定数量的失配标签。代码段 6-5 给出了一个 Python 函数，这个函数实现在一个代表 HTML 文本的字符串中进行标签匹配。我们从左往右扫描原始字符串，用符号 j 来跟踪我们的进度，并且用 str 类的 find 方法来定位定义了这个标签的“<and>”字符。开始标签被压入栈中，当其从栈中弹出时，即与结束标签进行匹配。正如我们在代码段 6-4 中匹配分隔符所做的那样。通过相似的分析，这个算法的运行时间为 $O(n)$ ，其中 n 是这个原始 HTML 文本中字符的数量。

代码段 6-5 测试一个 HTML 文本是否有匹配标签的函数

```

1 def is_matched_html(raw):
2     """Return True if all HTML tags are properly matched; False otherwise."""
3     S = ArrayStack()
4     j = raw.find('<')
5     while j != -1:
6         k = raw.find('>', j+1)
7         if k == -1:
8             return False
9         tag = raw[j+1:k]
10        if not tag.startswith('/'):
11            S.push(tag)
12        else:
13            if S.is_empty():

```

```

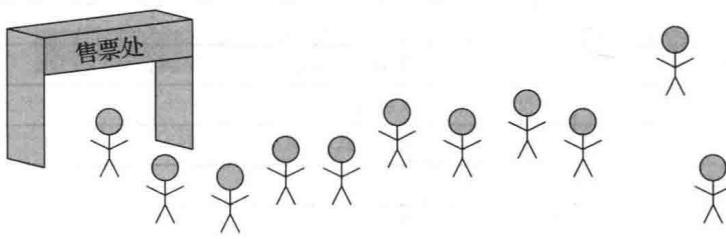
14     return False
15     if tag[1:] != S.pop():
16         return False
17     j = raw.find('<', k+1)
18     return S.is_empty()
19
20     # nothing to match with
21
22     # mismatched delimiter
23     # find next '<' character (if any)
24     # were all opening tags matched?

```

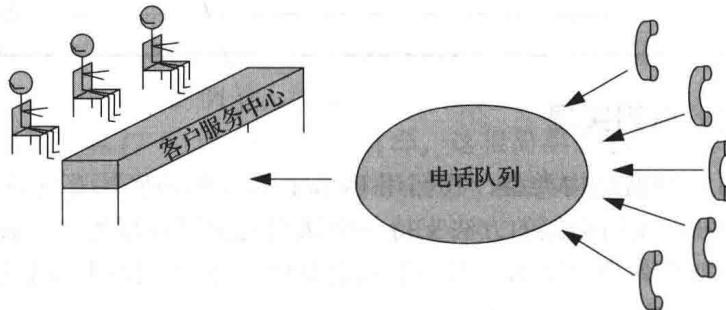
6.2 队列

队列是另一种基本的数据结构，它与栈互为“表亲”关系，队列是由一系列对象组成的集合，这些对象的插入和删除遵循先进先出（First in First out, FIFO）的原则。也就是说，元素可以在任何时刻进行插入，但是只有处在队列最前面的元素才能被删除。

我们通常将队列中允许插入的一端称为队尾，将允许删除的一端则称为队头。对这个术语的一个形象比喻就是一队人在排队进入游乐场。人们从队尾插入排队等待进入游乐场，而从这个队的队头进入游乐场。还有许多其他关于队列的应用，如图 6-4 所示。商店、影院、预订中心和其他类似的服务场所通常按照“先进先出”的原则处理客户的请求。对于顾客服务中心的电话呼叫或者餐厅的等候顾客而言，队列会成为一个合乎逻辑的选择。FIFO 队列还广泛应用于许多计算设备中，比如一个网络打印机或者一个响应请求的 Web 服务器。



a) 人们排队购票



b) 电话被路由到一个客户服务中心

图 6-4 现实世界中一个先进先出的队列实例

6.2.1 队列的抽象数据类型

通常来说，队列的抽象数据类型定义了一个包含一系列对象的集合，其中元素的访问和删除被限制在队列的第一个元素，而且元素的插入被限制在序列的尾部。这个限制根据先进先出原则执行元素的插入和删除操作。对于队列 Q 而言，队列的抽象数据类型（ADT）支持如下两个基本方法：

- $Q.enqueue(e)$: 向队列 Q 的队尾添加一个元素。
- $Q.dequeue()$: 从队列 Q 中移除并返回第一个元素，如果队列为空，则触发一个错误。

队列的抽象数据类型 (ADT) 还包括如下方法 (第一个类似于堆栈的 pop 方法):

- `Q.first()`: 在不移除的前提下返回队列的第一个元素; 如果队列为空, 则触发一个错误。
- `Q.is_empty()`: 如果队列 Q 没有包含任何元素则返回布尔值 “True”。
- `len(Q)`: 返回队列 Q 中元素的数量; 在 Python 中, 我们通过 `_len_` 这个特殊的方法实现。

按照惯例, 假设一个新创建的队列为空, 并且队列的容量没有先天的上限。添加进去的元素也没有任何类型限制。

例题 6-4 : 下表列出了一系列队列的操作和在最初为空的整数类型队列中实施这些操作后的效果。

操作	返回值	<code>first—Q—last</code>
<code>Q.enqueue(5)</code>	—	[5]
<code>Q.enqueue(3)</code>	—	[5, 3]
<code>len(Q)</code>	2	[5, 3]
<code>Q.dequeue()</code>	5	[3]
<code>Q.is_empty()</code>	False	[3]
<code>Q.dequeue()</code>	3	[]
<code>Q.is_empty()</code>	True	[]
<code>Q.dequeue()</code>	“error”	[]
<code>Q.enqueue(7)</code>	—	[7]
<code>Q.enqueue(9)</code>	—	[7, 9]
<code>Q.first()</code>	7	[7, 9]
<code>Q.enqueue(4)</code>	—	[7, 9, 4]
<code>len(Q)</code>	3	[7, 9, 4]
<code>Q.dequeue()</code>	7	[9, 4]

6.2.2 基于数组的队列实现

对于栈这种抽象数据结构类型, 我们用 Python 列表作为底层存储创造了一个非常简单的适配器类, 也可以使用类似的方法支持一个队列的抽象数据类型。我们可以通过调用 `append(e)` 方法将 e 加至列表的尾部。当一个元素退出队列时, 我们可以使用 `pop(0)` 而不是 `pop()` 从列表中来有意移除第一个元素。

由于这个实现很容易, 因此它也最为低效。正如我们在 5.4.1 节讨论的, 当 `pop` 操作在一个列表中以非索引的方式调用时, 可以通过执行一个循环将所有在特定索引另一边的元素转移到它的左边, 目的是为了填补由 `pop` 操作给序列造成的“洞”。因此, 一个 `pop(0)` 操作的调用总是处于最坏的情况, 耗时为 $\Theta(n)$ 。

我们可以改进上面的策略, 完全避免调用 `pop(0)`。可以用一个指代为空的指针代替这个数组中离队的元素, 并且保留一个显式的变量 `f` 来存储当前在最前面的元素的索引。这样一个算法对于离队操作而言耗时为 $O(1)$ 。几次离队操作后, 这个方法可能会导致如图 6-5 所示的情景。

不幸的是, 修改后的方法仍然有一个缺点。

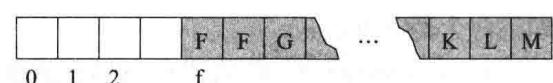


图 6-5 允许队列的前端远离索引 0

在一个栈的设计中，列表的长度就是栈的大小（甚至列表底层的存储数组略大）。对于我们正在考虑的队列的设计，情况更糟。例如，建立一个含有相对较少元素的队列时，系统可能让这些元素存储在一个任意大的列表中。如果不断重复地往一个队列中添加一个新的元素，然后删除另一个（允许最前端向右漂移），就会发生这样的情况，即随着时间的推移，底层列表的大小将逐渐增长到 $O(m)$ ，其中 m 值等于自队列创建以来对队列进行追加元素操作的数量总和，而不是当前队列中元素的数量。

这种设计会在一些所需队列的大小相对稳定却被长时间使用的应用程序中产生不利的影响。例如，餐厅点餐队列的长度在某一个时刻基本上不可能超过 30 个，但是在一天（或者一周），排队的总长度将非常大。

循环使用数组

为了开发一种更加健壮的队列实现方法，我们让队列的前端趋向右端，并且让队列内的元素在底层数组的尾部“循环”。假定底层数组的长度为固定值 N ，它比实际队列中元素的数量大。新的元素在当前队列的尾部利用入队列操作进入队列，逐步将元素从队列的前面插入索引为 $N - 1$ 的位置，然后紧接着是索引为 0 的位置，接下来是索引为 1 的位置。图 6-6 所示为一个第一个元素为 E 最后一个元素为 M 的队列，可用于说明这一过程。



图 6-6 用一个首尾相连的循环数组模拟一个队列

实现这种循环的方法并不困难。当从队列中删除一个元素并欲更新前面的索引时，我们可以使用算式 $f = (f + 1) \% N$ 进行计算。回想一下在 Python 中 % 操作指的是“模”运算操作，它是整数除法之后取余数的值。例如， $14 \div 3$ 整除得到的商为 4 余数为 2，即 $\frac{14}{3} = 4\frac{2}{3}$ 。因此，在 Python 中， $14 // 3$ 得到的结果为 4，而 $14 \% 3$ 的结果为 2。取模操作是处理一个循环数组的理想操作。举一个具体的例子，如果有一个长度为 10 的列表，并且一个索引为 7 的首部，我们可以通过计算 $(7 + 1) \% 10$ 来更新首部，这很简单地就计算出是 8，因为 8 除以 10 商为 0，余数是 8。同样，更新索引为 8 后将会进入索引为 9 的单元。但是当从索引为 9（数组的最后一个单元）处更新时，需要计算 $(9 + 1) \% 10$ ，其结果为得到索引为 0 的位置（因为 10 被 10 整除，余数为 0）。

Python 队列的实现方法

在代码段 6-6 和代码段 6-7 中，我们给出了通过使用 Python 列表以循环的方式来实现一个队列的抽象数据类型的完整方法。其中，这个队列类维护如下 3 个实例变量：

- `_data`：指一个固定容量的列表实例。
- `_size`：是一个整数，代表当前存储在队列内的元素的数量（与 `_data` 列表的长度正好相对）。
- `_front`：是一个整数，代表 `_data` 实例队列中第一个元素的索引（假定这个队列不为空）。

尽管这个队列的大小通常为 0，但我们还是初始化一个可以保存中等大小的列表用于存储数据。同时，我们还将这个队列 `_front` 索引初始化为 0。

当队列为空，`front` 或者 `dequeue` 操作被调用时，系统会抛出一个 `Empty` 异常实例，在代码段 6-1 中，我们为栈定义了这个异常操作。

代码段 6-6 基于数组的队列的实现 (下接代码段 6-7)

```

1 class ArrayQueue:
2     """FIFO queue implementation using a Python list as underlying storage."""
3     DEFAULT_CAPACITY = 10          # moderate capacity for all new queues
4
5     def __init__(self):
6         """Create an empty queue."""
7         self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
8         self._size = 0
9         self._front = 0
10
11    def __len__(self):
12        """Return the number of elements in the queue."""
13        return self._size
14
15    def is_empty(self):
16        """Return True if the queue is empty."""
17        return self._size == 0
18
19    def first(self):
20        """Return (but do not remove) the element at the front of the queue.
21
22        Raise Empty exception if the queue is empty.
23        """
24        if self.is_empty():
25            raise Empty('Queue is empty')
26        return self._data[self._front]
27
28    def dequeue(self):
29        """Remove and return the first element of the queue (i.e., FIFO).
30
31        Raise Empty exception if the queue is empty.
32        """
33        if self.is_empty():
34            raise Empty('Queue is empty')
35        answer = self._data[self._front]
36        self._data[self._front] = None           # help garbage collection
37        self._front = (self._front + 1) % len(self._data)
38        self._size -= 1
39        return answer

```

代码段 6-7 一个基于数组的队列的实现 (上接代码段 6-6)

```

40    def enqueue(self, e):
41        """Add an element to the back of queue."""
42        if self._size == len(self._data):
43            self._resize(2 * len(self._data))      # double the array size
44        avail = (self._front + self._size) % len(self._data)
45        self._data[avail] = e
46        self._size += 1
47
48    def _resize(self, cap):                  # we assume cap >= len(self)
49        """Resize to a new list of capacity >= len(self)."""
50        old = self._data                   # keep track of existing list
51        self._data = [None] * cap          # allocate list with new capacity
52        walk = self._front
53        for k in range(self._size):       # only consider existing elements
54            self._data[k] = old[walk]      # intentionally shift indices
55            walk = (1 + walk) % len(old)  # use old size as modulus
56        self._front = 0                  # front has been realigned

```

限于本书的篇幅，此处省略了 `__len__` 和 `is_empty` 方法的具体实现。`front` 方法的实现也十分简单，因为当假定列表不为空时，`front` 索引能够精确地告诉我们目标元素在 `_data` 列表的什么位置。

添加和删除元素

入队方法的目的是在队列的尾部添加一个新的元素。我们需要确定适当的索引，并将新元素插入对应的位置中。虽然我们没有明确地为队列的尾部信息维护一个实例化变量，但是可以利用下面的公式计算下一个插入的位置：

```
avail = (self._front + self._size) % len(self._data)
```

注意，在插入新元素时，要使用这个队列的大小。例如，考虑一个存储容量为 10 的队列，当前的队列长度为 3，并且第一个元素所在的索引为 5，这个队列中已有的 3 个元素的存储位置即为索引 5、6 和 7，因此，新的元素应该被放置在索引为 $(\text{front} + \text{size}) = 8$ 的位置上。在一个首尾相连的循环队列实例中，利用模运算可以实现这种想要的循环语义。例如，如果假设的队列有 3 个元素并且第一个元素在索引 8 的位置上，我们通过计算 $(8 + 3)\%10$ 得到结果为 1，这样的结果完全正确，因为 3 个现有的元素占据索引为 8、9 和 0 对应的位置。

当调用 `dequeue` 操作时，`self._front` 的当前值指明将要被删除和返回的值的索引。我们为将要返回的元素保存一个本地的引用，在从列表中删除该对象的引用之前，设 `answer=self._data[self._front]`，并设 `self._data[self._front]=None`。设为 `None` 的原因与 Python 回收未使用空间的机制有关。在内部，Python 对已存的对象维护了一个对其的引用计数的计数器。如果计数变为 0，这个对象实际上就无法访问，那么系统会回收这部分的内存以备将来使用（详细内容参见 15.1.2 节）。由于我们不再负责存储一个已经离队元素，因此将从列表中删除该元素的引用以减少这个元素的引用计数。

`dequeue` 操作的第二个重要任务是更新 `_front` 的值以反映元素的移除，并将第二个元素变成新的第一个元素。在大多数情况下，我们可以简单地通过让索引值加“1”更新，但是由于存在环式处理的可能，我们通常是依靠模运算处理，这在本节前面已经有详细的描述。

调整队列的大小

当依次调用 `enqueue` 操作，且队列的大小恰好和底层存储的列表大小相等时，我们可以使用倍增底层列表存储大小的标准技术。通过这种方式，我们可以用与 5.3.1 节实现 `DynamicArray` 方法类似的方式实现这个操作。

然而，在队列中的 `_resize` 方法上，要比在实现 `DynamicArray` 类的相关方法上更加谨慎：在对这个旧列表创建一个临时的引用后，我们分配了一个是原来旧列表 2 倍大小的新列表，并且将引用信息从旧列表复制到新列表中。在传输内容的同时，我们故意在新的数组中将队列的首部索引调整为 0，如图 6-7 所示。这种调整并不单纯为了好看。由于这个模算法依赖于数组的大小，因此将每个元素转移到新的数组并维持与原来相同的索引时，状态将会存在问题。

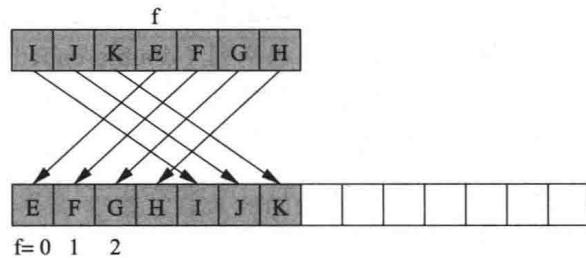


图 6-7 调整队列的大小，同时为新的元素分配 0 号索引

缩减底层数组

队列实现过程中，理想的性能是有 $O(n)$ 的空间复杂度，其中 n 指的是当前队列中元素的个数。正如代码段 6-6 和代码段 6-7 给出的，`ArrayQueue` 的实现并不具备这种属性。当在队列满的状态下调用 `enqueue` 操作时，底层的存储数组就要进行扩展，但是调用出队操作的时候却并不会进行缩减数组的大小的处理。这样处理的结果是，底层的存储数组的大小是队列曾存储的最多元素的个数，而不是当前元素的个数。

我们在 5.3.2 节的动态数组部分讨论过这个问题，在随后的练习 C-5.16 ~ C-5.20 中将继续讨论这个问题。无论什么时候，当所存储的元素降低到数组总存储能力的 $1/4$ 时，一个健壮的方法是将这个数组大小缩减到当前容量的一半。这一处理可以通过在 `dequeue` 方法中插入如下两行代码来实现，只需要追加在代码段 6-6 的第 38 行减少 `self._size` 处理部分之后即可，用于反映一个元素的丢失。

```
if 0 < self._size < len(self._data) // 4:  
    self._resize(len(self._data) // 2)
```

对基于数组的队列实现的分析

在考虑利用上述改进方法来不时缩小数组的大小进行维护队列处理的前提下，表 6-3 列出了基于数组实现队列抽象数据类型的性能。除了 `_resize` 程序，所有的方法都依赖于一个常数数量的算术操作、比较和赋值等语句。因此，除了 `enqueue` 和 `dequeue` 操作是具有均摊复杂度边界为 $O(1)$ ，其余的每一个方法在最坏的情况下运行时间为 $O(1)$ ，其原因与 5.3 节给出的相似。

表 6-3 基于数组实现队列的性能。`enqueue` 和 `dequeue` 操作的时间复杂度边界会因对数组大小重新调整的处理而被均摊。空间利用率为 $O(n)$ ，其中 n 是当前队列中的元素数量

操作	运行时间
<code>Q.enqueue(e)</code>	$O(1)^*$
<code>Q.dequeue()</code>	$O(1)^*$
<code>Q.first()</code>	$O(1)$
<code>Q.is-empty()</code>	$O(1)$
<code>len(Q)</code>	$O(1)$

* 摊销的

6.3 双端队列

接下来考虑一个类队列数据结构，它支持在队列的头部和尾部都进行插入和删除操作。这样一种结构被称为双端队列（double-ended queue 或者 deque），它的发音通常为“deck”，以免与通常的队列抽象数据类型的方法 `dequeue` 相混淆，后者的发音类似于“D.Q”的缩写。

双端队列的抽象数据类型比栈和队列的抽象数据类型要更普遍。在一些应用中，这些额外的普遍性是非常有用的，例如使用一个队列来描述餐馆当中的等餐队列。一般情况下，第一个人会在发现餐馆中没有空闲的桌子时从队列的前面离开，而这个时候餐馆会重新在队列的前面插入一个人。同样，处于队列尾部的顾客也可能由于不耐烦而离开队伍。（如果想模拟顾客从其他位置离开，我们需要一个更加通用的数据结构）

6.3.1 双端队列的抽象数据类型

为了提供一个相类似的抽象，可以定义双端队列的抽象数据类型 D，这个 ADT 支持如

下方法：

- D.add_first(e)：向双端队列的前面添加一个元素 e。
- D.add_last(e)：在双端队列的后面添加一个元素 e。
- D.delete_first()：从双端队列中移除并返回第一个元素。若双端队列为空，则触发一个错误。
- D.delete_last()：从双端队列中移除并返回最后一个元素。若双端队列为空，则触发一个错误。

此外，双端队列的抽象数据类型还包括如下的方法：

- D.first()：返回（但不移除）双端队列的第一个元素。若双端队列为空，则触发一个错误。
- D.last()：返回（但不移除）双端队列的最后一个元素。若双端队列为空，则触发一个错误。
- D.is_empty()：如果双端队列不包含任何一个元素，则返回布尔值“True”。
- len(D)：返回当前双端队列中的元素个数。在 Python 中，我们用 `__len__` 这个特殊的方法实现。

例题 6-5：下表展示了一系列双端队列的操作和它们在一个初始化为整数类型的空双端队列中的效果。

操作	返回值	双端队列
D.add_last(5)	—	[5]
D.add_first(3)	—	[3, 5]
D.add_first(7)	—	[7, 3, 5]
D.first()	7	[7, 3, 5]
D.delete_last()	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last()	3	[7]
D.delete_last()	7	[]
D.add_first(6)	—	[6]
D.last()	6	[6]
D.add_first(8)	—	[8, 6]
D.is_empty()	False	[8, 6]
D.last()	6	[8, 6]

6.3.2 使用环形数组实现双端队列

我们可以使用与代码段 6-6 和代码段 6-7 提供的实现 `ArrayQueue` 类相同的方法来实现双端队列的抽象数据类型（我们把通过 `ArrayQueue` 实现双端队列的细节留在了练习 P-6.3.2 中）。我们建议保持 3 个相同的实例变量：`_data`、`_size` 和 `_front`。无论什么时候，只要想知道双端队列的尾部索引，或者超过队尾的第一个可用的位置，我们就可以通过模运算计算得出。例如，方法 `last()` 就是使用如下索引公式来实现的

```
back = (self._front + self._size - 1) % len(self._data)
```

对于方法 `ArrayDeque.add_last`，我们采用了与方法 `ArrayQueue.add_last` 相同的实现方

法，也利用了一个 `_resize` 程序。类似地，`ArrayDeque.delete_first` 方法的实现与 `ArrayQueue.dequeue` 方法相同。实现 `add_first` 与 `delete_last` 采用了相似的技术，其中的一处不同是，在调用 `add_first` 时需要循环处理数组起始位置，因此我们借助模（取余）运算来循环地计算索引值

```
self._front = (self._front - 1) % len(self._data)      # cyclic shift
```

基于数组的双端队列 `ArrayDeque` 与基于数组的队列 `ArrayQueue` 的效率很相似，所有操作都能在 $O(1)$ 内能完成，但是由于有些操作的时间边界将会摊销，这可能会改变底层数组的大小。

6.3.3 Python collections 模块中的双端队列

Python 的标准 `collections` 模块中包含对一个双端队列类的实现方法。表 6-4 给出了 `collections.deque` 类最常用的方法。这里使用了比之前的抽象数据类型更加不对称的命名。

表 6-4 双端队列的抽象数据类型与 `collections.deque` 类的比较

我们的双端队列 ADT	<code>collections.deque</code>	描述
<code>len(D)</code>	<code>len(D)</code>	元素数量
<code>D.add_first()</code>	<code>D.appendleft()</code>	加到开头
<code>D.add_last()</code>	<code>D.append()</code>	加到结尾
<code>D.delete_first()</code>	<code>D.popleft()</code>	从开头移除
<code>D.delete_last()</code>	<code>D.pop()</code>	从结尾移除
<code>D.first()</code>	<code>D[0]</code>	访问第一个元素
<code>D.last()</code>	<code>D[-1]</code>	访问最后一个元素
	<code>D[j]</code>	通过索引访问任意一项
	<code>D[j]=val</code>	通过索引修改任意一项
	<code>D.clear()</code>	清除所有内容
	<code>D.rotate(k)</code>	循环右移 k 步
	<code>D.remove(e)</code>	移除第一个匹配的元素
	<code>D.count(e)</code>	统计对于 e 匹配的数量

双端队列集合的接口选用与已经建立的 Python 列表类命名约定一致，因为 `pop` 方法与 `append` 方法都被认为是在列表的尾部操作。因此，`appendleft` 和 `popleft` 都指的是在列表的首部操作。库双端队列同样也模仿了一个列表，因为它是一个带索引的序列，允许使用 `D[j]` 的语法任意访问和修改。

库双端队列的构造函数同样支持一个可选的 `maxlen` 参数以建立一个固定长度的双端队列。然而，当双端队列满时，如果在队列的任意一端调用 `append` 方法，它并不会触发一个错误；相反，这会导致在相反一端移除一个元素。也就是说，当队列满时，调用 `appendleft` 方法会导致右端一个隐藏的 `pop` 调用发生，以便为新加入的元素腾出空间。

当前 Python 版本使用了一个混合的方法实现 `collection.deque`，这种方法使用了循环数组，这些循环数组被组合到块中，而这些块本身又被组织进一个双向链表中（我们将在下一章介绍这种数据结构）。双端队列类保证在任何一端操作的耗时为 $O(1)$ ，但在最坏的操作情况下，当使用靠近双端队列中部附近的索引时，耗时将为 $O(n)$ 。

6.4 练习

请访问 www.wiley.com/college/goodrich，以获得练习帮助。

巩固

- R-6.1 如果在一个初始化为空的栈上执行如下一系列操作，将返回什么值？push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop()。
- R-6.2 假设一初始化为空的栈 S 已经执行了 25 个 push 操作、12 个 top 操作和 10 个 pop 操作，其中 3 个触发了栈空错误。请问 S 目前的大小是多少？
- R-6.3 实现一个函数 transfer(S, T) 将栈 S 中的所有元素转移到栈 T 中，使位于 S 栈顶的元素被第一个插入栈 T 中，使位于 S 栈底的元素最后被插入栈 T 的顶部。
- R-6.4 给出一个用于从栈中移除所有元素的递归实现方法。
- R-6.5 实现一个函数，通过将一个列表内的元素按顺序压入堆栈中，然后逆序把它们写回到列表中，实现列表的逆置。
- R-6.6 给出一个算术表达式中分组符号匹配的精确而完整的定义。应保证定义可以是递归的。
- R-6.7 如果在一个初始化为空的队列上执行如下一系列操作后，返回值是什么？enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue()。
- R-6.8 假设一个初始化为空的队列 Q 已经执行了共 32 次入队操作、10 次取首部元素操作和 15 次出队操作，其中 5 次触发了队列为空的错误。队列 Q 目前的大小是什么？
- R-6.9 假定先前所述问题的队列是 `ArrayQueue` 的实例且初始化为 30，并且假定它的大小不会超过 30，那么 `front` 实例变量的最终值是多少？
- R-6.10 试想，如果代码段 6-7 `ArrayQueue.Resize` 方法中的第 53 ~ 55 行执行如下 loop 循环将会发生什么？给出错误的详细解释。

```
for k in range(self._size):
    self._data[k] = old[k] # rather than old[walk]
```

- R-6.11 给出一个简单的适配器实现队列 ADT，其中采用一个 `collections.deque` 实例做存储。
- R-6.12 在一个初始化为空的双端队列中执行以下一系列操作，将会返回什么结果？`add_first(4), add_last(8), add_last(9), add_first(5), back(), delete_first(), delete_last(), add_last(7), first(), last(), add_last(6), delete_first(), delete_first()`。
- R-6.13 假设有一个含有数字 (1, 2, 3, 4, 5, 6, 7, 8) 并按这一顺序排列的双端队列 D ，并进一步假设有一个初始化为空的队列 Q 。给出一个只用 D 和 Q （不包含其他变量）实现的代码片段，将元素 (1, 2, 3, 5, 4, 6, 7, 8) 按这一顺序存储在 D 中。
- R-6.14 使用双端队列 D 和一个初始化为空的栈 S 重复做上一问题。

创新

- C-6.15 假设爱丽丝选择了 3 个不同的整数，并将它们以随机顺序放置在栈 S 中。写一个简短的顺序型的伪代码（不包含循环或递归），其中只包含一次比较和一个变量 x ，使得爱丽丝的 3 个整数中最大的以 2/3 概率存储在变量 x 中，试说明你的方法为什么是正确的。
- C-6.16 修改基于数组的栈的实现方法，使栈的容量限制在最大元素数量 `maxlen` 之内。该最大数量对于构造函数（默认值为 `None`）是一个可选参数。如果 `push` 操作在栈满时被调用，则抛出一个“栈满”异常（与栈空异常定义类似）。

- C-6.17 在之前实现栈的练习中，假设底层列表是空的。重做该练习，此时预分配一个长度等于堆栈最大容量的底层列表。
- C-6.18 如何用练习 R-6.3 中描述的转换函数和两个临时栈来取代一个给定相同元素但顺序逆置的栈。
- C-6.19 在代码段 6-5 中，假设 HTML 的开始标签具有 `<name>` 与 `` 的形式。更普遍的是，HTML 允许可选的属性作为开始标签的一部分。所用的一般格式是 `<name attribute1="value1" attribute2="value2">`；例如，表可以通过使用开始标签 `<table border="3" cellpadding="5">` 被赋予一个边界和附加数据。修改代码段 6-5，使得即使在一个开始标签包含一个或多个这样的属性时，也可以正确匹配标记。
- C-6.20 通过一个栈实现一个非递归算法来枚举 $\{1, 2, \dots, n\}$ 所有排列数结果。
- C-6.21 演示如何使用栈 S 和队列 Q 非递归地生成一个含 n 个元素的集合所有可能的子集集合 T 。
- C-6.22 后缀表示法是一种书写不带括号的算术表达式的简明方法。它是这样定义的：如果 “ $(\text{exp}_1) \text{OP}(\text{exp}_2)$ ” 是一个普通、完整的括号表达式，它的操作符是 OP，那么它的后缀版本为 “ $\text{pexp}_1 \text{ pexp}_2 \text{ OP}$ ”，其中 pexp_1 是 exp_1 的后缀表示形式， pexp_2 是 exp_2 的后缀表示形式。一个单一的数字或变量的后缀表示形式就是这个数字或变量。例如，“ $((5 + 2)*(8 - 3))/4$ ” 的后缀版本为 “ $52 + 83 - *4/$ ”。写出一种非递归方式实现的后缀表达式转换算法。
- C-6.23 假设有 3 个非空栈 R 、 S 、 T 。请通过一系列操作，将 S 中的元素以其原始的顺序存储到 T 中原有元素的后面，最终 R 中元素的顺序不变。例如， $R=[1, 2, 3]$ ， $S=[4, 5]$ ， $T=[6, 7, 8, 9]$ ，则最终的结果应为 $R=[1, 2, 3]$ ， $S=[6, 7, 8, 9, 4, 5]$ 。
- C-6.24 描述如何用一个简单的队列作为实例变量实现堆栈 ADT，在方法体中，只有常量占用本地内存。在你所设计的方法中，`push()`、`pop()`、`top()` 的运行时间分别是多少？
- C-6.25 描述如何用两个栈作为实例变量实现队列 ADT，这样使得所有队列操作的平均时间开销为 $O(1)$ 。给出一个正式的证明。
- C-6.26 描述如何使用一个双端队列作为实例变量实现队列 ADT。该方法的运行时间是多少？
- C-6.27 假设有一个包含 n 个元素的栈 S 和一个初始为空的队列 Q ，描述如何用 Q 扫描 S 来查看其中是否包含某一特定元素 x ，算法必须返回到元素在 S 中原来的位置。算法中只能使用 S 、 Q 和固定数量的变量。
- C-6.28 修改 `ArrayQueue` 实现方法，使队列的容量由 `maxlen` 限制，其中该最大长度对于构造函数（默认为 `none`）来说是一个可选参数。如果在栈满的时候调用 `enqueue` 操作，则触发一个队列满异常（与队列空异常定义类似）。
- C-6.29 在队列 ADT 的某些特定应用中，以某种方式对一个元素反复执行入队出队操作是很常见的。改造基于数组的队列实现方法，加入一个 `rotate()` 操作，这个操作与 `Q.enqueue` 和 `Q.dequeue` 的结合具有相同的语义特征。然而，它的执行效率应当比分别调用两个方法更有效（例如，该方法中不需要修改队列的长度 `_size`）。
- C-6.30 爱丽丝有两个用于存储整数的队列 Q 和 R 。鲍勃给了爱丽丝 50 个奇数和 50 个偶数，并坚持让她在队列 Q 和 R 存储所有 100 个整数。然后他们玩了一个游戏，鲍勃从队列 Q 和 R 中随机选择元素（采用在本章所描述的循环调度，其对于选择队列的次数是随机的）。如果在游戏结束时被处理的最后一个数是奇数，则鲍勃胜。爱丽丝能如何分配整数到队列中来优化她获胜的机会？她获胜的机会是什么？
- C-6.31 假设鲍勃有 4 头牛，他要过一座桥，但只有一个轭，如果牛绑轭肩并肩过桥，一次只能两头牛通过。轭太重了，他无法扛着过桥，但可以立刻将其绑在牛上或从牛身上拆下来。4 头牛中，

Mazie 可以在 2 分钟内过桥，Daisy 可以在 4 分钟内过桥，Crazy 要花 10 分钟，Lazy 则要花 20 分钟。当然，当两只牛拴在一起时，它们必须以走的慢的牛的速度前进。描述鲍勃应该如何带着他的所有牛在 34 分钟内过桥。

项目

- P-6.32 如 6.3.2 节描述的那样，给出一个完整的基于数组的双端队列 ADT 的队列实现方法。
- P-6.33 给定一个基于数组实现双端队列的实现方法，使其支持表 6-4 列出的 `collection.deque` 类的所有公共操作，包括使用 `maxlen` 这个可选参数。当一个限制长度的队列已满时，提供与 `collections.deque` 类相似的语义，使一个调用将一个元素插入双端队列的尾部时造成相反方向一个元素的丢失。
- P-6.34 实现一个程序，可以输入以后缀形式表示的算数表达式（见练习 C-6.22）并且输出它的运算结果。
- P-6.35 6.1 节的介绍表明，栈通常用于在应用程序中提供“撤销”支持，如网络浏览器或文本编辑器。虽然支持撤销可以用无界堆栈来实现，但许多应用程序只使用容量固定的堆栈提供有限步的撤销操作。当压栈操作在满栈时被调用，并不是抛出栈满异常，见练习 C-6.16），一个更典型的语义是接受在顶部压栈的元素，同时在栈底部“漏出”最老的元素来腾出空间。
- P-6.36 当出售一支由若干家公司共享的公共股票时，共享售出价和原始买入价的资本收益（或者，有时候亏损）是不同的。对于单一共享的股票这个规则很容易理解，但如果出售的是一支已经购买了很久的共享股票时，我们必须鉴别这支共享股票是真的在出售。在这个例子中，对于识别哪个共享股票被卖掉的问题，一个标准的判断原则就是采用 FIFO 协议——这支被共享出售的股票，往往是那些持有时间最长的（实际上，这种默认的方法已被封装到了几款个人投资软件包中）。例如，假设买 100 股共享股票，第一天的价格为每股 20 美元，第二天有 20 股的价格是 24 美元，第三天有 200 股的价格为 36 美元，而在第四天以每股 30 美元的价格卖出 150 股。根据 FIFO 的原则，意味着 150 股被卖掉，第一天买了 100 股，第二天买了 20 股，第三天买了 30 股，因此在这个例子中的资本收益就应该是： $100 \times 10 + 20 \times 6 + 30 \times (-6)$ 或者 940 美元。写一个程序，用于表示形如“以每股 y 美元的价格购买了 x 股 share(s) 股票”或者“以每股 y 美元的价格卖出 x 股 share(s) 股票”的一组事务的序列，假定这些事务发生在连续的几天之内，同时 x 和 y 的值都是整数。当给定了一个输入序列，运用 FIFO 协议来识别共享股票，对应的输出序列应该是整个序列总的资本收益的值（或者资本亏损的值）。
- P-6.37 设计一个两色双向栈 ADT，其中包含两个栈，即一个“红”栈和一个“蓝”栈，并包含和常规模操作一致的有颜色编码的栈操作。例如，在这个 ADT 中，支持一个“红”压栈操作和一个“蓝”压栈操作。给出一种有效的实现方法，即采用一个限定容量为 N 的单个数组来实现这个 ADT，假定 N 值始终大于单个的“红”栈与“蓝”栈大小之和。

扩展阅读

本章先介绍以数据结构的 ADT 来定义数据结构的方法，然后以 Aho、Hopcroft 和 Ullman 等人的经典书籍^[5, 6]中的所给出模式具体实现这些方法。练习 C-6.30 和 C-6.31 与一些著名软件公司经常选用的面试问题非常相似。如果需要进一步学习和了解抽象数据类型，可以参考 Liskov 和 Cardelli 的著作^[7]以及 Wegner^[23]或者 Demurjian^[33]的相关书籍。

链 表

在第 5 章，我们仔细探讨了 Python 的基于数组的 `list` 类。在第 6 章，我们着重讨论使用这个类来实现经典的栈、队列、双向队列的抽象数据类型（Abstract Data Type, ADT）。Python 的 `list` 类是高度优化的，并且通常是考虑存储问题时很好的选择。除此之外，`list` 类也有一些明显的缺点：

- 1) 一个动态数组的长度可能超过实际存储数组元素所需的长度。
- 2) 在实时系统中对操作的摊销边界是不可接受的。
- 3) 在一个数组内部执行插入和删除操作的代价太高。

在本章，我们介绍一个名为链表的数据结构，它为基于数组的序列提供了另一种选择（例如 Python 列表）。基于数组的序列和链表都能够对其中的元素保持一定的顺序，但采用的方式截然不同。数组提供更加集中的表示法，一个大的内存块能够为许多元素提供存储和引用。相对地，一个链表依赖于更多的分布式表示方法，采用称作节点的轻量级对象，分配给每一个元素。每个节点维护一个指向它的元素的引用，并含一个或多个指向相邻节点的引用，这样做的目的是为了集中地表示序列的线性顺序。

我们将对比基于数组序列和链表的优缺点。通过数字索引 k 无法有效地访问链表中的元素，而仅仅通过检查一个节点，我们也无法判断出这个节点到底是表中的第 2 个、第 5 个还是第 20 个元素。然而，链表避免了上面提到的基于数组序列的 3 个缺点。

7.1 单向链表

单向链表最简单的实现形式就是由多个节点的集合共同构成一个线性序列。每个节点存储一个对象的引用，这个引用指向序列中的一个元素，即存储指向列表中的下一个节点，如图 7-1 和图 7-2 所示。

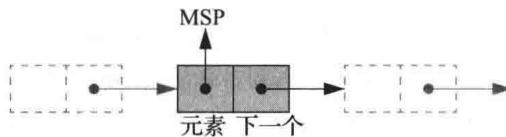


图 7-1 节点实例的示例，用于构成单向链表的一部分。这个节点含有两个成员：元素成员引用一个任意的对象，该对象是序列中的一个元素（在这个例子中，序列指的是机场节点 MSP）；指针域成员指向单向链表的后继节点（如果没有后继节点，则为空）

链表的第一个和最后一个节点分别为列表的头节点和尾节点。从头节点开始，通过每个节点的“`next`”引用，可以从一个节点移动到另一个节点，从而最终到达列表的尾节点。若当前节点的“`next`”引用指向空时，我们可以确定该节点为尾节点。这个过程通常叫作遍历链表。由于一个节点的“`next`”引用可以被视为指向下一个节点的链接或者指针，遍历列表的过程也称为链接跳跃或指针跳跃。

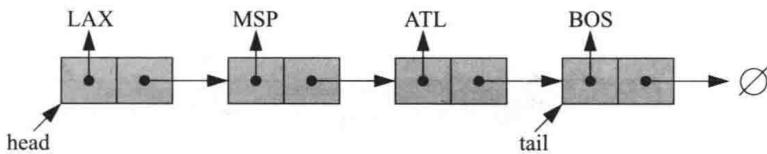


图 7-2 元素是用字符串表示机场代码的单向链表示例。列表实例中维护了一个叫作头节点 (head) 的成员，它标识列表的第一个节点。在某些应用程序中，另有一个叫作尾节点 (tail) 的成员，它标识列表的最后一个节点。空对象被表示为 \emptyset

链表在内存中的表示依赖于许多对象的协作。每个节点被表示为唯一的对象，该对象实例存储着指向其元素成员的引用和指向下一个节点的引用（或者为空）。另一个对象用于代表整个链表。链表实例至少必须包括一个指向链表头节点的引用。没有一个明确的头的引用，就没有办法定位节点（或间接地定位其他任何节点）。没有必要直接存储一个指向列表尾节点的引用，因为尾节点可以通过从头节点开始遍历链表中的其余节点来定位。不管怎样，显式地保存一个指向尾节点的引用，是避免为访问尾节点而进行链表遍历的常用方法。类似地，链表实例保存一定数量的节点总数（通常称为列表的大小）也是比较常见的，这样就可以避免为计算链表中的节点数量而需要遍历整个链表。

在本章的其余部分，我们将继续把节点称为“对象”，而把每个节点指向“next”节点的引用称为“指针”。但是，为简单起见，我们将一个节点的元素直接嵌入该节点的结构中，尽管元素实际上是一个独立的对象。对此，图 7-3 以更简洁的方式展示了图 7-2 的链表。

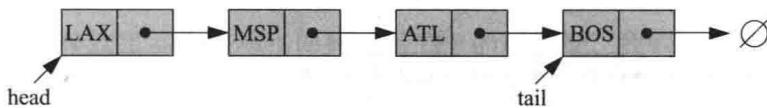


图 7-3 单向链表的一个简洁示例，元素嵌入在节点中（而不是更精确地画为外部对象的引用）

在单向链表的头部插入一个元素

单向链表的一个重要属性是没有预先确定的大小，它的占用空间取决于当前元素的个数。当使用一个单向链表时，我们可以很容易地在链表的头部插入一个元素，如图 7-4 所示，伪代码描述如代码段 7-1 所示。其基本思想是创建一个新的节点，将新节点的元素域设置为新元素，将该节点的“next”指针指向当前的头节点，然后设置列表的头指针指向新节点。

代码段 7-1 在单向链表 L 头部插入一个元素。注意，要在为新节点分配 L.head 变量之前设置新节点的“next”指针。如果初始列表为空（即 L.head 为空），那么就将新节点的“next”指针指向空（None）

Algorithm add_first(L,e):

```

newest = Node(e) {create new node instance storing reference to element e}
newest.next = L.head {set new node's next to reference the old head node}
L.head = newest      {set variable head to reference the new node}
L.size = L.size + 1   {increment the node count}

```

在单向链表的尾部插入一个元素

只要保存了尾节点的引用（指向尾节点的指针），就可以很容易地在链表的尾部插入一个元素，如图 7-5 所示。在这种情况下，创建一个新的节点，将其“next”指针设置为空，并设置尾节点的“next”指针指向新节点，然后更新尾指针指向新节点，伪代码描述如代码段 7-2 所示。

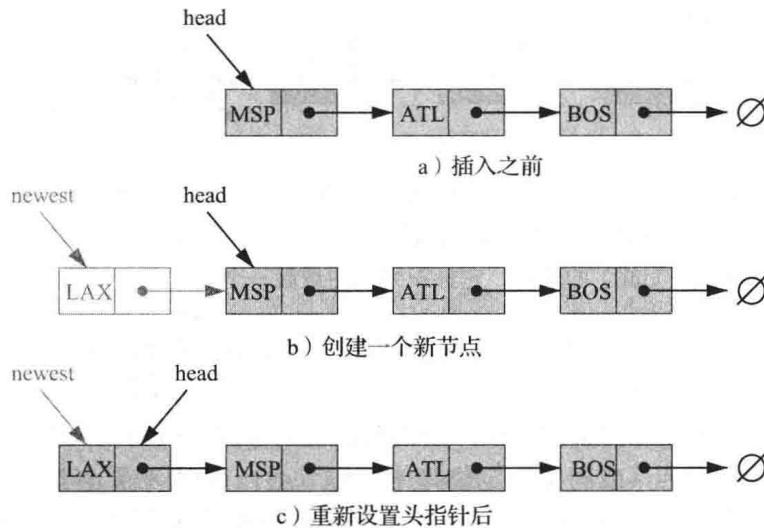


图 7-4 在单向链表的头部插入一个元素

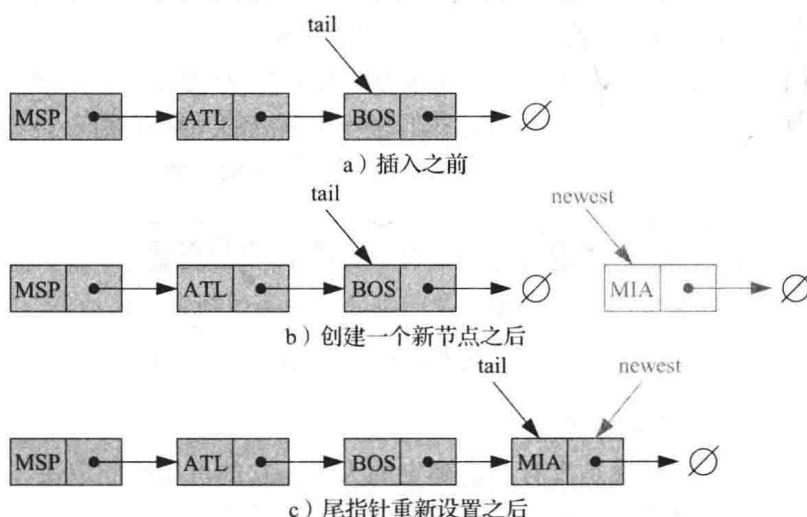


图 7-5 在单向链表的尾部插入一个元素

注意，必须在 c) 中设置尾指针变量指向新的节点之前设置 b) 中尾部的“next”指针。

代码段 7-2 在单向链表的尾部插入一个新的节点。注意，在设置尾指针指向新节点之前，设置尾节点的“next”指针指向原来的尾节点。当向一个空链表中插入新节点时，需要对这段代码进行一定的调整，因为空链表不存在尾节点

```
Algorithm addLast(L, e):
    newest = Node(e) {create new node instance storing reference to element e}
    newest.next = None {set new node's next to reference the None object}
    L.tail.next = newest {make old tail node point to new node}
    L.tail = newest {set variable tail to reference the new node}
    L.size = L.size + 1 {increment the node count}
```

从单向链表中删除一个元素

从单向链表的头部删除一个元素，基本上是在头部插入一个元素的反向操作。这个操作的详细过程如图 7-6 和代码段 7-3 所示。

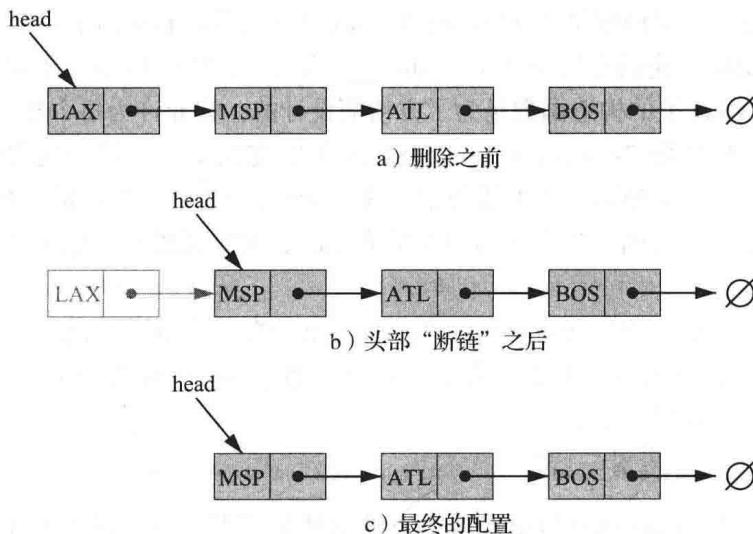


图 7-6 在单向链表的头部删除一个节点

代码段 7-3 在单向链表的头部删除一个节点

```

Algorithm remove_first(L):
    if L.head is None then
        Indicate an error: the list is empty.
    L.head = L.head.next           {make head point to next node (or None)}
    L.size = L.size - 1            {decrement the node count}

```

不幸的是，即使保存了一个直接指向列表尾节点的尾指针，我们也不能轻易地删除单向链表的尾节点。为了删除链表的最后一个节点，我们必须能够访问尾节点之前的节点。但是我们无法通过尾节点的“next”指针找到尾节点的前一个节点，访问此节点的唯一方法是从链表的头部开始遍历整个链表。但是这样序列遍历的操作需要花费很长的时间，如果想要有效地实现此操作，需要实现双向列表（见 7.3 节）。

7.1.1 用单向链表实现栈

在这一部分，我们将通过给出一个完整栈 ADT 的 Python 实现来说明单向链表的使用（见 6.1 节）。设计这样的实现，我们需要决定用链表的头部或尾部来实现栈顶。最好的选择显而易见：因为只有在头部，我们才能在一个常数时间内有效地插入和删除元素。由于所有栈操作都会影响栈顶，因此规定栈顶在链表的头部。

为了表示列表中的单个节点，我们创建了一个轻量级 `_Node` 类。这个类将永远不会直接暴露给栈类的用户，所以被正式定义为非公有的、最终的 `LinkedStack` 类的嵌套类（见 2.5.1 节）。代码段 7-4 展示了 `_Node` 类的定义。

代码段 7-4 一个单向链表的轻量级 `_Node` 类

```

class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    __slots__ = '_element', '_next'          # streamline memory usage

    def __init__(self, element, next):      # initialize node's fields
        self._element = element              # reference to user's element
        self._next = next                   # reference to next node

```

一个节点只有两个实例变量：`_element` 和 `_next`（元素引用和指向下一个节点的引用），为了提高内存的利用率，我们专门定义了 `__slots__`（见 2.5.1 节），因为一个单向链表中可能有多个节点实例。`_Node` 类的构造函数是为了方便而设计的，它允许为每个新创建的节点赋值。

代码段 7-5 和代码段 7-6 给出了 `LinkedStack` 类的完整实现。每个栈实例都维护两个变量。头指针指向链表的头节点（如果栈为空，这个指针指向空）。我们需要用变量 `_size` 持续追踪当前元素的数量，否则，当需要返回栈的大小时，必须通过遍历整个列表来计算元素的数量。

将元素压栈（`push`）的实现与代码段 7-1 所给出的在单向链表头部插入一个元素的伪代码基本一致。向栈顶放入一个新的元素 `e` 时，可以通过调用 `_Node` 类的构造函数来完成链接结构的必要改变。代码如下：

```
self._head = self._Node(e, self._head)      # create and link a new node
```

注意，新节点的 `_next` 指针域被设置为当前的栈顶节点，然后将头指针（`self._head`）指向新节点。

代码段 7-5 单向链表实现栈 ADT (后续内容见代码段 7-6)

```

1 class LinkedStack:
2     """LIFO Stack implementation using a singly linked list for storage."""
3
4     #----- nested _Node class -----
5     class _Node:
6         """Lightweight, nonpublic class for storing a singly linked node."""
7         __slots__ = '_element', '_next'      # streamline memory usage
8
9         def __init__(self, element, next):    # initialize node's fields
10            self._element = element          # reference to user's element
11            self._next = next                # reference to next node
12
13     #----- stack methods -----
14     def __init__(self):
15         """Create an empty stack."""
16         self._head = None                  # reference to the head node
17         self._size = 0                     # number of stack elements
18
19     def __len__(self):
20         """Return the number of elements in the stack."""
21         return self._size
22
23     def is_empty(self):
24         """Return True if the stack is empty."""
25         return self._size == 0
26
27     def push(self, e):
28         """Add element e to the top of the stack."""
29         self._head = self._Node(e, self._head)      # create and link a new node
30         self._size += 1
31
32     def top(self):
33         """Return (but do not remove) the element at the top of the stack.
34         """
35         if self.is_empty():
36             raise Empty('Stack is empty')
37         return self._head._element
38
39     # top of stack is at head of list

```

代码段 7-6 单向链表实现栈 ADT (接代码段 7-5)

```

40 def pop(self):
41     """Remove and return the element from the top of the stack (i.e., LIFO).
42
43     Raise Empty exception if the stack is empty.
44     """
45     if self.is_empty():
46         raise Empty('Stack is empty')
47     answer = self._head._element
48     self._head = self._head._next           # bypass the former top node
49     self._size -= 1
50     return answer

```

实现 `top` 方法时，目标是返回栈顶部的元素。当栈为空时，我们会抛出 `Empty` 异常，这个异常在第 6 章代码段 6-1 中已经定义过了。当栈不为空时，头指针 (`self._head`) 指向链表的第一个节点，栈顶元素可以表示为 `self._head._element`。

元素出栈操作 (`pop`) 的实现与代码段 7-3 中的伪代码基本一致。我们利用一个本地的指针指向要删除的节点中所保存的成员元素 (`element`)，并将该元素返回给调用者 `pop`。

表 7-1 给出了 `LinkedStack` 操作的分析。可以看到，所有方法在最坏情况下都是在常数时间内完成的。这与表 6-2 给出的数组栈的摊销边界形成了对比。

表 7-1 链式栈实现的性能，所有边界都是在最坏情况下确定的，空间利用率为 $O(n)$ 。其中 n 为当前栈中元素的个数

操作	运行时间
<code>S.push(e)</code>	$O(1)$
<code>S.pop()</code>	$O(1)$
<code>S.top()</code>	$O(1)$
<code>len(S)</code>	$O(1)$
<code>S.is_empty()</code>	$O(1)$

7.1.2 用单向链表实现队列

正如用单向链表实现栈 ADT 一样，我们可以用单向链表实现队列 ADT，且所有操作支持最坏情况的时间为 $O(1)$ 。由于需要对队列的两端执行操作，我们显式地为每个队列维护一个 `_head` 和一个 `_tail` 指针作为实例变量。一种很自然的做法是，将队列的前端和链表的头部对应，队列的后端与链表的尾部对应，因为必须使元素从队列的尾部进入队列，从队列的头部出队列（前面曾提到，我们很难高效地从单向链表的尾部删除元素）。链表队列 (`LinkedQueue`) 类的实现如代码段 7-7 和代码段 7-8 所示。

代码段 7-7 用单向链表实现队列 ADT (后续内容见代码段 7-8)

```

1 class LinkedQueue:
2     """FIFO queue implementation using a singly linked list for storage."""
3
4     class _Node:
5         """Lightweight, nonpublic class for storing a singly linked node."""
6         (omitted here; identical to that of LinkedStack._Node)
7
8     def __init__(self):
9         """Create an empty queue."""

```

```

10     self._head = None
11     self._tail = None
12     self._size = 0                               # number of queue elements
13
14 def __len__(self):
15     """Return the number of elements in the queue."""
16     return self._size
17
18 def is_empty(self):
19     """Return True if the queue is empty."""
20     return self._size == 0
21
22 def first(self):
23     """Return (but do not remove) the element at the front of the queue."""
24     if self.is_empty():
25         raise Empty('Queue is empty')
26     return self._head._element                  # front aligned with head of list

```

代码段 7-8 用单向链表实现队列 ADT (接代码段 7-7)

```

27 def dequeue(self):
28     """Remove and return the first element of the queue (i.e., FIFO).
29
30     Raise Empty exception if the queue is empty.
31     """
32     if self.is_empty():
33         raise Empty('Queue is empty')
34     answer = self._head._element
35     self._head = self._head._next
36     self._size -= 1
37     if self.is_empty():                         # special case as queue is empty
38         self._tail = None                      # removed head had been the tail
39     return answer
40
41 def enqueue(self, e):
42     """Add an element to the back of queue."""
43     newest = self._Node(e, None)               # node will be new tail node
44     if self.is_empty():
45         self._head = newest                  # special case: previously empty
46     else:
47         self._tail._next = newest
48     self._tail = newest                     # update reference to tail node
49     self._size += 1

```

用单向链表实现队列的很多方面和用 LinkedStack 类实现非常相似，如嵌套 _Node 类的定义。链表队列的 LinkedQueue 的实现类似于 LinkedStack 的出栈，即删除队列的头部节点，但也有一些细微的差别。因为队列必须准确地维护尾部的引用（栈的实现中没有维持这样的变量）。通常，在头部的操作对尾部不产生影响。但在一个队列中调用元素出队列操作时，我们要同时删除列表的尾部。同时，为了确保一致性，还要设置 self._tail 为 None。

在 LinedQueue 的实现问题中，有一个相似的复杂操作。最新的节点往往成为新的链表尾部，然而当这个新节点是列表中的唯一节点时，就会有所不同。在这种情况下，该节点也将变成新的链表头部；否则，新的节点必须被立即链接到现有的尾部节点之后。

在性能方面，LinkedQueue 与 LinkedStack 类似，所有操作在最坏情况下运行的时间为常数，而空间使用率与当前元素数量呈线性关系。

7.2 循环链表

在 6.2.2 节中，我们引入了“循环”数组的概念，并且说明了如何用其实现队列 ADT。在实现中，循环数组的概念是人为定义的，因此，在数组内部自身的表示中没有任何循环结构。这是我们在使用模运算中，将一个索引从最后一个位置“推进”到第一个位置时所提供的一个抽象概念。

在链表中，我们可以使链表的尾部节点的“next”指针指向链表的头部，由此来获得一个更切实际的循环链表的概念。我们称这种结构为循环链表，如图 7-7 所示。

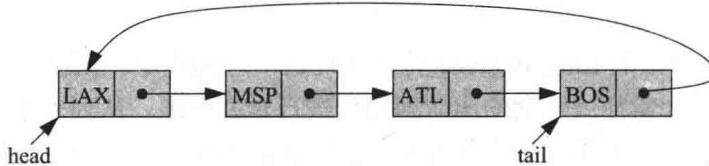


图 7-7 一个具有循环结构的单向链表示例

与标准的链表相比，循环链表为循环数据集提供了一个更通用的模型，即标准链表的开始和结束没有任何特定的概念。图 7-8 给出了一个相对图 7-7 中循环列表的结构更对称的示意图。

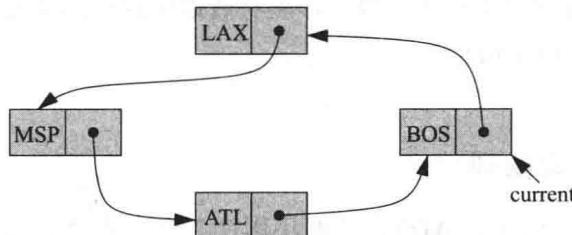


图 7-8 一个用“current”指针标明引用一个选定的节点的循环链表的例子

我们也可以使用其他类似于图 7-8 所示的环形视图，例如，描述美国芝加哥环线上的火车站点顺序或选手在比赛中的轮流顺序。虽然一个循环链表可能并没有开始或者结束节点，但是必须为一个特定的节点维护一个引用，这样才能使用该链表。我们采用“current”标识符来表示一个指定的节点。通过设置 `current=current.next`，我们可以有效地遍历链表中的各个节点。

7.2.1 轮转调度

为了说明循环链表的使用，我们来讨论一个循环调度程序，在这个调度程序中，以循环的方式迭代地遍历一个元素的集合，并通过执行一个给定的动作作为集合中的每个元素进行“服务”。例如，使用这种调度程序，可以公平地分配那些必须为一个用户群所共享的资源。比如，循环调度经常用于为同一计算机上多个并发运行的应用程序分配 CPU 时间片。

使用普通队列 ADT，在队列 Q 上反复执行以下步骤（见图 7-9），这样就可以实现循环调度程序：

1. $e = Q.\text{dequeue}()$
2. Service element e
3. $Q.\text{enqueue}(e)$

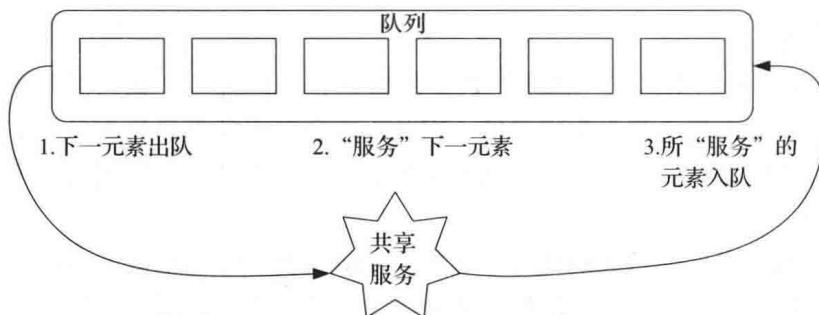


图 7-9 使用队列实现循环调度的三个迭代步骤

如果用 7.1.2 节介绍的 `LinkedQueue` 类来实现这个应用程序，则没有必要急于对那种在结束不久后就将同一元素插入队列的出队列操作进行合并处理。从列表中删除一个节点，相应地要适当调整列表的头并缩减列表的大小；对应地，当创建一个新的节点时，应将其插入列表的尾部并且增加列表的大小。

如果使用一个循环列表，有效地将一个项目从队列头部转换成队列尾部，可以通过访问标记队列边界的引用来实现。接下来，我们会给出一个用于支持整个队列 ADT 的循环队列类的实现，并介绍一个附加的方法 `rotate()`，该方法用于将队列中的第一个元素移动到队列尾部（在 python 模块集合的双端队列类中，支持一个类似的方法，参见表 6-4）。使用这个操作循环调度程序，可以通过重复执行以下步骤有效地实现循环调度算法：

- 1) Service element $Q.front()$.
- 2) $Q.rotate()$.

7.2.2 用循环链表实现队列

为了采用循环链表实现队列 ADT，我们用图 7-7 给出直观示意：队列有一个头部和一个尾部，但是尾部的“`next`”指针指向头部的。对于这样一个模型，我们显然不需要同时保存指向头部和尾部的引用（指针）。只要保存一个指向尾部的引用（指针），我们就总能通过尾部的“`next`”引用找到头部。

代码段 7-9 和代码段 7-10 给出了基于这个模型实现的循环队列类。该类只有两个实例变量：一个是 `_tail`，用于指向尾部节点的引用（当队列为空时指向 `None`）；另一个是 `_size`，用于记录当前队列中元素的数量。当一个操作涉及队列的头部时，我们用 `self._tail._next` 标识队列的头部。当调用 `enqueue` 操作时，一个新的节点将被插入队列的尾部与当前头部之间，然后这个新节点变成了新的尾部。

除了传统的队列操作，`CircularQueue` 类还支持一个循环的方法，该方法可以更有效地实现删除队首的元素以及将该元素插入队列尾部这两个操作的合并处理。用循环来表示，简单地设 `self._tail=self._tail._next`，以使原来的头部变成新的尾部。（原来头部的后继节点成为新的头部）

代码段 7-9 用循环链表存储实现循环队列类（后续代码段 7-10）

```

1 class CircularQueue:
2     """Queue implementation using circularly linked list for storage."""
3
4     class _Node:
5         """Lightweight, nonpublic class for storing a singly linked node."""

```

```

6      (omitted here; identical to that of LinkedStack._Node)
7
8  def __init__(self):
9      """Create an empty queue."""
10     self._tail = None          # will represent tail of queue
11     self._size = 0             # number of queue elements
12
13  def __len__(self):
14      """Return the number of elements in the queue."""
15      return self._size
16
17  def is_empty(self):
18      """Return True if the queue is empty."""
19      return self._size == 0

```

代码段 7-10 用循环链表存储实现循环队列类（接代码段 7-9）

```

20  def first(self):
21      """Return (but do not remove) the element at the front of the queue.
22
23      Raise Empty exception if the queue is empty.
24      """
25
26      if self.is_empty():
27          raise Empty('Queue is empty')
28      head = self._tail._next
29      return head._element
30
31  def dequeue(self):
32      """Remove and return the first element of the queue (i.e., FIFO).
33
34      Raise Empty exception if the queue is empty.
35      """
36
37      if self.is_empty():
38          raise Empty('Queue is empty')
39      oldhead = self._tail._next
40      if self._size == 1:           # removing only element
41          self._tail = None        # queue becomes empty
42      else:
43          self._tail._next = oldhead._next    # bypass the old head
44      self._size -= 1
45      return oldhead._element
46
47  def enqueue(self, e):
48      """Add an element to the back of queue."""
49      newest = self._Node(e, None)      # node will be new tail node
50      if self.is_empty():
51          newest._next = newest       # initialize circularly
52      else:
53          newest._next = self._tail._next    # new node points to head
54          self._tail._next = newest        # old tail points to new node
55          self._tail = newest            # new node becomes the tail
56      self._size += 1
57
58  def rotate(self):
59      """Rotate front element to the back of the queue."""
60      if self._size > 0:
61          self._tail = self._tail._next        # old head becomes new tail

```

7.3 双向链表

在单向链表中，每个节点为其后继节点维护一个引用。我们已经说明了在管理一个序列

的元素时如何使用这样的表示方法。然而，单向链表的不对称性产生了一些限制。在 7.1 节的开头，我们强调过可以有效地向一个单向链表内部的任意位置插入一个节点，也可以在头部轻松地删除一个节点，但是不能有效地删除链表尾部的节点。更一般化的说法是，如果仅给定链表内部指向任意一个节点的引用，我们很难有效地删除该节点，因为我们无法立即确定待删除节点的前驱节点（而且删除处理中该前驱节点需要更新它的“next”引用）。

为了提供更好的对称性，我们定义了一个链表，每个节点都维护了指向其先驱节点以及后继节点的引用。这样的结构被称为双向链表。这些列表支持更多各种时间复杂度为 $O(1)$ 的更新操作，这些更新操作包括在列表的任意位置插入和删除节点。我们会继续用“next”表示指向当前节点的后继节点的引用，并引入“prev”引用其前驱节点。

头哨兵和尾哨兵

在操作接近一个双向链表的边界时，为了避免一些特殊情况，在链表的两端都追加节点是很有用处的：在列表的起始位置添加头节点（header），在列表的结尾位置添加尾节点（tailer）。这些“特定”的节点被称为哨兵（或保安）。这些节点中并不存储主序列的元素。图 7-10 中给出了一个带哨兵的双向链表。

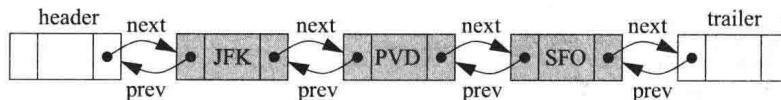


图 7-10 用一个使用 header 和 trailer 哨兵来区分列表的端部的双向链表表示序列 {JFK, PVD, SFO}

当使用哨兵节点时，一个空链表需要初始化，使头节点的“next”域指向尾节点，并令尾节点的“prev”域指向头节点。哨兵节点的剩余域是无关紧要的。对于一个非空的列表，头节点的“next”域将指向一个序列中第一个真正包含元素的节点，对应的尾节点的“prev”域指向这个序列中最后一个包含元素的节点。

使用哨兵的优点

虽然不使用哨兵节点就可以实现双向链表（正如 7.1 节中的单向链表那样），但哨兵只占用很小的额外空间就能极大地简化操作的逻辑。最明显的是，头和尾节点从来不改变——只改变头节点和尾节点之间的节点。此外，可以用统一的方式处理所有插入节点操作，因为一个新节点总是被放在一对已知节点之间。类似地，每个待删除的元素都是确保被存储在前后都有邻居的节点中的。

相比之下，回顾 7.1.2 节中 `LinkedQueue` 的实现（其入 `enqueue` 方法在代码段 7-8 中给出），一个新节点是在列表的尾部进行添加的。然而，它需要设置一个条件去管理向空列表插入节点的特例情况。在一般情况下，新节点被连接在列表现在的尾部之后。但当插入空列表中时，不存在列表的尾部，因此必须重新给 `self._head` 赋值为新节点的引用。在实现中，使用哨兵节点可以消除这种特例的处理，就好像在新节点之前总是有一个已存在的节点。

双端链表的插入和删除

向双向链表插入节点的每个操作都将发生在两个已有节点之间，如图 7-11 所示。例如，当一个新元素被插在序列的前面时，我们可以简单地将这个新节点插入头节点和当前位于头节点之后的节点之间，如图 7-12 所示。

图 7-13 所示的是和插入相反的过程——删除节点。被删除节点的两个邻居直接相互连接起来，从而绕过被删节点。这样一来，该节点将不再被视作列表的一部分，它也可以被系

统收回。由于用了哨兵，可以使用相同的方法实现删除序列中的第一个或最后一个元素，因为一个元素必然存储在位于某两个已知节点之间的节点上。

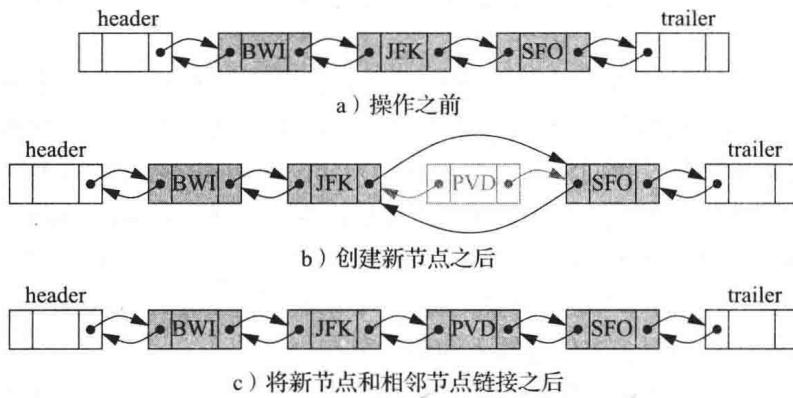


图 7-11 在带有头、尾哨兵的双向链表中添加一个节点

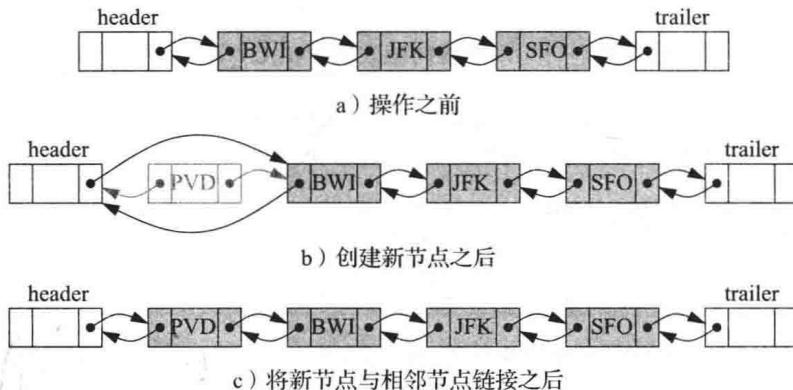


图 7-12 在带有头和尾哨兵的双向链表序列的前端添加一个元素

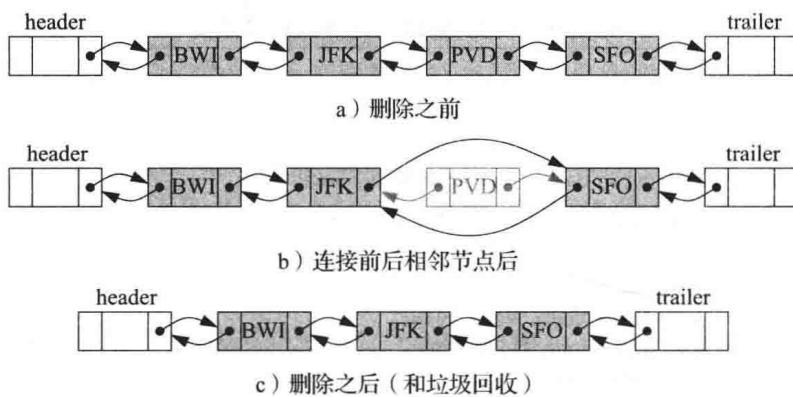


图 7-13 从双向链表中删除 PVD 元素

7.3.1 双向链表的基本实现

我们首先给出一个双向链的初步实现，这个实现是在一个名为 `_DoublyLinkedListBase` 的类中定义的。由于我们不打算为一般应用提供一个常规的公共接口，因此有意将这个类名定义为以下划线开头。我们会看到链表可以支持一般在最坏情况下时间复杂度为 $O(1)$ 的插入和

删除，但这仅限于当一个操作的位置可以被简单地识别出来的情况。对于基于数组的序列，用整数作为索引是描述序列中某个位置的一种便利之法。然而，当没有给出一种有效的方法来查找一个链表中的第 j 个元素时，索引并不是合适的方法，因为这种方法将需要遍历链表的一部分。

当处理一个链表时，描述一个操作的位置最直接的方法是找到与这个列表相关联的节点。但是，我们倾向于将数据结构的内部处理封装起来，从而避免用户直接访问到列表的节点。在本章的剩余部分，我们将开发两个从 `_DoublyLinkedListBase` 类继承而来的公有类，从而提供更一致的概念。尤其是在 7.3.2 节中，我们将提供一个 `LinkedDeque` 类，用于实现在 6.3 节中介绍的双头队列 ADT。这个类只支持在队列末端的操作，所以用户不需要查找其在内部列表中的位置。在 7.4 节中，我们将引入一个新的概念 `PositionalList`，这个类提供一个公共接口，以允许从一个列表中任意插入和删除节点。

低级 `_DoublyLinkedListBase` 类使用一个非公有的节点类 `_Node`，这个非公有类类似于一个单向链表。如代码段 7-4 给出的，这个双向链表的版本除了包括 `_prev` 属性，还包含 `_next` 和 `_element` 属性，如代码段 7-11 所示。

代码段 7-11 用于双向链表的 Python `_Node` 类

```
class _Node:
    """Lightweight, nonpublic class for storing a doubly linked node."""
    __slots__ = '_element', '_prev', '_next' # streamline memory

    def __init__(self, element, prev, next):
        # initialize node's fields
        self._element = element # user's element
        self._prev = prev # previous node reference
        self._next = next # next node reference
```

`_DoublyLinkBase` 类中定义的其余内容在代码段 7-12 中给出。构造函数实例化两个哨兵节点并将这两个节点直接链接。我们维护了一个 `_size` 成员以及公有成员 `__len__` 和 `is_empty`，以使这些行为可以直接被子类继承。

代码段 7-12 管理双向链表的基本类

```
1 class _DoublyLinkedListBase:
2     """A base class providing a doubly linked list representation."""
3
4     class _Node:
5         """Lightweight, nonpublic class for storing a doubly linked node."""
6         (omitted here; see previous code fragment)
7
8     def __init__(self):
9         """Create an empty list."""
10    self._header = self._Node(None, None, None)
11    self._trailer = self._Node(None, None, None)
12    self._header._next = self._trailer # trailer is after header
13    self._trailer._prev = self._header # header is before trailer
14    self._size = 0 # number of elements
15
16    def __len__(self):
17        """Return the number of elements in the list."""
18        return self._size
19
20    def is_empty(self):
21        """Return True if list is empty."""
22        return self._size == 0
```

```

23
24 def _insert_between(self, e, predecessor, successor):
25     """ Add element e between two existing nodes and return new node."""
26     newest = self._Node(e, predecessor, successor) # linked to neighbors
27     predecessor._next = newest
28     successor._prev = newest
29     self._size += 1
30     return newest
31
32 def _delete_node(self, node):
33     """Delete nonsentinel node from the list and return its element."""
34     predecessor = node._prev
35     successor = node._next
36     predecessor._next = successor
37     successor._prev = predecessor
38     self._size -= 1
39     element = node._element # record deleted element
40     node._prev = node._next = node._element = None # deprecate node
41     return element # return deleted element

```

这个类的其他两个方法是私有的应用程序，即 `_insert_between` 和 `_delete_node`。这些方法分别为插入和删除提供通用的支持，但需要以一个或多个节点的引用作为参数。`_insert_between` 方法是根据图 7-11 所示的算法模型化实现的。该方法创建一个新节点，节点字段初始化链接到指定的邻近节点，然后邻近节点的字段要进行更新，以获得最新节点的相关信息。为后继处理方便，这个方法返回新创建的节点的引用。

`_delete_node` 方法是根据图 7-13 所示的算法模块化进行实现的。与被删除节点相邻的两个点，直接相链接，从而使列表绕过这个被删除节点，作为一种形式，我们故意重新设置被删除节点的 `_prev`、`_next` 和 `_element` 域为空（在记录要返回的元素之后）。虽然被删除的节点会被列表的其余部分忽略，但设置该节点的域为 `None` 是有利的，这样一来，该节点与其他节点不必要的链接和存储元素将会被消除，从而帮助 Python 进行垃圾回收。我们还将依赖这个配置识别因不再是列表的一部分而“被弃用”的节点。

7.3.2 用双向链表实现双端队列

6.3 节中介绍了双端队列 ADT。由于偶尔需要调整数组的大小，我们基于数组实现的所有操作都在平均 $O(1)$ 的时间复杂度下得以完成。在一个基于双向链表的实现中，我们能够在最坏情况下以时间复杂度为 $O(1)$ 完成双端队列的所有操作。

代码段 7-13 给出了 `LinkedDeque` 类的实现，它继承自前一节中介绍的双端队列 `_DoublyLinkedListBase` 类。由于 `LinkedDeque` 类中的一系列继承方法就可以初始化一个新的实例，所以我们不再提供一个明确的方法来初始化链式队列类。我们还借助于 `__len__` 和 `is_empty` 等继承而得的方法来满足双端队列 ADT 的要求。

代码段 7-13 从继承双向链基类而实现的链式双端队列类

```

1 class LinkedDeque(_DoublyLinkedListBase): # note the use of inheritance
2     """Double-ended queue implementation based on a doubly linked list."""
3
4     def first(self):
5         """Return (but do not remove) the element at the front of the deque."""
6         if self.is_empty():
7             raise Empty("Deque is empty")
8         return self._header._next._element # real item just after header
9

```

```

10  def last(self):
11      """ Return (but do not remove) the element at the back of the deque."""
12      if self.is_empty():
13          raise Empty("Deque is empty")
14      return self._trailer._prev._element           # real item just before trailer
15
16  def insert_first(self, e):
17      """ Add an element to the front of the deque."""
18      self._insert_between(e, self._header, self._header._next)  # after header
19
20  def insert_last(self, e):
21      """ Add an element to the back of the deque."""
22      self._insert_between(e, self._trailer._prev, self._trailer)  # before trailer
23
24  def delete_first(self):
25      """ Remove and return the element from the front of the deque.
26
27      Raise Empty exception if the deque is empty.
28      """
29      if self.is_empty():
30          raise Empty("Deque is empty")
31      return self._delete_node(self._header._next)        # use inherited method
32
33  def delete_last(self):
34      """ Remove and return the element from the back of the deque.
35
36      Raise Empty exception if the deque is empty.
37      """
38      if self.is_empty():
39          raise Empty("Deque is empty")
40      return self._delete_node(self._trailer._prev)        # use inherited method

```

在使用哨兵时，实现方法的关键是要记住双端队列的第一个元素并不存储在头节点，而是存储在头节点后的第一个节点（假定双端队列是非空的）。同样，尾节点之前的一个节点中存储的是双端队列的最后一个元素。

我们使用通过继承得到的方法 `_insert_between` 向双端队列的两端进行插入操作。为了向双端队列前端插入一个元素，我们需要将这个元素立即插入头节点和其后的一个节点之间。如果是在双端队列末尾插入节点，则可直接将节点置于尾节点之前。值得注意的是，这些操作即使在双端队列为空时也能成功：在这种情况下，新节点将被放置在两个哨兵之间。当从一个非空队列删除一个元素，且明确知道目标节点肯定有前驱和后继节点时，我们可以利用继承得到的 `_delete_node` 方法来实现。

7.4 位置列表的抽象数据类型

到目前为止，我们所讨论的抽象数据类型包括栈、队列和双向队列等，并且仅允许在序列的一端进行更新操作。有时，我们希望有一个更一般的概念。例如，虽然我们采用队列的 FIFO 语义作为一种模型，来描述正在等待与客户服务代表对话的顾客或者正在排队买演出门票的粉丝，但是队列 ADT 有很大的局限。如果等待的顾客在到达顾客服务队列列首之前决定离开，或者排队买票的人允许他的朋友“插队”到他所站的位置呢？我们希望能够设计一个抽象数据类型来为用户提供一种可以定位到序列中任何元素的方法，并且能够执行任意的插入和删除操作。

在处理基于数组的序列（如 Python 列表）时，整数索引提供了一种很好的方式来描述一

个元素的位置，或者描述一个即将发生插入和删除操作的位置。然而，数字索引并不适用于描述一个链表内部的位置，因为我们不能有效地访问一个只知道其索引的条目。找到链表中一个给定索引的元素，需要从链表的开始或者结束的位置起逐个遍历从而计算出目标元素的位置。

此外，在描述某些应用程序中的本地位置时，索引并非好的抽象，因为序列中不停地发生插入或删除操作，条目的索引值会随着时间的推移发生变化。例如，一个排队者的具体位置并不能通过精确地知道队列中在他之前到底有多少人而很容易地描述出来。我们提出一个抽象，如图 7-14 所示，用一些其他方法描述位置。然后我们希望给一些情况建模，例如，当一个指定的排队者在到达队首之前离开队列，或立即在队列中一个指定的排队者之后增加一个新人。

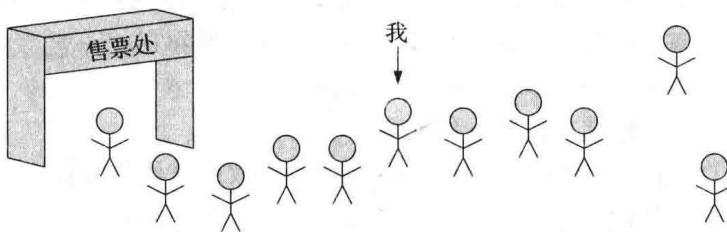


图 7-14 我们希望能够识别序列中一个元素的位置，而无须使用整数索引

再如，一个文本文档可以被视为一个长的字符序列。文字处理器使用游标的抽象描述文档中的一个位置，而没有明确地使用整数索引，支持如“删除此游标处的字符”或者“在当前游标之后插入新的字符”这样的操作。此外，我们可以引用文档中一个固有的位置，比如一个特定章节的开始，但不能依赖于一个字符索引（甚至一个章节编号），因为这个索引可能会随着文档的演化而改变。

节点的引用表示位置

链表结构的好处之一是：只要给出列表相关节点的引用，它可以实现在列表的任意位置执行插入和删除操作的时间复杂度都是 $O(1)$ 。因此，很容易开发一个 ADT，它以一个节点引用实现描述位置的机制。事实上，7.3.1 节 `_DoublyLinkedListBase` 基础类中的 `_insert between` 和 `_delete node` 方法都接受节点引用作为参数。

然而，这样直接使用节点的方式违反了在第 2 章中介绍的抽象和封装这两个面向对象的设计原则。为了我们自己和抽象的用户的利益，有几个原因致使我们倾向于封装一个链表中的节点：

- 对于用户来说，如果不被数据结构的实现中那些例如节点的低级操作，或依赖哨兵节点的使用等不必要的细节所干扰，那么使用这些数据结构会更加简单。注意，在 `_DoubleyLinkedListBase` 类中使用 `_insert between` 方法来向一个序列的起始位置添加节点时，头部哨兵必须作为参数传递进去。
- 如果不允许用户直接访问或操作节点，我们可以提供一个更健壮的数据结构。这样就可以确保用户不会因无效管理节点的连接而致使列表的一致性变成无效。如果允许用户调用我们定义的 `_DoubleyLinkedListBase` 类中的 `_insert between` 或 `_delete node` 方法，并将一个不属于给定列表的节点作为参数传递进去，则会发生更微妙的问题（回头看看这段代码，看看为什么它会引起这个问题）。
- 通过更好地封装实施的内部细节，我们可以获得更大的灵活性来重新设计数据结构以及改善性能。事实上，通过一个设计良好的抽象，我们可以提供一个非数字的位

置的概念，即使使用一个基于数组的序列。

由于这些原因，我们引入一个独立的位置抽象表示列表中一个元素的位置，而不是直接依赖于节点，进而引入一个可以封装双向链表的（甚至是基于数组序列的，参见练习 P-7.46）完整的含位置信息的列表 ADT。

7.4.1 含位置信息的列表抽象数据类型

为了给具有标识元素位置能力的元素序列提供一般化抽象，我们定义了一个含位置信息的列表 ADT 以及一个更简单的位置抽象数据类型，来描述列表中的某个位置。将一个位置作为更广泛的位置列表中的一个标志或标记。改变列表的其他位置不会影响位置 p 。使一个位置变得无效的唯一方法就是直接显式地发出一个命令来删除它。

位置实例是一个简单的对象，只支持以下方法：

- $p.element()$: 返回存储在位置 p 的元素。

在位置列表 ADT 中，位置可以充当一些方法的参数或是作为其他方法的返回值。在描述位置列表的行为时，我们介绍如下列表 L 所支持的访问器方法：

- $L.first()$: 返回 L 中第一个元素的位置。如果 L 为空，则返回 None 。
- $L.last()$: 返回 L 中最后一个元素的位置。如果 L 为空，则返回 None 。
- $L.before(p)$: 返回 L 中 p 紧邻的前面元素的位置。如果 p 为第一个位置，则返回 None 。
- $L.after(p)$: 返回 L 中 p 紧邻的后面元素的位置。如果 p 为最后一个位置，则返回 None 。
- $L.is_empty()$: 如果 L 列表不包含任何元素，返回 True 。
- $\text{len}(L)$: 返回列表元素的个数。
- $\text{iter}(L)$: 返回列表元素的前向迭代器。见 1.8 节中有关 Python 迭代器的讨论。

位置列表 ADT 也包括以下更新方法：

- $L.add_first(e)$: 在 L 的前面插入新元素 e ，返回新元素的位置。
- $L.add_last(e)$: 在 L 的后面插入新元素 e ，返回新元素的位置。
- $L.add_before(p, e)$: 在 L 中位置 p 之前插入一个新元素 e ，返回新元素的位置。
- $L.add_after(p, e)$: 在 L 中位置 p 之后插入一个新元素 e ，返回新元素的位置。
- $L.replace(p, e)$: 用元素 e 取代位置 p 处的元素，返回之前 p 位置处的元素。
- $L.delete(p)$: 删除并且返回 L 中位置 p 处的元素，取消该位置。

ADT 的这些方法以参数形式接收 p 的位置，如果列表 L 中 p 不是有效的位置信息，则发生错误。

注意，含位置信息列表 ADT 中 $frist()$ 和 $last()$ 方法的返回值是相关的位置，不是元素（这一点与双向队列中相应的 $frist()$ 和 $last()$ 的方法相反）。含位置信息列表的第一个元素可以通过随后调用这个位置上的元素的方法来确定，即 $L.first().element()$ 。将位置作为返回值来接收的优势是我们可以使用这个位置为列表导航。例如，下面代码片段将打印一个名为 $data$ 的含位置信息列表的所有元素。

```
cursor = data.first()
while cursor is not None:
    print(cursor.element())      # print the element stored at the position
    cursor = data.after(cursor)  # advance to the next position (if any)
```

上述代码依赖于这样的规定，在对列表最后面的位置调用“after”时，就会返回 None 对象。这个返回值可以明确地从所有合法位置区分出来。类似地，这个含位置信息的列表 ADT 在对列表最前面的位置调用“before”方法时返回值为 None，或者在空列表调用 first 和 last 方法时，也会返回 None。因此，即使列表为空，上面的代码片段也可正常运行。

因为这个 ADT 包括支持 python 的 iter 函数。用户可以采用传统的 for 循环语法向前遍历这样一个命名数据列表。

```
for e in data:  
    print(e)
```

位置列表 ADT 更为一般化的引导和更新方法如下面示例所示。

例题 7-1：下表显示了一个初始化为空的位置列表 L 上的一些列操作。为了区分位置实例，我们使用了变量 p 和 q。为了便于展示，当展示列表内容时，我们使用下标符号来表示它的位置。

操作	返回值	L
L.add_last(8)	p	8p
L.first()	p	8p
L.add_after(p, 5)	q	8p, 5q
L.before(q)	p	8p, 5q
L.add_before(q, 3)	r	8p, 3r, 5q
r.element()	3	8p, 3r, 5q
L.after(p)	r	8p, 3r, 5q
L.before(p)	None	8p, 3r, 5q
L.add_first(9)	s	9s, 8p, 3r, 5q
L.delete(L.last())	5	9s, 8p, 3r
L.replace(p, 7)	8	9s, 7p, 3r

7.4.2 双向链表实现

在本节中，我们呈现一个使用双向链表完整实现位置列表类 PositionalList 的方法，并满足以下重要的命题。

命题 7-2：当使用双向链表实现时，位置列表 ADT 每个方法的运行时间为最坏情况 $O(1)$ 。

我们采用 7.3.1 节中的 _DoublyLinkedList 类作为底层的表示，新类主要用于按照位置列表 ADT 提供一个公共的接口。我们在代码段 7-14 中从定义公共类 Position 开始在 PositionalList 类中嵌套定义类。Position 实例将用来表示列表中元素的位置。各种 PositionalList 方法可能会创建冗余的 Position 实例引用相同的底层节点（例如，开始和最后是相同的）。出于这个原因，Position 类定义了 __eq__ 和 __ne__ 这两个特殊方法，从而使一个如 $p==q$ 判断的测试在两个位置引用同一个节点的情况下能够得出 True 的结论。

确认位置

每当 PositionalList 类的一个方法以参数形式接收一个位置信息时，我们想确认这个位置是有效的，以确定与这个位置关联的底层的节点。这个功能是由一个名叫 _validate 的非公有的方法实现的。在内部，一个位置为链表的相关节点维护着引用信息，并且列表实例的引用包含指定的节点。利用这种容器的引用，当调用者发送不属于指定列表的位置实例时，

我们可以轻易地检测到。

我们也能够检测到一个属于列表，但其指向节点不再是列表一部分的位置实例。回想，基类的 `_delete_node` 将被删除节点的前驱和后继的引用设置为 `None`；我们可以通过识别这一条件来检测被弃用的节点。

访问和更新方法

`Positiona` 类的访问方法在代码段 7-15 中给出，更新方法在代码段 7-16 中给出。所有这些方法非常适用于底层双向链表实现支持位置列表 ADT 的公共接口。这些方法依赖于 `_validate` 工具“打开”发送的任何位置。它们还依赖于一个 `_make_position` 工具来“包装”节点作为 `Position` 实例返回给用户，确保不要返回一个引用哨兵的位置。为了方便起见，我们已经重载了继承的实用程序方法中的 `_insert_between` 方法，这样可以返回一个相对应的新创建节点的位置（继承版本则返回节点本身）。

代码段 7-14 基于双向链表的 `PositionalList` 类（后接代码段 7-15 和代码段 7-16）

```

1 class PositionalList(_DoublyLinkedList):
2     """A sequential container of elements allowing positional access."""
3
4     #----- nested Position class -----
5     class Position:
6         """An abstraction representing the location of a single element."""
7
7         def __init__(self, container, node):
8             """Constructor should not be invoked by user."""
9             self._container = container
10            self._node = node
11
12
13         def element(self):
14             """Return the element stored at this Position."""
15             return self._node._element
16
17         def __eq__(self, other):
18             """Return True if other is a Position representing the same location."""
19             return type(other) is type(self) and other._node is self._node
20
21         def __ne__(self, other):
22             """Return True if other does not represent the same location."""
23             return not (self == other)      # opposite of __eq__
24
25     #----- utility method -----
26     def _validate(self, p):
27         """Return position's node, or raise appropriate error if invalid."""
28         if not isinstance(p, self.Position):
29             raise TypeError('p must be proper Position type')
30         if p._container is not self:
31             raise ValueError('p does not belong to this container')
32         if p._node._next is None:          # convention for deprecated nodes
33             raise ValueError('p is no longer valid')
34         return p._node

```

代码段 7-15 基于双向链表的 `PositionalList` 类（前继代码段 7-14，后续代码段 7-16）

```

35     #----- utility method -----
36     def _make_position(self, node):
37         """Return Position instance for given node (or None if sentinel)."""
38         if node is self._header or node is self._trailer:
39             return None                      # boundary violation

```

```

40     else:
41         return self.Position(self, node)           # legitimate position
42
43     #----- accessors -----
44 def first(self):
45     """ Return the first Position in the list (or None if list is empty)."""
46     return self._make_position(self._header._next)
47
48 def last(self):
49     """ Return the last Position in the list (or None if list is empty)."""
50     return self._make_position(self._trailer._prev)
51
52 def before(self, p):
53     """ Return the Position just before Position p (or None if p is first)."""
54     node = self._validate(p)
55     return self._make_position(node._prev)
56
57 def after(self, p):
58     """ Return the Position just after Position p (or None if p is last)."""
59     node = self._validate(p)
60     return self._make_position(node._next)
61
62 def __iter__(self):
63     """ Generate a forward iteration of the elements of the list."""
64     cursor = self.first()
65     while cursor is not None:
66         yield cursor.element()
67         cursor = self.after(cursor)

```

代码段 7-16 基于双向链表的 PositionalList 类 (接代码段 7-14 和代码段 7-15)

```

68     #----- mutators -----
69     # override inherited version to return Position, rather than Node
70 def _insert_between(self, e, predecessor, successor):
71     """ Add element between existing nodes and return new Position."""
72     node = super()._insert_between(e, predecessor, successor)
73     return self._make_position(node)
74
75 def add_first(self, e):
76     """ Insert element e at the front of the list and return new Position."""
77     return self._insert_between(e, self._header, self._header._next)
78
79 def add_last(self, e):
80     """ Insert element e at the back of the list and return new Position."""
81     return self._insert_between(e, self._trailer._prev, self._trailer)
82
83 def add_before(self, p, e):
84     """ Insert element e into list before Position p and return new Position."""
85     original = self._validate(p)
86     return self._insert_between(e, original._prev, original)
87
88 def add_after(self, p, e):
89     """ Insert element e into list after Position p and return new Position."""
90     original = self._validate(p)
91     return self._insert_between(e, original, original._next)
92
93 def delete(self, p):
94     """ Remove and return the element at Position p."""
95     original = self._validate(p)
96     return self._delete_node(original)    # inherited method returns element
97
98 def replace(self, p, e):

```

```

99     """Replace the element at Position p with e.
100
101    Return the element formerly at Position p.
102
103    original = self._validate(p)
104    old_value = original._element           # temporarily store old element
105    original._element = e                 # replace with new element
106    return old_value                     # return the old element value

```

7.5 位置列表的排序

在 5.5.2 节中，我们介绍了在一个基于数组的序列中的插入排序算法。在本节中，我们开发一个在 PositionalList 上进行操作的实现，这个实现同样是依赖于在对元素进行排序并不断增长的集合中实现的高级算法。

我们维护一个名为 marker 的变量，这个变量表示一个列表当前排序部分最右边的位置。我们每次考虑用 pivot 标记 marker 刚过去的位置和 pivot 的元素属于相对排序的部分。我们使用另一个被命名为 walk 的变量，从 marker 向左移动，只要还有一个前驱元素的值大于 pivot 元素的值，就一直移动。这些变量的典型配置如图 7-15 所示。采用 Python 对这个策略的实现如代码段 7-17 所示。

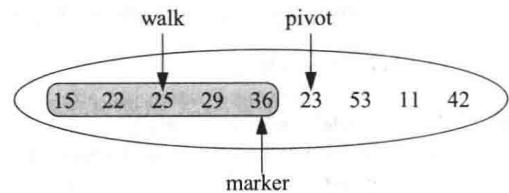


图 7-15 插入排序中一个步骤的示意图。阴影部分的元素（一直到 marker）已经排好序。在这一步中，pivot 的元素应该在 walk 位置之前被立即重新定位

代码段 7-17 在位置列表中执行插入排序的 python 代码

```

1 def insertion_sort(L):
2     """Sort PositionalList of comparable elements into nondecreasing order."""
3     if len(L) > 1:                      # otherwise, no need to sort it
4         marker = L.first()
5         while marker != L.last():
6             pivot = L.after(marker)        # next item to place
7             value = pivot.element()
8             if value > marker.element():   # pivot is already sorted
9                 marker = pivot            # pivot becomes new marker
10            else:                      # must relocate pivot
11                walk = marker            # find leftmost item greater than value
12                while walk != L.first() and L.before(walk).element() > value:
13                    walk = L.before(walk)
14                L.delete(pivot)
15                L.add_before(walk, value)  # reinsert value before walk

```

7.6 案例研究：维护访问频率

在很多设置中，位置列表 ADT 都是有用的。例如，在一个模拟纸牌游戏的程序中，可以对每个人的手用位置列表进行建模（练习 P-7.47）。因为大多数人会把相同花色的纸牌放在一起，所以从一个人手中插入和拿出纸牌可以使用位置列表 ADT 的方法实现，其位置是由各个花色的自然顺序决定的。同样，一个简单的文本编辑器嵌入含位置的插入和删除的概念，因为这类编辑器的所有更新都是相对于一个游标执行的，该游标表示列表文本中正在被编辑的当前位置的字符。

在本节中，当跟踪每个元素被访问的次数时，我们考虑维护一个元素的集合。保存元素的访问数量，使我们知道集合中的哪些元素是最受欢迎的。这种场景的例子包括能够跟踪用户访问最多的 URL 信息的 Web 浏览器，或者是那种能够保存用户最常播放歌曲列表的音乐收藏夹。我们用新的 favorites list ADT 来建模，它支持 len 和 is_empty 方法，还支持以下的方法：

- access(e)：访问元素 e，增加其访问数量。如果它尚未存在于收藏夹列表中，会将它添加至列表中。
- remove(e)：从收藏夹列表中移除元素 e，前提是存在这样的 e。
- top(k)：返回前 k 个访问最多的元素的迭代器。

7.6.1 使用有序表

管理收藏夹的第一种方法是在链表中存储元素，按访问次数的降序顺序来存储这些元素。在访问或者移除一个元素时，通过从最经常访问的到最少经常访问的元素的方式查询列表的方法进行元素定位。返回前 k 个访问最频繁的元素很容易，因为只要返回列表中的前 k 个元素记录即可。

为了使列表以元素访问次数降序排列的方式保持不变，我们必须考虑一个单次访问操作对元素的排列顺序会产生怎样的影响。被访问的元素的访问次数加一，它的访问次数就可能比原来在它之前的一个或者几个元素的都多了，这样就会破坏了列表的不变性。

所幸，我们可以采用前一节中介绍的一个类似于单向插入排序算法对列表重新排序。我们可以从访问数量增加的元素的位置开始，执行一个列表的向后遍历，直至找到一个元素可以被重定位的有效位置之后。

使用组合模式

我们希望利用 PositionalList 类作为存储辅助实现一个收藏夹列表。如果位置列表的元素是收藏夹的简单元素，我们将面临的挑战是当列表的内容被重新排序时，维护访问次数以及保持列表中相关联元素的适当数量。我们使用一个通用的面向对象的设计模式——组合模式。在这个模式中，我们定义了一个由两个或两个以上其他对象组成的单一对象。具体地说，我们定义了一个名为 _Item 的非公有嵌套类，用于存储元素并以其访问次数作为一个实例。然后，将收藏夹作为 item 实例以 PositionalList 来维护，这样用户元素的访问次数就都可以被嵌入我们的表示方法中（_Item 从来不会暴露给 FavoritesList 的用户，见代码段 7-18 和 7-19）。

代码段 7-18 FavoritesList 类（后续代码段 7-19）

```

1 class FavoritesList:
2     """List of elements ordered from most frequently accessed to least."""
3
4     #----- nested _Item class -----
5     class _Item:
6         __slots__ = '_value', '_count'           # streamline memory usage
7         def __init__(self, e):
8             self._value = e                      # the user's element
9             self._count = 0                      # access count initially zero
10
11    #----- nonpublic utilities -----
12    def _find_position(self, e):
13        """Search for element e and return its Position (or None if not found)."""
14        walk = self._data.first()
15        while walk is not None and walk.element() != e:

```

```

16     walk = self._data.after(walk)
17     return walk
18
19 def _move_up(self, p):
20     """Move item at Position p earlier in the list based on access count."""
21     if p != self._data.first():                                # consider moving...
22         cnt = p.element()._count
23         walk = self._data.before(p)
24         if cnt > walk.element()._count:                         # must shift forward
25             while (walk != self._data.first() and
26                     cnt > self._data.before(walk).element()._count):
27                 walk = self._data.before(walk)
28             self._data.add_before(walk, self._data.delete(p))    # delete/reinsert

```

代码段 7-19 FavoritesList 类 (前继代码段 7-18)

```

29 #----- public methods -----
30 def __init__(self):
31     """Create an empty list of favorites."""
32     self._data = PositionalList()                               # will be list of _Item instances
33
34 def __len__(self):
35     """Return number of entries on favorites list."""
36     return len(self._data)
37
38 def is_empty(self):
39     """Return True if list is empty."""
40     return len(self._data) == 0
41
42 def access(self, e):
43     """Access element e, thereby increasing its access count."""
44     p = self._find_position(e)                                # try to locate existing element
45     if p is None:
46         p = self._data.add_last(self._Item(e))                # if new, place at end
47         p.element()._count += 1                             # always increment count
48         self._move_up(p)                                    # consider moving forward
49
50 def remove(self, e):
51     """Remove element e from the list of favorites."""
52     p = self._find_position(e)                                # try to locate existing element
53     if p is not None:
54         self._data.delete(p)                                # delete, if found
55
56 def top(self, k):
57     """Generate sequence of top k elements in terms of access count."""
58     if not 1 <= k <= len(self):
59         raise ValueError('Illegal value for k')
60     walk = self._data.first()
61     for j in range(k):
62         item = walk.element()                                # element of list is _Item
63         yield item._value                                  # report user's element
64         walk = self._data.after(walk)

```

7.6.2 启发式动态调整列表

先前收藏夹列表的实现所执行的 `access(e)` 方法与收藏夹列表中 `e` 的索引存在时间上的比例关系。也就是说，如果 `e` 是收藏夹列表中第 k 个最受欢迎的元素，那么访问元素 `e` 的时间复杂度就是 $O(k)$ 。在许多实际的访问序列中（如，用户访问网页），如果一个元素被访问，那么它很有可能在不久的将来再次被访问。这种情况被称为具有访问的局部性。

启发式算法（或称为经验法则），尝试利用访问的局部性，就是在访问序列中采用 Move-to-Front 启发式。为了应用启发式算法，我们每访问一个元素，都会把该元素移动到列表的最前面。当然，我们这么做是希望这个元素在近期可以被再次访问。例如，考虑一个场景，在这个场景中，我们有 n 个元素和以下 n^2 次访问：

- 元素 1 被访问 n 次。
- 元素 2 被访问 n 次。
-
- 元素 n 被访问 n 次。

如果将元素按它们被访问的次数进行存储，当元素第一次被访问时将元素插入队列，则：

- 对元素 1 的每次访问所花费的时间为 $O(1)$ 。
- 对元素 2 的每次访问所花费的时间为 $O(2)$ 。
-
- 对元素 n 的每次访问所花费的时间为 $O(n)$ 。

因此，执行一系列访问的总时间就可以按比例地计算为：

$$n + 2n + 3n + \dots + n \cdot n = n(1 + 2 + 3 + \dots + n) = n \cdot n(n + 1)/2, \text{ 即 } O(n^3)。$$

但是，如果使用 Move-to-Front 启发式算法，在每个元素第一次被访问时将它插入，则

- 元素 1 的每个后续访问所花费的时间为 $O(1)$ 。
- 元素 2 的每个后续访问所花费的时间为 $O(1)$ 。
-
- 元素 n 的每个后续访问所花费的时间为 $O(1)$ 。

所以，在这个案例中，执行所有访问的运行时间为 $O(n^2)$ 。因此，这个场景的 Move-to-Front 实现具有更短的访问时间。然而，Move-to-Front 只是一个启发式算法，因为这种使用 Move-to-Front 方法访问序列比简单地保存根据访问数量排序的收藏夹列表更慢。

Move-to-Front 启发式的权衡

当要求寻找收藏夹列表中前 k 个访问最多的元素时，如果不再保存列表中通过访问次数排序的元素，就需要搜索所有元素。实现 top(k) 方法的步骤如下：

- 1) 将所有收藏夹列表中的元素复制到另一个列表，并将该列表命名为 temp。
- 2) 扫描 temp 列表 k 次，每次扫描时，找出访问量最大的元素记录，从 temp 中移除这条记录，并且在结果中给出报告。

实现 top 方法的时间复杂度是 $O(kn)$ ，因此，当 k 是一个常数时，top 方法的运行时间复杂度为 $O(n)$ 。例如，想得到“top ten”列表，就是这种情况。但是，如果 k 和 n 是成比例的，那么 top 运行时间复杂度为 $O(n^2)$ ，例如，我们需要一个“top 25%”列表时。

在第 9 章中，我们将介绍一种以 $O(n + k\log n)$ 的时间复杂度实现 top 方法的数据结构（见练习 P-9.54），并且可以使用更多先进的技术在 $O(n + k\log n)$ 时间复杂度内来实现 top 方法。

如果在报告前 k 个元素之前，使用一个标准的排序算法来对临时列表重新排序（见 12 章），很容易地实现 $O(n \log n)$ 的时间复杂度。这种方法在 k 是 $\Omega(\log n)$ 的情况下优于原始方法（回想 3.3.1 节中介绍的大 Ω 概念，它给出了一个更接近运行时间下限的排序算法）。还有更多专门的排序算法（见 12.4.2 节），这些算法可以借助访问次数是整数实现对任何一个 k 值，top 方法的时间复杂度为 $O(n)$ 。

用 Python 实现 More-to-Front 启发式

在代码段 7-20 中，我们给出了一个采用 Move-to-Front 启发式实现的收藏夹列表。其中的新 FavoritesListMTF 类继承了原始 FavoritesList 基类的绝大部分功能。

在最初的设计中，原始类的 access 方法依赖于一个非公共的实体 _move_up，在列表中，一个元素的访问次数增加之后，使该元素向潜在的向前的位置调整。因此，我们通过简单地重载 _move_up 方法的方式实现 More-to-Front 启发式，从而使每个被访问的元素都被直接移动到列表的前端（如果之前不在前端的话）。这个动作很容易通过位置列表的方法来实现。

FavoritesListMTF 类中更复杂的部分是 top 方法的新定义。我们借助上文所概述的第一种方法，将条目的副本插入临时列表中，然后重复地查找、报告，移除在剩余元素中访问量最大的元素。

代码段 7-20 FavoritesListMTF 类实现 Move-to-Front 启发式。这个类继承 FavoritesList（代码段 7-18 和代码段 7-19）和重载 _move_up 和 top 两个方法

```

1 class FavoritesListMTF(FavoritesList):
2     """List of elements ordered with move-to-front heuristic."""
3
4     # we override _move_up to provide move-to-front semantics
5     def _move_up(self, p):
6         """Move accessed item at Position p to front of list."""
7         if p != self._data.first():
8             self._data.add_first(self._data.delete(p))      # delete/reinsert
9
10    # we override top because list is no longer sorted
11    def top(self, k):
12        """Generate sequence of top k elements in terms of access count."""
13        if not 1 <= k <= len(self):
14            raise ValueError('Illegal value for k')
15
16        # we begin by making a copy of the original list
17        temp = PositionalList()
18        for item in self._data:                      # positional lists support iteration
19            temp.add_last(item)
20
21        # we repeatedly find, report, and remove element with largest count
22        for j in range(k):
23            # find and report next highest from temp
24            highPos = temp.first()
25            walk = temp.after(highPos)
26            while walk is not None:
27                if walk.element()._count > highPos.element()._count:
28                    highPos = walk
29                    walk = temp.after(walk)
30            # we have found the element with highest count
31            yield highPos.element()._value           # report element to user
32            temp.delete(highPos)                   # remove from temp list

```

7.7 基于链接的序列与基于数组的序列

我们以思考之前介绍过的基于数组和基于链接的数据结构的 pros 和 cons 之间的联系来作为本章的结尾。当选择一个合适的数据结构的实现方法时，这些方法中呈现了一个共同的设计结果，即两面性。就像每个人都有优点和缺点一样，没办法找到一个万全的解决方案。

基于数组的序列的优点

- 数组提供时间复杂度为 $O(1)$ 的基于整数索引的访问一个元素的方法。对于任何 k 值以时间复杂度 $O(1)$ 访问第 k 个元素的能力是一个数组的优点（见 5.2 节）。相应地，在一个链表中定位第 k 个元素要从起始位置遍历列表，其时间复杂度为 $O(k)$ 。如果是反向遍历双向链表，则时间复杂度为 $O(n - k)$ 。
- 通常，具有等效边界的操作使用基于数组的结构运行一个常数因子比基于链表的结构运行更有效率。例如，考虑一个针对队列的典型的 enqueue 操作。忽略调整数组大小的问题，ArrayQueue 类上的这个操作（见代码段 6-7）包括一个新索引的计算算法、一个整数的增量，并在数组中为元素存储一个引用。相反，LinkedQueue 的程序（见代码段 7-8）要求节点的实例化、节点的合适链接和整数的增量。当这个操作用另一个模型在 $O(1)$ 内完成时，链接版本中 CPU 操作的实际数量会更多，特别是考虑到新节点的实例化。
- 相较于链式结构，基于数组的表示使用存储的比例更少。这个优点似乎是有悖于直觉的，特别是考虑到一个动态数组的长度可能超过它存储的元素的数量。基于数组的列表和链接列表都是可引用的结构，所以主存储器用于存储两种结构的元素的实际对象是相同的。而两者的不同点在于这两种结构使用的备用内存的数量。对于基于数组的 n 个元素的容器，一种典型的最坏情况是最近调整动态数组已经为 $2n$ 对象引用分配内存。而对于链表，内存不仅要存储每个所包含的对象的引用，还要明确地存储链接这各个节点的引用。一个长度为 n 的单向链表至少需要 $2n$ 个引用（每个节点的元素引用和指向下一个节点引用）。

基于链表的序列的优点

- 基于链表的结构为它们的操作提供最坏情况的时间界限。这与动态数组的扩张和收缩相关联的摊销边界相对应（见 5.3 节）。

当许多单个操作是一个大型计算的一部分时，我们仅关心计算的总时间，摊销边界和最坏情况的边界一样精确，因为它可以确保花费所有单个操作的时间总和。

然而，如果数据结构操作用于一个实时系统，旨在提供更迅速的反应（如，操作系统、Web 服务器、空中交通控制系统），则单（摊销）操作导致的长时间延迟可能有不利影响。
- 基于链表的结构支持在任意位置进行时间复杂度为 $O(1)$ 的插入和删除操作。能够用 PositionalList 类实现常数时间复杂度的插入和删除操作，并通过使用 Position 有效地描述操作的位置，这可能是链表最显著的优势。

这与基于数组的序列形成了鲜明的对比。忽略调整数组大小的问题，任何从基于数组列表的末尾插入或删除一个元素的操作都可以在常数时间内完成。然而，更普遍的插入和删除代价是很大的。例如，用 Python 的基于数组列表类，调用索引为 k 的插入和删除使用的时间复杂度为 $O(n - k + 1)$ ，因为要循环替换所有后续元素（见 5.4 节）。

作为应用程序实例，考虑维护一个文件作为字符序列的文本编辑器。虽然用户经常在文件的末尾追加字符，还可能用光标在文件的任意位置插入和删除一个或多个字符。如果字符序列存储在一个基于数组的序列中（如，一个 Python 列表），每个编辑操作可能需要线性地调换许多字符的位置，导致每个编辑操作的 $O(n)$ 性能。若用链

表表示，任意一个编辑操作（在光标处的插入和删除）可以以最坏情况的时间复杂度 $O(1)$ 执行，假设所给定的位置是表示光标的位置。

7.8 练习

请访问 www.wiley.com/college/goodrich 以获得练习帮助。

巩固

- R-7.1 给出在单向链表中找到第二个节点到最后一个节点的算法，其中最后一个节点的 `next` 指针指向空。
- R-7.2 给出将两个单向链表 L 和 M 合并成一个新的单向链表 L' 的算法，只给出每个列表的一个头节点的指针，链表 L' 包括 L 和 M 的所有节点，且所有来自 M 的节点都在 L 的节点之后。
- R-7.3 给出计算一个单向链表所有节点数量的递归算法。
- R-7.4 在仅给出两个节点 x 和 y 的指针的情况下，详细描述怎样在一个单向链表中交换这两个节点（注意：不仅仅是交换两个节点的内容）。在 L 是双链表的情况下重复这个练习，哪个算法更耗时？
- R-7.5 实现统计一个循环链表节点个数的函数。
- R-7.6 假定 x 和 y 是循环链表的节点，但不必属于同一个链表。请给出一个快速有效的算法，判断 x 和 y 是否来自同一个链表。
- R-7.7 对于一个非空队列，我们在 7.2.2 节的 `CircularQueue` 类中给出了一个与 `Q.enqueue(Q, dequeu())` 语义相似的 `rotate()` 方法。在不创建任何新节点的情况下为 7.1.2 节的 `LinkedQueue` 类实现一个相似的方法。
- R-7.8 通过连接跳跃，给出寻找一个双向链表的中间节点的非递归算法。在节点数是偶数的情况下，链表的中间节点指的是中间偏左的节点（注意：这个方法必须使用链接跳跃，不能使用一个计数器），并指出这个方法的运行时间。
- R-7.9 给出含有头、尾哨兵，将两个双向链表 L 和 M 合并为 L' 的高效算法。
- R-7.10 在含位置信息链表的抽象数据结构中存在一些冗余的方法，比如操作 `L.add_first(e)` 可以由可选的 `L.add_before(L.first(), e)` 实现，也可以由 `L.add_after(L.last(), e)` 实现。试解释为什么方法 `add_first` 和 `add_last` 是必需的。
- R-7.11 实现一个称为 `max(L)` 的函数，返回包含一系列可比较元素的 `PositionalList` 实例 L 中的最大元素。
- R-7.12 重做上述练习，将 `max` 作为方法放入带有信息链表的类中，以支持方法 `L.max()` 的调用。
- R-7.13 更新 `PositionalList` 类，使其能够支持方法 `find(e)`，该方法将返回元素 e （第一次出现）在链表中的位置（如果没有发现，则返回 `None`）。
- R-7.14 运用递归方法重复刚才的练习。实现方法不要包含任何循环。并说明该方法除了链表 L 所占的空间外还需要占用多少额外的空间。
- R-7.15 为 `PositionalList` 类提供一个类似于 `__iter__` 方法的 `__reversed__` 方法的支持，但以逆置的顺序迭代元素。
- R-7.16 通过只使用方法集 `{is_empty, first, last, prev, next, add_after, add_first}` 中的方法给出实现 `PositionalList` 的方法 `add_last` 和 `add_before` 的描述。
- R-7.17 在 `FavoritesListMTF` 类中，我们借助 `PositionallistADT` 的公共方法将一个元素从链表中的位置 p 移动到链表第一个元素的位置，同时保持其他元素的相对位置不变。在内部，这些操作造成了一个节点被移除而另一个新的节点被插入。给 `positionalList` 类增加一个新的方法

`move_to_front(P)`, 通过重新链接已存的节点更加直接地实现这个目标。

- R-7.18 给出一个存储在链表中的元素集 $\{a, b, c, d, e, f\}$, 假设根据 $\{a, b, c, d, e, f, a, c, f, b, d, e\}$ 的顺序通过使用 Move-to-Front 启发式操作元素, 请给出最终链表中元素的状态。
- R-7.19 假设已对含有 n 个元素的链表 L 做了 kn 次的访问操作, 其中整数 k 大于等于 1。如果被访问的次数少于 k 次, 那么最小和最大的元素个数是多少?
- R-7.20 假设 L 是含有根据递归操作 Move-to-Front 启发式得到的一系列 n 个元素的列表。试描述一些时间复杂度为 $O(n)$ 的逆置链表的访问方法。
- R-7.21 假设根据递归操作 Move-to-Front 得到一个含有 n 个元素的链表 L 。试着给出一个 n^2 次的访问序列, 确保其能够在 $\Omega(n^3)$ 的时间内完成。
- R-7.22 为 FavoritesList 类实现 `clear()` 方法, 用于清空列表。
- R-7.23 为 FavoritesList 类实现 `reset_counts()` 方法, 使其将列表中所有元素的访问计数重置为 0 (并保持链表中各元素的顺序不变)。

创新

- C-7.24 使用一个包含头哨兵的单向链表实现栈的抽象数据结构。
- C-7.25 使用一个包含头哨兵的单向链表实现队列的抽象数据结构。
- C-7.26 为 LinkedQueue 类实现一个 `concatenate(Q2)` 方法, 该方法将获取 LinkedQueueQ2 的所有元素, 并将其附加在原来队列的尾部。该方法必须在 $O(1)$ 的时间内完成, 并且最终的结果是 Q2 将会成为一个空队列。
- C-7.27 给出实现单向链表类的递归算法, 使得非空列表的一个实例存储它的第一个元素和余下元素的指针。
- C-7.28 给出一个快速高效的逆置单向链表的递归算法。
- C-7.29 仅使用固定数量的额外空间, 并且不使用任何递归, 详细阐述一个逆置单向链表 L 的算法。
- C-7.30 练习 P-6.35 描述了一个 LeakyStack 的抽象结构。使用单向链表作为存储实现它的抽象数据结构。
- C-7.31 在一个单向链表上抽象操作, 以设计一个 forward list 抽象数据结构, 就如同在带有位置信息的链表的抽象数据结构上使用抽象双向链表一样。实现一个能够支持这种抽象数据结构的 ForwardList 类。
- C-7.32 设计一个循环的含位置信息的链表的抽象数据结构, 它能够像抽象双向链表一样抽象单向链表。在列表中给出一个指定的光标的位置。
- C-7.33 改造 `_DoublyLinkedListBase` 类, 使其包括一个能够逆置链表元素的 `reverse` 方法, 而不生成或者破坏任何一个节点。
- C-7.34 修改 PositionalList 类, 使其支持方法 `swap(p, q)`, 该方法能够从根本上将节点处于 p 和 q 位置的节点进行交换。重新链接所有现存的节点而不生成任何新的节点。
- C-7.35 为了实现 PositionalList 类的 `iter` 方法, 我们用 Python 的 generator 语法和 `yield` 语句, 通过设计一个嵌套的 iterator 类, 给出 `iter` 方法的可选择的实现方式 (参考 2.3.4 节迭代器的讨论)。
- C-7.36 使用双向链表给出一种 PositionalList 抽象数据结构的实现方法, 该双向链表不包括任何哨兵节点。
- C-7.37 现有一个包含 n 个非降序整数的 PositionalList L , 给出一个整数 V , 试实现一个函数, 使其在 $O(n)$ 时间内能够判断出在 L 中是否存在两个元素的和等于 V 。如果找到了, 该函数需要返回该两个元素的位置信息; 反之, 返回 `None`。

- C-7.38 现有一个简单但并不高效的算法 bubble-sort，用于对列表 L 中所包含的 n 个可比较元素进行排序。该算法对列表进行 $n - 1$ 次扫描，在每次扫描过程中，算法对当前值与下一个值进行比较，并且当它们乱序时交换它们。将一个含位置信息的列表 L 作为参数实现 bubble_sort 函数。假设这个含位置信息的列表是通过一个双向链表实现的，那么该算法的执行时间是多少？
- C-7.39 为了对 FIFO 队列的元素可能在到达队首之前就被删除的情况更好地建模，设计一个 PositionalQueue 类，使其能够支持完全队列抽象数据类型。入队会返回一个位置实例并且支持一个新的 delete(p) 方法，该方法能够移除与位置 p 相关的元素。你需要使用 PositionalList 作为存储，然后使用 6.1.2 节所阐述的适配器进行模式设计。
- C-7.40 给出一个高效的维护链表 L 的方法，在 Move-to-Front 启发式递归下，将自动从列表中删除那些最近 n 次访问中没有被访问的元素。
- C-7.41 练习 C-5.29 介绍了两个数据的自然连接的概念。给出并分析一个将包含 n 个元素的链表 A 和包含 m 个元素的链表 B 进行自然连接的高效算法。
- C-7.42 不采用 5.5.1 中使用的数组，而是使用单向链表来写一个 Scoreboard 类，使其能够保存游戏应用程序中的前十名的分数。
- C-7.43 通过将一个链表分成两个的方式给出一个对含有 $2n$ 个元素的链表的洗牌的算法。一次洗牌，就是把链表 L 分成 L_1 和 L_2 两部分的一个排列，其中 L_1 是 L 的前面一半， L_2 是 L 的后面一半。然后，将两个队列合并，第一个元素放入 L_1 的第一个，第二个元素是 L_2 的第一个元素。接着第三个是 L_1 的第二个，第四个是 L_2 的第二个……以此类推。

项目

- P-7.44 编写一个简单的文字编辑器，使用位置列表的 ADT 和一个能够突出字符串位置的光标对象。该编辑器能够存储并显示字符串，一个简单的接口是打印出字符串，然后使第二行显示一个移动的光标。该编辑器应该支持如下操作：
- **left:** 将光标向左移动一个字符（如果光标在开始处，则不进行任何操作）。
 - **right:** 将光标向右移动一个字符（如果光标在末尾处，则不进行任何操作）。
 - **insert c :** 在光标后面插入一个字符 c 。
 - **delete:** 删除光标后面的字符（如果光标在末尾处，则不进行任何操作）。
- P-7.45 当一个数组 A 中的大部分记录为空时，我们称它为稀疏数组。我们可以使用一个列表 L 来有效地实现这样的数组。特别是，对于每个非空元素 $A[i]$ ，我们可以在列表 L 中存储一条 (i, e) 记录，其中 e 是存储在 $A[i]$ 中的元素。这个方法使我们能够使用 $O(m)$ 的空间代替数组 A 的存储，其中 m 是数组中非空元素的个数。提供一个 SparseArray 类，使其最少支持方法 `__getitem__(j)` 和 `__setitem__(j, e)`，以提供一个标准的索引操作。请分析这个方法的效率。
- P-7.46 尽管我们已经使用了一个双向链表实现了含位置信息的链表抽象数据结构，但是也可以基于一个数组进行实现。其关键在于使用组合模式以及存储位置条目的序列，其中每一项不仅存储一个元素，还存储这个元素当前在数组中的位置信息。无论何时，当数组中元素的位置发生改变时，都需要更新位置的索引记录以保持一致。试给出一个提供这样一种基于数组实现带有位置信息链表的抽象数据类型的类，并分析各种操作的效率。
- P-7.47 实现一个 CardHand 类，该类能够支持洗牌操作。模拟器使用一个含信息的单向链表的 ADT 来表示一副牌，这样相同花色的牌可以放在一起。借助 4 根手指实现这一策略，每两根手指夹住红桃、草花、黑桃和方块中的一种花色。这样可以在常数时间内添加一张牌或者取出一

张牌。这个类应该支持如下的方法：

- `add_card(r, s)`: 在花色 s 的牌位置 r 处添加一张新牌。
- `play(s)`: 从花色 s 的牌中移除或者取出一张牌，如果 s 中没有牌，则从手中任意移除或者取出一张牌。
- `__iter__()`: 遍历当前手中的所有牌。
- `all_of_suit(s)`: 遍历手中花色是 s 的所有牌。

扩展阅读

类似于集合的数据结构的综述（和其他面向对象设计的规则）可以在由 Booch^[17]、Budd^[20]、Goldberg 和 Robson^[42]、Liskov 和 Guttag^[71] 编写的面向对象设计的书中找到。位置信息表 ADT 源于 Aho、Hopcroft 和 Ullman^[6] 介绍的“位置”的抽象以及 Wood^[104] 的 ADT 列表。链表的实现由 Knuth^[64] 进行了讨论。

树

8.1 树的基本概念

生产力专家说，突破来源于“非线性”地思考问题。在本章中，我们来讨论一种最重要的非线性数据结构——树（tree）。在数据的组织中，树结构的确是一个突破，因为我们用它实现的一系列算法比使用线性数据结构（诸如基于数组的列表或者链表）要快得多。树也为数据提供了一个更加真实、自然的组织形式，并由此在文件系统、图形用户界面、数据库、网站和其他计算机系统中得以广泛使用。

生产力专家口中的“非线性”思维并不总是那么清晰明了，但是说树形结构是“非线性”时，我们指的是一种组织关系，这种组织关系要比一个序列中两个元素之间简单的“前”和“后”关系更加丰富和复杂。这种关系在树中是分层的（hierarchical），因为一些元素是处于“上面的”，而另一些是处于“下面的”。事实上，树形数据结构的主要术语来源于家谱，因为术语“双亲”“孩子”“祖先”和“子孙”在描述这些关系时最为常见。图 8-1 所示即为一个家谱图示例。

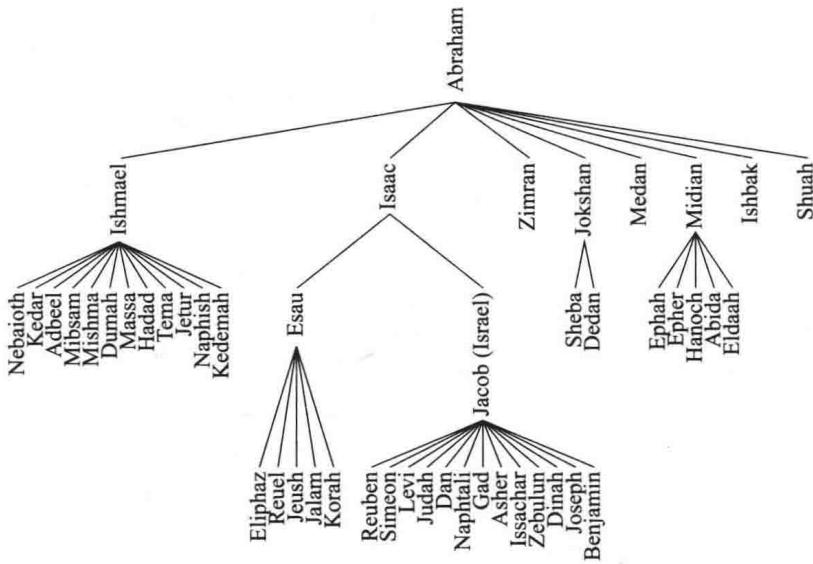


图 8-1 亚伯拉罕（Abraham）后代的家谱图，记录在《创世纪》(Genesis) 的第 25 ~ 36 章

8.1.1 树的定义和属性

树是一种将元素分层次存储的抽象数据类型。除了最顶部的元素，每个元素在树中都有一个双亲节点和零个或者多个孩子节点。通常，我们通过将元素放置在一个椭圆形或者圆形中并且通过直线将双亲节点与孩子节点相连来图示化一棵树，如图 8-2 所示。我们通常称最顶部元素为树根（root），在图示中它被作为最顶部的元素，因为其他元素都被连接在它的下面（这与一棵真实世界中的树恰恰相反）。

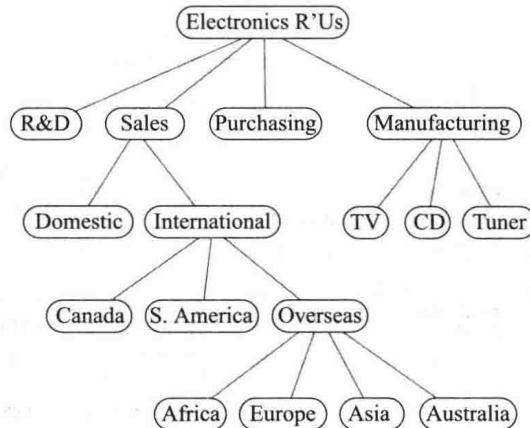


图 8-2 一棵代表一个虚拟公司组织的树，拥有 17 个节点。根存储的是 Electronics R'Us。根的孩子节点分别存储的是 R&D、Sales、Purchasing 和 Manufacturing。内部节点存储 Sales、International、Overseas、Electronics R'Us 和 Manufacturing

正式的树定义

通常我们将树 T 定义为存储一系列元素的有限节点集合，这些节点具有 parent-children 关系并且满足如下属性：

- 如果树 T 不为空，则它一定具有一个称为根节点的特殊节点，并且该节点没有父节点。
- 每个非根节点 v 都具有唯一的父节点 w ，每个具有父节点 w 的节点都是节点 w 的一个孩子。

注意，根据上述定义，一棵树可能为空，这意味着它不含有任何节点。这个约定也允许我们递归地定义一棵树，以使这棵树 T 要么为空，要么包含一个节点 r （其称为树 T 的根节点），其他一系列子树的根节点是 r 的孩子节点。

其他节点关系

同一个父节点的孩子节点之间是兄弟关系。一个没有孩子的节点 v 称为外部节点。一个有一个或多个孩子的节点 v 称为内部节点。外部节点也称为叶子节点。

例题 8-1：在 4.1.1 节中，我们讨论了计算机文件系统中文件与目录之间的分层关系，尽管那个时候没有强调文件系统是树关系。我们重温一下先前的例子，如图 8-3 所示。我们可以看到树的内部节点对应着文件的目录，而叶子节点对应着文件。在 UNIX 和 Linux 操作系统中，树的根节点称为“根目录”，用符号“/”表示。

如果 $u=v$ ，那么节点 u 是节点 v 的祖先或者是节点 v 父节点的祖先。相反，如果节点 u 是节点 v 的一个祖先，那么节点 v 就是节点 u 的一个子孙。例如，在图 8-3 中，cs252/ 是 papers/ 的一个祖先而 pr3 是 cs016/ 的一个子孙。以节点 v 为根节点的子树包含树 T 中节点 v 的所有子孙（包括节点 v 本身）。在图 8-3 中，以 cs016/ 为根节点的子树包含的节点为 cs016/、grades、homeworks/、programs/、hw1、hw2、hw3、pr1、pr2 和 pr3。

树的边和路径

树 T 的一条边指的是一对节点 (u, v) ， u 是 v 的父节点或 v 是 u 的父节点。树 T 当中的路径指的是一系列的节点，这些节点中任意两个连续的节点之间都是一条边。例如，图 8-3 包含了路径 (cs252/, projects/, demos/, market)。

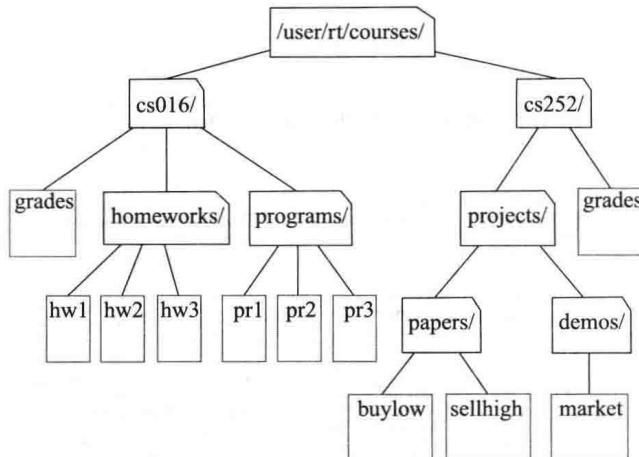


图 8-3 一棵表示一个部分文件系统的树

例题 8-2：在一个 Python 程序中，当使用单继承时，类与类之间的继承关系形成了一棵树。例如，2.4 节给出了 Python 异常类结构层次的总结，正如图 8-4 所示的一样（见图 2-5）。这个 BaseException 类是该层次结构的根，而所有用户自定义的异常类按照惯例都应该声明为更加具体的异常类的后代。（例如，第 6 章代码段 6-1 中的 Empty 类。）

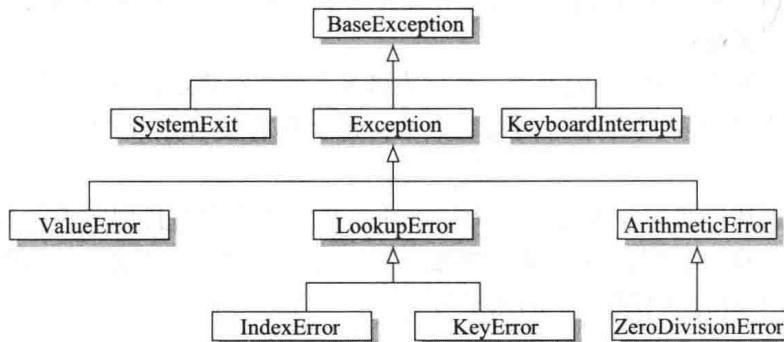


图 8-4 Python 异常类层次结构的一个部分

在 Python 中，所有类被组织成单一的层次结构，因为存在一个名为 `object` 的内置类作为最终的基类。在 Python 中，它是所有其他类型的直接或者间接的基类（即使在定义的时候并没有这样声明）。因此，图 8-4 所示的部分只是 Python 类层次结构的一部分。

作为对本章剩余部分的一个预览，图 8-5 所示即为类的层次结构，这些类用于表示各种形式的树。

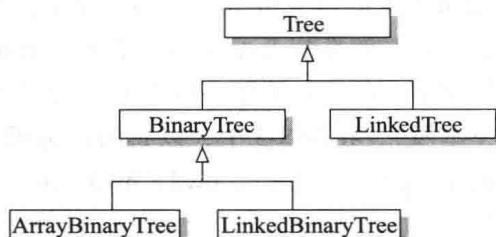


图 8-5 一个模拟各种树数据结构和各种抽象结构的层次结构。在本章的剩余部分，我们详细阐述了树的实现，二叉树、链式二叉树类，以及如何设计树的链式结构和基于数组的二叉树的高标准化示意图

有序树

如果树中每个节点的孩子节点都有特定的顺序，则称该树为有序树，我们将一个节点的孩子节点依次编号为第一个、第二个、第三个等。通常我们按照从左到右的顺序对兄弟节点进行排序。

例题 8-3：考虑结构化文档的内容，诸如一本书按树的样式分层组织，它的内部节点由章节构成，它的叶子节点由段落、表格、图片等构成（见图 8-6），树的根节点是书本身。事实上，我们可以进一步考虑对此进行扩展，如段落又是由句子组成的，而句子又是由单词构成的，单词又是由一个个字母组成的。这就是一棵有序树的典型例子。因为它们的每个孩子节点都具有很好的顺序。

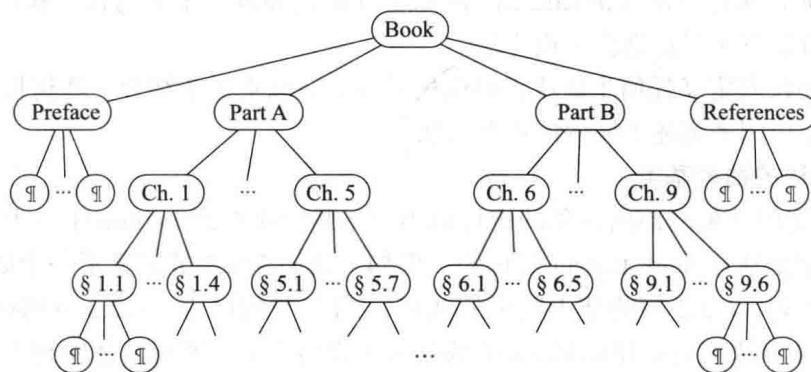


图 8-6 一棵与书相关的有序树

让我们回顾一下已经描述的树的例子，然后进一步深入思考孩子的顺序是否有意义。正如图 8-1 描述的家庭关系树，它总是根据成员的出生时间被模拟成一棵有序树。

相比之下，一个公司的组织结构图（见图 8-2）却通常被认为是一棵无序树。同样，当使用一棵树来描述继承关系的分层结构时，正如图 8-4 所述，对一个父类的子类而言顺序并没有特别的意义。最后，我们考虑用树来描述计算机的文件系统，如图 8-3 所示，尽管操作系统经常按照特定的顺序显示目录（例如，按字母或者时间顺序），但是这样的顺序对于文件系统的显示而言通常不是固定的。

8.1.2 树的抽象数据类型

正如我们在 7.4 节做的位置列表，我们用位置作为节点的抽象结构来定义树的抽象数据结构。一个元素存储在一个位置，并且位置信息满足树中的父节点与孩子节点的关系。一棵树的位置对象支持如下方法：

- `p.element()`: 返回存储在位置 p 中的元素。

树的抽象数据类型支持如下访问方法。允许使用者访问一棵树的不同位置：

- `T.root()`: 返回树 T 的根节点的位置。如果树为空，则返回 `None`。
- `T.is_root(p)`: 如果位置 p 是树 T 的根，则返回 `True`。
- `T.parent(p)`: 返回位置为 p 的父节点的位置。如果 p 的位置为树的根节点，则返回 `None`。
- `T.num_children(p)`: 返回位置为 p 的孩子节点的编号。
- `T.children(p)`: 产生位置为 p 的孩子节点的一个迭代。

- `T.is_leaf(p)`: 如果位置节点 `p` 没有任何孩子，则返回 `True`。
- `len(T)`: 返回树 `T` 所包含的元素的数量。
- `T.is_empty()`: 如果树 `T` 不包含任何位置，则返回 `True`。
- `T.positions()`: 迭代地生成树 `T` 的所有位置。
- `iter(T)`: 迭代地生成存储在树 `T` 中的所有元素。

以上所有方法都接受一个位置作为参数，但是如果树 `T` 中的这个位置是无效的，则调用它就会触发一个 `ValueError`。

如果一棵树 `T` 是有序树，那么执行方法 `T.children(p)` 就会返回孩子节点 `p` 本身的顺序。如果 `p` 是一个叶子节点，那么执行方法 `T.children(p)` 就会导致一个空的迭代。与此类似，如果树 `T` 为空，那么执行方法 `T.positions()` 和 `iter(T)` 也会导致一个空迭代。我们将在 8.4 节通过一棵树的所有位置来讨论迭代生成方法。

我们暂时还没有定义任何生成或者修改树的方法，而更乐于结合一些树接口的特定实现和一些树的特定应用来描述不同树的更新方法。

Python 中树的抽象基类

2.1.2 节讨论的抽象的面向对象的设计原则中，我们注意到 Python 中一个抽象数据类型的公共接口经常是通过 `duck typing` 管理的。例如，我们在 6.2 节定义了一个队列 ADT 的公共接口概念（例如，6.2.2 节的基于数组的队列，7.1.2 节的链表，7.2.2 节的循环链表）。但我们在 Python 中从来没有给出队列 ADT 的任何正式的定义；所有具体实现方法都是独立的类，这些独立的类遵循相同的公共接口。一种更正式的用于指定具有相同抽象但不同的实现方法之间的关系的机制，是通过类的定义，这个类是抽象的基类，它将通过继承产生一个或更多的具体类（见 2.4.3 节）。

在代码段 8-1 中，我们选择定义一个 `Tree` 类充当一个与树的抽象数据结构相关的抽象基类。之所以这样做，是因为我们可以提供相当多的可用代码，即使是在这个抽象级别，在随后树的具体方法实现中也允许更多代码的重用。树的类提供了嵌套类（这些类也是抽象的）的定义和树 ADT 中许多访问方法的申明。

然而，我们定义的 `Tree` 类并没有定义存储树的任何内部表示，并且在代码段中给出的 5 个方法（`root`、`parent`、`num_children`、`children` 和 `__len__`）仍然是抽象的。每个方法都会触发一个 `NotImplementedError()`（一个更加正式的定义抽象方法和基类的方法是使用 2.4.3 节描述的 Python 的 `abc` 模块）。比如孩子节点，为了给每个行为提供一个实现，子类基于它们自选的内部表示来重写抽象方法。

尽管 `Tree` 类是一个抽象的基类，但它包括几个具体的实现方法，这些方法依赖类的抽象方法的调用。在先前章节树的抽象数据结构的定义中，我们声明了 10 种访问方法。其中的 5 个是抽象的，在代码段 8-1 中给出。剩下的 5 个是基于前面 5 个实现的。代码段 8-2 列出了方法 `is_root`、`is_leaf` 和 `is_empty` 的具体实现。在 8.4 节中，我们将会探索树的遍历方法，其能够为位置的确定和 `__iter__` 方法提供一个具体的实现。这种设计的好处是，在树的抽象基类中定义的所有具体方法都能被它的子类所继承。这有助于代码重用，因为对子类而言没有重新实现这些方法的必要。

可以注意到，由于树类是抽象的，因此我们没有理由为其创建一个实例，或者即使创建了一个实例，这个实例也是没有用的。这个类的存在只是作为其他子类用于继承的基础，用户将会创建具体子类的实例。

代码段 8-1 树的抽象基类的一部分（后接代码段 8-2）

```

1 class Tree:
2     """Abstract base class representing a tree structure."""
3
4     #----- nested Position class -----
5     class Position:
6         """An abstraction representing the location of a single element."""
7
7         def element(self):
8             """Return the element stored at this Position."""
9             raise NotImplementedError('must be implemented by subclass')
10
11         def __eq__(self, other):
12             """Return True if other Position represents the same location."""
13             raise NotImplementedError('must be implemented by subclass')
14
15         def __ne__(self, other):
16             """Return True if other does not represent the same location."""
17             return not (self == other)           # opposite of __eq__
18
19     #----- abstract methods that concrete subclass must support -----
20     def root(self):
21         """Return Position representing the tree's root (or None if empty)."""
22         raise NotImplementedError('must be implemented by subclass')
23
24     def parent(self, p):
25         """Return Position representing p's parent (or None if p is root)."""
26         raise NotImplementedError('must be implemented by subclass')
27
28     def num_children(self, p):
29         """Return the number of children that Position p has."""
30         raise NotImplementedError('must be implemented by subclass')
31
32     def children(self, p):
33         """Generate an iteration of Positions representing p's children."""
34         raise NotImplementedError('must be implemented by subclass')
35
36     def __len__(self):
37         """Return the total number of elements in the tree."""
38         raise NotImplementedError('must be implemented by subclass')
39

```

代码段 8-2 抽象基类的一些具体方法

```

40     #----- concrete methods implemented in this class -----
41     def is_root(self, p):
42         """Return True if Position p represents the root of the tree."""
43         return self.root() == p
44
45     def is_leaf(self, p):
46         """Return True if Position p does not have any children."""
47         return self.num_children(p) == 0
48
49     def is_empty(self):
50         """Return True if the tree is empty."""
51         return len(self) == 0

```

8.1.3 计算深度和高度

假定 p 是树 T 中的一个节点，那么 p 的深度就是节点 p 的祖先的个数，不包括 p 本身。

例如，在图 8-2 的树中，节点 International 的深度为 2。需要注意的是，这种定义表明树的根节点的深度为 0。 p 的深度同样也可以按如下递归定义：

- 如果 p 是根节点，那么 p 的深度为 0。
- 否则， p 的深度就是其父节点的深度加 1。

基于这个定义，我们在代码段 8-3 中给出了计算树 T 中一个节点 p 的深度的简单递归算法。该算法递归地调用自身。

代码段 8-3 树类中计算深度的算法

```

52 def depth(self, p):
53     """Return the number of levels separating Position p from the root."""
54     if self.is_root(p):
55         return 0
56     else:
57         return 1 + self.depth(self.parent(p))

```

对于位置 p ，方法 $T.\text{depth}(p)$ 的运行时间是 $O(d_p + 1)$ ，其中 d_p 指的是树 T 中 p 节点的深度，因为该算法对于 p 的每个祖先节点执行的时间是常数。因此算法 $T.\text{depth}(p)$ 在最坏的情况下运行时间为 $O(n)$ 。其中 n 是树中节点的总个数。因为如果所有节点组成一个分支，那么其中存在一个节点的深度将为 $n - 1$ 。尽管这个运行时间是输入大小的函数，但是对于运行时间参数 d_p 更加具有决定性。因为这个参数通常情况下远小于 n 。

高度

树 T 中节点 p 的高度的定义如下：

- 如果 p 是一个叶子节点，那么它的高度为 0。
- 否则， p 的高度是它孩子节点中的最大高度加 1。

一棵非空树 T 的高度是树根节点的高度。例如，图 8-2 所示的树的高度为 4。除此之外，高度还可以定义如下：

命题 8-4：一棵非空树 T 的高度等于其所有叶子节点深度的最大值。

我们在练习 R-8.3 中给出了这个命题的证明。我们在代码段 8-4 中给出了一个算法 height1 ，其作为 Tree 类的一个私有方法。该算法基于命题 8-4 和代码段 8-3 的计算深度的算法来计算一棵非空树的高度。

代码段 8-4 Tree 类中的方法 $_height1$ 。需要注意的是，该方法调用了计算深度的算法

```

58 def _height1(self):           # works, but O(n^2) worst-case time
59     """Return the height of the tree."""
60     return max(self.depth(p) for p in self.positions() if self.is_leaf(p))

```

不幸的是，算法 height1 并不高效。我们目前还没有定义 $\text{position}()$ 方法，可以看到该算法的执行时间是 $O(n)$ ，其中 n 是树 T 中的节点个数。因为 height1 算法针对每个叶子节点都调用了算法 $\text{depth}(p)$ ，其执行时间为 $O(n + \sum_{p \in L}(d_p + 1))$ ，其中 L 是树 T 叶子节点的集合。在最坏情况下， $\sum_{p \in L}(d_p + 1)$ 与 n^2 成正比（详见练习 C-8.33）。因此，算法 height1 在最坏情况下的执行时间为 $O(n^2)$ 。

在最坏情况下，不依赖先前的递归定义，我们可以更加高效地计算树的高度，使其执行时间为 $O(n)$ 。为了这样做，我们将基于一棵树中的某个位置参数化一个函数，并计算以这个节点作为根节点的子树的高度。代码段 8-5 给出的算法 height2 就是通过这种方式来计算树的高度。

代码段 8-5 计算一个以 p 节点为根节点的子树的高度

```

61 def _height2(self, p):           # time is linear in size of subtree
62     """ Return the height of the subtree rooted at Position p """
63     if self.is_leaf(p):
64         return 0
65     else:
66         return 1 + max(self._height2(c) for c in self.children(p))

```

理解算法 `height2` 为什么比算法 `height1` 更高效很重要。该算法是递归的并且是从上到下执行的。如果该算法最初在根节点调用，那么树 T 的每个节点最终将会被调用。这是因为树的根节点最终将在其每个孩子节点上递归调用，这反过来又将在每个节点的孩子节点中继续递归调用下去。

我们可以通过加上所有花在每个节点上的递归调用的时间来计算算法 `height2` 的运行时间（复习 4.2 节递归调用的分析过程）。在实现方法中，对于每个节点，有一个不变的常量以及计算每个孩子节点的最大负载。尽管我们还没有构造 `children(p)` 的实现方法，但可以假设生成时间是 $O(c_p + 1)$ ，其中 c_p 是 p 节点孩子节点的个数。算法 `height2` 在每个节点上最多需要花 $O(c_p + 1)$ 的时间，所以整个时间为 $O(\sum_p (c_p + 1)) = O(n + \sum_p c_p)$ 。为了完成分析，我们使用如下定义。

命题 8-5：假设 T 是一棵有 n 个节点的树，并假设 c_p 代表树 T 中位置 p 的孩子节点的个数，那么 T 中所有节点的位置之和为 $\sum_p c_p = n - 1$ 。

证明：树 T 中除了根节点外的每个位置，都是另一个节点的孩子节点，并且都会成为上面公式的一项。 ■

由命题 8-5 可知，在根节点调用算法 `height2` 时，其执行时间为 $O(n)$ ，其中 n 为树中节点的个数。

重新访问 `Tree` 类的公共接口，计算子树的高度是有益的，但是用户可能希望能够计算整个树的高度而不需要显式地指定树的根节点。我们可以通过一个公有的 `height` 方法将非公有的方法 `_height2` 封装在实现方法中。在树 T 中调用 `T.height()` 方法时，`height` 方法提供了一个默认的解释。其实现的过程如代码段 8-6 所示。

代码段 8-6 计算整个树或者一个给定位置作为根节点的子树的高度的 `Tree.height` 方法

```

67 def height(self, p=None):
68     """ Return the height of the subtree rooted at Position p.
69
70     If p is None, return the height of the entire tree.
71     """
72     if p is None:
73         p = self.root()
74     return self._height2(p)      # start _height2 recursion

```

8.2 二叉树

二叉树是具有以下属性的有序树：

- 1) 每个节点最多有两个孩子节点。
- 2) 每个孩子节点被命名为左孩子或右孩子。
- 3) 对于每个节点的孩子节点，在顺序上，左孩子先于右孩子。

若子树的根为内部节点 v 的左孩子或右孩子，则该子树相应地被称为节点 v 的左子树或