

# 映射、哈希表和跳跃表

## 10.1 映射和字典

`dict` 类可以说是 Python 语言中最重要的数据结构。它表示一种称作字典的抽象，在其中每个唯一的关键字都被映射到对应的值上。由于字典所表示的键和值之间的关系，我们通常将其称为关联数组（associative array）或映射（map）。在本书中，我们使用术语字典（dictionary）来讨论 Python 的 `dict` 类，并且使用术语映射（map）来讨论抽象数据类型的更一般的概念。

图 10-1 给出了一个简单的例子，展示了一个从国家名字到其货币单位的对应关系的映射。

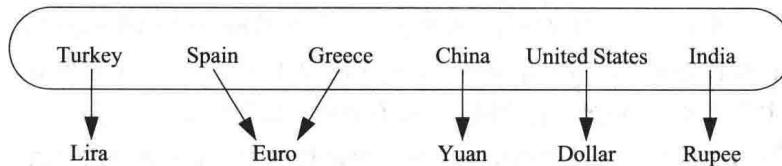


图 10-1 一个从国家（键）到它们对应的货币单位（值）的映射

我们指定键（国家名字）是唯一的，但是值（货币单位）不需要唯一。比如，我们对西班牙和希腊均指定欧元为货币。映射使用类似数组的语法来进行索引，比如用 `currency['Greece']` 来访问与给定键相关的值，或者用 `currency['Greece'] = 'Drachma'` 将其重新映射到一个新的值。与标准的数组不同，映射的索引不需要连续性和数字化。以下是几种常见的映射的应用。

- 一所大学的信息系统依赖于某种形式的映射，这种映射以学生 ID 作为键，并且将其映射到学生相关的记录（例如学生的姓名、地址和课程成绩）作为值。
- 域名系统（DNS）将主机名映射到一个网络协议（IP）地址，例如将 `www.wiley.com` 映射到 `208.215.179.146`。
- 社交媒体网站通常依赖于一个用户名（非数字）作为键，这样的键可以高效地映射到特定用户的相关信息上。
- 计算机图形系统可以将一个颜色名称映射到用于描述颜色 RGB（红 - 绿 - 蓝）的三元组上，如“天蓝色”可以映射为 `(64, 224, 208)`。
- Python 使用字典来表示每个命名空间，将一个标识字符串映射到相关的对象上，如将 PI 映射到 `3.14159`。

在这一章和下一章中我们将介绍如何实现这样的映射，以实现高效地搜索键和它相应的值，从而支持在应用中的快速查找。

### 10.1.1 映射的抽象数据类型

在这一部分，我们引入映射 ADT，并且定义其行为以使其与 Python 内建类 `dict` 一致。

首先，我们列出了映射 M 最为重要的五类行为：

- `M[k]`：如果存在，返回在映射 M 中与键 k 相对应的值，否则返回 `KeyError` 错误。在 Python 中，这个功能是由特定的方法 `__getitem__` 实现的。
- `M[k] = v`：将映射 M 中的值 v 与键 k 建立关联，如果映射中的键 k 已经有对应的值存在，则替换该值。在 Python 中，这个功能由特定的方法 `__setitem__` 实现。
- `del M[k]`：从映射 M 中删除键为 k 的元组，如果 M 中不存在这样的元组，则返回 `KeyError` 错误。在 Python 中，这个功能由特定的方法 `__delitem__` 实现。
- `len(M)`：返回在映射 M 中元组的数量。在 Python 中，这个功能由特定的方法 `__len__` 实现。
- `iter(m)`：默认的对一个映射迭代生成其中所包含的所有键的序列。在 Python 中，这个功能由特定的方法 `__iter__` 实现，并且它支持以 `for k in M` 形式控制的循环。

我们强调了上述五类行为，因为它们展示了映射的核心功能，即请求、增加、修改或者删除 key-value 键值对，以及输出所有这些键值对的功能。为了实现其他的方便功能，映射 M 应该也支持如下行为：

- `K in M`：如果映射中包含键为 k 的元组则返回 `True`。在 Python 中这个功能由特定的方法 `__contains__` 实现。
- `M.get(k, d = None)`：如果在映射中存在键 k 则返回 `M[k]`，否则返回缺省值 d。这种方法提供了一种避免返回 `KeyError` 风险的 `M[k]` 查询方法。
- `M.setdefault(k, d)`：如果在映射 M 中存在键 k，则简单返回 `M[k]`，如果键 k 不存在，则设置 `M[k] = d`，并返回这个值。
- `M.pop(k, d = None)`：从映射 M 中删除键为 k 的元组，并且返回与其对应的值 v。如果键 k 不在映射中，则返回缺省值 d（或者如果参数 d 为 `None`，则抛出 `KeyError`）。
- `M.popitem()`：从映射 M 中随机删除一个 key-value 键 - 值对，并返回一个用于表示被删除的键 - 值对的 (k, v) 数据元组。如果映射 M 为空，则抛出 `KeyError`。
- `M.clear()`：从映射中删除所有的 key-value 键值对。
- `M.keys()`：返回一个含有映射 M 中所有键的集合的视图。
- `M.values()`：返回一个含有映射 M 中所有值的集合的视图。
- `M.items()`：返回一个含有 M 中所有键值对元组的集合。
- `M.update(M2)`：对于 M2 中每一个 (k, v) 对进行复制，设置 `M[k] = v`。
- `M == M2`：如果映射 M 和 M2 中所有的 key-value 键值对完全相同，则返回 `True`。
- `M != M2`：如果映射 M 和 M2 包含有不同的 key-value 键值对，则返回 `True`。

**例题 10-1：**下表中展示了用单个字符作为键、用整数数字作为值来对一个初始化为空的映射进行一系列操作所产生的效果。我们使用 Python 中 `dict` 类的语法来描述映射的内容。

操 作	返 回 值	映 射
<code>len(M)</code>	0	{ }
<code>M['K'] = 2</code>	-	{'K': 2}
<code>M['B'] = 4</code>	-	{'K': 2, 'B': 4}
<code>M['U'] = 2</code>	-	{'K': 2, 'B': 4, 'U': 2}
<code>M['V'] = 8</code>	-	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
<code>M['K'] = 9</code>	-	{'K': 9, 'B': 4, 'U': 2, 'V': 8}

(续)

操作	返回值	映射
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	-	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

### 10.1.2 应用：单词频率统计

现在，考虑统计一个文档中单词出现频率的问题，以此作为使用映射的实例研究。例如，当对邮件和新闻文章进行分类时，这种单词频率统计是统计分析文档过程中的标准任务。映射在这里是一个理想的数据结构，因为我们能够使用单词作为键，单词的数量作为值。在代码段 10-1 中，我们展示了这样一个应用。

**代码段 10-1** 一个统计单词出现频率并报告出现最频繁单词的程序。我们使用 Python 的 dict 类实现这个映射，将输入转化为小写字母并忽略所有的非字母字符

```

1 freq = {}
2 for piece in open(filename).read().lower().split():
3     # only consider alphabetic characters within this piece
4     word = ''.join(c for c in piece if c.isalpha())
5     if word:    # require at least one alphabetic character
6         freq[word] = 1 + freq.get(word, 0)
7
8 max_word = ''
9 max_count = 0
10 for (w,c) in freq.items():      # (key, value) tuples represent (word, count)
11     if c > max_count:
12         max_word = w
13         max_count = c
14 print('The most frequent word is', max_word)
15 print('Its number of occurrences is', max_count)

```

我们结合使用 file 和 string 方法来分割原文档，从而导致文档的所有空白分割文件的小写版本循环。我们忽略所有的非字母字符，这样，括号、引号和其他标点符号都不视为单词的组成部分。

对于映射的操作，我们以 Python 中一个名为 freq 的空字典开始。在算法的第一段，我们对于每一个单词的出现执行如下命令：

```
freq[word] = 1 + freq.get(word, 0)
```

由于当前这个单词可能不存在于字典中，因此我们使用 get 方法。在该实例中采用 0 作为缺省值比较合适。

在算法的第二段，即在整个文档已经处理结束后，我们检查频率映射的内容，循环遍历 freq.items() 从而决定哪个单词具有最高的频率。

### 10.1.3 Python 的 MutableMapping 抽象基类

2.4.3 节已经给出了对一个抽象基类概念的介绍，以及这些类在 Python 集合模块中的作用。在这样的抽象基类中，被定义为抽象的方法必须由具体的子类实现。然而，一个抽象的基类可以提供其他方法的具体实现，这取决于所使用的假定的抽象方法。（这是模板设计模式的一个例子。）

该集合组件提供了两个与我们现在所讨论的内容相关的抽象基类：Mapping 和 MutableMapping。Mapping 类包含由 Python 的 dict 类支持的所有不变方法，而 MutableMapping 类扩展包含所有可变方法。我们在 10.1.1 节所定义 map 的 ADT 与在 Python 集合组件中的 MutableMapping 抽象基类是相似的。

这些抽象基类的意义在于它们提供了一个框架以帮助创建用户定义的 map 类。特别是，MutableMapping 类为所有行为提供具体的实现，这些行为不包含 10.1.1 节所描述的五个行为：`__getitem__`、`__setitem__`、`__delitem__`、`__len__` 和 `__iter__`。只要提供这五大核心的行为，当使用各种数据结构实现 map 抽象类的时候，就可以通过简单地将 MutableMapping 申明为父类来继承所有的衍生行为。

为了更好地理解 MutableMapping 类，提供几个可以由五个核心抽象方法派生的具体行为的例子。例如，支持语法 `k in M` 的 `__contains__` 方法，可以通过生成一个有保护的检索 `self[k]` 来实现，从而判断键是否存在。

```
def __contains__(self, k):
    try:
        self[k]                      # access via __getitem__ (ignore result)
    except KeyError:
        return False                # attempt failed
```

可以用相似的方式来实现 `setdefault` 方法。

```
def setdefault(self, k, d):
    try:
        return self[k]              # if __getitem__ succeeds, return value
    except KeyError:
        self[k] = d                # set default value with __setitem__
        return d                   # and return that newly assigned value
```

我们把 MutableMappling 类剩余的具体方法的实现留作练习。

### 10.1.4 我们的 MapBase 类

我们将提供许多 map ADT 的不同实现，在本章剩余部分以及下一章中，我们使用各种数据结构展示了对这些实现的优点和缺点的权衡。图 10-2 提供了这些类的预览。

MutableMapping 这个来自于 Python 的 collections 模块的抽象基类，是实现 map 的一个有价值的工具。然而，为了更好地实现代码的重用，我们定义了自己的 MapBase 类，它本身是 MutableMapping 类的子类。我们定义的 MapBase 类对组成设计模式提供额外的支持。

这种技术供我们在内部使用，通过将一个键 - 值对作为一个实例进行分组的方式实现优先级队列（见 9.2.1 节）时曾介绍过这一方法。

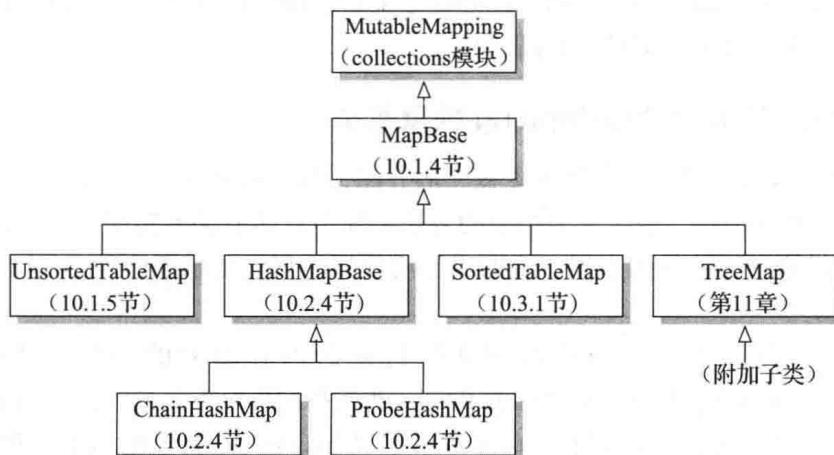


图 10-2 map 的层次结构（参考定义这些类的章节）

更正式地说，在代码段 10-2 中我们定义的 MapBase 类扩展了现有的 MutableMapping 抽象基类，这样我们便可以继承该类提供的许多具体的方法。然后，我们定义了一个非公有的嵌套 \_Item 类，它的实例可以同时存储 key 和 value。这个嵌套类与我们在 9.2.1 节定义在 PriorityQueueBase 类中的 Item 类很相似，除了对于 map 我们提供了对相等测试和比较的支持，且这两种操作都依赖于元组的键。相等的概念对于我们所有的 map 实现都是必要的，因为，我们可以利用这种方式来判断一个给定的键是否与已经存储在 map 中的某一个键相等。稍后，我们将介绍有序的 map ADT (10.3 节)，使用操作符 < 来比较两个键之间的关系是比较合适的。

#### 代码段 10-2 通过扩展 MutableMapping 抽象基类实现非公有类 \_Item，以满足各种映射应用

```

1 class MapBase(MutableMapping):
2     """Our own abstract base class that includes a nonpublic _Item class."""
3
4     #----- nested _Item class -----
5     class _Item:
6         """Lightweight composite to store key-value pairs as map items."""
7         __slots__ = '_key', '_value'
8
9     def __init__(self, k, v):
10        self._key = k
11        self._value = v
12
13    def __eq__(self, other):
14        return self._key == other._key      # compare items based on their keys
15
16    def __ne__(self, other):
17        return not (self == other)        # opposite of __eq__
18
19    def __lt__(self, other):
20        return self._key < other._key    # compare items based on their keys

```

#### 10.1.5 简单的非有序映射实现

我们通过一个简单的 map ADT 的具体实现来说明 MapBase 类的使用。代码段 10-3 给出

了一个通过在 Python 列表内以任意顺序存储 key-value 对来实现的 UnsortedTableMap 类。

代码段 10-3 一个用 Python 列表作为非排序表的 map 实现方法，代码段 10-2 给出了父类 MapBase 的实现

---

```

1 class UnsortedTableMap(MapBase):
2     """Map implementation using an unordered list."""
3
4     def __init__(self):
5         """Create an empty map."""
6         self._table = []                      # list of _Item's
7
8     def __getitem__(self, k):
9         """Return value associated with key k (raise KeyError if not found)."""
10    for item in self._table:
11        if k == item._key:
12            return item._value
13    raise KeyError('Key Error: ' + repr(k))
14
15    def __setitem__(self, k, v):
16        """Assign value v to key k, overwriting existing value if present."""
17        for item in self._table:
18            if k == item._key:                  # Found a match:
19                item._value = v               # reassign value
20                return                      # and quit
21        # did not find match for key
22        self._table.append(self._Item(k,v))
23
24    def __delitem__(self, k):
25        """Remove item associated with key k (raise KeyError if not found)."""
26        for j in range(len(self._table)):
27            if k == self._table[j]._key:          # Found a match:
28                self._table.pop(j)              # remove item
29                return                      # and quit
30        raise KeyError('Key Error: ' + repr(k))
31
32    def __len__(self):
33        """Return number of items in the map."""
34        return len(self._table)
35
36    def __iter__(self):
37        """Generate iteration of the map's keys."""
38        for item in self._table:
39            yield item._key                  # yield the KEY

```

---

在我们的 map 构造器中，将一个空的表格初始化为 `self._table`。当一个新的键被放入 map 中，通过 22 行的 `__setitem__` 方法，我们创建了一个嵌套类 `_Item` 的实例，该嵌套类继承自 `MapBase` 类。

这个基于列表的 map 实现很简单，但不是很有效率。每一个基本方法，`__getitem__`、`__setitem__` 和 `__delitem__`，都依赖于一个 `for` 循环扫描列表中的元组，以搜索匹配的键。在最好的情况下，这样的匹配可以在列表的开头附近找到，并且循环终止；在最坏的情况下，则需要搜索整个列表。因此，在包含  $n$  个元组的 map 中，这些方法都可以在  $O(n)$  时间复杂度内完成。

## 10.2 哈希表

在这一部分，我们介绍一个最实用的实现 map 的数据结构，而且 Python 还用它来实现

dict类，这种结构被称为哈希表。

直观地说，映射 $M$ 支持使用键作为索引的抽象，它的语法如 $M[k]$ 。先考虑一种有限制的设置，在这个设置的映射中含有 $n$ 个元组，对于一些 $N \geq n$ 情况，使用范围在0到 $N-1$ 的整数值作为键。在这种情况下，我们可以使用长度为 $N$ 的查找表来表示这个映射，如图10-3所示。

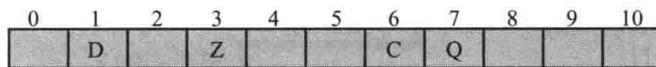


图10-3 一个包含(1, D)、(3, Z)、(6, C)和(7, Q)且长度为11的哈希表

在这种表示下，我们将键值 $k$ 对应的值存储在表中索引值为 $k$ 的位置上（假定我们有一个明确的方式表示空槽）。`__getitem__`、`__setitem__`和`__delitem__`等基本映射操作能够在最坏情况下以 $O(1)$ 的时间复杂度完成。

将这个框架扩展到更一般的映射设置有两个挑战。首先，如果在 $N \gg n$ 的情况下，我们并不希望将一个长度为 $N$ 的数组分配给这个映射。第二，我们一般不会要求一个映射的键必须是整数。哈希表的一个新概念是使用哈希函数将每个一般的键映射到一个表中的相应索引上。在理想情况下，键将由哈希函数分布到从0到 $N-1$ 的范围内，但是在实践中可能有两个或者更多的不同键被映射到同一个索引上。因此，我们将表概念化为桶数组，具体如图10-4所示，其中每个桶都管理一个元组集合，而这些元组则通过哈希函数发送到具体的索引。（为了节约空间，空桶可以用None代替。）

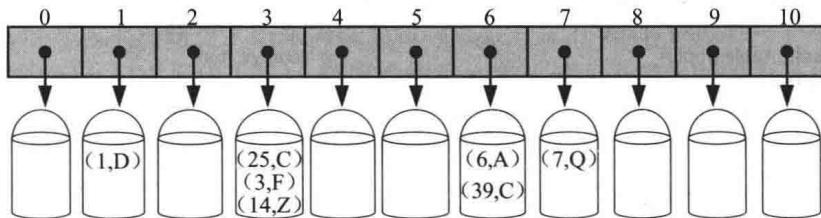


图10-4 一个使用哈希函数的桶，桶中包含(1, D)、(25, C)、(3, F)、(14, Z)、(6, A)、(39, C)和(7, Q)，容量为11

### 10.2.1 哈希函数

哈希函数 $h$ 的目标就是把每个键 $k$ 映射到 $[0, N-1]$ 区间内的整数，其中 $N$ 是哈希表的桶数组的容量。使用这种哈希函数 $h$ 的主要思想是使用哈希函数值 $h(k)$ 作为哈希桶数组 $A$ 内部的索引，而不用键 $k$ 做索引（直接用键 $k$ 作索引可能不合适）。也就是说，我们在桶 $A[h(k)]$ 中存储元组 $(k, v)$ 。

如果有两个或者更多的键具有相同的哈希值，那么两个不同的元组将被映射到相同的桶 $A$ 中。在这种情况下，我们说发生了一次冲突。虽然不可否认，有很多方法可解决冲突，且我们将在稍后讨论，但是最好的策略是在最初尽量避免其发生。如果一个哈希函数能在映射map中的键时最小化冲突的发生，我们就说该哈希函数是“好的”。出于实际的需要，我们也同时希望哈希函数是快速且易于计算的。

评价哈希函数 $h(k)$ 常见的方法由两部分组成：一个哈希码，将一个键映射到一个整数；一个压缩函数，将哈希码映射到一个桶数组的索引，这个索引是范围在区间 $[0, N-1]$ 的一

个整数（见图 10-5）。

将哈希函数分成这样的两个组件的优势是：哈希码计算部分独立于具体的哈希表的大小。这样就可以为每个对象开发一个通用的哈希码，并且可以用于任何大小的哈希表，只有压缩函数与表的大小有关。这样就特别方便，因为哈希表底层的桶数组可以根据当前存储在映射（map）中的元组数动态调整大小（见 10.2.3 节）。

## 10.2.2 哈希码

哈希函数执行的第一步是取出映射中的任意一个键  $k$ ，并且计算得到一个整数作为键  $k$  的哈希码；这个整数不需要在  $[0, N - 1]$  范围内，甚至可以是负数。我们希望分配给键的哈希码集合尽可能避免冲突。因为，如果键的哈希码产生了冲突，那么我们的压缩函数也无法回避这种冲突。在本节中，我们首先讨论哈希码的理论。接下来，我们讨论 Python 中哈希码的具体实现。

### 将位作为整数处理

首先，我们注意到，对于任何数据类型  $X$  使用尽可能多的位作为我们的整数哈希码，可以简单地把用于表示整数  $X$  的各个位作为它的哈希码。例如，键 314 可以简单地用 314 作为哈希码。浮点数的哈希码（如 3.14）可以由该浮点数各个位上的数所构成的整数来表示（314）。

以上方案不能直接适用于一个按位表示长于所需的哈希代码长度的类型。例如，Python 中的哈希码的长度是 32 位。如果一个浮点数是采用 64 位表示的，则它的按位表示的形式就不能直接作为哈希码使用。一种可能的解决方法是只使用高阶 32 位（或低阶 32 位）。当然这种哈希码将忽略在原来键中的一半信息，如果我们的映射中许多键只在这些忽略的位上不同，那么采用这种简化的哈希码将会发生冲突。

更好的解决办法是将 64 位键的高阶 32 位和低阶 32 位采用一定方式进行合并，生成一个 32 位的哈希码，这样就将所有的原始位信息都考虑在内了。一个简单的实现是把两个部分作为 32 位数字相加（忽略溢出），或者将两部分做异或操作。这些合并两部分的方法能够扩展到任意对象  $x$ ，且对象  $x$  的二进制表示可以视为 32 位整数的  $n$  元组  $(x_0, x_1, \dots, x_{n-1})$ ，则可以用  $\sum_{i=0}^{n-1} x_i$  或者  $x_0 \oplus x_1 \oplus \dots \oplus x_{n-1}$  来生成  $x$  的哈希码，这里符号  $\oplus$  代表按位异或操作（在 Python 中用  $\wedge$  表示）。

### 多项式哈希码

上面所描述的用求和或异或计算哈希码的方法对于字符串或其他用  $(x_0, x_1, \dots, x_{n-1})$  元组形式表示的可变长度对象并不是好选择，这里元组中  $x_i$  的顺序很重要。比如考虑一个字符串  $s$ ，用  $s$  中各字符的 Unicode 值的和生成 16 位哈希码。不幸的是，这种哈希代码对于常见的字符串组而言会产生大量的不必要的冲突。使用此方法，形如 "temp01" 和 "temp10" 的字符串的哈希码会产生冲突，"stop" "tops" "pots" 和 "spot" 的哈希码也会产生冲突。更好的哈希码应该通过某种方式考虑  $x_i$  的位置。一种可选的哈希码计算方法可以满足这样的要求：选择一个非零常数  $a$  且  $a \neq 1$ ，并这样计算哈希码：

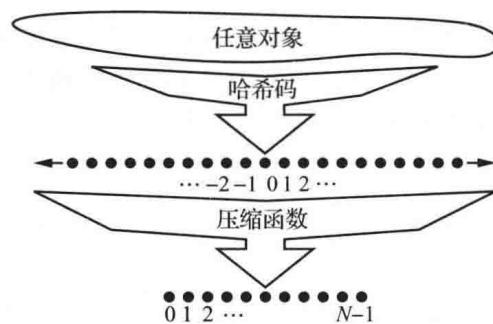


图 10-5 哈希函数的两个部分：哈希码和压缩函数

$$x_0a^{n-1} + x_1a^{n-2} + \dots + x_{n-2}a + x_{n-1}$$

从数学上讲，这仅仅是包含  $a$  并以表示对象  $x$  的元组  $(x_0, x_1, \dots, x_{n-1})$  中的元素为系数的一个多项式表示。因此这种哈希码称为多项式哈希码。利用 Horner 规则（见练习 C-3.50），这个多项式可以按如下表达式计算：

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \dots + a(x_2 + a(x_1 + ax_0)) \dots))$$

直观地说，一个多项式的哈希码通过乘以不同权值的方式来分散每一部分对哈希码结果的影响。

当然，在一个典型的计算机上，将通过使用有限位数表示哈希代码来评估一个多项式，因此，这些用于表示整数的位的值会周期性溢出。由于我们更感兴趣的是一个相对于其他键具有很好的传播性的对象  $x$ ，因此我们直接忽略了这样的溢出。不过，我们仍然会关注这种溢出的发生，并且选择一个常量  $a$ ，以便于它包含一些非零的低阶位，使其能够在即使发生了溢出的状态下，仍然能保留一些信息内容。

我们已做过的一些实验研究表明：在处理英文字符串时，33、37、39 和 41 是特别适合选作  $a$  值的。事实上，在超过 50 000 个英语单词形成的联合单词列表中提供两种 Unix 变种，我们发现当  $a$  取 33、37、39 或者 41 时在每个用例中产生的冲突将少于 7 个。

### 循环移位哈希码

一个多项式哈希码的变种，是用一定数量的位循环位移得到的部分和来替代乘以  $a$ 。例如，一个 32 位数 00111011001011010100010101000 的五位循环移位值，是取其最左边五位，并且将它们放置到数据的最右边，得到结果 10110010110101010001010100000111。虽然这种操作在算术方面具有很小的实际意义，但是它完成了改变二进制位的计算目标。在 Python 中，二进制位循环移位可以通过使用按位运算符 `<<` 和 `>>` 完成，从而截取结果为 32 位整数。

在 Python 中字符串循环移位的哈希码计算的实现如下：

```
def hash_code(s):
    mask = (1 << 32) - 1           # limit to 32-bit integers
    h = 0
    for character in s:
        h = (h << 5 & mask) | (h >> 27)   # 5-bit cyclic shift of running sum
        h += ord(character)                 # add in value of next character
    return h
```

就像传统的多项式哈希码，在使用循环移位哈希码时需要微调，因为我们必须仔细地对于每一个新字符选择移位的位数。通过在超过 230 000 个英文单词的列表上，对不同移位位数所产生的冲突数的实验结果的比较，我们决定选择 5 位移位（见表 10-1）。

表 10-1 循环移位哈希码应用于 230 000 个英语单词列表的冲突行为的比较。Total 列记录至少与一个其他单词发生冲突的单词的总数量，而 Max 列记录与任何一个哈希码产生冲突的单词的最大数量。注意，当循环移位位数为 0 时，循环移位哈希码就退化成对所有字符求和的方法

移 位	冲 突	
	Total	Max
0	234 735	623
1	165 076	43
2	38 471	13
3	7 174	5

(续)

移位	冲突	
	Total	Max
4	1 379	3
5	190	3
6	502	2
7	560	2
8	5 546	4
9	393	3
10	5 194	5
11	11 559	5
12	822	2
13	900	4
14	2 001	4
15	1 9251	8
16	211 781	37

### Python 中的哈希码

在 Python 中计算哈希码的标准机制是一个内置签名 `hash(x)` 函数，该函数将返回一个整型值作为对象 `x` 的哈希码。然而在 Python 中，只有不可变的数据类型是可哈希的。这个限制是为了确保在一个对象的生命周期期间，其哈希码保持不变。这是对于对象在哈希表中作为键的一个重要属性。如果哈希表中插入新的键则可能会产生问题，即在这个键插入之后，针对这个键的查找会根据不同的哈希码进行，而不是该键被插入时的哈希码，而且会在错误的桶上进行搜索。

在 Python 的内置数据类型中，不可变的数据类型如 `int`、`float`、`str`、`tuple`、和 `frozenset` 等会通过哈希函数和之前讨论过的类似技术生成健壮的哈希码。基于类似于多项式哈希码的技术，精心设计了的字符串的哈希码，没有使用异或也不是相加计算。如果我们使用 Python 内置的哈希码重复在表 10-1 中所表述的实验，将会发现只有 8 个字符串超过 230000 的集合与其他字符串发生冲突。使用相似的基于元组的单个元素的哈希码的组合技术来计算元组的哈希码。元组的哈希码是通过基于元组每个元素的哈希码的组合相似的技术计算而来的。当对一个 `frozenset` 集对象进行哈希时，元素的顺序应该是无关的，因此一个自然的选择是用异或值计算单个哈希码而不用任何移位。如果 `hash(x)` 被一个可变类型的实例 `x` 调用，比如 `list`，则将会发生 `TypeError`。

在默认情况下，用户定义的类的实例被视为是不可哈希的，并且哈希函数会产生 `TypeError`。然而，计算哈希码的函数能够由在类中的一个名为 `__hash__` 的特殊方法实现。返回的哈希码应该反映一个实例的不可变属性。通过计算组合属性的哈希码来返回哈希值是很常见的。比如，一个 `Color` 类维护着红、黄、蓝三种颜色的数字组件，可以用如下方法实现：

```
def __hash__(self):
    return hash( (self._red, self._green, self._blue) ) # hash combined tuple
```

一个需要遵守的重要规则是，如果通过 `__eq__` 定义一个类的等价类，则 `__hash__` 的任何实现必须是一致的，即如果 `x == y`，则 `hash(x) == hash(y)`。这一点是非常重要的，因为如果两个实例被判定为是等价的，并且其中一个在哈希表中被作为键使用，则搜寻第二个实例的操作返回的结果应该是找到了第一个键。因此，第二个哈希码与第一个哈希码匹配是非

常重要，只有这样才能在恰当的桶中查找。这一规则可以扩展到任何不同类别的对象之间的比较。比如，由于 Python 中视表达式  $5 == 5.0$  为 true，因此要确保  $\text{hash}(5)$  和  $\text{hash}(5.0)$  是相等的。

### 10.2.3 压缩函数

通常，键  $k$  的哈希码不会直接适合使用一个桶数组，因为整数哈希码可能是负的或可能超过桶数组的容量。因此，当我们决定对于一个对象  $k$  的键使用整数哈希码时，还有一个问题是需要把整数映射到  $[0, N - 1]$  区间上。这是整个哈希函数处理中实施的第二个动作，称为压缩函数。一个很好的压缩函数会使给定的一组哈希码的冲突数达到最小。

#### 划分方法

一个简单的压缩函数是划分方法，它将一个整数  $i$  映射到  $N$ :

$$i \bmod N$$

在这里  $N$  是桶数组的大小，是一个固定的正整数。此外，如果我们把  $n$  设置为一个素数，那么这个压缩函数有助于“传播”哈希值的分布。事实上，如果  $n$  不是素数，那么有更大的风险，即哈希码分布的模式将在哈希值的分布中重复出现，因而造成冲突。比如，如果我们将哈希码为  $\{200, 205, 210, 215, 220, \dots, 600\}$  的一组键插入大小为 100 的哈希数组桶中，则每一个哈希码都将与其他的某三个哈希码相冲突。但是如果我们将哈希码插入大小为 101 的桶数组，则不会发生冲突。如果选择了一个好的哈希函数，应该确保两个不同的键获取相同哈希桶的可能性为  $1/N$ 。选择  $N$  为素数并不总能充分地解决问题，对于不同的  $p, pN + q$  形式的哈希码是重复的，那么仍然将发生冲突。

#### MAD 方法

有一个更复杂的压缩函数可以帮助一组整数键消除重复模式，即 Multiply-Add-and-Divide（或“MAD”）方法。这个方法通过

$$[(ai + b) \bmod p] \bmod N$$

对  $i$  进行映射，这里  $N$  是桶数组的大小， $p$  是比  $N$  大的素数， $a$  和  $b$  是从区间  $[0, p - 1]$  任意选择的整数，并且  $a > 0$ 。选择这个压缩函数是为了消除在哈希码集合中的重复模式，并且得到更好的哈希函数，因为该函数使得任意两个不同键冲突的概率为  $1/N$ 。如果这些键被随机均匀地抛到  $A$  中，那么这就是我们期望的好动作行为。

### 10.2.4 冲突处理方案

哈希表的主要思想是使用一个哈希桶数组  $A$  和一个哈希函数  $h$ ，并用它们通过对桶  $A[h(k)]$  中存储的每个元组  $(k, v)$  进行排序实现映射。但是，当有两个不同的关键字  $k_1$  和  $k_2$  且  $h(k_1) = h(k_2)$  时，这个简单的思想就会遇到问题。由于存在这样的冲突，使得我们不能简单地将一个新的元组  $(k, v)$  直接插入桶  $A[h(k)]$  中。这个问题使我们的程序执行插入、搜索和删除等操作都变得复杂了。

#### 分离链表

处理冲突的一个简单并且有效的方式是使每个桶  $A[j]$  存储其自身的二级容器，容器存储元组  $(k, v)$ ，如  $h(k) = j$ 。正如 10.1.5 节所描述的那样，用一个很小的 list 来实现 map 实例是实现二级容器很自然的选择。这种解决冲突的方法称为分离链表（separate chaining），如图 10-6 所示。

最坏的情况下，单独的一个桶的操作时间与桶的大小成正比。假设我们使用一个比较合适的哈希函数来在容量为  $N$  的哈希桶中索引 map 中的  $n$  个元组，则桶的理想大小为  $n/N$ 。因此，如果给定一个很适合的哈希函数，核心 map 操作的时间复杂度为  $O(\lceil n/N \rceil)$ 。比值  $\lambda = n/N$  被称为哈希表的负载因子 (load factor)，这个系数应该选择一个较小常数，最好不大于 1。只要  $\lambda$  是  $O(1)$ ，则哈希表的核心操作的时间复杂度也将为  $O(1)$ 。

### 开放寻址

分离链表规则有很多很好的属性，如为映射操作提供简单的实现，但它仍然有一个小不足：需要使用一个链表作为辅助的数据结构来保存键值存在冲突的元组。如果空间是一个额外的消耗（比如，我们正在写一个用于手持设备的小程序），那么我们采用将每个元组直接存储到一个小的列表插槽中作为替代的方法。由于这种方法没有采用辅助结构，因此节省了空间，但它需要一个更为复杂的机制来处理冲突。这个方法有几个变种，统称为开放寻址模式的解决方案。开放寻址需要负载因子总是最大不超过 1，并且元组直接存储在桶数组自身的单元中。

### 线性探测及其变种

使用开放寻址处理冲突的一个简单方法是线性探测。使用这种方法时，如果我们想要将一个元组  $(k, v)$  插入桶  $A[j]$  处，在这里  $j = h(k)$ ，但  $A[j]$  已经被占用，那么，我们将尝试插入  $A[(j + 1) \bmod N]$ ；若  $A[(j + 1) \bmod N]$  也被占用，则我们尝试使用  $A[(j + 2) \bmod N]$ ，如此重复操作，直到找到一个可以接受新元组的空桶。一旦定位这个空桶，我们即可简单地将元组插入这个位置。当然这种冲突解决策略需要我们修改 `__getitem__`、`__setitem__` 或者 `__delitem__` 等所有操作的第一步的实现方式，来查找已存在的键。特别是在我们试图查找键等于  $k$  的元组时，必须从  $A[h(k)]$  开始检测连续的空间，直到找到一个键为  $k$  的元组或者发现一个空桶为止（见图 10-7）。“线性探测”之所以得名，是因为访问桶数组的单元的操作可以被视为“探测”。

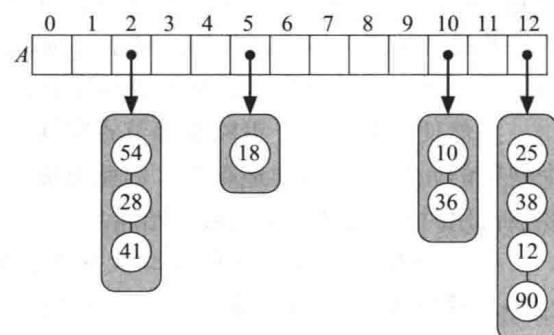


图 10-6 一个用单独列表处理冲突的大小为 13 且存储 10 个以整数为键的元组的哈希表。压缩函数是  $h(k) = k \bmod 13$ ，为简单起见，我们没有展示相关键的值

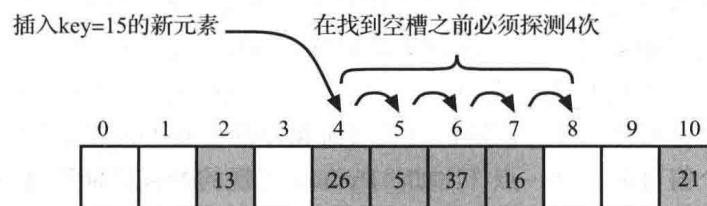


图 10-7 用线性探测的方法向哈希表中插入整数键，哈希函数是  $h(k) = k \bmod 11$ ，图中未给出键对应的值

为了实现删除操作，我们不能把找到的元组简单地从插槽中移除。比如，如图 10-7 所描述的，在插入键 15 之后，如果简单地删除键为 37 的元组，则随后的一个搜寻键为 15 的操作将会失败，因为搜寻将会从索引 4 开始，然后是索引 5，接着是索引 6，而在此处会找到一个空的单元。解决这一问题的典型方法是用一个带标记的特殊对象来替换被删除的对象。这种

解决方法会占用哈希表中的空间，同时，相应地我们也会修改查找键为  $k$  的元组的实现方法：搜索将跳过所有包含可用标记的单元，并继续探测直到找到目标元组或一个空桶（或返回到我们开始的位置）为止。此外，对于 `_setitem_` 的算法应该在搜索  $k$  的过程中记住找到的可用单元，因为在没有找到要查找的元组  $(k, v)$  时，这是一个可以插入该新元组的有效位置。

虽然使用开放寻址策略能够节省空间，但是线性探测仍然存在其他问题。它倾向于将一个映射的元组集中连续地存储，因此可能造成重叠（尤其是在哈希表中的一半以上的单元已经被占用时）。这种使用连续的哈希单元的运行方式会导致搜索速度大大降低。

另一个开放寻址策略称为二次探测，它将反复探测桶  $A[(h(k) + f(i)) \bmod N]$ ,  $i = 0, 1, 2, \dots$ , 其中  $f(i) = i^2$ , 直到发现一个空桶。与线性探测相同，二次探测策略会使删除操作更复杂，但它确实可以避免在线性探测中发生的聚集模式。而且，这种策略还创建了自己的聚集方法，称为二次聚集，即使我们假设原来的哈希码是统一的分布，其中填充的阵列单元组仍然是非统一的模式。当  $N$  是素数并且桶数组填充了不到一半时，二次探测策略保证可以找到一个空闲位置。然而，一旦哈希表元组填充了超过一半或者  $N$  不是素数时，这种策略就无法保证能找到空闲位置。我们将在练习 C-10.36 中探讨这类聚集产生的原因。

一种将不会引起如线性探测或二次探测所产生的聚集问题的开放寻址策略称为双哈希策略。在这种方法中，我们选择了一个二次哈希函数  $h'$ ，如果函数  $h$  将一些键  $k$  映射到已经被占据的桶  $A[h(k)]$  中，则我们将迭代探测桶  $A[(h(k) + f(i)) \bmod N]$ ,  $i = 0, 1, 2, \dots$ , 其中  $f(i) = i \cdot h'(k)$ 。在这种情况下，不允许将二次哈希函数设为 0。 $h'(k) = q - (k \bmod q)$  是一个常被选用的函数，其中对于素数  $q$  满足  $q < N$ ，且  $N$  也应该是素数。

另一种避免聚集的开放寻址方法是迭代地探测桶  $A[(h(k) + f(i)) \bmod N]$ ，这里  $f(i)$  是一个基于伪随机数产生器的函数，它提供一个基于原始哈希码位的可重复的但是随机的、连续的地址探测序列。Python 的字典类现在就是使用的这种方法。

### 10.2.5 负载因子、重新哈希和效率

到目前为止讨论的哈希表策略中，保证负载因子  $\lambda = n/N$  总是小于 1 是非常重要的。使用分离链表，在  $\lambda$  的值非常接近 1 时，冲突发生的概率将急剧增加，这会给我们的操作带来额外开销，因为在桶中发生冲突时，我们必须重新回到具有线性时间的基于列表的方法。实验和平均实例分析表明，使用分离链表时我们应该保持  $\lambda < 0.9$ 。

另一方面，使用开放寻址方式，随着负载因子  $\lambda$  增长到大于 0.5 并且向 1 逼近时，在桶数组中的元组集群也开始随之增长。这些集群的探测策略引起桶数组“反弹”，需要花费很多时间去遍历找到一个空的位置。在练习 C-10.36 中，我们将探讨当  $\lambda \geq 0.5$  时二次探测的降级问题。实验表明，当使用线性探测的开放寻址策略时，我们应该维持  $\lambda < 0.5$ ，而对于其他开放地址策略这个值可能会高一点（比如，Python 实现的开放寻址策略规定  $\lambda < 2/3$ ）。

如果一个哈希表的插入操作引起的负载因子超过了指定的阈值，那么调整表的大小（重新获取指定的负载因子）并且将所有的对象重新插入新表中是很常见的现象。虽然我们不需要为每个对象定义一个新的哈希码，但是我们需要基于新的哈希表大小重新设计一个压缩函数。每次重新哈希都会将元组分布到整个新桶数组中。当在一个新表上重新哈希时，新数组大小至少是之前的一倍，这是一个合理的需求。事实上，如果我们每次重新哈希时总是把表格的大小设置为原来的 2 倍，那么我们将分期承担重新哈希表格中所有元组的开销，而不是在最初插入这些元组时一次性承担（就像动态数组，见 5.3 节）。

## 哈希表的效率

虽然哈希的平均实例分析的细节超出了本书的范围，但是它的概率的基础很容易理解。如果我们的哈希函数足够好，那么所有的元组应该均匀分布在桶数组的  $N$  个单元中。那么，为了存储  $n$  个元组，在一个桶中期望的键的数量应该是  $\lceil n/N \rceil$ ，如果  $n$  是  $O(N)$ ，那么这个键的数量就是  $O(1)$ 。

由于偶尔的插入或者删除操作后要重新调整表格大小，进行周期性重新哈希所产生的相关开销可以单独计算，而这导致 `__setitem__` 和 `__getitem__` 摊销所增加的时间复杂度为  $O(1)$  的额外开销。

最坏的情况下，一个比较差的哈希函数会将所有的元组映射到同一个桶中。这将导致无论是使用分离链表还是使用任何开放式寻址策略的核心映射操作的性能是线性增长的，因为这些操作的二次序列探测仅仅与哈希码有关。在表 10-2 中汇总了这些方法的开销情况。

表 10-2 采用未排序列表或哈希表实现 map 的各个方法的运行时间对比。 $n$  用于表示 map 的元组数，并且假设桶数组所支持的哈希表的容量与 map 中的元组数成正比

操作	列 表	哈希表	
		期望值	最坏情况
<code>__getitem__</code>	$O(n)$	$O(1)$	$O(n)$
<code>__setitem__</code>	$O(n)$	$O(1)$	$O(n)$
<code>__delitem__</code>	$O(n)$	$O(1)$	$O(n)$
<code>__len__</code>	$O(1)$	$O(1)$	$O(1)$
<code>__iter__</code>	$O(n)$	$O(n)$	$O(n)$

在实践中，哈希表是实现 map 的最有效的方式之一，程序员相信这样使得映射的核心操作的运行时间是常量。Python 的 dict 类使用哈希方式实现，并且 Python 解释器依赖词典来检索获取给定的命名空间内由标识符引用的对象（见 1.10 节和 2.5 节）。基本的命令  $c = a + b$  在本地词典的命名空间中两次调用 `__getitem__` 来检索标识符  $a$  和  $b$  的值，并且调用一次 `__setitem__` 来在该命名空间中存储与  $c$  相关联的结果。在我们自己的算法分析中，简单地假设这样的字典操作的运行时间是常量，独立于命名空间中条目的数量。（诚然，在一个典型的命名空间中的条目数量基本上是一个有界的常数。）

在 2003 年的一篇学术论文中<sup>[31]</sup>，研究者讨论利用最坏情况下的哈希表而导致遭受互联网技术的服务拒绝（DoS）攻击的可能性。文中提到，对于许多已发表的哈希码的计算方法，攻击者可以预先计算大量的中等长度的字符串，并且将所有的字符串哈希到相同的 32 位哈希码上。（回想一下我们所描述的所有哈希方案，除了双哈希，如果两个关键字被映射到相同的哈希码，它们在冲突解决方案中是不可分离的。）

在 2011 年下半年，另一个研究团队给出了这类攻击的一个实现<sup>[61]</sup>。Web 服务中允许使用形如 `?key1 = val1&key2 = val2&key3 = val3` 的语法将一串 key-value 参数嵌入 URL 中。一般，这些 key-value 对会立即由服务器存储在一个 map 中，并假设在 map 中存储时间与条目的数量呈线性关系，并对这些参数的长度和数量加以限制。如果所有的键都发生冲突，则存储需要平方级的时间（因为服务器需要进行大量的处理工作）。在 2012 年春天，Python 开发者发布了一个安全补丁，该补丁将随机机制引入到字符串哈希码的计算中，这使得翻转工程师的一组冲突字符串更难处理。

### 10.2.6 Python 哈希表的实现

在这部分，我们介绍两种哈希表的实现，一种使用分离链表，而另一种使用包含线性探测的开放寻址。虽然这些解决冲突的方法差异很大，但是也有很多共性。由于这个原因我们通过扩展 MapBase 类（基于代码段 10-2）来定义一个新的 HashMapBase 类（见代码段 10-4），它为我们的两种哈希实现提供了大量的通用功能。HashMapBase 类主要的设计元素是：

- 桶数组由一个 Python 列表表示，名为 self.\_table，并且所有的条目初始为 None。
- 我们维护一个 self.\_n 的实例变量用来表示当前存储在哈希表中不同元组的个数。
- 如果表格的负载因子增加到超过 0.5，我们会将哈希表的大小扩大 2 倍并且将所有元组重新哈希到新的表中。
- 我们定义一个 \_hash\_ 函数的实用方法，该方法依靠 Python 内置哈希函数来生成键的哈希码，并用随机乘 - 加 - 切分 (MAD) 公式生成压缩函数。

代码段 10-4 一个哈希表实现的基类，基于代码段 10-2 中的 MapBase 类扩展实现的

```

1 class HashMapBase(MapBase):
2     """Abstract base class for map using hash-table with MAD compression."""
3
4     def __init__(self, cap=11, p=109345121):
5         """Create an empty hash-table map."""
6         self._table = cap * [None]
7         self._n = 0                               # number of entries in the map
8         self._prime = p                           # prime for MAD compression
9         self._scale = 1 + randrange(p-1)          # scale from 1 to p-1 for MAD
10        self._shift = randrange(p)               # shift from 0 to p-1 for MAD
11
12    def _hash_function(self, k):
13        return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)
14
15    def __len__(self):
16        return self._n
17
18    def __getitem__(self, k):
19        j = self._hash_function(k)
20        return self._bucket_getitem(j, k)          # may raise KeyError
21
22    def __setitem__(self, k, v):
23        j = self._hash_function(k)
24        self._bucket_setitem(j, k, v)             # subroutine maintains self._n
25        if self._n > len(self._table) // 2:       # keep load factor <= 0.5
26            self._resize(2 * len(self._table) - 1)  # number 2^x - 1 is often prime
27
28    def __delitem__(self, k):
29        j = self._hash_function(k)
30        self._bucket_delitem(j, k)                # may raise KeyError
31        self._n -= 1
32
33    def _resize(self, c):                      # resize bucket array to capacity c
34        old = list(self.items())               # use iteration to record existing items
35        self._table = c * [None]              # then reset table to desired capacity
36        self._n = 0                            # n recomputed during subsequent adds
37        for (k,v) in old:                   # reinsert old key-value pair
38            self[k] = v

```

在基类中，如何表示一个“桶”的任何概念都没有实现。通过使用单链表，每个桶将是一个独立的结构。然而，使用开放寻址策略时，每个桶都没有一个有形的容器，且探测序列

使桶得到有效的交叉存取。

在我们的设计中，HashMapBase 类假定有以下抽象方法，且每一个方法必须在具体子类中实现。

- `_bucket_getitem(j, k)`。这个方法在桶 j 中搜索查找键为 k 的元组，如果找到了则返回对应的值，如果找不到则抛出 `KeyError`。
- `_bucket_setitem(j, k, v)`。这个方法将桶 j 中键 k 的值修改为 v。如键 k 的值已经存在，则新的值覆盖已经存在的值。否则，将这个新元组插入桶中，并且这个方法负责增加 `self._n` 的值。
- `_bucket_delitem(j, k)`。这个方法删除桶 j 中键为 k 的元组，如果这样的元组不存在则抛出 `KeyError` 异常（在这个方法之后 `self._n` 的值会减小）。
- `__iter__`。这是遍历 map 所有键的标准 `map` 方法。在每个桶的基础上我们的基类不代表这个方法，因为在开放寻址中桶并不是固有不相交的。

### 分离链表

代码段 10-5 给出了以 ChainHashMap 类的形式实现含有分离链表的哈希表。它采用代码段 10-3 中 `UnsortedTableMap` 类的一个实例来表示单个的桶。

类中的前三个方法使用索引 `j` 来访问在桶数组中的潜在桶，并检测表中的元组为空的特殊情况。只有当 `_bucket_setitem` 被其他的空位置调用时，我们才需要一个新的桶。剩余的依赖于 `map` 行为的功能已经由单个的 `UnsortedTableMap` 实例所支持。我们需要提前一点决定是否在链上的 `__setitem__` 的应用会引起 `map` 大小的净增加（即是否给定的键是新的）。

代码段 10-5 用分离链表实现的具体哈希 map 类

```

1 class ChainHashMap(HashMapBase):
2     """ Hash map implemented with separate chaining for collision resolution. """
3
4     def _bucket_getitem(self, j, k):
5         bucket = self._table[j]
6         if bucket is None:
7             raise KeyError('Key Error: ' + repr(k))      # no match found
8         return bucket[k]                                # may raise KeyError
9
10    def _bucket_setitem(self, j, k, v):
11        if self._table[j] is None:
12            self._table[j] = UnsortedTableMap()          # bucket is new to the table
13            oldsize = len(self._table[j])
14            self._table[j][k] = v
15            if len(self._table[j]) > oldsize:           # key was new to the table
16                self._n += 1                             # increase overall map size
17
18    def _bucket_delitem(self, j, k):
19        bucket = self._table[j]
20        if bucket is None:
21            raise KeyError('Key Error: ' + repr(k))      # no match found
22            del bucket[k]                            # may raise KeyError
23
24    def __iter__(self):
25        for bucket in self._table:
26            if bucket is not None:                      # a nonempty slot
27                for key in bucket:
28                    yield key

```

## 线性探测

我们使用含线性探测的开放寻址实现 ProbeHashMap 类，并且在代码段 10-6 和 10-7 中给出详细描述。为了支持删除操作，我们使用了 10.2.2 节介绍的技术，该技术是在已被删除的表的位置上做一个特殊的标记，以此来将它和一个总为空的位置区分开来。在实现中，我们声明了一个类级的属性 \_AVAIL 作为哨兵。(因为我们不关心任何哨兵类的行为，仅仅是用来与其他对象相区分，所以我们使用一个内置的对象类的实例。)

**代码段 10-6 用线性探测处理冲突的 ProbeHashMap 类的具体实现（在代码段 10-7 中继续）**

---

```

1 class ProbeHashMap(HashMapBase):
2     """Hash map implemented with linear probing for collision resolution."""
3     _AVAIL = object()      # sentinel marks locations of previous deletions
4
5     def _is_available(self, j):
6         """Return True if index j is available in table."""
7         return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL
8
9     def _find_slot(self, j, k):
10        """Search for key k in bucket at index j.
11
12        Return (success, index) tuple, described as follows:
13        If match was found, success is True and index denotes its location.
14        If no match found, success is False and index denotes first available slot.
15        """
16        firstAvail = None
17        while True:
18            if self._is_available(j):
19                if firstAvail is None:
20                    firstAvail = j                      # mark this as first avail
21                if self._table[j] is None:
22                    return (False, firstAvail)          # search has failed
23                elif k == self._table[j]._key:
24                    return (True, j)                  # found a match
25                j = (j + 1) % len(self._table)       # keep looking (cyclically)

```

---

开放寻址最具挑战性的一面是在插入或搜寻一个元组的过程中发生冲突时，合理地跟踪探测序列。为此，我们定义一个非公共的实用工具 \_find\_slot，用于在桶  $j$  中搜寻含有键  $k$  的元组（即这里的  $j$  是哈希函数对键  $k$  返回的索引）。

**代码段 10-7 线性探测处理冲突的 ProbeHashMap 类的具体实现（前接代码段 10-6 中的代码）**

---

```

26     def _bucket_getitem(self, j, k):
27         found, s = self._find_slot(j, k)
28         if not found:
29             raise KeyError('Key Error: ' + repr(k))           # no match found
30         return self._table[s]._value
31
32     def _bucket_setitem(self, j, k, v):
33         found, s = self._find_slot(j, k)
34         if not found:
35             self._table[s] = self._Item(k, v)                 # insert new item
36             self._n += 1                                     # size has increased
37         else:
38             self._table[s]._value = v                      # overwrite existing
39
40     def _bucket_delitem(self, j, k):
41         found, s = self._find_slot(j, k)
42         if not found:

```

---

```

43     raise KeyError('Key Error: ' + repr(k))           # no match found
44     self._table[s] = ProbeHashMap._AVAIL               # mark as vacated
45
46     def __iter__(self):
47         for j in range(len(self._table)):                 # scan entire table
48             if not self._is_available(j):
49                 yield self._table[j]._key

```

有三个主要的 map 操作都依赖于 `_find_slot` 程序实现。当试图检索给定对应的元组时，我们必须继续探测直到找到该键，或者找到表中的一个值为插槽。我们要一直搜索直到找到一个 `_AVAIL` 哨兵为止，因为它代表插入目标元组时被填充的位置。

当要将一个 key-value 对插入 map 时，我们必须先尝试找到一个键为给定值的元组，这样我们就可以用新的元组覆盖这个查找到的元组，而不是向 map 中插入一个新的元组。因此，必须在插入前搜索所有的 `_AVAIL` 哨兵出现的情况。但是，如果没有找到匹配的项，我们更倾向于将第一个插槽位置标记为 `_AVAIL`，如果找到了，我们便把新的元组存入表里。`_find_slot` 方法制定了这个逻辑：持续搜索直到找到一个真正的空插槽，返回第一个可用的插槽的索引用于插入操作。

当使用 `_bucket_delitem` 删除一个元组时，为了与我们的策略保持一致，专门将表的条目设为 `_AVAIL` 哨兵标识。

### 10.3 有序映射

传统的映射 ADT 允许用户查找与给定键关联的值，这种键的查找被称为精确查找。

例如，计算机系统经常维护已发生事件的信息（如金融交易），我们依据时间戳来组织这些的事件。如果我们可以假定时间戳对于一个特定的系统是唯一的，那么我们就可以以时间戳为键组织一个映射，并将发生在那个时间的事件作为值。一个特定的时间戳可以作为事件的引用标识，这样就可以快速地从映射中检索出该事件的信息。然而，映射 ADT 不提供任何方式来获得一个按时间排序的所有已发生事件的列表，或去查找最接近一个特定的时间所发生的事件。事实上，映射 ADT 基于哈希算法实现高性能，依赖于键的故意分散，这使得键在原有的域中彼此似乎离得很“近”，从而使它们在哈希表中的分布更均匀。

在这一部分，我们介绍一个称为有序映射的映射 ADT 的扩展，它包括标准映射的所有行为，还增加了以下行为：

- M.`find_min()`: 用最小键返回 (键, 值) 对 (或 None, 如果映射为空)。
- M.`find_max()`: 用最大键返回 (键, 值) 对 (或 None, 如果映射为空)。
- M.`find_lt(k)`: 用严格小于 k 的最大键返回 (键, 值) 对 (或 None, 如果没有这样的项存在)。
- M.`find_le(k)`: 用严格小于等于 k 的最大键返回 (键, 值) 对 (或 None, 如果没有这样的项存在)。
- M.`find_gt(k)`: 用严格大于 k 的最小的键返回 (键, 值) 对 (或 None, 如果没有这样的项存在)。
- M.`find_ge(k)`: 用严格大于或等于 k 的最小的键返回 (键, 值) 对 (或 None, 如果没有这样的项存在)。

- `M.find-range(start, stop)`：用  $\text{start} \leq \text{键} < \text{stop}$  迭代遍历所有（键，值）。如果 `start` 指定为 `None`，从最小的键开始迭代；如果 `stop` 指定为 `None`，到最大键迭代结束。
- `iter(M)`：根据自然顺序从最小到最大迭代遍历映射中的所有键。
- `reversed(M)`：根据逆序迭代映射中的所有键 `r`，这在 Python 中是用 `__reversed__` 来实现的。

### 10.3.1 排序检索表

一些数据结构能有效地支持排序映射 ADT，我们将在 10.4 节和第 11 章中讨论一些先进的技术。在本节中，首先，我们从探索一个简单有序映射的实现开始。我们将映射的元组存储在一个基于数组的序列  $A$  中，以键的升序排列，假定键是天然定义的顺序（见图 10-8）。我们将这个映射实现为排序检索表（sorted search table）。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

图 10-8 一个通过排序检索表实现的映射。图中我们仅展示了映射的键，以凸显它们的顺序

对于 10.1.5 节中以未排序表实现映射的例子，若根据映射中的元组数量按比例增减数组的大小，其空间的需求量是  $O(n)$ 。我们坚持  $A$  基于数组存储元组以及这种表示的最主要优势是，它支持用二分查找算法来做各种有效的操作。

#### 二分查找和不精确查找

我们在 4.1.3 节中介绍了二分查找算法，能检测一个给定的目标是否存储在已排序的序列中。在原来的介绍中（代码段 4-3），`binary search` 函数返回 `True` 或 `False` 来检测指定目标是否被发现。由于这样的方法可以用来实现映射 ADT 的 `__contains__` 方法，我们可以在实现以各种形式的不精确查找支持有序映射 ADT 时，应用二分查找算法以提供更多有用信息。

当实施二分查找时，一个重要的实现是我们可以决定要查找的目标或是临近目标的项目的索引。在查找成功时，一个标准实现会决定所找到目标的精确索引。在一次失败的查找中，即使目标没有被发现，算法也会有确定一组索引有效地指定集合中的元素是小于还是大于未找到的目标。

作为引入的例子，我们原来在图 4-5 中的模拟，展示了一个成功查找目标 22 的二分查找，在图 10-8 中又用相同的数据进行描述。如果我们要查找 21，算法的前四个步骤和原来是相同的，后继的差异是我们将调用倒置参数，即 `high = 9, low = 10`，这将有效得出未找到的目标值位于值 19 和 22 之间。

#### 实现

在代码段 10-8 ~ 10-10 中，我们提出一个支持排序表映射 ADT 的 `SortedTableMap` 类的完整实现方法。该设计中最值得注意的特性是含有 `_find_index` 这个功能函数。这个方法使用二分查找算法，但是按照惯例，返回搜索区间中键大于等于  $k$  的最左侧元组的索引。然而，如果是当前的键，它将返回键为该值的元组的索引。（想一下，键在一个映射中是唯一的。）如果键找不到，函数返回搜索区间各元组的索引，这个区间位于未能找到的键所在位置附近。技术上，该方法返回索引 +1 表示区间中没有元组的键大于  $k$ 。

## 代码段 10-8 SortedTableMap 类的实现（代码段 10-9 和 10-10 为后续代码）

```

1 class SortedTableMap(MapBase):
2     """Map implementation using a sorted table."""
3
4     #----- nonpublic behaviors -----
5     def _find_index(self, k, low, high):
6         """Return index of the leftmost item with key greater than or equal to k.
7
8         Return high + 1 if no such item qualifies.
9
10        That is, j will be returned such that:
11            all items of slice table[low:j] have key < k
12            all items of slice table[j:high+1] have key >= k
13
14        if high < low:
15            return high + 1                         # no element qualifies
16        else:
17            mid = (low + high) // 2
18            if k == self._table[mid]._key:
19                return mid                         # found exact match
20            elif k < self._table[mid]._key:
21                return self._find_index(k, low, mid - 1)  # Note: may return mid
22            else:
23                return self._find_index(k, mid + 1, high) # answer is right of mid
24
25     #----- public behaviors -----
26     def __init__(self):
27         """Create an empty map."""
28         self._table = []
29
30     def __len__(self):
31         """Return number of items in the map."""
32         return len(self._table)
33
34     def __getitem__(self, k):
35         """Return value associated with key k (raise KeyError if not found)."""
36         j = self._find_index(k, 0, len(self._table) - 1)
37         if j == len(self._table) or self._table[j]._key != k:
38             raise KeyError('Key Error: ' + repr(k))
39         return self._table[j]._value

```

在实现传统的映射操作和新的有序映射操作时，我们依赖这个实用方法。方法 `__getitem__`、`__setitem__` 和 `__delitem__` 中的每一个函数体都从调用 `_find_index` 函数开始，以决定候选索引来匹配要找的键。对于 `__getitem__` 方法，我们简单地检查是否包含确认目标存在的索引。而对 `__setitem__` 方法，我们的目标是如果找到一个键为 `k` 的元组，就替换这个已有元组的值，否则需要在映射中插入一个新的条目。如果 `_find_index` 返回的索引存在匹配的索引，就返回该索引，否则返回将插入的新元组的位置的索引。对于 `__delitem__`，如果找到目标索引，则我们将利用 `_find_index` 的方法，决定要返回的元组的位置。

`_find_index` 方法的作用与代码段 10-10 中给出的非精确查找的实现方法是同样有价值的。对于 `find_lt`、`find_le`、`find_gt` 和 `find_ge` 方法的实现，都是从调用 `_find_index` 开始，如果存在大于等于 `k` 的键，则将索引定位在第一个大于等于 `k` 的键的索引位置。如果这样的操作有效，正是我们想要 `find_ge` 实现的，且刚好是超过 `find_lt` 的索引。对于 `find_gt` 和 `find_le`，需要一些额外的案例分析来辨别指定的索引是否有等于 `k` 的键。例如，如果指定的元组有一个匹配的键，`find_gt` 的实现中，要在继续该过程前对索引做增量操作。（为了简洁

起见，我们省略了 `find_le` 的实现。) 在所有案例中，我们必须妥善处理边界情况，如果无法找到一个与所需的属性相匹配的键，则报告 `None`。

我们实现 `find_range` 的策略是使用 `_find_index` 函数来定位第一个键  $\geq start$  的元组（假设 `start` 是非 `None` 的值）。据此，我们使用 `while` 循环按顺序逐个报告表中的元组直到达到一个键大于或等于 `stop` 值的元组（或者直到到达表的末尾元组）。值得注意的是，如果第一个键大于等于 `start` 值，或者它正好也大于等于 `stop` 值，则 `while` 循环将迭代零次，这表示映射中的一个空范围（即没有元组包含在指定的范围）。

代码段 10-9 SortedTableMap 类的实现（与代码段 10-8 和 10.10 共同组成该实现）

---

```

40 def __setitem__(self, k, v):
41     """Assign value v to key k, overwriting existing value if present."""
42     j = self._find_index(k, 0, len(self._table) - 1)
43     if j < len(self._table) and self._table[j]._key == k:
44         self._table[j]._value = v                      # reassign value
45     else:
46         self._table.insert(j, self._Item(k,v))        # adds new item
47
48 def __delitem__(self, k):
49     """Remove item associated with key k (raise KeyError if not found)."""
50     j = self._find_index(k, 0, len(self._table) - 1)
51     if j == len(self._table) or self._table[j]._key != k:
52         raise KeyError('Key Error: ' + repr(k))
53     self._table.pop(j)                            # delete item
54
55 def __iter__(self):
56     """Generate keys of the map ordered from minimum to maximum."""
57     for item in self._table:
58         yield item._key
59
60 def __reversed__(self):
61     """Generate keys of the map ordered from maximum to minimum."""
62     for item in reversed(self._table):
63         yield item._key
64
65 def find_min(self):
66     """Return (key,value) pair with minimum key (or None if empty)."""
67     if len(self._table) > 0:
68         return (self._table[0]._key, self._table[0]._value)
69     else:
70         return None
71
72 def find_max(self):
73     """Return (key,value) pair with maximum key (or None if empty)."""
74     if len(self._table) > 0:
75         return (self._table[-1]._key, self._table[-1]._value)
76     else:
77         return None

```

---

代码段 10-10 SortedTableMap 类的实现（接续代码段 10-8 和 10-9）。由于篇幅限制，我们省略了 `find-le` 方法

---

```

78 def find_ge(self, k):
79     """Return (key,value) pair with least key greater than or equal to k."""
80     j = self._find_index(k, 0, len(self._table) - 1)      # j's key >= k
81     if j < len(self._table):
82         return (self._table[j]._key, self._table[j]._value)
83     else:

```

```

84     return None
85
86     def find_lt(self, k):
87         """Return (key,value) pair with greatest key strictly less than k."""
88         j = self._find_index(k, 0, len(self._table) - 1)           # j's key >= k
89         if j > 0:
90             return (self._table[j-1].key, self._table[j-1].value) # Note use of j-1
91         else:
92             return None
93
94     def find_gt(self, k):
95         """Return (key,value) pair with least key strictly greater than k."""
96         j = self._find_index(k, 0, len(self._table) - 1)           # j's key >= k
97         if j < len(self._table) and self._table[j].key == k:
98             j += 1                                              # advanced past match
99         if j < len(self._table):
100            return (self._table[j].key, self._table[j].value)
101        else:
102            return None
103
104    def find_range(self, start, stop):
105        """Iterate all (key,value) pairs such that start <= key < stop.
106
107        If start is None, iteration begins with minimum key of map.
108        If stop is None, iteration continues through the maximum key of map.
109        """
110        if start is None:
111            j = 0
112        else:
113            j = self._find_index(start, 0, len(self._table)-1)      # find first result
114        while j < len(self._table) and (stop is None or self._table[j].key < stop):
115            yield (self._table[j].key, self._table[j].value)
116            j += 1

```

## 分析

我们通过分析 SortedTableMap 实现的性能得出结果。有序映射 ADT (包括传统映射操作) 所有方法的运行时间如表 10-3 所示。可以清楚地看到 `__len__`, `find_min` 和 `find_max` 方法的运行时间为  $O(1)$ , 而且对代表中的键执行任何方向的迭代都可以在  $O(n)$  时间内完成。

表 10-3 SortedTableMap 实现的有序映射的性能。我们用  $n$  来表示映射中在操作执行时元组的数量。空间需求为  $O(n)$

操作	运行时间
<code>len(M)</code>	$O(1)$
<code>k in M</code>	$O(\log n)$
<code>M[k] = v</code>	最坏情况下为 $O(n)$ , 如果存在 <code>k</code> 则为 $O(\log n)$
<code>del M[k]</code>	最坏情况下为 $O(n)$
<code>M.find_min()</code> , <code>M.find_max()</code>	$O(1)$
<code>M.find_lt(k)</code> , <code>M.find_gt(k)</code> <code>M.find_le(k)</code> , <code>M.find_ge(k)</code>	$O(\log n)$
<code>M.find_range(start, stop)</code>	$O(s + \log n)$ , 报告 $s$ 项
<code>iter(M)</code> , <code>reversed(M)</code>	$O(n)$

分析可知, 各种形式的查找都取决于  $n$  个条目的表上运行时间为  $O(\log n)$  时间的二分查找。这种说法首次出现在 4.2 节中的命题 4-2 中, 且这一分析结果显然也适用于我们的

`_find_index` 方法。因此，我们断言对于 `__getitem__`、`find_lt`、`find_gt`、`find_le` 和 `find_ge`，最坏情况下的运行时间是  $O(\log n)$ 。因为这几个方法在基于索引执行一些常数数量的步骤后，都会调用一次方法 `_find_index` 来获取合适的结果。`find-range` 的分析结果更加有趣，它先在指定范围（如果有的话）内用二分查找找到第一个符合条件的元组，之后，执行循环依次报告后续元组的值，每次循环的时间花销为  $O(1)$ ，直至执行到指定范围的末尾。如果在循环范围内报告了  $s$  项元组，则该方法总的运行时间为  $O(s + \log n)$ 。

与高效的查找操作形成对比，排序表的更新操作要花费相当多的时间。尽管用二分查找可以辨别出插入和删除等更新操作发生在哪一个索引中，在最坏的情况下，为了维持表中元组的顺序，表中许多元素都要调整位置。特别地，潜在地调用 `__setitem__` 中的 `_table.insert` 和 `__delitem__` 中的 `_table.pop` 在最坏情况下的时间复杂度是  $O(n)$ 。（参考 5.4.1 节有关链表类中的相应操作的讨论。）

由此可见，排序映射主要是用于预计含有查找较多但更新相对较少的情况。

### 10.3.2 有序映射的两种应用

在这一部分，我们将探讨使用排序映射而不是传统的映射时特别有优势的应用。要运用一个有序映射，键必须来自一个完全有序的域。此外，为了合理利用不精确查找和排序映射提供的范围查找的优势，则查找中相互邻近的键之间有关联也是有原因的（或者说有迹可循的）。

#### 航班数据库

互联网上有些网站允许用户特别是有意向买票的用户查询航班数据库来查找不同城市之间的航班。此时，用户会指定出发地和目的地城市，以及一个确切的出发日期和时间。为了支持这样的查询，我们可以将航班数据模拟为一个映射，其中键为 `Flight` 对象，它所包含的域（field）对应四个参数。也就是说，键是一个元组。

$$k = (\text{origin}, \text{destination}, \text{date}, \text{time})$$

关于航班的附加信息，航班号和座位的数量分别在 `first (F)` 类和 `coach (Y)` 类中提供，飞行时间和费用可以存储在值对象中。

找到一个目标航班与给定的查询条件匹配并不简单。尽管用户通常匹配出发和目的城市，然而出发日期可以有一定的灵活性，而且在具体的某一天里出发的时间也可以有灵活性。我们可以按词典式排序的键来查询。那么，实现一个有效的有序映射，将是满足用户的查询需求的好方式。例如，给定一个查询的键  $k$ ，我们将调用 `find-ge(k)` 来返回符合查询的城市区间要求，并且匹配出发日期和时间或是晚于指定时间的第一个航班。更好的方法是，利用组织合理的键，我们使用函数 `find-range(k1, k2)` 来找到所有符合给定时间范围的航班。例如，如果  $k1 = (\text{ORD}, \text{PWD}, \text{05May}, 09:30)$ ,  $k2 = (\text{ORD}, \text{PWD}, \text{05May}, 20:00)$ ，相应的调用 `find-range(k1, k2)` 将获得以下的键值对序列：

(ORD, PWD, 05May, 09:53) :	(AA 1840, F5, Y15, 02:05, \$251),
(ORD, PWD, 05May, 13:29) :	(AA 600, F2, Y0, 02:16, \$713),
(ORD, PWD, 05May, 17:39) :	(AA 416, F3, Y9, 02:09, \$365),
(ORD, PWD, 05May, 19:50) :	(AA 1828, F9, Y25, 02:13, \$186)

#### 最大值集

生活中充满了权衡。我们经常需要权衡所需的性能与价格。举个例子，假设我们对于维护一个对汽车的最大速度和价格排序的数据库感兴趣。我们会允许具有一定资金量的用户在数据中查询，以便找到他可以买得起的最快的汽车。

我们可以建立这样一个模型，通过使用一个键值对来模拟权衡时所使用的两个参数，由此，在这个例子中，价格 - 速度对即是这样的两个参数。需要注意的是，在使用这种度量方法时，有些汽车是严格好于其他汽车的，如一个价格 - 速度对为 (20 000, 100) 的汽车严格好于价格 - 速度对为 (30 000, 90) 的汽车。与此同时，价格 - 速度对为 (20 000, 100) 的汽车可能会好于或差于价格 - 速度对 (30 000, 120) 为的汽车，这将取决于我们需要花多少钱（如图 10-9 所示）。

形式上，如果  $a \leq c$  且  $b \geq d$ ，我们说价格 - 性能对  $(a, b)$  管辖着  $(c, d)$ ，其中  $(c, d) \neq (a, b)$ ，即第一个价格 - 性能对较第二个价格 - 性能对具有较少的花费和至少一样好的性能。如果  $(a, b)$  不被其他价格 - 性能对所管辖的话，则称其为一个最大值对。我们更热衷于在价格 - 性能对的集合中维护的严格最大值对的集合。也就是说，我们将往集合中加入新的对（例如有新车生产发布时），并且会根据一个给出的美元价格  $d$  来在这个集合上进行查询，找出那些价格不超过  $d$  美元的最快的车。

### 在有序映射中维护最大值集

我们可以在有序映射中存储最大值集  $M$ ，这样，价格即为键 (key) 域，性能 (速度) 就是值 (value) 域。然后，我们可以实现  $\text{add}(c, p)$  操作，这个操作用于加入一个新的价格 - 性能对  $(c, p)$ ，并且实现  $\text{best}(c)$  操作，用于返回价格至多为  $c$  的最好的价格 - 性能对，如代码段 10-11 所示。

代码段 10-11 一个使用有序映射维持最大值集的类的实现

```

1 class CostPerformanceDatabase:
2     """ Maintain a database of maximal (cost,performance) pairs."""
3
4     def __init__(self):
5         """ Create an empty database."""
6         self._M = SortedTableMap()           # or a more efficient sorted map
7
8     def best(self, c):
9         """ Return (cost,performance) pair with largest cost not exceeding c.
10
11        Return None if there is no such pair.
12
13        return self._M.find_le(c)
14
15    def add(self, c, p):
16        """ Add new entry with cost c and performance p."""
17        # determine if (c,p) is dominated by an existing pair
18        other = self._M.find_le(c)          # other is at least as cheap as c
19        if other is not None and other[1] >= p: # if its performance is as good,
20            return                           # (c,p) is dominated, so ignore
21        self._M[c] = p                      # else, add (c,p) to database

```

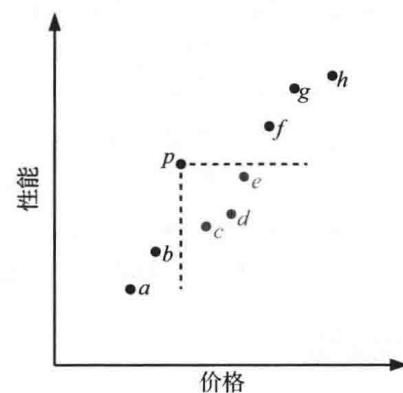


图 10-9 用平面上的点代表价格 - 性能对的权衡。值得注意的是点  $p$  严格好于点  $c$ 、 $d$  和  $e$ ，但是可能好于或等于点  $a$ 、 $b$ 、 $f$ 、 $g$  和  $h$ ，这决取决于我们想要花多少钱。因此，如果我们想要在点集中加入  $p$ ，可以移除点  $c$ 、 $d$  和  $e$ ，但是不要移除其他的点

```

22      # and now remove any pairs that are dominated by (c,p)
23      other = self._M.find_gt(c)           # other more expensive than c
24      while other is not None and other[1] <= p:
25          del self._M[other[0]]
26          other = self._M.find_gt(c)

```

不幸的是，如果我们使用 SortedTableMap 来执行  $M$ , add 操作运行时间为最坏情况下的  $O(n)$ 。另一方面，如果我们使用一个跳跃表（接下来介绍这个表）来实现  $M$ ，则可以在一个确定的  $O(\log n)$  时间里执行 best( $c$ ) 查询并在确定的  $O((1+r)\log n)$  时间里执行 add( $c, p$ ) 更新操作，其中  $r$  是从表中移除的点的数量。

## 10.4 跳跃表

一种实现排序映射 ADT 的有趣的数据结构是跳跃表。在 10.3.1 节中，一个排序数组允许通过二分搜索以  $O(\log n)$  时间做查询。不幸的是，由于需要调整元素位置，排序数组更新操作的最坏情况下的运行时间需要  $O(n)$ 。在第 7 章，我们讲过只要列表中的位置是明确的，用链表可以非常有效地支持更新操作。不幸的是，我们不能在一个标准链表中执行快速查找。举例来说，二分查找算法需要一个有效的手段来通过索引直接访问一个元素的序列。

跳跃表提供一个聪明的折衷方式以有效地支持查找和更新操作。一个映射  $M$  的跳跃表  $S$  包含一列表序列  $\{S_0, S_1, \dots, S_h\}$ 。每一个列表  $S_i$  依照键的升序存储着  $M$  的一个元组子集，用两个标注为  $-\infty$  和  $+\infty$  的哨兵键追加元组，其中  $-\infty$  比每一个可能的插入  $M$  的键都小， $+\infty$  比每一个可能插入  $M$  的键都大。此外，列表  $S$  还要满足下面的条件：

- 列表  $S_0$  包含映射  $M$  中的每一项（包含  $-\infty$  和  $+\infty$ ）。
- 对于  $i=1, \dots, h-1$ ，列表  $S_i$  包含一个列表  $S_{i-1}$  随机生成的元组的子集（还有  $-\infty$  和  $+\infty$ ）。
- 列表  $S_h$  仅包含  $-\infty$  和  $+\infty$ 。

一个跳跃表如图 10-10 所示。这是一个常用的可视化表示，在列表  $S$  中，列表  $S_0$  在最底部，在  $S_0$  之上有列表  $S_1, \dots, S_h$ 。并且，我们称  $h$  为列表  $S$  的高度。

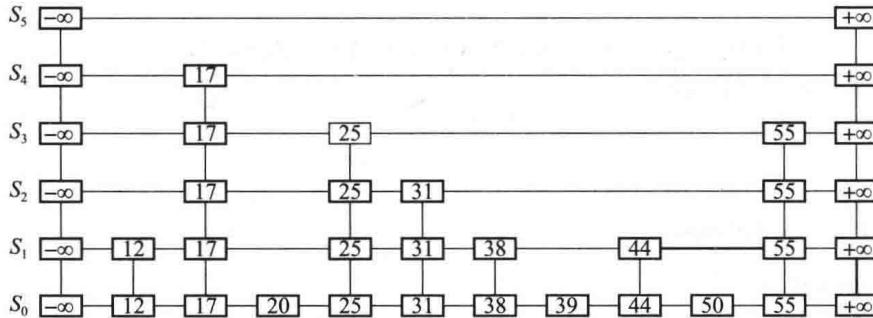


图 10-10 存储有 10 个元组的跳跃表。为简单起见，我们仅展示每个元组的键而不包含其相关的值

直观地，列表  $S_{i+1}$  建立时包含更多或更少的  $S_i$  中的备选元组。当我们在观察插入方法的细节时会发现， $S_{i+1}$  中的元组是从  $S_i$  中随机挑选出来的，从  $S_i$  挑选到  $S_{i+1}$  中的概率也为  $1/2$ ，大体上， $S_i$  中的每一项都是通过“抛硬币”的方式挑选出来的，如果正面朝上则将该项置于  $S_{i+1}$  中。因此，我们希望  $S_1$  含有  $n/2$  个元组， $S_2$  有  $n/4$  个元组，一般地说就是  $S_i$  含有  $n/2^i$  个元组。换言之，我们希望  $S$  的高度为  $\log n$ 。从一个列表到下一个列表，对含有的元组数做折半处理并不是强制地作为跳跃表的一个明确的特性。取而代之的是采用随机化的方法。

生成数字的功能可以视为大多数现代计算机内置的随机数字，因为它们被广泛使用在电脑游戏、密码学和计算机仿真中。一些叫作伪随机数生成器的功能，从一颗初始种子开始生成类似随机的数（参考 1.11 节中有关随机模块的讨论）。其他方法使用硬件设备来从自然中提取“真”随机数。无论如何，我们假设从电脑中访问的数对我们的分析而言都是完全随机的数。

在数据结构和算法设计中使用随机选择主要的优势在于它的结构和函数的结果会变得简单和高效。跳跃表的查找时间和二分查找一样是限定在对数级的范围，在插入和删除元组时，它也扩展了更新算法的性能。然而，当二分查找的性能在对于一个排序表有最坏情况下的范围时，跳跃表也有一个预期的范围。

跳跃表在组织它的结构时，通过平均时间为  $O(\log n)$  的查找和更新方法做随机选择，其中  $n$  是映射中的元组项目数。有趣的是，这里使用的平均时间复杂度的概念不是由输入的键的分布概率决定的。取而代之的是，它取决于在实现用于帮助决定在哪安插新条目的插入函数中所使用的随机数生成器。运行时间是用于插入条目的所有可能的随机数输出的平均值。

利用列表和树使用的位置抽象，我们视跳跃表为一个水平组织成层（level）、垂直组织成塔（tower）的二维位置集合。每个水平层是一个列表  $S_i$ ，每个垂直塔包含了存储着相同元组的位置，这些元组跨越连续的列表。可以使用以下操作遍历跳跃表中的每个位置：

- `next(p)`: 返回在同一水平层位置上紧接着  $p$  的位置。
- `prev(p)`: 返回在同一水平层位置上在  $p$  之前的位置。
- `below(p)`: 返回在同一垂直塔位置上在  $p$  下面的位置。
- `above(p)`: 返回在同一垂直塔位置上在  $p$  上面的位置。

我们通常假设对于上述操作，如果要求的位置是不存在的，则返回 `None`。不必考虑细节，我们注意到可以通过链结构简单地实现一个跳跃表，给定一个跳跃表  $p$  的位置，每一个单独的遍历方法需要  $O(1)$  时间。这样的链结构本质上是在垂直塔方向上对齐的双链表集合  $h$ ，这样的链表本身也是双链表。

#### 10.4.1 跳跃表中的查找和更新操作

跳跃表结构提供简单的映射查找和更新算法。事实上，所有的跳跃表查找和更新算法都依赖于一个简洁的 `SkipSearch` 方法，其需要一个键  $k$  并发现  $S$  列表中具有小于等于键  $k$ （可能为  $-\infty$ ）的最大键的元组  $p$  的位置。

##### 在跳跃表中查找

假设给出一个搜索键  $k$ 。我们开始 `SkipSearch` 方法，在跳跃表  $S$  中最顶层靠左的位置设立一个位置变量  $p$ ，并称为  $S$  的开始位置。这就是说，开始位置是在  $S_h$  中存储键为  $-\infty$  的特殊条目。然后我们执行以下步骤（如图 10-11 所示），`key(p)` 表示在位置  $p$  处的元组的键：

- 1) 如果  $S.below(p)$  为空，那么查找结束——我们在底部并且已经定位到了小于等于键  $k$  的最大值对应的元组在  $S$  中的位置。否则，我们通过设置  $p = S.below(p)$  从当前垂直塔位置下降到下一个水平层。
- 2) 从位置  $p$  开始，我们将  $p$  向前移动直到它在当前水平层的最右边的位置，这样 `key(p) ≤ k`。我们把这称为正向扫描步骤。注意，由于每个水平层都包含键  $+\infty$  和  $-\infty$ ，因此这一位置总是存在的。我们在这一水平层上执行扫描操作后  $p$  可能仍然在它开始的位置。
- 3) 返回到第 1 步。

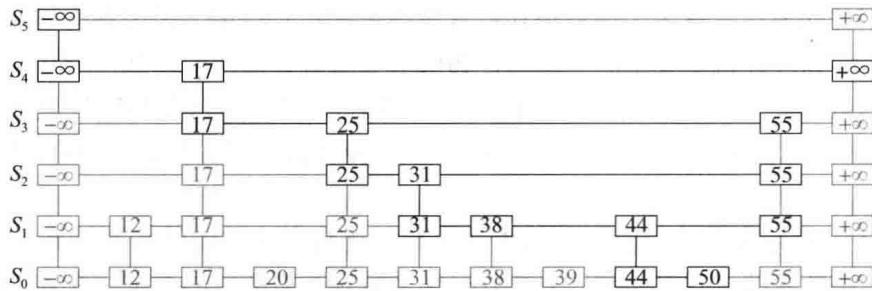


图 10-11 一个在跳跃表中搜索的例子。重点标记了查找键 50 时所检测的位置

我们在代码段 10-12 中给出了一个跳跃表查询算法 SkipSearch 的伪代码描述。鉴于这种方法，映射操作  $M[k]$  的执行是通过处理  $p = \text{SkipSearch}(k)$  和判断是否有  $\text{key}(p) = k$  来进行的。如果这两个键是相同的，我们将返回  $k$  对应的值，否则返回 KeyError。

#### 代码段 10-12 在跳跃表 $S$ 中查找键 $k$ 的算法

---

```

Algorithm SkipSearch( $k$ ):
    Input: A search key  $k$ 
    Output: Position  $p$  in the bottom list  $S_0$  with the largest key such that  $\text{key}(p) \leq k$ 
     $p = \text{start}$                                 {begin at start position}
    while  $\text{below}(p) \neq \text{None}$  do
         $p = \text{below}(p)$                         {drop down}
        while  $k \geq \text{key}(\text{next}(p))$  do
             $p = \text{next}(p)$                       {scan forward}
    return  $p$ .

```

---

事实证明，在含有  $n$  个条目的跳跃表中执行算法 SkipSearch 时期望的运行时间是  $O(\log n)$ 。我们把这个结论的验证推迟到讨论跳跃表更新的实现方法之后。可以简单地从 SkipSearch( $k$ ) 确认的位置开始导航，以便在有序映射 ADT 中提供其他的搜索形式（如 find\_gt、find\_range）。

#### 跳跃表中的插入操作

映射操作  $M[k] = v$  的执行是从 SkipSearch( $k$ ) 的调用开始的。这给了我们小于等于  $k$  的最大键的底层元组项的位置  $p$ （注意  $p$  可能是键为  $-\infty$  的特殊元组项）。如果  $\text{key}(p) = k$ ，其对应的值将被  $v$  覆盖。否则，我们需要为元组项  $(k, v)$  创造一个新的垂直塔。我们快速地  $S_0$  中将  $(k, v)$  插入  $p$  后面的位置。在最底层插入新元组后，我们使用随机方式来决定每个新元组的垂直塔高度。我们抛一枚硬币，如果出现反面，那么就停在这里。否则（出现正面），我们回溯到其前面（更高）的水平层并且在这一层的合适位置上插入  $(k, v)$ 。我们再次抛硬币，如果出现的是正面，就去下一个更高的水平层并重复相同操作。同时，我们向列表中重复插入元组  $(k, v)$  直到硬币抛出一个反面。我们将在这个过程中生成的新元组项  $(k, v)$  链接在一起并创建一个垂直塔。抛硬币可以由 Python 内置的随机数产生器模拟，通过调用 randrange(2) 来产生随机数，返回 0 或 1，生成这两个数的概率均为 1/2。

我们在代码段 10-13 中给出一个跳跃表  $S$  的插入算法并且在图 10-12 中作了解释。算法使用 `insertAfterAbove(p, q, (k, v))` 方法在位置  $p$  之后（在与  $p$  相同的层）且在位置  $q$  之上插入一个位置存储元组项  $(k, v)$ ，并且返回这个新位置  $r$ （并设置内部引用，以使得 `next`、`prev`、`above` 和 `below` 方法可以直接正常地为  $p$ 、 $q$  和  $r$  工作）。一个含有  $n$  个条目的跳跃表的插入算法的运行时间为  $O(\log n)$ ，这将在 10.4.2 节中进行说明。

代码段 10-13 跳跃表的插入操作。方法 coinFlip() 返回“正面”或“反面”，每一个值的出现概率都为 1/2。实例变量  $n$ 、 $h$  和  $s$  分别表示条目的数量、高度和跳跃表的开始节点

**Algorithm SkipInsert( $k, v$ ):**

**Input:** Key  $k$  and value  $v$

**Output:** Topmost position of the item inserted in the skip list

$p = \text{SkipSearch}(k)$

$q = \text{None}$  { $q$  will represent top node in new item's tower}

$i = -1$

**repeat**

$i = i + 1$

**if**  $i \geq h$  **then**

$h = h + 1$  {add a new level to the skip list}

$t = \text{next}(s)$

$s = \text{insertAfterAbove}(\text{None}, s, (-\infty, \text{None}))$  {grow leftmost tower}

$\text{insertAfterAbove}(s, t, (+\infty, \text{None}))$  {grow rightmost tower}

**while** above( $p$ ) is **None** **do**

$p = \text{prev}(p)$  {scan backward}

$p = \text{above}(p)$  {jump up to higher level}

$q = \text{insertAfterAbove}(p, q, (k, v))$  {increase height of new item's tower}

**until** coinFlip() == tails

$n = n + 1$

**return**  $q$

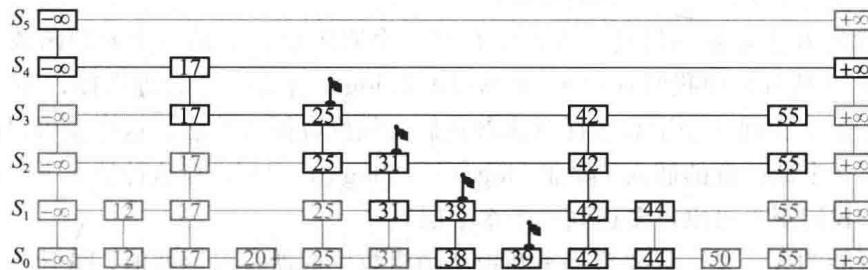


图 10-12 在图 10-10 的跳跃表中插入一个键为 42 的条目。我们假设为新条目随机“抛硬币”出现了三次正面，紧随其后的是反面。突出显示了访问过的位置。用粗线画出了插入的用于存储新条目的位置，并且它们之前的位置已经被标记了

### 在跳跃表中移除

跳跃表中的移除算法和搜索、插入算法是类似的。事实上，甚至比插入算法更简单。为了执行映射操作  $\text{del } M[k]$ ，我们首先执行方法  $\text{SkipSearch}(k)$ 。如果位置  $p$  存储的条目与键  $k$  不同，则返回 `KeyError`。否则，我们将移除  $p$  和  $p$  之上所有的位置，因为用 `above` 方法来访问  $S$  中从位置  $p$  开始的垂直塔更容易访问到这个条目。当移除塔中的各层时，我们将重新建立每一个移除位置与水平邻居之间的链接。删除算法在图 10-13 中阐述并将它的一个细节性的描述留作练习 R-10.24。正如我们在下一个部分中展示的，含有  $n$  个条目的跳跃表中删除操作的运行时间为  $O(\log n)$ 。

然而，在给出这个分析之前，我们想讨论一下对于跳跃表数据结构的一些小的改进。首先，我们实际上不需要存储在跳跃表底层之上的层中各值的引用，因为这些层中需要键的引用。事实上，我们可以更有效地视垂直塔为一个单独的对象，其能够存储键值对，如果垂直塔到达了  $S_j$  层，则维护  $j$  的 `previous` 引用和  $j$  的 `next` 引用。其次，对于水平轴能够保持只存储 `next` 引用的单向链表。我们可以通过自顶向下、正向扫描的更新来执行插入和删除操作。

我们将在练习 C-10.44 中探讨这种优化的细节。这两种优化都不能提升跳跃表的超过一个常数因子的性能，但是这些进步在实践中是有意义的。事实上，实验结果表明，在实际中优化跳跃表比 AVL 树和其他的平衡搜索树都快，这将在第 11 章中讨论。

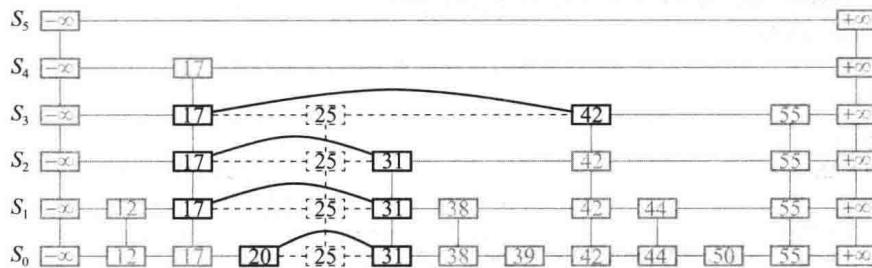


图 10-13 从图 10-12 中的跳跃表中删除键为 25 的条目。 $S_0$  中在搜索访问过的位置上的条目被加重显示了。移除的位置用虚线画出

### 维护最高水平层

跳跃表  $S$  必须维护一个引用初始位置作为实例变量（最顶部、最左方的位置），并且必须有一个策略服务于任何期望继续越过  $S$  顶层的插入一个新的条目的插入操作。我们可能采取两种不同路线的方法，每一种都有其优点。

一种可能的方法是限制最高层  $h$  保持在某一个固定值，这是一个  $n$  的函数，代表当前 map 的条目数。（从分析中我们看到  $h = \max\{10, 2 \lceil \log n \rceil\}$  是一个合理选择，并且挑选  $h = 3 \lceil \log n \rceil$  更安全。）实现这个函数选择意味着我们必须修改插入算法，这样就可以在一旦到达最高层时停止一个新位置的插入（除非  $\lceil \log n \rceil < \lceil \log(n - 1) \rceil$ ，在这种情况下，由于高度的边界在增长，我们至少可以再多达到一个水平层）。

另一种可能的方法是，只要头部不断地从随机数生成器获得返回值，就让插入操作持续插入新的位置。代码段 10-13 中的 `SkipInsert` 算法就是采用的这种方法。正如我们展示的跳跃表的分析那样，插入一个水平层的时间复杂度大于  $O(\log n)$  的概率非常低，所以这种方案可以正常工作。

以上任何一种方法都需要  $O(\log n)$  的时间复杂度来执行查找、插入和移除操作。这些我们将在下一小节进行说明。

### 10.4.2 跳跃表的概率分析 \*

正如我们上面所讨论的，跳跃表为有序映射提供了一个简单实现方法。从最坏的情况来看，跳跃表并不是一个较好的数据结构。事实上，如果我们不能正式地阻止一个插入持续通过当前的最高水平层，则插入算法可能会进入一个接近无限的循环（实际上不是一个无限循环，因为硬币永远出现正面的概率是 0）。此外，我们不能在不耗尽内存的情况下向一个列表中无限地添加新的位置。在任何情况下，如果我们在最高层  $h$  中停止位置插入操作，则在一个条目数为  $n$ 、高度为  $h$  的跳跃表  $S$  中运行 `__getitem__`、`__setitem__` 和 `__delitem__` 的操作时间是  $O(n + h)$ 。这种最坏的情况在每一个条目的垂直塔到达层  $h - 1$  时出现，这里  $h$  为  $S$  的高度。然而，发生这种情况的概率很低。根据这个最坏情况，我们可以得出这样的结论：跳跃表结构严格差于本章前面所讨论过的其他映射实现方法。但对于这种最坏情况下的行为总体上被高估了，这样的分析并不准确。

### 跳过跳跃表的高度

由于插入步骤包含随机化的内容，因此更精确的跳跃表应该适当考虑概率的问题。首先，关于完备和彻底的概率分析可能需要深入的数学知识（在数据结构研究文献中有一些深入分析），这似乎是首要的任务。幸运的是，这样的分析没有必要了解跳跃表的预期渐近行为。下面我们只用概率论的基本概念给出非正式的和直观的概率分析。

首先，让我们确定含  $n$  个条目的跳跃表的高度  $h$  的期望值（假设我们不会提前终止插入操作）。一个给定的条目中垂直塔的高度  $i \geq 1$  的概率等于抛一枚硬币连续  $i$  次出现正面的概率，即概率为  $1/2^i$ 。因此，水平层  $i$  至少有一个位置的概率  $P_i$  至多为

$$P_i \leq \frac{n}{2^i}$$

因为任何  $n$  个不同的事件同时发生的概率最多是每一个事件发生的概率的总和。

$S$  的高度为  $h$  的概率与层  $i$  至少有一个位置的概率相同，也就是说，它是不超过  $P_i$  的。这意味着  $h$  大于  $3\log n$  的可能性至多为

$$P_{3\log n} \leq \frac{n}{2^{3\log n}} = \frac{n}{n^3} = \frac{1}{n^2}$$

例如，如果  $n = 1000$ ，这个概率是一百万分之一。更一般的说法是，给出一个常量  $c > 1$ ， $h$  大于  $c \log n$  的概率至多为  $1/n^{c-1}$ 。也就是说， $h$  小于  $c \log n$  的概率至少为  $1 - 1/n^{c-1}$ 。因此， $S$  的高度为  $h$  的概率很可能为  $O(\log n)$ 。

### 分析跳跃表搜索时间

接下来，考虑一个跳跃表  $S$  在搜索时的运行时间，回想一下，这样一个搜索包含两层嵌套的 while 循环。只要下一个键不大于搜索键  $k$ ，内循环就一直在  $S$  的一个水平层上执行正向扫描，且外循环会降到下一层，重复这种扫描。由于  $S$  的高度  $h$  为  $O(\log n)$  的概率较高，降层循环的步数为  $O(\log n)$  的概率也较高。

我们还没有限制向前的步骤。令  $n_i$  为在层  $i$  上正向扫描时扫描过的键的数量。

可以看到，从开始位置之后，在层  $i$  中正向扫描扫描过的每一个额外的键都不能同时属于层  $i+1$ 。如果任何一个键在前一层，我们将在扫描前一层的过程中遇到这个键。因此，任何键被计数为  $n_i$  的概率都是  $1/2$ 。然而， $n_i$  的期望值与抛硬币时出现正面之前需要抛的次数是相等的。这个期望值是 2。因此，在任何层  $i$  正向扫描的期望时间为  $O(1)$ ，因为  $S$  很可能有  $O(\log n)$  层， $S$  中的搜索预期时间也就是  $O(\log n)$ 。通过类似的分析，我们可以得出插入或删除操作的预期运行时间为  $O(\log n)$ 。

### 跳跃表中的空间使用

最后，让我们看一下包含  $n$  个条目的跳跃表  $S$  的空间需求。正如我们在上面观察到的，层  $i$  可能含有的位置数为  $n/2^i$ ，这意味着  $S$  中准确的位置总数为

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^i}$$

用命题 3-5 几何求和。我们得到

$$\sum_{i=0}^h \frac{1}{2^i} = \frac{\left(\frac{1}{2}\right)^{h+1} - 1}{\frac{1}{2} - 1} = 2 \cdot \left(1 - \frac{1}{2^{h+1}}\right) < 2 \quad h \geq 0$$

因此， $S$  预期的空间需求为  $O(n)$ 。

表 10-4 总结了跳跃表实现的排序表的性能。

表 10-4 通过跳跃表实现有序映射的性能。我们使用  $n$  来表示执行操作时字典中条目的数量，预期的空间需求为  $O(n)$

操作	运行时间
len( $M$ )	$O(1)$
$k \in M$	期望为 $O(\log n)$
$M[k] = v$	期望为 $O(\log n)$
del $M[k]$	期望为 $O(\log n)$
$M.\text{find\_min}()$ , $M.\text{find\_max}()$	$O(1)$
$M.\text{find\_lt}(k)$ , $M.\text{find\_gt}(k)$ $M.\text{find\_le}(k)$ , $M.\text{find\_ge}(k)$	期望为 $O(\log n)$
$M.\text{find\_range}(\text{start}, \text{stop})$	期望为 $O(s + \log n)$ , 报告 $s$
iter( $M$ ), reversed( $M$ )	$O(n)$

## 10.5 集合、多集和多映射

我们通过分析几个和映射 ADT 密切相关并且用类似映射的数据结构实现的补充抽象来总结这一章。

- 集合 (set) 是无序元素的一个聚集，这些元素不重复并且通常支持高效的成员检测。从本质上说，集合中的元素像是映射中的键，但是它没有任何的附加值。
- 多集 (multiset)(也称为包 (bag)) 是一个允许有重复元素的类集合 (set-like) 容器。
- 多映射 (multimap) 与传统的映射类似，在映射中它将键和值联系起来。然而，在多映射中多个值可以映射到同一个键上。

### 10.5.1 集合的抽象数据类型

Python 通过内置类 frozenset 和 set 为表示集合中的数学概念提供支持，就像在第 1 章中讨论的，内置类 frozenset 是一个不可变的形式。这两个类都是使用 Python 中的哈希表来实现的。

Python 的 collections 模块定义了本质上反映这些内置类的抽象基类。然而对于名字的选择却是和直觉不同的，尽管抽象基类 collections.MutableSet 类似于具体的 set 类，但是抽象基类 collections.Set 匹配具体的 frozenset 类。

在讨论中，我们把“集合 ADT”等同于内置 set 类的行为（就是 collections.MutableSet 基类）。首先，我们列出了在集合  $S$  中最基本的五个行为：

- $S.\text{add}(e)$ : 向集合中添加元素  $e$ 。如果集合中已经包含了元素  $e$ ，则该行为无效。
- $S.\text{discard}(e)$ ：如果集合中包含元素  $e$ ，则从集合中删除该元素。如果集合中不包含元素  $e$ ，则该行为无效。
- $e \in S$ ：如果集合中包含元素  $e$ ，则返回 `True`，该行为是通过特定的方法 `__contains__` 来实现的。
- $\text{len}(S)$ ：返回集合  $S$  中的元素个数。在 Python 中，它是通过特定的方法 `__len__` 来实现的。

- `iter(S)`：生成集合中所有元素的迭代。在 Python 中，它是通过特定的方法 `__iter__` 来实现的。

在下一节中，我们将看到上述五种方法足以派生出一个集合的其他所有行为。这些剩余的行为可以自然地做如下归纳。首先，我们描述下列从一个集合中删除一个或多个元素的补充操作：

- `S.remove(e)`：将元素 `e` 从集合中删除。如果集合中不包含元素 `e`，将会产生一个错误 `KeyError`。
- `S.pop()`：从集合中删除并返回一个任意元素。如果集合为空，将会产生一个错误 `KeyError`。
- `S.clear()`：删除集合中的所有元素。

下一组行为将在两个集合之间进行布尔比较。

- `S == T`：如果集合 `S` 和集合 `T` 的内容相同，返回 `True`。
- `S != T`：如果集合 `S` 和集合 `T` 的内容不相同，返回 `True`。
- `S <= T`：如果集合 `S` 是集合 `T` 的子集，返回 `True`。
- `S < T`：如果集合 `S` 是集合 `T` 的真子集，返回 `True`。
- `S >= T`：如果集合 `T` 是集合 `S` 的子集，返回 `True`。
- `S > T`：如果集合 `T` 是集合 `S` 的真子集，返回 `True`。
- `S.isdisjoint(T)`：如果集合 `S` 和集合 `T` 没有公共元素，返回 `True`。

最后，还有一些基于经典集合理论操作的其他行为，它们要么是更新现有的集合，要么是计算一个新的集合实例。

- `S | T`：返回一个表示集合 `S` 和 `T` 的并集的新集合。
- `S |= T`：将 `S` 更新为集合 `S` 和 `T` 的并集。
- `S & T`：返回一个表示集合 `S` 和 `T` 的交集的新集合。
- `S &= T`：将 `S` 更新为集合 `S` 和 `T` 的交集。
- `S ^ T`：返回一个表示集合 `S` 和 `T` 的对称差集的新集合，也就是说，一组仅属于集合 `S` 或者仅属于集合 `T` 的元素。
- `S ^= T`：将集合 `S` 更新为它本身和集合 `T` 的对称差集。
- `S - T`：返回一个新的集合，该集合中包含集合 `S` 中的元素，但不包含集合 `T` 中的元素。
- `S -= T`：将 `S` 更新为删除集合 `S` 中与 `T` 相同的元素。

### 10.5.2 Python 的 `MutableSet` 抽象基类

为了辅助自定义 `set` 类的设计，Python 的 `collections` 模块提供了一个 `MutableSet` 抽象基类（就像在 10.1.3 节中讨论的，提供 `MutableMapping` 抽象基类）。`MutableSet` 抽象基类为 10.5.1 节中所描述的除了五种核心行为（`add`, `discard`, `__contains__`, `__len__`, `__iter__`）以外的所有方法提供了具体的实现方法，因为这五个核心行为必须通过任意具体的子类来实现。本设计是被称为模板方法模式的一个例子，因为 `MutableSet` 类的具体方法依赖于接下来的将由子类提供的假定抽象方法。

为了解释说明，我们对一些 `MutableSet` 基类的衍生方法的实现进行了研究。例如，为了确定一个集合是否是另一个集合的子集，我们必须验证两个条件：一个适合的子集大小必

须严格地小于它的超集，并且子集的每个元素必须包含在超集中。代码段 10-14 基于这个逻辑实现了相应的方法 `__lt__`。

**代码段 10-14 一种 MutableSet.`__lt__` 方法的实现，该方法检测一个集合是否恰好是另一个集合的子集**

---

```
def __lt__(self, other):      # supports syntax S < T
    """Return true if this set is a proper subset of other."""
    if len(self) >= len(other):
        return False           # proper subset must have strictly smaller size
    for e in self:
        if e not in other:
            return False       # not a subset since element missing from other
    return True                 # success; all conditions are met
```

---

在另外一个例子中，我们考虑两个集合并集的计算。集合 ADT 计算一个并集包括了两个形式。语法  $S \mid T$  应该产生一个新的集合，该集合的内容等于现有集合  $S$  和  $T$  的并集。这个操作是通过 Python 中的特殊方法 `__or__` 实现的。另一个语法  $S |= T$  用来更新现有的集合  $S$ ，使之成为它本身和集合  $T$  的并集。因此，集合  $T$  之前所有不包含在集合  $S$  中的元素应该被添加到集合  $S$  中。我们注意到可以比使用语法  $S = S \mid T$  的形式，更有效地实现这种“in-place”操作，其中标识符  $S$  被分配给表示并集的新集合实例。为方便起见，Python 内置的集合类支持这些行为的指定版本，`S.union(T)` 等价于  $S \mid T$ ，而 `S.update(T)` 等价于  $S |= T$ （然而，`MutableSet` 抽象基类没有正式地支持这些指定版本）。

在代码段 10-15 中，以的特定方法 `__or__` 的形式给出计算新集合作为另外两个集合并集的实现方法。在这个实现方法中一个重要的细节是结果集合的实例化。由于 `MutableSet` 类被设计成一个抽象基类，实例必须属于一个具体的子类。当计算这样两个具体实例的并集的时候，结果可能是一个和操作数相同的类的实例。函数 `type(self)` 返回一个指向标记为 `self` 的实例的实际类（actual class）的引用，并且在表达式 `type(self)()` 中，后面的括号里为这个类调用默认的构造函数。

**代码段 10-15 MutableSet.`__or__` 方法的实现，该方法计算两个集合的并集**

---

```
def __or__(self, other):      # supports syntax S | T
    """Return a new set that is the union of two existing sets."""
    result = type(self)()      # create new instance of concrete class
    for e in self:
        result.add(e)
    for e in other:
        result.add(e)
    return result
```

---

在效率方面，我们分析  $S \mid T$  这样的集合运算，其中用  $n$  表示  $S$  的大小，用  $m$  表示集合  $T$  的大小。如果用哈希实现具体的集合，则代码段 10-15 中的实现方法预期的运行时间是  $O(m + n)$ ，因为它在两个集合上循环，因此在一个包含检查和一个向结果集合中的插入操作的执行时间都是常数。

在代码段 10-16 中，给出了支持语法  $S |= T$  的特殊方法 `__ior__` “in-place” 版本的集合并操作的实现方法。注意，在这种情况下，我们不创建新的集合实例，而是更新返回现有的集合，更改集合的内容以反映集合并操作。这个版本的并集实现预计的运行时间为  $O(m)$ ，这里的  $m$  是第二个集合的大小，因为我们只需要在第二个集合中循环。

代码段 10-16 MutableSet.\_\_ior\_\_ 方法的实现，它执行一个集合和另一个集合的 in-place 并集

```

def __ior__(self, other):      # supports syntax S |= T
    """Modify this set to be the union of itself and another set."""
    for e in other:
        self.add(e)
    return self               # technical requirement of in-place operator

```

### 10.5.3 集合、多集和多映射的实现

#### 集合

虽然集合和映射有完全不同的公共接口，但是它们真的很相似。一个集合是一个简单的映射，这个映射中键没有相关联的值。任何一个数据结构实现的映射都可以改造以实现集合 ADT，并能够保障具有相似的性能。我们可以通过存储集合元素作为键，并使用 None 作为一个不相关的值来随便地应用任何映射类实现集合类，但是这样的实现造成了不必要的浪费。一个有效的集合类的实现方法应该放弃在 MapBase 类中使用的 \_Item 组合模式，而在数据结构中直接存储集合元素。

#### 多集

在多集中同一个元素可能出现多次。所有我们见过的数据结构都可以重新实现，并允许重复的元素作为不同的元素分别独立存在。然而，另外一种实现多集的方法是使用映射，该映射的键是多集中的元素（不同的），而键所关联的值是这个元素在多集中出现的次数。事实上，这本质上与我们在 10.1.2 节中所做的计算文档中单词出现次数的例子相同。

Python 的标准 collections 模块包括一个名为 Counter 类的定义，它本质上是一个多集。形式上，Counter 类是 dict 类的一个子类，它所包含的值最好都是整数，并且包含一些类似于 most\_common(*n*) 方法的附加函数，这里的函数 most\_common(*n*) 返回前 *n* 个最常见元素的列表。标准 \_\_iter\_\_ 方法对每个元素只报告一次（因为它们形式上是字典的键）。这里还有另外一个名为 elements() 的方法，该方法从头到尾地按元素的计数来重复遍历多集的每个元素。

#### 多映射

虽然在 Python 的标准库中没有多映射，但是一个常见的实现方法是使用一个标准映射，该映射与值相关联的键是一个本身存储任意数量的关联值的容器类。在代码段 10-17 中，我们举这样的一个 MultiMap 类的例子。我们用标准 dict 类实现映射，并且使用值的列表作为字典的组合值。我们设计该类，以使得不同的映射可以通过简单地重写第三行的类级 MapType 属性内容的方法进行实现。

代码段 10-17 一个使用 dict 作存储的 MultiMap 的实现。返回 self.\_n 的 \_\_len\_\_  
方法已从这个列表中省略

```

1 class MultiMap:
2     """A multimap class built upon use of an underlying map for storage."""
3     _MapType = dict           # Map type; can be redefined by subclass
4
5     def __init__(self):
6         """Create a new empty multimap instance."""
7         self._map = self._MapType()          # create map instance for storage
8         self._n = 0
9
10    def __iter__(self):
11        """Iterate through all (k,v) pairs in multimap."""

```

```

12     for k,secondary in self._map.items():
13         for v in secondary:
14             yield (k,v)
15
16     def add(self, k, v):
17         """Add pair (k,v) to multimap."""
18         container = self._map.setdefault(k, [ ])      # create empty list, if needed
19         container.append(v)
20         self._n += 1
21
22     def pop(self, k):
23         """Remove and return arbitrary (k,v) with key k (or raise KeyError)."""
24         secondary = self._map[k]                      # may raise KeyError
25         v = secondary.pop()
26         if len(secondary) == 0:
27             del self._map[k]                         # no pairs left
28         self._n -= 1
29         return (k, v)
30
31     def find(self, k):
32         """Return arbitrary (k,v) pair with given key (or raise KeyError)."""
33         secondary = self._map[k]                      # may raise KeyError
34         return (k, secondary[0])
35
36     def find_all(self, k):
37         """Generate iteration of all (k,v) pairs with given key."""
38         secondary = self._map.get(k, [ ])            # empty list, by default
39         for v in secondary:
40             yield (k,v)

```

## 10.6 练习

请访问 [www.wiley.com/college/goodrich](http://www.wiley.com/college/goodrich) 以获得练习帮助。

### 巩固

- R-10.1 只依靠类的五个主要的抽象方法，在 `MutableMapping` 类的背景下给出一个具体的 `pop` 方法的实现方法。
- R-10.2 只依靠五个主要的类的抽象方法，在 `MutableMapping` 类的背景下给出一个具体的 `items()` 方法的实现方法。如果直接应用 `UnsortedTableMap` 子类，它的运行时间将会是多少？
- R-10.3 直接在 `UnsortedTableMap` 类中给出一个具体的 `items()` 方法的实现方法，要确保整个迭代运行时间在  $O(n)$  之内。
- R-10.4 对一个用 `UnsortedTableMap` 类实现的初始为空的映射  $M$  插入  $n$  个键 - 值对，最坏情况下的运行时间是多少？
- R-10.5 使用 7.4 节中的 `PositionalList` 类而不是 Python 列表，重新实现 10.1.5 节中的 `UnsortedTableMap` 类。
- R-10.6 哪一个哈希表冲突处理方案可以允许一个负载因子在 1 以上，哪一个不能？
- R-10.7 列表和树的 `Position` 类支持 `__eq__` 方法，如果两个位置实例是指向同一个结构中的同一个基本节点，那么这两个位置实例被认为是等价的。允许位置作为哈希表中的键，必须有一个和等价的概念一致的 `__hash__` 方法的定义。请给出一个这样的 `__hash__` 方法。
- R-10.8 对于一个车辆识别码来说什么是好的哈希码？该车辆识别码是形为“9X9XX99X9XX999999”的一串数字和字母，其中“9”代表一个数字，“X”代表一个字母。
- R-10.9 使用哈希函数  $h(i) = (3i + 5) \bmod 11$  画出一个含有 11 个条目的哈希表，用来映射键 12、44、13、88、23、94、11、39、20、16 和 5，假设链已经处理了冲突。

- R-10.10 假设线性探测已经处理了冲突，那么上题的结果是什么？
- R-10.11 假设二次探测已经处理了冲突，演示练习 R-10.9 的结果，直到该方法失败的位置。
- R-10.12 当二次哈希已经使用二次哈希函数  $h(k) = 7 - (k \bmod 7)$  处理了冲突时，练习 R-10.9 的结果是什么？
- R-10.13 假设链表已经处理了冲突，把  $n$  个条目置于初始为空的哈希表中最坏情况下的时间是多少？最好情况下是多少？
- R-10.14 给出使用一个新哈希函数  $h(k) = 3k \bmod 17$  将图 10-6 中的哈希表重映射到一个新的表中的结果。
- R-10.15 我们的 HashMapBase 类维护了一个负载因子  $\lambda \leq 0.5$ 。重新实现类使之允许用户指定最大负载，并相应地调整具体子类。
- R-10.16 写出一个使用二次探测解决冲突的哈希表插入算法的伪代码，假设我们也使用一个特殊的“关闭条目”对象来替换删除条目的方法。
- R-10.17 使用二次探测修改 ProbeHashMap 类。
- R-10.18 试说明为什么哈希表不适合实现排序映射。
- R-10.19 描述如何用排序表实现的双向列表来实现有序映射 ADT？
- R-10.20 对一个初始包含  $2n$  项的 SortedTableMap 实例执行  $n$  次删除，最坏情况下的渐近运行时间是多少？
- R-10.21 在 SortedTableMap 类的背景下，考虑以下对代码段 10-8 中 `_find_index` 方法的变形。

```
def _find_index(self, k, low, high):
    if high < low:
        return high + 1
    else:
        mid = (low + high) // 2
        if self._table[mid].key < k:
            return self._find_index(k, mid + 1, high)
        else:
            return self._find_index(k, low, mid - 1)
```

这是否能产生和原始版本产生相同的结果？证明你的结论。

- R-10.22 如果我们做  $n$  项的插入操作，其中每项的性能和价格低于它的前一项，维护一个最大集的方法的预期运行时间是多少？在有序映射中一系列操作的最后包含了什么？如果每项比之前的一项有更低的成本和更高的性能呢？
- R-10.23 在图 10-13 所示的跳跃表中，画一个跳跃表 S 执行操作序列 `delS[38]`, `S[48] = 'x'`, `S[24] = 'y'`, `del S[55]` 的结果。同时记录你抛的硬币的结果。
- R-10.24 给出一个使用跳跃表的映射操作 `__delitem__` 的伪代码。
- R-10.25 给出一个在 MutableSet 抽象基类的背景下 `pop` 方法的具体实现方法，只依靠 10.5.2 节中五个核心集合行为来描述。
- R-10.26 在 MutableSet 抽象基类的背景下给出一个具体的 `isdisjoint` 方法的实现方法，只依靠该类的五个主要的抽象方法。这个算法应该在  $O(\min(n, m))$  内运行，其中  $n$  和  $m$  表示两个集合各自的基。
- R-10.27 你会用什么样的抽象来管理一个朋友生日的数据库，以支持“找到所有生日为今天的朋友”或者“找到谁将是下一个庆祝生日的朋友”这样的有效查询。

## 创新

- C-10.28 在 10.1.3 节中，我们给出了一个应该出现在 MutableMapping 抽象基类中的 `setdefault` 方法的实现方法。而当该方法以一般的方式完成目标时，它的效率并不理想。特别的，若键是新的，由于 `__getitem__` 的初次使用，并随后通过 `__setitem__` 来执行插入，会导致搜索失败。

对于一个具体的实现，例如 `UnsortedTableMap`，这是两倍的工作量，因为在 `__getitem__` 失败期间会发生一个完整的表的扫描，并且接下来因为 `__setitem__` 的实现会生成另一个完整的表的扫描。一个更好的解决方案是为 `UnsortedTableMap` 类重写 `setdefault` 以提供一个直接的执行单个搜索的解决方案。给出这样一个 `UnsortedTableMap.setdefault` 的实现方法。

- C-10.29 重新实现练习 C-10.28 的 `ProbeHashMap` 类。
- C-10.30 重新实现练习 C-10.28 的 `ChainHashMap` 类。
- C-10.31 对于一个理想的压缩函数，哈希表桶（bucket）数组的容量应该是一个素数。那么，让我们考虑在  $[M, 2M]$  的范围内定位一个素数的问题。试实现一个方法，通过使用筛选法找到这样的素数。在该算法中，我们分配一个含  $2M$  个布尔型单元（cell）的数组  $A$ ，其中的单元  $i$  与整数  $i$  相关联。接下来将该数组所有的单元都初始化为“真”，并“标出”所有  $2, 3, 5, 7$  等素数倍数的单元。在达到一个大于  $\sqrt{2M}$  的数字后，这个过程可以停止。（提示：考虑拔靴方法（bootstrapping）来寻找素数到  $\sqrt{2M}$ 。）
- C-10.32 在 `ChainHashMap` 和 `ProbeHashMap` 类上进行试验，测量二者在使用随机密钥集和改变负载因子限制时的效率（参见练习 R-10.15）。
- C-10.33 我们在 `ChainHashMap` 中实现的分离链表，通过用 `None` 表示空桶而不是二级结构的空实例来节省内存。由于其中的很多桶将保持一个项目，因此更好的优化方法是使表中的这些位置直接引用 `_Item` 实例，并且对含有两个或更多项目的桶使用二次容器。重写这个实现以提供这种额外的优化。
- C-10.34 计算一个哈希代码，尤其是当键较长时计算的代价可能是昂贵的。在我们的哈希表实现中，第一次插入项目时我们计算哈希代码，且每次重新计算条目的哈希代码时都要调整表的大小。Python 的 `dict` 类有一个有趣的折衷，在插入一个项目时计算一次哈希码，并存储哈希码为项目组合的一个附加的域，这样就不需要重新计算了。使用这样的方法重新实现我们的 `HashMapBase` 类。
- C-10.35 描述怎样从哈希表中进行删除操作，在这个哈希表中我们不用特殊标记表示已删除的元素，而是用线性探测来解决冲突。也就是说，我们必须重新整理内容，以使得已经删除的条目不会再插入表中的第一个位置。
- C-10.36 二次探测策略有一个与寻找开放位置方法相关的聚类问题。就是说，当在桶  $h(k)$  中发生冲突时，会检查桶  $A[(h(k) + i^2) \bmod N], i = 1, 2, \dots, N - 1$ 。
  - a) 对于素数  $N$ ,  $i$  的范围从 1 到  $N - 1$ ，假设  $i^2 \bmod N$  至多有  $(n + 1)/2$  个不同的值。基于这个假设，注意对于所有的  $i$ ，有  $i^2 \bmod N = (N - i)^2 \bmod N$ 。
  - b) 更好的策略是选择一个素数  $N$ ，其中  $N \bmod 4 = 3$ ，然后检查桶  $A[(h(k) \pm i^2) \bmod N], i$  从 1 到  $(N - 1)/2$ ，正负交替。证明这种交替版本可以保证  $A$  中每个桶都会检查到。
- C-10.37 重构 `ProbeHashMap` 的设计，以使二次探测序列能更方便地定制解决冲突。通过分别为线性探测和二次探测提供具体的子类来证明这个新框架。
- C-10.38 重新设计一个使用排序查找表的二分法查找，实现多集操作 `find all( $k$ )`，表中包括重复项，并证明它的运行时间是  $O(s + \log n)$ 。其中  $n$  是字典中元素的个数， $s$  是键为  $k$  的项目的个数。
- C-10.39 尽管映射中的键是不同的，但是二分查找算法可以应用于更一般的环境中，在这样的环境中用一个数组以非降序的方式存储可能存在重复的各个元素。考虑识别最左边键大于等于给定  $k$  的元素索引的目的。代码段 10-8 所给出的 `_find_index` 方法是否能保证这一结果？在练习

R-10.21 中给出的 `_find_index` 方法是否能保证这种结果？证明你的结论。

- C-10.40 假设我们给出了两个排序搜索表  $S$  和  $T$ ，每个表中都有  $n$  个条目（ $S$  和  $T$  都通过数组实现），描述一个运行时间为  $O(\log^2 n)$  的算法来在  $S$  和  $T$  的并集中找到第  $k$  小的键（假设没有重复）。
- C-10.41 给出对上一个问题运行时间为  $O(\log n)$  的解决方案。
- C-10.42 假设一个  $n \times n$  的数组  $A$  每行都由 1 和 0 组成，在任何一行中，所有的 1 都在 0 之前出现。假设  $A$  已经载入内存，描述一个  $O(n \log n)$  时间内运行（不是  $O(n^2)$  时间内）的计算  $A$  中 1 的个数的方法。
- C-10.43 给出一个含有  $n$  个价格 - 性能对  $(c, p)$  的集合  $C$ ，描述一个在  $O(n \log n)$  时间内发现  $C$  的极值对的算法。
- C-10.44 证明方法 `above(p)` 和 `prev(p)` 实际上不需要使用跳跃表来实现映射。也就是说，我们可以在跳跃表中，通过使用严格的自上而下、正向扫描方法实现插入和删除，而不需要使用 `above` 或者 `prev` 方法。（提示：在插入算法中，首先通过反复地掷硬币来确定应该在哪个水平层开始插入新的条目。）
- C-10.45 描述如何修改一个基于索引操作的跳跃表形式，例如在索引  $j$  中检索条目，可以在预期时间  $O(\log n)$  内完成。
- C-10.46 对于集合  $S$  和  $T$ ，语法  $S \setminus T$  返回一个称为对称差的新集合，即这个集合中的元素包含在  $S$  或者  $T$  两者之一中。`__xor__` 方法支持该语法。在 `MutableSet` 抽象基类的背景下，给出一个该方法的实现方法，只依靠该抽象基类的五个主要的抽象方法。
- C-10.47 描述一个基于 `MutableSet` 抽象基类的 `__and__` 方法的具体实现方法，该方法支持计算两个现有集合交集的语法  $S \& T$ 。
- C-10.48 对于实现搜索引擎或书的索引，倒排文件是一个重要的数据结构。给定的文件  $D$  可以被视为一个单词无序的、编号的列表；倒排文件则是一个单词的排序列表，例如列表  $L$ ，对于每一个在  $L$  中的单词  $w$ ，我们存储  $D$  中出现  $w$  的位置的索引。设计一个在  $D$  中构造  $L$  的有效算法。
- C-10.49 Python 的 `collections` 方法提供了一个 `OrderedDict` 类，它和有序映射抽象无关。`OrderedDict` 类是标准的基于映射的 `dict` 类的子类，它的主要映射操作保持预期的时间执行为  $O(1)$ ，但是它也保证 `__iter__` 方法依先进先出（FIFO）的顺序报告映射中的条目。这就是说，最先报告字典中保存时间最长的键。（当已有键的值被重写时，顺序是不受影响的。）写一个符合这样的性能要求的算法。

## 项目

- P-10.50 进行一项比较分析，研究各种字符串的哈希代码的冲突率，例如比较各种参数  $a$  值不同的多项式哈希代码。使用哈希表来检测冲突，但只计算那些不同字符串映射到相同哈希代码中的冲突（除非它们映射到该哈希表的同一位置）。用在互联网上找到的文本文件来测试这些哈希函数。
- P-10.51 在 10 位数字的电话号码而不是字符串的哈希码上实施上一练习中的比较分析。
- P-10.52 实现一个 `OrderedDict` 类，如练习 C-10.49 中描述的那样，确保主要的映射操作预期的运行时间为  $O(1)$ 。
- P-10.53 设计一个实现跳跃表数据结构的 Python 类。使用这个类创建一个完整的有序映射 ADT 的实现。

P-10.54 通过提供跳跃表操作的图形动画扩展前一个问题。可视化展示如何在插入时完全移动跳跃表，以及如何在删除时和跳跃表断开联系。此外，在搜索操作中，设想正向扫描和下降动作。

P-10.55 写一个存储 Python 集合中的单词  $W$  的拼写检查器类，并实现  $\text{check}(s)$  方法，该方法在关于单词集合  $W$  的字符串  $s$  中执行拼写检查。如果  $s$  在  $W$  中，那么调用  $\text{check}(s)$  返回一个只包含  $s$  的列表，假定  $s$  在该情况下是拼写正确的。如果  $s$  不在  $W$  中，则调用  $\text{check}(s)$  返回一个  $W$  中每个可能是  $s$  的正确拼写的单词列表。程序应该能够处理所有常见的问题， $s$  有可能是  $W$  中一个拼错的词，包括：单词中相邻的字母顺序颠倒，在单词中两个相邻字母间插入一个字母，从单词中删除一个字母，单词中的一个字母被另外一个字母代替。也考虑发音相似的替换，这是一个额外的挑战。

## 扩展阅读

哈希是一个被深入研究的技术。感兴趣的读者可以进一步研究 Knuth<sup>[65]</sup>、Vitter 和 Chen<sup>[100]</sup> 的书。Pugh<sup>[86]</sup> 介绍了跳跃表。我们对于跳跃表的研究是 Motwani 和 Raghavan<sup>[80]</sup> 所给出的报告的一个简化。对于跳跃表更深入的分析，请参阅数据结构文献 [ 59, 81, 84 ] 中各种跳跃表的研究论文。练习 C-10.36 是 James Lee 的研究内容。

## 搜索 树

## 11.1 二叉搜索树

在第 8 章中，我们介绍了树型数据结构，并且演示了多种应用程序。树型数据结构的一个重要用途是用作搜索树。在本章中，我们使用搜索树结构来有效地实现有序映射。映射 M 的三种最基本的方法（见 10.1.1 节）为：

- $M[k]$ : 在映射  $M$  中，如果存在与键  $k$  相关联的值  $v$ ，返回  $v$ ；否则，抛出 `KeyError`。用 `__getitem__` 方法来实现。
- $M[k] = v$ : 在映射  $M$  中，将键  $k$  与值  $v$  相关联，如果映射中已经包含键等于  $k$  的项，则用  $v$  替换现有值。用 `__setitem__` 方法来实现。
- `del M[k]`: 从映射  $M$  里删除键等于  $k$  的项；如果  $M$  中没有这样的项，则引发 `KeyError`。用 `__delitem__` 方法来实现。

有序映射 ADT 包括许多附加功能（见 10.3 节），以保证迭代器按照一定顺序输出键，并且支持额外的搜索，如 `find_gt(k)` 和 `find_range`( $start, stop$ )。

假设已经根据键得到次序关系，对于存储这些数据，二叉树是一个很好的数据结构。在本章中，二叉搜索树是每个节点  $p$  存储一个键值对  $(k, v)$  的二叉树  $T$ ，使得：

- 存储在  $p$  的左子树的键都小于  $k$ 。
- 存储在  $p$  的右子树的键都大于  $k$ 。

图 11-1 给出了二叉搜索树的例子。为了方便，我们在本章中不会用图解法表示与键关联的值，因为这些值不影响这些项在搜索树中的位置。

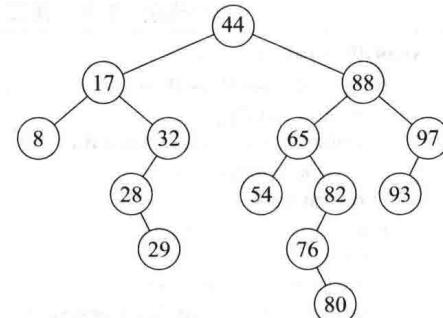


图 11-1 用整数键表示的二叉搜索树。在本章中我们省略关联的值，因为它们在一个搜索树中与项的顺序无关

## 11.1.1 遍历二叉搜索树

我们首先指明，二叉搜索树分层地表示了键的排列顺序。特别地，二叉搜索树中关于键的位置的结构特性使得树的遍历是中序遍历（见 8.4.3 节）。

**命题 11-1:** 二叉搜索树的中序遍历是按照键增加的顺序进行的。

**证明:** 我们通过对子树的大小进行归纳来证明这一命题。如果一个子树至多有一个节点，它的键都是按照顺序访问的。一般来说，(子) 树的中序遍历首先是左子树（可能为空）的递归遍历，其次是根节点访问，最后是右子树（可能为空）的递归遍历。综上所述，左子树递归地进行中序遍历会在该子树上以递增的顺序产生键的迭代。而且，根据二叉搜索树的特性，左子树上所有节点的键都比根节点的小。因此，在按值的递增顺序访问完左子树之后再访问其根节点。最后，根据搜索树的特性，右子树上所有节点的键都比根节点的大，通过

归纳可知，该子树的中序遍历将按键的递增顺序访问右子树。

由于中序遍历可以在线性时间内被执行，当对二叉搜索树进行中序遍历时，根据以上定理我们可以在线性时间内产生一个映射中所有键的有序迭代。

虽然通常使用自顶向下的递归来表示中序遍历，我们还是可以提供一些操作的非递归说明，这些操作即允许在与键的顺序相关的二叉搜索的位置之中进行更细粒度地遍历。第 8 章的一般二叉树 ADT 被定义成一个位置结构，允许使用诸如 `parent(p)`、`left(p)` 和 `right(p)` 的方法直接定位。对于二叉搜索树，我们可以基于存储在树中的键的自然顺序提供额外的定位。特别地，我们可以支持下面的方法——类似于由 `PositionalList`（见 7.4.1 节）提供的方法。

- `frist()`: 返回一个包含最小键的节点，如果树为空，则返回 `None`。
- `last()`: 返回一个包含最大键的节点，如果树为空，则返回 `None`。
- `before(p)`: 返回比节点  $p$  的键小的所有节点中键最大的节点（即中序遍历中在  $p$  之前最后一个被访问的节点），如果  $p$  是第一个节点，则返回 `None`。
- `after(p)`: 返回比节点  $p$  的键大的所有节点中键最小的节点（即中序遍历中在  $p$  之后第一个被访问的节点），如果  $p$  是最后一个节点，则返回 `None`。

二叉搜索树的“第一个”位置可以从根开始，并且只要左子树存在就继续搜索左子树。与之相对的，通过从根开始向右进行重复的步骤到达最后的位置。

节点的后继 `after(p)` 由下述算法确定。

#### 代码段 11-1 在二叉搜索树中计算某一位置的后继节点

---

**Algorithm** `after(p)`:

```

if right(p) is not None then {successor is leftmost position in p's right subtree}
    walk = right(p)
    while left(walk) is not None do
        walk = left(walk)
    return walk
else {successor is nearest ancestor having p in its left subtree}
    walk = p
    ancestor = parent(walk)
    while ancestor is not None and walk == right(ancestor) do
        walk = ancestor
        ancestor = parent(walk)
    return ancestor
```

---

这个过程的基本原理完全是基于中序遍历的算法，与命题 11-1 相一致。如果  $p$  节点有一个右子树， $p$  节点被访问之后右子树立即被递归遍历，所以  $p$  节点之后第一个被访问到的节点是其右子树的最左节点。如果  $p$  节点没有右子树，则中序遍历的控制流返回到  $p$  节点的父节点。如果  $p$  节点是在父节点的右子树，那么父节点的子树遍历完成，控制流前进到该父节点的父节点并继续执行。一旦递归从其左子树回来到达一个祖先节点，那么这个祖先节点变成遍历的下一个节点，因而是  $p$  节点的后继。请注意，只有在  $p$  节点是整棵树的最右（最后）节点并发现没有这样的祖先的情况下，没有后继节点。

节点的前驱可以使用对称的算法来确定，即 `before(p)`。在这一点上，我们发现单独调用 `after(p)` 或者 `before(p)` 的运行时间受整棵树高度  $h$  的约束，因为它要么是向下走，要么是向上走。在最坏情况下运行时间为  $O(h)$ ，上述两种方法执行的摊销时间为  $O(1)$ ，从第一个节点开始  $n$  次调用 `after(p)` 的总时间为  $O(n)$ 。我们将这一证明留作练习 C-11.34，这直观地模拟了中序遍历向上和向下的操作步骤（相关参数在命题 9-3 中）。

### 11.1.2 搜索

二叉搜索树的结构特性产生的最重要的结果是搜索算法。我们可以尝试在一棵二叉搜索树中通过把它表示成决策树的形式定位一个特定的键(见图 8-7)。在这种情况下, 在每个节点  $p$  的问题就是期望的键  $k$  是否小于、等于或大于存储在节点  $p$  的键, 这表示为  $p.key()$ 。如果答案是“小于”, 那么继续搜索左子树。如果答案是“等于”, 那么搜索成功终止。如果答案是“大于”, 那么继续搜索右子树。最后, 如果得到空的子树, 那么就是没有搜索到, (如图 11-2 所示)。

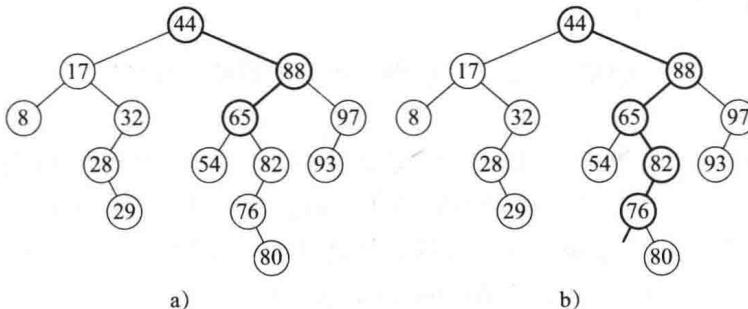


图 11-2 a) 在二叉树上成功搜索键 65; b) 在二叉树上没有搜索到键 68, 因为键 76 的左边没有子树

我们将在代码段 11-2 中描述这种方法。如果要搜索的键为  $k$ , 它出现在以  $p$  节点为根的子树中, 调用  $\text{TreeSearch}(T, p, k)$  可以得到键  $k$  的位置; 在这种情况下,  $\text{__getitem__}$  映射操作将返回相关联的值。在寻找未果的情况下,  $\text{TreeSearch}$  算法返回搜索路径的最终位置(我们将会使用该位置决定在哪里插入一个新的节点)。

#### 代码段 11-2 二叉树搜索的递归调用

```
Algorithm TreeSearch(T, p, k):
    if k == p.key() then
        return p
    else if k < p.key() and T.left(p) is not None then
        return TreeSearch(T, T.left(p), k)
    else if k > p.key() and T.right(p) is not None then
        return TreeSearch(T, T.right(p), k)
    return p
```

{successful search}  
 {recur on left subtree}  
 {recur on right subtree}  
 {unsuccessful search}

#### 二叉树搜索的分析

二叉树  $T$  搜索的最坏运行时间的分析很容易。 $\text{TreeSearch}$  算法是递归的, 并且每个递归调用执行恒定数量的原语操作。 $\text{TreeSearch}$  的每次递归调用是对前一个位置的子节点做的。也就是说,  $\text{TreeSearch}$  在  $T$  的路径中的各个节点上被调用, 从根节点开始每一次下降一层。因此, 节点的数目被限定为  $h + 1$ , 其中  $h$  是  $T$  的高度。换句话说, 因为每一个节点的搜索时间为  $O(1)$ , 则总的搜索运行时间为  $O(h)$ , 其中  $h$  是二叉搜索树  $T$  的高度(见图 11-3)。

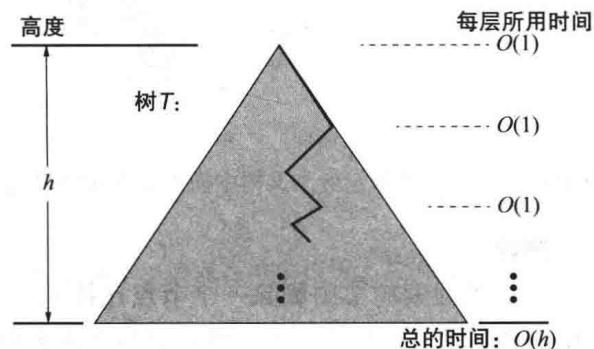


图 11-3 说明二叉搜索树的运行时间。其中将二叉搜索树看作一个大三角形, 那么从根节点开始的搜索路径就是该三角形内的锯齿形线

在有序映射 ADT 中，搜索将作为实现 `__getitem__` 以及 `__setitem__` 和 `__delitem__` 方法的子程序，因为这些方法都需要通过一个给定的键查找一个现有节点。为了实现有序映射操作（如 `find_lt` 和 `find_gt`），我们将把搜索和遍历方法 `before` 和 `after` 结合起来使用。当树的高度为  $h$  时，所有这些操作在最坏情况下的时间复杂度将为  $O(h)$ 。我们可以使用修改后的算法在时间  $O(s + h)$  内来实现 `find_range` 方法，其中  $s$  是节点数（见练习 C-11.34）。

当然， $T$  的高度  $h$  可以和节点的数量  $n$  一样大，但一般情况下小得多。在本章后面，我们将展示各种策略，使得搜索树  $T$  的高度的上界为  $O(\log n)$ 。

### 11.1.3 插入和删除

插入或删除二叉搜索树的项的算法虽然很常用，但是相当简单。

#### 插入

映射命令  $M[K] = v$ ，在 `__setitem__` 方法的支持下，首先搜索键为  $k$  的项（假设映射不能为空）。如果找到，该节点将被重新赋值；否则，新的节点可以插到树  $T$  的下一层，代替搜索失败结束时得到的空子树。在二叉搜索树持续操作该位置（注意，恰好放置在一个搜索期望的地方）。代码段 11-3 给出了 `TreeInsert` 算法的伪代码。

代码段 11-3 在表示为二叉搜索树的映射中插入键 – 值对的算法

---

```
Algorithm TreeInsert( $T, k, v$ ):
    Input: A search key  $k$  to be associated with value  $v$ 
     $p = \text{TreeSearch}(T, T.\text{root}(), k)$ 
    if  $k == p.\text{key}()$  then
        Set  $p$ 's value to  $v$ 
    else if  $k < p.\text{key}()$  then
        add node with item  $(k, v)$  as left child of  $p$ 
    else
        add node with item  $(k, v)$  as right child of  $p$ 
```

---

图 11-4 所示为插入二叉搜索树的一个例子。

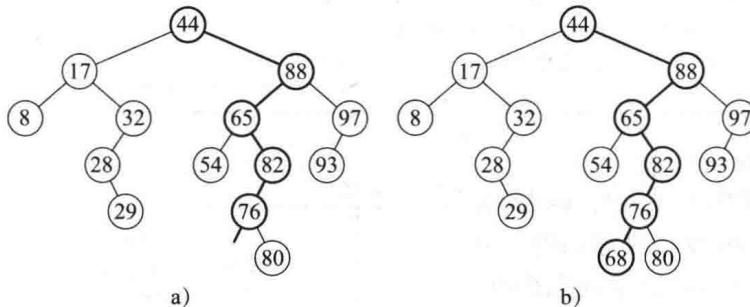


图 11-4 在图 11-2 所示的二叉树中插入键为 68 的节点。a) 表示找出插入位置，b) 表示最终插入后的树

#### 删除

从二叉搜索树  $T$  中删除一个节点比插入一个新的节点更复杂，因为删除的位置可能在树中的任何地方（相比之下，插入总是在搜索路径的最后位置）。要使用键  $k$  删除一个节点，首先通过调用 `TreeSearch(T, T.root(), K)` 找到  $T$  中键等于  $k$  的节点的位置  $p$ 。如果搜索成功，则分成以下两种情况（难度增加）：

- 如果  $p$  最多有一个孩子，删除位置  $p$  上的节点就很容易实现。在 8.3.1 节介绍 `LinkedBinary`

Tree 类的更新方法时，我们就定义了一个非公开的实体 `_delete(p)`，假设  $p$  至多有一个孩子，就删除位置  $p$  的节点并用其子节点替换它（如果有子节点的话）。这正是我们所期望的行为。从映射中删除与键  $k$  有关联的节点，同时保持其他所有祖先 - 后继在树中的关系，从而维持了二叉搜索树的属性（见图 11-5）。

- 如果位置  $p$  有两个孩子，我们不能简单地去除  $T$  中的节点，因为这将创建一个“漏洞”并使两个子节点成为孤儿。所以，应采用如下操作步骤（见图 11-6）：
  - 通过 11.1.1 节的公式  $r = \text{before}(p)$ ，定位到小于位置  $p$  的键的键最大的节点的位置  $r$ 。由于  $p$  有两个孩子，其前继是  $p$  的左子树中最右边的位置。
  - 使用位置  $r$  的节点作为位置  $p$  被删除的节点的替代。因为  $r$  在映射中具有紧邻的前一个键， $p$  节点右子树中的任何一个节点都有比  $r$  节点更大的键， $p$  的左子树中的任何一个节点都有比  $r$  节点更小的键。因此，在替换后维持了二叉树的属性。
  - 使用  $r$  节点作为  $p$  节点的替代以后，我们从树中删除原来  $r$  位置的节点。幸运的是，因为  $r$  节点被定位为在子树中最右边的位置，所以  $r$  节点没有右子树。因此，它可以使用第一种方法（更简单）来进行删除。

就像搜索和插入一样，删除算法涉及从根开始的单一路径的遍历，可能移动节点或者移除路径中的节点并提升其子节点。因此，当树的高度是  $h$  时，执行时间复杂度为  $O(h)$ 。

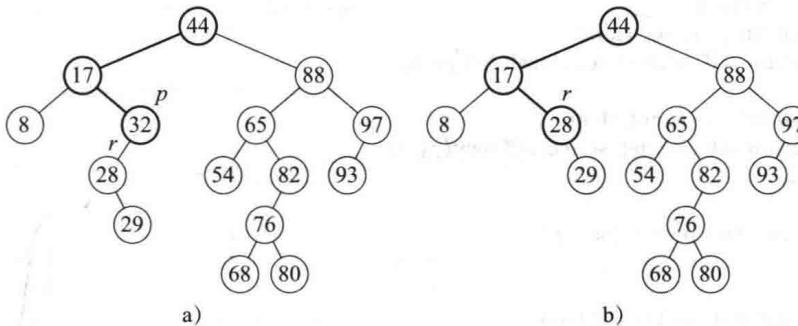


图 11-5 从图 11-4b 所示的二叉树中删除  $p$  位置的节点（键为 32）， $p$  节点有一个子节点  $r$ 。a) 是删除之前，b) 是删除之后

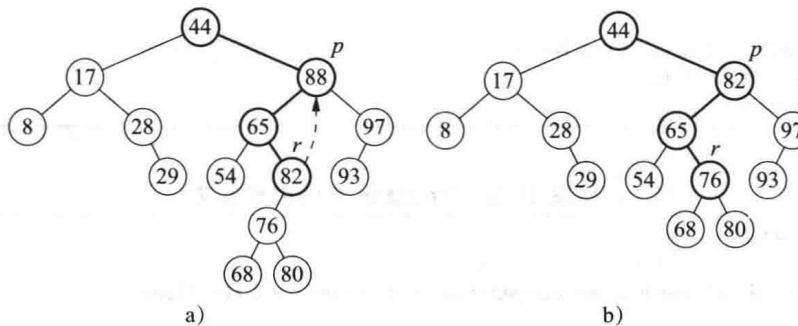


图 11-6 从图 11-5b 所示的二叉树中删除  $p$  节点（键为 88）， $p$  节点有两个孩子，它的位置将被其前驱  $r$  代替。a) 是删除之前，b) 是删除之后

#### 11.1.4 Python 实现

在代码段 11-4 ~ 11-8 中，我们定义了一个使用二叉搜索树实现有序映射 ADT 的

TreeMap 类。事实上，我们的实现更普通。我们支持所有标准映射操作（见 10.1.1 节）、所有附加有序映射操作（见 10.3 节）和对位置的操作（包括 first()、last()、find position(k)、before(p)、after(p) 以及 delete(p)）。

#### 代码段 11-4 基于二叉搜索树的 TreeMap 类的开始

```

1 class TreeMap(LinkedBinaryTree, MapBase):
2     """Sorted map implementation using a binary search tree."""
3
4     #----- override Position class -----
5     class Position(LinkedBinaryTree.Position):
6         def key(self):
7             """Return key of map's key-value pair."""
8             return self.element().__key
9
10        def value(self):
11            """Return value of map's key-value pair."""
12            return self.element().__value
13
14        #----- nonpublic utilities -----
15        def _subtree_search(self, p, k):
16            """Return Position of p's subtree having key k, or last node searched."""
17            if k == p.key():                                # found match
18                return p
19            elif k < p.key():                             # search left subtree
20                if self.left(p) is not None:
21                    return self._subtree_search(self.left(p), k)
22                else:                                     # search right subtree
23                    if self.right(p) is not None:
24                        return self._subtree_search(self.right(p), k)
25                return p                                  # unsuccessful search
26
27        def _subtree_first_position(self, p):
28            """Return Position of first item in subtree rooted at p."""
29            walk = p
30            while self.left(walk) is not None:           # keep walking left
31                walk = self.left(walk)
32            return walk
33
34        def _subtree_last_position(self, p):
35            """Return Position of last item in subtree rooted at p."""
36            walk = p
37            while self.right(walk) is not None:          # keep walking right
38                walk = self.right(walk)
39            return walk

```

#### 代码段 11-5 TreeMap 类的引导方法

```

40    def first(self):
41        """Return the first Position in the tree (or None if empty)."""
42        return self._subtree_first_position(self.root()) if len(self) > 0 else None
43
44    def last(self):
45        """Return the last Position in the tree (or None if empty)."""
46        return self._subtree_last_position(self.root()) if len(self) > 0 else None
47
48    def before(self, p):
49        """Return the Position just before p in the natural order.
50
51        Return None if p is the first position.

```

```

52     """
53     self._validate(p)           # inherited from LinkedBinaryTree
54     if self.left(p):
55         return self._subtree_last_position(self.left(p))
56     else:
57         # walk upward
58         walk = p
59         above = self.parent(walk)
60         while above is not None and walk == self.left(above):
61             walk = above
62             above = self.parent(walk)
63         return above
64
65     def after(self, p):
66         """Return the Position just after p in the natural order.
67
68         Return None if p is the last position.
69         """
70         # symmetric to before(p)
71
72     def find_position(self, k):
73         """Return position with key k, or else neighbor (or None if empty)."""
74         if self.is_empty():
75             return None
76         else:
77             p = self._subtree_search(self.root(), k)
78             self._rebalance_access(p)      # hook for balanced tree subclasses
79             return p

```

#### 代码段 11-6 TreeMap 类的一些有序映射操作

```

80     def find_min(self):
81         """Return (key,value) pair with minimum key (or None if empty)."""
82         if self.is_empty():
83             return None
84         else:
85             p = self.first()
86             return (p.key(), p.value())
87
88     def find_ge(self, k):
89         """Return (key,value) pair with least key greater than or equal to k.
90
91         Return None if there does not exist such a key.
92         """
93         if self.is_empty():
94             return None
95         else:
96             p = self.find_position(k)          # may not find exact match
97             if p.key() < k:                  # p's key is too small
98                 p = self.after(p)
99             return (p.key(), p.value()) if p is not None else None
100
101    def find_range(self, start, stop):
102        """Iterate all (key,value) pairs such that start <= key < stop.
103
104        If start is None, iteration begins with minimum key of map.
105        If stop is None, iteration continues through the maximum key of map.
106        """
107        if not self.is_empty():
108            if start is None:
109                p = self.first()

```

```
110 else:  
111     # we initialize p with logic similar to find_ge  
112     p = self.find_position(start)  
113     if p.key( ) < start:  
114         p = self.after(p)  
115     while p is not None and (stop is None or p.key( ) < stop):  
116         yield (p.key(), p.value( ))  
117         p = self.after(p)
```

代码段 11-7 TreeMap 类中访问和插入节点的映射操作。反向迭代可以使用与 `_iter_` 对称的方法 `_reverse_` 实现

```

118 def __getitem__(self, k):
119     """Return value associated with key k (raise KeyError if not found)."""
120     if self.is_empty():
121         raise KeyError('Key Error: ' + repr(k))
122     else:
123         p = self._subtree_search(self.root(), k)
124         self._rebalance_access(p)           # hook for balanced tree subclasses
125         if k != p.key():
126             raise KeyError('Key Error: ' + repr(k))
127         return p.value()
128
129 def __setitem__(self, k, v):
130     """Assign value v to key k, overwriting existing value if present."""
131     if self.is_empty():
132         leaf = self._add_root(self._Item(k,v))           # from LinkedBinaryTree
133     else:
134         p = self._subtree_search(self.root(), k)
135         if p.key() == k:
136             p.element()._value = v                         # replace existing item's value
137             self._rebalance_access(p)                      # hook for balanced tree subclasses
138             return
139         else:
140             item = self._Item(k,v)
141             if p.key() < k:
142                 leaf = self._add_right(p, item) # inherited from LinkedBinaryTree
143             else:
144                 leaf = self._add_left(p, item) # inherited from LinkedBinaryTree
145             self._rebalance_insert(leaf)      # hook for balanced tree subclasses
146
147 def __iter__(self):
148     """Generate an iteration of all keys in the map in order."""
149     p = self.first()
150     while p is not None:
151         yield p.key()
152         p = self.after(p)

```

代码段 11-8 利用 TreeMap 类删除节点，通过位置或者键进行定位

```

153 def delete(self, p):
154     """ Remove the item at given Position."""
155     self._validate(p) # inherited from LinkedBinaryTree
156     if self.left(p) and self.right(p): # p has two children
157         replacement = self._subtree_last_position(self.left(p))
158         self._replace(p, replacement.element()) # from LinkedBinaryTree
159         p = replacement
160     # now p has at most one child
161     parent = self.parent(p)
162     self._delete(p) # inherited from LinkedBinaryTree

```

```

163     self._rebalance_delete(parent)      # if root deleted, parent is None
164
165 def __delitem__(self, k):
166     """Remove item associated with key k (raise KeyError if not found)."""
167     if not self.is_empty():
168         p = self._subtree_search(self.root(), k)
169         if k == p.key():
170             self.delete(p)          # rely on positional version
171             return                 # successful deletion complete
172         self._rebalance_access(p)    # hook for balanced tree subclasses
173     raise KeyError('Key Error: ' + repr(k))

```

TreeMap 类利用了代码重用的多重继承的优势：继承 8.3.1 节的 LinkedBinaryTree 类作为二叉树的再现，并且 10.1.4 节的代码段 10-2 中的 MapBase 类提供了键 - 值的复合项以及 collections 模块中的具体行为。MutableMapping 对基类进行抽象。对于映射，继承嵌套的 Position 类以支持更具体的 p.key() 和 p.value() 访问，而不是从树 ADT 继承 p.element() 语法。

我们定义几个非公开的公用程序，最显著的是 \_subtree\_search(p, k) 方法，它相当于代码段 11-2 中的 TreeSearch 算法。该方法返回一个位置，理想的返回位置要么是包含键 k 的位置，要么是搜索路径上访问的最后一个位置。我们的依据是不成功的搜索过程中最后的位置或者是比 k 小的最近的键或者是比 k 大的最近的键。该搜索方法成了公共的 find\_position(k) 方法的基础，也成了在映射中搜索、插入或删除节点时的内部使用的基础，同时也成了有序映射 ADT 的强大搜索的基础。

当对树进行结构修改时，我们依靠非公开的更新方法（如 \_add\_right），其继承于 LinkedBinaryTree 类（见 8.3.1 节）。这些继承的方法保持非公开很重要，因为通过这种操作的误操作可能违背搜索树的属性。

最后，我们注意到，代码充斥着名为 \_rebalance\_insert、\_rebalance\_delete 和 \_rebalance\_access 的推测方法的调用。这些方法作为以后平衡搜索树时的钩子函数使用；（见 11.2 节）。我们将给出相关代码的概览。

- 代码段 11-4：以 TreeMap 类开始，该类包括重定义的 Position 类和非公共的搜索实用程序。
- 代码段 11-5：有关位置类的函数 first()、last()、before(p)、after(p) 和 find position(p) 的访问。
- 代码段 11-6：有序映射 ADT 的一些方法，即 find min()、find ge(k) 和 find range(start, stop)。为了简洁起见，省略了相关方法。
- 代码段 11-7：\_\_getitem\_\_(k)、\_\_setitem\_\_(k, v) 和 \_\_iter\_\_()
- 代码段 118：通过位置删除的函数 delete(p)；通过键值删除函数 \_\_delitem\_\_(k)。

### 11.1.5 二叉搜索树的性能

表 11-1 中给出了 TreeMap 类的操作的分析。几乎所有操作都有一个最坏的运行时间，它取决于树的高度  $h$ 。这是因为大多数操作依赖于一个常数数量的操作，每个节点的操作沿着树的特定路径，且最大的路径长度与树的高度成正比。最值得注意的是，与映射相关的操作 \_\_getitem\_\_、\_\_setitem\_\_ 和 \_\_delitem\_\_，都是从树的根节点开始调用 \_subtree\_search

方法向下搜索，在每个节点上使用  $O(1)$  的时间来决定如何继续搜索。在删除时寻找一个替代位置，或者计算一个位置的前驱或者后继时都有类似的路径。我们注意到，虽然 `after` 方法的单个调用最糟糕的时间复杂度是  $O(h)$ ， $n$  次连续调用 `__iter__` 需要  $O(n)$  的时间，因为每个边最多被追踪两次；在某种意义上，这些调用有  $O(1)$  的摊销时间界限。类似的参数可以用来证明调用 `find_range` 方法找到  $s$  个结果的最坏的时间复杂度是  $O(s + h)$ （见练习 C-11.34）。

表 11-1 TreeMap T 的操作的最坏时间复杂度。用  $h$  表示当前树的高度，用  $s$  表示 `find_range` 函数的节点数量。空间使用度是  $O(n)$ ，其中  $n$  是映射的节点数量

操作	运行时间
<code>k in T</code>	$O(h)$
<code>T[k], T[k] = v</code>	$O(h)$
<code>T.delete(p), del T[k]</code>	$O(h)$
<code>T.find_position(k)</code>	$O(h)$
<code>T.first(), T.last(), T.find_min(), T.find_max()</code>	$O(h)$
<code>T.before(p), T.after(p)</code>	$O(h)$
<code>T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)</code>	$O(h)$
<code>T.find_range(start, stop)</code>	$O(s + h)$
<code>iter(T), reversed(T)</code>	$O(n)$

只有在树的高度比较小的情况下，二叉搜索树  $T$  才是实现有  $n$  个实体的映射的高效算法。在最好的情况下，树  $T$  的高度  $h = \lceil \log(n + 1) \rceil - 1$ ，这对所有映射都能产生对数的时间性能。然而在最坏的情况下， $T$  的高度为  $n$ ，在这种情况下，它会像映射的一个有序列表的实现。如果根据键值的升序或者降序插入节点，最坏的情况可能会发生（见图 11-7）。

不过，值得欣慰的是，通常来说，通过一系列随机的插入或删除键操作，生成有  $n$  个键的二叉搜索树的期望复杂度是  $O(\log n)$ 。这个定理的证明超出了本书的范围，需要用数学语言精确地定义一系列随机的插入和删除的过程，并且要使用复杂的概率理论知识才能得到证明。

在一个不能保证更新的随机特性的应用程序中，最好依靠本章剩余部分提到的搜索树的改进，可以保证最坏情况下高度为  $O(\log n)$ ，因此最坏情况下，搜索、插入和删除的时间复杂度也是  $O(\log n)$ 。

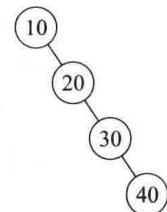


图 11-7 线性二叉搜索树的例子，根据键值的升序插入节点

## 11.2 平衡搜索树

在前一节的结尾处，我们注意到，如果假设有一系列随机的插入和删除操作，标准二叉搜索树基本映射操作的运行时间是  $O(\log n)$ 。然而，我们可能只能声称最坏的情况为  $O(n)$ ，因为一些操作序列可能导致一棵高度与  $n$  成正比的不平衡树。

在本章的其余部分，我们探讨 4 种能提供更强性能保证的搜索树算法。其中 3 种数据结构（AVL 树、伸展树和红黑树）是基于用少量操作对标准二叉搜索树进行扩展去重新调整树并降低树的高度。

平衡二叉搜索树的主要操作是旋转。在旋转中，我们“旋转”大于其父亲节点的孩子节点，如图 11-8 所示。

通过一个旋转来保持二叉搜索树的属性，我们注意到，在旋转之前，如果位置  $x$  是  $y$  位置的左子树（因此  $x$  的键小于  $y$  的键），旋转之后， $y$  成为  $x$  的右孩子，反之亦然。此外，我们必须重新利用被旋转的两个位置之间的键连接子树节点。举个例子，在图 11-8 标记为  $T_2$  的子树表示具有比  $x$  位置的键小，比  $y$  位置的键大的键的节点。在图中第一次配置时， $T_2$  是  $x$  位置的右子树；在第二次配置时，它是位置  $y$  的左子树。

因为单个旋转修改了常数数量的父子关系，在一个二叉树中实现它用  $O(1)$  时间。

在 tree-balancing 算法情况下，旋转允许修改树的形状同时并保持搜索树的性质。如果使用得当，这样的操作可以避免非常不平衡的树结构。例如，图 11-8 中第一个向右旋转到第二个向右旋转使子树  $T_1$  中的每个节点的深度减少了 1，同时使子树  $T_3$  的每个节点的深度增加了 1（注意， $T_2$  子树的节点的深度没有受旋转的影响）。

在一棵树内部，可以将一个或多个旋转合并来提供更广泛的平衡。这样的复合操作，我们称之为 trinode 重组。对于这个操作，我们考虑一个位置  $x$ ，其父亲节点为  $y$ ，其祖父节点为  $z$ 。目标是重建以  $z$  为根的子树，以缩短到  $x$  位置和其子树的总体路径长度。代码段 11-9 和图 11-9 分别给出了 restructure( $x$ ) 函数的伪代码和示意图。在描述重建平衡树的过程中，我们暂时命名位置  $x$ 、 $y$  及  $z$  分别为  $a$ 、 $b$  和  $c$ 。因此在  $T$  的中序遍历中， $a$  先于  $b$  并且  $b$  先于  $c$ 。如图 11-9 所示，有 4 种可能的方向来映射  $x$ 、 $y$ 、 $z$  到  $a$ 、 $b$ 、 $c$ 。旋转重建用  $b$  表示的节点来替换  $z$  节点，使得该节点的孩子是  $a$  和  $c$ ，并使  $a$  和  $c$  的孩子节点是  $x$ 、 $y$  和  $z$ （除了  $x$  和  $y$ ）先前的 4 个孩子节点，同时保持了  $T$  中所有节点的中序次序关系。

#### 代码段 11-9 二叉搜索树的重构操作

**Algorithm** restructure( $x$ ):

**Input:** A position  $x$  of a binary search tree  $T$  that has both a parent  $y$  and a grandparent  $z$

**Output:** Tree  $T$  after a trinode restructuring (which corresponds to a single or double rotation) involving positions  $x$ ,  $y$ , and  $z$

- 1: Let  $(a, b, c)$  be a left-to-right (inorder) listing of the positions  $x$ ,  $y$ , and  $z$ , and let  $(T_1, T_2, T_3, T_4)$  be a left-to-right (inorder) listing of the four subtrees of  $x$ ,  $y$ , and  $z$  not rooted at  $x$ ,  $y$ , or  $z$ .
- 2: Replace the subtree rooted at  $z$  with a new subtree rooted at  $b$ .
- 3: Let  $a$  be the left child of  $b$  and let  $T_1$  and  $T_2$  be the left and right subtrees of  $a$ , respectively.

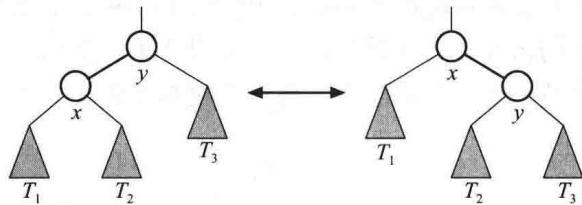


图 11-8 二叉搜索树的旋转操作。可以从左到右执行一个旋转，或者从右到左执行一个旋转。注意，在子树  $T_1$  中，所有键都比  $x$  位置的键小；在子树  $T_2$  中，所有键的大小都在  $x$  位置和  $y$  位置的键的大小之间；在子树  $T_3$  中，所有键比  $y$  位置的键大

- 4: Let  $c$  be the right child of  $b$  and let  $T_3$  and  $T_4$  be the left and right subtrees of  $c$ , respectively.

在实践中，由旋转重建操作造成的树  $T$  的修改可以通过单个旋转（见图 11-9a 和图 11-9b）或者双旋转（见图 11-9c 和图 11-9d）的案例分析来实现。当位置  $x$  是 3 个相关联的键的中间值并且首先被旋转到其父母之上，然后又被旋转于原来的祖父母之上的情况下会产生双旋转。在任何情况下，旋转重建都可以在  $O(1)$  时间内完成。

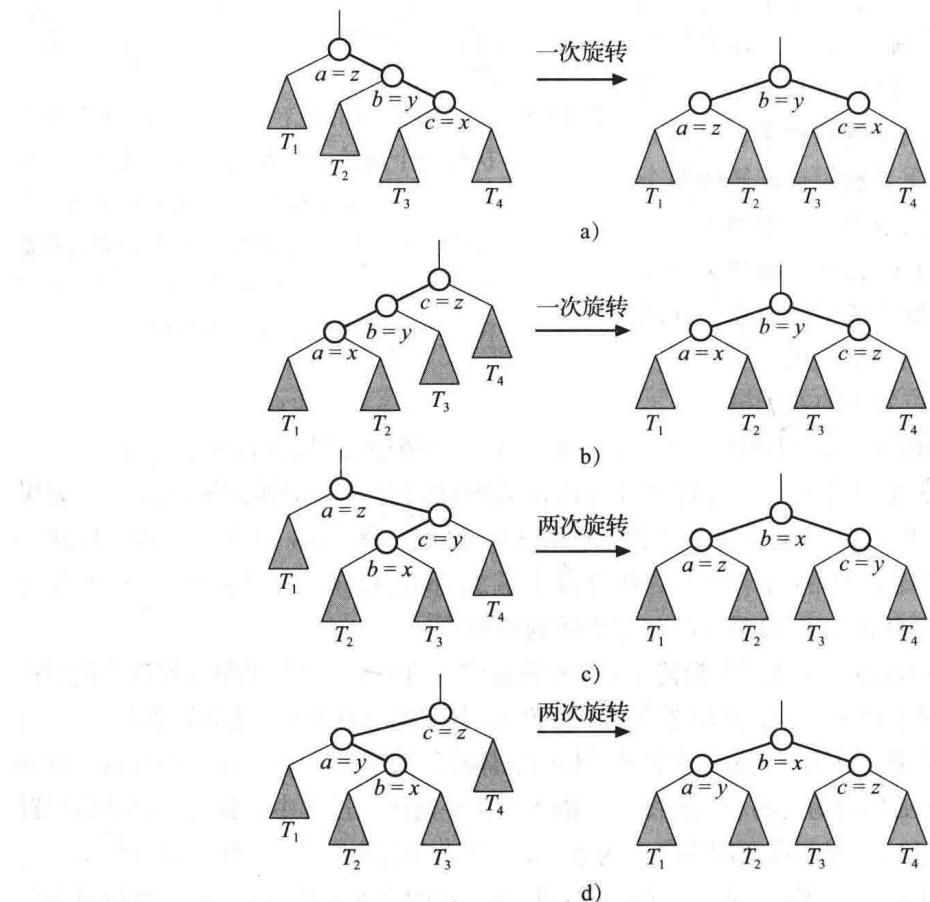


图 11-9 旋转重建操作的示意图：a) 和 b) 需要一次旋转，c) 和 d) 需要两次旋转

## 平衡搜索树的 Python 框架

我们在 11.1.4 节介绍了 TreeMap 类，它是一个具体的映射实现，不执行任何显式的平衡操作。然而，我们设计了一个类作为基类以便实现更高级的 tree-balancing 算法的子类。继承层次结构的总结如图 11-10 所示。

### 平衡操作的钩子

11.1.4 节的基本映射操作实现部分主要包括调用 3 个非公开方法作为平衡

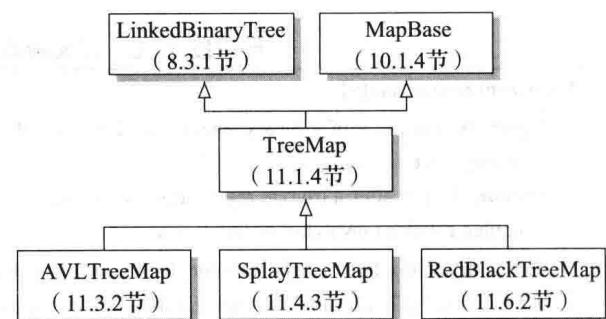


图 11-10 平衡搜索树的层次（引用平衡搜索树的定义）。  
回想一下，TreeMap 继承了 LinkedBinaryTree 和 MapBase 类

算法的钩子：

- 在位置  $p$  添加新节点后立即执行 `__setitem__` 方法，`__setitem__` 方法内部会调用 `_rebalance_insert(p)`。
- 每次一个节点从树中删除时调用 `_rebalance_delete(p)`，位置  $p$  的父节点已经确认被移除了。在形式上，这个钩子从内部被称为公共的 `delete(p)` 方法，它是间接调用公共方法 `__delitem__(k)` 的方法。
- 我们还提供一个钩子 `_rebalance_access(p)`，当使用如 `__getitem__` 等公共方法访问树中位置  $p$  的节点时被调用。伸展树结构（见 11.4 节）使用这个钩子重建一棵树，使得接近于根的节点被更频繁地访问。

我们在代码段 11-10 中提供了这 3 种方法的简单声明——只有函数名，没有函数体（使用 `pass` 语句）。`TreeMap` 的一个子类可能会重写这些方法来实现一个重要的方法来平衡树。这是模板方法设计模式的另一个例子，和 8.4.6 节中描述的类似。

代码段 11-10 `TreeMap` 类的附加代码（接代码段 11-8），提供平衡挂钩的存根

---

```
174 def _rebalance_insert(self, p): pass
175 def _rebalance_delete(self, p): pass
176 def _rebalance_access(self, p): pass
```

---

### 旋转和重组的非公开方法

第二种支持平衡搜索树的形式是非公开的 `_rotate` 和 `_restructure` 方法，它们分别实现单一旋转和 `trinode` 重组（在 11.2 节开头描述）。尽管这些方法并不被公开的 `TreeMap` 操作调用，但是我们通过在这个类中提供这些实现来让它们被所有平衡树的子类继承，从而促进代码重用。

实现在代码段 11-11 中给出。为了简化代码，我们定义一个额外的 `_relink` 实用方法，用以正确关联父亲和孩子节点，包括没有孩子节点的特殊情况。`_rotate` 方法的焦点就变成了重新定义父亲和孩子之间的联系，直接将旋转节点和原来的祖父母进行关联，然后在旋转节点中移除“中间”子树（在图 11-8 中用  $T_2$  表示）。对于 `trinode` 重组，我们决定执行是否单个旋转还是双旋转，如图 11-9 所示的那样。

代码段 11-11 `TreeMap` 类的附加代码（接代码段 11-10），为平衡搜索树的子类提供非公开的实用程序

---

```
177 def _relink(self, parent, child, make_left_child):
178     """Relink parent node with child node (we allow child to be None)."""
179     if make_left_child:
180         parent._left = child
181     else:
182         parent._right = child
183     if child is not None:
184         child._parent = parent
185
186 def _rotate(self, p):
187     """Rotate Position p above its parent."""
188     x = p._node
189     y = x._parent
190     z = y._parent
191     if z is None:
192         self._root = x
193         x._parent = None
194     else:
195         self._relink(z, x, y == z._left)
```

---

```

196     # now rotate x and y, including transfer of middle subtree
197     if x == y._left:
198         self._relink(y, x._right, True)           # x._right becomes left child of y
199         self._relink(x, y, False)                 # y becomes right child of x
200     else:
201         self._relink(y, x._left, False)          # x._left becomes right child of y
202         self._relink(x, y, True)                 # y becomes left child of x
203
204     def _restructure(self, x):
205         """Perform tri-node restructure of Position x with parent/grandparent."""
206         y = self.parent(x)
207         z = self.parent(y)
208         if (x == self.right(y)) == (y == self.right(z)): # matching alignments
209             self._rotate(y)                           # single rotation (of y)
210             return y                               # y is new subtree root
211         else:                                     # opposite alignments
212             self._rotate(x)                         # double rotation (of x)
213             self._rotate(x)
214             return x                             # x is new subtree root

```

### 创建树节点工厂

在设计 TreeMap 类和原始的 LinkedBinaryTree 子类时，我们注意到一个重要的微妙细节。LinkedBinaryTree 类的内嵌套类 \_Node 类提供节点的底层定义。然而，我们的几个树平衡策略要求辅助信息被存储在每个节点来指导平衡过程。这些类将会重写嵌套类 \_Node 类来为一个额外的字段提供存储。

每当将新节点添加到树中时，在 LinkedBinaryTree（最初在代码段 8-10 中给定）的 \_add\_right 方法中，我们特意使用语法 self.\_Node 实例化节点，而不是限定名称 LinkedBinaryTree.Node。这对框架很重要！当表达式 self.\_Node 是应用于一个（子）树的类的一个实例时，Python 的名称解析遵循继承结构（如 2.5.2 节中所述）。如果一个子类重写 \_Node 类的定义，self.\_Node 实例化时将使用新定义的节点类。这种技术是工厂方法设计模式的一个例子，我们提供了一个子类的方法控制节点的类型，它是在父类的方法内创建的。

## 11.3 AVL 树

使用标准二叉搜索树作为数据结构的 TreeMap 类，应该是一种有效的映射数据结构，但对于各种操作其最糟糕的表现是线性的时间，因为有可能是一系列的操作结果产生了具有线性高度的树。在本节中，我们描述一种简单的平衡策略，可保证对所有基本的映射操作来说最坏情况下是对数的运行时间。

### AVL 树的定义

对二叉搜索树的定义简单地进行修正是添加一条规则：对树维持对数的高度。虽然我们最初定义以位置  $p$  为根的子树的高度为从根节点位置  $p$  到叶子节点的最长路径的边的数量（见 8.1.3 节），但是本节考虑在最长路径上节点的数量作为树的高度更容易理解。根据这个定义，一片叶子位置高度为 1，我们定义“null”孩子的高度是 0。

在本节中，我们考虑下面的高度平衡属性，就其节点的高度而言，这体现了二叉搜索树  $T$  的结构。

**高度平衡属性：**对于  $T$  中每一个位置  $p$ ， $p$  的孩子的高度最多相差 1。

任何满足高度平衡属性的二叉搜索树  $T$  被称为 AVL 树，以发明家的名字的首字母命名：

Adel'son-Vel'skii 和 Landis。AVL 树的一个例子如图 11-11 所示。

高度平衡所带来的一个直接结果是 AVL 树子树本身就是一棵 AVL 树。高度平衡属性也带来同样一个重要的结果，即可以保持高度最小，如下面的命题。

**命题 11-2：**一棵存有  $n$  个节点的 AVL 树的高度是  $O(\log n)$ 。

**证明：**我们不是试图直接找到一个 AVL 树的高度的上限，而更容易找到一个“反问题”，即找到一个高度为  $h$  的树的最小节点数  $n(h)$  的下界。我们将证明  $n(h)$  至少成指数增长。由此，很容易得到存有  $n$  个节点 AVL 树的高度是  $O(\log n)$ 。

首先指出  $n(1) = 1$  和  $n(2) = 2$ ，因为一棵高度为 1 的 AVL 树必须只有一个节点且一棵高度为 2 的 AVL 树必须至少有两个节点。现在，一棵高度为  $h$  的 AVL 树的最小节点数为  $h \geq 3$ ，这样两棵子树都是具有最小节点数的 AVL 树：一棵高度  $h-1$ ，另一棵高度为  $h-2$ 。从根开始计算，我们得到以下  $n(h)$  与  $n(h-1)$  和  $n(h-2)$  关系的公式，其中  $h \geq 3$ ：

$$n(h) = 1 + n(h-1) + n(h-2) \quad (11-1)$$

在这一点上，熟悉斐波那契数列的性质（1.8 节和练习 C-3.49）的读者已经知道  $n(h)$  是一个关于  $h$  的指数函数。为了形式化这一观察，我们进行如下操作。

式 (11-1) 意味着  $n(h)$  是关于  $h$  的严格递增函数。因此，我们知道  $n(h-1) > n(h-2)$ 。在式 (11-1) 中用  $n(h-2)$  替代  $n(h-1)$  并且舍弃 1，我们得到  $h \geq 3$  时，

$$n(h) > 2n(h-2) \quad (11-2)$$

式 (11-2) 表明每次  $h$  增加 2 时， $n(h)$  至少增加一倍，这意味着  $n(h)$  会成指数增长。为了用一个正式的方式展示这一事实，我们重复应用式 (11-2)，产生以下一系列不等式：

$$n(h) > 2n(h-2) > 4n(h-4) > 8n(h-6) \dots > 2^i n(h-2i) \quad (11-3)$$

也就是说，对于任何整数  $i$ ，有  $n(h) > 2^i n(h-2i)$ ，因此  $h-2i \geq 1$ 。因为已经知道  $n(1)$  的值和  $n(2)$  的值，所以选择使得  $h-2i$  等于 1 或 2 的  $i$ 。也就是说，选择：

$$i = \left\lceil \frac{h}{2} \right\rceil - 1$$

将上面  $i$  的值代入式 (11-3) 中，得到，对于  $h \geq 3$ ：

$$n(h) > 2^{\left\lceil \frac{h}{2} \right\rceil - 1} \cdot n\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \geq 2^{\left\lceil \frac{h}{2} \right\rceil - 1} \cdot n(1) \geq 2^{\frac{h}{2} - 1} \quad (11-4)$$

通过对式 (11-4) 两边取对数，得到：

$$\log(n(h)) > \frac{h}{2} - 1$$

进而得到：

$$h < 2\log(n(h)) + 2$$

说明了存有  $n$  个节点的 AVL 树的高度最大为  $2\log n + 2$ 。 ■

由命题 11-2 和 11.1 节中给出的二叉搜索树的分析，针对 `__getitem__` 操作，映射用 AVL 树实现，运行时间为  $O(\log n)$ ，其中  $n$  是映射中项的数量。当然，我们仍然需要展示在

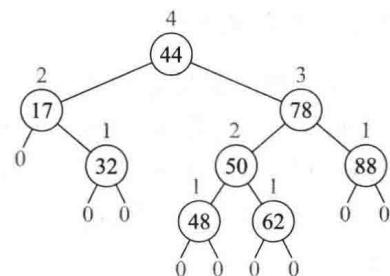


图 11-11 AVL 树的一个例子，项的键显示在节点里，节点的高度显示在节点上面（空子树的高度为 0）

插入或者删除之后如何保持高度平衡属性。

### 11.3.1 更新操作

给定一棵二叉搜索树  $T$ ，如果孩子之间的高度差的绝对值最多为 1，则该位置是平衡的，否则我们说它是不平衡的。因此，AVL 树的高度平衡属性相当于每个位置都是平衡的。

AVL 树的插入和删除操作开始类似于相应的（标准）二叉搜索树的操作，但因为受到改变的不利影响对每一部分恢复平衡所进行操作的后期处理是不一样的。

#### 插入

假设在插入一个新项目之前，树  $T$  满足高度平衡属性，则树  $T$  是一棵 AVL 树。在一棵二叉搜索树中插入新节点，如 11.1.3 节所述，在叶子节点  $p$  的位置产生了一个新节点。这个操作可能违反了高度平衡属性（见图 11-12a），然而，唯一可能会变得不平衡的位置是  $p$  的祖先，因为那些位置是其子树唯一变化过的位置。因此，我们接下来描述如何重建  $T$ ，以解决任何可能发生的不平衡。

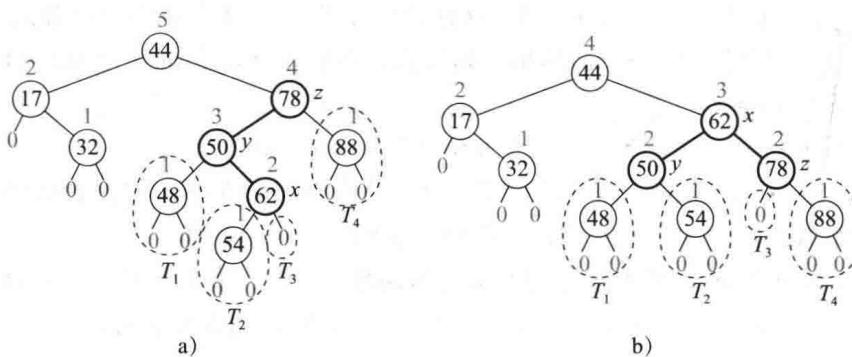


图 11-12 图 11-1 的一个例子：在 AVL 树中插入键为 54 的项：a) 加入键为 54 的新节点后，键为 78 和 44 的节点变得不平衡；b) 高度平衡属性的重构。把节点的高度写在了上面，在重构操作过程中定义节点  $x$ 、 $y$ 、 $z$  和子树  $T_1$ 、 $T_2$ 、 $T_3$  和  $T_4$

我们通过一个简单的“查找和修复”策略来恢复二叉搜索树中节点的平衡。特别是，用  $z$  是从  $p$  到根  $T$  的方向中遇到的第一个位置，因此  $z$  是不平衡的（见图 11-12a）。同样，用  $y$  表示  $z$  的具有更高高度的孩子（注意， $y$  必须是  $p$  的一个祖先）。最后，假设  $x$  是  $y$  具有更高高度的孩子（不能有并列，并且  $x$  也必须是  $p$  的一个祖先或者  $p$  自身）。我们通过调用 trinode 重组方法 `restructure(x)`（最初在 11.2 节中描述的）对以  $z$  为根的子树进行再平衡。图 11-12 描述了这样一个 AVL 树插入重组的例子。

为了正式证明这个过程在重建 AVL 高度平衡属性时的正确性，我们考虑  $z$  是插入  $p$  之后变得不平衡的最近的  $p$  的祖先。 $y$  的高度由于插入增加了 1，并且现在比它的兄弟节点大 2。因为  $y$  保持了平衡，它原来的子树必须具有相同高度，而且包含  $x$  的子树高度增加了 1。它的子树或者因为  $x = p$  高度增加了，所以它的高度从 0 变到 1，或者因为  $x$  之前有等高子树然后包含  $p$  的子树高度增加了 1。令  $h \geq 0$  表示  $x$  的最高的孩子的高度，这个场景如图 11-13 所示。

`trinode` 重组后，我们可以看到  $x$ 、 $y$ 、 $z$  都平衡了。此外，在重组之后成为子树的根的节点的高度为  $h + 2$ ，这正是  $z$  在插入新节点之前的高度。因此，任何变得暂时不平衡的  $z$  的祖先又恢复了平衡，这一重组恢复了全局的高度平衡属性。

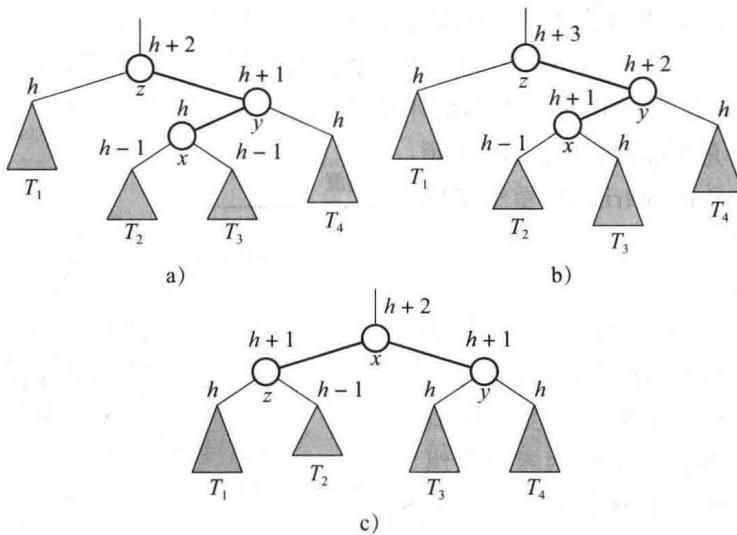


图 11-13 在对 AVL 树进行典型的插入操作期间子树的再平衡过程：a) 插入之前；b) 在子树  $T_3$  进行插入操作导致了  $z$  的不平衡；c) 用 trinode 重组进行重建平衡之后。注意，在插入操作之后，子树的总高度和插入操作之前一样

### 删除

回想一下，对一个普通二叉搜索树结构进行删除操作将导致一个节点拥有零或一个孩子。这样的改变可能违反 AVL 树的高度平衡属性。特别是，如果  $p$  代表在树  $T$  中删除节点的父节点，可能有一个不平衡的节点在  $p$  到根节点之间的路径上（见图 11-14a）。事实上，最多可以有一个这种不平衡的节点（这一事实的证明留作练习 C-11.49）。

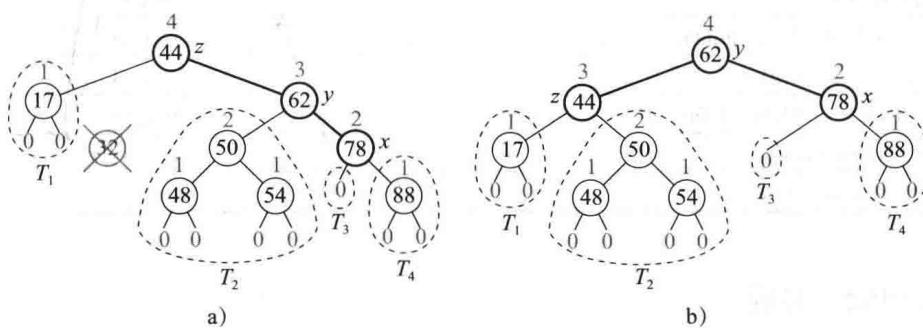


图 11-14 删除图 11-12b 中 AVL 树上键为 32 的项。a) 删除存储键为 32 的节点之后，根变得不平衡；b) 一次（单次）旋转会恢复高度平衡属性

与插入一样，我们使用 trinode 重组恢复树  $T$  的平衡。特别是，用  $z$  表示在  $T$  中从  $p$  向根的方向上遇到的第一个不平衡位置。同样，用  $y$  表示  $z$  的具有更高高度的孩子（注意， $y$  是  $z$  的孩子，但不是  $p$  的祖先），并按如下定义令  $x$  是  $y$  的孩子：如果  $y$  的一个孩子比另一个高，令  $x$  是  $y$  的较高的孩子；否则（ $y$  的两个孩子有相同的高度），令  $x$  是与  $y$  在同一边的孩子（也就是说，如果  $y$  是  $z$  的左孩子，令  $x$  为  $y$  的左孩子；否则，令  $x$  为  $y$  的右孩子）。在以上任何情况下，我们进行 `restructure(x)` 操作（见图 11-14b）。

在 trinode 重组操作过程中，重组子树是以中间位置  $b$  为根。在  $b$  的子树内局部地重建可以保证高度平衡属性（见练习 R-11.11b 和 R-11.12）。不幸的是，这种 trinode 重组可能会

使以  $b$  为根的子树的高度减少 1，这可能会导致  $b$  的祖先变得不平衡。所以，对  $z$  进行再平衡之后，我们继续在  $T$  中寻找不平衡的位置。如果找到另一个，则执行重组操作来恢复它的平衡，并且继续沿着  $T$  向上寻找更多的不平衡位置，一直到根节点。不过， $T$  的高度是  $O(\log n)$ ，其中  $n$  是项的数量，由命题 11-2 可知， $O(\log n)$  内的 trinode 重组足以恢复高度平衡属性。

### AVL 树的性能

由命题 11-2 可知，有  $n$  个节点的 AVL 树的高度是  $O(\log n)$ 。因为标准二叉搜索树的操作的运行时间受高度的限制（见表 11-1），因为在保持平衡因素和重组一棵 AVL 树的额外工作中受树中路径的长度的限制，对于 AVL 树，传统的映射操作的运行时间为最坏的对数时间。我们在表 11-2 中总结了这些结果，并在图 11-15 中举例说明了这种性能。

表 11-2 对有  $n$  个节点的 AVL 树进行操作的最坏运行时间，其中  $s$  表示由 `find_range` 报告的项的数目

操作	运行时间
$k \in T$	$O(\log n)$
$T[k] = v$	$O(\log n)$
$T.delete(p)$ , $\text{del } T[k]$	$O(\log n)$
$T.\text{find position}(k)$	$O(\log n)$
$T.\text{first}()$ , $T.\text{last}()$ , $T.\text{find min}()$ , $T.\text{find max}()$	$O(\log n)$
$T.\text{before}(p)$ , $T.\text{after}(p)$	$O(\log n)$
$T.\text{find lt}(k)$ , $T.\text{find le}(k)$ , $T.\text{find gt}(k)$ , $T.\text{find ge}(k)$	$O(\log n)$
$T.\text{find range}(start, stop)$	$O(s + \log n)$
$\text{iter}(T)$ , $\text{reversed}(T)$	$O(n)$

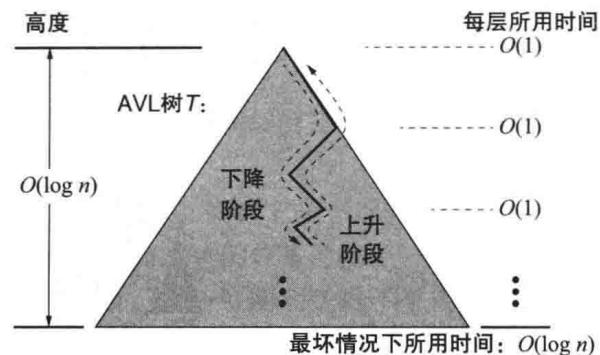


图 11-15 对 AVL 树进行搜索和更新的运行时间。每级性能是  $O(1)$ ，分为下降阶段（一般包括搜索过程）和上升阶段（一般包括更新高度值和执行局部 trinode 重组（旋转））

### 11.3.2 Python 实现

代码段 11-12 和代码段 11-13 给出了一个 `AVLTreeMap` 类的完整实现。它继承了标准 `TreeMap` 类并且依赖 11.2 节中描述的平衡框架。我们强调两个重要方面：首先，`AVLTreeMap` 重写了嵌套类 `_Node` 的定义（如代码段 11-12 所示），目的是为了将存储在一个节点的子树的高度保存起来提供支持。我们还提供了几个包含节点高度和关联位置的实用程序。

代码段 11-12 `AVLTreeMap` 类（后接代码段 11-13）

```

1 class AVLTreeMap(TreeMap):
2     """Sorted map implementation using an AVL tree."""
3
4     #----- nested _Node class -----
5     class _Node(TreeMap._Node):
6         """Node class for AVL maintains height value for balancing."""
7         __slots__ = '_height'           # additional data member to store height
8

```

```

9  def __init__(self, element, parent=None, left=None, right=None):
10 super().__init__(element, parent, left, right)
11 self._height = 0           # will be recomputed during balancing
12
13 def left_height(self):
14     return self._left._height if self._left is not None else 0
15
16 def right_height(self):
17     return self._right._height if self._right is not None else 0

```

## 代码段 11-13 AVLTreeMap 类 (接代码段 11-12)

```

18 #----- positional-based utility methods -----
19 def _recompute_height(self, p):
20     p._node._height = 1 + max(p._node.left_height(), p._node.right_height())
21
22 def _isbalanced(self, p):
23     return abs(p._node.left_height() - p._node.right_height()) <= 1
24
25 def _tall_child(self, p, favorleft=False): # parameter controls tiebreaker
26     if p._node.left_height() + (1 if favorleft else 0) > p._node.right_height():
27         return self.left(p)
28     else:
29         return self.right(p)
30
31 def _tall_grandchild(self, p):
32     child = self._tall_child(p)
33     # if child is on left, favor left grandchild; else favor right grandchild
34     alignment = (child == self.left(p))
35     return self._tall_child(child, alignment)
36
37 def _rebalance(self, p):
38     while p is not None:
39         old_height = p._node._height          # trivially 0 if new node
40         if not self._isbalanced(p):          # imbalance detected!
41             # perform trinode restructuring, setting p to resulting root,
42             # and recompute new local heights after the restructuring
43             p = self._restructure(self._tall_grandchild(p))
44             self._recompute_height(self.left(p))
45             self._recompute_height(self.right(p))
46             self._recompute_height(p)          # adjust for recent changes
47             if p._node._height == old_height: # has height changed?
48                 p = None                   # no further changes needed
49             else:
50                 p = self.parent(p)          # repeat with parent
51
52 #----- override balancing hooks -----
53 def _rebalance_insert(self, p):
54     self._rebalance(p)
55
56 def _rebalance_delete(self, p):
57     self._rebalance(p)

```

为了实现 AVL 平衡策略的核心逻辑，我们定义了一个名为 `_rebalance` 的实用程序，它可以在插入或删除之后恢复高度平衡属性时作为一个挂钩。虽然针对插入和删除继承行为有很大的不同，但是对 AVL 树的必要后期处理是一致的。在这两种情况下，我们从发生变化的位置  $p$  向上，重新根据（更新的）孩子的高度计算每个位置的高度，如果到达一个不平衡位置，就使用 trinode 重组操作。如果到达一个通过整体映射操作高度也不变的祖先，或者

执行 trinode 重组使得子树拥有和映射操作之前相同的高度，我们会停止该过程；更高层次的祖先的高度将不会改变。为了检测停止条件，我们记录每个节点的“旧”的高度，并将其和最新计算的高度进行比较。

## 11.4 伸展树

下一个学习的搜索树的结构，我们称之为伸展树。这种结构从概念上完全不同于本章中讨论的其他平衡搜索树，因为伸展树在树的高度上没有一个严格的对数上界。事实上，伸展树无须有额外的高度、平衡或与此树节点关联的其他辅助数据。

伸展树的效率取决于某一位置移动到根的操作（称为伸展），每次在插入、删除或者甚至搜索都要从最底层的位置  $p$  开始（在本质上，这是 7.6.2 节探讨的向前启发式搜索树的一个变形）。直观上讲，伸展操作会使得被频繁访问的元素更快接近于根，从而减少典型的搜索时间。关于伸展的令人惊讶的事情是，它对插入、删除、搜索操作保证了对数的运行时间。

### 11.4.1 伸展

已知二叉搜索树  $T$  的一个节点  $x$ ，我们通过一系列的重组将  $x$  移动到  $T$  的根来对  $x$  进行扩展。进行特定的重组是很重要的，因为将节点  $x$  移动到根节点  $T$  仅仅通过一些序列的重组是不够的。我们将  $x$  向上移动执行的特定操作取决于  $x$  的相对位置、其父节点  $y$  和  $x$  的祖先节点  $z$ （如果存在的话）。我们考虑如下三种情况：

**zig-zig 型：**节点  $x$  和父节点  $y$  都在树的左边或者树的右边，如图 11-16 所示。我们在保持树的节点中序的情况下移动节点  $x$ ，使  $y$  节点为  $x$  节点的一个孩子，并且使  $z$  节点为  $y$  节点的一个孩子。

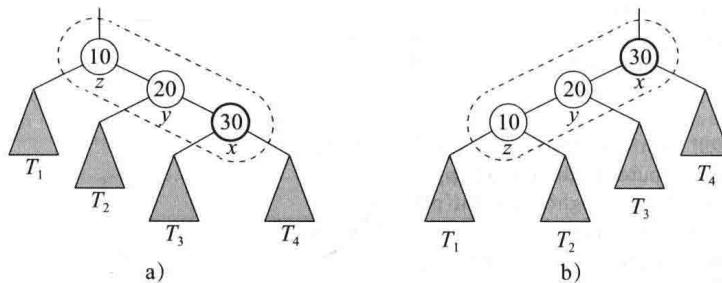


图 11-16 zig-zig 型：a) 操作前；b) 操作后。还有另一种对称的结构是节点  $x$  和  $y$  都是左孩子

**zig-zag 型：**节点  $x$  和节点  $y$  一个是左孩子，另一个是右孩子（见图 11-17）。在这种情况下，我们在保持树的节点中序的情况下移动节点  $x$ ，使其拥有孩子节点  $y$  和  $z$ 。

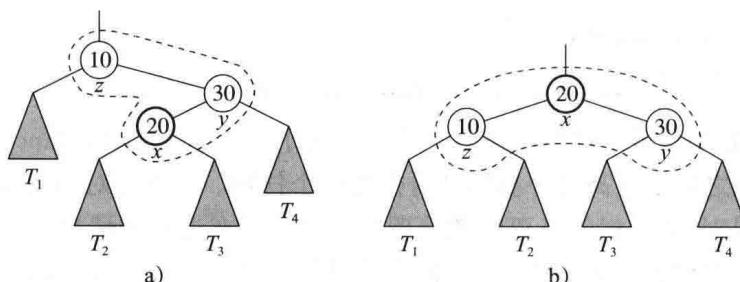


图 11-17 zig-zag 型：a) 操作前；b) 操作后。还有另一种对称的结构是节点  $x$  为右孩子，而  $y$  为左孩子

**zig型:**  $x$  没有祖父节点(见图 11-18)。在这种情况下,我们在保持树节点中序的情况下进行单次旋转,将 $x$ 提升到 $y$ 之上,使得节点 $y$ 为节点 $x$ 的孩子节点。

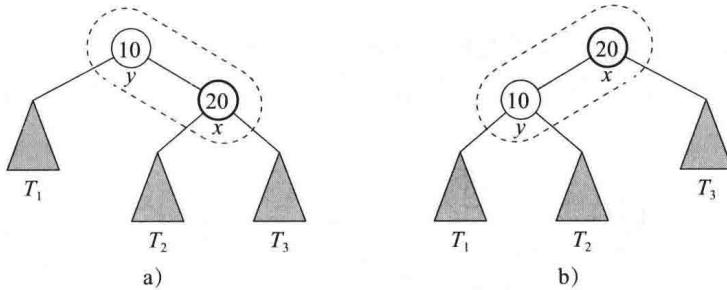


图 11-18 zig 型: a) 操作前; b) 操作后。还有另一种对称的结构是节点 $x$ 为节点 $y$ 的左孩子

可以发现,当节点 $x$ 有一个祖父节点时,可以执行 zig-zig 型或 zig-zag 型,当节点 $x$ 没有祖先节点时可以执行 zig 型,我们通过对节点 $x$ 进行重复的重组进行伸展,直到节点 $x$ 变为伸展树的根节点。伸展的一个节点例子如图 11-19 和图 11-20 所示。

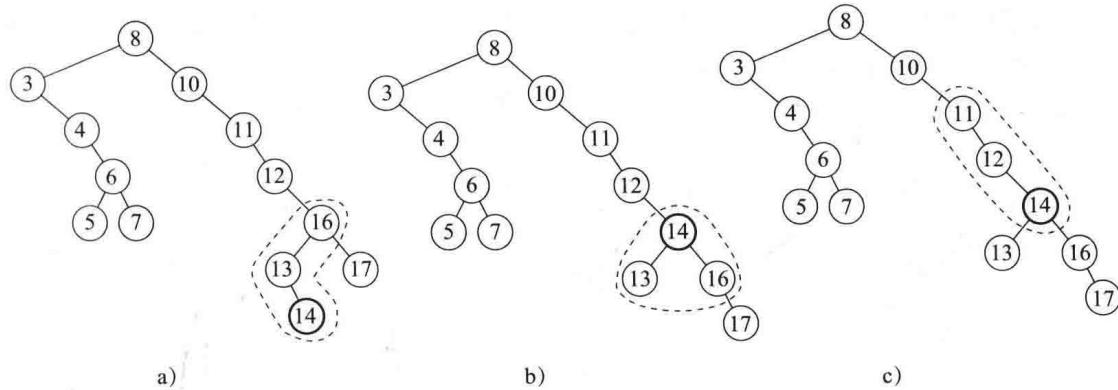


图 11-19 伸展一个节点的例子: a) 从节点 14 开始用 zig-zag 型; b) 使用 zig-zag 型旋转后; c) 下一步将使用 zig-zig 型 (后接图 11-20)

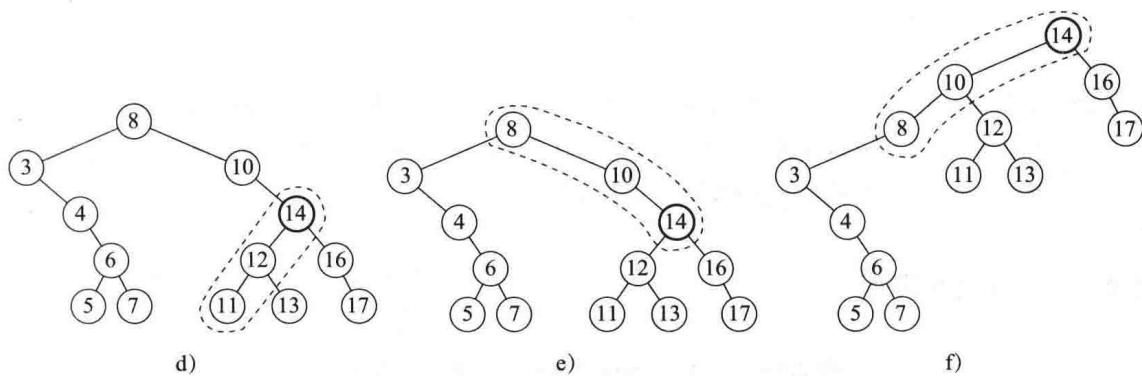


图 11-20 伸展一个节点的例子: d) 使用 zig-zig 型伸展后; e) 下一步又是使用 zig-zig 型; f) 使用 zig-zig 型伸展后 (接图 11-19)

#### 11.4.2 何时进行伸展

何时进行伸展的规则如下:

- 当搜索键  $k$  时，如果发现  $k$  在位置  $p$ ，则伸展  $p$ ；否则，在搜索失败的位置伸展叶子节点。例如，图 11-19 和图 11-20 分别展示了当搜索键 14 成功或者搜索键 15 失败时的伸展情况。
- 当插入键  $k$  时，我们将伸展新插入的内部节点  $k$ 。例如，图 11-19 和图 11-20 展示了如果 14 是新插入的键的情况。图 11-21 展示了在伸展树中的一系列插入操作。

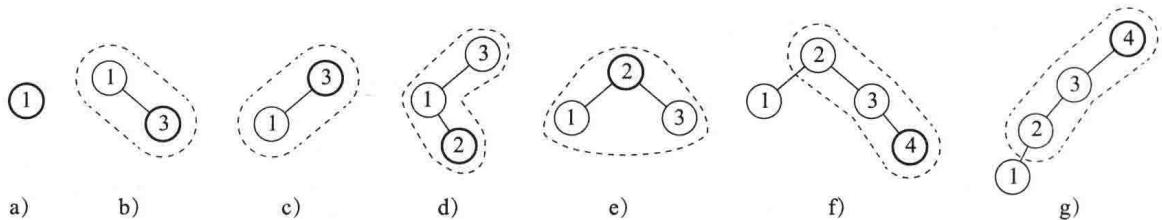


图 11-21 在伸展树中的一系列插入：a) 初始树；b) 插入 3 后，但是在 zig 型变化前；c) 伸展后；d) 插入 2 后，但是在 zig 型变化前；e) 伸展后；f) 插入 4 后，但是在 zig-zig 型变化前；g) 伸展后

- 当删除键  $k$  时，在位置  $p$  进行伸展，其中  $p$  是被移除节点的父节点；回想二叉搜索树的删除算法，删除节点可能是原来包含的节点  $k$ ，或一个有替代键的后代节点。删除节点的一个例子如图 11-22 所示。

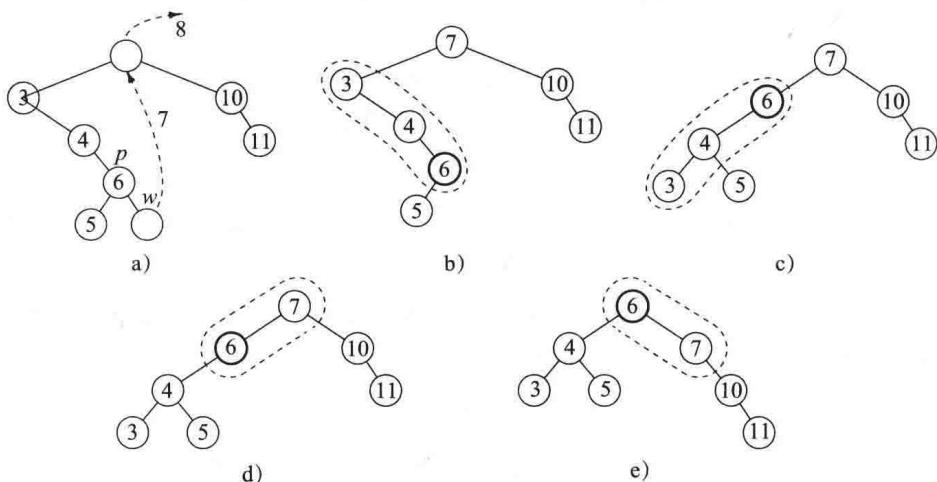


图 11-22 从伸展树中删除：a) 从根节点删除 8 是通过将中序次序的先驱  $w$  的键移动到根，删除  $w$ ，然后对  $w$  的父节点  $p$  进行伸展来实现；b) 利用 zig-zig 型伸展树的节点  $p$ ；c) zig-zig 型伸展后；d) 下一步将是 zig 型旋转；e) zig 型伸展后

### 11.4.3 Python 实现

#### 代码段 11-14 Splay TreeMap 类的完整实现

```

1 class SplayTreeMap(TreeMap):
2     """Sorted map implementation using a splay tree."""
3     #----- splay operation -----
4     def _splay(self, p):
5         while p != self.root():
6             parent = self.parent(p)
7             grand = self.parent(parent)

```

```

8   if grand is None:
9     # zig case
10    self._rotate(p)
11 elif (parent == self.left(grand)) == (p == self.left(parent)):
12   # zig-zig case
13   self._rotate(parent)           # move PARENT up
14   self._rotate(p)              # then move p up
15 else:
16   # zig-zag case
17   self._rotate(p)              # move p up
18   self._rotate(p)              # move p up again
19
20 #----- override balancing hooks -----
21 def _rebalance_insert(self, p):
22   self._splay(p)
23
24 def _rebalance_delete(self, p):
25   if p is not None:
26     self._splay(p)
27
28 def _rebalance_access(self, p):
29   self._splay(p)

```

虽然伸展树性能的数学分析是复杂的（见 11.4.4 节），但对一棵标准二叉搜索树进行伸展实现是相当简单的。代码段 11-14 基于底层的 TreeMap 类并且利用 11.2 节的平衡框架描述对 SplayTreeMap 类提供了一个完整的实现。重要的是，要注意原来的 TreeMap 类调用 `_rebalance_access` 方法，不仅在 `__getitem__` 方法内部调用 `_rebalance_access` 方法，还在修改与现有的键关联的值，并在任何导致搜索失败的映射操作之后使用 `__setitem__` 方法时调用 `_rebalance_access` 方法。

#### 11.4.4 伸展树的摊销分析 \*

经过 zig-zig 型或 zig-zag 型伸展后， $p$  节点深度减少两层；经过 zig 型伸展后， $p$  节点深度减少一层。因此，如果  $p$  节点的深度为  $d$ ，则伸展树的  $p$  节点由一系列  $\lfloor d/2 \rfloor$  的 zig-zig 型和 / 或 zig-zag 型组成，如果  $d$  是奇数，最后再加上一个 zig 型。因为一个简单的 zig-zig 型、zig-zag 型或 zig 型伸展影响一定常数数量的节点，它可以在  $O(1)$  时间完成。因此，在一棵二叉搜索树中对位置  $p$  进行伸展需要的时间为  $O(d)$ ，其中  $d$  是  $T$  树中位置  $p$  的深度。换句话说，从位置  $p$  的伸展所消耗的时间等同于从根的位置到  $p$  位置自上而下的搜索。

##### 最坏情况下的时间

在最坏情况下，因为搜索的位置可能是树上最深的位置，所以对一棵高度为  $h$  的伸展树进行搜索、插入或删除的全部运行时间是  $O(h)$ 。此外，如图 11-21 所示， $h$  最大可能接近  $n$ 。因此，从最坏情况来看，伸展树不是一个好的数据结构。

尽管在最坏情况下的性能较差，但伸展树在摊销的意义上表现良好。那是因为在一系列的混合搜索、插入和删除操作中，平均每一个操作需要的时间为对数时间。下面运用统计方法对伸展树进行摊销分析。

##### 伸展树的摊销性能

对于我们的分析，可以注意到，进行搜索、插入或删除的时间与进行伸展的开销时间成正比。所以接下来我们只考虑伸展时间。

设  $T$  是有  $n$  个节点的伸展树， $w$  是树  $T$  的一个节点，定义以  $w$  为根的子树的节点数量

大小为  $n(w)$ 。我们可以注意到这个定义意味着非叶子节点的数量是超过它的孩子节点数量的。定义节点  $w$  的阶为  $r(w)$ ,  $r(w)$  是以 2 为底的对数的结果, 即  $r(w) = \log(n(w))$ 。显然,  $T$  的根有最大的大小  $n$  以及最大阶  $\log(n)$ , 每片叶子的大小是 1 且阶为 0。

用 cyber-dollars 来表示在树  $T$  中伸展一个位置  $p$  的花费, 假设一个 zig 型伸展需要一个 cyber-dollar, zig-zig 型或 zig-zag 型则需要两个 cyber-dollar。因此, 在深度为  $d$  的位置  $p$  的花费是  $d$  cyber-dollars。我们在  $T$  树中每个位置保留一个虚拟的账户存储 cyber-dollar。注意, 这个账户只在进行摊销分析的时候存在, 而不包含在实现一个伸展树的数据结构中。

### 进行伸展时的统计分析

进行伸展时, 我们需要一定数量的 cyber-dollars (具体的开销将由最终的分析决定)。将分为三种情况:

- 如果开销等于伸展工作, 我们用全部的 cyber-dollar 来支付伸展。
- 如果开销大于伸展工作, 我们把多出的 cyber-dollar 存入几个节点的账户。
- 如果开销小于伸展工作, 我们从几个节点的账户取款, 以补偿不足之处。

下面证明每次操作  $O(\log(n))$  cyber-dollars 足够保持系统的工作, 即确保每个节点保持非负账户余额。

### 不变账户的伸展树

在需要向外伸展的工作时, 我们使用一个计划转移账户之间的节点以确保总是会有足够的 cyber-dollars 支付伸展工作。

为了使用会计方法来执行分析, 我们保持下列引理不变:

在伸展之前和之后,  $T$  中每个节点的  $w$  在它的账户中有  $r(w)$  cyber-dollars。

请注意, 不变的是“财政稳健”, 因为它不需要我们做一个初步存款来赋予一棵树。

令  $r(T)$  是  $T$  中所有节点的阶的总和。为了保持在伸展之后不变, 我们必须使支付等于伸展工作加上  $r(T)$  的总和。我们指的是以单个的 zig、zig-zig 或者 zig-zag 操作作为一个子步骤。此外, 我们分别表示一个节点前和之后的一个节点的阶为  $r(w)$  和  $r'(w)$ 。以下命题给出了一个  $r(T)$  由于单个伸展子步骤造成的上限。我们会反复在从一个节点到根的全伸展的分析中使用这个引理。

**命题 11-3:** 对于  $T$  中的节点  $x$ , 令  $\delta$  是由于单个伸展子步骤 (一个 zig、zig-zig 或者 zig-zag) 造成的  $r(T)$  的变化。我们有以下:

- $\delta \leq 3(r'(x) - r(x)) - 2$ , 如果子步骤是 zig-zig 或者 zig-zag。
- $\delta \leq 3(r'(x) - r(x))$ , 如果子步骤是 zig。

**证明:** 使用如下事实 (参见命题 B-1, 附录 A), 即如果  $a > 0$ ,  $b > 0$  并且  $c > b + a$ ,

$$\log a + \log b < 2 \log c - 2 \quad (11-6)$$

考虑每种类型的向外伸展的子步骤造成的  $r(T)$  的变化。

**zig-zig:** 如图 11-16 所示, 由于每个节点的大小是比它的两个孩子大 1 或者 2, 注意, 在单次 zig-zig 操作中只有  $x$ 、 $y$ 、 $z$  的阶变化,  $y$  是  $x$  的父节点,  $z$  也是  $y$  的父节点。而且  $r'(x) = r(z)$ ,  $r'(y) \leq r(x)$ , 并且  $r(x) \leq r(y)$ 。所以,

$$\begin{aligned} \delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &= r'(y) + r'(z) - r(x) - r(y) \leq r'(x) + r'(z) - 2r(x) \end{aligned} \quad (11-7)$$

注意,  $n(x) + n'(z) < n'(x)$ 。所以  $r'(x) + r'(z) < 2r'(x) - 2$ , 就像式 (11-6):

$$r'(z) < 2r'(x) - r(x) - 2$$

这个不等式和式(11-7)可以简写为:

$$\delta \leq r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \leq 3(r'(x) - r(x)) - 2$$

**zig-zag**: 如图11-17所示,一开始,定义大小和阶,仅仅是x、y、z的阶改变。y为x的父节点,z是y的父节点,且 $r(x) < r(y) < r(z) = r'(x)$ 。因此:

$$\delta = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) = r'(y) + r'(z) - r(x) - r(y) \leq r'(y) + r'(z) - 2r(x)$$

注意 $n'(y) + n'(z) < n'(x)$ ,因此 $r'(y) + r'(z) < 2r'(x) - 2$ ,例如式(11-6)。因此,

$$\delta \leq 2r'(x) - 2 - 2r(x) = 2(r'(x) - r(x)) - 2 \leq 3(r'(x) - r(x)) - 2$$

**Zig**: 如图11-18所示,在这种情况下,x和y的阶改变。y是x的父节点。而且 $r'(y) \leq r(y)$ , $r'(x) \geq r(x)$

$$\delta = r'(y) + r'(x) - r(y) - r(x) \leq r'(x) - r(x) \leq 3(r'(x) - r(x))$$

**命题11-4**: 令 $T$ 为根为 $t$ 的伸展树,令 $\Delta$ 为 $r(T)$ 在一个深度为 $d$ 的节点的全变化。我们有:

$$\Delta \leq 3(r(t) - r(x)) - d + 2$$

**证明**: 伸展包含 $c = \lceil d/2 \rceil$ 伸展子步骤的节点 $x$ ,每个子步骤是zig-zig或zig-zag。如果 $d$ 是奇数,则最后一个步骤是zig。令 $r_0(x) = r(x)$ 为 $x$ 的最初的阶,对于 $i = 1, \dots, c$ ,令 $r_i(x)$ 为第*i*个子步骤之后 $x$ 的阶,并且令 $\delta_i$ 为由第*i*个子步骤造成的 $r(T)$ 的变化。由命题11-3可知,由 $x$ 的伸展造成的 $r(T)$ 的总变化 $\Delta$ :

$$\begin{aligned} \Delta &= \sum_{i=1}^c \delta_i \leq 2 + \sum_{i=1}^c 3(r_i(x) - r_{i-1}(x)) - 2 \\ &= 3(r_c(x) - r_0(x)) - 2c + 2 \leq 3(r(t) - r(x)) - d + 2 \end{aligned}$$

由命题11-4可知,如果对节点 $x$ 的伸展支付 $3(r(t) - r(x)) + 2$ cyber-dollars,我们有足够的cyber-dollars保持不变,在 $T$ 的每个节点 $w$ 中保持 $r(w)$ ,并为伸展工作支付 $d$ cyber-dollars。由于根 $t$ 的大小是 $n$ ,它的阶 $r(t) = \log n$ 。鉴于 $r(x) \geq 0$ ,伸展工作的花费是 $O(\log n)$ cyber-dollars。为了完成分析,我们要对一个节点插入或删除时保持不变计算成本。

向一个有 $n$ 个键的伸展树中插入一个新节点 $w$ 时, $w$ 的所有祖先的阶都增加了。也就是,令 $w_0, w_1, \dots, w_d$ 为 $w$ 的祖先,其中 $w_0 = w$ , $w_i$ 是 $w_{i-1}$ 的父节点, $w_d$ 是根。对于 $i = 1, \dots, d$ ,令 $n'(w_i)$ 和 $n(w_i)$ 分别为 $w_i$ 插入前后的大小,并且令 $r'(w_i)$ 和 $r(w_i)$ 分别为 $w_i$ 插入之前和之后的阶。我们有

$$n'(w_i) = n(w_i) + 1$$

而且,由于 $n(w_i) + 1 \leq n(w_{i+1})$ ,对于 $i = 0, 1, \dots, d-1$ ,每一个*i*在以下这个范围内:

$$r'(w_i) = \log(n'(w_i)) = \log(n(w_i) + 1) \leq \log(n(w_{i+1})) = r(w_{i+1})$$

因此,由插入引起的 $r(T)$ 的总变化为:

$$\begin{aligned} \sum_{i=1}^d (r'(w_i) - r(w_i)) &\leq r'(w_d) + \sum_{i=1}^{d-1} (r(w_{i+1}) - r(w_i)) \\ &= r'(w_d) - r(w_0) \leq \log n \end{aligned}$$

因此,当一个新节点插入时 $O(\log n)$ ,cyber-dollars足以维持不变。

当从有 $n$ 个键的伸展树中删除一个节点 $w$ 时,所有 $w$ 的祖先的阶都降低了。因此,由于删除造成的 $r(T)$ 的总变化是负的,当一个节点被删除时,我们不需要任何支付维持不变。因此,我们可以在下列命题中总结摊销分析(有时被称为伸展树的“平衡命题”)。

**命题 11-5：**考虑一棵伸展树中一系列的  $m$  个操作，每一个是搜索、插入或删除，从只有零键的伸展树开始。同时，令  $n_i$  为操作  $i$  之后树中键的数量， $n$  是插入操作总数，则执行一系列操作的总运行时间：

$$O(m + \sum_{i=1}^m \log n_i)$$

即  $O(m \log n)$ 。

换句话说，在一棵伸展树中执行一个搜索、插入或删除的摊销运行时间复杂度是  $O(\log n)$ ，其中  $n$  是伸展树的大小。因此，伸展树可以以对数时间摊销的性能实现有序 ADT 映射。其摊销性能匹配 AVL 树、(2, 4) 树和红黑树在最坏情况下的性能，但它仅使用一棵不需要存储每个节点的附加平衡信息的简单二叉树就能实现这样的性能。此外，伸展树有许多和这些其他平衡搜索树不同的有趣属性。我们在以下命题中探讨一个这样的额外属性（有时被称为伸展树的“静态最优”的命题）。

**命题 11-6：**在伸展树上考虑一系列的  $m$  个操作，每一个是搜索、插入或删除，从一棵具有零键值的伸展树  $T$  开始。同时，用  $f(i)$  表示在伸展树中实体  $i$  被访问的数量，即它的频率，用  $n$  表示条目的总数。假设每个条目至少被访问一次，那么执行这一系列操作的总运行时间为：

$$O(m + \sum_{i=1}^n f(i) \log(m / f(i)))$$

此处省略这一命题的证明，但它并不像别人说的那样难以证明和想象。值得注意的是，这个命题说明摊销访问一个节点  $i$  的运行时间是  $O(\log(m / f(i)))$ 。

## 11.5 (2, 4) 树

在本节中，我们考虑一种称为 (2, 4) 树的数据结构。它是多路搜索树这种通用数据结构的一个特殊例子。在多路搜索树中，内部节点的孩子节点可能会超过两个。其他形式的多路搜索树将在 15.3 节中讨论。

### 11.5.1 多路搜索树

回想一下，通用树被定义为内部节点可能会有很多的孩子。在本节中，我们将讨论如何将通用树作为多路搜索树使用。映射项以  $(k, v)$  形式存储在搜索树中， $k$  是键， $v$  是和键相关联的值。

#### 多路搜索树的定义

令  $w$  为有序树的一个节点。如果  $w$  有  $d$  个孩子，则称  $w$  是  $d$ -node。我们将一棵多路搜索树定义为一棵有以下属性的有序树  $T$ （其属性在图 11-23a 中阐明）：

- $T$  的每个内部节点至少有两个孩子。也就是说，每个内部节点是一个  $d$ -node，其中  $d \geq 2$ 。
- $T$  的每个内部  $d$ -node  $w$ ，其孩子  $c_1, \dots, c_d$  按顺序存储  $d - 1$  个键 - 值对  $(k_1, v_1), \dots, (k_{d-1}, v_{d-1})$ ， $k_i \leq \dots \leq k_{d-1}$ 。
- 通常定义  $k_0 = -\infty$  和  $k_d = +\infty$ 。每个条目  $(k, v)$  储存在一个以  $c_i$  为根的  $w$  的子树的一个节点上，其中  $i = 1, \dots, d$ ， $k_{i-1} \leq k \leq k_i$ 。

也就是说，如果认为存储在  $w$  的键的集合包含特殊的虚拟键  $k_0 = -\infty$  和  $k_d = +\infty$ ，那么存储

在以孩子节点  $c_i$  为根的  $T$  的子树上的键  $k$  一定是存储在  $w$  上的两个键“之间”的一个。这个简单的观点产生了以下规则： $d$ -node 存有  $d - 1$  个常规键，并且它也形成了在多路搜索树中搜索算法的基础。

根据上述定义，多路搜索的外部节点不存储任何数据并且仅仅作为“占位符”。这些外部节点可以有效地以 None 引用表示，就像我们在二叉搜索树中约定的那样（11.1 节）。然而，为了拓展，我们将讨论这些不存储任何东西的实际节点。基于这个定义，在一棵多路搜索树中，键-值对的数目和外部节点的数目存在有趣的关系。

**命题 11-7：**一棵有  $n$  个节点的多路搜索树有  $n + 1$  外部节点。我们把这个命题的证明留作练习（C-11.52）。

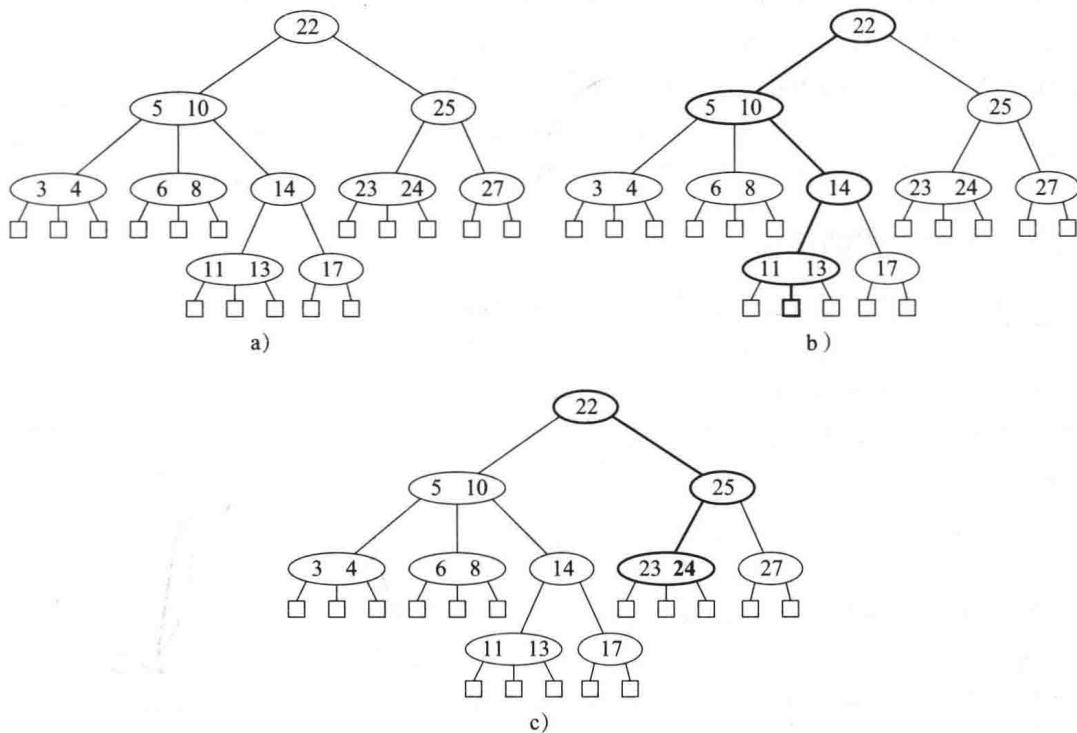


图 11-23 a) 多路搜索树  $T$ ; b) 在  $T$  中搜索键为 12 的路径（不成功搜索）; c) 在  $T$  中搜索键为 24 的路径（成功搜索）

### 多路树搜索

在多路搜索树  $T$  中搜索键为  $k$  的节点很简单。我们在  $T$  中从根节点开始跟踪路径执行搜索（见图 11-23b 和图 11-23c）。在搜索  $d$ -node 节点  $w$  时，我们比较键  $k$  和存储在  $w$  上的键  $k_1, \dots, k_{d-1}$ 。如果  $k = k_i$ ，搜索就成功完成了；否则，继续搜索  $w$  的孩子  $c_i$ ，使得  $k_{i-1} < k < k_i$ （通常定义  $k_0 = -\infty$  和  $k_d = +\infty$ ）。如果到达外部节点，那么可以知道树  $T$  中没有键为  $k$  的节点，搜索不成功并且终止。

### 主要的多路径搜索树数据结构

在 8.3.3 节中，我们讨论了表示通用树的有关数据结构。当然，这也可以用来表示一棵多路搜索树。当使用通用树来实现多路搜索树时，我们必须在每个节点存储一个或多个与该节点相关联的键-值对。也就是说，我们需要存储  $w$  集合的一个引用，集合中存储的是  $w$  的项。

在多路搜索树中搜索键  $k$  时，基本操作是找到键比  $k$  大或者相等的节点中最小的节点。出于这个原因，很自然地将节点本身的信息作为一个有序映射，同时允许使用 `find_ge(k)` 方法。我们说这样的映射可以作为二级数据结构来支持由整个多路搜索树表示的初级数据结构。这种推理看起来就像一个循环论证，因为需要用（二级）有序映射来表示（初级）映射。但是，我们可以使用一种简单的、更先进的解决方案，即通过使用 bootstrapping 技术来避免循环依赖。

在多路搜索树的背景下，每个节点的二级数据结构的理想选择是 10.3.1 节的 Sorted Table Map 类。因为希望确定和键  $k$  匹配的关联值，相应的孩子  $c_i$  使得  $k_{i-1} < k < k_i$ ，我们建议在二级数据结构上将每个键  $k_i$  映射到  $(v_i, c_i)$  对。有了多路搜索树  $T$  的上述实现，处理一个  $d$ -node  $w$  节点的同时对具有键  $k$  的  $T$  进行搜索可以通过二叉搜索操作在  $O(\log d)$  内实现。用  $d_{\max}$  表示  $T$  的任何节点的孩子的最大数目，并用  $h$  表示树  $T$  的高度。多路搜索树的搜索时间为  $O(h \log d_{\max})$ 。如果  $d_{\max}$  是一个常数，则执行一个搜索的运行时间为  $O(h)$ 。

多路搜索树的首要效率目标是保持高度尽可能小。接下来讨论的策略是： $d_{\max}$  距离限制在 4，同时保证高度  $h$  是  $n$  的对数，其中  $n$  为保存在映射中节点的总数。

### 11.5.2 (2, 4) 树的操作

一棵多路搜索树需要保持存储在每个节点的二级数据结构很小，同时需要保持一级多路平衡树是 (2, 4) 树（有时也被称为 2-4 树或 2-3-4 树）。这种数据结构通过维护如下两个简单的属性来实现上述目标（见图 11-24）：

- 大小属性：每个内部节点最多有 4 个孩子。
- 深度属性：所有外部节点具有相同的深度。

再次强调，假设外部节点是空的，并且为了简单起见，描述搜索和更新方法时，假设外部节点是真实的节点，尽管后面的要求并不严格。

维护 (2, 4) 树的大小属性使多路搜索树中的节点非常简单。它也有一个另类的名字“2-3-4 树”，因为它意味着每个内部节点的树有 2 个、3 或 4 个孩子。这条规则的另一个含义是，我们可以使用一个无序列表或有序数组来表示存储在每个内部节点的二级映射，而且所有操作仍可以达到  $O(1)$  的时间性能（因为  $d_{\max} = 4$ ）。对于深度属性，需要在 (2, 4) 树的高度上执行一个重要的约束。

**命题 11-8：** 存储  $n$  个节点的 (2, 4) 树的高度为  $O(\log n)$ 。

**证明：**令  $h$  为存储  $n$  个节点的 (2, 4) 树  $T$  的高度。我们通过式 (11-9) 证明该命题

$$\frac{1}{2} \log(n+1) \leq h \leq \log(n+1) \quad (11-9)$$

为了证明这种说法应先注意到：对于大小属性，深度为 1 时最多可以有 4 个节点，深度为 2 时最多可以有  $4^2$  个节点，以此类推。因此， $T$  树的外部节点的个数最多为  $4^h$ 。同样，由深度属性和 (2, 4) 树的定义，我们必须至少有 2 个深度为 1 的节点，至少有  $2^2$  个深度为 2 的节点，以此类推。因此，在  $T$  树中，外部节点的数量至少为  $2^h$ 。此外，由命题 11-7 可知，

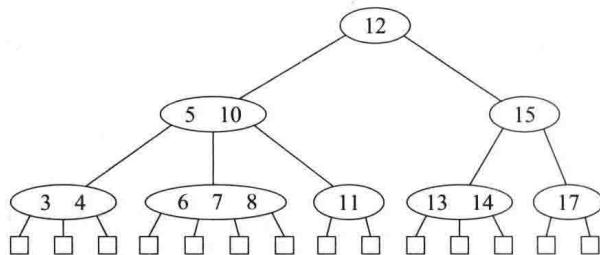


图 11-24 (2, 4) 树

外部节点的数量为  $(n + 1)$ , 因此得到

$$2^h \leq n + 1 \leq 4^h$$

对以上公式以 2 为底去对数, 得到

$$h \leq \log(n + 1) \leq 2h$$

当项被重新排列时, 这证明了式 11-9。 ■

命题 11-8 声明大小和深度属性足以保持多路树的平衡。此外, 这一命题意味着在  $(2, 4)$  树中执行搜索需要  $O(\log n)$  的时间复杂度, 且节点的二级数据结构的具体实现不是一个关键的设计选择, 因为最大孩子数量  $d_{\max}$  是一个常数。

然而对  $(2, 4)$  树进行插入和删除之后, 保持大小和深度属性需要一些操作。接下来我们将讨论这些操作。

### 插入

插入一个键为  $k$  的新节点  $(k, v)$  到  $(2, 4)$  树  $T$  中, 首先对键  $k$  执行搜索。假设  $T$  中没有键为  $k$  的节点, 这个搜索非正常终止于外部节点  $z$  中。令  $w$  成为  $z$  的父母节点。我们在节点  $w$  上插入新的项, 并且在  $z$  的左边对  $w$  添加一个新的孩子节点  $y$  (外部节点)。

上述插入方法维持了深度属性, 因为我们在和现有外部节点相同的层级添加一个新的外部节点。然而, 它可能违反了大小属性。

事实上, 如果一个节点  $w$  以前是 4-node, 那么插入后它将成为一个 5-node, 导致  $T$  树不再是  $(2, 4)$  树。这种违反节点大小属性的情况称为在  $w$  节点溢出, 我们必须解决这一问题以恢复  $(2, 4)$  树的属性。令  $c_1, \dots, c_5$  是  $w$  的孩子,  $k_1, \dots, k_5$  键存储在  $w$  中。为了修复节点  $w$  的溢出问题, 我们对  $w$  执行以下分裂操作 (见图 11-25):

- 用  $w'$  和  $w''$  来代替  $w$ , 其中:
  - $w'$  是存储  $k_1$  和  $k_2$  的 (其孩子节点为  $c_1, c_2, c_3$ ) 的 3-node。
  - $w''$  是存储  $k_4$  (其孩子节点为  $c_4, c_5$ ) 的 2-node。
- 如果  $w$  是  $T$  的根节点, 创建一个新的根节点  $u$ , 让  $u$  成为  $w$  的父节点。
- 插入键值  $k_3$  到  $u$  中, 并使得  $w''$  和  $w'$  成为  $u$  的孩子节点。如果  $w$  是  $u$  的第  $i$  个孩子, 那么  $w'$  和  $w''$  将分别为  $u$  的第  $i$  个和第  $i + 1$  个的孩子节点。

由于节点  $w$  的分裂操作,  $w$  的父节点  $u$  可能会发生溢出。如果发生溢出, 它会在节点  $u$  触发一个分裂 (见图 11-26)。分裂操作消除了溢出或传播到当前节点的父节点。在  $(2, 4)$  树中进行一系列的插入操作如图 11-27 所示。

### $(2, 4)$ 树中插入操作的分析

因为  $d_{\max}$  最多为 4, 对新键  $k$  最初的位置搜寻在每个阶段会用  $O(1)$  时间, 因此总体时间为  $O(\log n)$ , 由命题 11-8 得到树的高度为  $O(\log n)$ 。

在单个节点插入一个新键和孩子节点的修改可以在  $O(1)$  时间内实现, 一个分裂操作也是如此。级联分裂操作的数量受到树的高度限制, 这一阶段插入过程运行时间为  $O(\log n)$ 。因此, 在  $(2, 4)$  树中执行插入操作的总时间是  $O(\log n)$ 。

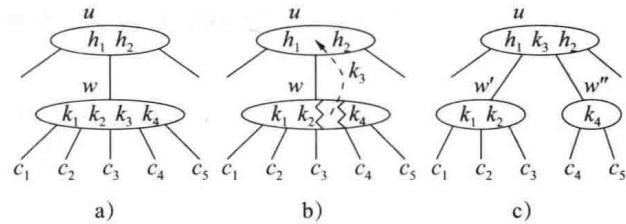


图 11-25 节点分裂: a) 5-node  $w$  节点的溢出; b)  $w$  的第三个键插入到  $w$  的父节点  $u$ ; c) 用 3-node  $w'$  和 2-node  $w''$  替换节点  $w$

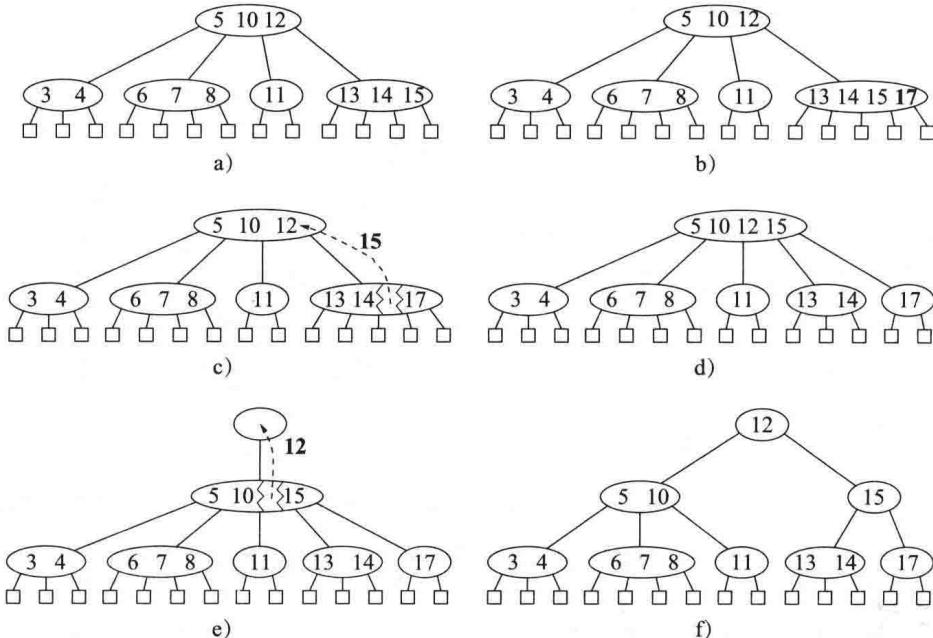


图 11-26 (2, 4) 树的插入操作发生了分裂: a) 插入前; b) 插入 17 发生了溢出; c) 一个分裂; d) 在分裂之后出现了一个新的溢出; e) 另一个分裂, 创建一个新的根节点; f) 最后的树

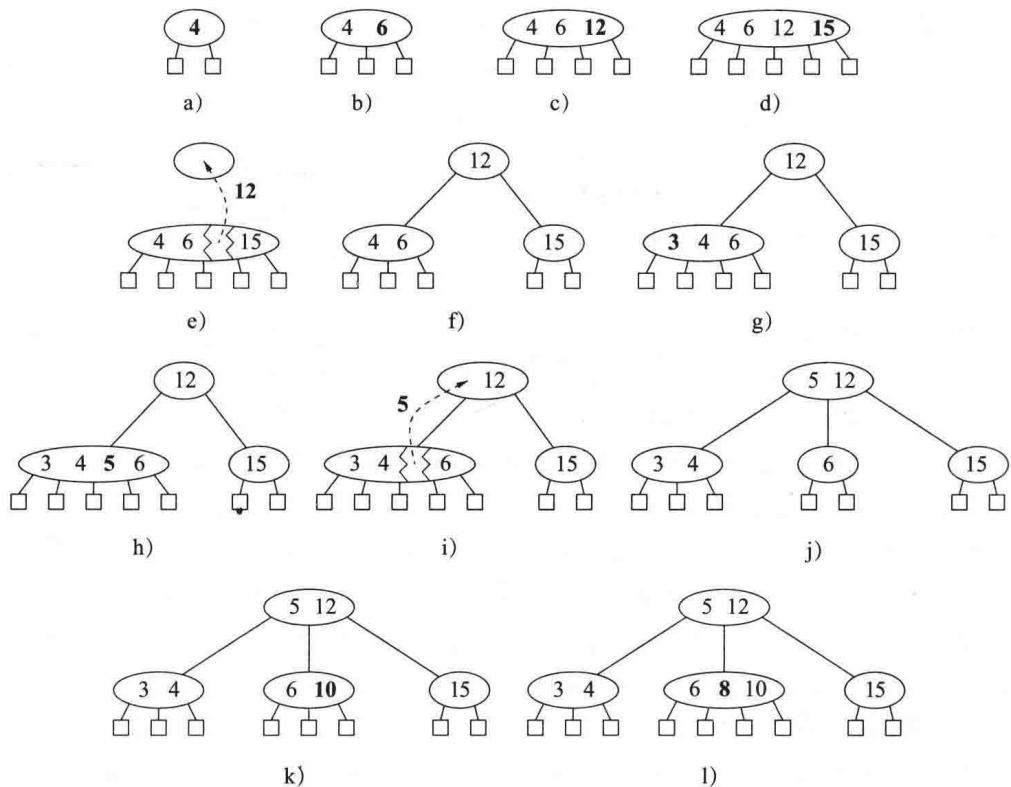


图 11-27 (2, 4) 树的一系列插入操作: a) 一个节点的初始树; b) 插入键为 6 的节点; c) 插入键为 12 的节点; d) 插入键为 15 的节点, 将会引起溢出; e) 分裂, 这将会引起创建一个新的根节点; f) 分裂之后; g) 插入键为 3 的节点; h) 插入键为 5 的节点, 引发了溢出; i) 分裂; j) 分裂之后; k) 插入键为 10 的节点; l) 插入键为 8 的节点

### 删除

现在考虑在(2, 4)树T中删除键为k的节点。我们以在T中搜索键为k的项开始。从一个(2, 4)树中删除项可以简化为这样的情况：删除的节点存储在节点w(w的孩子节点是外部节点)。假设，实例中，所要移除的键为k的项存储在节点z的第i个项( $k_i, v_i$ )，节点z存储的( $k_i, v_i$ )只有内部节点的孩子节点。在这种情况下，我们和一个合适的项交换了( $k_i, v_i$ )，该项存储在带有外部节点的孩子节点的如下节点w中(见图11-28d)：

1) 最右边的子树的内部节点w在以z的第i个孩子为根的子树上。注意，w的所有孩子节点都是外部节点。

2) 用w的最后一个节点交换节点z的( $k_i, v_i$ )。

一旦确定要删除的项存储在一个只有外部孩子的节点w(因为它已经在w或我们交换成w)，我们可以轻而易举地从w删除节点并删除w的第i个外部节点。

如上所述，从节点w上删除一个项(及其孩子)，应先保存深度的属性，因为我们总是删除只有外部孩子的节点w。然而，在消除这样的外部节点时，我们可能会违反w节点的大小属性。的确，如果w以前是2-node，那么它就变成删除之后没有项的一个1-node(见图11-28a和图11-28d)，在(2, 4)树中这是不允许的。这种违反大小属性的情况称为在节点w下溢。为了弥补下溢，我们立即检查w的兄弟节点是否是一个3-node或4-node。如果发现这样一个兄弟s，就进行转移操作，也就是将s的一个孩子移到w上，将s的一个键移动到w和s的父节点u上，将u的一个键移动到w(见图11-28b和图11-28c)。如果w只有一个兄弟或者兄弟都是2-node，就进行融合操作，合并w及其一个兄弟，创建一个新节点w'并将w的父亲节点u的键移动到w'。

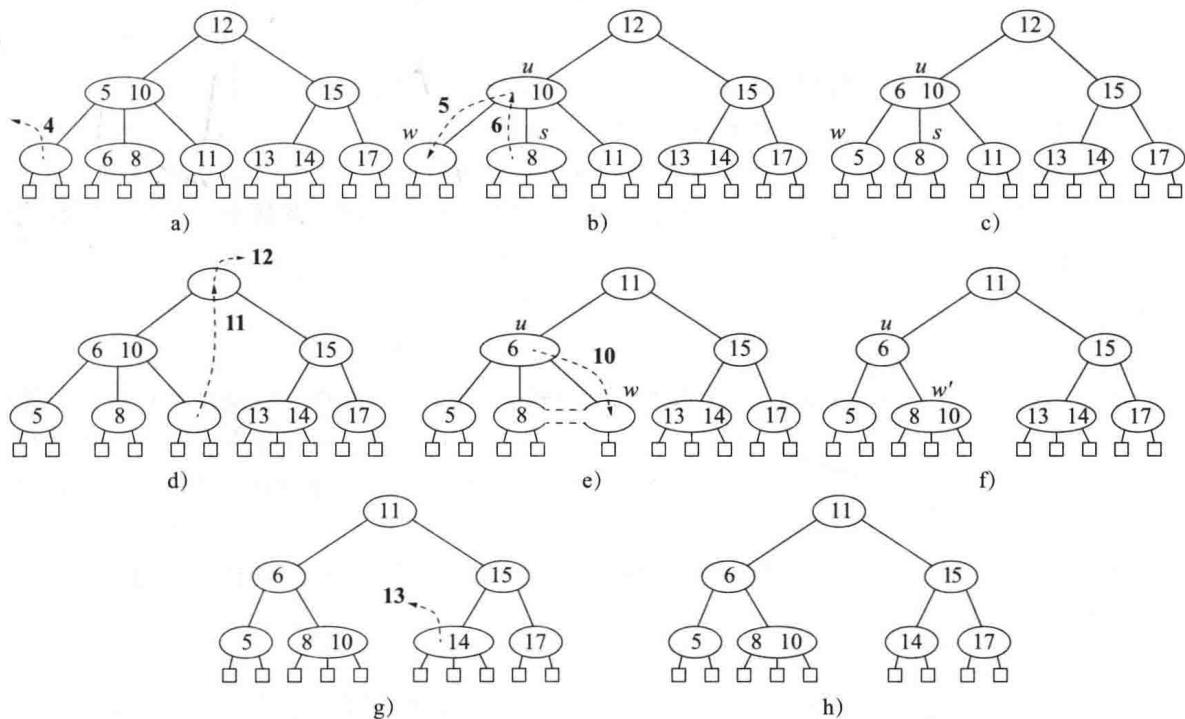


图11-28 (2, 4)树的一系列删除操作；a) 删除键为4的节点，引起下溢。b) 交换操作；c) 交换操作之后；d) 删除键为12的节点，引起下溢；e) 合并操作；f) 合并操作之后；g) 删除键为13的节点；h) 删除操作之后

节点  $w$  处的融合操作可能导致一个新的下溢发生在  $w$  的父亲节点  $u$  上，进而触发  $u$  交换或合并（见图 11-29）。因此，合并操作的数量是有界的，被树的高度限制，这被命题 11-8 证明是  $O(\log n)$ 。如果下溢一直传播到根，那么根被删除（见图 11-29c 和图 11-29d）。

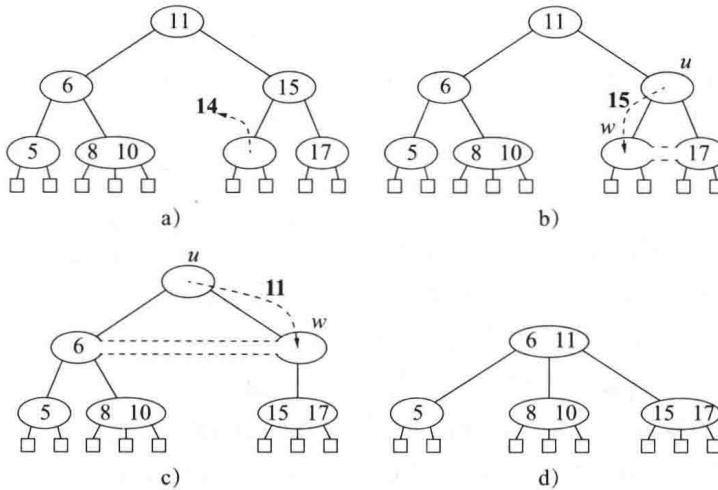


图 11-29 在  $(2, 4)$  树中一个合并的传播：a) 删除键为 14 的节点，引起下溢；b) 合并，引起其他下溢；c) 第二个合并引起根被删除；d) 最后的树

### $(2, 4)$ 树的性能

在有序映射 ADT 方面， $(2, 4)$  树的渐近性能是和 AVL 树一样的（见表 11-2），对大多数操作都保证了对数限制。有  $n$  个键 - 值对的  $(2, 4)$  树的时间复杂度的分析基于以下几点：

- 由命题 11-8 可知，存储  $n$  节点的  $(2, 4)$  树的高度是  $O(\log n)$ 。
- 分裂、交换或合并操作需要  $O(1)$  时间。
- 搜索、插入或删除一个节点需要访问  $O(\log n)$  个节点。

因此， $(2, 4)$  树提供了快速映射搜索和更新操作。 $(2, 4)$  树也和接下来要讨论的数据结构有一种有趣的关系。

## 11.6 红黑树

虽然 AVL 树和  $(2, 4)$  树具有许多很好的特性，但是它们也有一些缺点。例如，AVL 树删除后可能需要执行的多重组操作（旋转）， $(2, 4)$  树在插入和删除之后可能需要进行许多分裂或融合操作。在本章中，我们所讨论的数据结构——红黑树没有这些缺点，在一次更新之后，它使用  $O(1)$  次结构变化来保持平衡。

从形式上讲，红黑树是一棵带有红色和黑色节点的二叉搜索树，其具有下面的属性：

- 根属性：根节点是黑色的。
- 红色属性：红色节点（如果有的话）的子节点是黑色的。
- 深度属性：具有零个或一个子节点的所有节点都具有相同的黑色深度（被定义为黑色祖先节点的数量）。（回想一下，一个节点是它自己的祖先）

红黑树的一个例子如图 11-30 所示。

可以注意到，红黑树和  $(2, 4)$  树（不包括它们的琐碎外部节点）之间有一个有趣的对

应使红黑树的定义更为直观，即给定一棵红黑树，我们可以构建一棵相应的(2, 4)树：合并每一个红色节点 $w$ 到它的父节点，从 $w$ 存储条目到其父节点，并使 $w$ 的子节点变得有序。例如，图11-30的红黑树对应图11-24的(2, 4)树，如图11-31所示。

红黑树的深度属性与(2, 4)树的深度属性相对应，因为红黑树的黑色节点为相应的(2, 4)树的每个节点提供了帮助。

相反，我们可以通过给每一个 $w$ 节点着黑色，然后执行下面的转换（见图11-32），将任何(2, 4)树改变为相应的红黑树。

- 如果 $w$ 是2-node，那么保持 $w$ 的子节点（黑色）是2-node。
- 如果 $w$ 是3-node，那么创建一个新的红色节点 $y$ ，把 $w$ 最后的两个子节点（黑色）给 $y$ ，然后把 $y$ 和 $w$ 的第一个子节点作为 $w$ 的两个子节点。
- 如果 $w$ 是4-node点，那么创建两个新的红色节点 $y$ 和 $z$ ，把 $w$ 的前两个子节点（黑色）给 $y$ ，把 $w$ 的最后两个子节点（黑色）给 $z$ ，最后使 $y$ 和 $z$ 成为 $w$ 的两个子节点。

值得注意的是，在这种结构中，一个红色节点总是有一个黑色父节点。

**命题11-9：**红黑树存储 $n$ 个条目的高度是 $O(\log n)$ 。

**证明：**设 $T$ 是存储 $n$ 个条目的红黑树，并设 $h$ 为 $T$ 的高度。我们通过建立以下事实证明这一命题：

$$\log(n+1)-1 \leq h \leq 2\log(n+1)-2$$

设 $d$ 是具有零个或一个子结点的 $T$ 所有节点的常见黑色深度。令 $T'$ 为 $T$ 相关联的(2, 4)的树，并且令 $h'$ 是 $T'$ 的高度（不包括琐碎的叶子节点）。由红黑树和(2, 4)树之间的对应关系可知 $h' = d$ 。因此，由命题11-8可得， $d = h' \leq \log(n+1)-1$ 。由于红色属性得， $h \leq 2d$ 。因此得到 $h \leq 2\log(n+1)-2$ 。

其他不等式 $\log(n+1)-1 \leq h$ 由命题8-8以及 $T$ 具有 $n$ 个节点的事实可以得出。 ■

### 11.6.1 红黑树的操作

红黑树 $T$ 中的搜索算法与标准二叉树的搜索算法是相同的（见11.1节），因此，在一棵红黑树中进行搜索所花费的时间与树的高度成正比，由命题11-9可知是 $O(\log n)$ 。

(2, 4)树和红黑树之间的对应关系提供了很重要的知识，我们将会在讨论如何在红黑

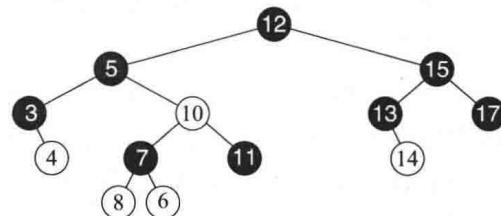


图11-30 红黑树的一个例子，在白色上面绘制“红”的节点。这棵树的常见黑色深度为3

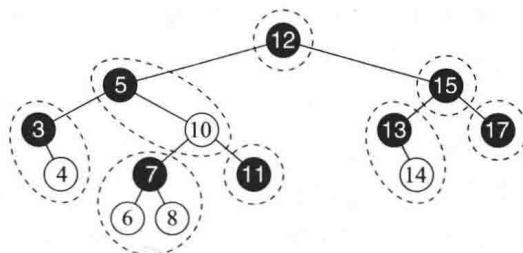


图11-31 一个例子，图11-30的红黑树对应于图11-24的(2, 4)树，基于红色节点及其黑色父节点的高亮分组

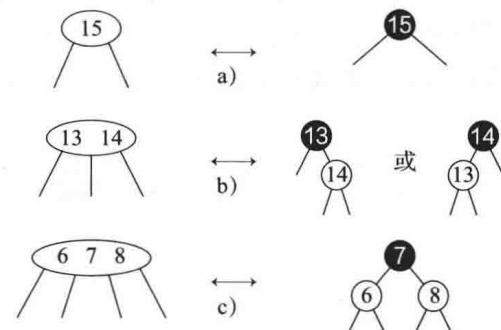


图11-32 一棵红黑树和一棵(2, 4)树节点之间的对应：a) 2-node；b) 3-node；c) 4-node

树上执行更新时用到。事实上，如果没有这个知识，对于红黑树的更新算法就显得神秘复杂。 $(2, 4)$  树的分裂和融合将会通过重新着色邻居的红黑树节点被有效地模仿。如图 11-32b 所示的两种形式，红黑树的旋转将被用于改变 3-node 的方向。

### 插入

现在考虑将键值对  $(k, v)$  插入到红黑树  $T$  中。该算法最初进程是作为一个标准的二叉搜索树（见 11.1.3 节）。也就是说，在  $T$  中搜索  $k$ ，直到达到一个空子树，然后在这个位置插入一个新的叶子节点  $x$ ，存储项。在特殊情况下， $x$  是  $T$  的唯一节点，因此将根着色为黑色。在其他情况下，我们将  $x$  着色为红色。这个动作对应于用外部子节点将  $(k, v)$  插入到  $(2, 4)$  树  $T$  的节点中。这种插入维持了  $T$  的根属性和深度属性，但它可能违反红色属性。事实上，如果  $x$  不是  $T$  的根， $x$  的父结点  $y$  是红色的，那么父结点和子结点（即  $y$  和  $x$ ）都是红色的。值得注意的是，由根属性可知  $y$  不能是  $T$  的根，并且由红色属性（这在以前是满足的）可知  $y$  的父母  $z$  必须是黑色的。由于  $x$  和其父节点是红色的，但  $x$  的祖先  $z$  是黑色的，我们将这种违反红色属性的情况称为节点  $x$  处的双红色。为了解决双红色问题，我们考虑以下两种情况。

**情况 1：** $y$  的兄弟姐妹为黑色（或无）。如图 11-33 所示，在这种情况下，双红色表示，我们已经添加了新节点到对应的  $(2, 4)$  树  $T'$  的 3-node 处，从而有效地创建异常的 4-node。此形式有一个红色的节点 ( $y$ ) 是另一个红色节点 ( $x$ ) 的父节点，而我们希望它有两个红色节点作为兄弟姐妹。要解决这个问题，我们进行了  $T$  的 trinode 重组。该 trinode 重组由操作 `restructure(x)` 来实现，具体步骤如下（再次参考图 11-33，该操作也在 11.2 节进行讨论）：

- 对节点  $x$ ，其父节点  $y$  和祖先节点  $z$ ，按照从左到右的顺序，暂时重新标记它们为  $a$ 、 $b$  和  $c$ ，以使  $a$ 、 $b$  和  $c$  将按照顺序树被有序地遍历。
- 将祖先节点  $z$  用标记节点  $b$  取代，使  $a$  和  $c$  成为  $b$  的子节点，并保持次序关系不变。

在进行 `restructure(x)` 的操作后，我们将  $b$  着色为黑色，将  $a$  和  $c$  着色为红色。因此，重组消除了双红色问题。可以注意到，在树的重组部分的任何路径的一部分确实只有一个黑色的节点，在进行 trinode 重构前后都是这样的。因此，树的黑色深度不受影响。

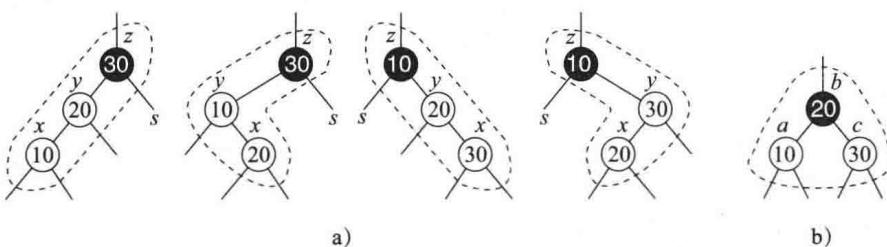


图 11-33 重组红黑树补救双红问题：a) 对于  $x$ 、 $y$ 、 $z$  重组之前的 4 种配置；b) 重组之后

**情况 2：** $y$  的兄弟姐妹是红色的。如图 11-34 所示，在这种情况下，双红色表示在相应的  $(2, 4)$  树  $T'$  中溢出。为了解决这个问题，我们进行了一个相当于分裂的操作，即重新着色：将  $y$  和  $s$  着色为黑色，将其父节点  $z$  着色为红色（除非  $z$  是根节点，在这种情况下，它仍然是黑色的）。在这里我们可以注意到，除非  $z$  是根节点，通过该树的有影响的部分的任何路径部分恰好是一个黑色节点，无论着色前和着色后。因此，树的黑色深度不被重新着色影响，除非  $z$  是根节点，在这种情况下，它增加 1。

然而，双红问题在这种重新着色问题之后可能再次出现，尽管在  $T$  树的更高位置，因为  $z$  可能有一个红色的父节点。如果双红问题再次出现在  $z$  节点，那么在  $z$  上重复考虑两种情

况。因此，在节点  $x$  上重新着色消除了双红问题，或者将它传播到  $x$  的祖先节点  $z$ 。我们继续深度搜索  $T$  进行重新着色直到解决双红问题（最后重新着色或者 trinode 重组）。因此，通过插入引起重新着色的数量不超过树  $T$  深度的一半，即由命题 11-9 提出的  $O(\log n)$ 。

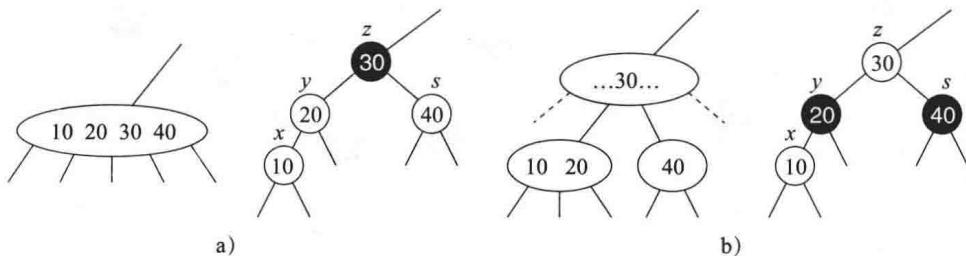


图 11-34 重新着色补救双红问题：a) 分裂之前，对与相关联的  $(2, 4)$  树中对应的 5-node 重新着色；b) 在分裂之后，对与相关联的  $(2, 4)$  的树中的相应节点重新着色

作为进一步的例子，图 11-35 和图 11-36 显示了在红黑树中的一系列插入操作。

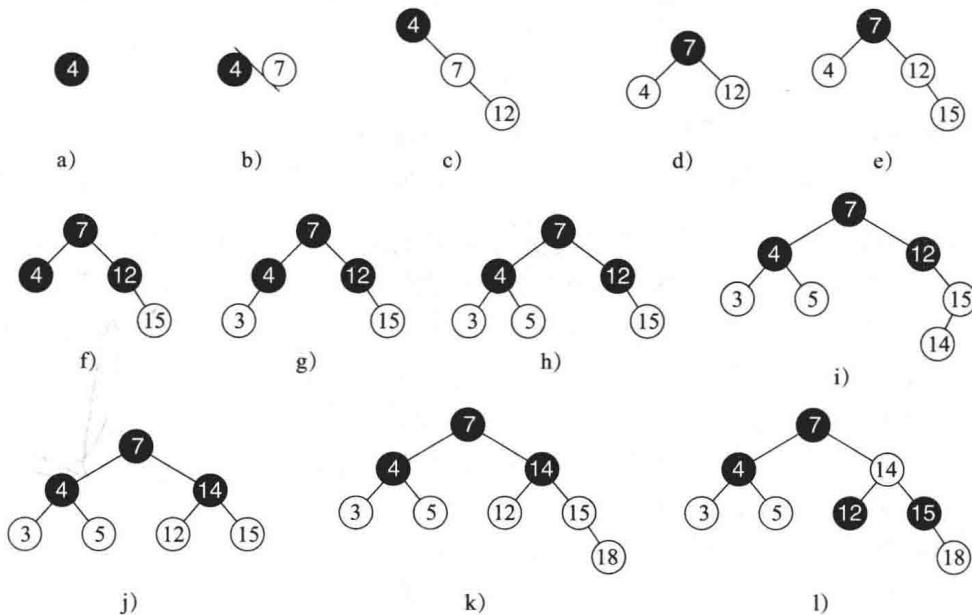


图 11-35 在红黑树中进行一系列插入操作：a) 初始树；b) 7 的插入；c) 12 的插入，引起双红现象；d) 重构之后；e) 插入 15，引起双红现象；f) 重新着色（根仍然是黑色）；g) 插入 3；h) 插入 5；i) 14 的插入，引起双红现象；j) 重构之后；k) 插入 12，引起双红现象；l) 重新着色之后。（后接图 11-36）

### 删除

从红黑树  $T$  中删除键为  $k$  的项和二叉搜索树的删除过程相似（见 11.1.3 节）。在结构上，这种处理结果导致删除至多有一个孩子的节点（或者是最初包含  $k$  的节点或者是它的前继），并提升其剩余的子节点（如果有的话）。

如果删除节点是红色的，这种结构性的变化不会影响树中任何路径的黑色深度，也没有任何违反红色属性，所以结果树仍然是有效的红黑树。在相应的  $(2, 4)$  树  $T'$  中，这表示 3-node 或 4-node 的萎缩。如果删除的节点是黑色的，那么它要么没有孩子要么它有一个子

节点，这个子节点是一个红色的叶子节点（因为删除的节点的空子树黑色高度为 0）。在后一种情况下，将除去的节点代表一个相应的 3-node 的黑色部分，我们通过重新将提升的孩子着色为黑色来恢复红黑属性。

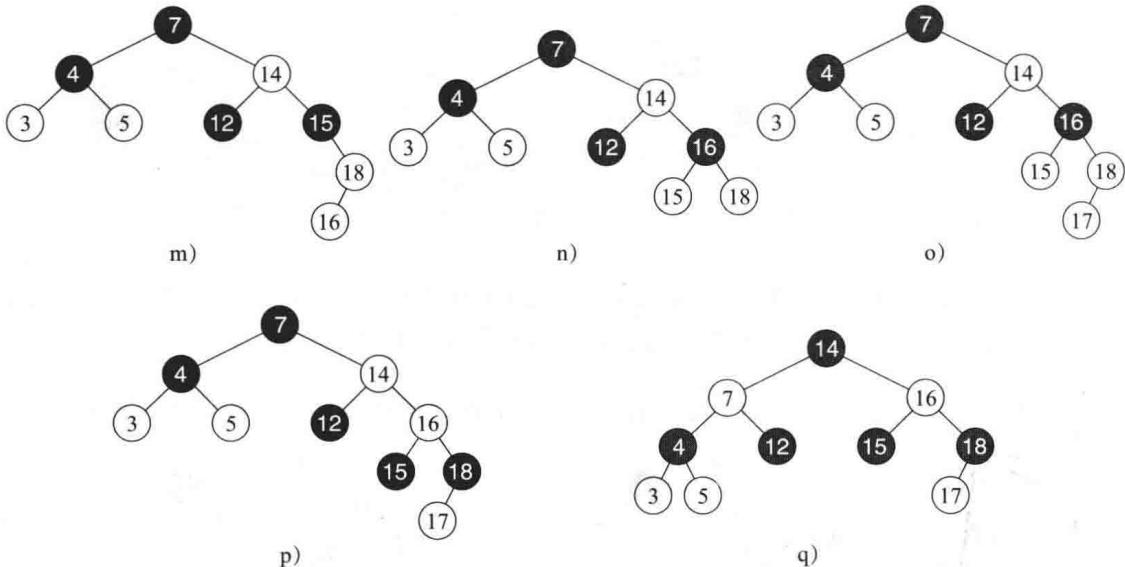


图 11-36 在红黑树中插入一个序列：m) 插入 16，引起双红现象；n) 重构之后；o) 插入 17，引起双红现象；p) 重新着色后，再次出现双红现象，通过重构进行处理；) 重构之后（接图 11-35）

更为复杂的情况就是一个（非根）黑色叶节点被删除。在相应的 2-4 树中，这表示从一个 2 节点中除去一个项目。这种变化会导致沿着通往删除项的路径的黑色深度不足。根据需要，被删除的节点必须有子树黑色高度为 1 的兄弟（假定在黑色叶节点删除之前是有效的红黑树）。

为了补救这种情况，我们考虑到一个更一般的设置，用一个已知有两个子树的  $z$  节点： $T_{\text{heavy}}$  和  $T_{\text{light}}$ ，正好  $T_{\text{light}}$ （如果有）的根节点是黑色，同时  $T_{\text{heavy}}$  的黑色深度恰好比  $T_{\text{light}}$  高 1，如图 11-37 所示。在一个除去黑色叶子的情况下， $z$  是该叶子的父亲， $T_{\text{light}}$  是删除之后仍然存在的空子树。我们描述更一般情况下的不足，因为重新平衡树的算法在某些情况下将把树中的不足推向更高（就像（2, 4）树的删除解决方案有时会级联向上）。我们用  $y$  表示  $T_{\text{heavy}}$  的根（存在这样的节点，因为  $T_{\text{heavy}}$  的黑色深度至少为 1）。

我们考虑三种可能的情况以弥补不足。

情况 1：节点  $y$  是黑色的，同时有一个红色的孩子节点  $x$ （见图 11-38）。执行 trinode 重组，正如最初 11.2 节所描述的。操作 restructure( $x$ ) 需要节点  $x$ 、它的父节点  $y$  和祖先节点  $z$ ，从左到右暂时将它们标记为  $a$ 、 $b$  和  $c$ ，并用标记为  $b$  的节点取代  $z$ ，使其成为其他两个节点的父节点。我们将  $a$  和  $c$  染色为黑色，并给  $b$  着上之前  $z$  的颜色。

注意，重组之后的结果显示  $T_{\text{light}}$  路径中包括了一个额外的黑色节点，从而弥补了不足。相反，图 11-38 中任何其他三个子树黑色节点的数量仍然保持不变。

解决这种情况对应于（2, 4）树  $T$  中在  $z$  的两个子节点之间的转换操作。 $y$  有一个红色节点

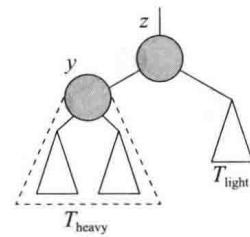


图 11-37 节点  $z$  的子树的黑色高度之间的不足。灰度颜色说明  $y$  和  $z$  表示着以下事实：这些节点可被着色为黑色或红色。

的事实向我们保证它代表一个 3-node 或 4-node。实际上，先前存储在  $z$  中的项被降级为一个新的 2-node 来解决不足，而存储在  $y$  中的项或者它的子节点得以提升，取代原先存储在  $z$  中的项。

情况 2：节点  $y$  是黑色，并且  $y$  的两个子节点是黑色（或无）。解决这种情况相当于在相当的 (2, 4) 树  $T'$  中进行一个融合操作。同时  $y$  必须代表一个 2-node。我们做了重新着色：将  $y$  着为红色，如果  $z$  是红色的，则将它着为黑色（见图 11-39）。这并没有违反任何红色属性，因为  $y$  没有红色的子节点。

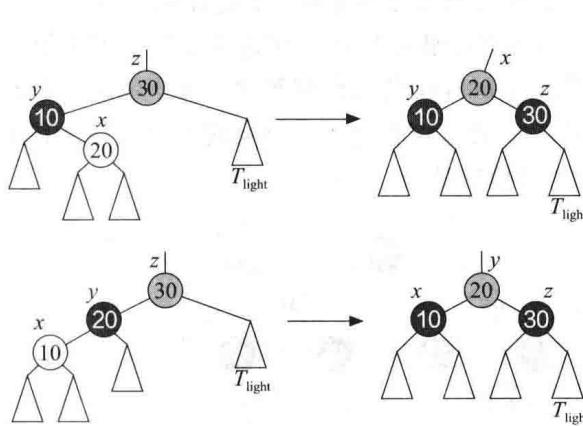


图 11-38 通过执行 trinode 重组  $\text{restructure}(x)$  解决  $T_{\text{light}}$  处的一个黑色不足。两种可能的配置如图所示（其他两个配置是对称的）。左侧图中  $z$  的灰色部分表示这个节点可能被着成黑色或者红色。重组部分的根被赋予了相同颜色，而该节点的子节点最后都被着成黑色

在  $z$  原来为红色的情况下，相应的 (2, 4) 树中，其父节点是 3-node 或者 4-node，这样重新着色解决了不足。（见图 11-39a）这种方法结果导致  $T_{\text{light}}$  增加了一个额外的黑色节点，而重新着色没有影响  $T_{\text{heavy}}$  的子树路径中的黑色节点的数目。

在  $z$  原来的颜色为黑色的情况下，相应的 (2, 4) 树中，其父节点是 2-node，重新着色没有增加  $T_{\text{light}}$  路径中黑色节点的数目。事实上，它减少了  $T_{\text{heavy}}$  路径中黑色节点的数目（见图 11-39b）。此步骤完成后， $z$  的两个孩子将具有相同的黑色高度。然而，位于  $z$  的整个树根变得不足，从而传播问题显得更高了，我们必须重复考虑  $z$  的父亲节点的所有三种情况作为补救。

情况 3：节点  $y$  是红色的（见图 11-40）。因为  $y$  是红色的，同时  $T_{\text{heavy}}$  有至少为 1 的黑

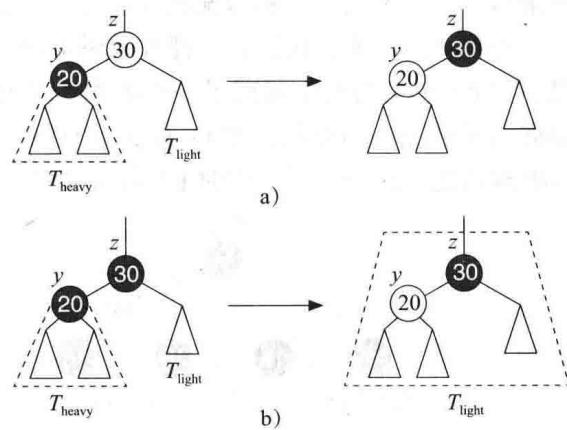


图 11-39 在  $T_{\text{light}}$  中通过重新着色操作来解决黑色不足。a)  $z$  原本是红色的，颠倒  $y$  和  $z$  的颜色解决了黑色的不足，结束进程；b)  $z$  原本是黑色的，重新着色  $y$  时， $z$  的整个子树有一个黑色的不足，需要级联的补救措施

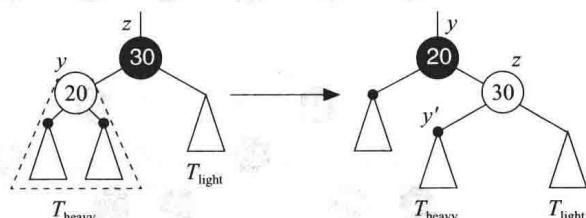


图 11-40 关于红色节点  $y$  和黑色节点  $z$  的反转和重新着色，假设  $z$  有一个黑色的不足。这相当于在 (2, 4) 树中相应的 3-node 的方向变化。通过树这一部分这个操作不会影响任何路径的黑色深度。此外，因为  $y$  为原本是红色的， $z$  的新子树必须有一个黑色的根  $y'$  和一个等同于  $T_{\text{heavy}}$  原先的黑色深度。因此，转变之后，节点  $z$  仍然存在一个黑色的不足

色深度， $z$  必须是黑色的， $y$  的两个子树必须每一个都有一个黑色的根，并且黑色的深度等于  $T_{\text{heavy}}$  的深度。这种情况下，我们把  $y$  和  $z$  进行旋转，然后重新将  $y$  着为黑色，将  $z$  着为红色。这是指在一个相关的  $(2, 4)$  树  $T'$  中一个 3-node 的重新调整。

这并不能立即解决不足，因为  $z$  的新子树是拥有黑色根  $y'$  的一个  $y$  的旧的子树，同时其黑色高度等于  $T_{\text{heavy}}$  的原有高度。我们重新采用算法来解决  $z$  的不足，已知新的子节点  $y'$  (即  $T_{\text{heavy}}$  的根)，现在是黑色的，因此这种情况适用于情况一或者情况二。此外，下一个应用将是最后一次，因为第 1 种情况始终能终止并且第 2 种情况将会终止假定  $z$  是红色的。

在图 11-41 中，我们在一棵红黑树上展示了一系列的删除操作。在这些图片中虚线的边缘，如 c) 中 7 的右边展现了一个有黑色不足的分支，目前尚未得到解决。我们在 c) 和 d) 中展示了情况一的重组，在 f) 和 g) 中展示了情况二的重新着色。最终反转 i) 和 j) 两个部分展现情况三这个例子，同时 k) 表示情况二重新着色的结束。

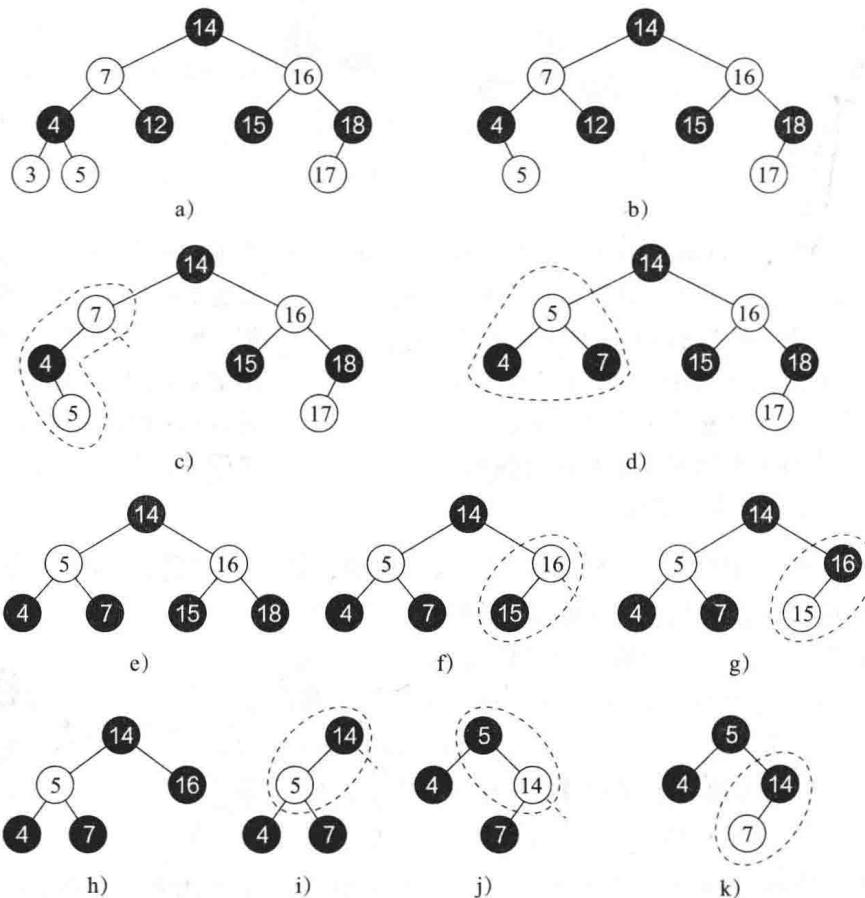


图 11-41 在红黑树中的一系列删除操作：a) 初始树；b) 删除 3；c) 删除 12，造成 7 右边的黑色不足（通过重组处理）；d) 重组之后；e) 删除 17；f) 删除 18，造成 16 右边的黑色不足（通过重新着色处理）；g) 重新着色之后；h) 删除 15；i) 删除 16，造成 14 右边的黑色不足（由最初的旋转处理）；j) 在旋转后的黑赤字需要由重新着色处理；k) 重新着色

### 红黑树的性能

红黑树的渐近性能与 AVL 树或有序映射 ADT 类型的  $(2, 4)$  树的渐进性能相同，对于大多数操作保证了对数时间界限 (AVL 性能的总结见表 11-2)。红黑树的主要优点在于插入

或删除只需要常数步的调整操作（这是相对于 AVL 树和 (2, 4) 树，在最坏的情况下，两者的每个映射的结构调整均需要对数倍的时间操作）。也就是说，在红黑树的插入或删除操作中，搜索一次需要对数级时间，并且可能需要对数倍的级联向上重新着色操作。下面的命题显示，对于单个的映射操作有一个常数数量的旋转或者调整操作。

**命题 11-10：**在一棵存储  $n$  个项目的红黑树中插入一个项可在  $O(\log n)$  的时间内完成，并且需要  $O(\log n)$  的重新着色且至多需要一次的 trinode 重组。

**证明：**回想一下，插入开始的时候会向下搜索，创建一个新的叶节点，然后一个潜在向上操作会造成双红问题。可能有很多对数运算重新着色，由于情况 2 应用的向上级联，但情况 1 行动单一的应用消除了一个 trinode 重组双红问题。因此，一个红黑树的插入至多需要一次重组操作。■

**命题 11-11：**在一棵存储  $n$  个项目的红黑树中删除一个项可在  $O(\log n)$  的时间内完成，并且需要  $O(\log n)$  的重新着色且最多需要两次调整操作。

**证明：**删除操作始于标准二叉搜索树的删除算法，所需要的时间与树的深度成正比。对于红黑树，深度为  $O(\log n)$ 。随着从删除节点的父节点一直向上操作又重新平衡。

我们考虑三种情况以补救造成的黑色不足。情况 1 需要一次 trinode 重组操作来完成过程，所以这种情况下最多应用一次。情况 2 可能被应用对数级次数，但是它仅涉及每个应用最多两个节点的重新着色。情况 3 需要旋转，但这种情况下只应用一次，因为如果旋转不能解决问题，下一个动作将是情况 1 或情况 2 终止。

在最坏情况下，将会有情况 2 的  $O(\log n)$  次重新着色、情况 3 的单个旋转以及情况 1 的一次 trinode 重组。■

### 11.6.2 Python 实现

RedBlackTreeMap 类通过代码段 11-15 ~ 11-17 的代码实现。它继承自标准 TreeMap 的类，并依赖于 11.2 节中描述的平衡框架。

代码段 11-15 RedBlack TreeMap 类的开始 (后接代码段 11-16)

```

1 class RedBlackTreeMap(TreeMap):
2     """Sorted map implementation using a red-black tree."""
3     class _Node(TreeMap._Node):
4         """Node class for red-black tree maintains bit that denotes color."""
5         __slots__ = '_red'      # add additional data member to the Node class
6
7     def __init__(self, element, parent=None, left=None, right=None):
8         super().__init__(element, parent, left, right)
9         self._red = True        # new node red by default

```

代码段 11-16 RedBlack TreeMap 类的继续。(接代码段 11-15, 后接代码段 11-17)

```

10     #----- positional-based utility methods -----
11     # we consider a nonexistent child to be trivially black
12     def _set_red(self, p): p._node._red = True
13     def _set_black(self, p): p._node._red = False
14     def _set_color(self, p, make_red): p._node._red = make_red
15     def _is_red(self, p): return p is not None and p._node._red
16     def _is_red_leaf(self, p): return self._is_red(p) and self.is_leaf(p)
17
18     def _get_red_child(self, p):
19         """Return a red child of p (or None if no such child)."""

```

```

20     for child in (self.left(p), self.right(p)):
21         if self._is_red(child):
22             return child
23     return None
24
25     #----- support for insertions -----
26 def _rebalance_insert(self, p):
27     self._resolve_red(p)                                # new node is always red
28
29 def _resolve_red(self, p):
30     if self.is_root(p):
31         self._set_black(p)                            # make root black
32     else:
33         parent = self.parent(p)
34         if self._is_red(parent):
35             uncle = self.sibling(parent)
36             if not self._is_red(uncle):
37                 middle = self._restructure(p)
38                 self._set_black(middle)
39                 self._set_red(self.left(middle))
40                 self._set_red(self.right(middle))
41             else:
42                 grand = self.parent(parent)
43                 self._set_red(grand)
44                 self._set_black(self.left(grand))
45                 self._set_black(self.right(grand))
46                 self._resolve_red(grand)

```

### 代码段 11-17 RedBlack TreeMap 类的总结 (接代码段 11-16)

```

47     #----- support for deletions -----
48 def _rebalance_delete(self, p):
49     if len(self) == 1:
50         self._set_black(self.root())      # special case: ensure that root is black
51     elif p is not None:
52         n = self.num_children(p)
53         if n == 1:                      # deficit exists unless child is a red leaf
54             c = next(self.children(p))
55             if not self._is_red_leaf(c):
56                 self._fix_deficit(p, c)
57         elif n == 2:                    # removed black node with red child
58             if self._is_red_leaf(self.left(p)):
59                 self._set_black(self.left(p))
60             else:
61                 self._set_black(self.right(p))
62
63 def _fix_deficit(self, z, y):
64     """Resolve black deficit at z, where y is the root of z's heavier subtree."""
65     if not self._is_red(y): # y is black; will apply Case 1 or 2
66         x = self._get_red_child(y)
67         if x is not None: # Case 1: y is black and has red child x; do "transfer"
68             old_color = self._is_red(z)
69             middle = self._restructure(x)
70             self._set_color(middle, old_color)      # middle gets old color of z
71             self._set_black(self.left(middle))      # children become black
72             self._set_black(self.right(middle))
73         else: # Case 2: y is black, but no red children; recolor as "fusion"
74             self._set_red(y)
75             if self._is_red(z):
76                 self._set_black(z)                  # this resolves the problem
77             elif not self.is_root(z):

```

```

78     self._fix_deficit(self.parent(z), self.sibling(z)) # recur upward
79 else: # Case 3: y is red; rotate misaligned 3-node and repeat
80     self._rotate(y)
81     self._set_black(y)
82     self._set_red(z)
83     if z == self.right(y):
84         self._fix_deficit(z, self.left(z))
85     else:
86         self._fix_deficit(z, self.right(z))

```

从代码段 11-15 开始，通过重写嵌套 `_Node` 类的定义引入一个附加的布尔值来表示节点的当前颜色。我们的构造函数有意将新节点的颜色设置为红色，以符合插入节点的方法。在代码段 11-16 的开头定义几个附加的实用功能，帮助设置节点的颜色和查询各种条件。

当一个元素已作为一个叶子节点被插入树中，`_rebalance_insert` 的钩子将被调用，使我们有机会修改树的结构。新节点默认情况下是红色的，所以我们只需要寻找新节点的特殊情况，新节点是根（在这种情况下，它应该是黑色的），或者有一个双重红色问题，因为新节点的父节点可能是红色的。为了弥补这种违规行为，我们严格遵循 11.6.1 节所描述的情况分析。

删除后的再平衡也遵循 11.6.1 节描述的情况分析。一个额外的挑战是，在 `_rebalance_delete` 被调用的同时，旧节点已经从树中移除。在移除节点的父节点上调用钩子。某些情况下分析取决于知道关于被去除的节点的属性。幸运的是，我们可以根据红黑树的信息进行逆向处理。特别是，如果  $p$  表示移除节点的父节点，它必须是：

- 如果  $p$  没有子节点，移除的节点是红叶（练习 R-11.26）。
- 如果  $p$  有一个子节点，已删除节点是一个黑叶，造成空缺，除非剩下的一个子节点是一个红色的叶子（练习 R-11.27）。
- 如果  $p$  有两个子节点，则移除的是一个被升级并且有红色子节点的黑色节点（练习 R-11.28）。

## 11.7 练习

请访问 [www.wiley.com/college/goodrich](http://www.wiley.com/college/goodrich) 以获得练习帮助。

### 巩固

- R-11.1 如果插入项  $(1, A)、(2, B)、(3, C)、(4, D)、(5, E)$ （按照这个顺序）到初始为空的二叉搜索树，它会是什么样子？
- R-11.2 将键为  $30、40、24、58、48、26、11、13$ （按顺序）的项插入一棵空的二叉搜索树。每次插入后绘制树。
- R-11.3 有多少种可以存储键值  $\{1, 2, 3\}$  的不同的二叉搜索树？
- R-11.4 Amongus 博士声称，一个固定集合的条目被插入到二叉搜索树中的顺序是不重要的——每次都会产生相同的树的结果。请给出一个小例子，证明他的观点是错的。
- R-11.5 Amongus 博士声称，一个固定集合的条目被插入到 AVL 树的顺序是不重要的——每次都会产生相同的树的结果。请给出一个小例子，证明他的观点是错的。
- R-11.6 从代码段 11-4 中发现，`TreeMap._subtree_search` 实用程序的实现依赖于递归。对于一棵大的不平衡树，Python 对于递归深度的默认限制可能是禁止递归的。给出一种非递归的实现