云南大学数学与统计学院
上机实践报告

课程名称： 数据结构与算法实验	**年级：** 2015 级	**上机实践成绩：**
指导教师： 陆正福	**姓名：** 刘鹏	
上机实践名称： 映射实验	**学号：** 20151910042	**上机实践日期：** 2017-05-14
上机实践编号： No.10	**组号：**	**上机实践时间：** 上午 3、4 节

一、实验目的

1. 熟悉与映射、字典等有关的数据结构与算法
2. 熟悉主讲教材 Chapter 10 的代码片段

二、实验内容

1. 与关联存取有关的数据结构设计与算法设计
2. 调试主讲教材 Chapter 10 的 Python 程序

三、实验平台

Windows 10 1703 Enterprise 中文版；
Python 3.6.0；
Wing IDE Professional 6.0.5-1 集成开发环境。

四、实验记录与实验结果分析

1 题

统计文本文件的词频。

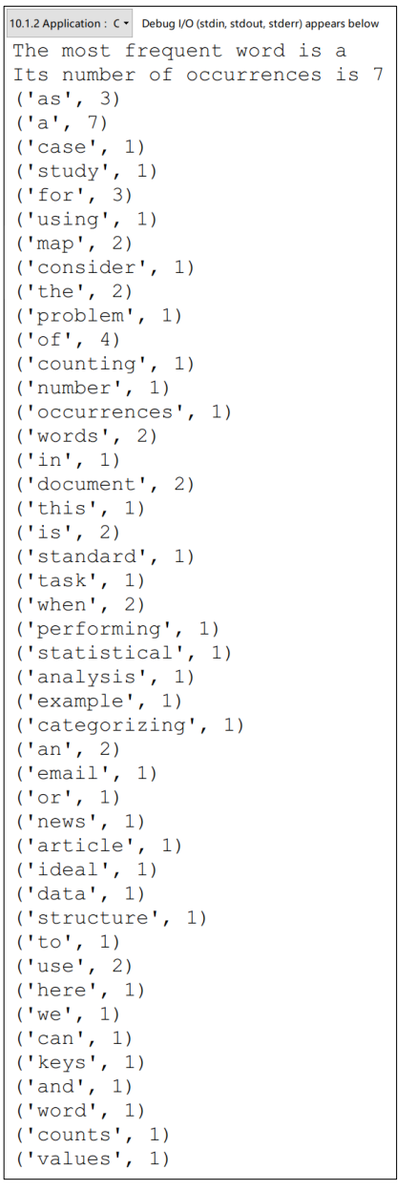
程序代码：

```
1   # 10.1.2 Application: Counting Word Frequencies
2
3   def word_freq_count(filename):
4      freq = {}
5      for piece in open(filename).read().lower().split():
6         # only consider alphabetic characters within this piece
7         word = ''.join(c for c in piece if c.isalpha())
8         if word:   # require at least one alphabetic character
9            freq[word] = 1 + freq.get(word,0)
10
11     max_word = ''
12     max_count = 0
13     for (w,c) in freq.items():  # (key,value) tuples represent (word,count)
14        if c > max_count:
15           max_word = w
16           max_count = c
17     print('The most frequent word is',max_word)
18     print('Its number of occurrences is',max_count)
19     return freq
20
21  #---------------------------- my main function ----------------------------
22  freq = word_freq_count('Test_10.1.2.txt')
23  all_in_one = freq.items()
```

```
24  for i in all_in_one:
25    print(i)
```

程序代码 1

运行结果：



运行结果 1

代码分析：

我对原来的代码进行了函数封装，定义为 word_freq_count 函数，然后进行了调用。这个函数比较简单，值得注意的是，程序员将所有的字符进行了 lower 转换，所以都变成了小写，没有大小写区分，所以这个函数不能对区分大小写的文件进行统计。

2 题

无序 map 的实现。

程序代码：

```
# 10.1.5 Simiple Unsorted Map Implementation

from collections import MutableMapping

class MapBase(MutableMapping):
    """Our own abstract base class that includes a nonpublic _Item class."""

    #------------------------ nested _Item class -----------------------
    class _Item:
        """Lightweight composite to store key-value pairs as map items."""
        __slots__ = '_key','_value'

        def __init__(self,k,v):
            self._key = k
            self._value = v

        def __eq__(self,other):
            return self._key == other._key  # compare items vased on their keys

        def __ne__(self,other):
                return not (self == other)      # opposite of __eq__

        def __It__(self,other):
            return self._key < other._key# compare items based on their keys

class UnsortedTableMap(MapBase):
    """Map implementation usiong an unordered list."""

    def __init__(self,):
        """Create an empty map."""
        self._table = []

    def __getitem__(self,k):
        """Return value associated with key k (raise KeyError if not found)."""
        for item in self._table:
            if k == item._key:
                return item._value
        raise KeyError('key Error: ' + repr(k))

    def __setitem__(self,k,v):
        """Assign value v to key k, overwriting existing value if present."""
        for item in self._table:       # Found a match
            if k == item._key:         # reassign a value
                item._value = value    # and quit
                return
```

```
46          # did not find match for key
47          self._table.append(self._Item(k,v))
48
49      def __delitem__(self,k):
50          """Remove item associated with key k (raise KeyError if not found)."""
51          for j in range(len(self._table)):
52              if k == self._table[j]._key:       # found a match
53                  self._table.pop(j)            # remove item
54                  return
55          raise KeyError('Key Error: ' + repr(k))
56
57      def __len__(self):
58          """Return number of items in the map."""
59          return len(self._table)
60
61      def __iter__(self):
62          """Generate iteration of the map's keys."""
63          for item in self._table:
64              yield item._key                   # yield the KEY
65
66  #------------------------------ my main function ------------------------------
67  a = UnsortedTableMap()
68
69  a.__setitem__('LiuPeng','A')
70  a.__setitem__('LiYue','B')
71  a.__setitem__('God','C')
72
73  c = a.  iter  ()
74  for k in c:
75      print('(',k,',',a.__getitem__(k),')')
76  print(a.__len__())
77  a.__delitem__('LiYue')
78  c = a.__iter__()
79  for k in c:
80      print('(',k,',',a.__getitem__(k),')')
81  print(a.__len__())
```

程序代码 2

运行结果：

```
10.1.5 Simiple Unsorte ▾   Debug I/O (stdin, stdout, stderr) appears below
( LiuPeng , A )
( LiYue , B )
( God , C )
3
( LiuPeng , A )
( God , C )
2
```

运行结果 2

代码分析：

这个代码很简单，也很没用。就是拿列表作为容器，搞了一个 Item class 进行封装，得到了一个类。每一次查找都可能需要遍历数组，所以当需要经常查找时，这个代码实现是灾难级的。

五、教材翻译

Translation

Chapter 10 Maps, Hash Tables, and Skip Lists
＊第十章 映射、哈希表与跳表

10.1 Maps and Dictionaries
＊映射与字典

Python's dict class is arguably the most significant data structure in the language. It represents an abstraction known as a ***dictionary*** in which unique ***keys*** are mapped to associated ***values***. Because of the relationship they express between keys and values, dictionaries are commonly known as ***associative arrays*** or ***maps***. In this book, we use the term *dictionary* when specifically discussing Python's dict class, and the term *map* when discussing the more general notion of the abstract data type.
＊Python 的 dict 类可以说是最重要的数据结构。它表示一种被称为字典的抽象概念，其中一个 key 被映射到相关联的 value。由于在键和值之间存在关系，所以字典通常被称为关联数组或映射。在本书中，我们使用 dictionary 来表示 Python 的 dict 类，而更加一般的映射我们就用 map 来表示。

As a simple example, Figure 10.1 illustrates a map from the names of countries to their associated units of currency.
＊举一个简单的例子，图 10.1 表示了从国家名称到相关货币的映射。

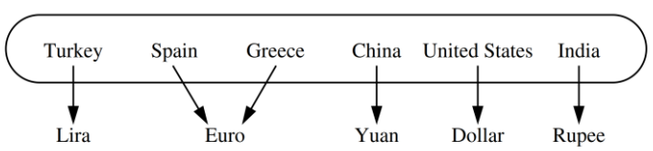


Figure 10.1: A map from countries (the keys) to their units of currency (the values).

We note that the keys (the country names) are assumed to be unique, but the values (the currency units) are not necessarily unique. For example, we note that Spain and Greece both use the euro for currency. Maps use an array-like syntax for indexing, such as currency['Greece'] to access a value associated with a given key or currency['Greece'] = 'Drachma' to remap it to a new value. Unlike a standard array, indices for a map need not be consecutive nor even numeric. Common applications of maps include the following.
＊我们注意到，国家名字被认为是唯一的，但货币单位并不是唯一的。例如，我们注意到西班牙和希腊都用欧元。map 使用类似数组的语法进行索引，例如可以用 currency['Greece']来获取与给定键相关联的值，而命令 currency['Greece'] = 'Drachma'将其重新映射到新值。与标准数组不同，map 的下标不必是连续的，也不用是数字。地图的常见应用包括以下内容。

- A university's information system relies on some form of a student ID as a key that is mapped to that student's associated record (such as the student's name, address, and course grades) serving as the value.
  ＊大学的信息系统，学生 ID 是 key，学生的相关信息是 value（如学生姓名，地址，课程成绩）

- The domain-name system (DNS) maps a host name, such as www.wiley.com, to an Internet-Protocol (IP) address, such as 208.215.179.146.
  ＊域名系统（DNS）将主机名（例如 www.wiley.com）映射到互联网协议（IP）地址，例如 208.215.179.146。

- A social media site typically relies on a (nonnumeric) username as a key that can be efficiently mapped to a particular user's associated information.
  ＊社交媒体网站通常使用（非数字）用户名作为能够有效映射到特定用户相关信息的 key。

- A computer graphics system may map a color name, such as turquoise , to the triple of numbers that describes the color's RGB (red-green-blue) representation, such as (64, 224, 208).
  ＊计算机图形系统可以将颜色名称映射到描绘颜色的 RGB（红 - 绿 - 蓝）表示的三元数字，例如（64,224,208）。

- Python uses a dictionary to represent each namespace, mapping an identifying string, such as 'pi' , to an associated object, such as 3.14159.
  ＊Python 使用字典来表示每个命名空间，将诸如“pi”的标识字符串映射到关联对象，如 3.14159。

In this chapter and the next we demonstrate that a map may be implemented so that a search for a key, and its associated value, can be performed very efficiently, thereby supporting fast lookup in such applications.
＊在本章和下一章中，我们将要展示 map 的实现，从而可以非常有效地执行搜索密钥及其相关联的值，从而支持在上述这些应用程序中的快速查找。

10.1.1 The Map ADT
＊映射的抽象数据类型

In this section, we introduce the ***map ADT***, and define its behaviors to be consistent with those of Python's built-in dict class. We begin by listing what we consider the most significant five behaviors of a map *M* as follows:
＊在本节中，我们介绍 map 的 ADT，将其行为做得与 Python 内置 dict 类的行为一致。我们首先列出我们认为 map M 最重要的五个行为：

M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError. In Python, this is implemented with the special method __getitem__.
＊如果存在，就返回与地图 M 中的关键字 k 相关联的值 v；否则会引发 KeyError。在 Python 中，这是用__getitem__的特殊方法实现的。

M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k. In Python, this is implemented with the special method __setitem__.
＊将值 v 与 M 中的关键字 k 相关联，如果地图已经包含密钥等于 k 的项目，则替换现有值。在 Python 中，这是用__setitem__的特殊方法实现的。

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError. In Python, this is implemented with the special method __delitem__.
＊从 map 上移除 k 的关联值 M[k]；如果没有，则引发 KeyError。在 Python 中，这是使用__delitem__的特殊方法实现的。

len(M): Return the number of items in map M. In Python, this is implemented with the special method __len__.
＊返回 M 中的项目数。在 Python 中，这是使用特殊方法__len__实现的。

iter(M): The default iteration for a map generates a sequence of *keys* in the map. In Python, this is implemented with the special method __iter__ , and it allows loops of the form, **for** k **in** M.
＊这是 map 的默认迭代，会生成 key 的一个序列。在 Python 中，这是用__iter__的特殊方法实现的，它允许 for 循环：for k in M。

We have highlighted the above five behaviors because they demonstrate the core functionality of a map—namely, the ability to query, add, modify, or delete a key-value pair, and the ability to report all such pairs. For additional convenience, map *M* should also support the following behaviors:
＊我们已经叙述以上 5 种行为，因为它们是 map 的核心功能，即查询，添加，修改或删除键值对，以及报告所有这些对的能力。为了更好的适应性，M 还应该支持以下行为：

k in M: Return True if the map contains an item with key k. In Python, this is implemented with the special __contains__ method.
＊如果 map 包含带有键 k 的项，则返回 True。在 Python 中，这是通过特定的__contains__方法实现的。

M.get(k, d=None): Return M[k] if key k exists in the map; otherwise return default value d. This provides a form to query M[k] without risk of a KeyError.
＊如果地图中存在 key k，返回 M[k]否则返回默认值 d。这提供了一个查询 M[k]的方法，而不会有 KeyError 的风险。

M.setdefault(k, d): If key k exists in the map, simply return M[k]; if key k does not exist, set M[k] = d and return that value.
＊如果地图中存在 key k，只需返回 M[k]；如果键不存在，则令 M[k] = d 并返回 d。

M.pop(k, d=None): Remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d (or raise KeyError if parameter d is None).
＊从 map 中删除与关键字 k 相关联的项目，并返回其关联的值 v。如果关键字 k 不在映射中，则返回默认值 d（如果参数 d 未说明），则返回默认值。

M.popitem(): Remove an arbitrary key-value pair from the map, and return a (k, v) tuple representing the removed pair. If map is empty, raise a KeyError.
＊从 map 中删除一个任意键值对(k, v)，并返回(k, v)。如果 map 为空，则引发 KeyError。

M.clear(): Remove all key-value pairs from the map.
＊从 map 中删除所有键值对。

M.keys(): Return a set-like view of all keys of M.
＊返回 M 的所有键的集合

M.values(): Return a set-like view of all values of

M.
*返回所有值的集合。

M.items(): Return a set-like view of (k, v) tuples for all entries of M.
*返回所有条目的(k, v)元组的集合。

M.update(M2): Assign M[k] = v for every (k, v) pair in map M2.
*以 M2 为标准更新 M

M == M2: Return True if maps M and M2 have identical key-value associations.
*如果 M 和 M2 具有相同的键值关联，返回 True。

M != M2: Return True if maps M and M2 do not have identical key-value associations.
*如果 M 和 M2 没有相同的键值关联，返回 True。

Example 10.1: In the following, we show the effect of a series of operations on an initially empty map storing items with integer keys and single-character values. We use the literal syntax for Python's dict class to describe the map contents.
*示例 10.1：在下面，我们将显示一系列操作对空映射的影响，该映射将存储整数 key 和单字符 value。我们使用 Python 的 dict 类的语法来描述 map。

Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
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 Application: Counting Word Frequencies
＊应用：词频统计

As a case study for using a map, consider the problem of counting the number of occurrences of words in a document. This is a standard task when performing a statistical analysis of a document, for example, when categorizing an email or news article. A map is an ideal data structure to use here, for we can use words as keys and word counts as values. We show such an application in Code Fragment 10.1.
＊作为使用 map 的一个案例研究，请考虑计算文档中单词出现次数的问题。执行文档的统计分析时，例如在分类电子邮件或新闻文章时，这是一项正式而有用的任务。map 是理想数据结构，因为我们可以将字作为 key，将频数作为 value。我们在 Code Fragment 10.1 中展示了这样一个应用程序。

We break apart the original document using a combination of file and string methods that results in a loop over a lowercased version of all whitespace separated pieces of the document. We omit all nonalphabetic characters so that parentheses, apostrophes, and other such punctuation are not considered part of a word.
＊我们使用文件和字符串方法的组合形式来分离原始文档，经过这样的一个循环，可以使文档中用空格分隔的单词被分离处理。我们省略所有非字母字符，以便括号，撇号和其他此类标点符号不被视为单词的一部分。

In terms of map operations, we begin with an empty Python dictionary named freq. During the first phase of the algorithm, we execute the command
＊在 map 操作方面，我们从空的 Python 字典开始，命名为 freq。在算法的第一阶段，我们执行命令

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

for each word occurrence. We use the get method on the right-hand side because the current word might not exist in the dictionary; the default value of 0 is appropriate in that case.
＊来记录每个单词的频数增加。因为当前单词可能不存在于字典中，所以我们要使用 get 方法；在这种情况下，将 value 的默认值设置为 0 是合适的。

During the second phase of the algorithm, after the full document has been processed, we examine the contents of the frequency map, looping over freq.items() to determine which word has the most occurrences.
＊在算法的第二阶段，在完整的文档被处理之后，我们检查频率图的内容，循环使用 freq.items()来确定哪个字最多出现。

10.1.3 Python's MutableMapping Abstract Base Class
＊Python 的 MutableMapping 抽象基类

Section 2.4.3 provides an introduction to the concept of an ***abstract base class*** and the role of such classes in Python's collections module. Methods that are declared to be abstract in such a base class must be implemented by concrete subclasses. However, an abstract base class may provide concrete implementation of other methods that depend upon use of the presumed abstract methods. (This is an example of the ***template method design pattern***.)
＊2.4.3 节介绍了抽象基类的概念以及这些类在 Python 集合模块中的作用。在这样一个基类中，被声明为抽象的方法必须由具体子类实现。然而，抽象基类可以提供一些其他方法的具体实现，而这些方法有可能被后期用到。（这是模板方法设计模式的一个例子。）

The collections module provides two abstract base classes that are relevant to our current discussion: the Mapping and MutableMapping classes. The Mapping class includes all nonmutating methods supported by Python's dict class, while the MutableMapping class extends that to include the mutating methods. What we define as the map ADT in Section 10.1.1 is akin to the MutableMapping abstract base class in Python's collections module.
＊collections 模块提供了与我们当前讨论相关的两个抽象基础组件：Mapping 和 MutableMapping 类。Mapping 类包含 Python 中 dict 类支持的所有不可变方法，而 MutableMapping 类则包含了所有的可变方法。10.1.1 节中的 ADT 类似于 Python 集合模块中的 MutableMapping 抽象基类。

The significance of these abstract base classes is that they provide a framework to assist in creating a user-defined map class. In particular, the MutableMapping class provides *concrete* implementations for all behaviors other than the first five outlined in Section 10.1.1: __getitem__, __setitem__, __delitem__, __len__, and __iter__. As we implement the map abstraction with various data structures, as long as we provide the five core behaviors, we can inherit all other derived behaviors by simply declaring MutableMapping as a parent class.
＊这些抽象基类的重要性在于它们提供了一个框架，用来协助创建用户定义的 map 类。特别地，MutableMapping 类提供了除了第 10.1.1 节中介绍的第一个以外的所有行为的具体实现：__getitem__，__setitem__，__delitem__，__len__和__iter__。当我们实现具有各种数据结构的 map 抽象时，只要我们提供五个核心行为，我们可以通过简单地将 MutableMapping 声明为父类来继承所有其他派生行为。

To better understand the MutableMapping class, we provide a few examples of how concrete behaviors can be derived from the five core abstractions. For example, the __contains__ method, supporting the syntax k in M, could be implemented by making a guarded attempt to retrieve self[k] to determine if the key exists.
＊为了更好地了解 MutableMapping 类，我们提供了一些例子，说明从五个核心抽象中可以得到具体的行为。例如，k in M 的__contains__方法可以通过 try self[k]来实现。

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

A similar approach might be used to provide the logic of the setdefault method.
＊可以使用类似的方法来提供 setdefault 方法的逻辑。

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

We leave as exercises the implementations of the remaining concrete methods of the MutableMapping class.
＊MutableMapping 类剩余的具体方法留作练习。

10.1.4 Our MapBase Class

*

We will be providing many different implementations of the map ADT, in the remainder of this chapter and next, using a variety of data structures demonstrating a trade-off of advantages and disadvantages. Figure 10.2 provides a preview of those classes.

*我们将在本章的其余部分中提供许多不同的 map ADT 的实现，并使用各种数据结构来展示优缺点。图 10.2 提供了这些类的预览。

The MutableMapping abstract base class, from Python's collections module and discussed in the preceding pages, is a valuable tool when implementing a map. However, in the interest of greater code reuse, we define our own MapBase class, which is itself a subclass of the MutableMapping class. Our MapBase class provides additional support for the composition design pattern. This is a technique we introduced when implementing a priority queue (see Section 9.2.1) in order to group a key-value pair as a single instance for internal use.

*来自 Python collections 模块的 MutableMapping 抽象基类在前面的页面中讨论过，是实现映射的有价值的工具。然而，为了更好地进行代码重用，我们定义了我们自己的 MapBase 类，它本身就是 MutableMapping 类的一个子类。我们的 MapBase 类为组合设计模式提供了额外的支持。这是我们在实现优先级队列时引入的技术（见第 9.2.1 节），以便将键值对分组为单个实例用于内部使用。

More formally, our MapBase class is defined in Code Fragment 10.2, extending the existing MutableMapping abstract base class so that we inherit the many useful concrete methods that class provides. We then define a nonpublic nested _Item class, whose instances are able to store both a key and value. This nested class is reasonably similar in design to the Item class that was defined within our PriorityQueueBase class in Section 9.2.1, except that for a map we provide support for both equality tests and comparisons, both of which rely on the item's key. The notion of equality is necessary for all of our map implementations, as a way to determine whether a key given as a parameter is equivalent to one that is already stored in the map. The notion of comparisons between keys, using the < operator, will become relevant when we later introduce a ***sorted map*** ADT (Section 10.3).

*更正式地，我们的 MapBase 类定义在 Code Fragment 10.2 中，扩展了现有的 MutableMapping 抽象基类，以便我们继承 MutableMapping 提供的许多有用的函数。然后，我们定义一个非公开的嵌套类_Item，它的实例能够存储一个 k, v 对。这个嵌套类同 9.2.1 节中的 PriorityQueueBase 类中定义的 Item 类一样，在设计中是合理的，除了 map 需要相等测试与比较之外，这两者都依赖于 Item 中的 key。对于我们所有的 map 实现来说，相等的概念是必要的，它可以确认我们给出的 k 是否在 map 里面已经存在。key 之间的对比也是重要的，我们使用<符号进行操作，这个操作与我们接下来给出的 sorted map ADT 相关。

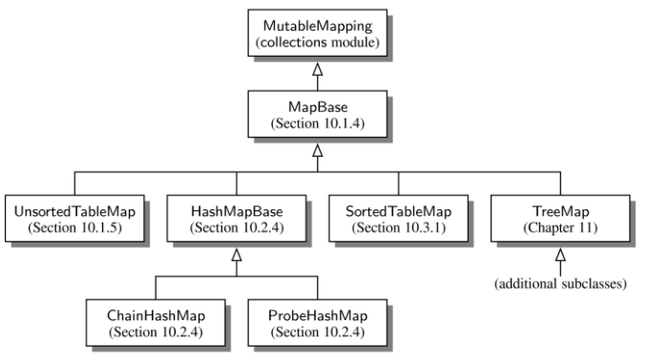


Figure 10.2: Our hierarchy of map types (with references to where they are defined).

10.1.5 Simple Unsorted Map Implementation *

We demonstrate the use of the MapBase class with a very simple concrete implementation of the map ADT. Code Fragment 10.3 presents an UnsortedTableMap class that relies on storing key-value pairs in arbitrary order within a Python list.

*我们演示了使用 MapBase 类与 map ADT 的一个非常简单的具体实现。代码片段 10.3 提供了一个 UnsortedTableMap 类，它依赖于在 Python 列表中以任意顺序存储键值对。

An empty table is initialized as self._table within the constructor for our map. When a new key is entered into the map, via line 22 of the __setitem__ method, we create a new instance of the nested Item class, which is inherited from our MapBase class.

*self._table 是一个空的表。当通过__setitem__方法的第 22 行将新密钥输入到地图中时，我们创建一个嵌套 Item 类的继承自 MapBase 类的新实例。

This list-based map implementation is simple, but it is not particularly efficient. Each of the fundamental methods, __getitem__, __setitem__, and __delitem__, relies on a for loop to scan the underlying list of items in search of a matching key. In a best-case scenario, such a match may be found near the beginning of the list, in which case the loop terminates; in the worst case, the entire list will be examined. Therefore, each of these methods runs in $O(n)$ time on a map with n items.

*这种基于列表的地图实现很简单，但并不是特别有效。每个基本方法__getitem__，__setitem__和__delitem__都依赖于 for 循环来扫描项目的底层列表，以搜索匹配的键。在最佳情况下，可能会在列表开头附近找到这样的匹配，在这种情况下，循环终止；在最坏的情况下，需要遍历整个列表。因此，这些方法中的时间复杂度是 O(n)。

10.2 Hash Tables

＊哈希表

In this section, we introduce one of the most practical data structures for implementing a map, and the one that is used by Python's own implementation of the dict class. This structure is known as a ***hash table***.

＊在本节中，我们介绍实现 map 的最有用的数据结构之一，而 Python 的 dict 类的实现也正是基于这个结构。这就是哈希表。

Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k]. As a mental warm-up, consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to $N - 1$ for some $N \geq n$. In this case, we can represent the map using a ***lookup table*** of length N, as diagrammed in Figure 10.3.

＊直观来看，映射 M 支持使用 k 映射到 M[k]。提起做个心理准备，我们考虑一个受限制的情况，map 里面有 n 个条目，所有的 key 都是取自 0 到 N - 1 的整数，当然，N > n。 在这种情况下，我们可以使用长度为 N 的查找表来表示地图，如图 10.3 所示。

Figure 10.3: A lookup table with length 11 for a map containing items (1,D), (3,Z), (6,C), and (7,Q).

In this representation, we store the value associated with key k at index k of the table (presuming that we have a distinct way to represent an empty slot). Basic map operations of __getitem__, __setitem__, and __delitem__ can be implemented in $O(1)$ worst-case time.

＊在这种表示中，我们将与 k 相关联的值存储在表的索引 k 处（假设我们有一种不同的方式来表示一个空位）。这样一来，我们就可以在最坏 O(1)的情况下实现__getitem__，__setitem__和__delitem__这些基本的操作。

There are two challenges in extending this framework to the more general set- ting of a map. First, we may not wish to devote an array of length N if it is the case that $N \gg n$. Second, we do not in general require that a map's keys be integers. The novel concept for a hash table is the use of a ***hash function*** to map general keys to corresponding indices in a table. Ideally, keys will be well distributed in the range from 0 to $N - 1$ by a hash function, but in practice there may be two or more distinct keys that get mapped to the same index. As a result, we will conceptualize our table as a ***bucket array***, as shown in Figure 10.4, in which each bucket may manage a collection of items that are sent to a specific index by the hash function. (To save space, an empty bucket may be replaced by None.)

＊将这个框架扩展到更一般的 map 有两个挑战。首先，如果是 N >> n 的话，这个长度为 N 的数组可能就不太好使。第二，我们通常不要求 map 的键是整数。哈希表的新颖概念是使用哈希函数将一般键映射到表中的对应索引。理想情况下，密钥将通过哈希函数在 0 到 N-1 的范围内良好分布，但实际上可能会有两个或更多个不同的密钥映射到相同索引的情况。因此，我们将我们的表格概念化为桶数组，如图 10.4 所示，其中每个桶可以管理通过散列函数发送到特定索引的项目集合。（要节省空间，可以使用 None 替换空桶。）

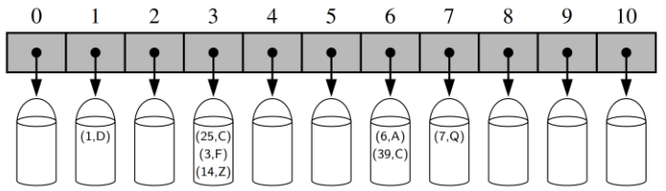


Figure 10.4: A bucket array of capacity 11 with items (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

10.2.1 Hash Functions

＊哈希函数

The goal of a ***hash function***, h, is to map each key k to an integer in the range $[0, N - 1]$, where N is the capacity of the bucket array for a hash table. Equipped with such a hash function, h, the main idea of this approach is to use the hash function value, $h(k)$, as an index into our bucket array, A, instead of the key k (which may not be appropriate for direct use as an index). That is, we store the item (k, v) in the bucket $A[h(k)]$.

＊哈希函数 h 的目标是将每个密钥 k 映射到范围[0，N-1]中的整数，其中 N 是哈希表的桶阵列的容量。配有这样一个哈希函数 h，这个方法的主要思想是使用哈希函数值 h(k)作为我们的数组 A 中的下标，而不是 k（可能不是适当的）直接用作下标。也就是说，我们将(k, v)存储在桶 A[h(k)]中。

If there are two or more keys with the same hash value, then two different items will be mapped to the same bucket in A. In this case, we say that a ***collision*** has occurred. To be sure, there are ways of dealing with collisions, which we will discuss later, but the best strategy is to try to avoid them in the first place. We say that a hash function is "good" if it maps the keys in our map so as to sufficiently minimize collisions. For practical reasons, we also would like a hash function to be fast and easy to compute.

＊如果有两个或更多个具有相同哈希值的密钥，则两个不同的项目将被映射到 A 中相同的 bucket 数组。在这种情况下，我们说发生了冲突。可以肯定的是，存在一些处理碰撞的方法，这个我们将在稍后讨论，但最好的策略是尽量避免在最初的时候产生冲突。我们说，一个函数能使得每一个 key 都能映射到数组里而且冲突最小，那么这个函数是"良性"的。出于实际原因，我们也希望这个哈希函数能够快速，容易地进行计算。

It is common to view the evaluation of a hash function, $h(k)$, as consisting of two portions—a ***hash code*** that maps a key k to

an integer, and a ***compression*** function that maps the hash code to an integer within a range of indices, [0, $N-1$], for a bucket array. (See Figure 10.5.)

＊通常，查看散列函数 h(k)的优良成都，通常看两部分：将密钥 k 映射到整数的哈希码，以及能将散列码映射到[0, N-1]范围内的整数的压缩函数（见图 10.5）

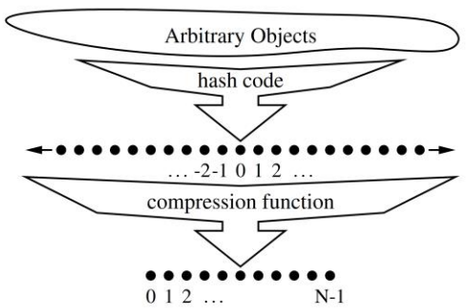


Figure 10.5: Two parts of a hash function: a hash code and a compression function.

The advantage of separating the hash function into two such components is that the hash code portion of that computation is independent of a specific hash table size. This allows the development of a general hash code for each object that can be used for a hash table of any size; only the compression function depends upon the table size. This is particularly convenient, because the underlying bucket array for a hash table may be dynamically resized, depending on the number of items currently stored in the map. (See Section 10.2.3.)

＊将散列函数分为两个这样的组件的优点是散列码计算部分与散列表大小无关。这允许为可用于任何大小的哈希表的每个对象开发通用散列码；而压缩功能取决于表的大小。所以可以根据当前存储在 map 中的项目数，使得散列表的底层桶数组可以动态地调整大小。（见 10.2.3 节）

Hash Codes

＊散列码

The first action that a hash function performs is to take an arbitrary key k in our map and compute an integer that is called the ***hash code*** for *k*; this integer need not be in the range [0, $N-1$], and may even be negative. We desire that the set of hash codes assigned to our keys should avoid collisions as much as possible. For if the hash codes of our keys cause collisions, then there is no hope for our compression function to avoid them. In this subsection, we begin by discussing the theory of hash codes. Following that, we discuss practical implementations of hash codes in Python.

＊哈希函数执行的第一个动作是在我们的 map 中取一个任意的关键字 k，并计算一个有关 k 的名为哈希码的整数；该整数不需要在[0, N-1]范围内，甚至可能为负数。我们希望分配给我们的哈希码集合尽可能避免冲突。因为如果我们的哈希码发生了冲突，那么我们的压缩函数就不可能避免它们。在本小节中，我们首先讨论散列码的理论。接下来，我们讨论 Python 中散列码的实际实现。

Treating the Bit Representation as an Integer

＊将二进制形式表示为整数

To begin, we note that, for any data type *X* that is represented using at most as many bits as our integer hash codes, we can simply take as a hash code for *X* an integer interpretation of its bits. For example, the hash code for key 314 could simply be 314. The hash code for a floating-point number such as 3.14 could be based upon an interpretation of the bits of the floating-point representation as an integer.

＊首先，我们注意到，对于任何数据类型 X，使用最多与整数散列码一样多的位，我们可以简单地将 X 作为其位的整数解释的哈希码。例如，密钥 314 的哈希码可以简单地为 314。诸如 3.14 之类的浮点数的散列码可以基于将小数点表示的位的解释作为整数哈希码。

For a type whose bit representation is longer than a desired hash code, the above scheme is not immediately applicable. For example, Python relies on 32-bit hash codes. If a floating-point number uses a 64-bit representation, its bits cannot be viewed directly as a hash code. One possibility is to use only the high-order 32 bits (or the low-order 32 bits). This hash code, of course, ignores half of the information present in the original key, and if many of the keys in our map only differ in these bits, then they will collide using this simple hash code.

＊对于其位表示长于哈希码的类型，上述方案不能立即适用。例如，Python 依赖于 32 位哈希码。如果浮点数使用 64 位表示，则其位不能直接视为哈希码。一个可能性是仅使用高阶 32 位（或低位 32 位）。当然，这个哈希码忽略了原始密钥中存在的信息的一半，如果我们的 map 中的许多键仅在这些没有使用到的位中有所不同，那么使用这个简单的哈希码将会产生冲突。

A better approach is to combine in some way the high-order and low-order portions of a 64-bit key to form a 32-bit hash code, which takes all the original bits into consideration. A simple implementation is to add the two components as 32- bit numbers (ignoring overflow), or to take the exclusive-or of the two components. These approaches of combining components can be extended to any object x whose binary representation can be viewed as an n-tuple $(x_0, x_1, ..., x_{n-1})$ of 32-bit integers, for example, by forming a hash code for x as $\sum_{i=0}^{n-1} x_i$, or as $x_0 \oplus x_1 + \ldots \oplus x_{n-1}$, where the $\oplus$ symbol represents the bitwise exclusive-or operation (which is ^ in Python).

＊更好的方法是以某种方式组合 64 位密钥的高阶部分和低阶部分以形成 32 位哈希码，这将考虑所有原始比特。一个简单的实现是将这两个高低位作为两个 32 位数字相加（忽略溢出），之后用这个和来取代这个 64 位数。这些方法可以扩展到其他的那种二进制表示可以被视为 32 位整数的 n 元组，如类似于二进制表示为$(x_0, x_1, ..., x_{n-1})$的任何对象 x，（其中 x_i 是按段取二进制之后的数字），可以把这些片段数字之和的哈希值作为原来数字的哈希值。而这个和可以通过异或运算轻易得到。

Polynomial Hash Codes

＊多项式形式的哈希码计算

The summation and exclusive-or hash codes, described above, are not good choices for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{n-1})$, where the order of the x_i's is significant. For example, consider a 16-bit hash code for a character string s that sums the Unicode values of the characters in *s*. This hash code unfortunately produces lots of unwanted collisions for common groups of strings. In particular, "temp01" and "temp10" collide using this function, as do "stop", "tops", "pots", and "spot". A better hash code should somehow take into consideration the positions of the xi's. An alternative hash code, which does exactly this, is to choose a nonzero constant, $a \neq 1$, and use as a hash code the value

＊上述的求和与异或运算类型的哈希码，对于可被视为形如$(x_0, x_1, \ldots, x_{n-1})$的元组的字符串或其他可变长度对象来说不是很好的选择，其中 x_i 的顺序是显然的。例如，考虑一个字符串 s 的 16 位哈希码，它与 s 中的字符的 Unicode 值相加。不幸的是，这个哈希码对于常见的字符串组产生了许多不必要的冲突。特别地，“temp01”和“temp10”会在这种情况下产生冲突与“stop”，“tops”，“pots”和“spot”一样也会在这种方式下产生冲突。一个更好的哈希码应该以某种方式考虑到 xi 的位置。一个替代的哈希码，正是这样做，就是选择一个非零常数，并用作哈希码的值

$$x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}$$

Mathematically speaking, this is simply a polynomial in a that takes the components $(x_0, x_1, \ldots, x_{n-1})$ of an object x as its coefficients. This hash code is therefore called a ***polynomial hash code***. By Horner's rule (see Exercise C-3.50), this polynomial can be computed as

＊在数学上，这只是一个多项式，它将一个对象 x 的组成成分$(x_0, x_1, \ldots, x_{n-1})$作为它的系数。因此，这个哈希代码被称为多项式哈希码。按照霍纳的规则（见练习 C-3.50），这个多项式可以计算为

$$x_{n-1} + a\big(x_{n-2} + a\big(x_{n-3} + \cdots + a\big(x_2 + a\big(x_1 + a x_0\big)\big)\big)\big).$$

Intuitively, a polynomial hash code uses multiplication by different powers as a way to spread out the influence of each component across the resulting hash code.

＊直观地看，多项式哈希码使用不同幂指数的乘法，将不同位置的成员的影响变得不同。

Of course, on a typical computer, evaluating a polynomial will be done using the finite bit representation for a hash code; hence, the value will periodically overflow the bits used for an integer. Since we are more interested in a good spread of the object x with respect to other keys, we simply ignore such overflows. Still, we should be mindful that such overflows are occurring and choose the constant *a* so that it has some nonzero, low-order bits, which will serve to preserve some of the information content even as we are in an overflow situation.

＊当然，在典型的计算机上，多项式的计算会通过有限位来进行；因此，该值将周期性地超过用于存储的字节数而发生溢出。因为我们对 x 相对于其他键的良好扩展更感兴趣，所以我们忽略这样的溢出。不过，我们应该注意到，这种溢出正在发生，我们一般选择一个常数，以便它具有一些非零，低阶位，即使我们处于一个溢出的情况，这也将有助于保留一些信息，。

We have done some experimental studies that suggest that 33, 37, 39, and 41 are particularly good choices for a when working with character strings that are English words. In fact, in a list of over 50,000 English words formed as the union of the word lists provided in two variants of Unix, we found that taking a to be 33, 37, 39, or 41 produced less than 7 collisions in each case!

＊我们做了一些实验研究，表明 33，37，39 和 41 在使用英文单词的字符串时是特别好的选择。事实上，在 Unix 的两个变体中提供的单词列表联合的 50,000 个英文单词的列表中，我们发现，在 33，37，39 或 41 中，每个单词中的每一个都产生少于 7 个的冲突！

Cyclic-Shift Hash Codes

＊循环移位哈希码

A variant of the polynomial hash code replaces multiplication by a with *a* cyclic shift of a partial sum by a certain number of bits. For example, a 5-bit cyclic shift of the 32-bit value 0011 1101 1001 0110 1010 1000 1010 1000 is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in 1011 0010 1101 0101 0001 0101 0000 0111. While this operation has little natural meaning in terms of arithmetic, it accomplishes the goal of varying the bits of the calculation. In Python, a cyclic shift of bits can be accomplished through careful use of the bitwise operators << and >>, taking care to truncate results to 32-bit integers.

＊多项式散列代码的一个变体是用二进制循环唯一来替代惩罚。例如，32 比特值 0011 1101 1001 0110 1010 1000 1010 1000 的 5 比特循环移位是通过取最左边的比特并将它们放置在右边，从而得到 1011 0010 1101 0101 0001 0101 0000 0111。虽然这种操作在算术方面几乎没有自然的意义，但它实现了改变计算位的目标。在 Python 中，可以通过使用按位运算符<<和>>来实现位的循环移位，注意将结果截断为 32 位整数。

An implementation of a cyclic-shift hash code computation for a character string in Python appears as follows:

＊Python 中字符串的循环移位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)
```

```
        h = (h << 5) | (h >> 27)          # 5-bit cyclic shift of running sum
        h += ord(character)               # add in value of next character
    return h
```

As with the traditional polynomial hash code, fine-tuning is required when using a cyclic-shift hash code, as we must wisely choose the amount to shift by for each new character. Our choice of a 5-bit shift is justified by experiments run on a list of just over 230,000 English words, comparing the number of collisions for various shift amounts (see Table 10.1).

*与传统的多项式散列码一样，当使用循环移位哈希码时，需要进行微调，因为我们必须明智地选择每个新字符移位的量。我们选择一个 5 位的位移是实验过 23 万英语单词之后的结果（见表 10.1）。

	Collisions	
Shift	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

Table 10.1: Comparison of collision behavior for the cyclic-shift hash code as applied to a list of 230,000 English words. The "Total" column records the total number of words that collide with at least one other, and the "Max" column records the maximum number of words colliding at any one hash code. Note that with a cyclic shift of 0, this hash code reverts to the one that simply sums all the characters.

Hash Codes in Python

*Python 中的哈希码

The standard mechanism for computing hash codes in Python is a built-in function with signature hash(x) that returns an integer value that serves as the hash code for object x. However, only ***immutable*** data types are deemed hashable in Python. This restriction is meant to ensure that a particular object's hash code remains constant during that object's lifespan. This is an important property for an object's use as a key in a hash table. A problem could occur if a key were inserted into the hash table, yet a later search were performed for that key based on a different hash code than that which it had when inserted; the wrong bucket would be searched.

*用于计算 Python 中哈希码的内建函数 hash(x)，会返回对象 x 的整数哈希码。但是，只有不可变数据类型在 Python 中被认为是可计算哈希码的。此限制旨在确保特定对象的哈希码在该对象的使用期间保持不变。这对与一个对象能用作键值而言有着重要意义。如果将密钥插入散列表，则可能会出现问题，但是基于与插入时不同的哈希码，为该密钥执行后续搜索；将搜索错误的桶。

Among Python's built-in data types, the immutable int, float, str, tuple, and frozenset classes produce robust hash codes, via the hash function, using techniques similar to those discussed earlier in this section. Hash codes for character strings are well crafted based on a technique similar to polynomial hash codes, except using exclusive-or computations rather than additions. If we repeat the experiment described in Table 10.1 using Python's built-in hash codes, we find that only 8 strings out of the set of more than 230,000 collide with another. Hash codes for tuples are computed with a similar technique based upon a combination of the hash codes of the individual elements of the tuple. When hashing a frozenset, the order of the elements should be irrelevant, and so a natural option is to compute the exclusive-or of the individual hash codes without any shifting. If hash(x) is called for an instance x of a mutable type, such as a list, a TypeError is raised.

*在 Python 的内置数据类型中，不可变的 int，float，str，tuple 和 frozenset 类通过哈希函数，使用类似于本节前面讨论的技术，可以生成很不错的哈希码。字符串的哈希代码基于类似于多项式哈希码，而这个多项式哈希码却是通过异或运算得到的而非是简单的相加。如果我们使用 Python 的内置哈希码重复表 10.1 中描述的实验，我们发现超过 23 万的集合中只有 8 个字符串与另一个冲突相冲突。元组的哈希码，也是基于每个元素的组合。计算 frozenset 的哈希码时，元素的顺序是无关紧要的，因此一个自然的选择是不加转换地用异或计算每一个单独元素的哈希码。如果为可变类型（例如列表）的实例 x 调用了哈希函数，则会引发 TypeError。

Instances of user-defined classes are treated as unhashable by default, with a TypeError raised by the hash function. However, a function that computes hash codes can be implemented in the form of a special method named __hash__ within a class. The returned hash code should reflect the immutable attributes of an in- stance. It is common to return a hash code that is itself based on the computed hash of the combination of such attributes. For example, a Color class that maintains three numeric red, green, and blue components might implement the method as:

*默认情况下，用户定义类的类无法直接使用 hash 函数，强行使用会引发的 TypeError。然而，计算哈希码的功能可以以类中名为__hash__的特殊方法的形式来实现。返回的哈希码应该反映不可变的属性。通常返回的时基于计算这些组合的哈希码。例如，一个存储着三个数字的 RGB 值的 Color 类可能会执行以下哈希方法：

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

An important rule to obey is that if a class defines equivalence through __eq__, then any implementation of __hash__

must be consistent, in that if x == y, then hash(x) == hash(y). This is important because if two instances are considered to be equivalent and one is used as a key in a hash table, a search for the second instance should result in the discovery of the first. It is therefore important that the hash code for the second match the hash code for the first, so that the proper bucket is examined. This rule extends to any well-defined comparisons between objects of different classes. For example, since Python treats the expression 5 == 5.0 as true, it ensures that hash(5) and hash(5.0) are the same.

*必须遵守一个重要的规则是，如果一个类通过__eq__定义等价性，则__hash__的任何实现必须是一致的，因为如果 x == y，则 hash(x)== hash(y)。这很重要，因为如果两个实例被认为是等效的，并且一个被用作哈希表中的一个密钥，那么对第二个实例的搜索将导致首先发现。因此，第二个哈希码与第一个哈希码匹配是非常重要的，因此选择适当的桶数组。如果两个来自不同类的实例之间也能比较，那么这个哈希函数就可以进行此方面的扩展。例如，由于 Python 将表达式 5 == 5.0 视为 true，它确保 hash(5)和 hash(5.0)是相同的。

Compression Functions

*压缩函数

The hash code for a key *k* will typically not be suitable for immediate use with a bucket array, because the integer hash code may be negative or may exceed the capacity of the bucket array. Thus, once we have determined an integer hash code for a key object *k*, there is still the issue of mapping that integer into the range [0, *N* - 1]. This computation, known as a ***compression function***, is the second action per- formed as part of an overall hash function. A good compression function is one that minimizes the number of collisions for a given set of distinct hash codes.

*密钥 k 的哈希码通常不适合立即与桶阵列一起使用，因为整数散列码可能是负的或可能超过桶阵列的容量。因此，一旦我们确定了密钥对象 k 的整数哈希码，仍然存在将该整数映射到范围[0，N 1]的问题。压缩函数是散列函数的第二个动作。良好的压缩函数，可以使给定的不同哈希码集合之间的冲突次数最小。

The Division Method

*整除方法

A simple compression function is the ***division method***, which maps an integer *i* to

*可以用整除方法来简单地实现压缩函数，具体是用整数 i 来做模运算

$$i \bmod N,$$

where *N*, the size of the bucket array, is a fixed positive integer.

Additionally, if we take N to be a prime number, then this compression function helps “spread out” the distribution of hashed values. Indeed, if *N* is not prime, then there is greater risk that patterns in the distribution of hash codes will be repeated in the distribution of ash values, thereby causing collisions. For example, if we insert keys with hash codes {200, 205, 210, 215, 220, ... , 600} into a bucket array of size 100, then each hash code will collide with three others. But if we use a bucket array of size 101, then there will be no collisions. If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is 1/*N*. Choosing *N* to be a prime number is not always enough, however, for if there is a repeated pattern of hash codes of the form $pN + q$ for several different *p*’s, then there will still be collisions.

*其中桶阵列的大小 N 是固定的正整数。另外，如果我们把 N 作为一个素数，这个压缩函数就有助于“散布”散列值的分布。实际上，如果 N 不是素数，很有可能引起冲突。例如，如果我们将具有散列码{200, 205, 210, 215, 220, ..., 600}的密钥插入到 N 为 100 的桶阵列中，那么每个散列码将与另外三个哈希码相冲突。但是如果我们使用大小为 101 的 bucket 数组，那么就不会有任何碰撞。如果哈希函数选择得很好，那么应该确保两个不同的密钥在同一个桶上散列的概率是 1 / N。 然而，选择 N 作为素数并不总是足够的，因为如果有几个形式为 pN + q 的哈希码，在这种情况下，则仍然存在冲突。

The MAD Method

*MAD 方法

A more sophisticated compression function, which helps eliminate repeated patterns in a set of integer keys, is the Multiply-Add-and-Divide (or “MAD”) method. This method maps an integer *i* to

*Multiply-Add-and-Divide（或“MAD”）方法的更复杂，有助于消除一组整数键中的重复模式。该方法将整数 i 映射到

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

where *N* is the size of the bucket array, *p* is a prime number larger than *N*, and *a* and *b* are integers chosen at random from the interval $[0, p - 1]$, with $a > 0$. This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and get us closer to having a “good” hash function, that is, one such that the probability any two different keys collide is 1 / *N*. This good behavior would be the same as we would have if these keys were “thrown” into *A* uniformly at random.

*其中 N 是桶阵列的大小，p 是大于 N 的素数，a 和 b 是来自区间[0，p-1]的任意常数，其中 a > 0。为了消除哈希码集合中的重复模式，选择该压缩函数，并使我们更接近于获得“良性”散列函数，也就是说，两个不同的键碰撞的概率是 1 / N。这将与我们将这些键随机“均匀地”投入到 A 中的情况相同。

10.2.2 Collision-Handling Schemes
＊冲突处理机制

The main idea of a hash table is to take a bucket array, A, and a hash function, h, and use them to implement a map by storing each item (k, v) in the "bucket" $A[h(k)]$. This simple idea is challenged, however, when we have two distinct keys, k_1 and k_2, such that $h(k_1) = h(k_2)$. The existence of such collisions prevents us from simply inserting a new item (k, v) directly into the bucket $A[h(k)]$. It also complicates our procedure for performing insertion, search, and deletion operations.

＊散列表的主要思想是采用一个桶阵列 A 和一个散列函数 h，将每个项(k, v)存储在 $A[h(k)]$中。然而，当我们有两个不同的键 k_1 和 k_2，使得 $h(k_1) = h(k_2)$时，这个简单的想法就遇到了麻烦。这种碰撞的存在阻止了我们将一个新项目(k, v)直接插入到 $A[h(k)]$中。这也使插入，搜索和删除操作的过程变得复杂。

Separate Chaining
＊独立链

A simple and efficient way for dealing with collisions is to have each bucket $A[j]$ store its own secondary container, holding items (k, v) such that $h(k) = j$. A natural choice for the secondary container is a small map instance implemented using a list, as described in Section 10.1.5. This ***collision resolution*** rule is known as ***separate chaining***, and is illustrated in Figure 10.6.

＊处理冲突的一个简单而有效的方法是让每个 A[j]存储一个的二次容器，用以保存(k, v)，使得 h(k) = j。辅助容器的一个自然选择是使用列表实现的小型映射实例，如第 10.1.5 节所述。这种冲突解决规则被称为单独链接，如图 10.6 所示。

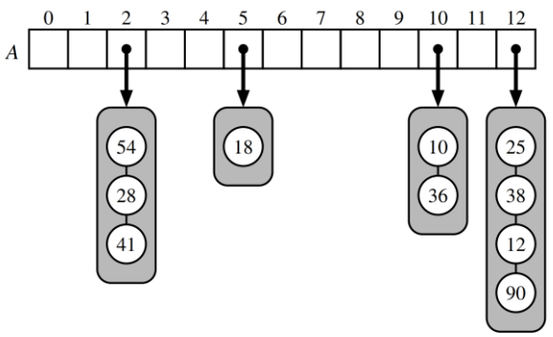


Figure 10.6: A hash table of size 13, storing 10 items with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

In the worst case, operations on an individual bucket take time proportional to the size of the bucket. Assuming we use a good hash function to index the n items of our map in a bucket array of capacity N, the expected size of a bucket is n / N. Therefore, if given a good hash function, the core map operations run in $O(\lceil n / N \rceil)$. The ratio $\lambda = n / N$, called the ***load factor*** of the hash table, should be bounded by a small constant, preferably below 1. As long as λ is $O(1)$, the core operations on the hash table run in $O(1)$ expected time.

＊在最糟糕的情况下，单个桶上的操作取决于桶的尺寸。假设我们使用一个好的散列函数来索引一个容量为 N 的桶数组中的 n 个项，那么桶的预期大小是 n / N。因此，如果给出了一个很好的散列函数，核心操作将为 $O(\lceil n / N \rceil)$量级。散列表的负载因子 $\lambda = n / N$ 应该被一个小常数限制，最好在 1 以下。只要λ是$O(1)$，散列表上的核心操作就在$O(1)$）预计时间之内完成。

Open Addressing
＊开放地址

The separate chaining rule has many nice properties, such as affording simple implementations of map operations, but it nevertheless has one slight disadvantage: It requires the use of an auxiliary data structure—a list—to hold items with colliding keys. If space is at a premium (for example, if we are writing a program for a small handheld device), then we can use the alternative approach of always storing each item directly in a table slot. This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to deal with collisions. There are several variants of this approach, collectively referred to as ***open addressing*** schemes, which we discuss next. Open addressing requires that the load factor is always at most 1 and that items are stored directly in the cells of the bucket array itself.

＊separate chaining 规则具有许多不错的属性，例如提供 map 操作的简单实现，但是它仍然存在一个轻微的缺点：它需要使用辅助数据结构（列表）来保存具有冲突的项目。如果空间有限（例如，如果我们正在为小型手持设备编写程序），那么我们要用直接将每个项目存储在列表中的方法。这种方法节省空间，因为没有辅助结构被使用，但是它需要更复杂的处理碰撞。这种方法有几种变体，它们被统一称为开放式寻址方案，下面我们将讨论。开放寻址要求负载因子始终为 1，并且项目直接存储在桶阵列本身的单元格中。

Linear Probing and Its Variants
＊线性探测及其变体

A simple method for collision handling with open addressing is ***linear probing***. With this approach, if we try to insert an item (k, v) into a bucket $A[]$ that is already occupied, where $j = h(k)$, then we next try $A[(j + 1) \bmod N]$. If $A[(j + 1) \bmod N]$ is also occupied, then we try $A[(j + 2) \bmod N]$, and so on, until we find an empty bucket that can accept the new item. Once this bucket is located, we simply insert the item there. Of course, this collision resolution strategy requires that we change the implementation when searching for an existing key—the first step of all __getitem__, __setitem__, or __delitem__ operations. In particular, to attempt to locate an item with key equal to k, we must examine consecutive slots, starting from $A[h(k)]$, until we either find an item with that key or we find an empty bucket. (See Figure 10.7.) The name "linear probing" comes from the fact that accessing a cell of the bucket array can be viewed as a "probe."

＊使用开放寻址的一种简单的碰撞处理方法是线性探测。使用这种方法，如果我们尝试将一个项目(k, v)插入已经存在的桶 A[]中，其中 j = h(k)，那么我们接着尝试 A[(j + 1) mod N]。如果 A[(j + 1) mod N]也被占用，那么我们尝试 A[(j + 2) mod N]等等，直到找到一个可以接受新项目的空桶。一旦这个位置是空的，那么我们就在这里插入项目。当然，这种碰撞解决策略要求我们在搜索现有密钥时改变实现__getitem__，__setitem__以及__delitem__操作的第一步。特别地，为了试图定位具有等于 k 的密钥的项目，我们必须检查从 A[h(k)]开始的连续时隙，直到找到具有该密钥的项目进而停止或者继续直至找到一个空桶。（见图 10.7）“线性探测”这个名字来自于探测性地访问桶阵列的单元这一背景。

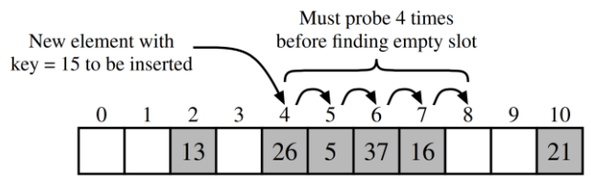


Figure 10.7: Insertion into a hash table with integer keys using linear probing. The hash function is $h(k) = k \bmod 11$. Values associated with keys are not shown.

To implement a deletion, we cannot simply remove a found item from its slot in the array. For example, after the insertion of key 15 portrayed in Figure 10.7, if the item with key 37 were trivially deleted, a subsequent search for 15 would fail because that search would start by probing at index 4, then index 5, and then index 6, at which an empty cell is found. A typical way to get around this difficulty is to replace a deleted item with a special “available” marker object. With this special marker possibly occupying spaces in our hash table, we modify our search algorithm so that the search for a key *k* will skip over cells containing the available marker and continue probing until reaching the desired item or an empty bucket (or returning back to where we started from). Additionally, our algorithm for __setitem__ should remember an available cell encountered during the search for *k*, since this is a valid place to put a new item (*k*, *v*), if no existing item is found.

＊要实现删除，我们不能简单地从数组中中删除找到的项。例如，在插入 15 之后，如果 37 被删除，则后续的搜索 15 将失败，因为搜索在索引 4 开始，然后是索引 5，然后索引 6，找到一个空单元格（被删掉的 37）。解决困难的一个典型方法是用特殊的“可用”标记对象去替换已删除的项目。使用这个特殊标记可以在我们的哈希表中占据空间，我们修改我们的搜索算法，以便搜索关键字 k 的时候将跳过包含可用标记的单元格，并继续探测，直到达到所需的项目或空的（或返回到我们开始的地方）。另外，__setitem__的算法应该记住在搜索 k 期间遇到的可用单元格，因为如果没有找到现有的 key 为 k 的项，那么“可用单元格”就是放置新项目(k, v)的有效位置。

Although use of an open addressing scheme can save space, linear probing suffers from an additional disadvantage. It tends to cluster the items of a map into contiguous runs, which may even overlap (particularly if more than half of the cells in the hash table are occupied). Such contiguous runs of occupied hash cells cause searches to slow down considerably.

＊尽管使用开放式寻址方案可以节省空间，但线性探测也会受到其他不利影响。它倾向于将 map 的项目聚类成连续的样式，甚至可能重叠（特别是如果散列表中的一半以上的单元格被占用）。占用的散列单元的连续运行导致搜索明显运行缓慢。

Another open addressing strategy, known as ***quadratic probing***, iteratively tries the buckets $A[(h(k) + f(i)) \bmod N]$, for $i = 0, 1, 2, \ldots$, where $f(i) = i^2$, until finding an empty bucket. As with linear probing, the quadratic probing strategy complicates the removal operation, but it does avoid the kinds of clustering patterns that occur with linear probing. Nevertheless, it creates its own kind of clustering, called ***secondary clustering***, where the set of filled array cells still has a non-uniform pattern, even if we assume that the original hash codes are distributed uniformly. When *N* is prime and the bucket array is less than half full, the quadratic probing strategy is guaranteed to find an empty slot. However, this guarantee is not valid once the table becomes at least half full, or if *N* is not chosen as a prime number; we explore the cause of this type of clustering in an exercise (C-10.36).

＊另一个开放寻址策略称为二次探测，它迭代地尝试 A(h(k) + f(i))，其中 f（i）= i2，i = 0,1,2, ...，直到找到一个空桶。与线性探测一样，二次探测策略使删除操作复杂了，但它避免了线性探测发生的聚类模式。然而，它创建了自己的聚类，称为二次聚类，其中填充的阵列单元集仍然具有不均匀的模式，即使我们假设原始哈希码是均匀分布的。当 N 是素数并且桶阵列尚未半满时，二次探测策略保证能找到一个空的时隙。但是，一旦该表满一半之后，或者如果不选择 N 作为素数，则此保证无效；我们在练习中进行探索（C-10.36）。

An open addressing strategy that does not cause clustering of the kind produced by linear probing or the kind produced by quadratic probing is the ***double hashing*** strategy. In this approach, we choose a secondary hash function, *h'*, and if *h* maps some key *k* to a bucket $A[h(k)]$ that is already occupied, then we iteratively try the buckets $A[(h(k) + f(i)) \bmod N]$ next, for $i = 1, 2, 3, \ldots$, where $f(i) = i \cdot h'(k)$. In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is $h'(k) = q - (k \bmod q)$, for some prime number $q < N$. Also, *N* should be a prime.

＊如果采用双重哈希策略，那么我们可以避免向线性探测或者二次探测那样的扎堆现象。在这种方法中，我们选择二次哈希函数 *h'*，如果 *h* 将一些密钥 *k* 映射到已经被占用的存储区 $A[h(k)]$，则我们尝试使用 $A[(h(k) + f(i)) \bmod N]$，其中 $f(i) = i \cdot h'(k)$，i = 1, 2, 3, ...。在这个方案中，二次哈希函数不允许值为零；对于某些素数 q < N，常用的选择是 h'(k) = q- (k mod q)。另外，N 应该是素数。

Another approach to avoid clustering with open addressing is to iteratively try buckets $A[(h(k)+f(i)) \bmod N]$ where $f(i)$ is based on a pseudo-random number generator, providing a repeatable, but somewhat arbitrary, sequence of subsequent probes that depends upon bits of the original hash code. This is the approach currently used by Python's dictionary class.

⋆另一个可以避免扎堆的方法是使用 $A[(h(k)+f(i)) \bmod N]$，f(i)由伪随机数生成器生成。这是 Python 的字典类目前使用的方法。

10.2.3 Load Factors, Rehashing, and Efficiency

In the hash table schemes described thus far, it is important that the load factor, $\lambda = n / N$, be kept below 1. With separate chaining, as λ gets very close to 1, the probability of a collision greatly increases, which adds overhead to our operations, since we must revert to linear-time list-based methods in buckets that have collisions. Experiments and average-case analyses suggest that we should maintain $\lambda < 0.9$ for hash tables with separate chaining.

With open addressing, on the other hand, as the load factor λ grows beyond 0.5 and starts approaching 1, clusters of entries in the bucket array start to grow as well. These clusters cause the probing strategies to "bounce around" the bucket array for a considerable amount of time before they find an empty slot. In Exercise C-10.36, we explore the degradation of quadratic probing when λ 0.5. Experiments suggest that we should maintain $\lambda < 0.5$ for an open addressing scheme with linear probing, and perhaps only a bit higher for other open addressing schemes (for ex- ample, Python's implementation of open addressing enforces that $\lambda < 2/3$).

If an insertion causes the load factor of a hash table to go above the specified threshold, then it is common to resize the table (to regain the specified load factor) and to reinsert all objects into this new table. Although we need not define a new hash code for each object, we do need to reapply a new compression function that takes into consideration the size of the new table. Each rehashing will generally scatter the items throughout the new bucket array. When rehashing to a new table, it is a good requirement for the new array's size to be at least double the previous size. Indeed, if we always double the size of the table with each rehashing operation, then we can amortize the cost of rehashing all the entries in the table against the time used to insert them in the first place (as with dynamic arrays; see Section 5.3).

Efficiency of Hash Tables

Although the details of the average-case analysis of hashing are beyond the scope of this book, its probabilistic basis is quite intuitive. If our hash function is good, then we expect the entries to be uniformly distributed in the N cells of the bucket array. Thus, to store n entries, the expected number of keys in a bucket would be ⌈n/N1, which is O(1) if n is O(N).

The costs associated with a periodic rehashing, to resize a table after occasional insertions or deletions can be accounted for separately, leading to an additional O(1) amortized cost for setitem and getitem .

In the worst case, a poor hash function could map every item to the same bucket. This would result in linear-time performance for the core map operations with separate chaining, or with any open addressing model in which the secondary sequence of probes depends only on the hash code. A summary of these costs is given in Table 10.2.

Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

Table 10.2: Comparison of the running times of the methods of a map realized by means of an unsorted list (as in Section 10.1.5) or a hash table. We let n denote the number of items in the map, and we assume that the bucket array supporting the hash table is maintained such that its capacity is proportional to the number of items in the map.

In practice, hash tables are among the most efficient means for implementing a map, and it is essentially taken for granted by programmers that their core operations run in constant time. Python's dict class is implemented with hashing, and the Python interpreter relies on dictionaries to retrieve an object that is referenced by an identifier in a given namespace. (See Sections 1.10 and 2.5.) The basic command c = a + b involves two calls to getitem in the dictionary for the local namespace to retrieve the values identified as a and b, and a call to setitem to store the result associated with name c in that namespace. In our own algorithm analysis, we simply presume that such dictionary operations run in constant time, independent of the number of entries in the namespace. (Admittedly, the number of entries in a typical namespace can almost surely be bounded by a constant.)

In a 2003 academic paper [31], researchers discuss the possibility of exploiting a hash table's worst-case performance to cause a denial-of-service (DoS) attack of Internet technologies. For many published algorithms that compute hash codes, they note that an attacker could precompute a very large number of moderate-length strings that all hash to the identical 32-bit hash code. (Recall that by any of the hashing schemes we describe, other than double hashing, if two keys are mapped to the same hash code, they will be inseparable in the collision resolution.)

In late 2011, another team of researchers demonstrated an implementation of just such an attack [61]. Web servers allow a series of key-value parameters to be embedded in a URL using a syntax such as ?key1=val1&key2=val2&key3=val3. Typically, those key-value pairs are immediately stored in a map by the server, and a limit is placed on the length and number of such parameters presuming that storage time in the map will be linear in the number of entries. If all keys were to collide, that storage requires quadratic time (causing the server to perform an inordinate amount of work). In spring of 2012, Python developers distributed a security patch that introduces randomization into the computation of hash codes for strings, making it less tractable to reverse engineer a set of colliding strings.

10.2.4 Python Hash Table Implementation

In this section, we develop two implementations of a hash table, one using separate chaining and the other using open addressing with linear probing. While these approaches to collision resolution are quite different, there are a great many commonalities to the hashing algorithms. For that reason, we extend the MapBase class (from Code Fragment 10.2), to define a new HashMapBase class (see Code Fragment 10.4), providing much of the common functionality to our two hash table implementations. The main design elements of the HashMapBase class are:

- The bucket array is represented as a Python list, named self. table, with all entries initialized to None.
- We maintain an instance variable self. n that represents the number of distinct items that are currently stored in the hash table.
- If the load factor of the table increases beyond 0.5, we double the size of the table and rehash all items into the new table.
- We define a hash function utility method that relies on Python's built-in hash function to produce hash codes for keys, and a randomized Multiply- Add-and-Divide (MAD) formula for the compression function.

What is not implemented in the base class is any notion of how a "bucket" should be represented. With separate chaining, each bucket will be an independent structure. With open addressing, however, there is no tangible container for each bucket; the "buckets" are effectively interleaved due to the probing sequences.

In our design, the HashMapBase class presumes the following to be abstract methods, which must be implemented by each concrete subclass:

- _bucket_getitem(j, k)
  This method should search bucket j for an item having key k, returning the associated value, if found, or else raising a KeyError.
- _bucket_setitem(j, k, v)
  This method should modify bucket j so that key k becomes associated with value v. If the key already exists, the new value overwrites the existing value. Otherwise, a new item is inserted and this method is responsible for *incrementing* self. n.
- _bucket_delitem(j, k)
  This method should remove the item from bucket j having key k, or raise a KeyError if no such item exists. (self. n is decremented after this method.)
- __iter__
  This is the standard map method to iterate through all keys of the map. Our base class does not delegate this on a per-bucket basis because "buckets" in open addressing are not inherently disjoint.

Separate Chaining

Code Fragment 10.5 provides a concrete implementation of a hash table with separate chaining, in the form of the ChainHashMap class. To represent a single bucket, it relies on an instance of the UnsortedTableMap class from Code Fragment 10.3.

The first three methods in the class use index j to access the potential bucket in the bucket array, and a check for the special case in which that table entry is None. The only time we need a new bucket structure is when bucket setitem is called on an otherwise empty slot. The remaining functionality relies on map behaviors that are already supported by the individual UnsortedTableMap instances. We need a bit of forethought to determine whether the application of setitem on the chain causes a net increase in the size of the map (that is, whether the given key is new).

Linear Probing

Our implementation of a ProbeHashMap class, using open addressing with linear probing, is given in Code Fragments 10.6 and 10.7. In order to support deletions, we use a technique described in Section 10.2.2 in which we place a special marker in a table location at which an item has been deleted, so that we can distinguish between it and a location that has always been empty. In our implementation, we declare a class-level attribute, AVAIL, as a sentinel. (We use an instance of the built-in object class because we do not care about any behaviors of the sentinel, just our ability to differentiate it from other objects.)

The most challenging aspect of open addressing is to properly trace the series of probes when collisions occur during an insertion or search for an item. To this end, we define a nonpublic utility, find slot, that searches for an item with key k in "bucket" j (that is, where j is the index returned by the hash function for key k).

The three primary map operations each rely on the find slot utility. When at- tempting to retrieve the value associated with a given key, we must continue probing until we find the key, or until we reach a table slot with the None value. We cannot stop the search upon reaching an AVAIL sentinel, because it represents a location that may have been filled when the desired item was once inserted.

When a key-value pair is being assigned in the map, we must attempt to find an existing item with the given key, so that we might overwrite its value, before adding a new item to the map. Therefore, we must search beyond any occurrences of the AVAIL sentinel when inserting. However, if no match is found, we prefer to repurpose the first slot marked with AVAIL, if any, when placing the new element in the table. The find slot method enacts this logic, continuing the search until a truly empty slot,

but returning the index of the first available slot for an insertion.

When deleting an existing item within bucket delitem, we intentionally set the table entry to the AVAIL sentinel in accordance with our strategy.

10.3 Sorted Maps

The traditional map ADT allows a user to look up the value associated with a given key, but the search for that key is a form known as an exact search.

For example, computer systems often maintain information about events that have occurred (such as financial transactions), organizing such events based upon what are known as time stamps. If we can assume that time stamps are unique for a particular system, then we might organize a map with a time stamp serving as the key, and a record about the event that occurred at that time as the value. A particular time stamp could serve as a reference ID for an event, in which case we can quickly retrieve information about that event from the map. However, the map ADT does not provide any way to get a list of all events ordered by the time at which they occur, or to search for which event occurred closest to a particular time. In fact, the fast performance of hash-based implementations of the map ADT relies on the intentionally scattering of keys that may seem very "near" to each other in the original domain, so that they are more uniformly distributed in a hash table.

In this section, we introduce an extension known as the sorted map ADT that includes all behaviors of the standard map, plus the following:

M.find min(): Return the (key,value) pair with minimum key (or None, if map is empty).
M.find max(): Return the (key,value) pair with maximum key (or None, if map is empty).
M.find lt(k): Return the (key,value) pair with the greatest key that is strictly less than k (or None, if no such item exists).
M.find le(k): Return the (key,value) pair with the greatest key that is less than or equal to k (or None, if no such item exists).
M.find gt(k): Return the (key,value) pair with the least key that is strictly greater than k (or None, if no such item exists).
M.find ge(k): Return the (key,value) pair with the least key that is greater than or equal to k (or None, if no such item).
M.find range(start, stop): Iterate all (key,value) pairs with start <= key < stop.
If start is None, iteration begins with minimum key; if stop is None, iteration concludes with maximum key.
iter(M): Iterate all keys of the map according to their natural order, from smallest to largest.
reversed(M): Iterate all keys of the map in reverse order; in Python, this is implemented with the reversed method.

10.3.1 Sorted Search Tables

Several data structures can efficiently support the sorted map ADT, and we will examine some advanced techniques in Section 10.4 and Chapter 11. In this section, we begin by exploring a simple implementation of a sorted map. We store the map's items in an array-based sequence A so that they are in increasing order of their keys, assuming the keys have a naturally defined order. (See Figure 10.8.) We refer to this implementation of a map as a ***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

Figure 10.8: Realization of a map by means of a sorted search table. We show only the keys for this map, so as to highlight their ordering.

As was the case with the unsorted table map of Section 10.1.5, the sorted search table has a space requirement that is O(n), assuming we grow and shrink the array to keep its size proportional to the number of items in the map. The primary advantage of this representation, and our reason for insisting that A be array-based, is that it allows us to use the binary search algorithm for a variety of efficient operations.

Binary Search and Inexact Searches

We originally presented the binary search algorithm in Section 4.1.3, as a means for detecting whether a given target is stored within a sorted sequence. In our original presentation (Code Fragment 4.3 on page 156), a binary search function returned True of False to designate whether the desired target was found. While such an approach could be used to implement the contains method of the map ADT, we can adapt the binary search algorithm to provide far more useful information when performing forms of inexact search in support of the sorted map ADT.
The important realization is that while performing a binary search, we can determine the index at or near where a target might be found. During a successful search, the standard implementation determines the precise index at which the tar- get is found. During an unsuccessful search, although the target is not found, the algorithm will effectively determine a pair of indices designating elements of the collection that are just less than or just greater than the missing target.
As a motivating example, our original simulation from Figure 4.5 on page 156 shows a successful binary search for a target of 22, using the same data we portray in Figure 10.8. Had we instead been searching for 21, the first four steps of the algorithm would be the same. The subsequent difference is that we would make an additional call with inverted parameters high=9 and low=10, effectively concluding that the missing target lies in the gap between values 19 and 22 in that example.

Implementation

In Code Fragments 10.8 through 10.10, we present a complete implementation of a class, SortedTableMap, that supports the sorted map ADT. The most notable fea- ture of our design is the inclusion of a find index utility function. This method using the binary search algorithm, but by convention returns the index of the left- most item in the search interval having key greater than or equal to k. Therefore, if the key is present, it will return the index of the item having that key. (Recall that keys are unique in a map.) When the key is missing, the function returns the index of the item in the search interval that is just beyond where the key would have been located. As a technicality, the method returns index high + 1 to indicate that no items of the interval

had a key greater than k.

We rely on this utility method when implementing the traditional map operations and the new sorted map operations. The body of each of the getitem , setitem , and delitem methods begins with a call to find index to deter- mine a candidate index at which a matching key might be found. For getitem , we simply check whether that is a valid index containing the target to determine the result. For setitem , recall that the goal is to replace the value of an existing item, if one with key k is found, but otherwise to insert a new item into the map. The index returned by find index will be the index of the match, if one exists, or otherwise the exact index at which the new item should be inserted. For delitem , we again rely on the convenience of find index to determine the location of the
item to be popped, if any.

Our find index utility is equally valuable when implementing the various in- exact search methods given in Code Fragment 10.10. For each of the methods find lt, find le, find gt, and find ge, we begin with a call to find index utility, which locates the first index at which there is an element with key $\geqslant$ k, if any. This
is precisely what we want for find ge, if valid, and just beyond the index we want for find lt. For find gt and find le we need some extra case analysis to distinguish whether the indicated index has a key equal to k. For example, if the indicated item has a matching key, our find gt implementation increments the index before continuing with the process. (We omit the implementation of find le, for brevity.) In all cases, we must properly handle boundary cases, reporting None when unable to find a key with the desired property.

Our strategy for implementing find range is to use the find index utility to locate the first item with key $\geqslant$ start (assuming start is not None). With that knowledge, we use a while loop to sequentially report items until reaching one that has
a key greater than or equal to the stopping value (or until reaching the end of the table). It is worth noting that the while loop may trivially iterate zero items if the first key that is greater than or equal to start also happens to be greater than or equal to stop. This represents an empty range in the map.

Analysis

We conclude by analyzing the performance of our SortedTableMap implementation. A summary of the running times for all methods of the sorted map ADT (including the traditional map operations) is given in Table 10.3. It should be clear that the len , find min, and find max methods run in O(1) time, and that iterating the keys of the table in either direction can be performed in *O*(n) time.

The analysis for the various forms of search all depend on the fact that a binary search on a table with n entries runs in O(log n) time. This claim was originally shown as Proposition 4.2 in Section 4.2, and that analysis clearly applies to our find index method as well. We therefore claim an O(log n) worst-case running time for methods getitem , find lt, find gt, find le, and find ge. Each of these makes a single call to find index, followed by a constant number of additional steps to determine the appropriate answer based on the index. The analysis of find range is a bit more interesting. It begins with a binary search to find the first item within the range (if any). After that, it executes a loop that takes O(1) time per iteration to report subsequent values until reaching the end of the range. If there are s items reported in the range, the total running time is O(s + log n).

In contrast to the efficient search operations, update operations for a sorted table may take considerable time. Although binary search can help identify the index at which an update occurs, both insertions and deletions require, in the worst case, that linearly many existing elements be shifted in order to maintain the sorted order of the table. Specifically, the potential call to table.insert from within __setitem__ and table.pop from within __delitem__ lead to O(n) worst-case time. (See the discussion of corresponding operations of the list class in Section 5.4.1.)

In conclusion, sorted tables are primarily used in situations where we expect many searches but relatively few updates.

Operation	Running Time
len(M)	$O(1)$
k in M	$O(\log n)$
M[k] = v	$O(n)$ worst case; $O(\log n)$ if existing k
del M[k]	$O(n)$ worst case
M.find_min( ), M.find_max( )	$O(1)$
M.find_lt(k), M.find_gt(k) M.find_le(k), M.find_ge(k)	$O(\log n)$
M.find_range(start, stop)	$O(s+\log n)$ where s items are reported
iter(M), reversed(M)	$O(n)$

Table 10.3: Performance of a sorted map, as implemented with SortedTableMap. We use n to denote the number of items in the map at the time the operation is performed. The space requirement is $O(n)$.

10.3.2 Two Applications of Sorted Maps

In this section, we explore applications in which there is particular advantage to using a sorted map rather than a traditional (unsorted) map. To apply a sorted map, keys must come from a domain that is totally ordered. Furthermore, to take advantage of the inexact or range searches afforded by a sorted map, there should be some reason why nearby keys have relevance to a search.

Flight Databases

There are several Web sites on the Internet that allow users to perform queries on flight databases to find flights between various cities, typically with the intent to buy a ticket. To make a query, a user specifies origin and destination cities, a departure date, and a departure time. To support such queries, we can model the flight database as a map, where keys are Flight objects that contain fields corresponding to these four parameters. That is, a key is a tuple

k = (origin, destination, date, time).

Additional information about a flight, such as the flight number, the number of seats still available in first (F) and coach (Y) class, the flight duration, and the fare, can be stored in the value object. Finding a requested flight is not simply a matter of finding an exact match for a requested query. Although a user typically wants to exactly match the origin and destination cities, he or she may have flexibility for the departure date, and certainly will have some flexibility for the departure time on a specific day. We can handle such a query by ordering our keys lexicographically. Then, an efficient implementation for a sorted map would be a good way to satisfy users' queries. For instance, given a user query key k, we could call find ge(k) to return the first flight between the desired cities, having a departure date and time matching the desired query or later. Better yet, with well-constructed keys, we could use find range(k1, k2) to find all flights within a given range of times. For example, if k1 = (ORD, PVD, 05May, 09:30), and k2 = (ORD, PVD, 05May, 20:00),
a respective call to find range(k1, k2) might result in the following sequence of key-value pairs:

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

Maxima Sets

Life is full of trade-offs. We often have to trade off a desired performance measure against a corresponding cost. Suppose, for the sake of an example, we are interested in maintaining a database rating automobiles by their maximum speeds and their cost. We would like to allow someone with a certain amount of money to query our database to find the fastest car they can possibly afford.

We can model such a trade-off problem as this by using a key-value pair to model the two parameters that we are trading off, which in this case would be the pair (cost, speed) for each car. Notice that some cars are strictly better than other cars using this measure. For example, a car with cost-speed pair (20000, 100) is strictly better than a car with cost-speed pair (30000, 90). At the same time, there are some cars that are not strictly dominated by another car. For example, a car with cost-speed pair (20000, 100) may be better or worse than a car with cost-speed pair (30000, 120), depending on how much money we have to spend. (See Figure 10.9.)

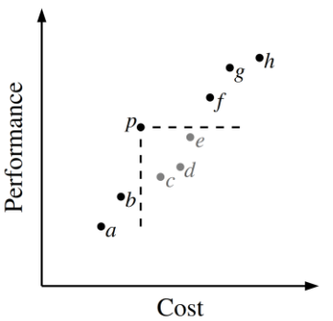


Figure 10.9: Illustrating the cost-performance trade-off with pairs represented by points in the plane. Notice that point *p* is strictly better than points *c*, *d*, and *e*, but may be better or worse than points *a*, *b*, *f*, *g*, and *h*, depending on the price we are willing to pay. Thus, if we were to add *p* to our set, we could remove the points *c*, *d*, and *e*, but not the others.

Formally, we say a cost-performance pair (a, b) dominates pair (c, d) = (a, b) if a $\leqslant$ c and b $\geqslant$ d, that is, if the first pair has no greater cost and at least as good performance. A pair (a, b) is called a maximum pair if it is not dominated by any
other pair. We are interested in maintaining the set of maxima of a collection of cost-performance pairs. That is, we would like to add new pairs to this collection (for example, when a new car is introduced), and to query this collection for a given dollar amount, d, to find the fastest car that costs no more than d dollars.

Maintaining a Maxima Set with a Sorted Map

We can store the set of maxima pairs in a sorted map, M, so that the cost is the
key field and performance (speed) is the value field. We can then implement operations add(c, p), which adds a new cost-performance pair (c, p), and best(c), which
returns the best pair with cost at most c, as shown in Code Fragment 10.11.

Unfortunately, if we implement M using the SortedTableMap,

the add behavior has O(n) worst-case running time. If, on the other hand, we implement M using a skip list, which we next describe, we can perform best(c) queries in O(log n) expected time and add(c, p) updates in O((1 + r) log n) expected time, where r is the number of points removed.

10.4 Skip Lists

An interesting data structure for realizing the sorted map ADT is the skip list. In Section 10.3.1, we saw that a sorted array will allow O(log n)-time searches via the binary search algorithm. Unfortunately, update operations on a sorted array have O(n) worst-case running time because of the need to shift elements. In Chapter 7 we demonstrated that linked lists support very efficient update operations, as long as the position within the list is identified. Unfortunately, we cannot perform fast searches on a standard linked list; for example, the binary search algorithm requires an efficient means for direct accessing an element of a sequence by index.

Skip lists provide a clever compromise to efficiently support search and update operations. A skip list S for a map M consists of a series of lists {S0, S1,..., Sh}. Each list Si stores a subset of the items of M sorted by increasing keys, plus items with two sentinel keys denoted $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in M and $+\infty$ is larger than every possible key

that can be inserted in M. In addition, the lists in S satisfy the following:

- List S0 contains every item of the map M (plus sentinels $-\infty$ and $+\infty$).
- For i = 1,..., h − 1, list Si contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the items in list Si−1.
- List Sh contains only $-\infty$ and $+\infty$.

An example of a skip list is shown in Figure 10.10. It is customary to visualize a skip list S with list S0 at the bottom and lists S1,..., Sh above it. Also, we refer to h as the height of skip list S.

Intuitively, the lists are set up so that Si+1 contains more or less alternate items of Si. As we shall see in the details of the insertion method, the items in Si+1 are chosen at random from the items in Si by picking each item from Si to also be in Si+1 with probability 1/2. That is, in essence, we "flip a coin" for each item in S_i

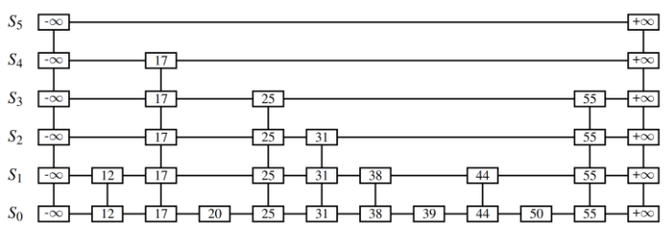


Figure 10.10: Example of a skip list storing 10 items. For simplicity, we show only the items' keys, not their associated values.

and place that item in Si+1 if the coin comes up "heads." Thus, we expect S1 to have about n/2 items, S2 to have about n/4 items, and, in general, Si to have about n/2i items. In other words, we expect the height h of S to be about log n. The halving of the number of items from one list to the next is not enforced as an explicit property

of skip lists, however. Instead, randomization is used.

Functions that generate numbers that can be viewed as random numbers are built into most modern computers, because they are used extensively in computer games, cryptography, and computer simulations, Some functions, called pseudo- random number generators, generate random-like numbers, starting with an initial seed. (See discusion of random module in Section 1.11.1.) Other methods use hardware devices to extract "true" random numbers from nature. In any case, we will assume that our computer has access to numbers that are sufficiently random for our analysis.

The main advantage of using randomization in data structure and algorithm design is that the structures and functions that result are usually simple and efficient. The skip list has the same logarithmic time bounds for searching as is achieved by the binary search algorithm, yet it extends that performance to update methods when inserting or deleting items. Nevertheless, the bounds are expected for the skip list, while binary search has a worst-case bound with a sorted table.

A skip list makes random choices in arranging its structure in such a way that search and update times are O(log n) on average, where n is the number of items in the map. Interestingly, the notion of average time complexity used here does not depend on the probability distribution of the keys in the input. Instead, it depends on the use of a random-number generator in the implementation of the insertions to help decide where to place the new item. The running time is averaged over all possible outcomes of the random numbers used when inserting entries.

Using the position abstraction used for lists and trees, we view a skip list as a two-dimensional collection of positions arranged horizontally into levels and ver- tically into towers. Each level is a list Si and each tower contains positions storing the same item across consecutive lists. The positions in a skip list can be traversed using the following operations:

next(p): Return the position following p on the same level.

prev(p): Return the position preceding p on the same level. below(p): Return the position below p in the same tower. above(p): Return the position above p in the same tower.

We conventionally assume that the above operations return None if the position

requested does not exist. Without going into the details, we note that we can easily implement a skip list by means of a linked structure such that the individual traversal methods each take O(1) time, given a skip-list position p. Such a linked structure is essentially a collection of h doubly linked lists aligned at towers, which are also doubly linked lists.

10.4.1 Search and Update Operations in a Skip List

The skip-list structure affords simple map search and update algorithms. In fact, all of the skip-list search and update algorithms are based on an elegant SkipSearch

method that takes a key k and finds the position p of the item in list S0 that has the largest key less than or equal to k (which is

possibly $-\infty$).

Searching in a Skip List

Suppose we are given a search key k. We begin the SkipSearch method by setting a position variable p to the topmost, left position in the skip list S, called the start

position of S. That is, the start position is the position of Sh storing the special entry with key $-\infty$. We then perform the following steps (see Figure 10.11), where key(p) denotes the key of the item at position p:

1. If S.below(p) is None, then the search terminates—we are at the bottom and have located the item in S with the largest key less than or equal to the search key k. Otherwise, we drop down to the next lower level in the present tower by setting p = S.below(p).
2. Starting at position p, we move p forward until it is at the rightmost position on the present level such that key(p) $\leqslant$ k. We call this the scan forward step. Note that such a position always exists, since each level contains the keys

+ __and__. It may be that p remains where it started after we perform such a forward scan for this level.

3. Return to step 1.

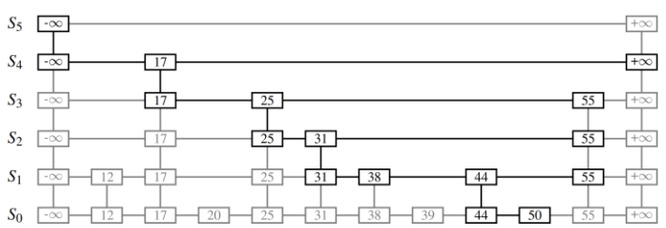


Figure 10.11: Example of a search in a skip list. The positions examined when searching for key 50 are highlighted.

We give a pseudo-code description of the skip-list search algorithm, SkipSearch, in Code Fragment 10.12. Given this method, the map operation M[k] is performed by computing p = SkipSearch(k) and testing whether or not key(p)= k. If these two keys are equal, we return the associated value; otherwise, we raise a KeyError.

As it turns out, the expected running time of algorithm SkipSearch on a skip list with n entries is O(log n). We postpone the justification of this fact, however, until after we discuss the implementation of the update methods for skip lists. Navigation starting at the position identified by SkipSearch(k) can be easily used to provide the additional forms of searches in the sorted map ADT (e.g., find gt, find range).

Insertion in a Skip List

The execution of the map operation M[k] = v begins with a call to SkipSearch(k). This gives us the position p of the bottom-level item with the largest key less than or equal to k (note that p may hold the special item with key $-\infty$). If key(p)= k, the associated value is overwritten with v. Otherwise, we need to create a new tower for item (k, v). We insert (k, v) immediately after position p within S0. After inserting the new item at the bottom level, we use randomization to decide the height of the tower for the new item. We "flip" a coin, and if the flip comes up tails, then we stop here. Else (the flip comes up heads), we backtrack to the previous (next higher) level and insert (k, v) in this level at the appropriate position. We again flip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new item (k, v) in lists until we finally get a flip that comes up tails. We link together all the references to the new item (k, v) created in this process to create its tower. A coin flip can be simulated with Python's built-in pseudo-random number generator from the random module by calling randrange(2), which returns 0 or 1, each with probability 1/2.

We give the insertion algorithm for a skip list S in Code Fragment 10.13 and we illustrate it in Figure 10.12. The algorithm uses an insertAfterAbove(p, q,(k, v)) method that inserts a position storing the item (k, v) after position p (on the same level as p) and above position q, returning the new position r (and setting internal references so that next, prev, above, and below methods will work correctly for p, q, and r). The expected running time of the insertion algorithm on a skip list with n entries is O(log n), which we show in Section 10.4.2.

```
Algorithm SkipInsert(k,v):
    Input: Key k and value v
    Output: Topmost position of the item inserted in the skip list
    p = SkipSearch(k)
    q = None                          {q will represent top node in new item's tower}
    i = −1
    repeat
      i = i + 1
      if i ≥ h then
        h = h + 1                                   {add a new level to the skip list}
        t = next(s)
        s = insertAfterAbove(None, s, (−∞, None))   {grow leftmost tower}
        insertAfterAbove(s, t, (+∞, None))          {grow rightmost tower}
      while above(p) is None do
        p = prev(p)                                 {scan backward}
      p = above(p)                                  {jump up to higher level}
      q = insertAfterAbove(p, q, (k, v))            {increase height of new item's tower}
    until coinFlip() == tails
    n = n + 1
    return q
```

Code Fragment 10.13: Insertion in a skip list. Method coinFlip() returns "heads" or "tails", each with probability 1/2. Instance variables n, h, and s hold the number of entries, the height, and the start node of the skip list.

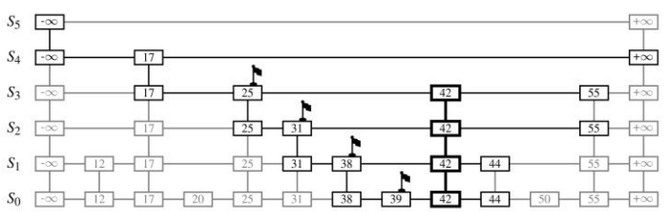


Figure 10.12: Insertion of an entry with key 42 into the skip list of Figure 10.10. We assume that the random "coin flips" for the new entry came up heads three times in a row, followed by tails. The positions visited are highlighted. The positions inserted to hold the new entry are drawn with thick lines, and the positions preceding them are flagged.

Removal in a Skip List

Like the search and insertion algorithms, the removal algorithm for a skip list is quite simple. In fact, it is even easier than the insertion algorithm. That is, to per- form the map operation del M[k] we begin by executing method SkipSearch(k). If the position p stores an entry with key different from k, we raise a KeyError. Otherwise, we remove p and all the positions above p, which are easily accessed by using above operations to climb up the tower of this entry in S starting at position p. While removing levels of the tower, we reestablish links between the horizontal neighbors of each removed position. The removal algorithm is illustrated in Figure 10.13 and a detailed description of it is left as an exercise (R-10.24). As we show in the next subsection, deletion operation in a skip list with n entries has O(log n) expected running time.

Before we give this analysis, however, there are some minor improvements to the skip-list data structure we would like to discuss. First, we do not actually need to store references to values at the levels of the skip list above the bottom level, because all that is needed at these levels are references to keys. In fact, we can more efficiently represent a tower as a single object, storing the key-value pair, and maintaining j previous references and j next references if the tower reaches level Sj. Second, for the horizontal axes, it is possible to keep the list singly linked, storing only the next references. We can perform insertions and removals in strictly a top-down, scan-forward fashion. We explore the details of this optimization in Exercise C-10.44. Neither of these optimizations improve the asymptotic performance of skip lists by more than a constant factor, but these improvements can, nevertheless, be meaningful in practice. In fact, experimental evidence suggests that optimized skip lists are faster in practice than AVL trees and other balanced search trees, which are discussed in Chapter 11.

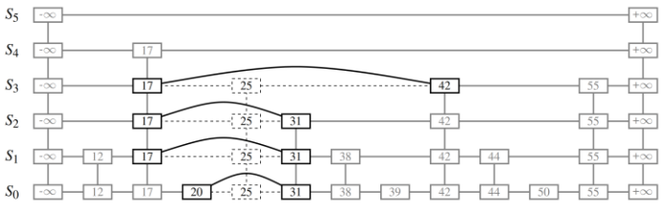


Figure 10.13: Removal of the entry with key 25 from the skip list of Figure 10.12. The positions visited after the search for the position of S_0 holding the entry are highlighted. The positions removed are drawn with dashed lines.

Maintaining the Topmost Level

A skip list S must maintain a reference to the start position (the topmost, left position in S) as an instance variable, and must have a policy for any insertion that wishes to continue inserting a new entry past the top level of S. There are two possible courses of action we can take, both of which have their merits.
One possibility is to restrict the top level, h, to be kept at some fixed value that is a function of n, the number of entries currently in the map (from the analysis we will see that h = max{10, 2⌈log n1} is a reasonable choice, and picking h = 3⌈log n1 is even safer). Implementing this choice means that we must modify the insertion algorithm to stop inserting a new position once we reach the topmost level (unless ⌈log n1 < ⌈log(n + 1)1, in which case we can now go at least one more level, since the bound on the height is increasing).
The other possibility is to let an insertion continue inserting a new position as long as heads keeps getting returned from the random number generator. This is the approach taken by algorithm SkipInsert of Code Fragment 10.13. As we show in the analysis of skip lists, the probability that an insertion will go to a level that is more than O(log n) is very low, so this design choice should also work.
Either choice will still result in the expected O(log n) time to perform search, insertion, and removal, however, which we show in the next section.

10.4.2 Probabilistic Analysis of Skip Lists *

As we have shown above, skip lists provide a simple implementation of a sorted map. In terms of worst-case performance, however, skip lists are not a superior data structure. In fact, if we do not officially prevent an insertion from continuing significantly past the current highest level, then the insertion algorithm can go into what is almost an infinite loop (it is not actually an infinite loop, however, since the prob- ability of having a fair coin repeatedly come up heads forever is 0). Moreover, we cannot infinitely add positions to a list without eventually running out of memory. In any case, if we terminate position insertion at the highest level h, then the worst- case running time for performing the getitem , setitem , and delitem
map operations in a skip list S with n entries and height h is *O*(n + h). This worst- case performance occurs when the tower of every entry reaches level h − 1, where h is the height of S. However, this event has very low probability. Judging from
this worst case, we might conclude that the skip-list structure is strictly inferior to the other map implementations discussed earlier in this chapter. But this would not be a fair analysis, for this worst-case behavior is a gross overestimate.

Bounding the Height of a Skip List

Because the insertion step involves randomization, a more accurate analysis of skip lists involves a bit of probability. At first, this might seem like a major undertaking, for a complete and thorough probabilistic analysis could require deep mathematics (and, indeed, there are several such deep analyses that have appeared in data structures research literature). Fortunately, such an analysis is not necessary to un- derstand the expected asymptotic behavior of skip lists. The informal and intuitive probabilistic analysis we give below uses only basic concepts of probability theory.
Let us begin by determining the expected value of the height h of a skip list S with n entries (assuming that we do not terminate insertions early). The probability that a given entry has a tower of height i 1 is equal to the probability of getting i consecutive heads when flipping a coin, that is, this probability is 1/2i. Hence, the probability Pi that level i has at least one position is at most

n Pi ≤ 2i ,
for the probability that any one of n different events occurs is at most the sum of the probabilities that each occurs.
The probability that the height h of S is larger than i is equal to the probability that level i has at least one position, that is, it is no more than Pi. This means that h is larger than, say, 3 log n with probability at most

P3log n ≤
=

n

23log n
n
n3 =

1
n2 .

For example, if n = 1000, this probability is a one-in-a-million long shot. More generally, given a constant c > 1, h is larger than c log n with probability at most 1/nc−1. That is, the probability that h is smaller than c log n is at least 1 − 1/nc−1. Thus, with high probability, the height h of S is O(log n).

Analyzing Search Time in a Skip List

Next, consider the running time of a search in skip list S, and recall that such a search involves two nested while loops. The inner loop performs a scan forward on a level of S as long as the next key is no greater than the search key k, and the outer loop drops down to the next level and repeats the scan forward iteration. Since the height h of S is O(log n) with high probability, the number of drop-down steps is O(log n) with high probability.

So we have yet to bound the number of scan-forward steps we make. Let ni be the number of keys examined while scanning forward at level i. Observe that, after the key at the starting position, each additional key examined in a scan-forward at level i cannot also belong to level i + 1. If any of these keys were on the previous level, we would have encountered them in the previous scan-forward step. Thus, the probability that any key is counted in ni is 1/2. Therefore, the expected value of ni is exactly equal to the expected number of times we must flip a fair coin before it comes up heads. This expected value is 2. Hence, the expected amount of time spent scanning forward at any level i is O(1). Since S has O(log n) levels with high probability, a search in S takes expected time O(log n). By a similar analysis, we can show that the expected running time of an insertion or a removal is O(log n).

Space Usage in a Skip List
Finally, let us turn to the space requirement of a skip list S with n entries. As we observed above, the expected number of positions at level i is n/2i, which means that the expected total number of positions in S is
h n h 1
2i = n 2i .
i=0 i=0
Using Proposition 3.5 on geometric summations, we have

h 1 (1)h+1 − 1

= 2 ·

（1 −

1
< 2 for all h ≥ 0.

i=0 2i

2 − 1

2h+1

Hence, the expected space requirement of S is O(n).

Table 10.4 summarizes the performance of a sorted map realized by a skip list.

Operation	Running Time
len(M)	$O(1)$
k **in** M	$O(\log n)$ expected
M[k] = v	$O(\log n)$ expected
del M[k]	$O(\log n)$ expected
M.find_min( ), M.find_max( )	$O(1)$
M.find_lt(k), M.find_gt(k) M.find_le(k), M.find_ge(k)	$O(\log n)$ expected
M.find_range(start, stop)	$O(s+\log n)$ expected, with s items reported
iter(M), reversed(M)	$O(n)$

Table 10.4: Performance of a sorted map implemented with a skip list. We use n to denote the number of entries in the dictionary at the time the operation is performed. The expected space requirement is $O(n)$.

10.5 Sets, Multisets, and Multimaps

We conclude this chapter by examining several additional abstractions that are closely related to the map ADT, and that can be implemented using data structures similar to those for a map. A set is an unordered collection of elements, without duplicates, that typically supports efficient membership tests. In essence, elements of a set are like keys of a map, but without any auxiliary values.

- A multiset (also known as a bag) is a set-like container that allows duplicates.
- A multimap is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple val- ues. For example, the index of this book maps a given term to one or more locations at which the term occurs elsewhere in the book.

10.5.1 The Set ADT

Python provides support for representing the mathematical notion of a set through the built-in classes frozenset and set, as originally discussed in Chapter 1, with frozenset being an immutable form. Both of those classes are implemented using hash tables in Python.

Python's collections module defines abstract base classes that essentially mirror these built-in classes. Although the choice of names is counterintuitive, the abstract base class collections.Set matches the concrete frozenset class, while the abstract base class collections.MutableSet is akin to the concrete set class.

In our own discussion, we equate the "set ADT" with the behavior of the built- in set class (and thus, the collections.MutableSet base class). We begin by listing what we consider to be the five most fundamental behaviors for a set S:

S.add(e): Add element e to the set. This has no effect if the set already contains e.

S.discard(e): Remove element e from the set, if present. This has no effect if the set does not contain e.

e in S: Return True if the set contains element e. In Python, this is implemented with the special containsmethod.

len(S): Return the number of elements in set S. In Python, this is implemented with the special method len .

iter(S): Generate an iteration of all elements of the set. In Python, this is implemented with the special method iter

In the next section, we will see that the above five methods suffice for deriving all other behaviors of a set. Those remaining behaviors can be naturally grouped as follows. We begin by describing the following additional operations for removing one or more elements from a set:

S.remove(e): Remove element e from the set. If the set does not contain e, raise a KeyError.
S.pop( ): Remove and return an arbitrary element from the set. If the set is empty, raise a KeyError.
S.clear( ): Remove all elements from the set.

The next group of behaviors perform Boolean comparisons between two sets.

S == T: Return True if sets S and T have identical contents.
S != T: Return True if sets S and T are not equivalent.
S <= T: Return True if set S is a subset of set T.
S < T: Return True if set S is a proper subset of set T.
S >= T: Return True if set S is a superset of set T.
S > T: Return True if set S is a proper superset of set T.
S.isdisjoint(T): Return True if sets S and T have no common elements.

Finally, there exists a variety of behaviors that either update an existing set, or compute a new set instance, based on classical set theory operations.
S | T: Return a new set representing the union of sets S and T.
S |= T: Update set S to be the union of S and set T.
S & T: Return a new set representing the intersection of sets S and T.
S &= T: Update set S to be the intersection of S and set T.
S ^ T: Return a new set representing the symmetric difference of sets S and T, that is, a set of elements that are in precisely one of S or T.
S ^= T: Update set S to become the symmetric difference of itself and set T.
S − T: Return a new set containing elements in S but not T.
S −= T: Update set S to remove all common elements with set T.

10.5.2 Python's MutableSet Abstract Base Class

To aid in the creation of user-defined set classes, Python's collections module pro- vides a MutableSet abstract base class (just as it provides the MutableMapping abstract base class discussed in Section 10.1.3). The MutableSet base class provides concrete implementations for all methods described in Section 10.5.1, except for five core behaviors (add, discard, __contains__, __len__, and __iter___) that must be implemented by any concrete subclass. This design is an example of what is known as the template method pattern, as the concrete methods of the MutableSet class rely on the presumed abstract methods that will subsequently be provided by a subclass.

For the purpose of illustration, we examine algorithms for implementing several of the derived methods of the MutableSet base class. For example, to determine if one set is a proper subset of another, we must verify two conditions: a proper subset must have size strictly smaller than that of its superset, and each element of a subset must be contained in the superset. An implementation of the corresponding __lt__ method based on this logic is given in Code Fragment 10.14.

```
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
```

Code Fragment 10.14: A possible implementation of the MutableSet.__lt__ method, which tests if one set is a proper subset of another.

As another example, we consider the computation of the union of two sets. The set ADT includes two forms for computing a union. The syntax S | T should produce a new set that has contents equal to the union of existing sets S and T . This operation is implemented through the special method or in Python. Another syntax, S |= T is used to update existing set S to become the union of itself and set T . Therefore, all elements of T that are not already contained in S should be added to S. We note that this "in-place" operation may be implemented more efficiently than if we were to rely on the first form, using the syntax S = S | T, in which identifier S is reassigned to a new set instance that represents the union. For convenience, Python's built-in set class supports named version of these behaviors, with S.union(T) equivalent to S | T, and S.update(T) equivalent to S |= T (yet, those named versions are not formally provided by the MutableSet abstract base class).

```
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
```

Code Fragment 10.15: An implementation of the MutableSet.__or__ method, which computes the union of two existing sets.

An implementation of the behavior that computes a new set as a union of two others is given in the form of the or special method, in Code Fragment 10.15. An important subtlety in this implementation is the instantiation of the resulting set. Since the MutableSet class is designed as an abstract base class, instances must belong to a concrete subclass. When computing the union of two such concrete instances, the result should presumably be an instance of the same class as the operands. The function type(self) returns a reference to the actual class of the in- stance identified as self, and the subsequent parentheses in expression type(self)( ) call the default constructor for that class.

In terms of efficiency, we analyze such set operations while letting n denote the size of S and m denote the size of set T for an operation such as S | T. If the concrete sets are implemented with hashing, the expected running time of the implementation in Code Fragment 10.15 is O(m + n), because it loops over both sets, performing constant-time operations in the form of a containment check and a possible insertion into the result.

Our implementation of the in-place version of a union is given in Code Fragment 10.16, in the form of the ior special method that supports syntax S |= T. Notice that in this case, we do not create a new set instance, instead we modify and return the existing set, after updating its contents to reflect the union operation. The in-place version of the union has expected running time O(m) where m is the size of the second set, because we only have to loop through that second set.

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

Code Fragment 10.16: An implementation of the MutableSet.__ior__ method, which performs an in-place union of one set with another.

10.5.3 Implementing Sets, Multisets, and Multimaps

Sets

Although sets and maps have very different public interfaces, they are really quite similar. A set is simply a map in which keys do not have associated values. Any data structure used to implement a map can be modified to implement the set ADT with similar performance guarantees. We could trivially adapt any map class by storing set elements as keys, and using None as an irrelevant value, but such an implementation is unnecessarily wasteful. An efficient set implementation should abandon the Item composite that we use in our MapBase class and instead store set elements directly in a data structure.

Multisets

The same element may occur several times in a multiset. All of the data structures we have seen can be reimplemented to allow for duplicates to appear as separate elements. However, another way to implement a multiset is by using a map in which the map key is a (distinct) element of the multiset, and the associated value is a count of the number of occurrences of that element within the multiset. In fact, that is essentially what we did in Section 10.1.2 when computing the frequency of words within a document.

Python's standard collections module includes a definition for a class named Counter that is in essence a multiset. Formally, the Counter class is a subclass of dict, with the expectation that values are integers, and with additional functionality like a most common(n) method that returns a list of the n most common elements. The standard iter reports each element only once (since those are formally the keys of the dictionary). There is another method named elements( ) that iterates through the multiset with each element being repeated according to its count.

Multimaps

Although there is no multimap in Python's standard libraries, a common implementation approach is to use a standard map in which the value associated with a key is itself a container class storing any number of associated values. We give an example of such a MultiMap class in Code Fragment 10.17. Our implementation uses the standard dict class as the map, and a list of values as a composite value in the dictionary. We have designed the class so that a different map implementation can easily be substituted by overriding the class-level MapType attribute at line 3.

END

六、实验体会

七、参考文献

[1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Data Structures and Algorithms in Python
[2] 数据结构与算法分析：C 语言描述（原书第二版），（美）维斯著；冯舜玺译. 北京：机械工业出版社
[3] 算法导论（原书第三版），（美）科尔曼（Cormen，T.H.）等；殷建平等译. 北京：机械工业出版社