Hash Codes in Python
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 hashtable, yet a later search were pe
Among Python·s built-in data types, the immutable int,·oat ,str,tuple ,a n d
frozenset classes produce robust hash codes, via the hash function, using tech-
niques similar to those discussed earlier in this section. Hash codes for characterstrings are well crafted based o
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 ,t h e
order of the elements should be irrelevant, and so a natural option is to compute theexclusive-or of the individual
for an instance xof a mutable type, such as a list,aTypeError is raised.
Instances of user-de·ned 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 re·ect the immutable attributes of an in-stance. It is common to return a
three numeric red, green, and blue components might implement the method as:
def
hash
(self):
return hash((self.
red,self.
green, self.
blue)) # hash combined tuple
An important rule to obey is that if a class de·nes equivalence through
eq
,
then any implementation of
hash
must be consistent, in that if x= =y ,t h e n
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 secondinstance should result in the di
true, it ensures that hash(5) andhash(5.0) are the same.

Hash Codes

The ·rst action that a hash function performs is to take an arbitrary key kin our
map and compute an integer that is called the hash code fork; 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 ofhash codes. Following that, we di

Treating the Bit Representation as an Integer

To begin, we note that, for any data type Xthat is represented using at most as many
bits as our integer hash codes, we can simply take as a hash code for Xan integer
interpretation of its bits. For example, the hash code for key 314 could simply be
314. The hash code for a ·oating-point number such as 3 .14 could be based upon
an interpretation of the bits of the ·oating-point representation as an integer.
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 ·oating-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 informationpresent in the original key, an
A better approach is to combine in some way the high-order and low-order por-
tions of a 64-bit key to form a 32-bit hash code, which takes all the original bitsinto consideration. A simple imple
binary representation can be viewed as an n-tuple (x
0,x1,..., xn·1)of 32-bit inte-
gers, for example, by forming a hash code for xas·n·1
i=0xi,o ra s x0·x1····· xn·1,
where the ·symbol represents the bitwise exclusive-or operation (which is ·in
Python).

Polynomial Hash Codes

The summation and exclusive-or hash codes, described above, are not good choicesfor character strings or othe
0,x1,..., xn·1), where the order of the xi·s is signi·cant. For example,
consider a 16-bit hash code for a character string sthat sums the Unicode values
of the characters in s. This hash code unfortunately produces lots of unwanted

10.2.1 Hash Functions

The goal of a hash function ,h, is to map each key kto an integer in the range [0,N·1],w h e r e Nis 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)].

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 willdiscuss later, but the best strategy is
be fast and easy to compute.

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 kto 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.)

-1hash code

120 -2... ...

compression function

120N - 1 ...Arbitrary Objects

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 speci·c hash table size. This allows the development of a general hash code for each object that canbe used for a hash table of any
currently stored in the map. (See Section 10.2.3.)

Compression Functions

The hash code for a key kwill typically not be suitable for immediate use with a bucket array, because the integer hash code may be negative or may exceed the ca-pacity 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.

The Division Method

A simple compression function is the division method , which maps an integer ito

imod N,

where N, the size of the bucket array, is a ·xed positive integer. Additionally, if we take Nto be a prime number, then this compression function helps ·spread out· the distribution of hashed values. Indeed, if Nis not prime, then there is greater risk that patterns in the distribution of hash codes will be repeated in the distribution ofhash values, thereby causing c
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 ensurethat the probability of two differe
Choosing Nto be a prime number is not always enough, however, for if there is a repeated pattern of hash codes of the form pN +qfor several different p·s, then there will still be collisions.

The MAD Method

A more sophisticated compression function, which helps eliminate repeated pat-terns in a set of integer keys, is t
This method maps an integer ito

[(ai+b)mod p]mod N,

where Nis the size of the bucket array, pis a prime number larger than N,a n d a andbare 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 ofhash codes and get us closer
the same as we would have if these keys were ·thrown· into Auniformly at random.

collisions for common groups of strings. In particular, "temp01" and"temp10"
collide using this function, as do "stop", "tops", "pots",a n d "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/negationslash=1, and use as a hash code the value
x0an·1+x1an·2+···+xn·2a+xn·1.
Mathematically speaking, this is simply a polynomial in athat takes the compo-
nents (x0,x1,..., xn·1)of an object xas its coef·cients. This hash code is therefore
called a polynomial hash code . By Horner·s rule (see Exercise C-3.50), this poly-
nomial can be computed as
xn·1+a(xn·2+a(xn·3+···+a(x2+a(x1+ax0))···)).
Intuitively, a polynomial hash code uses multiplication by different powers as a
way to spread out the in·uence of each component across the resulting hash code.
Of course, on a typical computer, evaluating a polynomial will be done using
the ·nite bit representation for a hash code; hence, the value will periodically over-·ow the bits used for an intege
should be mindful that such over·ows are occurring and choose the constant aso
that it has some nonzero, low-order bits, which will serve to preserve some of theinformation content even as we
We have done some experimental studies that suggest that 33, 37, 39, and 41
are particularly good choices for awhen working with character strings that are
English words. In fact, in a list of over 50,000 English words formed as the unionof the word lists provided in two
37, 39, or 41 produced less than 7 collisions in each case!

Cyclic-Shift Hash Codes

A variant of the polynomial hash code replaces multiplication by awith a cyclic
shift of a partial sum by a certain number of bits. For example, a 5-bit cyclic shiftof the 32-bit value 00111
101100101101010100010101000 is achieved by taking
the leftmost ·ve bits and placing those on the rightmost side of the representation,resulting in 101100101101010
. While this operation has little
natural meaning in terms of arithmetic, it accomplishes the goal of varying the bitsof the calculation. In Python, a
32-bit integers.

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 ndenote
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 ofitems in the map.
In practice, hash tables are among the most ef·cient means for implementing
a map, and it is essentially taken for granted by programmers that their core oper-
ations run in constant time. Python·s dictclass is implemented with hashing, and
the Python interpreter relies on dictionaries to retrieve an object that is referenced
by an identi·er in a given namespace. (See Sections 1.10 and 2.5.) The basic com-mand c=a+b involves two cal
getitem
in the dictionary for the local
namespace to retrieve the values identi·ed as aandb, and a call to
setitem
to store the result associated with name cin that namespace. In our own algorithm
analysis, we simply presume that such dictionary operations run in constant time,independent of the number of e
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) attackof Internet technologies. For ma
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 thehashing schemes we describe, ot
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 beembedded in a URL using a sy
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 wereto collide, that storage requires qua

10.2.3 Load Factors, Rehashing, and E·ciency
In the hash table schemes described thus far, it is important that the load factor,
·=n/N, be kept below 1. With separate chaining, as ·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 col-
lisions. Experiments and average-case analyses suggest that we should maintain·<0.9 for hash tables with sepa
With open addressing, on the other hand, as the load factor ·grows beyond 0 .5
and starts approaching 1, clusters of entries in the bucket array start to grow as well.These clusters cause the pr
gest that we should maintain ·<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 implementatio
If an insertion causes the load factor of a hash table to go above the speci·ed
threshold, then it is common to resize the table (to regain the speci·ed load factor)
and to reinsert all objects into this new table. Although we need not de·ne a new
hash code for each object, we do need to reapply a new compression function thattakes into consideration the si
scatter the items throughout the new bucket array. When rehashing to a new table, itis a good requirement for th
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 timeused to insert them in the ·rst pl
E·ciency of Hash Tables
Although the details of the average-case analysis of hashing are beyond the scopeof this book, its probabilistic b
array. Thus, to store nentries, the expected number of keys in a bucket would
be·n/N·,w h i c hi s O(1)ifnisO(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 sepa-
rate chaining, or with any open addressing model in which the secondary sequenceof probes depends only on th
Table 10.2.

10.6 Exercises
For help with exercises, please visit the site, www.wiley.com/college/goodrich.
Reinforcement
R-10.1 Give a concrete implementation of the popmethod in the context of the MutableMapping class, relying only on the ·ve primary abstract methods of that class.
R-10.2 Give a concrete implementation of the items() method in the context of theMutableMapping class, relying only on the ·ve primary abstract meth-ods of that class. What would its running time be if directly applied to the UnsortedTableMap subclass?
R-10.3 Give a concrete implementation of the items() method directly within the UnsortedTableMap class, ensuring that the entire iteration runs in O(n) time.
R-10.4 What is the worst-case running time for inserting nkey-value pairs into an initially empty map Mthat is implemented with the UnsortedTableMap class?
R-10.5 Reimplement the UnsortedTableMap class from Section 10.1.5, using the PositionalList class from Section 7.4 rather than a Python list.
R-10.6 Which of the hash table collision-handling schemes could tolerate a load factor above 1 and which could not?
R-10.7 OurPosition classes for lists and trees support the
eq
method so that
two distinct position instances are considered equivalent if they refer to thesame underlying node in a structure. I
hash
method that
is consistent with this notion of equivalence. Provide such a
hash
method.
R-10.8 What would be a good hash code for a vehicle identi·cation number thatis a string of numbers and letters
R-10.9 Draw the 11-entry hash table that results from using the hash function,h(i)=(3i+5)mod 11, to hash the ke
16, and 5, assuming collisions are handled by chaining.
R-10.10 What is the result of the previous exercise, assuming collisions are han-dled by linear probing?
R-10.11 Show the result of Exercise R-10.9, assuming collisions are handled by quadratic probing, up to the point where the method fails.

10.2 Hash Tables
In this section, we introduce one of the most practical data structures for imple-
menting a map, and the one that is used by Python·s own implementation of the
dictclass. This structure is known as a hash table .
Intuitively, a map Msupports 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 nitems uses keys that are known to be integers in a range from 0 to
N·1f o rs o m e N·n. In this case, we can represent the map using a lookup table
of length N, as diagrammed in Figure 10.3.
0 123456789 1 0
DZ C Q
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 kat index kof the
table (presuming that we have a distinct way to represent an empty slot). Basic mapoperations of
getitem
,
setitem
,a n d
delitem
can be implemented in
O(1)worst-case time.
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 Nif it is the case
thatN/greatermuchn. 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 therange from 0 to N·1 by a hash
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 speci·c index by the hash function.
(To save space, an empty bucket may be replaced by None .)
0 123456789 1 0
(1,D) (25,C)
(3,F)
(14,Z)(39,C)(6,A) (7,Q)
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.

10.2.2 Collision-Handling Schemes

The main idea of a hash table is to take a bucket array, A, and a hash function, h,a n d 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, k1andk2, such that h(k1)=h(k2). 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.

Separate Chaining

A simple and ef·cient 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.

A123456789 1 0 01 112
123825
9054
28413618 10

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)=kmod 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 nitems 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(·n/N·). The ratio ·=n/N, called the load factor of the hash table, should be bounded by a small constant, preferably below 1. As long as ·isO(1), the core operations on the hash table run in O(1)expected time.

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 wouldfail because that search would sta
this special marker possibly occupying spaces in our hash table, we modify our
search algorithm so that the search for a key kwill skip over cells containing the
available marker and continue probing until reaching the desired item or an emptybucket (or returning back to wh
for
setitem
should remember an available cell encountered during the search
fork, since this is a valid place to put a new item (k,v), if no existing item is found.
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 intocontiguous runs, which may ev
in the hash table are occupied). Such contiguous runs of occupied hash cells causesearches to slow down consi
Another open addressing strategy, known as quadratic probing , iteratively tries
the buckets A[(h(k)+f(i))mod N],f o ri=0,1,2,...,w h e r e f(i)=i
2, until ·nding
an empty bucket. As with linear probing, the quadratic probing strategy compli-cates the removal operation, but i
pattern, even if we assume that the original hash codes are distributed uniformly.When Nis prime and the bucket
strategy is guaranteed to ·nd an empty slot. However, this guarantee is not validonce the table becomes at least
we explore the cause of this type of clustering in an exercise (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
/prime,a n di f hmaps
some key kto a bucket A[h(k)]that is already occupied, then we iteratively try
the buckets A[(h(k)+f(i))mod N]next, for i=1,2,3,...,w h e r e f(i)=i·h/prime(k).
In this scheme, the secondary hash function is not allowed to evaluate to zero; acommon choice is h
/prime(k)=q·(kmod q), for some prime number q<N.A l s o , N
should be a prime.
Another approach to avoid clustering with open addressing is to iteratively try
buckets A[(h(k)+f(i))mod N]where f(i)is based on a pseudo-random number
generator, providing a repeatable, but somewhat arbitrary, sequence of subsequentprobes that depends upon bi
rently used by Python·s dictionary class.

10.2.4 Python Hash Table Implementation
In this section, we develop two implementations of a hash table, one using sepa-
rate chaining and the other using open addressing with linear probing. While theseapproaches to collision resolu
class (from Code Fragment 10.2), to de·ne a new HashMapBase class (see Code
Fragment 10.4), providing much of the common functionality to our two hash tableimplementations. The main de
·The bucket array is represented as a Python list, named self.
table , with all
entries initialized to None .
·We maintain an instance variable self.
nthat represents the number of dis-
tinct 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 de·ne 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 independentstructure. With open address
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 jfor 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 jso that key kbecomes 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 incre-
menting self.
n.
·
bucket
delitem(j, k)
This method should remove the item from bucket jhaving key k, or raise a
KeyError if no such item exists. (self.
nis decremented after this method.)
·
iter
This is the standard map method to iterate through all keys of the map. Ourbase class does not delegate this on

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

```
defhash
code(s):
mask = (1 <<32)·1 # limit to 32-bit integers
h=0
forcharacter ins:
h=( h <<5&m a s k ) |(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, ·ne-tuning is required when using a
cyclic-shift hash code, as we must wisely choose the amount to shift by for eachnew character. Our choice of a 5
shift amounts (see Table 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	

C-10.32 Perform experiments on our ChainHashMap andProbeHashMap classes
to measure its ef·ciency using random key sets and varying limits on the
load factor (see Exercise R-10.15).
C-10.33 Our implementation of separate chaining in ChainHashMap conserves
memory by representing empty buckets in the table as None , rather than
as empty instances of a secondary structure. Because many of these buck-ets will hold a single item, a better op
the table directly reference the
Item instance, and to reserve use of sec-
ondary containers for buckets that have two or more items. Modify our
implementation to provide this additional optimization.
C-10.34 Computing a hash code can be expensive, especially for lengthy keys. Inour hash table implementation
serting an item, and recompute each item·s hash code each time we resize
our table. Python·s dict class makes an interesting trade-off. The hash
code is computed once, when an item is inserted, and the hash code isstored as an extra ·eld of the item compo
C-10.35 Describe how to perform a removal from a hash table that uses linear
probing to resolve collisions where we do not use a special marker to
represent deleted elements. That is, we must rearrange the contents so thatit appears that the removed entry wa
C-10.36 The quadratic probing strategy has a clustering problem related to the wayit looks for open slots. Namel
checks buckets A[(h(k)+ i
2)mod N],f o r i=1,2,..., N·1.
a. Show that i2mod Nwill assume at most (N+1)/2 distinct values,
for Nprime, as iranges from 1 to N·1. As a part of this justi·ca-
tion, note that i2mod N=(N·i)2mod Nfor all i.
b. A better strategy is to choose a prime Nsuch that Nmod 4 =3a n d
then to check the buckets A[(h(k)±i2)mod N]asiranges from 1
to(N·1)/2, alternating between plus and minus. Show that this
alternate version is guaranteed to check every bucket in A.
C-10.37 Refactor our ProbeHashMap design so that the sequence of secondary
probes for collision resolution can be more easily customized. Demon-strate your new framework by providing se
C-10.38 Design a variation of binary search for performing the multimap opera-tion·nd
all(k) implemented with a sorted search table that includes du-
plicates, and show that it runs in time O(s+logn),w h e r e nis the number
of elements in the dictionary and sis the number of items with given key k.

1classHashMapBase(MapBase):
2···Abstract base class for map using hash-table with MAD compression.···
3
4def
init
(self, cap=11, p=109345121):
5 ···Create an empty hash-table map.···
6 self.
table = cap
[None]
7 self.
n=0 # number of entries in the map
8 self.
prime = p # prime for MAD compression
9 self.
scale = 1 + randrange(p ·1) #s c a l ef r o m1t op - 1f o rM A D
10 self.
shift = randrange(p) # shift from 0 to p-1 for MAD
11
12def
hash
function(self,k) :
13 return (hash(k)
self.
scale + self.
shift) % self.
prime % len(self.
table)
14
15def
len
(self):
16 return self .
n
1718def
getitem
(self,k) :
19 j=self.
hash
function(k)
20 return self .
bucket
getitem(j, k) # may raise KeyError
21
22def
setitem
(self,k ,v) :
23 j=self.
hash

R-10.12 What is the result of Exercise R-10.9 when collisions are handled by dou-
ble hashing using the secondary hash function h/prime(k)=7·(kmod 7)?
R-10.13 What is the worst-case time for putting nentries in an initially empty hash
table, with collisions resolved by chaining? What is the best case?
R-10.14 Show the result of rehashing the hash table shown in Figure 10.6 into a
table of size 19 using the new hash function h(k)=3kmod 17.
R-10.15 OurHashMapBase class maintains a load factor ··0.5. Reimplement
that class to allow the user to specify the maximum load, and adjust the
concrete subclasses accordingly.
R-10.16 Give a pseudo-code description of an insertion into a hash table that uses
quadratic probing to resolve collisions, assuming we also use the trick ofreplacing deleted entries with a special
R-10.17 Modify our ProbeHashMap to use quadratic probing.
R-10.18 Explain why a hash table is not suited to implement a sorted map.
R-10.19 Describe how a sorted list implemented as a doubly linked list could beused to implement the sorted ma
R-10.20 What is the worst-case asymptotic running time for performing ndeletions
from a SortedTableMap instance that initially contains 2 nentries?
R-10.21 Consider the following variant of the
·nd
index method from Code Frag-
ment 10.8, in the context of the SortedTableMap class:
def
·nd
index(self,k ,l o w ,h i g h) :
ifhigh<low:
return high + 1
else:
mid = (low + high) // 2
if self.
table[mid].
key<k:
return self .
·nd
index(k, mid + 1, high)
else:
return self .
·nd
index(k, low, mid ·1)
Does this always produce the same result as the original version? Justifyyour answer.
R-10.22 What is the expected running time of the methods for maintaining a max-
ima set if we insert npairs such that each pair has lower cost and perfor-
mance than one before it? What is contained in the sorted map at the end
of this series of operations? What if each pair had a lower cost and higher
performance than the one before it?
R-10.23 Draw an example skip list Sthat results from performing the following
series of operations on the skip list shown in Figure 10.13: del S[38] ,
S[48] =
x
,S[24] =
y

C-10.49 Python·s collections module provides an OrderedDict class that is unre-
lated to our sorted map abstraction. An OrderedDict is a subclass of the
standard hash-based dictclass that retains the expected O(1)performance
for the primary map operations, but that also guarantees that the
iter
method reports items of the map according to ·rst-in, ·rst-out (FIFO)
order. That is, the key that has been in the dictionary the longest is re-
ported ·rst. (The order is unaffected when the value for an existing key
is overwritten.) Describe an algorithmic approach for achieving such per-formance.
Projects
P-10.50 Perform a comparative analysis that studies the collision rates for various
hash codes for character strings, such as various polynomial hash codes
for different values of the parameter a. Use a hash table to determine
collisions, but only count collisions where different strings map to thesame hash code (not if they map to the sam
Test these hash codes on text ·les found on the Internet.
P-10.51 Perform a comparative analysis as in the previous exercise, but for 10-digit
telephone numbers instead of character strings.
P-10.52 Implement an OrderedDict class, as described in Exercise C-10.49, en-
suring that the primary map operations run in O(1)expected time.
P-10.53 Design a Python class that implements the skip-list data structure. Use
this class to create a complete implementation of the sorted map ADT.
P-10.54 Extend the previous project by providing a graphical animation of the
skip-list operations. Visualize how entries move up the skip list duringinsertions and are linked out of the skip list
P-10.55 Write a spell-checker class that stores a lexicon of words, W, in a Python
set, and implements a method, check (s), which performs a spell check
on the string swith respect to the set of words, W.I f sis in W,t h e n
the call to check (s) returns a list containing only s, as it is assumed to
be spelled correctly in this case. If sis not in W, then the call to check (s)
returns a list of every word in Wthat might be a correct spelling of s. Your
program should be able to handle all the common ways that smight be a
misspelling of a word in W, including swapping adjacent characters in a
word, inserting a single character in between two adjacent characters in aword, deleting a single character from a
a word with another character. For an extra challenge, consider phonetic
substitutions as well.

Open Addressing
The separate chaining rule has many nice properties, such as affording simple im-
plementations 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 collid-ing keys. If space is at a premium (
structures are employed, but it requires a bit more complexity to deal with colli-
sions. 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.
Linear Probing and Its Variants
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[j]that is already
occupied, where j=h(k), then we next try A[(j+1)mod N].I fA[(j+1)mod N]
is also occupied, then we try A[(j+2)mod N], and so on, until we ·nd an empty
bucket that can accept the new item. Once this bucket is located, we simply in-
sert the item there. Of course, this collision resolution strategy requires that wechange the implementation when
getitem
 ,
setitem
 ,o r
 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 ·nd an item with that key or we ·nd an empty bucket.
(See Figure 10.7.) The name ·linear probing· comes from the fact that accessing acell of the bucket array can be
26123456789 1 0 0New element with
key = 15 to be insertedMust probe 4 times
before ·nding empty slot
53 7 1 6 2 1 13
Figure 10.7: Insertion into a hash table with integer keys using linear probing. The
hash function is h(k)=kmod 11. Values associated with keys are not shown.

Separate Chaining
Code Fragment 10.5 provides a concrete implementation of a hash table with sepa-
rate 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 ·rst three methods in the class use index jto 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 thatare already supported by the in
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).
1classChainHashMap(HashMapBase):
2···Hash map implemented with separate chaining for collision resolution.···
34def
bucket
getitem(self,j ,k) :
5 bucket = self.
table[j]
6 ifbucket is None :
7 raiseKeyError(
Key Error:
+r e p r (k)) # no match found
8 return bucket[k] # may raise KeyError
9
10def
bucket
setitem(self,j ,k ,v) :
11 if self.
table[j] is None :
12 self.
table[j] = UnsortedTableMap( ) # bucket is new to the table
13 oldsize = len(self.
table[j])
14 self.
table[j][k] = v
15 iflen(self.
table[j]) >oldsize: # key was new to the table
16 self.
n+ =1 # increase overall map size
1718def
bucket
delitem(self,j ,k) :
19 bucket = self.
table[j]
20 ifbucket is None :
21 raiseKeyError(

Chapter

10Maps, Hash Tables, and Skip Lists

Contents

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 distinguishbetween it and a location that ha
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 thisend, we de·ne a nonpublic utility
·nd
slot, that searches for an item with key k
in ·bucket· j(that is, where jis the index returned by the hash function for key k).
1classProbeHashMap(HashMapBase):
2···Hash map implemented with linear probing for collision resolution.···
3
 AVAIL = object( ) # sentinal marks locations of previous deletions
45def
is
available(self,j) :
6 ···Return True if index j is available in table.···
7 return self .
table[j] is None or self .
table[j] isProbeHashMap.
 AVAIL
89def
·nd
slot(self,j ,k) :
10 ···Search for key k in bucket at index j.
1112 Return (success, index) tuple, described as follows:
13 If match was found, success is True and index denotes its location.
14 If no match found, success is False and index denotes ·rst available slot.
15 ···
16 ·rstAvail = None
17 while True :
18 if self.
is
available(j):
19 if·rstAvail is None :
20 ·rstAvail = j # mark this as ·rst avail
21 if self.
 table[j] is None :
22 return (False,· r s t A v a i l) # search has failed
23 elifk= =self.
table[j].
 key:
24 return (True,j) # found a match
25 j=(j+1)%l e n (self.
table) # keep looking (cyclically)

10.3.1 Sorted Search Tables
Several data structures can ef·ciently 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 sim
their keys, assuming the keys have a naturally de·ned order. (See Figure 10.8.) We
refer to this implementation of a map as a sorted search table .
9 2 4 5 7 8 12 14 17 19 22 25 27 28 335
3701234 6789 1 0 1 1 1 2 1 3 1 4 1 5
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 advan-
tage of this representation, and our reason for insisting that Abe array-based, is that
it allows us to use the binary search algorithm for a variety of ef·cient 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 ofFalse 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 de-
termine 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 thecollection that are just less than o
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 portrayin Figure 10.8. Had we inste
ing that the missing target lies in the gap between values 19 and 22 in that example.

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 structuressimilar to those for a map.
·Asetis an unordered collection of elements, without duplicates, that typi-
cally supports ef·cient membership tests. In essence, elements of a set arelike keys of a map, but without any au
·Amultiset (also known as a bag) is a set-like container that allows duplicates.
·Amultimap 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 throughthe built-in classes frozenset a
frozenset being an immutable form. Both of those classes are implemented using
hash tables in Python.
Python·s collections module de·nes abstract base classes that essentially mirror
these built-in classes. Although the choice of names is counterintuitive, the abstractbase class collections.Set ma
base class collections.MutableSet is akin to the concrete setclass.
In our own discussion, we equate the ·set ADT· with the behavior of the built-
insetclass (and thus, the collections.MutableSet base class). We begin by listing
what we consider to be the ·ve most fundamental behaviors for a set S:
S.add(e) :Add element eto the set. This has no effect if the set
already contains e.
S.discard(e) :Remove element efrom the set, if present. This has no
effect if the set does not contain e.
ei nS :Return True if the set contains element e. In Python, this
is implemented with the special
contains
method.
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
.

R-10.24 Give a pseudo-code description of the
 delitem
 map operation when
using a skip list.
R-10.25 Give a concrete implementation of the pop method, in the context of a
MutableSet abstract base class, that relies only on the ·ve core set behav-
iors described in Section 10.5.2.
R-10.26 Give a concrete implementation of the isdisjoint method in the context
of the MutableSet abstract base class, relying only on the ·ve primary
abstract methods of that class. Your algorithm should run in O(min (n,m))
where nand mdenote the respective cardinalities of the two sets.
R-10.27 What abstraction would you use to manage a database of friends· birth-
days in order to support ef·cient queries such as ··nd all friends whosebirthday is today· and ··nd the friend who
Creativity
C-10.28 On page 406 of Section 10.1.3, we give an implementation of the methodsetdefault as it might appear i
While that method accomplishes the goal in a general fashion, its ef·-ciency is less than ideal. In particular, whe
getitem
 , and then a subse-
quent insertion via
 setitem
 . For a concrete implementation, such as
theUnsortedTableMap , this is twice the work because a complete scan
of the table will take place during the failed
 getitem
 , and then an-
other complete scan of the table takes place due to the implementation of
setitem
 . A better solution is for the UnsortedTableMap class to over-
ridesetdefault to provide a direct solution that performs a single search.
Give such an implementation of UnsortedTableMap.setdefault .
C-10.29 Repeat Exercise C-10.28 for the ProbeHashMap class.
C-10.30 Repeat Exercise C-10.28 for the ChainHashMap class.
C-10.31 For an ideal compression function, the capacity of the bucket array for ahash table should be a prime nu
·nding such a prime by using the sieve algorithm . In this algorithm, we
allocate a 2 Mcell Boolean array A, such that cell iis associated with the
integer i. We then initialize the array cells to all be ·true· and we ·mark
off· all the cells that are multiples of 2, 3, 5, 7, and so on. This processcan stop after it reaches a number larger t
2M. (Hint: Consider a
bootstrapping method for ·nding the primes up to·
2M.)

A trie Tfor a set Sof strings can be used to implement a set or map whose keys are the strings of S. Namely, we perform a search in Tfor a string Xby tracing down from the root the path indicated by the characters in X. If this path can be traced and terminates at a leaf node, then we know Xis a key in the map. For example, in the trie in Figure 13.10, tracing the path for ·bull· ends up at a leaf. If the path cannot be traced or the path can be traced but terminates at an internal node, then Xis not a key in the map. In the example in Figure 13.10, the path for ·bet· cannot be traced and the path for ·be· ends at an internal node. Neither such word is in the map.

It is easy to see that the running time of the search for a string of length mis O(m·|·|), because we visit at most m+1 nodes of Tand we spend O(|·|)time at each node determining the child having the subsequent character as a label. The O(|·|)upper bound on the time to locate a child with a given label is achievable, even if the children of a node are unordered, since there are at most |·|children. We can improve the time spent at a node to be O(log|·|)or expected O(1),b y mapping characters to children using a secondary search table or hash table at each node, or by using a direct lookup table of size |·|at each node, if |·|is suf·ciently small (as is the case for DNA strings). For these reasons, we typically expect a search for a string of length mto run in O(m)time.

From the discussion above, it follows that we can use a trie to perform a spe- cial type of pattern matching, called word matching , where we want to determine whether a given pattern matches one of the words of the text exactly. Word match- ing differs from standard pattern matching because the pattern cannot match anarbitrary substring of the text·onl
word of the original document must be added to the trie. (See Figure 13.11.) A simple extension of this scheme supports pre·x-matching queries. However, ar-bitrary occurrences of the pattern

To construct a standard trie for a set Sof strings, we can use an incremental algorithm that inserts the strings one at a time. Recall the assumption that no stringofSis a pre·x of another string. To insert a string Xinto the current trie T,w e trace the path associated with XinT, creating a new chain of nodes to store the remaining characters of Xwhen we get stuck. The running time to insert Xwith length mis similar to a search, with worst-case O(m·|·|)performance, or expected O(m)if using secondary hash tables at each node. Thus, constructing the entire trie for set Stakes expected O(n)time, where nis the total length of the strings of S.

There is a potential space inef·ciency in the standard trie that has prompted the development of the compressed trie , which is also known (for historical reasons) as the Patricia trie . Namely, there are potentially a lot of nodes in the standard trie that have only one child, and the existence of such nodes is a waste. We discuss thecompressed trie next.

```
#------------------------- nested Vertex class -------------------------
classVertex:
···Lightweight vertex structure for a graph.···
 slots
 =
_element

def
 init
(self,x ) :
···Do not call constructor directly. Use Graph
 si n s e r t
 vertex(x).···
self.
element = x

defelement( self):
···Return element associated with this vertex.···
return self .
element

def
hash
 (self): # will allow vertex to be a map/set key
return hash(id( self))
#------------------------- nested Edge class -------------------------
classEdge:
···Lightweight edge structure for a graph.···

slots
 =
_origin
 ,
_destination
 ,
_element
def
init
(self,u ,v ,x ) :
···Do not call constructor directly. Use Graph
 si n s e r t
 edge(u,v,x).···
self.
origin = u
self.
destination = v
self.
element = x
```

Preface vii

Contents and Organization

The chapters for this book are organized to provide a pedagogical path that starts
with the basics of Python programming and object-oriented design. We then add
foundational techniques like algorithm analysis and recursion. In the main portionof the book, we present fundam

1.Python Primer
2.Object-Oriented Programming
3.Algorithm Analysis
4.Recursion
5.Array-Based Sequences
6.Stacks, Queues, and Deques
7.Linked Lists
8.Trees
9.Priority Queues
10.Maps, Hash Tables, and Skip Lists
11.Search Trees
12.Sorting and Selection
13.Text Processing
14.Graph Algorithms
15.Memory Management and B-Trees
A.Character Strings in Python
B.Useful Mathematical Facts

A more detailed table of contents follows this preface, beginning on page xi.

Prerequisites

We assume that the reader is at least vaguely familiar with a high-level program-ming language, such as C, C++

·Variables and expressions.
·Decision structures (such as if-statements and switch-statements).
·Iteration structures (for loops and while loops).
·Functions (whether stand-alone or object-oriented methods).

For readers who are familiar with these concepts, but not with how they are ex-pressed in Python, we provide a p
give a comprehensive treatment of Python.

The set and frozenset Classes
Python·s setclass represents the mathematical notion of a set, namely a collection
of elements, without duplicates, and without an inherent order to those elements.
The major advantage of using a set, as opposed to a list,i st h a ti th a sah i g h l y
optimized method for checking whether a speci·c element is contained in the set.This is based on a data structur
topic of Chapter 10). However, there are two important restrictions due to the
algorithmic underpinnings. The ·rst is that the set does not maintain the elements
in any particular order. The second is that only instances of immutable types can be
added to a Python set. Therefore, objects such as integers, ·oating-point numbers,
and character strings are eligible to be elements of a set. It is possible to maintain a
set of tuples, but not a set of lists or a set of sets, as lists and sets are mutable. The
frozenset class is an immutable form of the settype, so it is legal to have a set of
frozensets.
Python uses curly braces {and}as delimiters for a set, for example, as {17}
or{
red
,
green
,
blue
}. The exception to this rule is that {}does not
represent an empty set; for historical reasons, it represents an empty dictionary(see next paragraph). Instead, th
If an iterable parameter is sent to the constructor, then the set of distinct elements
is produced. For example, set(
hello
)produces {
h
,
e
,
l
,
o
}.
The dict Class
Python·s dict class represents a dictionary ,o rmapping , from a set of distinct keys
to associated values . For example, a dictionary might map from unique student ID
numbers, to larger student records (such as the student·s name, address, and course
grades). Python implements a dictusing an almost identical approach to that of a
set, but with storage of the associated values.
A dictionary literal also uses curly braces, and because dictionaries were intro-
duced in Python prior to sets, the literal form {}produces an empty dictionary.
A nonempty dictionary is expressed using a comma-separated series of key:valuepairs. For example, the diction
ga
:
Irish
,
de
:

Common Built-In Functions
Calling Syntax
Description
abs(x)
Return the absolute value of a number.
all(iterable)
Return True ifbool(e) isTrue for each element e.
any(iterable)
Return True ifbool(e) isTrue for at least one element e.
chr(integer)
Return a one-character string with the given Unicode code point.
divmod(x, y)
Return (x // y, x % y) as tuple, if xandyare integers.
hash(obj)
Return an integer hash value for the object (see Chapter 10).
id(obj)
Return the unique integer serving as an ·identity· for the object.
input(prompt)
Return a string from standard input; the prompt is optional.
isinstance(obj, cls)
Determine if objis an instance of the class (or a subclass).
iter(iterable)
Return a new iterator object for t he parameter (see Section 1.8).
len(iterable)
Return the number of elements in the given iteration.
map(f, iter1, iter2, ...)
Return an iterator yielding the result of function calls f(e1, e2, ...)
for respective elements e1·iter1,e2·iter2,...
max(iterable)
Return the largest element of the given iteration.
max(a, b, c, ...)
Return the largest of the arguments.
min(iterable)
Return the smallest element of the given iteration.
min(a, b, c, ...)
Return the smallest of the arguments.
next(iterator)
Return the next element reported by the iterator (see Section 1.8).
open(·lename, mode)
Open a ·le with the given name and access mode.
ord(char)
Return the Unicode code point of the given character.
pow(x, y)
Return the value xy(as an integer if xandyare integers);
equivalent to x
y.
pow(x, y, z)
Return the value (xymod z)as an integer.
print(obj1, obj2, ...)

next number in a sequence based upon one or more past numbers that it has gen-
erated. Indeed, a simple yet popular pseudo-random number generator chooses its
next number based solely on the most recently chosen number and some additionalparameters using the followi
next =(a*current +b)%n;
where a,b,a n d nare appropriately chosen integers. Python uses a more advanced
technique known as a Mersenne twister . It turns out that the sequences generated
by these techniques can be proven to be statistically uniform, which is usually
good enough for most applications requiring random numbers, such as games. For
applications, such as computer security settings, where one needs unpredictablerandom sequences, this kind of
from outer space.
Since the next number in a pseudo-random generator is determined by the pre-
vious number(s), such a generator always needs a place to start, which is called its
seed. The sequence of numbers generated for a given seed will always be the same.
One common trick to get a different sequence each time a program is run is to use
a seed that will be different for each run. For example, we could use some timed
input from a user or the current system time in milliseconds.
Python·s random module provides support for pseudo-random number gener-
ation by de·ning a Random class; instances of that class serve as generators with
independent state. This allows different aspects of a program to rely on their ownpseudo-random number genera
random module (essentially using a single generator instance for all top-level calls).
Syntax
Description
seed(hashable)
Initializes the pseudo-random number generator
based upon the hash value of the parameter
random()
Returns a pseudo-random ·oating-point
value in the interval [0.0,1.0).
randint(a,b)
Returns a pseudo-random integer
in the closed interval [a,b].
randrange(start, stop, step)
Returns a pseudo-random integer in the standard
Python range indicated by the parameters.
choice(seq)
Returns an element of the given sequence
chosen pseudo-randomly.
shu·e(seq)
Reorders the elements of the given
sequence pseudo-randomly.
Table 1.8: Methods supported by instances of the Random class, and as top-level
functions of the random module.

Common Syntax
Special Method Form
a+b
a.
add
(b); alternatively b.
radd
(a)
a·b
a.
sub
(b); alternatively b.
rsub
(a)
a
b
a.
mul
(b); alternatively b.
rmul
(a)
a/b
a.
truediv
(b); alternatively b.
rtruediv
(a)
a/ /b
a.
·oordiv
(b); alternatively b.
r·oordiv
(a)
a%b
a.
mod
(b); alternatively b.
rmod
(a)
a
b
a.
pow
(b); alternatively b.
rpow
(a)
a<<b
a.
lshift

10.1 Maps and Dictionaries
Python·s dict class is arguably the most signi·cant 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 ormaps .I n t h i s book, we use the term dictionary when speci·cally discussing Python·s dictclass, and the term map when discussing the more general notion of the abstract data type.
As a simple example, Figure 10.1 illustrates a map from the names of countries to their associated units of currency.
RupeeTurkey Spain China United States India Greece
Lira Euro Yuan Dollar
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 in-dexing, such as currency[
Greece
]to access a value associated with a given key
orcurrency[
Greece
]=
Drachma
to remap it to a new value. Unlike a stan-
dard array, indices for a map need not be consecutive nor even numeric. Common applications of maps include the following.
·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.
·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.
·A social media site typically relies on a (nonnumeric) username as a key thatcan be ef·ciently mapped to a parti
·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) rep-resentation, such as (64,224,208) .
·Python uses a dictionary to represent each namespace, mapping an identifyingstring, such as
pi
, to an associated object, such as 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 ef·ciently,thereby supporting fast lookup

M.popitem() :Remove an arbitrary key-value pair from the map, and re-
turn a (k,v) tuple representing the removed pair. If map is
empty, raise a KeyError .
M.clear() :Remove all key-value pairs from the map.
M.keys() :Return a set-like view of all keys of M.
M.values() :Return a set-like view of all values of M.
M.items() :Return a set-like view of (k,v) tuples for all entries of M.
M.update(M2) :Assign M[k] = v for every (k,v) pair in map M2.
M= =M 2 :Return True if maps MandM2have identical key-value
associations.
M! =M 2 :Return True if maps MandM2do not have identical key-
value associations.
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 dictclass to describe the map contents.
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[

10.1.3 Python·s MutableMapping Abstract Base Class

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 de-
clared to be abstract in such a base class must be implemented by concrete sub-
classes. 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 .)

Thecollections module provides two abstract base classes that are relevant to
our current discussion: the Mapping andMutableMapping classes. The Mapping
class includes all nonmutating methods supported by Python·s dictclass, while the
MutableMapping class extends that to include the mutating methods. What we
de·ne as the map ADT in Section 10.1.1 is akin to the MutableMapping abstract
base class in Python·s collections module.

The signi·cance of these abstract base classes is that they provide a framework
to assist in creating a user-de·ned map class. In particular, the MutableMapping
class provides concrete implementations for all behaviors other than the ·rst ·ve
outlined in Section 10.1.1:

getitem

,

setitem

,

delitem

,

len

,a n d

iter

. As we implement the map abstraction with various data structures, as

long as we provide the ·ve core behaviors, we can inherit all other derived behav-iors by simply declaring Mutabl

To better understand the MutableMapping class, we provide a few examples of
how concrete behaviors can be derived from the ·ve core abstractions. For example,the

contains

method, supporting the syntax ki nM , could be implemented by
making a guarded attempt to retrieve self[k] to determine if the key exists.

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.

defsetdefault(self,k ,d) :

try:

return self [k] #i f

getitem

succeeds, return value

1classMapBase(MutableMapping):
2⋯Our own abstract base class that includes a nonpublic
Item class.⋯
3
4#------------------------------- nested
Item class -------------------------------
5class
Item:
6 ⋯Lightweight composite to store key-value pairs as map items.⋯
7
slots
=
_key
,
_value
89 def
init
(self,k ,v) :
10 self.
key = k
11 self.
value = v
1213 def
eq
(self,o t h e r) :
14 return self .
key == other.
key # compare items based on their keys
1516 def
ne
(self,o t h e r) :
17 return not (self== other) #o p p o s i t eo f
eq
18
19 def
lt
(self,o t h e r) :
20 return self .
key<other.
key # compare items based on their keys
Code Fragment 10.2: Extending the MutableMapping abstract base class to provide
a nonpublic
Item class for use in our various map implementations.
10.1.5 Simple Unsorted Map Implementation
We demonstrate the use of the MapBase class with a very simple concrete imple-
mentation 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.
An empty table is initialized as self.
table within the constructor for our map.

```
def
bucket
 getitem( self,j ,k ) :
found, s = self.
·nd
slot(j, k)
if not found:
raiseKeyError(
 Key Error:
 +r e p r ( k ) ) # no match found
return self .
table[s].
 value

def
bucket
 setitem( self,j ,k ,v ) :
found, s = self.
·nd
slot(j, k)
if not found:
self.
table[s] = self.
Item(k,v) # insert new item
self.
n+ =1 # size has increased
else:
self.
table[s].
 value = v # overwrite existing

def
bucket
 delitem( self,j ,k ) :
found, s = self.
·nd
slot(j, k)
if not found:
raiseKeyError(
 Key Error:
 +r e p r ( k ) ) # no match found
self.
table[s] = ProbeHashMap.
 AVAIL # mark as vacated

def
iter
(self):
forjinrange(len( self.
table)): # scan entire table
if not self .
```

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 ·nancial 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 servingas the key, and a record about t
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 relieson the intentionally scattering
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.·nd
min() :Return the (key,value) pair with minimum key(orNone , if map is empty).
M.·nd
max() :Return the (key,value) pair with maximum key
(orNone , if map is empty).
M.·nd
lt(k) :Return the (key,value) pair with the greatest key that
is strictly less than k(orNone ,i fn os u c hi t e me x i s t s) .
M.·nd
le(k) :Return the (key,value) pair with the greatest key thatis less than or equal to k(orNone ,i fn os u c hi t e m
exists).
M.·nd
gt(k) :Return the (key,value) pair with the least key that isstrictly greater than k(orNone ,i fn os u c hi t e me x i s t
M.·nd
ge(k) :Return the (key,value) pair with the least key that isgreater than or equal to k(orNone ,i fn os u c hi t e m)
M.·nd
range(start, stop) :Iterate all (key,value) pairs with start<=k e y <stop.
Ifstart isNone , iteration begins with minimum key; if
stop isNone , 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.

```
classSortedTableMap(MapBase):
···Map implementation using a sorted table.···

#---------------------------- nonpublic behaviors ----------------------------
def ·nd index(self,k ,l o w ,h i g h ) :
···Return index of the leftmost item with key greater than or equal to k.
Return high + 1 if no such item quali·es.

That is, j will be returned such that:
all items of slice table[low:j] have key <k
all items of slice table[j:high+1] have key >=k
···
ifhigh<low:
return high + 1 # no element quali·es
else:
mid = (low + high) // 2
ifk= =self. table[mid]. key:
return mid # found exact match
elifk<self. table[mid]. key:
return self . ·nd index(k, low, mid ·1) # Note: may return mid
else:
return self . ·nd index(k, mid + 1, high) # answer is right of mid

#---------------------------- public behaviors ----------------------------
def init (self):
···Create an empty map.···
self. table = [ ]

def len (self):
···Return number of items in the map.···
return len(self. table)

def getitem (self,k ) :
···Return value associated with key k (raise KeyError if not found).···
```

def·nd
ge(self,k) :
···Return (key,value) pair with least key greater than or equal to k.···
j=self.
·nd
index(k, 0, len(self.
table) ·1) #j
sk e y>=k
ifj<len(self.
table):
return (self.
table[j].
 key,self.
table[j].
 value)
else:
return None
def·nd
lt(self,k) :
···Return (key,value) pair with greatest key strictly less than k.···
j=self.
·nd
index(k, 0, len(self.
table) ·1) #j
sk e y>=k
ifj>0:
return (self.
table[j ·1].
key,self.
table[j ·1].
value) # Note use of j-1
else:
return None
def·nd
gt(self,k) :
···Return (key,value) pair with least key strictly greater than k.···
j=self.
·nd
index(k, 0, len(self.
table) ·1) #j
sk e y>=k
ifj<len(self.
table) and self .
table[j].
 key == k:
j+ =1 # advanced past match
ifj<len(self.
table):

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·ight databases to ·nd ·ights be
buy a ticket. To make a query, a user speci·es origin and destination cities, a depar-
ture date, and a departure time. To support such queries, we can model the ·ightdatabase as a map, where keys
to these four parameters. That is, a key is a tuple
k=(origin,destination ,date,time).
Additional information about a ·ight, such as the ·ight number, the number of seatsstill available in ·rst (F) and co
be stored in the value object.
Finding a requested ·ight is not simply a matter of ·nding an exact match
for a requested query. Although a user typically wants to exactly match the ori-
gin and destination cities, he or she may have ·exibility for the departure date,and certainly will have some ·exibi
We can handle such a query by ordering our keys lexicographically. Then, an ef-
·cient implementation for a sorted map would be a good way to satisfy users·queries. For instance, given a user
ge(k) to return
the ·rst ·ight between the desired cities, having a departure date and time match-ing the desired query or later. B
use·nd
range(k1, k2) to ·nd all ·ights within a given range of times. For exam-
ple, if k1=(ORD, PVD, 05May, 09:30),a n d k2=(ORD, PVD, 05May, 20:00),
a respective call to ·nd
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)

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 ·eld and performance (speed) is the value ·eld. We can then implement operationsadd (c,p), which adds a new cost-performance pair (c,p),a n dbest (c),w h i c h returns the best pair with cost at most c, as shown in Code Fragment 10.11.

1classCostPerformanceDatabase:
2···Maintain a database of maximal (cost,performance) pairs.···
3
4def
 init
(self):
5 ···Create an empty database.···
6 self.
M = SortedTableMap( ) # or a more e·cient sorted map
78defbest(self,c) :
9 ···Return (cost,performance) pair with largest cost not exceeding c.
1011 Return None if there is no such pair.
12 ···
13 return self .
M.·nd
 le(c)
1415defadd(self,c ,p) :
16 ···Add new entry with cost c and performance p.···
17 # determine if (c,p) is dominated by an existing pair
18 other = self.
M.·nd
 le(c) # other is at least as cheap as c
19 ifotheris not None and other[1] >=p :# if its performance is as good,
20 return # (c,p) is dominated, so ignore
21 self.
M[c] = p # else, add (c,p) to database
22 # and now remove any pairs that are dominated by (c,p)
23 other = self.
M.·nd
 gt(c) # other more expensive than c
24 while otheris not None and other[1] <=p :
25 del self .
M[other[0]]
26 other = self.
M.·nd
 gt(c)

Code Fragment 10.11: An implementation of a class maintaining a set of maxima cost-performance pairs using a sorted map.

Unfortunately, if we implement Musing the SortedTableMap ,t h eaddbehavior has O(n)worst-case running time. If, on the other hand, we implement Musing a skip list, which we next describe, we can perform best (c)queries in O(logn) expected time and add (c,p)updates in O((1+r)logn)expected time, where ris the number of points removed.

and place that item in Si+1if the coin comes up ·heads.· Thus, we expect S1to have
about n/2 items, S2to have about n/4 items, and, in general, Sito have about n/2i
items. In other words, we expect the height hofSto 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 computergames, cryptography, and con
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, wewill assume that our computer h
The main advantage of using randomization in data structure and algorithm
design is that the structures and functions that result are usually simple and ef·cient.
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 methodswhen inserting or deleting items.
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(logn)on average ,w h e r e nis the number of items
in the map. Interestingly, the notion of average time complexity used here does notdepend on the probability dist
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 S
iand each tower contains positions storing
the same item across consecutive lists. The positions in a skip list can be traversedusing the following operations
next(p) :Return the position following pon the same level.
prev(p) :Return the position preceding pon the same level.
below(p) :Return the position below pin the same tower.
above(p) :Return the position above pin 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 eas-ily implement a skip list by mea
structure is essentially a collection of hdoubly linked lists aligned at towers, which
are also doubly linked lists.

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

Code Fragment 10.12: Algorthm to search a skip list Sfor key k.

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

I n s e r t i o ni naS k i pL i s t

The execution of the map operation M[k] = v begins with a call to SkipSearch (k). This gives us the position pof the bottom-level item with the largest key less than or equal to k(note that pmay hold the special item with key ··). Ifkey (p)= k,t h e 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 pwithin S_0. After inserting the new item at the bottom level, we use randomization to decide the height of the tower for the new item. We ··ip· a coin, and if the ·ip comes up tails, then we stop here. Else (the ·ip comes up heads), we backtrack to the previous (next higher) level and insert (k,v)in this level at the appropriate position. We again ·ip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continueto insert the new item (k,v)in lis
We link together all the references to the new item (k,v)created in this process to create its tower. A coin ·ip can be simulated with Python·s built-in pseudo-randomnumber generator from the ran
0 or 1, each with probability 1 /2.

We give the insertion algorithm for a skip list Sin 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 ,a n dbelow methods will work correctly for p, q,a n d r). The expected running time of the insertion algorithm on a skip list with nentries is O(logn), which we show in Section 10.4.2.

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 perform the map operation del M[k] we begin by executing method SkipSearch (k). If the position pstores an entry with key different from k, we raise a KeyError . Otherwise, we remove pand all the positions above p, which are easily accessed by using above operations to climb up the tower of this entry in Sstarting at position p. While removing levels of the tower, we reestablish links between the horizontal neighbors of each removed position. The removal algorithm is illustratedin Figure 10.13 and a detailed de
O(logn)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 needto store references to values at
more ef·ciently represent a tower as a single object, storing the key-value pair, and maintaining jprevious references and jnext references if the tower reaches level S
j. 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 strictlya top-down, scan-forward fas
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 p
search trees, which are discussed in Chapter 11.

31S5
S4
S3
S2
S1-·-·
-· 1212 -·
1717 25
25 31
3142
55 5055+·
+·+·
+·
+·-·-·
17
38
38 39 424242
44
445555+·
17
17
20 2525
S0

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 S0holding the entry are highlighted. The positions removed are drawn with dashed lines.

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 ·rst, this might seem like a major undertaking,
for a complete and thorough probabilistic analysis could require deep mathemat-ics (and, indeed, there are seve
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 hof a skip list S
with nentries (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 ·ipping a coin, that is, this probability is 1 /2
i. Hence, the
probability Pithat level ihas at least one position is at most
Pi·n
2i,
for the probability that any one of ndifferent events occurs is at most the sum of
the probabilities that each occurs.
The probability that the height hofSis larger than iis equal to the probability
that level ihas at least one position, that is, it is no more than Pi. This means that h
is larger than, say, 3log nwith 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,his larger than clognwith probability at most
1/nc·1. That is, the probability that his smaller than clognis at least 1 ·1/nc·1.
Thus, with high probability, the height hofSisO(logn).
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
al e v e lo f Sas 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 theheight hofSisO(logn)with high
O(logn)with high probability.

10.5.2 Python·s MutableSet Abstract Base Class

To aid in the creation of user-de·ned set classes, Python·s collections module provides 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 ·ve core behaviors (add,discard ,
contains
 ,
len
 ,a n d
 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 bya 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,o t h e r ) : # supports syntax S <T
···Return true if this set is a proper subset of other.···iflen(self)>= len(other):
return False # proper subset must have strictly smaller size
forein self :
ifenot 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|Tshould produce a new set that has contents equal to the union of existing sets Sand T.T h i s operation is implemented through the special method
 or
 in Python. Another
syntax, S|=T is used to update existing set Sto become the union of itself and set T. Therefore, all elements of Tthat are not already contained in Sshould be added to S. We note that this ·in-place· operation may be implemented more ef·ciently than if we were to rely on the ·rst form, using the syntax S=S |T,i n which identi·er Sis reassigned to a new set instance that represents the union. For convenience, Python·s built-in set class supports named version of these behaviors,withS.union(T) equivalent to
those named versions are not formally provided by the MutableSet abstract base class).

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. Anydata structure used to implement
storing set elements as keys, and using None as an irrelevant value, but such an
implementation is unnecessarily wasteful. An ef·cient 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 structureswe have seen can be reimpler
which the map key is a (distinct) element of the multiset, and the associated valueis a count of the number of occ
Python·s standard collections module includes a de·nition 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 amost
common(n) method that returns a list of the nmost 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 imple-mentation approach is to use a st
uses the standard dictclass 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 implementationcan easily be substituted by ov
MapType attribute at line 3.

C-10.39 Although keys in a map are distinct, the binary search algorithm can be
applied in a more general setting in which an array stores possibly duplica-
tive elements in nondecreasing order. Consider the goal of identifying theindex of the leftmost element with key g
Does the
·nd
index method as given in Code Fragment 10.8 guarantee
such a result? Does the
·nd
index method as given in Exercise R-10.21
guarantee such a result? Justify your answers.
C-10.40 Suppose we are given two sorted search tables Sand T, each with nentries
(with Sand Tbeing implemented with arrays). Describe an O(log2n)-
time algorithm for ·nding the kthsmallest key in the union of the keys
from Sand T(assuming no duplicates).
C-10.41 Give an O(logn)-time solution for the previous problem.
C-10.42 Suppose that each row of an n×narray Aconsists of 1·s and 0·s such that,
in any row of A, all the 1·s come before any 0·s in that row. Assuming A
is already in memory, describe a method running in O(nlogn)time (not
O(n2)time!) for counting the number of 1·s in A.
C-10.43 Given a collection Cofncost-performance pairs (c,p), describe an algo-
rithm for ·nding the maxima pairs of CinO(nlogn)time.
C-10.44 Show that the methods above (p)andprev (p)are not actually needed to
ef·ciently implement a map using a skip list. That is, we can imple-ment insertions and deletions in a skip list usi
In the insertion algorithm, ·rst repeatedly ·ip the coin to determine thelevel where you should start inserting the n
C-10.45 Describe how to modify a skip-list representation so that index-based
operations, such as retrieving the item at index j, can be performed in
O(logn)expected time.
C-10.46 For sets Sand T, the syntax S·T returns a new set that is the symmet-
ric difference, that is, a set of elements that are in precisely one of Sor
T. This syntax is supported by the special
xor
method. Provide an
implementation of that method in the context of the MutableSet abstract
base class, relying only on the ·ve primary abstract methods of that class.
C-10.47 In the context of the MutableSet abstract base class, describe a concrete
implementation of the
and
method, which supports the syntax S&T
for computing the intersection of two existing sets.
C-10.48 Aninverted ·le is a critical data structure for implementing a search en-
gine or the index of a book. Given a document D, which can be viewed
as an unordered, numbered list of words, an inverted ·le is an ordered list
of words, L, such that, for each word winL, we store the indices of the
places in Dwhere wappears. Design an ef·cient algorithm for construct-
ing Lfrom D.

Chapter Notes
Hashing is a well-studied technique. The reader interested in further study is encouraged to explore the book by Knuth [65], as well as the book by Vitter and Chen [100]. Skip lists were introduced by Pugh [86]. Our analy sis of skip lists is a simpli·cation of a presentation given by Motwani and Raghavan [80]. For a more in-depth analysis of skip lists, please see the various research papers on skip lists that have appeared in the data structures literature [59, 81, 84]. Exercise C-10.36 was contributed by James Lee.

another occurrence. The ef·ciency of the Boyer-Moore algorithm relies on creating a lookup table that quickly determines where a mismatched character occurs elsewhere in the pattern. In particular, we de·ne a function last (c)as

·Ifcis inP,last (c)is the index of the last (rightmost) occurrence of cinP. Otherwise, we conventionally de·ne last (c)=·1.

If we assume that the alphabet is of ·xed, ·nite size, and that characters can be converted to indices of an array (for example, by using their character code), the lastfunction can be easily implemented as a lookup table with worst-case O(1)-time access to the value last (c). However, the table would have length equal to the size of the alphabet (rather than the size of the pattern), and time would be required to initialize the entire table.

We prefer to use a hash table to represent the lastfunction, with only those characters from the pattern occurring in the structure. The space usage for thisapproach is proportional to the nu
the pattern, and thus O(m). The expected lookup time remains independent of the problem (although the worst-case bound is O(m)). Our complete implementation of the Boyer-Moore pattern-matching algorithm is given in Code Fragment 13.2.

```
def·nd
boyer
 moore(T, P):
···Return the lowest index of T at which substring P begins (or else -1).···
n, m = len(T), len(P) # introduce convenient notations
ifm= =0 : return 0 # trivial search for empty string
last = {} # build ·last· dictionary
forkinrange(m):
l a s t [P [ k ]]=k # later occurrence overwrites
# align end of pattern at index m-1 of text
i=m ·1 # an index into T
k=m ·1 # an index into P
while i<n:
ifT[i] == P[k]: # a matching character
ifk= =0 :
return i # pattern begins at index i of text
else:
i·=1 # examine previous character
k·=1 #o fb o t hTa n dP
else:
j = last.get(T[i], ·1) # last(T[i]) is -1 if not found
i+ =m ·min(k, j + 1) # case analysis for jump step
k=m ·1 # restart at end of pattern
return ·1
```

Code Fragment 13.2: An implementation of the Boyer-Moore algorithm.

The correctness of the Boyer-Moore pattern-matching algorithm follows from
the fact that each time the method makes a shift, it is guaranteed not to ·skip· over
any possible matches. For last(c)is the location of the lastoccurrence of cinP.
In Figure 13.4, we illustrate the execution of the Boyer-Moore pattern-matchingalgorithm on an input string simila
c
abc d
last (c)
453· 1
a c d a b a a cb aabc Text:
Pattern: b aaa b c1
baaa b c2 3 4
baaa b c5
baaa b c6baaa b c7baaa b c8 9 10 11 12 13b b a a a b a
Figure 13.4: An illustration of the Boyer-Moore pattern-matching algorithm, in-
cluding a summary of the last (c)function. The algorithm performs 13 character
comparisons, which are indicated with numerical labels.
Performance
If using a traditional lookup table, the worst-case running time of the Boyer-Moorealgorithm is O(nm +|·|). Namely
O(m+|·|), and the actual search for the pattern takes O(nm)time in the worst case,
the same as the brute-force algorithm. (With a hash table, the dependence on |·|is
removed.) An example of a text-pattern pair that achieves the worst case is
T = n/bracehtipdownleft
/bracehtipupright/bracehtipupleft
/bracehtipdownrightaaaaaa ···a
P = bm·1/bracehtipdownleft
/bracehtipupright/bracehtipupleft
/bracehtipdownrightaa···a
The worst-case performance, however, is unlikely to be achieved for English text,for, in that case, the Boyer-Moo
comparisons done per character is 0 .24 for a ·ve-character pattern string.
We have actually presented a simpli·ed version of the Boyer-Moore algorithm.
The original algorithm achieves running time O(n+m+|·|)by using an alternative
shift heuristic to the partially matched text string, whenever it shifts the pattern
more than the character-jump heuristic. This alternative shift heuristic is based on
applying the main idea from the Knuth-Morris-Pratt pattern-matching algorithm,
which we discuss next.

14.2.5 Python Implementation
In this section, we provide an implementation of the Graph ADT. Our implementa-
tion will support directed or undirected graphs, but for ease of explanation, we ·rstdescribe it in the context of an
We use a variant of the adjacency map representation. For each vertex v,w e
use a Python dictionary to represent the secondary incidence map I(v).H o w e v e r ,
we do not explicitly maintain lists Vand E, as originally described in the edge list
representation. The list Vis replaced by a top-level dictionary Dthat maps each
vertex vto its incidence map I(v); note that we can iterate through all vertices by
generating the set of keys for dictionary D. By using such a dictionary Dto map
vertices to the secondary incidence maps, we need not maintain references to thoseincidence maps as part of th
inO(1)expected time. This greatly simpli·es our implementation. However, a
consequence of our design is that some of the worst-case running time bounds forthe graph ADT operations, giv
than maintain list E, we are content with taking the union of the edges found in the
various incidence maps; technically, this runs in O(n+m)time rather than strictly
O(m)time, as the dictionary Dhas nkeys, even if some incidence maps are empty.
Our implementation of the graph ADT is given in Code Fragments 14.1 through
14.3. Classes Vertex andEdge , given in Code Fragment 14.1, are rather simple,
and can be nested within the more complex Graph class. Note that we de·ne the
hash
method for both Vertex andEdge so that those instances can be used as
keys in Python·s hash-based sets and dictionaries. The rest of the Graph class is
given in Code Fragments 14.2 and 14.3. Graphs are undirected by default, but canbe declared as directed with a
Internally, we manage the directed case by having two different top-level dictio-
nary instances,
outgoing and
incoming , such that
outgoing[v] maps to another
dictionary representing Iout(v),a n d
incoming[v] maps to a representation of Iin(v).
In order to unify our treatment of directed and undirected graphs, we continue touse the
outgoing and
incoming identi·ers in the undirected case, yet as aliases
to the same dictionary. For convenience, we de·ne a utility named is
directed to
allow us to distinguish between the two cases.
For methods degree andincident
edges , which each accept an optional param-
eter to differentiate between the outgoing and incoming orientations, we choose theappropriate map before proc
vertex , we always initial-
ize
outgoing[v] to an empty dictionary for new vertex v. In the directed case, we
independently initialize
incoming[v] as well. For the undirected case, that step is
unnecessary as
outgoing and
incoming are aliases. We leave the implementations
of methods remove
vertex andremove
edge as exercises (C-14.37 and C-14.38).

Index

14.3.2 DFS Implementation and Extensions

We begin by providing a Python implementation of the basic depth-·rst search algorithm, originally described with pseudo-code in Code Fragment 14.4. Our DFS function is presented in Code Fragment 14.5.

```
defDFS(g, u, discovered):
···Perform DFS of the undiscovered portion of Graph g starting at Vertex u.
discovered is a dictionary mapping each vertex to the edge that was used to
discover it during the DFS. (u should be ·discovered· prior to the call.)
Newly discovered vertices will be added to the dictionary as a result.
···
foreing.incident
edges(u): # for every outgoing edge from u
v=e . o p p o s i t e ( u )
ifvnot in discovered: # v is an unvisited vertex
discovered[v] = e # e is the tree edge that discovered v
DFS(g, v, discovered) # recursively explore from v
```

Code Fragment 14.5: Recursive implementation of depth-·rst search on a graph, starting at a designated vertex u.

In order to track which vertices have been visited, and to build a representation of the resulting DFS tree, our implementation introduces a third parameter, nameddiscovered . This parameter sl graph to the tree edge that was used to discover that vertex. As a technicality, weassume that the source vertex value. Thus, a caller might start the traversal as follows:

```
result = {u:None} # a new dictionary, with u trivially discovered
DFS(g, u, result)
```

The dictionary serves two purposes. Internally, the dictionary provides a mecha-nism for recognizing visited verti values within the dictionary are the DFS tree edges at the conclusion of the process.

Because the dictionary is hash-based, the test, · ifvnot in discovered ,· and the record-keeping step, · discovered[v] = e ,· run in O(1)expected time, rather than worst-case time. In practice, this is a compromise we are willing to accept,but it does violate the formal anal could be used as indices into an array-based lookup table rather than a hash-basedmap. Alternatively, we could tree edge directly as part of the vertex instance.

Chapter Notes
Some of the data structures discussed in this chapter are extensively covered by Knuth in his Sorting and Searching book [65], and by Mehlhorn in [76]. A VL trees are due to Adel·son-Vel·skii and Landis [2], who invented this class of balanced search trees in 1962. Binary search trees, A VL trees, and hashing are described in Knuth·s Sorting and Searching[65] book. Average-height analyses for binary search trees can be found in the books by Aho, Hopcroft, and Ullman [6] and Cormen, Leiserson, Rivest and Stein [29]. The handbook by Gonnet and Baeza-Yates [44] contains a number of theoretical and experimentalcomparisons among m
trees, which are similar to (2,4)trees. Red-black trees were de·ned by Bayer [10]. Variations and interesting properties of red-black trees are presented in a paper by Guibas and Sedgewick [48]. The reader interested in learning more about different balanced tree datastructures is referred to
by Mehlhorn and Tsakalidis [78]. Knuth [65] i s excellent additional r eading that includes early approaches to balancing trees. Splay trees were invented by Sleator and Tarjan [89] (see also [95]).

[85] R. C. Prim, ·Shortest connection networks and some generalizations,· Bell Syst. Te ch . J. , vol. 36, pp. 1389·1401, 1957.
[86] W. Pugh, ·Skip lists: a probabilistic alternative to balanced trees,· Commun. ACM , vol. 33, no. 6, pp. 668·676, 1990.
[87] H. Samet, The Design and Analysis of Spatial Data Structures . Reading, MA: Addison-Wesley, 1990.
[88] R. Schaffer and R. Sedgewick, ·The analysis of heapsort,· Journal of Algorithms , vol. 15, no. 1, pp. 76·100, 1993.
[89] D. D. Sleator and R. E. Tarjan, ·Self-adjusting binary search trees,· J. ACM , vol. 32, no. 3, pp. 652·686, 1985.
[90] G. A. Stephen, String Searching Algorithms . World Scienti·c Press, 1994.
[91] M. Summer·eld, Programming in Python 3: A Complete Introduction to the Python Language . Addison-Wesley Professional, 2nd ed., 2009.
[92] R. Tamassia and G. Liotta, ·Graph drawing,· in Handbook of Discrete and Computational Geometry (J. E. Goodman and J. O·Rourke, eds.), ch. 52, pp. 1163·1186, CRC Press LLC, 2nd ed., 2004.
[93] R. Tarjan and U. Vishkin, ·An ef·cient parallel biconnectivity algorithm,· SIAM J. Comput. , vol. 14, pp. 862·874, 1985.
[94] R. E. Tarjan, ·Depth ·rst search and linear graph algorithms,· SIAM J. Comput. , vol. 1, no. 2, pp. 146·160, 1972.
[95] R. E. Tarjan, Data Structures and Network Algorithms ,v o l .4 4o f CBMS-NSF Regional Conference Series in Applied Mathematics . Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.
[96] A. B. Tucker, Jr., The Computer Science and Engineering Handbook . CRC Press, 1997.
[97] J. D. Ullman, Principles of Database Systems . Potomac, MD: Computer Science Press, 1983.
[98] J. van Leeuwen, ·Graph algorithms,· in Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 525·632, Amsterdam: Elsevier, 1990.
[99] J. S. Vitter, ·Ef·cient memory access in large-scale computation,· in Proc. 8th Sympos. Theoret. Aspects Comput. Sci. , Lecture Notes Comput. Sci., Springer-Verlag, 1991.
[100] J. S. Vitter and W. C. Chen, Design and Analysis of Coalesced Hashing .N e wY o r k : Oxford University Press, 1987.
[101] J. S. Vitter and P. Flajolet, ·Average-case analysis of algorithms and data structures,· inAlgorithms and Complexity (J. van Leeuwen, ed.), vol. A of Handbook of Theoretical Computer Science , pp. 431·524, Amsterdam: Elsevier, 1990.
[102] S. Warshall, ·A theorem on boolean matrices,· Journal of the ACM , vol. 9, no. 1, pp. 11·12, 1962.
[103] J. W. J. Williams, ·Algorithm 232: Heapsort,· Communications of the ACM ,v o l .7 , no. 6, pp. 347·348, 1964.
[104] D. Wood, Data Structures, Algorithms, and Performance . Reading, MA: Addison-Wesley, 1993.
[105] J. Zelle, Python Programming: An Introduciton to Computer Science . Franklin, Beedle & Associates Inc., 2nd ed., 2010.

def
 or
(self,o t h e r) : # 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
forein self :
result.add(e)
foreinother:
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 resultingset. Since the MutableSet class is
must belong to a concrete subclass. When computing the union of two such con-
crete 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 identi·ed as self, and the subsequent parentheses in expression type(self)()
call the default constructor for that class.
In terms of ef·ciency, we analyze such set operations while letting ndenote
the size of Sand mdenote the size of set Tfor an operation such as S|T.I f
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 Frag-
ment 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 andreturn the existing set, after
in-place version of the union has expected running time O(m)where mis the size
of the second set, because we only have to loop through that second set.
def
ior
(self,o t h e r) : # supports syntax S |=T
···Modify this set to be the union of itself an another set.···
foreinother:
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.

14.2.3 Adjacency Map Structure

In the adjacency list structure, we assume that the secondary incidence collections are implemented as unordered linked lists. Such a collection $I(v)$uses space proportional to $O(\deg(v))$, allows an edge to be added or removed in $O(1)$time, and allows an iteration of all edges incident to vertex $v$in$O(\deg(v))$time. However, the best implementation of get
edge(u,v) requires $O(\min(\deg(u),\deg(v)))$time,
because we must search through either $I(u)$or$I(v)$.

We can improve the performance by using a hash-based map to implement $I(v)$ for each vertex v. Speci·cally, we let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value. We call such a graph representation an adjacency map . (See Figure 14.6.) The space usage for an adjacency map remains $O(n+m)$, because $I(v)$uses $O(\deg(v))$space for each vertex v, as with the adjacency list.

The advantage of the adjacency map, relative to an adjacency list, is that the
get
edge(u,v) method can be implemented in expected $O(1)$time by searching for vertex $u$as a key in $I(v)$, or vice versa. This provides a likely improvement over the adjacency list, while retaining the worst-case bound of $O(\min(\deg(u),\deg(v)))$.

In comparing the performance of adjacency map to other representations (see Table 14.1), we ·nd that it essentially achieves optimal running times for all methods, making it an excellent all-purpose choice as a graph representation.

he g
vu
wzf gh
w
huuw v
g e
f ew
vuz
fv
w
zV
(a) (b)

Figure 14.6: (a) An undirected graph G; (b) a schematic representation of the adjacency map structure for G. Each vertex maintains a secondary map in which neighboring vertices serve as keys, with the connecting edges as associated values. Although not diagrammed as such, we presume that there is a unique Edge instance for each edge of the graph, and that it maintains references to its endpoint vertices.