10.4 Skip Lists
An interesting data structure for realizing the sorted map ADT is the skip list .I n
Section 10.3.1, we saw that a sorted array will allow O(logn)-time searches via the
binary search algorithm. Unfortunately, update operations on a sorted array have
O(n)worst-case running time because of the need to shift elements. In Chapter 7
we demonstrated that linked lists support very ef·cient update operations, as long
as the position within the list is identi·ed. Unfortunately, we cannot perform fastsearches on a standard linked list
an ef·cient means for direct accessing an element of a sequence by index.
Skip lists provide a clever compromise to ef·ciently support search and update
operations. A skip list Sfor a map Mconsists of a series of lists {S
0,S1,..., Sh}.
Each list Sistores a subset of the items of Msorted by increasing keys, plus items
with two sentinel keys denoted ·· and +·,w h e r e ·· is smaller than every
possible key that can be inserted in Mand +·is larger than every possible key
that can be inserted in M. In addition, the lists in Ssatisfy the following:
·List S0contains every item of the map M(plus sentinels ·· and +·).
·For i=1,..., h·1, list Sicontains (in addition to ·· and +·) a randomly
generated subset of the items in list Si·1.
·List Shcontains only ·· and +·.
An example of a skip list is shown in Figure 10.10. It is customary to visualize a
skip list Swith list S0at the bottom and lists S1,..., Shabove it. Also, we refer to h
as the height of skip list S.
Intuitively, the lists are set up so that Si+1contains more or less alternate items
ofSi. As we shall see in the details of the insertion method, the items in Si+1are
chosen at random from the items in Siby picking each item from Sito also be in
Si+1with probability 1 /2. That is, in essence, we ··ip a coin· for each item in Si
3125
25
-·-·-·-·
-·
-·1717
17
17 12S5
S4
S3
S2
S1
S055
55
55
55 12 17 20 25 31 38 39 44 50 +·+·+·+·+·+·
44 383125
Figure 10.10: Example of a skip list storing 10 items. For simplicity, we show only
the items· keys, not their associated values.

and place that item in Si+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 operation
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.

Removal in a Skip List
Like the search and insertion algorithms, the removal algorithm for a skip list is
quite simple. In fact, it is even easier than the insertion algorithm. That is, to per-
form the map operation del M[k] we begin by executing method SkipSearch (k).
If the position 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 posi-
tion p. While removing levels of the tower, we reestablish links between the hor-
izontal 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 perfor-
mance 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.

Algorithm SkipSearch(k) :
Input: A search key k
Output: Position pin the bottom list S0with 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 S0. After inserting
the new item at the bottom level, we use randomization to decide the height of the
tower for the new item. We ··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 li
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.

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.

Maintaining the Topmost Level
A skip list Smust maintain a reference to the start position (the topmost, left po-
sition in S) as an instance variable, and must have a policy for any insertion that
wishes to continue inserting a new entry past the top level of S.T h e r e a r e t w o
possible courses of action we can take, both of which have their merits.
One possibility is to restrict the top level, h, to be kept at some ·xed value that
is a function of n, the number of entries currently in the map (from the analysis we
will see that h=max{10,2·logn·}is a reasonable choice, and picking h=3·logn·
is even safer). Implementing this choice means that we must modify the insertion
algorithm to stop inserting a new position once we reach the topmost level (unless
·logn·<·log (n+1)·, in which case we can now go at least one more level, since
the bound on the height is increasing).
The other possibility is to let an insertion continue inserting a new position as
long as heads keeps getting returned from the random number generator. This isthe approach taken by algorithr
in the analysis of skip lists, the probability that an insertion will go to a level that ismore than O(logn)is very low, s
Either choice will still result in the expected O(logn)time to perform search,
insertion, and removal, however, which we show in the next section.
10.4.2 Probabilistic Analysis of Skip Lists ·
As we have shown above, skip lists provide a simple implementation of a sortedmap. In terms of worst-case perf
ability of having a fair coin repeatedly come up heads forever is 0). Moreover, we
cannot in·nitely add positions to a list without eventually running out of memory.In any case, if we terminate posit
case running time for performing the
getitem
,
setitem
,a n d
delitem
map operations in a skip list Swith nentries and height hisO(n+h). This worst-
case performance occurs when the tower of every entry reaches level h·1, where
his the height of S. However, this event has very low probability. Judging from
this worst case, we might conclude that the skip-list structure is strictly inferior tothe other map implementations c
be a fair analysis, for this worst-case behavior is a gross overestimate.

10.4.1 Search and Update Operations in a Skip List
The skip-list structure affords simple map search and update algorithms. In fact,
all of the skip-list search and update algorithms are based on an elegant SkipSearch
method that takes a key kand ·nds the position pof the item in list S0that has the
largest key less than or equal to k(which is possibly ··).
Searching in a Skip List
Suppose we are given a search key k. We begin the SkipSearch method by setting
a position variable pto the topmost, left position in the skip list S, called the start
position ofS. That is, the start position is the position of Shstoring the special
entry with key ··. We then perform the following steps (see Figure 10.11), where
key (p)denotes the key of the item at position p:
1. If S.below (p)isNone , then the search terminates·we are at the bottom and
have located the item in Swith the largest key less than or equal to the search
key k. Otherwise, we drop down to the next lower level in the present tower
by setting p=S.below (p).
2. Starting at position p, we move pforward until it is at the rightmost position
on the present level such that key (p)·k. We call this the scan forward step.
Note that such a position always exists, since each level contains the keys+·and··. It may be that premains wher
such a forward scan for this level.
3. Return to step 1.
55S1S2S3S4S5
+·
+·+·
+·
+·+·
-·-·
-· 1212 -·17
17 25
25 20 17 31 38 39-·
-· 1717 25
25 31
31 38 44
44 50555555
S0
Figure 10.11: Example of a search in a skip list. The positions examined when
searching for key 50 are highlighted.
We give a pseudo-code description of the skip-list search algorithm, SkipSearch ,
in Code Fragment 10.12. Given this method, the map operation M[k]is performed
by computing p=SkipSearch (k)and testing whether or not key (p)= k. If these
two keys are equal, we return the associated value; otherwise, we raise a KeyError .

Algorithm SkipInsert(k,v) :
Input: Keykand value v
Output: Topmost position of the item inserted in the skip list
p=SkipSearch (k)
q=None {qwill represent top node in new item·s tower }
i=·1
repeat
i=i+1
ifi·hthen
h=h+1 {add a new level to the skip list }
t=next(s)
s=insertAfterAbove (None,s,(··,None)) {grow leftmost tower }
insertAfterAbove (s,t,(+·,None)) {grow rightmost tower }
while above (p)isNone do
p=prev (p) {scan backward }
p=ab ove (p) {jump up to higher level }
q=insertAfterAbove (p,q,(k,v)){increase height of new item·s tower }
untilcoinFlip ()==tails
n=n+1
return q
Code Fragment 10.13: Insertion in a skip list. Method coinFlip ()returns ·heads· or
·tails·, each with probability 1 /2. Instance variables n,h,a n d shold the number
of entries, the height, and the start node of the skip list.
55 S1S2S3S4S5
+·
+·+·
+·
+·+·
-·-·
-· 1212 -·17
17 25
25 20 17 31-·
-· 1717 25
25 31
31 38 44
44424242
5555
55 38 39 42 50 S0
Figure 10.12: Insertion of an entry with key 42 into the skip list of Figure 10.10. We
assume that the random ·coin ·ips· for the new entry came up heads three times in a
row, followed by tails. The positions visited are highlighted. The positions insertedto hold the new entry are draw

C-15.11 Describe an external-memory data structure to implement the queue ADT
so that the total number of disk transfers needed to process a sequence of
kenqueue anddequeue operations is O(k/B).
C-15.12 Describe an external-memory version of the PositionalList ADT (Sec-
tion 7.4), with block size B, such that an iteration of a list of length nis
completed using O(n/B)transfers in the worst case, and all other methods
of the ADT require only O(1)transfers.
C-15.13 Change the rules that de·ne red-black trees so that each red-black tree T
has a corresponding (4,8)tree, and vice versa.
C-15.14 Describe a modi·ed version of the B-tree insertion algorithm so that eachtime we create an over·ow bec
keys among all of w·s siblings, so that each sibling holds roughly the same
number of keys (possibly cascading the split up to the parent of w). What
is the minimum fraction of each block that will always be ·lled using this
scheme?
C-15.15 Another possible external-memory map implementation is to use a skip
list, but to collect consecutive groups of O(B)nodes, in individual blocks,
on any level in the skip list. In particular, we de·ne an order-d B-skip
listto be such a representation of a skip list structure, where each block
contains at least ·d/2·list nodes and at most dlist nodes. Let us also
choose din this case to be the maximum number of list nodes from a level
of a skip list that can ·t into one block. Describe how we should modify
the skip-list insertion and removal algorithms for a B-skip list so that the
expected height of the structure is O(logn/logB).
C-15.16 Describe how to use a B-tree to implement the partition (union-·nd) ADT
(from Section 14.7.3) so that the union and·nd operations
each use at
most O(logn/logB)disk transfers.
C-15.17 Suppose we are given a sequence Sofnelements with integer keys such
that some elements in Sare colored ·blue· and some elements in Sare
colored ·red.· In addition, say that a red element epairs with a blue
element fif they have the same key value. Describe an ef·cient external-
memory algorithm for ·nding all the red-blue pairs in S.H o wm a n yd i s k
transfers does your algorithm perform?
C-15.18 Consider the page caching problem where the memory cache can hold m
pages, and we are given a sequence Pofnrequests taken from a pool
ofm+1 possible pages. Describe the optimal strategy for the of·ine
algorithm and show that it causes at most m+n/mpage misses in total,
starting from an empty cache.
C-15.19 Describe an ef·cient external-memory algorithm that determines whetheran array of nintegers contains a

So we have yet to bound the number of scan-forward steps we make. Let nibe
the number of keys examined while scanning forward at level i. Observe that, after
the key at the starting position, each additional key examined in a scan-forward at
level icannot also belong to level i+1. If any of these keys were on the previous
level, we would have encountered them in the previous scan-forward step. Thus,the probability that any key is co
iis 1/2. Therefore, the expected value of
niis exactly equal to the expected number of times we must ·ip a fair coin before
it comes up heads. This expected value is 2. Hence, the expected amount of timespent scanning forward at any
probability, a search in Stakes expected time O(logn). By a similar analysis, we
can show that the expected running time of an insertion or a removal is O(logn).

Space Usage in a Skip List

Finally, let us turn to the space requirement of a skip list Swith nentries. As we
observed above, the expected number of positions at level iisn/2i, which means
that the expected total number of positions in Sis
h
·
i=0n
2i=nh
·
i=01
2i.
Using Proposition 3.5 on geometric summations, we have
h
·
i=01
2i=/parenleftbig1
2/parenrightbigh+1·1
1
2·1=2·/parenleftbigg
1·1
2h+1/parenrightbigg
<2f o r a l l h·0.
Hence, the expected space requirement of SisO(n).
Table 10.4 summarizes the performance of a sorted map realized by a skip list.

Operation	Running Time
len(M)	O(1)
kinM	O(logn)expected
M[k] = v	O(logn)expected
del M[k]	O(logn)expected
M.·nd min() ,M.·nd max()	O(1)
M.·nd	

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-10.39 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 usir
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.

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.

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.

Chapter

10Maps, Hash Tables, and Skip Lists

Contents

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.

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	

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 sta
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.

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 ir
While that method accomplishes the goal in a general fashion, its ef·-ciency is less than ideal. In particular, wher
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.)

15.3 External Searching and B-Trees
Consider the problem of maintaining a large collection of items that does not ·t in
main memory, such as a typical database. In this context, we refer to the secondary-memory blocks as disk bloc
secondary memory and primary memory as a disk transfer . Recalling the great
time difference that exists between main memory accesses and disk accesses, themain goal of maintaining such
count as the I/O complexity of the algorithm involved.
Some Ine·cient External-Memory Representations
A typical operation we would like to support is the search for a key in a map. If wewere to store nitems unordered
key within the list requires ntransfers in the worst case, since each link hop we
perform on the linked list might access a different block of memory.
We can reduce the number of block transfers by using an array-based sequence.
A sequential search of an array can be performed using only O(n/B)block transfers
because of spatial locality of reference, where Bdenotes the number of elements
that ·t into a block. This is because the block transfer when accessing the ·rst
element of the array actually retrieves the ·rst Belements, and so on with each
successive block. It is worth noting that the bound of O(n/B)transfers is only
achieved when using a compact array representation (see Section 5.2.2). The
standard Python listclass is a referential container, and so even though the sequence
of references are stored in an array, the actual elements that must be examinedduring a search are not generally
transfers in the worst case.
We could alternately store a sequence using a sorted array. In this case, a search
performs O(log
2n)transfers, via binary search, which is a nice improvement. But
we do not get signi·cant bene·t from block transfers because each query duringa binary search is likely in a differ
operations are expensive for a sorted array.
Since these simple implementations are I/O inef·cient, we should consider the
logarithmic-time internal-memory strategies that use balanced binary trees (for ex-
ample, AVL trees or red-black trees) or other search structures with logarithmicaverage-case query and update t
cally, each node accessed for a query or update in one of these structures will be in
a different block. Thus, these methods all require O(log
2n)transfers in the worst
case to perform a query or update operation. But we can do better! We can performmap queries and updates us
Bn)= O(logn/logB)transfers.

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.

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.

```
classCostPerformanceDatabase:
···Maintain a database of maximal (cost,performance) pairs.···

def
 init
(self):
 ···Create an empty database.···
 self.
M = SortedTableMap( ) # or a more e·cient sorted map
defbest(self,c ) :
 ···Return (cost,performance) pair with largest cost not exceeding c.
 Return None if there is no such pair.
 ···
 return self .
M.·nd
 le(c)
defadd(self,c ,p ) :
 ···Add new entry with cost c and performance p.···
 # determine if (c,p) is dominated by an existing pair
 other = self.
M.·nd
 le(c) # other is at least as cheap as c
 ifotheris not None and other[1] >=p :# if its performance is as good,
 return # (c,p) is dominated, so ignore
 self.
M[c] = p # else, add (c,p) to database
 # and now remove any pairs that are dominated by (c,p)
 other = self.
M.·nd
 gt(c) # other more expensive than c
 while otheris not None and other[1] <=p :
 del self .
M[other[0]]
 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.

[63] D. E. Knuth, ·Big omicron and big omega and big theta,· in SIGACT News ,v o l .8 , pp. 18·24, 1976.
[64] D. E. Knuth, Fundamental Algorithms , vol. 1 of The Art of Computer Programming . Reading, MA: Addison-Wesley, 3rd ed., 1997.
[65] D. E. Knuth, Sorting and Searching ,v o l .3o f The Art of Computer Programming . Reading, MA: Addison-Wesley, 2nd ed., 1998.
[66] D. E. Knuth, J. H. Morris, Jr., and V . R. Pratt, ·Fast pattern matching in strings,· SIAM J. Comput. , vol. 6, no. 1, pp. 323·350, 1977.
[67] J. B. Kruskal, Jr., ·On the shortest spanning subtree of a graph and the traveling salesman problem,· Proc. Amer . Math. Soc. , vol. 7, pp. 48·50, 1956.
[68] R. Lesuisse, ·Some lessons drawn from the history of the binary search algorithm,· The Computer Journal , vol. 26, pp. 154·163, 1983.
[69] N. G. Leveson and C. S. Turner, ·An investigation of the Therac-25 accidents,· IEEE Computer , vol. 26, no. 7, pp. 18·41, 1993.
[70] A. Levitin, ·Do we teach the right algorithm design techniques?,· in 30th ACM SIGCSE Symp. on Computer Science Education , pp. 179·183, 1999.
[71] B. Liskov and J. Guttag, Abstraction and Speci·cation in Program Development . Cambridge, MA/New York: The MIT Press/McGraw-Hill, 1986.
[72] M. Lutz, Programming Python . O·Reilly Media, 4th ed., 2011.
[73] E. M. McCreight, ·A space-economical suf·x tree construction algorithm,· Journal of Algorithms , vol. 23, no. 2, pp. 262·272, 1976.
[74] C. J. H. McDiarmid and B. A. Reed, ·Building heaps fast,· Journal of Algorithms , vol. 10, no. 3, pp. 352·365, 1989.
[75] N. Megiddo, ·Linear programming in linear time when the dimension is ·xed,· J. ACM , vol. 31, pp. 114·127, 1984.
[76] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching ,v o l .1 ofEATCS Monographs on Theoretical Computer Science . Heidelberg, Germany: Springer-Verlag, 1984.
[77] K. Mehlhorn, Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness , vol. 2 of EATCS Monographs on Theoretical Computer Science .H e i - delberg, Germany: Springer-Verlag, 1984.
[78] K. Mehlhorn and A. Tsakalidis, ·Data structures,· in Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 301· 341, Amsterdam: Elsevier, 1990.
[79] D. R. Morrison, ·PATRICIA·practical algorithm to retrieve information coded in alphanumeric,· Journal of the ACM , vol. 15, no. 4, pp. 514·534, 1968.
[80] R. Motwani and P. Raghavan, Randomized Algorithms . New York, NY: Cambridge University Press, 1995.
[81] T. Papadakis, J. I. Munro, and P. V . Poblete, ·Average search and update costs in skip lists,· BIT, vol. 32, pp. 316·332, 1992.
[82] L. Perkovic, Introduction to Computing Using Python: An Application Development Fo c u s . Wiley, 2011.
[83] D. Phillips, Python 3: Object Oriented Programming . Packt Publishing, 2010.
[84] P. V . Poblete, J. I. Munro, and T. Papadaki s, ·The binomial transform and its application to the analysis of skip lists,· in Proceedings of the European Symposium on Algorithms (ESA) , pp. 554·569, 1995.

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

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.

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

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.

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

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.

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.

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.

10.2.4 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

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(

26def
bucket
 getitem(self,j ,k) :
27 found, s = self.
·nd
slot(j, k)
28 if not found:
29 raiseKeyError(
 Key Error:
 +r e p r (k)) # no match found
30 return self .
table[s].
 value
31
32def
bucket
 setitem(self,j ,k ,v) :
33 found, s = self.
·nd
slot(j, k)
34 if not found:
35 self.
table[s] = self.
Item(k,v) # insert new item
36 self.
n+ =1 # size has increased
37 else:
38 self.
table[s].
 value = v # overwrite existing
3940def
bucket
 delitem(self,j ,k) :
41 found, s = self.
·nd
slot(j, k)
42 if not found:
43 raiseKeyError(
 Key Error:
 +r e p r (k)) # no match found
44 self.
table[s] = ProbeHashMap.
 AVAIL # mark as vacated
4546def
iter
(self):
47 forjinrange(len(self.
table)): # scan entire table
48 if not self .

```
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 b
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)

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 typically 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 values. 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
 .

10.5.2 Python·s MutableSet Abstract Base Class
To aid in the creation of user-de·ned set classes, Python·s collections module pro-
vides a MutableSet abstract base class (just as it provides the MutableMapping ab-
stract 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).

10.6 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 key
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.

C-11.56 The standard splaying step requires two passes, one downward pass to
·nd the node xto splay, followed by an upward pass to splay the node
x. Describe a method for splaying and searching for xin one downward
pass. Each substep now requires that you consider the next two nodes
in the path down to x, with a possible zig substep performed at the end.
Describe how to perform the zig-zig, zig-zag, and zig steps.
C-11.57 Consider a variation of splay trees, called half-splay trees , where splaying
a node at depth dstops as soon as the node reaches depth ·d/2·. Perform
an amortized analysis of half-splay trees.
C-11.58 Describe a sequence of accesses to an n-node splay tree T,w h e r e nis odd,
that results in Tconsisting of a single chain of nodes such that the path
down Talternates between left children and right children.
C-11.59 As a positional structure, our TreeMap implementation has a subtle ·aw.
A position instance passociated with an key-value pair (k,v)should re-
main valid as long as that item remains in the map. In particular, thatposition should be unaffected by calls to ins
may fail to provide such a guarantee, in particular because of our rule for
using the inorder predecessor of a key as a replacement when deleting akey that is located in a node with two ch
C-11.60 How might the TreeMap implementation be changed to avoid the ·aw
described in the previous problem?
Projects
P-11.61 Perform an experimental study to compare the speed of our AVL tree,
splay tree, and red-black tree implementations for various sequences of
operations.
P-11.62 Redo the previous exercise, including an implementation of skip lists.(See Exercise P-10.53.)
P-11.63 Implement the Map ADT using a (2,4)tree. (See Section 10.1.1.)
P-11.64 Redo the previous exercise, including all methods of the Sorted Map ADT.
(See Section 10.3.)
P-11.65 Redo Exercise P-11.63 providing positional support, as we did for bi-
nary search trees (Section 11.1.1), so as to include methods ·rst() ,last() ,
before(p) ,after(p) ,a n d·nd
position(k) . Each item should have a dis-
tinct position in this abstraction, even though several items may be storedat a single node of a tree.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software . Reading, MA: Addison-Wesley, 1995.
[42] A. Goldberg and D. Robson, Smalltalk-80: The Language . Reading, MA: Addison-Wesley, 1989.
[43] M. H. Goldwasser and D. Letscher, Object-Oriented Programming in Python . Upper Saddle River, NJ: Prentice Hall, 2008.
[44] G. H. Gonnet and R. Baeza-Yates, Handbook of Algorithms and Data Structures in Pascal and C . Reading, MA: Addison-Wesley, 1991.
[45] G. H. Gonnet and J. I. Munro, ·Heaps on heaps,· SIAM J. Comput. , vol. 15, no. 4, pp. 964·971, 1986.
[46] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, ·External-memory computational geometry,· in Proc. 34th Annu. IEEE Sympos. F ound. Comput. Sci. , pp. 714·723, 1993.
[47] R. L. Graham and P. Hell, ·On the history of the minimum spanning tree problem,· Annals of the History of Computing , vol. 7, no. 1, pp. 43·57, 1985.
[48] L. J. Guibas and R. Sedgewick, ·A dichromatic framework for balanced trees,· in Proc. 19th Annu. IEEE Sympos. F ound. Comput. Sci. , Lecture Notes Comput. Sci., pp. 8·21, Springer-Verlag, 1978.
[49] Y . Gurevich, ·What does O(n)mean?,· SIGACT News , vol. 17, no. 4, pp. 61·63, 1986.
[50] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach . San Francisco: Morgan Kaufmann, 2nd ed., 1996.
[51] C. A. R. Hoare, ·Quicksort,· The Computer Journal , vol. 5, pp. 10·15, 1962.
[52] J. E. Hopcroft and R. E. Tarjan, ·Ef·cient algorithms for graph manipulation,· Communications of the ACM , vol. 16, no. 6, pp. 372·378, 1973.
[53] B.-C. Huang and M. Langston, ·Practical in-place merging,· Communications of the ACM , vol. 31, no. 3, pp. 348·352, 1988.
[54] J. J´ aJ´ a, An Introduction to Parallel Algorithms . Reading, MA: Addison-Wesley, 1992.
[55] V . Jarn ´ ·k, ·O jistem problemu minimalnim,· Praca Moravske Prirodovedecke Spolecnosti , vol. 6, pp. 57·63, 1930. (in Czech).
[56] R. Jones and R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management . John Wiley and Sons, 1996.
[57] D. R. Karger, P. Klein, and R. E. Tarjan, ·A randomized linear-time algorithm to ·nd minimum spanning trees,· Journal of the ACM , vol. 42, pp. 321·328, 1995.
[58] R. M. Karp and V . Ramachandran, ·Parallel algorithms for shared memory machines,· in Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), pp. 869·941, Amsterdam: Elsevier/The MIT Press, 1990.
[59] P. Kirschenhofer and H. Prodinger, ·The pat h length of random skip lists,· Acta Informatica , vol. 31, pp. 775·792, 1994.
[60] J. Kleinberg and ´E. Tardos, Algorithm Design . Reading, MA: Addison-Wesley, 2006.
[61] A. Klink and J. W¨ alde, ·Ef·cient denial of service attacks on web application platforms.· 2011.
[62] D. E. Knuth, Sorting and Searching ,v o l .3o f The Art of Computer Programming . R
eading, MA: Addison-Wesley, 1973.