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 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.

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.

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 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.

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 algorithm
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