9.4 Skip Lists

An interesting data structure for efficiently realizing the ordered map ADT is the ***skip list***. This data structure makes random choices in arranging the entries in such a way that search and update times are $O(\log n)$ ***on average***, where n is the number of entries in the dictionary. Interestingly, the notion of average time complexity used here does not depend on the probability distribution of the keys in the input. Instead, it depends on the use of a random-number generator in the implementation of the insertions to help decide where to place the new entry. The running time is averaged over all possible outcomes of the random numbers used when inserting entries. Interestingly, Java includes an implementation of the ordered map ADT using a skip list, in the ConcurrentSkipListMap class, which guarantees $O(\log n)$ expected time performance for the get, put, and remove methods and their variants.

Because they are used extensively in computer games, cryptography, and computer simulations, methods that generate numbers that can be viewed as random numbers are built into most modern computers. Some methods, called ***pseudo-random number generators*** (Section 3.1.3), generate random-like numbers, starting with an initial ***seed***. Other methods use hardware devices to extract "true" random numbers from nature. In any case, we will assume that our computer has access to numbers that are sufficiently random for our analysis.

The main advantage of using ***randomization*** in data structure and algorithm design is that the structures and methods that result are usually simple and efficient. We can devise a simple randomized data structure, called the skip list, which has the same logarithmic time bounds for searching as is achieved by the binary searching algorithm. Nevertheless, the bounds are ***expected*** for the skip list, while they are ***worst-case*** bounds for binary searching in a look-up table. On the other hand, skip lists are much faster than look-up tables for map updates.

A ***skip list*** S for a map M consists of a series of lists $\{S_0, S_1, \ldots, S_h\}$. Each list S_i stores a subset of the entries of M sorted by increasing keys plus entries with two special keys, denoted $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in M and $+\infty$ is larger than every possible key that can be inserted in M. In addition, the lists in S satisfy the following:

- List S_0 contains every entry of the map M (plus the special entries with keys $-\infty$ and $+\infty$).
- For $i = 1, \ldots, h-1$, list S_i contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the entries in list S_{i-1}.
- List S_h contains only $-\infty$ and $+\infty$.

An example of a skip list is shown in Figure 9.9. It is customary to visualize a skip list S with list S_0 at the bottom and lists $S_1, \ldots, S_h$ above it. Also, we refer to h as the ***height*** of skip list S.

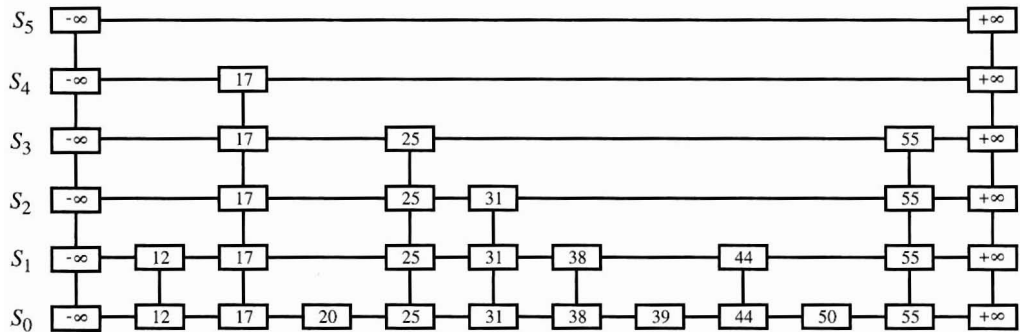

Figure 9.9: Example of a skip list storing 10 entries. For simplicity, we show only the keys of the entries.

Intuitively, the lists are set up so that S_{i+1} contains more or less every other entry in S_i. As we shall see in the details of the insertion method, the entries in S_{i+1} are chosen at random from the entries in S_i by picking each entry from S_i to also be in S_{i+1} with probability $1/2$. That is, in essence, we "flip a coin" for each entry in S_i and place that entry in S_{i+1} if the coin comes up "heads." Thus, we expect S_1 to have about $n/2$ entries, S_2 to have about $n/4$ entries, and, in general, S_i to have about $n/2^i$ entries. In other words, we expect the height h of S to be about $\log n$. The halving of the number of entries from one list to the next is not enforced as an explicit property of skip lists, however. Instead, randomization is used.

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 vertically into ***towers***. Each level is a list S_i and each tower contains positions storing the same entry across consecutive lists. The positions in a skip list can be traversed using the following operations:

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

prev(p): Return the position preceding p on the same level.

below(p): Return the position below p in the same tower.

above(p): Return the position above p in the same tower.

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

9.4.1 Search and Update Operations in a Skip List

The skip list structure allows for simple map search and update algorithms. In fact, all of the skip list search and update algorithms are based on an elegant SkipSearch method that takes a key k and finds the position p of the entry e in list S_0 such that e has the largest key (which is possibly $-\infty$) less than or equal to k.

Searching in a Skip List

Suppose we are given a search key k. We begin the SkipSearch method by setting a position variable p to the top-most, left position in the skip list S, called the ***start position*** of S. That is, the start position is the position of S_h storing the special entry with key $-\infty$. We then perform the following steps (see Figure 9.10), where key(p) denotes the key of the entry at position p:

1. If S.below(p) is null, then the search terminates—we are ***at the bottom*** and have located the largest entry in S with 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 $p \leftarrow S$.below(p).
2. Starting at position p, we move p forward until it is at the right-most position on the present level such that key$(p) \leq k$. We call this the ***scan forward*** step. Note that such a position always exists, since each level contains the keys $+\infty$ and $-\infty$. In fact, after we perform the scan forward for this level, p may remain where it started. In any case, we then repeat the previous step.

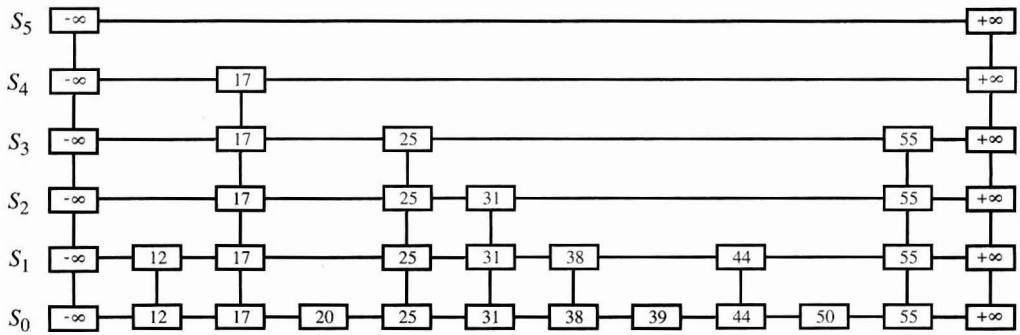

Figure 9.10: Example of a search in a skip list. The positions visited when searching for key 50 are highlighted in blue.

We give a pseudo-code description of the skip-list search algorithm, SkipSearch, in Code Fragment 9.10. Given this method, it is now easy to implement the operation get(k)—we simply perform $p \leftarrow$ SkipSearch(k) and test whether or not key$(p) = k$. If these two keys are equal, we return p; otherwise, we return **null**.

Algorithm SkipSearch(k):
Input: A search key k
Output: Position p in the bottom list S_0 such that the entry at p has the largest key less than or equal to k

```
p ← s
while below(p) ≠ null do
  p ← below(p)          {drop down}
  while k ≥ key(next(p)) do
    p ← next(p)         {scan forward}
return p.
```

Code Fragment 9.10: Search in a skip list S. Variable s holds the start position of S.

As it turns out, the expected running time of algorithm SkipSearch on a skip list with n entries is $O(\log n)$. We postpone the justification of this fact, however, until after we discuss the implementation of the update methods for skip lists.

Insertion in a Skip List

The insertion algorithm for skip lists uses randomization to decide the height of the tower for the new entry. We begin the insertion of a new entry (k,v) by performing a SkipSearch(k) operation. This gives us the position p of the bottom-level entry with the largest key less than or equal to k (note that p may hold the special entry with key $-\infty$). We then insert (k,v) immediately after position p. After inserting the new entry at the bottom level, we "flip" a coin. If the flip comes up tails, then we stop here. Else (the flip comes up heads), we backtrack to the previous (next higher) level and insert (k,v) in this level at the appropriate position. We again flip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new entry (k,v) in lists until we finally get a flip that comes up tails. We link together all the references to the new entry (k,v) created in this process to create the tower for the new entry. A coin flip can be simulated with Java's built-in pseudo-random number generator java.util.Random by calling nextInt(2), which returns 0 of 1, each with probability $1/2$.

We give the insertion algorithm for a skip list S in Code Fragment 9.11 and we illustrate it in Figure 9.11. The algorithm uses method insertAfterAbove($p,q,(k,v)$) that inserts a position storing the entry (k,v) after position p (on the same level as p) and above position q, returning the position r of the new entry (and setting internal references so that next, prev, above, and below methods will work correctly for p, q, and r). The expected running time of the insertion algorithm on a skip list with n entries is $O(\log n)$, which we show in Section 9.4.2.

Algorithm SkipInsert(k,v):
Input: Key k and value v
Output: Topmost position of the entry inserted in the skip list
$p \leftarrow$ SkipSearch(k)
$q \leftarrow$ **null**
$e \leftarrow (k,v)$
$i \leftarrow -1$
repeat
  $i \leftarrow i+1$
  if $i \geq h$ **then**
    $h \leftarrow h+1$ {add a new level to the skip list}
    $t \leftarrow$ next(s)
    $s \leftarrow$ insertAfterAbove(**null**, s, $(-\infty$, **null**))
    insertAfterAbove$(s,t,(+\infty$, **null**))
  while above$(p) =$ **null do**
    $p \leftarrow$ prev(p) {scan backward}
  $p \leftarrow$ above(p) {jump up to higher level}
  $q \leftarrow$ insertAfterAbove(p,q,e) {add a position to the tower of the new entry}
until coinFlip() = tails
$n \leftarrow n+1$
return q

Code Fragment 9.11: Insertion in a skip list. Method coinFlip() returns "heads" or "tails", each with probability $1/2$. Variables n, h, and s hold the number of entries, the height, and the start node of the skip list.

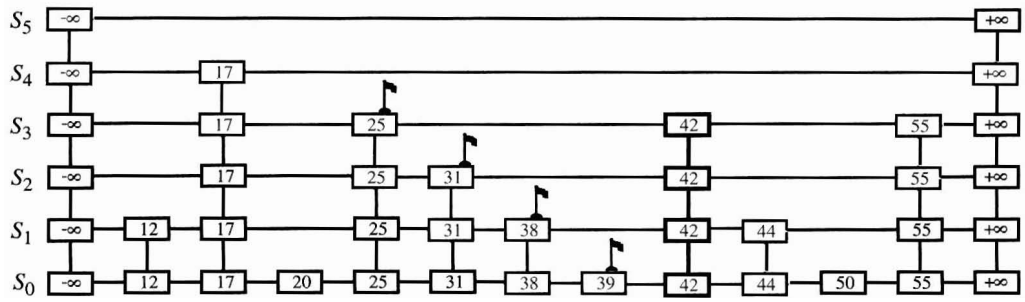

Figure 9.11: Insertion of an entry with key 42 into the skip list of Figure 9.9. We assume that the random "coin flips" for the new entry came up heads three times in a row, followed by tails. The positions visited are highlighted in blue. The positions inserted to hold the new entry are drawn with thick lines, and the positions preceding them are flagged.

Removal in a Skip List

Like the search and insertion algorithms, the removal algorithm for a skip list is quite simple. In fact, it is even easier than the insertion algorithm. That is, to perform a remove(k) operation, we begin by executing method SkipSearch(k). If the position p stores an entry with key different from k, we return **null**. Otherwise, we remove p and all the positions above p, which are easily accessed by using above operations to climb up the tower of this entry in S starting at position p. The removal algorithm is illustrated in Figure 9.12 and a detailed description of it is left as an exercise (R-9.19). As we show in the next subsection, operation remove in a skip list with n entries has $O(\log n)$ expected running time.

Before we give this analysis, however, there are some minor improvements to the skip list data structure we would like to discuss. First, we don't actually need to store references to entries at the levels of the skip list above the bottom level, because all that is needed at these levels are references to keys. Second, we don't actually need the above method. In fact, we don't need the prev method either. We can perform entry insertion and removal in strictly a top-down, scan-forward fashion, thus saving space for "up" and "prev" references. We explore the details of this optimization in Exercise C-9.11. Neither of these optimizations improve the asymptotic performance of skip lists by more than a constant factor, but these improvements can, nevertheless, be meaningful in practice. In fact, experimental evidence suggests that optimized skip lists are faster in practice than AVL trees and other balanced search trees, which are discussed in Chapter 10.

The expected running time of the removal algorithm is $O(\log n)$, which we show in Section 9.4.2.

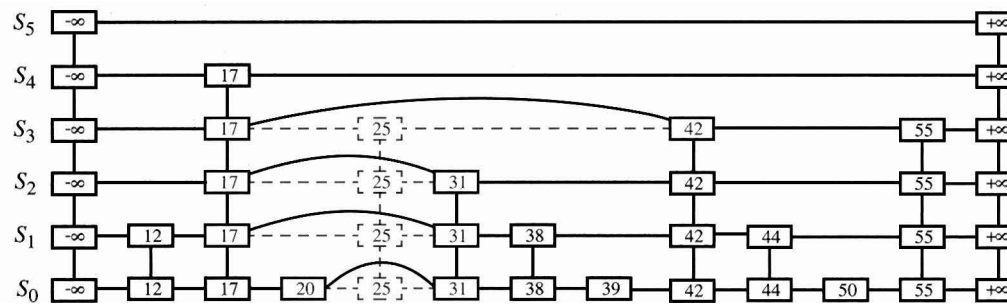

Figure 9.12: Removal of the entry with key 25 from the skip list of Figure 9.11. The positions visited after the search for the position of S_0 holding the entry are highlighted in blue. The positions removed are drawn with dashed lines.

Maintaining the Top-most Level

A skip-list S must maintain a reference to the start position (the top-most, left position in S) as an instance variable, and must have a policy for any insertion that wishes to continue inserting a new entry past the top level of S. There are two possible courses of action we can take, both of which have their merits.

One possibility is to restrict the top level, h, to be kept at some fixed value that is a function of n, the number of entries currently in the map (from the analysis we will see that $h = \max\{10, 2\lceil \log n \rceil\}$ is a reasonable choice, and picking $h = 3\lceil \log n \rceil$ is even safer). Implementing this choice means that we must modify the insertion algorithm to stop inserting a new position once we reach the top-most level (unless $\lceil \log n \rceil < \lceil \log(n+1) \rceil$, in which case we can now go at least one more level, since the bound on the height is increasing).

The other possibility is to let an insertion continue inserting a new position as long as heads keeps getting returned from the random number generator. This is the approach taken in Algorithm SkipInsert of Code Fragment 9.11. As we show in the analysis of skip lists, the probability that an insertion will go to a level that is more than $O(\log n)$ is very low, so this design choice should also work.

Either choice will still result in the expected $O(\log n)$ time to perform search, insertion, and removal, however, which we show in the next section.

9.4.2 A Probabilistic Analysis of Skip Lists ⋆

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

*We use a star (⋆) to indicate sections containing material more advanced than the material in the rest of the chapter; this material can be considered optional in a first reading.

Bounding the Height of a Skip List

Because the insertion step involves randomization, a more accurate analysis of skip lists involves a bit of probability. At first, this might seem like a major undertaking, for a complete and thorough probabilistic analysis could require deep mathematics (and, indeed, there are several such deep analyses that have appeared in data structures research literature). Fortunately, such an analysis is not necessary to understand the expected asymptotic behavior of skip lists. The informal and intuitive probabilistic analysis we give below uses only basic concepts of probability theory.

Let us begin by determining the expected value of the height h of a skip list S with n entries (assuming that we do not terminate insertions early). The probability that a given entry has a tower of height $i \geq 1$ is equal to the probability of getting i consecutive heads when flipping a coin, that is, this probability is $1/2^i$. Hence, the probability P_i that level i has at least one position is at most

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

for the probability that any one of n different events occurs is at most the sum of the probabilities that each occurs.

The probability that the height h of S is larger than i is equal to the probability that level i has at least one position, that is, it is no more than P_i. This means that h is larger than, say, $3\log n$ with probability at most

$$\begin{aligned} P_{3\log n} &\leq \frac{n}{2^{3\log n}} \\ &= \frac{n}{n^3} = \frac{1}{n^2}. \end{aligned}$$

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

Analyzing Search Time in a Skip List

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

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

Space Usage in a Skip List

Finally, let us turn to the space requirement of a skip list S with n entries. As we observed above, the expected number of positions at level i is $n/2^i$, which means that the expected total number of positions in S is

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

Using Proposition 4.5 on geometric summations, we have

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

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

Table 9.3 summarizes the performance of an ordered map realized by a skip list.

Operation	Time
size, isEmpty	$O(1)$
firstEntry, lastEntry	$O(1)$
keySet, values, entrySet	$O(n)$
get, put, remove	$O(\log n)$ (expected)
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$ (expected)

Table 9.3: Performance of an ordered map implemented with a skip list, as in the class, java.util.concurent.ConcurrentSkipListMap. We use n to denote the number of entries in the dictionary at the time the operation is performed. The expected space requirement is $O(n)$.