9.4.1 Search and Update Operations in a Skip List

The skip list structure allows for simple dictionary 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.

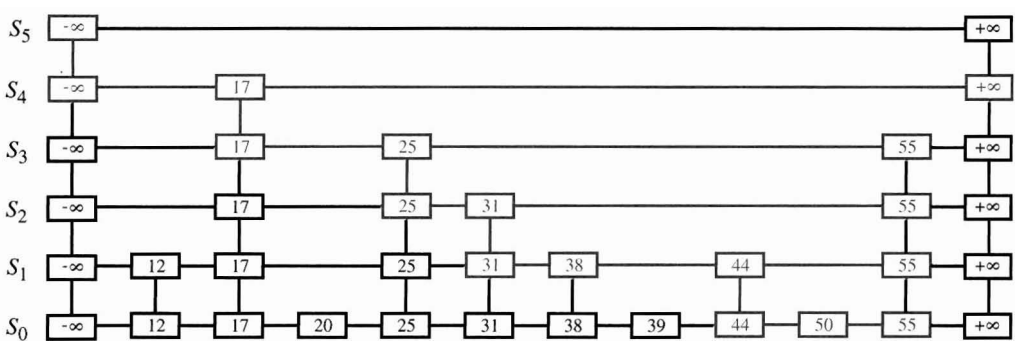

Figure 9.10: Example of a search in a skip list. The positions visited when searching for key 50 are highlighted in blue.

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 find(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: Entry inserted in the skip list
    p ← SkipSearch(k)
    q ← insertAfterAbove(p, null, (k, v))        {we are at the bottom level}
    e ← q.element()
    i ← 0
    while coinFlip() = heads do
        i ← i + 1
        if i ≥ h then
            h ← h + 1                {add a new level to the skip list}
            t ← next(s)
            s ← insertAfterAbove(null, s, (−∞, null))
            insertAfterAbove(s, t, (+∞, null))
        while above(p) = null do
            p ← prev(p)              {scan backward}
        p ← above(p)                 {jump up to higher level}
        q ← insertAfterAbove(p, q, e) {add a position to the tower of the new entry}
    n ← n + 1
    return e
```

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.

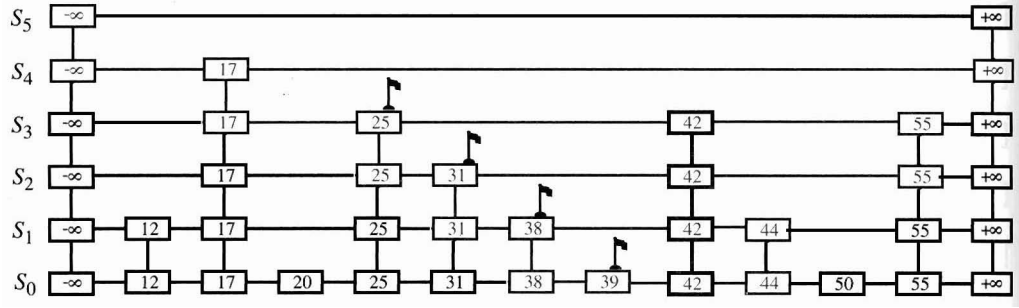

Figure 9.11: Insertion of an entry with key 42 into the skip list of Figure 9.9. We assume that the random "coin flips" for the new entry came up heads three times in a row, followed by tails. The positions visited are highlighted in blue. The positions inserted to hold the new entry are drawn with thick 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-9.16). 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-9.10. 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.

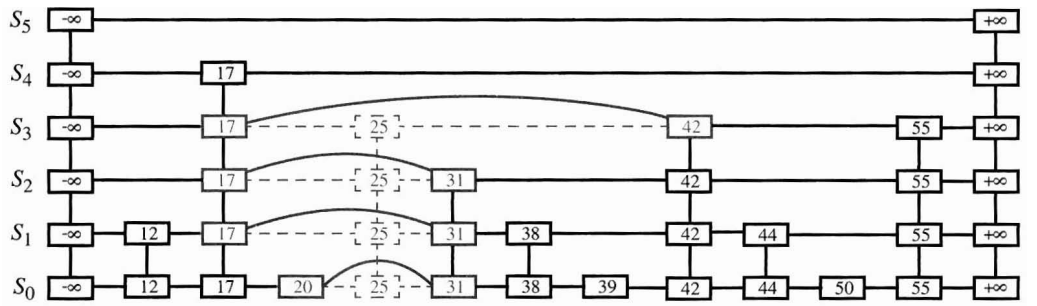

Figure 9.12: Removal of the entry with key 25 from the skip list of Figure 9.11. The positions visited after the search for the position of S_0 holding the entry are highlighted in blue. The positions removed are drawn with dashed lines.

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 dictionary (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 dictionary. 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 find, insert, 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 dictionary 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 ($\star$) 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.