10.4 Skip Lists

In Section 10.3.1, we saw that a sorted table will allow $O(\log n)$-time searches via the binary search algorithm. Unfortunately, update operations on a sorted table have $O(n)$ worst-case running time because of the need to shift elements. In Chapter 7 we demonstrated that linked lists support very efficient update operations, as long as the position within the list is identified. Unfortunately, we cannot perform fast searches on a standard linked list; for example, the binary search algorithm requires an efficient means for direct accessing an element of a sequence by index.

An interesting data structure for efficiently realizing the sorted map ADT is the skip list. Skip lists provide a clever compromise to efficiently support search and update operations; they are implemented as the java.util.ConcurrentSkipListMap class. 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 sentinel 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 sentinels $-\infty$ 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 10.10. 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 $S$.

Intuitively, the lists are set up so that $S_{i+1}$ contains roughly alternate entries of $S_i$. However, the halving of the number of entries from one list to the next is not enforced as an explicit property of skip lists; instead, randomization is used. As

![Figure 10.10: Example of a skip list storing 10 entries. For simplicity, we show only the entries' keys, not their associated values.](image-url)
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. As a consequence, we expect the height $h$ of $S$ to be about $\log n$.

Functions that generate random-like numbers are built into most modern computers, because they are used extensively in computer games, crytography, and computer simulations. Some functions, called pseudorandom number generators, generate such numbers, starting with an initial seed. (See discussion of the java.util.Random class in Section 3.1.3.) 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.

An advantage of using randomization in data structure and algorithm design is that the structures and methods that result can be simple and efficient. 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 methods when inserting or deleting entries. Nevertheless, the bounds are expected for the skip list, while binary search of a sorted table has a worst-case bound.

A skip list makes random choices in arranging its structure in such a way that search and update times are $O(\log n)$ on average, where $n$ is the number of entries in the map. 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.

As with 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($p$): Returns the position following $p$ on the same level.
- prev($p$): Returns the position preceding $p$ on the same level.
- above($p$): Returns the position above $p$ in the same tower.
- below($p$): Returns the position below $p$ in the same tower.

We conventionally assume that these operations return null 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 individual 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.
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 \( k \) and finds the position \( p \) of the entry in list \( S_0 \) that has the largest key less than or equal to \( k \) (which is possibly \(-\infty\)).

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 topmost, 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 10.11), where \( \text{key}(p) \) denotes the key of the entry at position \( p \):

1. If \( S.\text{below}(p) \) is null, then the search terminates—we are at the bottom and have located the entry in \( S \) with 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 \( p = S.\text{below}(p) \).

2. Starting at position \( p \), we move \( p \) forward until it is at the rightmost position on the present level such that \( \text{key}(p) \leq k \). We call this the scan forward step. Note that such a position always exists, since each level contains the keys \(+\infty\) and \(-\infty\). It may be that \( p \) remains where it started after we perform such a forward scan for this level.

3. Return to step 1.

![Diagram of a skip list](image)

**Figure 10.11:** Example of a search in a skip list. The positions examined when searching for key 50 are highlighted.

We give a pseudocode description of the skip-list search algorithm, SkipSearch, in Code Fragment 10.14. Given this method, we perform the map operation \( \text{get}(k) \) by computing \( p = \text{SkipSearch}(k) \) and testing whether or not \( \text{key}(p) = k \). If these two keys are equal, we return the associated value; otherwise, we return null.
10.4. Skip Lists

**Algorithm** SkipSearch($k$):

- **Input:** A search key $k$
- **Output:** Position $p$ in the bottom list $S_0$ with the largest key having $\text{key}(p) \leq k$

\[
p = s \quad \{\text{begin at start position}\}
\]

while below($p$) ≠ null do
    \[
p = \text{below}(p) \quad \{\text{drop down}\}
\]

while $k \geq \text{key}(\text{next}(p))$ do
    \[
p = \text{next}(p) \quad \{\text{scan forward}\}
\]

return $p$

**Code Fragment 10.14:** Algorithm to search a skip list $S$ for key $k$. 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. Navigation starting at the position identified by SkipSearch($k$) can be easily used to provide the additional forms of searches in the sorted map ADT (e.g., ceilingEntry, subMap).

**Insertion in a Skip List**

The execution of the map operation put($k, v$) begins with a call to SkipSearch($k$). 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 $-\infty$). If $\text{key}(p) = k$, the associated value is overwritten with $v$. Otherwise, we need to create a new tower for entry ($k, v$). We insert ($k, v$) immediately after position $p$ within $S_0$. After inserting the new entry at the bottom level, we use randomization to decide the height of the tower for the new entry. We "flip" a coin, and 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 its tower. A fair coin flip can be simulated with Java's built-in pseudorandom number generator java.util.Random by calling nextBoolean(), which returns true or false, each with probability 1/2.

We give the insertion algorithm for a skip list $S$ in Code Fragment 10.15 and we illustrate it in Figure 10.12. The algorithm uses an insertAfterAbove($p, q, (k, v)$) method that inserts a position storing the entry ($k, v$) 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, 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 $O(\log n)$, as we show in Section 10.4.2.
Algorithm SkipInsert\((k, v)\):

**Input:** Key \(k\) and value \(v\)

**Output:** Topmost position of the entry inserted in the skip list

1. \(p = \text{SkipSearch}(k)\) \{position in bottom list with largest key less than \(k\)\}
2. \(q = \) null \{current node of new entry’s tower\}
3. \(i = -1\) \{current height of new entry’s tower\}

repeat

1. \(i = i + 1\) \{increase height of new entry’s tower\}
2. if \(i \geq h\) then
   1. \(h = h + 1\) \{add a new level to the skip list\}
   2. \(t = \text{next}(s)\) \{grow leftmost tower\}
   3. \(s = \text{insertAfterAbove}(\text{null}, s, (-\infty, \text{null}))\)
   4. \(\text{insertAfterAbove}(s, t, (+\infty, \text{null}))\) \{grow rightmost tower\}
3. \(q = \text{insertAfterAbove}(p, q, (k, v))\) \{add node to new entry’s tower\}
4. while \(\text{above}(p) == \text{null}\) do
   1. \(p = \text{prev}(p)\) \{scan backward\}
   2. \(p = \text{above}(p)\) \{jump up to higher level\}

until \(\text{coinFlip}() == \text{tails}\)

5. \(n = n + 1\) \{top node of new entry’s tower\}

return \(q\)

**Code Fragment 10.15:** Insertion in a skip list of entry \((k, v)\) We assume the skip list does not have an entry with key \(k\). Method \(\text{coinFlip}()\) returns “heads” or “tails”, each with probability 1/2. Instance variables \(n\), \(h\), and \(s\) respectively hold the number of entries, the height, and the start node of the skip list.

**Figure 10.12:** Insertion of an entry with key 42 into the skip list of Figure 10.10 using method SkipInsert (Code Fragment 10.15). 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 of the tower of the new entry (variable \(q\)) are drawn with thick lines, and the positions preceding them (variable \(p\)) are flagged.
10.4. Skip 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. To perform the map operation $\text{remove}(k)$, we will begin by executing method $\text{SkipSearch}(k)$. If the returned 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 $p$. While removing levels of the tower, we reestablish links between the horizontal neighbors of each removed position. The removal algorithm is illustrated in Figure 10.13 and a detailed description of it is left as an exercise (R-10.24). As we show in the next subsection, the remove operation 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 do not actually need to store references to values at the levels of the skip list above the bottom level, because all that is needed at these levels are references to keys. In fact, we can more efficiently represent a tower as a single object, storing the key-value pair, and maintaining $j$ previous references and $j$ next 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 strictly a top-down, scan-forward fashion. We explore the details of this optimization in Exercise C-10.55. 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 11.

![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 $S_0$ holding the entry are highlighted in blue. The positions removed are drawn with dashed lines.](image-url)