Lecture Notes on Skip Lists

Prepared by Ge Xia for CPSC 211
Modified by Jennifer Welch
Based on the presentation in [1]

1 Introduction

If you have a dictionary of n items, the most common operations are search, insert and delete. Two common data structures for implementing a dictionary are an *array* and a *linked list*. These two simple data structures have somewhat complementary properties in terms of the performances. A dictionary stored in a sorted array allows for fast searches — $O(\log n)$ using binary search — but insertions and deletions are slow — $O(n)$, even if the right positions are known, due to the necessity of shifting. On the other hand, if the dictionary is implemented with a linked list, searches are slow — $O(n)$ — but once the right positions are found, insertions and deletions are extremely fast – $O(1)$. You may wonder if there are ways to combine the strengths of both data structures. An elegant yet simple solution is the *skip list*.

A key feature of the sorted array that allows fast searches is that it allows elements in the array to be *skipped* in the search process. If we could do something similar in a linked list, we might have a data structure that is fast for both search and update operations. The intuition behind the skip list is a "zoom-in" procedure on linked lists: we begin our search at a larger scale, and then gradually tune to finer scales to find the target item, similar to the procedure of finding a location on a map: first look for the particular state, then the city, the street, and so on.

We begin by informally describing how the skip list is constructed, followed by implementation of the search and delete operations on skip lists. We will show that the skip list implementation of an n-element dictionary has $O(n)$ entries (space usage) *on average*, and searching and updating cost $O(\log n)$ time *on average*. In fact, more carefully analysis shows that all the above happen with high probability. In these notes, we limit our attention to the analysis of expectations. For more advanced topics, refer to Chapter 8.3 of [2] or the original paper by W. Pugh [3].

2 Structure of Skip Lists

The idea is to have a series of linked lists: the bottom-most list contains all the items, the next list up contains about half the items (ideally, every other item), the next list after that contains about a quarter of the items (ideally, every other item that was in the second list), etc.

To implement this idea, given an ordered dictionary of n items, create a doubly-linked list S_0 with two special keys, $-\infty$ and $+\infty$, of the form

$$-\infty \leftrightarrow e_1 \leftrightarrow e_2 \leftrightarrow \ldots \leftrightarrow e_n \leftrightarrow +\infty.$$

We repeat the following procedure for $h = 0, 1, \ldots$ until S_h consists of only $-\infty$ and $+\infty$: make a copy of S_h, remove each element e_i in S_h with probability $1/2$, and denote the new doubly linked list S_{h+1}. Finally, the lists $S_0, S_1, \ldots, S_h$ are piled up in increasing order and the same elements in adjacent lists are vertically doubly-linked. In the discussions that follows, we refer to S_i as level i of the skip list. Figure 1 is an example of a skip list storing 10 entries.

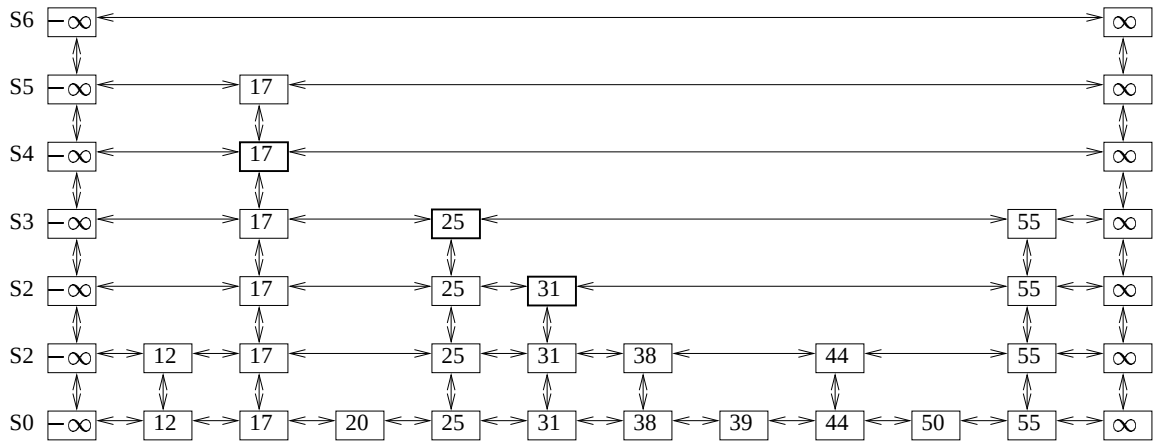

Figure 1: Example of a skip list storing 10 entries.

3 Search and Update Operations in a Skip List

Searching for a key k in a skip list is quite simple. We always start from the top-most, left position storing the special entry with key $-\infty$. We repeat the following procedure until we find the largest entry in S_0 with key less than or equal to k: drop down to the next lower level by a vertical link,

then move forward by horizontal links until reaching the right-most position with key less than or equal to the search key k. The pseudo-code of the search algorithm is given as follows:

SkipSearch(k)

Input: A search key k

Output: Position p of the entry in S_0 with the largest key less than or equal to k

1. $p \leftarrow$ topleft
2. **while** below(p) $\neq$ null **do**
   $p \leftarrow$ below(p)
   while $k \geq$ key(next(p)) **do**
   $p \leftarrow$ next(p)
3. return p.

Figure 2 is an example of searching for the key 44 in the previous skip list.

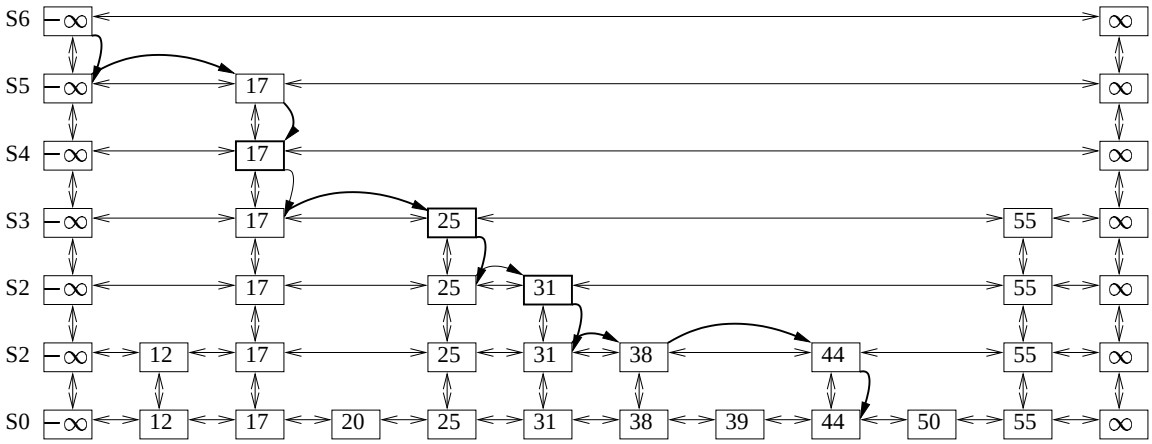

Figure 2: Example of searching for the key 44 in a skip list.

The insertion of an entry with key k is a skip list is essentially the same as searching. We first call the search operation to find the largest entry in S_0 with key less than or equal to k, then insert the new entry immediately after. We use coin flips to decide in how many levels k will appear, and then backtrack to insert k into higher levels. The pseudo-code of the insertion algorithm is given as follows:

SkipInsert(k, v)

Input: Key k and value v

Output: Entry inserted in the skip list

1. $p \leftarrow$ SkipSearch(k)
2. $q \leftarrow$ insertAfterAbove$(p,$ null, $(k, v))$ // insert (k, v) after p and above *null*
3. **while** coinFlip() = heads **do**
   while above(p) = null **do**
   $p \leftarrow$ prev(p)
   if p = topleft **then**
   add a new level above the top level
   $p \leftarrow$ above(p)
   $q \leftarrow$ insertAfterAbove$(p, q, (k, v))$ // insert (k, v) entry after p and above q
4. return q.element().

Figure 3 is an example of inserting an entry with key 42 into the previous skip list.

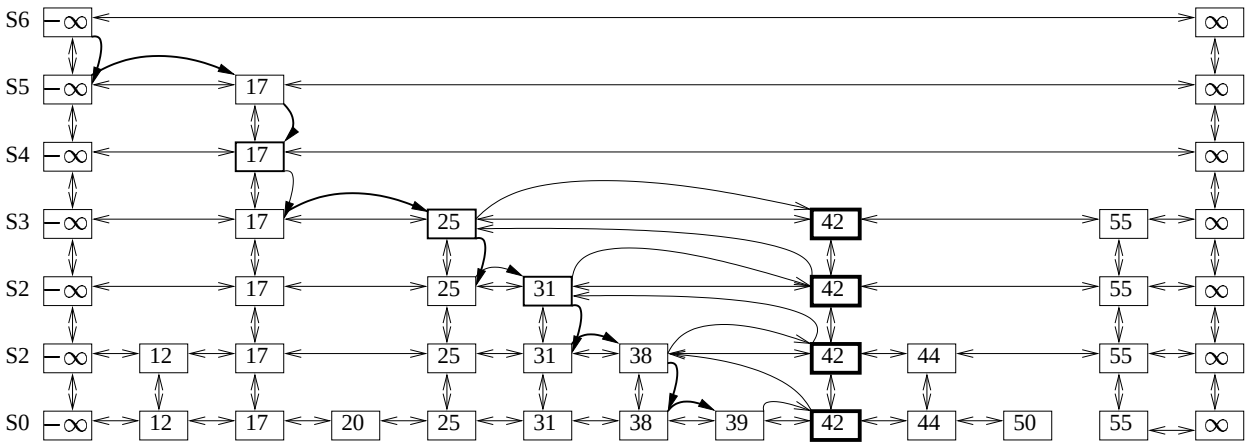

Figure 3: Example of inserting an entry with key 42 in a skip list.

The deletion of an entry with key k is a skip list is similar. We first call the search operation to find the position of the entry with key k, then remove the entry and all the positions above it from the skip list. The doubly linked lists become handy in this operation, but they are not necessary. In fact, we can replace all the doubly linked lists in the skip list with linked lists without affecting

the asymptotic performance. Figure 4 is an example of deleting an entry with key 25 from the previous skip list.

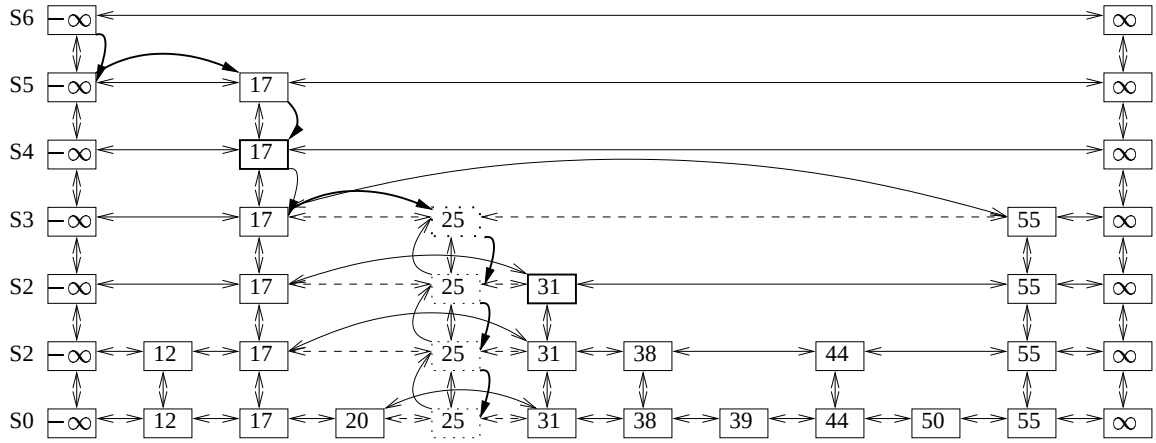

Figure 4: Example of deleting an entry with key 25 from a skip list.

4 An Analysis of Skip Lists

An important feature of the skip list data structure is that it uses randomization to construct a skip list or insert a new entry. Most programming languages provide various random number generating functions. One may wonder how could a computer, an inherently deterministic device, generate random numbers? In fact, most of the random numbers we use are so called "pseudo-random numbers", that are generated by *pseudo-random number generators*, starting with an initial number called a *seed*. For most practical purposes, carefully generated pseudo-random numbers are indistinguishable from true random numbers. In our discussion, the probability space is the set of all possible outcomes of the random numbers used in the construction and insertions.

To determine the number of entries in the skip list, we estimate how many times a particular key will appear in the skip list. It is clear that the number is equal to τ – the number of times we must flip a fair coin to come up the first tails. The expected value of τ is 2, since $\mathrm{E}[\tau] =$ E[Position of the first tail] $= \sum_{i \geq 1} i/2^i = 2$. We have that each key appears in two entries on average, and there are n distinctive keys. Hence there are $2n$ entries in the skip list on average.[1]

[1] Another way to think about this is that, if we indeed achieved our goal of having each list contain half the keys of the previous list, the total number of keys would be $n + n/2 + n/4 + n/8 + \ldots \leq 2n$.

To determine the running time of a search operation in the skip list, we first estimate the number of scan-forward steps we make at any level i. Observe that after the key at the starting position, each additional key examined in a scan-forward at level i cannot belong to level $i+1$. For each key appears in level i, the probability that it does not appear in level $i+1$ is $1/2$. Hence the number of scan-forward steps we make at any level i is equal to τ, and recall that the expected value of τ is 2. (Note that it is not correct to multiply $E[\tau]$ with the expected height of the skip list $E[h]$ to obtain a upper bound of the running time for searching, since the two random variables are not independent. Instead we need to bound the height with high probability.)

Next we bound the number of levels in the skip list. For each key, the probability that it appears in level i is at most $1/2^i$. Summing over all n keys, the probability that the skip list has h or more levels is at most $n/2^h$. In particular, the probability that the the skip list has $3\log n$ or more levels is at most $1/n^2$. Since the total number of steps in each search is at most $O(n)$, the case $h \geq 3\log n$ does not contribute significantly to the expected steps of searching, and hence the running time searching is dominated by the case $h < 3\log n$. Combining with the fact that number of scan-forward steps we make at any level i is $O(1)$, we have that the expected running time of searching is $O(\log n)$.

It is clear from the description of the previous section that the insert and delete operations are essentially the same as the search operation. The overheads do not change the asymptotic performance. Hence, the expected running times of insert and delete are both $O(\log n)$.

Finally, it is worthwhile to point out that we only estimated the expected space usage and the expected running time in this note. It is possible that with non-negligible probability, the actual cost is asymptotically worse. Therefore, it is often desirable to bound the cost with high probability, which is usually more difficult than merely computing the expectations.

References

[1] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java, 3rd Edition*, John Wiley & Sons, Inc., 2004.

[2] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York, 1995.

[3] W. Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6): 668-676 1990.