1 Skip lists

Continuing on the theme of using randomization to decouple the performance of an algorithm from the structure of any particular input—and thereby reducing the likelihood of both the best- and worst-case performance—we'll explore a randomized data structure called a *skip list*. We'll also meet a few new tools from probability theory for analyzing algorithm performance.

Skip lists were introduced in 1990 by William Pugh[1] as a probabilistic alternative to balanced tree structures liked AVL trees and red-black trees. (Other data structures with similar behavior include treaps, B-trees and splay trees.) Like balanced trees, skip lists are a data structure for storing information and thus can be thought of like any other *abstract data type* (ADT).[2]

The skip list ADT supports the standard operations for managing a set of data:

Skip List		
`Add(x)`	$O(\log n)$ time	Adds or inserts a value x to the data structure.
`Find(x)`	$O(\log n)$ time	Return *true* if x is in the data set, otherwise return *false*.
`Remove(x)`	$O(\log n)$ time	Removes or deletes a value x from the data structure.

As with other data structures, these behaviors can be easily generalized to store $(key, data)$ pairs, where the *data* associated with a particular key x can be arbitrarily complex. In this case, `Find(x)` will return the *data* associated with the key x iff x is in the data structure; otherwise, it returns `NULL` or *false*.

A skip list is a kind of generalized or "multi-level" linked list. At the bottom-most level is a standard linked list that contains all the $(key, data)$ pairs stored in the skip list. Each element in the skip list is a member of $O(\log n)$ *additional* linked lists, which are cross-linked with the bottom-most level. Crucially, membership in these additional lists is determined probabilistically. The skip list uses these additional lists to emulate the behavior of *binary search* thereby yielding fast search times.[3]

The additional lists do not come for free. Not only do they impose an additional cost in the number of atomic operations during a function call, they also impose an overhead cost in the form of managing the extra lists' structure whenever we alter the stored data (e.g., adding or removing an element). A cost for maintenance is required because it takes some effort to preserve the special properties of the skip list that produce its performance guarantees. Fortunately, as we will see, these additional costs do not change the asymptotic cost of the operations.

[1] Pugh's original paper describing the skip list, which includes pseudo-code, is on the class webpage.

[2] An abstract data type is a kind of logical object that provides a consistent *interface* to the external world, with certain guarantees, that allows external programs to interact with the object's internal data in specified ways, e.g., adding, changing or deleting some data from the store. (An ADT is like an *application programming interface* (API) for a data structure.)

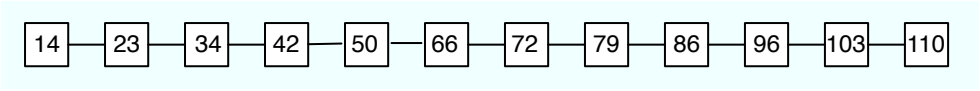
[3] Recall that binary search takes $O(\log n)$ time to find an element x in a sorted array of n items.

1.1 Searching lists: the naïve approach

Before diving into skip lists, we briefly consider the simpler problem of the `Find(x)` or `Search(x)` operation on a simple *linked list*. Linked lists offer the following ADT:

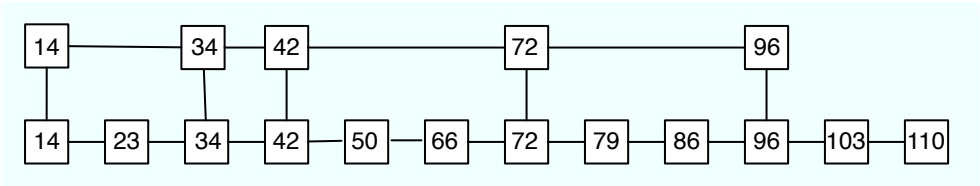
Linked List		
`Add(x)`	$O(1)$ time	adds or inserts a value x to the data structure
`Find(x)`	$\Theta(n)$ time	return *true* if x is in the data set, otherwise return *false*.
`Remove(x)`	$O(1)$ time	removes or deletes a value x from the data structure.

For the moment, ignore the `Add(x)` and `Remove(x)` operations. Searching a linked list, even if it is sorted, takes $\Theta(n)$ time (do you see why it being sorted makes no difference? how does maintaining a sorted linked list change the `Add(x)` and `Remove(x)` running times?).[4] Here's a simple example of a sorted linked list, where a solid line could indicate either a directed link or a bidirectional link.



1.2 A clever idea: express stops

Here's a clever idea: what if we use *two* sorted linked lists in parallel to speed up the search times?[5] One of the linked lists would need to be a standard linked list, as above, but the other could represent "express stops" that allow us to skip over large sections of the "local stops" in the event that our target is greater than the value stored at the next express stop. For instance, using the example above, if we let the values 14, 34, 42, 72 and 96 be express stops, this parallel list structure might look like this:



[4] There are other ways to improve the running time of the `Search(x)` operation in a linked list, but only for certain distributions over the query sequence $\sigma = y_1, y_2, \ldots$. The *move-to-front* heuristic is a classic example: much like the *splay tree* data structure, each time a call to `Search(x)` occurs, after we find x, we move its entry to the front of the list (how long does this "rewiring" take?), thereby reducing the search time for finding x if x appears in the query sequence again soon. (What is the worst σ possible for this heuristic? Can we improve its performance by using randomization?)

[5] This way of introducing skip lists comes from Erik Demaine at MIT.

A search query on this data structure should always starts on the express line (the sparser list). Let x be the search key and z_i be the ith element in the list. If $x > z_{i+1}$ then we move along the express line to the next stop (element z_{i+1}) and repeat the query; otherwise, we know that our search target is between z_i and z_{i+1} on the local line, and we move to the local line and proceed until we reach the target. To ensure that we can move back and forth between the express and local lines, each element z_i that appears in both lists contains a pointer to itself in the other list.

How much does searching this structure cost? Let L_2 denote the express stops list and L_1 denote the local stops list. The cost of a search query is then the number of (express) stops on L_2 plus the number of (local) stops we make on L_1:

$$\text{cost} = \text{length}(L_2) + \frac{\text{length}(L_1)}{\text{length}(L_2)} = \text{length}(L_2) + \frac{n}{\text{length}(L_2)} \ .$$

Thus, the more densely distributed the express stops, the longer the express L_2 list is and the lower the cost savings. How densely should we distribute them? The cost is minimized when we spend equal amounts of time on each list:

$$\begin{aligned} & \text{length}(L_2) = \frac{n}{\text{length}(L_2)} \\ \Longrightarrow\ & \text{length}(L_2) = \sqrt{n} \\ & \text{cost} = 2\sqrt{n} = \Theta(\sqrt{n}) \ . \end{aligned}$$

Thus, using two linked lists, we would want to distribute $\sqrt{n}$ express stops along the length of the local line, such that each express stop covers $\sqrt{n}$ local stops.

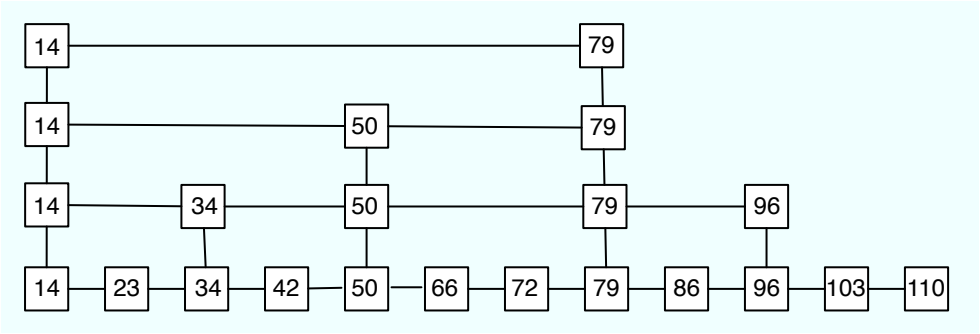
1.3 A cleverer idea: multiple express lines

We now simply repeat the trick and and add an "express-express" line. It is easy to show, by generalizing the above analysis, that a 3-list structure would yield a search cost of $3 \cdot \sqrt[3]{n} = \Theta(n^{1/3})$; a 4-list structure would yield $4 \cdot \sqrt[4]{n} = \Theta(n^{1/4})$; etc., and a k-list structure yields cost $= k \cdot \sqrt[k]{n} = \Theta(n^{1/k})$. (Can you show this?)

But, how many lists should we use? If we let $k = \log_2 n$, then we have

$$\text{cost} = \log_2 n \cdot \sqrt[\log_2 n]{n} = \log_2 n \cdot n^{1/\log_2 n} = 2\log_2 n = \Theta(\log n) \ ,$$

which is the conventional definition for "fast" search. The express lines form a kind of binary tree, and each time we move "down" the express-line hierarchy, we divide the remaining distance to the target in half. Continuing our running example from above, this is what such a structure might look like:



Note that the first element of this structure is a member of every express line, since that is where every search query begins. If the first element is the target, we need a way to get down to the bottom-most level to finish the query and return. A simple way to make this work is to use a special element representing $-\infty$ that serves as the head of all the "lines" within the skip list. This structure, of multiple parallel linked lists, is precisely what a skip list guarantees, probabilistically.

1.4 What about insert?

Given a skip list with correct internal structure, we know that search is fast, taking $\Theta(\log n)$ time. But, how do we *construct* and *maintain* such a structure, without violating our performance promises? Specifically, when we add an element to the skip list, we certainly must add it to the local line, but to which "express lines" should it also be added?

This is where probability enters into the picture: we will use a probabilistic approach to decide which express lines ("levels") the inserted item will intersect. This will produce a probabilistic guarantee of the maintenance of the skip list's internal structure. Here's the idea:

- When we add an element x to the lowest level, how many additional levels should it have?
- To mimic the balanced binary tree structure, we want roughly half of all search queries to move down the "express hierarchy" at x.
- This is achieved by adding an express stop at x to the ith express line with probability $(1/2)^i$.
- Thus, in expectation, 1/2 of stored elements will have a stop on the 1st express line, 1/4 on the 2nd line, 1/8 on the 3rd line, etc.

In practice, when we add an element to the lowest level in the hierarchy, we can achieve the desired number of levels above that element—the height of its "tower"—by tossing a fair coin ($p = 1/2$). If we succeed, i.e., if $p \leq 1/2$, then we add a level and continue flipping the coin; otherwise, we stop. (Do you see why this produces a tower with height h distributed according to a geometric

distribution?) Once we have chosen the height of the new element, there is some additional but modest overhead necessary to wire it into the express lines it now intersects.

1.5 Analysis: Time

We now analyze the behavior of the skip list data structure to show that this way of inserting an element into the data structure will maintain the $O(\log n)$ search time. The total running time is the number of elements we traverse in total, which is simply the product of the number of elements per level multiplied by the number of levels.

Thus, our strategy for this proof is two-fold. First, we will prove that the height of the entire skip list (i.e., the height of the tallest tower) is $O(\log n)$. This provides an upper bound on the number of levels we traverse to find an element. Second, we will prove that the number of elements we traverse within each level (number of stops on that express line) is $O(1)$. It follows that the total running time of a search query is $O(\log n)$.

1.5.1 The height of the tallest tower

Lemma: With high probability[6] (w.h.p.), a skip list with n elements has $O(\log n)$ levels.[7]

Proof: Coin tosses for adding a new level are independent and occur with probability $p = 1/2$. Thus, the probability that an element has a height h is given by a *geometric* distribution, which is the discrete version of the *exponential* distribution. The geometric distribution is defined as $\Pr(X = h) \propto \sum_{i=1}^{h} p^i = (1-p)^{h-1}p$, which has mean $1/p$. Its cumulative distribution function (the fraction of results with value at most h) is $\Pr(X \leq h) = 1 - (1-p)^h$.

The expected height of the skip list is given by calculating the expected maximum value for a set of n geometric random variables. To calculate the expected maximum in a set of n i.i.d. random variables, we simply solve the following equation:

$$\begin{aligned} \frac{1}{n} &= \Pr(X > h) \\ &= 1 - \Pr(X \leq h) \\ &= 1 - (1 - (1-p)^h) \ . \end{aligned}$$

[6]The technical definition of "with high probability" is the following: A (parameterized) event E_α occurs *with high probability* if, for any $\alpha \geq 1$, E_α occurs with probability at least $1 - c_\alpha/n^\alpha$, where c_α is a "constant" depending only on α. Informally, something happens w.h.p. if it occurs with probability $1 - O(1/n^\alpha)$. The term $O(1/n^\alpha)$ is called the *error probability*. The idea is that the error probability can be made extremely small by setting α large.

[7]In fact, it's $\Theta(\log n)$, but we only need an upper bound.

Taking logarithms of both sides and simplifying yields the result,

$$\begin{aligned} \lg(1/n) &= h \lg(1/2) \\ h &= \Theta(\lg n) \ , \end{aligned}$$

that is, the expected maximum height is $\Theta(\lg n)$.

We can go further and bound the deviations about this expected value by using Boole's Inequality, also called the Union Bound. (This is what is called a "concentration bound", because we prove that the probability is concentrated around its expected value.) The claim is that the height of the skip list is $\Theta(\lg n) = c \lg n$, and we now claim that this results holds *with high probability* (w.h.p.). The probability of seeing an element with a greater height is

$$\begin{aligned} \Pr(\text{element } x \text{ has more than } c \lg n \text{ levels}) &= (1-p)^{c \lg n} \\ &= (1/2)^{c \log_2 n} \\ &= 1/n^c \ . \end{aligned}$$

The Union Bound says that for a set of events $A_1, A_2, \ldots$, the probability that at least one of the events happens is bounded from above by the sum of the probabilities of the individual events.[8] Mathematically, we say

$$\begin{aligned} \Pr(A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_n) &\leq \Pr(A_1) \text{ or } \Pr(A_2) \text{ or } \ldots \text{ or } \Pr(A_n) \\ &\leq \sum_{i=1}^{n} \Pr(A_i) \ . \end{aligned}$$

There are n random variables we need to bound, each of which is drawn from the geometric distribution. Because each of these events is independent and occurs with equal probability, we may apply the Union Bound. This reduces the sum to simply multiplying the probability that one event occurs times the number of trials n:

$$\Pr(\text{element } x \text{ has more than } c \log n \text{ levels}) \leq n \cdot 1/n^c = 1/n^{c-1} \ .$$

Thus, the deviations from our claim are polynomially small and the exponent $c-1$ can be made as large as we want (making the deviation as small as we want) by an appropriate choice of the constant in $O(\log n)$.

[8] The union bound can be derived by induction on n of the statement $\Pr(A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_n) \leq \Pr(A_1) + \Pr(A_2 \text{ or } A_3 \text{ or } \ldots \text{ or } A_n)$, using the identity $\Pr(A \text{ or } B) = \Pr(A) + \Pr(B) - \Pr(A \text{ and } B)$.

1.5.2 The number of stops on level k

Our approach for analyzing the number of stops at level k is to run the search query's sequence of operations *backwards*, moving from the target element's location at the lowest level in the skip list upward and leftward through the different levels until we reach the "root" of the effective tree structure. (This approach is like analyzing the "depth" of a search through a binary tree structure by moving from the leaf to the root.)

Let us begin at the end, when we have found the target x in the lowest level of the skip list. Tracing backward in the search path, at each node we visited, we can ask the following:

- If the node wasn't promoted higher, then we must have gotten a `tails` on the coin flip here when we generated this element's tower height, which implies that the search path came from the next element to the left in this level.
- If the node was promoted higher, then we must have gotten a `heads` here, which implies that the search path came from the next level up.

This backward-search process stops at the "head" element storing $-\infty$. Note that each of these two cases (move left or move up) occurs with probability 1/2.

Given that we're in a particular level k, the probability that we move "up" in the hierarchy versus "left" along the current level is equal to the probability that the tower we're currently on is taller than k, which occurs with $p = 1/2$. Thus, in expectation, we will make $1/p = O(1)$ steps on the kth level before we find a taller tower, at which point we will move to the $(k+1)$th tower.

In fact, this result holds w.h.p. Because the probability of encountering a taller tower as we move left is p, the number of steps we take on a given level is given by a geometric distribution. Thus, the probability that we take more than s steps on a given level is

$$\Pr(\text{take more than } 1/p \text{ steps}) = p^{1/p} \ .$$

So long as p is some constant, this probability will also be a constant, and thus w.h.p. we take only a constant number of steps on any particular level in the skip list.

Ergo, the overall search time is $O(\log n)$ w.h.p.

1.6 Analysis: Space

Recall that the space required by each of the n elements in the skip list is given by its number of levels. The total space requirements is just the sum of these heights. Recall from Section 1.5.1 that the number of levels of a given element is given by a geometric distribution, which has mean $1/p$.

Because the coin we use is fair ($p = 1/2$), the expected height h of any tower is $E(h) = 2$. There exactly n elements in the skip list, therefore the expected space usage is simply $\Theta(n)$. (Can you further bound the space usage w.h.p.?)

2 Next time

1. Finish up skip lists (concentration bound on length of search)
2. Hash tables