Design and Analysis of Algorithms CSCI 5454
Lecture 3 24 January 2011
Prof. Aaron Clauset

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—today we'll explore a randomized data structure called a *skip list*. We'll also meet a few new tools from probability theory for analyzing algorithms performance.

Skip lists were introduced in 1990 by William Pugh as a probabilistic alternative to balanced tree structures liked AVL trees and red-black trees, both of which we briefly discussed in Lecture 2. (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).

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 x if x is in the data structure, and will return `NULL` or *false* otherwise.

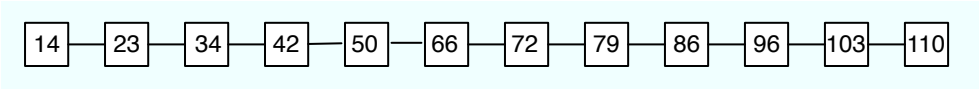
Generally, you can think of a skip list data structure as a kind of generalized linked list. At the very bottom level, is a simple linked list of all the elements. Crucially, however, each element in the skip list is a member of $O(\log n)$ *additional* linked lists, but membership is probabilistic. Using these additional lists, we can emulate the behavior of *binary search* thereby yielding fast search times. The cost of managing these additional linked lists, however, increases the cost of the add and delete operations a little.

1.1 Searching lists: the naïve approach

Before diving into skip lists, let's briefly consider the simpler problem of the `Find(x)` or `Search(x)` operation on a simpler data structure, an sorted *linked list*. Linked lists offer the following features:

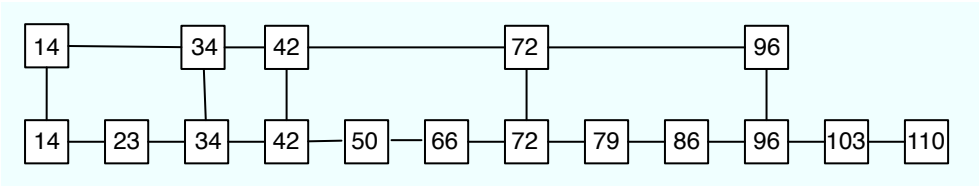
Linked List		
`Add(x)`	$O(1)$ time	adds or inserts a value x to the data structure
`Find(x)`	$\Theta(n)$ time	return *true* if x is in the data set, otherwise return *false*.
`Remove(x)`	$O(1)$ time	removes or deletes a value x from the data structure.

Ignoring the insert and delete 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?).[1] 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

What if we use *two* sorted linked lists in parallel to speed up the search times. One of the linked lists would need to be a standard linked list, as in Section 1.1, but the other could represent "express stops" that allow us to skip over large sections of the "local stops". 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:



To search this structure, we would start 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 element z_{i+1} and repeat the query; otherwise, we know that our search target is between z_i and z_{i+1} in the local line, and we move our search to the local line and proceed. To ensure that we can move back and forth between the express and local lines, each element that appears in both

[1] 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?)

lists is cross-linked.

How much does searching this structure cost? Let LL1 denote the "express stops" list and LL2 denote the "local stops" list. Then,

$$\text{cost} = \text{length(LL1)} + \frac{\text{length(LL2)}}{\text{length(LL1)}} = \text{length(LL1)} + \frac{n}{\text{length(LL1)}} \ .$$

Thus, the more densely distributed the express stops, the longer the LL1 list is and the lower the cost savings. How densely should we distribute them? The cost is minimized when

$$\begin{aligned} & \text{length(LL1)} = \frac{n}{\text{length(LL1)}} \\ \Longrightarrow\ & \text{length(LL1)} = \sqrt{n} \\ & \text{cost} = 2\sqrt{n} \ . \end{aligned}$$

Thus, in this data structure, 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.

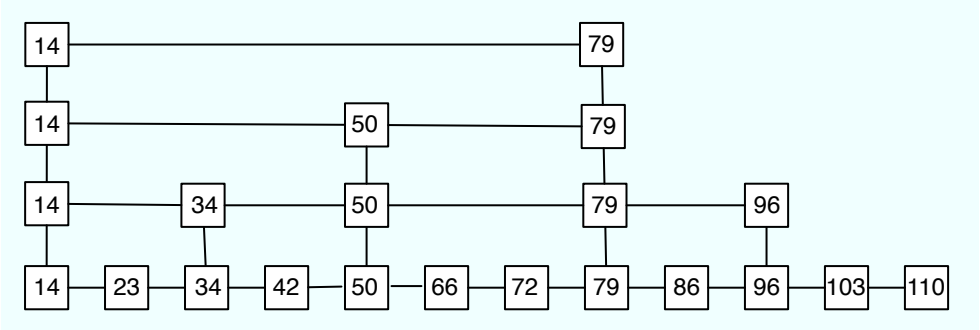
1.3 A cleverer idea: multiple express lines

There's no reason we can't repeat this trick and add a "super-express" line. It's easy to show, by generalizing the above analysis, that a 3-list structure would yield a search cost of $3 \cdot \sqrt[3]{n}$; a 4-list structure would yield $4 \cdot \sqrt[4]{n}$; etc., and a k-list structure yields cost $= k \cdot \sqrt[k]{n}$. (Can you show this?)

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

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

a conventional target for "fast" search. The express lines effectively form a binary tree, and each time we move "down" the hierarchy of express lines, we divide the remaining distance to the target in half. For instance, continuing our running example, see the Figure on the next page:



Note that the first element of this structure must be a member of every express line, since that's where the search begins. 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 the underlying structure of a skip list.

1.4 What about insert?

Given a skip list with the appropriate structure, we know that search is fast, taking $\Theta(\log n)$ time. But, how do we *construct* and *maintain* such a structure? That is, when we add an element to the skip list, it's clear that to be findable it must be added to the local line, but to which "express lines" should it also be added?

This is where probability enters into the picture. That is, we'll use a probabilistic approach to answer the question of which express lines ("levels") to intersect, which will provide a probabilistic guarantee of the maintenance of the skip list 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'd like roughly half of the searches to move down the "express hierarchy" at x.
- We can mimic this by adding an express stop to the ith express line with probability $(1/2)^i$.
- Thus, in expectation, 1/2 of 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've chosen the height of the new element, there is some additional overhead necessary to wire it into the express lines it now intersects.

1.5 Analysis: Time

Now we'll analyze the behavior of the skip list data structure, showing that this way of inserting an element into the data structure will maintain the $O(\log n)$ search time. Our strategy here is two-fold. First, we'll prove that the height of the entire skip list is $O(\log n)$. This bounds the number of levels we traverse to find an element. Second, we'll prove that the number of steps we take within each level is $O(1)$.

1.5.1 The height of the tallest tower

Lemma: With high probability[2] (w.h.p.), a skip list with n elements has $O(\log n)$ levels.[3]

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 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:

$$\begin{aligned} \frac{1}{n} &= \Pr(X > h) \\ &= 1 - \Pr(X \leq h) \\ &= 1 - (1 - (1-p)^h) \\ \lg(1/n) &= h \lg(1/2) \\ h &= \Theta(\lg n) \ . \end{aligned}$$

Thus, in expectation, the 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. (I mentioned last week that we would rarely see concentration bounds, but here is one.) We've claimed that the height of the skip list is $c \lg n$; we also claim that this

[2]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.

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

holds *with high probability* (w.h.p.). The probability of seeing an element with a greater height is

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

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.[4] Mathematically, we say

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

There are n random variables we need to bound, each of which is drawn from the geometric distribution. Thus, applying the Union Bound to these, we see that

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

which shows that the deviations 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 $O(\log n)$.

1.5.2 The number of stops on level k

The approach we'll take here is to analyze the search *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.

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

[4]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 $\Pr(A \text{ or } B) = \Pr(A) + \Pr(B) - \Pr(A \text{ and } B)$.

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.

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

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