CMSC 420: Skip Lists

Justin Wyss-Gallifent

April 11, 2024

1 Introduction

Skip lists were invented in 1990 by Bill Pugh, at UMD. These are a modification of a regular sorted linked list which provides logarithmic search, insertion and deletion, much like AVL trees. On the down side they require about twice as much storage.

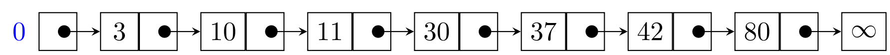
They do this by introducing an element of randomness to the sorted linked list construction, as we'll see.

2 Ideal Skip Lists

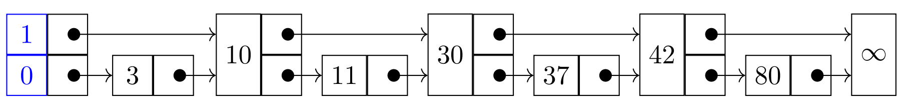
Suppose we have a sorted linked list with n keys. Clearly it takes $\mathcal{O}(n)$ time to find, insert and delete (insertion and deletion since we must find first). Is there a way to speed this up?

Let's start by hypothesizing a sorted linked list with just 8 elements. Suppose we also have a head node which is simply a pointer to the first element in the list and a tail node which is pointed to by the last element in the list.

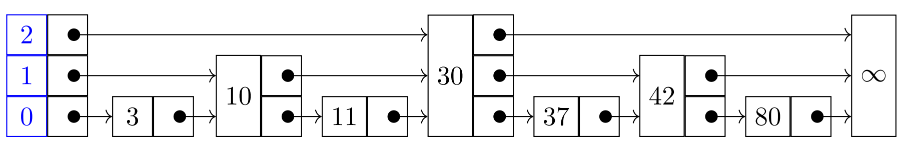
In the picture below ignore the 0 in the head node for now, don't treat it as a key in the list.



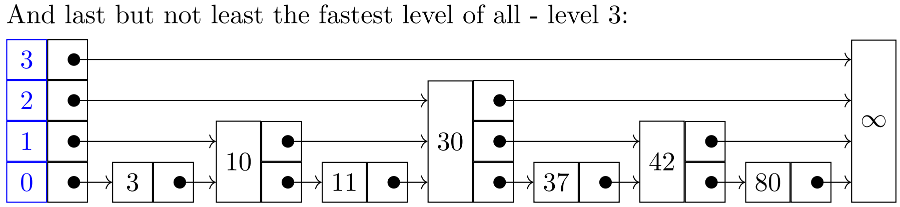
As mentioned at the start worst-case for find, insert, and delete is $\mathcal{O}(n)$ in this case. But suppose we added a "fast level" of pointers. To elaborate, think of the 0 in the head node above as level 0, the really slow level. We'll add a fast level of pointers at level 1 which skips every other node in level 0:



Great, so what if we added an extra fast level - level 2:



And last but not least the fastest level of all - level 3:



Okay, let's pause for a second to see why this might be useful. Suppose we are looking for the key 37.

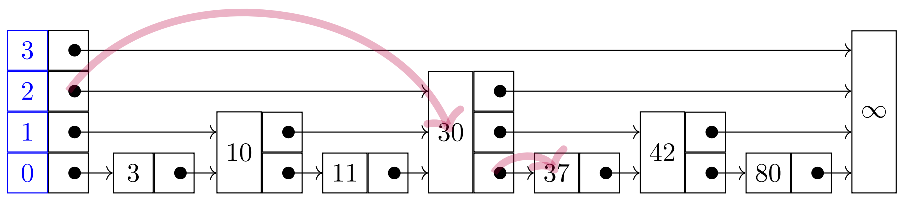
We start at the top pointer in level 3 and observe that it points to ∞. This is too large so we drop down to level 2.

We check the pointer in level 2 and observe that it points to 30. This is below our target of 37 so we follow the pointer to that node. We then check the next pointer in level 2 and observe that it points to ∞. This is too large and so we drop down to level 1.

We check the next pointer in level 1 and observe that it points to 42. This is too large and so we drop down to level 0.

We check the next pointer in level 0 and observe that it points to 37. This is our target and we're done!

Here is an illustration:



In an ideal skip list with n elements we would have level 0 which links all nodes, level 1 which skips every other node in level 0, level 2 which skips every other node in level 1, and so on until adding more levels adds nothing. Then we would find our target key using a generalization of the above approach.

3 Randomized Skip Lists

This is all very quaint but the truth of the matter is that once a key is inserted or deleted the entire structure is ruined and we certainly don't want to have to rebuild all the pointers every time this happens.

Instead then we'll add an element of randomness to the procedure as follows. First we set an *enforced maximum level*. We create a head node which reaches this enforced maximum level and has no value (sometimes $-\infty$) and we create a tail node which also reaches this enforced maximum level and has a value of ∞.

Note that in all of what follows, as with our non-randomized case, the levels are 0-indexed starting at the bottom.

1. In level 0 we will have a regular linked list, starting with the head node, progressing through our keys, and ending with the tail node.
2. In level 1 we take each internal node which reaches level 0 (all of them) and say that there is a $p = 0.5$ probability that it also reaches level 1. We then form a linked list using these nodes, again starting with the head node, progressing through the nodes that do extend to level 1, and ending

with the tail node.

3. In level 2 we take each internal node which reaches level 1 (about half of them) and say that there is a $p = 0.5$ probability that it also reaches level 2. We then form a linked list using these nodes, again starting with the head node, progressing through the nodes that do extend to level 2, and ending with the tail node.
4. In level 3 we take each internal node which reaches level 2 (about quarter of them) and say that there is a $p = 0.5$ probability that it also reaches level 3. We then form a linked list using these nodes, again starting with the head node, progressing through the nodes that do extend to level 3, and ending with the tail node.
5. We continue this until we "run out of" nodes. Note that in theory nodes could keep being included in higher and higher levels albeit with lower and lower probability so we stop for sure when we hit our enforced maximum level.

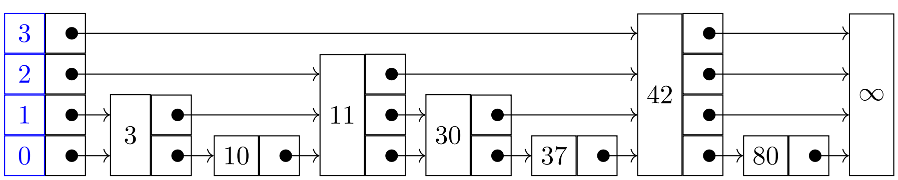
A few definitions. The first is repeated.

Definition 3.0.1. We have:

- The *(enforced) maximum level* is the upper limit that we set on how far each node could reach.
- The *top level for the skip list* is the topmost level that we actually reach.
- The *top level* for a given node is simply the largest level index it reaches.

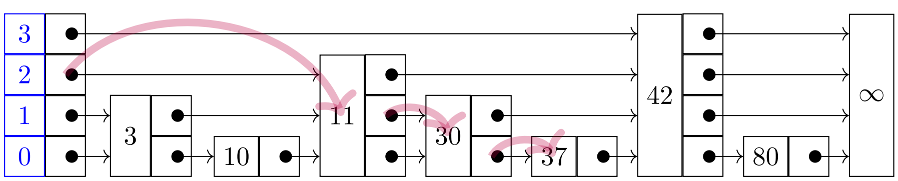
The latter two are of course less than or equal to the enforced maximum level.

Example 3.1. A randomized skip list might look like the following. The top levels of the nodes were generated by flipping a coin with an enforced maximum level of 3 (indexed 0 to 3):



So now if we're looking for 37 our path will be $H \longrightarrow 11 \longrightarrow 30 \longrightarrow 37$ and our path to 80 will be a really fast $H \longrightarrow 42 \longrightarrow 80$.

Here's the path for 37:



4 Measurements

4.1 Important Note

All of the analysis here is based upon having no enforced maximum level. Having an enforced maximum level doesn't change the $\mathcal{O}$ bounds.

4.2 Levels

Theorem 4.2.1. Regarding levels in a skip list with n nodes and for which $TL(n)$ is the top level (index).

(a) For a given $L \geq 0$, the probability that at least one node reaches level L or beyond equals:

$$P = 1 - \left(1 - \frac{1}{2^L}\right)^n$$

(b) The expected top level of a skip list with n nodes is $\Theta(\lg n)$.

(c) The expected maximum number of levels in a skip list with n nodes is $\Theta(\lg n)$.

Proof. We have:

(a) For any level L, the probability that a single node reaches level L (and possibly beyond) is $1/2^L$ and so the probability that a single node does not reach level L (or beyond) is $1 - 1/2^L$.

Because the nodes are independent, it follows that the probability that none of the nodes reaches level L (or beyond) equals:

$$\left(1 - \frac{1}{2^L}\right)^n$$

Hence the probability that at least one node does reach level L (or beyond) is:

$$1 - \left(1 - \frac{1}{2^L}\right)^n$$

(b) Intuitively since on average the number of nodes is half as many at each level it makes sense that we expect a logarithmic number of levels. At some point I'll put in a more rigorous proof.

(c) Follows from (c).

$\mathcal{QED}$

Example 4.1. Here are some consequential statistics for the case where we have 100 nodes:

(a) The probability that at least one node reaches level $L = 10$ (or beyond) equals:

$$P = 1 - \left(1 - \frac{1}{2^{10}}\right)^{100} \approx 0.09308265650895886$$

(b) Not really specific.

Moreover it's worth noting that even if we didn't set an enforced maximum level for our nodes we still have a very low probability of achieving a high number of levels. For example with $n = 100$ nodes the probability of reaching level 15 is only:

$$P = 1 - \left(1 - \frac{1}{2^{15}}\right)^{100} \approx 0.0030471523581468984$$

This feels somewhat intuitive given that the expected top level is about 7.6.

4.3 Storage

Theorem 4.3.1. Denote by $S(n)$ the storage needed for a skip list with n entries. Then the expected storage is $E(S(n)) = \mathcal{O}(n)$.

Proof. As mentioned we are going to ignore the enforced maximum level here but we are also going to ignore the head and tail nodes. However you are encouraged to think about the fact that this doesn't affect the outcome.

First off, the keys take up $\Theta(n)$ space for n nodes so the real issue is how much space the pointers take up.

For each level $i = 0, 1, 2, ...$ there is a $1/2^i$ probability that each node reaches level i and hence we expect $n/2^i$ pointers at level i.

Consequently the storage needed is essentially the sum:

$$E(S(n)) = \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$$

$\mathcal{QED}$

Note 4.3.1. In the computation above we are using the fact that $E(x + y) = E(x) + E(y)$ for expected value calculations, a basic fact from probability.

Theorem 4.3.2. The worst-case storage without an enforced maximum level is infinite.

Proof. There is no bound on the number of pointers for any one node, let alone all n of them! $\mathcal{QED}$

5 Search

5.1 Algorithm

The search process is very easy. We start at the header node all the way on the left, at the highest level. We follow this level across until the final node before we overshoot our target. When we reach this node, we drop down a level and repeat.

Essentially we are staying in the topmost level as long as possible and then dropping down when we would miss the target.

5.2 Time Complexity

It turns out that the average case time complexity for search is $\mathcal{O}(\lg n)$.

The analysis of this is a bit sneaky. As a precursor let's refine our description of the search procedure into *steps*. A step will consist of checking a pointer, going right if we can (if we don't overshoot the target) and going down one level if we can't.

The time complexity analysis is based on following this procedure backwards. More specifically, starting at the target node the reverse of search works as follows:

(i) If we can go up one level in the node, do so (one step).

(ii) If we cannot, then go left (one step).

(iii) Go back to (i) and repeat until we reach the head node.

Each of (i) and (ii) should then be thought of as a step which takes constant time.

Theorem 5.2.1. The expected number of steps in a skip list with n nodes is $\mathcal{O}(\lg n)$.

Proof. Assume that the skip list has top level L. This could be because we enforced this level or because the nodes stopped growing there. For each i let $E(S(i))$ equal the expected number of steps taken in the top i levels of the skip list. Let's pretend that the skip list extends infinitely far left. This is of course unrealistic but will suffice for now.

At any given current node we have a 0.5 probability of moving up a level, meaning we we we go up (one step) and then we have $E(S(i-1))$ more steps because there are one fewer levels above, and we have a 0.5 probability of going left at the same level, meaning we go left (one step) and then we have $E(S(i))$ more steps because there are exactly as many levels above.

This means we have:

$$E(S(i)) = 0.5(1 + E(S(i-1))) + 0.5(1 + E(S(i)))$$

We can rewrite this with basic algebra to get a recurrence relation:

$$\begin{aligned}
E(S(i)) &= 0.5(1 + E(S(i-1))) + 0.5(1 + E(S(i))) \\
E(S(i)) &= 1 + 0.5E(S(i-1)) + 0.5E(S(i)) \\
0.5E(S(i)) &= 1 + 0.5E(S(i-1)) \\
E(S(i)) &= E(S(i-1)) + 2
\end{aligned}$$

Along with the fact that $E(S(0)) = 0$ (in the top 0 levels there are 0 nodes) this can be expanded via digging down to show that $E(S(i)) = 2i$:

$$\begin{aligned}
E(S(i)) &= E(S(i-1)) + 2 \\
&= E(S(i-2)) + 4 \\
&= E(S(i-3)) + 6 \\
&= \qquad \vdots \\
&= E(S(i-i)) + 2i \\
&= 0 + 2i
\end{aligned}$$

Returning to our assumption that the list continues infinitely far left we now comment that this is not of course true, and so in fact $E(S(i)) \leq 2i$.

Lastly note that we're starting our backwards journey from level 0 and recall that the expected number of levels is $\Theta(\lg n)$ and since $E(S(i))$ is linear as a function of i, we get our result.

$\mathcal{QED}$

Theorem 5.2.2. Search is average case $\mathcal{O}(\lg n)$ and worst-case $\mathcal{O}(n)$.

Proof. Assuming it takes constant time for each step and since on average we perform $\mathcal{O}(\lg n)$ steps, the first result follows immediately.

The second result follows from the fact that there is a probability, albeit very low, that the skip list turns out to be a regular linked list. $\mathcal{QED}$

6 Insertion Algorithm and Time Complexity

Consider the average-case time complexity of inserting a new key/node.

We first figure what the top level of this new node is using the 0.5 approach. Since the expected height of a single node is 2 we know this process is average-case $\mathcal{O}(1)$.

We then traverse the skip list to figure out where to insert the new node. This is essentially search, and is $\mathcal{O}(\lg n)$. More specifically we search for the key and since we won't find it what will happen instead is that we will end up at level 0 and overshooting the key. This will tell us where to insert the new node.

Before traversing we initialize an array A indexed by level up to the top level of the newly inserted node. During the traverse as we visit each node we update $A[i]$ to store each pointer that overshoots the target at level i, noting that we are finding these as part of the search process.

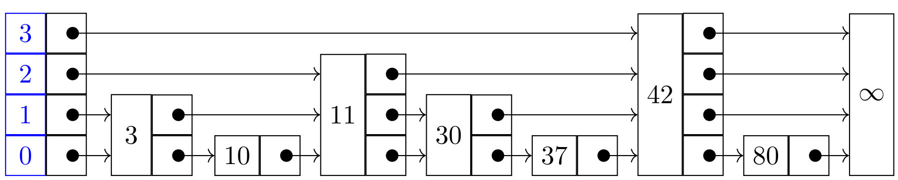
After the traverse A will contain the pointers which miss the target which are precisely those which need to be updated. Similarly to the analysis for search, this process will be $\mathcal{O}(\lg n)$.

Once we finish this we insert the node at the correct location which means we split the appropriate pointers (appropriate for each level of the newly inserted node) that we stored while traversing. Each split is $\mathcal{O}(1)$ and since the expected length of A is $\mathcal{O}(\lg n)$ this process is also $\mathcal{O}(\lg n)$.

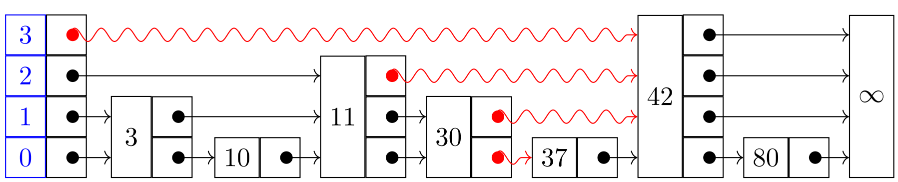
Consequently the overall time required is $\mathcal{O}(\lg n)$.

Note that this takes a bit of tweaking if the newly created node is taller than the head and tail nodes, but this doesn't affect the overall time complexity.

Example 6.1. Consider this skip list from earlier:



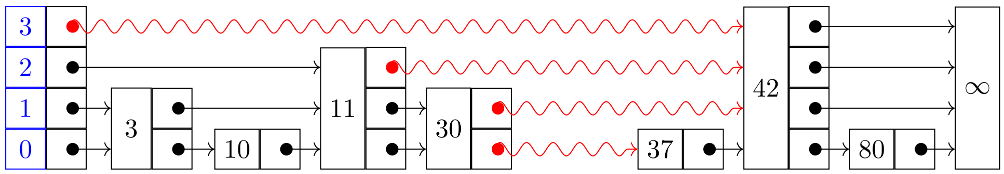
Suppose we want to insert 32 into this skip list and randomization tells us the top level should be 3. When we traverse looking for 32 we find 37 instead (because we get to level 0 and are forced to overshoot) and the pointers that we test and which overshoot the target are indicated here:



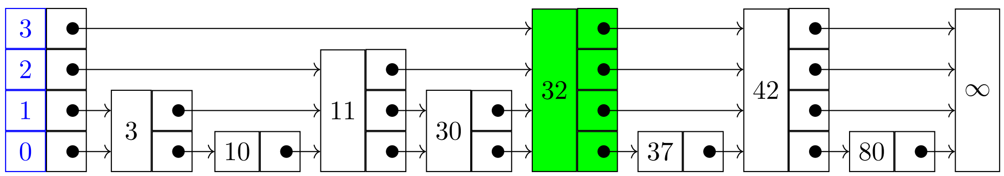
We have stored these pointers in our array A, so something like this, abstractly:

$$A = [30 \longrightarrow 37, 30 \longrightarrow 42, 11 \longrightarrow 42, H \longrightarrow 42]]$$

Then we make space for our new node:



And we insert and split up the pointers stored in the array:



7 Deletion Algorithm and Time Complexity

Deletion is a bit sneakier but the analysis is similar.

Without going into too much detail, we search for the target node but we treat it as if it only exists at level 0. As we are searching we record all the pointers which point to it although the only pointer which points to it that we actually follow is the one at level 0. The end result is that we obtain a list of pointers which point to the node.

We then delete the node and update the pointers accordingly.

Example 7.1. In the example above if we wished to delete 32 the process would go as follows:

1. Record but don't follow $H \longrightarrow 32$ on level 3.
2. Follow $H \longrightarrow 11$ on level 2.
3. Record but don't follow $11 \rightarrow 32$ on level 2.
4. Follow $11 \longrightarrow 30$ on level 2.
5. Record but don't follow $30 \rightarrow 32$ on level 1.
6. Record and follow $30 \rightarrow 32$ on level 0.

We store this:

$$A = [30 \longrightarrow 32, 30 \longrightarrow 32, 11 \longrightarrow 32, H \longrightarrow 32]$$

Then we splice these pointers with the pointers from 32 and we are done.