6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 12

Good morning. Today we're going to talk about it a balanced search structure, so a data structure that maintains a dynamic set subject to insertion, deletion, and search called skip lists. So, I'll call this a dynamic search structure because it's a data structure. It supports search, and it's dynamic, meaning insert and delete. So, what other dynamic search structures do we know, just for sake of comparison, and to wake everyone up? Shut them out, efficient, I should say, also good, logarithmic time per operation. So, this is a really easy question to get us off the ground.

You've seen them all in the last week, so it shouldn't be so hard. Treap, good. On the problems that we saw treaps. That's, in some sense, the simplest dynamic search structure you can get from first principles because all we needed was a bound on a randomly constructed binary search tree. And then treaps did well. So, that was sort of the first one you saw depending on when you did your problem set. What else?

Charles? Red black trees, good answer. So, that was exactly one week ago. I hope you still remember it. They have guaranteed log n performance. So, this was an expected bound. This was a worst-case order log n per operation, insert, delete, and search. And, there was one more for those who want to recitation on Friday: B trees, good. And, by B trees, I also include two-three trees, two-three-four trees, and all those guys. So, if B is a constant, or if you want your B trees knows a little bit cleverly, that these have guaranteed order log n performance, so, worst case, order log n. So, you should know this. These are all balanced search structures. They are dynamic. They support insertions and deletions. They support searches, finding a given key. And if you don't find the key, you find its predecessor and successor pretty easily in all of these structures.

If you want to augment some data structure, you should think about which one of these is easiest to augment, as in Monday's lecture. So, the question I want to pose to you is supposed I gave you all a laptop right now, which would be great. Then I asked you, in order to keep this laptop you have to implement one of these data structures, let's say, within this class hour. Do you think you could do it? How many people think you could do it? A couple people, a few people, OK, all front row people, good. I could probably do it. My preference would be B trees. They're sort of the simplest in my mind. This is without using the textbook. This would be a closed book exam. I don't have enough laptops to do it, unfortunately.

So, B trees are pretty reasonable. Deletion, you have to remember stealing from a sibling and whatnot. So, deletions are a bit tricky. Red black trees, I can never remember it. I'd have to look it up, or re-derive the three cases. treaps are a bit fancy. So, that would take a little while to remember exactly how those work. You'd have to solve your problem set again, if you don't have it memorized. Skip lists, on the other hand, are a data structure you will never forget, and something you can implement within an hour, no problem. I've made this claim a couple times before, and I always felt bad because I had never actually done it.

So, this morning, I implemented skip lists, and it took me ten minutes to implement a linked list, and 30 minutes to implement skip lists. And another 30 minutes debugging them. There you go. It can be done. Skip lists are really simple. And, at no point writing the code did I have to think, whereas every other structure I would have to think. There was one moment when I thought, ah, how do I flip a coin? That was the entire amount of thinking. So, skip lists are a randomized structure. Let's add in another adjective here, and let's also add in simple. So, we have a simple, efficient, dynamic, randomized search structure: all those things together. So, it's sort of like treaps and that the bound is only a randomized bound. But today, we're going to see a much stronger bound than an expectation bound.

So, in particular, skip lists will run in order log n expected time. So, the running time for each operation will be order log n in expectation. But, we're going to prove a much stronger result that their order log n, with high probability. So, this is a very strong claim. And it means that the running time of each operation, the running time of every operation is order log n almost always in a certain sense. Why don't I foreshadow that? So, it's something like, the probability that it's order log n is at least one minus one over some polynomial, and n.

And, you get to set the polynomial however large you like. So, what this basically means is that almost all the time, you take your skip lists, you do a polynomial number of operations on it, because presumably you are running a polynomial time algorithm that using this data structure. Do polynomial numbers of inserts, delete searches, every single one of them will take order log n time, almost guaranteed.

So this is a really strong bound on the tail of the distribution. The mean is order log n. That's not so exciting. But, in fact, almost all of the weight of this probability distribution is right around the log n, just tiny little epsilons, very tiny probabilities you could be bigger than log n. So that's where we are going. This is a data structure by Pugh] in 1989. This is the most recent.

Actually, no, sorry, treaps are more recent. They were like '93 or so, but a fairly recent data structure for just insert, delete, search. And, it's very simple. You can derive it if you don't know anything about data structures, well, almost nothing. Now, analyzing that the performance is log n, that, of course, takes our sophistication. But the data structure itself is very simple. We're going to start from scratch. Suppose you don't know what a red black tree is. You don't know what a B tree is. Suppose you don't even know what a tree is. What is the simplest data structure for storing a bunch of items for storing a dynamic set? A list, good, a linked list. Now, suppose that it's a sorted linked list. So, I'm going to be a little bit fancier there.

So, if you have a linked list of items, here it is, maybe we'll make it doubly linked just for kicks, how long does it take to search in a sorted linked list? Log n is one answer. n is the other answer. Which one is right? n is the right answer. So, even though it's sorted, we can't do binary search because we don't have random-access into a linked list. So, suppose I'm only given a pointer to the head. Otherwise, I'm assuming it's an array. So, in a sorted array you can search in log n. Sorted linked list: you've still got to scan through the darn thing. So, theta n, worst case search. Not so good, but if we just try to improve it a little bit, we will discover skip lists automatically.

So, this is our starting point: sorted linked lists, data n time. And, I'm not going to think too much about insertions and deletions for the moment. Let's just get search better, and then we'll worry about dates. Updates are where randomization will come in. Search: pretty easy idea. So, how can we make a linked list better? Suppose all we know about our linked lists. What can I do to make it faster? This is where you need a little bit of innovation, some creativity.

More links: that's a good idea. So, I do try to maybe add pointers to go a couple steps ahead. If I had log n pointers, I could do all powers of two ahead. That's a pretty good search structure. Some people use that; like, some peer-to-peer networks use that idea. But that's a little too fancy for me. Ah, good. You could try to build a tree on this linear structure. That's essentially where we're going. So, you could try to put pointers to, like, the middle of the list from the roots. So, you search between either here. You point to the median, so you can compare against the median, and know whether you should go in the first half or the second half that's definitely on the right track, also a bit too sophisticated. Another list: yes. Yes, good.

So, we are going to use two lists. That's sort of the next simplest thing you could do. OK, and as you suggested, we could maybe have pointers between them. So, maybe we have some elements down here, some of the elements up here. We want to have pointers between the lists. OK, it gets a little bit crazy in how exactly you might do that. But somehow, this feels good. So this is one linked list: L_1. This is another linked list: L_2.

And, to give you some inspiration, I want to give you, so let's play a game. The game is, what is this sequence? So, the sequence is 14. If you know the answer, shout it out. Anyone yet? OK, it's tricky. It's a bit of a small class, so I hope someone knows the answer. How many TA's know the answer? Just a couple, OK, if you're looking at the slides, probably you know the answer. That's cheating. OK, I'll give you a hint. It is not a mathematical sequence. This is a real-life sequence. Yeah?

Yeah, and what city? New York, yeah, this is the 7th Ave line. This is my favorite subway line in New York. But, what's a cool feature of the New York City subway? OK, it's a skip list. Good answer. [LAUGHTER] Indeed it is. Skip lists are so practical. They've been implemented in the subway system. How cool is that? OK, Boston subway is pretty cool because it's the oldest subway definitely in the United States, maybe in the world. New York is close, and it has other nice features like it's open 24 hours. That's a definite plus, but it also has this feature of express lines. So, it's a bit of an abstraction,

but the 7th Ave line has essentially two kinds of cars. These are street numbers by the way. This is, Penn Station, Times Square, and so on. So, there are essentially two lines. There's the express line which goes 14, to 34, to 42, to 72, to 96. And then, there's the local line which stops at every stop. And, they accomplish this with four sets of tracks. So, I mean, the express lines have their own dedicated track. If you want to go to stop 59 from, let's say, Penn Station, well, let's say from lower west side, you get on the express line.

You jump to 42 pretty quickly, and then you switch over to the local line, and go on to 59 or wherever I said I was going. OK, so this is express and local lines, and we can represent that with a couple of lists. We have one list, sure, we have one list on the bottom, so leave some space up here. This is the local line, L_2, 34, 42, 50, 59, 66, 72, 79, and so on. And then we had the express line on top, which only stops at

14, 34, 42, 72, and so on. I'm not going to redraw the whole list. You get the idea. And so, what we're going to do is put links between in the local and express lines, wherever they happen to meet.

And, that's our two linked list structure. So, that's what I actually meant what I was trying to draw some picture. Now, this has a property that in one list, the bottom list, every element occurs. And the top list just copies some of those elements. And we're going to preserve that property. So, L_2 stores all the elements, and L_1 stores some subset. And, it's still open which ones we should store. That's the one thing we need to think about. But, our inspiration is from the New York subway system. OK, there, that the idea. Of course, we're also going to use more than two lists. OK, we also have links. Let's say it links between equal keys in L_1 and L_2.

Good. So, just for the sake of completeness, and because we will need this later, let's talk about searches before we worry about how these lists are actually constructed. Of course, if I wanted that board. So, if you want to search for an element, x, what do you do? Well, this is the taking the subway algorithm. And, suppose you always start in the upper left corner of the subway system, if you're always in the lower west side, 14th St, and I don't know exactly where that is, but more or less, somewhere down at the bottom of Manhattan. And, you want to go to a particular station like 59.

Well, you'd stay on the express line as long as you can because it happens that we started on the express line. And then, you go down. And then you take the local line the rest of the way. That's clearly the right thing to do if you always start in the top left corner. So, I'm going to write that down in some kind of an algorithm because we will be generalizing it. It's pretty obvious at this point. It will remain obvious.

So, I want to walk right in the top list until that would go too far. So, you imagine giving someone directions on the subway system they've never been on. So, you say, OK, you start at 14th. Take the express line, and when you get to 72nd, you've gone too far. Go back one, and then go down to the local line. It's really annoying directions. But this is what an algorithm has to do because it's never taken the subway before. So, it's going to check, so let's do it here. So, suppose I'm aiming for 59. So, I started 14, say the first thing I do is go to 34. Then from there, I go to 42. Still good because 59 is bigger than 42. I go right again. I say, oops, 72 is too big. That was too far.

So, I go back to where it just was. Then I go down and then I keep going right until I find the element that I want, or discover that it's not in the bottom list because bottom list has everyone. So, that's the algorithm. Stop when going right would go too far, and you discover that with a comparison. Then you walk down to L_2. And then you walk right in L_2 until you find x, or you find something greater than x, in which case x is definitely not on your list.

And you found the predecessor and successor, which may be your goal. If you didn't find where x was, you should find where it would go if it were there, because then maybe you could insert there. We're going to use this algorithm in insertion. OK, but that search: pretty easy at this point. Now, what we haven't discussed is how fast the search algorithm is, and it depends, of course, which elements we're going to store in L_1, which subset of elements should go in L_1. Now, in the subway system, you probably put all the popular stations in L_1. But here, we want worst-case

performance. So, we don't have some probability distribution on the nodes. We just like every node to be accessed sort of as quickly as possible, uniformly.

So, we want to minimize the maximum time over all queries. So, any ideas what we should do with L_1? Should I put all the nodes of L_1 in the beginning? OK, it's a strict subset. Suppose I told you what the size of L_1 was. I can tell you, I could afford to build this many express stops. How should you distribute them among the elements of L_2? Uniformly, good. So, what nodes, sorry, what keys, let's say, go in L_1? Well, definitely the best thing to do is to spread them out uniformly, OK, which is definitely not what the 7th Ave line looks like.

But, let's imagine that we could reengineer everything. So, we're going to try to space these things out a little bit more. So, 34 and 42nd are way too close. We'll take a few more stops. And, now we can start to analyze things. OK, as a function of the length of L_1. So, the cost of a search is now roughly, so, I want a function of the length of L_1, and the length of L_2, which is all the elements, n.

What is the cost of the search if I spread out all the elements in L_1 uniformly? Yeah? Right, the total number of elements in the top lists, plus the division between the bottom and the top. So, I'll write the length of L_1 plus the length of L_2 divided by the length of L_1. OK, this is roughly, I mean, there's maybe a plus one or so here because in the worst case, I have to search through all of L_1 because the station I could be looking for could be the max. OK, and maybe I'm not lucky, and the max is not on the express line. So then, I have to go down to the local line. And how many stops will I have to go on the local line?

Well, L_1 just evenly partitions L_2. So this is the number of consecutive stations between two express stops. So, I take the express, possibly this long, but I take the local possibly this long. And, this is an L_2. And there is, plus, a constant, for example, go walking down. But that's basically the number of nodes that I visit. So, I'd like to minimize this function. Now, L_2, I'm going to call that n because that's the total number of elements. L_1, I can choose to be whatever I want. So, let's go over here.

So, I want to minimize L_1 plus n over L_1. And I get to choose L_1. Now, I could differentiate this, set it to zero, and go crazy. Or, I could realize that, I mean, that's not hard. But, that's a little bit too fancy for me. So, I could say, well, this is clearly best when L_1 is small. And this is clearly best when L_1 is large. So, there's a trade-off there. And, the trade-off will be roughly minimized up to constant factors when these two terms are equal. That's when I have pretty good balance between the two ends of the trade-off. So, this is up to constant factors. I can let L_1 equal n over L_1, OK, because at most I'm losing a factor of two there when they happen to be equal.

So now, I just solve this. This is really easy. This is (L_1)^2 equals n. So, L_1 is the square root of n. OK, so the cost that I'm getting over here, L_1 plus L_2 over L_1 is the square root of n plus n over root n, which is, again, root n. So, I get two root n. So, search cost, and I'm caring about the constant here, because it will matter in a moment. Two square root of n: I'm not caring about the additive constant, but the multiplicative constant I care about. OK, that seems good. We started with a linked list that searched in n time, theta n time per operation. Now we have two linked lists, search and theta root n time.

It seems pretty good. This is what the structure looks like. We have root n guys here. This is in the local line. And, we have one express stop which represents that. But we have another root n values in the local line. And we have one express stop that represents that. And these two are linked, and so on. Well, I should put some dot, dot, dots in there. OK, so each of these chunks has length root n, and the number of representatives up here is square root of n. The number of express stops is square root of n. So clearly, things are balanced now. I search for, at most, square root of n up here. Then I search in one of these lists for, at most, square root of n.

So, every search takes, at most, two root n. Cool, what should we do next? So, again, ignore insertions and deletions. I want to make searches faster because square root of n is not so hot as we know. Sorry? More lines. Let's add a super express line, or another linked list. OK, this was two. Why not do three? So, we started with a sorted linked list. Then we went to two. This gave us two square root of n. Now, I want three sorted linked lists. I didn't pluralize here. Any guesses what the running time might be? This is just guesswork.

Don't think. From two square root of n, you would go to, sorry? Two square root of two, fourth root of n? That's on the right track. Both the constant and the root change, but not quite so fancily. Three times the cubed root: good. Intuition is very helpful here. It doesn't matter what the right answer is. Use your intuition. You can prove that. It's not so hard. You now have three lists, and what you want to balance are at the length of the top list, the ratio between the top two lists, and the ratio between the bottom two lists. So, you want these three to multiply out to n, because the top times the ratio times the ratio: that has to equal n. And, so that's where you get the cubed root of n. Each of these should be equal. So, you set them because the cost is the sum of those three things.

So, you set each of them to cubed root of n, and there are three of them. OK, check it at home if you want to be more sure. Obviously, we want a few more. So, let's think about k sorted lists. k sorted lists will be k times the k'th root of n. You probably guessed that by now. So, what should we set k to? I don't want the exact minimum. What's a good value for k? Should I set it to n? n's kind of nice, because the n'th root of n is just one. Now that's n. So, this is why I cared about the lead constant because it's going to grow as I add more lists. What's the biggest reasonable value of k that I could use?

Log n, because I have a k out there. I certainly don't want to use more than log n. So, log n times the log n'th root, and this is a little hard to draw of n. Now, what is the log n'th root of n? That's what you're all thinking about. What is the log n'th root of n minus two? It's one of these good questions whose answer is? Oh man. Remember the definition of root? OK, the root is n to the one over log n. OK, good, remember the definition of having a power, A to the B? It was like two to the power, B log A? Does that sound familiar? So, this is two to the log n over log n, which is, I hope you can get it at this point, two.

Wow, so the log n'th root of n minus two is zero: my favorite answer. OK, this is to. So this whole thing is two log n: pretty nifty. So, you could be a little fancier and tweak this a little bit, but two log n is plenty good for me. We clearly don't want to use any more lists, but log n lists sounds pretty good. I get, now, logarithmic search time. Let's check. I mean, we sort of did this all intuitively. Let's draw what the list looks like. But, it will work. So, I'm going to redraw this example because you have to, also.

So, let's redesign that New York City subway system. And, I want you to leave three blank lines up here. So, you should have this memorized by now. But I don't. So, we are not allowed to change the local line, though it would be nice, add a few more stops there. OK, we can stop at 79th Street. That's enough. So now, we have log n lists. And here, log n is about four. So, I want to make a bunch of lists here. In particular, 14 will appear on all of them. So, why don't I draw those in?

And, the question is, which elements go in here? So, I have log n lists. And, my goal is to balance the number of items up here, and the ratio between these two lists, and the ratio between these two lists, and the ratio between these two lists. I want all these things to be balanced. There are log n of them. So, the product of all those ratios better be n, the number of elements down here. So, the product of all these ratios is n.

And there's log n of them; how big is each ratio? So, I'll call the ratio r. The ratio's r. I should have r to the power of log n equals n. What's r? What's r minus two? Zero. OK, this should be two to the power of log n. So, if the ratio between the number of elements here and here is to all the way down, then I will have an elements at the bottom, which is what I want. So, in other words, I want half the elements here, a quarter of the elements here, an eighth of the elements here, and so on. So, I'm going to take half of the elements evenly spaced out: 34th, 50th, 66th, 79th, and so on.

So, this is our new semi-express line: not terribly fast, but you save a factor of two for going up there. And, when you're done, you go down, and you walk, at most, one step. And you find what you're looking for. OK, and then we do the same thing over and over and over until we run out of elements. I can't read my own writing. It's 79th. OK, if I had a bigger example, I would be more levels, but this is just barely enough. Let's say two elements is where I stop. So, this looks good. Does this look like a structure you've seen before, at all, vaguely? Yes? A tree: yes. It looks a lot like a binary tree.

I'll just leave it at that. In your problem set, you'll understand why skip lists are really like trees. But it's more or less a tree. Let's say at this level, it looks sort of like binary search. You look at 14; you look at 15, and therefore, you decide whether you are in the left half for the right half. And that's sort of like a tree. It's not quite a tree because we have this element repeated all over. But more or less, this is a binary tree. At depth I, we have two to the I nodes, just like a tree, just like a balanced tree.

I'm going to call this structure an ideal skip list. And, if all we are doing our searches, ideal skip lists are pretty good. Maybe at practice: not quite as good as a binary search tree, but up to constant factors: just as good. So, for example, I mean, we can generalize search, just check that it's log n. So, the search procedure is you start at the top left. So, let's say we are looking for 72. You start at the top left. 14 is smaller than 72, so I try to go right. 79 is too big. So, I follow this arrow, but I say, oops, that's too much. So, instead, I go down 14 still. I go to the right: oh, 50, that's still smaller than 72: OK. I tried to go right again. Oh: 79, that's too big. That's no good. So, I go down. So, I get 50. I do the same thing over and over.

I try to go to the right: oh, 66, that's OK. Try to go to the right: oh, 79, that's too big. So I go down. Now I go to the right and, oh, 72: done. Otherwise, I'd go too far

and try to go down and say, oops, element must not be there. It's a very simple search algorithm: same as here except just remove the L_1 and L_2. Go right until that would go too far. Then go down. Then go right until we'd go too far, and then go down. You might have to do this log n times.

In each level, you're clearly only walking a couple of steps because the ratio between these two sizes is only two. So, this will cost two log n for search. Good, I mean, so that was to check because we were using intuition over here; a little bit shaky. So, this is an ideal skip list, we have to support insertions and deletions. As soon as we do an insert and delete, there's no way we're going to maintain the structure. It's a bit too special.

There is only one of these where everything is perfectly spaced out, and everything is beautiful. So, we can't do that. We're going to maintain roughly this structure as best we can. And, if anyone of you knows someone in New York City subway planning, you can tell them this. OK, so: skip lists. So, I mean, this is basically our data structure. You could use this as a starting point, but then you start using skip lists. And, we need to somehow implement insertions and deletions, and maintain roughly this structure well enough that the search still costs order log n time.

So, let's focus on insertions. If we do insertions right, it turns out deletions are really trivial. And again, this is all from first principles. We're not allowed to use anything fancy. But, it would be nice if we used some good chalk. This one looks better. So, suppose you want to insert an element, x. We said how to search for an element. So, how do we insert it? Well, the first thing we should do is figure out where it goes. So, we search for x. We call search of x to find where x fits in the bottom list, not just any list.

Pretty easy to find out where it fits in the top list. That takes, like, constant time. What we want to know: because the top list has constant length, we want to know where x goes in the bottom list. So, let's say we want to insert a search for 80. Well, it is a bit too big. Let search for 75. So, we'll find the 75 fits right here between 72 and 79 using the same path. OK, if it's there already, we complain because I'm going to assume all keys are distinct for now just so the picture stays simple.

But this works fine even if you are inserting the same key over and over. So, that seems good. One thing we should clearly do is insert x into the bottom list. We now know where it fits. It should go there. Because we want to maintain this invariant, that the bottom list contains all the elements. So, there we go. We've maintained the invariant. The bottom list contains all the elements. So, we search for 75. We say, oh, 75 goes here, and we just sort of link in 75. You know how to do a linked list, I hope. Let me just erase that pointer.

All the work in implementing skip lists is the linked list manipulation. Is that enough? No, it would be fine for now because now there's only a chain of length three here that you'd have to walk over if you're looking for something in this range. But if I just keep inserting 75, and 76, than 76 plus epsilon, 76 plus two epsilon, and so on, just pack a whole bunch of elements in here, this chain will get really long. Now, suddenly, things are not so balanced. If I do a search, I'll pay an arbitrarily long amount time here to search for someone.

If I insert k things, it'll take k time. I want it to stay log n. If I only insert log n items, it's OK for now. What I want to do is decide which of these lists contain 75.

So, clearly it goes on the bottom. Every element goes in the bottom. Should it go up a level? Maybe. It depends. It's not clear yet. If I insert a few items here, definitely some of them should go on the next level. Should I go to levels up? Maybe, but even less likely. So, what should I do? Yeah?

Right, so you maintain the ideal partition size, which may be like the length of this chain. And you see, well, if that gets too long, then I should split it in the middle, promote that guy up to the next level, and do the same thing up here. If this chain gets too long between two consecutive next level express stops, then I'll promote the middle guy. And that's what you'll do in your problem set. That's too fancy for me. I don't need no stinking counters.

What else could I do? I could try to maintain the ideal skip list structure. That will be too expensive. Like I say, 75 is the guy that gets promoted, and this guy gets demoted all the way down. But that will propagate everything to the right. And that could cost linear time for update. Other idea? If I only want half of them to go up, I could flip a coin. Good idea. All right, for that, I will give you a quarter. It's a good one. It's the old line state, Maryland. There you go. However, you have to perform some services for that quarter, namely, flip the coin.

Can you flip a coin? Good. What did you get? Tails, OK, that's the first random bit. But we are going to do is build a skip list. Maybe I should tell you how first. OK, but the idea is flip a coin. If it's heads, so, sorry, if it's heads, we will promote it to the next level, and flip again. So, this is an answer to the question, which other lists should store x? How many other lists should we add x to? Well, the algorithm is, flip a coin, and if it comes out heads, then promote x. to the next level up, and flip again.

OK, that's key because we might want this element to go arbitrarily high. But for starters, we flip a coin. It doesn't go to the next level. Well, we'd like it to go to the next level with probability one half because we want the ratio between these two sizes to be a half, or sorry, two, depending which way you take the ratio. So, I want roughly half the elements up here. So, I flip a coin. If it comes up heads, I go up here. This is a fair coin.

So I want it 50-50. OK, then how many should that element go up to the next level up? Well, with 50% probability again. So, I flip another point. If it comes up heads, I'll go up another level. And that will maintain the approximate ratio between these two guys as being two. The expected ratio will definitely be two, and so on, all the way up. If I go up to the top and flip a coin, it comes up heads, I'll make another level. This is the insertion algorithm: dead simple.

The fancier one you will see on your problem set. So, let's do it. OK, I also need someone to generate random numbers. Who can generate random numbers? Pseudo-random? I'll give you a quarter. I have one here. Here you go. That's a boring quarter. Who would like to generate random numbers? Someone volunteering someone else: that's a good way to do it. Here you go. You get a quarter, but you're not allowed to flip it. No randomness for you; well, OK, you can generate bits, and then compute a number. So, give me a number. 44, can answer. OK, we already flipped a coin and I got tails. Done. That's the insertion algorithm.

I'm going to make some more space actually, put it way down here. OK, so 44 does not get promoted because we got a tails. So, give me another number. Nine, OK, I

search for nine in this list. I should mention one other thing, sorry. I need a small change. This is just to make sure searches still work. So, the worry is suppose I insert something bigger and then I promote it. This would look very bad for a skip list data structure because I always want to start at the top left, and now there's no top left. So, just minor change: just let me remember that. The minor change is that I'm going to store a special value minus infinity in every list. So, minus infinity always gets promoted all the way to the top, whatever the top happens to be now.

So, initially, that way I'll always have a top left. Sorry, I forgot to mention that. So, initially I'll just have minus infinity. Then I insert 44. I say, OK, 44 goes there, no promotion, done. Now, we're going to insert nine. Nine goes here. So, minus infinity to nine, flip your coin, heads. Did he actually flip it? OK, good. He flipped it before, yeah, sure. I'm just giving you a hard time. So, we have nine up here. We need to maintain this minus infinity just to make sure it gets promoted along with everything else. So, that looks like a nice skip list. Flip it again. Tails, good. OK, so this looks like an ideal skip list. Isn't that great? It works every time. OK, give me another number. 26, OK, so I search for 26. 26 goes here.

It clearly goes on the bottom list. Here we go, 26, and then I you raised 44. Flip. Tails, OK, another number. 50, oh, a big one. It costs me a little while to search, and I get over here. 50. Flip. Heads, good. So 50 gets promoted. Flip it again. Tails, OK, still a reasonable number. Another number? 12, it takes a little while to get exciting here. OK, 12 goes here between nine and 26. You're giving me a hard time here. OK, flip. Heads, OK, 12 gets promoted. I know you have to work a little bit, but we just came here to search for 12.

So, we know that nine was the last point we went down. So, we promote 12. It gets inserted up here. We are just inserting into this particular linked list: nothing fancy. We link the two twelves together. It still looks kind of like a linked list. Flip again. OK, tails, another number. 37. Jeez. It's a good test of memory. 37, what was it, 44 and 50? And 50 was at the next level up. I think I should just keep appending elements and have you flip coins. OK, we just inserted 37. Tails. OK, that's getting to be a long chain. That looks a bit worse. OK, give me another number larger than 50. 51, good answer.

Thank you. OK, flip again. And again. Tails. Another number. Wait, someone else should pick a number. It's not working. What did you say? 52, good answer. Flip. Tails, not surprising. We've gotten a lot of heads there. OK, another number. 53, thank you. Flip. Heads, heads, OK. Heads, heads, you didn't flip. All right, 53, you get the idea. If you get two consecutive heads, then the guy goes up two levels. OK, now flip for real.

Heads. Finally. Heads we've been waiting for. If you flipped three heads in a row, you go three levels. And each time, we keep promoting minus infinity. Look again. Heads, oh my God. Where were they before? Flip again. It better be tails this time. Tails, good. OK, you get the idea. Eventually you run out of board space. Now, it's pretty rare that you go too high. What's the probability that you go higher than log n? Another easy log computation. Each time, I have a 50% probability of going up. One in n probability of going up log n levels because half to the power of log n is one out of n. So, it depends on n, but I'm not going to go too high.

And, intuitively, this is not so bad. So, these are skip lists. You have the ratios right in expectation, which is a pretty weak statement. This doesn't say anything about

the lengths of these change. But intuitively, it's pretty good. Let's say pretty good on average. So, I had two semi-random processes going on here. One is picking the numbers, and that, I don't want to assume anything about. The numbers could be adversarial. It could be sequential. It could be reverse sorted. It could be random. I don't know. So, it didn't matter what he said. At least, it shouldn't matter. I mean, it matters here. Don't worry. You're still loved. You still get your $0.25. But what the algorithm cares about is the outcomes of these coins. And the probability, the statement that this data structure is fast with high probability is only about the random coins.

Right, it doesn't matter what the adversary chooses for numbers as long as those coins are random, and the adversary doesn't know the coins. It doesn't know the outcomes of the coins. So, in that case, on average, overall of the coin flips, you should be OK. But the claim is not just that it's pretty good on average. But, it's really, really good almost always. OK, with really high probability it's log n. So, for example, with probability, one minus one over n, it's order of log n, with probability one minus one over n^2 it's log n, probability one minus one over n^100, it's order log n. All those statements are true for any value of 100. So, that's where we're going.

OK, I should mention, how do you delete in a skip list? Find the element. You delete it all the way. There's nothing fancy with delete. Because we have all these independent, random choices, all of these elements are sort of independent from each other. We don't really care. So, delete an element, just throw it away. The tricky part is insertion. When I insert an element, I'm just going to randomly see how high it should go. With probability one over two to the i, it will go to height i.

Good, that's my time. I've been having too much fun here. I've got to go a little bit faster, OK. So here's the theorem. Let's see exactly what we are proving first. With high probability, this is a formal notion which I will define a second. Every search in n elements skip lists costs order of log n. So, that's the theorem. Now I need to define with high probability. So, with high probability. And, it's a bit of a long phrase. So, often we will, and you can abbreviate it WHP. So, if I have a random event, and the random event here is that every search in an n element skip list costs order log n, I want to know what it means for that event E to occur with high probability.

So this is the definition. So, the statement is that for any alpha greater than or equal to one, there is a suitable choice of constants -- -- for which the event, E, occurs with this probability I keep mentioning. So, the probability at least one minus one over n to the alpha. So, this is a bit imprecise, but it will suffice for our purposes. If you want to really formal definition, you can read the lecture notes. There are special lecture notes for this lecture on the stellar site. And, there's the PowerPoint notes on the SMA site.

But, right, there's a bit of a subtlety in the choice of constants here. There is a choice of this constant. And there's a choice of this constant. And, these are related. And, there's alpha, which we get to whatever we want. But the bottom line is, we get to choose what probability we want this to be true. If I want it to be true, with probability one minus one over n^100, I can do that. I just sat alpha to a hundred, and up to this little constant that's going to grow much slower than n to the alpha.

I get the error probability. So this thing is called the error probability. The probability that I fail is polynomially small, for any polynomial I want. Now, with the same data

structure, right, I fixed the data structure. It doesn't depend on alpha. Anything you want, any alpha value you want, this data structure will take order of log n time. Now, this constant will depend on alpha. So, you know, you want error probability one over n^100 is probably going to be, like, 100 log n. It's still log n. OK, this is a very strong claim about the tale of the distribution of the running time of search, very strong. Let me give you an idea of how strong it is.

How many people know what Boole's inequality is? How many people know what the union bound is in probability? You should. It's in appendix c. Maybe you'll know it by the theorem. It's good to know it by name. It's sort of like linearity of expectations. It's a lot easier to communicate to someone. Linearity of expectations: instead of saying, you know that thing where you sum up all the expectations of things, and that's the expectation of the sum? It's a lot easier to say linearity of expectation. So, let me quiz you in a different way. So, if I take a bunch of events, and I take their union, either this happens or this happens, or so on. So, this is the inclusive OR of k events. And, instead, I look at the sum of the probabilities of those events.

OK, easy question: are these equal? No, unless they are independent. But can I say anything about them, any relation? Smaller, yeah. This is less than or equal to that. OK, this should be intuitive to you from a probability point of view. Look at the textbook. OK: very basic result, trivial result almost. What does this tell us? Well, suppose that E_i is some kind of error event. We don't want it to happen. OK, and suppose, mix some letters here. Suppose I have a bunch of events which occur with high probability. OK, call those E_i complement. So, suppose, so this is the end of that statement, E_i complement occurs with high probability.

OK, so then the probability of E_i is very small, polynomially small. One over n to the alpha for any alpha I want. Now, suppose I take a whole bunch of these events, and let's say that k is polynomial in n. So, I take a bunch of events, which I'd like to happen. They all occur with high probability. There is only polynomially many of them. So let's say, let me give this constant a name. Let's call it c.

Let's say I take n to the c such events. Well, what's the probability that all those events occur together? Because they should rest of the time occurred together because each one occurs most of the time, occurs with high probability. So, I want to look at E_1 bar intersect, E_2 bar, and so on. So, each of these occurs as high probability. What's the chance that they all occur? It's also with high probability. I'm changing the alpha. So, the union bound tells me the probability of any one of these failing, the probability of this failing, or this failing, or this failing, which is this thing, is, at most, the sum of the probabilities of each failure. These are the error probabilities. I know that each of them is, at most, one over n to the alpha, with a constant in front. If I add them all up, there's only n to the c of them. So, I take this error probability, and I multiply by n to the c.

So, I get like n to the c over n to the alpha, which is one over n to the alpha minus c. I can set alpha as big as I want. So, I said it much, much bigger than c, and this event occurs with high probability. I sort of made a mess here, but this event occurs with high probability because of this. Whatever the constant is here, however many events I'm taking, I just set alpha to be bigger than that.

And, this event will occur with high probability, too. So, when I say here that every search of cost order log n with high probability, not only do I mean that if you look at one search, it costs order log n with high probability. You look at another search, and

it costs log n with high probability. I mean, if you take every search, all of them take order log n time with high probability. So, this event that every single search you do takes order log n, is true with high probability estimate the number of searches you are doing is polynomial in n.

So, I'm assuming that I'm not using this data structure forever, just for a polynomial amount of time. But, who's got more than a polynomial amount of time anyway? This is MIT. So, hopefully that's clear. We'll see it a few more times. Yeah? The algorithm doesn't depend on Alpha. The question is how do you choose alpha in the algorithm. So, we don't need to. This is just sort of for an analysis tool. This is saying that the farther out you get, so you say, well, what's the probability that more than ten log n. Well, it's like one over n^10. Let's say it's linear. Well, what's the chance that you're more than 20 log n? Well that's one over n^20. So, the point is the tail of this distribution is getting a really small, really fast. And, such using alpha is more like sort of for your own feeling good. OK, you can set it to 100, and then n is at least two.

So, that's like one over 2^100 chance that you fail. That's damn small. If you've got a real random number generator, the chance that you're going to hit one over 2^200 is pretty tiny, right? So, let's say you set alpha to 256, which is always a good number. 2^256 is much bigger than the number of particles in the known universe, so, the light matter. So, actually I think this even accounts for some notion of dark matter. So, this is really, really, really big. So, the chance that you pick a random particle in the universe that happens to be your favorite particle, this one right here, that's over one over 2^256, or even smaller. So, set alpha to 256, the chance to your algorithm takes more than order log n time is a lot smaller than the chance that a meteor strikes your computer at the same time that it has a flooding point error, at the same time that the earth explodes because they're putting a transport through this part of the solar system at the same time, I mean, I could go on, right?

It's really, really unlikely that you are more than log n. And how unlikely: you get to choose. But it's just in the analysis the algorithm doesn't depend on it. It's the same algorithm, very cool. Sometimes, with high probability, bounds depends on alpha. I mean, the algorithm depends on alpha. But here, it will not. OK, away we go. So now you all understand the claim. So let's do a warm up. We will also need this fact. But it's pretty easy. The lemma is that with high probability, the number of levels in the skip list is order log n.

I think it's order log n, certainly. So, how do we prove that something happens with high probably? Compute the probability that it happened; show that it's high. Even if you don't know what high probability means, in fact, I used to ask that earlier on. So, let's compute the chance that it doesn't happen, the error probability, because that's just a one minus the cleaner. So, I'd like to say, let's say, that it's, at most, c log n levels. So, what's the error probability for that event? This is sort of an event. I'll put it in squiggles just for, all set. This is the probability that they are strictly greater than c log n levels. So, I want to say that that probability is particularly small, polynomially small.

Well, how do I make levels? When I insert an element, the probability half, it goes up. And, the number of levels in the skip list is the max over all the elements of how high it goes up. But, max, oh, that's a mess. All right, you can compute the expectation of the max if you have a bunch of unknown variables; there is

expectation there is a constant, and you take the max. It's like log in and expectation, but we want a much stronger statement.

And, we have this Boole's inequality that says I have a bunch of things, polynomially many things. Let's say we have n items. Each one independently, I don't even care if it's a dependent. If it goes up more than c log n, yeah, the number of levels is more than c log n. So, this is, at most, and then I want to know, do any of those events happen for any of the n elements? So, I just multiplied by n. It's certainly, at most, n times the probability that x gets promoted, this much here, greater than or equal to log n times.

OK, if I pick, for any element, x, because it's the same for each element. They are done independently. So, I'm just summing over x here, and that's just a factor of n. Clear? This is Boole's inequality. Now, what's the probability that x gets promoted c log n times? We did this before for log n. It was one over n. For c log n, it's one over n to the c. OK, this is n times two. Let's be nicer: one half to the power of c log n. One half to the power of c log n is one over two to the c log n. The log n comes out here, becomes an n. We get n to the c. So, this is n divided by n to the c, which is n to the c minus one.

And, I get to choose c to be whatever I want. So, I choose c minus one to be alpha. I think exactly that. Oh, sorry, one over n to the c minus one. Thank you. It better be small. This is an upper bound. So, probability is polynomially small. I get to choose, and this is a bit of the trik. I'm choosing this constant to be large, large enough for alpha. The point is, as c grows, alpha grows. Therefore, I can set alpha to be whatever I want, set c accordingly. So, there's a little bit more words that have to go here. But, they're in the notes. I can set alpha to be as large as I want. So, I can make this probability as small as I want in the polynomial sets. So, that's it. Number of levels, order log n: wasn't that easy?

Rules and equality, the point is that when you're dealing with high probability, use Boole's inequality. And, anything that's true for one element is true for all of them, just like that. Just lose a factor of n, but that's just one in the alpha, and alpha is big: big constant, but it's big. OK, so let's prove the theorem. High probability searches cost order log n. We now know the height is order log n, but it depends how balanced this thing is.

It depends how long the chains are to really know that a search costs log n. Just knowing a bound on the height is not enough, unlike a binary tree. So, we have one cool idea for this analysis. And it's called backwards analysis. So, normally you think of a search as starting in the top left corner going left and down until you get to the item that you're looking for. I'm going to look at the reverse process. You start at the item you're looking for, and you go left and up until you get to the top left corner. The number of steps in those two walks is the same. And, I'm not implementing an algorithm here, I'm just doing analysis. So, those are the same processes, just in reverse. So, here's what it looks like. You have a search, and it starts, which really means that it ends at a node in the bottom list.

Then, each time you visit a node in this search, you either go left or up. And, when do you go left or up? Well, it depends with the coin flip was. So, if the node wasn't promoted at this level. So, if it wasn't promoted higher, and that happened exactly when we got a tails. Then, we go left, which really means we came from the left. Or,

if we got a heads, so if this node was promoted to the next level, which happened whenever we got a heads at that particular moment.

This is in the past some time when we did the insertion. Then we go, or came from, up. And, we stop at the root. This is really where we start; same thing. So, either at the root or I'm also going to think of this as stopping at minus infinity. OK, that was a bit messy, but let me review. So, normally we start up here. Well, just looking at everything backwards, and in brackets is what's really happening. So, this search ends at the node you were looking for. It's always in the bottom list. Then it says, well, was this node promoted higher? If it was, I came from above. If not, I came to the left. It must have been in the bottom chain somewhere. OK, and that's true at every node you visit.

It depends whether that quite slipped heads or tails at the time that you inserted that node into that level. But, these are just a bunch of events. I'm just going to check, what is the probability that its heads, and what is the probability that a tails? It's always a half. Every time I look at a coin flip, when it was flipped, there was a probability of half going out of their way. That's the magic. And, I'm not using that these events are independent anyway.

For every element that I search for, for every value, x, that's another search. Those events may not be independent. I can still use Boole's inequality and conclude that all of them are order log n with high probability. As long as I can prove that any one event happens with high probability. So, I don't need independence between, I knew that these coin flips in a single search are independent, but everything else, for different searches I don't care.

So, how long can this process go on? We want to know how many times can I make this walk? Well, when I hit the root node, I'm done. Well, how quickly would I hit the root node? Well, with probability, a half, I go up each step. The number of times I go up is, at most, the number of levels minus one. And that's order log n with high probability. So, this is the only other idea. So, we are now improving this theorem. So, the number of up moves in a search, which are really down moves, but same thing, is less than the number of levels. Certainly, you can't go up more than there are levels in the search. And in insert, you can go arbitrarily high. But a search: as high as you can go.

And this is, at most, c log n with high probability. This is what we proved in the lemma. So, we have a bound on the number of up moves. Half of the moves, roughly, are going to be up moves. So, this pretty much down to the number of moves. Not quite. So, what this means is that with high probability, so this is the same probability, but I could choose that as high as I want by setting c large enough.

The number of moves, in other words, the cost of the search is at most the number of coin flips until we get c long n heads, right, because in every step of the search, I make a move, and then I flip another coin, conceptually. There is another independent coin lying there. And it's either heads or tails. Each of those is independent. So, how many independent coin flips does it take until I get c log n heads?

The claim is that that's order log n with high probability. But we need to prove that. So, this is a claim. So, if you just sit there with a coin, and you want to know how

many times does it take until I get c log n heads, the claim is that that number is order log n with high probability. As long as I prove that, I know that the total number of steps I make, which is the number of heads and tails is order log n because I definitely know the number of heads is, at most, c log n. The claim is that the number of tails can't be too much bigger. Notice, I can't just say c here. OK, it's really important that I have log n. Why? Because with high probability, it depends on n. This notion depends on n.

Log n: it's true. Anything bigger that log n: it's true, like n. If I put n here, this is also true. But, if I put a constant or a log log n, this is not true. It's really important that I have log n here because my notion of high probability depends on what's written here. OK, it's clear so far. We're almost done, which is good because I just ran out of time. Sorry, we're going to go a couple minutes over.

So, I want to compute the error probability here. So, I want to compute the probability that there is less than c log n heads. Let me skip this step. So, I will be approximate and say, what's the probability that there is, at most, c log n heads? So, I need to say how many coins we are flipping here for what this event is. So, I need to specify this constant. Let's say we flip ten c log n coins. Now I want to look at the error probability under that event. The probability that there is at most c log n heads among those ten c log n flips. So, the claim is this should be pretty small. It's going to depend on ten. Then I'll choose ten to be arbitrarily large, and I'll be done, OK, make my life a little bit easier.

Well, I would ask you normally, but this is 6.042 material. So, what's the probability that we have, at most, this many heads? Well, that means that nine c log n of the coins, because there are ten c log n flips, c log n heads at most, nine c log n at least better be tails. So this is the probability that all those other guys become tails, which is already pretty small. And then, there is this permutation thing. So, if I had exactly c log n heads, this would be the number of ways to rearrange c log n heads among ten c log n coin flips. OK, that's just the number of permutations. So, this is a bit big, which is kind of annoying. This is really, really small. The claim is this is much smaller than that is big.

So, this is just some math. I'm going to whiz through it. So, you don't have to stay too long. But you should go over it. You should know that y choose x is, at most, ey over x to the x, good fact. Therefore, this is, at most, ten c log n over c log n, also known as ten. These cancel. There's an e out here. And then I raise that to the c log n power. OK, then I divide by two to the power, nine c log n. OK, so what's this? This is e times ten to the c log n divided by two to the nine c log n.

OK, claim this is very big. This is not so big, because I have a nine here. So, let's work it out. This e times ten, that's a good number, we can put upstairs. So, we get log of e times ten, ten times, e, and then c log n. And then, we have over two to the nine c log n. So, we have this two to the c log n in both cases. So, this is two to the log, ten e minus nine, c, log n: some basic algebra. So, I'm going to set, not quite. This is one over two to the nine minus log: so, just inverting everything here, negating the sign in here. And, this is my alpha because the rest is n.

So, this is one over n to the alpha when alpha is this particular value: nine minus log of ten times e times c. It's a bit of a strange thing. But, the point is, as ten goes to infinity, nine here is the number one smaller than ten, right? We subtracted one somewhere along the way. So, as ten goes to infinity, this is basically, this is ten

minus one. This is log of ten times e. e doesn't really matter. The point is, this is logarithmic in ten. This is linear in ten. The thing that's linear in ten is much bigger than the thing that's logarithmic in ten. This is called abusive notation. OK, as ten goes to infinity, this goes to infinity, gets bigger. And, there is a c out here. But, for any value of c that you want, whatever value of c you wanted in that claim, I can make alpha arbitrarily large by changing the constant in the big O, which here was ten.

OK, so that claim is true with high probability. Whatever probability you want, which tells you alpha, you set a constant effort of the log N to be this number, which grows, and you're done. You get the claim that is order log N heads, order log N flips with the high probability, therefore. [None of the steps?] in the search is order log N with high probability. Really cool stuff; read the notes. Sorry I went so fast at the end.