Skip Lists

Step 1

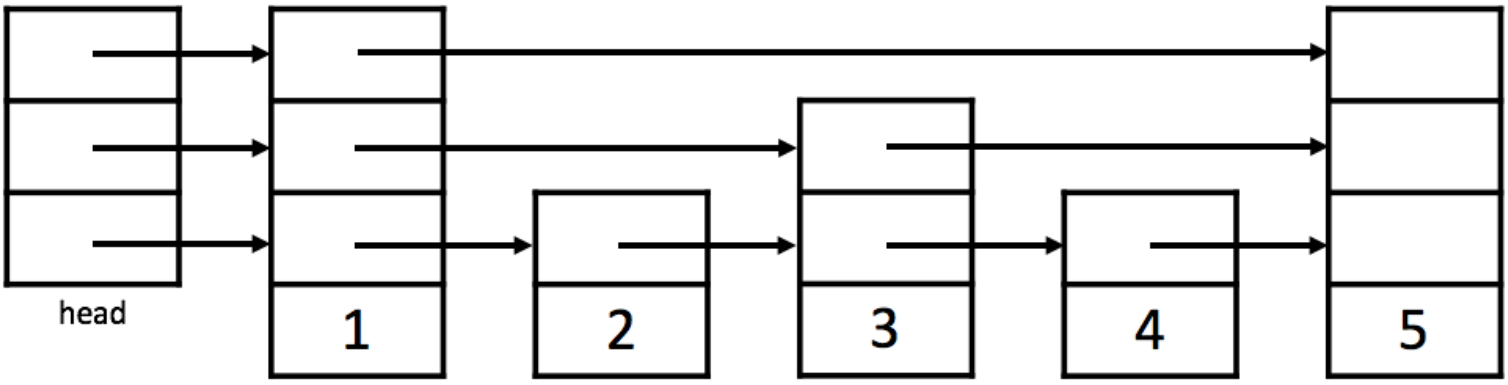
We have thus far discussed two basic data structures: the **Array List** and the **Linked List**. We saw that we could obtain worst-case O(log *n*) find operations with an **Array List** if we kept it sorted and used *binary search*, but insert and remove operations would always be O(*n*). Also, to avoid having to keep rebuilding the backing array, we had to allocate extra space, which was was wasteful. We also saw that, with a **Linked List**, we could obtain worst-case O(1) insert and remove operations to the front or back of the structure, but finding elements would be O(*n*), even if we were to keep the elements sorted. This is because **Linked Lists** lack the *random access* property that **Array Lists** have. Also, because each node in a **Linked List** is created on-the-fly, we don't have to waste extra space like we did with the **Array List**.

Is there any way for us to somehow reap the benefits of both data structures? In 1989, computer scientist William Pugh invented the **Skip List**, a data structure that expands on the **Linked List** and uses extra forward pointers with some random number generation to simulate the binary search algorithm achievable in **Array Lists**.

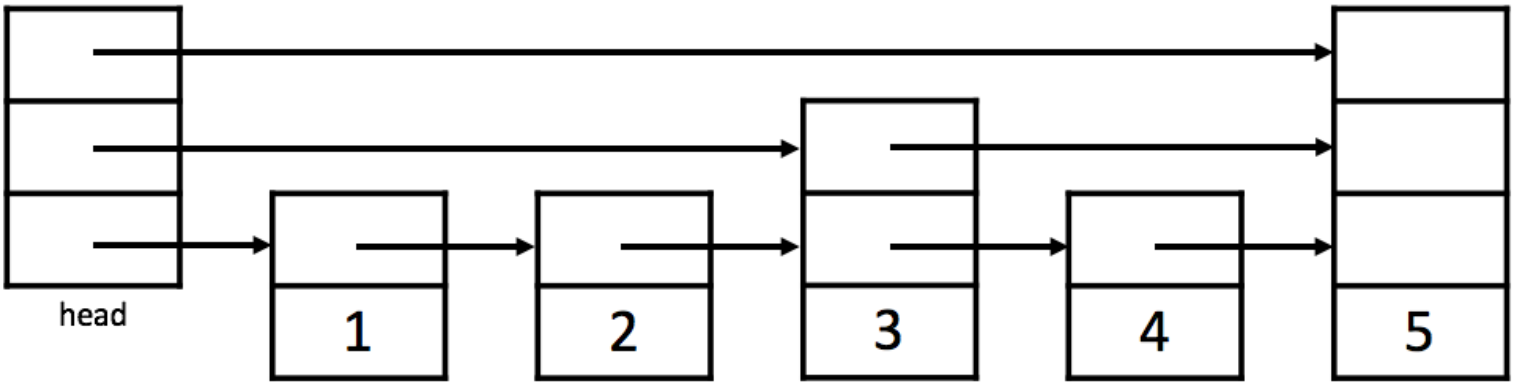
Figure: William Pugh (from his website)

Step 2

A **Skip List** is effectively the same as a **Linked List**, except every node in the **Skip List** has *multiple layers*, where each *layer* of a node is a forward pointer. For our purposes, we will denote the number of layers a node reaches as its *height*. The very bottom layer is exactly a regular **Linked List**, and each higher layer acts as an "express lane" for the layers below. Also, the elements in a **Skip List** must be **sorted**. The sorted property of the elements in a **Skip List** will let us perform a find algorithm that is functionally similar to binary search. Below is an example of a **Skip List**:



Just like with a **Linked List**, we have a *head*. However, because **Skip Lists** have *multiple* layers, the *head* also has multiple layers. Specifically, for each layer *i*, the *i*-th pointer in *head* points to the first node that has a height of *i*. In the example above, the first node (1) happens to reach every layer, so each pointer in *head* points to 1, but this does not have to be the case. Take the example below:



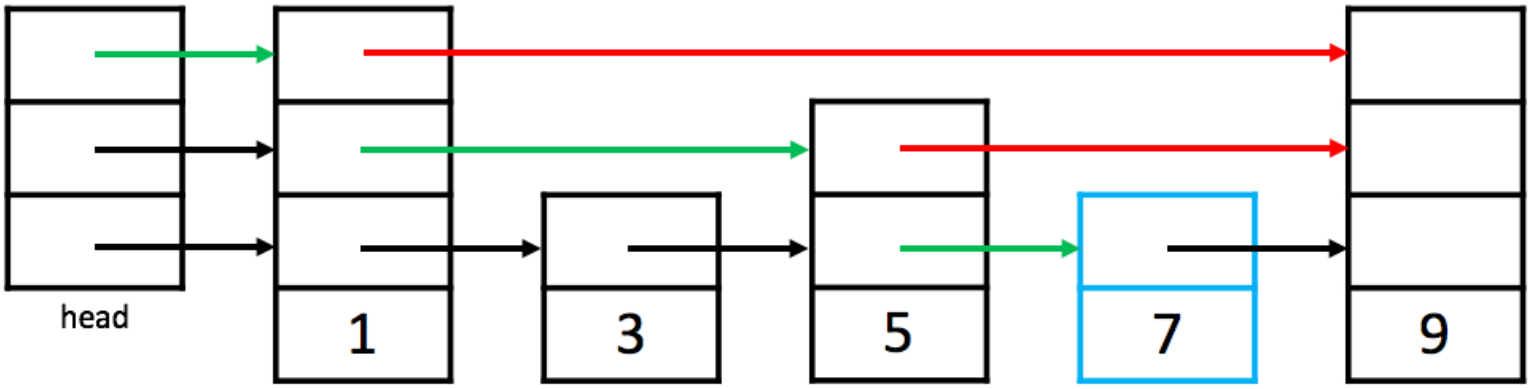
Now, the three pointers in *head* point to three different nodes. For our purposes, we will number the bottom layer as layer 0, the next layer as layer 1, etc. In this example, *head*'s pointer in layer 0 points to node 1, its pointer in layer 1 points to node 3, and its pointer in layer 2 points to node 5.

In this example, nodes 1, 2, and 4 have heights of 0, node 3 has a height of 1, and node 5 has a height of 2.

Step 3

To **find** an element *e* in a **Skip List**, we start our list traversal at *head*, and we start at the highest layer. When we are on a given layer *i*, we follow forward pointers on layer *i* until *just before* we reach a node that is larger than *e* **or**, until there are no more forward pointers on level *i*. Once we reach this point, we move down one layer and continue the search. If *e* exists in our **Skip List**, we will eventually find it (because the bottom layer is a regular **Linked List**, so we would step through the elements one-by-one until we find *e*), or if we reach a point where we want to move down one layer but we're already on layer 0, *e* does not exist in the **Skip List**.

Below is an example in which we attempt to find the element 7. **Red** arrows denote pointers that we *could* have traversed, but would have taken us to a node too big (so we instead chose to go down 1 level), and **green** arrows denote pointers we actually took.

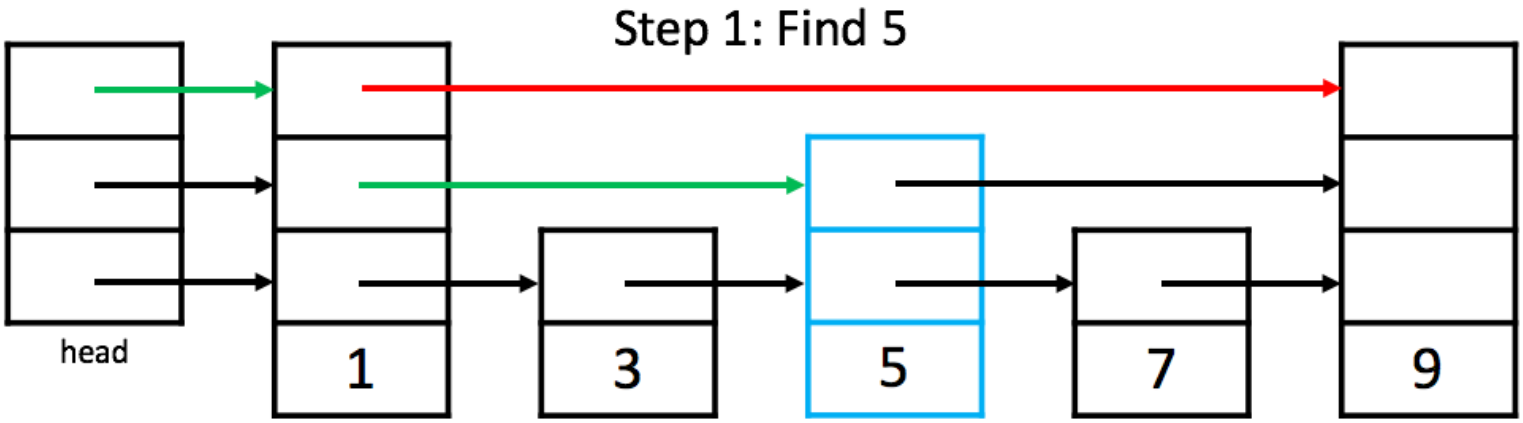


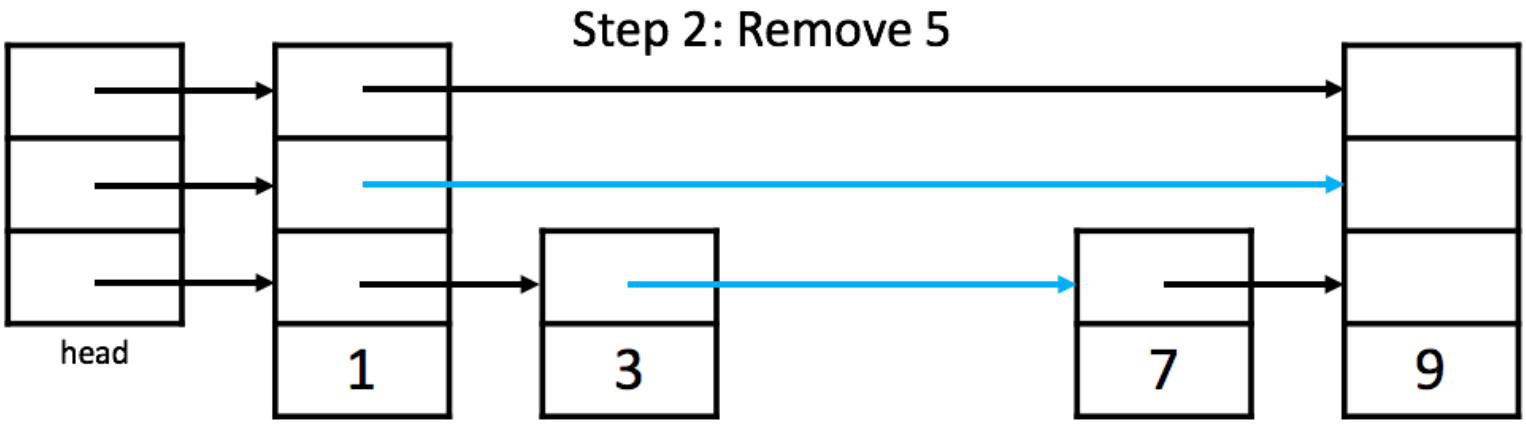
Below is formal pseudocode to describe the "find" algorithm. In the pseudocode, `head` is *head* and `head.height` is the highest layer in *head* (which is the highest layer in the **Skip List**, by definition). Also, for a given node `current`, `current.next` is a list of forward-pointers, where `current.next[i]` is the forward-pointer at layer *i*.

```
find(element): // returns True if element exists in Skip List, otherwise returns False
    current = head
    layer = head.height
    while layer >= 0:           // can't go lower than layer 0
        if current.next[layer] is NULL or current.next[layer] > element:
            layer = layer - 1   // drop one layer if we can't go further
        else:
            current = current[layer].next
        if current == element: // return True if we found element
            return True
    return False                // we failed on every layer, so element does not exist
```

Step 4

To **remove** an element from a **Skip List**, we simply perform the "find" algorithm to find the node we wish to remove, which we will call *node*. Then, for each layer *i* that *node* reaches, whatever is *pointing to node* on layer *i* should instead point to whatever *node points to* on layer *i*. Below is an example in which we remove 5 from the given **Skip List**. In the "Step 1: Find" figure, **red** arrows denote pointers that we *could* have traversed, but would have taken us to a node too big (so we instead chose to go down 1 level), and **green** arrows denote pointers we actually took. In the "Step 2: Remove" figure, **blue** arrows denote pointers that were updated to actually perform the removal.





Below is formal pseudocode to describe the "remove" algorithm. In the pseudocode, `head` is *head* and `head.height` is highest layer in *head* (which is the highest layer in the **Skip List**, by definition). Also, for a given node `current`, `current.next` is a list of forward-pointers, where `current.next[i]` is the forward-pointer at layer *i*, and `current.prev` is a list of reverse-pointers, where `current.prev[i]` is a pointer to the node that points to `current` on layer *i*.

```
remove(element): // removes element if it exists in the list
    current = head
    layer = head.height
    while layer >= 0:           // can't go lower than layer 0
        if current.next[layer] is NULL or current.next[layer] > element:
            layer = layer - 1   // drop one layer if we can't go further
        else:
            current = current[layer].next
        if current == element: // if we found element, break so we can fix pointers
            break
    if layer >= 0:              // if we found element, fix pointers
        for i from 0 to layer:
            current.prev[i].next[i] = current.next[i]
```

Step 5

EXERCISE BREAK: What is the *worst*-case time complexity to find or remove elements from a **Skip List**?

To solve this problem please visit https://stepik.org/lesson/30029/step/5

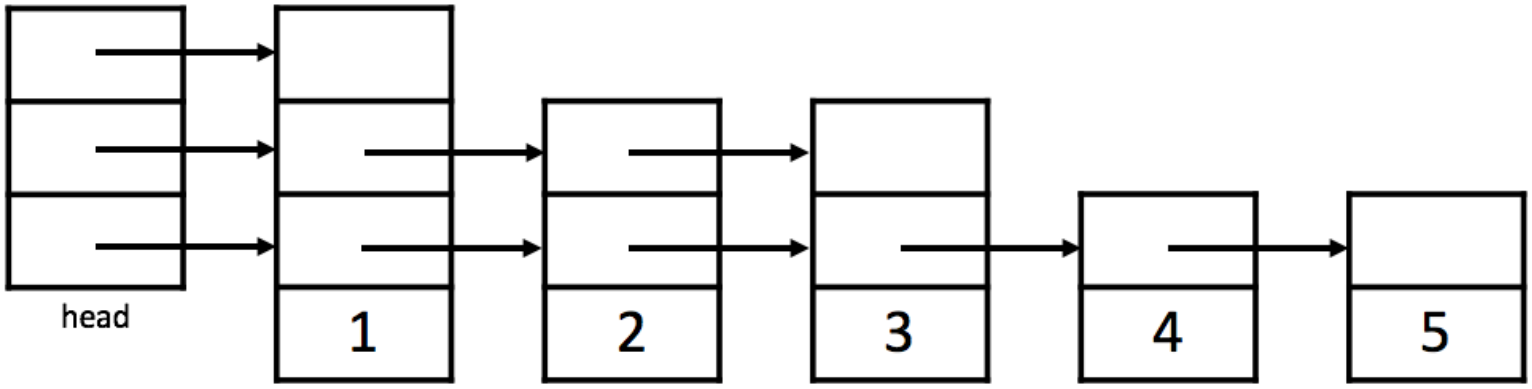
Step 6

EXERCISE BREAK: Assuming the heights of the nodes in a **Skip List** are optimally-distributed (i.e., each "jump" allows you to traverse half of the remainder of the list), what is the time complexity to find or remove elements from a **Skip List**?

To solve this problem please visit https://stepik.org/lesson/30029/step/6

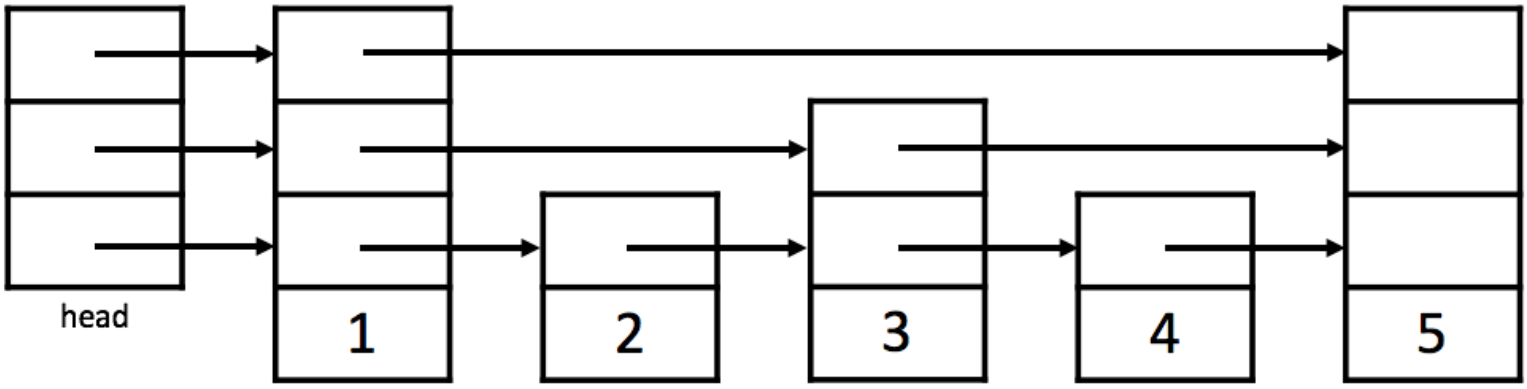
Step 7

Hopefully the previous questions made you think about how the distribution of *heights* in a **Skip List** affects the *performance* of the **Skip List**. To emphasize this thought, look at the following **Skip List**, and notice how using it is no faster than using an ordinary **Linked List** because we have to potentially iterate over all *n* elements in the worst case:



For example, finding node 5 would require 5 node traversals and finding node 3 would require 3 node traversals.

However, this next example of a **Skip List** has a better distribution of heights, meaning we check less elements to find the one we want:



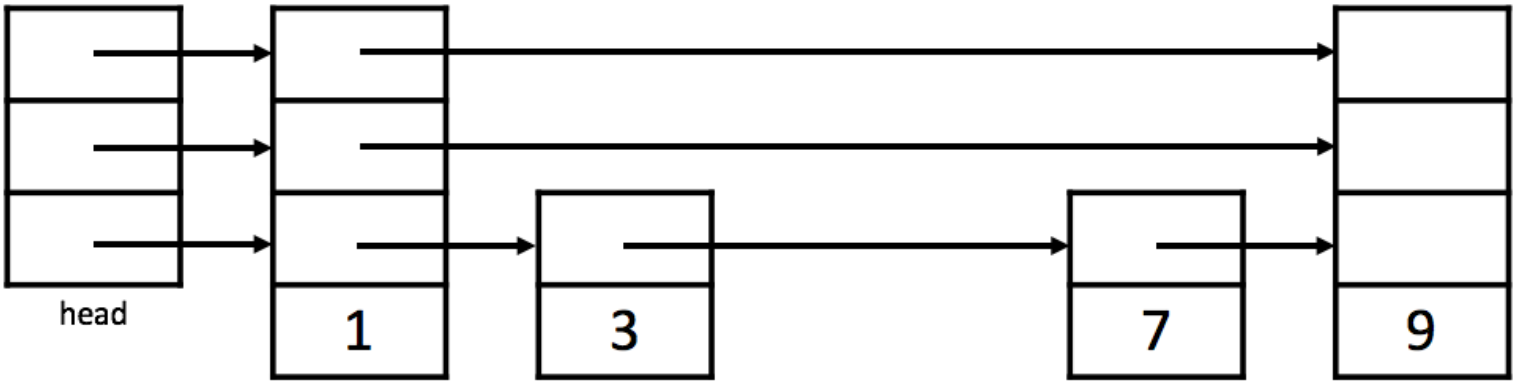
For example, finding node 5 would require 1 node traversal and finding node 3 would require 2 node traversals.

Clearly, the distribution of heights has a significant impact on the performance of a **Skip List**. With the best distribution of heights, the **Skip List** "find" algorithm effectively performs binary search, resulting in a O(log *n*) time complexity. Thus, we must ask ourselves the following question: How can we design our **Skip List** such that heights are automatically "well-distributed"?

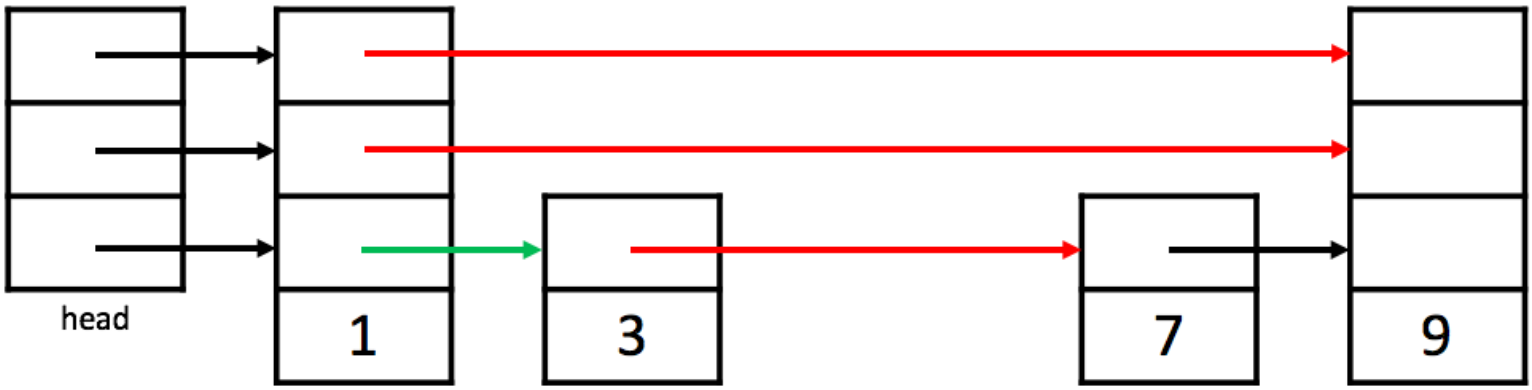
Step 8

The task of optimizing the distribution of heights in a **Skip List** falls under the **insert** operation. We do so by first performing the regular "find" algorithm to find where we will insert our new element. Then, we determine our new node's height. By definition, the new node must have a height of at least 0 (i.e., the 0-th layer). So how many layers higher should we build the new node? To answer this question, we will play a simple coin-flip game (we will explain in the next step what this game formally represents): starting at our base height of 0, we flip a coin, where the coin's probability of heads is *p*. If we flip heads, we increase our height by 1. If we flip tails, we stop playing the game and keep our current height.

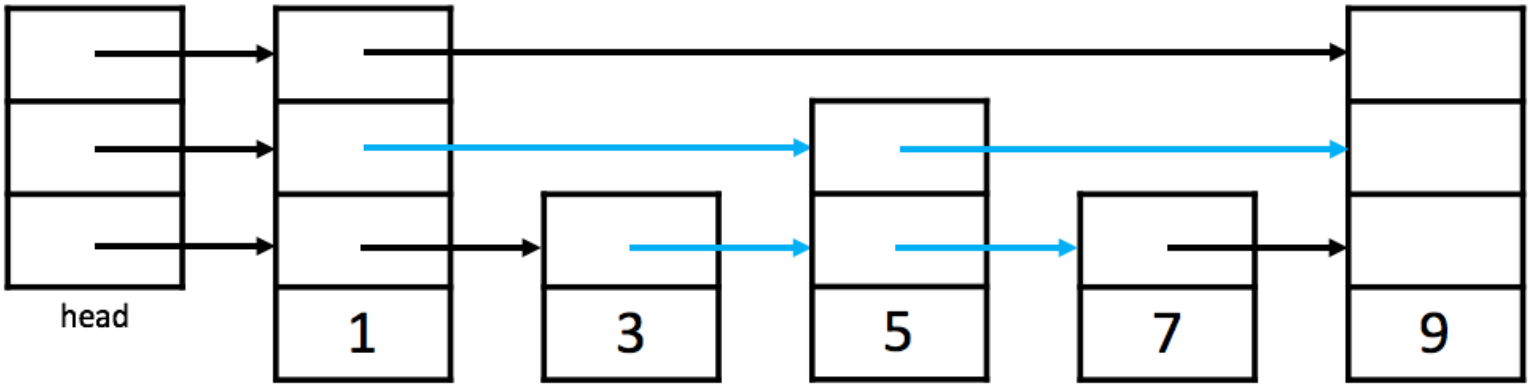
Say, for example, we wish to insert 5 into the following **Skip List**:



First, we perform the regular "find" algorithm to find the insertion site for 5. In the figure below, **red** arrows denote pointers that we *could* have traversed, but would have taken us to a node too big (so we instead chose to go down 1 level), and **green** arrows denote pointers we actually took. As you hopefully inferred, the **red** arrows are the only ones that we might have to fix upon insertion.



Now that we have found our insertion site, we must determine the height of our new node. We know that the coin must have a height of at least 0, so we start our height at 0. Then, we flip a coin where the probability of heads is p. Let's say we flipped heads (with probability p): we now increase our height from 0 to 1. Then, we flip the coin again. This time, let's say we flipped tails (with probability $1-p$): we stop playing the game and keep our height of 1. We perform the insertion with this new node by simply updating two pointers: one on layer 0 and one on layer 1.



Below is formal pseudocode to describe the "remove" algorithm. In the pseudocode, `head` is *head* and `head.height` is highest layer in *head* (which is the highest layer in the **Skip List**, by definition). Also, for a given node `current`, `current.next` is a list of forward-pointers, where `current.next[i]` is the forward-pointer at layer *i*. Lastly, note that the probability of coin-flip success, *p*, must be a variable defined in the **Skip List** itself (we refer to it as p below).

```
insert(element): // inserts element if it doesn't exist in the list
    current = head
    layer = head.height
    toFix = empty list of nodes of length head.height + 1 (one slot for each layer)
    while layer >= 0:                   // can't go lower than layer 0
        if current.next[layer] is NULL or current.next[layer] > element:
            toFix[layer] = current      // we might have to fix a pointer here
            layer = layer - 1           // drop one layer if we can't go further
        else:
            current = current[layer].next
        if current == element:          // if we found element, return (no duplicates)
            return

    // if we reached here, we can perform the insertion
    newNode = new node containing element, starting with height = 0
    while newNode.height < head.height: // can't go higher than head node's height
        result = result of coin-flip with probability p of heads
        if result is heads:
            height = height + 1
    for i from 0 to newNode.height:     // fix pointers
        newNode.next[i] = toFix[i].next[i]
        toFix[i].next[i] = newNode
```

Step 9

For those of you who are interested in probability and statistics, it turns out that we don't have to do multiple coin-flips: we can determine the height of a new node in a single trial. A coin-flip is a **Bernoulli distribution** (or a **binary distribution**), which is just the formal name for a probability distribution in which we only have two possible outcomes: *success* and *failure* (which we often call *heads* and *tails*, *yes* and *no*, etc.). We say that *p* is the probability of *success*, and we say that *q*, which equals 1–*p*, is the probability of *failure*.

What we described above, where we perform multiple coin-flips until we get our first tails, is synonymous to saying "Sample from a Bernoulli distribution until you see the first failure." It turns out that this statement is actually a probability distribution in itself: the **Geometric distribution**. The Geometric distribution tells us, given a Bernoulli distribution with probability *p* of *success*, what is the probability that our *k*-th trial (i.e., our *k*-th coin-flip) results in the *last failure in a row before* the *first success*? Formally, for a Geometric random variable *X*, we say that $\Pr(X = k) = (1 - p)^k p$: we need *k failures* (each with probability 1–*p*), and then we need one *success* (with probability *p*).

But wait! The Geometric distribution tells us about the number of flips until the first *success*, but we want the number of flips until the first *failure*! It turns out that we can trivially modify what we just said to make the general definition of a Geometric distribution work for us. Recall that *success* and *failure* are just two events, where the probability of *success* is *p* and the probability of *failure* is *q* = 1–*p*. If we simply swap them when we plug them into the Geometric distribution, we will have swapped *success* and *failure*, so the expected value of the Geometric distribution would tell us the expected number of trials until the first *failure*. Formally, $\Pr(X = k) = p^k(1 - p)$: we need *k successes* (each with probability *p*), and then we need one *failure* (with probability 1–*p*).

Therefore, if we were to sample from a Geometric distribution following $\Pr(X = k) = p^k(1 - p)$, we would get a random value for *k*, the number of coin-flips it *before* the first *failure*, which is exactly our new node's height!

If this probability and statistics detour didn't make much sense to you, that's perfectly fine. Performing separate individual coin-flips, each with probability *p* of success, vs. performing a single trial from the Geometric distribution above are both completely mathematically equivalent. It's just an interesting connection we wanted to make for those of you who were interested.

Step 10

EXERCISE BREAK: To determine the heights of new nodes, you use a coin with a probability of success of p = 0.3. What is the probability that a new node will have a height of 0? (Enter your answer as a decimal rounded to the nearest thousandth)

To solve this problem please visit https://stepik.org/lesson/30029/step/10

Step 11

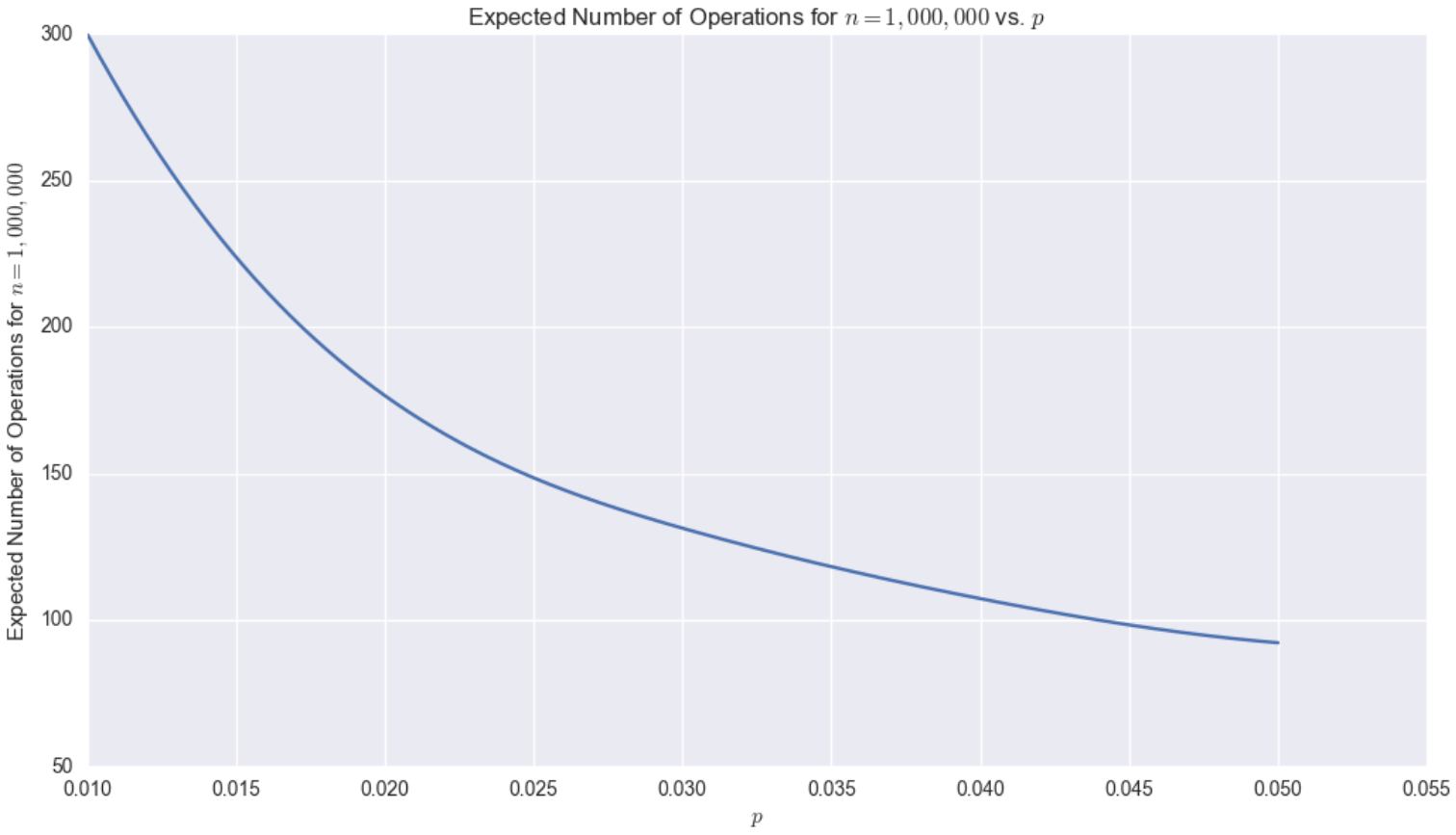
EXERCISE BREAK: To determine the heights of new nodes, you use a coin with a probability of success of p = 0.3. What is the probability that a new node will have a height of 2? (Enter your answer as a decimal rounded to the nearest thousandth)

To solve this problem please visit https://stepik.org/lesson/30029/step/11

Step 12

Of course, as we mentioned before, the *worst*-case time complexity to find, insert, or remove an element in a **Skip List** is O(n): if the heights of the nodes in the list are distributed poorly, we will effectively have a regular **Linked List**, and we will have to iterate through the elements of the list one-by-one. However, it can be formally proven (via a proof that is a bit out-of-scope) that, with "smart choices" for p and the maximum height of any given node, the expected number of comparisons that must be done for a "find" operation is $1 + \frac{1}{p}\log_{\frac{1}{p}} n + \frac{1}{1-p}$, meaning that the **average-case** time complexity of a **Skip List** is **O(log *n*)**.

The term "smart choices" is vague, so we will attempt to be a bit more specific. In the paragraph above, we mentioned that the expected number of comparisons that must be done for a "find" operation is $1 + \frac{1}{p}\log_{\frac{1}{p}} n + \frac{1}{1-p}$. However, as p increases, the amount of space we need to use to store pointers also increases. Therefore, when we pick a value for p to use in our **Skip List**, assuming we have a ballpark idea of how large n will be, we should choose the smallest value of p that results in a reasonable expected number of operations. For example, below is a plot of $y = \frac{1}{p}\log_{\frac{1}{p}} n$ for n = 1,000,000:



The curve begins to flatten at around *p* = 0.025, so that might be a good value to pick for *p*. Once we've chosen a value for *p*, it can be formally proven (via a proof that is a bit out-of-scope) that, for good performance, the maximum height of the **Skip List** should be no smaller than $\log_{\frac{1}{p}} n$.

Step 13

EXERCISE BREAK: Imagine you are implementing a **Skip List**, and you chose a value for *p* = 0.1. If you are expecting that you will be inserting roughly *n* = 1,000 elements, what should you pick as your **Skip List's** maximum height?

To solve this problem please visit https://stepik.org/lesson/30029/step/13

Step 14

In summation, we have now learned about the **Skip List**, a data structure based off of the **Linked List** that uses random number generation to mimic the binary search algorithm of a sorted **Array List** in order to achieve an **average-case** time complexity of **O(log *n*)**. Also, we learned that, in order to achieve this average-case time complexity, we need to be smart about choosing a value of *p*: if *p* is too small or too big, we will effectively just have a regular **Linked List**, and as *p* grows, the memory usage of our **Skip List** grows with it. Once we have chosen a value of *p*, we should choose the maximum height of our **Skip List** to be roughly $\log_{\frac{1}{p}} n$.

In this section, we discussed a modified **Linked List** data structure that mimicked some properties of an **Array List**, and in the next section, we will do the reverse: we will discuss a modified **Array List** data structure that mimics some properties of a **Linked List**: the **Circular Array**.