In this unit, we are going to look at some randomized data structures. We’ll go into more detail on this later, but it is important to recognize up front what is random and what is not. In a randomized data structure, like any other data structure, the contents are not chosen randomly; they are picked by whoever is using the data structure. What is going to be random is the way those items are organized internally, the links and ordering of the user’s input to facilitate efficient retrieval. We will see how choosing this structure randomly gives the same expected behavior as the more complicated data structures you’ve already learned about.

1 Fun with probability distributions

This unit is going to do a lot of big-O running time analysis with randomized data structures. Let’s start by reviewing some probability tools that will come into play. Actually, you’ve already been using some of these ideas in your problems - now let’s write them down and give them some names.

**Disjunction**: A *disjunction* of two events $A$ and $B$, written $A \cup B$, means the event where either $A$ or $B$ (or both) happens. It’s an or operation with probabilities, and you can write down the probability using the inclusion-exclusion principle as follows:

$$\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$$

This corresponds to what you see if you look at a Venn diagram.

Frequently, we would rather not try and compute the intersection probability $\Pr[A \cap B]$. In that case, we can get an upper-bound estimate by just ignoring that part of the equation. And then it extends trivially to any number of events as well:

**Union Bound**

$$\Pr[A_1 \cup A_2 \cup \cdots \cup A_n] \leq \Pr[A_1] + \Pr[A_2] + \cdots + \Pr[A_n]$$

I know, it seems pretty obvious at first glance, but it will be useful especially when the events $A_1, \ldots, A_n$ are all pretty rare and we just need to say “none of these things will (probably) happen.”

**Conjunction**: This is the and operation with probabilities, written $\Pr[A \cap B]$. There’s not always a simple formula for this, but there is when $A$ and $B$ are independent events. In probability, “independence” just means that one event has no influence on the other one. For example, if a couple has two children, then (as far as I know) the gender of each is an independent random event. If one sibling is a girl, it doesn’t make the other sibling any more or less likely to be a girl. But other genetic traits - maybe hair color for example - are not independent. You are more likely to have brown hair if you have a sibling that also has brown hair.

OK, so here’s the formula for conjunction of independent events. Again, it’s pretty simple; the main thing is to remember *not* to apply it if the events are not actually independent!
Conjunction of independent events

\[ \Pr[A \cap B] = \Pr[A] \Pr[B] \]

if \( A \) and \( B \) are independent events.

**PDF, CDF, CCDF.** We’ve talked a bit so far about probability distributions, which is the way the total realm of possibility is divided up among the possible values of a random variable.

There are three functions that can be used to describe a probability distribution. They are, the probability distribution function (PDF), the cumulative distribution function (CDF), and the complementary cumulative distribution function (CCDF). Formal definitions are as follows:

**Distribution functions**

If \( X \) is a random variable, its distribution is defined by any one of the three functions:

- **PDF:** \( f(k) = \Pr[X = k] \)
- **CDF:** \( f(k) = \Pr[X \leq k] \)
- **CCDF:** \( f(k) = \Pr[X > k] \)

The PDF is the probability that \( X \) is exactly equal to some value. This is the most obvious and basic of the three. The CDF and CCDF are obviously related to each other (each equals 1 minus the other), and the crucial difference is the inequality (\( \leq \) or \( > \)) in their definitions. For some complicated probability distributions that come up with randomized data structures, it’s easier to get an inequality like this than it is to get the exact PDF.

To illustrate what this means, here is a table showing the PDF, CDF, and CCDF for the random variable \( X \) that represents the sum of 2 fair 6-sided dice rolls:

<table>
<thead>
<tr>
<th>( k )</th>
<th>PDF((k))</th>
<th>CDF((k))</th>
<th>CCDF((k))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1/36</td>
<td>1/36</td>
<td>35/36</td>
</tr>
<tr>
<td>3</td>
<td>2/36</td>
<td>3/36</td>
<td>33/36</td>
</tr>
<tr>
<td>4</td>
<td>3/36</td>
<td>6/36</td>
<td>30/36</td>
</tr>
<tr>
<td>5</td>
<td>4/36</td>
<td>10/36</td>
<td>26/36</td>
</tr>
<tr>
<td>6</td>
<td>5/36</td>
<td>15/36</td>
<td>21/36</td>
</tr>
<tr>
<td>7</td>
<td>6/36</td>
<td>21/36</td>
<td>15/36</td>
</tr>
<tr>
<td>8</td>
<td>5/36</td>
<td>26/36</td>
<td>10/36</td>
</tr>
<tr>
<td>9</td>
<td>4/36</td>
<td>30/36</td>
<td>6/36</td>
</tr>
<tr>
<td>10</td>
<td>3/36</td>
<td>33/36</td>
<td>3/36</td>
</tr>
<tr>
<td>11</td>
<td>2/36</td>
<td>35/36</td>
<td>1/36</td>
</tr>
<tr>
<td>12</td>
<td>1/36</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

2  

**2  Expected Value and Probability**

(Wikipedia)

We saw expected value defined in Unit 1, and it’s going to become more and more important as we go along. If you read in the textbook or on most online sources, you’ll see a bunch of discussion about random variables. We’ll try to avoid the formalisms and complications to the extent possible here though.

Remember that expected value is, intuitively, just the average of some quantity weighted according to the probabilities of each quantity value occurring. Take lottery winnings as a simple example:

Consider a very simple lottery, where each ticket costs 1 dollar, there is one winner who gets \( J \) dollars, and
the chances of winning are one in a million. We want to decide how large $J$ needs to be before it’s worth it to play. This will correspond to the expected value of the payoff. Here there are two possibilities: with probability $\frac{999999}{1000000}$ we will lose 1 dollar, and with probability $\frac{1}{1000000}$ we will win $J - 1$ dollars. Therefore the expected payoff is

$$E[\text{winnings}] = \frac{999999}{1000000}(-1) + \frac{1}{1000000}(J - 1) = J \cdot 10^{-6} - 1$$

You could solve this equation for an expected value of 0 to discover that, if the jackpot is worth at least a million dollars, we expect to gain money on average by playing.

OK, so in this simple example that’s not so surprising. But the same idea holds more generally. For example, there might be some intermediate wins in the lottery with lower payoffs but higher probability, or we might account for the probability of a split jackpot, etc. The point is that expected value is a useful tool to make decisions in the face of uncertainty, when we know the probabilities of each thing occurring.

The definition of expected value connects directly with the definition of a PDF:

**PDF to Expected Value**

If $X$ is a nonnegative integer-valued random variable, then

$$E[X] = \sum_{i=1}^{\infty} \Pr[X = i] \cdot i.$$  

Sometimes those sums can be a little annoying to compute because of the multiplication by $i$, and it’s easier to use the CCDF instead. (For fun, see if you can work out the proof!)

**CCDF to Expected Value**

If $X$ is a nonnegative integer-valued random variable, then

$$E[X] = \sum_{i=0}^{\infty} \Pr[X > i].$$

Sometimes we have to deal with multiple random variables - corresponding to multiple events that we’re interested in. When those random variables get combined in complicated ways, it can be difficult to work out the resulting expectation.

But if they’re just added up, it’s easy! (Can you prove this one?)

**Linearity of expectation**

For any random variables $X_1, X_2, \ldots, X_n$,

$$E[X_1 + X_2 + \ldots + X_n] = E[X_1] + E[X_2] + \cdots + E[X_n]$$

In other words, the expected value of the sum of a bunch of things, is exactly the same as the sum of the expected values. This means that, if the expected value of my winnings after playing the lottery once is even as small as $0.01$, then if I play 1000 times the expected winnings go up to $10$. If I could afford to play a million times, well then the expected winnings go up substantially.

If this all seems way to obvious, consider (as we will) the expected value of the maximum. You might be tempted to say that $E[\max(X_1, X_2)] = \max(E[X_1], E[X_2])$. Not so! Consider the height of U.S. males. The average height of adult males is 5’10”, or 70 inches. That means that the expected height of a single randomly-chosen adult male is exactly 70 inches.
But what if I choose 100 U.S. males randomly, and consider the maximum height among those 100? Since they all have the same expected height, the maximum expected height among them is exactly 70 inches. But would you actually expect the tallest among those 100 to be 70 inches? I think not. We would need more information to determine what the expected value of the maximum is precisely, but certainly it is greater than the maximum of the expected values. More on this later.

One last thing on expected value: sometimes you need to connect expectation back to probability. For example, consider two gambling games. In the first, I bet you a dollar that a coin flip comes up heads. It costs me a dollar to play, and if it’s heads, I get two dollars back. So the expected value here would be 1 dollar.

The second gambling game is a lottery where you spend one dollar to play, and there is a million-dollar jackpot which you have a one-in-a-million chance of winning. (Otherwise you get nothing.) The expected value is again 1 dollar.

You can see that these two situations have very different probability distributions. The first one is uniformly distributed, and the second is definitely not; there’s a much greater chance to win nothing.

So we need a way to connect expected value back to probability, but because very different distributions can have the same expected value, it’s not going to be a precise connection. The Markov Bound below is a way of saying that the actual value of a (nonnegative) random variable is probably not too much larger than its expected value:

**Markov Inequality**

For any nonnegative random variable $X$ and any $a \geq 0$,

$$\Pr [X > a \cdot E[X]] \leq \frac{1}{a}.$$

For example, if you let $a = 10$, then this tells you that in any gambling game with an expected payout of $1, the probability of winning at least $10 is less than 10%. It’s important to emphasize that this depends on nothing other than the expected value - in particular, it would apply to both the lottery and the coin-flipping games above.

### 3 Skip Lists

- Wikipedia
- Article by the inventor

Skip lists are given excellent coverage in your textbook and online, so I won’t go into as much detail in this section of the notes. We’ll just cover the highlights.

Consider a linked list. It’s a wonderfully simple data structure, efficient in storage, blazingly fast to iterate through, and easy to implement. However, there’s a big disadvantage: accessing the middle. In a normal linked list, there’s no way to get to the middle of the list other than to go through half of the list, which is not very fast.

Hence the skip list. The idea is to start with a simple linked list (this will be “level 0”), and then build up more linked lists on top, each of which contains a subset of the nodes below it. So the higher-level linked lists “skip” over some items in the lower-level linked lists. Extending this idea up gives us a lot of the simplicity of linked lists, with the added ability to be able to quickly jump to any spot.

Oh, and one more thing: Skip lists are always ordered, and there are always two special values $-\infty$ and $+\infty$ at the ends. Now behold the skip list:

Some terminology:
This skip list six items and we write \( n = 6 \).

The 6 keys are 33, 51, 68, 71, 92, and 93. There may be some values associated with each item as well, but we haven’t drawn that because it doesn’t affect the way the data structure works.

Each item has a tower in the skip list, with varying heights. The height of the tower for 33 is 0, the height of the tower for 51 is 1, and the height of the tower for 71 is 2.

The height of the skip list is the height of the infinity towers, which is one more than the height of the tallest item tower. The skip list above has height 3.

The skip list consists of linked lists at different levels. In the example above, level 0 has all the items, level 1 has only 3, level 2 has just 71, and level 3 doesn’t have anything in it. The highest level in any skip list is always empty.

Each tower consists of some number of nodes, according to the height of the tower. The total size of the skip list is simply the number of nodes in it. The example above has size 10 (not counting the infinity towers on the ends).

In practice, the towers are usually stored as a single struct, with a list of forward links corresponding to the tower’s height.

We usually use skip lists to implement the dictionary ADT, supporting the search, insert, and delete operations. We’ll talk more about insert and delete later, but first let’s cover how searching is performed in a skip list, since that’s kind of the whole point!

**Skip List Search**

To search for key \( k \) in a skip list \( S \), do the following:

1. Start with \( v \) as the top-left most node in \( S \)
2. If the node to the right of \( v \) has key less than \( k \), then set \( v \) its neighbor to the right.
3. Else if \( v \) is not at level 0, set \( v \) to the node below it.
4. Else, we are done. Either the key of \( v \) is equal to \( k \) or \( k \) is not in \( S \).
5. Go back to step 2 and repeat until step 4 is triggered.

For example, to search for 80 in the skip list above, we start at the top-left infinity node, then move down to level 2, then right to 71 on level 2, then down to 71 on level 1, and finally down to 71 on level 0. Now the node following \( v \) has key larger than \( k \) (92 > 80), and we can’t go down past level 0, so we conclude that 80 is not in this skip list.

### 4 Randomized Data Structures and Skip-Insert

To insert in a skip list, you first choose the height of the new node to be inserted, then insert it where it goes at each level up to that height.

You can see the details on the insertion part from my code in the problems. The interesting thing here is actually the height choosing algorithm:

```python
def chooseHeight():
    ...
```
h = 0
while randomBit() != 1:
    h = h + 1
return h

Pretty simple, huh? Basically, we randomly choose bits until we get a 1, and then the number of zeros that were chosen is the height of the new node that will be inserted.

5 Skip List Analysis

The question is, what is the expected height of a new node? Remember that expected height means each possible height gets multiplied by the probability of getting that height, then these are all added up to give the weighted sum. Here’s the formula:

\[ E[h] = \frac{1}{2} \cdot 0 + \frac{1}{4} \cdot 1 + \frac{1}{8} \cdot 2 + \cdots = \sum_{i=0}^{\infty} \frac{i}{2^{i+1}} \]

This is a pretty standard summation that appears on the much-beloved theoretical computer science cheat sheet and the sum comes to (drum roll..) 1. So the expected height of a skip-list node is 1.

We also showed in class how this does NOT imply the expected height of the entire skip list is 1. This is because the expected value of the maximum (maximum height of any item in the skip list) is NOT the same as the maximum expected value! In fact, the expected height of a skip list is \( O(\log n) \), which we went over in class.

To analyze the cost of search is a little more tricky. Basically, you do it backwards. Starting with the node that gets searched for, trace the search path backwards to the top-left node in the skip list. If we define \( T(h) \) as the expected length of a search path in a height-\( h \) skip list, we get the following recurrence:

\[
T(n) \leq \begin{cases} 
0, & n = 0 \\
1 + T(h - 1), & \text{if last leg is vertical} \\
1 + T(h), & \text{if last leg is horizontal}
\end{cases}
\]

The big question is whether the last leg of the search is vertical or horizontal. Well, it just depends on whether the height of the last tower is greater than 0. But we know how the tower height was chosen - and the probability that it’s greater than 0 is exactly \( \frac{1}{2} \). So the \( n > 0 \) case of the recurrence becomes:

\[ T(h) \leq \frac{1}{2} (1 + T(h - 1)) + \frac{1}{2} (1 + T(h)) \]

Multiply both sides by 2 and subtract \( T(h) \) from both sides, and we get:

\[ T(h) \leq 2 + T(h - 1) \]

which means \( T(h) \in \Theta(h) \). In other words, the expected cost of search is the same (up to a constant) as the expected height, which we already know is \( \Theta(\log n) \). This is the same as AVL trees! Hooray!
6 Random BSTs

- Great reference: notes by Conrado Martinez

A randomized binary search tree is just like a regular BST, except that we make random choices on insertion in order to keep it balanced. The way it works specifically is that, when we insert a new node into a tree that already has \( n - 1 \) nodes, the new node will become the root with probability exactly \( \frac{1}{n} \). In other words, every time a new node is inserted, there is a one out of (number of nodes in the tree) chance that it becomes the new root.

We went through a number of examples in the class. The key is that this algorithm for insertion gets applied recursively at every level down the tree. So when we insert a new node, it has a small probability of becoming the new root. But then it’s recursively inserted into one of the two subtrees, and it has a less-small probability of being the root of that subtree. And so on down to the bottom.

Something really cool happens when we consider the probability that any element is the root of the final tree. Say we have the \( k \)th inserted item. It had probability \( \frac{1}{k} \) of being the root when it was inserted. Then in order for it to stay being the root, the next element must not have been chosen to be the root: this happens with probability \( \frac{k}{k+1} \). Then the next one must not have been the root either; the probability there is \( \frac{k+1}{k+2} \). Multiply all these out and you get the probability that the \( k \)th item is the root equals:

\[
\frac{1}{k} \cdot \frac{k}{k+1} \cdot \frac{k+1}{k+2} \cdots \frac{n-1}{n} = \frac{1}{n}
\]

So every single element has equal probability of being the root. Awesome! When you apply this recursively down the tree, you see that the expected height of the tree is \( \Theta(\log n) \). So this is a way of getting the same balance as an AVL tree or 2-3 tree, but by using randomness to do it instead of all those annoying tree rotations and extra complications.

7 Review: Hash functions and hash tables

- Wikipedia: Hash table

Hash tables are all about using mathematical functions to implement lookup dictionaries faster than the linked structures such as AVL trees or Skip Lists, at the cost of potentially using extra memory.

There are two main drawbacks to hash tables, which we won’t dwell on, but which you should be aware of. First, iterating through the elements of a hash table in order is going to be very slow - it would require you to write out all the elements and then sort them. Since this is usually very easy to do in other dictionary data structures, it might be a reason not to choose hash tables if you need to iterate through the data with some high frequency.

The other drawback is the difficulty in choosing good hash functions and getting all the details right with the size, dealing with collisions, and so on. These difficulties are often ignored when teaching hash tables, to make things simpler. So of course we’re going to drive right at them!

We will however make the following assumptions at all times:

- All the keys that are being hashed are integers. There is a lot of interesting theory in how to map other things like arrays or strings to integers so that they can be hashed. We will not talk about that step much here.
- Any collision resolution is done by separate chaining with linked lists, unless otherwise stated.

Our typical set-up for the hash table problem will be as follows: Each key is an integer from 0 up to \( m - 1 \), where \( m \) might be very large. This is usually called the “universe”. Out of this size-\( m \) universe, \( n \) distinct keys are to be stored in the hash table. The size (a.k.a. number of “buckets” or “slots”) in the hash table is \( s \). The hash function \( h(x) \) maps integers from the universe down to the range of hash table slots \( \{0, 1, 2, 3, \ldots, s - 1\} \).
Typically, the goal will be linear size (so that $s \in O(n)$) and $O(1)$-time to look up, insert, or delete something from the hash table.

8 Max bucket size

In a hash table that uses separate chaining, the size of the largest bucket determines the worst-case cost of a single lookup, insert, or delete operation. So it makes sense to investigate this quantity as the most important one in measuring the performance of a hashing scheme.

To begin, assume that we have $s = 2n$ and $h(x)$ behaves totally randomly. Even though hash functions are never randomized (you would never be able to look up the thing you just stored!), we often use the same techniques from randomized analysis to analyze their performance.

(Actually, a little later, we will use random numbers to choose a hash function, but still random numbers are never used by the hash function to compute its output.)

Problem 76 looks at the “nuts and bolts” of the computation to show that, with probability at least $1/2$, the largest bucket in this hash table will have

$$\Theta\left(\frac{\log n}{\log \log n}\right)$$

elements in it. In cat, the expected size of the largest bucket has that same asymptotic behavior as well.

Now the question becomes “so what?”. Should we be happy with this max bucket size? Well, it means that the worst-case cost of all of the dictionary operations will be $\Theta((\log n)/(\log \log n))$, and since $\log \log n$ is a very slow-growing function, the worst case cost is very close to $O(\log n)$. That’s the same as AVL trees or Skip Lists! So this is really not very good; we haven’t gained anything over the much simpler, and much smaller, tree-like dictionary structures.

And remember, this is the very best kind of hash function we could imagine, one that behaves just like random. If we don’t choose such a great hash function, the behavior could be even worse. Fortunately, there are some pretty simple ways to make this work better!

9 2-Choice Hashing

- Wikipedia: 2-choice hashing

As we will say, the key to many improvements on hashing is: use more than one hash function. This approach is not an instant solution; there must be some clever, “intelligent” way to make use of more hash functions. And it also brings in the difficult challenge of choosing multiple hash functions that have absolutely no relationship to each other (mathematically).

With that in mind, let’s look at a simple way of using two hash functions to improve the performance of a regular hash table. The basic idea is this: we will have two hash functions $h_1(x)$ and $h_2(x)$, each of which works the same (mapping integers from the entire universe into the range $\{0, 1, 2, \ldots, s - 1\}$ of hash table indices). Then element $x$ can either be stored at position $h_1(x)$ or position $h_2(x)$. Specifically, here is how the two most important hash table functions would work:

- **Insertion**: To insert $x$ into a 2-choice hash table, we compute the two hash values $h_1(x)$ and $h_2(x)$. Then, whichever of those two buckets has a smaller size determines which bucket $x$ will be inserted into.
- **Search**: To search for $x$ in a 2-choice hash table, just look through both buckets at $h_1(x)$ and $h_2(x)$, and return the location if $x$ is found in either bucket.
Once again, the performance will depend on the size of the largest bucket in this hash table. And again, we could get some very “unlucky” situations where all the elements inserted happen to hash to just a small number of buckets, which will then all be relatively large in size, destroying the performance of the hash table.

But now that there are two choices, and since we’re assuming the hash functions have no correlation between them, the chance of this happening is greatly reduced. Think about it: even if we happen to pick a bunch of insertions whose \(h_1(x)\) values are all the same, their \(h_2(x)\) values should still be different, since the two are unrelated, and then everything will be fine.

What difference does this make? Well, it would seem logical to assume that, since there are 2 hash functions, the expected max bucket size is maybe half as large as what it would be with a single hash function. But actually, it’s much, much better than that. The expected size of the largest bucket, in 2-choice hashing, turns out to be only

\[
\Theta(\log \log n)
\]

This is a massive improvement from the \(\Theta((\log n)/(\log \log n))\) of regular hashing. In fact, it’s like improving a \(\Theta(n)\)-cost algorithm to \(\Theta(\log n)\). Like I said, massive! This should be surprising, that we get such a tremendous improvement with only 2 hash functions instead of 1. This idea gets used in a number of different places in randomized algorithms, so much so that it even has a name: the power of two choices.

Now you might think, if 2 hash functions gives that much improvement, 3 hash functions must be even better! Unfortunately, the answer here is negative. The improvement from 2-choice to 3-choice hashing is marginal, and the improvement from 3-choice to 4-choice is even less, and so on. Plus, every time we add another hash function, that increases the time of doing every operation, since you have to actually compute all those hash values, look through all those buckets, and so on. This is in fact the lesson of the “empirical study” in Problem 68: 2-choice hashing gives the best balance between bucket size and number of hash functions, both in theory and in practice.

10 Storing a Set

- Wikipedia: Set ADT

So far we have looked at hash tables as a way to implement the Dictionary ADT that supports the operations of search, insert, and delete.

But in many applications, we don’t actually need all this functionality. For example, a password checker might be interested in whether your new password is the same as any of your previous passwords. This program doesn’t need to know what your previous passwords are, it just needs to check whether this new password is among those previously entered.

What we are talking about here is a set ADT that supports just two operations:

- **Insert**: Add \(x\) to the set, if it’s not already there.
- **Membership**: Test whether or not \(x\) is already in the set.

The important thing to notice here is that there is no way to actually retrieve an element of the set, once it has been inserted. We can only test whether a given element is present or not. So unlike in a dictionary, in a set we can get away with (potentially) smaller storage. Let’s see how.

The first idea is to use a normal hash table, but instead of storing buckets at each position, we will just store true/false boolean values. So when a new element \(x\) is inserted, we compute \(h(x)\) and set that table position to “true”, and to check membership we simply look up the true/false value stored in the hash value’s position in the table.
Now consider the costs: since there are no buckets, every entry in the hash table just has $O(1)$ size. Since it’s also $O(1)$ to compute the actual hash function, every operation on this kind of a hashed set is $O(1)$ in the worst case. Hooray!

Except that there’s a caveat: we can get wrong answers. As you know, the problem with all hash tables is collisions, when two different elements hash to the same value. So what happens if $h(x) = h(y)$, and we insert $x$, and then test $y$ for membership? Well, since $x$ was inserted, the hash table value at index $h(x)$ will be “true”, and since $h(y)$ has the same hash value, the membership test for $y$ will incorrectly return “true” as well.

This is called a false positive and it’s going to cause problems with this particular idea for storing a set. In particular, the birthday problem (sometimes called “birthday paradox”) tells us that the size must be at least $s \in \Theta(n^2)$ in order to guarantee a small probability of a collision. But that’s ridiculous - any savings we would have over a regular hash table (since we don’t have to store entire buckets) would be immediately wiped out by this huge table size requirement. So far, it seems that regular hash tables are still the best way to implement the set ADT.

11 Bloom filters

- Wikipedia: Bloom filter

But wait! We know a trick to improve the performance of hashing algorithms: use multiple hash functions. But we will have to change the way things work for sets. Just using the idea of 2-choice hashing directly doesn’t even make sense any more, since there are no buckets whose sizes we can compare!

Instead, we will use $k$ different hash functions $h_1(x), h_2(x), \ldots, h_k(x)$, and on each insertion, we set every one of these hash table entries to “true”. Membership tests will then involve looking up every one of the hash table indices in the table, and checking whether they are all true.

Why is this any better than using a single hash function? We’ll see the answer mathematically in a moment, but intuitively the reason is that we are now storing patterns instead of individual entries. Think back to the birthday problem: why is it so likely that 2 people will share the same birthday, even with a relatively small group of people compared to the number of possible birthdays? It’s because the number of combinations of two people can be much larger than the actual number of people. And it’s this number of potential pairs of individuals that makes the birthday problem work the way it does.

So we are now applying this same paradox in our favor to build a more compact data structure! The number of distinct hash values is going to be relatively small, resulting in a large number of collisions. But the number of combinations of hash values can still be very large, so if we use more hash functions, this number of combinations increases, and the number of collisions for that same combination should decrease correspondingly.

This data structure is called a Bloom filter, and its performance depends critically on two parameters: the number of hash functions $k$, and the size of the table $s$.

To analyze how big we want these parameters to be, we need some math of course! Here’s an overview of how we’ll do this: First, we figure out the probability of a membership test returning a false positive. This will be a formula in terms of $k$ and $s$. Next, we set that probability to a constant value (assuming we want a small, constant failure probability), and figure out the optimal values of $k$ and $s$ to achieve that failure rate.

Step 1: Probability of a table entry set to “true”

The probability that any given entry in the table is set to “true” is the chance that any of the $k$ hash functions, for any of the $n$ insertions, returned that index. As is often the case, the easiest way to compute this probability is actually to do the opposite: compute the chance that a given entry is not set to “true”. This means the probability that all of those $nk$ hash function evaluations returned something other than the given entry. Since there are $s$ possibilities for each hash function value, this works out to:
Pr[.table entry is False] = \((1 - \frac{1}{s})^{nk}\)

Now here we can use an important approximation. Remember from calculus that the special constant \(e\) can be defined as:

\[
= \lim_{n \to +\infty} \left(1 + \frac{1}{n}\right)^n
\]

Well we can take the inverse of both sides to get:

\[
\frac{1}{e} = \lim_{n \to +\infty} \left(1 - \frac{1}{n}\right)^n
\]

This tells us that, when the value of \(n\) is really large, the left and right-hand sides are going to be very close to each other. So we have a general approximation that’s useful in many probability proofs:

\[
\left(1 - \frac{1}{a}\right)^b \approx -^{b/a}
\]

(if \(a\) and \(b\) are both large integers)

Plugging this in to the probability above, we know that

\[
Pr[.table entry is False] \approx -^{nk/s}
\]

and therefore

\[
Pr[.table entry is True] \approx 1 - -^{nk/s}
\]

**Step 2: Probability of a false positive**

To get a false positive, the table entries at all \(k\) of the hash values must be “true”. Using the formula above, we know that this probability is

\[
Pr[.false positive] = p \approx \left(1 - -^{nk/s}\right)^k
\]

**Step 3: Optimal value of \(k\)**

We naturally want the probability \(p\) above to be as small as possible. To achieve that, notice first that \(-^{nk/s}\) is going to be some number between 0 and 1, and therefore the base of this big power is going to be something between 0 and 1. If you differentiate the equation above in terms of \(k\), set it to 0, and solve for \(k\) (which I’m not going to show you!), you will discover that this function is minimized when the base of the exponent equals one half:

\[
1 - -^{nk/s} = \frac{1}{2}
\]

Solve that for \(k\) to discover the optimal value for \(k\), given \(n\) and \(s\), is

\[
k = \frac{s}{n} \ln 2
\]
Step 4: How to figure out s
Given the optimal value of $k$ that we’ve just figured out, we can plug this back into the formula for $p$ to discover that

$$\Pr[\text{false positive}] = p \approx \left(\frac{1}{2}\right)^k = \frac{1}{2^{(s/n) \ln 2}}$$

Now solve that equation for $s$. You take the log base 2 of both sizes, then multiply both sides by $n/\ln 2$ to get:

$$s = -\frac{n \lg p}{\ln 2}$$

Summary of choosing Bloom filter parameters
To summarize all this, if you want to choose your parameters for a Bloom filter, given the number of things that are going to be in it $n$, here’s what you do:

1) Pick the desired failure probability $p$
2) Compute the size of the Bloom filter $s = (-n \lg p)/(\ln 2)$. Be careful about $\lg$ and $\ln$ - they aren’t the same!
3) Compute the optimal number of hash functions $k = (s/n) \ln 2$

And that’s it! It’s important to notice that, as our desired failure probability $p$ decreases, $-\lg p$ will increase, and so will $s$ and $k$ proportionally. This is what we expect!

12 Universal hash functions

- Wikipedia: Universal hashing

Now that we have all these fancy ways to use multiple hash functions and make things better, the question of choosing high-quality hash functions becomes really important. So how are we going to choose all these hash functions?

So far we’ve assumed that our hash functions are going to behave exactly like random numbers. Well, achieving this goal perfectly is an impossibility if we want the hash functions to also be fast. We saw in class why this is: because of the extremely large number of possible functions from the universe of $m$ possible keys to the $s$ possible hash values.

However, if we back off on the requirements slightly, it is possible to achieve what we are after. The key here is that we will choose the hash function randomly from a set, or “family”, of possible hash functions. Once the hash function is chosen - presumably when the data structure is first initialized - the choice of hash function never changes. However, this single random choice allows us to use the power of randomization to prove good expected performance of the hashing techniques we have covered so far.

So what properties of random hash functions do we really want? The most basic would be that the probability of collisions in the random hash function is the same as if the hash values were just chosen randomly. This requirement is captured in the following definition:

Definition

For a universe of $m$ integers, and a hash table size $s$, a universal family of hash functions if, for a randomly chosen function $h(x)$ from the family, and for any two elements $a, b$ in the universe of $m$ integers, we have

$$\Pr[h(a) = h(b)] = \frac{1}{s}$$
Again, this corresponds exactly to what the probability would be if the individual hash values $h(a)$ and $h(b)$ were chosen randomly, so the statement above is as good as random in terms of collisions between a pair of elements.

Now the question is, what mysterious functions might satisfy this property? The simplest and perhaps most useful family of universal hash functions were discovered by Carter & Wegman, and they are structured as follows:

**Carter & Wegman Hash Functions**

Let $p$ be any prime larger than $m$. Then the Carter and Wegman family of hash functions consist of functions $h(x)$ which satisfy

$$h(x) = ((ax + c) \mod p) \mod s,$$

for integers $a, c$ chosen randomly from the set $\{0, 1, 2, \ldots, p - 1\}$.

Note that the prime number $p$ does not need to be chosen randomly - it should be the same for every possible hash function $h(x)$. Note that sometimes these hash functions are written like $h_{a,b}(x)$, to emphasize that the function itself will be different for every different choice of $a$ and $b$.

In class we saw the proof that the Carter & Wegman functions are universal. You should also notice the striking similarity of this hash function to the mixed LCG pseudo-random number generator from the beginning of the course. It’s not a coincidence - good hash functions behave like random numbers!

Because of this, you should also not be surprised that there are more sophisticated families of universal hash functions based on more complicated formulas. A straightforward extension of the Carter & Wegman hash functions is the following family:

$$h(x) = ((a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k) \mod p) \mod s$$

where all of the “coefficients” $a_0, a_1, \ldots, a_k$ are randomly chosen. These hash functions take a bit longer to construct and to compute, but they satisfy even stronger randomness properties. (In particular, they form a $k$-universal family, if you want to look up what that means.)

## 13 Static and perfect hashing

- Wikipedia: nothing!

The final hashing problem we will consider is called static hashing. It’s just like the regular dictionary ADT problem, except that there are no “insert” or “delete” operations, only look-ups. In other words, the contents of the dictionary never change after it is constructed.

There are many applications of static hashing: namely, any kind of database whose keys will never or rarely change. For example, a dictionary that actually stores definitions of English words would not change very frequently. Neither would a look-up table for chemical elements, or the database mountain peak locations... you get the idea.

There is a very simple and fairly effective way to store this kind of data: just put it in a sorted array. Then running the binary search algorithm on the sorted array could achieve $O(\log n)$ time for any look-up operation. Since this is close to the performance of most dictionary data structures, but far simpler, it seems like a good idea.

Well, as it turns out, there is a better way. Perfect hashing is the name given to a way of storing a set of $n$ elements in a kind of a hash table that has $O(n)$ total size and worst-case cost $O(1)$ for any look-up operation. This solution was discovered by Fredman, Komlos, and Szemeredi in 1984, and is sometimes also called “FKS hashing.”
13.1 Warm-up: Single table with no collisions

Here is a first idea for perfect hashing: use a hash table large enough so that there will be no collisions among the \( n \) elements. Since the set of elements is unchanging, we can just try different randomly-chosen hash functions until we find one with no collisions.

But how hard will we have to search to find such a hash function? Well, that depends on the size of the table. But the birthday problem has already shown us that, unless the size of the table is \( O(n^2) \) in size or greater, there is a high likelihood that at least 2 of the elements will have a hash collision.

If we set \( s = n^2 \), then the probability of any two elements colliding is \( \frac{1}{s} = \frac{1}{n^2} \). Furthermore, the number of pairs of elements that could collide is \( \binom{n}{2} = \frac{n(n-1)}{2} \). Therefore the probability that there are no collisions at all is at most

\[
\frac{n(n-1)}{2} \cdot \frac{1}{n^2} = \frac{n^2 - n}{2n^2} < \frac{1}{2}.
\]

Now a 50% chance of success doesn’t seem very good, but by now we know that this probability decreases exponentially after repeated tries. The expected number of hash functions we will have to try before finding one that has no collisions (for \( s = n^2 \)) is just \( O(1) \).

So this idea satisfies the time requirements of perfect hashing: since every bucket has size 1 or 0, every look-up operation is just \( O(1) \) time in the worst case. But the space requirement of perfect hashing is definitely not satisfied here.

13.2 FKS Perfect Hashing

The big trick to get \( O(1) \) time and \( O(n) \) space for static hashing is to use a 2-level hash table. Here’s how it works: the top-level hash table will have size \( n \) and numerous collisions, so the elements are stored in buckets. But each bucket is itself another hash table, and the 2nd-level bucket hash tables are constructed in such a way that they don’t have any collisions. So overall, the cost of any look-up is just computing two hash functions, and since there are no collisions on the second level, this is \( O(1) \) time in the worst case.

Specifically, there is a single top-level hash function \( h(x) \), that is chosen randomly according to the constraints that will be outlined above. After choosing \( h(x) \), there will be \( n \) buckets of varying sizes; call these sizes \( k_1, k_2, \ldots, k_n \).

Each second-level hash table (the buckets) will have its own size \( s_i \) and its own hash function \( h_i(x) \). We will use the idea from above to construct these so that there are no collisions among the \( k_i \) elements: set \( s_i = k_i^2 \) and choose random hash functions \( O(1) \) times until we find one that produces no collisions.

There are two remaining questions: how the top-level hash function \( h(x) \) is chosen, and why in the world we would expect the total size of this structure to be only \( O(n) \).

The crucial value to consider is the expected total number of pairwise collisions in the top-level hash table. Since there are \( \binom{n}{2} \) pairs of elements, and each pair has a collision with probability \( 1/n \), the total expected number of collisions is at most

\[
\binom{n}{2} \cdot \frac{1}{n} = \frac{n-1}{2} < \frac{n}{2}.
\]

Since this is just the expected number, it won’t come out this way for every choice of a random hash function. But after expected \( O(1) \) choices of a random top-level hash function \( h(x) \), we will find one that has only \( n/2 \) total pairwise collisions.

Why is this important? Well, the total size of the data structure is the size of all the 2nd-level hash tables:
\[ s_1 + s_2 + \cdots + s_n = k_1^2 + k_2^2 + \cdots + k_n^2 \]

In class, we saw that each \( k_i^2 \) corresponds to the number of pairwise collisions within that bucket. So the total size above is actually just equal to \( n \) plus 2 times the total number of pairwise collisions in the top-level hash table. Since \( h(x) \) was chosen so that this number of collisions would be only \( n/2 \), the total size of all the 2nd-level bucket tables is at most \( 2n \), making the total size of this FKS perfect hashing structure \( O(n) \), as required. Hooray!