

COS 330: Great Ideas in Theoretical Computer Science

Fall 2026

Lecture 8

Hashing

Module: Randomization

1 Hashing and Hashtables

Consider the problem of building a distributed web cache. This cache stores a subset of the pages on the web, so that later requests for those pages can be served without fetching them again from their original source. It's also distributed, which means that the cached pages are spread across several machines, and so we need a quick way to determine where to look for a page. Searching every machine to find a particular page would defeat much of the purpose of having a cache in the first place. At the same time, we want to avoid overloading any one machine with too many pages, while others have little to store.

To highlight why this problem is hard, let's consider the following natural approach, that unfortunately doesn't work. Have some fixed machine whose job is to keep track of which pages are cached on which machine, and thus it can make sure that no machine is overloaded. When a page is requested, we first ask that machine where to look for it, and then query the indicated machine. This successfully balances the load, but it is not really distributed. This fixed machine needs to be queried for every page request, and it can become a bottleneck.

Let U be the universe of possible pages, with $|U| = n$, and let $S \subseteq U$ be the set of pages currently cached, with $|S| = m \ll n$. We would like to assign each page to one of q machines, with $1 \leq q \leq n$, so that we can always find a cached page without searching every machine, and without overloading any one of them.

1.1 The dictionary problem

The above is an instance of a more general problem. Many algorithms need to maintain a set of items and repeatedly ask whether a given item belongs to the set, add new items, or remove old ones: the *dictionary problem*. If the universe U of possible items is small, we can maintain the set with a *direct-address table*: an array indexed by all of U , storing each item at the array slot matching its own value. Every operation then costs $O(1)$ time, but the array itself costs $\Theta(|U|)$ space — and, exactly as with the universe of all possible web pages, U is often astronomically larger than the set S we actually want to store.

A *hashtable* keeps the $O(1)$ -time spirit of a direct-address table while using much less space. Instead of one slot per element of U , we use only $q \ll |U|$ slots, together with a function that assigns each element of U to one of them.

Definition 1.1 (Hash function). Let U be a universe and $q \geq 1$ an integer. A *hash function* is a function

$$h : U \longrightarrow [q] = \{0, 1, \dots, q - 1\}.$$

We call $[q]$ the set of *locations* — for our cache, the q machines.

Definition 1.2 (Collision). Two distinct elements $x, y \in U$ *collide* under h if $h(x) = h(y)$.

Since $q < |U|$, a hash function cannot be injective, so collisions between some pairs of elements are unavoidable. There are many ways to handle them, as you might remember from COS 226. To keep the analysis relevant to the cache problem, in this lecture we will consider *separate chaining*: each location stores a list of all the items currently assigned to it. $\text{INSERT}(x)$ appends x to the list at $h(x)$; $\text{QUERY}(x)$ and $\text{DELETE}(x)$ find that same list and search it for x . Hashing tells us which list to search — it does not, by itself, tell us whether x is actually in it.

Definition 1.3 (Load). Fix a hash function h and a set $S \subseteq U$. The *load* of a location $i \in [q]$ is

$$X_i = |\{x \in S : h(x) = i\}|,$$

the number of elements of S that hash to i .

In the above definition, think of S as the set of elements currently stored in the hashtable. In the cache problem, S is the set of pages currently cached, and X_i is the number of pages assigned to machine i .

The cost of an operation on x is governed by the load of $h(x)$, so a hash function is “good” for our purposes if it tends to keep every load small. Returning to our cache, we take $n \geq 2$, and we want three things from h :

1. **Few collisions:** items should be spread out over the q locations, so no load is ever much larger than $|S|/q$ — no machine gets overloaded.
2. **A small table:** we want q close to $|S|$, not close to $|U|$ — far fewer than n machines, not one per possible page.
3. **Fast evaluation:** computing $h(x)$ should itself be cheap, ideally $O(1)$, since we pay this cost on every operation — deciding a page’s machine should be quick.

These goals pull against each other. The rest of this lecture is about how much randomness is needed to get all three at once.

We choose one function and keep using it as pages enter and leave the cache. In this sense, the function must be *oblivious* to the particular set S : we do not want to redesign it whenever the contents change. A natural first attempt is to choose a fixed rule that spreads the whole universe evenly among the locations.

For any fixed hash function h , the n pages are partitioned among q locations. By the pigeonhole principle, some $i \in [q]$ satisfies

$$|h^{-1}(i)| \geq \frac{n}{q}, \quad h^{-1}(i) = \{x \in U : h(x) = i\}.$$

If $m \leq |h^{-1}(i)|$, a set of m pages chosen from this one preimage all lands at location i . More generally, a set of size m can put at least $\min\{m, \lceil n/q \rceil\}$ pages there. Even a rule that balances the entire universe can be badly unbalanced on a particular subset.

We can address this by introducing randomness into the choice of h . As in Lecture 7, we will keep the input fixed and introduce randomness into the algorithm. Here S is fixed and arbitrary, and we choose h randomly, independently of S . The desired statement is that *for each fixed set* S , a random choice of h is unlikely to perform badly. We do not ask one function to work well for every possible set simultaneously. The distinction matters: choosing S after inspecting h could recreate the bad example above. For a changing cache, our bounds apply to a set of contents fixed independently of the choice of h .

2 A Fully Random Hash Function

Start with the most generous source of randomness: choose $h(x)$ independently and uniformly in $[q]$ for every $x \in U$. This gives a useful benchmark before we worry about how to store or evaluate h . Each page in our fixed set S chooses a uniformly random location, independently of all other pages.

Fix a location i , and recall the load X_i from Definition 1.3. It counts pages satisfying a condition, so write it as a sum of indicators. For each $j \in S$, define

$$X_{ij} = \begin{cases} 1, & \text{if } j \text{ is mapped to location } i, \\ 0, & \text{otherwise.} \end{cases} \quad X_i = \sum_{j \in S} X_{ij}.$$

Since $h(j)$ is uniform in $[q]$,

$$\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{1}{q}.$$

Linearity of expectation now gives

$$\mathbb{E}[X_i] = \sum_{j \in S} \mathbb{E}[X_{ij}] = \frac{|S|}{q} = \frac{m}{q}.$$

This is the load we would get under a perfectly even assignment, but an expectation alone does not say that each location is close to that value.

For the rest of the load analysis, assume $m \leq q$, so the mean load is at most one. This is a convenient regime that illustrates the main ideas while simplifying the calculations. You can check that the same approach works for larger m with a few extra steps. The question is how large the *largest* load can be.

2.1 A Chebyshev bound

For fixed i , the indicators X_{ij} are independent because the values $h(j)$ are independent for different pages. Each indicator is Bernoulli with parameter $1/q$. Using the variance formulas from Lecture 6,

$$\text{Var}(X_i) = \text{Var}\left(\sum_{j \in S} X_{ij}\right) = \sum_{j \in S} \text{Var}(X_{ij}) = \sum_{j \in S} \frac{1}{q} \left(1 - \frac{1}{q}\right) \leq \frac{m}{q} \leq 1.$$

Unlike the expectation calculation, adding these variances requires information about independence.

Write $\mu_i = \mathbb{E}[X_i]$ and $\sigma_i = \sqrt{\text{Var}(X_i)}$ for the standard deviation of X_i . We can now use Chebyshev's inequality:

$$\Pr[|X_i - \mu_i| \geq t\sigma_i] \leq \frac{1}{t^2}.$$

Here $\mu_i \leq 1$ and $\sigma_i \leq 1$, so a single standard deviation spans a distance of at most 1. A load of at least $1 + t$ is therefore at least t above the mean in absolute terms; since each standard deviation is worth at most 1 of that distance, this gap of t already amounts to at least t standard deviations. Thus

$$\Pr[X_i \geq 1 + t] \leq \frac{1}{t^2}.$$

Now let B_i be the event that $X_i \geq 1 + t$. The union bound gives

$$\Pr[B_0 \cup B_1 \cup \dots \cup B_{q-1}] \leq \sum_{i=0}^{q-1} \Pr[B_i] \leq \frac{q}{t^2}.$$

Taking $t = \sqrt{2q}$ makes this bound at most $1/2$. With probability at least $1/2$, every location therefore has load less than $1 + \sqrt{2q}$.

The loads at different locations are not independent: assigning a page to one location prevents it from going to another. The union bound does not require that independence. The independence used earlier was across pages for a *single fixed location*.

This gives a simultaneous guarantee, but a load on the order of $\sqrt{q}$ is still much larger than the mean of at most one. It is the bound that is weak; we have not shown that such a large load actually occurs. In theory “slang” we are asking whether this is a *tight* bound.

2.2 A Chernoff bound

The load is a sum of independent indicators, exactly the setting for Chernoff. Recall that, for $\delta > 0$,

$$\Pr[X_i \geq (1 + \delta)\mu_i] \leq \exp\left(-\frac{\delta^2\mu_i}{2 + \delta}\right).$$

Here we care about deviations much larger than the mean, so we use the form valid for all positive δ , rather than the simplification restricted to $\delta \leq 1$. When $\delta \geq 2$, we have $2 + \delta \leq 2\delta$, and hence

$$\exp\left(-\frac{\delta^2\mu_i}{2 + \delta}\right) \leq \exp\left(-\frac{\delta\mu_i}{2}\right).$$

The exponent is now proportional to the additive deviation $\delta\mu_i$.

To make a union bound over q locations useful, we want the probability for one location to contain a factor $1/q$. Set $L = \ln q + 100$ and choose $\delta = 2L/\mu_i$. We use natural logarithms throughout this calculation. Since $0 < \mu_i \leq 1$, we have $\delta \geq 2$, and

$$(1 + \delta)\mu_i = \mu_i + 2L \leq 1 + 2L.$$

Exceeding $1 + 2L$ therefore implies exceeding the Chernoff threshold. It follows that

$$\begin{aligned} \Pr[X_i \geq 1 + 2(\ln q + 100)] &\leq \Pr[X_i \geq (1 + \delta)\mu_i] \\ &\leq \exp\left(-\frac{2(\ln q + 100)}{2}\right) \\ &= e^{-\ln q - 100} = \frac{1}{q}e^{-100}. \end{aligned}$$

Let B_i now be the event that $X_i \geq 1 + 2(\ln q + 100)$. Another union bound finishes the calculation:

$$\Pr[B_0 \cup \dots \cup B_{q-1}] \leq q \cdot \frac{1}{q}e^{-100} = e^{-100}.$$

Thus, with probability at least $1 - e^{-100}$, every location has at most $1 + 2 \ln q + 200$ pages. The constant 100 simply makes the failure probability very small. You can redo the calculation with $L = \ln q + \ln(1/\epsilon)$ to get a failure probability of at most ϵ for any $\epsilon > 0$. The important improvement is from a square-root bound to a logarithmic one, while also obtaining a much smaller failure probability.

We can again ask whether this bound is tight. The answer is no, the right behavior is actually $\Theta(\ln q / \ln \ln q)$. You'll prove the upper bound in precept, but the lower bound is a bit more involved (but it's a really beautiful argument, so you should look it up if you're curious). Also, note that this analysis can be applied to the exact same problem that was mentioned in Lecture 1, and it's also known as the “balls and bins” problem.

3 2-Universal Hash Families

We started the lecture by asking for a hash function that spreads the pages evenly among the machines, and we constructed a fully random hash function that does just that. The analysis was instructive, however this is still not quite a solution to our original problem. It has given us a strong load guarantee (goal 1 from Section 1), but it hides an implementation problem for goals 2 and 3. Its values at different elements of U are unrelated. An explicit representation is a lookup table with n entries, each storing a value in $[q]$. For $q \geq 2$, this takes $n \lceil \log_2 q \rceil$ bits, even though only $m \ll n$ pages are currently cached — exactly the large representation goal 2 was meant to avoid. Choosing an independent value anew on each query would not work either: we must return the same location every time we see the same page.

Instead, we would like to choose h from a small family of functions. A short description should determine all its values, and evaluating $h(x)$ should be quick. We should not expect the values of such a function to be fully independent. The question is how much randomness our desired guarantee actually needs.

Consider looking up a fixed page x . If each location stores its pages in a list, the work after computing $h(x)$ is controlled by the number of pages sharing x 's location. We will bound the *expected* size of this list. This is a different, weaker target than bounding the maximum load with high probability, but it is enough to give a useful expected query cost.

Definition 3.1 (2-universal hash family). A finite family $\mathcal{H}$ of functions $h : U \rightarrow [q]$ is *2-universal* if, for every distinct $x, y \in U$, the chance that x and y collide (Definition 1.2) is at most $1/q$:

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{q}.$$

The probability is over the random choice of the hash function.

We choose uniformly from the family. When the functions are described by parameters, we regard the family as indexed by those parameters and choose the parameters uniformly. The condition controls each fixed pair of distinct pages: they collide with probability no greater than under fully random hashing. It does not assert that all hash values are independent, or even that each hash value is uniform. In particular, we cannot simply reuse the Chernoff analysis from Section 2 for a 2-universal family.

Lemma 3.1. Fix $S \subseteq U$ with $|S| = m$ and $x \in S$, independently of the random choice of h from a 2-universal family. Let X count the elements $y \in S$ such that $h(y) = h(x)$, including x itself. Then

$$\mathbb{E}[X] \leq 1 + \frac{m-1}{q} \leq 2 \quad \text{if } m \leq q.$$

Proof. The page x always shares its own location. For each other $y \in S$, let

$$I_y = \begin{cases} 1, & \text{if } h(y) = h(x), \\ 0, & \text{otherwise.} \end{cases}$$

Then

$$X = 1 + \sum_{\substack{y \in S \\ y \neq x}} I_y.$$

By linearity of expectation and 2-universality,

$$\begin{aligned} \mathbb{E}[X] &= 1 + \sum_{\substack{y \in S \\ y \neq x}} \mathbb{E}[I_y] \\ &= 1 + \sum_{\substack{y \in S \\ y \neq x}} \Pr[h(y) = h(x)] \\ &\leq 1 + \sum_{\substack{y \in S \\ y \neq x}} \frac{1}{q} = 1 + \frac{m-1}{q}. \end{aligned}$$

When $m \leq q$, this is at most 2. ■

For a fixed query $x \notin S$, the same calculation has no initial 1: all m stored pages are distinct from x , so the expected number at $h(x)$ is at most $m/q \leq 1$. Thus a list search examines only a constant expected number of pages when $m \leq q$. Assuming constant-time hash evaluation and identifier comparisons, query, insertion with a check for duplicates, and deletion each take constant expected time for a fixed operation and a fixed set of contents independent of h .

So we conclude that if we can find a 2-universal family with a short description and fast evaluation, we can implement a hashtable with constant expected cost per operation. The next section gives a simple construction that meets these requirements.

3.1 Construction of a 2-universal hash family

It remains to find a family with the promised collision bound and a short description. Identify the elements of U with the integers $\{0, 1, \dots, n-1\}$. Choose a prime p with $n < p < 2n$: such a prime exists by Bertrand's postulate¹. The prime is fixed once, rather than chosen anew for each operation.

For $a \in \{1, \dots, p-1\}$ and $b \in \{0, \dots, p-1\}$, define

$$f_{a,b}(x) = ax + b \pmod{p}, \quad h_{a,b}(x) = f_{a,b}(x) \pmod{q}.$$

Here each reduction returns its representative in $\{0, \dots, p-1\}$ or $\{0, \dots, q-1\}$, respectively. The order matters: first reduce modulo p , then reduce the resulting integer modulo q . Our family is

$$\mathcal{H} = \{h_{a,b} : a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\}\},$$

with a and b chosen independently and uniformly from their respective ranges. Storing these two parameters takes $O(\log n)$ bits. Evaluating the function uses a constant number of arithmetic operations; as before, we count arithmetic on integers of this size as constant time.

¹Bertrand's postulate states that for every integer $n > 1$, there is at least one prime p such that $n < p < 2n$.

The first reduction mixes the page identifiers using arithmetic modulo a prime; the second places the result in the desired range of q locations. We didn't prove that this family is 2-universal in lecture, but for completeness we include the proof here. It requires a little number theory that you aren't expected to know, but the ideas are not too complicated. In particular, it uses the fact that if p is prime and $x \not\equiv y \pmod{p}$, then $x - y$ has a multiplicative inverse modulo p , i.e., there exists a unique integer $z \in \{1, \dots, p-1\}$ such that $(x - y)z \equiv 1 \pmod{p}$.

Theorem 3.1. The family $\mathcal{H}$ above, sampled by its parameters (a, b) , is 2-universal.

Proof. Fix distinct $x, y \in U$. For any two target values $u, v \in \{0, \dots, p-1\}$, we ask which parameters (a, b) send x, y to u, v before the final reduction modulo q : that is, which (a, b) satisfy

$$ax + b \equiv u \pmod{p}, \quad ay + b \equiv v \pmod{p}.$$

Subtracting the second congruence from the first eliminates b and leaves $a(x - y) \equiv u - v \pmod{p}$. If $u = v$, this forces $a(x - y) \equiv 0 \pmod{p}$; since p is prime and $x \not\equiv y \pmod{p}$, that would force $a \equiv 0 \pmod{p}$, which no $h_{a,b} \in \mathcal{H}$ uses (recall $a \in \{1, \dots, p-1\}$). So no hash function in our family ever sends x and y to the same value before the final reduction, and it suffices to consider $u \neq v$.

For $u \neq v$, the element $x - y$ has a multiplicative inverse modulo p (again because p is prime and $x \not\equiv y \pmod{p}$), so multiplying $a(x - y) \equiv u - v \pmod{p}$ through by $(x - y)^{-1}$ determines a uniquely and nonzero; substituting this a back into $ax + b \equiv u \pmod{p}$ then determines b uniquely as well. The system therefore has a unique solution $a, b \in \{0, \dots, p-1\}$, namely

$$a = (u - v)(x - y)^{-1} \bmod p, \quad b = (vx - uy)(x - y)^{-1} \bmod p.$$

Since there are exactly $p(p-1)$ choices of (a, b) in total, and exactly one of them realizes any given pair $u \neq v$, we conclude

$$\Pr_{a,b}[f_{a,b}(x) = u \text{ and } f_{a,b}(y) = v] = \frac{1}{p(p-1)} \quad \text{for every } u \neq v.$$

Summing over the target pairs that still collide after the final reduction modulo q ,

$$\Pr_{a,b}[h_{a,b}(x) = h_{a,b}(y)] = \frac{C}{p(p-1)},$$

where C is the number of ordered pairs $(u, v) \in \{0, \dots, p-1\}^2$ with $u \neq v$ but $u \bmod q = v \bmod q$.

To bound C : for each fixed u , there are at most $\lfloor p/q \rfloor$ integers $v \in \{0, \dots, p-1\}$ with $v \neq u$ but $v \equiv u \pmod{q}$. Because p is prime, q cannot divide p for $1 < q < p$, so $\lfloor p/q \rfloor \leq (p-1)/q$. Summing over the p choices of u gives $C \leq p(p-1)/q$, and therefore

$$\Pr_{a,b}[h_{a,b}(x) = h_{a,b}(y)] = \frac{C}{p(p-1)} \leq \frac{1}{q},$$

as required. ■

We can now return to caching with an explicit function: store two parameters, compute a page's location, and search the pages assigned there. Fully random hashing showed what independent assignments could accomplish. For a constant expected search cost, however, we needed only to control collisions between pairs. Choosing a weaker guarantee let us replace a table over the whole universe with a function described by just two numbers.