10.1 Introduction

Let's go back to the dictionary setting, where we want to insert and lookup items, and possibly delete them. In previous lectures we showed how to do this with balanced search trees (B-trees, 2-3-4 trees, and red-black trees in particular). These structures allowed us to do these operations in $O(\log n)$ time (with the right choice of t in the case of B-trees). Today we're going to see another approach: hashing. You should all have seen the basics of hashing in Data Structures, and a lot of the book chapter talks about these basics, so I'm not going to go over them. Instead, we're going to talk about some interesting theoretical guarantees that you can get from hashing.

10.2 Hashing Basics

Let's set some notation and basic definitions.

- Keys come from some universe U. Think of U as being very large, e.g., all strings with at most 40 characters. Note that this assumption is stronger than the comparison model, but weaker than assuming specific structure on the keys (as in radix sort).
- There is a set $S \subseteq U$ of keys that we actually care about. Think about S as relatively small, e.g., the names of the people in this room. For the rest of today, we'll let $N = |S|$ (which we may or may not know ahead of time, depending on the setting).
- We're going to have an array A of size M, called a *hash table*, and a *hash function* $h : U \to [M]$ (recall that $[M] = \{1, 2, \ldots M\}$). Think of M as being significantly smaller than $|U|$. The idea is that we will store key x in $A[h(x)]$.
- We also need a method of resolving *collisions*: elements that hash to the same value. If $h(x) = h(y)$, then we want to store them both in the same spot in the hash table. We will do this with a method known as *separate chaining*: $A[i]$ will actually be a linked list of elements rather than just one, with all keys that we've inserted that hash to i being in this list.

This approach makes it easy to implement the basic dictionary operations.

- Lookup(x): Walk down the list at $A[h(x)]$ until we find x (or walk to the end of the list)
- Insert(x): Add x to the beginning of the last at $A[h(x)]$.
- Delete(x): Walk down the list at $A[h(x)]$ until we find x. Remove it from the list.

Of course, this leaves out the most important detail: what should we use as our hash function h? Let's think about what we want from a hash function.

1. Clearly the running time of lookup and delete depends on how many collisions we have. Slightly more formally, the running to do lookup or delete of x is $O(\text{length of list at } A[h(x)])$. So we want h to not result in too many collisions. Note that inserts only take $O(1)$ time no matter what.
2. $M = O(N)$: we would like the table to be about the same size as the number of elements that we actually care about.
3. h should be fast to compute. Today we'll generally be thinking about h as constant time to compute, but this isn't necessarily reasonable. So we should remember that h should be easy to compute when we're designing hash functions.

So our goal is to hash onto a relatively small size table in a way that keeps the lists in each table entry small. Unfortunately there's some very bad news: this is impossible.

Theorem 10.2.1 *For any hash function h, if $|U| \geq (N-1)M + 1$, then there exists a set S of N elements that all hash to the same location.*

Proof: Basically by the pigeonhole principle. Slightly more formally, let's prove the contrapositive. Suppose that for every $i \in [M]$, the number of elements of U that hash onto i has size less than N. Then the total number of elements in U is at most $(N-1)M$. ■

So this is one reason why hashing seems so mysterious. How can we claim that hashing works well, if we can always find an instance on which it does poorly? One answer is the usual cop-out – "real" instances are not adversarially chosen, so as long as our hash function isn't completely braindead we're OK. This may or may not be true, but in either case, we still want to understand if there's any true theoretical guarantees that we can make about hashing.

This discussion should remind you of *quicksort*, where if we chose an arbitrary pivot we could do very badly ($\Omega(n^2)$ comparisons). One way we fixed that was to choose a *random* pivot, giving randomized quicksort, which had expected running time of $O(n \log n)$ on *every* input. Can we do something similar here?

The first obvious approach would be to make h a *random function*. Clearly it has to be deterministic once chosen, i.e., $h(x)$ always needs to equal the same value in order to do lookups – it wouldn't work if $h(x)$ was one value at one point in time and a different value later. Instead, the obvious way is to think about designing h by choosing, for each $x \in U$, a random value $y \in [M]$ and setting $h(x) = y$. This intuitively will let us get the same kinds of guarantees as for randomized quicksort – while for any h we know from Theorem 10.2.1 that there will be *some* bad set of elements, we will have the guarantee that *for every* set of elements, the expected performance is good. Again, this is like randomized quicksort, where for any sequences of pivots there is *some* bad starting array, but for every starting array the expected running time is good.

But this also has a major problem: since h doesn't have any real structure, the only way to store and compute h is to store the whole thing! That is, when we first choose to set $h(x)$ to y, we need to somehow remember this so that later we can lookup x by computing $h(x)$. And since h doesn't have any structure, we have to actually store $h(x)$. So now we're stuck – given a key, we need to store something about it. That's the exact problem we're trying to solve! Can we get the properties of random functions that we like, while still making h easy to store and compute?

10.3 Universal Hashing

Let's start out with some definitions, before we see any constructions.

Definition 10.3.1 *A probability distribution H over hash functions $\{h : U \to [M]\}$ is* universal *if*

$$\Pr_{h \sim H}[h(x) = h(y)] \leq 1/M$$

for all $x, y \in U$ with $x \neq y$.

Note that a random function definitely has this property, or slightly more formally, the distribution H which is uniform over all functions from U to $[M]$ has this property. Why is this the key property that we care about? Mostly because it allows us to prove the next theorem.

Theorem 10.3.2 *If H is universal, then for every set $S \subseteq U$ with $|S| = N$ and for every $x \in U$, the expected number of collisions (when we draw h from H) between x and elements of S is at most N/M.*

In other words, if we do a lookup on x, then the expected time is only $O(N/M)$, which if we've sized our table correctly ($M \approx N$) is only $O(1)$!

Proof: For every $y \in S$ with $y \neq x$, let C_{xy} be an indicator random variable which we set to 1 if $h(x) = h(y)$ (i.e., x and y collide) and otherwise set to 0. Then by the definition of universal, for every $y \neq x$ we know that

$$\mathbf{E}[C_{xy}] = \Pr_{h \sim H}[h(x) = h(y)] \leq 1/M.$$

The number of collisions between x and elements of S is precisely $\sum_{y \in S : y \neq x} C_{xy}$. Thus linearity of expectations implies that

$$\mathbf{E}\left[\sum_{y \in S : y \neq x} C_{xy}\right] = \sum_{y \in S : y \neq x} \mathbf{E}[C_{xy}] \leq \sum_{y \in S : y \neq x} \frac{1}{M} \leq N/M,$$

as claimed. ∎

Now this gives us an easy but important corollary:

Corollary 10.3.3 *If H is universal, then for any* sequence *of L insert, lookup, and delete operations in which there are at most $O(M)$ elements in the system at any time, the expected total cost of the whole sequence is only $O(L)$ (assuming h takes constant time to compute)*

Proof: Consider the i'th operation of the sequence. If it is an insert, then it takes time $O(1)$. If it is a lookup or delete, then since the number of elements in the system at time i is only $O(M)$, Theorem 10.3.2 implies that the expected time is $O(1)$. Thus every operation takes $O(1)$ time in expectation, and the corollary is implied by linearity of expectations. ∎

10.3.1 Constructing a universal hash family

Of course, this is all pretty pointless if we can't actually construct a universal H. Luckily, it turns out that we can. I'm going to show you a way which I think is pretty intuitive and doesn't require

much math, but there's a different method in the book that is a little bit better but also a little bit more complicated.

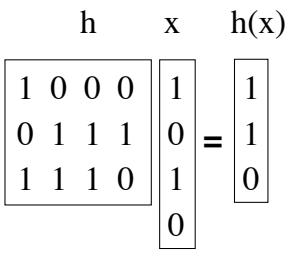
First, let's set up one more definition which will allow us to abuse notation in useful ways.

Definition 10.3.4 *If H is universal and is a uniform distribution over a set of functions $\{h_1, h_2, \dots\}$, then that set is called a* universal hash family*.*

We will frequently use "H" to mean both the set and the uniform distribution over it. This makes things a little easier to state, and the precise meaning should be clear from context.

Let's say that keys are u bits long, so $U \subseteq \{0,1\}^u$. And let's say that the hash table size M is 2^b, so an index into the table is an element of $\{0,1\}^b$.

Consider the set H of all $b \times u$ binary matrices, i.e., all matrices in $\{0,1\}^{b \times u}$. We can think of each matrix as mapping an element of U to an index in $\{0,1\}^b$ in the obvious way: given an element $x \in \{0,1\}^u$, we map it to $h(x) = hx \in \{0,1\}^b$ (where we're doing everything mod 2). You can think of these matrices as being "short and fat". For example:



I claim that H is a universal hash family. In other words, we will prove the following theorem. This turns out to have a surprisingly simple but subtle proof.

Theorem 10.3.5 $\Pr_{h \sim H}[h(x) = h(y)] = 1/M$ *for all* $x \neq y \in \{0,1\}^u$.

Proof: Let's think about matrix-vector multiplication for a minute. It's not hard to see that, in this setting, hx is precisely the sum of the columns of h in which x_i is 1. In other words, $hx = \sum_{i:x_i=1} h^i$, where h^i denotes the i'th column of h. This is just from the definition of matrix-vector multiplication.

Now consider two arbitrary keys $x, y \in \{0,1\}^u$ with $x \neq y$. Since $x \neq y$, there is some $i \in [u]$ such that $x_i \neq y_i$. Without loss of generality, we may assume that $x_i = 0$ and $y_i = 1$. Imagine that we first draw all of the entries of h *except* the i'th column. Then even without drawing the entries of this column, we already know what $h(x)$ is, i.e., it is already fixed. But each of the 2^b different possibilities for this column results in a different value of $h(y)$, and exactly one of those possibilities is equal to $h(x)$. Thus the probability that $h(y) = h(x)$ is exactly $1/2^b$. ■

10.4 Perfect Hashing

Now that we know how to construct a universal hash family, let's use it for something really cool: *perfect hashing*. Suppose that the elements of the dictionary are known and fixed: I tell you ahead

of time *exactly* which set S of N keys you care about, and there are never any insertions or deletions, just lookups. This is like a real physical dictionary – once it's printed, the dictionary never changes and I know exactly what's in it. But I still want to use to for lookups. Note that I don't particularly care how long it takes to build the dictionary (do the inserts) – I care far more about how long a lookup takes.

What's the best way of doing this? Based on the techniques we already know, there are a couple obvious options. I could take inspiration from a real dictionary, and just keep all N keys in a sorted array. Now I can lookup using binary search, for a running time of $O(\log n)$ (assuming comparisons are constant time). Or I could use a balanced search tree, also getting lookup time of $O(\log n)$. Can we do better with hashing?

It turns out that the answer is yes! We will say that a hash function is *perfect* for our set S of N keys if all lookups only take $O(1)$ time. We want to build low-space perfect hash functions, and we'll do it using universal hashing.

10.4.1 $O(N^2)$-space perfect hashing

Suppose that we're willing to use a table of size $M = \Theta(N^2)$. Then there's a very simple solution: construct a universal hash family, and sample a hash function from it. I claim that with probability at least $1/2$, the hash function you've sampled is actually perfect! So you can just repeat this (a constant number of times in expectation) until you actually get a perfect hash function (since of course you can test for a given h whether it is perfect by hashing all of S and seeing if you have too many collisions).

Theorem 10.4.1 *Let H be universal with $M = N^2$. Then $\Pr_{h\sim H}[\text{no collisions in } S] \geq 1/2$.*

Proof: For any pair x, y of distinct elements of S, the probability that they collide is at most $1/M$ by the definition of universal. There are exactly $\binom{N}{2}$ such pairs. Thus the probability that there exists a collision is at most

$$\binom{N}{2}/M = \frac{N(N-1)}{2N^2} \leq 1/2$$

as claimed. ■

So if we're willing to spend space N^2, it's easy enough to ensure constant time lookups. What if we don't want to use space N^2? In particular, what if we only want space $O(N)$?

10.4.2 $O(N)$-space perfect hashing

Getting $O(N)$ space with constant time lookups was actually a big open question for a long time. Clearly it's something that we'd like to do – we want to be able to store a bunch of things and then look them up quickly without using any extra space. But that's easier said than done. Fortunately, we now have a nice way of doing this using *two levels* of universal hashing.

To begin, we'll first use universal hashing to choose a hash function which hashes into a table of size N. Unless we are insanely lucky, this will likely result in some collisions – too many for perfect hashing. However, we can now look at each of the N bins, and for every bin we will *rehash* the keys into a table of size quandratic in the size of the bin.

More formally, we have a first-level hash function h (drawn from some universal family) and a first-level table A of size N. Now for every $i \in \{0, 1, \ldots, N-1\}$, let n_i denote the number of keys of S which hash onto i, i.e., $n_i = |\{x \in S : h(x) = i\}|$. Then for every such i, we will use our quadratic size perfect hashing method: we will create a second-level hash table of size n_i^2 and a second-level hash function h_i onto $[n_i^2]$ which is perfect for these keys (as in the last subsection), and we will use this hash table to store all of the keys that hashed onto i. So to lookup a key x, we first compute $i = h(x)$ and then look for x in $A_i[h_i(x)]$.

Theorem 10.4.2 *All lookups only take $O(1)$ time.*

Proof: Clearly computing $i = h(x)$ takes constant time, so we just need to argue that looking up x in $A_i[h_i(x)]$ is fast. But this is true because we chose h_i to be perfect for the n_i keys which hashed onto i. ■

The more interesting question is why this whole structure only has size $O(N)$. To see this, first note that the total size is $O(N + \sum_{i=1}^{N} n_i^2)$. So we really just need to prove that $\sum_{i=1}^{N} n_i^2 \leq O(N)$.

Theorem 10.4.3 *Let H be universal onto a table of size N. Then*

$$\Pr_{h \sim H}\left[\sum_{i=1}^{N} n_i^2 > 4N\right] < 1/2.$$

Proof: We will prove that $\mathbf{E}\left[\sum_{i=1}^{N} n_i^2\right] \leq 2N$. This then implies the theorem by Markov's inequality.

Now we'll use a neat counting trick: $\sum_{i=1}^{N} n_i^2$ is precisely the number of *ordered* pairs that collide, including an element colliding with itself! For example, if we have the i'th bucket has contains three elements $\{a, b, c\}$, then $n_i^2 = 9$, and this is precisely the number of ordered colliding pairs $(a,a), (a,b), (a,c), (b,a), (b,b), (b,c), (c,a), (c,b), (c,c)$. So, as before, let $C_{xy} = 1$ if x and y collide and 0 otherwise (but now we'll allow elements to collide with themselves). So we get that

$$\begin{aligned}
\mathbf{E}\left[\sum_{i=1}^{N} n_i^2\right] &= \mathbf{E}\left[\sum_{x \in S}\sum_{y \in S} C_{xy}\right] \\
&= N + \sum_{x \in S}\sum_{y \in S : y \neq x} \mathbf{E}[C_{xy}] && \text{(linearity of expectations)} \\
&\leq N + \frac{N(N-1)}{M} && \text{(definition of universal} \\
&< 2N && \text{(since } M = N\text{)}
\end{aligned}$$

This implies the theorem by Markov's inequality. ■