Lecture 13: Universal and Perfect Hashing

Overview

1. Review: dictionaries, chaining, simple uniform hashing assumption
2. Universal hashing
3. Perfect hashing

For both of the hashing techniqes described today, we'll try to get contant expected time – note that this is not quite the same thing as "constant time with high probability" (think about the difference between those two things).

Review

Dictionary Problem

A Dictionary is an Abstract Data Type(ADT) that maintains a set of items. Each item is associated with a key. It subjects to the following operations:

- **insert(item)**: add something to set
- **delete(key)**: remove something from set
- **search(key)**: return item with key if it exists

We assume that items have distinct keys (or that inserting new ones clobbers old ones). This is an exact search: either you find something or you do not – there's nothing like nearby / successor values, etc.

People sometimes get a bit hazy about what's a *key* and what's an *item*. Sometimes they're the same thing, sometimes not. Think of it as a word and its definition in a real-life dictionary: you insert an item = (word, def); you can call delete(word); you can search(word). In many cases your delete may be on an item – this will depend upon the specific problem, but what I've given you here is a pretty general definition.

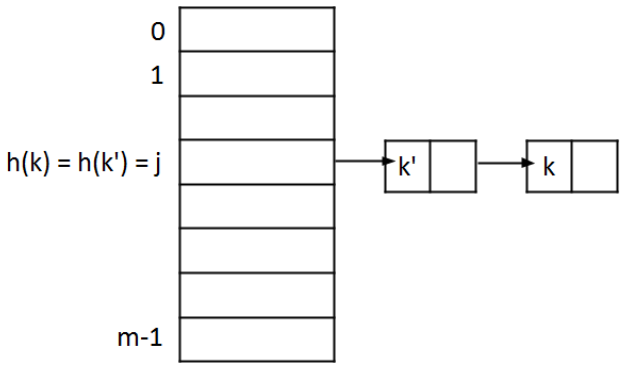
It is important, at a higher level, to differentiate betweent the problem, or the Abstract Data Type (or interface, or whatever terminology works for you) and the methods used to solve it. One is basically a set of specifications (what) and the other is the mechanism used to satisfy those specifications (how).

Hashing for kids

Goal: $O(1)$ time per operation and $O(n)$ space complexity.
Definitions:

- u = number of all possible keys (*e.g.*, all possible MIT IDs, which would be around 10^9)
- n = number of keys in the table (this is the actual data: # of students in 6.046, so on the order of 10^2)
- m = number of slots in the table (we want this to be on the order of 10^2, and to be not too much bigger than n)
- $h : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}$
  (We're assuming that everything has been mapped to integers)

Solution: hashing with chaining



Our hash function maps each item in U (the universe of keys/items) into one of the m slots, so we go through each of the n data items and insert them. We want a good hash function, which means that the probability of two distinct items hashing to the same slot in the table to be be $1/m$ (if there are m slots). That's pretty much the best we can expect. If everything were to be completely random, that is, if h were to choose a random number for every key, that's what we'd expect to happen. We can't actually do that (choosing a random number each time) because whatever we do needs to be repeatable. It wouldn't exactly be very useful if we could insert items but not retrieve them.

Assuming simple uniform hashing,

$$\Pr_{k \neq k'} \{h(k) = h(k')\} = \frac{1}{m}$$

we achieve $\Theta(1+\alpha)$ time per operation, where $\alpha = \frac{n}{m}$ is called load factor.

But notice that the "randomness" here, the thing over which we are computing or estimating this probability, is the keys. If k and k' are really random, this is great. If my hash function isn't too bad (say, $h(x) = 1$), and my keys are random, we're ok.

We want to achieve constant expected time no matter what our keys are. What we had before was really an average case analysis. We saw this with basic quicksort, which chooses $a[1]$ as pivot – for an average input, this works fine, because choosing $a[1]$ is "as good as" choosing a random element if the input is reasonably random. Of course, in real life, inputs may have all sorts of patterns, so this might or might not be a good idea, depedning upon the situation.

We saw how you could avoid assuming things about the input data by choosing a random pivot. Notice the change here – there's a very big difference between assuming things about the input, and about what you'll do once you have it. It may not always be reasonable to assume that your input will be sufficiently random, but you can (almost) always do a random coin flip.

That, in some sense, is exactly what we're going to do: we're going to choose the hash function randomly to avoid assuming things about the keys. (We are going to remove the unreasonable simple uniform hashing assumption.)

Universal Hashing

Note: We'll have no collisions *in expectation* in Universal, and we'll guarantee there are 0 conflicts in Perfect, with the caveat that perfect hashing only works for static sets. (This is ok if your set changes only occasionally.)

First, let's set up an outline of what we want:

- Choose a random hash function h from $\mathcal{H}$
- Require $\mathcal{H}$ to be a *universal hash function family* such that

$$\Pr_{h \in \mathcal{H}}\{h(k) = h(k')\} \leq \frac{1}{m} \text{ for all } k \neq k'$$

- Assume h is random, and make no assumption about input keys. (like Randomized Quicksort)

If we want to choose a hash function h randomly from some family of hash functions $\mathcal{H}$, we'll need to add some constraints to $\mathcal{H}$: what if $\mathcal{H}$ is made up entirely of functions that map to 42 in various ways? That's not going to work. So we're going to require $\mathcal{H}$ to be "Universal".

As the second bullet above describes, we want a random h to have at most a $1/m$ chance to have k and k' collide. You choose the keys, and then I choose a hash function. The hash function will not depend on the keys, but I choose them after you choose the keys. (This ensures you can't tailor your keys to my choice.) Note that this is a only a description of the properties we'd like such a family to possess – we have not yet shown that such a family even exists, let alone how to find one.

If h is chosen randomly from a universal hash function family, we want to show that the load factor, or the the the expected number of keys sent to each slot, is what we want it to be (i.e., $1 + n/m$). Let's say this a bit more formally:

Theorem: For n arbitrary distinct keys and random $h \in \mathcal{H}$, where $\mathcal{H}$ is a universal hashing family,

$$E[\text{ number of keys in a slot }] \leq 1 + \alpha \quad \text{where} \quad \alpha = \frac{n}{m}$$

Note that the expectation, like the probability, is over our choice of h.

Proof:

Hint: whenever you want to count something in expectation, use indicator random variables. Name the events you want to count, and sum them up. Note that I below is a random variable because it depends on h, which is random.The indices i and j are given to you, so k_i and k_j are not random.

Consider keys $k_1, k_2, ..., k_n$. Let $I_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise} \end{cases}$. Then we have

$$\begin{aligned} E[I_{i,j}] &= Pr\{I_{i,j} = 1\} \\ &= Pr\{h(k_i) = h(k_j)\} \\ &\leq \frac{1}{m} \text{ for any } j \neq i \end{aligned} \tag{1}$$

$$\begin{aligned} E[\#\text{ keys hashing to the same slot as } k_i] &= E[\sum_{j=1}^{n} I_{i,j}] \\ &= \sum_{j=1}^{n} E[I_{i,j}] \text{ (linearity of expectation)} \\ &= \sum_{j \neq i} E[I_{i,j}] + E[I_{i,i}] \\ &\leq \frac{n}{m} + 1 \end{aligned} \tag{2}$$

□

From the above theorem, we know that Insert, Delete, and Search all take $O(1+\alpha)$ expected time. That's all fine, but do there exist any universal hash function families? Yes! And there're many of them.*E.g.*, the set of all hash functions.

All hash functions: $\mathcal{H} = \{\text{all hash functions } h : \{0, 1, ..., u-1\} \to \{0, 1, ..., m-1\}\}$. Apparently, $\mathcal{H}$ is universal, but it is useless. On one hand, storing a single hashing function h takes $\log(m^u) = u\log(m)$ bits $\gg n$ bits. On the other hand, we would need to precompute u values, which takes $\Omega(u)$ time. You could try to avoid storing them all and use a hash table instead, but we'd need to have a hash-table construction in the first place to be able to do this.

Dot-product hash family

Assumptions

- m is a prime
- $u = m^r$ where r is an integer

We want m to be prime for reasons we'll see in a bit. Now m may be doubling if you're using table doubling (remember this from the far reaches of time?). But we have polylog algorithms which can find a nearby prime, etc. We're not going to cover that.

For the second bullet, in real cases, we can always round up m and u to satisfy our requirement. Once we have this, it's going to be pretty natural to view our keys in base m. Now let's view a key as a vector of length r: $k =< k_0, k_1, \ldots, k_{r-1} >$, and $0 \leq k_i < m$. Note that r is going to be pretty small/bounded for most things we'll do, so we can treat it like a constant.

Now, we need to have some way to specify h and what it means to have a random h. So we'll also have another vector $a =< a_0, a_1, a_2, \ldots, a_{r-1} >$, which is also a "key" data type, and will specify the function h.

$$\begin{aligned} h_a(k) &= a \cdot k \bmod m \text{ (dot product)} \\ &= \sum_{i=0}^{r-1} a_i k_i \bmod m \end{aligned} \tag{3}$$

Then our hash family is $\mathcal{H} = \{h_a | a \in \{0, 1, ..., u-1\}\}$. ($a$ is expressed as a vector, as shown above.)

We're going to assume that you can compute this thing, $h_a(k)$ in constant time, since the size of k (and a) is predetermined. The reason we can do this, is the same reason we can do 64 bit stuff in one step – hardware. If you do this explicitly,

$r = log_m u$, so it's kinda loggish time (m, remember, is going to be doubling during this). So we assume we can store the k_is in a constant number of words.

Storing $h_a \in \mathcal{H}$ requires just storing one key, which is a. If we apply "word RAM model", which is manipulating $O(1)$ machine words takes $O(1)$ time and "objects of interest" (here: keys) fit into a machine word, computing $h_a(k)$ takes $O(1)$ time. The summation is very similar to multiplication, but we're getting rid of the carries. So in some sense it's even easier than multiplication.

Theorem: Dot-product hash family $\mathcal{H}$ is universal.

Proof:

So we want to say that if we choose a randomly, the probability of two keys mapping to the same value is at most $1/m$. We're given 2 keys (we have no control over them), but we know they are distinct. If two vectors are distinct, then at least one "digit", say, the d^{th} one, must be distinct.

Take any two keys $k \neq k'$. They must differ in some digits. Say $k_d \neq k_{d'}$. Define $notd = \{0, 1, ..., r-1\} \setminus \{d\}$. Now we have

$$\begin{aligned}
\underset{a}{Pr}\{h_a(k) = h_a(k')\} &= \underset{a}{Pr}\{\sum_{i=0}^{r-1} a_i k_i = \sum_{i=0}^{r-1} a_i k'_i (\text{ mod } m)\} \\
&= \underset{a}{Pr}\{\sum_{i\neq d} a_i k_i + a_d k_d = \sum_{i\neq d} a_i k'_i + a_d k'_d (\text{ mod } m)\} \\
&= \underset{a}{Pr}\{\sum_{i\neq d} a_i (k_i - k'_i) + a_d (k_d - k'_d) = 0 (\text{ mod } m)\} \\
&= \underset{a}{Pr}\{a_d = -(k_d - k'_d)^{-1} \sum_{i\neq d} a_i (k_i - k'_i) (\text{ mod } m)\} \\
&(m \text{ is prime} \Rightarrow \mathbb{Z}_m \text{ has multuplicative inverses}) \\
&= \underset{a_{notd}}{E}[\underset{a_d}{Pr}\{a_d = f(k, k', a_{notd})\}] \\
&(= \sum_x Pr\{a_{notd} = x\} \underset{a_d}{Pr}\{a_d = f(k, k', x)\}) \\
&= \underset{a_{notd}}{E}[\frac{1}{m}] \\
&= \frac{1}{m}
\end{aligned} \tag{4}$$

Math notes: Multiplicative inverses – for every value x, there exists y such that $xy = 1$ mod m, given that m is prime. The expectation thing I did towards the end is just algebra used to isolate a_d, but worth thinking about.

Since we're choosing a_d at random, and this thing is just asking whether a_d is equal to some value (we don't care what that value is, or if it's biased in some way), our

probability is at most $1/m$. So our question is what happens if you take a uniformly random integer (and a_d is part of a key) hitting a particular value, mod m. What's the probability? $1/m$, as long as a_d is chosen independently.

Also notice that we're going to choose a new hash function each time we double our table, becasue of the new value of m.

Another universal hash family from CLRS: Choose prime $p \geq u$ (once). Define $h_{ab}(k) = [(ak + b)(\text{ mod } p)](\text{ mod } m)$. We have $\mathcal{H} = \{h_{ab} | a, b \in \{0, 1, ..., u - 1\}\}$.

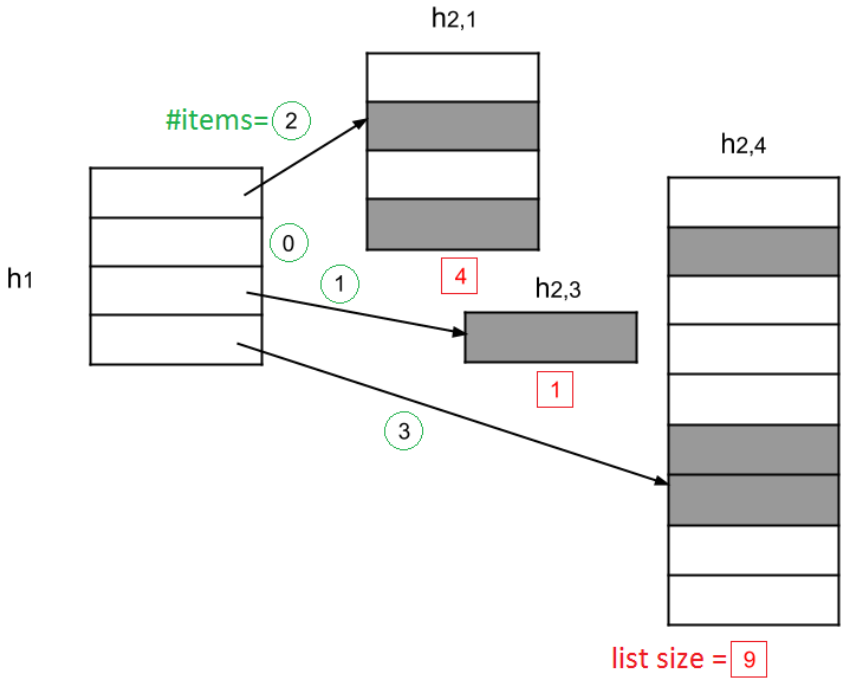
Perfect Hashing

Static dictionary problem: Given n keys to store in table, only need to support search(k). No insertion or deletion will happen.

Perfect hashing: [Fredman, Komlós, Szemerédi 1984]

- polynomial build time with high probability (w.h.p.)
- $O(1)$ time for search in worst case
- $O(n)$ space in worst case

Idea: 2-level hashing



Building this data structure is polytime w.h.p., in practice this will be nearly linear. The basic idea is that instead of using linked-lists for collisions, use... hash-tables!

The algorithm contains the following two major steps:

Step 1: Pick $h_1 : \{0, 1, ..., u-1\} \to \{0, 1, ..., m-1\}$ from a universal hash family for $m = \Theta(n)$ (e.g. nearby prime). Hash all items with chaining using h_1.

Step 2: For each slot $j \in \{0, 1, ..., m-1\}$, let l_j be the # of items in slot j.
(Remember, we know all the items in advance!)
We're going to do something interesting now: rather than having l_j slots in this second-layer table, we're going to have l_j^2! That is, given $l_j = |\{i|h(k_i) = j\}|$, we pick $h_{2,j} : \{0, 1, ..., u-1\} \to \{0, 1, ..., l_j^2 - 1\}$ from a universal hash family for $l_j^2 \leq m_j \leq O(l_j^2)$. (e.g. nearby prime) Replace chain in slot j with hashing-with-chaining using $h_{2,j}$.
"Hash a second time": Why are we making this second table have $m_j = l_j^2$ slots? Because of the *birthday paradox*. If you have n people and n^2 possible birthdays, you have a $1/2$ chance of getting a collision. What we're going to do here is say that there are enough birthdays/slots that the chances of two birthdays or keys colliding is at most $1/2$.

The space complexity is $O(n + \sum_{j=0}^{m-1} l_j^2)$. In order to reduce it to $O(n)$, we need to add two more steps:

Step 1.5: If $\sum_{j=0}^{m-1} l_j^2 > cn$ where c is a constant (we'll figure out what c is later), then redo step 1. That is, after the first step, if we see that the sum of the sizes of the tables will be bigger than linear, do it again.

Step 2.5: If there's a collision in a second-layer table, redo it. That is, while $h_{2,j}(k_i) = h_{2,j}(k_{i'})$ for any $i \neq i', j$, repick $h_{2,j}$ and rehash those l_j. (Of course, the first hash function must already have hashed both k_i and $k_{i'}$ to the same slot)

The two steps above guarantee that there are no collisions at second level, and the space complexity is $O(n)$. As a result, search time is $O(1)$. Now let's look at the build time of the algorithm. Both step 1 and step 2 are $O(n)$. How about step 1.5 and step 2.5? Our main worry is that the redoing actions may need to occur too many times.

For step 2.5, let's do the previously-handwaved birthday paradox bit a little more concretely: what is the probability of two randomly chosen hashes landing on the

same second-layer slot?

$$\begin{aligned}\Pr_{h_{2,j}}\{h_{2,j}(k_i) = h_{2,j}(k_i') \text{ for some } i \neq i'\} &\leq \sum_{i \neq i'} \Pr_{h_{2,j}}\{h_{2,j}(k_i) = h_{2,j}(k_i')\} (\text{union bound}) \\ &\leq \binom{l_j}{2} \times \frac{1}{l_j^2} (\text{recall that } m = l_j^2) \\ &< \frac{1}{2}\end{aligned} \tag{5}$$

As a result, each trial is like a coin flip. If the outcome is "tail", we move to the next step. We have $E[\#\text{trials}] \leq 2$ and $\#\text{trials} = O(\log n)$ w.h.p. (think about why!).

We have to do this (in expectation) twice for each of the secondary tables. What if one of these tables is huge? If it's linear size, we're in trouble. But we can use a Chernoff bound that all the l_is are pretty small, logarithmic, in fact. By using Chernoff bound, $l_j = O(\log n)$ w.h.p., so each trial takes $O(\log n)$ time. (If you do the whole analysis properly, you'll get log/log log, but log is good enough for our purposes.) Since we have to do this for each j, the total time complexity is $O(\log n) \times O(\log n) \times O(n) = O(n \log^2 n)$ w.h.p.

Step 1.5 is a bit more intersting. Since we know that each table has $O(\log n)$ elements, and since the size of each table is the square of the # of elements, (and $m = \Theta(n)$) we could have said we're going to get $n \log^2 n$ size. But we're claiming we can make do with linear space!

Let's first calculate the expectation, and then we'll do a tail bound, *i.e.*, the probability that we are much bigger than the expectation. We'll use an indicator random variable, and linearity of expectation. We define $I_{i,i'} = \begin{cases} 1 & \text{if } h(k_i) = h(k_i') \\ 0 & \text{otherwise} \end{cases}$. Then we have:

$$\begin{aligned}E[\sum_{j=0}^{m-1} l_j^2] &= E[\sum_{i=1}^{n}\sum_{i'=1}^{n} I_{i,i'}] \\ &= \sum_{i=1}^{n}\sum_{i'=1}^{n} E[I_{i,i'}] \text{ (linearlity of expectation)} \\ &\leq n + 2\binom{n}{2} \times \frac{1}{m} \\ &= O(n) \text{ because } m = \Theta(n)\end{aligned} \tag{6}$$

Now, we use Markov and choose the constant c to be twice whatever constant we had in the $O(n)$ above for the expectation, resulting in the probability being at most $1/2$.

By Markov inequality, we have

$$\Pr_{h_1}\{\sum_{j=0}^{m-1} l_j^2 \geq cn\} \leq \frac{E[\sum_{j=0}^{m-1} l_j^2]}{cn} \leq \frac{1}{2}$$

for sufficiently large constant c. By lecture 7, we have $E[\#\text{trials}] \leq 2$ and $\#\text{trials} = O(\log n)$ w.h.p. As a result, step 1 and step 1.5 combined takes $O(n \log n)$ time w.h.p.

So expected time is linear for everything, whp is polylog because of all of those log factors in there.

Math notes: The first expectation is really counting how many collisions we have at the first level, with double counting. That is, each time we hit a slot, we check how many other things have hit that slot and add. Notice how we went from m to n. So, if you have 10 numbers, and 3 go to the same slot, each time you insert one of those 3, you go through all 10 and check how many collisions you get, which will add up to 9. You can also think of a complete graph on l_j items – you'll get a squared number of edges, and we can multiply by two for directed edges.

Later, also notice that we're going to have at least n collisions in this – basically the diagonal of this I matrix. And then we have all the things that don't match (i and j are different) but we have to double count them, so twice n choose 2, multilpied by the probability of a collision, $1/m$. So this whole thing is O(n).