COS 340: Reasoning About Computation[*]
Hashing

Moses Charikar

October 25, 2011

1 Introduction

A hash table is a commonly used data structure to store a set of items, allowing fast inserts, lookups and deletes. Every item consists of a unique identifier called a *key* and a piece of information. For example, the key might be a Social Security Number, a driver's licence number, or an employee ID number. For our purposes, we focus only on the *key*.

Recall that the operations we would like to support are:

1. Insert(k): Insert key k into the hash table.
2. Lookup(k): Check if key k is present in the table.
3. Delete(k): Delete the key k from the table.

Let U be the universe of all keys. For example, U could be the set of all 64 bit strings. In this case $|U| = 2^{64}$. Consider a hash table of size n. The keys are mapped to locations (also called buckets) in the hash table by a hash function $h : U \to [n]$. Multiple keys could map to the same hash bucket. For every bucket in the table, we maintain a linked list of all the keys that map to that bucket. Note that the actual subset of keys stored in the hash table is much smaller than the size of the universe. The hash table size is usually chosen so that the size of the hash table is at least as large as the maximum number of keys we will need to store at any point of time. (If this condition is violated and the number of keys stored grows much larger than the size of the hash table, an implementation will usually increase the size of the table, and recompute the new table from scratch by mapping all keys to the bigger table. Our analyis ignores these complications and assumes that the number of keys is at most the hash table size).

The time required to perform an Insert, Lookup or Delete operation on key k is linear in the length of the linked list for the bucket that key k maps to. (Note that an

[*] Princeton University, Fall 2011.

INSERT could be performed in constant time by always inserting at the head of the list, but we first need to check if key k is already present).

In order for the operations to be implemented efficiently, we would like the keys to be distributed uniformly amongst the buckets in the hash table. We might hope that all buckets have at most a constant number of keys mapped to them, so that all operations could be performed in constant time. What hash function should we pick and what kind of guarantees can we give for running times of the hash table operations ? For any fixed choice hash function h, one can always produce a subset of keys S such that all keys in S are mapped to the same location in the hash table. In this case, the running times of all operations will be linear in the number of keys – far from the constant we were hoping for. Thus, for a fixed hash function h, it is impossible to give worst case guarantees running times on hash table operations.

There are two styles of analysis that we could use to circumvent this problem:

1. Assume that the set of keys stored in the hash table is random, or

2. Assume that the hash function h is random.

Both are plausible alternatives. The problem with the first alternative is that it is hard to justify that the set of keys stored in the hash table is truly random. It would be more satisfying to have an analysis that works for any subset of keys currently in the hash table. In these notes, we will explore the second alternative, i.e assume that the hash function h is random.

What does it mean for h to be random ? One possibility is that h is chosen uniformly and at random from amongst the set of all hash functions $h : U \to [n]$. In fact picking such a hash function is not really practical. Note that there are $n^{|U|}$ possible hash functions. Representing one of these hash functions requires $\log n^{|U|} = |U| \log n$ bits. In fact, this means we need to write down $h(x)$ for every $x \in U$ in order to represent h. That's a lot of storage space ! Much more than the size of the set we are trying to store in the hash table. One could optimize this somewhat by only recording $h(x)$ for all keys x seen so far (and generating $h(x)$ randomly on the fly when a new x is encountered), but this is impractical too. How would we check if a particular key x has already been encountered ? Looks like we would need a hash table for that. But wait, isn't that we set out to implement ? Overall, it is clear that picking a completely random hash function is completely impractical.

Despite this, we will analyze hashing assuming that we have a completely random hash function and then explain how this assumption can be replaced by something that is practical.

1.1 Balls and Bins

A useful abstraction in thinking about hashing with random hash functions is the following experiment: Throw m balls randomly into n bins. (The connection to hashing should be clear: the balls represent the keys and the bins represent the hash buckets). The balls into bins experiment arises in several other problems as well, e.g. analysis of load balacing). In the context of hashing, the following questions arise about the balls and bins experiment:

- How large does m have to be so that with probability greater than 1/2, we have (at least) two balls in the same bin ? This tells us how large our hash table needs to be to avoid any collisions.
- Suppose $m = n$, What is the maximum number of balls that fall into a bin ? This tells us the size of the largest bucket in the hash table when the number of keys is equal to the number of buckets in the table.

No collisions

The first question is related to the so called *birthday paradox* we discussed in class earlier. A quick reminder: Suppose you have 23 people in a room. Then (somewhat surprisingly) the probability that there exists some pair with the same birthday is greater than 1/2 ! (This assumes that birthdays are independent and randomly distributed.) 23 seems like an awfully small number to get a pair with the same birthday. There are 365 days in a year ! How do we explain this ? Consider throwing m balls into n bins. The expected number of pairs that fall into the same bucket is $m(m-1)/2n$. (This follows from linearity of expectation. Note that the probability that a fixed pair falls into the same bucket is $1/n$). Thus the probability that there is a collision is upper bounded by the expected number of collisions which is $m(m-1)/2n$. (If you don't see why this is true right away, take a moment to convince yourself that this is true.) On the other hand, we can also show that the probability that all m balls fall into distinct bins is at most $e^{-m(m-1)/2n}$. (In fact, earlier in class, we went through some elaborate calculations to justify that this is close to the actual probability of having no collisions). For m about $\sqrt{(2\ln 2)n} \approx 1.18\sqrt{n}$ this probability is less than 1/2, i.e. the probability of a collision is greater than 1/2.

This is a useful design principle to keep in mind: If we want to design a hash table with no collisions, then the size of the hash table should be larger than the square of the number of elements we need to store in it. For our purposes in this note, insisting on no collisions means that the number of elements in the hash table can only be a small fraction of the hash table size which is quite wasteful.

The birthday problem calculation is useful in other contexts. Here is an application: Suppose we assign random b bit IDs to m users. How large does b have to be to ensure that all users have distinct IDs with probability $1-\delta$. Here $\delta > 0$ is a given error tolerance. Assigning b bit IDs is identical to mapping to $n = 2^b$ buckets. The birthday calculation shows us that the probability of a collision is at most $m^2/2n = m^2/2^{b+1}$. We should set b large enough such that this bound is at most δ. Thus b should be at least $2\log m - 1 + \log(1/\delta)$.

Maximum bin size

Now lets consider the balls and bins experiment with $m = n$. For a fixed bin B_i, the expected load on B_i (i.e the number of balls that map to B_i) is 1. (This is easy to compute by linearity of expectation. Note that the probability that the any fixed ball ball falls into B_i is $1/n$).

By Chernoff bounds, the probability that the load on B_i exceeds c is at most $e^{c\ln c - c + 1}$. (We will choose an appropriate value for c later). Let A_i be an indicator variable corresponding to the event that bin B_i has at least c balls in it. Then $\Pr[A_i] \leq e^{c \ln c - c + 1}$. We would like to pick c large enough such that with high probability, none of the n bins have load exceeding c. What does high probability mean ? Let's say we would like this event (no bin with large load) to happen with probability at least $1 - 1/n$. In order to do this, we choose c such that $e^{c \ln c - c + 1} \leq 1/n^2$. Applying union bound over the n bins, $\Pr[\cup_i A_i] \leq \sum_{i=1}^n \Pr[A_i]$. Hence the probability that some bin has load exceeding c (i.e the event $\cup_i A_i$) is at most $ne^{c \ln c - c + 1} \leq 1/n$. How large does c need to be ? $c = \ln n$ certainly works (for large enough n), but this is overkill. It turns out that $c = \frac{e \ln n}{\ln \ln n}$ suffices. We conclude that with probability $1 - 1/n$, no bin has load exceeding $\frac{e \ln n}{\ln \ln n}$.

This bound assumes that the balls are assigned to bins at random and these assignments are mutually independent. In class and on the homework, we explored bounds on the maximum load on a bin when balls are assigned to bins by a k-wise independent hash function instead.

1.2 Expected cost for hashing

Let's return to the analysis of hashing, continuing with the (impractical) assumption that the hash function h is completely random. What is the expected cost of performing any of the operations Insert, Lookup or Delete ? Suppose that the keys currently in the hash table are $k_1, \ldots, k_n$. Consider an operation involving key k_i. The cost of the operation is linear in the size of the hash bucket that k_i maps to. Let X be the size of the hash bucket that k_i maps to. X is a random variable and

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{j=1}^{n} \Pr[h(x_i) = h(x_j)] \\
&= 1 + \sum_{j \neq i} \Pr[h(x_i) = h(x_j)] \\
&= 1 + (n-1)/n \leq 2
\end{aligned}$$

Here the last step follows from the fact that $\Pr[h(x_i) = h(x_j)] = 1/n$ when h is random.

Thus the expected cost of any hashing operation is a constant.

1.3 A small random like family

Can we retain the expected cost guarantee of the previous section with a much simpler (i.e. practical) family of hash functions? Let's think about what property of random hash functions we used in the analysis. It turns out that the only fact we used was that $\Pr[h(x_i) = h(x_j)] = 1/n$. Is it possible to construct simpler hash functions with this property?

Thinking along these lines, in 1978, Carter and Wegman introduced the notion of *universal hashing*: Consider a family F of hash functions from U to $[n]$. We say that F is universal if, for a h chosen randomly from F, $\Pr[h(x_i) = h(x_j)] \leq 1/n$.

Clearly the analysis of the previous section shows that for any universal family, the constant expected time guarantee applies. The family of all hash functions is clearly universal. Is there a simpler one ?

It turns out that there is such a family. In order to construct it, we first introduce a clever construction of a pairwise independent hash function. Recall that U denotes the universe of keys (or elements) we want to map to locations in the table. Suppose that the elements of the U are encoded as non-negative integers in the range $\{0, \ldots |U| - 1\}$. Pick a prime $p \geq |U|$. For $a, b \in \{0, \ldots p - 1\}$, consider the family of hash functions $h_{a,b}(x) = ax + b \mod p$. Note that these hash functions have domain and range $\{0, \ldots p - 1\}$.

Consider a hash function h drawn uniformly and at random from this family. We claim that, for any $x_1 \neq x_2$, $h(x_1)$ and $h(x_2)$ are independent. This seems like an amazing property! This implies that for a set of distinct elements $x_1, \ldots x_n$, the values $h(x_1), \ldots h(x_n)$ are pairwise independent.

The proof of this claim is not hard. Consider the pair $(h(x_i), h(x_j))$. For any (y_1, y_2) we will show that there is exactly one h in the family such that $h(x_1) = y_1$ and $h(x_2) = y_2$.

$$\begin{aligned} ax_1 + b &= y_1 \mod p \\ ax_2 + b &= y_2 \mod p \end{aligned}$$

We can solve these equations to find a and b, similar to solving linear equations over the reals. Subtracting the second equation from the first, we get that

$$a(x_1 - x_2) = y_1 - y_2 \mod p$$

This determines a uniquely. Note that $x_1 - x_2 \neq 0$ is required to ensure that there exists a solution to the above equation. Now we can substitute this value of a in any of the two equations to get the value of b. Convince yourself that the values of a, b thus obtained satisfy both the original congruences modulo p.

Note that here are p^2 choices of a and b and consequently p^2 hash functions h in the family. Also we argued that the pair $(h(x_1), h(x_2))$ ranges over all p^2 possible pairs of values (y_1, y_2). For any pair (y_1, y_2), $y_1, y_2 \in \{0, \ldots, p - 1\}$, $Pr[h(x_1) = y_1, h(x_2) = y_2] = 1/p^2$. Thus $h(x_1)$ and $h(x_2)$ are independent. Knowing the value of one reveals no information about the other. Although note that knowing both of them allows us to compute a and b and this determines h completely.

This is a useful family of pairwise independent hash functions to keep in mind and arises in a number of different settings. We will use it to construct a universal hash family. First a slight tweak of this family we just introduced: Notice that the choice of $a = 0$ gives a rather uninteresting family of hash functions $h_{a,b}$. For $a = 0$, $h_{a,b}$ maps all elements to b. Consider instead the family of hash functions $h_{a,b}$ defined before with the additional restriction $a = 0$. There are $p(p - 1)$ such hash functions in this new family. What can we say about the pair $(h(x_1), h(x_2))$ as h ranges over all hash functions in the family ? Similar to the claim we proved before, we can show that, for the new family, the pair $(h(x_1), h(x_2))$ ranges over all $p(p - 1)$ pairs (y_1, y_2) such that $y_1 \neq y_2$. In the proof of the claim before, $a = 0$ only when $y_1 = y_2$.

We aren't quite done yet, since the new family maps numbers in $\{0,\ldots,p-1\}$ to $\{0,\ldots,p-1\}$, but we want to map to one of n buckets. Our final hash family is the following:

$$f_{a,b}(x) = (ax + b \mod p)(\mod n)$$

where $a, b \in \{0,\ldots,p-1\}, a \neq 0$.

We claim that for any $x_1 \neq x_2$, for f chosen randomly from this family, $\Pr[f(x_1) = f(x_2)] \leq 1/n$. In other words, this is a universal family of hash functions, and clearly much simpler than the family of all hash functions !

We prove this here and you will think about a related problem in precept this week.

Theorem 1.1. *Consider the family of hash functions $h_{a,b}(x) = (ax + b \mod p)(\mod n)$ where $a, b \in \{0,\ldots p-1\}$ and $a \neq 0$. Consider a hash function $h_{a,b}$ drawn uniformly and at random from this family. For any $x_1, x_2 \in \{0,\ldots,p-1\}, x_1 \neq x_2$, show that*

$$\Pr[h_{a,b}(x_1) = h_{a,b}(x_2)] \leq 1/n.$$

Proof. Let $f_{a,b}(x) = (ax + b \mod p)$. We showed above that for any $x_1, x_2 \in \{0,\ldots,p-1\}, x_1 \neq x_2$, as a, b range over all values in $\{0,\ldots p-1\}, a \neq 0$, the pair $(f_{a,b}(x_1), f_{a,b}(x_2))$ ranges over all $p(p-1)$ pairs (y_1, y_2) such that $y_1, y_2 \in \{0,\ldots,p-1\}, y_1 \neq y_2$.

Note that $h_{a,b}(x) = f_{a,b}(x) \mod n$. Thus the number of $h_{a,b}$ for which $h_{a,b}(x_1) = h_{a,b}(x_2)$ is exactly the number of pairs (y_1, y_2), $y_1, y_2 \in \{0,\ldots,p-1\}, y_1 \neq y_2$ such that $y_1 = y_2 \mod n$. Let us count the number of such pairs as follows: There are p choices for y_1. For each choice of y_1, there are $\lceil p/n \rceil - 1$ choices of y_2 such that $y_1 = y_2 \mod n$. Now $\lceil p/n \rceil - 1 \leq (p + n - 1)/n - 1 = (p-1)/n$. Hence the number of pairs (y_1, y_2) such that $y_1 = y_2 \mod p$ is at most $p(p-1)/n$. By the above discussion, the number of choices of $h_{a,b}$ such that $h_{a,b}(x_1) = h_{a,b}(x_2)$ is at most $p(p-1)/n$, which is a $1/n$ fraction of the total number of choices of (y_1, y_2). Thus, for a randomly chosen $h_{a,b}$, the probability that $h_{a,b}(x_1) = h_{a,b}(x_2)$ is at most $1/n$. □

Wrapping up the discussion on hashing, if we pick a random hash function from this family, then the expected cost of any hashing operation is constant. Note that picking a random hash function simply involves picking a, b – significantly simpler than picking a completely random hash function.