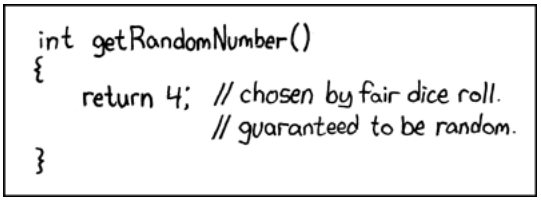
Insanity is repeating the same mistakes and expecting different results.

— *Narcotics Anonymous* (1981)

Calvin: *There! I finished our secret code!*
Hobbes: *Let's see.*
Calvin: *I assigned each letter a totally random number, so the code will be hard to crack. For letter "A", you write 3,004,572,688. "B" is 28,731,569½.*
Hobbes: *That's a good code all right.*
Calvin: *Now we just commit this to memory.*
Calvin: *Did you finish your map of our neighborhood?*
Hoobes: *Not yet. How many bricks does the front walk have?*

— Bill Watterson, "Calvin and Hobbes" (August 23, 1990)



[RFC 1149.5 specifies 4 as the standard IEEE-vetted random number.]

— Randall Munroe, *xkcd* (http://xkcd.com/221/)
Reproduced under a Creative Commons Attribution-NonCommercial 2.5 License

12 Hash Tables

12.1 Introduction

A *hash table* is a data structure for storing a set of items, so that we can quickly determine whether an item is or is not in the set. The basic idea is to pick a *hash function* h that maps every possible item x to a small integer $h(x)$. Then we store x in slot $h(x)$ in an array. The array is the hash table.

Let's be a little more specific. We want to store a set of n items. Each item is an element of a fixed set $\mathcal{U}$ called the *universe*; we use u to denote the size of the universe, which is just the number of items in $\mathcal{U}$. A hash table is an array $T[1..m]$, where m is another positive integer, which we call the *table size*. Typically, m is much smaller than u. A *hash function* is any function of the form

$$h : \mathcal{U} \to \{0, 1, \ldots, m-1\},$$

mapping each possible item in $\mathcal{U}$ to a slot in the hash table. We say that an item x *hashes* to the slot $T[h(x)]$.

Of course, if $u = m$, then we can always just use the trivial hash function $h(x) = x$; n other words, we can use the item itself as the index into the table. This is called a *direct access table*, or more commonly, an *array*. In most applications, though, this approach requires much more space than we can reasonably allocate; on the other hand, we rarely need need to store more than a tiny fraction of $\mathcal{U}$. Ideally, the table size m should be roughly equal to the number n of items we actually want to store.

The downside of using a smaller table is that we must deal with *collisions*. We say that two items x and y *collide* if their hash values are equal: $h(x) = h(y)$. We are now left with two different (but interacting) design decisions. First, how to we choose a hash function h that can

© Copyright 2014 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.

be evaluated quickly and that keeps the number of collisions as small as possible? Second, when collisions do occur, how do we deal with them?

12.2 The Importance of Being Random

If we already knew the precise data set that would be stored in our hash table, it is possible (but not particularly easy) to find a *perfect* hash function that avoids collisions entirely. Unfortunately, for most applications of hashing, we don't know what the user will put into the table. Thus, it is impossible *even in principle* to devise a perfect hash function in advance; no matter what hash function we choose, some pair of items from $\mathcal{U}$ *will* collide. Worse, for any fixed hash function, there is a subset of at least $|U|/m$ items that all hash to the same location. If our input data happens to come from such a subset, either by chance or malicious intent, our code will come to a grinding halt. This is a real security issue with core Internet routers, for example; every router on the Internet backbone survives millions of attacks per day, including timing attacks, from malicious agents.

The *only* way to provably avoid this worst-case behavior is to choose our hash functions *randomly*. Specifically, we will fix a set $\mathcal{H}$ of functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$, and then at run time, we choose our hash function randomly from the set $\mathcal{H}$ according to some fixed distribution. Different sets $\mathcal{H}$ and different distributions over that set imply different theoretical guarantees. Screw this into your brain:

Input data is *not* random!
So good hash functions *must be* random!

In particular, the simple deterministic hash function $h(x) = x \bmod m$, which is often taught and recommended under the name "the division method", is ***utterly stupid***. Many textbooks correctly observe that this hash function is bad when m is a power of 2, because then $h(x)$ is just the low-order bits of m, but then they bizarrely recommend making m prime to avoid such obvious collisions. But even when m is prime, any pair of items whose difference is an integer multiple of m collide with absolute certainty; for all integers a and x, we have $h(x + am) = h(x)$. Why would anyone use a hash function where they *know* certain pairs of keys *always* collide? Sheesh!

12.3 ...But Not Too Random

Most theoretical analysis of hashing assumes ***ideal random*** hash functions. Ideal randomness means that the hash function is chosen ***uniformly*** at random from the set of ***all*** functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$. Intuitively, for each new item x, we roll a new m-sided die to determine the hash value $h(x)$. Ideal randomness is a clean theoretical model, which provides the strongest possible theoretical guarantees.

Unfortunately, ideal random hash functions are a theoretical fantasy; evaluating such a function would require recording values in a separate data structure which we could access using the items in our set, which is exactly what hash tables are for! So instead, we look for families of hash functions with *just enough* randomness to guarantee good performance. Fortunately, most hashing analysis does not actually *require* ideal random hash functions, but only some weaker consequences of ideal randomness.

One property of ideal random hash functions that seems intuitively useful is ***uniformity***. A family $\mathcal{H}$ of hash functions is uniform if choosing a hash function uniformly at random from $\mathcal{H}$ makes every hash value equally likely for every item in the universe:

$$\textbf{Uniform: } \Pr_{h\in\mathcal{H}}\big[h(x)=i\big]=\frac{1}{m} \quad \text{for all } x \text{ and all } i$$

We emphasize that this condition must hold for *every* item $x \in \mathcal{U}$ and *every* index i. Only the hash function h is random.

In fact, despite its intuitive appeal, uniformity is not terribly important or useful by itself. Consider the family $\mathcal{K}$ of *constant* hash functions defined as follows. For each integer a between 0 and $m-1$, let $const_a$ denote the constant function $const_a(x) = a$ for all x, and let $\mathcal{K} = \{const_a \mid 0 \le a \le m-1\}$ be the set of all such functions. It is easy to see that the set $\mathcal{K}$ is both perfectly uniform and utterly useless!

A much more important goal is to minimize the number of collisions. A family of hash functions is ***universal*** if, for any two items in the universe, the probability of collision is as small as possible:

$$\textbf{Universal: } \Pr_{h\in\mathcal{H}}\big[h(x)=h(y)\big]=\frac{1}{m} \quad \text{for all } x \neq y$$

(Trivially, if $x = y$, then $\Pr[h(x) = h(y)] = 1$!) Again, we emphasize that this equation must hold for *every* pair of distinct items; only the function h is random. The family of constant functions is uniform but not universal; on the other hand, universal hash families are not necessarily uniform.[1]

Most elementary hashing analysis requires a weaker versions of universality. A family of hash functions is ***near-universal*** if the probability of collision is *close* to ideal:

$$\textbf{Near-universal: } \Pr_{h\in\mathcal{H}}\big[h(x)=h(y)\big]\le\frac{2}{m} \quad \text{for all } x \neq y$$

There's nothing special about the number 2 in this definition; any other explicit constant will do.

On the other hand, some hashing analysis requires reasoning about larger sets of collisions. For any integer k, we say that a family of hash functions is ***strongly k-universal*** or ***k-uniform*** if for any sequence of k disjoint keys and any sequence of k hash values, the probability that each key maps to the corresponding hash value is $1/m^k$:

$$\textbf{k-uniform: } \Pr\left[\bigwedge_{j=1}^{k} h(x_j)=i_j\right]=\frac{1}{m^k} \quad \text{for all distinct } x_1,\ldots,x_k \text{ and all } i_1,\ldots,i_k$$

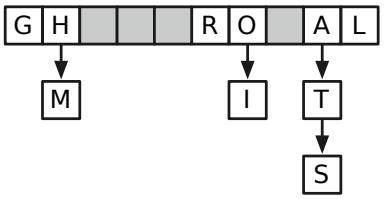
Ideal random hash functions are k-uniform for every positive integer k.

12.4 Chaining

One of the most common methods for resolving collisions in hash tables is called *chaining*. In a *chained* hash table, each entry $T[i]$ is not just a single item, but rather (a pointer to) a linked list of all the items that hash to $T[i]$. Let $\ell(x)$ denote the length of the list $T[h(x)]$. To see if

[1] Confusingly, universality is often called the ***uniform hashing assumption***, even though it is not an assumption that the hash function is uniform.

an item x is in the hash table, we scan the entire list $T[h(x)]$. The worst-case time required to search for x is $O(1)$ to compute $h(x)$ plus $O(1)$ for every element in $T[h(x)]$, or $O(1+\ell(x))$ overall. Inserting and deleting x also take $O(1+\ell(x))$ time.



A chained hash table with load factor 1.

Let's compute the expected value of $\ell(x)$ under this assumption; this will immediately imply a bound on the expected time to search for an item x. To be concrete, let's suppose that x is not already stored in the hash table. For all items x and y, we define the indicator variable

$$C_{x,y} = \big[h(x) = h(y)\big].$$

(In case you've forgotten the bracket notation, $C_{x,y} = 1$ if $h(x) = h(y)$ and $C_{x,y} = 0$ if $h(x) \neq h(y)$.) Since the length of $T[h(x)]$ is precisely equal to the number of items that collide with x, we have

$$\ell(x) = \sum_{y \in T} C_{x,y}.$$

Assuming h is chosen from a ***universal*** set of hash functions, we have

$$\mathrm{E}[C_{x,y}] = \Pr[C_{x,y} = 1] = \begin{cases} 1 & \text{if } x = y \\ 1/m & \text{otherwise} \end{cases}$$

Now we just have to grind through the definitions.

$$\mathrm{E}[\ell(x)] = \sum_{y \in T} \mathrm{E}[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}$$

We call this fraction n/m the *load factor* of the hash table. Since the load factor shows up everywhere, we will give it its own symbol α.

$$\boxed{\alpha := \frac{n}{m}}$$

Similarly, if h is chosen from a ***near***-universal set of hash functions, then $\mathrm{E}[\ell(x)] \leq 2\alpha$. Thus, the expected time for an unsuccessful search in a chained hash table, using near-universal hashing, is $\Theta(1+\alpha)$. As long as the number of items n is only a constant factor bigger than the table size m, the search time is a constant. A similar analysis gives the same expected time bound (with a slightly smaller constant) for a successful search.

Obviously, linked lists are not the only data structure we could use to store the chains; any data structure that can store a set of items will work. For example, if the universe $\mathcal{U}$ has a total ordering, we can store each chain in a balanced binary search tree. This reduces the expected time for any search to $O(1+\log \ell(x))$, and under the simple uniform hashing assumption, the expected time for any search is $O(1+\log \alpha)$.

Another natural possibility is to work recursively! Specifically, for each $T[i]$, we maintain a hash table T_i containing all the items with hash value i. Collisions in those secondary tables are

resolved recursively, by storing secondary overflow lists in tertiary hash tables, and so on. The resulting data structure is a tree of hash tables, whose leaves correspond to items that (at some level of the tree) are hashed without any collisions. If every hash table in this tree has size m, then the expected time for any search is $O(\log_m n)$. In particular, if we set $m = \sqrt{n}$, the expected time for any search is *constant*. On the other hand, there is no inherent reason to use the same hash table size everywhere; after all, hash tables deeper in the tree are storing fewer items.

Caveat Lector! The preceding analysis does *not* imply bounds on the expected *worst-case* search time is constant. The expected worst-case search time is $O(1 + L)$, where $L = \max_x \ell(x)$. Under the uniform hashing assumption, the maximum list size L is *very* likely to grow faster than any constant, unless the load factor α is *significantly* smaller than 1. For example, $\mathrm{E}[L] = \Theta(\log n / \log\log n)$ when $\alpha = 1$. We've stumbled on a powerful but counterintuitive fact about probability: When several individual items are distributed independently and uniformly at random, the resulting distribution is *not* uniform in the traditional sense! Later in this lecture, I'll describe how to achieve constant expected worst-case search time using secondary hash tables.

12.5 Multiplicative Hashing

Perhaps the simplest technique for near-universal hashing, first described by Carter and Wegman in the 1970s, is called ***multiplicative hashing***. I'll describe two variants of multiplicative hashing, one using modular arithmetic with prime numbers, the other using modular arithmetic with powers of two. In both variants, a hash function is specified by an integer parameter a, called a *salt*. The salt is chosen uniformly at random when the hash table is created and remains fixed for the entire lifetime of the table. All probabilities are defined with respect to the random choice of salt.

For any non-negative integer n, let $[\![n]\!]$ denote the n-element set $\{0, 1, \ldots, n-1\}$, and let $[\![n]\!]^+$ denote the $(n-1)$-element set $\{1, 2, \ldots, n-1\}$.

12.5.1 Prime multiplicative hashing

The first family of multiplicative hash function is defined in terms of a prime number $p > |\mathcal{U}|$. For any integer $a \in [\![p]\!]^+$, define a function $multp_a : U \to [\![m]\!]$ by setting

$$\boxed{multp_a(x) = (ax \bmod p) \bmod m}$$

and let

$$\mathcal{MP} := \left\{ multp_a \;\middle|\; a \in [\![p]\!]^+ \right\}$$

denote the set of all such functions. Here, the integer a is the salt for the hash function $multp_a$. We claim that this family of hash functions is universal.

The use of prime modular arithmetic is motivated by the fact that *division* modulo prime numbers is well-defined.

Lemma 1. *For every integer $z \in [\![p]\!]^+$, there is a unique integer $a \in [\![p]\!]^+$ such that $az \bmod p = 1$.*

Proof: Let z be an arbitrary integer in $[\![p]\!]^+$.

Suppose $az \bmod p = bz \bmod p$ for some integers $a, b \in [\![p]\!]^+$. Then $(a-b)z \bmod p = 0$, which means $(a-b)z$ is divisible by p. Because p is prime, the inequality $1 \le z \le p-1$ implies that $a-b$ must be divisible by p. Similarly, the inequality $2-p < a-b < p-2$ implies that a and b must be equal. Thus, for each $z \in [\![p]\!]^+$, there is *at most* one $a \in [\![p]\!]^+$ such that $ax \bmod p = z$.

Similarly, suppose $az \bmod p = 0$ for some integer $a \in [p]^+$. Then because p is prime, either a or z is divisible by p, which is impossible.

We conclude that the set $\{az \bmod p \mid a \in [p]^+\}$ has $p-1$ distinct elements, all non-zero, and therefore is equal to $[p]^+$. In other words, multiplication by z defines a permutation of $[p]^+$. The lemma follows immediately. □

For any integers $x, y \in \mathcal{U}$ and any salt $a \in [p]^+$, we have

$$\begin{aligned} multp_a(x) - multp_a(y) &= (ax \bmod p) \bmod m - (ay \bmod p) \bmod m \\ &= (ax \bmod p - ay \bmod p) \bmod m \\ &= ((ax - ay) \bmod p) \bmod m \\ &= (a(x-y) \bmod p) \bmod m \\ &= multp_a(x-y). \end{aligned}$$

Thus, we have a collision $multp_a(x) = multp_a(y)$ if and only if $multp_a(x-y) = 0$. Thus, to prove that $\mathcal{MP}$ is universal, it suffices to prove the following lemma.

Lemma 2. *For any $z \in [p]^+$, we have $\Pr_a[\text{multp}_a(z) = 0] \le 1/m$.*

Proof: Fix an arbitrary integer $z \in [p]^+$. The previous lemma implies that for any integer $1 \le x \le p-1$, there is a unique integer a such that $(az \bmod p) = x$; specifically, $a = x \cdot z^{-1} \bmod p$. There are exactly $\lfloor (p-1)/m \rfloor$ integers k such that $1 \le km \le p-1$. Thus, there are exactly $\lfloor (p-1)/m \rfloor$ salts a such that $multp_a(z) = 0$. □

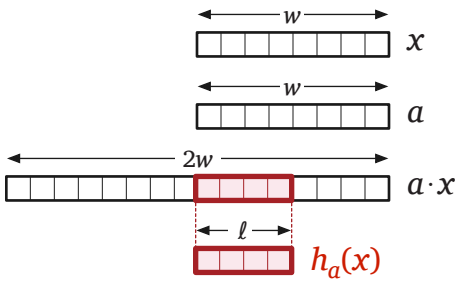
12.5.2 Binary multiplicative hashing

A slightly simpler variant of multiplicative hashing that avoids the need for large prime numbers was first analyzed by Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen in 1997. For this variant, we assume that $\mathcal{U} = [2^w]$ and that $m = 2^\ell$ for some integers w and ℓ. Thus, our goal is to hash w-bit integers ("words") to ℓ-bit integers ("labels").

For any *odd* integer $a \in [2^w]$, we define the hash function $multb_a : \mathcal{U} \to [m]$ as follows:

$$\boxed{multb_a(x) := \left\lfloor \frac{(a \cdot x) \bmod 2^w}{2^{w-\ell}} \right\rfloor}$$

Again, the odd integer a is the salt.



Binary multiplicative hashing.

If we think of any w-bit integer z as an array of bits $z[0..w-1]$, where $z[0]$ is the least significant bit, this function has an easy interpretation. The product $a \cdot x$ is $2w$ bits long; the hash value $multb_a(x)$ consists of the top ℓ bits of the bottom half:

$$multb_a(x) := (a \cdot x)[w-1..w-\ell]$$

Most programming languages automatically perform integer arithmetic modulo some power of two. If we are using an integer type with w bits, the function $multb_a(x)$ can be implemented by a single multiplication followed by a single right-shift. For example, in C:

```
#define hash(a,x) ((a)*(x) >> (WORDSIZE-HASHBITS))
```

Now we claim that the family $\mathcal{MB} := \{multb_a \mid a \text{ is odd}\}$ of all such functions is near-universal. To prove this claim, we again need to argue that division is well-defined, at least for a large subset of possible words. Let W denote the set of odd integers in $[2^w]$.

Lemma 3. *For any integers $x, z \in W$, there is exactly one integer $a \in W$ such that $ax \bmod 2^w = z$.*

Proof: Fix an integer $x \in W$. Suppose $ax \bmod 2^w = bx \bmod 2^w$ for some integers $a, b \in W$. Then $(b-a)x \bmod 2^w = 0$, which means $x(b-a)$ is divisible by 2^w. Because x is odd, $b-a$ must be divisible by 2^w. But $-2^w < b-a < 2^w$, so a and b must be equal. Thus, for each $z \in W$, there is *at most* one $a \in W$ such that $ax \bmod 2^w = z$. In other words, the function $f_x : W \to W$ defined by $f_x(a) := ax \bmod 2^w$ is injective. Every injective function from a finite set to itself is a bijection. □

Lemma 4. *$\mathcal{MB}$ is near-universal.*

Proof: Fix two distinct words $x, y \in \mathcal{U}$ such that $x < y$. If $multb_a(x) = multb_a(y)$, then the top ℓ bits of $a(y-x) \bmod 2^w$ are either all 0s (if $ax \bmod 2^w \le ay \bmod 2^w$) or all 1s (otherwise). Equivalently, if $multb_a(x) = multb_a(y)$, then either $multb_a(y-x) = 0$ or $multb_a(y-x) = m-1$. Thus,

$$\Pr[multb_a(x) = multb_a(y)] \;\le\; \Pr[multb_a(y-x) = 0] \;+\; \Pr[multb_a(y-x) = m-1].$$

We separately bound the terms on the right side of this inequality.

Because $x \neq y$, we can write $(y-x) \bmod 2^w = q2^r$ for some odd integer q and some integer $0 \le r \le w-1$. The previous lemma implies that $aq \bmod 2^w$ consists of $w-1$ random bits followed by a 1. Thus, $aq2^r \bmod 2^w$ consists of $w-r-1$ random bits, followed by a 1, followed by r 0s. There are three cases to consider:

- If $r < w-\ell$, then $multb_a(y-x)$ consists of ℓ random bits, so
$$\Pr[multb_a(y-x) = 0] \;=\; \Pr[multb_a(y-x) = m-1] \;=\; 1/2^\ell.$$
- If $r = w-\ell$, then $multb_a(y-x)$ consists of $\ell-1$ random bits followed by a 1, so
$$\Pr[multb_a(y-x) = 0] = 0 \quad \text{and} \quad \Pr[multb_a(y-x) = m-1] = 2/2^\ell.$$
- Finally, if $r < w-\ell$, then $multb_a(y-x)$ consists of zero or more random bits, followed by a 1, followed by one or more 0s, so
$$\Pr[multb_a(y-x) = 0] \;=\; \Pr[multb_a(y-x) = m-1] \;=\; 0.$$

In all cases, we have $\Pr[multb_a(x) = multb_a(y)] \le 2/2^\ell$, as required. □

*12.6 High Probability Bounds: Balls and Bins

Although any particular search in a chained hash tables requires only constant expected time, but what about the *worst* search time? Assuming that we are using ***ideal random*** hash functions, this question is equivalent to the following more abstract problem. Suppose we toss n balls independently and uniformly at random into one of n bins. Can we say anything about the number of balls in the fullest bin?

Lemma 5. *If n balls are thrown independently and uniformly into n bins, then with high probability, the fullest bin contains $O(\log n / \log\log n)$ balls.*

Proof: Let X_j denote the number of balls in bin j, and let $\hat{X} = \max_j X_j$ be the maximum number of balls in any bin. Clearly, $\mathrm{E}[X_j] = 1$ for all j.

Now consider the probability that bin j contains at least k balls. There are $\binom{n}{k}$ choices for those k balls; each chosen ball has probability $1/n$ of landing in bin j. Thus,

$$\Pr[X_j \geq k] \;=\; \binom{n}{k}\left(\frac{1}{n}\right)^k \;\leq\; \frac{n^k}{k!}\left(\frac{1}{n}\right)^k \;=\; \frac{1}{k!}$$

Setting $k = 2c\lg n / \lg\lg n$, we have

$$k! \geq k^{k/2} = \left(\frac{2c\lg n}{\lg\lg n}\right)^{2c\lg n/\lg\lg n} \geq \left(\sqrt{\lg n}\right)^{2c\lg n/\lg\lg n} = 2^{c\lg n} = n^c,$$

which implies that

$$\Pr\left[X_j \geq \frac{2c\lg n}{\lg\lg n}\right] < \frac{1}{n^c}.$$

This probability bound holds for every bin j. Thus, by the union bound, we conclude that

$$\Pr\Big[\max_j X_j > \frac{2c\lg n}{\lg\lg n}\Big] \;=\; \Pr\Big[X_j > \frac{2c\lg n}{\lg\lg n} \text{ for all } j\Big] \;\leq\; \sum_{j=1}^{n} \Pr\Big[X_j > \frac{2c\lg n}{\lg\lg n}\Big] \;<\; \frac{1}{n^{c-1}}. \qquad \square$$

A somewhat more complicated argument implies that if we throw n balls randomly into n bins, then with high probability, the most popular bin contains at least $\Omega(\log n / \log\log n)$ balls.

However, if we make the hash table large enough, we can expect every ball to land in its own bin. Suppose there are m bins. Let C_{ij} be the indicator variable that equals 1 if and only if $i \neq j$ and ball i and ball j land in the same bin, and let $C = \sum_{i<j} C_{ij}$ be the total number of pairwise collisions. Since the balls are thrown uniformly at random, the probability of a collision is exactly $1/m$, so $\mathrm{E}[C] = \binom{n}{2}/m$. In particular, if $m = n^2$, the expected number of collisions is less than $1/2$.

To get a high probability bound, let X_j denote the number of balls in bin j, as in the previous proof. We can easily bound the probability that bin j is empty, by taking the two most significant terms in a binomial expansion:

$$\Pr[X_j = 0] \;=\; \left(1 - \frac{1}{m}\right)^n \;=\; \sum_{i=1}^{n}\binom{n}{i}\left(\frac{-1}{m}\right)^i \;=\; 1 - \frac{n}{m} + \Theta\left(\frac{n^2}{m^2}\right) \;>\; 1 - \frac{n}{m}$$

We can similarly bound the probability that bin j contains exactly one ball:

$$\Pr[X_j = 1] \;=\; n \cdot \frac{1}{m}\left(1 - \frac{1}{m}\right)^{n-1} \;=\; \frac{n}{m}\left(1 - \frac{n-1}{m} + \Theta\left(\frac{n^2}{m^2}\right)\right) \;>\; \frac{n}{m} - \frac{n(n-1)}{m^2}$$

It follows immediately that $\Pr[X_j > 1] < n(n-1)/m^2$. The union bound now implies that $\Pr[\hat{X} > 1] < n(n-1)/m$. If we set $m = n^{2+\varepsilon}$ for any constant $\varepsilon > 0$, then the probability that no bin contains more than one ball is at least $1 - 1/n^{\varepsilon}$.

Lemma 6. *For any $\varepsilon > 0$, if n balls are thrown independently and uniformly into $n^{2+\varepsilon}$ bins, then with high probability, no bin contains more than one ball.*

We can give a slightly weaker version of this lemma that assumes only near-universal hashing. Suppose we hash n items into a table of size m. Linearity of expectation implies that the expected number of pairwise collisions is

$$\sum_{x<y} \Pr[h(x) = h(y)] \le \binom{n}{2}\frac{2}{m} = \frac{n(n-1)}{m}.$$

In particular, if we set $m = cn^2$, the expected number of collisions is less than $1/c$, which implies that the probability of even a single collision is less than $1/c$.

12.7 Perfect Hashing

So far we are faced with two alternatives. If we use a small hash table to keep the space usage down, even if we use ideal random hash functions, the resulting worst-case expected search time is $\Theta(\log n / \log\log n)$ with high probability, which is not much better than a binary search tree. On the other hand, we can get constant worst-case search time, at least in expectation, by using a table of roughly quadratic size, but that seems unduly wasteful.

Fortunately, there is a fairly simple way to combine these two ideas to get a data structure of linear expected size, whose expected worst-case search time is constant. At the top level, we use a hash table of size $m = n$, but instead of linked lists, we use secondary hash tables to resolve collisions. Specifically, the jth secondary hash table has size $2n_j^2$, where n_j is the number of items whose primary hash value is j. Our earlier analysis implies that with probability at least $1/2$, the secondary hash table has no collisions at all, so the worst-case search time in any secondary hash table is $O(1)$. (If we discover a collision in some secondary hash table, we can simply rebuild that table with a new near-universal hash function.)

Although this data structure apparently needs significantly more memory for each secondary structure, the overall increase in space is insignificant, at least in expectation.

Lemma 7. *Assuming near-universal hashing, we have $\mathrm{E}[\sum_i n_i^2] < 3n$.*

Proof: let $h(x)$ denote the position of x in the primary hash table. We rewrite $\sum_i \mathrm{E}[n_i^2]$ in terms of the indicator variables $[h(x) = i]$ as follows. The first equation uses the definition of n_i; the rest is just routine algebra.

$$\begin{aligned}
\sum_i n_i^2 &= \sum_i \left(\sum_x [h(x) = i] \right)^2 \\
&= \sum_i \left(\sum_x \sum_y [h(x) = i][h(y) = i] \right) \\
&= \sum_i \left(\sum_x [h(x) = i]^2 \;+\; 2 \sum_{x<y} [h(x) = i][h(y) = i] \right) \\
&= \sum_x \sum_i [h(x) = i]^2 \;+\; 2 \sum_{x<y} \sum_i [h(x) = i][h(y) = i] \\
&= \sum_x \sum_i [h(x) = i] \;+\; 2 \sum_{x<y} [h(x) = h(y)]
\end{aligned}$$

The first sum is equal to n, because each item x hashes to exactly one index i, and the second sum is just the number of pairwise collisions. Linearity of expectation immediately implies that

$$\mathrm{E}\left[\sum_i n_i^2 \right] \;=\; n + 2 \sum_{x<y} \Pr[h(x) = h(y)] \;\le\; n + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{2}{n} \;=\; 3n - 2. \qquad \square$$

This lemma immediately implies that the expected size of our two-level hash table is $O(n)$. By our earlier analysis, the expected worst-case search time is $O(1)$.

12.8 Open Addressing

Another method used to resolve collisions in hash tables is called *open addressing*. Here, rather than building secondary data structures, we resolve collisions by looking elsewhere in the table. Specifically, we have a sequence of hash functions $\langle h_0, h_1, h_2, \ldots, h_{m-1} \rangle$, such that for any item x, the *probe sequence* $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x) \rangle$ is a permutation of $\langle 0, 1, 2, \ldots, m-1 \rangle$. In other words, different hash functions in the sequence always map x to different locations in the hash table.

We search for x using the following algorithm, which returns the array index i if $T[i] = x$, 'absent' if x is not in the table but there is an empty slot, and 'full' if x is not in the table and there no no empty slots.

```
OpenAddressSearch(x):
  for i ← 0 to m − 1
      if T[h_i(x)] = x
          return h_i(x)
      else if T[h_i(x)] = ∅
          return 'absent'
  return 'full'
```

The algorithm for inserting a new item into the table is similar; only the second-to-last line is changed to $T[h_i(x)] \leftarrow x$. Notice that for an open-addressed hash table, the load factor is never bigger than 1.

Just as with chaining, we'd like to pretend that the sequence of hash values is truly random, for purposes of analysis. Specifically, most open-addressed hashing analysis uses the following assumption, which is impossible to enforce in practice, but leads to reasonably predictive results for most applications.

Strong uniform hashing assumption:

> For any item x, the probe sequence $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x)\rangle$ is equally likely to be any permutation of the set $\{0, 1, 2, \ldots, m-1\}$.

Let's compute the expected time for an unsuccessful search in light of this assupmtion. Suppose there are currently n elements in the hash table. The strong uniform hashing assumption has two important consequences:

- **Uniformity:** Each hash value $h_i(x)$ is equally likely to be any integer in the set $\{0, 1, 2, \ldots, m-1\}$.
- **Indpendence:** If we ignore the first probe, the remaining probe sequence $\langle h_1(x), h_2(x), \ldots, h_{m-1}(x)\rangle$ is equally likely to be any permutation of the smaller set $\{0, 1, 2, \ldots, m-1\} \setminus \{h_0(x)\}$.

The first sentence implies that the probability that $T[h_0(x)]$ is occupied is exactly n/m. The second sentence implies that if $T[h_0(x)]$ is occupied, *our search algorithm recursively searches the rest of the hash table!* Since the algorithm will never again probe $T[h_0(x)]$, for purposes of analysis, we might as well pretend that slot in the table no longer exists. Thus, we get the following recurrence for the expected number of probes, as a function of m and n:

$$\mathrm{E}[T(m,n)] = 1 + \frac{n}{m}\,\mathrm{E}[T(m-1,n-1)].$$

The trivial base case is $T(m,0) = 1$; if there's nothing in the hash table, the first probe always hits an empty slot. We can now easily prove by induction that $\mathbf{E[}\boldsymbol{T(m,n)}\mathbf{] \le m/(m-n)}$:

$$\begin{aligned}
\mathrm{E}[T(m,n)] &= 1 + \frac{n}{m}\,\mathrm{E}[T(m-1,n-1)] \\
&\le 1 + \frac{n}{m}\cdot\frac{m-1}{m-n} && \text{[induction hypothesis]} \\
&< 1 + \frac{n}{m}\cdot\frac{m}{m-n} && [m-1 < m] \\
&= \frac{m}{m-n}\ \checkmark && \text{[algebra]}
\end{aligned}$$

Rewriting this in terms of the load factor $\alpha = n/m$, we get $\mathbf{E[}\boldsymbol{T(m,n)}\mathbf{] \le 1/(1-\alpha)}$. In other words, the expected time for an unsuccessful search is $O(1)$, unless the hash table is almost completely full.

12.9 Linear and Binary Probing

In practice, however, we can't generate ideal random probe sequences, so we must rely on a simpler probing scheme to resolve collisions. Perhaps the simplest scheme is ***linear probing***—use a single hash function $h(x)$ and define

$$\boxed{h_i(x) := (h(x) + i) \bmod m}$$

This strategy has several advantages, in addition to its obvious simplicity. First, because the probing strategy visits consecutive entries in the has table, linear probing exhibits better cache performance than other strategies. Second, as long as the load factor is strictly less than 1, the expected length of any probe sequence is provably constant; moreover, this performance is guaranteed even for hash functions with limited independence. On the other hand, the number

or probes grows quickly as the load factor approaches 1, because the occupied cells in the hash table tend to cluster together. On the gripping hand, this clustering is arguably an *advantage* of linear probing, since any access to the hash table loads several nearby entries into the cache.

A simple variant of linear probing called ***binary probing*** is slightly easier to analyze. Assume that $m = 2^\ell$ for some integer ℓ (in a binary multiplicative hashing), and define

$$\boxed{h_i(x) := h(x) \oplus i}$$

where $\oplus$ denotes bitwise exclusive-or. This variant of linear probing has slightly better cache performance, because cache lines (and disk pages) usually cover address ranges of the form $[r2^k .. (r+1)2^k - 1]$; assuming the hash table is aligned in memory correctly, binary probing will scan one entire cache line before loading the next one.

Several more complex probing strategies have been proposed in the literature. Two of the most common are ***quadratic probing***, where we use a single hash function h and set $h_i(x) := (h(x) + i^2) \bmod m$, and ***double hashing***, where we use two hash functions h and h' and set $h_i(x) := (h(x) + i \cdot h'(x)) \bmod m$. These methods have some theoretical advantages over linear and binary probing, but they are not as efficient in practice, primarily due to cache effects.

*12.10 Analysis of Binary Probing

Lemma 8. *In a hash table of size $m = 2^\ell$ containing $n \le m/4$ keys, built using binary probing, the expected time for any search is $O(1)$,* **assuming ideal random hashing**.

Proof: The hash table is an array $H[0..m-1]$. For each integer k between 0 and ℓ, we partition H into $m/2^k$ ***level-k blocks*** of length 2^k; each level-k block has the form $H[c2^k .. (c+1)2^k - 1]$ for some integer c. Each level-k block contains exactly two level-$(k-1)$ blocks; thus, the blocks implicitly define a complete binary tree of depth ℓ.

Now suppose we want to search for a key x. For any integer k, let $B_k(x)$ denote the range of indices for the level-k block containing $H[h(x)]$:

$$B_k(x) = \left[2^k \lfloor h(x)/2^k \rfloor .. 2^k \lfloor h(x)/2^k \rfloor + 2^k - 1\right]$$

Similarly, let $B'_k(x)$ denote the sibling of $B_k(x)$ in the block tree; that is, $B'_k(x) = B_{k+1}(x) \setminus B_k(x)$. We refer to each $B_k(x)$ as an ***ancestor*** of x and each $B'_k(x)$ as an ***uncle*** of x. The proper ancestors of any uncle of x are also proper ancestors of x.

The binary probing algorithm can be recast conservatively as follows:

```
BinaryProbe(x) :
  if H[h(x)] = x
      return True
  if H[h(x)] is empty
      return False

  for k = 0 to ℓ − 1
      for each index j in B'_k(x)
      if H[j] = x
          return True
      if H[j] is empty
          return False
```

For purposes of analysis, suppose the target item x is not in the table. (The time to search for an item that is in the table can only be faster.) Then the expected running time of BINARYPROBE(x) can be expressed as follows:

$$\mathrm{E}[T(x)] \le \sum_{k=0}^{\ell-1} O(2^k) \cdot \Pr[B'_k(x) \text{ is full}].$$

Assuming ideal random hashing, all blocks at the same level have equal probability of being full. Let F_k denote the probability that a fixed level-k block is full. Then we have

$$\mathrm{E}[T(x)] \le \sum_{k=0}^{\ell-1} O(2^k) \cdot F_k.$$

Call a level-k block B ***popular*** if there are at least 2^k items y in the table such that $h(y) \in B$. Every popular block is full, but full blocks are not necessarily popular.

If block $B_k(x)$ is full but not popular, then $B_k(x)$ contains at least one item whose hash value is not in $B_k(x)$. Let y be the first such item inserted into the hash table. When y was inserted, some uncle block $B'_j(x) = B_j(y)$ with $j \ge k$ was already full. Let $B'_j(x)$ be the first uncle of $B_k(x)$ to become full. The only blocks that can overflow into $B_j(y)$ are its uncles, which are all either ancestors or uncles of $B_k(x)$. But when $B_j(y)$ became full, no other uncle of $B_k(x)$ was full. Moreover, $B_k(x)$ was not yet full (because there was still room for y), so no ancestor of $B_k(x)$ was full. It follows that $B'_j(x)$ is popular.

We conclude that if a block is full, then either that block or one of its uncles is popular. Thus, if we write P_k to denote the probability that a fixed level-k block is popular, we have

$$F_k \le 2P_k + \sum_{j>k} P_j.$$

We can crudely bound the probability P_k as follows. Each of the n items in the table hashes into a fixed level-k block with probability $2^k/m$; thus,

$$P_k = \binom{n}{2^k}\left(\frac{2^k}{m}\right)^{2^k} \le \frac{n^{2^k}}{(2^k)!}\frac{2^{k2^k}}{m^{2^k}} < \left(\frac{en}{m}\right)^{2^k}$$

(The last inequality uses a crude form of Stirling's approximation: $n! > n^n/e^n$.) Our assumption $n \le m/4$ implies the simpler inequality $P_k < (e/4)^{2^k}$. Because $e < 4$, it is easy to see that $\boldsymbol{P_k < 4^{-k}}$ for all sufficiently large k.

It follows that $F_k = O(4^{-k})$, which implies that the expected search time is at most $\sum_{k\ge0} O(2^k) \cdot O(4^{-k}) = \sum_{k\ge0} O(2^{-k}) = O(1)$. □

12.11 Cuckoo Hashing

★★★ Write this.

Exercises

1. Your boss wants you to find a *perfect* hash function for mapping a known set of n items into a table of size m. A hash function is *perfect* if there are *no* collisions; each of the n items

is mapped to a different slot in the hash table. Of course, a perfect hash function is only possible if $m \geq n$. (This is a different definition of "perfect" than the one considered in the lecture notes.) After cursing your algorithms instructor for not teaching you about (this kind of) perfect hashing, you decide to try something simple: repeatedly pick ideal random hash functions until you find one that happens to be perfect.

(a) Suppose you pick an ideal random hash function h. What is the *exact* expected number of collisions, as a function of n (the number of items) and m (the size of the table)? Don't worry about how to resolve collisions; just count them.

(b) What is the *exact* probability that a random hash function is perfect?

(c) What is the *exact* expected number of different random hash functions you have to test before you find a perfect hash function?

(d) What is the *exact* probability that none of the first N random hash functions you try is perfect?

(e) How many ideal random hash functions do you have to test to find a perfect hash function *with high probability*?

2. (a) Describe a set of hash functions that is uniform but not (near-)universal.

(b) Describe a set of hash functions that is universal but not (near-)uniform.

(c) Describe a set of hash functions that is universal but (near-)3-universal.

(d) A family of hash function is ***pairwise independent*** if knowing the hash value of any one item gives us absolutely no information about the hash value of any other item; more formally,

$$\Pr_{h \in \mathcal{H}}[h(x) = i \mid h(y) = j] = \Pr_{h \in \mathcal{H}}[h(x) = i]$$

or equivalently,

$$\Pr_{h \in \mathcal{H}}[(h(x) = i) \wedge (h(y) = j)] = \Pr_{h \in \mathcal{H}}[h(x) = i] \cdot \Pr_{h \in \mathcal{H}}[h(y) = j]$$

for all distinct items $x \neq y$ and all (possibly equal) hash values i and j.

Describe a set of hash functions that is uniform but not pairwise independent.

(e) Describe a set of hash functions that is pairwise independent but not (near-)uniform.

(f) Describe a set of hash functions that is universal but not pairwise independent.

(g) Describe a set of hash functions that is pairwise independent but not (near-)uniform.

(h) Describe a set of hash functions that is universal and pairwise independent but not uniform, or prove no such set exists.

3. (a) Prove that the set $\mathcal{MB}$ of binary multiplicative hash functions described in Section 12.5 is not uniform. *[Hint: What is* $\text{multb}_a(0)$*?]*

(b) Prove that $\mathcal{MB}$ is not pairwise independent. *[Hint: Compare* $\text{multb}_a(0)$ *and* $\text{multb}_a(2^{w-1})$*.]*

(c) Consider the following variant of multiplicative hashing, which uses slightly longer salt parameters. For any integers $a, b \in [2^{w+\ell}]$ where a is odd, let

$$h_{a,b}(x) := \big((a \cdot x + b) \bmod 2^{w+\ell}\big) \operatorname{div} 2^w = \left\lfloor \frac{(a \cdot x + b) \bmod 2^{w+\ell}}{2^w} \right\rfloor,$$

and let $\mathcal{MB}^+ = \{h_{a,b} \mid a, b \in [2^{w+\ell}] \text{ and } a \text{ odd}\}$. Prove that the family of hash functions $\mathcal{MB}^+$ is ***strongly near-universal***:

$$\Pr_{h \in \mathcal{MB}^+}\big[(h(x) = i) \wedge (h(y) = j)\big] \le \frac{2}{m^2}$$

for all items $x \neq y$ and all (possibly equal) hash values i and j.

4. Suppose we are using an *open-addressed* hash table of size m to store n items, where $n \le m/2$. Assume an ideal random hash function. For any i, let X_i denote the number of probes required for the ith insertion into the table, and let $X = \max_i X_i$ denote the length of the longest probe sequence.

   (a) Prove that $\Pr[X_i > k] \le 1/2^k$ for all i and k.

   (b) Prove that $\Pr[X_i > 2 \lg n] \le 1/n^2$ for all i.

   (c) Prove that $\Pr[X > 2 \lg n] \le 1/n$.

   (d) Prove that $\mathrm{E}[X] = O(\log n)$.

© Copyright 2014 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms for the most recent revision.