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)

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 some finite[1] 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

$$h\colon \mathcal{U} \to \{0, 1, \dots, m-1\},$$

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 $h(x) = x$. In other words, 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, the universe of possible keys is orders of magnitude too large for this approach to be practical. Even when it is possible to allocate enough memory, we usually need to store only a small fraction of the universe. Rather than wasting lots of space, we should make m roughly equal to n, the number of items in the set we want to maintain.

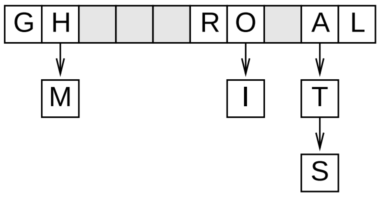
What we'd like is for every item in our set to hash to a different position in the array. Unfortunately, unless $m = u$, this is too much to hope for, so we have to deal with *collisions*. We say that two items x and y *collide* if the have the same hash value: $h(x) = h(y)$. Since we obviously can't store two items in the same slot of an array, we need to describe some methods for *resolving* collisions. The two most common methods are called *chaining* and *open addressing*.

12.2 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 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

[1]This finiteness assumption is necessary for several of the technical details to work out, but can be ignored in practice. To hash elements from an infinite universe (for example, the positive integers), pretend that the universe is actually finite but very very large. In fact, in *real* practice, the universe actually *is* finite but very very large. For example, on most modern computers, there are only 2^{64} integers (unless you use a big integer package like GMP, in which case the number of integers is closer to $2^{2^{32}}$.)

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.

In the worst case, every item would be hashed to the same value, so we'd get just one long list of n items. In principle, for any deterministic hashing scheme, a malicious adversary can always present a set of items with exactly this property. In order to defeat such malicious behavior, we'd like to use a hash function that is as random as possible. Choosing a truly random hash function is completely impractical, but there are several heuristics for producing hash functions that behave randomly, or at least close to randomly on real data. Thus, we will analyze the performance as though our hash function were truly random. More formally, we make the following assumption.

Simple uniform hashing assumption: $\boxed{\text{If } x \neq y \text{ then } \Pr\big[h(x)=h(y)\big]=1/m.}$

In the next section, I'll describe a small set of functions with the property that a random hash function in this set satisfies the simple uniform hashing assumption. Most actual implementations of has tables use *deterministic* hash functions. These clearly violate the uniform hashing assumption—the collision probability is either 0 or 1, depending on the pair of items! Nevertheless, it is common practice to adopt the uniform hashing assumption as a convenient fiction for purposes of analysis.

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

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

We can rewrite the simple uniform hashing assumption as follows:

$$x \neq y \implies \mathrm{E}[C_{x,y}] = \Pr[C_{x,y}=1] = \frac{1}{m}.$$

Now we just have to grind through the definitions.

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

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.

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

Our analysis implies that the expected time for an unsuccessful search in a chained hash table is $\Theta(1+\alpha)$. As long as the number if 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![2] 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.

12.3 Universal Hashing

Now I'll describe a method to generate random hash functions that satisfy the simple uniform hashing assumption. We say that a set $\mathcal{H}$ of hash function is *universal* if it satisfies the following property: For any items $x \neq y$, if a hash function h is chosen *uniformly at random* from the set $\mathcal{H}$, then $\Pr[h(x) = h(y)] = 1/m$. Note that this probability holds for *any* items x and y; the randomness is entirely in choosing a hash function from the set $\mathcal{H}$.

To simplify the following discussion, I'll assume that the universe $\mathcal{U}$ contains exactly m^2 items, each represented as a pair (x, x') of integers between 0 and $m-1$. (Think of the items as two-digit numbers in base m.) I will also assume that m is a prime number.

For any integers $0 \le a, b \le m-1$, define the function $h_{a,b} \colon \mathcal{U} \to \{0, 1, \dots, m-1\}$ as follows:

$$h_{a,b}(x, x') = (ax + bx') \bmod m.$$

Then the set

$$\mathcal{H} = \left\{ h_{a,b} \;\middle|\; 0 \le a, b \le m-1 \right\}$$

of all such functions is universal. To prove this, we need to show that for any pair of distinct items $(x, x') \neq (y, y')$, exactly m of the m^2 functions in $\mathcal{H}$ cause a collision.

[2]Latin for "Beware of cannibals."

Choose two items $(x, x') \neq (y, y')$, and assume without loss of generality[3] that $x \neq y$. A function $h_{a,b} \in \mathcal{H}$ causes a collision between (x, x') and (y, y') if and only if

$$\begin{aligned} h_{a,b}(x, x') &= h_{a,b}(y, y') \\ (ax + bx') \bmod m &= (ay + by') \bmod m \\ ax + bx' &\equiv ay + by' \pmod{m} \\ a(x - y) &\equiv b(y' - x') \pmod{m} \\ a &\equiv \frac{b(y' - x')}{x - y} \pmod{m}. \end{aligned}$$

In the last step, we are using the fact that m is prime and $x - y \neq 0$, which implies that $x - y$ has a unique multiplicative inverse modulo m. (For example, the multiplicative inverse of 12 modulo 17 is 10, since $12 \cdot 10 = 120 \equiv 1 \pmod{17}$.) For each possible value of b, the last identity defines a *unique* value of a such that $h_{a,b}$ causes a collision. Since there are m possible values for b, there are exactly m hash functions $h_{a,b}$ that cause a collision, which is exactly what we needed to prove.

Thus, if we want to achieve the constant expected time bounds described earlier, we should choose a random element of $\mathcal{H}$ as our hash function, by generating two numbers a and b uniformly at random between 0 and $m - 1$. This is *precisely* the same as choosing a element of $\mathcal{U}$ uniformly at random.

One perhaps undesirable 'feature' of this construction is that we have a small chance of choosing the trivial hash function $h_{0,0}$, which maps everything to 0. So in practice, if we happen to pick $a = b = 0$, we should reject that choice and pick new random numbers. By taking $h_{0,0}$ out of consideration, we reduce the probability of a collision from $1/m$ to $(m-1)/(m^2-1) = 1/(m+1)$. In other words, the set $\mathcal{H} \setminus \{h_{0,0}\}$ is slightly *better* than universal.

This construction can be easily generalized to larger universes. Suppose $u = m^r$ for some constant r, so that each element $x \in \mathcal{U}$ can be represented by a vector $(x_0, x_1, \ldots, x_{r-1})$ of integers between 0 and $m - 1$. (Think of x as an r-digit number written in base m.) Then for each vector $a = (a_0, a_1, \ldots, a_{r-1})$, define the corresponding hash function h_a as follows:

$$h_a(x) = (a_0 x_0 + a_1 x_1 + \cdots + a_{r-1} x_{r-1}) \bmod m.$$

Then the set of all m^r such functions is universal.

*12.4 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? Under a stronger version[4] of the uniform hashing assumption, this 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 1. *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.*

[3]'Without loss of generality' is a phrase that appears (perhaps too) often in combinatorial proofs. What it means is that we are considering one of many possible cases, but once we see the proof for one case, the proofs for all the other cases are obvious thanks to some inherent symmetry. For this proof, we are not explicitly considering what happens when $x = y$ and $x' \neq y'$.

[4]The simple uniform hashing assumption requires only *pairwise* independence, but the following analysis requires *full* independence.

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 exactly k balls. There are $\binom{n}{k}$ choices for those k balls; each chosen ball has probability $1/n$ of landing in bin j; and each of the remaining balls has probability $1 - 1/n$ of missing bin j. Thus,

$$\begin{aligned}\Pr[X_j = k] &= \binom{n}{k}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k}\\ &\le \frac{n^k}{k!}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k}\\ &< \frac{1}{k!}\end{aligned}$$

This bound shrinks super-exponentially as k increases, so we can *very* crudely bound the probability that bin 1 contains *at least* k balls as follows:

$$\Pr[X_j \ge k] < \frac{n}{k!}$$

To prove the high-probability bound, we need to choose a value for k such that $n/k! \approx 1/n^c$ for some constant c. Taking logs of both sides of the desired approximation and applying Stirling's approximation, we find

$$\begin{aligned}\ln k! \approx k\ln k - k &\approx (c+1)\ln n\\ \iff k &\approx \frac{(c+1)\ln n}{\ln k - 1}\\ &\approx \frac{(c+1)\ln n}{\ln\frac{(c+1)\ln n}{\ln k-1} - 1}\\ &= \frac{(c+1)\ln n}{\ln\ln n + \ln(c+1) - \ln(\ln k - 1) - 1}\\ &\approx \frac{(c+1)\ln n}{\ln\ln n}.\end{aligned}$$

We have shown (modulo some algebraic hand-waving that is easy but tedious to clean up) that

$$\Pr\left[X_j \ge \frac{(c+1)\ln n}{\ln\ln n}\right] < \frac{1}{n^c}.$$

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

$$\begin{aligned}\Pr\left[\max_j X_j > \frac{(c+1)\ln n}{\ln\ln n}\right] &= \Pr\left[X_j > \frac{(c+1)\ln n}{\ln\ln n} \text{ for all } j\right]\\ &\le \sum_{j=1}^{n}\Pr\left[X_j > \frac{(c+1)\ln n}{\ln\ln n}\right]\\ &< \frac{1}{n^{c-1}}.\end{aligned}$$

□

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 2. *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.*

12.5 Perfect Hashing

So far we are faced with two alternatives. If we use a small hash table, to keep the space usage down, 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 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 n, but instead of linked lists, we use secondary hash tables to resolve collisions. Specifically, the jth secondary hash table has size n_j^2, where n_j is the number of items whose primary hash value is j. The expected worst-case search time in any secondary hash table is $O(1)$, by our earlier analysis.

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

Lemma 3. *The simple uniform hashing assumption implies* $\mathrm{E}[n_j^2] < 2$.

Proof: Let X_{ij} be the indicator variable that equals 1 if item i hashes to slot j in the primary hash table. Linearity of expectation implies that

$$\mathrm{E}[n_j^2] \;=\; \mathrm{E}\left[\sum_{i=1}^{n}\sum_{k=1}^{n} X_{ij}X_{kj}\right] \;=\; \sum_{i=1}^{n}\sum_{k=1}^{n} \mathrm{E}[X_{ij}X_{kj}] \;=\; \sum_{i=1}^{n} \mathrm{E}[X_{ij}^2] + 2\sum_{i=1}^{n}\sum_{k=i+1}^{n} \mathrm{E}[X_{ij}X_{kj}].$$

Because X_{ij} is an indicator variable, we have $X_{ij}^2 = X_{ij}$, which implies that $\mathrm{E}[X_{ij}^2] = \mathrm{E}[X_{ij}] = 1/n$ by the uniform hashing assumption. The uniform hashing assumption also implies that X_{ij} and X_{kj} are independent whenever $i \neq k$, so $\mathrm{E}[X_{ij}X_{kj}] = \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{kj}] = 1/n^2$. Thus,

$$\mathrm{E}[n_j^2] \;=\; \sum_{i=1}^{n} \frac{1}{n} + 2\sum_{i=1}^{n}\sum_{k=i+1}^{n} \frac{1}{n^2} \;=\; 1 + 2\binom{n}{2}\frac{1}{n^2} \;=\; 2 - \frac{1}{n}. \qquad \Box$$

This lemma 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 $O(1)$.

12.6 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 $\langle h_0, h_1, h_2, \dots, h_{m-1}\rangle$, such that for any item x, the *probe sequence* $\langle h_0(x), h_1(x), \dots, h_{m-1}(x)\rangle$ is a permutation of $\langle 0, 1, 2, \dots, m-1\rangle$. 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 $i \gets 0$ to $m-1$
      if $T[h_i(x)] = x$
          return $h_i(x)$
      else if $T[h_i(x)] = \varnothing$
          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 $T[h_i(x)] \gets x$. 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 random. For purposes of analysis, there is a stronger uniform hashing assumption that gives us constant expected search and insertion time.

Strong uniform hashing assumption:

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

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

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

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
EMPHE$[T(m,n)] \le m/(m-n)$:

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

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

In practice, however, we can't generate truly random probe sequences, so instead we use one of the following probing strategies:

- **Linear probing:** *Use a single hash function $h(x)$, and define $h_i(x) = (h(x) + i) \bmod m$.*

  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 $\alpha \le 1/4$ (for example), 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, performance degrades quickly as the load factor approaches 1, because the occupied cells in the hash table tend to cluster together.

- **Quadratic probing:** *Use a single hash function $h(x)$, and define $h_i(x) = (h(x) + i^2) \bmod m$.*

  This slightly more complex strategy avoids the poor clustering behavior of linear probing, although it suffer from a weaker clustering problem: If two items have the same initial hash value, their entire probe sequences are identical. A more significant limitation of quadratic probing is that the sequence of hash values $\langle h_i(x)\rangle$ hits at most half of the entries in the table if the table size m is prime, and possibly even less if m is composite. Thus, once the hash table is more than half full, quadratic probing is not even guaranteed to terminate.

- **Double hashing:** *Use two hash functions $h(x)$ and $h'(x)$, and let $h_i(x) = (h(x) + i \cdot h'(x)) \bmod m$.*

  To guarantee that the probe sequence visits every slot in the table, the *stride* function $h'(x)$ and the table size m must be relatively prime. We can guarantee this by making m prime, but a simpler solution is to make m a power of 2 and choose a stride function that is always odd. Double hashing avoids the clustering problems of linear and quadratic probing; however, this gain is arguably offset by worse cache behavior. For sufficiently large tables and sufficiently independent hash functions h and h, the performance of double hashing in practice is almost the same as predicted by the uniform hashing assumption.

12.7 Cuckoo Hashing

To be written!

Exercises

1. Suppose we are using an *open-addressed* hash table of size m to store n items, where $n \le m/2$. Assume that the strong uniform hashing assumption holds. 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 $\mathrm{E}[X] = O(\lg n)$.

2. 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, this requires that $m \ge n$. (This is a different definition of perfect hashing than the one considered in the lecture notes.) After cursing your algorithms instructor for not teaching you about perfect hashing, you decide to try something simple: repeatedly pick random hash functions until you find one that happens to be perfect. A random hash function h satisfies two properties:

   - $\Pr[h(x) = h(y)] = 1/m$ for any pair of items $x \ne y$.
   - $\Pr[h(x) = i] = 1/m$ for any item x and any integer $1 \le i \le m$.

   (a) Suppose you pick a 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 random hash functions do you have to test to find a perfect hash function *with high probability*?

★3. Recall that F_k denotes the kth Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for all $k \ge 2$. Suppose we are building a hash table of size $m = F_k$ using the hash function

$$h(x) = (F_{k-1} \cdot x) \bmod F_k$$

Prove that if the consecutive integers $0, 1, 2, \ldots, F_k - 1$ are inserted in order into an initially empty table, each integer will be hashed into the largest contiguous empty interval in the table. In particular, show that there are no collisions.