Aitch	*Ex*
Are	*Eye*
Ay	*Gee*
Bee	*Jay*
Cue	*Kay*
Dee	*Oh*
Double U	*Pea*
Ee	*See*
Ef	*Tee*
El	*Vee*
Em	*Wy*
En	*Yu*
Ess	*Zee*

— Sidney Harris, "The Alphabet in Alphabetical Order"

6 Hash Tables (September 24)

6.1 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 a function

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

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

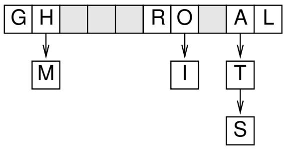
Of course, if $u = m$, then we can always just use the trivial hash function $h(x) = x$. In other words, use the item itself as the index into the table. This is called a *direct access table* (or more commonly, an *array*). In most applications, though, the universe of possible keys is orders of magnitude 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*.

[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 $2^{2^{32}}$.)

6.2 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 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 since there are several heuristics for producing hash functions that behave close to randomly (on real data), we will analyze the performance *as though our hash function were completely random.* More formally, we make the following assumption.

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

Let's compute the expected value of $\ell(x)$ under the simple uniform hashing 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:

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

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.

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

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[2] 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 worst-case time for a search to $O(1 + \log \ell(x))$, and under the simple uniform hashing assumption, the expected time for a search is $O(1 + \log \alpha)$.

Another possibility is to keep the overflow lists in hash tables! Specifically, for each $T[i]$, we maintain a hash table T_i containing all the items with hash value i. To keep things efficient, we make sure the load factor of each secondary hash table is always a constant less than 1; this can be done with only constant *amortized* overhead.[3] Since the load factor is constant, a search in any secondary table always takes $O(1)$ expected time, so the total expected time to search in the top-level hash table is also $O(1)$.

6.3 Open Addressing

Another method we can use to resolve collisions 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 the sequence of hash values to be random, and 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 $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x)\rangle$ is equally likely to be any permutation of the set $\{0, 1, 2, \ldots, m-1\}$.

[2]but with smaller constants hidden in the $O()$—see p.225 of CLR for details.

[3]This means that a single insertion or deletion may take more than constant time, but the total time to handle any sequence of k insertions of deletions, for any k, is $O(k)$ time. We'll discuss amortized running times after the first midterm. This particular result will be an easy homework problem.

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 $\{0, 1, 2, \ldots, m-1\}$.
- 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 $\boxed{\mathrm{E}[T(m,n)] \le m/(m-n)}$:

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

Rewriting this in terms of the load factor $\alpha = n/m$, we get $\boxed{\mathrm{E}[T(m,n)] \le 1/(1-\alpha)}$. 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 we use one of the following heuristics:

- **Linear probing:** We use a single hash function $h(x)$, and define $h_i(x) = (h(x) + i) \bmod m$. This is nice and simple, but collisions tend to make items in the table clump together badly, so this is not really a good idea.
- **Quadratic probing:** We use a single hash function $h(x)$, and define $h_i(x) = (h(x) + i^2) \bmod m$. Unfortunately, for certain values of m, the sequence of hash values $\langle h_i(x)\rangle$ does not hit every possible slot in the table; we can avoid this problem by making m a prime number. (That's often a good idea anyway.) Although quadratic probing does not suffer from the same clumping problems as linear probing, it does have a weaker clustering problem: If two items have the same initial hash value, their entire probe sequences will be the same.
- **Double hashing:** We use two hash functions $h(x)$ and $h'(x)$, and define h_i as follows:

  $$h_i(x) = (h(x) + i \cdot h'(x)) \bmod m$$

  To guarantee that this can hit 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. In fact, the actual performance of double hashing is almost the same as predicted by the uniform hashing assumption, at least when m is large and the component hash functions h and h' are sufficiently random. This is the method of choice![4]

6.4 Deleting from an Open-Addressed Hash Table

Deleting an item x from an open-addressed hash table is a bit more difficult than in a chained hash table. We can't simply clear out the slot in the table, because we may need to know that $T[h(x)]$ is occupied in order to find some other item!

Instead, we should delete more or less the way we did with scapegoat trees. When we delete an item, we mark the slot that used to contain it as a *wasted* slot. A sufficiently long sequence of insertions and deletions could eventually fill the table with marks, leaving little room for any real data and causing searches to take linear time.

However, we can still get good *amortized* performance by using two rebuilding rules. First, if the number of items in the hash table exceeds $m/4$, double the size of the table ($m \leftarrow 2m$) and rehash everything. Second, if the number of wasted slots exceeds $m/2$, clear all the marks and rehash everything in the table. Rehashing everything takes m steps to create the new hash table and $O(n)$ expected steps to hash each of the n items. By charging a \$4 tax for each insertion and a \$2 tax for each deletion, we expect to have enough money to pay for any rebuilding.

In conclusion, the *expected amortized* cost of any insertion or deletion is $O(1)$, under the uniform hashing assumption. Notice that we're doing two very different kinds of averaging here. On the one hand, we are averaging the possible costs of *each individual search* over all possible probe sequences ('expected'). On the other hand, we are also averaging the costs of *the entire sequence* of operations to 'smooth out' the cost of rebuilding ('amortized'). Both randomization and amortization are necessary to get this constant time bound.

6.5 Universal Hashing

Now I'll describe how to generate hash functions that (at least in expectation) satisfy the 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 $0 \leq a, b \leq m-1$, define the function $h_{a,b}\colon \mathcal{U} \to \{0, 1, \ldots, m-1\}$ as follows:

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

Then the set

$$\mathcal{H} = \{h_{a,b} \mid 0 \leq a, b \leq m-1\}$$

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.

[4]...unless your hash tables are really huge, in which case linear probing has *far* better cache behavior, especially when the load factor is small.

Choose two items $(x, x') \neq (y, y')$, and assume without loss of generality[5] 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.[6] Now notice 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 $m - 1$. (Notice that this is *exactly* 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 generalized easily 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.

[5]'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 $x' \neq y'$.

[6]For example, the multiplicative inverse of 12 modulo 17 is 10, since $12 \cdot 10 = 120 \equiv 1 \pmod{17}$.