Contents

1 Open-address hash tables

Open-address hash tables deal differently with collisions. Instead of using a list to chain items whose keys collide, in open-addressing we attempt to find an alternative location in the hash table for the keys that collide. Concretely, if we cannot place key k at location $h(k, 0)$ in the hash table, we try the next location given by $h(k, 1)$ (and so on).

In *open-addressing*, a hash function of the form $h_m : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$ is used. Remember that in chaining the hash function was $h_m : U \to \{0, 1, \ldots, m-1\}$.

As a result, now we can reason about all the locations in the hash table where a key k can be placed. We call this sequence of locations the *probe sequence* of key k, and it is given by:

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

With this new type of hash function comes a new *uniform hashing assumption* (UHA) that is slightly different than the *simple uniform hashing assumption* (SUHA).

The new *uniform hashing assumption* says that "Each key k is equally likely to hash to any of the $m!$ permutations of $\langle 0, 1, \ldots, (m-1) \rangle$." Stated differently, the probe sequence of any key k is equally likely to be any of the $m!$ permutations of the m locations in the hash table.

Note: This is a much stronger statement than simply stating that $\forall i, h(k, i)$ is equally likely to be any value in $\langle 0, 1, \ldots, (m-1) \rangle$. The uniform hashing assumption tells us that the probe sequence **is a permutation**, while this last statement does not.

1.1 Deleting from an open-address hash table

Insertions in an open-address hash-table are pretty straightforward: $\forall i \in \{0, 1, \ldots, m-1\}$, try inserting the new key k at location $h(k, i)$ in the hash table. If you've exhausted all possible m locations, then the hash table is full or your hash function is ill-formed (i.e. it does not output a permutation) and you're stuck in a *hashing loop* where you try previous locations multiple times.

Searches follow the same idea: First look for the key k in $h(k, 0)$. If you find a different key k' at location $h(k, 0)$, then look in $h(k, 1)$. Keep looking (go through $h(k, 2), h(k, 3), \ldots, h(k, m-1)$) until you either find k, **OR** you've looked through m locations, **OR** you find an empty location in the hash table.

- If you've looked through all possible m locations given by the probe sequence of k and you did not find k, then k is clearly not in the table. Otherwise, it would have been found at one of these locations.
- If you've found an empty location, then k is not in the table. Why? Because k should have been in this location if it did not appear in any of the earlier ones. (i.e. Why would k skip an empty location when it was inserted in the hash table? It would have been inserted at this empty location that we just found).

This last point about empty locations might have raised a few eyebrows. "But what about deletes?" you might say. "Isn't it possible that we deleted a key k' from the hash-table, that appeared before k in k's probe sequence? What if we search for k now? We are not going to find it. We will find the empty location where k' was and give up.".

Wow, what an astute student! This, and a few other scenarios, are why we must handle DELETE's carefully in an open-address hash-table. Instead of leaving the location empty after deleting a key k, we will mark that location with a special DELETED constant. This indicates that the DELETED location is indeed *free* but also that a SEARCH operation should continue going through the probe sequence as if there was a key in the DELETED location.

What about INSERT? Should we INSERT in a location with a DELETED item? Suppose we:

- INSERT keys 1, 2, 3 and 4. They hash to locations 0, 1, 2 and 3 respectively.
- DELETE key 2. Now location 1 in the array is empty.
- INSERT key 4, whose first location in the probe sequence happens to be location 1.
  - If we allow inserting over DELETED locations, then we'll reinsert key 4 at location 1, instead of updating its value at location 3. Now we have duplicate keys in the hash-table, which is **not good**.
  - Thus, it follows that before inserting a key k in the hash table, we must first SEARCH for it. If k was already inserted, then we just update the value associated with it. Otherwise, we insert k at the first *free* location found in the probe sequence. **Note:** The first *free* location could be marked as DELETED, but now we know k is not in the hash table so we can safely place it there.

Long story short: Always SEARCH(k) before an INSERT(k, v) to ensure that k is not already in the hash table. If k is in the hash table, just update the value associated with k to v.

Implemented carefully, this special DELETED constant saves us from the cases mentioned above.

Next, we show how open addressing can be implemented with various types of hash functions which results in different kinds of probing sequences.

1.2 Linear probing

Linear probing is the simplest way of implementing *open-addressing*. Unfortunately, it's also the slowest way too because it does not distribute they keys very well throughout the table.

Suppose you had an *auxiliary hash function* $h'(k)$, where $h : U \to \{0, 1, \dots, m-1\}$. You can use h' to build a new hash function $h_m : U \times \{0, 1, \dots, m-1\} \to \{0, 1, \dots, m-1\}$.

$$h_m(k) = (h'(k) + i) \bmod m$$

First, observe that $\langle h_m(k, 0), h_m(k, 1), \dots, h_m(k, m-1)\rangle$ will give you a permutation on $\langle 0, 1, \dots, m-1\rangle$. Thus, we can use h_m to implement an open-addressing scheme.

Second, notice that when keys collide, *linear probing* tries consecutive locations one by one until it finds a free one (wrapping around if it reaches the end of the array). This creates long runs of occupied locations that tend to get even longer when new keys are inserted. These long runs increase both the INSERT time and the SEARCH time in the hash table.

1.3 Double hashing

Double hashing does not suffer from the clustering problem that *linear probing* exhibits. With double hashing we are given two *auxiliary hash functions* h_1 and h_2 and we build h_m as follows:

$$h_m = (h_1(k) + i \cdot h_2(k)) \bmod m$$

In *double hashing*, we multiply the *probe number* i by the output of another hash function which means the next probe in the sequence could be some random location in the hash-table, most likely not adjacent to the previous probe.

Of course, we are only interested in **good** h_2's. If $h_2(k) = 1, \forall k$, then we are back to linear probing. In addition, we want the probe sequence $\langle h_m(k, 0), h_m(k, 1), \dots, h_m(k, m-1)\rangle$ to be a permutation on $\langle 0, 1, \dots, m-1\rangle$. How can we guarantee that?

Note that if h_m did not give us a permutation on $\langle 0, 1, \dots, m-1\rangle$, then there exist $i \neq j$ such that $h_m(k, i) = h_m(k, j)$. This means that for a certain key k, we will search the same location twice in k's probe sequence given by h_m. This means that,

$$\begin{align} (h_1(k) + i \cdot h_2(k)) \bmod m = (h_1(k) + j \cdot h_2(k)) \bmod m \Leftrightarrow \tag{1} \\ i \cdot h_2(k) \bmod m = j \cdot h_2(k) \bmod m \Leftrightarrow \tag{2} \\ i \bmod m = j \bmod m \Leftrightarrow \tag{3} \\ i - j \equiv 0 \pmod m \Leftrightarrow \tag{4} \\ m \text{ divides } i - j \tag{5} \end{align}$$

But this cannot be true because $i - j$ is less than m and greater than $-m$ and there are no numbers in that range divisible by m. Other than zero of course. But in that case $i = j$, which is not a collision. Remember we conditioned on $i \neq j$.

Keep reading for the good stuff: For this to work though, m and $h_2(k)$ have to be relatively prime, $\forall k$. Otherwise, we could not go from Equation (2) to Equation (4) in the proof by contradiction shown above because $h_2(k)$ would not have an inverse modulo m.

Now, how do you pick the right m and the right $h_2(k)$? We can let $m = 2^p$ and ensure $h_2(k)$ is odd for all k. This way, m and $h_2(k)$ will be relatively prime: there is no number that divides both m and $h_2(k)$ because m is only divisible by powers of 2 and $h_2(k)$ is odd.

On a final note, know that double hashing comes pretty close in practice to satisfying the *uniform hashing assumption*.

1.4 Analysis: Expected number of probes for inserting a new key

Suppose we wanted to INSERT a **new key** k in an open-address hash table of capacity m with n keys already in it. We will show that under the uniform hashing assumption this takes time $\leq \frac{1}{1-\alpha}$, where $\alpha = n/m$ is the *load factor* of the hash table.

Proof:
In the INSERT call, we will probe for a free location for k using our hash function h. The probe sequence will be:

$$h(k, 0), h(k, 1), \ldots, h(k, m-1)$$

Note: This analysis assumes that k is a *new* key, so it is **not** in the hash table. Take note of this while reading the proof below.

Now we will look at the probabilities that we can find a free location for k in the hash table on the i'th probe.

We have probability $\frac{m-n}{m}$ of finding an empty location on the first probe when we compute $h(k,0)$. This is because, for a key that's not in the table, there are $m-n$ *free* locations that $h(k,0)$ could hash to out of the total m locations and $h(k,0)$ is uniformly distributed on $[0, m-1]$ (UHA).

Thus,

$$P(\text{found free location on 1st probe}) \geq \frac{m-n}{m} = p$$

If we collided on the first probe and we search the next location $h(k,1)$, the probability of finding a free location (given that a collision has occurred at $h(k,0)$) is:

$$P(\text{found free location on 2nd probe} \mid \text{1st probe failed}) = \frac{m-n}{m-1} \geq \frac{m-n}{m} = p$$

Why? After colliding on the first probe, we still have $m-n$ free locations out of the remaining $m-1$ locations that we can search in the table. There are $m-1$ locations left to search because $h(k,1)$ will never collide with $h(k,0)$ since the probe sequence is a permutation of $\langle 0, 1, \ldots m-1 \rangle$, which means we will never try the same location twice.

Similarly, the probability of finding a free location on the ith probe given that we collided on all the previous $i-1$ probes will be:

$$P(\text{found free location on } i\text{th probe} \mid \text{all previous probes failed}) = \frac{m-n}{m-i+1} \geq \frac{m-n}{m} = p$$

Thus, every probe has probability $\geq p$ of finding a free location for the key k.

The *geometric distribution* tells us that the *expected* number of trials needed to obtain a success when the probability of success in each trial is p is $1/p$.

Thus, the *expected* number of probes needed to insert a new key k in our hash table will be:

$$\frac{1}{p} = \frac{1}{\frac{m-n}{m}} = \frac{m}{m-n} = \frac{1}{1-n/m} = \frac{1}{1-\alpha}$$

2 Cryptographic Hash Functions

Before we begin discussing cryptography, let's define a new term: *infeasible*. When we say, for example that given some information X, it is *infeasible* to compute information Y, we mean that while figuring out Y may be *technically possible*, it make no economic sense to do so because the cost (in terms of time, power, money, work, etc.) greatly outweighs the benefit. **Example:** If the expected time to compute something is greater than the expected remaining life of the Sun, such a computation is certainly *infeasible*.

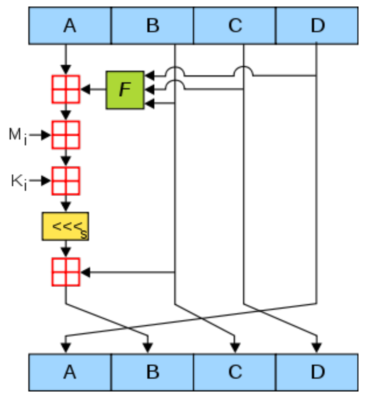
We use the term *cryptographic hash function* to describe a hash function of the form $h(k)$ with certain properties we have not considered before, namely:

- **One-Way (OW)**: Given h, and a value y such that $h(k) = y$ for some (unknown) key k, it is *infeasible* to find k or any other key k' that hashes to y. In English, this means that the hash function is not feasibly invertible, and finding any key that hashes to a specific value is very hard.
- **Collision-Resistant (CR)**: Given h, it is infeasible to find any two keys k_1, k_2 such that $h(k_1) = h(k_2)$. In English, this means that finding two keys with an identical hash is ridiculously difficult. A weaker form of this statement is also useful: **Target collision-resistance (TCR)** states that given h and some key k_1, it is infeasible to find any key k_2 such that $h(k_1) = h(k_2)$. This statement is weaker than **CR** because **CR** trivially implies **TCR**, but not the other way around.

Cryptographic hash functions are hash functions designed with the above properties in mind. Before deferring this topic to other (more focused) courses, consider one concrete example of a cryptographic hash function **MD5**

MD5 takes a variable-length message M, and partitions it into a sequence $M = [M_0, M_1, ...M_i]$ of consecutive 512-bit "chunks". Let M_i be a 512-bit chunk of M. Let $[A, B, C, D] = M_i$ be a sequence of 128-bit consecutive chunks within M_i. Let K be a set of carefully-chosen constants that help make MD5 a cryptographic hash. Let F be a set of some well-chosen non-linear functions. MD5 is defined by 64 rounds of the operation chosen, with a different choice of F for each round.

Below is one of the 64 "rounds" of the MD5 algorithm:



As you can see, the hash function is a bit arbitrary, and somewhat complicated. Designing cryptographic hash functions is a *black art*. This means that $99.99999...\%$ of the time it is better to use an *existing* one, than to roll your own.