Universal hashing

No matter how we choose our hash function, it is always possible to devise a set of keys that will hash to the same slot, making the hash scheme perform poorly.

To circumvent this, we *randomize* the choice of a hash function from a carefully designed set of functions. Let U be the set of universe keys and $\mathcal{H}$ be a finite collection of hash functions mapping U into $\{0, 1, \ldots, m-1\}$. Then $\mathcal{H}$ is called *universal* if, for $x, y \in U, (x \neq y)$,

$$|\{h \in \mathcal{H} : h(x) = h(y)\}| = \frac{|\mathcal{H}|}{m} \quad .$$

In other words, the probability of a collision for two different keys x and y given a hash function randomly chosen from $\mathcal{H}$ is $1/m$.

Theorem. If h is chosen from a universal class of hash functions and is used to hash n keys into a table of size m, where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

Universal hashing (2)

How can we create a set of universal hash functions? One possibility is as follows:

1. Choose the table size m to be prime.
2. Decompose the key x into $r+1$ "bytes" so that $x = \langle x_0, x_1, \ldots, x_r \rangle$, where the maximal value of any x_i is less than m.
3. Let $a = \langle a_0, a_1, \ldots, a_r \rangle$ denote a sequence of $r+1$ elements chosen randomly such that $a_i \in \{0, 1, \ldots, m-1\}$. There are m^{r+1} possible such sequences.
4. Define a hash function h_a with $h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m$.
5. $\mathcal{H} = \bigcup_a \{h_a\}$ with m^{r+1} members, one for each possible sequence a.

Universal hashing (3)

Theorem.
The class $\mathcal{H}$ defined above defines a universal class of hash functions.

Universal hashing (4)

Proof. Consider any pair of distinct keys x and y and assume $h(x) = h(y)$ as well as w.l.o.g. $x_0 \neq y_0$. Then for any fixed $\langle a_1, a_2, \ldots, a_r \rangle$ it holds:

$$\sum_{i=0}^{r} a_i x_i \mod m = \sum_{i=0}^{r} a_i y_i \mod m \quad .$$

Hence:

$$\sum_{i=0}^{r} a_i (x_i - y_i) \mod m = 0$$

Hence:

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i x_i \mod m \quad .$$

Note that m is prime and $(x_0 - y_0)$ is non-zero, hence it has a (unique) multiplicative inverse modulo m. Multiplying both sides of the equation with this inverse yields:

$$a_0 \equiv -\sum_{i=1}^{r}(a_i x_i) \cdot (x_0 - y_0)^{-1} \mod m \ .$$

and there is a unique $a_0 \mod m$ which allows $h(x) = h(y)$.

Each pair of keys x and y collides for exactly m^r values of a, once for each possible value of $\langle a_1, a_2, \ldots, a_r \rangle$. Hence, out of m^{r+1} combinations of $a_0, a_1, a_2, \ldots, a_r$, there are exactly m^r collisions of x and y, and hence the probability that x and y collide is $m^r / m^{r+1} = 1/m$. Hence $\mathcal{H}$ is universal. ■

Open addressing

The idea of open addressing is to trade table size for pointers. All elements are directly stored in the hash table.

To perform an insertion we now *probe* the hash table for an empty slot in some systematic way. Instead of using a fixed order, the sequence of positions probed depends on the key to be inserted.

The hash function is redefined as

$$h : U \times \{0, 1, \ldots, m-1\} \mapsto \{0, 1, \ldots, m-1\}$$

For every key k the probe sequence

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

is considered. If no free position is found in the sequence the hash table overflows.

Open addressing (2)

The main problem with open addressing is the deletion of elements. We cannot simply set an element to *NIL*, since this could break a probe sequence for other elements in the table.

It is possible to use a special purpose marker instead of *NIL* when an element is removed. However, using this approach the search time is no longer dependent on the load factor α. Because of those reasons, open-address hashing is usually not done when delete operations are required.

Probe sequences

In the analysis of open addressing we make the assumption of *uniform hashing*.

To compute the probe sequences there are three different techniques commonly used.

1. linear probing
2. quadratic probing
3. double hashing

These techniques guarantee that $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ for each k, but none fullfills the assumption of uniform hashing, since none can generate more than m^2 sequences.

Probe sequences (2)

Given $h' : U \mapsto \{0, 1 \ldots, m-1\}$, *linear probing* uses the hash function:

$$h(k, i) = (h'(k) + i) \mod m \qquad \text{for } i = 0, 1, \ldots, m-1 \ .$$

Given key k, the first slot probed is $T[h'(k)]$ then $T[h'(k)+1]$ and so on. Hence, the first probe determines the remaining probe sequence.

This methods is easy to implement but suffers from *primary clustering*, that is, two hash keys that hash to different locations compete with each other for successive rehashes. Hence, long runs of occupied slots build up, increasing search time.

Probe sequences (3)

For example, if we have $n = m/2$ keys in the table, where every even-indexed slot is occupied and every odd-indexed slot is free, then the average search time takes 1.5 probes.

If the first $n = m/2$ locations are the ones occupied, however, the average number of probes increases to $n/4 = m/8$.

Probe sequences (4)

Clusters are likely to arise, since if an empty slot is preceded by i full slots, then the probability that the empty slot is the next one filled is $(i+1)/m$ compared with the probability of $1/m$ if the preceding slot was empty.

Thus, runs of occupied slots tend to get longer, and linear probing is not a very good approximation to uniform hashing.

Probe sequences (5)

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m \qquad \text{for } i = 0, 1, \ldots, m-1 \ ,$$

where $h' : U \mapsto \{0, 1 \ldots, m-1\}$ is an auxiliary hash function and $c_1, c_2 \neq 0$ auxiliary constants. Note that c_1 and c_2 must be carefully choosen.

Quadratic probing is better than linear probing, because it spreads subsequent probes out from the initial probe position. However, when two keys have the same initial probe position, their probe sequences are the same, a phenomenon known as *secondary clustering*.

Probe sequences (6)

Double hashing is one of the best open addressing methods, because the permutations produced have many characteristics of randomly chosen permutations. It uses a hash function of the form

$$h(k, i) = (h_1(k) + i h_2(k)) \mod m \qquad \text{for } i = 0, 1, \ldots, m-1 \ ,$$

where h_1 and h_2 are auxiliary hash functions.

The initial position probed is $T[h_1(k) \mod m]$, with successive positions offset by the amount $i h_2(k) \mod m$. Now keys with the same initial probe position can have different probe sequences.

Probe sequences (7)

Note that $h_2(k)$ must be relatively prime to m for the entire hash table to be accessible for insertion and search. Or, to put it differently, if $d = gcd(h_2(k), m) > 1$ for some key k, then the search for key k would only access $1/d$-th of the table.

A convenient way to ensure that $h_2(k)$ is relatively prime to m is to select m as a power of 2 and design h_2 to produce an odd positive integer. Or, select a prime m and let h_2 produce a positive integer less than m.

Double hashing is an improvement over linear and quadratic probing in that $\Theta(m^2)$ sequences are used rather than $\Theta(m)$ since every $(h_1(k), h_2(k))$ pair yields a distinct probe sequence, and the initial probe position, $h_1(k)$, and offset $h_2(k)$ vary independently.

Analysis of open addressing

Theorem.
Given an open address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$, assuming simple uniform hashing.

Analysis of open addressing (2)

Proof. Define $p_i = Pr($ exactly i probes access occupied slots $)$ for $i = 0, 1, 2, \ldots$ (Note that for $i > n$, $p_i = 0$). The expected number of probes is then $1 + \sum_{i=0}^{\infty} i \cdot p_i$. Now define $q_i = Pr($ at least i probes access occupied slots), then $\sum_{i=0}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} q_i$ (why? (exercise)).

The probability that the first probes accesses an occupied slot is $\frac{n}{m}$, so $q_1 = \frac{n}{m}$. A second probe, if needed, will access one of the remaining $m-1$ locations which contain $n-1$ possible keys, so $q_2 = \frac{n}{m} \cdot \frac{n-1}{m-1}$. Hence for $i = 1, 2, \ldots, n$

$$q_i = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \left(\frac{n}{m}\right)^i = \alpha^i.$$

Hence the following holds:

$$1 + \sum_{i=0}^{\infty} i \cdot p_i = 1 + \sum_{i=1}^{\infty} q_i \leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots = \frac{1}{1-\alpha} \quad . \qquad \blacksquare$$

Analysis of open addressing

Hence, if the table is half full, at most 2 probes will be required on average, but if it is 80% full, then on average up to 5 probes are needed.

Corollary. Inserting an item into an open-address hash table with load factor α requires at most $\frac{1}{1-\alpha}$ probes on average, assuming uniform hashing.

Proof. An insert operation amounts to an unsuccessful search followed by a placement of the key in the first empty slot found. Thus, the expected number of probes equals the one for unsuccessful search.

Analysis of open addressing

Theorem. Given an open address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Analysis of open addressing (2)

Proof. A successful search has the same probe sequence as when the element was inserted. Averaging this time over all elements yields:

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} \\
&= \frac{m}{n}\sum_{i=m-n+1}^{m}\frac{1}{i} \\
&\leq \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}
\end{aligned}$$

■

Hence, if the table is half full, the expected number of probes in a successful search is $\frac{1}{0.5}\ln\frac{1}{0.5} = 1.387$.

Perfect Hashing

The ultimate combination of the the ideas presented above leads to *perfect hashing*.

In (static) perfect hashing we can achieve a worst case search time of $O(1)$ while using only $O(n)$ space. This is achieved by a clever two step hashing scheme similar to the double hashing scheme in open adressing.

The idea is as follows. One uses a first hash function to hash the n keys to a table of size $O(n)$, and then hashes all elements n_j that are in the same table slot to a secondary hash table of size $O(n_j^2)$. Allocating enough space this scheme guarantees, that we can find in a constant number of steps a hash function without collision while still using linear space.

This sounds too good to be true, but here is the argument:

Perfect Hashing (2)

A table of size n^2 makes it easy to find a perfect hash function.

Theorem 1. *If we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions, then the probability of there being any collisions is less than $1/2$.*

Proof: There are $\binom{n}{2}$ pairs that could collide, each with prob $1/m = 1/n^2$. The probability of having at least one collision is bounded by the sum of the probabilities of those collisions. Hence $Pr(\text{any collision}) \leq \binom{n}{2}\frac{1}{n^2} = \frac{n(n-1)}{2n^2} \leq \frac{1}{2}$.

Hence we just need to repeatedly and randomly pick a hash function until we find one without collisions. The expected number of times we need to test is a small constant.

Perfect Hashing (3)

What is the space consumption for the two way scheme? First, we use a table of size n for the first universal hash function. Now let n_j be the number of keys that hash to bucket j, we will then allocate n_j^2 space for each bucket. Then we expect to need space

$$\begin{aligned}
E(\sum_{j=0}^{n-1} n_j^2) &= E(\sum_{j=0}^{m-1} n_j) + 2E(\sum_{j=0}^{m-1}\binom{n_j}{2}) \\
&= n + 2E(\#\text{ collisions}) \\
&= n + 2\binom{n}{2}\frac{1}{m} \\
&\leq n + (n-1) \leq 2n
\end{aligned}$$

This is a rough argument. Making the odds higher and counting more precisely it is convenient and works with $6n$.

Perfect Hashing (4)

The hash function used in perfect hashing is of the form $h_k(x) = (kx \bmod p) \bmod s$, where p is a prime. It was introduced and analyzed in the paper of Fredman, Komlós, and Szemerédi in 1984. A proof that it is universal is similar to the one conducted in the lecture.

We give now here an example of the two stage hashing scheme. Assume that $p = 31, n = 6$ and $\mathcal{S} = \{2, 4, 5, 15, 18\}$. We try out a number of hashfunctions and find $k = 2$ sufficient, that means, the overall space consumption is linear. We allocate for each table two slots more and store the value k and n_j in the first two positions.

This gives the following picture:

Perfect Hashing (5)

In the example we show the primary table and the secondary tables which are allocated in a consecutive piece of memory.

```
        k
        0   1   2   3   4    5    6
        2       7      10   16   22

  7  8  9| 10 11 12 13 14 15| 16 17 18 19 20 21| 22 23 24
  1  1  4|  2  1     5  2   |  2  3 30       18|  1  1 15
 n2 k2   | n4 k4            | n5 k5             | n6 k6
```

The query for 30 is processed as follows:

1. $k = T[0] = 2$, $j = (30 \cdot 2 \bmod 31) \bmod 6 = 5$.
2. $T[5] = 16$, and from cells $T[16]$ and $T[17]$ we learn that block 5 has two elements and that $k_3 = 3$
3. $(30 \cdot 3 \bmod 31) \bmod 2^2 = 0$. Hence we check the $0 + 2 = 2$th cell of block 5 and find that 30 is indeed present.

Perfect Hashing (6)

Mehlhorn et al showed that you can also use a simple doubling technique in conjunction with static perfect hashing, such that you can construct a dynamic hash table that support insertion, deletion and lookup time in expected, amortized time $O(1)$.