

Hashing

1 Hashing

In applications one often has to maintain key-value pairs (x, z) . To simplify matters a bit, we ignore the value fields and simply discuss how to maintain a subset S of some fixed universe U of size N . The dictionary operations insert, delete, and search should all be constant time. For simplicity we assume that U is just $\{0, 1, \dots, N - 1\}$.

In the simplest case N is reasonably small, say, around 2^8 . Then we can use a bit-vector

```
boolean bv[N];
```

Clearly all operations are $O(1)$. A direct-address table is a slight generalization of this idea: replace the boolean values by pointers to the actual objects; a null pointer indicates absence of the element.

1.1 Hash Tables

If N becomes larger bit-vectors and direct-address tables fail. Instead use a small table of size m and store object x in slot $h(x)$ of the table. In order to assign slots in the table to elements in U we need a *hash function* of the form

$$h : U \rightarrow \{0, \dots, m - 1\}.$$

h must be easy to compute and should distribute the elements of the universe evenly over the hash table. We will sometimes use the von Neumann notation $m = \{0, \dots, m - 1\}$.

Key parameters:

- N , the size of the universe (huge),
- n , the number of elements in S ,
- m , the size of the hash table,
- $\alpha = n/m$, the load factor.

Note, though, that since the universe has size (much) larger than m there is always the possibility of *collisions*: we may have $h(x) = h(y)$ for some $x \neq y$. As a matter of fact, in the worst case, all elements in S may hash to the same slot.

There are two basic methods to resolve these collisions:

- Chaining: keep elements in containers (bins) hanging off the actual table.
- Open Addressing: place elements into the table, moving to different slots if the original one is already taken.

More below, first some comments about hash functions.

1.2 Hash Functions

The requirements for a good hash function are somewhat contradictory: on the one hand it should behave just like a random function and distribute all the keys evenly over the table, but on the other hand it must be perfectly deterministic and easily computable (no large hidden tables anywhere). If the data are already random there is no problem, but if the data are highly regular (say, an arithmetic progression) we have to be careful to introduce enough chaos to get good distribution over the table. The “Handbook of Algorithms and Datastructures” by G. H. Gonnet and R. Baeza-Yates has lots of background information on various methods.

Here are some very standard methods.

Division Method

A particularly simple type of hash function uses modular arithmetic

$$h(x) = x \bmod m$$

where the modulus m is a prime: primality ensures reasonable distribution (compare this to, say, a modulus $m = 2^8$).

Some libraries such as the STL have a table of primes hard-coded. The primes are chosen to be close to 2^i for $i = 5, \dots, 20$ so that one can nearly double the size of the table whenever necessary.

Multiplication Method

Let $0 < r < 1$ be irrational. Then $\{i \cdot r \bmod 1 \mid i < n\}$ is very evenly distributed over the interval $[0, 1]$. Gives rise to a hash function

$$h(x) = \lfloor m(xr \bmod 1) \rfloor.$$

A typical choice is $r = (\sqrt{5} - 1)/2 \approx 0.618033988749894813$, and $m = 2^k$. Note that implementation is a bit problematic here, we cannot directly deal with irrationals in the light of the obvious efficiency constraints for hash functions.

Bit Operations

Alternatively, one can use a variety of bit-operations (shifting, logical ops) to obtain hash values.

For example, the following scheme uses a pre-initialize table that holds a permutation of the 256 bytes and then uses

```
char hash( char *key, int len, char perm[256])
{
    char h;
    int i, len;
    for( i=0, h=0; i < len; i++ )
        h = perm[ h^key[i] ];
    return h;
}
```

In the next example the low-order byte of the table are initialized to a permutation of the 256 bytes.

```
int hash( char *key, int len, int perm[256])
{
```

```

int h, i;
for( h=len, i=0; i<len; i++ )
h = (h<<8) ^perm[ (h>>24) ^key[i] ];
return h;
}

```

See www.faqs.org/rfcs/rfc3174.html for a much more complicated method used to produce message digests (secure hash).

1.3 Open Addressing

We digress shortly on open addressing where all the items are stored directly in the hash table: If slot $h(x)$ is occupied, try another. More precisely, we use a hash function of the type

$$h : U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}.$$

to produce a probe sequence

$$\langle h(x, 0), h(x, 1), \dots, h(x, m-1) \rangle.$$

Standard methods are *linear probing*

$$h(x, i) = f(x) + i \bmod m.$$

and *quadratic probing*

$$h(x, i) = f(x) + c_1 i + c_2 i^2 \bmod m.$$

where $f : U \rightarrow m$ is a plain hash function. Note that linear probing leads to clustering: collisions produce larger and larger blocks of occupied slots. Furthermore, in both cases only m probe sequences are used, out of potentially $m!$ many.

A significant improvement is *double hashing*:

$$h(x, i) = h_1(x) + h_2(x)i \bmod m.$$

which produces up to m^2 probe sequences. Typical choices for h_i : $h_1(x) = x \bmod m$ and $h_2(x) = 1 + x \bmod m'$ where m' is close to m (not necessarily prime).

Analysis requires some assumption about the likelihood of probe sequences. One often assumes “strong uniform hashing”: all probe sequences are equally likely. Surprisingly, double hashing in practice seems to be close enough to this assumption to provide similar performance. Suppose we have a load factor $\alpha = n/m < 1$.

Lemma 1.1

$$E[\text{probes in unsuccessful search}] \leq 1/(1 - \alpha).$$

Proof. Define

$$\begin{aligned}
p_i &= \Pr[i \text{ probes hit occupied slots}] \\
q_i &= \Pr[\text{at least } i \text{ probes hit occupied slots}]
\end{aligned}$$

Then

$$E[\text{ probes }] = 1 + \sum_{i \geq 0} i p_i = 1 + \sum_{i \geq 1} q_i.$$

Now

$$q_k = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-k+1}{m-k+1} \leq \alpha^k.$$

Hence

$$E[\text{ probes }] \leq 1 + \sum_{i \geq 1} \alpha^i = 1/(1 - \alpha).$$

□

So to get at most 5 expected probes we have to keep the load factor to 80%. For a successful search the situation is a bit more complicated.

1.4 Chaining

The standard solution in chaining is to use linked lists to store all elements with the same hash value. Insert is constant time in this setting, but to perform searches (and deletions) we need to traverse the linked lists. Hence the running time is proportional to length of lists. We note in passing that other containers are plausible as well. E.g., a search tree could reduce the time for search (but would increase the time for insert operations).

To analyze search in the linked list setting we need to make some sanity assumption about the hash function: “uniform simple hashing” which says that for all $x \in U$, $0 \leq a < m$ and independent of other entries in the table we have

$$\Pr[x \text{ hashes into slot } a] = 1/m.$$

Independence is important here, we require in particular that $\Pr[h(x) = h(y)] = 1/m$ for all $x \neq y$.

Theorem 1.1 *Searching and deleting in chaining takes $\Theta(1 + \alpha)$ steps on average.*

Proof. If the search is unsuccessful we have to traverse a whole list, and clearly:

$$E[\text{ length of list number } a] = \alpha$$

Hence, if the search is unsuccessful it takes $1 + \alpha$ steps on average. Similarly, a successful search has length $\alpha/2$ on average. More precisely,

$$E[\text{ length of succ. search }] = 1 + \alpha/2 - 1/2m.$$

□

2 Perfect Hashing

For the sake of simplicity, assume we are dealing with a static set $S \subseteq \mathcal{U}$. The “best possible” hash function S would arguably be one that generates no collisions whatsoever on S . Such a hash function is called *perfect*. Clearly, we need $n \leq m$ for a perfect hash function to exist.

On the other hand, for cardinality reasons this condition is also sufficient in a general sense: there always exists an function $h : \mathcal{U} \rightarrow m$ whose restriction to S is injective. However, that alone is not enough: we want to be able to construct such a function easily, and to evaluate it quickly, without the use of extra tables.

2.1 Universal Hashing

Informally, a good hash function is supposed to behave much like a random function – though, of course, it has to be perfectly deterministic. Nonetheless, randomness can be very helpful in the construction of a good hash function: we could pick a function at random out of a collection of candidates (think of foiling an adversary who is trying to make our hash table misbehave).

To this end, let \mathcal{H} be a collection of hash functions and define the collision set on $x \neq y \in \mathcal{U}$ to be

$$\mathcal{H}_{xy} = \{ h \in \mathcal{H} \mid h(x) = h(y) \}$$

Call \mathcal{H} *universal* iff

$$\forall x \neq y \in U \ (|\mathcal{H}_{xy}| \leq |\mathcal{H}| / m).$$

Equivalently, we have

$$\Pr[\text{collision between } x \text{ and } y] \leq 1/m.$$

Example 2.1 As a toy example, consider $m = 2$, $N = 7$ and $|\mathcal{H}| = 4$. So we need to avoid more than two collisions for any pair $1 \leq x < y \leq 7$.

	1	2	3	4	5	6	7
h_1	0	0	0	0	1	1	1
h_2	0	0	1	1	1	1	0
h_3	0	1	0	1	1	0	1
h_4	1	0	0	1	0	1	1

It might be tempting to lower the bound $1/m$ in the definition of universality, thus further reducing the probability of collisions. Alas, for large N there is not much one can do. More precisely, for any family of functions \mathcal{H} and bound β such that $|\mathcal{H}_{xy}| / |\mathcal{H}| \leq \beta$ for all $x \neq y$ we must have $\beta \geq 1/m - 1/N$.

Here is the key consequence of universality in terms of the expected number of collisions.

Lemma 2.1 *Suppose \mathcal{H} is universal and we hash $n \leq m$ elements where h is chosen at random from \mathcal{H} . Then for any key x*

$$\mathbb{E}[\text{collisions on } x] < 1.$$

Proof. We need to compute

$$e = \sum_h \Pr[h \text{ chosen}] \cdot \# \text{ collisions using } h.$$

Since $\Pr[h \text{ chosen}] = 1/|\mathcal{H}|$ the claim follows from universality: sum over all $y \neq x$ to get $e \leq (n-1)/m < 1$. \square

Example 2.2 All Functions

Note that $\mathcal{H} = \mathcal{U} \rightarrow m$ is a universal family: the number of collisions on $x \neq y$ is m^{N-1} , so $|\mathcal{H}_{xy}|/|\mathcal{H}| = 1/m$. But this collection is useless computationally: it is tedious to generate its members at random, and we do not wish to store large tables (most of these functions can only be represented by a table of size about $N \log m$).

Example 2.3 Prime Table Size

We need to be able to construct a computationally useful universal family. As a warm-up, here is a simple example when the table size $m = p$ is prime. We can think of item x as a number being written in base p notation, using, say, d digits: $x = x_0, \dots, x_{d-1}$. Now consider a vector $\vec{a} = a_0, \dots, a_{d-1}$ of numbers $0 \leq a_i < p$. Each such vector gives rise to a function

$$h_{\vec{a}}(x) = \sum_{i < d} a_i x_i \pmod{p}.$$

Lemma 2.2 $\mathcal{H} = \{h_{\vec{a}} \mid \vec{a} \in p^d\}$ is a universal collection of hash functions.

Proof. Note that $h_{\vec{a}}(x) = h_{\vec{a}}(y)$ for $x \neq y$ implies that

$$a_i = (y_i - x_i)^{-1} \sum_{j \neq i} a_j (x_j - y_j) \pmod{p}$$

for some $i < d$. Hence there are p^{d-1} possible values of \vec{a} which produce a collision. But the total number of choices for \vec{a} is p^d . \square

Example 2.4 Arbitrary Table Size

To remove the prime table size condition, choose a prime $p > m$ larger than the size of the universe. Define

$$\begin{aligned} f_{a,b}(x) &= ax + b \pmod{p} \\ h_{a,b}(x) &= f_{a,b} \pmod{m} \end{aligned}$$

where $0 < a < p$, $0 \leq b < p$. Thus, we have a family \mathcal{H} of size $p(p-1)$.

Note that the functions $f_{a,b}$ are all linear bijections. In fact, for any source point $x_0 \neq y_0 \in \mathbb{Z}_p$ and any target point $x_1 \neq y_1 \in \mathbb{Z}_p$ there is precisely one choice of parameters a and b such that $f_{a,b}(x_0) = x_1$ and $f_{a,b}(y_0) = y_1$ (recall that \mathbb{Z}_p is a field). More precisely,

$$a = \frac{x_1 - y_1}{x_0 - y_0} \quad b = \frac{x_0 y_1 - x_1 y_0}{x_0 - y_0}.$$

Now consider the collision set

$$C = \{(x, y) \in \mathbb{Z}_p^2 \mid x \neq y, x = y \pmod{m}\}.$$

Write $p = q \cdot m + r$ where $0 \leq r < m$.

Claim: $|C| = (q-1)p + (q+1)r$.

To see this, note that the kernel relation of $x \mapsto x \pmod{m}$ on \mathbb{Z}_p has r classes of size $q+1$, and $m-r$ classes of size q . Computing squares and subtracting the diagonal we get the desired result.

It is easy to see that $(q-1)p + (q+1)r \leq p(p-1)/m$, so it follows that \mathcal{H} is universal.

Example 2.5 No Primes

A method that does not require the use of primes is based on matrix-vector multiplications over \mathbb{Z}_2 . Suppose $m = 2^k$ so that the hash values are k -bit strings. The universe consists of all r -bit strings: $\mathcal{U} = \mathbf{2}^r$. For any matrix $M \in \mathbf{2}^{k \times r}$ we can set

$$h_M(x) = M \cdot x$$

where all the arithmetic is in \mathbb{Z}_2 . This produces a family of hash functions of size 2^{kr} .

Note that for $x \neq y \in \mathbf{2}^k$ a collision $h_M(x) = h_M(y)$ implies that at least one column in M is 0. It follows that at most $2^{k(r-1)} = |\mathcal{H}|/m$ matrices can produce a collision on x and y , and \mathcal{H} is dully universal.

2.2 Perfect Hashing

A hash function is *perfect* if it produces no collisions whatsoever on a given set $S \subseteq \mathcal{U}$ – though you have to take this with a grain of salt (we will use a two-level scheme, and there are collisions on the top level). Of course, we must have $m \geq n = |S|$ in this case. The question arises if there is some reasonable way to construct a perfect hash function. We will only deal with the static case where S is fixed.

The first attempt simply uses a lot of space to insure the absence of collisions: $m = n^2$.

Lemma 2.3 *Let $m = n^2$ and pick a hash function randomly from a universal class. Then the probability of having collisions is less than $1/2$.*

Proof. The expected number of collisions is

$$\mathbb{E}[C] = \binom{n}{2} \frac{1}{m} = \frac{n-1}{2n} < 1/2.$$

Done by Markov's inequality. □

Lemma 2.4 *Markov's Inequality*

Let X be a random variable assuming non-negative integer values. Then $\Pr[X \geq k\mathbb{E}[X]] \leq 1/k$.

Of course, a quadratic size table is quite unrealistic. However, we can get away with many smaller tables, each of which is still quadratic in the number of elements stored in that particular table.

To be more precise, let $m = n$ and write n_i , $i = 1, \dots, m$ for the number of elements that hash to slot i under h . Thus, n_i is the size of one equivalence class in the kernel equivalence of h . Now consider the Boolean matrix representation A of that equivalence relation. Clearly the total number of 1's in this matrix is none other than $\sum n_i^2$, the total number of collisions including the phony type $h(x) = h(x)$. But since h is a universal the expected number of these collisions is $n + 2\binom{n}{2}\frac{1}{m} = 2n - 1$. Hence we have shown

Lemma 2.5 $\mathbb{E}[\sum n_i^2] < 2n$.

We can now design a perfect hashing scheme using at most $4n$ space as follows.

Step 1.

- Pick a universal hash function at random for table size $n = m$.

- If $\sum n_i^2 \leq 4n$ go to step 2, otherwise repeat.

Step 2.

Do the following for each of the (non-empty) buckets obtained from Step 1.

- Pick a universal hash function at random for table of size $m = n_i^2$.
- If there are no collisions on the i th bucket, stop; otherwise repeat.

We can find the requisite hash functions in expected linear time.

Note that we could use larger bounds $\sum n_i^2 \leq \alpha n$ and $m = \beta n_i^2$ to decrease the expected number of searches for a good hash function in exchange for using more space.