Hashing

The exposition is based on the following sources, which are all recommended reading:

1. Cormen, Leiserson, Rivest: Introduction to Algorithms, 1991, McGraw Hill (Chapter 12)
2. Sedgewick: Algorithmen in C++, 2002, Pearsons, (Chapter 14)
3. Motwani: Randomized Algorithms, Chapter 8.4

Problem definition

Many applications require the support of operations INSERT, DELETE and SEARCH on a *dynamic set* which can grow and shrink over time.

Each element that can be inserted in the set has a *key* which is drawn from the *universe* U. The subset S that is stored in our set is comparatively very small, that is $|S| << |U|$.

What we would like is a data structure that supports the above operations if possible in time $O(1)$ while using only $O(|S|)$ space.

Assume now that the subset S of U has size $n << |U|$. Obviously it takes to much space to allocate a table of size $|U|$.

Hence we simply allocate a table T of size $m = O(n)$ and *map* each element of S to a position in T. This is done by means of a *hash function* $h : U \mapsto \{0, 1, ..., m-1\}$. A hash function should be computable in time $O(1)$.

The obvious problem that occurs in hashing schemes is that of *collisions*, that is the case that for two keys x and y with $x \neq y$ holds $h(x) = h(y)$.

There are several ways to deal with collisions. One is to avoid them altogether (which is possible using *perfect hashing*). If we allow for collisions there are two common methods to deal with them:

1. Chaining: We keep a linked list of the keys that hash to the same position in the hash table.
2. Open adressing: We store all keys in the table itself, and whenever a collision occurs, we use a secondary methods to locate another, free position in the table.

We will first talk about chaining and open adressing.

Hashing with chaining

The technique is straightforward and leads to the following running times for the basic operations:

1. SEARCH(T, k): This is proportional to the length of the list to which k hashes to. In the worst case $O(n)$.
2. INSERT(T, k): Here we have to append to a list in time $O(1)$.
3. DELETE(T, k): This is proportional to the length of the list to which k hashes to. Again in the worst case $O(n)$.

Of course the worst case scenario is not common. Usually the analysis is conducted under the assumption of *simple uniform hashing*, that means any element is equally likely to hash into any of the hash table slots, independently of the other elements. In that case we conduct the running time analysis in terms of the *load factor* $\alpha = n/m$.

It is not hard to show the following theorem:

Theorem 1. *In a hash table in which collisions are resolved by chaining, a (successful or unsuccessful) search takes time* $\Theta(1+\alpha)$*, on average, under the assumption of simple uniform hashing.*

Proof: To determine the expected number of elements examined during a successful search we take the average over all n elements in the table. More precisely, we take the average over 1 plus the expected length of the list when the i-th element is added. Under the assumption of simple uniform hashing this length is $\frac{i-1}{m}$, hence:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}(1+\frac{i-1}{m}) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) && (2.1)\\
&= 1+\frac{1}{nm}(\frac{(n-1)n}{2}) && (2.2)\\
&= 1+\frac{\alpha}{2}-\frac{1}{m} && (2.3)
\end{aligned}$$

Hence, if $n = O(m)$, searching takes constant time on average.

As a side remark. Even under the assumption of simple uniform hashing, the length of the longest chain is not constant (come back to that during the skip list lecture).

In practice we choose – depending on the hash function – m a prime or a power of two (avoids modulo computations but has other disadvantages).

The problem with the above analysis is of course, that the assumption of simple uniform hashing does not always hold in reality. What can we do in that case? We then have to have a closer look at the hash function used.

Hash functions

A good hash function is of course crucial to the performance of any hashing scheme. It should come close to satisfying the asumption of simple uniform hashing, or more formally, $\sum_{k:h(k)=j} P(k) = 1/m, j = 0,1,\dots,m-1$, where $P(k)$ is the probability that the key k is drawn.

Unfortunately P is generally unknown. We will now introduce some common classes of hash functions and for simplicity assume, that the keys are natural numbers:

In the *division method* we use $h(k) = k \mod m$. Here a few rules of thumb for choosing m.

1. Avoid powers of 2 as a value of m. Otherwise only the lowest order p bits will be used in the hash if $m = 2^p$.
2. Avoid powers of 10 when decimal numbers are used as keys.
3. Good values of p are primes not too close to a power of 2.

In the *multiplication method* we use $h(k) = \lfloor m(kA \mod 1)\rfloor$.

1. The selection of m is not critical for this method. One can pick a power of 2.
2. The method works well with any A, but some values are better than others. According to Knuth a "good" value of A is 0.61803...

Here an example of how the method works. Assume we have $k = 123456$, $m = 10000$ and $A = 0.61803$, then

$$\begin{aligned}
h(k) &= \lfloor 10000\cdot(123456\cdot 0,61803 \mod 1)\rfloor\\
&= \lfloor 10000\cdot(76300.0041151 \mod 1)\rfloor\\
&= \lfloor 41.151\rfloor\\
&= 41
\end{aligned}$$

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 $\{0,1,...,m-1\}$. Then $\mathcal{H}$ is called *universal* if for $x, y \in U, (x \neq y)$

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

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

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 $x = \langle x_0, x_1, ..., x_r\rangle$, where the maximal value of any x_i is less than m.
3. Let $a = \langle a_0, a_1, ..., a_r\rangle$ denote a sequence of $r+1$ elements chosen randomly such that $a_i \in \{0,1,...,m-1\}$. There are m^{r+1} possible such sequences.
4. Define a hash function h_a with $h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$.
5. $\mathcal{H} = \cup_a\{h_a\}$ with m^{r+1} members, one for each possible sequence a.

Theorem 2. *The class $\mathcal{H}$ defined above defines a universal class of hash functions.*

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

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

Hence:

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

Hence:

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

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:

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

and there is a unique $a_0 \bmod m$ 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 $\langle a_1, a_2, ..., a_r\rangle$. Hence, out of m^{r+1} combinations of $a_0, a_1, a_2, ..., a_r$, 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.

What we will need to find a class of universal hash functions is some knowledge about how to solve the equation

$$ax \equiv b(\bmod n), n > 0.$$

Denote $\langle a\rangle$ the subgroup of Z_n generated by a. Then of course the above equation only has a solution if $b \in \langle a\rangle$.

The below theorem tells us that $|\langle a\rangle|$ must be a divisor of n.

Theorem 3 (Langrange's theorem)**.** *if $(S, \oplus)$ is a finite group and $(S', \oplus)$ is a subgroup of $(S, \oplus)$, then $|S'|$ is a divisor of $|S|$.*

The following theorem gives us a more precise characterization of $\langle a \rangle$.

Theorem 4. *For any positive integers a and n, if d = gcd(a, n), then*

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\},$$

and thus

$$|\langle a \rangle| = n/d.$$

Proof: (exercise).

Corollary 5. *The equation $ax \equiv b(\bmod n)$ is solvable for the unknown x if and only if gcd(a, n) divides b.*

Particular useful corollaries are:

Corollary 6. *The equation $ax \equiv b(\bmod n)$ has either d distinct solutions modulo n, where d = gcd(a, n), or it has no solution.*

Corollary 7. *For any $n > 1$, if $gcd(a, n) \equiv 1$, then the equation $ax \equiv b(\bmod n)$ has a unique solution modulo n.*

So what we should keep in mind is that if we choose the n to be prime, we can be sure to have a multiplicative inverse (i.e. $b = 1$) modulo n for each x.

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.

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 simple 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.

Given $h' : U \mapsto \{0,1\ldots,m-1\}$, linear probing uses the hash function: $h(k,i) = (h'(k)+i) \bmod m$ 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.

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 $n/4 = m/8$.

Clusters are likely to arise, since if an empty slot is preceeded 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 preceeding slot was empty.

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

Quadratic hashing uses a hash function of the form $h(k,i) = (h'(k)+c_1 i + c_2 i^2) \bmod m$ for $i = 0,1,\ldots,m-1$, where $h' : U \mapsto \{0,1\ldots,m-1\}$ is an auxilliary 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*.

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 $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$ for $i = 0,1,\ldots,m-1$, where h_1, and h_2 are auxilliary hash functions.

The initial position probed is $T[h_1(k) \bmod m]$, with successive positions offset by the amount $(ih_2(k)) \bmod m$. Now keys with the same initial probe position can have different probe sequences.

Note that $h_2(k)$ must be relatively prime to m for the entire hash table to be accessible for insertion and search. Or 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 then $\Theta(m)$ 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 8. *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.*

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 probe 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

$i = 1, 2, \ldots, n$

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

Hence the following holds:

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

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.

Analysis of open addressing

Theorem 9. *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.*

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

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1} \frac{1}{1-i/m} &= \frac{1}{n}\sum_{i=0}^{n-1} \frac{m}{m-i} && (2.4)\\
&= \frac{m}{n}\sum_{i=m-n+1}^{m} \frac{1}{i} && (2.5)\\
&\leq \frac{1}{\alpha}\int_{m-n}^{m} \frac{1}{x} dx && (2.6)\\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} && (2.7)\\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} && (2.8)\\
& && (2.9)
\end{aligned}$$

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*. A hash funciton is called *perfect* if it causes *no* collisions. (Please note, that the notation in Motwani p. 221ff is different).

In (static) perfect hashing we can achieve a worst case search time of $O(1)$ while using only $O(n)$ space by using a *perfect hash family* of functions. That means for a given set S, a family of hash functions is *perfect*, if it contains at least one perfect function for S.

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 $O(n)$ *such that* the overall number of collisions is linear, and then hashes all elements n_j that are in the same table slot to a secondary hash table of size $O(n_j^2)$. Then the overall space consumption is linear.

By allocating enough space this scheme guarantees, that we can find in an expected 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.

The hash function used in perfect hashing is of the form $h_k(x) = (kx \bmod p) \bmod r$, where p is a prime. It was introduced and analyzed in the paper of Fredman, Komlós, and Szemerédi in 1984.

Definition 10. Consider any $V \subseteq U$ with $|V| = v$, and let $R = \{0, ..., r-1\}$ with $r \geq v$. For $1 \leq k \leq p-1$, define the function $h_k(x) = (kx \bmod p) \bmod r$. Further define for each $i \in R$ the bins corresponding to the keys colliding at i as $B_i(k, r, V) = \{x \in V | h_k(x) = i\}$ and their cardinality as $b_i(k, r, V)$.

r is a parameter which we will choose in our scheme in different ways.

Lemma 11. *For all $V \subseteq U$ of size v, and all $r \geq v$,*

$$\sum_{k=1}^{p-1} \sum_{i=0}^{r-1} \binom{b_i(k,r,V)}{2} < \frac{(p-1)v^2}{r}$$

Proof: The left hand side of the inequality counts exactly the number of collisions for all possible hash functions h_k, or put it differently, the number of tuples $(k, \{x, y\})$ with:

1. $x, y \in V$ with $x \neq y$, and
2. $((kx \bmod p) \bmod r) = ((ky \bmod p) \bmod r)$

Fix now $x \neq y$. The total contribution for this pair is the number of ks for which $k(x-y) \bmod p \equiv 0 \bmod r$. Since p is prime, there is only one solution for each multiple of r. k can assume at most $2(p-1)/r$ values that are multiples of r. There are exactly $\binom{v}{2}$ different pairs $\{x, y\}$ and thus it follows:

$$\sum_{k=1}^{p-1} \sum_{i=0}^{r-1} \binom{b_i(k,r,V)}{2} \leq \binom{v}{2} \frac{2(p-1)}{r} < \frac{(p-1)v^2}{r}.$$

The pidgeonhole principle immediately yields:

Lemma 12. *For all $V \subseteq U$ of size v, and all $r \geq v$ there exists a $k \in \{1, ..., p-1\}$ such that*

$$\sum_{i=0}^{r-1} \binom{b_i(k,r,V)}{2} < \frac{v^2}{r}.$$

Also, for all $V \subseteq U$ of size v, and all $r \geq v$

$$\sum_{i=0}^{r-1} \binom{b_i(k,r,V)}{2} < 2\frac{v^2}{r}$$

for at least one-half of the choices of k.

Applying the above lemma to an input of size v and using a hash table of size $r = v^2$ means, that there is at least one perfect hash function in the family. Allocating space $r = 2v^2$ ensures that at least half of the hash functions are perfect. Also, allocating linear space (v or $2v$) ensures that *the sum* of all collisions is linear.

Lets summarize the perfect hashing scheme

Theorem 13. *For all $S \subseteq U$ of size n, and all $m \geq n$ there exists a hash table representation of S that uses space $O(n)$ and permits the* `Search` *operation in time $O(1)$ without any collisions.*

We choose a primary hash function h_k that maps S into a primary table of size n. This requires $n+1$ memory cells. The keys in $B_i(k, r, V)$ are then hashed into a secondary table of size $b_i(k, r, V)^2$ using a perfect hash function.

Applying the above lemma, there exists a hash function such that the sum of collisions is linear or put it differently:

$$\sum_{i=0}^{n-1} \binom{b_i(k,n,S)}{2} < n.$$

doing some algebra this yields:

$$\sum_{i=0}^{n-1} b_i(k,n,S)^2 < 3n.$$

So for the secondary hash tables we need $3n$ memory cells. In addition we need $2n$ for the k_i and size of the secondary hash tables, yielding all together $6n+1$ memory cells. The processing of a search operation requires the inspection of $5n$ memory cells (the value of k and the primary hash table cell, the value of the corresponding k_i and n_i and the value storing the key).

We give now here an example of the two stage hashing scheme. Assume that $p = 31, n = 6$ and $\mathcal{S} = \{2,4,5,15,18,30\}$. 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:

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  22       7        10   16

 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. $k = T[0] = 2$, $j = (30 \cdot 2 \bmod 31) \bmod 6 = 5$. Hence we have to check the $5+1$th position.
2. $T[6] = 16$, 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.

How can we now achieve a good run time?

We have seen in the above lemma that spending some extra space changes the chances of finding the the correct functions for the two level perfect hashing scheme quite dramatically. By doubling the allowed space we can now find:

1. a first level function with the property that the sum of collisions is linear with probability $p > 0.5$.
2. each perfect second level hash function with probability $p > 0.5$.

Hence we randomly choose a first level hash function and test whether the sum of the collisions is linear. In expectation we have to do this at most twice, hashing the keys into the bins thus needs expected time $O(n)$.

Now we randomly choose a hash function for each bin. To find a perfect function we again need in expectation two trials. Hashing the keys in a bin takes linear time in the number of elements and hence in expectation $O(n)$ time altogether.

Hence it follows, that (static) perfect hashing for a set S of size n takes expected time $O(n)$ using $O(n)$ space.

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 in expected time and space $O(n)$ that supports insertion, deletion and lookup time in *expected, amortized* time $O(1)$.

The idea is to use the static scheme until a collision occurs, then new hash functions are randomly choosen until no collisions occur. In addition the tabel sizes are kept in a linear range (if elements are deleted we might have to make the table smaller, if too many elements are inserted we have to make it bigger). In your exercises you showed, that the table doubling can be done in amortized time $O(1)$.