Hash Functions

A *hash function* is a function that maps data of arbitrary size to an integer of some fixed size.

Example: Java's class `Object` declares function `ob.hashCode()` for `ob` an object. It's a hash function written in OO style, as are the next two examples. Java version 7 says that its value is its address in memory turned into an **int**.

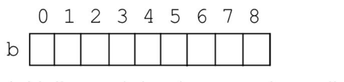
Example: For `in` an object of type `Integer`, `in.hashCode()` yields the **`int`** value that is wrapped in `in`.

Example: Suppose we define a class `Point` with two fields `x` and `y`. For an object `pt` of type `Point`, we could define `pt.hashCode()` to yield the value of `pt.x + pt.y`.

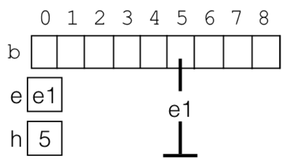
Hash functions are definitive indicators of inequality but only probabilistic indicators of equality —their values typically have smaller sizes than their inputs, so two different inputs may hash to the same number. If two different inputs should be considered "equal" (e.g. two different objects with the same field values), a hash function must respect that. Therefore, in Java, *always* override method `hashCode()` when overriding `equals()` (and vice-versa).

Why do we need hash functions? Well, they are critical in (at least) three areas: (1) hashing, (2) computing *checksums* of files, and (3) areas requiring a high degree of information security, such as saving passwords. Below, we investigate the use of hash functions in these areas and discuss important properties hash functions should have.

Hash functions in hash tables


Initially, each bucket contains null

In the tutorial on hashing using chaining[1], we introduced a hash table `b` to implement a set of some kind. Each element of `b`, called a *bucket*, contains either null or a linked list of values that are in the set. Initially, as shown to the right, each element of `b`, contains null, so the set of values is empty.



Suppose `e1` is to be inserted into the set. A hash function `f` is used to compute an integer. Let's assume that `f(e1) = 14`. `14` is outside the range of array `b`. To get an integer in its range, take `f(e1)` modulo the table size, giving us bucket `h` = `f(e1) % 9` = `5`.[2] Therefore, place `e1` in the linked list in bucket 5, as shown to the right.

The following has been proved: Provided hash function `f` has certain properties and the size of the set is only 1/2 — or even 3/4— of the size of array `b`, the expected time to insert a value into the hash table is in O(1). That's amazing! If the set contains 1,000 elements and `b.length =` 2,000, insertion takes expected constant time!

We state three important points about hash functions used with hash tables.

1. **`f` does not depend on the table size**. Above, `f(e1)` = 14, whether the size of `b` is 9, 10, or 2000. This may be confusing, because one *can* find places in the literature where the hash function includes taking the value mod the table size. The hash functions we use have no knowledge of table `b`, so they cannot depend on its size.
2. **Uniformity**: f should map its possible arguments evenly over its output range, independent of the input distribution. In terms of statistics, the output values should be *uniformly distributed*.
   Function `f(x) = 5x % 20 + 2` for $x \geq 0$ is not a good hash function since it has only 20 possible values.
   Function `f(s) = s.length()` for `s` a `String` is not a good hash function since all strings of the same length map to the same bucket. That's not a uniform distribution.
   Many classic hash functions, like memory addresses and integer values, are poor in this regard because patterns in the input will be reflected in the output.
3. **Speed**: If `f(e)` is not in O(1), then insertion can't be constant time, because computing `f(e)` is part of inserting `e` into a hash table. For example, Java's class `String` implements `s.hashCode()` like this:
   `s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]`. It takes time O(length of `s`). If it is known that the strings are short, e.g. at most 10 chars, then the use of String's `hashcode()` can be considered O(1).

[1] https://www.cs.cornell.edu/courses/JavaAndDS/hashing/01hashing.html

[2] If hash function `f` can deliver negative values, then the formula `Math.abs(f(e1) % b.length` must be used. This is because operator `%` yields a negative value if the first operand is negative. Eg. -14 `%` 9 is -5.

Computing checksums

A *checksum* of a file (or any block of data) is a hash function that yields an integer that is calculated using all parts of the file.

Example. Regard the file as a sequence of 32-bit words (Java **int**s) and add them up.
Example. Regard the file as a sequence of 16-bit words and compute the exclusive or (XOR) of those words.

Here's how checksums are generally used. Suppose you download a large file, for example, the Eclipse app. To ensure the file was not corrupted during the download, a checksum is calculated and sent along with the file. After the file is completely downloaded, the checksum is recalculated on your computer and compared with the downloaded checksum. If they are different, you will get a message saying that the file is corrupted and unusable.

A checksum should yield uniformly distributed values, as suggested above for hash functions used with hash tables. However, a checksum will take time at least linear in the size of the file, rather than constant time, because it references all parts of the file.

The two examples of checksum given above are simple to compute but not effective. For example, the XOR algorithm will detect a single bit being changed, but it won't detect changing the first bit of two different words. A better choice is a Cyclic Redundancy Check (CRC), which is very good at detecting common errors. They are used behind-the-scenes almost everywhere data is stored and moved (Ethernet, Bluetooth, USB, …).

Small checksums can be good at catching accidental corruption but aren't effective at preventing intentional forgeries. For that, we turn to cryptographic hashes with longer outputs. Two checksums in use today are MD5 and the SHA family. MD5 produces a 128–bit hash value. Developed by Ronald Rivest in 1991, it no longer provides much protection against a dedicated forger. The SHA algorithms are national standards, most of which are still considered secure. You can see MD5 and SHA-256 (a 256–bit hash value) in action on this website: http://onlinemd5.com. You can drag a file on your computer onto the webpage and see its checksum. You can also type in text, such as

The quick brown fox jumped over the moon.

and see its checksum, then change "over" to "ower" and see the checksum change. Please do this! Cornell students: you can use this to check the integrity of your submissions to our Course Management System.

Cryptographic hashes and passwords

Passwords are not saved in their original form; instead, they are hashed, and the hashed value is saved. However, this protects against malicious hackers only if the hash function is not reversible: If h = f(x), then it should be extremely difficult to calculate x from h. Also, rarely should two different passwords hash to the same value —otherwise, hacking into one account gives access to the second.

A cryptographic hash function makes it almost impossible to "invert" the hash value and recover the original password, or to find two passwords that hash to the same value. Unfortunately, passwords chosen by people tend to be easy to guess, so an attacker can bypass the need to invert a hash function by instead compiling a dictionary of common passwords and their corresponding codes. If all authentication services used the same hash function, this would provide a kind of master key.

To prevent this, passwords are combined with a random number, called a *salt*, before being hashed, and the salt is stored alongside the hash. This means a different dictionary would be needed for each possible salt value, and that quickly becomes infeasible to store.

Still, once the salt for a particular account is known, it is easy to guess lots of passwords, especially as computers become faster and faster. To mitigate this, specialized procedures for hashing passwords are designed to be *slow* and to require lots of space (unlike most problems in computing, where we want the solution to be fast and small). In cryptography, many recent standards have been chosen through competitions (including the AES block cipher and the SHA-3 hash function), and in 2015 the Password Hashing Competition recommended the Argon2 algorithm for this purpose. Security using cryptography is a continuing, important, field of research.

Other applications

Many more exotic algorithms make use of hash functions as well. Bloom filters are used to determine (approximately) whether a value has been seen before without actually having to store all of the values ever seen. And the MinHash algorithm is an easy way to characterize how similar two large pieces of data (like documents or DNA sequences) are to one another. The ability to represent data by a small, uniformly-distributed number has applications far and wide in computer science.