Hash Functions

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

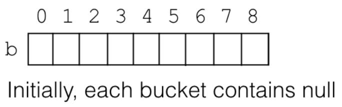
Example: Java's class `Object` declares function `ob.hashCode()` for `ob` an object. It's a hash function written in OO style. Java version 7 says that its value is its address in memory turned into an **int**.

Example: For `in` an object of type `Integer`, `in.hashCode()` yields the **`int`** value that is wrapped in `in`. It's a hash function written in OO style.

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`. It's a hash function written in OO style.

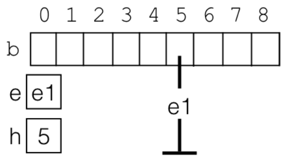
Why, for heaven's sake, do we need hash functions? Well, they are critical in (at least) three areas: (1) hashing, (2) computing *checksums* of files, and (3) areas requiring a high degree of information security, such as saving passwords in some form. Below, we investigate the use of hash functions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

Hash functions in hashing



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 value `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`. The integer `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, it takes expected constant time to insert a new element into the set!

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

1. **`f` does not depend on the table size**. In the above example, `f(e1)` = 14, whether the table size 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. 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.
3. **Speed**: Computing `f(e)` should take constant time. This is because computing `f(e)` is part of inserting `e` into a hash table, so insertion can't be constant time if computing `f(e)` is not. For example, Java's class `String` implements hash function `s.hashCode()` for `t` of type `String`. This function depends on *all* the characters of the string[3]. Therefore, computing the hash function takes time `O`(length of `s`). Therefore, use this hash function for hash tables only if the strings being considered are short, e.g. at most 10 chars.

[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.

[3] `s.hashCode()` is: `s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1].`

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.

Two checksums that are in use today are MD5 and SHA. MD5 produces a 128-bit hash value. It was developed by Ronald Rivest in 1991. The SHA algorithms are cryptographic hash functions (see below). You can see MD5 and SHA-256 (the hash value is 256 bits long) in action on this website: http://onlinemd5.com. Here, 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!

Cryptographic checksums

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 hash value — otherwise, hacking into one account gives access to the second.

A major difficulty is that as computers become faster and faster, it becomes easier and easier to break cryptographic checksums. For example, MD5, which produces a 128-bit hash value, can no longer be used as a cryptographic checksum. It's OK as a vehicle for checking for corruption in a file that has been downloaded but not as a security measure.

The US government's Capstone project, started in 1993, is a long-term project to develop cryptography standards for government and public use. This project developed SHA-1 (Secure Hash Algorithm 1),[4] which produced a 160-bit hash value. But by 2005, it was considered insecure, and its use was deprecated in 2011. A number of successor algorithms have been developed, for example, SHA-256. Further, you can search the web and in particular wikipedia, to find many other cryptographic hash functions as well as discussions of how one can attack and break many of these hash functions. Security using cryptography is a continuing, important, field of research.

[4] See https://en.wikipedia.org/wiki/SHA-1.