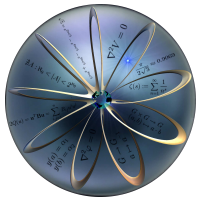

planetmath.org

Math for the people, by the people.

hashing

Canonical name	Hashing
Date of creation	2013-03-22 12:57:34
Last modified on	2013-03-22 12:57:34
Owner	akrowne (2)
Last modified by	akrowne (2)
Numerical id	8
Author	akrowne (2)
Entry type	Topic
Classification	msc 68P05
Classification	msc 68P10
Classification	msc 68P20
Defines	load factor
Defines	hash table
Defines	collision resolution policy
Defines	collision resolution function
Defines	linear probing
Defines	chaining

1 Introduction

Hashing refers to an information storage and retrieval technique which is very widely used in real-world applications. There many more potential places it could profitably be applied as well. In fact, some programming languages these days (such as Perl) are designed with hashing built-in, so the programmer does not even have to know about them (or know much about them) to benefit.

Hashing is inspired by both the classical searching and sorting problems. We know that with comparison-based sorting, the quickest we can put a set of n items in lexicographic order is $\mathcal{O}(n \log n)$. We can then update the sorted structure with new values either by clumsy reallocations of memory and shifting of elements, or by maintaining list structures. Searching for an item in a sorted set of n items is then no faster than $\mathcal{O}(\log n)$.

While fast compared to other common algorithms, these time complexities pose some problems for very large data sets and real-time applications. In addition, there is no hassle-free way to add items to a sorted list of elements; some overhead always must be maintained.

Hashing provides a better way, utilizing a bit of simple mathematics. With hashing, we can typically achieve an average-case time complexity of $\mathcal{O}(1)$ for storage and retrieval, with none of the updating overhead we see for either lists or arrays.

2 The Basics of Hashing

We begin with a set of objects which are referenced by keys. The keys are just handles or labels which uniquely describe the objects. The objects could be any sort of digital information, from a single number to an entire book worth of text[1]. The key could also be anything (textual, or numerical), but what is important for hashing is that it can be efficiently reduced to a numerical representation. The invariance of hashing with respect to the character of the underlying data is one of the main reasons it is such a useful technique.

We "hash" an object by using a function h to place it in a hash table (which is just an array). Thus, h takes the object's key and returns a memory

[1] In fact, modern filesystems use hashing. The keys are the file names, and the objects are the files themselves.

location in a hash table. If our set of keys is contained in a key space K, and T is the set of memory locations in the hash table, then we have

$$h : K \mapsto T$$

Since T is memory locations, we must have $T \subseteq Z$.

Since the size of the object set (n) has nothing to do with h, evaluating h is $\mathcal{O}(1)$ in n. In other words, optimally we can keep throwing stuff into the hash table (until it's full) without expecting any slowdown.

Similarly, we can "look up" an element in $\mathcal{O}(1)$ time, because our query consists of (1) sending our query key through h, and then (2) checking to see if anything is in the hash table location.

Of course, in order to be able to tell if a location in the hash table is occupied, we need a special value called a "tombstone" to delimit an empty spot. A value such as "0" will do, assuming the objects we are storing cannot possibly have the digital representation "0"[2].

3 The Hash Function

The hash function[3] has two main characteristics:

1. it should evaluate in a small amount of time (it should be mathematically simple)

2. it should be relatively unlikely that two different keys are mapped to the same hash table location.

[2]An abstract hash table implementation, however, will have to go to extra lengths to ensure that the tombstone is an "out-of-band" value so that no extra restrictions are put on the values of the objects which the client can store in the hash table.

[3]There is another kind of "hash" commonly referred to that has nothing to do with storage and retrieval, but instead is commonly used for checksums and verification. This kind of hash has to do with the production of a summary key for (usually large) sized objects. This kind of hash function maps the digital object space (infinitely large, but working with some functional range of sizes) into a much smaller but still astronomically large key space (typically something like 128 bits). Like the hash functions discussed above, these hash functions are also very sensitive to change in the input data, so a single bit changed is likely to drastrically change the output of the hash. This is the reason they are useful for verification. As for collisions, they are possible, but rather than being routine, the chances of them are infitesimally small because of the large size of the output key space.

There are two classes of hash functions, those that use integer modulo (*division hashing*) and those which use multiplication by a real number and then a truncation to integer (*multiplication hashing*). In general, division hash functions look like:

$$h(k) = f(k) \pmod{n}$$

Where n is the hash table size and f is a function which expresses the key in numerical form.

This works because the value of x mod n will be a "random" number between

For example, $f(k) = k$ (k integer values) and $n = p$ (a prime) is a very simple and widely used class of hash function. Were k a different type of data than integer (say, strings), we'd need a more complicated f to produce integers.

Multiplication hash functions look like:

$$h(k) = \lfloor n \cdot ((f(k) \cdot r) \pmod{1}) \rfloor$$

Where $0 < r < 1$.

Intuitively, we can expect that multiplying a "random" real number between 0 and 1 with an integer key should give us another "random" real number. Taking the decimal part of this should give us most of the digits of precision (i.e., randomness) of the original, and at the same time act as the analog of the modulo in the division hash to restrict output to a range of values. Multiplying the resulting "random" number between 0 and 1 with the size of the hash table (n) should then give us a "random" index into it.

4 Collision

In general, the hash function is not a one-to-one function. This means two different keys can hash to the same table entry. Because of this, some policy for placing a colliding key is needed. This is called a *collision resolution policy*.

The collision resolution policy must be designed such that if there is a free space in the hash table, it must eventually find it. If not, it must indicate failure.

One collision resolution policy is to use a *collision resolution function* to compute a new location. A simple collision resolution function is to add a constant integer to the hash table location until a free space is found (*linear probing*). In order to guarantee that this will eventually get us to an empty space, hashing using this policy works best with a prime-sized hash table.

Collisions are the reason the hash function needs to spread the objects out in the hash table so they are distant from each other and "evenly" distributed. The more bunched-up they are, the more likely collisions are. If n is the hash table size and t is the number of places in the hash table which are taken, then the value

$$l = \frac{t}{h}$$

is called *load factor* of the hash table, and tells us how "full" it is. As the load factor nears 1, collisions are unavoidable, and the collision resolution policy will be invoked more often.

There are three ways to avoid this quagmire:

- Make the hash table much bigger than the object set.
- Maintain a list at each hash table location instead of individual objects.
- Use a special hash function which is one-to-one.

Option 1 makes it statistically unlikely that we will have to use the collision resolution function. This is the most common solution. If the key space is K, the set of actual keys we are hashing is A ($A \subseteq K$), and the table space is T, then this solution can be phrased as:

$$|A| < c\,|T|$$

with c some fractional constant. In practice, $1/2 < c < 3/4$ gives good results[4].

Option 2 is called *chaining* and eliminates the need for a collision resolution function[5]. Note that to the extent that it is not combined with #1, we

[4]For $|T|$ prime, this condition inspires a search for "good primes" that are approximately one-half greater or double $|A|$. Finding large primes is a non-trivial task; one cannot just make one up on the spot. See good hash table primes.

[5]It also allows over-unity load factors, since we can have more items "in" the hash table than actual "locations" in the hash table.

are replacing a pure hashing solution with a hash-and-list hybrid solution. Whether this is useful depends on the application.

Option 3 is referring to a perfect hash function, and also eliminates the need for a collision resolution function. Depending on how much data we have and how much memory is available to us, it may be possible to select such a hash function. For a trivial example, $h(k) = k$ suffices as a perfect hash function if our keys are integers and our hash table is an array which is larger than the key space. The downside to perfect hash functions is that you can never use them in the generic case; you always have to know something about the keyspace and the data set size to guarantee perfectness.

5 Variations

In addition to perfect hash functions, there is also a class of hash functions called minimal perfect hash functions. These functions are one-to-one and onto. Thus, they map a key space of size n to a hash table space of size n, all of our objects will map perfectly to hash table locations, and there will be no leftover spots wasted and no collisions. Once again an array indexed by integers is a simple example, but obviously there is more utility to be had from keys which are not identical to array indices. Minimal perfect hash functions may seem too good to be true– but they are only applicable in special situations.

There is also a class of hash functions called order-preserving hash functions. This just means that the lexicographic order of the hashed object locations is the same as the lexicographic order of the keys:

$$k_1 < k_2 \implies h(k_1) < h(k_2)$$

for keys k_1 and k_2.

The benefit of an order-preserving hash is that it performs a sort with no "extra" work[6]; we can print out the hashed objects in sorted order simply by scanning the table linearly.

In principle, an order-preserving hash is always possible when the used key space is smaller than the hash table size. However, it may be tricky to design the right hash function, and it is sure to be very specific to the key

[6]In other words, we escape the $\mathcal{O}(n \log n)$ bounds on sorting because we aren't doing a comparison-based sort at all.

space. In addition, some complexity is likely to be introduced in the hash function which will slow down its evaluation.

6 References

Coming soon!