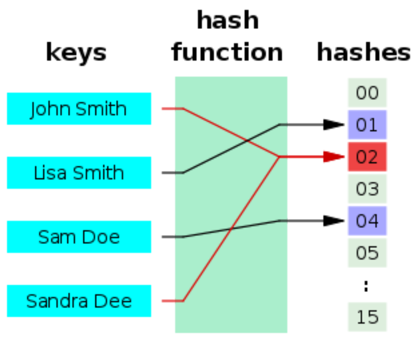
WIKIPEDIA
The Free Encyclopedia

Hash function

A **hash function** is any function that can be used to map data of arbitrary size to fixed-size values, though there are some hash functions that support variable length output.[1] The values returned by a hash function are called *hash values*, *hash codes*, *digests*, or simply *hashes*. The values are usually used to index a fixed-size table called a *hash table*. Use of a hash function to index a hash table is called *hashing* or *scatter storage addressing*.



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

Hash functions and their associated hash tables are used in data storage and retrieval applications to access data in a small and nearly constant time per retrieval. They require an amount of storage space only fractionally greater than the total space required for the data or records themselves. Hashing is a computationally and storage space-efficient form of data access that avoids the non-constant access time of ordered and unordered lists and structured trees, and the often exponential storage requirements of direct access of state spaces of large or variable-length keys.

Use of hash functions relies on statistical properties of key and function interaction: worst-case behaviour is intolerably bad with a vanishingly small probability, and average-case behaviour can be nearly optimal (minimal collision).[2]:527

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, lossy compression, randomization functions, error-correcting codes, and ciphers. Although the concepts overlap to some extent, each one has its own uses and requirements and is designed and optimized differently. The hash function differs from these concepts mainly in terms of data integrity.

Overview

A hash function takes a key as an input, which is associated with a datum or record and used to identify it to the data storage and retrieval application. The keys may be fixed length, like an integer, or variable length, like a name. In some cases, the key is the datum itself. The output is a hash code used to index a hash table holding the data or records, or pointers to them.

A hash function may be considered to perform three functions:

- Convert variable-length keys into fixed length (usually machine word length or less) values, by folding them by words or other units using a parity-preserving operator like ADD or XOR.
- Scramble the bits of the key so that the resulting values are uniformly distributed over the keyspace.
- Map the key values into ones less than or equal to the size of the table

A good hash function satisfies two basic properties: 1) it should be very fast to compute; 2) it should minimize duplication of output values (collisions). Hash functions rely on generating favourable probability distributions for their effectiveness, reducing access time to nearly constant. High table loading factors, pathological key sets and poorly designed hash functions can result in access times approaching linear in the number of items in the table. Hash functions can be designed to give the best worst-case performance,[Notes 1] good performance under high table loading factors, and in special cases, perfect (collisionless) mapping of keys into hash codes. Implementation is based on parity-preserving bit operations (XOR and ADD), multiply, or divide. A necessary adjunct to the hash function is a collision-resolution method that employs an auxiliary data structure like linked lists, or systematic probing of the table to find an empty slot.

Hash tables

Hash functions are used in conjunction with hash tables to store and retrieve data items or data records. The hash function translates the key associated with each datum or record into a hash code, which is used to index the hash table. When an item is to be added to the table, the hash code may index an empty slot (also called a bucket), in which case the item is added to the table there. If the hash code indexes a full slot, some kind of collision resolution is required: the new item may be omitted (not added to the table), or replace the old item, or it can be added to the table in some other location by a specified procedure. That procedure depends on the structure of the hash table: In *chained hashing*, each slot is the head of a linked list or chain, and items that collide at the slot are added to the chain. Chains may be kept in random order and searched linearly, or in serial order, or as a self-ordering list by frequency to speed up access. In *open address hashing*, the table is probed starting from the occupied slot in a specified manner, usually by linear probing, quadratic probing, or double hashing until an open slot is located or the entire table is probed (overflow). Searching for the item follows the same procedure until the item is located, an open slot is found or the entire table has been searched (item not in table).

Specialized uses

Hash functions are also used to build caches for large data sets stored in slow media. A cache is generally simpler than a hashed search table since any collision can be resolved by discarding or writing back the older of the two colliding items.[3]

Hash functions are an essential ingredient of the Bloom filter, a space-efficient probabilistic data structure that is used to test whether an element is a member of a set.

A special case of hashing is known as geometric hashing or *the grid method*. In these applications, the set of all inputs is some sort of metric space, and the hashing function can be interpreted as a partition of that space into a grid of *cells*. The table is often an array with two or more indices (called a *grid file*, *grid index*, *bucket grid*, and similar names), and the hash function returns an index tuple. This principle is widely used in computer graphics, computational geometry and many other disciplines, to solve many proximity problems in the plane or in three-dimensional space, such as finding closest pairs in a set of points, similar shapes in a list of shapes, similar images in an image database, and so on.

Hash tables are also used to implement associative arrays and dynamic sets.[4]

Properties

Uniformity

A good hash function should map the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same probability. The reason for this last requirement is that the cost of hashing-based methods goes up sharply as the number of *collisions*—pairs of inputs that are mapped to the same hash value—increases. If some hash values are more likely to occur than others, a larger fraction of the lookup operations will have to search through a larger set of colliding table entries.

Note that this criterion only requires the value to be *uniformly distributed*, not *random* in any sense. A good randomizing function is (barring computational efficiency concerns) generally a good choice as a hash function, but the converse need not be true.

Hash tables often contain only a small subset of the valid inputs. For instance, a club membership list may contain only a hundred or so member names, out of the very large set of all possible names. In these cases, the uniformity criterion should hold for almost all typical subsets of entries that may be found in the table, not just for the global set of all possible entries.

In other words, if a typical set of m records is hashed to n table slots, the probability of a bucket receiving many more than m/n records should be vanishingly small. In particular, if m is less than n, very few buckets should have more than one or two records. A small number of collisions is virtually inevitable, even if n is much larger than m – see the birthday problem.

In special cases when the keys are known in advance and the key set is static, a hash function can be found that achieves absolute (or collisionless) uniformity. Such a hash function is said to be *perfect*. There is no algorithmic way of constructing such a function - searching for one is a factorial function of the number of keys to be mapped versus the number of table slots they're tapped into. Finding a perfect hash function over more than a very small set of keys is usually computationally infeasible; the resulting function is likely to be more computationally complex than a standard hash function and provides only a marginal advantage over a function with good statistical properties that yields a minimum number of collisions. See **universal hash function**.

Testing and measurement

When testing a hash function, the uniformity of the distribution of hash values can be evaluated by the chi-squared test. This test is a goodness-of-fit measure: it's the actual distribution of items in buckets versus the expected (or uniform) distribution of items. The formula is:

$$\frac{\sum_{j=0}^{m-1}(b_j)(b_j+1)/2}{(n/2m)(n+2m-1)}$$

where: n is the number of keys, m is the number of buckets, b_j is the number of items in bucket j

A ratio within one confidence interval (0.95 - 1.05) is indicative that the hash function evaluated has an expected uniform distribution.

Hash functions can have some technical properties that make it more likely that they'll have a uniform distribution when applied. One is the strict avalanche criterion: whenever a single input bit is complemented, each of the output bits changes with a 50% probability. The reason for this property is that selected subsets of the keyspace may have low variability. For the output to be uniformly distributed, a low amount of variability, even one bit, should translate into a high amount of variability (i.e. distribution over the tablespace) in the output. Each bit should change with a probability of 50% because if some bits are reluctant to change, the keys become clustered around those values. If the bits want to change too readily, the mapping is approaching a fixed XOR function of a single bit. Standard tests for this property have been described in the literature.[5] The relevance of the criterion to a multiplicative hash function is assessed here.[6]

Efficiency

In data storage and retrieval applications, the use of a hash function is a trade-off between search time and data storage space. If search time were unbounded, a very compact unordered linear list would be the best medium; if storage space were unbounded, a randomly accessible structure indexable by the key-value would be very large, very sparse, but very fast. A hash function takes a finite amount of time to map a potentially large keyspace to a feasible amount of storage space searchable in a bounded amount of time regardless of the number of keys. In most applications, the hash function should be computable with minimum latency and secondarily in a minimum number of instructions.

Computational complexity varies with the number of instructions required and latency of individual instructions, with the simplest being the bitwise methods (folding), followed by the multiplicative methods, and the most complex (slowest) are the division-based methods.

Because collisions should be infrequent, and cause a marginal delay but are otherwise harmless, it's usually preferable to choose a faster hash function over one that needs more computation but saves a few collisions.

Division-based implementations can be of particular concern because the division is microprogrammed on nearly all chip architectures. Divide (modulo) by a constant can be inverted to become a multiply by the word-size multiplicative-inverse of the constant. This can be done by the programmer, or by the compiler. Divide can also be reduced directly into a series of shift-subtracts and shift-adds, though minimizing the number of such operations required is a daunting problem; the number of assembly instructions resulting may be more than a dozen, and swamp the pipeline. If the architecture has hardware multiply functional units, the multiply-by-inverse is likely a better approach.

We can allow the table size n to not be a power of 2 and still not have to perform any remainder or division operation, as these computations are sometimes costly. For example, let n be significantly less than 2^b. Consider a pseudorandom number generator function $P(\text{key})$ that is uniform on the interval $[0, 2^b - 1]$. A hash function uniform on the interval $[0, n\text{-}1]$ is $n\ P(\text{key})/2^b$. We can replace the division by a (possibly faster) right bit shift: $nP(\text{key}) >> b$.

If keys are being hashed repeatedly, and the hash function is costly, computing time can be saved by precomputing the hash codes and storing them with the keys. Matching hash codes almost certainly means the keys are identical. This technique is used for the transposition table in game-playing programs, which stores a 64-bit hashed representation of the board position.

Universality

A *universal hashing* scheme is a randomized algorithm that selects a hashing function h among a family of such functions, in such a way that the probability of a collision of any two distinct keys is $1/m$, where m is the number of distinct hash values desired—independently of the two keys. Universal hashing ensures (in a probabilistic sense) that the hash function application will behave as well as if it were using a random function, for any distribution of the input data. It will, however, have more collisions than perfect hashing and may require more operations than a special-purpose hash function.

Applicability

A hash function is applicable in a variety of situations. A hash function that allows only certain table sizes, strings only up to a certain length, or can't accept a seed (i.e. allow double hashing) isn't as useful as one that does.

Deterministic

A hash procedure must be deterministic—meaning that for a given input value it must always generate the same hash value. In other words, it must be a function of the data to be hashed, in the mathematical sense of the term. This requirement excludes hash functions that depend on external variable parameters, such as pseudo-random number generators or the time of day. It also excludes functions that depend on the memory address of the object being hashed in cases that the address may change during execution (as may happen on systems that use certain methods of garbage collection), although sometimes rehashing of the item is possible.

The determinism is in the context of the reuse of the function. For example, Python adds the feature that hash functions make use of a randomized seed that is generated once when the Python process starts in addition to the input to be hashed.[7] The Python hash (SipHash) is still a valid hash function when used within a single run. But if the values are persisted (for example, written to disk) they can no longer be treated as valid hash values, since in the next run the random value might differ.

Defined range

It is often desirable that the output of a hash function have fixed size (but see below). If, for example, the output is constrained to 32-bit integer values, the hash values can be used to index into an array. Such hashing is commonly used to accelerate data searches.[8] Producing fixed-length output from variable length input can be accomplished by breaking the input data into chunks of specific size. Hash functions used for data searches use some arithmetic expression that iteratively processes chunks of the input (such as the characters in a string) to produce the hash value.[8]

Variable range

In many applications, the range of hash values may be different for each run of the program or may change along the same run (for instance, when a hash table needs to be expanded). In those situations, one needs a hash function which takes two parameters—the input data z, and the number n of allowed hash values.

A common solution is to compute a fixed hash function with a very large range (say, 0 to $2^{32} - 1$), divide the result by n, and use the division's remainder. If n is itself a power of 2, this can be done by bit masking and bit shifting. When this approach is used, the hash function must be chosen so that the result has fairly uniform distribution between 0 and $n - 1$, for any value of n that may occur in the application. Depending on the function, the remainder may be uniform only for certain values of n, e.g. odd or prime numbers.

Variable range with minimal movement (dynamic hash function)

When the hash function is used to store values in a hash table that outlives the run of the program, and the hash table needs to be expanded or shrunk, the hash table is referred to as a dynamic hash table.

A hash function that will relocate the minimum number of records when the table is resized is desirable. What is needed is a hash function $H(z,n)$ – where z is the key being hashed and n is the number of allowed hash values – such that $H(z,n + 1) = H(z,n)$ with probability close to $n/(n + 1)$.

Linear hashing and spiral storage are examples of dynamic hash functions that execute in constant time but relax the property of uniformity to achieve the minimal movement property. Extendible hashing uses a dynamic hash function that requires space proportional to n to compute the hash function, and it becomes a function of the previous keys that have been inserted. Several algorithms that preserve the uniformity property but require time proportional to n to compute the value of $H(z,n)$ have been invented.

A hash function with minimal movement is especially useful in distributed hash tables.

Data normalization

In some applications, the input data may contain features that are irrelevant for comparison purposes. For example, when looking up a personal name, it may be desirable to ignore the distinction between upper and lower case letters. For such data, one must use a hash function that is compatible with the data equivalence criterion being used: that is, any two inputs that are considered equivalent must yield the same hash value. This can be accomplished by normalizing the input before hashing it, as by upper-casing all letters.

Hashing integer data types

There are several common algorithms for hashing integers. The method giving the best distribution is data-dependent. One of the simplest and most common methods in practice is the modulo division method.

Identity hash function

If the data to be hashed is small enough, one can use the data itself (reinterpreted as an integer) as the hashed value. The cost of computing this *identity* hash function is effectively zero. This hash function is perfect, as it maps each input to a distinct hash value.

The meaning of "small enough" depends on the size of the type that is used as the hashed value. For example, in Java, the hash code is a 32-bit integer. Thus the 32-bit integer `Integer` and 32-bit floating-point `Float` objects can simply use the value directly; whereas the 64-bit integer `Long` and 64-bit floating-point `Double` cannot use this method.

Other types of data can also use this hashing scheme. For example, when mapping character strings between upper and lower case, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character ("A" for "a", "8" for "8", etc.). If each character is stored in 8 bits (as in extended ASCII[Notes 2] or ISO Latin 1), the table has only $2^8 = 256$ entries; in the case of Unicode characters, the table would have $17 \times 2^{16} = 1\,114\,112$ entries.

The same technique can be used to map two-letter country codes like "us" or "za" to country names ($26^2 = 676$ table entries), 5-digit zip codes like 13083 to city names (100 000 entries), etc. Invalid data values (such as the country code "xx" or the zip code 00000) may be left undefined in the table or mapped to some appropriate "null" value.

Trivial hash function

If the keys are uniformly or sufficiently uniformly distributed over the key space, so that the key values are essentially random, they may be considered to be already 'hashed'. In this case, any number of any bits in the key may be extracted and collated as an index into the hash table. For example, a simple hash function might mask off the least significant m bits and use the result as an index into a hash table of size 2^{m}.

Folding

A folding hash code is produced by dividing the input into n sections of m bits, where 2^{m} is the table size, and using a parity-preserving bitwise operation such as ADD or XOR to combine the sections, followed by a mask or shifts to trim off any excess bits at the high or low end. For example, for a table size of 15 bits and key value of 0x0123456789ABCDEF, there are five sections consisting of 0x4DEF, 0x1357, 0x159E, 0x091A and 0x8. Adding, we obtain 0x7AA4, a 15-bit value.

Mid-squares

A mid-squares hash code is produced by squaring the input and extracting an appropriate number of middle digits or bits. For example, if the input is 123,456,789 and the hash table size 10,000, squaring the key produces 15,241,578,750,190,521, so the hash code is taken as the middle 4 digits of the 17-

digit number (ignoring the high digit) 8750. The mid-squares method produces a reasonable hash code if there is not a lot of leading or trailing zeros in the key. This is a variant of multiplicative hashing, but not as good because an arbitrary key is not a good multiplier.

Division hashing

A standard technique is to use a modulo function on the key, by selecting a divisor M which is a prime number close to the table size, so $h(K) = K \bmod M$. The table size is usually a power of 2. This gives a distribution from $\{0, M-1\}$. This gives good results over a large number of key sets. A significant drawback of division hashing is that division is microprogrammed on most modern architectures including x86 and can be 10 times slower than multiply. A second drawback is that it won't break up clustered keys. For example, the keys 123000, 456000, 789000, etc. modulo 1000 all map to the same address. This technique works well in practice because many key sets are sufficiently random already, and the probability that a key set will be cyclical by a large prime number is small.

Algebraic coding

Algebraic coding is a variant of the division method of hashing which uses division by a polynomial modulo 2 instead of an integer to map n bits to m bits.[2]:512–513 In this approach, $M = 2^m$ and we postulate an mth degree polynomial $\mathrm{Z}(x) = x^m + \zeta_{m-1}x^{m-1} + \ldots + \zeta_0$. A key $K = (k_{n-1} \ldots k_1 k_0)_2$ can be regarded as the polynomial $K(x) = k_{n-1}x^{n-1} + \ldots + k_1 x + k_0$. The remainder using polynomial arithmetic modulo 2 is $K(x) \bmod Z(x) = h_{m-1}x^{m-1} + \ldots + h_1 x + h_0$. Then $h(K) = (h_{m-1} \ldots h_1 h_0)_2$. If $Z(x)$ is constructed to have t or fewer non-zero coefficients, then keys which share less than t bits are guaranteed to not collide.

Z a function of k, t and n, a divisor of 2^k-1, is constructed from the GF(2^k) field. Knuth gives an example: for n=15, m=10 and t=7, $Z(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$. The derivation is as follows:

Let S be the smallest set of integers such that $\{1, 2, \ldots, t\} \subseteq S$ and $(2j \bmod n) \in S \quad \forall j \in S$.[Notes 3]

Define $P(x) = \prod_{j \in S}(x - \alpha^j)$ where $\alpha \in^n GF(2^k)$ and where the coefficients of $P(x)$ are computed in this field. Then the degree of $P(x) = |S|$. Since α^{2j} is a root of $P(x)$ whenever α^j is a root, it follows that the coefficients p^i of $P(x)$ satisfy $p_i^2 = p_i$ so they are all 0 or 1. If $R(x) = r_{(n-1)}x^{n-1} + \ldots + r_1 x + r_0$ is any nonzero polynomial modulo 2 with at most t nonzero coefficients, then $R(x)$ is not a multiple of $P(x)$ modulo 2.[Notes 4] If follows that the corresponding hash function will map keys with fewer than t bits in common to unique indices.[2]:542–543

The usual outcome is that either n will get large, or twill gets large, or both, for the scheme to be computationally feasible. Therefore, it's more suited to hardware or microcode implementation.[2]:542–543

Unique permutation hashing

See also unique permutation hashing, which has a guaranteed best worst-case insertion time.[9]

Multiplicative hashing

Standard multiplicative hashing uses the formula $h_a(K) = \lfloor (aK \bmod W)/(W/M) \rfloor$ which produces a hash value in $\{0, \ldots, M-1\}$. The value a is an appropriately chosen value that should be relatively prime to W; it should be large and its binary representation a random mix of 1's and 0's. An important practical special case occurs when $W = 2^w$ and $M = 2^m$ are powers of 2 and w is the machine word size. In this case this formula becomes $h_a(K) = \lfloor (aK \bmod 2^w)/2^{w-m} \rfloor$. This is special because arithmetic modulo 2^w is done by default in low-level programming languages and integer division by a power of 2 is simply a right-shift, so, in C, for example, this function becomes

```
unsigned hash(unsigned K)
{
   return (a*K) >> (w-m);
}
```

and for fixed m and w this translates into a single integer multiplication and right-shift making it one of the fastest hash functions to compute.

Multiplicative hashing is susceptible to a "common mistake" that leads to poor diffusion—higher-value input bits do not affect lower-value output bits.[10] A transmutation on the input which shifts the span of retained top bits down and XORs or ADDs them to the key before the multiplication step corrects for this. So the resulting function looks like:[6]

```
unsigned hash(unsigned K)
{
   K ^= K >> (w-m);
   return (a*K) >> (w-m);
}
```

Fibonacci hashing

Fibonacci hashing is a form of multiplicative hashing in which the multiplier is $2^w/\phi$, where w is the machine word length and ϕ (phi) is the golden ratio (approximately 5/3). A property of this multiplier is that it uniformly distributes over the table space, blocks of consecutive keys with respect to any block of bits in the key. Consecutive keys within the high bits or low bits of the key (or some other field) are relatively common. The multipliers for various word lengths ^{w} are:

- 16: $a=40503_{10}$
- 32: $a=2654435769_{10}$

- 48: $a=173961102589771_{10}$[Notes 5]
- 64: $a=11400714819323198485_{10}$[Notes 6]

Zobrist hashing

Tabulation hashing, more generally known as *Zobrist hashing* after Albert Zobrist, an American computer scientist, is a method for constructing universal families of hash functions by combining table lookup with XOR operations. This algorithm has proven to be very fast and of high quality for hashing purposes (especially hashing of integer-number keys).[11]

Zobrist hashing was originally introduced as a means of compactly representing chess positions in computer game-playing programs. A unique random number was assigned to represent each type of piece (six each for black and white) on each space of the board. Thus a table of 64×12 such numbers is initialized at the start of the program. The random numbers could be any length, but 64 bits was natural due to the 64 squares on the board. A position was transcribed by cycling through the pieces in a position, indexing the corresponding random numbers (vacant spaces were not included in the calculation), and XORing them together (the starting value could be 0, the identity value for XOR, or a random seed). The resulting value was reduced by modulo, folding or some other operation to produce a hash table index. The original Zobrist hash was stored in the table as the representation of the position.

Later, the method was extended to hashing integers by representing each byte in each of 4 possible positions in the word by a unique 32-bit random number. Thus, a table of 2^8×4 of such random numbers is constructed. A 32-bit hashed integer is transcribed by successively indexing the table with the value of each byte of the plain text integer and XORing the loaded values together (again, the starting value can be the identity value or a random seed). The natural extension to 64-bit integers is by use of a table of 2^8×8 64-bit random numbers.

This kind of function has some nice theoretical properties, one of which is called *3-tuple independence* meaning every 3-tuple of keys is equally likely to be mapped to any 3-tuple of hash values.

Customised hash function

A hash function can be designed to exploit existing entropy in the keys. If the keys have leading or trailing zeros, or particular fields that are unused, always zero or some other constant, or generally vary little, then masking out only the volatile bits and hashing on those will provide a better and possibly faster hash function. Selected divisors or multipliers in the division and multiplicative schemes may make more uniform hash functions if the keys are cyclic or have other redundancies.

Hashing variable-length data

When the data values are long (or variable-length) character strings—such as personal names, web page addresses, or mail messages—their distribution is usually very uneven, with complicated dependencies. For example, text in any natural language has highly non-uniform distributions of characters, and character pairs, characteristic of the language. For such data, it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way.

Middle and ends

Simplistic hash functions may add the first and last *n* characters of a string along with the length, or form a word-size hash from the middle 4 characters of a string. This saves iterating over the (potentially long) string, but hash functions that do not hash on all characters of a string can readily become linear due to redundancies, clustering or other pathologies in the key set. Such strategies may be effective as a custom hash function if the structure of the keys is such that either the middle, ends or other fields are zero or some other invariant constant that doesn't differentiate the keys; then the invariant parts of the keys can be ignored.

Character folding

The paradigmatic example of folding by characters is to add up the integer values of all the characters in the string. A better idea is to multiply the hash total by a constant, typically a sizable prime number, before adding in the next character, ignoring overflow. Using exclusive 'or' instead of add is also a plausible alternative. The final operation would be a modulo, mask, or other function to reduce the word value to an index the size of the table. The weakness of this procedure is that information may cluster in the upper or lower bits of the bytes, which clustering will remain in the hashed result and cause more collisions than a proper randomizing hash. ASCII byte codes, for example, have an upper bit of 0 and printable strings don't use the first 32 byte codes, so the information (95-byte codes) is clustered in the remaining bits in an unobvious manner.

The classic approach dubbed the PJW hash based on the work of Peter. J. Weinberger at ATT Bell Labs in the 1970s, was originally designed for hashing identifiers into compiler symbol tables as given in the "Dragon Book".[12] This hash function offsets the bytes 4 bits before ADDing them together. When the quantity wraps, the high 4 bits are shifted out and if non-zero, XORed back into the low byte of the cumulative quantity. The result is a word size hash code to which a modulo or other reducing operation can be applied to produce the final hash index.

Today, especially with the advent of 64-bit word sizes, much more efficient variable-length string hashing by word chunks is available.

Word length folding

Modern microprocessors will allow for much faster processing if 8-bit character strings are not hashed by processing one character at a time, but by interpreting the string as an array of 32 bit or 64-bit integers and hashing/accumulating these "wide word" integer values by means of arithmetic operations (e.g. multiplication by constant and bit-shifting). The final word, which may have unoccupied byte positions, is filled with zeros or a specified "randomizing" value before being folded into the hash. The accumulated hash code is reduced by a final modulo or other operation to yield an index into the table.

Radix conversion hashing

Analogous to the way an ASCII or EBCDIC character string representing a decimal number is converted to a numeric quantity for computing, a variable length string can be converted as $(x_0 a^{k-1} + x_1 a^{k-2} + \ldots + x_{k-2} a + x_{k-1})$. This is simply a polynomial in a radix $a > 1$ that takes the components $(x_0, x_1, \ldots, x_{k-1})$ as the characters of the input string of length k. It can be used directly as the hash code, or a hash function applied to it to map the potentially large value to the hash table size. The value of a is usually a prime number at least large enough to hold the number of different characters in the character set of potential keys. Radix conversion hashing of strings minimizes the number of collisions.[13] Available data sizes may restrict the maximum length of string that can be hashed with this method. For example, a 128-bit double long word will hash only a 26 character alphabetic string (ignoring case) with a radix of 29; a printable ASCII string is limited to 9 characters using radix 97 and a 64-bit long word. However, alphabetic keys are usually of modest length, because keys must be stored in the hash table. Numeric character strings are usually not a problem; 64 bits can count up to 10^{19}, or 19 decimal digits with radix 10.

Rolling hash

In some applications, such as substring search, one can compute a hash function h for every k-character substring of a given n-character string by advancing a window of width k characters along the string; where k is a fixed integer, and n is greater than k. The straightforward solution, which is to extract such a substring at every character position in the text and compute h separately, requires a number of operations proportional to $k \cdot n$. However, with the proper choice of h, one can use the technique of rolling hash to compute all those hashes with an effort proportional to $mk + n$ where m is the number of occurrences of the substring.

The most familiar algorithm of this type is Rabin-Karp with best and average case performance $O(n+mk)$ and worst case $O(n \cdot k)$ (in all fairness, the worst case here is gravely pathological: both the text string and substring are composed of a repeated single character, such as t="AAAAAAAAAAA", and s="AAA"). The hash function used for the algorithm is usually the Rabin fingerprint, designed to avoid collisions in 8-bit character strings, but other suitable hash functions are also used.

Analysis

Worst case result for a hash function can be assessed two ways: theoretical and practical. Theoretical worst case is the probability that all keys map to a single slot. Practical worst case is expected longest probe sequence (hash function + collision resolution method). This analysis considers uniform hashing, that is, any key will map to any particular slot with probability $1/m$, characteristic of universal hash functions.

While Knuth worries about adversarial attack on real time systems,[14] Gonnet has shown that the probability of such a case is "ridiculously small". His representation was that the probability of k of n keys mapping to a single slot is $\frac{e^{-\alpha}\alpha^k}{k!}$ where α is the load factor, n/m.[15]

History

The term *hash* offers a natural analogy with its non-technical meaning (to chop up or make a mess out of something), given how hash functions scramble their input data to derive their output.[16]:514 In his research for the precise origin of the term, Donald Knuth notes that, while Hans Peter Luhn of IBM appears to have been the first to use the concept of a hash function in a memo dated January 1953, the term itself would only appear in published literature in the late 1960s, in Herbert Hellerman's *Digital Computer System Principles*, even though it was already widespread jargon by then.[16]:547–548

See also

- List of hash functions
- Nearest neighbor search
- Cryptographic hash function
- Distributed hash table
- Identicon
- Low-discrepancy sequence
- Transposition table

Notes

1. This is useful in cases where keys are devised by a malicious agent, for example in pursuit of a DOS attack.
2. Plain ASCII is a 7-bit character encoding, although it is often stored in 8-bit bytes with the highest-order bit always clear (zero). Therefore, for plain ASCII, the bytes have only 2^7 = 128 valid values, and the character translation table has only this many entries.
3. For example, for n=15, k=4, t=6, $S = \{1, 2, 3, 4, 5, 6, 8, 10, 12, 9\}$ [Knuth]
4. Knuth conveniently leaves the proof of this to the reader.
5. Unisys large systems
6. 11400714819323198486 is closer, but the bottom bit is zero, essentially throwing away a bit. The next closest odd number is that given.

References

1. Aggarwal, Kirti; Verma, Harsh K. (March 19, 2015). *Hash_RC6 — Variable length Hash algorithm using RC6* (https://ieeexplore.ieee.org/document/7164747). 2015 International Conference on Advances in Computer Engineering and Applications (ICACEA). doi:10.1109/ICACEA.2015.7164747 (https://doi.org/10.1109%2FICACEA.2015.7164747). Retrieved January 24, 2023.
2. Knuth, Donald E. (1973). *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Reading, MA., United States: Addison-Wesley. Bibcode:1973acp..book.....K (https://ui.adsabs.harvard.edu/abs/1973acp..book.....K). ISBN 978-0-201-03803-3.
3. Stokes, Jon (2002-07-08). "Understanding CPU caching and performance" (https://arstechnica.com/gadgets/reviews/2002/07/caching.ars). *Ars Technica*. Retrieved 2022-02-06.
4. Menezes, Alfred J.; van Oorschot, Paul C.; Vanstone, Scott A (1996). *Handbook of Applied Cryptography* (https://archive.org/details/handbookofapplie0000mene). CRC Press. ISBN 978-0849385230.

5. Castro, Julio Cesar Hernandez; et al. (3 February 2005). "The strict avalanche criterion randomness test". *Mathematics and Computers in Simulation*. Elsevier. **68** (1): 1–7. doi:10.1016/j.matcom.2004.09.001 (https://doi.org/10.1016%2Fj.matcom.2004.09.001). S2CID 18086276 (https://api.semanticscholar.org/CorpusID:18086276).
6. Sharupke, Malte (16 June 2018). "Fibonacci Hashing: The Optimization that the World Forgot" (https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-integer-modulo/). *Probably Dance*.
7. "3. Data model — Python 3.6.1 documentation" (https://docs.python.org/3/reference/datamodel.html#object.__hash__). *docs.python.org*. Retrieved 2017-03-24.
8. Sedgewick, Robert (2002). "14. Hashing". *Algorithms in Java* (3 ed.). Addison Wesley. ISBN 978-0201361209.
9. Dolev, Shlomi; Lahiani, Limor; Haviv, Yinnon (2013). "Unique permutation hashing" (https://doi.org/10.1016%2Fj.tcs.2012.12.047). *Theoretical Computer Science*. **475**: 59–65. doi:10.1016/j.tcs.2012.12.047 (https://doi.org/10.1016%2Fj.tcs.2012.12.047).
10. "CS 3110 Lecture 21: Hash functions" (http://www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec21.html). Section "Multiplicative hashing".
11. Zobrist, Albert L. (April 1970), *A New Hashing Method with Application for Game Playing* (https://www.cs.wisc.edu/techreports/1970/TR88.pdf) (PDF), Tech. Rep. 88, Madison, Wisconsin: Computer Sciences Department, University of Wisconsin.
12. Aho, A.; Sethi, R.; Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley. p. 435. ISBN 0-201-10088-6.
13. Ramakrishna, M. V.; Zobel, Justin (1997). "Performance in Practice of String Hashing Functions" (https://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.18.7520&rep=rep1&type=pdf). *Database Systems for Advanced Applications '97*. DASFAA 1997. pp. 215–224. CiteSeerX 10.1.1.18.7520 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7520). doi:10.1142/9789812819536_0023 (https://doi.org/10.1142%2F9789812819536_0023). ISBN 981-02-3107-5. S2CID 8250194 (https://api.semanticscholar.org/CorpusID:8250194). Retrieved 2021-12-06.
14. Knuth, Donald E. (1975). *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Reading, MA: Addison-Wesley. p. 540.
15. Gonnet, G. (1978). *Expected Length of the Longest Probe Sequence in Hash Code Searching* (Technical report). Ontario, Canada: University of Waterloo. CS-RR-78-46.
16. Knuth, Donald E. (2000). *The Art of Computer Programming, Vol. 3, Sorting and Searching* (2. ed., 6. printing, newly updated and rev. ed.). Boston [u.a.]: Addison-Wesley. ISBN 978-0-201-89685-5.

External links

- Calculate hash of a given value (http://tools.timodenk.com/?p=hash-function) by Timo Denk
- The Goulburn Hashing Function (http://www.sinfocol.org/archivos/2009/11/Goulburn06.pdf) (PDF) by Mayur Patel
- Hash Function Construction for Textual and Geometrical Data Retrieval (https://dspace5.zcu.cz/bitstream/11025/11784/1/Skala_2010_Corfu-NAUN-Hash.pdf) (PDF) Latest Trends on Computers, Vol.2, pp. 483–489, CSCC Conference, Corfu, 2010