In Binary Search Trees I, and II, we considered methods for efficient searching of an ordered collection by using key comparisons. While these methods were indeed very fast, they were limited to O(log N) performance due to the comparison tree inherent in the data structure. Binary search trees are also somewhat complicated, especially when the chance of encountering a degenerate tree is minimized, or removed entirely.

An alternative method for searching uses they key itself an an address into the data structure, thus breaking the O(log N) barrier and allowing searches to be performed with an expected time complexity of O(1), which is as good as it gets when it comes to searching, and algorithms in general.

Unfortunately, not all keys are easily used as a table address. Strings, for example, are not suitable, because array subscripting requires an integer. Even when using integers, if the value is outside of the range of the array, using it as an index will cause an out of bounds access. Hashing is the solution to this problem.

The simplest case when it comes to converting a key into a table address is when you have an integer key. In that case it is a simple matter of forcing the integer into the range of the table, most commonly with the remainder operator:

```
unsigned index = key % N;
```

Unfortunately, this is insufficient most of the time for two reasons. First, keys are more likely to be strings than integers. There has to be some way to convert a string into a table address, or hashing would die a quick death. Second, by forcing the integer into a smaller range, there is a good chance that unique keys will have the same table address after the conversion. This is called a collision, and collisions are the bane of hash function designers.

Hashing a string into a table address is more complicated because instead of a single numeric value there are several, and a good hash function should take them all into account when constructing a table address. Fortunately, hashing strings can be generalized into hashing a sequence of memory blocks, so it is possible to avoid having to develop a different hash function for different key types. The rest of this tutorial will assume that this generalization has been made and we will look at several general hash functions.

Uses for hashing

By far, the most well known use for hashing is to convert a key into an array index for table lookup. Such data structures are called hash tables, and they are incredibly useful for a wide range of fields. For example, in a compiler, a hash table will likely be used for keyword and identifier storage because a compiler needs quick access to this information. A compiler may (should!) also use a hash table for optimizing switch statements, and a number of other purposes. A hash table is also surprisingly well suited to the implementation of a cache, and many web browsers and operating systems will use a hash table for just that.

Probably the second most well known use for hashing is in cryptography, where algorithms that are geared more toward security are used to create digital fingerpints for authentication and data integrity. While lookup hashes are not as complicated or as powerful as cryptographic hashes, adherence to many of the design principles for a cryptographic hash will typically result in a very good lookup hash, so even though there is a clear distinction between hashing for lookup and hashing for cryptography, one can encourage improvement in the other.

This tutorial will focus on hashing for lookup.

Pigeonhole Principle

When hashing a key so that it can be used as an array index, it should be immediately obvious that the size of the array should be at least as large as the input collection. This observation is simple, and some might think that it goes without saying, but it is actually a specialization of the Pigeonhole Principle, where if M items are placed in N buckets, and M is greater than N, one or more buckets contain two or more items. This is one of the two principles that pave the way toward understanding collisions.

A collision is when two keys hash to the same index. The pigeonhole principle proves that no hashing algorithm can hash every key to a unique index if the possible keys exceeds the size of the array. Since most uses of hashing for lookup involve trying to take keys in a broad range and force them into indices for a smaller range, it stands to reason that no hash algorithm can perfectly hash a sequence of unknown keys into unique indices.

Perfect hashing

After having told you that collisions are a necessary evil, and dashing your hopes that there could be a “perfect” hash function, I will now contradict myself and say that there is such a thing as perfect hashing! However, despite the existence of perfect hash algorithms, they are dreadfully difficult to discover for all but the smallest inputs where the number of keys and the exact construction of every possible key is known. As such, while a perfect hash algorithm exists for every input, it is unreasonable to expect that it can be found, so instead of searching for a perfect hash function, it is better to simply build a hash function that minimizes collisions instead of denying them completely.

Constructing a perfect hash

A perfect hash can be created if the number and construction of the keys are known factors. For example, a perfect hash for a list of ten product numbers that are sure to differ in their 4th and 5th digits can be easily constructed:

```
unsigned hash(unsigned pid)
{
    return pid / 1000 % 100;
}
```

However, notice that even though there are only ten product numbers, a table of 100 buckets must be created because the resulting hash values are at least two digits. This is a terrible waste of space, but if the range is forced into an array of 10 buckets, the right digit of the hash values may cause a collision. In such a case, the pigeonhole principle applies, and we can no longer guarantee that this hash algorithm is perfect.

Minimal perfect hashing

A minimal perfect hash algorithm maps every key to every bucket, without any unused buckets. Unlike the perfect hash function above, a minimal perfect hash function would not have any collisions when hashing the 4th and 5th digits of the product numbers and placing the result in an array of 10 buckets. Discovering this function is considerably more difficult, and we are still only dealing with two digit numbers.

A minimal perfect hash algorithm can only be found manually with a great deal of work, or in exceptionally simple cases. For example, hashing ten single digit integers into an array with 10 buckets is trivial, but by adding another digit to the integers, the problem becomes much harder. In general, a minimal perfect hash function can only be found through an exhaustive search of the possibilities.

Algorithms have been devised that will exhaustively search for perfect and minimal perfect hash functions. If one is needed then these tools will be a much better option than trying to discover an algorithm manually. Many of these tools are freely available on the web.

Properties of an ideal hash

With the understanding that collisions will be inevitable in all but the most specialized of cases, a primary goal in developing a hash function for lookup is to minimize collisions. This typically means forcing a uniform distribution of the hash value, much like a random number generator. However, unlike a random number generator, the process must be repeatable so that the same key hashes to the same index, while different keys do not. This goal can be broken down into two general properties of an ideal hash.

An ideal hash will permute its internal state such that every resulting hash value has exactly one input that will produce it. Any hash function that uses every part of the key to calculate the hash value will typically meet this requirement, so general hash functions that only process a part of the key will almost always be very poor because the differing parts of the key may not be the parts involved in creating the hash value. A good example of this is only hashing the first four or five characters of a string, and then using the algorithm to hash URLs that start with “http://”.

A hash function is said to achieve avalanche if the resulting hash value is wildly different if even a single bit is different in the key. This effect aids distribution because similar keys will not have similar hash values. A hash function that distributes the hash values in a uniform manner will minimize collisions and fill the array more evenly. Avalanche is a concept derived from cryptographic hashing, and it offers a way to ensure that a hash function is good when used for table lookup.

Often you will see hash functions that claim to be "best for strings" or “best for integers”. These functions should be avoided because if a hash function is not good for all types of data then it is probably a poor algorithm in general. Sometimes, on the other hand, a hash function may be optimized for a single type for performance reasons. It is good to learn to tell the difference between the two, but a safe practice is to only use general hash functions that are known to be good. This tutorial offers several examples of good general hash functions.

Using existing hash functions

Designing a hash function is a black art. As such, it is always better to use a known good algorithm than to try and invent one. Hash functions are similar to random number generators in many ways, and just as with random number generators, it is easier to design a very poor hash function than to design even a mediocre one. This tutorial will describe several good hash functions so that you can avoid the temptation to write an ad hoc algorithm when the time comes. We will also look at a few not so good hash functions so that you will be able to recognize them in the real world.

Additive hash

Probably the simplest algorithm for hashing a sequence of integral values (such as a string), is to add all of the characters together and then force the range into something suitable for lookup with the remainder of division. I will give an example of this algorithm only because books commonly suggest it in their rush to get past the topic of hash functions on their way to collision resolution methods. This algorithm is very bad:

```
unsigned add_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;

    for (i = 0; i < len; i++)
    {
        h += p[i];
    }

    return h;
}
```

Generally, any hash algorithm that relies primarily on a commutitive operation will have an exceptionally bad distribution. This hash fails to treat permutations differently, so “abc”, “cba”, and “cab” will all result in the same hash value.

Despite the suckiness of this algorithm, the example is useful in that it shows how to create a general hash function. **add_hash** can be used to hash strings, single integers, single floating-point values, arrays of scalar values, and just about anything else you can think of because it is always legal to pun a simple object into an array of unsigned char and work with the individual bytes of the object.

XOR hash

The XOR hash is another algorithm commonly suggested by textbooks. Instead of adding together the bytes of an object as the additive hash does, the XOR hash repeatedly folds the bytes together to produce a seemingly random hash value:

```
unsigned xor_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;

    for (i = 0; i < len; i++)
    {
        h ^= p[i];
    }

    return h;
}
```

Unfortunately, this algorithm is too simple to work properly on most input data. The internal state, the variable **h**, is not mixed nearly enough to come close to achieving avalanche, nor is a single XOR effective at permuting the internal state, so the resulting distribution, while better than the additive and multiplicative hashes, is still not very good.

Rotating hash

```
unsigned rot_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;

    for (i = 0; i < len; i++)
    {
        h = (h << 4) ^ (h >> 28) ^ p[i];
    }

    return h;
}
```

The rotating hash is identical to the XOR hash except instead of simply folding each byte of the input into the internal state, it also performs a fold of the internal state before combining it with the each byte of the input. This extra mixing step is enough to give the rotating hash a much better distribution. Much of the time, the rotating hash is sufficient, and can be considered the minimal acceptable algorithm. Notice that with each improvement, the internal state is being mixed up more and more. This is a key element in a good hash function.

Bernstein hash

Dan Bernstein created this algorithm and posted it in a newsgroup. It is known by many as the Chris Torek hash because Chris went a long way toward popularizing it. Since then it has been used successfully by many, but despite that the algorithm itself is not very sound when it comes to avalanche and permutation of the internal state. It has proven very good for small character keys, where it can outperform algorithms that result in a more random distribution:

```
unsigned djb_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;

    for (i = 0; i < len; i++)
    {
        h = 33 * h + p[i];
    }

    return h;
}
```

Bernstein's hash should be used with caution. It performs very well in practice, for no apparently known reasons (much like how the constant 33 does better than more logical constants for no apparent reason), but in theory it is not up to snuff. Always test this function with sample data for every application to ensure that it does not encounter a degenerate case and cause excessive collisions.

Modified Bernstein

A minor update to Bernstein's hash replaces addition with XOR for the combining step. This change does not appear to be well known or often used, the original algorithm is still recommended by nearly everyone, but the new algorithm typically results in a better distribution:

```
unsigned djb_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;

    for (i = 0; i < len; i++)
    {
        h = 33 * h ^ p[i];
    }

    return h;
}
```

Shift-Add-XOR hash

The shift-add-XOR hash was designed as a string hashing function, but because it is so effective, it works for any data as well with similar efficiency. The algorithm is surprisingly similar to the rotating hash except a different choice of constants for the rotation is used, and addition is a preferred operation for mixing. All in all, this is a surprisingly powerful and flexible hash. Like many effective hashes, it will fail tests for avalanche, but that does not seem to affect its performance in practice.

```
unsigned sax_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;

    for (i = 0; i < len; i++)
    {
        h ^= (h << 5) + (h >> 2) + p[i];
    }

    return h;
}
```

FNV hash

The FNV hash, short for Fowler/Noll/Vo in honor of the creators, is a very powerful algorithm that, not surprisingly, follows the same lines as Bernstein's modified hash with carefully chosen constants. This algorithm has been used in many applications with wonderful results, and for its simplicity, the FNV hash should be one of the first hashes tried in an application. It is also recommended that the FNV website (http://www.isthe.com/chongo/tech/comp/fnv/) be visited for useful descriptions of how to modify the algorithm for various uses.

```
unsigned fnv_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 2166136261;
    int i;

    for (i = 0; i < len; i++)
    {
        h = (h * 16777619) ^ p[i];
    }

    return h;
}
```

One-at-a-Time hash

Bob Jenkins is a well known authority on designing hash functions for table lookup. In fact, one of his hashes is considered state of the art for lookup, which we will see shortly. A considerably simpler algorithm of his design is the One-at-a-Time hash:

```
unsigned oat_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0;
    int i;

    for (i = 0; i < len; i++)
    {
        h += p[i];
        h += (h << 10);
        h ^= (h >> 6);
    }

    h += (h << 3);
    h ^= (h >> 11);
    h += (h << 15);

    return h;
}
```

This algorithm quickly reaches avalanche and performs very well. This function is another that should be one of the first to be tested in any application, if not the very first. This algorithm is my personal preference as a first test hash, and it has seen effective use in several high level scripting languages as the hash function for their associative array data type.

JSW hash

This is a hash of my own devising that combines a rotating hash with a table of randomly generated numbers. The algorithm walks through each byte of the input, and uses it as an index into a table of random integers generated by a good random number generator. The internal state is rotated to mix it up a bit, then XORed with the random number from the table. The result is a uniform distribution if the random numbers are uniform. The size of the table should match the values in a byte. For example, if a byte is eight bits then the table would hold 256 random numbers:

```
unsigned jsw_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 16777551;
    int i;

    for (i = 0; i < len; i++)
    {
        h = (h << 1 | h >> 31) ^ tab[p[i]];
    }

    return h;
}
```

In general, this algorithm is among the better ones that I have tested in terms of both distribution and performance. I may be slightly biased, but I feel that this function should be on the list of the first to test in a new application using hash lookup.

ELF hash

The ELF hash function has been around for a while, and it is believed to be one of the better algorithms out there. In my experience, this is true, though ELF hash does not perform sufficiently better than most of the other algorithms presented in this tutorial to justify its slightly more complicated implementation. It should be on your list of first functions to test in a new lookup implementation:

```
unsigned elf_hash(void *key, int len)
{
    unsigned char *p = key;
    unsigned h = 0, g;
    int i;

    for (i = 0; i < len; i++)
    {
        h = (h << 4) + p[i];
        g = h & 0xf0000000L;

        if (g != 0)
        {
            h ^= g >> 24;
        }

        h &= ~g;
    }

    return h;
}
```

Jenkins hash

The dreaded Jenkins hash has been thoroughly tested and passes all kinds of tests for avalanche and permutations. As such it is considered to be one of the best and most thoroughly analyzed algorithms on the market presently. Unfortunately, it is also ridiculously complicated compared to the other hashes examined in this tutorial:

```
#define hashsize(n) (1U << (n))
#define hashmask(n) (hashsize(n) - 1)

#define mix(a,b,c) \
{ \
    a -= b; a -= c; a ^= (c >> 13); \
    b -= c; b -= a; b ^= (a << 8); \
    c -= a; c -= b; c ^= (b >> 13); \
    a -= b; a -= c; a ^= (c >> 12); \
    b -= c; b -= a; b ^= (a << 16); \
    c -= a; c -= b; c ^= (b >> 5); \
    a -= b; a -= c; a ^= (c >> 3); \
    b -= c; b -= a; b ^= (a << 10); \
    c -= a; c -= b; c ^= (b >> 15); \
}

unsigned jen_hash(unsigned char *k, unsigned length, unsigned initval)
{
    unsigned a, b;
    unsigned c = initval;
    unsigned len = length;

    a = b = 0x9e3779b9;

    while (len >= 12)
    {
        a += (k[0] + ((unsigned)k[1] << 8) + ((unsigned)k[2] << 16) + ((unsigned)k[3] << 24));
        b += (k[4] + ((unsigned)k[5] << 8) + ((unsigned)k[6] << 16) + ((unsigned)k[7] << 24));
        c += (k[8] + ((unsigned)k[9] << 8) + ((unsigned)k[10] << 16) + ((unsigned)k[11] << 24));

        mix(a, b, c);

        k += 12;
        len -= 12;
    }

    c += length;

    switch (len)
    {
    case 11: c += ((unsigned)k[10] << 24);
    case 10: c += ((unsigned)k[9] << 16);
    case 9: c += ((unsigned)k[8] << 8);
        /* First byte of c reserved for length */
    case 8: b += ((unsigned)k[7] << 24);
    case 7: b += ((unsigned)k[6] << 16);
    case 6: b += ((unsigned)k[5] << 8);
    case 5: b += k[4];
    case 4: a += ((unsigned)k[3] << 24);
    case 3: a += ((unsigned)k[2] << 16);
    case 2: a += ((unsigned)k[1] << 8);
    case 1: a += k[0];
    }

    mix(a, b, c);

    return c;
}
```

For details on how this algorithm works, feel free to visit Bob Jenkins' website (http://burtleburtle.net/bob/).

Designing hash functions

Designing a hash function is more trial and error with a touch of theory than any well defined procedure. For example, beyond making the connection between random numbers and the desirable random distribution of hash values, the better part of the design for my JSW hash involved playing around with constants to see what would work best.

I would also be lying if I said that the choice of operations was not almost completely random trial and error by pulling from a pool of combinations of XOR, OR, AND, addition, subtraction, and multiplication. The biggest design element of JSW hash is using the table of random numbers keyed on each byte of the input to act as a mixing step. Then it was simply a matter of choosing a less complicated hash algorithm to paste that addition onto, and finally the fiddling and testing to get the algorithm performing well.

I will not claim that hash function design is always like that, but the evidence certainly suggests it. If you design hash functions in a deterministic way, I would love to hear from you.

Testing hash functions

The most important test for a hash function is distribution on a sample of the expected input. No hash function is best for all possible inputs, which is why I introduced a large number of algorithms in this tutorial. Sometimes one of the good hash functions will work better than the others, and that one should be used. Other times they will all be fairly equal and you can choose at random. The key here is that a hash function should never be used blindly without testing it, no matter how good the author says it is. Often, hash tests are designed around a hash function, and that introduces a bias in favor of that function.

A test for distribution is a simple affair. Just take a sample of the expected input and insert it into a chained hash table with the chosen function. Work out the statistics of chain length, and you can determine how many collisions were detected. Alternatively, a quick test for collisions can be performed without the use of a hash table by simply taking a sample input with no duplicates and counting collisions with an existence table:

```
function collisions: file, n
begin
    table[n] = {0}

    while get ( file, buffer ) do
        table[hash ( buffer ) % n] +:= 1
    loop

    for i := 0 to n do
        println table[i]
    loop
end
```

Another alternative is to plug the hash function into my jsw_hlib (../libs/jsw_hlib.html) library and run it with a few files of sample input. This saves you the trouble of having to work out a collision test, or designing and implementing a working hash table while calculating several more useful measurements than just collisions.