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 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;
}
```