Hash Functions

Hash functions

A hash function maps keys to small integers (buckets). An ideal hash function maps the keys to the integers in a random-like manner, so that bucket values are evenly distributed even if there are regularities in the input data.

This process can be divided into two steps:

- Map the key to an integer.
- Map the integer to a bucket.

Taking things that really aren't like integers (e.g. complex record structures) and mapping them to integers is icky. We won't discuss this. Instead, we will assume that our keys are either integers, things that can be treated as integers (e.g. characters, pointers) or 1D sequences of such things (lists of integers, strings of characters).

Simple hash functions

The following functions map a single integer key (k) to a small integer bucket value h(k). m is the size of the hash table (number of buckets).

Division method (Cormen) Choose a prime that isn't close to a power of 2. h(k) = k mod m. Works badly for many types of patterns in the input data.

Knuth Variant on Division h(k) = k(k+3) mod m. Supposedly works much better than the raw division method.

Multiplication Method (Cormen). Choose m to be a power of 2. Let A be some random-looking real number. Knuth suggests M = 0.5*(sqrt(5) - 1). Then do the following:

```
s = k*A
x = fractional part of s
h(k) = floor(m*x)
```

This seems to be the method that the theoreticians like.

To do this quickly with integer arithmetic, let w be the number of bits in a word (e.g. 32) and suppose m is 2^p. Then compute:

```
s = floor(A * 2^w)
x = k*s
h(k) = x >> (w-p)      // i.e. right shift x by (w-p) bits
                       // i.e. extract the p most significant bits from x
```

Hashing sequences of characters

The hash functions in this section take a sequence of integers k=k1,...,kn and produce a small integer bucket value h(k). m is the size of the hash table (number of buckets), which should be a prime number. The sequence of integers might be a list of integers or it might be an array of characters (a string).

The specific tuning of the following algorithms assumes that the integers are all, in fact, character codes. In C++, a character is a char variable which is an 8-bit integer. ASCII uses only 7 of these 8 bits. Of those 7, the common characters (alphabetic and number) use only the low-order 6 bits. And the first of those 6 bits primarily indicates the case of characters, which is relatively insignificant. So the following algorithms concentrate on preserving as much information as possible from the last 5 bits of each number, and make less use of the first 3 bits.

When using the following algorithms, the inputs ki **must** be unsigned integers. Feeding them signed integers may result in odd behavior.

For each of these algorithms, let h be the output value. Set h to 0. Walk down the sequence of integers, adding the integers one by one to h. The algorithms differ in exactly how to combine an integer ki with h. The final return value is h mod m.

CRC variant: Do a 5-bit left circular shift of h. Then XOR in ki. Specifically:

```
highorder = h & 0xf8000000     // extract high-order 5 bits from h
                               // 0xf8000000 is the hexadecimal representation
                               //   for the 32-bit number with the first five
                               //   bits = 1 and the other bits = 0
h = h << 5                     // shift h left by 5 bits
h = h ^ (highorder >> 27)      // move the highorder 5 bits to the low-order
                               //   end and XOR into h
h = h ^ ki                     // XOR h and ki
```

PJW hash (Aho, Sethi, and Ullman pp. 434-438): Left shift h by 4 bits. Add in ki. Move the top 4 bits of h to the bottom. Specifically:

```
// The top 4 bits of h are all zero
h = (h << 4) + ki              // shift h 4 bits left, add in ki
g = h & 0xf0000000             // get the top 4 bits of h
if (g != 0)                    // if the top 4 bits aren't zero,
   h = h ^ (g >> 24)           //   move them to the low end of h
   h = h ^ g
// The top 4 bits of h are again all zero
```

PJW and the CRC variant both work well and there's not much difference between them. We believe that the CRC variant is probably slightly better because

- It uses all 32 bits. PJW uses only 24 bits. This is probably not a major issue since the final value m will be much smaller than either.
- 5 bits is probably a better shift value than 4. Shifts of 3, 4, and 5 bits are all supposed to work OK.
- Combining values with XOR is probably slightly better than adding them. However, again, the difference is slight.

BUZ hash: Set up a function R that takes 8-bit character values and returns random numbers. This function can be precomputed and stored in an array. Then, to add each character ki to h, do a 1-bit left circular shift of h and then XOR in the random value for ki. That is:

```
highorder = h & 0x80000000    // extract high-order bit from h
h = h << 1                    // shift h left by 1 bit
h = h ^ (highorder >> 31)     // move them to the low-order end and
                              // XOR into h
h = h ^ R[ki]                 // XOR h and the random value for ki
```

Rumor has it that you may have to run a second hash function on the output to make it random enough. Experimentally, this function produces good results, but is a bit slower than the CRC variant and PJW.

References

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest (1990) Introduction to Algorithms, MIT Press, Cambridge MA and McGraw-Hill, New York.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

Donald Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, 1973.