Additional Notes on Hash Functions

(adapted from http://www.sparknotes.com/cs/searching/hashtables/section2.rhtml)

There are multiple ways for constructing a hash function. A hash function takes the data (often a string) as input and returns an integer ***in the entire range of possible indices*** into the hash table. A hash function must also be ***fast to evaluate***, for obvious reasons. Every candidate hash function, including bad ones, should meet these requirements. Given several candidate hash functions, what other characteristics can we use to assess whether a particular one is good?

Characteristics of a Good Hash Function

There are four main characteristics of a good hash function: 1) The ***hash value is fully determined by the data*** being hashed. 2) The hash function ***uses all the input data***. 3) The hash function ***"uniformly" distributes the data across the entire set of possible hash values***. 4) The hash function generates ***very different hash values for similar data items***.

Let's examine why each of these is important: **Rule 1**: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values. **Rule 2**: If the hash function doesn't use all the input data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions. **Rule 3**: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table. **Rule 4**: In real world applications, many data sets contain very similar data elements. We would like these data elements to still be distributable over a hash table.

So let's take as an example the following hash function:

```
int hash(char *str, int table_size)
{
   int sum = 0;
   if (str == NULL) return -1;       // make sure string is valid
   for( ; *str; str++ ) sum += *str; // sum up all characters (ASCII values) in string
   return sum % table_size;          // return sum mod table size
}
```

Which rules does it break and satisfy? **Rule 1**: Satisfied. The hash value is fully determined by the data being hashed. The hash value is just the sum of all the input characters. **Rule 2**: Satisfied. Every character is summed. **Rule 3**: Not satisfied. From looking at it, it isn't obvious that it doesn't uniformly distribute the strings, but if you were to analyze this function for a large input you would see certain statistical properties bad for a hash function. **Rule 4**: Not satisfied. Hash the string "bog". Now hash the string "gob". They're the same. Slight variations in the string should result in different hash values, but with this function they often don't.

So this hash function isn't so good. It's a good introductory example but not so good in the long run.

There are many possible ways to construct a better hash function (doing a web search will turn up hundreds). Below are a few decent examples of hash functions:

```
/* Peter Weinberger's */
int hashpjw(char *s)
{
   char *p;
   unsigned int h, g;
   h = 0;
   for(p = s; *p != '\0'; p++)
   {
      h = (h << 4) + *p;
      if (g = h & 0xF0000000)
      {
         h ^= g >> 24;
         h ^= g;
      }
   }
   return h % 211;
}
```

Another one:

```
/* UNIX ELF hash
 * Published hash algorithm used in the UNIX ELF format for object files
 */
unsigned long hash(char *name)
{
   unsigned long h = 0, g;

   while ( *name )
   {
      h = ( h << 4 ) + *name++;
      if ( g = h & 0xF0000000 )
      h ^= g >> 24;
      h &= ~g;
   }

   return h;
}
```

Yet another one:

```
/* This algorithm was created for the sdbm (a reimplementation of ndbm)
 * database library and seems to work relatively well in scrambling bits
 */
static unsigned long sdbm(unsigned char *str)
{
   unsigned long hash = 0;
   int c;
   while (c = *str++) hash = c + (hash << 6) + (hash << 16) - hash;
   return hash;
}
```

Still yet another one:

```
/* djb2
 * This algorithm was first reported by Dan Bernstein
 * many years ago in comp.lang.c
 */
unsigned long hash(unsigned char *str)
{
   unsigned long hash = 5381;
   int c;
   while (c = *str++) hash = ((hash << 5) + hash) + c;  // hash*33 + c
   return hash;
}
```

More of still yet another one:

```
char XORhash( char *key, int len)
{
   char hash;
   int  i;
   for (hash = 0, i = 0; i < len; ++i) hash = hash ^ key[i];
   return (hash % 101); /* 101 is prime */
}
```

You get the idea... there are many possible hash functions. For coding up a hash function quickly, djb2 is usually a good candidate as it is easily implemented and has relatively good statistical properties.