9.2 Hash Functions

The Meriam-Webster dictionary defines a hash as *a restatement of something that is already known*. The ultimate goal will be to *hash* any object to a value from 0 to $M - 1$. Any function that performs this operation is called a *hash function*.

9.2.1 Design

In order to simplify the overall process of mapping an arbitrary object onto a number on the range 0 to $M - 1$, we will divide this operation into two steps:

A function $h{:}D \to \mathbf{Z}_{2^{32}}$ maps our space of objects onto positive 32-bit integers (from 0 to $2^{32} - 1$). A second function $h_M{:}\ \mathbf{Z}_{2^{32}} \to \mathbf{Z}_M$ maps positive 32-bit integers onto the range 0, ..., $M - 1$. This is shown graphically in Figure 1.

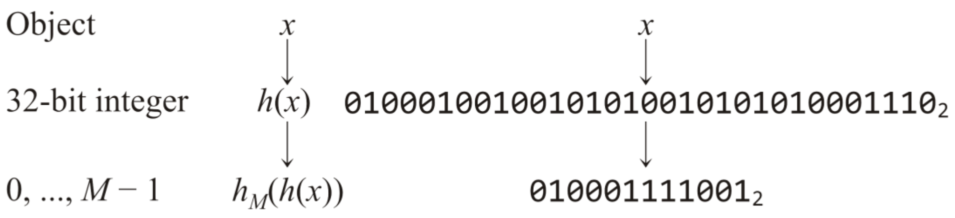

Figure 1. The process of mapping an object onto a positive 32-bit number and then further mapped onto an integer on the range 0, ..., $M - 1$.

By breaking the process into two steps, the second function h_M can be written to account for any possible value of M and thus the original hash function is need not concern itself over which interval the hash value must appear.

In C++, the first hash function h will usually be implemented as a member function

```
unsigned int Class_name::hash() const;
```

while the second hash function h_M would likely be defined within the hash-table class:

```
unsigned int Hash_table::hash_M( unsigned int ) const;
```

Thus, the function

```
template <typename Type>
void Hash_table::insert( Type const &obj ) {
    int bin = hash_M( obj.hash() );
    // ...
}
```

9.2.2 Properties

We will require certain properties of each of these hash functions. The properties for the function *h* are:

1a. It must be fast—ideally Θ(1),
1b. It must be deterministic: it must always return the same value for the same object,
1c. If two objects *x* and *y* are considered *equal*, their hash values must be equal, *i.e.*, $h(x) = h(y)$,
1d. If two objects are chosen at random, there should be a very low probability (approximately 1 in 2^{32}) that they have the same hash value, and
1e. The distribution of hash values in 0 to $2^{32} - 1$ should be approximately even (that is, the distribution of values is either uniform distribution or a Poisson distribution).

The second hash function h_M must have the properties that:

2a. It must be Θ(1),
2b. It must be deterministic: given a positive integer *n* and range *M*, $h_M(n)$ must always return the same value,
2c. If two integers are chosen at random, there should be a 1 in *M* chance that they will map onto the same integer in 0, ..., $M - 1$, and
2d. It must create an even distribution in 0, ..., $M - 1$; that is, there should not be any clustering.

In this topic, we will look at various ways of finding appropriate functions *h* that satisfy Properties 1a through 1e and the next topic will look at functions h_M that satisfy Properties 2a through 2d.

9.2.3 Types of Hash Functions

We will look at two general classes of hash functions:

1. Predetermined hash functions (either implicit or explicit), and
2. Arithmetic hash functions (implicitly defined).

9.2.4 Predetermined Hash Values

One of the easiest ways to assign a hash value to an object is to give each newly created object its own unique hash value in the constructor. This can most easily be achieved by having a counter that is incremented each time an object is constructed:

```
class Class_name {
    private:
        unsigned int hash_value;
        // Static variables are class variables shared by all instances of the class
        static unsigned int hash_count;
    public:
        Class_name();
        unsigned int hash() const;
};

// Static variables must be initialized outside the class definition:
unsigned int Class_name::hash_count = 0;

Class_name::Class_name() {
    hash_value = hash_count;
    ++hash_count;
}

unsigned int Class_name::hash() const {
    return hash_value;
}
```

All UW Co-op students will have two pre-determined unique numbers hash values that could be considered hash values: the UW Student ID Number (*e.g.*, 20123456) and the Canadian Social Insurance Number (S.I.N.) (*e.g.*, 123-456-789). Any 9-digit number can be interpreted as a 32-bit number.

Another possibility is to use the address of the object that was created. This is useful if the hash value of the object is only required for the duration of its existence in memory:

```
unsigned int Class_name::hash() const {
    return reinterpret_cast<unsigned int>( this );
}
```

9.2.4.1 Weaknesses with Predetermined Hash Values

There are some cases where such predetermined hash values are not appropriate. Suppose, for example, we define two strings that are identical:

```
#include <string>
std::string str1 = "Hello world!";
std::string str2 = "Hello world!";
```

The addresses of these two strings will be different even though their contents are the same.

Suppose we define a rational number class:

```
Rational p( 1, 2 );
Rational q( 3, 6 );
```

In this case, both of these represent the same rational number ½; therefore, any hash function should return the same value: `p.hash() == q.hash()`.

The usual solution is to create a hash function that is derived from the member variables of the objects using arithmetic functions.

9.2.5 Arithmetic Hash Functions

We will look at defining hash values for rational numbers, double-precision floating-point numbers and strings. In each case, we try to satisfy all the requirements 1a through 1e.

9.2.5.1 A Rational Number Class

```
class Rational {
    private:
        int numer, denom;
    public:
        Rational( int, int );
        unsigned int Rational::hash() const;
};

unsigned int Rational::hash() const {
    return static_cast<unsigned int>(numer) + static_cast<unsigned int>(denom);
}
```

Unfortunately, this scheme does not work too well: All of these will have the same hash value:

```
Rational p(  1, 2 );
Rational q( -1, 4 );
Rational r(  2, 1 );
Rational s( -2, 5 );
```

and it would not be too difficult to find more. As an alternative, consider multiplying the denominator by a very large prime number:

```
class Rational {
    private:
        int numer, denom;
    public:
        Rational( int, int );
};

unsigned int Rational::hash() const {
        return static_cast<unsigned int>( numer ) + 429496751*static_cast<unsigned int>(
          denom );
}
```

Let us consider the four requirements for a hash function:

Requirement 1a: This is a still a fast hash function running in Θ(1) time.

Requirement 1b: This hash value is deterministic—it will always be the same for a rational number with the same numerator and denominator.

We will pass on Requirement 1c to go onto Requirement 1d:

In this case, two rational numbers that are close in both the numerator and denominator will have different hash values. For example, consider Table 1 which lists a number of rational numbers and their corresponding hash values. Recall that integer operations wrap on overflow.

Table 1. Rational numbers a/b and their hash values found by multiplying calculating a + 429496751b.

Rational Object	**Hash Value**
`Rational(  0,   1 )`	429496751
`Rational(  1,   1 )`	429496752
`Rational(  2,   1 )`	429496753
`Rational(  1,   2 )`	858993503
`Rational(  2,   3 )`	1288490255
`Rational( 99, 100 )`	2239

Aside: it is very easy to find a large prime number in Maple. Give the procedure **`nextprime`** a large integer value and it will return the next largest prime number.

```
> nextprime( 9385293292438 );
                                  9385293292439
> nextprime( % );  # The symbol % in Maple is similar to 'ans' in Matlab
                                  9385293292489
```

Given that you know the prime number 429496751, it is easy to generate two rational numbers that have the same hash value; however, finding two numbers at random requires more work. It required the creation of 1.5 billion random rational numbers before the following three rationals were found to have the same hash values:

Fixed Rational	Random Rational with Same Hash Value
0/1	1327433019/800977868
1/2	534326814/1480277007
2/3	820039962/1486995867

With respect to Requirement 1e, if we were to generate all possible pairs of numerators and denominators, we would find that every possible has value has exactly the same number of different rational numbers mapping onto it: 2^{32}.

The last requirement is Requirement 1d: two objects that are considered equal should have the same hash value. With our default constructor, this is not possible:

Rational Object	**Hash Value**
`Rational( 1, 2 )`	`858993503`
`Rational( 2, 4 )`	`1717987006`

Thus, 1/2 and 2/4 are conceptually the same rational number but they have different hash values. To solve this, we could write a greatest common divisor function:

```
int gcd( int a, int b) {
    while( true ) {
        if ( a == 0 ) {
            return (b >= 0) ? b : -b;
        }
        b %= a;
        if ( b == 0 ) {
            return (a >= 0) ? a : -a;
        }
        a %= b;
    }
}
```

and then rewrite our constructor as:

```
Rational::Rational( int a, int b ):numer(a), denom(b) {
    int divisor = gcd( numer, denom );
    numer /= divisor;
    denom /= divisor;
}
```

Now, both 1/2 and 2/4 will normalize to 1/2. This, however, isn't all: consider:

Rational Object	**Hash Value**
`Rational(  1,  2 )`	`858993503`
`Rational( -1, -2 )`	`3435973793`

Again, 1/2 and -1/-2 are conceptually the same rational number but, too, have different hash values. Thus, we can again normalize these by ensuring that if there is a negative sign, it only appears in the numerator:

```
Rational::Rational( int a, int b ):numer(a), denom(b) {
    int divisor = gcd( numer, denom );
    divisor = (denom >= 0) ? divisor : -divisor;
    numer /= divisor;
    denom /= divisor;
}
```

With these changes, our hash function satisfies all of the requirements.

9.2.5.2 Hash Values for Double-precision Floating-point Numbers (aside)

As an aside, recall that a double-precision floating-point number is 64 bits. To convert this to a 32-bit hash value, we could interpret the first 32 and the last 32 bits as two 32-bit integers and repeat the processes from the rational number class. To do this, we will reinterpret the address of a double as the address of an unsigned integer and assign this to the pointer `ptr`. Next, `ptr[0]` treats the first 32 bits as an unsigned integer and `ptr[1]` treats the second 32 bits also as an unsigned integer. Thus, we have:

```
unsigned int hash( double d ) {
    unsigned int *ptr = reinterpret_cast<unsigned int *>( &d );
    return ptr[0] + 429496751*ptr[1];
}
```

9.2.5.3 Strings

In order to create a hash function for a string, because the hash values of identical strings should be the same, it is necessary to deterministically build the hash value from the characters (*e.g.*, we cannot use the address). Now, in C++, the string class has a member function `length()` and the individual characters can be accessed using indices as if the string was an array. Each character can be treated as an unsigned integer between the values of 0 and 255.

9.2.5.3.1 First Attempt

We could simply add the characters:

```
unsigned int hash( const string &str ) {
    unsigned int hash_vaalue = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value += str[k];
    }

    return hash_value;
}
```

Unfortunately, this creates neither large nor evenly distributed hash values. If we consider all words in Moby^{TM} Words II (Grady Ward), we note that there is a clear pattern emerging with no word having a hash value greater than 4000, as is shown in Figure 2.

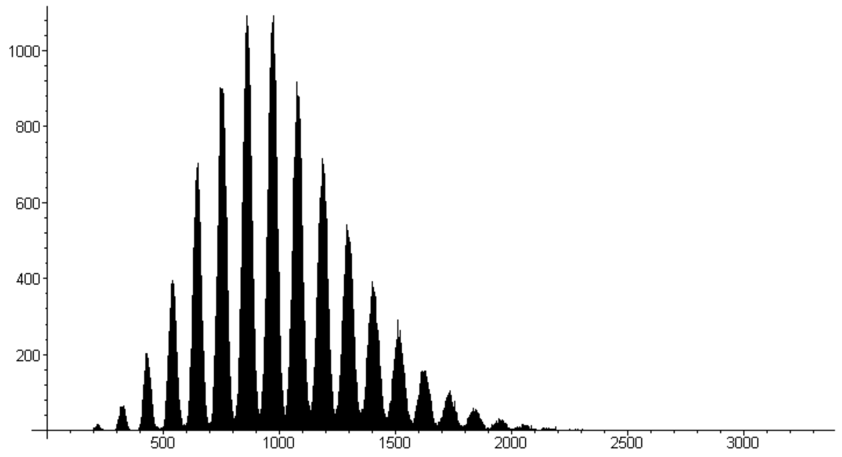

Figure 2. The hash values of all words in Moby™ Words II by Grady Ward.

Note that there are hash values in the range from 800 to 1000 for which there are over 1100 words mapping to the same hash value while for all hash values greater than 4000, there are no words mapping to such a value.

9.2.5.3.2 Horner's Rule

We could consider each character in a string of length *n* to be the coefficient of a $(n-1)$st-degree polynomial in a variable *x*:

$$p(x) = c_0x^{n-1} + c_1x^{n-2} + \cdots + c_{n-3}x^2 + c_{n-2}x + c_{n-1}.$$

We can then use Horner's rule to evaluate this polynomial at some prime value, say $x = 12347$.

```
unsigned int hash( string const &str ) {
    unsigned int hash_value = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value = 12347*hash_value + str[k];
    }

    return hash_value;
}
```

In order to determine whether or not this is a good hash function, we need to quickly introduce Poisson distributions (covered in Appendix A). We will test all 354985 strings in the Moby™ Words II file `single.txt` and divide the interval up into just as many approximately equally sized intervals of width 12099. We then count the proportion of these intervals that have 0, 1, 2, *etc*. hash values mapped into their interval.

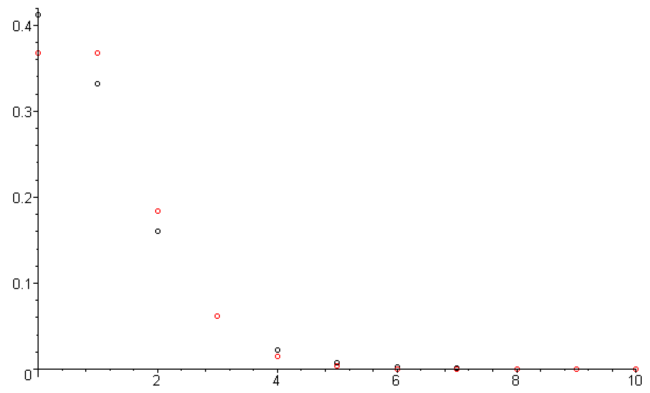

Figure 3. The difference between the observed proportions (black) and the expected proportions from a Poisson distribution with $\lambda = 1$.

Calculating the $\chi^2 = 0.044$, we see that we have approximately an 85 % confidence that this data comes from a Poisson distribution and therefore we can say that this is a good hash function.

9.2.5.3.3 Horner's Rule on Longer Strings

Suppose we have a significantly longer string, for example, consider this string by J.R.R. Tolkien in *The Lord of the Rings*:

```
string str = "A Elbereth Gilthoniel,\n";
str +=       "Silivren penna miriel\n";
str +=       "O menal aglar elenath!\n";
str +=       "Na-chaered palan-diriel\n";
str +=       "O galadhremmin ennorath,\n";
str +=       "Fanuilos, le linnathon\n";
str +=       "nef aear, si nef aearon!";
```

If we attempt to apply Horner's rule to this string, the run time will be Θ(n). While using Horner's rule may be appropriate for shorter strings, it will not be appropriate for larger strings such as paragraphs, chapters, books, *etc*.

One alternative is to use only the locations $2^k - 1$ for $k = 0, 1, 2, \ldots$, as is shown below:

```
AƀElbereth Gilthoniel,\n
Silivrenƀpenna miriel\n
O menal aglar elenath!\n
Na-chaered palan-diriel\n
O galadhremmin ennorath,\n
Fanuilos, le linnathon\n
nef aear, si nef aearon!
```

In this case, the run time is now Θ(ln(n)) in the length of the string.

```
unsigned int hash( string const &str ) {
    unsigned int hash_value = 0;

    for ( int k = 1; k <= str.length(); k *= 2 ) {
        hash_value = 12347*hash_value + str[k - 1];
    }

    return hash_value;
}
```

Note that while this hash function may be very appropriate for creating hash values, it is not appropriate for the hash functions used in digital signatures. For digital signatures, it is absolutely necessary the hash value changes if even one character in the text changes.

Appendix A. Poisson Distributions

Suppose we have two individuals with rifles shooting at a target that is divided into 9 squares, as is shown in Figure 3.

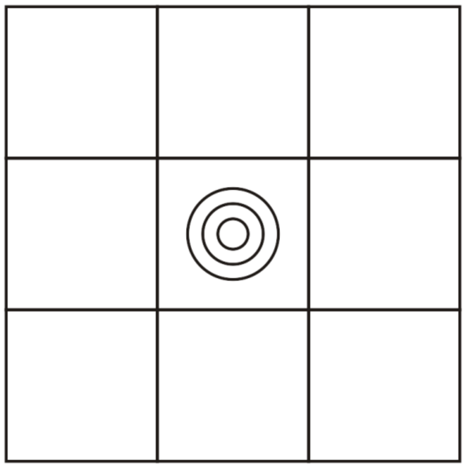

Figure 3. A target divided into nine square regions.

Suppose one individual is so inexperienced that after firing 27 bullets, he has an equal chance of hitting each square on the target with any one shot. The other individual, Tanya Chernova, is a sniper and has a much higher probability of striking the middle square. Their targets look like those in Figure 4.

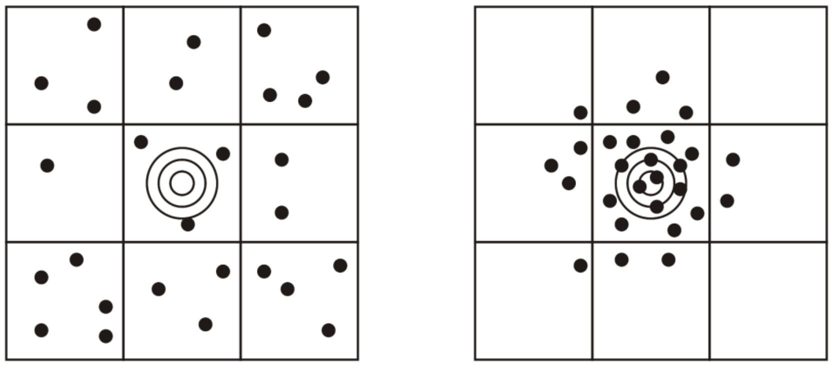

Figure 4. The targets of inexperienced and experienced individuals.

How can we tell if the first individual is actually shooting randomly? With 27 shots and nine squares, one would expect, on average, $\lambda = 3$ shots per square. On the right target, one square has 15 hits, two each have three, two, one, and no hits—this is likely not random. The left target, however, has a much more even distribution:

Hits per Square	0	1	2	3	4	5	6	7
Count	0	1	2	3	2	1	0	0

How can we tell if this is actually random? Fortunately, if we expect, on average, λ hits per square, such a pattern can be described by a Poisson distribution. This indicates that we would expect a proportion of squares equal to

$$e^{-\lambda}\frac{\lambda^k}{k!}$$

to contain k hits. If $\lambda = 3$, we get the following table:

k	0	1	2	3	4	5	6	7	8	9	10
Proportion	0.050	0.149	0.224	0.224	0.168	0.101	0.050	0.022	0.008	0.002	0.001

Therefore, we would expect approximately 0.224 × 9 = 2.016 squares to have two and three shots, while only 0.168 × 2 = 1.512 squares would have four shots. We would have less than a 1 % chance that, if the shots were random, that one of the squares would have 8 hits.

How can we use this to determine if a hash function is good?

Suppose we divide the integers 0 through $2^{32} - 1$ into L equally sized sub-intervals. If we then choose L randomly chosen 32-bit integers, we would expect, on average, $\lambda = 1$ value to fall in each of the intervals. For example, suppose we choose sixteen 32-bit numbers and each interval has a width of $2^{28} = 268435456$. This was done three times, and Table 2 shows the count for how often an integer fell into the kth interval.

Table 2. Three runs of choosing 32-bit numbers and determining which of 16 bins they fell into.

	Interval															
Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
A	1	2	1	0	1	1	1	1	0	1	3	2	0	1	0	1
B	2	0	2	0	1	0	2	1	1	1	1	1	1	2	1	0
C	1	1	0	2	0	0	0	1	2	3	1	0	3	0	1	1

As before, we will count the number of times that a bin is empty, how often a bin has one hit, how often a bin has two hits, *etc*. This is summarized in Table 3.

Table 3. The proportion of bins with n hits.

	Proportion Containing *n* Hits				
Run	0	1	2	3	4
A	0.25	0.5625	0.125	0.0625	0
B	0.25	0.5	0.25	0	0
C	0.375	0.375	0.125	0.125	0

We are expecting $\lambda = 1$ hit per bin and thus it is not unexpected that in each case, the highest proportion of bins contain one hit. But how often should we expect a bin to have two, three, or no hits?

The Poisson distribution tells us that if these are random, then the proportion of bins that have *n* hits will be $\frac{1}{ek!}$. We can tabulate the values shown in Table 4.

	Proportion of Bins with *n* Hits								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Proportion	0.36788	0.36788	0.18394	0.06131	0.01533	0.00307	0.00051	0.00007	0.00001

Thus, approximately 37 % of the bins should have no hits, 37 % should have one hit, 18 % should have two hits, only 6 % should have three hits, 1.5 % should have four hits, *etc*. as is shown in Figure 3.

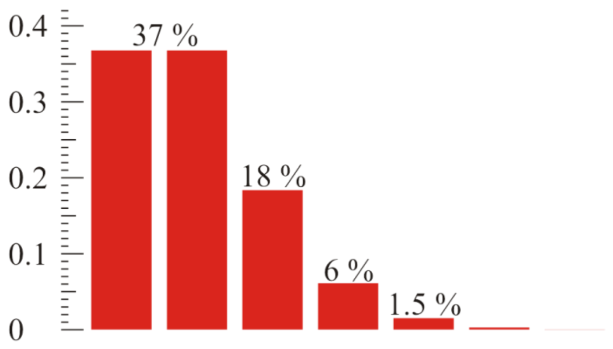

Figure 3. The distribution and cumulative distributions of the Poisson distribution with $\lambda = 1$.

The best way to determine if the distributions in Table 3 actually match are to compare the cumulative distributions, as are tabulated in Table 5.

Table 5. The cumulative proportion of bins with ≤ *n* hits.

	Proportion Containing *n* Hits				
Run	0	1	2	3	4
A	0.25	0.5625	0.125	0.0625	0
B	0.25	0.5	0.25	0	0
C	0.375	0.375	0.125	0.125	0
Expected	0.36788	0.36788	0.18394	0.06131	0.01533

If we wish to test how good this test is, we must calculate the sum of the squares of the differences between the observed proportions o_k and the expected proportions $e_k = e^{-\lambda}\frac{\lambda^k}{k!}$ all over λ. This gives us the *chi-squared* statistic:

$$\chi^2 = \sum_{k=0}^{\infty} \frac{\left(o_k - e^{-\lambda}\frac{\lambda^k}{k!}\right)^2}{e^{-\lambda}\frac{\lambda^k}{k!}}.$$

In this case, $\lambda = 1$ and thus this simplifies to

$$\chi^2 = \sum_{k=0}^{\infty} \frac{\left(o_k - \frac{1}{ek!}\right)^2}{\frac{1}{ek!}} = 1 + \sum_{k=0}^{\infty}\left(o_k\left(ek!o_k - 2\right)\right)$$

where in our cases, $o_k = 0$ for $k > 3$. In our three test runs, we get the following three statistics:

Run	**χ^2 statistic**
A	0.179
B	0.189
C	0.104

A 0.179
B 0.189
C 0.104

Comparing these with a χ^2 distribution with one degree of freedom, we have

χ^2 statistic	0.004	0.02	0.06	0.15	0.46	1.07	1.64	2.71	3.84	6.64	10.83
p-value	95 %	90 %	80 %	70 %	50 %	30 %	20 %	10 %	5 %	1 %	0.1 %

Each of these values have *p*-values that lie between 80 % and 50 % and thus it is not unreasonable to assume that the data comes from a Poisson distribution.