ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 9.3.
Assistances and comments will be acknowledged.

9.3 Mapping Down to 0, ..., $M - 1$

In our previous step, we discussed methods for taking various objects and deterministically creating a 32-bit hash value based on the properties of the object. Hash tables, however, are usually restricted to an array of size M and thus we must map this 32-bit number onto the range 0, ..., $M - 1$.

9.3.1 Requirements

Recall that we have certain requirements for hash functions:

1. The hash values must be approximately uniformly distributed among all possible values, and
2. The probability of two objects having the same hash value should be low (approximately 1 in 2^{32} for 32-bit hash values and 1 in M for hash values on the range 0, ..., $M - 1$.

Assuming that the 32-bit hash function satisfies these properties, it is only necessary for this mapping to preserve those properties. We will look at using the modulus operator and the multiplicative method.

9.3.2 Modulus

The easiest implementation is to use the modulus operator:

```
unsigned int hash_M( unsigned int n, unsigned int M ) {
    return n % M;
}
```

The modulus operator, however, can be expensive and thus we will look at a technique for calculating the modulus in a very special case: when $M = 2^m$. From this point on, we will usually assume that hash tables have a size that is a power of 2.

9.3.2.1 Modulus Base 2

Consider the actions taken on positive integers when considering decimal values shown in Table 1 where the division discards the remainder.

Table 1. Operations on decimal numbers with powers of ten.

Operation	Action	Examples	
Modulo 10^m	Take the last m digits	$53247 \bmod 10^3 = 247$	$68091 \bmod 10^4 = 8091$
Multiplying by 10^m	Add m zeros	$53247 \times 10^3 = 53247000$	$68091 \times 10^4 = 680910000$
Dividing by 10^m	Remove the last m digits	$53247 / 10^3 = 53$	$68091 / 10^4 = 6$

All of these operations are mechanical and can be performed with exceptional efficiency—even an elementary school child could easily perform these operations. On the other hand, performing any of the operations

$$53247 \bmod 79,\ 53247 \times 79, \text{ and } 53247 / 79$$

would all require a significant number of operations. Similarly, if we are dealing with base-2 numbers, operations involving powers of two suddenly become mechanical, as is shown in Table 2.

Table 2. Operations on binary numbers with powers of two.

Operation	Action	Examples	
Modulo 2^m	Take the last m bits	$10111 \bmod 2^3 = 111$	$101100 \bmod 2^4 = 1100$
Multiplying by 2^m	Add m zeros	$10111 \times 2^3 = 10111000$	$101100 \times 2^4 = 1011000000$
Dividing by 2^m	Remove the last m bits	$10111 / 2^3 = 10$	$101100 / 2^4 = 10$

Fortunately, there are operators can be performed easily using the bit-wise operations available in C++. Two operations translate quickly, as is shown in

Table 3. Corresponding bit-wise operations.

Operation	Operator	Description	Bitwise Operation
Multiplying n by 2^m	Left-shift operator `<<`	Move the bits m places to the left	`n << m`
Dividing n by 2^m	Right-shift operator `>>`	Move the bits m places to the right	`n >> m`

Specifically, we can calculate powers of two with relative ease. The code fragment

```
for ( int i = 0; i <= 10; ++i ) {
    std::cout << (1 << i) << std::endl;
}
```

will print 1, 2, 4, 8, 16, 32, 64, 128, 256, and 1024.

For calculating the modulus, we note that we must simply select the last m bits. To do this, we must consider the Boolean *and* operation: $1 \,\&\, x = x$ while $0 \,\&\, x = 0$. Thus, we can use a bit-wise *and* operation to select or zero out certain bits; specifically, we must select the last m bits to calculate a number modulus 2^m. Thus, to calculate 2011 modulo 2^7, we must select the last 7 bits:

```
  00000000000000000000011111011011
& 00000000000000000000000001111111
  00000000000000000000000001011011
```

The last number is $2^7 - 1$ and from the previous example, we see that this can be calculated as `(1 << 7) - 1`. Thus, if we are taking a modulus by a power of 2, it is easiest to calculate

```
unsigned int hash_M( unsigned int n, unsigned int m ) {
    return n & ((1 << m) - 1);
}
```

9.3.2.2 Problems with the Modulus Operator

Using the modulus can be appropriate in many cases; especially if the 32-bit hash function is close to random (all 32-bit values have equal probability of being selected). There may, however, still be patterns. For example, if you were to create a large number of dynamically allocated singly linked list objects (`Single_list<int>`) from Project 1, you will note that the last five bits of the address on ecelinux are always `10000`. Thus, if you were to take the last 7 bits of the address, the only possible hash values would be

`0010000`, `0110000`, `1010000`, and `1110000`.

Out of $2^7 = 128$ possible hash values, this is exceptionally bad. This is similar to using the first four decimal digits of any student with a UW Student ID Number: they are likely to be one of fourty possible values: 201*n* , 202*n* , 203*n* , or 204*n*. Fortunately, there is a relatively easy solution: the multiplicative method.

9.3.3 The Multiplicative Method

The multiplicative method converts a 32-bit hash value *n* by taking the middle *m* bits of *Cn* where *C* is a prime number at least as large as 2^{31} (that is, a prime between $2147483659 > 2^{31}$ and $4294967291 < 2^{32}$). Note, there are approximately $\frac{2^{32}}{\ln\left(2^{32}\right)} - \frac{2^{31}}{\ln\left(2^{31}\right)} \approx 94000000$ on this range (1 in 23 integers). You only need to choose one.

Aside: there are approximately $\frac{n}{\ln(n)}$ prime numbers less than or equal to *n*.

To extract the middle *m* bits, we must discard the first (32 – *m*)/2 bits. For example, if we want to extract the middle 7 bits of a 32-bit number, we must discard the last (32 – 7)/2 = 12 bits. This can be done through integer division or, equivalently, a right-shift of 12 bits. Having done this, we take the last 7 bits of what remains. This can be done by the following routine:

```
unsigned int const HASH_MULTIPLIER = 581869333;  // a large prime number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;

    return ((HASH_MULTIPLIER*n) >> shift) & ((1 << m) – 1);
}
```

9.3.3.1 Example

Suppose we want to calculate the hash of the 32-bit number $n = 42$ by taking the middle 10 bits of Cn where C is the constant used in the example above. In binary, 42 is represented by

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0

First we calculate the shift: `shift = (32 - m)/2` or $(32 - 10)/2 = 11$.

Next, we calculate Cn: `C*n` or 42×581869333:

1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1	0	1	1	1	0	0	1	0

We calculate a left-shift `(C*n) >> shift` or

1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1	0	1	1	1	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	0	0	0	0	1	1

We calculate $2^{10} - 1$: `(1 << m) - 1` or

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1

Finally, we take the bit-wise and of these last two numbers:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1

This produces the value 195.

9.3.4 Calculating Hash Values with Negative Numbers

Some languages, for example, Java, do not have unsigned integers. Consequently, we cannot simply use the modulus operator. The definition of the modulus operator is that number that satisfies the equality

```
n == M*(n/M) + n%M
```

Because the integer division truncates toward zero, *e.g.*, `27/4 == 6` and `-27/4 == -6`, the modulus for positive numbers will be a value between `0` and `M - 1` while the modulus for negative numbers will be a value between `-(M - 1)` and `0`, *e.g.*, `27%4 == 3` and `-27%4 == -3`.

Therefore, it would be recommended to use

```
int hash_M( int n, int M ) {
    int hash_value = n % M;
    return (hash_value >= 0) ? hash_value : hash_value + M;
}
```

There are two reasons for not using `abs( hash_value )`:

1. Not all signed integers have absolute values, specifically, the smallest. The output of

```
#include <iostream>
int main() {
        int m = -2147483647;
        int n = -2147483648;
        std::cout << abs( m ) << std::endl;
        std::cout << abs( n ) << std::endl;

        return 0;
}
```

is

```
2147483647
-2147483648
```

and

2. Adding M results in periodicity that can be exploited elsewhere, as is shown in Figure 1.

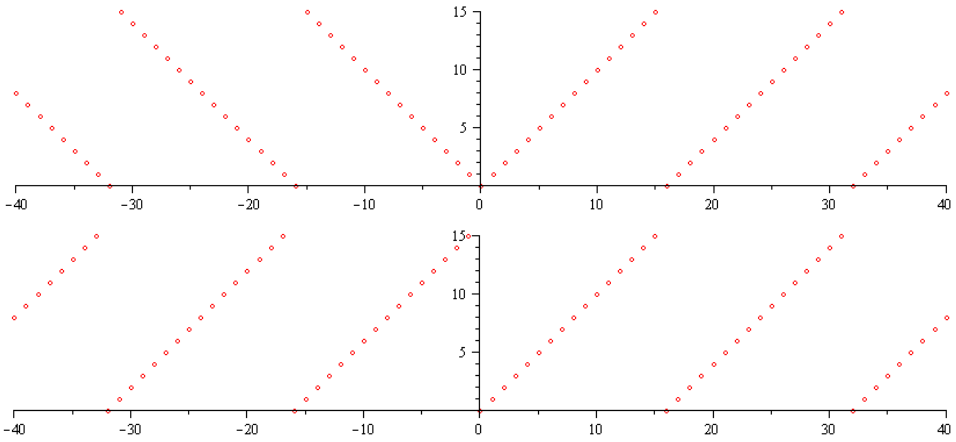

Figure 1. A plot of `abs(n % 16)` and one of `(n >= 0) ? n : n + 16`.