Hash Tables (Cont'd)

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Hash Tables

- Last time we introduced the idea behind Hash Tables, and discussed hash functions and the issue of collisions.
- Today, we'll:
  - Expand on the issue of collisions and see why this is not a problem (or in any case, why we can easily deal with it)
  - Talk about better and more general hash functions

Hash Tables

- Thinking big picture from what we saw last time...
- What could be a good use for these hash tables? (i.e., what type of data or data sets could we store in a hash table?)

Hash Tables

- Thinking big picture from what we saw last time...
- What could be a good use for these hash tables? (i.e., what type of data or data sets could we store in a hash table?)
- Two main types:
  - Storing a *set* (in the strict mathematical sense; an unordered collection of distinct, unrelated elements)
  - An associative container

Hash Tables

- Thinking big picture ...
- For a set, we would compute the hash of the values being stored, and we would then store the values at the position (usually referred to as *bin*) given by their hash.
  - Operations on this hash table:
    - check whether a given value is in the set or not.
    - Iterate over all values in the set (in any arbitrary order; after all, in a set, order plays no role — the actual collection of elements is what defines it).

Hash Tables

- Thinking big picture ...
- Associative containers: you could think of them as an array where the subscript is an arbitrary data type.
- In this case, the idea is that a given subscript or index, or more formally, a given *key* will be associated with a given *value*, and you can access this value by its key.

Hash Tables

- Thinking big picture ...
- For associative containers, then, we would compute the hash of the key, and then, in the bin given by the hash, we store the key along with the value.

Hash Tables

- Thinking big picture ...
- For associative containers, then, we would compute the hash of the key, and then, in the bin given by the hash, we store the key along with the value.
  - Operations on this associative container:
    - Given a key, look up (i.e., retrieve) the associated value (if key is found)
    - Iterate over all pairs (key,value)

Hash Tables

- Thinking big picture ...
- In our example, the values would be the collection of student's data, and the key would be the student ID.
  - The “associative” container in this case associates student IDs with the corresponding student's data.

Hash Tables

- So, we had from last time (maybe) the pending question of how to deal with collisions.
  - That is, how to store things when several different elements produce the same hash?

Hash Tables

- So, we had from last time (maybe) the pending question of how to deal with collisions.
  - That is, how to store things when several different elements produce the same hash?
- The simplest method is known as *chained* hash or chained hashing.
  - It is the most immediate solution that would come to mind: at each bin (each location of the array), store a linked list (or a dynamic array) of elements.

Hash Tables

- We can either make it explicitly an array of linked lists:

```
Array< Single_list<Student> > students_data;
```

- Or we could implicitly make a linked list as part of our hash table:

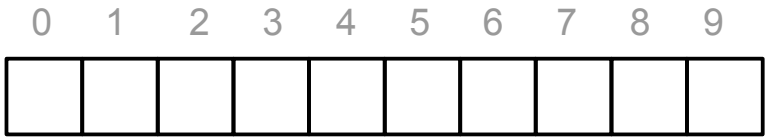
```
Array< Node<Student> *> students_data;
```

Hash Tables

- This goes quite well with our analogy of the mailboxes — each location in the array is not one element, but one “box” (one *container* or, well, one *bin*) where several elements go.

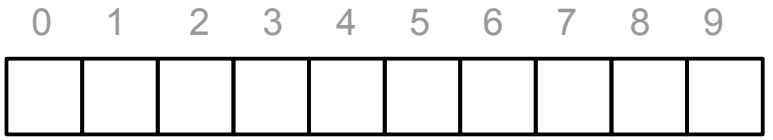
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Hash table initially empty:



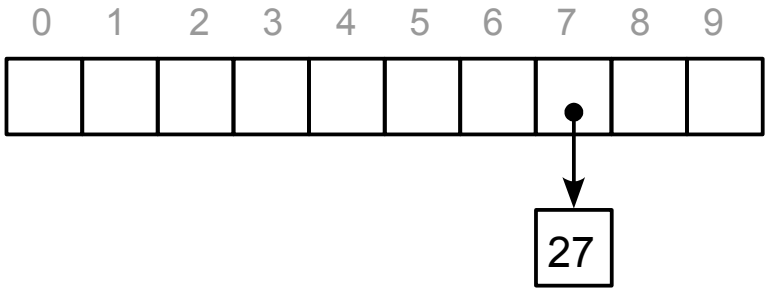
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Insert 27:



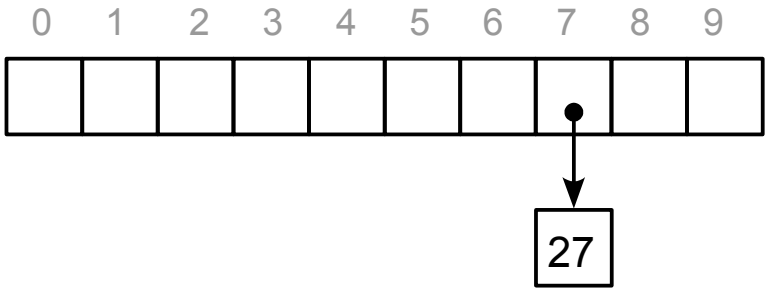
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Insert 27:



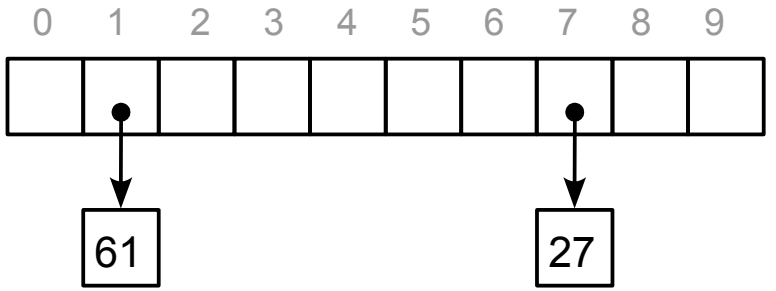
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Now insert 61:



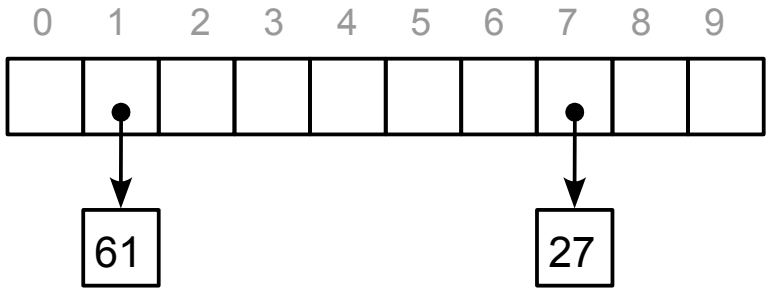
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Now insert 61:



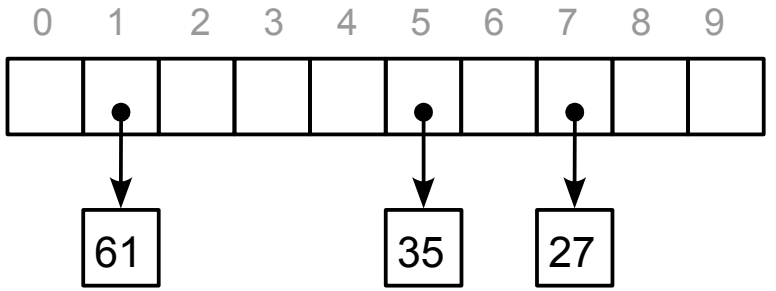
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Now insert 35:



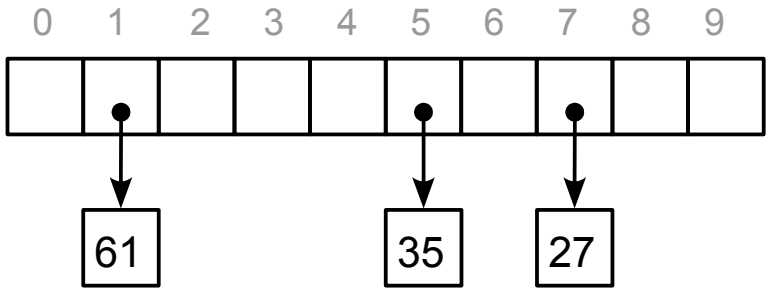
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Now insert 35:



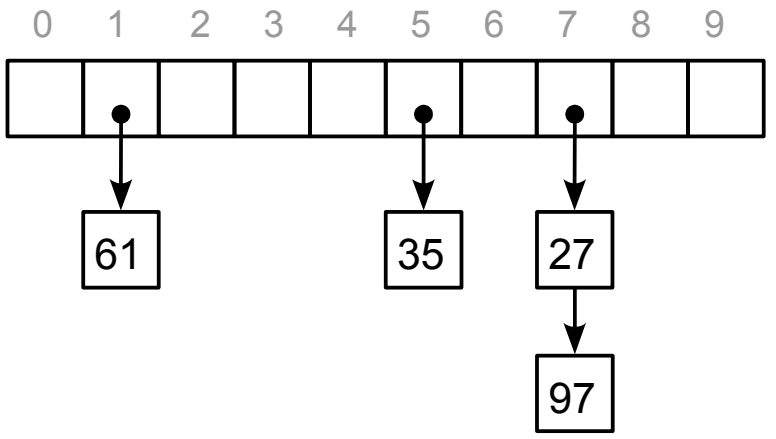
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Now insert 97:



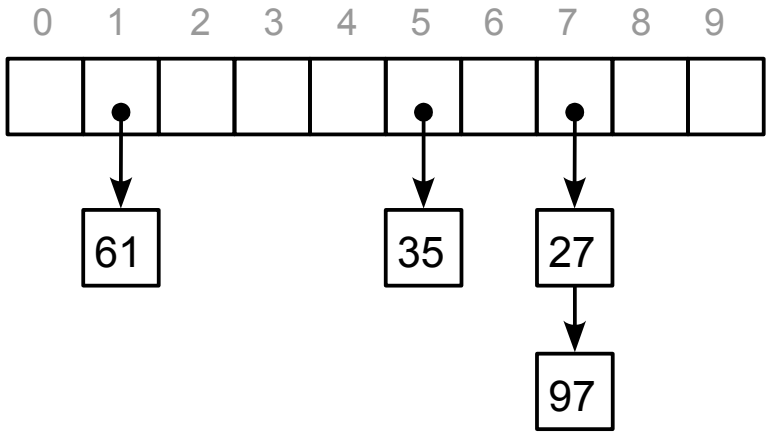
Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - Now insert 97:



Hash Tables

- Graphically, it would look something like this (example storing values from 0 to 99 with a hash function being **h(*n*) = *n* % 10** — i.e., the right-most digit):
  - How do we look up values? (say, 35, 24, 97, 17?)



Hash Tables

- However, this chained hashing technique is not the preferred approach, given the extra "housekeeping" required, and the storage overhead.

Hash Tables

- However, this chained hashing technique is not the preferred approach, given the extra "housekeeping" required, and the storage overhead.

- Before investigating alternative techniques, we'll address the issue of getting better hash functions.

Hash Tables

- We mentioned that this hash function can not be injective (1-to-1), since it maps a large range to a smaller one.
  - Thus, there are not enough distinct values for the result as there are distinct values of the argument.

Hash Tables

- However, we want this function to behave as close as possible to an injective function.
- That is, we want that, probabilistically, the function tends to map different data to different hash values (with high probability).
  - That is, given two different values, we want to have a probability as high as possible to map those two different values to two different hash values.

Hash Tables

- However, we want this function to behave as close as possible to an injective function.
- That is, we want that, probabilistically, the function tends to map different data to different hash values (with high probability).
  - That is, given two different values, we want to have a probability as high as possible to map those two different values to two different hash values.
  - This leads to a low (as low as possible, anyway) probability of collisions.

Hash Tables

- BTW ... If we're handling collisions that nicely and that easily, why would we want to minimize the rate of collisions?
- If anything, it would be worth guaranteeing no collisions, since we would get rid of the need for a mechanism to deal with collisions.
- But since we need to put that mechanism (since collisions *can* happen), then one collision or many collisions should be the same ... (yes?)

Hash Tables

- Hopefully you noticed that the answer is NO (*emphatically* no!)
  - The key detail being: lots of collisions means that it takes longer to find the value! (the time it takes to find a value in a bin is proportional to the number of elements in that bin)

Hash Tables

- Another condition for the hash function is that it must be efficiently computable.
  - We not only want it to run in constant time; we also want it to run fast !!

Hash Tables

- Another condition for the hash function is that it must be efficiently computable.
  - We not only want it to run in constant time; we also want it to run fast !!
    - Huh?? Isn't that the same??

Hash Tables

- Another condition for the hash function is that it must be efficiently computable.
  - We not only want it to run in constant time; we also want it to run fast !!
    - Huh?? Isn't that the same??
    - Not really — constant time means asymptotically faster than any other class; but if the proportionality constant is high, we're not in good shape
      - In other words, we want constant time with a proportionality constant as low as possible!

Hash Tables

- The simplest hash function (i.e., one that meets the minimal requirements to a reasonable extent) is what we've used so far, using modulo:

```
int hash (int n)
{
    return n % m;
}
```

Hash Tables

- However, this function only maps to “random” evenly distributed hash values if the input values are random (or at least random-ish)
  - Examples:
    - We already discussed the situation with prices; if we're using modulo 10 to obtain the hash of a price, the value 9 will show up extremely often.
    - With e-mail addresses — if we keep the last few bits of the binary (ASCII) representation of the last character, we're going to obtain the bits corresponding to m (`.com`) and those corresponding to a (`.ca`) very often.

Hash Tables

- So, the idea is that we want to scramble the data (including *all* bits of the data) in a way that we remove any obvious pattern, making the output look random(ish) regardless of whether the input is truly random or truly patternless.

Hash Tables

- So, the idea is that we want to scramble the data (including *all* bits of the data) in a way that we remove any obvious pattern, making the output look random(ish) regardless of whether the input is truly random or truly patternless.
- This property is known as the *avalanche effect*:
  - We want a minor change in the input data to have a major effect on the output (ideally, producing a hash that looks completely unrelated to the one for the similar data)

Hash Tables

- A true avalanche effect is not only difficult to design, but also very hard to achieve without sacrificing efficiency (it would typically require a loop, applying some formula several times over, etc. etc.)
- So, we're typically happy with a moderate avalanche effect, in which at least the hash values are spread out through the range, even for close or related inputs.

Hash Tables

- One of the simple operations that accomplishes this is multiplication by a number (preferably prime) that is large enough to produce overflow for most input values.

Hash Tables

- One of the simple operations that accomplishes this is multiplication by a number (preferably prime) that is large enough to produce overflow for most input values.
- So, the following would be a better hash function for an int value:

```
int hash (int n)
{
    return (n * 581869333) % m;
}
```

Hash Tables

- Several caveats:
  - Signed arithmetic is not particularly friendly to modular arithmetic. So, often enough, we prefer to work with `unsigned int` data types for the hash value and the intermediate calculations.
  - Modular arithmetic (modulo operations) are not as fast as we would like — they involve a division.
    - An analogy: what is 7315 modulo 100 ? (easy — right?)

Hash Tables

- Several caveats:
  - Signed arithmetic is not particularly friendly to modular arithmetic. So, often enough, we prefer to work with `unsigned int` data types for the hash value and the intermediate calculations.
  - Modular arithmetic (modulo operations) are not as fast as we would like — they involve a division.
    - An analogy: what is 7315 modulo 100 ? (easy — right?)
    - What is 7315 modulo 87 ? (in the spirit of fairness, try to do this one with paper and pencil — i.e., *without* a calculator; after all, you didn't need it for the first one!)

Hash Tables

- Now, the operation modulo 100 was particularly easy for us because our brains work with numbers in base 10, and taking the remainder of a division by a power of 10 is trivial! (we don't need to do a division at all — we don't need to do *any arithmetic* at all!)

Hash Tables

- Now, the operation modulo 100 was particularly easy for us because our brains work with numbers in base 10, and taking the remainder of a division by a power of 10 is trivial! (we don't need to do a division at all — we don't need to do *any arithmetic* at all!)
- The important detail is that it's not the number 10 (ten) that has anything special — it is the fact that we're dividing by a power of the base!! (in our case, 10)

Hash Tables

- So, if for our brains, multiplying, dividing, and taking remainders by powers of 10 is trivial and efficient, what do we think would be trivial and efficient for a computer to do?

Hash Tables

- So, if for our brains, multiplying, dividing, and taking remainders by powers of 10 is trivial and efficient, what do we think would be trivial and efficient for a computer to do?
- Computers work in base 2, so multiplication, division, and modulo powers of two are particularly easy operations

Hash Tables

- For the same reason that multiplying in decimal times, say, 10^3 means adding three zeros to the right, and dividing by 10^3 means chopping off the three right-most digits, we have that:
- In binary, multiplying times 8 (2^3) means adding three zeros to the right, and dividing by 8 means chopping off the three least-significant bits
  - Of course, this applies in general for 2^n.

Hash Tables

- Question: how do we add three (or *n*) zeros to the right?

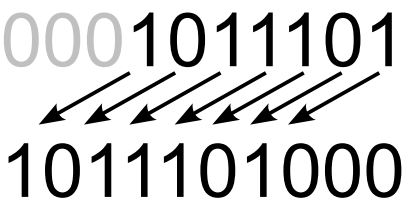
Hash Tables

- Question: how do we add three (or *n*) zeros to the right?
  - If we keep the bits aligned, we notice that this is the same as shifting the bits three (or n) positions to the left, filling with zeros:

000**1011101**

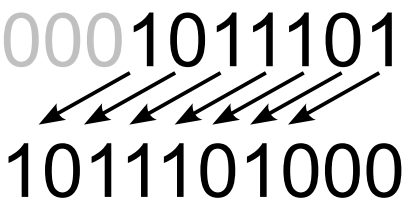
Hash Tables

- Question: how do we add three (or *n*) zeros to the right?
  - If we keep the bits aligned, we notice that this is the same as shifting the bits three (or n) positions to the left, filling with zeros:



Hash Tables

- Question: how do we add three (or *n*) zeros to the right?
  - If we keep the bits aligned, we notice that this is the same as shifting the bits three (or n) positions to the left, filling with zeros:



  - In C++, we do this with the << operator:

```
x = x << 3;   // or x <<= 3;
```

Hash Tables

- Perhaps more interesting is: what is, in binary, the result of a number modulo, say, 2^3 ?
  - For example, 1001101 modulo 2^3

Hash Tables

- Perhaps more interesting is: what is, in binary, the result of a number modulo, say, 2^3 ?
  - For example, 1001101 modulo 2^3
  - Ok, so it is 101 (the three right-most bits) — for the exact same reason that 73215 modulo 10^3 is 215 (the three right-most digits)

Hash Tables

- Follow-up question: How do we extract the three (or in general, the n, for a given n) right-most (least-significant) bits in C++?

Hash Tables

- Follow-up question: How do we extract the three (or in general, the n, for a given n) right-most (least-significant) bits in C++?
  - You recall AND operations, right?

0 AND 0 = 0	1 AND 0 = 0
0 AND 1 = 0	1 AND 1 = 1

Hash Tables

- Follow-up question: How do we extract the three (or in general, the n, for a given n) right-most (least-significant) bits in C++?
  - You recall AND operations, right?

0 AND 0 = 0 1 AND 0 = 0
0 AND 1 = 0 1 AND 1 = 1

Conclusion: 0 AND x is 0; 1 AND x is x
where x is a value 0 or 1 (a single bit)

Hash Tables

- In C++, we have the bitwise and operator, the **&**

Hash Tables

- In C++, we have the bitwise and operator, the **&**

  (and you thought that using that symbol for address-of and for references and pass-by-reference was confusing... HA!! :-))

Hash Tables

- In C++, we have the bitwise and operator, the **&**

  So, if we have a variable x with some value, and a variable mask with a "bit mask" — zeros where we want to "mask out" those bits and ones where we want to let those bits through, then we do:

```
x = x & mask;   // or  x &= mask;
```

  to clear the bits in x where mask has zeros.

Hash Tables

- Example:

```
unsigned int x    = 0xA5B7;
unsigned int mask = 0xFF;  // eight ones

x &= mask;
    // now, x contains 0xB7
```

```
x:          1010 0101 1011 0111
mask:       0000 0000 1111 1111
x & mask:   0000 0000 1011 0111
```

Hash Tables

- So, how do we easily obtain the mask with the right amount of ones in the least-significant bits?
  - Hopefully we remember, from assignment 1 (actually, hopefully you remember this from your Digital Circuits course!) that *n* ones in binary is the value $2^n - 1$ (2^n is 1 followed by n zeros — if we subtract one, we get n ones)

Hash Tables

- And hopefully you remember from a few slides ago that multiplying times 2^n is adding n zeros to the right.
- In particular, 2^n, which is 1×2^n , can be obtained by taking 1 and adding *n* zeros to the right — or equivalently, shift-left n positions!

Hash Tables

- Summarizing: the "ultra efficient" way to obtain *n* ones in C++ is through the expression
  `(1 << n) - 1`
- And of course, the "ultra efficient" way to obtain a value (say, the variable x) modulo 2^n is:

```
x &= (1 << n) - 1
```

Hash Tables

- If we have the size of our hash table in a variable `size` and we know that that size is guaranteed to be a power of 2, then the efficient way of taking a value (say, x) modulo size would simply be:

```
x & (size - 1)  // equivalent to x % size
```

- Warning: the above works ONLY if we know that size is a power of 2 !!!

Hash Tables

- So, you'll understand why in practice, we always want the size of hash tables to be a power of 2 !!
  - As much as in our examples (which were intended for our brains to process, not for a computer!) we always wanted hash tables of size one hundred, or one thousand!

Hash Tables

- Now, how's this for a plot twist:
  - In general, when doing these types of scrambling operations to values, in the resulting values, the bits from around the middle of the result tend to be more “random(ish)” than the least-significant ones.

Hash Tables

- Now, how's this for a plot twist:
  - In general, when doing these types of scrambling operations to values, in the resulting values, the bits from around the middle of the result tend to be more “random(ish)” than the least-significant ones.
  - So, let's take a look at this demo, from Professor Douglas Harder's slides (taken directly for convenience, to avoid switching documents)

The Multiplicative Method

Suppose that the value $m = 10$ ($M = 1024$) and $n = 42$

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```

The Multiplicative Method

$m = 10$
$n = 42$

First calculate the shift

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```

shift = 11

The Multiplicative Method

$m = 10$
$n = 42$

Next, $n = 42$ or 101010_2

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```

shift = 11

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

The Multiplicative Method

$m = 10$
$n = 42$

Calculate Cn

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```

shift = 11

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

The Multiplicative Method

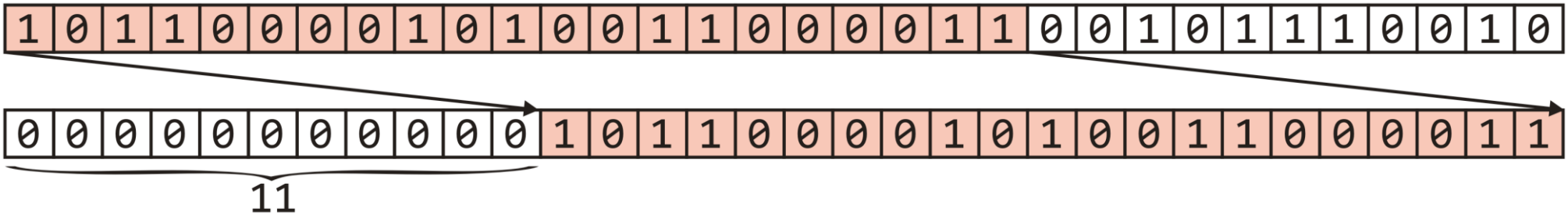
$m = 10$
$n = 42$

Right shift this value 11 bits—equivalent to dividing by 2^{11}

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```

shift = 11



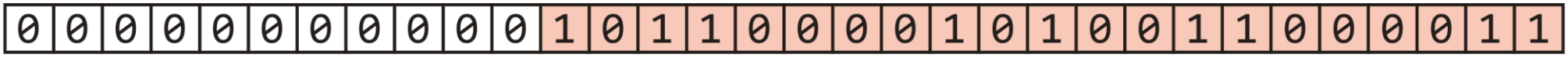
The Multiplicative Method

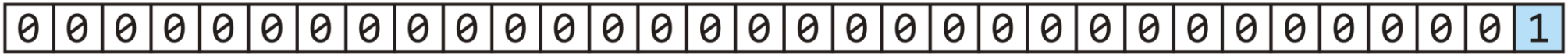
$m = 10$
$n = 42$

Next, start with 1

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```





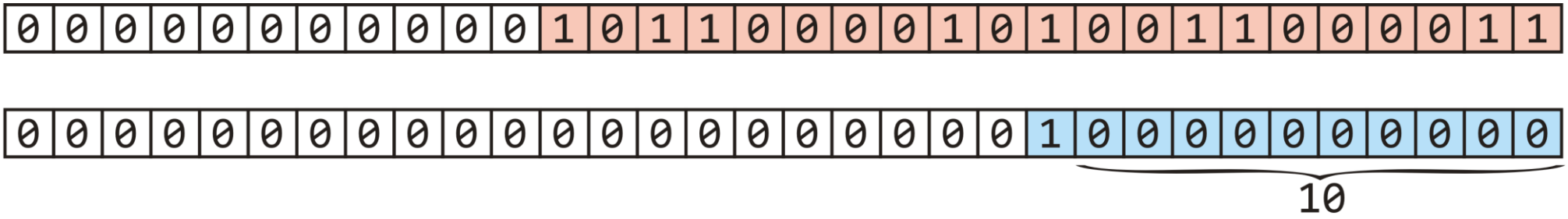
The Multiplicative Method

$m = 10$
$n = 42$

Left shift 1 $m = 10$ bits yielding 2^{10}

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```



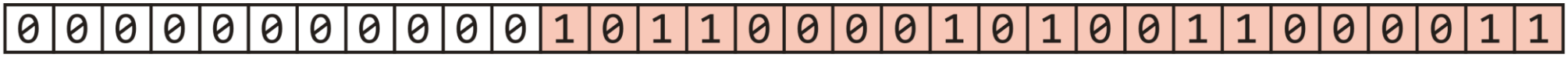
The Multiplicative Method

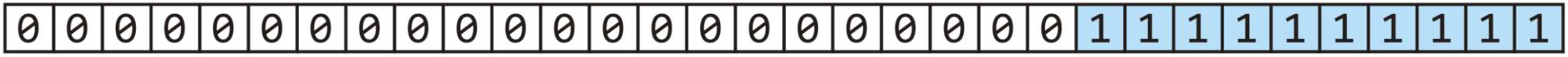
$m = 10$
$n = 42$

Subtracting 1 yields $m = 10$ ones

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```





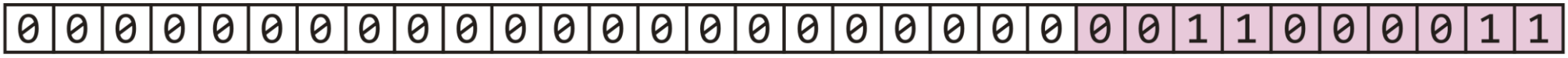
The Multiplicative Method

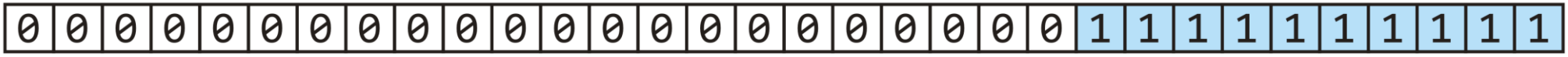
$m = 10$
$n = 42$

Taken the bitwise to clear all but the last 10 bits

```
const unsigned int C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```





Summary

- During this class, we discussed:
  - Typical uses for hash tables (representing sets and associative containers)
  - Chained hashing
  - Good and efficient hash functions
    - In particular, we looked at several tricks to improve efficiency, based on the binary nature of a computer's representation of numbers and arithmetic.