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.
  - Operation on this hash table: check whether a given value is in the set or not.

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 store the key together with the value in the bin given by the hash.

Hash Tables

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

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:

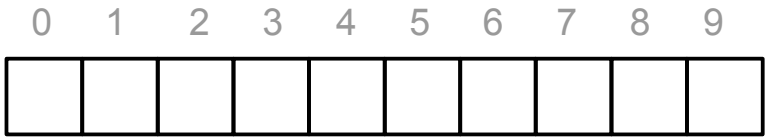
```
Array< Node<Student> *> students_data;
```

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.

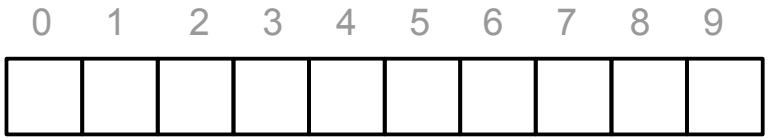
Hash Tables

- 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:



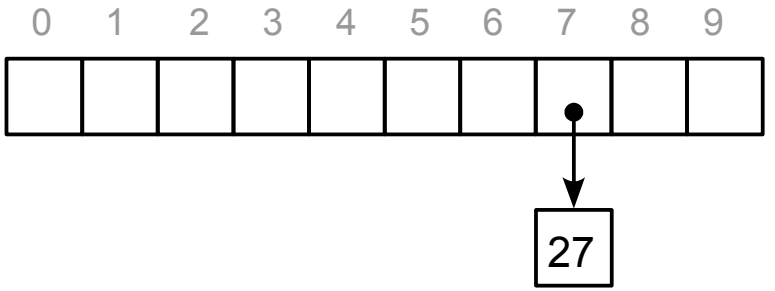
Hash Tables

- 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:



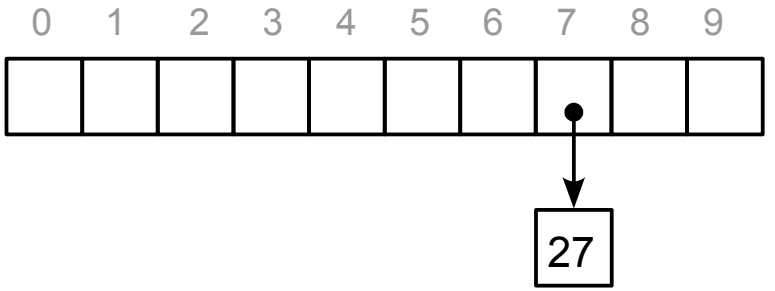
Hash Tables

- 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:



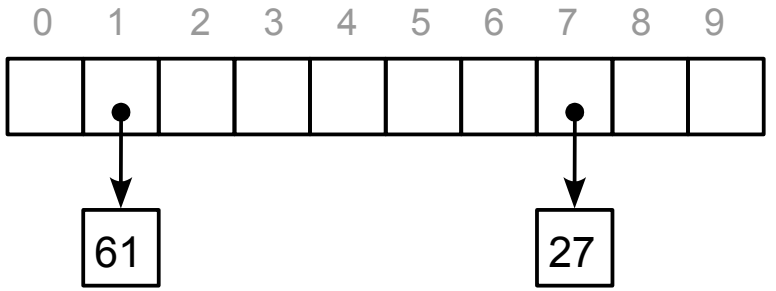
Hash Tables

- 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:



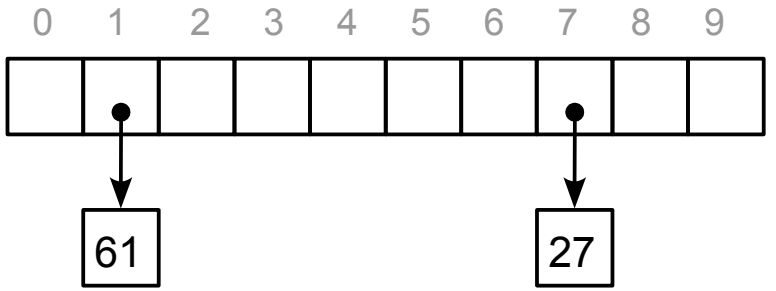
Hash Tables

- 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:



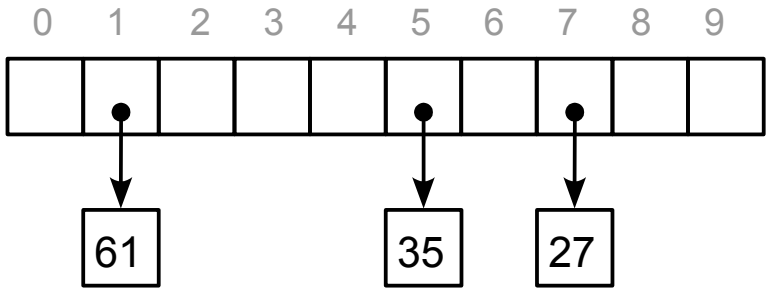
Hash Tables

- 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:



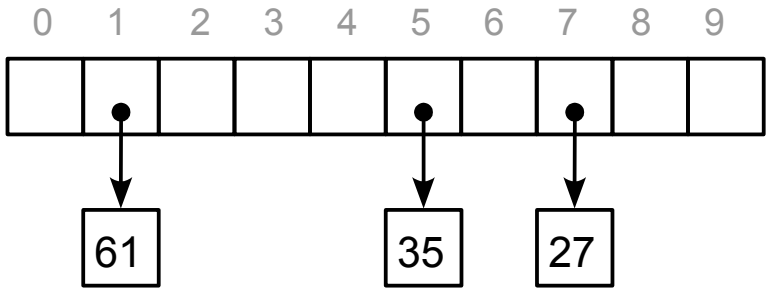
Hash Tables

- 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:



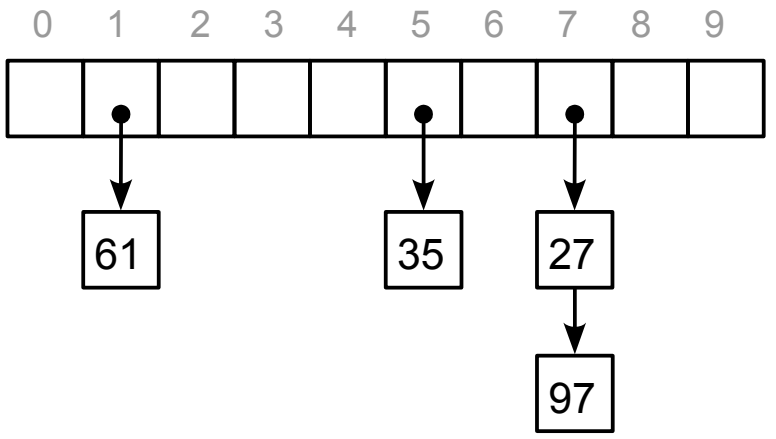
Hash Tables

- 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:



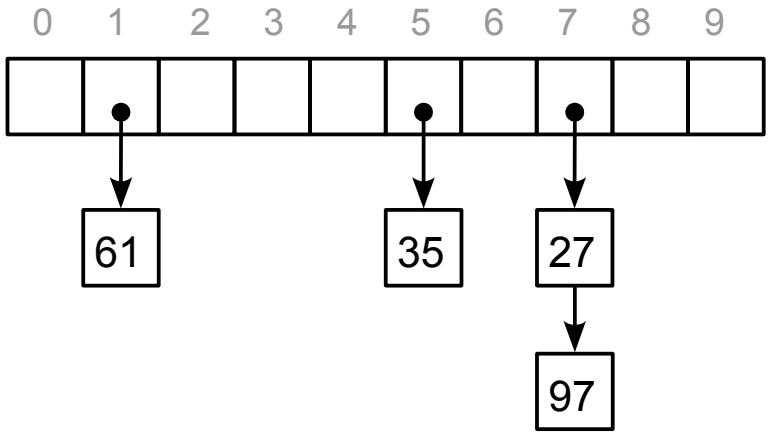
Hash Tables

- 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

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

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 want it to run fast also !!
    - 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 want it to run fast also !!
    - Huh?? Isn't that the same??
    - The answer is no — *why?* (we'll discuss it in class)

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?
- In class, we'll discuss how to efficiently:
  - Compute 2^n given n
  - Compute a value module 2^n (i.e., modulo something that is a power of 2)
  - Multiply or divide by powers of 2

Hash Tables

- For now, I'll leave it for you to think about it
  - Hint: we'll use bitwise operations; in particular, shifting bits (<< to shift left; that is, towards the most-significant bits, MSB; >> to shift right), and bitwise AND (to clear the bits we don't want and keep the bits we need)

Hash Tables

- BTW ... Given that in all of our examples, we've used hash values in the range of 100 or 1000. etc. (because for our brains, taking modulo those is trivial), we'll also discuss in class the detail that in practice, we always want hash tables to have sizes that are powers of 2