Data Structures with C++ : CS189

Lecture 12-2: Hash

Collision Chaining

- When two different input keys want to go in the same bucket, that's a collision
  - They might have had different hash values, but not crashing the table is more important than splitting them up
- So we could just make each bucket a List
  - When inserting, just push_back instead
    - I'll mention preventing dupes in a minute
- With a bad hash function, we all end up in the same list
- We are also allocating a lot of dynamic memory

Collision Probing

- If the spot we want is taken, we could just pick another spot
- We wanted 16, so try +1, +3, +5, +prime numbers or squares or whatever
- If our table is nearly full, this could take forever
- If an entry is later deleted, we need to make sure we can still find it
    - The probe is a bouncing ball, and you make a hole where it bounced for the third time (I need to draw this)
- Not dynamically allocating memory is a nice bonus.

Duplicate Data

- Part of checking for a collision is first making sure the new data isn't already there
- Since different data can get the same hash bucket, we can't tell just from the bucket number
- Functor to the rescue. Classes might not have a == operator so we need to know what equality means
- unordered_map is <K,V,H,E>
    - Key, value, hashor, equalitor

Rehashing

- As a table fills, the chance for collisions increases
  - A full probing table is useless
  - A chaining table doesn't get full, but it slows down as the lists get longer
- We could fix this by taking all the data out, making a bigger table, and putting everything back in
  - Since the last step is % tablesize, they'll move
  - Potentially huge speed hit
- We could also make each bucket a hash table with a different table size
  - A different kind of chaining, but potentially huge memory cost

Passwords

Passwords

Did you Know?
Modern computers can brute force 50 billion hashes a second. Any 8 letter password takes 4 minutes to crack.

- Since hashing turns one thing in to another, and you can't go backwards, it could also be considered encryption
- If we published a hash function, call it md5, then my password doesn't have to be stored on the server. Just the hash
- But if I can access the hashed password table, I can check what "password" is in MD5 and still get it right
  - A Rainbow Table is processing a billion different guesses ahead of time and checking all of them
    - Only password length matters, btw
  - MD5 itself was completely abandoned 10 years ago

Salt

- A way to fight rainbow tables would be to make "password" look different for each person
- When you make a password, the server picks a random huge bit string and tacks it on to the end of your password before it hashes it
  - You effectively have two passwords. The one you picked and the one they picked
- Defeats rainbow table. Even with the salt I have to try every guess on just you, not everyone
  - Works if you want one person though

Pepper

- If you have complete access to my database, you can get every single hash and salt
- But you can't get my computer. Pepper is an extra salt that is computed the moment it is needed
  - So password input and salt are arguments, pepper is server process output
  - password + salt + pepper == hash to validate
- Brute force relies on giant databases and offline processes that run for minutes/ hours/ days. You need the whole computer to beat pepper

End