Hashing
Way Back When...
int nameHash(string first, string last) {
    /* This hashing scheme needs two prime numbers, a large prime and a small
     * prime. These numbers were chosen because their product is less than
     * 2^31 - kLargePrime - 1.
     */
    static const int kLargePrime = 16908799;
    static const int kSmallPrime = 127;

    int hashVal = 0;

    /* Iterate across all the characters in the first name, then the last
     * name, updating the hash at each step.
     */
    for (char ch: first + last) {
        /* Convert the input character to lower case. The numeric values of
         * lower-case letters are always less than 127.
         */
        ch = tolower(ch);
        hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
    }
    return hashVal;
}
A hash function is a function

\[
\text{int \ hashCode(} \text{Type \ arg})\; ;
\]

that is

1. \textit{deterministic} (the same input always produces the same output) and
2. \textit{well-distributed} (The numbers produced are as spread out as possible.)
I’ve got a secret!
A hash function is a function

```
int hashCode(\texttt{Type} \ arg);
```

that is

1. \textit{deterministic} (the same input always produces the same output) and
2. \textit{well-distributed} (The numbers produced are as spread out as possible.)
This is how passwords are typically stored. Look up *salting and hashing* for more details!

And look up *commitment schemes* if you want to see some even cooler things!
Did I hear that correctly?
A hash function is a function

```java
int hashCode(Type arg);
```

that is

1. **deterministic** (the same input always produces the same output) and
2. **well-distributed** (The numbers produced are as spread out as possible.)
This is done in practice!

Look up **SHA-256**, the *Luhn algorithm*, and **CRC32** for some examples!
And, of course, something to do with data structures.
HashMap and HashSet
HashMap and HashSet

- The HashMap and HashSet types work just like Map and Set, except that they do *not* store their keys/elements in sorted order.
- In practice, they are *much* faster than Map and Set, and they should likely be your defaults going forward.
- **Recall:** all the major operations (insertions, deletions, lookups) on Map and Set run in time $O(\log n)$.
- So how on earth are these things faster?
The Juicy Details
An Example: Clothes
For Large Values of $n$
Our Strategy

- Maintain a large number of small collections called *buckets* (think drawers).
- Find a *rule* that lets us tell where each object should go (think knowing which drawer is which.)
- To find something, only look in the bucket assigned to it (think looking for socks.)
Our Strategy

Maintain a large number of small collections called *buckets* (think drawers).

- Find a *rule* that lets us tell where each object should go (think knowing which drawer is which.)

To find something, only look in the bucket assigned to it (think looking for socks.)

Use a hash function!
Buckets

[0] calliope
[1] polyhymnia
[2] euterpe
[3] clio
[4] melpomene
[5] terpsichore
[6] erato
[7] thalia

erato
bool OurHashSet::contains(const string& value) const {
    int bucket = hashCode(value) % buckets.size();
    for (string elem: buckets[bucket]) {
        if (elem == value)
            return true;
    }
    return false;
}
```cpp
bool OurHashSet::contains(const string& value) const {
    int bucket = hashCode(value) % buckets.size();
    for (string elem: buckets[bucket]) {
        if (elem == value)
            return true;
    }
    return false;
}
```
bool OurHashSet::contains(const string& value) const {
    int bucket = hashCode(value) % buckets.size();
    for (string elem: buckets[bucket]) {
        if (elem == value)
            return true;
    }
    return false;
}
bool OurHashSet::contains(const string& value) const {
    int bucket = hashCode(value) % buckets.size();

    for (string elem: buckets[bucket]) {
        if (elem == value) return true;
    }

    return false;
}
Buckets

1. calliope
2. polyhymnia
3. euterpe
4. clio
5. melpomene
6. terpsichore
7. erato
8. thalia
9. urania
void OurHashSet::add(const string& value) {
    int bucket = hashCode(value) % buckets.size();
    buckets[bucket] += value;
}

Buckets:
[0] calliope
[1] polyhymnia
[2] euterpe
[3] clio
[4] melpomene
[5] terpsichore
[6] erato
[7] thalia
[8] urania
void OurHashSet::add(const string& value) {
    int bucket = hashCode(value) % buckets.size();
}

void OurHashSet::add(const string& value) {
    int bucket = hashCode(value) % buckets.size();
}

Buckets
[0] calliope
[1] polyhymnia
[2] euterpe
[3] clio
[4] melpomene
[5] terpsichore

erato

urania

thalia
void OurHashSet::add(const string& value) {
  int bucket = hashCode(value) % buckets.size();
}

(bucket 2)
void OurHashSet::add(const string& value) {
    int bucket = hashCode(value) % buckets.size();
    buckets[bucket] += value;
}
```cpp
void OurHashSet::add(const string& value) {
    int bucket = hashCode(value) % buckets.size();
    buckets[bucket] += value;
}
```

(Bucket 2)
<table>
<thead>
<tr>
<th>Bucket</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>calliope</td>
</tr>
<tr>
<td>[1]</td>
<td>polyhymnia</td>
</tr>
<tr>
<td>[2]</td>
<td>euterpe</td>
</tr>
<tr>
<td>[3]</td>
<td>clio</td>
</tr>
<tr>
<td>[4]</td>
<td>melpomene</td>
</tr>
<tr>
<td>[5]</td>
<td>thalia</td>
</tr>
</tbody>
</table>

```cpp
void OurHashSet::add(const string& value) {
    int bucket = hashCode(value) % buckets.size();
    for (string elem: buckets[bucket]) {
        if (elem == value) return;
    }
    buckets[bucket] += value;
}
```
Time-Out for Announcements!
Assignment 6

- Assignment 5 was due today at the start of class.
  - Using a late day? Turn it in by Monday of next week!
- Assignment 6 (Huffman Encoding) goes out today. It’s due next Friday, March 10.
  - Play around with binary trees in a whole new way!
  - Get some practice with tree recursion!
  - And make your files smaller!
- Anton is holding YEAH hours *today* at 3PM in room 420-041.
- *This assignment must be completed individually.* We’ve broken it down into a bunch of independent, easily-testable, bite-size pieces and included a lot of guidance in the assignment handout.
Need More Practice?

• Many of you have asked about where to go to get extra practice with the material.

• Up on the course website, we have
  • all three versions of the midterm exam (the main exam plus the two alternates), plus
  • section handouts with way more problems on them than anyone could reasonably expect to do in section.

• Feel free to take advantage of these resources, and let us know if you need more!
Change of Grading Basis

- A number of you have asked about the change of grading basis deadline today.
- **Before you make a decision, work out the math on your grade.**
  - Assignments are 40% of your grade. If you’re averaging a ✔+, figure you’ve got roughly a 95%. With a ✔ average, figure you’ve got roughly 85%. With a ✔- average, figure you’ve got roughly a 75%.
  - The midterm is 25% of your grade.
  - The final is the remaining 35%.
- **Unless you earned a low-single-digit score on the midterm and have extremely low assignment grades, it is absolutely still possible to pass this class and do well in it.** A single bad midterm score will not cause you to fail the class, though it may knock you out of contention for a solid A grade.
- **We never curve grades down.** A raw score of 90% is never lower than an A-, a raw score of 80% is never lower than a B-, and a raw score of 70% is never lower than a C-.
- **Compute your raw score before making a switch.** Every quarter I give CR grades to a bunch of folks who earn raw A’s, A-’s, B+’s, and B’s, and I always feel bad when that happens.
Back to CS106B!
So how efficient is our solution?
Efficiency Concerns

• Each hash table operation
  • chooses a bucket and jumps there, then
  • potentially scans everything in the bucket.

• *Claim:* The efficiency of our hash table depends on how well-spread the elements are.
Efficiency Concerns

- Each hash table operation
  - chooses a bucket and jumps there, then
  - potentially scans everything in the bucket.
- **Claim:** The efficiency of our hash table depends on how well-spread the elements are.
Hash Table Performance

• Suppose that we have $n$ elements and $b$ buckets.

• If the elements are distributed as evenly as possible over the buckets as possible, the cost of an operation is $O(1 + n / b)$.
  - The ratio $n / b$ is called the load factor and is sometimes denoted $\alpha$.

• If the elements are all distributed into a single bucket, the cost of an operation is $O(n)$.

• *It’s really important to choose a good hash function!*
Distributing Keys

• We want to choose a hash function that will distribute elements as evenly as possible to try to guarantee a nice, even spread.

• Suppose you want to build a hash function for names.

• **Idea:** Hash each last name to the first letter of that last name.

• How well will this distribute elements?
CS106B Name Distributions

By First Letter of Last Name
Benford's Law

http://en.wikipedia.org/wiki/File:Benford-physical.svg
Good Hash Functions

- A good hash function typically will scramble all of the bits of the input together in a way that appears totally random.
- Hence the name “hash function.”
Implementing a Hash Code

- There’s a lot of *beautiful* mathematical theory behind the design of hash functions.
  - Take CS109, CS161, CS166, or CS255 for details!
  - Or come talk to me after class – this stuff is *super cool!*

- **Claim:** With well-chosen and well-implemented hash functions, you can assume the *expected* cost of an operation in a hash table is $O(1 + \alpha)$.
  - $\alpha$ is the load factor, the ratio of the number of elements to the number of buckets.
What does $O(1 + \alpha)$ mean?
The expected cost of an operation on a hash table is $O(1 + \alpha)$, where $\alpha$ is the ratio of the number of elements ($n$) to the number of buckets ($b$).

**Observation:** If we can keep $\alpha$ small – say, at most two – then this cost is $O(1)$!

**Claim:** The expected cost of an operation on a well-implemented hash table is $O(1)$.

But how do we keep $\alpha$ small?
Hashing and Rehashing

```
0  1  2

Dumbledore  Harry  Lily

Draco  Hermione  McGonnagall

Ron  Hagrid  Snape
```
Hashing and Rehashing

Voldemort

0 1 2

Dumbledore  Harry  Lily

Draco  Hermione  McGonnagall

Ron  Hagrid  Snape
Hashing and Rehashing

Voldemort

0 1 2 3 4 5

McGonnagall Lily Dumbledore Hagrid Harry Snape

Ron Draco Hermione
Hashing and Rehashing

Totally unrelated: look up the term "Voldemort Type."
Hashing and Rehashing

- **Idea:** Track the number of buckets $b$ and the number of total elements $n$.
- When inserting, if $n / b$ exceeds some small constant (say, 2), double the number of buckets and redistribute the elements into the new table.
  - As with Stack, this rehashing happens so infrequently that it’s extremely fast on average.
- This makes $\alpha \leq 2$, so the expected lookup time in a hash table is $O(1)$.
- On average, the lookup time is *independent* of the total number of elements in the table!
To Summarize

- The cost of an insertion, lookup, or deletion in a hash table is, on average, $O(1)$.
  - This assumes you have a good choice of hash function. Unless you have a background in abstract algebra, just follow the template we’ll provide in a second. 😊

- This is why hash tables are one of the single most common data structures used today!
Custom Types in Hash Tables
Custom Types in Hash Tables

- In order to store a custom type in a hash table, you need to be able to
  - get a hash code for it, and
  - compare whether two objects of that type are equal.
- This first task is handled by writing
  ```cpp
  int hashCode(const Type& value);
  ```
- This second task is handled by writing
  ```cpp
  bool operator==(const Type& lhs, const Type& rhs);
  ```
Implementing a Hash Code

- Implementing a good hash function is hard. It’s better to follow a template than to try to be creative.

- Best advice we can offer: write your hash function by combining a bunch of smaller hash functions together.

- One technique:

  ```
  int hashCode(const Type& value) {
      int result = hashCode(value.field1);
      result = 31 * result + hashCode(value.field2);
      result = 31 * result + hashCode(value.field3);
      ...
      result = 31 * result + hashCode(value.fieldN);
      return result & 0x7FFFFFFF;
  }
  ```

- Come talk to me after class for a discussion of why this works!
Implementing Equality

- To implement an equality operator, you typically just return whether all the fields are equal:

```cpp
bool operator==(const Type& lhs, const Type& rhs) {
    return lhs.field1 == rhs.field1 &&
           lhs.field2 == rhs.field2 &&
           ... // Add conditions for all fields
           lhs.fieldN == rhs.fieldN;
}
```
To Summarize

- Hash tables are very fast! You should use them.
- They’re powered by hash functions, which are the Cool Kids at the Function Party.
- Writing your own hash function is hard. Follow a template.
- Don’t forget to implement `operator==`!
Next Time