Announcements

- HW 10 up, check point Tues. after break
- HW9 due Sat.
- Class as usual on Monday
We want to be able to retrieve a name quickly when given a locker number.

(\text{Let } n = \# \text{ of people} \quad \# \text{ of people})

m = \# \text{ of lockers} \quad m \geq n
Dictionaries

A data structure which supports the following:

```c
void insert (keyType &k, dataType &d);
dataType find (keyType &k);
void remove (keyType &k);
```

Note: Everything is based on keys!

Don't know keyType - might not correspond to an int.
Good hash functions:

- Are fast goal: $O(1)$
- Don't have collisions
  these are unavoidable,
  but we want to minimize

\[
\begin{align*}
\text{key space} \quad (\text{size } m) \quad (k,e) & \quad h(k) \quad \mathcal{O}(1) & \quad \text{space: } \mathcal{O}(N) \\
& \quad \mathcal{O}(k) \quad \text{when } k_1 \neq k_2 \quad \text{but } h(k_1) = h(k_2)
\end{align*}
\]
Step 1: Get a number (to avoid collisions)
Char (32-bit) → ASCII

float (64-bit)

Ex:

```
int hashCode (long x) {
    return int(unsigned long(x >> 32)) + int(x);}
```
What about strings?

(Think ASCII.)

69 + 114 + 105 + 110 = 32-bit single representation

fast

Goal: a single int.
But, in some cases, a strategy like this can backfire.

temple and templo and promo

collide under simple XOR

We want to avoid collisions between "similar" strings (or other types).
A Better Idea: Polynomial Hash Codes

Pick $a \neq 1$ and split data into $k$ 32-bit parts: $x = (x_0, x_1, x_2, x_3, \ldots, x_k)$

Goal: Permutations won't collide.

Let $h(x) = x_0 a^{k-1} + x_1 a^{k-2} + \ldots + x_{k-2} a + x_{k-1}$

Ex: Erin with $a = 37$

$69 \ 114 \ 105 \ 110$

$h(\text{"Erin"}) = 69 \cdot 37^3 + 114 \cdot 37^2 + 105 \cdot 37 + 110$

$h(\text{"riE n"}) = 114 \cdot 37^3 + 105 \cdot 37^2 + 69 \cdot 37 + 110$
Side Note: How long does this take? (In terms of $k = \# \text{ of parts}$)

$$h(x) = x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$

\[ \sum_{i=0}^{k} \binom{k}{i} = O(k^2) \text{ multiplications} \]

$k$ multiplication, $k-1$ mult, $k-2$ \cdots 1 \to 0$

Alternate idea: Horners rule: $x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots ))$

$O(k)$ additions + mult.
Polynomial Hashing

This strategy makes it less likely that similar keys will collide.
(Works for floats, strings, etc.)

What about overflow?

$37^{60}$ is huge
(integers stop at $2^{32}$)

truncate!
Cyclic shift hash codes

Alternative to polynomial hashing

Instead of multiplying by \( p \) shift each 32-bit piece (by some \# of bits).

Also works well in practice.

Advantage: fast.

\[
E + r + 1 + n \quad r + i + E + r
\]
Step 2: Compression maps

Now we can assume every key $k$ is an integer.
Need to make it between $0$ and $2^{n-1}$ (not 0 and $2^{32}$).

Goal: Find a "good" map.
"Good": - Fast
- Minimize collisions
Modular compression maps

Take $h(k) = k \mod N$

What does mod mean again?

remainder

\[ 3 \mod 10 = 3 \]
\[ 50 \mod 16 = 0 \]
\[ 14 \mod 10 = 4 \]
Example: \( h(k) = k \mod 11 \)

\[
\begin{align*}
A: & \quad \text{collision} \\
\quad \text{key} & \quad \text{data} \\
& \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \\
& \quad (2, F) \quad (37, D) \quad (16, N) \quad (26, C) \quad (5, H) \\
\end{align*}
\]

Insert: 

\[
\begin{align*}
(12, E) & \quad h(12) = 12 \mod 11 = 1 \\
(21, R) & \quad h(21) = 21 \mod 11 = 10 \\
(37, I) & \quad h(37) = 4 \\
(16, N) & \quad h(16) = 5 \\
(26, C) & \quad h(26) = 4 \\
(5, H) & \\
\end{align*}
\]

(Still need collision strategy...)
Some Comments:

This works best if the size of the table is a prime number.

Why?

Go take number theory & cryptography

Idea: more "prime" numbers are less likely to have things collide
Strategy 2: MAD (multiply, add, divide)

First idea: take \( h(k) = k \mod N \)

Better: \( h(k) = |ak + b| \mod N \)

where \( a \) and \( b \) are:

- not equal
- less than \( N \)
- relatively prime
- no common divisors
  \( 15, 8 \)

(Why? Go take number theory!)
Example: \[ h(k) = \lfloor ak + b \rfloor \mod 11 \]

\[ a = 3 \]
\[ b = 5 \]

A: 0 1 2 3 4 5 6 7 8 9 10

\[ \begin{align*}
(2, R) \\
(4, E) \\
(3, Y) \\
(1, B) \\
(2, E) \\
(5, H) \\
\end{align*} \]

Insert: \[ \begin{align*}
(12, E) \\
(21, R) \\
(32, X) \\
(16, N) \\
(26, C) \\
\end{align*} \]

\[ h(12) = 3 \cdot 12 + 5 \mod 11 = 8 \]
\[ h(21) = 3 \cdot 21 + 5 \mod 11 = 2 \]

\[ \Rightarrow \text{collisions are much less likely} \]
This is a lot of work!

Why bother?

In practice, drastically reduces collisions. $a + b$ can be small in practice.
End Goal: Simple Uniform Hashing Assumption

For any $k \in \text{key space}$,

$$\Pr \left[ h(k) = i \right] = \frac{1}{N}$$

(Essentially, elements are "thrown randomly" into buckets.)

Impossible in practice, still goal we work towards.
Collisions

Can we ever totally avoid collisions?

No
Step 3: Handle collisions (gracefully & quickly)

So how can we handle collisions?

[hint: Do we have any data structures that can store more than 1 element?]

- list
- tree
Example:

A

Running times:

with list:

insert: \( O(1) \)

find: \( O(n) \)

remove: \( O(n) \)

with balanced BST:

all: \( O(\log n) \)
Linear Probing

Instead of lists, if we hash to a full spot, just keep checking next spot (as long as the next spot is not empty).

Find? $O(n)$
Example: \( h(k) = k \mod 11 \)

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>(L, E)</td>
<td>(3, F)</td>
<td>(16, C)</td>
<td>(5, H)</td>
<td>(15, A)</td>
<td>(21, R)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Insert: 
- \( (12, E) \) \( \Rightarrow h(12) = 1 \)
- \( (21, R) \) \( \Rightarrow h(21) = 10 \)
- \( (37, I) \) \( \Rightarrow h(37) = 4 \)
- \( (26, N) \) \( \Rightarrow h(26) = 4 \)
- \( (16, C) \) \( \Rightarrow h(16) = 5 \)
- \( (5, H) \) \( \Rightarrow h(5) = 5 \)
- \( (15, A) \) \( \Rightarrow h(15) = 4 \)

**find** (17, Z) : \( O(n) \)  
**delete** (26)
Issue

How can we remove here?

If you remove create "gap" that
linder probing won't know was
full at time of insertion.

Solution: "dirty bit":
don't actually remove
instead have a bit that
gets flipped when value
is removed.
Running Time for Linear Probing

Insert: $O(n)$

Remove: $O(n)$

Find: $O(n)$

Worst case in practice: fast
**Quadratic Probing**

Linear probing checks \( A[h(k) + 1 \mod N] \) if \( A[h(k) \mod N] \) is full.

To avoid these "primary clusters", try:

\[
A[h(k) + j^2 \mod N] \quad \text{where} \quad j=0, 1, 2, 3, 4, \ldots
\]

- if \( A[h(k)] \) is full
- if \( A[h(k)+1] \) is full
- if \( A[h(k)+2^2] \) is full
  \[
  \ldots
  \]
Example \( h(k) = k \mod 11 \)

<table>
<thead>
<tr>
<th>Index</th>
<th>Inserted Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(2, E)</td>
</tr>
<tr>
<td>1</td>
<td>(4, M)</td>
</tr>
<tr>
<td>2</td>
<td>(5, I)</td>
</tr>
<tr>
<td>3</td>
<td>(26, N)</td>
</tr>
<tr>
<td>4</td>
<td>(16, C)</td>
</tr>
<tr>
<td>5</td>
<td>(15, A)</td>
</tr>
<tr>
<td>6</td>
<td>(5, H)</td>
</tr>
<tr>
<td>7</td>
<td>(15, A)</td>
</tr>
<tr>
<td>8</td>
<td>(4, M)</td>
</tr>
<tr>
<td>9</td>
<td>(2, R)</td>
</tr>
<tr>
<td>10</td>
<td>(26, N)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
  h(12) &= 2 \mod 11 = 2 \\
  h(2) &= 2 \\
  h(37) &= 4 \\
  h(26) &= 4, \text{ full } \Rightarrow h(26) + 1^2 = 5 \\
  h(16) &= 5 \\
  h(5) &= 5, \text{ full } \Rightarrow h(5) + 1^2 = 5 \\
  h(3) &= 4 \\
  h(4) &= 4
\end{align*}
\]
Issues with Quadratic Probing:

- Can still cause "secondary" clustering
- \( N \) really must be prime for this to work
- Even with \( N \) prime, starts to fail when array gets half full, \( n > \frac{12k}{c} \)

(Runtimes are essentially the same)
Rehashing

(twice as big)

pick new

\[ a + b \]

compute \( h'(\text{all entries}) \)