CS180 - Hashing (part 2)

Announcements
- Checkpoint today
- Program due Thursday
- Last HW out today; due next Monday

(Note: Will include topics you haven't seen yet!)

- Review session: Friday, Dec. 16, at 10:30am
- Teacher evals later this week—please come!!
We want to be able to retrieve a name quickly when given a locker number.

\[
\text{Let } n = \# \text{ of people, } m = \# \text{ of lockers} \Rightarrow m \geq n
\]
Dictionaries

A data structure which supports the following:

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

\[ h(k) \] \(O(1)\)

key space (size \(m\)) \(\xrightarrow{(k,e)}\) \(h(k)\)

\[
\begin{align*}
0 & \quad 1 \\
1 & \quad 2 \\
\vdots & \quad \vdots \\
N-2 & \quad \vdots \\
N-1 & \\
\end{align*}
\]

space: \(O(N)\)
Step 1: Get a number (avoid collisions)

Char (32-bit) → ASCII

Float (64-bit) →

String: Erin

\[ 69 + 114 + 105 + 110 = 32\text{-bits} \]

\[ h(\text{Erin}) = h(\text{erin E}) \]
But, in some cases, a strategy like this can backfire:

temp01 and temp10 and pm01
all hash to same int

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: \( x = (x_0, x_1, x_2, x_3, \ldots, x_{k-1}) \)

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

Ex: Erin with \( a = 37 \)

\[ 69 \cdot 37^3 + 114 \cdot 37^2 + 105 \cdot 37 + 110 \]

r En: 2

\[ 114 \cdot 37^3 + 105 \cdot 37^2 + 69 \cdot 37 + 110 \]
Polynomial Hashing

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

What about overflow?

△ truncate
or
take remainders
Cyclic shift hash codes

Alternative to polynomial hashing

Instead of multiplying by a shift each 32-bit piece by some # of bits.

Also works well in practice.

Erin 11 shift by 5  

n 15 shift by 5
Step 2: Compression maps

Now we can assume every key $k$ is an integer.

Need to make it between $0$ and $2^{32}$ (not $0$ and $2^{32}$).

Goal: Find a "good" map.

"Good": - Fast $O(1)$
- Minimize collisions
Modular compression maps

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

What does mod mean again?

\[ 3 \mod 10 = 3 \]
\[ 50 \mod 16 = 0 \quad \text{in C++} \]
\[ 14 \mod 10 = 4 \]
Example: \( h(k) = k \mod 11 \)

\[
\begin{array}{c}
\text{A:} \\
(12, F) \quad (37, I) \quad (46, N) \quad (26, C) \quad (5, H) \\
0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10
\end{array}
\]

Insert: \( (21, R) \)

\[
\begin{align*}
(12, F) & \Rightarrow h(12) = 12 \mod 11 = 1 \\
(21, R) & \Rightarrow h(21) = 21 \mod 11 = 10 \\
(37, I) & \Rightarrow h(37) = 37 \mod 11 = 4 \\
(46, N) & \Rightarrow h(46) = 5 \\
(26, C) & \Rightarrow h(26) = 4 \\
(5, H) & \Rightarrow h(5) = 5
\end{align*}
\]

Find? Compute \( h(v) \) or then find in our "list". Remove? \( h(v) \) & then delete in list.
Some Comments:

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

Why? Go take number theory & cryptography

Collisions are more common the "less prime" a number is.

12 = 2^2 * 3
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

$\gcd(a, b) = 1$

$12 = 2 \cdot 2 \cdot 3$

$20 = 2 \cdot 2 \cdot 5$

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

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

A: \[
\begin{array}{ccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>E</td>
</tr>
<tr>
<td>21</td>
<td>R</td>
</tr>
<tr>
<td>37</td>
<td>I</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
</tr>
<tr>
<td>26</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>H</td>
</tr>
</tbody>
</table>

\( h(k) = \frac{41}{3 \cdot 12 + 5} \mod 11 = 8 \)
\( h(21) = 3 \cdot 21 + 5 \mod 11 = 2 \)
\( h(37) \) - - -

Why bother? In practice, fewer collisions.
End Goal: Simple Uniform Hashing Assumption

For any key in the key space,

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

(Essentially, elements are “thrown randomly” into buckets.)
Collisions

Can we ever totally avoid collisions?

No

key space $\Rightarrow N$

m
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?]

- lists
- AVL trees
- vectors
Ex: list → chaining

Running times:

- **Arrays:**
  - **insert:** $O(1)$
  - **remove:** $O(n)$
  - **find:** $O(n)$
  - **0(n) worst-case**

- **Vectors:**
  - **insert:** $O(1)$ amortized
  - **remove:** $O(n)$
  - **find:** $O(n)$

- **AVL trees:** $O(\log n)$
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).
Example

\[ h(k) = k \mod 11 \]

\( N > n \)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>12</td>
<td>6</td>
<td>(37)</td>
<td>(26)</td>
<td>(16)</td>
<td>(59)</td>
<td>(54)</td>
<td>(2)</td>
<td>(R)</td>
<td></td>
</tr>
</tbody>
</table>

Insert: \( (12, E) \) = 1 primary cluster \( N-1 \)
\( (21, R) \) = 2 \( \mod 11 = 10 \)
\( (37, I) \) = 4
\( (26, N) \) = 4
\( (16, C) \) = 5
\( (5, H) \) = 5
\( (15, A) \) = 4

remove \( (37) \)

find \( (15) \) \( h(15) \leftarrow \text{start at } h(k) \) at keep looking until a blank
**Issue**

How can we remove here?

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

**Solution:** "dirty bit": if $= 1$, then this value has been deleted

If some fraction have dirty bit set, stop & rehash.
Running Time for Linear Probing

Insert: \( O(n) \)

Remove: \( O(n) \)  (since remove calls find, then sets a bit)

Find: \( O(n) \)

rehash: Allocate bigger table, for each entry table, compute \( h(k) \)
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:

$$A[h(k) + j^2 \mod N]$$

where $j = 0, 1, 2, 3, 4, \ldots$

$h(k)$ full, check $h(k) + 1^2$

$h(k) + 1$ full, check $h(k) + 2^2 = h(k) + 4$

$h(k) + 4$ full, check $h(k) + 3^2 = h(k) + 9$
Example

$h(k) = k \mod 11$

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>I</th>
<th>N</th>
<th>C</th>
<th>A</th>
<th>H</th>
<th>R</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>
</tr>
</tbody>
</table>

Insert:

- (12, E) $h(12) = 1$
- (21, R) $h(21) = 10$
- (37, I) $h(37) = 4$
- (26, N) $h(26) = 4$
- (16, C) $h(16) = 5$
- (5, H) $h(5) = 5$, $h(5) + 1^2 = h(5) + 2^2$
- (15, A) $h(15) = 5$
- (4, M) $h(4) = 4$

might actually
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.

(Runtimes are essentially the same.)