Announcements

- Checkpoint Monday
- Program due Friday
- Review in class on last day
**Data Storage - Dictionary**

<table>
<thead>
<tr>
<th>Locker #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Dan</td>
</tr>
<tr>
<td>355</td>
<td>Kevin</td>
</tr>
<tr>
<td>101</td>
<td>Tracy</td>
</tr>
<tr>
<td>53</td>
<td>Nitish</td>
</tr>
<tr>
<td>201</td>
<td>David</td>
</tr>
</tbody>
</table>

We want to be able to retrieve a name quickly when given a locker number.

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

\( n \leq m \)
Good hash functions:

- Are fast: goal: $O(1)$
- Don't have collisions — when $k_1 = k_2$
  these are unavoidable, but we want to minimize

Key space (size $m$) \[ (k, e) \rightarrow h(k) \rightarrow h(0) \rightarrow 0 \rightarrow 2 \rightarrow \ldots \rightarrow 20 \text{ things} \]

$N-2$ $N-1$
Step 1: Get a number (and avoid collisions)

Char (32-bits) → cast (using ASCII)

Float (64-bits)

String: Erin

ASCII: 69 + 114 + 105 + 110 = \text{\textasciitilde}
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: \( x = (x_0, x_1, x_2, x_3, \ldots, x_k) \)

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

Ex: \( \text{Erin} \) with \( a = 37 \)

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

\( \text{rin} \): \( 2 \)

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

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

$k$ additions

$k + (k-1) + (k-2) + \cdots + 1 = \sum_{i=1}^{k} i$

$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$

$1$ addition $+ \frac{k}{2}$ multiplication per term: $O(k)$

Horner's rule: $x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots ))$
Polynomial Hashing

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

What about overflow? (32 bits is our goal)
- wrap around
- truncation
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.
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 $2^{32}$.

Ideas:
- map everything to $0$

Why is that bad?

all collisions
Modular compression maps

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

What does mod mean again?

remainder

\( 3 \mod 10 = 3 \)

\( 56 \mod 10 = 6 \)

(in C++, \( \% \))
Example: \( h(k) = k \mod 11 \)

A:

<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>(12, E)</td>
<td>(37, I)</td>
<td>(16, N)</td>
<td>(5, H)</td>
<td>(26, C)</td>
<td>(5, H)</td>
<td>(21, R)</td>
<td>(21, R)</td>
<td>(26, C)</td>
<td>(5, H)</td>
<td>(12, E)</td>
</tr>
</tbody>
</table>

Insert:

- \((12, E)\)
- \((21, R)\)
- \((37, I)\)
- \((16, N)\)
- \((26, C)\)
- \((5, H)\)

\( h(12) = 12 \mod 11 = 1 \)  \( h(21) = 21 \mod 11 = 10 \)
\( h(37) = 37 \mod 11 = 4 \)  \( h(16) = 16 \mod 11 = 5 \)
\( h(26) = 26 \mod 11 = 5 \)  \( h(5) = 5 \)

Node pair

Pointer to a list:

```
list<\text{char}>* A[11];
```
Some Comments:

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

Why?

Go take number theory & cryptography

This minimizes collisions.

Even base on experimental studies.
Strategy 2: MAD : Multiply, add, & divide
First idea: take \( h(k) = k \mod N \)
Better: \( h(k) = |ak + b| \mod N \)
where \( a \neq b \) are:
- not equal
- less than \( N \)
- relatively prime (no common divisors: \( \gcd(a, b) = 1 \))

(Why? Go take number theory!)
End Goal: Simple Uniform Hashing Assumption

For any \( k \in \text{key space} \),

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

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

- trees
- lists
- vectors
hashing $h(k) = O(1)$

**Ex:**

```
A
0
1 41 → 28 → 54
2
3
4 19
5
6
7
8
9
10
11 90 12 38 25
12 90 12 38 25 10
```

Space: $O(n+N)$

Running times:

- **List:**
  - insert: $O(n)$
  - find: $O(n)$
  - remove: $O(n)$

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

In practice, much better:

- (bad) $O(1)$

- (better) $O(1)$
Next time:

Other strategies to handle collisions.

\[ m = \# \text{ keys} \]

\[ h = \# \text{ of pieces of data} \]

\[ N = \text{size of table} \]