Announcements

- Checkpoint due on Monday
- Lab tomorrow
**Data Storage**

Ex: | Locker # | Name |
--- | --- | --- |
26 | Dan |
355 | Kevin |
101 | Tracy |
53 | Nitesh |
201 | David |

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

\[
\text{(Let } n = \# \text{ of people } + m = \# \text{ of lockers}) \quad m \geq n
\]
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

when $k_1 \neq k_2$ but $h(k_1) = h(k_2)$

key space (size m) \(\xrightarrow{(k,e)}\) h(k) \(\xrightarrow{O(1)}\) \(\xrightarrow{32-64/4} \)

\(\vdots\)

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

char (32-bit) \rightarrow \text{ASCII}

float (64-bit)

Ex:

int hashCode (long x)

\begin{align*}
\text{return } \ & 
\quad \text{int (unsigned long} (x \gg 32) \\
\quad \quad + \text{int} (x))
\end{align*}
What about strings?

(Think ASCII.)

\[ \text{Erin} \]

\[ 69 + 114 + 105 + 110 = 32 \text{-bit single representation} \]

\[ \text{fast} \]

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

temp01 and temp10 and pm0te1 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: $x = (x_0, x_1, x_2, x_3, \ldots, x_{k-1})$

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

Ex: Erin with $a = 37$

$p(“Erin”) = 69 \cdot 37^3 + 114 \cdot 37^2 + 105 \cdot 37 + 110$
Side Note: How long does this take?
(In terms of $k =$ # of parts)

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

- $k-1$ mult.
- $k-2$ mult.
- 0 mult.

+ $k-1$ additions

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

$k-1$ mult. + $k-1$ additions
Polynomial Hashing

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

What about overflow?

trunc, cat, XOR, ...
Cyclic shift hash codes

Alternative to polynomial hashing

Instead of multiplying by \( a^p \), 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 $2^{32}$).

Goal: Find a "good" map.

"Good":
- Fast
- Minimize collisions
Modular compression maps

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

What does mod mean again?

remainder

$3 \mod 10 = 3$

$50 \mod 10 = 0$

$14 \mod 10 = 4$
Example: \( h(k) = k \mod 11 \)

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

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

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

Why?

Go take number theory & cryptography
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

\( \Rightarrow \) no common divisors
\( \gcd(a, b) = 1 \)

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

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

Insert:

\[
\begin{align*}
(12, E) \quad & h(12) = \lfloor 3 \cdot 12 + 5 \rfloor \mod 11 = 8 \\
(21, R) \quad & h(21) = \lfloor 3 \cdot 21 + 5 \rfloor \mod 11 = \_
\end{align*}
\]
This is a lot of work! Why bother?

In practice, drastically reduces collisions.

(These are what actually make hashing slow)
End Goal: Simple Uniform Hashing Assumption

For any key space, \( \Pr \left[ h(k) = i \right] = \frac{1}{N} \)

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

Impossible in practice.
Collisions

Can we ever totally avoid collisions?

No
Step 3: Handle collisions (gracefully and quickly)

So how can we handle collisions?

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

Data structure: find, insert, remove

1. Vector
2. List
3. Balanced BST
Ex: A

Running times:

- Lists: $O(n)$
- Vectors: $O(n)$

BST
AVLs: $O(\log n)$
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 \( h(k) = k \mod 11 \)

<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>( 21, R )</td>
<td>( 37, I )</td>
<td>( 26, N )</td>
<td>( 16, C )</td>
<td>( 5, H )</td>
<td>( 15, A )</td>
<td>( 21, R )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Insert: \( (12, E) \)

- \( h(12) = 1 \)
- \( h(21) = 10 \)
- \( h(37) = 4 \)
- \( h(26) = 4 \)
- \( h(16) = 5 \)
- \( h(5) = 5 \)
- \( h(15) = 4 \)

\( \text{Should be } \text{MATD} \) but spare me arithmetic

\( \text{Remove } (26) \)

\( \text{And } (16) \)

\( \text{And } (26) \) is \( \text{no} \).
Issue

How can we remove here?

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

Solution: "dirty bit":

bit to mark if I've been deleted
Running Time for Linear Probing

Insert:

Remove:

Find:
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 \( j = 0, 1, 2, 3, 4, \ldots \)
Example

$h(k) = k \mod 11$

- Insert: 
  - (12, E)
  - (21, R)
  - (37, I)
  - (26, N)
  - (16, C)
  - (5, H)
  - (15, A)
  - (4, M)
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)