Announcements

- Checkpoint due today
- Program due Friday
- Last HW out today due Monday (not graded)
InBitmapStream

char ch;

BinaryTree < string > myTree;
int input;
InBitmapStream variable;

variable.open("banana.my.zip");
input = variable.read();
if (input == 0)
    myTree.create_root('0');
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.

\[
\begin{align*}
& \text{(Let } n = \# \text{ of people)} \\
& \text{(and } m = \# \text{ of lockers)}
\end{align*}
\]
Good hash functions:

- Are fast \( \text{o}(1) \) when \( k_1 = k_2 \),
- Don’t have collisions, but \( h(k_1) = h(k_2) \) are unavoidable, but we want to minimize.

Key space (size \( m \)) \( \rightarrow (k,e) \rightarrow h(k) \rightarrow \text{buckets} \rightarrow 0, 1, 2, 3 \rightarrow 20 \text{ things}

\( O(1) \not= m \)
Step 1: Turn key into an integer
polyhashing $\Rightarrow$ integer (32-bits)

Step 2: Compression map
Take integer and make it $< N$
mod (or %)
MAD
Collisions

Can we ever totally avoid collisions?

No

m is bigger than n or N
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?]

- Inserted a list in each array space

lists = separate chaining

O(n)  \quad AVL tree  \quad Vector
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

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

\[ i = 0, 1, 2, \ldots \]

<table>
<thead>
<tr>
<th>ME</th>
<th>ANCHOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

**Insert:**

(12, E) \[12 \mod 11 = 1\]
(21, R) \[21 \mod 11 = 10\]
(37, I) \[37 = 4\]
(26, N) \[26 = 4\]
(16, C) \[16 = 5\]
(5, H) \[5 = 5\]
(15, A) \[15 = 4\]
(18, M)

**Find:** (48, -)
Running Time for Linear Probing

Insert: $O(n)$ (not $O(N)$)

- # of elements in array

Remove: remove 5th (don't actually remove)
- "dirty bits"

$O(n)$ if enough of array is dirty
- rehash everything

Find: hash to value & keep checking
- next spot while not found or

$O(n)$ next not empty
Quadratic Probing

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

To avoid clusters, try

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

where \( j = 1, 2, 3, 4, \ldots \)

\[
\begin{align*}
h(k) + 1^2 &= h(k) + 1 \\
h(k) + 2^2 &= h(k) + 4 \\
h(k) + 3^2 &= h(k) + 9 \\
\vdots
\end{align*}
\]
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>L</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>
<td>8</td>
</tr>
</tbody>
</table>

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

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

Can fail entirely!
Secondary hashing

- Try $A[h(k)]$

- If full, try $A[h(k) + f(j)] \mod NJ$
  for $j = 1, 2, 3, \ldots$

  where
  $$f(j) = j \cdot h'(k)$$
  with $h'$ a different hash function

Best if we want to avoid clusters.
Load Factors

Separate chaining actually works as well as most others in practice, although it does use more space.

Most of these methods only work well if \( \frac{n}{N} < 0.5 \).

(Even chaining starts to fail if \( \frac{n}{N} > 0.9 \))

Usually \( 0.25 < \frac{n}{N} < 0.5 \)
Because we need $\frac{n}{N} < .5$, most hash code checks if the array has become more than half full.

If so, it stops and recomputes everything for a larger $N$, usually at least twice as big.

(Still not too bad in an amortized sense -- think vectors.)