Announcements

- HW due today

- Graded HW will come back from me this Wednesday. It will only be emailed to 1 person.

- Next HW - decode - due next Wednesday.
Data Storage - Dictionary:

<table>
<thead>
<tr>
<th>Ex.</th>
<th>Locker #</th>
<th>Name</th>
<th>&lt;data</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>Dan</td>
<td></td>
</tr>
<tr>
<td>355</td>
<td></td>
<td>Kevin</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>Tracy</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td></td>
<td>Nitish</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td></td>
<td>David</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\text{Let } n = \# \text{ of people} + m = \# \text{ of lockers}^8
\]
Good hash functions:

- Are fast goal: \(O(1)\), when \(k_1 \neq k_2\)
- Don’t have collisions, these are unavoidable
  but we want to minimize

\[
\text{key space (size } m) \xrightarrow{(k,e)} h(k) \xrightarrow{O(1)} \text{size of table}
\]

\( h < N \ll m \)
Step 1: Turn key into an integer
  XOR/bit manipulation
  $\rightarrow$ polynomial hashing

Step 2: Compression map

MAD, cyclic permutation, etc.
Collisions

Can we ever totally avoid collisions?

NO:

\[ m \text{ is larger than } N \]

\[ n = 2 \]
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?]

Possibilities:

- Vector
- Linked
- Tree structure

Diagram:

```
  15  ->  56
  ↓    ↓
  116  ↓
  ↓    ↓
  50   56
```

Simple Chaining
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

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

\begin{align*}
0 & \mid 1 & \mid 2 & \mid 3 & \mid 4 & \mid 5 & \mid 6 & \mid 7 & \mid 8 & \mid 9 & \mid 10 \\
(13, E) & \mid (37, I) & \mid (26, N) & \mid (16, C) & \mid (5, H) & \mid (15, A) & \mid (21, R) & \mid (12, E) & \mid (21, R)
\end{align*}

Insert:

\begin{itemize}
\item \((12, E)\):
  \[ h(12) = 12 \mod 11 = 1 \]
\item \((21, R)\):
  \[ h(21) = 21 \mod 11 = 10 \]
\item \((37, I)\):
  \[ h(37) = 4 \]
\item \((26, N)\):
  \[ h(26) = 4 \text{, try } h(26) + 1 \]
\item \((16, C)\):
  \[ h(16) = 5 \]
\item \((5, H)\):
  \[ h(5) = 5 \text{, try } 5 + 1 \]
\item \((15, A)\):
  \[ h(15) = 4 \text{, must walk through array until empty space} \]
\end{itemize}
Issue:

How do we delete? Simply erasing will leave “holes” in arrays and would not work!

Solution: “dirty” bit:

When removing, don’t remove in set dirty bit.

And knows dirty bit means it’s been deleted.
Running Time for Linear Probing

- Insert: $O(n)$ worst case
  - expected: $O(1)$
- Remove: same
- Find: same
Issues with linear probing

- "Clusters" form
  - worse if #’s not "good" in hash function
  - terrible when array nears $\frac{1}{2}$ full

- Removing doesn’t actually reduce # of elements – just sets the "dirty" bit.

(→ frequent re-hashing)
Quadratic Probing

Linear probing checks $A[h(k)+j \text{ mod } N]$, if previous spot is full (for $j = 1, 2, \ldots$)

To avoid clusters, try

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

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

so:

- $h(k)$ first
- if full: $h(k) + 1$
  - if full: $h(k) + 2^2 = h(k) + 4$
  - if full: $h(k) + 3^2 = h(k) + 9$
  - \vdots
Example

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

Insert:

- \((12, E)\) = \(h(12) = 12 \mod 11 = 1\)
- \((21, R)\) = \(h(21) = 10\)
- \((37, I)\) = \(h(37) = 4\)
- \((26, N)\)
- \((16, C)\)
- \((5, H)\)
- \((15, A)\)
- \((4, M)\)
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
- Can fail entirely even if array not full.

(Runtimes are essentially the same)
Secondary Hashing
- Try \( A[h(k)] \)
- If full, try \( A[h(k)] + f(j) \mod NJ \) for \( j = 1, 2, 3, \ldots \)

where \( f(j) = j \cdot l(k) \) with \( l \) a different hash function

using \( l(k) \), a hash function, instead of \( j^2 \)
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 \( \frac{n}{N} > 0.9 \))
Because we need $\frac{n}{N} < 0.5$, 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.)