CS 180 - Hashing

Announcements

- Program due tomorrow
- Next program is posted
- Next HW will be due the last day of class. It will be pass/fail.
- Review in class on Monday (the last day)
**Data Storage**

<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.

\[
\begin{align*}
\text{Let } n &= \# \text{ of people} \\
\text{Let } m &= \# \text{ of lockers}
\end{align*}
\]

\[n \leq m\]
How could we store this?

1) Array
   - Space: $O(m)$
   - Find: $O(1)$
   - Insert: $O(1)$
   [No] m-1 ↦ lock-#
   [Yes] ↦ lock-#

2) AVL tree
   - Space: $O(n)$
   - Find: $O(\log n)$
   - Insert: $O(\log n)$

3) Vector (like Array)
   → Same

4) List
   - Space: $O(n)$
   - Find: $O(n)$
   - Insert: $O(1)$
   [25 Erin] → [11 Dan] → ...
Other examples

- Course # and Schedule info
- Flight # and arrival info
- URL and html page
- Color and BMP

Not always easy to figure out how to store and look up.
Dictionaries

A data structure which supports the following:

- void insert (keyType &k, dataType &d)
- void find (keyType &k)
- void remove (keyType &k)

2 data types:

- keys
- data, or element

Note: Everything is based on keys!

Many possible implementations:

- trees
- lists
- vectors...
Data Structures

First thing to note: An array is a dictionary.

- key: array index
- data: elements

Other alternatives:

goal: $O(n)$ space  
$O(1)$ insert, find, remove

(see prev. slide)
Hashing

Assuming \( m > n \), an array is not very space efficient.

We would like to use \( O(n) \) space, not \( O(m) \).

But then the key needs to get smaller.

\[ m \text{ keys} \rightarrow [] \]

\[ n \]
**Definition:** A hash function \( h \) maps each key in our dictionary to an integer in the range \([0, N-1]\).

(N should be much smaller than \( m = \# \text{ of keys} \).) Usually \( n < N < 2n \).

Then given \((k, e)\), we store \((k, e)\) in array spot \( A[h(k)] \).
Good hash functions:

- Are fast
- Don't have collisions

These are unavoidable.

*goal: \( O(1) \)*

when \( k_1 \neq k_2 \)

but \( h(k_1) = h(k_2) \)

But we want to minimize

\[ O(1) \rightarrow \begin{array}{c}
0 \\
1 \\
2 \\
3 \\
\vdots \\
N-2 \\
N-1
\end{array} \]

20 things

Key space
(size \( m \))
So we have a few steps.

1. Take k and make it a number.
   (Remember, keys can be anything!)
   Ex: char, int, or short (all 32-bits)

   ↓
   ↓
   ↓

   ASCII # → int → Convert (without truncating)
Ex: long or float - 64 bits
(K needs to be 32 bits)

< could give lots of collisions

\[
\begin{align*}
32 \text{ bits} & \quad 32 \text{ bits} \\
\downarrow & \quad \downarrow \\
x & + \quad y \\
\text{or XOR}
\end{align*}
\]
int hashCode (long x) {
    return int(unsigned long(x >> 32))
        + int(x) * (1 << sig) + keeps least significant bits from unsigned long x
}
What about strings?
(Think ASCII.)

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

tempo1 and tempo10 and pm0te1
all hash to same int

We want to avoid collisions between "similar" strings (or other types).