Searching

- Dictionaries
- Hash Tables
- Hash Functions
Dictionaries

- \( n \) items, key length \( k \)

<table>
<thead>
<tr>
<th>Structure</th>
<th>Find</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>List (unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array (unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary tree (unbalanced)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary tree (balanced)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hash Tables
applications:

1. dictionaries
2. object method tables
3. string matching
4. set operations: $\cup, \cap, -$ 

first methods:

1. direct-address tables: small key range. eg, bit vectors.
2. chaining: deletion?
$n$ items in $m$ buckets

time complexity of search =
$n$ items in $m$ buckets

time complexity of search = number of items per bucket

assume nice hash: $P(h(i) = x) = 1/m$
$n$ items in $m$ buckets

time complexity of search = number of items per bucket

assume nice hash: $P(h(i) = x) = 1/m$

let $X_i$ be 1 iff $h(i) = x$, 0 otherwise

\[
E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}
\]

let $\alpha = \frac{n}{m}$ ‘load factor’

expected number of items per bucket is $\alpha$

expected time is $\Theta(1 + \alpha)$
More Collisions

probability that \( k \) of \( n \) elements land in same of \( m \) bins:

let \( \alpha = \frac{n}{m} \) ‘load factor’

\[
\binom{n}{k} \left( \frac{1}{m} \right)^k \left( 1 - \frac{1}{m} \right)^{n-k} \approx \frac{\alpha^k}{e^{\alpha k}}
\]

<table>
<thead>
<tr>
<th>( k )</th>
<th>probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.37</td>
</tr>
<tr>
<td>1</td>
<td>0.37</td>
</tr>
<tr>
<td>2</td>
<td>0.18</td>
</tr>
<tr>
<td>3</td>
<td>0.06</td>
</tr>
<tr>
<td>4</td>
<td>0.015</td>
</tr>
<tr>
<td>5</td>
<td>0.003</td>
</tr>
<tr>
<td>&gt; 5</td>
<td>0.002 total</td>
</tr>
</tbody>
</table>

if \( n = m, \approx \frac{1}{e\cdot k!} \):
Open Addressing

1. **linear probing:** \( h(k, i) = (h_1(k) + i) \mod m \) for increasing \( i \)
   - the runs

2. **double hashing:** \( h(k, i) = (h_1(k) + ih_2(k)) \mod m \) for increasing \( i \)
   - requires: \( h_2 \neq 0, h_2(k) \) and \( m \) relatively prime
   - eg, \( m \) prime and \( h_2(k) < m \)
   - or, \( m = 2^x \) and \( h_2(k) \) odd

3. **cuckoo hashing:** lookups \( O(1) \), insertions amortized expected \( O(1) \)
   - moral: low load factor

deletion?
Searching

Hash Tables

- Hash Tables
- Time Complexity
- More Collisions
- Open Addressing

Break

Hash Functions

asst 2
asst 3
Hash Functions
\[ h : \text{key} \rightarrow 0..m - 1 \]

1. mediocre is easy, good takes effort
2. want time (at most) linear in key size
3. perfect hashing is possible (and efficient) if keys known
   - linear time to construct, linear space to store
4. minimal perfect hashing is possible!
Hash Functions

\[ h : \text{key} \rightarrow 0..m - 1 \]

1. mediocre is easy, good takes effort
2. want time (at most) linear in key size
3. perfect hashing is possible (and efficient) if keys known
   - linear time to construct, linear space to store
4. minimal perfect hashing is possible!

bad news:
- if \(|\text{keys}| \geq m\), there must be collisions
- if \(|\text{keys}| \geq n \cdot m\), then \(\exists\) set of \(n\) that map to same bin
Desiderata:

- make collisions unlikely
  - spread keys across all hashes
  - for each key, each hash equally likely
- similar keys get different hashes
  - all bits of key affect the hash
  - every bit of key affects every bit of hash
- no input always gives worst-case behavior
- fast to compute
- low memory requirement
- easy to implement
1. $hash \leftarrow 0$
2. for each byte of key
3. $hash \leftarrow (hash \times multiplier) + byte$
5. return $hash \mod m$

want $multiplier$ to smear bits, not shift them (to avoid interaction with table size)

$multiplier = 31$ or $127$
assume we have an array of 256 random integers

1. $hash ← 0$
2. for each byte of key
3. rotate the bits in $hash$ by 1
4. $hash ← hash \ xor \ array[byte]$
5. return $hash \ mod \ m$

each byte affects all bits
rotate makes order matter

**universal** class of hash functions: for randomly chosen keys, randomly chosen function from class has $P(\text{collision}) = \frac{1}{m}$

good on average case (over inputs) ≠ good average case on any input
EOLQs

- What’s still confusing?
- What question didn’t you get to ask today?
- What would you like to hear more about?

Please write down your most pressing question about algorithms and put it in the box on your way out.

Thanks!