Announcements

- Submit Prelim 2 conflicts by tomorrow night
- A7 Due FRIDAY
- A8 will be released on Thursday

Hash Functions

- Requirements:
  1) deterministic
  2) return a number in \([0..n]\)

- Properties of a good hash:
  1) fast
  2) collision-resistant
  3) evenly distributed
  4) hard to invert

Example: SHA-256

Example: hashCode()
Application: Error Detection

- Hash functions are used for error detection
- E.g., hash of uploaded file should be the same as hash of original file (if different, file was corrupted)

Application: Integrity

- Hash functions are used to "sign" messages
- Provides integrity guarantees in presence of an active adversary
- Principals share some secret sk
- Send \((m, h(m, sk))\)

Application: Password Storage

- Hash functions are used to store passwords
- Could store plaintext passwords
- Problem: Password files get stolen
- Could store \((\text{username}, h(\text{password}))\)
- Problem: password reuse
- Instead, store \((\text{username}, s, h(\text{password}, s))\)

Application: Hash Set

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>(\text{add}(\text{val} x))</th>
<th>(\text{lookup}(\text{int} i))</th>
<th>(\text{find}(\text{val} x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>LinkedList</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>TreeSet</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>HashSet</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

Expected time

Worst-case: \(O(n)\)

Hash Tables

Idea: finding an element in an array takes constant time when you know which index it is stored in

\[\text{add}(\"CA\")\]

So what goes wrong?
Can we have perfect hash functions?
- Perfect hash functions map each value to a different index in the hash table
- Impossible in practice
  - don’t know size of the array
  - Number of possible values far far exceeds the array size
  - no point in a perfect hash function if it takes too much time to compute

Collision Resolution
- Two ways of handling collisions:
  1. Chaining
  2. Open Addressing

Chaining
- add("NY")
- add("CA")
- lookup("CA")

Open Addressing
- probing: Find another available space
- add("CA")

Different probing strategies
- linear probing: search the array in order: 1, i+4, i+2, i+3, …
- quadratic probing: search the array in nonlinear sequence: i, i+1^2, i+2^2, i+3^2, …
- clustering: problem where nearby hashes have very similar probe sequence so we get more collisions

Load Factor
- \( \lambda = \frac{\# \text{ of entries}}{\text{length of array}} \)
- What happens when the array becomes too full? i.e. load factor gets a lot bigger than \( \frac{1}{2} \)?
  - no longer expected constant time operations
- waste of memory
- too slow

0 1 2 3 4 5
bucket/chain (linked list)
hashIndex
0 1 2 3 4 5
NY
CA
VA
hashIndex
0 1 2 3 4 5
MA
NY
CA
VA

Resizing

Solution: Dynamic resizing
- double the size.
- reinsert / rehash all elements to new array
- Why not simply copy into first half?

Let's try it

Insert the following elements (in order) into an array of size 6:

<table>
<thead>
<tr>
<th>element</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode</td>
<td>0</td>
<td>9</td>
<td>17</td>
<td>11</td>
<td>19</td>
</tr>
</tbody>
</table>

Insert the following elements (in order) into an array of size 6:

<table>
<thead>
<tr>
<th>element</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode</td>
<td>0</td>
<td>9</td>
<td>17</td>
<td>11</td>
<td>19</td>
</tr>
</tbody>
</table>

What is the final state of the hash table if you use open addressing with quadratic probing (assume no resizing)?

Note: Using linear probing, no resizing

Note: Using quadratic probing, no resizing

Note: Using quadratic probing, resizing if load > \( \frac{1}{2} \)
**Collision Resolution Summary**

<table>
<thead>
<tr>
<th>Chaining</th>
<th>Open Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>□ store entries in separate chains (linked lists)</td>
<td>□ store all entries in table</td>
</tr>
<tr>
<td>□ can have higher load factor/degrades gracefully as load factor increases</td>
<td>□ use linear or quadratic probing to place items</td>
</tr>
<tr>
<td></td>
<td>□ uses less memory</td>
</tr>
<tr>
<td></td>
<td>□ clustering can be a problem — need to be more careful with choice of hash function</td>
</tr>
</tbody>
</table>

**Application: Hash Map**

```java
Map<String, String>
void put(K key, V value);
void update(K key, V value);
V get(K key);
V remove(K key);
```

**Application: Hash Map**

Idea: finding an element in an array takes constant time when you know which index it is stored in.

```
put("California", "CA")
get("California")
```

**HashMap in Java**

- Computes hash using key.hashCode()
- No duplicate keys
- Uses chaining to handle collisions
- Default load factor is .75
- Java 8 attempts to mitigate worst-case performance by switching to a BST-based chaining!

**Hash Maps in the Real World**

- Network switches
- Distributed storage
- Database indexing
- Index lookup (e.g., Dijkstra’s shortest-path algorithm)
- Useful in lots of applications…