HASHING

Spring 2018

CS2110
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
Hash Functions

Requirements:
1) deterministic
2) return a number in $[0..n]$

Which of the following functions $f: \text{Object} \rightarrow \text{int}$ are hash functions:

a) $f(x) = x$

b) $f(x) = x.\text{hashCode}()$

c) $f(x) = &x$

d) $f(x) = 0$
Example: SHA-256

GV_SHA256 Hash Core Logic

- A_i
- A_o
- B_i
- B_o
- C_i
- C_o
- D_i
- D_o
- E_i
- E_o
- F_i
- F_o
- G_i
- G_o
- H_i
- H_o

- 32 bit reg ‘a’
- 32 bit reg ‘b’
- 32 bit reg ‘c’
- 32 bit reg ‘d’
- 32 bit reg ‘e’
- 32 bit reg ‘f’
- 32 bit reg ‘g’
- 32 bit reg ‘h’

- Kt_i
- Wt_i

- +
- Maj
- \( \Sigma_0 \)
- \( \Sigma_1 \)
- Ch

- \( \cdot 2 \)
- \( \cdot 13 \)
- \( \cdot 22 \)
- \( \cdot 6 \)
- \( \cdot 11 \)
- \( \cdot 25 \)
Example: hashCode()

- Method defined in java.lang.Object
- Default implementation: uses memory address of object
  - If you override equals, you must override hashCode!!!
- String overrides hashCode:
  
  \[ s.\text{hashCode}() := s[0] \times 31^{n-1} + s[1] \times 31^{n-2} + \ldots + s[n-1] \]
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 $(m, h(m, sk))$
Hash functions are used to store passwords

Could store plaintext passwords

- Problem: Password files get stolen

Could store (username, h(password))

- Problem: password reuse

Instead, store (username, s, h(password, s))
# Application: Hash Set

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

Expected time
Worst-case: \(O(n)\)
Idea: finding an element in an array takes constant time when you know which index it is stored in.

```plaintext
add(“CA”)  
```

Diagram:
- **CA** → Hash function → mod 6 → 5
- Array: MA, , NY, CA

Binary representation: b
- 0: MA
- 1: 
- 2: 
- 3: NY
- 4: 
- 5: 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

When a collision occurs, how do we search for an empty space?

*linear probing*: search the array in order: \(i, i+1, i+2, i+3\ldots\)

*quadratic probing*: search the array in nonlinear sequence: \(i, i+1^2, i+2^2, i+3^2\ldots\)

*clustering*: problem where nearby hashes have very similar probe sequence so we get more collisions
Load Factor

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

0
waste of memory

best range

1
too slow
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>

```
0 1 2 3 4 5
```

```
a e b c d
```
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>

```
0 1 2 3 4 5
a d e b c
```

Note: Using linear probing, no resizing
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)?
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>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>e</td>
<td>d</td>
<td>b</td>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>

Note: Using quadratic probing, no resizing
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>

Note: Using quadratic probing, resizing if load > ½
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

Map<K,V>{{
    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")

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA</td>
<td></td>
<td></td>
<td>NY</td>
<td></td>
<td>CA</td>
</tr>
</tbody>
</table>
```
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...