Hash tables - review

Supports the basic dynamic dictionary ops: insert, find, remove

Does not need class to be Comparable

Three design decisions: tableSize, hash function, collision resolution

Table size

- a prime of the form \((4k+3)\), keeping load factor constraints in mind

Hash function

- should "randomize" the items
- Java's `hashCode()` method

Collision resolution: chaining

Collision resolution: probing (open addressing)

Hash tables - collision resolution: probing

1. Chaining (closed addressing)
2. Probing (open addressing)  
   - Linear probing
   - Quadratic probing
   - Double Hashing
   - Perfect Hashing
   - Cuckoo Hashing

Avoids the use of dynamic memory

In case of collision, try alternative locations until an empty cell is found

Probe sequence: \(h_0(x), h_1(x), h_2(x), \ldots\), with \(h_i(x) = [\text{hash}(x) + f(i)] \mod \text{tableSize}\)

The function \(f(i)\) is different for the different probing methods
Hash tables - collision resolution: **probing**

1. Chaining (closed addressing)
2. **Probing** (open addressing)  
   a. Linear probing  
   b. Quadratic probing  
   c. Double Hashing  
   d. Perfect Hashing  
   e. Cuckoo Hashing

In case of collision, try alternative locations until an empty cell is found.  
**Probe sequence:** $h_0(x), h_1(x), h_2(x), \ldots$, with $h_i(x) = [\text{hash}(x) + f(i)] \mod \text{tableSize}$

The function $f(i)$ is **different** for the different probing methods.

---

**Hash tables - collision resolution: linear probing**

<table>
<thead>
<tr>
<th>Empty Table</th>
<th>After 89</th>
<th>After 18</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>49</td>
<td>49</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>58</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
</tr>
</tbody>
</table>

**Figure 5.11** Hash table with linear probing, after each insertion.

**Example:** insert 89, 18, 49, 58, and 69 into a table of size 10, using linear probing.
Hash tables - collision resolution: **linear probing**

\[ f(i) \text{ can be any } \textbf{linear} \text{ function } (a \times i + b) \]

If \( \gcd(a, \text{tableSize}) = 1 \), then linear probing will probe the entire table

**Primary clustering**: blocks of occupied cells start forming even in a relatively empty table

---

**Hash tables - collision resolution: linear probing**

\[ f(i) \text{ can be any } \textbf{linear} \text{ function } (a \times i + b) \]

If \( \gcd(a, \text{tableSize}) = 1 \), then linear probing will probe the entire table

**Primary clustering**: blocks of occupied cells start forming even in a relatively empty table

---

---
Hash tables - collision resolution: linear probing

\[ f(i) \] can be any linear function \((a \cdot i + b)\).

If \( \gcd(a, \text{tableSize}) = 1 \), then linear probing will probe the entire table.

**Primary clustering**: blocks of occupied cells start forming even in a relatively empty table.

---

**Figure 5.12** Number of probes plotted against load factor for linear probing (dashed) and random strategy (S is successful search, U is unsuccessful search, and I is insertion).
Hash tables - clustering

Two causes of clustering:
- multiple keys hash on to the same location (secondary clustering)
- multiple keys hash on to the same cluster (primary clustering)

Secondary clustering caused by hash function; primary, by choice of probe sequence

Number of probes per operation increases with load factor.

Hash tables – linear probing: remove

- Load factor computed including lazy-deleted items
- In inserts, may "reclaim" lazy-deleted cells
Hash tables - collision resolution: probing

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing \( f(i) \) is a quadratic function of \( i \) (e.g., \( f(i) = i^2 \))
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using quadratic probing

<table>
<thead>
<tr>
<th>Empty Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using quadratic probing
Hash tables - collision resolution: quadratic probing

Two causes of clustering:
- multiple keys hash on to the same location (secondary clustering)
- multiple keys hash on to the same cluster (primary clustering)

Which one does quadratic probing solve?
- primary clustering

Choosing tableSize:
- prime: at least half the table gets probed
- prime of the form \(4k+3\) and probe sequence is \(\pm i^2\): entire table gets probed

Remove: lazy delete must be used

Hash tables - collision resolution: probing

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

To get rid of secondary clustering
Use two hash functions: \(\text{hash}_1()\) and \(\text{hash}_2()\)
Probe sequence "step" size is \(\text{hash}_2()\)

- (Unlikely distinct items agree on both \(\text{hash}_1()\) and \(\text{hash}_2()\))

\(\text{hash}_2()\) must never evaluate to zero!

A common (good) choice: \(R - (x \mod R)\), for \(R\) a prime smaller than tableSize

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using double hashing with \(\text{hash}_2(x) = 7 - x \mod 7\)
Hash tables - collision resolution: double hashing

Empty Table

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using double hashing with hash2(x) = 7 - x mod 7
Hash tables - collision resolution: **probing**

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

---

Hash tables - collision resolution: **Cuckoo hashing**

**Goal**: constant-time $O(1)$ find in the worst case

Example application: network routing tables

[remove also takes $O(1)$ time]

**Insert** has worst-case $O(N)$ run-time

Keep two hash tables, and use two different hash functions
### Hash tables – collision resolution: Cuckoo hashing

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

A: \( h_1(A) = 0, \ h_2(A) = 2 \)

B: \( h_1(B) = 0, \ h_2(B) = 0 \)

---

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

A: \( h_1(A) = 0, \ h_2(A) = 2 \)

B: \( h_1(B) = 0, \ h_2(B) = 0 \)

C: \( h_1(C) = 1, \ h_2(C) = 4 \)

D: \( h_1(D) = 1, \ h_2(D) = 0 \)
## Hash tables - collision resolution: Cuckoo hashing

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>

A: hash1(A) = 0, hash2(A) = 2
B: hash1(B) = 0, hash2(B) = 0
C: hash1(C) = 1, hash2(C) = 4
D: hash1(D) = 1, hash2(D) = 0
E: hash1(E) = 3, hash2(E) = 2
F: hash1(F) = 3, hash2(F) = 4
### Hash tables – collision resolution: Cuckoo hashing

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>

- A: hash₁(A) = 0, hash₂(A) = 2
- B: hash₁(B) = 0, hash₂(B) = 0
- C: hash₁(C) = 1, hash₂(C) = 4
- D: hash₁(D) = 1, hash₂(D) = 0
- E: hash₁(E) = 3, hash₂(E) = 2
- F: hash₁(F) = 3, hash₂(F) = 4
Hash tables - collision resolution: **Cuckoo hashing**

Insert
- Insert into Table 1, using hash₁
- If cell is already occupied
  - **bump** item into other table (using appropriate hash function)
  - **Repeat**
- **Rehash** after k repetitions

**Each** table should be more than half empty

**Stronger** condition than load factor ≤ ½