Maps & Hashing

Fawzi Emad
Chau-Wen Tseng

Department of Computer Science
University of Maryland, College Park

Overview

- Maps
- Hashing
  - Hash function
  - Hash table
Set Data Structures

Types
- Set
- Map
- Hash Table

Maps

Map (associative array)
- Unordered collection of objects (values)
- One or more keys associated with each object
- Can use key to retrieve object

Example
- A[“key1”] = …
Maps

Methods
- Void Put(Key, Object) // inserts element
- Object Get(Key) // returns element
- Void Remove(Key) // removes element

Implementation approaches
- Two parallel arrays
  - Unsorted
  - Sorted
- Linked list
- Binary search tree
- Hash table

```
<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>k2</td>
</tr>
<tr>
<td>[1]</td>
<td>k1</td>
</tr>
<tr>
<td>[2]</td>
<td>k3</td>
</tr>
<tr>
<td>[3]</td>
<td>Δ</td>
</tr>
<tr>
<td></td>
<td>Δ</td>
</tr>
</tbody>
</table>
```

...
Maps

Complexity (average case)

<table>
<thead>
<tr>
<th></th>
<th>Put</th>
<th>Get</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted array</td>
<td>O(n)</td>
<td>O(log(n))</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linked list</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Binary search tree</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>Hash table</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Hashing

Approach

- Transform key into number (hash value)
- Use hash value to index object in hash table
- Use hash function to convert key to number
Hashing

- **Hash Table**
  - Array indexed using hash values
  - Hash Table A with size N
  - Indices of A range from 0 to N-1
  - Store in A[ hashValue % N]

  ![Hash Table Diagram]

- **Hash Function**
  - **Goal**
    - Scatter values uniformly across range
    - Hash(<everything>) = 0
      - Satisfies definition of hash function
      - But not very useful
  - **Multiplicative congruency method**
    - Produces good hash values
    - Hash value = (a × int(key)) % N
    - Where
      - N is table size
      - a, N are large primes

  ![Hash Function Table]

<table>
<thead>
<tr>
<th>Location</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Hash Function

**Example**

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```

**Perfect hash function**

- Unique values for each key

```
0  kiwi
1  banana
2  watermelon
3  apple
4  mango
5  grapes
6  strawberry
```

**Suppose now**

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
hashCode("orange") = 3
```

**Collision**

- Same hash value for multiple keys

```
0  kiwi
1  banana
2  watermelon
3  apple
4  mango
5  grapes
6  strawberry
```
Types of Hash Tables

- **Open addressing**
  - Store objects in each table entry

- **Chaining (bucket hashing)**
  - Store lists of objects in each table entry

### Open Addressing Hashing

**Approach**
- Hash table contains objects
- Probe ⇒ examine table entry
- Collision
  - Move K entries past current location
  - Wrap around table if necessary
- Find location for X
  1. Examine entry at $A[\text{key}(X)]$
  2. If entry = $X$, found
  3. If entry = empty, $X$ not in hash table
  4. Else increment location by $K$, repeat
Open Addressing Hashing

**Approach**
- Linear probing
  - $K = 1$
  - May form clusters of contiguous entries
- Deletions
  - Find location for $X$
  - If $X$ inside cluster, leave non-empty marker
- Insertion
  - Find location for $X$
  - Insert if $X$ not in hash table
  - Can insert $X$ at first non-empty marker

Open Addressing Example

**Hash codes**
- $H(A) = 6$  \( \Lambda \)
- $H(B) = 7$  \( \Lambda \)
- $H(C) = 6$  \( \Lambda \)
- $H(D) = 7$  \( \Lambda \)

**Hash table**
- Size = 8 elements
- \( \Lambda \) = empty entry
- * = non-empty marker

**Linear probing**
- Collision $\Rightarrow$ move 1 entry past current location
Open Addressing Example

Operations
- Insert A, Insert B, Insert C, Insert D

Find A, Find B, Find C, Find D
Open Addressing Example

Operations
- Delete A, Delete C, Find D, Insert C

```
D 1  D 1  D 1  D 1
A 2  A 2  A 2  A 2
A 3  A 3  A 3  A 3
A 4  A 4  A 4  A 4
A 5  A 5  A 5  A 5
* 6  * 6  * 6  C 6
B 7  B 7  B 7  B 7
C 8  * 8  * 8  * 8
```

Efficiency of Open Hashing

- Load factor = entries / table size
- Hashing is efficient for load factor < 90%

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>Number of Comparisons</th>
<th>Approximate Behavior</th>
<th>(Table Size N = 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1.06</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>1.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>1.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td>1.33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>1.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td>1.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>2.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>3.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td>5.50</td>
<td>O(log N)</td>
<td></td>
</tr>
<tr>
<td>0.95</td>
<td>10.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.98</td>
<td>26.5</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>0.99</td>
<td>50.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chaining (Bucket Hashing)

Approach
- Hash table contains lists of objects
- Find location for X
  - Find hash code key for X
  - Examine list at table entry A[key]
- Collision
  - Multiple entries in list for entry

Chaining Example

Hash codes
- \( H(A) = 6 \) \( H(C) = 6 \)
- \( H(B) = 7 \) \( H(D) = 7 \)

Hash table
- Size = 8 elements
- \( \Lambda \) = empty entry

\[
\begin{array}{cccccccc}
1 & \Lambda & 2 & \Lambda & 3 & \Lambda & 4 & \Lambda \\
\Lambda & \Lambda & \Lambda & \Lambda & 5 & \Lambda & 6 & \Lambda \\
\Lambda & \Lambda & \Lambda & \Lambda & 7 & \Lambda & 8 & \Lambda \\
\end{array}
\]
Chaining Example

Operations

- **Insert A**, **Insert B**, **Insert C**

1. Λ
2. Λ
3. Λ
4. Λ
5. Λ
6. A
7. Λ
8. Λ

Find B, Find A

1. Λ
2. Λ
3. Λ
4. Λ
5. Λ
6. C
7. B
8. Λ
Efficiency of Chaining

- Load factor = entries / table size
- Average case
  - Evenly scattered entries
  - Operations = O( load factor )
- Worse case
  - Entries mostly have same hash value
  - Operations = O( entries )

Hashing in Java

- Collections
  - hashMap & hashSet implement hashing
- Objects
  - Built-in support for hashing
    - boolean equals(object o)
    - int hashCode()
  - Can override with own definitions
  - Must be careful to support Java contract
Java Contract

hashCode()
- Must return same value for object in each execution, provided no information used in equals comparisons on the object is modified

equals()
- if a.equals(b), then a.hashCode() must be the same as b.hashCode()
- if a.hashCode() != b.hashCode(), then !a.equals(b)

a.hashCode() == b.hashCode()
- Does not imply a.equals(b)
- Though Java libraries will be more efficient if it is true