Hashing

Department of Computer Science
University of Maryland, College Park

Overview

- Hashing
  - Scattering Hash Values
  - Hash Function
- Hash Tables
  - Open Addressing
  - Chaining
Hashing

**Approach**
- Use hash function to convert key into number (hash value) used as index in hash table

<table>
<thead>
<tr>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_3$</th>
<th>$v_4$</th>
<th>…</th>
<th>$v_n$</th>
</tr>
</thead>
</table>

$\overset{\uparrow}{f(k_1)} \quad \overset{\uparrow}{f(k_2)} \quad \overset{\uparrow}{f(k_3)} \quad \overset{\uparrow}{f(k_4)} \quad \ldots$

Hash function $f$

| Hash table $h$ |

**Hash Table**
- Array indexed using hash values
- Hash table $A$ with size $N$
- Indices of $A$ range from 0 to $N-1$
- Store in $A[\text{hashValue} \% N]$
Hash Function

- Function for converting key into hash value

- For Java
  - Hash value $\Rightarrow$ 32-bit signed int
  - Default hash function $\Rightarrow$ int hashCode()

- For hash table of size N
  - Must reduce hash value to 0..N – 1

Scattering Hash Values

- Should scatter hash values uniformly across range of possible values
  - Hash( <everything> ) = 0
    - Satisfies definition of hash function
    - But not very useful (all keys at same location)

- Could use Math.abs(key.hashCode( ) % N)
  - Might not distribute values well
  - Particularly if N is a power of 2
Scattering Hash Values

- Multiplicative congruency method
  - Produces good hash values
  - Hash value = Math.abs((a * key.hashCode()) % N)
  - Where
    - N is table size
    - a is large prime number

Beware of % (Modulo Operator)

- The % operator is integer remainder
  \[ x \% y = x - y \times \left( \frac{x}{y} \right) \]
- Result may be negative
  \[ -|y| < x \% y < +|y| \]
- x \% y has same sign as x
  - -3 % 2 = -1
  - -3 % -2 = -1
- Use Math.abs( x \% N )
  - Rather than Math.abs(x) \% N
  - In case N is negative
Art and Magic of hashCode()

- There is no “right” hashCode function
  - Art involved in finding good hashCode function
  - Also for finding hashCode to hashBucket function

From java.util.HashMap

```java
static int hashBucket(Object x, int N) {
    int h = x.hashCode();
    h += ~(h << 9);
    h ^= (h >>> 14);
    h += (h << 4);
    h ^= (h >>> 10);
    h ^= (h >>> 10);
    return Math.abs(h % N);
}
```

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>kiwi</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>banana</td>
</tr>
<tr>
<td>3</td>
<td>watermelon</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>apple</td>
</tr>
<tr>
<td>6</td>
<td>mango</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>grapes</td>
</tr>
<tr>
<td>9</td>
<td>strawberry</td>
</tr>
</tbody>
</table>
Hash Function

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

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>kiwi</td>
<td>banana</td>
<td>watermelon</td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>apple</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mango</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>grapes</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>strawberry</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Types of Hash Tables

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

  ![Open addressing diagram](image)

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

  ![Chaining diagram](image)
Open Addressing Hashing

Approach
- Hash table contains objects
- Probe $\Rightarrow$ 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**
  - $H(A) = 6$  $H(C) = 6$
  - $H(B) = 7$  $H(D) = 7$

- **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

```
1  2  3  4  5  6  7  8
A  Λ  Λ  Λ  Λ  Λ  Λ  Λ
2  3  4  5  6  7  8  Λ
A  B  Λ  Λ  Λ  Λ  Λ  Λ
7  8  1  2  3  4  5  Λ
A  B  Λ  Λ  Λ  Λ  Λ  Λ
6  7  8  1  2  3  4  5
A  B  Λ  Λ  Λ  Λ  Λ  Λ
```

- $\Lambda$ 1
- $\Lambda$ 2
- $\Lambda$ 3
- $\Lambda$ 4
- $\Lambda$ 5
- $\Lambda$ 6
- $\Lambda$ 7
- $\Lambda$ 8
### Open Addressing Example

#### Operations
- Find A, Find B, Find C, Find D

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
```

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

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
```
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>O(log N)</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(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></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 \( A[ \text{key} ] \)
  - Collision
    - Multiple entries in list for entry
Chaining Example

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

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

---

Chaining Example

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th>A</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \Lambda )</td>
<td>1</td>
<td>( \Lambda )</td>
<td>1</td>
<td>( \Lambda )</td>
</tr>
<tr>
<td>2</td>
<td>( \Lambda )</td>
<td>2</td>
<td>( \Lambda )</td>
<td>2</td>
<td>( \Lambda )</td>
</tr>
<tr>
<td>3</td>
<td>( \Lambda )</td>
<td>3</td>
<td>( \Lambda )</td>
<td>3</td>
<td>( \Lambda )</td>
</tr>
<tr>
<td>4</td>
<td>( \Lambda )</td>
<td>4</td>
<td>( \Lambda )</td>
<td>4</td>
<td>( \Lambda )</td>
</tr>
<tr>
<td>5</td>
<td>( \Lambda )</td>
<td>5</td>
<td>( \Lambda )</td>
<td>5</td>
<td>( \Lambda )</td>
</tr>
<tr>
<td>6</td>
<td>( \Lambda )</td>
<td>6</td>
<td>( \Lambda )</td>
<td>6</td>
<td>( \Lambda )</td>
</tr>
<tr>
<td>7</td>
<td>( \Lambda )</td>
<td>7</td>
<td>( \Lambda )</td>
<td>7</td>
<td>( \Lambda )</td>
</tr>
<tr>
<td>8</td>
<td>( \Lambda )</td>
<td>8</td>
<td>( \Lambda )</td>
<td>8</td>
<td>( \Lambda )</td>
</tr>
</tbody>
</table>
Chaining Example

Operations
- Find B, Find A

1 Λ 1 Λ
2 Λ 2 Λ
3 Λ 3 Λ
4 Λ 4 Λ
5 Λ 5 Λ
6 C - A 6 C - A
7 B 7 B
8 Λ 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**
    - if a.equals(b) == true
      - then a.hashCode( ) == b.hashCode( ) must be true