## Hashing

Chapter 5 in Weiss

CSE 373  
Data Structures and Algorithms  
Ruth Anderson

---

### Today’s Outline

- **Announcements**
  - Homework #4 coming soon:
    - Java programming: disjoint sets and mazes
    - due Thurs, Nov 8<sup>th</sup>
    - partners allowed—MUST declare by 11pm Wed Oct 31st
  - Midterm #2 – Fri, Nov 16

- **Today’s Topics:**
  - Hashing

---

### The Dictionary ADT

- **Data:**
  - a set of (key, value) pairs

- **Operations:**
  - Insert (key, value)
  - Find (key)
  - Remove (key)

---

### Dictionary Implementations

For dictionary with $n$ key/value pairs

- Insert: $O(1)$  
- Find: $O(n)$  
- Delete: $O(n)$

---

#### Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:
  
  ![Hash Table Diagram]

  - Key space (e.g., integers, strings)
  - TableSize = TableSize

---

#### Hash Tables

Key space of size $M$, but we only want to store subset of size $N$, where $N < M$

- Keys are identifiers in programs. Compiler keeps track of them in a symbol table.
- Keys are student names. We want to look up student records quickly by name.
- Keys are chess configurations in a chess playing program.
- Keys are URLs in a database of web pages.
**Example**

- key space = integers
- TableSize = 10
- \( h(K) = K \mod 10 \)
- Insert: 7, 18, 41, 94

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

**Another Example**

- key space = integers
- TableSize = 6
- \( h(K) = K \mod 6 \)
- Insert: 7, 18, 41, 34

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

**Hash Functions**

1. simple/fast to compute,
2. Avoid collisions
3. have keys distributed even among cells.

Perfect Hash function:

**Sample Hash Functions:**

- key space = strings
- \( s = s_0 \ s_1 \ s_2 \ldots \ s_{k-1} \)

1. \( h(s) = s_0 \mod \) TableSize
2. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \right) \mod \) TableSize
3. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \cdot 37 \right) \mod \) TableSize

**Designing a Hash Function for web URLs**

\( s = s_0 \ s_1 \ s_2 \ldots \ s_{k-1} \)

Issues to take into account:

\( h(s) = \)

**Collision Resolution**

**Collision:** when two keys map to the same location in the hash table.

Two ways to resolve collisions:
1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)
Separate Chaining

- Separate chaining: All keys that map to the same hash value are kept in a list ("bucket").

Analysis of find

- The load factor, $\lambda$, of a hash table is the ratio:
  
  \[
  \lambda = \frac{\text{no. of elements}}{\text{table size}}
  \]

  For separate chaining, $\lambda = \text{average # of elements in a bucket}$

  - unsuccessful:
  - successful:

How big should the hash table be?

- For Separate Chaining:

Open Addressing

- Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Terminology Alert!

- “Open Hashing” equals “Open Addressing”
- “Closed Hashing” equals “Separate Chaining”
Linear Probing

\[ f(i) = i \]

- Probe sequence:
  - 0th probe = \( h(k) \mod \text{TableSize} \)
  - 1st probe = \( (h(k) + 1) \mod \text{TableSize} \)
  - 2nd probe = \( (h(k) + 2) \mod \text{TableSize} \)
  
  \[ \ldots \]
  - \( i \)th probe = \( (h(k) + i) \mod \text{TableSize} \)

Load Factor in Linear Probing

- For any \( \lambda < 1 \), linear probing will find an empty slot
- Expected # of probes (for large table sizes)
  - successful search:
    \[ \frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)} \right) \]
  - unsuccessful search:
    \[ \frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right) \]
- Linear probing suffers from primary clustering
- Performance quickly degrades for \( \lambda > 1/2 \)

Quadratic Probing

\[ f(i) = i^2 \]

- Probe sequence:
  - 0th probe = \( h(k) \mod \text{TableSize} \)
  - 1st probe = \( (h(k) + 1) \mod \text{TableSize} \)
  - 2nd probe = \( (h(k) + 4) \mod \text{TableSize} \)
  - 3rd probe = \( (h(k) + 9) \mod \text{TableSize} \)
  
  \[ \ldots \]
  - \( i \)th probe = \( (h(k) + i^2) \mod \text{TableSize} \)

Quadratic Probing:

- \( h(k) = k \mod 7 \)
- Perform these inserts:
  - Insert(89)
  - Insert(18)
  - Insert(49)
  - Insert(58)
  - Insert(79)

Quadratic Probing:

- \( h(k) = k \mod 7 \)
- Perform these
  - Insert(85)
  - Insert(10)
  - Insert(47)

Insert:

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} 
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
\end{array} \]

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} 
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
\end{array} \]
Quadratic Probing Example

<table>
<thead>
<tr>
<th>Insert Value</th>
<th>Hash Value</th>
<th>Modulo 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>6</td>
<td>76%7 = 6</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>40%7 = 5</td>
</tr>
<tr>
<td>48</td>
<td>6</td>
<td>48%7 = 6</td>
</tr>
<tr>
<td>55</td>
<td>5</td>
<td>55%7 = 6</td>
</tr>
<tr>
<td>47</td>
<td>5</td>
<td>47%7 = 5</td>
</tr>
</tbody>
</table>

Quadratic Probing: Success guarantee for $\lambda < \frac{1}{2}$

- If size is prime and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in size/2 probes or fewer.
  - Show for all $0 \leq i, j \leq \text{size}/2$ and $i \neq j$:
    - $(h(x) + i^2) \mod \text{size} \neq (h(x) + j^2) \mod \text{size}$
  - By contradiction: suppose that for some $i \neq j$:
    - $(h(x) + i^2) \mod \text{size} = (h(x) + j^2) \mod \text{size}$
    - $i^2 \mod \text{size} = j^2 \mod \text{size}$
    - $(i^2 - j^2) \mod \text{size} = 0$
    - $(i + j)(i - j) \mod \text{size} = 0$
    - BUT size does not divide $(i-j)$ or $(i+j)$

Quadratic Probing: Properties

- For any $\lambda < \frac{1}{2}$, quadratic probing will find an empty slot; for bigger $\lambda$, quadratic probing may find a slot
- Quadratic probing does not suffer from primary clustering: keys hashing to the same area are not bad
- But what about keys that hash to the same spot? – Secondary Clustering!

Double Hashing

$f(i) = i \times g(k)$

where $g$ is a second hash function

- Probe sequence:
  - 0th probe = $h(k) \mod \text{TableSize}$
  - 1st probe = $(h(k) + g(k)) \mod \text{TableSize}$
  - 2nd probe = $(h(k) + 2\times g(k)) \mod \text{TableSize}$
  - 3rd probe = $(h(k) + 3\times g(k)) \mod \text{TableSize}$
    ...
  - $i$th probe = $(h(k) + i\times g(k)) \mod \text{TableSize}$

Double Hashing Example

$i$th probe = $(h(k) + i\times g(k)) \mod \text{TableSize}$

$h(k) = k \mod 7$ and $g(k) = 5 - (k \mod 5)$

<table>
<thead>
<tr>
<th>Probe</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>93</td>
<td>40</td>
<td>47</td>
<td>10</td>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resolving Collisions with Double Hashing

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

- 13
- 28
- 33
- 147
- 43
Rehashing

**Idea:** When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

---

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity.