Hashing
Chapter 5 in Weiss

CSE 373
Data Structures and Algorithms
Ruth Anderson

Today’s Outline
• Announcements
  – Homework #4 due Fri, May 7 at the beginning of class.

• Today’s Topics:
  – Disjoint Sets & Dynamic Equivalence
  – Hashing

Hash Tables
• Constant time accesses!
• A hash table is an array of some fixed size, usually a prime number.
• General idea: key space (e.g., integers, strings) → hash function: hash table

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

Another Example
• key space = integers
• TableSize = 6
• \( h(K) = K \mod 6 \)
• Insert: 7, 18, 41, 34
Hash Functions

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

Perfect Hash function:

Sample Hash Functions:

- key space = strings
- \( s = s_0 \ s_1 \ s_2 \ldots \ s_{k-1} \)
1. \( h(s) = s_0 \mod \text{TableSize} \)
2. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \right) \mod \text{TableSize} \)
3. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \cdot 37^i \right) \mod \text{TableSize} \)

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

<table>
<thead>
<tr>
<th>Insert:</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></td>
<td>10</td>
<td>22</td>
<td>107</td>
<td>12</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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:
  \( N \leftarrow \text{no. of elements} \)
  \( M \leftarrow \text{table size} \)
  For separate chaining, \( \lambda = \text{average # of elements in a bucket} \)

- unsuccessful:
  - successful:
How big should the hash table be?

- For Separate Chaining:

**tableSize: Why Prime?**

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10
    - data hashes to 0, 3, 0, 5, 1, 0, 0
  - tableSize = 11
    - data hashes to 10, 9, 5, 0, 2, 2, 7

Real-life data tends to have a pattern
Being a multiple of 11 is usually not the pattern

Open Addressing

<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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Insert:

- 38
- 19
- 8
- 109
- 10

- **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 “Separate Chaining”
- “Closed Hashing” equals “Open Addressing”

Linear Probing

\[ f(i) = i \]

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

Linear Probing – Clustering

[Diagram showing linear probing with clustering]

[R. Sedgewick]
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:
    \[
    \frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)}\right)
    \]
- Linear probing suffers from primary clustering
- Performance quickly degrades for $\lambda > 1/2$

Quadratic Probing

- $f(i) = i^2$
- Probe sequence:
  0th probe = $h(k) \mod$ TableSize
  1st probe = $(h(k) + 1) \mod$ TableSize
  2nd probe = $(h(k) + 4) \mod$ TableSize
  3rd probe = $(h(k) + 9) \mod$ TableSize
  ... 
  $i$th probe = $(h(k) + i^2) \mod$ TableSize

Quadratic Probing Example

<table>
<thead>
<tr>
<th>Insertions</th>
<th>Hashes</th>
<th>Probed Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

But... Insert(47)

Quadratic Probing: Less likely to encounter Primary Clustering

- Success guarantee for $\lambda < 1/2$
  - If size is prime and $\lambda < 1/2$, 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 $i \neq j$
      - $(h(x) + i^2) \mod$ size $\neq (h(x) + j^2) \mod$ size
      - by contradiction: suppose that for some $i \neq j$
        - $(h(x) + i^2) \mod$ size $= (h(x) + j^2) \mod$ size
        - $i^2 \mod$ size $= j^2 \mod$ size
        - $(i^2 - j^2) \mod$ size $= 0$
        - $(i + j)(i - j) \mod$ 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:
  - $0^{th}$ probe: $h(k) \mod \text{TableSize}$
  - $1^{st}$ probe: $(h(k) + g(k)) \mod \text{TableSize}$
  - $2^{nd}$ probe: $(h(k) + 2g(k)) \mod \text{TableSize}$
  - $3^{rd}$ probe: $(h(k) + 3g(k)) \mod \text{TableSize}$
  - ... $i^{th}$ probe: $(h(k) + i \times g(k)) \mod \text{TableSize}$

Resolving Collisions with Double Hashing

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

<table>
<thead>
<tr>
<th>0</th>
<th>13</th>
<th>28</th>
<th>33</th>
<th>147</th>
<th>43</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>76</td>
<td>67</td>
<td>40</td>
<td>50</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>

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.