Hash Tables

Chapter 5 in Weiss

Hash Tables

• Constant time accesses!

• A hash table is an array of some fixed size, usually a prime number.

• General idea:

  \[ \text{hash function: } h(K) \]

  key space (e.g., integers, strings) \hspace{1cm} \text{TableSize - 1} \]

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.

Sample Hash Functions:

• key space = strings
• \[ s = s_0 \cdot s_1 \cdot s_2 \ldots \cdot s_{k-1} \]
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>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:

10
22
107
12
42

• Separate chaining:
  All keys that map to the same hash value are kept in a list (or “bucket”).

Analysis of find

• Defn: The load factor, \( \lambda \), of a hash table is the ratio:
  \[ \frac{\text{no. of elements}}{\text{table size}} \]

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

• unsuccessful:
  \[ \lambda \]

• successful:
  \[ 1 + \lambda/2 \]

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, 9, 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>Insert</th>
<th>38</th>
<th>19</th>
<th>8</th>
<th>109</th>
<th>10</th>
</tr>
</thead>
</table>

- **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 “Closed Hashing”

Weiss

“Separate Chaining” “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 = \( (h(k) + 2) \mod \text{TableSize} \)
  - \( i \)th probe = \( (h(k) + i) \mod \text{TableSize} \)

Linear Probing – Clustering

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: \( \frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right) \)
  - unsuccessful search: \( \frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right) \)

- Linear probing suffers from primary clustering
- Performance quickly degrades for \( \lambda > 1/2 \)

Quantitatively

Number of probes vs \( \lambda \):
- dashed = linear
- solid = random resolution