CSE 326 Data Structures

Dave Bacon

Hashing
Logistics

• Turn in Homework 4

• Reading: Chapter 5, Start Chapter 8
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)  \[ \text{TableSize} - 1 \]

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

  hash table
Sample Hash Functions:

1. \( h(s) = s_0 \mod \text{TableSize} \)
2. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \right) \mod \text{TableSize} \)
3. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \cdot 37^i \right) \mod \text{TableSize} \)
tableSize: Why Prime?
Separate Chaining

Insert:
- 10
- 22
- 107
- 12
- 42

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

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.
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 $\lambda > 1/2$. 
Primary Clustering
Quadratic Probing

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

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

<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>

Insert:
- 89
- 18
- 49
- 58
- 79
Quadratic Probing Example

\[
\begin{align*}
\text{insert(76)} & \quad \text{insert(40)} & \quad \text{insert(48)} & \quad \text{insert(5)} & \quad \text{insert(55)} \\
76 \% 7 &= 6 & 40 \% 7 &= 5 & 48 \% 7 &= 6 & 5 \% 7 &= 5 & 55 \% 7 &= 6 \\
\end{align*}
\]

But... \quad \text{insert(47)}

\[
\begin{align*}
47 \% 7 &= 5 \\
\end{align*}
\]
Quadratic Probing:
Success guarantee for $\lambda < \frac{1}{2}$
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 $i \neq j$:
    
    $$(h(x) + i^2) \mod \text{size} = (h(x) + j^2) \mod \text{size}$$
    
    $\Rightarrow i^2 \mod \text{size} = j^2 \mod \text{size}$
    
    $\Rightarrow (i^2 - j^2) \mod \text{size} = 0$
    
    $\Rightarrow [(i + j)(i - j)] \mod \text{size} = 0$
    
    BUT size does not divide $(i-j)$ or $(i+j)$
Secondary Clustering
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 = \( (h(k) + 3g(k)) \mod \text{TableSize} \)
  
  \ldots

  \( i^{th}\) probe = \( (h(k) + ig(k)) \mod \text{TableSize} \)
Double Hashing Example

\[ h(k) = k \mod 7 \text{ and } g(k) = 5 - (k \mod 5) \]

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

Probes: 1 1 1 2 1 2
Resolving Collisions with Double Hashing

Hash Functions:
\[ H(K) = K \mod M \]
\[ H_2(K) = 1 + ((K/M) \mod (M-1)) \]

M =

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?**
<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
<th>Ugly</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
Disjoint Sets

Chapter 8
Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
- Each set has a unique name, one of its members
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
Union

- Union(x, y) – take the union of two sets named x and y
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
  - Union(5,1)
    - \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
Find

• Find(x) – return the name of the set containing x.
  – \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
  – Find(1) = 5
  – Find(4) = 8