Today’s Outline

• Announcements
  – Homework #4 coming:
    • Java programming: disjoint sets and mazes
    • due Thurs, Nov 10th
    • partners allowed

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

The Dictionary ADT

• Data:
  – a set of
    (key, value) pairs

• Operations:
  – Insert (key, value)
  – Find (key)
  – Remove (key)

The Dictionary ADT is sometimes called the "Map ADT"

Dictionary Implementations

For dictionary with \( n \) key/value pairs

<table>
<thead>
<tr>
<th>Insert</th>
<th>Find</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) ) *</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

*Note: If we do not allow duplicate values to be inserted, we would need to do \( O(n) \) work (a find operation) to check for a key’s existence before insertion.

Hash Tables

• Constant time accesses!
• A hash table is an array of some fixed size, usually a prime number.
• General idea:

  hash function: \( h(K) \)

  \[
  \begin{array}{c|c|c|c|c}
    0 & | & | & | \\
  \end{array}
  \]

  key space (e.g., integers, strings)
  TableSize = 1

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.

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

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

Insert:

0
1
2
3
4
5
6
7
8
9

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, \( \lambda \) = average # of elements in a bucket

unsuccessful:

successful:

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

- 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.

Insert:

0
1
2
3
4
5
6
7
8
9

Terminology Alert!

- "Open Hashing" equals "Separate Chaining"
- "Closed Hashing" equals "Open Addressing"

Weiss
Linear Probing

- $f(i) = i$

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

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

  - $i^{th}$ probe = $(h(k) + i^2) \mod \text{TableSize}$

Quadratic Probing: Less likely to encounter Primary Clustering

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

  
  Insert:
  - 89
  - 18
  - 49
  - 58
  - 79

  
  0 1 2 3 4 5 6 7 8 9
### Quadratic Probing Example

<table>
<thead>
<tr>
<th>Insertion Index</th>
<th>Inserted Value</th>
<th>Hash Value</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>76</td>
<td>76%7 = 6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>40%7 = 5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>48%7 = 6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5%7 = 5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
<td>55%7 = 6</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>47</td>
<td>47%7 = 5</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

But... 479^2 = 5

### 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 $0 \leq i, j \leq \frac{\text{size}}{2}$ 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 $i \neq j$:
  \[(h(x) + i^2) \mod \text{size} \neq (h(x) + j^2) \mod \text{size} \]
  \[\Rightarrow i^2 \mod \text{size} = j^2 \mod \text{size} \]
  \[\Rightarrow (i^2 - j^2) \mod \text{size} = 0 \]
  \[\Rightarrow (i - j)(i + j) \mod \text{size} = 0 \]
  BUT size does not divide $(i-j)$ or $(i+j)$

### 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 = $(h(k) + 2k g(k)) \mod \text{TableSize}$
  - 3rd probe = $(h(k) + 3g(k)) \mod \text{TableSize}$
    
    \[\ldots\]
    
    \[i^{th} \text{ probe} = (h(k) + i\times g(k)) \mod \text{TableSize}\]

### Double Hashing Example

- $i^{th}$ probe = $(h(k) + i \times g(k)) \mod \text{TableSize}$
- $h(k) = k \mod 7$ and $g(k) = 5 - (k \mod 5)$

### Resolving Collisions with Double Hashing

<table>
<thead>
<tr>
<th>Hash Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(k) = k \mod M$</td>
</tr>
<tr>
<td>$h_2(k) = 1 + ((k/M) \mod (M-1))$</td>
</tr>
<tr>
<td>$M = 5$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>147</td>
<td>3</td>
</tr>
<tr>
<td>43</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

Probes 1 1 1 1 2 1 1 2
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.