Hash Tables

CSE 373
Data Structures & Algorithms
Ruth Anderson

Today’s Outline

• Announcements
  – Assignment #4 due this Friday Feb 13th at the beginning of lecture.

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

Dictionary Implementations

<table>
<thead>
<tr>
<th></th>
<th>Unsorted linked list</th>
<th>Sorted Array</th>
<th>Binary Search Tree</th>
<th>AVL Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td></td>
<td></td>
<td>O(log N)</td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td></td>
<td>O(N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td>O(N)</td>
<td>O(log N)</td>
<td></td>
</tr>
</tbody>
</table>

Constant Time Access

Data Set:
• 100 students
• Keys = Student numbers between 0 and 99.

Solution:
• Array of size 0-99.
• One-to-one mapping: e.g. student number 2 goes in location 2

Hash Tables

• A hash table is an array of some fixed size.
• General idea:

  Key Space (e.g., integers, strings)  Table Size – 1

  Hash Function: h(K)

  0  1  2  …

  Hash Table

  0  1  2  …
Hash Functions

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

Perfect Hash function:

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. \( h(s) = \left( \sum_{i=0}^{k-1} s_i - 26 \right) \mod \text{TableSize} \)

Collison 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

\[ h(K) = K \mod 10 \]

**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

The load factor, \( \lambda \), of a hash table is the ratio:

\[
\lambda = \frac{\text{# of elements}}{\text{table size}}
\]

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

Average # of values needed to examine for a:
- unsuccessful find:
- successful find:

---

How Big Should the Hash Table Be?

For Separate Chaining, if we want \( \lambda = 1 \)
(e.g. the average # of values per bucket = 1)
- How large should I make the hash table, in terms of \( N \)?

TableSize =

---

Open Addressing

\[ h(K) = K \mod 10 \]

**Insert:**
- 38
- 19
- 8
- 109
- 10

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

---

Terminology Alert!

“WeOpen Hashing” equals “Closed Hashing”

“Separate Chaining” “Open Addressing”

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

\[ f(i) = i \]

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

Write pseudocode for find\( (k) \) for Open Addressing with linear probing
- Find\( (k) \) returns \( i \) where \( T(i) = k \)

Linear Probing – Clustering

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

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 Example

- \( \text{insert}(76) \)
  - \( 76 \mod 7 = 6 \)
- \( \text{insert}(40) \)
  - \( 40 \mod 7 = 5 \)
- \( \text{insert}(48) \)
  - \( 48 \mod 7 = 6 \)
- \( \text{insert}(5) \)
  - \( 5 \mod 7 = 5 \)
- \( \text{insert}(55) \)
  - \( 55 \mod 7 = 6 \)
- \( \text{insert}(47) \)
  - \( 47 \mod 7 = 5 \)

But… \( \text{insert}(47) \)  \( 47 \mod 7 = 5 \)

Quadratic Probing:

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 \( i \neq j \):
    - \( (h(x) + i^2) \mod \text{size} \neq (h(x) + j^2) \mod \text{size} \)
  - 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 \( \lambda < 1/2 \), 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

\[ f(i) = i \ast g(k) \]

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 = \( (h(k) + 2 \ast g(k)) \mod \text{TableSize} \)
  - 3rd probe = \( (h(k) + 3 \ast g(k)) \mod \text{TableSize} \)
  - ...
  - \( i \)th probe = \( (h(k) + i \ast g(k)) \mod \text{TableSize} \)

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>
</tbody>
</table>

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

- 13
- 28
- 33
- 147
- 43

Double Hashing Example

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

<table>
<thead>
<tr>
<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>1</td>
<td>40</td>
<td>40</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>2</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</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 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.