Today’s Outline

- **Admin:**
  - HW #5 – due Monday May 19
  - Wed at beginning of class = latest accepted
  - Midterm #2 – Friday May 23
  - Feedback Survey

- **Hash Tables** (Weiss Chapter 5)

### 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><strong>Insert</strong></td>
<td></td>
<td></td>
<td></td>
<td>O(log N)</td>
</tr>
<tr>
<td><strong>Find</strong></td>
<td>O(N)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Delete</strong></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

### Constant Time Access?

**Data Set:**
- 100 students
- Keys = Student numbers between 0 and 999999999.

**Solution:**
- Array of size ?
- Mapping ?

### Hash Tables

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

```
Key Space (e.g., integers, strings) → Hash Table

Hash Function: h(K) → TableSize – 1
```
Example

- Key space = integers
- TableSize = 10
- \( h(K) = K \mod 10 \)
- **Insert:** 207, 18, 41, 194, 19, 43

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
  - \( A=0, B=1, \ldots, Z=25 \)
  - \( s = s_0 \ s_1 \ s_2 \ldots \ s_{k-1} \)

1. \( h(s) = s_0 \mod \text{TableSize} \)
2. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \right) \mod \text{TableSize} \)
3. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \cdot 26^i \right) \mod \text{TableSize} \)

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

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

Insert:

0 10
1 22
2 107
3 12
4 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:

\[ \frac{N}{\text{TableSize}} \]

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?

\[ \text{TableSize} = \]

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, 2, 9, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually not the pattern ☹

Open Addressing

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

Insert:

0 38
1 19
2 8
3 109
4 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!

“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 = \( (h(k) + 2) \mod \text{TableSize} \)
  \ldots
  ith 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

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 \( \lambda > 1/2 \)

Quadratic Probing

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

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

Quadratic Probing

Less likely to encounter Primary Clustering

Insert:

```
0
1
2
3
4
5
6
7
8
9
```

<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>89</td>
<td>18</td>
<td>49</td>
<td>58</td>
<td>79</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quadratic Probing Example

<table>
<thead>
<tr>
<th>Insertion</th>
<th>Hash Modulo Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(76)</td>
<td>6</td>
</tr>
<tr>
<td>(40)</td>
<td>5</td>
</tr>
<tr>
<td>(48)</td>
<td>6</td>
</tr>
<tr>
<td>(5)</td>
<td>5</td>
</tr>
<tr>
<td>(55)</td>
<td>6</td>
</tr>
<tr>
<td>(47)</td>
<td>5</td>
</tr>
</tbody>
</table>

But…

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 $i \neq j$:
  
  $$(h(x) + i^2) \mod \text{size} = (h(x) + j^2) \mod \text{size}$$

  $\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 = $(h(k) + 2g(k)) \mod \text{TableSize}$

  3rd probe = $(h(k) + 3g(k)) \mod \text{TableSize}$

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

Resolving Collisions with Double Hashing

<table>
<thead>
<tr>
<th>Hash Functions:</th>
<th>Insert these values into the hash table in this order. Resolve any collisions with double hashing:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H(K) = K \mod M$</td>
<td>13</td>
</tr>
<tr>
<td>$H_2(K) = 1 + ((K/M) \mod (M-1))$</td>
<td>28</td>
</tr>
<tr>
<td>$M = 147$</td>
<td>33</td>
</tr>
<tr>
<td>43</td>
<td>147</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.