Review: Hash Table Approach

```plaintext
Review: A Good Hash Function…

…is easy (fast) to compute
(O(1) and practically fast)

…distributes the data evenly ⇒ few collisions
(ideally, hash(a) % size ≠ hash(b) % size ⇒ no collision)

…uses the whole hash table
(∀ k, 0 ≤ k < size, ∃ i such that hash(i) % size = k)

Collisions

• Pigeonhole principle says we can’t avoid all collisions
  – try to hash without collision m keys into n slots with m > n
  – e.g., try to put 7 pigeons into 5 holes

• What do we do when two keys hash to the same entry?
  1. Separate chaining: put little dictionaries in each entry
    above extra pigeons in one hole!
  2. Open addressing: pick a next entry to try
```

Today’s Outline

- Admin
  - Project 2 due tonight!
  - Pick up Homework 2; due Friday
  - A word on collaboration and acknowledgement

- Hashing: collision resolution strategies
  - Separate chaining
  - Open addressing
    - Linear probing, quadratic probing, double hashing
  - Rehashing

Review: Hash Table Code

```plaintext
value find(Key k) {
  int index = hash(k) % tableSize;
  return Table[index];
}
```
Load Factor

How often do collisions occur?

- Depends on the load factor, \( \lambda \)

\[
\lambda = \frac{\text{# of entries in table}}{\text{tableSize}}
\]

High \( \lambda \) \( \Rightarrow \) more collisions, bad performance
Low \( \lambda \) \( \Rightarrow \) less collisions, good performance

1. Separate Chaining

- Put a mini-Dictionary at each entry
  - Usually a linked list
  - Why not a search tree?
- Properties
  - Average list size =
  - Works even when \( \lambda > 1 \)
  - Performance degrades with length of chains

Remember Splay Trees?

- Where in the list would you put a new entry?
- What might you do when you perform find on a key?

Load Factor in Separate Chaining

- Search cost
  - unsuccessful search:
- successful search:
- Desired load factor:

2. Open Addressing

What if we only allow one Key at each entry?
- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must probe for another spot

- Properties
  - Requires \( \lambda \leq 1 \)
  - Performance degrades with difficulty of finding right spot

Salary-Boosting Obfuscation

“Open Hashing” equals “Closed Hashing” equals
“Separate Chaining” “Open Addressing”
Probing Function, $f(x)$

- The Probing process
  - First probe: given a key $k$, hash to $h(k)$
  - Second probe: if $h(k)$ is occupied, try $h(k) + f(1)$
  - Third probe: if $h(k) + f(1)$ is occupied, try $h(k) + f(2)$
  - And so on.

- Probing properties
  - force $f(0) = 0$
  - the $i$th probe is to $(h(k) + f(i)) \mod \text{size}

- When does the probe fail?

- Does that mean the table is full?

2a. Linear Probing

- Probe sequence is
  - $h(k) \mod \text{size}$
  - $(h(k) + 1) \mod \text{size}$
  - $(h(k) + 2) \mod \text{size}$
  - ...

Linear Probing Example

<table>
<thead>
<tr>
<th>Insert 76</th>
<th>Insert 93</th>
<th>Insert 40</th>
<th>Insert 47</th>
<th>Insert 10</th>
<th>Insert 55</th>
</tr>
</thead>
<tbody>
<tr>
<td>76/7 = 6</td>
<td>93/7 = 2</td>
<td>40/7 = 5</td>
<td>47/7 = 5</td>
<td>10/7 = 3</td>
<td>55/7 = 6</td>
</tr>
</tbody>
</table>

Problem?

Load Factor in Linear Probing

- Search cost
  - Unsuccessful search
  - Successful search

Load Factor in Linear Probing

- For any $\lambda < 1$, linear probing will find an empty slot
- Search cost (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)} \right)

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

2b. Quadratic Probing

- Probe sequence is
  - $h(k) \mod \text{size}$
  - $(h(k) + 1) \mod \text{size}$
  - $(h(k) + 4) \mod \text{size}$
  - $(h(k) + 9) \mod \text{size}$
  - ...

- Implementation trick: $f(i+1) = f(i^2)$
  - No multiplication!
Quadratic Probing Example

<table>
<thead>
<tr>
<th>insert(76)</th>
<th>insert(40)</th>
<th>insert(48)</th>
<th>insert(5)</th>
<th>insert(55)</th>
</tr>
</thead>
<tbody>
<tr>
<td>76 mod 7 = 6</td>
<td>40 mod 7 = 5</td>
<td>48 mod 7 = 6</td>
<td>5 mod 7 = 5</td>
<td>55 mod 7 = 6</td>
</tr>
</tbody>
</table>

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 $\frac{\text{size}}{2}$ probes or fewer.
- show for all $0 \leq i, j \leq \text{size}/2$ and $i \neq j$:
  - by contradiction: suppose that for some $i \neq j$:
    - $(h(x) + i^2)$ mod size $= (h(x) + j^2)$ mod size
    - $i^2$ mod size $= j^2$ mod size
    - $(i^2 - j^2)$ mod size $= 0$
    - but how can $i = j = 0$ or $i = j = \text{size}$ when $i \neq j$ and $i, j \leq \text{size}/2$?
    - same for $i - j$ mod size $= 0$

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!

A Good Double Hash Function…

…is quick to evaluate.
…differs from the original hash function – keys that $h_1$ hashes close by must hash far away using $h_2$
…never evaluates to 0 (mod size).

One good choice is to choose a prime $R < \text{size}$ and:

$$\text{hash}_2(k) = R - (k \mod R)$$

What could go wrong if table size $S$ were not prime?

Double Hashing Example (R=5)

<table>
<thead>
<tr>
<th>insert(76)</th>
<th>insert(93)</th>
<th>insert(40)</th>
<th>insert(47)</th>
<th>insert(10)</th>
<th>insert(55)</th>
</tr>
</thead>
<tbody>
<tr>
<td>76 mod 7 = 6</td>
<td>93 mod 7 = 2</td>
<td>40 mod 7 = 5</td>
<td>47 mod 7 = 5</td>
<td>10 mod 7 = 3</td>
<td>55 mod 7 = 6</td>
</tr>
</tbody>
</table>

Probes:

$$f(i) = i \cdot \text{hash}_2(k)$$

• Probe sequence is
  - $h_2(k)$ mod size
  - $(h_1(k) + 1 \cdot h_2(k))$ mod size
  - $(h_1(k) + 2 \cdot h_2(k))$ mod size
  - …

• Goal?
Load Factor in Double Hashing

- For any $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and hash$_2$).
- Search cost appears to approach optimal (random hash):
  - successful search: $\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$
  - unsuccessful search: $\frac{1}{1-\lambda}$
- No primary clustering and no secondary clustering
- Cost?

The Squished Pigeon Principle

- An insert using open addressing cannot work with a load factor of 1 or more.
- An insert using open addressing with quadratic probing may not work with a load factor of $\frac{1}{2}$ or more.
- Whether you use separate chaining or open addressing, large load factors lead to poor performance!

*How can we relieve the pressure on the pigeons?*

Deletion with Open Addressing

- When the load factor gets "too large" (over a constant threshold on $\lambda$), rehash all the elements into a new, larger table:
  - spreads keys back out, may drastically improve performance
  - avoids failure for open addressing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items
- Cost?

Rehashing Example

- Can we just copy over into a bigger array?