CSE 326: Data Structures
It’s an open-and-closed hash!

Luke McDowell
Summer Quarter 2003

Reminder: Dictionary ADT

- Dictionary ADT:
  - Maps values to user-specified keys
  - Or: a set of (key, value) pairs
  - Keys may be any (homogeneous) type
  - Values may be any (homogeneous) type
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

The Dictionary ADT is sometimes called the "Map ADT"

Implementations So Far

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted list</td>
<td>O(n)</td>
<td>O(log n)?</td>
<td>O(n)</td>
</tr>
<tr>
<td>Trees</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

How about O(1) insert/find/delete?

Hash Table Goal

We can do:

```
a[2] = "erbad"
```

We want to do:

```
a["erbad"] = "Aiman..."
```

What could go wrong?

Hash Table Code

First Pass

```
Value find(Key k) {
  int index = hash(k) % tableSize;
  return Table[index];
}
```

Key Questions:
1. What should the hash function be?
2. How should we resolve collisions?
3. What should the table size be?
A Good Hash Function…

…is easy (fast) to compute (O(1) and practically fast).
…distributes the data evenly (hash(a) % size = hash(b) % size).
…uses the whole hash table (for all 0 ≤ k < size, there’s an i such that hash(i) % size = k).

Good Hash Function for Integers

- Choose
  - tableSize is prime
  - hash(n) = n
- Example:
  - tableSize = 7
    - insert(4)
    - insert(17)
    - find(12)
    - insert(9)
    - delete(17)

Easy/boring stuff we’re going to skip

- Why does the table size have to be prime?
- Picking a good hash function for strings

Read Section 5.2 of the text!

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?
  - Separate chaining: put little dictionaries in each entry
  - Open addressing: pick a next entry to try
- Frequency depends on load factor
  - load factor = \# of entries in table / tableSize

Separate Chaining

- Put a mini-Dictionary at each entry
  - Usually a linked list
  - Why not a search tree?
- Properties
  - ? can be greater than 1
  - performance degrades with length of chains

Load Factor in Separate Chaining

- Search cost
  - unsuccessful search:
  - successful search:
- Desired load factor:
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
  - ? ? !
  - performance degrades with difficulty of finding right spot

Salary-Boosting Obfuscation

“Open Hashing” equals “Closed Hashing”
“Separate Chaining” equals “Open Addressing”

Probing

• Probing how to:
  - 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 forth

• Probing properties
  - we force f(0) = 0
  - the i-th probe is to (h(k) + f(i)) mod size

• When does the probe fail?

  • Does that mean the table is full?

Linear Probing

f(i) = i

• Probe sequence is
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size
  - ...

Linear Probing Example

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

Problem?

Load Factor in Linear Probing

• For any \( \beta < 1 \), linear probing will find an empty slot
• Search cost (for large table sizes)
  - successful search:
    \[ \frac{1 \times \beta + 1}{2 \times \beta} \]
  - unsuccessful search:
    \[ \frac{1 \times \beta + 1}{3 \times \beta} \]

• Linear probing suffers from primary clustering
• Performance quickly degrades for \( \beta > 1/2 \)
Quadratic Probing
\( f(i) = i^2 \)

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

Quadratic Probing Example

<table>
<thead>
<tr>
<th>Insertion Value</th>
<th>Insertion Value % 7</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>48</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

But...

Quadratic Probing Succeeds (for \( \frac{\text{load factor}}{2} < \frac{1}{2} \))

- If size is prime and \( \frac{\text{load factor}}{2} < \frac{1}{2} \), 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 \( i \neq j \)
  1. \( h(x) + i^2 \mod \text{size} = h(x) + j^2 \mod \text{size} \)
  2. By contradiction: suppose that for some \( i \neq j \):
     - \( h(x) + i^2 \mod \text{size} = h(x) + j^2 \mod \text{size} \)
     - \( i^2 \mod \text{size} = j^2 \mod \text{size} \)
     - \( (i^2 - j^2) \mod \text{size} = 0 \)
     - But how can \( i + j \) = 0 or \( i - j \) = size when
     - \( i \neq j \) and \( i, j \leq \text{size}/2 \)
  - Same for \( i - j \mod \text{size} = 0 \)

Load Factor in Quadratic Probing

- For any \( \frac{\text{load factor}}{2} < \frac{1}{2} \), quadratic probing will find an empty slot; for greater \( \frac{\text{load factor}}{2} \), quadratic probing may find a slot
- Quadratic probing does not suffer from primary clustering
- But what about keys that hash to the same spot?

Double Hashing
\( f(i) = i \mod \text{hash}_2(x) \)

- Probe sequence is
  - \( \text{hash}_2(x) \mod \text{size} \)
  - \( (\text{hash}_2(k) + 1 \mod \text{size}) \mod \text{size} \)
  - \( (\text{hash}_2(k) + 2 \mod \text{size}) \mod \text{size} \)
  - ...

- Goal?

A Good Double Hash Function...

...is quick to evaluate.
...differs from the original hash function.
...never evaluates to 0 (mod size).

One good choice is to choose a prime \( R < \text{size} \) and:
\( \text{hash}_2(x) = R - (x \mod R) \)
Double Hashing 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>

probes: 1 1 1 2 1 2

Load Factor in Double Hashing

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

Deletion with Open Addressing

<table>
<thead>
<tr>
<th>Insert (7)</th>
<th>Delete (2)</th>
<th>Find (7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

- Solution?

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?

Rehashing

- When the load factor gets "too large" (over a constant threshold on \( \alpha \)), rehash all the elements into a new, larger table:
  - spreads keys back out, may drastically improve performance
  - avoids failure for closed hashing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items
- Cost?
- Can we just copy over into a bigger array?

Rehashing Example

<table>
<thead>
<tr>
<th>20</th>
<th>96</th>
<th>82</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Extendible Hashing
Hashing technique for huge data sets
- Optimizes to reduce disk accesses
- Each hash bucket fits on one disk block
- Better than B-Trees if order is not important

Table contains:
- Buckets, each fitting in one disk block, with the data
- A directory is used to hash to the correct bucket

Extendible Hash Table
- Directory entry: key prefix (first $k$ bits) and a pointer to the bucket with all keys starting with its prefix
- Each bucket contains keys matching on first $j \geq k$ bits, plus the value associated with each key

If Extendible Hashing Doesn’t Cut It
Option 1: Store only pointers/references to the items: (key, value) pairs are in disk
Option 2: Improve Hash + Rehash

The One-Slide Hash
Hash function: maps keys to integers
Collision resolution
- Separate Chaining
  - Expand beyond hashable via secondary Dictionaries
  - Allows $7 > 1$
- Open Addressing
  - Expand within hashable
  - Secondary probing: linear, quadratic, double hash
  - $\approx 1$ (by definition)
  - $\approx 1$ (by preference)

Choosing a Hash Function
- Make sure table size is prime
- Careful choice for strings
- "Perfect hashing"
  - If keys known in advance, tune hash function for them!

Rehashing
- Tunes up hashable when $?$ crosses the line
Exmalelible hashing
- For disk-based data
- Combine with B-tree directory if needed

Implementations So Far
- Unsorted list $O(1)$ $O(n)$ $O(n)$
- Sorted list $O(n)$ $O(\log n)$? $O(n)$
- Trees $O(\log n)$ $O(\log n)$ $O(\log n)$
- Hash Table $O(1)$ $O(1)$ $O(1)$

Is there anything a hash table can’t do fast?