Hash Tables

Open Address Hashing
Goals

• Open Address Hashing
There are two general approaches to resolving collisions:

1. Open address hashing: if a spot is full, probe for next empty spot

2. Chaining (or buckets): keep a collection at each table entry
Open Address Hashing

• All values are stored in an array
• Hash value is used to find initial index
• If that position is filled, the next position is examined, then the next, and so on until you find the element OR an empty position is found
• The process of looking for an empty position is termed **probing**, specifically linear probing when we look to the next element
Open Address Hashing: Example

Eight element table using Amy’s hash function (alphabet position of the 3rd letter of the name -1):

Already added: Amina, Andy, Alessia, Alfred, and Aspen

<table>
<thead>
<tr>
<th>Amina</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
</tr>
<tr>
<td>5-fnv</td>
<td>6-gow</td>
<td>7-hpx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: We’ve shown where each letter of the alphabet maps to for simplicity here (given a table size of 8) …so you don’t have to calculate it!

e.g. Y is the 25th letter (we use 0 index, so the integer value is 24) and 24 mod 8 is 0
Now we need to add: Aimee

Add: Aimee

The hashed index position (4) is filled by Alessia: so we probe to find next free location
Suppose **Anne** wants to join:

Add: **Anne**

The hashed index position (5) is filled by Alfred:

– Probe to find next free location ➔ **what happens when we reach the end of the array**
Open Address Hashing: Adding (cont.)

Suppose Anne wants to join:

Add: Anne

The hashed index position (5) is filled by Alfred:
- Probe to find next free location
- When we get to end of array, wrap around to the beginning
- Eventually, find position at index 1 open
Finally, **Alan** wants to join:

Add: **Alan**

The hashed index position (0) is filled by Amina:
- Probing finds last free position (index 2)
- Collection is now completely filled
- What should we do if someone else wants to join? (More on this later)
Open Address Hashing: Contains

- Hash to find initial index
- probe forward until
  - value is found, or (return 1)
  - empty location is found (return 0)

<table>
<thead>
<tr>
<th>Amina</th>
<th>0-aiqy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>1-bjrz</td>
</tr>
<tr>
<td>Alessia</td>
<td>2-cks</td>
</tr>
<tr>
<td>Alfred</td>
<td>3-dlt</td>
</tr>
<tr>
<td>Aimee</td>
<td>4-emu</td>
</tr>
<tr>
<td>Aspen</td>
<td>5-fnv</td>
</tr>
<tr>
<td></td>
<td>6-gow</td>
</tr>
<tr>
<td></td>
<td>7-hpx</td>
</tr>
</tbody>
</table>

- Notice that search time is not uniform
Open Address Hashing: Remove

- Remove is tricky
- What happens if we delete **Anne**, then search for **Alan**?

**Remove:** **Anne**

<table>
<thead>
<tr>
<th>Amina</th>
<th>Anne</th>
<th>Alan</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aimee</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
<td>5-fnv</td>
<td>6-gow</td>
<td>7-hpx</td>
</tr>
</tbody>
</table>

**Find:** **Alan**

Hashes to

Probing finds null entry ➔ **Alan not found**

<table>
<thead>
<tr>
<th>Amina</th>
<th>Alan</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aimee</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
<td>5-fnv</td>
<td>6-gow</td>
</tr>
</tbody>
</table>
Open Address Hashing: Handling Remove

- Simple solution: Don’t allow removal (e.g. words don’t get removed from a spell checker!)
- Better solution: replace removed item with “tombstone”
  - Special value that marks deleted entry
  - Can be replaced when adding new entry
  - But doesn’t halt search during contains or remove

Find: Alan

Hashes to

<table>
<thead>
<tr>
<th>Amina</th>
<th><em>TS</em></th>
<th>Alan</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aimee</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
<td>5-fnv</td>
<td>6-gow</td>
<td>7-hpx</td>
</tr>
</tbody>
</table>

Probing skips tombstone → Alan found
Hash Table Size: Load Factor

Load factor:

\[ \lambda = \frac{n}{m} \]

- represents the portion of the tables that is filled
- For open address hashing, load factor is between 0 and 1 (often somewhere between 0.5 and 0.75)

Want the load factor to remain small in order to avoid collisions - space/speed tradeoff again!
Hash Tables: Algorithmic Complexity

• Assumptions:
  – Time to compute hash function is constant
  – Worst case analysis → All values hash to same position
  – Best case analysis → Hash function uniformly distributes the values and there are no collisions

• Find element operation:
  – Worst case for open addressing → O(n)
  – Best case for open addressing → O(1)
Hash Tables: Average Case

• What about average case for successful, S, and unsuccessful searches, U?

\[ S \approx \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right) \]

\[ U \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right) \]

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>S</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>0.75</td>
<td>2.5</td>
<td>8.5</td>
</tr>
<tr>
<td>0.9</td>
<td>5.5</td>
<td>50.5</td>
</tr>
</tbody>
</table>

• If \( \lambda \) is constant, average case is O(1), but want to keep \( \lambda \) small
• Assuming uniform distribution of hash values, what’s the probability that the next value will end up in index 6? in index 2? in index 1?

- Amina: 0-aiqy (3/8)
- Andy: 1-bjrz (1/8)
- Alessia: 2-cks (3/8)
- Alfred: 3-dlt
- 4-emu
- 5-fnv
- 6-gow (4/8)
- Aspen: 7-hpx

• As load factor gets larger, the tendency to cluster increases, resulting in longer search times upon collision.
Performance vs. Load Factor

http://en.wikipedia.org/wiki/Hash_table
Double Hashing

• Rather than use a linear probe (ie. looking at successive locations)...
  – Use a second hash function to determine the probe step

• Helps to reduce clustering
Large Load Factor: What to do?

- Common solution: When load factor becomes too large (say, bigger than 0.75) → Reorganize

- Create new table with twice the number of positions

- Copy each element, *rehashing* using the new table size, placing elements in new table

- Delete the old table
Hashing in Practice

• Need to find good hash function → uniformly distributes keys to all indices

• Open address hashing:
  – Need to tell if a position is empty or not
  – One solution → store only pointers & check for null (== 0)
Your Turn

• Complete Worksheet #37: Open Address Hashing