Hash Tables: Review

- Aim for constant-time (i.e., $O(1)$) find, insert, and delete
  - “On average” under some reasonable assumptions
- A hash table is an array of some fixed size
  - But growable as we’ll see

Collision resolution

Collision:
When two keys map to the same location in the hash table
We try to avoid it, but number-of-keys exceeds table size
So hash tables should support collision resolution
  - Ideas?

Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")
As easy as it sounds
Example:
insert 10, 22, 107, 12, 42 with mod hashing and $\text{TableSize} = 10$
Separate Chaining

Chaining:
All keys that map to the same
table location are kept in a list
(a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10

Thoughts on chaining

- Worst-case time for find?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each
    bucket)

- Beyond asymptotic complexity, some "data-structure
  engineering" may be warranted
  - Linked list vs. array vs. chunked list (lists should be short!)
  - Move-to-front
  - Maybe leave room for 1 element (or 2?) in the table itself, to
    optimize constant factors for the common case
  - A time-space trade-off...

More rigorous chaining analysis

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}}$$

number of elements

Under chaining, the average number of elements per bucket is ___
More rigorous chaining analysis

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}}$$

$\leftarrow$ number of elements

Under chaining, the average number of elements per bucket is $\lambda$.

So if some inserts are followed by random finds, then on average:

• Each unsuccessful find compares against ____ items

• Each successful find compares against ____ items

So we like to keep $\lambda$ fairly low (e.g., 1 or 1.5 or 2) for chaining.

Alternative: Use empty space in the table

• Another simple idea: If $h(\text{key})$ is already full,
  – try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,
  – try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,
  – try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

• Example: insert 38, 19, 8, 109, 10

  0  /  1  /  2  /  3  /  4  /  5  /  6  /  7  /  8  38  9  19
**Probing hash tables**

Trying the next spot is called **probing** (also called open addressing)

- We just did linear probing
- \( h(key) + i \mod \text{TableSize} \)
- In general have some probe function \( f \) and use \( h(key) + f(i) \mod \text{TableSize} \)

Open addressing does poorly with high load factor \( \lambda \)

- So want larger tables
- Too many probes means no more \( O(1) \)

---

**Alternative: Use empty space in the table**

- Another simple idea: If \( h(key) \) is already full,
  - try \( h(key) + 1 \mod \text{TableSize} \). If full,
  - try \( h(key) + 2 \mod \text{TableSize} \). If full,
  - try \( h(key) + 3 \mod \text{TableSize} \). If full...
- Example: insert 38, 19, 8, 109, 10

---

**Other operations**

**insert** finds an open table position using a probe function

What about **find**?

- Must use same probe function to "retrace the trail" for the data
- Unsuccessful search when reach empty position

What about **delete**?

- Must use "lazy" deletion. Why?
  - Marker indicates "no data here, but don’t stop probing"
  - Note: **delete** with chaining is plain-old list-remove

---

**Analysis of Linear Probing**

- Trivial fact: For any \( \lambda < 1 \), linear probing will find an empty slot
  - It is "safe" in this sense: no infinite loop unless table is full

- Non-trivial facts we won’t prove:
  - Average # of probes given \( \lambda \) (in the limit as \( \text{TableSize} \to \infty \))
    - Unsuccessful search: \( \frac{1}{2} \left( 1 - \frac{1}{1 - \lambda} \right) \)
    - Successful search: \( \frac{1}{2} \left( 1 - \frac{1}{1 - \lambda} \right) \)
  - This is pretty bad: need to leave sufficient empty space in the table to get decent performance (see chart)

---

**(Primary) Clustering**

It turns out linear probing is a bad idea, even though the probe function is quick to compute (which is a good thing)

Tends to produce clusters, which lead to long probing sequences

- Called **primary clustering**
- Saw this starting in our example

---

**R. Sedgewick**

Tends to produce clusters, which lead to long probing sequences

- Called **primary clustering**
- Saw this starting in our example

---

**Other operations**

**insert** finds an open table position using a probe function

What about **find**?

- Must use same probe function to "retrace the trail" for the data
- Unsuccessful search when reach empty position

What about **delete**?

- Must use "lazy" deletion. Why?
  - Marker indicates "no data here, but don’t stop probing"
  - Note: **delete** with chaining is plain-old list-remove

---

**Analysis of Linear Probing**

- Trivial fact: For any \( \lambda < 1 \), linear probing will find an empty slot
  - It is "safe" in this sense: no infinite loop unless table is full

- Non-trivial facts we won’t prove:
  - Average # of probes given \( \lambda \) (in the limit as \( \text{TableSize} \to \infty \))
    - Unsuccessful search: \( \frac{1}{2} \left( 1 - \frac{1}{1 - \lambda} \right) \)
    - Successful search: \( \frac{1}{2} \left( 1 - \frac{1}{1 - \lambda} \right) \)
  - This is pretty bad: need to leave sufficient empty space in the table to get decent performance (see chart)
In a chart

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes "large table" but point remains)
- By comparison, chaining performance is linear in $\lambda$ and has no trouble with $\lambda > 1$

Quadratic probing

- We can avoid primary clustering by changing the probe function
  \[ (h(key) + f(i)) \mod \text{TableSize} \]
- A common technique is quadratic probing:
  \[ f(i) = i^2 \]
  - So probe sequence is:
    0th probe: $h(key) \mod \text{TableSize}$
    1st probe: $(h(key) + 1) \mod \text{TableSize}$
    2nd probe: $(h(key) + 4) \mod \text{TableSize}$
    3rd probe: $(h(key) + 9) \mod \text{TableSize}$
    ...
    $i^{th}$ probe: $(h(key) + i^2) \mod \text{TableSize}$
- Intuition: Probes quickly "leave the neighborhood"
### Quadratic Probing Example

Table Size = 10

<table>
<thead>
<tr>
<th>0</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>79</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
</tr>
</tbody>
</table>

Insert: 89

<table>
<thead>
<tr>
<th>0</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>79</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>79</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
</tr>
</tbody>
</table>

Insert: 18

<table>
<thead>
<tr>
<th>0</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>79</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
</tr>
</tbody>
</table>

### Another Quadratic Probing Example

Table Size = 7

<table>
<thead>
<tr>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
| 2   | 76  
   |    |
| 3   | 40  
   |    |
| 4   | 48  
   |    |
| 5   | 5   
   |    |
| 6   | 47  
   |    |

Insert: 76 (76 % 7 = 6)

<table>
<thead>
<tr>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
| 2   | 40  
   |    |
| 3   | 48  
   |    |
| 4   | 5   
   |    |
| 5   | 55  
   |    |
| 6   | 47  
   |    |

Insert: 76 (76 % 7 = 6)

<table>
<thead>
<tr>
<th>0</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
| 2   | 76  
   |    |
| 3   | 40  
   |    |
| 4   | 48  
   |    |
| 5   | 5   
   |    |
| 6   | 55  
   |    |

Insert: 48 (76 % 7 = 6)
Another Quadratic Probing Example

Table Size = 7

Insert:
76 (76 % 7 = 6)
40 (40 % 7 = 5)
48 (48 % 7 = 6)
5 (  5 % 7 = 5)
55 (55 % 7 = 6)
47 (47 % 7 = 5)

Another Quadratic Probing Example

Table Size = 7

Insert:
76 (76 % 7 = 6)
40 (40 % 7 = 5)
48 (48 % 7 = 6)
5 (  5 % 7 = 5)
55 (55 % 7 = 6)
47 (47 % 7 = 5)

Another Quadratic Probing Example

Table Size = 7

Insert:
0
48
1
48
2
5
3
55
4
5
55
5
40
6
5
6
76

Another Quadratic Probing Example

Table Size = 7

Insert:
0
48
1
48
2
5
3
55
4
5
55
5
40
6
5
6
76

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering:
  no problem with keys initially hashing to the same neighborhood
- But it’s no help if keys initially hash to the same index
  - Called secondary clustering
- Can avoid secondary clustering with a probe function that depends on the key: double hashing...

From Bad News to Good News

- Bad news:
  - Quadratic probing can cycle through the same full indices, never
    terminating despite table not being full
- Good news:
  - If TableSize is prime and \( \lambda \leq \frac{1}{2} \), then quadratic probing will find an
    empty slot in at most TableSize/2 probes
  - So: If you keep \( \lambda \leq \frac{1}{2} \) and TableSize is prime, no need to detect
    cycles
  - Optional: Proof is available at
    http://courses.cs.washington.edu/courses/cse373/14au/uwnetid/quadraticProbingProof.txt

Double hashing

Idea:
- Given two good hash functions \( h \) and \( g \), it is very unlikely
  that for some key, \( h(\text{key}) = g(\text{key}) \)
- So make the probe function \( f(i) = h(i\cdot g(\text{key})) \)

Probe sequence:
- 0th probe: \( h(\text{key}) \mod \text{TableSize} \)
- 1st probe: \( h(\text{key}) + g(\text{key}) \mod \text{TableSize} \)
- 2nd probe: \( h(\text{key}) + 2\cdot g(\text{key}) \mod \text{TableSize} \)
- 3rd probe: \( h(\text{key}) + 3\cdot g(\text{key}) \mod \text{TableSize} \)
- ...
- \( i \)-th probe: \( h(\text{key}) + i\cdot g(\text{key}) \mod \text{TableSize} \)

Detail: Make sure \( g(\text{key}) \) cannot be 0
Double-hashing analysis

- Intuition: Because each probe is “jumping” by \( g(\text{key}) \) each time, we “leave the neighborhood” and “go different places from other initial collisions”

- But we could still have a problem like in quadratic probing where we are not “safe” (infinite loop despite room in table)
  - It is known that this cannot happen in at least one case:
    - \( h(\text{key}) = \text{key} \mod p \)
    - \( g(\text{key}) = q - (\text{key} \mod q) \)
    - \( 2 < q < p \)
    - \( p \) and \( q \) are prime

More double-hashing facts

- Assume “uniform hashing”
  - Means probability of \( g(\text{key1}) \mod p = g(\text{key2}) \mod p \) is \( 1/p \)
- Non-trivial facts we won’t prove:
  - Average # of probes given \( \lambda \) (in the limit as TableSize → ∞)
    - Unsuccessful search (intuitive): \( \frac{1}{1-\lambda} \)
    - Successful search (less intuitive): \( \frac{1}{\lambda} \log_e \left( \frac{1}{1-\lambda} \right) \)
  - Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

Rehashing

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything
- With chaining, we get to decide what “too full” means
  - Keep load factor reasonable (e.g., < 1)?
  - Consider average or max size of non-empty chains?
- For probing, half-full is a good rule of thumb
- New table size
  - Twice-as-big is a good idea, except that won’t be prime!
  - So go about twice-as-big
  - Can have a list of prime numbers in your code since you won’t grow more than 20-30 times

Hashtable Scenarios

- For each of the scenarios, answer the following questions:
  - Is a hashtable the best-suited data structure?
  - If so, what would be used at the keys? Values?
  - If not, what data structure would be best-suited?
  - What other assumptions, if any, about the scenario must you make to support your previous answers?

- Catalog of items (product id, name, price)
- Bookmarks in a web browser (favicon, URL, bookmark name)
- IT support requests (timestamp, ticket id, description)
- Character frequency analysis (character, # of appearances)
- Activation records for nested function calls (return addresses, local variables, etc.)