CSE 332: Data Structures & Parallelism
Lecture 11: More Hashing

Ruth Anderson
Autumn 2016
Today

- Dictionaries
  - Hashing
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

![Diagram of hash table process]

```
client | hash table library
E     | int
      | table-index
      | collision?
      | collision resolution
```

TableSize – 1

10/24/2016
Hashing Choices

1. Choose a Hash function
2. Choose Table Size
3. Choose a Collision Resolution Strategy from these:
   – Separate Chaining
   – Open Addressing
     • Linear Probing
     • Quadratic Probing
     • Double Hashing
   • Other issues to consider:
     – Deletion?
     – What to do when the hash table gets “too full”?
Open Addressing: Linear Probing

• Why not use up the empty space in the table?
• Store directly in the array cell (no linked list)
• How to deal with collisions?
• If $h(key)$ is already full,
  – try $(h(key) + 1) \% TableSize$. If full,
  – try $(h(key) + 2) \% TableSize$. If full,
  – try $(h(key) + 3) \% TableSize$. If full…

• Example: insert 38, 19, 8, 109, 10

<table>
<thead>
<tr>
<th></th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Open addressing**

Linear probing is *one example* of open addressing

In general, **open addressing** means resolving collisions by trying a sequence of other positions in the table.

Trying the *next* spot is called **probing**
- We just did *linear probing*:
  - \( i^{th} \) probe: \( (h(key) + i) \mod TableSize \)
- In general have some probe function \( f \) and:
  - \( i^{th} \) probe: \( (h(key) + f(i)) \mod TableSize \)

Open addressing does poorly with high load factor \( \lambda \)
- So want larger tables
- Too many probes means no more \( O(1) \)
Terminology

We and the book use the terms
  – “chaining” or “separate chaining”
  – “open addressing”

Very confusingly,
  – “open hashing” is a synonym for “chaining”
  – “closed hashing” is a synonym for “open addressing”
Open Addressing: Linear Probing

What about \textbf{find}? If value is in table? If not there? Worst case?

What about \textbf{delete}?

How does open addressing with linear probing compare to separate chaining?
Primary Clustering

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

- Tends to produce *clusters*, which lead to long probe sequences
- Called primary clustering
- Saw the start of a cluster in our linear probing example
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 $\text{TableSize} \rightarrow \infty$)
  - Unsuccessful search: $\frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right)$
  - 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)
Analysis in chart form

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes “large table” but point remains)

By comparison, separate chaining performance is linear in $\lambda$ and has no trouble with $\lambda > 1$
Open Addressing: Linear probing

\[(h(key) + f(i)) \mod \text{TableSize}\]

- For linear probing:
  \[f(i) = i\]

- So probe sequence is:
  - 0\textsuperscript{th} probe: \(h(key) \mod \text{TableSize}\)
  - 1\textsuperscript{st} probe: \((h(key) + 1) \mod \text{TableSize}\)
  - 2\textsuperscript{nd} probe: \((h(key) + 2) \mod \text{TableSize}\)
  - 3\textsuperscript{rd} probe: \((h(key) + 3) \mod \text{TableSize}\)
  - ...
  - i\textsuperscript{th} probe: \((h(key) + i) \mod \text{TableSize}\)
Open Addressing: Quadratic probing

• We can avoid primary clustering by changing the probe function...

\[(h(key) + f(i)) \mod \text{TableSize}\]

– For quadratic probing:

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

– So probe sequence is:

• 0\text{th} probe: \(h(key) \mod \text{TableSize}\)
• 1\text{st} probe: \((h(key) + 1) \mod \text{TableSize}\)
• 2\text{nd} probe: \((h(key) + 4) \mod \text{TableSize}\)
• 3\text{rd} probe: \((h(key) + 9) \mod \text{TableSize}\)
• ...
• \(i\text{th} \) probe: \((h(key) + i^2) \mod \text{TableSize}\)

• Intuition: Probes quickly “leave the neighborhood”
Quadratic Probing Example

\[ \text{ith probe: } (h(\text{key}) + i^2) \mod \text{TableSize} \]

TableSize = 10
Insert:
89
18
49
58
79
Another Quadratic Probing Example

TableSize = 7

Insert:

<table>
<thead>
<tr>
<th>Key</th>
<th>Position</th>
<th>Probe Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>47</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

ith probe: \( h(key) + i^2 \) \% TableSize
From bad news to good news

Bad News:
• After TableSize quadratic probes, we cycle through the same indices

Good News:
• If TableSize is prime and \( \lambda < \frac{1}{2} \), then quadratic probing will find an empty slot in at most TableSize/2 probes
• So: If you keep \( \lambda < \frac{1}{2} \) and TableSize is prime, no need to detect cycles

• Proof posted in lecture11.txt (slightly less detailed proof in textbook)
  – For prime \( T \) and \( 0 \leq i, j \leq T/2 \) where \( i \neq j \),
    \[
    (h(key) + i^2) \mod T \neq (h(key) + j^2) \mod T
    \]
    That is, if \( T \) is prime, the first \( T/2 \) quadratic probes map to different locations
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 $0 \leq i,j \leq \text{size}/2$ 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 $i \neq j$:

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

    $$\implies i^2 \mod \text{size} = j^2 \mod \text{size}$$

    $$\implies (i^2 - j^2) \mod \text{size} = 0$$

    $$\implies [(i + j)(i - j)] \mod \text{size} = 0$$

    BUT size does not divide $(i-j)$ or $(i+j)$

First size/2 probes will be distinct, and if less than half of table is full then after size/2 probes you will find one of those empty spots

How can $i+j = 0$ or $i+j = \text{size}$ when:

- $i \neq j$ and $0 \leq i,j \leq \text{size}/2$?

Similarly how can $i-j = 0$ or $i-j = \text{size}$?

Size would need to divide one of these
Clustering reconsidered

• Quadratic probing does not suffer from primary clustering: As we resolve collisions we are not merely growing “big blobs” by adding one more item to the end of a cluster, we are looking \( i^2 \) locations away, for the next possible spot.

• But quadratic probing does not help resolve collisions between keys that initially hash to the same index
  – Any 2 keys that initially hash to the same index will have the same series of moves after that looking for any empty spot
  – Called secondary clustering

• Can avoid secondary clustering with a probe function that depends on the key: double hashing…
Open Addressing: Double hashing

Idea: Given two good hash functions $h$ and $g$, it is very unlikely that for some key $key$, $h(key) == g(key)$

$$(h(key) + f(i)) \mod \text{TableSize}$$

– For double hashing:

$$f(i) = i \times g(key)$$

– So probe sequence is:

• 0\text{th} probe: $h(key) \mod \text{TableSize}$
• 1\text{st} probe: $(h(key) + g(key)) \mod \text{TableSize}$
• 2\text{nd} probe: $(h(key) + 2 \times g(key)) \mod \text{TableSize}$
• 3\text{rd} probe: $(h(key) + 3 \times g(key)) \mod \text{TableSize}$
• ...

• $i\text{th}$ probe: $(h(key) + i \times g(key)) \mod \text{TableSize}$

• Detail: Make sure $g(key)$ can’t be 0
Open Addressing: Double Hashing

T = 10 (TableSize)

Hash Functions:
- \( h(key) = key \mod T \)
- \( g(key) = 1 + ((key/T) \mod (T-1)) \)

ith probe: \((h(key) + i \times g(key)) \mod \text{TableSize}\)

Insert these values into the hash table in this order. Resolve any collisions with double hashing:
- 13
- 28
- 33
- 147
- 43
Double-hashing analysis

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

But, as in quadratic probing, we could still have a problem where we are not "safe" due to an infinite loop despite room in table
  – It is known that this cannot happen in at least one case:
    For primes $p$ and $q$ such that $2 < q < p$
    $$h(\text{key}) = \text{key} \% p$$
    $$g(\text{key}) = q - (\text{key} \% q)$$
More double-hashing facts

• Assume “uniform hashing”
  – Means probability of \( g(key_1) \% p = g(key_2) \% p \) is \( 1/p \)

• Non-trivial facts we won’t prove:
  Average # of probes given \( \lambda \) (in the limit as TableSize \( \rightarrow \infty \))
  – 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
Where are we?

- **Separate Chaining** is easy
  - `find`, `delete` proportional to load factor on average
  - `insert` can be constant if just push on front of list

- **Open addressing** uses probing, has clustering issues as table fills

Why use it:
- Less memory allocation?
  - Some run-time overhead for allocating linked list (or whatever) nodes; open addressing could be faster
  - Easier data representation?

- Now:
  - Growing the table when it gets too full (aka “rehashing”)
  - Relation between hashing/comparing and connection to Java
Rehashing

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything over

- With **separate 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 **open addressing**, half-full is a good rule of thumb

- New table size
  - Twice-as-big is a good idea, except, uhm, that won’t be prime!
  - So go *about* twice-as-big
  - Can have a list of prime numbers in your code since you probably won’t grow more than 20-30 times, and then calculate after that
More on rehashing

• What if we copy all data to the same indices in the new table?
  – Will not work; we calculated the index based on TableSize

• Go through table, do standard insert for each into new table
  – Iterate over old table: O(n)
  – n inserts / calls to the hash function: n \cdot O(1) = O(n)

• Is there some way to avoid all those hash function calls?
  – Space/time tradeoff: Could store $h(key)$ with each data item
  – Growing the table is still $O(n)$; saving $h(key)$ only helps by a constant factor
Hashing and comparing

• Our use of int key can lead to us overlooking a critical detail:
  – We initially hash $E$ to get a table index
  – While chaining or probing we need to determine if this is the $E$ that I am looking for. Just need equality testing.

• So a hash table needs a hash function and a equality testing
  – In the Java library each object has an equals method and a hashCode method

```java
class Object {
    boolean equals(Object o) {...}
    int hashCode() {...}
    ...
}
```
Equal objects must hash the same

• The Java library (and your project hash table) make a very important assumption that clients must satisfy…

• Object-oriented way of saying it:
  
  \[
  \text{If } a.\text{equals}(b), \text{ then we must require} \\
  a.\text{hashCode}() == b.\text{hashCode}() \\
  \]

• Function object way of saying it:
  
  \[
  \text{If } c.\text{compare}(a,b) == 0, \text{ then we must require} \\
  h.\text{hash}(a) == h.\text{hash}(b) \\
  \]

• If you ever override equals
  – You need to override hashCode also in a consistent way
  – See CoreJava book, Chapter 5 for other "gotchas" with equals
By the way: comparison has rules too

We have not emphasized important “rules” about comparison for:
  – All our dictionaries
  – Sorting (next major topic)

Comparison must impose a consistent, total ordering:

For all $a$, $b$, and $c$,
  – If $\text{compare}(a,b) < 0$, then $\text{compare}(b,a) > 0$
  – If $\text{compare}(a,b) == 0$, then $\text{compare}(b,a) == 0$
  – If $\text{compare}(a,b) < 0$ and $\text{compare}(b,c) < 0$, then $\text{compare}(a,c) < 0$
A Generally Good hashCode()

int result = 17; // start at a prime

foreach field f
    int fieldHashCode =
        boolean: (f ? 1 : 0)
        byte, char, short, int: (int) f
        long: (int) (f ^ (f >>> 32))
        float: Float.floatToIntBits(f)
        double: Double.doubleToLongBits(f), then above
        Object: object.hashCode()

    result = 31 * result + fieldHashCode;

return result;
Final word on hashing

- The hash table is one of the most important data structures
  - Efficient find, insert, and delete
  - Operations based on sorted order are not so efficient
  - Useful in many, many real-world applications
  - Popular topic for job interview questions
- Important to use a good hash function
  - Good distribution, Uses enough of key’s values
  - Not overly expensive to calculate (bit shifts good!)
- Important to keep hash table at a good size
  - Prime #
  - Preferable \( \lambda \) depends on type of table
- What we skipped: Perfect hashing, universal hash functions, hopscotch hashing, cuckoo hashing
- Side-comment: hash functions have uses beyond hash tables
  - Examples: Cryptography, check-sums