Outline

- Why Maps?
- HashMap: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- HashMap: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
Map Implementation: Chaining

- **Chaining method**
  - Use an array-based *Hashtable* as before
  - Each table element refers to a chain of \((key, value)\) entries
    - Usually an unordered, singly linked list

- **Depiction:**

```
(k2,v2) ----> [0] ----> (k2,v2)
(k1,v1) ----> [1] ----> null
(k3,v3) ----> [2] ----> null
(k3,v3) ----> [3] ----> null
[..] ----> [.] ----> (k1,v1)
```

Index calculation
Chaining: `put(K key, V value)`

- **Step 1: Index calculation:** `Math.abs(key.hashCode()) % table.length`
- **Step 2: Chain walk**
  - If entry is non-null, walk the chain
  - Insert if key is not found in chain, or replace when key is found. Return old value.

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>1</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tony&quot;</td>
<td>2612646</td>
<td>1</td>
</tr>
<tr>
<td>&quot;America&quot;</td>
<td>775550446</td>
<td>1</td>
</tr>
</tbody>
</table>
Chaining: `put(K key, V value)`

- **Step 1: Index calculation:**
  \[
  \text{Math.abs(key.hashCode())} \mod \text{table.length}
  \]

- **Step 2: Chain walk**
  
  - If entry is non-null, walk the chain
    
    - Insert if key is not found in chain, or replace when key is found. Return old value.

```
put("David","david@pugetsound.edu")
```

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>1</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tony&quot;</td>
<td>2612646</td>
<td>1</td>
</tr>
<tr>
<td>&quot;America&quot;</td>
<td>775550446</td>
<td>1</td>
</tr>
</tbody>
</table>

```java
null
[0] ("Adam","adam@pugetsound.edu")
[1] ("Brad","brad@pugetsound.edu")
[2] ("David","david@pugetsound.edu")
[3] null
[4] null
```
Chaining: `put(K key, V value)`

- **Step 1: Index calculation:** \[\text{Math.abs(key.hashCode()) \% table.length}\]

- **Step 2: Chain walk**
  - If entry is non-null, walk the chain
    - Insert if key is not found in chain, or replace when key is found. Return old value.

**Example:**

```
put("Tony", "tony@pugetsound.edu")
```

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>1</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tony&quot;</td>
<td>2612646</td>
<td>1</td>
</tr>
<tr>
<td>&quot;America&quot;</td>
<td>775550446</td>
<td>1</td>
</tr>
</tbody>
</table>

Table:

- [0]: null
- [1]: ("Adam", "adam@pugetsound.edu")
- [2]: ("Brad", "brad@pugetsound.edu")
- [3]: ("David", "david@pugetsound.edu")
- [4]: null

Null pointer indicated with "..."
Chaining: \texttt{put(K key, V value)}

- **Step 1: Index calculation:** \texttt{Math.abs(key.hashCode()) \% table.length}

- **Step 2: Chain walk**
  - If entry is non-null, walk the chain
    - Insert if key is not found in chain, or \textit{replace when key is found}. Return old value.

```plaintext
put("Adam","aasmith@pugetsound.edu")
```

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>1</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tony&quot;</td>
<td>2612646</td>
<td>1</td>
</tr>
<tr>
<td>&quot;America&quot;</td>
<td>775550446</td>
<td>1</td>
</tr>
</tbody>
</table>

```

null

("Tony", ...)

("Adam","aasmith@pugetsound.edu")

("Brad","brad@pugetsound.edu")

("David","david@pugetsound.edu")

null
Chaining: put(K key, V value)

- Step 1: Index calculation: \[ \text{Math.abs(key.hashCode()) \% table.length} \]
- Step 2: Chain walk
  - If entry is non-null, walk the chain
    - Insert if key is not found in chain, or replace when key is found. Return old value.

### Example: put("America","america@pugetsound.edu")

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>1</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tony&quot;</td>
<td>2612646</td>
<td>1</td>
</tr>
<tr>
<td>&quot;America&quot;</td>
<td>775550446</td>
<td>1</td>
</tr>
</tbody>
</table>

Collision!
Outline

- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
### Chaining: `put(K key, V value)`

- **Step 1: Index calculation:**
  \[ \text{Math.abs(key.hashCode()) % table.length} \]

- **Step 2: Chain walk**
  - If entry is null, return null
  - If entry is non-null, walk the chain
    - Return if entry found in chain, or return null if not

---

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>1</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tony&quot;</td>
<td>2612646</td>
<td>1</td>
</tr>
<tr>
<td>&quot;America&quot;</td>
<td>775550446</td>
<td>1</td>
</tr>
</tbody>
</table>

### Example

- `get("America")`

**Table:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td>(&quot;Adam&quot;, &quot;<a href="mailto:aasmith@pugetsound.edu">aasmith@pugetsound.edu</a>&quot;)</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>(&quot;Brad&quot;, &quot;<a href="mailto:brad@pugetsound.edu">brad@pugetsound.edu</a>&quot;)</td>
<td>(&quot;America&quot;, ..)</td>
</tr>
<tr>
<td>[3]</td>
<td>(&quot;America&quot;, ..)</td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td>null</td>
<td>(&quot;Tony&quot;, ..)</td>
</tr>
</tbody>
</table>

3 probes to find America
Chaining: `get(K key)`

- **Step 1: Index calculation:** \(\text{Math.abs(key.hashCode())} \mod\ \text{table.length}\)

- **Step 2: Chain walk**
  - If entry is null, return null
  - If entry is non-null, walk the chain
    - Return if entry found in chain, or return null if not

```
get("Bobert")
```

```
<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>1</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tony&quot;</td>
<td>2612646</td>
<td>1</td>
</tr>
<tr>
<td>&quot;America&quot;</td>
<td>775550446</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Bobert&quot;</td>
<td>1995055026</td>
<td>1</td>
</tr>
</tbody>
</table>
```

![Diagram with data points and hash codes]
Outline

- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
Performance of Chained Map

- What is the performance of a chained hashmap?

- The time complexity of chained maps, $T_{chain}$, can be approximated by the following equation (D. Knuth, 1973):

$$T_{chain} \approx 1 + \frac{L}{2}$$

where $L = \text{stored\_entries} / \text{hashtable\_size}$ is the **Load Factor**

  - *In a chained map, $L$ is between $[0,n]$ where $n = \text{number of entries}*
Load Factor $L$ is the average chain size, storing $n$ entries.

<table>
<thead>
<tr>
<th>Load Factor ($L$)</th>
<th>Number of Probes ($T_{\text{chain}}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>0.25</td>
<td>1.13</td>
</tr>
<tr>
<td>0.5</td>
<td>1.25</td>
</tr>
<tr>
<td>0.75</td>
<td>1.38</td>
</tr>
<tr>
<td>0.85</td>
<td>1.43</td>
</tr>
<tr>
<td>0.9</td>
<td>1.45</td>
</tr>
<tr>
<td>0.95</td>
<td>1.48</td>
</tr>
<tr>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>51</td>
</tr>
<tr>
<td>1000</td>
<td>501</td>
</tr>
<tr>
<td>$n$</td>
<td>$n/2$</td>
</tr>
</tbody>
</table>

O(1) when $L$ is low

O(n) when $L$ approaches $n$
Chaining Performance Summary

Pros:

• Like open-addressing, $O(1)$-time insert/search/delete if $L$ is low
• Generally better performance than open addressing
  - Especially for higher load factors ($L > 0.75$)

Cons:

• Generally requires much more space than open-addressing!
• Example
  - Suppose a key-value entry is 4 bytes.
  - We want ~2.5 probes per operation for 60,000 entries. Compare the space usage of open-addressing vs. chaining.
Chained Map Summary

Things to consider

• Why use a singly linked list for each chain?
  - Why not ArrayList? Why not (doubly) LinkedList?

• Need to rehash occasionally to shorten average chain length \( (L) \)
  - Just not as critical to performance as in Open Addressing
  - Could monitor longest or average chain length, rehash when appropriate

For what it's worth:

• Java's HashMap\(<K,V>\) class is a chained map
Outline

- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
Maps Summary

Don't use if:

- You need a specific ordering of the items
- A simple array will work for the problem
- Space is at a premium on the device you're programming for
- You need to return values for a range of keys (why?)
  - "Find all keys between 20 and 40"

Use if:

- You need to index by something besides int
- You usually just need to perform exact-match searches
- You have space to spare
Data Structure Summary

- Important: You must know how to choose structures based on your problem.
  - Evil Hangman -- use map, not lists!
  - Dijkstra's -- use heaps (priority queue), not BSTs!

- *Know the caveats.* Each structure has strengths and weaknesses!

<table>
<thead>
<tr>
<th>Operations</th>
<th>Lists</th>
<th>BST</th>
<th>HashMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, val)</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$ avg/worst</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>get(key)</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$ avg/worst</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove(key)</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$ avg/worst</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>others...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reminders:

- Hwk 7 (Evil Hangman) due next wednesday
  - Use of HashMaps (in multiple places) makes your life easier
- Math/CS winter party
  - Wednesday 12/11, 5pm @ TH 391

Last time... Can we achieve O(1)-time search, remove, and insert?

- If we're willing to trade space for time, sure!
- HashMaps
  - Object's hashCode() method
  - First implementation: open-addressing (linear probing)
Reminders:

- Hwk 7 (Evil Hangman) due next Wednesday
- Math/CS winter party
  - Wednesday 12/11, 5pm @ TH 391

Last time.. HashMaps

- Performance largely dependent on load factor
  - L --> 0: no collisions yay. L --> 1: collisions galore, more probing.
  - To keep L low, we rehash periodically.

Today:

- Can we do better than Open Addressing?
- Start sorting algorithms