Your hash bucket was tasty
Recap: So Far

• Current Design: Hash Table with Chaining
  - Array of Buckets
  - Each bucket is the head of a linked list (a “chain”)
Recap: apply(x)

- Expected Cost
  - Find the bucket: $O(c_{hash})$
  - Find the record: $O(\alpha \cdot c_{equality})$
  - **Total**: $O(c_{hash} + \alpha \cdot c_{equality}) \approx O(1 + 1) = O(1)$

- Worst-Case Cost
  - Find the record: $O(n \cdot c_{equality})$
  - **Total**: $O(c_{hash} + n \cdot c_{equality}) \approx O(1 + n) = O(n)$
Recap: remove(x)

- **Expected Cost**
  - Find the bucket: $O(c_{\text{hash}})$
  - Find the record: $O(\alpha \cdot c_{\text{equality}})$
  - Remove from linked-list: $O(1)$
  - **Total**: $O(c_{\text{hash}} + \alpha \cdot c_{\text{equality}} + 1) \approx O(1 + 1 + 1) = O(1)$

- **Worst-Case Cost**
  - Find the record: $O(n \cdot c_{\text{equality}})$
  - **Total**: $O(c_{\text{hash}} + n \cdot c_{\text{equality}} + 1) \approx O(1 + n + 1) = O(n)$
Recap: \( \text{insert}(x) \)

- **Expected Cost**
  - Find the bucket: \( O(c_{\text{hash}}) \)
  - Remove the key, if present: \( O(\alpha \cdot c_{\text{equality}} + 1) \)
  - Prepend to linked-list: \( O(1) \)
  - Rehash: \( O(n \cdot c_{\text{hash}} + N) \); amortized: \( O(1) \)
  - **Total**: \( O(c_{\text{hash}} + \alpha \cdot c_{\text{equality}} + 1) \approx O(1 + 1 + 2) = O(1) \)

- **Worst-Case Cost (amortized)**
  - Remove the key, if present: \( O(n \cdot c_{\text{equality}} + 1) \)
  - **Total**: \( O(c_{\text{hash}} + n \cdot c_{\text{equality}} + 1 + 1) \approx O(1 + n + 2) = O(n) \)
Variations

• **Hash Table with Chaining**
  - ... but re-use empty hash buckets instead of chaining
  • **Hash Table with Open Addressing**
  • **Cuckoo Hashing** (Double Hashing)
  - ... but avoid bursty rehashing costs
  • **Dynamic Hashing**
  - ... but avoid $O(N)$ iteration cost
  • **Linked Hash Table**
Chaining

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{hash}(A) = 1 \\
\text{hash}(B) = 2 \\
\text{hash}(C) = 2 \\
\text{hash}(D) = 4 \\
\text{hash}(E) = 3
\]
Open Addressing

hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

"Cascade" collisions to the next available spot
Open Addressing

apply(A)

"Cascade" collisions to the next available spot

hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3
Open Addressing

apply(C)

"Cascade" collisions to the next available spot

hash(A) = 1  
hash(B) = 2  
hash(C) = 2  
hash(D) = 4  
hash(E) = 3
Open Addressing

apply(E)

hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

“Cascade” collisions to the next available spot
Open Addressing

- **insert(X)**
  - While bucket hash(X)+i %N is occupied, i = i + 1
  - Insert at bucket hash(X)+i %N

- **apply(X)**
  - While bucket hash(X)+i %N is occupied
    - If the element at bucket hash(X)+i %N is X, return it
    - Otherwise i = i + 1
  - Element not found
Open Addressing

• remove(X)
  - While bucket hash(X)+i is occupied
    • If the element at bucket hash(X)+i is X, remove it
    • Otherwise i = i + 1 %N

What about elements that were cascaded?
Removals Under Open Addressing

- Check each element in a contiguous block, starting at hash(X)
  - Move elements up
  - Don’t move any element Y ahead of hash(Y)
Open Addressing

- **Linear Probing**: Offset to hash(X) + ci for some constant c
- **Quadratic Probing**: Offset to hash(X) + ci^2 for some constant c
- Follow Probing Strategy to find the next bucket

- Runtime Costs
  - Chaining: Dominated by following chain
  - Open Addressing: Dominated by probing
- With a low enough α_{max}, operations still O(1)
Cuckoo Hashing

- Dynamic Hashing can have arbitrarily long cascade chains
  - Can we reduce the chance of a cascade chain for some operations?
Cuckoo Hashing

- Use two hash functions: hash\(_1\), hash\(_2\)
  - Each record is stored at one of the two
- insert(x)
  - If both buckets are available: pick at random
  - If one bucket is available: insert record there
  - If neither bucket is available, pick one at random
    - “Displace” the record there, move it to the other bucket
    - Repeat displacement until an empty bucket is found
Cuckoo Hashing

\[
\begin{align*}
\text{hash}_1(A) &= 1 \\
\text{hash}_1(B) &= 2 \\
\text{hash}_1(C) &= 2 \text{ !} \\
\text{hash}_1(D) &= 4 \\
\text{hash}_1(E) &= 3 \\
\text{hash}_2(A) &= 3 \\
\text{hash}_2(B) &= 4 \\
\text{hash}_2(C) &= 1 \text{ !} \\
\text{hash}_2(D) &= 6 \\
\text{hash}_2(E) &= 3
\end{align*}
\]
Cuckoo Hashing

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>3</th>
<th>B</th>
<th>5</th>
<th>D</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{hash}_1(A)$</td>
<td>$= 1$</td>
<td>$\text{hash}_1(B)$</td>
<td>$= 2$</td>
<td>$\text{hash}_1(C)$</td>
<td>$= 2$</td>
<td>$\text{hash}_1(D)$</td>
<td>$= 4$</td>
</tr>
<tr>
<td>$\text{hash}_2(A)$</td>
<td>$= 3$</td>
<td>$\text{hash}_2(B)$</td>
<td>$= 4$</td>
<td>$\text{hash}_2(C)$</td>
<td>$= 1$</td>
<td>$\text{hash}_2(D)$</td>
<td>$= 6$</td>
</tr>
</tbody>
</table>
Cuckoo Hashing

\[
\begin{align*}
\text{hash}_1(A) &= 1 \\
\text{hash}_1(B) &= 2 \\
\text{hash}_1(C) &= 2 \\
\text{hash}_1(D) &= 4 \\
\text{hash}_1(E) &= 1 \\
\text{hash}_2(A) &= 3 \\
\text{hash}_2(B) &= 4 \\
\text{hash}_2(C) &= 1 \\
\text{hash}_2(D) &= 6 \\
\text{hash}_2(E) &= 4 
\end{align*}
\]
Cuckoo Hashing

apply(x) and remove(x) is guaranteed $O(1)$
insert(x) is expected $O(1)$ if $\alpha$ is low enough
Dynamic Hashing

- Rehash is expensive!
  - Amortized cost of rehash is still $O(1)$
  - ... but every so often everything grinds to a halt!
Dynamic Hashing

- Contrast $h(x) \% 4$ with $h(x) \% 8$
  
  - e.g. $h(x) = 7069$; $h(x) \% 8 = 5$
- If we rehash from $h(x) \% N$ to $h(x) \% 2N$ either:
  
  - $h(x) \% 2N = h(x) \% N$
    
  or
  
  - $h(x) \% 2N = (h(x) \% N) + N$
- **Idea**: Only rehash “full” buckets
  
  - An element $x$ can be located at any of the following buckets: $h(x) \% N$ or $h(x) \% 2N$ or $h(x) \% 4N$ or ...
Dynamic Hashing

- $\text{hash}(A) = 1$
- $\text{hash}(B) = 6$
- $\text{hash}(C) = 3$
- $\text{hash}(D) = 4$
- $\text{hash}(E) = 9$
- $\text{hash}(F) = 7$

- $\text{insert}(x)$ is always $O(1)$
- $\text{apply}(x), \text{remove}(x)$ are $O(\log(n))$
Dynamic Hashing

- Keep log(n) levels
  - Each level i contains hash buckets for $h(x) \% 2^i \cdot N$
  - Any record will be stored at exactly one level
    - When a level fills up, split its records at the next level
    - When a level empties out, merge with its counterpart
- Keep an array of $2^i \cdot N$ entries
  - Indicate which level $h(x) \% 2^i \cdot N$ is located at
Linked Hash Table

- Iteration over Hash Table is $O(N + n)$
  - Can be much slower than $O(n)$
- **Idea**: Connect entries together in a Doubly Linked List
Linked Hash Table

- **Athos**
- **B**
- **C**
- **D**
- **...**
- **Porthos**
- **...**
- **Y**
- **Z**

Head: Athos
Tail: Porthos
Linked Hash Table

- **Athos**
- **B**
- **C**
- **D**
- **...**
- **Porthos**
- **...**
- **Y**
- **Z**

The diagram shows a linked hash table with nodes for Athos, B, C, D, ..., Porthos, ..., Y, and Z. The head and tail nodes are indicated, with Athos and Porthos highlighted in red.
Linked Hash Table

head

Athos

B

C

D

...

Porthos

...

Y

Z

Aramis

∅ ∅

tail

∅ ∅
Linked Hash Table

- O(n) Iteration
- apply(x)
  - O(1) increase in cost
- insert(x)
  - O(1) increase in cost
- remove(x)
  - O(1) increase in cost