What if $O(\log_m n)$ is not fast enough?

Do you feel lucky?
A Hash Table based Dictionary

Client Code:

```csharp
1  Dictionary<KeyType, ValueType> d;
2  d[k] = v;
```

A **Hash Table** consists of three things:

1. A hash function, f(k)

2. An array

3. Something to handle chaos when it occurs!
A Perfect Hash Function

Angrave, CS 241
(Beckman, CS 421)
(Challen, CS 125)
(Davis, CS 101)
(Evans, CS 126)
(Fagen-Ulmschneider, CS 107)
(Gunter, CS 422)
(Herman, CS 233)
Hash Function

Our hash function consists of two parts:

• A hash:

• A compression:

Choosing a good hash function is tricky...
• Don’t create your own (yet*)
• Very smart people have created very bad hash functions
Hash Function

Characteristics of a good hash function:

1. Computation Time:

2. Deterministic:

3. Satisfy the SUHA:
General Purpose Hash Function

Easy to create if: $|\text{KeySpace}| \sim N$
General Purpose Hash Function

Keyspaces

Easy to create if: $|\text{KeySpace}| \sim N$

Difficult to Create:
General Purpose Hash Function

Easy to create if: $|KeySpace| \sim N$

Difficult to Create:
Hash Function

In CS 225, we focus on general purpose hash functions.

Other hash functions exists with different properties (eg: cryptographic hash functions)
Collision Handling: Separate Chaining

\[ S = \{ 16, 8, 4, 13, 29, 11, 22 \} \quad |S| = n \]
\[ h(k) = k \% 7 \quad |Array| = N \]

Worst Case | SUHA
---|---
Insert | 
Remove/Find |
Collision Handling: Probe-based Hashing

\[ S = \{ 16, 8, 4, 13, 29, 11, 22 \} \quad |S| = n \]
\[ h(k) = k \% 7 \quad |Array| = N \]
Collision Handling: Linear Probing

$S = \{16, 8, 4, 13, 29, 11, 22\}$  \hspace{1cm} |S| = n

$h(k) = k \% 7$  \hspace{1cm} |Array| = N

(Example of closed hashing)

Try $h(k) = (k + 0) \% 7$, if full...
Try $h(k) = (k + 1) \% 7$, if full...
Try $h(k) = (k + 2) \% 7$, if full...
Try ...

<table>
<thead>
<tr>
<th></th>
<th>Worst Case</th>
<th>SUHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove/Find</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Problem w/ Linear Probing

Primary clustering:

Description:

Remedy:
Collision Handling: Double hashing

\[ S = \{ 16, 8, 4, 13, 29, 11, 22 \} \quad |S| = n \]
\[ h(k) = k \mod 7 \quad |\text{Array}| = N \]

Try \( h(k) = (k + 0 \times h_2(k)) \mod 7 \), if full...
Try \( h(k) = (k + 1 \times h_2(k)) \mod 7 \), if full...
Try \( h(k) = (k + 2 \times h_2(k)) \mod 7 \), if full...
Try ...

\[ h(k, i) = (h_1(k) + i \times h_2(k)) \mod 7 \]
Running Times
The expected number of probes for find(key) under SUHA

Linear Probing:
- Successful: \( \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right) \)
- Unsuccessful: \( \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)^2 \)

Double Hashing:
- Successful: \( \frac{1}{\alpha} \ln \left( \frac{1}{1-\alpha} \right) \)
- Unsuccessful: \( \frac{1}{1-\alpha} \)

Separate Chaining:
- Successful: \( 1 + \frac{\alpha}{2} \)
- Unsuccessful: \( 1 + \alpha \)

(Don’t memorize these equations, no need.)

Instead, observe:
- As \( \alpha \) increases:
  - If \( \alpha \) is constant:
Running Times

The expected number of probes for find(key) under SUHA

**Linear Probing:**
- **Successful:** $\frac{1}{2}(1 + 1/(1-\alpha))$
- **Unsuccessful:** $\frac{1}{2}(1 + 1/(1-\alpha))^2$

**Double Hashing:**
- **Successful:** $1/\alpha \times \ln(1/(1-\alpha))$
- **Unsuccessful:** $1/(1-\alpha)$
ReHashing

What if the array fills?
Which collision resolution strategy is better?

• Big Records:

• Structure Speed:

What structure do hash tables replace?

What constraint exists on hashing that doesn’t exist with BSTs?

Why talk about BSTs at all?
## Running Times

<table>
<thead>
<tr>
<th></th>
<th>Hash Table</th>
<th>AVL</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Find</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Amortized:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Worst Case:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Insert</strong></td>
<td>Amortized:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Worst Case:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Storage Space</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
std data structures

std::map
std data structures

**std::map**
- ::operator[]
- ::insert
- ::erase

- ::lower_bound(key) ➔ Iterator to first element ≤ key
- ::upper_bound(key) ➔ Iterator to first element > key
std data structures

`std::unordered_map`
  `.operator[]`
  `.insert`
  `.erase`

- `.lower_bound(key)` ➔ Iterator to first element ≤ key
- `.upper_bound(key)` ➔ Iterator to first element > key
std data structures

std::unordered_map
  ::operator[]
  ::insert
  ::erase

  ::lower_bound(key)  ➔  Iterator to first element ≤ key
  ::upper_bound(key)  ➔  Iterator to first element > key

  ::load_factor()
  ::max_load_factor(ml)  ➔  Sets the max load factor