Data Structures and Algorithms
Hashing 3

CS 225
G Carl Evans

April 21, 2023

Department of Computer Science
A Hash Table based Dictionary

**Client Code:**

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

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

1. A hash function

2. A data storage structure

3. A method of addressing *hash collisions*
Resizing a hash table

How do we resize?

\[ h(k, i) = \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>
# 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><strong>Insert</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Storage Space</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hash Function

Characteristics of a good hash function:

1. Computation Time:

2. Deterministic:

3. ...
Simple Uniform Hashing Assumption

Given table of size $m$, a simple uniform hash, $h$, implies

\[ \forall k_1, k_2 \in U \text{ where } k_1 \neq k_2, \ Pr(h[k_1] = h[k_2]) = \frac{1}{m} \]

Uniform:

Independent:
Separate Chaining Under SUHA

Given table of size $m$ and $n$ inserted objects

**Claim:** Under SUHA, expected length of chain is $\frac{n}{m}$
Running Times  *(Don’t memorize these equations, no need.)*

*(Expectation under SUHA)*

**Open Hashing:**

insert: ____________.

find/ remove: ___________.

**Closed Hashing:**

insert: ____________.

find/ remove: ___________.
Running Times  *(Don’t memorize these equations, no need.)*

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

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

**Double Hashing:**
- Successful: $\frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha})$
- Unsuccessful: $\frac{1}{1-\alpha}$

**Separate Chaining:**
- Successful: $1 + \frac{\alpha}{2}$
- Unsuccessful: $1 + \alpha$

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: \( \frac{1}{\alpha} \times \ln(1/(1-\alpha)) \)
- Unsuccessful: \( 1/(1-\alpha) \)

When do we resize?
Which collision resolution strategy is better?

- Big Records:

- Structure Speed:

What structure do hash tables implement?

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>Expectation*:</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>Expectation*:</td>
<td>Worst Case:</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`
::operator[]
::insert
::erase

::lower_bound(key) \rightarrow \text{Iterator to first element } \leq \text{key}
::upper_bound(key) \rightarrow \text{Iterator to first element } > \text{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
Hashing in the real world

Even under SUHA, our estimates are *in expectation*.
Hash Table

Worst-Case behavior is bad — but what about randomness?

1) **Fix** $h$, our hash, and assume it is good for *all keys*:

2) Create a *universal hash function family*:
Hash Function (Division Method or Identity Hash)

Hash of form: $h(k) = k \% m$
Hash Function (Mid-Square Method)

Hash of form: $h(k) = (k \times k)$ and take $b$ middle bits where $m = 2^b$
Hash Function (Multiplication Method)

Hash of form: \( h(k) = \lfloor m(\text{remain}(kA)) \rfloor, \ 0 \leq A \leq 1 \)
Hash Function (Universal Hash Family)

Pick a random $h \in H$ s.t. $\forall k_1, k_2 \in U$, $Pr(h[k_1] = h[k_2]) \leq \frac{1}{m}$
Hash Function (Universal Hash Family)

Hash of form: $h_{ab}(k) = ((ak + b)\%p)\%m$, $a, b \in Z_p^*, Z_p$

$\forall k_1 \neq k_2, \ Pr_{a,b}(h_{ab}[k_1] = h_{ab}[k_2]) \leq \frac{1}{m}$