CS 225
Data Structures

April 7 – Hashing
Brad Solomon
Team Contract and Proposal Due April 9th

Team Contract:

Be sure to ‘sign’ electronically.

Non-participants may be removed from groups!

Project Proposal:

One of your three algorithms should be completed by mid-project check-in.
Learning Objectives

• Motivate and formally define a hash table

• Discuss what a ‘good’ hash function looks like

• Identify the key weakness of a hash table

• Introduce strategies to “correct” this weakness
Dictionary ADT

```cpp
#pragma once

#include <unordered_map>

template<typename K, typename V>
class Dictionary {
public:
    // Find key and return value
    V find (const &k key) const;
    // Insert to dict
    void insert (const &k key, const &v value);
    // Remove
    void remove (const &k key);

private:
    // Traversal using iterators
};

#endif
```
Summary of Balanced BST

Pros:
- Running Time: $O(\log n)$
- Improvement Over: lists and arrays as dictionaries
- Great for specific applications:
  - approximate find
  - range find
What if $O(\log n)$ isn’t good enough?
What if $O(\log n)$ isn’t good enough?
Mr and Mrs Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal, thank you very much. They were the last people you’d expect to be involved in anything strange or mysterious, because they just didn’t hold with such nonsense.

Mr Dursley was the director of a firm called Grunnings, which made drills. He was a big, beefy man with hardly any neck, although he did have a very large moustache. Mrs Dursley was thin and blonde and had nearly twice the usual amount of neck, which came in very useful as she spent so much of her time craning over garden fences, spying on the neighbours. The Dursleys had a small son called Dudley and in their opinion there was no finer boy anywhere.

The Dursleys had everything they wanted, but they also had a secret, and their greatest fear was that somebody would discover it. They didn’t think they could bear it if anyone found out about the Potters.
A Hash Table consists of three things:
1.
2.
3.

Client Code:

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

Maps a **keyspace**, a (mathematical) description of the keys for a set of data, to a set of integers.
Hash Function

(Angrave, CS 241)
(Beckman, CS 421)
(Challon, CS 125)
(Davis, CS 101)
(Evans, CS 225)
(Fagen-Ulmschneider, CS 107)
(Gunter, CS 422)
(Herman, CS 233)
Hash Function

(Angrave, CS 241)
(Beckman, CS 421)
(Challon, CS 125)
(Davis, CS 101)
(Evans, CS 225)
(Fagen-Ulmschneider, CS 107)
(Gunter, CS 422)
(Herman, CS 233)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angrave</td>
<td>241</td>
</tr>
<tr>
<td>Beckman</td>
<td>421</td>
</tr>
<tr>
<td>Challon</td>
<td>125</td>
</tr>
<tr>
<td>Davis</td>
<td>101</td>
</tr>
<tr>
<td>Evans</td>
<td>225</td>
</tr>
<tr>
<td>Fagen-Ulmschneider</td>
<td>107</td>
</tr>
<tr>
<td>Gunter</td>
<td>422</td>
</tr>
<tr>
<td>Herman</td>
<td>233</td>
</tr>
</tbody>
</table>

Hash function: \((\text{key}[0] - 'A')\)
Hash Function

A hash function *must* be:

- **Deterministic:**

- **Efficient:**

- **Defined for a certain size table:**
General Hash Function

An $O(1)$ deterministic operation that maps all keys in a universe $U$ to a defined range of integers $[0, \ldots, m - 1]$

- A hash:
- A compression:

Choosing a good hash function is tricky...
- Don’t create your own (yet*)
Hash Function

\[ h(k) = (k \cdot firstName[0] + k \cdot lastName[0]) \mod m \]

\[ h(k) = (rand() \times k \cdot numPages) \mod m \]

\[ h(k) = (Order \cdot insert [Order \cdot seen]) \mod m \]
Hash Function
Hash Function

Author Name

Hash Function

‘J’ + ‘R’ = 28

Harry Potter

25
26
27
28
29

...
Hash Function

Author Name

Hash Function

‘R’ + ‘L’ = 25

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 25 | Goosebumps
| 26 | Ø
| 27 | Ø
| 28 | Harry Potter
| 29 | Ø
|   |   |   |
Hash Function

Aardvarks
Anonymous
By Jim Realman

Author Name Hash Function

'J' + 'R' = 28

<table>
<thead>
<tr>
<th>25</th>
<th>Goosebumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>∅</td>
</tr>
<tr>
<td>27</td>
<td>∅</td>
</tr>
<tr>
<td>28</td>
<td>Harry Potter</td>
</tr>
<tr>
<td>29</td>
<td>∅</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Hash Collision

A **hash collision** occurs when multiple unique keys hash to the same value

J.K Rowling = 28!

Jim Realman = 28!

<table>
<thead>
<tr>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ø</td>
<td>Ø</td>
<td>???</td>
<td>Ø</td>
<td>...</td>
</tr>
</tbody>
</table>

---

Aardvarks
Anonymous
By Jim Realman

Harry Potter
AND THE SORCERER'S STONE

The Extraordinary Hairy Beast
Scholastic
Perfect Hashing

If $m \geq S$, we can write a perfect hash with no collisions

$m$ elements

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S$, a finite Keyspace
General Purpose Hashing

In CS 225, we want our hash functions to work *in general*.
General Purpose Hashing

If $m < U$, there must be at least one hash collision.
General Purpose Hashing

By fixing $h$, we open ourselves up to adversarial attacks.

$U$, Universe of Keys

$m$ elements

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Image by Matthew Loffhagen
A Hash Table based Dictionary

**Client Code:**

```
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*
Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

- Open Hashing:

- Closed Hashing:
Open Hashing

In an **open hashing** scheme, key-value pairs are stored externally (for example as a linked list).
A hash collision in an open hashing scheme can be resolved by ______________________. This is called separate chaining.
Insertion (Separate Chaining)

_\text{insert}(\text{"Bob"})\_ 
_\text{insert}(\text{"Anna"})_ 

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>B+</td>
<td>2</td>
</tr>
<tr>
<td>Anna</td>
<td>A-</td>
<td>4</td>
</tr>
<tr>
<td>Alice</td>
<td>A+</td>
<td>4</td>
</tr>
<tr>
<td>Betty</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>Brett</td>
<td>A-</td>
<td>2</td>
</tr>
<tr>
<td>Greg</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>Sue</td>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>Ali</td>
<td>B+</td>
<td>4</td>
</tr>
<tr>
<td>Laura</td>
<td>A</td>
<td>7</td>
</tr>
<tr>
<td>Lily</td>
<td>B+</td>
<td>7</td>
</tr>
</tbody>
</table>
### Insertion (Separate Chaining)

```java
_insert(“Alice”)
```

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>B+</td>
<td>2</td>
</tr>
<tr>
<td>Anna</td>
<td>A-</td>
<td>4</td>
</tr>
<tr>
<td><strong>Alice</strong></td>
<td><strong>A+</strong></td>
<td><strong>4</strong></td>
</tr>
<tr>
<td>Betty</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>Brett</td>
<td>A-</td>
<td>2</td>
</tr>
<tr>
<td>Greg</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>Sue</td>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>Ali</td>
<td>B+</td>
<td>4</td>
</tr>
<tr>
<td>Laura</td>
<td>A</td>
<td>7</td>
</tr>
<tr>
<td>Lily</td>
<td>B+</td>
<td>7</td>
</tr>
</tbody>
</table>
Insertion (Separate Chaining)

Where does Alice end up relative to Anna in the chain?

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>B+</td>
<td>2</td>
</tr>
<tr>
<td>Anna</td>
<td>A-</td>
<td>4</td>
</tr>
<tr>
<td>Alice</td>
<td>A+</td>
<td>4</td>
</tr>
<tr>
<td>Betty</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>Brett</td>
<td>A-</td>
<td>2</td>
</tr>
<tr>
<td>Greg</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>Sue</td>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>Ali</td>
<td>B+</td>
<td>4</td>
</tr>
<tr>
<td>Laura</td>
<td>A</td>
<td>7</td>
</tr>
<tr>
<td>Lily</td>
<td>B+</td>
<td>7</td>
</tr>
</tbody>
</table>
## Insertion (Separate Chaining)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>B+</td>
<td>2</td>
</tr>
<tr>
<td>Anna</td>
<td>A-</td>
<td>4</td>
</tr>
<tr>
<td>Alice</td>
<td>A+</td>
<td>4</td>
</tr>
<tr>
<td>Betty</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>Brett</td>
<td>A-</td>
<td>2</td>
</tr>
<tr>
<td>Greg</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>Sue</td>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>Ali</td>
<td>B+</td>
<td>4</td>
</tr>
<tr>
<td>Laura</td>
<td>A</td>
<td>7</td>
</tr>
<tr>
<td>Lily</td>
<td>B+</td>
<td>7</td>
</tr>
</tbody>
</table>

### Insertion Visualization

- **Step 0**: Initial empty hash table.
- **Step 1**: Insert Bob (B+, 2) into hash table.
- **Step 2**: Insert Anna (A-, 4) into hash table.
- **Step 3**: Insert Alice (A+, 4) into hash table.
- **Step 4**: Insert Betty (B, 2) into hash table.
- **Step 5**: Insert Brett (A-, 2) into hash table.
- **Step 6**: Insert Sue (B, 7) into hash table.
- **Step 7**: Insert Ali (B+, 4) into hash table.
- **Step 8**: Insert Laura (A, 7) into hash table.
- **Step 9**: Insert Lily (B+, 7) into hash table.
- **Step 10**: Final hash table with all elements inserted.

### Visual Representation

- Insert Bob, Anna, Alice, Betty, Brett, Sue, Ali, Laura, Lily into hash table.
- Each element is added to its corresponding bucket in the hash table.
## Insertion (Separate Chaining)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>B+</td>
<td>2</td>
</tr>
<tr>
<td>Anna</td>
<td>A-</td>
<td>4</td>
</tr>
<tr>
<td>Alice</td>
<td>A+</td>
<td>4</td>
</tr>
<tr>
<td>Betty</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>Brett</td>
<td>A-</td>
<td>2</td>
</tr>
<tr>
<td>Greg</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>Sue</td>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>Ali</td>
<td>B+</td>
<td>4</td>
</tr>
<tr>
<td>Laura</td>
<td>A</td>
<td>7</td>
</tr>
<tr>
<td>Lily</td>
<td>B+</td>
<td>7</td>
</tr>
</tbody>
</table>

```
0 1 2 3 4 5 6 7 8 9 10
∅ 2 3 ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅
```

- Greg: A
- Brett: A-
- Betty: B
- Ali: B+
- Alice: A+
- Anna: A-
- Sue: B
- Laura: A
- Lily: B+

---

- Bob: B+
- Anna: A-
- Alice: A+
- Begy: B
- Breg: A-
- Sue: B
- Ali: B+
- Laura: A
- Lily: B+
Find (Separate Chaining)

```
_key = _find("Sue")
```

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>7</td>
</tr>
</tbody>
</table>
Remove (Separate Chaining) 

\[ _\text{remove}(\text{“Betty”}) \]

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Betty</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10 \\
\end{array} \]

- **Brett**
  - A-

- **Betty**
  - B

- **Bob**
  - B+
For hash table of size $m$ and $n$ elements:

find runs in: __________.

insert runs in: __________.

remove runs in: __________.
Hash Table

Two ways forward:

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

2) Create a *universal hash function family*:
Simple Uniform Hashing Assumption

Given a 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

Under SUHA, a hash table of size $m$ and $n$ elements:

Expected length of chain is ______________.
Separate Chaining Under SUHA

Under SUHA, a hash table of size \( m \) and \( n \) elements:

- find runs in: \( \)\hspace{1cm} \( \)\hspace{1cm}.
- insert runs in: \( \)\hspace{1cm} \( \)\hspace{1cm}.
- remove runs in: \( \)\hspace{1cm} \( \)\hspace{1cm}.
Separate Chaining Under SUHA

Pros:

Cons: