CSCI 261
Computer Science II
Outline

- Why Maps?
- Map: List Implementation
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Performance
- Conclusion
Motivation for Maps

- Why Maps?
  - Because computer scientists are bad at giving up
  - We keep exploring for better, faster structures.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Lists</th>
<th>BST</th>
<th>Is this attainable?</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>
</tbody>
</table>

- Maps will get us there
  - But at what cost...? *(Play eerie foreshadowing music)*
A map is a set of ordered \((key, value)\) entries

- (key, value) pairs are unique
  - Enforced by: keys must be unique in a map; values are not
  - Many-to-one mapping: One value may be mapped by multiple keys

Maps Strings (key) to Integers (value):

```java
Map<String, Integer> myMap = new HashMap<>();
```

Maps Strings to Strings:

```java
Map<String, String> myContacts = new HashMap<>();
```
Maps

- To insert an entry, use `put(key, value)`
  - There's no `add` method

```java
Map<String, String> myContacts = new HashMap<>();
myContacts.put("brad", "brad@pugetsound.edu");
myContacts.put("adam", "adam@pugetsound.edu");
myContacts.put("tony", "tony@pugetsound.edu");
myContacts.put("david", "david@pugetsound.edu");
```
No duplicate keys allowed, so `put` replaces the existing value if a key exists.

```java
Map<String,String> myContacts = new HashMap<>();
myContacts.put("brad", "brad@pugetsound.edu");
myContacts.put("adam", "adam@pugetsound.edu");
myContacts.put("tony", "tony@pugetsound.edu");
myContacts.put("david", "david@gmail.com");
```
Duplicate values are allowed.

```java
Map<String, String> myContacts = new HashMap<>();
myContacts.put("brad", "brad@pugetsound.edu");
myContacts.put("adam", "adam@pugetsound.edu");
myContacts.put("tony", "tony@pugetsound.edu");
myContacts.put("david", "david@pugetsound.edu");
myContacts.put("david", "david@gmail.com");
myContacts.put("david2", "david@gmail.com");
```
## Java's `Map<K, V>` Interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public V get(K key)</code></td>
<td>Returns the element at given key, or null if key not present</td>
</tr>
<tr>
<td><code>public V put(K key, V value)</code></td>
<td>Associates given value with given key in the map. Returns old value if replaced, or null.</td>
</tr>
<tr>
<td><code>public void clear()</code></td>
<td>Removes all elements from list</td>
</tr>
<tr>
<td><code>public boolean containsKey(K key)</code></td>
<td>Searches for the given key in the map</td>
</tr>
<tr>
<td><code>public boolean containsValue(V value)</code></td>
<td>Searches for the given value in the map</td>
</tr>
<tr>
<td><code>public Set&lt;K&gt; keySet()</code></td>
<td>Returns a set of keys in the map</td>
</tr>
<tr>
<td><code>public V remove(K key)</code></td>
<td>Removes the element at given key. Returns the deleted element</td>
</tr>
<tr>
<td><code>public int size()</code></td>
<td>Returns the number key-value mappings in the map</td>
</tr>
<tr>
<td><code>public Collection&lt;V&gt; values()</code></td>
<td>Returns the Collection of values in the map</td>
</tr>
</tbody>
</table>
Example Usage: Contact list

```java
Map<String,String> myContacts = new HashMap<>();
myContacts.put("brad", "brad@pugetsound.edu");
myContacts.put("adam", "adam@pugetsound.edu");
myContacts.put("tony", "tony@pugetsound.edu");
myContacts.put("david", "david@gmail.com");

// Print Tony's contact info
System.out.println(myContacts.get("tony");

// Remove Adam's info
myContacts.remove("adam");

// Who are my friends?
for (String friend : myContacts.keySet()) {
    System.out.println(friend);
}
```
The real Map<K,V> interface has too many methods to write

- We'll implement the following reduced interface

```java
import java.util.Set;

/**
 * A very loose Map<K,V> interface.
 * @author David
 * @version 4/8/18
 */
public interface MapInt<K,V> {
    V get(K key);
    V put(K key, V value);
    V remove(K key);
    Set<K> keySet();
    int size();
    String toString();
}
```
Outline

- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
Use Lists to Store Map Entries

- Just like how trees and lists store Nodes, a map stores Entries
  - An Entry<\(K, V\)> stores a \((key, value)\) pair in the map

- Example of an Entry<String, String>
  - "david"
  - "david@gmail.com"

- Question is, \textit{how} are these entries stored?
  - In a list (or array)? In a tree? In a heap?
What we want

- \(O(1)\)-time search, insertion, deletion
  - How? Well, heaps showed how we could trade space for speed
- Arrays have \(O(1)\)-time access to its elements. Let's start there.

**Idea:** Use an array to hold Entry\(<K,V>\)s
To project Objects onto an array, we need a quick way to assign any Object to an integer.

Sure would be nice if Java gave Object a unique identifier...

<table>
<thead>
<tr>
<th>Object Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object other)</td>
<td>Returns true if the given object is &quot;equal to&quot; this one</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns a distinct integer representation of this object. Useful for identifying an object within a Collection.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a string representation of this object</td>
</tr>
<tr>
<td>(others omitted)</td>
<td></td>
</tr>
</tbody>
</table>
A couple caveats on `int hashCode()`

- Not *really* guaranteed to be unique, but Java tries
- The "code" may be large (below); it may be negative

Example:

```java
String x = "Andrew";
System.out.println(x.hashCode());
> 1965574029

Double n = 3.14;
System.out.println(n.hashCode());
> 300063655

Scanner keyboard = new Scanner(System.in);
System.out.println(keyboard.hashCode());
> 41997370
```
Map Implementation: Open Addressing (OA)

- **Open Addressing** Implementation
  - Exploits hash codes and a hash table
  - Each array element represents a single *(key, value)* Entry
    - Important: Array position referencing null means the key does not exist in the map
Consider the following Entries are to be inserted:

- Entry(k1,v1), Entry(k2,v2), Entry(k3,v3), and Entry(k4,v4)

Assume the following hash codes:

- k1.hashCode() == -4302
- k2.hashCode() == 0
- k3.hashCode() == 20100021
- k4.hashCode() == 1938821
Problem with Direct Hashing

- Assume these keys have the following hash codes:
  - k1.hashCode() == -4302
  - k2.hashCode() == 0
  - k3.hashCode() == 20100021
  - k4.hashCode() == 1938821

**Problem:** HashCodes can be large! I don't want to create a huuuuuuge (and mostly empty) array.
Problem with Direct Hashing

- **Problem**: Hash range is too large. Can't use huge array
  - Solution: Use a fixed table size, and shrink the hash range down
Dealt with this before... how to restrict a large range within a smaller range?

- Given any integer \( i : i \mod n \in [0, n - 1] \)
- Examples:
  - \( 30000 \mod 3 = 0 \)
  - \( 35 \mod 3 = 2 \)
  - \( 884848222 \mod 3 = 1 \)
Our Starter Code

```java
public class OpenMap<K,V> implements MapInt<K,V> {
    private Entry<K,V>[] table;
    private int size;

    /**
     * Our inner Entry class, used to hold (key,value) pairs.
     */
    private static class Entry<K,V> {
        private K key;
        private V value;

        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }

        // We should really write accessors as well...

        public String toString() {
            return key + "=" + value;
        }
    }
}
```
OA: put(K key, V value)

- **Step 1: Index calculation:** \[ \text{Math.abs(key.hashCode()) \% table.length} \]
- **Step 2: Linear probing**
  - If entry is non-null, try next one down (wraparound at the end)
    - Insert when null is found, or replace when key is found. Return old value.
    - If all entries are non-null, throw IllegalArgumentException

<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>
**put(): Trivial Case**

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

- **Step 2: Linear probing**
  - If entry is non-null, try next one down (wraparound at the end)
    - Insert when null is found, or replace when key is found. Return old value.
    - If all entries are non-null, throw IllegalArgumentException

---

**put("Brad","brad@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]: null
- [2]: null
- [3]: ("Brad","brads@pugetsound.edu")
- [4]: null
put(): Trivial Case (Cont.)

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

- Step 2: Linear probing
  - If entry is non-null, try next one down (wraparound at the end)
    - Insert when null is found, or replace when key is found. Return old value.
    - If all entries are non-null, throw IllegalArgumentException

```
put("Adam","adam@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

null

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

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

null

null

null```
put(): Trivial Case (Cont.)

- **Step 1: Index calculation:** `Math.abs(key.hashCode()) % table.length`

- **Step 2: Linear probing**
  - If entry is non-null, try next one down (wraparound at the end)
    - Insert when null is found, or replace when key is found. Return old value.
    - If all entries are non-null, throw `IllegalArgumentException`
put(): Linear Probing

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

- **Step 2: Linear probing**
  - If entry is non-null, try next one down (wraparound at the end)
    - Insert when null is found, or replace when key is found. Return old value.
    - If all entries are non-null, throw IllegalArgumentException

```
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>
put() : Linear Probing (Cont.)

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

- **Step 2: Linear probing**
  - If entry is non-null, try next one down (wraparund at the end)
    - Insert when null is found, or replace when key is found. Return old value.
    - If all entries are non-null, throw IllegalArgumentException

```
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>

```
Check: key.equals("Adam")? No. Move on!
Check: key.equals("Brad")? No. Move on!
Check: key.equals("David")? No. Move on!
```
put(): Linear Probing (Wraparound)

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

- Step 2: Linear probing
  - If entry is non-null, try next one down (wraparound at the end)
    - Insert when null is found, or replace when key is found. Return old value.
    - If all entries are non-null, throw IllegalArgumentException
**put():** Linear Probing (Replacement)

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

- **Step 2:** Linear probing
  - If entry is non-null, try next one down (wraparound at the end)
    - Insert when null is found, or replace when key is found. Return old value.
    - If all entries are non-null, throw IllegalArgumentException

```
put("Tony","tony22@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]: ("Tony","tony22@pugetsound.edu")

Check: key.equals("Adam")? No. Move on!
Check: key.equals("Brad")? No. Move on!
Check: key.equals("David")? No. Move on!
Check: key.equals("Tony")? Yes! Replace!
Step 1: Index calculation: $\text{Math.abs(key.hashCode())} \mod \text{table.length}$

Step 2: Linear probing

- If entry is non-null, try next one down (wraparound at the end)
  - Insert when null is found, or replace when key is found. Return old value.
  - If all entries are non-null, throw IllegalArgumentException

`put()` : Table is full!

(Why do we still need to probe if we knew table was full already?)