CSCI 261
Computer Science II
Outline

- Sets and Baby Set Theory
- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Performance
- Conclusion
New Problem

- Data management is hard.
  - Consider PeopleSoft... student info stored in a list

<table>
<thead>
<tr>
<th>SSN</th>
<th>FName</th>
<th>LName</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Adam</td>
<td>Smith</td>
<td>CS</td>
</tr>
<tr>
<td>7</td>
<td>Brad</td>
<td>Richards</td>
<td>CS</td>
</tr>
<tr>
<td>10</td>
<td>Adam</td>
<td>Smith</td>
<td>GEOL</td>
</tr>
<tr>
<td>21</td>
<td>Sigrun</td>
<td>Bodine</td>
<td>MATH</td>
</tr>
<tr>
<td>10</td>
<td>Adam</td>
<td>Smith</td>
<td>MATH</td>
</tr>
</tbody>
</table>
New Problem

- Common query:
  - How many different majors does Puget Sound have?

<table>
<thead>
<tr>
<th>SSN</th>
<th>FName</th>
<th>LName</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Adam</td>
<td>Smith</td>
<td>CS</td>
</tr>
<tr>
<td>7</td>
<td>Brad</td>
<td>Richards</td>
<td>CS</td>
</tr>
<tr>
<td>10</td>
<td>Adam</td>
<td>Smith</td>
<td>GEOL</td>
</tr>
<tr>
<td>21</td>
<td>Sigrun</td>
<td>Bodine</td>
<td>MATH</td>
</tr>
<tr>
<td>10</td>
<td>Adam</td>
<td>Smith</td>
<td>MATH</td>
</tr>
</tbody>
</table>

Assume each row is a Student object
Here's one solution...

- It runs in $O(n)$ time, where $n = \text{number of students}$

```java
/**
 * @param a list of students
 * @return the number of majors at Puget Sound
 */
public int numMajors(List<Student> list) {
    List<String> majors = new ArrayList<>();
    for (int i = 0; i < list.size(); i++) {
        Student s = list.get(i);
        majors.add(s.getMajor());  // Add the student's major to the list
    }
    return majors.size();
}
```

- But the count won't be accurate because we'd expect lots of duplicates
But not very elegant:

- Added a helper method just to de-duplicate...
- Complexity now $O(nm)$, where $m$ is the number of unique majors

```java
public int numMajors(List<Student> list) {
    List<String> majors = new ArrayList<>();
    for (int i = 0; i < list.size(); i++) {
        Student s = list.get(i);
        if (!isDuplicate(majors, s.getMajor())) {
            majors.add(s.getMajor());
        }
    }
    return majors.size();
}

private boolean isDuplicate(List<String> list, String major) {
    for (int j = 0; j < list.size(); j++) {
        if (list.get(j).equals(major)) { // Already exists.
            return true;
        }
    }
    return false;
}
```
Other "Data Management" Routines

- Find every Student with the last name "Smith"
  - Get a list of all students with last name Smith
  - Remove anyone who shares the same ID

- List all the CS and Math majors
  - Get a list of CS majors; Get a list of Math majors
  - Add all students from both lists to a new list, ignoring dupes

- (Lots more...)

- Clearly, data duplication is a problem in any database system...
Want

- We need a new data structure that efficiently deals with duplicates

- Want to use the first algorithm without the hassle!

```java
/**
 * @param a list of students
 * @return the number of majors at Puget Sound
 */
public int numMajors(List<Student> list) {
    majors = new Majors();
    for (int i = 0; i < list.size(); i++) {
        Student s = list.get(i);
        majors.add(s.getMajor()); // Automatically ignores duplicates when adding
    }
    return majors.size();
}
```
Sets: Dealing with Duplication

- Mathematically, a set is an unordered collection of unique objects:
  - Yay no duplicates by definition

- In Java, a Set<E> is a Java interface that models the mathematical
definition of a set.
  - Unique objects only (e.g., add() method ignores dupes)
  - No index associated with objects (i.e., unlike Lists)
    - (How do you traverse a Set then?)

```java
import java.util.Set;
```
Java Collection Interface
Baby Set Theory: Set Expression

- Sets can be expressed mathematically in several ways
  - **Roster Notation**
    \[ A = \{0, 1, 3, 4, 6\} \]
    Note: \( \emptyset \) or \( \{\} \) is called the *empty set* (a set containing no elements)
  - **Set-Builder Notation**
    \[ B = \{b | b \% 2 == 0 \text{ and } b > 0\} \]
  - **Venn Diagram**
Baby Set Theory: Set Operations

- **Cardinality:** $|S|$ returns the number of elements in the set $S$

  $|\emptyset| = 0 \quad |\{a, b, c\}| = 3$
Baby Set Theory: Set Operations

- **Cardinality:** $|S|$ returns the number of elements in the set $S$

  $$|\emptyset| = 0 \quad |\{a, b, c\}| = 3$$

- **Membership:** $x \in S$ is *true* if element $x$ is found in $S$ and *false* otherwise

  $$a \in \{\} = \text{false} \quad 64 \in B = \text{true}$$

  $$a \notin \{\} = \text{true} \quad b \in \{a, b, c\} = \text{true}$$
Baby Set Theory: Set Operations

- **Cardinality:** \(|S|\) returns the number of elements in the set \(S\)
  \[
  |\emptyset| = 0 \quad |\{a, b, c\}| = 3
  \]

- **Membership:** \(x \in S\) is \(true\) if element \(x\) is found in \(S\) and \(false\) otherwise
  \[
  a \in \{} = false \quad 64 \in B = true
  \]
  \[
  a \notin \{} = true \quad b \in \{a, b, c\} = true
  \]

- **Set Existence:** \(\exists S\) is \(true\) if the set \(S\) exists, and \(false\) otherwise
  \[
  \exists \{x | x \text{ is prime} \land x \leq 1\} = false
  \]
Baby Set Theory: Set Operations

\[ A = \{0, 1, 3, 4, 6\} \quad B = \{b \mid b \% 2 == 0 \text{ and } b > 0\} \]

- **Subset:** \( S_1 \subseteq S_2 \) is **true** if all elements in \( S_1 \) are in \( S_2 \) and **false** otherwise

\[ \emptyset \subseteq \{a, b, c\} = \text{true} \]

\[ A \nsubseteq B = \text{true} \]

\[ \{16, 32, 64, 128\} \subseteq B = \text{true} \]

\[ \{c, d\} \subseteq \{c, d\} = \text{true} \]
Baby Set Theory: Set Operations

- **Set Equivalence:** $S_1 = S_2$ is **true** if all elements in $S_1$ are in $S_2$ and vice versa.

  $$\{c, d\} = \{d, c\} = \text{true}$$

- **Proper Subset:** $S_1 \subset S_2$ is **true** if all elements in $S_1$ are in $S_2$ and the two sets are not equal, and **false** otherwise.

  $$\emptyset \subset \{a, b, c\} = \text{true}$$

  $$\{c\} \subset \{c, d\} = \text{true}$$

  $$\{c, d\} \subset \{c, d\} = \text{false}$$
Baby Set Theory: Set Operations

- **Union:** $S_1 \cup S_2$ returns the set of all elements in $S_1$ and $S_2$

  $$\{0, 1, 3, 4, 6\} \cup \{0, 4, 6, 8, 14\} = \text{??}$$

- Venn diagram interpretation

![Venn diagram example](image-url)
Baby Set Theory: Set Operations

- **Intersection:** \( S_1 \cap S_2 \) returns the set of elements in both \( S_1 \) and \( S_2 \)

\[
\{0, 1, 3, 4, 6\} \cap \{0, 4, 6, 8, 14\} = \text{??}
\]

- Venn diagram interpretation

\[
\begin{align*}
A & \quad B \\
3 & \quad 0 \\
1 & \quad 6 \\
4 & \quad 4 \\
14 & \quad 8 \\
\end{align*}
\]

\[
A \cap B = \begin{align*}
0 & \\
6 & \\
4 & \\
\end{align*}
\]
Baby Set Theory: Set Operations

- **Set Difference**: \( S_1 \setminus S_2 \) (or \( S_1 - S_2 \)) returns the set of elements in \( S_1 \) but not in \( S_2 \)

\[
\{0, 1, 3, 4, 6\} \setminus \{0, 4, 6, 8, 14\} = \text{??}
\]

- Venn diagram interpretation
## Java's Set<E> Interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean add(E obj)</td>
<td>$S = S \cup {obj}$</td>
</tr>
<tr>
<td>public boolean addAll(Collection&lt;E&gt; c)</td>
<td>$S = S \cup C$</td>
</tr>
<tr>
<td>public boolean contains(Object obj)</td>
<td>$obj \in S$</td>
</tr>
<tr>
<td>public boolean containsAll(Collection&lt;E&gt; c)</td>
<td>$C \subseteq S$</td>
</tr>
<tr>
<td>public boolean isEmpty()</td>
<td>$</td>
</tr>
<tr>
<td>public boolean remove(Object obj)</td>
<td>$S = S \setminus {obj}$</td>
</tr>
<tr>
<td>public boolean removeAll(Collection&lt;E&gt; c)</td>
<td>$S = S \setminus C$</td>
</tr>
<tr>
<td>public boolean retainAll(Collection&lt;E&gt; c)</td>
<td>$S = S \cap C$</td>
</tr>
<tr>
<td>public int size()</td>
<td>$</td>
</tr>
<tr>
<td>public Iterator&lt;E&gt; iterator()</td>
<td>Returns an iterator over elements in this set: $\forall s_i \in S \ldots$</td>
</tr>
</tbody>
</table>
import java.util.*;

public class Test {
    public static void main(String[] args) {
        String[] listA = {"Ann", "Sally", "Jill", "Sally"};
        String[] listB = {"Bob", "Bill", "Ann", "Jill"};

        Set<String> A = new HashSet<>();
        Set<String> Acopy = new HashSet<>();
        Set<String> B = new HashSet<>();

        for (int i = 0; i < listA.length; i++) {
            A.add(listA[i]);
            Acopy.add(listA[i]);
        }

        for (int i = 0; i < listB.length; i++) {
            B.add(listB[i]);
        }

        A.addAll(B);
        Acopy.retainAll(B);

        Iterator<String> itr = A.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }

        for (String s : A) {
            System.out.println(s);
        }
    }
}
Your Turn: What's the Output?

```java
import java.util.*;

class Test {
    public static void main(String[] args) {
        Set<String> s = new HashSet<>();
        s.add("hello");
        s.add("bye");
        s.addAll(s);

        Set<String> t = new HashSet<>();
        t.add("123");
        s.addAll(t);
        System.out.println(s.containsAll(t));
        System.out.println(t.containsAll(s));
        System.out.println(s.contains("ace"));
        System.out.println(s.contains("123"));
        s.retainAll(t);
        System.out.println(s.contains("123"));
        t.retainAll(s);
        System.out.println(s.contains("123"));
    }
}
```
Outline

- Sets and Baby Set Theory
- Why Maps?
- Map: List Implementation
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Performance
- Conclusion
Motivation: Maps

- Consider programming up a contact directory...
  - You'd probably use a List to store contacts
Using Lists to Store Contacts

```java
List<String> myContacts = new ArrayList<>();
myContacts.add("brad@pugetsound.edu");
myContacts.add("adam@pugetsound.edu");
myContacts.add("tony@pugetsound.edu");
myContacts.add("david@pugetsound.edu");

// Print Tony's contact info
```
Using Lists to Store Contacts

```java
List<String> myContacts = new ArrayList<>();
myContacts.add("brad@pugetsound.edu");
myContacts.add("adam@pugetsound.edu");
myContacts.add("tony@pugetsound.edu");
myContacts.add("david@pugetsound.edu");

// Print Tony's contact info
System.out.println(myContacts.get(2));

// Remove Adam's info
```
Using Lists to Store Contacts

```
List<String> myContacts = new ArrayList<>();
myContacts.add("brad@pugetsound.edu");
myContacts.add("adam@pugetsound.edu");
myContacts.add("tony@pugetsound.edu");
myContacts.add("david@pugetsound.edu");

// Print Tony's contact info
System.out.println(myContacts.get(2));

// Remove Adam's info
myContacts.remove(1);
```

(1) What's unsettling about printing Tony and removing Adam? What if I sorted the List?

(2) How would I search a contact by name?

```
Problem: Integers are not natural indices for contacts, names are!
```
We want to index by name (which are Strings), not by ints

- Still want to store emails (also Strings)

These data structures are called **Maps**

**What We Want: Our Own Index!**

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>&quot;<a href="mailto:brad@pugetsound.edu">brad@pugetsound.edu</a>&quot;</td>
</tr>
<tr>
<td>[1]</td>
<td>&quot;<a href="mailto:adam@pugetsound.edu">adam@pugetsound.edu</a>&quot;</td>
</tr>
<tr>
<td>[2]</td>
<td>&quot;<a href="mailto:tony@pugetsound.edu">tony@pugetsound.edu</a>&quot;</td>
</tr>
<tr>
<td>[3]</td>
<td>&quot;<a href="mailto:david@pugetsound.edu">david@pugetsound.edu</a>&quot;</td>
</tr>
</tbody>
</table>

**With Lists/Array**

**With Maps**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;brad&quot;</td>
<td>&quot;<a href="mailto:brad@pugetsound.edu">brad@pugetsound.edu</a>&quot;</td>
</tr>
<tr>
<td>&quot;adam&quot;</td>
<td>&quot;<a href="mailto:adam@pugetsound.edu">adam@pugetsound.edu</a>&quot;</td>
</tr>
<tr>
<td>&quot;tony&quot;</td>
<td>&quot;<a href="mailto:tony@pugetsound.edu">tony@pugetsound.edu</a>&quot;</td>
</tr>
<tr>
<td>&quot;david&quot;</td>
<td>&quot;<a href="mailto:david@pugetsound.edu">david@pugetsound.edu</a>&quot;</td>
</tr>
</tbody>
</table>
Maps

- A map $M$ is a set of ordered key-value pairs: $M = \{(k_1, v_1), (k_2, v_2), \ldots\}$
  - Key-value pairs are unique
    - Enforced by: keys must be unique; values are not
  - Many-to-one mapping: One value can be mapped by multiple keys

Maps Strings (key) to Integers (val):

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

Maps Strings to Strings:

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

- A map \( M \) is a set of ordered key-value pairs: \( M = \{(k_1, v_1), (k_2, v_2), \ldots\} \)
  - Key-value pairs are unique
    - Enforced by: keys must be unique; values are not
  - Many-to-one mapping: One value can be mapped by multiple keys

```java
Map<String, String> myContacts = new HashMap<>();
myContacts.put("brad", "brad@pugetsound.edu");
```
A map $M$ is a set of ordered key-value pairs: $M = \{(k_1, v_1), (k_2, v_2), \ldots\}$

- Key-value pairs are unique
  - Enforced by: keys must be unique; values are not
- Many-to-one mapping: One value can be mapped by multiple keys

```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");
```
Maps

- A map $M$ is a set of ordered key-value pairs: $M = \{(k_1, v_1), (k_2, v_2), \ldots\}$
  - Key-value pairs are unique
    - Enforced by: keys must be unique; values are not
    - Old value (if exists) is overwritten
  - Many-to-one mapping: One value can be mapped by multiple keys

```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");
```
A map $M$ is a set of ordered key-value pairs: $M = \{(k_1, v_1), (k_2, v_2), \ldots \}$

- Key-value pairs are unique
  - Enforced by: keys must be unique; values are not
- Many-to-one mapping: Same value can be mapped by multiple keys

```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>public Map&lt;K, V&gt;()</td>
<td>Constructs a new map that maps indices of type K to values of type V</td>
</tr>
<tr>
<td>public V get(K key)</td>
<td>Returns the element at given key, or null if key not present</td>
</tr>
<tr>
<td>public V put(K key, V value)</td>
<td>Associates given value with given key in the map. Returns old value if replaced, or null.</td>
</tr>
<tr>
<td>public void clear()</td>
<td>Removes all elements from list</td>
</tr>
<tr>
<td>public boolean containsKey(K key)</td>
<td>Searches for the given key in the map</td>
</tr>
<tr>
<td>public boolean containsValue(V value)</td>
<td>Searches for the given value in the map</td>
</tr>
<tr>
<td>public Set&lt;K&gt; keySet()</td>
<td>Returns a set of keys in the map</td>
</tr>
<tr>
<td>public V remove(K key)</td>
<td>Removes the element at given key. Returns the deleted element</td>
</tr>
<tr>
<td>public int size()</td>
<td>Returns the number key-value mappings in the map</td>
</tr>
<tr>
<td>public Collection&lt;V&gt; values()</td>
<td>Returns the Collection of values in the map</td>
</tr>
</tbody>
</table>
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

// Remove Adam's info

// Who are my friends?
Example Usage: Contact list

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?
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?
Set<String> friends = myContacts.keySet(); // returns the set of the keys
for (String f : friends) {
  System.out.println(f);
}
```
Again, We'll Roll Our Own Map Interface

- 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: List Implementation**
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Performance
- Conclusion
Use Lists to Store Map Entries

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

  Key is type K  
  Value is type V

  "david"  "david@gmail.com"

- Question is, \textit{how}, are these entries stored?
  - In a list? In a tree? etc.
  - Let's try List
public class ListMap<K,V> implements MapInt<K,V> {
    private List<Entry<K,V>> list;

    /**
     * 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;
        }
    }
}

[Let's Implement the rest]
ListMap Performance

- **Pros**: Easy to implement
- **Cons**: performance sucks
  - **put()**: May need to iterate through all entries before insertion
  - **get()**: Ditto
  - **remove()**: Ditto
  - **Goal is**: Can we do better than BSTs and Heaps?

<table>
<thead>
<tr>
<th>Operations</th>
<th>ListMap</th>
<th>Is this attainable?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(key, val)</code></td>
<td>$O(n)$ avg/worst</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>get(key)</code></td>
<td>$O(n)$ avg/worst</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>remove(key)</code></td>
<td>$O(n)$ avg/worst</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Outline

- Why Maps?
- Map: List Implementation
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
Hashing and HashTables

- What we (still) 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

- Idea: Use an array to hold $\text{Entry}<K,V>$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>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object other)</td>
<td></td>
<td>Returns true if the given object is &quot;equal to&quot; this one</td>
</tr>
<tr>
<td>int hashCode()</td>
<td></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></td>
<td>Returns a string representation of this object</td>
</tr>
<tr>
<td>(others omitted)</td>
<td></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
```
Open Addressing

- Exploits hash codes and a hash table
- Each array element represents a single \((key,value)\) Entry
  - Important: Array position referencing \texttt{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
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 Straight Hashing

- Problem: Hash range is too large. Can't use huge array
  - Solution: Use a fixed table size, and shrink the hash range down

```
Math.abs(k1.hashcode())
Math.abs(k2.hashcode())
Math.abs(k3.hashcode())
Math.abs(k4.hashcode())
```

Table (array)

Hashtable (size 6)

- [0]
- [1]
- [2]  (k1, v1)
- [3]  (k4, v4)
- [4]  (k2, v2)
- [5]  (k3, v3)
Dealt with this before... how to restrict a large range within a smaller range?

- Given any integer $i$ : $i \% n \in [0, n - 1]$
- Examples:
  - $30000 \% 3 = 0$
  - $35 \% 3 = 2$
  - $884848222 \% 3 = 1$

Math.\texttt{abs}(k1.hashcode())
Math.\texttt{abs}(k2.hashcode())
Math.\texttt{abs}(k3.hashcode())
Math.\texttt{abs}(k4.hashcode())

% table.length
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: `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]: ("Brad","brads@pugetsound.edu")
- [3]: null
- [4]: null
put(): Trivial Case (Cont.)

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

- 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("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
[0] null
[1] ("Adam","adam@pugetsound.edu")
[2] null
[3] ("Brad","brad@pugetsound.edu")
[4] null
```
**put()**: Trivial Case (Cont.)

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

- **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("David","david@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

null
[0] null
[1] ("Adam","adam@pugetsound.edu")
[2] null
[3] ("Brad","brad@pugetsound.edu")
[4] ("David","david@pugetsound.edu")
null
```
put(): Linear Probing

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

- **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")
```
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 (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

### Example

```java
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>

```
null
[0] ("Adam", "adam@pugetsound.edu")
[1] ("Brad", "brad@pugetsound.edu")
[2] ("David", "david@pugetsound.edu")
[3] ("Tony", "tony@pugetsound.edu")
[4] collision!
```
put(): Linear Probing (Wraparound)

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

- **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("America","america@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>

```
Collision!
```

Table:

<table>
<thead>
<tr>
<th>[0]</th>
<th>(&quot;America&quot;,&quot;<a href="mailto:america@pugetsound.edu">america@pugetsound.edu</a>&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>(&quot;Adam&quot;,&quot;<a href="mailto:adam@pugetsound.edu">adam@pugetsound.edu</a>&quot;)</td>
</tr>
<tr>
<td>[2]</td>
<td>(&quot;Brad&quot;,&quot;<a href="mailto:brad@pugetsound.edu">brad@pugetsound.edu</a>&quot;)</td>
</tr>
<tr>
<td>[3]</td>
<td>(&quot;David&quot;,&quot;<a href="mailto:david@pugetsound.edu">david@pugetsound.edu</a>&quot;)</td>
</tr>
<tr>
<td>[4]</td>
<td>(&quot;Tony&quot;,&quot;<a href="mailto:tony@pugetsound.edu">tony@pugetsound.edu</a>&quot;)</td>
</tr>
</tbody>
</table>
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 \text{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>

```
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("Ann","ann@pugetsound.edu")
```

Back to original spot! throw IllegalArgumentException!
(Why probe? We knew table was full already)

**Key** | **hashCode()** | **hashCode() % 5**
--- | --- | ---
"Brad" | 2078867 | 2
"Adam" | 2035631 | 1
"David" | 65805908 | 3
"Tony" | 2612646 | 1
"America" | 775550446 | 1
"Ann" | 65985 | 0

Table:

```
[0] ("America","america@pugetsound.edu")
[1] ("Adam","aasmith@pugetsound.edu")
[2] ("Tony","tony@pugetsound.edu")
[3] ("David","david@pugetsound.edu")
[4] ("Brad","brad@pugetsound.edu")
```
Outline

▶ Why Maps?

▶ Map: Open Addressing
  • Put Implementation
  • Get Implementation
  • Performance

▶ Map: Chaining
  • Put Implementation
  • Get Implementation
  • Performance

▶ Conclusion
The get() method searches the map for the given key.

- Implementation is similar to put()
- Two cases:
  - Key is found: Linearly probe table and key is found
  - Key is not found: Linearly probe table and null is found, or you've seen all entries

```java
get("Tony")
```

<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>

**Collision!**

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!
The get() method searches the map for the given key.

- Implementation is similar to put()
- Two cases:
  - Key is found: Linearly probe table and key is found
  - Key is not found: Linearly probe table and null is found, or you've seen all entries

```
get("Pat")
```

<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;Pat&quot;</td>
<td>80003</td>
<td>3</td>
</tr>
</tbody>
</table>

collision!

Check:  key.equals("David")? No. Move on!
Check:  key.equals("Tony")? No. Move on!
Found null entry! Not in table!!!

null

("Adam","adam@pugetsound.edu")
("Brad","brad@pugetsound.edu")
("David","david@pugetsound.edu")
("Tony","tony@pugetsound.edu")
Outline

- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Remove Implementation
  - Performance (and Rehashing)
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
The remove() method searches the map for the given key also...

- Returns victim's element if key exists, or null if not
- Deceptively hard to perform...

```
remove("David")
```

<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:

```
null
[0]  ("Adam", "adam@pugetsound.edu")
[1]  ("Brat", "brad@pugetsound.edu")
[2]  ("David", "david@pugetsound.edu")
[3]  ("Tony", "tony@pugetsound.edu")
[4]  
```
The remove() method searches the map for the given key also...

- Returns victim's element if key exists, or null if not
- Deceptively hard to perform...

```
remove("David")
```

<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>
The remove() method searches the map for the given key also...

- Returns victim's element if key exists, or null if not
- Deceptively hard to perform...

Consider this problem...

```
public V remove(K key)
```

```
put("Tony", "tonayyy@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: Woohoo free spot!! Insert Tony!
Problem: Can't remove by replacing entry with null

- Solution: Replace with a "dummy" value instead
- Consequences:
  - Put() and Get() will work now (because it only stops when it sees null)
  - But when putting a new entry, it must use-up DELETED spots!
Outline

- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance (and Rehashing)
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
Performance of Open-Addressing HashMaps

- No hash collision: $O(1)$

- Then how **frequently** do we have hash collisions?
  - Depends on how full the hash table is (i.e., load factor)
  - After collision, number of probes depends dominates performance
    - Holes in the hashtable are our friend; more holes, better performance

```java
put("America","america@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>

Collision!

```

("America","america@pugetsound.edu")
("Adam","adam@pugetsound.edu")
("Brad","brad@pugetsound.edu")
("David","david@pugetsound.edu")
("Tony","tony@pugetsound.edu")
```
But what if we had a larger hash table begin with?

- **Bad:** more wasted space
- **Good:**
  - Fewer collisions
  - Shorter average probe lengths
- *(Space-time tradeoff)*

```
put("America", "america@pugetsound.edu")
```

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 15</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>11</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>8</td>
</tr>
<tr>
<td>&quot;Tony&quot;</td>
<td>2612646</td>
<td>6</td>
</tr>
<tr>
<td>&quot;America&quot;</td>
<td>775550446</td>
<td>1</td>
</tr>
</tbody>
</table>
What is the performance of the Open-Addressing (OA) map?

- Range between $O(1)$ and $O(n)$...
- If we can keep the hash table relatively *sparse*, performance leans toward $O(1)$

The time complexity of OA, $T_{OA}$, can be approximated by the following equation (D. Knuth, 1973):

$$T_{OA} \approx \frac{1}{2} \left(1 + \frac{1}{1 - L}\right)$$

where $L$ is the *Load Factor* = *stored entries / hashtable size*
Performance of Open-Addressing HashMaps

- Performance per operation worsens quadratically over L
  - Important: The following results are true regardless of size of hashtable
  - Takeaway: we need to keep L at manageable levels!

<table>
<thead>
<tr>
<th>Load Factor</th>
<th>Number of Probes (T_{OA})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>0.25</td>
<td>1.17</td>
</tr>
<tr>
<td>0.5</td>
<td>1.50</td>
</tr>
<tr>
<td>0.75</td>
<td>2.50</td>
</tr>
<tr>
<td>0.85</td>
<td>3.83</td>
</tr>
<tr>
<td>0.9</td>
<td>5.50</td>
</tr>
<tr>
<td>0.95</td>
<td>10.50</td>
</tr>
<tr>
<td>1.0</td>
<td>Infinity (well, n in our case)</td>
</tr>
</tbody>
</table>
Reducing Collisions: Rehashing

- Hash collisions leads to lengthier linear probing
  - How to reduce collisions?

- One way to reduce hash collisions is to "rehash" the hashtable:
  - Like how ArrayList and Heap works when they run out of space!
  - Let's say we expand our size=5 hashtable to a size=17

<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>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>7</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>13</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>8</td>
</tr>
</tbody>
</table>
Rehashing (Cont.)

old (L = 1.0)

<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>

ew (L = 0.29)

<table>
<thead>
<tr>
<th>Key</th>
<th>hashCode()</th>
<th>hashCode() % 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Brad&quot;</td>
<td>2078867</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>7</td>
</tr>
<tr>
<td>&quot;David&quot;</td>
<td>65805908</td>
<td>13</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>8</td>
</tr>
</tbody>
</table>
Rehashing (Cont.)

- Great, each map operation is back to ~1.20 probes after rehashing to size 17!
  - Rehashing takes linear time; but do it rarely, so it's amortized $O(1)$

Rehashing steps:
- Instantiate a new array of larger size
- For each $(key, value)$ entry in old hashtable:
  - Insert $(key, value)$ into new table using new hash function
- Replace hashtable to reference the new one

- Rule of thumb: Rehash when load factor > 0.5
OA Map Performance Summary

- **Pros:**
  - \(O(1)\)-time insert/delete/search
    - If load factor \(L\) is controlled \((L \leq 0.5)\)

- **Cons:**
  - Performance of Open Addressing largely depends on load factor, \(L\).
    - Complexity worsens as \(L\) approaches 100%
    - Combat by reducing load factor: Expand hash table + rehash
  - The real problem with OA is space complexity:
    - Lowering \(L\) to a reasonable number means a lot of wasted space
Outline

- Why Maps?
- HashMap: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- HashMap: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
Map Implementation: Chaining

- **Chaining method**
  - Use an array-based *Hash Table* as before
  - Each array element references a list of *(key, value)* pairs
    - "Chain" also called a *bucket*

- **Consider the following entries are to be inserted:**
  - *(k1,v1), (k2,v2), (k3,v3), (k4,v4)*
Chaining: put(K key, V value)

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

- Step 2: Chain walk
  - If entry is non-null, walk the chain
    - Insert if key is not found in chain, or replace when key is found. Return old value.

<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>
Chaining: `put(K key, V value)`

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

- **Step 2: Chain walk**
  
  - If entry is non-null, walk the chain
    
    - Insert if key is not found in chain, or replace when key is found. Return old value.

```
put("David","david@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>
Chaining: `put(K key, V value)`

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

- **Step 2: Chain walk**
  - If entry is non-null, walk the chain
    - Insert if key is not found in chain, or replace when key is found. Return old value.

```
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>

**Table:**

- [0]: null
- [1]: ("Adam","adam@pugetsound.edu")
- [2]: ("Brad","brad@pugetsound.edu")
- [3]: ("David","david@pugetsound.edu")
- [4]: null

("Tony", ...)
Chaining: `put(K key, V value)`

- **Step 1:** Index calculation: \[
    \text{Math.abs(key.hashCode())} \% \text{table.length}
\]
- **Step 2:** Chain walk
  - If entry is non-null, walk the chain
    - Insert if key is not found in chain, or replace when key is found. Return old value.

```
put("Adam","aasmith@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>
Chaining: `put(K key, V value)`

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

- **Step 2: Chain walk**
  - If entry is non-null, walk the chain
    - Insert if key is not found in chain, or replace when key is found. Return old value.

```
put("America","america@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>
Outline

- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
Chaining: get(K key)

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

- Step 2: Chain walk
  - If entry is null, return null
  - If entry is non-null, walk the chain
    - Return if entry found in chain, or return null if not

```java
get("Bobert")
```

<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>
<tr>
<td>&quot;Bobert&quot;</td>
<td>1995055026</td>
<td>0</td>
</tr>
</tbody>
</table>

```
null
("Tony", ...)
("America", ..)
null
("Brad", "brad@pugetsound.edu")
("David", "david@pugetsound.edu")
("Adam", "aasmith@pugetsound.edu")
Not found! Return null
```
Step 1: Index calculation:

Math.abs(key.hashCode()) % table.length

Step 2: Chain walk

- If entry is null, return null
- If entry is non-null, walk the chain
  - Return if entry found in chain, or return null if not

Chaining: put(K key, V value)
Outline

- Why Maps?
- Map: Open Addressing
  - Put Implementation
  - Get Implementation
  - Performance
- Map: Chaining
  - Put Implementation
  - Get Implementation
  - Performance
- Conclusion
What is the performance of the chained map?

The time complexity of chained maps, $T_{\text{chain}}$, can be approximated by the following equation (D. Knuth, 1973):

$$T_{\text{chain}} \approx 1 + \frac{L}{2}$$

where $L = \text{(stored entries / hashtable size)}$ is the *Load Factor*.
Performance of Chained Map

\[ T_{\text{chain}} \approx 1 + \frac{L}{2} \]

where \( L = \text{(stored entries / hashtable size)} \) is the \textit{Load Factor}

- Suppose \( n \) Entries are stored in a chained HashMap as follows
  - hashCodes are randomly/uniformly distributed across the array

1. Hash into table
2. Walk half the chain. (Average case of linear search for key)
   Avg size of chain: stored entries / hashtable size (= L)
## OA Map vs. Chained Map

### Load Factor vs. Number of Steps (T_chain)

<table>
<thead>
<tr>
<th>Load Factor</th>
<th>Number of Steps (T_chain)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>0.25</td>
<td>1.13</td>
</tr>
<tr>
<td>0.5</td>
<td>1.25</td>
</tr>
<tr>
<td>0.75</td>
<td>1.38</td>
</tr>
<tr>
<td>0.85</td>
<td>1.43</td>
</tr>
<tr>
<td>0.9</td>
<td>1.45</td>
</tr>
<tr>
<td>0.95</td>
<td>1.48</td>
</tr>
<tr>
<td>1.0</td>
<td>1.50</td>
</tr>
</tbody>
</table>

![Graph showing the comparison between Chaining and OA methods](image.png)

**Legend:**
- **Chaining**
- **OA**
Chaining Performance Summary

- **Pros:**
  - O(1)-time insert/search/delete on average
  - Generally better performance than open addressing
    - Especially for high load factors (L > 0.75)

- **Cons:**
  - Requires far more space than open addressing!
  - Example: We want 2.5 comparisons on average for 60,000 items.
    - Using OA, we need \( N = 80,000 \) hashtable to obtain \( L = 0.75 \)
    - Using chaining, we need \( N = 20,000 \) hashtable
      - 60,000 references for singly linked list + 20,000 hashtable
      - 120,000 references for doubly linked list + 20,000 hashtable
Chained Map Summary

- Things to consider
  - Still need to rehash occasionally to shorten average chain length
    - Just not as critical to performance as in Open Addressing
    - Could monitor longest or average chain length, rehash when appropriate

- For what it's worth: Java's HashMap<K,V> class is a chained map
Outline

» Why Maps?
» Map: Open Addressing
  • Put Implementation
  • Get Implementation
  • Performance
» Map: Chaining
  • Put Implementation
  • Get Implementation
  • Performance
» Conclusion
Administrivia 4/9

- Final exam schedule is up

- New code up:
  - ListMap
  - HashMap (Open Addressing)

- Homework 5 due next Monday 4/16

- Tomorrow's lab:
  - Binary Search Tree
Lab 11 due tonight!
  • largest(), smallest(), size(), height()
  • remove() is nontrivial; try to write it, but okay if you can't finish

Homework 5 due Monday 4/16

Last time...
  • Map Interface and the ListMap implementation
Lab 10 BST solution up; Hwk 5 due Monday!

Exam II graded: Avg = 81; no curve

- Review today
Hwk 5 due tonight!

Hwk 6 posted -- find partners!
- Due 5/11 (last day of finals week)
- Get an early start on this one...

Today:
- Finish HashMaps (Open Addressing method)
- Discuss HashMaps (Chaining method)
Lab 11 Post-Mortem

- HashMap implementation (using open-addressing)
  - We'll see chaining today
- On average, O(1) operations but we can do better
- Knuth's formula underestimates for higher load factors
- Rehashing required when load factor becomes high

Today:

- Discuss HashMaps (Chaining method)