Hashing
Performance of Set (and Map)

• How long does it take to execute each of the methods of Set2 (similarly Map2), which use a Queue as the data representation?

• Assume that each call to a Queue kernel method executes in constant time, i.e., that the duration of a call is independent of the values of all the arguments, including the receiver
Standard Methods

• For almost every type in the OSU CSE components, including Queue, each of the three Standard methods (newInstance, clear, and transferFrom) takes constant time to execute
  – Exception: Array
Queue Kernel Methods

<table>
<thead>
<tr>
<th>Method (Op)</th>
<th>Execution Time ($T_{Op}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enqueue(x)</code></td>
<td>$T_{enqueue} = c_1$</td>
</tr>
<tr>
<td><code>dequeue(x)</code></td>
<td>$T_{dequeue} = c_2$</td>
</tr>
<tr>
<td><code>length</code></td>
<td>$T_{length} = c_3$</td>
</tr>
</tbody>
</table>
## Set Kernel Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>remove(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>contains(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>size</code></td>
<td></td>
</tr>
</tbody>
</table>
# Set Kernel Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)</td>
<td></td>
</tr>
<tr>
<td>remove(x)</td>
<td></td>
</tr>
<tr>
<td>contains(x)</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
</tr>
</tbody>
</table>

Look at the method body in `Set2`, and figure out how much work it does...
# Set Kernel Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)</td>
<td>$C_4$</td>
</tr>
<tr>
<td>remove(x)</td>
<td></td>
</tr>
<tr>
<td>contains(x)</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
</tr>
</tbody>
</table>

It simply enqueues its argument; plus, there is some constant-time overhead just to make the call to `add`. 
## Set Kernel Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x)</code></td>
<td>( C_4 )</td>
</tr>
<tr>
<td><code>remove(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>contains(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>size</code></td>
<td></td>
</tr>
</tbody>
</table>

Look at the method body in `Set2`, and figure out how much work it does...
## Set Kernel Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x)</code></td>
<td>$c_4$</td>
</tr>
<tr>
<td><code>remove(x)</code></td>
<td>$c_5 \cdot</td>
</tr>
<tr>
<td><code>contains(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>size</code></td>
<td></td>
</tr>
</tbody>
</table>

It has to search through a Queue containing all the Set’s elements.
## Set Kernel Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x)</code></td>
<td>( C_4 )</td>
</tr>
<tr>
<td><code>remove(x)</code></td>
<td>( C_5 \cdot</td>
</tr>
<tr>
<td><code>contains(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>size</code></td>
<td></td>
</tr>
</tbody>
</table>

Raising the question: a worst case, an average case, ...?
## Set Kernel Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)</td>
<td>$c_4$</td>
</tr>
<tr>
<td>remove(x)</td>
<td>$c_5 \cdot</td>
</tr>
<tr>
<td>contains(x)</td>
<td>$c_7 \cdot</td>
</tr>
<tr>
<td>size</td>
<td>$c_9$</td>
</tr>
</tbody>
</table>
Linear Search

• *Linear search* is the algorithm that examines—potentially—every *item* in a collection (e.g., code like `moveToFront` in `Set2` and `Map2`) until it finds what it’s looking for

  – The name reflects the fact that its execution time is a *linear function* of the size of the collection (e.g., \( c_7 \cdot |\text{this}| + c_8 \))
Some Common Execution Times

Execution ("running") time of some code as a function of the "size" of its input.
Some Common Execution Times

T(n)

“Size” of the input for some code.
Some Common Execution Times

Constant time, e.g., $C$
Some Common Execution Times

Log time, e.g.,
\[ a \cdot \log(n) + b \]
Some Common Execution Times

Linear time, e.g.,
\[ a \cdot n + b \]
Some Common Execution Times

$n \log n$ time, e.g., $a \cdot n \cdot \log(n) + b$
Some Common Execution Times

Quadratic time, e.g.,

\[ a \cdot n^2 + b \cdot n + c \]
Some Common Execution Times

Exponential time, e.g., \(2^n\)
Faster Execution?

• Option 1 (preferred): Reduce the order of magnitude of the running time
  – Example: Change from quadratic time to linear time, or linear time to log time

• Option 2 (better than nothing): Reduce the constant factor that multiplies the dominant term of the running time
  – Example: Change from a larger slope for a linear function to a smaller slope
Faster Execution

Reduce by order of magnitude:

\[ a \cdot n + b \]
Faster Execution

Reduce by $\text{order of magnitude: } c \cdot \log(n) + d$
Faster Execution

Reduce by a constant factor: $a \cdot n + b$
Faster Execution

Reduce by a constant factor:

\((a/10) \cdot n + b\)
Example: Faster Linear Search

• Goal: Reduce the constant factor in the execution time of linear search, i.e., reduce it from \( a \cdot n + b \) to something like \( (a/10) \cdot n + b \)

• Approach: Reduce the number of items that need to be examined to find the one you’re looking for, because, e.g.:

\[
(a/10) \cdot n + b = a \cdot (n/10) + b
\]
Hashing: The Intuition

• Instead of searching through all the items, store the items in many smaller buckets and search through only one bucket that
  1. Can be quickly identified, and
  2. Must contain the item you’re looking for for
Hashing: The Intuition

• Instead of searching through *all* the items, store the items in many smaller *buckets* and search through *only one bucket* that
  1. Can be *quickly* identified, and
  2. *Must* contain the item you’re looking for
Hashing: The Intuition

• Instead of searching through *all* the items, store the items in many smaller *buckets* and search through *only one bucket* that
  1. Can be *quickly* identified, and
  2. *Must* contain the item you’re looking for
How To Identify The Bucket

• Suppose you need to search through \( n \) items of type \( T \), and you decide to organize the items into \( m \) buckets

• Given \( x \) of type \( T \), compute from it some integer value \( h(x) \)

• Look in bucket number \( h(x) \mod m \)
How To Identify The Bucket

• Suppose you need to search through \( n \) items of type \( T \), and you decide to organize the items into \( m \) buckets
• Given \( x \) of type \( T \), compute from it some integer value \( h(x) \)
• Look in bucket number \( h(x) \mod m \)

The buckets have indices \( 0, 1, \ldots, m-1 \) in an Array of buckets called a **hashtable**.
How To Identify The Bucket

Suppose you need to search through $n$ items of type $T$, and you decide to organize the items into $m$ buckets.

- Given $x$ of type $T$, compute from it some integer value $h(x)$.
- Look in bucket number $h(x) \mod m$.

The function that maps each value of type $T$ to an integer is called the hash function.
How To Identify The Bucket

By “reducing” the hash function result modulo $m$, you are guaranteed to get the index of some bucket.

- Given $x$ of type $T$, compute from it some integer value $h(x)$.
- Look in bucket number $h(x) \mod m$. 
How To Identify: The Bucket

• Suppose you need to search through $n$ items of type $T$, and you decide to organize the items into $m$ buckets.

• Given $x$ of type $T$, compute from it some integer value $h(x)$.

• Look in bucket number $h(x) \mod m$.

The insight for hashing: if you put the item in this bucket when you store it, then it is the only place you need to look for it when searching.
Set Representation With Hashing

• Suppose the data representation for a new Set implementation, say Set4, uses an instance variable like this:

```java
/**
 * Buckets for hashing.
 */
private Array<Set<T>> hashTable;
```
Abstract Set:

Data representation using several "little Sets":

* Buckets for hashing.

*/

private Array<Set<T>> hashTable;
Set Representation With Hashing

• Suppose the data representation for a new Set implementation, say Set4, uses an instance variable like this:

```java
/**
 * Buckets for hashing.
 */
private Array<Set<T>> hashTable;
```

Can we really do this: use Sets in the representation of a Set? Why is it not circular?
• Suppose further (for illustration purposes) that:
  - $T = \text{Integer}$
  - $h(x) = x$
  - $m = |\text{this.hashTable.entries}| = 3$
## Examples

<table>
<thead>
<tr>
<th>Abstract (this)</th>
<th>Concrete ($this.hashTable.entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{}</code></td>
<td><code>&lt;{}&gt;, {} , {}&gt;</code></td>
</tr>
<tr>
<td><code>{13}</code></td>
<td></td>
</tr>
<tr>
<td><code>{5, 13}</code></td>
<td></td>
</tr>
<tr>
<td><code>{−2, 13}</code></td>
<td></td>
</tr>
</tbody>
</table>
## Examples

<table>
<thead>
<tr>
<th>Abstract (this)</th>
<th>Concrete ($this.hashTable.entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{}</code></td>
<td><code>{}, {}, {}</code></td>
</tr>
<tr>
<td><code>{13}</code></td>
<td><code>{}, {13}, {}</code></td>
</tr>
<tr>
<td><code>{5, 13}</code></td>
<td></td>
</tr>
<tr>
<td><code>{−2, 13}</code></td>
<td></td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th>Abstract (this)</th>
<th>Concrete ($this.hashTable.entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>&lt;{}, {}, {}&gt;</td>
</tr>
<tr>
<td>{13}</td>
<td>&lt;{}, {13}, {}&gt;</td>
</tr>
<tr>
<td>{5, 13}</td>
<td></td>
</tr>
<tr>
<td>{-2, 13}</td>
<td></td>
</tr>
</tbody>
</table>

**Why is 13 in bucket 1?**

\[
h(x) \mod m = h(13) \mod 3 = 13 \mod 3 = 1
\]
# Examples

<table>
<thead>
<tr>
<th>Abstract (this)</th>
<th>Concrete ($this.hashTable.entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>&lt;{}, {}, {}&gt;</td>
</tr>
<tr>
<td>{13}</td>
<td>&lt;{}, {13}, {}&gt;</td>
</tr>
<tr>
<td>{5, 13}</td>
<td>&lt;{}, {13}, {5}&gt;</td>
</tr>
<tr>
<td>{-2, 13}</td>
<td></td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th><strong>Abstract</strong> (this)</th>
<th><strong>Concrete</strong> ($this.hashTable.entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{}</code></td>
<td><code>&lt;{}, {}, {}&gt;</code></td>
</tr>
<tr>
<td><code>{13}</code></td>
<td><code>&lt;{}, {13}, {}&gt;</code></td>
</tr>
<tr>
<td><code>{5, 13}</code></td>
<td><code>&lt;{}, {13}, {5}&gt;</code></td>
</tr>
<tr>
<td><code>{−2, 13}</code></td>
<td><code>&lt;{}, {-2, 13}, {}&gt;</code></td>
</tr>
</tbody>
</table>
Two-Level Thinking

\[
\begin{array}{c}
<\{\}, \{\}, \{\}, > \\
<\{\}, \{\}, \{\}, > \\
<\{\}, \{13\}, \{\}, > \\
<\{\}, \{13\}, \{5\}, > \\
<\{5, 13\}, > \\
<\{-2, 13\}, > \\
\end{array}
\]
The `hashCode` Method

• In Java, the type `Object` defines this instance method to compute $h$, i.e., as the programmatic version of a hash function:

```java
public int hashCode()
```

• As a **best practice**, nearly every type should override the default implementation of this method, which by default rarely meets the requirements necessary for the hashing idea to work!
Requirements

• The only requirement for the hashing idea to give correct behavior is that the hash function \( h \) should be a total function.

• In programming terms:
  - `hashCode` has no precondition (so every `PhoneNumber` value has an `int` hash value).
  - `hashCode` always returns the same `int` hash value for the same `PhoneNumber` value.
Requirements

• The only requirement for the hashing idea to give correct behavior is that the hash function $h$ should be a total function.

In programming terms:

- `hashCode` has no precondition (so every `PhoneNumber` value has an `int` hash value).
- `hashCode` always returns the same `int` hash value for the same `PhoneNumber` value.

This is the part that is not satisfied by the default implementation of `hashCode` that comes with `Object`. 

8 February 2019 OSU CSE
Good Hash Functions

• To result in *good performance* of the hashing idea (not just *correct behavior*), `hashCode` should also:
  – Give different output values for different input values
  – Execute in constant time
Good Hash Functions

• To result in **good performance** of the hashing idea (not just correct **behavior**), `hashCode` should also:
  – Give different output values for different input values
  – Execute in constant time

Why can this not always be achieved?
Suppose type $T$ is PhoneNumber, modeled as follows:

```plaintext
PHONE_NUMBER_MODEL is
  string of integer
exemplar $p$
constraint
  $|p| = 10$ and
entries($p$) is subset of
  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
```
Example

• Suppose type $T$ is `PhoneNumber`, modeled as follows:

```java
PHONE_NUMBER_MODEL = string of integer exemplar $p$
constraint $|p| = 10$ and entries($p$) is subset of
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Maybe you need a `Set<PhoneNumber>` in developing a “contacts app” for a phone.
Possible Hash Functions

• The length of the phone-number string:
  \[ h("6142926446") = 10 \]

• The numerical value of the area code (first three digits) of the phone-number string:
  \[ h("6142926446") = 614 \]

• The numerical value of the last four digits of the phone-number string:
  \[ h("6142926446") = 6446 \]
There are many more options as well; how do you choose one?

- The length of the phone-number string:
  \[ h("6142926446") = 10 \]

- The numerical value of the area code (first three digits) of the phone-number string:
  \[ h("6142926446") = 614 \]

- The numerical value of the last four digits of the phone-number string:
  \[ h("6142926446") = 6446 \]
An Empirical Matter

• How well hashing distributes the data among buckets depends, in part, on the data themselves

• Your worst enemy, knowing your hash function, could always provide data that would result in no performance gain over linear search
  – Everything might fall into one bucket...
Resources

• *Big Java Late Objects*, Sections 16.4.1-16.4.4