Hashing
Performance of \textit{Set} (and \textit{Map})

- How long does it take to execute each of the methods of \texttt{Set2} (similarly \texttt{Map2}), which use a \texttt{Queue} as the data representation?
- Assume that each call to a \texttt{Queue} kernel method executes in \textit{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
## Queue Kernel Methods

<table>
<thead>
<tr>
<th>Method (Op)</th>
<th>Execution Time ($T_{Op}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue ($x$)</td>
<td>$T_{enqueue} = c_1$</td>
</tr>
<tr>
<td>dequeue ($x$)</td>
<td>$T_{dequeue} = c_2$</td>
</tr>
<tr>
<td>length</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>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>
# 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`. 
<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>

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., $c_7 \cdot |this| + c_8$)
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) vs. 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

\[ T(n) \]

\[ n \log n \text{ time}, \text{ e.g.,} \quad a \cdot n \cdot \log(n) + b \]
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 *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
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

• 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 \( h(x) \mod m \)

By “reducing” the hash function result modulo \( m \), you are guaranteed to get the index of some bucket.
How To Identify The Bucket

- Suppose you need to search through $n$ items of type $T$.
- 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 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
private Set<T>[][] hashTable;
```

Data representation using several “little `Sets`”:

* Buckets for hashing.

* /

Abstract `Set`:
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 Set<T>[] hashTable;
```

Can we really do this: use Sets in the representation of a Set? Why is it not circular?
Details

• Suppose further (for illustration purposes) that:

- $T = \text{Integer}$
- $h(x) = x$
- $m = |\$this.hashTable| = 3$
• Suppose further (for illustration purposes) that:
  \(- T = \text{Integer} \)
  \(- h(x) = x \)
  \(- m = |\$this.hashTable| = 3 \)

Here and in upcoming contracts, we’ll model Java arrays as mathematical *strings*.
## Examples

<table>
<thead>
<tr>
<th>Abstract (this)</th>
<th>Concrete ($this.hashTable)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{}</code></td>
<td><code>&lt;{}, {}, {}&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)</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>
Abstract (this)

<table>
<thead>
<tr>
<th></th>
<th>&lt;{}, {}, {}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{}</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)</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></td>
</tr>
</tbody>
</table>
## Examples

<table>
<thead>
<tr>
<th><strong>Abstract (this)</strong></th>
<th><strong>Concrete ($this.hashTable)</strong></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

\[
\begin{align*}
\langle \{\}, \{\}, \{\} \rangle & \rightarrow \langle \{\}, \{-2, 13\}, \{\} \rangle \\
\langle \{\}, \{13\}, \{\} \rangle & \rightarrow \langle \{\}, \{13\}, \{5\} \rangle
\end{align*}
\]
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
The only requirement for the hashing idea to give correct behavior is that the hash function 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
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?
Example

• Suppose type $T$ is $\text{PhoneNumber}$, modeled as follows:

$$
\text{PHONE\_NUMBER\_MODEL is string of integer}
$$

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:
  \[ h("6142926446") = 6446 \]
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?
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 (4th ed)*, Sections 16.3-16.4
  – [https://library.ohio-state.edu/record=b8540788~S7](https://library.ohio-state.edu/record=b8540788~S7)