Hashing

Kyle Pierce
thanks to Kyle Pierce & Marty Stepp
Why Hashing?

- used to implement structures like Java’s HashMap and HashSet
  - no guarantee about ordering of elements
  - constant-time add, contains, and remove methods
  - can store any type of Object
Why Hashing?

- used to implement structures like Java’s `HashMap` and `HashSet`
  - no guarantees about ordering of elements
  - constant-time add, contains, and remove methods
  - can store any type of `Object`

how is this possible?
Arrays

- **Good:** it takes $O(1)$ time to add or access at an index
- **Bad:** it takes $O(n)$ time to check if an (unsorted) array contains an element
Arrays

- **Good:** it takes $O(1)$ time to add or access at an index
- **Bad:** it takes $O(n)$ time to check if an (unsorted) array contains an element

how can we fix this?
Arrays

- **Good:** it takes O(1) time to add or access at an index
- **Bad:** it takes O(n) time to check if an (unsorted) array contains an element

How can we fix this?

What if we knew the index the element *would* be at?
Hash Functions

**Hash**: to map a value to an index

**Hash Table**: array that stores elements at hashed indices

**Hash Function**: an algorithm that maps values to indices

One possible hash function:

$$\text{hash}(i) = i \mod \text{table.length}$$

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>
Using our Hash Function

```java
public static int hash(int i) {
    return Math.abs(i) % table.length;
}
```

Add to table: `table[hash(i)] = i;`
Search table: `table[hash(i)] == i`
Remove from table: `table[hash(i)] = 0;`

What are the runtimes of these?
Using our Hash Function

```java
public static int hash(int i) {
    return Math.abs(i) % table.length;
}
```

Add to table: `table[hash(i)] = i;`
Search table: `table[hash(i)] == i`
Remove from table: `table[hash(i)] = 0;`

What are the runtimes of these? \(O(1)\)
Hash Functions (continued)

**Hash**: to map a value to an index

**Hash Table**: array that stores elements at hashed indices

**Hash Function**: an algorithm that maps values to indices

What makes a good hash function?

- spread out from 0 to `table.length`
  - will help minimize collisions
- hash of a value is always the same
  - otherwise can’t find anything
- should be fast to compute
Hashing Objects

- all Java objects have a built-in `hashCode()` method that we can call

```java
// returns an integer hash code for this object
public int hashCode() {
    ...
}
```

- how is it implemented?
  - depends on the type of object and its fields
  - you can define the `hashCode()` method in classes you write
Hashing Strings

- this is what the `hashCode()` method for Strings looks like:

```java
// returns an integer hash code for this object
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
}
```

- some Strings still map to the same hash -- a “collision”
  e.g. “Ea” and “FB”
Using our (new) Hash Function

```java
public static int hash(E e) {
    return Math.abs(e.hashCode()) % table.length;
}
```

Add to table: `table[hash(e)] = e;`
Search table: `table[hash(e)].equals(e)`
Remove from table: `table[hash(e)] = null;`
Collisions

**Collision**: when a hash function maps two values to the same index

**Collision Resolution**: an algorithm for fixing collisions

```plaintext
hash(i) = i % table.length

set.add(11)
set.add(49)
set.add(24)
set.add(7)
set.add(54)  // collides with 24
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>
Chaining

- resolve collisions by storing a list at each index
  - add/search/remove have to traverse lists, but we will keep them short

```
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
11
14
54
24
7
49
```
Rehashing: growing into a larger array when the table becomes too full
  ● cannot simply copy over the array (why not?)

Load Factor: ratio of (# elements) / (hash table length)
  ● typically rehash when load factor $\approx 0.75$
  ● large prime as hash table length reduces collisions