Hashing

- **hash**: To map a value to an integer index.
  - **hash table**: An array that stores elements via hashing.

- **hash function**: An algorithm that maps values to indexes.
  - one possible hash function for integers: \( HF(I) = I \% \text{length} \)

```java
set.add(11); // 11 % 10 == 1
set.add(49); // 49 % 10 == 9
set.add(24); // 24 % 10 == 4
set.add(7);  // 7 % 10 == 7
```

<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>
Hash Functions

• Maps an object to a number
  • result should be constrained to some range
  • passing in the same object should always give the same result

• Results from a hash function should be distributed over a range
  • very bad if everything hashes to 1!
  • should "look random"

• How would we write a hash function for String objects?
String's hashCode

- The `hashCode` function inside `String` objects looks like this:

```java
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
    return hash;
}
```

- As with any general hashing function, collisions are possible.
  - Example: "Ea" and "FB" have the same hash value.

- Early versions of Java examined only the first 16 characters. For some common data this led to poor hash table performance.
Collisions

- **collision**: When hash function maps 2 values to same index.

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

- **collision resolution**: An algorithm for fixing collisions.

<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>54</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>
Probing

- **probing**: Resolving a collision by moving to another index.
  - **linear probing**: Moves to the next index.

```java
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);
set.add(54); // collides with 24; must probe
```

<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>54</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>

- Is this a good approach?
  - variation: **quadratic probing** moves increasingly far away
Chaining

- **chaining**: Resolving collisions by storing a list at each index.
  - add/search/remove must traverse lists, but the lists are short
  - impossible to "run out" of indexes, unlike with probing

<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>
<tr>
<td></td>
<td>11</td>
<td>24</td>
<td>7</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Rehashing**

- **rehash**: Growing to a larger array when the table is too full.
  - Cannot simply copy the old array to a new one. (Why not?)

- **load factor**: ratio of \( \text{(# of elements)} / \text{(hash table length)} \)
  - many collections rehash when load factor \( \approx .75 \)
  - can use big prime numbers as hash table sizes to reduce collisions

<table>
<thead>
<tr>
<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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>24</td>
<td>7</td>
<td>49</td>
<td>11</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
</tbody>
</table>
Implementing a hash map

A hash map is just a set where the lists store key/value pairs:

```java
// key    value
map.put("Marty", 14);
map.put("Jeff", 21);
map.put("Kasey", 20);
map.put("Stef", 35);
```

- Instead of a `List<Integer>`, write an inner `Entry` node class with `key` and `value` fields; the map stores a `List<Entry>`