Efficiency

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>remove</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted array</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>sorted array</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logN)</td>
</tr>
<tr>
<td>unsorted linked list</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>sorted linked list</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>binary search tree</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>O(logN)</td>
</tr>
<tr>
<td>hash table</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Hash Functions

- Maps a key to a number
  - result should be constrained to some range
  - passing in the same key should always give the same result

- Keys 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?
Hashing objects

• All Java objects contain the following method:

```
public int hashCode()
```
Returns an integer hash code for this object.

• We can call `hashCode` on any object to find its preferred index.

• How is `hashCode` implemented?
  • Depends on the type of object and its state.
    • Example: a String's `hashCode` adds the ASCII values of its letters.
  • You can write your own `hashCode` methods in classes you write.
    • All classes come with a default version based on memory address.
The `hashCode` function inside `String` objects could look 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. For 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><strong>54</strong></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
Clustering

- **clustering**: Clumps of elements at neighboring indexes.
  - slows down the hash table lookup; you must loop through them.

```java
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);
set.add(14); // collides with 24
set.add(14); // collides with 24, then 54
set.add(86); // collides with 14, then 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>54</td>
<td>14</td>
<td>7</td>
<td>86</td>
<td>49</td>
</tr>
</tbody>
</table>

- How many indexes must a lookup for 94 visit?
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>54</td>
<td>14</td>
<td>24</td>
<td>7</td>
<td>49</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 11
- 24
- 54
- 14
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 (# of elements) / (hash table length)
  - many collections rehash when load factor $\approx .75$
  - can use big prime numbers as hash table sizes to reduce collisions

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

- 24
- 7
- 49
- 11
- 54
- 14
Rehashing code

... // Grows hash array to twice its original size.
private void rehash() {
    List<Integer>[] oldElements = elements;
    elements = (List<Integer>[]) new List[2 * elements.length];
    for (List<Integer> list : oldElements) {
        if (list != null) {
            for (int element : list) {
                add(element);
            }
        }
    }
}
Other questions

- How would we implement `toString` on a `HashSet`?

<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>

current element: 24
current index: 4
sub-index: 0
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>`. 