GOOD MORNING HASH-BROWN

HASHING
Run times of common Set operations

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>contains(element)</th>
<th>add(element)</th>
<th>remove(element)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted ArrayList</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsorted LinkedList</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arrays

- **Pros:** $O(1)$ time to set() or get() at a given index
- **Cons:** $O(n)$ time to see if an element is in the array

What if we *knew* what index an object would be at?
Hash Function

- A function that maps any input deterministically to some output
  - If two objects are “equal”, their hash function must produce the same value
- We are concerned specifically with a hash function that maps Object -> int
- All Java Objects have a hashCode() method!

"Spongebob".hashCode() == 907493499
"Patrick".hashCode() == 873506786
"Squidward".hashCode() == -759989618
Hash Table

- Array where we store elements at their hashed indexes

```java
String[] hashTable = new String[10]
```

<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>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
```

Where should these Strings go?

"Spongebob".hashCode() == 907493499
"Patrick".hashCode() == 873506786
"Squidward".hashCode() == -759989618

```java
int index = Math.abs(hashcode % hashTable.length)
```
public static int hashCode(E element) {
    return Math.abs(element.hashCode() % hashTable.length);
}

contains(element): return hashTable[hashIndex(element)] != null
add(element)     : hashTable[hashIndex(element)] = element
remove(element)  : hashTable[hashIndex(element)] = null
What issues do we have?

Two elements might hash to the same spot!
This is called a collision
What Makes a Hash Function Good?

- To avoid collisions, we want the elements to be evenly spread out
  - We want the hash function to appear random

Rank these Hash Functions!

```java
// Returns the length of the given String
public int hash(String s) {
    return s.length;
}

// Returns 0
public int hash(String s) {
    return 0;
}

// Returns the sum of the ascii values of
// the characters in the given string
public int hash(String s) {
    int hash = 0;
    for (int i = 0; i < s.length(); i++) {
        hash += (int) s.charAt(i);
    }
    return hash;
}

// Returns a random number between 0 and 100000
public int hash(String s) {
    Random r = new Random();
    return r.nextInt(100000);
}
```
What Makes a Hash Function Good? Java’s String hashCode()

```java
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```
What issues do we have?

Two elements might hash to the same spot!
This is called a collision

We can only have 10 elements!
Break and Discuss!
Separate Chaining

- Solve collisions \textit{and} running out of space by storing a list at each index!
  - contains/add/remove must now traverse lists
Is this really $O(1)$ though?

How long do you expect the average chain to be if there are 30 elements in a hash table of size 10?

**Load Factor** : $(\# \text{ of elements in hash table}) / (\text{length of hash table})$

As long as we limit the length of each chain to a constant number, it will be $O(1)$!
Rehashing

- **Load Factor**: \( \frac{\text{# of elements in hash table}}{\text{length of hash table}} \)
  - The length of the average chain

- **Rehashing**: Once the load factor becomes too high, we hash everything again into a bigger array
  - Usually rehash when load factor is around 0.75
  - Why can’t we copy into the new array?

This is **Amortized** \( O(1) \)