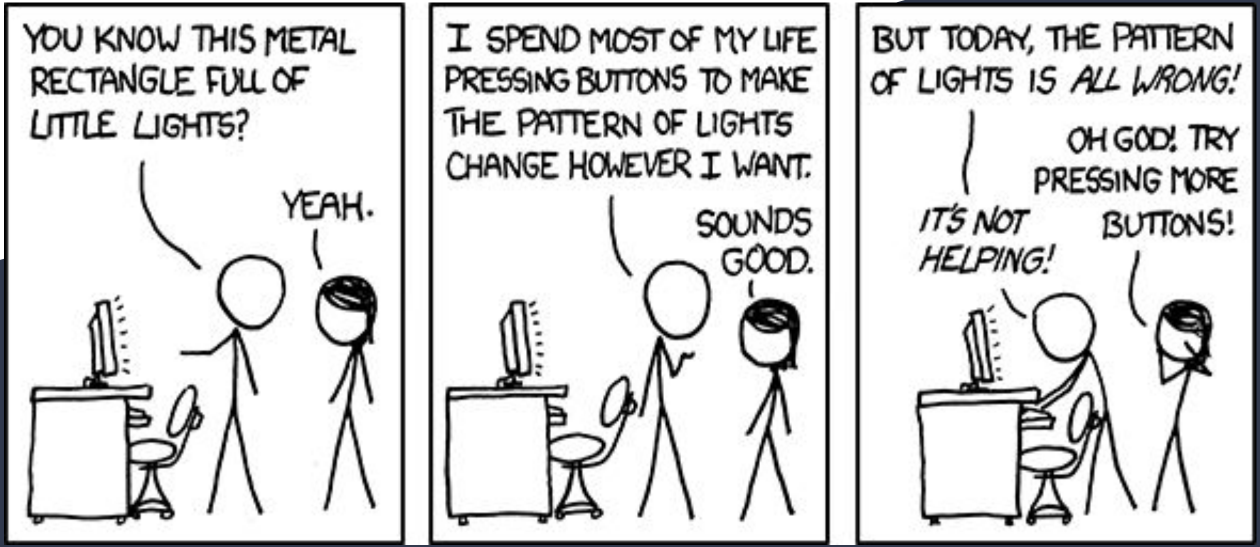
Hashing

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:

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

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

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49

Using our Hash Function

```
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

```
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:

```
// 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*

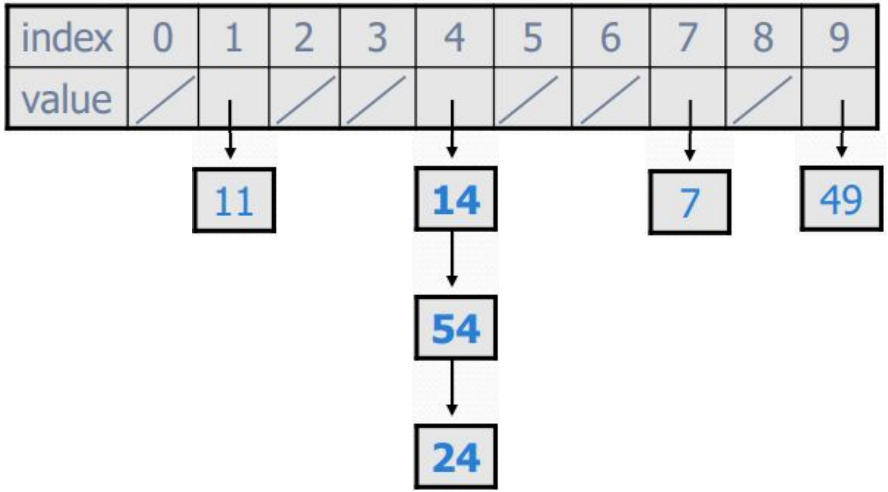
```
hash(i) = i % table.length
```

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

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49

Chaining

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



Rehashing

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 ≅ 0.75
- large prime as hash table length reduces collisions