CSE 373

Hash set implementation, continued

read: Weiss 5.1 - 5.2, 5.4, 5.5

slides created by Marty Stepp
http://www.cs.washington.edu/373/

The remove operation

- We cannot remove by simply zeroing out an element:

```
set.remove(54);     // set index 5 to 0
set.contains(14)    // false???  oops
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	**0**	14	34	0	49
size	5									

- Instead, we replace it by a special "removed" placeholder value
  - (can be re-used on `add`, but keep searching on `contains`)

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	XX	14	34	0	49
size	5									

Implementing remove

```
public void remove(int value) {
    int h = hash(value);
    while (elements[h] != 0 && elements[h] != value) {
        h = (h + 1) % elements.length;
    }
    if (elements[h] == value) {
        elements[h] = -999;   // "removed" flag value
        size--;
    }
}
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	-999	14	34	0	49
size	5									

```
set.remove(54);   // client code
set.remove(11);
set.remove(34);
```

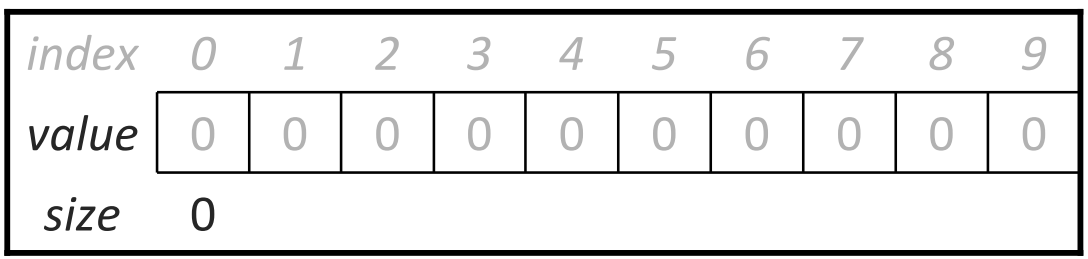
Patching add, contains

```
private static final int REMOVED = -999;

public void add(int value) {
    int h = hash(value);
    while (elements[h] != 0 && elements[h] != value &&
           elements[h] != REMOVED) {
        h = (h + 1) % elements.length;
    }
    if (elements[h] != value) {
        elements[h] = value;
        size++;
    }
}

// contains does not need patching;
// it should keep going on a -999, which it already does
public boolean contains(int value) {
    int h = hash(value);
    while (elements[h] != 0 && elements[h] != value) {
        h = (h + 1) % elements.length;
    }
    return elements[h] == value;
}
```

Problem: full array

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

```
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);  // collides with 24
set.add(14);  // collides with 24, then 54
set.add(86);  // collides with 14, then 37
```



- Where does each value go in the array?
- How many indexes must be examined to answer `contains(94)`?
- What will happen if the array completely fills?

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 ≅ .75

index	0	1	2	3	4	5	6	7	8	9
value	95	11	0	0	24	54	14	37	66	48
size	8									

index	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
value	0	0	0	0	24	0	66	0	48	0	0	11	0	0	54	95	14	37	0	0
size	8																			

Implementing rehash

```
// Grows hash table to twice its original size.
private void rehash() {
    int[] old = elements;
    elements = new int[2 * old.length];
    size = 0;
    for (int value : old) {
        if (value != 0 && value != REMOVED) {
            add(value);
        }
    }
}

public void add(int value) {
    if ((double) size / elements.length >= 0.75) {
        rehash();
    }
    ...
}
```

Hash table sizes

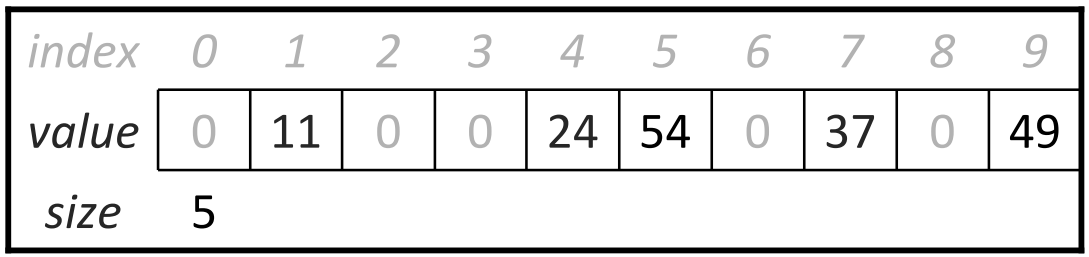
- Can use prime numbers as hash table sizes to reduce collisions.
- Also improves spread / reduces clustering on rehash.

```
set.add(11);  //  11 % 13 == 11
set.add(39);  //  39 % 13 ==  0
set.add(21);  //  21 % 13 ==  8
set.add(29);  //  29 % 13 ==  3
set.add(71);  //  81 % 13 ==  6
set.add(41);  //  41 % 13 ==  2
set.add(99);  // 101 % 13 == 10
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	39	0	41	29	0	0	71	0	21	0	101	11	0
size	7												

Other details

- How would we implement `toString` on our `HashIntSet`?



index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	54	0	37	0	49
size	5									

```
System.out.println(set);
// [11, 24, 54, 37, 49]
```