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 % 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

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:

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

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

  - 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.

```
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.

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	54	0	0	7	0	49

Probing

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

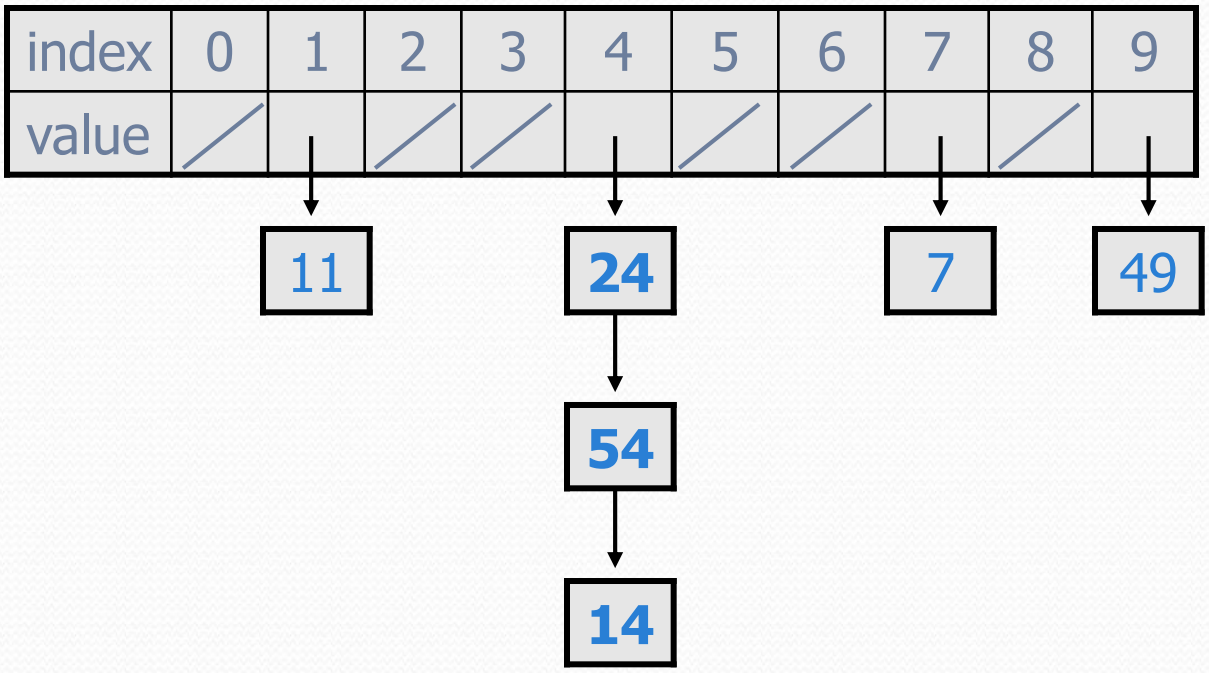
```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);   // collides with 24; must probe
```

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

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

Chaining

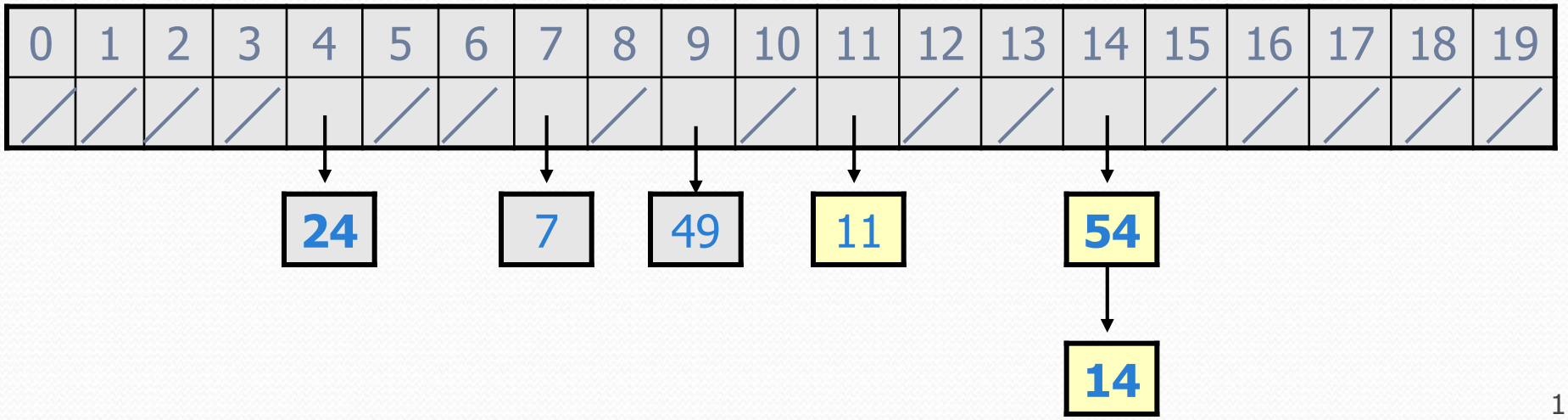
- **chaining**: Resolving collisions by storing a list at each index
  - add/search/remove must traverse lists, but the lists are short
  - impossible to "run out" of indexes, unlike with probing



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?)

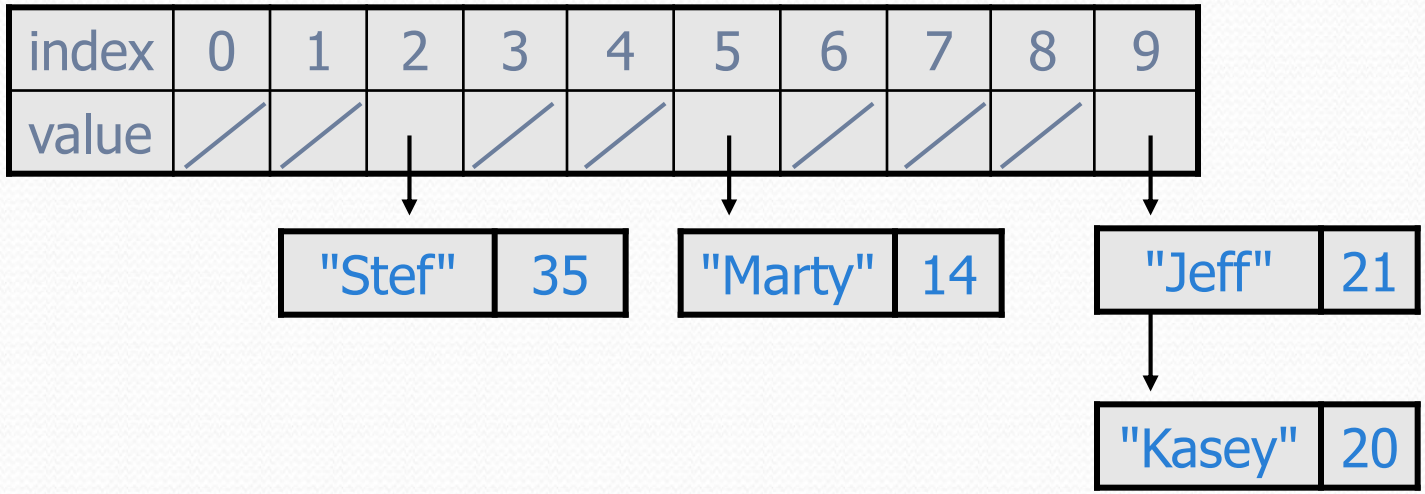
- **load factor**: ratio of (*# of elements*) / (*hash table length*)
  - many collections rehash when load factor ≅ .75
  - can use big prime numbers as hash table sizes to reduce collisions



Implementing a hash map

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

```
//          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>`