0	
1	
2	
3	

A HashTable<Key, Value> using Separate Chaining has:
- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):


  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket



Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Example:
Start buckets array with size 4
Use string length as the hash function

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

How many elements in bucket 0?
A: 0 B: 1 C: 2 D: 3 E: more than 3

How many elements in bucket 1?
A: 0 B: 1 C: 2 D: 3 E: more than 3

How many elements in bucket 2?
A: 0 B: 1 C: 2 D: 3 E: more than 3

How many **entries** are checked for `get("purplish")`?
A: 1 B: 2 C: 3 D: 4 E: more than 4

0	
1	
2	
3	

0	
1	
2	
3	
4	
5	
6	
7	

```
void set(key, value):

  if ____________________________: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

Example:
Start buckets array with size 4
Use string length as the hash function

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

```
int hash1(String s) {
    return s.length();
}


int hash2(String s) {
    int hash = 0;
    for(int i = 0; i < s.length(); i += 1) {
        hash += Character.codePointAt(s, i);
    }
    return hash;
}



public int hash3(String s) {
    int h = 0;
    for (int i = 0; i < s.length(); i++) {
        h = 31 * h + Character.codePointAt(s, i);
    }
    return h;
}
```

"dog" "dim"

"dog" "log"

"ok" "ALM"

"unheavenly"
"hypoplankton"

All strings (arbitrarily many!): "" ... "a" "b" ... "aa" "ab" ... "zz" ... "aaa" "aab" ... "zzzzzzzz..." ...

Hashed to 2^32 ints: - 2147483648 - 2147483647 ... - 1001 - 1000 ... - 2 - 1 0 1 2 ... 1000 1001 ... 2147483647

Made into array indices by %: 0 1 2 3 4 5 6 7 8 9 10 ...

Further Exploration:

(Just for fun and profit!)

1. Write an method to determine if two Strings are anagrams of one another. (EX: “rats” and ‘star” are anagrams).
    a. What is the runtime of your algorithm? How did HashMap help, compared to a more naive approach?
    b. Follow-up: Can you do this without using any built-in Java libraries like HashMap? (Hint: what do you know about what makes up a String?)

2. A *perfect hash function* is one that uniquely maps each key to a value. Write a method that, given a set of Strings, creates a perfect hash function, i.e. maps each String to a unique Integer key.
    a. Follow-up: What if you were to add a new String to your list? How long does your algorithm take to update the key-value pairs?
    b. How does your hash function handle Strings *not* in your set?

3. Another way to resolve collisions in a hash table is to use multiple hash functions, i.e. h_1, h_2, ...h_n, where if an index is already occupied, we’ll try the next hash function to determine a new index to place our element, and so on.
    a. How does this compare to the other collision resolution strategies we discussed in class? What is the runtime to check if an element is in our hash table?
    b. Is this enough to resolve all potential collisions? Would adding more hash functions solve the problem?