LECTURE 10

MONDAY OCTOBER 7

Testing Equality of Person/PersonCollector in JUnit (3)

```
pc1.addPerson(p1);
assertFalse(pc1.equals(pc2));

pc2.addPerson(p2);
assertFalse(pc1.persons[0] == pc2.persons[0]);
assertTrue(pc1.persons[0].equals(pc2.persons[0]));
assertTrue(pc1.equals(pc2));

pc1.addPerson(p3); pc2.addPerson(p4);
assertTrue(pc1.persons[1] == pc2.persons[1]);
assertTrue(pc1.persons[1].equals(pc2.persons[1]));
assertTrue(pc1.equals(pc2));
```

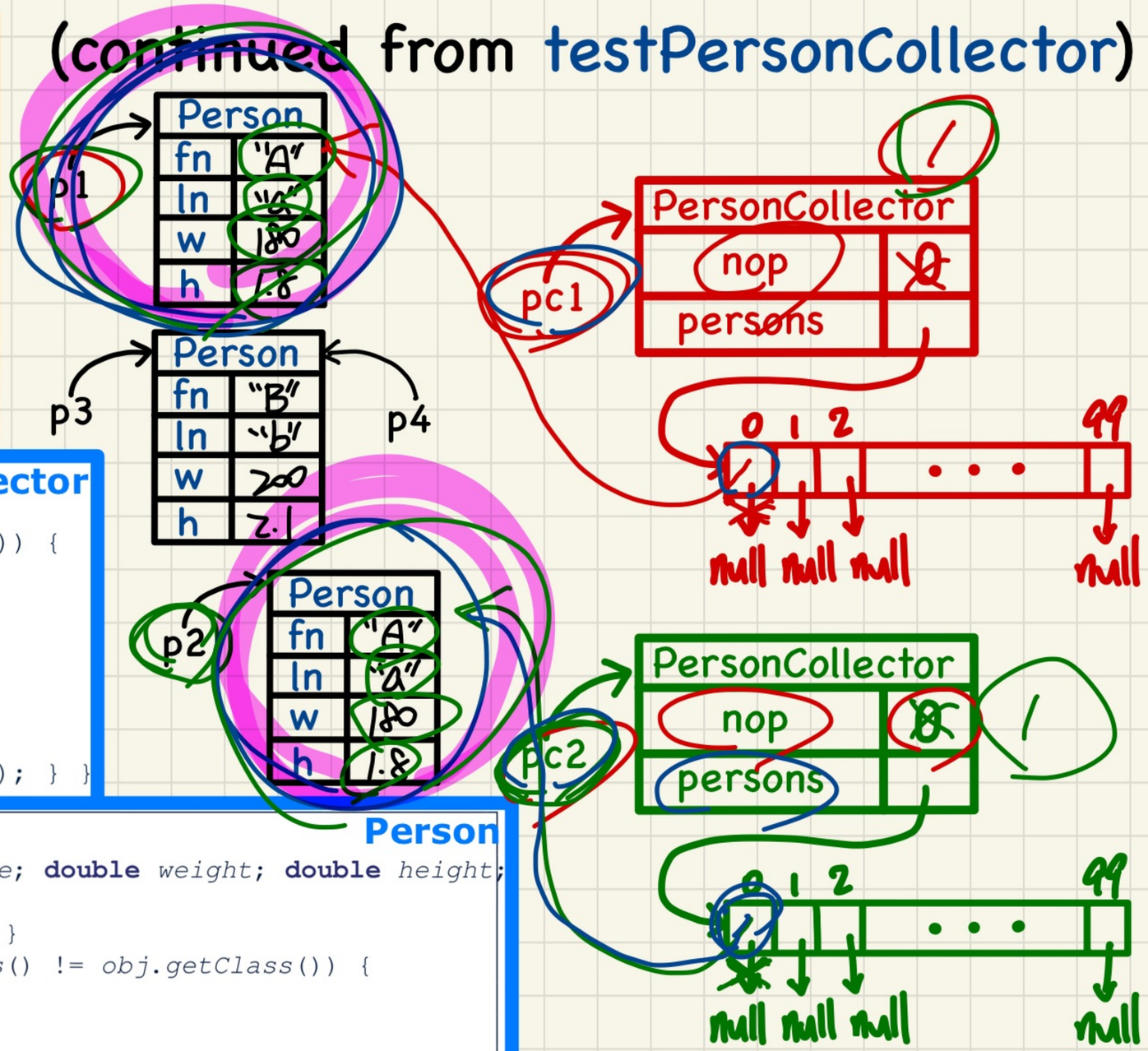
PersonCollector

```
1  boolean equals(Object obj) {
2    if(this == obj) { return true; }
3    if(obj == null || this.getClass() != obj.getClass()) {
4      return false; }
5    PersonCollector other = (PersonCollector) obj;
6    boolean equal = false;
7    if(this.nop == other.nop) {
8      equal = true;
9      for(int i = 0; equal && i < this.nop; i ++) {
10       equal = this.persons[i].equals(other.persons[i]); } }
11   return equal;
12 }
```

Person

```
1  class Person {
2    String firstName; String lastName; double weight; double height;
3    boolean equals(Object obj) {
4      if(this == obj) { return true; }
5      if(obj == null || this.getClass() != obj.getClass()) {
6        return false; }
7      Person other = (Person) obj;
8      return
9          this.weight == other.weight && this.height == other.height
10       && this.firstName.equals(other.firstName)
11       && this.lastName.equals(other.lastName); } }
```

(continued from testPersonCollector)



Ordering between Employees

name	id	salary
alan	2	4500.31
mark	3	3450.67
tom	1	3450.67

tom.id < alan.id < mark.id

mark.salary = mark.salary < alan.salary

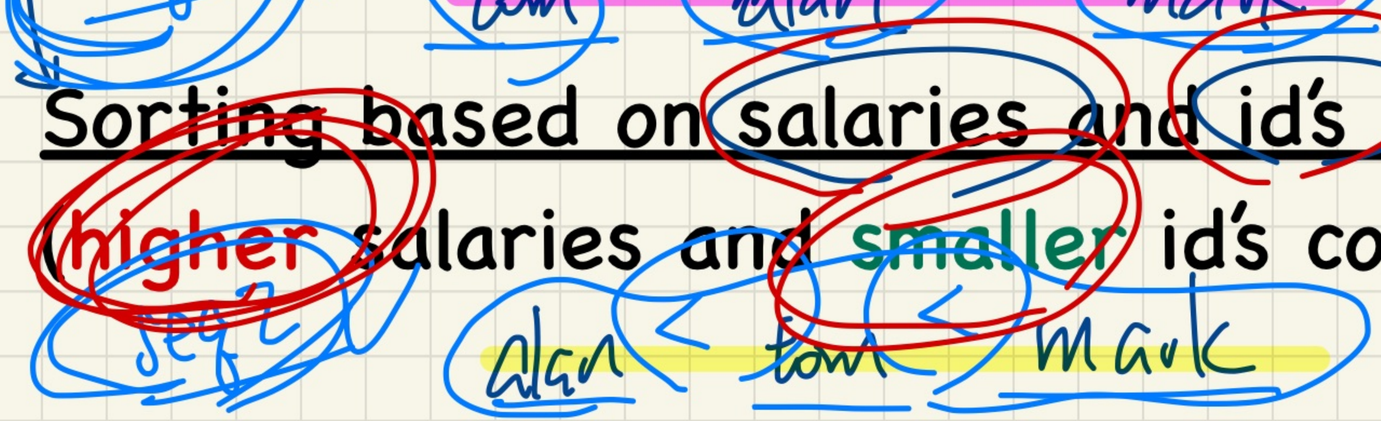
Sorting: from smallest to largest

Sorting based on id's

(smaller id's come first)

Sorting based on salaries and id's

(higher salaries and smaller id's come first)



Unknown Ordering between Employees

Say: Sorting based on ~~salaries and~~ id's

(higher salaries and smaller id's come first)

```
class Employee {
  int id; double salary;
  Employee(int id) { this.id = id; }
  void setSalary(double salary) { this.salary = salary; } }
```

```
1  @Test
2  public void testUncomparableEmployees() {
3    Employee alan = new Employee(2);
4    Employee mark = new Employee(3);
5    Employee tom = new Employee(1);
6    Employee[] es = {alan, mark, tom};
7    Arrays.sort(es);
8    Employee[] expected = {tom, alan, mark};
9    assertArrayEquals(expected, es); }
```

Comparable Employees: Version 1

```
class CEmployee1  implements Comparable<CEmployee1> {
  ... /* attributes, constructor, mutator similar to Employee */
  @Override
  public int compareTo(CEmployee1 e) { return this.id - e.id; }
}
```

```
@Test
public void testComparableEmployees_1() {
  /*
   * CEmployee1 implements the Comparable interface.
   * Method compareTo compares id's only.
   */
  CEmployee1 alan = new CEmployee1(2);
  CEmployee1 mark = new CEmployee1(3);
  CEmployee1 tom = new CEmployee1(1);
  alan.setSalary(4500.34);
  mark.setSalary(3450.67);
  tom.setSalary(3450.67);
  CEmployee1[] es = {alan, mark, tom};
  /* When comparing employees,
   * their salaries are irrelevant.
   */
  Arrays.sort(es);
  CEmployee1[] expected = {tom, alan, mark};
  assertArrayEquals(expected, es);
}
```

alan

CEmployee1	
id	2
salary	4500

mark

CEmployee1	
id	3
salary	3450

tom

CEmployee1	
id	1
salary	3450

compareTo	alan	mark	tom
alan	0	<	
mark			
tom			

Comparable Employees: Version 2.1

```
1  class CEmployee2  implements Comparable<CEmployee2> {
2    ... /* attributes, constructor, mutator similar to Employee */
3    @Override
4    public int compareTo(CEmployee2 other) {
5      if (this.salary > other.salary) {
6        return -1;
7      }
8      else if (this.salary < other.salary) {
9        return 1;
10     }
11     else { /* equal salaries */
12       return this.id - other.id;
13     }
14   }
```

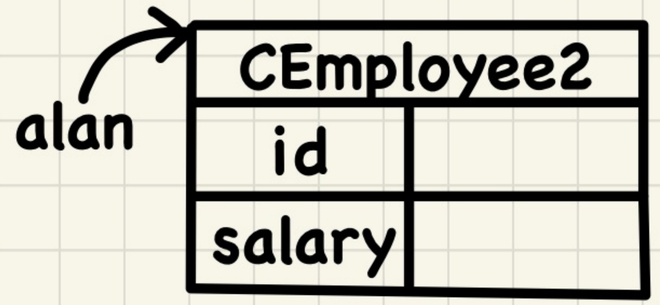
this should come first

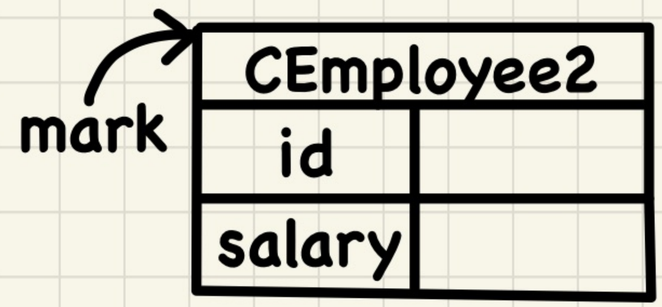
Why not -1 ?

-1

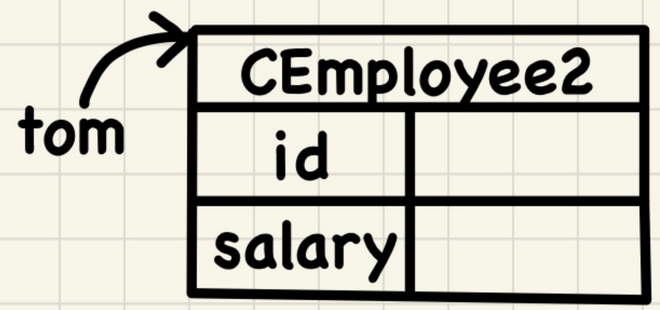
salary is a stronger criterion to consider first.

2 3






```
1  @Test
2  public void testComparableEmployees_2() {
3    /*
4     * CEmployee2 implements the Comparable interface.
5     * Method compareTo first compares salaries, then
6     * compares id's for employees with equal salaries.
7     */
8    CEmployee2 alan = new CEmployee2(2);
9    CEmployee2 mark = new CEmployee2(3);
10   CEmployee2 tom = new CEmployee2(1);
11   alan.setSalary(4500.34);
12   mark.setSalary(3450.67);
13   tom.setSalary(3450.67);
14   CEmployee2[] es = {alan, mark, tom};
15   Arrays.sort(es);
16   CEmployee2[] expected = {alan, tom, mark};
17   assertArrayEquals(expected, es);
18 }
```



compareTo	alan	mark	tom
alan			
mark			
tom			

Smaller id's come first.

If equal id's, then higher salary come first.

id	salary
A	2
	45000
B	1
	30000

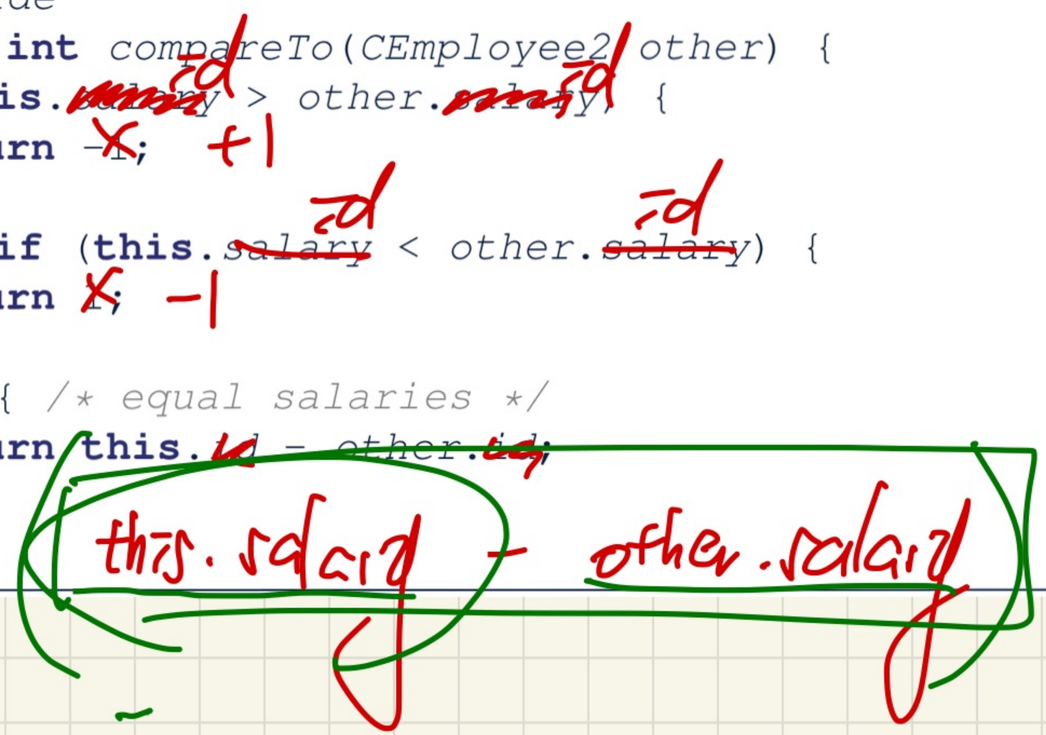
original criterion:

A B

revised criterion:

B A

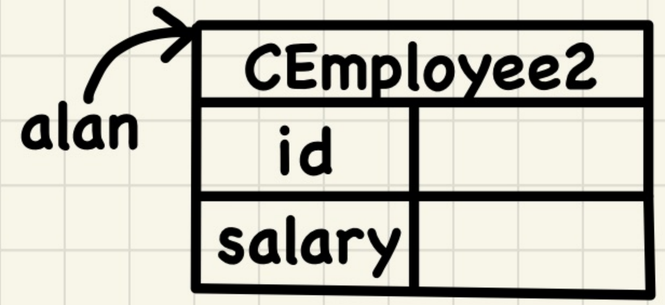
```
1  class CEmployee2  implements Comparable<CEmployee2> {
2    ... /* attributes, constructor, mutator similar to Employee */
3    @Override
4    public int compareTo(CEmployee2 other) {
5      if(this.salary > other.salary) {
6        return -1;
7      }
8      else if (this.salary < other.salary) {
9        return 1;
10     }
11   else { /* equal salaries */
12     return this.id - other.id;
13   }
14 }
```

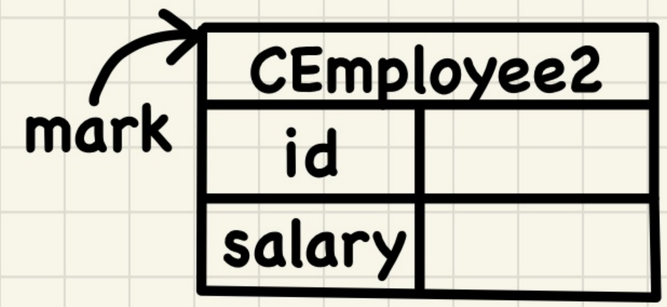



```
if( this.s - other.s < 0){
    return +1;
```

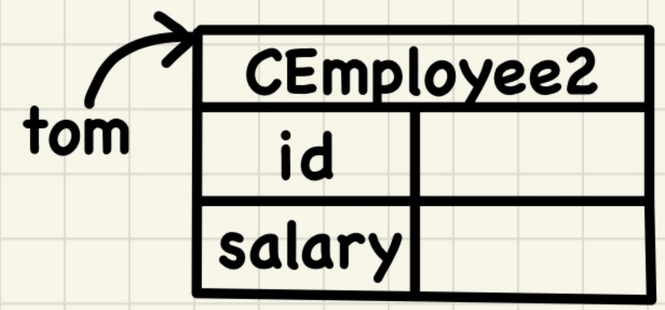
Comparable Employees: Version 2.2

```
1  class CEmployee2  implements Comparable<CEmployee2> {
2    ... /* attributes, constructor, mutator similar to Employee */
3    @Override
4    public int compareTo(CEmployee2 other) {
5      int salaryDiff = Double.compare(this.salary, other.salary);
6      int idDiff = this.id - other.id;
7      if(salaryDiff != 0) { return -salaryDiff; }
8      else { return idDiff; } } }
```






```
1  @Test
2  public void testComparableEmployees_2() {
3    /*
4     * CEmployee2 implements the Comparable interface.
5     * Method compareTo first compares salaries, then
6     * compares id's for employees with equal salaries.
7     */
8    CEmployee2 alan = new CEmployee2(2);
9    CEmployee2 mark = new CEmployee2(3);
10   CEmployee2 tom = new CEmployee2(1);
11   alan.setSalary(4500.34);
12   mark.setSalary(3450.67);
13   tom.setSalary(3450.67);
14   CEmployee2[] es = {alan, mark, tom};
15   Arrays.sort(es);
16   CEmployee2[] expected = {alan, tom, mark};
17   assertArrayEquals(expected, es);
18 }
```



compareTo	alan	mark	tom
alan			
mark			
tom			

Design Principles of the compareTo Method

4 < 2

2 < 4

Asymmetric:

c1 < c2 c2 < c1

negation

$$\neg(c1.compareTo(c2) < 0 \wedge c2.compareTo(c1) < 0)$$
$$\neg(c1.compareTo(c2) > 0 \wedge c2.compareTo(c1) > 0)$$

i < j j < k

↳ i < k

Transitive: c1 < c2 c2 < c3 c1 < c3

$$c1.compareTo(c2) < 0 \wedge c2.compareTo(c3) < 0 \Rightarrow c1.compareTo(c3) < 0$$
$$c1.compareTo(c2) > 0 \wedge c2.compareTo(c3) > 0 \Rightarrow c1.compareTo(c3) > 0$$

a R b
b S c
c P a

a < b
b < c

a < c X

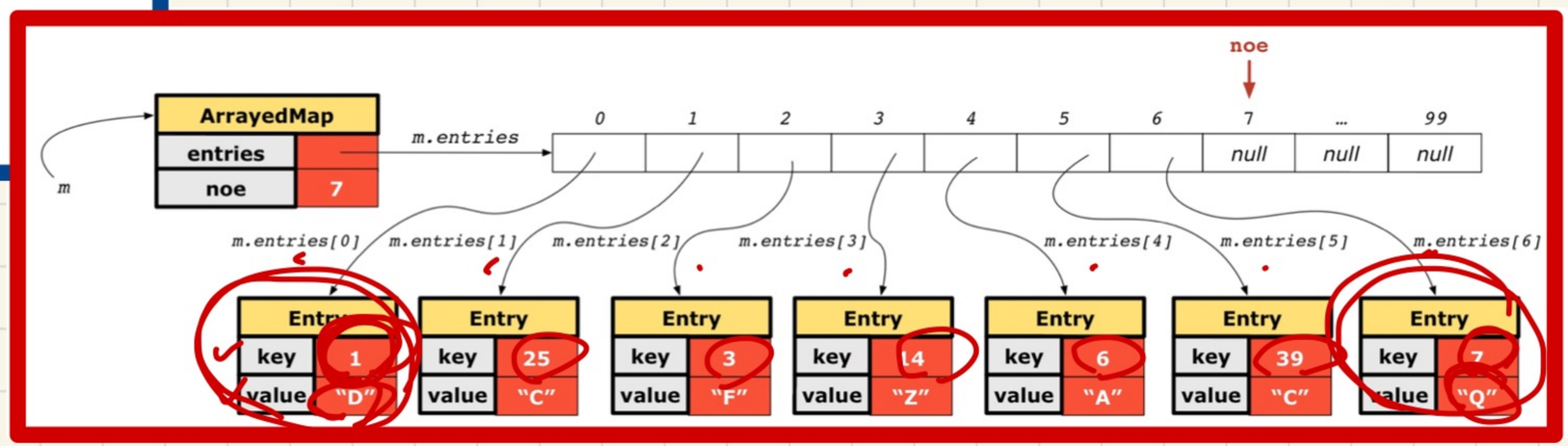
```java
public class Entry {
  private int key;
  private String value;

  public Entry(int key, String value) {
    this.key = key;
    this.value = value;
  }
}
```

```java
public class ArrayedMap {
  private final int MAX_CAPCAITY = 100;
  private Entry[] entries;
  private int noe; /* number of entries */
  public ArrayedMap() {
    entries = new Entry[MAX_CAPCAITY];
    noe = 0;
  }
  public int size() {
    return noe;
  }
  public void put(int key, String value) {
    Entry e = new Entry(key, value);
    entries[noe] = e;
    noe ++;
  }
```

```java
@Test
public void testArrayedMap() {
  ArrayedMap m = new ArrayedMap();
  assertTrue(m.size() == 0);
  m.put(1, "D");
  m.put(25, "C");
  m.put(3, "F");
  m.put(14, "Z");
  m.put(6, "A");
  m.put(39, "C");
  m.put(7, "Q");
  assertTrue(m.size() == 7);
  /* inquiries of existing key */
  assertTrue(m.get(1).equals("D"));
  assertTrue(m.get(7).equals("Q"));
  /* inquiry of non-existing key */
  assertTrue(m.get(31) == null);
}
```

no duplicates of keys



Naive Implementation of a Map

Naive Implementation of a Map: Retrieval of an Entry

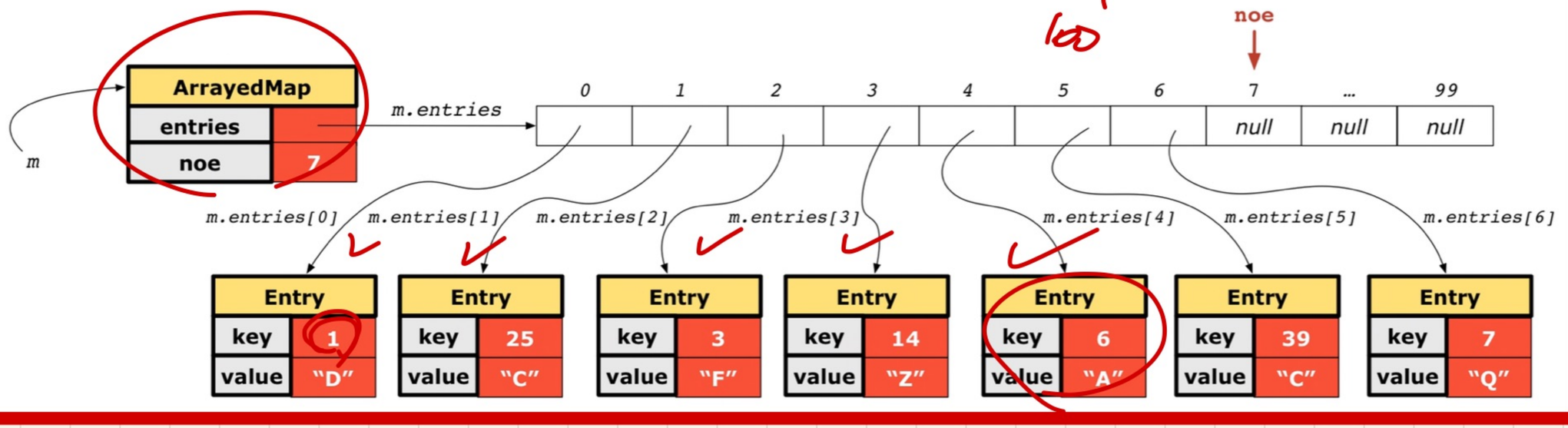
```
public class ArrayedMap {
  private final int MAX_CAPCAITY = 100;
  public String get (int key) {
    for(int i = 0; i < noe; i ++) {
      Entry e = entries[i];
      int k = e.getKey();
      if(k == key) { return e.getValue(); }
    }
    return null;
  }
```

iterations

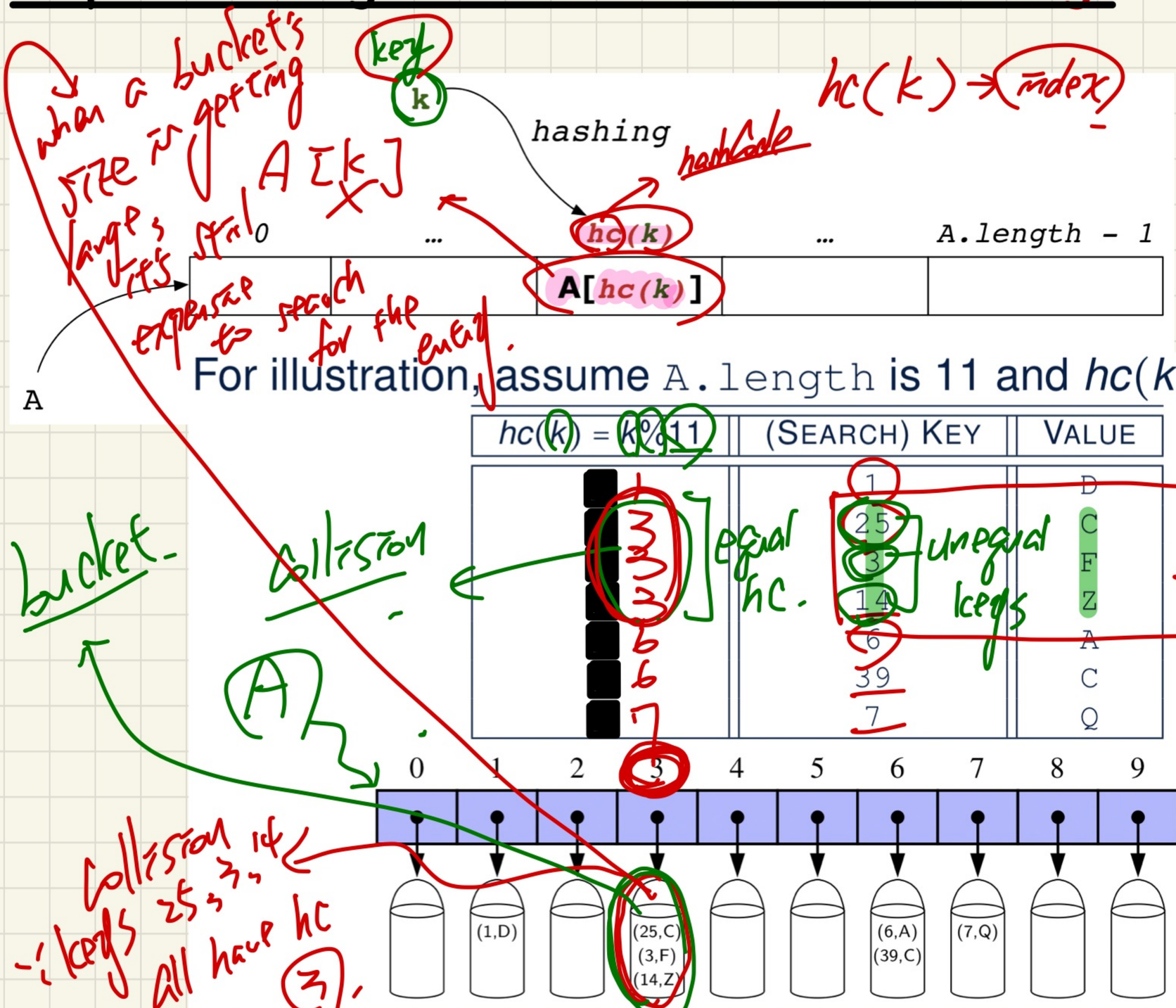
m.get(1) 1
m.get(6) 5

worst case: 7

100



Implementing a Hash Table via Hashing



- Converting k to hc(k)
- Indexing into A[hc(k)]

Testing Overridden/Redefined hashCode()

```java
@Test
public void testCustomizedHashFunction() {
  IntegerKey ik1 = new IntegerKey(1);
  /* 1 % 11 == 1 */
  assertTrue(ik1.hashCode() == 1);

  IntegerKey ik39_1 = new IntegerKey(39); /* 39 % 11 == 6 */
  IntegerKey ik39_2 = new IntegerKey(39);
  IntegerKey ik6 = new IntegerKey(6); /* 6 % 11 == 6 */

  assertTrue(ik39_1.hashCode() == 6);
  assertTrue(ik39_2.hashCode() == 6);
  assertTrue(ik6.hashCode() == 6);

  assertTrue(ik39_1.hashCode() == ik39_2.hashCode());
  assertTrue(ik39_1.equals(ik39_2));

  assertTrue(ik39_1.hashCode() == ik6.hashCode());
  assertFalse(ik39_1.equals(ik6));
}
```

① ik39_1.equals(ik6);

② ik39_1.hashCode() == ik6.hashCode

```java
1  public class IntegerKey {
2    private int k;
3    public IntegerKey(int k) { this.k = k; }
4    @Override
5    public int hashCode() { return k % 11; }
6    @Override
7    public boolean equals(Object obj) {
8      if(this == obj) { return true; }
9      if(obj == null) { return false; }
10     if(this.getClass() != obj.getClass()) { return false; }
11     IntegerKey other = (IntegerKey) obj;
12     return this.k == other.k;
13   } }
```